Chapter 1: The Study of Data Structures

The study of data structures has long been considered the cornerstone and starting point for the systematic examination of computer science as a discipline. There are many reasons for this. There is the practical realization that almost every program of more than trivial complexity will need to manage collections of information, and so will require the use of one or more data structures. Learning the tools and techniques that have proven over a long period of time to be useful for software development will help the student become a more productive programmer. But that is only the first, and most practical, of many reasons.

Data structures are one of the easiest ideas to visualize *abstractly*, as most data structures can be matched with a metaphor that the reader will already be familiar with from everyday life. A *stack*, for example, is a collection that is managed in much the same fashion as a stack of dishes; namely, only the topmost item is readily available, and the top must be removed before the item underneath can be accessed. The ability to deal with abstract ideas, and the associated concept of *information hiding*, are the primary tools that computer scientists (or, for that matter, all scientists) use to manage and manipulate complex systems. Learning to program with abstractions, rather than entirely with concrete representations, is a sign of programming maturity.

The analysis of data structures involves a variety of mathematical and other analytical techniques. These help reinforce the idea that computer science is a *science*, and that there is much more to the field than simple programming.

It is in the examination of data structures that the student will likely first encounter the problems involved in the management of large applications. Modern software development typically involves teams or programmers, often dozens or more, working on separate components of a larger system. The difficulties in such development are typically not algorithmic in nature, but deal more with management of information. For example, if the software developed by programmer A must interact with the software developed by programmer B, what is the minimal amount of information that must be communicated between A and B in order to ensure that their components will correctly interact with each other? This is not a problem that the student will likely have seen in their beginning programming courses. Once more, abstraction and information hiding are keys to success. Each of these topics will be explored in more detail in the chapters that follow.

The book is divided into three sections. In the first section you will learn about tools and techniques used in the analysis of data structures. In the second part you will learn about the abstractions that are considered the classic core of the study of data structures. The third section consists of worksheets.

Active Learning

It is in the third section that this book distinguishes itself most clearly from other examinations of the same topic. The third section is a series of worksheets, each of which is tied to the material presented in earlier chapters. Simply stated, this book asks you to *do* more, and to *read* (or, if you are in a traditional classroom setting, *listen to*) less. By becoming a more *active* participant in the educational process, the knowledge you gain will become more familiar to you, and you will hopefully be more comfortable with your abilities. This approach has been used in a variety of different forms for many years, with great success.

While the worksheets can be used in an individual situation, it is the author's opinion (supported by experience), that they work best in a group setting. When students work in a group, they can help each other learn the basic material. Additionally, group work helps students develop their communication skills, as well as their programming skills. In the authors use, a typical class consists of a short fifteen to twenty minute lecture, followed by one or two worksheets completed in class.

At the end of each chapter are study questions that you can use to measure your understanding of the material. These questions are not intended to be difficult, and if you have successfully mastered the topic of the chapter you should be able to immediately write short answers for each of these questions. These study questions are followed by more traditional short answer questions, analysis questions that require more complex understanding, and programming projects. Finally, in recent years there has been an explosion of information available on the web, much of it actually accurate, truthful and useful. Each chapter will end with references to where the student can learn more about the topic at hand.

A Note on Languages

Most textbooks are closely tied to a specific programming language. In recent years it has also become commonplace to emphasize a particular programming paradigm, such as the use of object-oriented programming. The author himself is guilty of writing more than a few books of this sort. However, part of the beauty of the study of data structures is that the knowledge transcends any particular programming language, or indeed most programming paradigms. While the student will of necessity need to write in a particular programming language if they are to produce a working application, the presentation given in this book is purposely designed to emphasize the more abstract details, allowing the student to more easily appreciate the essential features, and not the unimportant aspects imposed by any one language. An appendix at the end of the book is devoted to describing a few language-specific features.

Chapter 2: Algorithms

A computer must be given a series of commands, termed a *program*, in order to make it perform a useful task. Because programs can be large, they are divided into smaller units, termed (depending upon your programming language), functions, procedures, methods, subroutines, subprograms, classes, or packages. In programs that you have written you have undoubtedly used some of these features.

```
double sqrt (double val) {
   /* return square root of argument */
   /* works only for positive values */
   assert (val >= 0);
   double guess = val / 2.0;
   …
   return guess;
}
```

A function, for example, is used to package a task to be performed. Let us use as an example a function to compute the square root of a floating-point number. Among other benefits, the function is providing an *abstraction*. By this we mean that from the outside, from the point of view of the person calling the function, they only need to know the name of the action to be performed. On the other hand, inside the function the programmer must have written the exact series of instructions needed to perform the task. Since to use the function you do not need to know the internal details, we say that the function is a form of *encapsulation*.

The difference between the information necessary to *use* a function and the information necessary to *write* a function, the fact that you can use a function without knowing how it is implemented, is one of the most important tools used to manage complex systems. In your own programming experience you have undoubtedly used libraries of functions, for performing tasks such as input and output, or manipulating graphical interfaces. You can use these functions with only a minimal knowledge of how they go about their task. This makes developing programs infinitely easier than if every programmer had to write all of these actions from scratch.

There is another level of abstraction that is equally important. Let us again illustrate this using the function to compute a square root. In order to make this function available for your use, at some point in the past there must have been a programmer who developed the code that you invoke when you call the square root function. Although this programmer may never have written this particular function, they did not start entirely from scratch. Fortunately for this programmer, there are well known techniques that can be used to solve this problem. In this case, the technique was most likely something termed Newton's Method. This technique, discovered by Issac Newton (1643-1727), finds a square root by first making a guess, then using the first guess to find a better guess, and so on until the correct answer is located.

Newton, of course, lived well before the advent of computers, and so never wrote his technique as a computer function. Instead, he described an *algorithm*. An algorithm is a description of how a specific problem can be solved, written at a level of detail that can

be followed by the reader. Terms that have a related meaning include process, routine, technique, procedure, pattern, and recipe.

It is important that you understand the distinction between an *algorithm* and a *function*. A function is a set of instructions written in a particular programming language. A function will often embody an algorithm, but the essence of the algorithm transcends the function, transcends even the programming language. Newton described, for example, the way to find square roots, and the details of the algorithm will remain the same no matter what programming language is used to write the code.

Properties of Algorithms

Once you have an algorithm, to solve a problem is simply a matter of executing each instruction of the algorithm in turn. If this process is to be successful there are several characteristics an algorithm must possess. Among these are:

input preconditions. The most common form of algorithm is a transformation that takes a set of input values and performs some manipulations to yield a set of output values. (For example, taking a positive number as input, and returning a square root). An algorithm must make clear the number and type of input values, and the essential initial conditions those input values must possess to achieve successful operation. For example, Newtons method works only if the input number is larger than zero.

precise specification of each instruction. Each step of an algorithm must be well defined. There should be no ambiguity about the actions to be carried out at any point. Algorithms presented in an informal descriptive form are sometimes ill-defined for this reason, due to the ambiguities in English and other natural languages.

correctness. An algorithm is expected to solve a problem. For any putative algorithm, we must demonstrate that, in fact, the algorithm will solve the problem. Often this will take the form of an argument, mathematical or logical in nature, to the effect that if the input conditions are satisfied and the steps of the algorithm executed then the desired outcome will be produced.

termination, time to execute. It must be clear that for any input values the algorithm is guaranteed to terminate after a finite number of steps. We postpone until later a more precise definition of the informal term “steps.'” It is usually not necessary to know the exact number of steps an algorithm will require, but it will be convenient to provide an upper bound and argue that the algorithm will always terminate in fewer steps than the upper bound. Usually this upper bound will be given as a function of some values in the input. For example, if the input consists of two integer values n and m, we might be able to say that a particular algorithm will always terminate in fewer than n + m steps.

description of the result or effect. Finally, it must be clear exactly what the algorithm is intended to accomplish. Most often this can be expressed as the production of a result value having certain properties. Less frequently algorithms are executed for a *side effect*,

such as printing a value on an output device. In either case, the expected outcome must be completely specified.

Human beings are generally much more forgiving than computers about details such as instruction precision, inputs, or results. Consider the request "Go to the store and buy something to make lunch". How many ways could this statement be interpreted? Is the request to buy ingredients (for example, bread and peanut butter), or for some sort of mechanical "lunch making" device (perhaps a personal robot). Has the input or expected result been clearly identified? Would you be surprised if the person you told this to tried to go to a store and purchase an automated lunch-making robot? When dealing with a computer, every step of the process necessary to achieve a goal must be outlined in detail.

A form of algorithm that most people have seen is a recipe. In worksheet 1 you will examine one such algorithm, and critique it using the categories given above. Creating algorithms that provide the right level of detail, not too abstract, but also not too detailed, can be difficult. In worksheet 2 you are asked to describe an activity that you perform every day, for example getting ready to go to school. You are then asked to exchange your description with another student, who will critique your algorithm using these objectives.

Specification of Input

An algorithm will, in general, produce a result only when it is used in a proper fashion. Programs use various different techniques to ensure that the input is acceptable for the algorithm. The simplest of these is the idea of *types*, and a function type signature. The function shown at right, for instance, returns the smaller of two integer values. The compiler can check that when you call the function the arguments are integer, and that the result is an integer value.

```
int min (int a, int b) {
   /* return smaller argument */
   if (a < b) return a;
   else return b;
}
```

Some requirements cannot be captured by type signatures. A common example is a range restriction. For instance, the square root program we discussed earlier only works for positive numbers. Typically programs will check the range of their input at run-time, and issue an error or exception if the input is not correct.

Description of Result

Just as the input conditions for computer programs are specified in a number of ways, the expected results of execution are documented in a variety of fashions. The most obvious is the result type, defined as part of a function signature. But this only specifies the type of the result, and not the relationship to the inputs. So equally important is some sort of documentation, frequently written as a comment. In the min function given earlier it is noted that the result is not only an integer, but it also represents the smaller of the two input values.

Instruction Precision

Algorithms must be specified in a form that can be followed to produce the desired outcome. In this book we will generally present algorithms in a high level pseudo-code, a form that can easily be translated into a working program in a given programming language. Using a pseudo-code form allows us to emphasize the important properties of the algorithm, and downplay incidental details that, while they may be necessary for a working program, do nothing to assist in the understanding of the procedure.

Time to execute

Traditionally the discussion of execution time is divided into two separate questions. The first question is showing that the algorithm will terminate for all legal input values. Once this question has been settled, the next question is to provide a more accurate characterization of the amount of time it will take to execute. This second question will be considered in more detail in a later section. Here, we will address the first.

The basic tool used to prove that an algorithm terminates is to find a property or value that satisfies the following three characteristics:

1. The property or value can be placed into a correspondence with integer values.
2. The property is nonnegative.
3. The property or value decreases steadily as the algorithm executes.

The majority of the time this value is obvious, and is left unstated. For instance, if an algorithm operates by examining each element of an array in turn, we can use the remaining size of the array (that is, the unexplored elements) as the property we seek. It is only in rare occasions that an explicit argument must be provided.

An example in which termination may not be immediately obvious occurs in Euclid's Greatest Common Divisor algorithm, first developed in third century B.C. Greece. Donald Knuth, a computer scientist who has done a great deal of research into the history of algorithms, has claimed that this is the earliest-known nontrivial completely specified mathematical algorithm. The algorithm is intended to compute, given two positive integer values n and m, the largest positive integer that evenly divides both values. The algorithm can be written as shown at right.

```
int gcd (int n, int m) {
        /* compute the greatest common divisor
        of two positive integer values */
        while (m != n) {
                if (n > m)
                        n = n - m;
                else
                        m = m - n;
        }
        return n;
}
```

Because there is no definite limit on the number of iterations, termination is not obvious. But we can note that either n or m

becomes smaller on each iteration, and so the *sum* n+m satisfies our three conditions, and hence can be used to demonstrate that the function will halt.

Zeno's Paradox

The most nonintuitive of the three properties required for proving termination is the first: that the quantity or property being discussed can be placed into a correspondence with a diminishing list of integers. Why integer? Why not just numeric? Why diminishing? To illustrate the necessity of this requirement, consider the following "proof" that it is possible to share a single candy bar with all your friends, no matter how many friends you may have. Take a candy bar and cut it in half, giving one half to your best friend. Take the remaining half and cut it in half once more, giving a portion to your second best friend. Assuming you have a sufficiently sharp knife, you can continue in this manner, at each step producing ever-smaller pieces, for as long as you like. Thus, we have no guarantee of termination for this process. Certainly the sequence here (1/2, ¼, 1/8, 1/16 …) consists of all nonnegative terms, and is decreasing. But a correspondence with the integers would need to increase, not decrease; and any diminishing sequence would be numeric, but not integer.

In the 5th century B.C. the Greek philosopher Zeno used similar arguments to show that no matter how fast he runs, the famous hero Achilles cannot overtake a Tortoise, if the Tortoise is given a head start. Suppose, for example, that the Tortoise and Achilles start as below at time A. When, at time B, Achilles reaches the point from which the Tortoise started, the Tortoise will have proceeded some distance ahead of this point. When Achilles reaches this further point at time C, the Tortoise will have proceeded yet further ahead. At each point when Achilles reaches a point from which the Tortoise began, the Tortoise will have proceeded at least some distance further. Thus, it was impossible for Achilles to ever overtake and pass the Tortoise.

Achiles	A	->	B	->	C	->	D
Tortoise			A	->	B	->	C -> D

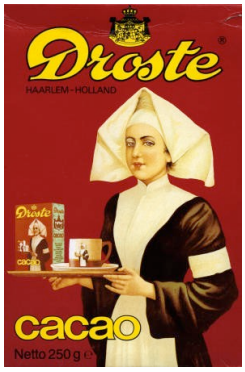
The paradox arises due to an infinite sequence of ever-smaller quantities, and the nonintuitive possibility that an infinite sum can nevertheless be bounded by a finite value. By restricting our arguments concerning termination to use only a decreasing set of *integer* values, we avoid problems such as those typified by Zeno's paradox.

Recursive Algorithms

Recursion is a very common technique used for both algorithms and the functions that result from implementing algorithms. The basic idea of recursion is to "reduce" a complex problem by "simplifying" it in some way, then invoking the algorithm on the simpler problem. This is performed repeated until one or more special cases, termed *base*

cases, are encountered. Arguments that do not correspond to base cases are called *recursive cases*, or sometimes *inductive cases*.

Recursion is a common theme in Art or commercial graphics. A seemingly infinite series of objects can be generated by looking at two opposing mirrors, for example. The image on box of a popular brand of cocoa shows a woman holding a tray of cocoa, including a box of cocoa, on which is found a picture of a woman holding a tray of cocoa, and so on.



You have undoubtedly seen mathematical definitions that are defined in a recursive fashion. The exponential function, for instance, is traditionally defined as follows:

N! = N * (N-1)! for all values N > 0
0! = 1

Here zero is being used as the base case, and the simplification consists of subtracting one, then invoking the factorial on the smaller number. The definition suggests an obvious algorithm for computing the factorial.

> **How to compute the factorial of N**. If N is less than zero, issue an error. Otherwise, if N is equal to zero, return 1. Otherwise, compute the factorial of (N-1), then multiply this result by N and return the result.

```
int factorial (int N) {
  assert (N >= 0);
  if (N == 0) return 1;
  return N * factorial (N – 1);
}
```

The box at right shows this algorithm expressed as a function. To prove termination we would use the same technique as before; that is, identify a value that is integer, decreasing, and non-negative. In this case the value n itself suffices for this argument.

Many algorithms are most easily expressed in a recursive form. The base case need not be a single value, it can sometimes be a condition that the argument must satisfy. A simple example that illustrates this is printing a decimal number, such as 4973. It is relatively easy to handle the single digits, zero to 9, as special cases. Here there are ten base cases. For larger numbers, recursively print the value you get by dividing the number by ten, then print the single digit you get when you take the remainder divided by ten. In our example printing 4973 would recursively call itself to print 497, which would in turn recursively call itself to print 49, which would yet again recursively call itself to print 4. The final call results in one of our base cases, so 4 would be printed. Once this function returns the value 9 is printed, which is the remainder left after dividing 49 by ten. Once this call returns the value 7 would be printed, which is the remainder left after dividing 497 by ten. And finally the value 3 would be printed, which is the remainder left after diving 4973 by ten.

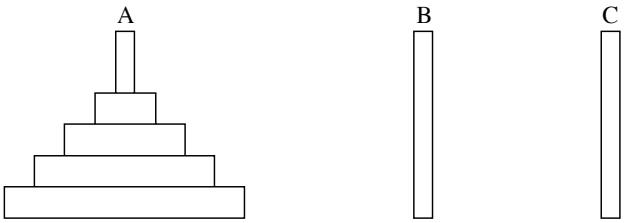
```
void printInteger (int n) {
  assert (n > 0);
  if (n > 9)
    printInteger (n / 10);
  printDigit (n % 10);
}
```

The function resulting from this algorithm is shown at left. Here the base cases have been hidden in another function,

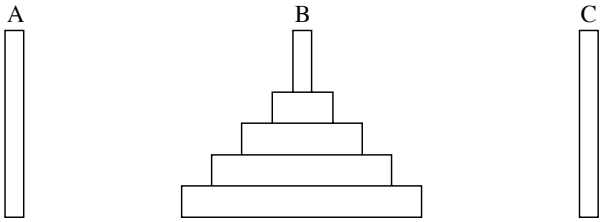
printDigit, that handles only the values zero to nine. In trying to understand how a recursive function can work, it is important to remember that every time a procedure is invoked, new space is allocated for parameters and local variables. So each time the function printInteger is called, a new value of n is created. Previous values of n are left pending, stacked up so that they can be restored after the function returns. This is called the *activation record stack*. A portion of the activation record stack is shown at right, a snapshot showing the various values of n that are pending when the value 4 is printed at the end of the series of recursive calls. Once the latest version of printInteger finishes the caller, who just happens to be printInteger, will be restarted. But in the caller the value of variable n will have been restored to the value 49.

N = 4
N = 49
N = 497
N = 4973

A classic example of a recursive algorithm is the puzzle known as the *Towers of Hanoi*. In this puzzle, three poles are labeled A, B and C. A number of disks of decreasing sizes are, initially, all on pole A.

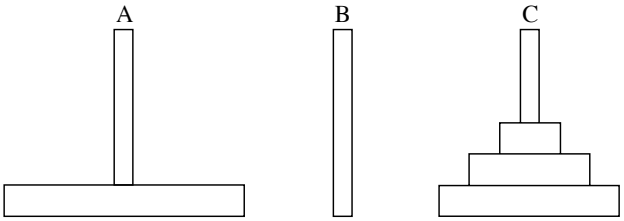


The goal of the puzzle is to move all disks from pole A to pole B, without ever moving a disk onto another disk with a smaller size. The third pole can be used as a temporary during this process. At any point, only the topmost disk from any pole may be moved.



It is obvious that only two first steps are legal. These are moving the smallest disk from pole A to pole B, or moving the smallest disk from pole A to pole C. But which of these is the correct first move?

As is often the case with recursive algorithms, a greater insight is found by considering a point in the middle of solving the problem, rather that thinking about how to begin. For example, consider the point we would like to be one step before we move the largest disk. For this to be a legal move, we must have already moved all the disks except for the largest from pole A to pole C.



Notice that in this picture we have moved all but the last disk to pole C. Once we move the largest disk, pole A will be empty, and we can use it as a temporary. This observation hints at how to express the solution to the problem as a recursive algorithm.

How to move N disks from pole A to pole B using pole C as a temporary: If N is 1, move disk from A to B. Otherwise, move (N – 1) disks from pole A to pole C using pole B as a temporary. Move the largest disk from pole A to pole B. Then move disks from pole C to pole B, using pole A as a temporary.

```
void solveHanoi (int n, char a, char b, char c) {
  if (n == 1) print ("move disk from pole ", a, " to pole ", b);
  else {
    solveHanoi (n – 1, a, c, b);
    print ("move disk from pole ", a, " to pole ", b);
    solveHanoi (n – 1, c, b, a);
  }
}
```

We could express this algorithm as pseudo-code as shown. Here the arguments are used to represent the size of the stack and the names of the poles, and the instructions for solving the puzzle are printed.

We will see many recursive algorithms as we proceed through the text.

Study Questions

1. What is abstraction? Explain how a function is a form of abstraction.

2. What is an algorithm?

3. What is the relationship between an algorithm and a function?

4. Give an example input condition that cannot be specified using only a type.

5. What are two different techniques used to specify the input conditions for an algorithm?

6. What are some ways used to describe the outcome, or result, of executing an algorithm?

7. In what way does the precision of instructions needed to convey an algorithm to another human being differ from that needed to convey an algorithm to a computer?

8. In considering the execution time of algorithms, what are the two general types of questions one can ask?

9. What are some situations in which termination of an algorithm would not be immediately obvious?

10. What are the three properties a value must possess in order to be used to prove termination of an algorithm?

11. What is a recursive algorithm? What are the two sections of a recursive algorithm?

12. What is the activation record stack? What values are stored on the activation record stack? How does this stack simplify the execution of recursive functions?

Analysis Exercises

1. Examine a recipe from your favorite cookbook. Evaluate the recipe with respect to each of the characteristics described at the beginning of this chapter.

2. This chapter described how termination of an algorithm can be demonstrated by finding a value or property that satisfies three specific conditions. Show that all three conditions are necessary. Do this by describing situations that satisfy two of the three, but not the third, and that do not terminate. For example, the text explained how you can share a single candy bar with an infinite number of friends, by repeatedly dividing the candy bar in half. What property does this algorithm violate?

3. The version of the towers of Hanoi used a stack of size 1 as the base case. Another possibility is to use a stack of size zero. What actions need to be performed to move a stack of size zero from one pole to the next? Rewrite the algorithm to use this formulation.

4. What integer value suffices to demonstrate that the towers of Hanoi algorithm must eventually terminate for any size tower?

Exercises

1. Assuming you have the ability to concatenate a character and a string, describe in pseudo-code a recursive algorithm to reverse a string. For example, the input "function" should produce the output "noitcnuf".

2. Express the algorithm to compute the value a raised to the n^{th} power as a recursive algorithm. What are your input conditions? What is your base case?

3. Binomial coefficients (the number of ways that n elements can be selected out of a collection of m values) can be defined recursively by the following formula:

picture

Write a recursive procedure comb (n, m) to compute this value.

4. Rewrite the GCD algorithm as a recursive procedure.

5. List the sequence of moved the Towers of Hanoi problem would make in moving a tower of four disks from the source pole to the destination pole.

6. The *Fibonacci sequence* is another famous series that is defined recursively. The definition is as follows:

$$f_1 = 1$$
$$f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

Write in pseudo-code a function to compute the n^{th} Fibonacci number.

7. A *palindrome* is a word that reads the same both forward and backwards, such as rotor. An algorithm to determine whether or not a word is a palindrome can be expressed recursively. Simply strip off the first and last letters; if they are different, the word is not a palindrome. If they are, test the remaining string (after the first and last letters have been removed) to see if it is a palindrome. What is your base case for this procedure?

8. Extend the procedure you wrote in the previous question so that it can handle (by ignoring them) spaces and punctuation. An example of this sort is "A man, a plan, a canal, panama!"

9. Describe in pseudo-code, but do not implement, a recursive algorithm for finding the smallest number in an Array between two integer bounds (that is, a typical call such as smallest(a, 3, 7) would find the smallest value in the array a among those elements with indices between 3 and 7. What is the base case for your algorithm? How does the inductive case simplify the problem?

10. Using the algorithm you developed in the preceding question describe in pseudo-code a recursive sorting algorithm that works by finding the smallest element, swapping it into the first position, then finding the smallest element in the remainder, swapping it into the next position, and so on. What is the base case for your algorithm?

Programming Projects

1. Write a recursive procedure to print the octal (base-8) representation of an integer value.

2. Write a program that takes an integer argument and prints the value spelled out as English words. For example, the input - 3472 would produce the output "negative three thousand four hundred seventy-two". After removing any initial negative signs, this program is most easily handled in a recursive fashion. Numbers greater than millions can be printed by printing the number of millions (via a recursive call), then printing the work "million", then printing the remainder. Similarly with thousands, hundreds, and most values larger than twenty. Base cases are zero, numbers between 1 and 20.

3. Write a program to compute the fibonocci sequence recursively (see earlier exercise). Include a global variable fibCount that is incremented every time the function calls itself recursively. Using this, determine the number of calls for various values of N, such as n from 1 to 20. Can you determine a relationship between n and the resulting number of calls?

4. Do the same sort of analysis as described in the preceding question, but this time for the towers of Hanoi.

On the Web

Wikipedia (http://en.wikipedia.org) has a detailed entry for Algorithm that provides a history of the word, and several examples illustrating the way algorithms can be presented. The GCD algorithm is described in an entry "Euclidian Algorithm". Another interesting entry can be found on "recursive algorithms". The entry on "Fibonacci Numbers" provides an interesting history of the sequence, as well as many of the mathematical relationships of these numbers. The entry of "Towers of Hanoi" includes an animation that demonstrates the solution for N equal to 4. The more mathematically inclined might want to explore the entries on "Fractals" and "Mathematical Induction", and the relationship of these to recursion. Newtons method of computing square roots is described in an entry on "Methods of computing square roots". Another example of recursive art in commercial advertising is the "Morton Salt girl", who is spilling a container of salt, on which is displayed a picture of the Morton salt girl.

Chapter 3: Debugging, Testing and Proving Correctness

In this chapter we investigate tools that will help you to produce reliable and correct programs. During development of any program you will undoubtedly need to remove errors, and this will involve *debugging*. Once you believe your program (or portions of it) is correct you will want to increase your confidence in the program by systematic testing. Typically testing will uncover errors, which will lead to further debugging. Finally, the most powerful tool you can use to increase your confidence in a program or function is a proof of correctness. All of these tools are useful, and none should be considered to be a substitute for the others.

Hints on Debugging

There is no question that programming is a difficult task. Few nontrivial programs can be expected to run correctly the first time without error. Fortunately, there are many hints that can be used to help make debugging easier. Here are some of the more useful suggestions:

- Test small sections of a program in isolation. When you can identify a section of a program that is doing a specific task (this could be a loop or a function), write a small bit of code that tests this functionality. Gain confidence in the small pieces before considering the larger whole.

- When you see an error produced for a given input, try to find the simplest input that consistently reproduces the same error. Errors that cannot be reproduced are very difficult to eliminate, and simple inputs are much easier to reason about than more complex inputs.

- Once you have a simple test input that you know is handled incorrectly, play the role of the computer in your mind, and simulate execution of this test input. This will frequently lead you the location of your logical error.

- Think about what occurs before the point the error is noticed. An incorrect result is simply the symptom, and you must look earlier to find the cause.

- Use breakpoints or print statements to view the state of the computation in the middle. Starting with an input that produces the wrong result, try to reason backwards and determine what the values of variables would need to be to produce the output you see. Then check the state using break points or print statements. This can help isolate the portion of the program that contains the error.

- Don’t assume that just because one input is handled correctly that your program is correct.

- Use assertions and invariants to reason logically about your program.

- Most importantly, make sure you have the right mindset. Don't naturally assume that just because one section of a program works for most inputs, it must be correct. Question everything. Be open to any possibility. Look everywhere.

Assertions and Invariants

When analyzing algorithms two questions are important: Is the algorithm correct, and how fast does it run. In this chapter we are describing techniques that can be used to address the first. The second will be the subject of the next chapter.

One of the most powerful tools used to analyze algorithms is an *assertion*. An assertion is simply a comment that explains what you know to be true when execution reaches a specific point in the program. Assertions can include information you know from a specific statement, as well as information you know from tracing a flow of control.

```
int triangle(int a, int b, int c) {
  if (a == b)
    if (b == c)  return 1;
    else
      return 2;
  else if (b == c)
    return 2;
  else
    return 3;
}
```

For example, suppose you have written the function shown at the left. The program takes three integer values representing the sides of a triangle, and is intended to return a value that is 1 if the triangle is equilateral, 2 if it is isosceles (two sides equal in length), and 3 if it is scalene (no sides are equal). How would you increase your confidence that you have written the correct function?

```
int triangle(int a, int b, int c) {
  if (a == b)      /* we know a == b */
    if (b == c)  /* we know a==b & b==c */
      return 1;
    else /* we know a==b and b != c */
      return 2;
  else if (b == c) /* we know a!=b and b== c*/
    return 2;
  else /* we know a != b and b != c */
    return 3; /* ERROR! */
}
```

Assertions keep track of the information you know from following a path through the program. For example, after the first if statement we know that a is the same as b. After the second statement we know that b is the same as c, but we also remember the information from the first. Since a is equal to b and b is equal to c, it must be the case that all three are equal. Along the else path, however, we know that the original condition must be false, and so we can carry along that information. In this program keeping track of this information leads to the discovery of an error. After the last else statement we know that a is not equal to b and that b is not equal to c. But this does not imply, as the program is claiming, that all three values are different. It could still be the case that a is equal to c. The detection of this error has been simplified by explicitly keeping track of information using assertions.

Assertions become most useful when they are combined with loops. An assertion inside a loop is often termed an *invariant*, since it must describe a condition that does not vary during the course of executing the loop. To discover an invariant simply ask yourself why you think a program loop is doing the right thing, and then try to express "right thing" as a statement. For example, the function at right is computing the sum of an array of values. It does this by computing a partial sum, a sum up to a given point. So assertion number 3 is the easiest to discover. Once you discover assertion 3, then assertion 2 becomes clear – it is whatever is needed to ensure that assertion 3 will be true after the assignment. Assertion 4 is stating the expected result, while assertion 1 is asserting what is true before the loop begins.

```
double sum (double data[ ], int n) {
  double s = 0.0;
  /* 1. s is the sum of an empty array */
  for (int i = 0; i < n; i++) {
    /* 2. s is the sum of values from 0 to i-1 */
    s = s + data[i];
    /* 3. s is the sum of values from 0 to i */
  }
  /* 4. s is the sum from 0 to n-1 */
  return s;
}
```

Later in this chapter you will learn how to use invariants and assertions to prove that an algorithm or program is correct.

```
void bubbleSort (double data [ ], int n) {
  for (int i = n-1; i > 0; i--) {
    for (int j = 0; j < i; j++) {
      // data[j] is largest value in 0 .. j
      if (data[j] > data[j+1])
        swap(data, j, j+1)
      // data[j+1] is largest value in 0 .. j+1
    }
    data[i] is largest value in 0 .. i
  }
  // array is sorted
}
```

Notice how assertions require you to understand a high level description of what the algorithm is trying to do, and not simply a low level understanding of what the individual statements are doing. For example, consider the bubble sort algorithm shown at left. Bubble sort has two nested loops. The outer loop is the position of the array being filled. The inner loop is "bubbling" the largest element into this position. So once again at the end of the inner loop you want to make an assertion not only about the particular values at index locations j and j+1, but about everything the loop has seen before (namely, the elements with index values less than j). Once you identify this assertion, then the assertion at the beginning of the loop must be whatever is needed to prove the assertion at the end of the loop, and together these must be whatever is necessary to prove the assertion at the end of the outer loop.

Bubble Sort and Sorting

Bubble sort is the first of many sorting algorithms we will encounter in this book. Bubble sort is examined not because it is useful, it is not, but because it is very simple to analyze and understand. But there are many other sorting algorithms that are far more efficient. You should never use bubble sort for any real application, since there are many better alternatives.

In Worksheet 4 you will practice writing invariants and assertions that could be used to prove the correctness of a variety of programs.

Assertions and the assertion statement

Most programming languages include a statement, often termed the *assertion statement*, that performs a task that is similar but not exactly the same as the concept of the assertion described above. The assertions described here are written as comments, and are not executed by the program during execution. They need not be written in an executable form. The assertion statement, on the other hand, takes as argument an expression, and typically will halt execution and print an error message if the statement is not true. This can be very useful for verifying that input values satisfy whatever conditions are necessary for execution. Our square root example from the previous chapter, for example, could use an assertion statement to check that the input is a positive number:

```
double sqrt (double val) {
  assert (val >= 0); /* halt execution if value is not legal */
```

Because assertion statements are executed at run time, and halt execution if they are not satisfied, they should be used sparingly, but can be useful during debugging. The wikipedia entry for "assertions" contains a good discussion of assertions as used for program proofs compared with assertions used for routine error checking.

Introduction to the Binary Search Algorithm

An important algorithm that we will see many times in many different forms is the *binary search* algorithm. Binary search is similar to the way you guess a number if somebody says "I'm thinking of a value between 0 and 100. Can you find my number?" If you have played this game, you know the optimal strategy is to guess the value in the middle: "Is it larger or smaller than 50?" Suppose the other person answers "smaller". Then you again divide the range: "is it larger or smaller than 25?" By repeatedly apply this technique you very quickly find the hidden value. The binary search algorithm works in a similar fashion, but instead of numbers it looks for a specific value in an array of sorted numbers. Like the guessing game, it starts in the middle of the array, in one question eliminating one half of the possibilities, then in the next question breaking that subsection in half, and so on.

One version of the binary search algorithm is shown at right. Here n represents the number of values in the sorted data array, and the variable test is the value being searched for. You can verify that the algorithm is correct using the following invariant:

```
int binarySearch(double data[ ], int n, double test) {
  /* data is size n sorted array */
  int low = 0;
  int high = n;
  while (low < high) {
    mid = (low + high) / 2;
    if (data[mid] == test) return 1; /* true */
    if (data[mid] < testValue)
      low = mid + 1;
    else
      high = mid;
  }
  return 0; /* false */
}
```

Binary Search Invariant: All values with index positions smaller than low are less than test, and all values with index

positions larger than or equal to high are larger than or equal to test.

Prove to your satisfaction that the invariant is true at the beginning of the program (immediately after the assignments to low and high), and that it remains true at the end of the loop. Note that initially the variable high is not a legal index, and so the set of values with index positions larger than or equal to high is an empty set. In exercises at the end of the chapter we will use these to prove the algorithm is correct.

Testing and Boundary Cases

After you have translated an algorithm into a function or program, using assertions and invariants to increase your confidence in the correctness of your code, you should then always use *testing* to further assure yourself that your program works correctly. Testing is performed at many levels. You can (and should) test individual functions before you have a working application. This is termed *unit testing*. To perform a test, you need a main method. This is different from the main method you will eventually use for your program. A special purpose main method used in testing is termed a *test harness*. The test harness will feed one or more values into the function under test, and check the result. The box shows a test harness for the summation program given earlier in this chapter.

```
int main ( ) {
   double dataSetOne[ ] = {1.0, 2.0, 3.0, 4.0, 5.0};
   double sm = sum(dataSetOne, 5);
   println("result 1 should be 15, is: %g ", sm);
   return 0;
}
```

You should never content yourself with just one test case. A single test case cannot exercise all the possible ways a function can be used. As you develop test cases, think about the input data. If there are limits to the data, try to exercise values that are just at the edge of the limits. If the program uses conditional statements, use some data values that evaluate the condition true, and others that evaluate the condition false. This process is termed *boundary testing*. For example, think about testing the method min given earlier. Will the method find the correct value if the minimum is the first element in the array? If it is the last? If it is in the middle? If all values are the same? What about if there is only one element? What if there are no elements? Create a test case for each condition, and verify the result is correct. A collection of test cases is termed a *test suite*.

```
int main ( ) {
   double test1 [ ] = {1.0, 2.0, 3.0}; /* smallest first */
   double test2 [ ] = {3.0, 2.0, 1.0}; /* smallest last */
   double test3 [ ] = {2.0, 1.0, 3.0}; /* smallest middle */
   double test4 [ ] = {3.0, 1.0, 1.0, 2.0}; /* repeated smallest */
   double test5 [ ] = { }; /* no elements */
   double t1, t2, t3, t4, t5;
   t1 = min(test1, 3);
   t2 = min(test2, 3);
   t3 = min(test3, 3);
   t4 = min(test4, 4);
   printf("test cases 1, 2, 3, and 4: %g %g %g %g \n", t1, t2, t3, t4);
   t5 = min(test5, 0); /* should generate assertion error */
```

```
  printf("test case 5: %g \n", t5);
  return 0;
}
```

Notice that we expected one of these data sets to halt execution with an assertion error. After verifying that this is correct, you can comment out that particular test case while the others are processed.

In the test harness shown above we simply print the result, and count on the programmer running the test harness to verify the result. Sometimes it is possible to check the result directly. For example, if you were testing a method to compute a square root you could simply multiply the result by itself and verify that it produced the original number.

Question: Think about testing a sorting algorithm. Can you write a function that would test the result, rather than simply printing it out for the user to validate?

Once you are convinced that individual functions are working correctly, the next step is to combine calls on functions into more complex programs. Again you should perform testing to increase your confidence in the result. This is termed *integration testing*. Often you will uncover errors during integration testing. Once you fix these you should go back and re-execute the earlier test harness to ensure that the changes have not inadvertently introduced any new errors. This process is termed *regression testing*.

Some testing considers only the structure of the input and output values, and ignores the algorithm used to produce the result. This is termed *black box testing*. Other times you want to consider the structure of the function, for example to ensure that every if statement is exercised both with a value that makes it true and a value that makes it false. This is termed *white box testing*. Goals for white box testing should include that every statement is executed, and that every condition is evaluated both true and false. Other more complex test conditions can test the boundaries of a computation.

Testing alone should never be used to guarantee a program is working correctly. The famous computer scientist Edsger Dijkstra pointed out that testing can show the presence of errors but never their absence. Testing should be used in combination with logical thought, assertions, invariants, and proofs of correctness. All have a part to play in the development of a reliable program.

In worksheet 5 you will think about test cases for a variety of simple programs.

More on Program Proofs

We noted earlier in this chapter that the most powerful way to gain confidence in the correctness of a function or program is to develop a *proof* that the function is correct. In this section we will investigate this process in more detail, by examining another classic algorithm, a sorting algorithm named *selection sort*. Selection sort is easy to describe, which is why we study it. But like bubble sort it is also slow, so is not generally used in practice. In later lessons we will examine faster algorithms.

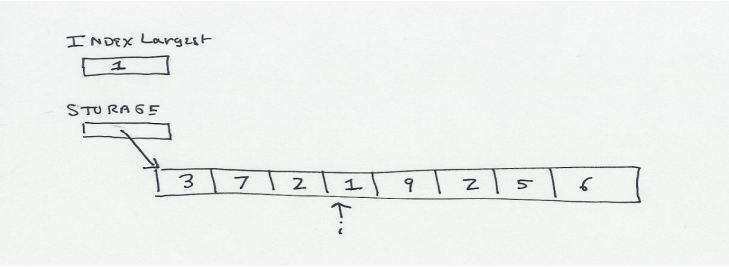
How to sort an array using selection sort: Find the index of the largest element in an array. Swap the largest value into the final location in the array. Then do the same with the next largest element. Then the next largest, and so on until you reach the smallest element. At that point the array will be sorted.

To develop this algorithm as executable code the first step is to isolate the smallest portion of the problem description that could be independently programmed, tested, and debugged. In this case you might select that first sentence: "find the index of the largest element in an array". How do you do that? The best way seems to be a loop.

```
double storage [ ]; /* size is n */
  ...
int indexLargest = 0;
for (int i = 1; i <=  n-1; i++) {
  if (storage[i] > storage[indexLargest])
    indexLargest = i;
}
```

How do we know this small bit of code is correct? As we discussed in the previous lessons, there are two general techniques that are used, and you should *always* use *both* of them. These two techniques are proofs and testing.

A *proof of correctness* is an informal argument that explains why you believe the code is correct. As you learned earlier, such proofs are built around *assertions*, which are statements that describe the relationships between variables when the computer reaches a point in execution. Using assertions, you simulate the execution of the algorithm in your mind, and argue both that the assertions are valid, and that they lead to the correct outcome.



In the code fragment above, we know that in the middle of execution the variable i represents some indefinite memory location. We have examined all values up to i, and indexLargest represents the largest value in that range. The values beyond index i have not yet been examined, and are therefore unknown. A drawing helps illustrate the relationships. What can you say about the relationship between i, indexLargest, and the data array? Invariants are written as comments, as in the following:

```
double storage [ ];
  ...
int position = n – 1;
int indexLargest = 0;
for (int i = 1; i <=  position; i++) {
  // inv: indexLargest is the index of the largest element in the range 0 .. (i-1)
  // (see picture)
  if (storage[i] > storage[indexLargest])
    indexLargest = i;
  // inv: indexLargest is the index of the largest element in the range 0 .. i
}
```

Notice how the invariant that comes after the if statement is a simple variation on the one that comes before. This is almost always the case. Once you find the pattern, then it becomes clear what the assertions must be that begin and end the loop. The first asserts what must be true before the loop starts, and the last must be what we want to be true after the loop finishes, which is normally the outcome we desire. These could be written as follows. We have numbered the invariants to help in the subsequent discussion:

```
double storage [ ];
   ...
int indexLargest = 0;
int position = n – 1;
// 1. indexLargest is the index of the largest element in the range 0 .. 0
for (int i = 1; i <=  position; i++) {
   // 2. indexLargest is the index of the largest element in the range 0 .. (i-1)
   if (storage[i] > storage[indexLargest])
      indexLargest = i;
   // 3. indexLargest is the index of the largest element in the range 0 .. i
}
// 4. indexLargest is index of largest element in  the range 0 .. (storage.length-1)
```

Identifying invariants is the first step. The next step is to form these into a proof that the program fragment produces the correct outcome. This is accomplished by a series of small arguments. Each argument moves from one invariant to the next, and use the knowledge you have of how the programming language changes the value of variables as execution progresses. Typically these arguments are very simple.

From invariant 2 to invariant 3. Here we assume invariant 2 is true and that we know nothing more about the variable i. But the if statement is checking the value of storage location i. If location i is the new largest element the value of indexLargest is changed. If it is not, then the previous largest value remains the correct index. Hence, if invariant 2 is true prior to the if statement, then invariant 3 must be true following the statement.

From invariant 3 back to invariant 2. This moves from the bottom of the loop back to the top. But during this process variable i is incremented by one. So if invariant 3 is true, and i is incremented, then invariant 2 is asserting the same thing.

All this may seem like a lot of work, but with practice loop invariants and proofs of correctness can become second nature, and seldom require as much analysis as we have presented here.

We return now to the development of the selection sort algorithm:

> **How to sort an array using selection sort:** Find the index of the largest element in an array. Swap this value into the final location in the array. Then do the same with the next largest element. Then the next largest, and so on until you reach the smallest element. At that point the array will be sorted.

Finding the largest value is a task that must be performed repeatedly. It is first performed to find the largest element in the array, and then the next largest, and then the one before that, and so on. So again a loop seems to be called for. Since we are looking for the largest value to fill a given position, let us name this loop variable position. The loop that is filling this variable looks as follows:

```
for (position = n – 1; position > 0; position--) {
  // find the largest element in 0 .. position
  int indexLargest = 0;
  ...
  // then swap into place
  swap(storage, indexLargest, position);
  // what is the invariant here?
}
```

Question: What can you say is true when execution reaches the comment at the end of the loop? What is true the first time the loop is executed? What can you say about the elements with index values larger than or equal to position after each succeeding iteration?

The selection sort algorithm combines the outer loop with the code developed earlier, wrapping this into a general method that we will name selectionSort.

In worksheets 6 and 7 you gain more experience working with invariants and proofs of correctness. These will introduce you to yet more sorting algorithms, termed gnome sort and insertion sort. The latter is very practical, and is often used to sort small arrays. (We will subsequently describe other algorithms that are even more efficient on large collections).

Proofs involving Multiple Functions

When a function calls another function, a proof assumes the called function will work as advertised. Assume that you have previously verified that the isPrime function works correctly. Use this fact to show the following prints the values of all the prime numbers from 2 to n:

```
void printPrimes (int n) {
  for (int i = 2; i <= n; i++) {
    if (isPrime(i))
      System.out.println("Number " + i + " is prime");
  }
}
```

A special case is the analysis of recursive functions. Here you first argue that the base case is correct, and then argue that the recursive case must be correct, assuming that the base case performs as defined. Provide assertions that demonstrate the following prints the binary representation of a number, assuming that the argument is positive.

```
void printBinary (int i) {
  assert (i >= 0);
  if ((i == 0) || (i == 1))
    print(i);
  else {
    printBinary(i/2);
    print(i%2);
  }
}
```

Recursion and Mathematical Induction

The analysis of recursive functions frequently mirrors the technique of *mathematical induction*. Both work by establishing a base case, then *reducing* other inputs until they reach the base case. In the printBinary function shown above, the base cases are the values zero and one. These are handed as a special case. All other values are handed by stripping off one binary digit, then invoking the function with a smaller number. Since on each iteration the number is smaller, it must eventually reach the base case.

To illustrate the similarities, recall the mathematical induction proof that the sum 1 + 2 + … + n is (n * (n + 1)) / 2. To prove this, we first verify that it is true for simple base cases, such as n=0, n=1, and n=2. Next, we will *assume* that it is true for all values smaller than n, and prove it is true for n. But if we have a sum from 1 to n, we can group all but the last term together.

$1 + 2 + \ldots + n = (1 + 2 + \ldots + (n-1)) + n$

Our induction hypothesis tells us that the first part is ((n - 1) * n) / 2. So the entire sum is ((n – 1) * n) / 2 + n. Simple arithmetic then shows that this is (n * (n + 1)) / 2. The argument shows that the result will hold for all positive integers

```
double exp (double a, int n) {
  if (n == 0) return 1.0;
  if (n == 1) return a;
  if (0 == n%2) return exp(a*a, n/2);
  else return a * exp(a, n-1);
}
```

Compare the structure of the preceding argument to the type of analysis you would do on a recursive function. Suppose, for example, we want to show that the function shown at right computes the value a raised to the n^{th} power. To prove this, we first verify that it works for simple base cases, such as n=0 and n=1. Next, we will assume that it works correctly for all values less than n, and prove that the function works correctly for n. To do this, we note two simple observations. If n is even, then a^n is the same as $(a*a)^{(n/2)}$. And if n is odd, then a^n is the same as $a * a^{(n-1)}$. Since both of these have exponents that are smaller than n, our assumption tells us that they will be correctly handled. Therefore the correct result is produced.

In later chapters we will encounter many recursive algorithms, and so it is important to become comfortable with the technique and understand the analysis of these functions.

Self Study Questions

1. What does it mean to say you are debugging a function?

2. What are some useful hints to help you debug a function or program?

3. What is an assertion?

4. What is an invariant?

5. Once you have identified assertions and invariants, how do you create a proof of correctness?

6. How is an assertion different from an assertion statement?

7. What is testing?

8. What is unit testing?

9. What is a test harness?

10. What is boundary testing?

11. What are some example boundary conditions?

12. What is a test suite?

13. What is integration testing?

14. What is regression testing and why is it performed?

15. What is black box testing?

16. What is white box testing?

17. Give an informal description in English (not code) explaining how the bubble sort algorithm operates.

18. Give a similar description of the selection sorting algorithm.

19. In what ways does the analysis of recursive algorithms mirror the idea of mathematical induction?

Exercises

1. Using only the specification (that is, black box testing), what are some test cases for the sqrt function described in the previous chapter?

2. What would be some good test cases for the min function described in the previous chapter?

3. Using assertions and invariants, prove that the min function described in the previous chapter produces the correct result.

4. The GCD function in the previous chapter required the input values to be positive, but did not check this condition. Add assertion statements that will verify this property.

5. Prove, by mathematical induction, that the sum of powers of 2 is one less than the next higher power. That is, for any non-negative integer n:

   sum I = 0 to n of 2^I equals $2^{n+1} - 1$

Analysis Exercises

1. Compare the selection sort algorithm with the bubble sort algorithm described in Lesson 4. How are they similar? How are they different?

2. What would be good test cases to exercise the bubble sort algorithm? Explain what property is being tested by our test cases.

3. What would be good test cases to exercise the selection sort algorithm?

4. From the specifications alone develop test cases for the triangle program. Then determine what value the program would produce for your test cases. Would your test cases have exposed the error?

5. What would be good test cases to exercise the binary search algorithm? Explain what property is being tested by each test case.

6. Using the invariants you discovered earlier, provide a proof of correctness for bubble sort. Do this by showing arguments that link each invariant to the next.

7. Using the invariant described in the section on binary search, provide a proof of correctness. Do this by showing the invariant is true at the start of execution, remains true after each execution of the while loop, and is still true when the while loop terminates.

8. In worksheet 4 you are asked to develop invariants for a number of functions. Having done so, number your invariants and provide short proofs for each path that leads from one invariant to the next.

9. Finish writing the invariants for selection sort, and then provide a proof of correctness by presenting arguments that link each invariant to the next.

10. Although Euclids GCD algorithm, described in the previous chapter, is one of the first algorithms, a proof of correctness is subtle. Traditionally it is divided into two steps. First, showing that the algorithm produces a value that is the divisor of the two input values. Second showing that it is the smallest such number. We will show how to do the first. The argument begins with the assumption that the divisor exists, even if we do not yet know its value. Let us call this divisor d. From basic arithmetic, we

know that if a and b are integers, and a > b, and d divides both a and b, then d must also divide (a-b). Using this hint, show that if d is a divisor of n and m when the function is first called, then d will be a divisor of n and m at each iteration of the while loop. The algorithm halts when n is equal to m, and so d is a divisor to both.

11. A sorting algorithm is said to be *stable* if two equal values retain their same relative ordering after sorting. Can you prove that bubble sort is stable? What about insertion sort?

12. The wikipedia entry for bubble sort includes a Boolean variable, named swapped, that is initially false inside the inner loop, and set to true if any two values are swapped. Explain why this can be used to improve the speed of the algorithm.

13. What is wrong with the following induction proof that for all positive numbers a and integer n, it must be true that a^{n-1} is 1. For the base case, have that for n = 1, a^{n-1} is a^0 which is 1. For the induction case assume it is true for 1, 2, 3, … n. To verify the condition for n+1, we have

$$a^{(n+1)-1} = a^n = (a^{n-1} * a^{n-1}) / a^{n-2} = 1 * 1 / 1 = 1$$

14. Explain why the following inductive argument cannot be used to demonstrate that all horses in a given corral are the same color. Suppose there is one horse in a correct, and that it is white. Thus, we have a base case, since for N equal to 1, all horses in the correct are white. Now add a second horse. The corral still contains the first horse, so we remove it. We have now reduced to our base case, and the horse that we removed was white, so both horses must be white. So for N equal to 2 we have our proof. We can continue in this fashion and show, no matter how many horses are in the corral, that they are all white.

Programming Assignments

1. Create a test harness to test the bubble sort algorithm. Feed the algorithm a variety of test cases and verify that it produces the correct result? Can you write a function that verifies the correct result instead of simply printing the values and asking the programmer if they are correct?

2. Do a similar task for the selection sort algorithm.

3. Compare empirically the running time of bubble sort and insertion sort. Do this by creating an array of size N containing random values, then time the sorting algorithm as it sorts the array. Print out the times for values of N ranging from 100 to 1000 in increments of 100.

4. In the next chapter we will show that the running time of both bubble sort and selection sort is proportional to the square of the number of elements. You can easily

see this empirically. Program a test harness that creates an array of random values of size n. Then time the execution of the bubble sort algorithm as it sorts this random array. Plot the running times for n = 100, 200, 300 up to 1000, and see what sort of graph this resembles.

On the Web

Wikipedia includes a very complete discussion of testing under the entry “Software Testing”. Related entries include “test case”, “unit testing”, “integration testing”, “white-box testing”, “black-box testing”, “debugging” and “software verification”. The entry for “bubble sort” includes an interactive demonstration, as well as detailed discussions of why it is not a good algorithm in practice. Selection sort is also the topic of a wikipedia entry. The wikipedia entry on “assertion(computing)” contains a link to an excellent article by computer science pioneer Tony Hoare on the development and use of assertions.

Chapter 4: Measuring Execution Time

You would think that measuring the execution time of a program would be easy. Simply use a stopwatch, start the program, and notice how much time it takes until the program ends. But this sort of measurement, called a wall-clock time, is for several reasons not the best characterization of a computer algorithm. A benchmark describes only the performance of a program on a specific machine on a given day. Different processors can have dramatically different performances. Even working on the same machine, there may be a selection of many alternative compilers for the same programming language. For this and other reasons, computer scientists prefer to measure execution time in a more abstract fashion.

Algorithmic Analysis

Why do dictionaries list words in alphabetical order? The answer may seem obvious, but it nevertheless can lead to a better understanding of how to abstractly measure execution time. Perform the following mind experiment. Suppose somebody were to ask you to look up the telephone number for "Chris Smith" in the directory for a large city. How long would it take? Now, suppose they asked you to find the name of the person with number 564-8734. Would you do it? How long do you think it would take?

```
Abby Smith    954-9845
Chris Smith   234-7832
Fred Smith    435-3545
Jaimie Smith  845-2395
Robin Smith   436-9834
```

Is there a way to quantify this intuitive feeling that searching an ordered list is faster than searching an unordered one? Suppose *n* represents the number of words in a collection. In an unordered list you must compare the target word to each list word in turn. This is called a *linear search*. If the search is futile; that is, the word is not on the list, you will end up performing *n* comparisons. Assume that the amount of time it takes to do a comparison is a constant. You don't need to know what the constant is; in fact it really doesn't matter what the constant is. What is important is that the total amount of work you will perform is *proportional* to *n*. That is to say, if you were to search a list of 2000 words it would take twice as long as it would to search a list of 1000 words. Now suppose the search is successful; that is, the word is found on the list. We have no idea where it might be found, so on average we would expect to search half the list. So the expectation is still that you will have to perform about *n/2* comparisons. Again, this value is proportional to the length of the list – if you double the length of the list, you would expect to do twice as much work.

What about searching an ordered list? Searching a dictionary or a phonebook can be informally described as follows: you divide the list in half after each comparison. That is, after you compare to the middle word you toss away either the first half or the last half of the list, and continue searching the remaining, and so on each step. In an earlier chapter

you learned that this is termed a *binary search*. If you have been doing the worksheets you have seen binary search already in worksheet 5. Binary search is similar to the way you guess a number if somebody says "I'm thinking of a value between 0 and 100. Can you find my number?" To determine the speed of this algorithm, we have to ask how many times can you divide a collection of *n* elements in half.

To find out, you need to remember some basic facts about two functions, the *exponential* and the *logarithm*. The exponential is the function you get by repeated multiplication. In computer science we almost always use powers of two, and so the exponential sequence is 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, and so on. The logarithm is the inverse of the exponential. It is the number that a base (generally 2) must be raised to in order to find a value. If we want the log (base 2) of 1000, for example, we know that it must be between 9 and 10. This is because 2^9 is 512, and 2^{10} is 1024. Since 1000 is between these two values, the log of 1000 must be between 9 and 10. The log is a very slow growing function. The log of one million is less than 20, and the log of one billion is less than 30.

It is the log that provides the answer to our question. Suppose you start with 1000 words. After one comparison you have 500, after the second you have 250, after the third 125, then 63, then 32, then 16, 8, 4, 2 and finally 1. Compare this to the earlier exponential sequence. The values are listed in reverse order, but can never be larger than the corresponding value in the exponential series. The log function is an approximation to the number of times that a group can be divided. We say *approximation* because the log function returns a fractional value, and we want an integer. But the integer we seek is never larger than 1 plus the ceiling (that is, next larger integer) of the log.

Logarithms

To the mathematician, a logarithm is envisioned as the inverse of the exponential function, or as a solution to an integral equation. However, to a computer scientist, the log is something very different. The log, base 2, of a value n is approximately equal to the number of times that n can be split in half. The word approximately is used because the log function yields a fractional value, and the exact figure can be as much as one larger than the integer ceiling of the log. But integer differences are normally ignored when discussing asymptotic bounds. Logs occur in a great many situations as the result of dividing a value repeatedly in half.

Performing a binary search of an ordered list containing *n* words you will examine approximately *log n* words. You don't need to know the exact amount of time it takes to perform a single comparison, as it doesn't really matter. Represent this by some unknown quantity *c,* so that the time it takes to search a list of *n* words is represented by *c * log n*. This analysis tells us is the amount of time you expect to spend searching if, for example, you double the size of the list. If you next search an ordered collection of *2n* words, you would expect the search to require *c * log (2n)* steps. But this is *c * (log 2 + log n*). The log 2 is just 1, and so this is nothing more than *c + c * log n* or *c * (1 + log n)*. Intuitively, what this is saying is that if you double the size of the list, you will expect to perform just one more comparison. This is considerably better than in the unordered list,

where if you doubled the size of the list you doubled the number of comparisons that you would expect to perform.

Big Oh notation

There is a standard notation that is used to simplify the comparison between two or more algorithms. We say that a linear search is a O(n) algorithm (read "big-Oh of n"), while a binary search is a O(log n) algorithm ("big-Oh of log n").

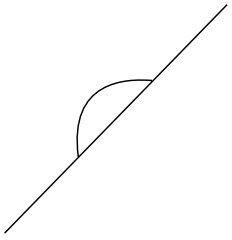
The idea captured by big-Oh notation is like the concept of the *derivative* in calculus. It represents the rate of growth of the execution time as the number of elements increases, or ∂-time versus ∂-size. Saying that an algorithm is O(n) means that the execution time is bounded by some constant times n. Write this as c*n. If the size of the collection doubles, then the execution time is c*(2n). But this is 2*(c*n), and so you expect that the execution time will double as well. On the other hand, if a search algorithm is O(log n) and you double the size of the collection, you go from c*(log n) to c*(log 2n), which is simply c + c*log n. This means that O(log n) algorithms are much faster than O(n) algorithms, and this difference only increases as the value of n increases.

A task that requires the same amount of time regardless of the input size is described as O(1), or "constant time". A task that is O(n) is termed a *linear* time task. One that is O(log n) is called *logarithmic*. Other terms that are used include *quadratic* for $O(n^2)$ tasks, and *cubic* for $O(n^3)$ algorithms.

What a big-oh characterization of an algorithm does is to abstract away unimportant distinctions caused by factors such as different machines or different compilers. Instead, it goes to the heart of the key differences between algorithms. The discovery of a big-oh characterization of an algorithm is an important aspect of algorithmic analysis. Due to the connection to calculus and the fact that the big-oh characterization becomes more relevant as *n* grows larger this is also often termed *asymptotic analysis*. We will use the two terms interchangeably throughout this text.

In worksheet 8 you will investigate the big-Oh characterization of several simple algorithms.

Summing Execution Times



If you have ever sat in a car on a rainy day you might have noticed that small drops of rain can remain in place on the windscreen, but as a drop gets larger it will eventually fall. To see why, think of the forces acting on the drop. The force holding the drop in place is surface tension. The force making it fall is gravity. If we imagine the drop as a perfect sphere, the surface tension is proportional to the square of the radius, while the force of gravity is proportional to the volume, which grows as the cube of the radius. What you observe is that no matter what constants are placed in front of these two

values, a cubic function (c * r^3 , i.e. gravity) will eventually grow larger than a quadratic function (d * r^2 , i.e. surface tension).

The same idea is used in algorithmic analysis. We say that one function *dominates* another if as the input gets larger the second will always be larger than the first, regardless of any constants involved. To see how this relates to algorithms consider the function to initialize an identity matrix. If you apply the techniques from the worksheets it is clear the first part is performing $O(n^2)$ work, while the second is O(n). So you might think that the algorithm is $O(n^2+n)$. But instead, the rule is that when summing big-Oh values you throw away everything except the dominating function. Since n^2 dominates n, the algorithm is said to be $O(n^2)$.

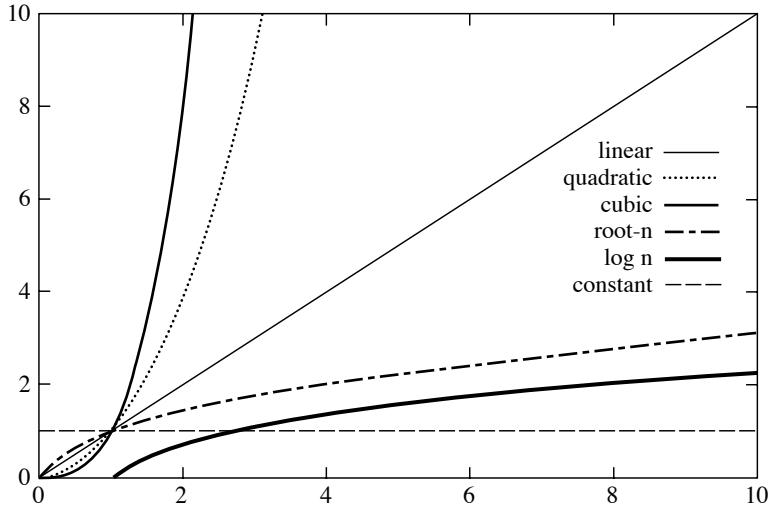
```
void makeIdentity (int m[N][N]) {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      m[i][j] = 0;
  for (int i = 0; i < N; i++)
    m[i][i] = 1;
}
```

Function	Common name	Running time
N!	Factorial	
2^n	Exponential	> century
N^d, d > 3	Polynomial	
N^3	Cubic	31.7 years
N^2	Quadratic	2.8 hours
N sqrt n		31.6 seconds
N log n		1.2 seconds
N	Linear	0.1 second
sqrt (n)	Root-n	3.2 * 10^{-4} seconds
Log n	Logarithmic	1.2 * 10^{-5} seconds
1	Constant	

What functions dominate each other? The table below lists functions in order from most costly to least. The middle column is the common name for the function. The third column can be used to illustrate why the dominating rule makes sense. Assume that an input is size 10^5, and you can perform 10^6 operations per second. The third column shows the approximate time it would take to perform a task if the algorithm is of the given complexity. So imagine that we had an algorithm had one part that was $O(n^2)$ and another that was O(n). The $O(n^2)$ part would be taking about 2.8 hours, while the O(n) part would contribute about 0.1 seconds. The smaller value gets overwhelmed by the larger.

In worksheet 9 you will explore several variations on the idea of dominating functions.

Each of the functions commonly found as execution times has a characteristic shape when plotted as a graph. Some of these are shown at right. With experience you should be able to look at a plot and recognize the type of curve it represents.



The Limits of big-Oh

The formal definition of big-Oh states that if f(n) is a function that represents the actual execution time of an algorithm, the algorithm is O(g(n)) if, for all values n larger than a fixed constant n_0, there exists a constant c, such that f(n) is always bounded by (that is, smaller than or equal to) the quantity c * g(n). Although this formal definition may be of critical importance to theoreticians, an intuitive felling for algorithm execution is of more practical importance.

The big-oh characterization of algorithms is designed to measure the behavior as inputs become ever larger. They may not be useful when n is small. In fact, for small values of n it may be that an $O(n^2)$ algorithm, for example, is faster than an O(n log n) algorithm, because the constants involve mask the asymptotic behavior. A real life example of this is matrix multiplication. There are algorithms for matrix multiplication that are known to be faster than the naïve $O(n^3)$ algorithm shown in worksheet 8. However, these algorithms are much more complicated, and since the value n (representing the number of rows or columns in a matrix) seldom gets very large, the classic algorithm is in practice faster.

Another limitation of a big-oh characterization is that it does not consider memory usage. There are algorithms that are theoretically fast, but use an excessively large amount of memory. In situations such as this a slower algorithm that uses less memory may be preferable in practice.

Using Big-Oh to Estimate Wall Clock Time

A big-Oh description of an algorithm is a characterization of the *change* in execution time as the input size changes. If you have actual execution timings ("wall clock time") for an algorithm with one input size, you can use the big-Oh to estimate the execution time for a different input size. The fundamental equation says that the ratio of the big-Oh's is equal to the ratio of the execution times. If an algorithm is O(f(n)), and you know that on input n_1 it takes time t_1, and you want to find the time t_2 it will take to process an input of size n_2, you create the equation

$$f(n_1) / f(n_2) = t_1 / t_2$$

To illustrate, suppose you want to actually perform the mind experiment given at the beginning of this chapter. You ask a friend to search for the phone number for "Chris Smith" in 8 pages of a phone book. Your friend does this in 10 seconds. From this, you can estimate how long it would take to search a 256 page phone book. Remembering that binary search is O(log n), you set up the following equation:

$$\log(8)/\log(256), \text{ which is } 3 / 8 = 10 / X$$

Solving for X gives you the answer that your friend should be able to find the number in about 24 seconds. Now you time your friend perform a search for the name attached to a

given number in the same 8 pages. This time your friend takes 2 minutes. Recalling that a linear search is O(n), this tells you that to search a 256 page phone could would require:

$$8/256 = 2 / X$$

Solving for X tells you that your friend would need about 64 minutes, or about an hour. So a binary search is really faster than a linear search.

In Worksheet 10 you will use this equation to estimate various different execution times.

Recursive Functions and Recurrence Relations

The analysis of recursive functions is slightly more complicated than the analysis of algorithms with loops. A useful technique is to describe the execution time using a *recurrence relation*. To illustrate, consider a function to print a positive integer value in binary format. Here n will represent the number of binary digits in the printed form. Because the argument is divided by 2 prior to the recursive call, it will have one fewer digit. Everything else, outside of the recursive call, can be performed in constant time. The base case can also be performed in constant time. If we let T(n) represent the time necessary to print an n-digit number, we have the following equation:

```
void printBinary (int v) {
  if ((v == 0) || (v == 1))
     print(n);
  else {
     printBinary(v/2);
     print(v%2);
  }
}
```

$$T(n) = T(n-1) + c$$
$$T(1) = c$$

Read this as asserting that the time for n is equal to the time for n-1 plus some unknown constant value. Whenever you see this relation you can say that the running time of the recursive function is O(n). To see why, remember that O(n) means that the actual running time is some constant times n plus other constant values. Let us write this as $c_1 n + c_2$. To verify that this is the solution to the equation, substitute for both sides, and show that the results are equal if the constants are in the correct relation. T(n) is $c_1 n + c_2$. We want this to be equal to $c_1(n-1) + c_2 + c$. But the latter is $c_1 n + c_2 + c - c_1$. Hence the two sides are equal if c is equal to c_1. In general we can just merge all constants into one constant value.

$T(n) = T(n-1) + c$	$O(n)$
$T(n) = T(n/2) + c$	$O(\log n)$
$T(n) = 2 * T(n/2) + c_a n + c_b$	$O(n \log n)$
$T(n) = 2 * T(n-1) + c$	$O(2^n)$

The four most common recurrence relations and their solutions are shown at left. Here we simply assert these solutions without proof. However, it is not difficult to check that the solutions are reasonable. For example, to verify the second you can observe the following

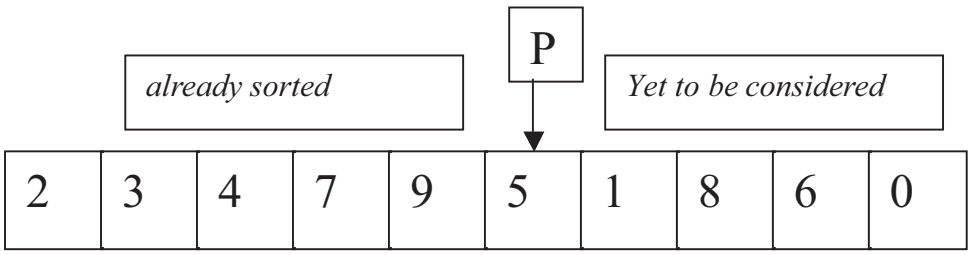
$$c_1 * \log n + c = c_1 * (\log (n/2)) + c = c_1 * (\log n - 1) + c = c_1 * \log n + c$$

In worksheet 11 you will use these forms to estimate the running time of various recursive algorithms. An analysis exercise at the end of this chapter asks you to verify the solution of each of these equations.

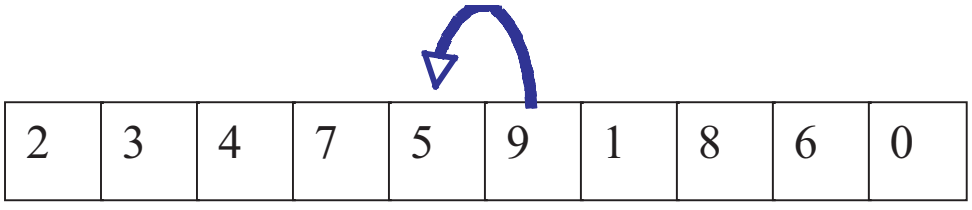
Best and Worst Case Execution Time

The bubble sort and selection sort algorithms require the same amount of time to execute, regardless of the input. However, for many algorithms the execution time will depend upon the input values, often dramatically so. In these situations it is useful to understand the best situation, termed the *best case time*, and the worst possibility, termed the *worst case time*. We can illustrate this type of analysis with yet another sorting algorithm. This algorithm is termed *insertion sort*. If you have been doing the worksheets you will remember this from worksheet 7.

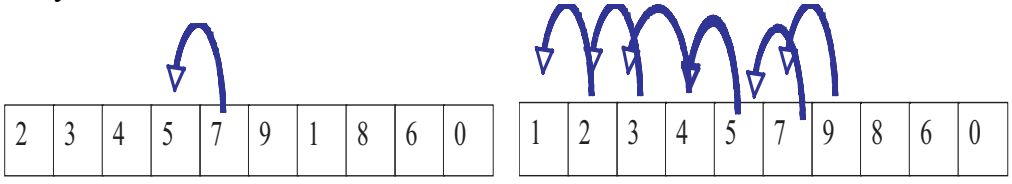
The insertion sort algorithm is most easily described by considering a point in the middle of execution:



Let p represent an index position in the middle of an array. At this point the elements to the left of p (those with index values smaller than p) have already been sorted. Those to the right of p are completely unknown. The immediate task is simply to place the one element that is found at location p so that it, too, will be part of the sorted portion. To do this, the value at location p is compared to its neighbor. If it is smaller, it is swapped with its neighbor. It is then compared with the next element, possibly swapped, and so on.



This process continues until one of two conditions occur. Either a value is found that is smaller, and so the ordering is established, or the value is swapped clear to the start of the array.



Since we are using a while loop the analysis of execution time is not as easy as it was for selection sort.

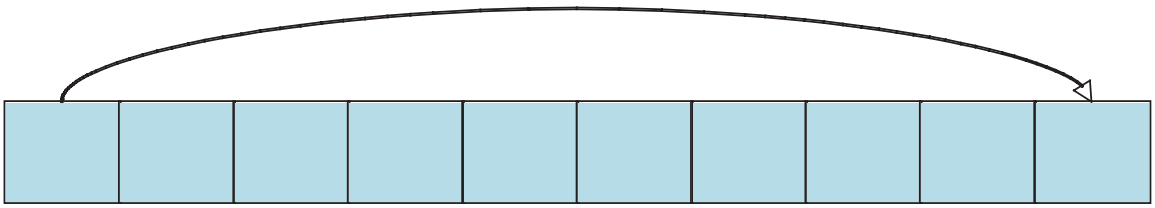
Question: What type of value will make your loop execute the fewest number of iterations? What type of value will make your loop execute the largest number of times? If there are n elements to the left of p, how many times will the loop iterate in the worst case? Based on your observations, can you say what type of input will make the insertion sort algorithm run most quickly? Can you characterize the running time as big-Oh? What type of input will make the insertion sort algorithm run most slowly? Can you characterize the running time as big-Oh?

We describe this difference in execution times by saying that one value represents the *best case* execution time, while the other represents the *worst case* time. In practice we are also interested in a third possibility, the *average case* time. But averages can be tricky – not only can the mathematics be complex, but the meaning is also unclear. Does average mean a random collection of input values? Or does it mean analyzing the sorting time for all permutations of an array and computing their mean?

Most often one is interested in a bound on execution time, so when no further information is provided you should expect that a big-Oh represents the worst case execution time.

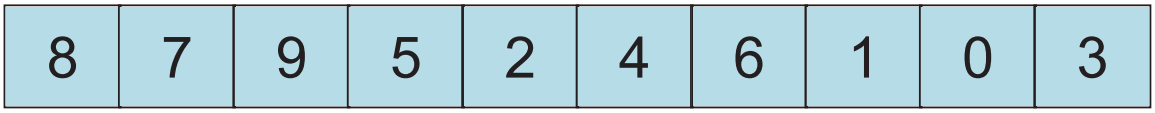
Shell Sort

In 1959 a computer scientist named Donald Shell argued that any algorithm that sorted by exchanging values with a neighbor must be $O(n^2)$. The argument is as follows. Imagine the input is sorted exactly backwards. The first value must travel all the way to the very end, which will requires n steps.

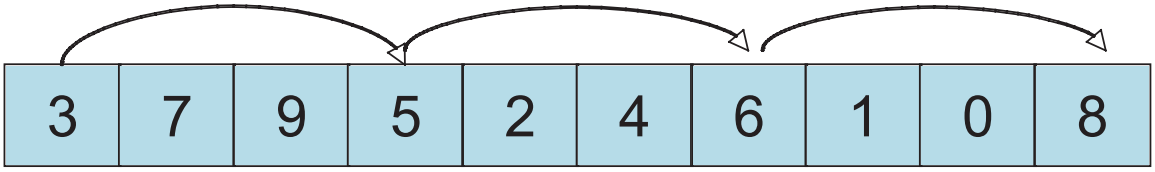


The next value must travel almost as far, taking n-1 steps. And so on through all the values. The resulting summation is $1 + 2 + \ldots + n$ which, we have seen earlier, results in $O(n^2)$ behavior.

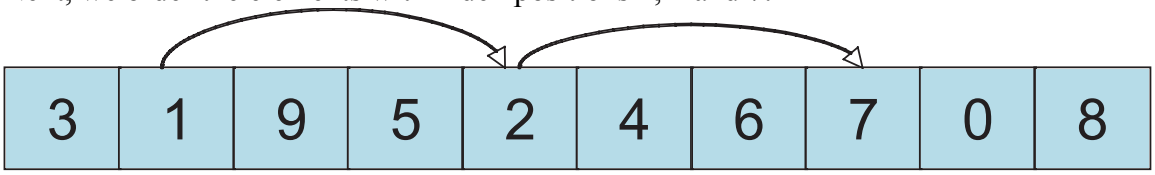
To avoid this inevitable limit, elements must "jump" more than one location in the search for their final destination. Shell proposed a simple modification too insertion sort to accomplish this. The outermost loop in the insertion sort procedure would be surrounded by yet another loop, called the *gap* loop. Rather than moving elements one by one, the outer loop would, in effect, perform an insertion sort every *gap* values. Thus, elements could jump *gap* steps rather than just a single step. For example, assume that we are sorting the following array of ten elements:



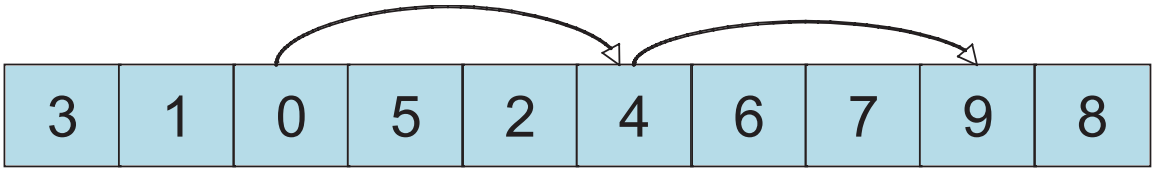
Imagine that we are sorting using a gap of 3. We first sort the elements with index positions 0, 3, 6,and 9 placing them into order:



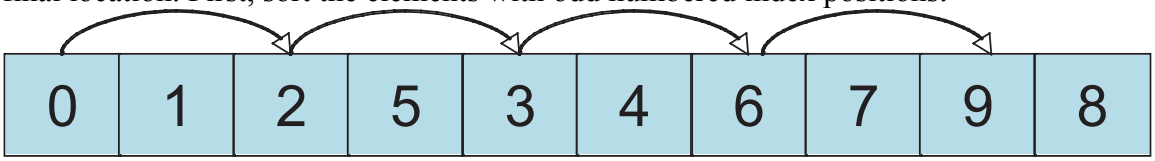
Next, we order the elements with index positions 1, 4 and 7:



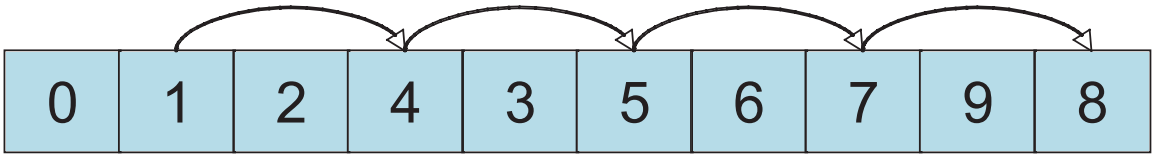
Finally values with index positions 2, 5 and 8 are placed in order.



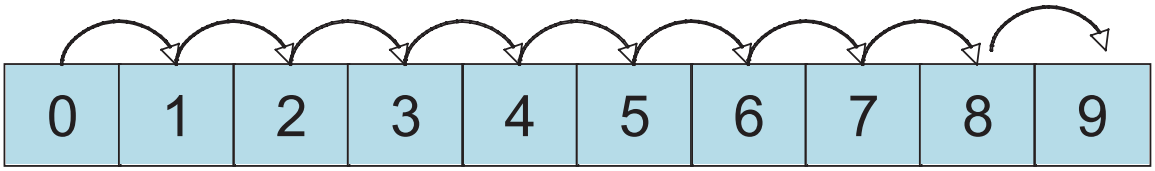
Next, we reduce the gap to 2. Now elements can jump two positions before finding their final location. First, sort the elements with odd numbered index positions:



Next, do the same with elements in even numbered index positions:



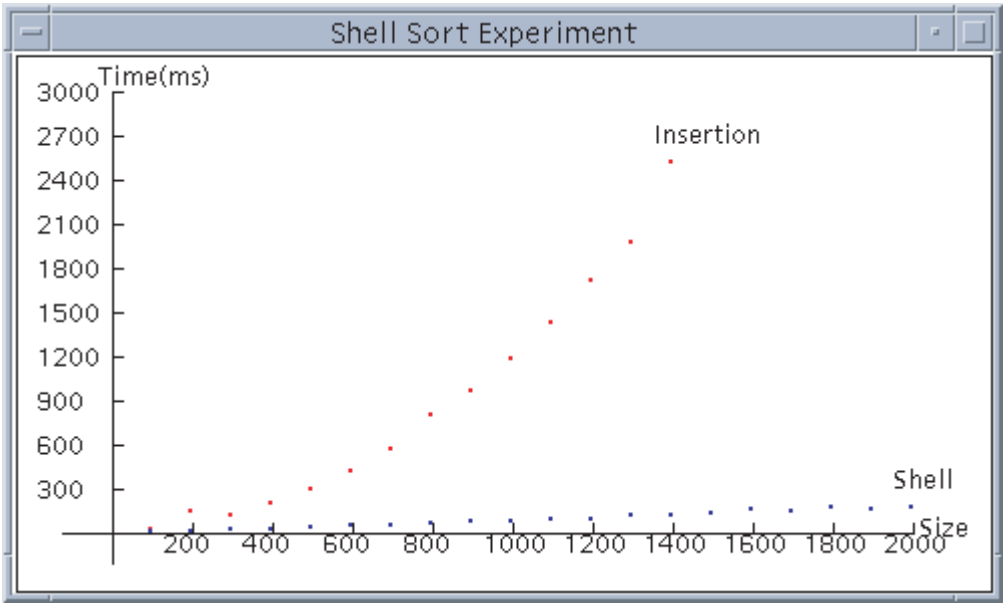
The final step is to sort with a gap of 1. This is the same as our original insertion sort. However, remember that insertion sort was very fast if elements were "roughly" in the correct place. Note that only two elements are now out of place, and they each move only one position.



The gap size can be any decreasing sequence of values, as long as they end with 1. Shell suggested the sequence n/2, n/4, n/8, … 1. Other authors have reported better results with different sequences. However, dividing the gap in half has the advantage of being extremely easy to program.

With the information provided above you should now be able to write the shell sort algorithm (see questions at the end of the chapter). Remember, this is simply an insertion sort with the adjacent element being the value gap elements away:

The analysis of shell sort is subtle, as it depends on the mechanism used in selecting the gaps. However, in practice it is considerably faster than simple insertion sort–despite the fact that shell sort contains three nested loops rather than two. The following shows the result of one experiment sorting random data values of various sizes.



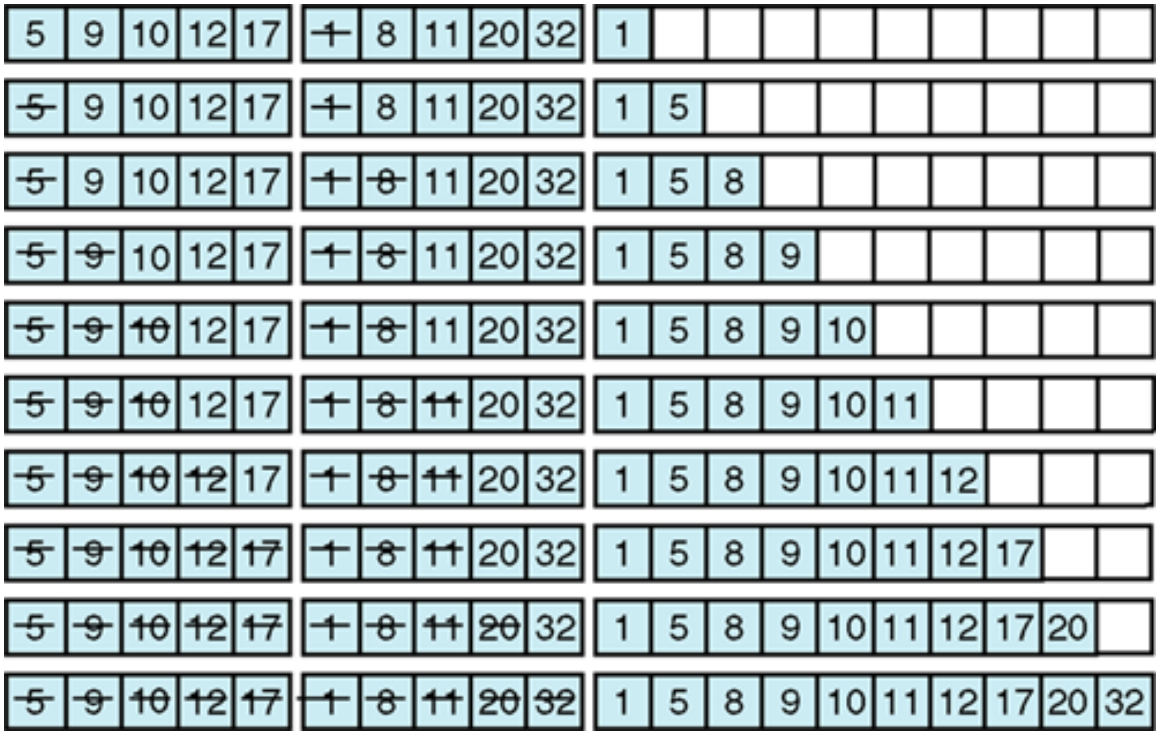
Merge Sort

In this chapter we have introduced several classic sorting algorithms: selection sort, insertion sort and shell sort. While easy to explain, both insertion sort and selection sort are $O(n^2)$ worst case. Many sorting algorithms can do better. In this lesson we will explore one of these.

Divide and Conquer

The general idea of dividing a problem into two roughly equal problems of the same form is known as the "divide and conquer" heuristic. We have earlier shown that the number of time a collection of size n can be repeatedly split in half is log n. Since log n is a very small value, this leads to many efficient algorithms. We will see a few of these in later lessons.

As with many algorithms, the intuition for merge sort is best understood if you start in the middle, and only after having seen the general situation do you examine the start and finish as special cases. For merge sort the key insight is that two already sorted arrays can be very rapidly merged together to form a new collection. All that is necessary is to walk down each of the original lists in order, selecting the smallest element in turn:

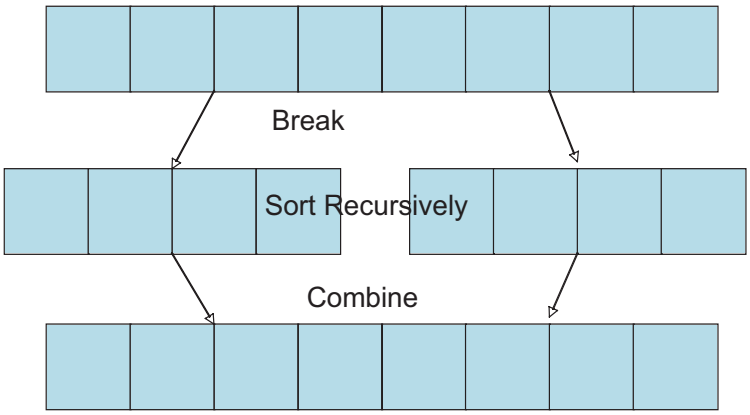


When you reach the end of one of the arrays (you cannot, in general, predict which list will end first), you must copy the remainder of the elements from the other.

Based on this description you should now be able to complete the implementation of the *merge* method. This you will do in worksheet 12. Let n represent the length of the result. At each step of the loop one new value is being added to the result. Hence the merge algorithm is O(n).

A Recursive Sorting Algorithm

So how do you take the observation that it is possible to quickly merge two ordered arrays, and from this produce a fast sorting algorithm? The key is to think *recursively*. Imagine sorting as a three-step process. In the first step an unordered array of length *n* is broken into two unordered arrays each containing approximately half the elements of the original. (Approximately, because if the size of the original is odd, one of the two will have one more element than the other). Next, each of these smaller lists is sorted by means of a recursive call. Finally, the two sorted lists are merged back to form the original.



Notice a key feature here that is characteristic of all recursive algorithms. We have solved a problem by assuming the ability to solve a "smaller" problem of the same form. The meaning of "smaller" will vary from task to task. Here, "smaller" means a smaller length array. The algorithm demonstrates how to sort an array of length *n*, assuming you know how to sort an array (actually, two arrays) of length *n/2*.

A function that calls itself must eventually reach a point where things are handled in a different fashion, without a recursive call. Otherwise, you have an infinite cycle. The case that is handled separately from the general recursive situation is called the *base case*. For the task of sorting the base case occurs when you have a list of either no elements at all, or just one element. Such a list is by definition already sorted, and so no action needs to be performed to place the elements in sequence.

7

With this background you are ready to write the mergeSort algorithm. The only actions are to separate the array into two parts, recursively sort them, and merge the results. If the length of the input array is sufficiently small the algorithm returns immediately. However, the merge operation cannot be performed in place. Therefore the merge sort algorithm requires a second temporary array, the same size as the original. The merge operation copies values into this array, then copies the array back into the original location. We can isolate the creation of this array inside the mergeSort algorithm, which simply invokes a second, internal algorithm for the actual sorting. The internal algorithm takes as arguments the lowest and highest index positions.

```
void mergeSort (double data [ ], int n) {
  double * temp = (double *) malloc (n * sizeof(double));
  assert (temp != 0); /* make sure allocation worked */
  mergeSortInternal (data, 0, n-1, temp);
  free (temp);
```

```
}

void mergeSortInternal (double data [ ], int low, int high, double temp [ ]) {
  int i, mid;
  if (low >= high) return; /* base case */
  mid = (low + high) / 2;
  mergeSortInternal(data, low, mid, temp); /* first recursive call */
  mergeSortInternal(data, mid+1, high, temp); /* second recursive call */
  merge(data, low, mid, high, temp); /* merge into temp */
  for (i = low; i <= high; i++)  /* copy merged values back */
    data[i] = temp[i];
}
```

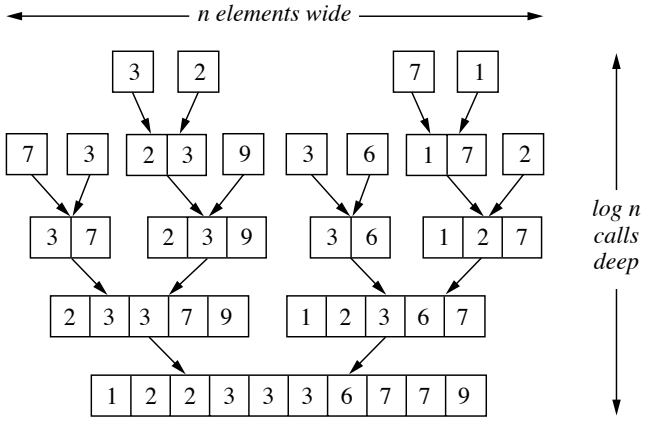
All that remains is to write the function merge, which you will do in Worksheet 12.

Algorithmic Analysis of Merge Sort

Recall that the analysis of recursive algorithms involves defining the execution time as a recurrence relation. Merge sort is breaking an array of size n into two smaller arrays, and then sorting the smaller arrays. So the equation is roughly

$T(N) = 2 * T(N / 2) + c_1*n + c_2$

You will remember that the solution to this relation is approximately n log n. Another way to think about this is the recursive calls on merge sort will be approximately O(log n) levels deep, and at each level it will be doing approximately O(n) operations.



An O(n log n) algorithm is a vast improvement over an $O(n^2)$ one. To illustrate, you can try executing the merge sort algorithm on arrays of various sizes, and comparing the execution times to that of selection sort. The following table shows some of the values in milliseconds that you would typically discover:

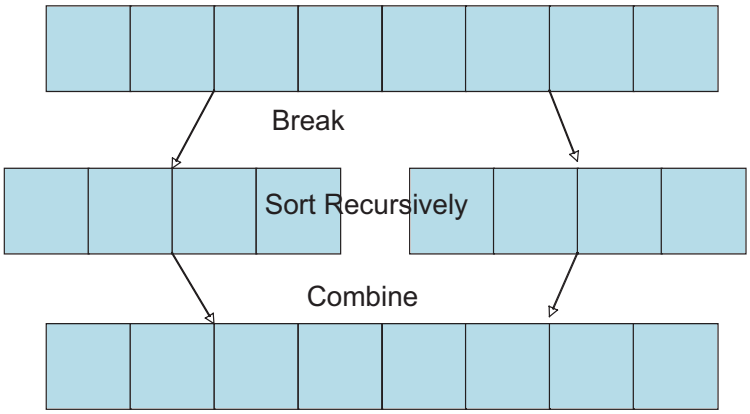
size	5000	6000	7000	8000	9000	10000
selection sort	104	165	230	317	402	500
merge sort	11	12	15	15	19	21

Is it possible to do better? The answer is yes *and* no. It is possible to show, although we will not do so here, that any algorithm that depends upon a comparison of elements must have at least an O(n log n) worst case. In that sense merge sort is about as good as we can expect. On the other hand, merge sort does have one annoying feature, which is that it uses extra memory. It is possible to discover algorithms that, while not asymptotically faster than merge sort, do not have this disadvantage. We will examine one of these in the next section.

Quick Sort

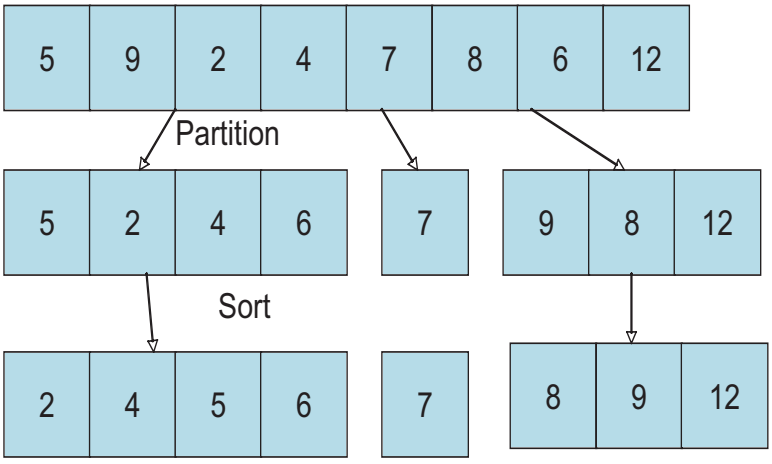
In this section we introduce *quick sort*. Quick sort, as the name suggests, is another fast sorting algorithm. Quick sort is recursive, which will give us the opportunity to once again use the recursive analysis techniques introduced earlier in this chapter. Quick sort has differing best and worst case execution times, similar in this regard to insertion sort. Finally, quick sort presents an unusual contrast to the merge sort algorithm.

The quick sort algorithm is in one sense similar to, and in another sense very different from the merge sort algorithm. Both work by breaking an array into two parts, recursively sorting the two pieces, and putting them back together to form the final result. Earlier we labeled this idea *divide and conquer*.



Where they differ is in the way that the array is broken into pieces, and hence the way that they are put back together. Merge sort devotes very little effort to the breaking apart phase, simply selecting the first half of the array for the first list, and the second half for the second. This means that relatively little can be said about the two sub-arrays, and significant work must be performed to merge them back together.

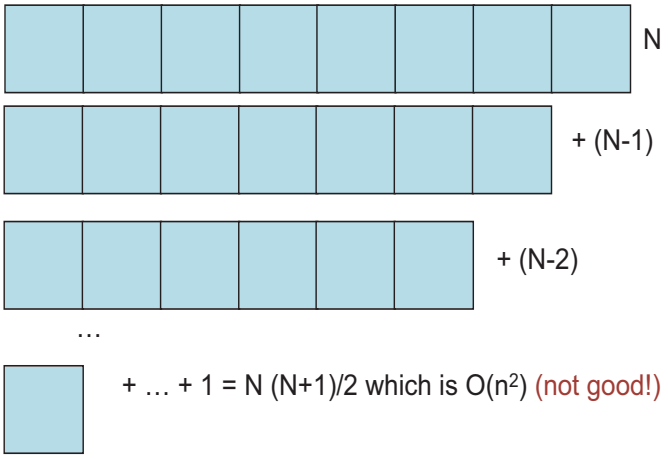
Quick sort, on the other hand, spends more time on the task of breaking apart. Quick sort selects one element, which is called the *pivot*. It then divides the array into two sections with the following property: every element in the first half is smaller than or equal to the pivot value, while every



element in the second half is larger than or equal to the pivot. Notice that this property does not guarantee sorting. Each of the smaller arrays must then be sorted (by recursively calling quick sort). But once sorted, this property makes putting the two sections back together much easier. No merge is necessary, since we know that elements in the first part of the array must all occur before the elements in the second part of the array.

Because the two sections do not have to be moved after the recursive call, the sorting can be performed in-place. That is, there is no need for any additional array as there is when using the merge sort algorithm. As with the merge sort algorithm, it is convenient to have the main function simply invoke an interior function that uses explicit limits for the upper and lower index of the area to be sorted.

```
void quickSort (double storage [ ], int n)
    { quickSortInternal (storage, 0, n-1); }

void quickSortInternal (double storage [ ], int low, int high) {
    if (low >= high) return; // base case
    int pivot = (low + high)/2; // one of many techniques
    pivot = partition(storage, low, high, pivot);
    quickSortInternal (storage, low, pivot-1); // first recursive call
    quickSortInternal (storage, pivot+1, high); // second recursive call
 }
```

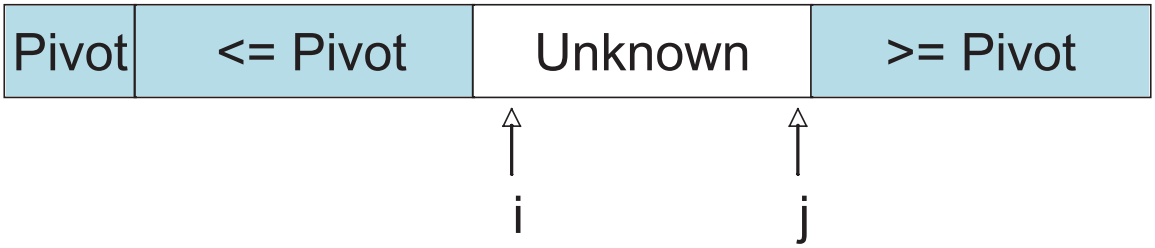


There is, however, a danger in this process. In merge sort it was easy to guarantee that the two halves were roughly equal size, yielding a fast O(n log n) process. In quick sort this is much more difficult to guarantee. If we are unlucky then in the worst case one partition contains no values, and the other is just one element smaller. This leads to poor $O(n^2)$ performance. Following the discussion of the partition algorithm we will describe some of the techniques that are used to try and avoid this bad behavior.

Partitioning

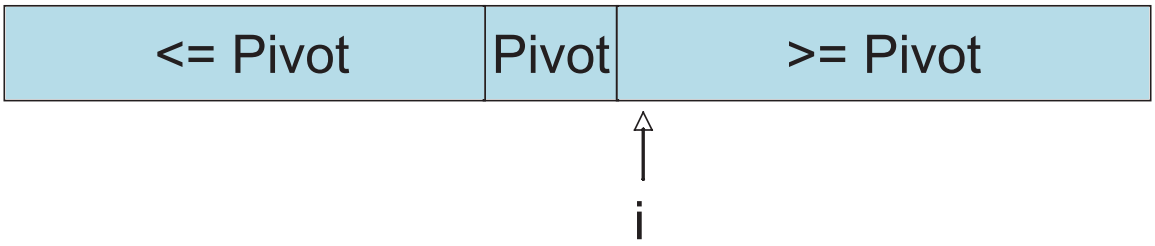
The process of dividing a portion of an array into two sections is termed *partitioning*. The limits of the partition are described by a pair of values: low and high. The first represents the lowest index in the section of interest, and the second the highest index. In addition there is a third element that is selected, termed the pivot. The first step is to swap the element at the pivot location and the first position. This moves the pivot value out of way of the partition step. The variable i is set to the next position, and the variable j to high. The heart of the partition algorithm is a while loop. The invariant that is going to be preserved is that all the elements with index values smaller than i are themselves smaller than or equal to the pivot, while all the elements with index values larger than j are

themselves larger than the pivot. At each step of the loop the value at the i position is compared to the pivot. If it is smaller or equal, the invariant is preserved, and the i position can be advanced:



Otherwise, the location of the j position is compared to the pivot. If it is larger then the invariant is also preserved, and the j position is decremented. If neither of these two conditions is true the values at the i and j positions can be swapped, since they are both out of order. Swapping restores our invariant condition.

The loop proceeds until the values of i and j meet and pass each other. When this happens we know that all the elements with index values less than i are less than or equal to the pivot. This will be the first section. All those elements with index values larger than or equal to i are larger than or equal to the pivot. This will be the second section. The pivot is swapped back to the top of the first section, and the location of the pivot is returned



The execution time of quicksort depends upon selecting a good pivot value. Many heuristics have been proposed for this task. These include

- Selecting the middle element, as we have done
- Selecting the first element. This avoids the initial swap, but leads to $O(n^2)$ performance in the common case where the input is already sorted
- Selecting a value at random
- Selecting the median of three randomly chosen values

From the preceding description you should be able to complete the partition algorithm, and so complete the quick sort algorithm. You will do this in worksheet 13.

Study Questions

1. What is a linear search and how is it different from a binary search?

2. Can a linear search be performed on an unordered list? Can a binary search?

3. If you start out with n items and repeatedly divide the collection in half, how many steps will you need before you have just a single element?

4. Suppose an algorithm is O(n), where n is the input size. If the size of the input is doubled, how will the execution time change?

5. Suppose an algorithm is O(log n), where n is the input size. If the size of the input is doubled, how will the execution time change?

6. Suppose an algorithm is $O(n^2)$, where n is the input size. If the size of the input is doubled, how will the execution time change?

7. What does it mean to say that one function *dominates* another when discussing algorithmic execution times?

8. Explain in your own words why any sorting algorithm that only exchanges values with a neighbor must be in the worst case $O(n^2)$.

9. Explain in your own words how the shell sort algorithm gets around this limitation.

10. Give an informal description, in English, of how the merge sort algorithm works.

11. What is the biggest advantage of merge sort over selection sort or insertion sort? What is the biggest disadvantage of merge sort?

12. In your own words given an informal explanation of the process of forming a partition.

13. Using the process of forming a partition described in the previous question, give an informal description of the quick sort algorithm.

14. Why is the pivot swapped to the start of the array? Why not just leave it where it is? Give an example where this would lead to trouble.

15. In what ways is quick sort similar to merge sort? In what ways are they different?

16. What does the quick sort algorithm do if all elements in an array are equal? What is the big-Oh execution time in this case?

17. Suppose you selected the first element in the section being sorted as the pivot. What advantage would this have? What input would make this a very bad idea? What would be the big-Oh complexity of the quick sort algorithm in this case?

18. Compare the partition median finding algorithm to binary search. In what ways are they similar? In what ways are they different?

Exercises

1. Suppose a algorithm takes 5 second to handle an input of 1000 elements. Fill in the following table with the approximate execution times assuming that the algorithm has the given big-Oh execution time.

	O(n)	$O(n^2)$	$O(n^3)$	O(n log n)	$O(2^n)$
1000	5	5	5	5	5
2000					
3000		45			
10000					

2. Suppose you have an n^2 algorithm that for n = 80 runs slightly longer than one hour. One day you discover an alternative algorithm that runs in time n log n. If you assume the constants of proportionality are about the same, about how long would you expect the new program to run?

3. Can you write the insertion portion of the insertion sort algorithm as a recursive routine? Rewrite the insertion sort function to use this new routine.

4. There is one previous algorithm you examined that also had different best and worst case execution times. What can you say about the execution times for the function isPrime?

```
int isPrime (int n) {
  for (int i = 2; i * i <= n; i++)
    if (0 == n % i)
      return 0; /* false */
  return 1; /* true */
}
```

5

Analysis Exercises

1. The interface file named time.h provides access to a millisecond timer, as well as a number of userful symbolic constants. You can use these to determine how long some actions takes, as follows:

```
# include <time.h>

double getMilliseconds( ) {
  return 1000.0 * clock( ) / CLOCKS_PER_SEC;
}

int main ( ) {
  double elapsed;

  elapsed = getMilliseconds();
… // perform a task
  elapsed = getMilliseconds() – elapsed;
  printf("Elapsed milliseconds = %g\n", elapsed);
}
```

Using this idea write a program that will determine the execution time for selectionSort for inputs of various sizes. Sort arrays of size n where n ranges from 1000 to 5000 in increments of 500. Initialize the arrays with random values. Print a table of the input sizes and execution times. Then plot the resulting values. Does the shape of the curve look like what you would expect from an n^2 algorithm?

2. Recall the function given in the previous chapter that you proved computed a^n. You can show that this function takes logarithmic number of steps as a function of n. This may not be obvious, since in some steps it only reduces the exponent by subtracting one. First, show that the function takes a logarithmic number of steps if n is a power of n. (Do you see why?). Next argue that every even number works by cutting the argument in half, and so should have this logarithmic performance. Finally, argue that every odd number will subtract one and become an even number, and so the number of times the function is called with an odd number can be no larger than the number of times it is called with an even number.

```
double exp (double a, int n) {
  if (n == 0) return 1.0;
  if (n == 1) return a;
  if (0 == n%2) return exp(a*a, n/2);
  else return a * exp(a, n-1);
}
```

3. A sorting algorithm is said to be *stable* if the relative positions of two equal elements are the same in the final result as in the original vector. Is insertion sort stable? Either give an argument showing that it is, or give an example showing that it is not.

4. Once you have written the merge algorithm for merge sort, provide invariants for your code and from these produce a proof of correctness.

5. Is merge sort stable? Explain why or why not.

6. Assuming your have proved the partition algorithm correct, provide a proof of correctness for the quick sort algorithm.

7. Is quick sort stable?

Programming Projects

1. Experimentally evaluate the running time of Shell Sort versus Insertion sort and Selection Sort. Are your results similar to those reported here?

2. Experimentally evaluate the running time of Merge sort to that of shell sort and insertion sort.

3. Rewrite the quick sort algorithm to select a random element in the section being sorted as the pivot. Empirically compare the execution time of the middle element as pivot version of quick sort to this new version. Are there any differences in execution speed?

4. Experimentally compare the execution time of the partition median finding algorithm to the naïve technique of sorting the input and selecting the middle element. Which one is usually faster?

On the Web

Wikipedia has an excellent discussion of big-Oh (sometimes called big-O) notation and its variations.

The Dictionary of Algorithms and Data Structures provided by the National Institute of Standards and Technology (http://www.nist.gov/dads/) has entries on binary and linear search, as well as most other standard algorithms.

The standard C library includes a version of quick sort, termed qsort. However, the interface is clumsy and difficult to use.

Chapter 5: Abstraction and Abstract Data Types

Abstraction is the process of trying to identify the most important or inherent qualities of an object or model, and ignoring or omitting the unimportant aspects. It brings to the forefront or highlights certain features, and hides other elements. In computer science we term this process *information hiding*.

Abstraction is used in all sorts of human endeavors. Think of an atlas. If you open an atlas you will often first see a map of the world. This map will show only the most significant features. For example, it may show the various mountain ranges, the ocean currents, and other extremely large structures. But small features will almost certainly be omitted.

A subsequent map will cover a smaller geographical region, and will typically possess more detail. For example, a map of a single continent (such as South America) may now include political boundaries, and perhaps the major cities. A map over an even smaller region, such as a country, might include towns as well as cities, and smaller geographical features, such as the names of individual mountains. A map of an individual large city might include the most important roads leading into and out of the city. Maps of smaller regions might even represent individual buildings.

Notice how, at each level, certain information has been included, and certain information has been purposely omitted. There is simply no way to represent all the details when an artifact is viewed at a higher level of abstraction. And even if all the detail could be described (using tiny writing, for example) there is no way that people could assimilate or process such a large amount of information. Hence details are simply left out.

Abstraction is an important means of controlling complexity. When something is viewed at an abstract level only the most important features are being emphasized. The details that are omitted need not be remembered or even recognized.

Another term that we often use in computer science for this process is *encapsulation*. An encapsulation is a packaging, placing items into a unit, or capsule. The key consequence of this process is that the encapsulation can be viewed in two ways, from the inside and from the outside. The outside view is often a description of the task being performed, while the inside view includes the implementation of the task.

An example of the benefits of abstraction can be seen by imagining calling the function used to compute the square root of a double precision number. The only information you typically need to know is the name of the function (say, sqrt), the argument types, and perhaps what it will do in exceptional conditions (say, if you pass it a negative number). The computation of the square root is actually a

```
double sqrt (double n) {
   double result = n/2;
   while (… ) {
     …
   }
   return result;
}
```

nontrivial process. As we described in Chapter 2, the function will probably use some sort of approximation technique, such as Newtons iterative method. But the details of how the result is produced have been abstracted away, or encapsulated within the function boundary, leaving you only the need to understand the description of the desired result.

Programming languages have various different techniques for encapsulation. The previous paragraph described how functions can be viewed as one approach. The function cleanly separates the outside, which is concerned with the "what" – what is the task to be performed, from the inside, the "how" – how the function produces its result. But there are many other mechanisms that serve similar purposes.

```
public interface Stack {
   public void push (Object a);
   public Object top ();
   public void pop ();
   public boolean isEmpty ();
};
```

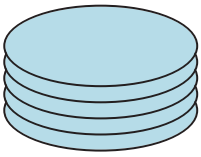
Some languages (but not C) include the concept of an *interface*. An interface is typically a collection of functions that are united in serving a common purpose. Once again, the interface shows only the function names and argument types (this is termed the function *signature*), and not the bodies, or implementation of these actions. In fact, there might be more than one implementation for a single interface. At a higher level, some languages include features such as modules, or packages. Here, too, the intent is to provide an encapsulation mechanism, so that code that is outside the package need only know very limited details from the internal code that implements the package.

Interface Files

The C language, which we use in this book, has an older and more primitive facility. Programs are typically divided into two types of files. Interface files, which traditionally end with a .h file extension, contain only function prototypes, interface descriptions for individual files. These are matched with an implementation file, which traditionally end with a .c file extension. Implementation files contain, as the name suggests, implementations of the functions described in the interface files, as well as any supporting functions that are required, but are not part of the public interface. Interface files are also used to describe standard libraries. More details on the standard C libraries are found in Appendix A.

Abstract Data Types

The study of data structures is concerned largely with the need to maintain *collections* of values. These are sometimes termed *containers*. Even without discussing how these collections can be implemented, a number of different types of containers can be identified purely by their purpose or behavior. This type of description is termed an *abstract data type*.

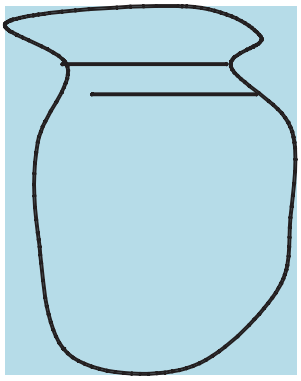


A simple example will illustrate this idea. A *stack* is a collection in which the order that elements are inserted is critically important. A metaphor, such as a stack of plates, helps in envisioning the idea. Only the topmost item in the stack (the topmost plate, for example), is accessible. To second element in the stack can only be accessed by first removing the topmost item. Similarly, when a new item is placed into the collection (a new plate placed on the stack, for example), the former top of the stack is now inaccessible, until the new top is removed.

Notice several aspects of this description. The first is the important part played by *metaphor*. The characteristics of the collection are described by appealing to a common experience with non-computer related examples. The second is that it is the *behavior* that is important in defining the type of collection, not the particular names given to the operations. Eventually the operations will be named, but the names selected (for example, push, add, or insert for placing an item on to the stack) are not what makes the collection into a stack. Finally, in order to be useful, there must eventually be a concrete realization, what we term an *implementation*, of the stack behavior. The implementation will, of course, use specific names for the operations that it provides.

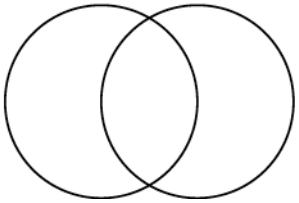
The Classic Abstract Data Types

There are several abstract data types that are so common that the study of these collection types is considered to be the heart of a fundamental understanding of computer science. These can be described as follows:

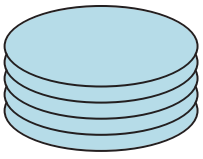


A *bag* is the simplest type of abstraction. A good metaphor is a bag of marbles. Operations on a bag include adding a value to the collection, asking if a specific value is or is not part of the collection, and removing a value from the collection.

A *set* is an extension of a bag. In addition to bag operations the set makes the restriction that no element may appear more than once, and also defines several functions that work with entire sets. An example would be set intersection, which constructs a new set consisting of values that appear in two argument sets. A venn diagram is a good metaphor for this type of collection.



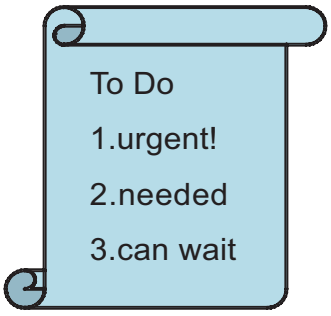
The order that elements are placed into a bag is completely unimportant. That is not true for the next three abstractions. For this reason these are sometimes termed *linear* collections. The simplest of these is the *stack*. The stack abstraction was described earlier. The defining characteristic of the stack is that it remembers the order that values were placed into the container. Values must be removed in a strict LIFO order (last-in, first-out). A stack of plates is the classic metaphor.

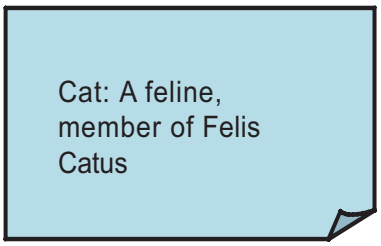


A *queue*, on the other hand, removes values in exactly the same order that they were inserted. This is termed FIFO order (first-in, first-out). A queue of people waiting in line to enter a theater is a useful metaphor.

The *deque* combines features of the stack and queue. Elements can be inserted at either end, and removed from either end, but only from the ends. A good mental image of a deque might be placing peas in a straw. They can be inserted at either end, or removed from either end, but it is not possible to access the peas in the middle without first removing values from the end.

A *priority queue* maintains values in order of importance. A metaphor for a priority queue is a to-do list of tasks waiting to be performed, or a list of patients waiting for an operating room in a hospital. The key feature is that you want to be able to quickly find the most important item, the value with highest priority.





A *map*, or *dictionary*, maintains pairs of elements. Each key is matched to a corresponding value. They keys must be unique. A good metaphor is a dictionary of word/definition pairs.

Each of these abstractions will be explored in subsequent chapters, and you will develop several implementations for all of them.

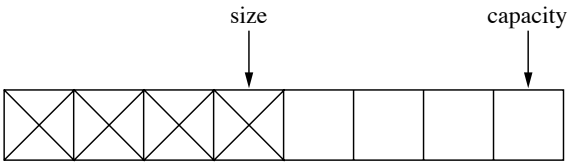
Implementations

Before a container can be used in a running program it must be matched by an *implementation*. The majority of this book will be devoted to explaining different implementations techniques for the most common data abstractions. Just as there are only a few classic abstract data types, with many small variations on a common theme, there are only a handful of classic implementation techniques, again with many small variations.

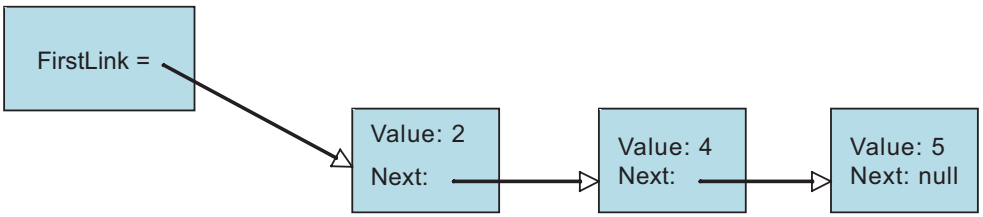
The most basic way to store a collection of values is an *array*. An array is nothing more than a fixed size block of memory, with adjacent cells in memory holding each element in the collection:

element 0	element 1	element 2	element 3	element 4

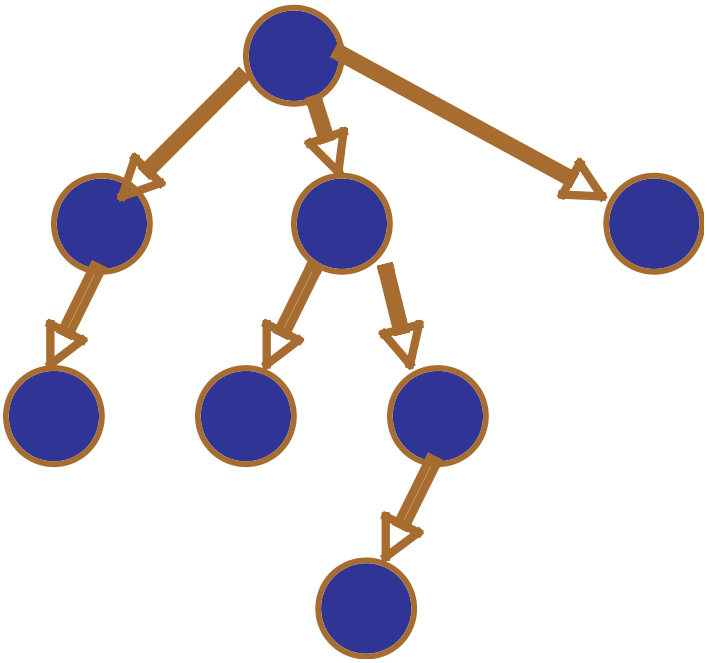
A disadvantage of the array is the fixed size, which typically cannot be changed during the lifetime of the container. To overcome this we can place one level of indirection between the user and the storage. A *dynamic array* stores the size and capacity of a container, and a pointer to an array in which the actual elements are stored. If necessary, the internal array can be increased during the course of execution to allow more elements to be stored. This increase can occur without knowledge of the user. Dynamic arrays are introduced in Worksheet 14, and used in many subsequent worksheets.



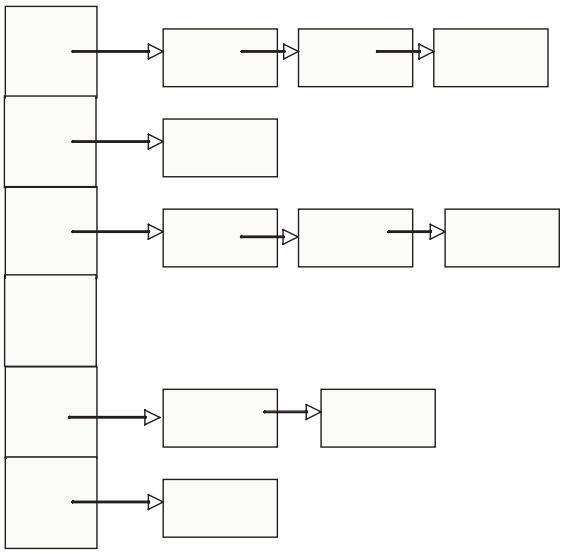
The fact that elements in both the array and the dynamic array are stored in a single block is both an advantage and a disadvantage. When collections remain roughly the same size during their lifetime the array uses only a small amount of memory. However, if a collection changes size dramatically then the block can end up being largely unused. An alternative is a *linked list*. In a linked list each element refers to (points to) the next in sequence, and are not necessary stored in adjacent memory locations.



Both the array and the linked list suffer from the fact that they are linear organizations. To search for an element, for example, you examine each value one after another. This can be very slow. One way to speed things up is to use a tree, specifically a *binary tree*. A search in a binary tree can be performed by moving from the top (the root) to the leaf (the bottom) and can be much faster than looking at each element in turn.



There are even more complicated ways to organize information. A *hash table*, for example, is basically a combination of an array and a linked list. Elements are assigned positions in the array, termed their *bucket*. Each bucket holds a linked list of values. Because each list is relatively small, operations on a hash table can be performed very quickly.



Many more variations on these themes, such as skip lists (a randomized tree structure imposed on a simple linked list), or heaps (a binary tree organized as a priority queue) will be presented as we explore this topic.

Some Words Concerning C

In the first part of this book we have made little reference to the type of values being held in a collection. Where we have used a data structure, such as the array used in sorting algorithms, the element type has normally been a simple floating-point number (a double). In the development that follows we want to generalize our containers so that they can maintain values of many different types. Unfortunately, C provides only very primitive facilities for doing so.

A major tool we will use is symbolic name replacement provided by the C preprocessor. This facility allows us to define a name and value pair. Prior to compilation, the C preprocessor will systematically replace each occurrence of the name with a value. For example, we will define our element type as follows:

```
# define TYPE double
```

Following this definition, we can use the name TYPE to represent the type of value our container will hold. This way, the user need only change the one definition in order to modify the type of value a collection can maintain.

Another feature of the preprocessor allows us to make this even easier. The statement #ifndef informs the preprocessor that any text between the statement and a matching #endef statement should only be included if the argument to the ifndef is *n*ot already *def*ined. The definition of TYPE in the interface file will be written as follows

```
# ifndef TYPE
# define TYPE double
# endif
```

These statements tell the preprocessor to only define the name TYPE if it has not already been defined. In effect, it makes double into our default value, but allows the user to provide an alternative, by preceding the definition with an alternative definition. If the user wants to define the element type as an integer, for example, they simple precede the above with a line

```
# define TYPE integer
```

A second feature of C that we make extensive use of in the following chapters is the equivalence between arrays and pointers. In particular, when an array must be allocated dynamically (that is, at run time), it is stored in a pointer variable. The function used to allocate memory is termed malloc. The malloc function takes as argument an integer representing the number of bytes to allocate. The computation of this quantity is made easier by another function, sizeof, which computes the size of its argument type.

You saw an example of the use of malloc and sizeof in the merge sort algorithm described in Chapter 4. There the malloc was used to create a temporary array. Here is another example. The following bit of code takes an integer value stored in the variable n, and allocates an array that can hold n elements of whatever type is represented by TYPE. Because the malloc function can return zero if there is not enough memory for the request, the result should always be checked. Because malloc returns an indetermined pointer type, the result must be cast to the correct form.

```
int n;
TYPE * data;
…
n = 42; /* n is given some value */
…
data = (TYPE *) malloc(n * sizeof(TYPE)); /* array of size n is allocated */
assert (data != 0);  /* check that allocation worked */
…
free (data);
```

Dynamically allocated memory must be returned to the memory manager using the free operation. You should always make sure that any memory that is allocated is eventually freed.

This is an idiom you will see repeatedly starting in worksheet 14. We will be making extensive use of pointers, but treating them as if they were arrays. Pointers in C can be indexed, exactly as if were arrays.

Study questions

1. What is abstraction? Give three examples of abstraction from real life.

2. What is information hiding? How is it related to abstraction?

3. How is encapsulation related to abstraction?

4. Explain how a function can be viewed as a type of encapsulation. What information is being hidden or abstracted away?

5. What makes an ADT description abstract? How is it different from a function signature or an interface?

6. Come up with another example from everyday life that illustrates the behavior of each of the six classic abstractions (bag, stack, queue, deque, priority queue, map).

7. For each of the following situations, describe what type of collection seems most appropriate, and why. Is order important? Is time of insertion important?

a. The names of students enrolled in a class.

b. Files being held until they can be printed on a printer.

c. URLs for recently visited web pages in a browser.

d. Names of patients waiting for an operating room in a hospital emergency ward.

e. Names and associated Employee records in a company database.

8. In what ways is a set similar to a bag? In what ways are they different?

9. In what ways is a priority queue similar to a queue? In what ways are they different?

10. Once you have completed worksheet 14, answer this question and the ones that follow. What is a dynamic array?

11. What does the term capacity refer to in a dynamic array?

12. What does the term size refer to in a dynamic array?

13. Can you describe the set of legitimate subscript positions in a dynamic array? Is this different from the set of legal positions in an ordinary array?

14. How does the add function in a dynamic array respond if the user enters more values than the current capacity of the array?

15. Suppose the user enters 17 numbers into a dynamic array that was initialized with a capacity of 5? What will be the final length of the data array? How many times will it have been expanded?

16. What is the algorithmic execution time for the `_dyArrayDoubleCapacity`, if n represents the size of the final data array?

17. Based on your answer to the previous question, what is the worst case algorithmic complexity of the function `dyArrayAdd`?

Analysis Exercises

1. Explain, in your own words, how calling a function illustrates the ideas of abstraction and information hiding. Can you think of other programming language features that can also be explained using these ideas?

2. Even without knowing the implementation, you can say that it would be an error to try to perform a *pop* operation on a stack if there has not been a preceding *push*. For

each of the classic abstractions describe one or more sequences of actions that should always produce an error.

3. This question builds on the work you began with the preceding question. Without even looking at the code some test cases can be identified from a specification alone, independent of the implementation. As you learned in Chapter 3, this is termed *black box testing*. For example, if you push an item on to a stack, then perform a pop, the item you just pushed should be returned. For each of the classic data structures come up with a set of test cases derived simply from the description.

4. Contrast an interface description of a container with the ADT description of behavior. In what ways is one more precise than the other? In what ways is it less precise?

5. Once you have completed worksheet 14 you should be able to answer the following. When using a partially filled array, why do you think the data array is doubled in size when the size exceeds the capacity? Why not simply increase the size of the array by 1, so it can accommodate the single new element? Think about what would happen if, using this alternative approach, the user entered 17 data values. How many times would the array be copied?

Programming Projects

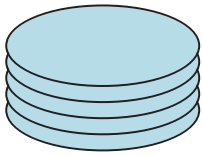
1. In worksheet 15 you explore why a dynamic array doubles the size of its internal array value when the size must be increased. In this project you can add empirical evidence to support the claim that this is a good idea. Take the dynamic array functions you developed in the worksheets 14 and 15 and add an additional variable to hold the "unit cost". As we described in worksheet 15, add 1 each time an element is added without reallocation, and add the size of the new array each time a reallocation occurs, plus 1 for the addition of the new element. Then write a main program that will perform 200 additions, and print the average cost after each insertion. Do the values remain relatively constant?

On the Web

Wikipedia (http://en.wikipedia.org/wiki/Abstract_data_type) has a good explanation of the concept of abstract data types. Links from that page explore most of the common ADTs. Another definition of ADT, as well as definitions of various forms of ADTs, can be found on DADS (http://www.nist.gov/dads/) the *Dictionary of Algorithms and Data Structures* maintained by the National Institute of Standards and Technology. Wikipedia has entries for many common C functions, such as malloc. There are many on-line tutorials for the C programming language. A very complete tutorial has been written in Brian Brown, and is mirrored at many sites. You can find this by googling the terms "Brian Brown C programming".

Chapter 6: Stacks

You are familiar with the concept of a *stack* from many everyday examples. For example, you have seen a stack of books on a desk, or a stack of plates in a cafeteria. The common characteristic of these examples is that among the items in the collection, the easiest element to access is the topmost value. In the stack of plates, for instance, the first available plate is the topmost one. In a true stack abstraction that is the *only* item you are allowed to access. Furthermore, stack operations obey the *last-in, first-out* principle, or LIFO. If you add a new plate to the stack, the previous topmost plate is now inaccessible. It is only after the newly added plate is removed that the previous top of the stack once more becomes available. If you remove all the items from a stack you will access them in reverse chronological order – the first item you remove will be the item placed on the stack most recently, and the last item will be the value that has been held in the stack for the longest period of time.



Stacks are used in many different types of computer applications. One example you have probably seen is in a web browser. Almost all web browsers have *Back* and *Forward* buttons that allow the user to move backwards and forwards through a series of web pages. The Back button returns the browser to the previous web page. Click the back button once more, and you return to the page before that, and so on. This works because the browser is maintaining a stack containing links to web pages. Each time you click the back button it removes one link from this stack and displays the indicated page.

The Stack Concept and ADT specification

Suppose we wish to characterize the stack metaphor as an abstract data type. The classic definition includes the following four operations:

Push (newEntry)	Place a new element into the collection. The value provided becomes the new topmost item in the collection. Usually there is no output associated with this operation.
Pop ()	Remove the topmost item from the stack.
Top ()	Returns, but does not remove, the topmost item from the stack.
isEmpty ()	Determines whether the stack is empty

Note that the names of the operations do not specify the most important characteristic of a stack, namely the LIFO property that links how elements are added and removed. Furthermore, the names can be changed without destroying the stack-edness of an abstraction. For example, a programmer might choose to use the names add or insert rather than push, or use the names peek or inspect rather than top. Other variations are also common. For example, some implementations of the stack concept combine the pop and top operations by having the pop method return the value that has been removed from the stack. Other implementations keep these two tasks separate, so that the only access to the topmost element is through the function named top. As long as the

fundamental LIFO behavior is retained, all these variations can still legitimately be termed a stack.

Finally, there is the question of what to do if a user attempts to apply the stack operations incorrectly. For example, what should be the result if the user tries to pop a value from an empty stack? Any useful implementation must provide some well-defined behavior in this situation. The most common implementation technique is to throw an exception or an assertion error when this occurs, which is what we will assume. However, some designers choose to return a special value, such as null. Again, this design decision is a secondary issue in the development of the stack abstraction, and whichever design choice is used will not change whether or not the collection is considered to be a stack, as long as the essential LIFO property of the collection is preserved.

The following table illustrates stack operations in several common languages:

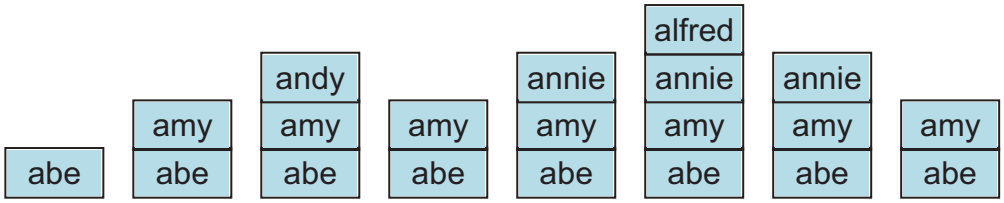
	Java class Stack	C++ stack adapter	Python list
push	push(value)	push(value)	lst.append(value)
pop	pop()	pop	Del lst[-1]
top	peek()	top()	lst[-1]
isEmpty	empty()	empty()	len(lst) == 0

In a pure stack abstraction the only access is to the topmost element. An item stored deeper in the stack can only be obtained by repeatedly removing the topmost element until the value in question rises to the top. But as we will see in the discussion of implementation alternatives, often a stack is combined with other abstractions, such as a dynamic array. In this situation the data structure allows other operations, such as a search or direct access to elements. Whether or not this is a good design decision is a topic explored in one of the lessons described later in this chapter.

To illustrate the workings of a stack, consider the following sequence of operations:

push(“abe”)
push(“amy”)
push(“andy”)
pop()
push(“anne”)
push(“alfred”)
pop()
pop()

The following diagram illustrates the state of the stack after each of the eight operations.

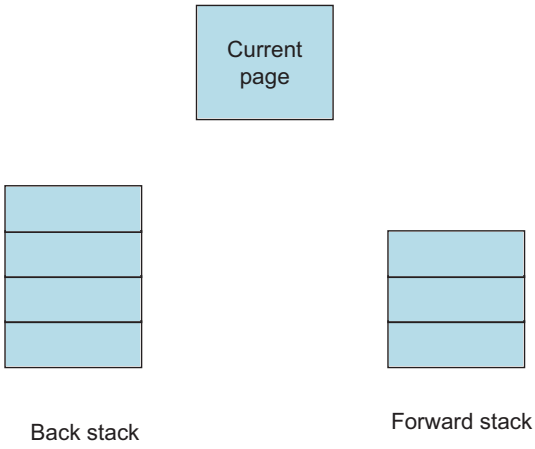


Applications of Stacks

Back and Forward Buttons in a Web Browser

In the beginning of this chapter we noted how a stack might be used to implement the Back button in a web browser. Each time the user moves to a new web page, the current web page is stored on a stack. Pressing the back button causes the topmost element of this stack to be popped, and the associated web page is displayed.

However, that explanation really provided only half the story. To allow the user to move both forward and backward two stacks are employed. When the user presses the back button, the link to the current web page is stored on a separate stack for the forward button. As the user moved backward through previous pages, the link to each page is moved in turn from the back to the forward stack.



When the user pushes the forward button, the action is the reverse of the back button. Now the item from the forward stack is popped, and becomes the current web page. The previous web page is pushed on the back stack.

Question: The user of a web browser can also move to a new page by selecting a hyperlink. In fact, this is probably more common than using either the back or forward buttons. When this happens how should the contents of the back and forward stacks be changed?

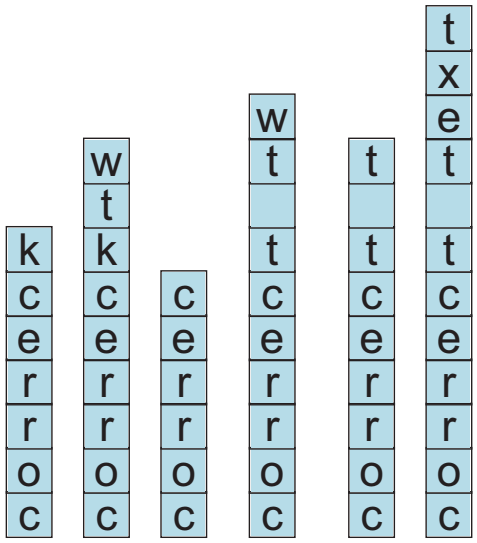
Question: Web browsers often provide a *history* feature, which records all web pages accessed in the recent past. How is this different from the back stack? Describe how the history should change when each of the three following conditions occurs: (a) when the user moves to a new page by pressing a hyperlink, (b) when the user restores an old page by pressing the back button, and (c) when the user moves forward by pressing the forward button.

Buffered Character Input

An operating system uses a stack in order to correctly process backspace keys in lines of input typed at a keyboard. Imagine that you enter several keys and then discover a mistake. You press the backspace key to move backward over a previously entered character. Several backspaces may be used in turn to erase more than one character. If we use < to represent the backspace character, imagine that you typed the following:

```
correcktw<<<<t tw<ext
```

The operating system function that is handling character input will arrive at the correct text because it stores the characters as they are read in a stack-like fashion. Each non-backspace character is simply pushed on the stack. When a backspace is typed, the topmost character is popped from the stack and erased.



Question: What should be the effect if the user enters a backspace key and there are no characters in the input?

Activation Record Stack

Another example of a stack that we discussed briefly in an earlier chapter is the activation record stack. This term describes the spaced used by a running program to store parameters and local variables. Each time a function or method is invoked, space is set aside for these values. This space is termed an activation record. For example, suppose we execute the following function

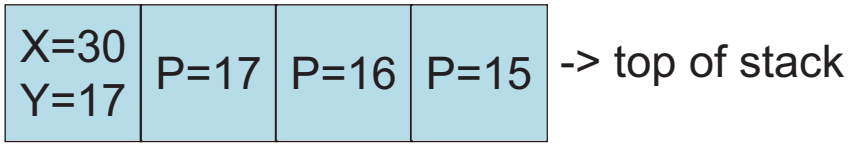
```
void a (int x)
int y
  y = x - 23;
  y = b (y)
```

When the function a is invoked the activation record looks something like the following:

Imagine that b has the following recursive definition

```
int b (int p)
  if (p < 15) return 1;
  else return 1 + b(p-1)
```

Each time the function b is invoked a new activation record is created. New local variables and parameters are stored in this record. Thus there may be many copies of a local variable stored in the stack, one for each current activation of the recursive procedure.



Functions, whether recursive or not, have a very simple execution sequence. If function a calls function b, the execution of function a is suspended while function b is active. Function b must return before function a can resume. If function b calls another function, say c, then this same pattern will follow. Thus, function calls work in a strict stack-like fashion. This makes the operation of the activation record stack particularly easy. Each time a function is called new area is created on the activation record stack. Each time a function returns the space on the activation record stack is popped, and the recovered space can be reused in the next function call.

Question: What should (or what does) happen if there is no available space in memory for a new activation record? What condition does this most likely represent?

Checking Balanced Parenthesis

A simple application that will illustrate the use of the stack operations is a program to check for balanced parenthesis and brackets. By balanced we mean that every open parenthesis is matched with a corresponding close parenthesis, and parenthesis are properly nested. We will make the problem slightly more interesting by considering both parenthesis and brackets. All other characters are simply ignored. So, for example, the inputs (x(y)(z)) and a({(b)}c) are balanced, while the inputs w)(x) and p({(q)r)} are not.

To discover whether a string is balanced each character is read in turn. The character is categorized as either an opening parenthesis, a closing parenthesis, or another type of character. Values of the third category are ignored. When a value of the first category is encountered, the corresponding close parenthesis is stored on the stack. For example, when a "(" is read, the character ")" is pushed on the stack. When a "{" is encountered, the character pushed is "}". The topmost element of the stack is therefore the closing value we expect to see in a well balanced expression. When a closing character is encountered, it is compared to the topmost item in the stack. If they match, the top of the stack is popped and execution continues with the next character. If they do not match an error is reported. An error is also reported if a closing character is read and the stack is empty. If the stack is empty when the end of the expression is reached then the expression is well balanced.

The following illustrates the state of the stack at various points during the processing of the expression a (b { d e [f] g { h } I } j k) l m.

picture

The following illustrates the detection of an error when a closing delimiter fails to match the correct opening character:

picture

Another error occurs when there are opening delimiters but no closing character:

picture

Question: Show the state of the stack after each character is read in the following expression: (a b { c } d ([e [f] g]) (j))

Evaluating Expressions

Two standard examples that illustrate the utility of the stack expression involve the evaluation of an arithmetic expression. Normally we are used to writing arithmetic expressions in what is termed *infix form*. Here a binary operator is written between two arguments, as in 2 + 3 * 7. Precedence rules are used to determine which operations should be performed first, for example multiplication typically takes precedence over addition. Associativity rules apply when two operations of the same precedence occur one right after the other, as in 6 – 3 – 2. For addition, we normally perform the left most operation first, yielding in this case 3, and then the second operation, which yields the final result 1. If instead the associativity rule specified right to left evaluation we would have first performed the calculation 3 – 2, yielding 1, and then subtracted this from 6, yielding the final value 5. Parenthesis can be used to override either precedence or

associativity rules when desired. For example, we could explicitly have written 6 – (3 – 2).

The evaluation of infix expressions is not always easy, and so an alternative notion, termed *postfix notation*, is sometimes employed. In postfix notation the operator is written after the operands. The following are some examples:

Infix	2 + 3	2 + 3 * 4	(2 + 3) * 4	2 + 3 + 4	2 - (3 – 4)
Postfix	2 3 +	2 3 4 * +	2 3 + 4 *	2 3 + 4 +	2 3 4 - -

Notice that the need for parenthesis in the postfix form is avoided, as are any rules for precedence and associativity.

We can divide the task of evaluating infix expressions into two separate steps, each of which makes use of a stack. These steps are the conversion of an infix expression into postfix, and the evaluation of a postfix expression.

Conversion of infix to postfix

To convert an infix expression into postfix we scan the value from left to right and divide the tokens into four categories. This is similar to the balanced parenthesis example. The categories are left and right parenthesis, operands (such as numbers or names) and operators. The actions for three of these four categories is simple:

Left parenthesis	Push on to stack
Operand	Write to output
Right parenthesis	Pop stack until corresponding left parenthesis is found. If stack becomes empty, report error. Otherwise write each operator to output as it is popped from stack

The action for an operator is more complex. If the stack is empty or the current top of stack is a left parenthesis, then the operator is simply pushed on the stack. If neither of these conditions is true then we know that the top of stack is an operator. The precedence of the current operator is compared to the top of the stack. If the operator on the stack has higher precedence, then it is removed from the stack and written to the output, and the current operator is pushed on the stack. If the precedence of the operator on the stack is lower than the current operator, then the current operator is simply pushed on the stack. If they have the same precedence then if the operator associates left to right the actions are as in the higher precedence case, and if association is right to left the actions are as in the lower precedence case.

Evaluation of a postfix expression

The advantage of postfix notation is that there are no rules for operator precedence and no parenthesis. This makes evaluating postfix expressions particularly easy. As before, the postfix expression is evaluated left to right. Operands (such as numbers) are pushed on the stack. As each operator is encountered the top two elements on the stack are removed, the operation is performed, and the result is pushed back on the stack. Once all the input has been scanned the final result is left sitting in the stack.

Question: What error conditions can arise if the input is not a correctly formed postfix expression? What happens for the expression 3 4 + + ? How about 3 4 + 4 5 + ?

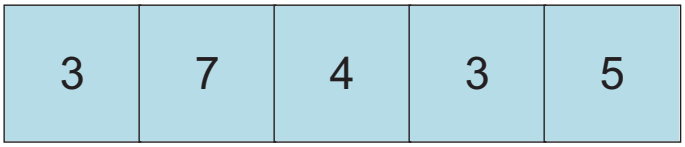
Stack Implementation Techniques

In the worksheets we will discuss two of the major techniques that are typically used to create stacks. These are the use of a dynamic array, and the use of a linked list. Study questions that accompany each worksheet help you explore some of the design tradeoffs a programmer must consider in evaluating each choice. The self-study questions given at the end of this chapter are intended to help you measure your own understanding of the material. Exercises and programming assignments that follow the self-study questions will explore the concept in more detail.

Worksheet 14	Introduction to the Dynamic Array
Worksheet 16	Dynamic Array Stack
Worksheet 17	Introduction to Linked List, Linked List Stack

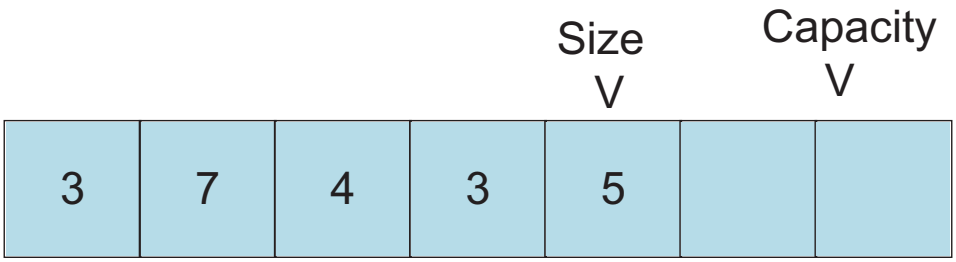
Building a Stack using a Dynamic Array

An array is a simple way to store a collection of values:

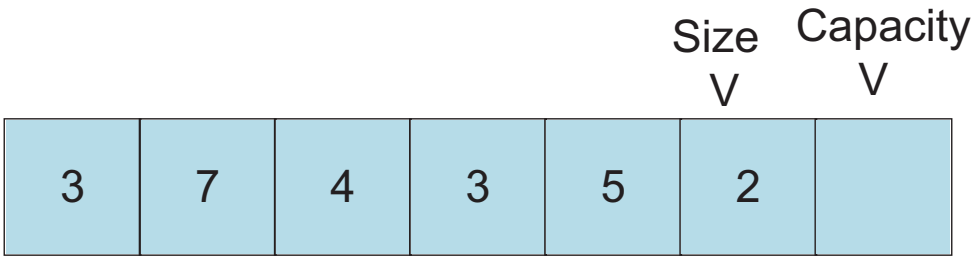


One problem with an array is that memory is allocated as a block. The size of this block is fixed when the array is created. If the size of the block corresponds directly to the number of elements in the collection, then adding a new value requires creating an entirely new block, and copying the values from the old collection into the new.

This can be avoided by purposely making the array larger than necessary. The values for the collection are stored at the bottom of the array. A counter keeps track of how many elements are currently being stored in the array. This is termed the *size* of the stack. The size must not be confused with the actual size of the block, which is termed the *capacity*.

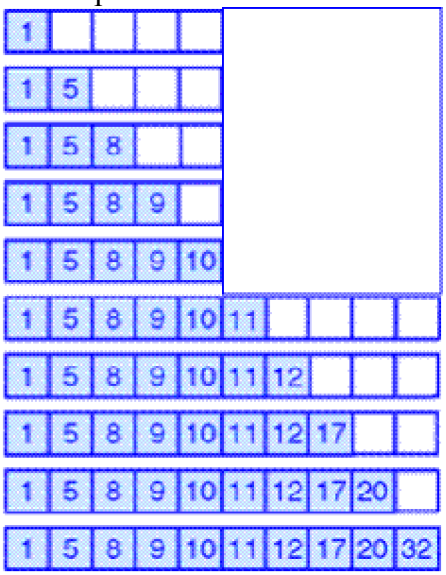


If the size is less than the capacity, then adding a new element to the stack is easy. It is simply a matter of incrementing the count on the size, and copying the new element into the correct location. Similarly, removing an element is simply a matter of setting the topmost location to null (thereby allowing the garbage collection system to recover the old value), and reducing the size.



Because the number of elements held in the collection can easily grow and shrink during run-time, this is termed a *dynamic array*. There are two exceptional conditions that must be handled. The first occurs when an attempt is made to remove a value from an empty stack. In this situation you should throw a StackUnderflow exception.

The second exceptional condition is more difficult. When a push instruction is requested but the size is equal to the capacity, there is no space for the new element. In this case a new array must be created. Typically, the size of the new array is twice the size of the current. Once the new array is created, the values are copied from existing array to the new array, and the new array replaces the current

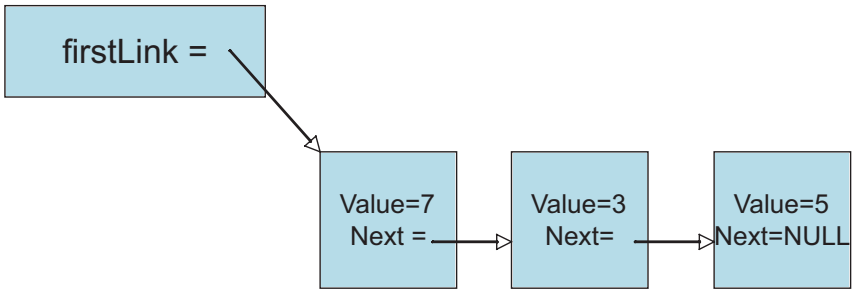


array. Since there is now enough room for the new element, it can be inserted.

Worksheet 16 explores the implementation of a dynamic array stack. In the exercises at the end of the chapter you will explore the idea that while the worst case execution time for push is relatively slow, the worst case occurs relatively infrequently. Hence, the expectation is that in the average execution of push will be quite fast. We describe this situation by saying that the method push has constant *amortized* execution time.

Linked List Implementation of Stack

An alternative implementation approach is to use a linked list. Here, the container abstraction maintains a reference to a collection of elements of type Link. Each Link maintains two data fields, a value and a reference to another link. The last link in the sequence stores a null value in its link.



The advantage of the linked list is that the collection can grow as large as necessary, and each new addition to the chain of links requires only a constant amount of work. Because there are no big blocks of memory, it is never necessary to copy an entire block from place to place.

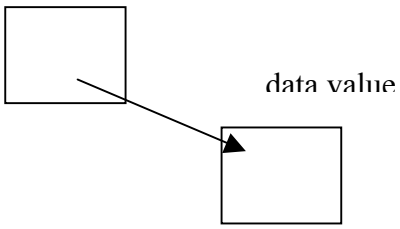
Worksheet 17 will introduce the idea of a linked list, and explore how a linked list can be used to implement a stack.

Memory Management

The linked list and the Dynamic Array data structures take different approaches to the problem of memory management. The Dynamic Array uses a large block of memory. This means that memory allocation is much less common, but when it occurs much more work must be performed. The linked list allocates a new link every time a new element is added. This makes memory allocation more frequent, but as the memory blocks are small less work is performed on each allocation.

If, as is often the case, a linked list is used to store pointers to a dynamically allocated value, then there are two dynamically allocated spaces to manage, the link and the data field itself:

link



An important principle of good memory management is "everybody must clean up their own mess". (This is sometimes termed the *kindergarten principle*). The linked list allocates space for the link, and so must ensure that the space is freed by calling the associated pop routine. The user of the list allocates space for the data value, and must therefore ensure that the field is freed when no longer needed. Whenever you create a dynamically allocated value you need to think about how and when it will be freed.

Earlier we pointed out the problem involved in placing a new element into the middle of a Dynamic Array (namely, that the following elements must then be moved). Linked lists will help solve this problem, although we have not yet demonstrated that in this chapter.

For the Dynamic Array we created a single general-purpose data structure, and then showed how to use that data structure in a variety of ways. In examining the linked list we will take a different approach. Rather than making a single data abstraction, we will examine the *idea* of the linked list in a variety of different forms. In subsequent lessons we will examine a number of variations on this idea, such as header or sentinel links, single versus double links, maintaining a pointer to the last as well as the first link, and more.

Occasionally you will find links placed directly into a data object. For example, suppose you were creating a card game, and needed a list of cards. One way to do this would be the following:

Each card can then be used as a link in a linked list.

```
struct Card {
   int suit;
   int rank;
   /* link to next card */
   struct  Card * next;
};
```

Although this approach is easy to implement, it should be avoided, for several reasons. It confuses two issues, the management of cards, and the manipulation of the list. These problems should be dealt with independently. It makes your code very rigid; for example, you cannot move the Card abstraction to another program in which cards are not on a list, or must be placed in two different lists at the same time. And finally, you end up duplicating code that you can more easily write once and reuse by using a standard container.

Self Study Questions

1. What are the defining characteristics of the stack abstraction?

2. Explain the meaning of the term LIFO. Explain why FILO might have been equally appropriate.

3. Give some examples of stacks found in real life.

4. Explain how the use of an activation record stack simplifies the allocation of memory for program variables. Explain how an activation record stack make it possible to perform memory allocation for recursive procedures.

5. Evaluate the following postfix polish expressions:
    a. 2 3 + 5 9 - *
    b. 2 3 5 9 + - *

6. How is the memory representation of a linked list different from that of a Dynamic Array?

7. What information is stored in a link?

8. How do you access the first element in a linked list? How would you get to the second element?

9. In the last link, what value is stored in the next field?

10. What is the big-Oh complexity of pushing a value on the front of a linked list stack? Of popping?

11. What would eventually happen if the pop function did not free the first link in the list?

12. Suppose you wanted to test your linked list class. What would be some boundary value test cases? Develop a test harness program and execute your code with these test cases.

Short Exercises

1. [Java] Describe the state of an initially empty stack after each of the following sequence of operations. Indicate the values held by any variables that are declared, and also indicate any errors that may occur:

a. que.addLast(new Integer(3));
Object a = que.getLast();
que.addLast(new Integer(5));
que.removeLast();

b. que.addLast(new Integer(3));
que.addLast(new Integer(4));
que.removeLast();
que.removeLast();
Object a = que.getLast();

2. What is the Polish notation representation of the following expression?

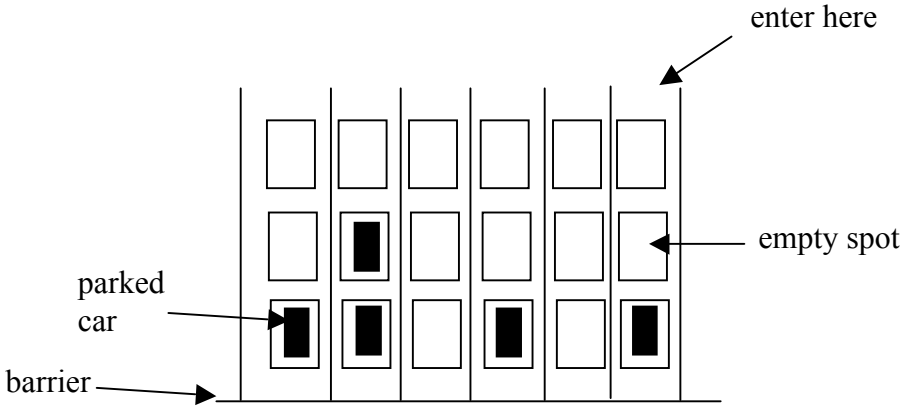
(a * (b + c)) + (b / d) * a

3. One problem with polish notation is that you cannot use the same symbol for both unary and binary operators. Illustrate this by assuming that the minus sign is used for both unary and binary negation, and explain the two alternative meanings for the following postfix polish expression:

7 5 - -

4. Write an algorithm to translate a postfix polish expression into an infix expression that uses parenthesis only where necessary.

5. Phil Parker runs a parking lot where cars are stored in six stacks holding at most three cars each. Patrons leave the keys in their cars so that they can be moved if necessary.



Assuming that no other parking space is available, should Phil allow his parking space to become entirely full? Assuming that Phil cannot predict the time at which patrons will

return for their cars, how many spaces must he leave empty to ensure that he can reach any possible car?

6. [Java] Some people prefer to define the Stack data type by means of a series of *axioms*. An example axiom might be that if a push operation is followed by a top, the value returned will be the same as the value inserted. We can write this in a code like fashion as follows:

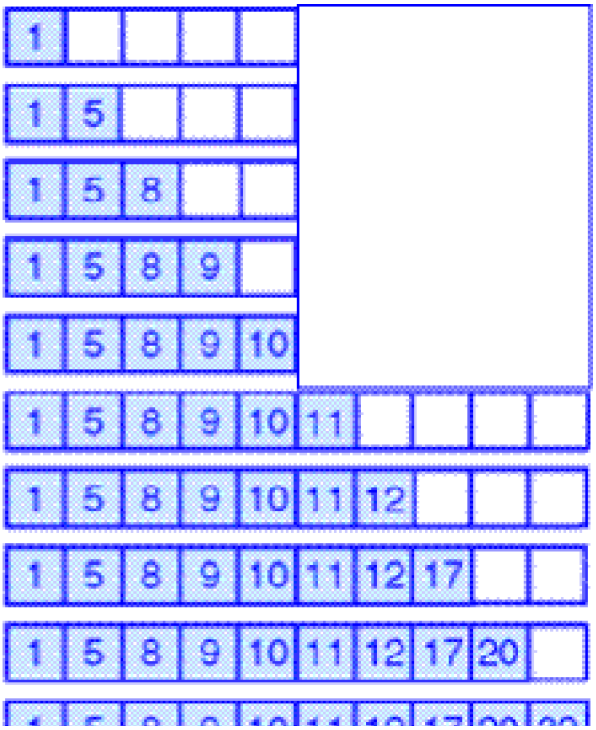
```
Object val = new Integer(7);
stk.push(val);
boolean test = (val == stk.top()); // test must always be true
```

Trace the ArrayStack implementation of the stack and verify the following axioms:

```
Stack stk = new Stack();
boolean test = stk.isEmpty(); // should always be true

stk.push(new Integer(2));
boolean test = stk.isEmpty(); // should always be false

Stack stk = new Stack();
boolean test = stk.pop(); // should always raise error
```

7. Using a ListStack as the implementation structure do the same analysis as in previous question.

8. [Java] Does an ArrayStack or a ListStack use less memory? Assume for this question that a data value requires 1 unit of memory, and each memory reference (such as the next field in a Link, or the firstLink field in the ListStack, or the data field in the ArrayStack) also requires 1 unit of memory. How much memory is required to store a stack of 100 values in an ArrayStack? How much memory in a ListStack?



Analysis Exercises

1. When you developed the ArrayStack you were asked to determine the algorithmic execution time for the push operation. When the capacity was less than the size, the execution time was constant. But when a reallocation became necessary, execution time slowed to O(n).

This might at first seem like a very negative result, since it means that the

worst case execution time for pushing an item on to the stack is O(n). But the reality is not nearly so bleak. Look again at the picture that described the internal array as new elements were added to the collection.

Notice that the costly reallocation of a new array occurred only once during the time that ten elements were added to the collection. If we compute the *average* cost, rather than the *worst case* cost, we will see that the ArrayStack is still a relatively efficient container.

To compute the average, count 1 "unit" of cost each time a value is added to the stack without requiring a reallocation. When the reallocation occurs, count ten "units" of cost for the assignments performed as part of the reallocation process, plus one more for placing the new element into the newly enlarged array. How many "units" are spent in the entire process of inserting these ten elements? What is the average "unit" cost for an insertion?

When we can bound an "average" cost of an operation in this fashion, but not bound the worst case execution time, we call it *amortized constant* execution time, or *average* execution time. Amortized constant execution time is often written as O(1)+, the plus sign indicating it is not a guaranteed execution time bound.

Do a similar analysis for 25 consecutive add operations, assuming that the internal array begins with 5 elements (as shown). What is the cost when averaged over this range?

This analysis can be made into a programming assignment. Rewrite the ArrayStack class to keep track of the "unit cost" associated with each instruction, adding 1 to the cost for each simple insertion, and n for each time an array of n elements is copied. Then print out a table showing 200 consequitive insertions into a stack, and the value of the unit cost at each step.

2. [Java] The Java standard library contains a number of classes that are implemented using techniques similar to those you developed in the programming lessons described earlier. The classes Vector and ArrayList use the dynamic array approach, while the class LinkedList uses the idea of a linked list. One difference is that the names for stack operations are different from the names we have used here:

Stack	Vector	ArrayList	LinkedList
Push(newValue)	Add(newValue)	Add(newValue)	addFirst(newObject)
Pop()	Remove(size()-1)	Remove(size()-1)	removeFirst(ewObject)
Top()	lastElement()	Get(size()-1)	getFirst()
IsEmpty()	Size() == 0	isEmpty()	isEmpty()

Another difference is that the standard library classes are designed for many more tasks than simply representing stacks, and hence have a much larger interface.

An important principle of modern software development is an emphasis on software reuse. Whenever possible you should leverage existing software, rather than rewriting

new code that matches existing components. But there are various different techniques that can be used to achieve software reuse. In this exercise you will investigate some of these, and explore the advantages and disadvantages of each. All of these techniques leverage an existing software component in order to simplify the creation of something new.

Imagine that you are a developer and are given the task of implementing a stack in Java. Part of the specifications insist that stack operations must use the push/pop/top convention. There are at least three different approaches you could use that

1. If you had access to the source code for the classes in the standard library, you could simply add new methods for these operations. The implementation of these methods can be pretty trivial, since they need do nothing more than invoke existing functions using different names.

2. You could create a new class using inheritance, and subclass from the existing class.

```
class Stack extends Vector {
   …
}
```

   Inheritance implies that all the functionality of the parent class is available automatically for the child class. Once more, the implementation of the methods for your stack can be very simple, since you can simply invoke the functions in the parent class.

3. The third alternative is to use composition rather than inheritance. You can create a class that maintains an internal data field of type Vector (alternatively, ArrayList or LinkedList). Again, the implementation of the methods for stack operations is very simple, since you can use methods for the vector to do most of the work.

```
class Stack<T> {
   private Vector<T> data;
   …
}
```

Write the implementation of each of these. (For the first, just write the methods for the stack operations, not the other vector code). Then compare and contrast the three designs. Issues to consider in your analysis include readability/usability and encapsulation. By readability or usability we mean the following: how much information must be conveyed to a user of your new class before they can do their job. By encapsulation we mean: How good a job does your design do in guaranteeing the safety of the data? That is, making sure that the stack is accessed using only valid stack instructions. On the other hand, there may be reasons why you might want to allow the stack to be accessed using non-stack instructions. A common example is

allowing access to all elements of the stack, not just the first. Which design makes this easier?

If you are the developer for a collection class library (such as the developer for the Java collection library), do you think it is a better design choice to have a large number of classes with very small interfaces, or a very small number of classes that can each be used in a number of ways, and hence have very large interfaces? Describe the advantages and disadvantages of both approaches.

The bottom line is that all three design choices have their uses. You, as a programmer, need to be aware of the design choices you make, and the reasons for selecting one alternative over another.

3. In the previous question you explored various alternative designs for the Stack abstraction when built on top of an existing class. In some of those there was the potential that the end user could manipulate the stack using commands that were not part of the stack abstraction. One way to avoid this problem is to define a stack interface. An *interface* is similar to a class, but only describes the signatures for operations, not their implementations. An interface for the Stack abstraction can be given as follows:

```
interface Stack<T> {
   public void push (T newValue);
   public void pop ();
   public T top ();
   public Boolean isEmpty();
}
```

Rewrite your two stack implementations (ArrayStack and ListStack) using the stack interface. Verify that you can now declare a variable of type Stack, and assign it a value of either ArrayStack or ListStack. Rewrite the implementations from the previous analysis lesson so that they use the stack interface. Verify that even when the stack is formed using inheritance from ArrayList or Vector, that a variable declared as type Stack cannot use any operations except those specified by the interface.

Can you think of a reason why the designer of the Java collection classes did not elect to define interfaces for the common container types?

4. Whenever you have two different implementations with the same interface, the first question you should ask is whether the algorithmic execution times are the same. If not, then select the implementation with the better algorithmic time. If, on the other hand, you have similar algorithmic times, then a valid comparison is to examine actual clock times (termed a benchmark). In C, you can determine the current time using the function named clock that is defined in the <time.h> interface file:

```
# include <time.h>
```

```
double getMilliseconds() {
      return 1000.0 * clock() / CLOCKS_PER_SEC;
}
```

As the name suggests, this function returns the current time in milliseconds. If you subtract an earlier time from a later time you can determine the amount of time spent in performing an action.

In the first experiment, try inserting and removing ten values from a stack, doing these operations *n* times, for various values of *n*. Plot your results in a graph that compares the execution time to the value of *n* (where n ranges, say, from 1000 to 10,000 in increments of 1000).

This first experiment can be criticized because once the vector has reached its maximum size it is never enlarged. This might tend to favor the vector over the linked list. An alternative exercise would be to insert and remove *n* values. This would force the vector to continually increase in size. Perform this experiment and compare the resulting execution times.

Programming Assignments

1. Complete the implementation of a program that will read an infix expression from the user, and print out the corresponding postfix expression.

2. Complete the implementation of a program that will read a postfix expression as a string, break the expression into parts, evaluate the expression and print the result.

3. Combine parts 1 and 2 to create a program that will read an infix expression from the user, convert the infix expression into an equivalent postfix expression, then evaluate the postfix expression.

4. Add a graphical user interface (GUI) to the calculator program. The GUI will consist of a table of buttons for numbers and operators. By means of these, the user can enter an expression in infix format. When the calculate button is pushed, the infix expression is converted to postfix and evaluated. Once evaluated, the result is displayed.

5. Here is a technique that employs two stacks in order to determine if a phrase is a palindrome, that is, reads the same forward and backward (for example, the word “rotator"” is a palindrome, as is the string “rats live on no evil star”).

    - Read the characters one by one and transfer them into a stack. The characters in the stack will then represent the reversed word.

- Once all characters have been read, transfer half the characters from the first stack into a second stack. Thus, the order of the words will have been restored.

- If there were an odd number of characters, remove one further character from the original stack.

- Finally, test the two stacks for equality, element by element. If they are the same, then the word is a palindrome.

Write a procedure that takes a String argument and tests to see if it is a palindrome using this algorithm.

6. In Chapter 5 you learned that a Bag was a data structure characterized by the following operations: add an element to the collection, test to see if an element is in the collection, and remove an element from the collection. We can build a bag using the same linked list techniques you used for the linked list stack. The add operation is the same as the stack. To test an element, simply loop over the links, examining each in turn. The only difficult operation is remove, since to remove a link you need access to the immediately preceding link. To implement this, one approach is to loop over the links using a pair of pointers. One pointer will reference the current link, while the second will always reference the immediate predecessor (or null, if the current link is the first element). That way, when you find the link to remove, you simply update the predecessor link. Implement the three bag operations using this approach.

On the Web

The wikipedia entry for "Stack (data structure)" provides another good introduction to the concept. Other informative wikipedia entries include LIFO, call stack, and stack-based memory allocation. Wikipedia also provides a bibliographical sketch of Friedrich L. Bauer, the computer scientist who first proposed the use of stack for evaluating arithmetic expressions.

The NIST *Dictionary of Algorithms and Data Structures* also has a simple description of the stack data type. (http://www.nist.gov/dads/HTML/stack.html)

If you google "Stack in Java" (or C++, or C, or any other language) you will find many good examples.

Chapter 7: Queues and Deques

After the stack, the next simplest data abstraction is the *queue*. As with the stack, the queue can be visualized with many examples you are already familiar with from everyday life. A simple illustration is a line of people waiting to enter a theater. The fundamental property of the queue is that items are inserted at one end (the rear of the line) and removed from the other (the door to the theater). This means that the order that items are removed matches the order that they are inserted. Just as a stack was described as a LIFO (last-in, first-out) container, this means a queue can be described as FIFO (first in, first out).

A variation is termed the *deque*, pronounced "deck", which stands for *d*ouble-*e*nded *que*ue. In a deque values can be inserted at *either* the front or the back, and similarly the deck allows values to be removed from either the front or the back. A collection of peas in a straw is a useful mental image. (Or, to update the visual image, a series of tapioca buds in a bubble-tea straw).

Queues and deques are used in a number of ways in computer applications. A printer, for example, can only print one job at a time. During the time it is printing there may be many different requests for other output to be printed. To handle these the printer will maintain a queue of pending print tasks. Since you want the results to be produced in the order that they are received, a queue is the appropriate data structure.

The Queue ADT specification

The classic definition of the queue abstraction as an ADT includes the following operations:

addBack (newElement)	Insert a value into the queue
front()	Return the front (first) element in queue
removeFront()	Remove the front (first) element in queue
isEmpty()	Determine whether the queue has any elements

The deque will add the following:

addFront(newElement)	Insert a value at front of deque
back()	Return the last element in the queue
removeBack()	Return the last element in the queue

Notice that the queue is really just a special case of the deque. Any implementation of the deque will also work as an implementation of the queue. (A deque can also be used to implement a stack, a topic we will explore in the exercises).

As with the stack, it is the FIFO (first in, first out) property that is, in the end, the fundamental defining characteristic of the queue, and not the names of the operations. Some designers choose to use names such as “add”, “push” or “insert”, leaving the location unspecified. Similarly some implementations elect to make a single operation that will both return and remove an element, while other implementations separate these two tasks, as well do here. And finally, as in the stack, there is the question of what the effect should be if an attempt is made to access or remove an element from an empty collection. The most common solutions are to throw an exception or an assertion error (which is what we will do), or to return a special value, such as Null.

For a deque the defining property is that elements can only be added or removed from the end points. It is not possible to add or remove values from the middle of the collection.

The following table shows the names of deque and queue operations in various programming languages:

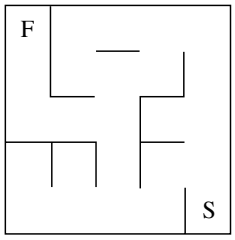
operation	C++	Java	Perl	Python
insert at front	push_front	addFirst	unshift	appendleft
Insert at back	Push_back	addLast	Push	append
remove last	pop_back	removeLast	pop	pop
remove first	pop_front	removeFirst	shift	popleft
examine last	back	getLast	$_[-1]	deque[-1]
examine first	front	getFirst	$_[0]	deque[0]

Applications of Queues

Given that queues occur frequently in real life, it is not surprising that they are also frequently used in simulations. To model customers using a bank, for example, you could use a queue of waiting patrons. A simulation might want to ask questions such as the how the average waiting time would change if you added or removed a teller position.

Queues are also used in any time collection where time of insertion is important. We have noted, for example, that a printer might want to keep the collection of pending jobs in a queue, so that they will be printed in the same order that they were submitted.

Depth-first and Breadth-first search

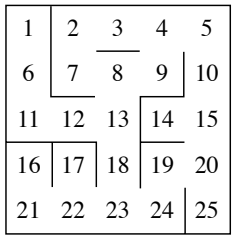


Imagine you are searching a maze, such as the one shown at right. The goal of the search is to move from the square marked S, to the square marked F.

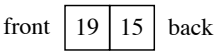
A simple algorithm would be the following:

How to search a maze:
Keep a list of squares you have visited, initially empty.
Keep a stack of squares you have yet to visit. Put the starting square in this stack
While the stack is not empty:
    Remove an element from the stack
    If it is the finish, then we are done
    Otherwise if you have already visited this square, ignore it
    Otherwise, mark the square on your list of visited positions, and
    add all the neighbors of this square to your stack.
If you eventually reach an empty stack and have not found the start, there is no solution

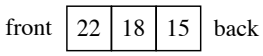
To see the working of this algorithm in action, let us number of states of our maze from 1 to 25, as shown. There is only one cell reachable from the starting position, and thus after the first step the queue contains only one element:



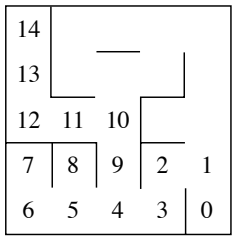
This value is pulled from the stack, and the neighbors of the cell are inserted back. This time there are two neighbors, and so the stack will have two entries.



Only one position can be explored at any time, and so the first element is removed from the stack, leaving the second waiting for later exploration. Two steps later we again have a choice, and both neighbors are inserted into the stack. At this point, the stack has the following contents:



The solution is ultimately found in fifteen steps. The following shows the path to the solution, with the cells numbered in the order in which they were considered.



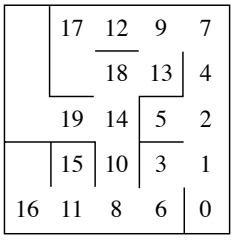
The strategy embodied in this code doggedly pursues a single path until it either reaches a dead end or until the solution is found. When a dead end is encountered, the most recent alternative path is reactivated, and the search continues. This approach is called a *depth-first search*, because it moves deeply into the structure before examining alternatives. A depth-first search is the type of search a single individual might perform in walking through a maze.

Suppose, on the other hand, that there were a group of people walking together. When a choice of alternatives was encountered, the group might decide to split itself into smaller groups, and explore each alternative simultaneously. In this fashion all potential paths are investigated at the same time. Such a strategy is known as a *breadth-first* search.

What is intriguing about the maze-searching algorithm is that the exact same algorithm can be used for both, changing only the underlying data structure. Imagine that we change the stack in the algorithm to a queue:

> **How to search a maze using breadth-first search**:
> Keep a list of squares you have visited, initially empty.
> Keep a queue of squares you have yet to visit. Put the starting square in this queue
> While the queue is not empty:
> Remove an element from the queue
> If it is the finish, then we are done
> Otherwise if you have already visited this square, ignore it
> Otherwise, mark the square on your list of visited positions, and
> add all the neighbors of this queue to your stack.
> If you eventually reach an empty queue and have not found the start, there is no solution

As you might expect, a breadth-first search is more thorough, but may require more time than a depth-first search. While the depth-first search was able to find the solution in 15 steps, the breadth-first search is still looking after 20. The following shows the search at this point. Trace with your finger the sequence of steps as they are visited. Notice how the search jumps all over the maze, exploring a number of different alternatives at the same time. Another way to imagine a breadth-first first is as what would happen if ink were poured into the maze at the starting location, and slowly permeates every path until the solution is reached.



We will return to a discussion of depth-first and breadth-first search in a later chapter, after we have developed a number of other data structures, such as graphs, that are useful in the representation of this problem.

Queue Implementation Techniques

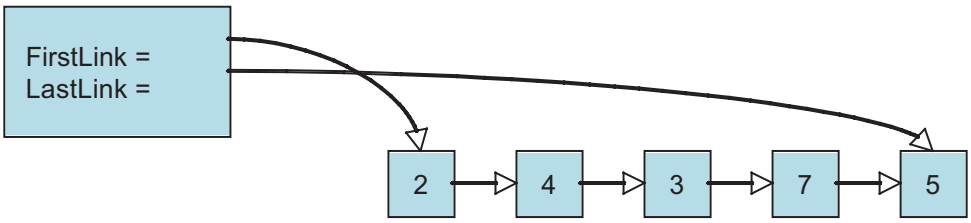
As with the stack, the two most common implementation techniques for a queue are to use a linked list or to use an array. In the worksheets you will explore both of these alternatives.

Worksheet 18	Linked List Queue, Introduction to Sentinels
Worksheet 19	Linked List Deque
Worksheet 20	Dynamic Array Queue

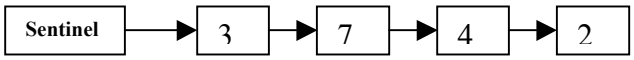
Building a Linked List Queue

A stack only needs to maintain a link to one end of the chain of values, since both insertions and removals occur on the same side. A queue, on the other hand, performs

insertions on one side and removals from the other. Therefore it is necessary to maintain a links to both the front and the back of the collection.



We will add another variation to our container. A *sentinel* is a special link, one that does not contain a value. The sentinel is used to mark either the beginning or end of a chain of links. In our case we will use a sentinel at the front. This is sometimes termed a *list header*. The presence of the sentinel makes it easier to handle special cases. For example, the list of links is never actually empty, even when it is logically empty, since there is always at least one link (namely, the sentinel). A new value is inserted after the end, after the element pointed to by the last link field. Afterwards, the last link is updated to refer to the newly inserted value.

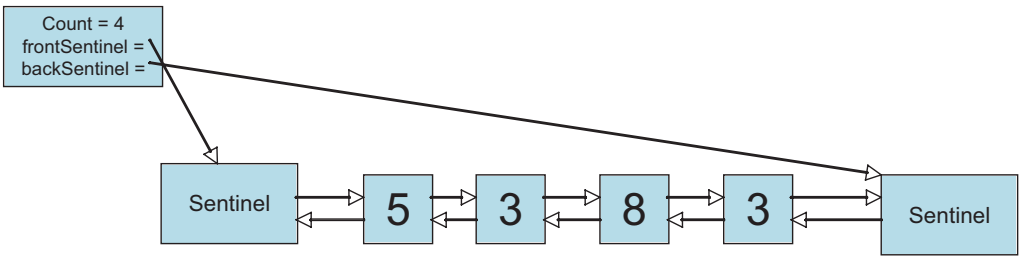


Values are removed from the front, as with the stack. But because of the sentinel, these will be the element right after the sentinel.

You will explore the implementation of the queue using linked list techniques in Worksheet 18.

A Linked List Deque – using Double Links

You may have noticed that removal from the end of a ListQueue is difficult because with only a single link it is difficult to “back up”. That is, while you have a pointer to the end sentinel, you do not have an easy way to back up and find the link immediately preceding the sentinel.



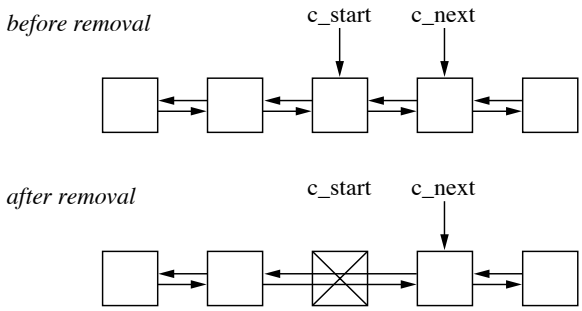
One solution to this problem is to use a *doubly-linked* list. In a doubly-linked list, each link maintains two pointers. One link, the forward link, points forward toward the next link in the chain. The other link, the prev link, points backwards towards the previous

element. Anticipating future applications, we now also keep a count of the number of elements in the list.

With this picture, it is now easy to move either forward or backwards from any link. We will use this new ability to create a linked list deque. In order to simplify the implementation, we will this time include sentinels at both the beginning and the end of the chain of links. Because of the sentinels, both adding to the front and adding to the end of a collection can be viewed as special cases of a more general "add to the middle of a list" operation. That is, perform an insertion such as the following:

picture

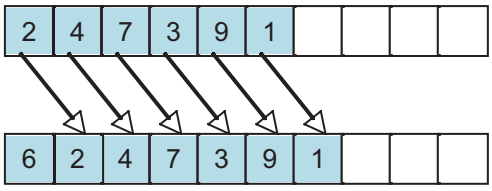
Similarly, removing a value (from other the front or the back) is a special case of a more general remove operation:



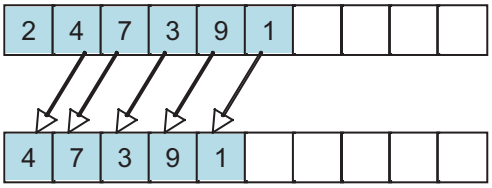
In worksheet 19 you will complete the implementation of the list deque based on these ideas.

A Dynamic Array Deque

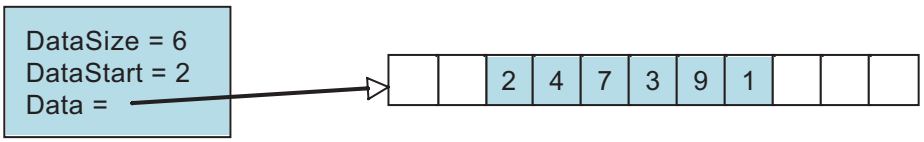
The dynamic array that we used in implementing the ArrayStack will not work for either a queue or a deque. The reason is that inserting an element at the front of the array requires moving every element over by one position. A loop to perform this movement would require O(n) steps, far too slow for our purposes.



If we tried adding elements to the end (as we did with the stack) then the problem is reversed. Now it is the remove operation that is slow, since it requires moving all values down by one position



The root of the problem is that we have fixed the start of the collection at array index position zero. If we simply loosen that requirement, then things become much easier. Now in addition to the data array our collection will maintain two integer data fields; a size (as before) and a starting location.



Adding a value to the end is simlar to the ArrayStack. Simply increase the size, and place the new element at the end.

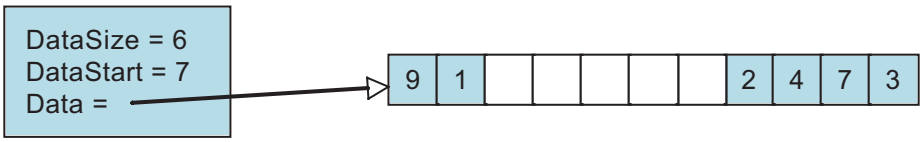
picture

But adding a value to the front is also simple. Simply decrement the starting point by one, then add the new element to the front:

picture

Removing elements undo these operations. As before, if an insertion is performed when the size is equal to the capacity, the array must be doubled in capacity, and the values copied into the new array.

There is just one problem. Nothing prevents the data values from wrapping around from the upper part of the array to the lower:



To accommodate index values must be computed carefully. When an index wraps around the end of the array it must be altered to point to the start of the array. This is easily done by subtracting the capacity of the array. That is, suppose we try to index the fifth element in the picture above. We start by adding the index, 5, to the starting location, 7. The resulting sum is 12. But there are only eleven values in the collection. Subtracting 11 from 12 yields 1. This is the index for the value we seek.

In worksheet 20 you will explore the implementation of the dynamic array deque constructed using these ideas. A deque implemented in this fashion is sometimes termed a *circular buffer*, since the right hand side circles around to begin again on the left.

Self Study Questions

1. What are the defining characteristics of the queue ADT?

2. What do the letters in FIFO represent? How does this describe the queue?

3. What does the term deque stand for?

4. How is the deque ADT different from the queue abstraction?

5. What will happen if an attempt is made to remove a value from an empty queue?

6. What does it mean to perform a depth-first search? What data structure is used in performing a depth-first search?

7. How is a breadth-first search different from a depth-first search? What data structure is used in performing a breadth-first search?

8. What is a sentinel in a linked list?

9. What does it mean to say that a list is singly-linked?

10. How is a doubly-linked list different from a singly-linked list? What new ability does the doubly-linked feature allow?

11. Why is it difficult to implement a deque using the same dynamic array implementation that was used in the dynamic array stack?

Analysis Exercises

1. A deque can be used as either a stack or a queue. Do you think that it is faster or slower in execution time than either a dynamic array stack or a linked list stack? Can you design an exercise to test your hypothesis? In using a Deque as a stack there are two choices; you can either add and remove from the front, or add and remove from the back. Is there a measurable difference in execution time between these two alternatives?

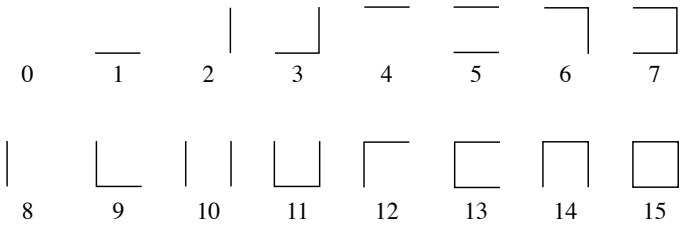
Programming Assignments

1. Many computer applications present practice drills on a subject, such as arithmetic addition. Often such systems will present the user with a series of problems and keep track of those the user answered incorrectly. At the end of the session, the incorrect problems will then be presented to the user a second time. Implement a practice drill system for addition problems having this behavior.

2. FollowMe is a popular video game. The computer displays a sequence of values, and then asks the player to reproduce the same sequence. Points are scores if the entire

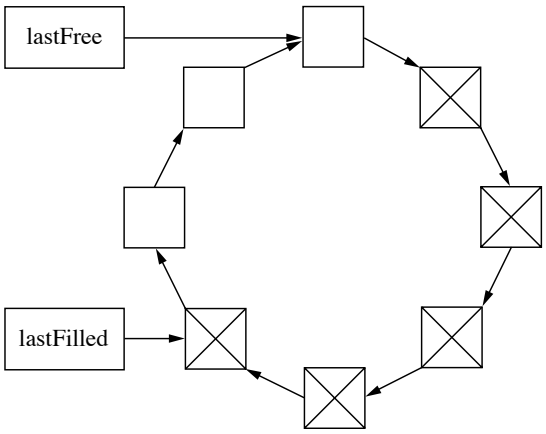
sequence was produced in order. In implementing this game, we can use a queue to store the sequence for the period of time between generation by the computer and response by the player.

3. To create a maze-searching program you first need some way to represent the maze. A simple approach is to use a two-dimensional integer array. The values in this array can represent the type of room, as shown below. The values in this array can then tell you the way in which it is legal to move. Positions in the maze can be represented by (I,j) pairs. Use this technique to write a program that reads a maze description, and prints a sequence of moves that are explored in either a depth-first or breadth-first search of the maze.



4. In Chapter 5 you learned that a *Bag* was a data structure characterized by the following operations: add an element to the collection, test to see if an element is in the collection, and remove an element from the collection. We can build a bag using the same linked list techniques you used for the linked list queue. The add operation is the same as the queue. To test an element, simply loop over the links, examining each in turn. The only difficult operation is remove, since to remove a link you need access to the immediately preceding link. To implement this, one approach is to loop over the links using a pair of pointers. One pointer will reference the current link, while the second will always reference the immediate predecessor (or the sentinel, if the current link is the first element). That way, when you find the link to remove, you simply update the predecessor link. Implement the three bag operations using this approach. Does the use of the sentinel make this code easier to understand than the equivalent code was for the stack version?

5. Another implementation technique for a linked list queue is to link the end of the queue to the front. This is termed a circular buffer. Two pointers then provide access to the last filled position, and the last free position. To place an element into the queue, the last filled pointer is advanced, and the value stored. To remove a value the last free pointer is advanced, and the value returned. The following picture shows this structure. How do you know when the queue is completely filled? What action should you take to increase the size of the structure in this situation? Implement a circular buffer queue based on these ideas.

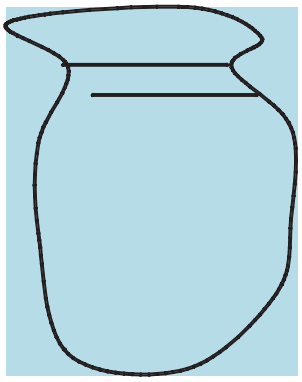


On The Web

Wikipedia has well written entries for queue, deque, FIFO, as well as on the implementation structures dynamic array, linked list and sentinel node. The NIST *Dictionary of Algorithms and Data Structures* also has a good explanation.

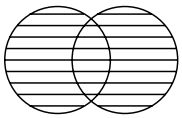
Chapter 8: Bags and Sets

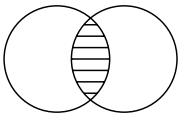
In the stack and the queue abstractions, the order that elements are placed into the container is important, because the order elements are removed is related to the order in which they are inserted. For the *Bag*, the order of insertion is completely irrelevant. Elements can be inserted and removed entirely at random.



By using the name *Bag* to describe this abstract data type, the intent is to once again to suggest examples of collection that will be familiar to the user from their everyday experience. A bag of marbles is a good mental image. Operations you can do with a bag include inserting a new value, removing a value, testing to see if a value is held in the collection, and determining the number of elements in the collection. In addition, many problems require the ability to loop over the elements in the container. However, we want to be able to do this without exposing details about how the collection is organized (for example, whether it uses an array or a linked list). Later in this chapter we will see how to do this using a concept termed an *iterator*.

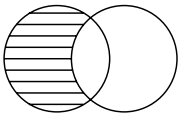
A *Set* extends the bag in two important ways. First, the elements in a set must be unique; adding an element to a set when it is already contained in the collection will have no effect. Second, the set adds a number of operations that combine two sets to produce a new set. For example, the set *union* is the set of values that are present in either collection.





The *intersection* is the set of values that appear in both collections.

A set *difference* includes values found in one set but not the other.



Finally, the subset test is used to determine if all the values found in one collection are also found in the second. Some implementations of a set allow elements to be repeated more than once. This is usually termed a *multiset*.

The Bag and Set ADT specifications

The traditional definition of the Bag abstraction includes the following operations:

Add (newElement)	Place a value into the bag
Remove (element)	Remove the value
Contains (element)	Return true if element is in collection
Size ()	Return number of values in collection
Iterator ()	Return an iterator used to loop over collection

As with the earlier containers, the names attached to these operations in other implementations of the ADT need not exactly match those shown here. Some authors

prefer “insert” to “add”, or “test” to “contains”. Similarly, there are differences in the exact meaning of the operation “remove”. What should be the effect if the element is not found in the collection? Our implementation will silently do nothing. Other authors prefer that the collection throw an exception in this situation. Either decision can still legitimately be termed a bag type of collection.

The following table gives the names for bag-like containers in several programming languages.

operation	Java Collection	C++ vector	Python
Add	Add(element)	Push_back(element)	Lst.append(element)
remove	Remove(element)	Erase(iterator)	Lst.remove(element)
contains	Contains(element)	Count(iterator)	Lst.count(element)

The set abstraction includes, in addition to all the bag operations, several functions that work on two sets. These include forming the intersection, union or difference of two sets, or testing whether one set is a subset of another. Not all programming languages include set abstractions. The following table shows a few that do:

operation	Java Set	C++ set	Python list comprehensions
intersection	retainAll	Set_intersection	[x for x in a if x in b]
union	addAll	Set_union	[x if (x in b) or (x in a)]
difference	removeAll	Set_difference	[x for x in a if x not in b]
subset	containsAll	includes	Len([x for x in a if x not in b]) != 0

Python list comprehensions (modeled after similar facilities in the programming languages ML and SETL) are a particularly elegant way of manipulating set abstractions.

Applications of Bags and Sets

The bag is the most basic of collection data structures, and hence almost any application that does not require remembering the order that elements are inserted will use a variation on a bag. Take, for example, a spelling checker. An on-line checker would place a dictionary of correctly spelled words into a bag. Each word in the file is then tested against the words in the bag, and if not found it is flagged. An off-line checker could use set operations. The correctly spelled words could be placed into one bag, the words in the document placed into a second, and the difference between the two computed. Words found in the document but not the dictionary could then be printed.

Bag and Set Implementation Techniques

For a Bag we have a much wider range of possible implementation techniques than we had for stacks and queues. So many possibilities, in fact, that we cannot easily cover them in contiguous worksheets. The early worksheets describe how to construct a bag using the techniques you have seen, the dynamic array and the linked list. Both of these require the

use of an additional data abstraction, the iterator. Later, more complex data structures, such as the skip list, avl tree, or hash table, can also be used to implement bag-like containers.

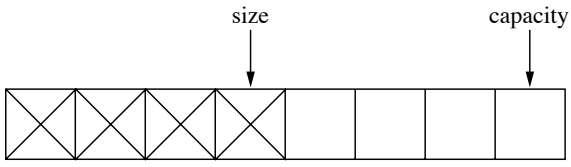
Another thread that weaves through the discussion of implementation techniques for the bag is the advantages that can be found by maintaining elements in order. In the simplest there is the sorted dynamic array, which allows the use of binary search to locate elements quickly. A skip list uses an ordered linked list in a more subtle and complex fashion. AVL trees and similarly balanced binary trees use ordering in an entirely different way to achieve fast performance.

The following worksheets describe containers that implement the bag interface. Those involving trees should be delayed until you have read the chapter on trees.

Worksheet 21	Dynamic Array Bag
Worksheet 22	Linked List Bag
Worksheet 23	Introduction to the Iterator
Worksheet 24	Linked List Iterator
Worksheet 26	Sorted Array Bag
Worksheet 28	Skip list bag
Worksheet 29	Binary Search Trees
Worksheet 31	AVL trees
Worksheet 37	Hash tables

Building a Bag using a Dynamic Array

For the Bag abstraction we will start from the simpler dynamic array stack described in Chapter 6, and not the more complicated deque variation you implemented in Chapter 7. Recall that the Container maintained two data fields. The first was a reference to an array of objects. The number of positions in this array was termed the *capacity* of the container. The second value was an integer that represented the number of elements held in the container. This was termed the *size* of the collection. The size must always be smaller than or equal to the capacity.



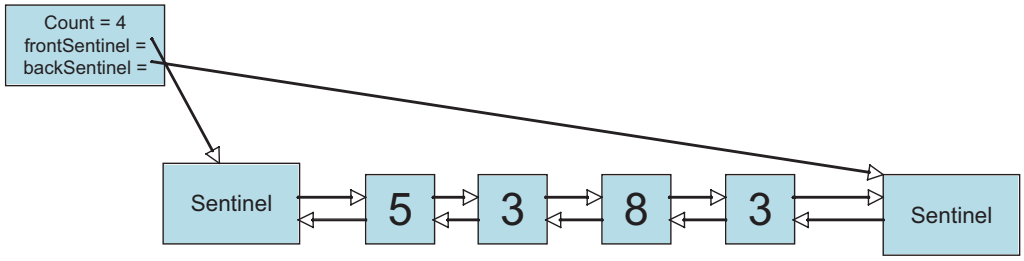
As new elements are inserted, the size is increased. If the size reaches the capacity, then a new array is created with twice the capacity, and the values are copied from the old array into the new. This process of reallocating the new array is an issue you have already solved back in Chapter 6. In fact, the function *add* can have exactly the same behavior as the function *push* you wrote for the dynamic array stack. That is, add simply inserts the new element at the end of the array.

The contains function is also relatively simple. It simply uses a loop to cycle over the index values, examining each element in turn. If it finds a value that matches the argument, it returns true. If it reaches the end of the collection without finding any value, it returns false.

The remove function is the most complicated of the Bag abstraction. To simplify this task we will divide it into two distinct steps. The remove function, like the contains function, will loop over each position, examining the elements in the collection. If it finds one that matches the desired value, it will invoke a separate function, removeAt, that removes the value held at a specific location. You will complete this implementation in Worksheet 21.

Constructing a Bag using a Linked List

To construct a Bag using the idea of a Linked List we will begin with the list deque abstraction you developed in Chapter 7. Recall that this implementation used a sentinel at both ends and double links.



The *contains* function must use a loop to cycle over the chain of links. Each element is tested against the argument. If any are equal, then the Boolean value true is returned. Otherwise, if the loop terminates without finding any matching element, the value False is returned.

The *remove* function uses a similar loop. However, this time, if a matching value is found, then the function *removeLink* is invoked. The remove function then terminates, without examining the rest of the collection. (As a consequence, only the first occurrence of a value is removed. Repeated values may still be in the collection. A question at the end of this chapter asks you to consider different implementation techniques for the *removeAll* function.)

Introduction to the Iterator

As we noted in Chapter 5, one of the primary design principles for collection classes is *encapsulation*. The internal details concerning how an implementation works are hidden behind a simple and easy to remember interface. To use a Bag, for example, all you need know is the basic operations are add, collect and remove. The inner workings of the implementation for the bag are effectively hidden.

When using collections a common requirement is the need to loop over all the elements in the collection, for example to print them to a window. Once again it is important that this process be performed without any knowledge of how the collection is represented in memory. For this reason the conventional solution is to use a mechanism termed an *Iterator*.

```
/* conceptual interface */
 Boolean (or int) hasNext ( );
 TYPE next ( );
 void remove ( );
```

```
LinkedListIterator itr;
TYPE current;
…
initListIterator (aList, itr);
while (hasNextListIterator (itr)) {
   current = nextListIterator (itr);
   … /* do something with current
*/
}
```

Each collection will be matched with a set of functions that implement this interface. The functions next and hasNext are used in combination to write a simple loop that will cycle over the values in the collection.

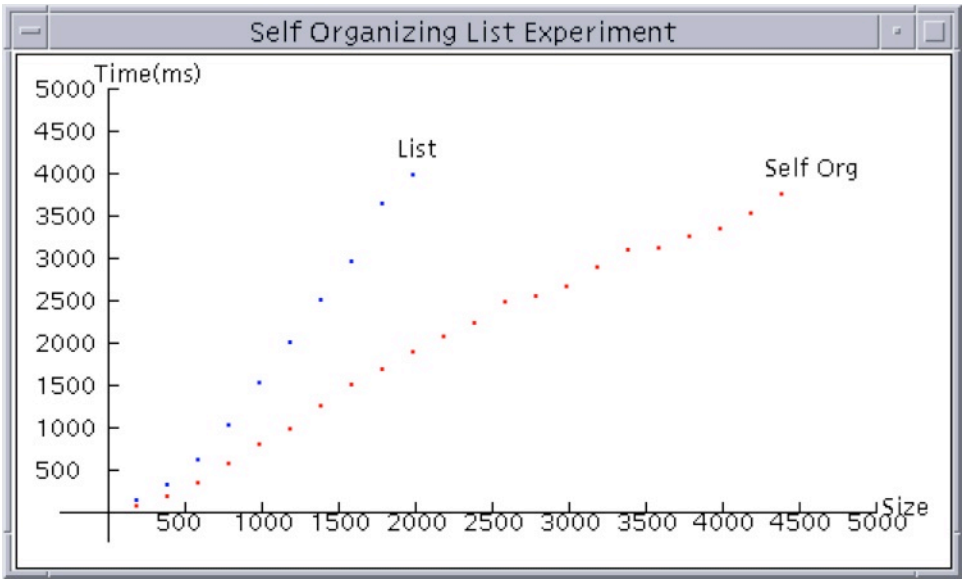
The iterator loop exposes nothing regarding the structure of the container class. The function *remove* can be used to delete from the collection the value most recently returned by next.

Notice that an iterator is an object that is separate from the collection itself. The iterator is a *facilitator object*, that provides access to the container values. In worksheets 23 and 24 you complete the implementation of iterators for the dynamic array and for the linked list.

Self Organizing Lists

We have treated all list operations as if they were equally likely, but this is not always true in practice. Often an analysis of the frequency of operations will suggest ways that a data structure can be modified in order to improve performance. For example, one common situation is that a successful search will frequently be followed relatively soon by a search for the same value. One way to handle this would be for a successful search to remove the value from the list and reinsert it at the front. By doing so, the subsequent search will be much faster.

A data structure that tries to optimize future performance based on the frequency of past operations is called *self-organizing*. We will subsequently encounter a number of other self-organizing data structures. Given the right circumstances self-organization can be very effective. The following chart shows the results of one simple experiment using this technique.



Simple Set Operations

Any bag can be used to construct a set. Among the basic functions the only change is in the addition operation, which must check that the value is not already in the collection. Operations such as set union, intersection and difference can all be implemented with simple loops. The following pseudo-code shows set union:

```
setUnion (one, two, three) /* assume set three is initially empty */
  for each element in one
    if value is found in two
      add to set three
```

The algorithmic execution time for this operation depends upon a combination of the time to search the second set, and the time to add into the third set. If we use a simple dynamic array or linked list bag, then both of these operations are O(n) and so, since the outer loop cycles over all the elements in the first bag, the complete union operation is $O(n^2)$.

Simple algorithms for set intersection, difference, and subset are similar, with similar performance. These are analyzed in questions at the end of this chapter. Faster set algorithms require adding more structure to the collection, such as keeping elements in sorted order.

The Bit Set

A specialized type of set is used to represent positive integers from a small range, such as a set drawn from the values between 0 and 75. Because only integer values are allowed, this can be represented very efficiently as binary values, and is therefore termed a *bit set*. Worksheet 25 explores the implementation of the bit set.

Advanced Bag Implementation Techniques

Several other bag implementation techniques use radically different approaches. Indeed, three of the following chapters will be devoted in some fashion or another to bag data structures. Chapter 9 explores the saving possible when elements are maintained in order. Chapter 10 introduces a new type of list, the skip list, as well as an entirely different form of organization, the binary tree. Finally, Chapter 12 introduces yet another technique, hashing, which produces some of the fastest implementations of bag operations.

Self Study Questions

1. What features characterize the Bag data type?

2. How is a set different from a bag?

3. When a dynamic array is used as a bag, how is the add operation different from operations you have already implemented in the dynamic array stack?

4. When a dynamic array is used as a set, how is the add operation different from operations you have already implemented in the dynamic array stack?

5. Using a dynamic array as the implementation technique, what is the big-oh algorithmic execution time for each of the bag operations? Is this different if you use a linked list as the implementation technique?

6. What is an iterator? What purpose does the iterator address?

7. Any bag can be used as the basis for a set. When using a dynamic array or linked list as the bag, what is the algorithmic execution time for the set operations?

8. What is a bit set?

Short Exercises

1. Assume you wanted to define an equality operator for bags. A bag is equal to another bag if they have the same number of elements, and each element occurs in the other bag the same number of times. Notice that the order of the elements is unimportant. If you implement bags using a dynamic array, how would you implement the equality-testing operator? What is the algorithmic complexity of your operation? Can you think of any changes to the representation that would make this operation faster?

Analysis Exercises

1. There are two simple approaches to implementing bag structures; using dynamic arrays or linked list. The only difference in algorithmic execution time performance between these two data structures is that the linked list has guaranteed constant time insertion, while the dynamic array only has amortized constant time insertion. Can you design an experiment that will determine if there is any measurable difference caused by this? Does your experiment yield different results if a test for inclusion is performed more frequently that insertions?

2. What should the remove function for a bag do if no matching value is identified? One approach is to silently do nothing. Another possibility is to return a Boolean flag indicating whether or not removal was performed. A third possibility is to throw an exception or assertion error. Compare these three possibilities and give advantages and disadvantages of each.

3. Finish describing the naïve set algorithms that are built on top of bag abstractions. Then, using invariants, provide an informal proof of correctness for each of these algorithms.

4. Assume that you are assigned to test a bag implementation. Knowing only the bag interface, what would be some good example test cases? What are some of the boundary values identified in the specification?

5. The entry for Bag in the *Dictionary of Algorithms and Data Structures* suggests a number of axioms that can be used to characterize a bag. Can you define test cases that would exercise each of these axioms? How do your test cases here differ from those suggested in the previous question.

6. Assume that you are assigned to test a set implementation. Knowing only the set interface, what would be some good example test cases? What are some of the boundary values identified in the specification?

7. The bag data type allows a value to appear more than once in the collection. The remove operation only removes the first matching value it finds, potentially leaving other occurrences of the value remaining in the collection. Suppose you wanted to implement a removeAll operation, which removed all occurrences of the argument. If you did so by making repeated calls on remove what would be the algorithmic complexity of this operation? Can you think of a better approach if you are using a dynamic array as the underlying container? What about if you are using a linked list?

Programming Projects

1. Implement an experiment designed to test the two simple bag implementations, as described in analysis exercise 1.

2. Using either of the bag implementations implement the simple set algorithms and empirically verify that they are $O(n^2)$. Do this by performing set unions for two sets

with n elements each, of various values of n. Verify that as n increases the time to perform the union increases as a square. Plot your results to see the quadratic behavior.

On the Web

The wikipedia has (at the time of writing) no entry for the bag data structure, but does have an entry for the related data type termed the *multiset*. Other associated entries include bitset (termed a bit array or bitmap). The *Dictionary of Algorithms and Data Structures* does have an entry that describes the bag.

Chapter 9: Searching, Ordered Collections

In Chapter 8 we said that a *bag* was a data structure characterized by three primary operations. These operations were inserting a new value into the bag, testing a bag to see if it contained a particular value, and removing an element from the bag. In our initial description we treated all three operations as having equal importance. In many applications, however, one or another of the three operations may occur much more frequently than the others. In this situation it may be useful to consider alternative implementation techniques. Most commonly the favored operation is searching.

We can illustrate an example with the questions examined back in Chapter 4. Why do dictionaries or telephone books list their entries in sorted order? To understand the reason, Chapter 4 posed the following two questions. Suppose you were given a telephone book and asked to find the number for an individual named Chris Smith. Now suppose you were asked to find the name of the person who has telephone number 543-7352. Which task is easier?

```
Abby Smith   954-9845
Chris Smith  234-7832
Fred Smith   435-3545
Jaimie Smith 845-2395
Robin Smith  436-9834
```

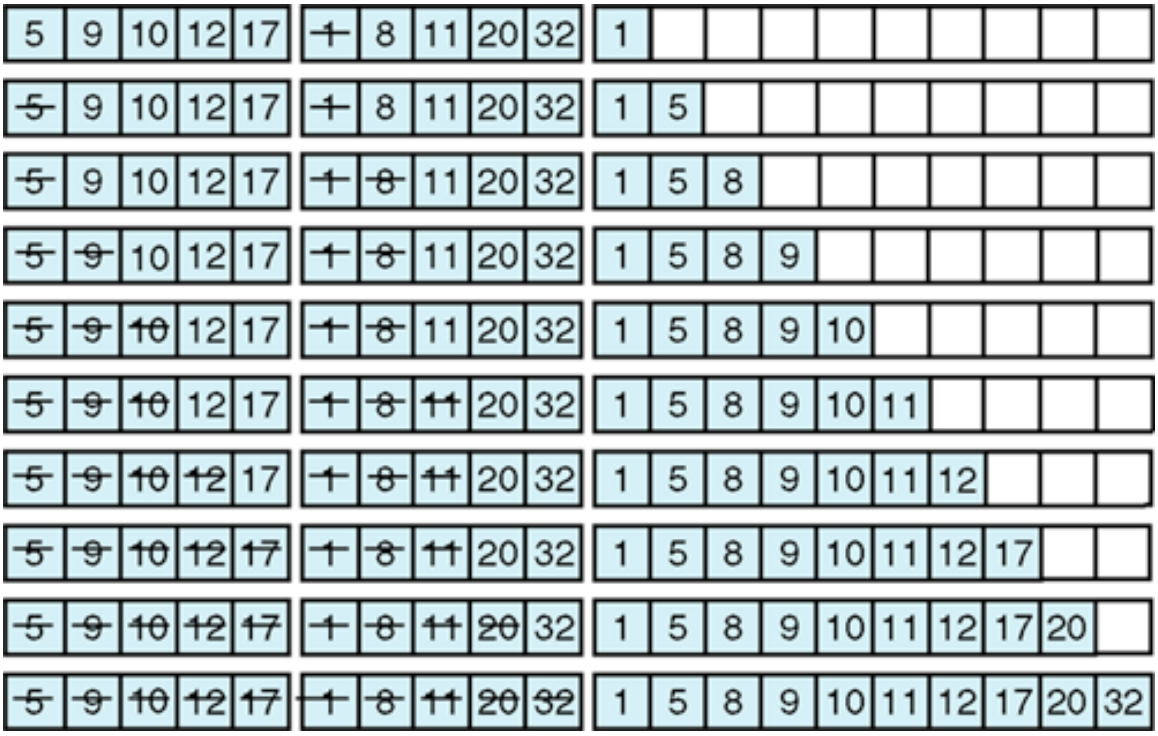
The difference in the two tasks represents the difference between a *sequential*, or *linear* search and a *binary search*. A linear search is basically the approach used in our Bag abstraction in Chapter 8, and the approach that you by necessity need to perform if all you have is an unsorted list. Simply compare the element you seek to each value in the collection, element by element, until either you find the value you want, or exhaust the collection. A binary search, on the other hand, is much faster, but works only for ordered lists. You start by comparing the test value to the element in the middle of the collection. In one step you can eliminate half the collection, continuing the search with either the first half or the last half of the list. If you repeat this halving idea, you can search the entire list in O(log n) steps. As the thought experiment with the telephone book shows, binary search is much faster than linear search. An ordered array of one billion elements can be searched using no more than twenty comparisons using a binary search.

Before a collection can be searched using a binary search it must be placed in order. There are two ways this could occur. Elements can be inserted, one by one, and ordered as they are entered. The second approach is to take an unordered collection and rearrange the values so that they become ordered. This process is termed *sorting*, and you have seen several sorting algorithms already in earlier chapters. As new data structures are introduced in later chapters we will also explore other sorting algorithms.

Fast Set Operations

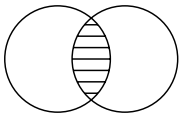
Keeping a dynamic array in sorted order allows a fast search by means of the binary search algorithms. But there is another reason why you might want to keep a dynamic array, or even a linked list, in sorted order. This is that two sorted collections can be *merged* very quickly into a new, also sorted collection. Simply walk down the two collections in order, maintaining an index into each one. At each step select the smallest

value, and copy this value into the new collection, advancing the index. When one of the pointers reaches the end of the collection, all the values from the remaining collection are copied.



The merge operation by itself is sometimes a good enough reason to keep a sorted collection. However, more often this technique is used as a way to provide fast set operations. Recall that the *set* abstraction is similar to a bag, with two important differences. First, elements in a set are unique, never repeated. Second, a set supports a number of collection with collection operations, such as set union, set intersection, set difference, and set subset.

All of the set operations can be viewed as a variation on the idea of merging two ordered collections. Take, for example, set intersection. To form the intersection simply walk down the two collections in order. If the element in the first is smaller than that in the second, advance the pointer to the first. If the element in the second is smaller than that in the first, advance the pointer to the second. Only if the elements are both equal is the value copied into the set intersection.



This approach to set operations can be implemented using either ordered dynamic arrays, or ordered linked lists. In practice ordered arrays are more often encountered, since an ordered array can also be quickly searched using the binary search algorithm.

Implementation of Ordered Bag using an Ordered Array

In an earlier chapter you encountered the *binary search* algorithm. The version shown below takes as argument the value being tested, and returns in O(log n) steps either the location at which the value is found, *or if it is not in the collection* the location the value can be inserted and still preserve order.

```
int binarySearch (TYPE * data, int size, TYPE testValue) {
  int low = 0;
  int high = size;
  int mid;
  while (low < high) {
    mid = (low + high) / 2;
    if (LT(data[mid], testValue))
      low = mid + 1;
    else
      high = mid;
  }
  return low;
}
```

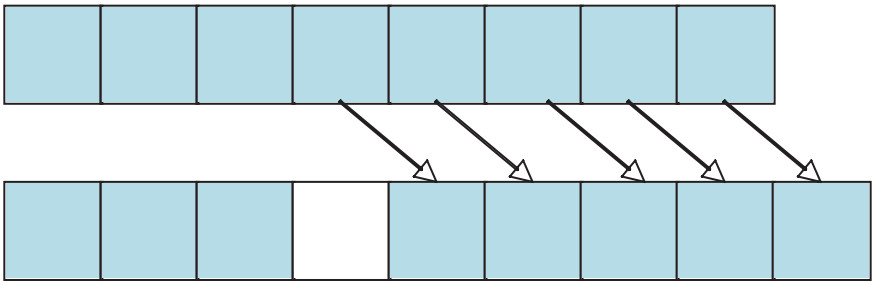
Consider the following array, and trace the execution of the binary search algorithm as it searches for the element 7:

2	4	5	7	8	12	24	37	40	41	42	50	68	69	72

Do the same for the values 1, 25, 37 and 76. Notice that the value returned by this function need not be a legal index. If the test value is larger than all the elements in the array, the only position where an insertion could be performed and still preserve order is the next index at the top. Thus, the binary search algorithm might return a value equal to size, which is not a legal index.

If we used a dynamic array as the underlying container, and if we kept the elements in sorted order, then we could use the binary search algorithm to perform a very rapid contains test. Simply call the binary search algorithm, and save the resulting index position. Test that the index is legal; if it is, then test the value of the element stored at the position. If it matches the test argument, then return true. Otherwise return false. Since the binary search algorithm is O(log n), and all other operations are constant time, this means that the contains test is O(log n), which is much faster than either of the implementations you developed in the preceding chapter.

Inserting a new value into the ordered array is not quite as easy. True, we can discover the position where the insertion should be made by invoking the binary search algorithm. But then what? Because the values are stored in a block, the problem is in many ways the opposite of the one you examined in Chapter 8. Now, instead of moving values down to delete an entry, we must here move values up to make a “hole” into which the new element can be placed:



As we did with remove, we will divide this into two steps. The add function will find the correct location at which to insert a value, then call another function that will insert an element at a given location:

```
void addOrderedArray (struct dynArray *dy, TYPE newElement) {
  int index = binarySearch(dy->data, dy->size, newElement);
  addAtDynArray (dy, index, newElement);
}
```

The method addAt must check that the size is less than the capacity, calling setCapacity if not, loop over the elements in the array in order to open up a hole for the new value, insert the element into the hole, and finally update the variable count so that it correctly reflects the number of values in the container.

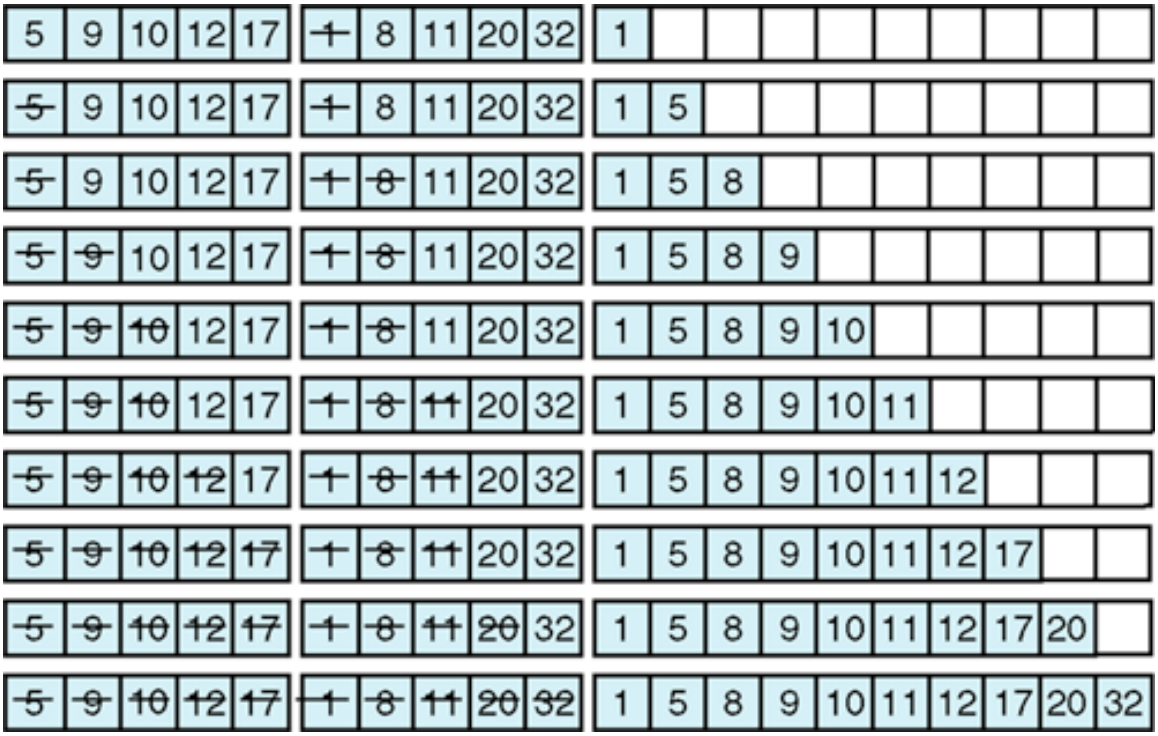
```
void addAtDynArray (struct dynArray *dy, int index, TYPE newElement) {
  int i;
  assert(index > 0 && index <= dy->size);
  if (dy->size >= dy->capacity)
          _setCapacityDynArray(dy, 2 * dy->capacity);
  …  /* you get to fill this in */
}
```

The function remove could use the same implementation as you developed in Chapter 8. However, whereas before we used a linear search to find the position of the value to be deleted, we can here use a binary search. If the index returned by binary search is a legal position, then invoke the method removeAt that you wrote in Chapter 8 to remove the value at the indicated position.

In worksheet 26 you will complete the implementation of a bag data structure based on these ideas.

Fast Set Operations for Ordered Arrays

Two sorted arrays can be easily merged into a new array. Simply walk down both input arrays in parallel, selecting the smallest element to copy into the new array:



The set operations of union, intersection, difference and subset are each very similar to a merge. That is, each algorithm can be expressed as a parallel walk down the two input collections, advancing the index into the collection with the smallest value, and copying values into a new vector.

```
void setIntersectArray (struct dynArray *left, struct dynArray *right, struct dynArray *to) {
	int i = 0;
	int j = 0;
	while ((i < sizeDynArray(left)) && (j < sizeDynArray(right))) {
		if (LT(getDynArray(left, i), getDynArray(right, j)) {
			i++;
		} else if (EQ(getDynArray(left, i), getDynArray(right, j))) {
			addDynArray(to, getDynArray(left, i)); i++; j++;
		} else {
			j++;
		}
	}
}
```

Take, for example, set intersection. The intersection copies a value when it is found in both collections. Notice that in this abstraction it is more convenient to have the set operations create a new set, rather than modifying the arguments. Union copies the smaller element when they are unequal, and when they are equal copies only one value and advances both pointers (remember that in a set all elements are unique, each value appears only once). The difference copies values from the first collection when it is smaller than the current element in the second, and ignores elements that are found in both collections. Finally there is the subset test. Unlike the others this operation does not

produce a new set, but instead returns false if there are any values in the first collection that are not found in the second. But this is the same as returning false if the element from the left set is ever the smallest value (indicating it is not found in the other set).

Question: The parallel walk halts when one or the other array reaches the end. In the merge algorithm there was an additional loop after the parallel walk needed to copy the remaining values from the remaining array. This additional step is necessary in some but not all of the set operations. Can you determine which operations need this step?

In worksheet 27 you will complete the implementation of the sorted array set based on these ideas.

In Chapter 8 you developed set algorithms that made no assumptions concerning the ordering of elements. Those algorithms each have $O(n^2)$ behavior, where n represents the number of elements in the resulting array. What will be the algorithmic execution times for the new algorithms?

	Dynamic Array Set	Sorted Array Set
unionWith	$O(n^2)$	O(
intersectionWith	$O(n^2)$	O(
differenceWith	$O(n^2)$	O(
subset	$O(n^2)$	O(

Set operations Using Ordered Linked Lists

We haven't seen ordered linked lists yet for the simple reason that there usually isn't much reason to maintain linked lists in order. Searching a linked list, even an ordered one, is still a sequential operation and is therefore O(n). Adding a new element to such a list still requires a search to find the appropriate location, and there therefore also O(n). And removing an element involves finding it first, and is also O(n). Since none of these are any better than an ordinary unordered linked list, why bother?

One reason is the same as the motivation for maintaining a dynamic array in sorted order. We can quickly merge two ordered linked lists to produce a new list that is also sorted. And, as we discovered in the last worksheet, the set operations of intersection, union, difference and subset can all be thought of as simple variations on the idea of a merge. Thus we can quickly make fast implementations of all these operations as well.

The motivation for keeping a list sorted is not the same as it was for keeping a vector sorted. With a vector one could use binary search quickly find whether a collection contained a specific value. The sequential nature of a linked list prevents the use of the binary search algorithm (but not entirely, as we will see in a later chapter).

Self Study Questions

1. What two reasons are identified in this chapter for keeping the elements of a collection in sorted order?

2. What is the algorithmic execution time for a binary search? What is the time for a linear search?

3. If an array contains n values, what is the range of results that the binary search algorithm might return?

4. The function dyArrayGet produced an assertion error if the index value was larger than or equal to the array size. The function dyArrayAddAt, on the other hand, allows the index value to be equal to the size. Explain why. (Hint: How many locations might a new value be inserted into an array).

5. Explain why the binary search algorithm speeds the test operation, but not additions and removals.

6. Compare the algorithm execution times of an ordered array to an unordered array. What bag operations are faster in the ordered array? What operations are slower?

7. Explain why merging two ordered arrays can be performed very quickly.

8. Explain how the set operations of union and instersection can be viewed as variations on the idea of merging two sets.

Short Exercises

1. Show the sequence of index values examined when a binary search is performed on the following array, seeking the value 5. Do the same for the values 14, 41, 70 and 73.

2	4	5	7	8	12	24	37	40	41	42	50	68	69	72

Analysis Exercises

1. The binary search algorithm as presented here continues to search until the range of possible insertion points is reduced to one. If you are searching for a specific value, however, you might be lucky and discover it very quickly, before the values low and high meet. Rewrite the binary search algorithm so that it halts if the element examined at position mid is equal to the value being searched for. What is the algorithmic complexity of your new algorithm? Perform an experiment where you search for random values in a given range. Is the new algorithm faster or slower than the original?

2. Provide invariants that could be used to produce a proof of correctness for the binary search algorithm. Then provide the arguments used to join the invariants into a proof.

3. Provide invariants that could be used to produce a proof of correctness for the set intersection algorithm. Then provide the arguments used to join the invariants into a proof.

4. Do the same for the intersection algorithm.

5. Two sets are equal if they have exactly the same elements. Show how to test equality in O(n) steps if the values are stored in a sorted array.

6. A set is considered a subset of another set if all values from the first are found in the second. Show how to test the subset condition in O(n) steps if the values are stored in a sorted array.

7. The binary search algorithm presented here finds the midpoint using the formula (low + high) / 2. In 2006, Google reported finding an error that was traced to this formula, but occurred only when numbers were close to the maximum integer size. Explain what error can occur in this situation. This problem is easily fixed by using the alternative formula low + (high – low) / 2. Verify that the values that cause problems with the first formula now work with the second. (See http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html for a discussion of the bug)

Programming Projects

1. Rewrite the binary search algorithm so that it halts if it finds an element that is equal to the test value. Create a test harness to test your new algorithm, and experimentally compare the execution time to the original algorithm.

2. Write the function to determine if two sets are equal, as described in an analysis exercise above. Write the similar function to determine if one set is a subset of the second.

3. Create a test harness for the sorted dynamic array bag data structure. Then create a set of test cases to exercise boundary conditions. What are some good test cases for this data structure?

On the Web

Wikipedia contains a good explanation of binary search, as well as several variations on binary search. Binary search is also explained in the *Dictionary of Algorithms and Data Structures.*

Chapter 10: Efficient Collections (skip lists, trees)

If you performed the analysis exercises in Chapter 9, you discovered that selecting a bag-like container required a detailed understanding of the tasks the container will be expected to perform. Consider the following chart:

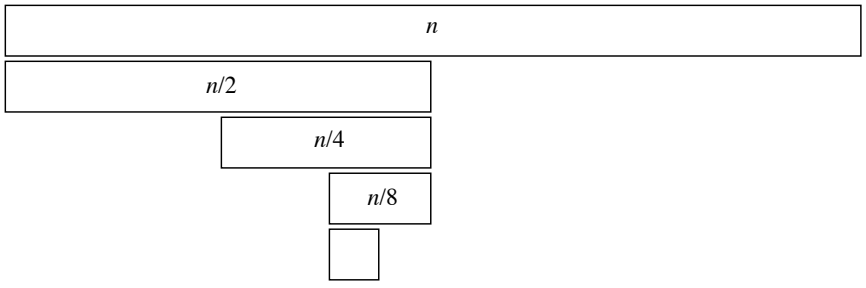
	Dynamic array	Linked list	Ordered array
add	O(1)+	O(1)	O(n)
contains	O(n)	O(n)	O(log n)
remove	O(n)	O(n)	O(n)

If we are simply considering the cost to insert a new value into the collection, then nothing can beat the constant time performance of a simple dynamic array or linked list. But if searching or removals are common, then the O(log n) cost of searching an ordered list may more than make up for the slower cost to perform an insertion. Imagine, for example, an on-line telephone directory. There might be several million search requests before it becomes necessary to add or remove an entry. The benefit of being able to perform a binary search more than makes up for the cost of a slow insertion or removal.

What if all three bag operations are more-or-less equal? Are there techniques that can be used to speed up all three operations? Are arrays and linked lists the only ways of organizing a data for a bag? Indeed, they are not. In this chapter we will examine two very different implementation techniques for the Bag data structure. In the end they both have the same effect, which is providing O(log n) execution time for all three bag operations. However, they go about this task in two very different ways.

Achieving Logarithmic Execution

In Chapter 4 we discussed the log function. There we noted that one way to think about the log was that it represented "the number of times a collection of n elements could be cut in half". It is this principle that is used in binary search. You start with n elements, and in one step you cut it in half, leaving n/2 elements. With another test you have reduced the problem to n/4 elements, and in another you have n/8. After approximately log n steps you will have only one remaining value.



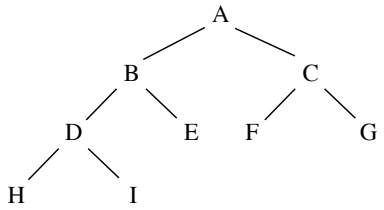
There is another way to use the same principle. Imagine that we have an organization based on layers. At the bottom layer there are n elements. Above each layer is another

that has approximately half the number of elements in the one below. So the next to the bottom layer has approximately n/2 elements, the one above that approximately n/4 elements, and so on. It will take approximately log n layers before we reach a single element.

How many elements are there altogether? One way to answer this question is to note that the sum is a finite approximation to the series n + n/2 + n/4 + … . If you factor out the common term n, then this is 1 + ½ + ¼ + … This well known series has a limiting value of 2. This tells us that the structure we have described has approximately 2n elements in it.

The *skip list* and the *binary tree* use these observations in very different ways. The first, the skip list, makes use of non-determinism. Non-determinism means using random chance, like flipping a coin. If you flip a coin once, you have no way to predict whether it will come up heads or tails. But if you flip a coin one thousand times, then you can confidently predict that about 500 times it will be heads, and about 500 times it will be tails. Put another way, randomness in the small is unpredictable, but in large numbers randomness can be very predictable. It is this principle that casinos rely on to ensure they can always win.

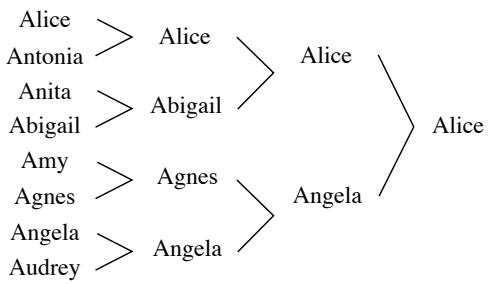
The second technique makes use of a data organization technique we have not yet seen, the binary tree. A binary tree uses nodes, which are very much like the links in a linked list. Each node can have zero, one, or two children.



Compare this picture to the earlier one. Look at the tree in levels. At the first level (termed the root) there is a single node. At the next level there can be at most two, and the next level there can be at most four, and so on. If a tree is relatively "full" (a more precise definition will be given later), then if it has n nodes the height is approximately log n.

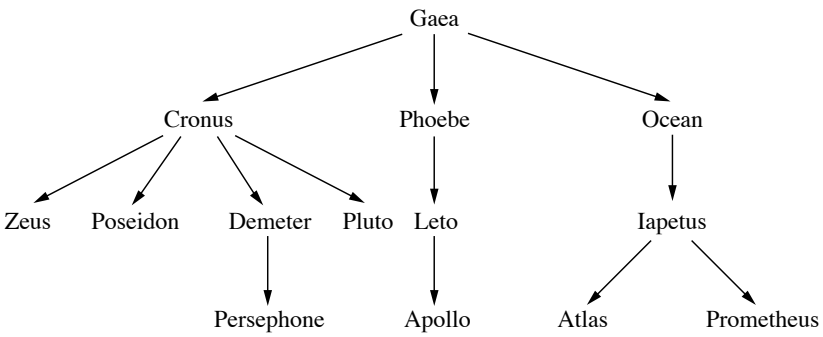
Tree Introduction

Trees are the third most used data structure in computer science, after arrays (including dynamic arrays and other array variations) and linked lists. They are ubiquitous, found everywhere in computer algorithms.

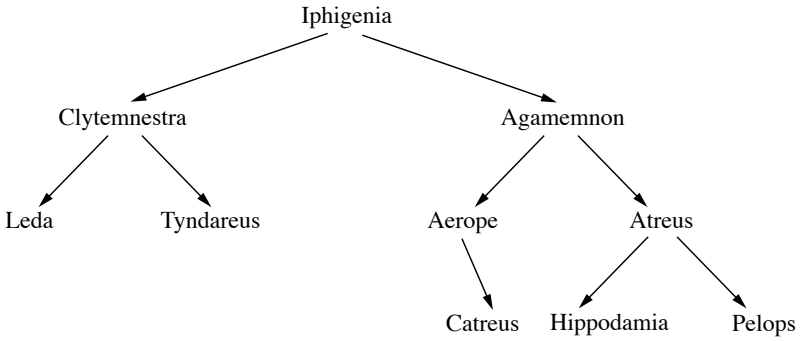


Just as the intuitive concepts of a stack or a queue can be based on everyday experience with similar structures, the idea of a tree can also be found in everyday life, and not just the arboreal variety. For example, sports events are often organized using binary trees, with each node representing a pairing of contestants, the winner of each round advancing to the next level.

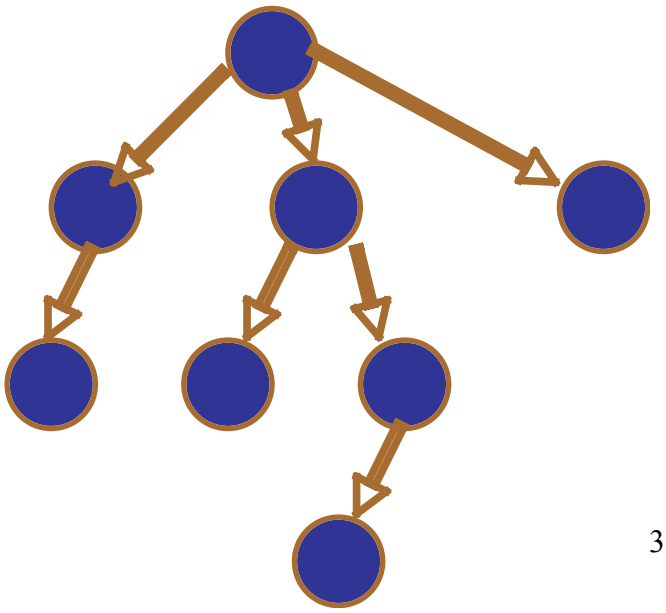
Information about ancestors and descendants is often organized into a tree structure. The following is a typical family tree.



In a sense, the inverse of a family tree is an ancestor tree. While a family tree traces the descendants from a single individual, an ancestor tree records the ancestors. An example is the following. We could infer from this tree, for example, that Iphigenia is the child of Clytemnestra and Agamemnon, and Clytemnestra is in turn the child of Leda and Tyndareus.



The general characteristics of trees can be illustrated by these examples. A tree consists of a collection of *nodes* connected by directed *arcs*. A tree has a single *root node*. A node that points to other nodes is termed the *parent* of those nodes while the nodes pointed to are the *children*. Every node except the root has exactly one parent. Nodes with no children are termed *leaf nodes*, while nodes with children are termed *interior nodes*. Identify the root, children of the root, and leaf nodes in the following tree.



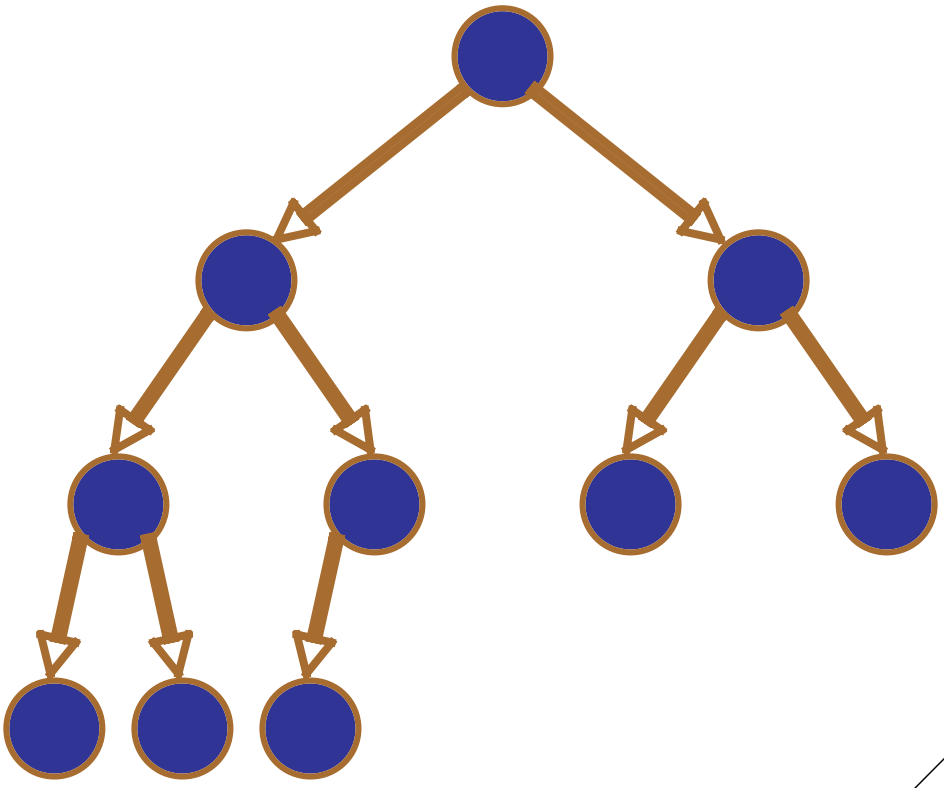
There is a single unique path from the root to any node; that is, arcs don't join together. A path's *length* is equal to the number of arcs traversed. A node's *height* is equal to the maximum path length from that node to a left node. A leaf node has height 0. The height of a tree is the height of the root. A nodes depth is equal to the path length

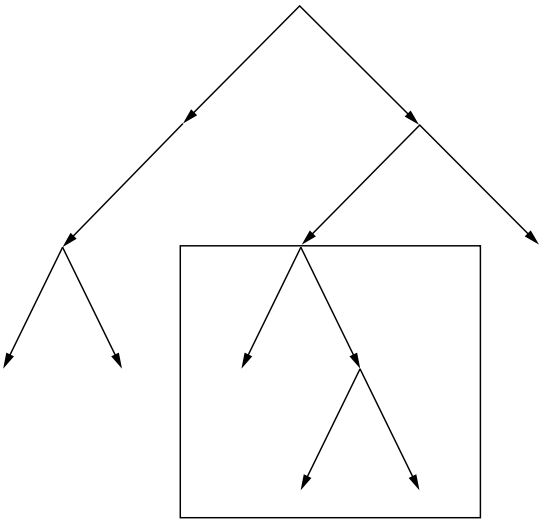
from root to that node. The root has depth 0.

A *binary tree* is a special type of tree. Each node has at most two children. Children are either left or right.

Question: Which of the trees given previously represent a binary tree?

A *full* binary tree is a binary tree in which every leaf is at the same depth. Every internal node has exactly 2 children. Trees with a height of n will have $2^{n+1} - 1$ nodes, and will have 2^n leaves. A *complete* binary tree is a full tree except for the bottom level, which is filled from left to right. How many nodes can there be in a complete binary tree of height n? If we flip this around, what is the height of a complete binary tree with n nodes?

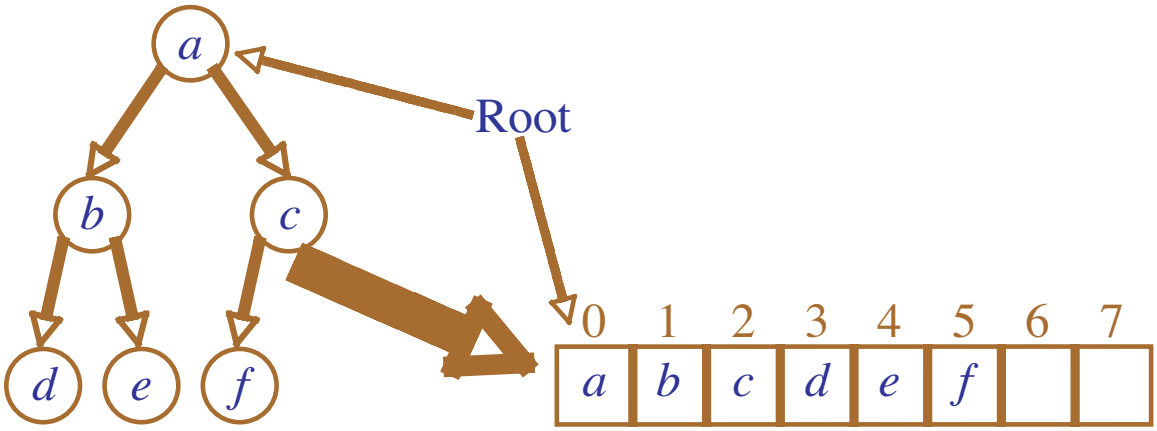




Trees are a good example of a *recursive* data structure. Any node in a tree can be considered to be a root of its own tree, consisting of the values below the node.

A binary tree can be efficiently stored in an array. The root is stored at position 0, and the children of the node stored at position i are stored in positions

2i+1 and 2i+2. The parent of the node stored at position i is found at position floor((i-1)/2).



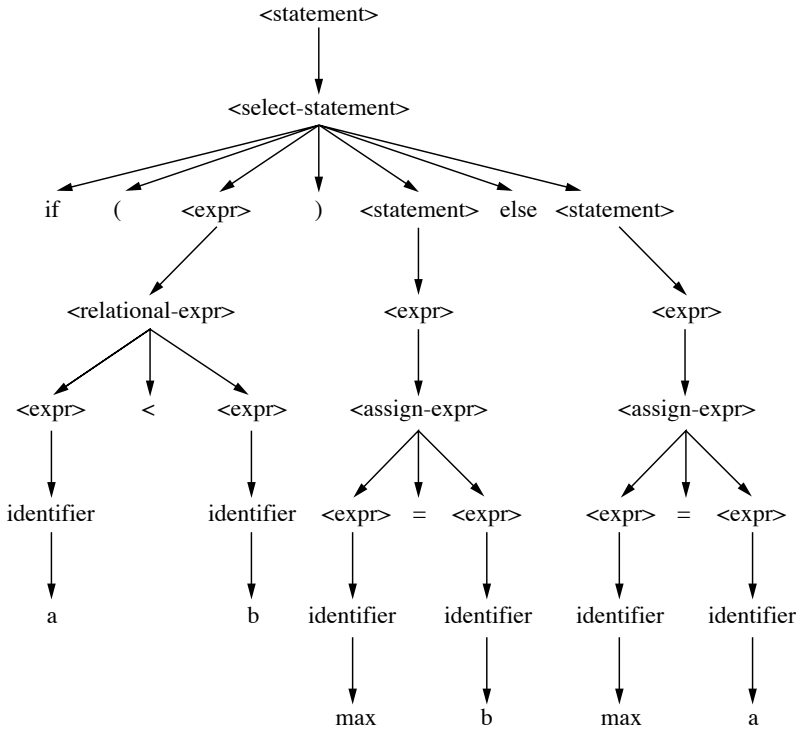
Question: What can you say about a complete binary tree stored in this representation? What will happen if the tree is not complete?

```
struct Node {
   TYPE value;
   struct Node * left;
   struct Node * right;
}
```

More commonly we will store our binary trees in instances of struct **Node**. This is very similar to the struct DLink we used in the linked list. Each node will have a value, and a left and right child, as opposed to the next and prev pointers of the DLink.

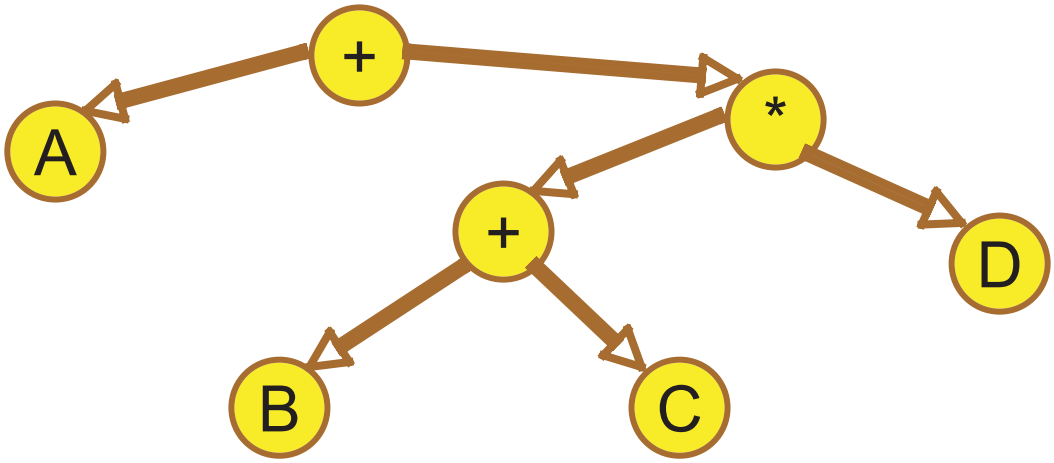
Trees appear in computer science in a surprising large number of varieties and forms. A common example is a parse tree. Computer languages, such as C, are defined in part using a grammar. Grammars provide rules that explain how the tokens of a language can be put together. A statement of the language is then constructed using a tree, such as the one shown.

Leaf nodes represent the tokens, or symbols, used in the statement. Interior nodes represent syntactic categories.



Syntax Trees and Polish Notation

A tree is a common way to represent an arithmetic expression. For example, the following tree represents the expression A + (B + C) * D. As you learned in Chapter 5, Polish notation is a way of representing expressions that avoids the need for parentheses by writing the operator for an expression first. The polish notation form for this expression is + A * + B C D. Reverse polish writes the operator after an expression, such as A B C + D * +. Describe the results of each of the following three traversal algorithms on the following tree, and give their relationship to polish notation.

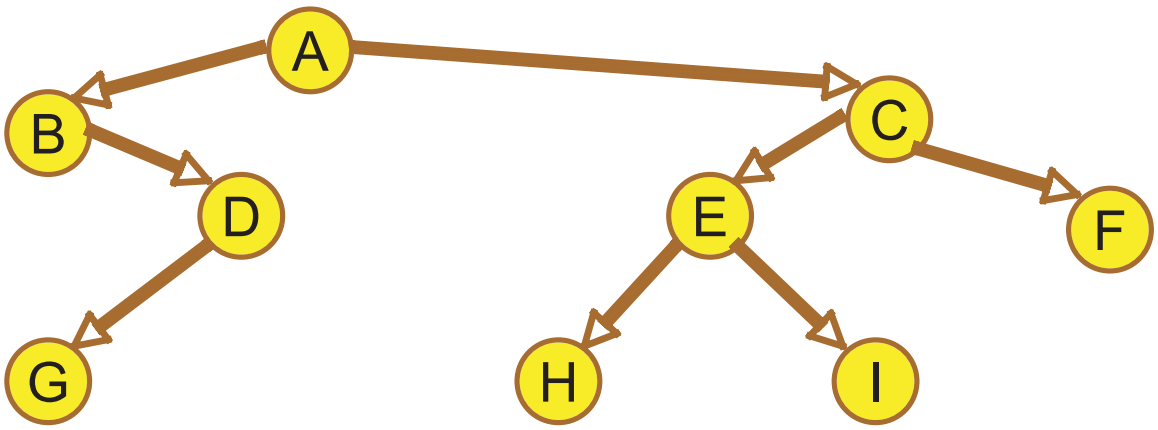


Tree Traversals

Just as with a list, it is often useful to examine every node in a tree in sequence. This is a termed a *traversal*. There are four common traversals:

preorder: Examine a node first, then left children, then right children
Inorder: Examine left children, then a node, then right children
Postorder: Examine left children, then right children, then a node
Levelorder: Examine all nodes of depth n first, then nodes depth n+1, etc

Question: Using the following tree, describe the order that nodes would be visited in each of the traversals.



In practice the inorder traversal is the most useful. As you might have guessed when you simulated the inorder traversal on the above tree, the algorithm makes use of an internal stack. This stack represents the current path that has been traversed from root to left. Notice that the first node visited in an inorder traversal is the leftmost child of the root. A useful function for this purpose is slideLeft. Using the slide left routine, you should verify that the following algorithm will produce a traversal of a binary search tree. The algorithm is given in the form of an iterator, consisting of tree parts: initialization, test for completion, and returning the next element:

```
slideLeft(node n)
  while n is not null
    add n to stack
    n = left child of n
```

```
initialization:
   create an empty stack

has next:
  if stack is empty
     perform slide left on root
  otherwise
     let n be top of stack. Pop topmost element
     slide left on right child of n
  return true if stack is not empty

current element:
  return value of top of stack
```

You should simulate the traversal on the tree given earlier to convince yourself it is visiting nodes in the correct order.

Although the inorder traversal is the most useful in practice, there are other tree traversal algorithms. Simulate each of the following, and verify that they produce the desired traversals. For each algorithm characterize (that is, describe) what values are being held in the stack or queue.

```
PreorderTraversal
  initialize an empty stack
```

```
 has next
    if stack is empty then push the root on to the stack
    otherwise
      pop the topmost element of the stack,
      and push the children from left to right
    return true if stack is not empty

 current element:
    return the value of the current top of stack

PostorderTraversal
  intialize a stack by performing a slideLeft from the root

 has Next
    if top of stack has a right child
      perform slide left from right child
    return true if stack is not empty

  current element
    pop stack and return value of node

LevelorderTraversal
  initialize an empty queue

 has Next
    if queue is empty then push root in to the queue
    otherwise
      pop the front element from the queue
      and push the children from right to left in to the queue
    return true if queue is not empty

  current element
    return the value of the current front of the queue
```

Euler Tours

The three common tree-traversal algorithms can be unified into a single algorithm by an approach that visits every node three times. A node will be visited before its left children (if any) are explored, after all left children have been explored but before any right child, and after all right children have been explored. This traversal of a tree is termed an *Euler tour*. An Euler tour is like a walk around the perimeter of a binary tree.

```
 void EulerTour (Node n ) {
    beforeLeft(n);
    if (n.left != null) EulerTour (n.left);
```

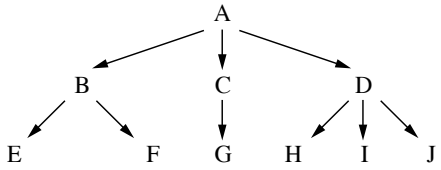
```
    inBetween (n);
    if (n.right != null) EulerTour (n.right);
    afterRight(n);
  }

  void beforeLeft (Node n) { … }
  void inBetween (Node n) { … }
  void afterRight (Node n) { … }
```

The user constructs an Euler tour by providing implementations of the functions beforeLeft, inBetweeen and afterRight. To invoke the tour the root node is passed to the function EulerTour. For example, if a tree represents an arithmetic expression the following could be used to print the representation.

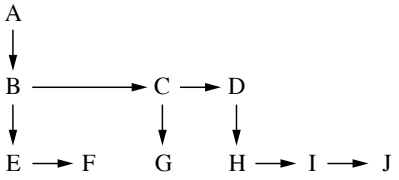
```
  void beforeLeft (Node n) { print("("); }
  void inBetween (Node n) { printl(n.value); }
  void afterRight (Node n) { print(")"); }
```

Binary Tree Representation of General Trees

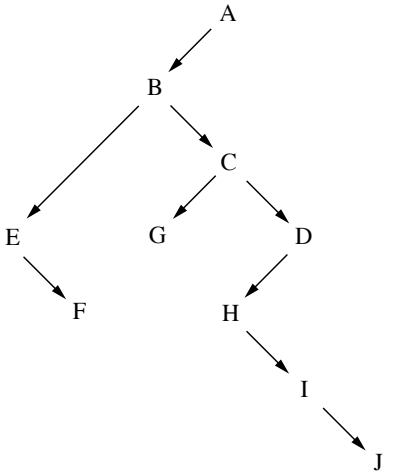
The binary tree is actually more general than you might first imagine. For instance, a binary tree can actually represent any tree structure. To illustrate this, consider an example tree such as the following:



To represent this tree using binary nodes, use the left pointer on each node to indicate the first child of the current node, then use the right pointer to indicate a "sibling", a child with the same parents as the current node. The tree would thus be represented as follows:



Turning the tree 45 degrees makes the representation look more like the binary trees we have been examining in earlier parts of this chapter:



Question: Try each of the tree traversal techniques described earlier on this resulting tree. Which algorithms correspond to a traversal of the original tree?

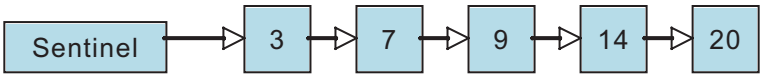
Efficient Logrithmic Implementation Techniques

In the worksheets you will explore two different efficient (that is, logarithmic) implementation techniques. These are the *skip list* and the *balanced binary tree*.

The Skip List

The SkipList is a more complex data structure than we have seen up to this point, and so we will spend more time in development and walking you through the implementation. To motivate the need for the skip list, consider that ordinary linked lists and dynamic arrays have fast (O(1)) addition of new elements, but a slow time for search and removal. A sorted array has a fast O(log n) search time, but is still slow in addition of new elements and in removal. A skip list is fast O(log n) in all three operations.

We begin the development of the skip list by first considering an simple ordered list, with a sentinel on the left, and single links:



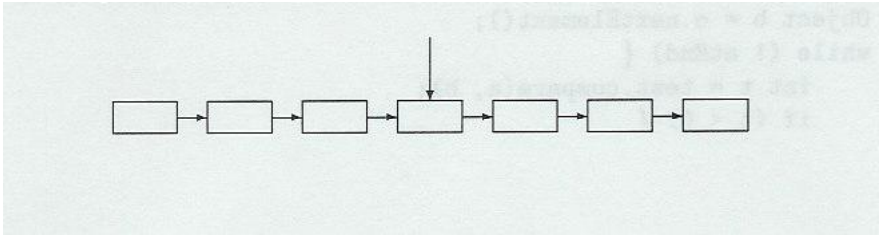
To add a new value to such a list you find the correct location, then set the value. Similarly to see if the list contains an element you find the correct location, and see if it is what you want. And to remove an element, you find the correct location, and remove it. Each of these three has in common the idea of finding the location at which the operation will take place. We can generalize this by writing a common routine, named **slideRight**. This routine will move to the right as long as the next element is smaller than the value being considered.

```
slide right (node n, TYPE test) {
  while (n->next != nil and n->next->value < test)
    n = n->next;
  return n;
}
```

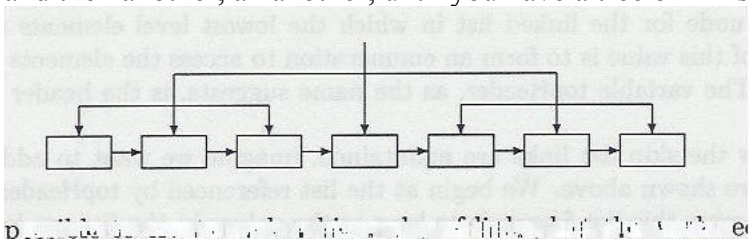
Try simulating some of the operations on this structure using the list shown until you understand what the function slideRight is doing. What happens when you insert the value 10? Search for 14? Insert 25? Remove the value 9?

By itself, this new data structure is not particularly useful or interesting. Each of the three basic operations still loop over the entire collection, and are therefore O(n), which is no better than an ordinary dynamic array or linked list.

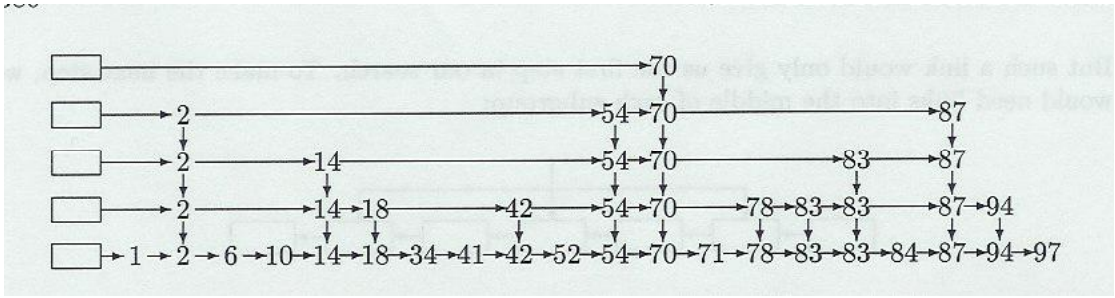
Why can't one do a binary search on a linked list? The answer is because you cannot easily reach the middle of a list. But one could imagine keeping a pointer into the middle of a list:



and then another, an another, until you have a tree of links



In theory this would work, but the effort and cost to maintain the pointers would almost certainly dwarf the gains in search time. But what if we didn't try to be precise, and instead maintained links *probabilistically*? We could do this by maintaining a stack of ordered lists, and links that could point down to the lower link. We can arrange this so that each level has approximately half the links of the next lower. There would therefore be approximately log n levels for a list with n elements.



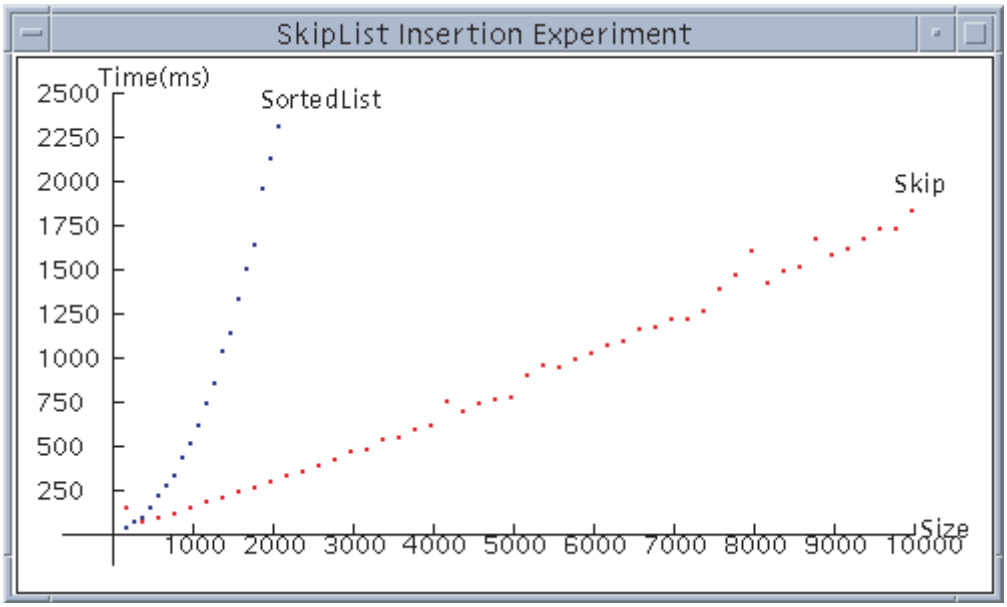
This is the basic idea of the skip list. Because each level has half the elements as the one below, the height is approximately log n. Because operations will end up being proportional to the height of the structure, rather than the number of elements, they will also be O(log n).

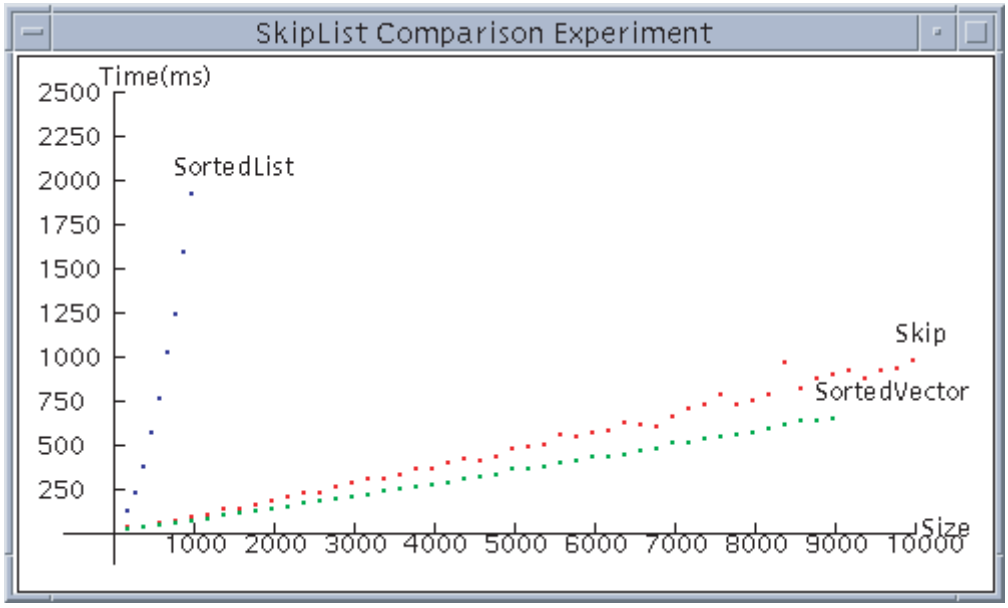
Worksheet 28 will lead you through the implementation of the skip list.

Other data structures have provided fast implementations of one or more operations. An ordinary dynamic array had fast insertion, a sorted array provided a fast search. The skip list is the first data structure we have seen that provides a fast implementation of all three of the Bag operations. This makes it a very good general-purpose data structure. The one disadvantage of the skip list is that it uses about twice as much memory as needed by the bottommost linked list. In later lessons we will encounter other data structures that also have fast execution time, and use less memory.

The skip list is the first data structure we have seen that uses *probability*, or *random chance*, as a technique to ensure efficient performance. The use of a coin toss makes the class non-deterministic. If you twice insert the same values into a skip list, you may end up with very different internal links. Random chance used in the right way can be a very powerful tool.

In one experiment to test the execution time of the skip list we compared the insertion time to that of an ordered list. This was done by adding n random integers into both collections. The results were as shown in the first graph below. As expected, insertion into the skip list was much faster than insertion into a sorted list. However, when comparing two operations with similar time the results are more complicated. For example, the second graph compares searching for elements in a sorted array versus in a skip list. Both are O(log n) operations. The sorted array may be slightly faster, however this will in practice be offset by the slower time to perform insertions and removals.



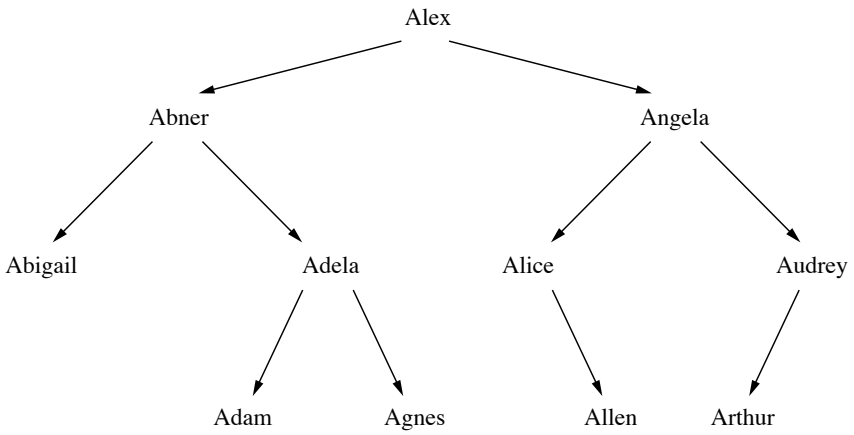


The bottom line says that to select an appropriate container one must consider the entire mix of operations a task will require. If a task requires a mix of operations the skip list is a good overall choice. If insertion is the dominant operation then a simple dynamic array or list might be preferable. If searching is more frequent than insertions then a sorted array is preferable.

The Binary Search Tree

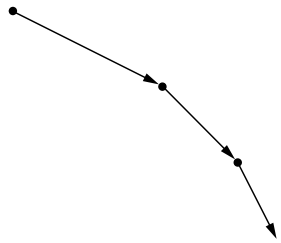
As we noted earlier in this chapter, another approach to finding fast performance is based on the idea of a binary tree. A binary tree consists of a collection of nodes. Each node can have zero, one or two children. No node can be pointed to by more than one other node. The node that points to another node is known as the parent node.

To build a collection out of a tree we construct what is termed a binary search tree. A *binary search tree* is a binary tree that has the following additional property: for each node, the values in all descendants to the left of the node are less than or equal to the value of the node, and the values in all descendants to the right are greater than or equal. The following is an example binary search tree:



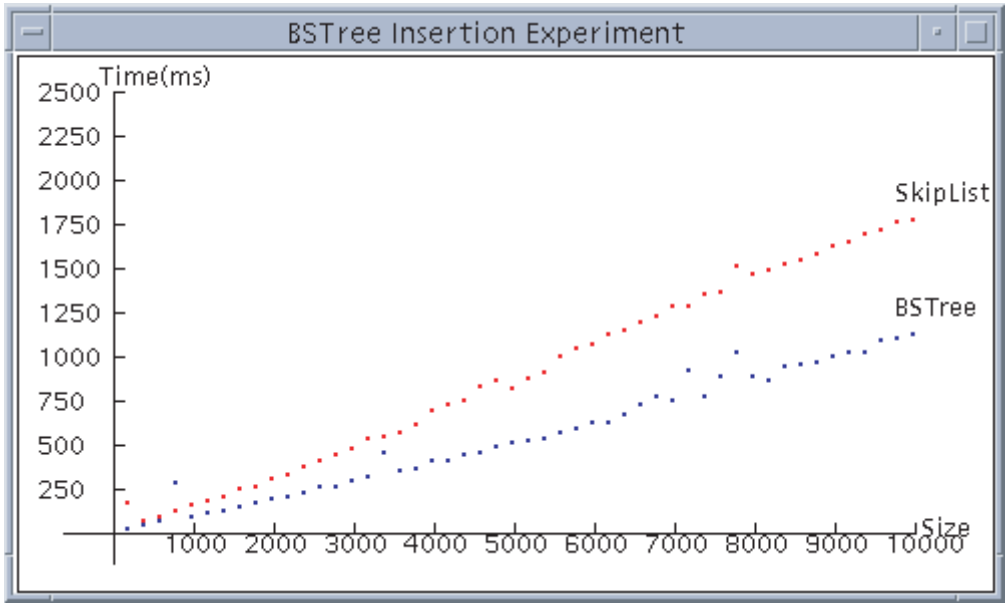
Notice that an inorder traversal of a BST will list the elements in sorted order. The most important feature of a binary search tree is that operations can be performed by walking the tree from the top (the root) to the bottom (the leaf). This means that a BST can be used to produce a fast bag implementation. For example, suppose you wish to find out if the name "Agnes" is found in the tree shown. You simply compare the value to the root (Alex). Since Agnes comes before Alex, you travel down the left child. Next you compare "Agnes" to "Abner". Since it is larger, you travel down the right. Finally you find a node that matches the value you are searching, and so you know it is in the collection. If you find a null pointer along the path, as you would if you were searching for "Sam", you would know the value was not in the collection.

Adding a value to a binary search tree is easy. You simply perform the same type of traversal as described above, and when you find a null value you insert a new node. Try inserting the value "Amina". Then try inserting "Sam".



The development of a bag abstraction based on these ideas occurs in two worksheets. In worksheet 29 you explore the basic algorithms. Unfortunately, bad luck in the order in which values are inserted into the bag can lead to very poor performance. For example, if elements are inserted in order, then the resulting tree is nothing more than a simple linked list. In Worksheet 30 you explore the AVL tree, which rebalances the tree as values are inserted in order to preserve efficient performance.

As long as the tree remains relatively well balanced, the addition of values to a binary search tree is very fast. This can be seen in the execution timings shown below. Here the time required to place n random values into a collection is compared to a Skip List, which had the fastest execution times we have seen so far.



Functional versus State Change Data Structures

In developing the AVL tree, you create a number of functions that operate differently than those we have seen earlier. Rather than making a change to a data structure, such as modifying a child field in an existing node, these functions leave the current value unchanged, and instead create a new value (such as a new subtree) in which the desired modification has been made (for example, in which a new value has been inserted). The methods **add**, **removeLeftmostChild**, and **remove** illustrate this approach to the manipulation of data structures. This technique is often termed the *functional* approach, since it is common in functional programming languages, such as ML, Clojure and Haskell. In many situations it is easier to describe how to build a new value than it is to describe how to change an existing value. Both approaches have their advantages, and you should add this new way of thinking about a task to your toolbox of techniques and remember it when you are faced with new problems.

Self Study Questions

1. In what operations is a simple dynamic array faster than an ordered array? In what operations is the ordered array faster?

2. What key concept is necessary to achieve logarithmic performance in a data structure?

3. What are the two basic parts of a tree?

4. What is a root note? What is a leaf node? What is an interior node?

5. What is the height of a binary tree?

6. If a tree contains a node that is both a root and a leaf, what can you say about the height of the tree?

7. What are the characteristics of a binary tree?

8. What is a full binary tree? What is a complete binary tree?

9. What are the most common traversals of a binary tree?

10. How are tree traversals related to polish notation?

11. What key insight allows a skip list to achieve efficient performance?

12. What are the features of a binary search tree?

13. Explain the operation of the function **slideRight** in the skip list implementation. What can you say about the link that this method returns?

14. How are the links in a skip list different from the links in previous linked list containers?

15. Explain how the insertion of a new element into a skip list uses random chance.

16. How do you know that the number of levels in a skip list is approximately log n?

Analysis Exercises

1. Would the operations of the skip list be faster or slower if we added a new level one-third of the time, rather than one-half of the time? Would you expect there to be more or fewer levels? What about if the probability were higher, say two-thirds of the time? Design an experiment to discover the effect of these changes.

2. Imagine you implemented the naïve set algorithms described in Lesson 24, but used a skip list rather than a vector. What would the resulting execution times be?

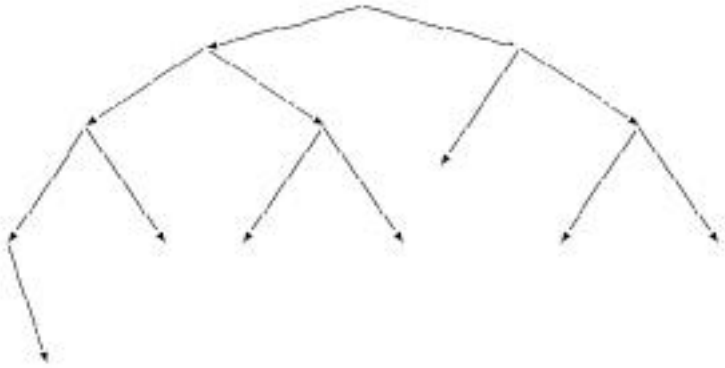
3. Prove that a complete binary tree of height n will have 2^n leaves. (Easy to prove by induction).

4. Prove that the number of nodes in a complete binary tree of height n is $2^{n+1} - 1$.

5. Prove that a binary tree containing n nodes must have at least one path from root to leaf of length floor(log n).

6. Prove that in a complete binary tree containing n nodes, the longest path from root to leaf traverses no more than ceil(log n) nodes.

7. So how close to being well balanced is an AVL tree? Recall that the definition asserts the difference in height between any two children is no more than one. This property is termed a *height-balanced tree*. Height balance assures that locally, at each node, the balance is roughly maintained, although globally over the entire tree differences in path lengths can be somewhat larger. The following shows an example height-balanced binary tree.



A complete binary tree is also height balanced. Thus, the largest number of nodes in a balanced binary tree of height h is 2^{h+1}-1. An interesting question is to discover the *smallest* number of nodes in a height-balanced binary tree. For height zero there is only one tree. For height 1 there are three trees, the smallest of which has two nodes. In general, for a tree of height h the smallest number of nodes is found by connecting the smallest tree of height h-1 and h-2.

If we let M_h represent the function yielding the minimum number of nodes for a height balanced tree of height h, we obtain the following equations:

$M_0 = 1$
$M_1 = 2$
$M_{h+1} = M_{h-1} + M_h + 1$

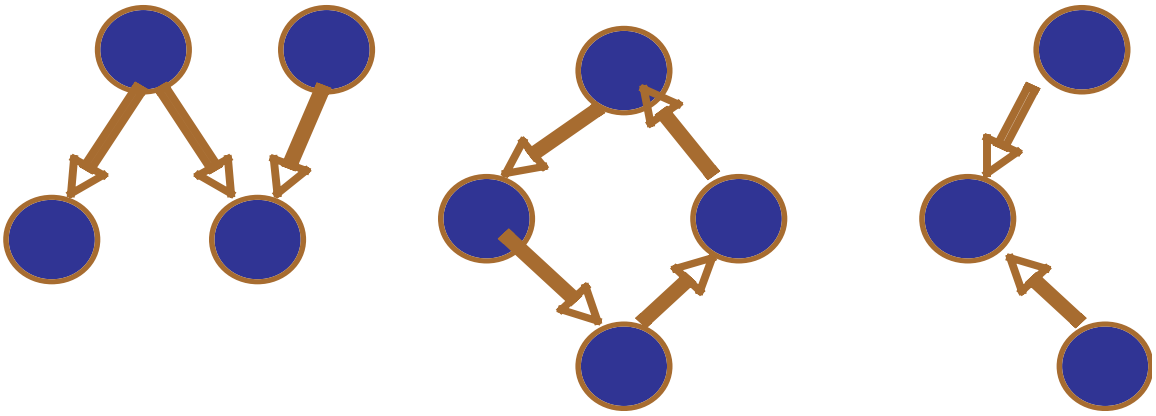
These equations are very similar to the famous *Fibonacci numbers* defined by the formula $f_0 = 0$, $f_1 = 1$, $f_{n+1} = f_{n-1} + f_n$. An induction argument can be used to show that $M_h = f_{h+3} - 1$. It is easy to show using induction that we can bound the Fibonacci numbers by 2^n. In fact, it is possible to establish an even tighter bounding value. Although the details need not concern us here, the Fibonacci numbers have a closed form solution; that is, a solution defined without using recursion. The value F_h is approximately $\frac{\phi^i}{\sqrt{5}}$, where ϕ is the golden mean value $\frac{1+\sqrt{5}}{2}$, or approximately *1.618*. Using this information, we can show that the function M_h also has an approximate closed form solution:

$$M_h = \frac{\phi^{h+3}}{\sqrt{5}} - 1$$

By taking the logarithm of both sides and discarding all but the most significant terms we obtain the result that h is approximately 1.44 log M_h. This tells us that the longest path in a height-balanced binary tree with n nodes is at worst only 44 percent larger than the log n minimum length. Hence algorithms on height-balanced binary trees that run in time proportional to the length of the path are still O(log n). More importantly, preserving the height balanced property is considerably easier than maintaining a completely balanced tree.

Because AVL trees are fast on all three bag operations they are a good general purpose data structure useful in many different types of applications. The Java standard library has a collection type, TreeSet, that is based on a data type very similar to the AVL trees described here.

8. Explain why the following are not legal trees.



Programming Projects

1. The bottom row of a skip list is a simple ordered list. You already know from Lesson 36 how to make an iterator for ordered lists. Using this technique, create an iterator for the skip list abstraction. How do you handle the remove operation?

2. The smallest value in a skip list will always be the first element. Assuming you have implemented the iterator described in question 6, you have a way of accessing this value. Show how to use a skip list to implement a priority queue. A priority queue, you will recall, provides fast access to the smallest element in a collection. What will be the algorithmic execution time for each of the priority queue operations?

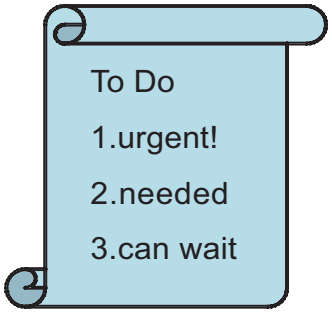
```
void skipPQaddElement (struct skipList *, TYPE newElement);
TYPE skipPQsmallestElement (struct skipList *);
void skipPQremoveSmallest (struct skipList *);
```

On the Web

The wikipedia has a good explanation of various efficient data structures. These include entries on skip lists, AVL trees and other self-balancing binary search trees. Two forms of tree deserve special note. Red/Black trees are more complex than AVL trees, but require only one bit of additional information (whether a node is red or black), and are in practice slightly faster than AVL trees. For this reason they are used in many data structure libraries. B-trees (such as 2-3 B trees) store a larger number of values in a block, similar to the dynamic array block. They are frequently used when the actual data is stored externally, rather than in memory. When a value is accessed the entire block is moved back into memory.

Chapter 11: Priority Queues and Heaps

In this chapter we examine yet another variation on the simple Bag data structure. A *priority queue* maintains values in order of importance. A metaphor for a priority queue is a to-do list of tasks waiting to be performed, or a list of patients waiting for an operating room in a hospital. The key feature is that you want to be able to quickly find the most important item, the value with highest priority.



Like a bag, you can add new elements into the priority queue. However, the only element that can be accessed or removed is the one value with highest priority. In this sense the container is like the stack or queue, where it was only the element at the top or the front of the collection that could be removed.

The Priority Queue ADT specification

The traditional definition of the Priority Queue abstraction includes the following operations:

add (newelement)	Add a new value to queue
first ()	Return first element in queue
removeFirst ()	Remove first element in queue
isEmpty()	Return true if collection is empty

Normally priority is defined by the user, who provides a comparison function that can be applied to any two elements. The element with the largest (or sometimes, the smallest) value will the deemed the element with highest priority.

A priority queue is not, in the technical sense, a true queue as described in Chapter 7. To be a queue, elements would need to satisfy the FIFO property. This is clearly not the case for the priority queue. However, the name is now firmly attached to this abstraction, so it is unlikely to change.

The following table shows priority queue operation names in the C++ STL and in the Java class PriorityQueue.

Operation	C++ STL class priorityQueue<T>	Java Container class priorityQueue
Add value	push(E)	add(E)
First Value	top()	peek()
Remove First Value	pop()	poll()
Size, or test size	empty()	size() = = 0

In C++ it is the element with the largest value, as defined by the user provided comparison function, that is deemed to be the first value. In the Java library, on the other hand, it is the element with the smallest value. However, since the user provides the comparison function, it is easy to invert the sense of any test. We will use the smallest value as our first element.

Applications of Priority Queues

The most common use of priority queues is in simulation. A simulation of a hospital waiting room, for example, might prioritize patients waiting based on the severity of their need.

A common form of simulation is termed a "discrete, event-driven simulation". In this application the simulation proceeds by a series of "events". An event is simply a representation of an action that occurs at a given time. The priority queue maintains a collection of events, and the event with highest priority will be the event with lowest time; that is, the event that will occur next.

For example, suppose that you are simulating patrons arriving at a small resturant. There are three main types of event of interest to the simulation, namely patrons arriving (the arrival event), patrons ordering (the order event) and patrons leaving (the leave event). An event might be represented by a structure, such as the following:

```
struct eventStruct {
  int time;
  int groupsize;
  enum {arrival, order, leave} eventType;
};
```

To initialize the simulation you randomly generate a number of arrival events, for groups of various sizes, and place them into the queue. The execution of the simulation is then described by the following loop:

```
while the event queue is not empty
  select and remove the next event
  do the event, which may generate new events
```

To "do" the event means to act as if the event has occurred. For example, to "do" an arrival event the patrons walk into the resturant. If there is a free table, they are seated and an subsequent "order" event is added to the queue. Otherwise, if there is not a free table, the patrons either leave, or remain in a queue of waiting patrons. An "order" event produces a subsequent "leave" event. When a "leave" event occurs, the newly emptied table is then occupied by any patrons waiting for a table, otherwise it remains empty until the next arrival event.

Many other types of simulations can be described in a similar fashion.

Priority Queue Implementation Techniques

We will explore three different queue implementation techniques, two of which are developed in the worksheets. The first is to examine how any variety of ordered bag (such as the SortedBag, SkipList, or AVLtree) can be used to implement the priority queue. The second approach introduces a new type of binary tree, termed the *heap*. The classic heap (known simply as the heap) provides a very memory efficient representation for a priority queue. Our third technique, the *skew heap*, uses an interesting variation on the heap technique.

> **A note regarding the name heap.**
>
> The term *heap* is used for two very different concepts in computer science. The heap data structure is an abstract data type used to implement priority queues. The terms *heap*, *heap memory*. *heap allocation*, and so on are used to describe memory that is allocated directly by the user, using the **malloc** function. You should not confuse the two uses of the same term.

Building a Priority Queue using an Ordered Bag

In earlier chapters you have encountered a number of different bag implementation techniques in which the underlying collection was maintained in sequence. Examples included the sorted dynamic array, the skip list, and the AVL tree. In each of these containers the smallest element is always the first value. While the bag does not support direct access to the first element (as does, say, the queue), we can nevertheless obtain access to this value by means of an iterator. This makes it very easy to implement a priority queue using an ordered bag as a storage mechanism for the underlying data values, as the following:

add(E). Simply add the value to the collection using the existing insertion functions.

first(). Construct an iterator, and return the first value produced by the iterator.

removeFirst(). Use the existing remove operation to delete the element returned by first().

isEmpty(). Use the existing size operation for the bag, and return true if the size is zero.

We have seen three ordered collections, the sorted array, the skip list, and the AVL tree. Based on your knowledge of the algorithmic execution speeds for operations in those data structures, fill in the following table with the execution times for the bag heap constructed in a fashion described above.

Operation	SortedArray	SkipList	AVLtree
Add(newElement)			
First()			

removeFirst()			

A note on Encapsulation

There are two choices in developing a new container such as the one described above. One choice is to simply add new functions, or extend the interface, for an existing data structure. Sometimes these can make use of functionality already needed for another purpose. The balanced binary tree, for example, already needed the ability to find the leftmost child in a tree, in order to implement the remove operation. It is easy to use this function to return the first element in the three. However, this opens the possibility that the container could be used with operations that are not part of the priority queue interface.

An alternative would have been to create a new data structure, and encapsulate the underlying container behind a structure barrier, using something like the following:

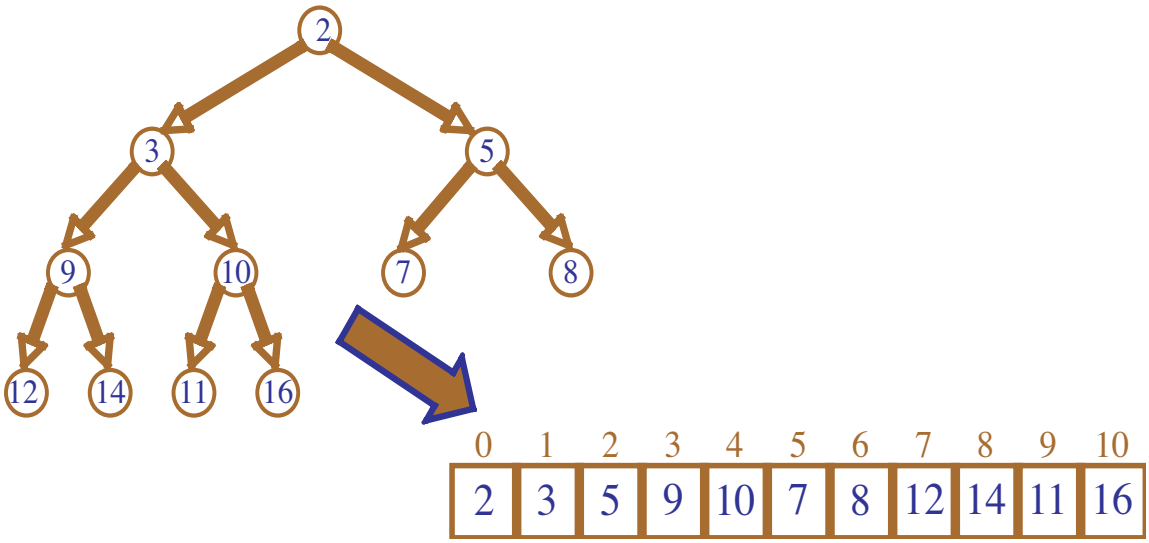
```
struct SortedHeap {
  struct AVLtree data;
};
```

We would then need to write routines to initialize this new structure, rather than relying on the existing routines to initialize an AVL tree. At the cost of an additional layer of indirection we can then more easily guarantee that the only operations performed will be those defined by the interface.

There are advantages and disadvantages to both. The bottom line is that you, as a programmer, should be aware of both approaches to a problem, and more importantly be aware of the implications of whatever design choice you make.

Building a Priority Queue using a Heap

In a worksheet you will explore two alternative implementation techniques for priority queues that are based around the idea of storing values in a type of representation termed a *heap*. A heap is a binary tree that also maintains the property that the value stored at every node is less than or equal to the values stored at either of its child nodes. This is termed the *heap order property.* The classic heap structure (known just as the heap) adds the additional requirement that the binary tree is complete. That is, the tree if full except for the bottom row, which is filled from left to right. The following is an example of such a heap:

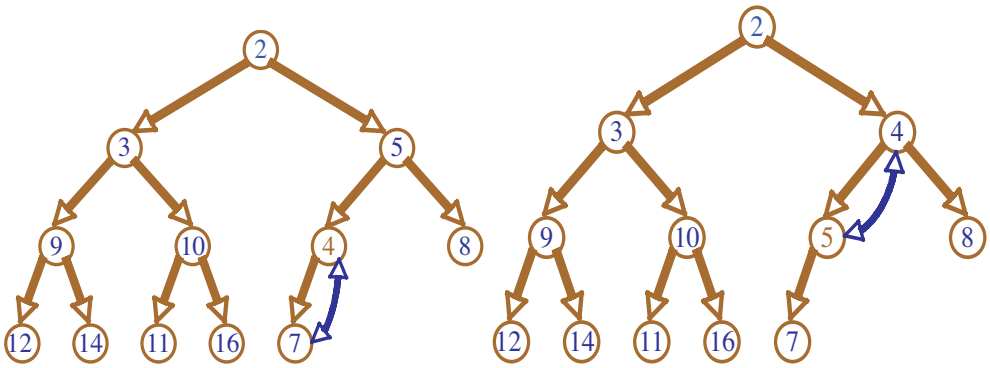


Notice that a heap is partially ordered, but not completely. In particular, the smallest element is always at the root. Although we will continue to think of the heap as a tree, we will make use of the fact that a complete binary tree can be very efficiently represented as an array. To root of the tree will be stored as the first element in the array. The children of node i are found at positions 2i+1 and 2i+2, the parent at (i-1)/2. You should examine the tree above, and verify that the transformation given will always lead you to the children of any node. To reverse the process, to move from a node back to the parent, simply subtract 1 and divide by 2. You should also verify that this process works as you would expect.

We will construct our heap by defining functions that will use an underlying dynamic array as the data container. This means that users will first need to create a new dynamic array before they can use our heap functions:

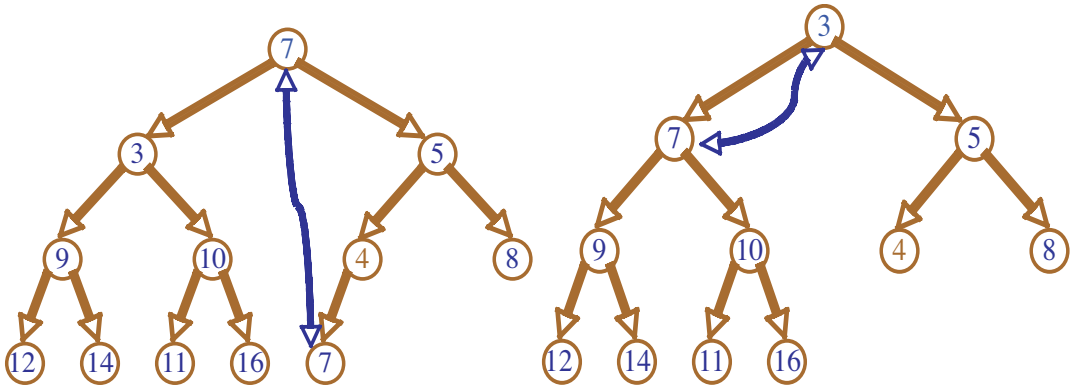
```
struct dynArray heap;  /* create a new dynamic array */
initDynArray (&heap, 10);  /* initialize the array to 10 elements */
…
```

To insert a new value into a heap the value is first added to the end. (This operation has actually already been written in the function addDynArray). Adding an element to the end preserves the complete binary tree property, but not the heap ordering. To fix the ordering, the new value is *percolated up* into position. It is compared to its parent node. If smaller, the node and the parent are exchanged. This continues until the root is reached, or the new value finds its correct position. Because this process follows a path in a complete binary tree, it is O(log n). The following illustrates adding the value 4 into a heap, then percolating it up until it reaches its final position. When the value 4 is compared to the 2, the parent node containing the 2 is smaller, and the percolation process halts.



Because the process of percolating up traverses a complete binary tree from leaf to root, it is O(log n), where n represents the number of nodes in the tree.

Percolating up takes care of insertion into the heap. What about the other operations? The smallest value is always found at the root. This makes accessing the smallest element easy. But what about the removal operation? When the root node is removed it leaves a "hole." Filling this hole with the last element in the heap restores the complete binary tree property, but not the heap order property. To restore the heap order the new value must *percolate down* into position.

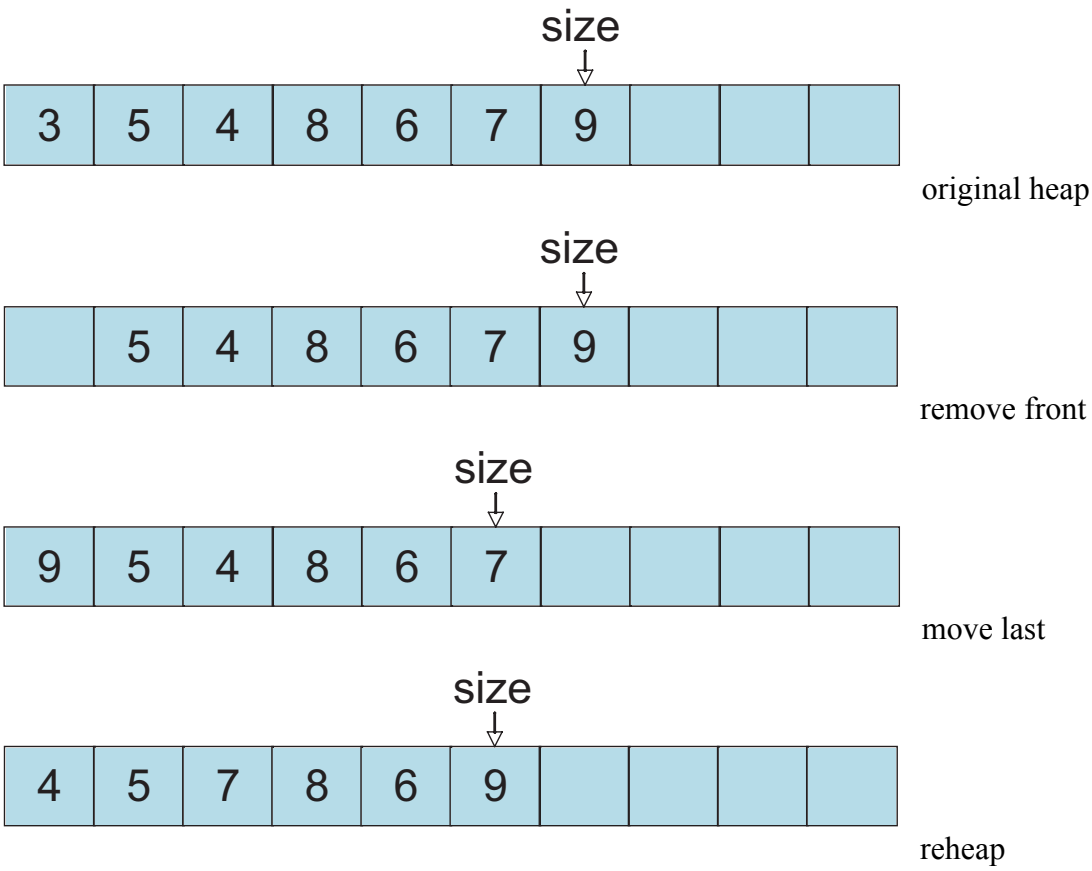


To percolate down a node is compared to its children. If there are no children, the process halts. Otherwise, the value of the node is compared to the value of the smallest child. If the node is larger, it is swapped with the smallest child, and the process continues with the child. Again, the process is traversing a path from root to leaf in a complete binary tree. It is therefore O(log n).

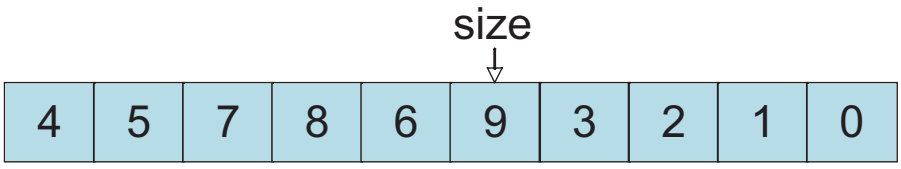
In worksheet 33 you will complete the implementation of the priority queue constructed using a heap by providing the functions to percolate values into position, but up and down.

Heap Sort

When a value is removed from a heap the size of the heap is reduced. If the heap is being represented in an array, this means that the bottom-most element of the array is no longer being used. (Using the terminology of the dynamic array we examined earlier, the size of the collection is smaller, but the capacity remains unchanged).[In our class slides, this size is equivalent to our Last]

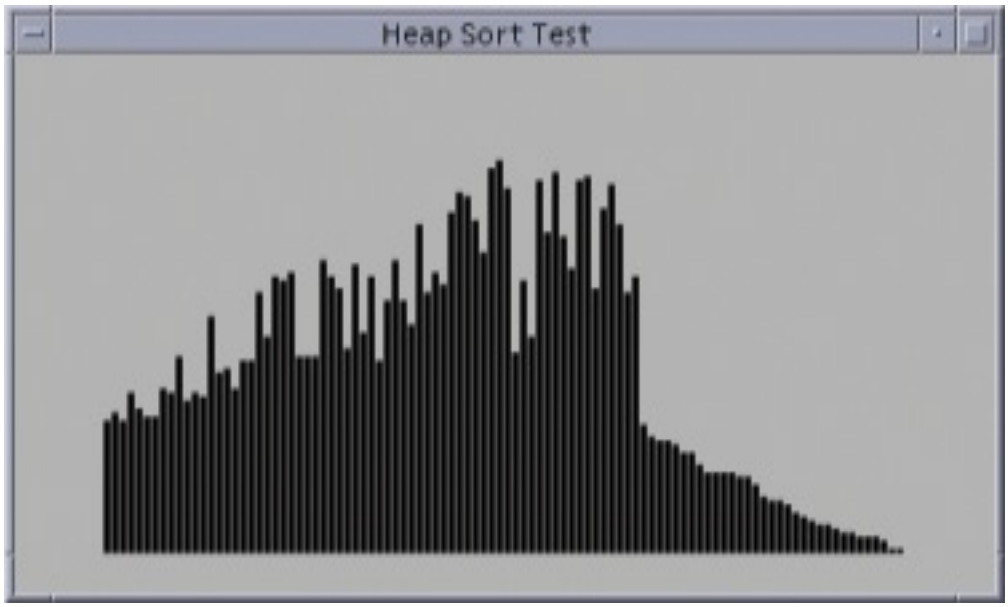


What if we were to store the removed values in the now unused section of the array? In fact, since the last element in the array is always moved into the hole made by the removal of the root, it is a trivial matter to simply swap this value with the root.



Suppose we repeat this process, always swapping the root with the currently last element, then reheaping by percolating the new root down into position. The result would be a sorting algorithm, termed *heap sort*. The following is a snapshot illustrating heap sort in the middle of execution. Notice that the smallest elements have been moved to the right. The current size of the dynamic array is indicated by the sharp drop in values. The

elements to the left of this point are organized in a heap. Notice that the heap is not completely ordered, but has a tendency towards being ordered.



To determine the algorithmic execution time for this algorithm, recall that adjustHeap requires O(log n) steps. There are n executions of adjustHeap to produce the initial heap. Afterwards, there are n further executions to reheap values during the process of sorting. Altogether the running time is O(n log n). This matches that of merge sort, quick sort, and tree sort. Better yet, heap sort requires no additional storage.

Question: Simulate execution of the Heap sort algorithm on the following values:

9 3 2 4 5 7 8 6 1 0

First make the values into a heap (the graphical representation is probably easier to work with than the vector form). Then repeatedly remove the smallest value, and rebuild the heap.

Skew Heaps

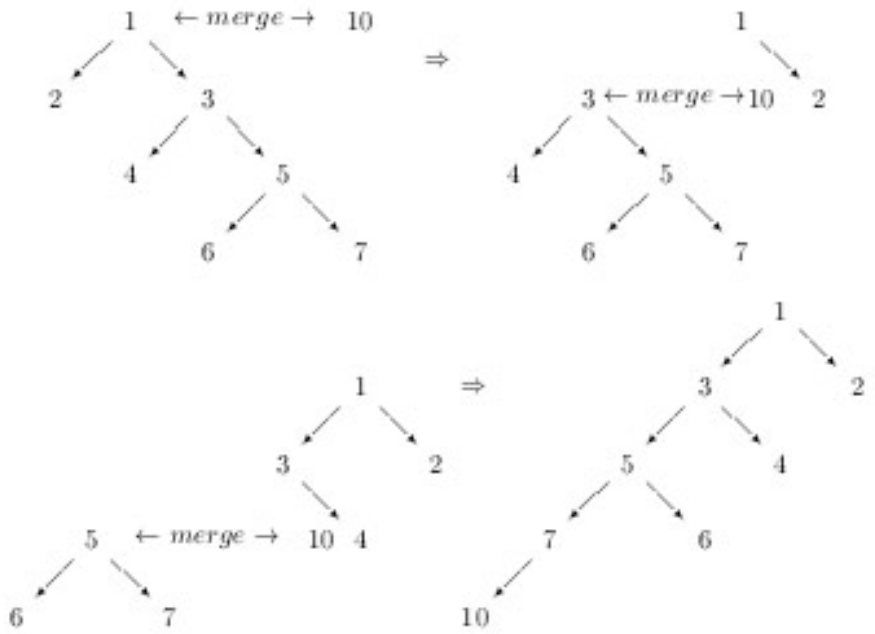
In a heap the relative order of the left and right children is unimportant. The only property that must be preserved is that each node is smaller than either of its children. However, the heap order property does not insist that, for example, a left child is smaller than a right child. The *skew heap* builds on this flexibility, and results in a very different organization from the traditional heap. The skew heap makes two observations. First, left and right children can always be exchanged with each other, since their order is unimportant. Second, both insertions and removals from a heap can be implemented as special cases of a more general task, which is to merge two heaps into one.

It is easy to see how the remove is similar to a merge. When the smallest (that is, root) element is removed, you are left with two trees, namely the left and right child trees. To build a new heap you can simply merge the two.

To view addition as a merge, consider the existing heap as one argument, and a tree with only the single new node as the second. Merge the two to produce the new heap.

Unlike the classic heap, a skew heap does not insist that the binary tree is complete. Furthermore, a skew heap makes no attempt to guarantee the balance of its internal tree. Potentially, this means that a tree could become thin and unbalanced. But this is where the first observation is used. During the merge process the left and right children are systematically swapped. The result is that a thin and unbalanced tree cannot remain so. It can be shown (although the details are not presented here) that amortized over time, each operation in a skew heap is no worst than O(log n).

The following illustrates the addition of the value 10 to an existing tree. Notice how a tree with a long right path becomes a tree with a long left path.



The merge algorithm for a skew heap can be described as follows:

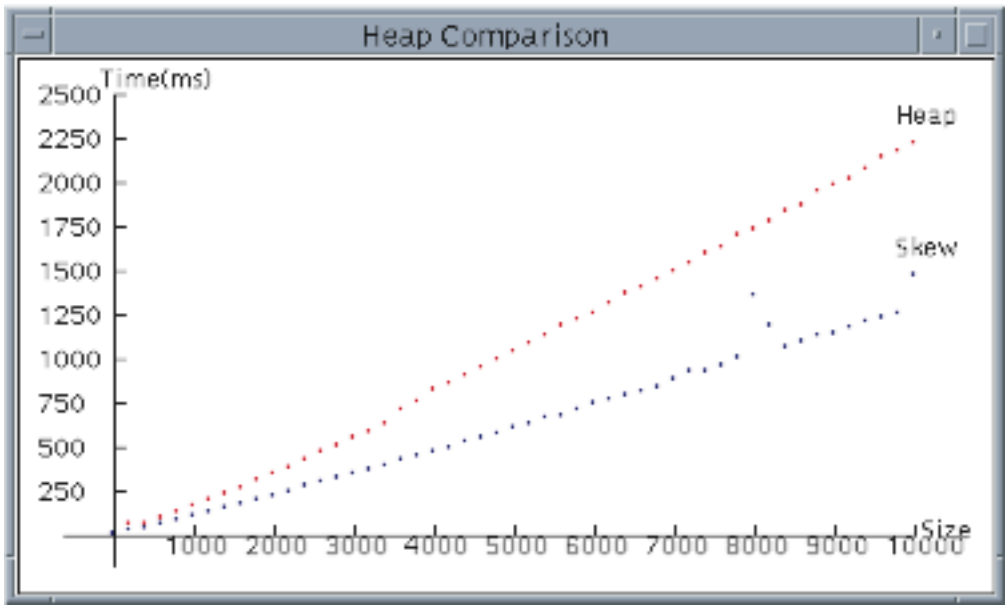
```
Node merge (Node left, Node right)
  if (left is null) return right
  if (right is null) return left
  if (left child value < right child value) {
    Node temp = left.left;
    left.left = merge(left.right, right)
    left.right = temp
    return left;
```

```
} else {
  Node temp = right.right
  right.right = merge(right.left, left)
  right.left = temp
  return right
}
```

In worksheet 35 you will complete the implementation of the SkewHeap based on these ideas.

Like the self organizing list described in Chapter 8, a skew heap is an example of a data structure that tries to optimize future performance based on past operations. It makes no guaranteed that poorly balanced trees cannot arise, but if they do they are quickly taken apart so that they cannot have a lasting impact on execution time.

To measure the relative performance of the Heap and SkewHeap abstractions, an experiment was conducted in which n random integers between 0 and 100 were inserted into a heap and then removed. A plot of the execution times for various values of n was obtained as follows. The result indicates that even through the SkewHeap is performing many more memory allocations than the Heap, the overall execution time is still faster.



Short Exercises

1. Given an example of a priority queue that occurs in a non-computer science situation.

2. Where is the smallest value found in a heap?

3. Where is the 2nd smallest element in a heap? The third smallest element?

4. What is the height of a heap that contains 10 elements?

5. If you interpret a sorted array as a heap, is it guaranteed to have the heap order property?

6. Could you implement the tree sort algorithm using a heap? Recall that the tree sort algorithm simply copies values from a vector into a container (that is, the heap), the repeatedly removes the smallest element and copies it back to the vector. What is the big-oh execution time for this algorithm?

7. Suppose you wanted to test the Heap data structure. What would be some good boundary value test cases?

8. Program a test driver for the Heap data type, and execute the operations using the test cases you identified in the previous question.

9. An operation we have not discussed is the ability to change the value of an item already in the heap. Describe an algorithm that could be used for this purpose. What is the complexity of your algorithm?

Analysis Exercises

1. Show that unless other information is maintained, finding the *maximum* value in a heap must be an O(n) operation. Hint: How many elements in the heap could potentially be the maximum?

2. Imagine a heap that contains 2^n values. This will naturally represent a complete binary tree. Notice that the heap property is retained if the left and right subtrees of any node are exchanged. How many different equivalent heaps can be produced using only such swappings?
3.

Self Study Questions

1. What are the abstract operations that characterize a priority queue?

2. Give an example of a priority queue that occurs in a non-computer science situation.

3. Describe how a priority queue is similar to a bag, and how it is different. Describe how a priority queue is similar to a queue, and how it is different.

4. Why is a priority queue not a true queue in the sense described in Chapter 7?

5. What is discrete event driven simulation? How is it related to priority queues?

6. What property characterizes the nodes in a heap?

7. When a binary tree, such as a heap, is represented as a dynamic array, where are the children of node i? What is the index of the parent of node i?

8. When a heap is represented in an array, why is it important that the tree being represented in complete? What happens if this condition is not satisfied?

9. Illustrate the process of percolating a value down into position. Explain why this operation is O(log n).

10. Illustrate the process of percolating a value up into position. Explain why this operation is O(log n).

11. Place the following values, in the order shown, into an initially empty heap, and show the resulting structure: 5 6 4 2 9 7 8 1 3

12. How is a skew heap different from a heap? What property of the heap data type does this data structure exploit?

13. Perform the same insertions from question 10 into an initially empty skew heap, and show the resulting structure.

Chapter Summary

Key Concepts

- Priority Queue
- Heap
- Heap order property
- Heap sort
- Skew heap

A *priority queue* is not a true queue at all, but is a data structure designed to permit rapid access and removal of the smallest element in a collection. One way to build a priority queue is to use an ordered collection. However, they can also be constructed using an efficient array representation and an idea termed the heap. A heap is a binary tree that supports the heap order property. The heap order property says that the value stored at every node is smaller than the value stored at either of its child node. The classic heap stores elements in a complete binary tree. Because the tree is complete, it can be represented in an array form without any holes appearing in the collection. In addition to providing the basis for implementing a priority queue, the heap structure forms the basis of a very efficient sorting algorithm.

A skew heap is a form of heap that does not have the fixed size characteristic of the vector heap. The skew heap data structure is interesting in that it can potentially have a very poor worst case performance. However, it can be shown that the worst case performance cannot be maintained, and following any occurrence the next several

operations of insertion or removal must be very rapid. Thus, when measured over several operations the performance of a skew heap is very impressive.

Programming Exercises

1. Complete the implementation of the priority queue using a sorted dynamic array as the underlying container. This technique allows you to access both the smallest and the largest element with equal ease. What is the algorithmic execution time for operations with this representation?

2. Consider using a balanced binary tree, such as an AVL tree, for an implementation technique. Recall that as part of the process of removing an element from an AVL tree you needed the ability to find the leftmost child of the right subtree. Show how using the ability to identify the leftmost child can be used to find the smallest element in the tree. Write priority queue operations based on this observation.

3. Complete the implementation of the resturant simulation described in this chapter. Initialize your simulation with a number of arrival events selected at random points over the period of an hour. Assume that patrons take between five and ten minuets (selected randomly) between the time they are seated and the time they order. Assume that patrons take between 30 to 50 minuits (again, selected randomly) to eat. Those patrons who are not able to seat wait on a queue for a table to become available. Print out a message every time an event occurs indicating the type of the event, and the time for the new events generated by the current event. Keep track of the number of patrons serviced in the period of two hours.

4. Program an implementation of an airport. The airport has two runways. Planes arrive and request permission to land and, independently, planes on the ground request permission to talk off.

5. Program a discrete event driven simulation of a hospital emergency room. Upon arrival, a triage doctor assigns each patient a number based on the severity of his or her harm. The room has a fixed number of beds. As each bed is freed, the next most urgent patient is handled. Patients take varying amounts of time to handle depending upon their condition. Collect statistics on the length of time each patient will have to wait.

6. An alternative approach to the skew heap is a leftist heap. In a leftist heap, the height of the left child is always larger than or equal to the height of the right child. As two children in a heap can always be swapped without violating the heap-order property, this is an easy condition to impose. To construct a leftist heap, nodes are modified so as to remember their height. (We saw how to do this in the AVL tree). As with the skew heap, all operations on a leftist heap are implemented using the single task of merging two heaps. The following steps are used to merge heaps T1 and T2:
    a. If the value in T2 is less than the value in T1, merge in the opposite order

b. The root of T1 is the root of the new heap. The right shild is recursively merged with T2
c. If the resulting height of the right child is larger than the left child, the two children are reversed
d. The height of the result is set to one lrger than the height of the left child.

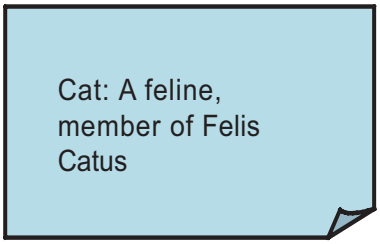
Because the merge is always performed with a right child, which is always has a height smaller than the left child, the result is produced very quickly. Provide an implementation of a heap data structure based on this principle.

On the Web

The wikipedia contains articles explaining the priority queue abstraction, as well as the heap data structure, and the associated heap sort algorithm. The entry for heap contains links to several variations, including the skew heap, as well as others. Wikipedia also contains a good explanation of the concept of Discrete Event Simlation. The on-line *Dictionary of Algorithms and Data Structures* contains explanations of heap, the heapify algorithm, and other topics discussed in this chapter.

Chapter 12: Dictionary (or Map) ADT and Hash Tables

In the containers we have examined up to now, the emphasis has been on the values themselves. A bag, for example, is used to hold a collection of elements. You can add a new value to a bag, test to see whether or not a value is found in the bag, and remove a value from the bag. In a *dictionary*, on the other hand, we separate the data into two parts. Each item stored in a dictionary is represented by a key/value pair. The key is used to access the item. With the key you can access the value, which typically has more information.



The name itself suggests the metaphor used to understand this abstract data type. Think of a dictionary of the English language. Here a word (the key) is matched with a definition (the value). You use the key to find the definition. But, the connection is one way. You typically cannot search a dictionary using a definition to find the matching word.

A keyed container, such as a dictionary, can be contrasted with an indexed container, such as an array. Index values are a form of key; however, they are restricted to being integers and must be drawn from a limited range: usually zero to one less than the number of elements. Because of this restriction the elements in an array can be stored in a contiguous block; and the index can be used as part of the array offset calculation. In an array, the index need not be stored explicitly in the container. In a dictionary, on the other hand, a key can be any type of object. For this reason, the container must maintain both the key and its associated value.

The Dictionary ADT

The abstract data type that corresponds to the dictionary metaphor is known by several names. Other terms for keyed containers include the names *map*, *table*, *search table*, *associative array*, or *hash*. Whatever it is called, the idea is a data structure optimized for a very specific type of search. Elements are placed into the dictionary in key/value pairs. To do a retrieval, the user supplies a key, and the container returns the associated value. Each key identifies one entry; that is, each key is unique. However, nothing prevents two different keys from referencing the same value. The contains test is in the dictionary replaced by a test to see if a given key is legal. Finally, data is removed from a dictionary by specifying the key for the data value to be deleted.

As an ADT, the dictionary is represented by the following operations:

get(key)	Retrieve the value associated with the given key.
put(key, value)	Place the key and value association into the dictionary
containsKey(key)	Return true if key is found in dictionary

removeKey(key)	Remove key from association
keys()	Return iterator for keys in dictionary
size()	Return number of elements in dictionary

The operation we are calling put is sometimes named set, insertAt, or atPut. The get operation is sometimes termed at. In our containers a get with an invalid key will produce an assertion error. In some variations on the container this operation will raise an exception, or return a special value, such as null. We include an iterator for the key set as part of the specification, but will leave the implementation of this feature as an exercise for the reader.

The following illustrates some of the implementations of the dictionary abstraction found in various programming languages.

Operation	C++ Map<keytype, valuetype>	Java HashMap<keytype, valuetype>	C# hashtable
get	Map[key]	Get(key)	Hash[key]
put	Insert(key, value)	Put(key, value)	Add(key, value)
containsKey	Count(key)	containsKey(key)	
removeKey	Erase(key)	Remove(key)	

Applications of the Dictionary data type

Maps or dictionaries are useful in any type of application where further information is associated with a given key. Obvious examples include dictionaries of word/definition pairs, telephone books of name/number pairs, or calendars of date/event-description pairs. Dictionaries are used frequently in analysis of printed text. For example, a concordance examines each word in a text, and constructs a list indicating on which line each word appears.

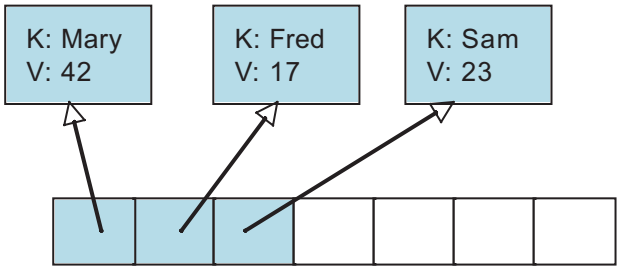
Implementations of the Dictionary ADT

We will describe several different approaches to implementing the dictionary data type. The first has the advantage of being easy to describe, however it is relatively slow. The remaining implementation techniques will introduce a new way of organizing a collection of data values. This technique is termed hashing, and the container built using this technique is called a hash table.

The concept of hashing that we describe here is useful in implementing both Bag and Dictionary type collections. Much of our presentation will use the Bag as an example, as dealing with one value is easier than dealing with two. However, the generalization to a Dictionary rather than a Bag is straightforward, and will be handled in programming projects described at the end of the chapter.

A Dictionary Built on top of a Bag

The basic idea of the first implementation approach is to treat a dictionary as simply a bag of key/value pairs. We will introduce a new name for this pair, calling it an *Association*. An Association is in some ways similar to a link. Instances of the class Association maintain a key and value field.



But we introduce a subtle twist to the definition of the type Association. When one association is compared to another, the result of the comparison is determined solely by the key. If two keys are equal, then the associations are considered equal. If one key is less than another, then the first association is considered to be smaller than the second.

With the assistance of an association, the implementation of the dictionary data type is straightforward. To see if the dictionary contains an element, a new association is created. The underlying bag is then searched. If the search returns true, it means that there is an entry in the bag that matches the key for the collection. Similarly, the remove method first constructs a new association. By invoking the remove method for the underlying bag, any element that matches the new association will be deleted. But we have purposely defined the association so that it tests only the key. Therefore, any element that matches the key will be removed.

This type of Map can be built on top of any of our Bag abstractions, and is explored in worksheet 36.

Hashing Background

We have seen how containers such as the skip list and the AVL tree can reduce the time to perform operations from O(n) to O(log n). But can we do better? Would it be possible to create a container in which the average time to perform an operation was O(1)? The answer is both yes and no.

To illustrate how, consider the following story. Six friends; Alfred, Alessia, Amina, Amy, Andy and Anne, have a club. Amy is in charge of writing a program to do bookkeeping. Dues are paid each time a member attends a meeting, but not all members attend all meetings. To help with the programming Amy uses a six-element array to store the amount each member has paid in dues.

Alfred	$2.65	F = 5 % 6 = 5
Alessia	$6.75	E = 4 % 6 = 4
Amina	$5.50	I = 8 % 6 = 2
Amy	$10.50	Y = 24 % 6 = 0
Andy	$2.25	D = 3 % 6 = 3
Anne	$0.75	N = 13 % 6 = 1

Amy uses an interesting fact. If she selects the third letter of each name, treating the letter as a number from 0 to 25, and then mods (%) the number by 6, each name yields a different number. So in O(1) time Amy can change a

name into an integer index value, then use this value to index into a table. This is faster than an ordered data structure, indeed almost as fast as a subscript calculation.

What Amy has discovered is called a *perfect hash function*. A *hash function* is a function that takes as input an element and returns an integer value. Almost always the index used by a hash algorithm is the remainder after dividing this value by the hash table size. So, for example, Amy's hash function returns values from 0 to 25. She mods (%) by the table size (6) in order to get an index.

The idea of *hashing* can be used to create a variety of different data structures. Of course, Amy's system falls apart when the set of names is different. Suppose Alan wishes to join the club. Amy's calculation for Alan will yield 0, the same value as Amy. Two values that have the same hash are said to have *collided*. The way in which collisions are handed is what separates different hash table techniques.

Almost any process that converts a value into an integer can be used as a hash function. Strings can interpret characters as integers (as in Amy's club), doubles can use a portion of their numeric value, structures can use one or more fields. Hash functions are only required to return a value that is integer, not necessarily positive. So it is common to surround the calculation with abs( ) to ensure a positive value.

Open Address Hashing

There are several ways we can use the idea of hashing to help construct a container abstraction. The first technique you will explore is termed *open-address hashing*. (Curiously, also sometimes called *closed hashing*). We explain this technique by first constructing a Bag, and then using the Bag as the source for a Dictionary, as described in the first section. When open-address hashing is used all elements are stored in a single large table. Positions that are not yet filled are given a **null** value. An eight-element table using Amy's algorithm would look like the following:

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina			Andy	Alessia	Alfred		Aspen

Notice that the table size is different, and so the index values are also different. The letters at the top show characters that hash into the indicated locations. If Anne now joins the club, we will find that the hash value (namely, 5) is the same as for Alfred. So to find a location to store the value Anne we *probe* for the next free location. This means to simply move forward, position by position, until an empty location is found. In this example the next free location is at position 6.

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina			Andy	Alessia	Alfred	Anne	Aspen

No suppose Agnes wishes to join the club. Her hash value, 6, is already filled. The probe moves forward to the next position, and when the end of the array is reached it continues with the first element, in a fashion similar to the dynamic array deque you examined in Chapter 7. Eventually the probing halts, finding position 1:

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina	Agnes		Andy	Alessia	Alfred	Anne	Aspen

Finally, suppose Alan wishes to join the club. He finds that his hash location, 0, is filled by Amina. The next free location is not until position 2:

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina	Agnes	Alan	Andy	Alessia	Alfred	Anne	Aspen

We now have as many elements as can fit into this table. The ratio of the number of elements to the table size is known as the *load factor*, written λ. For open address hashing the load factor is never larger than 1. Just as a dynamic array was doubled in size when necessary, a common solution to a full hash table is to move all values into a new and larger table when the load factor becomes larger than some threshold, such as 0.75. To do so a new table is created, and every entry in the old table is rehashed, this time dividing by the new table size to find the index to place into the new table.

To see if a value is contained in a hash table the test value is first hashed. But just because the value is not found at the given location doesn't mean that it is not in the table. Think about searching the table above for the value Alan, for example. Instead of immediately halting, an unsuccessful test must continue to probe, moving forward until either the value is found or an empty location is encountered.

Removing an element from an open hash table is problematic. We cannot simply replace the location with a null entry, as this might interfere with subsequent search operations. Imagine that we replaced Agnes with a null value in the table given above, and then once more performed a search for Alan. What would happen?

One solution to this problem is to not allow removals. This is the technique we will use. The second solution is to create a special type of marker termed a *tombstone*. A tombstone replaces a deleted value, can be replaced by another newly inserted value, but does not halt the search.

λ	(1/(1-λ))
0.25	1.3
0.5	2.0
0.6	2.5
0.75	4.0
0.85	6.6
0.95	19.0

How fast are hash table operations? The analysis depends upon several factors. We assume that the time it takes to compute the hash value itself is constant. But what about distribution of the integers returned by the hash function? It would be perfectly legal for a hash function to always return the value zero – legal, but not very useful.

The best case occurs when the hash function returns values that are uniformly distributed among all possible index values; that is, for any input value each index is equally likely. In this situation one can show that the number of elements that will be examined in performing an addition, removal or test will be roughly $1/(1 - \lambda)$. For a small load factor this is acceptable, but degrades quickly as the load factor increases. This is why hash tables typically increase the size of the table if the load factor becomes too large.

Worksheet 37 explores the implementation of a bag using open hash table techniques.

Caching

Indexing into a hash table is extremely fast, even faster than searching a skip list or an AVL tree. There are many different ways to exploit this speed. A *cache* is a data structure that uses two levels of storage. One level is simply an ordinary collection class, such as a bag dictionary. The second level is a hash table, used for its speed. The cache makes no attempt to handle the problem of collisions within the hash table. When a search request is received, the cache will examine the hash table. If the value is found in the cache, it is simply returned. If it is not found, then the original data structure is examined. If it is found there, the retrieved item *replaces* the value stored in the cache. Because the new item is now in the cache, a subsequent search for the same value will be very fast.

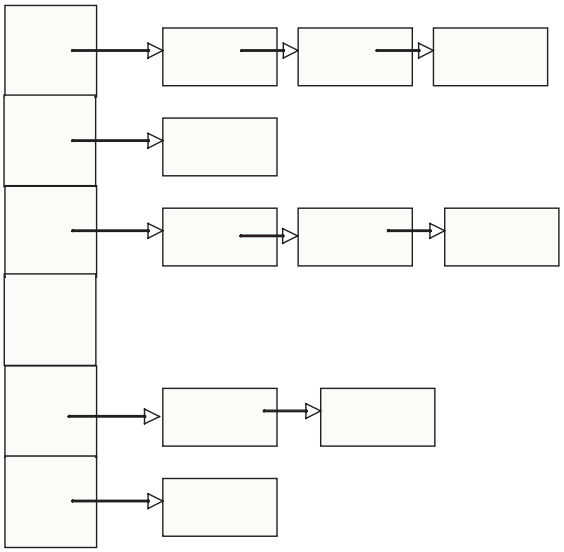
The concept of a cache is generally associated with computer memory, where the underlying container represents paged memory from a relatively slow device (such as a disk), and the cache holds memory pages that have been recently accessed. However the concept of a two-level memory system is applicable in any situation where searching is a more common operation than addition or removal, and where a search for one value means that it is very likely the value will again be requested in the near future. For example, a cache is used in the interpreter for the language Smalltalk to match a message (function applied to an object) with a method (code to be executed). The cache stores the name and class for the most recent message. If a message is sent to the same class of object, the code is found very quickly. If not, then a much slower search of the class hierarchy is performed. However, after this slower search the code is placed into the cache. A subsequent execution of the same message (which, it turns out, is very likely) will then be much faster.

Like the self-organizing linked list (Chapter 8) and the skew heap (worksheet 35), a cache is a self-organizing data structure. That is, the cache tries to improve future performance based on current behavior.

Hash Table with Buckets

In earlier sections you learned about the concept of hashing, and how it was used in an open address hash table. An entirely different approach to dealing with collisions is the idea of hash tables using buckets.

A hash table that uses buckets is really a combination of an array and a linked list. Each element in the array (the hash table) is a header for a linked list. All elements that hash into the same location will be stored in the list. For the Bag type abstraction the link stores only a value and a pointer to the next link. For a dictionary type abstraction, such as we will construct, the link stores the key, the value associated with the key, and the pointer to the next link.



Each operation on the hash table divides into two steps. First, the element is hashed and the remainder taken after dividing by the table size. This yields a table index. Next, linked list indicated by the table index is examined. The algorithms for the latter are very similar to those used in the linked list.

As with open address hash tables, the load factor (λ) is defined as the number of elements divided by the table size. In this structure the load factor can be larger than one, and represents the average number of elements stored in each list, assuming that the hash function distributes elements uniformly over all positions. Since the running time of the contains test and removal is proportional to the length of the list, they are O(λ). Therefore the execution time for hash tables is fast only if the load factor remains small. A typical technique is to resize the table (doubling the size, as with the vector and the open address hash table) if the load factor becomes larger than 10.

Hash tables with buckets are explored in worksheet 39.

Bit Set Spell Checker

In Chapter 8 you learned about the bit set abstraction. Also in that chapter you read how a set can be used as part of a spell checker. A completely different technique for creating a spelling checker is based on using a BitSet in the fashion of a hash table. Begin with a BitSet large enough to hold t elements. We first take the dictionary of correctly spelled

words, hash each word to yield a value h, then set the position h % t. After processing all the words we will have a large hash table of zero/one values. Now to test any particular word we perform the same process; hash the word, take the remainder after dividing by the table size, and test the entry. If the value in the bit set is zero, then the word is misspelled. Unfortunately, if the value in the table is 1, it may still be misspelled since two words can easily hash into the same location.

To correct this problem, we can enlarge our bit set, and try using more than one hash function. An easy way to do this is to select a table size t, and select some number of prime numbers, for example five, that are larger than t. Call these p_1, p_2, p_3, p_4 and p_5. Now rather than just using the hash value of the word as our starting point, we first take the remainder of the hash value when divided by each of the prime numbers, yielding five different values. Then we map each of these into the table. Thus each word may set five different bits. This following illustrates this with a table of size 21, using the values 113, 181, 211, 229 and 283 as our prime numbers:

Word	Hashcode	H%113	H%181	H%211	H%229	H%283
This	6022124	15(15)	73(10)	184(16)	111(6)	167(20)
Text	6018573	80(17)	142(16)	9(9)	224(14)	12(12)
Example	1359142016	51(9)	165(18)	75(12)	223(13)	273(0)

1 0 0 0 0 0 1 0 0 1 1 0 1 1 1 1 1 1 1 0 1

The key assumption is that different words may map into the same locations for one or two positions, but not for all five. Thus, we reduce the chances that a misspelled word will score a false positive hit.

Analysis shows that this is a very reliable technique as long as the final bit vector has roughly the same number of zeros as ones. Performance can be improved by using more prime numbers.

Chapter Summary

Key concepts

- Map
- Association
- Hashing
- Hash functions
- collisions
- Hash Tables
- Open address hashing
- Hash tables with Buckets

Whereas bags, stacks, queues and the other data abstractions examined up to this point emphasize the collection of individual values, a dictionary is a data abstraction designed to maintain *pairs* consisting of a key and a value. Entries are placed into the dictionary using both key and value. To recover or delete a value, the user provides only the key.

In this chapter we have explored several implementation techniques that can be used with the dictionary

abstraction. Most importantly, we have introduced the idea of a hash table.

To *hash* a value means simply to apply a function that transforms a possibly non-integer key into an integer value. This simple idea is the basis for a very powerful data structuring technique. If it is possible to discover a function that transforms a set of keys via a one-to-one mapping on to a set of integer index values, then it is possible to construct a vector using non-integer keys. More commonly, several key values will map into the same integer index. Two keys that map into the same value are said to have *collided*.

The problem of collisions can be handled in a number of different ways. We have in this chapter explored three possibilities. Using open-address-hashing we probe, or search, for the next free unused location in a table. Using caching, the hash table holds the most recent value requested for the given index. Collisions are stored in a slower dictionary, which is searched only when necessary. Using hash tables with buckets, collisions are stored in a linked list of associations.

The state of a hash table can be described in part by the load factor, which is the number of elements in a table divided by the size of the hash table. For open address hashing, the load factor will always be less than 1, as there can be no more elements than the table size. For hash tables that use buckets, the load factor can be larger than 1, and can be interpreted as the average number of elements in each bucket.

The process of hashing permits access and testing operations that potentially are the fastest of any data structure we have considered. Unfortunately, this potential depends upon the wise choice of a hash function, and luck with the key set. A good hash function must uniformly distribute key values over each of the different buckets. Discovering a good hash function is often the most difficult part of using the hash table technique.

Self-Study Questions

1. In what ways is a dictionary similar to an array? In what ways are they different?

2. What does it mean to hash a value?

3. What is a hash function?

4. What is a perfect hash function?

5. What is a collision of two values?

6. What does it mean to probe for a free location in an open address hash table?

7. What is the load factor for a hash table?

8. Why do you not want the load factor to become too large?

9. In searching for a good hash function over the set of integer elements, one student thought he could use the following:

```
int hash = (int)Math.sin(value);
```

explain why this was a poor choice.

Short Exercises

1. In the dynamic array implementation of the dictionary, the **put** operation removes any prior association with the given key before insertion a new association. An alternative would be to search the list of associations, and if one is found simply replace the value field. If no association is found, then insert a new association. Write the **put** method using this approach. Which is easier to understand? Which is likely to be faster?

2. When Alan wishes to join the circle of six friends, why can't Amy simply increase the size of the vector to seven?

3. Amy's club has grown, and now includes the following members:

Abel	Abigail	Abraham	Ada
Adam	Adrian	Adrienne	Agnes
Albert	Alex	Alfred	Alice
Amanda	Amy	Andrew	Andy
Angela	Anita	Anne	Antonia
Arnold	Arthur	Audrey	

Find what value would be computed by Amy's hash function for each member of the group.

4. Assume we use Amy's hash function and assign each member to a bucket by simply dividing the hash value by the number of buckets. Determine how many elements would be assigned to each bucket for a hash table of size 5. Do the same for a hash table of size 11.

5. In searching for a good hash function over the set of integer values, one student thought he could use the following:

```
int index = (int) Math.sin(value);
```

What was wrong with this choice?

6. Can you come up with a perfect hash function for the names of the week? The names of the months? The names of the planets?

7. Examine a set of twelve or more telephone numbers, for example the numbers belonging to your friends. Suppose we want to hash into seven different buckets. What would be a good hash function for your set of telephone numbers? Will your function continue to work for new telephone numbers?

8. Experimentally test the birthday paradox. Gather a group of 24 or more people, and see if any two have the same birthday.

9. Most people find the birthday paradox surprising because they confuse it with the following: if there are *n* people in a room, what is the probability that somebody else shares *your* birthday. Ignoring leap years, give the formula for this expression, and evaluate this when *n* is 24.

10. The function containsKey can be used to see if a dictionary contains a given key. How could you determine if a dictionary contains a given value? What is the complexity of your procedure?

11.

Analysis Exercises

1.A variation on a map, termed a multi-map, allows multiple entries to be associated with the same key. Explain how the multimap abstraction complicates the operations of search and removal.

2. (The birthday Paradox) The frequency of collisions when performing hashing is related to a well known mathematical puzzle. How many randomly chosen people need be in a room before it becomes likely that two people will have the same birth date? Most people would guess the answer would be in the hundreds, since there are 365 possible birthdays (excluding leap years). In fact, the answer is only 24 people.

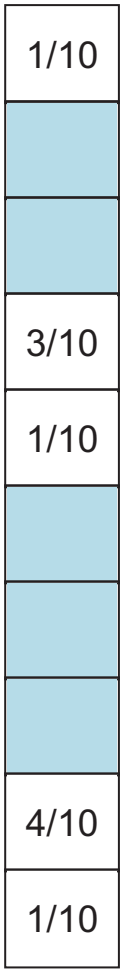
To see why, consider the opposite question. With n randomly chosen people in a room, what is the probability that no two have the same birth date? Imagine we take a calendar, and mark off each individual's birth date in turn. The probability that the second person has a different birthday from the first is 364/365, since there are 364 different possibilities not already marked. Similarly the probability that the third person has a different birthday from the first two is 363/365. Since these two probabilities are independent of each other, the probably that they are *both* true is their product. If we continue in this fashion, if we have n-1 people
all with different birthdays, the probability that individual n has a different birthday is: 364/365 * 363/365 * 363/365 * … * 365-n+1/365. When n >= 24 this expression

becomes less than 0.5. This means that if 24 or more people are gathered in a room the odds are better than even that two individuals have the same birthday.

The implication of the birthday paradox for hashing is to tell us that for any problem of reasonable size we are almost certain to have some collisions. Functions that avoid duplicate indices are surprisingly rare, even with a relatively large table.

2. (Clustering) Imagine that the colored squares in the ten-element table at right indicate values in a hash table that have already been filled. Now assume that the next value will, with equal probability, be any of the ten values. What is the probability that each of the free squares will be filled? Fill in the remaining squares with the correct probability.

Here is a hint: since both positions 1 and 2 are filled, any value that maps into these locations must go into the next free location, which if 3. So the probability that square 3 will be filled is the sum of the probabilities that the next item will map into position 1 (1/10) plus the probability that the next item will map into position 2 (which is 1/10) plus the probability that the next item will map into position 3 (also 1/10). So what is the final probability that position 3 will be filled? Continue with this type of analysis for the rest of the squares.



This phenomenon, where the larger a block of filled cells becomes, the more likely it is to become even larger, is known as clustering. (A similar phenomenon explains why groups of cars on a freeway tend to become larger).

Clustering is just one reason why it is important to keep the load factor of hash tables low.

Simply moving to the next free location is known as linear probing. Many alternatives to linear probing have been studied, however as open address hash tables are relatively rare we will not examine these alternatives here.

Show that probing by any constant amount will not reduce the problem caused by clustering, although it may make it more difficult to observe since clusters are not adjacent. To do this, assume that elements are uniformly inserted into a seven element hash table, with a linear probe value of 3. Having inserted one element, compute the probability that any of the remaining empty slots will be filled. (You can do this by simply testing the values 0 to 6, and observing which locations they will hash into. If only one element will hash into a location, then the probability is 1/7, if two the probability is 2/7, and so on). Explain why the empty locations do not all have equal probability. Place a value in the location with highest probability, and again compute the likelihood that any of the remaining empty slots will be filled. Extrapolate from these observations and explain how clustering will manifest itself in a hash table formed using this technique.

3. You can experimentally explore the effect of the load factor on the efficiency of a hash table. First, using an open address hash table, allow the load factor to reach 0.9 before you reallocate the table. Next, perform the same experiment, but reallocate as soon as the table reaches 0.7. Compare the execution times for various sets of operations. What is the practical effect? You can do the same for the hash table with bucket abstraction, using values larger than 1 for the load factor. For example, compare using the limit of 5 before reallocation to the same table where you allow the lists to grow to length 20 before reallocation.

4. To be truly robust, a hash table cannot perform the conversion of hash value into index using integer operations. To see why, try executing using the standard abs function, and compute and print the absolute value of the smallest integer number. Can you explain why the absolute value of this particular integer is not positive? What happens if you negate the value v? What will happen if you try inserting the value v into the hash table containers you have created in this chapter?

To avoid this problem the conversion of hash value into index must be performed first as a long integer, and then converted back into an integer.

```
Long longhash = abs((Long) hash(v));
int hashindex = (int) longhash;
```

Verify that this calculation results in a positive integer. Explain how it avoids the problem described earlier.

Programming Projects

1. Sometimes the dictionary abstraction is defined using a number of additional functions. Show how each of these can be implemented using a dynamic array. What is the big-oh complexity of each?

    a. int containsValue (struct dyArray *v, ValueType testValue)
    b. void clear (struct dyArray *v) // empty all entries from map
    c. boolean equals (struct dyArray *left, struct dyArray *right)
    d. boolean isEmpty (struct dyArray * v)
    e. void keySet (struct dyArray *v, struct dyArray *e) /* return a set containing all keys */
    f. void values (struct dyArray *v, struct dyArray *e) // return a bag containing all values

1. All our Bag abstractions have the ability to return an iterator. When used in the fashion of Worksheet D1, these iterators will yield a value of type Association. Show how to create a new inner class that takes an iterator as argument, and when requested

returns only the key porition of the association. What should this iterator do with the remove operation?

2. Although we have described hash tables in the chapter on dictionaries, we noted that the technique can equally well be applied to create a Bag like container. Rewrite the Hashtable class as a Hashbag that implements Bag operations, rather than dictionary operations.

3. An iterator for the hash table is more complex than an iterator for a simple bag. This is because the iterator must cycle over two types of collections: the array of buckets, and the linked lists found in each bucket. To do this, the iterator will maintain two values, an integer indicating the current bucket, and a reference to the link representing the current element being returned. Each time hasMore is called the current link is advanced. If there are more elements, the function returns true. If not, then the iterator must advance to the next bucket, and look for a value there. Only when all buckets have been exhausted should the function hasNext return false. Implement an iterator for your hash table implementation.

4. If you have access to a large online dictionary of English words (on unix systems such a dictionary can sometimes be found at /usr/lib/words) perform the following experiment. Add all the words into an open address hash table. What is the resulting size of the hash table? What is the resulting load factor? Change your implementation so that it will keep track of the number of probes used to locate a value. Try searching for a few words. What is the average probe length?

5. Another approach to implementing the Dictionary is to use a pair of parallel dynamic arrays. One array will maintain the keys, and the other one will maintain the values, which are stored in the corresponding positions.

picture

If the key array is sorted, in the fashion of the SortedArrayBag, then binary search can be used to quickly locate an index position in the key array. Develop a data structure based on these ideas.

6. Individuals unfamiliar with a foreign language will often translate a sentence from one language to another using a dictionary and word-for-word substitution. While this does not produce the most elegant translation, it is usually adequate for short sentences, such as ``Where is the train station?'' Write a program that will read from two files. The first file contains a series of word-for-word pairs for a pair of languages. The second file contains text written in the first language. Examine each word in the text, and output the corresponding value of the dictionary entry. Words not in the dictionary can be printed in the output surrounded by square brackets, as are [these] [words].

7. Implement the bit set spell checker as described earlier in this chapter.

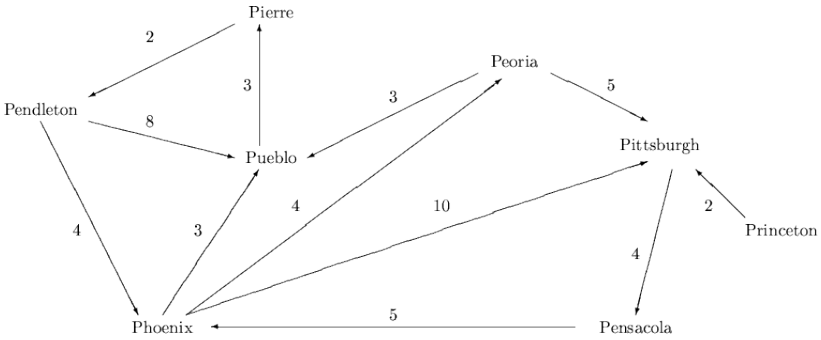
On the Web

The dictionary data structure is described in wikipedia under the entry “Associative Array”. A large section of this entry describes how dictionaries are written in a variety of languages. A separate entry describes the Hash Table data structure. Hash tables with buckets are described in the entry “Coalesced hashing”.

Chapter 13: Graphs and Matricies

Graphs and matrices occur in a large variety of applications in computer science. In this chapter, we provide a brief introduction to these concepts, illustrating the two most commonly used data structures for representing graphs. Example problems are then examined, which illustrate both the benefits and the limitations of each of these representations.

A graph, such as the one shown at right, is composed of *vertices* (or nodes) and *edges* (or arcs). Either may carry additional information. If a graph is being used to model real-world objects, such as a roadmap, the value of a vertex might represent the name of a city, whereas an edge could represent a road from one city to another.



Graphs come in various forms. A graph is said to be *directed* if the edges have a designed beginning and ending point; otherwise, a graph is said to be *undirected*. We will restrict our discussion in this chapter to directed graphs. A second source of variation is the distinction between a *weighted*, or labeled, graph and an *unweighted* graph. In a weighted graph each edge is given a numerical value. For example, the value might represent the distance of a highway connecting two cities, or the cost of the airfare between two airports. An unweighted graph carries no such information. The graph shown above is both directed and weighted.

1	0	0	1	0	0	0	1
0	1	0	1	0	0	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	0	1
1	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	1	1	0
0	0	0	0	1	0	0	1

The two most common representations for a graph are an adjacency matrix and an edge list. An adjacency matrix represents a graph as a two-dimensional matrix. The vertices in the graph are numbered, and used as index values for the matrix. If the cities listed in the graph shown earlier are numbered in alphabetical order (Pendleton is 0, Pensacola is 1, and so on), then the resulting adjacency matrix is shown at left. Here a 1 value in position (i, j) indicates that there is a link between city i and j, and zero value indicates no such link. By convention, a node is always considered to be connected to itself.

The adjacency matrix representation has the disadvantage that it always requires $O(v^2)$ space to store a matrix with v verticies, regardless of the number of edges. An alternative representation, the edge list, stores only the edges and is thus advantageous if the graph is relatively sparse. The basic idea is that each vertex will hold a set of adjacent edges:

Pendleton: {Pueblo, Phoenix}
Pensacola: {Phoenix}

Peoria: {Pueblo, Pittsburgh}
Phoenix: {Pueblo, Peoria, Pittsburgh}
Pierre: {Pendleton}
Pittsburgh: {Pensacola}
Princeton: {Pittsburgh}
Pueblo: {Pierre}

This edge list can easily be stored in a dictionary, where the index is the vertex (say, the city named), and the value is a list of adjacent cities. If the graph is weighted the list is replaced by a second dictionary, again indexed by verticies, which contains the weight for each edge.

Adjacency Matrix Representation

The unweighted adjacency matrix representation of the graph given earlier can be defined in C as shown at right. We have used a symbolic constant to represent the size of the array.

```
# define N 8

int adjacency[N][N] = { { 1, 0, 0, 1, 0, 0, 0, 1},
                        { 0, 1, 0, 1, 0, 0, 0, 0},
                        { 0, 0, 1, 0, 0, 1, 0, 1},
                        { 0, 0, 1, 1, 0, 1, 0, 1},
                        { 1, 0, 0, 0, 1, 0, 0, 0},
                        { 0, 1, 0, 0, 0, 1, 0, 0},
                        { 0, 0, 0, 0, 0, 1, 1, 0},
                        { 0, 0, 0, 0, 1, 0, 0, 1} };
```

There are a number of questions one could ask concerning any particular graph. One of the fundamental problems is the question of *reachability*. That is, what is the set of vertices that can be reached by starting from a particular vertex and moving only along the given arcs in the graph. Conversely, are there any vertices that cannot be reached by traveling in such a fashion.

There are two basic variations on this question. The *single source* question poses a specific initial starting vertex, and requests the set of vertices reachable from this vertex. The *all pairs* question seeks to generate this information simultaneously for all possible vertices. Of course, a solution to the all pairs question will answer the single source question as well. For this reason, we would expect that an algorithm that solves the all pairs problem will require at least as much effort as any algorithm for the single source problem.

```
void warshall (int a[N][N]) {
        int i,j,k;
        for (k = 0; k < N; k++) {
                for (i = 0; i < N; i++)
                        if (a[i][k] != 0)
                          for (j = 0; j < N; j++)
                                a[i][j] |= a[i][k] & a[k][j];
        }
}
```

The algorithm we will describe to illustrate the use of the adjacency matrix representation of graphs is known as *Warshall's algorithm*, after the computer scientist credited with its discovery. The heart of Warshall's algorithm is a triple of loops, which operate much like the loops in the classic algorithm for matrix multiplication. The key idea is that at each

iteration through the outermost loop (index k), we add to the graph any path of length 2 that has node k as its center.

The process of adding information to the graph is performed through a combination of *bitwise* logical operations. The expression `a[i][k] & a[k][j]` is true if there is a path from node i to node k, and if there is a path from node k to node j. By using the bitwise or-assignment operator we add this to position `a[i][j]`. The use of bitwise-or ensures that if there was a 1 bit already in the position, indicating we had already discovered a previous path, then the earlier information will not be erased.

It is easy to see that Warshall's algorithms is $O(n^3)$, where n is the number of nodes in the graph. It is less easy to give a formal proof of correctness for the algorithm. To do so is beyond the purpose of the text here; however, suggestions on how such a proof could be developed are presented in the exercises at the end of the chapter.

The weighted adjacency matrix version of Warshall's algorithm is known as Floyd's algorithm; again, named for the computer scientist credited with the discovery of the algorithm. In the weighted version of the adjacency matrix some value must be identified as representing "not connected". For instance, we could use a large positive number for this purpose. In place of the bitwise-and operation in Warshall's algorithm, an addition is used in Floyd's; and in place of the bitwise-or used to update the path matrix, Floyd's algorithm uses a minimum value calculation. Each time a new path through the matrix is discovered, the cost of the path is compared to that of the previously known best, and if it is shorter than the new cost is recorded.

Edge List Representation

The adjacency matrix representation has the disadvantage that it always requires $O(v^2)$ space to store a matrix with v vertices, regardless of the number of arcs. An alternative representation stores only the arcs, and is thus advantageous if, as is common, the graph is relatively sparse. The basic idea is for each vertex to maintain both a value, and a list of those vertices to which it is connected. Because the function calls necessary to manipulate lists and maps can get in the way of understanding the fundamental algorithms, the techniques here are presented in the form of pseudo-code, rather than working C program.

Reachability for the edge list representation is more commonly expressed as a single-source problem. That is, given a single starting vertex, produce the set of vertices that can be reached starting from the initial location. We can solve this problem using a technique termed *depth-first search*. The basic algorithm can be described as follows:

Depth-first search shortest distance algorithm:
To find the set of vertices reachable from an initial vertex v_s
Create and Initialize set of *reachable* vertices
Add v_s to a stack
While stack is not empty

Get and remove (pop) last vertex v from stack
If v is known to be reachable, discard
Otherwise, add v to set of reachable vertices, then
    For all neighbors, v_j, of v
    If v_j is not is set of reachable vertices, add to stack

At each step of processing one node is removed from the stack. If it is a node that has not previously been reported as reachable, then the neighbors of the node are pushed on the stack. When the stack is finally empty, then all nodes that can be reached will have been investigated.

Question: Simulate the execution of this algorithm on the graph given earlier, using Pierre as your starting location. What city is not reachable from Pierre?

The type of search performed by this algorithm is known as a *depth-first* search. This mirrors a search that might be performed by an ant walking along the edges of the graph, and returning to a previous node when it discovers it has reached a vertex it has already seen. What is interesting about this algorithm is that if we replace the data structure, the stack, with an alternative, the queue, we get an entirely different type of search. Now, this is known as a *breadth-first* search. A breadth-first search looks at all possible paths at the same time. A mental image for this type of search would be to pour ink at the starting vertex, and see where it travels along all possible paths. In Worksheet 41 you will explore the differences between depth and breadth first search.

The weighted graph version of this algorithm is known as Dijkstra's algorithm, again in honor of the computer scientist who first discovered it. Dijkstra's algorithm uses a priority queue instead of a stack. It returns a dictionary of vertex/distance pairs. Because the priority queue orders values on shortest distance, it is guaranteed to find the shortest path. Dijkstra's algorithm can be described as follows:

Dijkstra's Shortest Path Algorithm:
To find the shortest distance to vertices reachable from an initial vertex v_s
Initialize dictionary of *reachable* vertices with v_s having cost zero,
and add v_s to a priority queue with distance zero
While priority queue is not empty
    Get and remove (pop) vertex v with shortest distance from priority queue
    If v is known to be reachable, discard
    Otherwise, add v with given cost to dictionary of reachable vertices
        For all neighbors, v_j, of v
        If v_j is not is set of reachable vertices, combine cost of reaching v
        with cost to travel from v to v_j, and add to priority queue

It is important that you add a vertex to the set of reachable nodes only when it is removed from the collection. As you will see when you complete worksheet 42, the cost when a vertex is first encountered in the inner loop may not be the shortest path possible.

Because a priority queue is used, rather than the stack as in the earlier algorithm, a shorter path that is subsequently uncovered will nevertheless be placed earlier in the queue, and thus will be the first to be removed from the queue.

Other Graph Algorithms

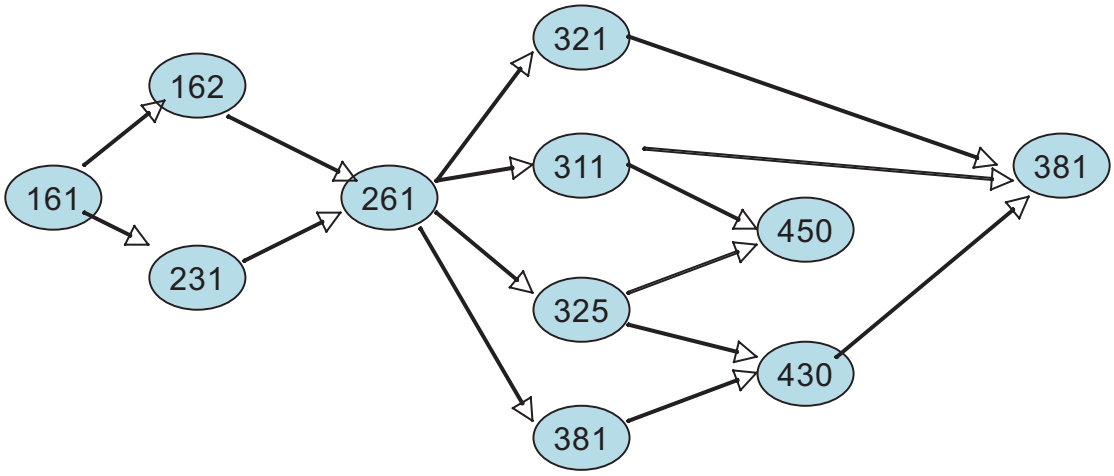
As we noted at the start of this chapter, graphs and matrices appear in a wide variety of forms in computer science applications. In this section we will present a number of other classic graph algorithms.

Greedy Algorithms

Many problems, particularly search problems, can be phrased in a form so that at each step there is a choice of many alternatives. The greedy method is a heuristic that provides guidance in how to construct a solution in these situations. The greedy technique says simply to take the next step that seems best from the current choices (that is, without resorting to any long-term planning). Dijkstra's algorithm is a classic greedy algorithm. At each step, we simply select the move that gets us to a new destination with the least amount of work.

Topological Sorting

Oftentimes a graph is used to represent a sequence of tasks, where a link between one task and the next indicates that the first must be completed before the second. When discussing university courses this is often termed a prerequisite chain. The following graph, for example, shows a typical table of courses and their prerequisites.



A topological order is a listing of vertices (in this case, courses), having the property that if b is a successor to a, then b occurs later in the ordering than a. For example, that you do not take any course without first taking its prerequisites. In effect, the topological ordering imposes a linear order on a graph. Of course, the topological order may not be unique, and many orderings may satisfy the sequence property.

A topological ordering can be found by placing all the vertices into a bag. The bag is then searched to find a vertex that has no successor. (As long as the graph has no cycles, such

a vertex must exist). This will be the end of our sorted list. The vertex is removed, and the bag examined once more. The result will be a topological sort.

Spanning Tree

A tree is a form of graph. Although all trees are graphs, not all graphs are trees. For example, trees have no cycles, and no vertex in a tree is pointed to by more than one other vertex. A spanning tree of a graph is a subset of the edges that form a tree that that includes all the vertices.

To construct a spanning tree you can perform a depth-first search of the graph, starting from a given starting node. If you keep the list of neighbor nodes in a priority queue, as in dijkstra's algorithm, you can find the lowest-cost spanning tree. This can be accomplished by modifying dijkstra's algorithm to record an edge each time it is added to the set of reachable nodes.

The Traveling Salesman

Another graph problem that is of both theoretical and practical interest is that of the traveling salesman. In this problem a weighted graph represents the cost to travel between cities. The task is to find a path that touches every city, crosses no city more than once, and has least total cost.

This problem is interesting because there is no known polynomial time algorithm to find a solution. The problem belongs to a large class of problems termed NP, which stands for nondeterministic polynomial. The nondeterministic characterization comes from the fact that once can *guess* a solution, and in polynomial time check to determine whether it is the correct solution. But the nature of NP problems gets even more curious. The traveling salesman problem is what is known as NP-complete. This is a large collection of problems that has the property that if any of them could be solved in polynomial time, they all could. Unfortunately, nobody has ever been able to find a fast algorithm for any of these; nor has anybody been able to prove that such an algorithm cannot exist.

Study Questions

1. What are the two components of a graph?

2. What is the difference between a directed and an undirected graph?

3. What is the difference between a weighted and an unweighted graph?

4. What is an adjacency matrix?

5. For a graph with V vertices, how much space will an adjacency matrix require?

6. what are the features of an edge-list representation of a graph?

7. How much space does an edge-list representation of a graph require?

8. What are the two varieties of reachability questions for a graph?

9. What is the asymptotic complexity of Warhsall's algorithm?

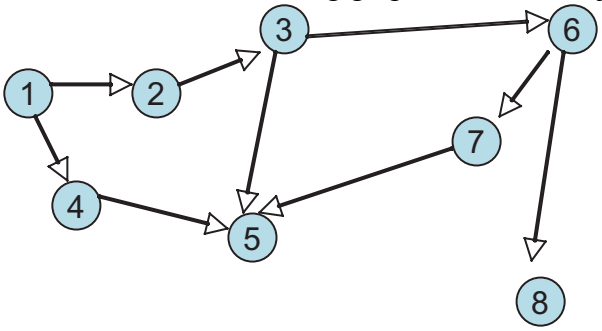
10. What problem is being addressed using Dijkstra's algorithm?

11. Why is it important that Dijkstra's algorithm stores intermediate results in a priority queue, rather than in an ordinary stack or queue?

12. In your own words, give an informal description of the difference between depth and breadth first search.

Exercises

1. Describe the following graph as both an adjacency matrix and an edge list:



2. Construct a graph in which a depth first search will uncover a solution (a path from one vertex to another) in fewer steps than will a breadth first search. You may need to specify an order in which neighbor vertices are placed into the container (e.g., lowest to highest). Construct another graph in which a breadth-first search will uncover a solution in fewer steps.

Analysis Exercises

1. Notice that the depth-first reachability algorithm did not specify what order neighboring cells should be placed into the container. Does the order matter? You can experiment by trying two different order, say listing the neighbors lowest-to-highest, and then highest-to-lowest.

2. A proof of correctness for Warshall's algorithm is based around the following induction technique. Prove, for each value of k, that if there is a path from node I to node j that does not go through any vertex with number higher than k, then a[i][j] is 1.

It is easy to establish the base case. For the induction step, assume that we are on the kth iteration, and that all paths that traverse only vertices labeled k or less have been marked. Assume that a path from vertex I to j goes through vertex k, and uses no vertices numbered larger than k. Argue why at the end of the loop the position a[i][j] will be set to 1.

3. A proof of correctness for the depth first reachability argument can be formed using induction, where the induction quantity is the length of the smallest path from the source to the destination. It is easy to argue that all vertices for which the shortest path is length 1 will be marked as reachable. For the induction step, argue why if all vertices with shortest path less than n are marked as reachable, then all vertices whose shortest path is length n will also be marked as reachable.

4. Give an example to illustrate why Dijkstra's algorithm will not work if there is an edge with a negative weight that is part of a cycle.

5. Rather than using induction, a proof of correctness for Dijkstra's algorithm is more easily constructed using proof by contradiction. Assume that the algorithm fails to find the shortest path of length greater than 1 from the source node to some node v, but instead finds a path of longer length. Because the path is larger than 1, there must be a next to last node along this path. Argue, from the properties of the priority queue, why this node must be processed before the node in question. Then argue why this will show that the shorter path will be discovered before the longer path that was uncovered.

6. Suppose you have a graph representing a maze that is infinite in size, but there is a finite path from the start to the finish. Is a depth first search guaranteed to find the path? Is a breadth-first search? Explain your answer.

7. Use the web to learn more about NP complete problems. What are some other problems that are NP complete? Why is the fact that factoring large prime number is NP complete a very practical importance at the present time?

Programming Projects

1. Using one of the representations for sets described in this text, provide an implementation of the depth-first shortest path algorithm.

2. Produce an implementation of Dijkstra's shortest path algorithm.

On the Web

Wikipedia has a collection of links to various algorithms under an entry “List of Algorithms”. Algorithms described include most of those mentioned in this chapter, as well as many others.

Worksheet 0: Building a Simple ADT Using an Array

In Preparation: Read about basic ADTs.

In this worksheet we will construct a simple BAG and STACK abstraction on top of an array. Assume we have the following interface file (arrayBagStack.h) :

```
# ifndef ArrayBagStack
# define ArrayBagStack

# define TYPE int
# define EQ(a, b) (a == b)

struct arrayBagStack {
     TYPE data [100];
     int count;
};

void initArray(struct arrayBagStack * b);
void addArray (struct arrayBagStack * b, TYPE v);
int containsArray (struct arrayBagStack * b, TYPE v);
void removeArray (struct arrayBagStack * b, TYPE v);
int sizeArray (struct arrayBagStack * b);

void pushArray (struct arrayBagStack * b, TYPE v);
TYPE topArray (struct arrayBagStack * b);
void popArray (struct arrayBagStack * b);
int isEmptyArray (struct arrayBagStack * b);
# endif
```

Your job, for this worksheet, is to provide implementations for all these operations.

```
void initArray (struct arrayBagStack * b){




}
```

```
/* Stack Interface  Functions */
void pushArray (struct arrayBagStack * b, TYPE v) {



}
TYPE topArray (struct arrayBagStack * b) {



}
void popArray (struct arrayBagStack * b) {



}
int isEmptyArray (struct arrayBagStack * b) {



}

/* Bag Interface Functions */
void addArray (struct arrayBagStack * b, TYPE v) {



}
```

```
int containsArray (struct arrayBagStack * b, TYPE v){









}

void removeArray (struct arrayBagStack * b, TYPE v) {









}

int sizeArray (struct arrayBagStack * b) {




}
```

A Better Solution…

This solution has one problem. The arrayBagStack structure is in the .h file and therefore exposed to the users of the data structure. How can we get around this problem? Think about it…we'll return to this question soon.

Worksheet 9: Summing Execution Times

In preparation: Read Chapter 4 to learn more about big-Oh notation.

Function	Common name	Running time
N!	Factorial	
2^n	Exponential	> century
N^d, d > 3	Polynomial	
N^3	Cubic	31.7 years
N^2	Quadratic	2.8 hours
N sqrt n		31.6 seconds
N log n		1.2 seconds
N	Linear	0.1 second
sqrt (n)	Root-n	$3.2 * 10^{-4}$ seconds
Log n	Logarithmic	$1.2 * 10^{-5}$ seconds
1	Constant	

The table at left, also found in Chapter 4, lists functions in order from most costly to least. The middle column is the common name for the function.

Suppose by careful measurement you have discovered that a program has the running time as shown at right. Describe the running time of each function using big-Oh notation.

$3n^3 + 2n + 7$	
(5 * n) * (3 + log n)	
1 + 2 + 3 + … + n	
$n + \log n^2$	
((n+1) log n) / 2	
$n^3 + n! + 3$	
$2^n + n^2$	
n (sqrt(n) + log n)	

Worksheet 9: Summing Execution Times Name:

Using the idea of dominating functions, give the big-Oh execution time for each of the following sequences of code. When elipses (…) are given you can assume that they describe only constant time operations.

for (int i = n; i > 0; i = i / 2) {
  …
}
for (int j = 0; j * j < n; j++) …	

for (int i = 0; i < n; i++) {
  for (int j = n; j > 0; j = j / 2) {
  …
  }
  for (int k = 0; k < n; k++) {
  …
  }
}	

for (int i = 0; i < n; i++)
  …
for (int j = 0; j * j < n; j++)
  …	

for (int i = 0; i < n; i++)
  …
for (int j = n; j > 0; j--)
  …	

for (int i = 1; i * i < n; i += 2)
  …
for (int i = 1; i < n; i += 5)
  …	

Worksheet 10: Using Big-Oh to Estimate Wall Clock Time

In preparation: Read Chapter 4 to learn more about the concept of big-oh notation.

As you learned in Chapter 4, a big-Oh description of an algorithm is a characterization of the change in execution time as the input size changes. If you have actual execution timings ("wall clock time") for an algorithm with one input size, you can use the big-Oh to estimate the execution time for a different input size. The fundamental equation says that the ratio of the big-Oh's is equal to the ratio of the execution times. If an algorithm is O(f(n)), and you know that on input n_1 it takes time t_1, and you want to find the time t_2 it will take to process an input of size n_2, you create the equation

$$f(n_1) / f(n_2) = t_1 / t_2$$

To illustrate, suppose you want to actually perform the mind experiment from Lesson 7. You ask a friend to search for the phone number for "Chris Smith" in 8 pages of a phone book. Your friend does this in 10 seconds. From this, you can estimate how long it would take to search a 256 page phone book. Remembering that binary search is O(log n), you set up the following equation:

$$\log(8)/\log(256), \text{ which is } 3 / 8 = 10 / X$$

Solving for X gives you the answer that your friend should be able to find the number in about 24 seconds. Now you time your friend perform a search for the name attached to a given number in the same 8 pages. This time your friend takes 2 minutes. Recalling that a linear search is O(n), this tells you that to search a 256 page phone could would require:

$$8/256 = 2 / X$$

Solving for X tells you that your friend would need about 64 minutes, or about an hour. So a binary search is really faster than a linear search.

Linear search	O(n)
Binary search	O(log n)
countOccurrences	O(n)
isPrime	O(Sqrt(n))
printPrimes	O(nSqrt(n))
matMult	O(n^3)
SelectionSort	O(n^2)

Recall from the last lesson the table of execution times for common algorithms. Using these, estimate the actual execution times for various tasks in the following problems.

Worksheet 10: Using Big-Oh to Estimate Wall Clock Time Name:

Problem 1. Warehouse video keeps a list of titles for their 7000 item inventory in a simple unsorted list. To find out how many copies of “Kill Bill” they have using the countOccurrences algorithm takes about 45 seconds. They recently acquired a competing video store, and now their inventory has 43000 items. How long will it take to search?

Problem 2. Suppose you can multiply two 17 by 17 matrices in 33 seconds. How long will it take to multiply two 51 by 51 element matrices. (By the way 51 is 17 times 3).

Problem 3. If you can print all the primes between 2 and 10000 in 92 seconds, how long will it take to print all the primes between 2 and 160000?

Worksheet 11: Recursive Functions and Recurrence Relations

In Preparation: Read the section on recursive functions and recurrence relations in Chapter 4.

```
void printBinary (int v) {
   if ((v == 0) || (v == 1))
       print(n);
   else {
      printBinary(v/2);
      print(v%2);
   }
}
```

As described in Chapter 4, a very useful technique for analyzing recursive functions is to form *recurrence relations*. For example, suppose we are analyzing the function shown at left. Let n will represent the number of binary digits in the printed form. Because the argument is divided by 2 prior to the recursive call, it will have one fewer digit. Everything else, outside of the recursive call, can be performed in constant time. The base case can also be performed in constant time. If we let T(n) represent the time necessary to print an n-digit number, we have the following equation:

$$T(n) = T(n-1) + c$$
$$T(1) = c$$

Read this as asserting that the time for n is equal to the time for n-1 plus some unknown constant value. This is termed a recurrence relation.

$T(n) = T(n-1) + c$	$O(n)$
$T(n) = T(n/2) + c$	$O(\log n)$
$T(n) = 2 * T(n/2) + c_a n + c_b$	$O(n \log n)$
$T(n) = 2 * T(n-1) + c$	$O(2^n)$

The four most common recurrence relations and their solutions are shown at left.

The following are some example recursive functions. Describe the running time for each as a recurrence relation, and then give the big-Oh execution time:

```
factorial(int n) {
  if (n = 0) return 1;
  else return n * factorial(n-1)
}
```

```
double exp (double a, int n) {
  if (n == 0) return 1.0:
  else return a * exp (a, n-1);
}
```

```
void Hanoi (int n, int a, int b, int c) {
  if (n > 0) {
```

```
    Hanoi(n-1, a, c, b);
    System.out.println("Move from " + a + " to " + b  + "using " + c);
    Hanoi(n-1, c, b, a);
  }
}
```

For the following, let n represent the size of the region of the array between low and high

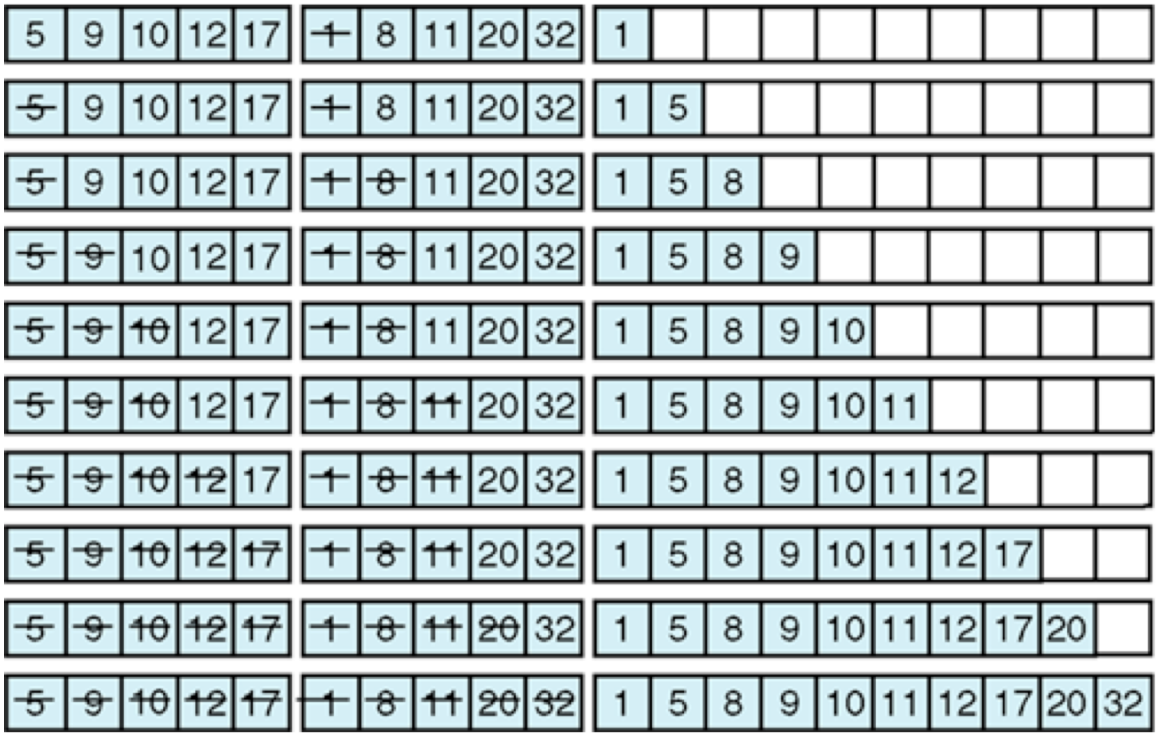
```
int binarySearch (double data[ ], int low, int high, double test) {
  if (low == high) return low;
  int mid = (low + high)/2;
  if (test < data[mid]) return binarySearch(data, low, mid, test)
  else return binarySearch(data, mid, high, test);
}
```

Worksheet 12: Merge Sort – A Fast Recursive Sorting Algorithm

In Preparation: Read the description of Merge Sort in Chapter 4.

In earlier lessons you were introduced to two classic sorting algorithms, selection sort and insertion sort. While easy to explain, both are $O(n^2)$ worst case. Many sorting algorithms can do better. In this lesson we will explore one of these.

As with many algorithms, the intuition for merge sort is best understood if you start in the middle, and only after having seen the general situation do you examine the start and finish as special cases. For merge sort the key insight is that two already sorted arrays can be very rapidly merged together to form a new collection. All that is necessary is to walk down each of the original lists in order, selecting the smallest element in turn:



When you reach the end of one of the arrays (you cannot, in general, predict which list will end first), you must copy the remainder of the elements from the other.

Chapter 4 described how a sorting algorithm, Merge sort, could be constructed based on these ideas. Merge sort can be written as follows:

```
void mergeSort (double data [ ], int n) {
  double * temp = (double *) malloc (n * sizeof(double));
  assert (temp != 0); /* make sure allocation worked */
  mergeSortInternal (data, 0, n-1, temp);
```

```
}

void mergeSortInternal (double data [ ], int low, int high, double temp [ ]) {
  int i, mid;
  if (low >= high) return; /* base case */
  mid = (low + high) / 2;
  mergeSortInternal(data, low, mid, temp); /* first recursive call */
  mergeSortInternal(data, mid+1, high, temp); /* second recursive call */
  merge(data, low, mid, high, temp); /* merge into temp */
  for (i = low; i <= high; i++)  /* copy merged values back */
    data[i] = temp[i];
}
```

The only part left unfinished is to write the routine merge. Complete this routine in the space below:

```
void merge (double [ ] data, int low, int mid, int high, double [ ] temp) {









}
```

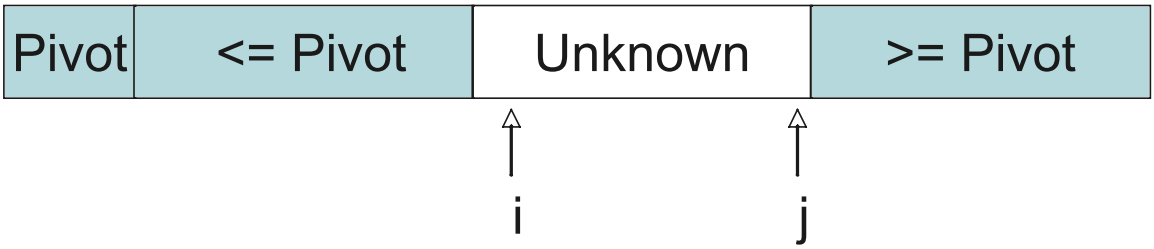
Worksheet 13: Quick Sort – A *Usually* Fast Sorting Algorithm

In Preparation: Reach the description of quick sort in Chapter 4.

As you learned in Chapter 4, quick sort is another fast sorting algorithm. Even better, quick sort orders an array in-place, using no additional memory. The quick sort algorithm can be written as follows:

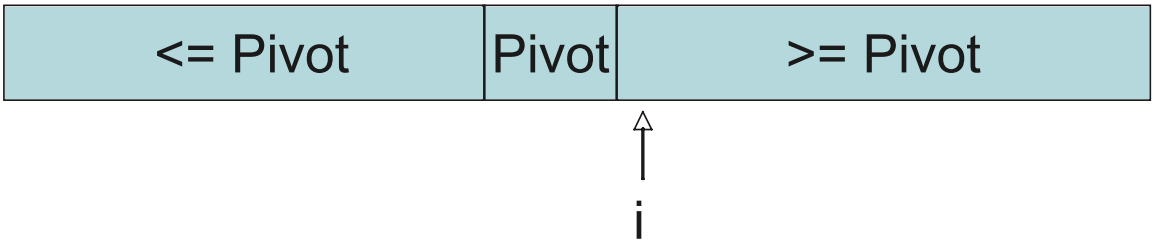
```
public void quickSort (double storage[ ], int n)
  { quickSortInternal (storage, 0, n-1); }

private void quickSortInternal (double storage [ ], int low, int high) {
  if (low >= high) return; // base case
  int pivot = (low + high)/2; // one of many techniques
  pivot = partition(storage, low, high, pivot);
  quickSortInternal (storage, low, pivot-1); // first recursive call
  quickSortInternal (storage, pivot+1, high); // second recursive call
}
```

To make this a sorting algorithm it is only necessary to write the function named partition. The process of dividing a portion of an array into two sections is termed *partitioning*. The limits of the partition are described by a pair of values: low and high. The first represents the lowest index in the section of interest, and the second the highest index. In addition there is a third element that is selected, termed the pivot. The first step is to swap the element at the pivot location and the first position. This moves the pivot value out of way of the partition step. The variable i is set to the next position, and the variable j to high. The heart of the partition algorithm is a while loop. The invariant that is going to be preserved is that all the elements with index values smaller than i are themselves smaller than or equal to the pivot, while all the elements with index values larger than j are themselves larger than the pivot. At each step of the loop the value at the i position is compared to the pivot. If it is smaller or equal, the invariant is preserved, and the i position can be advanced:



Otherwise, the location of the j position is compared to the pivot. If it is larger then the invariant is also preserved, and the j position is decremented. If neither of these two conditions is true the values at the i and j positions can be swapped, since they are both out of order. Swapping restores our invariant condition.

The loop proceeds until the values of i and j meet and pass each other. When this happens we know that all the elements with index values less than i are less than or equal to the pivot. This will be the first section. All those elements with index values larger than or equal to i are larger than or equal to the pivot. This will be the second section. The pivot is swapped back to the top of the first section, and the location of the pivot is returned



From the preceding description you should be able to complete the partition algorithm, and so complete the quick sort algorithm.

```
int partition(double storage [ ], int low, int high, int pivot) {

}
```

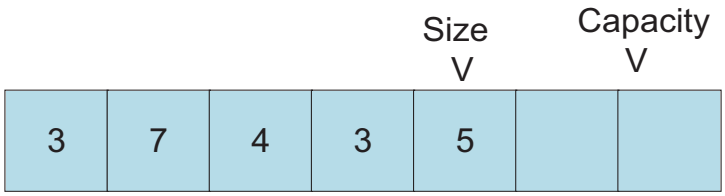
Worksheet 14: Introduction to the Dynamic Array

In Preparation: Read Chapter 5 to learn more about abstraction and the basic abstract data types, and the dynamic array as a n implementation technique.

Suppose you want to write a program that will read a list of numbers from a user, place them into an array and then compute their average. How large should you make the array? It must be large enough to hold all the values, but how many numbers will the user enter? A common solution to this difficulty is to create an array that is larger than necessary, and then only use the initial portion. This is termed a *partially filled array*.

When you use a partially filled array there are two integer values of interest. First is the array length. When discussing partially filled array this is sometimes termed the *capacity* of the array. Second is the current *size*, that is, the amount of the array that is currently being used. The size is generally maintained by a separate integer variable. If we place the values into a structure it will be easier to keep them together.

```
struct partFillArray {
   double data[50];
   int size;
   int capacity;
};
```



It is important to not confuse the size and the capacity. For example, in computing the sum and average you do not want to use the capacity:

```
double average (struct partFillArray * pdata) {
   double sum = 0.0;
   int i;
   for (i = 0; i < 50; i++)  /* Error—loop using length */
      sum = sum + pdata->data[i]; /* Error-uninitialized value */
   return sum / 50; /* Error—average is under estimated */
}
```

Instead, you want to use the size:

```
double average (struct partFillArray * pdata) {
   double sum = 0.0;
   int i;
   for (i = 0; i < pdata->size; i++)
      sum = sum + pdata->data[i];
   return sum / pdata->size;
}
```

The technique of partially filled arrays works fine until the first time that the user enters more numbers than were originally anticipated. When this happens, the size can exceed the capacity, and unless some remedial action is taken an array indexing error will occur. Worse yet, since the validity of index values is not checked in C, this error will not be reported and may not be noticed.

A common solution to this problem is to use a pointer to a ***dynamically allocated array***, rather than a fixed length array as shown above in the partFillArray. Of course, this means that the array must be allocated before it can be used. We can write an initialization routine for this purpose, and a matching routine to free the data. Next, we likewise encapsulate the action of adding a new element into a function. This function can check that the size does not exceed the capacity, and if it does increase the length of the array (generally by doubling) and copying all the elements into the new area. Now the user can enter any number of values and the data array will be automatically expanded as needed.

```
struct DynArr
{
	TYPE *data;	/* pointer to the data array */
	int size;	/* Number of elements in the array */
	int capacity;	/* capacity ofthe array */
};

void initDynArr(struct DynArr v, int capacity)  /* internal function , thus the _underscore */
{
	v->data = malloc(sizeof(TYPE) * capacity);
	assert(v->data != 0);

	v->size = 0;
	v->capacity = capacity;

}

struct DynArr* createDynArr(int cap)
{
	DynArr *r;
	assert(cap > 0);
	r = malloc(sizeof( DynArr));
	assert(r != 0);
	_initDynArr(r,cap);
	return r;
}
void freeDynArr(struct DynArr *v)
{
	if(v->data != 0)
	{
		free(v->data);	/* free the space on the heap */
		v->data = 0;	/* make it point to null */
	}
	v->size = 0;
	v->capacity = 0;
}
```

```
void deleteDynArr(DynArr *v)
{
        assert (v!= 0);
        freeDynArr(v);
        free(v);
}

void sizeDynArr( struct DynArr *v)
{
        return v->size;
}

void   addDynArr(struct DynArr *v, TYPE val)
{
        /* Check to see if a resize is necessary */
        if(v->size >= v->capacity)
                _setCapacityDynArr(v, 2 * v->capacity);

        v->data[v->size] = val;
        v->size++;
}
```

The only thing missing now is the _setCapacityDynArr function. Complete the implementation of _setCapacityDynArr. Pay careful attention to the order of operations. Remember that since you're creating a new array, you'll want to eventually get rid of the old one to avoid a 'memory leak'.

```
void _setCapacityDynArr(struct DynArr *v, int newCap)
{









}
```

The code shown above introduces a number of features of the C language that you may not have seen previously. Let us describe some of these here.

`TYPE`. We want to create a library of general-purpose functions for managing collections of various types of elements. In order to make our code completely independent from the type of value being stored, we have defined the element type using a symbolic name, `TYPE`. The use of ifdef surrounding the definition is a common C idiom. If the user has already provided an alternative definition we will use that, otherwise the symbolic name is given a default value of `int`.

`initDynArr` and `freeDynArr`. The C language does not provide a way to automatically initialize a structure, such as a constructor does for you in Java or C++. Instead, programmers must typically write a special initialization function. Programmers must then remember to initialize a structure, and to free memory when they are finished with the structure. In our case, we've also provided functions that create and initialize the struct (as well as free and delete it).

```
struct DynArr *myData;
myData = createDynArr(50); /* creates and initializes a dynamic array with capacity of 50 */
…
deleteDynArr(myData);  /*frees the memory of the internal array AND the struct DynArr*/
```

`da->size`. Whenever you use a pointer you must make it clear when you are referring to the pointer itself and when you are referring to the value it points to. Normally a pointer value must first be dereferenced, using the * operator, to indicate that you mean the value it points to, not the pointer itself. Accessing a field in a structure referred to by a pointer could be written using the dereference operator, as in (*da).size. However, this combination of pointer dereferencing and field access occurs so frequently that the designers of C provided a convenient shorthand.

`malloc`. The function `malloc` is used to perform *m*emory *alloc*ation in C. The argument is an integer indicating the number of bytes requested. In order to determine how many bytes are required for each element the function `sizeof` is invoked. Multiply the number of elements you need by the size of each element, and you have a block of memory that can be used as an array.

`assert`. The malloc function will return zero if there is insufficient memory to satisfy a request. The `assert` macro will halt execution with an error message if its argument expression is not true. Assertions can be used any time a condition must be satisfied in order to continue.

`free`. The function `free` is the opposite of `malloc`. It is used to return a block of memory to the free store. Such memory might later be reused to satisfy a subsequent `malloc`

request. You should never use `malloc` without knowing where and when the memory will subsequently be freed.

Defensive programming. When the memory is released in the function `freeDynArr` the size and the capacity are both set to zero. This ensures that if a subsequent attempt is made to insert a value into the container, there will not be an attempt to index into an already deleted array.

`sizeDynArr`. Since the size field is stored as part of the dynamic array structure there really is no need for this function, since the user can always access the field directly. However, this function helps preserve encapsulation. The end user for our container need not understand the structure definition, only the functions needed to manipulate the collection. *How would your code have to change in order to completely hide the structure away from the user?*

_setCapacityDynArr. An underscore is treated as a legal letter in the C language definition for the purposes of forming identifiers. There is a common convention in the C programming community that function names beginning with an underscore are used "internally", and should never be directly invoked by the end user. We will follow that convention in our code. The function _setCapacityDynArr can be called by dynamic array functions, but not elsewhere.

Note carefully the order of operations in the function _setCapacityDynArr. First, the new array is created. Next, the old values are copied into the new array. The free statement then released the old memory. Finally, the pointer is changed to reference the new array.

In order to allow a dynamically allocated array to be used in the same fashion as a normal array, we need functions that will get and set values at a given position. We can also make our function more robust than the regular C array by checking index positions. Complete the implementation of the following functions. *Use assert to check that index positions are legal*.

```
TYPE getDynArr (struct DynArr * da, int position) {




}
```

```
void putDynArr(struct DynArr * da, int position, TYPEvalue) {




}
```

Write the function swap, which will exchange the values in two positions of a dynamic array. We will use this function in later chapters.

```
void swapDynArr (struct DynArr * da, int i, int j) {




}
```

Write the function removeAtDynArr, which will remove a value at a specified index. Remember, we do not want to leave gaps in the partially filed array. We will use this function in later chapters.

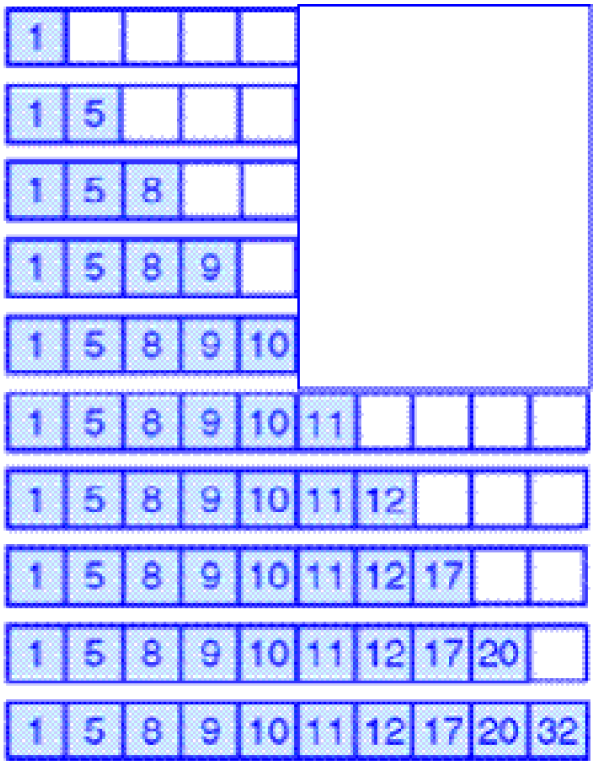
```
void removeAtDynArr (struct DynArr * da, int index) {




}
```

Worksheet 15: Amortized Constant Execution Time

In preparation: Read Chapter 5 to learn more about the basic abstract data types, and the introduction to the dynamic array. If you have not done so already you should complete worksheet 14, which introduces the dynamic array.

In the previous worksheet you analyzed the algorithmic execution time for the **add** operation in a dynamic array. When the capacity was less than the size, the execution time was constant. But when a reallocation became necessary, execution time slowed to O(n).

This might at first seem like a very negative result, since it means that the worst case execution time for addition to a dynamic array is O(n). But the reality is not nearly so bleak. Look again at the picture that described the internal array as new elements were added to the collection.



Notice that the costly reallocation of a new array occurred only once during the time that ten elements were added to the collection. If we compute the *average* cost, rather than the *worst case* cost, we will see that the dynamic array is still a relatively efficient container.

To compute the average, count 1 "unit" of cost each time a value is added to the dynamic array without requiring a reallocation. When the reallocation occurs, count one "unit" of cost for each assignment performed as part of the reallocation process, plus one more for placing the new element into the newly enlarged array. How many "units" are spent in the entire process of inserting these ten elements? What is the average "unit" cost for an insertion?

When we can bound an "average" cost of an operation in this fashion, but not bound the worst case execution time, we call it *amortized constant* execution time. Amortized constant execution time is often written as O(1)+, the plus sign indicating it is not a guaranteed execution time bound.

Do a similar analysis for 25 consecutive add operations, assuming that the internal array begins with 5 elements (as shown). What is the cost when averaged over this range?

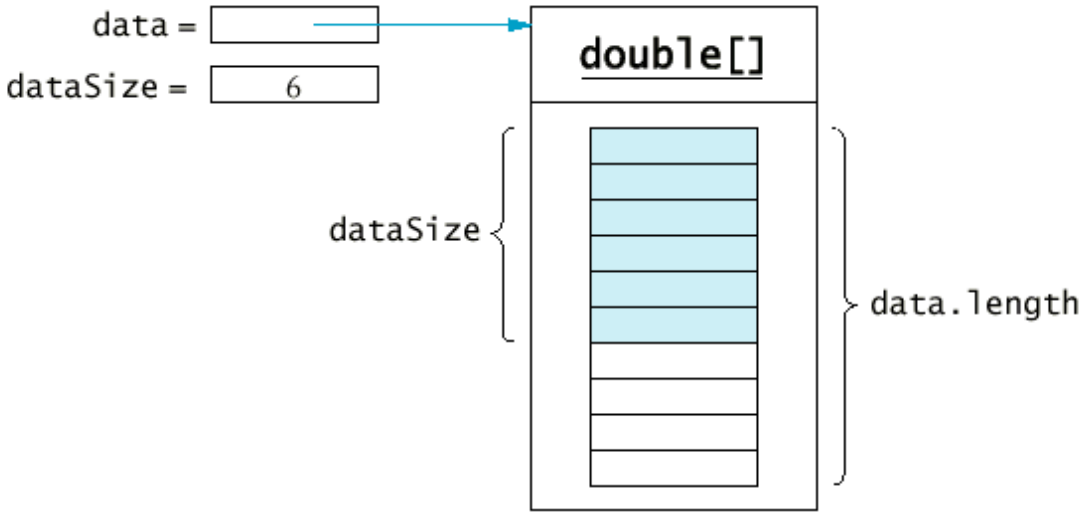
Worksheet 16: Dynamic Array Stack

In Preparation: Read Chapter 6 to learn more about the Stack data type. If you have not done so already, you should complete worksheets 14 and 15 to learn about the basic features of the dynamic array.

In Chapter 6 you read about the Stack data abstraction. A stack maintains values in order based on their time of insertion. When a value is removed from the stack it is the value that has been most recently added to the stack. The abstract definitions of the stack operations are shown at right.

Conceptual Stack Operations
void push (TYPE newValue)
TYPE top ()
void pop ()
Boolean isEmpty ()

As you learned in Worksheet 14, a positive feature of the array is that it provides random access to values. Elements are accessed using an index, and the time it takes to access any one element is no different from the time it takes to access another. However, a fundamental problem of the simple array is that the size must be specified at the time the array is created. Often the size cannot be easily predicted; for example if the array is being filled with values being read from a file. A solution to this problem is to use a *partially filled array*; an array that is purposely larger than necessary. A separate variable keeps track of the number of elements in the array that have been filled.



The *dynamic array* data type uses this approach. The array of values is encapsulated within a structure boundary, as is the current *size* of the collection. The size represents the number of elements in the array currently in use. The size is different from the *capacity*, which is the actual size of the array. Because the array is referenced by a pointer, an allocation routine must be called to set the initial size and create the initial memory area. A separate destroy routine frees this memory. You wrote these earlier in Worksheet 14.

The function add`DynArray(struct DynArr * da, TYPE v)` adds a new value to end of a dynamic array. Recall from Worksheet 14 that this function could potentially increase the size of the internal buffer if there was insufficient space for the new value. This is shown in the following two pictures. In the first picture there is space for the new value, so no reallocation is needed. In the second picture there is no longer enough space, and so a new buffer is created, the elements are copied from the old buffer to the new, and the value is then inserted into the new buffer. You wrote the function `dynArrayAdd` in worksheet 14. Do you remember the worst-case algorithmic execution time for this function?

Your task in this worksheet is to write the code for the Stack functions push, pop, top and isEmpty. These functions should use a dynamic array (passed as an argument) for the storage area. Use an assertion to check that the stack has at least one element when the functions top or pop are called. Your job will be greatly simplified by making use of the following functions, which you developed in previous lessons:

```
struct DynArr {
      TYPE * data;
      int size;
      int capacity;
      };

      /* initialize a dynamic array structure with given capacity */
void initDynArr (struct DynArr * da, int initialCapacity);

      /* internal method to double the capacity of a dynamic array */
void _setCapacityDynArr (struct DynArr * da);

      /* release dynamically allocated memory for the dynamic array */
void freeDynArr (struct DynArr * da);

      /* return number of elements stored in dynamic array */
int sizeDynArr (struct DynArr * da);

      /* add a value to the end of a dynamically array */
void addDynArr (struct DynArr * da, TYPE e);

      /* remove the value stored at position in the dynamic array */
void removeAtDynArr (struct DynArr * da, int position);

      /* retrieve element at a given position */
TYPE getDynArray (struct DynArr * da, int position);

      /* store element at a given position */
void putDynArr (struct DynArr * da, int position, TYPE value);
```

```
# define TYPE int

struct DynArr {
        TYPE *data;
        int size;
        int capacity;
};


/* Dynamic Array implementation of the Stack Interface */

void pushDynArray (struct DynArr * da, TYPE e) {



}

TYPE topDynArray (struct DynArr * da) {




}

void popDynArray (struct DynArr * da) {



}

int isEmptyDynArray (struct DynArr * da) {


}
```

Questions

1. What is the algorithmic execution time for the operations **pop** and **top**?

2. What is the algorithmic execution time for the operation **push,** assuming there is sufficient capacity for the new elements?

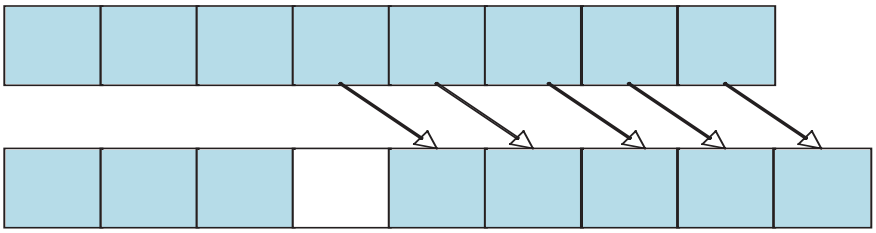
3. What is the algorithmic execution time for the internal method **_setCapacityDynArr**?

4. Using as a basis your answer to 3, what is the algorithmic execution time for the operation **push** assuming that a new array must be created.

Worksheet 17: Linked List Introduction, List Stack

In Preparation: Read Chapter 6 to learn more about the Stack data type.

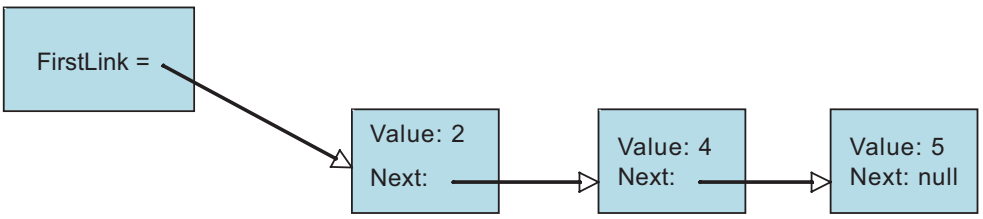
In Worksheet 14, you built a Stack using a dynamic array as the underlying container. A weakness of the Dynamic Array is that elements are stored in a contiguous block. As a consequence, when a new element is inserted into the middle of the collection, all the adjacent elements must be moved in order to make space for the new value.



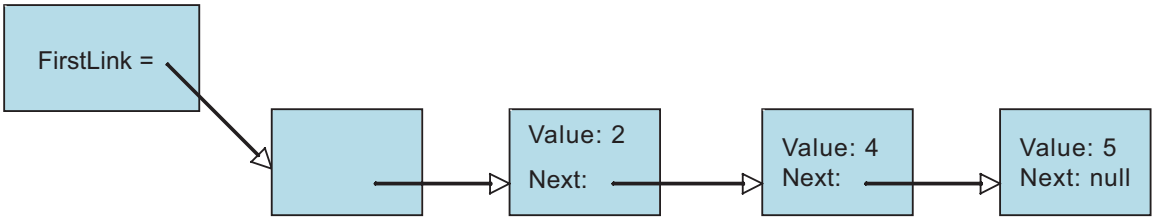
An alternative approach is to use the idea of a *Linked List*. In a linked list each value is stored in a separate block of memory, termed a *link*. In addition to a value, each link contains a reference to the next link in sequence. As a data structure, a link can be described as shown at right.

```
struct link {
   TYPE value;
   struct link * next;
};
```

We can visualize collection formed out of links as follows. A data field named firstLink will hold the first link in the collection. Each link refers to the next. The final link will have a **null** value in the next field:



The simplest data structure to create using links is a Stack. When a new element is pushed on the stack a new link will be created and placed at the front of the chain.



To remove a link the variable **firstLink** is simply changed to point to the next element in the chain. The space for the Link must then be freed.

The following is the beginning of an implementation of a **LinkedListStack** based on these ideas. Complete the implementation. Each operation should have constant time performance. Use an assertion to ensure that when a top or pop is performed the stack has at least one element. When you pop a value from the stack, make sure you free the link field.

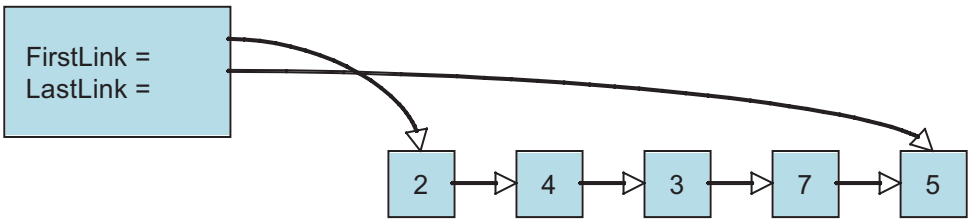
```
struct link {
   TYPE value;
   struct link * next;
};

struct linkedListStack {
   struct link *firstLink;
}

void initLinkedListStack (struct linkedListStack * s)
            { s->firstLink = 0; }

void freeLinkedListStack (struct linkedListStack *s)
      { while (! isEmptyLinkedListStack (s)) popLinkedListStack (s); }

void pushLinkedListStack (struct linkedListStack *s, TYPE d) {
   struct link * newLink = (struct link *) malloc(sizeof(struct link));
   assert (newLink != 0);









}

double topLinkedListStack (struct linkedListStack *s) {




}

void popLinkedListStack (struct linkedListStack *s) {



}
int isEmptyLinkedListStack (struct linkedListStack *s) {



}
```

Worksheet 18: Linked List Queue, pointer to Tail

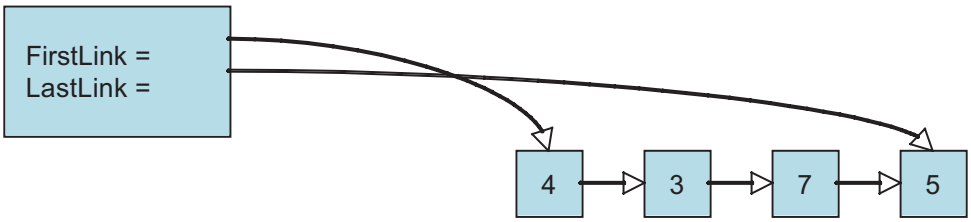
In Preparation: Read Chapters 6 and 7 to learn more about the Stack and Queue data types. If you have not done so already, you should do Worksheet 17 to learn about the basic features of a linked list.

One problem with the linked list *stack* is that it only permits rapid access to the front of the collection. This was no problem for the stack operations described in Worksheet 17, but would present a difficulty if we tried to implement a *queue*, where elements were inserted at one end and removed from the other.

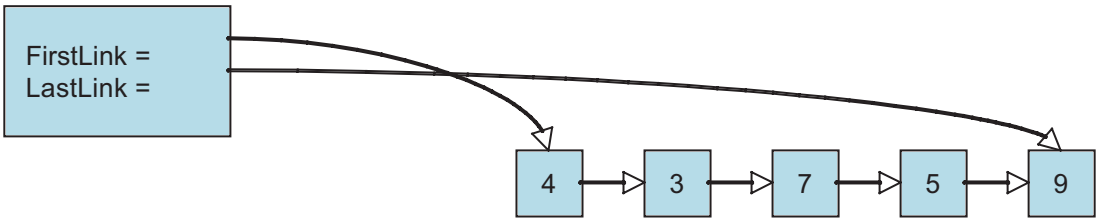
A solution to this problem is to maintain two references into the collection of links. As before, the data field firstLink refers to the first link in the sequence. But now a second data field named lastLink will refer to the final link in the sequence:



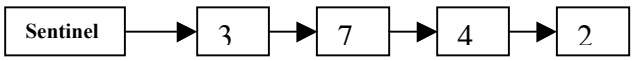
To reduce the size of the image the fields in the links have been omitted, replaced with the graphic arrow. Both the firstLink and lastLink fields can be **null** if a list is empty. Removing a data field from the front is the same as before, with the additional requirement that when the last value is removed and the list becomes empty the variable **lastLink** must also be set to null.



Because we have a link to the back, it is easy to add a new value to the end of the list.



We will add another variation to our container. A *sentinel* is a special link, one that does not contain a value. The sentinel is used to mark either the beginning or end of a chain of links. In our case we will use a sentinel at the front. This is sometimes termed a list header. The presence of the sentinel makes it easier to address special cases. For example, the list of links is never actually empty, even when it is logically empty, since there is always at least one link. A new value is inserted after the end.



Values are removed from the front, as with the stack. But because of the sentinel, the element removed will be the element right after the sentinel. Make sure that when you remove a value from the collection you free the memory for the associated link.

You should complete the following skeleton code for the ListQueue. The structures have been written for you, as well as the initialization routine. The function isEmpty must determine whether or not the collection has any elements. What property characterizes an empty queue?

On Your Own

1. Draw a picture showing the values of the various data fields in an instance of ListQueue when it is first created.

2. Draw a picture showing what it looks like after one element has been inserted.

3. Based on the previous two drawings, can you determine what feature you can use to tell if a list is empty?

4. Draw a picture showing what it looks like after two elements have been inserted.

5. What is the algorithmic complexity of each of the queue operations?

6. How difficult would it be to write the method addFront(newValue) that inserts a new element into the front of the collection? A container that supports adding values at either and, but removal from only one side, is sometimes termed a *scroll*.

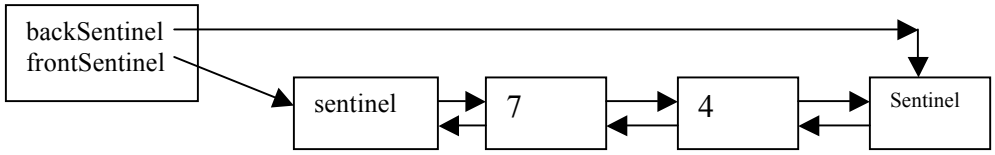
7. Explain why removing the value from the back would be difficult for this container. What would be the algorithmic complexity of the removeLast operation?

```
struct link {
   TYPE value;
   struct link * next;
};

struct listQueue {
   struct link *firstLink;
   struct link *lastLink;
};

void initListQueue (struct listQueue *q) {
   struct link *lnk = (struct link *) malloc(sizeof(struct link));
   assert(lnk != 0); /* lnk is the sentinel */
   lnk->next = 0;
   q->firstLink = q->lastLink = lnk;
}

void addBackListQueue (struct listQueue *q, TYPE e) {









}

TYPE frontListQueue (struct listQueue *q) {




}


void removeFrontListQueue (struct listQueue *q) {









}

int isEmptyListQueue (struct listQueue *q) {




}
```

Worksheet 19: Linked List Deque

In Preparation: Read Chapter 7 to learn more about the Deque data type. If you have not done so already, you should complete worksheets 17 and 18 to learn about the basic features of a linked list.

In this lesson we continue looking at variations on the theme of linked lists, this time including double links and sentinels on both the front and the back of the list. This will be the first of several lessons that will develop a very general purposed linked list abstraction. In this lesson we will emphasize the deque aspects of the list. In the following lesson we will add more operations.



A deque, you recall, allows insertions at both the beginning and the end of the container. The conceptual interface is shown at right. (Recall that the actual functions may differ from the conceptual interface in two ways. First, the names are likely differ to accommodate the C restriction that all functions have unique names. Second, the actual functions will pass the underlying data area as an argument).

Conceptual Deque operations

```
void addFront (TYPE e)
void addBack (TYPE e)
TYPE front ()
TYPE back ()
void removeFront ()
void removeBack ()
int isEmpty ()
```

Our linkedList data structure will have two major variations from the linked list stack and queue you have examined earlier. First, an integer data field will remember the number of elements (that is, the size) in the container. Second, we will use sentinels at both the front and back of the linked list.

Sentinels ensure the list is never completely empty. They also mean that all instructions can be described as special cases of more general routines. The internal method `_addBefore` will add a link before the given location. The second routine `_removeLink`, will remove its argument link. Both of these use the underscore convention to indicate they are internal methods. When a value is removed from a list make sure you free the associated link fields. Use an assertion to check that allocations of new links are successful, and that you do not attempt to remove a value from an empty list. The following instructions will help you understand how to implement these operations.

1. Draw a picture of an initially empty LinkedList, including the two sentinels.

2. Draw a picture of the LinkedList after the insertion of one value.

3. Based on the previous two pictures, can you describe what property characterizes an empty collection?

4. Draw a picture of a LinkedList with three or more values (in addition to the sentenels).

5. Draw a picture after a value has been inserted into the front of the collection. Notice that this is between the front sentinel and the following element. Draw a picture showing an insertion into the back. Notice that this is again between the last element and the ending sentinel. Abstracting from these pictures, what would the function addBefore need to do, where the argument is the link that will follow the location in which the value is inserted.

6. Draw a picture of a linkedList with three or more values, then examine what is needed to remove both the first and the last element. Can you see how both of these operations can be implemented as a call on a common operation, called _removeLink?

7. What is the algorithmic complexity of each of the deque operations?

```
struct dlink {
   TYPE value;
   struct dlink * next;
   struct dlink * prev;
};

struct linkedList {
   int size;
   struct dlink * frontSentinel;
   struct dlink * backSentinel;
};

/* these functions are written for you */
void  initLinkedList (struct linkedList *q) {
   q->frontSentinel = malloc(sizeof(struct dlink));
   assert(q->frontSentinel != 0);
   q->backSentinel = malloc(sizeof(struct dlink));
   assert(q->backSentinel);
   q->frontSentinel->next = q->backSentinel;
   q->backSentinel->prev = q->frontSentinal;
   q->size = 0;
}

void freeLinkedList (struct linkedList *q) {
   while (q->size > 0)
      linkedListRemoveFront(q);
   free (q->frontSentinel);
   free (q->backSsentinel);
   q->frontSentinel = q->backSentinel = null;
}

void addFrontLinkedList (struct linkedList *q, TYPE e)
  { _addBefore(q, q->frontSentinel_>next, e); }

void addBackLinkedList (struct linkedList *q, TYPE e)
  { _addBefore(q, q->backSentinel, e); }

void removeFrontLinkedList (struct linkedList *q) {
   assert(! linkedListIsEmpty(q));
   _removeLink (q, q->frontSentinal->next);
}

void removeBackLinkedList (struct linkedList *q) {
   assert(! linkedListIsEmpty(q));
   _removeLink (q, q->backSentinel->prev);
}

int isEmptyLinkedList (struct linkedList *q) {
   return q->size == 0;
}
```

/* write _addBefore and _removeLink. Make sure they update the size field correctly */

/* _addBefore places a new link BEFORE the provide link, lnk */
void _addBefore (struct linkedList *q, struct dlink *lnk, TYPE e) {

}

void _removeLink (struct linkedList *q, struct dlink *lnk) {

}

TYPE frontLinkedList (struct linkedList *q) {

}

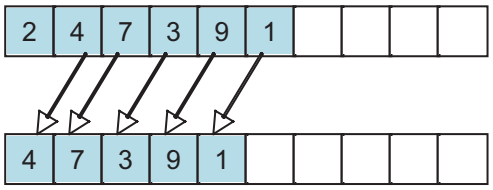
TYPE backLinkedList(struct linkedList *q) {

}

Worksheet 20: Dynamic Array Deque and Queue

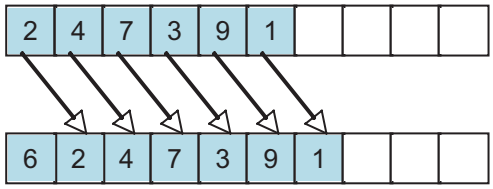
In Preparation: Read Chapter 7 to learn more about the Queue and Deque data types. If you have not done it already, complete worksheets 14 and 16 to learn more about the basic features of the dynamic array data type.

A question in the worksheet on the dynamic array stack asked why you cannot use a *Dynamic Array* as the basis for an efficient queue. This is because adding to or removing from the first location (that is, index position 0) is very slow O(n).

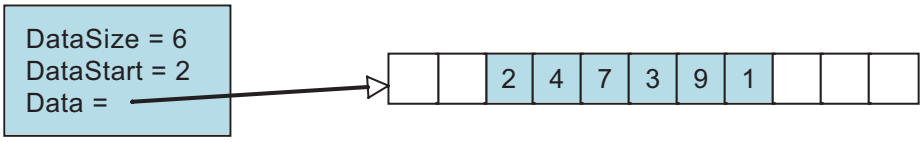


Removing a value requires elements to slide left. Sliding left means every element must be moved, and hence is an O(n) operation.

Adding a value requires elements to slide right. Sliding right opens up the location where a new value can be inserted. Once more, however, every element must be moved, and hence this is an O(n) operation.

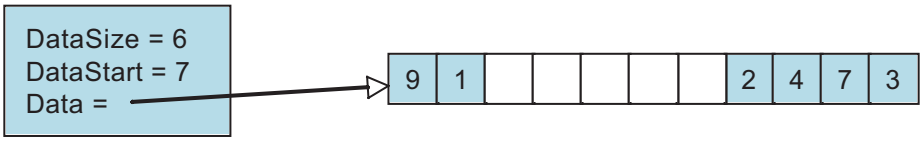


This does not mean that a Dynamic Array-*like* data structure cannot be used to implement a queue. The key insight is that we can allow the starting location of the block of elements to "float", rather than being fixed at location zero. An internal integer data field records the current starting location.



Notice that the "logical" index no longer corresponds to the physical index. The value with logical index zero is found in this diagram at array location 2. The value with logical index 1 is found at location 3, and so on.

With this change, it is now easy to implement additions or removal from either front or back. To add to the front, simply decrement the starting location, and place the new element in the new starting place. To add to the back, simply increase the size, and place the new value in the location determined by the addition of the starting location and size. But there is one subtle complexity. The block of values stored in the collection can wrap around from the end back to the beginning:



Here the block of elements begins at index position 7. The next three elements are found in index positions 8, 9 and 10. But the element after that is found at index position zero. As with the Dynamic Array, the internal array must be reallocated and doubled if the count of elements becomes equal to the array capacity. The internal function `_dequeSetCapacity` will set the size of the internal buffer to the passed in capacity. The code for this function is written for you below. You should study this to see how the variable named "j" can wrap around the end of the data array as the values are copied into the new array. The new array always begins with the starting position set to zero.

Using this idea, complete the implementation of the Deque. Implement the methods that will add, retrieve, or remove an element from either the front or back. Explain why each operation will have constant (or amortized constant) execution time.

```
void _setCapacityDeque (struct deque *d, int newCap) {
        int i;

        /* Create a new underlying array*/
        TYPE *newData = (TYPE*)malloc(sizeof(TYPE)*newCap);
        assert(newData != 0);

        /* copy elements to it */
        int j = v->beg;
        for(i = 0; i <  v->size; i++)
        {
         newData[i] = v->data[j];
         j = j + 1;
         if(j >= v->capacity)
                j = 0;
        }

        /* Delete the oldunderlying array*/
        free(v->data);
        /* update capacity and size and data*/
        v->data = newData;
        v->capacity = newCap;
        v->beg = 0;
}

void freeDeque (struct deque *d) {
  free(d->data); d->size = 0; d->capacity = 0;
}
```

```
struct deque {
  TYPE * data;
  int capacity;
  int size;
  int start;
};

void initDeque (struct deque *d, int initCapacity) {
  d->size = d->beg = 0;
  d->capacity = initCapacity; assert(initCapacity > 0);
  d->data = (TYPE *) malloc(initCapacity * sizeof(TYPE));
  assert(d->data != 0);
}

int sizeDeque (struct deque *d) { return d->size; }

void _doubleCapacityDeque (struct deque *d);

void addFrontDeque (struct deque *d, TYPE newValue) {
  if (d->size >= d->capacity) _setCapacityDeque(d, 2*d->capacity);









}

void addBackDeque (struct deque *d, TYPE newValue) {
  if (d->size >= d->capacity) _setCapacityDeque(d, 2* d->capacity);









}
```

```
TYPE frontDeque (struct deque *d) {




}
TYPE backDeque (struct deque *d) {




}

void removeFrontDeque (struct deque *d) {




}
void removeBackDeque (struct deque *d) {




}
```

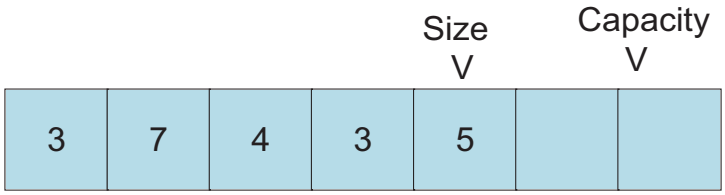
Worksheet 21: Building a Bag using a Dynamic Array

In preparation: Read Chapter 8 to learn more about the Bag data type. If you have not done so already, complete Worksheet 14 to learn about the basic features of the dynamic array.

Conceptual Bag interface
void add (TYPE newValue);
Boolean contains (TYPE testValue);
void remove (TYPE testValue);

The stack, queue and deque data abstractions are all characterized by maintaining values in the order they were inserted. In many situations, however, it is the values themselves, and not their time of insertion, that is of primary importance. The simplest data structure that is simply concerned with the values, and not their time of insertion, is the Bag. A conceptual definition of the Bag operations is shown at right. In subsequent lessons we will encounter several different implementation techniques for this abstraction. In this lesson we will explore how to create a bag using a dynamic array as the underlying storage area.

Recall that the dynamic array structure maintained three data fields. The first was a reference to an array of objects. The number of positions in this array, held in an integer data field, was termed the *capacity* of the container. The third value was an integer that represented the number of elements held in the container. This was termed the *size* of the collection. The size must always be smaller than or equal to the capacity.



As new elements are inserted, the size is increased. If the size reaches the capacity, then a new internal array is created with twice the capacity, and the values are copied from the old array into the new. In Worksheet14 you wrote a routine `_setCapacityDynArr`, to perform this operation.

To add an element to the dynamic array you can simply insert it at the end. This is exactly the same behavior as the function `addDynArray` you wrote in Worksheet 14.

The contains function is also relatively simple. It simply uses a loop to cycle over the index values, examining each element in turn. If it finds a value that matches the argument, it returns true. If it reaches the end of the collection without finding any value, it returns false. Because we want the container to be generalized, we define equality using a macro definition. This is similar to the symbolic constant trick we used to define the type TYPE. The macro is defined as follows:

```
# ifndef EQ
# define EQ(a, b) (a == b)
```

```
# endif
```

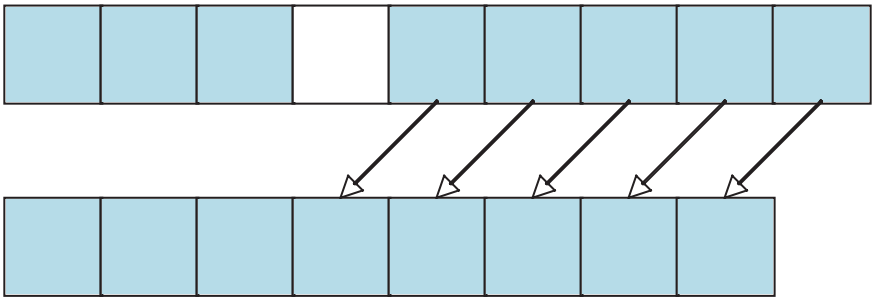
The ifndef preprocessor instruction allows the user to provide an alternative definition for the EQ function. If none is provided, the primitive equality operator will be used.

The remove function is the most complicated of the Bag abstraction. To simplify this task we will divide it into two distinct steps. The remove function, like the function contains, will loop over each position, examining the elements in the collection. If it finds one that matches the desired value, it will invoke a separate function, removeAt (from Worksheet 14), that removes the value held at a specific location.

```
void removeDynArr (struct DynArr * dy, TYPE test) {
   int i;
   for (i = 0; i < dy->size; i++) {
      if (EQ(test, dy->data[i])) { /* found it */
         _dynArrayRemoveAt(dy, i);
         return;
      }
   }
}
```

Notice two things about the remove function. First, if no matching element is found, the loop will terminate and the function return without making any change to the data. Second, once an element has been found, the function returns. This means that if there were two or more occurrences of the value that matched the test element, only the first would be removed.

The removeAt function takes as argument an index position in the array, and removes the element stored at that location. This is complicated by the fact that when the element is removed, all values stored at locations with higher index values must be “moved down”.



Once the values are moved down, the count must be decremented to indicate the size of the collection is decreased.

Based on these ideas, complete the following skeleton implementation of the bag functions for the dynamic array. You can use any of the functions you have previously written in earlier lessons.

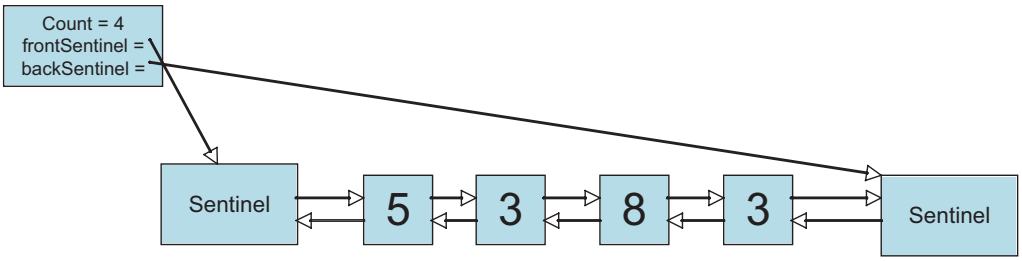
```
struct DynArr {
        TYPE * data;
        int size;
        int capacity;
};

            /* the following were written in earlier lessons */
void initDynArr (struct DynArr * da, int initCap);
void addDynArr(struct DynArr * da, TYPE e);

            /* remove was shown earlier, to use removeAt */
void removeDynArr (struct DynArr * da, TYPE test) {
  int i;
  for (i = 0; i < dy->size; i++) {
    if (EQ(test, dy->data[i])) { /* found it */
      _dynArrayRemoveAt(dy, i);
      return;
    }
  }
}

/* you must write the following */

int containsDynArr (struct DynArr * da, TYPE e) {









}
```

1. What should the removeAt method do if the index given as argument is not in range?
2. What is the algorithmic complexity of the method removeAt?
3. Given your answer to the previous question, what is the worst-case complexity of the method remove?
4. What are the algorithmic complexities of the operations add and contains?

Worksheet 22: Constructing a Bag using a Linked List

In Preparation: Read Chapter 8 to learn more about the Bag data abstraction. If you have not done so already, complete Worksheets 17 and 18 to learn about the basic features of the linked list.

In this lesson we continue developing the LinkedList data structure started in Worksheet 19. In the earlier worksheet you implemented operations to add and remove values from either the front or the back of the container. Recall that this implementation used a sentinel at both ends and double links. Because we want to quickly determine the number of elements in the collection, the implementation also maintained an integer data field named count, similar to the count field in the dynamic array bag.



Also recall that adding or removing elements is a problem that you have already solved. Adding a new value at either end was implemented using a more general internal function, termed addLink:

```
void _addBefore (struct LinkedList * lst, struct dlink * lnk, TYPE e);
```

Similarly removing a value, from either the front or the back, used the following function:

```
void _removeLink (struct linkedList * lst, struct dlink * lnk);
```

To create a bag we need three operations: add, contains, and remove. The add operation can simply add the new value to the front, and so is easy to write. The method contains must use a loop to cycle over the chain of links. Each element is tested against the argument, using the EQ macro. If any are equal, then the Boolean value true is returned. Otherwise, if the loop terminates without finding any matching element, the value False is returned.

The remove method uses a similar loop. However, this time, if a matching value is found, then the method removeLink is invoked. The method then terminates, without examining the rest of the collection.

Complete the implementation of the ListBag based on these ideas:

```
struct dlink {
   TYPE value;
   struct dlink * next;
   struct dlink * prev;
};

struct linkedList {
   struct dlink * frontSentinel;
   struct dlink * backSentinel;
   int size;
};

            /* the following functions were written in earlier lessons */
struct LinkedList *createLinkedList ();
void  initLinkedList (struct linkedList *q)
void deleteLinkedList (struct linkedList *lst);
void freeLinkedList (struct linkedList *q)
void _addBefore (struct linkedList * lst, struct dlink * lnk, TYPE e);
void _removeLink (struct linkedList * lst, struct dlink * lnk);

void addLinkedList (struct linkedList * lst, TYPE e)
   { _addLink(lst, lst->frontSentinel->next, e); }

/* you must write the following */

int containsLinkedList (struct linkedlist *lst, TYPE e) {









}
```

```
void removeLinkedList (struct linkedList *lst, TYPE e) {

}
```

1. What were the algorithmic complexities of the methods _addBefore and _removeLink that you wrote back in Chapter Q?
2. Given your answer to the previous question, what are the algorithmic complexities of the three principle Bag operations?

Worksheet 23: Introduction to the Iterator

In Preparation: Read Chapter 8 to learn more about the idea of encapsulation and the iterator. If you have not done it already, complete Worksheets 14 and 15 to learn more about adding and removing values from a dynamic array.

One of the primary design principles for collection classes is *encapsulation*. The internal details concerning how an implementation works are hidden behind a simple and easy to remember interface. To use a Bag, for example, all you need know is the basic operations are add, collect and remove. The inner workings of the implementation for the bag are effectively hidden.

When using collections a common requirement is the need to loop over all the elements in the collection, for example to print them to a window. Once again it is important that this process be performed without any knowledge of how the collection is represented in memory. For this person the conventional solution is to use a mechanism termed an *Iterator*.

```
/* conceptual interface */
Boolean  hasNext ( );
TYPE next ( );
void remove ( );
```

```
dynArrayIterator itr;
TYPE current;
dynArray data;
…
initDynArrayIterator (&data,&itr);
while (hasNextDynArray(&itr)) {
   current = nextDynArray(&itr);
   … /* do something with current */
}
```

Each collection provides an associated iterator structure. Functions are used to initialize and manipulate this iterator. These functions are used in combination to write a simple loop that will cycle over the values in the collection. For example, the code at left illustrates the use of our dynamic array iterator.

Notice that the iterator loop exposes nothing of the structure of the container class. The method remove is used to delete from the collection the value most recently returned by next. Calls on the functions hasNext and next must always be interleaved, as shown.

Note that an iterator is an object that is separate from the collection itself. The iterator is a *facilitator object*, one that provides access to the container values.

An iterator for the dynamic array will work by maintaining an index into the array representing the current location. This value is initially zero, and must be incremented by either the function hasNext or the function next? Think about this for a moment. Which one makes more sense? Then think carefully about the third function, **remove**. You must ensure that when the loop continues after a remove the next value following the removed element is not skipped. You can assume you have access to the `removeAtDynArray` function you wrote in Lesson 21.

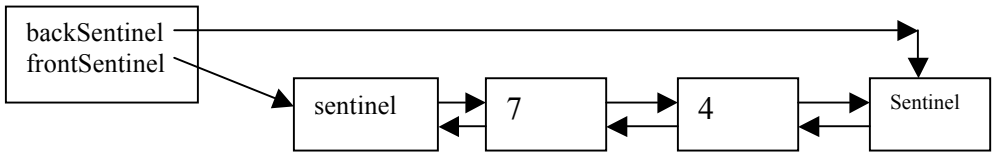
```
void removeAtDynArray (struct dynArray * dy, int index);
```

```
struct dynArrayIterator {
   struct dynArray * da;
   int currentIndex;
};

void initDynArrayIterator (struct dynArray *da, struct dynArrayIterator *itr) {




}

int hasNextDynArrayIterator (struct dynArrayIterator *itr) {




}

TYPE nextDynArrayIterator (struct dynArrayIterator *itr) {




}

void removeDynArrayIterator (struct dynArrayIterator *itr) {




}
```

Worksheet 24: LinkedList Iterator

In preparation: Read Chapter 8 to learn more about the concept of Iterators. If you have not done it already, you should complete Worksheets 17 and 23 to learn more about the basic features of the linked list, as well as the dynamic array iterator.

In this worksheet you will extend the design of the LinkedList abstraction you created in Worksheet 19, creating for it an iterator that uses the same interface as the dynamic array iterator you developed in Worksheet 23. Recall that for the LinkedList that double links are being used, and that there is both a starting and ending sentinel.



An iterator, you will remember, is defined by four functions. The first initializes the iterator, associating it with the container it will iterate over. The function hasNext returns true if there are more elements, false otherwise. The function next returns the current element. The function remove can be used to remove the element last returned by next.

```
/* conceptual interface */
 Boolean  hasNext ( );
 TYPE next ( );
 void remove ( );
```

The dynamic array iterator maintained an integer index into the container. The linked list iterator will do something similar. It will maintain a pointer to one of the links in the container. The functions hasNext and Next must move this pointer forward, so as to eventually reference every method.

Be careful with the remove method. You want to make sure that when the next iteration of the loop is performed the next element in sequence will be produced. However, the actual removal can be made easier using the function you wrote in Worksheet 19.

```
void _removeLink (struct linkedList * lst, struct dlink * lnk);
```

You may find it useful to draw a picture of the linked list, adding both the front and the back sentinels, to help you better understand how the fields in the linked list iterator should change as each element is returned.

```
struct linkedlistIterator {
   struct linkedList * lst;
   struct dlink * currentLink;
}

struct linkedListIterator * createLinkedListIterator (struct linkedList *lst) {
  struct linkedListIterator *newItr;
  newItr = malloc(sizeof(struct linkedListIterator));
  assert(newItr != 0);

   newItr->lst = lst;
   newItr->currentLink = lst->frontSentinel;
}

void _removeLink (struct linkedList * lst, struct dlink * lnk);

int hasNextLinkedListIterator (struct linkedListIterator *itr) {









}

TYPE nextLinkedListIterator (struct linkedListIterator *itr) {









}
```

/* think about the next one carefully. Draw a picture. what should itr->currentLink be pointing to after the remove operation ? */

```
void removeLinkedListIterator (struct linkedListIterator *itr) {




}
```

Worksheet 25: Bit Set

In Preparation: Read Chapter 8 to learn more about the Set abstraction.

A *bit set* is designed to represent a set of positive integers from a small range. Our implementation will set this range to be 0 to 255. Unlike the other set abstractions, a bit set doesn't need to save the elements, since one integer is indistinguishable from another of the same magnitude. Instead, the bit set can simply mark which elements have been inserted into the set, and use those marks to test whether or not a value is found in the collection. The advantage of marking is that several marks can be efficiently packed into a single integer. A 64-bit long integer, for example, can hold the marks that represent 64 elements. To accommodate more than 64 values an array of integers can be used.

To access a particular element an integer key must be converted into two separate integer quantities. The first is the *index* of the element in the array in which the mark is stored. This can be determined by simply dividing an integer by 64. The second is a position within the integer. This is formed by creating a *mask*, an integer in which only one binary position is set. This mask is created by shifting an integer 1 left by the appropriate number of elements.

There are two types of operations in the BitSet abstraction. The first takes an integer argument and modifies a single bit. A second class of operations take another bit set as argument, and performs set to set operations.

To turn a bit on (function set) a mask is *or*-ed with the data array. This is shown at the right. The operator | is used to form a bit-wise or.

```
Data:   10011001
Mask:   00100000
Result: 10111001
```

```
Data:   10011001
Mask:   00100000
Result: 00000000
```

To test whether a value is on (function get) the data array is *and*-ed with the mask. If the resulting value is nonzero, the bit was set. If the result is zero, the bit was not set.

To clear an element a logical *and* is performed with the *inverse* of the mask. The inverse retains all bit values, except for the single element being cleared.

```
Data:   10011001
Mask:   00100000
inverse:11011111
Result: 10011001
```

In addition to the ability to set and test individual bits, the bit set provides methods for forming the intersection and union of two bit sets. The union (operation or) can be formed using the bit-wise method or on the corresponding integer elements. The intersection (operation and) uses the bit-wise and. (Note that while the other set implementations we have described form a new set, the bit set computes the union and intersection in-place).

Using these ideas, complete the implementation of the bit-set abstraction.

```
struct BitSet {
   long int bits[4]; /* allows 256 positions */
};

int _bitSetIndex (int i) { return i / 64; }

long int _bitSetMask (int i) { return 1L << (i % 64); }

void bitSetSet (int indx) { bits[_bitSetIndex(indx)] |= _bitSetMask(indx); }

int bitSetGet (int indx) {









}

void bitSetClear (int indx) {









}

void and (Bitset right) {









}

void or (BitSet right) {









 }
```

Worksheet 26: Ordered Bag using a Sorted Array

In Preparation: Read Chapter 8 to learn more about the Bag data type and Chapter 9 to learn more about the advantages of ordered collections. If you have not done it previously, you should review Chapter 2 on the binary search algorithm, and do Worksheets14 and 16 to learn about the basic features of the dynamic array.

In an earlier lesson you encountered the *binary search* algorithm. The version shown below takes as argument the value being tested, and returns, in O(log n) steps, either the location at which the value is found, *or if it is not in the collection* the location the value can be inserted and still preserve order.

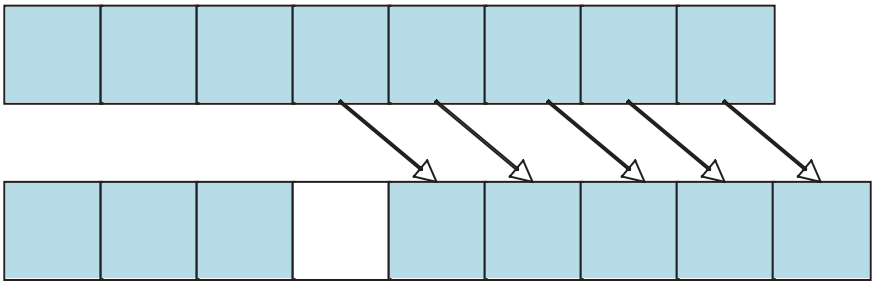
Notice that the value returned by this function need not be a legal index. If the test value is larger than all the elements in the array, the only position where an insertion could be performed and still preserve order is the next index at the top. Thus, the binary search algorithm might return a value equal to size, which is not a legal index.

```
int binarySearch (TYPE * data, int size, TYPE testValue) {
   int low = 0;
   int high = size;
   int mid;
   while (low < high) {
     mid = (low + high) / 2;
     if (LT(data[mid], testValue))
        low = mid + 1;
      else
         high = mid;
   }
   return low;
}
```

If we used a dynamic array as the underlying container, and if we kept the elements in sorted order, then we could use the binary search algorithm to perform a very rapid contains test. Simply call the binary search algorithm, and save the resulting index position. Test that the index is legal; if it is, then test the value of the element stored at the position. If it matches the test argument, then return true. Otherwise return false. Since the binary search algorithm is O(log n), and all other operations are constant time, this means that the contains test is O(log n), which is much faster than the bags developed in Worksheets 21 and 22.

Inserting a new value into the ordered array is not quite as easy. True, we can discover the position where the insertion should be made by invoking the binary search algorithm. But then what? Because the values are stored in a block, the problem is in many ways the opposite of the one you examined in Worksheet 21. Now, instead of moving values down to delete an entry, we must here move values up to make a “hole” into which the new element can be placed:

Worksheet 26: Ordered Bag using a Sorted Array Name:



As we did with remove, we will divide this into two steps. The add method will find the correct location at which to insert a value, then call another method that will insert an element at a given location:

```
void addOrderedArray (struct dyArray *da, TYPE newElement) {
  int index = binarySearch(da->data, da->size, newElement);
  addAtDynArray (da, index, newElement);
}
```

The method addAt must check that the size is less than the capacity, calling doubleCapacityDynArray if not, loop over the elements in the array in order to open up a hole for the new value, insert the element into the hole, and finally update the variable count so that it correctly reflects the number of values in the container.

```
void addAtDynArray (struct dyArray *da, int index, TYPE newElement) {
  int i;
  assert(index >= 0 && index <= da->size);
  if (da->size >= da->capacity)
          _doubleCapacityDynArray(da);
  …  /* you get to fill this in */
}
```

The method remove could use the same implementation as you developed in Chapter B. However, whereas before we used a linear search to find the position of the value to be deleted, we can here use a binary search. If the index returned by binary search is a legal position, then invoke the function `removeAtDynArray` that you wrote in Worksheet14 to remove the value at the indicated position.

On Your Own

Fill in the following Chart with the big-oh execution times for the simple unordered dynamic array bag (Lesson X), the linked list bag (Lesson x) and the ordered dynamic array bag (this worksheet).

	Dynamic Array Bag	LinkedListBag	Ordered Array Bag
Add	O(	O(	O(
Contains	O(	O(	O(
Remove	O(	O(	O(

Worksheet 26: Ordered Bag using a Sorted Array Name:

Which operations are faster in an unordered collection? Which operations are faster in an ordered collection? Based on this information, can you think of an example problem where it would be better to use an unordered collection? What about an example problem where it would be better to use an ordered collection?

Short Answers

1. What is the algorithmic complexity of the binary search algorithm?

2. Using your answer to the previous question, what is the algorithmic complex of the method contains for an OrderedArrayBag?

3. What is the algorithmic complexity of the method addAt?

4. Using your answer to the previous question, what is the algorithmic complexity of the method add for an OrderedArrayBag?

5. What is the complexity of the method removeAt? of the method remove?

```
/*Assume you have access to all existing dynArr functions */
/* _binarySearch is also included in dynArr.c now as an internal function! */
int _binarySearch(TYPE *data, int size, TYPE testValue);


/* These are the new functions to take advantage of the ordered dynamic array
   and available to the end user , so they will be declared in the .h file */

int binarySearchDynArray (struct dyArray * dy, TYPE testValue) {
   return _binarySearch (dy->data, dy->size, testValue); }

void addOrderedArray (struct dyArray *dy, TYPE newElement) {
   int index = _binarySearch(dy->data, dy->size, newElement);
   addAtDynArray (dy, index, newElement);  /* takes care of resize if necessary*/
}
```

Worksheet 26: Ordered Bag using a Sorted Array Name:

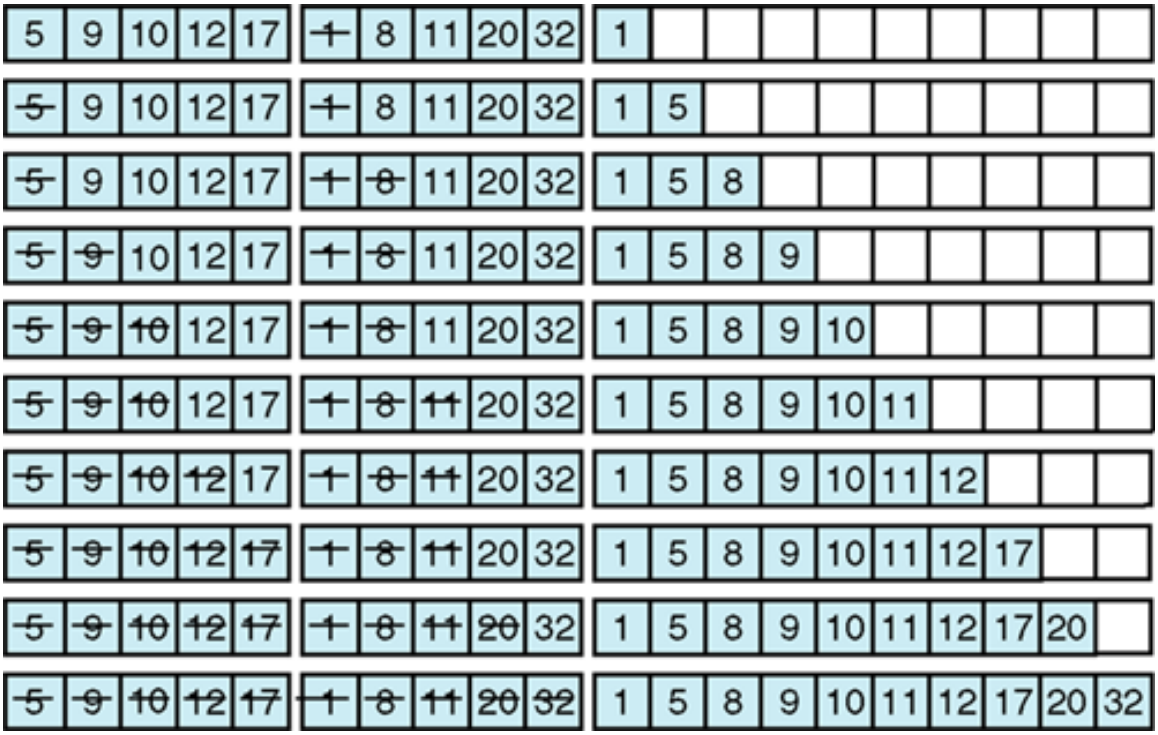
```
    /* you must complete the following */
int containsOrderedArray (struct dyArray *dy, TYPE testElement) {









}
 void removeOrderedArray (struct dyArray *dy, TYPE testElement) {









}
```

Worksheet 27: Sorted Array Sets

In Preparation: Read Chapter 8 to learn more about the Set data type. If you have not done so previously, you should do Lesson 26 to learn more about the basic features of the ordered array.

There are two important reasons to keep an array in order. One is so you can use binary search to perform a fast searching operation. We explored this in the previous worksheet when you developed the ordered dynamic array bag. But there is a second reason you might way to keep array elements in sequence. This is because two sorted arrays can be rapidly *merged* into a third.

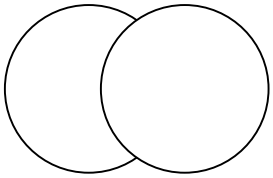
But the idea that two arrays can be rapidly merged into a third, as shown in the following picture. Simply walk down both collections in sequence, at each step copying the smallest element into a new array:



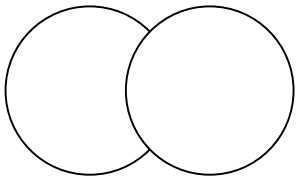
We can use this insight to make a much faster Set abstraction. Recall from Chapter 8 that the naive implementations of set union, intersection and difference are each $O(n^2)$. In this lesson you will make a container that is much faster. We will build on the ordered array abstraction you created in Lesson X. The add, contains and remove operations you implemented in Lesson X can be used as before.

Now consider the Set operations. The simplest operation to understand is intersection. To form an intersection simply walk through the two lists, forming a new list that will

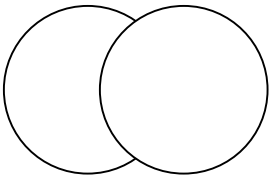
contain the intersection. If an element is found in both, add it the new intersection list. Otherwise if an element is found in only one of the collections, ignore it.



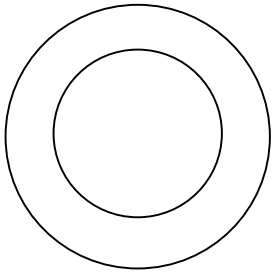
Only slightly more complex is set union. Remember that elements in a union must be unique. If a value is found in only one of the collections add it to the union. If it is found in both, add only one copy to the union. A complication to the union code is that when the first loop finishes you still need to copy the remainder of the elements from the other list to the collection.



A set difference is the elements from one set that are not found in the second. Forming this is similar to intersection.



Finally, to determine if a set is a subset of another walk down both lists in parallel. If you ever find a value that is in the first set but not in the second, then return false. If you finish looping over the first set and have not yet returned false, then return true.



```
        /* in each of the following, assume that the third argument is initially empty */
void orderedArrayIntersection (struct dyArray * dy1, struct dyArray * dy2, struct dyArray * dy3) {
  int i = 0;
  int j = 0;
  while (i < dy1->size && j < dy2->size) {
    if (LT(dy1->data[i], dy2->data[j])) {
      i++;
    } else if (EQ(dy1->data[i], dy2->data[j])) {
      dyArrayAdd(dy3, dy1->data[i]);
      i++;
      j++;
    } else {
      j++;
    }
  }
}

void orderedArrayUnion (struct dyArray * dy1, struct dyArray * dy2, struct dyArray * dy3) {
  int i = 0;
  int j = 0;
  while (i < dy1->size && j < dy2->size) {
    if (LT(dy1->data[i], dy2->data[j])) {


      i++;
    } else if (EQ(dy1->data[i], dy2->data[j])) {



      i++;
      j++;
    } else {


      j++;
    }
  }









}
```

```
void orderedArrayDifference (struct dyArray * dy1, struct dyArray * dy2, struct dyArray * dy3) {
  int i = 0;
  int j = 0;
  while (i < dy1->size && j < dy2->size) {
    if (LT(dy1->data[i], dy2->data[j])) {


      i++;
    } else if (EQ(dy1->data[i], dy2->data[j])) {



      i++;
      j++;
    } else {



      j++;
    }
  }
}


int orderedArraySubset (struct dyArray * dy1, struct dyArray * dy2) {
  int i = 0;
  int j = 0;
  while (i < dy1->size && j < dy2->size) {
    if (LT(dy1->data[i], dy2->data[j])) {



      i++;
    } else if (EQ(dy1->data[i], dy2->data[j])) {



      i++;
      j++;
    } else {



      j++;
    }
  }



}
```

Worksheet 28: Binary Search Trees

In Preparation: Read Chapter 8 to learn more about the Bag data type, and chapter 10 to learn more about the basic features of trees. If you have not done so already, read Worksheets 21 and 22 for alternative implementation of the Bag.

In this worksheet we will practice the concepts of using a Binary Search Tree for the Bag interface. For each of the following problems, draw the resulting Binary Search Tree.

1. Add the following numbers, in the order given to a binary search tree. 45, 67, 22, 100, 75, 13, 11, 64, 30
2. What is the height of the tree from #1? What is the height of the subtree rooted at the node holding the value 22? What is the depth of the node holding the value 22?
3. Add the following numbers, in the order given to a binary search tree. 3, 14, 15, 20, 25, 30, 33, 62, 200.
4. Is the tree from #3 balanced? Why not? What is the execution time required for searching for a value in this tree?
5. Add a new value, 145, to the tree from #1
6. Remove the value 67 from the tree from #1. What value did you replace it with and why?

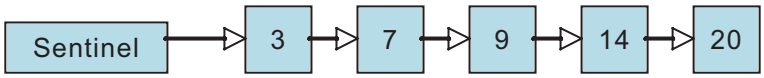
Worksheet 28b: Skip Lists

In Preparation: Read Chapter 8 to learn more about the Bag data type. If you have not done so already, you should do Lessons 17 and 18 to learn more about the basic features of the linked list.

We have seen two implementation techniques for the Bag, using a dynamic array and a linked list. However, because the order of insertion is unimportant for a Bag, a container is free to reorganize the elements to make various operations run more quickly. We will see in this lesson and future lessons several different data structures that make use of this principle.

The SkipList is a little more complex data structure than we have seen up to this point, and so we will spend more time in development and walking you through the implementation. To motivate the need for the skip list, consider that ordinary linked lists and vectors have fast (O(1)) addition of new elements, but a slow time for search and removal. A sorted vector has a fast O(log n) search time, but is still slow in addition of new elements and in removal. A skip list is fast O(log n) in all three times.

We begin the development of the skip list by first considering an simple ordered list, with a sentinel on the left, and single links:



To add a new value to such a list you find the correct location, then set the value. Similarly to see if the list contains an element you find the correct location, and see if it is what you want. And to remove an element, you find the correct location, and remove it. Each of these three has in common the idea of finding the location at which the operation will take place. We can generalize this by writing a common routine, named **slideRight**, that will move to the right as long as the next element is smaller than the value being considered. The implementation of each of the three basic operations is then a simple matter:

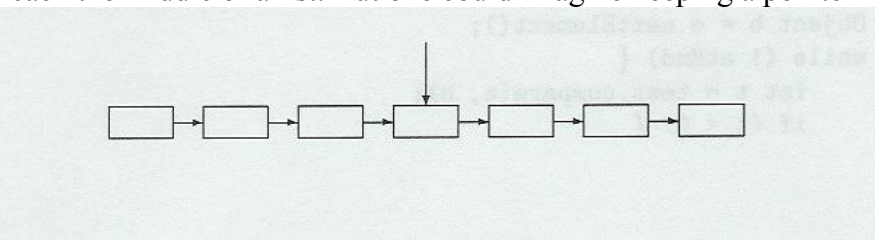
```
struct skipLink * slideRight (struct skipLink * current, TYPE d) {
   while ((current->next != 0) && LT(current->next->value, d))
      current = current->next;
   return current;
}

void add (struct skipList *sl, TYPE d) {
   struct skipLink * current = slideRight(sl->sentinel, d);
   struct skipLink * newLink = (struct skipLink *)
                        malloc(sizeof(struct skipLink));
   assert(newLink != 0);
   newLink->next = current->next;
   current->next = newLink;
   st->size++;
}
```

```
 int contains (struct skipList *sl, TYPE d) {
   struct skipLink *current = slideRight(sl->sentinel, d);
   if ((current->next != null) && EQ(current->next->value, d))  return 1;
   return 0;
}

void remove (struct skipList *sl, TYPE d) {
   struct skipLink *current = slideRight(sl->sentinel, d);
   if ((current->next != 0) && EQ(current->next->value, d)) {
      struct lnk = current->next;
      current->next = lnk->next;
      free(lnk);
      sl->size--;
   }
}
```

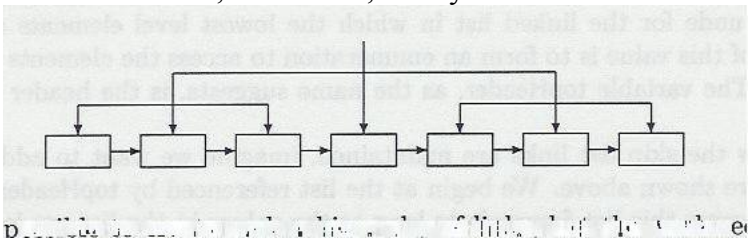
Try simulating some of the operations on this structure using the list shown until you understand what the function slideRight is doing. What happens when you insert the value 10? Search for 14? Insert 25? Remove the value 9?

By itself, this new data structure is not particularly useful or interesting. Each of the three basic operations still loop over the entire collection, and are therefore O(n), which is no better than an ordinary vector or linked list.

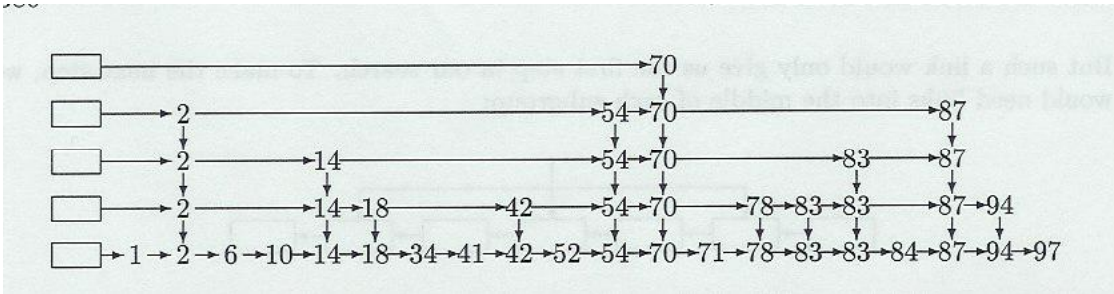
Why can't one do a binary search on a linked list? The answer is because you cannot easily reach the middle of a list. But one could imagine keeping a pointer into the middle of a list:



and then another, an another, until you have a tree of links



In theory this would work, but the effort and cost to maintain the pointers would almost certainly dwarf the gains in search time. But what if we didn't try to be precise, and instead maintained links *probablistically*? We could do this by maintaining a stack of ordered lists, and links that could point down to the lower link. We can arrange this so that each level has *approximately* half the links of the next lower. There would therefore be approximately log n levels for a list with n elements.



We are going to lead you through the implementation of the Skip List. The sentinel will be the topmost sentinel in the above picture. As you are developing the code you should simulate the execution of your routine on the picture shown.

The simplest operation is the contains test. The pseudo-code for this algorithm is as follows:

```
skipListContains (struct skipList *slst, TYPE testValue) {
   current = topmost sentinel;
   while (current is not null) {
      current = slide right (current, testValue)
      if the next element after current is not null AND the next elelment
after current is equal to testValue
            then return true
      current = current.down
   }
   return false;
}
```

Try tracing the execution of this routine as you search for the following values in the picture above: 70, 85, 1, 5, 6. Once you are convinced that the pseudo-code is correct, then rewrite it in C. The end of this worksheet as the skeleton of these functions.

The next operation is **remove**. This is similar to the contains test, with the additional problems that (a) a value can be on more than one list, and you need to remove them all, and (b) you need to reduce the element count only when removing from the bottommost level. In pseudo-code this is written as follows:

```
skipListRemove (struct skipList *slst, TYPE testValue) {
   current = sentinel;
   while (current is not null) {
      current = slide right (current, testValue)
      if the next element after current is not null and the next element
after current is equal to testValue then {
         remove the next node
         if current.down is null reduce the element count
      }
      current = current.down
   }
}
```

Notice that finding the value on one level does not immediately stop the operation - you must proceed all the way down to the bottom level. Try simulating the execution of this algorithm on the earlier picture, removing the following elements: 97, 78, 41, 85, 70.

The addition operation is the most complex of the three basic operations. The **add** operation is the place where probability comes into play. This operation is most easily written using a recursive helper function that takes a link and a value and may (probabilistically) either return a newly installed link, or a null value. The link that is the first argument is the position at which a value should be inserted (e.g., the result of calling **slideRight**). Imagine, for example, that we are inserting the value 15. The result of calling slideRight on the structure shown earlier would be simply the sentinel. The following describes the add operation in pseudo-code:

```
add (Link current, TYPE newValue) {
   Link downlink;  //temp skiplink pointer.
   if we are at the bottom (that is, current.down is null) then
      make a new link for the new value and insert it following current
      return the new link
   else {  // we are not at the bottm level
      downlink= add(slideRight(current.down, testValue), testValue)
      if (downLink is not null and coin flip is heads then {
         make new link for test value and install it after current
         return new link
      }
   }
   return null
}
```

Flipping a coin can be simulated by using random numbers. The method rand() returns a random integer value. If this value is even, assume the coin is heads. If off, assume tails.The topmost add operation performs an initial slide before calling the helper routine. If the helper has returned a link the coin is flipped one more time, and if heads a new level of list is created. This involves creating a new sentinel and installing a new link as the one element in the list. This is written in pseudo-code as follows:

```
add (SkipList l, TYPE newValue) {
   Link downLink = add(slideRight(sentinel, testValue), testValue);
   if (downLink is not null and coin flip is heads then {
      // make new level of list
      make new link for newValue, pointing down to downLink
      make new sentinel, pointing right to new link and down to existing
sentinel
   }
   add 1 to the counter for the collection size

}
```

You should try simulating the addition of new values to the picture shown earlier.

The following is the beginning of an implementation of this data abstraction. Much of the structure has been provided for you. Your task is to fill in the remaining parts. The task of

allocating and initializing a new link has been moved into a separate helper function named `createSkipLink(double, struct skipLink* nlnk, struct skipLink *dlnk).`

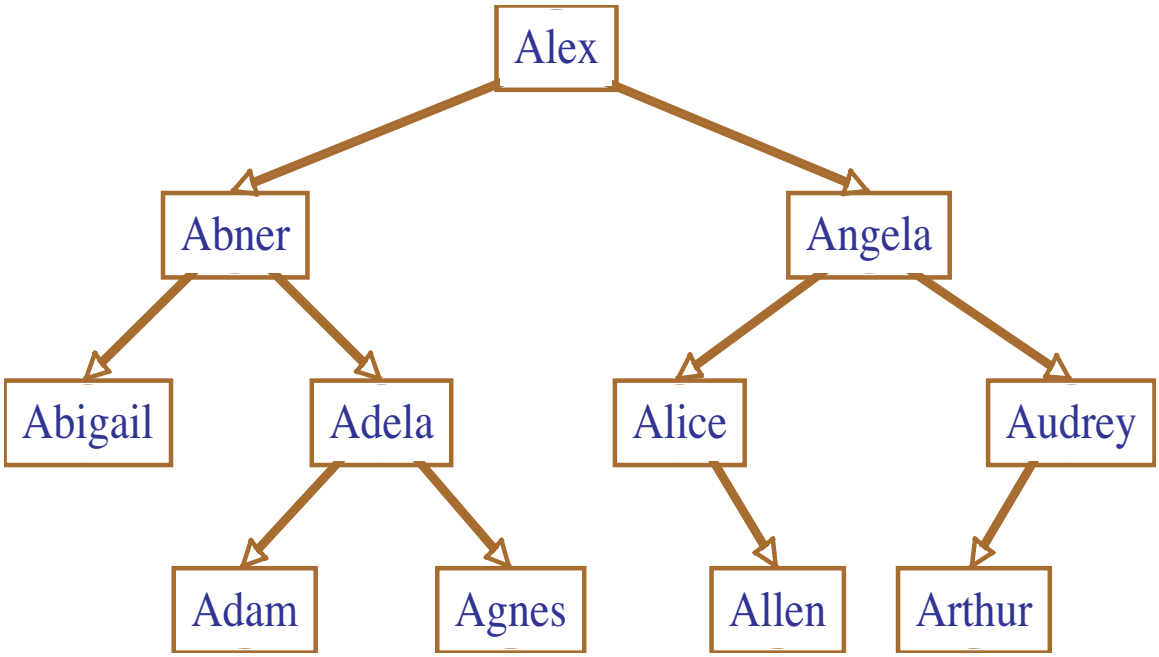
```
struct skipList {
        struct skipLink *topSentinel;
        int size;
};

struct skipLink {
        TYPE value;
        struct skipLink *next;
        struct skipLink *down;
};

struct skipLink * skipLinkCreate (TYPE d, struct skipLink *nlnk, struct
skipLink *dlnk) {
   /* create a new link and initialize it with the given values */









}


void skipListInit (struct skipList *slst) {
   /* initialize a skip list structure */




}

int skipFlip() { return rand() % 2; }

struct skipLink * skipLinkAdd (struct skipLink * current, TYPE d) {
   /* inner helper routine, add value at current location in list */









}

void skipListAdd (struct skipList *slst, TYPE d) {
```

```
}
int skipListContains (struct skipList *slst, TYPE d) {









}
void skipListRemove (struct skipList *slst, TYPE d) {




}
int skipListSize (struct skipList *slst) { return slst->size; }
```

Worksheet 29: Binary Search Trees

In Preparation: Read Chapter 8 to learn more about the Bag data type, and chapter 10 to learn more about the basic features of trees. If you have not done so already, read Worksheets 21 and 22 for alternative implementation of the Bag.

In this worksheet we will start to explore how to make a useful container class using the idea of a binary tree. A *binary search tree* is a binary tree that has the following additional property: for each node, the values in all descendants to the left of the node are less than the value of the node, and the values in all descendants to the right are greater than or equal. The following is an example binary search tree:



Notice that an inorder traversal of a BST will list the elements in sorted order. The most important feature of a binary search tree is that operations can be performed by walking the tree from the top (the root) to the bottom (the leaf). This means that a BST can be used to produce a fast **Bag** implementation. For example, suppose you find out if the name "Agnes" is found in the tree shown. You simply compare the value to the root (Alex). Since Agnes comes before Alex, you travel down the left child. Next you compare "Agnes" to "Abner". Since it is larger, you travel down the right. Finally you find a node that matches the value you are searching, and so you know it is in the collection. If you find a null pointer along the path, as you would if you were searching for "Sam", you would know the value was not in the collection.

Adding a value to a binary search tree is easy. You simply perform the same type of traversal as described above, and when you find a null value you insert a new node. Try inserting the values "Amelia". Then try inserting "Sam".

Insertion is most easily accomplished by writing a private internal function that takes a Node and a value, and returns the new tree in which the Node has been inserted. In pseudo-code this routine is similar to the following:

```
Node add (Node start, E newValue)
  if start is null then return a new Node with newValue
  otherwise if newValue is less than the value at start then
    set the left child to be the value returned by add(leftChild, newValue)
  otherwise set the right child to be add(rightChild, newValue)
  return the current node
```

Removal is the most complex of the basic Bag operations. The difficulty is that removing a node leaves a "hole". Imagine, for example, removing the value "Alex" from the tree shown. What value should be used in place of the removed element?

The answer is the *leftmost child of the right node*. This is because it is this value that is the smallest element in the right subtree. The leftmost child of a node is the value found by running through left child Nodes as far as possible. The leftmost child of the original tree shown above is "Abigail". The leftmost child of the right child of the node "Alex" is the node "Alice". It is a simple matter to write a routine to find the value of the leftmost child of a node. You should verify that in each case if you remove an element the value of the node can be replaced by the leftmost child of the right node without destroying the BST property.

A companion routine (removeLeftmost) is a function to return a tree with the leftmost child removed. Again, traverse the tree until the leftmost child is found. When found, return the right child (which could possibly be null). Otherwise make a recursive call and set the left child to the value returned by the recursive call, and return the current Node.

Armed with these two routines, the general remove operation can be described as follows. Again it makes sense to write it as a recursive routine that returns the new tree with the value removed.

```
Node remove (Node start, E testValue)
  if start.value is the value we seek
    decrease the value of dataSize
    if right child is null
      return left child
    otherwise
      replace value of node with leftmost child of right child
      set right child to be removeLeftmost(right child)
  otherwise if testValue is smaller than start.value
    set left child to remove (left child, testValue)
  otherwise
    set right child to remove (right child, testValue)
  return current node
```

Try executing this function on each of the values of the original binary search tree in turn, and verifying that the result is a valid binary search tree.

Using the approach described, complete the following implementation:

struct node {
  TYPE value;
  struct node * left;
  struct node * right;
};
	struct BinarySearchTree {
	  struct node *root;
	  int size;
	};

```
void initBST(struct BinarySearchTree *tree) { tree->size = 0; tree->root = 0: }

void addBST(struct BinarySearchTree *tree, TYPE newValue) {
  tree->root = _nodeAddBST(tree->root, newValue);  tree->size++; }

int sizeBST (struct binarySearchTree *tree) { return tree->size; }

struct node * _nodeAddBST (struct node *current, TYPE newValue) {









}
```

```
int containsBST (struct binarySearchTree *tree, TYPE d) {









}

void removeBST (struct binarySearchTree *tree, TYPE d) {
  if (containsBST(tree, d) {
    tree->root = _nodeRemoveBST(tree->root, d);
    tree->size--;
  }
}

TYPE _leftMostChild (struct node * current) {









}
```

```
struct node * _removeLeftmostChild (struct node *current) {




}

struct node * _nodeRemoveBST (struct node * current, TYPE d) {









}
```

On Your Own

1. What is the primary characteristic of a binary search tree?

2. Explain how the search for an element in a binary search tree is an example of the idea of divide and conquer.

3. Try inserting the values 1 to 10 in order into a BST. What is the height of the resulting tree?

4. Why is it important that a binary search tree remain reasonably balanced? What can happen if the tree becomes unbalanced?

5. What is the maximum height of a BST that contains 100 elements? What is the minimum height?

6. Explain why removing a value from a BST is more complicated than insertion.

7. Suppose you want to test our BST algorithms. What would be some good boundary value test cases?

8. Program a test driver for the BST algorithm and execute the operations using the test cases identified in the previous question.

9. The smallest element in a binary search tree is always found as the leftmost child of the root. Write a method getFirst to return this value, and a method removeFirst to modify the tree so as to remove this value.

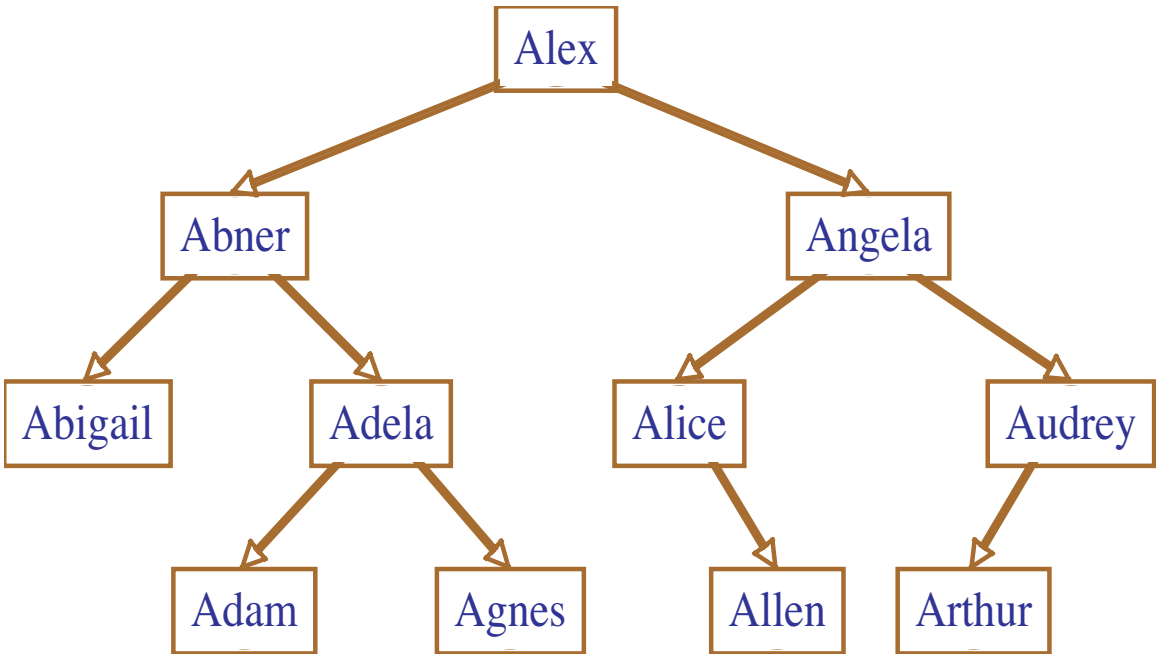
10. With the methods described in the previous question, it is easy to create a data structure that stores values in a BST and implements the Priority Queue interface. Show this implementation, and describe the algorithmic execution time for each of the Priority Queue operations.

11. Suppose you wanted to add the equals method to our BST class, where two trees are considered to be equal if they have the same elements. What is the complexity of your operation?

Worksheet 30: Binary Search Tree Iterator

In preparation: If you have not done so already, complete worksheet 29 to learn the basic features of the binary search tree.

To make the BST a useful container we need an iterator. This iterator must produce the elements in sequence. As we noted previously, this means making an in-order traversal of the tree. Examine the following tree and see if you can guess an algorithm that will do this. Remember that, in our implementation, nodes do not point back up to their parent nodes. How will you remember the path you have followed down the tree?



As you might have guessed, one way to do this is to have the iterator maintain an internal stack. This stack will represent the current path that has been traversed . Notice that the first element that our iterator should produce is the leftmost child of the root, and that furthermore all the nodes between the root and this point should be added to the stack. A useful routine for this purpose is slideLeft:

```
void slideLeft (Node n)
   while n is not null
      stack n
      n = left child of n
```

Using the slideLeft routine, verify that the following algorithm will produce the desired iterator traversal of the binary search tree. The remove portion of the iterator interface is more complex, and we will not consider it here.

initBSTIterator
   to initialize, create an empty stack

worksheet 30: Binary Search Tree Iterator Name:

```
int hasNextBSTIterator
  if stack is empty
    perform slideLeft on root
  otherwise
    let n be top of stack. Pop topmost element.
    slideLeft on right child of n
  return true if stack is not empty

double nextBSTIterator
  let n be top of stack. Return value of n
```

Can you characterize what elements are being held on the stack? How large can the stack grow?

Assuming that we have already defined a Stack that will hold pointers to nodes, show the implementation of the functions that implement the iterator algorithms described above.

```
struct BSTIterator {
  struct DynArr     *stk;
  struct BSTree   *tree;
};

void initBSTIterator (struct BSTree *tree, struct BSTIterator *itr) {




}

int hasNextBSTIterator (struct BSTIterator * itr) {
```

```
}

TYPE nextBSTIterator(struct BSTIterator *itr) {




}

void _slideLeft(struct Node *cur, struct BSTIterator *itr)
{
        While(cur->left != null)
          {
                pushDynArray(Itr->stk, cur->val);
                cur = cur->left;
          }
}
```

Worksheet: AVL Tree Practice

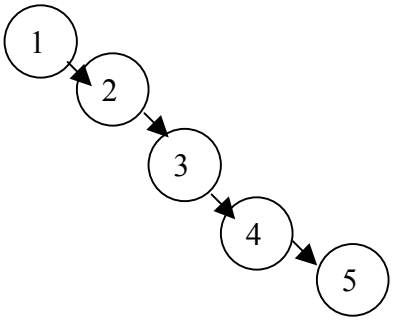
In Preparation: Read Chapters 8 and 10 on Bags and Trees, respectively. In you have not done so ready, do Worksheets 29 and 30 on Binary Search Trees.

Insert the following values, in the order that they are given into an AVL Tree.

1) 30,20,50,40,60,70
2) 50,22,80,70,75,73
3) 75, 70, 100, 60, 80, 105,77, 120
4) Take the tree from #3 and remove 60
5) Take the tree from #3. Remove 120. Remove 60.

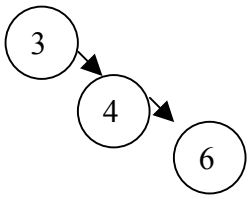
Worksheet 31: AVL Trees

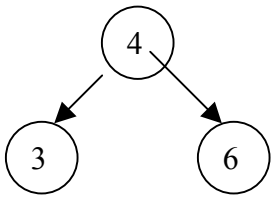
In Preparation: Read Chapters 8 and 10 on Bags and Trees, respectively. In you have not done so ready, do Worksheets 29 and 30 on Binary Search Trees.



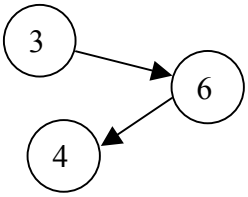
The time required to perform operations on a binary search tree is proportional to the length of the path from root to leaf. This isn't bad in a well-balanced tree. But nothing prevents a tree from becoming unbalanced. In the extreme, such as the tree shown on the right, the tree reduces to nothing more than a linked list, and the path from root to leaf is O(n).

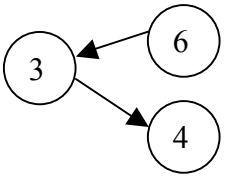
To preserve fast performance we need to ensure that the tree remains well balanced. One way to do this is to notice that the search tree property has some flexibility. Three nodes that are unbalanced can be restored by a *rotation*, making the right child into the new root, with the previous root as the new left child. Any existing left child of the old right child becomes the new right child of the new left child. The resulting tree is still a binary search tree, and has better balance.



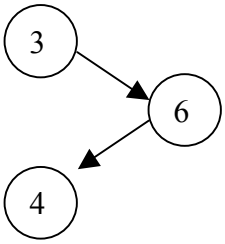


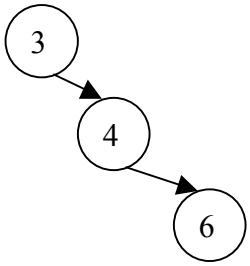
This is known as a *left rotation*. There is also a corresponding right rotation. There is one case where a simple rotation is not sufficient. Consider an unbalanced tree with a right child that itself has a left child. If we perform a rotation, the result is still unbalanced.

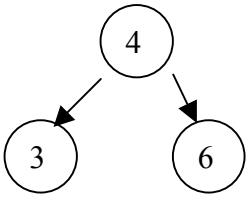




The solution is to first perform a rotation on the child, and then rotate the parent. This is termed a *double rotation*.







The data structure termed the AVL tree was designed using these ideas. The name honors the inventors of the data structure, the Russian mathematicians G. M. Adelson-Velskii and E.M. Landis.

In order to know when to perform a rotation, it is necessary to know the height of a node. We could calculate this amount, but that would slow the algorithm. Instead, we modify the Node so that each node keeps a record of its own height.

```
struct AVLnode {
   TYPE value;
   struct AVLnode *left;
   struct AVLnode *right;
   int height;
};
```

A function h(Node) will be useful to determine the height of a child node. Since leaf nodes have height zero, a value of -1 is returned for a null value. Using this, a function setHeight can be defined that will set the height value of a node, assuming that the height of the child nodes is known:

```
int _h(struct AVLnode * current)
       {if (current == 0) return -1;  return current->height;}

 void _setHeight (struct AVLnode * current) {
      int lch = h(current->left);
      int rch = h(current->right);
      if (lch < rch) current->height = 1 + rch;
      else current->height = 1 + lch;
   }
```

Armed with the height information, the AVL tree algorithms are now easy to describe. The addition and removal algorithms for the binary search tree are modified so that their very last step is to invoke a method **balance**:

```
struct AVLnode * _AVLnodeAdd (struct AVLnode* current, TYPE newValue) {
   struct AVLnode * newnode;
   if (current == 0) {
      newnode = (struct AVLnode *) malloc(sizeof(struct AVLnode));
      assert(newnode != 0);
      newnode->value = newValue;
      newnode->left = newnode->right = 0;
      return newnode;  //why don't we balance here ??
   } else if (LT(newValue, current->value))
      current->left = AVLnodeAdd(current->left, newValue);
   else current->right = AVLnodeAdd(current->right, newValue);
   return balance(current);    /* <-   NEW  the call on balance */
}
```

The function balance performs the rotations necessary to restore the balance in the tree. Let the *balance factor* be the difference in height between the right and left child trees. This is easily computed using a function. If the balance factor is more than 2, that is, if one subtree is more than two levels different in height from the other, then a rebalancing is performed. A check must be performed for double rotations, but again this is easy to determine using the balance factor function. Once the tree has been rebalanced the height is set by calling setHeight:

```
int _bf (struct AVLnode * current)
        { return h(current->right) - h(current->left); }

struct AVLnode * _balance (struct AVLnode * current) {
     int cbf = bf(current);
     if (cbf < -1) {
        if (bf(current->left) > 0) // double rotation
           current->left = rotateLeft(current->left);
        return rotateRight(current); // single rotation
     } else if (cbf > 1) {
        if (bf(current->right) < 0)
           current->right = rotateRight(current->right);
        return rotateLeft(current);
     }
     setHeight(current);
     return current;
  }
```

Since the balance function looks only at a node and its two children, the time necessary to perform rebalancing is proportional to the length of the path from root to leaf.

Insert the values 1 to 7 into an empty AVL tree and show the resulting tree after each step. Remember that rebalancing is performed bottom up after a new value has been inserted, and only if the difference in heights of the child trees are more than one.

Complete the implementation of the AVLtree abstraction by writing the methods to perform a left and right rotation. Both these methods should call setHeight on both the new interior node that has been changed and the new root. Other methods that are similar to those of the Binary Search Tree have been omitted:

```
struct AVLnode * _rotateLeft (struct AVLnode * current) {









}
```

```
struct AVLnode * _rotateRight (struct AVLnode * current) {









}
```

Finally, let's write the remove function for the AVL Tree. It is very similar to the **remove()** for a BST, however, you must be sure to balance when appropriate. We've provide the remove function, you must finish the implementation of the removeHelper. Assume you have access to removeLeftMost and LeftMost which we have already written for the BST.

```
void removeAVLTree(struct AVLTree *tree, TYPE val) {
   if (containsAVLTree(tree, val)) {
      tree->root = _removeNode(tree->root, val);
      tree->cnt--;
   }
}

TYPE _leftMost(struct AVLNode *cur) {
   while(cur->left != 0) {
      cur = cur->left;
```

```
   }
    return cur->val;
}


struct AVLNode *_removeLeftmost(struct AVLNode *cur) {
struct AVLNode *temp;

   if(cur->left != 0)
   {
      cur->left = _removeLeftmost(cur->left);
      return _balance(cur);
   }

   temp = cur->rght;
   free(cur);
   return temp;
}



struct AVLNode *_removeNode(struct AVLNode *cur, TYPE val) {









}
```

Worksheet 32: Tree Sort

In Preparation: Read chapter 10 on the Tree data type.

Both the Skip List and the AVL tree provide a very fast O(log n) execution time for all three of the fundamental Bag operations: addition, contains, and remove. They also maintain values in sorted order. If we add an iterator then we can loop over all the values, in order, in O(n) steps. This makes them a good general purpose container.

Here is an application you might not have immediately thought about: sorting. Recall some of the sorting algorithms we have seen. Insertion sort and selection sort are $O(n^2)$, although insertion sort can be linear if the input is already nearly sorted. Merge sort and quick sort are faster at O(n log n), although quick sort can be $O(n^2)$ in its worst case if the input distribution is particularly bad.

Consider the following sorting algorithm:

How to sort a array A

Step 1: copy each element from A into an AVL tree
Step 2: copy each element from the AVL Tree back into the array

Assuming the array has n elements, what is the algorithmic execution time for step 1? What is the algorithmic execution time for step 2? Recall that insertion sort and quick sort are examples of algorithms that can have very different execution times depending upon the distribution of input values. Is the execution time for this algorithm dependent in any way on the input distribution?

A sorted dynamic array bag also maintained elements in order. Why would this algorithm not work with that data structure? What would be the resulting algorithmic execution time if you tried to do this?

Can you think of any disadvantages of this algorithm?

The algorithm shown above is known as tree sort (because it is traditionally performed with AVL trees). Complete the following implementation of this algorithm. Note that you have two ways to form a loop. You can use an indexing loop, or an iterator loop. Which is easier for step 1? Which is easier for step 2?

```
void treeSort (TYPE *data, int n) { /* sort values in array data */
  AVLtree tree;


  AVLtreeInit (&tree);









}
```

Worksheet: Heap Practice

In Preparation: Read Chapter 11 on the Priority Queue ADT and Heaps

Insert the following values, in the order that they are given into a Min Heap. Show the tree after each insertion.

1) 30,20,50,10,5,70
2) Remove Min from the heap
3) Add 8 to the heap
4) Remove Min from the heap

Worksheet 33: Heaps and Priority Queues

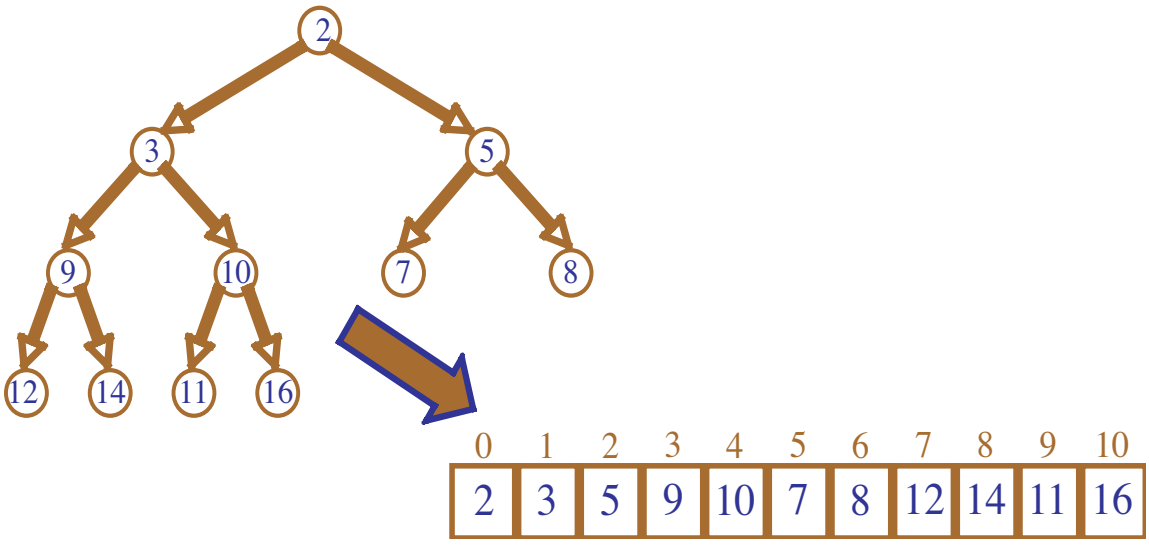
In preparation: Read Chapter 10 to learn more about trees, and chapter 11 to learn more about the Priority Queue data type. In you have not done so already, complete Worksheets 29 and 30 to explore the idea of a binary tree.

Recall that a *priority queue* is a collection designed to make it easy to find the element with highest priority. Such a data structure might be used, for example, in a hospital emergency room to schedule an operating table. The conceptual interface for this data type is shown at right.

```
Priority Queue Conceptual Interface

void add (TYPE newValue);
TYPE getFirst ();
void removeFirst ();
```

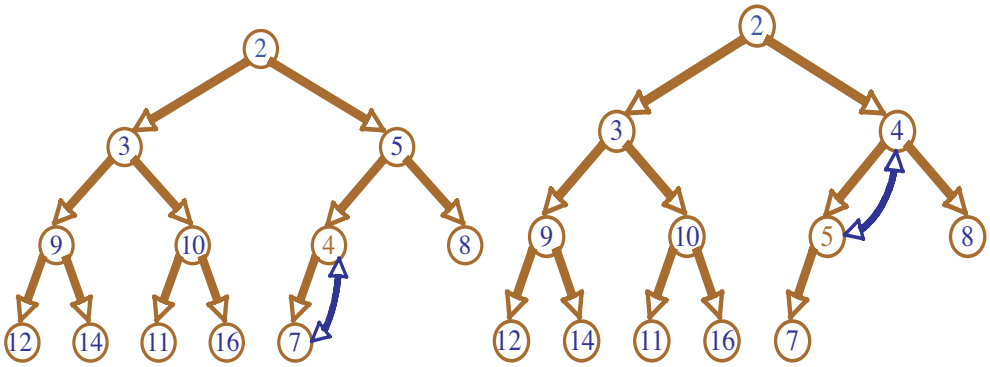
A *heap* is an efficient technique to implement a Priority Queue. A heap stores values in a complete binary tree in which the value of each node is less than or equal to each of its child nodes. This is termed the *heap order property*.

Notice that a heap is partially ordered, but not completely. In particular, the smallest element is always at the root. Although we will continue to think of the heap as a tree, the internal representation will be in a dynamic array. Recall that a complete binary tree can be efficiently stored as a dynamic array. The children of node i are found at positions 2i+1 and 2i+2, the parent at (i-1)/2.

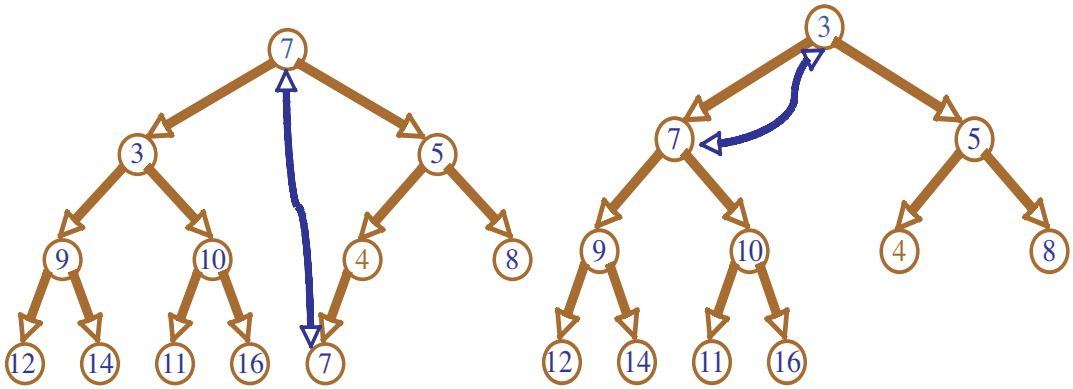


To insert a new value into a heap the value is first added to the end. This preserves the complete binary tree property, but not the heap ordering. To fix the ordering, the new value is *percolated* into position. It is compared to its parent node. If smaller, the node and the parent are exchanged. This continues until the root is reached, or the new value

finds its correct position. Because this process follows a path in a complete binary tree, it is O(log n).



The smallest value is always found at the root. But when this value is removed it leaves a "hole." Filling this hole with the last element in the heap restores the complete binary tree property, but not the heap order property. To restore the heap order the new value must *percolate down* into position.



We abstract the restoring heap property into a new routine termed adjustHeap:

```
void removeFirstHeap(struct dyArray *heap) {
    int last = sizeDynArray(heap)-1;
    assert (last != 0); /* make sure we have at least one element */
            /* Copy the last element to the first  position */
    putDynArray(heap, 0, getDynArray(heap, last));
    removeAtDynArray(heap, last);      /* Remove last element.*/
    adjustHeap(heap, last, 0);/* Rebuild heap */
  }
```

AdjustHeap can easily be written as a recursive routine:

```
void _adjustHeap (struct dyArray * heap, int max, int pos)
  int leftChild = 2*pos + 1; int rightChild = 2 * pos + 2;
  if (rightChild < max) { /* we have two children */
     get index of smallest child
     if value at pos > value of smallest child
        swap with smallest child,  call adjustHeap (max, index of smallest child)
 else if (leftchild < max) { /* we have one child */
     if value at pos > value of child
       swap with smallest child, call adjustHeap (max, index of left child)
  /* else no children, done */
```

The process follows a path in the complete tree, and hence is O(log n).

Two internal routines help in the completion of both these routines. The function *swap* , which you wrote in an earlier worksheet, will exchange two dynamic array positions. The function *indexSmallest* takes two index values, and returns the position of the smaller of the two.

Using these ideas, complete the implementation of the Heap data structure:

```
void swap (struct dyArray * v, int i, int j) {  /* swap elements i j */
     TYPE temp = getDynArray(v, i);
     putDynArray(v, i, getDynArray(v, j));
     putDynArray(v, j, temp);
  }

 int indexSmallest (struct dyArray * v, int i, int j) {
     /* return index of smallest element */
     if (LT(getDynArray(v, i), getDynArray(v, j))
        return i;
     return j;
  }

 TYPE getFirstHeapGet (struct dyArray *heap) {
     assert(sizeDynArray(heap) > 0);
     return getDynArray(heap, 0);
  }

void removeFirstHeap(struct dyArray *heap) {
     int last = sizeDynArray(heap)-1;
     assert (last != 0); /* make sure we have at least one element */
                 /* Copy the last element to the first  position */
     putDynArray(heap, 0, getDynArray(heap, last));
     removeAtDynArray(heap, last);       /* Remove last element.*/
     adjustHeap(heap, last, 0);/* Rebuild heap */
  }
```

```
void _adjustHeap (struct dyArray * heap, int max, int pos) {









}

void addHeap (struct dyArray * heap, TYPE newValue) {
        addDynArray(heap, newValue); /* adds to end – now need to adjust position */









}
```

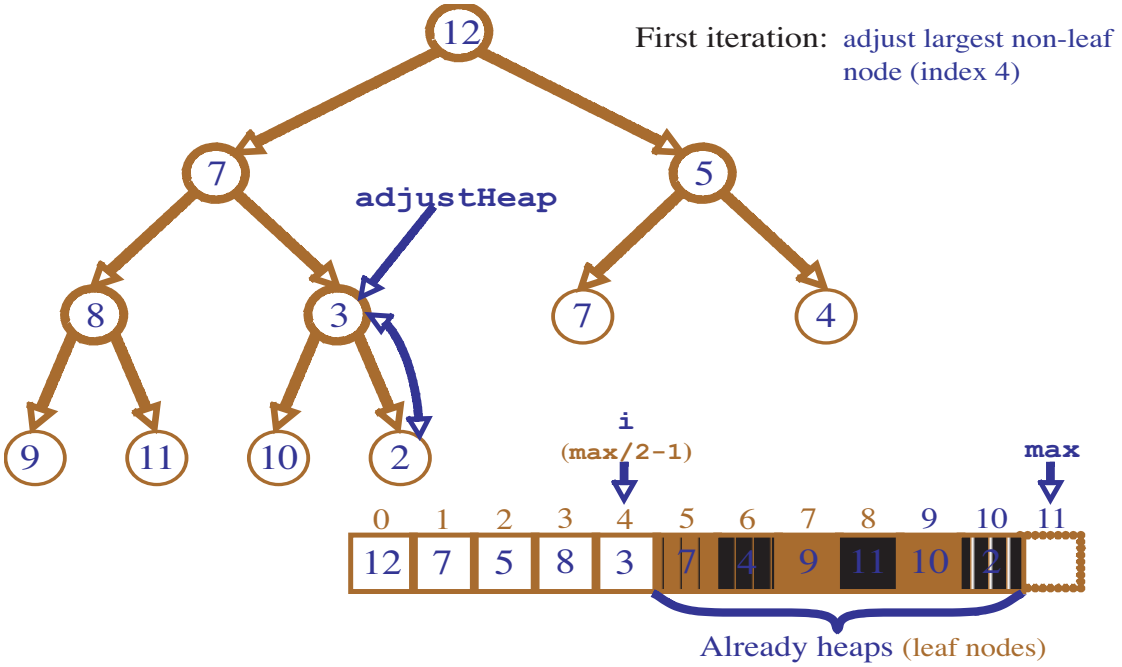
Worksheet 34: BuildHeap and Heap Sort

In preparation: If you have not done so already, you should complete Worksheet 33 to learn more about the heap data structure.

In some applications it is useful to initialize a Heap with an existing vector of values. The values are not assumed to be organized into a heap, and so a routine named buildHeap is invoked for this purpose.

```
void buildHeap (struct dyArray *heap) {
    int max = dyArraySize(heap);  int i;
    for ( i = max/2-1; i >= 0; i--)
      _adjustHeap(heap, max, i);
}
```

To understand the buildHeap algorithm, consider the following diagram:



All values indexed after max/2 are leaves, and are therefore already a heap. The first value that could potentially not be a heap is found at max/2. Walking backwards from this value until the root is reached eventually makes all nodes into a heap.

The heap data structure provides an elegant technique for sorting a vector. First form the vector into a heap. To sort the vector, the top of the heap (the smallest element) is swapped with the last element, and the size of the heap is reduced by 1 and readjusted. Repeat until all elements have been processed.

```
void heapsort (struct dyArray * v) { int i;
  buildHeap(v);
  for (i = dyArraySize(v) – 1; i > 0; i--)  {
    dyArraySwap(v, 0, i);
    _adjustHeap (v, i, 0);
  }
 }
```

worksheet 34: BuildHeap and Heap Sort Name:

Simulate execution of the Heap sort algorithm on the following values:

9 3 2 4 5 7 8 6 1 0

First make the values into a heap (the graphical representation is probably easier to work with than the vector form). Then repeatedly remove the smallest value, and rebuild the heap.

Worksheet 35: Skew Heaps

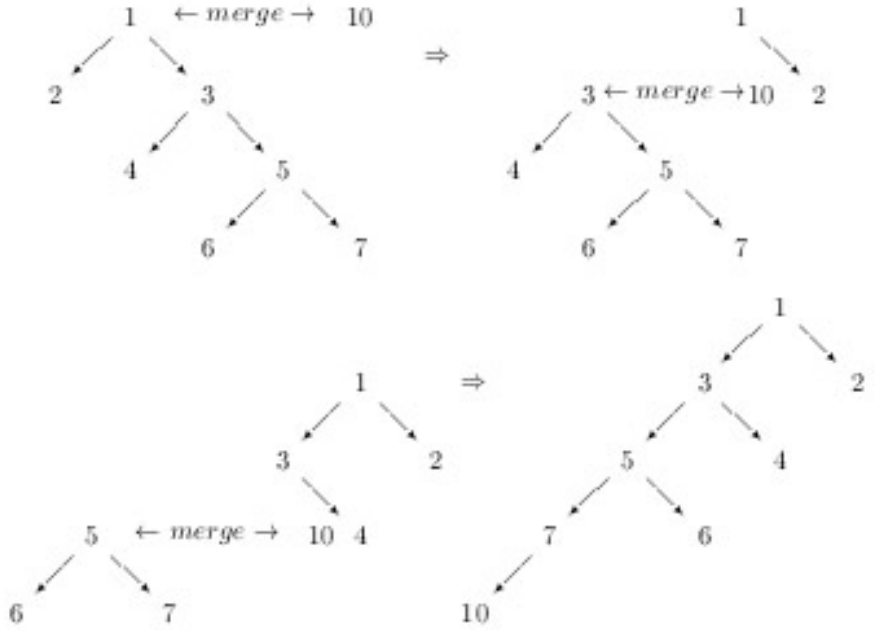
In Preparation: Read Chapter 11 to learn more about the Priority queue data type. If you have not done so already, complete worksheet 33 on the Heap data type.

For any node in a heap the relative order of the left and right children is unimportant. The *skew heap* builds on this property, and results in a very different organization from the traditional heap. The skew heap makes two observations. First, left and right children can always be exchanged with each other, since their order is unimportant. Second, both insertions and removals from a heap can be implemented as special cases of a more general task to merge two heaps into one.

It is easy to see how the remove is similar to a merge. When the smallest (that is, root) element is removed, you are left with two trees, namely the left and right child trees. To build a new heap you can simply merge the two. To view addition as a merge, consider the existing heap as one argument, and a tree with only the single new node as the second. Merge the two to produce the new heap.

A skew heap makes no attempt to guarantee the balance of its internal tree. Potentially, this means that a tree could become thin and unbalanced. But this is where the first observation is used. During the merge process the left and right children are systematically swapped. The result is that a thin and unbalanced tree cannot remain so. It can be shown (although the details are complex and not presented here) that amortized over time, each operation in a skew heap is no worst than O(log n).

The following illustrates the addition of the value 10 to an existing tree. Notice how a tree with a long right path becomes a tree with a long left path.



The merge algorithm for a skew heap can be described as follows:

```
Node merge (Node left, Node right)
  if (left is null) return right
  if (right is null) return left
  if (left child value < right child value) {
     Node temp = left.left;
     left.left = merge(left.right, right)
     left.right = temp
     return left;
  } else {
     Node temp = right.right
     right.right = merge(right.left, left)
     right.left = temp
     return right
  }
```

Complete the implementation of the SkewHeap based on these ideas. The only function you need to implement is merge.

```
Struct skewHeap {
  struct node * root;
};

void skewHeapInit (struct skewHeap * sk) { sk->root = 0; }

void skewHeapAdd (struct skewHeap *sk) {
  struct node *n = (struct node *) malloc(sizeof(struct node));
  assert(n != 0);  n->value = 0; n->leftchild = 0; n->rightchild = 0;
  s->root = skewHeapMerge(s->root, n);
}

EleType skewHeapGetFirst (struct skewHeap *sk) {
  assert (sk->root != 0); return sk->root->value; }

void skewHeapRemoveFirst (struct skewHeap *sk) {
  struct node * n = sk->root; free(n);
  sk->root = skewHeapMerge(n->leftchild, n->rightchild); }

stuct node * skewHeapMerge (struct node * left, struct node *right) {









}
```

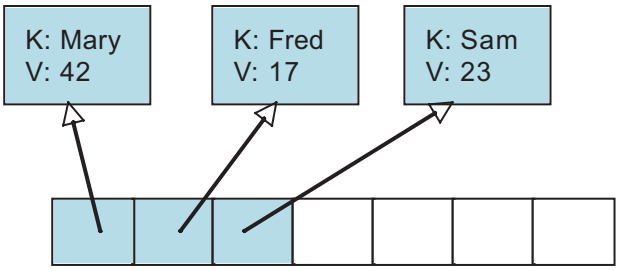
Worksheet 36: Dynamic Array Dictionary

In Preparation: Read Chapter 12 to learn more about the Dictionary data structure. If you have not done so already, complete worksheets 14 and 16 to learn more about the dynamic array.

In this lesson you will once again use a dynamic array as an underlying data container, only this time implementing a dictionary. A dictionary is an indexed collection class. This means that elements are always provided in a pair consisting of a key and a value. Just as we used a symbolic constant to define the TYPE for a vector, we will use two symbolic types to define both the KEYTYPE and the VALUETYPE for our vector. The interface file for this abstraction is shown on the next page. By default, we will use a character pointer for the key, and a double for the value. Instead of the EQ and LE macros used with the vector, we will have macros for LtKey and EqKey. The dictionary API is shown near the bottom of the page. When a value is inserted both the key and the value are provided. To search the collection the user provides a key, and the corresponding value is returned. A dictionary that associates words and definitions is a good mental model of a Map.

The idea behind the **DynamicArrayDictionary** is that internally elements in the dynamic array are stored as instances of struct **Association**. The internal struct **Association** stores a key and a value.

```
struct association {
   KEYTYPE key;
   VALUETYPE value;
};
```



Each element in the Dynamic Array is a pointer to an Association:

When searching to see if there is an entry with a given key, for example, each element of the dynamic array is examined in turn. The key is tested against the search key, and if it matches the element has been found.

A similar approach is used to delete a value. A loop is used to find the association with the key that matches the argument. Once the index of this association is found, the dynamic array **remove** operation is used to delete the value.

Elements in a dictionary must have unique keys. Within the method **put** one easy way to assure this is to first call **containsKey**, and if there is already an entry with the given key call remove to delete it. Then the new association is simply added to the end.

```
void putDynArrayDictionary (struct dyArray *data, KEYTYPE key, VALUETYPE val) {
        struct association * ap;
        if (containsKeyDynArrayDictionary(vec, key))
                removeKeyDynArrayDictionary (vec, key);
        ap = (struct association *) malloc(sizeof(struct association));
        assert(ap != 0);
        ap->key = key;
        ap->value = val;
        addDynArray(vec, ap);
}
```

To extract a value you use the function get. Rather than return the association element, this function takes as argument a pointer to a memory address where the value will be stored. This technique allows us to ignore an extra error check; if there is no key with the given value the function should do nothing at all.

```
void getDynArrayDictionary (struct dyArray *dy, KEYTYPE key, VALUETYPE *valptr);
```

Based on your implementation, fill in the following table with the algorithmic execution time for each operation:

int containsKey (KEYTYPE key)	O(
VALUETYPE get (KEYTYPE key)	O(
put (KEYTYPE key, VALUETYPE value)	O(
void remove (KEYTYPE key)	O(

```
# ifndef DYARRAYDICTH
# define DYARRAYDICTH

/*
  dynamic array dictionary interface file
*/

# ifndef KEYTYPE
# define KEYTYPE char *
# endif

# ifndef VALUETYPE
# define VALUETYPE double
# endif

struct association {
  KEYTYPE key;
  VALUETYPE value;
```

```
};

# define TYPE struct association *

# include "dynArray.h"

/* dictionary */
VALUETYPE getDynArrayDictionary (struct dynArray * dy, KEYTYPE key);
void putDynArrayDictionary (struct dynArray * dy, KEYTYPE key, VALUETYPE val);
int containsKeyDynArrayDictionary (struct dynArray * da, KEYTYPE key);
void removeKeyDynArrayDictionary (struct dynArray * da, KEYTYPE key);

# endif
```

```
# include "dynArrayDictionary.h"
# include "dynArrayDictionary.c"

/*finds and places the value associated with key in valptr */
VALUETYPE  getDynArrayDictionary (struct dynArr *da, KEYTYPE key) {









}

void putDynArrayDictionary (struct dynArr *da, KEYTYPE key, VALUETYPE val) {
        struct association * ap;
        if (containsKeyDynArrayDictionary(da, key))
                removeDynArrayDictionary(da, key);
        ap = (struct association *) malloc(sizeof(struct association));
        assert(ap != 0);
        ap->key = key;
        ap->value = val;
        addDynArray(da, ap);
}

int containsKeyDynArrayDictionary (struct dynArr *da, KEYTYPE key) {







}
```

```
void removeDynArrayDictionary (struct  dynArr *da, KEYTYPE key) {




}
```

Worksheet 37: Hash Tables (Open Address Hashing)

In preparation: Read Chapter 12 on dictionaries and hash tables.

In this chapter we will explore yet another technique that can be used to provide a very fast Bag abstraction. We have seen how containers such as the skip list and the AVL tree can reduce the time to perform operations from O(n) to O(log n). But can we do better? Would it be possible to create a container in which the average time to perform an operation was O(1)? The answer is both yes and no.

In chapter 12 you learned about the idea of hashing. To hash a value means to convert it into an integer index. This index is then used to access an array. Unfortunately, two different values may result in the same index. This is termed a collision. There are several different techniques used to deal with the problem of collisions. These will be explored in this and subsequent lessons.

The first technique you will explore is termed *open-address hashing*. Here all elements are stored in a single large table. Positions that are not yet filled are given a **null** value. An eight-element table using Amy's algorithm would look like the following:

0-aigy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina			Andy	Alessia	Alfred		Aspen

Notice that the table size is different, and so the index values are also different. The letters at the top show characters that hash into the indicated locations. If Anne now joins the club, we will find that the hash value (namely, 5) is the same as for Alfred. So to find a location to store the value Anne we *probe* for the next free location. This means to simply move forward, position by position, until an empty location is found. In this example the next free location is at position 6.

0-aigy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina			Andy	Alessia	Alfred	Anne	Aspen

No suppose Agnes wishes to join the club. Her hash value, 6, is already filled. The probe moves forward to the next position, and when the end of the array is reached it continues with the first element, eventually finding position 1:

0-aigy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina	Agnes		Andy	Alessia	Alfred	Anne	Aspen

Finally, suppose Alan wishes to join the club. He finds that his hash location, 0, is filled by Amina. The next free location is not until position 2:

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina	Agnes	Alan	Andy	Alessia	Alfred	Anne	Aspen

To see if a value is contained in a hash table the test value is first hashed. But just because the value is not found at the given location doesn't mean that it is not in the table. Think about searching the table above for the value Alan, for example. Instead, an unsuccessful test must continue to probe, moving forward until either the value is found or an empty location is encountered. (We will assume that our hash table contains pointers to elements, so that an empty position is indicated by an empty pointer).

Removing an element from an open hash table is problematic. We cannot simply replace the location with a null entry, as this might interfere with subsequent search operations. Imagine that we replaced Agnes with a null value in the table given above, and then once more performed a search for Alan. What would happen?

One solution to this problem is to not allow removals. This is the technique we will use. The second solution is to create a special type of marker termed a *tombstone*. A tombstone replaces a deleted value, can be replaced by another newly inserted value, but does not halt the search.

Complete the implementation of the HashTableBag based on these ideas. The initial size of the table is here fixed at 17. The data field tablesize holds the size of the table. The table should be resized if the load factor becomes larger than 0.75. Because the table contains pointers, the field in the struct is declared as a pointer to a pointer, using the double star notation. The variable count should represent the number of elements in the table. The macro HASH is used to compute the hash value. The calculation of the hash index is performed using long integers. The reason for this is explored in Chapter 12.

```
struct openHashTable {
  TYPE ** table;
  int tablesize;
  int count;
};

void initOpenHashTable (struct openHashTable * ht, int size) {
  int i;
  assert (size > 0);
  ht->table = (TYPE *) malloc(size * sizeof(TYPE *));
  assert(ht->table != 0);
  for (I = 0; I < size; i++)
          ht->table[i] = 0; /* initialize empty */
  ht->tablesize = size;
  ht->count = 0;
}

int sizeOpenHashTable (struct openHashTable *ht) { return ht->count; }

void addOpenHashTable (struct openHashTable * ht, TYPE  *newValue) {
   int idx;

   /* Make sure we have space and under the load factor threshold*/
   if ((ht->count / (double) ht->tablesize) > 0.75)
         _resizeOpenHashTable(ht);
   ht->count++;

  idx = HASH(newValue) % ht->tablesize;

  /* To be safe, use only positive arithmetic.  % may behave very differently on diff
     implementations or diff languages .  However, you can do the following to deal with a
     negative result from HASH  */

   if (idx < 0) idx += ht->tablesize;









}
```

```
void containsOpenHashTable (struct openHashTable *ht, TYPE  testValue) {

int idx;

idx = HASH(newValue) % ht->tablesize;
    if (idx < 0) idx += ht->tablesize;









  }

void _resizeOpenHashTable (struct openHashTable *ht) {









  }
```

Worksheet 38: Hash Tables using Buckets

In Preparation: Read Chapter 12 to learn more about hash tables. If you have not done so already, complete Worksheet 37 on open address hashing.

In the previous lesson you learned about the concept of hashing, and how it was used in an open address hash table. In this lesson you will explore a different approach to dealing with collisions, the idea of hash tables using buckets.

A hash table that uses buckets is really a combination of an array and a linked list. Each element in the array (the hash table) is a header for a linked list. All elements that hash into the same location will be stored in the list.

Each operation on the hash table divides into two steps. First, the element is hashed and the remainder taken after dividing by the table size. This yields a table index. Next, linked list indicated by the table index is examined. The algorithms for the latter are very similar to those used in the linked list. For example, to add a new element is simply the following:

```
void addHashTable (struct hashTable * ht, TYPE newValue) {
      // compute hash value to find the correct bucket
    int hash = HASH(newValue);
    int hashIndex = (int) (labs(hash) % ht.tablelength);
    struct link * newLink = (struct link *) malloc(sizeof(struct link));
    assert(newLink);
    newLink->value = newValue; newLink->next = ht->table[hashIndex];
    ht->table[hashIndex] = newLink; // add to bucket
    dataCount++; // Note: later might want to add resizing the table (below)
  }
```

The contains test is performed as a loop, but only on the linked list stored at the table index. The removal operation is the most complicated, since like the linked list it must modify the previous element. The easiest way to do this is to maintain a pointer to both the current element and to the previous element, as you did in Lesson 32. When the current element is found, the next pointer for the previous is modified.

As with open address hash tables, the load factor (l) is defined as the number of elements divided by the table size. In this structure the load factor can be larger than one, and represents the average number of elements stored in each list, assuming that the hash function distributes elements uniformly over all positions. Since the running time of the contains test and removal is proportional to the length of the list, they are O(l). Therefore the execution time for hash tables is fast only if the load factor remains small. A typical technique is to resize the table (doubling the size, as with the vector and the open address hash table) if the load factor becomes larger than 10.

Complete the implementation of the HashTable class based on these ideas.

```
struct hlink {
   TYPE value;
   struct hlink *next;
};
```

```
struct hashTable {
   struct hlink ** table;
   int tablesize;
   int count;
};
```

```
void initHashTable (struct hashTable * ht, int size) {









}
```

```
int sizeHashTable (struct hashTable * ht) { return ht->count; }
```

```
void addHashTable (struct hashTable *ht, TYPE newValue) {
        // compute hash value to find the correct bucket
     int hash = HASH(newValue);
     int hashIndex = (int) (labs(hash) % ht->tableSize);
     struct link * newLink = (struct hlink *) malloc(sizeof(struct hlink));
     assert(newLink);
     newLink->value = newValue; newLink->next = ht->table[hashIndex];
     ht->table[hashIndex] = newLink; /* add to bucket */
     ht->count++;
     if ((ht->count / (double) ht->tablesize) > 8.0) _resizeHashTable(ht);
  }
```

```
int containsHashTable (struct hashTable * ht, TYPE testElement) {









}
```

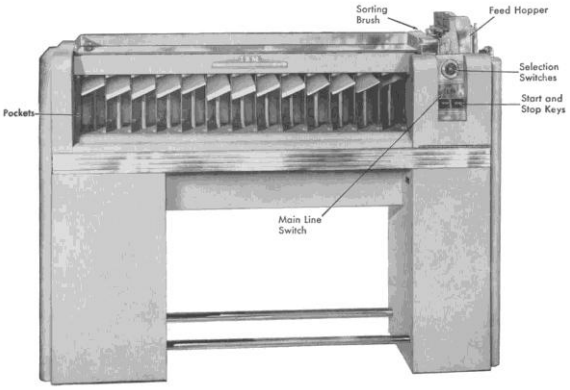
```
void removeHashTable (struct hashTable * ht, TYPE testElement) {









}

void resizeTable (struct hashTable *ht) {









}
```

Worksheet 39: Radix Sorting

In preparation: If you have not done so already, you should read chapter 12 to learn more about the concept of hashing.

Although less ubiquitous today than they once were, computer punch cards were for many years commonly used to store information both for computer use and for other business applications. A punch card was made out of stiff paper, and holes in the card indicated a specific letter. A card could typically hold 80 columns of information.

Because it was far too easy to drop a tray of cards and get them out of order, a convention developed to place a *sequencing number* somewhere on the card, often in the last eight columns. A common task was to order a collection of cards according to their sequencing number.



A machine called a *sorter* would divide a deck of cards into buckets, based on their value in **one** specific column. This would seem to be of little use, however the technique described in this lesson, radix sorting, shows how this one column sorting could be extended to sort on a larger range of numbers.

Suppose, for example, that the sequencing numbers appeared in columns 78, 79 and 80. The deck would first be sorted on column 80. The person doing the sorting would then, by hand, gather the resulting decks together and sort them once again, this time on column 79. Gathering together for a third time, the deck would be sorted on column 78. After the third sort, the deck would be completely ordered.

Radix Sorting

Radix sorting is not a general purpose sorting technique. It works only on positive integer values. But in this limited domain of application it is very fast. It is also interesting because of its historical import (see above) and because of its novel use of both hash tables and queues.

Imagine a table of ten queues that can hold integer values. The elements are successively ordered on digit positions, from right to left. This is accomplished by copying the elements into the queues, where the index for the bucket is given by the position of the digit being sorted. Once all digit positions have been examined, the collection will be completely sorted.

To illustrate, assume the list to be sorted is as follows:

624 762 852 426 197 987 269 146 415 301 730 78 593

The following table shows the sequences of elements found in each bucket during the four steps involved in sorting the list. During pass 1 the one's place digits are ordered. Each value that ends in the same digit is placed into the same bucket. During pass 2, the ten's place digits are ordered. However, because we are using a queue the relative positions set by the earlier pass are maintained. Again all elements that are the same in the 10's place are placed in the same bucket. On pass 3, the hundred's place digits are ordered. After three passes, the result is a completely ordered collection.

Bucket	Pass 1	Pass 2	Pass 3			
0	730	301	78			
1	301	415	146,197			
2	762, 852	624, 426	269			
3	593	730	301			
4	624	146	415, 426			
5	415	852	593			
6	426, 146	762, 269	624			
7	197, 987	78	730, 762			
8	78	987	852			
9	269	593, 197	987			

As a function of d, the number of digits in each number, and n, the number of values being sorted, what is the algorithmic complexity of radix sorting?

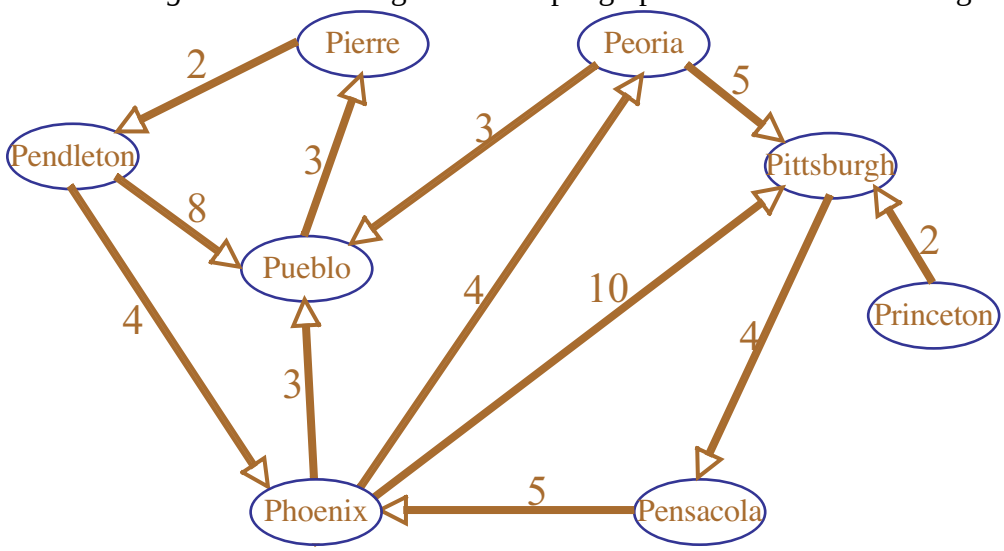
How does this compare to merge sort, quick sort, or any of the other fast sorting algorithms we have seen?

In the space in the right of the table above perform a radix sort on the following values, showing the result after each pass: 742 247 391 382 616 872 453 925 732 142 562.

Worksheet 40: Graph Representations

In preparation: Reach Chapter 13 to learn more about graphs and graph algorithms.

As you learned in Chapter 13, a graph is composed of *vertices*, connected by *edges*. Vertices can carry information, often just a name. Edges can either be directed, which indicates an asymmetric relationship, or undirected. In addition, edges can carry a value, termed a *weight*. The following is an example graph with directed and weighted edges.



There are two common ways of representing a graph as a data structure. These are as an *adjacency matrix*, and as an *edge list*. To form an *adjacency matrix* the vertices are assigned a number. For example, we could number the vertices of the graph above by listing the cities in alphabetical order: 0-Pendleton, 1-Pensacola, 2-Peoria, 3-Phoenix, 4-Pierre, 5-Pittsburgh, 6-Princeton, and 7-Pueblo. An 8 by 8 matrix is then constructed. Each (i,j) entry describes the association between vertex i and vertex j. In an unweighted graph this entry is either 0 if there is no connection, or 1 if there is a connection between the two cities. In a weighted matrix the value is the weight on the arc linking the two cities, or the value infinity if there is no connection. In the space at right enter the edge list representation for the graph shown above. An adjacency matrix requires $O(v^2)$ space, where v is the number of vertices. This is true regardless of the number of edges.

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								

The alternative representation, the *edge list*, stores only the edges of a graph. This is advantageous if the graph is relatively sparse. Fundamentally, the edge list uses a map indexed by the vertex (or vertex label). For an unweighted graph the value is a set of vertices that represent the neighbors of the key vertex. In a weighted graph the value is itself represented by another map. In this map the key is the neighboring vertex, and the

value is the weight of the arc that connects the two vertices. Complete the following edge list representation of the earlier graph. One entry has been made for you already.

Pendleton: {Pueblo: 8, Phoenix: 4}

Pensacola:

Peoria:

Phoenix:

Pierre:

Pittsburgh:

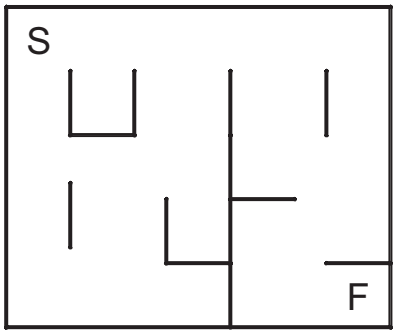
Princeton:

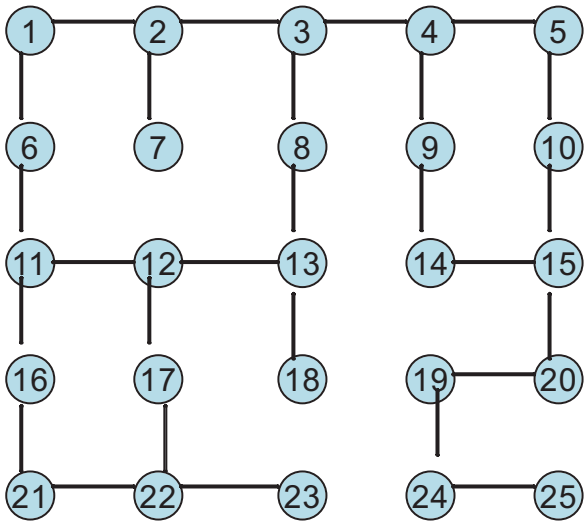
Pueblo:

Let v represent the number of vertices in a graph, and e the number of edges. Assuming that e > v, the edge list representation requires O(e) storage.

Worksheet 41: Depth First and Breadth First Search

Many different types of data can be represented by a graph. For example, consider the maze shown at right. We can encode the maze as an undirected graph by assigning each cell a value, and forming an arc when you can move from one cell to the next. The resulting graph is as follows:





The *reachablility problem* asks what cells (vertices) can be reached starting from the initial vertex. Of course, as anybody who has tried traversing a maze knows, it is not a simple matter of walking a fixed path, since you frequently have a choice of two or more unexplored alternatives. You will often find yourself in a dead-end, and must backtrack to investigate another possibility.

This problem can be expressed in data structure form as the following. You are given a graph and an initial vertex. The result you seek is a *set* of reachable vertices. To discover this you will use another container of vertices known to be reachable but possibly not yet explored. The reachability algorithm can be expressed in pseudo-code as follows

```
findReachable (graph g, vertex start) {
  create a set of reachable vertices, initially empty. call this r.
  create a container for vertices known to be reachable. call this c
  add start vertex to container c
  while the container c is not empty {
    remove first entry from the container c, assign to v
    if v is not already in the set of reachable vertices r {
      add v to the reachable set r
      add the neighbors of v, not already reachable, to the container c
    }
  }
  return r
}
```

What is interesting about this algorithm is that the container c can be either a stack or a queue. The resulting search will be very different depending upon the data structure is selected. If a stack is used, it is termed *depth-first search*. If a queue is used, it is termed

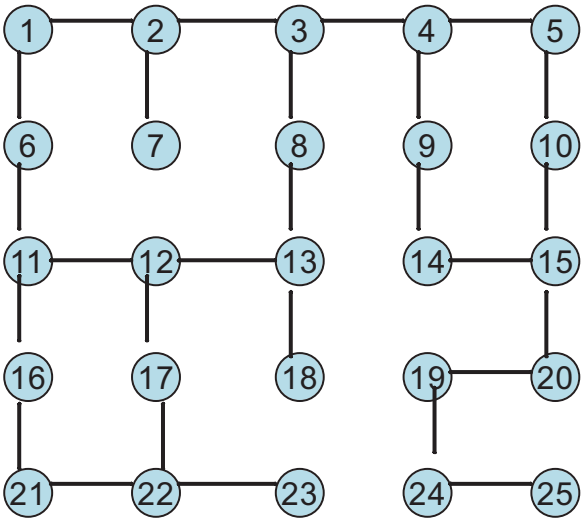
breadth-first search. To explore this, simulate the algorithm on the graph given, and record the vertices in r in the order that they are placed into the collection. For the purpose of being able to reproduce a trace, please push the neighbors onto the stack (or add to the end of the queue), in clockwise order starting at the node to directly to the north.

Depth first search [First 12 iterations] (stack version)

Iteration	Stack(T—B)	Reachable
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

Breadth first search [first 14 iterations] (queue version)

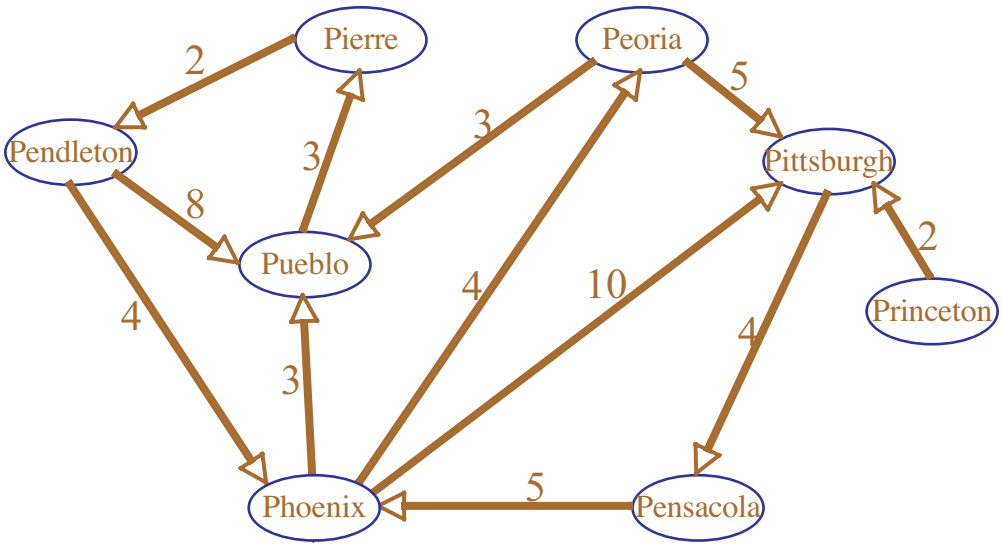
Iteration	Queue (F---B)	Reachable
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		



Using your finger, trace the sequence of vertices in the graph in the order that they are listed in r. Notice that the stack version moves in more of a continuous line, while the queue version seems to jump all over the graph. One way to visualize the two is that a depth first search is like a person walking through the maze. As long as they can move forward, they continue. It is only when they reach a dead-end that they move back to a point where there was a previous choice, selecting a different alternative. A breadth-first search, on the other hand, is like pouring a bottle of ink on the initial vertex. The ink moves from cell to cell, uniformly moving in all possible directions at the same time. Eventually the ink will find all reachable locations.

Worksheet 42: Dijkstra's Algorithm

In worksheet 41 you investigated an algorithm to solve the problem of graph reachablility. The exact same algorithm would perform two different types of search, either depth-first or breadth-first search, depending upon whether a stack or a queue was used to hold intermediate location. When dealing with weighted and directed graphs, there is a third possibility. A common question in such graphs is not whether a given vertex is reachable, but what is the lowest cost (that is, sum of arc weights) to reach the vertex. This problem can be solved by using the same algorithm, storing intermediate values in a *priority queue*, where the priority is given by the cost to reach the vertex. This is known as Dijkstra's algorithm (in honor of the computer scientist who first discovered it). Imagine a graph such as the following:



We want to find the shortest distance to various cities starting from Pierre. The algorithm uses an internal priority queue of distance/city pairs. This queue is organized so that the value with smallest distance is at the top of the queue. Initially the queue contains the starting city and distance zero. The map of reachable cities is initially zero. As a city if pulled from the pqueue, if it is already known to be reachable it is ignored, just as before. Otherwise, it is placed into the collection of reachable cities, and the neighbors of the new city are placed into the pqueue, adding the distance to the city and the distance to the new neighbor.

The following table shows the values of the priority queue at various stages. Note that we have shown only the latest node added to the Reachable collection at each iteration.

Iteration	Pqueue	Reachable with Costs
0	Pierre: 0	{}
1	Pendleton: 2	Pierre: 0
2	Phoenix: 6, Pueblo: 10	Pendleton: 2
3	Pueblo: 9, Peoria 10, Pueblo: 10, Pittsburgh: 16	Phoenix: 6
4	Peoria: 10, Pueblo: 10, Pittsburgh: 16	Pueblo: 9
5	Pueblo: 10, Pittsburgh: 15, Pittsburgh: 16	Peoria: 10

6	Pittsburgh:15, Pittsburgh: 16	--
7	Pittsburgh: 16, Pensacola: 19	Pittsburgh: 15
8	Pensacold:19	--
9	{}	Pensacola: 19

Notice how duplicates are removed only when pulled from the pqueue.

Simulate Dijkstra's algorithm, only this time using Pensacola as the starting city: