Data Structures Notes

Patrick McDowell
May 22, 2012
Patrick McDowell, Kuo-Pao Yang
Summer 2022

Table of Contents

Data Structures Notes 1
Table of Contents 2
Preface 5
Chapter 1. Introduction 6
1.1 What is a data type? 6
1.2 What about interpretation of data? 7
1.3 Standard types 8
1.4 Word alignment 8
1.5 What is a data structure and why are they useful? 11
1.6 Exercises 12
Chapter 2. Arrays, Memory Arrangement, and Pointers 14
2.1 Arrays in Memory 17
2.2 Pointers and Pointer Arithmetic 19
2.21 General Rules for Working With Pointers 20
Example 1 - Basic Declaration and use of Pointers 21
Example 2 - Pass by Reference in C 22
Example 3 - Reference Variable in C++ 23
Example 4 - One dimensional Array Traversal 24
Example 4 : Two dimensional Array Traversal 26
Example 5 - Using pointers to exchange rows in a 2-D array; 27
2.3 Circular Arrays 28
2.4 Exercises 31
Chapter 3. Stacks and Queues 33
3.1 Implementation Fundamentals 34
3.2 Basic methods for a Stack 35
3.3 Pseudo Code for Stack Methods 35
3.4 Pseudo Code for Queue Methods 36
3.5 Applications 37
3.51 Parentheses/Equation Validation 38
3.52 Postfix Expression Evaluation 40
3.52-1 A Quick Word About Software Development 45
3.53 Infix to Postfix Expression Conversion 45
3.5 Exercises 49
Chapter 4. Recursion 51
4.1 Summary 54
4.2 Exercises 55
Chapter 5. Complexity 56
5.1 Big O Notation 58
5.3 More Time Complexity: Log base two 59
5.4 Efficiency Calculations and Comparisons 63
5.5 A Quick Guide to Algorithm Analysis 65
5.5.1 Single for-loop, order O(N) 65
5.5.2 Double Nested for-loop $O(N^2)$, but not always 66

5.5.3 Nested for-loops, order O($N^{\text{number of nested loops}}$) 70
5.5.4 Recursive Calls 70
5.6 Exercises 74
Chapter 6: Linked Lists 76
6.1 Fundamental Methods for Lists 78
6.1.1 Initialization Method - init() 79
6.1.2 Node Creation - makeNode() 79
6.1.3 Finding the Last Node of a Linked List - findTail() 80
6.1.4 Adding Nodes to the End of the List - addAtEndOfList() 80
6.1.5 Adding Nodes in the Interior of a List - addAfter() 83
6.1.6 Deleting Nodes in a List - deleteAfter() 84
6.2 Fundamental List Applications 85
6.2.1 Fundamental Exercises 85
6.3 Building an Ordered List 86
6.3.1 Ordered List Insert Function 87
6.4 More Complex Lists, with Applications 88
6.4.1 Building a Multi-Indexed List 88
6.4.2 Circular Linked Lists 92
6.4.3 Doubly Linked Lists 93
6.5 Complexity Issues 95
6.6 Exercises 95
Chapter 7. Trees 100
7.1 Node Structure and Basic Tree Methods 103
7.1.2 Fundamental Tree Methods 105
7.1.2.1 Initialization Method - init() 106
7.1.2.2 Node Creation - makeTreeNode() 106
7.1.2.3 Adding Nodes to the Tree - setLeft(), setRight() 107
7.1.2.4 Building the Binary Tree 108
7.1.2.2 Showing the contents the Tree - Inorder Traversal - inOrder() 111
7.1.2.3 Iterative Traversal of a Tree 114
7.2 Complexity Issues 115
7.3 Exercises 116
Chapter 8. Graphs 117
8.1 Definitions and Concepts 117
8.1.1 Graphs and the Adjacency Matrix 118
8.2 Graph Traversals 119
8.2.1 Depth First Search 120
8.2.3 Breadth First Search 125
8.3 Path Problems 127
8.3.1 Transitive Closure 128
8.3.2 Dijkstra's Single Point Shortest Path Algorithm 131
8.4 Exercises 135
Chapter 9. Hash Tables and Hashing 136
9.1 Summary of Definitions and Concepts 139
Hash Function 139
Collisions and Collision Management 139

Effective Use of Memory 139
Criteria for a Good Hash Function 140
9.2 Basic methods 140
9.3 Exercises 141
Chapter 10. Heaps 146
10.1 Array Implementation of the Heap 147
10.2 Basic Methods 148
10.21 buildHeap() 148
10.22 heapify() 149
10.23 heapSort() 149
10.24 reheapify() 149
Chapter 11. Sorting 151
11.1 Exchange/Brute Force Sorts – $O(n^2)$ 151
11.1.1 Bubble Sort 151
11.1.2 Selection Sort 152
11.2 The More Efficient Sorts – $O(n*\log_2 n)$, $O(n*\log_2 n)$ on average, and $O(m*n)$ ... 153
11.2.1 Tree Sorts 154
11.2.2 Merge Sort 154
11.2.3 Quick Sort 157
11.2.4 Radix Sort 159
11.3 Exercises 163
Chapter 12. Searching 164
12.1 Basic Search Algorithms 164
12.1.1 Simple Sequential Search of Unordered Data 164
12.1.2 Binary Search of Ordered Data 165
12.2 Basic Searches with Efficiency Modifications 166
12.3 Exercises 169
References 170

Preface

This “book” started off as a set of notes in 1983 at the University of Idaho. In about 2009, when I first started teaching this class, I remembered how much I enjoyed the class when I took it, and how much I learned. So, I dug through some old boxes that had been soaked by Katrina and found my old notebook and rewrote the notes from my undergraduate class in Idaho taught to me by Dr. Karen Van Houten. For several years I lectured from those notes, but soon realized that I was saying much more in class than I was writing on the board. That is where this book came from. It has been used by countless classes, and over the years I have revised it a few times.

The philosophy behind this work is simple. I feel that it is better to learn the basics, that is, create a solid foundation, and then use this as a basis to solve a relevant, non-trivial problem. For many, the act of implementation often time illuminates various concepts in a better way than reviewing of literature about the algorithms and whatnot. And for most, programming and problem solving is an important if not central part of being a professional computer scientist. Expressed in a more colloquial way, “You can read about doing pullups all you want, but there is nothing like going to the gym and doing them.” The same applies here. This set of notes covers the basics of the majority the concepts in data structures but does not cover many variations. Instead of elaborating, it contains implementation details, traces, and other material, whose purpose is to provide a deep understanding of the foundational concepts.

This book is designed to be used for a one semester course in data structures. The student or instructor can pick and choose the order in which they want to cover the topics, but it is arranged so that the student can work from familiar topics and build upon those. For the most part, later chapters rely on work from earlier chapters, and early chapters try to rely on basics taught in introductory programming courses. This book does assume that the student is proficient in basic programming and logic.

Chapter 1. Introduction

The study of Data Structures is a central topic in Computer Science. As it turns out, many of the courses in the first two years of typical computer science curriculums directly or indirectly prepare the student for the data structures course. Also, the topics studied in this course provide a foundation for many courses that follow.

An important thing to remember about Data Structures is that it is a universal topic in computer science. It is not tied to any single language or area of computer science. Data Structures can be studied using many/most languages. It is true that some languages lend themselves more easily to creating good data structures than others, but still the topic is nearly a universal to Computer Science. Some languages, like JAVA, have many built in data structures, which is good, and bad. It is good because it is very easy and quick to use these features, but it is bad because these built-in features can help the student avoid learning the nuts and bolts of how they work. For the long-term benefit of students studying this topic, it is best to initially create the various data structures and algorithms by hand, so that a complete understanding of the concepts and details can be acquired.

In this chapter we want to provide some background material, describe what data structures are, how they are used, and how they can affect the use of computer resources such as memory and CPU time. We start with a discussion of data types, and word alignment, then with a brief discussion of data interpretation as it relates to data types. Next, there is a discussion about the need for data structures, followed by a brief introduction to the topics of memory and CPU time optimization.

1.1 What is a data type?

Data types are central to all computer operations. For data to make sense to us and the computer, we must know its type. For instance, we must know if it is an integer, character, or floating-point number. For example, when we are reading this page, we recognize the difference between characters and numbers. If we wanted to, we could pretend that each of these words is a number made of digits in the base 26 system ('a' could be 0, 'z' could be a 25), but that would not make a lot of sense. And that is the point. The computer needs to know how to interpret data. For example, it uses a different means to interpret floating point numbers than it does integers, or characters.

Another thing that each piece of data has to have besides its type is a size. Characters are typically represented in 1 byte of space, thus there can be 2^8 (256) different kinds of characters. This includes all the letters (lower and upper case) numbers, symbols, control characters, etc. Integers usually default to the computer's/OS's word size. For instance, a 32-bit computer will typically have integers that occupy 4 bytes. Maximum integer value of an unsigned integer would be $2^{32} – 1$. Signed integers would go from (-2^{31} to $2^{31} - 1$). Floating point numbers would have 4 bytes, while double floats would be 8 bytes.

Another thing that a computer needs to know about its data is its location. What is meant by location? Location refers to the address of where the value of the data is kept in memory. Think of location like a mailbox. The mailbox has an address on it, and it holds letters; the letters are analogous to the data values, which is the last characteristic of data.

So in summary, in the computer, each piece of data has a type (way to interpret the bits), a size, and a location, and while it has not been discussed up to this point, the data needs a value.

1.2 What about interpretation of data?

In the previous section of this chapter, it was noted that data has the following properties:

- data type - (examples include integer, character, floating point number, complex, etc.)
- size - the size of data is measured in bytes. Each type has a size associated with it. For example, character data is usually 1 byte in size (ASCII), but it may be that it is two bytes or more (Unicode).
- location - the location of the data is an address. The address is a logical (meaning not an absolute physical location in the computer's memory) unit usually based on an offset (a distance from the start of the program, or base address).
- value - the value of the variable/data.

When a program is compiled, each of the variables is assigned a location, and based on its data type a size. As the program runs, when the code references data (commonly in the form of variables), the data is stored at the location given to it by the compiler, the routines that are operating on the data interpret the data based on its type. Also, these same routines know the size of the data because of its type.

The computer uses the data type to indicate how it is to interpret the 1's and 0's at the location. For instance, the internal format of floating-point numbers is quite different from those of integers. If we were to add two integers using floating point math routines, our answers would certainly be not we expect them to be. For a more common example, consider a 1-byte number. It can take a value of -128 to 127. As long as our math operations stay within this range, the 1-byte number is sufficient. In fact, if we can guarantee that the number is always in this range, the 1-byte size may be best because it saves memory. On modern PC's this is not usually a consideration, but on microprocessors this topic is constantly being dealt with. Back to the 1-byte number, it can also be interpreted as an unsigned number with a range of 0 to 255, or it can be interpreted as a character in the ASCII character set. Below is a table of the standard data types, their sizes, and typical uses.

1.3 Standard types

The standard data types common to most languages are as follows:

data type	***size in bytes***	***remarks***
Character	1	for alpha-numerics or small integers
Integer	4	for signed integers (2^{-31} to 2^{31}-1) or unsigned integers in the range of 0 to 2^{32}-1
Float	4	single point floating precision numbers
Double	8	double point floating precision numbers

1.4 Word alignment

The computer accesses memory by units called words. Each computer/OS has its own word size. Currently most PC based systems are using 32- and 64-bit words. In a 32-bit word computer, memory is accessed every 4 bytes/32 bits. That is, the computer does not access (do read operations or write operations) at the bit level, but at the word level. A 32-bit computer moves data in and out of memory in groups of 32 bits or 4 bytes. Likewise, in a 64-bit computer, memory is accessed every 8 bytes. It used to be that commonly available computers (older PCs) used an 8- or 16-bit word.

One of the less obvious consequences of this method of memory access can be seen when the computer is moving pieces of data in and out of the memory that are not an even multiple of the computers word size. For example, if a 32-bit system (4-byte word size) is moving data that is 10 bytes long, the end of the data does not fall on an even word boundary. That is, since the end of the data is at the 10-byte mark, the nearest word boundaries are at the 8 byte and 12-byte marks. When the computer reads the data, it must retrieve 12 bytes to get the 10 bytes it needs, and likewise, when it writes the data, it will write 12 bytes out. Those last two bytes on the end are not used, but they are along for the ride. This is illustrated in figure 1.1 below.

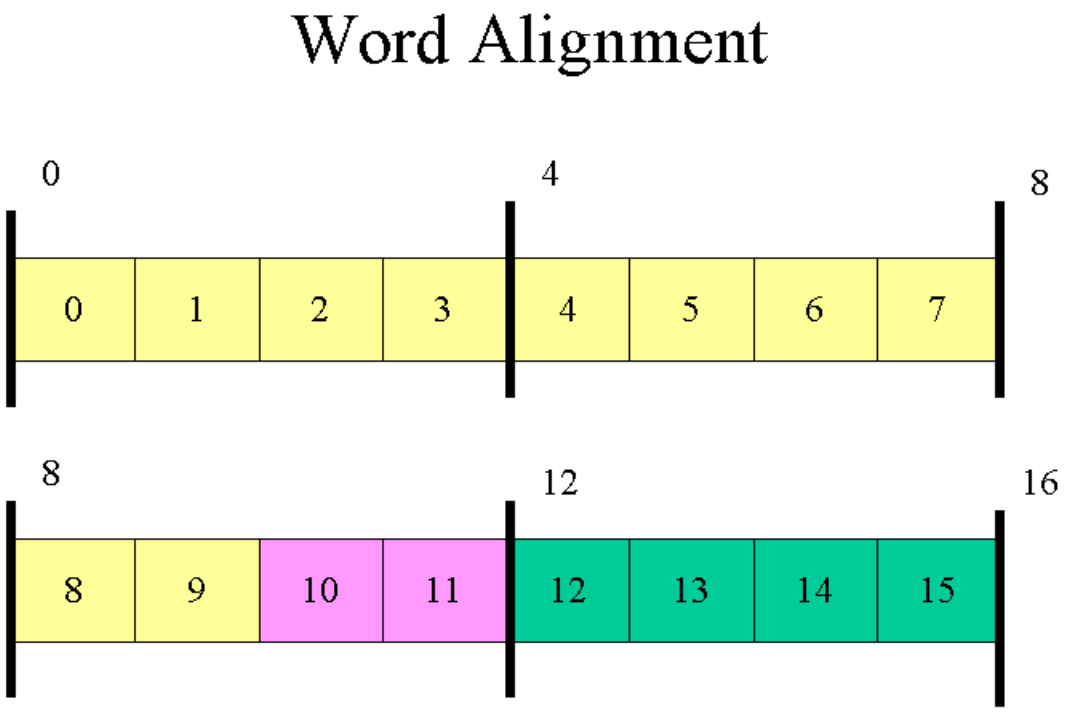


Figure 1.1 This figure shows a memory map of bytes 0 through 15 in computer with a 32-bit word (4 bytes). Each rectangle represents a single 8-bit byte; each of the heavy black vertical lines shows a word boundary. As in the example described in the text, bytes 0 through 9 (the first 10 bytes) represent our data. When the computer reads or writes the data, it moves bytes 0 through 11 in and out of the computer, because it can only transfer memory at the word level.

Another consequence of this system is the way data is put into classes and structures. Most computers will not put a data item that requires 4 bytes of memory across a word boundary. The effects of this can be seen by checking the size of the data in a class or structure. Looking at the two structures in figures 1.3 and 1.4 below we can see how the arrangement of the data (in the structures in figure 1.2) can affect the size of each class.

Two classes, same data, different memory requirements

```
class myData0 {
    short integer height;
    long integer hairCount;
    long integer numFreckles;
    short integer weight;
}
```

```
class myData1 {
    short integer height;
    short integer weight;
    long integer hairCount;
    long integer numFreckles;
}
```

Figure 1.2. Two classes with same data content, but because the data is arranged differently, the class sizes will be different. This effect is caused by word alignment.

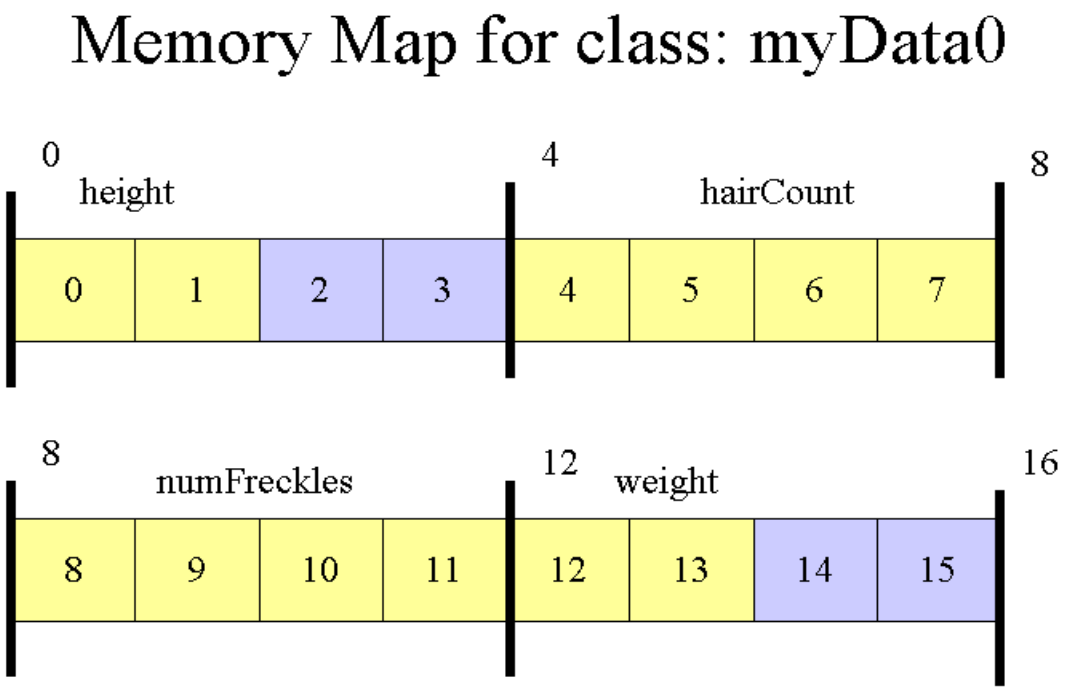


Figure 1.3. Data map for the myData0 class. The yellow block represents bytes that hold data. The light blue bytes represent bytes that are not used because the computer wants to start word length data items on a word boundary. Class size is 16 bytes.

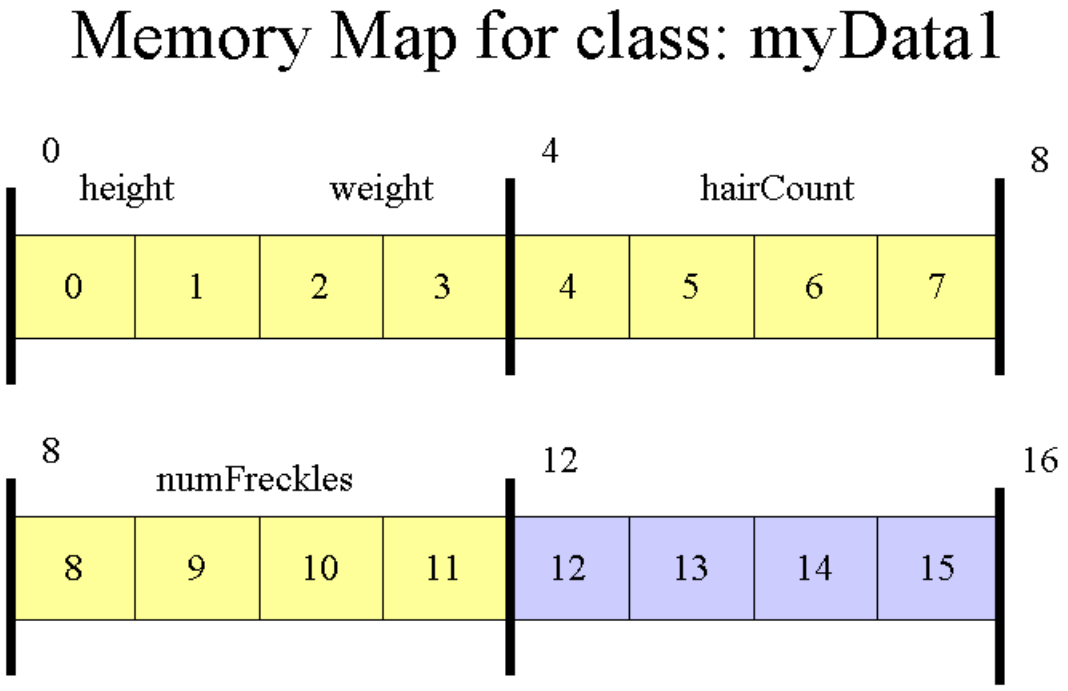


Figure 4. This figure illustrates how rearranging of the data in the class can save 4 bytes. Here the class size is 12 bytes versus 16 for the myData0 class as shown in figure 3. Once again yellow blocks represent bytes that are used by the class; blue blocks represent blocks not used by the structure; the heavy vertical lines represent word boundaries.

In summary, the computer system reads and writes data to memory in words or multiples of words. Each system has a word size; most often a modern PC will have a word size of 32, 64, or 128 bits (4, 8, or 16 bytes). The system does not like split a word across a word boundary, so if needed, it will move down to the next word boundary and start there.

For the most part this process is transparent. One of the times when it is not is when reading and writing formatted binary data. This type of data is often composed of data packets created by the data portion of a class or structure. In this case the structure can usually be read and written in binary with no consideration of the word boundaries, unless the reading and writing machines have different word sizes. Another time when it is not is when transparent is when reading and writing an indexed binary file such as that of a compressed image with a table of contents portion. In these cases, sometimes it is sometimes necessary for the program to "hop" from location to location in the file to retrieve the data of interest. In these cases, and others like it, being aware of word boundaries can be helpful.

1.5 What is a data structure and why are they useful?

The standard data types are useful for a myriad of problems, but there are times when either they are inadequate for the application, or by bundling them together in a specific arrangement, the logic of the program can be made much cleaner. It is these bundles of standard data types that we call a data structure. Often times, data structures are created to make it easier to deal with groups of data. For instance, if we had data comprised of a person's name, height, and weight, we could create 3 separate arrays to hold the data, or a "data record" could be made comprised of the data items name, height, and weight. By doing this we would have a program with 1 array of records, instead of three separate arrays. Another advantage of arranging data in this way is that there is less (almost zero) chance of mixing up two people's weights and heights.

The above example helps make the logic of the program a bit cleaner, but it does not really help to make the program more efficient in terms of CPU time or memory usage. One of the primary goals of the study of data structures is to learn how to create data arrangements and algorithms that operate on them that optimize CPU time and memory usage. Often these come at a price, usually program/code complexity is increased, but good programming practices can mitigate these effects.

In this course the student will study data structures, their associated algorithms, and their effects on CPU and memory usage. To study these effects, time order complexity will be introduced (commonly known as Big O notation) and explained. Other topics, including recursion and algebraic techniques related to time order complexity will be touched upon in order to support the study of data structures.

1.6 Exercises

Most of the exercises below use the C/C++ language. It's syntax is very similar to that of Java. The main difference is that C/C++ does not require the main program to be part of an object. In order to do these exercises, the use of an online compiler such as JDoodle or Repl is encouraged. The advantage to these is that they require no compiler installation, and their environments are set up so that the user can paste code into their development windows, which always hold a small basic program that compiles and runs, thus making the system easier to get acquainted with.

1. Create a program that starts a loop at the value of 48 and runs for 10 iterations. In the loop print the integer value of loop counter and the character value of the loop counter. See the snippet of C code below for an example:

```
.
.
.
int j;

for(j=48;j<48+10;j++) {
     printf("%d %c \n", j, j);
}
printf("\n");
.
```

2. Use the "sizeof()" keyword to determine the size in bytes of the basic data types: int, short int, long int, float, char, unsigned char. Check the example below for hints. This example is from JDoodle – if you were to run this in Repl, you may have to switch the "%d" to a "%lu".

```
#include<stdio.h>

int main() {
    printf("The size of an int is %d \n", sizeof(int));
}
```

3. In C/C++ create the classes myData0 and myData1 as shown at top of page 9 and in the example below.

```
#include <stdio.h>

class myData0 {
     short int height;
     long int hairCount;
     long int numFreckles;
     short int weight;
};

class myData1 {
     short int height;
     short int weight;
```

```
      long int hairCount;
      long int numFreckles;
};
```

Use a main program like the one shown below to verify the actual size of the two classes.

```
void main(void)
{
      printf("size of myData0 in bytes = %d \n", sizeof(myData0));

      printf("size of myData0 in bytes = %d \n", sizeof(myData1));

}
```

4. Declare 3 character arrays, bink[13], twink[14], and fwink[16]. Use the sizeof() function to determine their sizes in bytes. Can you explain your results?

Chapter 2. Arrays, Memory Arrangement, and Pointers

One of the most fundamental data structures is the 1-dimensional array. It's benefits include simplicity of both concept, functionality and implementation. The array data structure that provides random access to data elements using an easy-to-use indexing scheme. When we think of the typical array, we picture a linear list of data elements which can be accessed by element number. The element number is most often referred to as the array index. Figure 2.1 below shows a 1-dimensional array.

1 Dimensional Array

- Here an array of integers is illustrated. The left column is presented for clarity, it is the index of each data item in the right hand column. For this illustration the data is CS course numbers.
- The length of the array is 10
- Array length can be found by using the following formula:
  - arrayLength = (endIndex – startIndex) + 1
  - For this array: arrayLength = (9 – 0) + 1
    arrayLength = 10
- Note that the first index in this array is element 0. Not all languages use 0 for the first index.
  - First Index = 0; C, C++, JAVA…..
  - First Index = 1; FORTRAN

Index	Course numbers
0	101
1	110
2	285
3	375
4	389
5	390
6	401
7	411
8	431
9	482

Figure 2.1. This figure illustrates a one-dimensional array and its indices.

Accessing the elements of the array is accomplished using the array name and index and the assignment operator as show below:

```
array[0] = 101;
```

This is read as "array subzero equals 101", meaning that the integer 101 is assigned to the 0^{th} element of the array. Remember that the 0^{th} element is the first element of the array in languages like C and JAVA. Other languages, like FORTRAN or BASIC, refer to the

first element of the array using a 1 instead of the 0. A handy way to fill an array is use a loop, like a for loop or while loop as shown below in pseudo code. *Note: The pseudo code is non language specific. Its purpose is to show logical flow of algorithms. The reader is encouraged to translate the pseudo code to his/her favorite language to test the concepts for themselves.*

```
for(j=0;j<10;j++) {
     array[j] = j;
}
```

or

```
j = 0;
while(j < 10) {
     array[j] = j;
     j = j+1;
}
```

The above pseudo code snippets would produce an array loaded as illustrated in table 2.11 below:

Index	Value
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9

Table 2.11 This table shows array index value along with the array value as produced by the code snippets above.

While a one-dimensional array can be thought of a linear list of data elements which can be accessed in any order using the array index, a two-dimensional array can be thought of as a square grid of data elements. A two-dimensional array is commonly thought of as a series of rows of data, or conversely, a series of columns of data. Individual data items in the grid are accessed using the row and column numbers. Figure 2.2 below shows a two-dimensional array with 9 elements.

2-Dimensional Array of letters

- Here a 3 row by 3 column two-dimensional array is shown.
- The blue numbers across the top indicate the column indices, while the yellow numbers on the left side indicate the row indices.
- The letters from 'a' to 'i' in the pink area represent the data.
- The number of elements in the array can be found using the following equation:

  nElements = number of rows * number of coloums
- In this example we have:

  nElements = 3 * 3

  nElements = 9

Columns / Rows	0	1	2
0	a	b	c
1	d	e	f
2	g	h	i

Figure 2.2. This figure shows a two-dimensional array of size 3 rows by 3 columns.

Filling the two dimensional array can be done by using nested "for" loops. The pseudo code example below illustrates the nested for loops.

```
letter = 'a';
for(row = 0;row < 3;row++) {
     for(col = 0; col < 3; col++) {
          array2D[row][col] = letter;
          letter = getNextLetter(letter);
     }
}
```

In the pseudo code above, the letter variable is initialized to 'a'. The getNextLetter() function returns the letter in the alphabet that follows the letter it is passed. So, if it is passed an 'a', it returns a 'b' and so on and so forth. The nested for loops will fill the array row by row. It does this because for each row (controlled by the outer for loop), the inner for loop executes it self from start to end. So, starting at row 0, the inner for loop visits columns 0, 1, and 2. When the outer for loop switches to row 1, the inner for loop again visits columns 0, 1, and 2, thus moving down the row one column at a time. Loading a two-dimensional array in this manner can be described as a "row wise load". If the outer loop traversed the columns and inner loop traversed the rows, the array would be loaded "column wise".

2.1 Arrays in Memory

In general arrays are laid out in memory in linear fashion. When the compiler is converting the program into an executable form, it reserves space for whatever arrays the programmer declared during the coding process. Each array will have a base address and a length. The compiler determines the arrays total length by taking into account the number of elements in the array and its data type. As an example, if the programmer declares an integer array as follows:

```
int bigArray[10];
```

For arguments sake, suppose the compiler assigns the array's starting address to location 200. Since it is an integer array, each element will need 4 bytes, so the arrays total length in bytes will be 40 bytes. Most of the time an arrays length is measured in terms of the number of elements, but in the computer, the number of bytes is important also. In this example, the first element of the array starts at memory location 200, and the last element starts at memory location 236 (remember, these are logical memory locations). The table below shows the addresses associated with each of the elements in bigArray.

Address	***Array element***	***Bytes Element Occupies (inclusive)***
200	bigArray[0]	200, 201, 202, 203
204	bigArray[1]	204, 205, 206, 207
208	bigArray[2]	208 – 211
212	bigArray[3]	212 – 215
216	bigArray[4]	216 – 219
220	bigArray[5]	220 – 223
224	bigArray[6]	224 – 227
228	bigArray[7]	228 – 231
232	bigArray[8]	232 – 235
236	bigArray[9]	236, 237, 238, 239

Table 2.12 This table assumes that the integer array bigArray starts at a memory location of 200. Note that each array element occupies 4 bytes.

Looking at table 2.12 above, we can note that the array does indeed use 40 bytes, even though at first glance it appears to be 39 bytes because it starts at position 200 and ends at 239. Remember it is inclusive starting from 0. It is like if you counted the fingers on one hand but started at 0. Using this method, you end at 4! The equation below shows how to calculate the byte count:

```
byte count = (ending position - starting position) + 1
```

For this example it comes out as follows:

```
byte count = (239 - 200) + 1
byte count = 39 + 1 = 40
```

But what about two-dimensional arrays? It is easy to think of them as a square formation of memory elements in the computer (see figure 2.3 below). Notice that in the two-dimensional array of size 4, 5 (4 rows by 5 columns), the element in the upper left of the array is labeled as element 0, 0, that is row 0, column 0. Likewise, the element in the lower right is element 3, 4.

In reality, 2D (two-dimensional) arrays are arranged linearly in the computer's memory. In a row wise system, a 2D array is arranged with the rows placed one after another, much like arranging a series of one-dimensional arrays one after another. A column wise system operates in a similar manner, except that the array is arranged as a series of sequential columns. Figure 2.4 below illustrates this concept.

The advantage of laying the rows into memory one after another is that the memory addresses are all sequential. That is, if starting at array[0][0] incrementing the address by 1 element at a time, the entire array can be traversed using only adding. So, being sequential gives some advantages in access speed when working at a very low level. In the next section these topics will be discussed.

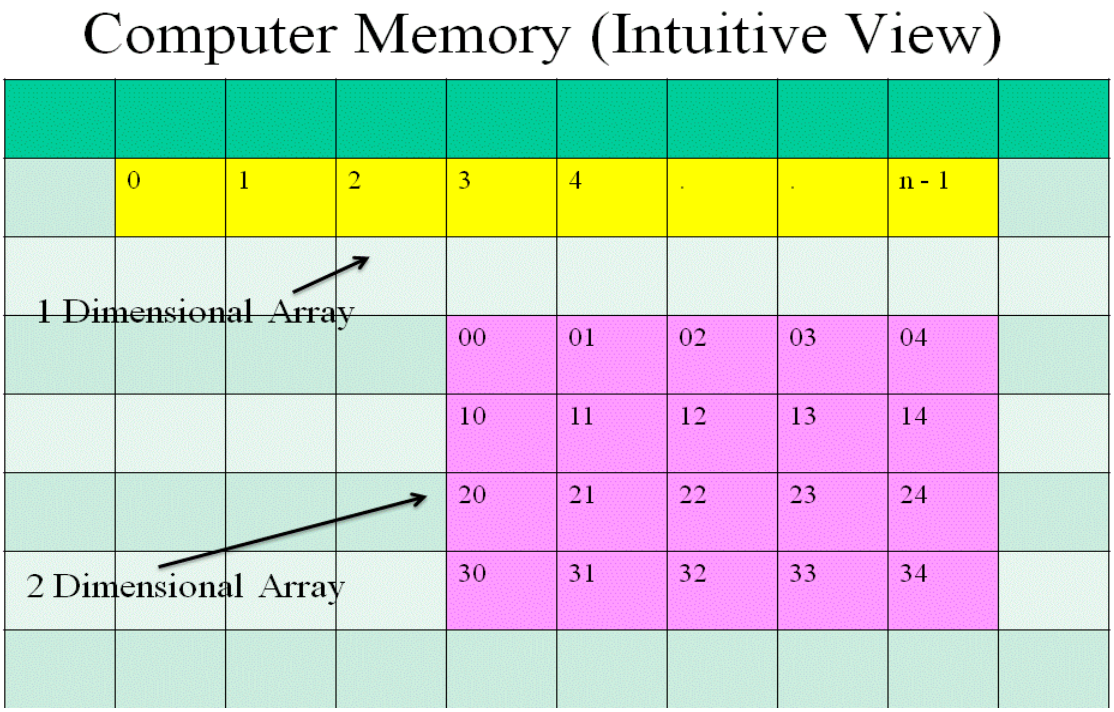


Figure 2.3. This figure shows an intuitive view of a computer's memory in which a 1-dimensional array (the yellow shaded boxes) and a 2 dimensional array (the pink boxes) reside. It is somewhat natural to picture the 2-dimensional arrays as being a square formation in the computer's memory.

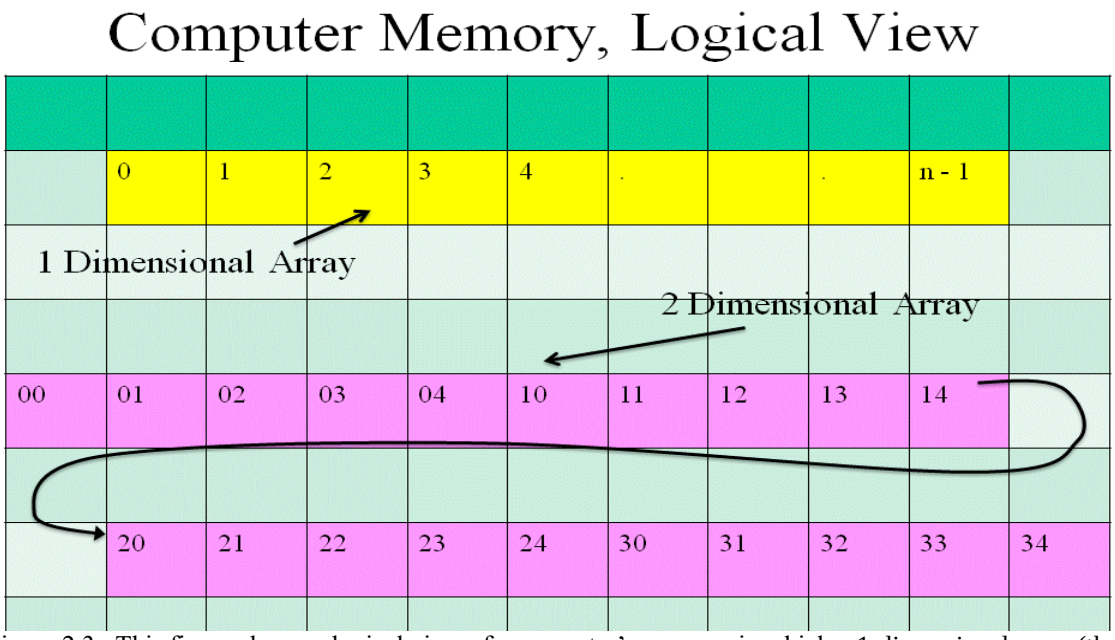


Figure 2.3. This figure shows a logical view of a computer's memory in which a 1-dimensional array (the yellow shaded boxes) and a 2-dimensional array (the pink boxes) reside. The two-dimensional array is arranged linearly one row after another in memory. Note that memory element [1][4] is physically adjacent to memory element [2][0], as indicated by the black arrow.

2.2 Pointers and Pointer Arithmetic

In this section low level addressing will be discussed. In order to do this, the code snippets will be written in the "C" language, because it is one of the most common languages that allows direct address manipulation.

The word "pointer" commonly refers to address. For instance, a pointer variable is a variable that holds addresses. There are several common analogies, one of the most common being the mailbox analogy. Think of a variable as a mailbox, the address of the variable is the number on the side of the mailbox; the value of the variable is the mail in the mailbox.

The only thing this analogy leaves out is the "pointer variable" part. The way to think of this is as follows:

Suppose the mailman is asked to check some of the mailboxes for missing flags on his route. The number of each box to be checked is written on a note card that the post office uses specifically for this purpose. As the mailman travels his route, he looks at his collection of cards which indicate which boxes need to be checked for the missing flags. In this example the note card with the mailbox number on it represents a "pointer variable". The pointer variable is a variable that holds an address (just like the note card is a piece of paper that holds the number of a mailbox) of a variable. In our example, the note card was initially blank, then someone in the post office filled it in. In "C", we can declare a pointer variable (sometimes called an address variable), and then give it the address of a variable of its type. For example, we can declare an integer pointer variable, and then give it the address of an integer variable. This is just as in the mailbox analogy;

we have a note card that is of mailbox type (it is mailbox type because our note cards are only used for mailbox numbers); a postal worker fills in the card with the mail box number of the mail box that the post man needs to check. The mailbox number is the location of that mail box (note the mail box number is the location, and the mail box is the physical entity).

Some of the uses for pointer variables (also called memory pointers or address variables) are listed below:

- Pointers are used commonly in "C" to pass the address of variables whose contents are to be changed by a function. This is referred to as "pass by reference".
- Pointers can be used to quickly traverse an array.
- Pointers to functions can be made, so that a calling program can pass a function to another function.
- Pointers can be used to change the order that data is accessed without physically moving the data. This is useful when working with large data items.

2.21 General Rules for Working With Pointers

When declaring variables:

- When a variable is declared, the compiler does the following saves the following attributes for that variable.
    - Data type - integer, floating point, character, etc.…
    - Data size - common space requirements are 4 bytes for integers and floating-point numbers, 1 byte for characters, 2 bytes for short integers, 8 bytes for double precision floating point numbers
    - Location – memory address of the reserved space
    - Value – this is highly language dependent, some language use default values, others do no preloading.
- The '*' (Asterix) operator indicates that the variable will be a pointer.
- In a method's parameter list:
    - The "*" indicates that the variable is a pointer.
    - The "&" indicates that the variable is a reference variable. This means that if the value of the variable is changed in the method, it will retain its value when the method ends. The nice thing about reference variables is that the syntax of pointers is not needed, but the ability of the variable to hold its value upon return from the method is there.

- In the logic of the program:
    - The '*' operator:
        - If on the right hand side of an equation, next to a pointer variable, the '*' operator returns the value at the location pointed to by the pointer variable.
        - If on the left hand side of an equation, the memory location pointed to by the pointer variable next to the '*' is loaded with whatever the right hand side of the equation evaluates to.

- The '&' operator returns the address of a variable

- Other notable pointer/address tidbits:
  - An array name is equivalent to the address of the first element of the array.
  - When a pointer variable is declared, it has no value. I must be initialized to some value within the program's memory space.

Example 1 - Basic Declaration and use of Pointers

Below is an example piece of code that declares an integer pointer and sets it equal to the value of the location of one of the other integer variables in the program. It goes on to do some addition and output. *Note: Some compilers like the "main" program to be of type "void", while others prefer it to be of type "int". If it is of type "int", it must return something, usually just a zero value; if it is of type "void" nothing needs to be returned.*

```
#include <stdio.h>

void main()
{
      int *num;
      int x, y;

      x = 3;
      num = &x;
      y = *num + 4;

      printf("y = %d \n", y);
}
```

It prints out:

```
y = 7
```

Why? From the top:

- num is declared as an integer pointer variable.
- x and y are declared as regular integers.
- x is given the value of 3.
- num is given the address of the integer variable x.
- y is assigned the contents of what is at the location pointed to by num, + 4. So y is assigned to be 3 plus 4.
- The printf statements prints out the value of the variable y.

Example 2 - Pass by Reference in C

Below is an example piece of code, run in Repl -C, that declares an integer pointer and sets equal to the value of the location of one of the other integer variables in the program. It goes on to do some addition and output. *See the note from the previous example.*

```
#include <stdio.h>
#include <math.h>

void normalizeVector(float *x, float *y);

int main()
{
      float x, y;

      x = 3.0;
      y = 4.0;

      normalizeVector(&x, &y);

      printf("normalized vector = ( %f, %f ) \n", x, y);

  return 0;
}


void normalizeVector(float *x, float *y)
{
      float magnitude;
      float xWork, yWork;

      xWork = *x;
      yWork = *y;

      magnitude = sqrt(xWork*xWork + yWork*yWork);
      *x = xWork/magnitude;
      *y = yWork/magnitude;
}
```

If run in Repl C, it prints out:

```
normalized vector = ( 0.600000, 0.800000 )
```

Does this make sense? The following bullets serve as to a guide for the above code:

- Before the main program:
  - The standard i/o and math libraries are included.
  - The normalizeVector function is declared.
- In the main program:
  - x and y are declared as floating-point numbers.
  - x and y are assigned the values of 3.0 and 4.0 respectively.
  - The normalizeVector method is called, the addresses of the variables x and y are passed to the method (this is the call by reference).
  - When the normalizeVector completes, the printf statement writes out the values of x and y, which were changed by the normalizeVector routine.

- In the normalizeVector method
    - The addresses of the x and y variables are received.
    - The floating-point variables magnitude, xWork, and yWork are declared.
    - The xWork and yWork variables are assigned the values located at the addresses of x and y (3 and 4). Here the "*" operator is used recover these values, as explained in section 2.21.
    - The magnitude of the vector is calculated using the sqrt() function.
    - Using the "*" operator, the locations pointed to by the x and y address variables are loaded with normalized vector component values.
    - Control returns to the main program.

Example 3 - Reference Variable in C++

Here, the previous example is rewritten using a reference variable. The advantage of this implementation is that it avoids the use of the "*" operator in order to assign or access values of a variable. *Note: this example is very similar to example 2 above, but for it to work it must run in C++.*

```
#include <stdio.h>
#include <math.h>

void normalizeVector(float &x, float &y);

int main()
{
      float x, y;

      x = 3.0;
      y = 4.0;

      normalizeVector(x, y);

      printf("normalized vector = ( %f, %f ) \n", x, y);

  return 0;
}


void normalizeVector(float &x, float &y)
{
      float magnitude;

      magnitude = sqrt(x*x + y*y);
      x = x/magnitude;
      y = y/magnitude;
}
```

If run in Repl C++, it prints out:

```
normalized vector = ( 0.600000, 0.800000 )
```

This code is nearly identical to the code of the previous example. The difference is in the way the parameters are declared in the normalizeVector function. By using the "&" operator, the normalizeVector function has access to the variables being passed to it, not just access to the values of those variables, access to the original location. This is

functionally the same as having the address of those variables, but the syntax is nicer for those not used to using C pointers. Since the "&" is used, all of the work de-referencing the variables using the "*", as in the previous example, can be avoided. As stated earlier, this is a C++ feature, so if the programmer is using regular C, the ampersand and asterix will be required.

Example 4 - One dimensional Array Traversal

In this example an array is filled with consecutive numbers using standard array access methods. Then its contents is printed out twice, the first time uses a pointer to the front of the array and adds an offset to each time. The second time sets the pointer to the front of the array and increments the pointer in the loop each time. This code runs in Repl C.

```
#include <stdio.h>

int main()
{
      int j;
      int x[10], *xptr;

      for(j=0;j<10;j++) {
            x[j] = j;
      }

      printf("Contents of the x array... \n");

      /* Set xptr to the front of the x array. */
      xptr = &(x[0]);

      /* Print out the contents of the array. */
      for(j=0;j<10;j++){
            printf("%d ", *(xptr + j));
      }
      printf("\n");

      /* Print out contents again, this time increment the
         pointer. */
      printf("Contents of the x array again... \n");

      /* Set xptr to the front of the x array. */
      xptr = &(x[0]);

      /* Print out the contents of the array. */
      for(j=0;j<10;j++) {
            printf("%d ", *xptr);
            xptr++;
      }

  return 0;
}
```

It prints out:

```
Contents of the x array...
0 1 2 3 4 5 6 7 8 9
Contents of the x array again...
0 1 2 3 4 5 6 7 8 9
```

The question that needs to be answered about this example is, "What is the difference in these array access methods?" The answer is efficiency. In the standard access method, the code:

```
x[4] = 7;
```

sets the 4th element of the x array to the value 7. To access the 4th element of the x array, the computer must calculate that element's address. It does so by knowing where the beginning of the array is, and then adding the appropriate offset. The offset is calculated by knowing the size of the data type (in this case it is 4 because x is declared as an integer array) and multiplying that size times the desired element number. In the case of:

```
x[4] = 7;
```

16 is added to the address of the start of the x array. Remember, x[0] occupies the first 4 bytes of array, x[1] the next four bytes, and so on. So, in reality, when the code is compiled, the standard syntax is converted to something that looks almost identical to the code in the first output loop in the example. The only real difference is that the statement:

```
printf("%d ", *(xptr + j));
```

really works out to be something like:

```
printf("%d ", *(xptr + (j*sizeof(int)));
```

In the next loop, the contents of the array are printed out by incrementing a pointer. It is done with the statement:

```
xptr++;
```

Because xptr is a pointer to an integer, it gets incremented by the size of the integer data type, which again is 4. This method is more efficient than the previous methods because it avoids the multiplies. For an array size of 10, the efficiency gain may be small, but what if the array size were a million, and the size of the data elements were large. What would happen then?

The question posed above about the savings of computational time by using a more efficient method is extremely relevant in cases involving animation and video. In these cases, the video images are held in frames which are 2D arrays, usually whose minimum size is 640 by 480, and the frames get displayed at a frame rate of a minimum of around 24 frames per second. The need for effective and efficient data transfer in these cases is warranted.

Example 4 : Two dimensional Array Traversal

This example is similar to the previous example, except that the array is two dimensional. This means that the row major effect (discussed in the "Arrays in Memory" section) comes into play. The example is done in Repl C.

```
#include <stdio.h>

int main()
{
      int row, col, j;
      int x[4][4], *xptr;

      for(row=0;row<4;row++) {
            for(col=0;col<4;col++) {
                  x[row][col] = (row*4)+col;
            }
      }

      printf("Contents of the x array... \n");

      /* Set xptr to the front of the x array. */
      xptr = &(x[0][0]);

      /* Print out the contents of the array. */
      for(row=0;row<4;row++) {
            for(col=0;col<4;col++) {
                  printf("%d ", *(xptr + (row*4)+col));
            }
      }
      printf("\n");


      /* Print out contents again, this time increment the
         pointer. */
      printf("Contents of the x array again... \n");

      /* Set xptr to the front of the x array. */
      xptr = &(x[0][0]);

      /* Print out the contents of the array. */
      for(j=0;j<16;j++){
            printf("%d ", *xptr);
            xptr++;
      }

  return 0;
}
```

It prints out:

```
Contents of the x array...
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Contents of the x array again...
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

This example is very similar to the previous on in that it uses conventional syntax to fill the array, pointers with offsets to write the array out the first time, and an incremented pointer to write the array the second time. This example is set up to take advantage of

the fact that two dimensional arrays are arranged as a series of one-dimensional array in memory. In "C" the arrays are arranged with row0 being first, followed by row1, row2, and so forth. For this reason, the outer loops all reference the rows, while the inner loops reference the columns.

Once again, multiplications are avoided by using the incremented pointer. Look to the previous example for more detail.

Example 5 - Using pointers to exchange rows in a 2-D array;

Below is an example piece of code that declares an integer pointer and sets equal to the value of the location of one of the other integer variables in the program. It goes on to do some addition and output.

```
#include <stdio.h>

int main()
{
      int row, col, j;
      int x[4][4], *xptr;
      int *row0, *row2, *temp, *ptr;

      for(row=0;row<4;row++) {
            for(col=0;col<4;col++) {
                  x[row][col] = (row*4)+col;
            }
      }

      row0 = &(x[0][0]);
      row2 = &(x[2][0]);

      /* Exchange rows. */
      temp = row0;
      row0 = row2;
      row2 = temp;

      /* Print row0. */
      ptr = row0;
      for(j=0;j<4;j++) {
            printf(" %d", *ptr);
            ptr++;
      }
      printf("\n");

  return 0;
}
```

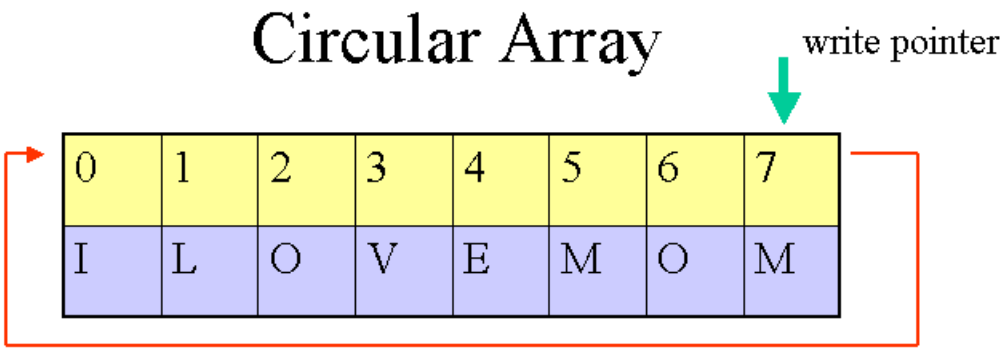
It prints out:

```
8 9 10 11
```

Here, data was never moved, only pointers into the data were modified. See if you can understand why the output is the way it is.

2.3 Circular Arrays

One array of particular use for data collection or data sharing among processes is the circular array. Typically, one process loads data into the array, while another process does some operation on the data, such as conversion and storage of the relevant information, display, etc. The loading process is often referred to the "writer process" or the "producer", while the process consuming the data is called the "reader process" or the "consumer". If this operation were done without a circular array and it were run for any duration of time with a high rate of writing and reading, memory use would quickly become an issue. The circular array allows for the reuse of memory.



•Array Indices are in yellow
• Data is in blue.

Incrementing the write pointer in a Circular Array

```
write pointer = write pointer + 1;
If (write pointer == arrayLen –1) {
  write pointer = 0;
}
```

Or

```
write pointer = modulo((write pointer +1), arrayLen);
```

Figure 2.4. This figure illustrates a circular array of length 8. In the figure, the write pointer is at element number 7 or the array, that is the M has just been written. When the write pointer is incremented, it takes the value of 0; i.e. the next data element will be written to array position 0.

In order to implement a circular array, we allocate a regular array, and load it as usual, starting our writing index at 0 and incrementing by 1 each time a new element is added to the array. When the last element in the array is filled (the index with the value of array length - 1) the write pointer is set back to 0. This can be done using either if/then logic or the modulo function. Figure 2.4 above illustrates these principles.

The pseudo-code below implements a pair of processes loading and unloading a common buffer. In this example the writer process is collecting data from a sensor at a constant frequency and placing it in a common buffer, while the reader process is unloading the

data and sending to a plotting routine for display. Each process has an index into the array; the write process has a write pointer and read process has a read pointer. For this system to work correctly, the write process must stay ahead of the read process, but not overtake it and the read process must stay behind the write process, and again, not overtake it. If the write process overtakes the read process, it will overwrite unprocessed data; if the read process overtakes the write process it will essentially process data that has been already examined. In later classes, such as Operating Systems, the methods needed to synchronize the read and write pointers is discussed in detail.

```
Writer(int bufferLen)
{
     array buffer[];

     int writePointer;

     external common int count;

     If (buffer does not already exist) {
          buffer = createBuffer();
     }

     writePointer = 0;
     count = 0;

     while(collecting data) {

          if (count < bufferLen) {
               dataItem = readSensor();

               buffer[writePointer] = dataItem;

               count++;
               writePointer = writePointer + 1;
               if (writePointer == bufferLen) {
                    writePointer = 0;
               }
          }
          else {
               print("buffer full, check read process.");
          }

          delay(1/sampling frequency);
     }
}
```

```
reader(int bufferLen)
{
     array buffer[];

     int readPointer

     external common int count;

     If (buffer does not already exist) {
          buffer = createBuffer();
     }

     readPointer = 0;

     while(reading data) {

          if (count == 0) {
               print("buffer empty, check write process.");
          }
          else {

               dataItem = buffer[readPointer];

               sendData2Display(dataItem);

               count--;
               readPointer = readPointer + 1;
               if (readPointer == bufferLen) {
                    readPointer = 0;
               }
          }

          delay(1/sampling frequency);
     }
}
```

Strictly speaking, for these examples to be guaranteed to work, the routines that increment and decrement (++ and --) the shared variable count must be atomic. In this context the word atomic means that once the routine starts it cannot be interrupted. If not, process synchronization issues can occur.

2.4 Exercises

1. Write a C program that does the following:
    a. Declare a 1-dimensional array of length 10.
    b. Using conventional array access methods, fill the array with consecutive numbers starting with 0.
    c. Print out the contents of the array using conventional array access methods.
    d. Now using a memory pointer, fill the array with consecutive numbers starting with 100.
    e. Print out the contents of the array using a memory pointer.
2. Write a C program that fills a 5 by 5 2d array with consecutive numbers, in row major fashion. An example of the array is below:

    0 1 2 3 4
    5 6 7 8 9
    10 11 12 13 14
    15 16 17 18 19
    20 21 22 23 24

    a. Use conventional array accessing methods to create a 5 by 5 array.
    b. Use conventional array accessing methods to display the array.
    c. Use the pointer/addressing methods discussed in the chapter to display the array.
    d. Use the pointer/addressing methods discussed in the chapter to fill the array.
    e. Now load the matrix so that consecutive number go down the columns (similar to column major fashion) using pointer/addressing methods as discussed in the chapter.
    f. Write out the result; it should appear as below:

    0 5 10 15 20
    1 6 11 16 21
    2 7 12 17 22
    3 8 13 18 23
    4 9 14 19 24

3. Write a C program that declares a 2 dimensional integer array with N rows and M columns and loads it with consecutive numbers using conventional array element access methods. Let N and M be equal to 10. Using the pointer/addressing techniques discussed in the chapter do the following exercises:
    a. Write a function that will display a specified row of the array. Use the function prototype below:

    ```
    void showRow(int *arrayName, int rowNumber, int nColsInRow)
    ```

    b. Write a function that uses the showRow function to display the entire 2d array.

c. Write a function that will display a specified column of the array. Use the function prototype below:

```
void showCol(int *arrayName, int colNumber, int nRowsInCol)
```

d. Write a function that uses the showCol function to display all the columns of the array.

4. Write a C program that fills creates an identity matrix. (An identity matrix is a square matrix that has all 1's on the diagonal.)

Example of a 5 by 5 identity matrix:

1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1

a. Use conventional array accessing methods to create a 5 by 5 identity matrix.
b. Use conventional array accessing methods to display the matrix.
c. Use the pointer/addressing methods discussed in the chapter to display the matrix.
d. Use the pointer/addressing methods discussed in the chapter to fill the matrix.
e. Use pointer/addressing methods to exchange rows 0 and 4 of the identity matrix.
f. Write out the result; it should appear as below:

0 0 0 0 1
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
1 0 0 0 0

Chapter 3. Stacks and Queues

Two of the most basic data structures are stacks and queues. Our discussion of data structures will begin with stacks and queues because they are easy to implement with arrays and they are so fundamental; in fact, they are used by many of the other data structures that will be studied in later chapters. In this chapter, stacks will be discussed first, and then queues will follow. Along the way, some basic applications will be described.

Stacks are called stacks because of the way things are put into and removed from a stack. When data is put into a stack, it is placed on the top of the stack, when it is removed it taken from the top.

It is akin to the stack of plates at a cafeteria. When a person goes to eat, they grab the plate off the top of the plates. No one gets a plate from the middle unless they want to get smacked by a grouchy cafeteria worker. When the workers are setting out the plates, they put all the new clean plates on the top of the stack. They don't put them in the middle, or on the bottom, it would be harder, plus it could cause the plates to fall, inevitably causing some of the cafeteria workers to have to clean up a mess, making them grouchy.

So, the plates at bottom of the pile were put there by the workers first, and since everyone takes plates from the top, the plates at the bottom will be last to be used. This order of insertion and removal is called:

Last In First Out - or LIFO

That is, the last item inserted onto the top of the stack will be the first one removed.

also, it can be called

First In Last Out - or FILO

So far only stacks have been discussed. As it turns out stacks and queues are very similar in that they can be both be easily implemented using single dimensional arrays, but functionally they almost the opposite of each other. Instead of using the top of the stack to add and remove items, the queue is like a line at a supermarket cash register. That is, items are added at the back of the queue, and removed from the front. The queues order of insertion and removal is referred to as:

First In First Out - or FIFO

The diagram in figure 3.1 below illustrates the stack and the queue. In these diagrams it is easy to envision the stack and queue as being implemented using arrays. The numbers

to the left of the stack and below the queue diagrams are the array indices. The top of the stack is at index 3; the front of the queue is index 0, and the back is index 3.

New items are placed on the stack by incrementing the top and placing the item at that location. To remove an item from the stack, the value of the item pointed to by the top is retrieved, and the top is decremented.

New items are placed in the queue by incrementing the back and placing an item at that location. To remove an item from the queue, the value of the item pointed to by the front is retrieved, and the front is incremented.

For implementations of both the stack an queue, considerations must be taken to ensure that the top, back, and front variables stay within array bounds.

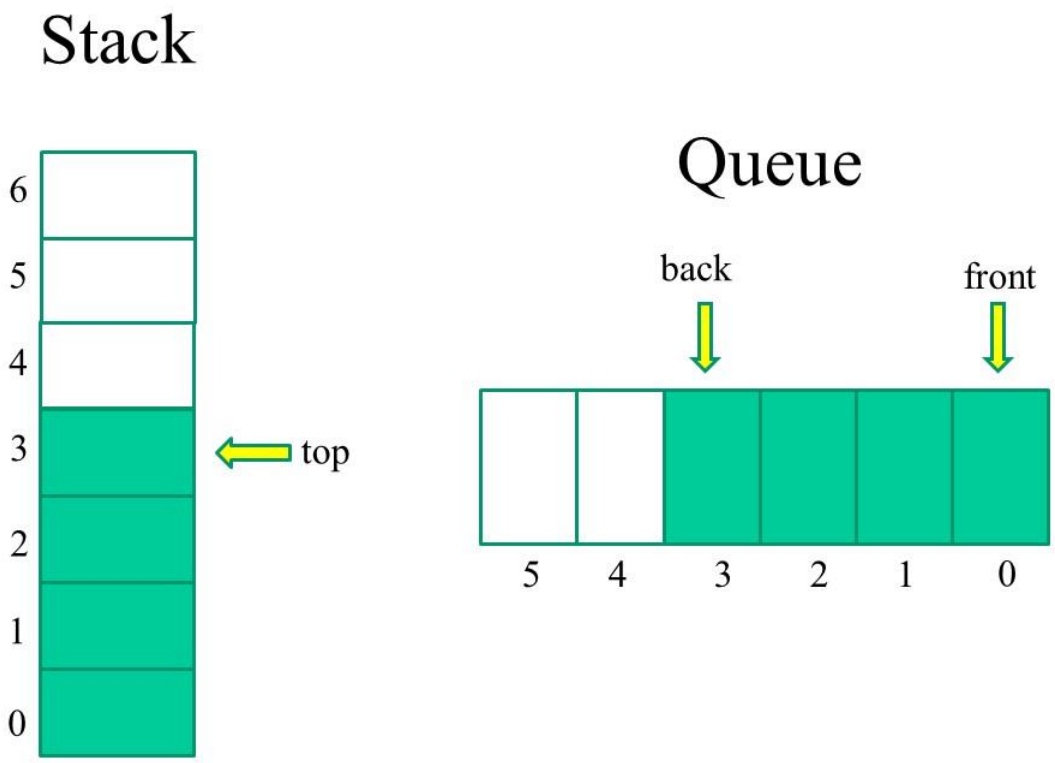


Figure 3.1 The diagram to the left illustrates a stack, to the right the queue is illustrated. The numbers to the left of the stack and below the queue are array indices. Data is added and removed from the stack at the location pointed to by the "top" variable. Data is added to the queue using the "back" variable and removed using the "front" variable.

3.1 Implementation Fundamentals

Stacks and Queues share many of the same set of basic methods. There is an initialization method that allocates memory and sets the pointer/index to the stack top, or the queue front and back, there are push and pop methods that add and remove items, and there is a method for checking if the stack or queue is empty. The major difference in stacks and queues is found in the implementation of the methods, most notably the push and pop methods. Below is a quick summary of the implementation details for a stack.

For the next several sections stacks, and stack applications will be covered. Following that will be some discussion of queues. In a later chapter, queues will be used in graph traversal algorithms, and in various sorting algorithms.

3.2 Basic methods for a Stack

- init - sets the pointer to the top of the stack (called “top”) to -1. Can also be used to allocate memory for the stack. Although -1 is not a legal index of an array, setting it up this way offers benefits that will become apparent later.
- push - increments the pointer to the top of the stack (the pointers name is usually “top”) and places an item on the stack's top.
- pop - retrieves the item pointed to by “top” and then decrements “top”.
- isStackEmpty - returns **true** if the stack is empty. An empty stack is one who’s “top” is equal to -1. If the stack is not empty, **false** is returned.

3.3 Pseudo Code for Stack Methods

The pseudo code below is for an integer stack, and it is very "C-ish". The code is very basic; there is no error checking, so it would be easy to overflow the stack, or attempt to remove items from an empty stack. The code is written this way for illustration purposes.

```
/* Global to stack methods. */
int stack[max_height];
int top;

init()
{
     /* Set stack top pointer to -1. */
     top = -1;
}

push(int x)
{
     /* Increment stack pointer. */
     top = top+1;

     /* Place x on top of stack. */
     stack[top] = x;
}



int pop()
{
     int x;

     /* Retrieve item from top of stack. */
     x = stack[top];

     /* Decrement stack. */
```

```
     top = top-1;

     return x;
}

boolean isStackEmpty()
{
     boolean empty = false;


     /* If top = -1, that indicates an empty stack. */
     if (top == -1) {
          empty = true;
     }

     return empty;
}
```

3.4 Pseudo Code for Queue Methods

The pseudo code below is very similar to that from the previous section's stack code. Once again, there is no error checking, so it would be easy to create an error by overfilling the queue, or by attempting to remove items from an empty queue. Also, note that this implementation wastes memory.

In order to see how memory is wasted, experiment with putting some items in to the queue using pushQ(); note where the front and rear of the queue are. Next, remove them using popQ(); again note the values of the front and rear of the queue. Finally, add some more items using pushQ(). What happened to the front and rear of the queue? Hint: problem can be solved if the queue is a circular queue. Please remember, if the code were to be used for anything more than learning purposes, it would prudent to remedy these deficiencies.

```
/* Global to queue methods. */
int queue[max_length];
int front, rear;

init()
{
     /* Set que front so that it greater than the rear. */
     front = 0;
     rear = -1;
}

pushQ(int x)
{
     /* Increment queue's rear pointer. */
     rear = rear+1;

     /* Place x in queue. */
     queue[rear] = x;
```

```
}


int pop()
{
     int x;

     /* Retrieve item from front of queue. */
     x = queue[front];

     /* Increment front. */
     front = front+1;

     return x;
}

boolean isQueueEmpty()
{
     boolean empty;

     empty = false;

     /* If top = -1, that indicates an empty stack. */
     if (front >= rear) {
          empty = true;
     }

     return empty;
}
```

3.5 Applications

In this course three classic applications of stacks will be studied, parentheses/equation validation, postfix expression evaluation, and infix to postfix expression conversion. Using these three applications, an equation in traditional format, infix notation (i.e. x = (2 + 3)*4) can be checked for parentheses correctness, converted to a form in which it is easy to evaluate (postfix form), and finally evaluated (i.e. x = 20). The applications are studied in a least difficult to most difficult order; the order is as follows:

- Parentheses validation
- Postfix expression evaluation
- Infix to postfix conversion

When completed, an equation in infix notation will be able to be validated, converted and solved. Below are some examples:

```
(5+4)    /* Infix equation, valid because parentheses are balanced. */
5 4 +    /* Infix to postfix format. */
9        /* Postfix expression evaluates to 9. */

((5+4)   /* Invalid, extra left parenthesis. */
```

```
(5+4))      /* Invalid, extra right parenthesis. */

5+4
5 4 +
9

(5+4)*3
5 4 + 3 *
27

5+4)*(3*7)  /* Invalid, extra right parenthesis. */

(5+4)*(3*7)
5 4 + 3 7 * *
189
```

3.51 Parentheses/Equation Validation

In this exercise we are going to make the not-so-great assumption that an equation is valid if the parentheses are balanced. For the sake of simplicity and clarity, these examples will be set using only regular parentheses, i.e., "(" and ")". Later on, in the exercises, curly braces, brackets, etc. will be considered, but in order to learn the concepts, we will keep the problem simple. One other simplification is worth noting: for now, we will work with numbers that only have a single digit so that we can avoid more complex parsing.

The first step in the process is to make sure that the difference between valid and invalid equations is established. Table 3.1 below shows several examples of equations both valid and invalid.

Expression	***Valid/Invalid***	***Remarks***
(5+3)	Valid	equal number of "(" and ")"
(5+3)*8+(2)	Valid	
(5+3)*((2+4)*3))	Invalid	Extra ")"
(((4)))	Valid	
(1/2)*(2+3)/2	Valid	
(2+3)*(8*9)/2)	Invalid	Missing "("
((2+3)*((8*9)/2)	Invalid	Extra "("
(2+3)*((8*9)/2)	Valid	

Table 3.1 Examples of valid and invalid equations.

The idea behind the validation algorithm is to make sure that every open parenthesis has a matching close parenthesis. This problem can be solved without using a stack (can you figure out how?), but since our interest is in data structures in general and stacks in particular, we present a simple stack-based solution. The algorithm uses the stack methods outlined in section 3.1. We present the algorithm below:

```
.
.
.
character stack[max_height];
charStack s; /* Declare a character stack. */
```

```
character c;

s.init();

while(there are still symbols in the equation) {
     c = getNextSymbol(equation);

     if (c == '(') {
          s.push(c);
     }
     else if (c == ')') {
          if (s.isStackEmpty()) {
               print("Error: Too many ')'");
          }
          else {
               closeParen = s.pop();
          }
     }
} /* End while. */

if (s.isStackEmpty()) {
     print("Valid equation!");
}
else {
     print("Error: Too many '(');
}
.
.
.
```

The idea is to go through the equation symbol by symbol and make sure that for each open parenthesis, there is a matching close parenthesis. The getNextSymbol() function is used to return the next symbol in the equation.

Each '(' encountered in the equation is placed on the stack. Each time a ')' is encountered, the stack is popped. If an equation is balanced, by the time each symbol has been examined, the stack should be empty. If there are any remaining '('s, then that means there was not enough ')'s to match them. If a ')' is found and the stack is empty, that indicates that there are too many ')'s. Table 3.2 shows a trace of the algorithm for the equation (2+3)*(3+2).

Symbols to be processed	***Current Symbol***	***Stack Contents***	***Remarks***
(2+3)*(3+2)		empty	start of trace
2+3)*(3+2)	(	(	open parenthesis pushed
+3)*(3+2)	2	(	
3)*(3+2)	+	(	
)*(3+2)	3	(	
*(3+2)	)	empty	closed parenthesis clears stack
(3+2)	*		
3+2)	(	(	open parenthesis pushed
+2)	3	(	
2)	+	(	
)	2	(	
	)	empty	closed parenthesis clears stack

Table 3.2 This table shows a trace for the equation validation algorithm.

3.52 Postfix Expression Evaluation

The way we usually see equations written is in what is called *infix* form. This form is easy for us to read, and it relies on left to right context, parentheses and operator hierarchy to determine the order of the operations. For example, the equation:

```
x = 2+3*4
```

could take a few different values, if we did not have rules of evaluation. We could do the adding of 2 and 3 first and end up with an answer of 20, or we could do the multiplication first and end up with a value of 24. In this type of operation, multiplication takes precedence over addition, so the answer to the equation is 24. In order to get the answer to be 20, parentheses would be used to group the add operations together, as shown below:

```
x = (2+3)*4
```

The context, parentheses, and operator hierarchy are fine for us because we can easily see the context of the equation, but that kind functionality is hard to provide the computer. The solution is to write the equation in a form that does not need the context, parentheses or hierarchy. One of the easiest ways is to write the equation in *postfix* form. The post in postfix refers to the operators following the operands that they operate on. Table 3.3 below shows several equations in both infix and postfix form. Later on, the algorithm for converting infix to postfix will be discussed, but for now the focus is understanding that we can represent equations both ways, and that it is easy to evaluate a postfix expression.

Infix	***Postfix***
3+2	3 2 +
(3+2)	3 2 +
(3+2)*5	3 2 + 5 *
(3+2)*(3-2)	3 2 + 3 2 - *
8 * (3 - 2)	8 3 2 - *

Table 3.3 This table shows infix equations and their postfix equivalents. Notice that the postfix equations have no parentheses or any other means of grouping numbers or showing operator precedence. Postfix evaluation is based solely on the order of symbols encountered.

The basic thing to remember when evaluating a postfix equation is that as every time you see an operator, you apply that operator to the two preceding operands. Once the operation is complete you replace the two operands and operator with the result. When all symbols in the expression have been exhausted, the only thing that remains is the answer. Look at the example below:

3 2 + 5 *

The first operator is the "+". As soon as the "+" is encountered, it is used on the two preceding operands, 3 and 2. The result is 5, which replaces the 3 2 + to give:

5 5 *

The next operator encountered is the "*". As soon as it is encountered, it is applied to the two preceding operands, 5 and 5, to give:

25

Now there are no more symbols, so the 25 is our answer.

Trying another example:

3 2 + 3 2 - *

The first operator is the "+", so it is applied to the preceding 3 and 2, resulting in a 5. The equation is now:

5 3 2 - *

Moving along, the next operand is the "-", so it applied to preceding 2 operands, the 3 and 2, resulting in a 1. The 1 replaces the "3 2 -" portion of the equation, resulting in:

5 1 *

And finally, the "*" operator is encountered. It is applied to the 5 and 1, resulting in 5. The 5 replaces the "5 1 *", resulting in:

5

There are no more symbols, so 5 is our answer, which turns out to be correct.

So, using the ideas of the procedure outlined in the examples above, our algorithm can be developed. In a nutshell, each operand that is encountered will be pushed onto a stack. When an operator is encountered, two operands will be popped off the stack, the operator will be applied to them, and the result will be pushed back onto the stack. This procedure will be followed until there are no more symbols in the expression. At the end of the procedure, the answer will reside at the top of the stack. Once again, the algorithm uses the stack methods outlined in section 3.1. This routine also uses the getNextToken() function whose purpose is to get the next operand or operator from the equation and the stringToNumber() function whose purpose is to convert a string like "15" into the integer 15. We present the algorithm below:

```
int stack[max_height];
int top;
int number, x, y, z;
string s;

init();

while(there are still symbols in the equation) {
      s = getNextToken(equation);
      if (s is a number) {
            number = stringToNumber(s);
            push(number);
      }
      else if (s is an operator) {
            y = pop();
            x = pop();

            if (s == "+") {
                  z = x + y;
            }
            else if (s == "-") {
                  z = x - y;
            }
            else if (s == "*") {
                  z = x*y;
            }
            else {
                  z = x/y;
            }
            push(z);
      }
} /* End while. */

print("The answer to the equation is ", z);
```

The two utility functions, getNextToken() and stringToNumber() are not difficult to write, and are very useful in many functions.

The getNextToken() function returns a string with the next token in it. To do so, it must keep track of where in the string it currently is looking. For the purposes here, getNextToken() can use a blank or an operator to know when it has found the barrier between tokens. Looking at the equations below, the tokens are shaded yellow and light blue, and the blanks are marked by underline symbols.

12_16+

3_2+3_2-*

Table 3.4 shows a trace of the algorithm.

Symbols to be processed	***Current Symbol***	***Stack Contents***	***Remarks***
3 2 + 3 2 - *		empty	start of trace
2 + 3 2 - *	3	3	3 pushed
+ 3 2 - *	2	2 3	2 pushed
3 2 - *	+	2 3	operator encountered
3 2 - *	+	3	stack popped y = 2
3 2 - *	+	empty	stack popped x = 3
3 2 - *	+	empty	z = x + y z = 3 + 2 z = 5
3 2 - *	+	5	z pushed
2 - *	3	3 5	3 pushed
- *	2	2 3 5	2 pushed
*	-	2 3 5	operator encountered
*	-	3 5	stack popped y = 2
*	-	5	stack popped x = 3
*	-	5	z = x - y z = 3 - 2 z = 1
*	-	1 5	z pushed
string empty	*	1 5	operator encountered
string empty	*	5	stack popped y = 1
string empty	*	empty	stack popped x = 5
string empty	*	empty	z = x * y z = 5 * 1 z = 5
string empty	*	5	z pushed end of trace

Table 3.4 This table shows a trace of the postfix equation evaluation algorithm. Notice that when an operator is encountered the stack is popped twice to get the operands. In order to get the order of the operands correct, the first one popped is assigned to y, and second x. If not done in this order subtractions and divisions not be calculated correctly.

So, in order to find a token, the program must load a string until it finds a blank or an operator. When it finds a blank or token the loop ends. At the beginning of the search, it must ignore blanks until it moves to the next number or operator.

The stringToNumber() function will have to do some multiplying in order to convert the characters to numbers. To get the idea, consider the equations below:

$127 = (1*100) + (2*10) + (7*1)$

$127 = (1 * 10^2) + (2*10^1) + (7*10^0)$

$127 = (\text{ascii2Number('1')} * 10^2) + (\text{ascii2Number('2')} *10^1) + (\text{ascii2Number('7')} *10^0)$

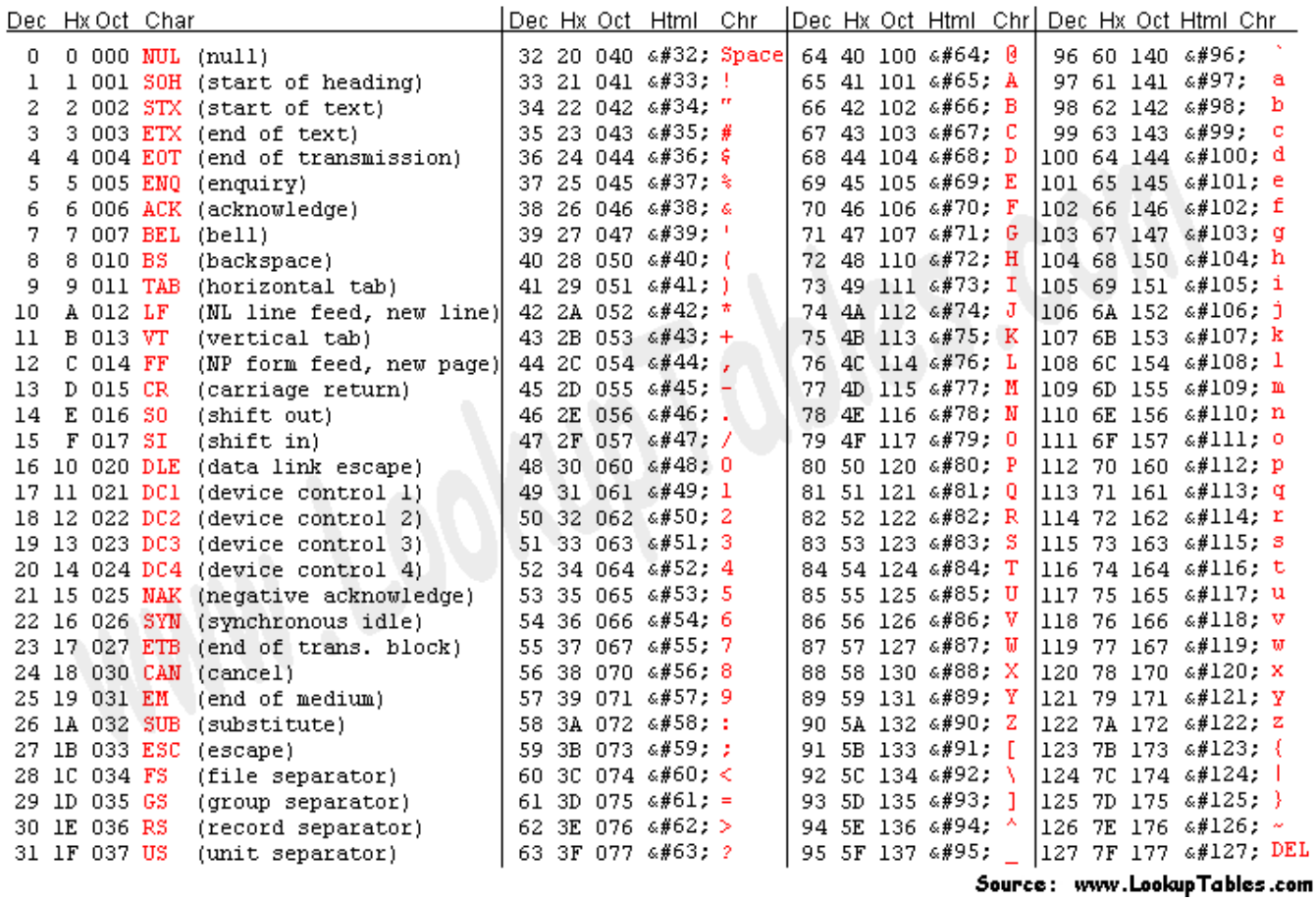
The ascii2Number() function looks up the ascii code of a numeric symbol like '0', '1', '2'.... '9', and subtracts 48. The number 48 is subtracted because the symbol '0' has the ascii code of 48, the symbol '1' is assigned ascii 49, and so forth (see figure 3.5 below). Using this technique, the single digits of the string are converted to their integer values.

If our program only had to deal with single digit numbers, we would be finished, but usually it is good to be able to work with numbers that have more than 1 digit. In order to handle this part, the digit must be multiplied by correct power of ten. So for the number 127, the '1' must be multiplied by 10^2, the 2 multiplied by 10^1, and so forth. In general, if a number has n digits, the most significant digit (the leftmost digit in the number) is multiplied by 10^{n-1}, the next digit to the right is multiplied by 10^{n-2}, and so forth.

The pseudocode snippet below shows the logic of converting a 3-digit contained in a string to an integer.

```
myNumber = “127”;
d0 = myNumber[0];
d1 = myNumber[1];
d2 = myNumber[2];

myInteger = (d0 – ‘0’)*100 + (d1 – ‘0’)*10 + (d2 – ‘0’)
```



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Figure 3.5 This is the standard ASCII table [1]. Each character ('a' through 'z', '0' through '9', 'A' through 'Z', etc.) has its own unique index. Note that character '0' has index of 48. In order to convert '0' to the number 0, we retrieve the index of '0', and subtract 48 from it.

3.52-1 A Quick Word About Software Development

When writing programs that have small functions such as getNextToken() and stringToNumber(), it is always prudent to test these functions in small test programs before integrating them into your target application. This practice will help make finding errors much easier. It is best to get all the small pieces coded, tested, and working first. Make sure the utility functions work, make sure that what is being read from the input file is being read correctly, etc. Then, each time a piece is integrated into the program, retest.

3.53 Infix to Postfix Expression Conversion

In order to convert equations from a standard format (infix) to a format that can be solved by the postfix equation evaluation algorithm, the infix to postfix expression the infix to postfix conversion routine is presented. This algorithm differs from the two previous algorithms in that it can be implemented with both a queue and stack, two stacks, or a just a stack. The algorithm presented here will use two stacks. But we will need to add a low level function to our stack object that prints out the contents of the stack from index 0 to the top of the stack.

In this algorithm, one stack is used to store the postfix expression, while the other stack is used to hold the operators until they are needed. Anytime an operand is encountered in the infix input string, it is placed in postfix stack. Anytime an operator is encountered, it is placed in the operator stack. When an open parenthesis, '(', is encountered in the infix input string, it is ignored, but when a close parenthesis, ')' is encountered it is triggers the algorithm to pop an operator off the operator stack, and push it onto the postfix stack. When all symbols in the input string have been exhausted, each of the operators in the operator stack is popped and then pushed onto the postfix stack. The algorithm is presented in pseudo code below.

Looking at the pseudo code, make sure to note that each of thestacks will need to be initialized. Also, one of the features of this algorithm is that there is no conversion from character data to integer data; it is this way because there is no evaluation occurring. One thing to note, we are assuming that all numbers are a single digit, for now. We do this so that we do not get caught up in a parsing exercise. An excellent exercise would be to do the parsing required, but only after the proof of concept is completed using single digit numbers. This is another good software trick; solve simple versions of the problem before moving on.

Of final note is the `showStack()` method called at the end of the routine. This method shows the contents of the postfix stack from index 0 to the index pointed to by the "top" variable.

```
.
.
.
characterStack op[max_height];
characterStack post[max_height];
character s, myOp;

op.init();
post.init();

while(there are still symbols in the equation) {
     s = getNextSymbol(equation);

     if (s is a number) {
          post.push(s);
     }
     else if (s is an operator) {
          op.push(s);
     }
     else if (s == ')') {
          myOp = op.pop();
          post.push(myOp);
     }

} /* End while. */

While(op.isStackEmpty() == false) {
     myOp = op.pop();
     post.push(myOp);
}

print("the postfix equation is ");
post.showStack();
```

To illustrate the algorithm working, Table 3.5 below shows a trace is using the sample equation:

(3+2)*5

Before implementing the algorithm in code, it is good practice to trace through several examples in order to insure a detailed understanding. Also, incremental integration and testing of each method helps to reduce unforeseen errors.

Symbols to be processed	***Current Symbol***	***Postfix Queue Contents***	***Operator Stack Contents***	***Remarks***
(3+2)*5	Empty	Empty	Empty	Start of process
3+2)*5	(	Empty	Empty	Ignore open parenthesis
+2)*5	3	3	Empty	Push the '3' to the Postfix stack
2)*5	+	3	+	Push '+' operators to Operator Stack
)*5	2	3 2	+	'2' pushed to postfix queue
*5	)	3 2 +	Empty	')' triggers operator stack to be popped, contents is pushed to postfix stack.
5	*	3 2 +	*	'*' pushed to operator stack
Empty	5	3 2 + 5	*	'5' pushed to postfix stack
Empty	Empty	3 2 + 5 *	Empty	Upon end of infix equation, operator stack is popped, contents is pushed to postfix stack, until operator stack is empty

Table 3.5 This table shows a trace of the infix to postfix equation conversion algorithm using the equation (3 +2)*5.

3.5 Exercises

1. Implement the basic methods of the stack in procedural code. Set the stack up so that it can work with characters. Use the push and pop methods to load and unload the following strings from the stack, print out the results.
   a. "bob"
   b. "aet nomel"
   c. "snoino dna revil etah I"
   d. "SELUR LOBOC"
2. Make sure that your code from exercise 1 is using objects and retest the code.
3. Implement the basic methods of the queue, use either procedural or object oriented methods. Use the push and pop methods to load and unload the following strings from the queue, print out the results.
   a. "bob"
   b. "eat too much"
   c. "I love greasy food"
   d. "FORTRAN 77 RULES"
4. Manually solve the following problems. First do a parentheses check. If the equation passes the parentheses check, convert the equation to postfix, then evaluate the postfix equation.
   a. (5+3)
   b. (5+3)*8+(2)
   c. (5+3)*((2+4)*3))
   d. (((4)))
   e. (1/2)*(2+3)/2
   f. (2+3)*(8*9)/2)
   g. ((2+3)*((8*9)/2)
   h. (2+3)*((8*9)/2)
5. Write an algorithm that implements parentheses/equation validation without using stacks or queues.
6. Implement the following algorithms in code using the stack and queue methods developed in exercises 1, 2 and 3:
   a. Parentheses/equation validation
   b. Infix to postfix expression conversion
   c. Postfix equation evaluation

   Create a text file with the following data input.

```
(5+3)
(5+3)*8+(2)
(5+3)*((2+4)*3))
(((4)))
(1/2)*(2+3)/2
(2+3)*(8*9)/2)
((2+3)*((8*9)/2)
(2+3)*((8*9)/2)
```

Using your programs, first do a parentheses check. If the equation passes the parentheses check, convert the equation to postfix, then evaluate the postfix equation.

Chapter 4. Recursion

Recursion is a powerful tool that can be used to create elegant solutions to complex problems. So far in our discussion of data structures we have not needed it, but recursion will prove to be invaluable in the upcoming chapters. We will see that it can be used for finding the end of a linked list, but where recursion will really make a difference is binary tree and graph traversal.

Key questions include:

- What is recursion, and when can use it?
- Is it good to use it whenever we can, or are there instances when non-recursive solutions are preferred?

Simply stated, a recursive function (a recursive function is one that uses recursion) is a function that calls itself. Consider the factorial of a number. It is found by multiplying the number times one less than the number, then multiplying that product times two less than the original number and so forth, counting down all the way to 1. Below we show the pseudo code for a factorial function.

```
int factorial(int num)
{
     int j, prod;

     prod = 1;
     for(j = 1;j<=num;j++) {
          prod = prod * j;
     }

     return prod;
}
```

Table 4.11 below shows a trace for the call : `factorial(7)`.

j	***prod***	***Remarks***
-	1	just before start of for loop
2	2	prod multiplied by j for the first time
3	6	prod = prod * j
4	24	prod = prod * j
5	120	prod = prod * j
6	720	prod = prod * j
7	5040	prod = prod * j
-	5040	prod is returned to calling program

This is an *iterative* function. It is described as iterative because it arrives at a solution by using a for loop structure to construct a solution one step at a time, with a step being defined as a single pass through the for loop. Each pass through the loop is called an iteration, thus the description "iterative solution". Equation 4.12 below shows the mathematical approach used by the iterative solution described by the pseudocode above.

```
n! = 1 * 2 * 3 * ..... * (n - 2) * (n - 1) * n
```

Equation 4.12 shows the arithmetic used by the iterative factorial function.

Note that most math literature would describe the factorial function by starting at n and working down to 1, as shown below.

```
n! = n * (n-1) * (n-2) * . . . . * 2 * 1
```

There is another mathematical approach to the factorial problem. The equation series 4.13 illustrates this approach. Here we define the factorial of a number by multiplying the number by the factorial of one less than the number. This is a circular definition.

```
n! = n * (n - 1)!

(n - 1)! = (n - 1) * (n - 2)!
(n - 2)! = (n - 2) * (n - 3)!
.
.
.
.
(n - (n - 1))! = 1! = 1
```

Equation series 4.13 shows the arithmetic used by the recursive factorial function.

For this approach to work, there must be some point at which the process of referring to oneself ends. For the factorial this occurs when we reach 1, because the factorial of 1 is defined as 1. The pseudo code below illustrates the recursive version of the factorial function. Notice that there are two distinct parts of this function, one that checks for an end condition, and another that defines the answer using a recursive call, a call to itself.

```
int factorial(int num)
{
     if (num == 1) {
          return 1;
     }
     else {
          return num * factorial(num - 1);
     }
}
```

The recursive implementation's code is very different from that of the iterative solution. There are no loops in this implementation. Instead, it follows the equation series 4.13. The recursive code is logically simple and elegant, no loops, no indices, no temporary variables. Table 4.14 below shows a trace of the code for the call : `factorial(7)`.

value of num for this call to factorial	***method call***	***return value***	***Remarks***
7	factorial(7)		initial call in calling program
7	7 * factorial(6)		first recursive call
6	6 * factorial(5)		recursive call
5	5 * factorial(4)		recursive call
4	4 * factorial(3)		recursive call
3	3 * factorial(2)		recursive call
2	2 * factorial(1)		recursive call
1	no method call	1	end condition is met
2	no method call	2	returning 2 * factorial(1)
3	no method call	6	returning 3 * factorial(2)
4	no method call	24	returning 4 * factorial(3)
5	no method call	120	returning 5 * factorial(4)
6	no method call	720	returning 6 * factorial(5)
7	no method call	5040	returning 7 * factorial(6) to initial calling program

Table 4.14 shows a trace of what occurs when a call to the recursive function factorial(7) is made. Notice that the function recursively calls itself until it reaches the end condition. After that it works itself out of the calls, each time making one more of the required multiplications.

The recursive code is in a way simpler than the iterative code, but notice that the trace required about twice the number of steps. Half of the steps were making recursive calls until the end condition was met, the other half was taken in working out of those calls and along the way calculating the answer.

So, which is better? In general, recursion is the choice if by using it the code can be simplified, and not many recursive calls are expected. If the code is not significantly simpler and many recursive calls are expected, the iterative solution is preferred.

The reason is that each time a call to a method is made, the system must store the calling function's vital information, such as values of all local variables, where the program will return when the method call is complete, etc. and memory must be allocated to house all of these same parameters for the method that is being called. If a recursive routine makes 5 calls to itself, this process must occur 5 times. Suppose each call requires 1 kilobyte of memory, 5 calls would only require 5k of memory, not really a big deal. But 1000 recursive calls would change the story. The overhead would be manageable on today's memory rich computing systems, but for smaller systems, such as embedded microprocessors, this would be a serious mistake.

4.1 Summary

The use of recursion can provide simple and elegant solutions to complex problems. Many times, these solutions are preferable to iterative solutions because they reduce the complexity of the code. In general, recursion trades coding simplicity for time and memory inefficiency. Iterative solutions are often better for memory intense problems or problems in which there would be many recursive calls.

4.2 Exercises

1. What are the advantages of using recursive solutions? Disadvantages?
2. Develop both iterative and recursive algorithms for the summation of a number.
    a. Show traces of each.
    b. How many steps for the iterative trace?
    c. How many steps for the recursive trace?
3. Implement your algorithms from exercise 2 using C/C++ or JAVA.
4. Write a recursive routine called logMeR that returns the number of times that an input number can be divided by 2.
    a. Initially, let the input be a power of 2, examples include 32, 64, 512, etc.
    b. Change your routine so that any number, not just powers of 2 can be used.

Chapter 5. Complexity

Complexity can refer to different things when it comes to software and algorithm development. Intuitively we think of complexity as a measure of how many moving pieces a machine may have, or how intricate, difficult to understand, and/or convoluted the inner workings/logic of a machine or piece of software is. That being said, when it comes to software and computer science, when we talk about program complexity, we are primarily talking about program run time complexity. That is, given a data set size of *N*, what is the relationship between the data set size and the time it takes for the program to run.

For example, suppose that we want to write an equation that describes the run time complexity of the code snippet below:

```
x = 0;
y = 0;
z = 0;
for(j=0;j<N;j++) {
     x = x+1;
     y = y+2
     z = x-y;
}
print(x, y, z);
```

A good start could be the following equation:

run time = (3*N) + 4

- The (3*N) portion of the equation accounts for the loop portion of the code snippet. The loops executes N times and there are 3 statements in the loop, thus 3*N.
- The 4 portion of the equation comes from the observation that there are 4 statements outside of the loop.

One thing to remember is that the equation really looks like this:

run time = ((3*N) + 4) * *machine instruction run time*

When discussing complexity issues, we leave off the machine instruction run time because we treat it as a constant. Really it reflects the average time an instruction runs on our machine, but if we are running algorithms of different complexity on our machine, this time is constant for all those algorithms on our machine, thus we do not include it in our calculations.

Let's look at how the input size effects the run time. Table 5.1 below shows the value of run time for various sizes of N.

Input size - N	***Run time Complexity - run time = (3*N) + 4***
1	7
2	10
4	18
10	34
100	304
1000	3004
10000	30004

Table 5.1 This table shows the effect of the input size on the runtime complexity. Notice that as the size of N grows, the effect of the constant 4 decreases.

Looking at the table, it can be seen that when N is small that the constant 4 contributes greatly to the total runtime. Table 5.2 below shows the percent of the contribution the constant 4 makes in relation to input size. As can be seen from the table, once input size exceeds 100 elements, the 4 contributes 1 percent or less of the program's total runtime.

Input size - N	***Percent of run time that the constant 4 is.***
1	57
2	40
4	22
10	12
100	1
1000	0.001
10000	0.0001

Table 5.2 This table percent of run time that the constant 4 is.

Let's look at another code snippet:

```
x = 0;
y = 0;
z = 0;
for(j=0;j<N;j++) {
     for(k=0;k<N;k++) {
          x = x+1;
          y = y+2
          z = x-y;
     }
}
print(x, y, z);
```

This piece of code is almost identical to the previous one, except that it has two nested loops instead of a single loop. So, for each iteration of the outer loop, in inner loop goes through N iterations. An equation describing this snippets runtime follows:

run time = N*(3*N) + 4

$= 3N^2 + 4$

- The (3*N) portion of the equation accounts for the loop portion of the code snippet. The loops executes N times and there are 3 statements in the loop, thus 3*N.
- We multiply N times (3*N) to take into account that for each iteration of the j loop, the k loop iterates N times, and there are N iterations of the j loop.
- Just as before, the 4 portion of the equation comes from the observation that there are 4 statements outside of the loop.

Table 5.3 below shows the relationships between input size and time complexity for the nested for loops in the previous code snippet. Notice that the 4 statements that are not in the for loops contribute just barely 1% of the total complexity at an input size of N = 10. Another feature to note is that the fact that the input size being squared has a tremendous influence on the run time growth rate. In fact, it can be argued that the growth rate of the time complexity is influenced much more strongly by squaring the input size than by multiplying it by 3.

Input size - N	***Run time Complexity run time*** = $3N^2 + 4$	***Percent of run time that the constant 4 is.***
1	7	57.14286
2	16	25
4	52	7.692308
10	304	1.315789
100	30004	0.013332
1000	3000004	0.000133
10000	3E+08	1.33E-06

Table 5.3 This table shows the effect of input size on growth rate for a code snippet with nested loops.

5.1 Big O Notation

In the previous sections an algorithm's time complexity as a function of input size was discussed. As can be seen in the tables in the previous section, as input size grows, the influence of constants in the growth equation decreases. For these reasons, we often leave out these constants and use just the portions of the equation that have the most influence on the rate of time complexity growth. When equations are written in this manner, it is referred to as "Big O" notation. In Computer Science when we say that "the order of an algorithm is N^2 we are referring to the algorithms run time complexity in Big O notation.

In earlier sections we had the equation :

run time = (3*N) + 4

In Big O notation this becomes
run time = O(N)

or just:

O(N)

Note: many times this is written using the small 'n' – O(n), it has the same meaning.

Likewise we had the equation:

run time = $3N^2 + 4$

In Big O notation this becomes:

$O(N^2)$

Table 5.4 below shows some examples of time complexity equations and their Big O counterparts.

Time complexity equation	Big O Notation
rt = N + 7	O(N)
rt = N^2 + 6N + 4	$O(N^2)$
rt = 22N + 7 + .25N^3	$O(N^3)$
rt = 2N^2 + 100N + 3000	$O(N^2)$
rt = N*(N/2) + 1 = $N^2/2$ + 1	$O(N^2)$

Table 5.4 The time complexity equation represents the actual number of time units used by a *particular* algorithm, while the Big O notation represents the general runtime complexity of the algorithm.

Looking at the last equation in the table, can you imagine a configuration of code that would fit this runtime equation? The N^2 portion of the equation should suggest a set of nested loops, but what about the part where it is divided by 2? The code snippet below fulfils the equation. Make sure you understand why. What is the value of z after the code snippet is complete?

```
z = 0;
for(j=0;j<N;j++) {
     for(k=0;k<N/2;k++) {
          z = z+1;
     }
}
```

5.3 More Time Complexity: Log base two

On common way to solve problems is break down the data set in halves and work on each separate half. Many algorithms rely on this technique, including binary searches,

merge sorts, quick sort (not exactly half), etc. By doing this, a large degree of efficiency can be gained, because each time a comparison is made, roughly half the area to be searched is eliminated. To get a good feel for this concept, consider the old game of guessing a number between 1 and 100. In this game, one player gets a number in their head, and the other player tries to figure it out in as few guesses as possible. The player with the number in their head has to tell the person guessing whether they are too high or too low. So, each time a guess is made, a large chunk of numbers is eliminated from set of possible numbers, roughly half of them.

For example, imagine that the number to be found between 1 and 100 is 71. Your job is to guess numbers until you determine the value of the number. Each time you guess, you are told if your number is too high or too low. If the first guess is 50, then you are told "too low". Then you may guess 75, to which the reply is "too high". Your next guess may be a number like 67, to which the reply is "low". And so forth. Table 5.5 below shows the progression of guesses when trying to find the number 71

Number of Guesses	***Guess***	***High or Low***
1	50	low
2	75	high
3	68	low
4	71	You found it!

Table 5.5 This table shows the progress of a number guessing game when one person thinks of a number between 1 and 100 and another person tries to figure it out. In this case, the number to be guessed was 71. Notice the guesses follow a binary search pattern.

What if the number to be guessed were a harder one than 71? Lets take another example, this time using a hard number like 47. Table 5.6 below shows the progression of finding the 47 in the number guessing game.

Number of Guesses	***Guess***	***High or Low***
1	50	high
2	25	low
3	37	low
4	43	low
5	46	low
6	48	high
7	47	You found it!

Table 5.6 This table shows the progress of a number guessing game when one person thinks of a number between 1 and 100 and another person tries to figure it out. In this case, the number to be guessed was 71. Notice that the values of guesses follow a binary search pattern.

Thinking of this game, you may be able to determine the maximum number of guesses a player may need, that is if the player followed a good strategy.

It took 7 guesses to find the 47. When the algorithm approached the target number, it still jumped around on either side of it a bit. Why? The short answer, if you have not already guessed it, this is a binary search, and the guesses are calculated by dividing the

set of numbers in half each time. For example, the initial guess is 50, which is splits the set of numbers in half. If it is learned that the number is less than 50, the next guess is 25, if the number is greater, the next guess is 75. Both 25 and 75 are mid-point of their respective number sets.

The real question is how many times can you split something in half? That is, how many times can you cut a something in half, without cutting one the data sets members in half. Looking at the set A of size 16 numbers below, we can see that the first time we half the set, we get two sets, B and C, of size 8:

set A = {1, 2, 3 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}

set B = {1, 2, 3 4, 5, 6, 7, 8}, set C = {9, 10, 11, 12, 13, 14, 15, 16}

now suppose we halve set B, resulting in sets D and E of size 4:

set D = {1, 2, 3 4}, set E = {5, 6, 7, 8}

and now we halve set D which gives sets F and G of size 2:

set F = {1, 2}, set G = {3 4}

and splitting set F into two sets of size 1 results in sets H and I:

set H = {1}, set I = {2}

Now we are at the point at which we cannot split either set in half without cutting one of member in half, so we are finished. In order to get to the point in which there was only 1 member in the sets H and I, it took 4 times of halving the data sets. So, from an initial size of 16 we went to a size of 1 in four steps, as follows below:

16 / 2 = 8

8 / 2 = 4

4 / 2 = 2

2 / 2 = 1

The opposite works also:

2 * 1 = 2

2 * 2 = 4

2 * 4 = 8

2 * 8 = 16

Either way, it takes 4 steps to go from 16 to 1 or from 1 to 16. Looking again at the above progression we see that the following statements are true:

Recall that the log of a number is essentially the exponent in which the base must be raised to be equal to that number. For example $\log_{10}100 = 2$, because 10 must be raised to the power of 2 to equal 100, that is $10^2 = 100$.

2 * 1 = 2, which is the same as 2^1, $\log_2 2 = 1$

2 * 2 = 4, which is the same as 2^2, $\log_2 4 = 2$

2 * 4 = 8, which is the same as 2^3, $\log_2 8 = 3$

2 * 8 = 16, which is the same as 2^4, $\log_2 16 = 4$

If it is a number set of size N, the answer is $\log_2 N$. In the previous example in which we were trying to guess the number 47, the number set in which we were searching was size 100, thus the maximum number of searches is $\log_2 100$ which is between 6 and 7, which is about how many guesses it took.

Table 5.7 below shows the relationship between logs of base 2 and numbers of the power of 2. One thing to remember about logs is that you are asking this question, "To what power must I raise the base to get the number you are trying to find the log of?". For example, if you want to find the log (base 2) of 64, you ask yourself "What power must I raise 2 in order to get 64?". If you look at the table, you see that 2 raised to the 6^{th} power is indeed 64.

exponent	$2^{exponent}$	$\log_2(2^{exponent})$
0	$2^0 = 1$	$\log_2 1 = 0$
1	$2^1 = 2$	$\log_2 2 = 1$
2	$2^2 = 4$	$\log_2 4 = 2$
3	$2^3 = 8$	$\log_2 8 = 3$
4	$2^4 = 16$	$\log_2 16 = 4$
5	$2^5 = 32$	$\log_2 32 = 5$
6	$2^6 = 64$	$\log_2 64 = 6$
7	$2^7 = 128$	$\log_2 128 = 7$
8	$2^8 = 256$	$\log_2 256 = 8$
9	$2^9 = 512$	$\log_2 512 = 9$
10	$2^{10} = 1024$	$\log_2 1024 = 10$

Table 5.7 This table shows the relationship between exponents, $2^{exponent}$, and the log base 2 of that number.

It can be noted that a binary search is of order $\log_2 N$. Looking at the code snippet below, we can see that it is of order $N\log_2 N$. The outer loop executes about $\log_2 N$ times, and the inner loop executes N times.

```
x = 0;
for(j=1;j<=N;j=j*2) {
     for(k=0;k<N;k++) {
          print(x);
          x = x + 1;
     }
}
```

5.4 Efficiency Calculations and Comparisons

In previous sections the idea of run time complexity was introduced as a tool to determine the efficiency of an algorithm. In this section these ideas will be put to use, along with the concept of average efficiency.

To demonstrate the idea of average efficiency, consider an array of numbers in no particular order (they are not sorted in any way). We want to know the expected run time efficiency for finding a number in the array. That being said, in order to find the number, since the array is not sorted, the only way is sequentially search the array. The pseudo code below illustrates our search.

```
found = false;
j = 0;
while((found == false)&&(j < N)) {
     if (array[j] == myNumber) {
          found = true;
     }
     else {
          j = j + 1;
     }
}
```

This above code will search the array until it finds the number, or until the entire array has been searched. Since there is no order to the array, the number that is being looked for can be in any of the N positions of the array. The best-case scenario is if our number is in the first position of the array, i.e. position 0 of the array; the worst case if it is in the last position, i.e. position N-1. To find the average complexity the best case and worst case number of comparisons can be averaged:

$$\text{average time complexity} = \frac{\text{best case + worst case}}{2}$$

$$\text{average time complexity} = (1 + N)/2$$

average time complexity = O(N)

Now suppose that the numbers are arranged from low to hi and we use a binary search to find the number. As shown in the earlier sections, the binary search is of order $\log_2 N$. Which search is more efficient, the sequential search or the binary search? Since $\log_2 N$ is always less than N, (unless N = 1), the binary search is more efficient. Looking at table 5.8 below, it can be seen that as the size of the input set increases, the binary search's advantage in efficiency grows rapidly.

Array size to be searched	Max Sequential Search Comparisons	Max Binary Search Comparisons
16	16	4
32	32	5
64	64	6
128	128	7
256	256	8
512	512	9
1024	1024	10
2048	2048	11
4096	4096	12
8192	8192	13

Table 5.8 This table shows the huge advantage that a binary search of an ordered list has over a sequential search.

Let's take an example. Suppose we have an unordered list of 1024 numbers, and we want to know the average number of comparisons that will be made when searching the list.

Best case = 1
Worst case = 1024

$$\text{Average case} = \frac{1 + 1024}{2} = \frac{1025}{2} \approx 513$$

Now suppose that the list is ordered from low to high and we want to use a binary search to find our number. In this case the best/worst case does not really apply because the likelihood of our number being picked in the beginning of the search process is not the same as it is in the end of the process. For this reason, we will use the worst-case scenario as our metric.

worst case = $\log_2 1024 = 10$

The conclusion is that the binary search is far more efficient, but the array must be arranged from low to high or vice versa for it to work. For this condition to be present,

the array must be sorted, or built in order. Later chapters will describe efficient methods of building and searching the list of numbers.

5.5 A Quick Guide to Algorithm Analysis

In the previous sections various loop constructs and their runtime complexities were discussed. In this section, a number of examples of pseudo code will be provided along with their runtime complexities. Most of these examples are straight forward; a couple are not. Being able to recognize the order of complexity is a vital skill in both design and analysis of code, so a good grasp of these concepts is important.

5.5.1 Single for-loop, order O(N)

The code below shows a single for loop that runs through its entirety; that is there is no premature exit:

```
x = 0;
for(j=0;j<N;j++) {
     x = x+j;
}
```

Runtime: N+1
Order: O(N)

This code shows two sequential loops:

```
/* Increasing loop */
x = 0;
for(j=0;j<N;j++) {
     x = x+j;
}
/* Decreasing loop */
for(j=0;j<N;j++) {
     x = x-j;
}
```

Runtime: N + N+1 = 2N + 1
Order: O(N)

The next loop sometimes exits without completing:

```
found = false;
j = 0;
while((found == false)&&(j < N)) {
     if (array[j] == myNumber) {
          found = true;
     }
     else {
          j = j + 1;
     }
}
```

Runtime: $(1 + N)/2 \approx N/2$
Order: O(N)

5.5.2 Double Nested for-loop $O(N^2)$, but not always

The code below shows a set of for loops that are nested, i.e., one inside the other. They run through their entirety; that is there is no premature exit. For each iteration of the outer loop, the inner loop runs N times. Since the outer loop runs from 0 to N-1, it also runs N times, so the major component of the run time equation is N^2.

```
x = 0;
for(j=0;j<N;j++) {
     for(k=0;k<N;k++) {
          x = x + 1;
     }
}
```

Runtime: $1 + N*N = N^2 + 1$
Order: $O(N^2)$

The above loop is straight forward, each loop incrementing by 1 (a sequential incrementation of the loop variable), giving a time complexity of order $O(N^2)$, but recall the code snippet from earlier in the chapter shown below. In this set of nested loops, the outer loop increments by doubling the index (j = j*2), resulting in a time complexity of $\log_2 N$. The inner loop works in the usual way, incrementing by 1, thus resulting in O(N). For each of the $\log_2 N$ iterations of the outer loop, the inner loop iterates N times, resulting in total complexity of $O(N\log_2 N)$.

```
x = 0;
for(j=1;j<=N;j=j*2) {
     for(k=0;k<N;k++) {
          print(x);
          x = x + 1;
     }
}
```

Runtime: $1 + \log_2 N * (N*2) = 2N\log_2 N + 1$

Order: $O(N\log_2 N)$

The nested loops below both increment by 1, but the inner loop starts at j+1 index and then goes to the n-1 element. So, this is clearly not a case of the inner loop executing the same number of times as the out loop. That being said, it is not like the previous example either, so the question is, what is the run time complexity?

```
x = 0;
for(j=1;j<=N;j++) {
     for(k=j;k<=N;k++) {
          x = x + 1;
     }
}
```

Let's look at the number of times the inner loop runs. For a value of N=10, table 5.9 below shows the number of iterations that the inner loop (for k loop) iterates. Note that the j loop starts at 1 and goes through N (*Most of time for loops start at 0 and stop at N-1. Starting at 0 and going to N-1 is similar, but for explanation purposes it is easier to start at 1 and go to N.)*

Value of j index	1	2	3	4	5	6	7	8	9	10
Number of times k loop iterates	10	9	8	7	6	5	4	3	2	1

Table 5.9 This table shows the number of times the inner loop executes for each value of the outer loop. Notice it is a decreasing sequence.

We want to know how many times the statement in the k loop is going to be executed. Looking at the table, we see that the k loop executes N times the first time it is entered, then N-1, then N-2, and so forth. It is a decreasing sequence. To find the total number time the statement in the k loop executes the sum of the iterations of the k loop is taken.

$$\text{k loop iterations} = 10 + 9 + 8 + \ldots\ldots + 3 + 2 + 1$$
$$\text{k loop iterations} = \sum i \text{ where } i = 10 \text{ to } 1.$$

The goal is to come up with what is called a closed form solution, that is, an algebraic equation so that the number of iterations can be expressed in terms of N. While the summation is accurate, the algebraic expression is easier to work with, and also provides the correct form in order to find the runtime and order of the algorithm.

There are a couple of easy ways to convert the summation into an algebraic equation.

To do this we will rely on the communitive property of addition, meaning that if you are finding the sum of a group of numbers, the order in which you add them does not change sum.

The first requires an observation of the sequence of numbers that is as follows.

If we sum the first number with the N_{th} number it results in (1 + N)
If we sum the second number with the $(N-1)_{th}$ number it results in (2 + (N-1)) = (1 + N)
If we sum the third number with the $(N-2)_{th}$ number it results in (3 + (N-2)) = (1 + N)
...
...
Now we can keep doing this until we reach the middle of the list of numbers as follows:

Sum the $(N/2)_{th}$ number with the $(N/2 + 1)_{th}$ number it results in (N/2 + N/2 + 1) = (N+1)

Now we are out of numbers to sum. We stopped in the middle of list so there are (N/2) numbers summed, and each one was equal to (1+N). From this the following equation can be written:

$$\text{iterations} = (N/2)*(N+1)$$

$$\text{iterations} = \frac{N*(N+1)}{2}$$

This technique is a variation of Gausses' [3] method. So, the following can be said about the runtime complexity and order of the algorithm:

Runtime: (N * (N*1))/2
Order: $O(N^2)$

The key observations in the above discussion are:

1. When summing a series of numbers, you can sum them in any order and still get the same answer (commutative property of addition).
2. As a result of observation 1, summing the last number and the first number results, in (N+1), summing the second to last number and the second number results in (N+1), and so forth.
3. Since the numbers are summed in pairs, there are N/2 of the numbers, which leads to the quantity:

$$\text{iterations} = \frac{N*(N+1)}{2}$$

A similar approach can be taken by summing two rows of numbers, one ascending, the other descending, as shown below:

$$\sum_{1}^{N} i = 1 + 2 + 3 + 4 + 5 + 6 + \ldots\ldots + (N-1) + N$$

$$\sum_{N}^{1} i = N + (N-1) + (n-2) + \ldots\ldots\ldots\ldots\ldots + 2 + 1$$

These two sequences are equivalent because the commutative property of addition, that is summing of a sequence of numbers is order independent.

$$\sum_{1}^{N} i = \sum_{N}^{1} i$$

Now if we sum the two sequences of numbers we get:

$$\sum_{1}^{N} i + \sum_{N}^{1} i = (1 + 2 + 3 + \ldots\ldots + (N-1) + N) +$$
$$(N + (N-1) + \ldots\ldots\ldots + 2 + 1)$$

Again, relying on the commutative property of addition, we pair the numbers to create n quantities of (n+1).

$$\sum_{1}^{N} i + \sum_{N}^{1} i = n*(n+1)$$

Then doing the algebra:

$$\sum_{1}^{N} i + \sum_{N}^{1} i = 2*\sum_{1}^{N} i = 2*\sum_{N}^{1} i$$

$$2*\sum_{1}^{N} i = n*(n+1)$$

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

This proof is very similar to the previous proof in that it relies on the commutative property of addition. One more observation that is worth noting and can come in very useful:

$$\sum_{0}^{N} i = \sum_{1}^{N} i$$

5.5.3 Nested for-loops, order O($N^{\text{number of nested loops}}$)

As seen in the previous section, when two for loops were nested using loop variables that increment by 1 (or any other linear count), the order of the loops was O(N^2). In general, it is safe to say that long as the loop counters increment by a fixed count, the time complexity will be to the power of the number nested loops. For example, a nesting of 3 for loops is O(N^3), a nesting of 4 for loops is O(N^4) and so forth.

5.5.4 Recursive Calls

The time complexity of recursive calls can be evaluated in a similar way to that of loops. While they look different, there complexity is similar, except the notable fact that they require substantially more memory than iterative solutions (as briefly described in the previous chapter). Looking at the code snippet below:

```
int recurseMe(n, z)
{
     print(n);

     if (n == z) {
          return 0;
     }
     else {
          return(1 + recurseMe(n+1, z));
     }
}

note: z is a constant > n
```

Intuitively it can be seen that the function is going to call itself z - n times counting up from n to z and then return a sum equal to z - n. The common way to describe this behavior is with a *recurrence relation*. A recurrence relation is an equation that can be used to analyze run time complexity. The recurrence relation for the recurseMe() is shown below:

```
T(n) = 1 + T(n-1)
```

The 1 in the equation represents the print(n) statement, and the T(n-1) represents the recursive call recurseMe(n+1, z). By expanding the recurrence relation, we can begin to see what the time complexity of the function is:

```
T(n) = 1 + T(n-1)
T(n) = 1 +    1 + T(n-2)
T(n) = 1 +    1 +   1 +T(n-3)
T(n) = 1 +    1 +   1 +  1 + T(n-4)
```

The expansion will stop at when n is equal to z by virtue of the if statements condition, so we can write:

```
T(n) = 1 +    1 +   1 +  1 +  ......... + 1 + T(z-n)
```

The T(z-n)th call ends the sequence of calls so we can substitute a 1 for its quantity (because of the print statement)

```
T(n) = 1*(z-n)
T(n) = z-n

0 < z-n < z which is O(N)
```

Given this, what is the order of recurseMe()? It is O(N) because it is always somewhere between 1 and z (remember n < z), and more importantly, each recursive call is incrementing the variable n by 1, so it is situation in which there is a constant, sequential incrementation of the variable that controls the number of calls.

Looking at the pseudo-code below we see a recursive function that has a loop in it before the recursive call. The loop is a regular for loop with a single statement in it that increments the variable x. Before the recursive call, n is halved using integer division, meaning that when n = 1 and it is halved, n will become 0. Once n becomes 0, the sum x is returned through the recursively to the until it reaches the original call.

```
int recurseMeAgain(n)
{
     x = 0;
     if (n > 0) {
          for(j=0;j<n;j++) {
               x = x+1;
          }
          n = (int)(n/2);
          recurseMeAgain(n);
     }
     return x;
}
```

The recurrence relation for this code is shown below:

```
T(n) = n + T(n/2)
```

Expanding the relation:

```
T(n) = n + T(n/2)
T(n) = n + (n/2 + T(n/4) )
T(n) = n + (n/2 +(n/4 + T(n/8) ) )
T(n) = n + (n/2 +(n/4 + (n/8 + T(n/16) ) ) )
T(n) = n + (n/2 +(n/4 + (n/8 + (n/16 + T(n/32) ) ) ) )
```

Remembering that eventually n will become 0, we will simplify the equation.

```
T(n) = n + n/2 + n/4 + n/8 + n/16 + T(n/32)
T(n) = n + (15/16)*n + T(n/32)
```

Realizing that the quantity represented by T(n/32) is less than n/16, we can go ahead and sum the terms and write a relationship that represents the run time complexity.

$T(n) = 1^{15/16}n + T(n/32)$

giving

$T(n) < 2n$ which is $O(N)$

Look at this very closely. It is tempting to label this as $O(\log_2 N)$ because of the way n is halved before the recursive call. To be fair there are $\log_2 N$ recursive calls made, but the loop itself iterates in a sequential fashion, just under 2N times.

Looking at one more example, recurseMeYetAgain() changes the decrement of n right before the recursive call to a standard n = n-1, vs. n = n/2 from the previous example.

```
int recurseMeYetAgain(n)
{
     x = 0;
     if (n > 0) {
          for(j=0;j<n;j++) {
               x = x+1;
          }
          n = n - 1;
          recurseMeYetAgain(n);
     }
     return x;
}
```

The recurrence relation for this code is shown below:

```
T(n) = n + T(n-1)
```

Expanding the relation:

```
T(n) = n + T(n-1)
T(n) = n + (n-1 + T(n-2) )
T(n) = n + (n-1 +(n-2 + T(n-3) ) )
T(n) = n + (n-1 +(n-2 + (n-3 + T(n-4) ) ) )
T(n) = n + (n-1 +(n-2 + (n-3 + (n-4 + T(n-5) ) ) ) )
```

Remembering that eventually n will become 0, we will simplify the equation.

$$T(n) = n + (n-1) + (n-2) + (n-3) + ... + 2 + 1$$
$$T(n) = \sum_{n}^{1} i$$

This equation still needs to be converted to a closed form solution, but since we have covered the solution in the previous sections, we write the following:

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

$$T(n) = \frac{n*(n+1)}{2} \quad \text{which is } O(N^2)$$

5.6 Exercises

- Work the following math problems:
    a. $x = \log_2 128$
    b. $x = 2^4 * 2^3$
    c. $x = (2^4)^2 / 2$

- What is the big O notation for each of the following code snippets:

    a.

```
for(j=0;j<N;j++) {
     for(k=0;k<N;k++) {
          x = x + 1;
     }
}
```

    b.

```
for(j=0;j<N;j++) {
     for(k=1;k<=N;k=k*2) {
          x = x + 1;
     }
}
```

    c.

```
for(j=Nj<=0; j = j/2) {
     for(k=0;k<N;k++) {
          x = x + 1;
     }
}
```

    d.

```
for(j=0;j<N;j++) {
     for(k=0;k<N;k++) {
          for(z=0;z<N;z++) {
               x = x + 1;
          }
     }
}
```

- Given that N = 16, for each of the following code snippets, what will x be equal at the completion of the running of each snippet. Assume that before each piece of code is run, x is initialized to 0.

a.

```
for(j=0;j<N;j++) {
     for(k=0;k<N;k++) {
          x = x + 1;
     }
}
```

b.

```
for(j=0;j<N;j++) {
     for(k=1;k<=N;k=k*2) {
          x = x + 1;
     }
}
```

c.

```
for(j=Nj<=0; j = j/2) {
     for(k=0;k<N;k++) {
          x = x + 1;
     }
}
```

d.

```
for(j=0;j<N;j++) {
     for(k=0;k<N;k++) {
          for(z=0;z<N;z++) {
               x = x + 1;
          }
     }
}
```

Chapter 6: Linked Lists

So far in our discussion of data structures we have concentrated on stacks and queues. In each, data was inserted or deleted from the ends of the data structure, thus an array was an ideal simple method of implementation. Our exercises did not require large stacks or queues, so even if we oversized the arrays to accommodate the application, in the grand scheme of things, the amount of memory wasted was not an undermining factor. Since there was no insertion of data in the middle of the stack or queue, there was no need make room in the middle by shifting data around, or by forming a new array with the new element inserted in the middle.

But nevertheless, these problems exist. There are times when the maximum amount of data needed far exceeds the average amount of data needed. If, for example, our application usually works with around 100 records, but on occasion must service up to 5000, if we implement with arrays, our program will be wasting 4900 records worth of memory most of the time. While this may be okay for a test program, or a proof of concept, this would be ill advised for a piece of commercial software. If each record were only 4 bytes in size, again it may be acceptable for short periods of time, but if each record required 40k worth of data (40k is the size of 200 by 200 image) our application would be wasting almost 196 megabytes of memory most of time; EEEK!

Another problem with an array is that in order to insert data into the middle of it, elements must be moved to accommodate the piece of data. If the array does not have many elements, this may not be a concern, but if there are many elements, or the size of each element is substantial, then program efficiency suffers.

As it turns out, a linked list is the ideal data structure to solve these problems. The linked list uses only the number data records as the data, and it is easy to insert and remove data items anywhere in the linked list without have to recreate the list or move existing list members in order to accommodate a new data item or fill in the void left by a deleted item.

How does it work? The linked list is a structure that is kind of like a line of elephants (see figure 6.1) led by the matriarch elephant. The next elephant holds the matriarch's tail, followed by an elephant holding his tail and so forth. Finally, the last elephant in the line has no one holding its tail. If an elephant wants to leave the line, he simply lets go of the tail he is holding, lets the elephant behind him grab the tail he just let go of, and now he can wander off, and most importantly, the elephant line stays intact. If an elephant wants to join the line, he can go grab the last elephant's tail, or he can cut into the middle, by nudging his way in, grabbing a tail, and letting the elephant that he cut in front of grab his tail. If a blind man were as asked to find the last elephant in the line, he would follow the tail/trunk links until he found an elephant whose tail was not being held.

Figure 6.1 shows a line of elephants [2]. Each elephant is "linked" to the one in front of it by using its trunk to hold the preceding elephant's tail. A blind man would find the last elephant in the line by following tail/trunk links until he found an elephant's tail that was not being held.

Somewhat like the elephant line, a linked list is a data structure that is composed of a head (often times called “front”), a sequence of elements, called nodes, connected to one another on after the other, and a tail (see figure 6.2). Each node has a data section and a link to the next element. The data section holds the information of interest, and the link (often called "next") connects it to the next node in the sequence. The head of the list is the first node in the list and is referenced by a variable of "node" type, usually called head or front. The link in the last node of the linked list, often referred to as the tail, points to the null value.

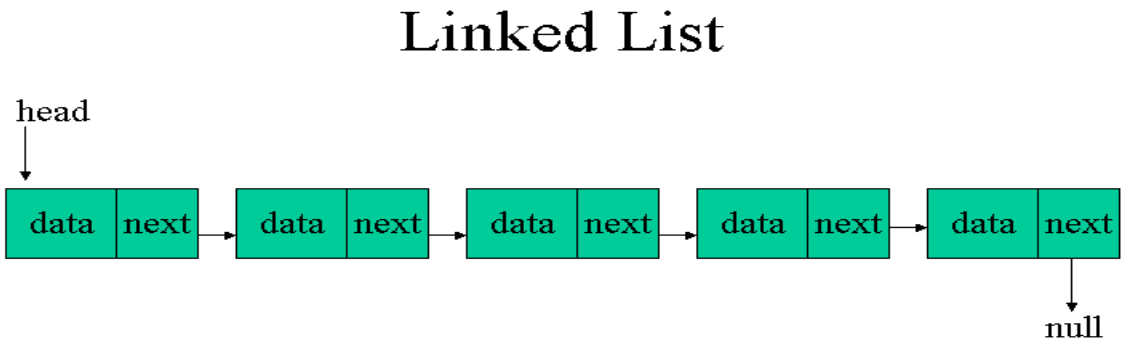


Figure 6.2 shows a linked list with five nodes. The "head" variable points to the first node in the first node in the list. Each node has a data section and a link to the next node in the list. The links can be followed from one node to another until the last node in the list (sometimes referred to as the tail node) is reached. The tail node's link points to null.

The fundamental data type of the linked list is the node. As stated earlier the node has a data section and link section that points to the next node in the list. Below is some pseudo code for a node.

```
class node {
     int data;
     node next;
}
```

This code is nearly identical to what would be used in JAVA. In JAVA, the data items would need methods to access them, or they could be declared as "public". Below is a sample of JAVA code for a node object.

```
class node
{
     int data;
```

```
     node next; /* This is a reference to an object.
                   It will be get a value when
                             an object is created with "new"
                             and next is set to that value. */
}
```

In C, that is non object-oriented C, we would use a structure to create the node. The link to the next node will be an address (pointer) variable of the type “node”. The C code is presented below:

```
struct node {
     int data;
     node *next;
};
```

In C++ instead of using a "struct", a class is used. A class in C++ is much like a class in JAVA. A struct (structure) in C is what in older literature is referred to as a record. That is, it is a collection of data that is often associated together. A class implies that there is the associated data along with methods that are commonly used to work with the data. Below, a very basic C++ class for the node is presented:

```
class node {
     public : int data;
     public :node *next;
};
```

Note that once again, the next variable is declared as a pointer of type node. Also, the keywords public are used so that access to these variable can be made outside the class. A more proper version of the C++ class, as well as the JAVA class, would have class methods to control access to the classes’ variables.

In order to effectively use linked list, some fundamental methods need to be developed. In the next section, these methods are discussed, and pseudo code is provided.

6.1 Fundamental Methods for Lists

The pseudo code below is for a singly linked list, and it is C/JAVA like. At the risk of being repetitive, the pseudo code for the node class described above, and is very basic for illustration purposes. Since it is pseudo code for illustrative purposes, variable scope keywords like "public" and "private" are not used, nor are object guidelines adhered to.

The code uses the class "node" as described earlier.

```
class node {
     int data;
     node next;
}
```

The front of the linked list is of type "node" and is set up as a global variable for this discussion. If this were coded as a JAVA class, the font of the list would be a class variable and would be global to all class methods.

```
node front;  /* Global variable. */
```

6.1.1 Initialization Method - init()

The init method sets the front of the list to null, essentially creating an empty list. Figure 6.3 below shows a diagram of the empty list created by a call to the init() function.

```
init()
{
     front = null;
}
```

Figure 6.3 shows an empty list. Upon initialization, the pointer to the head of the list is set to a null pointer. The null pointer is a defined variable in JAVA; in C/C++ it defined using a statement like:
#define null (char)0

6.1.2 Node Creation - makeNode()

The makeNode() method allocates a new node object, initializes its data field, and sets its link to the next node to null; The diagram in figure 6.4 illustrates the results of a call to makeNode().

```
node makeNode(int data)
{
     node newNode;

     newNode = allocate memory for a new node;
     newNode.data = data;
     newNode.next = null;

     return newNode;
}
```

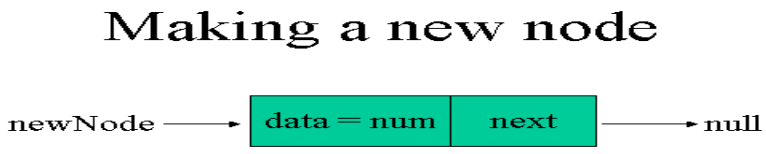


Figure 6.4 shows the results of a call to makeNode.

6.1.3 Finding the Last Node of a Linked List - findTail()

The findTail method returns the address of the last node in the list. It does this by starting at the front of the list and following the links until it reaches a node whose next pointer is equal to null. Figure 6.5 below shows a linked list with 5 nodes. Table 6.6 shows a trace of the findTail method.

```
node findTail()
{
     node current;

     current = front;
     while(current.next != null) {
          current = current.next;
     }
     return current;
}
```

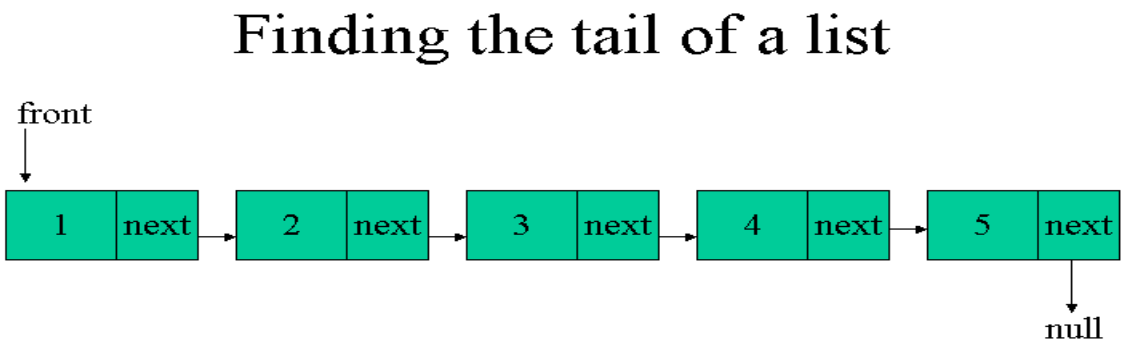


Figure 6.5 shows a linked list with 5 nodes. Note that the last node's "next" parameter points to null. The findTail() method looks for the pointer to null in order to find the last node in the linked list.

current points to:	***current.next point to:***	***data of node that current points to***	***Remarks***
head	node 2	1	start of method
node 2	node 3	2	
node 3	node 4	3	
node 4	node 5	4	
node 5	null	5	node 5 is returned as the last node in the list

Table 6.6 shows a trace of the findTail method. The "current" variable is set to the head of the list. At each iteration of the list current follows the next pointer, until the next pointer is equal to null. When current.next is equal to null, the method ends, passing the address of the last node in the list back to the calling program.

6.1.4 Adding Nodes to the End of the List - addAtEndOfList()

The addAtEndOfList() method finds the end of the list using findTail() and appends a new node to the end, unless the list is empty. If the list is empty, it creates a new node and sets the front of the list to point at the new list. Figure 6.7a through 6.7d illustrate the process.

```
addAtEndOfList(node spot, int data)
{
      node tail;

      if (front == null) {
            front = makeNode(data);
      }
      else {
            tail = findTail();
            tail.next = makeNode(data);
      }
}
```

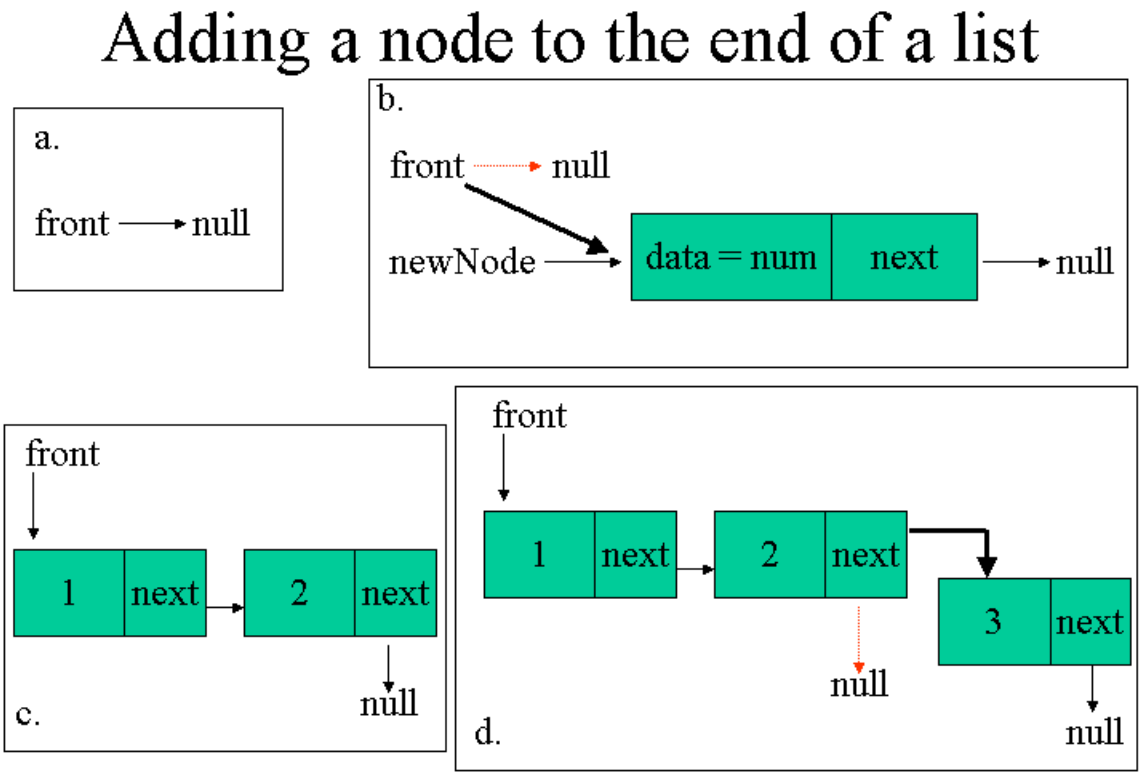


Figure 6.7a and 6.7b show what happens when addAtEndOfList() is called and list is empty. In 6.7a the list is empty; in 6.7b the new node has been created and the pointer to the front is reassigned to the new node. The dotted red line from front to null indicates that it has been overwritten by the bold arrow pointing to the new node. Figure 6.7c shows a list with two nodes. 6.7d shows how the next pointer of the 2nd node has been overwritten by the pointer to the new node.

Below is a listing of Java code run in Repl that illustrates most of previously discussed concepts. It does differ in that instead of calling the addAtEndOfList() function, it encapsulates those ideas in the buildSimpleList() function.

```
class node {
  int data;
  node next;
}

class list {
  node front;

  public void init()
  {
    front = null;
  }

  public node makeNode(int num)
  {
    node newNode;

    newNode = new node();
    newNode.data = num;
    newNode.next = null;
    return newNode;
  }

  public node findTail()
  {
    node curr;

    curr = front;

    while(curr.next != null) {
      curr = curr.next;
    }
    return curr;
  }

  public void buildSimpleList(int len)
  {
    int j;
    node tail;

    init();

    for(j=0;j<len;j++) {
      if (j == 0) {
        front = makeNode(j);
      }
      else {
        tail = findTail();
        tail.next = makeNode(j);
      }
    }
  }

  public void showList()
  {
    node curr;
    System.out.println("list ...");
    curr = front;
    while(curr != null) {
      System.out.println(curr.data);
      curr = curr.next;
    }
  }
}
```

```
class Main {
  public static void main(String[] args) {
    list l;

    System.out.println("Hello TV Land!");

    l = new list();
    l.init();
    l.buildSimpleList(10);
    l.showList();
  }
}
```

6.1.5 Adding Nodes in the Interior of a List - addAfter()

The addAfter method inserts a new node into the list after the node pointed to by the variable "spot". To do so the links must be arranged so that spot points to the new node, and the new node points to the node pointed to by spot. This can be done using a temporary variable whose purpose is to save the location that spot points to, or it can be done as in the code below, where for a brief moment, both spot and the new node point to the node that was originally after spot. Figure 6.8 illustrates the process.

```
addAfter(node spot, int data)
{
     node newNode;

     newNode = makeNode(data);

     newNode.next = spot.next;
     spot.next = newNode;
}
```

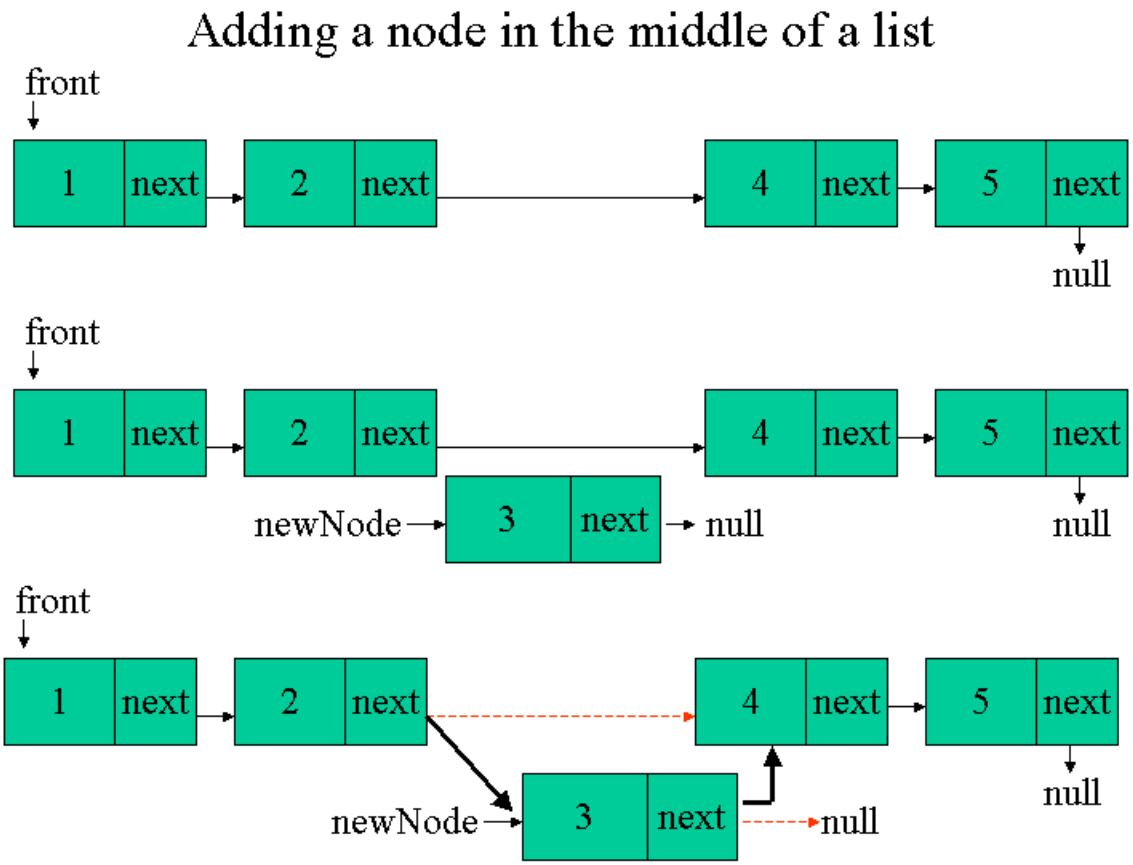


Figure 6.8 shows the process of adding a node into the middle of a linked list. The list at the top of the figure has 4 nodes. If the variable "spot" points to the 2nd node in the list and the "data" variable is set to 3, then after the call to makeNode() the result would be illustrated in the list in the center of the figure. Note that at this point the newNode has its data but is not attached to anything. When newNode.next is set to spot.next, the next parameter of newNode will point to the node whose data is equal to 4. When spot.next is assigned to newNode, the newNode has been sewn into the list.

6.1.6 Deleting Nodes in a List - deleteAfter()

The deleteAfter function removes a node from the list. It works by redirecting the spot.next pointer so that it skips the node that immediately follows the pointed to by spot and instead points to the 2nd node down from the node pointed to by spot. One thing to remember is that in JAVA, the system's garbage collection function will collect up the deleted node, but in a C based language it is the responsibility of the programmer to free any memory that is no longer needed. Figure 6.9 shows the deletion process.

```
deleteAfter(node spot)
{
     node nextNode;

     nextNode = spot.next;
     spot.next = nextNode.next;
}
```

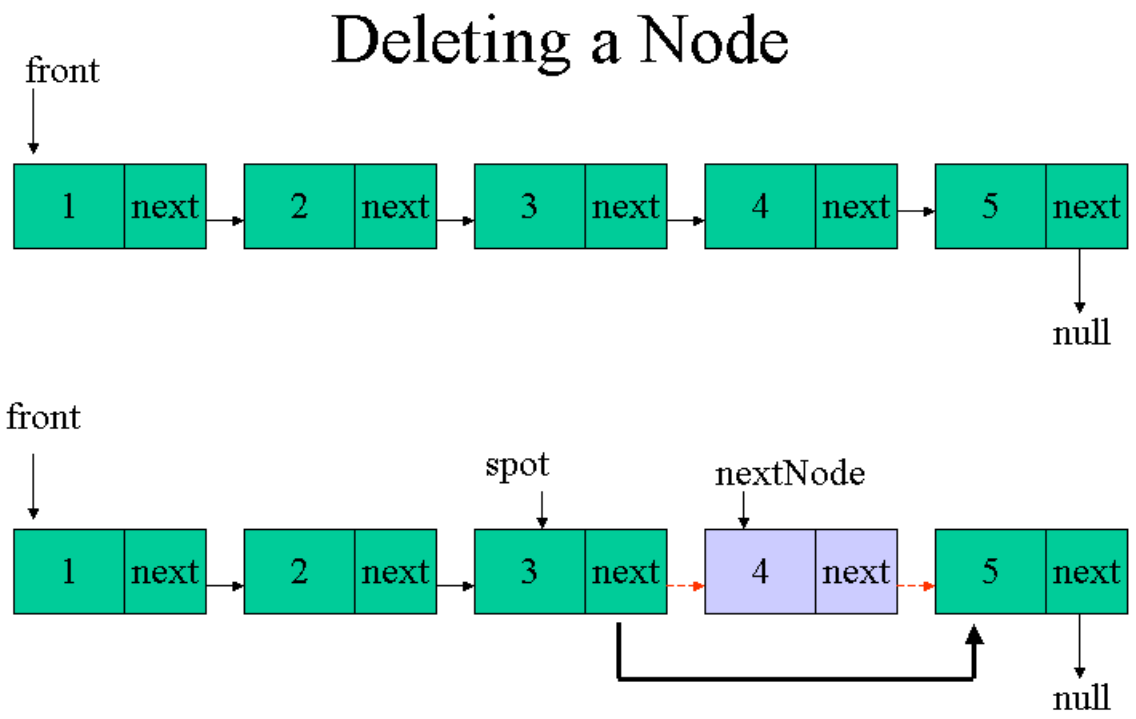


Figure 6.9 shows the node deletion process. Once spot.next has been set to point to the node after nextNode, nextNode is freed, indicated by the blue color. The JAVA system will automatically reclaim the memory, but C/C++ programmers need to manually return nextNode's memory to the system.

6.2 Fundamental List Applications

In the previous sections, the concepts and basic methods of linked list were explored. At this point is best to get some hands on experience with lists. The reason for this is so that the programmer can test each of the basic list methods before moving on to more complex programs. A big part of software development is the process of testing and integrating. By proving that the list utility methods work on a simple program, it will be easier to isolate errors on a more complex program that uses those same list methods.

6.2.1 Fundamental Exercises

1. Write a program that creates a linked list and loads it with the numbers 0 to 9. To do this, use a "for loop" with a call to the addNodeAtEndOfList nested in the loop. Create the functions, init, makeNode, and findTail to support the addNodeAtEndOfList function. In order to make sure everything is working, the contents of the list will have to be displayed. Make a showList function and call it after the list has been loaded by the for loop.
2. Make a small text menu for your program so that you ask the user how many nodes they want in the list.
3. Add the capability to add a node between two nodes. Make sure and ask the user which node they want to place the new node after, and the value of the node to be

created. You will need to make a function that finds the node the user requested. This node will be sent to addAfter function.

4. Add the capability to delete a node. Fashion your solution in a manner similar to what was done in exercise 3.

6.3 Building an Ordered List

Building an ordered list is one of the central topics of working with linked lists. To do so the software needs to find where in a list a data item is to be inserted and then carry out the insertion. **The catch is that in a singly linked list insertions can only occur *after* a node, so the position immediately before where the data is to be inserted must be found.**

This is a fundamental part of working with linked lists. To implement this search, two node pointers are used, one that points to the current node, and one that points to the previous node. The search begins at the front of the list, with the current and previous positions initialized at the front of the list. At each iteration of the search, the data at the current position is analyzed. In the case of a list of ordered numbers, if the data at the current node is greater than the number to be inserted, the search ends. If not, the previous position is set to the current position and the current position is moved to the next node. An example helps. Looking at the set of numbers below, let's say that the front of the list is the number 1, 39 is the last number in the list, and we want to insert a 17 into the list.

1, 6, 7, 11, 12, 15, 19, 32, 39

If we traverse the list, starting at the number 0, that is, our current and previous variables are set to point at the 0 initially, each iteration will move, current and previous by one list node, and previous will always be pointing to the number behind current. When current reaches the 19, the algorithm will detect that 19 is greater than 17 and it will end. The algorithm will return the previous variable, it points at the 15, and the addAfter (often times called insertAfter) function will insert the 17 between the 15 and the 19.

Iteration	***Current*** variable points to:	***Previous*** variable points to:
0	1	1
1	6	1
2	7	6
3	11	7
4	12	11
5	15	12
6	19	15

The pseudo code that implements these ideas is shown below:

```
node curr, prev;

curr = front;
prev = curr;
while(not at the end of the list) and
      (curr.data < number) {
      prev = curr;
      curr = curr.next;
}
return prev;
```

Once the correct position is located, the addAfter(), or insertAfter() function can be called to insert the number into the position pointed to by the "prev" variable.

This method of locating a position in an ordered list is a fundamental part of working with linked lists and the relationship between the current and previous pointers and the insert after function is key to working with singly linked lists.

6.3.1 Ordered List Insert Function

The previous sections have shed light on various list operation such as inserting node, finding where to insert nodes, deleting nodes, etc. In some instances, it is easier and cleaner in the code to combine the finding of where to insert a node and the insertion functionality into a single insert function. The code below is designed to illustrate this idea. The code can be made more efficient; it is presented in this manner so that all of the cases that need to be considered can be seen clearly.

```
void insert(int num)
{
      node t, curr, prev;
      boolean searching;

      if (front == null){
            front = makeNode(num);
      }
      else if (num < front.data) {
            t = makeNode(num);
            t.next = front;
            front = t;
      }
      else {
            searching = true;
            curr = front;
            prev = curr;

            while(searching == true) {
                  if (curr.data == num) {
                        //System.out.println("found a duplicate!");
                        curr.count++;
                        searching = false;
                  }
                  else if (curr.data < num) {
                        if (curr.next == null) {
                              curr.next = makeNode(num);
```

```
                        searching = false;
                    }
                    else {
                        prev = curr;
                        curr = curr.next;
                    }
                }
                else if (curr.data > num) {
                    t = makeNode(num);
                    t.next = curr;
                    prev.next = t;
                    searching = false;
                }
            }

        }
}
```

6.4 More Complex Lists, with Applications

So far the lists that have been dealt with have all been singly linked lists. In this section we will discuss multi-indexed lists, circular linked lists, and doubly linked lists.

6.4.1 Building a Multi-Indexed List

Multi-indexed lists are very much like singly linked lists, with the exception that they provide the programmer/user with the ability to enter the list from some other place than the front of the list. By being able to jump into the list at multiple points, the efficiency of finding data can be increased.

Consider an unordered list that has 10,000 nodes; each time data is requested from the list, on average, 5,000 nodes will need to be searched (if each data item is equally likely to be requested). Since the list is not ordered, if the data is not present in the list, this fact would not be known until the entire list were searched. That is a lot of links to follow, just to find out that what you were looking for was not present. If the list were ordered, the search could determine that an element was not present by finding the position that it should be, but for items in the list, the average number of elements to search would still be around 5000.

A quick remedy would be to order the list and then create a index with multiple entries into the list. For example, if the data was phone book information (name, address, phone number), the list would be built in alphabetical order. Then an index array with 26 elements could be created with the first element pointing into the list to first name in the 'A' section, the second element pointing to the first name in the 'B' section and so on. Now when a search for a data item were executed, the search no longer starts from the front and follow the links until either the item were found, it could go directly to the section and start searching there. Instead of searching a large portion of the list (somewhere around half of list on average), the search portion of the list to be searched would be reduced to about 1/26 of the un-indexed search. With an indexed list size of 10,000 the average number of nodes to be searched would be around 193. That is quite an improvement over the 5000, required for the unordered list, and ordered non-indexed list.

To illustrate the development process of the indexed linked list it is useful to consider a similar but slightly less complex problem. Suppose that the instead of the phone book data as used in the above discussion, the data consisted of a person's name and their weight. We want to arrange the list based on weight, so when the list is built by placing the nodes in their proper place, it is based on the weight parameter. So, the lightest people are in the front of the list, and the heaviest are toward the rear of the list.

To build the list in order, each time a new node is retrieved, a function must examine each node of the list until a node with a weight parameter greater than the new node's is found. The new node is placed before the node whose weight parameter is greater. The pseudo code below illustrates the process:

```
class node {
     string name;
     int weight;
}
.
.
.
init()

while(data is still available) {
     newRecord = getOneDataRecord();

     if (list is empty) {
          front = makeNode(newRecord);
     }
     else {
          spot = findSpot(newRecord.weight);
          if (spot is the front of the list) {
               add the new record at the front of the list.
               make front point to the new node.
          }
          else {
               addAfter(spot, newRecord);
          }
     }
}

makeIndex(front, index);
```

The key to this pseudo code is the findSpot function. Below is some pseudo code for findSpot();

```
.
.
```

```
findSpot(int w)
{
     node curr, prev;

     curr = front;
     prev = curr;
     while(not the end of the list) and
          (curr.weight < w) {
          prev = curr;
          curr = curr.next;
     }
     return prev;
}
```

The findspot() function finds the first node whose weight is higher than that of the "w" parameter, and returns the previous node. Why return the previous node? So that the addAfter() function can insert the new node in the correct position. That is, the new node will be inserted right after the last node whose data was greater than that of the new node. There are some special cases to be aware of; what happens if the new node's data is less than the data in the very first node?; what happens if the new nodes data is greater than all of the existing nodes in the list. In the first case, the new node will have to become the first node of the list, and the front variable will have to be assigned to point to the new node. In the last case, the new node will be added to the back of the list. These questions are effectively addressed by the insert function discussed in section 6.3.1

To make indices into the list, we can develop a process called makeIndexList(). After the list is built, the makeIndexList() loads an index table for later use.

```
.
.
.
class indexNode {
     int weight;
     node firstOne;
}

makeIndexList()
{
     node spot;
     int currWeight;

     indexNode weightIndex[maxWeight/10];


     /* Fill in the index array. */
     for (j = 0 to maxWeight/10) {
          currWeight = j*10;
          weightIndex[j].weight = currWeight;
          spot = findSpot(currWeight);

          if (spot is equal to the front of the list) and
```

```
                (spot.weight > (currWeight + 10)) {
                  weightIndex[j].firstOne = null;
            }
            else {
                  weightIndex[j].firstOne = spot;
            }
      }
}
```

The makeIndexList() method sets up an index that has pointers into the list at 10 pound intervals. It uses the findSpot() function to search for the positions in the list; if the position pointed to by spot is the front of the list and the node at the front of the list is greater than currWeight by more than 10 pounds, then that index is not valid, so it is set to null. Figure 6.10 below shows an illustration of how the indices point into the linked list at 50 pound intervals.

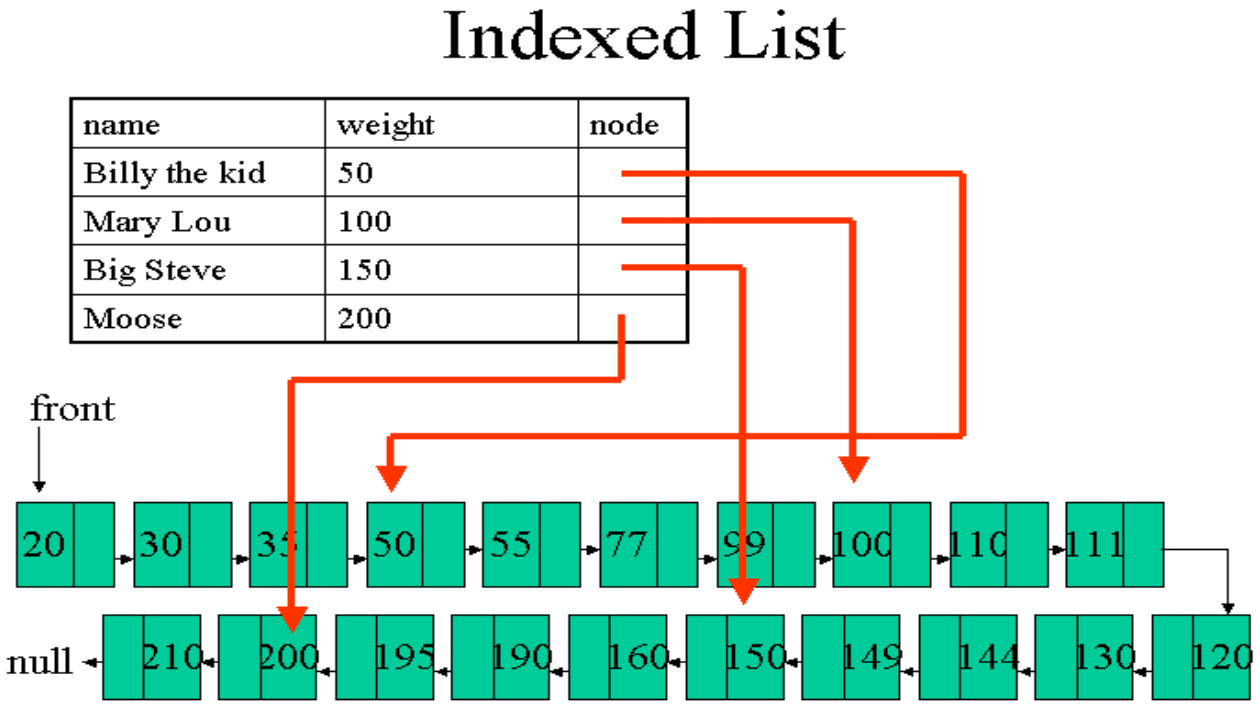


Figure 6.10. This figure illustrates an Indexed Linked List. After the list has been built an index is created so that search efficiency can be improved. For example, if we are interested in the people whos weight is between 100 and 150 pounds we can use the index to jump straight to the node occupied by Mary Lou. Normally we would have traverse the nodes containing 20, 30, 35, 50, 55, 77, and 99 to get 100. Using the index we examine the first entry, Bill the kid, and then jump straight to Mary Lou.

There are many improvements that can be made to the above pseudo code. The main one is in area of efficiency. The list traversed around maxWieght/10 times because of the findSpot() routine. A better design would only traverse the list once.

6.4.2 Circular Linked Lists

A circular linked list is a list whose tail node points to the head/front instead of null. When traversing the list, the links are followed as in a normal list from the head to the tail, but when tail is reached, the next link leads to front of the list. If desired the whole list can't be easily continually traversed.

One example of when a circular linked list would be useful is in the implementation of game in which several agents take turns. The beauty of using the circular linked list is that new "players" can come and go without having to move chunks of player data (as it would be if it were stored in an array). That is, players data could be simply inserted into the correct place in the list when they arrive and deleted when they leave.

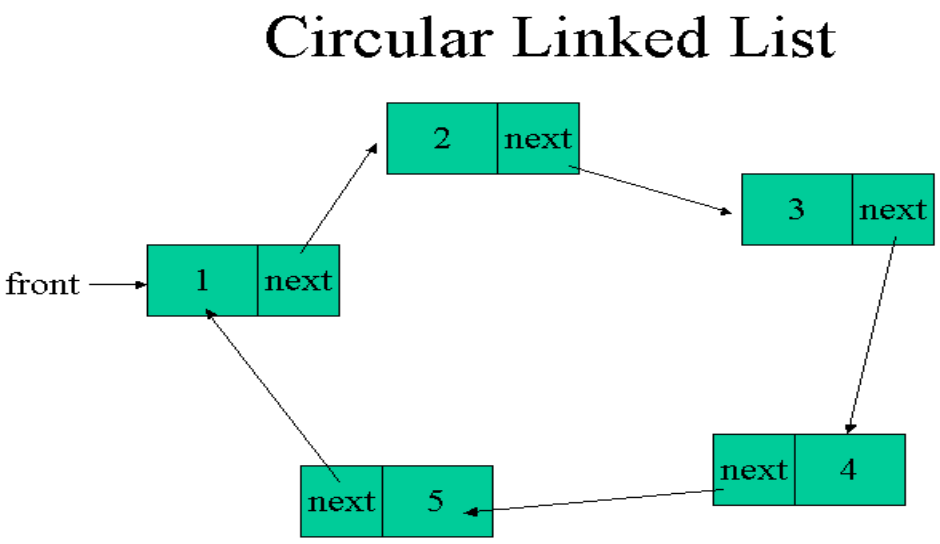


Figure 6.11 The figure above show a linked list with 5 nodes. Notice that the last node's next pointer points to node 1 instead of null.

Converting a singly linked list to a circular linked list is accomplished by finding the tail of the list and pointing it's "next" pointer to the front of the list. Converting a circular linked list to singly linked list is a bit trickier; the list must be traversed until the current node's next pointer is found to point to the front of the list. Once this node is found, it's next pointer can then be set to null, thus breaking the circular link.

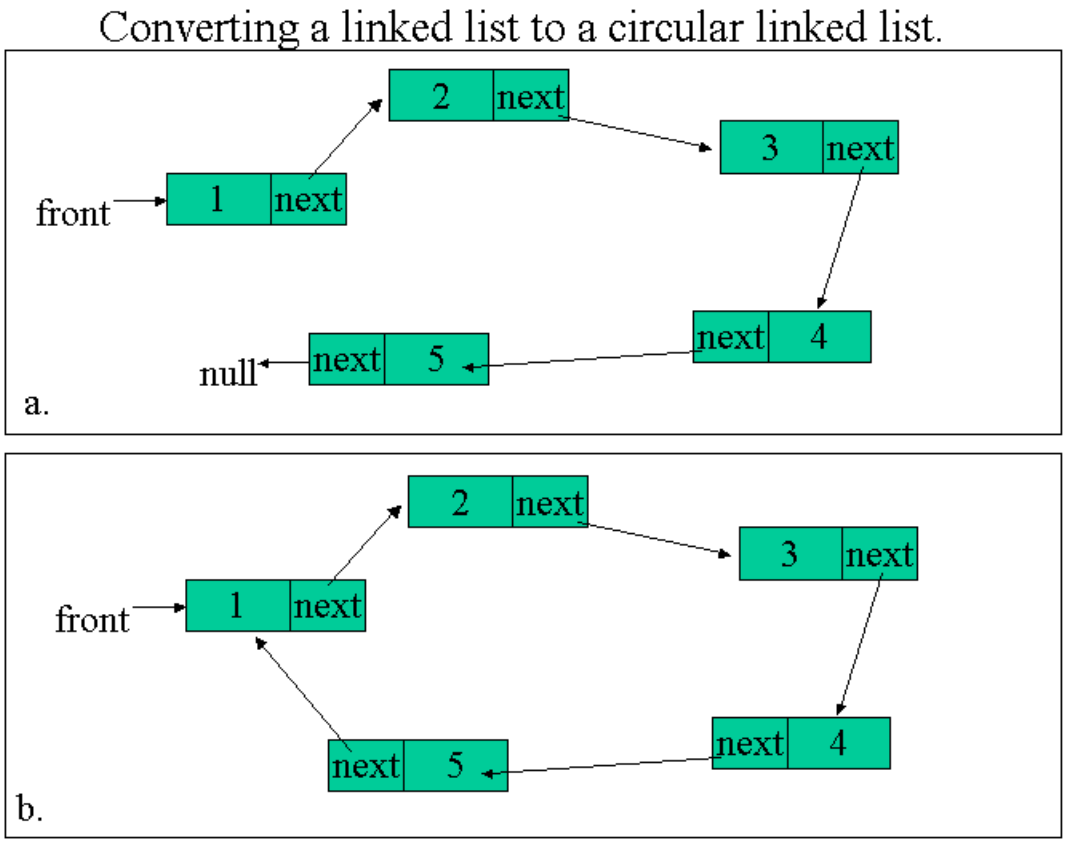


Figure 6.12 Parts a and b of the above illustration show a regular linked list and circular linked list. To implement the conversion, the findTail() routine can be used to locate the last node in the list. Once the tail node is found, its next pointer is set to the front of the list as in 6.12b.

6.4.3 Doubly Linked Lists

A doubly linked list is a list that can traversed in both the forward and reverse directions. To accomplish this, the list's node structure is modified by adding a previous node pointer. An example of a doubly linked list node used for holding text data is shown below. Remember, if using JAVA, to access the data items in the class, the appropriate methods will need to be written, or the data items will have to be public.

```
class doubleNode {
      string name;
      doubleNode next;
      doubleNode prev;
}
```

As in the regular linked list node, there is a next pointer, but in the doubleNode there is also a prev pointer. Notice that both of these data items are of the same type, i.e. doubleNode. As explained in the first sections of this chapter, this is because they point to data items of that type. Figure 6.13 below shows the structure of a doubly linked list.

Doubly Linked Lists

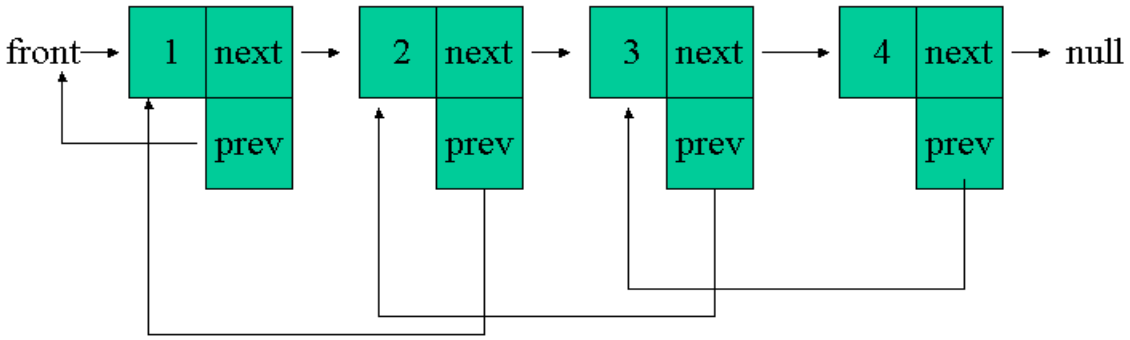


Figure 6.13 This figure illustrates the structure of the doubly linked list. Note that the only real difference between it and a normal singly linked list is "prev" pointer. The prev pointer enables the list to be traversed in reverse.

A classic data structures problem is "How do you turn a singly linked list into a doubly linked list?". One answer to this question is that the list is traversed in the forward direction while keeping track of the previous node. Each time a new node is entered, it's prev entry is loaded with the previous node data. The pseudo-code below illustrates the point.

```
curr = front;
prev = curr;
while(not the end of the list) {
     curr.prev = prev;
     prev = curr;
     curr = curr.next;
}
```

Another more syntactically complex way is look ahead from the current node. In this strategy the next node's prev entry is set to current. The pseudo-code below illustrates this method.

```
curr = front;
while(not the end of the list) {
     if (curr.next != null) {
          curr.next.prev = curr;
     }
     curr = curr.next;
}
```

6.5 Complexity Issues

Since linked lists are traversed (searched) sequentially the search is of order N, in big O notation we say it is O(n). The reasoning is straight forward and is as follows. Since the search begins at the front of the list, and proceeds one node at a time, the search must always examine some fraction of the N nodes in the list. The luckiest search will find the desired node in one comparison, the least lucky search will traverse the entire list and make N comparisons. If the desired item is equally likely to be in any position of the list, then the average number of comparisons is about N/2, which is O(n).

It can be no other way because unlike a binary search, a location to make the next comparison cannot be calculated, because of the nature of the list. The location of the nodes is determined by the system as the nodes are allocated, so they cannot be programmatically calculated.

The efficiency gains realized by the indices described in section 6.4 are real and substantial, but the fact remains that once the index has been followed, the search is sequential. The index effectively reduces the size of N, but it does not alter the search method, so it remains O(n).

6.6 Exercises

For this exercise create a text file full of names called names.txt. Use the following list of names for data.

```
joe
bob
harry
mary
brian
tom
jerry
bullwinkle
pam
ellis
dale
bill
barrack
george
gertrude
zack
zeus
apollo
gemini
greg
larry
meriam
webster
thomas
stewart
dianna
theresa
billyjoe
carl
karl
```

```
charles
karla
donna
tena
kerry
howard
johnson
ulyssess
paul
peter
issaac
marvin
dudz
chuck
ellie
anny
judy
matt
ross
dan
robert
kim
eric
junkun
ghassan
cris
raymond
avery
roy
halley
mitzee
ziggy
rocky
twirly
max
huey
dewy
hongkongfooey
clarence
lala
sammy
fred
francis
```

1. Write a program that reads a data file of names and inserts them into a list in alphabetical order, based on name. Your program should have the following functionality:

    a. Ask user for the file name to read from (see the supplemental JAVA code for reading name from a file at the end of this exercise).
    b. Read in names from the file, names.txt
    c. Names should be inserted in the appropriate place (alphabetically).
    d. ***Do not presort the list***
    e. You are to create routines (methods) such as:
        i. init()
        ii. makeNode()
        iii. insert()
        iv. buildIndex()

v. etc.

f. Have menu options (just text using a console application) to do the following:
   i. Display the list.
   ii. Request the length of the list
   iii. The user should be able to delete a single name from the list.
   iv. Request the length of a section of the list. (How many people have a name that starts with "B")
   v. Also, your program should be able to print out sections of names, for example it should be able to print all peoples names that start with an "A", "B", etc.
      1. To do so, you need to make an index that has links into your list for each letter of the alphabet. If there are no entries for a particular letter, your link should point to NULL and if those names are requested, your program should inform the user.

Issues that need to be addressed:

- Converting a name to something that can analyzed like a number for the purpose of alphabetical organization of the list is required. For the purposes of this program, use the first three letters of each name. Each letter will be given a value, between 0 and 25 (we will use only lower case letters to make this easier). A small 'a' will have a letter value of 0, a 'b' will give a 1, c – 2, d -3, ….. , z – 25. In order to get these values we use equation 1 below

Equation 1:

letterVal = letter - 'a'

The idea of equation 1 can be incorporated into a function of the following form:

int getLetVal(char ch)

Then a value for the name can be generated using an equation similar to equation 2 below:

Equation 2:

$$\text{nameCode} = \text{getLetVal(name[0])}*26^2 + \text{getLetVal(name[1])}*26^1 + \text{getLetVal(name[0])} * 26^0;$$

What is going here is the name is being treated as a base 26 number. The first letter is multiplied by 26^2 just as in the number 727, the first digit of the number is multiplied by 10^2.

- Software development issues are important in this assignment. That is, testing of each part before it is integrated into the system will become important. Make sure names like “ava” generate numbers that are less than eddy and before integration.
- Before any names are put into the list, the list building software could be tested with a list of unordered numbers. If it can read the numbers, build a list, and print out the list, and the printed-out list is in low to high order, the next step would be to integrate the name conversion functionality.
- By testing pieces by themselves, and integrating them into the system one at a time, errors can be found more easily.

Here is some example code for reading some names from a file in JAVA

```
/**
 * fileIn.java
 *
 *
 * pm
 * @version 1.00 2009/9/16
 */

import java.util.Scanner;

import java.io.*;

public class fileIn {
      String fname;

    public fileIn() {
      System.out.println("Constructor");
      getFileName();
      readFileContents();
    }

    public void readFileContents()
    {
      boolean looping;
      DataInputStream in;
      String line;
      int j, len;
      char ch;

      /* Read input from file and process. */
        try {
            in = new DataInputStream(new FileInputStream(fname));

            looping = true;
            while(looping) {
                /* Get a line of input from the file. */
                if (null == (line = in.readLine())) {
                    looping = false;
                    /* Close and free up system resource. */
                    in.close();
                }
                else {
                  System.out.println("line = "+line);
                  j = 0;
                  len = line.length();
```

```
                    for(j=0;j<len;j++){
                          System.out.println("line["+j+"] =
"+line.charAt(j));
                    }
                  }
            } /* End while. */

        } /* End try. */

        catch(IOException e) {
            System.out.println("Error " + e);
        } /* End catch. */
    }

    public void getFileName()
    {
      Scanner in = new Scanner(System.in);

      System.out.println("Enter file name please.");
      fname = in.nextLine();
      System.out.println("You entered "+fname);
    }

    public static void main(String[] args)
      {

            System.out.println("Hello TV land!");

            fileIn f = new fileIn();

            System.out.println("Bye-bye!");
      }
}
```

Chapter 7. Trees

In this chapter we discuss trees. Trees are a fundamental data structure whose configuration can lead to excellent gains in computational efficiency. In fact, order of complexity of a search of a basic binary tree is of $O(\log_2 n)$ vs $O(n)$ for a linked list. This efficiency comes at price; the code to implement the building and traversing a tree is more complex than that of working with arrays, and similar but arguably more complex than that of linked list code. The implementations discussed in this chapter are mostly for binary trees, and the basic tree node structure will look familiar, it is similar to the node structure of the linked lists presented in the previous chapter.

The tree is a data structure with a root from which sub-trees branch out. The entry point into the tree is the root. The root is attached to any number of child nodes. Each child node can have children also, thus each child node is the root of its sub-tree. When a child node does not have children, it is called a leaf. One special feature to remember about trees is that child nodes never branch upward to their parent node or other ancestors, thus trees are said to be acyclic. Much more will be said about cycles and acyclic node formations in later sections and chapters. Figure 7.1 below the basic structure of a tree.

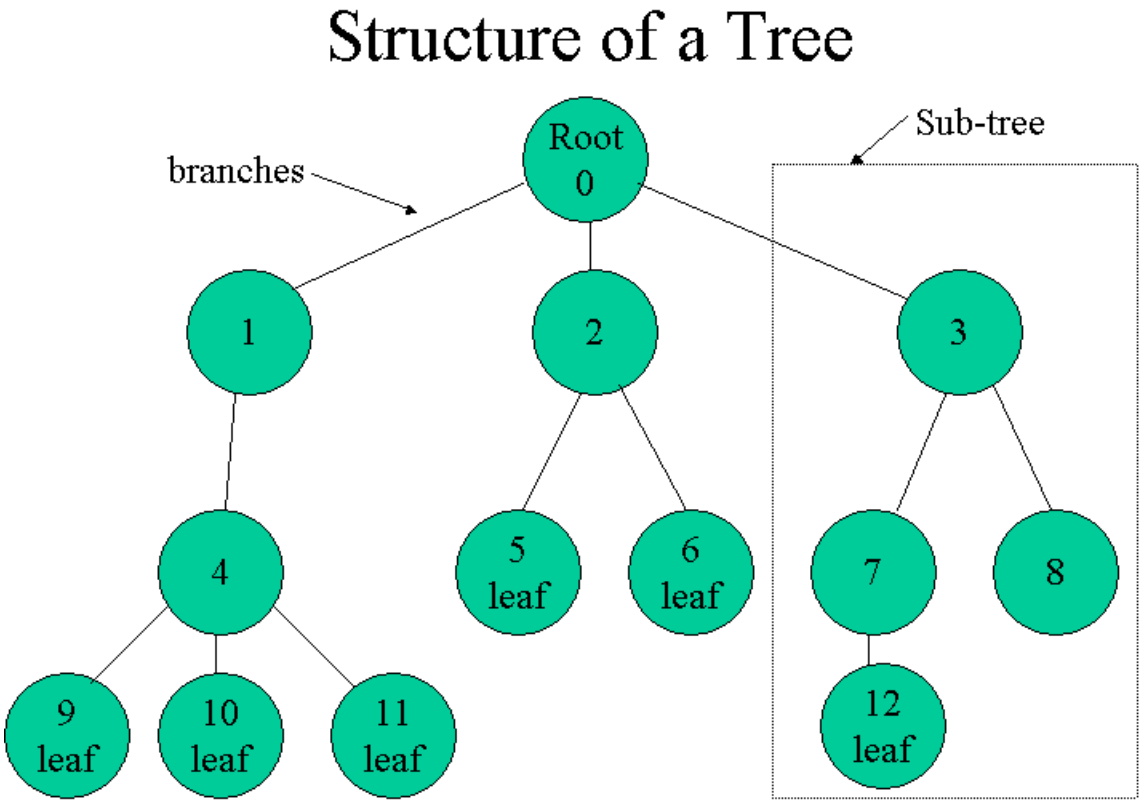


Figure 7.1 This figure illustrates the various structures of a tree. As can be seen, the root of the tree is shown at the top (node 0). While in life/biology roots are at the bottom of a tree, in computer science trees are traditionally drawn with the root of the structure at the top. Nodes 1, 2, and 3 are children of node 0, the root, and form the root of their respective sub-trees. Notice that nodes 5, 6, 9, 10, 11, and 12 have no children; we call these leaves.

As can be seen in figure 7.1, the root has 3 children. Nodes 1, 2, and 3 are the roots of their respective sub-trees. Notice that the tree's nodes seem to be grouped in levels. We say that the root is at level 0, and nodes 1,2, and 3 are in level 1, nodes 4, 5, 6, 7, and 8 are in level 2 and so forth. This tree has a total of 4 levels: levels 0, 1, 2, and 3.

When a tree's nodes are limited to having only two children it is called a binary tree. Binary trees have many interesting features, thus the majority of the text in this chapter will be devoted to them. When a binary tree's nodes always have 2 children except for its leaves, it is referred to as a "complete binary tree". Figure 7.2 below shows examples of binary trees.

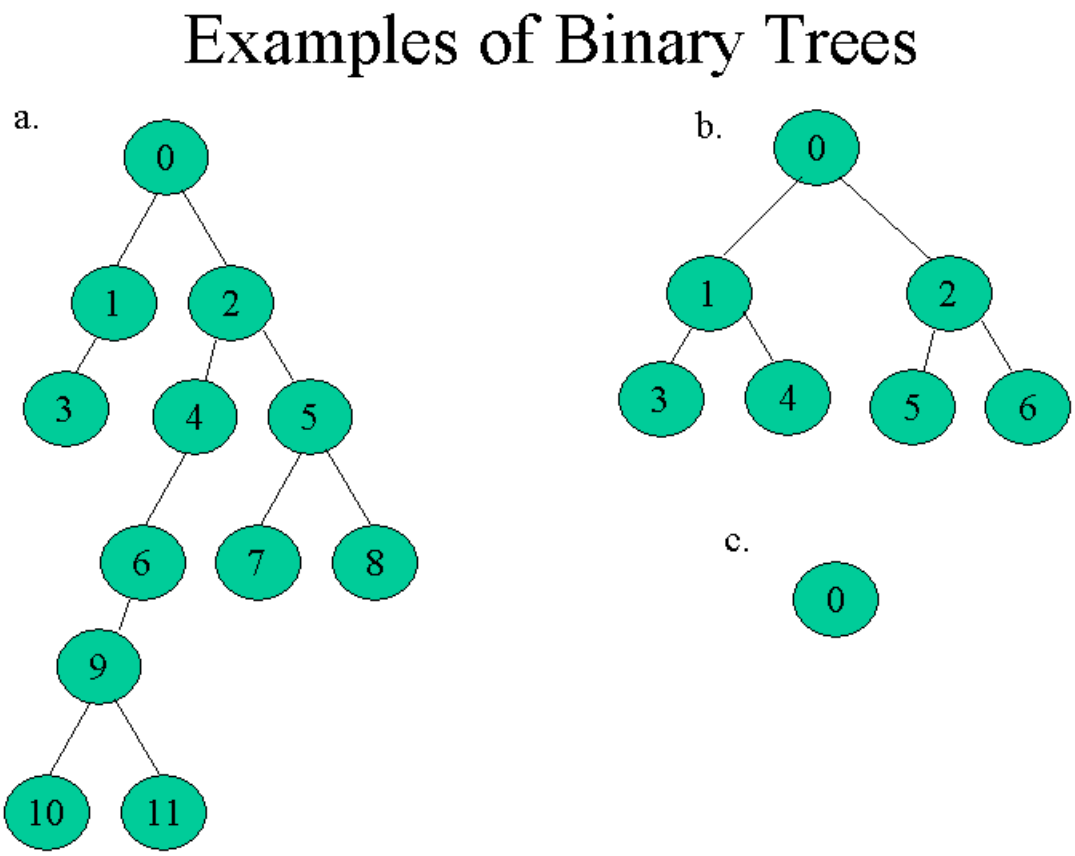


Figure 7.2 This figure shows 3 examples of binary trees. 7.2.a is a binary tree with 6 levels. 7.2.b is a complete binary tree with 3 levels, and 7.2.c is a complete binary tree with 1 level.

Complete binary trees have some interesting properties related to the height of the tree. For example, the number of nodes in the tree can be calculated based on its height.

$$\text{total nodes in tree} = 2^{\text{tree height}} - 1$$

or

$$N = 2^{h} - 1$$

Using figure 7.1.b as an example:

$N = 2^h - 1$
$N = 2^3 - 1$
$N = 8 - 1$
$N = 7$

This is easily verified by counting the nodes. What about the single node shown in 7.2.c?

$N = 2^h - 1$
$N = 2^1 - 1$
$N = 2 - 1$
$N = 1$

Again, it is easy to see that there is only one node in 7.2.c.

A good question is "Why does this equation work?". It works because in a complete binary tree every node has 2 children, except the leaves. This has an interesting effect on the number of nodes in each level. If the root (level 0) has one node (it does, and it is the only level in which the number of nodes is not a multiple of 2), but must have 2 children, then level 1 must have 2 nodes, a left and right child. If the left child of level 1 has two children and right child has 2 children, then level 2 will have 4 children. If each of those 4 children has 2 children, then the next level will have 8 children and so forth. So, the total number of nodes can be described by the following equation:

$N = 1 + 2 + 4 + 8 ....$
$N = 2^0 + 2^1 + 2^2 + 2^3 + ...$

$N = \sum 2^i$,where i = 0 to the number of levels - 1.

It can shown through induction that the following is true:

$2^h - 1 = \sum 2^i$,where i = 0 to the number of levels - 1.

The inductive proof follows:

$2^h - 1 = \sum 2^i \quad i = 0, h-1$

The strategy for this inductive proof is to first show that this statement is true for h=1, then we show it is true for h = h+1 by assuming that it is true for h.

h = 1

$2^h - 1 = \sum 2^i \quad i = 0, h-1$

$2^1 - 1 = 2^{1-1}$

$2 - 1 = 2^0$

$1 = 1$

Now for h = h+1

$$2^{h+1} - 1 = \sum 2^i + 2^h$$

$$= 2^h - 1 + 2^h$$

$$= 2(2^h) - 1$$

$$= (2^1 * 2^h) - 1$$

$$2^{h+1} - 1 = 2^{h+1} - 1$$

So, using induction, it has been shown that summing the number of nodes in each level of complete binary tree (cbt) follows the equation:

$$N = 2^{h+1} - 1$$

Intuitively, each new level has twice as many nodes as the previous level, because each node in the previous level must have 2 children, as specified by the definition of the complete binary tree.

7.1 Node Structure and Basic Tree Methods

The basic methods for creating and using trees are very similar to those for linked lists. The most obvious difference is that the node structure needs to have more than a single link. For this discussion the focus is going to be on binary trees, so each node will have links to a right and left child. Figure 7.1 shows a diagram for a binary tree node and pseudo-code for the basic node structure is as follows:

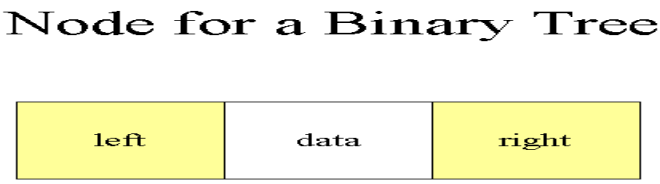


Figure 7.1 This figure illustrates the data fields of a binary tree node. The left and right links are of type treeNode. The data field contains application specific data; for simplicity and clarity a single integer is used here.

```
class treeNode {
     int data;
     treeNode left, right;
}
```

This code is nearly identical to what would be used in JAVA. In JAVA, the data items would need methods to access them, or they could be declared as "public". Below is a sample of JAVA code for a node object. Note that many times the “public” keyword is not required, depending on how the treenode is being used.

```
class treeNode {
     public int data;
     public treeNode left, right;
}
```

As discussed earlier in the linked list chapter, in C a structure would be used to create the tree node. The link to the next node will be an address (pointer) variable of the type node. The C code is presented below:

```
struct treeNode {
     int data;
     treeNode *left, *right;
};
```

In C++ instead of using a "struct", a class is used. Below, a very basic C++ class for the node is presented. Note that the word public is used so that object data can be accessed without using get and set methods.

```
class treeNode {
     public : int data;
     public :treeNode *left, *right;
};
```

The root of the tree is of type treeNode, just as the head/front of the linked list of type node. Figure 7.2 below shows the node structure of a complete binary tree with 7 nodes. Notice that the left and right links of the leaf nodes point to null.

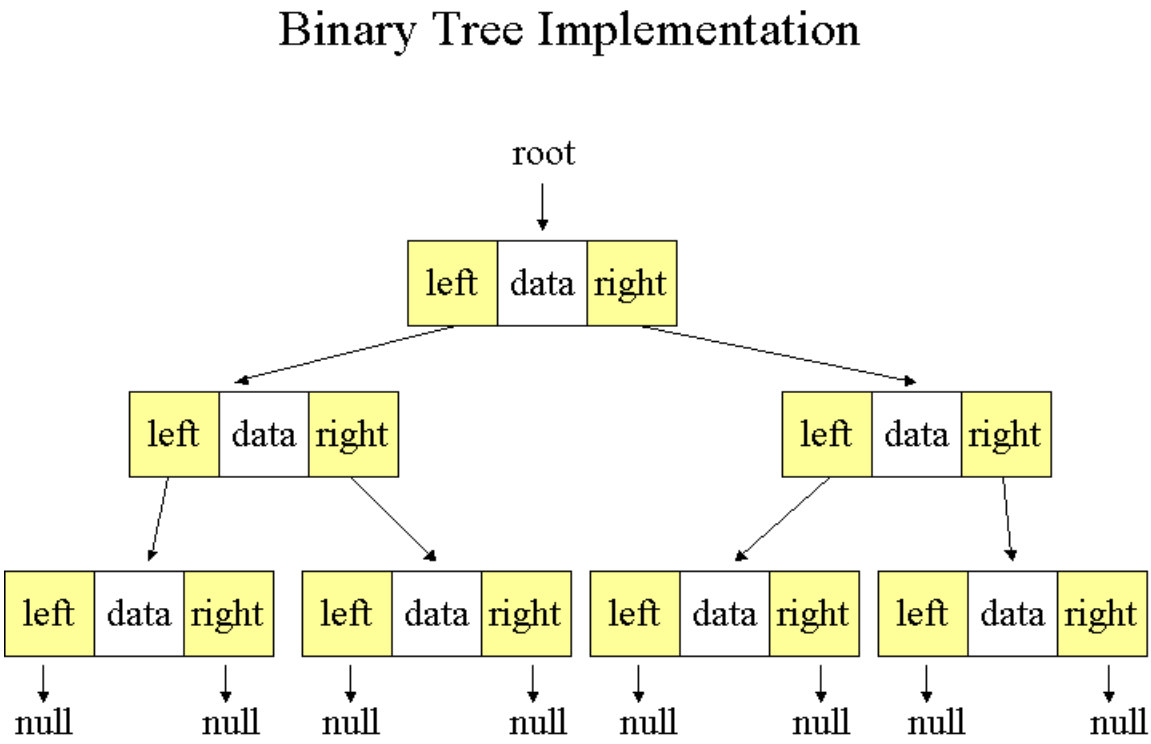


Figure 7.2 Shown is a diagram illustrating the nodes and links of a 3 level complete binary tree. The variable root is of type treeNode; it affords an entry point into the tree. Note that each of leave's left and right pointers are directed to null.

7.1.2 Fundamental Tree Methods

The methods for building and using tree are (once again) similar to those of lists.

- initTree()
- makeTreeNode()
- setLeft()
- setRight()
- inOrder()

The initTree() function sets the root variable to null. When adding nodes to the list, the makeTreeNode() function is used. It works in a very similar manner to the makeNode() function used in lists section. The setLeft() and setRight() functions add nodes (leaves) to the tree during the tree building process. To traverse the tree from left to right, the inOrder() function is called. The next sections provide a brief description of each of the basic tree methods.

7.1.2.1 Initialization Method - init()

The init method sets the root of the tree to null, creating an empty tree.

```
initTree()
{
     root = null;
}
```

7.1.2.2 Node Creation - makeTreeNode()

The makeTeeNode method allocates a new tree node object, initializes its data field, and sets the right and left links to null; The diagram in figure 7.3 illustrates the results of a call to makeTreeNode.

```
treeNode makeTreeNode(int data)
{
     teeNode treeNode;

     treeNode = allocate memory for a new node;
     treeNode.data = data;
     teeNode.left = null;
     treeNode.right = null;

     return treeNode;
}
```

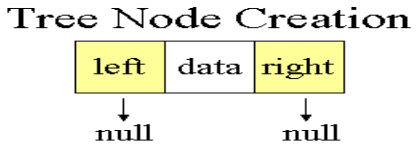


Figure 7.3 shows the results of a call to makeTreeNode().

The pseudo code below is a start on a procedure to create a binary tree that can used to sort numbers. The idea is start with a list of numbers and place them into the tree one at a time in the correct place so that later the tree can be traversed and the numbers can be printed out in ascending order. The code first initializes the root to null and then enters a loop that reads the numbers from the list one at a time. If the tree is empty, the number read from the list becomes the root (using a call to makeTreeNode()), otherwise it will be put into the tree in the correct place.

```
BuildBinaryTree(integer data list)
{
     initTree();
     while (there is data in the list) {
          s = getIntData();
          if (root == null) {
               root = makeTreeNode(s);
          }
          else {
          ....
          }
     }
}
```

In order to finish the procedure, a few more fundamental routines are needed. They are covered in the following sections.

7.1.2.3 Adding Nodes to the Tree - setLeft(), setRight()

The setLeft() and setRight() methods use the makeTreeNode() method to create a node and attach it to an existing tree. Before a node is added, they check to make sure that one does not already exist; if a node exist and it is written over, data will be lost. It can be noted that many implementations of a basic tree building algorithm incorporate the checks made by setLeft() and setRight(), so often times these routines are not used, as will be detailed in later sections. The pseudo code follows below:

```
void setLeft(treeNode t, int num)
{
     if (t.left != null) {
          print("Error: Left node already exists.");
     }
     else {
          t.left = makeTreeNode(num);
     }
}


void setRight(treeNode t, int num)
{
     if (t.right != null) {
          print("Error: Right node already exists.");
     }
     else {
          t.right = makeTreeNode(num);
     }
}
```

7.1.2.4 Building the Binary Tree

To finish out the construction of the binary tree used for sorting, we must develop the ability to put the numbers in the correct place in the tree. Given a list of unsorted numbers of an arbitrary length with no duplicates, the strategy to build a tree to be used for sorting is as follows:

- Let the first number in the list be the root.
- For each of the following numbers, find their position in the tree by starting at the root and working down the tree until the correct place is found.
    - To do so, starting at the root of the tree, if the current number is less than the number at the root, follow the left branch, otherwise, follow the right branch.
    - Continue following the branches using the strategy of going left if the number is less than the current nodes value, and right otherwise.
    - At some point the branch that needs to be followed will be null, this is when the number is added using the setLeft, or setRight method.

The pseudo-code below, using setRight() and setLeft(), shows the whole process:

```
BuildBinaryTree(integer data list)
{
      initTree();
      while (there is data in the list) {
            s = getIntData();

            /* Create the root of the tree. */
            if (root == null) {
                  root = makeTreeNode(s);
            }
            else {
                  /* Find number's position in the tree. */
                  curr = root;
                  searching = TRUE;
                  while(searching) {
                        if (s < curr.data) {
                              if (curr.left != null) {
                                    curr = curr.left;
                              }
                              else {
                                    searching = FALSE;
                              }
                        }
                        else {
                              if (curr.right != null) {
                                    curr = curr.right;
                              }
                              else {
                                    searching = FALSE;
                              }
                        }
                  } /* End while searching. */

                  /* Add the number into the tree. */
                  if (num < curr.data) {
                        setLeft(curr, num);
                  }
```

```
                  else {
                        setRight(curr, num);
                  }
            } /* End else. */

      } /* End while there is still data in the list. */
}
```

Alternatively, a bit cleaner algorithm follows, this one does not make use of setRight() and setLeft(). For some this routine is preferable. Notice that in the while(searching) loop, each of the comparison cases, equal, less than, and greater than, are clearly shown. Also, the insertion into the tree is handled in the search loop, instead of after the fact. This makes for a more compact, clear, and arguably cleaner routine.

```
buildSimple(integer data list)
{
      initTree();

      while (there is data in the list) {
            s = getListDataItem();

            /* Create the root of the tree. */
            if (root == null) {
                  root = makeTreeNode(s);
            }
            else {
                  /* Find number's position in the tree. */
                  curr = root;
                  searching = TRUE;
                  while(searching) {
                        if (s == curr.data) {
                              print("Found a duplicate!");
                              searching = FALSE;
                        }
                        else if (s < curr.data) {
                              if (curr.left != null) {
                                    curr = curr.left;
                              }
                              else {
                                    curr.left = makeTreeNode(s);
                                    searching = FALSE;
                              }
                        }
                        else if (s > curr.data) {
                              if (curr.right != null) {
                                    curr = curr.right;
                              }
                              else {
                                    curr.right = makeTreeNode(s);
                                    searching = FALSE;
                              }
                        }
                  } /* End while searching. */

            } /* End else. */

      } /* End while there is still data in the list. */
}
```

It is useful to trace the code with a short list of numbers. The following trace is for the BuildBinaryTree() algorithm.

Using the numbers from the list:

list = {4, 6, 5, 8, 7, 9, 1, 11}

Because 4 is the first number in the list, it becomes the root. The variable s is given the value of the next number in the list, 6.

Since the root has already been created, the proper place must be found for the 6. The program enters the while(searching) loop and follows the branches. 6 is determined to be greater than 4, but since there is no right branch from the 4, the search ends and the loop exits. Next, again since the 6 is greater than the 4, the setRight() method is called and the 6 is added as the right child of the 4.

The 5 is removed from the list and once again the while(seraching) loop is entered. The 5 is greater than the 4, so the right branch of the tree is followed. Next the 5 is compared to 6. It is less than the 6, so an attempt to follow the 6's left branch is made, but the branch does not exist, so the search loop is exited, and a node containing the 5 is created as the 6's left child.

The procedure is similar for the remaining numbers. This is left as an exercise for the reader. Figure 7.4 below shows the resulting trees at each phase of construction.

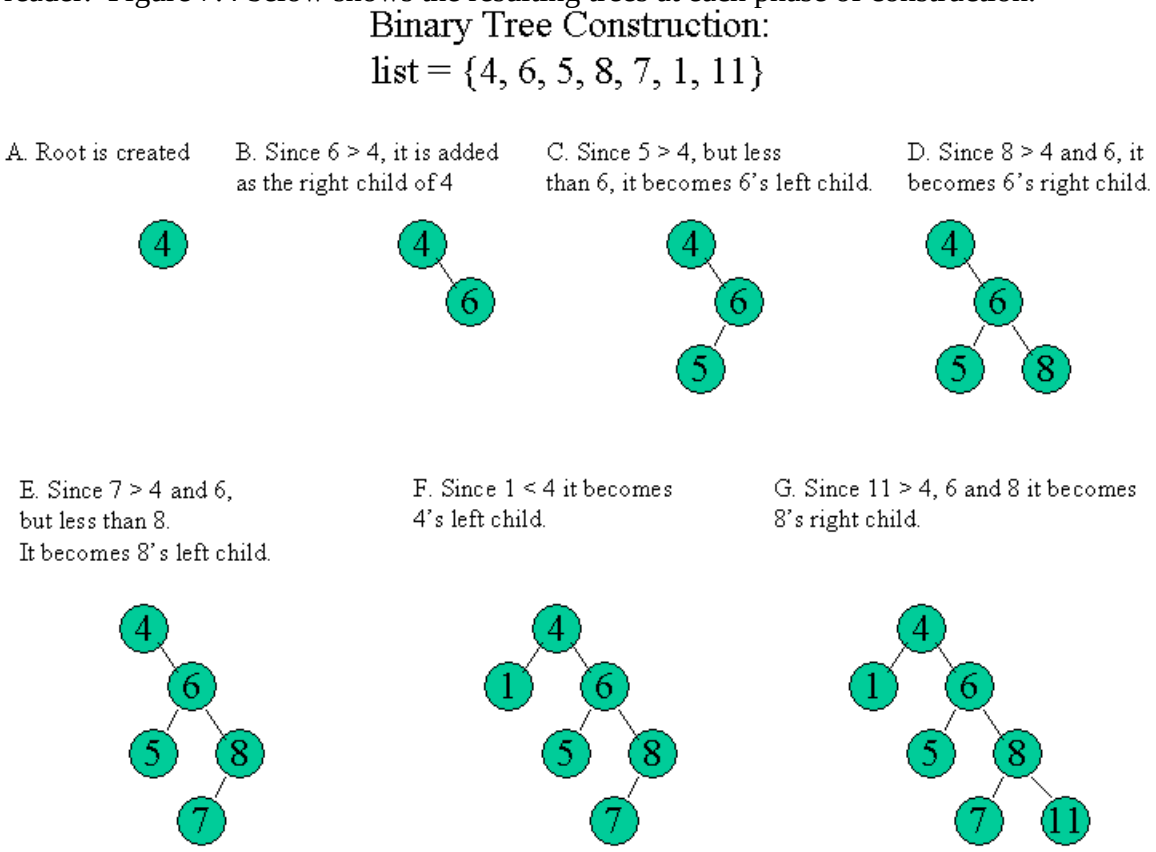


Figure 7.4 Binary tree construction for the list {4,6, 5, 8, 7, 1, 11}

The reader is encouraged to complete this trace and do the trace with the buildSimple() algorithm.

7.1.2.2 Showing the contents the Tree - Inorder Traversal - inOrder()

One of the easiest ways to show the contents of a tree is use a recursive traversal. When traversing binary trees, there are three primary ways to do so; the methods differ in when they visit the root of a sub-tree.

To demonstrate the idea, consider a binary tree with just 3 nodes, as shown in figure 7.5 below. If the root is always visited first, and then the left sub-tree, followed by the right, we call this "pre-order". If the traversal follows the left branch, visits that node, returns to the root, and makes a visit, and finally follows the right branch to visit that node, we call this "in-order". When the root is visited after first visiting the nodes at the left and right branches, we call this "post-order". The pre-fix (pre, in, post) of the visitation method refers to when the root is visited.

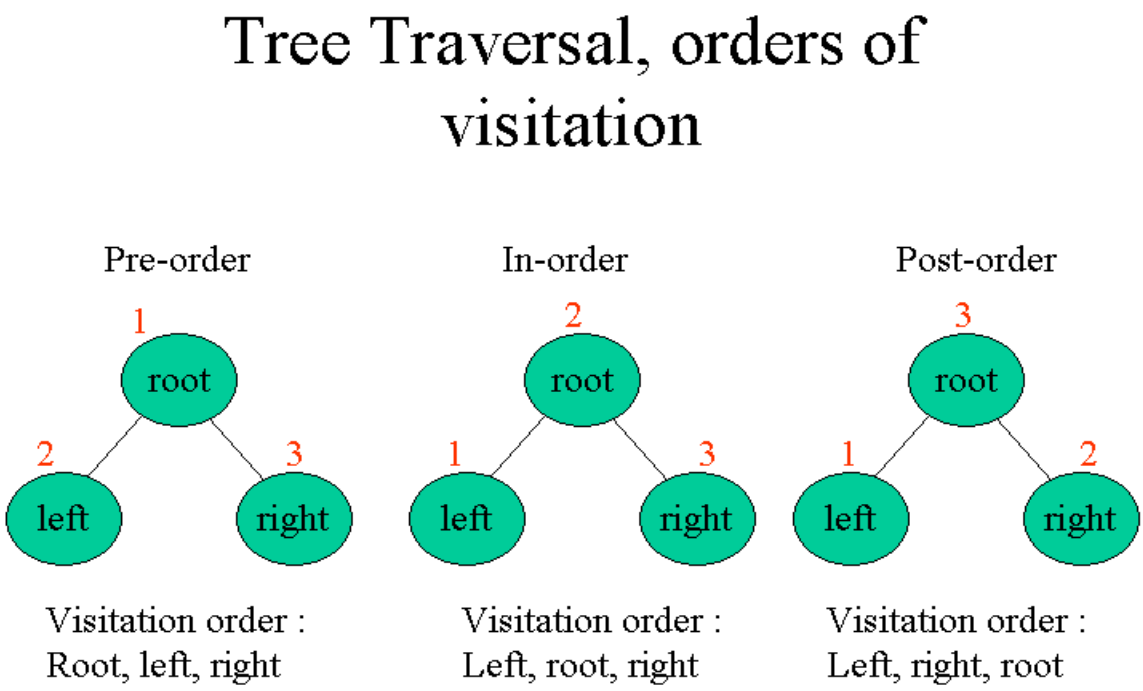


Figure 7.5 This figure illustrates the three primary ways to traverse a binary tree. Note that the red numbers near the nodes indicate the order of visitation.

With trees that have more than 3 nodes, the visitation order is very similar. Consider a complete binary tree with 7 nodes as shown in figure 7.6 below.

7 Node Complete Binary Tree

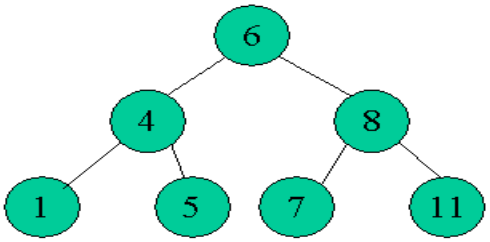


Figure 7.6. This figure illustrates a 7-node complete binary tree that can be used for sorting. It was built using the algorithm outlined in the pseudo code shown in section 7.1.2.4. Notice that all nodes to left of the root node are less than the root, and all nodes to the right of the root are greater than the root. This convention also holds for each of the sub-trees.

Earlier, we stated that the in-order traversal operates by going left first, visiting the root, then going right. For the 3 node trees, once this sequence was complete the tree traversal was complete. For a larger tree, the same general rules apply, that is left first, root, then right, except we go as far left as we can, then visit the root, then go right, and then start the process all over again. That means that as soon as we go right, we go left if we can.

When we get to a leaf node what happens? If you cannot go left (you can't, it's a leaf) you print the root (the value of the leaf), then you try to go right, but you can't, it's a leaf. What now? You return to the root (which has already been visited), and then try to go right again. This idea is encapsulated in the following recursive in-order traversal algorithm:

```
in-order(treeNode t)
{
     /* Go Left! */
     if (t.left != null) {
          in-order(t.left);
     }

     /* Visit root! */
     print(t.data);

     /* Go right. */
     if (t.right != null) {
          in-order(t.right);
     }
}
```

The table below shows a trace of the in-order algorithm on the tree from figure 7.6.

Step number	***Action***	***Result***
0	call in-order with root of tree in-order(6)	in-order(6)
1	try to go left	in-order(4)

2	try to go left	in-order(1)
3	try to go left, but cannot because no branch exist. Drop to print statement.	print "1"
4	try to go right, but cannot because no branch exist. The call to in-order(1) is now complete, so control returns to in-order(4) just after the call to in-order(1), i.e. the print statement.	print "4"
5	try to go right	in-order(5)
6	try to go left, cannot, move to print statement	print "5"
7	try to go right, cannot, return to in-order(4) after the call to go right, in-order(4) completes, returning to in-order(6) just after the call to go left, i.e. the print statement.	print "6"
8	try to go right	in-order(8)
9	try to go left	in-order(7)
10	try to go left, cannot, so move to print statement.	print "7"
11	try to right, cannot, return to in-order(8), after call to go left, i.e. the print statement	print "8"
12	from in-order(8) try to right	in-order(11)
13	try to go left, cannot, move to print statement	print "11"
14	try to go right, cannot, return to in-order(8) after the call to go right	
15	in-order(8) completes, returning to in-order(6), after the call to go right. Now there are no more recursive calls, so traversal is complete	Recursive process is complete.

Notice that the in-order traversal prints the numbers out from low to high. A good exercise would be to create a routine to print the numbers out from high to low. Thinking about this, there are two easy ways to do it. One has to do with modification of the tree build, the other has to do with modification of the tree traversal. What happens if a

student is very enthusiastic and does both? Well, the short answer is that the tree will be printed out from low to high, just as if they had made no modifications. Why?

7.1.2.3 Iterative Traversal of a Tree

The iterative tree traversal works almost just like the recursive traversal, but instead using recursion to work down to the leftmost tree node, it uses a stack. Just as in the recursive routine, once it cannot go any further left, it prints the data from that node, and then tries to go right. Below we present some very JAVA like pseudo-code for the iterative inorder traversal.

```
public void inOrderI(treeNode root)
{
   treeNode curr;
   nodeStack s;

   System.out.println("Semi-horrible iterative
     Traversal.....");

   s = new nodeStack();
   s.init();

   s.push(root);
   curr = root;
   do {
      /* Go left. */
      while(curr != null) {
         s.push(curr);
         curr = curr.left;
      }

      /* Print data, and try to go right. */
      if (!s.empty()) {
         curr = s.pop();
         System.out.println("visiting "+curr.info);
         curr = curr.right;
      }

   }while(!s.empty()||(curr!=null));
}
```

Looking at the code, the moving left portion is accomplished by pushing tree nodes on to the stack in a while loop that ends when curr.left is equal to null. Once that occurs, the while loop is exited, and if the stack is not empty, the last item pushed onto the stack is popped and it's value is printed. Now, just as in the recursive version, a move to the right is attempted. This process continues until there are no more items on the stack, and the current position points to null. Thinking about it, if the traversal has worked it's way to the rightmost lower node of the tree, the node does not have to be a leaf, it can have a left branch, but that one will have already been explored, just as in the recursive search, and all of the tree nodes above it will have been pushed and popped, so the stack will be empty, and since curr was set to curr.right, curr will be null, so the conditions for ending

the traversal will have been satisfied. To really understand how this routine works it is best to draw out a small tree and trace the code.

7.2 Complexity Issues

A good question to ask would be "Why go to all the trouble to put data in a tree when using an array would be much easier to implement?". The answer is efficiency. When looking for a number in a binary tree, the most comparisons that can be made is going to equal to the height of the tree. If using an array, the search of array is of order O(n). With small data sets, it may not make a difference, but with large ones it will. With a perfectly formed binary tree, the search will take no more than $\log_2 n$ comparisons. Looking at table 7.1 below we can see that as the data set size grows, an O(n) search is at a huge disadvantage to an order O($\log_2 n$) search (remember, the height of the tree is estimated by taking the log base 2 of the number of items in the tree).

Array size to be searched	Max Sequential Search Comparisons	Max Tree Comparisons for a Complete Binary Tree
16	16	4
128	128	7
1024	1024	10
8192	8192	13
1048576	1048576	20

Table 7.1 This table shows the number of comparisons needed in a worst case scenario when searching an array versus searching a complete binary tree. If the number being searched for is the last number in the array, it takes n comparisons. If the number is at the very bottom of the tree, its takes $\log_2 n$ comparisons.

One thing to remember is that most of the time when building a binary tree, the tree will not be a complete binary tree. At the worst all the data could be to one side or the other of the root, making searches of this horribly formed tree O(n). But that situation is most unlikely. However, it would not be unexpected for the tree to be twice or even three times the height of the complete binary tree. But even in such un-ideally formed trees, if the data set is larger than 8 or so, the binary tree comes out on top. For example, suppose that we build a somewhat scraggly tree out of 8192 data items and our tree is 3 times the height of a nice complete binary tree. Even with the scraggly tree, the max number of comparisons is 40, which is way less than 8192.

In summary, when working with trees we trade code complexity for speed. If the data sets are consistently small, then it may not be worth the extra trouble. But once they get just a bit bigger the binary tree has a huge speed advantage.

The last point to be made is about the traversals. As mentioned in chapter 4, recursion is a very powerful tool, but it does take up resources. If the data sets are small the inorder recursive traversal is the way to go, and for searching the recursive search will work fine. But if the data sets are large and more than 20 or so recursive calls are needed to get to the bottom of the tree, the iterative searches and traversals should be used.

7.3 Exercises

1. Write a program that will build an ordered binary tree from a list of numbers that has duplicates in it. Do not store the duplicates, instead modify the treeNode class to take this into account. Your answer should contain:
    a. A modified treeNode class.
    b. An in-order traversal of the tree should produce a list of sorted numbers.
2. Write recursive code that accomplishes the following tree traversals.
    a. Pre-order
    b. In-order
    c. Post-order
3. Modify the in-order traversal function so that numbers can be printed out from high to low.
4. This exercise is about trees and computational efficiency. Your job is to write/use the work done in exercise 1 to create a tree from a data set read in from a file. Using the tree, you are to sort the numbers, and write out the data set to an output file. The sort will take place as you read the numbers and place them in the correct locations within the tree.
    a. The data part of your tree node should store the number and keep track of the number of occurrences of the number stored; this will be how you track duplicates.
    b. While you are building the tree, you need to count the number of compares you make.
    c. After you have built the tree, you need to write out the data so that the data is sorted from low to high.
    d. You need to write the tree out using both the recursive and iterative methods.
    e. At the end of the file, you need to write out your program's operating statistics, that is, how many compares were made in creating the tree and writing the data out.
    f. One aspect of the assignment is to generate a list of 1000 random integers between 0 and 100. One easy way to do this is use a program like Micro Soft Excel. You can use the random number generator by typing into a spreadsheet cell "= rand()*100". Then format the number so that it has no decimal places (now it is an integer), and finally just extend it out so that you 1000 numbers. Once this is accomplished, highlight the numbers and copy them, create a text file using the notepad program, and copy the numbers into the file and save it.
    g. Write a brute force sort and sort the same data. In your sorting program, you are to record the number of compares during the sort. So in this program you will read the data into a large array, do your brute force sort (a bubble sort will work; the bubble sort is covered in many texts and in this set of notes in chapter 11), then write the array to a file. The output of the program will be the number of comparisons made during the sort, and the sorted numbers.
    h. Which program is more efficient? Based on the number of compares made? Based on the memory used?

Chapter 8. Graphs

In the two previous chapters, linked lists and trees were discussed. Both linked lists and trees have unique configurations, but each has a specific entry point. In the case of the linked list, it is the head/front of the list, and in the case of the tree, it is the root. In this chapter, graphs are discussed and like trees and lists, graphs are comprised of nodes (called *vertices*) linked together with connections called *edges*, but unlike trees and lists, graphs are not required to have a designated "root" or "head". In very general language, graphs have less requirements about their general structure than either trees or lists. In fact, it is not a stretch to say that trees and linked lists are a subset of the class called graphs.

Figure 8.1 shows three examples of graphs. 8.1a shows a 5 node graph in which vertex c is disjoint, meaning there is no way to get to from any of the other vertices, and there is no way to get from it to any of the other vertices. 8.1b shows the utility of graphs. This figure could represent a computer network, a map, or some other application in which distances between vertices can be assigned costs. Figure 8.1.c shows a graph that looks like a tree.

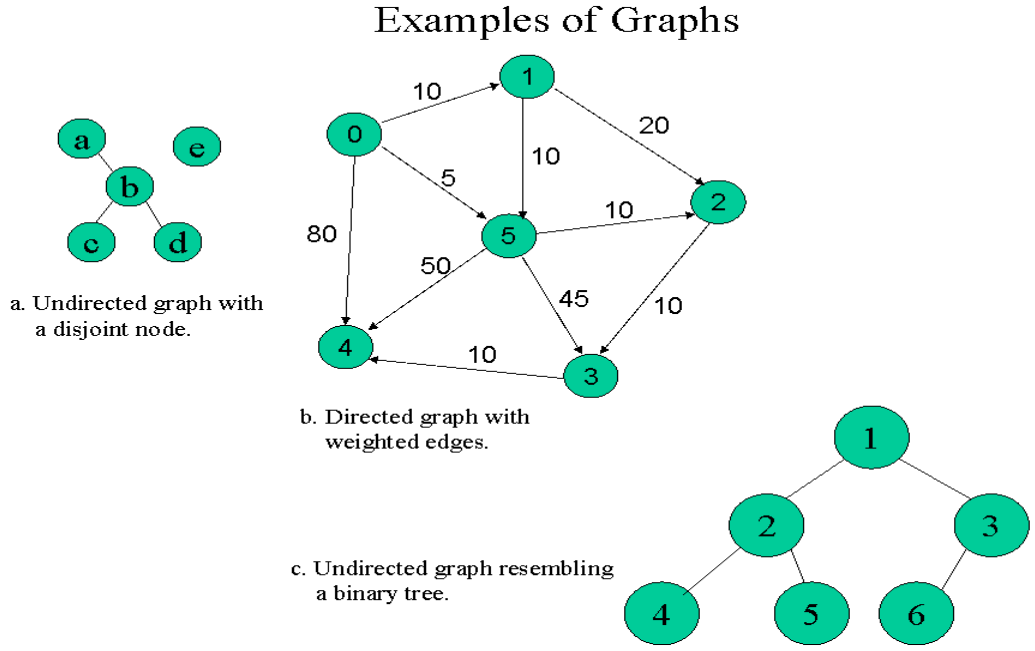


Figure 8.1 This figure shows 3 examples of graphs. In example a, there is a vertex that is not connected to any of the other vertices. It is still part of the graph; it is disjoint. In example b, the edges are directed and have weights representing costs or distances. Example c shows a graph that looks like a tree.

8.1 Definitions and Concepts

More formally, a graph is a collection of nodes called vertices, connected by edges. Edges perform a similar job as the links between nodes in trees and lists. But these edges can be directional, meaning if there is a directional edge from vertex A to vertex B, then a direct path exists from A to B, but not from B to A. In the case that the edge is non-

directional, undirected, a path is present in both directions. We say that a graph G is the set of vertices and edges, G = {V, E}. The connectivity of G is described by a special matrix called an adjacency matrix, which will be described in later sections. A *path* from one vertex, A, to another, Z, exists if there is a sequence of (vertex, edge) pairs that can be followed from vertex A, and eventually arrive at vertex Z. A *cycle* exists if the beginning and end a path are the same vertex. The fact that cycles are a legal part of graphs is a large distinction between graphs and trees.

8.1.1 Graphs and the Adjacency Matrix

The adjacency matrix is a square two-dimensional matrix which holds the connectivity information of a graph. The size of the matrix is based on the number of vertices in the matrix. For example, if a graph has 4 vertices the adjacency matrix will be a 4 by 4 matrix.

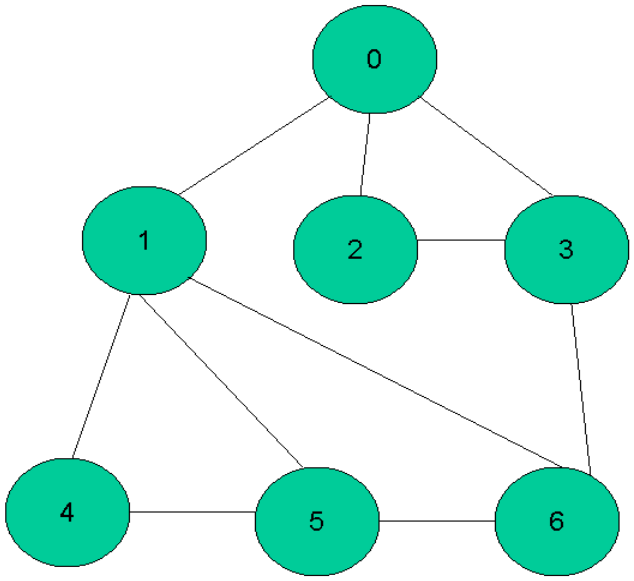
A path represented in the matrix has source vertex and a destination vertex. In the matrix, the rows are the source vertices and columns are the destination vertices. So if a path exists in a graph from node 2 to node 4, the array/matrix position adjacency[2][4] would have 1 in it (or a positive number representing the paths length) to indicate that a direct connection exists from node 2 to node 4. If no direct path exists, then the position has a 0 in it. Often in the literature, for clarity purposes, the matrix is displayed showing only the paths (the 0's and the diagonal of the matrix are left blank).

The matrix can of type integer or Boolean, depending on what type of problem is being solved. For problems like traversals, a Boolean matrix is sufficient, but problems involving, path connection counts or shortest paths require an integer or floating-point matrix.

Figure 8.2 below shows an example of an undirected graph with 7 nodes, labeled 0 through 6. The matrix to the right of the graph is the adjacency matrix. It is a square matrix of size (7 x 7). In this adjacency matrix the diagonal is filled with dashes to indicate that there is no direct path from a node to itself, but there are instances where this type of path can exist. Looking closely at the diagram and the matrix it can be seen that if a path exists between two nodes, its existence has been indicated with a 1 in the adjacency[source][destination] position. In later sections, the edges will be given values that may be 1 or more to indicate the value of parameters such as distance or cost between two nodes.

As stated earlier, most algorithms that work with graphs implanted it using a two-dimensional array. In languages like C and Java the array indices start at 0, so labeling the nodes from 0 to the number of nodes will create less confusion that starting the node count at 1. Also, regarding undirected graphs, the adjacency matrix is always symmetric about the diagonal; this is an easy way to check to see if the adjacency matrix and the graph match each other. Looking at figure 8.1 the symmetry can be seen, starting with the row 1 matching column 1, row 2 matching column 2 and so forth.

Undirected Graph and Accompanying Adjacency Matrix



Destination Node

Source Node

	0	1	2	3	4	5	6
0	-	1	1	1			
1	1	-			1	1	1
2	1		-	1			
3	1		1	-			1
4		1			-	1	
5		1			1	-	1
6		1		1		1	-

Figure 8.2 In this figure there is an undirected graph with 7 vertices. The lines between the graphs are edges. The table to the right of the graph is the graphs adjacency matrix. The adjacency matrix is a square matrix that has a 1 in every position where there is direct connection between two nodes. For example, row 0 has 1's in columns 1, 2, and 3, representing those edges.

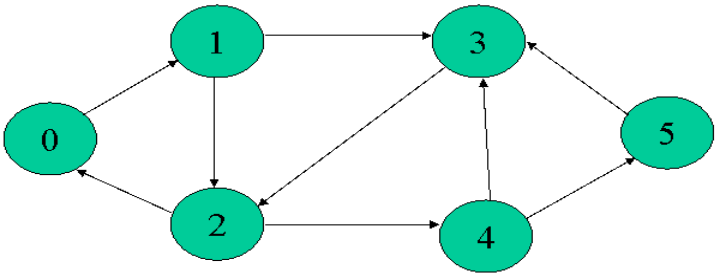
Figure 8.3 below shows a directed graph and its accompanying adjacency matrix. It is very similar to the undirected example in most every aspect, except that it is symmetric.

8.2 Graph Traversals

In the previous chapter traversal of trees was discussed using both recursion and iterative implementations. Graph traversals are similar except that because graphs can have cycles the traversal algorithm must keep track of which vertices have been visited. If it does not, the traversal will get caught looping through the cycles and will not be able to complete. The simplest way to keep track of which vertices have been visited is by using a Boolean array. The elements of the array are initially false, and as each individual vertex is visited, its corresponding element is set to be true.

In this section, recursive and iterative traversals will be presented; both will use the Boolean tracking array. Also, two different orders of visitation, depth first and breadth first, will be discussed. In the iterative implementations the algorithms are very closely related to one another, with the difference being that the depth first search uses a stack, while the breadth first search uses a queue.

Directed Graph and Accompanying Adjacency Matrix



Destination Node

Source Node

	0	1	2	3	4	5
0	-	1				
1		-	1	1		
2	1		-		1	
3			1	-		
4				1	-	1
5				1		-

Figure 8.3 In this figure there is a directed graph with 6 vertices. The arrowed lines between the vertices indicate one-way direct paths. For instance, there is a direct path from vertex 0 to vertex 1, but no direct path exists from 1 back to 0. The adjacency matrix to the right of the graph defines the connectivity between the vertices of the graph.

8.2.1 Depth First Search

The depth first search algorithm is named for the pattern of visitation. It digs down into a graph as opposed to the gradually spreading pattern generated by the breadth first search. As stated earlier, the key to making the algorithm work is recording which vertices have been visited. By doing this, the algorithm can avoid continually revisiting vertices. Why was this not a consideration in the tree traversals of the previous chapters? Because by definition trees do not have cycles! Below the pseudo code for a recursive depth first search is presented.

Prior to the call to depthFirst(), the visited array is initialized as shown below:

```
for(j=0;j<nVertices;j++) {
     visited[j] = false;
}
```

Now we present the recursive depth first search:

```
depthFirst(vertex v)
{
     print("we are visiting vertex", v);
     vistited[v] = true;

     for(j=0;j<nVertices;j++) {
        if (j != v) {
          if ((adjacency[v][j]==1)and(visited[j]==false)) {
               newV = j;
```

```
                depthFirst(newV);
            }
        }
    }
}
```

Figure 8.4 below shows the search pattern for a 7-vertex graph, as shown in 8.4a. For this example, the search was started at vertex 0. The algorithm works by looking into the current vertex's row in the adjacency matrix and making a recursive call to each vertex that has not been visited. 8.4b shows that the algorithm marked vertex 0 as visited, went into the for loop and found that node 1 was first node that was still unvisited and not the same node that the algorithm is currently visiting.

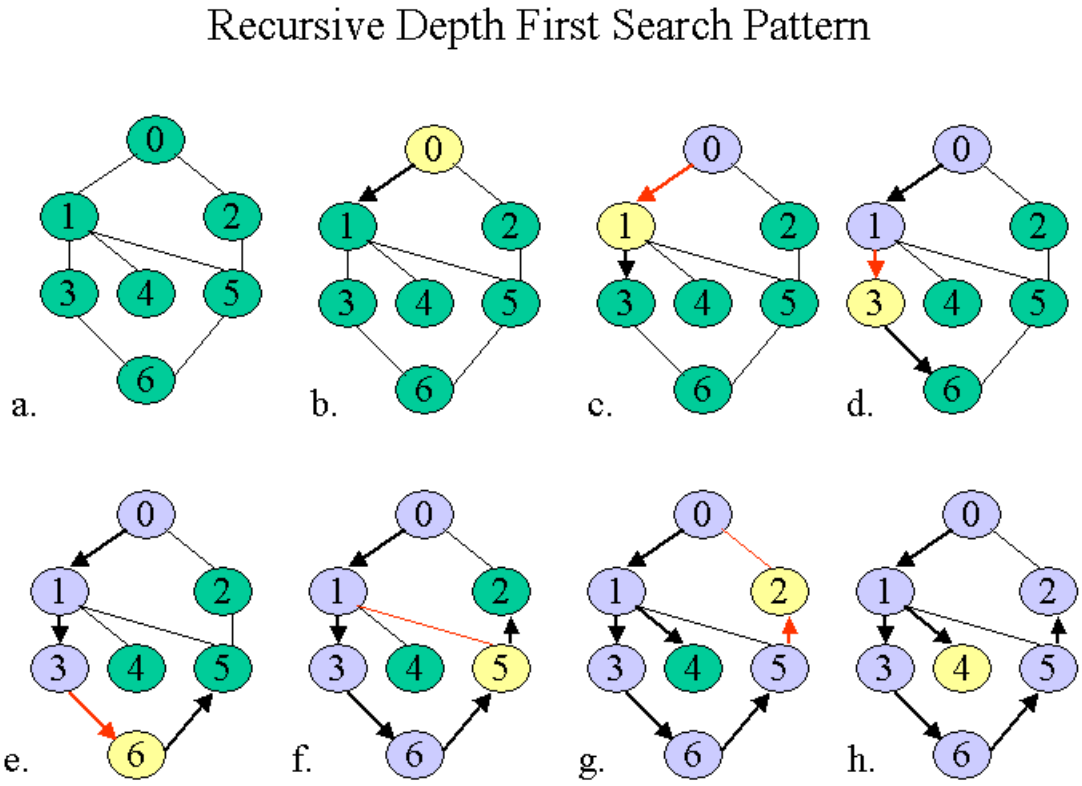


Figure 8.4 This figure show the pattern of visitation of a recursive depth first search on an undirected graph. The search starts at vertex 0. A green vertex indicates it has not been visited; yellow indicates the vertex that holds the current call to depthFirst(); blue indicates visited. A dark arrow such as the one in figure b that leads from vertex 0 to vertex 1 indicates the direction of progress of the search pattern. In this instance it is depthFirst(1). Red edges such as the one in figure f from vertex 5 to vertex 1 mean that the algorithm has checked to see if it go that direction but will not because the vertex has already been visited.

Table 8.5 below describes the algorithms actions for figure 8.4

Figure 8.4 Sub-diagram	***Actions***	***Results***
a	Graph initialized and ready to be traversed.	

b	• Call made to depthFirst(0)
	• Vertex 0 marked as visited;
	  ○ visited[0] = true;
	• for loop entered
	  ○ vertex 0 - visited
	  ○ vertex 1- not visited.
		Call made to depthFirst(1)

c	• Vertex 1 marked as visited;
	• for loop entered
	  ○ vertex 0 - visited
	  ○ vertex 1- do not check own vertex connectivity
	  ○ vertex 2 - not adjacent
	  ○ vertex 3 - not visited
		Call made to depthFirst(3)

d	• Vertex 3 marked as visited;
	• for loop entered
	  ○ vertex 0 - visited/not adjacent
	  ○ vertex 1 - visited
	  ○ vertex 2- not adjacent
	  ○ vertex 3 - do not check own vertex connectivity
	  ○ vertex 4 - not adjacent
	  ○ vertex 5 - not adjacent
	  ○ vertex 6 - not visited
		Call made to depthFirst(6)

e	• Vertex 6 marked as visited;
	• for loop entered
	  ○ vertex 0 - visited/not adjacent
	  ○ vertex 1 - visited/not adjacent
	  ○ vertex 2- not adjacent
	  ○ vertex 3 - visited
	  ○ vertex 4 - not adjacent
	  ○ vertex 5 - not visited
		Call made to depthFirst(5)

f	• Vertex 5 marked as visited;
	• for loop entered
	  ○ vertex 0 - visited/not adjacent
	  ○ vertex 1 - visited
		Call made to depthFirst(2)

	○ vertex 2 - not visited	
g	• Vertex 2 marked as visited; • for loop entered ○ vertex 0 - visited ○ vertex 1 - visited/not adjacent ○ vertex 2 - do not check own vertex connectivity ○ vertex 3 - visited/not adjacent ○ vertex 4 - not adjacent ○ vertex 5 - visited ○ vertex 6 - visited/not adjacent	• return from depthFirst(2) to depthFirst(5); • no call can be made, no unvisited adjacent nodes, return from depthFirst(5) to depthFirst(6) • no call can be made, return from depthFirst(6) to depthFirst(3) • no call can be made, return from depthFirst(3) to depthFirst(1) • Call can be made: depthFirst(4)
h	• Vertex 4 marked as visited; • for loop entered ○ all nodes visited, no calls made	• return from depthFirst(4) to depthFirst(1) • no call can be made return from depthFirst(1) to depthFirst(0) • no call can be made return from depthFirst(0) to calling routine. Depth first search completed.

Table 8.5. This table describes the action of the depth first search for the graph in figure 8.4

The depth first search can also be programmed iteratively. There are some advantages to the iterative solution, primarily because iterative solutions are more memory efficient. This will make a difference when working with a graph with many interconnected vertices because iterative version will not suffer from the memory use of multiple recursive calls.

The iterative version of the depth first search uses a stack to store the vertices adjacent to the current node, versus a recursive call as shown in the algorithm in the previous section. The first thing the algorithm does is push the starting node of the traversal. The outer loop consists of a while loop whose ending condition is an empty stack. Alternatively, the while loop could end when all nodes of the graph were visited. The inner loop of the

algorithm works in similar manner to the for loop in the recursive version. That is, the for loop's job is to find the unvisited adjacent nodes. The pseudo-code for the iterative depth first search is shown below:

As before, the visited array is initialized as shown below:

```
for(j=0;j<nVertices;j++) {
     visited[j] = false;
}
```

Also, do not forget to initialize the ***stack***. Below is the pseudo-code for the iterative depth first search:

```
void depthFirstIterative(vertex v)
{
   int j;

   push(v);

   while(stack is not empty) {
      v = pop stack until unvisited vertex is found;
      visit[v] = true;

      for(j=0;j<nVertices;j++) {
         if (j != v) {
            if (adjacency[v][j] == 1) &&
               (visited[j] == false) {
               push(j);
            }
         }

      } /* End for */

   } /* End while */
}
```

The algorithms accomplish the same thing, but the search pattern is different. This is because of the order that the stack stores and retrieves the unvisited nodes. The graphs in figure 8.6 below show the order of visitation for the iterative depth first search.

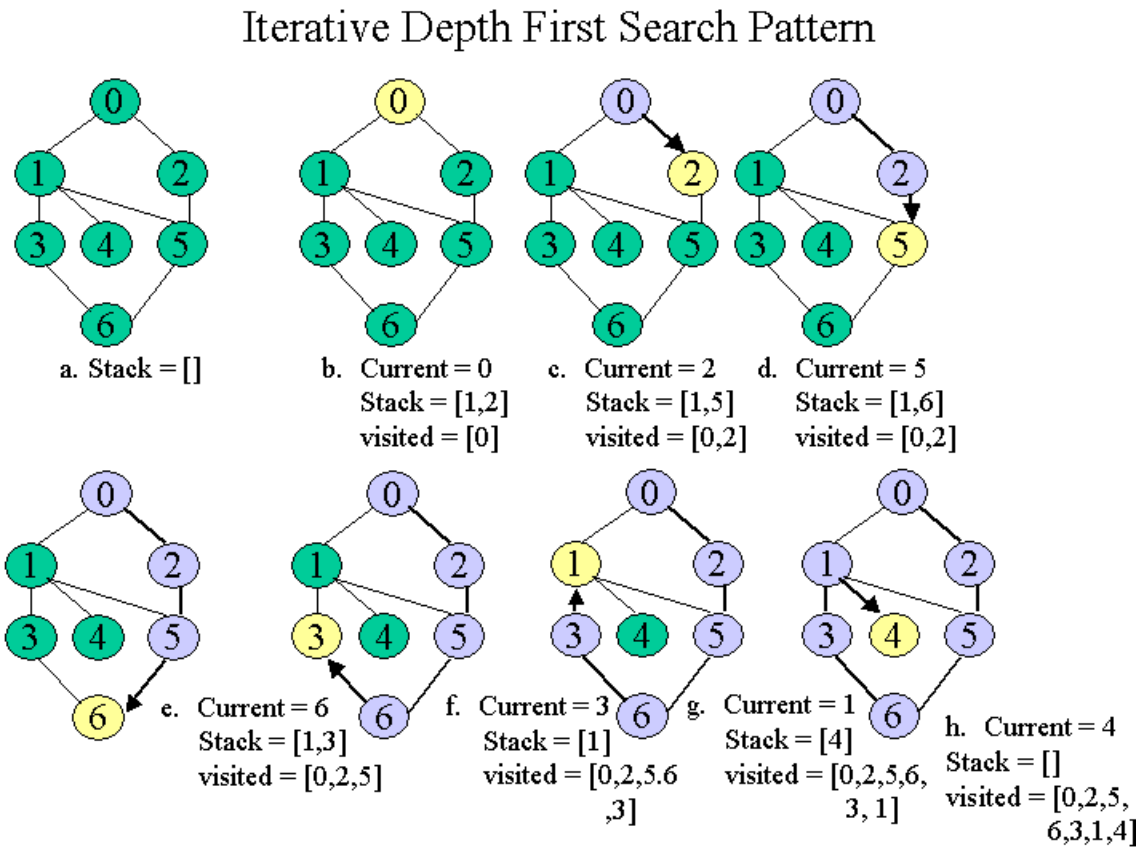


Figure 8.6 This figure shows the visitation pattern of the iterative depth first search starting at node 0 of an undirected graph. If a vertex is green that indicates it has not been visited; yellow indicates the vertex that holds the current vertex popped from the stack; blue indicates visited. A dark arrow such as the one in figure c that leads from vertex 0 to vertex 2 indicates the current direction of progress through the graph.

As can be seen, the iterative search pattern is nearly a mirror image of the pattern produced by the recursive version. The reason lies in the way the adjacency matrix is examined. In both cases a for loop is used, and the for loop starts at column 0 of the current vertex's row. In the recursive search, this results in the vertices of with low numbers being visited first. The iterative version pushes the low numbered vertices first, but because of the nature of a stack, they get popped last, so the high numbered vertices get visited first. To make the search patterns the same, putting the vertices onto the stack starting with the high numbers first would give the desired results.

8.2.3 Breadth First Search

Closely related to the depth first search is the breadth first search. In fact, the only difference between the iterative implementations is the use of a queue instead of a stack. The pseudo-code is shown below:

First, the visited array is initialized:

```
for(j=0;j<nVertices;j++) {
     Visited[j] = false;
}
```

Next, make sure to initialize the ***queue***. After that the code is nearly identical to that of the depth first search:

```
void depthFirstIterative(vertex v)
{
   int j;

   pushQue(v);

   while(queue is not empty) {
      v = pop queue until unvisited vertex is found;
      visit[v] = true;

      for(j=0;j<nVertices;j++) {
         if (j != v) {
            if (adjacency[v][j] == 1) &&
               (visited[j] == false) {
               pushQue(j);
            }
         }

      } /* End for */

   } /* End while */
}
```

This seemingly small change in the algorithm makes a big difference in the pattern of search as can be seen in figure 8.7 below. Notice the exploration pattern flows from top to bottom of the graph, in contrast to the depth first search that makes its way down into the graph. The breadth first search is the ideal way to accomplish a level-by-level search of a tree.

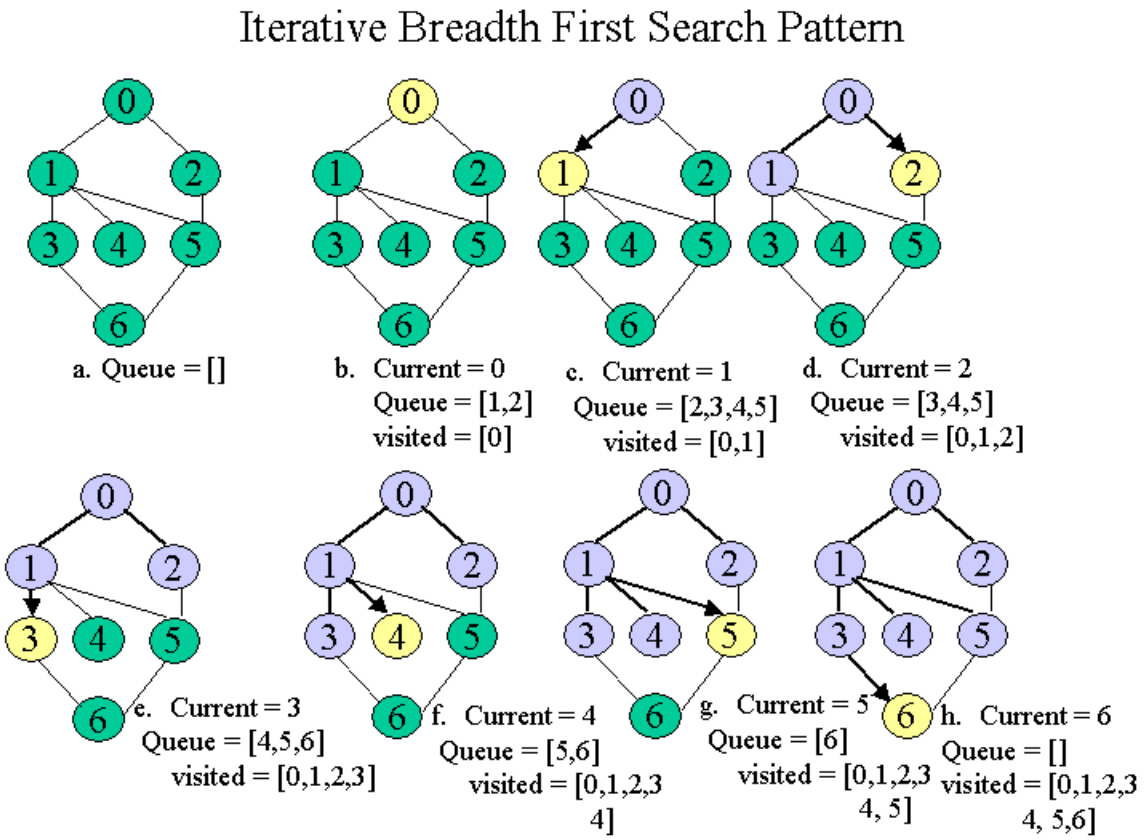


Figure 8.7 This figure shows the visitation pattern of the iterative breadth first search starting at node 0 of an undirected graph. If a vertex is green that indicates it has not been visited; yellow indicates the vertex that holds the current vertex popped from the queue; blue indicates visited. A dark arrow such as the one in figure c that leads from vertex 0 to vertex 1 indicates the current direction of progress through the graph. Notice the exploration pattern flows from top to bottom of the graph, in contrast to the depth first search that makes its way down into the graph.

8.3 Path Problems

One of the most common and useful aspects of graphs is the study of path problems. Common applications include shortest path problems as applied to geography and mapping, computer networking, pipe routing, etc.

In this section, two topics will be studied. The first, transitive closure, can be used to identify the number of paths and their lengths between the vertices in a graph. For instance, if we wanted to know the number of paths of length 4 between vertex A and vertex B, transitive closure could be used to tell us. Although it will tell you the number of paths, it will not provide the sequence of vertices in each path.

The next algorithm is Dijkstra's single source shortest path algorithm. It finds the shortest path from a source node to each of the other nodes in the graph. In doing so it also records the paths.

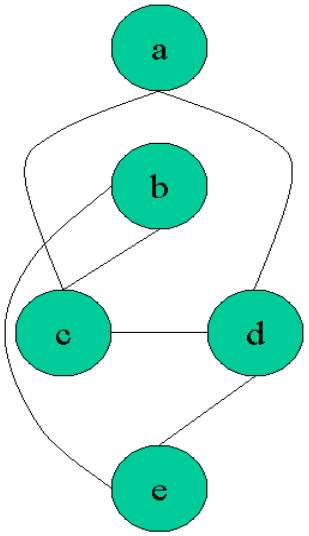
Finally, and worth mentioning, there is Floyds Algorithm. It finds the shortest path from every node to every other node. It works in a very similar way to Dijkstra's algorithm,

but in its basic form, it does not record the paths. Curious readers are encouraged to review Floyd's algorithm and other similar algorithms in the literature.

8.3.1 Transitive Closure

Transitive closure is a useful, straight forward method of determining the reachability of each vertex from each other vertex. For example, it may be useful to determine if paths of length 3 from vertex 'a' to vertex 'd' exist in the graph shown in figure 8.8 below.

Transitive Closure



Adjacency matrix (adj1)

Destination

Source	a	b	c	d	e
a	-		1	1	
b		-	1		1
c	1	1	-	1	
d	1		1	-	1
e		1		1	-

Figure 8.8 This figure shows an undirected graph with 5 vertices and its accompanying adjacency matrix. The diagonal of the matrix is denoted with dashes to indicate that no paths exist from a node to itself. Also, if a path does not exist, the position is left blank for clarity. Notice that the adjacency matrix shows symmetry.

Looking at the graph, we can find the following paths of length three from vertiex 'a' to 'd'.

- a – c – a – d
- a – d – a - d
- a – d – c – d
- a – d – e – d

This task can easily be done by inspection on small graphs. But, sometimes even on small graphs, the paths are not that obvious. Notice that in the case above, each of these paths includes cycles. Clearly, as the number of vertices increases the task gets more difficult.

To computationally solve the problem, multiplication of matrices is used. Recalling that the adjacency matrix indicates direct connections from source to destination vertices, i.e., the reachability of the vertices by using paths of length 1, the reachability of vertices

using a single intermediate vertex (two hops) can be determined by squaring the adjacency matrix. To find the reachability using three hops, the two-hop matrix can be multiplied by the adjacency matrix. To find the reachability using four hops, the three-hop matrix can be multiplied by adjacency matrix, and so on.

The equations below summarize the situation:

Let adj^1 = adjacency matrix

Let adj^n = matrix showing reachability in n-hops

$adj^2 = adj^1 \times adj^1 = (adj_1)^2$

$adj^3 = adj^1 \times adj^1 \times adj^1 = adj^1 \times adj^2 = (adj_1)^3$

$adj^n = adj^1 \times adj^1 \times adj^1 \ . \ . \ . \ adj^1 = adj^1 \times adj^{(n-1)} = (adj_1)^n$

It is probably more common use a logical "and" instead of the multiplication operator. By using the "and", the existence of paths is found instead of the number of paths.

To see why this process works, look at the adj2 matrix in figure 8.9. In the a to c position, there is 1 indicating a single path of length 2 from vertex a to vertex c. To get that value, row 'a' from the first adj1 matrix was multiplied by column 'c' in the second adj1 matrix.

adj2(a, c) = adj1(row a) x adj1(col c)

= (0*1) + (0*1) + (1*0) + (1*1) + (0*0)

= 1

The only place where there were two 1's multiplied by each other was in the (a to d) * (d to c) position of the multiplication; all other products resulted in 0. Looking closer, we can think of the a to d in the first matrix as a path of length 1 from the source vertex to the intermediate vertex d. Then looking at the second matrix, we can think of the path from d to c as a path from the intermediate vertex d to the destination vertex c. When the second matrix is not adj1, say it is adj4, then there is a path of length 1 to the intermediate node and a path of 4 from the intermediate node to the destination node.

Essentially, to find the number of paths of length k in a graph of N vertices, the N x N adjacency matrix is raised to the power of k. If k >= N and the resulting matrix has non-zero entries, these indicate cycles. This is not to say that non-zero entries in matrices where k<N do not sometimes represent cycles, but when k > N, cycles are guaranteed. Unfortunately, the vertex sequences are not recorded, leaving it to the programmer to verify results.

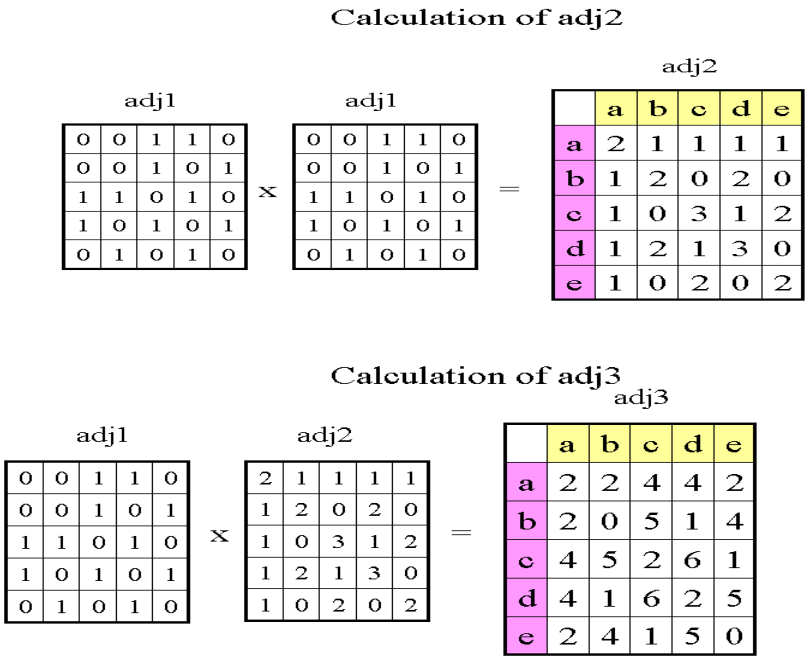


Figure 8.9 This figure shows the calculation of the adj2 and adj3 matrices. In general, adj2 shows the number of paths of length 2 from a source node to a destination node; adj3 indicates paths of length 3. One thing to note is that it is more common to treat the multiplication as a logical "and" process. When using this process, all resulting matrices are populated by 0's and 1's, instead of integer numbers. It can be said that by using the "and" technique the existence of paths is found, not the numbers of those paths.

Looking again at figure 8.9, notice that in adj3[a][d] there is a 4. Earlier we identified the paths from vertex a to vertex d of length 3 to be:

- a – c – a – d
- a – d – a - d
- a – d – c – d
- a – d – e – d

The matrix multiplication and the results of inspection agree (thank goodness). Lets see why.

adj3(a, d) = adj1(row a) x adj2(col d)

= (0*1) + (0*2) + (1*1) + (1*3) + (0*0)

= (1*1) + (1*3)

= 4

These paths that contribute to the 4 are the following -

Represented by the (1*1) from adj1[a][c] * adj{c][d] :

(a to c)$^{length = 1}$ then (c to d)$^{length = 2}$ = (a to c)$^{length = 1}$ then (c to a)$^{length = 1}$then (a to d)$^{length = 1}$

Plus the factor (1*3) from adj1[a][d] * adj{d][d] :

(a to d)$^{length = 1}$ then (d to d)$^{length = 2}$ = (a to d)$^{length = 1}$ then (d to a)$^{length = 1}$then (a to d)$^{length = 1}$

And

(a to d)$^{length = 1}$ then (d to d)$^{length = 2}$ = (a to d)$^{length = 1}$ then (d to c)$^{length = 1}$then (c to d)$^{length = 1}$

And

(a to d)$^{length = 1}$ then (d to d)$^{length = 2}$ = (a to d)$^{length = 1}$ then (d to e)$^{length = 1}$then (e to d)$^{length = 1}$

8.3.2 Dijkstra's Single Point Shortest Path Algorithm

Dijkstra's algorithm finds all the shortest paths and their associated costs from a source node in a graph to all other nodes in the graph. This algorithm has many practical uses, such as finding the best routes between computers in a network, or finding the best path to take from one location to another, or in laying out circuits on a complex board, etc. etc.

The algorithm is of order $O(n^2)$. It works by first finding all the direct routes available from the source node to the other nodes. Then it repeatedly selects the closest available node and uses it as an intermediate node between the source and each of the destination nodes. If the alternate route using the intermediate node has a lower cost than the existing route, the new lower cost route replaces the existing route, otherwise, the existing route is kept. The process is repeated until all destination nodes have been used as intermediate nodes.

In the implementation of the algorithm a cost matrix records the costs from the source to each of the destination nodes, and a path matrix is used so that the node sequence can be backed out for each of the corresponding shortest paths. Figure 8.10 below shows a directed graph with 4 vertices.

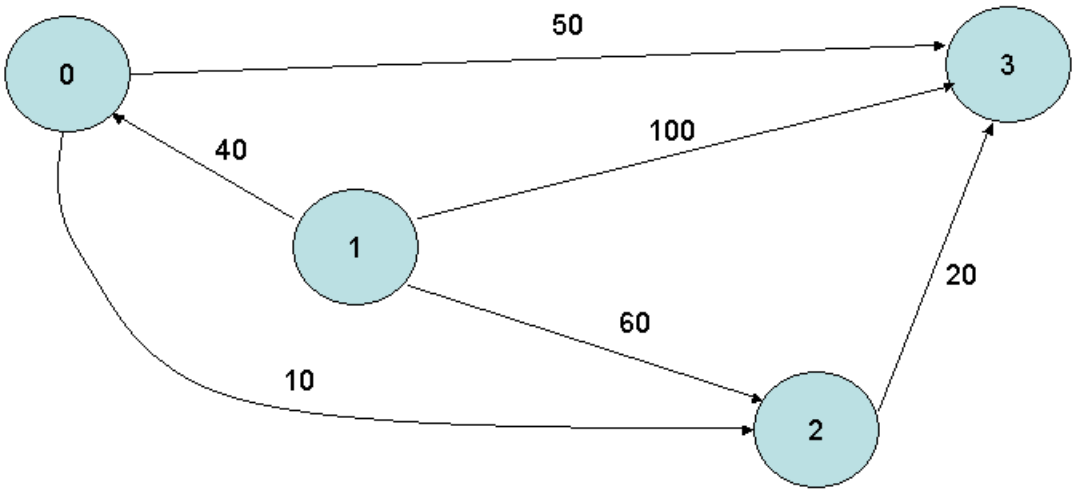


Figure 8.10 shows a directed graph with 4 vertices and several edges. The numbers next to the edges represent the cost associated with moving from one vertex to another. For example, if we are currently located at vertex 1 and we want to move to vertex 3, if we take a direct path, the cost is 100. It can be seen that there exist alternate paths that are lower in cost. The path from 1 to 0 to 3 has a cost of 90 (1 to 0 costs 40 and from vertex 0 to 3 costs 50. Summing the two results in a total cost of 90. Likewise, there is a path from vertex 1 to 2 to 3 that costs 80.

The initial cost and path tables for graph 8.10 are show below. For this example, vertex 1 will be the source, and we will be using Dijkstra's algorithm to find the shortest paths and their costs to each of the remaining vertices.

Cost Table: source = vertex 1			
Visited nodes	Destination nodes		
	0	2	3
{1}	40	60	100

Path Table: source = vertex 1			
Visited nodes	Destination nodes		
	0	2	3
{1}	1	1	1

Looking at the cost table, we can see that the closest node to node 1 is node 0, because the cost (distance) to node 0 is 40, which is less than the cost to node 2 (cost from 1 to 2 is 60), and node 3 (cost from 1 to 3 is 100). Since 0 is the closest we choose it as our intermediate node.

Now using 0 as the intermediate node we will check to see if we can reach the other nodes going through node 0. Looking at 2, we see that the current cost from 1 to 2 is 60, but using node 0 we can make a path 1 to 0 to 2 and its total cost is 50 (1 to 0 costs 40, and 0 to 2 costs 10, resulting in a cost of 50 for the path 1 to 0 to 2). Since the new cost, 50 is less than the existing cost, 60, we discard the old path in favor of the new lower cost path, 1 to 0 to 2. We follow the same process for the remaining nodes, checking to see if the the alternate through node 0 is of lower cost than the existing path. We reflect this by adding a new entry into the cost table, as follows below.

Cost Table: source = vertex 1			
Visited nodes	Destination nodes		
	0	2	3
{1}	40	60	100
{1, 0}	40	50	90

Now, we modify the path table to reflect the updates done in the cost table. Notice that the intermediate node is put into the set of visited nodes. The updated path table is below:

Path Table: source = vertex 1			
Visited nodes	Destination nodes		
	0	2	3
{1}	1	1	1
{1, 0}	1	0	0

How do we get a path out of the table? We back it out. For example, if we want to know how to get to node 3 we, find the 3 in the Destination nodes column and we see that to get to node 3, we get there from node 0 (we can see that node 0 is the most efficient path to 3 because the 0 is in bottom of node 3's Destination node column. Next, we need to see how to get to node 0, so in a similar fashion we go look in node 0's column of the Destination nodes section of the table. We can see that to get to node 0 we came from node 1, which is our source node. So, we backed out the path from the destination. To get to node 3, we come from node 0, to get node 0 we come from node 1, and node 1 is the source node, so the path is 1 to 0 to 3.

Continuing the process results in the following completed cost and path tables:

Cost Table: source = vertex 1			
Visited nodes	Destination nodes		
	0	2	3
{1}	40	60	100
{1, 0}	40	50	90
{1, 0, 2}	40	50	70
{1, 0, 2, 3}	-	-	-

Path Table: source = vertex 1			
Visited nodes	Destination nodes		
	0	2	3
{1}	1	1	1
{1, 0}	1	0	0
{1, 0, 2}	1	0	2
{1, 0, 2, 3}	1	0	2

It is always a good exercise to check the completed tables against the graph.

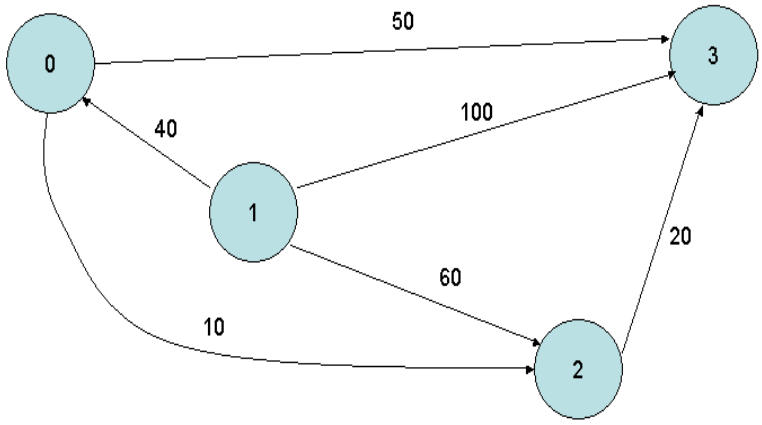


Figure and Tables 8.11 – The completed tables should sanity checked against the graph.

8.4 Exercises

1. Write 3 graph traversals, depth first recursive, depth first iterative, and breadth first iterative.
    a. The input of the program will be:
        i. The number of nodes in the gaph.
        ii. An adjacency matrix that represents a graph.
    b. Output will be the order in which each vertex was visited.
    c. Create an adjacency matrix from the graph in figure 8.4.
2. Write a program that will calculate the transitive closure of a graph.
    a. The input of the program will be:
        i. The number of nodes in the gaph.
        ii. An adjacency matrix that represents a graph.
    b. Use the adjacency matrix from figure 8.8 as input.
    c. Calculate the transitive closure up to length 5.
    d. Output should be each adjacency matrix; i.e. adj2, adj3, adj4, etc. You can use figure 8.9 to check you answers up to adj3.
3. Using the graph from figure 8.10, use Dijkstra's algorithm to find the shortest paths using vertex 0 as the source node.
4. Write a program that will read an adjacency matrix from an input file and then use Dijkstra's algorithm to compute length and vertex sequences of the shortest paths from a selected source node. Use the graph in figure 8.11 as a test graph.

Chapter 9. Hash Tables and Hashing

Hash tables and hashing is an interesting subject because it offers us the ability to quickly load and retrieve data in a way that is somewhat different from all methods studied up to this point. While many of our previous methods relied upon clever arrangements of the data using links from one datum to another, hashing uses a direct indexing scheme. It does so by placing the data into memory in a position that is determined by some characteristic of the data.

A hash table can be thought of as a large array in memory that is ideally only sparsely populated by data items. The location of each item is determined by a function called a hash function. The hash function generates an index into the array based on characteristics each of the datum.

For example, suppose we wanted to create a database of students based on their height. When we print out the database, we want the shorter people to be printed first and tallest people last. The data will be introduced in random order over time, so we cannot presort the data and load it sequentially into an array, and since we want to be able to load and retrieve data nearly instantaneously, a linked list, indexed linked list, or tree will be too slow. Table 9.1 below shows an example of our data.

Person	***Height***	
	feet	*inches*
Bobby	5	8
Jim	6	1
Anny	5	4
Zack	5	2
Milly	5	6
Jack	6	8
Kristy	4	10
Tiny	7	2
Moose	4	4
Pam	5	3
Marcello	6	0
Dave	5	11
Ghassan	5	7
Jenny	5	6
Kathryn	5	2

Table 9.1 Name/height data example data. Notice that some of the people are the same height.

Let our hash index be generated by the following equation:

```
index = person.feet * 100 + person.inches
```

It is common to call such an equation a "hash function". Table 9.2 below shows the indices for all our names:

Person	***Height***		***Hash Index***
	feet	*inches*	
Bobby	5	8	508
Jim	6	1	601
Anny	5	4	504
Zack	5	2	502
Milly	5	6	506
Jack	6	8	608
Kristy	4	10	410
Tiny	7	2	702
Moose	4	4	404
Pam	5	3	503
Marcello	6	0	600
Dave	5	11	511
Ghassan	5	7	507
Jenny	5	6	506
Kathryn	5	2	502

Table 9.2 This table shows the indices generated by our hash function. Notice that people with the same height share the same hash index.

Notice that the hash function generates the same index, 502, for Zack and Kathryn. This makes a lot of sense; they are the same height. But it creates a problem; the data for Zack cannot be stored at the same location as the data for Kathryn. Such a condition is called a collision, or a "hash clash". To handle collisions, a strategy is needed. A very simple strategy is the following:

- Generate hash key.
- Check to see if the position in the table pointed to by the hash key is un-occupied. If so, insert the data.
- Otherwise, sequentially search the table for the next available position.

In the example above, when the collision is detected by Kathryn, the next position, 503, would be checked. Unfortunately, Pam is at 503, and the next place is occupied by Anny, but 505 is empty, so Kathryn's data can be placed there. Table 9.3 shows the hash table in order of the indices.

Hash Table Index	***Person***	***Height***		***Generated Hash Index***
		feet	*inches*	
400				
401				
402				
403				
404	Moose	4	4	404
.				
.				
409				
410	Kristy	4	10	410
.				
.				
498				
499				
500				
501				
502	Zack	5	2	502
503	Pam	5	3	503
504	Anny	5	4	504
505	Kathryn	5	2	502
506	Milly	5	6	506
507	Ghassan	5	7	507
508	Bobby	5	8	508
509	Jenny	5	6	506
510				
511	Dave	5	11	511
512				
513				
.				
.				
600	Marcello	6	0	600
601	Jim	6	1	601
.				
.				
608	Jack	6	8	608
.				
.				
699				
700				
701				
702	Tiny	7	2	702

Table 9.2 This table shows indices vs. actual placement in the hash table. The yellow shaded cells are where data was moved to the next available location due to a clash. The blue cells are places are where clashes occurred.

A quick analysis of the hash function and the resulting table (Table 9.2) will reveal that this function will be collision intense. Why? Because it will try to place all the data in the first 12 entries of each group of 100. That is, everyone who is between five foot 0 inches and five foot 11 inches will generate hash indices between 500 and 511, crowding this area, and wasting 512 to 599. Another major drawback of this function is that, the first 300 to 400 entries of the table will likely be wasted. Again, why? Because the likely hood of an adult being smaller than 3 feet tall is very low; absolutely no offense intended to small people. A better hash function is shown below:

```
index = ((person.feet * 100) + (person.inches * 8)) - 300
```

By multiplying the inches by 8, this function will distribute data in height order throughout the table, leave room for collisions at each inch range. By subtracting 300, the first 300 positions of the table are not wasted. For instance, from the table above, Zack and Kathryn, both of height 5'2". The index generated for their height is 216. When placed into the table, Zack will occupy the 216^{th} position, and Kathryn will be placed into the 217^{th} position, due to the collision with Zack. The collision is not avoided, but it is managed better because the function will group the first 8 people of that height together, instead of mixing them in with people of other heights as before.

9.1 Summary of Definitions and Concepts

Hashing is an order of n, O(1), method of arranging of data in memory. It relies on a hash function, collision strategy and a table in memory to locate data items. Because the hash function calculates the hash index directly, it is ideal for speed intense applications. To place or retrieve a single item, it is O(1); to do a data set, it is O(n).

Hash Function

hashKey = function of data characteristics

Collisions and Collision Management

When two unique pieces of data generate identical hash indices this is called a collision, or a hash clash. Various schemes are used to handle collisions, the simplest being to search the memory for the next available space. When collisions become frequent, the data can be re-hashed into a larger table. When doing this, the hash function will have to take the new table size into account.

Effective Use of Memory

In general, hashing does not make effective use of memory. This is the trade-off for speed, i.e. O(1) time complexity. Nevertheless, a good hash function will spread the data evenly throughout the available memory, and leave room for collision handling.

Criteria for a Good Hash Function

1. Even distribution of data in hash table.
2. Leave room for collisions so that data from collision can be as close to it's index as possible.
3. For non-password applications, logical ordering of data in table. By doing this, when the table is printed out, it is in near sorted order.

9.2 Basic methods

The following basic methods can be used as a starting point for implementing a simple hashing system. In the exercises, there is a considerable piece of code that can also be used as a guide.

- init() - mark all positions of the table as empty.
- hashMe() - calculate the hash index based on properties of the data.
- getNextOpenPosition() - in the event of a collision, search table for the next available piece of memory.
- showTable() - prints out occupied positions in table.
- rehash() - when collisions become frequent, allocate a new table with a larger size and then rehash the data from the current table into the new table using a hash function that is modified to take the new table's size into account.

9.3 Exercises

1) You are to create hash table and an appropriate hash function for use in storage and retrieval of character data. You will vary the size of the table and count the number of collisions that occurs with each table.

    a) Create a hash table. Use your table to "sort" a list of names. A list of test names is provided below. Notice that they are all using lower case letters. The hash index can be calculated in a similar manner that the "namecode" was calculated in the index linked list exercise from an earlier chapter.

```
joe
bob
harry
mary
brian
tom
jerry
bullwinkle
pam
ellis
dale
bill
barrack
george
gertrude
zack
zeus
apollo
gemini
greg
larry
meriam
webster
thomas
stewart
dianna
theresa
billyjoe
carl
karl
charles
karla
donna
tena
kerry
howard
johnson
ulyssess
paul
peter
issaac
marvin
dudz
chuck
ellie
anny
judy
```

```
matt
ross
dan
robert
kim
eric
junkun
ghassan
cris
raymond
avery
roy
halley
mitzee
ziggy
rocky
twirly
max
huey
dewy
hongkongfooey
clarence
lala
sammy
fred
francis
```

b) Your hash function can be one that you develop, or you can use the simple 3 letter function outlined below:

Let the letters of the alphabet be resolved to the numbers 0 through 25. This can be accomplished by subtracting the letter 'a' from each letter. Equation 1 shows this relationship.

Equation 1:
letterVal = letter - 'a'

The idea of equation 1 can be incorporated into a function of the following form:

int getLetVal(char ch)

Then a hash value can be generated using an equation similar to equation 2 below:

Equation 2:
nameHash = getLetVal(name[0])*26^2 + getLetVal(name[1])*26 +
getLetVal(name[0]);

c) You will run your program with three different sizes of hash tables:
   i) 200
   ii) 400
   iii) 700

d) You will have to adjust your hashing function based on the size of the hash table, meaning you will need to divide the nameHash value by a denominator that maps

the names into the table. For example, if using the hash table with size 200, a very rough estimate of the denominator is 100. Why? Because the max value of the hash function using a name that starts with "zzz", like "zzztop" (a very sleepy version of the band from Texas, "ZZTop"), is around 17, 500. That will not fit in a hash table of size 200, but if you divide it by 100, it maps "zzztop" to hash index 175. Look to the discussion in this chapter for more guidance.

e) You will need to have a collision management scheme.
   i) Note, record the number of collisions that occur for each different size. If it is done correctly, each time the table gets bigger, the number of collisions should decrease.
f) Print out the contents of the table for each different size of hash table. Look closely at the printout. Which size of table did the best job of arranging the names in alphabetical ordering in the memory.
g) Below is some code to help in program development.

```
/**
 * fileIn.java
 *
 *
 * pm
 * @version 1.00 2009/9/16
 */

import java.util.Scanner;

import java.io.*;

import java.lang.Math;


public class hashMe {
      String fname;
      String hashTable[];
      int tabLen;
      int numSmashes;

    public hashMe(int hashTableLen) {
      int j;

      /* Initialize variables. */
      tabLen = hashTableLen;
      numSmashes = 0;

      /* Initialize hash table. */
      hashTable = new String[tabLen];

      for(j=0;j<tabLen;j++) {
            hashTable[j] = "#";
      }

      getFileName();
      readFileContents();
      showHashTable();
    }

    public void readFileContents()
    {
```

```
  boolean looping;
  DataInputStream in;
  String line;
  int j, len, hashIndex, hInc, spot;

  /* Read input from file and process. */
    try {
        in = new DataInputStream(new FileInputStream(fname));

        looping = true;
        while(looping) {
            /* Get a line of input from the file. */
            if (null == (line = in.readLine())) {
                looping = false;
                /* Close and free up system resource. */
                in.close();
            }
            else {
              hashIndex = hashFun(line);
            }
        } /* End while. */

    } /* End try. */

    catch(IOException e) {
        System.out.println("Error " + e);
    } /* End catch. */
}



void showHashTable()
{
  int j;

  System.out.println("Number of Hash Clashes = ");


  for(j=0;j<tabLen;j++) {

  }
}



public int hashFun(String name)
{
  int hashVal;


  return hashVal;
}



public void getFileName()
{
  Scanner in = new Scanner(System.in);

  System.out.println("Enter file name please.");
  fname = in.nextLine();
  System.out.println("You entered "+fname);
}
```

```
    public static void main(String[] args)
      {
            System.out.println("Hello TV land!");
            hashMe h = new hashMe(200);
            System.out.println("Bye-bye!");
      }
}
```

Chapter 10. Heaps

A heap is a nearly complete binary tree in which the root contains the tree's largest number. Each sub-tree retains this property, making it so that each child node is always of lower value than its parent. This is referred to as the heap property. While the children of each tree node are less than their parents, there is no requirement that the left child be greater than the right child or vice versa. Figure 10.1 shows various heaps.

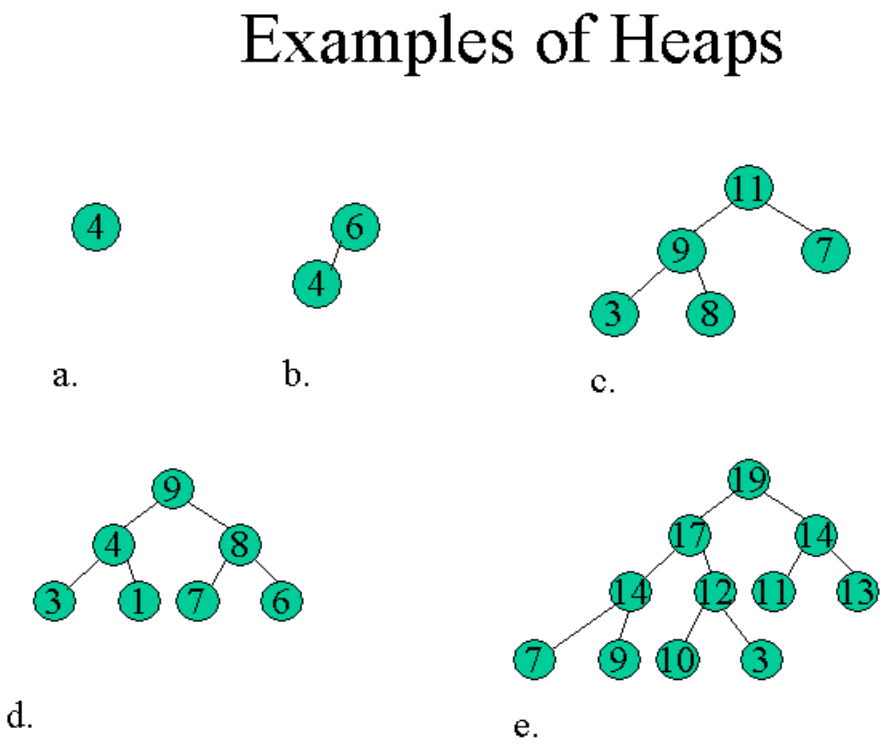


Figure 10.1 This figure shows several examples of heaps. Looking at 10.1a., even though there is only one node, it does retain the heap property. 10.1b. Shows a heap with 2 nodes, it is clear that the root is greater than the child. 10.1c shows a heap with 5 nodes. Notice that the root is the largest number and that the left sub-tree (rooted by the 9) also retains the heap property. Looking at 10.1d and 10.1e, it can be seen that the heap property is retained in each sub-tree, and that it is fine if the left child is greater than the right, and vice-versa.

Suppose we wanted to build a heap from the following list of numbers:

numbers = {8, 3, 9, 4, 7, 11, 6}

We would start with an empty tree, then add 8 as the root; next the 3 would be added as 8's left child, then the 9 would become the 9's right child. At this point the tree would lose the heap property because a child not has greater value than its parent. To reinstate the heap property, the right child (the 9) would be exchanged with the root. So, each time a node is added if it is greater than its parent, an exchange between child and parent

occurs. When this exchange is made, the heap property is again checked, and if it does not hold, another exchange is made between the child and its new parent. This process is repeated until the heap property exists again. Said another way, when a node is added that violates the heap property, it is bubbled up the tree by exchanges between child and parent until the heap property is re-established. Figure 10.2 below shows the construction process of the heap for the list of numbers in the previous example.

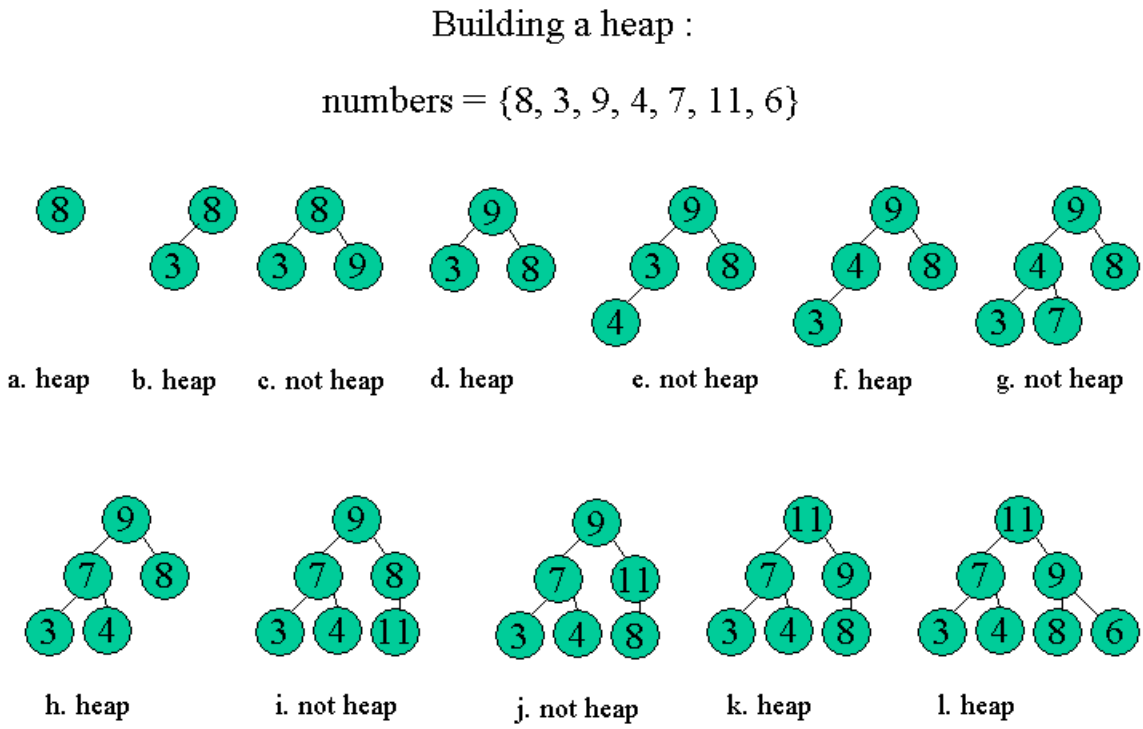


Figure 10.2 This figure shows the step-by-step construction of a heap from a list of numbers. Each time a number is added to the heap, the heap property is maintained by exchanging child and parent nodes until the heap property is re-established. Notice in figure c, the heap property is violated when the 9 is added as a child of the 8. In figure d, the 9 and 9 are exchanged, thus re-establishing the heap. In figure i, the 11 is added as a left child of the 8. It is exchanged with the 8 in figure j, but because it is greater than its parent node, the 9, the heap property does not exist. In figure k, the 11 and 9 are exchanged, thus re-establishing the heap.

10.1 Array Implementation of the Heap

It is common for a heap to be implemented using a one-dimensional array. By using an array, a relationship between child and parent indices can be established. The relationship is described by the equations below:

- parentIndex = (int) (childIndex/2)
- leftChildIndex = parentIndex * 2
- rightChildIndex = (parentIndex * 2) + 1
- rightChildIndex = leftChildIndex + 1

Figure 10.3 below shows the value of the indices for a heap with 16 nodes.

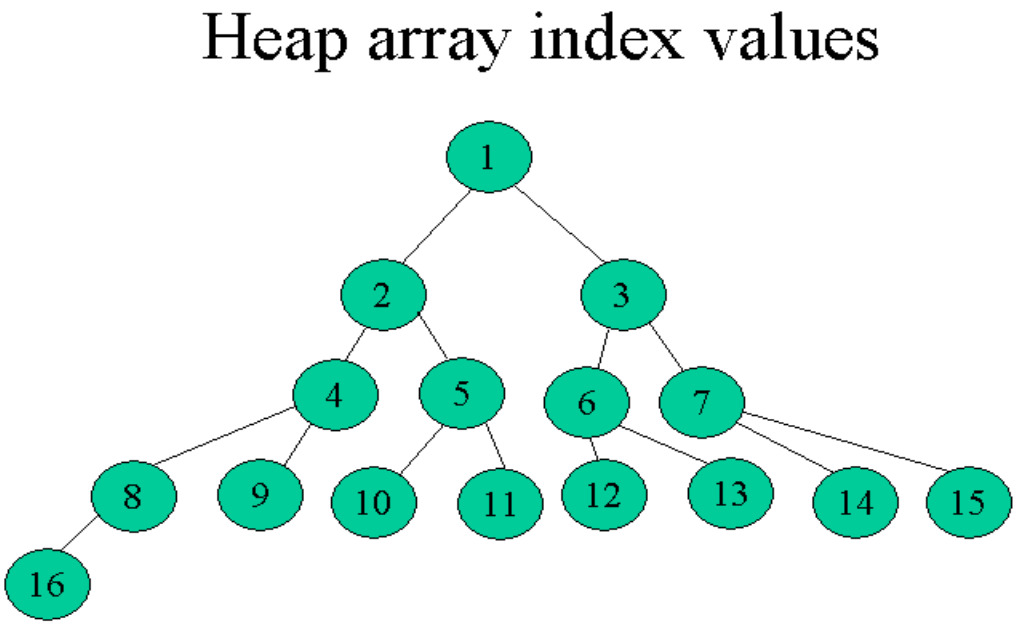


Figure 10.3 This figure shows the array index values for the nodes of a heap. Notice that the root node is placed in array index 1. Also notice that each parent node's value is half that of its children (using integer division).

10.2 Basic Methods

The basic methods for building a heap are:

- buildHeap
- heapify
- heapSort
- reHeapify

The buildHeap() function places numbers in an array one at a time and calls heapify() to ensure that the array retains the heap property. The heapify() function is a recursive routine that operates by "bubbling" numbers placed in the leave positions of the heap upwards until their parent node is greater than them.

The heapSort() function operates in a similar but opposite manner. It removes the root of the heap; remember the root of a heap is the largest number in the heap, and places it the last position of the sorted array, and then calls reheapify() to restore the heap property to the remaining numbers. Once the rehepify() function is completed the process repeats, taking the root and placing it the next to last position of the array. When the heap is empty, the array contains a sorted list numbers.

The pseudo-code for the various routines is presented below:

10.21 buildHeap()

```
buildHeap(list of numbers: list)
{

      integer j, num;

      integer array tree[];


      j=1;
      while(getNumber(list)){
            /* Put number into the array. */
            tree[j] = num;
            heapify(j, tree);
            j = j+1;
      }

}
```

10.22 heapify()

```
void heapify(integer leaf, integer array tree[])
{
      integer parent;

      if (leaf > 1) {
            parent = leaf/2;

            if (tree[leaf] > tree[parent]) {
                  exchange(leaf, parent, tree);
                  heapify(parent, tree);
            }
      }
}
```

10.23 heapSort()

```
void heapSort(integer len, integer array tree[])
{
      integer j, last;
      integer array sArray[];

      last = len-1;

      /* Do the heap sort. */
      for(j=1;j<len;j++) {
            exchange(1, last, tree);
            sArray[last] = tree[last];
            last = last - 1;
            reheapify(1, last, tree);
      }
}
```

10.24 reheapify()

```
void reheapify(integer j, integer last, integer array tree[])
{
      integer lChild, rChild;
      integer k;
```

```
        lChild = 2*j;
        rChild = 2*j + 1;

        if ((lChild < last)&&(rChild <= last)) {
              if ((tree[j] < tree[lChild]) || (tree[j] < tree[rChild])) {
                    k = rChild;
                    if (tree[lChild] > tree[rChild]) {
                          k = lChild;
                    }
                    exchange(j, k, tree);
                    reheapify(k, last, tree);
              }
        }
        else if (lChild <= last) {
              if (tree[j] < tree[lChild]) {
                    exchange(j, lChild, tree);
              }
        }
  }
```

Chapter 11. Sorting

Sorting and searching are fundamental exercises in computing and an important part of data storage and retrieval. In this chapter we discuss various techniques of sorting, starting with basic exchange/brute force sorts of $O(n^2)$, then moving on to the more efficient sorts, including quick sort, merge sort, and radix sort. In the following chapter searching is discussed.

Sorting in its most basic sense is the act of arranging data so that each data item is in its proper position in relation to the other data in the set. Depending on requirements, sorts can be in ascending or descending order. Sorted data is far easier to search, in fact the most efficient search of unsorted data requires the inspection of each data item until the item of interest is found. On the other hand, when searching sorted data, the data's arrangement can be used to develop efficiency; sorted data can be searched in $O(\log_2 n)$.

The basic sorts are easy to implement, and use only the memory that holds the data. Quick sort and merge sort still work by using only the memory that holds the data, but are more complex to implement. Although they use differing processes, they are both $O(n\log_2 n)$. Radix sort differs from them all in that it uses characteristics of the data, namely the value of the individual digits, to generate efficiency. It is O(m * n), where m is the number of digits in the data, and n is the size of the data set.

11.1 Exchange/Brute Force Sorts – $O(n^2)$

Exchange sorts work by comparing the values of the data against one another and exchanging their positions until the sort is complete. Another definitive feature of them is their nested loop structure. In this section Bubble Sort and Selection Sort are discussed. There other similar sorts, but these two are representative of this family of sorts. The pseudo code for Bubble Sort is shown below.

11.1.1 Bubble Sort

```
Void bubbleMe(integer array numbers[])
{
      boolean swapping;
      integer j;

      swapping = true;

      while(swapping) {
            swapping = false;

            for(j=1;j< lengthOf(numbers[]), j++) {
                  if (numbers[j-1] > numbers[j]) {
                        swapping = true;
                        exchange(j-1, j, numbers[]);
                  }
            } /* End for. */
      } /* End while. */
}
```

Notice that bubble sort acts by comparing and swapping adjacent numbers only. Any time there is a comparison that evaluates to true, a swap takes place. Since adjacent values are exchanged, smaller values migrate or "bubble" towards the front of the array, while larger values work their way to the rear of the list. Compared to other exchange sorts, bubble sort is heavy on exchanges, but fairly efficient on compares. When looking at total overall efficiency, the question becomes which operation takes more CPU time, a comparison or a swap?

Looking at the loop structure, it is clear that the inner loop always executes n – 1 times. On the other hand, the outer loop executes some fraction of n times; determined by the inner loop. The swapping variable is set to false before the inner loop starts, once a swap is made (because some of the numbers in the list are out of order), swapping is set to true. If swapping is never set to true, then no numbers are out of order, and the sort is complete. Because the outer loop will usually executes a significant fraction of n times, the efficiency of the sort is $O(n^2)$.

11.1.2 Selection Sort

Selection sort improves on bubble sorts swapping strategy by finding the smallest number in the list and moving it to front of the list each iteration its outer loop. Notice in the pseudo-code below that the variables "minIndex" and "min" are used for this purpose.

```
Void selectMe(integer array numbers[])
{
      integer j, k, min, minIndex;

      for(j=0; j<lengthOf(numbers[]), j++) { {
            minIndex = j;
            min = numbers[j];

            for(k=j+1;k< lengthOf(numbers[]), k++) {
                  if (numbers[k] < min) {
                        minIndex = k;
                        min = numbers[k];
                  }
            } /* End for. */
            exchange(j, minIndex, numbers[]);
      } /* End while. */
}
```

Selection sort has a twin nested loop structure, making it $O(n^2)$. Looking at the algorithm, it can be seen that there is one swap for each iteration of the outer loop. So while there are in the neighbor hood of n^2 comparisons (as shown in section 5.5.2 there are (n*(n+1)/2) comparisons made), there are only n swaps. Selection sort has far less swaps than bubble sort, but more comparisons (data set arrangement dependent).

11.2 The More Efficient Sorts – $O(n*\log_2 n)$, $O(n*\log_2 n)$ on average, and $O(m*n)$

While the basic sorts are easy to implement, and use only the memory that holds the data, they are very inefficient compared to more algorithmically complex sorts such as tree sorting, Quick Sort, and Merge Sort. Tree and heap sorts work by creating a tree, or heap and exploiting the arrangement so that each comparison made reduces the sort time/number of numbers to be examined by roughly half. Other sorts work by subdividing the data set and operating on the smaller parts. Although Quick Sort and Merge Sort divide their data sets using different methods, they both attempt to break/rearrange/rebuild by halving the data sets. As discussed earlier, this type of operation falls into $\log_2 n$ time order, and since each set has some factor of n items in it the total time complexity of the sort is $O(n*\log_2 n)$.

Radix sort works on an entirely different principle. It exploits properties of the digits of the data in order to create a very efficient sort. Sometimes radix sorts are called bucket sorts because they create sets of bins in which the data items are placed based on the current digit of the data being examined. The bins are loaded and unloaded m times where m is the number of digits in the data items, thus the time complexity of the sort is $O(m*n)$.

Table 11.1 below shows the approximate number comparisons needed for several different types of sorts, assuming a 4 digit data item.

Data set Size	Exchange Sorts $O(n^2)$ Bubble, Insertion, Selection, etc.	Efficient Sorts $O(n*\log_2 n)$ Merge, Quick, Tree, Heap	Radix Sort $O(m*n)$
10	100	34	40
100	10000	665	400
1000	1000000	9966	4000
10000	100000000	132878	40000

Table 11.1 shows the approximate number of comparisons needed for sorts of $O(n^2)$, $O(n*\log_2 n)$, and $O(m*n)$. Notice that with small data sets, the number of comparisons is relatively close, but as the data set size grows, the more efficient sorts and the radix sort comparison numbers are markedly better than those of the exchange sorts (brute force sorts).

From the table it can be seen that with small data sets, the number of comparisons among the different types of sorts is relatively close, but as the data set size grows, the more efficient sorts and the radix sort comparison numbers are markedly better than those of the exchange sorts (brute force sorts).

One question that may be asked could be “Why bubble sorts and like used when some of the other sorts are so much more efficient?”. The answer to this question is that the

efficiency comes at a price, and that price is program size and complexity. If a sort does not have to be quick or will not be used often and data set size is relatively manageable, then a simple brute force sort is sufficient. On the other hand, if the sort is called often and/or the data set size manageable to large, then the more efficient sorts are worth the effort.

11.2.1 Tree Sorts

To sort a data set with a tree sort, the process is as follows:

- Build a binary tree from the data set.
- Do an inorder traversal of the tree. Unless the data set is relatively small, the inorder traversal should be iterative in order to avoid inefficiencies encountered by stacking due to recursion.

We refer the reader to chapter 7 since it covers tree construction and traversal in detail.

11.2.2 Merge Sort

Merge Sort is an effective method of sorting whose time complexity is $O(n*\log_2 n)$. It works by first subdividing the array into pairs of numbers and sorting those. Then it merges adjacent pairs into sets of size 4, and then it merges the sets of 4 into sets of 8, and so forth. The example below illustrates the method:

Here is a set of unsorted numbers:

(25, 57, 48, 37, 12, 92, 86, 33, 0, 7, 55, 32, 19, 17, 45, 99)

Now, pair the numbers up:

(25, 57), (48, 37), (12, 92), (86, 33), (0, 7), (55, 32), (19, 17), (45, 99)

Next, make sure each pair of number is in the correct low to high order:

(25, 57), (37, 48), (12, 92), (33, 86), (0, 7), (32, 55), (17, 19), (45, 99)

Now, merge adjacent pairs into sets of size 4. To do this compare the first number of each set, and write down the lowest number. Once a number is written down, move to the next number in the set. If there are no more numbers in the set, write down the rest of the other set.

(25, 37, 48, 57), (12, 33, 86, 92), (0, 7, 32, 55), (17, 19, 45, 99)

Now merge the adjacent sets of 4 into sets of 8, using the rule above.

(12, 25, 33, 37, 48, 57, 86, 92), (0, 7, 17, 19, 32, 45, 55, 99)

And finally, merge the sets of 8 into the final set of 16.

(0, 7, 12, 17, 19, 25, 32, 33, 37, 45, 48, 55, 57, 86, 92, 99)

Below we present a very C-like pseudo code version of the merge sort. Notice that the majority of the work is done in the rearrange subroutine. The routine mergeSort's main function is manage the creation of various sized sets. The mergeMe routine carries out the actual merging of the sets. This version of merge sort is somewhat inefficient because it uses a scratch array. It is used for illustrative purposes and should be modified if the routine is going to be used for other purposes.

```
int scratch[16];
int array[16] = {25, 57, 48, 37, 12, 92, 86, 33, 0, 7, 55, 32, 19, 17,
45, 99};



void main(void)
{
      print("Hello TV Land! \n");

      mergeSort(16);
      showArray(16);
}



void mergeSort(int N)
{
      int exp;
      int mSize;
      int j;
      int a0, b0;

      exp = 0;
      mSize = (int)pow(2, exp);

      while(mSize < N) {

            for(j=0;j<N;j=j+(mSize*2)) {
                  a0 = j;
                  b0 = a0 + mSize;
                  mergeMe(a0, mSize, b0, mSize);
                  copyScratch2Array(a0, mSize*2);
            }

            exp = exp+1;
            mSize = (int)pow(2, exp);
      }
}



int mergeMe(int a0, int aLen, int b0, int bLen)
{
      int a, b, c;
      int mSize;


      if ((aLen == 1)&&(bLen == 1)) {
            if (array[a0] < array[b0]) {
```

```
                scratch[0] = array[a0];
                scratch[1] = array[b0];
        }
        else {
                scratch[0] = array[b0];
                scratch[1] = array[a0];
        }
        c = 2;
    }

    else {
        mSize = aLen + bLen;

        a = a0;
        b = b0;
        for(c=0;c<mSize;c++) {
            if ((a - a0) == aLen) {
                scratch[c] = array[b];
                b = b+1;
            }
            else if ((b - b0) == bLen) {
                scratch[c] = array[a];
                a = a+1;
            }
            else if (array[a] < array[b]) {
                scratch[c] = array[a];
                a = a+1;
            }
            else {
                scratch[c] = array[b];
                b = b+1;
            }
        }
    }

    return c;
}


void copyScratch2Array(int a0, int len)
{
    int j;

    for(j=0;j<len;j++) {
        array[a0+j] = scratch[j];
    }
}



void showScratch(int len)
{
    int j;

    for(j=0;j<len;j++) {
        print("%d ", scratch[j]);
    }
    printf("\n");
}
```

```
void showArray(int len)
{
	int j;

	for(j=0;j<len;j++) {
		print("%d ", array[j]);
	}
	printf("\n");
}
```

11.2.3 Quick Sort

QuickSort is a popular method of sorting whose time complexity is on average $O(n*\log_2 n)$. It is similar to merge sort in that works by subdividing the array into two pieces during each iteration of algorithm. However, it is not required, or normal, for the two pieces to be of equal size, but in an ideal world they would be very close to equal. While this part of the algorithm resembles that of merge sort, the rest is quite different. Quick sort uses the following ideas to do its work:

- Upon the start of each iteration of the sort, a "pivot value" is chosen.
- The goal of each iteration is twofold:
  - Put the pivot value in its correct place in the array.
  - Guarantee that every number to the left of the pivot value is less than the pivot value, and every number to the right of the pivot value is equal to or greater than the pivot value.

Below we present a very C-like pseudo code version of the quick sort. Notice that the majority of the work is done in the rearrange subroutine.

```
int list[10] = {25, 57, 48, 37, 12, 92, 86, 33, 99, 38};


void main(void)
{
	print("Hello TV Land! \n");
	showList(0, 10);
	quickSort(0, 10);
	showList(0, 10);
}


void showList(int start, int end)
{
	int j;

	print("\nlist...\n");
	for(j=start;j<=end;j++) {
		printf("%d  ", list[j]);
	}
	print("\n");
}
```

```
void quickSort(int lb, int ub)
{
	int pivot;

	if (lb < ub) {
		pivot = rearrange(lb, ub);
		quickSort(lb, pivot-1);
		quickSort(pivot+1, ub);
	}
}


int rearrange(int lb, int ub)
{
	int up, down;
	int pivot, pIndex;

	pivot = list[lb];

	up = ub;
	down = lb;

	while(up != down) {

		while(list[up] > pivot) {
			up = up - 1;
		}

		if (up != down) {
			exchange(up, down);
		}


		while(list[down] < pivot) {
			down = down + 1;
		}
		if (up != down) {
			exchange(up, down);
		}

	}

	pIndex = up;

	return pIndex;
}


void exchange(int j, int k)
{
	int temp;

	temp = list[j];
	list[j] = list[k];
	list[k] = temp;
}
```

11.2.4 Radix Sort

The radix sort works in a fundamentally different way than the sorts previously discussed. The following example illustrates the algorithm.

Suppose we want to sort the following numbers:

numbers = {390, 101, 431, 285, 280, 161, 385, 257, 482, 411}

Since the numbers all have 3 digits we make 3 passes through the list, starting with the least significant digit. During each pass we put the number into a bin/bucket that matches the digit. So, for each pass we need a set of buckets, one for each digit. Initially the buckets are empty, then, one by one we put the numbers into the appropriate bucket. The table 11.2 below shows the buckets 0 – 9 with the loaded with the numbers after the first pass.

Numbers = {390, 101, 431, 285, 280, 161, 385, 257, 482, 411}

The procedure is straight forward. Starting with the first number, 390, each number is placed into the bucket matching the digit being analyzed for the current pass. For this pass, we are looking at the least significant digit, the ones digit. So, the 390 is put into the 0 bucket, the 101 goes into the 1's bucket, the 431 goes into the 1's bucket, the 285 goes into the 5's bucket, and so forth.

Bucket	***Numbers in buckets (first pass)***			
0	390	280		
1	101	431	161	411
2	482			
3				
4				
5	285	385		
6				
7	257			
8				
9				

Table 11.2 shows the buckets/bins of the radix sort after the first pass. In the first pass the numbers are placed in the bucket that matches the least significant digit of the data. For example the 257 is placed in the 7's bucket.

Once the numbers are in all the buckets, we can rewrite the list by emptying the buckets in order starting at the 0 bucket. The list after the first pass is shown below.

Numbers = {390, 280, 101, 431, 161, 411, 482, 285, 385, 257}

The next pass keys on the second most least significant digit, the tens digit. So just like the previous pass the numbers are placed in the bins based on the digit, except this time the tens digit is used, and also, note that numbers are taken from the number list that was formed by unloading the numbers from the buckets after the first pass.

$Numbers^{after\ 1st\ pass}$ = {390, 280, 101, 431, 161, 411, 482, 285, 385, 257}

Bucket	***Numbers in buckets (second pass)***			
0	101			
1	411			
2				
3	431			
4				
5	257			
6	161			
7				
8	280	482	285	
9	390			

Table 11.3 shows the buckets/bins of the radix sort after the second pass. In the second pass the numbers are placed in the bucket that matches the second most least significant digit of the data (the tens digit). For example the 257 is placed in the 5's bucket.

$Numbers^{after\ 2nd\ pass}$ = {101, 411, 431, 257, 161, 280, 482, 285, 390}

Looking at the new number list, at first glance the numbers do not look sorted, and in a way they are not. But looking a bit closer, it can be seen that the first two digits are sorted completely. If we write out the first two digits of the numbers as shown below, they are sorted in low to high order.

First two digits of $Numbers^{after\ 2nd\ pass}$ = {01, 11, 31, 57, 61, 80, 82, 85, 90}

To finish the process the Numbers from the list after the second pass are put into the bins based on the most significant digit.

$Numbers^{after\ 2nd\ pass}$ = {101, 411, 431, 257, 161, 280, 482, 285, 390}

Bucket	***Numbers in buckets (second pass)***			
0				
1	101	161		
2	257	280	285	
3	290			
4	411	431	482	
5				
6				
7				
8				
9				

Now the only thing left to do is to unload the numbers from the table.

$Numbers^{after\ 3rd\ pass}$ = {101, 161, 257, 280, 285, 290, 411, 431, 482}

The numbers are now sorted. It took 3 passes on the number list, thus the run time complexity is O(number of digits * size of data set) or O(m*n).

To implement the radix sort, the bins can be made from arrays. Arrays are good for a proof of concept but will cause problems because they will be either too big or too small. The ideal method would some type of simple to use dynamically allocated technique, such as a linked list. Linked lists such as those covered in the earlier chapters would be ideal, or some languages such as JAVA have some built in constructs that would be very useful. Below we present a very general algorithm for the radix sort.

```
void radix(array numberList, int len)
{
      int bins[10][];

      int j, k;

      for(j=leastMostSignificantDigit to mostSignificantDigit) {
            Initialize_Bins();
            for(k=0;k<len;k++) {
                  digit = numberList[k].getSpecificDigit(j);
                  binLength = bins[digit].length;
                  bins[digit].numbers[binLength] = numberList[k];
                  bins[digit].length++;
            }
            numberList = UnloadBins2List();
      }

      writeList(numberList);
}
```

11.3 Exercises

For the following exercises, unless otherwise specified, generate a list of 1000 random numbers and store them in a file. The numbers can generated using the technique described in exercise 4 at the end of chapter 7.

1. Implement the bubble sort and selection sort routines.
    a. In each routine, put in code that will count the number of comparisons, and the number of exchanges made.
    b. For testing, run each program with the list of 1000 numbers from a file as described at the beginning of this section and in exercise 4 at the end of chapter 7.
    c. Which of these sorts is more efficient?
2. Implement the mergeSort based on the psuedo-code/code provided in the body of the chapter. Test it on a list of 1000 random numbers.
    a. In your code, count the number of comparisons made.
    b. How does the number of comparisons your program counts compare to that estimated by mergeSorts running time complexity $O(n\log_2 n)$?
3. Implement the quickSort based on the psuedo-code/code provided in the body of the chapter. Test it on a list of 1000 random numbers.
    a. In your code, count the number of comparisons made.
    b. How does the number of comparisons your program counts compare to that estimated by quickSorts running time complexity $O(n\log_2 n)$?
    c. If you implemented mergeSort, which is more efficient mergeSort or quickSort?
    d. See if you can improve the quickSort by breaking away to a bubble sort when the set size falls below a some certain size. Pick a size of 10 to start off with, and vary the size between runs to come up with an optimal size.
4. Implement the radix sort algorithm and use it to sort the file of 1000 random numbers.
    a. Count the number of steps/comparisons. It really is not a comparison, it is the placing of a number in the appropriate bin.
    b. Use an array of linked lists, 1 linked list for each digit, as buckets.
    c. After each pass, write out the buckets so that we can see the progression of the sort.
5. Now implement the radix sort and use it to sort the list of names from chapter 9, exercise 1. Use the following items as guidlines.
    a. Linked lists used for the buckets
    b. Sort on the fist 4 letters of the name
    c. Count the number of steps/comparisons
    d. After each pass, write out the buckets so that we can see the progression of the sort.

Chapter 12. Searching

Searching an unordered list of data requires that each item be sequentially observed. In the best case, the item being looked for will be found on the first comparison, at the worst it will the last to be found. On average, the search will take n/2 comparisons, making it O(n).

Average search time = (best search time + worst search time)/2

Average search time = $(1 + n)/2 \approx n/2$ which is O(n)

12.1 Basic Search Algorithms

In this section we present two algorithms, one for searching unordered data, the other, a binary search for searching data that has been sorted.

12.1.1 Simple Sequential Search of Unordered Data

The simple sequential search works by starting at the beginning of the list of numbers and examining each number in sequence, comparing it against the number to be found. It can be implemented using a for-loop or a while-loop with the while-loop being on average more efficient because it does not continue checking the list once the number has been found. Below the search with the while-loop is presented. It returns the index of the array that the target number occupies, or if the number is not in the array, it returns -1. As previously stated the algorithm is of O(n).

```
int findIt(int numberList[], int len, int findMe)
{
	int j;

	j = 0;

	found = false;

	while((!found)and(j < len)) {
		if (numberList[j] == findMe) {
			found = true;
		}
		else {
			j = j+1;
		}
	}

	if (!found) {
		j = -1;
	}

	return j;
}
```

A useful exercise for the reader would be to trace through the above pseudo-code to see how the algorithm works.

12.1.2 Binary Search of Ordered Data

The binary search works almost just like the old game of guess the number. It works by finding the value of the number in the middle of the array. If that number is the same as the number to be found, the search ends. But if the number to found is greater than the mid point, the mid point becomes the new lower boundary of the search. If the number to be found is less than the number at the mid point, the mid point becomes the new upper boundary of the search. This process is repeated until the number is found. The algorithm is presented below, but note this algorithm assumes that the number to be found exists in the list of numbers. It is not overly difficult to modify the algorithm to handle cases in which the number to found does not exist in the list.

```
int findMeBinary(int array[], int len, int num2Find)
{
      int low, high;

      low = 0;
      high = len - 1;

      found = false;

      while(!found) {
            mid = (low + high)/2;
            if (number2Find == array[mid]) {
                  print("We found it at index ", mid);
                  found = true;
            else if (number2Find > array[mid]) {
                  low = mid;
            }
            else {
                  high = mid;
            }
      }
      return mid;
}
```

Looking at an example will help illustrate how the binary search works. Suppose the user wants to find the number 13 in the follow list of numbers.

NumberList = {3, 6, 9, 10, 13, 17, 33, 34, 41, 53, 57, 59, 61, 99, 101, 188}

When the algorithm is first entered, low is 0, high is 15. In the first iteration of the while-loop, mid is set to 7, (0 + 15)/2 is 7 in integer arithmetic. NumbeList[7] is 34; 13 is less than 34 so high is set to 7.

In the second iteration, mid is set to 3; number2Find, 13, is greater than NumberList[3], 10, so low is set to 3.

In the third iteration mid is set to 5; numbe2Find, 13, is less than NumberList[5], 17, so high is set to 5.

In the fourth iteration mid is set to 4, number2Find, 13, is equal to NumberList[4], so the search ends.

Because the binary search works by splitting the size of set of numbers to be searched at each iteration, it is of $O(\log_2 n)$.

12.2 Basic Searches with Efficiency Modifications

In order to increase the efficiency of search, we present an method; very similar to the method of indexed linked lists presented in 6.3.1. The difference here is that an array is being used instead of a linked list, so the modification is a bit simpler.

Imagine a list of names that is in alphabetical order, with the first name, a name that starts with an 'a' at index 0. Suppose that there are 3 names that start with 'a', and then at index 3 the first name that starts with a 'b' resides, then after 3 names starting with 'b', the first name that starting with a 'c' appears, and so forth and so on. In order to decrease our search time we record the index of the first occurrence of each letter and store the information in an array. If searching for a particular name in the array, we can use the array to jump to the beginning of the section in which the name resides. This section can then be searched using a simple sequential search or even a binary search. Table 12.1 below shows an index table. The indices point in this talbe point into table 12.2 which holds the data.

letter	***index***
a	0
b	3
c	6
d	9
e	11
f	12
g	13
h	14
i	-
j	15
k	-
l	17
m	-
n	18
o	19
p	-
q	20
r	-
s	-

t	-
u	21
v	22
w	25
x	28
y	29
z	30

Table 12.1. This table shows an index table for an alphabetically arranged list of names. The data is shown in Table 12.2 below. In order to make the search more efficient, the first letter of the name to be looked for is used to find the index (array index) of the section where the name resides. For example, suppose we are looking for "vicky". Using the index table, we find that we can jump straight to index 25 by getting setting our index to : index = indexTalbe['v']. Once we are in the correct section, we can sequentially search just that section, or if the sections are large, like those of a phone book, we can use a binary search in the section. The entries that have '-' in them indicate that there is no name that starts with this letter.

index	***name***
0	aaron
1	abby
2	anny
3	bart
4	billy
5	bob
6	carl
7	chuck
8	cris
9	dan
10	dale
11	ed
12	fred
13	greg
14	howie
15	jethro
16	jim
17	lance
18	nick
19	ollie
20	quincy
21	ursula
22	vanessa
23	vicky
24	virginia
25	wanda
26	wendy
27	wilma
28	xavier
29	yvonne

30	zack
31	ziggy

Table 12.2. This table hold the data for the index name search example. Notice that the yellow shaded entries are contain the first name that starts with a letter. For example, entry 3 is shaded, and it contains the first name that begins with a 'b'. Entry 9 is shaded, it contains the first name that starts with 'd', and so forth.

So in summary, if the data is going to only be searched once, the most efficient way to find the target datum is a simple sequential search, unless the data is already ordered. If that is the case, then a binary search is best. If the data is going to searched multiple times, ordering the data is a good idea so that a binary search can be used. If the number of items is large, an indexing system will make the search go much quicker. When searching the individual sections, a simple sequential search is fine, as long as the sections are small. If each section contains many data items, then a binary search of the section will result in a very efficient search.

12.3 Exercises

For the following exercises, unless otherwise specified, generate a list of 1000 random numbers and store them in a file. The numbers can generated using the technique described in exercise 4 at the end of chapter 7.

1. Create a sequential search in that finds the first instance of a particular number.
    a. Record the number of comparisons required to find the number.
    b. Now for each number between 0 and 100, find the first occurrence of the number, recording the number of comparisons required.
    c. Calculate the average number of comparisons needed to find the first occurrence of all the numbers.
2. Sort the list of numbers.
    a. Create an index that points to the first occurrence of each multiple of 10. That is, create an index that points at the into the following sections, 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100.
    b. Now for each number between 0 and 100, find the first occurrence of the number, recording the number of comparisons required. Use the index.
    c. Calculate the average number of comparisons needed to find the first occurrence of all the numbers.
3. Sort the list of numbers.
    a. For each number use a binary search to find 0 and 100, find the first occurrence of the number, recording the number of comparisons required.
    b. Calculate the average number of comparisons needed to find the first occurrence of all the numbers.
4. Using the names list from exercise 1 of section 9.3, sort the list of names, and create an index table for the list. Create a simple text menu that lets the user search for a name. Count the number of comparisons made when using the index, and when not using the index.

References

1. ascii table - http://www.asciitable.com/
2. http://www.vectorstock.com/royalty-free-vector/elephant-line-vector-5987
3. http://betterexplained.com/articles/techniques-for-adding-the-numbers-1-to-100/