data structures and problem solving

Textbook for Dickinson College COMP232

Assembled primarily from materials provided by OpenDSA and distributed under OpenDSA License

Primary authors and copyright holders:
Ville Karavirta and Cliff Shaffer

Materials assembled by John MacCormick,
August 2021

About this book

This is an *offline, static* version of the textbook for the Dickinson College course COMP232, "Data Structures and Problem Solving." It is strongly recommended to use the *online, dynamic* version of the book. Instructions on how to view the online version are available on the course webpages. This offline version is provided as an additional convenience, especially for working without Internet access and to provide an easy way to search the entire textbook contents. This offline version has some disadvantages compared to the online version:

1. The offline version has many formatting problems, such as text and diagrams that do not fit on the page. No attempt has been made to correct these. Please consult the online version to see any content that is not visible due to formatting problems.
2. The offline version does not contain any of the dynamic examples available in the online version.

OpenDSA License

Contents

intentionally blank

intentionally blank

Chapter 0: Introduction

00.01 Data Structures and Algorithms

Due No Due Date **Points** 1 **Submitting** an external tool

00.01 Data Structures and Algorithms

0.1. Data Structures and Algorithms

0.1.1. Data Structures and Algorithms

0.1.1.1. Introduction

How many cities with more than 250,000 people lie within 500 miles of Dallas, Texas? How many people in my company make over $100,000 per year? Can we connect all of our telephone customers with less than 1,000 miles of cable? To answer questions like these, it is not enough to have the necessary information. We must organize that information in a way that allows us to find the answers in time to satisfy our needs.

Representing information is fundamental to computer science. The primary purpose of most computer programs is not to perform calculations, but to store and retrieve information—usually as fast as possible. For this reason, the study of data structures and the algorithms that manipulate them is at the heart of computer science. And that is what this book is about—helping you to understand how to structure information to support efficient processing.

Any course on Data Structures and Algorithms will try to teach you about three things:

1. It will present a collection of commonly used data structures and algorithms. These form a programmer's basic "toolkit". For many problems, some data structure or algorithm in the toolkit will provide a good solution. We focus on data structures and algorithms that have proven over time to be most useful.

2. It will introduce the idea of tradeoffs, and reinforce the concept that there are costs and benefits associated with every data structure or algorithm. This is done by describing, for each data structure, the amount of space and time required for typical operations. For each algorithm, we examine the time required for key input types.

3. It will teach you how to measure the effectiveness of a data structure or algorithm. Only through such measurement can you determine which data structure in your toolkit is most appropriate for a new problem. The techniques presented also allow you to judge the merits of new data structures that you or others might invent.

There are often many approaches to solving a problem. How do we choose between them? At the heart of computer program design are two (sometimes conflicting) goals:

1. To design an algorithm that is easy to understand, code, and debug.

2. To design an algorithm that makes efficient use of the computer's resources.

Ideally, the resulting program is true to both of these goals. We might say that such a program is “elegant.” While the algorithms and program code examples presented here attempt to be elegant in this sense, it is not the purpose of this book to explicitly treat issues related to goal (1). These are primarily concerns for the discipline of Software Engineering. Rather, we mostly focus on issues relating to goal (2).

How do we measure efficiency? Our method for evaluating the efficiency of an algorithm or computer program is called **asymptotic analysis**. Asymptotic analysis also gives a way to define the inherent difficulty of a problem. Throughout the book we use asymptotic analysis techniques to estimate the time cost for every algorithm presented. This allows you to see how each algorithm compares to other algorithms for solving the same problem in terms of its efficiency.

0.1.1.2. A Philosophy of Data Structures

You might think that with ever more powerful computers, program efficiency is becoming less important. After all, processor speed and memory size still continue to improve. Won’t today’s efficiency problem be solved by tomorrow’s hardware?

As we develop more powerful computers, our history so far has always been to use that additional computing power to tackle more complex problems, be it in the form of more sophisticated user interfaces, bigger problem sizes, or new problems previously deemed computationally infeasible. More complex problems demand more computation, making the need for efficient programs even greater. Unfortunately, as tasks become more complex, they become less like our everyday experience. So today’s computer scientists must be trained to have a thorough understanding of the principles behind efficient program design, because their ordinary life experiences often do not apply when designing computer programs.

In the most general sense, a **data structure** is any data representation and its associated operations. Even an integer or floating point number stored on the computer can be viewed as a simple data structure. More commonly, people use the term “data structure” to mean an organization or structuring for a collection of data items. A sorted list of integers stored in an array is an example of such a structuring. These ideas are explored further in a discussion of **Abstract Data Types**.

Given sufficient space to store a collection of **data items**, it is always possible to search for specified items within the collection, print or otherwise process the data items in any desired order, or modify the value of any particular data item. The most obvious example is an unsorted array containing all of the data items. It is possible to perform all necessary operations on an unsorted array. However, using the proper data structure can make the difference between a program running in a few seconds and one requiring many days. For example, searching for a given record in a **hash table** is much faster than searching for it in an unsorted array.

A solution is said to be **efficient** if it solves the problem within the required **resource constraints**. Examples of resource constraints include the total space available to store the data—possibly divided into separate main memory and disk space constraints—and the time allowed to perform each subtask. A solution is sometimes said to be efficient if it requires fewer resources than known alternatives, regardless of whether it meets any particular requirements. The **cost** of a solution is the amount of resources that the solution consumes. Most often, cost is measured in terms of one key resource such as time, with the implied assumption that the solution meets the other resource constraints.

0.1.1.3. Selecting a Data Structure

It should go without saying that people write programs to solve problems. However, sometimes programmers forget this. So it is crucial to keep this truism in mind when selecting a **data structure** to solve a particular **problem**. Only by first analyzing the problem to determine the performance goals that must be achieved can there be any hope of selecting the right data structure for the job. Poor program designers ignore this analysis step and apply a data structure that they are familiar with but which is inappropriate to the problem. The result is typically a slow program. Conversely, there is no sense in adopting a complex representation to “improve” a program that can meet its performance goals when implemented using a simpler design.

When selecting a data structure to solve a problem, you should follow these steps.

1. Analyze your problem to determine the **basic operations** that must be supported. Examples of basic operations include inserting a data item into the data structure, deleting a data item from the data structure, and finding a specified data item.
2. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

This three-step approach to selecting a data structure operationalizes a data-centered view of the design process. The first concern is for the data and the operations to be performed on them, the next concern is the representation for those data, and the final concern is the implementation of that representation.

Resource constraints on certain key operations, such as search, inserting data records, and deleting data records, normally drive the data structure selection process. Many issues relating to the relative importance of these operations are addressed by the following three questions, which you should ask yourself whenever you must choose a data structure.

1. Are all data items inserted into the data structure at the beginning, or are insertions interspersed with other operations? Static applications (where the data are loaded at the beginning and never change) typically get by with simpler data structures to get an efficient implementation, while dynamic applications often require something more complicated.
2. Can data items be deleted? If so, this will probably make the implementation more complicated.
3. Are all data items processed in some well-defined order, or is search for specific data items allowed? “Random access” search generally requires more complex data structures.

Each data structure has associated costs and benefits. In practice, it is hardly ever true that one data structure is better than another for use in all situations. If one data structure or algorithm is superior to another in all respects, the inferior one will usually have long been forgotten. For nearly every data structure and algorithm presented in this book, you will see examples of where it is the best choice. Some of the examples might surprise you.

A data structure requires a certain amount of space for each data item it stores, a certain amount of time to perform a single basic operation, and a certain amount of programming effort. Each problem has constraints on available space and time. Each solution to a problem makes use of the basic operations in some relative proportion, and the data structure selection process must account for this. Only after a careful analysis of your problem’s characteristics can you determine the best data structure for the task.

Example 0.1.1

A bank must support many types of transactions with its customers, but we will examine a simple model where customers wish to open accounts, close accounts, and add money or withdraw money from accounts. We can consider this problem at two distinct levels: (1) the requirements for the physical infrastructure and workflow process that the bank uses in its interactions with its customers, and (2) the requirements for the database system that manages the accounts.

The typical customer opens and closes accounts far less often than accessing the account. Customers are willing to spend many minutes during the process of opening or closing the account, but are typically not willing to wait more than a brief time for individual account transactions such as a deposit or withdrawal. These observations can be considered as informal specifications for the time constraints on the problem.

It is common practice for banks to provide two tiers of service. Human tellers or automated teller machines (ATMs) support customer access to account balances and updates such as deposits and withdrawals. Special service representatives are typically provided (during restricted hours) to handle opening and closing accounts. Teller and ATM transactions are expected to take little time. Opening or closing an account can take much longer (perhaps up to an hour from the customer’s perspective).

From a database perspective, we see that ATM transactions do not modify the database significantly. For simplicity, assume that if money is added or removed, this transaction simply changes the value stored in an account record. Adding a new account to the database is allowed to take several minutes. Deleting an account need have no time constraint, because from the customer’s point of view all that matters is that all the money be returned (equivalent to a withdrawal). From the bank’s point of view, the account record might be removed from the database system after business hours, or at the end of the monthly account cycle.

When considering the choice of data structure to use in the database system that manages customer accounts, we see that a data structure that has little concern for the cost of deletion, but is highly efficient for search and moderately efficient for insertion, should meet the resource constraints imposed by this problem. Records are accessible by unique account number (sometimes called an **exact-match query**). One data structure that meets these requirements is the **hash table**. Hash tables allow for extremely fast exact-match search. A record can be modified quickly when the modification does not affect its space requirements. Hash tables also support efficient insertion of new records. While deletions can also be supported efficiently, too many deletions lead to some degradation in performance for the remaining operations. However, the hash table can be reorganized periodically to restore the system to peak efficiency. Such reorganization can occur offline so as not to affect ATM transactions.

Example 0.1.2

A company is developing a database system containing information about cities and towns in the United States. There are many thousands of cities and towns, and the database program should allow users to find information about a particular place by name (another example of an exact-match query). Users should also be able to find all places that match a particular value or range of values for attributes such as location or population size. This is known as a **range query**.

A reasonable database system must answer queries quickly enough to satisfy the patience of a typical user. For an exact-match query, a few seconds is satisfactory. If the database is meant to support range queries that can return many cities that match the query specification, the user might tolerate the entire operation to take longer, perhaps on the order of a minute. To meet this requirement, it will be necessary to support operations that process range queries efficiently by processing all cities in the range as a batch, rather than as a series of operations on individual cities.

The hash table suggested in the previous example is inappropriate for implementing our city database, because it cannot perform efficient range queries. The **B^+-tree** supports large databases, insertion and deletion of data records, and range queries. However, a simple **linear index** would be more appropriate if the database is created once, and then never changed, such as an atlas distributed on a CD or accessed from a website.

0.1.1.4. Introduction Summary Questions

Practicing Introduction: Summary Questions

Current score: 0 out of 5

As computers have become more powerful:

○ We are better able to use our everyday intuition to solve problems

○ We have used that additional computing power to tackle more complex problems

○ The need for good algorithms has become less because processor speed can make up for a slow algorithm

○ The algorithms have become easier to understand

Answer

Check Answer

Need help?

I'd like a hint

Chapter 1: Recursion

1.1. Introduction

1.1.1. Introduction

An **algorithm** (or a function in a computer program) is **recursive** if it invokes itself to do part of its work. Recursion makes it possible to solve complex problems using programs that are concise, easily understood, and algorithmically efficient. Recursion is the process of solving a large problem by reducing it to one or more sub-problems which are identical in structure to the original problem and somewhat simpler to solve. Once the original subdivision has been made, the sub-problems divided into new ones which are even less complex. Eventually, the sub-problems become so simple that they can be then solved without further subdivision. Ultimately, the complete solution is obtained by reassembling the solved components.

For a recursive approach to be successful, the recursive “call to itself” must be on a smaller problem than the one originally attempted. In general, a recursive algorithm must have two parts:

1. The **base case**, which handles a simple input that can be solved without resorting to a recursive call, and
2. The recursive part which contains one or more recursive calls to the algorithm. In every recursive call, the parameters must be in some sense “closer” to the base case than those of the original call.

Recursion has no counterpart in everyday, physical-world problem solving. The concept can be difficult to grasp because it requires you to think about problems in a new way. When first learning recursion, it is common for people to think a lot about the recursive process. We will spend some time in these modules going over the details for how recursion works. But when writing recursive functions, it is best to stop thinking about how the recursion works beyond the recursive call. You should adopt the attitude that the sub-problems will take care of themselves, that when you call the function recursively it will return the right answer. You just worry about the base cases and how to recombine the sub-problems.

Newcomers who are unfamiliar with recursion often find it hard to accept that it is used primarily as a tool for simplifying the design and description of algorithms. A recursive algorithm does not always yield the most efficient computer program for solving the problem because recursion involves function calls, which are typically more expensive than other alternatives such as a while loop. However, the recursive approach usually provides an algorithm that is reasonably efficient. If necessary, the clear, recursive solution can later be modified to yield a faster implementation.

Imagine that someone in a movie theater asks you what row you’re sitting in. You don’t want to count, so you ask the person in front of you what row they are sitting in, knowing that they will tell you a number one less than your row number. The person in front could ask the person in front of them. This will keep happening until word reaches the front row and it is easy to respond: “I’m in row 1!” From there, the correct message (incremented by one each row) will eventually make it’s way back to the person who asked.

Imagine that you have a big task. You could just do a small piece of it, and then **delegate** the rest to some helper, as in this example.

1 / 7 << < > >>

You want to multiply two numbers x and y.

```
int multiply(int x, int y) {
  if (x == 1)
    return y;
  else
    return multiply(x - 1, y) + y;
}
```

x*y?

Let's look deeper into the details of what your friend does when you delegate the work. (Note that we show you this process once now, and once again when we look at some recursive functions. But when you are writing your own recursive functions, you shouldn't worry about all of these details.)

1 / 11 << < > >>

You want to multiply two numbers x and y.

```
int multiply(int x, int y) {
  if (x == 1)
    return y;
  else
    return multiply(x - 1, y) + y;
}
```

x*y?

In order to understand recursion, you need to be able to do two things. First, you have to understand how to read a

1.2. Writing a recursive function

1.2.1. Writing a recursive function

Solving a “big” problem recursively means to solve one or more smaller versions of the problem, and using those solutions of the smaller problems to solve the “big” problem. In particular, solving problems recursively means that smaller versions of the problem are solved in a similar way. For example, consider the problem of summing values of an array. What’s the difference between summing the first 50 elements in an array versus summing the first 100 elements? You would use the same technique. You can even use the solution to the smaller problem to help you solve the larger problem.

Here are the basic four steps that you need to write any recursive function.

1 / 17 << < > >>

Step 1: Write and define the prototype for the function.

Now le’t see some different ways that we could write `Sum` recursively.

1 / 6 << < > >>

Here are a few variations on how to solve the sum problem recursively.

Example 1.2.1

Our example for summing the first n numbers of an array could have been written just as easily using a loop. Here is an example of a function that is more naturally written using recursion.

The following code computes the Fibonacci sequence for a given number. The Fibonacci Sequence is the series of numbers: 1, 1, 2, 3, 5, 8, 13, 21, 34, … Any number in the sequence is found by adding up the two numbers before it. The base cases are that `Fibonacci(0) = 1` and `Fibonacci(1) = 1`.

Java | Java (Generic) | Toggle Tree View

```
long Fibonacci(int n) {
  if (n < 2) {
    return 1;
  }
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

01.03 Code Completion Practice Exercises

Due No Due Date **Points** 16 **Submitting** an external tool

01.03 Code Completion Practice Exercises

1.3. Code Completion Practice Exercises

1.3.1. Introduction

The most important step to learning recursion is doing a lot of practice. The rest of this tutorial will take you through the process with a series of practice exercises that will lead you to master recursion.

1.3.2. Recursion Programming Exercise: Largest

X263: Recursion Programming Exercise: Largest

Write the missing base case for function `largest`. Function `largest` should find the largest number in ar
called, `index` will equal `numbers.length-1`.

Examples:

```
largest({2, 4, 8}, 2) -> 8
```

Your Answer:

```
public int largest(int[] numbers, int index) {
  if <<Missing base case>>
    return numbers[0];
  return Math.max(numbers[index], largest(numbers, index-1));
}
```

Check my answer! Reset

Feedback

Your feedback will a
answer.

1.3.3. Recursion Programming Exercise: Multiply

X264: Recursion Programming Exercise: Multipl

For function `multiply`, write the missing base case condition and action. This function will multiply two n

that both `x` and `y` are positive.

Examples:

```
multiply(2, 3) -> 6
```

Your Answer:

```
public int multiply(int x, int y) {
  if <<Missing base case condition>> {
    <<Missing base case action>>
  } else {
    return multiply(x - 1, y) + y;
  }
}
```

Check my answer! Reset

Feedback

Your feedback will a

answer.

1.3.4. Recursion Programming Exercise: GCD

X265: Recursion Programming Exercise: GCD

The greatest common divisor (GCD) for a pair of numbers is the largest positive integer that divides both nu
function `GCD`, write the missing base case condition and action. This function will compute the greatest co
assume that `x` and `y` are both positive integers and that `x > y`. Greatest common divisor is computed as
`GCD(x, 0) = x` and `GCD(x, y) = GCD(y, x % y)`.

Examples:

```
GCD(6, 4) -> 2
```

Your Answer:

```
public int GCD(int x, int y) {
  if <<Missing base case condition>> {
    <<Missing base case action>>
  } else {
    return GCD(y, x % y);
  }
}

```

Feedback

Your feedback will a
answer.

Check my answer! Reset

1.3.5. Recursion Programming Exercise: log

X266: Recursion Programming Exercise: log

For function `log`, write the missing base case condition and the recursive call. This function computes the
example: log 8 to the base 2 equals 3 since 8 = 2*2*2. We can find this by dividing 8 by 2 until we reach 1,
divisions we make. You should assume that `n` is exactly `b` to some integer power.

Examples:

```
log(2, 4) -> 2

log(10, 100) -> 2
```

Your Answer:

```
public int log(int b, int n ) {
  if <<Missing base case condition>> {
    return 0;
  } else {
    return <<Missing a Recursive case action>>
  }
}

```

Feedback

Your feedback will a
answer.

Check my answer! Reset

Check my answer! Reset

1.3.6. Recursion Programming Exercise: Cummulative Sum

X267: Recursion Programming Exercise: Cumula

For function `sumtok`, write the missing recursive call. This function returns the sum of the values from 1 tc

Examples:

```
sumtok(5) -> 15
```

Your Answer:

```
public int sumtok(int k) {
  if (k <= 0) {
    return 0;
  } else {
    return <<Missing Recursive case action>>
  }
}

```

Feedback

Your feedback will a
answer.

Check my answer! Reset

1.3.7. Recursion Programming Exercise: Add odd values

X268: Recursion Programming Exercise: Add od

For function `addOdd(n)` write the missing recursive call. This function should return the sum of all postive

`n`.

Examples:

```
addOdd(1) -> 1

addOdd(2) -> 1

addOdd(3) -> 4

addOdd(7) -> 16
```

Your Answer:

```
public int addOdd(int n) {
  if (n <= 0) {
    return 0;
  }
  if (n % 2 != 0) { // Odd value
    return <<Missing a Recursive call>>
  } else { // Even value
    return addOdd(n - 1);
  }
}

```

Feedback

Your feedback will a

answer.

Check my answer! Reset

1.3.8. Recursion Programming Exercise: Sum Of the Digits

X269: Recursion Programming Exercise: Sum of

For function `sumOfDigits` , write the missing recursive call. This function takes a non-negative integer and

Examples:

```
sumOfDigits(1234) -> 10
```

Your Answer:

```
public int sumOfDigits(int number) {
  if (number < 10)
    return number;
  return <<Missing a Recursive case action>>
}

```

Feedback

Your feedback will a
answer.

Check my answer! Reset

1.3.9. Recursion Programming Exercise: Count Characters

X270: Recursion Programming Exercise: Count C

For function `countChr()` write the missing part of the recursive call. This function should return the numb appears in string "str".
Recall that `str.substring(a)` will return the substring of `str` from position `a` to the end of `str`, while the substring of `str` starting at position `a` and continuing to (but not including) the character at position

Examples:

```
countChr("ctcowcAt") -> 1
```

Your Answer:

```
public int countChr(String str) {
  if (str.length() == 0) {
    return 0;
  }
  int count = 0;
  if (str.substring(0, 1).equals("A")) {
    count = 1;
  }
  return count + <<Missing a Recursive call>>
}

```

Check my answer! Reset

Feedback

Your feedback will a answer.

1.4. Writing More Sophisticated Recursive Functions

Some recursive functions have only one base case and one recursive call. But it is common for there to be more than one of either or both.

The following is the general structure for a recursive function.

Java | Java (Generic) | Toggle Tree Vie

```
if ( base case 1 )
  // return some simple expression
else if ( base case 2 )
  // return some simple expression
else if ( base case 3 )
  // return some simple expression
else if ( recursive case 1 ) {
  // some work before
  // recursive call
  // some work after
 }
else if ( recursive case 2 ) {
  // some work before
  // recursive call
  // some work after
 }
else { // recursive case 3
  // some work before
  // recursive call
  // some work after
 }
```

Example 1.4.1

Consider a rather simple function to determine if an integer `X` is prime or not. `Y` is a helper variable that is used as the devisor. When calling the function initially, `Y = X - 1`

Java | Java (Generic) | Toggle Tree View

```
boolean prime(int x, int y) {
  if (y == 1)
    return true;
  else if (x % y == 0)
```

```
    return false;
  else
    return prime(x, y-1);
}
```

We see that `Prime` has two base cases and one recursive call.

Example 1.4.2

Here is a function that has multiple recursive calls. Given an `int` array named `set`, function `isSubsetSum` determines whether some of the values in `set` add up to `sum`. For example, given the number 3, 8, 1, 7, and -3, with `sum = 4`, the result is `true` because the values 3 and 1 sum to 4. If `sum = 6`, then the result will be `true` because the $8+1+-3=6$. On the other hand, if `sum = 2` then the result is `false` there is no combination of the five numbers that adds up to 2. In this code, variable `n` is the number of values that we look at. We don't want to just use `set.length` because the recursive calls need to limit their work to part of the array.

Java | Java (Generic) | Toggle Tree View

```
boolean isSubsetSum(int set[], int n, int sum) {
  if (sum == 0)
    return true;
  if ((n == 0) && (sum != 0))
    return false;
  if (set[n - 1] > sum)
    return isSubsetSum(set, n - 1, sum);
  return isSubsetSum(set, n - 1, sum) || isSubsetSum(set, n - 1, sum - set[n - 1]);
}
```

This example has two base cases and two recursive calls.

Example 1.4.3

Here is a function that has multiple base cases and multiple recursive calls. Function `paths` counts the number of different ways to reach a given basketball score. Recall that in Basketball, it is possible to get points in increments of 1, 2, or 3. So if `n = 3`, then `paths` will return 4, since there are four different ways to accumulate 3 points: $1+1+1, 1+2, 2+1$, and 3.

Java | Java (Generic) | Toggle Tree View

```
int paths(int n) {
  if (n == 1)
```

```
    return 1;
  if (n == 2)
    return 2;
  if (n == 3)
    return 4;
  return paths(n - 1) + paths(n - 2) + paths(n - 3);
}
```

This function has three base cases and three recursive calls.

01.05 Harder Code Completion Practice Exercises

Due No Due Date **Points** 6 **Submitting** an external tool

01.05 Harder Code Completion Practice Exercises

1.5. Harder Code Completion Practice Exercises

1.5.1. Recursion Programming Exercise: Minimum of array

X271: Recursion Programming Exercises: Minimu

For function `recursiveMin`, write the missing part of the recursive call. This function should return the mi integers. You should assume that `recursiveMin` is initially called with `startIndex` = 0.

Examples:

```
recursiveMin({2, 4, 8}, 0) -> 2
```

Your Answer:

```
public int recursiveMin(int numbers[], int startIndex) {
  if (startIndex == numbers.length - 1) {
    return numbers[startIndex];
  } else {
    return Math.min(numbers[startIndex], recursiveMin(numbers,
startIndex + 1));
  }
}

```

Check my answer! Reset

Feedback

1.0 / 1.0

Result	Behavic
✓	recursiv
✓	recursiv
✓	recursiv
✓	recursiv
✓	hidden

1.5.2. Recursion Programming Exercise: Is Reverse

X272: Recursion Programming Exercise: Is Reve

For function `isReverse`, write the two missing base case conditions. Given two strings, this function retur identical, but are in reverse order. Otherwise it returns false. For example, if the inputs are "tac" and "cat", t

Examples:

```
isReverse("tac", "cat") -> true
```

Your Answer:

```
public boolean isReverse(String s1, String s2) {
  if <<Missing condition 1>>
    return true;
  else if <<Missing condition 2>>
    return false;
  else {
    String s1first = s1.substring(0, 1);
    String s2last = s2.substring(s2.length() - 1);
    return s1first.equals(s2last) &&
           isReverse(s1.substring(1), s2.substring(0, s2.length() -
1));
  }
}
```

Check my answer! Reset

Feedback

Your feedback will a answer.

1.5.3. Recursion Programming Exercise: Decimal to Binary

X273: Recursion Programming Exercise: Decima

For function `decToBinary`, write the missing parts of the recursion case. This function should return a stri for int variable `num`. Example: The binary equivalent of 13 may be found by repeatedly dividing 13 by 2. Sc string "1101".

Examples:

```
decToBinary(13) -> "1101"
```

Your Answer:

```
public String decToBinary (int num) {
  if (num < 2)
    return Integer.toString(num);
  else
    return <<Missing recursive call>> + <<Missing calculation>>;
}
```

Check my answer! Reset

Feedback

Your feedback will a answer.

01.06 Writing Practice Exercises

Due No Due Date **Points** 8 **Submitting** an external tool

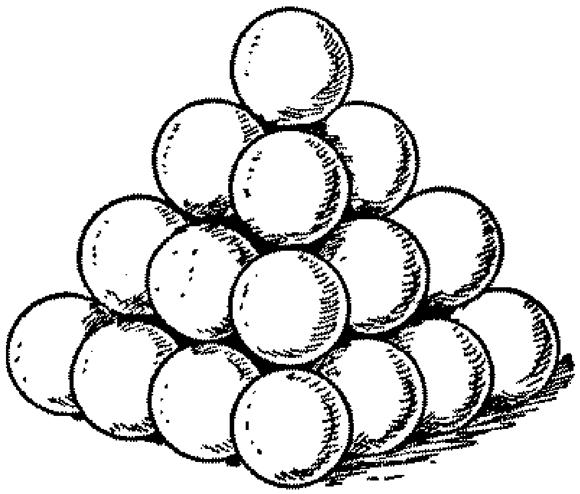
01.06 Writing Practice Exercises

1.6. Writing Practice Exercises

1.6.1. Recursion Programming Exercise: Cannonballs

X274: Recursion Programming Exercise: Cannon

Spherical objects, such as cannonballs, can be stacked to form a pyramid with one cannonball at the top, si
of four cannonballs, sitting on top of a square composed of nine cannonballs, and so forth.



Given the following recursive function signature, write a recursive function that takes as its argument the h
and returns the number of cannonballs it contains.

Examples:

```
cannonball(2) -> 5
```

Your Answer:

```
public int cannonball(int height) {


}
```

Feedback

Your feedback will a
answer.

```
5
```

Check my answer! Reset

1.6.2. Recursion Programming Exercise: Check Palindrome

X275: Recursion Programming Exercise: Check I

Write a recursive function named `checkPalindrome` that takes a string as input, and returns true if the stri
not a palindrome. A string is a palindrome if it reads the same forwards or backwards.
Recall that `str.charAt(a)` will return the character at position `a` in `str`. `str.substring(a)` will return
`a` to the end of `str`, while `str.substring(a, b)` will return the substring of `str` starting at position `a`
including) the character at position `b`.

Examples:

```
checkPalindrome("madam") -> true
```

Your Answer:

```
public boolean checkPalindrome(String s) {





}

```

Check my answer! Reset

Feedback

Your feedback will a
answer.

1.6.3. Recursion Programming Exercise: Subset Sum

X276: Recursion Programming Exercise: Subset

Write a recursive function that takes a start index, array of integers, and a target sum. Your goal is to find w
integers adds up to the target sum. The start index is initially 0.
A target sum of 0 is true for any array.

Examples:

```
subsetSum(0, {2, 4, 8}, 10) -> true
```

Your Answer:

```
public boolean subsetSum(int start, int[] nums, int target) {

}
```

Feedback

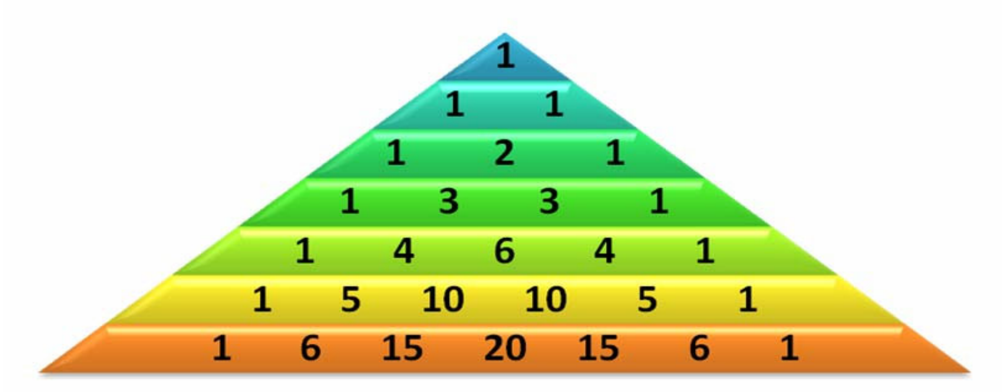
Your feedback will a
answer.

Check my answer! Reset

1.6.4. Recursion Programming Exercise: Pascal Triangle

X277: Recursion Programming Exercise: Pascal T

Pascal's triangle is a useful recursive definition that tells us the coefficients in the expansion of the polynoı the triangle has a coordinate, given by the row it is on and its position in the row (which you could call a cc triangle is defined as the sum of the item above it and the item above it and to the left. If there is a positio treat it as if we had a 0 there.



Given the following recursive function signature, write the recursive function that takes a row and a colum position in the triangle. Assume that the triangle starts at row 0 and column 0.

Examples:

```
pascal(2, 1) -> 2

pascal(1, 2) -> 0
```

Your Answer:

```
public int pascal(int row, int column) {



```

Feedback

Your feedback will app answer.

```
}
```

Check my answer! Reset

1.7. Tracing Recursive Code

1.7.1. Tracing Recursive Code

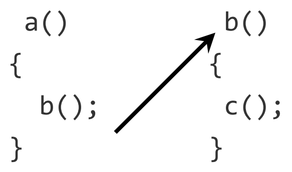
When writing a recursive function, you should think in a top-down manner. Do not worry about how the recursive call solves the sub-problem. Simply accept that it will solve it correctly. Use this result as though you had called some library function, to correctly solve the original problem.

When you have to read or trace a recursive function, then you do need to consider how the function is doing its work. Tracing a few recursive functions is a great way to learn how recursion behaves. But after you become comfortable with tracing, you will rarely need to work through so many details again. You will begin to develop confidence about how recursion works.

You know that information can be passed in (using a function parameter) from one recursive call to another, on ever smaller problems, until a base case is reached in the winding phase. Then, a return value is passed back as the series of recursive calls unwinds. Sometimes people forget about the “unwinding” phase.

1 / 16 &lt;&lt; &lt; &gt; &gt;&gt;

Suppose function a() has a call to function b().



```
 a()            b()
{              {
   b();          c();
}              }
```

During the winding phase, any parameter passed through the recursive call flows forward until the base case is reached. During the unwinding phase, the return value of the function (if there is one) flows backwards to the calling copy of the function. In the following example, a recursive function to compute factorial has information flowing forward during the winding phase, and backward during the unwinding phase.

1 / 11 &lt;&lt; &lt; &gt; &gt;&gt;

Suppose that we want to compute the value of factorial(5) using the following recursive factorial implementatio

```
int factorial(int n) {
```

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  return n * factorial(n-1);
}
```

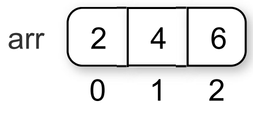
The recursive function may have information flow for more than one parameter. For example, a recursive function that sums the values in an array recursively may pass the array itself and the index through the recursive call in the winding phase and returns back the summed value so far in the unwinding phase.

1 / 15

<< < > >>

Now consider a simple recursive function to sum the elements of an array. The information flow passes the arr index forward during the winding phase. The sum of the values is passed backward during the unwinding phas

```
int sum(int arr[], int n) {
  if (n == 0) {
    return 0;
  }
  return sum(arr, n - 1) + arr[n - 1];
}
```

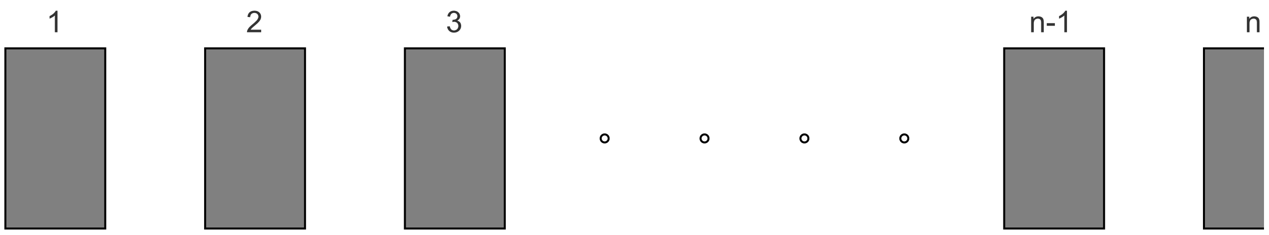


1.7.1.1. A Domino Analogy

1 / 9

<< < > >>

Let's think about the domino effect in terms of recursion. When you stand up a bunch of dominos properly, p
first one causes all of the others to fall over one by one until the last domino is reached.



```
Domino(n) {
  If(n == 1)
    TipOver(1) //manually tip the domino over.
  else{
    Domino(n-1) //to tip the first (n-1) dominos over
    TipOver(n) //the nth domino will be tipped over subsequently
  }
}
```

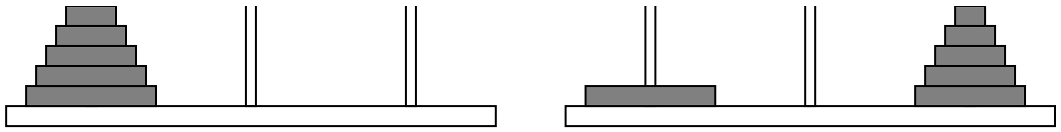
This recursive model for the domino effect can be used as a template for the solution to all linear recursive functions. Think of tipping over each domino as performing a further step of computation toward the final solution. Remember these rules:

1. Since the first domino has to be tipped over manually, the solution for the base case is computed non-recursively.

2. Before any given domino can be tipped over, all preceding dominos have to be tipped over.

1.7.1.2. Towers of Hanoi

Here is another example of recursion, based on a famous puzzle called “Towers of Hanoi”. The natural algorithm to solve this problem has multiple recursive calls. It cannot be rewritten easily using loops. “Towers of Hanoi” comes from an ancient Vietnamese legend. A group of monks is tasked with moving a tower of 64 disks of different sizes according to certain rules. The legend says that, when the monks will have finished moving all of the disks, the world will end.



The Towers of Hanoi puzzle begins with three poles and n rings, where all rings start on the leftmost pole (labeled Pole A). The rings each have a different size, and are stacked in order of decreasing size with the largest ring at the bottom, as shown in part (a) of the figure. The problem is to move the rings from the leftmost pole to the middle pole (labeled Pole B) in a series of steps. At each step the top ring on some pole is moved to another pole. What makes this puzzle interesting is the limitation on where rings may be moved: A ring may never be moved on top of a smaller ring.

How can you solve this problem? It is easy if you don't think too hard about the details. Instead, consider that all rings are to be moved from Pole A to Pole B. It is not possible to do this without first moving the bottom (largest) ring to Pole B. To do that, Pole B must be empty, and only the bottom ring can be on Pole A. The remaining $n-1$ rings must be stacked up in order on Pole C, as shown in part (b) of the figure. How can you do this? Assume that a function X is available to solve the problem of moving the top $n-1$ rings from Pole A to Pole C. Then move the bottom ring from Pole A to Pole B. Finally, again use function X to move the remaining $n-1$ rings from Pole C to Pole B. In both cases, "function X" is simply the Towers of Hanoi function called on a smaller version of the problem.

The secret to success is relying on the Towers of Hanoi algorithm to do the work for you. You need not be concerned about the gory details of *how* the Towers of Hanoi subproblem will be solved. That will take care of itself provided that two things are done. First, there must be a base case (what to do if there is only one ring) so that the recursive process will not go on forever. Second, the recursive call to Towers of Hanoi can only be used to solve a smaller problem, and then only one of the proper form (one that meets the original definition for the Towers of Hanoi problem, assuming appropriate renaming of the poles).

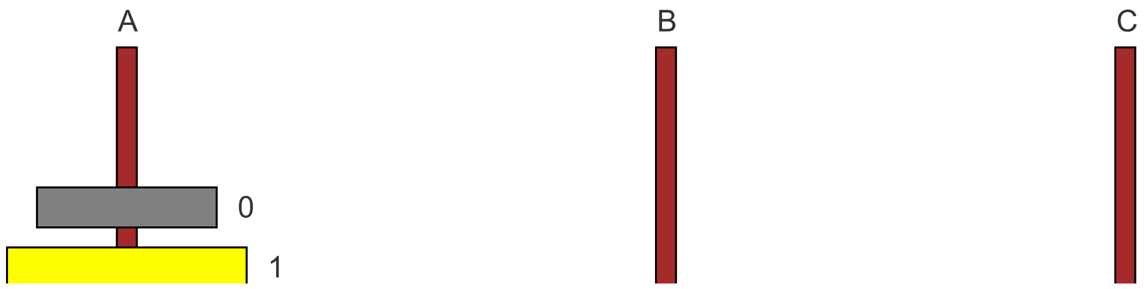
Here is an implementation for the recursive Towers of Hanoi algorithm. Function `move(start, goal)` takes the top ring from Pole `start` and moves it to Pole `goal`. If `move` were to print the values of its parameters, then the result of calling `TOHr` would be a list of ring-moving instructions that solves the problem.

Toggle Tree View

```
// Compute the moves to solve a Tower of Hanoi puzzle.
// Function move does (or prints) the actual move of a disk
// from one pole to another.
// n: The number of disks
// start: The start pole
// goal: The goal pole
// temp: The other pole
static void TOHr(int n, Pole start, Pole goal, Pole temp) {
  if (n == 0) { return; }          // Base case
  TOHr(n-1, start, temp, goal); // Recursive call: n-1 rings
  move(start, goal);            // Move bottom disk to goal
  TOHr(n-1, temp, goal, start); // Recursive call: n-1 rings
}
```

This next slideshow explains the solution to the Towers of Hanoi problem.

1 / 16



01.08 Tracing Practice Exercises

Due No Due Date **Points** 6 **Submitting** an external tool

01.08 Tracing Practice Exercises

1.8. Tracing Practice Exercises

1.8.1. Forward Flow Tracing Exercises

Practicing Recursion Tracing: Forward Flow

Current score: 0 out of 5

Consider the following function:

What is the return of calling mystery(2,0)?

(Either write a number, or write "infinite recursion".)

Answer

Check Answer

Need help?

I'd like a hint

1.8.2. Backward Flow Tracing Exercises

1.8.3. Find Error Tracing Exercises

1.8.4. Two Recursive Calls Tracing Exercises

1.8.5. How Many Times Tracing Exercises

1.8.6. Harder Tracing Exercises

01.09 Summary Exercises

Due No Due Date **Points** 1 **Submitting** an external tool

01.09 Summary Exercises

1.9. Summary Exercises

1.9.1. Summary Questions

Practicing Recursion: Summary Questions

Current score: 0 out of 5

Answer TRUE or FALSE.

A recursive function is invoked differently from a non-recursive method.

◯ True

◯ False

Answer

Check Answer

Need help?

I'd like a hint

Chapter 2: Algorithm Analysis

2.1. Chapter Introduction

How long will it take to process the company payroll once we complete our planned merger? Should I buy a new payroll program from vendor X or vendor Y? If a particular program is slow, is it badly implemented or is it solving a hard problem? Questions like these ask us to consider the difficulty of a problem, or the relative efficiency of two or more approaches to solving a problem.

This chapter introduces the motivation, basic notation, and fundamental techniques of algorithm analysis. We focus on a methodology known as **asymptotic algorithm analysis**, or simply **asymptotic analysis**. Asymptotic analysis attempts to estimate the resource consumption of an algorithm. It allows us to compare the relative costs of two or more algorithms for solving the same problem. Asymptotic analysis also gives algorithm designers a tool for estimating whether a proposed solution is likely to meet the resource constraints for a problem before they implement an actual program. After reading this chapter, you should understand

the concept of a **growth rate**, the rate at which the cost of an algorithm grows as the size of its input grows;

the concept of an **upper bound** and **lower bound** for a growth rate, and how to estimate these bounds for a simple program, algorithm, or problem; and

the difference between the cost of an **algorithm** (or program) and the cost of a **problem**.

The chapter concludes with a brief discussion of the practical difficulties encountered when empirically measuring the cost of a program, and some principles for code tuning to improve program efficiency.

02.02 Problems, Algorithms, and Programs

Due No Due Date **Points** 1 **Submitting** an external tool

02.02 Problems, Algorithms, and Programs

2.2. Problems, Algorithms, and Programs

2.2.1. Problems, Algorithms, and Programs

2.2.1.1. Problems

Programmers commonly deal with problems, algorithms, and computer programs. These are three distinct concepts.

As your intuition would suggest, a **problem** is a task to be performed. It is best thought of in terms of inputs and matching outputs. A problem definition should not include any constraints on *how* the problem is to be solved. The solution method should be developed only after the problem is precisely defined and thoroughly understood. However, a problem definition should include constraints on the resources that may be consumed by any acceptable solution. For any problem to be solved by a computer, there are always such constraints, whether stated or implied. For example, any computer program may use only the main memory and disk space available, and it must run in a “reasonable” amount of time.

Problems can be viewed as functions in the mathematical sense. A **function** is a matching between inputs (the **domain**) and outputs (the **range**). An input to a function might be a single value or a collection of information. The values making up an input are called the **parameters** of the function. A specific selection of values for the parameters is called an **instance** of the problem. For example, the input parameter to a sorting function might be an array of integers. A particular array of integers, with a given size and specific values for each position in the array, would be an instance of the sorting problem. Different instances might generate the same output. However, any problem instance must always result in the same output every time the function is computed using that particular input.

This concept of all problems behaving like mathematical functions might not match your intuition for the behavior of computer programs. You might know of programs to which you can give the same input value on two separate occasions, and two different outputs will result. For example, if you type `date` to a typical Linux command line prompt, you will get the current date. Naturally the date will be different on different days, even though the same command is given. However, there is obviously more to the input for the date program than the command that you type to run the program. The date program computes a function. In other words, on any particular day there can only be a single answer returned by a properly running date program on a completely specified input. For all computer programs, the output is completely determined by the program’s full set of inputs. Even a “random number generator” is completely determined by its inputs (although some random number generating systems appear to get generator" is completely determined by its inputs (although some random number generating systems appear to get around this by accepting a random input from a physical process beyond the user's control). The limits to what functions can be implemented by programs is part of the domain of **Computability**.

2.2.1.2. Algorithms

An **algorithm** is a method or a process followed to solve a problem. If the problem is viewed as a function, then an algorithm is an implementation for the function that transforms an input to the corresponding output. A problem can be solved by many different algorithms. A given algorithm solves only one problem (i.e., computes a particular function). OpenDSA modules cover many problems, and for several of these problems we will see more than one algorithm. For the important problem of sorting there are over a dozen commonly known algorithms!

The advantage of knowing several solutions to a problem is that solution $\mathbf{A}$ might be more efficient than solution $\mathbf{B}$ for a specific variation of the problem, or for a specific class of inputs to the problem, while solution $\mathbf{B}$ might be more efficient than $\mathbf{A}$ for another variation or class of inputs. For example, one sorting algorithm might be the best for sorting a small collection of integers (which is important if you need to do this many times). Another might be the best for sorting a large collection of integers. A third might be the best for sorting a collection of variable-length strings.

By definition, something can only be called an algorithm if it has all of the following properties.

1. It must be *correct*. In other words, it must compute the desired function, converting each input to the correct output. Note that every algorithm implements some function, because every algorithm maps every input to some output (even if that output is a program crash). At issue here is whether a given algorithm implements the *intended* function.

2. It is composed of a series of *concrete steps*. Concrete means that the action described by that step is completely understood — and doable — by the person or machine that must perform the algorithm. Each step must also be doable in a finite amount of time. Thus, the algorithm gives us a "recipe" for solving the problem by performing a series of steps, where each such step is within our capacity to perform. The ability to perform a step can depend on who or what is intended to execute the recipe. For example, the steps of a cookie recipe in a cookbook might be considered sufficiently concrete for instructing a human cook, but not for programming an automated cookie-making factory.

3. There can be *no ambiguity* as to which step will be performed next. Often it is the next step of the algorithm description. Selection (e.g., the `if` statement) is normally a part of any language for describing algorithms. Selection allows a choice for which step will be performed next, but the selection process is unambiguous at the time when the choice is made.

4. It must be composed of a *finite* number of steps. If the description for the algorithm were made up of an infinite number of steps, we could never hope to write it down, nor implement it as a computer program. Most languages for describing algorithms (including English and "pseudocode") provide some way to perform repeated actions, known as iteration. Examples of iteration in programming languages include the `while` and `for` loop constructs. Iteration allows for short descriptions, with the number of steps actually performed controlled by the input.

5. It must *terminate*. In other words, it may not go into an infinite loop.

2.2.1.3. Programs

We often think of a computer **program** as an instance, or concrete representation, of an algorithm in some programming language. Algorithms are usually presented in terms of programs, or parts of programs. Naturally, there are many programs that are instances of the same algorithm, because any modern computer programming language can be used to implement the same collection of algorithms (although some programming languages can make life easier for the programmer). To simplify presentation, people often use the terms “algorithm” and “program” interchangeably, despite the fact that they are really separate concepts. By definition, an algorithm must provide sufficient detail that it can be converted into a program when needed.

The requirement that an algorithm must terminate means that not all computer programs meet the technical definition of an algorithm. Your operating system is one such program. However, you can think of the various tasks for an operating system (each with associated inputs and outputs) as individual problems, each solved by specific algorithms implemented by a part of the operating system program, and each one of which terminates once its output is produced.

2.2.1.4. Summary

To summarize: A **problem** is a function or a mapping of inputs to outputs. An **algorithm** is a recipe for solving a problem whose steps are concrete and unambiguous. Algorithms must be correct, of finite length, and must terminate for all inputs. A **program** is an instantiation of an algorithm in a programming language. The following slideshow should help you to visualize the differences.

1 / 16 << < > >>

Here is a visual summary showing how to differentiate between a problem, a problem instance, an algorit
program.

2.2.1.5. Summary Questions

02.03 Comparing Algorithms

Due No Due Date **Points** 2 **Submitting** an external tool

02.03 Comparing Algorithms

2.3. Comparing Algorithms

2.3.1. Comparing Algorithms

2.3.1.1. Introduction

How do you compare two algorithms for solving some problem in terms of efficiency? We could implement both algorithms as computer programs and then run them on a suitable range of inputs, measuring how much of the resources in question each program uses. This approach is often unsatisfactory for four reasons. First, there is the effort involved in programming and testing two algorithms when at best you want to keep only one. Second, when empirically comparing two algorithms there is always the chance that one of the programs was “better written” than the other, and therefore the relative qualities of the underlying algorithms are not truly represented by their implementations. This can easily occur when the programmer has a bias regarding the algorithms. Third, the choice of empirical test cases might unfairly favor one algorithm. Fourth, you could find that even the better of the two algorithms does not fall within your resource budget. In that case you must begin the entire process again with yet another program implementing a new algorithm. But, how would you know if any algorithm can meet the resource budget? Perhaps the problem is simply too difficult for any implementation to be within budget.

These problems can often be avoided by using asymptotic analysis. Asymptotic analysis measures the efficiency of an algorithm, or its implementation as a program, as the input size becomes large. It is actually an estimating technique and does not tell us anything about the relative merits of two programs where one is always “slightly faster” than the other. However, asymptotic analysis has proved useful to computer scientists who must determine if a particular algorithm is worth considering for implementation.

The critical resource for a program is most often its running time. However, you cannot pay attention to running time alone. You must also be concerned with other factors such as the space required to run the program (both main memory and disk space). Typically you will analyze the *time* required for an *algorithm* (or the instantiation of an algorithm in the form of a program), and the *space* required for a *data structure*.

Many factors affect the running time of a program. Some relate to the environment in which the program is compiled and run. Such factors include the speed of the computer’s CPU, bus, and peripheral hardware. Competition with other users for the computer’s (or the network’s) resources can make a program slow to a crawl. The programming language and the quality of code generated by a particular compiler can have a significant effect. The “coding efficiency” of the programmer who converts the algorithm to a program can have a tremendous impact as well.

If you need to get a program working within time and space constraints on a particular computer, all of these factors can be relevant. Yet, none of these factors address the differences between two algorithms or data structures. To be fair, if you want to compare two programs derived from two algorithms for solving the same problem, they should both be compiled with the same compiler and run on the same computer under the same conditions. As much as possible, the same amount of care should be taken in the programming effort devoted to each program to make the implementations "equally efficient". In this sense, all of the factors mentioned above should cancel out of the comparison because they apply to both algorithms equally.

If you truly wish to understand the running time of an algorithm, there are other factors that are more appropriate to consider than machine speed, programming language, compiler, and so forth. Ideally we would measure the running time of the algorithm under standard benchmark conditions. However, we have no way to calculate the running time reliably other than to run an implementation of the algorithm on some computer. The only alternative is to use some other measure as a surrogate for running time.

2.3.1.2. Basic Operations and Input Size

Of primary consideration when estimating an algorithm's performance is the number of **basic operations** required by the algorithm to process an input of a certain size. The terms "basic operations" and "size" are both rather vague and depend on the algorithm being analyzed. Size is often the number of inputs processed. For example, when comparing sorting algorithms the size of the problem is typically measured by the number of records to be sorted. A basic operation must have the property that its time to complete does not depend on the particular values of its operands. Adding or comparing two integer variables are examples of basic operations in most programming languages. Summing the contents of an array containing n integers is not, because the cost depends on the value of n (i.e., the size of the input).

Example 2.3.1

Consider a simple algorithm to solve the problem of finding the largest value in an array of n integers. The algorithm looks at each integer in turn, saving the position of the largest value seen so far. This algorithm is called the *largest-value sequential search* and is illustrated by the following function:

Toggle Tree View

```
// Return position of largest value in integer array A
static int largest(int[] A) {
  int currlarge = 0;             // Position of largest element seen
  for (int i=1; i<A.length; i++) { // For each element
    if (A[currlarge] < A[i]) {    //   if A[i] is larger
       currlarge = i;            //     remember its position
    }
  }
  return currlarge;              // Return largest position
}
```

Here, the size of the problem is `A.length`, the number of integers stored in array `A`. The basic operation is to compare an integer's value to that of the largest value seen so far. It is reasonable to assume that it takes a fixed amount of time to do one such comparison, regardless of the value of the two integers or their positions in the array.

Because the most important factor affecting running time is normally size of the input, for a given input size n we often express the time $\mathbf{T}$ to run the algorithm as a function of n, written as $\mathbf{T}(n)$. We will always assume $\mathbf{T}(n)$ is a non-negative value.

Let us call c the amount of time required to compare two integers in function `largest`. We do not care right now what the precise value of c might be. Nor are we concerned with the time required to increment variable i because this must be done for each value in the array, or the time for the actual assignment when a larger value is found, or the little bit of extra time taken to initialize `currlarge`. We just want a reasonable approximation for the time taken to execute the algorithm. The total time to run `largest` is therefore approximately cn, because we must make n comparisons, with each comparison costing c time. We say that function `largest` (and by extension, the largest-value sequential search algorithm for any typical implementation) has a running time expressed by the equation

$$\mathbf{T}(n) = cn.$$

This equation describes the growth rate for the running time of the largest-value sequential search algorithm.

Example 2.3.2

The running time of a statement that assigns the first value of an integer array to a variable is simply the time required to copy the value of the first array value. We can assume this assignment takes a constant amount of time regardless of the value. Let us call c_1 the amount of time necessary to copy an integer. No matter how large the array on a typical computer (given reasonable conditions for memory and array size), the time to copy the value from the first position of the array is always c_1. Thus, the equation for this algorithm is simply

$$\mathbf{T}(n) = c_1,$$

indicating that the size of the input n has no effect on the running time. This is called a **constant running time**.

Example 2.3.3

Consider the following code:

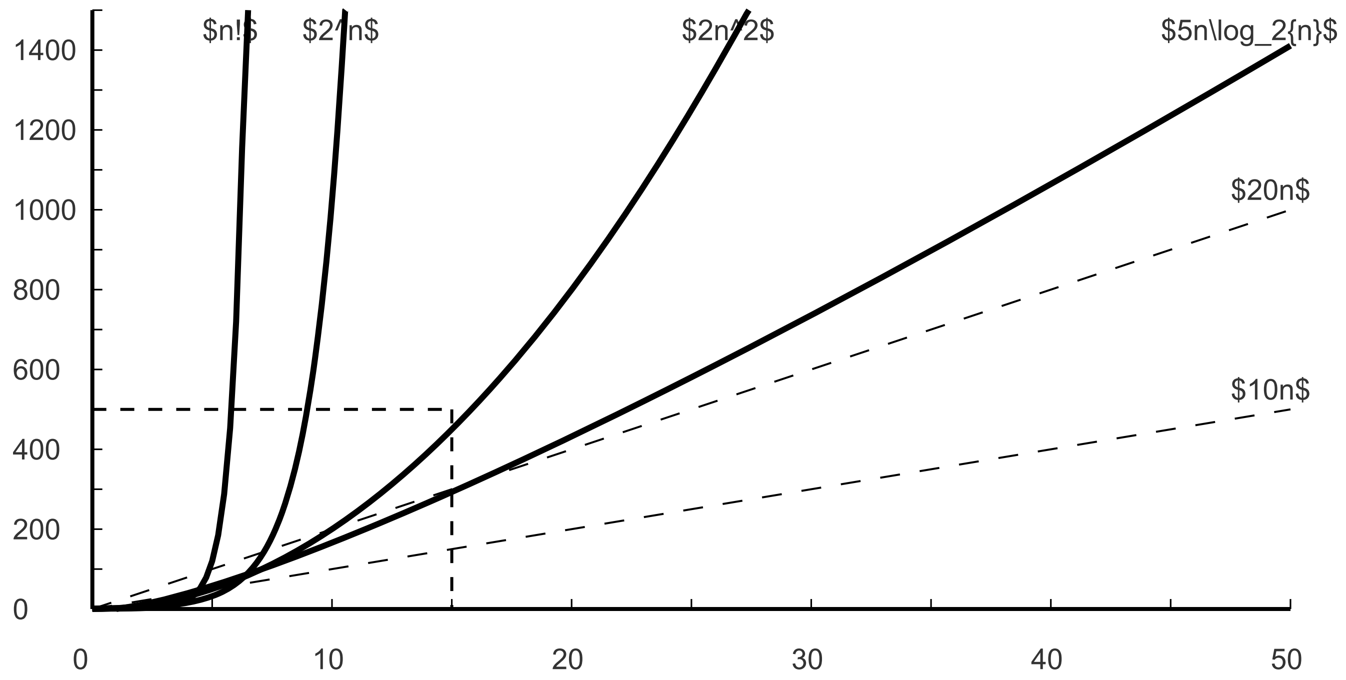
Toggle Tree View

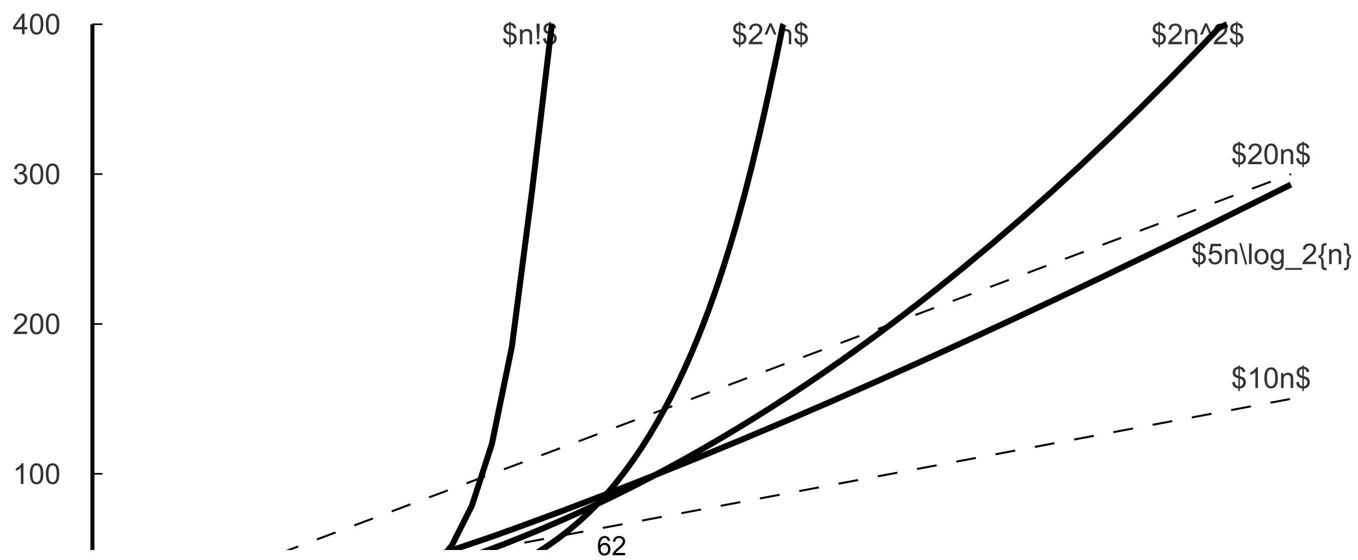
```
sum = 0;
for (i=1; i<=n; i++) {
  for (j=1; j<=n; j++) {
    sum++;
  }
}
```

What is the running time for this code fragment? Clearly it takes longer to run when n is larger. The basic operation in this example is the increment operation for variable `sum`. We can assume that incrementing takes constant time; call this time c_2. (We can ignore the time required to initialize `sum`, and to increment the loop counters i and j. In practice, these costs can safely be bundled into time c_2.) The total number of increment operations is n^2. Thus, we say that the running time is $\mathbf{T}(n) = c_2n^2$.

2.3.1.3. Growth Rates

The **growth rate** for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows. The following figure shows a graph for six equations, each meant to describe the running time for a particular program or algorithm. A variety of growth rates that are representative of typical algorithms are shown.





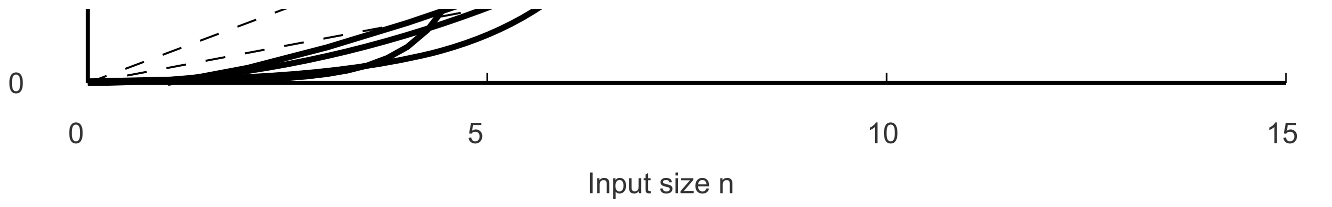


Figure 2.3.2: Two views of a graph illustrating the growth rates for six equations. The bottom view shows in detail the lower-left portion of the top view. The horizontal axis represents input size. The vertical axis can represent time, space, or any other measure of cost.

The two equations labeled $10n$ and $20n$ are graphed by straight lines. A growth rate of cn (for c any positive constant) is often referred to as a **linear growth rate** or running time. This means that as the value of n grows, the running time of the algorithm grows in the same proportion. Doubling the value of n roughly doubles the running time. An algorithm whose running-time equation has a highest-order term containing a factor of n^2 is said to have a **quadratic growth rate**. In the figure, the line labeled $2n^2$ represents a quadratic growth rate. The line labeled 2^n represents an **exponential growth rate**. This name comes from the fact that n appears in the exponent. The line labeled $n!$ also grows exponentially.

As you can see from the figure, the difference between an algorithm whose running time has cost $\mathbf{T}(n) = 10n$ and another with cost $\mathbf{T}(n) = 2n^2$ becomes tremendous as n grows. For $n > 5$, the algorithm with running time $\mathbf{T}(n) = 2n^2$ is already much slower. This is despite the fact that $10n$ has a greater constant factor than $2n^2$. Comparing the two curves marked $20n$ and $2n^2$ shows that changing the constant factor for one of the equations only shifts the point at which the two curves cross. For $n > 10$, the algorithm with cost $\mathbf{T}(n) = 2n^2$ is slower than the algorithm with cost $\mathbf{T}(n) = 20n$. This graph also shows that the equation $\mathbf{T}(n) = 5n \log n$ grows somewhat more quickly than both $\mathbf{T}(n) = 10n$ and $\mathbf{T}(n) = 20n$, but not nearly so quickly as the equation $\mathbf{T}(n) = 2n^2$. For constants $a, b > 1, n^a$ grows faster than either $\log^b n$ or $\log n^b$. Finally, algorithms with cost $\mathbf{T}(n) = 2^n$ or $\mathbf{T}(n) = n!$ are prohibitively expensive for even modest values of n. Note that for constants $a, b \geq 1, a^n$ grows faster than n^b.

We can get some further insight into relative growth rates for various algorithms from the following table. Most of the growth rates that appear in typical algorithms are shown, along with some representative input sizes. Once again, we see that the growth rate has a tremendous effect on the resources consumed by an algorithm.

Table 2.3.1

Costs for representative growth rates.

n	log log n	log n	n	n log n	n^2	n^3	2^n
16	2	4	2^4	$4 \cdot 2^4 = 2^6$	2^8	2^{12}	2^{16}
256	3	8	2^8	$8 \cdot 2^8 = 2^{11}$	2^{16}	2^{24}	2^{256}
1024	≈ 3.3	10	2^{10}	$10 \cdot 2^{10} \approx 2^{13}$	2^{20}	2^{30}	2^{1024}
64K	4	16	2^{16}	$16 \cdot 2^{16} = 2^{20}$	2^{32}	2^{48}	$2^{64\mathrm{K}}$
1M	≈ 4.3	20	2^{20}	$20 \cdot 2^{20} \approx 2^{24}$	2^{40}	2^{60}	$2^{1\mathrm{M}}$
1G	≈ 4.9	30	2^{30}	$30 \cdot 2^{30} \approx 2^{35}$	2^{60}	2^{90}	$2^{1\mathrm{G}}$

Practicing Comparing Growth Rates Exercise

Current score: 0 out of

You are given this set of growth functions: $n!, 2^n, 2n^2, 5n\log n, 20n, 10n$

For the growth function $10n$, type a value (a positive integer) for which this function is the most efficient of the six. If there is no integer value for which it is most efficent, type "none".

Answer

Check Answer

Need help?

I'd like a hint

2.3.2. Growth Rates Ordering Exercise

Practicing Growth Rates Ordering Exercise

Current score: 0 out of 5

Your task in this exercise is to put the following functions into their appropriate positions in the list so that finally the list will contain all the functions in ascending order of their growth rates. You can swap two functions by clicking on them.

Reset

Answer

Check Answer

Need help?

I'd like a hint

n^2 $\quad 2\log^3 n$ $\quad n\log\log n$ $\quad 2^{n^2}$ $\quad 2^{\sqrt{n}}$ $\quad 2^n$

2.4. Best, Worst, and Average Cases

2.4.1. Best, Worst, and Average Cases

1 / 11

Consider the problem of finding the factorial of n.

Factorial Problem

1 / 11

For some algorithms, different inputs of a given size require different amounts of time. For example, co
problem of searching an array containing n integers to find the one with a particular value K (assume that
exactly once in the array).

Sequential Search

When analyzing an algorithm, should we study the best, worst, or average case? Normally we are not interested in the best case, because this might happen only rarely and generally is too optimistic for a fair characterization of the algorithm's running time. In other words, analysis based on the best case is not likely to be representative of the behavior of the algorithm. However, there are rare instances where a best-case analysis is useful—in particular, when the best case has high probability of occurring. The **Shellsort** and **Quicksort** algorithms both can take advantage of the best-case running time of **Insertion Sort** to become more efficient.

How about the worst case? The advantage to analyzing the worst case is that you know for certain that the algorithm must perform at least that well. This is especially important for real-time applications, such as for the computers that monitor an air traffic control system. Here, it would not be acceptable to use an algorithm that can handle n airplanes quickly enough *most of the time*, but which fails to perform quickly enough when all n airplanes are coming from the same direction.

For other applications—particularly when we wish to aggregate the cost of running the program many times on many different inputs—worst-case analysis might not be a representative measure of the algorithm's performance. Often we prefer to know the average-case running time. This means that we would like to know the *typical* behavior of the algorithm on inputs of size n. Unfortunately, average-case analysis is not always possible. Average-case analysis first requires that we understand how the actual inputs to the program (and their costs) are distributed with respect to the set of all possible inputs to the program. For example, it was stated previously that the sequential search algorithm on average examines half of the array values. This is only true if the element with value K is equally likely to appear in any position in the array. If this assumption is not correct, then the algorithm does *not* necessarily examine half of the array values in the average case.

The characteristics of a data distribution have a significant effect on many search algorithms, such as those based on **hashing** and search trees such as the **BST**. Incorrect assumptions about data distribution can have disastrous consequences on a program's space or time performance. Unusual data distributions can also be used to advantage, such as is done by **self-organizing lists**.

In summary, for real-time applications we are likely to prefer a worst-case analysis of an algorithm. Otherwise, we often desire an average-case analysis if we know enough about the distribution of our input to compute the average

02.05 Faster Computer, or Faster Algorithm?

Due No Due Date **Points** 1 **Submitting** an external tool

02.05 Faster Computer, or Faster Algorithm?

2.5. Faster Computer, or Faster Algorithm?

2.5.1. Faster Computer, or Faster Algorithm?

Imagine that you have a problem to solve, and you know of an algorithm whose running time is proportional to n^2 where n is a measure of the input size. Unfortunately, the resulting program takes ten times too long to run. If you replace your current computer with a new one that is ten times faster, will the n^2 algorithm become acceptable? If the problem size remains the same, then perhaps the faster computer will allow you to get your work done quickly enough even with an algorithm having a high growth rate. But a funny thing happens to most people who get a faster computer. They don't run the same problem faster. They run a bigger problem! Say that on your old computer you were content to sort 10,000 records because that could be done by the computer during your lunch break. On your new computer you might hope to sort 100,000 records in the same time. You won't be back from lunch any sooner, so you are better off solving a larger problem. And because the new machine is ten times faster, you would like to sort ten times as many records.

If your algorithm's growth rate is linear (i.e., if the equation that describes the running time on input size n is $\mathbf{T}(n) = cn$ for some constant c), then 100,000 records on the new machine will be sorted in the same time as 10,000 records on the old machine. If the algorithm's growth rate is greater than cn, such as c_1n^2, then you will *not* be able to do a problem ten times the size in the same amount of time on a machine that is ten times faster.

How much larger a problem can be solved in a given amount of time by a faster computer? Assume that the new machine is ten times faster than the old. Say that the old machine could solve a problem of size n in an hour. What is the largest problem that the new machine can solve in one hour? The following table shows how large a problem can be solved on the two machines for five running-time functions.

Table 2.5.1

The increase in problem size that can be run in a fixed period of time on a computer that is ten times faster. The first column lists the right-hand sides for five growth rate equations. For the purpose of this example, arbitrarily assume that the old machine can run 10,000 basic operations in one hour. The second column shows the maximum value for n that can be run in 10,000 basic operations on the old machine. The third column shows the value for n', the new maximum size for the problem that can be run in the same time on the new machine that is ten times faster. Variable n' is the greatest size for the problem that can run in 100,000 basic operations. The fourth column shows how the size of n changed to become n' on the new machine. The fifth column shows the increase in the problem size as the ratio of n' to n.

increase in the problem size as the ratio of n' to n.

$\mathbf{f(n)}$	$\mathbf{n}$	$\mathbf{n'}$	**Change**	$\mathbf{n'/n}$
$10n$	1000	$10,000$	$n' = 10n$	10
$20n$	500	5000	$n' = 10n$	10
$5n \log n$	250	1842	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	$--$

This table illustrates many important points. The first two equations are both linear; only the value of the constant factor has changed. In both cases, the machine that is ten times faster gives an increase in problem size by a factor of ten. In other words, while the value of the constant does affect the absolute size of the problem that can be solved in a fixed amount of time, it does not affect the *improvement* in problem size (as a proportion to the original size) gained by a faster computer. This relationship holds true regardless of the algorithm's growth rate: Constant factors never affect the relative improvement gained by a faster computer.

An algorithm with time equation $\mathbf{T}(n) = 2n^2$ does not receive nearly as great an improvement from the faster machine as an algorithm with linear growth rate. Instead of an improvement by a factor of ten, the improvement is only the square root of that: $\sqrt{10} \approx 3.16$. Thus, the algorithm with higher growth rate not only solves a smaller problem in a given time in the first place, it *also* receives less of a speedup from a faster computer. As computers get ever faster, the disparity in problem sizes becomes ever greater.

The algorithm with growth rate $\mathbf{T}(n) = 5n \log n$ improves by a greater amount than the one with quadratic growth rate, but not by as great an amount as the algorithms with linear growth rates.

Note that something special happens in the case of the algorithm whose running time grows exponentially. If you look at its plot on a graph, the curve for the algorithm whose time is proportional to 2^n goes up very quickly as n grows. The increase in problem size on the machine ten times as fast is about $n + 3$ (to be precise, it is $n + \log_2 10$). The increase in problem size for an algorithm with exponential growth rate is by a constant addition, not by a multiplicative factor. Because the old value of n was 13, the new problem size is 16. If next year you buy another computer ten times faster yet, then the new computer (100 times faster than the original computer) will only run a problem of size 19. If you had a second program whose growth rate is 2^n and for which the original computer could run a problem of size 1000 in an hour, than a machine ten times faster can run a problem only of size 1003 in an hour! Thus, an exponential growth rate is radically different than the other growth rates shown in the table. The significance of this difference is an important topic in **computational complexity theory**.

Instead of buying a faster computer, consider what happens if you replace an algorithm whose running time is proportional to n^2 with a new algorithm whose running time is proportional to $n \log n$. In a graph relating growth rate functions to input size, a fixed amount of time would appear as a horizontal line. If the line for the amount of time available to solve your problem is above the point at which the curves for the two growth rates in question meet, then the algorithm whose running time grows less quickly is faster. An algorithm with running time $\mathbf{T}n = n^2$ requires $1024 \times 1024 = 1,048,576$ time steps for an input of size $n = 1024$. An algorithm with running time $\mathbf{T}(n) = n \log n$ requires $1024 \times 10 = 10,240$ time steps for an input of size $n = 1024$, which is an improvement of much more than a factor of ten when compared to the algorithm with running time $\mathbf{T}(n) = n^2$. Because $n^2 > 10n \log n$ whenever $n > 58$, if the typical problem size is larger than 58 for this example, then you would be much better off changing algorithms instead of buying a computer ten times faster. Furthermore, when you do buy a faster computer, an algorithm with a slower growth rate provides a greater benefit in terms of larger problem size that can run in a certain time on the new computer.

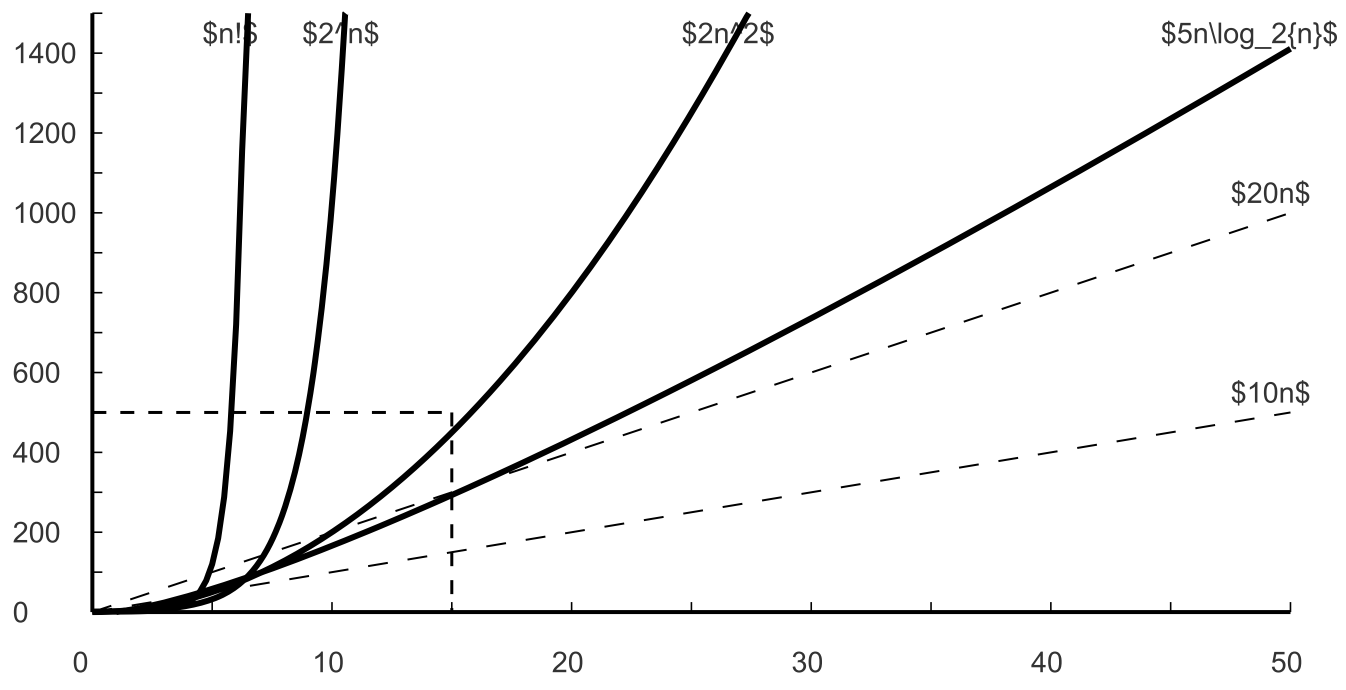
02.06 Asymptotic Analysis and Upper Bounds

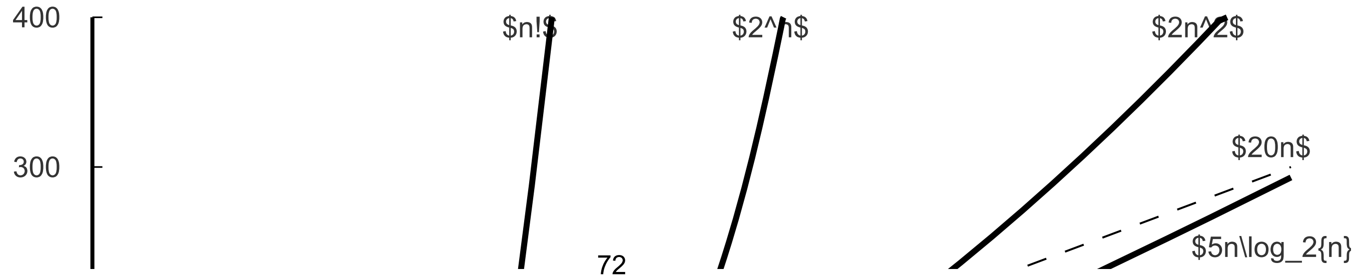
Due No Due Date **Points** 1 **Submitting** an external tool

02.06 Asymptotic Analysis and Upper Bounds

2.6. Asymptotic Analysis and Upper Bounds

2.6.1. Asymptotic Analysis and Upper Bounds





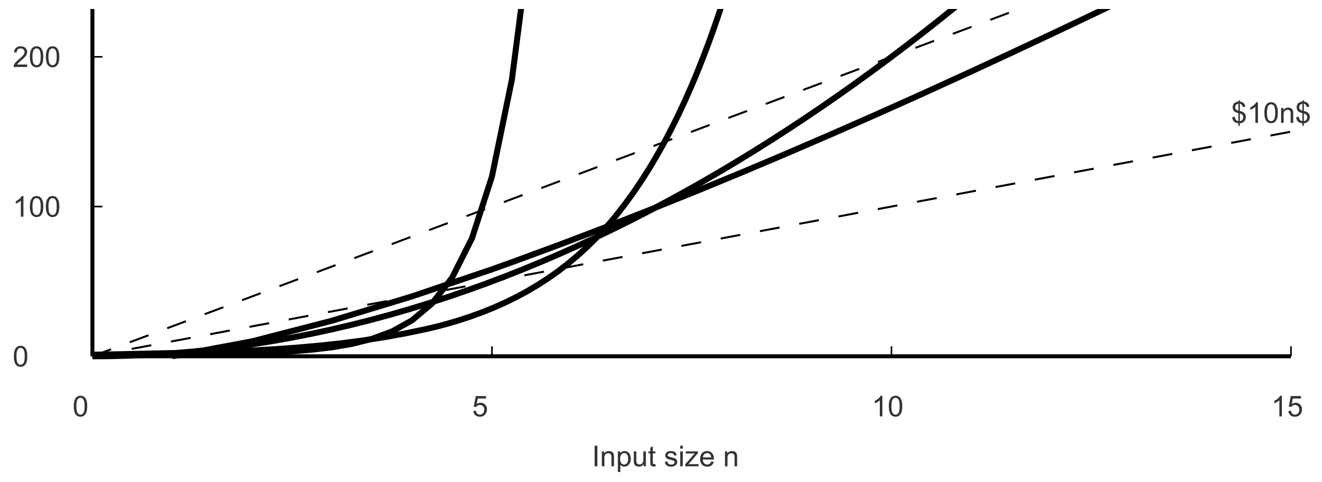


Figure 2.6.2: Two views of a graph illustrating the growth rates for six equations. The bottom view shows in detail the lower-left portion of the top view. The horizontal axis represents input size. The vertical axis can represent time, space, or any other measure of cost.

Despite the larger constant for the curve labeled $10n$ in the figure above, $2n^2$ crosses it at the relatively small value of $n = 5$. What if we double the value of the constant in front of the linear equation? As shown in the graph, $20n$ is surpassed by $2n^2$ once $n = 10$. The additional factor of two for the linear **growth rate** does not much matter. It only doubles the x-coordinate for the intersection point. In general, changes to a constant factor in either equation only shift *where* the two curves cross, not *whether* the two curves cross.

When you buy a faster computer or a faster compiler, the new problem size that can be run in a given amount of time for a given growth rate is larger by the same factor, regardless of the constant on the running-time equation. The time curves for two algorithms with different growth rates still cross, regardless of their running-time equation constants. For these reasons, we usually ignore the constants when we want an estimate of the growth rate for the running time or other resource requirements of an algorithm. This simplifies the analysis and keeps us thinking about the most important aspect: the growth rate. This is called **asymptotic algorithm analysis**. To be precise, asymptotic analysis refers to the study of an algorithm as the input size “gets big” or reaches a limit (in the calculus sense). However, it has proved to be so useful to ignore all constant factors that asymptotic analysis is used for most algorithm comparisons.

In rare situations, it is not reasonable to ignore the constants. When comparing algorithms meant to run on small values of n, the constant can have a large effect. For example, if the problem requires you to sort many collections of exactly five records, then a sorting algorithm designed for sorting thousands of records is probably not appropriate, even if its asymptotic analysis indicates good performance. There are rare cases where the constants for two algorithms under comparison can differ by a factor of 1000 or more, making the one with lower growth rate impractical for typical problem sizes due to its large constant. Asymptotic analysis is a form of “back of the envelope” **estimation** for algorithm resource consumption. It provides a simplified model of the running time or other resource needs of an algorithm. This simplification usually helps you understand the behavior of your algorithms. Just be aware of the limitations to asymptotic analysis in the rare situation where the constant is important.

2.6.1.1. Upper Bounds

Several terms are used to describe the running-time equation for an algorithm. These terms—and their associated symbols—indicate precisely what aspect of the algorithm’s behavior is being described. One is the **upper bound** for the growth of the algorithm’s running time. It indicates the upper or highest growth rate that the algorithm can have the growth of the algorithm's running time. It indicates the upper or highest growth rate that the algorithm can have.

Because the phrase "has an upper bound to its growth rate of $f(n)$" is long and often used when discussing algorithms, we adopt a special notation, called **big-Oh notation**. If the upper bound for an algorithm's growth rate (for, say, the worst case) is (f(n)), then we would write that this algorithm is "in the set $O(f(n))$ in the worst case" (or just "in $O(f(n))$ in the worst case"). For example, if n^2 grows as fast as $\mathbf{T}(n)$ (the running time of our algorithm) for the worst-case input, we would say the algorithm is "in $O(n^2)$ in the worst case".

The following is a precise definition for an upper bound. $\mathbf{T}(n)$ represents the true running time of the algorithm. $f(n)$ is some expression for the upper bound.

> For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in set $O(f(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \leq cf(n)$ for all $n > n_0$.

Constant n_0 is the smallest value of n for which the claim of an upper bound holds true. Usually n_0 is small, such as 1, but does not need to be. You must also be able to pick some constant c, but it is irrelevant what the value for c actually is. In other words, the definition says that for *all* inputs of the type in question (such as the worst case for all inputs of size n) that are large enough (i.e., $n > n_0$), the algorithm *always* executes in less than or equal to $cf(n)$ steps for some constant c.

Example 2.6.1

Consider the sequential search algorithm for finding a specified value in an array of integers. If visiting and examining one value in the array requires c_s steps where c_s is a positive number, and if the value we search for has equal probability of appearing in any position in the array, then in the average case $\mathbf{T}(n) = c_s n/2$. For all values of $n > 1$, $c_s n/2 \leq c_s n$. Therefore, by the definition, $\mathbf{T}(n)$ is in $O(n)$ for $n_0 = 1$ and $c = c_s$.

Example 2.6.2

For a particular algorithm, $\mathbf{T}(n) = c_1 n^2 + c_2 n$ in the average case where c_1 and c_2 are positive numbers. Then,

$$c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2) n^2$$

for all $n > 1$. So, $\mathbf{T}(n) \leq cn^2$ for $c = c_1 + c_2$, and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ is in $O(n^2)$ by the second definition.

Example 2.6.3

Assigning the value from the first position of an array to a variable takes constant time regardless of the size of the array. Thus, $\mathbf{T}(n) = c$ (for the best, worst, and average cases). We could say in this case that $\mathbf{T}(n)$ is in $O(c)$. However, it is traditional to say that an algorithm whose running time has a constant upper bound is in $O(1)$.

If someone asked you out of the blue "Who is the best?" your natural reaction should be to reply "Best at what?" In the same way, if you are asked "What is the growth rate of this algorithm", you would need to ask "When? Best case? Average case? Or worst case?" Some algorithms have the same behavior no matter which input instance of a given size that they receive. An example is finding the maximum in an array of integers. But for many algorithms, it makes a big difference which particular input of a given size is involved, such as when searching an unsorted array for a particular value. So any statement about the upper bound of an algorithm must be in the context of some for a particular value. So any statement about the upper bound of an algorithm must be in the context of some specific class of inputs of size n. We measure this upper bound nearly always on the best-case, average-case, or worst-case inputs. Thus, we cannot say, “this algorithm has an upper bound to its growth rate of n^2” because that is an incomplete statement. We must say something like, “this algorithm has an upper bound to its growth rate of n^2 *in the average case*”.

Knowing that something is in $O(f(n))$ says only how bad things can be. Perhaps things are not nearly so bad. Because sequential search is in $O(n)$ in the worst case, it is also true to say that sequential search is in $O(n^2)$. But sequential search is practical for large n in a way that is not true for some other algorithms in $O(n^2)$. We always seek to define the running time of an algorithm with the tightest (lowest) possible upper bound. Thus, we prefer to say that sequential search is in $O(n)$. This also explains why the phrase “is in $O(f(n))$” or the notation “$\in O(f(n))$” is used instead of “is $O(f(n))$” or “$= O(f(n))$”. There is no strict equality to the use of big-Oh notation. $O(n)$ is in $O(n^2)$, but $O(n^2)$ is not in $O(n)$.

2.6.1.2. Simplifying Rules

Once you determine the running-time equation for an algorithm, it really is a simple matter to derive the big-Oh expressions from the equation. You do not need to resort to the formal definitions of asymptotic analysis. Instead, you can use the following rules to determine the simplest form.

1. If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
2. If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
3. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n) + f_2(n)$ is in $O(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$.

The first rule says that if some function $g(n)$ is an upper bound for your cost function, then any upper bound for $g(n)$ is also an upper bound for your cost function.

The significance of rule (2) is that you can ignore any multiplicative constants in your equations when using big-Oh notation.

Rule (3) says that given two parts of a program run in sequence (whether two statements or two sections of code), you need consider only the more expensive part.

Rule (4) is used to analyze simple loops in programs. If some action is repeated some number of times, and each repetition has the same cost, then the total cost is the cost of the action multiplied by the number of times that the action takes place.

Taking the first three rules collectively, you can ignore all constants and all lower-order terms to determine the asymptotic growth rate for any cost function. The advantages and dangers of ignoring constants were discussed near the beginning of this section. Ignoring lower-order terms is reasonable when performing an asymptotic analysis. The higher-order terms soon swamp the lower-order terms in their contribution to the total cost as (n) becomes larger. Thus, if $\mathbf{T}(n) = 3n^4 + 5n^2$, then $\mathbf{T}(n)$ is in $O(n^4)$. The n^2 term contributes relatively little to the total cost for large n.

From now on, we will use these simplifying rules when discussing the cost for a program or algorithm.

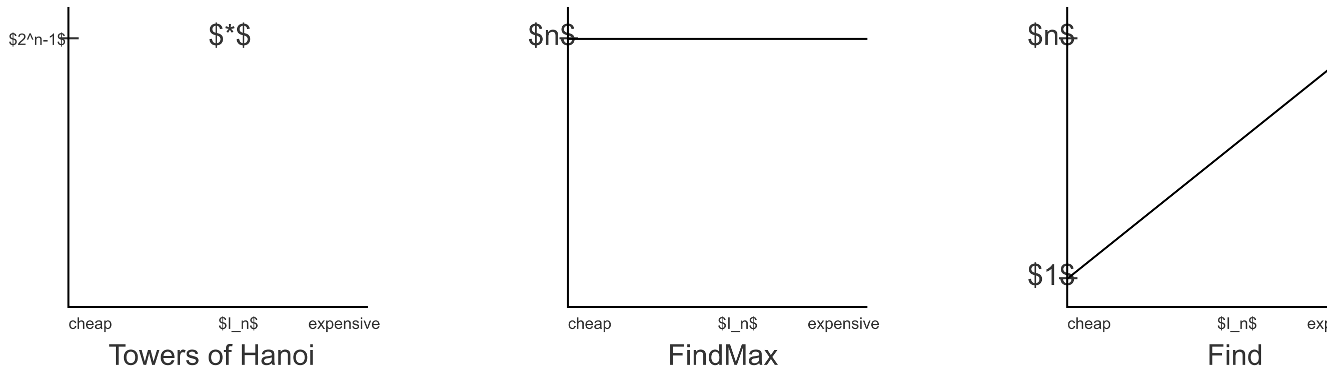
2.6.1.4. Summary

1 / 17

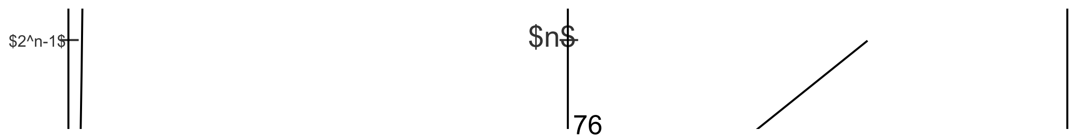
<< < > >>

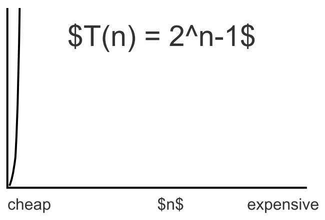
A mistake that people often make is to confuse the upper bound and the worst case.

Costs for all inputs of an arbitrary (but fixed) size n for three representative algorith

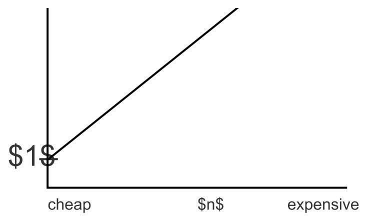


Costs, as n grows, for some representative algorithms

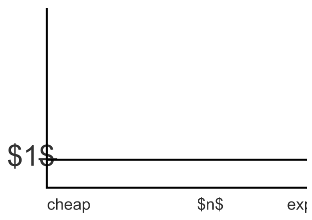




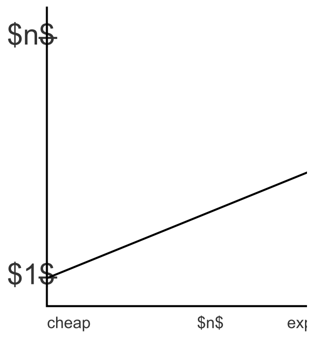
Towers of Hanoi



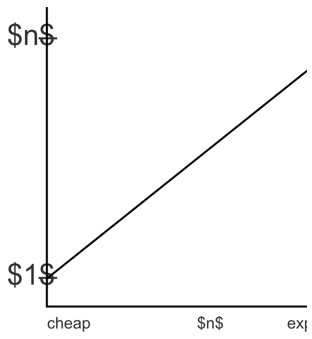
FindMax



Find (Best)



Find (Average



Find (Worst)

2.6.1.5. Practice Questions

02.07 Lower Bounds and Theta Notation

Due No Due Date **Points** 1 **Submitting** an external tool

02.07 Lower Bounds and Theta Notation

2.7. Lower Bounds and Θ Notation

2.7.1. Lower Bounds and Theta Notation

2.7.1.1. Lower Bounds

Big-Oh notation describes an upper bound. In other words, big-Oh notation states a claim about the greatest amount of some resource (usually time) that is required by an algorithm for some class of inputs of size n (typically the worst such input, the average of all possible inputs, or the best such input).

Similar notation is used to describe the least amount of a resource that an algorithm needs for some class of input. Like big-Oh notation, this is a measure of the algorithm's growth rate. Like big-Oh notation, it works for any resource, but we most often measure the least amount of time required. And again, like big-Oh notation, we are measuring the resource required for some particular class of inputs: the worst-, average-, or best-case input of size n.

The **lower bound** for an algorithm (or a problem, as explained later) is denoted by the symbol Ω, pronounced "big-Omega" or just "Omega". The following definition for Ω is symmetric with the definition of big-Oh.

> For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in set $\Omega(g(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \geq cg(n)$ for all $n > n_0$. 1

Example 2.7.1

Assume $\mathbf{T}(n) = c_1n^2 + c_2n$ for c_1 and $c_2 > 0$. Then,

$$c_1n^2 + c_2n \geq c_1n^2$$

for all $n > 1$. So, $\mathbf{T}(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

It is also true that the equation of the example above is in $\Omega(n)$. However, as with big-Oh notation, we wish to get the "tightest" (for Ω notation, the largest) bound possible. Thus, we prefer to say that this running time is in $\Omega(n^2)$.

Recall the sequential search algorithm to find a value K within an array of integers. In the average and worst cases this algorithm is in $\Omega(n)$, because in both the average and worst cases we must examine *at least* cn values (where c

is 1/2 in the average case and 1 in the worst case).

1

An alternate (non-equivalent) definition for Ω is

> $\mathbf{T}(n)$ is in the set $\Omega(g(n))$ if there exists a positive constant c such that $\mathbf{T}(n) \geq cg(n)$ for an infinite number of values for n.

This definition says that for an “interesting” number of cases, the algorithm takes at least $cg(n)$ time. Note that this definition is *not* symmetric with the definition of big-Oh. For $g(n)$ to be a lower bound, this definition *does not* require that $\mathbf{T}(n) \geq cg(n)$ for all values of n greater than some constant. It only requires that this happen often enough, in particular that it happen for an infinite number of values for n. Motivation for this alternate definition can be found in the following example.

Assume a particular algorithm has the following behavior:

$$\mathbf{T}(n) = \begin{cases} n & \text{for all odd } n \geq 1 \\ n^2/100 & \text{for all even } n \geq 0 \end{cases}$$

From this definition, $n^2/100 \geq \frac{1}{100}n^2$ for all even $n \geq 0$. So, $\mathbf{T}(n) \geq cn^2$ for an infinite number of values of n (i.e., for all even n) for $c = 1/100$. Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

For this equation for $\mathbf{T}(n)$, it is true that all inputs of size n take at least cn time. But an infinite number of inputs of size n take cn^2 time, so we would like to say that the algorithm is in $\Omega(n^2)$. Unfortunately, using our first definition will yield a lower bound of $\Omega(n)$ because it is not possible to pick constants c and n_0 such that $\mathbf{T}(n) \geq cn^2$ for all $n > n_0$. The alternative definition does result in a lower bound of $\Omega(n^2)$ for this algorithm, which seems to fit common sense more closely. Fortunately, few real algorithms or computer programs display the pathological behavior of this example. Our first definition for Ω generally yields the expected result.

As you can see from this discussion, asymptotic bounds notation is not a law of nature. It is merely a powerful modeling tool used to describe the behavior of algorithms.

2.7.1.2. Theta Notation

The definitions for big-Oh and Ω give us ways to describe the upper bound for an algorithm (if we can find an equation for the maximum cost of a particular class of inputs of size n) and the lower bound for an algorithm (if we can find an equation for the minimum cost for a particular class of inputs of size n). When the upper and lower bounds are the same within a constant factor, we indicate this by using Θ (big-Theta) notation. An algorithm is said to be $\Theta(h(n))$ if it is in $O(h(n))$ *and* it is in $\Omega(h(n))$. Note that we drop the word “in” for Θ notation, because there is a strict equality for two equations with the same Θ. In other words, if $f(n)$ is $\Theta(g(n))$, then $g(n)$ is $\Theta(f(n))$.

Because the sequential search algorithm is both in $O(n)$ and in $\Omega(n)$ in the average case, we say it is $\Theta(n)$ in the average case.

Given an algebraic equation describing the time requirement for an algorithm, the upper and lower bounds always meet. That is because in some sense we have a perfect analysis for the algorithm, embodied by the running-time equation. For many algorithms (or their instantiations as programs), it is easy to come up with the equation that defines their runtime behavior. The analysis for most commonly used algorithms is well understood and we can almost always give a Θ analysis for them. However, the class of **NP-Complete** problems all have no definitive Θ analysis, just some unsatisfying big-Oh and Ω analyses. Even some “simple” programs are hard to analyze. Nobody analysis, just some unsatisfying big-Oh and Ω analyses. Even some "simple" programs are hard to analyze. Nobody currently knows the true upper or lower bounds for the following code fragment.

Toggle Tree View

```
while (n > 1) {
  if (ODD(n)) {
    n = 3 * n + 1;
  }
   else{
     n = n / 2;
   }
}
```

While some textbooks and programmers will casually say that an algorithm is “order of” or “big-Oh” of some cost function, it is generally better to use Θ notation rather than big-Oh notation whenever we have sufficient knowledge about an algorithm to be sure that the upper and lower bounds indeed match. OpenDSA modules use Θ notation in preference to big-Oh notation whenever our state of knowledge makes that possible. Limitations on our ability to analyze certain algorithms may require use of big-Oh or Ω notations. In rare occasions when the discussion is explicitly about the upper or lower bound of a problem or algorithm, the corresponding notation will be used in preference to Θ notation.

2.7.1.3. Classifying Functions

Given functions $f(n)$ and $g(n)$ whose growth rates are expressed as algebraic equations, we might like to determine if one grows faster than the other. The best way to do this is to take the limit of the two functions as n grows towards infinity,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}.$$

If the limit goes to ∞, then $f(n)$ is in $\Omega(g(n))$ because $f(n)$ grows faster. If the limit goes to zero, then $f(n)$ is in $O(g(n))$ because $g(n)$ grows faster. If the limit goes to some constant other than zero, then $f(n) = \Theta(g(n))$ because both grow at the same rate.

Example 2.7.2

If $f(n) = n^2$ and $g(n) = 2n\log n$, is $f(n)$ in $O(g(n))$, $\Omega(g(n))$, or $\Theta(g(n))$? Since

$$\frac{n^2}{2n\log n} = \frac{n}{2\log n},$$

we easily see that

$$\lim_{n\to\infty} \frac{n^2}{2n\log n} = \infty$$

because n grows faster than $2\log n$. Thus, n^2 is in $\Omega(2n\log n)$.

1 / 10

<< < > >>

A mistake that people can make is to confuse the lower bound and the best case. In general, people find low
confusing, in part because for simple algorithms, they look just like the upper bound. Let's try to figure this out.

2.7.1.4. Summary Exercise

02.08 Calculating Program Running Time

Due No Due Date **Points** 2 **Submitting** an external tool

02.08 Calculating Program Running Time

2.8. Calculating Program Running Time

2.8.1. Calculating Program Running Time

This modules discusses the analysis for several simple code fragments. We will make use of the algorithm analysis simplifying rules:

1. If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
2. If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
3. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n) + f_2(n)$ is in $O(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$.

Example 2.8.1

We begin with an analysis of a simple assignment to an integer variable.

Toggle Tree View

```
a = b;
```

Because the assignment statement takes constant time, it is $\Theta(1)$.

Example 2.8.2

Consider a simple `for` loop.

Toggle Tree View

```
sum = 0;
for (i=1; i<=n; i++) {
   sum += n;
}
```

The first line is $\Theta(1)$. The `for` loop is repeated n times. The third line takes constant time so, by simplifying rule (4), the total cost for executing the two lines making up the `for` loop is $\Theta(n)$. By rule (3), the cost of the entire code fragment is also $\Theta(n)$.

Example 2.8.3

We now analyze a code fragment with several `for` loops, some of which are nested.

Toggle Tree View

```
sum = 0;
for (j=1; j<=n; j++) {      // First for loop
  for (i=1; i<=j; i++) {  //   is a double loop
    sum++;
  }
}
for (k=0; k<n; k++) {       // Second for loop
   A[k] = k;
}
```

This code fragment has three separate statements: the first assignment statement and the two `for` loops. Again the assignment statement takes constant time; call it c_1. The second `for` loop is just like the one in Example **2.8.2** and takes $c_2 n = \Theta(n)$ time.

The first `for` loop is a double loop and requires a special technique. We work from the inside of the loop outward. The expression `sum++` requires constant time; call it c_3. Because the inner `for` loop is executed j times, by simplifying rule (4) it has cost $c_3 j$. The outer `for` loop is executed n times, but each time the cost of the inner loop is different because it costs $c_3 j$ with j changing each time. You should see that for the first execution of the outer loop, j is 1. For the second execution of the outer loop, j is 2. Each time through the outer loop, j becomes one greater, until the last time through the loop when $j = n$. Thus, the total cost of the loop is c_3 times the sum of the integers 1 through n. We know that

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2},$$

which is $\Theta(n^2)$. By simplifying rule (3), $\Theta(c_1 + c_2 n + c_3 n^2)$ is simply $\Theta(n^2)$.

Example 2.8.4

Compare the asymptotic analysis for the following two code fragments.

Toggle Tree View

```
sum1 = 0;
for (i=1; i<=n; i++) {     // First double loop
  for (j=1; j<=n; j++) {  //   do n times
    sum1++;
  }
}

sum2 = 0;
for (i=1; i<=n; i++) {     // Second double loop
  for (j=1; j<=i; j++) {  //   do i times
    sum2++;
  }
}
```

In the first double loop, the inner `for` loop always executes n times. Because the outer loop executes n times, it should be obvious that the statement `sum1++` is executed precisely n^2 times. The second loop is similar to the one analyzed in the previous example, with cost $\sum_{j=1}^{n} j$. This is approximately $\frac{1}{2}n^2$. Thus, both double loops cost $\Theta(n^2)$, though the second requires about half the time of the first.

Example 2.8.5

Not all doubly nested `for` loops are $\Theta(n^2)$. The following pair of nested loops illustrates this fact.

Toggle Tree View

```
sum1 = 0;
for (k=1; k<=n; k*=2) {    // Do log n times
  for (j=1; j<=n; j++) {  // Do n times
    sum1++;
  }
}

sum2 = 0;
for (k=1; k<=n; k*=2) {    // Do log n times
  for (j=1; j<=k; j++) {  // Do k times
    sum2++;
  }
}
```

When analyzing these two code fragments, we will assume that n is a power of two. The first code fragment has its outer `for` loop executed $\log n + 1$ times because on each iteration k is multiplied by two until it reaches n. Because the inner loop always executes n times, the total cost for the first code fragment can be expressed as

$$\sum_{i=0}^{\log n} n = n \log n.$$

So the cost of this first double loop is $\Theta(n \log n)$. Note that a variable substitution takes place here to create the summation, with $k = 2^i$.

In the second code fragment, the outer loop is also executed $\log n + 1$ times. The inner loop has cost k, which doubles each time. The summation can be expressed as

$$\sum_{i=0}^{\log n} 2^i = \Theta(n)$$

where n is assumed to be a power of two and again $k = 2^i$.

What about other control statements? `While` loops are analyzed in a manner similar to `for` loops. The cost of an `if` statement in the worst case is the greater of the costs for the `then` and `else` clauses. This is also true for the average case, assuming that the size of n does not affect the probability of executing one of the clauses (which is usually, but not necessarily, true). For `switch` statements, the worst-case cost is that of the most expensive branch. For subroutine calls, simply add the cost of executing the subroutine.

There are rare situations in which the probability for executing the various branches of an `if` or `switch` statement are functions of the input size. For example, for input of size n, the `then` clause of an `if` statement might be executed with probability $1/n$. An example would be an `if` statement that executes the `then` clause only for the smallest of n values. To perform an average-case analysis for such programs, we cannot simply count the cost of the `if` statement as being the cost of the more expensive branch. In such situations, the technique of **amortized analysis** can come to the rescue.

Determining the execution time of a recursive subroutine can be difficult. The running time for a recursive subroutine is typically best expressed by a recurrence relation. For example, the recursive factorial function calls itself with a value one less than its input value. The result of this recursive call is then multiplied by the input value, which takes constant time. Thus, the cost of the factorial function, if we wish to measure cost in terms of the number of multiplication operations, is one more than the number of multiplications made by the recursive call on the smaller input. Because the base case does no multiplications, its cost is zero. Thus, the running time for this function can be expressed as

$$T(n) = T(n-1) + 1 \text{ for } n > 1;\ \ T(1) = 0.$$

The closed-form solution for this recurrence relation is $\Theta(n)$.

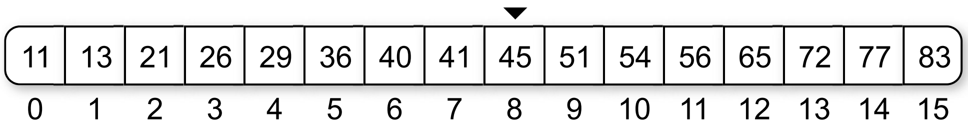
2.8.1.1. Case Study: Two Search Algorithms

The final example of algorithm analysis for this section will compare two algorithms for performing search in an array. Earlier, we determined that the running time for sequential search on an array where the search value K is equally likely to appear in any location is $\Theta(n)$ in both the average and worst cases. We would like to compare this running time to that required to perform a **binary search** on an array whose values are stored in order from lowest to highest. Here is a visualization of the binary search method.

1 / 12 &lt;&lt; &lt; &gt; &gt;&gt;

The input is a sorted array, and in this example we will search for the record with key value 45. We will pu
above 45 as a reminder that this is what we will be searching for.

```
// Return the position of an element in sorted array A with value K.
// If K is not in A, return A.length.
public static int binarySearch(int[] A, int K) {
  int low = 0;
  int high = A.length - 1;
  while(low <= high) {                  // Stop when low and high meet
    int mid = (low + high) / 2;         // Check middle of subarray
    if( A[mid] < K) low = mid + 1;      // In right half
    else if(A[mid] > K) high = mid - 1; // In left half
    else return mid;                    // Found it
  }
  return A.length;                      // Search value not in A
}
```

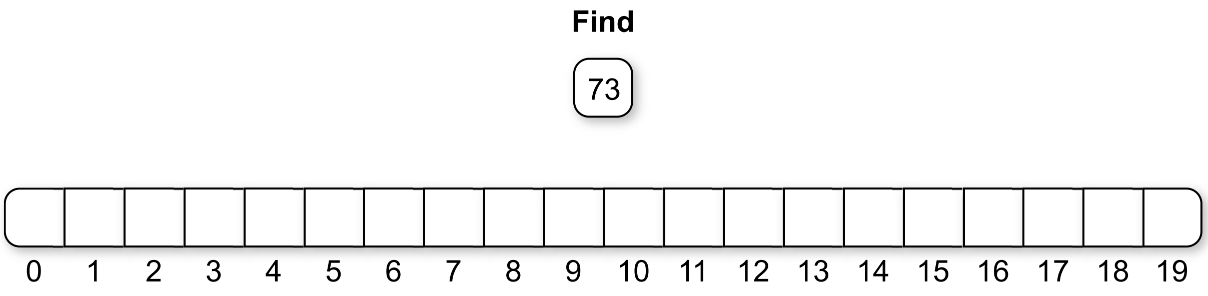


2.8.1.2. Binary Search Practice Exercise

Undo | Reset | Model Answer | Grade

Instructions:

The blue box contains a search key. The array stores values in ascending order, but these are intially hidden
Find the key in the array by clicking on the midpoint positions as they would be calculated by the binary searc
Whenever you click in the array, the value stored there will be displayed. Remember that midpoint calculation
arithmetic, so the position calculation rounds down.



2.8.1.3. Analyzing Binary Search

1 / 8

<< < > >>

To find the cost of binary search in the worst case, we can model the running time as a recurrence and the closed-form solution. Each recursive call to `binarySearch` cuts the size of the array approximately in half, model the worst-case cost as follows, assuming for simplicity that n is a power of two.
$\Theta(n) = \Theta(n/2) + 1, \Theta(1) = 1$

Function `binarySearch` is designed to find the (single) occurrence of K and return its position. A special value is returned if K does not appear in the array. This algorithm can be modified to implement variations such as returning the position of the first occurrence of K in the array if multiple occurrences are allowed, and returning the position of the greatest value less than K when K is not in the array.

Comparing sequential search to binary search, we see that as n grows, the $\Theta(n)$ running time for sequential search in the average and worst cases quickly becomes much greater than the $\Theta(\log n)$ running time for binary search. Taken in isolation, binary search appears to be much more efficient than sequential search. This is despite the fact that the constant factor for binary search is greater than that for sequential search, because the calculation for the next search position in binary search is more expensive than just incrementing the current position, as sequential search does.

Note however that the running time for sequential search will be roughly the same regardless of whether or not the array values are stored in order. In contrast, binary search requires that the array values be ordered from lowest to highest. Depending on the context in which binary search is to be used, this requirement for a sorted array could be detrimental to the running time of a complete program, because maintaining the values in sorted order requires a greater cost when inserting new elements into the array. This is an example of a tradeoff between the advantage of binary search during search and the disadvantage related to maintaining a sorted array. Only in the context of the complete problem to be solved can we know whether the advantage outweighs the disadvantage.

2.8.2. Summary Exercise

02.09 Analyzing Problems

Due No Due Date **Points** 1 **Submitting** an external tool

02.09 Analyzing Problems

2.9. Analyzing Problems

2.9.1. Analyzing Problems

You most often use the techniques of “algorithm” analysis to analyze an **algorithm**, or the instantiation of an algorithm as a **program**. You can also use these same techniques to analyze the cost of a **problem**. The key question that we want to ask is: How hard is a problem? Certainly we should expect that in some sense, the problem of sorting a list of records is harder than the problem of searching a list of records for a given key value. Certainly the algorithms that we know for sorting some records seem to be more expensive than the algorithms that we know for searching those same records.

What we need are useful definitions for the **upper bound** and **lower bound** of a problem.

One might start by thinking that the upper bound for a problem is how hard any algorithm can be for the problem. But we can make algorithms as bad as we want, so that is not useful. Instead, what is useful is to say that a problem is only as hard as what we CAN do. In other words, we should define the upper bound for a problem to be the **best** algorithm that we know for the problem. Of course, whenever we talk about bounds, we have to say when they apply. We we really should say something like the best algorithm that we know in the worst case, or the best algorithm that we know in the average case.

But what does it mean to give a lower bound for a problem? Lower bound refers to the minimum that any algorithm MUST cost. For example, when searching an unsorted list, we MUST look at every record. When sorting a list, we MUST look at every record (to even know if it is sorted).

It is much easier to show that an algorithm (or program) is in $\Omega(f(n))$ than it is to show that a problem is in $\Omega(f(n))$. For a problem to be in $\Omega(f(n))$ means that *every* algorithm that solves the problem is in $\Omega(f(n))$, even algorithms that we have not thought of! In other words, EVERY algorithm MUST have at least this cost. So, to prove a lower bound, we need an argument that is true, even for algorithms that we don’t know.

So far all of our examples of algorithm analysis give “obvious” results, with big-Oh always matching Ω. To understand how big-Oh, Ω, and Θ notations are properly used to describe our understanding of a problem or an algorithm, it is best to consider an example where you do not already know a lot about the problem.

Let us look ahead to analyzing the problem of sorting to see how this process works. What is the least possible cost for any sorting algorithm in the worst case? The algorithm must at least look at every element in the input, just to determine that the input is truly sorted. Thus, any sorting algorithm must take at least cn time. For many problems, this observation that each of the n inputs must be looked at leads to an easy $\Omega(n)$ lower bound.

In your previous study of computer science, you have probably seen an example of a sorting algorithm whose running time is in $O(n^2)$ in the worst case. The simple Bubble Sort and Insertion Sort algorithms typically given as examples in a first year programming course have worst case running times in $O(n^2)$. Thus, the problem of sorting can be said to have an upper bound in $O(n^2)$. How do we close the gap between $\Omega(n)$ and $O(n^2)$? Can there be a better sorting algorithm? If you can think of no algorithm whose worst-case growth rate is better than $O(n^2)$, and if you have discovered no analysis technique to show that the least cost for the problem of sorting in the worst case is greater than $\Omega(n)$, then you cannot know for sure whether or not there is a better algorithm.

Many good sorting algorithms have running time that is in $O(n \log n)$ in the worst case. This greatly narrows the gap. With this new knowledge, we now have a lower bound in $\Omega(n)$ and an upper bound in $O(n \log n)$. Should we search for a faster algorithm? Many have tried, without success. Fortunately (or perhaps unfortunately?), **we can prove that** any sorting algorithm must have running time in $\Omega(n \log n)$ in the worst case. **1** This proof is one of the most important results in the field of algorithm analysis, and it means that no sorting algorithm can possibly run faster than $cn \log n$ for the worst-case input of size n. Thus, we can conclude that the problem of sorting is $\Theta(n \log n)$ in the worst case, because the upper and lower bounds have met.

Knowing the lower bound for a problem does not give you a good algorithm. But it does help you to know when to stop looking. If the lower bound for the problem matches the upper bound for the algorithm (within a constant factor), then we know that we can find an algorithm that is better only by a constant factor.

So, to summarize: The upper bound for a problem is the best that you CAN do, while the lower bound for a problem is the least work that you MUST do. If those two are the same, then we say that we really understand our problem.

1

While it is fortunate to know the truth, it is unfortunate that sorting is $\Theta(n \log n)$ rather than $\Theta(n)$.

2.10. Recurrence Relations

2.10.1. Recurrence Relations

The running time for a recursive algorithm is most easily expressed by a recursive expression because the total time for the recursive algorithm includes the time to run the recursive call(s). A **recurrence relation** defines a function by means of an expression that includes one or more (smaller) instances of itself. A classic example is the recursive definition for the factorial function:

$$n! = (n-1)! \cdot n \text{ for } n > 1; \quad 1! = 0! = 1.$$

Another standard example of a recurrence is the Fibonacci sequence:

$$\mathrm{Fib}(n) = \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) \text{ for } n > 2; \quad \mathrm{Fib}(1) = \mathrm{Fib}(2) = 1.$$

From this definition, the first seven numbers of the Fibonacci sequence are

$$1, 1, 2, 3, 5, 8, \text{ and } 13.$$

Notice that this definition contains two parts: the general definition for $\mathrm{Fib}(n)$ and the base cases for $\mathrm{Fib}(1)$ and $\mathrm{Fib}(2)$. Likewise, the definition for factorial contains a recursive part and base cases.

Recurrence relations are often used to model the cost of recursive functions. For example, the number of multiplications required by a recursive version of the factorial function for an input of size n will be zero when $n = 0$ or $n = 1$ (the base cases), and it will be one plus the cost of calling `fact` on a value of $n - 1$. This can be defined using the following recurrence:

$$\mathbf{T}(n) = \mathbf{T}(n-1) + 1 \text{ for } n > 1; \quad \mathbf{T}(0) = \mathbf{T}(1) = 0.$$

As with summations, we typically wish to replace the recurrence relation with a closed-form solution. One approach is to expand the recurrence by replacing any occurrences of $\mathbf{T}$ on the right-hand side with its definition.

1 / 8 &lt;&lt; &lt; &gt; &gt;&gt;

We will use expansion to guess the closed form solution for the r
$\mathbf{T}(n) = \mathbf{T}(n-1) + 1 \text{ for } n > 1; \mathbf{T}(0) = \mathbf{T}(1) = 0.$

A slightly more complicated recurrence is

$$\mathbf{T}(n) = \mathbf{T}(n-1) + n; \quad \mathbf{T}(1) = 1.$$

Again, we will use expansion to help us find a closed form solution.

1 / 8

<< < > >>

We will use expansion to guess the closed form solution for the recurrence $\mathbf{T}(n) = \mathbf{T}(n-1) + n \text{ for } n > 1; \mathbf{T}(1)$

02.11 Common Misunderstandings

Due No Due Date **Points** 1 **Submitting** an external tool

02.11 Common Misunderstandings

2.11. Common Misunderstandings

2.11.1. Common Misunderstandings

Asymptotic analysis is one of the most intellectually difficult topics that undergraduate computer science majors are confronted with. Most people find **growth rates** and asymptotic analysis confusing and so develop misconceptions about either the concepts or the terminology. It helps to know what the standard points of confusion are, in hopes of avoiding them.

One problem with differentiating the concepts of **upper** and **lower bounds** is that, for most algorithms that you will encounter, it is easy to recognize the true growth rate for that algorithm. Given complete knowledge about a cost function, the upper and lower bound for that cost function are always the same. Thus, the distinction between an upper and a lower bound is only worthwhile when you have incomplete knowledge about the thing being measured. If this distinction is still not clear, then you should **read about analyzing problems**. We use Θ-notation to indicate that there is no meaningful difference between what we know about the growth rates of the upper and lower bound (which is usually the case for simple algorithms).

It is a common mistake to confuse the concepts of upper bound or lower bound on the one hand, and **worst case** or **best case** on the other. The best, worst, or **average cases** each **define a cost** for a specific input instance (or specific set of instances for the average case). In contrast, upper and lower bounds describe our understanding of the **growth rate** for that cost measure. So to define the growth rate for an algorithm or problem, we need to determine what we are measuring (the best, worst, or average case) and also our description for what we know about the growth rate of that cost measure (big-Oh, Ω, or Θ).

The upper bound for an algorithm is not the same as the worst case for that algorithm for a given input of size n. What is being bounded is not the actual cost (which you can determine for a given value of n), but rather the **growth rate** for the cost. There cannot be a growth rate for a single point, such as a particular value of n. The growth **rate** applies to the **change** in cost as a **change** in input size occurs. Likewise, the lower bound is not the same as the best case for a given size n.

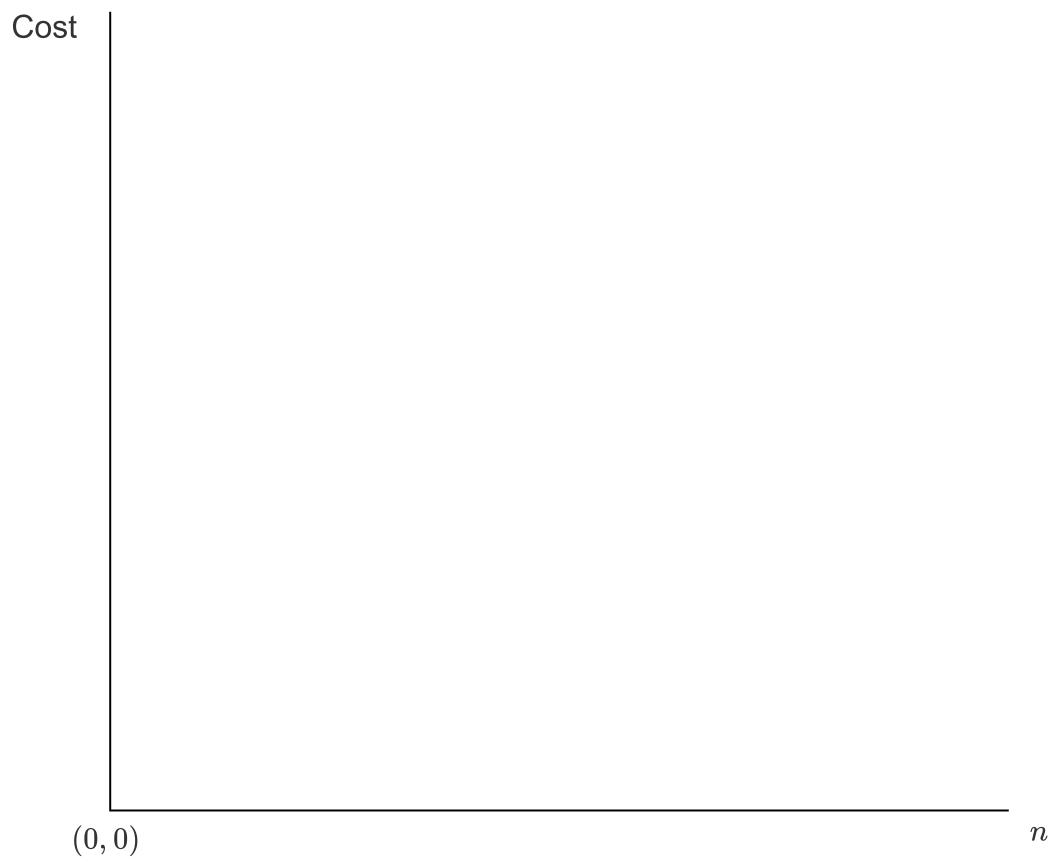
Another common misconception is thinking that the best case for an algorithm occurs when the input size is as small as possible, or that the worst case occurs when the input size is as large as possible. What is correct is that best- and worse-case instances exist for each possible size of input. That is, for all inputs of a given size, say i, one (or more) of the inputs of size i is the best and one (or more) of the inputs of size i is the worst. Often (but not always!), we can characterize the best input case for an arbitrary size, and we can characterize the worst input case for an arbitrary size. Ideally, we can determine the growth rate for the characterized best, worst, and average cases for an arbitrary size. Ideally, we can determine the growth rate for the characterized best, worst, and average cases as the input size grows.

Example 2.11.1

What is the growth rate of the best case for sequential search? For any array of size n, the best case occurs when the value we are looking for appears in the first position of the array. This is true regardless of the size of the array. Thus, the best case (for arbitrary size n) occurs when the desired value is in the first of n positions, and its cost is 1. It is *not* correct to say that the best case occurs when $n = 1$.

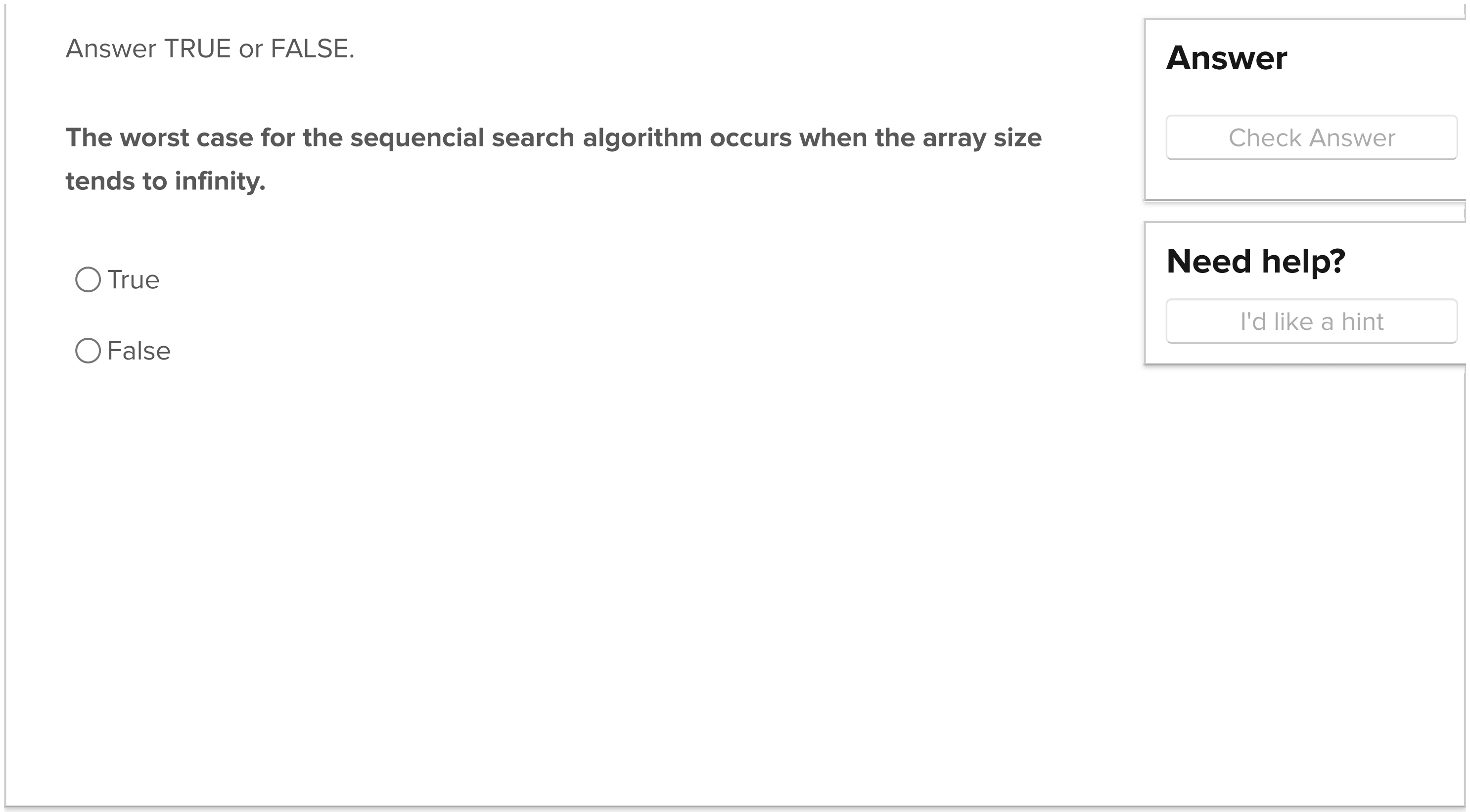
1 / 16 &lt;&lt; &lt; &gt; &gt;&gt;

Imagine drawing a graph to show the cost of finding the maximum value among n values, as n grows. That is, would be n, and the y value would be the cost for a given value of n.



Practicing Common Misunderstandings exercise

Current score: 0 out of 5



Answer TRUE or FALSE.

The worst case for the sequencial search algorithm occurs when the array size tends to infinity.

◯ True

◯ False

2.12. Multiple Parameters

Sometimes the proper analysis for an algorithm requires multiple parameters to describe the cost. To illustrate the concept, consider an algorithm to compute the rank ordering for counts of all pixel values in a picture. Pictures are often represented by a two-dimensional array, and a pixel is one cell in the array. The value of a pixel is either the code value for the color, or a value for the intensity of the picture at that pixel. Assume that each pixel can take any integer value in the range 0 to $C-1$. The problem is to find the number of pixels of each color value and then sort the color values with respect to the number of times each value appears in the picture. Assume that the picture is a rectangle with P pixels. A pseudocode algorithm to solve the problem follows.

Toggle Tree View

```
for (i=0; i<C; i++) {   // Initialize count
   count[i] = 0;
}
for (i=0; i<P; i++) {   // Look at all of the pixels
   count[value(i)]++; // Increment a pixel value count
}
sort(count);          // Sort pixel value counts
```

In this example, `count` is an array of size `C` that stores the number of pixels for each color value. Function `value(i)` returns the color value for pixel i.

The time for the first `for` loop (which initializes `count`) is based on the number of colors, C. The time for the second loop (which determines the number of pixels with each color) is $\Theta(P)$. The time for the final line, the call to `sort`, depends on the cost of the sorting algorithm used. We will assume that the sorting algorithm has cost $\Theta(P \log P)$ if P items are sorted, thus yielding $\Theta(P \log P)$ as the total algorithm cost.

Is this a good representation for the cost of this algorithm? What is actually being sorted? It is not the pixels, but rather the colors. What if C is much smaller than P? Then the estimate of $\Theta(P \log P)$ is pessimistic, because much fewer than P items are being sorted. Instead, we should use P as our analysis variable for steps that look at each pixel, and C as our analysis variable for steps that look at colors. Then we get $\Theta(C)$ for the initialization loop, $\Theta(P)$ for the pixel count loop, and $\Theta(C \log C)$ for the sorting operation. This yields a total cost of $\Theta(P + C \log C)$.

Why can we not simply use the value of C for input size and say that the cost of the algorithm is $\Theta(C \log C)$? Because, C is typically much less than P. For example, a picture might have 1000 $\times$ 1000 pixels and a range of 256 possible colors. So, P is one million, which is much larger than $C \log C$. But, if P is smaller, or C larger (even if it is still less than P), then $C \log C$ can become the larger quantity. Thus, neither variable should be ignored.

2.13. Space Bounds

Besides time, space is the other computing resource that is commonly of concern to programmers. Just as computers have become much faster over the years, they have also received greater allotments of memory. Even so, the amount of available disk space or main memory can be significant constraints for algorithm designers.

The analysis techniques used to measure space requirements are similar to those used to measure time requirements. However, while time requirements are normally measured for an algorithm that manipulates a particular data structure, space requirements are normally determined for the data structure itself. The concepts of asymptotic analysis for growth rates on input size apply completely to measuring space requirements.

Example 2.13.1

What are the space requirements for an array of n integers? If each integer requires c bytes, then the array requires cn bytes, which is $\Theta(n)$.

Example 2.13.2

Imagine that we want to keep track of friendships between n people. We can do this with an array of size $n \times n$. Each row of the array represents the friends of an individual, with the columns indicating who has that individual as a friend. For example, if person j is a friend of person i, then we place a mark in column j of row i in the array. Likewise, we should also place a mark in column i of row j if we assume that friendship works both ways. For n people, the total size of the array is $\Theta(n^2)$.

A data structure's primary purpose is to store data in a way that allows efficient access to those data. To provide efficient access, it may be necessary to store additional information about where the data are within the data structure. For example, each node of a linked list must store a pointer to the next value on the list. All such information stored in addition to the actual data values is referred to as **overhead**. Ideally, overhead should be kept to a minimum while allowing maximum access. The need to maintain a balance between these opposing goals is what makes the study of data structures so interesting.

One important aspect of algorithm design is referred to as the **space/time tradeoff** principle. The space/time tradeoff principle says that one can often achieve a reduction in time if one is willing to sacrifice space or vice versa. Many programs can be modified to reduce storage requirements by "packing" or encoding information. "Unpacking" or decoding the information requires additional time. Thus, the resulting program uses less space but runs slower. Conversely, many programs can be modified to pre-store results or reorganize information to allow faster running time at the expense of greater storage requirements. Typically, such changes in time and space are both by a constant factor.

A classic example of a space/time tradeoff is the **lookup table**. A lookup table pre-stores the value of a function that would otherwise be computed each time it is needed. For example, 12! is the greatest value for the factorial function that can be stored in a 32-bit `int` variable. If you are writing a program that often computes factorials, it is likely to be much more time efficient to simply pre-compute and store the 12 values in a table. Whenever the program needs the value of $n!$ it can simply check the lookup table. (If $n > 12$, the value is too large to store as an `int` variable anyway.) Compared to the time required to compute factorials, it may be well worth the small amount of additional space needed to store the lookup table.

Lookup tables can also store approximations for an expensive function such as sine or cosine. If you compute this function only for exact degrees or are willing to approximate the answer with the value for the nearest degree, then a lookup table storing the computation for exact degrees can be used instead of repeatedly computing the sine function. Note that initially building the lookup table requires a certain amount of time. Your application must use the lookup table often enough to make this initialization worthwhile.

Another example of the space/time tradeoff is typical of what a programmer might encounter when trying to optimize space. Here is a simple code fragment for sorting an array of integers. We assume that this is a special case where there are n integers whose values are a permutation of the integers from 0 to $n-1$. This is an example of a **binsort**. Binsort assigns each value to an array position corresponding to its value.

Java | Java (Generic) | Toggle Tree View

```
for (i=0; i<A.length; i++)
  B[A[i]] = A[i];
```

This is efficient and requires $\Theta(n)$ time. However, it also requires two arrays of size n. Next is a code fragment that places the permutation in order but does so within the same array (thus it is an example of an “in place” sort).

Java | Java (Generic) | Toggle Tree View

```
for (i=0; i<A.length; i++)
  while (A[i] != i) // Swap element A[i] with A[A[i]]
    Swap.swap(A, i, A[i]);
```

Function `swap(A, i, j)` exchanges elements `i` and `j` in array `A`. It may not be obvious that the second code fragment actually sorts the array. To see that this does work, notice that each pass through the `for` loop will at least move the integer with value i to its correct position in the array, and that during this iteration, the value of `A[i]` must be greater than or equal to i. A total of at most n `swap` operations take place, because an integer cannot be moved out of its correct position once it has been placed there, and each swap operation places at least one integer in its correct position. Thus, this code fragment has cost $\Theta(n)$. However, it requires more time to run than the first code fragment. On my computer the second version takes nearly twice as long to run as the first, but it only requires half

2.14. Code Tuning and Empirical Analysis

2.14.1. Code Tuning and Empirical Analysis

In practice, there is not such a big difference in running time between an algorithm with growth rate $\Theta(n)$ and another with growth rate $\Theta(n \log n)$. There is, however, an enormous difference in running time between algorithms with growth rates of $\Theta(n \log n)$ and $\Theta(n^2)$. As you shall see during the course of your study of common data structures and algorithms, there are many problems whose obvious solution requires $\Theta(n^2)$ time, but that also have a solution requiring $\Theta(n \log n)$ time. Examples include sorting and searching, two of the most important computer problems.

While not nearly so important as changing an algorithm to reduce its growth rate, "code tuning" can also lead to dramatic improvements in running time. Code tuning is the art of hand-optimizing a program to run faster or require less storage. For many programs, code tuning can reduce running time or cut the storage requirements by a factor of two or more. Even speedups by a factor of five to ten are not uncommon. Occasionally, you can get an even bigger speedup by converting from a symbolic representation of the data to a numeric coding scheme on which you can do direct computation.

Here are some suggestions for ways to speed up your programs by code tuning. The most important thing to realize is that most statements in a program do not have much effect on the running time of that program. There are normally just a few key subroutines, possibly even key lines of code within the key subroutines, that account for most of the running time. There is little point to cutting in half the running time of a subroutine that accounts for only 1% of the total running time. Focus your attention on those parts of the program that have the most impact.

When tuning code, it is important to gather good timing statistics. Many compilers and operating systems include profilers and other special tools to help gather information on both time and space use. These are invaluable when trying to make a program more efficient, because they can tell you where to invest your effort.

A lot of code tuning is based on the principle of avoiding work rather than speeding up work. A common situation occurs when we can test for a condition that lets us skip some work. However, such a test is never completely free. Care must be taken that the cost of the test does not exceed the amount of work saved. While one test might be cheaper than the work potentially saved, the test must always be made and the work can be avoided only some fraction of the time.

Example 2.14.1

A common operation in computer graphics applications is to find which among a set of complex objects contains a given point in space. Many useful data structures and algorithms have been developed to deal with variations of this problem. Most such implementations involve the following tuning step. Directly testing whether a given complex object contains the point in question is relatively expensive. Instead, we can screen for whether the point is contained within a **bounding box** for the object. The bounding box is simply the smallest rectangle (usually defined to have sides perpendicular to the x and y axes) that contains the object. If the point is not in the bounding box, then it cannot be in the object. If the point is in the bounding box, only then would we conduct the full comparison of the object versus the point. Note that if the point is outside the bounding box, we saved time because the bounding box test is cheaper than the comparison of the full object versus the point. But if the point is inside the bounding box, then that test is redundant because we still have to compare the point against the object. Typically the amount of work avoided by making this test is greater than the cost of making the test on every object.

Be careful not to use tricks that make the program unreadable. Most code tuning is simply cleaning up a carelessly written program, not taking a clear program and adding tricks. In particular, you should develop an appreciation for the capabilities of modern compilers to make extremely good optimizations of expressions. “Optimization of expressions” here means a rearrangement of arithmetic or logical expressions to run more efficiently. Be careful not to damage the compiler’s ability to do such optimizations for you in an effort to optimize the expression yourself. Always check that your “optimizations” really do improve the program by running the program before and after the change on a suitable benchmark set of input. Many times I have been wrong about the positive effects of code tuning in my own programs. Most often I am wrong when I try to optimize an expression. It is hard to do better than the compiler.

The greatest time and space improvements come from a better data structure or algorithm. The most important rule of code tuning is:

> **First tune the algorithm, then tune the code.**

2.14.1.1. Empirical Analysis

Asymptotic algorithm analysis is an analytic tool, whereby we model the key aspects of an algorithm to determine the growth rate of the algorithm as the input size grows. It has proved hugely practical, guiding developers to use more efficient algorithms. But it is really an **estimation** technique, and it has its limitations. These include the effects at small problem size, determining the finer distinctions between algorithms with the same growth rate, and the inherent difficulty of doing mathematical modeling for more complex problems.

An alternative to analytical approaches are empirical ones. The most obvious empirical approach is simply to run two competitors and see which performs better. In this way we might overcome the deficiencies of analytical approaches.

Be warned that comparative timing of programs is a difficult business, often subject to experimental errors arising from uncontrolled factors (system load, the language or compiler used, etc.). The most important concern is that you might be biased in favor of one of the programs. If you are biased, this is certain to be reflected in the timings. One look at competing software or hardware vendors’ advertisements should convince you of this. The most common pitfall when writing two programs to compare their performance is that one receives more code-tuning effort than the other, since code tuning can often reduce running time by a factor of five to ten. If the running times for two programs differ by a constant factor regardless of input size (i.e., their growth rates are the same), then differences in code tuning might account for any difference in running time. Be suspicious of empirical comparisons in this situation.

Another approach to analytical analysis is simulation. The idea of simulation is to model the problem with a computer program and then run it to get a result. In the context of algorithm analysis, simulation is distinct from empirical comparison of two competitors because the purpose of the simulation is to perform analysis that might otherwise be too difficult. A good example of this appears in the following figure.

4
Insert
Delete

02.15 Algorithm Analysis Summary Exercises

Due No Due Date **Points** 1 **Submitting** an external tool

02.15 Algorithm Analysis Summary Exercises

2.15. Algorithm Analysis Summary Exercises

2.15.1. Summary Exercise: CS2

Practicing Algorithm Analysis Summary Questions

Current score: 0 out of 5

Determine Θ for the following code fragment in the average case. Assume that all variables are of type "int".

```
for (i = 0; i < n - 1; i++)
  for (j = i + 1; j < n; j++) {
    tmp = AA[i][j];
    AA[i][j] = AA[j][i];
    AA[j][i] = tmp;
  }
```

Answer

Check Answer

Need help?

I'd like a hint

○ $\Theta(n^2)$

○ $\Theta(\log n)$

○ $\Theta(n^3)$

○ $\Theta(n)$

○ $\Theta(n^2 \log n)$

○ $\Theta(n \log n)$

○ $\Theta(2^n)$

○ $\Theta(1)$

02.16 Algorithm Analysis Summary Exercises

Due No Due Date **Points** 1 **Submitting** an external tool

02.16 Algorithm Analysis Summary Exercises

2.16. Algorithm Analysis Summary Exercises

2.16.1. Summary Exercise: CS3

Practicing Algorithm Analysis Summary Questions

Current score: 0 out of 5

Suppose that a particular algorithm has time complexity $T(n) = 8n$ and that executing an implementation of it on a particular machine takes t seconds for n inputs. Now suppose that we are presented with a machine that is 64 times as fast. How many inputs could we process on the new machine in t seconds?

○ n^2

○ $8n^2$

○ $8n$

○ $64n^2$

○ 2^n

○ 8

○ 64

○ $64n$

Answer

Check Answer

Need help?

I'd like a hint

Chapter 3: Generics

The Java™ Tutorials

Trail: Learning the Java Language
Lesson: Generics (Updated)

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Why Use Generics?

In a nutshell, generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar *formal parameters* used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Code that uses generics has many benefits over non-generic code:

- Stronger type checks at compile time.
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

- Elimination of casts.
  The following code snippet without generics requires casting:

  ```
  List list = new ArrayList();
  list.add("hello");
  String s = (String) list.get(0);
  ```

  When re-written to use generics, the code does not require casting:

  ```
  List<String> list = new ArrayList<String>();
  list.add("hello");
  String s = list.get(0);   // no cast
  ```

- Enabling programmers to implement generic algorithms.
  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Previous page: Generics (Updated)
Next page: Generic Types

The Java™ Tutorials

Trail: Learning the Java Language
Lesson: Generics (Updated)

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Generic Types

A *generic type* is a generic class or interface that is parameterized over types. The following `Box` class will be modified to demonstrate the concept.

A Simple Box Class

Begin by examining a non-generic `Box` class that operates on objects of any type. It needs only to provide two methods: `set`, which adds an object to the box, and `get`, which retrieves it:

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Since its methods accept or return an `Object`, you are free to pass in whatever you want, provided that it is not one of the primitive types. There is no way to verify, at compile time, how the class is used. One part of the code may place an `Integer` in the box and expect to get `Integer`s out of it, while another part of the code may mistakenly pass in a `String`, resulting in a runtime error.

A Generic Version of the Box Class

A *generic class* is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

The type parameter section, delimited by angle brackets (`<>`), follows the class name. It specifies the *type parameters* (also called *type variables*) `T1`, `T2`, ..., and `Tn`.

To update the `Box` class to use generics, you create a *generic type declaration* by changing the code "`public class Box`" to "`public class Box<T>`". This introduces the type variable, `T`, that can be used anywhere inside the class.

With this change, the `Box` class becomes:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

As you can see, all occurrences of `Object` are replaced by `T`. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

You'll see these names used throughout the Java SE API and the rest of this lesson.

Invoking and Instantiating a Generic Type

To reference the generic `Box` class from within your code, you must perform a *generic type invocation*, which replaces `T` with some concrete value, such as `Integer`:

```
Box<Integer> integerBox;
```

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a *type argument* — `Integer` in this case — to the `Box` class itself.

Type Parameter and Type Argument Terminology: Many developers use the terms "type parameter" and "type argument" interchangeably, but these terms are not the same. When coding, one provides type arguments in order to create a parameterized type. Therefore, the `T` in `Foo<T>` is a type parameter and the `String` in `Foo<String> f` is a type argument. This lesson observes this definition when using these terms.

Like any other variable declaration, this code does not actually create a new `Box` object. It simply declares that `integerBox` will hold a reference to a "`Box` of `Integer`", which is how `Box<Integer>` is read.

An invocation of a generic type is generally known as a *parameterized type*.

To instantiate this class, use the `new` keyword, as usual, but place `<Integer>` between the class name and the parenthesis:

```
Box<Integer> integerBox = new Box<Integer>();
```

The Diamond

In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called *the diamond*. For example, you can create an instance of `Box<Integer>` with the following statement:

```
Box<Integer> integerBox = new Box<>();
```

For more information on diamond notation and type inference, see Type Inference.

Multiple Type Parameters

As mentioned previously, a generic class can have multiple type parameters. For example, the generic `OrderedPair` class, which implements the generic `Pair` interface:

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;
    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

The following statements create two instantiations of the `OrderedPair` class:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

The code, `new OrderedPair<String, Integer>`, instantiates `K` as a `String` and `V` as an `Integer`. Therefore, the parameter types of `OrderedPair`'s constructor are `String` and `Integer`, respectively. Due to autoboxing, it is valid to pass a `String` and an `int` to the class.

As mentioned in The Diamond, because a Java compiler can infer the `K` and `V` types from the declaration `OrderedPair<String, Integer>`, these statements can be shortened using diamond notation:

```
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
```

To create a generic interface, follow the same conventions as for creating a generic class.

Parameterized Types

You can also substitute a type parameter (that is, `K` or `V`) with a parameterized type (that is, `List<String>`). For example, using the `OrderedPair<K, V>` example:

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```

Previous page: Why Use Generics?
Next page: Raw Types

The Java™ Tutorials

Trail: Learning the Java Language
Lesson: Generics (Updated)
Section: Generic Types

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Raw Types

A *raw type* is the name of a generic class or interface without any type arguments. For example, given the generic `Box` class:

```
public class Box<T> {
    public void set(T t) { /* ... */ }
    // ...
}
```

To create a parameterized type of `Box<T>`, you supply an actual type argument for the formal type parameter `T`:

```
Box<Integer> intBox = new Box<>();
```

If the actual type argument is omitted, you create a raw type of `Box<T>`:

```
Box rawBox = new Box();
```

Therefore, `Box` is the raw type of the generic type `Box<T>`. However, a non-generic class or interface type is *not* a raw type.

Raw types show up in legacy code because lots of API classes (such as the `Collections` classes) were not generic prior to JDK 5.0. When using raw types, you essentially get pre-generics behavior — a `Box` gives you `Object`s. For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;               // OK
```

But if you assign a raw type to a parameterized type, you get a warning:

```
Box rawBox = new Box();           // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;     // warning: unchecked conversion
```

You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8);  // warning: unchecked invocation to set(T)
```

The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

The Type Erasure section has more information on how the Java compiler uses raw types.

Unchecked Error Messages

As mentioned previously, when mixing legacy code with generic code, you may encounter warning messages similar to the following:

```
Note: Example.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

This can happen when using an older API that operates on raw types, as shown in the following example:

```
public class WarningDemo {
    public static void main(String[] args){
        Box<Integer> bi;
        bi = createBox();
    }

    static Box createBox(){
        return new Box();
    }
}
```

The term "unchecked" means that the compiler does not have enough type information to perform all type checks necessary to ensure type safety. The "unchecked" warning is disabled, by default, though the compiler gives a hint. To see all "unchecked" warnings, recompile with `-Xlint:unchecked`.

Recompiling the previous example with `-Xlint:unchecked` reveals the following additional information:

```
WarningDemo.java:4: warning: [unchecked] unchecked conversion
found   : Box
required: Box<java.lang.Integer>
        bi = createBox();
                      ^
1 warning
```

To completely disable unchecked warnings, use the `-Xlint:-unchecked` flag. The `@SuppressWarnings("unchecked")` annotation suppresses unchecked warnings. If you are unfamiliar with the `@SuppressWarnings` syntax, see Annotations.

Previous page: Generic Types
Next page: Generic Methods

Trail: Learning the Java Language
Lesson: Generics (Updated)

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Generic Methods

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

The syntax for a generic method includes a list of type parameters, inside angle brackets, which appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

The `Util` class includes a generic method, `compare`, which compares two `Pair` objects:

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}

public class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

The complete syntax for invoking this method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

This feature, known as *type inference*, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets. This topic is further discussed in the following section, Type Inference.

Previous page: Raw Types
Next page: Bounded Type Parameters

Chapter 4: Linear Structures

4.1. Chapter Introduction: Lists

If your program needs to store a few things—numbers, payroll records, or job descriptions for example—the simplest and most effective approach might be to put them in a list. Only when you have to organize and search through a large number of things do more sophisticated data structures like **search trees** become necessary. Many applications don't require any form of search, and they do not require that an ordering be placed on the objects being stored. Some applications require that actions be performed in a strict chronological order, processing objects in the order that they arrived, or perhaps processing objects in the reverse of the order that they arrived. For all these situations, a simple list structure is appropriate.

This chapter describes representations both for lists and for two important list-like structures called the **stack** and the **queue**. Along with presenting these fundamental data structures, the other goals of the chapter are to:

1. Give examples that show the separation of a logical representation in the form of an ADT from a physical implementation as a data structure.
2. Illustrate the use of asymptotic analysis in the context of simple operations that you might already be familiar with. In this way you can begin to see how asymptotic analysis works, without the complications that arise when analyzing more sophisticated algorithms and data structures.

We begin by defining an **ADT for lists**. Two implementations for the list ADT—the **array-based list** and the **linked list**—are covered in detail and their relative merits discussed. The chapter finishes with implementations for **stacks** and **queues**.

04.02 The List ADT

Due No Due Date **Points** 2 **Submitting** an external tool

04.02 The List ADT

4.2. The List ADT

4.2.1. The List ADT

We all have an intuitive understanding of what we mean by a “list”. We want to turn this intuitive understanding into a concrete data structure with implementations for its operations. The most important concept related to lists is that of **position**. In other words, we perceive that there is a first element in the list, a second element, and so on. So, define a **list** to be a finite, ordered sequence of data items known as **elements**. This is close to the mathematical concept of a **sequence**.

“Ordered” in this definition means that each element has a position in the list. So the term “ordered” in this context does **not** mean that the list elements are sorted by value. (Of course, we can always choose to sort the elements on the list if we want; it’s just that keeping the elements sorted is not an inherent property of being a list.)

Each list element must have some data type. In the simple list implementations discussed in this chapter, all elements of the list are usually assumed to have the same data type, although there is no conceptual objection to lists whose elements have differing data types if the application requires it. The operations defined as part of the list **ADT** do not depend on the elemental **data type**. For example, the list ADT can be used for lists of integers, lists of characters, lists of payroll records, even lists of lists.

A list is said to be **empty** when it contains no elements. The number of elements currently stored is called the **length** of the list. The beginning of the list is called the **head**, the end of the list is called the **tail**.

We need some notation to show the contents of a list, so we will use the same angle bracket notation that is normally used to represent **sequences**. To be consistent with standard array indexing, the first position on the list is denoted as 0. Thus, if there are n elements in the list, they are given positions 0 through $n-1$ as $\langle\, a_0,\, a_1,\, \ldots,\, a_{n-1}\, \rangle$. The subscript indicates an element’s position within the list. Using this notation, the empty list would appear as $\langle\,\rangle$.

4.2.1.1. Defining the ADT

What basic operations do we want our lists to support? Our common intuition about lists tells us that a list should be able to grow and shrink in size as we insert and remove elements. We should be able to insert and remove elements from anywhere in the list. We should be able to gain access to any element’s value, either to read it or to change it. We must be able to create and clear (or reinitialize) lists. It is also convenient to access the next or previous element from the “current” one.

Now we can define the ADT for a list object in terms of a set of operations on that object. We will use an interface to formally define the list ADT. `List` defines the member functions that any list implementation inheriting from it must support, along with their parameters and return types.

True to the notion of an ADT, an interface does not specify how operations are implemented. Two complete implementations are presented later in later modules, both of which use the same list ADT to define their operations. But they are considerably different in approaches and in their space/time tradeoffs.

The code below presents our list ADT. Any implementation for a **container class** such as a list should be able to support different data types for the elements. One way to do this in Java is to store data values of type `Object`. Languages that support generics (Java) or templates (C++) give more control over the element types.

The comments given with each member function describe what it is intended to do. However, an explanation of the basic design should help make this clearer. Given that we wish to support the concept of a sequence, with access to any position in the list, the need for many of the member functions such as `insert` and `moveToPos` is clear. The key design decision embodied in this ADT is support for the concept of a **current position**. For example, member `moveToStart` sets the current position to be the first element on the list, while methods `next` and `prev` move the current position to the next and previous elements, respectively. The intention is that any implementation for this ADT support the concept of a current position. The current position is where any action such as insertion or deletion will take place. An alternative design is to factor out position as a separate position object, sometimes referred to as an **iterator**.

Java | **Java (Generic)** | Toggle Tree Vie

```java
// List class ADT. Generalize by using "Object" for the element type.
public interface List { // List class ADT
  // Remove all contents from the list, so it is once again empty
  public void clear();

  // Insert "it" at the current location
  // The client must ensure that the list's capacity is not exceeded
  public boolean insert(Object it);

  // Append "it" at the end of the list
  // The client must ensure that the list's capacity is not exceeded
  public boolean append(Object it);

  // Remove and return the current element
  public Object remove() throws NoSuchElementException;

  // Set the current position to the start of the list
  public void moveToStart();

  // Set the current position to the end of the list
  public void moveToEnd();

  // Move the current position one step left, no change if already at beginning
  public void prev();
  // Move the current position one step right, no change if already at end
  public void next();

  // Return the number of elements in the list
  public int length();

  // Return the position of the current element
  public int currPos();

  // Set the current position to "pos"
  public boolean moveToPos(int pos);

  // Return true if current position is at end of the list
  public boolean isAtEnd();

  // Return the current element
  public Object getValue() throws NoSuchElementException;

  public boolean isEmpty();
}
```

1 / 9

<< < > >>

Since insertions take place at the current position, and since we want to be able to insert to the front or the b
list as well as anywhere in between, there are actually $n+1$ possible "current positions" when there are n
in the list.

The `List` member functions allow you to build a list with elements in any desired order, and to access any desired position in the list. You might notice that the `clear` method is a “convenience” method, since it could be implemented by means of the other member functions in the same asymptotic time.

A list can be iterated through as follows:

Java | Java (Generic) | Toggle Tree Vie

```
for (L.moveToStart(); !L.isAtEnd(); L.next()) {
  it = L.getValue();
  doSomething(it);
}
```

In this example, each element of the list in turn is stored in `it`, and passed to the `doSomething` function. The loop terminates when the current position reaches the end of the list.

The list class declaration presented here is just one of many possible interpretations for lists. Our list interface provides most of the operations that one naturally expects to perform on lists and serves to illustrate the issues relevant to implementing the list data structure. As an example of using the list ADT, here is a function to return `true` if there is an occurrence of a given integer in the list, and `false` otherwise. The `find` method needs no knowledge about the specific list implementation, just the list ADT.

Java | **Java (Generic)** | Toggle Tree Vie

```
// Return true if k is in list L, false otherwise
static boolean find(List L, Object k) {
  for (L.moveToStart(); !L.isAtEnd(); L.next())
    if (k == L.getValue()) return true; // Found k
  return false;                         // k not found
}
```

In languages that support it, this implementation for `find` could be rewritten as a generic or template with respect to the element type. While making it more flexible, even generic types still are limited in their ability to handle different data types stored on the list. In particular, for the `find` function generic types would only work when the description for the object being searched for (`k` in the function) is of the same type as the objects themselves. They also have to be comparable when using the `==` operator. A more realistic situation is that we are searching for a record that contains a **key** field whose value matches `k`. Similar functions to find and return a **composite type** based on a key value can be created using the list implementation, but to do so requires some agreement between the list ADT and the `find` function on the concept of a key, and on **how keys may be compared**.

There are two standard approaches to implementing lists, the **array-based list**, and the **linked list**.

4.2.2. List ADT Programming Exercise

X278: ListADT

Use appropriate method calls from the List ADT to create the following list:

< 4 19 | 23 30 >

You should assume that L is passed to the function as an empty list.

Your Answer:

```
public List buildList(List L)
{

}

```

Feedback

Your feedback will a

answer.

Check my answer! Reset

04.03 Array-Based List Implementation

Due No Due Date **Points** 3 **Submitting** an external tool

04.03 Array-Based List Implementation

4.3. Array-Based List Implementation

4.3.1. Array-Based List Implementation ¶

Here is an implementation for the array-based list, named `AList`. `AList` inherits from the **List ADT**,and so must implement all of the member functions of `List`.

Java | Java (Generic)

```
// Array-based list implementation
class AList<E> implements List<E> {
  private E listArray[];                   // Array holding list elements
  private static final int DEFAULT_SIZE = 10; // Default size
  private int maxSize;                     // Maximum size of list
  private int listSize;                    // Current # of list items
  private int curr;                        // Position of current element

  // Constructors
  // Create a new list object with maximum size "size"
  @SuppressWarnings("unchecked") // Generic array allocation
  AList(int size) {
    maxSize = size;
    listSize = curr = 0;
    listArray = (E[])new Object[size];          // Create listArray
  }
  // Create a list with the default capacity
  AList() {
    this(DEFAULT_SIZE);                    // Just call the other constructor
  }

  public void clear() {                     // Reinitialize the list
    listSize = curr = 0;                // Simply reinitialize values
  }

  // Insert "it" at current position
  public boolean insert(E it) {
    if (listSize >= maxSize) {
      return false;
    }
    for (int i=listSize; i>curr; i--) {  // Shift elements up
      listArray[i] = listArray[i-1];   //   to make room
    }
    listArray[curr] = it;
    listSize++;                        // Increment list size
    return true;
  }

  // Append "it" to list
  public boolean append(E it) {
    if (listSize >= maxSize) {
      return false;
    }
    listArray[listSize++] = it;
    return true;
  }

  // Remove and return the current element
  public E remove() throws NoSuchElementException {
    if ((curr<0) || (curr>=listSize)) {  // No current element
      throw new NoSuchElementException("remove() in AList has current of " + curr + " and
        + listSize + " that is not a a valid element");
    }
    E it = listArray[curr];            // Copy the element
    for(int i=curr; i<listSize-1; i++) {// Shift them down
      listArray[i] = listArray[i+1];
    }
    listSize--;                        // Decrement size
    return it;
  }

  public void moveToStart() {      // Set to front
    curr = 0;
  }
  public void moveToEnd() {  // Set at end
    curr = listSize;
  }
  public void prev() {  // Move left
    if (curr != 0) {
      curr--;
    }
  }
  public void next() {  // Move right
    if (curr < listSize) {
      curr++;
    }
  }
  public int length() {       // Return list size
    return listSize;
  }
  public int currPos() {          // Return current position
    return curr;
  }
  }

  // Set current list position to "pos"
  public boolean moveToPos(int pos) {
    if ((pos < 0) || (pos > listSize)) {
      return false;
    }
    curr = pos;
    return true;
  }

  // Return true if current position is at end of the list
  public boolean isAtEnd() {
    return curr == listSize;
  }

  // Return the current element
  public E getValue() throws NoSuchElementException {
    if ((curr < 0) || (curr >= listSize)) {// No current element
      throw new NoSuchElementException("getvalue() in AList has current of " + curr + " a
        + listSize + " that is not a a valid element");
    }
  return listArray[curr];
  }

  //Tell if the list is empty or not
  public boolean isEmpty() {
    return listSize == 0;
  }
}
```

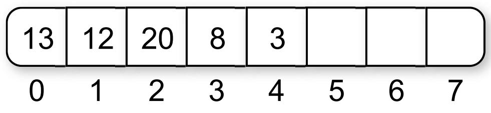
1 / 8 << < > >>

Let's take a look at the private data members for class `AList`.

```
class AList implements List {
  private Object listArray[];               // Array holding list elements
  private static final int DEFAULT_SIZE = 10; // Default size
  private int maxSize;                      // Maximum size of list
  private int listSize;                     // Current # of list items
  private int curr;                         // Position of current element
```

1 / 5

Class `AList` stores the list elements in the first `listSize` contiguous array positions. In this example, `listSiz`



4.3.1.1. Insert

Because the array-based list implementation is defined to store list elements in contiguous cells of the array, the `insert`, `append`, and `remove` methods must maintain this property.

1 / 6

<< < > >>

Inserting an element at the head of an array-based list requires shifting all existing elements in the array by o
toward the tail.

```
// Insert "it" at current position
public boolean insert(Object it) {
  if (listSize >= maxSize) return false;
  for (int i=listSize; i>curr; i--)  // Shift elements
    listArray[i] = listArray[i-1];   //   to make room
  listArray[curr] = it;
  listSize++;                        // Increment list
  return true;
}
```

4.3.1.2. Insert Practice Exericse

Practicing Array-Based List Insertion Proficiency Exercise

Current score: 0 out of 5

Your task in this exercise is to show the behavior for array-based list insertion. In the array displayed below, the "current" position is **2**.

Answer

Check Answer

The value to insert is **45**, and it is to be inserted into the "current" position.

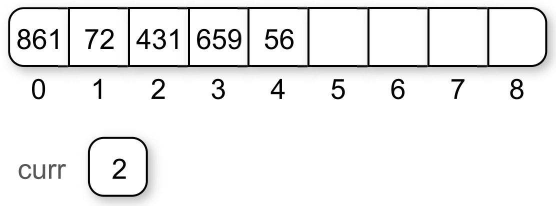
To move an element, click on it (to highlight it), then click on where you want it to go

Need help?

You can insert the new value **45** into a highlighted array position by clicking the "Insert" button.

I'd like a hint

Reset Insert

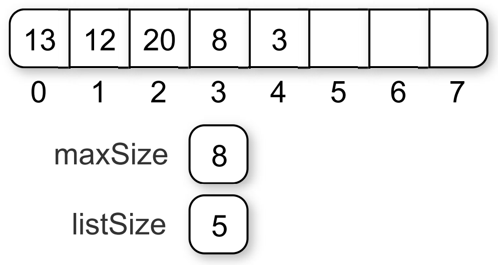


4.3.2. Append and Remove

1 / 5

<< < > >>

Inserting at the tail of the list is easy.



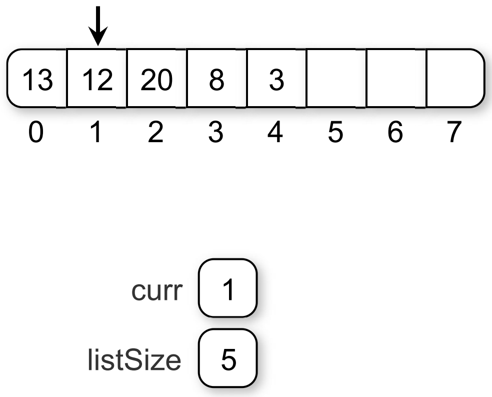
```
// Append "it" to list
public boolean append(Object it) {
  if (listSize >= maxSize) return false;
  listArray[listSize++] = it;
  return true;
}
```

Removing an element from the head of the list is similar to insert in that all remaining elements must shift toward the head by one position to fill in the gap. If we want to remove the element at position i, then $n - i - 1$ elements must shift toward the head, as shown in the following slideshow.

1 / 6

<< < > >>

Here is a list containing five elements. We will remove the value 12 in position 1 of the array, which is t
position.



```
// Remove and return the current element
public Object remove() throws NoSuchElementException {
  if ((curr<0) || (curr>=listSize))  // No current eler
    throw new NoSuchElementException("remove() in AList
      + listSize + " that is not a a valid element");
  Object it = listArray[curr];       // Copy the elemer
  for(int i=curr; i<listSize-1; i++) // Shift them dowr
    listArray[i] = listArray[i+1];
  listSize--;                        // Decrement size
  return it;
}
```

In the average case, insertion or removal each requires moving half of the elements, which is $\Theta(n)$.

4.3.2.1. Remove Practice Exericise

Practicing Array-Based List Remove Proficiency Exercise

Current score: 0 out of 5

Your task in this exercise is to show the behavior for array-based list deletion. In the array displayed below, the "current" position is **1**.

Click on a value to highlight it, then click on the return box to remember the highlighted element. Move elements in the array as appropriate by first clicking on the element that you want to move (to highlight it), then click on where it goes.

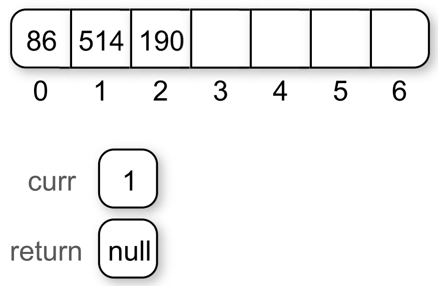
Reset

Answer

Check Answer

Need help?

I'd like a hint



Aside from `insert` and `remove`, the only other operations that might require more than constant time are the constructor and `clear`. The other methods for Class `AList` simply access the current list element or move the current position. They all require $\Theta(1)$ time.

4.3.3. Array-based List Practice Questions

Practicing Array List: Summary Questions

Current score: 0 out of 5

Given an array-based list implementation, inserting a new element to the current position takes how long in the average case?

Answer

Check Answer

04.04 Linked Lists

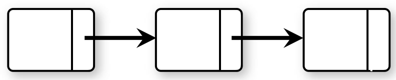
Due No Due Date **Points** 2 **Submitting** an external tool

04.04 Linked Lists

4.4. Linked Lists

4.4.1. Linked Lists

In this module we present one of the two traditional implementations for lists, usually called a **linked list**. The linked list uses **dynamic memory allocation**, that is, it allocates memory for new list elements as needed. The following diagram illustrates the linked list concept. Here there are three **nodes** that are “linked” together. Each node has two boxes. The box on the right holds a link to the next node in the list. Notice that the rightmost node has a diagonal slash through its link box, signifying that there is no link coming out of this box.



Because a list node is a distinct object (as opposed to simply a cell in an array), it is good practice to make a separate list node class. (We can also re-use the list node class to implement linked implementations for the **stack** and **queue** data structures. Here is an implementation for list nodes, called the `Link` class. Objects in the `Link` class contain an `element` field to store the element value, and a `next` field to store a pointer to the next node on the list. The list built from such nodes is called a **singly linked list**, or a **one-way list**, because each list node has a single pointer to the next node on the list.

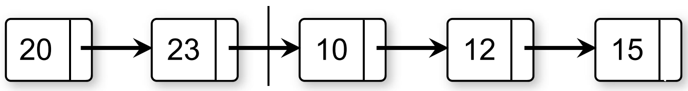
Java Java (Generic) Toggle Tree Vie

```
class Link {          // Singly linked list node class
  private Object e;   // Value for this node
  private Link n;     // Point to next node in list

  // Constructors
  Link(Object it, Link inn) { e = it; n = inn; }
  Link(Link inn) { e = null; n = inn; }

  Object element() { return e; }                     // Return the value
  Object setElement(Object it) { return e = it; } // Set element value
  Link next() { return n; }                          // Return next link
  Link setNext(Link inn) { return n = inn; }         // Set next link
  Link setNext(Link inn) { return n = inn; }       // Set next link
}
```

The `Link` class is quite simple. There are two forms for its constructor, one with an initial element value and one without. Member functions allow the link user to get or set the `element` and `link` fields.

1 / 3 &lt;&lt; &lt; &gt; &gt;&gt;

Here is a graphical depiction for a linked list storing five integers. The value stored in a pointer variable is indic arrow "pointing" to something. A NULL pointer is indicated graphically by a diagonal slash through a pointer box. The vertical line between the nodes labeled 23 and 10 indicates the current position (immediately to the r line).

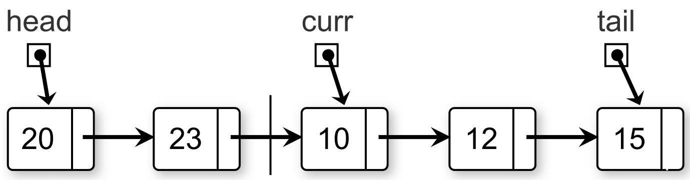


4.4.1.1. Why This Has Problems

There are a number of problems with the representation just described. First, there are lots of special cases to code for. For example, when the list is empty we have no element for `head`, `tail`, and `curr` to point to. Implementing special cases for `insert` and `remove` increases code complexity, making it harder to understand, and thus increases the chance of introducing bugs.

1 / 7 &lt;&lt; &lt; &gt; &gt;&gt;

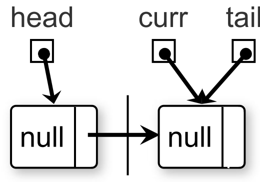
Another problem is that we have no link to get us to the preceding node (shown in yellow). So we have no ea update the yellow node's `next` pointer.



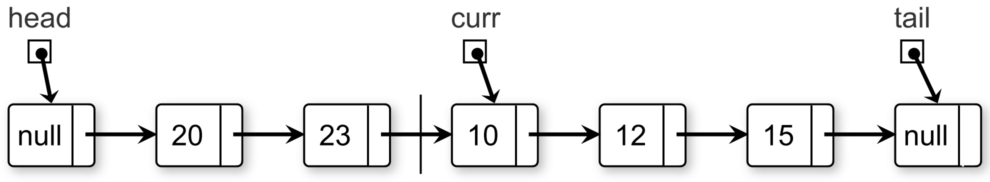
4.4.1.2. A Better Solution

Fortunately, there is a fairly easy way to deal with all of the special cases, as well as the problem with deleting the last node. Many special cases can be eliminated by implementing linked lists with an additional **header node** as the first node of the list. This header node is a link node like any other, but its value is ignored and it is not considered to be an actual element of the list. The header node saves coding effort because we no longer need to consider special cases for empty lists or when the current position is at one end of the list. The cost of this simplification is the space for the header node. However, there are space savings due to smaller code size, because statements to handle the special cases are omitted. We get rid of the remaining special cases related to being at the end of the list by adding a “trailer” node that also never stores a value.

The following diagram shows initial conditions for a linked list with header and trailer nodes.



Here is what a list with some elements looks like with the header and trailer nodes added.



Adding the trailer node also solves our problem with deleting the last node on the list, as we will see when we take a closer look at the remove method’s implementation.

4.4.1.3. Linked List Implementation

Here is the implementation for the linked list class, named `LList`.

Java | Java (Generic) | Toggle Tree Vie

```
import java.util.NoSuchElementException;

// Linked list implementation
class LList implements List {
  private Link head;         // Pointer to list header
  private Link tail;         // Pointer to last element
  private Link curr;         // Access to current element
  private int listSize;      // Size of list

  // Constructors
  LList(int size) { this(); }     // Constructor -- Ignore size
  LList() { clear(); }

  // Remove all elements
  public void clear() {
    curr = tail = new Link(null); // Create trailer
    head = new Link(tail);        // Create header
    listSize = 0;
```

```
}

// Insert "it" at current position
public boolean insert(Object it) {
  curr.setNext(new Link(curr.element(), curr.next()));
  curr.setElement(it);
  if (tail == curr) tail = curr.next();  // New tail
  listSize++;
  return true;
}

// Append "it" to list
public boolean append(Object it) {
  tail.setNext(new Link(null));
  tail.setElement(it);
  tail = tail.next();
  listSize++;
  return true;
}

// Remove and return current element
public Object remove () throws NoSuchElementException {
  if (curr == tail) // Nothing to remove
    throw new NoSuchElementException("remove() in LList has current of " + curr + " and
      + listSize + " that is not a a valid element");
  Object it = curr.element();             // Remember value
  curr.setElement(curr.next().element()); // Pull forward the next element
  if (curr.next() == tail) tail = curr;   // Removed last, move tail
  curr.setNext(curr.next().next());       // Point around unneeded link
  listSize--;                             // Decrement element count
  return it;                              // Return value
}

public void moveToStart() { curr = head.next(); } // Set curr at list start
public void moveToEnd() { curr = tail; }          // Set curr at list end

// Move curr one step left; no change if now at front
public void prev() {
  if (head.next() == curr) return; // No previous element
  Link temp = head;
  // March down list until we find the previous element
  while (temp.next() != curr) temp = temp.next();
  curr = temp;
}

// Move curr one step right; no change if now at end
public void next() { if (curr != tail) curr = curr.next(); }

public int length() { return listSize; } // Return list length


// Return the position of the current element
public int currPos() {
  Link temp = head.next();
  int i;
```

```
      for (i=0; curr != temp; i++)
        temp = temp.next();
      return i;
    }

    // Move down list to "pos" position
    public boolean moveToPos(int pos) {
      if ((pos < 0) || (pos > listSize)) return false;
      curr = head.next();
      for(int i=0; i<pos; i++) curr = curr.next();
      return true;
    }

    // Return true if current position is at end of the list
    public boolean isAtEnd() { return curr == tail; }

    // Return current element value.
    public Object getValue() throws NoSuchElementException {
      if (curr == tail) // No current element
        throw new NoSuchElementException("getvalue() in LList has current of " + curr + " a
          + listSize + " that is not a a valid element");
      return curr.element();
    }

    // Check if the list is empty
    public boolean isEmpty() { return listSize == 0; }
  }
```

1 / 7

<< < > >>

Let's look at the data members for class `LList`.

```
class LList implements List {
  private Link head;         // Pointer to list header
  private Link tail;         // Pointer to last element
  private Link curr;         // Access to current element
  private int listSize;      // Size of list
```

1 / 5

<< < > >>

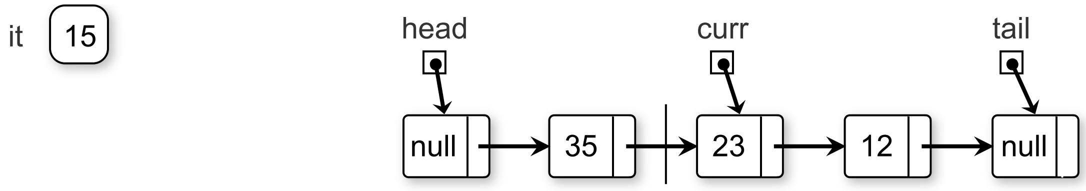
Now we look at the constructors for class `LList`.

```
// Constructors
LList(int size) { this(); }     // Constructor -- Ignore size
LList() { clear(); }

// Remove all elements
public void clear() {
  curr = tail = new Link(null); // Create trailer
  head = new Link(tail);        // Create header
  listSize = 0;
}
```

1 / 9

<< < > >>

The linked list before insertion. 15 is the value to be inserted.



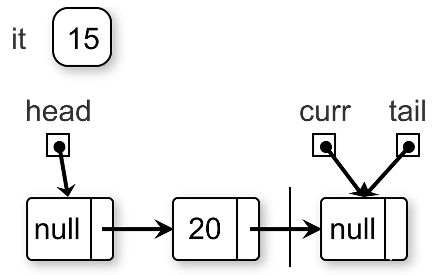
```
// Insert "it" at current position
public boolean insert(Object it) {
  curr.setNext(new Link(curr.element(), curr.next()));
  curr.setElement(it);
  if (tail == curr) tail = curr.next();  // New tail
  listSize++;
  return true;
}
```

Here are some special cases for linked list insertion: Inserting at the end, and inserting to an empty list.

1 / 16

<< < > >>

Here is an example showing insertion at the end of the list. 15 is the value to be inserted.



```
// Insert "it" at current position
public boolean insert(Object it) {
  curr.setNext(new Link(curr.element(), curr.ne›
  curr.setElement(it);
  if (tail == curr) tail = curr.next();  // New
  listSize++;
  return true;
}
```

Khan.randRange(4, 6) Khan.randRange(0, 999) Khan.randRange(0, arr_size-1 - 2) llistInsertPRO.initJSAV(arr_size, insert_pos,insert_value)

Your task in this exercise is to show the behavior for linked list insertion. You must insert the value ***insert_value*** to the current position. In the process, you will need to create a new node and move some node values and pointers.

To move an element value from one node to another, click on it (to highlight it), then click on the element position in the node where you want it to go. You can insert the new value ***insert_value*** into a highlighted node by clicking the "Insert" button. You can create a new link node by clicking the "NewNode" button. To change a pointer, click on its box (on the right side of the node) to highlight it, then click on the node that you want it to point to.

Reset NewNode Insert
[llistInsertPRO.userInput]
if (!llistInsertPRO.checkAnswer() && !guess[0]) { return ""; // User did not click, and correct answer is not // initial array state } else { return llistInsertPRO.checkAnswer(arr_size); }

You want to reproduce the behavior of the insertion function. Try starting with a new node.

Remember that the new node has to come after the current node in the list, but we want the new value to appear before the the current node's value.

So move the current node's value to the new node's value, then use "insert" to put the new value into the current node.

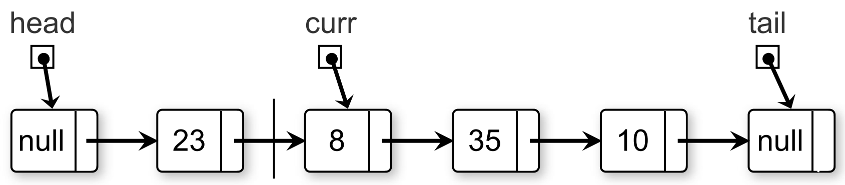
Now you need to fix up the pointers. Click the new node's pointer box, then click the node after current. Click the current node's pointer box, then click the new node.

4.4.2. Linked List Remove

1 / 7

<< < > >>

Now we look at the `remove` method.



```
// Remove and return current element
public Object remove () throws NoSuchElementException {
  if (curr == tail) // Nothing to remove
    throw new NoSuchElementException("remove() in LList has current of " + curr +
      + listSize + " that is not a a valid element");
  Object it = curr.element();             // Remember value
  curr.setElement(curr.next().element()); // Pull forward the next element
  if (curr.next() == tail) tail = curr;   // Removed last, move tail
  curr.setNext(curr.next().next());       // Point around unneeded link
  listSize--;                             // Decrement element count
  return it;                              // Return value
```

Khan.randRange(4, 6) Khan.randRange(0, arr_size-2) llistRemovePRO.initJSAV(arr_size, curr_pos)

Your task in this exercise is to show the behavior for Linked list deletion. You must delete the element in the current position.

To move an element, click on it (to highlight it), then click on the element position in the node where you want it to go. To set the "return" value, click on the element position in the node you want to return (to highlight it), then click on the return value box to set it to the highlighted value. You can make a node's "next" pointer point to "null" by first clicking the pointer for the node and then clicking the "makenull" button. To change the target of labels, such as "curr" and "tail", click on the label (to highlight it), then click on the node you want it to point to.

Reset makenull

[llistRemovePRO.userInput]
if (!llistRemovePRO.checkAnswer(arr_size, curr_pos) && !guess[0]) { return ""; // User did not click, and

correct answer is not // initial array state } else {return llistRemovePRO.checkAnswer(arr_size, curr_pos);}

If "curr" points to the same node as "tail", the default list is a correct answer. Otherwise, the first step could be to remember the value of the "curr" node.

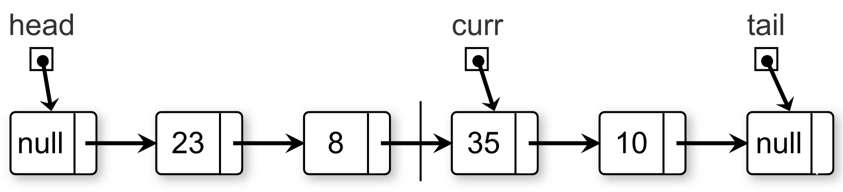
Copy value to "curr" from the node following 'curr'.

If "curr" points to the node proceding tail, sets the "curr" node to point to "null". Label "tail" should point to the current node. Otherwise, the "curr" node should point to the following node of the node it used to point to.

1 / 10

<< < > >>

Finally, we will look at how a few other methods work.



Implementations for the remaining operations each require $\Theta(1)$ time.

04.05 Comparison of List Implementations

Due No Due Date **Points** 2 **Submitting** an external tool

04.05 Comparison of List Implementations

4.5. Comparison of List Implementations

4.5.1. Space Comparison

Now that you have seen two substantially different implementations for lists, it is natural to ask which is better. In particular, if you must implement a list for some task, which implementation should you choose?

Given a collection of elements to store, they take up some amount of space whether they are simple integers or large objects with many fields. Any container data structure like a list then requires some additional space to organize the elements being stored. This additional space is called **overhead**.

Array-based lists have the disadvantage that their size must be predetermined before the array can be allocated. Array-based lists cannot grow beyond their predetermined size. Whenever the list contains only a few elements, a substantial amount of space might be tied up in a largely empty array. This empty space is the overhead required by the array-based list. **Linked lists** have the advantage that they only need space for the objects actually on the list. There is no limit to the number of elements on a linked list, as long as there is **free store** memory available. The amount of space required by a linked list is $\Theta(n)$, while the space required by the array-based list implementation is $\Omega(n)$, but can be greater.

Array-based lists have the advantage that there is no wasted space for an individual element. Linked lists require that an extra pointer for the `next` field be added to every list node. So the linked list has these `next` pointers as overhead. If the element size is small, then the overhead for links can be a significant fraction of the total storage. When the array for the array-based list is completely filled, there is no wasted space, and so no overhead. The array-based list will then be more space efficient, by a constant factor, than the linked implementation.

A simple formula can be used to determine whether the array-based list or the linked list implementation will be more space efficient in a particular situation. Call n the number of elements currently in the list, P the size of a pointer in storage units (typically four bytes), E the size of a data element in storage units (this could be anything, from one bit for a Boolean variable on up to thousands of bytes or more for complex records), and D the maximum number of list elements that can be stored in the array. The amount of space required for the array-based list is DE, regardless of the number of elements actually stored in the list at any given time. The amount of space required for the linked list is $n(P+E)$. The smaller of these expressions for a given value n determines the more space-efficient implementation for n elements. In general, the linked implementation requires less space than the array-based implementation when relatively few elements are in the list. Conversely, the array-based implementation becomes more space efficient when the array is close to full. Using the equation, we can solve for n to determine the **break-even point** beyond which the array-based implementation is more space efficient in any particular situation. This

even point beyond which the array-based implementation is more space efficient in any particular situation. This occurs when

$$n > DE/(P + E).$$

If $P = E$, then the break-even point is at $D/2$. This would happen if the element field is either a four-byte `int` value or a pointer, and the `next` field is a typical four-byte pointer. That is, the array-based implementation would be more efficient (if the link field and the element field are the same size) whenever the array is more than half full.

As a rule of thumb, linked lists are more space efficient when implementing lists whose number of elements varies widely or is unknown. Array-based lists are generally more space efficient when the user knows in advance approximately how large the list will become, and can be confident that the list will never grow beyond a certain limit.

*4 * Khan.randRange(1,2) Khan.randRange(1, 16) P+D listOverhead.genAnswer(P, D)*

Assume that for some list implementation, a pointer requires *P* bytes and a data object requires *D* bytes. Type a fraction (like "1/2") to show how full the array should be for the break even point, that is, the point beyond which the array-based list implementation needs less space than the linked list implementation. Give your fraction in lowest terms.

ANS

The overhead is *P* and the total space needed is *SUM*.

As the array fills up, its overhead decreases.

The pointer space (*P* bytes) is overhead.

The bigger the overhead fraction, the less the array needs to be better. The bigger the data field, the more the array needs to be better.

The break-even point is at *D*/*SUM*.

If you have values like 4/8, reduce to 1/2.

4.5.2. Time Comparison

Array-based lists are faster for access by position. Positions can easily be adjusted forwards or backwards by the `next` and `prev` methods. These operations always take $\Theta(1)$ time. In contrast, singly linked lists have no explicit next and prev methods. These operations always take $\Theta(1)$ time. In contrast, singly linked lists have no explicit access to the previous element, and access by position requires that we march down the list from the front (or the current position) to the specified position. Both of these operations require $\Theta(n)$ time in the average and worst cases, if we assume that each position on the list is equally likely to be accessed on any call to `prev` or `moveToPos`.

Given a pointer to a suitable location in the list, the `insert` and `remove` methods for linked lists require only $\Theta(1)$ time. Array-based lists must shift the remainder of the list up or down within the array. This requires $\Theta(n)$ time in the average and worst cases. For many applications, the time to insert and delete elements dominates all other operations. For this reason, linked lists are often preferred to array-based lists.

When implementing the array-based list, an implementor could allow the size of the array to grow and shrink depending on the number of elements that are actually stored. This data structure is known as a **dynamic array**. For example, both the Java and C++/STL `Vector` classes implement a dynamic array, and JavaScript arrays are always dynamic. Dynamic arrays allow the programmer to get around the limitation on the traditional array that its size cannot be changed once the array has been created. This also means that space need not be allocated to the dynamic array until it is to be used. The disadvantage of this approach is that it takes time to deal with space adjustments on the array. Each time the array grows in size, its contents must be copied. A good implementation of the dynamic array will grow and shrink the array in such a way as to keep the overall cost for a series of insert/delete operations relatively inexpensive, even though an occasional insert/delete operation might be expensive. A simple rule of thumb is to double the size of the array when it becomes full, and to cut the array size in half when it becomes one quarter full. To analyze the overall cost of dynamic array operations over time, we need to use a technique known as **amortized analysis**.

4.5.2.1. Practice Questions

4.6. Doubly Linked Lists

4.6.1. Doubly Linked Lists

The **singly linked list** allows for direct access from a list node only to the next node in the list. A **doubly linked list** allows convenient access from a list node to the next node and also to the preceding node on the list. The doubly linked list node accomplishes this in the obvious way by storing two pointers: one to the node following it (as in the singly linked list), and a second pointer to the node preceding it.

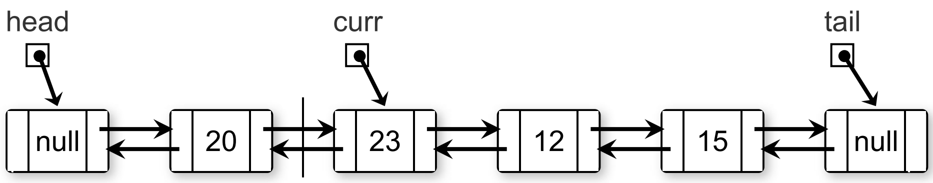


Figure 4.6.1: A doubly linked list.

The most common reason to use a doubly linked list is because it is easier to implement than a singly linked list. While the code for the doubly linked implementation is a little longer than for the singly linked version, it tends to be a bit more “obvious” in its intention, and so easier to implement and debug. Whether a list implementation is doubly or singly linked should be hidden from the `List` class user.

Like our singly linked list implementation, the doubly linked list implementation makes use of a **header node**. We also add a tailer node to the end of the list. The tailer is similar to the header, in that it is a node that contains no value, and it always exists. When the doubly linked list is initialized, the header and tailer nodes are created. Data member `head` points to the header node, and `tail` points to the tailer node. The purpose of these nodes is to simplify the `insert`, `append`, and `remove` methods by eliminating all need for special-case code when the list is empty, or when we insert at the head or tail of the list.

In our implementation, `curr` will point to the **current position** (or to the **trailer node** if the current position is at the end of the list).

Here is the complete implementation for a `Link` class to be used with doubly linked lists. This code is a little longer than that for the singly linked list node implementation since the doubly linked list nodes have an extra data member.

Java | **Java (Generic)** | Toggle Tree Vie

```
class Link {            // Doubly linked list node
  private Object e;     // Value for this node
  private Link n;       // Pointer to next node in list
  private Link p;       // Pointer to previous node

  // Constructors
  Link(Object it, Link inp, Link inn) { e = it;  p = inp; n = inn; }
  Link(Link inp, Link inn) { p = inp; n = inn; }
```

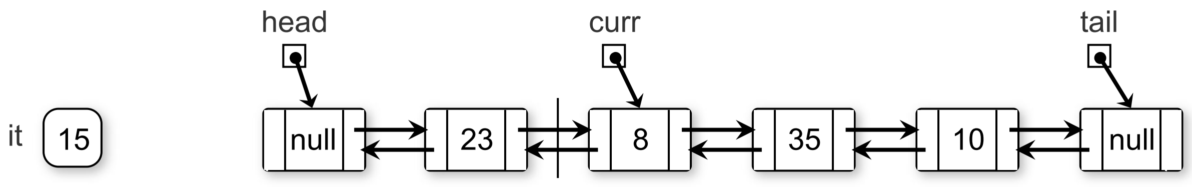
```
    Link(Link inp, Link inn) { p = inp; n = inn; }

    // Get and set methods for the data members
    public Object element() { return e; }                  // Return the value
    public Object setElement(Object it) { return e = it; }    // Set element value
    public Link next() { return n; }                       // Return next link
    public Link setNext(Link nextval) { return n = nextval; } // Set next link
    public Link prev() { return p; }                       // Return prev link
    public Link setPrev(Link prevval) { return p = prevval; } // Set prev link
  }
```

4.6.1.1. Insert

The following slideshows illustrate the `insert` and `append` doubly linked list methods. The class declaration and the remaining member functions for the doubly linked list class are nearly identical to the singly linked list version. While the code for these methods might be a little longer than their singly linked list counterparts (since there is an extra pointer in each node to deal with), they tend to be easier to understand.

1 / 10   <<   <   >   >>

The linked list before insertion. 15 is the value to be inserted.

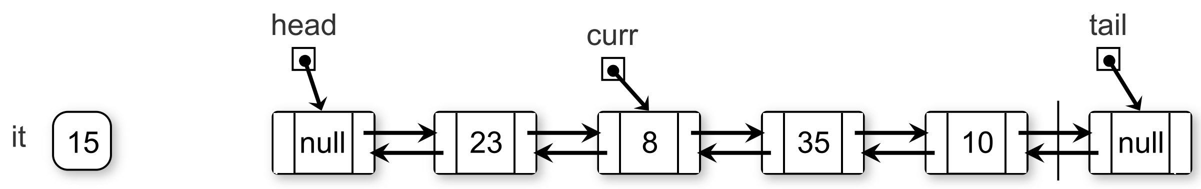


```
public boolean insert(Object it) {
  curr = new Link(it, curr.prev(), curr);
  curr.prev().setNext(curr);
  curr.next().setPrev(curr);
  listSize++;
  return true;
}
```

4.6.1.2. Append

1 / 8   <<   <   >   >>

The append method works almost the same as insertion. We will insert the value 15.



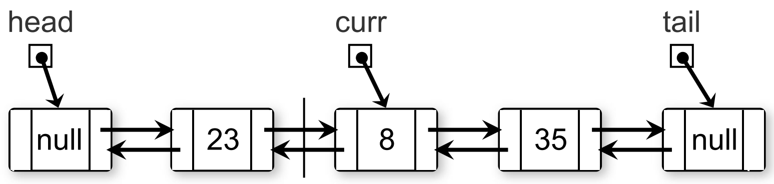
```
public boolean append(Object it) {
  tail.setPrev(new Link(it, tail.prev(), tail));
  tail.prev().prev().setNext(tail.prev());
  if (curr == tail) curr = tail.prev();
  listSize++;
  return true;
}
```

4.6.1.3. Remove

1 / 9

<< < > >>

Now we will look at the remove method. Here is the linked list before we remove the node with value 8.



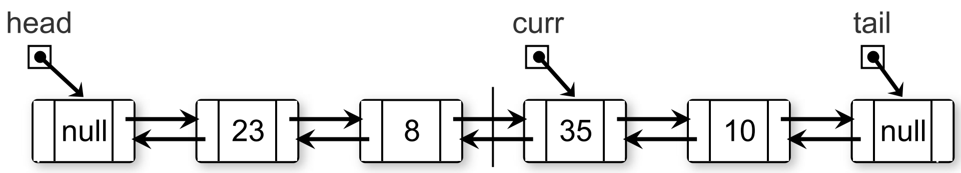
```
public Object remove() {
  if (curr == tail) return null;        // Nothing to remove
  Object it = curr.element();           // Remember value
  curr.prev().setNext(curr.next());     // Remove from list
  curr.next().setPrev(curr.prev());
  curr = curr.next();
  listSize--;                           // Decrement node count
  return it;                            // Return value removed
}
```

4.6.1.4. Prev

1 / 5

<< < > >>

The prev method is easy.



```
public void prev() {
  if (curr.prev() != head)   // Can't back up from list head
    curr = curr.prev();
}
```

The only disadvantage of the doubly linked list as compared to the singly linked list is the additional space used. The doubly linked list requires two pointers per node, and so in the implementation presented it requires twice as much overhead as the singly linked list.

4.6.1.5. Mangling Pointers

There is a space-saving technique that can be employed to eliminate the additional space requirement, though it will complicate the implementation and be somewhat slower. Thus, this is an example of a space/time tradeoff. It is based on observing that, if we store the sum of two values, then we can get either value back by subtracting the other. That is, if we store $a + b$ in variable c, then $b = c - a$ and $a = c - b$. Of course, to recover one of the values out of the stored summation, the other value must be supplied. A pointer to the first node in the list, along with the value of one of its two link fields, will allow access to all of the remaining nodes of the list in order. This is because the pointer to the node must be the same as the value of the following node's `prev` pointer, as well as the previous node's `next` pointer. It is possible to move down the list breaking apart the summed link fields as though you were opening a zipper.

The principle behind this technique is worth remembering, as it has many applications. The following code fragment will swap the contents of two variables without using a temporary variable (at the cost of three arithmetic operations).

Java | Java (Generic) | Toggle Tree Vie

```
a = a + b;
b = a - b; // Now b contains original value of a
a = a - b; // Now a contains original value of b
```

04.07 List Element Implementations

Due No Due Date **Points** 1 **Submitting** an external tool

04.07 List Element Implementations

4.7. List Element Implementations

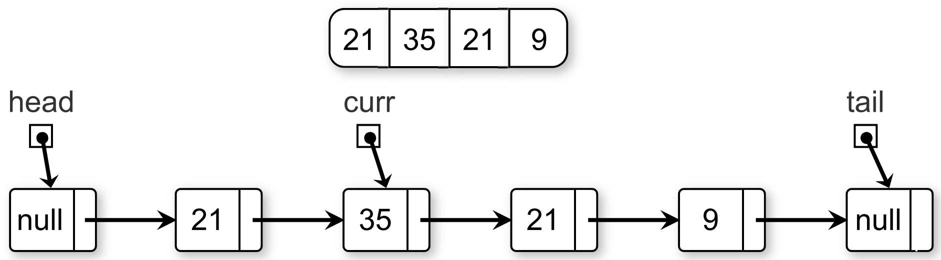
4.7.1. List Element Implementations

When designing any **container class**, there are a number of design choices to be made regarding the data elements.

What to do if something can appear multiple times on a list? One option is to use a reference to **elements**. Another is to store separate copies. In general, the larger the elements and the more that they are duplicated, the more likely that pointers to shared elements is the better approach.

1 / 3 << < > >>

This slide show values stored directly in the list elements. If something appears multiple times, then there a copies of that thing. For small elements such as an integer, this makes sense.



4.7.1.1. Homogeneity

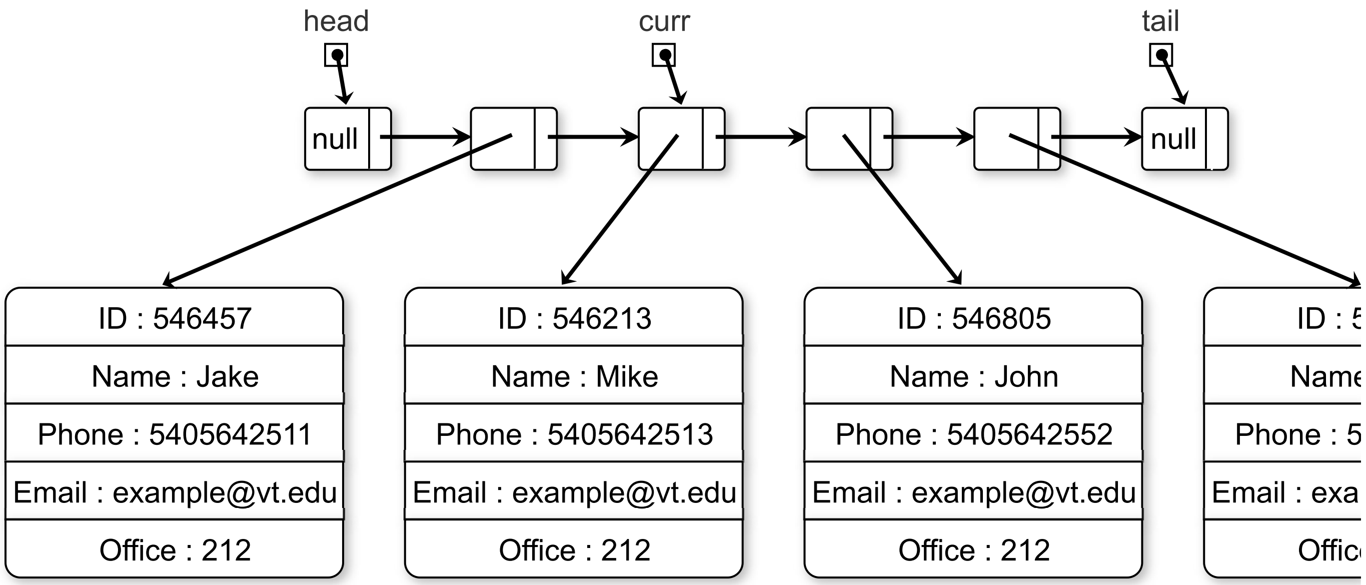
The next issue to consider is whether to enforce **homogeneity** in the list elements. That is, should lists be restricted so that all data elements stored are of the same object type? Or should it be possible to store different types?

If you want to enforce homogeneity, the most rigid way is to simply define the elements to be of a fixed type. But that does not help if you want one list to store integers while another stores strings. A much more flexible approach is to use Java generics or C++ templates. In this way, the compiler will enforce that a given list will only store a single data type, while still allowing different lists to have different data types. Another approach is to store an object of the appropriate type in the header node of the list (perhaps an object of the appropriate type is supplied as a parameter to the list constructor), and then check that all insert operations on that list use the same element type. This approach is useful in a language like JavaScript that does not use strong typing, but does allow a program to test the type of an object.

In some applications, the designer would like to allow a given list store elements with different types. In Java, declaring the element to be of type `Object` will stop the compiler from enforcing any type restrictions. In C++, a similar effect can be achieved by using `void*` pointers.

1 / 2 << < > >>

In some applications, the user would like to define the class of the data element that is stored on a given list never permit objects of a different class to be stored on that same list.

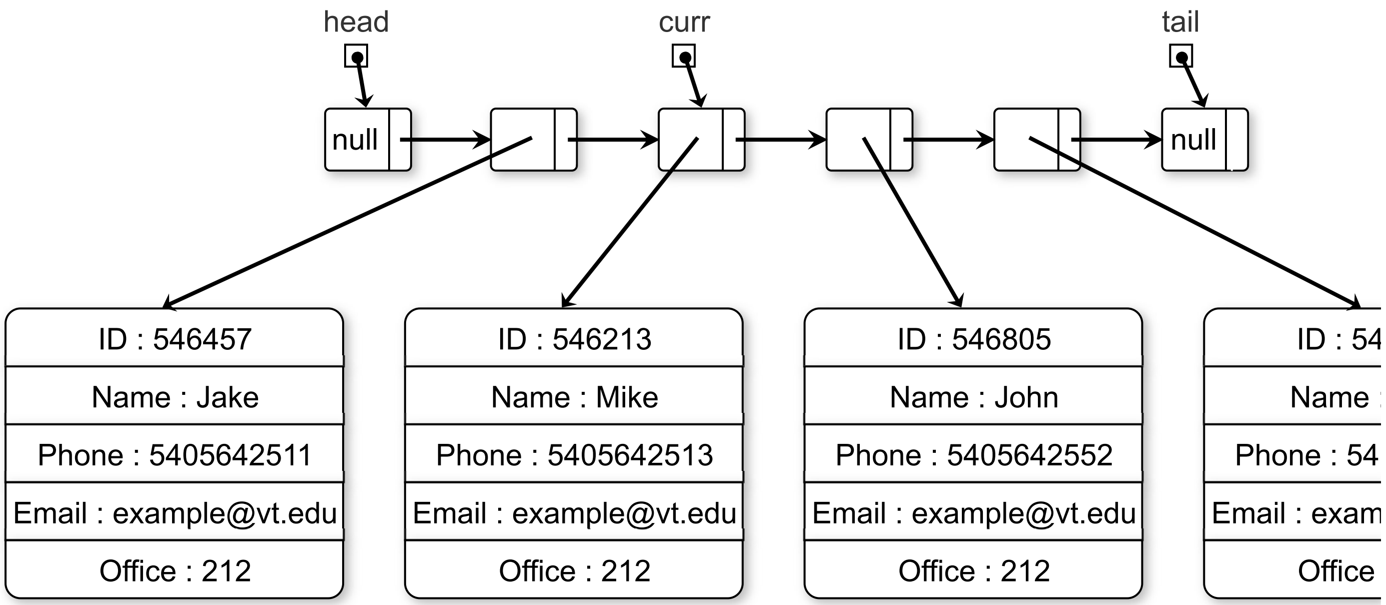


4.7.1.2. Element Deletion

Our last design issue is what to do to the list elements when the list itself is deleted? This is a serious concern in a language like C++ that does not support automatic garbage collection.

1 / 2 << < > >>

In this example, consider what happens when the list is deleted.



4.7.1.3. Practice Questions

Practicing List: Summary Questions

Current score: **0** out of **5**

Answer TRUE or FALSE.

Linked lists are better than array-based lists when the final size of the list is known in advance

○ True

○ False

Answer

Check Answer

Need help?

I'd like a hint

04.08 Stacks

Due No Due Date **Points** 2 **Submitting** an external tool

04.08 Stacks

4.8. Stacks

4.8.1. Stack Terminology and Implementation

The **stack** is a list-like structure in which elements may be inserted or removed from only one end. While this restriction makes stacks less flexible than lists, it also makes stacks both efficient (for those operations they can do) and easy to implement. Many applications require only the limited form of insert and remove operations that stacks provide. In such cases, it is more efficient to use the simpler stack data structure rather than the generic list. For example, the **freelist** is really a stack.

Despite their restrictions, stacks have many uses. Thus, a special vocabulary for stacks has developed. Accountants used stacks long before the invention of the computer. They called the stack a "**LIFO**" list, which stands for "Last-In, First-Out." Note that one implication of the LIFO policy is that stacks remove elements in reverse order of their arrival.

The accessible element of the stack is called the `top` element. Elements are not said to be inserted, they are **pushed** onto the stack. When removed, an element is said to be **popped** from the stack. Here is a simple stack **ADT**.

Java | Java (Generic) | Toggle Tree Vie

```
public interface Stack { // Stack class ADT
  // Reinitialize the stack.
  public void clear();

  // Push "it" onto the top of the stack
  public boolean push(Object it);

  // Remove and return the element at the top of the stack
  public Object pop();

  // Return a copy of the top element
  public Object topValue();

  // Return the number of elements in the stack
  public int length();
  // Return true if the stack is empty
  public boolean isEmpty();
}
```

As with lists, there are many variations on stack implementation. The two approaches presented here are the **array-based stack** and the **linked stack**, which are analogous to array-based and linked lists, respectively.

4.8.1.1. Array-Based Stacks

Here is a complete implementation for the array-based stack class.

Java | Java (Generic) | Toggle Tree Vie

```
class AStack implements Stack {
  private Object stackArray[];    // Array holding stack
  private static final int DEFAULT_SIZE = 10;
  private int maxSize;            // Maximum size of stack
  private int top;                // First free position at top

  // Constructors
  AStack(int size) {
    maxSize = size;
    top = 0;
    stackArray = new Object[size]; // Create stackArray
  }
  AStack() { this(DEFAULT_SIZE); }

  public void clear() { top = 0; }    // Reinitialize stack

// Push "it" onto stack
  public boolean push(Object it) {
    if (top >= maxSize) return false;
    stackArray[top++] = it;
    return true;
  }

// Remove and return top element
  public Object pop() {
    if (top == 0) return null;
    return stackArray[--top];
  }

  public Object topValue() {          // Return top element
    if (top == 0) return null;
    return stackArray[top-1];
  }

  public int length() { return top; } // Return stack size
```

```
    public boolean isEmpty() { return top == 0; } // Check if the stack is empty
  }
```

1 / 3

<< < > >>

As with any array-based implementation, `stackArray` must be declared of fixed size when the stack is createc

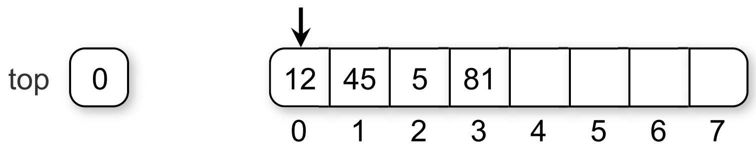
```
class AStack implements Stack {
  private Object stackArray[];    // Array holding stack
  private static final int DEFAULT_SIZE = 10;
  private int maxSize;            // Maximum size of stack
  private int top;                // First free position at top

  // Constructors
  AStack(int size) {
    maxSize = size;
    top = 0;
    stackArray = new Object[size]; // Create stackArray
  }
  AStack() { this(DEFAULT_SIZE); }
```

The array-based stack implementation is essentially a simplified version of the array-based list. The only important design decision to be made is which end of the array should represent the top of the stack.

1 / 6

<< < > >>

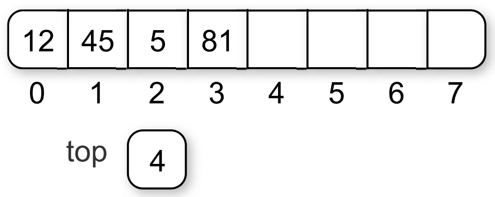
One choice is to make the top be at position 0 in the array. In terms of list functions, all push and pop operati then be on the element at position 0.



1 / 4

<< < > >>

Method push is easy.

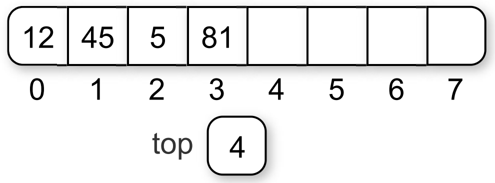


```
public boolean push(Object it) {
  if (top >= maxSize) return false;
  stackArray[top++] = it;
  return true;
}
```

4.8.2. Pop

1 / 4

Now, for the implementation of `pop`. `top` is at the first free position, which is index 4 on the array.



```
public Object pop() {
  if (top == 0) return null;
  return stackArray[--top];
}
```

04.09 Linked Stacks

Due No Due Date **Points** 2 **Submitting** an external tool

04.09 Linked Stacks

4.9. Linked Stacks

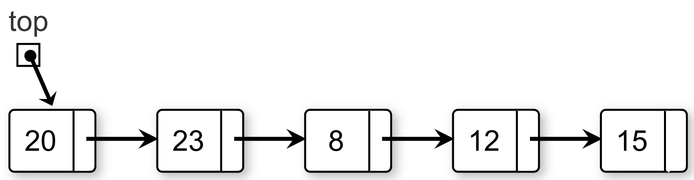
4.9.1. Linked Stack Implementation

The linked stack implementation is quite simple. Elements are inserted and removed only from the head of the list. A header node is not used because no special-case code is required for lists of zero or one elements. Here is the complete linked stack implementation.

Java Java (Generic) Toggle Tree Vie

```
// Linked stack implementation
class LStack implements Stack {
  private Link top;               // Pointer to first element
  private int size;               // Number of elements

  // Constructors
  LStack() { top = null; size = 0; }
  LStack(int size) { top = null; size = 0; }

  // Reinitialize stack
  public void clear() { top = null; size = 0; }

// Put "it" on stack
  public boolean push(Object it) {
    top = new Link(it, top);
    size++;
    return true;
  }

// Remove "it" from stack
  public Object pop() {
    if (top == null) return null;
    Object it = top.element();
    top = top.next();
    size--;
    return it;
  }
  public Object topValue() {      // Return top value
    if (top == null) return null;
    return top.element();
  }

  // Return stack length
  public int length() { return size; }

  // Check if the stack is empty
  public boolean isEmpty() { return size == 0; }
}
```

Here is a visual representation for the linked stack.

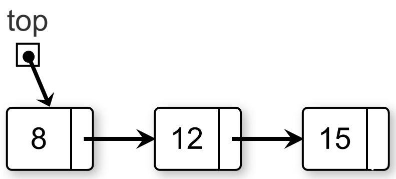


4.9.1.1. Linked Stack Push

1 / 6

<< < > >>

Here is the push operation. First we see the linked stack before push



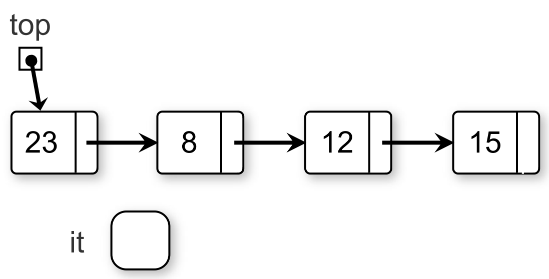
```
public boolean push(Object it) {
  top = new Link(it, top);
  size++;
  return true;
}
```

4.9.2. Linked Stack Pop

1 / 6

<< < > >>

Method pop is also quite simple.

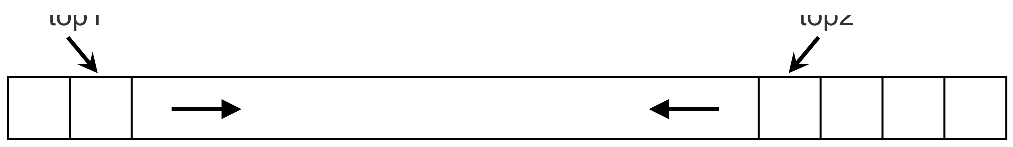


```
public Object pop() {
  if (top == null) return null;
  Object it = top.element();
  top = top.next();
  size--;
  return it;
}
```

4.9.2.1. Comparison of Array-Based and Linked Stacks

All operations for the array-based and linked stack implementations take constant time, so from a time efficiency perspective, neither has a significant advantage. Another basis for comparison is the total space required. The analysis is similar to that done for list implementations. The array-based stack must declare a fixed-size array initially, and some of that space is wasted whenever the stack is not full. The linked stack can shrink and grow but requires the overhead of a link field for every element.

When implementing multiple stacks, sometimes you can take advantage of the one-way growth of the array-based stack by using a single array to store two stacks. One stack grows inward from each end as illustrated by the figure below, hopefully leading to less wasted space. However, this only works well when the space requirements of the two stacks are inversely correlated. In other words, ideally when one stack grows, the other will shrink. This is particularly effective when elements are taken from one stack and given to the other. If instead both stacks grow at the same time, then the free space in the middle of the array will be exhausted quickly.


top1
top2

04.10 Queues

Due No Due Date **Points** 2 **Submitting** an external tool

04.10 Queues

4.10. Queues

4.10.1. Queue Terminology and Implementation

Like the stack, the **queue** is a list-like structure that provides restricted access to its elements. Queue elements may only be inserted at the back (called an **enqueue** operation) and removed from the front (called a **dequeue** operation). Queues operate like standing in line at a movie theater ticket counter. If nobody cheats, then newcomers go to the back of the line. The person at the front of the line is the next to be served. Thus, queues release their elements in order of arrival. In Britain, a line of people is called a “queue”, and getting into line to wait for service is called “queuing up”. Accountants have used queues since long before the existence of computers. They call a queue a “FIFO” list, which stands for “First-In, First-Out”. Here is a sample queue ADT. This section presents two implementations for queues: the array-based queue and the linked queue.

Java | Java (Generic) | Toggle Tree Vie

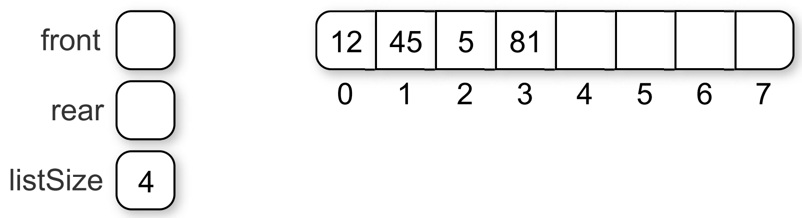
```
public interface Queue { // Queue class ADT
  // Reinitialize queue
  public void clear();

  // Put element on rear
  public boolean enqueue(Object it);

  // Remove and return element from front
  public Object dequeue();

  // Return front element
  public Object frontValue();

  // Return queue size
  public int length();

  // Return true if the queue is empty
  public boolean isEmpty();
}
```

4.10.1.1. Array-Based Queues

The array-based queue is somewhat tricky to implement effectively. A simple conversion of the array-based list implementation is not efficient.

1 / 5 << < > >>

Assume that there are *n* elements in the queue. By analogy to the array-based list implementation, we could r
all elements of the queue be stored in the first n positions of the array.

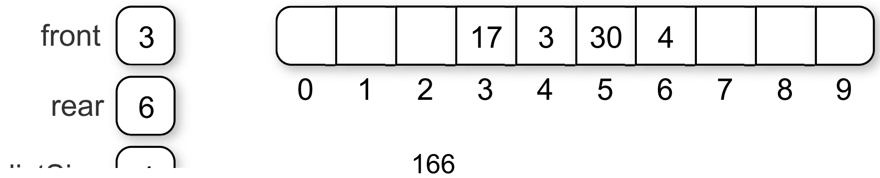


1 / 9 << < > >>

A more efficient implementation can be obtained by relaxing the requirement that all elements of the queue
the first n positions of the array. We still require that the queue be stored be in contiguous array positio
contents of the queue will be permitted to drift within the array.

1 / 7 << < > >>

This implementation raises a new problem. When elements are removed from the queue, the front index increa



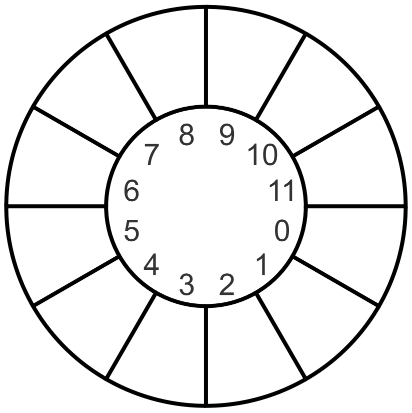
listSize 4

4.10.1.2. The Circular Queue

1 / 5

<< < > >>

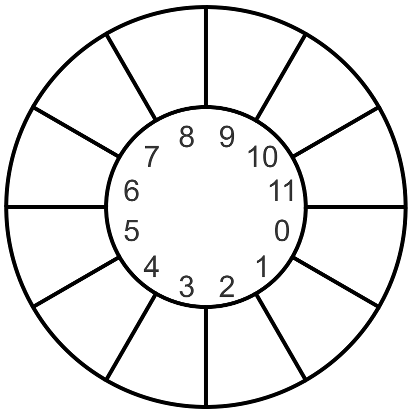
The "drifting queue" problem can be solved by pretending that the array is circular and so allow the queue t directly from the highest-numbered position in the array to the lowest-numbered position.



1 / 8

<< < > >>

There remains one more serious, though subtle, problem to the array-based queue implementation. Ho recognize when the queue is empty or full?



If the value of `front` is fixed, then $n+1$ different values for `rear` are needed to distinguish among the $n+1$ states. However, there are only n possible values for `rear` unless we invent a special case for, say, empty queues. This is an example of the **Pigeonhole Principle**. The Pigeonhole Principle states that, given n pigeonholes and $n+1$ pigeons, when all of the pigeons go into the holes we can be sure that at least one hole contains more than one pigeon. In similar manner, we can be sure that two of the $n+1$ states are indistinguishable by the n relative values of `front` and `rear`. We must seek some other way to distinguish full from empty queues.

One obvious solution is to keep an explicit count of the number of elements in the queue, or at least a Boolean variable that indicates whether the queue is empty or not. Another solution is to make the array be of size $n+1$, and only allow n elements to be stored. Which of these solutions to adopt is purely a matter of the implementor's taste in such affairs. Our choice here is to use an array of size $n+1$.

Here is an array-based queue implementation.

Java | Java (Generic) | Toggle Tree Vie

```
class AQueue implements Queue {
  private Object queueArray[]; // Array holding queue elements
  private static final int DEFAULT_SIZE = 10;
  private int maxSize;         // Maximum size of queue
  private int front;           // Index of front element
  private int rear;            // Index of rear element

  // Constructors
  AQueue(int size) {
    maxSize = size + 1;          // One extra space is allocated
    rear = 0; front = 1;
    queueArray = new Object[maxSize];  // Create queueArray
  }
  AQueue() { this(DEFAULT_SIZE); }

  // Reinitialize
  public void clear() { rear = 0; front = 1; }

  // Put "it" in queue
  public boolean enqueue(Object it) {
    if (((rear+2) % maxSize) == front) return false;  // Full
    rear = (rear+1) % maxSize; // Circular increment
    queueArray[rear] = it;
    return true;
  }

  // Remove and return front value
  public Object dequeue() {
    if(length() == 0) return null;
    Object it = queueArray[front];
    front = (front+1) % maxSize; // Circular increment
    return it;
  }
```

```
    // Return front value
    public Object frontValue() {
      if (length() == 0) return null;
      return queueArray[front];
    }

    // Return queue size
    public int length() { return ((rear+maxSize) - front + 1) % maxSize; }

    // Check if the queue is empty
    public boolean isEmpty() { return front - rear == 1; }
  }
```

4.10.1.3. Array-based Queue Implementation

1 / 5 << < > >>

Member `queueArray` holds the queue elements...

```
class AQueue implements Queue {
  private Object queueArray[]; // Array holding queue elements
  private static final int DEFAULT_SIZE = 10;
  private int maxSize;         // Maximum size of queue
  private int front;           // Index of front element
  private int rear;            // Index of rear element

  // Constructors
  AQueue(int size) {
    maxSize = size + 1;          // One extra space is allocated
    rear = 0; front = 1;
    queueArray = new Object[maxSize];  // Create queueArray
  }
  AQueue() { this(DEFAULT_SIZE); }
```

In this implementation, the front of the queue is defined to be toward the lower numbered positions in the array (in the counter-clockwise direction in the circular array), and the rear is defined to be toward the higher-numbered positions. Thus, `enqueue` increments the rear pointer (modulus `maxSize`), and `dequeue` increments the front pointer. Implementation of all member functions is straightforward.

4.10.2. Array-based Dequeue Practice

04.11 Linked Queues

Due No Due Date **Points** 3 **Submitting** an external tool

04.11 Linked Queues

4.11. Linked Queues

4.11.1. Linked Queues

The linked queue implementation is a straightforward adaptation of the linked list. Here is the linked queue class declaration.

Java Java (Generic) Toggle Tree Vie

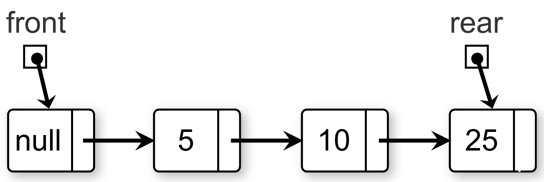
```
// Linked queue implementation
class LQueue implements Queue {
  private Link front; // Pointer to front queue node
  private Link rear;  // Pointer to rear queue node
  private int size;   // Number of elements in queue

  // Constructors
  LQueue() { init(); }
  LQueue(int size) { init(); } // Ignore size

  // Initialize queue
  void init() {
    front = rear = new Link(null);
    size = 0;
  }

  // Put element on rear
  public boolean enqueue(Object it) {
    rear.setNext(new Link(it, null));
    rear = rear.next();
    size++;
    return true;
  }

  // Remove and return element from front
  public Object dequeue() {
    if (size == 0) return null;
    Object it = front.next().element(); // Store the value
```

```
      front.setNext(front.next().next()); // Advance front
      if (front.next() == null) rear = front; // Last element
      size--;
      return it; // Return element
    }

    // Return front element
    public Object frontValue() {
      if (size == 0) return null;
      return front.next().element();
    }

    // Return queue size
    public int length() { return size; }

    // Check if the queue is empty
    public boolean isEmpty() { return size == 0; }
  }
```

1 / 4

<< < > >>

Members front and rear are pointers to the front and rear queue elements, respectively.

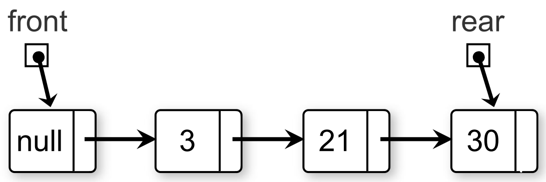


```
// Linked queue implementation
class LQueue implements Queue {
  private Link front; // Pointer to front queue node
  private Link rear;  // Pointer to rear queue node
  private int size;   // Number of elements in queue
  // Constructors
  LQueue() { init(); }
  LQueue(int size) { init(); } // Ignore size
  // Initialize queue
  void init() {
    front = rear = new Link(null);
    size = 0;
  }
```

1 / 5

<< < > >>

Let's look at how the enqueue operation works.



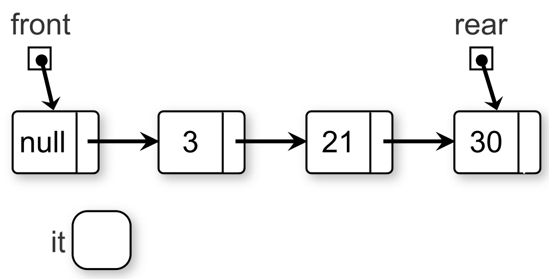
```
// Put element on rear
public boolean enqueue(Object it) {
  rear.setNext(new Link(it, null));
  rear = rear.next();
  size++;
  return true;
}
```

4.11.2. Linked Dequeue

1 / 8

<< < > >>

Now for the dequeue operation.



```
// Remove and return element from front
public Object dequeue() {
  if (size == 0) return null;
  Object it = front.next().element(); // Store the va
  front.setNext(front.next().next()); // Advance fron
  if (front.next() == null) rear = front; // Last ele
  size--;
  return it; // Return element
}
```

4.11.3. Comparison of Array-Based and Linked Queues

All member functions for both the array-based and linked queue implementations require constant time. The space comparison issues are the same as for the equivalent stack implementations. Unlike the array-based stack implementation, there is no convenient way to store two queues in the same array, unless items are always transferred directly from one queue to the other.

4.11.3.1. Stack and Queue Summary Questions

04.12 Linear Structure Summary Exercises

Due No Due Date **Points** 2 **Submitting** an external tool

04.12 Linear Structure Summary Exercises

4.12. Linear Structure Summary Exercises

4.12.1. Practice Questions

Here are some general practice questions about various data structures in this chapter.

Practicing List Data Structures General Questions

Current score: 0 out of 5

The term "FIFO" is associated with which data structure?

- ◯ Stack
- ◯ None of these
- ◯ All of these
- ◯ List
- ◯ Freelist
- ◯ Queue

Answer

Check Answer

Need help?

I'd like a hint

4.12.2. Chapter Review Questions

Here is a summary exercise with questions from everything in this chapter.

Chapter 5: Binary Trees

5.1. Binary Trees Chapter Introduction

Tree structures enable efficient access and efficient update to large collections of data. **Binary trees** in particular are widely used and relatively easy to implement. But binary trees are useful for many things besides searching. Just a few examples of applications that trees can speed up include **prioritizing jobs**, **describing mathematical expressions** and the syntactic elements of computer programs, or organizing the information needed to drive **data compression algorithms**.

This chapter covers terminology used for discussing binary trees, **tree traversals**, approaches to implementing tree **nodes**, and various examples of binary trees.

05.02 Binary Trees

Due No Due Date **Points** 2 **Submitting** an external tool

05.02 Binary Trees

5.2. Binary Trees

5.2.1. Definitions and Properties

A **binary tree** is made up of a finite set of elements called **nodes**. This set either is empty or consists of a node called the **root** together with two binary trees, called the left and right **subtrees**, which are disjoint from each other and from the root. (Disjoint means that they have no nodes in common.) The roots of these subtrees are **children** of the root. There is an **edge** from a node to each of its children, and a node is said to be the **parent** of its children.

If $n_1, n_2, \ldots, n_k$ is a sequence of nodes in the tree such that n_i is the parent of $n_i + 1$ for $1 \leq i < k$, then this sequence is called a **path** from n_1 to n_k. The **length** of the path is $k - 1$. If there is a path from node R to node M, then R is an **ancestor** of M, and M is a **descendant** of R. Thus, all nodes in the tree are descendants of the root of the tree, while the root is the ancestor of all nodes. The **depth** of a node M in the tree is the length of the path from the root of the tree to M. The **height** of a tree is the depth of the deepest node in the tree. All nodes of depth d are at **level** d in the tree. The root is the only node at level 0, and its depth is 0. A **leaf node** is any node that has two empty children. An **internal node** is any node that has at least one non-empty child.

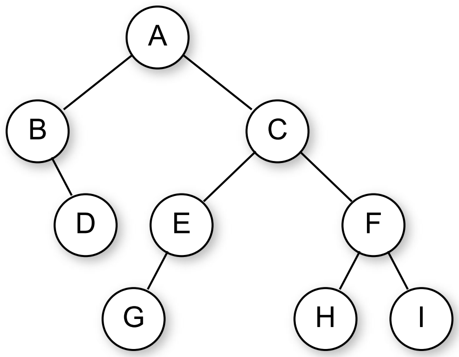


Figure 5.2.1: A binary tree. Node A is the root. Nodes B and C are A's children. Nodes B and D together form a subtree. Node B has two children: Its left child is the empty tree and its right child is D. Nodes A, C, and E are ancestors of G. Nodes D, E, and F make up level 2 of the tree; node A is at level 0. The edges from A to C to E to G form a path of length 3. Nodes D, G, H, and I are leaves. Nodes A, B, C, E, and F are internal nodes. The depth of I is 3. The height of this tree is 3.

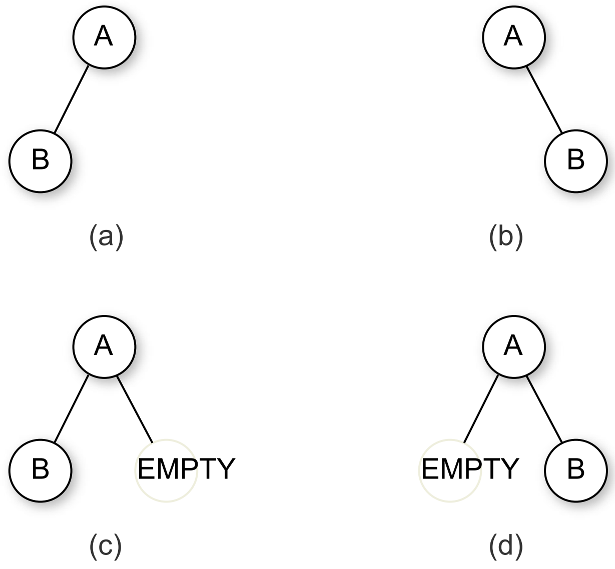


Figure 5.2.2: Two different binary trees. (a) A binary tree whose root has a non-empty left child. (b) A binary tree whose root has a non-empty right child. (c) The binary tree of (a) with the missing right child made explicit. (d) The binary tree of (b) with the missing left child made explicit.

Figure **5.2.1** illustrates the various terms used to identify parts of a binary tree. Figure **5.2.2** illustrates an important point regarding the structure of binary trees. Because *all* binary tree nodes have two children (one or both of which might be empty), the two binary trees of Figure **5.2.2** are *not* the same.

Two restricted forms of binary tree are sufficiently important to warrant special names. Each node in a **full binary tree** is either (1) an internal node with exactly two non-empty children or (2) a leaf. A **complete binary tree** has a restricted shape obtained by starting at the root and filling the tree by levels from left to right. In the complete binary tree of height d, all levels except possibly level d are completely full. The bottom level has its nodes filled in from the left side.

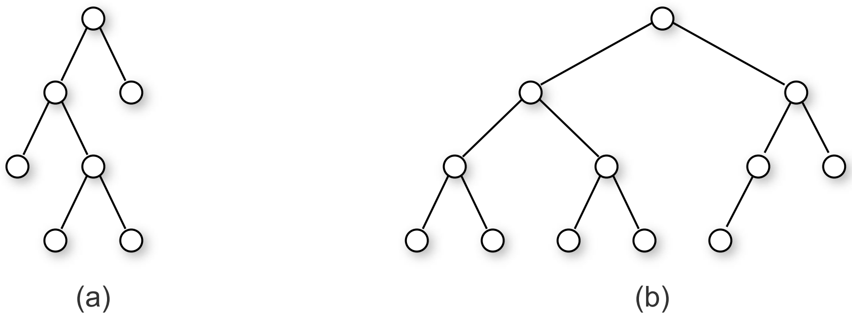


Figure 5.2.3: Examples of full and complete binary trees.

Figure **5.2.3** illustrates the differences between full and complete binary trees. **1** There is no particular relationship between these two tree shapes; that is, the tree of Figure **5.2.3** (a) is full but not complete while the tree of Figure **5.2.3** (b) is complete but not full. The **heap** data structure is an example of a complete binary tree. The **Huffman coding tree** is an example of a full binary tree.

1

While these definitions for full and complete binary tree are the ones most commonly used, they are not universal. Because the common meaning of the words “full” and “complete” are quite similar, there is little that you can do to distinguish between them other than to memorize the definitions. Here is a memory aid that you might find useful: “Complete” is a wider word than “full”, and complete binary trees tend to be wider than full binary trees because each level of a complete binary tree is as wide as possible.

Practicing Tree Definition Summary Questions

Current score: 0 out of 5

Which statement is false?

- ◯ Every non-root node in a binary tree has exactly one parent
- ◯ Every node in a binary tree has exactly two children
- ◯ Every binary tree has at least one node
- ◯ Every non-empty binary tree has exactly one root node
- ◯ None of the above

Answer

Check Answer

Need help?

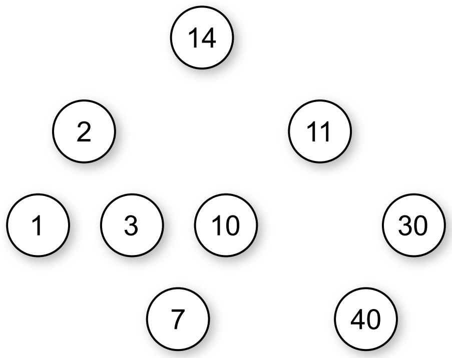
I'd like a hint

5.2.2. Practice Questions

Practicing Tree Definition Questions

Current score: 0 out of 5

How many nodes in the tree have at least one sibling?



Answer

Check Answer

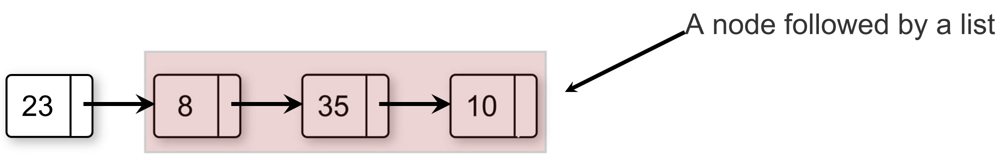
Need help?

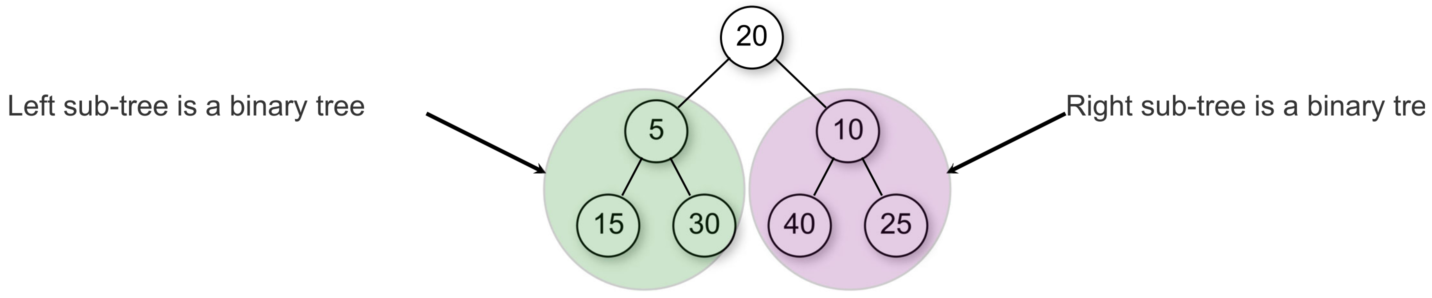
I'd like a hint

5.3. Binary Tree as a Recursive Data Structure

5.3.1. Binary Tree as a Recursive Data Structure

A **recursive data structure** is a data structure that is partially composed of smaller or simpler instances of the same data structure. For example, **linked lists** and **binary trees** can be viewed as recursive data structures. A list is a recursive data structure because a list can be defined as either (1) an empty list or (2) a node followed by a list. A binary tree is typically defined as (1) an empty tree or (2) a node pointing to two binary trees, one its left child and the other one its right child.



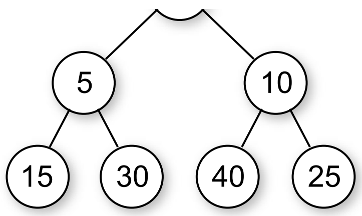

Left sub-tree is a binary tree
20
5
10
15
30
40
25
Right sub-tree is a binary tre

The recursive relationships used to define a structure provide a natural model for any recursive algorithm on the structure.

1 / 8

<< < > >>

Suppose that you want to compute the sum of the values stored in a binary tree.


5
10
15
30
40
25

5.4. The Full Binary Tree Theorem

Some binary tree implementations store data only at the **leaf nodes**, using the **internal nodes** to provide structure to the tree. By definition, a leaf node does not need to store pointers to its (empty) **children**. More generally, binary tree implementations might require some amount of space for internal nodes, and a different amount for leaf nodes. Thus, to compute the space required by such implementations, it is useful to know the minimum and maximum fraction of the nodes that are leaves in a tree containing n internal nodes.

Unfortunately, this fraction is not fixed. A binary tree of n internal nodes might have only one leaf. This occurs when the internal nodes are arranged in a chain ending in a single leaf as shown in Figure **5.4.1**. In this example, the number of leaves is low because each internal node has only one non-empty child. To find an upper bound on the number of leaves for a tree of n internal nodes, first note that the upper bound will occur when each internal node has two non-empty children, that is, when the tree is full. However, this observation does not tell what shape of tree will yield the highest percentage of non-empty leaves. It turns out not to matter, because all full binary trees with n internal nodes have the same number of leaves. This fact allows us to compute the space requirements for a full binary tree implementation whose leaves require a different amount of space from its internal nodes.

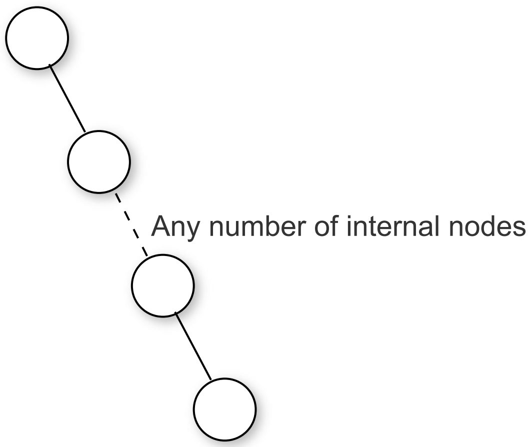


Figure 5.4.1: A tree containing many internal nodes and a single leaf.

Theorem 5.4.1

Full Binary Tree Theorem: The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.

Proof: The proof is by **mathematical induction** on n, the number of internal nodes. This is an example of the style of induction proof where we reduce from an arbitrary instance of size n to an instance of size $n-1$ that meets the induction hypothesis.

i. **Base Cases:** The non-empty tree with zero internal nodes has one leaf node. A full binary tree with one internal node has two leaf nodes. Thus, the base cases for $n = 0$ and $n = 1$ conform to the theorem.

ii. **Induction Hypothesis:** Assume that any full binary tree $\mathbf{T}$ containing $n-1$ internal nodes has n leaves.

iii. **Induction Step:** Given tree $\mathbf{T}$ with n internal nodes, select an internal node I whose children are both leaf nodes. Remove both of I's children, making I a leaf node. Call the new tree $\mathbf{T}'$. $\mathbf{T}'$ has $n-1$ internal nodes. From the induction hypothesis, $\mathbf{T}'$ has n leaves. Now, restore I's two children. We once again have tree $\mathbf{T}$ with n internal nodes. How many leaves does $\mathbf{T}$ have? Because $\mathbf{T}'$ has n leaves, adding the two children yields $n+2$. However, node I counted as one of the leaves in $\mathbf{T}'$ and has now become an internal node. Thus, tree $\mathbf{T}$ has $n+1$ leaf nodes and n internal nodes.

By mathematical induction the theorem holds for all values of $n > 0$.

When analyzing the space requirements for a binary tree implementation, it is useful to know how many empty subtrees a tree contains. A simple extension of the Full Binary Tree Theorem tells us exactly how many empty subtrees there are in *any* binary tree, whether full or not. Here are two approaches to proving the following theorem, and each suggests a useful way of thinking about binary trees.

Theorem 5.4.2

The number of empty subtrees in a non-empty binary tree is one more than the number of nodes in the tree.

Proof 1: Take an arbitrary binary tree $\mathbf{T}$ and replace every empty subtree with a leaf node. Call the new tree $\mathbf{T}'$. All nodes originally in $\mathbf{T}$ will be internal nodes in $\mathbf{T}'$ (because even the leaf nodes of $\mathbf{T}$ have children in $\mathbf{T}'$). $\mathbf{T}'$ is a full binary tree, because every internal node of $\mathbf{T}$ now must have two children in $\mathbf{T}'$, and each leaf node in $\mathbf{T}$ must have two children in $\mathbf{T}'$ (the leaves just added). The Full Binary Tree Theorem tells us that the number of leaves in a full binary tree is one more than the number of internal nodes. Thus, the number of new leaves that were added to create $\mathbf{T}'$ is one more than the number of nodes in $\mathbf{T}$. Each leaf node in $\mathbf{T}'$ corresponds to an empty subtree in $\mathbf{T}$. Thus, the number of empty subtrees in $\mathbf{T}$ is one more than the number of nodes in $\mathbf{T}$.

Proof 2: By definition, every node in binary tree $\mathbf{T}$ has two children, for a total of $2n$ children in a tree of n nodes. Every node except the root node has one parent, for a total of $n-1$ nodes with parents. In other words, there are

05.05 Binary Tree Traversals

Due No Due Date **Points** 4 **Submitting** an external tool

05.05 Binary Tree Traversals

5.5. Binary Tree Traversals

5.5.1. Binary Tree Traversals

Often we wish to process a binary tree by "visiting" each of its nodes, each time performing a specific action such as printing the contents of the node. Any process for visiting all of the nodes in some order is called a **traversal**. Any traversal that lists every node in the tree exactly once is called an **enumeration** of the tree's nodes. Some applications do not require that the nodes be visited in any particular order as long as each node is visited precisely once. For other applications, nodes must be visited in an order that preserves some relationship.

5.5.1.1. Preorder Traversal

For example, we might wish to make sure that we visit any given node *before* we visit its children. This is called a **preorder traversal**.

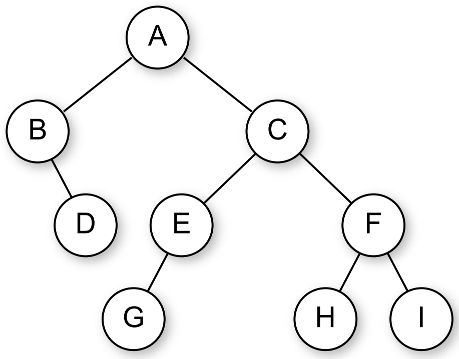


Figure 5.5.1: A binary tree for traversal examples.

Example 5.5.1

The preorder enumeration for the tree of Figure 5.5.1 is **A B D C E G F H I**.

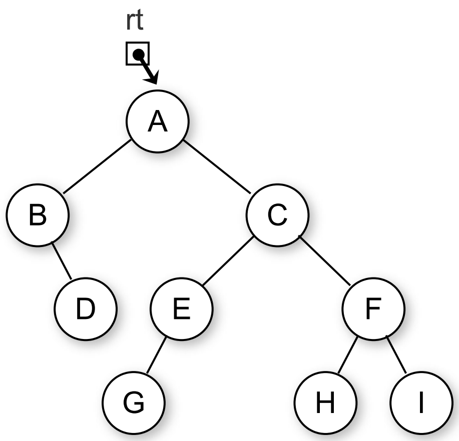
The first node printed is the root. Then all nodes of the left subtree are printed (in preorder) before any node of the right subtree.

1 / 56

<< < > >>

Preorder traversal begins.

```
static void preorder(BinNode rt) {
  if (rt == null) return; // Empty subtree - do nothing
  visit(rt);              // Process root node
  preorder(rt.left());    // Process all nodes in left
  preorder(rt.right());   // Process all nodes in right
}
```



5.5.1.2. Postorder Traversal

Alternatively, we might wish to visit each node only *after* we visit its children (and their subtrees). For example, this would be necessary if we wish to return all nodes in the tree to free store. We would like to delete the children of a node before deleting the node itself. But to do that requires that the children's children be deleted first, and so on. This is called a **postorder traversal**.

Example 5.5.2

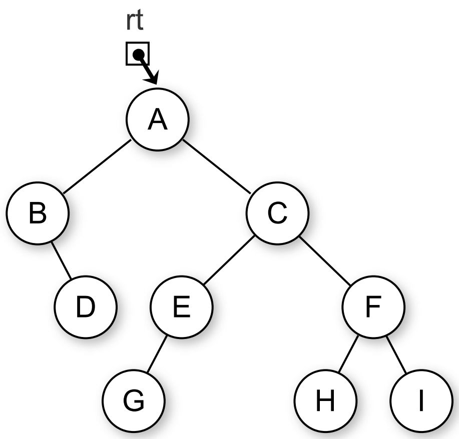
The postorder enumeration for the tree of Figure **5.5.1** is **D B G E H I F C A**.

1 / 56

<< < > >>

Postorder traversal begins.

```
static void postorder(BinNode rt) {
  if (rt == null) return;
  postorder(rt.left());
  postorder(rt.right());
  visit(rt);
}
```



5.5.1.3. Inorder Traversal

An **inorder traversal** first visits the left child (including its entire subtree), then visits the node, and finally visits the right child (including its entire subtree). The **binary search tree** makes use of this traversal to print all nodes in ascending order of value.

Example 5.5.3

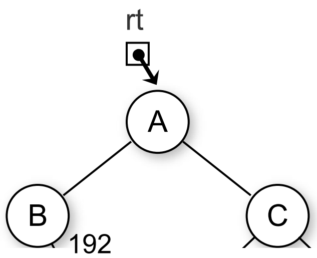
The inorder enumeration for the tree of Figure 5.5.1 is **B D A G E C H F I**.

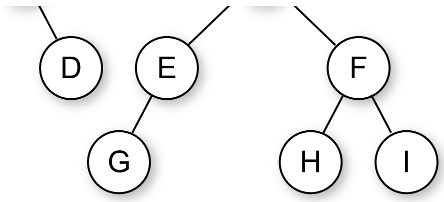
1 / 56

<< < > >>

Inorder traversal begins.

```
static void inorder(BinNode rt) {
  if (rt == null) return;
  inorder(rt.left());
  visit(rt);
  inorder(rt.right());
}
```





5.5.1.4. Implementation

Now we will discuss some implementations for the traversals, but we need to define a node ADT to work with. Just as a linked list is composed of a collection of link objects, a tree is composed of a collection of node objects. Here is an ADT for binary tree nodes, called `BinNode`. This class will be used by some of the binary tree structures presented later. Member functions are provided that set or return the element value, return a pointer to the left child, return a pointer to the right child, or indicate whether the node is a leaf.

Java | Java (Generic) | Toggle Tree Vie

```
interface BinNode { // Binary tree node ADT
  // Get and set the element value
  public Object value();
  public void setValue(Object v);

  // return the children
  public BinNode left();
  public BinNode right();

  // return TRUE if a leaf node, FALSE otherwise
  public boolean isLeaf();
}
```

A traversal routine is naturally written as a recursive function. Its input parameter is a pointer to a node which we will call `rt` because each node can be viewed as the root of a some subtree. The initial call to the traversal function passes in a pointer to the root node of the tree. The traversal function visits `rt` and its children (if any) in the desired order. For example, a preorder traversal specifies that `rt` be visited before its children. This can easily be implemented as follows.

Java | Java (Generic) | Toggle Tree Vie

```
static void preorder(BinNode rt) {
  if (rt == null) return; // Empty subtree - do nothing
  visit(rt);              // Process root node
  preorder(rt.left());    // Process all nodes in left
  preorder(rt.right());   // Process all nodes in right
}
```

Function `preorder` first checks that the tree is not empty (if it is, then the traversal is done and `preorder` simply returns). Otherwise, `preorder` makes a call to `visit`, which processes the root node (i.e., prints the value or performs whatever computation as required by the application). Function `preorder` is then called recursively on the left subtree, which will visit all nodes in that subtree. Finally, `preorder` is called on the right subtree, visiting all nodes in the right subtree. Postorder and inorder traversals are similar. They simply change the order in which the node and its children are visited, as appropriate.

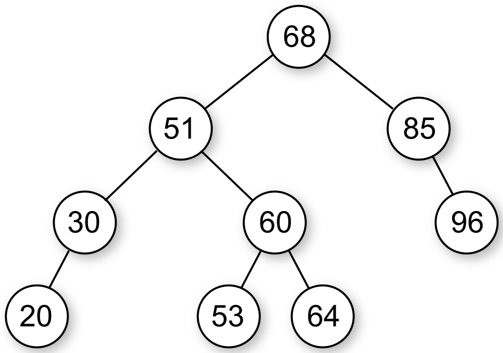
Help Reset Model Answer

Instructions:

Reproduce the behavior of binary tree preorder traversal. Click nodes to indicate the order in which the traversal algorithm would visit them.

```
static void preorder(BinNode rt) {
  if (rt == null) return; // Empty subtree - do nothing
  visit(rt);              // Process root node
  preorder(rt.left());    // Process all nodes in left
  preorder(rt.right());   // Process all nodes in right
}
```

Score: 0 / 9, Points remaining: 9, Points lost: 0



5.5.2. Postorder Traversal Practice

Help Reset Model Answer

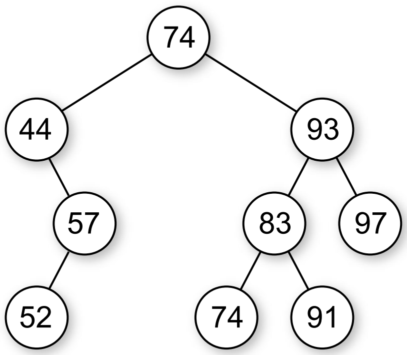
Instructions:

Reproduce the behavior of binary tree postorder traversal. Click nodes to indicate the order in which the traversal algorithm would visit them.

```
static void postorder(BinNode rt) {
  if (rt == null) return;
```

```
    postorder(rt.left());
    postorder(rt.right());
    visit(rt);
  }
```

Score: 0 / 9, Points remaining: 9, Points lost: 0



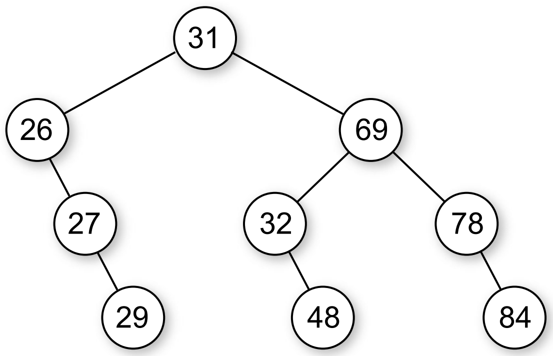
5.5.3. Inorder Traversal Practice

Instructions:

Reproduce the behavior of binary tree inorder traversal. Click nodes to indicate the order in which the traver
algorithm would visit them.

```
static void inorder(BinNode rt) {
  if (rt == null) return;
  inorder(rt.left());
  visit(rt);
  inorder(rt.right());
}
```

Score: 0 / 9, Points remaining: 9, Points lost: 0



5.5.4. Summary Questions

05.06 Implementing Tree Traversals

Due No Due Date **Points** 2 **Submitting** an external tool

05.06 Implementing Tree Traversals

5.6. Implementing Tree Traversals

5.6.1. Implementing Tree Traversals

Recall that any recursive function requires the following:

1. The base case and its action.
2. The recursive case and its action.

In this module, we will talk about some details related to correctly and clearly implementing recursive tree traversals.

5.6.1.1. Base Case

In binary tree traversals, most often the base case is to check if we have an empty tree. A common mistake is to check the child pointers of the current node, and only make the recursive call for a non-null child.

Recall the basic preorder traversal function.

Java | Java (Generic) | Toggle Tree Vie

```
static void preorder(BinNode rt) {
  if (rt == null) return; // Empty subtree - do nothing
  visit(rt);              // Process root node
  preorder(rt.left());    // Process all nodes in left
  preorder(rt.right());   // Process all nodes in right
}
```

Here is an alternate design for the preorder traversal, in which the left and right pointers of the current node are checked so that the recursive call is made only on non-empty children.

Java | Java (Generic) | Toggle Tree Vie

```java
// This is a bad idea
static void preorder2(BinNode rt) {
  visit(rt);
  if (rt.left() != null) preorder2(rt.left());
  if (rt.right() != null) preorder2(rt.right());
}
```

At first it might appear that `preorder2` is more efficient than `preorder`, because it makes only half as many recursive calls (since it won't try to call on a null pointer). On the other hand, `preorder2` must access the left and right child pointers twice as often. The net result is that there is no performance improvement.

Perhaps the writer of `preorder2` wants to protect against the case where the root is `null`. But `preorder2` has an error. While `preorder2` insures that no recursive calls will be made on empty subtrees, it will fail if the orignal call from outside passes in a null pointer. This would occur if the original tree is empty. Since an empty tree is a legitimate input to the initial call on the function, there is no safe way to avoid this case. So it is necessary that the first thing you do on a binary tree traversal is to check that the root is not `null`. If we try to fix `preorder2` by adding this test, then making the tests on the children is completely redundant because the pointer will be checked again in the recursive call.

The design of `preorder2` is inferior to that of `preorder` for a deeper reason as well. Looking at the children to see if they are `null` means that we are worrying too much about something that can be dealt with just as well by the children. This makes the function more complex, which can become a real problem for more complex tree structures. Even in the relatively simple `preorder2` function, we had to write two tests for `null` rather than the one needed by `preorder`. This makes it more complicated than the original version. The key issue is that it is much easier to write a recursive function on a tree when we only think about the needs of the current node. Whenever we can, we want to let the children take care of themselves. In this case, we care that the current node is not `null`, and we care about how to invoke the recursion on the children, but we do **not** have to care about how or when that is done.

5.6.1.2. The Recursive Call

The secret to success when writing a recursive function is to not worry about how the recursive call works. Just accept that it will work correctly. One aspect of this principle is not to worry about checking your children when you don't need to. You should only look at the values of your children if you need to know those values in order to compute some property of the current node. Child values should not be used to decide whether to call them recursviely. Make the call, and let their own base case handle it.

Example 5.6.1

Consider the problem of incrementing the value for each node in a binary tree. The following solution has an error, since it does redundant manipulation to left and the right children of each node.

Toggle Tree View

```java
static void ineff_BTinc(BinNode root) {
 if (root != null) {
   root.setValue((int)(root.value()) + 1);
  if (root.left() != null) {
   root.left().setValue((int)(root.left().value()) + 1);
   ineff_BTinc(root.left().left());
  }
  if (root.right() != null) {
   root.right().setValue((int)(root.right().value()) + 1);
   ineff_BTinc(root.right().right());
  }
 }
}
```

The efficient solution should not explicitly set the children values that way. Changing the value of a node does not depend on the child values. So the function should simply increment the root value, and make recursive calls on the children.

In rare problems, you might need to explicitly check if the children are null or access the children values for each node. For example, you might need to check if all nodes in a tree satisfy the property that each node stores the sum of its left and right children. In this situation you must look at the values of the children to decide something about the current node. You do **not** look at the children to decide whether to make a recursive call.

5.6.2. Binary Tree Increment By One Exercise

X295: Binary Tree Increment By One Exercise

Write a recursive function that increments by one the value for every node in the binary tree pointed at by tree. Assume that nodes store integer values.

Here are methods that you can use on the `BinNode` objects:

```
interface BinNode {
  public int value();
  public void setValue(int v);
  public BinNode left();
  public BinNode right();
  public boolean isLeaf();
}
```

Your Answer:

```
public BinNode BTinc(BinNode root)
{

}

```

Feedback

Your feedback will a
answer.

Check my answer! Reset

05.07 Information Flow in Recursive Functions

Due No Due Date **Points** 16 **Submitting** an external tool

05.07 Information Flow in Recursive Functions

5.7. Information Flow in Recursive Functions

5.7.1. Information Flow in Recursive Functions

Handling information flow in a recursive function can be a challenge. In any given function, we might need to be concerned with either or both of:

1. Passing down the correct information needed by the function to do its work,
2. Returning (passing up) information to the recursive function's caller.

Any given problems might need to do either or both. Here are some examples and exercises.

5.7.1.1. Local

Local traversal involves going to each node in the tree to do some operation. Such functions need no information from the parent (other than a pointer to the current node), and pass no information back. Examples include preorder traversal and incrementing the value of every node by one.

5.7.1.2. Passing Down Information

Slightly more complicated is the situation where every node needs the same piece of information to be passed to it. An example would be incrementing the value for all nodes by some amount. In this case, the value parameter is simply passed on unchanged in all recursive calls.

Many functions need information that changes from node to node. A simple example is a function to set the value for each node of the tree to be its depth. In this case, the depth is passed as a parameter to the function, and each recursive call must adjust that value (by adding one).

5.7.2. Binary Tree Set Depth Exercise

X281: Binary Tree Set Depth Exercise

Write a recursive function to set the value for each node in a binary tree to be its depth then return the mo integer values. On the initial call to `BTsetdepth`, `depth` is 0.

Here are methods that you can use on the `BinNode` objects:

```
interface BinNode {
  public int value();
  public void setValue(int v);
  public BinNode left();
  public BinNode right();
  public boolean isLeaf();
}
```

Your Answer:

```
public BinNode BTsetdepth(BinNode root, int depth)
{

}

```

Check my answer! Reset

Feedback

Your feedback will a
answer.

5.7.3. Collect-and-return

Collect-and-return requires that we communicate information back up the tree to the caller. Simple examples are to count the number of nodes in a tree, or to sum the values of all the nodes.

Example 5.7.1

Example 5.7.1

Consider the problem of counting (and returning) the number of nodes in a binary tree. The key insight is that the total count for any (non-empty) subtree is one for the root plus the counts for the left and right subtrees. Where do left and right subtree counts come from? Calls to function `count` on the subtrees will compute this for us. Thus, we can implement `count` as follows.

Java | Java (Generic) | Toggle Tree View

```
static int count(BinNode rt) {
  if (rt == null) return 0;  // Nothing to count
  return 1 + count(rt.left()) + count(rt.right());
}
```

The following solution is correct but inefficient as it does redundant checks on the left and the right child of each visited node.

Toggle Tree View

```
static int ineff_count(BinNode root) {
  if (root == null) { return 0; }  // Nothing to count
  int count = 0;
  if (root.left() != null) {
    count = 1 + ineff_count(root.left());
  }
  if (root.right() != null) {
    count = 1 + ineff_count(root.right());
  }
  if (root.left() == null && root.right() == null) {
    return 1;
  }
  return 1 + count;
}
```

When you write a recursive function that returns a value, such as counting the number of nodes in the subtree, you have to make sure that your function actually returns a value. A common mistake is to make a recursive call and not capture the returned value. Another common mistake is to not return a value.

1 / 4 &lt;&lt; &lt; &gt; &gt;&gt;

When you write a recursive method that traverses a binary tree, you should avoid the following common mistal

```
static int bad_count(BinNode root) {
  if (root == null) { return 0; }  // Nothing to count
```

```
      bad_count(root.left());
      1 + bad_count(root.left()) + bad_count(root.right());
  }
```

5.7.4. Binary Tree Check Sum Exercise

X286: Binary Tree Check Sum Exercise

Given a binary tree, check if the tree satisfies the property that for each node, the sum of the values of its l
the node's value. If a node has only one child, then the node should have the same value as that child. Lea
property.

Here are methods that you can use on the `BinNode` objects:

```
interface BinNode {
  public int value();
  public void setValue(int v);
  public BinNode left();
  public BinNode right();
  public boolean isLeaf();
}
```

Your Answer:

```
public boolean BTchecksum(BinNode root)
{

}

```

Check my answer! Reset

Feedback

Your feedback will a
answer.

5.7.5. Binary Tree Leaf Nodes Count Exercise

X287: Binary Tree Leaf Nodes Count Exercise

Write a recursive function `int BTleaf(BinNode root)` to count the number of leaf nodes in the binary tre
the `isLeaf()` method in the `BinNode` class to check if a node is a leaf. This is the definition of the `BinNo`

```
interface BinNode {
   public int value();
   public void setValue(int v);
   public BinNode left();
   public BinNode right();
   public boolean isLeaf();
}
```

Complete the `BTleaf` function below.

Your Answer:

```
public int BTleaf(BinNode root)
{

}

```

Check my answer! Reset

Feedback

Your feedback will a
answer.

5.7.6. Binary Tree Sum Nodes Exercise

X283: Binary Tree Sum Nodes Exercise

Write a recursive function `int BTsumall(BinNode root)` that returns the sum of the values for all of the r
`root`. Here are methods that you can use on the `BinNode` objects:

```
interface BinNode {
  public int value();
  public void setValue(int v);
  public BinNode left();
  public BinNode right();
  public boolean isLeaf();
}
```

Write the `BTsumall` function below:

Your Answer:

```
public int BTsumall(BinNode root)
{

}

```

Check my answer! Reset

Feedback

Your feedback will a
answer.

5.7.7. Combining Information Flows

Many functions require both that information be passed in, and that information be passed back. Let’s start with a relatively simple case. If we want to check if some node in the tree has a particular value, that value has to be passed down, and the count has to be passed back up. The downward flow is simple, as the value being checked for never changes. The information passed up has the simple collect-and-return style: Return True if and only if one of the children returns True.

5.7.8. Binary Tree Check Value Exercise

X280: Binary Tree Check Value Exercise

Write a recursive function that returns true if there is a node in the given binary tree with the given value,
tree is **not** a Binary Search Tree.

Here are methods that you can use on the `BinNode` objects:

```
interface BinNode {
  public int value();
  public void setValue(int v);
  public BinNode left();
  public BinNode right();
  public boolean isLeaf();
}
```

Write the `BTcheckval` function below:

Your Answer:

```
public boolean BTcheckval(BinNode root, int value)
{

}

```

Check my answer! Reset

Feedback

Your feedback will a
answer.

5.7.9. Combination Problems

Slightly more complicated problems combine what we have seen so far. Information passing down the tree changes from node to node. Data passed back up the tree uses the collect-and-return paradigm.

5.7.10. Binary Tree Height Exercise

X285: Binary Tree Height Exercise

The height of a binary tree is the length of the path to the deepest node. An empty tree has a height of 0, a 1, and so on. Write a recursive function to find the height of the binary tree pointed at by `root`.

Here are methods that you can use on the `BinNode` objects:

```
interface BinNode {
  public int value();
  public void setValue(int v);
  public BinNode left();
  public BinNode right();
  public boolean isLeaf();
}
```

Your Answer:

```
public int BTheight(BinNode root)
{

}

```

Feedback

Your feedback will ap answer.

Check my answer! Reset

5.7.11. Binary Tree Get Difference Exercise

X290: Binary Tree Get Difference Exercise

Given a binary tree, write a recursive function to return the difference between the sum of all node values a values at even levels. Define the root node to be at level 1.

Here are methods that you can use on the `BinNode` objects:

```
interface BinNode {
  public int value();
  public void setValue(int v);
  public BinNode left();
  public BinNode right();
  public boolean isLeaf();
}
```

Your Answer:

```
public int BTgetdiff(BinNode root)
{

}

```

Feedback

Your feedback will ap answer.

Check my answer! Reset

5.7.12. Binary Tree Has Path Sum Exercise

X282: Binary Tree Has Path Sum Exercise

We define a "root-to-node path" to be any sequence of nodes in a tree starting with the root and proceedin "root-to-node path sum" for that path is the sum of the values for all the nodes (including the root and the an empty tree to contain no root-to-node paths (and so its sum is zero). Define a tree with one node (equiv a root-to-node path consisting of just the root (and so its sum is the value of the root). Given a binary tree tree has some root-to-node path such that adding up all the values along the path equals `sum`. Return fals Here are methods that you can use on the `BinNode` objects:

```
interface BinNode {
  public int value();
  public void setValue(int v);
  public BinNode left();
  public BinNode right();
  public boolean isLeaf();
}
```

Your Answer:

```
public  boolean BTpathsum(BinNode root, int sum)
{

}

```

Feedback

Your feedback will a answer.

Check my answer! Reset

5.8. Binary Tree Node Implementations

5.8.1. Binary Tree Node Implementations

In this module we examine various ways to implement binary tree nodes. By definition, all binary tree nodes have two children, though one or both children can be empty. Binary tree nodes typically contain a value field, with the type of the field depending on the application. The most common node implementation includes a value field and pointers to the two children.

Here is a simple implementation for the `BinNode` interface, which we will name `BSTNode`. Its element type is an Object. When we need to support search structures such as the **Binary Search Tree**, the node will typically store a **key-value pair**. Every `BSTNode` object also has two pointers, one to its left child and another to its right child.

Java | Java (Generic)

```
// Binary tree node implementation: supports comparable objects
class BSTNode<E extends Comparable<? super E>> implements BinNode<E> {
  private E element;           // Element for this node
  private BSTNode<E> left;     // Pointer to left child
  private BSTNode<E> right;    // Pointer to right child

  // Constructors
  BSTNode() { left = right = null; }
  BSTNode(E val) { left = right = null; element = val; }
  BSTNode(E val, BSTNode<E> l, BSTNode<E> r)
    { left = l; right = r; element = val; }

  // Get and set the element value
  public E value() { return element; }
  public void setValue(E v) { element = v; }

  // Get and set the left child
  public BSTNode<E> left() { return left; }
  public void setLeft(BSTNode<E> p) { left = p; }

  // Get and set the right child
  public BSTNode<E> right() { return right; }
  public void setRight(BSTNode<E> p) { right = p; }

  // return TRUE if a leaf node, FALSE otherwise
  public boolean isLeaf() { return (left == null) && (right == null); }
}
```

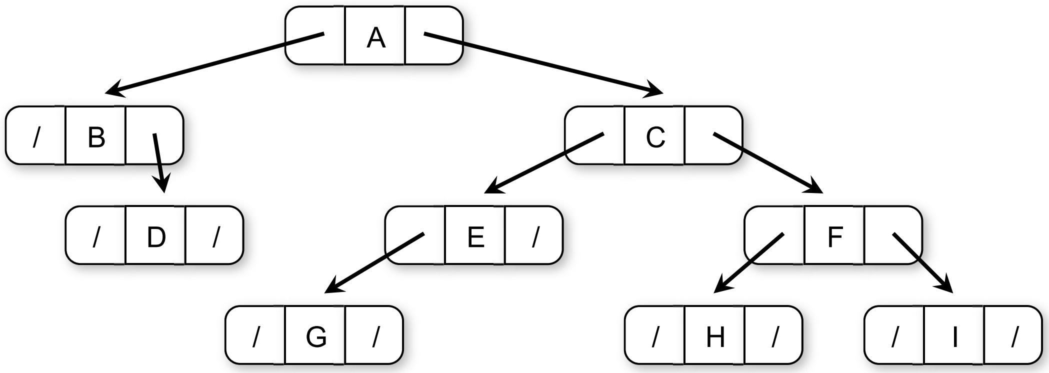


Figure 5.8.1: Illustration of a typical pointer-based binary tree implementation, where each node stores two child pointers and a value.

Some programmers find it convenient to add a pointer to the node's parent, allowing easy upward movement in the tree. Using a parent pointer is somewhat analogous to adding a link to the previous node in a doubly linked list. In practice, the parent pointer is almost always unnecessary and adds to the space overhead for the tree implementation. It is not just a problem that parent pointers take space. More importantly, many uses of the parent pointer are driven by improper understanding of recursion and so indicate poor programming. If you are inclined toward using a parent pointer, consider if there is a more efficient implementation possible.

An important decision in the design of a pointer-based node implementation is whether the same class definition will be used for **leaves** and **internal nodes**. Using the same class for both will simplify the implementation, but might be an inefficient use of space. Some applications require data values only for the leaves. Other applications require one type of value for the leaves and another for the internal nodes. Examples include the **binary trie**, the **PR Quadtree**, the **Huffman coding tree**, and the **expression tree** illustrated by Figure 5.8.2. By definition, only internal nodes have non-empty children. If we use the same node implementation for both internal and leaf nodes, then both must store the child pointers. But it seems wasteful to store child pointers in the leaf nodes. Thus, there are many reasons why it can save space to have separate implementations for internal and leaf nodes.

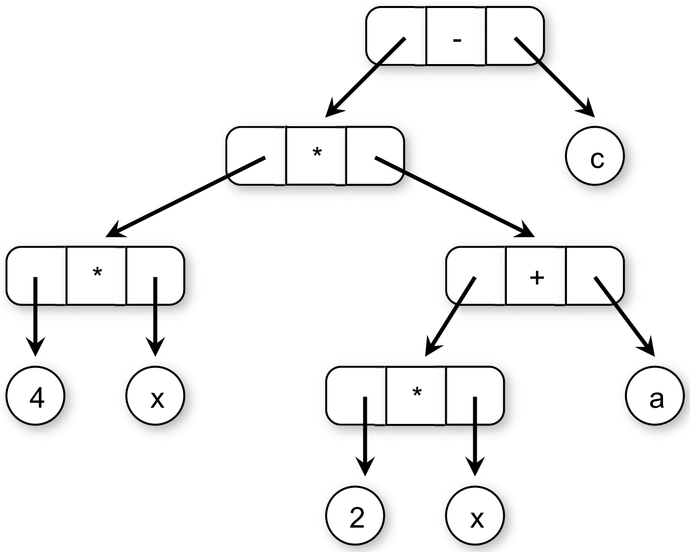


Figure 5.8.2: An expression tree for $4x(2x + a) - c$.

As an example of a tree that stores different information at the leaf and internal nodes, consider the expression tree illustrated by Figure **5.8.2**. The expression tree represents an algebraic expression composed of binary operators such as addition, subtraction, multiplication, and division. Internal nodes store operators, while the leaves store operands. The tree of Figure **5.8.2** represents the expression $4x(2x + a) - c$. The storage requirements for a leaf in an expression tree are quite different from those of an internal node. Internal nodes store one of a small set of operators, so internal nodes could store a small code identifying the operator such as a single byte for the operator's character symbol. In contrast, leaves store variable names or numbers, which is considerably larger in order to handle the wider range of possible values. At the same time, leaf nodes need not store child pointers.

Object-oriented languages allow us to differentiate leaf from internal nodes through the use of a **class hierarchy**. A **base class** provides a general definition for an object, and a **subclass** modifies a base class to add more detail. A base class can be declared for binary tree nodes in general, with subclasses defined for the internal and leaf nodes. The base class in the following code is named `VarBinNode`. It includes a virtual member function named `isLeaf`, which indicates the node type. Subclasses for the internal and leaf node types each implement `isLeaf`. Internal nodes store child pointers of the base class type; they do not distinguish their children's actual subclass. Whenever a node is examined, its version of `isLeaf` indicates the node's subclass.

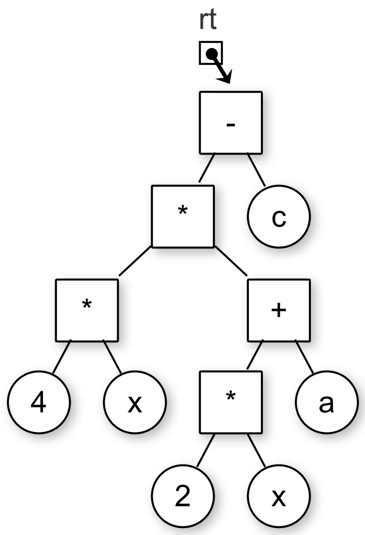
Toggle Tree View

```
// Base class for expression tree nodes
public interface VarBinNode {
  public boolean isLeaf(); // All subclasses must implement
}

/** Leaf node */
public class VarLeafNode implements VarBinNode {
  private String operand;                  // Operand value

  VarLeafNode(String val) { operand = val; }
  public boolean isLeaf() { return true; }
  public String value() { return operand; }
}

// Internal node
public class VarIntlNode implements VarBinNode {
  private VarBinNode left;                 // Left child
  private VarBinNode right;                // Right child
  private Character operator;              // Operator value

  VarIntlNode(Character op, VarBinNode l, VarBinNode r)
    { operator = op; left = l; right = r; }
  public boolean isLeaf() { return false; }
  public VarBinNode leftchild() { return left; }
  public VarBinNode rightchild() { return right; }
  public Character value() { return operator; }
}

// Preorder traversal
public static void traverse(VarBinNode rt) {
  if (rt == null) { return; }          // Nothing to visit
  if (rt.isLeaf()) {                    // Process leaf node
    Visit.VisitLeafNode(((VarLeafNode)rt).value());
  }
  else {                                // Process internal node
    Visit.VisitInternalNode(((VarIntlNode)rt).value());
    traverse(((VarIntlNode)rt).leftchild());
    traverse(((VarIntlNode)rt).rightchild());
  }
}
```

1 / 44

<< < > >>

Preorder traversal begins on pointer-based binary tree.



```
public static void traverse(VarBinNode rt) {
  if (rt == null) { return; }          // Nothing
  if (rt.isLeaf()) {                    // Process
    Visit.VisitLeafNode(((VarLeafNode)rt).value(
  }
  else {                                // Process in
    Visit.VisitInternalNode(((VarIntlNode)rt).va
    traverse(((VarIntlNode)rt).leftchild());
    traverse(((VarIntlNode)rt).rightchild());
  }
}
```

The Expression Tree implementation includes two subclasses derived from class `VarBinNode`, named `LeafNode` and `IntlNode`. Class `IntlNode` can access its children through pointers of type `VarBinNode`. Function `traverse` illustrates the use of these classes. When `traverse` calls method `isLeaf`, the language's runtime environment determines which subclass this particular instance of `rt` happens to be and calls that subclass's version of `isLeaf`. Method `isLeaf` then provides the actual node type to its caller. The other member functions for the derived subclasses are accessed by type-casting the base class pointer as appropriate, as shown in function `traverse`.

5.9. Composite-based Expression Tree

5.9.1. Composite-based Expression Tree

There is another approach that we can take to represent separate leaf and internal nodes, also using a virtual base class and separate node classes for the two types. This is to implement nodes using the **Composite design pattern**. This approach is noticeably different from the **procedural approach** in that the node classes themselves implement the functionality of `traverse`. Here is the implementation. Base class `VarBinNode` declares a member function `traverse` that each subclass must implement. Each subclass then implements its own appropriate behavior for its role in a traversal. The whole traversal process is called by invoking `traverse` on the root node, which in turn invokes `traverse` on its children.

Toggle Tree View

```
/** Base class: Composite */
public interface VarBinNode {
  public boolean isLeaf();
  public void traverse();
}

/** Leaf node: Composite */
public class VarLeafNode implements VarBinNode {
  private String operand;                 // Operand value

  VarLeafNode(String val) { operand = val; }
  public boolean isLeaf() { return true; }
  public String value() { return operand; }

  public void traverse() {
    Visit.VisitLeafNode(operand);
  }
}

/** Internal node: Composite */
public class VarIntlNode implements VarBinNode { // Internal node
  private VarBinNode left;                // Left child
  private VarBinNode right;               // Right child
  private Character operator;             // Operator value

  VarIntlNode(Character op,
                   VarBinNode l, VarBinNode r)
    { operator = op; left = l; right = r; }
  public boolean isLeaf() { return false; }
  public VarBinNode leftchild() { return left; }
  public VarBinNode rightchild() { return right; }
  public Character value() { return operator; }

  public void traverse() {
    Visit.VisitInternalNode(operator);
```

```
      if (left != null) { left.traverse(); }
      if (right != null) { right.traverse(); }
    }
  }

  /** Preorder traversal */
  public static void traverse(VarBinNode rt) {
    if (rt != null) { rt.traverse(); }
  }
```

When comparing the composite implementation to the **procedural approach**, each has advantages and disadvantages. The non-composite approach does not require that the node classes know about the `traverse` function. With this approach, it is easy to add new methods to the tree class that do other traversals or other operations on nodes of the tree. However, we see that `traverse` in the non-composite approach does need to be familiar with each node subclass. Adding a new node subclass would therefore require modifications to the `traverse` function. In contrast, the composite approach requires that any new operation on the tree that requires a traversal also be implemented in the node subclasses. On the other hand, the composite approach avoids the need for the `traverse` function to know anything about the distinct abilities of the node subclasses. Those subclasses handle the responsibility of performing a traversal on themselves. A secondary benefit is that there is no need for `traverse` to explicitly enumerate all of the different node subclasses, directing appropriate action for each. With only two node classes this is a minor point. But if there were many such subclasses, this could become a bigger problem. A disadvantage is that the traversal operation must not be called on a NULL pointer, because there is no object to catch the call. This problem could be avoided by using a **Flyweight** to implement empty nodes. If the composite implementation is for a **full tree**, then it is unnecesary to explicitly check if the children are null.

Typically, the non-composite version would be preferred in this example if `traverse` is a member function of the tree class, and if the node subclasses are hidden from users of that tree class. On the other hand, if the nodes are objects that have meaning to users of the tree separate from their existence as nodes in the tree, then the composite version might be preferred because hiding the internal behavior of the nodes becomes more important.

Another advantage of the composite design is that implementing each node type’s functionality might be easier. This is because you can focus solely on the information passing and other behavior needed by this node type to do its job. This breaks down the complexity that many programmers feel overwhelmed by when dealing with complex information flows related to recursive processing.

05.10 Binary Tree Space Requirements

Due No Due Date **Points** 1 **Submitting** an external tool

05.10 Binary Tree Space Requirements

5.10. Binary Tree Space Requirements

5.10.1. Binary Tree Space Requirements

This module presents techniques for calculating the amount of **overhead** required by a **binary tree**, based on its node implementation. Recall that overhead is the amount of space necessary to maintain the data structure. In other words, it is any space not used to store data records. The amount of overhead depends on several factors including which nodes store data values (all nodes, or just the leaves), whether the leaves store child pointers, and whether the tree is a **full binary tree**.

In a simple **pointer-based implementation for binary tree nodes**, every node has two pointers to its children (even when the children are NULL). This implementation requires total space amounting to $n(2P + D)$ for a tree of n nodes. Here, P stands for the amount of space required by a pointer, and D stands for the amount of space required by a data value. The total overhead space will be $2Pn$ for the entire tree. Thus, the overhead fraction will be $2P/(2P + D)$. The actual value for this expression depends on the relative size of pointers versus data fields. If we arbitrarily assume that $P = D$, then a binary tree has about two thirds of its total space taken up in overhead. Worse yet, the Full Binary Tree Theorem tells us that about half of the pointers are “wasted” NULL values that serve only to indicate tree structure, but which do not provide access to new data.

In many languages (such as Java or JavaScript), the most typical implementation is not to store any actual data in a node, but rather a pointer to the data record. In this case, each node will typically store three pointers, all of which are overhead, resulting in an overhead fraction of $3P/(3P + D)$.

If only leaves store data values, then the fraction of total space devoted to overhead depends on whether the tree is full. If the tree is not full, then conceivably there might only be one leaf node at the end of a series of internal nodes. Thus, the overhead can be an arbitrarily high percentage for non-full binary trees. The overhead fraction drops as the tree becomes closer to full, being lowest when the tree is truly full. In this case, about one half of the nodes are internal.

Great savings can be had by eliminating the pointers from leaf nodes in full binary trees. Again assume the tree stores a pointer to the data field. Because about half of the nodes are leaves and half internal nodes, and because only internal nodes now have child pointers, the overhead fraction in this case will be approximately

$$\frac{\frac{n}{2}(2P)}{\frac{n}{2}(2P) + Dn} = \frac{P}{P + D}$$

If $P = D$, the overhead drops to about one half of the total space. However, if only leaf nodes store useful information, the overhead fraction for this implementation is actually three quarters of the total space, because half of the “data” space is unused.

If a full binary tree needs to store data only at the leaf nodes, a better implementation would have the internal nodes store two pointers and no data field while the leaf nodes store only a pointer to the data field. This implementation requires

$$\frac{n}{2}2P + \frac{n}{2}(P + D)$$

units of space. If $P = D$, then the overhead is $3P/(3P + D) = 3/4$. It might seem counter-intuitive that the overhead ratio has gone up while the total amount of space has gone down. The reason is because we have changed our definition of “data” to refer only to what is stored in the leaf nodes, so while the overhead fraction is higher, it is from a total storage requirement that is lower.

There is one serious flaw with this analysis. When using separate implementations for internal and leaf nodes, there must be a way to distinguish between the node types. When separate node types are implemented via Java subclasses, the runtime environment stores information with each object allowing it to determine, for example, the correct subclass to use when the `isLeaf` virtual function is called. Thus, each node requires additional space. Only one bit is truly necessary to distinguish the two possibilities. In rare applications where space is a critical resource, implementors can often find a spare bit within the node’s value field in which to store the node type indicator. An alternative is to use a spare bit within a node pointer to indicate node type. For example, this is often possible when the compiler requires that structures and objects start on word boundaries, leaving the last bit of a pointer value always zero. Thus, this bit can be used to store the node-type flag and is reset to zero before the pointer is dereferenced. Another alternative when the leaf value field is smaller than a pointer is to replace the pointer to a leaf with that leaf’s value. When space is limited, such techniques can make the difference between success and failure. In any other situation, such “bit packing” tricks should be avoided because they are difficult to debug and understand at best, and are often machine dependent at worst.

05.11 Binary Search Trees

Due No Due Date **Points** 4 **Submitting** an external tool

05.11 Binary Search Trees

5.11. Binary Search Trees

5.11.1. Binary Search Tree Definition

A **binary search tree** (**BST**) is a **binary tree** that conforms to the following condition, known as the **binary search tree property**. All **nodes** stored in the left subtree of a node whose **key** value is K have key values less than or equal to K. All nodes stored in the right subtree of a node whose key value is K have key values greater than K. Figure **5.11.1** shows two BSTs for a collection of values. One consequence of the binary search tree property is that if the BST nodes are printed using an **inorder traversal**, then the resulting enumeration will be in sorted order from lowest to highest.

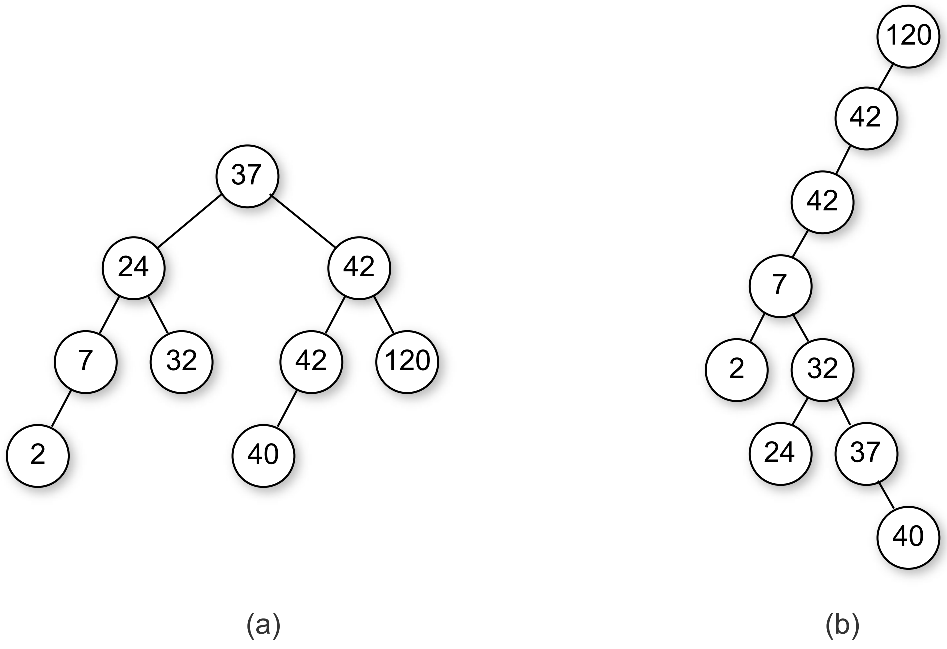


Figure 5.11.1: Two Binary Search Trees for a collection of values. Tree (a) results if values are inserted in the order 37, 24, 42, 7, 2, 40, 42, 32, 120. Tree (b) results if the same values are inserted in the order 120, 42, 42, 7, 2, 32, 37, 24, 40.

Here is a class declaration for the BST. Recall that there are various ways to deal with **keys** and **comparing records** Three typical approaches are **key-value pairs**, a special comparison method such as using the

`Comparator` class, and passing in a **comparator function**. Our BST implementation will require that records implement the `Comparable` interface.

Java | Java (Generic) | Toggle Tree Vie

```
// Binary Search Tree implementation
class BST {
  private BSTNode root; // Root of the BST
  private int nodecount; // Number of nodes in the BST

  // constructor
  BST() { root = null; nodecount = 0; }

  // Reinitialize tree
  public void clear() { root = null; nodecount = 0; }

  // Insert a record into the tree.
  // Records can be anything, but they must be Comparable
  // e: The record to insert.
  public void insert(Comparable e) {
    root = inserthelp(root, e);
    nodecount++;
  }

  // Remove a record from the tree
  // key: The key value of record to remove
  // Returns the record removed, null if there is none.
  public Comparable remove(Comparable key) {
    Comparable temp = findhelp(root, key); // First find it
    if (temp != null) {
      root = removehelp(root, key); // Now remove it
      nodecount--;
    }
    return temp;
  }

  // Return the record with key value k, null if none exists
  // key: The key value to find
  public Comparable find(Comparable key) { return findhelp(root, key); }

  // Return the number of records in the dictionary
  public int size() { return nodecount; }
```

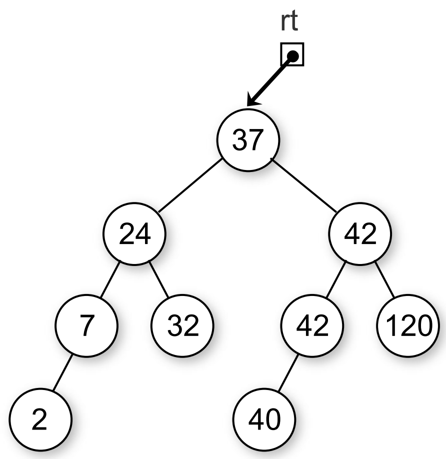
5.11.1.1. BST Search

The first operation that we will look at in detail will find the record that matches a given key. Notice that in the BST class, public member function `find` calls private member function `findhelp`. Method `find` takes the search key as an explicit parameter and its BST as an implicit parameter, and returns the record that matches the key. However, the find operation is most easily implemented as a recursive function whose parameters are the root of a subtree the find operation is most easily implemented as a recursive function whose parameters are the root of a subtree and the search key. Member `findhelp` has the desired form for this recursive subroutine and is implemented as follows.

1 / 17

<< < > >>

Consider searching for the record with key value 32 in this tree. We call the findhelp method with a pointer t root (the node with key value 37).



```
private Comparable findhelp(BSTNode rt, Comparable
  if (rt == null) return null;
  if (rt.value().compareTo(key) > 0)
    return findhelp(rt.left(), key);
  else if (rt.value().compareTo(key) == 0)
    return rt.value();
  else return findhelp(rt.right(), key);
}
```

Undo Reset Model Answer Grade

Instructions:

Use the BST search algorithm to find the key given in the exercise. Starting with the root, click on an empty reveal its value and its children. Work your way down the tree as the search algorithm would.

```
private Comparable findhelp(BSTNode rt, Comparable key) {
  if (rt == null) return null;
  if (rt.value().compareTo(key) > 0)
    return findhelp(rt.left(), key);
  else if (rt.value().compareTo(key) == 0)
    return rt.value();
  else return findhelp(rt.right(), key);
}
```

5.11.2. BST Insert

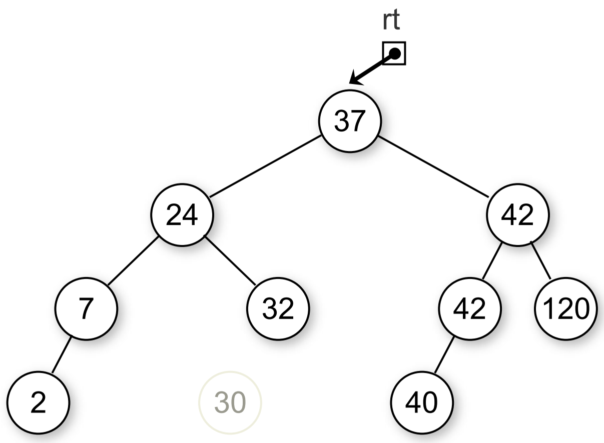
Now we look at how to insert a new node into the BST.

1 / 19

<< < > >>

Consider inserting a record with key value 30 in this tree. We call the findhelp method with a pointer to the BS node with value 37).

```
private BSTNode inserthelp(BSTNode rt, Comparable e) {
  if (rt == null) return new BSTNode(e);
  if (rt.value().compareTo(e) >= 0)
    rt.setLeft(inserthelp(rt.left(), e));
  else
    rt.setRight(inserthelp(rt.right(), e));
  return rt;
}
```



Note that, except for the last node in the path, `inserthelp` will not actually change the child pointer for any of the nodes that are visited. In that sense, many of the assignments seem redundant. However, the cost of these additional assignments is worth paying to keep the insertion process simple. The alternative is to check if a given assignment is necessary, which is probably more expensive than the assignment!

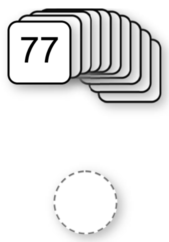
We have to decide what to do when the node that we want to insert has a key value equal to the key of some node already in the tree. If during insert we find a node that duplicates the key value to be inserted, then we have two options. If the application does not allow nodes with equal keys, then this insertion should be treated as an error (or ignored). If duplicate keys are allowed, our convention will be to insert the duplicate in the left subtree.

The shape of a BST depends on the order in which elements are inserted. A new element is added to the BST as a new leaf node, potentially increasing the depth of the tree. Figure **5.11.1** illustrates two BSTs for a collection of values. It is possible for the BST containing n nodes to be a chain of nodes with height n. This would happen if, for example, all elements were inserted in sorted order. In general, it is preferable for a BST to be as shallow as possible. This keeps the average cost of a BST operation low.

Undo | Reset | Model Answer | Grade

Instructions:

Use the BST Insert algorithm to insert values as they are shown at the top. Click on any empty node in the t
place the value to be inserted there. Remember that equal values go to the left.

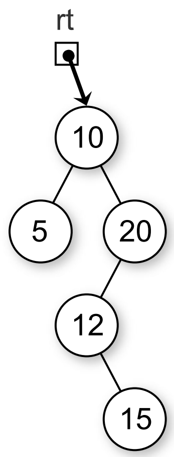


5.11.3. BST Remove

Removing a node from a BST is a bit trickier than inserting a node, but it is not complicated if all of the possible cases are considered individually. Before tackling the general node removal process, we will first see how to remove from a given subtree the node with the largest key value. This routine will be used later by the general node removal function.

1 / 6

To remove the node with the maximum key value from a subtree, first find that node by starting at the subtre
continuously move down the right link until there is no further right link to follow.



```
// Delete the maximum valued element in a subtree
private BSTNode deletemax(BSTNode rt) {
  if (rt.right() == null) return rt.left();
  rt.setRight(deletemax(rt.right()));
  return rt;
}
```

The return value of the `deletemax` method is the subtree of the current node with the maximum-valued node in the subtree removed. Similar to the `inserthelp` method, each node on the path back to the root has its right child pointer reassigned to the subtree resulting from its call to the `deletemax` method.

A useful companion method is `getmax` which returns a pointer to the node containing the maximum value in the subtree.

Java | Java (Generic) | Toggle Tree Vie

```
// Get the maximum valued element in a subtree
private BSTNode getmax(BSTNode rt) {
  if (rt.right() == null) return rt;
  return getmax(rt.right());
}
```

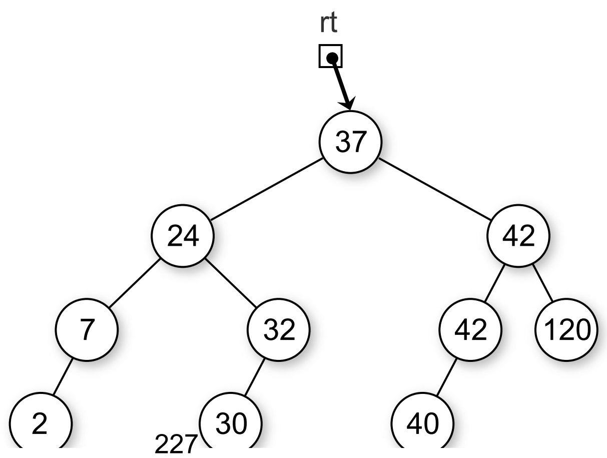
Now we are ready for the `removehelp` method. Removing a node with given key value R from the BST requires that we first find R and then remove it from the tree. So, the first part of the remove operation is a search to find R. Once R is found, there are several possibilities. If R has no children, then R's parent has its pointer set to NULL. If R has one child, then R's parent has its pointer set to R's child (similar to `deletemax`). The problem comes if R has two children. One simple approach, though expensive, is to set R's parent to point to one of R's subtrees, and then reinsert the remaining subtree's nodes one at a time. A better alternative is to find a value in one of the subtrees that can replace the value in R.

Thus, the question becomes: Which value can substitute for the one being removed? It cannot be any arbitrary value, because we must preserve the BST property without making major changes to the structure of the tree. Which value is most like the one being removed? The answer is the least key value greater than the one being removed, or else the greatest key value less than (or equal to) the one being removed. If either of these values replace the one being removed, then the BST property is maintained.

1 / 40 << < > >>

Let's look a few examples for removehelp. We will start with an easy case. Let's see what happens when we from this tree.

```
private BSTNode removehelp(BSTNode rt, Comparable key) {
  if (rt == null) return null;
  if (rt.value().compareTo(key) > 0)
    rt.setLeft(removehelp(rt.left(), key));
  else if (rt.value().compareTo(key) < 0)
    rt.setRight(removehelp(rt.right(), key));
  else { // Found it
    if (rt.left() == null) return rt.right();
    else if (rt.right() == null) return rt.left();
    else { // Two children
      BSTNode temp = getmax(rt.left());
      rt.setValue(temp.value());
      rt.setLeft(deletemax(rt.left()));
    }
  }
  return rt;
}
```



When duplicate node values do not appear in the tree, it makes no difference whether the replacement is the greatest value from the left subtree or the least value from the right subtree. If duplicates are stored in the left subtree, then we must select the replacement from the *left* subtree. 1 To see why, call the least value in the right subtree L. If multiple nodes in the right subtree have value L, selecting L as the replacement value for the root of the subtree will result in a tree with equal values to the right of the node now containing L. Selecting the greatest value from the left subtree does not have a similar problem, because it does not violate the Binary Search Tree Property if equal values appear in the left subtree.

1

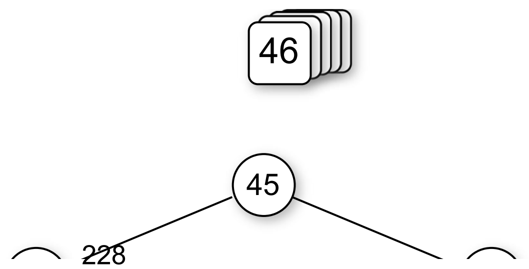
Alternatively, if we prefer to store duplicate values in the right subtree, then we must replace a deleted node with the least value from its right subtree.

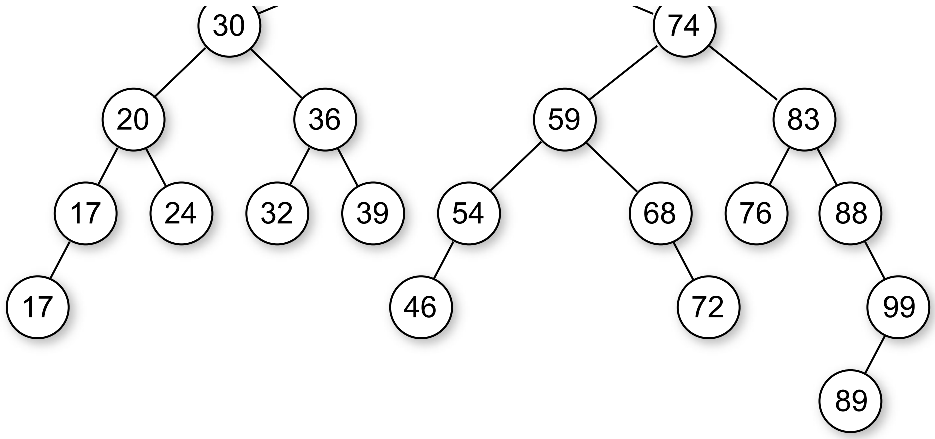
Reset Model Answer About

Instructions:

Use the BST Remove algorithm to remove values as they are shown at the top. Click on any node in the tree to erase its value. If necessary, you can click on another node in the tree to move that value to a node whose value you previously erased. Remember that equal values go to the left, so when necessary we will replace a node with the greatest value on its left side.

Score: 0 / 8, Points remaining: 8, Points lost: 0





5.11.4. BST Analysis

The cost for `findhelp` and `inserthelp` is the depth of the node found or inserted. The cost for `removehelp` is the depth of the node being removed, or in the case when this node has two children, the depth of the node with smallest value in its right subtree. Thus, in the worst case, the cost for any one of these operations is the depth of the deepest node in the tree. This is why it is desirable to keep BSTs **balanced**, that is, with least possible height. If a binary tree is balanced, then the height for a tree of n nodes is approximately $\log n$. However, if the tree is completely unbalanced, for example in the shape of a linked list, then the height for a tree with n nodes can be as great as n. Thus, a balanced BST will in the average case have operations costing $\Theta(\log n)$, while a badly unbalanced BST can have operations in the worst case costing $\Theta(n)$. Consider the situation where we construct a BST of n nodes by inserting records one at a time. If we are fortunate to have them arrive in an order that results in a balanced tree (a "random" order is likely to be good enough for this purpose), then each insertion will cost on average $\Theta(\log n)$, for a total cost of $\Theta(n \log n)$. However, if the records are inserted in order of increasing value, then the resulting tree will be a chain of height n. The cost of insertion in this case will be $\sum_{i=1}^{n} i = \Theta(n^2)$.

Traversing a BST costs $\Theta(n)$ regardless of the shape of the tree. Each node is visited exactly once, and each child pointer is followed exactly once.

Below is an example traversal, named `printhelp`. It performs an inorder traversal on the BST to print the node values in ascending order.

Java | **Java (Generic)** | Toggle Tree Vie

```
private void printhelp(BSTNode rt) {
  if (rt == null) return;
  printhelp(rt.left());
  printVisit(rt.value());
  printhelp(rt.right());
}
```

While the BST is simple to implement and efficient when the tree is balanced, the possibility of its being unbalanced is a serious liability. There are techniques for organizing a BST to guarantee good performance. Two examples are the **AVL tree** and the **splay tree**. There also exist other types of search trees that are guaranteed to remain balanced, such as the **2-3 Tree**.

5.12. Dictionary Implementation Using a BST

A simple implementation for the **Dictionary** ADT can be based on **sorted** or **unsorted lists**. When implementing the dictionary with an unsorted list, inserting a new record into the dictionary can be performed quickly by putting it at the end of the list. However, searching an unsorted list for a particular record requires $\Theta(n)$ time in the average case. For a large database, this is probably much too slow. Alternatively, the records can be stored in a sorted list. If the list is implemented using a **linked list**, then no speedup to the search operation will result from storing the records in sorted order. On the other hand, if we use a sorted **array-based list** to implement the dictionary, then **binary search** can be used to find a record in only $\Theta(\log n)$ time. However, insertion will now require $\Theta(n)$ time on average because, once the proper location for the new record in the sorted list has been found, many records might be shifted to make room for the new record.

Is there some way to organize a collection of records so that inserting records and searching for records can both be done quickly? We can do this with a **binary search tree** (**BST**). The advantage of using the BST is that all major operations (insert, search, and remove) are $\Theta(\log n)$ in the average case. Of course, if the tree is badly balanced, then the cost can be as bad as $\Theta(n)$.

Here is an implementation for the Dictionary interface, using a BST to store the records.

Toggle Tree View

```
// Dictionary implementation using BST
// This uses KVPair to manage the key/value pairs
public class BSTDict implements Dictionary {
  private BST theBST; // The BST that stores the records

  // constructor
  BSTDict() { theBST = new BST(); }

  // Reinitialize dictionary
  public void clear() { theBST = new BST(); }

  // Insert a record
  // k: the key for the record being inserted.
  // e: the record being inserted.
  public void insert(Comparable k, Object e) {
    theBST.insert(new KVPair(k, e));
  }

  // Remove and return a record.
  // k: the key of the record to be removed.
  // Return a maching record. If multiple records match "k", remove
  // an arbitrary one. Return null if no record with key "k" exists.
  public Object remove(Comparable k) {
    Object temp = theBST.remove(k);
    if (temp == null) { return temp; }
    else { return ((KVPair)temp).value(); }
  }
  // Remove and return an arbitrary record from dictionary.
  // Return the record removed, or null if none exists.
  public Object removeAny() {
    if (theBST.size() == 0) { return null; }
    Object temp = theBST.remove(((KVPair)(theBST.root().value())).key());
    return ((KVPair)temp).value();
  }

  // Return a record matching "k" (null if none exists).
  // If multiple records match, return an arbitrary one.
  // k: the key of the record to find
  public Object find(Comparable k) {
    Object temp = theBST.find(k);
    if (temp == null) { return temp; }
    else { return ((KVPair)temp).value() };
  }

  // Return the number of records in the dictionary.
  public int size() {
    return theBST.size();
  }
}
```

05.13 Binary Tree Guided Information Flow

Due No Due Date **Points** 2 **Submitting** an external tool

05.13 Binary Tree Guided Information Flow

5.13. Binary Tree Guided Information Flow

5.13.1. Binary Tree Guided Information Flow

When writing a recursive method to solve a problem that requires traversing a binary tree, we want to make sure that we are visiting the required nodes (no more and no less).

So far, we have seen several tree traversals that visited every node of the tree. We also saw the BST search, insert, and remove routines, that each go down a single path of the tree. **Guided traversal** refers to a problem that does not require visiting every node in the tree, though it typically requires looking at more than one path through the tree. This means that the recursive function is making some decision at each node that sometimes lets it avoid visiting one or both of its children. The decision is typically based on the value of the current node. Many problems that require information flow on binary search trees are “guided” in this way.

Example 5.13.1

An extreme example is finding the minimum value in a BST. A bad solution to this problem would visit every node of the tree. However, we can take advantage of the BST property to avoid visiting most nods in the tree. You know that the values greater than the root are always in the right subtree, and those values less than the root are in the left subtree. Thus, at each node we need only visit the left subtree until we reach a leaf node.

Here is a problem that typically needs to visit more than just a single path, but not all of the nodes.

1 / 12 << < > >>

Suppose that you want to write a recursive function named range that, given a root to a BST, a key value min, value max, returns the number of nodes having key values that fall between min and max. Function range sho few nodes in the BST as possible. An inefficient solution is shown.

```
int range(BSTNode root , int min , int max) {
  if(root == null)
    return 0;
```

```
            return 0;
        int result = 0;
        if ((min <= (Integer)root.element()) && (max >= (Integer)root.element()))
          result = result + 1;
        result += range(root.left(), min, max);
        result += range(root.right(), min, max);
        return result;
    }
```

5.13.2. Binary Search Tree Small Count Exercise

X279: Binary Search Tree Small Count Exercise

Write a recursive function `BSTsmallcount` that, given a BST and a value `key`, returns the number of nodes function should visit as few nodes in the BST as possible.

Here are methods that you can use on the `BinNode` objects:

```
interface BinNode {
   public int value();
   public void setValue(int v);
   public BinNode left();
   public BinNode right();
   public boolean isLeaf();
}
```

Your Answer:

```
public int BSTsmallcount(BinNode root, int key)
{

}

```

Feedback

Your feedback will a answer.

Check my answer! Reset

05.14 Multiple Binary Trees

Due No Due Date **Points** 6 **Submitting** an external tool

05.14 Multiple Binary Trees

5.14. Multiple Binary Trees

5.14.1. Mirror Image Binary Trees Exercise

X288: Mirror Image Binary Trees Exercise

Given two binary trees, return true if and only if they are mirror images of each other. Note that two empty

Here are methods that you can use on the `BinNode` objects:

```
interface BinNode {
   public int value();
   public void setValue(int v);
   public BinNode left();
   public BinNode right();
   public boolean isLeaf();
}
```

Your Answer:

```
public  boolean MBTmirror(BinNode root1, BinNode root2)
{

}
```

Check my answer! Reset

Feedback

Your feedback will a
answer.

5.14.2. Same Binary Tree Exercise

X284: Same Binary Tree Exercise

Given two binary trees, return true if they are identical (they have nodes with the same values, arranged in

Here are methods that you can use on the `BinNode` objects:

```
interface BinNode {
  public int value();
  public void setValue(int v);
  public BinNode left();
  public BinNode right();
  public boolean isLeaf();
}
```

Your Answer:

```
public  boolean MBTsame(BinNode root1, BinNode root2)
{

}

```

Feedback

Your feedback will a

answer.

Check my answer! Reset

5.14.3. Structurally Identical Binary Trees Exercise

X289: Structurally Identical Binary Trees Exercis

Given two binary trees, return true if and only if they are structurally identical (they have the same shape, b
values).

Here are methods that you can use on the `BinNode` objects:

```
interface BinNode {
  public int value();
  public void setValue(int v);
  public BinNode left();
  public BinNode right();
  public boolean isLeaf();
}
```

Your Answer:

```
public  boolean MBTstructure(BinNode root1, BinNode root2)
{

}

```

Check my answer! Reset

Feedback

Your feedback will a
answer.

5.15. A Hard Information Flow Problem

Sometimes, passing the right information up and down the tree to control a recursive function gets complicated. The information flow itself is simple enough, but deciding what to pass might be tricky.

A more difficult example is illustrated by the following problem. Given an arbitrary binary tree we wish to determine if, for every node A, are all nodes in A's left subtree less than the value of A, and are all nodes in A's right subtree greater than the value of A? (This happens to be the definition for a binary search tree.) Unfortunately, to make this decision we need to know some context that is not available just by looking at the node's parent or children.

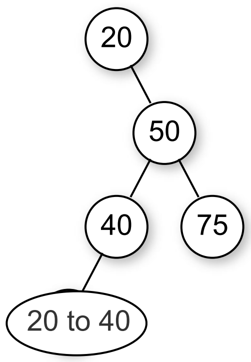


Figure 5.15.1: To be a binary search tree, the left child of the node with value 40 must have a value between 20 and 40.

As shown by Figure **5.15.1**, it is not enough to verify that A's left child has a value less than that of A, and that A's right child has a greater value. Nor is it enough to verify that A has a value consistent with that of its parent. In fact, we need to know information about what range of values is legal for a given node. That information might come from any of the node's ancestors. Thus, relevant range information must be passed down the tree. We can implement this function as follows.

Java | **Java (Generic)** | Toggle Tree Vie

```java
static boolean checkBST(BSTNode rt, Comparable low, Comparable high) {
  if (rt == null) return true; // Empty subtree
  Comparable rootval = rt.value();
  if ((rootval.compareTo(low) <= 0) || (rootval.compareTo(high) > 0))
    return false; // Out of range
  if (!checkBST(rt.left(), low, rootval))
    return false; // Left side failed
  return checkBST(rt.right(), rootval, high);
}
```

05.16 Array Implementation for Complete Binary Trees

Due No Due Date **Points** 1 **Submitting** an external tool

05.16 Array Implementation for Complete Binary Trees

5.16. Array Implementation for Complete Binary Trees

5.16.1. Array Implementation for Complete Binary Trees

From the **full binary tree theorem**, we know that a large fraction of the space in a typical binary tree node implementation is devoted to structural **overhead**, not to storing data. This module presents a simple, compact implementation for **complete binary trees**. Recall that complete binary trees have all levels except the bottom filled out completely, and the bottom level has all of its nodes filled in from left to right. Thus, a complete binary tree of n nodes has only one possible shape. You might think that a complete binary tree is such an unusual occurrence that there is no reason to develop a special implementation for it. However, the complete binary tree has practical uses, the most important being the **heap** data structure. Heaps are often used to implement **priority queues** and for **external sorting algorithms**.

We begin by assigning numbers to the node positions in the complete binary tree, level by level, from left to right as shown in Figure **5.16.1**. An array can store the tree's data values efficiently, placing each data value in the array position corresponding to that node's position within the tree. The table lists the array indices for the children, parent, and siblings of each node in Figure **5.16.1**.

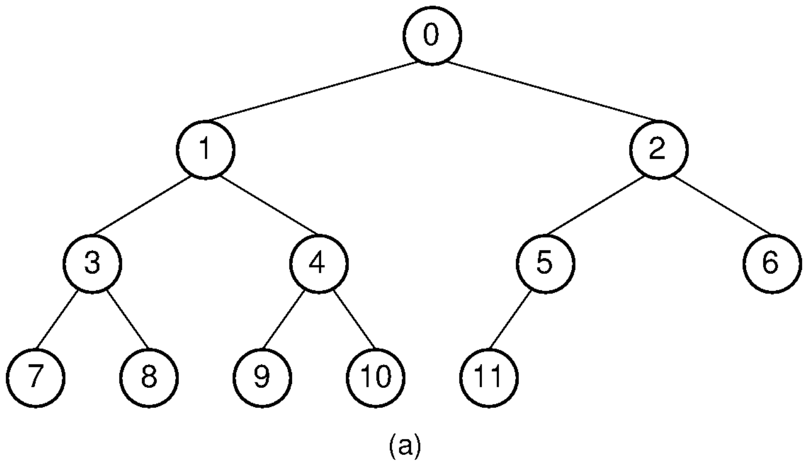


Figure 5.16.1: A complete binary tree of 12 nodes, numbered starting from 0.

Here is a table that lists, for each node position, the positions of the parent, sibling, and children of the node.

Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	$--$	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	$--$	$--$	$--$	$--$	$--$	$--$
Right Child	2	4	6	8	10	$--$	$--$	$--$	$--$	$--$	$--$	$--$
Left Sibling	$--$	$--$	1	$--$	3	$--$	5	$--$	7	$--$	9	$--$
Right Sibling	$--$	2	$--$	4	$--$	6	$--$	8	$--$	10	$--$	$--$

Looking at the table, you should see a pattern regarding the positions of a node's relatives within the array. Simple formulas can be derived for calculating the array index for each relative of a node R from R's index. No explicit pointers are necessary to reach a node's left or right child. This means there is no overhead to the array implementation if the array is selected to be of size n for a tree of n nodes.

The formulae for calculating the array indices of the various relatives of a node are as follows. The total number of nodes in the tree is n. The index of the node in question is r, which must fall in the range 0 to $n-1$.

Parent(r) $= \lfloor (r-1)/2 \rfloor$ if $r \neq 0$.

Left child(r) $= 2r+1$ if $2r+1 < n$.

Right child(r) $= 2r+2$ if $2r+2 < n$.

Left sibling(r) $= r-1$ if r is even and $r \neq 0$.

Right sibling(r) $= r+1$ if r is odd and $r+1 < n$.

Khan.randRange(0, 11) ["parent", "left child", "right child", "left sibling", "right sibling"] Khan.randRange(0, 4) completeFIB.genAnswer(node, randrel)

WARNING! Read the conditions for the problems in this set very carefully! The type of relation and node can change from problem to problem.

For a complete binary tree with nodes labeled as in the figure above, what is the *relation[randrel]* of the node labeled *node*? Type in the node number of the *relation[randrel]*, or if one does not exist, then type NONE.

ANS

To find the parent of node `R`, calculate `\lfloor (R - 1)/2 \rfloor`. If `R = 0`, then it has no parent.

To find the left child of node `R`, calculate `2R + 1`. If `2R + 1 \geq N`, then `R` has no left child.

To find the right child of node `R`, calculate `2R + 2`. If `2R + 2 \geq N`, then `R` has no right child.

To find the left sibling of node `R`, calculate `R - 1` if `R` is even and not 0. If `R` is odd or is 0, then it has no left sibling.

To find the right sibling of node `R`, calculate `R + 1` if `R` is odd, and `R + 1 < N`. If these conditions are not met, then `R` has no right sibling.

05.17 Heaps and Priority Queues

Due No Due Date **Points** 4 **Submitting** an external tool

05.17 Heaps and Priority Queues

5.17. Heaps and Priority Queues

5.17.1. Heaps and Priority Queues

There are many situations, both in real life and in computing applications, where we wish to choose the next “most important” from a collection of people, tasks, or objects. For example, doctors in a hospital emergency room often choose to see next the “most critical” patient rather than the one who arrived first. When scheduling programs for execution in a multitasking operating system, at any given moment there might be several programs (usually called **jobs**) ready to run. The next job selected is the one with the highest **priority**. Priority is indicated by a particular value associated with the job (and might change while the job remains in the wait list).

When a collection of objects is organized by importance or priority, we call this a **priority queue**. A normal queue data structure will not implement a priority queue efficiently because search for the element with highest priority will take $\Theta(n)$ time. A list, whether sorted or not, will also require $\Theta(n)$ time for either insertion or removal. A BST that organizes records by priority could be used, with the total of n inserts and n remove operations requiring $\Theta(n \log n)$ time in the average case. However, there is always the possibility that the BST will become unbalanced, leading to bad performance. Instead, we would like to find a data structure that is guaranteed to have good performance for this special application.

This section presents the **heap** [1] data structure. A heap is defined by two properties. First, it is a complete binary tree, so heaps are nearly always implemented using the **array representation for complete binary trees**. Second, the values stored in a heap are **partially ordered**. This means that there is a relationship between the value stored at any node and the values of its children. There are two variants of the heap, depending on the definition of this relationship.

1

Note that the term “heap” is also sometimes used to refer to **free store**.

A **max heap** has the property that every node stores a value that is *greater* than or equal to the value of either of its children. Because the root has a value greater than or equal to its children, which in turn have values greater than or equal to their children, the root stores the maximum of all values in the tree.

A **min heap** has the property that every node stores a value that is *less* than or equal to that of its children. Because the root has a value less than or equal to its children, which in turn have values less than or equal to their children, the root stores the minimum of all values in the tree.

Note that there is no necessary relationship between the value of a node and that of its sibling in either the min heap or the max heap. For example, it is possible that the values for all nodes in the left subtree of the root are greater than the values for every node of the right subtree. We can contrast BSTs and heaps by the strength of their ordering relationships. A BST defines a **total order** on its nodes in that, given the positions for any two nodes in the tree, the one to the "left" (equivalently, the one appearing earlier in an inorder traversal) has a smaller key value than the one to the "right". In contrast, a heap implements a partial order. Given their positions, we can determine the relative order for the key values of two nodes in the heap *only* if one is a descendant of the other.

Min heaps and max heaps both have their uses. For example, the Heapsort uses the max heap, while the Replacement Selection algorithm used for external sorting uses a min heap. The examples in the rest of this section will use a max heap.

Be sure not to confuse the logical representation of a heap with its physical implementation by means of the array-based complete binary tree. The two are not synonymous because the logical view of the heap is actually a tree structure, while the typical physical implementation uses an array.

Here is an implementation for max heaps. The class uses records that support the Comparable interface to provide flexibility.

Toggle Tree View

```
// Max-heap implementation
class MaxHeap {
  private Comparable[] Heap; // Pointer to the heap array
  private int size;          // Maximum size of the heap
  private int n;             // Number of things now in heap

  // Constructor supporting preloading of heap contents
  MaxHeap(Comparable[] h, int num, int max)
  { Heap = h;  n = num;  size = max;  buildheap(); }

  // Return current size of the heap
  int heapsize() { return n; }

  // Return true if pos a leaf position, false otherwise
  boolean isLeaf(int pos)
  { return (pos >= n/2) && (pos < n); }

  // Return position for left child of pos
  int leftchild(int pos) {
    if (pos >= n/2) { return -1; }
    return 2*pos + 1;
  }

  // Return position for right child of pos
  int rightchild(int pos) {
    if (pos >= (n-1)/2) { return -1; }
    return 2*pos + 2;
  }

  // Return position for parent
  int parent(int pos) {
    if (pos <= 0) { return -1; }
    return (pos-1)/2;
  }

  // Insert val into heap
  void insert(int key) {
    if (n >= size) {
      System.out.println("Heap is full");
      return;
    }
    int curr = n++;
    Heap[curr] = key;  // Start at end of heap
    // Now sift up until curr's parent's key > curr's key
    while ((curr != 0) && (Heap[curr].compareTo(Heap[parent(curr)]) > 0)) {
      Swap.swap(Heap, curr, parent(curr));
      curr = parent(curr);
    }
  }

  // Heapify contents of Heap
  void buildheap()
    { for (int i=n/2-1; i>=0; i--) { siftdown(i); } }

  // Put element in its correct place
  void siftdown(int pos) {
    if ((pos < 0) || (pos >= n)) { return; } // Illegal position
    while (!isLeaf(pos)) {
      int j = leftchild(pos);
      if ((j<(n-1)) && (Heap[j].compareTo(Heap[j+1]) < 0)) {
        j++; // j is now index of child with greater value
      }
      if (Heap[pos].compareTo(Heap[j]) >= 0) { return; }
      Swap.swap(Heap, pos, j);
      pos = j;  // Move down
    }
  }

  // Remove and return maximum value
  Comparable removemax() {
    if (n == 0) { return -1; }  // Removing from empty heap
    Swap.swap(Heap, 0, --n); // Swap maximum with last value
    siftdown(0);   // Put new heap root val in correct place
    return Heap[n];
  }

  // Remove and return element at specified position
  Comparable remove(int pos) {
    if ((pos < 0) || (pos >= n)) { return -1; } // Illegal heap position
    if (pos == (n-1)) { n--; } // Last element, no work to be done
    else {
      Swap.swap(Heap, pos, --n); // Swap with last value
      update(pos);
    }
    return Heap[n];
  }

  // Modify the value at the given position
  void modify(int pos, Comparable newVal) {
    if ((pos < 0) || (pos >= n)) { return; } // Illegal heap position
    Heap[pos] = newVal;
    update(pos);
  }

  // The value at pos has been changed, restore the heap property
  void update(int pos) {
    // If it is a big value, push it up
    while ((pos > 0) && (Heap[pos].compareTo(Heap[parent(pos)]) > 0)) {
      Swap.swap(Heap, pos, parent(pos));
      pos = parent(pos);
    }
    siftdown(pos); // If it is little, push down
  }
}
```

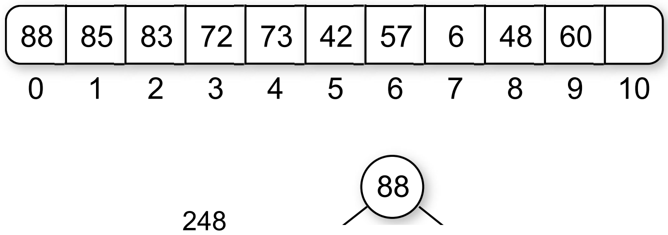
This class definition makes two concessions to the fact that an array-based implementation is used. First, heap nodes are indicated by their logical position within the heap rather than by a pointer to the node. In practice, the logical heap position corresponds to the identically numbered physical position in the array. Second, the constructor takes as input a pointer to the array to be used. This approach provides the greatest flexibility for using the heap because all data values can be loaded into the array directly by the client. The advantage of this comes during the heap construction phase, as explained below. The constructor also takes an integer parameter indicating the initial size of the heap (based on the number of elements initially loaded into the array) and a second integer parameter indicating the maximum size allowed for the heap (the size of the array).

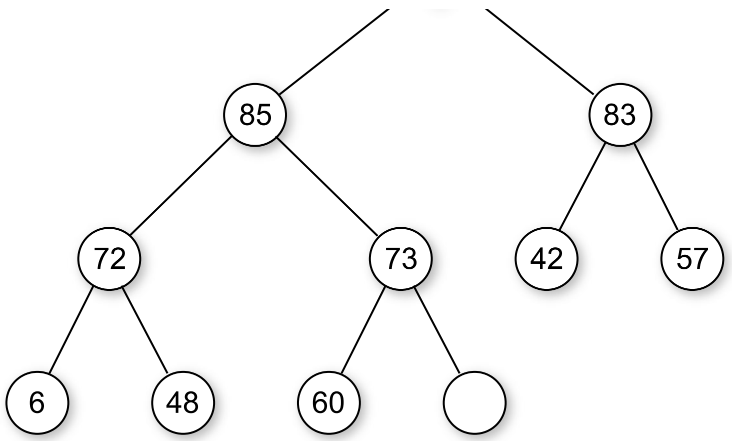
Method `heapsize` returns the current size of the heap. `H.isLeaf(pos)` returns TRUE if position `pos` is a leaf in heap `H`, and FALSE otherwise. Members `leftchild`, `rightchild`, and `parent` return the position (actually, the array index) for the left child, right child, and parent of the position passed, respectively.

One way to build a heap is to insert the elements one at a time. Method `insert` will insert a new element V into the heap.

1 / 6 << < > >>

Here is the process for inserting a new record into a heap.





You might expect the heap insertion process to be similar to the insert function for a BST, starting at the root and working down through the heap. However, this approach is not likely to work because the heap must maintain the shape of a complete binary tree. Equivalently, if the heap takes up the first n positions of its array prior to the call to `insert`, it must take up the first $n+1$ positions after. To accomplish this, `insert` first places V at position n of the array. Of course, V is unlikely to be in the correct position. To move V to the right place, it is compared to its parent's value. If the value of V is less than or equal to the value of its parent, then it is in the correct place and the insert routine is finished. If the value of V is greater than that of its parent, then the two elements swap positions. From here, the process of comparing V to its (current) parent continues until V reaches its correct position.

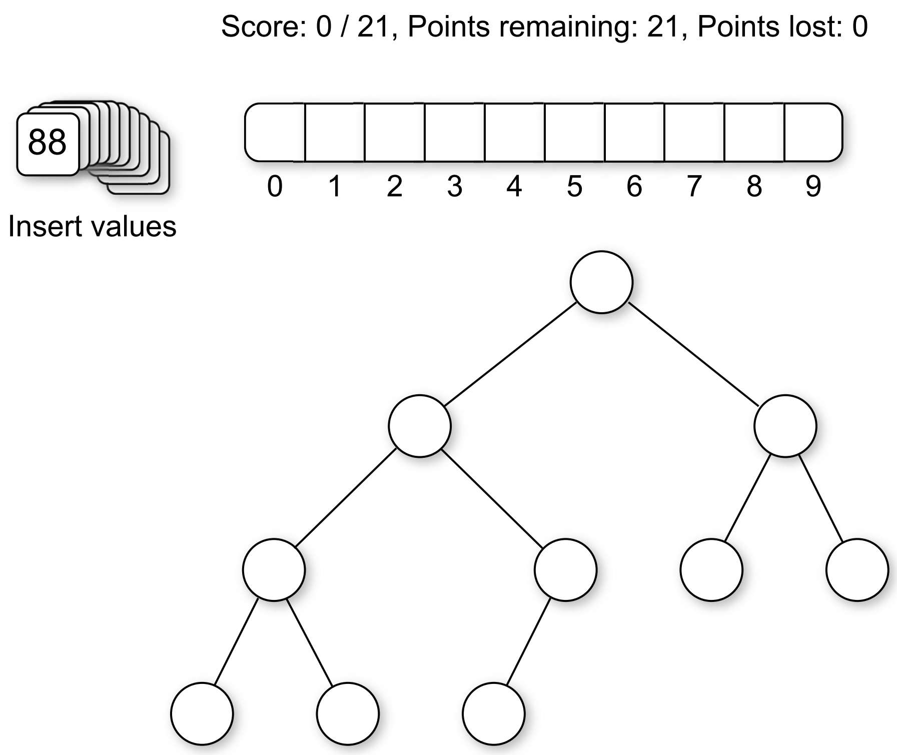
Since the heap is a complete binary tree, its height is guaranteed to be the minimum possible. In particular, a heap containing n nodes will have a height of $\Theta(\log n)$. Intuitively, we can see that this must be true because each level that we add will slightly more than double the number of nodes in the tree (the i th level has 2^i nodes, and the sum of the first i levels is $2^{i+1}-1$). Starting at 1, we can double only $\log n$ times to reach a value of n. To be precise, the height of a heap with n nodes is $\lceil \log n + 1 \rceil$.

Each call to `insert` takes $\Theta(\log n)$ time in the worst case, because the value being inserted can move at most the distance from the bottom of the tree to the top of the tree. Thus, to insert n values into the heap, if we insert them one at a time, will take $\Theta(n \log n)$ time in the worst case.

Reset | Model Answer

Instructions:

Insert the stream of keys shown in the upper-left corner into an originally empty **min** heap. The heap is displ
both its logical tree and physical array forms. Clicking on an element in one will also update the other. You c
a key by clicking an empty tree node or array cell. Once you insert a new key, be sure to update the key to r
min heap property as necessary. You can swap two records in the heap by first clicking one, then the other.



5.17.2. Building a Heap

If all n values are available at the beginning of the building process, we can build the heap faster than just inserting the values into the heap one by one. Consider this example, with two possible ways to heapify an initial set of values in an array.

1 / 3

<< < > >>

Two series of exchanges to build a max heap:

Figure 5.17.1: Two series of exchanges to build a max heap. (a) This heap is built by a series of nine exchanges in the order (4-2), (4-1), (2-1), (5-2), (5-4), (6-3), (6-5), (7-5), (7-6). (b) This heap is built by a series of four exchanges in the order (5-2), (7-3), (7-1), (6-1).

From this example, it is clear that the heap for any given set of numbers is not unique, and we see that some rearrangements of the input values require fewer exchanges than others to build the heap. So, how do we pick the best rearrangement?

One good algorithm stems from induction. Suppose that the left and right subtrees of the root are already heaps, and R is the name of the element at the root. This situation is illustrated by this figure:

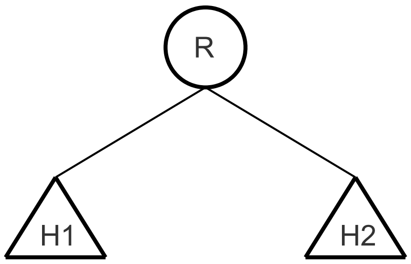


Figure 5.17.2: Final stage in the heap-building algorithm. Both subtrees of node R are heaps. All that remains is to push R down to its proper level in the heap.

In this case there are two possibilities.

1. R has a value greater than or equal to its two children. In this case, construction is complete.

2. R has a value less than one or both of its children.

R should be exchanged with the child that has greater value. The result will be a heap, except that R might still be less than one or both of its (new) children. In this case, we simply continue the process of “pushing down” R until it reaches a level where it is greater than its children, or is a leaf node. This process is implemented by the private method `siftdown`.

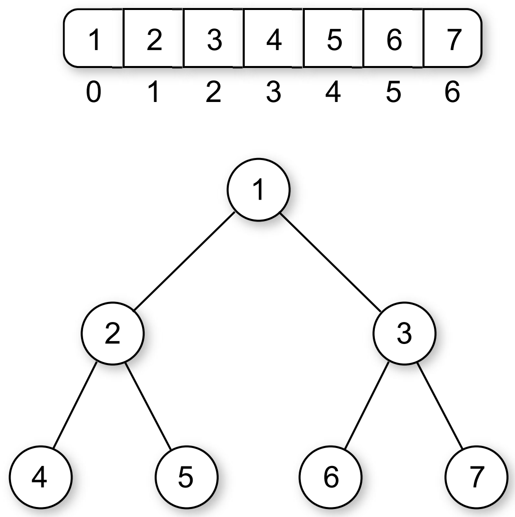
This approach assumes that the subtrees are already heaps, suggesting that a complete algorithm can be obtained by visiting the nodes in some order such that the children of a node are visited *before* the node itself. One simple way to do this is simply to work from the high index of the array to the low index. Actually, the build process need not visit the leaf nodes (they can never move down because they are already at the bottom), so the building algorithm can start in the middle of the array, with the first internal node.

Here is a visualization of the heap build process.

1 / 10

<< < > >>

Let's look at an efficient way to build the heap. We are going to make a max-heap from a set of input values.



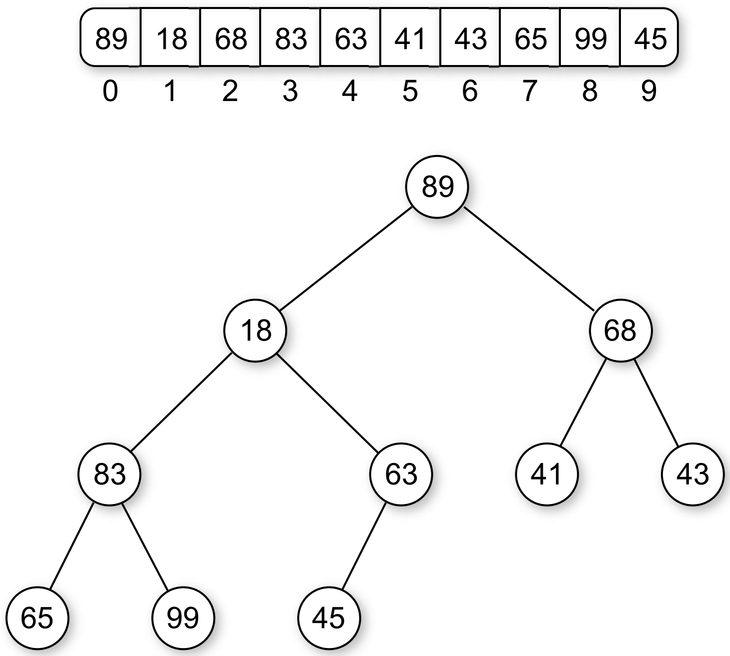
Method `buildHeap` implements the building algorithm.

Reset Model Answer

Instructions:

Your task is to reproduce the behavior of the buildheap algorithm to create a **min** heap from the data initially
array. The heap is displayed in both its logical tree and physical array forms. Clicking on an element in one
update the other. You can swap two records in the heap by first clicking one, then the other.

Score: 0 / 6, Points remaining: 6, Points lost: 0



What is the cost of `buildHeap`? Clearly it is the sum of the costs for the calls to `siftdown`. Each `siftdown` operation can cost at most the number of levels it takes for the node being sifted to reach the bottom of the tree. In any complete tree, approximately half of the nodes are leaves and so cannot be moved downward at all. One quarter of the nodes are one level above the leaves, and so their elements can move down at most one level. At each step up the tree we get half the number of nodes as were at the previous level, and an additional height of one. The maximum sum of total distances that elements can go is therefore

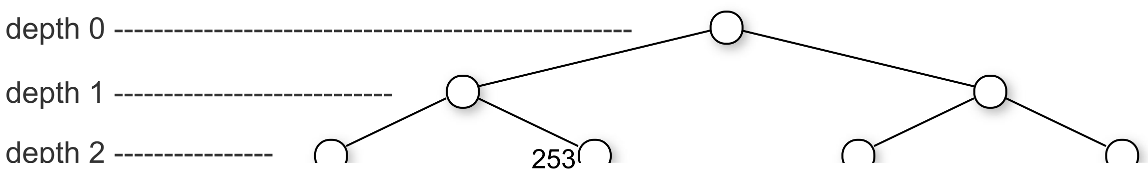
$$\sum_{i=1}^{\log n}(i-1)\frac{n}{2^i} = \frac{n}{2}\sum_{i=1}^{\log n}\frac{i-1}{2^{i-1}}.$$

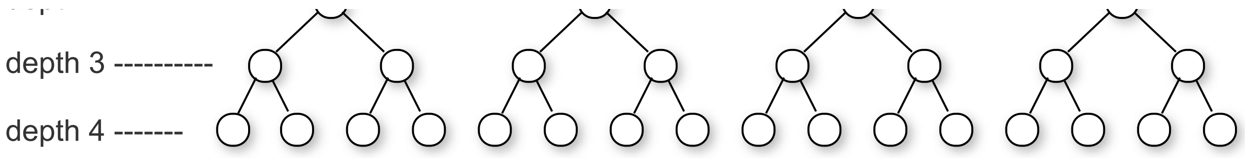
The summation on the right **is known** to have a closed-form solution of approximately 2, so this algorithm takes $\Theta(n)$ time in the worst case. This is far better than building the heap one element at a time, which would cost $\Theta(n \log n)$ in the worst case. It is also faster than the $\Theta(n \log n)$ average-case time and $\Theta(n^2)$ worst-case time required to build the BST.

1 / 13

<< < > >>

Let's look at a visualization to explain why the cost for buildheap is $\theta(n)$. We will use an example with
in the heap. This means that there are 16 leaf nodes and 15 internal nodes.

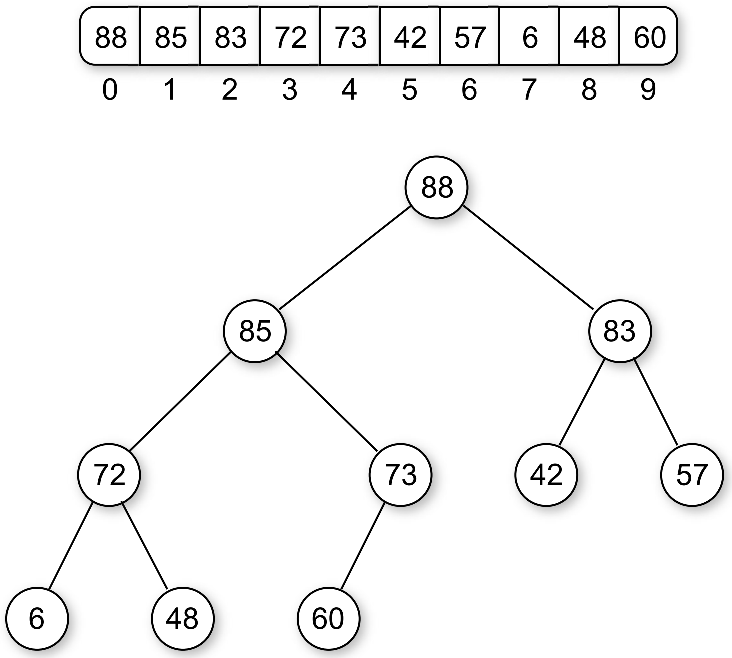




5.17.3. Removing from the heap or updating an object's priority

1 / 7

Here is the process for removing the maximum value from the max heap. We know that this value is at the position 0), but we also need to update the heap when we remove it.

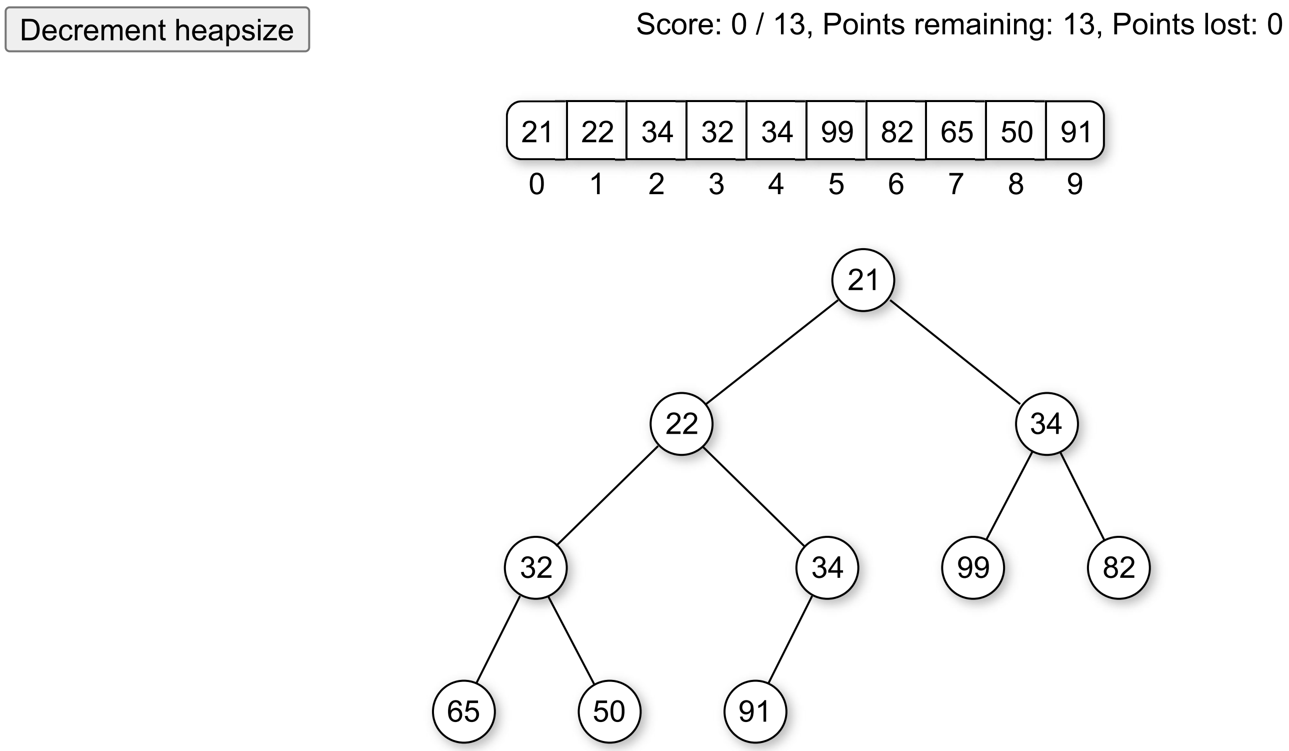


Because the heap is $\log n$ levels deep, the cost of deleting the maximum element is $\Theta(\log n)$ in the average and worst cases.

Instructions:

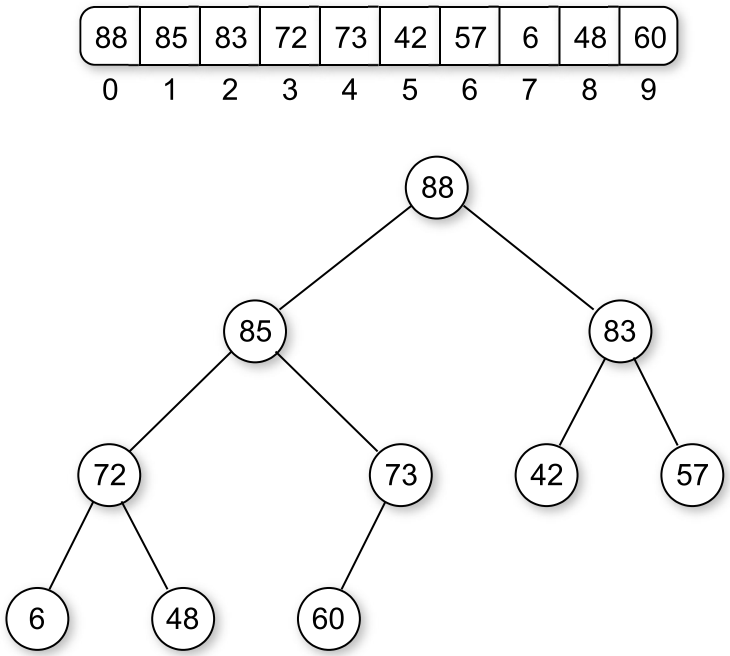
Perform DeleteMin three times. After each deletion, restore the **min** heap property. Click the "Decrement he

button to remove the last position from the heap. You can swap records in the heap by first clicking one key
another.



1 / 6

Perhaps we want to remove an arbitrary node from the heap. (Of course, the client that wants to do the dele
know the proper index for the thing to be deleted. Let's see how to remove the value at position 1 in the heap.



For some applications, objects might get their priority modified. One solution in this case is to remove the object and reinsert it. To do this, the application needs to know the position of the object in the heap. Another option is to change the priority value of the object, and then update its position in the heap. Note that a remove operation implicitly has to do this anyway, since when the last element in the heap is swapped with the one being removed, that value might be either too small or too big for its new position. So we use a utility method called `update` in both the `remove` and `modify` methods to handle this process.

5.17.4. Priority Queues

The heap is a natural implementation for the priority queue discussed at the beginning of this section. Jobs can be added to the heap (using their priority value as the ordering key) when needed. Method `removemax` can be called whenever a new job is to be executed.

Some applications of priority queues require the ability to change the priority of an object already stored in the queue. This might require that the object's position in the heap representation be updated. Unfortunately, a max heap is not efficient when searching for an arbitrary value; it is only good for finding the maximum value. However, if we already know the index for an object within the heap, it is a simple matter to update its priority (including changing its position to maintain the heap property) or remove it. The `remove` method takes as input the position of the node to be removed from the heap. A typical implementation for priority queues requiring updating of priorities will need to use an auxiliary data structure that supports efficient search for objects (such as a BST). Records in the auxiliary data structure will store the object's heap index, so that the object's priority can be updated. Priority queues can be helpful for solving graph problems such as **single-source shortest paths** and **minimal-cost spanning tree**.

05.18 Binary Tree Chapter Summary

Due No Due Date **Points** 1 **Submitting** an external tool

05.18 Binary Tree Chapter Summary

5.18. Binary Tree Chapter Summary

5.18.1. Summary Questions

Practicing Binary Tree Chapter Summary

Current score: 0 out of 5

The n nodes in a binary tree can be visited in:

○ $\Theta(1)$ time

○ $\Theta(\log n)$ time

○ $\Theta(n)$ time

○ $\Theta(n \log n)$ time

○ $\Theta(n^2)$ time

Answer

Check Answer

Need help?

I'd like a hint

Chapter 6: Sorting

6.1. Chapter Introduction: Sorting

We sort many things in our everyday lives: A handful of cards when playing Bridge; bills and other piles of paper; jars of spices; and so on. And we have many intuitive strategies that we can use to do the sorting, depending on how many objects we have to sort and how hard they are to move around. Sorting is also one of the most frequently performed computing tasks. We might sort the records in a database so that we can search the collection efficiently. We might sort customer records by zip code so that when we print an advertisement we can then mail them more cheaply. We might use sorting to help an algorithm to solve some other problem. For example, **Kruskal's algorithm** to find a **minimal-cost spanning tree** must sort the edges of a graph by their lengths before it can process them.

Because sorting is so important, naturally it has been studied intensively and many algorithms have been devised. Some of these algorithms are straightforward adaptations of schemes we use in everyday life. For example, a natural way to sort your cards in a bridge hand is to go from left to right, and place each card in turn in its correct position relative to the other cards that you have already sorted. This is the idea behind **Insertion Sort**. Other sorting algorithms are totally alien to how humans do things, having been invented to sort thousands or even millions of records stored on the computer. For example, no normal person would use **Quicksort** to order a pile of bills by date, even though Quicksort is the standard sorting algorithm of choice for most software libraries. After years of study, there are still unsolved problems related to sorting. New algorithms are still being developed and refined for special-purpose applications.

Along with introducing this central problem in computer science, studying sorting algorithms helps us to understand issues in algorithm design and analysis. For example, the sorting algorithms in this chapter show multiple approaches to using **divide and conquer**. In particular, there are multiple ways to do the dividing. **Mergesort** divides a list in half. **Quicksort** divides a list into big values and small values. **Radix Sort** divides the problem by working on one digit of the key at a time. Sorting algorithms can also illustrate a wide variety of algorithm analysis techniques. Quicksort illustrates that it is possible for an algorithm to have an **average case** whose growth rate is significantly smaller than its **worst case**. It is possible to speed up one sorting algorithm (such as **Shellsort** or Quicksort) by taking advantage of the **best case** behavior of another algorithm (Insertion Sort). Special case behavior by some sorting algorithms makes them a good solution for special niche applications (**Heapsort**). Sorting provides an example of an important technique for analyzing the lower bound for a problem. **External Sorting** refers to the process of sorting large files stored on disk.

This chapter covers several standard algorithms appropriate for sorting a collection of records that fit into the computer's main memory. It begins with a discussion of three simple, but relatively slow, algorithms that require $\Theta(n^2)$ time in the average and worst cases to sort n records. Several algorithms with considerably better

06.02 Sorting Terminology and Notation

Due No Due Date **Points** 1 **Submitting** an external tool

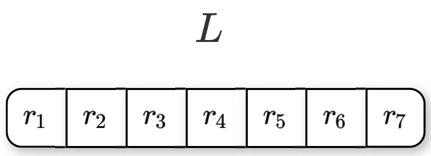
06.02 Sorting Terminology and Notation

6.2. Sorting Terminology and Notation

6.2.1. Sorting Terminology and Notation

1 / 4 << < > >>

Consider a list L containing seven records, named r_1 through r_7.



As defined, the **Sorting Problem** allows input with two or more records that have the same key value. Certain applications require that input not contain duplicate key values. Typically, sorting algorithms can handle duplicate key values unless noted otherwise. When duplicate key values are allowed, there might be an implicit ordering to the duplicates, typically based on their order of occurrence within the input. It might be desirable to maintain this initial ordering among duplicates. A sorting algorithm is said to be **stable** if it does not change the relative ordering of records with identical key values. Many, but not all, of the sorting algorithms presented in this chapter are stable, or can be made stable with minor changes.

When comparing two sorting algorithms, the simplest approach would be to program both and measure their running times. This is an example of **empirical comparison**. However, doing fair empirical comparisons can be tricky because the running time for many sorting algorithms depends on specifics of the input values. The number of records, the size of the keys and the records, the allowable range of the key values, and the amount by which the input records are "out of order" can all greatly affect the relative running times for sorting algorithms.

When analyzing sorting algorithms, it is traditional to measure the cost by counting the number of comparisons made between keys. This measure is usually closely related to the actual running time for the algorithm and has the advantage of being machine and data-type independent. However, in some cases records might be so large that their physical movement might take a significant fraction of the total running time. If so, it might be appropriate to measure the cost by counting the number of swap operations performed by the algorithm. In most applications we can assume that all records and keys are of fixed length, and that a single comparison or a single swap operation requires a constant amount of time regardless of which keys are involved. However, some special situations "change the rules" for comparing sorting algorithms. For example, an application with records or keys having widely varying length (such as sorting a sequence of variable length strings) cannot expect all comparisons to cost roughly the same. Not only do such situations require special measures for analysis, they also will usually benefit from a special-purpose sorting technique.

Other applications require that a small number of records be sorted, but that the sort be performed frequently. An example would be an application that repeatedly sorts groups of five numbers. In such cases, the constants in the runtime equations that usually get ignored in asymptotic analysis now become crucial. Note that recursive sorting algorithms end up sorting lots of small lists as well.

Finally, some situations require that a sorting algorithm use as little memory as possible. We will call attention to sorting algorithms that require significant extra memory beyond the input array.

Practicing Sorting Introduction: Summary Questions

Current score: 0 out of 5

Sometimes, the constant factors in an algorithm's runtime equation are more important thant its growth rate. When the problem is sorting, this can happen in which situation?

○ When the CPU is really fast

○ When we are sorting lots of small groups of records.

○ When the records are nearly sorted

○ When there are lots of records

○ When the amount of available space is small

○ When the records are nearly reverse sorted

Answer

Check Answer

Need help?

I'd like a hint

06.03 Insertion Sort

Due No Due Date **Points** 2 **Submitting** an external tool

06.03 Insertion Sort

6.3. Insertion Sort

6.3.1. Insertion Sort

What would you do if you have a stack of phone bills from the past two years and you want to order by date? A fairly natural way to handle this is to look at the first two bills and put them in order. Then take the third bill and put it into the right position with respect to the first two, and so on. As you take each bill, you would add it to the sorted pile that you have already made. This simple approach is the inspiration for our first sorting algorithm, called **Insertion Sort**.

Insertion Sort iterates through a list of records. For each iteration, the current record is inserted in turn at the correct position within a sorted list composed of those records already processed. Here is an implementation. The input is an array named `A` that stores n records.

Java | **Java (Generic)** | Toggle Tree Vie

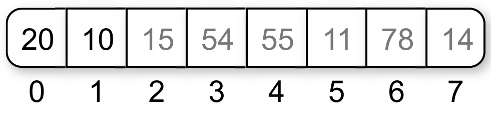
```
static <T extends Comparable<T>> void inssort(T[] A) {
  for (int i=1; i<A.length; i++) // Insert i'th record
    for (int j=i; (j>0) && (A[j].compareTo(A[j-1]) < 0); j--)
      Swap.swap(A, j, j-1);
}
```

(Note that to make the explanation for these sorting algorithms as simple as possible, our visualizations will show the array as though it stored simple integers rather than more complex records. But you should realize that in practice, there is rarely any point to sorting an array of simple integers. Nearly always we want to sort more complex records that each have a **key** value. In such cases we must **have a way** to associate a key value with a record. The sorting algorithms will simply assume that the records are **comparable**.)

Here we see the first few iterations of Insertion Sort.

1 / 11 ≪ < > ≫

Insertion Sort starts with the record in position 1.



This continues on with each record in turn. Call the current record x. Insertion Sort will move it to the left so long as its value is less than that of the record immediately preceding it. As soon as a key value less than or equal to x is encountered, `inssort` is done with that record because all records to its left in the array must have smaller keys.

Insertion Sort Visualization

Run Reset List size: 8

Your values: Type some array values, or click 'run' to use random values

Khan.randRange(6, 10) Khan.randRange(2, arrSize-1) inssortPRO.initJSAV(arrSize, sortPos)

Your task in this exercise is to show the behavior for one iteration of the outer for loop of Insertion Sort. In the array displayed below, the record at position *sortPos* is highlighted. Insertion Sort has already processed the values to the left of position *sortPos*, so those elements are sorted.

Perform the next part of Insertion Sort to move the highlighted record to its proper place in the array. To swap two elements, click on the first and then click on the second.

Reset

[inssortPRO.userInput]
if (!inssortPRO.checkAnswer(arrSize) && !guess[0]) { return ""; // User did not click, and correct answer is not // initial array state } else { return inssortPRO.checkAnswer(arrSize); }

Insert the highlighted element into its proper position in the array using Insertion Sort.

Nothing to the right of position *sortPos* should be changed.

All elements from position 0 to position *sortPos* should be in ascending order.

6.3.2. Insertion Sort Analysis

1 / 24 << < > >>

We first examine the worst case cost.

1 / 5

<< < > >>

Now we will consider the best case cost.

1 / 8

<< < > >>

Finally, consider the average case cost.

While the best case is significantly faster than the average and worst cases, the average and worst cases are usually more reliable indicators of the “typical” running time. However, there are situations where we can expect the input to be in sorted or nearly sorted order. One example is when an already sorted list is slightly disordered by a small number of additions to the list; restoring sorted order using Insertion Sort might be a good idea if we know that the disordering is slight. And even when the input is not perfectly sorted, Insertion Sort’s cost goes up in proportion to the number of inversions. So a “nearly sorted” list will always be cheap to sort with Insertion Sort. Examples of algorithms that take advantage of Insertion Sort’s near-best-case running time are **Shellsort** and **Quicksort**.

Counting comparisons or swaps yields similar results. Each time through the inner `for` loop yields both a comparison and a swap, except the last (i.e., the comparison that fails the inner `for` loop’s test), which has no swap. Thus, the number of swaps for the entire sort operation is $n-1$ less than the number of comparisons. This is 0 in the best case, and $\Theta(n^2)$ in the average and worst cases.

Later we will see algorithms whose growth rate is much better than $\Theta(n^2)$. Thus for larger arrays, Insertion Sort will not be so good a performer as other algorithms. So Insertion Sort is not the best sorting algorithm to use in most situations. But there are special situations where it is ideal. We already know that Insertion Sort works great when the input is sorted or nearly so. Another good time to use Insertion Sort is when the array is very small, since Insertion Sort is so simple. The algorithms that have better asymptotic growth rates tend to be more complicated, which leads to larger constant factors in their running time. That means they typically need fewer comparisons for larger arrays, but they cost more per comparison. This observation might not seem that helpful, since even an algorithm with high cost per comparison will be fast on small input sizes. But there are times when we might need to do many, many sorts on very small arrays. You should spend some time right now trying to think of a situation where you will need to sort many small arrays. Actually, it happens a lot.

See **Computational Fairy Tales: Why Tailors Use Insertion Sort** for a discussion on how the relative costs of search and insert can affect what is the best sort algorithm to use.

06.04 Selection Sort

Due No Due Date **Points** 2 **Submitting** an external tool

06.04 Selection Sort

6.4. Selection Sort

6.4.1. Selection Sort

Consider again the problem of sorting a pile of phone bills for the past year. Another intuitive approach might be to look through the pile until you find the bill for January, and pull that out. Then look through the remaining pile until you find the bill for February, and add that behind January. Proceed through the ever-shrinking pile of bills to select the next one in order until you are done. This is the inspiration for our last $\Theta(n^2)$ sort, called **Selection Sort**. The i'th pass of Selection Sort "selects" the i'th largest key in the array, placing that record at the end of the array. In other words, Selection Sort first finds the largest key in an unsorted list, then the next largest, and so on. Its unique feature is that there are few record swaps. To find the next-largest key value requires searching through the entire unsorted portion of the array, but only one swap is required to put the record into place. Thus, the total number of swaps required will be $n - 1$ (we get the last record in place "for free").

Here is an implementation for Selection Sort.

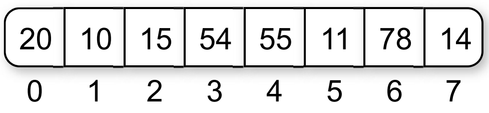
Java | Java (Generic) | Toggle Tree Vie

```
static <T extends Comparable<T>> void selsort(T[] A) {
  for (int i=0; i<A.length-1; i++) {       // Select i'th biggest record
    int bigindex = 0;                      // Current biggest index
    for (int j=1; j<A.length-i; j++)       // Find the max value
      if (A[j].compareTo(A[bigindex]) > 0) // Found something bigger
        bigindex = j;                      // Remember bigger index
    Swap.swap(A, bigindex, A.length-i-1);  // Put it into place
  }
}
```

Consider the example of the following array.

1 / 15 << < > >>

Moving from left to right, find the element with the greatest value.

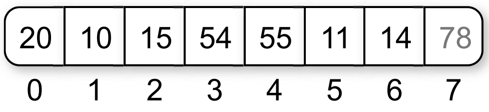


Now we continue with the second pass. However, since the largest record is already at the right end, we will not need to look at it again.

1 / 13

<< < > >>

Second pass: moving from left to right, find the element with the second greatest value.



Selection Sort continues in this way until the entire array is sorted.

The following visualization puts it all together.

Selection Sort Visualization

Run Reset List size: 8

Your values: Type some array values, or click 'run' to use random values

Now try for yourself to see if you understand how Selection Sort works.

Khan.randRange(7, 9) Khan.randRange(4, arrSize-1) selsortPRO.initJSAV(arrSize, sortPos)

In the array of size *arrSize* displayed below, the element at position *sortPos* is highlighted. The array represents an intermediate state in Selection Sort, with all elements to the right of the highlighted element holding the biggest values in the array.

Perform the next iteration of Selection Sort, to put the proper array element into the highlighted position. To swap two elements, click on the first and then click on the second.

Reset
[selsortPRO.userInput]
if (!selsortPRO.checkAnswer(arrSize) && !guess[0]) { return ""; // User did not click, and correct answer is not // initial array state } else { return selsortPRO.checkAnswer(arrSize); }

Determine the record that should appear at index *sortPos* using Selection Sort.

Selection sort will place the largest record in the range [0 to *sortPos*] at position *sortPos*.

6.4.2. Selection Sort Analysis

Any algorithm can be written in slightly different ways. For example, we could have written Selection Sort to find the smallest record, the next smallest, and so on. We wrote this version of Selection Sort to mimic the behavior of our Bubble Sort implementation as closely as possible. This shows that Selection Sort is essentially a Bubble Sort except that rather than repeatedly swapping adjacent values to get the next-largest record into place, we instead remember the position of the record to be selected and do one swap at the end.

This visualization analyzes the number of comparisons and swaps required by Selection Sort.

1 / 40 << < > >>

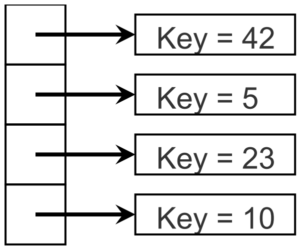
What is the cost for Selection Sort?

There is another approach to keeping the cost of swapping records low, and it can be used by any sorting algorithm even when the records are large. This is to have each element of the array store a pointer to a record rather than store the record itself. In this implementation, a swap operation need only exchange the pointer values. The large records do not need to move. This technique is illustrated by the following visualization. Additional space is needed to store the pointers, but the return is a faster swap operation.

1 / 4

<< < > >>

Here we see an array with references to four records.



Here are some review questions to check how well you understand Selection Sort.

06.05 The Cost of Exchange Sorting

Due No Due Date **Points** 2 **Submitting** an external tool

06.05 The Cost of Exchange Sorting

6.5. The Cost of Exchange Sorting

6.5.1. The Cost of Exchange Sorting

Here is a summary for the cost of Insertion Sort, Bubble Sort, and Selection Sort in terms of their required number of comparisons and swaps in the best, average, and worst cases. The running time for each of these sorts is $\Theta(n^2)$ in the average and worst cases.

	Insertion	**Bubble**	**Selection**
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

The remaining sorting algorithms presented in this tutorial are significantly better than these three under typical conditions. But before continuing on, it is instructive to investigate what makes these three sorts so slow. The crucial bottleneck is that only *adjacent* records are compared. Thus, comparisons and moves (for Insertion and Bubble Sort) are by single steps. Swapping adjacent records is called an **exchange**. Thus, these sorts are sometimes referred to as an **exchange sort**. The cost of any exchange sort can be at best the total number of steps that the records in the array must move to reach their "correct" location. Recall that this is at least the number of inversions for the record, where an inversion occurs when a record with key value greater than the current record's key value appears before it.

Khan.randRange(4, 6) findInversionsPRO.initJSAV(A) findInversionsPRO.genAnswer()
How many inversions are in the following array?

CorrectAnswer

To count the number of inversions look at each value and count the number of times that a bigger value is to its left.

The total number of inversions is *CorrectAnswer*

6.5.2. Analysis

1 / 16

<< < > >>

What is the average number of inversions?

06.06 Mergesort Concepts

Due No Due Date **Points** 2 **Submitting** an external tool

06.06 Mergesort Concepts

6.6. Mergesort Concepts

6.6.1. Mergesort Concepts

A natural approach to problem solving is divide and conquer. To use divide and conquer when sorting, we might consider breaking the list to be sorted into pieces, process the pieces, and then put them back together somehow. A simple way to do this would be to split the list in half, sort the halves, and then merge the sorted halves together. This is the idea behind **Mergesort**.

Mergesort is one of the simplest sorting algorithms conceptually, and has good performance both in the asymptotic sense and in empirical running time. Unfortunately, even though it is based on a simple concept, it is relatively difficult to implement in practice. Here is a pseudocode sketch of Mergesort:

```
List mergesort(List inlist) {
  if (inlist.length() <= 1) return inlist;;
  List L1 = half of the items from inlist;
  List L2 = other half of the items from inlist;
  return merge(mergesort(L1), mergesort(L2));
}
```

Here is a visualization that illustrates how Mergesort works.

Mergesort Visualization

Run Reset List size: 8

Your values: Type some array values, or click 'run' to use random values

The hardest step to understand about Mergesort is the merge function. The merge function starts by examining the first record of each sublist and picks the smaller value as the smallest record overall. This smaller value is removed from its sublist and placed into the output list. Merging continues in this way, comparing the front records of the sublists and continually appending the smaller to the output list until no more input records remain.

Here is pseudocode for merge on lists:

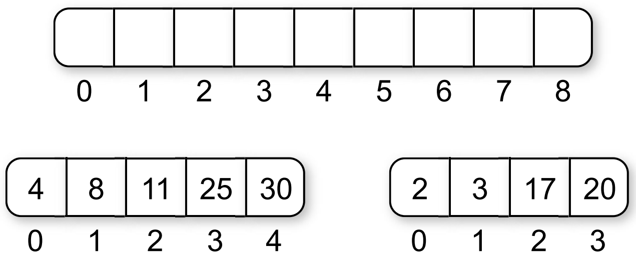
```
List merge(List L1, List L2) {
  List answer = new List();
  while (L1 != NULL || L2 != NULL) {
    if (L1 == NULL) { // Done L1
      answer.append(L2);
      L2 = NULL;
    }
    else if (L2 == NULL) { // Done L2
      answer.append(L1);
      L1 = NULL;
    }
    else if (L1.value() <= L2.value()) {
      answer.append(L1.value());
      L1 = L1.next();
    }
    else {
      answer.append(L2.value());
      L2 = L2.next();
    }
  }
  return answer;
}
```

Here is a visualization for the merge operation.

1 / 21

<< < > >>

We will merge two sorted lists into one.



Here is a mergesort warmup exercise to practice merging.

Khan.randRange(6, 10) mergesortMergePRO.initJSAV(arr_size)

Merge the two subarrays below into the larger array. To move a value in a subarray, click on it (it will then be highlighted in yellow), and then click on the proper position in the array above.

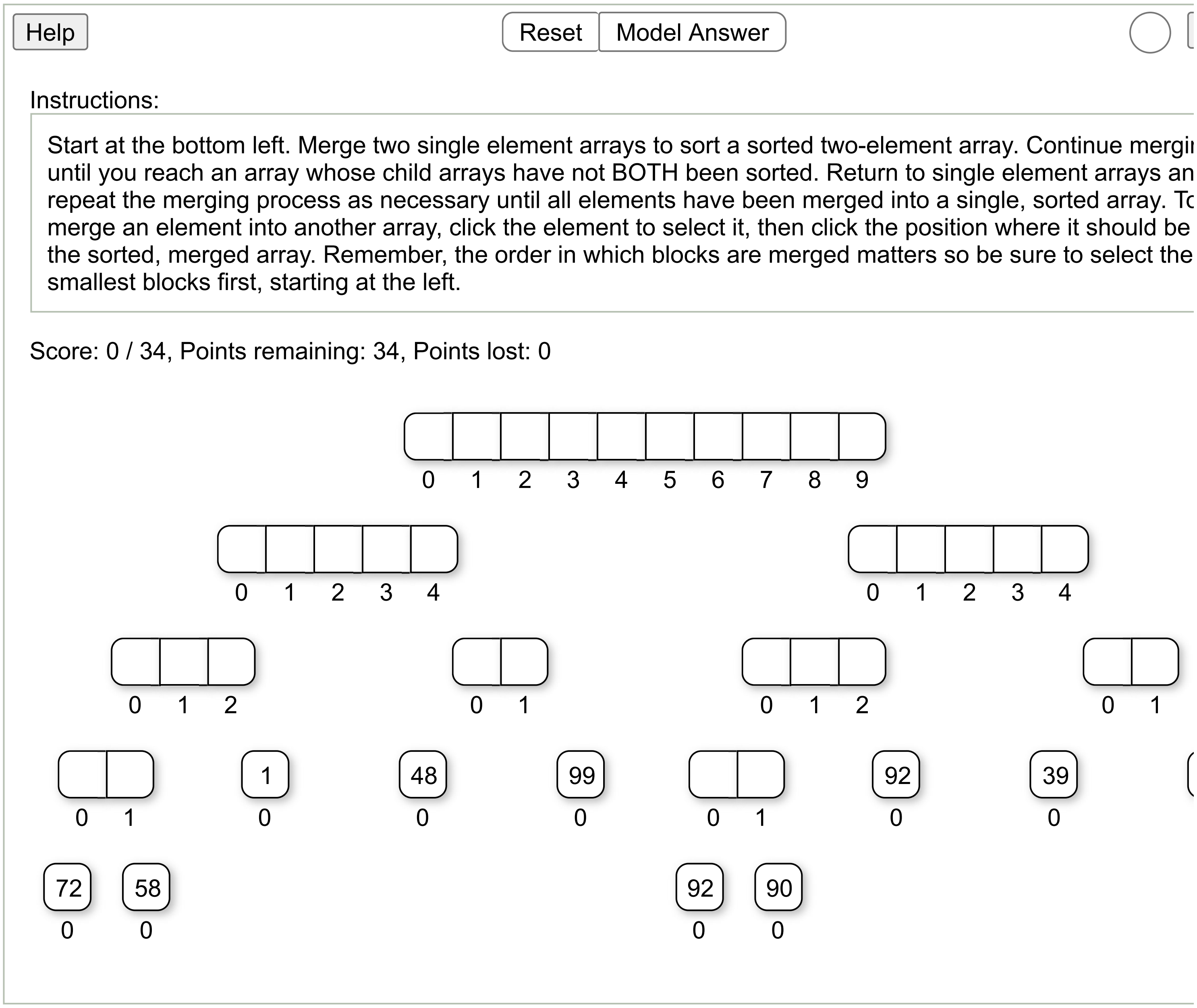
Reset

[mergesortMergePRO.userInput]
if (!mergesortMergePRO.checkAnswer(arr_size) && !guess[0]) { return ""; // User did not click, and correct answer is not // initial array state } else { return mergesortMergePRO.checkAnswer(arr_size); }

Merging the sorted subarrays results in a sorted final array.

6.6.2. Mergesort Practice Exercise

Now here is a full proficiency exercise to put it all together.



Instructions:

Start at the bottom left. Merge two single element arrays to sort a sorted two-element array. Continue mergi
until you reach an array whose child arrays have not BOTH been sorted. Return to single element arrays an
repeat the merging process as necessary until all elements have been merged into a single, sorted array. Tc
merge an element into another array, click the element to select it, then click the position where it should be
the sorted, merged array. Remember, the order in which blocks are merged matters so be sure to select the
smallest blocks first, starting at the left.

Score: 0 / 34, Points remaining: 34, Points lost: 0

This visualization provides a running time analysis for Merge Sort.

1 / 33

<< < > >>

The analysis of merge sort is straightforward. Consider the following array of 8 elements.

06.07 Implementing Mergesort

Due No Due Date **Points** 1 **Submitting** an external tool

06.07 Implementing Mergesort

6.7. Implementing Mergesort

6.7.1. Implementing Mergesort

Implementing Mergesort presents a number of technical difficulties. The first decision is how to represent the lists. Mergesort lends itself well to sorting a singly linked list because merging does not require random access to the list elements. Thus, Mergesort is the method of choice when the input is in the form of a linked list. Implementing `merge` for linked lists is straightforward, because we need only remove items from the front of the input lists and append items to the output list. Breaking the input list into two equal halves presents some difficulty. Ideally we would just break the lists into front and back halves. However, even if we know the length of the list in advance, it would still be necessary to traverse halfway down the linked list to reach the beginning of the second half. A simpler method, which does not rely on knowing the length of the list in advance, assigns elements of the input list alternating between the two sublists. The first element is assigned to the first sublist, the second element to the second sublist, the third to first sublist, the fourth to the second sublist, and so on. This requires one complete pass through the input list to build the sublists.

When the input to Mergesort is an array, splitting input into two subarrays is easy if we know the array bounds. Merging is also easy if we merge the subarrays into a second array. Note that this approach requires twice the amount of space as any of the sorting methods presented so far, which is a serious disadvantage for Mergesort. It is possible to merge the subarrays without using a second array, but this is extremely difficult to do efficiently and is not really practical. Merging the two subarrays into a second array, while simple to implement, presents another difficulty. The merge process ends with the sorted list in the auxiliary array. Consider how the recursive nature of Mergesort breaks the original array into subarrays. Mergesort is recursively called until subarrays of size 1 have been created, requiring $\log n$ levels of recursion. These subarrays are merged into subarrays of size 2, which are in turn merged into subarrays of size 4, and so on. We need to avoid having each merge operation require a new array. With some difficulty, an algorithm can be devised that alternates between two arrays. A much simpler approach is to copy the sorted sublists to the auxiliary array first, and then merge them back to the original array.

Here is a complete implementation for mergesort following this approach. The input records are in array `A`. Array `temp` is used as a place to temporarily copy records during the merge process. Parameters `left` and `right` define the left and right indices, respectively, for the subarray being sorted. The initial call to `mergesort` would be `mergesort(array, temparray, 0, n-1)`.

Toggle Tree View

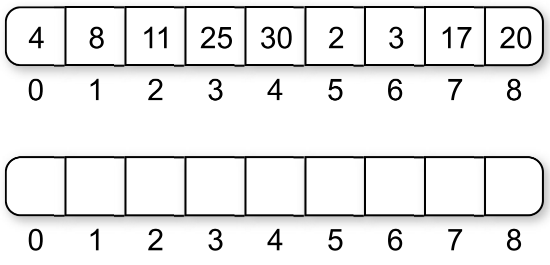
```
static void mergesort(Comparable[] A, Comparable[] temp, int left, int right) {
  if (left == right) { return; }       // List has one record
  int mid = (left+right)/2;           // Select midpoint
  mergesort(A, temp, left, mid);      // Mergesort first half
  mergesort(A, temp, mid+1, right);   // Mergesort second half
  for (int i=left; i<=right; i++) {    // Copy subarray to temp
    temp[i] = A[i];
  }
  // Do the merge operation back to A
  int i1 = left;
  int i2 = mid + 1;
  for (int curr = left; curr <= right; curr++) {
    if (i1 == mid+1) {                 // Left sublist exhausted
      A[curr] = temp[i2++];
    }
    else if (i2 > right) {             // Right sublist exhausted
      A[curr] = temp[i1++];
    }
    else if (temp[i1].compareTo(temp[i2]) <= 0) {  // Get smaller value
      A[curr] = temp[i1++];
    }
    else{
      A[curr] = temp[i2++];
    }
  }
}
```

Here is a visualization for the merge step.

1 / 21

<< < > >>

Initially, we have the two sorted sublists in array A, and an empty temp array.



An optimized Mergesort implementation is shown below. It reverses the order of the second subarray during the initial copy. Now the current positions of the two subarrays work inwards from the ends, allowing the end of each subarray to act as a sentinel for the other. Unlike the previous implementation, no test is needed to check for when one of the two subarrays becomes empty. This version also has a second optimization: It uses Insertion Sort to sort small subarrays whenever the size of the array is smaller than a value defined by THRESHOLD.

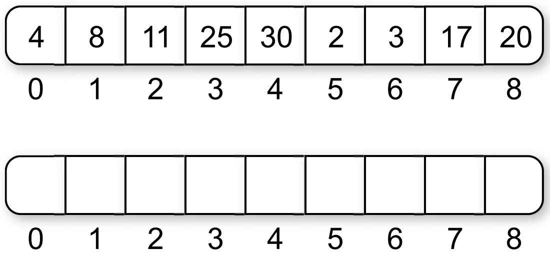
Toggle Tree View

```
static void mergesortOpt(Comparable[] A, Comparable[] temp, int left, int right) {
  int i, j, k, mid = (left+right)/2;  // Select the midpoint
  if (left == right) { return; }          // List has one record
  if ((mid-left) >= THRESHOLD) { mergesortOpt(A, temp, left, mid); }
  else { inssort(A, left, mid); }
  if ((right-mid) > THRESHOLD) { mergesortOpt(A, temp, mid+1, right); }
  else { inssort(A, mid+1, right); }
  // Do the merge operation.  First, copy 2 halves to temp.
  for (i=left; i<=mid; i++) { temp[i] = A[i]; }
  for (j=right; j>mid; j--) { temp[i++] = A[j]; }
  // Merge sublists back to array
  for (i=left,j=right,k=left; k<=right; k++) {
    if (temp[i].compareTo(temp[j]) <= 0) { A[k] = temp[i++]; }
    else {
      A[k] = temp[j--];
    }
  }
}
```

Here is a visualization for the optimized merge step.

1 / 22

Initially, we have the two sorted sublists in array A, and an empty temp array.



06.08 Heapsort

Due No Due Date **Points** 3 **Submitting** an external tool

06.08 Heapsort

6.8. Heapsort

6.8.1. Heapsort

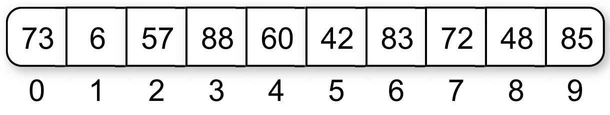
Our discussion of Quicksort began by considering the practicality of using a BST for sorting. The BST requires more space than the other sorting methods and will be slower than Quicksort or Mergesort due to the relative expense of inserting values into the tree. There is also the possibility that the BST might be unbalanced, leading to a $\Theta(n^2)$ worst-case running time. Subtree balance in the BST is closely related to Quicksort's partition step. Quicksort's pivot serves roughly the same purpose as the BST root value in that the left partition (subtree) stores values less than the pivot (root) value, while the right partition (subtree) stores values greater than or equal to the pivot (root).

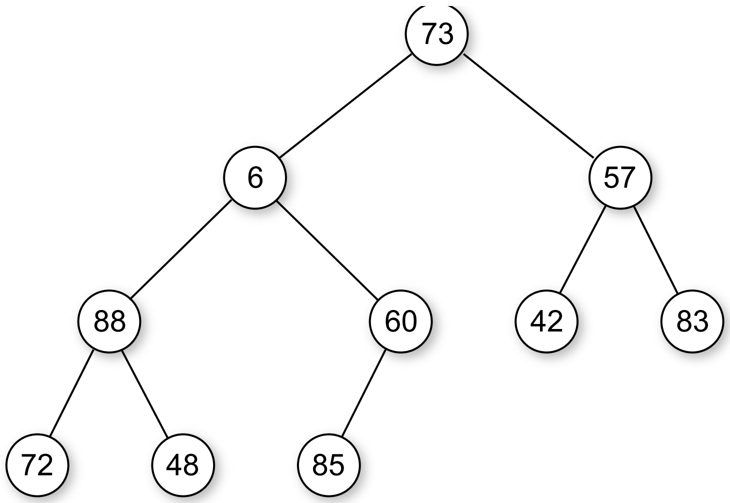
A good sorting algorithm can be devised based on a tree structure more suited to the purpose. In particular, we would like the tree to be balanced, space efficient, and fast. The algorithm should take advantage of the fact that sorting is a special-purpose application in that all of the values to be stored are available at the start. This means that we do not necessarily need to insert one value at a time into the tree structure.

Heapsort is based on the **heap** data structure. Heapsort has all of the advantages just listed. The complete binary tree is balanced, its array representation is space efficient, and we can load all values into the tree at once, taking advantage of the efficient `buildheap` function. The asymptotic performance of Heapsort when all of the records have unique key values is $\Theta(n \log n)$ in the best, average, and worst cases. It is not as fast as Quicksort in the average case (by a constant factor), but Heapsort has special properties that will make it particularly useful for **external sorting** algorithms, used when sorting data sets too large to fit in main memory.

1 / 51 << < > >>

Initially, we start with our unsorted array.





A complete implementation is as follows.

Toggle Tree View

```
static void heapsort(Comparable[] A) {
  // The heap constructor invokes the buildheap method
  MaxHeap H = new MaxHeap(A, A.length, A.length);
  for (int i=0; i<A.length; i++) {  // Now sort
    H.removemax(); // Removemax places max at end of heap
  }
}
```

Here is a warmup practice exercise for Heapsort.

Khan.randRange(6, 11) heapsortStepPRO.initJSAV(arrSize)

Perform one iteration of heap sort. Swap the last key with the largest key and restore the heap in the array/tree shown below. Use the "Decrement" button to reduce the size of the heap.

Reset | Decrement Heap Size

[heapsortStepPRO.userInput]
if (!guess[0]) { return ""; // User did not click, and correct answer is not // initial array state } else { return heapsortStepPRO.checkAnswer(arrSize); }

Select the maximum element and move it to the correct location. Then restore the heap order.

Don't forget to decrement the heap size.

6.8.2. Heapsort Proficiency Practice

Now test yourself to see how well you understand Heapsort. Can you reproduce its behavior?

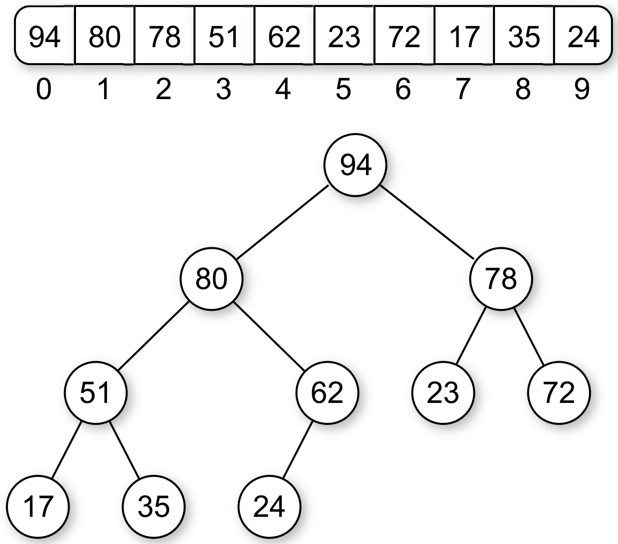
Help

Reset Model Answer

Instructions:

Reproduce the behavior of heapsort for the **maximum** heap below. You can swap keys by clicking the first c
and then the second one in either of the representations (array or binary tree). Begin by swapping the last k
with the largest key, and reducing the size of the heap by one (by clicking the "Decrement heap size" button
After that, restore the heap property again.

Decrement heap size   Score: 0 / 31, Points remaining: 31, Points lost: 0



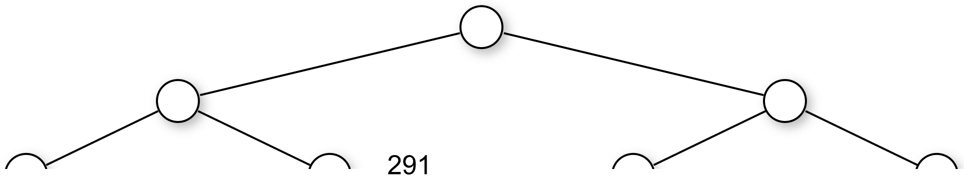
6.8.3. Heapsort Analysis

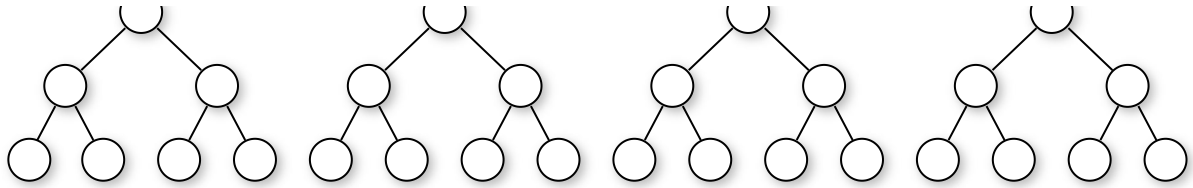
This visualization presents the running time analysis of Heap Sort

1 / 12

<< < > >>

The first step in heapsort is to heapify the array. This will cost $\theta(n)$ running time for an array of size n. Consider the following structure of a Max Heap





While typically slower than Quicksort by a constant factor (because unloading the heap using `removemax` is somewhat slower than Quicksort's series of partitions), Heapsort has one special advantage over the other sorts studied so far. Building the heap is relatively cheap, requiring $\Theta(n)$ time. Removing the maximum-valued record from the heap requires $\Theta(\log n)$ time in the worst case. Thus, if we wish to find the k records with the largest key values in an array, we can do so in time $\Theta(n + k \log n)$. If k is small, this is a substantial improvement over the time required to find the k largest-valued records using one of the other sorting methods described earlier (many of which would require sorting all of the array first). One situation where we are able to take advantage of this concept is in the implementation of **Kruskal's algorithm** for **minimal-cost spanning trees**. That algorithm requires that edges be visited in ascending order (so, use a min-heap), but this process stops as soon as the MST is complete. Thus, only a relatively small fraction of the edges need be sorted.

Another special case arises when all of the records being sorted have the same key value. This represents the best case for Heapsort. This is because removing the smallest value requires only constant time, since the value swapped to the top is never pushed down the heap.

06.09 Sorting Summary Exercises

Due No Due Date **Points** 1 **Submitting** an external tool

06.09 Sorting Summary Exercises

6.9. Sorting Summary Exercises

6.9.1. Sorting Summary Exercises

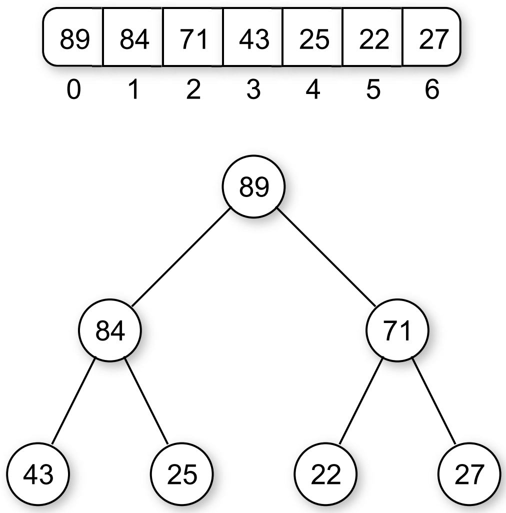
Here is a complete set of review questions, taken from all of the questions in the modules of this chapter.

Practicing Sorting Chapter Complete Review

Current score: 0 out of 5

Perform one iteration of heap sort. Swap the last key with the largest key and restore the heap in the array/tree shown below. Use the "Decrement" button to reduce the size of the heap.

Reset | Decrement Heap Size



Answer

Check Answer

Need help?

I'd like a hint

Chapter 7: Hashing

7.1. Introduction

7.1.1. Introduction

Hashing is a method for storing and retrieving records from a database. It lets you insert, delete, and search for records based on a search key value. When properly implemented, these operations can be performed in constant time. In fact, a properly tuned hash system typically looks at only one or two records for each search, insert, or delete operation. This is far better than the $O(\log n)$ average cost required to do a binary search on a sorted array of n records, or the $O(\log n)$ average cost required to do an operation on a binary search tree. However, even though hashing is based on a very simple idea, it is surprisingly difficult to implement properly. Designers need to pay careful attention to all of the details involved with implementing a hash system.

A hash system stores records in an array called a **hash table**, which we will call `HT`. Hashing works by performing a computation on a search key `K` in a way that is intended to identify the position in `HT` that contains the record with key `K`. The function that does this calculation is called the **hash function**, and will be denoted by the letter **h**. Since hashing schemes place records in the table in whatever order satisfies the needs of the address calculation, records are not ordered by value. A position in the hash table is also known as a **slot**. The number of slots in hash table `HT` will be denoted by the variable M with slots numbered from 0 to $M - 1$.

The goal for a hashing system is to arrange things such that, for any key value `K` and some hash function h, $i = \mathbf{h}(K)$ is a slot in the table such that $0 <= i < M$, and we have the key of the record stored at `HT[i]` equal to `K`.

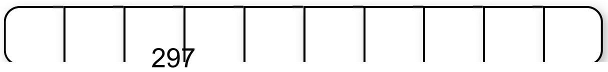
Hashing is not good for applications where multiple records with the same key value are permitted. Hashing is not a good method for answering range searches. In other words, we cannot easily find all records (if any) whose key values fall within a certain range. Nor can we easily find the record with the minimum or maximum key value, or visit the records in key order. Hashing is most appropriate for answering the question, ‘What record, if any, has key value `K`?’ **For applications where all search is done by exact-match queries, hashing is the search method of choice because it is extremely efficient when implemented correctly.** As this tutorial shows, however, there are many approaches to hashing and it is easy to devise an inefficient implementation. Hashing is suitable for both in-memory and disk-based searching and is one of the two most widely used methods for organizing large databases stored on disk (the other is the B-tree).

As a simple (though unrealistic) example of hashing, consider storing n records, each with a unique key value in the range 0 to $n - 1$. A record with key `k` can be stored in `HT[k]`, and so the hash function is $\mathbf{h}(k) = k$. To find the record with key value `k`, look in `HT[k]`.

1 / 10

<< < > >>

We will demonstrate the simplest hash function, storing records in an array of size 10.



In most applications, there are many more values in the key range than there are slots in the hash table. For a more realistic example, suppose the key can take any value in the range 0 to 65,535 (i.e., the key is a two-byte unsigned integer), and that we expect to store approximately 1000 records at any given time. It is impractical in this situation to use a hash table with 65,536 slots, because then the vast majority of the slots would be left empty. Instead, we must devise a hash function that allows us to store the records in a much smaller table. Because the key range is larger than the size of the table, at least some of the slots must be mapped to from multiple key values. Given a hash function **h** and two keys k_1 and k_2, if $\mathbf{h}(k_1) = \beta = \mathbf{h}(k_2)$ where β is a slot in the table, then we say that k_1 and k_2 have a **collision** at slot β under hash function **h**.

Finding a record with key value K in a database organized by hashing follows a two-step procedure:

07.02 Hash Function Principles

Due No Due Date **Points** 2 **Submitting** an external tool

07.02 Hash Function Principles

7.2. Hash Function Principles

7.2.1. Hash Function Principles

Hashing generally takes records whose key values come from a large range and stores those records in a table with a relatively small number of slots. Collisions occur when two records hash to the same slot in the table. If we are careful—or lucky—when selecting a hash function, then the actual number of collisions will be few. Unfortunately, even under the best of circumstances, collisions are nearly unavoidable. To illustrate, consider a classroom full of students. What is the probability that some pair of students shares the same birthday (i.e., the same day of the year, not necessarily the same year)? If there are 23 students, then the odds are about even that two will share a birthday. This is despite the fact that there are 365 days in which students can have birthdays (ignoring leap years). On most days, no student in the class has a birthday. With more students, the probability of a shared birthday increases. The mapping of students to days based on their birthday is similar to assigning records to slots in a table (of size 365) using the birthday as a hash function. Note that this observation tells us nothing about *which* students share a birthday, or on *which* days of the year shared birthdays fall.

Try it for yourself. You can use the calculator to see the probability of a collision. The default values are set to show the number of people in a room such that the chance of a duplicate is just over 50%. But you can set any table size and any number of records to determine the probability of a collision under those conditions.

Calculate the probability of a collision. About

Formula Used: $1 - \frac{t!}{(t-n)!(t^n)}$

where t is the table size and n is the number of records inserted.

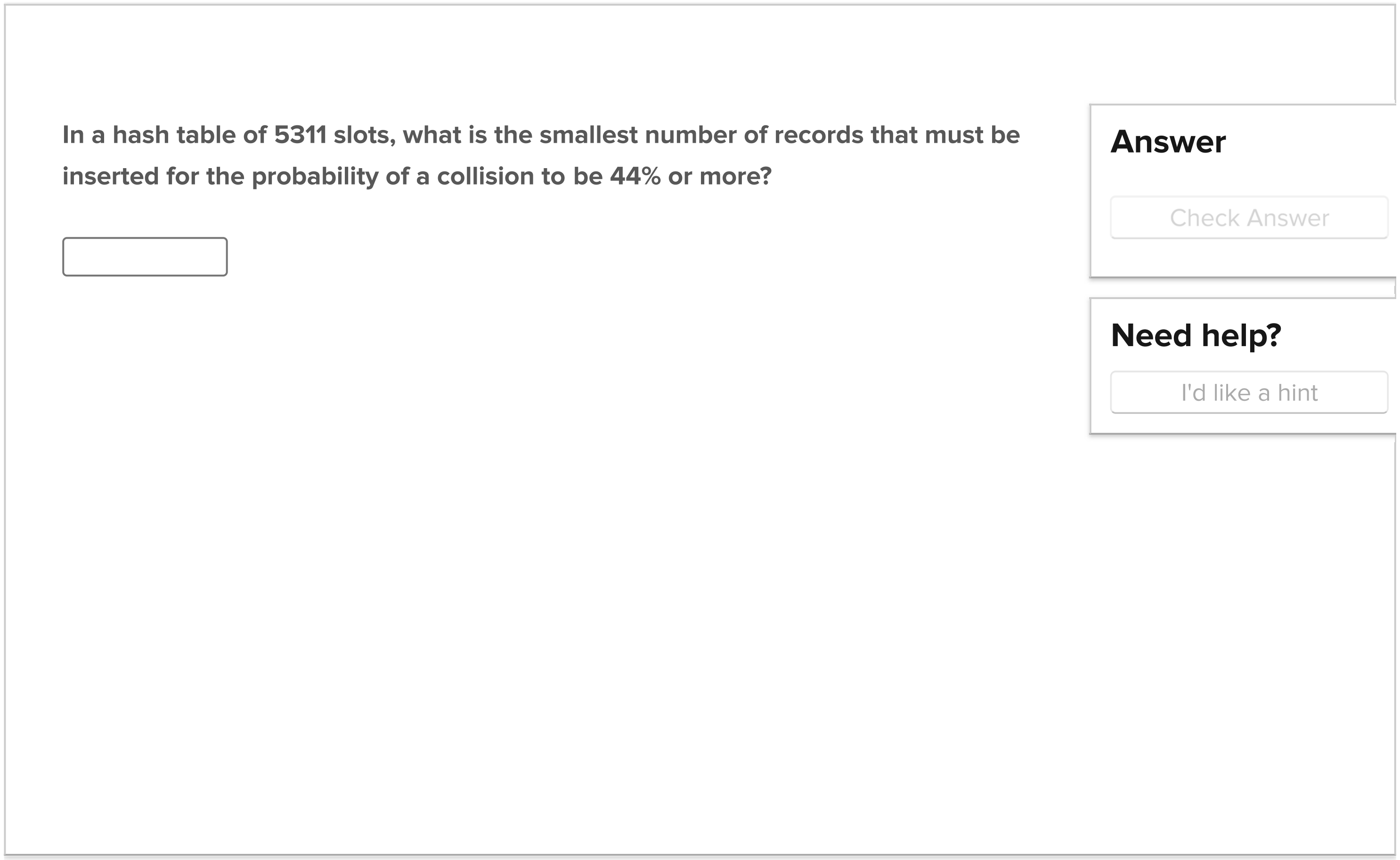
Table size:	365
# of records:	23

Calculate

Use the calculator to answer the following questions.

Practicing Hash Table Collision Probability Exercise

Current score: **0** out of **5**



In a hash table of 5311 slots, what is the smallest number of records that must be inserted for the probability of a collision to be 44% or more?

To be practical, a database organized by hashing must store records in a hash table that is not so large that it wastes space. To balance time and space efficiency, this means that the hash table should be **around half full**. Because collisions are extremely likely to occur under these conditions (by chance, any record inserted into a table that is half full should have a collision half of the time), does this mean that we need not worry about how well a hash function does at avoiding collisions? Absolutely not. The difference between using a good hash function and a bad hash function makes a big difference in practice in the number of records that must be examined when searching or inserting to the table. Technically, any function that maps all possible key values to a slot in the hash table is a hash function. In the extreme case, even a function that maps all records to the same slot in the array is a hash function, but it does nothing to help us find records during a search operation.

We would like to pick a hash function that maps keys to slots in a way that makes each slot in the hash table have equal probablility of being filled for the actual set keys being used. Unfortunately, we normally have no control over the distribution of key values for the actual records in a given database or collection. So how well any particular hash function does depends on the actual distribution of the keys used within the allowable key range. In some cases, incoming data are well distributed across their key range. For example, if the input is a set of random numbers selected uniformly from the key range, any hash function that assigns the key range so that each slot in the hash table receives an equal share of the range will likely also distribute the input records uniformly within the table. However, in many applications the incoming records are highly clustered or otherwise poorly distributed. When input records are not well distributed throughout the key range it can be difficult to devise a hash function that does a good job of distributing the records throughout the table, especially if the input distribution is not known in advance.

There are many reasons why data values might be poorly distributed.

1. Natural frequency distributions tend to follow a common pattern where a few of the entities occur frequently while most entities occur relatively rarely. For example, consider the populations of the 100 largest cities in the United States. If you plot these populations on a numberline, most of them will be clustered toward the low side, with a few outliers on the high side. This is an example of a Zipf distribution. Viewed the other way, the home town for a given person is far more likely to be a particular large city than a particular small town.
2. Collected data are likely to be skewed in some way. Field samples might be rounded to, say, the nearest 5 (i.e., all numbers end in 5 or 0).
3. If the input is a collection of common English words, the beginning letter will be poorly distributed.

Note that for items 2 and 3 on this list, either high- or low-order bits of the key are poorly distributed.

When designing hash functions, we are generally faced with one of two situations:

1. We know nothing about the distribution of the incoming keys. In this case, we wish to select a hash function that evenly distributes the key range across the hash table, while avoiding obvious opportunities for clustering such as hash functions that are sensitive to the high- or low-order bits of the key value.
2. We know something about the distribution of the incoming keys. In this case, we should use a distribution-dependent hash function that avoids assigning clusters of related key values to the same hash table slot. For example, if hashing English words, we should *not* hash on the value of the first character because this is likely to be unevenly distributed.

In the next module, you will see several examples of hash functions that illustrate these points.

07.03 Sample Hash Functions

Due No Due Date **Points** 5 **Submitting** an external tool

07.03 Sample Hash Functions

7.3. Sample Hash Functions

7.3.1. Sample Hash Functions

7.3.1.1. Simple Mod Function

Consider the following hash function used to hash integers to a table of sixteen slots.

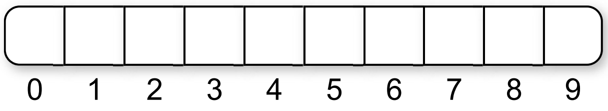
Toggle Tree View

```
int h(int x) {
  return x % 16;
}
```

Here "%" is the symbol for the mod function.

1 / 10 << < > >>

We will demonstrate the mod hash function. To make the compuation easy (because you can probably do mc
your head easily) we will store records in an array of size 10.



Recall that the values 0 to 15 can be represented with four bits (i.e., 0000 to 1111). The value returned by this hash function depends solely on the least significant four bits of the key. Because these bits are likely to be poorly distributed (as an example, a high percentage of the keys might be even numbers, which means that the low order bit is zero), the result will also be poorly distributed. This example shows that the size of the table M can have a big effect on the performance of a hash system because the table size is typically used as the modulus to ensure that the hash function produces a number in the range 0 to $M - 1$.

7.3.1.2. Binning

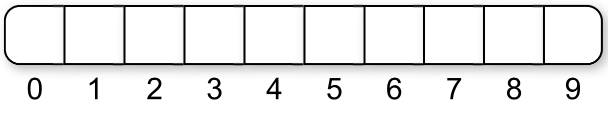
Say we are given keys in the range 0 to 999, and have a hash table of size 10. In this case, a possible hash function might simply divide the key value by 100. Thus, all keys in the range 0 to 99 would hash to slot 0, keys 100 to 199 would hash to slot 1, and so on. In other words, this hash function “bins” the first 100 keys to the first slot, the next 100 keys to the second slot, and so on.

Binning in this way has the problem that it will cluster together keys if the distribution does not divide evenly on the high-order bits. In the above example, if more records have keys in the range 900-999 (first digit 9) than have keys in the range 100-199 (first digit 1), more records will hash to slot 9 than to slot 1. Likewise, if we pick too big a value for the key range and the actual key values are all relatively small, then most records will hash to slot 0. A similar, analogous problem arises if we were instead hashing strings based on the first letter in the string.

1 / 8

<< < > >>

We will demonstrate the Binning hash function. To make the compuation easy (because you can probably divic your head easily) we will store records in an array of size 10.



In general with binning we store the record with key value i at array position i/X for some value X (using integer division). A problem with Binning is that we have to know the key range so that we can figure out what value to use for X. Let’s assume that the keys are all in the range 0 to 999. Then we want to divide key values by 100 so that the result is in the range 0 to 9. There is no particular limit on the key range that binning could handle, so long as we know the maximum possible value in advance so that we can figure out what to divide the key value by. Alternatively, we could also take the result of any binning computation and then mod by the table size to be safe. So if we have keys that are bigger than 999 when dividing by 100, we can still make sure that the result is in the range 0 to 9 with a mod by 10 step at the end.

Binning looks at the opposite part of the key value from the mod function. The mod function, for a power of two, looks at the low-order bits, while binning looks at the high-order bits. Or if you want to think in base 10 instead of base 2, modding by 10 or 100 looks at the low-order digits, while binning into an array of size 10 or 100 looks at the high-order digits.

As another example, consider hashing a collection of keys whose values follow a normal distribution, as illustrated by Figure 7.3.1. Keys near the mean of the normal distribution are far more likely to occur than keys near the tails of the distribution. For a given slot, think of where the keys come from within the distribution. Binning would be taking thick slices out of the distribution and assign those slices to hash table slots. If we use a hash table of size 8, we would divide the key range into 8 equal-width slices and assign each slice to a slot in the table. Since a normal would divide the key range into 8 equal-width slices and assign each slice to a slot in the table. Since a normal distribution is more likely to generate keys from the middle slice, the middle slot of the table is most likely to be used. In contrast, if we use the mod function, then we are assigning to any given slot in the table a series of thin slices in steps of 8. In the normal distribution, some of these slices associated with any given slot are near the tails, and some are near the center. Thus, each table slot is equally likely (roughly) to get a key value.

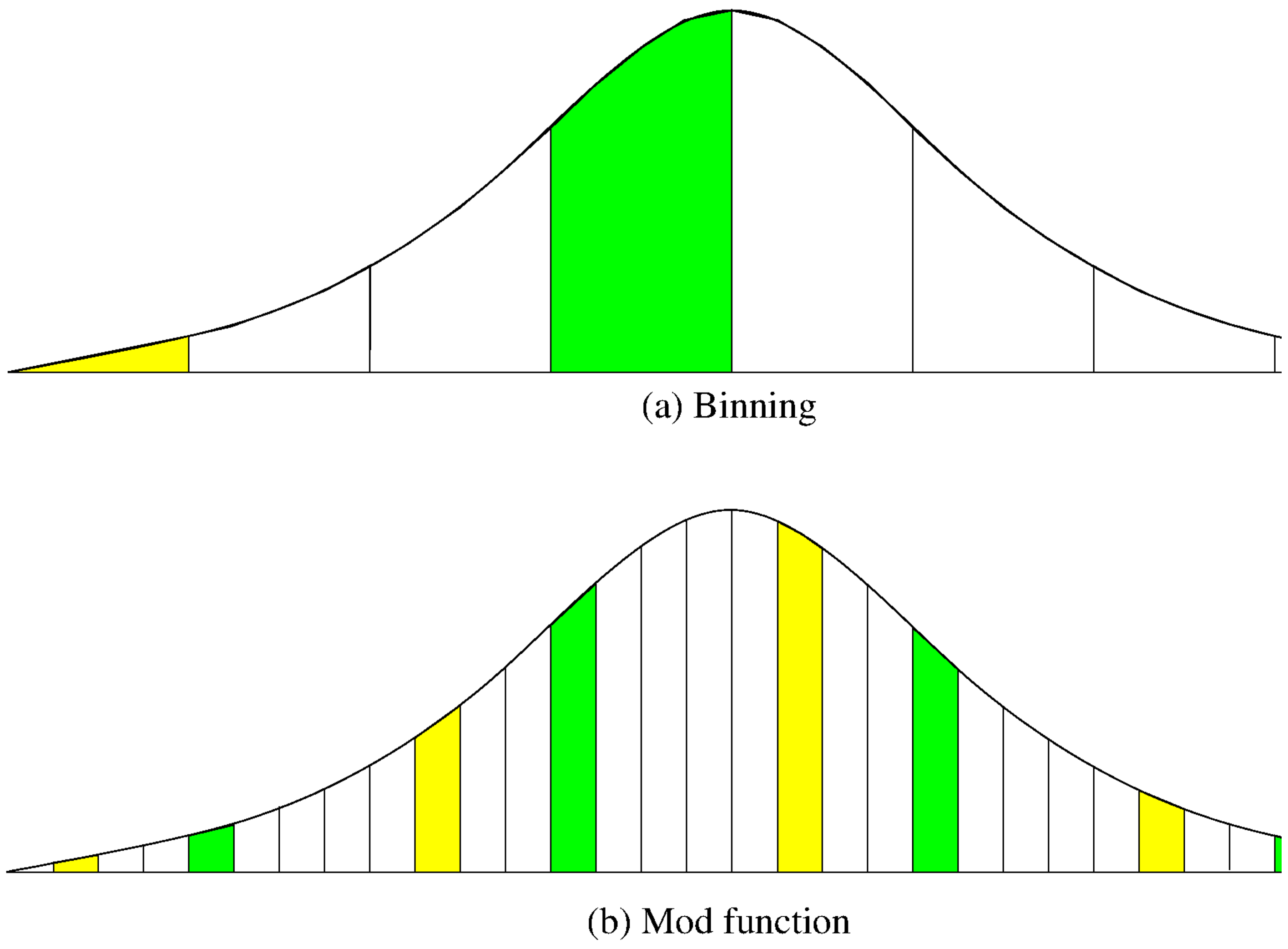


Figure 7.3.1: A comparison of binning vs. modulus as a hash function.

7.3.1.3. The Mid-Square Method

A good hash function to use with integer key values is the **mid-square method**. The mid-square method squares the key value, and then takes out the middle r bits of the result, giving a value in the range 0 to $2^r - 1$. This works well because most or all bits of the key value contribute to the result. For example, consider records whose keys are 4-digit numbers in base 10, as shown in Figure 7.3.2. The goal is to hash these key values to a table of size 100 (i.e., a range of 0 to 99). This range is equivalent to two digits in base 10. That is, $r = 2$. If the input is the number 4567, squaring yields an 8-digit number, 20857489. The middle two digits of this result are 57. All digits of the original key value (equivalently, all bits when the number is viewed in binary) contribute to the middle two digits of the squared value. Thus, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value. Of course, if the key values all tend to be small numbers, then their squares will only affect the low-order digits of the hash value.

4567
4567

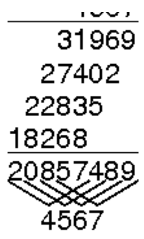


Figure 7.3.2: An example of the mid-square method. This image shows the traditional gradeschool long multiplication process. The value being squared is 4567. The result of squaring is 20857489. At the bottom, of the image, the value 4567 is show again, with each digit at the bottom of a “V”. The associated “V” is showing the digits from the result that are being affected by each digit of the input. That is, “4” affects the output digits 2, 0, 8, 5, an 7. But it has no affect on the last 3 digits. The key point is that the middle two digits of the result (5 and 7) are affected by every digit of the input.

Here is a little calculator for you to see how this works. Start with ‘4567’ as an example.

Type a number. This is tuned for 4-digit numbers, but you can use any value. The number will be squared, and the two middle digits (if the result is an 8-digit number) are highlighted.

Key Value: Calculate

7.3.2. A Simple Hash Function for Strings

Now we will examine some hash functions suitable for storing strings of characters. We start with a simple summation function.

Toggle Tree View

```
int sascii(String x, int M) {
  char ch[];
  ch = x.toCharArray();
  int xlength = x.length();

  int i, sum;
  for (sum=0, i=0; i < x.length(); i++) {
    sum += ch[i];
  }
  return sum % M;
}
```

This function sums the ASCII values of the letters in a string. If the hash table size M is small compared to the resulting summations, then this hash function should do a good job of distributing strings evenly among the hash table slots, because it gives equal weight to all characters in the string. This is an example of the **folding method** to designing a hash function. Note that the order of the characters in the string has no effect on the result. A similar method for integers would add the digits of the key value, assuming that there are enough digits to

1. keep any one or two digits with bad distribution from skewing the results of the process and
2. generate a sum much larger than M.

As with many other hash functions, the final step is to apply the modulus operator to the result, using table size M to generate a value within the table range. If the sum is not sufficiently large, then the modulus operator will yield a poor distribution. For example, because the ASCII value for ‘A’ is 65 and ‘Z’ is 90, `sum` will always be in the range 650 to 900 for a string of ten upper case letters. For a hash table of size 100 or less, a reasonable distribution results. For a hash table of size 1000, the distribution is terrible because only slots 650 to 900 can possibly be the home slot for some key value, and the values are not evenly distributed even within those slots.

Now you can try it out with this calculator.

Type a string. The sum of the ASCII values will be computed.

Key Value: Calculate

7.3.3. String Folding

Here is a much better hash function for strings.

Toggle Tree View

```
// Use folding on a string, summed 4 bytes at a time
int sfold(String s, int M) {
  long sum = 0, mul = 1;
  for (int i = 0; i < s.length(); i++) {
    mul = (i % 4 == 0) ? 1 : mul * 256;
    sum += s.charAt(i) * mul;
  }
  return (int)(Math.abs(sum) % M);
}
```

This function takes a string as input. It processes the string four bytes at a time, and interprets each of the four-byte chunks as a single long integer value. The integer values for the four-byte chunks are added together. In the end, the resulting sum is converted to the range 0 to $M-1$ using the modulus operator.

For example, if the string “aaaabbbb” is passed to `sfold`, then the first four bytes (“aaaa”) will be interpreted as the integer value 1,633,771,873, and the next four bytes (“bbbb”) will be interpreted as the integer value 1,650,614,882. Their sum is 3,284,386,755 (when treated as an unsigned integer). If the table size is 101 then the modulus function will cause this key to hash to slot 75 in the table.

Now you can try it out with this calculator.

Type a string. The sfold function will be computed.

Key Value: Calculate

For any sufficiently long string, the sum for the integer quantities will typically cause a 32-bit integer to overflow (thus losing some of the high-order bits) because the resulting values are so large. But this causes no problems when the goal is to compute a hash function.

The reason that hashing by summing the integer representation of four letters at a time is superior to summing one letter at a time is because the resulting values being summed have a bigger range. This still only works well for strings long enough (say at least 7-12 letters), but the original method would not work well for short strings either. There is nothing special about using four characters at a time. Other choices could be made. Another alternative would be to fold two characters at a time.

7.3.4. Hash Function Practice

Now here is an exercise to let you practice these various hash functions. You should use the calculators above for the more complicated hash functions.

Practicing Hash Functions: Proficency Summary

Current score: 0 out of 5

Given a hash table size of 100, a key in the range 0 to 9999, and using the **binning hash function**, what slot in the table will 5036 hash to?

Answer

Check Answer

Need help?

I'd like a hint

7.3.5. Hash Function Review Questions

Here are some review questions.

Practicing Hash Functions: Summary Questions

Current score: 0 out of 5

Answer TRUE or FALSE.

For the string hash functions, the size of the hash table limits the length of the string that can be hashed.

○ True

○ False

Answer

Check Answer

Need help?

I'd like a hint

07.04 Open Hashing

Due No Due Date **Points** 1 **Submitting** an external tool

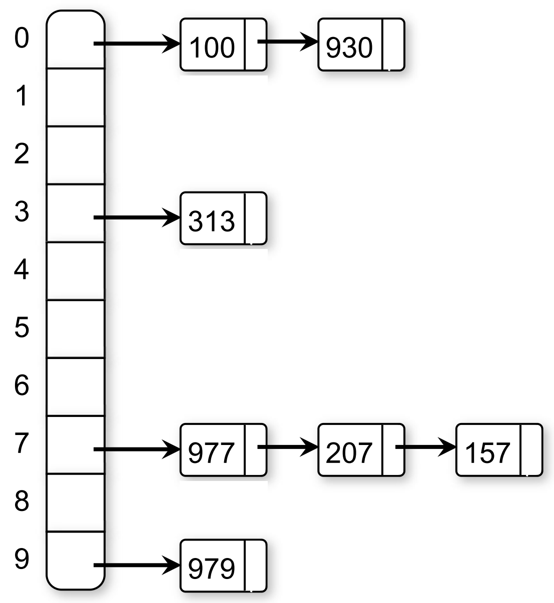
07.04 Open Hashing

7.4. Open Hashing

7.4.1. Open Hashing

While the goal of a hash function is to minimize collisions, some collisions are unavoidable in practice. Thus, hashing implementations must include some form of collision resolution policy. Collision resolution techniques can be broken into two classes: **open hashing** (also called **separate chaining**) and **closed hashing** (also called **open addressing**). (Yes, it is confusing when "open hashing" means the opposite of "open addressing", but unfortunately, that is the way it is.) The difference between the two has to do with whether collisions are stored outside the table (open hashing), or whether collisions result in storing one of the records at another slot in the table (closed hashing).

The simplest form of open hashing defines each slot in the hash table to be the head of a linked list. All records that hash to a particular slot are placed on that slot's linked list. The following figure illustrates a hash table where each slot points to a linked list to hold the records associated with that slot. The hash function used is the simple mod function.



Records within a slot's list can be ordered in several ways: by insertion order, by key value order, or by frequency-of-access order. Ordering the list by key value provides an advantage in the case of an unsuccessful search, because we know to stop searching the list once we encounter a key that is greater than the one being searched for. If records on the list are unordered or ordered by frequency, then an unsuccessful search will need to visit every record on the list.

Given a table of size M storing N records, the hash function will (ideally) spread the records evenly among the M positions in the table, yielding on average N/M records for each list. Assuming that the table has more slots than there are records to be stored, we can hope that few slots will contain more than one record. In the case where a list is empty or has only one record, a search requires only one access to the list. Thus, the average cost for hashing should be $\Theta(1)$. However, if clustering causes many records to hash to only a few of the slots, then the cost to access a record will be much higher because many elements on the linked list must be searched.

Open hashing is most appropriate when the hash table is kept in main memory, with the lists implemented by a standard in-memory linked list. Storing an open hash table on disk in an efficient way is difficult, because members of a given linked list might be stored on different disk blocks. This would result in multiple disk accesses when searching for a particular key value, which defeats the purpose of using hashing.

There are similarities between open hashing and Binsort. One way to view open hashing is that each record is simply placed in a bin. While multiple records may hash to the same bin, this initial binning should still greatly reduce the number of records accessed by a search operation. In a similar fashion, a simple Binsort reduces the number of records in each bin to a small number that can be sorted in some other way.

Practicing Open Hashing Proficiency Exercise

Current score: 0 out of 5

On the left is an array of numbers that are to be inserted (in order, from top to bottom) into the hash table on the right. The hash table uses open hashing to deal with collisions.

Move each record on the left to the appropriate bin on the right when using the simple mod hash function.

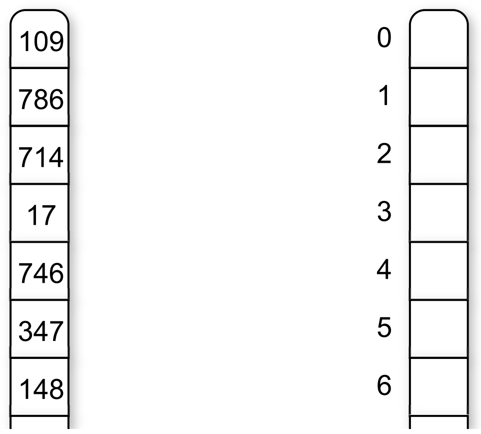
Reset

Answer

Check Answer

Need help?

I'd like a hint



873
7
8
9

07.05 Bucket Hashing

Due No Due Date **Points** 2 **Submitting** an external tool

07.05 Bucket Hashing

7.5. Bucket Hashing

7.5.1. Bucket Hashing

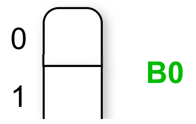
Closed hashing stores all records directly in the hash table. Each record R with key value k_R has a **home position** that is $\mathbf{h}(k_R)$, the slot computed by the hash function. If R is to be inserted and another record already occupies R's home position, then R will be stored at some other slot in the table. It is the business of the collision resolution policy to determine which slot that will be. Naturally, the same policy must be followed during search as during insertion, so that any record not found in its home position can be recovered by repeating the collision resolution process.

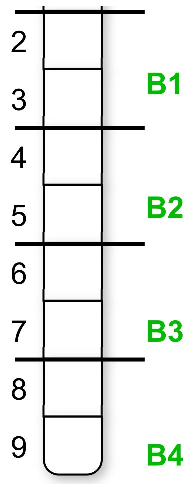
One implementation for closed hashing groups hash table slots into **buckets**. The M slots of the hash table are divided into B buckets, with each bucket consisting of M/B slots. The hash function assigns each record to the first slot within one of the buckets. If this slot is already occupied, then the bucket slots are searched sequentially until an open slot is found. If a bucket is entirely full, then the record is stored in an **overflow bucket** of infinite capacity at the end of the table. All buckets share the same overflow bucket. A good implementation will use a hash function that distributes the records evenly among the buckets so that as few records as possible go into the overflow bucket.

When searching for a record, the first step is to hash the key to determine which bucket should contain the record. The records in this bucket are then searched. If the desired key value is not found and the bucket still has free slots, then the search is complete. If the bucket is full, then it is possible that the desired record is stored in the overflow bucket. In this case, the overflow bucket must be searched until the record is found or all records in the overflow bucket have been checked. If many records are in the overflow bucket, this will be an expensive process.

1 / 27 << < > >>

Demonstration of bucket hash for an array of size 10 storing 5 buckets, each two slots in size. The alternatin
white cells indicate the buckets.

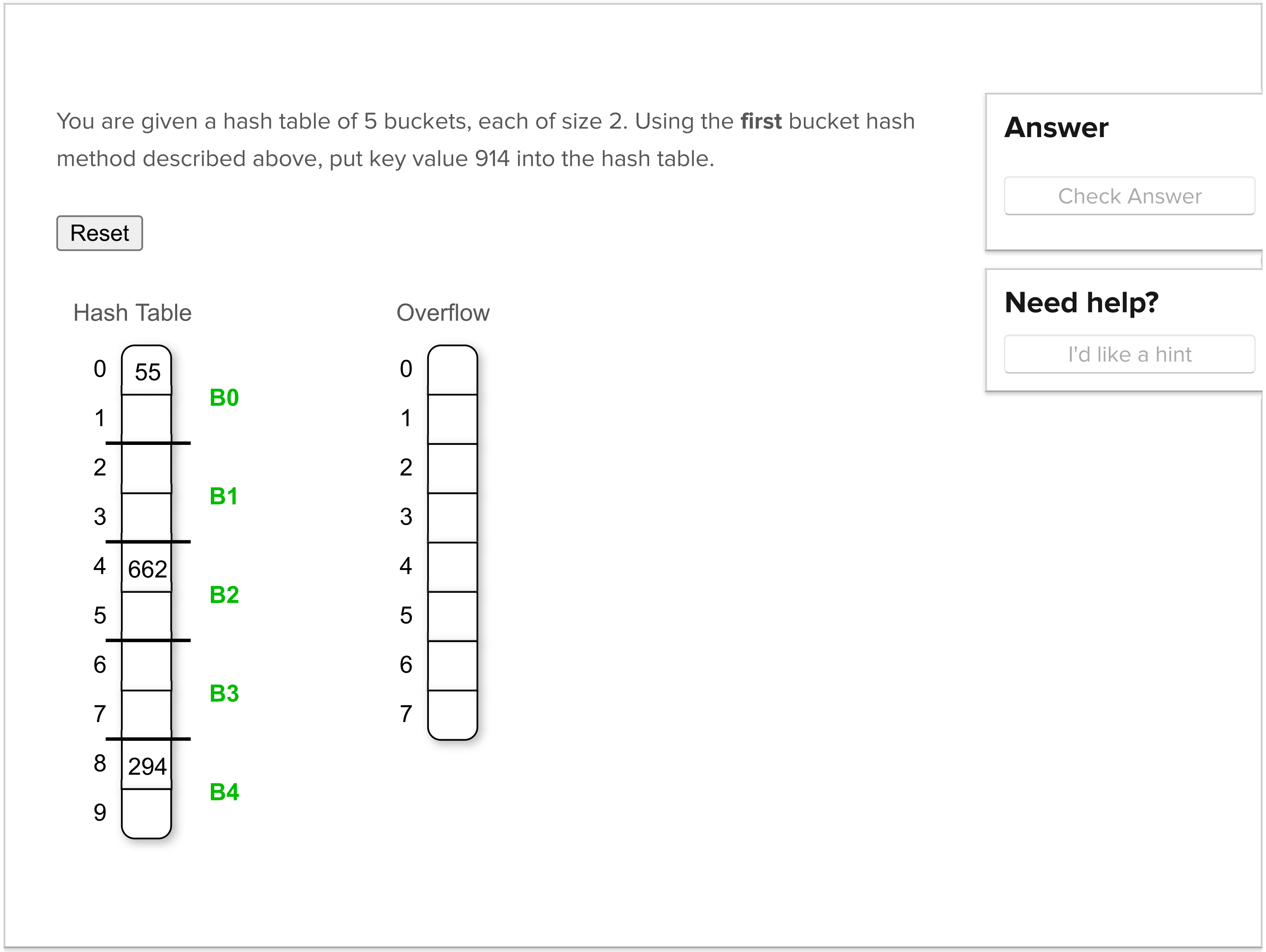




Now you can try it yourself.

Practicing Bucket Hashing Proficiency Exercise

Current score: 0 out of 5

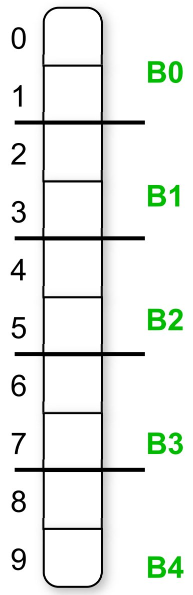

You are given a hash table of 5 buckets, each of size 2. Using the **first** bucket hash method described above, put key value 914 into the hash table.

7.5.2. An Alternate Approach

A simple variation on bucket hashing is to hash a key value to some slot in the hash table as though bucketing were not being used. If the home position is full, then we search through the rest of the bucket to find an empty slot. If all slots in this bucket are full, then the record is assigned to the overflow bucket. The advantage of this approach is that initial collisions are reduced, because any slot can be a home position rather than just the first slot in the bucket.

1 / 21 << < > >>

Demonstration of alternative bucket hash for an array of size 10 storing 5 buckets, each two slots in size. The gray and white cells indicate the buckets.



Bucket methods are good for implementing hash tables stored on disk, because the bucket size can be set to the size of a disk block. Whenever search or insertion occurs, the entire bucket is read into memory. Because the entire bucket is then in memory, processing an insert or search operation requires only one disk access, unless the bucket is full. If the bucket is full, then the overflow bucket must be retrieved from disk as well. Naturally, overflow should be kept small to minimize unnecessary disk accesses.

kept small to minimize unnecessary disk accesses.

Practicing Alternate Bucket Hashing Proficiency Exercise

Current score: 0 out of 5

You are given a hash table of 5 buckets, each of size 2. Using the **alternate** bucket hash method described above, put key value 157 into the hash table.

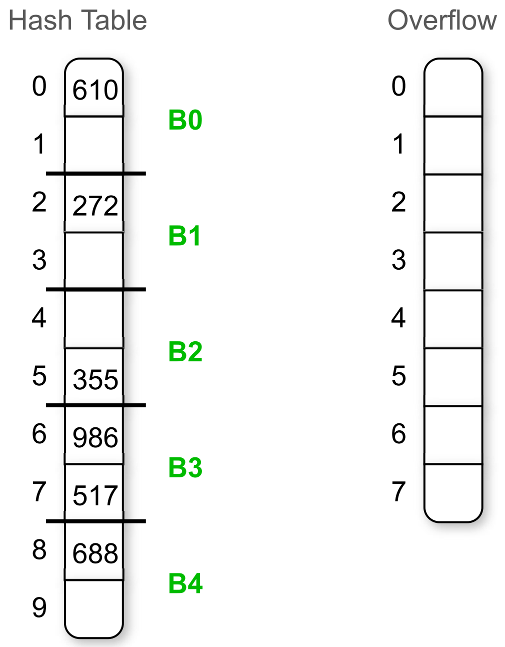
Reset

Answer

Check Answer

Need help?

I'd like a hint



07.06 Collision Resolution

Due No Due Date **Points** 1 **Submitting** an external tool

07.06 Collision Resolution

7.6. Collision Resolution

7.6.1. Collision Resolution

We now turn to the most commonly used form of hashing: **closed hashing** with no bucketing, and a **collision resolution policy** that can potentially use any slot in the hash table.

During insertion, the goal of **collision resolution** is to find a free slot in the hash table when the home position for the record is already occupied. We can view any collision resolution method as generating a sequence of hash table slots that can potentially hold the record. The first slot in the sequence will be the home position for the key. If the home position is occupied, then the collision resolution policy goes to the next slot in the sequence. If this is occupied as well, then another slot must be found, and so on. This sequence of slots is known as the **probe sequence**, and it is generated by some **probe function** that we will call **p**. Insertion works as follows.

Toggle Tree View

```
// Insert e into hash table HT
void hashInsert(Key k, Elem e) {
  int home;                     // Home position for e
  int pos = home = h(k);        // Init probe sequence
  for (int i=1; EMPTYKEY != (HT[pos]).key(); i++) {
    pos = (home + p(k, i)) % M; // probe
    if (k == HT[pos].key()) {
      println("Duplicates not allowed");
      return;
    }
  }
  HT[pos] = e;
}
```

Method `hashInsert` first checks to see if the home slot for the key is empty. If the home slot is occupied, then we use the probe function $\mathbf{p}(k, i)$ to locate a free slot in the table. Function **p** has two parameters, the key k and a count i of where in the probe sequence we wish to be. That is, to get the first position in the probe sequence after the home slot for key K, we call $\mathbf{p}(K, 1)$. For the next slot in the probe sequence, call $\mathbf{p}(K, 2)$. Note that the probe function returns an offset from the original home position, rather than a slot in the hash table. Thus, the `for` loop in `hashInsert` is computing positions in the table at each iteration by adding the value returned from the probe function to the home position. The i th call to **p** returns the i th offset to be used.

Searching in a hash table follows the same probe sequence that was followed when inserting records. In this way, a record not in its home position can be recovered. An implementation for the search procedure is as follows.

Toggle Tree View

```
// Search for the record with Key K
bool hashSearch(Key K, Elem e) {
  int home;              // Home position for K
  int pos = home = h(K); // Initial position is the home slot
  for (int i = 1;
       (K != (HT[pos]).key()) && (EMPTYKEY != (HT[pos]).key());
       i++) {
    pos = (home + p(K, i)) % M; // Next on probe sequence
       }
  if (K == (HT[pos]).key()) {   // Found it
    e = HT[pos];
    return true;
  }
  else { return false; }             // K not in hash table
}
```

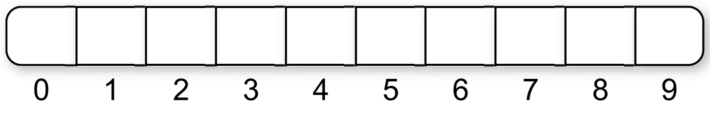
Both the insert and the search routines assume that at least one slot on the probe sequence of every key will be empty. Otherwise they will continue in an infinite loop on unsuccessful searches. Thus, the hash system should keep a count of the number of records stored, and refuse to insert into a table that has only one free slot.

The simplest approach to collsion resolution is simply to move down the table from the home slot until a free slot is found. This is known as **linear probing**. The probe function for simple linear probing is $\mathbf{p}(K, i) = i$. That is, the i th offset on the probe sequence is just i, meaning that the i th step is simply to move down i slots in the table. Once the bottom of the table is reached, the probe sequence wraps around to the beginning of the table (since the last step is to mod the result to the table size). Linear probing has the virtue that all slots in the table will be candidates for inserting a new record before the probe sequence returns to the home position.

1 / 11

<< < > >>

The simplest collsion resolution method is called linear probing. We simply move to the right in the table from slot, wrapping around to the beginning if necessary.



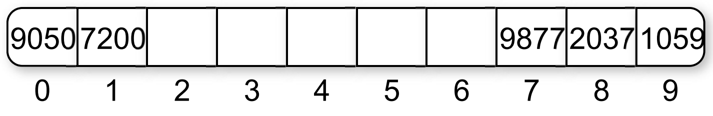
Can you see any reason why this might not be the best approach to collision resolution?

7.6.1.1. The Problem with Linear Probing

While linear probing is probably the first idea that comes to mind when considering collision resolution policies, it is not the only one possible. Probe function **p** allows us many options for how to do collision resolution. In fact, linear probing is one of the worst collision resolution methods. The main problem is illustrated by the next slideshow.

1 / 7 << < > >>

Consider the situation where we left off in the last slide show. If at this point we wanted to insert the value would have to probe all the way to slot 2.



Again, the ideal behavior for a collision resolution mechanism is that each empty slot in the table will have equal probability of receiving the next record inserted (assuming that every slot in the table has equal probability of being hashed to initially). This tendency of linear probing to cluster items together is known as **primary clustering**. Small clusters tend to merge into big clusters, making the problem worse. The objection to primary clustering is that it leads to long probe sequences.

10 "h(k) = k mod " + arrSize probeCommon.initJSAV("HashLinearPPRO", arrSize)

Given the following hash table, use hash function *hashFunction* and handle collisions using Linear Probing.

In which slot should the record with key value *probeCommon.currentKey* be inserted?

Reset
[probeCommon.userInput]
if (!probeCommon.checkAnswer(arrSize) && !guess[0]) { return ""; // User did not click, and correct answer is not // initial array state } else { return probeCommon.checkAnswer(arrSize); }

Linear probing is the simple one.

First use the hash function to computer the home slot.

If there is a collsion, then just step to the right by one step at a time until an empty slot is found.

If we reach the end of the array, then cycle around to the beginning.

07.07 Improved Collision Resolution

Due No Due Date **Points** 4 **Submitting** an external tool

07.07 Improved Collision Resolution

7.7. Improved Collision Resolution

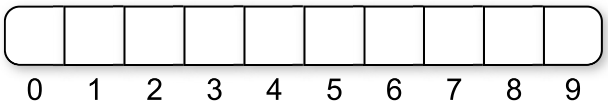
7.7.1. Linear Probing by Steps

How can we avoid primary clustering? One possible improvement might be to use linear probing, but to skip slots by some constant c other than 1. This would make the probe function $\mathbf{p}(K, i) = ci$, and so the i th slot in the probe sequence will be $(\mathbf{h}(K) + ic) \mod M$. In this way, records with adjacent home positions will not follow the same probe sequence.

1 / 10

<< < > >>

When doing collision resolution with linear probing by steps of size 2 on a hash table of size 10, a record that slot 4...



One quality of a good probe sequence is that it will cycle through all slots in the hash table before returning to the home position. Clearly linear probing (which "skips" slots by one each time) does this. Unfortunately, not all values for c will make this happen. For example, if $c = 2$ and the table contains an even number of slots, then any key whose home position is in an even slot will have a probe sequence that cycles through only the even slots. Likewise, the probe sequence for a key whose home position is in an odd slot will cycle through the odd slots. Thus, this combination of table size and linear probing constant effectively divides the records into two sets stored in two disjoint sections of the hash table. So long as both sections of the table contain the same number of records, this is not really important. However, just from chance it is likely that one section will become fuller than the other, leading to more collisions and poorer performance for those records. The other section would have fewer records, and thus better performance. But the overall system performance will be degraded, as the additional cost to the side that is more full outweighs the improved performance of the less-full side.

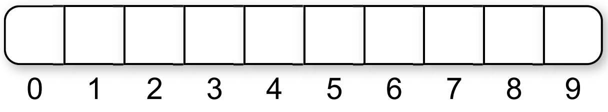
Constant c must be relatively prime to M to generate a linear probing sequence that visits all slots in the table (that

Constant c must be relatively prime to M to generate a linear probing sequence that visits all slots in the table (that is, c and M must share no factors). For a hash table of size $M = 10$, if c is any one of 1, 3, 7, or 9, then the probe sequence will visit all slots for any key. When $M = 11$, any value for c between 1 and 10 generates a probe sequence that visits all slots for every key.

1 / 11

<< < > >>

When doing collision resolution with linear probing by steps of size 3 on a hash table of size 10, a record that slot 4...



Now you can practice linear probing by different step sizes.

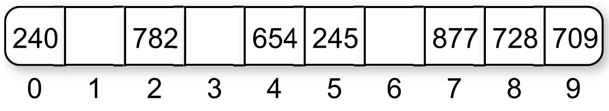
Practicing Hashing Linear Probing by Steps Proficiency Exercise

Current score: 0 out of 5

Given the following hash table, use hash function h(k) = k mod 10 and handle collisions using Linear Probing by Steps with probe function P(K, i) = 2i.

In which slot should the record with key value 845 be inserted?

Reset



Answer

Check Answer

Need help?

I'd like a hint

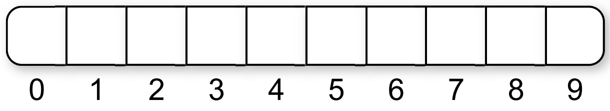
7.7.2. Pseudo-Random Probing

Consider the situation where $c = 2$ and we wish to insert a record with key k_1 such that $\mathbf{h}(k_1) = 3$. The probe sequence for k_1 is 3, 5, 7, 9, and so on. If another key k_2 has home position at slot 5, then its probe sequence will be 5, 7, 9, and so on. The probe sequences of k_1 and k_2 are linked together in a manner that contributes to clustering. In other words, linear probing with a value of $c > 1$ does not solve the problem of primary clustering. We would like to find a probe function that does not link keys together in this way. We would prefer that the probe sequence for k_1 after the first step on the sequence should not be identical to the probe sequence of k_2. Instead, their probe sequences should diverge.

The ideal probe function would select the next position on the probe sequence at random from among the unvisited slots; that is, the probe sequence should be a random permutation of the hash table positions. Unfortunately, we cannot actually select the next position in the probe sequence at random, because we would not be able to duplicate this same probe sequence when searching for the key. However, we can do something similar called **pseudo-random probing**. In pseudo-random probing, the i th slot in the probe sequence is $(\mathbf{h}(K) + r_i) \mod M$ where r_i is the i th value in a random permutation of the numbers from 1 to $M - 1$. All inserts and searches must use the same sequence of random numbers. The probe function would be $\mathbf{p}(K, i) = \mathbf{Permutation}[i]$ where **Permutation** is an array of length M that stores a value of 0 in position **Permutation[0]**, and stores a random permutation of the values from 1 to $M - 1$ in slots 1 to $M - 1$.

1 / 14 << < > >>

Let's see an example of collision resolution using pseudorandom probing on a hash table of size 10 using mod hash function.



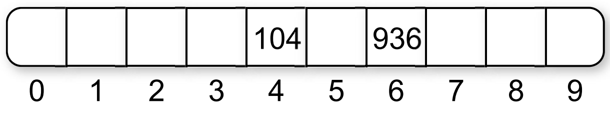
Here is a practice exercise for pseudo-random probing.

Pseudo-random probing exhibits another desirable feature in a hash function.

1 / 11

<< < > >>

First recall what happens with linear probing by steps of 2. Say that one record hashes to slot 4, and another slot 6.



7.7.3. Quadratic Probing

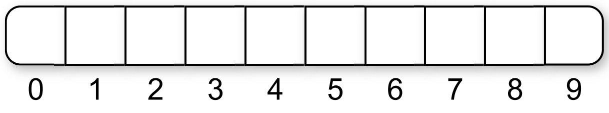
Another probe function that eliminates primary clustering is called **quadratic probing**. Here the probe function is some quadratic function $\mathbf{p}(K, i) = c_1 i^2 + c_2 i + c_3$ for some choice of constants c_1, c_2, and c_3.

The simplest variation is $\mathbf{p}(K, i) = i^2$ (i.e., $c_1 = 1$, $c_2 = 0$, and $c_3 = 0$). Then the i th value in the probe sequence would be $(\mathbf{h}(K) + i^2) \mod M$.

1 / 3

<< < > >>

Under quadratic probing, two keys with different home positions will have diverging probe sequences. Consic
that hashes to slot 5. Its probe sequence is 5, then 5 + 1 = 6, then 5 + 4 = 9, then (5 + 9) % 10 = 4, and so on.



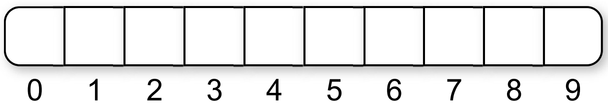
Now you can practice quadratic probing.

There is one problem with quadratic probing: Its probe sequence typically will not visit all slots in the hash table.

1 / 3

<< < > >>

Unfortunately, quadratic probing has the disadvantage that typically not all hash table slots will be on sequence.



For many hash table sizes, this probe function will cycle through a relatively small number of slots. If all slots on that cycle happen to be full, this means that the record cannot be inserted at all! A more realistic example is a table with 105 slots. The probe sequence starting from any given slot will only visit 23 other slots in the table. If all 24 of these slots should happen to be full, even if other slots in the table are empty, then the record cannot be inserted because the probe sequence will continually hit only those same 24 slots.

Fortunately, it is possible to get good results from quadratic probing at low cost. The right combination of probe function and table size will visit many slots in the table. In particular, if the hash table size is a prime number and the probe function is $\mathbf{p}(K, i) = i^2$, then at least half the slots in the table will be visited. Thus, if the table is less than half full, we can be certain that a free slot will be found. Alternatively, if the hash table size is a power of two and the probe function is $\mathbf{p}(K, i) = (i^2 + i)/2$, then every slot in the table will be visited by the probe function.

7.7.4. Double Hashing

Both pseudo-random probing and quadratic probing eliminate primary clustering, which is the name given to the the situation when keys share substantial segments of a probe sequence. If two keys hash to the same home position, however, then they will always follow the same probe sequence for every collision resolution method that we have seen so far. The probe sequences generated by pseudo-random and quadratic probing (for example) are entirely a function of the home position, not the original key value. This is because function **p** ignores its input parameter K for these collision resolution methods. If the hash function generates a cluster at a particular home position, then the cluster remains under pseudo-random and quadratic probing. This problem is called **secondary clustering**.

cluster remains under pseudo-random and quadratic probing. This problem is called **secondary clustering**.

To avoid secondary clustering, we need to have the probe sequence make use of the original key value in its decision-making process. A simple technique for doing this is to return to linear probing by a constant step size for the probe function, but to have that constant be determined by a second hash function, $\mathbf{h}_2$. Thus, the probe sequence would be of the form $\mathbf{p}(K, i) = i * \mathbf{h}_2(K)$. This method is called **double hashing**.

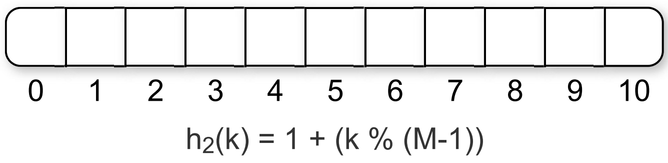
There are important restrictions on h_2. Most importantly, the value returned by h_2 must never be zero (or M) because that will immediately lead to an infinite loop as the probe sequence makes no progress. However, a good implementation of double hashing should also ensure that all of the probe sequence constants are relatively prime to the table size M. For example, if the hash table size were 100 and the step size for linear probing (as generated by function h_2) were 50, then there would be only one slot on the probe sequence. If instead the hash table size is 101 (a prime number), than any step size less than 101 will visit every slot in the table.

This can be achieved easily. One way is to select M to be a prime number, and have $\mathbf{h}_2$ return a value in the range $1 <= \mathbf{h}_2(k) <= M - 1$. We can do this by using this secondary hash function: $\mathbf{h}_2(k) = 1 + (k \mod (M - 1))$. An alternative is to set $M = 2^m$ for some value m and have $\mathbf{h}_2$ return an odd value between 1 and 2^m. We can get that result with this secondary hash function: $\mathbf{h}_2(k) = (((k/M) \mod (M/2)) * 2) + 1$. 1

1 / 9

<< < > >>

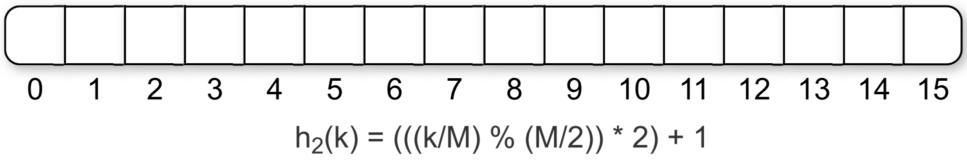
Let's see what happens when we use a hash table of size M = 11 (a prime number), our primary hash fu simple mod on the table size (as usual), and our secondary hash function is h$_2$(k) = 1 + (k % (M-1)).



1 / 7

<< < > >>

Now we try the alternate second hash function. Use a hash table of size M = 16 (a power of 2), our pri function is a simple mod on the table size (as usual), and our secondary hash function is h$_2$(k) = (((k/M) % (M 1.



Now you can try it.

1

The secondary hash function $\mathbf{h}_2(k) = (((k/M) \mod (M/2)) * 2) + 1$ might seem rather mysterious, so let's break this down. This is being used in the context of two facts: (1) We want the function to return an odd value that is less than M the hash table size, and (2) we are using a hash table of size $M = 2^m$, which means that taking the mod of size M is using the bottom m bits of the key value. OK, since $\mathbf{h}_2$ is multiplying something by 2 and adding 1, we guarentee that it is an odd number. Now, $((X \mod (M/2)) * 2) + 1$ must be in the range 1 and $M - 1$ (if you need to, play around with this on paper to convince yourself that this is true). This is exactly what we want. The last piece of the puzzle is the first part k/M. That is not strictly necessary. But remember that since the table size is $M = 2^m$, this is the same as shifting the key value right by m bits. In other words, we are since the table size is $M = 2^m$, this is the same as shifting the key value right by m bits. In other words, we are not using the bottom m bits to decide on the second hash function value, which is especially a good thing if we used the bottom m bits to decide on the first hash function value! In other words, we really do not want the value of the step sized used by the linear probing to be fixed to the slot in the hash table that we chose. So we are using the next m bits of the key value instead. Note that this would only be a good idea if we have keys in a large enough key range, that is, we want plenty of use of those second m bits in the key range. This will be true if the max key value uses at least $2m$ bits, meaning that the max key value should be at least the square of the hash table size. This is not a problem for typical hashing applications.

07.08 Analysis of Closed Hashing

Due No Due Date **Points** 1 **Submitting** an external tool

07.08 Analysis of Closed Hashing

7.8. Analysis of Closed Hashing

7.8.1. Analysis of Closed Hashing

How efficient is hashing? We can measure hashing performance in terms of the number of record accesses required when performing an operation. The primary operations of concern are insertion, deletion, and search. It is useful to distinguish between successful and unsuccessful searches. Before a record can be deleted, it must be found. Thus, the number of accesses required to delete a record is equivalent to the number required to successfully search for it. To insert a record, an empty slot along the record's probe sequence must be found. This is equivalent to an unsuccessful search for the record (recall that a successful search for the record during insertion should generate an error because two records with the same key are not allowed to be stored in the table).

When the hash table is empty, the first record inserted will always find its home position free. Thus, it will require only one record access to find a free slot. If all records are stored in their home positions, then successful searches will also require only one record access. As the table begins to fill up, the probability that a record can be inserted into its home position decreases. If a record hashes to an occupied slot, then the collision resolution policy must locate another slot in which to store it. Finding records not stored in their home position also requires additional record accesses as the record is searched for along its probe sequence. As the table fills up, more and more records are likely to be located ever further from their home positions.

From this discussion, we see that the expected cost of hashing is a function of how full the table is. Define the **load factor** for the table as $\alpha = N/M$, where N is the number of records currently in the table.

An estimate of the expected cost for an insertion (or an unsuccessful search) can be derived analytically as a function of α in the case where we assume that the probe sequence follows a random permutation of the slots in the hash table. Assuming that every slot in the table has equal probability of being the home slot for the next record, the probability of finding the home position occupied is α. The probability of finding both the home position occupied and the next slot on the probe sequence occupied is $(N(N-1))/(M(M-1))$. The probability of i collisions is $(N(N-1)\dots(N-i+1))/(M(M-1)\dots(M-i+1))$. If N and M are large, then this is approximately $(N/M)^i$. The expected number of probes is one plus the sum over $i >= 1$ of the probability of i collisions, which is approximately

$$1 + \sum_{i=1}^{\infty}(N/M)^i = 1/(1-\alpha).$$

The cost for a successful search (or a deletion) has the same cost as originally inserting that record. However, the expected value for the insertion cost depends on the value of α not at the time of deletion, but rather at the time of the original insertion. We can derive an estimate of this cost (essentially an average over all the insertion costs) by integrating from 0 to the current value of α, yielding a result of $(1/\alpha)\log_e 1/(1-\alpha)$.

It is important to realize that these equations represent the expected cost for operations when using the unrealistic assumption that the probe sequence is based on a random permutation of the slots in the hash table. We thereby avoid all the expense that results from a less-than-perfect collision resolution policy. Thus, these costs are lower-bound estimates in the average case. The true average cost under linear probing is $.5(1+1/(1-\alpha)^2)$ for insertions or unsuccessful searches and $.5(1+1/(1-\alpha))$ for deletions or successful searches.

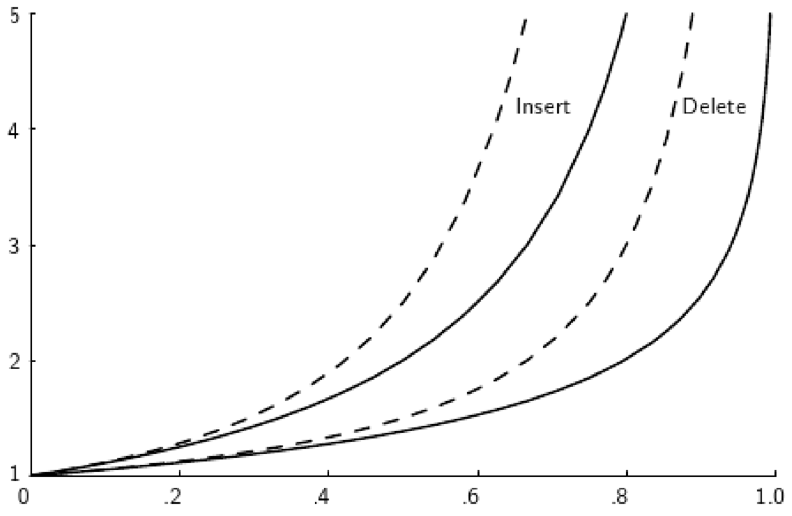


Figure 7.8.1: A plot showing the growth rate of the cost for insertion and deletion into a hash table as the load factor increases.

Figure **7.8.1** shows how the expected number of record accesses grows as α grows. The horizontal axis is the value for α , the vertical axis is the expected number of accesses to the hash table. Solid lines show the cost for “random” probing (a theoretical lower bound on the cost), while dashed lines show the cost for linear probing (a relatively poor collision resolution strategy). The two leftmost lines show the cost for insertion (equivalently, unsuccessful search); the two rightmost lines show the cost for deletion (equivalently, successful search).

From the figure, you should see that the cost for hashing when the table is not too full is typically close to one record access. This is extraordinarily efficient, much better than binary search which requires $\log n$ record accesses. As α increases, so does the expected cost. For small values of α, the expected cost is low. It remains below two until the hash table is about half full. When the table is nearly empty, adding a new record to the table does not increase the cost of future search operations by much. However, the additional search cost caused by each additional insertion increases rapidly once the table becomes half full. Based on this analysis, the rule of thumb is to design a hashing system so that the hash table never gets above about half full, because beyond that point performance will degrade rapidly. This requires that the implementor have some idea of how many records are likely to be in the table at maximum loading, and select the table size accordingly. The goal should be to make the table small enough so that it does not waste a lot of space on the one hand, while making it big enough to keep performance good on the other.

07.09 Deletion

Due No Due Date **Points** 2 **Submitting** an external tool

07.09 Deletion

7.9. Deletion

7.9.1. Deletion

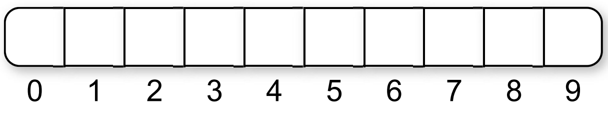
When deleting records from a hash table, there are two important considerations.

1. Deleting a record must not hinder later searches. In other words, the search process must still pass through the newly emptied slot to reach records whose probe sequence passed through this slot. Thus, the delete process cannot simply mark the slot as empty, because this will isolate records further down the probe sequence.
2. We do not want to make positions in the hash table unusable because of deletion. The freed slot should be available to a future insertion.

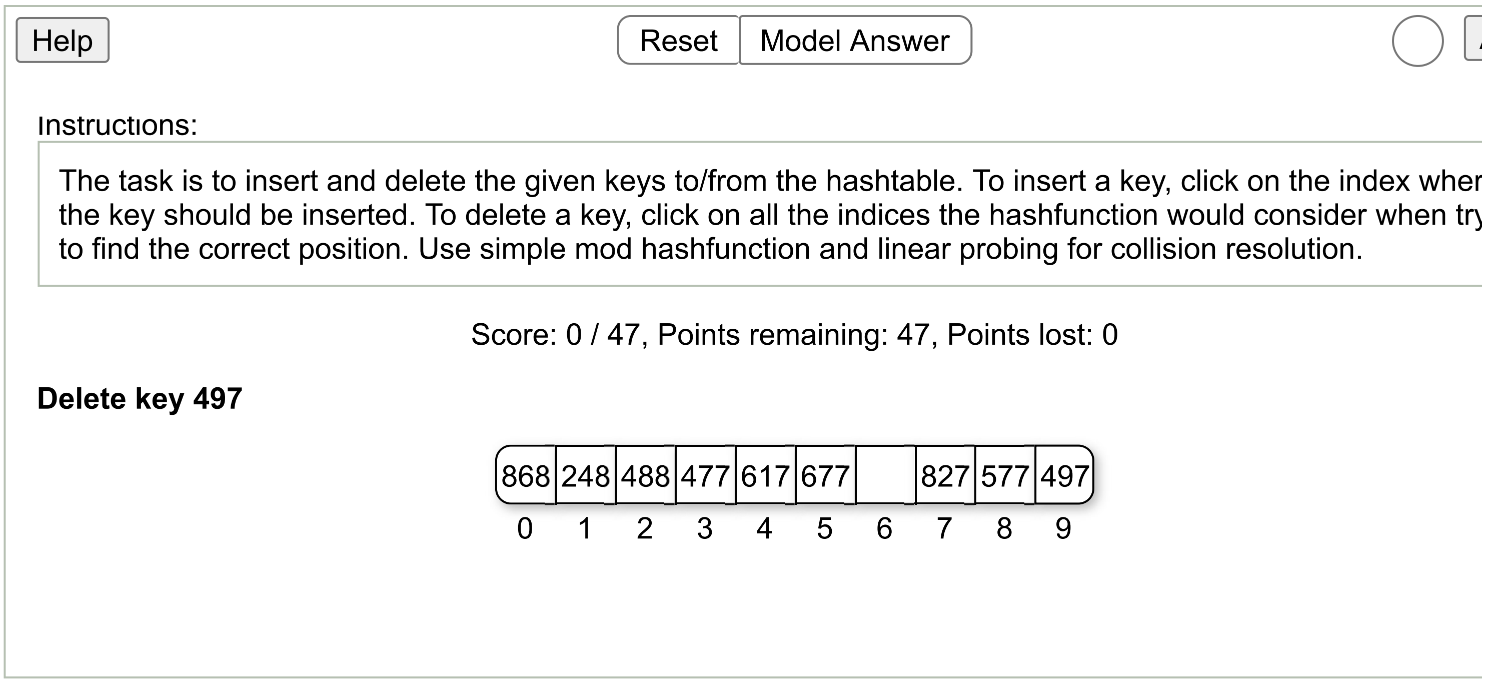
Both of these problems can be resolved by placing a special mark in place of the deleted record, called a **tombstone**. The tombstone indicates that a record once occupied the slot but does so no longer. If a tombstone is encountered when searching along a probe sequence, the search procedure continues with the search. When a tombstone is encountered during insertion, that slot can be used to store the new record. However, to avoid inserting duplicate keys, it will still be necessary for the search procedure to follow the probe sequence until a truly empty position has been found, simply to verify that a duplicate is not in the table. However, the new record would actually be inserted into the slot of the first tombstone encountered.

1 / 16 << < > >>

Let's see an example of the deletion process in action. As usual, our example will use a hash table of size 10, mod hash function, and collision resolution using simple linear probing.



Here is a practice exercise.



Instructions:

The task is to insert and delete the given keys to/from the hashtable. To insert a key, click on the index wher
the key should be inserted. To delete a key, click on all the indices the hashfunction would consider when try
to find the correct position. Use simple mod hashfunction and linear probing for collision resolution.

Score: 0 / 47, Points remaining: 47, Points lost: 0

Delete key 497

868	248	488	477	617	677		827	577	497
0	1	2	3	4	5	6	7	8	9

The use of tombstones allows searches to work correctly and allows reuse of deleted slots. However, after a series of intermixed insertion and deletion operations, some slots will contain tombstones. This will tend to lengthen the average distance from a record’s home position to the record itself, beyond where it could be if the tombstones did not exist. A typical database application will first load a collection of records into the hash table and then progress to a phase of intermixed insertions and deletions. After the table is loaded with the initial collection of records, the first few deletions will lengthen the average probe sequence distance for records (it will add tombstones). Over time, the average distance will reach an equilibrium point because insertions will tend to decrease the average distance by filling in tombstone slots. For example, after initially loading records into the database, the average path distance might be 1.2 (i.e., an average of 0.2 accesses per search beyond the home position will be required). After a series of insertions and deletions, this average distance might increase to 1.6 due to tombstones. This seems like a small increase, but it is three times longer on average beyond the home position than before deletions.

Two possible solutions to this problem are

1. Do a local reorganization upon deletion to try to shorten the average path length. For example, after deleting a key, continue to follow the probe sequence of that key and swap records further down the probe sequence into the slot of the recently deleted record (being careful not to remove any key from its probe sequence). This will not work for all collision resolution policies.

2. Periodically rehash the table by reinserting all records into a new hash table. Not only will this remove the tombstones, but it also provides an opportunity to place the most frequently accessed records into their home positions.

7.9.2. Hashing Deletion Summary Questions

Now here are some practice questions.

Congratulations! You have reached the end of the hashing tutorial. In summary, a properly tuned hashing system will return records with an average cost of less than two record accesses. This makes it the most effective way known to store a database of records to support exact-match queries. Unfortunately, hashing is not effective when implementing range queries, or answering questions like “Which record in the collection has the smallest key value?”

07.10 Hashing Chapter Summary Exercises

Due No Due Date **Points** 1 **Submitting** an external tool

07.10 Hashing Chapter Summary Exercises

7.10. Hashing Chapter Summary Exercises

7.10.1. Hashing Review

Here is a complete set of review questions, taken from all of the questions in the modules of this chapter. If anything goes wrong with one of the questions, or if you think that you are in a series of repeating questions, then just reload the page.

Chapter 8: Functional Programming and Streams

Functional Programming and Streams

John MacCormick, Dickinson College, August 2021.

1 Some background on functional programming

The design and implementation of programming languages is a large and important subfield of computer science. In this chapter, we examine *functional programming*, which is one of the most important ideas within the theory of programming languages. Programming languages are often described as belonging to various categories or *paradigms*. Examples of these programming language paradigms include *imperative* languages, *functional* languages, and *logic programming* languages. Most modern languages include features from multiple paradigms. For example, Java and Python were designed primarily as imperative languages, but they include many aspects of functional programming. Examples of languages that were designed as functional languages include Lisp, F#, and Haskell.

In this chapter, we first examine the most fundamental idea in functional programming: *lambda expressions*. Then, we see how lambda expressions can be used in Java's Stream API to process data sets efficiently and elegantly.

1.1 A note on the use of Java and Python in this chapter

Although Java and Python are not particularly good examples of functional programming languages, the examples in this chapter use only Java and Python. It would take us too far afield to study a more purely functional language. If some of the Java examples seem a little awkward, keep in mind that we are deliberately adopting a compromise between understanding the ideas of functional programming and exploiting our existing familiarity with Java. If you are not familiar with Python, do not be concerned. It is easier to demonstrate some of functional programming ideas in Python, compared to Java. Therefore, we use some initial examples from Python. But it will be sufficient to understand the Java examples without having a detailed understanding of the Python examples.

2 Lambda expressions

In the rest of this chapter, the word *function* usually refers to a subroutine in a programming language that can accept parameters and return values. Different programming languages refer to functions using different terminology. In Java, for example, functions are called *methods*. In Python, functions are simply called functions.

A key aspect of functional programming languages is that they treat functions the same as other data types. This is often described using the phrases "functions are first-class citizens" or "functions are first-class objects." The most important consequence of this first-class status is that in a functional language, a function can be a parameter in another function.

2.1 Functions as parameters in Python

It is particularly easy to demonstrate this in Python:

```
def add5(x):
    return x + 5

def multBy3IfPositive(x):
    if x > 0:
        return 3 * x
    else:
        return 0

def applyToSeven(f):
    return f(7)

def applyToMinusNine(f):
    return f(-9)
```

Here, the functions `applyToSeven` and `applyToMinusNine` both accept a single parameter which is expected to be a function. For example, we can use `add5` as the parameter for `applyToSeven`:

```
>>> applyToSeven(add5)
12
```

And of course, we can get a different result if we send in a different function as the parameter:

```
>>> applyToSeven(multBy3IfPositive)
21
```

Example problem 1.

Check your understanding by working out the results of the following two function calls:

```
>>> applyToMinusNine(add5)

>>> applyToMinusNine(multBy3IfPositive)
```

Solutions to all example problems are given at the end of this chapter.

2.2 Functions as parameters in Java

In Java, it is not quite so easy to pass a function as a parameter, when compared to Python. The Java code below is the simplest equivalent of the above Python code. As you will see, it is comparatively ugly. It relies on Java interfaces to simulate the ability to pass a function as a parameter. In this example, we import the `Function<T, R>` interface from `java.util.function`, which is the Java package that supports functional programming. The `Function<T, R>` interface represents a function that accepts a single parameter of type `T` and returns a value of type `R`. Because the `add5` function accepts a single integer parameter and returns an integer, we can create the effect of functional programming with `add5` by creating an `Add5` class that implements the `Function<Integer, Integer>` interface. In the `main()` method below, we create an instance of the `Add5` class. The name of that instance is `add5`, and we can pass this instance, which truly is a Java object, as a parameter to the `applyToSeven()` and `applyToMinusNine()` methods.

```
import java.util.function.Function;

public class FunctionParameterDemo {

      public static class Add5 implements Function<Integer, Integer> {
            public Integer apply(Integer x) {
                  return x + 5;
            }
      }

      public static class MultBy3IfPositive
                  implements Function<Integer, Integer> {
            public Integer apply(Integer x) {
                  if (x > 0) {
                        return 3 * x;
                  } else {
                        return 0;
                  }
            }
      }

      public static Integer applyToSeven(Function<Integer, Integer> f) {
            return f.apply(7);
      }

      public static Integer applyToMinusNine(Function<Integer, Integer> f) {
            return f.apply(-9);
      }

      public static void main(String[] args) {
            Add5 add5 = new Add5();
            MultBy3IfPositive multBy3IfPositive = new MultBy3IfPositive();
            int val1 = applyToSeven(add5); // val1 = 12
            int val2 = applyToSeven(multBy3IfPositive); // val2 = 21
      }
}
```

Example problem 2.

Determine the value of the following two method calls, assuming they were inserted at the end of the above `main()` method:

```
applyToMinusNine(add5)
applyToMinusNine(multBy3IfPositive)
```

2.3 Functional interfaces and lambda expressions in Java

Notice that the interface implemented by `add5` and `multBy3IfPositive` has the following special property: it has only a single abstract method. The name of this method is `apply()`. In Java, an interface with exactly one abstract method is called a *functional interface*. As we will see later, functional interfaces are important because they can be implemented efficiently and elegantly using *lambda expressions*. Lambda expressions are a shorthand notation for functional interfaces that allow us to avoid using the ugly style of Java functional programming above. To see lambda expressions in action, we will first return to Python.

2.4 Named and anonymous values

In any computer program written in any programming language, some of the values will typically be *named* whereas other values will be *anonymous*. For example, here are two different ways of printing out the square root of 5 in Python:

```
print(math.sqrt(5)) # anonymous

x = 5
print(math.sqrt(x)) # named
```

In the first call to `sqrt()`, the value 5 is anonymous because it is not referred to using the name of a variable. In the second call to `sqrt()`, the value 5 is referred to using the name `x`.

2.5 The lambda keyword in Python

Recall that in functional programming, we can use functions as parameters. Therefore, it should be possible to use the same two techniques when sending a function as a parameter: we can use either a named function, or an anonymous function. We have already seen how to do this using named functions. One of the examples from earlier is repeated here for concreteness:

```
>>> applyToSeven(add5)
```

This sends the named function `add5` as a parameter to an invocation of the function `applyToSeven()`. But how can we do this using an anonymous function? In Python, we can create an anonymous function using the keyword `lambda`, as in the following example.

```
>>> applyToSeven(lambda x: x+5)
```

So, a lambda expression is just a way of describing a function without giving it a name. The Python snippet "`lambda x: x+5`" means "the function that takes a parameter `x` and returns `x+5`." It would perhaps be less confusing if `lambda` were called something else like `anonFunction`, but there are good historical and theoretical reasons for adopting the keyword `lambda`. "Lambda" is the name of the Greek letter λ. In the 1930s, the mathematician Alonzo Church used the Greek letter λ as the main symbol in describing a framework for computing with functions—a framework that we now call the *lambda calculus*. The lambda calculus lies at the heart of functional programming and the theory of computation, but it would take us too far afield to study that connection here.

To summarize: in Python, a lambda expression defines an anonymous function. There is never any need to be confused by lambda expressions. If you see some code that contains a confusing lambda expression, you can simply rewrite it by first creating a named function that performs the effect of the lambda expression, then substituting the new name for the lambda expression. For example, the Python snippet

```
performStrangeAction("nonsense", \
      lambda apple, banana, grape: apple + banana.gimble(2*grape))
```

can be rewritten as

```
def weirdFunction(apple, banana, grape):
    return apple + banana.gimble(2*grape)

performStrangeAction("nonsense", weirdFunction)
```

The above example also shows how lambda expressions in Python can describe functions with multiple parameters.

Example problem 3.

Assuming the definitions given earlier, what is the output of the following snippets of Python?

```
applyToSeven(lambda potato: potato%4 + potato*potato)
applyToMinusNine(lambda oak: math.factorial(oak+12) * oak)
```

2.6 Lambda expressions in Java

In Java, lambda expressions do not use the keyword `lambda`. Instead, they employ the arrow notation "`->`" immediately after the function parameters. For example, the equivalent of the Python snippet "`lambda x: x+5`" in Java is "`x -> x+5`". In both cases, the meaning of the lambda expression is approximately "the function that receives a parameter `x` and returns `x+5`." The true meaning of a Java lambda expression is more complex, but we won't describe the details here. It is enough to know that the Java lambda expression uses the functional interfaces mentioned earlier, creating an anonymous class and an anonymous instance so that the desired functional programming effect is achieved.

We can rewrite our earlier Java examples using lambda expressions. The declarations of `applyToSeven()` and `applyToMinusNine()` remain the same, but they are repeated here for convenience:

```
      public static Integer applyToSeven(Function<Integer, Integer> f) {
            return f.apply(7);
      }

      public static Integer applyToMinusNine(Function<Integer, Integer> f) {
            return f.apply(-9);
      }
```

The rest of the code is much more compact because there is no need to declare any classes that implement functional interfaces. Our main method can instead be

```
      public static void main(String[] args) {
            int val3 = applyToSeven(x -> x + 5); // val3 = 12
            int val4 = applyToSeven(x -> {
                  if (x > 0) {
                        return 3 * x;
                  } else {
                        return 0;
                  }
            }); // val4 = 21
      }
```

Note the use of a lambda expression that spans multiple lines and contains multiple blocks of code. In practice, lambda expressions are usually kept short and simple, but they can be as complex as desired.

Example problem 4.

Determine the values of the following two method calls, assuming they were inserted at the end of the above `main()` method:

```
      int val5 = applyToMinusNine(x -> (x + 1) * (x + 2));
      int val6 = applyToMinusNine(z -> {
            if (z > 10) {
                  return 100;
            } else if (z < -100) {
                  return -100;
            } else {
                  return z * 10;
            }
      });
```

Lambda expressions in Java can also have multiple parameters. This is achieved by listing the parameters inside parentheses before the "`->`" symbol. For example, the lambda expression `(x, y) -> x*x + 2*y` could represent a method `f()` such as the following.

```
            public int f(int x, int y) {
                  return x*x + 2*y;
            }
```

Example problem 5.

Write the mathematical function $g(u, v, w) = \sqrt{u^2 + v^2 + w^2}$ as a Java lambda expression.

3 The Java Stream API

In Java, Stream<T> is an interface for performing operations on sequences of objects of type T. There are also specialized streams for performing operations on sequences of some primitive types: IntStream for int values, LongStream for long values, and DoubleStream for double values.

Common operations used on Streams include count(), filter(), map(), mapToInt(), foreach(), and reduce(). IntStream, LongStream, and DoubleStream also have the operation sum(). We study only these seven operations. The Java API documentation lists other available operations.

Each operation is defined as either an *intermediate operation* or a *terminal operation*. The seven common operations that we will study are classified as follows.

- Intermediate operations: filter(), map(), mapToInt().
- Terminal operations: count(), foreach(), reduce(), sum().

To perform a computation on a Stream, we can apply zero or more intermediate operations in sequence, followed by a single terminal operation. For example, a computation might consist of the following sequence of operations: filter(), filter(), map(), filter(), count().

3.1 Creating a Stream

There are numerous ways to create a Stream in Java. Several of these approaches are shown in the following code snippet.

```
// Approach 1: Create directly from an array via Stream.of()
String[] array = { "bat", "cat", "bird", "mad", "catch", "ditch" };
Stream<String> stream1 = Stream.of(array);

// Approach 2: Create directly from multiple arguments via
// String.of()
Stream<String> stream2 = Stream.of("bat", "cat", "bird", "mad",
            "catch", "ditch");

// Approach 3: Convert any Java Collection using the collection's
// stream() method
List<String> list = Arrays.asList("bat", "cat", "bird", "mad",
            "catch", "ditch");
Stream<String> stream3 = list.stream();

// Approach 4: Stream from a file using Files.lines
Stream<String> stream4 = Files.lines(Paths.get("data/words.txt"));

// Approach 5: Use range() or rangeClosed()
IntStream range1 = IntStream.range(5, 10); // 5,6,7,8,9
IntStream range2 = IntStream.rangeClosed(5, 10); // 5,6,7,8,9,10
```

Example problem 6.

(i) Write some code that would create a stream containing the following sequence as `Double` objects: 23.4, 69.7, -25.88, 31.3363.

(ii) Repeat part (i), this time creating a stream containing primitive `double` values.

(iii) Write code creating a stream consisting of integers from 100 to 200 inclusive.

The next seven sections introduce each of our seven common operations, while also gradually building our understanding of how to combine the operations into more interesting stream computations.

3.2 Stream operations

We now examine seven stream operations in detail.

3.2.1 count()

The `count()` operation returns the number of elements in a stream.

```
Stream<String> stream = Stream.of("bat", "cat", "bird”);
long numElements = stream.count(); // numElements = 3
```

Example problem 7.

Write a snippet of code that uses the Stream API to count the number of lines in a file called “`GreatGatsby.txt`”.

3.2.2 sum()

The `sum()` operation returns the sum of the elements in a stream. It can only be applied to the specialized numeric streams `IntStream`, `LongStream`, and `DoubleStream`, as in the following example.

```
DoubleStream stream = DoubleStream.of(1.5, 2.4, -0.1);
double total = stream.sum(); // total = 3.8
```

Example problem 8.

Write a snippet of code that uses the Stream API to add the integers from 27 to 159 inclusive. Hint: `IntStream.range()` or `IntStream.rangeClosed()` make this very easy.

3.2.3 filter()

The `filter()` operation applies a Boolean test function to every element in the input stream. Elements that fail the test (i.e. return `false`) are discarded, whereas elements that pass the test (i.e. return `true`) enter the output stream to be processed by the next operation in the computation. This is our first example of using lambda expressions with `Stream`s, because the Boolean test function can be described with a lambda expression. Consider the following example.

```
Stream<String> stream = Stream.of("bat", "cat", "bird", "mad",
                        "catch", "ditch");
Stream<String> newStream = stream.filter(word -> word.startsWith("ca"));
```

Here, `stream` is the input to the computation. It is a `Stream<String>` containing the elements "bat", "cat", "bird", "mad", "catch", "ditch". The output of the computation is `newStream`, which also has the datatype `Stream<String>`. But `newStream` contains only the elements beginning with “ca”, so it consists of the two elements “cat” and “catch”.

If the lambda expression in the snippet above is confusing, remember that we can always rewrite lambda expressions using named functions. Let’s do this now for the above lambda expression, `word -> word.startsWith("ca")`.

The first step is to determine what functional interface this lambda expression is an instance of. To do that, we consult the documentation of the `filter()` method in the `Stream` interface. In the documentation, the signature of the `filter()` method is given as

```
Stream<T> filter(Predicate<? super T> predicate)
```

Until we have more familiarity with streams and lambda expressions, it will be best to ignore the type wildcard (“`?`”). So let’s think of the parameter as having datatype `Predicate<T>`. We again consult the official Java documentation, this time looking up `Predicate<T>`. As expected, it is a functional interface, which means it has exactly one abstract method. The signature of this method is `boolean test(T t)`. Note that our `filter()` method is being applied to a stream of strings, `Stream<String>`. So for this example, the type parameter `T` has value `String`. Therefore, to rewrite the lambda expression `word -> word.startsWith("ca")`, we need to implement the interface `Predicate<String>`. Specifically, we will need to implement the method `test(String t)` so that its parameter is called `word` and its return value is `word.startsWith("ca")`. Putting this together, we obtain the following code.

```
class StartsWithCA implements Predicate<String> {
      public boolean test(String word) {
            return word.startsWith("ca");
      }
}
StartsWithCA startsWithCA = new StartsWithCA();
Stream<String> stream = Stream.of("bat", "cat", "bird", "mad",
                        "catch", "ditch");
Stream<String> newstream = stream.filter(startsWithCA);
```

Once we have obtained the new, filtered stream `newstream`, we can do further computations on it. For example, we can count the elements:

```
Stream<String> newstream = stream.filter(word -> word.startsWith("ca"));
long numStartWithCA = newstream.count(); // numStartWithCA = 2
```

However, this is usually written in a more compact form. Instead of defining a new local variable for each stream created by the intermediate operations, we can immediately apply the next operation via a method call:

```
long numStartWithCA = stream
            .filter(word -> word.startsWith("ca"))
            .count(); // numStartWithCA = 2
```

This is equivalent to the previous snippet, but it is more compact and more readable once you get used to the syntax. Each operation in the computation is written as a method call beginning with "`.`" and placed on a new line for readability.

Example problem 9.

Building on your solution to the previous example problem, write a snippet of code that uses the Stream API to add the *odd* integers from 27 to 159 inclusive.

3.2.4 foreach()

The `foreach()` operation applies an action to every element in the input stream. It is a terminal operation and should only be used for producing output at the end of a computation. Never use the `foreach()` operation to process elements in an intermediate operation. For our purposes, the only use of `foreach()` is to print out the elements of a stream using methods such as `System.out.print()` and `System.out.println()`, as in the following example.

```
Stream<String> stream = Stream.of("apple", "banana", "bagel");
stream.forEach(word -> System.out.println(word));
```

This snippet produces the output

```
apple
banana
bagel
```

We have already discussed how it is always possible to rewrite a lambda expression as a named instance of a functional interface. There is yet another way to rewrite certain simple lambda expressions. If the lambda expression does nothing except invoke a method that already has a name, you can refer directly to the method using Java's *method reference* operator, "`::`". For example, the snippet `System.out::println` is equivalent to the lambda expression `x -> System.out.println(x)`. Method references are often used with the `foreach()` operation, as in the following example (which is equivalent to the previous example).

```
Stream<String> stream5 = Stream.of("apple", "banana", "bagel");
stream5.forEach(System.out::println);
```

Example problem 10.

Write a snippet of code that uses the Stream API to print the integers from 27 to 159 inclusive on separate lines.

3.2.5 map()

The `map()` operation is used to apply a method to each element of the input stream. In mathematical notation, the result of mapping the function f onto the sequence $x_1, x_2, x_3, x_4, \ldots$ is $f(x_1), f(x_2), f(x_3), f(x_4), \ldots$. The following is an example using Java streams.

```
Stream<String> stream = Stream.of("apple", "banana", "bagel");
stream
      .map(word -> word.toUpperCase() + "***")
      .forEach(System.out::print);
```

The output is

```
            APPLE***BANANA***BAGEL***
```

Example problem 11.

Suppose a file `info.txt` stores a nonempty string on each line. Write a snippet of code that uses the Stream API to print the first character of each line in the file on a separate line.

3.2.6 mapToInt()

The `mapToInt()` operation is identical to `map()`, except that the mapped function must return an `int`, and therefore the resulting stream is an `IntStream`. For example, we can find the location of the letter "e" in each element of a `Stream<String>` as follows.

```
Stream<String> stream = Stream.of("apple", "banana", "bagel");
stream
      .mapToInt(word -> word.indexOf('e'))
      .forEach(System.out::println);
```

The output is

```
                  4
                  -1
                  3
```

Example problem 12.

Write a snippet of code that uses the Stream API to print the length each line in the file `info.txt` on a separate line.

3.2.7 reduce()

The `reduce()` operation is a terminal operation, which is used to combine all the elements of a stream into a single output. Perhaps this operation would be easier to understand if it were called "combine." But the term "reduce" also makes sense, because we are "reducing" an entire stream into a single output.

The reduce() operation is a little more elaborate than the others, so we introduce it by way of an example. For the moment, we abandon streams and go back to processing data in an ordinary array. Suppose we have an array of strings and we would like to take the first character from each string and combine these into a single string. For example, the array {"apple", "banana", "bagel"} will produce the output “abb”. This can be achieved by the following Java code.

```
      String[] array = { "apple", "banana", "bagel" };
      String initialValue = "";
      String resultSoFar = initialValue;
      for (String newElement: array) {
            resultSoFar = resultSoFar + newElement.charAt(0);
      }
```

The idea behind this algorithm is obvious. We move through the array accumulating any results in the variable resultSoFar. Each time we process a new element, we combine it with the existing results (by adding the first character of newElement, in this particular case), producing an updated value for resultSoFar. We also specify an initial value for the results, which is stored separately for clarity in the variable initialValue. When the algorithm terminates, the final result can be found in the accumulator variable resultSoFar.

We can refactor the above snippet so that the process of accumulating results is factored out into the separate method accumulate(), as follows.

```
public static String accumulate(String resultSoFar, String newElement) {
      return resultSoFar + newElement.charAt(0);
}
public static void main(String[] args) throws IOException {
      String[] array2 = { "apple", "banana", "bagel" };
      String initialValue = "";
      String resultSoFar = initialValue;
      for (String newElement : array2) {
            resultSoFar = accumulate(resultSoFar, newElement);
      }
      System.out.println(resultSoFar);
}
```

Now we can describe the reduce() streaming operation using the vocabulary from the above example. The operation has two parameters, which correspond to the initialValue variable and the accumulate() method. Thus, the reduce(initialValue, accumulate) streaming operation applies the accumulate method successively to each element of the stream, using the given initial value for initialization. The accumulate method is usually specified by a lambda expression.

Returning to our concrete example, the following code shows how to produce a string consisting of the initial characters of each element in a stream.

```
Stream<String> stream = Stream.of("apple", "banana", "bagel");
String firstLetters =
      stream.reduce(
            "", // first parameter is the initial value, an empty String
            (resultSoFar, newElement) -> resultSoFar + newElement.charAt(0)
            // second parameter (above) is the ‘accumulate’ function,
            // written as a lambda expression with two parameters
      );
```

Note how the lambda expression representing the accumulator function has two parameters, just like the `accumulate()` method in the earlier example above:

```
(resultSoFar, newElement) -> resultSoFar + newElement.charAt(0)
```

Note that we have studied the *two-parameter* form of the `reduce()` operation. There is also a one-parameter form and a three-parameter form, but we do not study those here.

Example problem 13.

Suppose a file `numbers.txt` stores a number written in decimal notation on each line. Write a snippet of code that uses the Stream API to compute the sum of the numbers in the file using the `java.math.BigDecimal` class, which guarantees that no precision will be lost when dealing with decimal numbers. (It would be important to use this approach when performing a financial calculation, for example.)

3.3 Parallel streams

One advantage of the Java Stream API is that computations can be parallelized with essentially no effort. Every stream has an internal setting that determines whether it operates in a sequential fashion or a parallel fashion. If a stream is set to operate in parallel, the Java stream library will attempt to split the stream into chunks which are fed into separate threads running simultaneously. Certain computations benefit greatly from this parallelism and will complete more quickly. Indeed, if there are N CPU cores available then the computation could in principle be sped up by a factor of N (or even more if we take hyperthreading into account). On the other hand, some computations cannot benefit from parallelization. When operating in parallel mode, such a computation may in fact run more slowly than its sequential version, due to the overhead of setting up the parallel streams.

Any stream can be converted to a parallel stream by applying the `parallel()` operation to it. A stream can be converted to sequential mode by applying the `sequential()` operation. The following is an example of code that benefits greatly from employing parallelism when the array `values[]` is very large. We assume a Boolean method `isPrime()` is available, which returns `true` if its argument is a prime number.

```
int[] values = …
numPrimes = IntStream.of(values)
                    .parallel()
                    .filter(n -> isPrime(n))
                    .count();
```

To perform the same computation sequentially, replace the "`parallel()`" with "`sequential()`". The program `ParallelPrimesDemo.java` (see listing below) compares the two approaches empirically.

Example problem 14.

Run the program `ParallelPrimesDemo.java` (see listing below) on your own computer and determine the average speed-up factor when the computation is done in parallel. How does this compare to the number of CPU cores and/or hyperthreads available on your computer?

3.4 Program listing of ParallelPrimesDemo.java

```
import java.util.Random;
import java.util.stream.IntStream;

/**
 * This class provides a demonstration of the speed-up that can be obtained
 * by using parallel streams rather than sequential streams.
*/
public class ParallelPrimesDemo {

	// A deliberately inefficient way of determining whether an integer
	// n>=2 is prime. We just want to perform a computationally intensive
	// task to demonstrate the benefits of parallelism.
	private static boolean isPrime(int n) {
		for (int i = 2; i < n; i++) {
			if (n % i == 0) {
				return false;
			}
		}
		return true;
	}

	public static void main(String[] args) {
		// Step 1. Create a large array of random integers. We ensure that
		// each integer is greater than or equal to 2, so that it makes
		// sense to ask whether the integer is prime.
		Random random = new Random();
		final int numValues = 2000000;
		final int maxValue = 10000;
		int[] values = new int[numValues];
		for (int i = 0; i < values.length; i++) {
			values[i] = 2 + random.nextInt(maxValue - 2);
		}
```

```
            // We repeat the remaining steps several times so that we can check
            // if the timing results are reliable.
            int numRepetitions = 5;
            for (int repetition = 0; repetition < numRepetitions; repetition++) {

                  // Step 2. Use a sequential stream to determine how many of the
                  // random integers are prime, and record the time taken.
                  long numPrimes1, numPrimes2;
                  long startTime, endTime;
                  startTime = System.nanoTime();
                  numPrimes1 = IntStream.of(values).sequential()
                               .filter(n -> isPrime(n)).count();
                  endTime = System.nanoTime();
                  long sequentialDuration = (endTime - startTime)
                    / 1000000; // milliseconds

                  // Step 3. The same as the previous step, but with a parallel
                  // stream.
                  startTime = System.nanoTime();
                  numPrimes2 = IntStream.of(values).parallel()
                               .filter(n -> isPrime(n)).count();
                  endTime = System.nanoTime();
                  long parallelDuration = (endTime - startTime)
                    / 1000000; // milliseconds

                  // The answers had better be the same!
                  assert numPrimes1 == numPrimes2;

                  // Step 4. Print the timing results.
                  double speedup = (double) sequentialDuration
                               / parallelDuration;
                  System.out.format(
                  "sequential %dms, parallel %dms, speedup factor %2.2f\n",
                        sequentialDuration, parallelDuration, speedup);
            }
      }
}
```

4 Solutions to example problems

Example problem 1.

```
applyToMinusNine(add5) → -4
applyToMinusNine(multBy3IfPositive) → 0
```

Example problem 2.

```
applyToMinusNine(add5) → -4
applyToMinusNine(multBy3IfPositive) → 0
```

Example problem 3.

```
applyToSeven(lambda potato: potato%4 + potato*potato) → 52
applyToMinusNine(lambda oak: math.factorial(oak+12) * oak) → -54
```

Example problem 4.

```
val5 = 56
val6 = -90
```

Example problem 5.

```
(u,v,w)->Math.sqrt(u*u + v*v + w*w)
```

Example problem 6.

There are many ways to achieve this. Here are some examples:

(i) `Stream<Double> doubleObjects = Stream.of(23.4, 69.7, -25.88, 31.3363);`

(ii) `DoubleStream doublePrimitives = DoubleStream.of(23.4, 69.7, -25.88, 31.3363);`

(iii) Either of the following approaches would work here:

```
IntStream range1 = IntStream.range(100, 201);
IntStream range2 = IntStream.rangeClosed(100, 200);
```

Example problem 7.

```
Stream<String> gatsby = Files.lines(Paths.get("GreatGatsby.txt"));
long numLines = gatsby.count();
```

Example problem 8.

```
int sum27to159 = IntStream.rangeClosed(27, 159).sum();
```

Example problem 9.

```
int sumOdd27to159 = IntStream.rangeClosed(27, 159)
                        .filter(x -> x % 2 == 1)
                        .sum();
```

Example problem 10.

```
IntStream.rangeClosed(27, 159).forEach(System.out::println);
```

Example problem 11.

```
Stream<String> info = Files.lines(Paths.get("info.txt"));
info.map(s->s.substring(0, 1)).forEach(System.out::println);
```

Example problem 12.

```
Stream<String> info = Files.lines(Paths.get("info.txt"));
info.mapToInt(s->s.length()).forEach(System.out::println);
```

Example problem 13.

```
Stream<String> numbers = Files.lines(Paths.get("numbers.txt"));
BigDecimal sum = numbers
            .map(s -> new BigDecimal(s))
            .reduce(new BigDecimal(0), (tot, val) -> tot.add(val));
System.out.println(sum);
```

Example problem 14.

The following results were obtained on an Intel Core i5-8350U Processor:

```
      sequential 2907ms, parallel 634ms, speedup factor 4.59
      sequential 2790ms, parallel 639ms, speedup factor 4.37
      sequential 2714ms, parallel 651ms, speedup factor 4.17
      sequential 2729ms, parallel 651ms, speedup factor 4.19
      sequential 2731ms, parallel 646ms, speedup factor 4.23
```

This processor is described by Intel as having 4 cores and 8 threads. One plausible interpretation of the above speedup factors is that each of the four cores was able to perform part of the processing simultaneously, and a small additional boost was provided by the hyperthreading technology on this processor.

Chapter 9: Graphs

09.01 Graphs Chapter Introduction

Due No Due Date **Points** 2 **Submitting** an external tool

09.01 Graphs Chapter Introduction

9.1. Graphs Chapter Introduction

9.1.1. Graph Terminology and Implementation

Graphs provide the ultimate in data structure flexibility. A graph consists of a set of nodes, and a set of edges where an edge connects two nodes. Trees and lists can be viewed as special cases of graphs.

Graphs are used to model both real-world systems and abstract problems, and are the data structure of choice in many applications. Here is a small sampling of the types of problems that graphs are routinely used for.

1. Modeling connectivity in computer and communications networks.
2. Representing an abstract map as a set of locations with distances between locations. This can be used to compute shortest routes between locations such as in a GPS routefinder.
3. Modeling flow capacities in transportation networks to find which links create the bottlenecks.
4. Finding a path from a starting condition to a goal condition. This is a common way to model problems in artificial intelligence applications and computerized game players.
5. Modeling computer algorithms, to show transitions from one program state to another.
6. Finding an acceptable order for finishing subtasks in a complex activity, such as constructing large buildings.
7. Modeling relationships such as family trees, business or military organizations, and scientific taxonomies.

The rest of this module covers some basic graph terminology. The following modules will describe fundamental representations for graphs, provide a reference implementation, and cover core graph algorithms including traversal, topological sort, shortest paths algorithms, and algorithms to find the minimal-cost spanning tree. Besides being useful and interesting in their own right, these algorithms illustrate the use of many other data structures presented throughout the course.

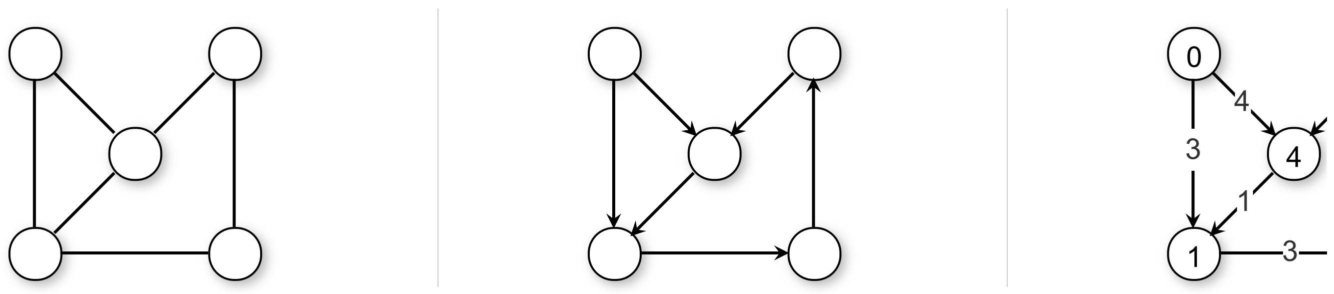
A **graph** $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ consists of a set of **vertices** $\mathbf{V}$ and a set of **edges** $\mathbf{E}$, such that each edge in $\mathbf{E}$ is a connection between a pair of vertices in $\mathbf{V}$. [1] The number of vertices is written $|\mathbf{V}|$, and the number of edges is written $|\mathbf{E}|$. $|\mathbf{E}|$ can range from zero to a maximum of $|\mathbf{V}|^2 - |\mathbf{V}|$.

[1]

Some graph applications require that a given pair of vertices can have multiple or parallel edges connecting

Some graph applications require that a given pair of vertices can have multiple or parallel edges connecting them, or that a vertex can have an edge to itself. However, the applications discussed here do not require either of these special cases. To simplify our graph API, we will assume that there are no dupicate edges, and no edges that connect a node to itself.

A graph whose edges are not directed is called an **undirected graph**, as shown in part (a) of the following figure. A graph with edges directed from one vertex to another (as in (b)) is called a **directed graph** or **digraph**. A graph with labels associated with its vertices (as in (c)) is called a **labeled graph**. Associated with each edge may be a cost or **weight**. A graph whose edges have weights (as in (c)) is said to be a **weighted graph**.

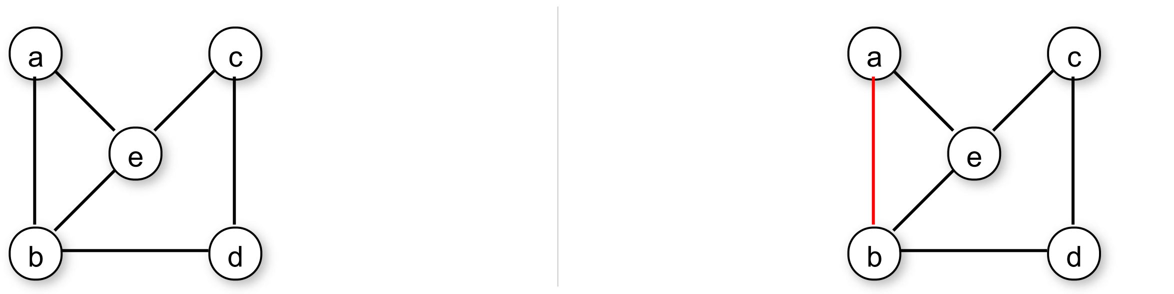


(a) undirected graph (b) directed graph (c) labeled

Figure 9.1.1: Some types of graphs.

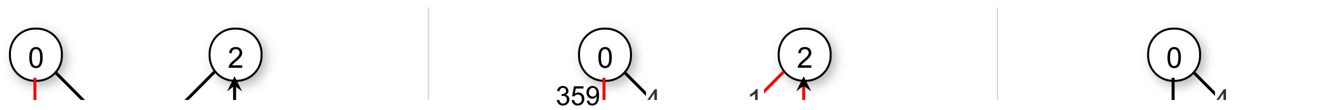
An edge connecting Vertices a and b is written (a, b). Such an edge is said to be **incident** with Vertices a and b. The two vertices are said to be **adjacent**. If the edge is directed from a to b, then we say that a is adjacent to b, and b is adjacent from a. The **degree** of a vertex is the number of edges it is incident with. For example, Vertex e below has a degree of three.

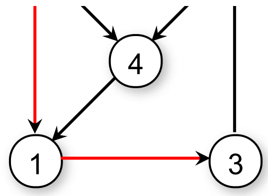
In a directed graph, the **out degree** for a vertex is the number of neighbors adjacent from it (or the number of edges going out from it), while the **in degree** is the number of neighbors adjacent to it (or the number of edges coming in to it). In (c) above, the in degree of Vertex 1 is two, and its out degree is one.

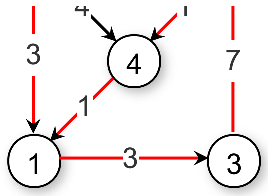


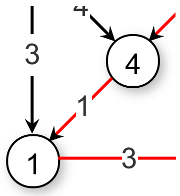
(a) Vertices a and b are *neighbors* (b) The red edge is *incident* with vertices a

A sequence of vertices $v_1, v_2, \ldots, v_n$ forms a **path** of length $n-1$ if there exist edges from v_i to v_{i+1} for $1 \leq i < n$. A path is a **simple path** if all vertices on the path are distinct. The **length** of a path is the number of edges it contains. A **cycle** is a path of length three or more that connects some vertex v_1 to itself. A cycle is a **simple cycle** if the path is simple, except for the first and last vertices being the same.

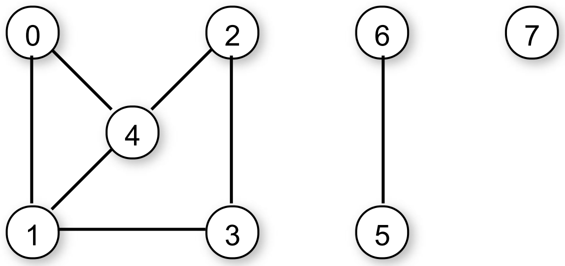



(a) A *simple* path from 0 to 3.

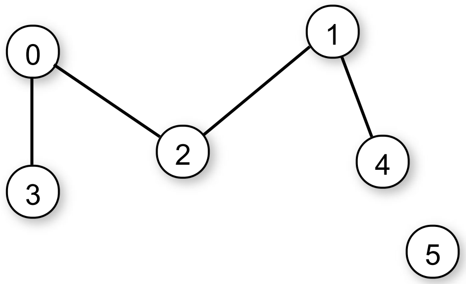

(b) Path 0, 1, 3, 2, 4, 1 is not simple

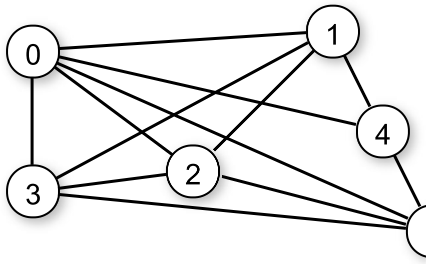

(c) *Simple cycle* 1

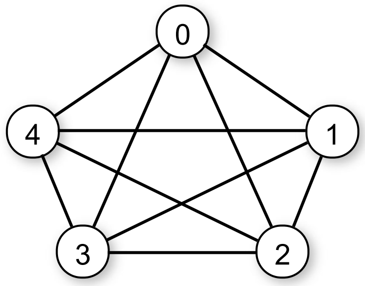
An undirected graph is a **connected graph** if there is at least one path from any vertex to any other. The maximally connected subgraphs of an undirected graph are called **connected components**. For example, this figure shows an undirected graph with three connected components.

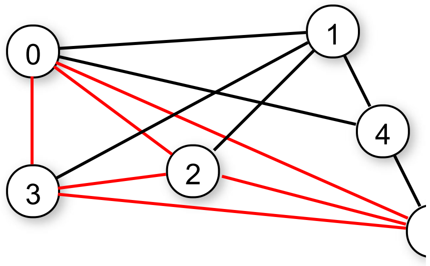


A graph with relatively few edges is called a **sparse graph**, while a graph with many edges is called a **dense graph**. A graph containing all possible edges is said to be a **complete graph**. A **subgraph** $\mathbf{S}$ is formed from graph $\mathbf{G}$ by selecting a subset $\mathbf{V}_s$ of $\mathbf{G}$'s vertices and a subset $\mathbf{E}_s$ of $\mathbf{G}$'s edges such that for every edge $e \in \mathbf{E}_s$, both vertices of e are in $\mathbf{V}_s$. Any subgraph of V where all vertices in the graph connect to all other vertices in the subgraph is called a **clique**.

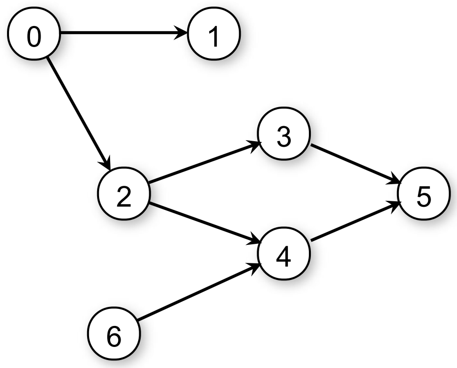

(a) A graph with relatively few edges is called a *sparse graph*.


(b) A graph with many edges is called a *den*

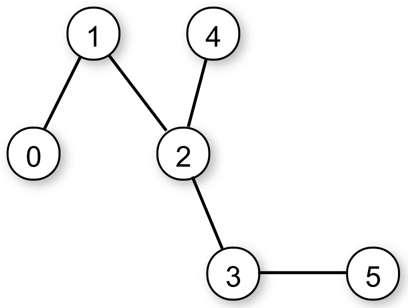

(c) A *complete graph* has edges connecting every pair of nodes.


(d) A *clique* is a subset of V where all vertice have edges to all other vertices in the subset

A graph without cycles is called an **acyclic graph**. Thus, a directed graph without cycles is called a **directed acyclic graph** or **DAG**.



(a) Directed Acyclic Graph



(b) Acyclic Graph

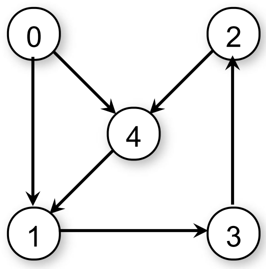
A **free tree** is a connected, undirected graph with no simple cycles. An equivalent definition is that a free tree is connected and has $|\mathbf{V}| - 1$ edges.

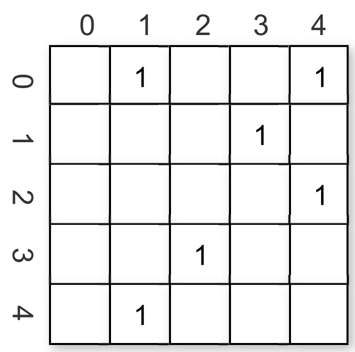
9.1.1.1. Graph Representations

There are two commonly used methods for representing graphs. The **adjacency matrix** for a graph is a $|\mathbf{V}| \times |\mathbf{V}|$ array. We typically label the vertices from v_0 through $v_{|\mathbf{V}|-1}$. Row i of the adjacency matrix contains entries for Vertex v_i. Column j in row i is marked if there is an edge from v_i to v_j and is not marked otherwise. The space requirements for the adjacency matrix are $\Theta(|\mathbf{V}|^2)$.

The second common representation for graphs is the **adjacency list**. The adjacency list is an array of linked lists. The array is $|\mathbf{V}|$ items long, with position i storing a pointer to the linked list of edges for Vertex v_i. This linked list represents the edges by the vertices that are adjacent to Vertex v_i.

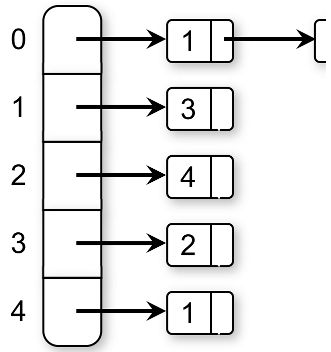
Here is an example of the two representations on a directed graph. The entry for Vertex 0 stores 1 and 4 because there are two edges in the graph leaving Vertex 0, with one going to Vertex 1 and one going to Vertex 4. The list for Vertex 2 stores an entry for Vertex 4 because there is an edge from Vertex 2 to Vertex 4, but no entry for Vertex 3 because this edge comes into Vertex 2 rather than going out.





	0	1	2	3	4
0		1			1
1				1	
2					1
3			1		
4		1			

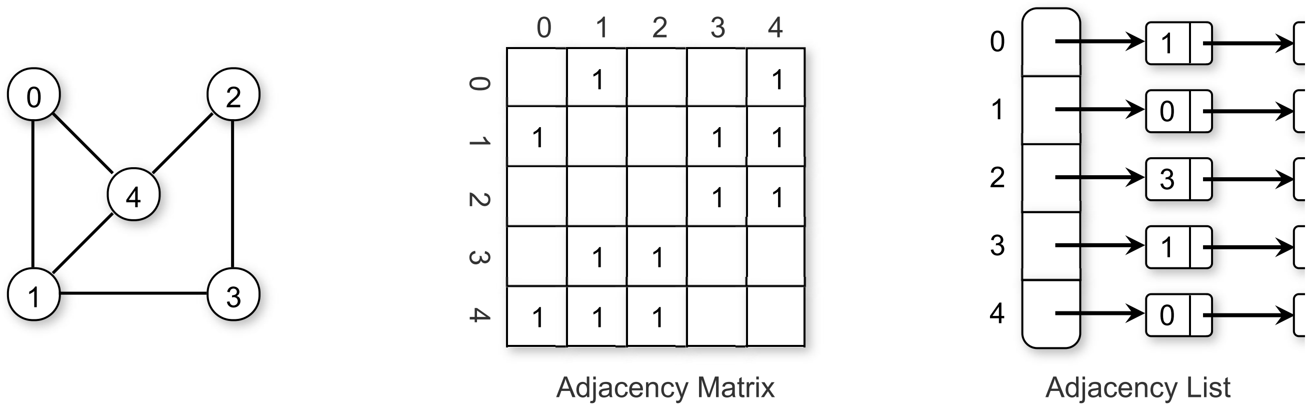
Adjacency Matrix



Adjacency List

Figure 9.1.7: Representing a directed graph.

Both the adjacency matrix and the adjacency list can be used to store directed or undirected graphs. Each edge of an undirected graph connecting Vertices u and v is represented by two directed edges: one from u to v and one from v to u. Here is an example of the two representations on an undirected graph. We see that there are twice as many edge entries in both the adjacency matrix and the adjacency list. For example, for the undirected graph, the list for Vertex 2 stores an entry for both Vertex 3 and Vertex 4.



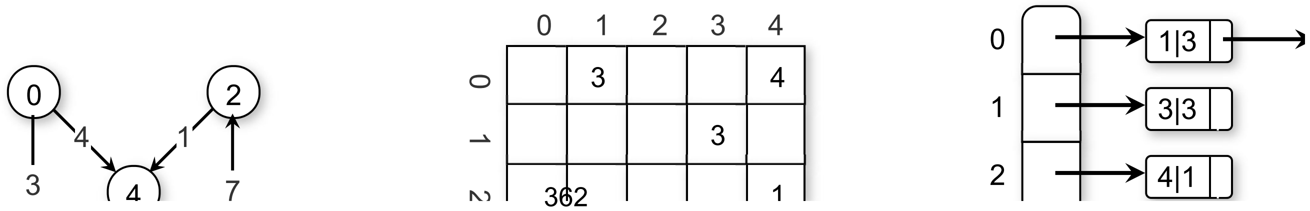
	0	1	2	3	4
0		1			1
1	1			1	1
2				1	1
3		1	1		
4	1	1	1		

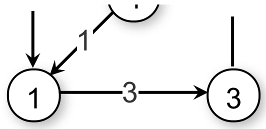
Figure 9.1.8: Representing an undirected graph.

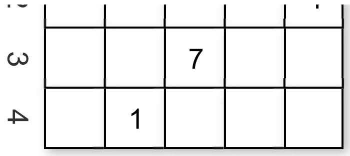
The storage requirements for the adjacency list depend on both the number of edges and the number of vertices in the graph. There must be an array entry for each vertex (even if the vertex is not adjacent to any other vertex and thus has no elements on its linked list), and each edge must appear on one of the lists. Thus, the cost is $\Theta(|\mathbf{V}| + |\mathbf{E}|)$.

Sometimes we want to store weights or distances with each each edge, such as in Figure 9.1.1 (c). This is easy with the adjacency matrix, where we will just store values for the weights in the matrix. In Figures 9.1.7 and 9.1.8 we store a value of "1" at each position just to show that the edge exists. That could have been done using a single bit, but since bit manipulation is typically complicated in most programming languages, an implementation might store a byte or an integer at each matrix position. For a weighted graph, we would need to store at each position in the matrix enough space to represent the weight, which might typically be an integer.

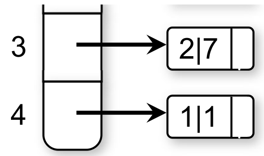
The adjacency list needs to explicitly store a weight with each edge. In the adjacency list shown below, each linked list node is shown storing two values. The first is the index for the neighbor at the end of the associated edge. The second is the value for the weight. As with the adjacency matrix, this value requires space to represent, typically an integer.







Adjacency Matrix: Weights



Adjacency List: Weigh

Which graph representation is more space efficient depends on the number of edges in the graph. The adjacency list stores information only for those edges that actually appear in the graph, while the adjacency matrix requires space for each potential edge, whether it exists or not. However, the adjacency matrix requires no overhead for pointers, which can be a substantial cost, especially if the only information stored for an edge is one bit to indicate its existence. As the graph becomes denser, the adjacency matrix becomes relatively more space efficient. Sparse graphs are likely to have their adjacency list representation be more space efficient.

Example 9.1.1

Assume that a vertex index requires two bytes, a pointer requires four bytes, and an edge weight requires two bytes. Then, each link node in the adjacency list needs $2+2+4=8$ bytes. The adjacency matrix for the directed graph above requires $2|\mathbf{V}^2|=50$ bytes while the adjacency list requires $4|\mathbf{V}|+8|\mathbf{E}|=68$ bytes. For the undirected version of the graph above, the adjacency matrix requires the same space as before, while the adjacency list requires $4|\mathbf{V}|+8|\mathbf{E}|=116$ bytes (because there are now 12 edges represented instead of 6).

The adjacency matrix often requires a higher asymptotic cost for an algorithm than would result if the adjacency list were used. The reason is that it is common for a graph algorithm to visit each neighbor of each vertex. Using the adjacency list, only the actual edges connecting a vertex to its neighbors are examined. However, the adjacency matrix must look at each of its $|\mathbf{V}|$ potential edges, yielding a total cost of $\Theta(|\mathbf{V}^2|)$ time when the algorithm might otherwise require only $\Theta(|\mathbf{V}| + |\mathbf{E}|)$ time. This is a considerable disadvantage when the graph is sparse, but not when the graph is closer to full.

9.1.2. Graph Terminology Questions

9.2. Graph Implementations

We next turn to the problem of implementing a general-purpose **graph** class. There are two traditional approaches to representing graphs: The **adjacency matrix** and the **adjacency list**. In this module we will show actual implementations for each approach. We will begin with an interface defining an ADT for graphs that a given implementation must meet.

Toggle Tree View

```
interface Graph { // Graph class ADT
  // Initialize the graph with some number of vertices
  void init(int n);

  // Return the number of vertices
  int nodeCount();

  // Return the current number of edges
  int edgeCount();

  // Get the value of node with index v
  Object getValue(int v);

  // Set the value of node with index v
  void setValue(int v, Object val);

  // Adds a new edge from node v to node w with weight wgt
  void addEdge(int v, int w, int wgt);

  // Get the weight value for an edge
  int weight(int v, int w);

  // Removes the edge from the graph.
  void removeEdge(int v, int w);

  // Returns true iff the graph has the edge
  boolean hasEdge(int v, int w);

  // Returns an array containing the indicies of the neighbors of v
  int[] neighbors(int v);
}
```

This ADT assumes that the number of vertices is fixed when the graph is created, but that edges can be added and removed. The `init` method sets (or resets) the number of nodes in the graph, and creates necessary space for the adjacency matrix or adjacency list.

Vertices are defined by an integer index value. In other words, there is a Vertex 0, Vertex 1, and so on through Vertex $n-1$. We can assume that the graph's client application stores any additional information of interest about a given vertex elsewhere, such as a name or application-dependent value. Note that in a language like Java or C++, this ADT would not be implemented using a language ture like a generic or template, because it is the `Graph` this ADT would not be implemented using a language feature like a generic or template, because it is the Graph class users' responsibility to maintain information related to the vertices themselves. The Graph class need have no knowledge of the type or content of the information associated with a vertex, only the index number for that vertex.

Interface Graph has methods to return the number of vertices and edges (methods n and e, respectively). Function weight returns the weight of a given edge, with that edge identified by its two incident vertices. For example, calling weight(0, 4) on the graph of Figure 9.1.1 (c) would return 4. If no such edge exists, the weight is defined to be 0. So calling weight(0, 2) on the graph of Figure 9.1.1 (c) would return 0.

Functions addEdge and removeEdge add an edge (setting its weight) and removes an edge from the graph, respectively. Again, an edge is identified by its two incident vertices. addEdge does not permit the user to set the weight to be 0, because this value is used to indicate a non-existent edge, nor are negative edge weights permitted. Functions getValue and setValue get and set, respectively, a requested value for Vertex v. In our example applications the most frequent use of these methods will be to indicate whether a given node has previously been visited in the process of the algorithm

Nearly every graph algorithm presented in this chapter will require visits to all neighbors of a given vertex. The neighbors method returns an array containing the indices for the neighboring vertices, in ascending order. The following lines appear in many graph algorithms.

Toggle Tree View

```
  int[] nList = G.neighbors(v);
  for (int i=0; i< nList.length; i++) {
    if (G.getValue(nList[i]) != VISITED) {
      DoSomething();
    }
  }
```

First, an array is generated that contains the indices of the nodes that can be directly reached from node v. The for loop then iterates through this neighbor array to execute some function on each.

It is reasonably straightforward to implement our graph ADT using either the adjacency list or adjacency matrix. The sample implementations presented here do not address the issue of how the graph is actually created. The user of these implementations must add functionality for this purpose, perhaps reading the graph description from a file. The graph can be built up by using the addEdge function provided by the ADT.

Here is an implementation for the adjacency matrix.

Toggle Tree View

```
class GraphM implements Graph {
  private int[][] matrix;
  private Object[] nodeValues;
  private int numEdge;

  // No real constructor needed
  GraphM() { }

  // Initialize the graph with n vertices
```

```
public void init(int n) {
  matrix = new int[n][n];
  nodeValues = new Object[n];
  numEdge = 0;
}

// Return the number of vertices
public int nodeCount() { return nodeValues.length; }

// Return the current number of edges
public int edgeCount() { return numEdge; }

// Get the value of node with index v
public Object getValue(int v) { return nodeValues[v]; }

// Set the value of node with index v
public void setValue(int v, Object val) { nodeValues[v] = val; }

// Adds a new edge from node v to node w
// Returns the new edge
public void addEdge(int v, int w, int wgt) {
  if (wgt == 0) { return; } // Can't store weight of 0
  if (matrix[v][w] == 0) {
    numEdge++;
  }
  matrix[v][w] = wgt;
}

// Get the weight value for an edge
public int weight(int v, int w) { return matrix[v][w]; }

// Removes the edge from the graph.
public void removeEdge(int v, int w) {
  if (matrix[v][w] != 0) {
    matrix[v][w] = 0;
    numEdge--;
  }
}

// Returns true iff the graph has the edge
public boolean hasEdge(int v, int w) { return matrix[v][w] != 0; }

// Returns an array containing the indicies of the neighbors of v
public int[] neighbors(int v) {
  int i;
  int count = 0;
  int[] temp;

  for (i=0; i<nodeValues.length; i++) {
    if (matrix[v][i] != 0) { count++; }
  }
  temp = new int[count];
  for (i=0, count=0; i<nodeValues.length; i++) {
    if (matrix[v][i] != 0) { temp[count++] = i; }
    }
    return temp;
  }
}
```

Array nodeValues stores the information manipulated by the setValue and getValue functions. The edge matrix is implemented as an integer array of size $n \times n$ for a graph of n vertices. Position (i, j) in the matrix stores the weight for edge (i, j) if it exists. A weight of zero for edge (i, j) is used to indicate that no edge connects Vertices i and j.

Given a vertex v, the neighbors method scans through row v of the matix to locate the positions of the various neighbors. If no edge is incident on v, then returned neighbor array will have length 0. Functions addEdge and removeEdge adjust the appropriate value in the array. Function weight returns the value stored in the appropriate position in the array.

Here is an implementation of the adjacency list representation for graphs. Its main data structure is an array of linked lists, one linked list for each vertex. These linked lists store objects of type Edge, which merely stores the index for the vertex pointed to by the edge, along with the weight of the edge.

Toggle Tree View

```
public class GraphL implements Graph {

  private class Edge { // Doubly linked list node
    int vertex, weight;
    Edge prev, next;

    Edge(int v, int w, Edge p, Edge n) {
      vertex = v;
      weight = w;
      prev = p;
      next = n;
    }
  }

  private Edge[] nodeArray;
  private Object[] nodeValues;
  private int numEdge;

  // No real constructor needed
  GraphL() {}

  // Initialize the graph with n vertices
  public void init(int n) {
    nodeArray = new Edge[n];
    // List headers;
    for (int i=0; i<n; i++) { nodeArray[i] = new Edge(-1, -1, null, null); }
    nodeValues = new Object[n];
    numEdge = 0;
  }

  // Return the number of vertices
  public int nodeCount() { return nodeArray.length; }
```

```
// Return the current number of edges
public int edgeCount() { return numEdge; }

// Get the value of node with index v
public Object getValue(int v) { return nodeValues[v]; }

// Set the value of node with index v
public void setValue(int v, Object val) { nodeValues[v] = val; }

// Return the link in v's neighbor list that preceeds the
// one with w (or where it would be)
private Edge find (int v, int w) {
  Edge curr = nodeArray[v];
  while ((curr.next != null) && (curr.next.vertex < w)) {
    curr = curr.next;
  }
  return curr;
}

// Adds a new edge from node v to node w with weight wgt
public void addEdge(int v, int w, int wgt) {
  if (wgt == 0) { return; } // Can't store weight of 0
  Edge curr = find(v, w);
  if ((curr.next != null) && (curr.next.vertex == w)) {
    curr.next.weight = wgt;
  }
  else {
    curr.next = new Edge(w, wgt, curr, curr.next);
    if (curr.next.next != null) { curr.next.next.prev = curr.next; }
  }
  numEdge++;
}

// Get the weight value for an edge
public int weight(int v, int w) {
  Edge curr = find(v, w);
  if ((curr.next == null) || (curr.next.vertex != w)) { return 0; }
  else { return curr.next.weight; }
}

// Removes the edge from the graph.
public void removeEdge(int v, int w) {
  Edge curr = find(v, w);
  if ((curr.next == null) || curr.next.vertex != w) { return; }
  else {
    curr.next = curr.next.next;
    if (curr.next != null) { curr.next.prev = curr; }
  }
  numEdge--;
}

// Returns true iff the graph has the edge
public boolean hasEdge(int v, int w) { return weight(v, w) != 0; }
```

```
  // Returns an array containing the indicies of the neighbors of v
  public int[] neighbors(int v) {
    int cnt = 0;
    Edge curr;
    for (curr = nodeArray[v].next; curr != null; curr = curr.next) {
      cnt++;
    }
    int[] temp = new int[cnt];
    cnt = 0;
    for (curr = nodeArray[v].next; curr != null; curr = curr.next) {
      temp[cnt++] = curr.vertex;
    }
    return temp;
  }
}
```

Implementation for `GraphL` member functions is straightforward in principle, with the key functions being `addEdge`, `removeEdge`, and `weight`. They simply start at the beginning of the adjacency list and move along it until the desired vertex has been found. Private method find is a utility for finding the last edge preceding the one that holds vertex v

09.03 Graph Traversals

Due No Due Date **Points** 2 **Submitting** an external tool

09.03 Graph Traversals

9.3. Graph Traversals

9.3.1. Graph Traversals

Many graph applications need to visit the vertices of a graph in some specific order based on the graph's topology. This is known as a graph **traversal** and is similar in concept to a **tree traversal**. Recall that tree traversals visit every node exactly once, in some specified order such as preorder, inorder, or postorder. Multiple tree traversals exist because various applications require the nodes to be visited in a particular order. For example, to print a BST's nodes in ascending order requires an inorder traversal as opposed to some other traversal. Standard graph traversal orders also exist. Each is appropriate for solving certain problems. For example, many problems in artificial intelligence programming are modeled using graphs. The problem domain might consist of a large collection of states, with connections between various pairs of states. Solving this sort of problem requires getting from a specified start state to a specified goal state by moving between states only through the connections. Typically, the start and goal states are not directly connected. To solve this problem, the vertices of the graph must be searched in some organized manner.

Graph traversal algorithms typically begin with a start vertex and attempt to visit the remaining vertices from there. Graph traversals must deal with a number of troublesome cases. First, it might not be possible to reach all vertices from the start vertex. This occurs when the graph is not connected. Second, the graph might contain cycles, and we must make sure that cycles do not cause the algorithm to go into an infinite loop.

Graph traversal algorithms can solve both of these problems by flagging vertices as `VISITED` when appropriate. At the beginning of the algorithm, no vertex is flagged as `VISITED`. The flag for a vertex is set when the vertex is first visited during the traversal. If a flagged vertex is encountered during traversal, it is not visited a second time. This keeps the program from going into an infinite loop when it encounters a cycle.

Once the traversal algorithm completes, we can check to see if all vertices have been processed by checking whether they have the `VISITED` flag set. If not all vertices are flagged, we can continue the traversal from another unvisited vertex. Note that this process works regardless of whether the graph is directed or undirected. To ensure visiting all vertices, `graphTraverse` could be called as follows on a graph $\mathbf{G}$:

Toggle Tree View

```
static void graphTraverse(Graph G) {
  int v;
  for (v=0; v<G.nodeCount(); v++) {
```

```
  for (v=0; v<G.nodeCount(); v++) {
    G.setValue(v, null); // Initialize
  }
  for (v=0; v<G.nodeCount(); v++) {
    if (G.getValue(v) != VISITED) {
      doTraversal(G, v);
    }
  }
}
```

Function `doTraversal` might be implemented by using one of the graph traversals described next.

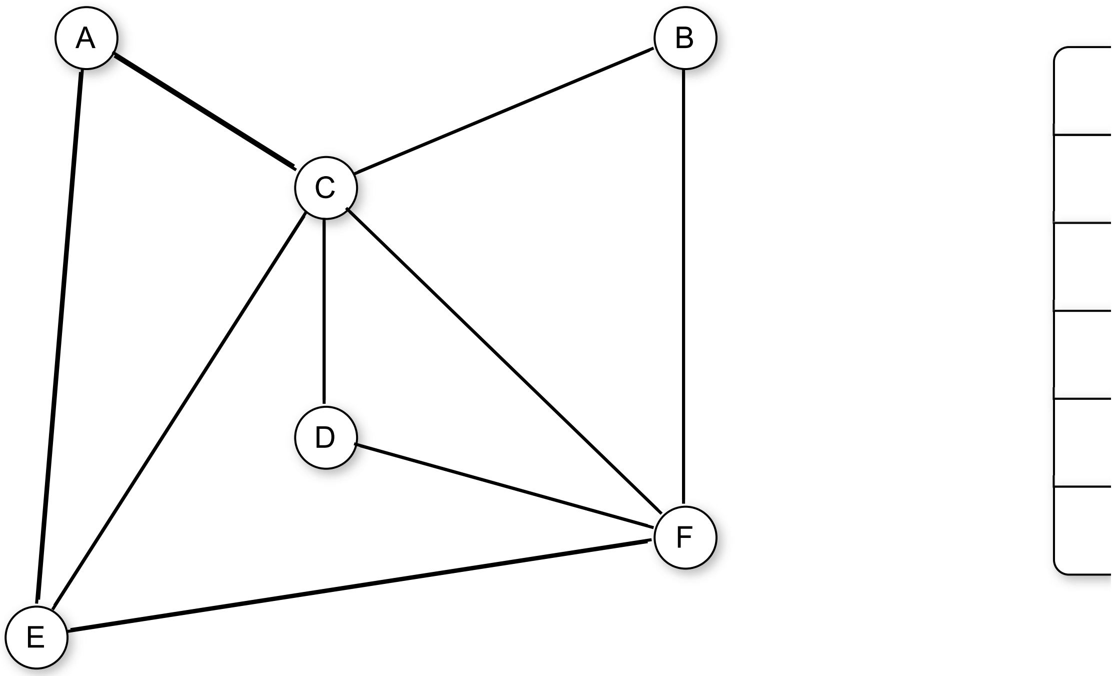
9.3.1.1. Depth-First Search

Our first method for organized graph traversal is called **depth-first search** (DFS). Whenever a vertex v is visited during the search, DFS will recursively visit all of v 's unvisited neighbors. Equivalently, DFS will add all edges leading out of v to a stack. The next vertex to be visited is determined by popping the stack and following that edge. The effect is to follow one branch through the graph to its conclusion, then it will back up and follow another branch, and so on. The DFS process can be used to define a **depth-first search tree**. This tree is composed of the edges that were followed to any new (unvisited) vertex during the traversal, and leaves out the edges that lead to already visited vertices. DFS can be applied to directed or undirected graphs.

This visualization shows a graph and the result of performing a DFS on it, resulting in a depth-first search tree.

1 / 55 << < > >>

Let's look at the details of how a depth-first seach works.



Here is an implementation for the DFS algorithm.

Toggle Tree View

```
static void DFS(Graph G, int v) {
  PreVisit(G, v);
  G.setValue(v, VISITED);
  int[] nList = G.neighbors(v);
  for (int i=0; i< nList.length; i++) {
    if (G.getValue(nList[i]) != VISITED) {
      DFS(G, nList[i]);
    }
  }
  PostVisit(G, v);
}
```

This implementation contains calls to functions `PreVisit` and `PostVisit`. These functions specify what activity should take place during the search. Just as a preorder tree traversal requires action before the subtrees are visited, some graph traversals require that a vertex be processed before ones further along in the DFS. Alternatively, some applications require activity *after* the remaining vertices are processed; hence the call to function `PostVisit`. This would be a natural opportunity to make use of the **visitor** design pattern.

The following visualization shows a random graph each time that you start it, so that you can see the behavior on different examples. It can show you DFS run on a directed graph or an undirected graph. Be sure to look at an example for each type of graph.

Depth-First Search

Undirected Directed

DFS processes each edge once in a directed graph. In an undirected graph, DFS processes each edge from both directions. Each vertex must be visited, but only once, so the total cost is $\Theta(|\mathbf{V}| + |\mathbf{E}|)$.

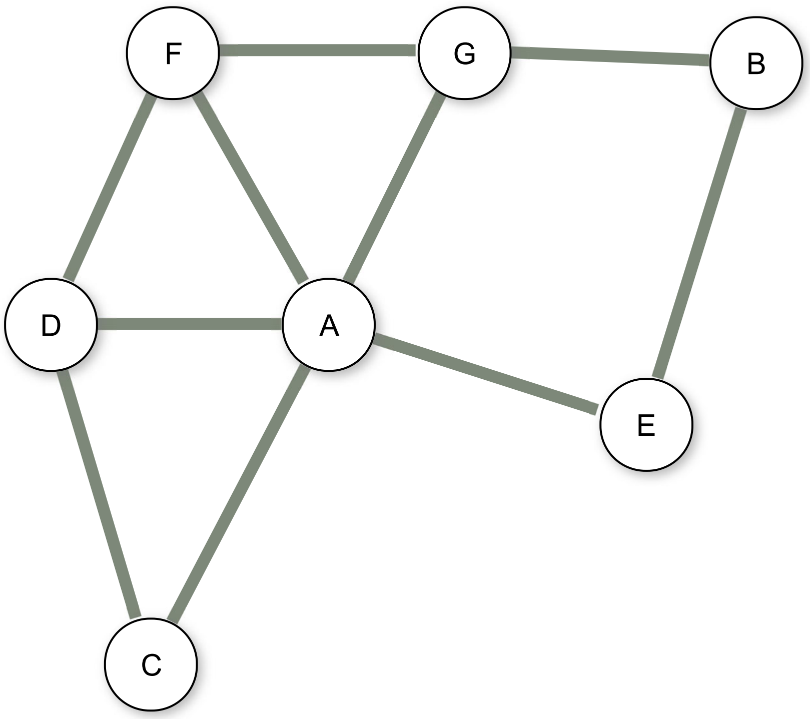
Here is an exercise for you to practice DFS.

Reset Model Answer

Instructions:

Reproduce the behavior of the DFS algorithm for the graph below. Just click on the **edges** in the order that t
will be traversed by the DFS algorithm. Start with Node A. If there is more than one node that could be visite
next, choose the one that comes first in alphabetical order.

Score: 0 / 6, Points remaining: 6, Points lost: 0



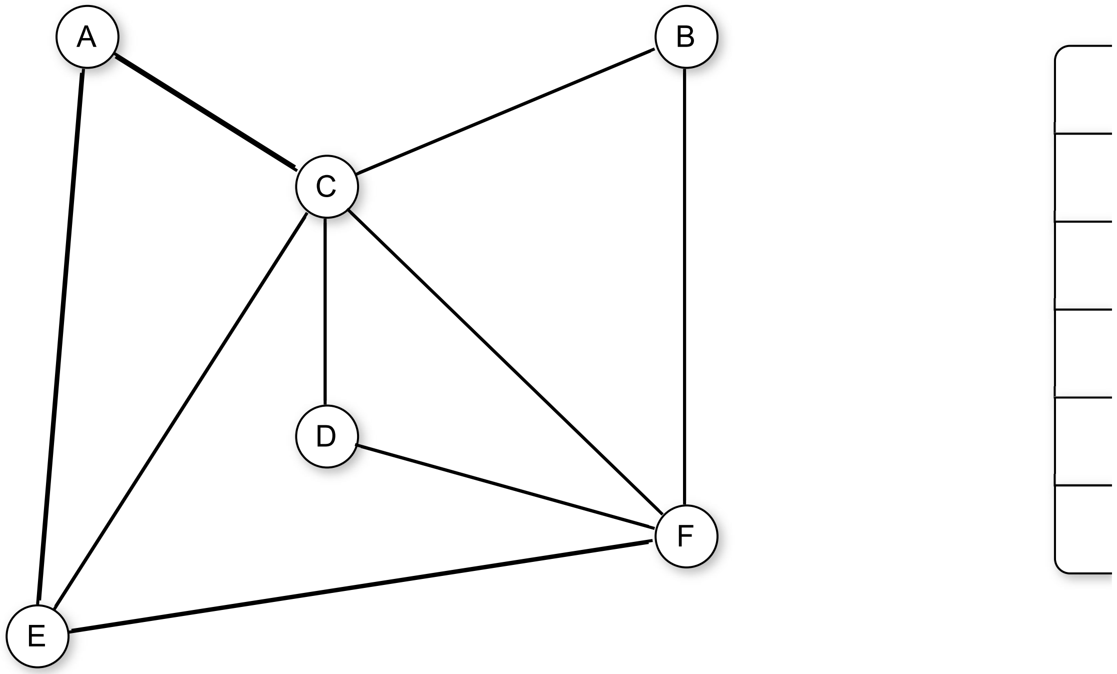
9.3.2. Breadth-First Search

Our second graph traversal algorithm is known as a **breadth-first search** (BFS). BFS examines all vertices connected to the start vertex before visiting vertices further away. BFS is implemented similarly to DFS, except that a queue replaces the recursion stack. Note that if the graph is a tree and the start vertex is at the root, BFS is equivalent to visiting vertices level by level from top to bottom.

This visualization shows a graph and the result of performing a BFS on it, resulting in a breadth-first search tree.

1 / 39

Let's look at the details of how a breadth-first seach works.



Here is an implementation for BFS.

Toggle Tree View

```
static void BFS(Graph G, int v) {
  LQueue Q = new LQueue(G.nodeCount());
  Q.enqueue(v);
  G.setValue(v, VISITED);
  while (Q.length() > 0) { // Process each vertex on Q
    v = (Integer)Q.dequeue();
    PreVisit(G, v);
    int[] nList = G.neighbors(v);
    for (int i=0; i< nList.length; i++) {
      if (G.getValue(nList[i]) != VISITED) { // Put neighbors on Q
        G.setValue(nList[i], VISITED);
        Q.enqueue(nList[i]);
      }
    }
    PostVisit(G, v);
  }
}
```

The following visualization shows a random graph each time that you start it, so that you can see the behavior on different examples. It can show you BFS run on a directed graph or an undirected graph. Be sure to look at an example for each type of graph.

Breadth-First Search

Undirected Directed

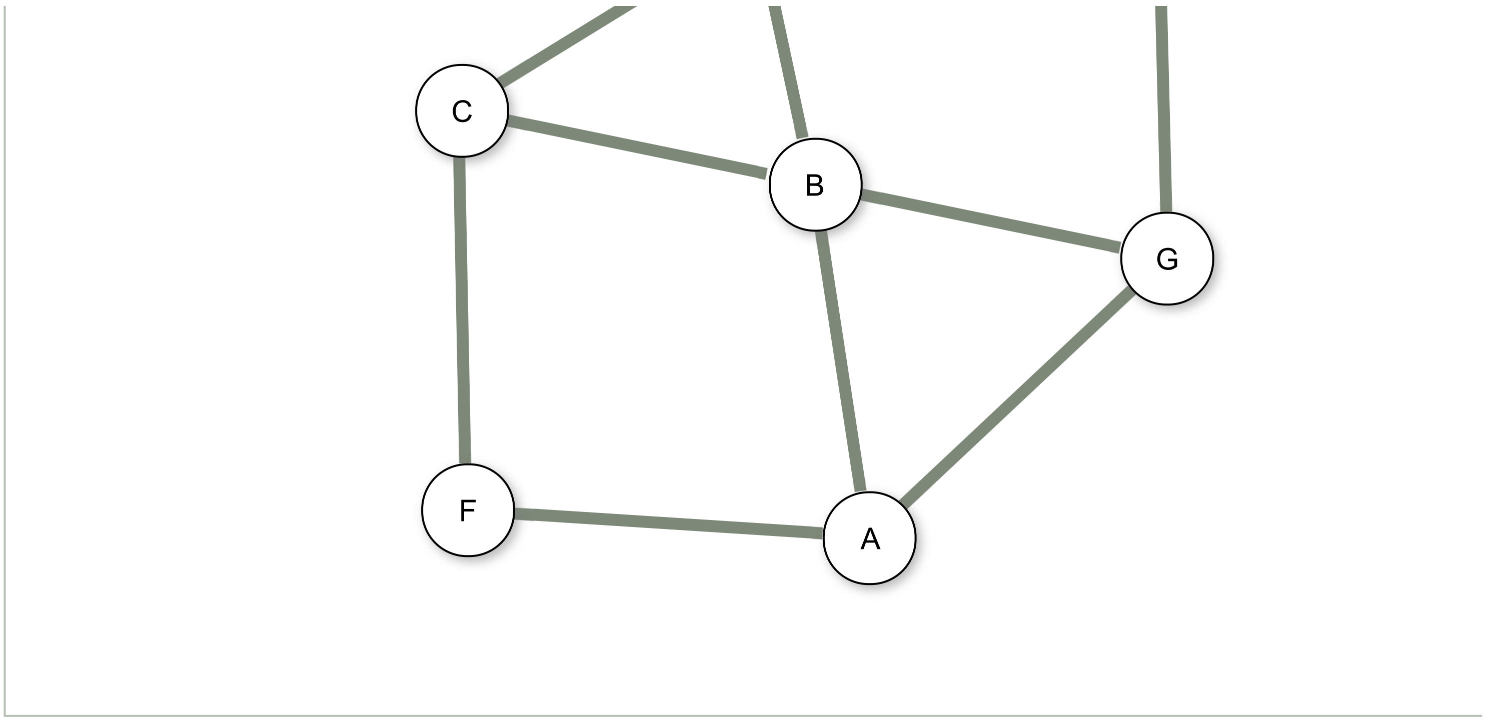
Here is an exercise for you to practice BFS.

Reset Model Answer

Instructions:

Reproduce the behavior of the BFS algorithm for the graph below. Just click on the **edges** in the order that t will be traversed by the BFS algorithm. Start with Node A. If there is more than one node that could be visite next, choose the one that comes first in alphabetical order.

Score: 0 / 6, Points remaining: 6, Points lost: 0


C
B
G
F
A

Appendix A: Glossary

10.1. Glossary

2-3 tree

A specialized form of the **B-tree** where each internal node has either 2 children or 3 children. Key values are ordered to maintain the **binary search tree property**. The 2-3 tree is always height balanced, and its insert, search, and remove operations all have $\Theta(\log n)$ cost.

80/20 rule

Given a typical application where there is a collection of records and a series of search operations for records, the 80/20 rule is an empirical observation that 80% of the record accessess typically go to 20% of the records. The exact values varies between data collections, and is related to the concept of **locality of reference**.

abstract data type

Abbreviated **ADT**. The specification of a **data type** within some language, independent of an implementation. The **interface** for the ADT is defined in terms of a **type** and a set of operations on that type. The behavior of each operation is determined by its inputs and outputs. An ADT does not specify *how* the data type is implemented. These implementation details are hidden from the user of the ADT and protected from outside access, a concept referred to as **encapsulation**.

accept

When a **finite automata** executes on a string and terminates in an **accepting state**, it is said to accept the string. The finite automata is said to accept the language that consists of all strings for which the finite automata completes execution in an accepting state.

accepting state

Part of the definition of a **finite automata** is to designate some **states** as accepting states. If the finite automata executes on an input string and completes the computation in an accepting state, then the machine is said to **accept** the string.

activation record

The entity that is stored on the **runtime stack** during program execution. It stores any active **local variable** and the return address from which a new subroutine is being called, so that this information can be recovered when the subroutine terminates.

acyclic graph

In **graph** terminology, a graph that contains no **cycles**.

address

A location in memory.

adjacency list

An implementation for a **graph** that uses an (array-based) **list** to represent the **vertices** of the graph, and each vertex is in turn represented by a (linked) list of the vertices that are **neighbors**.

adjacency matrix

An implementation for a **graph** that uses a 2-dimensional **array** where each row and each column corresponds to a **vertex** in the **graph**. A given row and column in the matrix corresponds to an edge from the **vertex** corresponding to the row to the vertex corresponding to the column.

adjacent

Two **nodes** of a **tree** or two **vertices** of a **graph** are said to be adjacent if they have an **edge** connecting them. If the edge is directed from a to b, then we say that a is adjacent to b, and b is adjacent from a.

ADT

Abbreviation for **abstract data type**.

adversary

A fictional construct introduced for use in an **adversary argument**.

adversary argument

A type of **lower bounds proof** for a problem where a (fictional) "adversary" is assumed to control access to an algorithm's input, and which yields information about that input in such a way that will drive the cost for any proposed algorithm to solve the problem as high as possible. So long as the adversary never gives an answer that conflicts with any previous answer, it is permitted to do whatever necessary to make the algorithm require as much cost as possible.

aggregate type

A **data type** whose **members** have subparts. For example, a typical database record. Another term for this is **composite type**.

algorithm

A method or a process followed to solve a **problem**.

algorithm analysis

A less formal version of the term **asymptotic algorithm analysis**, generally used as a synonym for **asymptotic analysis**.

alias

Another name for something. In programming, this usually refers to two **references** that refer to the same object.

all-pairs shortest paths problem

Given a **graph** with **weights** or distances on the **edges**, find the shortest paths between every pair of vertices in the graph. One approach to solving this problem is **Floyd's algorithm**, which uses the **dynamic programming** algorithmic technique.

allocated
allocation

Reserving memory for an object in the Heap memory.

alphabet

The characters or symbols that strings in a given language may be composed of.

alphabet trie

A **trie** data structure for storing variable-length strings. Level i of the tree corresponds to the letter in position i of the string. The root will have potential branches on each intial letter of string. Thus, all strings starting with "a" will be stored in the "a" branch of the tree. At the second level, such strings will be separated by branching on the second letter.

amortized analysis

An **algorithm analysis** techique that looks at the total cost for a series of operations and amortizes this total cost over the full series. This is as opposed to considering every individual operation to independently have the **worst case** cost, which might lead to an overestimate for the total cost of the series.

amortized cost

The total cost for a series of operations to be used in an **amortized analysis**.

ancestor

In a tree, for a given node A, any node on a **path** from A up to the root is an ancestor of A.

antisymmetric

In set notation, relation R is antisymmetric if whenever aRb and bRa, then $a = b$, for all $a, b \in \mathbf{S}$.

approximation algorithm

An algorithm for an **optimization problem** that finds a good, but not necessarily cheapest, solution.

arm

In the context of an **I/O head**, this attaches the sensor on the I/O head to the **boom**.

array

A **data type** that is used to store elements in consecutive memory locations and refers to them by an index.

array-based list

An implementation for the **list** ADT that uses an **array** to store the list elements. Typical implementations fix the array size at creation of the list, and the **overhead** is the number of array positions that are presently unused.

array-based queue

Analogous to an **array-based list**, this uses an **array** to store the elements when implementing the **queue** ADT.

array-based stack

Analogous to an **array-based list**, this uses an **array** to store the elements when implementing the **stack** ADT.

ASCII character coding

American Standard Code for Information Interchange. A commonly used method for encoding characters using a binary code. Standard ASCII uses an 8-bit code to represent upper and lower case letters, digits, some punctuation, and some number of non-printing characters (such as carrage return). Now largely replaced by UTF-8 encoding.

assembly code

A form of **intermediate code** created by a **compiler** that is easy to convert into the final form that the computer can execute. An assembly language is typically a direct mapping of one or a few instructions that the CPU can execute into a mnemonic form that is relatively easy for a human to read.

asymptotic algorithm analysis

A more formal term for **asymptotic analysis**.

asymptotic analysis

A method for estimating the efficiency of an algorithm or computer program by identifying its **growth rate**. Asymptotic analysis also gives a way to define the inherent difficulty of a **problem**. We frequently use the term **algorithm analysis** to mean the same thing.

attribute

In **object-oriented programming**, a synonym for **data members**.

automata

Synonym for **finite state machine**.

automatic variable

A synonym for **local variable**. When program flow enters and leaves the variable's scope, automatic variables will be allocated and de-allocated automatically.

average case

In **algorithm analysis**, the average of the costs for all **problem instances** of a given input size n. If not all problem instances have equal probability of occurring, then average case must be calculated using a weighted average.

average seek time

Expected (average) time to perform a **seek** operation on a **disk drive**, assuming that the seek is between two randomly selected tracks. This is one of two metrics commonly provided by disk drive vendors for disk drive performance, with the other being **track-to-track seek time**.

AVL Tree

A variant implementation for the **BST**, which differs from the standard BST in that it uses modified insert and remove methods in order to keep the tree **balanced**. Similar to a **Splay Tree** in that it uses the concept of **rotations** in the insert and remove operations.

B*-tree

A variant on the **B$^+$-tree**. The B^* tree is identical to the B^+ tree, except for the rules used to split and merge nodes. Instead of splitting a node in half when it overflows, the B^* tree gives some records to its neighboring sibling, if possible. If the sibling is also full, then these two nodes split into three. Similarly, when a node underflows, it is combined with its two siblings, and the total reduced to two nodes. Thus, the nodes are always at least two thirds full.

B$^+$-tree

The most commonly implemented form of **B-tree**. A B$^+$-tree does not store data at the **internal nodes**, but instead only stores **search key** values as direction finders for the purpose of searching through the tree. Only the **leaf nodes** store a **reference** to the actual data records.

B-tree

A method for **indexing** a large collection of records. A B-tree is a **balanced tree** that typically has high branching factor (commonly as much as 100 **children** per **internal node**), causing the tree to be very shallow. When stored on disk, the node size is selected to be same as the desired unit of I/O (so some multiple of the disk **sector** size). This makes it easy to gain access to the record associated with a given **search key** stored in the tree with few **disk accesses**. The most commonly implemented variant of the B-tree is the **B$^+$-tree**.

backing storage

In the context of a **caching** system or **buffer pool**, backing storage is the relatively large but slower source of data that needs to be cached. For example, in a **virtual memory**, the disk drive would be the backing storage. In the context of a web browser, the Internet might be considered the backing storage.

backtracking

A **heuristic** for brute-force search of a solution space. It is essentially a **depth-first search** of the solution space. This can be improved using a **branch-and-bounds algorithm**.

bad reference

A reference is referred to as a bad reference if it is allocated but not initialized.

bag

In set notation, a bag is a collection of elements with no order (like a set), but which allows for duplicate-valued elements (unlike a set).

balanced tree

A **tree** where the **subtrees** meet some criteria for being balanced. Two possibilities are that the tree is **height balanced**, or that the tree has a roughly equal number of **nodes** in each subtree.

base

Synonym for **radix**.

base case

In **recursion** or **proof by induction**, the base case is the termination condition. This is a simple input or value that can be solved (or proved in the case of induction) without resorting to a recursive call (or the **induction hypothesis**).

base class

In **object-oriented programming**, a class from which another class **inherits**. The class that inherits is called a **subclass**.

base type

The **data type** for the elements in a set. For example, the set might consist of the integer values 3, 5, and 7. In this example, the base type is integers.

basic operation

Examples of basic operations include inserting a data item into the data structure, deleting a **data item** from the data structure, and finding a specified **data item**.

best case

In algorithm analysis, the **problem instance** from among all problem instances for a given input size n that has least cost. Note that the best case is **not** when n is small, since we are referring to the best from a class of inputs (i.e, we want the best of those inputs of size n).

best fit

In a **memory manager**, best fit is a **heuristic** for deciding which **free block** to use when allocating memory from a **memory pool**. Best fit will always allocate from the smallest **free block** that is large enough to service the memory request. The rationale is that this will be the method that best preserves large blocks needed for unusually large requests. The disadvantage is that it tends to cause **external fragmentation** in the form of small, unuseable memory blocks.

BFS

Abbreviation for **breadth-first search**.

big-Oh notation

In **algorithm analysis**, a shorthand notation for describing the **upper bound** for an **algorithm** or **problem**.

binary insert sort

A variation on **insertion sort** where the position of the value being inserted is located by binary search, and then put into place. In normal usage this is not an improvement on standard insertion sort because of the expense of moving many items in the **array**. But it is directly useful if the cost of comparison is high compared to that of moving an element, or is theoretically useful if we only care to count the cost of comparisons.

binary search

A standard **recursive** algorithm for finding the **record** with a given **search key** value within a sorted list. It runs in $O(\log n)$ time. At each step, look at the middle of the current sublist, and throw away the half of the records whose keys are either too small or too large.

binary search tree

A binary tree that imposes the following constraint on its node values: The **search key** value for any node A must be greater than the (key) values for all nodes in the left **subtree** of A, and less than the key values for all nodes in the right subtree of A. Some convention must be adopted if multiple nodes with the same key value are permitted, typically these are required to be in the right subtree.

binary search tree property

The defining relationship between the **key** values for **nodes** in a **BST**. All nodes stored in the left subtree of a node whose key value is K have key values less than or equal to K. All nodes stored in the right subtree of a node whose key value is K have key values greater than K.

binary tree

A finite set of nodes which is either empty, or else has a root node together two binary trees, called the left and right **subtrees**, which are **disjoint** from each other and from the **root**.

binary trie

A **binary tree** whose structure is that of a **trie**. Generally this is an implementation for a **search tree**. This means that the **search key** values are thought of a binary digits, with the digit in the position corresponding to this a node's **level** in the tree indicating a left branch if it is "0", or a right branch if it is "1". Examples include the **Huffman coding tree** and the **Bintree**.

binning

In **hashing**, binning is a type of **hash function**. Say we are given keys in the range 0 to 999, and have a hash table of size 10. In this case, a possible hash function might simply divide the key value by 100. Thus, all keys in the range 0 to 99 would hash to slot 0, keys 100 to 199 would hash to slot 1, and so on. In other words, this hash function "bins" the first 100 keys to the first slot, the next 100 keys to the second slot, and so on. This approach tends to make the hash function dependent on the distribution of the high-order bits of the keys.

Binsort

A sort that works by taking each record and placing it into a bin based on its value. The bins are then gathered up in order to sort the list. It is generally not practical in this form, but it is the conceptual underpinning of the **radix sort**.

bintree

A **spatial data structure** in the form of binary **trie**, typically used to store point data in two or more dimensions. Similar to a **PR quadtree** except that at each level, it splits one dimension in half. Since many leaf nodes of the PR quadtree will contain no data points, implementation often makes use of the **flyweight design pattern**.

bitmap

bit vector

An **array** that stores a single bit at each position. Typically these bits represent **Boolean variables** associated with a collection of objects, such that the i th bit is the Boolean value for the i th object.

block

A unit of storage, usually referring to storage on a **disk drive** or other **peripheral storage** device. A block is the basic unit of I/O for that device.

Boolean expression

A Boolean expression is comprised of **Boolean variables** combined using the operators AND $(\cdot)$, OR $(+)$, and NOT (to negate Boolean variable x we write $\overline{x}$).

Boolean variable

A variable that takes on one of the two values `True` and `False`.

boom

In the context of an **I/O head**, is the central structure to which all of the I/O heads are attached. Thus, the all move together during a **seek** operation.

bounding box

A box (usually aligned to the coordinate axes of the reference system) that contains a (potentially complex) object. In graphics and computational geometry, complex objects might be associated with a bounding box for use by algorithms that search for objects in a particular location. The idea is that if the bounding box is not within the area of interest, then neither is the object. Checking the bounding box is cheaper than checking the object, but it does require some time. So if enough objects are not outside the area of interest, this approach will not save time. But if most objects are outside of the area of interest, then checking bounding boxes first can save a lot of time.

branch-and-bounds algorithm

A variation on **backtracking** that applies to **optimization problems**. We traverse the **solution tree** as with backtracking. Proceeding deeper in the solution tree generally requires additional cost. We remember the best-cost solution found so far. If the cost of the current branch in the tree exceeds the best tour cost found so far, then we know to stop pursuing this branch of the tree. At this point we can immediately back up and take another branch.

breadth-first search

A **graph traversal** algorithm. As the name implies, all immediate **neighbors** for a **node** are **visited** before any more-distant nodes are visited. BFS is driven by a **queue**. A start vertex is placed on the queue. Then, until the queue is empty, a node is taken off the queue, visited, and and then any **unvisited** neighbors are placed onto the queue.

break-even point

The point at which two costs become even when measured as the function of some variable. In particular, used to compare the space requirements of two implementations. For example, when comparing the space requirements of an **array-based list** implementation versus a **linked list** implementation, the key issue is how full the list is compared to its capacity limit (for the array-based list). The point where the two representations would have the same space cost is the break-even point. As the list becomes more full beyond this point, the array-based list implementation becomes more space efficent, while as the list becomes less full below this point, the linked list implementation becomes more space efficient.

BST

Abbreviation for **binary search tree**.

bubble sort

A simple sort that requires $Theta(n^2)$ time in **best**, **average**, and **worst** cases. Even an optimized version will normally run slower than **insertion sort**, so it has little to recommend it.

bucket

In **bucket hashing**, a bucket is a sequence of **slots** in the **hash table** that are grouped together.

bucket hashing

A method of **hashing** where multiple **slots** of the **hash table** are grouped together to form a **bucket**. The **hash function** then either hashes to some bucket, or else it hashes to a **home slot** in the normal way, but this home slot is part of some bucket. **Collision resolution** is handled first by attempting to find a free position within the same bucket as the home slot. If the bucket if full, then the record is placed in an **overflow bucket**.

bucket sort

A variation on the **Binsort**, where each bin is associated with a range of **key** values. This will require some method of sorting the records placed into each bin.

buddy method

In a **memory manager**, an alternative to using a **free block list** and a **sequential fit** method to seach for a suitable free block to service a **memory request**. Instead, the memory pool is broken down as needed into smaller chunks by splitting it in half repeatedly until the smallest power of 2 that is as big or bigger than the size of the memory request is reached. The name comes from the fact that the binary representation for the start of the block positions only differ by one bit for adjacent blocks of the same size. These are referred to as “buddies” and will be merged together if both are free.

buffer

A block of memory, most often in **primary storage**. The size of a buffer is typically one or a multiple of the basic unit of I/O that is read or written on each access to **secondary storage** such as a **disk drive**.

buffer passing

An approach to implementing the **ADT** for a **buffer pool**, where a pointer to a **buffer** is passed between the client and the buffer pool. This is in contrast to a **message passing** approach, it is most likely to be used for long messages or when the message size is always the same as the buffer size, such as when implementing a **B-tree**.

buffer pool

A collection of one or more **buffers**. The buffer pool is an example of a **cache**. It is stored in **primary storage**, and holds data that is expected to be used in the near future. When a data value is requested, the buffer pool is searched first. If the value is found in the buffer pool, then **secondary storage** need not be accessed. If the value is not found in the buffer pool, then it must be fetched from secondary storage. A number of traditional **heuristics** have been developed for deciding which data to **flush** from the buffer pool when new data must be stored, such as **least recently used**.

buffering

A synonym for **caching**. More specifically, it refers to an arrangement where all accesses to data (such as on a **peripheral storage** device) must be done in multiples of some minimum unit of storage. On a **disk drive**, this basic or smallest unit of I/O is a **sector**. It is called “buffering” because the block of data returned by such an access is stored in a **buffer**.

caching

The concept of keeping selected data in **main memory**. The goal is to have in main memory the data values that are most likely to be used in the near future. An example of a caching technique is the use of a **buffer pool**.

call stack

Known also as execution stack. A stack that stores the function call sequence and the return address for each function.

Cartesian product

For sets, this is another name for the **set product**.

ceiling

Written $\lceil x \rceil$, for real value x the ceiling is the least integer $\geq x$.

child

In a tree, the set of **nodes** directly pointed to by a node R are the **children** of R.

circular first fit

In a **memory manager**, circular first fit is a **heuristic** for deciding which **free block** to use when allocating memory from a **memory pool**. Circular first fit is a minor modification on **first fit** memory allocation, where the last free block allocated from is remembered, and search for the next suitable free block picks up from there. Like first fit, it has the advantage that it is typically not necessary to look at all free blocks on the free block list to find a suitable free block. And it has the advantage over first fit that it spreads out memory allocations evenly across the **free block list**. This might help to minimize **external fragmentation**.

circular list

A **list** ADT implementation variant where the last element of the list provides access to the first element of the list.

class

In the **object-oriented programming paradigm** an ADT and its implementation together make up a class. An instantiation of a class within a program is termed an **object**.

class hierarchy

In **object-oriented programming**, a set of classes and their interrelationships. One of the classes is the **base class**, and the others are **subclasses** that **inherit** either directly or indirectly from the base class.

clause

In a **Boolean expression**, a clause is one or more **literals** OR'ed together.

client

The user of a service. For example, the object or part of the program that calls a **memory manager** class is the client of that memory manager. Likewise the class or code that calls a **buffer pool**.

clique

In **graph** terminology, a clique is a **subgraph**, defined as any **subset** U of the graph's **vertices** such that every vertex in U has an **edge** to every other vertex in U. The size of the clique is the number of vertices in the clique.

closed

A set is closed over a (binary) operation if, whenever the operation is applied to two members of the set, the result is a member of the set.

closed hash system

A **hash system** where all records are stored in slots of the **hash table**. This is in contrast to an **open hash system**.

closed-form solution

An algebraic equation with the same value as a **summation** or **recurrence relation**. The process of replacing the summation or recurrence with its closed-form solution is known as solving the summation or recurrence.

cluster

In **file processing**, a collection of physically adjacent **sectors** that define the smallest allowed allocation unit of space to a disk file. The idea of requiring space to be allocated in multiples of sectors is that this will reduce the number of **extents** required to store the file, which reduces the expected number of **seek** operations reuquired to process a series of **disk accesses** to the file. The disadvantage of large cluster size is that it increases **internal fragmentation** since any space not actually used by the file in the last cluster is wasted.

code generation

A phase in a **compiler** that transforms **intermediate code** into the final executable form of the code. More generally, this can refer to the process of turning a parse tree (that determines the correctness of the structure of the program) into actual instructions that the computer can execute.

code optimization

A phase in a **compiler** that makes changes in the code (typically **assembly code**) with the goal of replacing it with a version of the code that will run faster while performing the same computation.

cohesion

In **object-oriented programming**, a term that refers to the degree to which a class has a single well-defined role or responsibility.

Collatz sequence

For a given integer value n, the sequence of numbers that derives from performing the following computatin on n

Toggle Tree View

```
while (n > 1)
  if (ODD(n))
    n = 3 * n + 1;
  else
    n = n / 2;
```

This is famous because, while it terminates for any value of n that you try, it has never been proven to be a fact that this always terminates.

collision

In a **hash system**, this refers to the case where two search **keys** are mapped by the **hash function** to the same slot in the **hash table**. This can happen on insertion or search when another record has already been hashed to that slot. In this case, a **closed hash system** will require a process known as **collision resolution** to find the location of the desired record.

collision resolution

The outcome of a **collision resolution policy**.

collision resolution policy

In **hashing**, the process of resolving a **collision**. Specifically in a **closed hash system**, this is the process of finding the proper position in a **hash table** that contains the desired record if the **hash function** did not return the correct position for that record due to a **collision** with another record.

comparable

The concept that two objects can be compared to determine if they are equal or not, or to determine which one is greater than the other. In set notation, elements x and y of a set are comparable under a given relation R if either xRy or yRx. To be reliably compared for a greater/lesser relationship, the values being compared must belong to a **total order**. In programming, the property of a data type such that two elements of the type can be compared to determine if they the same (a weaker version), or which of the two is larger (a stronger version). `Comparable` is also the name of an **interface** in Java that asserts a comparable relationship between objects with a class, and `.compareTo()` is the `Comparable` interface method that implements the actual comparison between two objects of the class.

comparator

A function given as a parameter to a method of a library (or alternatively, a parameter for a C++ template or a Java generic). The comparator function concept provides a generic way encapulates the process of performing a comparison between two objects of a specific type. For example, if we want to write a generic sorting routine, that can handle any record type, we can require that the user of the sorting routine pass in a comparator function to define how records in the collection are to be compared.

comparison

The act of comparing two **keys** or **records**. For many **data types**, a comparison has constant time cost. The number of comparisons required is often used as a **measure of cost** for sorting and searching algorithms.

compile-time polymorphism

A form of **polymorphism** known as Overloading. Overloaded methods have the same names, but different signatures as a method available elsewhere in the class. Compare to **run-time polymorphism**.

compiler

A computer program that reads computer programs and converts them into a form that can be directly excecuted by some form of computer. The major phases in a compiler include **lexical analysis**, **syntax analysis**, **intermediate code generation**, **code optimization**, and **code generation**. More broadly, a compiler can be viewed as **parsing** the program to verify that it is syntactically correct, and then doing **code generation** to convert the hig-level program into something that the computer can execute.

complete binary tree

A binary tree where the nodes are filled in row by row, with the bottom row filled in left to right. Due to this requirement, there is only one tree of n nodes for any value of n. Since storing the records in an **array** in row order leads to a simple mapping from a node's position in the array to its **parent**, **siblings**, and **children**, the array representation is most commonly used to implement the complete binary tree. The **heap** data structure is a complete binary tree with partial ordering constraints on the node values.

complete graph

A **graph** where every **vertex** connects to every other vertex.

complex number

In mathematics, an imaginary number, that is, a number with a real component and an imaginary component.

Composite design pattern

Given a class hierarchy representing a set of objects, and a container for a collection of objects, the composite **design pattern** addresses the relationship between the object hierarchy and a bunch of behaviors on the objects. In the composite design, each object is required to implement the collection of behaviors. This is in contrast to the procedural approach where a behavior (such as a tree **traversal**) is implemented as a method on the object collection (such as a **tree**). Procedural tree traversal requires that the tree have a method that understands what to do when it encounters any of the object types (**internal** or **leaf nodes**) that the tree might contain. The composite approach would have the tree call the "traversal" method on its root node, which then knows how to perform the "traversal" behavior. This might in turn require invoking the traversal method of other objects (in this case, the children of the root).

composite type

A type whose **members** have subparts. For example, a typical database record. Another term for this is **aggregate type**.

composition

Relationships between classes based on usage rather than **inheritance**, i.e. a **HAS-A** relationship. For example, some code in class 'A' has a **reference** to some other class 'B'.

computability

A branch of computer science that deals with the theory of solving problems through computation. More specificially, it deals with the limits to what problems (functions) are computable. An example of a famous problem that cannot in principle be solved by a computer is the **halting problem**.

computation

In a **finite automata**, a computation is a sequence of **configurations** for some length $n \geq 0$. In general, it is a series of operations that the machine performs.

computational complexity theory

A branch of the theory of computation in theoretical computer science and mathematics that focuses on classifying computational problems according to their inherent difficulty, and relating those classes to each other. An example is the study of **NP-Complete** problems.

configuration

For a **finite automata**, a complete specification for the current condition of the machine on some input string. This includes the current **state** that the machine is in, and the current condition of the string, including which character is about to be processed.

Conjunctive Normal Form
CNF

A **Boolean expression** written as a series of **clauses** that are AND'ed together.

connected component

In an **undirected graph**, a **subset** of the **nodes** such that each node in the subset can be reached from any other node in that subset.

connected graph

An **undirected graph** is a connected graph if there is at least one path from any **vertex** to any other.

constant running time

The cost of a function whose running time is not related to its input size. In Theta notation, this is traditionally written as $\Theta(1)$.

constructive induction

A process for finding the **closed form** for a **recurrence relation**, that involves substituting in a guess for the closed form to replace the recursive part(s) of the recurrence. Depending on the goal (typically either to show that the hypothesized growth rate is right, or to find the precise constants), one then manipulates the resulting non-recursive equation.

container
container class

A **data structure** that stores a collection of **records**. Typical examples are **arrays**, **search trees**, and **hash tables**.

context-free grammar

A **grammar** comprised only of productions of the form $A \to x$ where A is a **non-terminal** and x is a series of one or more **terminals** and non-terminals. That is, the given non-terminal A can be replaced at any time.

context-free language

The set of **languages** that can be defined by **context-sensitive grammars**.

context-sensitive grammar

A **grammar** comprised only of productions of the form $xAy \to xvy$ where A is a **non-terminal** and x and y are each a series of one or more **terminals** and non-terminals. That is, the given non-terminal A can be replaced only when it is within the proper context.

cost

The amount of resources that the solution consumes.

cost model

In **algorithm analysis**, a definition for the cost of each **basic operation** performed by the algorithm, along with a definition for the size of the input. Having these definitions allows us to calculate the **cost** to run the algorithm on a given input, and from there determine the **growth rate** of the algorithm. A cost model would be considered “good” if it yields predictions that conform to our understanding of reality.

countably infinite
countable

A **set** is countably infinite if it contains a finite number of elements, or (for a set with an infinite number of elements) if there exists a one-to-one mapping from the set to the set of integers.

CPU

Acronym for Central Processing Unit, the primary processing device for a computer.

current position

A property of some list ADTs, where there is maintained a “current position” state that can be referred to later.

cycle

In **graph** terminology, a **cycle** is a **path** of length three or more that connects some **vertex** v_1 to itself.

cylinder

A **disk drive** normally consists of a stack of **platters**. While this might not be so true today, traditionally all of the **I/O heads** moved together during a **seek** operation. Thus, when a given I/O head is positioned over a particular **track** on a platter, the other I/O heads are also positioned over the corresponding track on their platters. That collection of tracks is called a cylinder. A given cylinder represents all of the data that can be read from all of the platters without doing another seek operation.

cylinder index

In the **ISAM** system, a simple **linear index** that stores the lowest key value stored in each **cylinder**.

cylinder overflow

In the **ISAM** system, this is space reserved for storing any records that can not fit in their respective **cylinder**.

DAG

Abbreviation for **directed acyclic graph**.

data field

In **object-oriented programming**, a synonym for **data member**.

data item

A piece of information or a record whose value is drawn from a type.

data member

The variables that together define the space required by a data item are referred to as data members. Some of the commonly used synonyms include **data field**, **attribute**, and **instance variable**.

data structure

The implementation for an **ADT**.

data type

A type together with a collection of operations to manipulate the type.

deallocated
deallocation

Free the memory allocated to an unused object.

decision problem

A problem whose output is either “YES” or “NO”.

decision tree

A theoretical construct for modeling the behavior of algorithms. Each point at which the algorithm makes a decision (such as an if statement) is modeled by a branch in the tree that represents the algorithms behavior. Decision trees can be used in **lower bounds proofs**, such as the proof that sorting requires $\Omega(n \log n)$ comparisons in the **worst case**.

deep copy

Copying the actual content of a **pointee**.

degree

In **graph** terminology, the degree for a **vertex** is its number of **neighbors**. In a **directed graph**, the **in degree** is the number of edges directed into the vertex, and the **out degree** is the number of edges directed out of the vertex. In **tree** terminology, the degree for a **node** is its number of **children**.

delegation mental model for recursion

A way of thinking about the process of **recursion**. The recursive function “delegates” most of the work when it makes the recursive call. The advantage of the delegation mental model for recursion is that you don’t need to think about how the delegated task is performed. It just gets done.

dense graph

A **graph** where the actual number of **edges** is a large fraction of the possible number of edges. Generally, this is interpreted to mean that the **degree** for any **vertex** in the graph is relatively high.

depth

The depth of a node M in a tree is the length of the path from the root of the tree to M.

depth-first search

A **graph traversal** algorithm. Whenever a v is **visited** during the traversal, DFS will **recursively** visit all of v 's **unvisited neighbors**.

depth-first search tree

A **tree** that can be defined by the operation of a **depth-first search** (DFS) on a **graph**. This tree would consist of the **nodes** of the graph and a subset of the **edges** of the graph that was followed during the DFS.

dequeue

A specialized term used to indicate removing an element from a queue.

dereference

Accessing the value of the **pointee** for some **reference** variable. Commonly, this happens in a language like Java when using the “dot” operator to access some field of an object.

derivation

In formal languages, the process of executing a series of **production rules** from a **grammar**. A typical example of a derivation would be the series of productions executed to go from the **start symbol** to a given string.

descendant

In a tree, the set of all nodes that have a node A as an **ancestor** are the descendants of A. In other words, all of the nodes that can be reached from A by progressing downwards in tree. Another way to say it is: The **children** of A, their children, and so on.

deserialization

The process of returning a **serialized** representation for a data structure back to its original in-memory form.

design pattern

An abstraction for describing the design of programs, that is, the interactions of objects and classes. Experienced software designers learn and reuse patterns for combining software components, and design patterns allow this design knowledge to be passed on to new programmers more quickly.

deterministic

Any **finite automata** in which, for every pair of **state** and symbol, there is only a single transition. This means that whenever the machine is in a given state and sees a given symbol, only a single thing can happen. This is in contrast to a **non-deterministic** finite automata, which has at least one state with multiple transitions on at least one symbol.

deterministic algorithm

An algorithm that does not involve any element of randomness, and so its behavior on a given input will always be the same. This is in contrast to a **randomized algorithm**.

Deterministic Finite Automata
Deterministic Finite Acceptor

DFA

An **automata** or abstract machine that can process an input string (shown on a tape) from left to right. There is a control unit (with **states**), behavior defined for what to do when in a given state and with a given symbol on the current square of the tape. All that we can “do” is change state before going to the next letter to the right.

DFS

Abbreviation for **depth-first search**.

diagonalization argument

A proof technique for proving that a set is **uncountably infinite**. The approach is to show that, no matter what order the elements of the set are put in, a new element of the set can be constructed that is not in that ordering. This is done by changing the i th value or position of the element to be different from that of the i th element in the proposed ordering.

dictionary

An abstract data type or **interface** for a data structure or software subsystem that supports insertion, search, and deletion of records.

dictionary search

A close relative of an **interpolation search**. In a classical (paper) dictionary of words in a natural language, there are markings for where in the dictionary the words with a given letter start. So in typical usage of such a dictionary, words are found by opening the dictionary to some appropriate place within the pages that contain words starting with that letter.

digraph

Abbreviation for **directed graph**.

Dijkstra’s algorithm

An algorithm to solve the **single-source shortest paths problem** in a **graph**. This is a **greedy algorithm**. It is nearly identical to **Prim’s algorithm** for finding a **minimal-cost spanning tree**, with the only difference being the calculation done to update the best-known distance.

diminishing increment sort

Another name for **Shellsort**.

direct access

A storage device, such as a disk drive, that has some ability to move to a desired data location more-or-less directly. This is in contrast to a **sequential access** storage device such as a tape drive.

direct proof

In general, a direct proof is just a “logical explanation”. A direct proof is sometimes referred to as an argument by deduction. This is simply an argument in terms of logic. Often written in English with words such as “if … then”, it could also be written with logic notation such as $P \Rightarrow Q$.

directed acyclic graph

A **graph** with no cycles. Abbreviated as **DAG**. Note that a DAG is not necessarily a **tree** since a given **node** might have multiple **parents**.

directed edge

An **edge** that goes from **vertex** to another. In contrast, an **undirected edge** simply links to vertices without a direction.

directed graph

A **graph** whose **edges** each are directed from one of its defining **vertices** to the other.

dirty bit

Within a **buffer pool**, a piece of information associated with each **buffer** that indicates whether the contents of the buffer have changed since being read in from **backing storage**. When the buffer is **flushed** from the buffer pool, the buffer's contents must be written to the backing storage if the dirty bit is set (that is, if the contents have changed). This means that a relatively expensive write operation is required. In contrast, if the dirty bit is not set, then it is unnecessary to write the contents to backing storage, thus saving time over not keeping track of whether the contents have changed or not.

Discrete Fourier Transform
DFT

Let $a = [a_0, a_1, \ldots, a_{n-1}]^T$ be a vector that stores the coefficients for a polynomial being evaluated. We can then do the calculations to evaluate the polynomial at the n th $roots of unity < nth roots of unit >$ by multiplying the A_z matrix by the coefficient vector. The resulting vector F_z is called the Discrete Fourier Transform (or DFT) for the polynomial.

discriminator

A part of a **multi-dimensional search key**. Certain tree data structures such as the **bintree** and the **kd tree** operate by making branching decisions at nodes of the tree based on a single attribute of the multi-dimensional key, with the attribute determined by the level of the node in the tree. For example, in 2 dimensions, nodes at the odd levels in the tree might branch based on the x value of a coordinate, while at the even levels the tree would branch based on the y value of the coordinate. Thus, the x coordinate is the discriminator for the odd levels, while the y coordinate is the discriminator for the even levels.

disjoint

Two parts of a **data structure** or two collections with no objects in common are disjoint. This term is often used in conjunction with a data structure that has **nodes** (such as a **tree**). Also used in the context of **sets**, where two **subsets** are disjoint if they share no elements.

disjoint sets

A collection of **sets**, any pair of which share no elements in common. A collection of disjoint sets partitions some objects such that every object is in exactly one of the disjoint sets.

disk access

The act of reading data from a disk drive (or other form of **peripheral storage**). The number of times data must be read from (or written to) a disk is often a good measure of cost for an algorithm that involves disk I/O, since this is usually the dominant cost.

disk controller

The control mechanism for a **disk drive**. Responsible for the action of reading or writing a **sector** of data.

disk drive

An example of **peripheral storage** or **secondary storage**. Data access times are typically measured in thousandths of a second (milliseconds), which is roughly a million times slower than access times for **RAM**, which is an example of a **primary storage** device. Reads from and writes to a disk drive are always done in terms of some minimum size, which is typically called a **block**. The block size is 512 bytes on most disk drives. Disk drives and RAM are typical parts of a computer's **memory hierarchy**.

disk I/O

Refers to the act of reading data from or writing data to a **disk drive**. All disk reads and writes are done in units of a **sector** or **block**.

disk-based space/time tradeoff

In contrast to the standard **space/time tradeoff**, this principle states that the smaller you can make your disk storage requirements, the faster your program will run. This is because the time to read information from disk is enormous compared to computation time, so almost any amount of additional computation needed to unpack the data is going to be less than the disk-reading time saved by reducing the storage requirements.

distance

In **graph** representations, a synonym for **weight**.

divide and conquer

A technique for designing algorithms where a solution is found by breaking the problem into smaller (similar) subproblems, solving the subproblems, then combining the subproblem solutions to form the solution to the original problem. This process is often implemented using **recursion**.

divide-and-conquer recurrences

A common form of **recurrence relation** that have the form

$$\mathbf{T}(n) = a\mathbf{T}(n/b) + cn^k; \quad \mathbf{T}(1) = c$$

where a, b, c, and k are constants. In general, this recurrence describes a problem of size n divided into a subproblems of size n/b, while cn^k is the amount of work necessary to combine the partial solutions.

divide-and-guess

A technique for finding a **closed-form solution** to a **summation** or **recurrence relation**.

domain

The set of possible inputs to a function.

double buffering

The idea of using multiple **buffers** to allow the **CPU** to operate in parallel with a **peripheral storage** device. Once the first buffer's worth of data has been read in, the CPU can process this while the next block of data is being read from the peripheral storage. For this idea to work, the next block of data to be processed must be known or predicted with reasonable accuracy.

double hashing

A **collision resolution** method. A second hash function is used to generate a value c on the key. That value is then used by this key as the step size in **linear probing by steps**. Since different keys use different step sizes (as generated by the second hash function), this process avoids the clustering caused by standard linear probing by steps.

double rotation

A type of **rebalancing operation** used by the **Splay Tree** and **AVL Tree**.

doubly linked list

A **linked list** implementation variant where each list node contains access pointers to both the previous element and the next element on the list.

DSA

Abbreviation for Data Structures and Algorithms.

dynamic

Something that is changes (in contrast to **static**). In computer programming, dynamic normally refers to something that happens at run time. For example, run-time analysis is analysis of the program's behavior, as opposed to its (static) text or structure Dynamic binding or dynamic memory allocation occurs at run time.

dynamic allocation

The act of creating an object from **free store**. In C++, Java, and JavaScript, this is done using the `new` operator.

dynamic array

Arrays, once allocated, are of fixed size. A dynamic array puts an **interface** around the array so as to appear to allow the array to grow and shrink in size as necessary. Typically this is done by allocating a new copy, copying the contents of the old array, and then returning the old array to **free store**. If done correctly, the **amortized cost** for dynamically resizing the array can be made constant. In some programming languages such as Java, the term **vector** is used as a synonym for dynamic array.

dynamic memory allocation

A programming technique where linked objects in a data structure are created from **free store** as needed. When no longer needed, the object is either returned to **free store** or left as **garbage**, depending on the programming language.

dynamic programming

An approach to designing algorithms that works by storing a table of results for subproblems. A typical cause for excessive cost in **recursive** algorithms is that different branches of the recursion might solve the same subproblem. Dynamic programming uses a table to store information about which subproblems have already been solved, and uses the stored information to immediately give the answer for any repeated attempts to solve that subproblem.

edge

The connection that links two **nodes** in a **tree**, **linked list**, or **graph**.

edit distance

Given strings S and T, the edit distance is a measure for the number of editing steps required to convert S into T.

efficient

A solution is said to be efficient if it solves the problem within the required **resource constraints**. A solution is sometimes said to be efficient if it requires fewer resources than known alternatives, regardless of whether it meets any particular requirements.

element

One value or member in a set.

empirical comparison

An approach to comparing to things by actually seeing how they perform. Most typically, we are referring to the comparison of two programs by running each on a suite of test data and measuring the actual running times. Empirical comparison is subject to many possible complications, including unfair selection of test data, and inaccuracies in the time measurements due to variations in the computing environment between various executions of the programs.

empty

For a **container** class, the state of containing no **elements**.

encapsulation

In programming, the concept of hiding implementation details from the user of an ADT, and protecting **data members** of an object from outside access.

enqueue

A specialized term used to indicate inserting an element onto a queue.

entry-sequenced file

A file that stores records in the order that they were added to the file.

enumeration

The process by which a **traversal** lists every object in the **container** exactly once. Thus, a traversal that prints the **nodes** is said to enumerate the nodes. An enumeration can also refer to the actual listing that is produced by the traversal (as well as the process that created that listing).

equidistribution property

In random number theory, this means that a given series of random numbers cannot be described more briefly than simply listing it out.

equivalence class

An **equivalence relation** can be used to partition a set into equivalence classes.

equivalence relation

Relation R is an equivalence relation on set $\mathbf{S}$ if it is **reflexive**, **symmetric**, and **transitive**.

estimation

As a technical skill, this is the process of generating a rough estimate in order to evaluate the feasibility of a proposed solution. This is sometimes known as "back of the napkin" or "back of the envelope" calculation. The estimation process can be formalized as (1) determine the major parameters that affect the problem, (2) derive an equation that relates the parameters to the problem, then (3) select values for the parameters and apply the equation to yield an estimated solution.

evaluation

The act of finding the value for a polynomial at a given point.

exact-match query

Records are accessed by unique identifier.

exceptions

Exceptions are techniques used to predict possible runtime errors and handle them properly.

exchange

A swap of adjacent records in an **array**.

exchange sort

A sort that relies solely on exchanges (swaps of adjacent records) to reorder the list. **Insertion Sort** and **Bubble Sort** are examples of exchange sorts. All exchange sorts require $\Theta(n^2)$ time in the **worst case**.

expanding the recurrence

A technique for solving a **recurrence relation**. The idea is to replace the recursive part of the recurrence with a copy of recurrence.

exponential growth rate

A **growth rate** function where n (the input size) appears in the exponent. For example, 2^n.

expression tree

A **tree** structure meant to represent a mathematical expression. **Internal nodes** of the expression tree are operators in the expression, with the subtrees being the sub-expressions that are its operand. All **leaf nodes** are operands.

extent

A physically contiguous block of **sectors** on a **disk drive** that are all part of a given disk file. The fewer extents needed to store the data for a disk file, generally the fewer **seek** operations that will be required to process a series of **disk access** operations on that file.

external fragmentation

A condition that arises when a series of **memory requests** result in lots of small **free blocks**, no one of which is useful for servicing typical requests.

external sort

A sorting algorithm that is applied to data stored in **peripheral storage** such as on a **disk drive**. This is in contrast to an **internal sort** that works on data stored in **main memory**.

factorial

The factorial function is defined as $f(n) = nf(n-1)$ for $n > 0$.

failure policy

In a **memory manager**, a failure policy is the response that takes place when there is no way to satisfy a **memory request** from the current **free blocks** in the **memory pool**. Possibilities include rejecting the request, expanding the memory pool, collecting **garbage**, and reorganizing the memory pool (to collect together free space).

family of languages

Given some class or type of **finite automata** (for example, the **deterministic finite automata**), the set of languages accepted by that class of finite automata is called a family. For example, the **regular languages** is a family defined by the DFAs.

FIFO

Abbreviation for “first-in, first-out”. This is the access paradigm for a **queue**, and an old terminology for the queue is “FIFO list”.

file allocation table

A legacy file system architecture orginially developed for DOS and then used in Windows. It is still in use in many small-scale peripheral devices such as USB memory sticks and digital camera memory.

file manager

A part of the **operating system** responsible for taking requests for data from a **logical file** and mapping those requests to the physical location of the data on disk.

file processing

The domain with Computer Science that deals with processing data stored on a **disk drive** (in a file), or more broadly, dealing with data stored on any **peripheral storage** device. Two fundamental properties make dealing with data on a peripheral device different from dealing with data in main memory: (1) Reading/writing data on a peripheral storage device is far slower than reading/writing data to main memory (for example, a typical disk drive is about a million times slower than **RAM**). (2) All I/O to a peripheral device is typically in terms of a **block** of data (for example, nearly all disk drives do all I/O in terms of blocks of 512 bytes).

file structure

The organization of data on **peripheral storage**, such as a **disk drive** or DVD drive.

final state

A required element of any **acceptor**. When computation on a string ends in a final state, then the machine accepts the string. Otherwise the machine rejects the string.

FIND

One half of the **UNION/FIND** algorithm for managing **disjoint sets**. It is the process of moving upwards in a tree to find the tree's root.

Finite State Acceptor

A simple type of **finite state automata**, an acceptor's only ability is to accept or reject a string. So, a finite state acceptor does not have the ability to modify the input tape. If computation on the string ends in a **final state**, then the the string is accepted, otherwise it is rejected.

Finite State Machine
FSM
Finite State Automata
FSA
Finite Automata

Any abstract state machine, generally represented as a graph where the nodes are the **states**, and the edges represent transitions between nodes that take place when the machine is in that node (state) and sees an appropriate input. See, as an example, **Deterministic Finite Automata**.

first fit

In a **memory manager**, first fit is a **heuristic** for deciding which **free block** to use when allocating memory from a **memory pool**. First fit will always allocate the first **free block** on the **free block list** that is large enough to service the memory request. The advantage of this approach is that it is typically not necessary to look at all free blocks on the free block list to find a suitable free block. The disadvantage is that it is not "intelligently" selecting what might be a better choice of free block.

fixed-length coding

Given a collection of objects, a fixed-length coding scheme assigns a code to each object in the collection using codes that are all of the same length. Standard ASCII and Unicode representations for characters are both examples of fixed-length coding schemes. This is in contrast to **variable-length coding**.

floor

Written $\lfloor x \rfloor$, for real value x the floor is the greatest integer $\leq x$.

Floyd's algorithm

An algorithm to solve the **all-pairs shortest paths problem**. It uses the **dynamic programming** algorithmic technique, and runs in $\Theta(n^3)$ time. As with any **dynamic programming** algorithm, the key issue is to avoid duplicating work by using proper bookkeeping on the algorithm's progress through the solution space. The basic idea is to first find all the direct edge costs, then improving those costs by allowing paths through **vertex** 0, then the cheapest paths involving paths going through vertices 0 and 1, and so on.

flush

The act of removing data from a **cache**, most typically because other data considered of higher future value must replace it in the cache. If the data being flushed has been modified since it was first read in from **secondary storage** (and the changes are meant to be saved), then it must be written back to that secondary storage.

flush

The the context of a **buffer pool**, the process of removing the contents stored in a **buffer** when that buffer is required in order to store new data. If the buffer's contents have been changed since having been read in from **backing storage** (this fact would normally be tracked by using a **dirty bit**), then they must be copied back to the backing storage before the buffer can be reused.

flyweight

A **design pattern** that is meant to solve the following problem: You have an application with many objects. Some of these objects are identical in the information that they contain, and the role that they play. But they must be reached from various places, and conceptually they really are distinct objects. Because there is so much duplication of the same information, we want to reduce memory cost by sharing that space. For example, in document layout, the letter "C" might be represented by an object that describes that character's strokes and bounding box. However, we do not want to create a separate "C" object everywhere in the document that a "C" appears. The solution is to allocate a single copy of the shared representation for "C" objects. Then, every place in the document that needs a "C" in a given font, size, and typeface will reference this single copy. The various instances of **references** to a specific form of "C" are called flyweights. Flyweights can also be used to implement the empty leaf nodes of the **bintree** and **PR quadtree**.

folding method

In **hashing**, an approach to implementing a **hash function**. Most typically used when the key is a string, the folding method breaks the string into pieces (perhaps each letter is a piece, or a small series of letters is a piece), converts the letter(s) to an integer value (typically by using its underlying encoding value), and summing up the pieces.

Ford and Johnson sort

A sorting algorithm that is close to the theoretical minimum number of key comparisons necessary to sort. Generally not considered practical in practice due to the fact that it is not efficient in terms of the number of records that need to be moved. It consists of first sorting pairs of nodes into winners and losers (of the pairs comparisons), then (recursively) sorting the winners of the pairs, and then finally carefully selecting the order in which the losers are added to the chain of sorted items.

forest

A collection of one or more **trees**.

free block

A block of unused space in a **memory pool**.

free block list

In a **memory manager**, the list that stores the necessary information about the current **free blocks**. Generally, this is done with some sort of **linked list**, where each node of the linked list indicates the start position and length of the free block in the **memory pool**.

free store

Space available to a program during runtime to be used for **dynamic allocation** of objects. The free store is distinct from the **runtime stack**. The free store is sometimes referred to as the **heap**, which can be confusing because **heap** more often refers to a specific data structure. Most programming languages provide functions to allocate (and maybe to deallocate) objects from the free store, such as `new` in C++ and Java.

free tree

A connected, **undirected graph** with no simple cycles. An equivalent definition is that a free tree is connected and has $|\mathbf{V}| - 1$ edges.

freelist

A simple and faster alternative to using **free store** when the objects being dynamically allocated are all of the same size (and thus are interchangeable). Typically implemented as a **linked stack**, released objects are put on the front of the freelist. When a request is made to allocate an object, the freelist is checked first and it provides the object if possible. If the freelist is empty, then a new object is allocated from **free store**.

frequency count

A **heuristic** used to maintain a **self-organizing list**. Under this heuristic, a count is maintained for every record. When a record access is made, its count is increased. If this makes its count greater than that of another record in the list, it moves up toward the front of the list accordingly so as to keep the list sorted by frequency. Analogous to the **least frequently used** heuristic for maintaining a **buffer pool**.

full binary tree theorem

This theorem states that the number of leaves in a non-empty full binary tree is one more than the number of internal nodes. Equivalently, then number of null pointers in a standard **pointer-based implementation for binary tree nodes** is one more than the number of nodes in the binary tree.

full tree

A **binary tree** is full if every **node** is either a **leaf node** or else it is an **internal node** with two non-empty **children**.

function

In mathematics, a matching between inputs (the **domain**) and outputs (the **range**). In programming, a subroutine that takes input parameters and uses them to compute and return a value. In this case, it is usually considered bad practice for a function to change any global variables (doing so is called a side effect).

garbage

In **memory management**, any memory that was previously (dynamically) allocated by the program during runtime, but which is no longer accessible since all pointers to the memory have been deleted or overwritten. In some languages, garbage can be recovered by **garbage collection**. In languages such as C and C++ that do not support garbage collection, so creating garbage is considered a **memory leak**.

garbage collection

Languages with garbage collection such Java, JavaScript, Lisp, and Scheme will periodically reclaim **garbage** and return it to **free store**.

general tree

A tree in which any given node can have any number of **children**. This is in contrast to, for example, a **binary tree** where each node has a fixed number of children (some of which might be `null`). General tree nodes tend to be harder to implement for this reason.

grammar

A formal definition for what strings make up a **language**, in terms of a set of **production rules**.

graph

A **graph** $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ consists of a set of **vertices** $\mathbf{V}$ and a set of **edges** $\mathbf{E}$, such that each edge in $\mathbf{E}$ is a connection between a pair of vertices in $\mathbf{V}$.

greedy algorithm

An algorithm that makes locally optimal choices at each step.

growth rate

In **algorithm analysis**, the rate at which the cost of the **algorithm** grows as the size of its input grows.

guess-and-test

A technique used when trying to determine the **closed-form solution** for a **summation** or **recurrence relation**. Given a hypothesis for the closed-form solution, if it is correct, then it is often relatively easy to prove that using **induction**.

guided traversal

A **tree traversal** that does not need to visit every node in the tree. An example would be a **range query** in a **BST**.

halt state

In a **finite automata**, a designated **state** which causes the machine to immediately halt when it is entered.

halted configuration

A halted configuration occurs in a **Turing machine** when the machine transitions into the **halt state**.

halting problem

The halting problem is to answer this question: Given a computer program P and an input I, will program P halt when executed on input I? This problem has been proved impossible to solve in the general case. Thus, it is an example of an **unsolveable problem**.

handle

When using a **memory manager** to store data, the **client** will pass data to be stored (the **message**) to the memory manager, and the memory manager will return to the client a handle. The handle encodes the necessary information that the memory manager can later use to recover and return the message to the client. This is typically the location and length of the message within the **memory pool**.

hanging configuration

A hanging configuration occurs in a **Turing machine** when the I/O head moves to the left from the left-most square of the tape, or when the machine goes into an infinite loop.

hard algorithm

"Hard" is traditionally defined in relation to running time, and a "hard" algorithm is defined to be an algorithm with exponential running time.

hard problem

"Hard" is traditionally defined in relation to running time, and a "hard" problem is defined to be one whose best known algorithm requires exponential running time.

harmonic series

The sum of reciprocals from 1 to n is called the Harmonic Series, and is written $\mathcal{H}_n$. This sum has a value between $\log_e n$ and $\log_e n + 1$.

hash function

In a **hash system**, the function that converts a **key** value to a position in the **hash table**. The hope is that this position in the hash table contains the record that matches the key value.

hash system

The implementation for search based on hash lookup in a **hash table**. The **search key** is processed by a **hash function**, which returns a position in a **hash table**, which hopefully is the correct position in which to find the record corresponding to the search key.

hash table

The data structure (usually an **array**) that stores data records for lookup using **hashing**.

hashing

A search method that uses a **hash function** to convert a **search key** value into a position within a **hash table**. In a properly implemented **hash system**, that position in the table will have high probability of containing the record that matches the key value. Sometimes, the hash function will return an position that does not store the desired key, due to a process called **collision**. In that case, the desired record is found through a process known as **collision resolution**.

head

The beginning of a **list**.

header node

Commonly used in implementations for a **linked list** or related structure, this **node** preceeds the first element of the list. Its purpose is to simplify the code implementation by reducing the number of special cases that must be programmed for.

heap

This term has two different meanings. Uncommonly, it is a synonym for **free store**. Most often it is used to refer to a particular data structure. This data structure is a **complete binary tree** with the requirement that every **node** has a value greater than its **children** (called a **max heap**), or else the requirement that every node has a value less than its children (called a **min heap**). Since it is a complete binary tree, a heap is nearly always implemented using an **array** rather than an explicit tree structure. To add a new value to a heap, or to remove the extreme value (the max value in a max-heap or min value in a min-heap) and update the heap, takes $\Theta(\log n)$ time in the **worst case**. However, if given all of the values in an unordered array, the values can be re-arranged to form a heap in only $\Theta(n)$ time. Due to its space and time efficiency, the heap is a popular choice for implementing a **priority queue**.

heapsort

A sorting algorithm that costs $\Theta(n \log n)$ time in the **best**, **average**, and **worst** cases. It tends to be slower than **Mergesort** and **Quicksort**. It works by building a **max heap**, and then repeatedly removing the item with maximum **key** value (moving it to the end of the heap) until all elements have been removed (and replaced at their proper location in the array).

height

The height of a tree is one more than the **depth** of the deepest **node** in the tree.

height balanced

The condition the **depths** of each **subtree** in a tree are roughly the same.

heuristic

A way to solve a problem that is not guarenteed to be optimal. While it might not be guarenteed to be optimal, it is generally expected (by the agent employing the heuristic) to provide a reasonably efficient solution.

heuristic algorithm

A type of **approximation algorithm**, that uses a **heuristic** to find a good, but not necessarily cheapest, solution to an **optimization problem**.

home position

In **hashing**, a synonym for **home slot**.

home slot

In **hashing**, this is the **slot** in the **hash table** determined for a given key by the **hash function**.

homogeneity

In a **container** class, this is the property that all objects stored in the ncontainer are of the same class. For example, if you have a list intended to store Payroll records, is it possible for the programmer to insert an integer onto the list instead?

Huffman codes

The codes given to a collection of letters (or other symbols) through the process of Huffman coding. Huffman coding uses a **Huffman coding tree** to generate the codes. The codes can be of variable length, such that the letters which are expected to appear most frequently are shorter. Huffman coding is optimal whenever the true frequencies are known, and the frequency of a letter is independent of the context of that letter in the message.

Huffman coding tree

A Huffman coding tree is a **full binary tree** that is used to represent letters (or other symbols) efficiently. Each letter is associated with a node in the tree, and is then given a **Huffman code** based on the position of the associated node. A Huffman coding tree is an example of a binary **trie**.

Huffman tree

Shorter form of the term **Huffman coding tree**.

I/O head

On a **disk drive** (or similar device), the part of the machinery that actually reads data from the disk.

image-space decomposition

A from of **key-space decomposition** where the **key space** splitting points is predetermined (typically by splitting in half). For example, a **Huffman coding tree** splits the letters being coded into those with codes that start with 0 on the left side, and those with codes that start with 1 on the right side. This regular decomposition of the key space is the basis for a **trie** data structure. An image-space decomposition is in opposition to an **object-space decomposition**.

in degree

In **graph** terminology, the in degree for a **vertex** is the number of edges directed into the vertex.

incident

In **graph** terminology, an edge connecting two vertices is said to be incident with those vertices. The two vertices are said to be **adjacent**.

index file

A file whose records consist of **key-value pairs** where the pointers are referencing the complete records stored in another file.

indexing

The process of associating a **search key** with the location of a corresponding data record. The two defining points to the concept of an index is the association of a key with a record, and the fact that the index does not actually store the record itself but rather it stores a **reference** to the record. In this way, a collection of records can be supported by multiple indices, typically a separate index for each key field in the record.

induction hypothesis

The key assumption used in a **proof by induction**, that the theorem to be proved holds for smaller instances of the theorem. The induction hypothesis is equivalent to the **recursive** call in a recursive function.

induction step

Part of a **proof by induction**. In its simplest form, this is a proof of the implication that if the theorem holds for $n-1$, then it holds for n. As an alternative, see **strong induction**.

induction variable

The variable used to parameterize the theorem being proved by induction. For example, if we seek to prove that the sum of the integers from 1 to n is $n(n+1)/2$, then n is the induction variable. An induction variable must be an integer.

information theoretic lower bound

A **lower bound** on the amount of resources needed to solve a **problem** based on the number of bits of information needed to uniquely specify the answer. Sometimes referred to as a “Shannon theoretic lower bound” due to Shannon’s work on information theory and entropy. An example is that sorting has a lower bound of $\Omega(\log_2 n!)$ because there are $n!$ possible orderings for n values. This observation alone does not make the lower bound tight, because it is possible that no algorithm could actually reach the information theory lower limit.

inherit

In **object-oriented programming**, the process by which a **subclass** gains **data members** and **methods** from a **base class**.

initial state

A synonym for **start state**.

inode

Short for “index node”. In UNIX-style file systems, specific disk **sectors** that hold indexing information to define the layout of the file system.

inorder traversal

In a **binary tree**, a **traversal** that first **recursively visits** the left **child**, then visits the **root**, and then recursively visits the right child. In a **binary search tree**, this traversal will **enumerate** the nodes in sorted order.

Insertion Sort

A sorting algorithm with $\Theta(n^2)$ **average** and **worst case** cost, and $Theta(n)$ **best case** cost. This best case cost makes it useful when we have reason to expect the input to be nearly sorted.

instance variable

In **object-oriented programming**, a synonym for **data member**.

integer function

Any function whose input is an integer and whose output is an integer. It can be proved by **diagonalization** that the set of integer functions is **uncountably infinite**.

inter-sector gap

On a disk drive, a physical gap in the data that occurs between the **sectors**. This allows the **I/O head** detect the end of the sector.

interface

An interface is a class-like structure that only contains method signatures and fields. An interface does not contain an implementation of the methods or any **data members**.

intermediate code

A step in a typical **compiler** is to transform the original high-level language into a form on which it is easier to do other stages of the process. For example, some compilers will transform the original high-level source code into **assembly code** on which it can do **code optimization**, before translating it into its final executable form.

intermediate code generation

A phase in a **compiler**, that walks through a **parse tree** to produce simple **assembly code**.

internal fragmentation

A condition that occurs when more than m bytes are allocated to service a **memory request** for m bytes, wasting free storage. This is often done to simplify **memory management**.

internal node

In a tree, any node that has at least one non-empty **child** is an internal node.

internal sort

A sorting algorithm that is applied to data stored in **main memory**. This is in contrast to an **external sort** that is meant to work on data stored in **peripheral storage** such as on a **disk drive**.

interpolation

The act of finding the coefficients of a polynomial, given the values at some points. A polynomal of degree $n-1$ requires n points to interpolate the coefficients.

interpolation search

Given a sorted array, and knowing the first and last **key** values stored in some subarray known to contain **search key** K, interpolation search will compute the expected location of K in the subarray as a fraction of the distance between the known key values. So it will next check that computed location, thus narrowing the search for the next iteration. Given reasonable key value distribution, the **average case** for interpolation search will be $\Theta(\log \log n)$, or better than the expected cost of **binary search**. Nonetheless, binary search is expected to be faster in nearly all practical situations due to the small difference between the two costs, combined with the higher constant factors required to implement interpolation search as compared to binary search.

interpreter

In contrast to a **compiler** that translates a high-level program into something that can be repeatedly executed to perform a computation, an interpreter directly performs computation on the high-level langauge. This tends to make the computation much slower than if it were performed on the directly executable version produced by a compiler.

inversion

A measure of how disordered a series of values is. For each element X in the series, count one inversion for each element to left of X that is greater than the value of X (and so must ultimately be moved to the right of X during a sorting process).

inverted file

Synonym for **inverted list** when the inverted list is stored in a disk file.

inverted list

An **index** which links **secondary keys** to either the associated **primary key** or the actual record in the database.

irreflexive

In set notation, binary relation R on set S is irreflexive if aRa is never in the relation for any $a \in \mathbf{S}$.

ISAM

Indexed Sequential Access Method: an obsolete method for indexing data for (at the time) fast retrieval. More generally, the term is used also to generically refer to an **index** that supports both sequential and **keyed** access to data records. Today, that would nearly always be implemented using a **B-Tree**.

iterator

In a **container** such as a List, a separate class that indicates position within the container, with support for **traversing** through all **elements** in the container.

job

Common name for processes or tasks to be run by an operating system. They typically need to be processed in order of importance, and so are kept organized by a **priority queue**. Another common use for this term is for a collection of tasks to be ordered by a **topological sort**.

jump search

An algorithm for searching a sorted list, that falls between **sequential search** and **binary search** in both computational cost and conceptual complexity. The idea is to keep jumping by some fixed number of positions until a value is found that is bigger than **search key** K, then do a sequential search over the subarray that is now known to contain the search key. The optimal number of steps to jump will be $\sqrt{n}$ for an array of size n, and the **worst case** cost will be $\Theta(\sqrt{n})$.

K-ary tree

A type of **full tree** where every internal node has exactly K **children**.

k-path

In **Floyd's algorithm**, a k-path is a path between two vertices i and j that can only go through vertices with an index value less than or equal to k.

kd tree

A **spatial data structure** that uses a binary tree to store a collection of data records based on their (point) location in space. It uses the concept of a **discriminator** at each level to decide which single component of the **multi-dimensional search key** to branch on at that level. It uses a **key-space decomposition**, meaning that all data records in the left subtree of a node have a value on the corresponding discriminator that is less than that of the node, while all data records in the right subtree have a greater value. The **bintree** is the **image-space decomposition** analog of the kd tree.

key

A field or part of a larger record used to represent that record for the purpose of searching or comparing. Another term for **search key**.

key sort

Any sorting operation applied to a collection of **key-value pairs** where the value in this case is a **reference** to a complete record (that is, a pointer to the record in memory or a position for a record on disk). This is in contrast to a sorting operation that works directly on a collection of records. The intention is that the collection of key-value pairs is far smaller than the collection of records themselves. As such, this might allow for an **internal sort** when sorting the records directly would require an **external sort**. The collection of key-value pairs can also act as an **index**.

key space

The range of values that a **key** value may take on.

key-space decomposition

The idea that the range for a **search key** will be split into pieces. There are two general approaches to this: **object-space decomposition** and **image-space decomposition**.

key-value pair

A standard solution for solving the problem of how to relate a **key** value to a record (or how to find the key for a given record) within the context of a particular **index**. The idea is to simply store as records in the index pairs of keys and records. Specifically, the index will typically store a copy of the key along with a **reference** to the record. The other standard solution to this problem is to pass a **comparator** function to the index.

knapsack problem

While there are many variations of this problem, here is a typical version: Given knapsack of a fixed size, and a collection of objects of various sizes, is there a subset of the objects that exactly fits into the knapsack? This problem is known to be **NP-complete**, but can be solved for problem instances in practical time relatively quickly using **dynamic programming**. Thus, it is considered to have **pseudo-polynomial** cost. An **optimization problem** version is to find the subset that can fit with the greatest amount of items, either in terms of their total size, or in terms of the sum of values associated with each item.

Kruskal's algorithm

An algorithm for computing the **MCST** of a **graph**. During processing, it makes use of the **UNION/FIND** process to efficiently determine of two vertices are within the same **subgraph**.

labeled graph

A **graph** with labels associated with the **nodes**.

language

A set of strings.

Las Vegas algorithms

A form of **randomized algorithm**. We always find the maximum value, and "usually" we find it fast. Such algorithms have a guaranteed result, but do not guarantee fast running time.

leaf node

In a **binary tree**, leaf node is any node that has two empty **children**. (Note that a binary tree is defined so that every node has two children, and that is why the leaf node has to have two empty children, rather than no children.) In a general tree, any node is a leaf node if it has no children.

least frequently used

Abbreviated **LFU**, it is a **heuristic** that can be used to decide which **buffer** in a **buffer pool** to **flush** when data in the buffer pool must be replaced by new data being read into a **cache**. However, **least recently used** is more popular than LFU. Analogous to the **frequency count** heuristic for maintaining a **self-organizing list**.

least recently used

Abbreviated **LRU**, it is a popular **heuristic** to use for deciding which **buffer** in a **buffer pool** to **flush** when data in the buffer pool must be replaced by new data being read into a **cache**. Analogous to the **move-to-front** heuristic for maintaining a **self-organizing list**.

left recursive

In automata theory, a **production** is left recursive if it is of the form $A \rightarrow Ax$, $A \in V, x \in (V \cup T)^*$ where V is the set of **non-terminals** and T is the set of **terminals** in the **grammar**.

length

In a **list**, the number of elements. In a string, the number of characters.

level

In a tree, all nodes of **depth** d are at level d in the tree. The root is the only node at level 0, and its depth is 0.

lexical analysis

A phase of a **compiler** or **interpreter** responsible for reading in characters of the program or language and grouping them into **tokens**.

lexical scoping

Within programming languages, the convention of allowing access to a variable only within the block of code in which the variable is defined. A synonym for **static scoping**.

LFU

Abbreviation for **least frequently used**.

lifetime

For a variable, lifetime is the amount of time it will exist before it is destroyed.

LIFO

Abbreviation for “Last-In, First-Out”. This is the access paradigm for a **stack**, and an old terminolgy for the stack is “LIFO list”.

linear congruential method

In random number theory, a process for computing the next number in a **pseudo-random** sequence. Starting from a **seed**, the next term $r(i)$ in the series is calculated from term $r(i-1)$ by the equation

$$r(i) = (r(i-1) \times b) \bmod t$$

where b and t are constants. These constants must be well chosen for the resulting series of numbers to have desireable properties as a random number sequence.

linear growth rate

For input size n, a growth rate of cn (for c any positive constant). In other words, the cost of the associated function is linear on the input size.

linear index

A form of **indexing** that stores **key-value pairs** in a sorted array. Typically this is used for an index to a large collection of records stored on disk, where the linear index itself might be on disk or in **main memory**. It allows for efficient search (including for **range queries**), but it is not good for inserting and deleting entries in the array. Therefore, it is an ideal indexing structure when the system needs to do range queries but the collection of records never changes once the linear index has been created.

linear order

Another term for **total order**.

linear probing

In **hashing**, this is the simplest **collision resolution** method. Term i of the **probe sequence** is simply i, meaning that collision resolution works by moving sequentially through the hash table from the **home slot**. While simple, it is also inefficient, since it quickly leads to certain free **slots** in the hash table having higher probability of being selected during insertion or search.

linear probing by steps

In **hashing**, this **collision resolution** method is a variation on simple **linear probing**. Some constant c is defined such that term i of the **probe sequence** is ci. This means that collision resolution works by moving sequentially through the hash table from the **home slot** in steps of size c. While not much improvement on linear probing, it forms the basis of another collision resolution method called **double hashing**, where each key uses a value for c defined by a second **hash function**.

linear search

Another name for **sequential search**.

link node

A widely used supporting object that forms the basic building block for a **linked list** and similar **data structures**. A link node contains one or more fields that store data, and a **pointer** or **reference** to another link node.

linked list

An implementation for the list ADT that uses **dynamic allocation** of **link nodes** to store the list elements. Common variants are the **singly linked list**, **doubly linked list** and **circular list**. The **overhead** required is the pointers in each link node.

linked stack

Analogous to a **linked list**, this uses **dynamic allocation** of nodes to store the elements when implementing the stack ADT.

list

A finite, ordered sequence of **data items** known as **elements**. This is close to the mathematical concept of a **sequence**. Note that “ordered” in this definition means that the list elements have position. It does not refer to the relationship between **key** values for the list elements (that is, “ordered” does not mean “sorted”).

literal

In a **Boolean expression**, a **literal** is a **Boolean variable** or its negation. In the context of compilers, it is any constant value. Similar to a **terminal**.

load factor

In **hashing** this is the fraction of the **hash table slots** that contain a record. Hash systems usually try to keep the load factor below 50%.

local storage

local storage.

local variable

A variable declared within a function or method. It exists only from the time when the function is called to when the function exits. When a function is suspended (due to calling another function), the function’s local variables are stored in an **activation record** on the **runtime stack**.

locality of reference

The concept that accesses within a collection of records is not evenly distributed. This can express itself as some small fraction of the records receiving the bulk of the accesses (**80/20 rule**). Alternatively, it can express itself as an increased probability that the next or future accesses will come close to the most recent access. This is the fundamental property for success of **caching**.

logarithm

The *logarithm* of base b for value y is the power to which b is raised to get y.

logical file

In **file processing**, the programmer’s view of a **random access** file stored on **disk** as a contiguous series of bytes, with those bytes possibly combining to form data records. This is in contrast to the **physical file**.

logical form

The definition for a data type in terms of an ADT. Contrast to the **physical form** for the data type.

lookup table

A table of pre-calculated values, used to speed up processing time when the values are going to be viewed many times. The costs to this approach are the space required for the table and the time required to compute the table. This is an example of a **space/time tradeoff**.

lower bound

In **algorithm analysis**, a **growth rate** that is always less than or equal to the growth rate of the **algorithm** in question. In practice, this is the fastest-growing function that we know grows no faster than all but a constant number of inputs. It could be a gross under-estimate of the truth. Since the lower bound for the algorithm can be very different for different situations (such as the **best case** or **worst case**), we typically have to specify which situation we are referring to.

lower bounds proof

A proof regarding the lower bound, with this term most typically referring to the lower bound for any possible algorithm to solve a given **problem**. Many problems have a simple lower bound based on the concept that the minimum amount of processing is related to looking at all of the problem’s input. However, some problems have a higher lower bound than that. For example, the lower bound for the problem of sorting ($\Omega(n \log n)$) is greater than the input size to sorting (n). Proving such “non-trivial” lower bounds for problems is notoriously difficult.

LRU

Abbreviation for **least recently used**.

main memory

A synonym for **primary storage**. In a computer, typically this will be **RAM**.

map

A **data structure** that relates a **key** to a **record**.

mapping

A **function** that maps every element of a given **set** to a unique element of another set; a correspondence.

mark array

It is typical in **graph** algorithms that there is a need to track which nodes have been visited at some point in the algorithm. An **array** of bits or values called the **mark array** is often maintained for this purpose.

mark/sweep algorithm

An algorithm for **garbage collection**. All accessible variables, and any space that is reachable by a chain of pointers from any accessible variable, is “marked”. Then a sequential sweep of all memory in the pool is made. Any unmarked memory locations are assumed to not be needed by the program and can be considered as free to be reused.

master theorem

A theorem that makes it easy to solve **divide-and-conquer recurrences**.

matching

In graph theory, a pairing (or match) of various nodes in a graph.

matching problem

Any problem that involves finding a **matching** in a graph with some desired property. For example, a well-known **NP-complete** problem is to find a **maximum match** for an undirected graph.

max heap

A **heap** where every **node** has a **key** value greater than its **children**. As a consequence, the node with maximum key value is at the **root**.

maximal match

In a graph, any **matching** that leaves no pair of unmatched vertices that are connected. A maximal matching is not necessarily a **maximum match**. In other words, there might be a larger matching than the maximal matching that was found.

maximum lower bound

The **lower bound** for the **problem** of finding the maximum value in an unsorted list is $\Omega(n)$.

maximum match

In a graph, the largest possible **matching**.

MCST

MST

Abbreviation for **minimal-cost spanning tree**.

measure of cost

When comparing two things, such as two algorithms, some event or unit must be used as the basic unit of comparison. It might be number of milliseconds needed or machine instructions expended by a program, but it is usually desirable to have a way to do comparison between two algorithms without writing a program. Thus, some other measure of cost might be used as a basis for comparison between the algorithms. For example, when comparing two sorting algorithms it is traditional to use as a measure of cost the number of **comparisons** made between the key values of record pairs.

member

In set notation, this is a synonym for **element**. In abstract design, a **data item** is a member of a **type**. In an object-oriented language, **data members** are data fields in an object.

member function

Each operation associated with the ADT is implemented by a member function or **method**.

memory allocation

In a **memory manager**, the act of honoring a request for memory.

memory deallocation

In a **memory manager**, the act of freeing a block of memory, which should create or add to a **free block**.

memory hierarchy

The concept that a computer system stores data in a range of storage types that range from fast but expensive (**primary storage**) to slow but cheap (**secondary storage**). When there is too much data to store in **primary storage**, the goal is to have the data that is needed soon or most often in the primary storage as much as possible, by using **caching** techniques.

memory leak

In programming, the act of creating **garbage**. In languages such as C and C++ that do not support **garbage collection**, repeated memory leaks will evenually cause the program to terminate.

memory manager

Functionality for managing a **memory pool**. Typically, the memory pool is viewed as an **array** of bytes by the memory manager. The **client** of the memory manager will request a collection of (adjacent) bytes of some size, and release the bytes for reuse when the space is no longer needed. The memory manager should not know anything about the interpretation of the data that is being stored by the client into the memory pool. Depending on the precise implementation, the client might pass in the data to be stored, in which case the memory manager will deal with the actual copy of the data into the memory pool. The memory manager will return to the client a **handle** that can later be used by the client to retrieve the data.

memory pool

Memory (usually in **RAM** but possibly on disk or **peripheral storage** device) that is logically viewed as an array of memory positions. A memory pool is usually managed by a **memory manager**.

memory request

In a **memory manager**, a request from some **client** to the memory manager to reserve a block of memory and store some bytes there.

merge insert sort

A synonym for the **Ford and Johnson sort**.

Mergesort

A sorting algorithm that requires $\Theta(n \log n)$ in the **best**, **average**, and **worst** cases. Conceptually it is simple: Split the list in half, sort the halves, then merge them together. It is a bit complicated to implement efficiently on an **array**.

message

In a **memory manager** implementation (particularly a memory manager implemented with a **message passing** style of **interface**), the message is the data that the **client** of the memory manager wishes to have stored in the **memory pool**. The memory manager will reply to the client by returning a **handle** that defines the location and size of the message as stored in the memory pool. The client can later recover the message by passing the handle back to the memory manager.

message passing

A common approach to implementing the **ADT** for a **memory manager** or **buffer pool**, where the contents of a **message** to be stored is explicitly passed between the client and the memory manager. This is in contrast to a **buffer passing** approach.

metaphor

Humans deal with complexity by assigning a label to an assembly of objects or concepts and then manipulating the label in place of the assembly. Cognitive psychologists call such a label a metaphor.

method

In the **object-oriented programming paradigm**, a method is an operation on a **class**. A synonym for **member function**.

mid-square method

In **hashing**, an approach to implementing a **hash function**. The key value is squared, and some number of bits from the middle of the resulting value are extracted as the hash code. Some care must be taken to extract bits that tend to actually be in the middle of the resulting value, which requires some understanding of the typical key values. When done correctly, this has the advantage of having the hash code be affected by all bits of the key

min heap

A **heap** where every **node** has a **key** value less than its **children**. As a consequence, the node with minimum key value is at the **root**.

minimal-cost spanning tree

Abbreviated as MCST, or sometimes as MST. Derived from a **weighted graph**, the MCST is the **subset** of the graph's **edges** that maintains the connectivitiy of the graph while having lowest total cost (as defined by the sum of the **weights** of the edges in the MCST). The result is referred to as a **tree** because it would never have a **cycle** (since an edge could be removed from the cycle and still preserve connectivity). Two algorithms to solve this problem are **Prim's algorithm** and **Kruskal's algorithm**.

minimum external path weight

Given a collection of objects, each associated with a **leaf node** in a tree, the binary tree with minimum external path weight is the one with the minimum sum of **weighted path lengths** for the given set of leaves. This concept is used to create a **Huffman coding tree**, where a letter with high weight should have low depth, so that it will count the least against the total path length. As a result, another letter might be pushed deeper in the tree if it has less weight.

mod

Abbreviation for the **modulus** function.

model

A simplification of reality that preserves only the essential elements. With a model, we can more easily focus on and reason about these essentials. In **algorithm analysis**, we are especially concerned with the **cost model** for measuring the cost of an algorithm.

modulus

The modulus function returns the remainder of an integer division. Sometimes written $n \bmod m$ in mathematical expressions, the syntax in many programming languages is `n % m`.

Monte Carlo algorithms

A form of **randomized algorithm**. We find the maximum value fast, or we don't get an answer at all (but fast). While such algorithms have good running time, their result is not guaranteed.

move-to-front

A **heuristic** used to maintain a **self-organizing list**. Under this heuristic, whenever a record is accessed it is moved to the front of the list. Analogous to the **least recently used** heuristic for maintaining a **buffer pool**.

multi-dimensional search key

A search key containing multiple parts, that works in conjunction with a **multi-dimensional search structure**. Most typically, a **spatial** search key representing a position in multi-dimensional (2 or 3 dimensions) space. But a multi-dimensional key could be used to organize data within non-spatial dimensions, such as temperature and time.

multi-dimensional search structure

A data structure used to support efficient search on a **multi-dimensional search key**. The main concept here is that a multi-dimensional search structure works more efficiently by considering the multiple parts of the search key as a whole, rather than making independent searches on each one-dimensional component of the key. A primary example is a **spatial data structure** that can efficiently represent and search for records in multi-dimensional space.

multilist

A list that may contain sublists. This term is sometimes used as a synonym to the term **bag**.

natural numbers

Zero and the positive integers.

necessary fallacy

A common mistake in a **lower bounds proof** for a problem, where the proof makes an inappropriate assumption that any algorithm must operate in some manner (typically in the way that some known algorithm behaves).

neighbor

In a **graph**, a **node** w is said to be a neighbor of **node** v if there is an **edge** from v to w.

node

The objects that make up a linked structure such as a linked list or binary tree. Typically, nodes are allocated using **dynamic memory allocation**. In **graph** terminology, the nodes are more commonly called **vertices**.

non-deterministic

In a **finite automata**, at least one **state** has multiple transitions on at least one symbol. This means that it is not **deterministic** about what transition to take in that situation. A non-deterministic machine is said to **accept** a string if it completes execution on the string in an **accepting state** under at least one choice of non-deterministic transitions. Generally, non-determinism can be simulated with a deterministic machine by alternating between the execution that would take place under each of the branching choices.

non-deterministic algorithm

An algorithm that may operate using a **non-deterministic choice** operation.

non-deterministic choice

An operation that captures the concept of nondeterminism. A nondeterministic choice can be viewed as either "correctly guessing" between a set of choices, or implementing each of the choices in parallel. In the parallel view, the nondeterminism was successful if at least one of the choices leads to a correct answer.

non-deterministic polynomial time algorithm

An algorithm that runs in polynomial time, and which may (or might not) use **non-deterministic choice**.

non-strict partial order

In set notation, a relation that is **reflexive**, **antisymmetric**, and **transitive**.

non-terminal

In contrast to a **terminal**, a non-terminal is an abstract state in a **production rule**. Begining with the **start symbol**, all non-terminals must be converted into terminals in order to complete a **derivation**.

NP

An abbreviation for **non-deterministic polynomial**.

NP-Complete

A class of problems that are related to each other in this way: If ever one such problem is proved to be solvable in polynomial time, or proved to require exponential time, then all other NP-Complete problems will cost likewise. Since so many real-world problems have been proved to be NP-Complete, it would be extremely useful to determine if they have polynomial or exponential cost. But so far, nobody has been able to determine the truth of the situation. A more technical definition is that a problem is NP-Complete if it is in NP and is NP-hard.

NP-Completeness proof

A type of **reduction** used to demonstrate that a particular **problem** is **NP-complete**. Specifically, an NP-Completeness proof must first show that the problem is in class **NP**, and then show (by using a reduction to another NP-Complete problem) that the problem is **NP-hard**.

NP-hard

A problem that is "as hard as" any other problem in **NP**. That is, Problem X is NP-hard if any algorithm in NP can be **reduced** to X in polynomial time.

nth roots of unity

All of the points along the unit circle in the complex plane that represent multiples of the **primitive nth root of unity**.

object

An instance of a **class**, that is, something that is created and takes up storage during the execution of a computer program. In the **object-oriented programming paradigm**, objects are the basic units of operation. Objects have state in the form of **data members**, and they know how to perform certain actions (**methods**).

object-oriented programming paradigm

An approach to problem-solving where all computations are carried out using **objects**.

object-space decomposition

A from of **key-space decomposition** where the **key space** is determined by the actual values of keys that are found. For example, a **BST** stores a key value in its root, and all other values in the tree with lesser value are in the left

subtree. Thus, the root value has split (or decomposed) the key space for that key based on its value into left and right parts. An object-space decomposition is in opposition to an **image-space decomposition**.

octree

The three-dimensional equivalent of the **quadtree** would be a tree with 2^3 or eight branches.

Omega notation

In **algorithm analysis**, Ω notation is used to describe a **lower bound**. Roughly (but not completely) analogous to **big-Oh notation** used to define an **upper bound**.

one-way list

A synonym for a **singly linked list**.

open addressing

A synonym for **closed hashing**.

open hash system

A **hash system** where multiple records might be associated with the same slot of a **hash table**. Typically this is done using a linked list to store the records. This is in contrast to a **closed hash system**.

operating system

The control program for a computer. Its purpose is to control hardware, manage resources, and present a standard interface to these to other software components.

optimal static ordering

A theoretical construct defining the best static (non-changing) order in which to place a collection of records so as to minimize the number of records **visited** by a series of sequential searches. It is a useful concept for the purpose of defining a theoretical optimum against which to compare the performance for a **self-organizing list heuristic**.

optimization problem

Any problem where there are a (typically large) collection of potential solutions, and the goal is to find the best solution. An example is the Traveling Salesman Problem, where visiting n cities in some order has a cost, and the goal is to visit in the cheapest order.

out degree

In **graph** terminology, the out degree for a **vertex** is the number of edges directed out of the vertex.

overflow

The condition where the amount of data stored in an entity has exceeded its capacity. For example, a node in a **B-tree** can store a certain number of records. If a record is attempted to be inserted into a node that is full, then something has to be done to handle this case.

overflow bucket

In **bucket hashing**, this is the **bucket** into which a record is placed if the bucket containing the record's **home slot** is full. The overflow bucket is logically considered to have infinite capacity, though in practice search and insert will become relatively expensive if many records are stored in the overflow bucket.

overhead

All information stored by a data structure aside from the actual data. For example, the pointer fields in a **linked list** or **BST**, or the unused positions in an **array-based list**.

page

A term often used to refer to the contents of a single **buffer** within a **buffer pool** or other **virtual memory**. This corresponds to a single **block** or **sector** of data from **backing storage**, which is the fundamental unit of I/O.

parameter

The values making up an input to a **function**.

parent

In a tree, the **node** P that directly links to a node A is the parent of A. A is the **child** of P.

parent pointer representation

For **trees**, a **node** implementation where each node stores only a pointer to its **parent**, rather than to its **children**. This makes it easy to go up the tree toward the **root**, but not down the tree toward the **leaves**. This is most appropriate for solving the **UNION/FIND** problem.

parity

The concept of matching even-ness or odd-ness, the basic idea behind using a **parity bit** for error detection.

parity bit

A common method for checking if transmission of a sequence of bits has been performed correctly. The idea is to count the number of 1 bits in the sequence, and set the parity bit to 1 if this number is odd, and 0 if it is even. Then, the transmitted sequence of bits can be checked to see if its parity matches the value of the parity bit. This will catch certain types of errors, in particular if the value for a single bit has been reversed. This was used, for example, in early versions of **ASCII character coding**.

parse tree

A tree that represents the syntactic structure of an input string, making it easy to compare against a **grammar** to see if it is syntactically correct.

parser

A part of a **compiler** that takes as input the program text (or more typically, the tokens from the **scanner**), and verifies that the program is syntactically correct. Typically it will build a **parse tree** as part of the process.

partial order

In set notation, a binary relation is called a partial order if it is **antisymmetric** and **transitive**. If the relation is also **reflexive**, then it is a **non-strict partial order**. Alternatively, if the relation is also **irreflexive**, then it is a **strict partial order**.

partially ordered set

The set on which a **partial order** is defined is called a partially ordered set.

partition

In **Quicksort**, the process of splitting a list into two sublists, such that one sublist has values less than the **pivot** value, and the other with values greater than the pivot. This process takes $\Theta(i)$ time on a sublist of length i.

pass by reference

A **reference** to the variable is passed to the called function. So, any modifications will affect the original variable.

pass by value

A copy of a variable is passed to the called function. So, any modifications will not affect the original variable.

path

In **tree** or **graph** terminology, a sequence of **vertices** $v_1, v_2, \ldots, v_n$ forms a path of length $n-1$ if there exist edges from v_i to v_{i+1} for $1 \leq i < n$.

path compression

When implementing the **UNION/FIND** algorithm, path compression is a local optimization step that can be performed during the FIND step. Once the root of the tree for the current object has been found, the path to the root can be traced a second time, with all objects in the tree made to point directly to the root. This reduces the depth of the tree from typically $\Theta(\log n)$ to nearly constant.

peripheral storage

Any storage device that is not part of the core processing of the computer (that is, **RAM**). A typical example is a **disk drive**.

permutation

A permutation of a sequence $\mathbf{S}$ is the **elements** of $\mathbf{S}$ arranged in some order.

persistent

In the context of computer memory, this refers to a memory that does not lose its stored information when the power is turned off.

physical file

The collection of sectors that comprise a file on a **disk drive**. This is in contrast to the **logical file**.

physical form

The implementation of a data type as a data structure. Contrast to the **physical form** for the data type.

Pigeonhole Principle

A commonly used lemma in Mathematics. A typical variant states: When $n+1$ objects are stored in n locations, at least one of the locations must store two or more of the objects.

pivot

In **Quicksort**, the value that is used to split the list into sublists, one with lesser values than the pivot, the other with greater values than the pivot.

platter

In a **disk drive**, one of a series of flat disks that comprise the storage space for the drive. Typically, each surface (top and bottom) of each platter stores data, and each surface has its own **I/O head**.

point quadtree

A **spatial data structure** for storing point data. It is similar to a **PR quadtree** in that it (in two dimensions) splits the world into four parts. However, it splits using an **object-space decomposition**. That is, quadrant containing the point is split into four parts at the point. It is similar to the **kd tree** which splits alternately in each dimension, except that it splits in all dimensions at once.

point-region quadtree

Formal name for what is commonly referred to as a **PR quadtree**.

pointee

The term pointee refers to anything that is pointed to by a **pointer** or **reference**.

pointer

A variable whose value is the **address** of another variable; a link.

pointer-based implementation for binary tree nodes

A common way to implement **binary tree nodes**. Each node stores a data value (or a **reference** to a data value), and pointers to the left and right children. If either or both of the children does not exist, then a null pointer is stored.

polymorphism

An **object-oriented programming** term meaning *one name, many forms*. It describes the ability of software to change its behavior dynamically. Two basic forms exist: **run-time polymorphism** and **compile-time polymorphism**.

pop

A specialized term used to indicate removing an **element** from a **stack**.

poset

Another name for a **partially ordered set**.

position

The defining property of the list ADT, this is the concept that list elements are in a position. Many list ADTs support access by position.

postorder traversal

In a **binary tree**, a **traversal** that first **recursively visits** the left **child**, then recursively visits the right child, and then visits the **root**.

potential

A concept related to **amortized analysis**. Potential is the total or currently available amount of work that can be done.

powerset

For a **set** $\mathbf{S}$, the power set is the set of all possible **subsets** for $\mathbf{S}$.

PR quadtree

A type of **quadtree** that stores point data in two dimensions. The root of the PR quadtree represents some square region of 2d space. If that space stores more than one data point, then the region is decomposed into four equal subquadrants, each represented **recursively** by a subtree of the PR quadtree. Since many leaf nodes of the PR quadtree will contain no data points, implementation often makes use of the **Flyweight design pattern**. Related to the **bintree**.

prefix property

Given a collection of strings, the collection has the prefix property if no string in the collection is a prefix for another string in the collection. The significance is that, given a long string composed of members of the collection, it can be uniquely decomposed into the constituent members. An example of such a collection of strings with the prefix property is a set of **Huffman codes**.

preorder traversal

In a **binary tree**, a **traversal** that first **visits** the **root**, then **recursively** visits the left **child**, then recursively visits the right child.

Prim's algorithm

A **greedy algorithm** for computing the **MCST** of a **graph**. It is nearly identical to **Dijkstra's algorithm** for solving the **single-source shortest paths problem**, with the only difference being the calculation done to update the best-known distance.

primary clustering

In **hashing**, the tendency in certain **collision resolution** methods to create clustering in sections of the hash table. The classic example is **linear probing**. This tends to happen when a group of keys follow the same **probe sequence** during collision resolution.

primary index

Synonym for **primary key index**.

primary key

A unique identifier for a **record**.

primary key index

Relates each **primary key** value with a pointer to the actual record on disk.

primary storage

The faster but more expensive memory in a computer, most often **RAM** in modern computers. This is in contrast to **secondary storage**, which together with primary storage devices make up the computer's **memory hierarchy**.

primitive data type

In Java, one of a particular group of **simple types** that are not implemented as objects. An example is an `int`.

primitive element

In set notation, this is a single element that is a member of the base type for the set. This is as opposed to an element of the set being another set.

primitive nth root of unity

The n th root of 1. Normally a **complex number**. An intuitive way to view this is one n th of the unit circle in the complex plain.

priority

A quantity assigned to each of a collection of **jobs** or tasks that indicate importance for order of processing. For example, in an operating system, there could be a collection of processes (jobs) ready to run. The operating system must select the next task to execute, based on their priorities.

priority queue

An ADT whose primary operations of insert of records, and deletion of the greatest (or, in an alternative implementation, the least) valued record. Most often implemented using the **heap** data structure. The name comes from a common application where the records being stored represent tasks, with the ordering values based on the **priorities** of the tasks.

probabilistic algorithm

A form of **randomized algorithm** that might yield an incorrect result, or that might fail to produce a result.

probabilistic data structure

Any data structure that uses **probabilistic algorithms** to perform its operations. A good example is the **skip list**.

probe function

In **hashing**, the function used by a **collision resolution** method to calculate where to look next in the **hash table**.

probe sequence

In **hashing**, the series of **slots** visited by the **probe function** during **collision resolution**.

problem

A task to be performed. It is best thought of as a **function** or a mapping of inputs to outputs.

problem instance

A specific selection of values for the parameters to a problem. In other words, a specific set of inputs to a problem. A given problem instance has a size under some **cost model**.

problem lower bound

In **algorithm analysis**, the tightest **lower bound** that we can prove over all **algorithms** for that **problem**. This is often much harder to determine than the **problem upper bound**. Since the lower bound for the algorithm can be very different for different situations (such as the **best case** or **worst case**), we typically have to specify which situation we are referring to.

problem upper bound

In **algorithm analysis**, the **upper bound** for the best **algorithm** that we know for the **problem**. Since the upper bound for the algorithm can be very different for different situations (such as the **best case** or **worst case**), we typically have to specify which situation we are referring to.

procedural

Typically referring to the **procedural programming paradigm**, in contrast to the **object-oriented programming paradigm**.

procedural programming paradigm

Procedural programming uses a list of instructions (and procedure calls) that define a series of computational steps to be carried out. This is in contrast to the **object-oriented programming paradigm**.

production
production rule

A **grammar** is comprised of production rules. The production rules consist of **terminals** and **non-terminals**, with one of the non-terminals being the **start symbol**. Each production rule replaces one or more non-terminals (perhaps with associated terminals) with one or more terminals and non-terminals. Depending on the restrictions placed on the form of the rules, there are classes of languages that can be represented by specific types of grammars. A **derivation** is a series of productions that results in a string (that is, all non-terminals), and this derivation can be represented as a **parse tree**.

program

An instance, or concrete representation, of an algorithm in some programming language.

promotion

In the context of certain **balanced tree** structures such as the **2-3 tree**, a promotion takes place when an insertion causes the node to **overflow**. In the case of the 2-3 tree, the **key** with the middlemost value is sent to be stored in the parent.

proof

The establishment of the truth of anything, a demonstration.

proof by contradiction

A mathematical proof technique that proves a theorem by first assuming that the theorem is false, and then uses a chain of reasoning to reach a logical contradiction. Since when the theorem is false a logical contradiction arises, the conclusion is that the theorem must be true.

proof by induction

A mathematical proof technique similar to **recursion**. It is used to prove a parameterized theorem $S(n)$, that is, a theorem where there is a **induction variable** involved (such as the sum of the numbers from 1 to n). One first proves that the theorem holds true for a **base case**, then one proves the implication that whenever $S(n)$ is true then $S(n+1)$ is also true. Another variation is **strong induction**.

proving the contrapositive

We can prove that $P \Rightarrow Q$ by proving $(\text{not } Q) \Rightarrow (\text{not } P)$.

pseudo polynomial

In complexity analysis, refers to the time requirements of an algorithm for an **NP-Complete** problem that still runs acceptably fast for practical application. An example is the standard **dynamic programming** algorithm for the **knapsack problem**.

pseudo random

In random number theory this means that, given all past terms in the series, no future term of the series can be accurately predicted in polynomial time.

pseudo-random probing

In **hashing**, this is a **collision resolution** method that stores a random permutation of the values 1 through the size of the **hash table**. Term i of the **probe sequence** is simply the value of position i in the permuation.

push

A specialized term used to indicate inserting an **element** onto a **stack**.

pushdown automata
PDA

A type of **Finite State Automata** that adds a stack memory to the basic **Deterministic Finite Automata** machine. This extends the set of languages that can be recognize to the **context-free languages**.

quadratic growth rate

A growth rate function of the form cn^2 where n is the input size and c is a constant.

quadratic probing

In **hashing**, this is a **collision resolution** method that computes term i of the **probe sequence** using some quadratic equation $ai_b^2i + c$ for suitable constants a, b, c. The simplest form is simply to use i^2 as term i of the probe sequence.

quadtree

A **full tree** where each internal node has four children. Most typically used to store two dimensional **spatial data**. Related to the **bintree**. The difference is that the quadtree splits all dimensions simultaneously, while the bintree splits one dimension at each level. Thus, to extend the quadtree concept to more dimensions requires a rapid increase in the number of splits (for example, 8 in three dimensions).

queue

A list-like structure in which elements are inserted only at one end, and removed only from the other one end.

Quicksort

A sort that is $\Theta(n \log n)$ in the **best** and **average** cases, though $\Theta(n^2)$ in the **worst case**. However, a reasonable implmentation will make the worst case occur under exceedingly rare circumstances. Due to its tight inner loop, it tends to run better than any other known sort in general cases. Thus, it is a popular sort to use in code libraries. It works by divide and conquer, by selecting a **pivot** value, splitting the list into parts that are either less than or greater than the pivot, and then sorting the two parts.

radix

Synonym for **base**. The number of digits in a number representation. For example, we typically represent numbers in base (or radix) 10. Hexidecimal is base (or radix) 16.

radix sort

A sorting algorithm that works by processing records with k digit keys in k passes, where each pass sorts the records according to the current digit. At the end of the process, the records will be sorted. This can be efficient if the number of digits is small compared to the number of records. However, if the n records all have unique key values, than at least $\Omega(\log n)$ digits are required, leading to an $\Omega(n \log n)$ sorting algorithm that tends to be much slower than other sorting algorithms like **Quicksort** or **mergesort**.

RAM

Abbreviation for **Random Access Memory**.

random access

In **file processing** terminology, a **disk access** to a random position within the file. More generally, the ability to access an arbitrary record in the file.

random access memory

Abbreviated **RAM**, this is the principle example of **primary storage** in a modern computer. Data access times are typically measured in billionths of a second (microseconds), which is roughly a million times faster than data access from a disk drive. RAM is where data are held for immediate processing, since access times are so much faster than for **secondary storage**. RAM is a typical part of a computer's **memory hierarchy**.

random permutation

One of the $n!$ possible permutations for a set of n element is selected in such a way that each permutation has equal probability of being selected.

randomized algorithm

An algorithm that involves some form of randomness to control its behavior. The ultimate goal of a randomized algorithm is to improve performance over a deterministic algorithm to solve the same problem. There are a number of variations on this theme. A “Las Vegas algorithm” returns a correct result, but the amount of time required might or might not improve over a **deterministic algorithm**. A “Monte Carlo algorithm” is a form of **probabilistic algorithm** that is not guarenteed to return a correct result, but will return a result relatively quickly.

range

The set of possible outputs for a function.

range query

Records are returned if their relevant key value falls with a specified range.

read/write head

Synonym for **I/O head**.

rebalancing operation

An operation performed on balanced search trees, such as the **AVL Tree** or **Splay Tree**, for the purpose of keeping the tree **height balanced**.

record

A collection of information, typically implemented as an **object** in an **object-oriented programming language**. Many data structures are organized containers for a collection of records.

recurrence relation

A **recurrence relation** (or less formally, recurrence) defines a function by means of an expression that includes one or more (smaller) instances of itself. A classic example is the **recursive** definition for the factorial function, $F(n) = n * F(n-1)$.

recurrence with full history

A special form of **recurrence relation** that includes a summation with a copy of the recurrence inside. The recurrence that represents the average case cost for **Quicksort** is an example. This internal summation can typically be removed with simple techniques to simplify solving the recurrence.

recursion

The process of using recursive calls. An algorithm is recursive if it calls itself to do part of its work. See **recursion**.

recursive call

Within a **recursive function**, it is a call that the function makes to itself.

recursive data structure

A data structure that is partially composed of smaller or simpler instances of the same data structure. For example, **linked lists** and **binary trees** can be viewed as recursive data structures.

recursive function

A function that includes a **recursive call**.

recursively enumerable

A language L is recursively enumerable if there exists a **Turing machine** M such that $L = L(M)$.

Red-Black Tree

A balanced variation on a **BST**.

reduction

In **algorithm analysis**, the process of deriving **asymptotic bounds** for one **problem** from the asymptotic bounds of another. In particular, if problem A can be used to solve problem B, and problem A is proved to be in $O(f(n))$, then problem B must also be in $O(f(n))$. Reductions are often used to show that certain problems are at least as expensive as sorting, or that certain problems are **NP-Complete**.

reference

A value that enables a program to directly access some particular **data item**. An example might be a byte position within a file where the record is stored, or a pointer to a record in memory. (Note that Java makes a distinction between a reference and the concept of a pointer, since it does not define a reference to necessarily be a byte position in memory.)

reference count algorithm

An algorithm for **garbage collection**. Whenever a reference is made from a variable to some memory location, a counter associated with that memory location is incremented. Whenever the reference is changed or deleted, the reference count is decremented. If this count goes to zero, then the memory is considered free for reuse. This approach can fail if there is a cycle in the chain of references.

reference parameter

A **parameter** that has been **passed by reference**. Such a parameter can be modified inside the function or method.

reflexive

In set notation, binary relation R on set S is reflexive if aRa for all $a \in \mathbf{S}$.

Region Quadtree

A **spatial data structure** for storing 2D pixel data. The idea is that the root of the tree represents the entire image, and it is recursively divided into four equal subquadrants if not all pixels associated with the current node have the same value. This is structurally equivalent to a **PR quadtree**, only the decomposition rule is changed.

regular expression

A way to specify a set of strings that define a language using the operators of union, contatenation, and star-closure. A regular expression defines some **regular language**.

regular grammar

And grammar that is either right-regular or left-regular. Every regular grammar describes a regular language.

regular language

A language L is a regular language if and only if there exists a **Deterministic Finite Automata** M such that $L = L(M)$.

relation

In set notation, a relation R over set $\mathbf{S}$ is a set of ordered pairs from $\mathbf{S}$.

replacement selection

A variant of **heapsort** most often used as one phase of an **external sort**. Given a collection of records stored in an **array**, and a stream of additional records too large to fit into **working memory**, replacement selection will unload the **heap** by sending records to an output stream, and seek to bring new records into the heap from the input stream in preference to shrinking the heap size whenever possible.

reserved block

In a **memory manager**, this refers to space in the **memory pool** that has been allocated to store data received from the **client**. This is in contrast to the **free blocks** that represent space in the memory pool that is not allocated to storing client data.

resource constraints

Examples of resource constraints include the total space available to store the data (possibly divided into separate main memory and disk space constraints) and the time allowed to perform each subtask.

root

In a **tree**, the topmost **node** of the tree. All other nodes in the tree are **descendants** of the root.

rotation

In the **AVL Tree** and **Splay Tree**, a rotation is a local operation performed on a node, its children, and its grandchildren that can result in reordering their relationship. The goal of performing a rotation is to make the tree more **balanced**.

rotational delay

When processing a **disk access**, the time that it takes for the first byte of the desired data to move to under the **I/O head**. On average, this will take one half of a disk rotation, and so constitutes a substantial portion of the time required for the disk access.

rotational latency

A synonym for **rotational delay**.

run

A series of sorted records. Most often this refers to a (sorted) subset of records that are being sorted by means of an **external sort**.

run file

A temporary file that is created during the operation of an **external sort**, the run file contains a collection of **runs**. A common structure for an external sort is to first create a series of runs (stored in a run file), followed by merging the runs together.

run-time polymorphism

A form of **polymorphism** known as Overriding. Overridden methods are those which implement a new method with the same signature as a method inherited from its **base class**. Compare to **compile-time polymorphism**.

runtime environment

The environment in which a program (of a particular programming language) executes. The runtime environment handles such activities as managing the **runtime stack**, the **free store**, and the **garbage collector**, and it conducts the execution of the program.

runtime stack

The place where an **activation record** is stored when a subroutine is called during a program's runtime.

scanner

The part of a **compiler** that is responsible for doing **lexical analysis**.

scope

The parts of a program that can see and access a variable.

search key

A field or part of a record that is used to represent the record when searching. For example, in a database of customer records, we might want to search by name. In this case the name field is used as the search key.

search lower bound

The problem of searching in an **array** has provable lower bounds for specific variations of the problem. For an unsorted array, it is $\Omega(n)$ **comparisons** in the **worst case**, typically proved using an **adversary argument**. For a sorted array, it is $\Omega(\log n)$ in the worst case, typically proved using an argument similar to the **sorting lower bound** proof. However, it is possible to search a sorted array in the average case in $O(\log \log n)$ time.

search problem

Given a particular key value K, the search problem is to locate a **record** (k_j, I_j) in some collection of records **L** such that $k_j = K$ (if one exists). **Searching** is a systematic method for locating the record (or records) with key value $k_j = K$.

search tree

A **tree** data structure that makes search by **key** value more efficient. A type of **container**, it is common to implement an **index** using a search tree. A good search tree implementation will guarentee that insertion, deletion, and search operations are all $\Theta(\log n)$.

search trie

Any **search tree** that is a **trie**.

searching

Given a **search key** K and some collection of records **L**, searching is a systematic method for locating the record (or records) in **L** with key value $k_j = K$.

secondary clustering

In **hashing**, the tendency in certain **collision resolution** methods to create clustering in sections of the hash table. In **primary clustering**, this is caused by a cluster of keys that don't necessarily hash to the same slot but which following significant portions of the same **probe sequence** during collision resolution. Secondary clustering results from the keys hashing to the same slot of the table (and so a collision resolution method that is not affected by the key value must use the same probe sequence for all such keys). This problem can be resolved by **double hashing** since its probe sequence is determined in part by a second hash function.

secondary index

Synonym for **secondary key index**.

secondary key

A key field in a record such as salary, where a particular key value might be duplicated in multiple records. A secondary key is more likely to be used by a user as a search key than is the record's **primary key**.

secondary key index

Associates a **secondary key** value with the **primary key** of each record having that secondary key value.

secondary storage

Refers to slower but cheaper means of storing data. Typical examples include a **disk drive**, a USB memory stick, or a solid state drive.

sector

A unit of space on a **disk drive** that is the amount of data that will be read or written at one time by the disk drive hardware. This is typically 512 bytes.

sector header

On a disk drive, a piece of information at the start of a **sector** that allows the **I/O head** to recognize the identity (or equivalently, the address) of the current sector.

seed

In random number theory, the starting value for a random number series. Typically used with any **linear congruential method**.

seek

On a **disk drive**, the act of moving the **I/O head** from one **track** to another. This is usually considered the most expensive step during a **disk access**.

selection sort

While this sort requires $\Theta(n^2)$ time in the **best**, **average**, and **worst** cases, it requires only $\Theta(n)$ swap operations. Thus, it does relatively well in applications where swaps are expensive. It can be viewed as an optimization on **bubble sort**, where a swap is deferred until the end of each iteration.

self-organizing list

A list that, over a series of search operations, will make use of some **heuristic** to re-order its elements in an effort to improve search times. Generally speaking, search is done sequentially from the beginning, but the self-organizing heuristic will attempt to put the records that are most likely to be searched for at or near the front of the list. While typically not as efficient as **binary search** on a sorted list, self-organizing lists do not require that the list be sorted (and so do not pay the cost of doing the sorting operation).

self-organizing list heuristic

A **heuristic** to use for the purpose of maintaining a **self-organizing list**. Commonly used heuristics include **move-to-front** and **transpose**.

separate chaining

In **hashing**, a synonym for **open hashing**

sequence

In set notation, a collection of elements with an order, and which may contain duplicate-valued elements. A sequence is also sometimes called a **tuple** or a **vector**.

sequential access

In **file processing** terminology, the requirement that all records in a file are accessed in sequential order. Alternatively, a storage device that can only access data sequentially, such as a tape drive.

sequential fit

In a **memory manager**, the process of searching the **memory pool** for a **free block** large enough to service a **memory request**, possibly reserving the remaining space as a free block. Examples are **first fit**, **circular first fit**, **best fit**, and **worst fit**.

sequential search

The simplest search algorithm: In an **array**, simply look at the array elements in the order that they appear.

sequential tree representation

A representation that stores a series of node values with the minimum information needed to reconstruct the tree structure. This is a technique for **serializing** a tree.

serialization

The process of taking a data structure in memory and representing it as a sequence of bytes. This is sometimes done in order to transmit the data structure across a network or store the data structure in a **stream**, such as on disk. **Deserialization** reconstructs the original data structure from the serialized representation.

set

A collection of distinguishable **members** or **elements**.

set former

A way to define the membership of a set, by using a text description. Example: $\{x \mid x \text{ is a positive integer}\}$.

set product

Written $\mathbf{Q} \times \mathbf{P}$, the set product is a set of ordered pairs such that ordered pair (a, b) is in the product whenever $a \in \mathbf{P}$ and $b \in \mathbf{Q}$. For example, when $\mathbf{P} = \{2, 3, 5\}$ and $\mathbf{Q} = \{5, 10\}$, $\mathbf{Q} \times \mathbf{P} = \{(2, 5),\ (2, 10),\ (3, 5),\ (3, 10),\ (5, 5),\ (5, 10)\}$.

shallow copy

Copying the **reference** or **pointer** value without copying the actual content.

Shellsort

A sort that relies on the best-case cost of **insertion sort** to improve over $\Theta(n^2)$ **worst case** cost.

shifting method

A technique for finding a **closed-form solution** to a **summation** or **recurrence relation**.

shortest path

Given a **graph** with distances or **weights** on the **edges**, the shortest path between two nodes is the path with least total distance or weight. Examples of the shortest paths problems are the **single-source shortest paths problem** and the **all-pairs shortest paths problem**.

sibling

In a **tree**, a sibling of **node** A is any other node with the same **parent** as A.

signature

In a programming language, the signature for a function is its return type and its list of parameters and their types.

signature file

In document processing, a signature file is a type of **bitmap** used to indicate which documents in a collection contain a given keyword, such that there is a **bitmap** for each keyword.

simple cycle

In **graph** terminology, a **cycle** is simple if its corresponding **path** is simple, except that the first and last **vertices** of the cycle are the same.

simple path

In **graph** terminology, a **path** is simple if all vertices on the path are distinct.

simple type

A **data type** whose values contain no subparts. An example is the integers.

simulating recursion

If a programming language does not support **recursion**, or if you want to implement the effects of recursion more efficiently, you can use a **stack** to maintain the collection of subproblems that would be waiting for completion during the recursive process. Using a loop, whenever a recursive call would have been made, simply add the necessary program state to the stack. When a return would have been made from the recursive call, pop the previous program state off of the stack.

single rotation

A type of **rebalancing operation** used by the **Splay Tree** and **AVL Tree**.

single-source shortest paths problem

Given a **graph** with **weights** or distances on the **edges**, and a designated start **vertex** s, find the shortest path from s to every other vertex in the graph. One algorithm to solve this problem is **Dijkstra's algorithm**.

singly linked list

A **linked list** implementation variant where each list node contains access an pointer only to the next element in the list.

skip list

A form of **linked list** that adds additional links to improve the cost of fundamental operations like insert, delete, and search. It is a **probabilistic data structure** since it adds the additional links using a **probabilistic algorithm**. It can implement a **dictionary** more efficiently than a **BST**, and is roughly as difficult to implement.

slot

In **hashing**, a position in a **hash table**.

snowplow argument

An analogy used to give intuition for why **replacement selection** will generate **runs** that are on average twice the size of working memory. Records coming from the input stream have key values that might be of any size, whose size is related to the position of a falling snowflake. The replacement selection process is analogous to a snowplow that moves around a circular track picking up snow. In steady state, given a certain amount of snow equivalent to **working memory** size M, an amount of snow (incoming records from the input stream) is expected to fall ahead of the plow as the size of the working memory during one cycle of the plow (analogously, one run of the replacement selection algorithm). Thus, the snowplow is expected in one pass (one run of replacement selection) to pick up $2M$ snow.

software engineering

The study and application of engineering to the design, development, and maintenance of software.

software reuse

In **software engineering**, the concept of reusing a piece of software. In particular, using an existing piece of software (such as a function or library) when creating new software.

solution space

The possible solutions to a problem. This typically refers to an **optimization problem**, where some solutions are more desireable than others.

solution tree

An ordering imposed on the set of solutions within a **solution space** in the form of a tree, typically derived from the order that some algorithm would visit the solutions.

sorted list

A **list** where the records stored in the list are arranged so that their **key** values are in ascending order. If the list uses an **array-based list** implementation, then it can use **binary search** for a cost of $\Theta(\log n)$. But both insertion and deletion will be require $\Theta(n)$ time.

sorting lower bound

The lower bound for the **problem** of **sorting** is $\Omega(n \log n)$. This is traditionally proved using a **decision tree** model for sorting algorithms, and recognizing that the minimum depth of the decision tree for any sorting algorithm is $\Omega(n \log n)$ since there are $n!$ permutations of the n input records to distinguish between during the sorting process.

sorting problem

Given a set of records r_1, r_2, …, r_n with **key** values k_1, k_2, …, k_n, the sorting problem is to arrange the records into any order s such that records r_{s_1}, r_{s_2}, …, r_{s_n} have keys obeying the property $k_{s_1} \leq k_{s_2} \leq \ldots \leq k_{s_n}$. In other words, the sorting problem is to arrange a set of records so that the values of their key fields are in non-decreasing order.

space/time tradeoff

Many programs can be designed to either speed processing at the cost of additional storage, or reduce storage at the cost of additional processing time.

sparse graph

A **graph** where the actual number of **edges** is much less than the possible number of edges. Generally, this is interpreted to mean that the **degree** for any **vertex** in the graph is relatively low.

sparse matrix

A matrix whose values are mostly zero. There are a number of data structures that have been developed to store sparse matrices, with the goal of reducing the amount of space required to represent it as compared to simply using a regular matrix representation that stores a value for every matrix position.

spatial

Referring to a position in space.

spatial application

An application what has spatial aspects. In particular, an application that stores records that need to be searched by location.

spatial attribute

An attribute of a record that has a position in space, such as the coordinate. This is typically in two or more dimensions.

spatial data

Any object or record that has a position (in space).

spatial data structure

A **data structure** designed to support efficient processing when a **spatial attribute** is used as the key. In particular, a data structure that supports efficient search by location, or finds all records within a given region in two or more dimensions. Examples of spatial data structures to store point data include the **bintree**, the **PR quadtree** and the **kd tree**.

spindle

The center of a **disk drive** that holds the **platters** in place.

Splay Tree

A variant implementation for the **BST**, which differs from the standard BST in that it uses modified insert and remove methods in order to keep the tree **balanced**. Similar to an **AVL Tree** in that it uses the concept of **rotations** in the insert and remove operations. While a Splay Tree does not guarentee that the tree is balanced, it does guarentee that a series of n operations on the tree will have a total cost of $\Theta(n \log n)$ cost, meaning that any given operation can be viewed as having **amortized cost** of $\Theta(\log n)$.

splaying

The act of performing an **rebalancing operation** on a **Splay Tree**.

stable

A sorting algorithm is said to be stable if it does not change the relative ordering of records with identical **key** values.

stack

A list-like structure in which elements may be inserted or removed from only one end.

stack frame

Frame of data that pushed into and poped from call stack

stack variable

Another name for a **local variable**.

stale pointer

Within the context of a **buffer pool** or **memory manager**, this means a **reference** to a **buffer** or memory location that is no longer valid. For example, a program might make a memory request to a buffer pool, and be given a reference to the buffer holding the requested data. Over time, due to inactivity, the contents of this buffer might be flushed. If the program holding the buffer reference then tries to access the contents of that buffer again, then the data contents will have changed. The possibility for this to occur depends on the design of the interface to the buffer pool system. Some designs make this impossible to occur. Other designs make it possible in an attempt to deliver greater performance.

start state

In a **finite automata**, the designated state in which the machine will always begin a computation.

start symbol

In a **grammar**, the designated **non-terminal** that is the intial point for **deriving** a string in the langauge.

state

The condition that something is in at some point in time. In computing, this typically means the collective values of any existing variables at some point in time. In an **automata**, a state is an abstract condition, possibly with associated information, that is primarily defined in terms of the conditions that the automata may transition from its present state to another state.

State Machine

Synonym for **finite automata**.

static

Something that is not changing (in contrast to **dynamic**). In computer programming, static normally refers to something that happens at compile time. For example, static analysis is analysis of the program's text or structure, as opposed to its run-time behavior. Static binding or static memory allocation occurs at compile time.

static scoping

A synonym for **lexical scoping**.

Strassen's algorithm

A **recursive** algorithm for matrix multiplication. When multiplying two $n \times n$ matrices, this algorithm runs faster than the $\Theta(n^3)$ time required by the standard matrix multiplication algorithm. Specifically, Strassen's algorithm requires time $Theta(n^{\log_2 7})$ time. This is achieved by refactoring the sub-matrix multiplication and addition operations so as to need only 7 sub-matrix multiplications instead of 8, at a cost of additional sub-matrix addition operations. Thus, while the asymptotic cost is lower, the constant factor in the growth rate equation is higher. This makes Strassen's algorithm inefficient in practice unless the arrays being multiplied are rather large. Variations on Strassen's algorithm exist that reduce the number of sub-matrix multiplications even futher at a cost of even more sub-matrix additions.

strategy

An approach to accomplish a task, often encapsulated as an algorithm. Also the name for a **design pattern** that separates the algorithm for performing a task from the control for applying that task to each member of a collection. A good example is a generic sorting function that takes a collection of records (such as an **array**) and a "strategy" in the form of an algorithm that knows how to extract the key from a record in the array. Only subtly different from the **visitor** design pattern, where the difference is primarily one of intent rather than syntax. The strategy design pattern is focused on encapsulating an activity that is part of a larger process, so that different ways of performing that activity can be substituted. The visitor design pattern is focused on encapsulating an activity that will be performed on all members of a collection so that completely different activities can be substituted within a generic method that accesses all of the collection members.

stream

The process of delivering content in a **serialized** form.

strict partial order

In set notation, a relation that is **irreflexive**, **antisymmetric**, and **transitive**.

strong induction

An alternative formulation for the **induction step** in a **proof by induction**. The induction step for strong induction is: If **Thrm** holds for all $k, c \leq k < n$, then **Thrm** holds for n.

subclass

In **object-oriented programming**, any class within a **class hierarchy** that **inherits** from some other class.

subgraph

A subgraph $\mathbf{S}$ is formed from **graph** $\mathbf{G}$ by selecting a **subset** $\mathbf{V}_s$ of $\mathbf{G}$'s **vertices** and a subset $\mathbf{E}_s$ of $\mathbf{G}$'s **edges** such that for every edge $e \in \mathbf{E}_s$, both vertices of e are in $\mathbf{V}_s$.

subset

In set theory, a set A is a subset of a set B, or equivalently B is a **superset** of A, if all elements of A are also elements of B.

subtract-and-guess

A technique for finding a **closed-form solution** to a **summation** or **recurrence relation**.

subtree

A subtree is a **subset** of the nodes of a binary tree that includes some node R of the tree as the subtree **root** along with all the **descendants** of R.

successful search

When searching for a **key** value in a collection of records, we might find it. If so, we call this a successful search. The alternative is an **unsuccessful search**.

summation

The sum of costs for some **function** applied to a range of parameter values. Often written using Sigma notation. For example, the sum of the integers from 1 to n can be written as $\sum_{i=1}^{n} i$.

superset

In set theory, a set A is a **subset** of a **set** B, or equivalently B is a **superset** of A, if all **elements** of A are also elements of B.

symbol table

As part of a **compiler**, the symbol table stores all of the identifiers in the program, along with any necessary information needed about the identifier to allow the compiler to do its job.

symmetric

In set notation, relation R is symmetric if whenever aRb, then bRa, for all $a, b \in \mathbf{S}$.

symmetric matrix

A square matrix that is equal to its **transpose**. Equivalently, for a $n \times n$ matrix A, for all $i, j < n$, $A[i, j] = A[j, i]$.

syntax analysis

A phase of **compilation** that accepts **tokens**, checks if program is syntactically correct, and then generates a **parse tree**.

tail

The end of a **list**.

terminal

A specific character or string that appears in a **production rule**. In contrast to a **non-terminal**, which represents an abstract state in the production. Similar to a **literal**, but this is the term more typically used in the context of a **compiler**.

Theta notation

In **algorithm analysis**, Θ notation is used to indicate that the **upper bound** and **lower bound** for an **algorithm** or **problem** match.

token

The basic logical units of a program, as deterimined by **lexical analysis**. These are things like arithmetic operators, language keywords, variable or function names, or numbers.

tombstone

In **hashing**, a tombstone is used to mark a **slot** in the **hash table** where a record has been deleted. Its purpose is to allow the **collision resolution** process to probe through that slot (so that records further down the **probe sequence** are not unreachable after deleting the record), while also allowing the slot to be reused by a future insert operation.

topological sort

The process of laying out the **vertices** of a **DAG** in a **linear order** such that no vertex A in the order is preceded by a vertex that can be reached by a (directed) **path** from A. Usually the (directed) edges in the graph define a prerequisite system, and the goal of the topological sort is to list the vertices in an order such that no prerequisites are violated.

total order

A binary relation on a set where every pair of distinct elements in the set are **comparable** (that is, one can determine which of the pair is greater than the other).

total path length

In a **tree**, the sum of the **levels** for each **node**.

Towers of Hanoi problem

A standard example of a recursive algorithm. The problem starts with a stack of disks (each with unique size) stacked decreasing order on the left pole, and two additional poles. The problem is to move the disks to the right pole, with the constraints that only one disk can be moved at a time and a disk may never be on top of a smaller disk. For n disks, this problem requires $\Theta(2^n)$ moves. The standard solution is to move $n-1$ disks to the middle pole, move the bottom disk to the right pole, and then move the $n-1$ disks on the middle pole to the right pole.

track

On a **disk drive**, a concentric circle representing all of the **sectors** that can be viewed by the **I/O head** as the disk rotates. The significance is that, for a given placement of the I/O head, the sectors on the track can be read without performing a (relatively expensive) **seek** operation.

track-to-track seek time

Expected (average) time to perform a **seek** operation from a random **track** to an adjacent track. Thus, this can be viewed as the minimum possible seek time for the **disk drive**. This is one of two metrics commonly provided by disk drive vendors for disk drive performance, with the other being **average seek time**.

trailer node

Commonly used in implementations for a **linked list** or related structure, this **node** follows the last element of the list. Its purpose is to simplify the code implementation by reducing the number of special cases that must be programmed for.

transducer

A machine that takes an input and creates an output. A **Turing Machine** is an example of a transducer.

transitive

In set notation, relation R is transitive if whenever aRb and bRc, then aRc, for all $a, b, c \in \mathbf{S}$.

transpose

In the context of linear algebra, the transpose of a matrix A is another matrix A^T created by writing the rows of A as the columns of A^T. In the context of a **self-organizing list**, transpose is a **heuristic** used to maintain the list. Under this heuristic, whenever a record is accessed it is moved one position closer to the front of the list.

trap state

In a **FSA**, any state that has all transitions cycle back to itself. Such a state might be **final**.

traversal

Any process for visiting all of the objects in a collection (such as a **tree** or **graph**) in some order.

tree

A tree $\mathbf{T}$ is a finite set of one or more **nodes** such that there is one designated node R, called the **root** of $\mathbf{T}$. If the set $(\mathbf{T} - \{R\})$ is not empty, these nodes are partitioned into $n > 0$ **disjoint sets** $\mathbf{T}_0$, $\mathbf{T}_1$, ..., $\mathbf{T}_{n-1}$, each of which is a tree, and whose **roots** $R_1, R_2, \ldots, R_n$, respectively, are **children** of R.

tree traversal

A **traversal** performed on a tree. Traditional tree traversals include **preorder** and **postorder** traversals for both **binary** and **general** trees, and **inorder traversal** that is most appropriate for a **BST**.

trie

A form of **search tree** where an internal node represents a split in the **key space** at a predetermined location, rather than split based on the actual **key** values seen. For example, a simple binary search trie for key values in the range 0 to 1023 would store all records with key values less than 512 on the left side of the tree, and all records with key values equal to or greater than 512 on the right side of the tree. A trie is always a **full tree**. Folklore has it that the term comes from "retrieval", and should be pronounced as "try" (in contrast to "tree", to distinguish the differences in the space decomposition method of a search tree versus a search trie). The term "trie" is also sometimes used as a synonym for the **alphabet trie**.

truth table

In symbolic logic, a table that contains as rows all possible combinations of the boolean variables, with a column that shows the outcome (true or false) for the expression when given that row's truth assignment for the boolean variables.

tuple

In set notation, another term for a **sequence**.

Turing machine

A type of **finite automata** that, while simple to define completely, is capable of performing any computation that can be performed by any known computer.

Turing-acceptable

A language is $Turing-acceptable$ if there is some **Turing machine** that **accepts** it. That is, the machine will halt in an accepting configuration if the string is in the language, and go into a **hanging configuration** if the string is not in the language.

Turing-computable function

Any function for which there exists a Turing machine that can perform the necessary work to compute the function.

Turing-decidable

A language is Turing-decideable if there exists a Turing machine that can clearly indicate for every string whether that string is in the language or not. Every Turing-decidable language is also Turing-acceptable, because the Turing machine that can decide if the string is in the language can be modified to go into a **hanging configuration** if the string is not in the language.

two-coloring

An assignment from two colors to regions in an image such that no two regions sharing a side have the same color.

type

A collection of values.

unary notation

A way to represent **natural numbers**, where the value of zero is represented by the empty string, and the value n is represented by a series of n marks.

uncountably infinite
uncountable

An infinite set is uncountably infinite if there does not exist any mapping from it to the set of integers. This is often proved using a **diagonalization argument**. The real numbers is an example of an uncountably infinite set.

underflow

The condition where the amount of data stored in an entity has dropped below some minimum threshold. For example, a node in a **B-tree** is required to be at least half full. If a record deletion causes the node to be less than half full, then it is in a condition of underflow, and something has to be done to correct this.

undirected edge

An **edge** that connects two **vertices** with no direction between them. Many graph representations will represent such an edge with two **directed edges**.

undirected graph

A **graph** whose **edges** do not have a direction.

uninitialized

Uninitialized variable means it has no initial value.

UNION

One half of the **UNION/FIND** algorithm for managing **disjoint sets**. It is the process of merging two trees that are represented using the **parent pointer representation** by making the root for one of the trees set its parent pointer to the root of the other tree.

UNION/FIND

A process for mainining a collection of disjoint sets. The **FIND** operation determines which disjoint set a given object resides in, and the **UNION** operation combines two disjoint sets when it is determined that they are members of the same **equivalence class** under some **equivalence relation**.

unit production

A unit production is a **production** in a **grammar** of the form $A \rightarrow B$, where $A, B \in$ the set of **non-terminals** for the grammar. Any grammar with unit productions can be rewritten to remove them.

unsolveable problem

A problem that can proved impossible to solve on a computer. The classic example is the **halting problem**.

unsorted list

A **list** where the records stored in the list can appear in any order (as opposed to a **sorted list**). An unsorted list can support efficient ($\Theta(1)$) insertion time (since you can put the record anywhere convenient), but requires $\Theta(n)$ time for both search and and deletion.

unsuccessful search

When searching for a **key** value in a collection of records, we might not find it. If so, we call this an unsuccessful search. Usually we require that this means that no record in the collection actually has that key value (though a **probabilistic algorithm** for search might not require this to be true). The alternative to an unsuccessful search is a **successful search**.

unvisited

In **graph** algorithms, this refers to a node that has not been processed at the current point in the algorithm. This information is typically maintained by using a **mark array**.

upper bound

In **algorithm analysis**, a **growth rate** that is always greater than or equal to the growth rate of the **algorithm** in question. In practice, this is the slowest-growing function that we know grows at least as fast as all but a constant number of inputs. It could be a gross over-estimate of the truth. Since the upper bound for the algorithm can be very different for different situations (such as the **best case** or **worst case**), we typically have to specify which situation we are referring to.

value parameter

A **parameter** that has been **passed by value**. Changing such a parameter inside the function or method will not affect the value of the calling parameter.

variable-length coding

Given a collection of objects, a variable-length coding scheme assigns a code to each object in the collection using codes that can be of different lengths. Typically this is done in a way such that the objects that are most likely to be used have the shortest codes, with the goal of minimizing the total space needed to represent a sequence of objects, such as when representing the characters in a document. **Huffman coding** is an example of a variable-length coding scheme. This is in contrast to **fixed-length coding**.

vector

In set notation, another term for a **sequence**. As a data structure, the term vector usually used as a snyonym for a **dynamic array**.

vertex

Another name for a **node** in a **graph**.

virtual memory

A memory management technique for making relatively fast but small memory appear larger to the program. The large “virtual” data space is actually stored on a relatively slow but large **backing storage** device, and portions of the data are copied into the smaller, faster memory as needed by use of a **buffer pool**. A common example is to use **RAM** to manage access to a large virtual space that is actually stored on a **disk drive**. The programmer can implement a program as though the entire data content were stored in RAM, even if that is larger than the physical RAM available making it easier to implement.

visit

During the process of a **traversal** on a **graph** or **tree** the action that takes place on each **node**.

visited

In **graph** algorithms, this refers to a node that has previously been processed at the current point in the algorithm. This information is typically maintained by using a **mark array**.

visitor

A **design pattern** where a **traversal** process is given a function (known as the visitor) that is applied to every object in the collection being traversed. For example, a generic tree or graph traversal might be designed such that it takes a function parameter, where that function is applied to each node.

volatile

In the context of computer memory, this refers to a memory that loses all stored information when the power is turned off.

weight

A cost or distance most often associated with an **edge** in a **graph**.

weighted graph

A **graph** whose **edges** each have an associated **weight** or cost.

weighted path length

Given a tree, and given a **weight** for each leaf in the tree, the weighted path length for a leaf is its weight times its **depth**.

weighted union rule

When merging two disjoint sets using the **UNION/FIND** algorithm, the weighted union rule is used to determine which subtree's root points to the other. The root of the subtree with fewer nodes will be set to point to the root of the subtree with more nodes. In this way, the average depth of nodes in the resulting tree will be less than if the assignment had been made in the other direction.

working memory

The portion of **main memory** available to an algorithm for its use. Typically refers to main memory made available to an algorithm that is operating on large amounts of data stored in **peripheral storage**, the working memory represents space that can hold some subset of the total data being processed.

worst case

In algorithm analysis, the **problem instance** from among all problem instances for a given input size n that has the greatest cost. Note that the worst case is **not** when n is big, since we are referring to the wrost from a class of inputs (i.e, we want the worst of those inputs of size n).

worst fit

In a **memory manager**, worst fit is a **heuristic** for deciding which **free block** to use when allocating memory from a **memory pool**. Worst fit will always allocate from the largest free block. The rationale is that this will be the method least likely to cause **external fragmentation** in the form of small, unuseable memory blocks. The disadvantage is that it tends to eliminate the availability of large freeblocks needed for unusually large requests.

zigzig

A type of **rebalancing operation** used by **splay trees**.

Zipf distribution

A data distribution that follows Zipf's law, an emprical observation that many types of data studied in the physical and social sciences follow a power law probability distribution. That is, the frequency of any record in the data collection is inversely proportional to its rank when the collection is sorted by frequency. Thus, the most frequently appearing record has a frequency much higher than the next most frequently appearing record, which in turn has a frequency much higher than the third (but with ratio slightly lower than that for the first two records) and so on. The **80/20 rule** is a casual characterization of a Zipf distribution. Adherence to a Zipf distribution is important to the successful operation of a **cache** or **self-organizing list**.

zone

In **memory management**, the concept that different parts of the **memory pool** are handled in different ways. For example, some of the memory might be handled by a simple **freelist**, while other portions of the memory pool might be handled by a **sequential fit** memory manager. On a **disk drive** the concept of a zone relates to the fact that there are limits to the maximum data density, combined with the fact that the need for the same angular distance to be used for a sector in each track means that tracks further from the center of the disk will become progressively less dense. A zone in this case is a series of adjacent tracks whose data density is set by the maximum density of the innermost track of that zone. The next zone can then reset the data density for its innermost track, thereby gaining more total storage space while preserving angular distance for each sector.

Appendix B: Bibliography

10.2. Bibliography

Ahern05

Dennis Ahern et al., *CMMI Distilled: a practical introduction to integrated process improvement*, 2005. ISBN: 0-321-18613-3.

Bacon

Francis Bacon, *Novum Organum*, Google eBook, Clarendon Press, 1878.

Beck99

Kent Beck. *Extreme Programming Explained: Embrace Change*. 1999.

Bloch

Joshua Bloch, *Effective Java*, Second Edition, Addison-Wesley, 2008.

Boehm03

Barry Boehm and Richard Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, 2003. ISBN: 0-321-18612-5.

Booch

Grady Booch, *Object-Oriented Design With Applications*, Benjamin/Cummings, Menlo Park, California, 1991.

Brooks95

Frederick P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Second Edition, Addison-Wesley, 1995.

Cockburn04

Alistair Cockburn, *Crystal Clear: A Human-Powered Methodology for Small Teams*, 2004. ISBN: 0-201-69947-8

GalilItaliano91

Zvi Galil and Giuseppe F. Italiano, “Data Structures and Algorithms for Disjoint Set Union Problems”, *Computing Surveys 23*, 3(September 1991), 319-344.

Gauss65

Carl F. Gauss, Arthur A. Clarke (translator) *Disquisitiones Arithmeticae*, Yale University Press, 1965.

KnuthV3

Donald E. Knuth, *The Art of Computer Programming Volume 3: Sorting and Searching*, Second Edition, Addison-Wesley, Reading, MA, 1998.

Lafore

Robert Lafore, *Data Structures & Algorithms in Java*, Second Edition, Sams Publishing, 2003.

Sierra

Kathy Sierra and Bert Bates, *OCA/OCP Java 7 SE Programmer I & II Study Guide (Exams 1Z0-803 & 1Z0-804)*, McGraw-Hill Education, 2015.

Tarjan75

Robert E. Tarjan, “On the efficiency of a good but not linear set merging algorithm”, *Journal of the ACM 22*, 2(April 1975), 215-225.