

C, C++, and Data Structures

Dr. Fernando Gonzalez, Florida Gulf Coast University

Copyright 2017

Table of Contents

CHAPTER 1 INTRODUCTION TO PROGRAMMING.....	5
1.1 Introduction	6
1.2 Binary Numbers	9
1.3 Assembly and Machine Instructions	10
CHAPTER 2 COMPUTATIONS	12
2.1 Introduction	12
2.2 Output Formatting Using printf()	16
2.3 Variable Types	17
2.4 Programming	19
2.5 Operator precedence	21
CHAPTER 3 THE IF-ELSE INSTRUCTION.....	23
3.1 Introduction	23
CHAPTER 4 WHILE LOOPS	25
4.1 Introduction	25
4.2 Example – Counting until Overflow	27
CHAPTER 5 FUNCTIONS.....	29
5.1 Introduction	29
5.2 Recursive Functions	30
5.3 Simple Pointers.	33
5.4 Scope	35
CHAPTER 6 DO-WHILE AND FOR LOOPS	40
6.1 Introduction	43
6.2 Example – Errors in Computations	45
6.3 Example Using for Loops	46
6.4 Example – Square Root	47
6.5 Example – Root Finding using Newton’s Method	48
CHAPTER 7 SEQUENTIAL FILES	50
7.1 Introduction	50
7.2 Example – Plotting a Function	54
7.3 Example – Plotting a Function	58
7.4 Random Access Files	59
7.5 File encryption	61
CHAPTER 8 ARRAYS.....	64
8.1 Introduction	64
8.2 Arrays and Pointers	68
8.3 Multidimensional Arrays	68
8.4 Ex – Parsing using a Finite State Machine (FSM) algorithm	69

CHAPTER 9 STRUCTURES	75
9.1 Introduction	75
9.2 Advanced Structures	78
CHAPTER 10 CLASSES.....	82
10.1 Introduction	82
10.2 More on Classes	85
CHAPTER 11 COMPLEXITY ANALYSIS OF ALGORITHMS.....	101
11.1 Introduction	101
CHAPTER 12 LISTS	104
12.1 Introduction	104
12.2 The General List Abstract Data Type	104
12.4 Array implementation of the general list	104
The Stack.....	105
The Queue	106
12.5 Linked list implementation of general list	110
12.6 Doubly linked list	114
12.7 Linked List in Arrays	116
12.8 Comparisons between Different Implementation Methods	120
CHAPTER 13 SEARCHING ALGORITHMS.....	121
13.1 Introduction	121
13.2 Comparison Trees	124
Analysis of the laze binary search algorithm.	127
Analysis of the more complex binary search algorithm.....	127
CHAPTER 14 TREES.....	130
14.1 Introduction	130
14.2 Implementation of trees	130
14.3 Binary trees	131
14.4 Expression trees	132
14.5 Binary Search Trees	134
14.6 Average case analysis	139
CHAPTER 15 AVL TREES.....	141
15.1 Introduction	141
15.2 Single Rotation	143
15.4 Lazy Deletions	152
CHAPTER 16 SORTING ALGORITHMS.....	153
16.1 Introduction	153
16.2 Insertion Sort	153
16.3 Shell Sort	156
16.4 Selection Sort	160
16.5 Merge Sort	162
16.6 QuickSort	165

CHAPTER 17 HASH TABLES	173
17.1 Introduction	173
17.2 Open Hashing	176
17.3 Closed Hashing	176
 CHAPTER 18 HEAPS	 181
18.1 Introduction	181
18.2 Structural Property	181
18.3 Basic Heap Operations	182
18.4 The algorithm	184
 CHAPTER 19 GRAPH ALGORITHMS	 188
19.1 Introduction	188
19.2 Topological Sort	189
19.3 Shortest Path Algorithms	192
Unweighted Shortest path.....	192
19.4 Weighted Shortest Path (Dijkstra's Algorithm)	197
19.5 Network Flow Problems	202
19.6 The Time Complexity.	208
19.7 Minimum Spanning Tree Prim's Algorithm	208
19.8 Minimum Spanning Tree Kruskal's Algorithm	212
19.9 Euler Circuits	213
19.10 Introduction to NP-Complete	218
19.11 The Halting Problem	219

Preface

This document was written by Dr. Fernando Gonzalez for use in introductory C/C++ classes and in Data Structures. Work on this document started in 1997 at the University of Central Florida and is continuously being updated at Florida Gulf Coast University. This document may be freely used and distributed in its original form. Modifying this document in any form is strictly prohibited.

This document is intended to accompany the course text book by providing a small and concise explanation of the material requiring minimal reading. It can be used as a starting point with the course text used as a reference. This document does not include everything there is to learn in C, C++ or Data Structures. It can be used as if it were a set of notes taken from the lectures.

Dr. Fernando Gonzalez is an Associate Professor in the Software Engineering Department at Florida Gulf Coast University. Dr. Gonzalez received his Ph.D. degree in electrical engineering from the University of Illinois at Urbana-Champaign in 1997, his Master's degree in electrical engineering and Bachelor's degree in computer science from Florida International University in Miami Florida. He has been a faculty at the University of Central Florida, Texas A&M International University and is currently an Associate at Florida Gulf Coast University. He was a Computer Science Technical Staff Member at Los Alamos National Laboratory in Los Alamos New Mexico.

Chapter 1 Introduction to Programming

1.1 Introduction

The following is a simple C program that simply prints the phrase “Hello World” to the screen.

```
/* The traditional first program in honor
   of Dennis Ritchie who invented C at Bell Labs */

#include    <stdio.h>
int    main( )
{
    printf(“Hello World! \n”);
    return 0;
}
```

The following describes each section of the code.

```
/* The traditional first program in honor
   of Dennis Ritchie who invented C at Bell Labs */
```

These two lines are comments. Every character between “/*” and “*/” is ignored by the compiler.

The code is a list of ASCII characters. The end of line characters are ignored by the compiler so you may put several instructions per line or several lines per instruction.

The C language has only one build in function. However the manufacturer of the compiler provides a set of libraries that include many commonly used functions. When ever you need to use a predefined function that is in one of the compiler supplied libraries simply use it and the compiler will recognize the function call and bring it in. However, as with all functions, you must tell the compiler how to use it. That is, what variables go into it and out of it. For the library functions, a description of each function can be found in the supplied header files. The line:

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

tells the compiler to include the file called stdio.h. This file, called a header file, contains the description of the printf() function. There are many header files supplied with the compiler. Soon you will make your own header files.

The include statement is not an instruction but rather a command to the compiler to tell it to open the file in the < and > and copy it into your file. Its like a cut-and-paste operation except that your program file is not saved with the pasted stuff in it.

The line:

```
int    main( )
```

tells the compiler that a function is going to be described next and it is called “main”. To If the function required any input the list of required variables will be listed between the parenthesis. The int means the function will return as output an integer.

Unlike other languages, C does not have a main body. It consists only of functions. However the operating system, DOS called by Windows 98, expects a function called “main()”. This is in effect the main body of the program. It is the first function executed.

Next the body of the function is specified. So that the compiler knows where this function begins and ends, the instructions are enclosed in these braces ”{“ and “}”. Every instruction enclosed in the braces is part of the function.

The line:

```
printf(“Hello World! \n”);
```

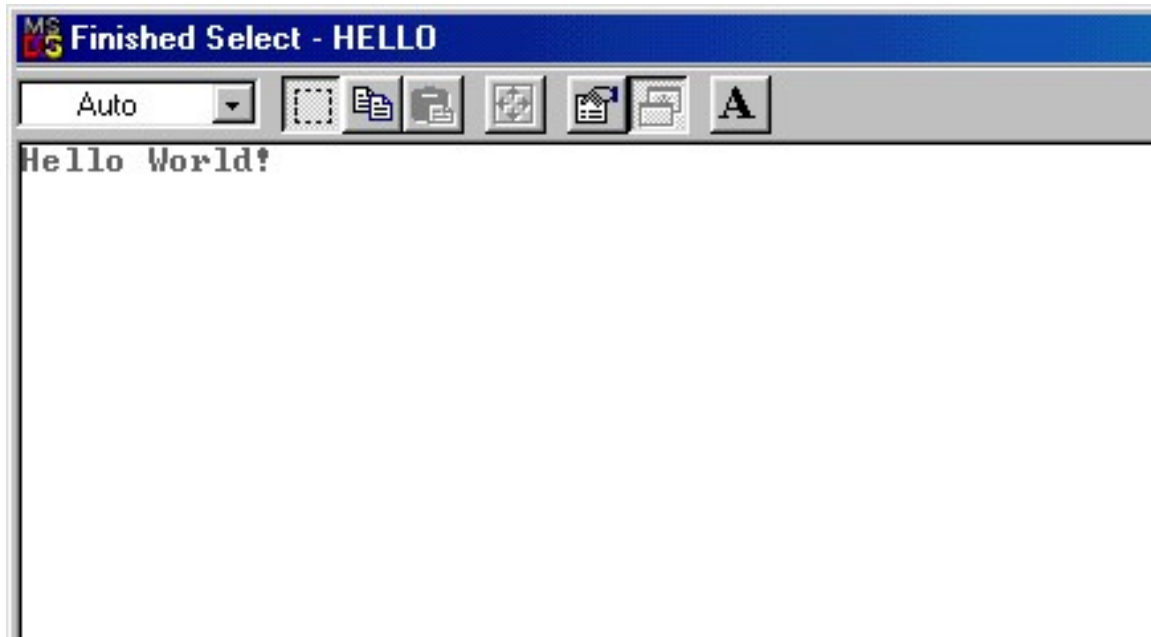
is a function call. This function is supplied by the compiler’s manufacturer and is linked into your program before executing. It comes precompiled. The “printf()” function sends ASCII data to the screen. The data is in terms of ASCII codes and is enclosed in parenthesis. Note the phrase “Hello World!” will be printed. Also the curser will be placed bellow the H on the next line. The ASCII code “\n” tells the function to print a carriage return. Note that when you call the function you send it a string. A string is a list of ASCII codes. This is the input to the function.

Some operating systems require that you return an integer from the main function. This integer can be used to tells the operating system that the program ran OK or did not. Today this returned value is seldom used but it is a giid idea to return it incase your platform requires it.

The line:

```
Return 0;
```

tells the function to return now and return the integer 0.
The output of the program is:



In Java the same program will look like this:

```
package example1;
public class Example1
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

The following describes each section of the code.

The 1st line includes the package this class is stored in. Unlike C++, in Java all code in must be in a class. The classes are grouped and stored in packages.

The 2nd line is the class definition. The public says that this class may be access from outside the package. Sometimes a class me only be used by other classes in the package and is not intended to be used outside of the package.

The class keyword simply states that this line is the definition of a new class. The class is enclosed in braces.

The next line is a member function declaration. In C/C++ you have functions that are independent to any class and there are functions that are part of a class. These are called member functions. In Java there are only member functions as there cannot be a function independent of a class. The public means the class may be accessed (called) from outside the class. The static means the function may be called from the class itself as opposed to creating an object of this class and calling the function via the object. More will be

discussed in the objet oriented chapter. The void is the same as in C and so is the main and its arguments. For a class to be executed directly it must have a main member function. Otherwise the class may be called from some other code but not executed directly.

To print to the console the predefined class System is used. Within the System class the out section is called and within this section the println() member function is called. This function accepts a string as input and prints it on the console.

1.2 Binary Numbers

In the computer everything is represented as binary. Each bit of information has two states: on or off. All of the data in the program including numerical and printable data as well as the programming instructions are stored as binary.

Binary numbers is a numbering system like decimal but where there are only 2 symbols as opposed to 10. One symbol represents the off state on the other the on state.

In decimal a number like 5472 is interpreted as:

$$5472 = 5 \times 10^3 + 4 \times 10^2 + 7 \times 10^1 + 2 \times 10^0$$

Note the 10 in the equation represents decimal. This is the number of symbols in the numerical system.

For binary we have 1101 represented as:

$$1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

The 2 here represents the fact that there are only 2 symbols.

A byte is the smallest unit of memory that the processor handles. Each byte in memory has an address associated with it. A byte contains 8 bits where a bit is 1 piece of information that can take on 1 of 2 states. This is usually implemented with a flip-flop. Since the output of assemblers and other programs that deal with the machine language must present the data in binary and since binary numbers are so long these numbers are usually represented using hexadecimal numbers. A hexadecimal number is one that uses 16 symbols. Hex was chosen since $2^4 = 16$ we can represent 4 bits exactly with 1 hex digit. And since there are exactly 2 sets of 4 bits in a byte, we could represent a byte with exactly 2 hex digits rather than 8 binary digits. This yields more compact representation. Using decimal we could only represent 3 bits but since we only need 8 symbols to represent 3 bits there will be 2 left over symbols. This will be messy.

For hex numbers since we need 16 symbols we use the first 10 symbols from the decimal numbers 0, 1, 2... 9 and add an additional 6 symbols from the alphabet. We use A, B, C, D, E, and F for the other 6 symbols. So the numbers go from 0 to F. A is 10 decimal, B is 11 and so on until F is 15 decimal.

For hexadecimal numbers we have A7E3 represented as:

$$A7E3 = 10 \times 16^3 + 7 \times 16^2 + 14 \times 16^1 + 3 \times 16^0$$

The 16 here represents the fact that there are 16 symbols.

1.3 Assembly and Machine Instructions

A program is actually a list of numbers. A computer can only store numbers in its memory. Since we use a logic "1" or "0" in the circuitry all numbers are in fact in binary. The numbers are represented in hexadecimal values so that we reduce the number of digits needed in the representation.

A compiler converts your program into machine code. One can program directly in machine code using an assembler. An assembler converts a program written in assembly code (almost machine code) into machine code. The assembly code is built using machine instructions but symbols are used instead of numbers to facilitate programming.

The following is the Hello World program written in assembly language:

```
title Hello World Program                (hello.asm)

; This program displays "Hello, world!"

dosseg
.model small
.stack 100h
.code
main proc
    mov     ax,@data
    mov     ds,ax

    mov     ah,09h
    mov     dx,offset msg
    int     21h

    mov     ax,4C00h
    int     21h
main endp

.data
msg db      'Hello World!',0Ah,0Dh,'$'
end main
```

The assembler converts the assembly code to machine code by looking up codes. The following is the assembled machine code:

```

Turbo Assembler      Version 2.0          02/02/98 13:03:54      Page 1
hello.ASM
Hello World Program          (hello.asm)

1
2                ; This program displays "Hello, world!"
3
4                dosseg
5      0000      .model    small
6      0000      .stack   100h
7      0000      .code
8      0000      main     proc
9      0000 B8 0000s      mov     ax,@data
10     0003 8E D8        mov     ds,ax
11
12     0005 B4 09        mov     ah,09h
13     0007 BA 0000r      mov     dx,offset msg
14     000A CD 21        int     21h
15
16     000C B8 4C00      mov     ax,4C00h
17     000F CD 21        int     21h
18     0011                main     endp
19
20     0011      .data
21     0000 48 65 6C 6C 6F 20 57+  msg     db 'Hello World!',0Ah,0Dh,'$'
22           6F 72 6C 64 21 0A 0D+
23           24
24                end     main

```

Notice the ASCII codes.

The actual program is the machine codes in the left column. The code is stores separately from the data. The following is the machine code for the program:

```
B8 00 00 8E D8 B4 09 BA 00 00 CD 21 B8 4C 00 CD 21
```

And the data is:

```
48 65 6C 5C 6F 20 57 6F 72 6C 64 21 0A 0D 24
```

The memory dump of the area in memory that the program resides is:

[Inactive C:\DISK6L~1\FERNANDO\MYDUMP.EXE]																
80	0B	00	09	68	65	6C	6C	6F	2E	61	73	6D	E9	88	1F	*
00	00	00	54	75	72	62	6F	20	41	73	73	65	6D	62	6C	*
65	72	20	20	56	65	72	73	69	6F	6E	20	32	2E	30	B9	*er
88	11	00	40	E9	E3	65	42	24	09	68	65	6C	6C	6F	2E	*
61	73	6D	04	88	03	00	40	E9	4C	96	02	00	00	68	88	*asm
03	00	40	A1	94	88	03	00	C0	9E	17	96	0C	00	05	5F	*
54	45	58	54	04	43	4F	44	45	96	98	07	00	48	11	00	*TEXT
02	03	01	02	96	0C	00	05	5F	44	41	54	41	04	44	41	*
54	41	C2	98	07	00	48	0F	00	04	05	01	00	96	00	05	*TA
53	54	41	43	48	05	53	54	41	43	48	67	98	07	00	74	*STACK
00	01	06	07	01	DE	96	08	00	06	44	47	52	4F	55	50	*
8B	9A	06	00	08	FF	03	FF	02	55	88	04	00	40	A2	01	*
91	A0	15	00	01	00	00	B8	00	00	8E	D8	B4	09	BA	00	*
00	CD	21	B8	00	4C	CD	21	D5	9C	0A	00	C8	01	55	01	*
C4	08	14	01	02	58	A0	13	00	02	00	00	48	65	6C	6C	*
6F	20	57	6F	72	6C	64	21	0A	24	D3	8A	07	00	C1	00	*o Wor
01	01	00	00	AC												*

You can dig out the machine code and data.

A compiler converts high-level source code like C into machine code. It skips the assembly code step.

The machine code is unique to each platform. An executable program is a program that is already compiled and is in machine code. An executable program compiled on a PC cannot run on a Sun or Mac for example.

Chapter 2 Computations

2.1 Introduction

The following is a simple C program that does some simple computation. It computes the pay an employee must receive given the number of hours worked, the pay rate and the tax rate.

```
/* This program computes the pay an employee must
receive given the number of hours worked, the pay rate and
the tax rate.*/
```

```
#include <stdio.h>

int main( )
{
    double    HoursWorked,
              PayRate,
              TaxRate,
              GrossPay,
              Tax,
              NetPay;

    printf ("Enter the number of hours worked : ");
    scanf( "%lf", &HoursWorked);
    printf ("\nEnter the pay rate : ");
    scanf( "%lf", &PayRate);
    printf ("\nEnter the tax rate : ");
    scanf( "%lf", &TaxRate);

    GrossPay = HoursWorked * PayRate;
    Tax = GrossPay * TaxRate;
    NetPay = GrossPay - Tax;

    printf("\nThe gross pay is %lf, ", GrossPay);
    printf("the tax is %lf, ", Tax);
    printf("and the net pay is %lf \n", NetPay);

    return 0;
}
```

The statement

```
double    HoursWorked,  
          PayRate,  
          TaxRate,  
          GrossPay,  
          Tax,  
          NetPay;
```

is a variable declaration. This statement is declaring 6 variables to be of type double. A variable declaration is a way of telling the compiler to reserve some memory. The reserved memory also has a label, which is the variable name. We access the memory by calling it by name. The type double means that the memory can hold a real number. It has the capability to store numbers in floating point representation. Type float can also be used but type double is a double precision version of type float.

The line:

```
printf ("Enter the number of hours worked  : ");
```

prints the phrase “Enter the number of hours worked : “ on the screen. It is asking the user for information.

The line:

```
scanf( "%lf", &HoursWorked);
```

gets a real number from the keyboard. The scanf() function is use to read information from the keyboard.

The data is entered using the keyboard which always enters data in ASCII code. The function scanf () not only reads the data but converts it to binary (numeric) form. The %lf means that it is to read a real number and convert it to a binary number of type double.

The argument &HoursWorked tells the function to store the number read from the keyboard into the variable named HoursWorked. The & sign tells the compiler to pass the address of the variable to the function and not its contents. The function needs the address so that it knows where to deliver the data. Its like buying a washing machine from Sears and giving the delivery man the address of your house.

The segment of code:

```
printf ("Enter the number of hours worked : ");
scanf( "%lf", &HoursWorked);
printf ("\nEnter the pay rate : ");
scanf( "%lf", &PayRate);
printf ("\nEnter the tax rate : ");
scanf( "%lf", &TaxRate);
```

gets the input data from the user. It ask a question using the printf () function then gets the user's response by using the function scanf () to read the keyboard.

The line:

```
GrossPay = HoursWorked * PayRate;
```

Computes the gross pay. It gets the contents of the variable HourWorked, then gets the contents of the variable PayRate. Next it multiplies them together and finally stores the result into the variable GrossPay. The symbol * means multiplication and the symbol = means to store the result found to the right of the symbol into the variable to the left of the symbol.

The line :

```
Tax = GrossPay * TaxRate;
```

Multiplies the contents of the variable GrossPay by the contents of the variable TaxRate and stores the result into the variable Tax.

The line:

```
NetPay = GrossPay - Tax;
```

Subtracts the contents of the variable Tax from the variable GrossPay and stores the result into the variable NetPay.

At this point the results have been computed and stored into the proper variables and the program is ready to output the results.

The lines:

```
printf("\nThe gross pay is %lf, ", GrossPay);
printf("the tax is %lf, ", Tax);
printf("and the net pay is %lf \n", NetPay);
```

output the results by sending the list of ASCII characters to the screen.

The line:

```
printf("\nThe gross pay is %lf, ", GrossPay);
```

prints the phrase “The gross pay is “ then prints the computed value of the gross pay that is stored in the variable GrossPay. The %lf is a place holder. It tells the printf () function to get the contents of the first variable in the list, convert it to ASCII using a floating point representation and print it where the place holder is at.

The other three lines work in the same way.

Finally the last line:

```
return 0;
```

tells the function to quit and return from where it came from.

Notice the flow of the program. We first get the data from the user using a combination of printf () and scanf () functions. Next we process the data to produce the results and finally we output the data using the printf () function. This is not always the flow of the program but is very common. In this class most of the programs will have this type of process flow.

2.2 Output Formatting Using printf()

Consider the following statement:

```
printf ("%8.3f %8.3f %10.3f\n", X, Y, Z);
```

The printf () function prints the formatted output to the console. The string argument is the format which contains printable characters as well as place holders. The percent indicates a place holder where a value is put in its place. The variables or values in the argument list are placed into the place holders in sequential order.

The 8 in 8.3 means that a total of 8 characters will be printed including the decimal point and the characters used to print the fraction. The 3 in 8.3 means 3 of the 8 characters will be used to print the fraction part.

The f means the data type of the variable is float.

The \n tells to print a new line at that point.

The \ means that an escape character follows. The escape characters are:

Name	Character	Effect
Newline	\n	carriage return, line feed
Backspace	\b	back space
Tab	\t	prints a tab
Alert	\a	sounds a beep
Vertical tab	\v	line feed, moves up one line
Carriage return	\r	returns cursor to the beggining of the line

The conversion type, the letter after the %, tells the type of the variable to convert. The following table lists the letters and its type conversion.

d,i	Integer
c	Character
f	Float
lf	Double
e	double with exponent
%	output %

Examples:

2.3 Variable Types

A variable is a memory location that is used by the program. Unlike code and constants, the information in a variable varies over time. In C one gives names to variables. Valid names are usually limited to 32 characters. They must not start with a number and must not have the same name as a reserved word. Some examples of legal names are:

X
X1
k95
slew_rate
SLEW_RATE
Hello

Some examples of invalid names are:

7X	must not start with a number
slew*rate	must not include *. The compiler will think you mean slew multiplied by rate.
slew-rate	same as above.
slew rate	the compiler will think you have two variables, slew and rate.

Data types are used in a language to describe what type of information or data will be stored in the variable. The compiler needs to know since different types require different

amount of data and must be handles differently. There are 3 basic types in C: integer, floating point also called real or double and character. Integers can only hold integers numbers. They cannot handle a fractional part. Floating point numbers can store real numbers. It allocates space for a fractional part. They are called floating point since by changing the value of the exponent one can move the decimal point. The computer changes or adds an exponent so that the number can be represented with all digits behind the decimal. In this form the number is said to be normalized. It always stores real numbers in normalized form. Characters are stored in a smaller space since all character codes (ASCII codes) only take 8 bits (1 byte).

In C the amount of space may vary from compiler to compiler. The most common values are 2 or 4 bytes for integers, 10 for floating point and 1 byte for characters. The declarations are described below.

```
int    X;    // declare an integer called X.
float  Y;    // declare a floating point called Y.
char   Z;    // declare a character called Z.
```

With integers using 4 bytes this gives a range of values for X of $0 \leq X \leq 2^{32} = 4,294,967,296$. Or if negative numbers are required then the range is cut in half and one half is used for negative and the other for positive numbers.

One can declare a long integer as:

```
long int    LX;    // long integer
```

This space uses 8 bytes. Now $0 \leq X \leq 2^{64} \approx 1.844674407 * 10^{19}$. Again half the range may be used for negative numbers.

Floating point numbers can have a very large range. The limit in space effects the accuracy of the number by limiting the number of digits stored. A type of double may be used that uses double the amount of memory and therefore may store numbers with much more accuracy.

Characters only need 1 byte. To store a string of characters, for example a name, you will have to declare an array of characters. This will be covered later, but briefly, an array is a list of consecutive memory location of some specified type.

In summery we have:

```
int          X;          // declare an integer called X.
long int     LX          // declares a long integer LX.
float        Y;          // declare a floating point called Y.
double       DY;         // declares a double sized floating point called DY.
char         Z;          // declare a character called Z.
char         AZ[80];      // declares an array of 80 characters called AZ.
double       AD[5];       // declares an array of 5 doubles.
```

Most programming languages have a type called Boolean. It can store true or false. In C any type can be used as a Boolean. If the number is 0 then the variable is considered to be false otherwise if the number is not 0 the variable is considered to be true. We normally use integer for Boolean.

2.4 Programming

There are 6 steps to problem solving and program development:

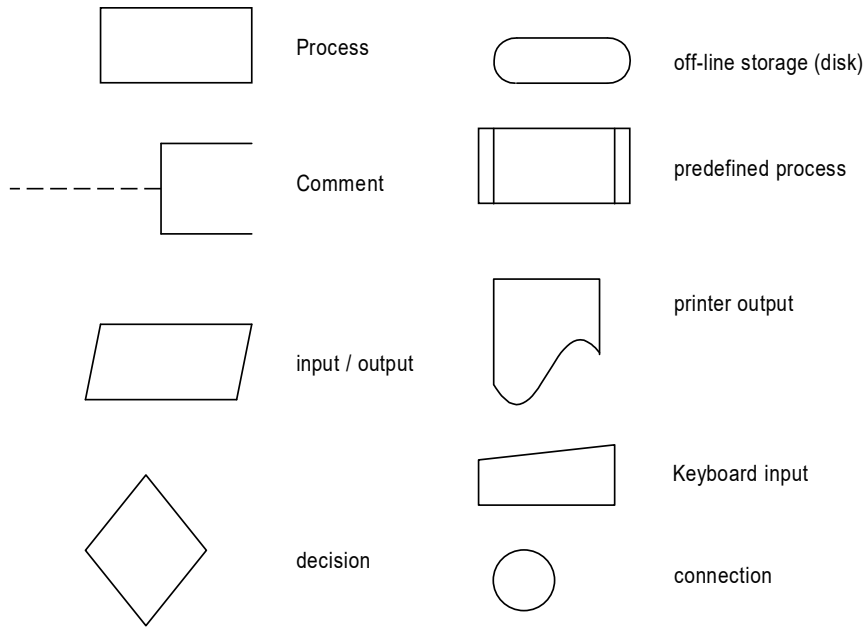
1. State the problem clearly.
2. Describe resources, data needed (input), expected results (output), and the variables required for the problem.
3. Work a sample data set by hand.
4. Develop an algorithm to solve the problem.
5. Code the algorithm.
6. Test the code using edit-compile-run cycle on a variety of data sets with known results.

The first step is perhaps the most difficult. This step will probably be revisited as the problem is better understood.

To represent the design of the algorithm one can use flow charts or pseudo code. Flow charts are visual but not as flexible as pseudo code.

For pseudo code, the code is like a general code that is only to be understood by people, not a compiler. One can put as much or as little detail as needed.

For a flow chart, the symbols are as follows.

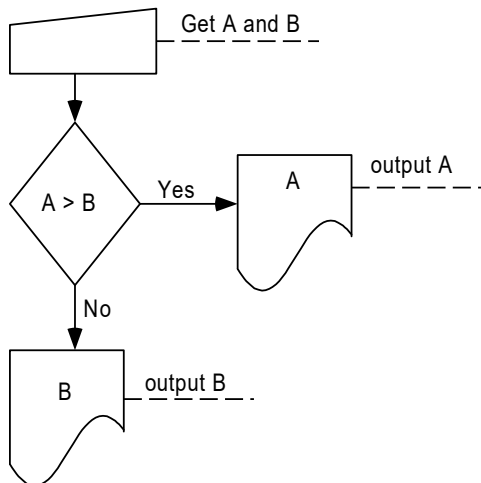


For example suppose I want to describe an algorithm that prints the larger of two number given.

In pseudo code:

```
get two number A and B
if A > B then
    output A
otherwise
    output B
```

With a flow chart:



2.5 Operator precedence

Operational precedence determines the order in which operations take place. The higher the operator in the table the higher priority it has. For example any operation in side of a parenthesis is performed first. All multiply operations are performed before and additive ones.

(,)	Parenthesis
++, --, +, -, !	Unary operators
*, /, %	Multiplicative operators
+, -	Additive operators
<, <=, >, >=	Relational operators
==, !=	Equality operators
&&	Logical AND operator
	Logical OR operator
?:	Conditional operator
==, +=, -=, *=, /=, %=	Assignment operators
,	coma

Consider the following program:

```
// precedence demo

#include      "stdio.h"

#defineFALSE      0
#defineTRUE !FALSE

void main()
{
    int x;

    x = 1 + 2 * 3 + 4;
    printf ("x = 1 + 2 * 3 + 4; results in x = %d\n",x);

    x = (1 + 2) * (3 + 4);
    printf ("x = (1 + 2) * (3 + 4); results in x = %d\n",x);

    x = 0;
    x = ++x * 2;
    printf ("x = 0; x = ++x * 2; results in x = %d\n",x);

    x = -1 + 2;
    printf ("x = -1 + 2; results in x = %d\n",x);

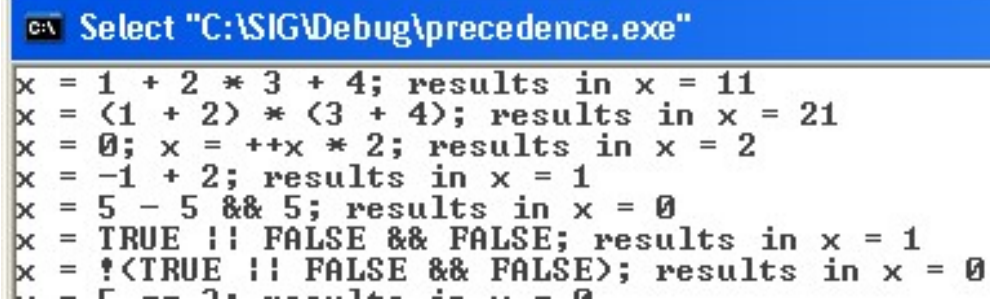
    x = 5 - 5 && 5;
    printf ("x = 5 - 5 && 5; results in x = %d\n",x);
```

```
x = TRUE || FALSE && FALSE;
printf ("x = TRUE || FALSE && FALSE; results in x = %d\n",x);

x = !(TRUE || FALSE && FALSE);
printf ("x = !(TRUE || FALSE && FALSE); results in x = %d\n",x);

x = 5 == 3;
printf ("x = 5 == 3; results in x = %d\n",x);
}
```

The output is:



```
C:\ Select "C:\SIG\Debug\precedence.exe"
x = 1 + 2 * 3 + 4; results in x = 11
x = <1 + 2> * <3 + 4>; results in x = 21
x = 0; x = ++x * 2; results in x = 2
x = -1 + 2; results in x = 1
x = 5 - 5 && 5; results in x = 0
x = TRUE || FALSE && FALSE; results in x = 1
x = !<TRUE || FALSE && FALSE>; results in x = 0
x = 5 == 3; results in x = 0
```

Chapter 3 The if-else Instruction

3.1 Introduction

The following is a simple C program that does some simple computation. It computes the pay an employee must receive given the number of hours worked, the pay rate, tax rate and the overtime pay rate.

```
/* This program computes the pay an employee must receive given the
number of hours worked, the pay rate and the tax rate.*/

#include    <stdio.h>

int    main( )
{
    double    HoursWorked,
              PayRate,
              TaxRate,
              GrossPay,
              Tax,
              NetPay,
              OverTimePayRate;

    printf ("Enter the number of hours worked  : ");
    scanf( "%lf", &HoursWorked);
    printf ("\nEnter the pay rate  : ");
    scanf( "%lf", &PayRate);
    printf ("\nEnter the tax rate  : ");
    scanf( "%lf", &TaxRate);
    printf ("\nEnter the over time pay rate  : ");
    scanf( "%lf", &OverTimePayRate);

    if (HoursWorked > 40)
        GrossPay = 40 * PayRate +
                  (HoursWorked - 40) * OverTimePayRate;
    else
        GrossPay = HoursWorked * PayRate;

    Tax = GrossPay * TaxRate;
    NetPay = GrossPay - Tax;

    printf("\nThe gross pay is %lf, ", GrossPay);
    printf("the tax is %lf, ", Tax);
    printf("and the net pay is %lf \n", NetPay);
    if (HoursWorked > 40)
        printf("Worked overtime this week \n");

    return 0;
}
```

The only new statement added to this program is the if statement. The if statement is a way to guide the process in one way versus another.

The statement:

```
if (HoursWorked > 40)
    GrossPay = 40 * PayRate +
                (HoursWorked - 40) * OverTimePayRate;
else
    GrossPay = HoursWorked * PayRate;
```

First compares the contents of the variable HoursWorked to 40. If the contents of the variable HoursWorked is greater than 40 then it executes the very next statement. If not, that is the contents of the variable is less than or equal to 40, then the statement after the else statement is executed instead. The else statement means that if the condition is false then execute the statement after the else instead of executing nothing.

Note that only one of the two statements will be executed, not both. After the if statement the flow resumed to a simple sequential ordering like before.

Also the statement:

```
GrossPay = 40 * PayRate +
            (HoursWorked - 40) * OverTimePayRate;
```

computes the pay knowing that there is overtime. In C the multiplications and divisions are performed before and additions and subtractions. Also and operation enclosed in parenthesis are performed before any other operation. In this statement the HoursWorked - 40 is performed first. Next the two multiplications are performed first then finally they are added together.

The statement:

```
if (HoursWorked > 40)
    printf("Worked overtime this week \n");
```

compares the contents of the variable HoursWorked to 40. If the contents of the variable HoursWorked is greater than 40 then it prints the phrase "Worked overtime this week". If the condition is false then the printf is skipped. Note, since we did not use the optional else statement nothing is executed when the condition is false. The process flow resumes after the if statement.

Chapter 4 While Loops

4.1 Introduction

The following is a simple C program that does some simple computation. It computes the pay an employee must receive given the number of hours worked, the pay rate, tax rate and the overtime pay rate.

```
/* This program computes the pay an employee must receive given the
number of hours worked, the pay rate and the tax rate and computes
overtime. It verifies the input for correct data*/

#include    <stdio.h>

#define     FALSE 0
#define     TRUE  !FALSE

int  main( )
{
    double      HoursWorked, PayRate, TaxRate, GrossPay, Tax,
                NetPay, OverTimePayRate;
    int         NeedData;

    NeedData = TRUE;

    while (NeedData)
    {
        printf ("Enter the number of hours worked  : ");
        scanf( "%lf", &HoursWorked);
        printf ("\nEnter the pay rate  : ");
        scanf( "%lf", &PayRate);
        printf ("\nEnter the tax rate  : ");
        scanf( "%lf", &TaxRate);
        printf ("\nEnter the over time pay rate  : ");
        scanf( "%lf", &OverTimePayRate);

        if ( HoursWorked > 0 && PayRate > 0 && TaxRate > 0 &&
            OverTimePayRate > 0)

            NeedData = FALSE;
    }

    if (HoursWorked > 40)
        GrossPay = 40 * PayRate +
                    (HoursWorked - 40) * OverTimePayRate;
    else
        GrossPay = HoursWorked * PayRate;

    Tax = GrossPay * TaxRate;
    NetPay = GrossPay - Tax;

    printf("\nThe gross pay is %lf, ", GrossPay);
    printf("the tax is %lf, ", Tax);
    printf("and the net pay is %lf \n", NetPay);
    if (HoursWorked > 40)
```

```
        printf("Worked overtime this week \n");  
    return 0;  
}
```

There are several new things in this program.

First is the use of the define directive. The define directive like the include directive is a message to the compiler only. It is not an executable statement. The define directive tells the compiler to replace all of the occurrence of a particular word with some other word.

The lines:

```
#define FALSE 0  
#define TRUE 1
```

tells the compiler to replace all of the occurrences of FALSE with the number 0 and all of the occurrences of TRUE with the number 1. It does this in the preprocessing stage.

Unlike in other languages, the C language does not have a variable type that is used for Boolean operations. The C language simply uses integers with the understanding that a value of 0 means false and any other non-zero value means true.

The next new statement is the int variable type. The int type is used to declare integers variables. A variable of type int can only store integer values. In fact in the PC platform it can only hold values from 0 to 65,535 or -32767 to +32768 depending on how you interpret the binary number.

The line:

```
int        NeedData;
```

declares a variable to be of type int. We will use this variable for Boolean operations.

The line:

```
NeedData = TRUE;
```

assigns the value of 1 to the variable NeedData. We are actually assigning a value of true since 1 is considered to be true when used in a Boolean operation.

In programming it is sometimes necessary to form loops. A loop is a segment of code that runs over and over until some condition is met. Some loops may be set up to run a predefined number of times.

The next statement is the while statement. This statement executes the very next statement repeatedly as long as the condition in the while statement is evaluated as true.

Also since only the statement following the while is part of the loop, and we want to put in several statements in the loop, we will use a compound statement. A compound statement is a group of statements enclosed by { and }. It is considered a single statement.

The statement:

```
while (NeedData)
{
    printf ("Enter the number of hours worked  : ");
    scanf( "%lf", &HoursWorked);
    printf ("\nEnter the pay rate  : ");
    scanf( "%lf", &PayRate);
    printf ("\nEnter the tax rate  : ");
    scanf( "%lf", &TaxRate);
    printf ("\nEnter the over time pay rate  : ");
    scanf( "%lf", &OverTimePayRate);

    if (  HoursWorked > 0 && PayRate > 0 && TaxRate > 0 &&
        OverTimePayRate > 0)

        NeedData = FALSE;
}
```

executes the segment of code enclosed in { and } while the condition is true. The condition is the value of the variable NeedData. This condition is true as long as the value in the variable is not 0. We will continue to execute this segment of code until the user enters all positive values. If any of the values entered are less than 0 then the process will loop again and the user will be asked to enter all values again. Once the user enters all positive values the if condition in the bottom of the loop evaluates to true and the variable NeedData is assigned the value of 0. Then when the process goes back up to the while condition this condition will evaluate as false and the loop will not be executed any more. The process will continue after the while statement (after the loop).

We do this to ensure that the user enters valid data. If the user enters invalid data, a negative amount, then the user will have to repeat the input process.

The statement within the loop:

```
if (  HoursWorked > 0 && PayRate > 0 && TaxRate > 0 &&
    OverTimePayRate > 0)

    NeedData = FALSE;
```

Has a complex condition. The && symbol means “and.” That is, the condition is evaluated to true if HoursWorked > 0 and PayRate > 0 and ... If all of these conditions are evaluated as true then the entire condition is true.

The symbol || is used for “or.” With the || and && symbols one can make complex decisions.

Notice also the fact that we are nesting statements. The if statement above is located inside of the while statement. Nesting of statements is how we build complex program flow.

4.2 Example – Counting until Overflow

The following program count until an overflow condition is met. The number displayed is the last number that was added before the number overflowed into the negative side of the number line. Note the algorithm first counts by 10,000 at a time to get to close to the edge of the number line then in counts by 1 at a time. This saves much running time.

```
////////////////////////////////////  
//  
// Author      : Dr. Fernando Gonzalez  
// Date       : 5/25/2004  
//  
// Program 4.1  
//  
// This program counts by 10,000 at a time until the integer is so  
// large that it wrapped around to the negative side. At this point  
// the second loop counts by 1 starting from the last number that  
// fit into an integer variable.  
//  
// Input       : None.  
// Output      : The largest integer this computer can represent to  
//               the console.  
// Assumptions : The largest integer > 10,000.  
//  
////////////////////////////////////  
  
#include "stdio.h"  
  
int main()  
{  
    int      next,  
           prev;  
  
    next = 1;  
    prev = next - 1;  
    while (next > prev)  
    {  
        prev = next;  
        next = next + 10000;  
    }  
  
    next = prev + 1;  
    while (next > prev)  
    {  
        prev = next;  
        next = next + 1;  
    }  
  
    printf(" The largest integer is %d \n",prev);  
  
    return 0;  
}
```

Chapter 5 Functions

5.1 Introduction

All we did to this program was to put the input part of the program into a separate function. The main function was getting too large and the details of how the program ask the user for data as well as how the data is collected are not relevant in the main program. Putting that part into a separate function allows us to remove this detail from the main program.

Consider the following program:

```
/* This program computes the pay an employee must receive given the
number of hours worked, the pay rate and the tax rate and overtime
rate. It verifies the input for correct data */

#include    <stdio.h>

#define     FALSE 0
#define     TRUE  !FALSE

void  GetData (    double *HW, double *PR, double *TR, double *OTR)
{
    int          NeedData = TRUE;

    while (NeedData)
    {
        printf ("Enter the number of hours worked  : ");
        scanf( "%lf", HW);
        printf ("\nEnter the pay rate    : ");
        scanf( "%lf", PR);
        printf ("\nEnter the tax rate    : ");
        scanf( "%lf", TR);
        printf ("\nEnter the over time pay rate  : ");
        scanf( "%lf", OTR);

        if ( *HW > 0 && *PR > 0 && *TR > 0 && *OTR > 0)
            NeedData = FALSE;
    }
}

int  main( )
{
    double          HoursWorked, PayRate, TaxRate, GrossPay, Tax,
                    NetPay, OverTimePayRate;

    GetData (&HoursWorked, &PayRate, &TaxRate, &OverTimePayRate);

    if (HoursWorked > 40)
        GrossPay = 40 * PayRate +
                    (HoursWorked - 40) * OverTimePayRate;
    else
        GrossPay = HoursWorked * PayRate;

    Tax = GrossPay * TaxRate;
```

```

    NetPay = GrossPay - Tax;

    printf("\nThe gross pay is %lf, ", GrossPay);
    printf("the tax is %lf, ", Tax);
    printf("and the net pay is %lf \n", NetPay);
    if (HoursWorked > 40)
        printf("Worked overtime this week \n");

    return 0;
}

```

Functions should be cohesive. That is they should consist of an independent task. For example if the function had asked the user for the hours worked and the pay rate in the function and then ask the user for the overtime rate and tax rate in the main function this will not be a cohesive function.

Since the function handles all of the user input, when we write this function we only need to concentrate on the task of gathering the data. We do not need to worry about the rest of the program. Furthermore when we write the main function we do not need to worry about getting the data from the user. Cohesive functions allows to decompose the program in logical parts and build each one independently.

The function consists of a header and a body.

The line:

```
void GetData ( double *HW, double *PR, double *TR, double *OTR)
```

is the header of the function. This line describes the input and output of the function. The input is the set of parameters enclosed in parenthesis. This function has 4 input variable, HW, PR, TR, and OTR. The function has the capability to output 1 value through the function's name. The void indicates that we are not going to output a value through this function.

The output of the function is indirect. Because we put an asterisk in front of each variable this means that it will receive the address of the variable instead of its contents. We need the address in order to know where to put the data. These variables are considered to be pass-by-reference.

The line:

```
int          NeedData = TRUE;
```

is a declaration of an integer. This variable is declared local to the function. That means only the function can see this variable.

The line:

```
GetData (&HoursWorked, &PayRate, &TaxRate, &OverTimePayRate);
```

Calls the function. By calling the function we execute the code in the body of the function. We must pass the data that the function needs. In this case the function needs the addresses of 4 variables. The & symbol in front of the variable tells the compiler to send the address of the variable instead of its contents.

5.2 Recursive Functions

A recursive function is a function that calls itself. Some applications lend itself to this type of programming while some do not.

The factorial of an integer is defined as:

$$n! = n(n-1)(n-2)\cdots(2)(1)$$

The following is a function that implements the factorial function using the first definition.

```
/*   This program computes the factorial of n where
    n!=n*(n-1)*(n-2)* ... 2*1 */

#include    <stdio.h>
#define     FALSE 0
#define     TRUE  !FALSE

long int factorial(int n)
{
    long int    prod = 1;

    for (; n > 1; n = n - 1)
        prod = prod * n;

    return prod;
}

main ()
{
    int    x;

    for (x = 0; x <= 10; x = x + 1)
        printf( "The factorial of %d is %ld\n", x, factorial(x) );
}
```

The output is:

```
The factorial of 0 is 1
The factorial of 1 is 1
The factorial of 2 is 2
The factorial of 3 is 6
The factorial of 4 is 24
The factorial of 5 is 120
The factorial of 6 is 720
The factorial of 7 is 5040
The factorial of 8 is 40320
The factorial of 9 is 362880
The factorial of 10 is 3628800
```

Alternatively we can define $n!$ as

$$n! = n(n-1)(n-2)\cdots(2)(1) = n(n-1)!$$

Now using the definition

$$n! = n(n-1)!$$

we can write a recursive function to implement it.

```
/* This program computes the factorial of n where
   n!=n*(n-1)! */

#include <stdio.h>
#define FALSE 0
#define TRUE !FALSE

long int factorial_recursive(int n)
{
    if (n <=1)
        return 1;
    else
        return n * factorial_recursive( n - 1);
}

main ()
{
    int x;

    for (x = 0; x <= 10; x = x + 1)
        printf("The factorial of %d is
%d\n",x,factorial_recursive(x));
}
```

The output is:

```
The factorial of 0 is 1
The factorial of 1 is 1
The factorial of 2 is 2
The factorial of 3 is 6
The factorial of 4 is 24
The factorial of 5 is 120
The factorial of 6 is 720
The factorial of 7 is 5040
The factorial of 8 is 40320
The factorial of 9 is 362880
The factorial of 10 is 3628800
```

The number of time a function can call itself before returning is limited by the size of the system stack. One must be careful not to exceed this limit as this will cause the program to crash.

5.3 Simple Pointers.

Variables are locations in memory reserved for the programmer use. Variables names are the labels that refer to the memory locations. Each variable therefore has a corresponding address. This address is the location where the reserved memory is stored. Pointers are simply a variable that is capable of holding an address. The pointer it self is a variable which means it is a location in memory reserved for its use. The only difference between a pointer variable and an integer variable, for example, is that the pointer's data is interpreted to be an address while the integer's data is interpreted to be simply a number of type integer. The data in the variables are simply numbers. Its only how the data is used that change.

To declare a pointer variable simply put an * in front of the variables name in the declaration as in:

```
int          *p;
```

A pointer must be declared to point to a specific type of variable. In the declaration above its defined to point to an integer type of variable. The following is a pointer declared to point to a double.

```
double       *p;
```

If one wants to have the pointer have the capability or actually permission, to point to any type of variable then it can be declared as follows:

```
void         *p;
```

If one wants to access what the pointer points to one can use *p. The * in front of the variable when its being used means that you are referring to the data at the location pointed to by the pointer. This means the address in the pointer is used to locate the data desired.

I one wants to refer to the address of some variable as opposed to its contents one can put & in front of the variable. The term &p refers to the address of the variable p not its contents. You can use & in front of any type of variable.

If the variable is of type pointer than any arithmetic to that variable will use Pointer Arithmetic. Pointer Arithmetic means that when you add 1 to an address it adds one address. That is it adds sufficient bytes to have the resulting address point to one variable more. The number that actually gets added depends on the size of the type of variable the pointer was declared to point to. A pointer to an integer add 4 to each unit while a long integer add 8.

The following program defines 3 integer and one pointer variables. The first group of printf() statements print the address of the variable followed by its contents and the variable's label (name).

The %X is a format that prints the number as it is in memory, 32 bits and represented in the computers natural hex format.

The second set of printf() statements prints several combinations of ip, &ip, and *ip so one can observe the difference between them.

```
//  
// This program demonstrates the use of pointers and that they actually are.  
  
#include      "stdio.h"  
  
void main()  
{  
    int      a,b,c;  
    int      *ip;  
    double   *dp;  
    double   d,e;  
  
    a = 5;  
    b = 6;  
    c = 7;  
    d = 37.45;  
    e = 100.25;  
    ip = &a;  
    dp = &d;  
  
    printf("\n\n");  
  
    printf(" address  contents  variable \n\n");  
    printf(" %08X | %08X | a \n",&a,a);  
    printf(" %08X | %08X | b \n",&b,b);  
    printf(" %08X | %08X | c \n",&c,c);  
    printf(" %08X | %08X | ip \n",&ip,ip);  
    printf(" %08X | %08X | dp \n",&dp,dp);  
    printf(" %08X | %8.3lf | d \n",&d,d);  
    printf(" %08X | %8.3lf | e \n",&e,e);  
  
    printf("\n\n");  
  
    printf(" a      has %x\n",a);  
    printf(" &a    has %x\n",&a);  
    printf(" ip    has %x\n",ip);
```

```

printf(" &ip    has %x\n",&ip);
printf(" *ip    has %x\n",*ip);
printf(" ip - 1  has %x\n",ip - 1);
printf(" *(ip - 1) has %x\n",*(ip - 1) );
printf(" dp - 1  has %x\n",dp - 1);
printf(" *(dp - 1) has %8.3f\n",*(dp - 1) );

printf("\n\n");
}

```

The output is

C:\ Select "C:\Documents and Settings\Fernando\Take to U		
address	contents	variable
0012FF7C	00000005	a
0012FF78	00000006	b
0012FF74	00000007	c
0012FF70	0012FF7C	ip
0012FF6C	0012FF64	dp
0012FF64	37.450	d
0012FF5C	100.250	e
a	has 5	
&a	has 12ff7c	
ip	has 12ff7c	
&ip	has 12ff70	
*in	has 5	

5.4 Scope

Each variable has a scope in which it is accessible. The scope of a variable is the area in which that variable can be accessed. The rules follows:

1. Any variable declared in a block, enclosed in { and }, has its scope limited to that block. This means that the variable is unknown out side of the block. Since a function is defined in a block, any variable declared inside of the function has its scope limited to that function.

```

{
int x;
your code

```

```
}
```

2. Any variable declared outside of a block has its scope unlimited. This variable is considered to be a global variable. In general this represents a bad programming practice unless you are a skilled programmer and can use them appropriately. If your program consists of several files that are individually compiled and linked, each file will have to “describe” the variable to the compiler by “redefining” it with the extern identifier. This is not a new declaration but rather a header. It simply tells the compiler that the variable is declared in another file and how it’s declared.

```
extern int x;
```

3. Any variable declared outside of a block but with the static identifier has its scope limited to all areas within the file. This is called file scope. If your program consists of several files that are individually compiled and linked the variables with file scope will be limited to only the file it is declared in. A header declaration using the extern identifier in another file will not bring the variable into scope.

```
static int x;
```

4. Any variable declared in a block is created when the block is executed and is removed and destroyed after execution of that block is terminated.
5. Any variable declared in a block using the static identifier will be kept, including its current value, after the block is terminated. The next time the block is executed the same valuable will be used and the last value it had will remain.

```
{
static int x;
your code
}
```

6. If there is more than 1 variable with the same name in scope then the one with the most immediate declaration is referenced. For example if there exist a global variable x and a local variable x as well, the local x will be used in the block and the global x will be used outside the block.

```
int x;
int y;
{
int x;
```

local x is referenced here.
global y is referenced here.

}

global x is referenced here.

global y is referenced here.

Consider the following program:

```
1 // This program demonstrates the variable scope rules.
2
3 #include    "stdio.h"
4
5 int    x = 10,
6        y = 100;
7
8 void a()
9 {
10     int x = 20;
11
12     x++;
12     y++;
13     printf("The x,y within function a is x = %d, y = %d\n",x,y);
14 }
15
16 void b()
17 {
18     static int z = 30;
19
20     z++;
21     printf("The z within function b is z = %d\n",z);
22 }
23
24 void main()
25 {
26     int    x = 40;
27
28     {
29         int x = 50;
30
31         x++;
32         y++;
33         printf("The x,y within function main inner block");
34         printf(" x = %d, y = %d\n",x,y);
35     }
36     x++;
37     y++;
38     printf("The x,y within function main is x = %d, y = %d\n",x,y);
```

39	a();
40	a();
41	b();
42	b();
43	}

In line 5 and 6 are global variables. They have scope throughout the program.

In line 10 the variable x has scope only in function a. Also function a can not access the global variable x declared in line 5 because its block by its local variable x.

In line 18 the variable z is declared as static. The first time function b is called the variable has 30 and the function add 1 to it making it 31. The next time its called it has a value of 31 in it from before.

In line 26 x is declared as local to main.

In line 29 x is declared within a local block within main.

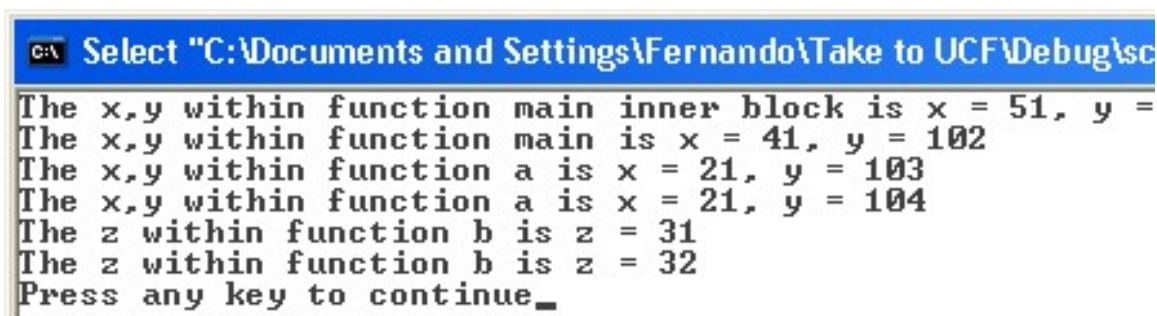
In line 31 the x declared in line 29 is referenced since it's the definition most immediate.

In line 32 the global y is referenced.

In line 36 the x declared in line 26 is referenced because it's the most immediate declaration. The x declared inline 29 is now out of scope and the x declared in line 5 is blocked by the x declared in line 26.

In line 37 the global y is referenced since it's the most immediate declaration.

The output is:



```

C:\ Select "C:\Documents and Settings\Fernando\Take to UCF\Debug\sc
The x,y within function main inner block is x = 51, y =
The x,y within function main is x = 41, y = 102
The x,y within function a is x = 21, y = 103
The x,y within function a is x = 21, y = 104
The z within function b is z = 31
The z within function b is z = 32
Press any key to continue_

```

5.5 Parameter Passing

There are three ways to pass parameters in C/C++. They include pass-by-value and pass-by-pointer, both in C and pass-by-reference, in C++. In general, a function cannot change

the state of any variable outside of its own function body by the scope rules. The only data passed out of the function is passed via its function name. The parameters are used as input to help it compute its output. In C the parameters are passed using pass-by-value. Consider the following function header and corresponding function call:

```
int    func(int x);
```

```
w = func( k );
```

The function receives the value that is stored in k at the time the function call is executed. The variable k itself is not passed. So, if k had the number 5 in it at that time, then the number 5 is passed. In reality, a copy of the data in the parameter is what is passed to the function. The function receives the data and stores it into a new local variable x.

Now consider the following function declaration and corresponding function call:

```
int    func(int *x);
```

```
w = function ( &k );
```

Now the address of variable k is what is being sent to the function. The function receives this address and stores it into its new local variable x that is declared to accommodate an address. In both cases, the variable x has a number and changing the data in x has no effect outside of the function. However, since k has an address, the function can grab the data from that address or deliver data to that address. This bypasses the scope rules and allows the function to modify the variable x that is declared outside of the function. This is called pass-by-pointer. Note, the actual parameter, x, is still a local variable and making changes to it has no effect outside of the function.

In general, if an address to a variable that is outside of its scope, that is, declared outside of the function body, is passed to a function, the function can “grab” or “deliver” data to that variable. The function, having the address of the variable can bypass the scope rules and access the variable’s content direct.

When C++ was created, pass-by-reference was included. Consider the following function declaration and corresponding function call:

```
int    func(int &x);
```

```
w = func( k );
```

Using this parameter passing method, one can think of the calling function sending the actual variable to the function. That is, x is the same variable as k. An understanding of how compilers create machine language program will show that this concept of sending an actual variable is absurd, however it serves as a conceptual way to think of this type of parameter passing. In reality, the method is pass-by-pointer, with the only difference is

that the compiler manages the pointers on behalf of the programmer. This makes coding much easier however, the calling code cannot send the null pointer when using pass-by-reference.

Method	Pass-by-value	Pass-by-pointer	Pass-by-reference
Header	<code>void func(int x);</code>	<code>void func(int *x);</code>	<code>void func(int &x);</code>
Calling code	<code>func(k); func(3);</code>	<code>func(&k);</code>	<code>func(k);</code>
Data reference	<code>x = 5;</code>	<code>*x = 5;</code>	<code>x = 5;</code>
Example Function declaration	<pre>int Add5(int x) { x = x + 1; return x + 5; }</pre>	<pre>int Add5(int *x) { *x = *x + 1; return *x + 5; }</pre>	<pre>int Add5(int &x) { x = x + 1; return x + 5; }</pre>
Example function call	<code>int k = 10; w = Add5(k);</code>	<code>int k = 10; w = Add5(&k);</code>	<code>int k = 10; w = Add5(k);</code>
Impact	k has 10 w has 16	k has 11 w has 16	k has 11 w has 16

5.6 Pointers to functions

Passing pointers to functions is used in event driven programs where the use of callback functions are used to select the code to execute when certain events occurs, used to reduce coupling in a subscription class scheme, used to make code more reusable such as passing a compare function to a sorting function and many other applications. Like arrays, functions are also pointers. You can call a function using a pointer just like if it were a function and you can assign pointers to functions directly since they are compatible types, both pointers to code.

In the following example fp is declared as a pointer to a function. The function it can point to must returns an integer and has as input an interger. Since functions are pointers, the address of the function can be copied into fp directly. Then you can use fp as though it was a function.

```
void (*CallBack)(int k);
int WhichOne;

int square(int n)
{
    return n * n;
}

int main()
```



```

{
    int(*fp)(int n);

    fp = square;

    printf("%d \n", fp(5) );

    getchar();
    return 0;
}

```

The output is

25

In the next example a pointer to a function that compares two integers is used to pass to a general sort function. The sort function does not need to know how to compare the integers as that function is being passed to it.

```

// This sort routine is best for data that is almost totally sorted.
// The running times are O(n^2) on average but O(n) if the array is almost sorted.
// The data is in A[1] to A[n].
void InsertionSort(int A[], int n, bool(*lessthan)(int l, int r))
{
    int p, t, j;

    // A[0] = minimum value.
    for (p = 1; p <= n; p++)
        if (lessthan(A[p], A[0]) ) // if (A[p] < A[0])
            A[0] = A[p];

    for (p = 2; p <= n; p++)
    {
        t = A[p];
        for (j = p; lessthan(t,A[j - 1]); j--) // for (j = p; t < A[j - 1]; j--)
            A[j] = A[j - 1];
        A[j] = t;
    }
}

bool cmp(int a, int b)
{
    return a < b;
}

Int data[11] = { 0,4,7,2,7,9,4,6,8,2,4 };

void main3()
{
    int i;

    printf("Before\n");
    for (i = 1; i <= 10; i++)
        printf("%d, ", data[i]);
    printf("\n");
}

```

```
InsertionSort(data, 10, cmp);

printf("after\n");
for (i = 1; i <= 10; i++)
    printf("%d, ", data[i]);
printf("\n");

getchar();
}
```

The output is:

```
Before
4, 7, 2, 7, 9, 4, 6, 8, 2, 4,
after
2, 2, 4, 4, 4, 6, 7, 7, 8, 9,
```

In the last example the same sort function is used this time to sort a list of objects. The comparison is different but the sort function is the same.

The output is:

sdsd

Chapter 6 do-while and for Loops

6.1 Introduction

All we did to this program was to change the while loop to a do-while loop and added a for loop at the end to display the values for several tax rates.

```
/* This program computes the pay an employee must receive given the
number of hours worked, the pay rate and the overtime rate. It
verifies the input for correct data. It computes the tax rate for
different rates */

#include <stdio.h>
#define FALSE 0
#define TRUE !FALSE

void GetData ( double *HW, double *PR, double *OTR)
{
    int      NeedData = TRUE;

    do
    {
        printf ("Enter the number of hours worked : ");
        scanf( "%lf", HW);
        printf ("\nEnter the pay rate : ");
        scanf( "%lf", PR);
        printf ("\nEnter the over time pay rate : ");
        scanf( "%lf", OTR);
        if ( *HW > 0 && *PR > 0 && *OTR > 0)
            NeedData = FALSE;
    }
    while (NeedData);
}

int main( )
{
    double      HoursWorked, PayRate, TaxRate, GrossPay, Tax,
                NetPay, OverTimePayRate;

    GetData (&HoursWorked, &PayRate, &OverTimePayRate);

    if (HoursWorked > 40)
        GrossPay = 40 * PayRate + (HoursWorked - 40) * OverTimePayRate;
    else
        GrossPay = HoursWorked * PayRate;

    printf("\nThe gross pay is %lf, \n", GrossPay);

    for (TaxRate = 0.1; TaxRate < 0.9; TaxRate = TaxRate + 0.1)
    {
        Tax = GrossPay * TaxRate;
        NetPay = GrossPay - Tax;

        printf("at %3.2lf tax rate the tax is %lf, ", TaxRate, Tax);
```

```

        printf("and the net pay is %lf \n", NetPay);
    }

    if (HoursWorked > 40)
        printf("Worked overtime this week \n");

    return 0;
}

```

The line:

```

do
{
    printf ("Enter the number of hours worked  : ");
    scanf( "%lf", HW);
    printf ("\nEnter the pay rate  : ");
    scanf( "%lf", PR);
    printf ("\nEnter the over time pay rate  : ");
    scanf( "%lf", OTR);
    if ( *HW > 0 && *PR > 0 && *OTR > 0)
        NeedData = FALSE;
}
while (NeedData);

```

loops until the condition "NeedData" is evaluated as false. This loop is guaranteed to execute at least once. In contrast the while loop may never execute if the condition is false initially.

Next we introduced the for loop. This loop is like the while loop but the looping conditions are enclosed in parenthesis. The looping condition has 3 fields separated by semicolons. The first field executes only the first time before entering the loop. The second field is the condition. Before executing the loop this condition is evaluated. If it is true then it executes the loop otherwise the loop is terminated and execution continues after the for loop. The last field executes after executing each loop. This is where you change the looping variable. The for loop structure is useful when one knows ahead of time the number of times the loop will be executed. For example a good application is in counting.

Note that whatever you can do with a for loop you can do with the while loop and the do-while loop. The three loops do exactly the same thing. Which one you use depends on the appropriateness of the application and the your style.

The line:

```

for (TaxRate = 0.1; TaxRate < 0.9; TaxRate = TaxRate + 0.1)
{
    Tax = GrossPay * TaxRate;
    NetPay = GrossPay - Tax;

    printf("at %3.2lf tax rate the tax is %lf, ",TaxRate,Tax);
    printf("and the net pay is %lf \n", NetPay);
}

```

executes the body of the loop 8 times. The first thing that happens is that TaxRate is initialized to 0.1, Next the condition is evaluated. Since $0.1 < 0.9$ the loop is executed. After executing the loop TaxRate is incremented by 0.1. The condition is reevaluated and since $0.2 < 0.9$ the loop is executed again. Then 0.1 is again added to TaxRate and the condition reevaluated. This reoccurs until TaxRate is incremented to the value of 0.9. The condition is then evaluated as false and execution continues after the for loop structure.

The output is:

Enter the number of hours worked : 55

Enter the pay rate : 25

Enter the over time pay rate : 30

The gross pay is 1450.000000,

at 0.10 tax rate the tax is 145.000000, and the net pay is 1305.000000

at 0.20 tax rate the tax is 290.000000, and the net pay is 1160.000000

at 0.30 tax rate the tax is 435.000000, and the net pay is 1015.000000

at 0.40 tax rate the tax is 580.000000, and the net pay is 870.000000

at 0.50 tax rate the tax is 725.000000, and the net pay is 725.000000

at 0.60 tax rate the tax is 870.000000, and the net pay is 580.000000

at 0.70 tax rate the tax is 1015.000000, and the net pay is 435.000000

at 0.80 tax rate the tax is 1160.000000, and the net pay is 290.000000

at 0.90 tax rate the tax is 1305.000000, and the net pay is 145.000000

Worked overtime this week

6.2 Example – Errors in Computations

```
#include "stdio.h"
#include "math.h"

double f(double x)
{
    return x*x;
}

main()
{
    double    sum,a,b,h,error,x;

    a = 10.0;
    b = 11.0;

    printf ("The actual value is %.15lf \n",331.0/3.0);

    for (h = 0.1;h > 0.00000001; h = h / 10.0)
    {
        sum = 0;
        for (x = a; x <= b; x = x + h)
```

```

        sum = sum + (h/2.0)*( f(x) + f(x+h) );
        sum = sum + ( (b-x)/2.0 )*( f(x) + f(x+h) );
        error = fabs(sum - 331.0/3.0);
        printf("h = %.10lf, %.15lf\n",h,sum,error);
    }
}

```

The output is

```

C:\ Select "C:\CD 3\Courses\EGN 3210\notes\Debug\program6_1.exe"
The actual value is 110.33333333333333
h = 0.1000000000, 110.113000000000470 0.2203333333332860
h = 0.0100000000, 110.331148000002410 0.0021853333330914
h = 0.0010000000, 110.333311498061260 0.000021835272065
h = 0.0001000000, 110.333333115254960 0.000000218078370
h = 0.0000100000, 110.333333335327110 0.000000001993783
h = 0.0000010000, 110.333333415883690 0.000000082550358
h = 0.0000001000, 110.333334003862430 0.000000670529104
h = 0.0000000100, 110.333324204311710 0.000009129021620
Press any key to continue

```

6.3 Example Using for Loops

This program prints the letter O on the screen. The function may be used to display several letters.

```

#include <stdio.h>
#define BLOCK "*"

void PrintBlock(int H1, int W1, char *C1, int W2, char *C2, int W3, char *C3)
{
    int r,c;

    for (r = 1; r <= H1; r++)
    {
        for (c = 1; c <= W1; c++)
            printf(C1);
        for (c = 1; c <= W2; c++)
            printf(C2);
        for (c = 1; c <= W3; c++)
            printf(C3);
    }
}

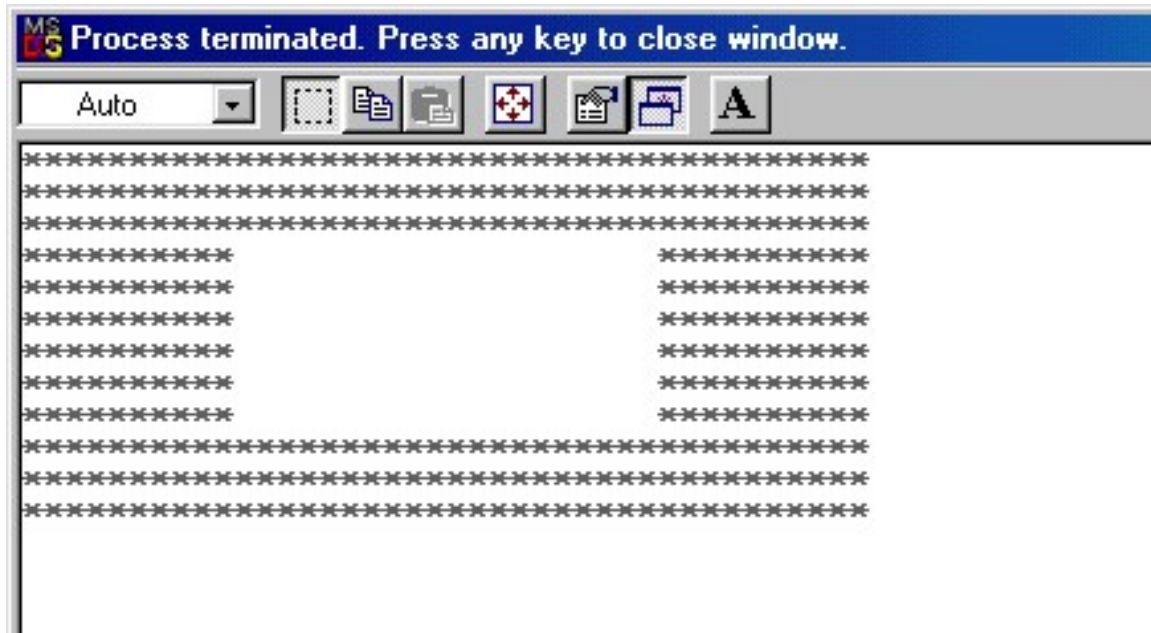
```

```

        printf("\n");
    }

int main( void )
{
    PrintBlock(3,40,"*",0," ",0," ");
    PrintBlock(6,10,"*",20," ",10,"*");
    PrintBlock(3,40,"*",0," ",0," ");
    return 0;
}

```



6.4 Example – Square Root

This function computes the square root of the number entered by the user.

```

// program 6.3

#include    "stdio.h"
#include    "math.h"

#define     FALSE 0
#define     TRUE  !FALSE

#define     TOH    0.001

double my_sqrt(double x)
{
    double    p,po = 1;

```

```

        for (;;)
        {
            p = 0.5 * (po + x / po);

            if (fabs(p - po) < TOH)
                return p;

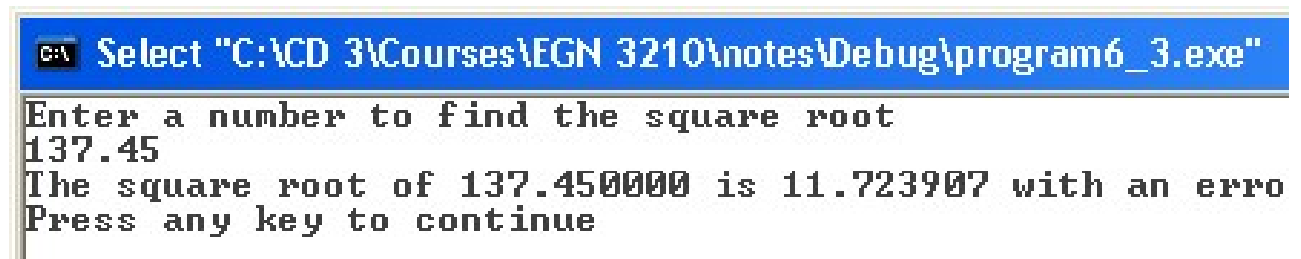
            po = p;
        }

main()
{
    double      x;

    printf("Enter a number to find the square root\n");
    scanf("%lf",&x);
    printf("The square root of %lf is %lf ", x, my_sqrt(x));
    printf("with an error of %.10lf\n",fabs(my_sqrt(x) - sqrt(x)) );
}

```

The output for the number 137.45 is:



The screenshot shows a Windows command prompt window with a blue title bar that reads "Select 'C:\CD 3\Courses\EGN 3210\notes\Debug\program6_3.exe'". The command prompt displays the following text:

```

Enter a number to find the square root
137.45
The square root of 137.450000 is 11.723907 with an erro
Press any key to continue

```

6.5 Example – Root Finding using Newton’s Method

This program uses Newton’s method to find the root of the function $f(x) = 5x^3 + 3x^2 + 6x + 37$.

```

#include      "stdio.h"
#include      "math.h"

#define      FALSE      0
#define      TRUE      !FALSE

#define      N      100
#define      TOH      0.001

double f(double x)
{
    return 5*x*x*x + 3*x*x + 6*x + 37;
}

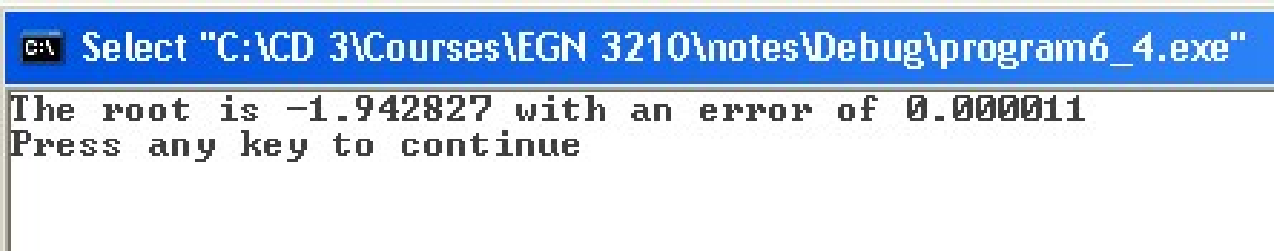
double df(double x)
{
    return 15*x*x + 6*x + 6;
}

```



```
    }  
main()  
{  
    double    p,po = 1;  
    int      i = 1;  
  
    while (i < N)  
    {  
        p = po - f(po) / df(po);  
        if (fabs(p - po) < TOH)  
        {  
            printf("The root is %lf ",p);  
            printf("with an error of %lf\n",fabs(p - po));  
            break;  
        }  
        i++;  
        po = p;  
    }  
  
    if (i == N)  
        printf("Warning output may be incorrect. Check error\n");  
}
```

The output is:



```
C:\ Select "C:\CD 3\Courses\EGN 3210\notes\Debug\program6_4.exe"  
The root is -1.942827 with an error of 0.000011  
Press any key to continue
```

Chapter 7 Sequential Files

7.1 Introduction

(eof(.) EOF,)

A file is a way to store data in the hard drive. All the data in the drive including programs are stored in files. The hard drive is like a large capacity automated filing cabinet. Your program can use files to read and store data. Consider the following program:

```
/* This program creates a file with the name TheData.dat in the current
directory. Then writes to the file the phrase "Hello Word!" and the
phrase "The anser is 45".
*/

#include <stdio.h>
#define FALSE 0
#define TRUE !FALSE

int main( )
{
    FILE *fp;
    int x = 45;

    fp = fopen("TheData.dat", "w");

    if (fp == NULL)
    {
        printf ("Unable to create the file\n");
        return 0;
    }

    fprintf(fp, "Hello World!\n");
    fprintf(fp, "The answer is %4d\n", x);

    fclose(fp);

    printf("I created a file called TheData.dat\n");
    return 0;
}
```

The function to open a file calls the operating system (OS). The OS allocates a block of memory and stores information relating to the file it just opened. This information, called File Control Block (FCB) is of no use to us but we need to supply the OS this information whenever we do any operation on the file. The `fopen ()` function opens a file. It returns the address of the FCB so that we can give it back to the OS later. The function

```
fp = fopen("TheData.dat", "w");
```

opens a file for writing. The filename is `TheData.dat`. The “w” tells the function to open the file for writing. This means that the function will first delete any file with the same name then create a new one. It returns the address of the FCB so we need to receive it in a pointer to FCB type. This pointer is declared by the line:

```
FILE *fp;
```

The segment:

```
if (fp == NULL)
{
    printf ("Unable to create the file\n");
    return 0;
}
```

checks to see that the file was properly opened. If something were to go wrong in opening or creating the file the OS returns a NULL pointer. We need to check that the pointer is not NULL before using the file.

The line:

```
fprintf(fp, "Hello World!\n");
```

prints to the file. Notice that this is the same file version of the `printf()` function that we have use in the past. The first parameter is the pointer to the file. The other parameters are exactly the same as the `printf()` function.

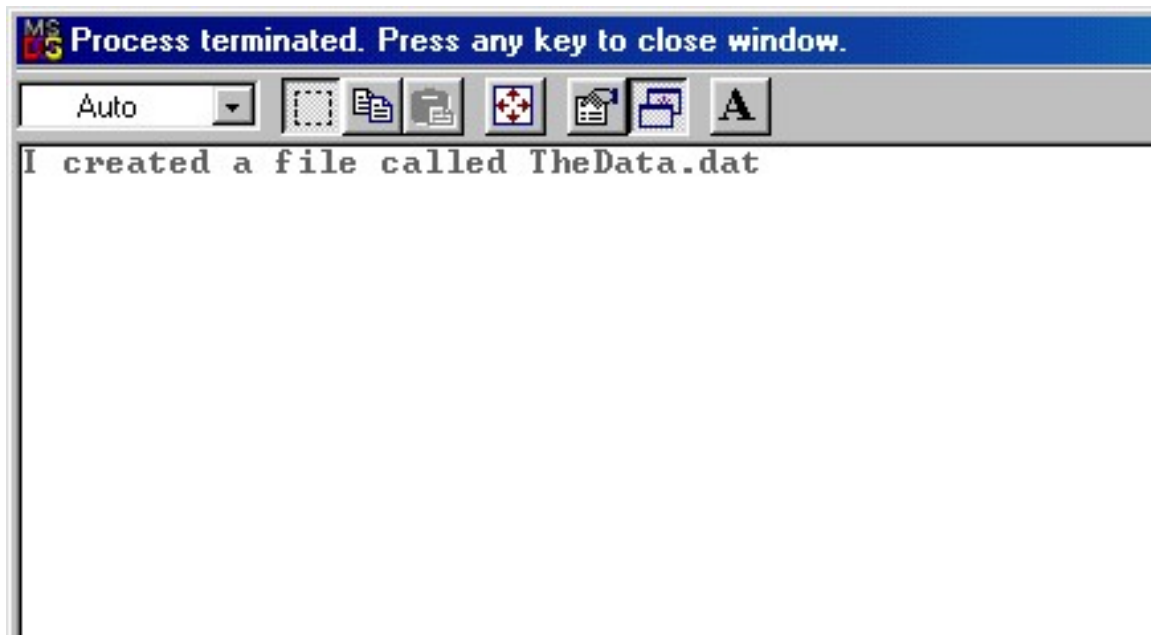
The line:

```
fclose(fp);
```

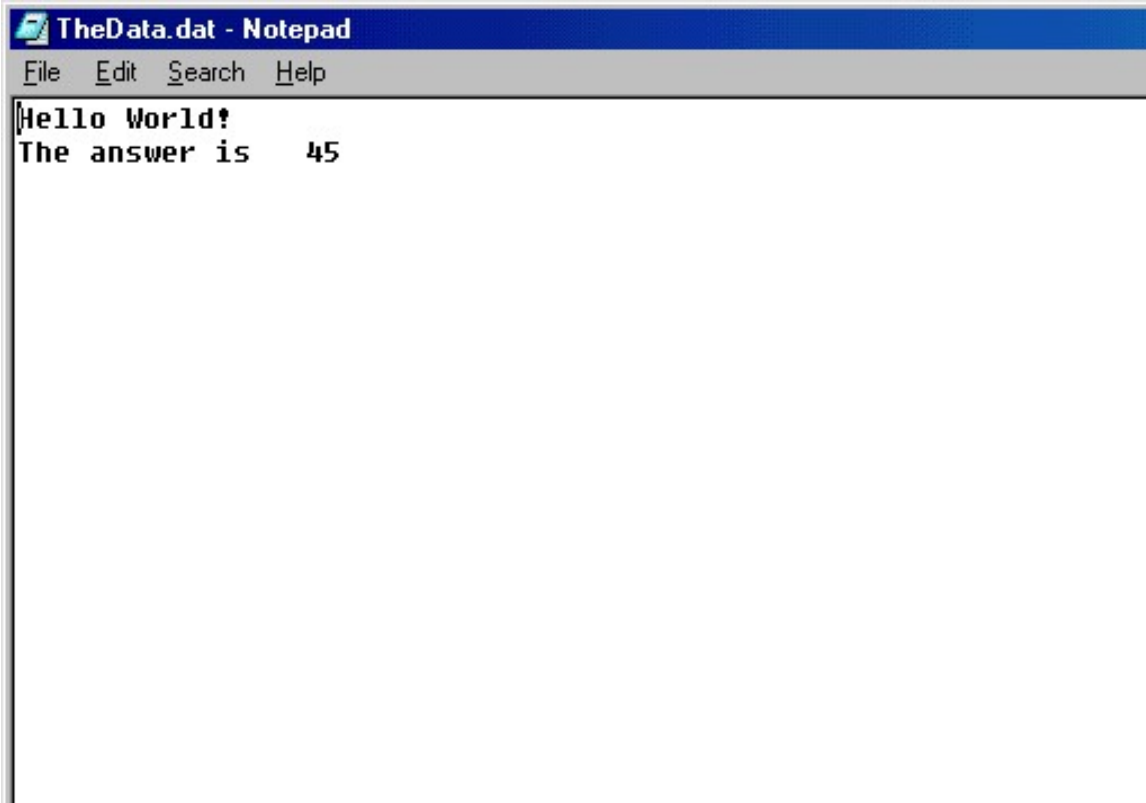
closes the file. If the file is left open the OS will close it for you when the program is done executing. However if the program crashes while the file is open the file will be lost so it is a good idea to close the file as soon as you are done with it.

Just because we are writing to a file does not mean that we cannot also output to the screen as well. The program has some output to the screen so that the user knows what is going on.

The output to the screen is



Using Notepad we can see the file created:



The program to read a file:

```
/* This program reads a file with the name TheData.dat in the current
directory. Then output the contents of the file.
*/

#include <stdio.h>
#define FALSE 0
#define TRUE !FALSE

int main( )
{
    FILE *fp;
    int x,i;
    char str[80];

    fp = fopen("TheData.dat","r");

    if (fp == NULL)
    {
        printf ("Unable to open the file\n");
        return 0;
    }

    printf("The 5 strings were read from the file:\n\n");

    for (i = 1; i <= 5; i++)
```

```

        {
            fscanf(fp, "%s", str);
            printf("%s \n", str);
        }

        fscanf(fp, "%d", &x);
        printf("\nand the number %d was read from the file\n", x);

        fclose(fp);

        return 0;
    }

```

The line:

```
fp = fopen("TheData.dat", "r");
```

opens a file for reading. The file must exist or the `fopen()` function will return a NULL pointer. Since we created this file before we ran this program it exist. We use the file version of `scanf()` that is called `fscanf()`. Like `fprintf` it works exactly the same as `scanf()` but requires the file pointer as the first parameter.

The line:

```
fscanf(fp, "%s", str);
```

reads a string from the file. Note that to `scanf()` a string is defined as all of the ASCII characters between any white space character. A white space character is any nonprintable character like the blank or end of line.

The segment:

```

for (i = 1; i <= 5; i++)
{
    fscanf(fp, "%s", str);
    printf("%s \n", str);
}

```

reads 5 strings and prints them to the screen. Putting this code in a loop saves from having to copy the two lines of code 5 times.

The segment:

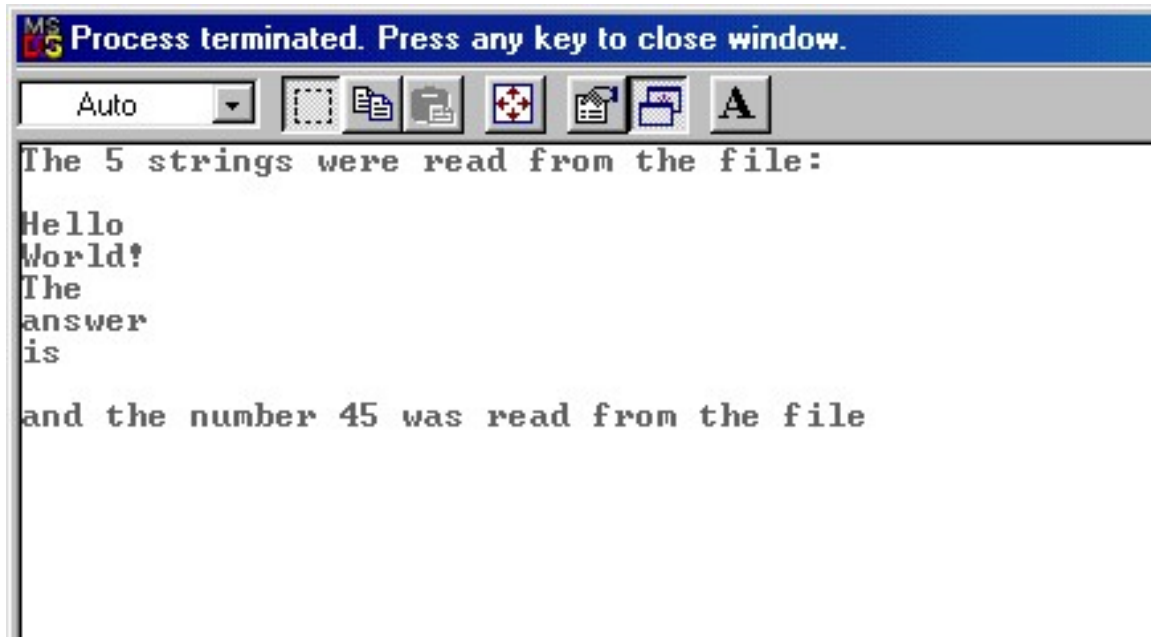
```

fscanf(fp, "%d", &x);
printf("\nand the number %d was read from the file\n", x);

```

reads an integer from the file and prints it. You must make sure that the next sets of characters in the file correspond to digits. Otherwise if the next character in the file is not a digit this character will be converted to integer anyways and you will get garbage returned in the integer variable.

The output is:



7.2 Example – Plotting a Function

The following program plots the function $f(x) = \sin(3x)$.

```
#include    "stdio.h"
#include    "math.h"

#define     FALSE 0
#define     TRUE  !FALSE

double     max,min,k;

double f(double x)
{
    return sin(3 * x);
}

int  v(double y)
{
    return (int)( (y - min)*k );
}

main()
{
    double     a,b,h,x,y;
    int        i,vy,v0,NoOrigin = FALSE;
    FILE       *fp;

    a = -10.0;
    b = 10.0;
    h = .1;

    min = f(a);
```

```
max = f(a);
for (x = a; x <= b; x = x + h)
{
    y = f(x);
    if (y < min)
        min = y;
    if (y > max)
        max = y;
}
k = 40.0 / (max - min);

if (min*max >= 0)
    NoOrigin = TRUE;

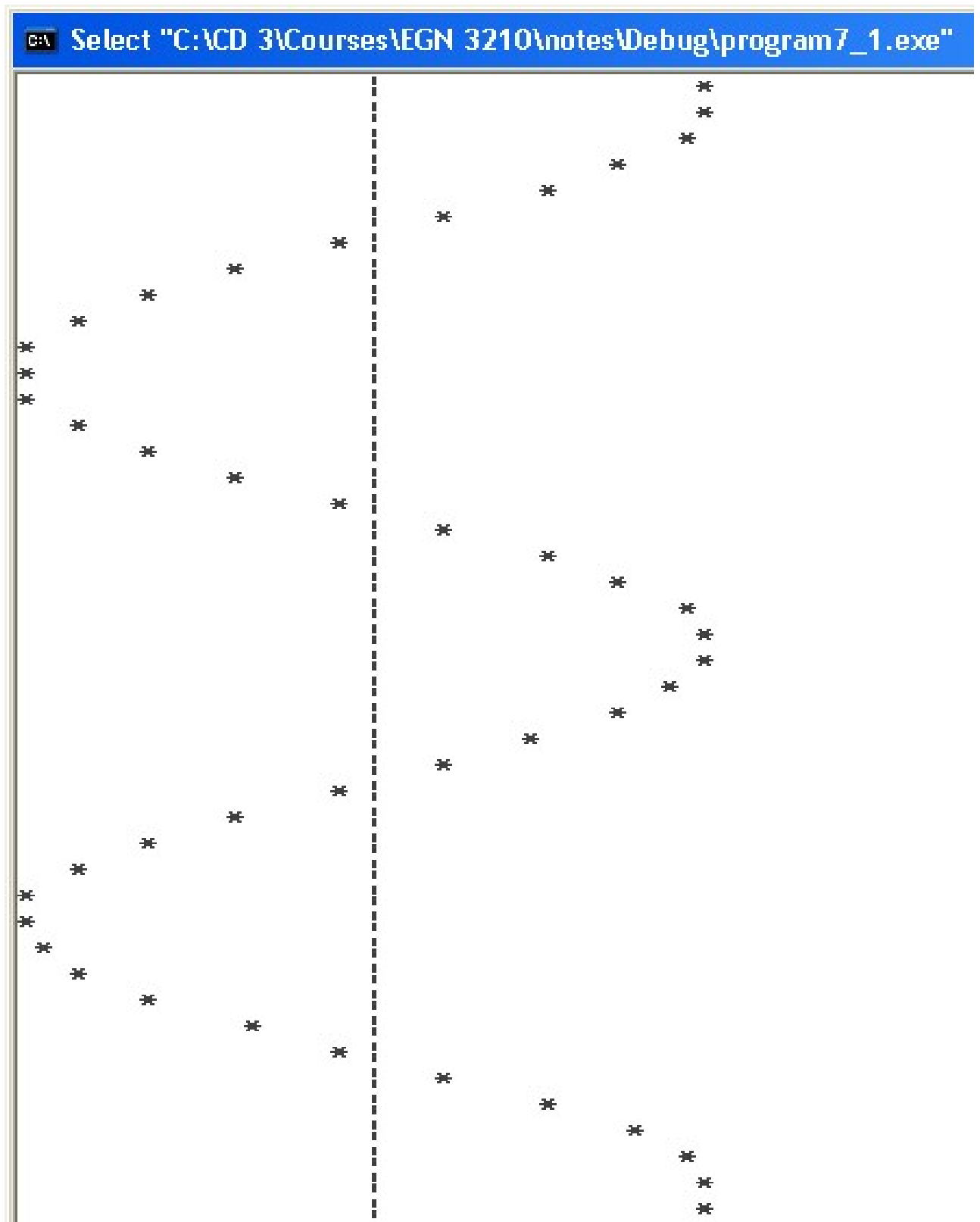
fp = fopen("output.dat", "w");

for (x = a; x <= b; x = x + h)
{
    vy = v(f(x));
    v0 = v(0);
    if (NoOrigin)
    {
        for (i = 0; i < vy; i++)
        {
            fprintf(fp, " ");
            printf(" ");
        }
        fprintf(fp, "*\n");
        printf("*\n");
    }
    else if (v0 < vy)
    {
        for (i = 0; i < v0; i++)
        {
            fprintf(fp, " ");
            printf(" ");
        }
        fprintf(fp, "|");
        printf("|");
        i++;
        for (; i < vy; i++)
        {
            fprintf(fp, " ");
            printf(" ");
        }
        fprintf(fp, "*\n");
        printf("*\n");
    }
    else
    {
        for (i = 0; i < vy; i++)
        {
            fprintf(fp, " ");
            printf(" ");
        }
        fprintf(fp, "*");
        printf("*");
    }
}
```

```
        i++;
        for (; i < v0; i++)
        {
            fprintf(fp, " ");
            printf(" ");
        }
        if (vy != v0)
        {
            fprintf(fp, "|\\n");
            printf("|\\n");
        }
        else
        {
            fprintf(fp, "\\n");
            printf("\\n");
        }
    }

    fclose(fp);
}
```

Part of the output is shown.



7.3 Example – Plotting a Function

This program plots the function $f(x) = \sin(3x)$ but does not plot the 0 axis line.

```
#include "stdio.h"
#include "math.h"

#define FALSE 0
#define TRUE !FALSE

double max,min,k;

double f(double x)
{
    return sin(3*x);
}

int v(double y)
{
    return (int)( (y - min)*k );
}

main()
{
    double a,b,h,x,y;
    int i,vy;
    FILE *fp;

    a = -10.0;
    b = 10.0;
    h = .1;

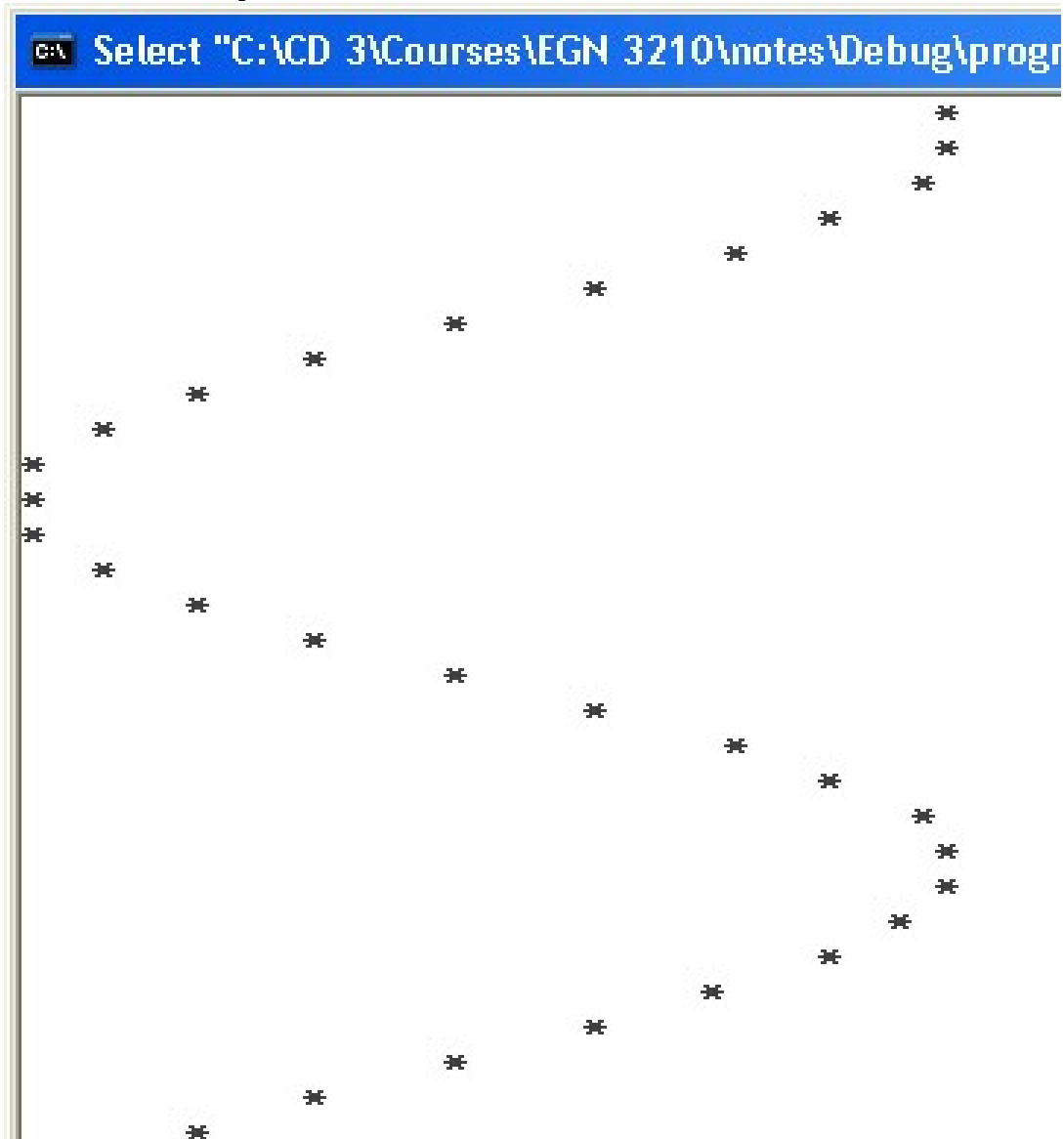
    min = f(a);
    max = f(a);
    for (x = a; x <= b; x = x + h)
    {
        y = f(x);
        if (y < min)
            min = y;
        if (y > max)
            max = y;
    }
    k = 40.0 / (max - min);

    fp = fopen("output.dat","w");

    for (x = a; x <= b; x = x + h)
    {
        vy = v(f(x));
        for (i = 0; i < vy; i++)
        {
            fprintf(fp," ");
            printf(" ");
        }
        fprintf(fp,"*\n");
        printf("*\n");
    }
}
```

```
    }  
  
    fclose(fp);  
}
```

Part of the output is shown:



7.4 Random Access Files

Random access file is a way to read data at an arbitrary location within the file without reading all of the data in front of it as is the case with sequential files.

Program7-3.h

```
struct date
{
    char        month[10];
    int         day;
    int         year;
};

struct record
{
    char        name[80];
    double      pay;
    date        DOB;
};
```

```
#include "program7-3.h"

record TheRecords[100] =
{
    {"bill",      42000,      {"July",      23,      1978} },
    {"Fred",      12000,      {"May",       29,      1956} },
    {"Tom",       76000,      {"April",    5,      1932} },
    {"Sam",       123000,     {"March",   12,      1981} },
    {"Carlos",    56000,      {"June",    25,      1965} },
    {"Ali",       32500,      {"Decamber", 19,      1987} },
    {"Chen",      26750,      {"October", 23,      1977} },
    {"Gary",      76800,      {"May",     8,      1983} },
    {"Anna",     145500,      {"August",  14,      1964} } };

int main()
{
    fp = fopen("TheRecords.bin","wb");
    if (pf == NULL)
    {
        printf("Error opening the file\n");
        return;
    }

    for (i = 0; i < 7; i++)
        fwrite( &(TheRocord[i]), sizeof(record), 1, fp);

    fclose(fp);
}
```

```
#include "program7-3.h"

int main()
{
    printf("ente record number to print : ");
    scanf("%d",&n);

    fseek(fp, sizeof(TheRecord) * (n - 1), SEEK_SET);
    fread(&TheRocord, sizeof(TheRecord), 1, fp);
    printf("Name:%-10s, DOB:%10s/%2d/%2d Pay $%8.2lf\n",
        TheRocord .name,
        TheRocord.DOB.month,
        TheRocord .DOB.day,
        TheRocord.DOB.year,
        TheRocord .pay);
}
```

7.5 File encryption

The following code will encrypt a file with a code. The first 10 bytes will be XOR with the encryption byte. Since most files use a header to indicate the type of file as well as other data in the header and since this header is the first few bytes of a file, we only need to encrypt those bytes to make the file unreadable to others.

The user will enter a byte in decimal for the encryption. The first 10 bytes will then be read. Each of those 10 bytes will be exclusive or (XOR) with the encryption byte entered. Then those 10 bytes are stored back into the front of the file where they were read from. The XOR operation has the effect of flipping the bits. This is why when you run the program a second time with the same byte code the bits are flipped again returning them to there original state.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

```
// program 7.3

#include      "stdio.h"
#include      "stdlib.h"
```

```
#define FALSE 0
#define TRUE !FALSE

char    buffer[10];
FILE    *fp;
int     i;
int     icode;
char    code;

void
main( int   argc,
      char  *argv[])
{
    if (argc < 3)
    {
        printf("format: code bill.dat 100 \n");
        printf("Where bill.dat is the file to code and 100 is the code (1 to 126)
\n");
        return;
    }

    code = (char)atoi(argv[2]);

    if ( (code > 126) || (code < 1))
    {
        printf("code is out of range\n");
        printf("format: code bill.dat 100 \n");
        printf("Where bill.dat is the file to code and 100 is the code (1 to 126)
\n");
        return;
    }

    fp = fopen(argv[1],"r+");
    if (fp == NULL)
        printf("Error reading %s \n",argv[1]);
    else
    {
        fseek(fp,0,SEEK_SET);
        fread(buffer,sizeof(buffer),1,fp);

        for (i = 0; i < sizeof(buffer) - 1; i++)
            buffer[i] = buffer[i] ^ code;

        fseek(fp,0,SEEK_SET);
        fwrite(buffer,sizeof(buffer),1,fp);
        printf("%s was modified!\n",argv[1]);
    }
}
```

```
        }  
    fclose(fp);  
}
```

Chapter 8 Arrays

8.1 Introduction

Arrays are a group of variables of the same type stored consecutive in memory. The array variable has a single label and an index is used to distinguish between the different components. For example an array of 10 integers may be declared as:

```
int    x[10];
```

The variable `x` is not a single integer but rather 10 consecutive integers. We identify the 10 integers by the subscript. For example the 7th integer in the array can be accessed using `x[6]`. So to store the number 37 into the 7th integer we can use:

```
x[6] = 37;
```

The array components are index from 0 to `n-1` where the array has `n` components. So the array declared above has components `x[0]`, `x[1]`, ..., `x[9]`.

The following program shows an example using array. Because the components in an array are addressed using an index we can access all components using a variable for the index and a for loop. So instead of using:

```
table[0] = 0;
table[1] = 0;
table[2] = 0;
table[3] = 0;
table[4] = 0;
table[5] = 0;
```

we can use:

```
for (i = 0; i < 6; i++)
    table[i] = 0;
```

This is especially useful if you consider the array having 10,000 components. We will need 10,000 statements and 10,000 variables with 10,000 unique names if we did not use arrays. Instead, when using arrays the for loop will simply go from 0 to 10,000 but still remain being 2 lines of code and 1 variable name.

The following program is an example of an application that uses arrays.

```

/* This program uses arrays to count the number of times each face of
a single dice is selected with N rolls. A random number generator is
used to simulate the rolling of the dice. */

#include <stdio.h>
#include "stdlib.h"

#define FALSE 0
#define TRUE !FALSE
#define N 100000

int main( )
{
    int table[6],count;
    long int i, j;
    double scale;

    for (i = 0; i < 6; i++)
        table[i] = 0;

    for (i = 0; i < N; i++)
        table[rand() % 6]++;

    printf("Roll a dice %d times \n\n",N);

    scale = (N/6.0) / (40);
    for (i = 0; i < 6; i++)
    {
        printf(" %d : ",table[i]);

        count = int(table[i] / scale);
        for (j = 0; j < count; j++)
            printf("*");

        printf("\n");
    }

    return 0;
}

```

The line:

```
int table[6],count;
```

creates an array consisting of 6 integers. An array is a data structure that has several locations associated with it as opposed to just 1 as in the case of the basic data types such as an integer. You can think of an array as:

0	1	2	3	4	5

An array allows you to give a single variable name to a collection of locations. For example in this line `table` refers to all 6 integers. To address each individual integer we need to specify which of the 6 integers we are referring to. We use indexes. For example when refereeing to the 3rd integer we use the index 2. The first index is always 0. This means that the 1st location is at index 0 and the 2nd is at index 1 and so on. So the syntax is `table[index]` where index is an integer.

Example the 3rd integer is reverred to as `table[2]` while the 1st location is `table[0]`.

If you execute the following statements:

```
table[0] = 37;
table[1] = 42;
table[2] = 67;
table[3] = 89;
table[4] = 103;
table[5] = 2;
```

then the array will look like:

0	1	2	3	4	5
37	42	67	89	103	2

If you execute the following statement:

```
printf("Component number %d in the table is %d \n", 2, table[2]);
```

then the output will be:

```
Component number 2 in the table is 67
```

The line:

```
table[rand() % 6]++;
```

calls function `rand()`. This function returns a random integer in the range 0 to some large number that is machine dependent.

The `%` symbol is the modulus function. The modulus is the remainder after division. For example 11 mod 4 written in C as `11 % 4` equals 2 because `11 / 4` equals 2 remainder 3. The modulus is used here to limit the random numbers to the range 0 to 5.

Once the random number generator returns a large number then the modulus wraps the number around the range 0 to 5. At this point resulting modulus is used as an index into the array. The proper array component is the incremented indicating that we have seen one more outcome of that value. For example if the `rand` function returns a 1857 then `1857 % 6` is 3. This simulated rolling a 3 so `table[3]` is incremented.

The statement

```
for (i = 0; i < N; i++)
    table[rand() % 6]++;
```

simulates rolling a dice N times. Since the statement that simulates the rolling of a dice is put into a loop and the loop cycles N times it simulates rolling a dice N times. After this statement executes the array calls

table will have tallies of the number of times each side was rolled. That is table[0] contains the number of times a face of 0 was rolled, table[1] contains the number times a face of 1 was rolled and so on.

In order to draw a graph using asterisk we need to scale the graph. We compute a scale relative to the size of the screen and N.

The line:

```
scale = (N/6.0) / (40);
```

Computes the scaling factor. Note the N / 6 is the average number in the array and the 40 is the maximum window size.

The segment:

```
count = int(table[i] / scale);
for (j = 0; j < count; j++)
    printf("*");
```

plots the asterisk. It first computes the number asterisk that must printed to represent the number. It uses the scaling factor. Next the loop prints the correct number of asterisk.

The program was compiled and run 5 times with values for N = 10 to 100,000:

Roll a dice 10 times

```
3 : *****
2 : *****
1 : *****
2 : *****
1 : *****
1 : *****
```

Roll a dice 100 times

```
20 : *****
19 : *****
14 : *****
16 : *****
14 : *****
17 : *****
```

Roll a dice 1000 times

```
157 : *****
177 : *****
157 : *****
161 : *****
165 : *****
183 : *****
```

Roll a dice 10000 times

```
1560 : *****
1706 : *****
1684 : *****
1679 : *****
1653 : *****
1718 : *****
```

Roll a dice 100000 times

```
16412 : *****
16932 : *****
16741 : *****
16497 : *****
16808 : *****
16610 : *****
```

8.2 Arrays and Pointers

Arrays are actually pointers. An array is simply a pointer to the block of memory that is allocated to the array. In the declaration

```
int x[10], *p;
```

Here the variable `x` is a pointer to a block of memory. The block of memory consists of 10 consecutive integers (40 bytes). The variable `x` itself is a constant pointer; it can not point to some other memory location like another array. In contrast the variable `p` is a pointer. When declared it does not point to any thing, it simply has the capability to store an address. The variable `x` in turn points to the 10 integer block.

The following are all equivalent:

```
p = x;          p = x;          p = x + 6;
p[6] = 37;      *(p + 6) = 37;    x[6] = 37;      *(x + 6) = 37;    p[0] = 37;
```

8.3 Multidimensional Arrays

An array may have more than 1 dimension. For example in a 2 dimensional array 2 indices must be specified. Consider the following declaration:

```
int y[10][5], x[50];
```

The variable `y` is a 2 dimensional array with size 10 by 5. This can be thought of as a square area consisting of 10 rows by 5 columns or 10 columns by 5 rows depending on how you interpret it. In reality it is simply a block of $10 \times 5 = 50$ consecutive integers just like `x` above. The only difference between `x` and `y` is the way you address the individual components; in `y` you need 2 indices as opposed to only 1 in `x`. In both cases the compiler will give you the pointer version. For example if I write:

```
y[7][3] = 37;
```

the compiler will translate this to

```
* (y + 7 * 5 + 3) = 37;
```

and then to

```
* (y + 38) = 37;
```

If I write:

```
x[38] = 37;
```

The compiler will translate this to:

```
(x + 38) = 37;
```

So at the end the only difference between a 1 and a 2 dimensional array is the way we address the components.

We can have as many dimension as we want, some limits apply. For example a 3 dimensional array of 3 by 5 by 10 of doubles will be declared as follows:

```
double      z[3][5][10];
```

A particular usage may be:

```
z[1][3][7] = 37;
```

Again this array is simply a block of $3*5*10 = 150$ consecutive doubles.

8.4 Ex – Parsing using a Finite State Machine (FSM) algorithm

A finite state machine also known as a finite state acceptor is a theoretical machine or algorithm that uses a table to determine its current state and its action. The system is described with states. Then depending on the current state and on the input you go to a new state and perform some action. The operation of parsing is to convert a string into its meaningful data. For example to convert a string into a real number. In this example we are going to convert a string into a real number or double. We decompose the problem into 4 states:

State 0: Represents the starting and ending state. It says we have not yet started and are still expecting the first meaningful data in the number.

State 1: In this state we have seen the sign of the value, if for example one put a “-” or “+” sign in front of the number.

State 2: In this state the algorithm is seeing the digits to the left of the decimal. It must multiply the sum by 10 and add the value of the new digit.

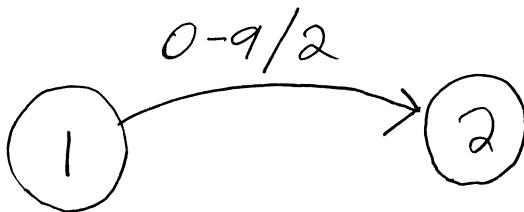
State 3: In this state the algorithm is seeing the digits to the right of the decimal. It must multiply the denominator by 10 and add the next digit divided by the denominator to the sum.

At the end the state goes back to 0 and the function quits. Note the sign, decimal and digits after the decimal are optional.

The list of actions are:

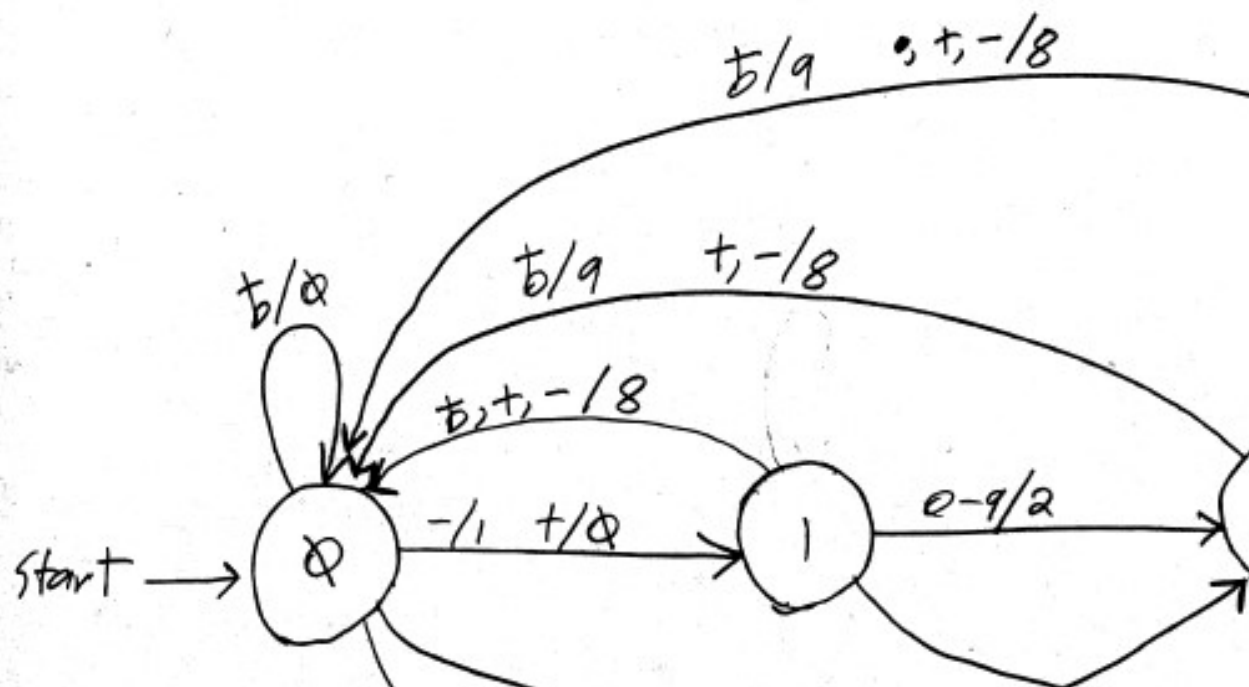
Action 0: Do nothing
 Action 1: Change the sign to -1
 Action 2: $\text{sum} = \text{sum} * 10 + (\text{ch} - '0')$
 Action 3: $\text{den} = \text{den} * 10$
 $\text{Sum} = \text{sum} + (\text{ch} - '0') / \text{den};$
 Action 8: $\text{sum} = 0$
 Error
 stop
 Action 9: $\text{sum} = \text{sum} * \text{sign}$
 stop

For the graphical view of the FSM we use



To mean that if the state is 1 and the next input is a digit then go to state 2 and perform action 2.

The following is the state diagram for our FSM.



The following table represents the FSM in the diagram. We use the notation X/Y to mean “go to state X and perform action Y.”

	-	+	digit	period	white space
State 0	1/1	1/0	2/2	3/0	0/0
State 1	0/8	0/8	2/2	3/0	0/8
State 2	0/8	0/8	2/2	3/0	0/9
State 3	0/8	0/8	3/3	0/8	0/9

The following is the code:

```
#include "stdio.h"
#include "conio.h"
#include "string.h"

#define MINUS 0
#define PLUS 1
#define DIGIT 2
#define DOT 3
#define WHITESPACE 4

#define FALSE 0
#define TRUE !FALSE

#define DEBUG TRUE

int toToken(int ch)
{
    int token;

    if (ch == '-')
        token = MINUS;
    else if (ch == '+')
        token = PLUS;
    else if (ch == '.')
        token = DOT;
    else if (ch >= '0' && ch <= '9')
        token = DIGIT;
    else
        token = WHITESPACE;

    return token;
}

double
ascii_to_double(char str[80])
{
    struct entry
    {
        int state;
        int action;
    };
```



```

    struct entry FSAtable[4][5] = {
        {{1,1},{1,0},{2,2},{3,0},{0,0}},
        {{0,8},{0,8},{2,2},{3,0},{0,8}},
        {{0,8},{0,8},{2,2},{3,0},{0,9}},
        {{0,8},{0,8},{3,3},{0,8},{0,9}}
    };

    int      state = 0,
            sign = 1;
    double   sum = 0,
            dem = 1;

    int      done = FALSE,
            i = 0,
            len,
            ch,
            NextState,
            Action,
            token;

    len = strlen(str);

    while (!done)
    {
        ch = str[i++];
        token = toToken(ch);

        NextState = FSAtable[state][token].state;
        Action = FSAtable[state][token].action;

        if (DEBUG)
            printf("got %c so went from state %d to %d with action\n",
                ch,state,NextState,Action);

        state = NextState;

        switch (Action)
        {
            case 0:
                break;

            case 1:
                sign = -1;
                break;

            case 2:
                sum = (sum * 10) + (ch - '0');
                break;
        }
    }

```

```
        case 3:
            dem = dem * 10;
            sum = sum + (ch - '0') / dem;
            break;

        case 8:
            sum = 0;
            printf("Error in ascii_to_double\n");
            printf("Can not convert %s to a double\n",str);
            done = TRUE;
            break;

        case 9:
            sum = sum * sign;
            done = TRUE;
            break;
    }

    if (i > len)
        done = TRUE;
    }

    return sum;
}

void main()
{
    char  str[80];

    printf("Enter a string to convert: ");
    gets(str); // includes all of the string including spaces until the new line.
    printf("The value of \"%s\" is %lf\n", str, ascii_to_double(str) );
}
```

Chapter 9 Structures

9.1 Introduction

Structures are another form of storage like the array. The difference between an array and a structure is the following:

In an array all components are of the same type where in a structure you can have different types.

In an array the components are identified by a number as in `table[37]` where in a structure we refer to the individual components by name.

Finally in an array we declare the array for each variable while in structure we declare a structure first then use our structure declaration as a new type and declare variables of this type.

Consider that I want to store the date 7/23/2003 into a variable. I can use an array or a structure. First the array implementation will be made then the structures.

<pre>int date[3]; date [0] = 7; date [1] = 23; date [2] = 2003;</pre>	<pre>Struct datatype { int month; int day; int year; }; struct datatype date; date.month = 7; date.day = 23; date.year = 2003;</pre>
Array implementation	Structure implementation

In both cases the variable `date` holds 3 integers representing the date. With arrays one must know that component 0 corresponds to the month, component 1 to the day and component 2 to the year where with the structure the descriptive field name helps the reader know what type of data the field holds.

Now let's assume that you want to store the month as a string so that it reads "July" instead of 7. With the array implementation we cannot do it since all array components must be of the same type. We will have to make the day and year a string as well. With a structure this can be done.

```

struct    datatype
{
    char    month[10];
    int     day;
    int     year;
};

struct datatype    date;

strcpy (date.month, "July");
date.day = 23;
date.year = 2003;

```

Remember that to copy a string to a variable we need to copy each component. The function `strcpy` does this. If we use

```
Date.month = "July"
```

we are copying only its pointer and `date.month` may change without the variable `date` being touched.

In the following program we will maintain an array of structures storing employee's data.

```

#include "stdafx.h"

struct date
{
    char    month[10];
    int     day;
    int     year;
};

struct    record
{
    char    name[80];
    double   pay;
    date    DOB;
};

record    TheRecords[100] =
{ {"bill",    42000,    {"July",    23,    1978} },
  {"Fred",    12000,    {"May",    29,
1956} },
  {"Tom",    76000,    {"April",    5,
1932} },

```

```
        {"Sam",          123000,    {"March",      12,
1981} },
        {"Carlos",      56000,     {"June",       25,
1965} },
        {"Ali",         32500,     {"Decamber",19,
1987} },
        {"Chen",        26750,     {"October",   23,
1977} },
        {"Gary",        76800,     {"May",       8,
1983} },
        {"Anna",        145500,    {"August",   14,
1964} } }];

void printrecords(record r[100], int n)
{
    int i;

    for (i = 0; i < n; i++)
        printf("Name:%-10s, DOB:%10s/%2d/%2d Pay
$%8.2lf\n",
                r[i].name,

                r[i].DOB.month,

                r[i].DOB.day,

                r[i].DOB.year,

                r[i].pay);
}

void get_new_record(int i)
{
    printf("enter person's name\n");
    scanf("%s",TheRecords[i].name);

    printf("enter person's pay\n");
    scanf("%lf", &(TheRecords[i].pay) );

    printf("enter person's DOB month\n");
    scanf("%s",TheRecords[i].DOB.month);

    printf("enter person's DOB day\n");
    scanf("%d", &(TheRecords[i].DOB.day) );

    printf("enter person's DOB year\n");
    scanf("%d", &(TheRecords[i].DOB.year) );
}
```

```

    }

int main()
{
    printrecords(TheRecords, 9);
    get_new_record(9);
    printrecords(TheRecords, 10);
    return 0;
}

```

9.2 Advanced Structures

Advanced structures are a C++ feature that allows one to put functions as field elements into a structure. Consider the datatype from before.

```

Struct    datatype
{
    int      month;
    int      day;
    int      year;
};

struct datatype    date;

date.month = 7;
date.day = 23;
date.year = 2003;

```

Now we want to add a function to this structure that will initialize the structure and another function to print the structure. We may then have:

```

Struct    datatype
{
    char      month[12];
    int      day;
    int      year;

    void      init(char m, int d, int y)
    {
        strcpy (month, m);
        day = d;
        year = y;
    }

    void      print()
    {

```

```

        printf("%10s/%2d/%2d",month, day, year);
    }
};

struct datatype    date;

```

Now to initialize the structure instead of doing:

```

strcpy(date.month, "July");
date.day = 23;
date.year = 2003;

```

We now simply use:

```
date.init("July", 23, 2003);
```

And to print a the structure we use:

```
date.print();
```

This reduces the effort on using the structure but more importantly the user of the structure does not need to worry about how to initialize or point the data in the structure since a function to do this is included in the structure.

Note in the following program that since struct DOB in part of struct records we simply call DOB's init and print function from the records init and print function.

The following function is similar to the previous one but uses the included functions.

```

#include "stdio.h"
#include "string.h"

struct date
{
    char    month[10];
    int     day;
    int     year;
    void     init(char *m, int d, int y)
    {
        strcpy (month, m);
        day = d;
        year = y;
    }

    void     print()
    {
        printf("%10s/%2d/%2d",month, day, year);
    }
}

```

```

};

struct    record
{
    char        name[80];
    double      pay;
    date        DOB;
    void        init(char *n, double p, char *m, int d, int
y)
        {
            strcpy (name, n);
            pay = p;
            DOB.init(m,d,y);
        }

    void        print()
        {
            printf("%10s $%10.2lf ",name, pay);
            DOB.print();
            printf("\n");
        }
};

record    TheRecords[100] =
    { {"bill", 42000, {"July", 23, 1978} },
      {"Fred", 12000, {"May", 19,
1956} },
      {"Tom", 76000, {"April", 5,
1932} },
      {"Sam", 123000, {"March", 12,
1981} },
      {"Carlos",56000, {"June", 25,
1965} },
      {"Ali", 32500, {"Decamber",
19, 1987} },
      {"Chen", 26750, {"October", 23,
1977} },
      {"Gary", 76800, {"May",
8, 1983} },
      {"Anna", 145500, {"August", 14,
1964} } };

void printrecords(record r[100], int n)
{
    int i;

    for (i = 0; i < n; i++)

```



```
        r[i].print();
    }

void get_new_record(int i)
{
    char        name[80],
                month[12];
    int         day,
                year;
    double      pay;

    printf("enter person's name\n");
    scanf("%s",name);

    printf("enter person's pay\n");
    scanf("%lf", &pay );

    printf("enter person's DOB month\n");
    scanf("%s",month);

    printf("enter person's DOB day\n");
    scanf("%d", &day );

    printf("enter person's DOB year\n");
    scanf("%d", &year );

    TheRecords[i].init(name,pay,month,day,year);
}

int main()
{
    printrecords(TheRecords,9);
    get_new_record(9);
    printrecords(TheRecords,10);
    return 0;
}
```

Chapter 10 Classes

10.1 Introduction

A class is basically a structure with the following differences.

- A class can select certain members to be private.
- A class allows for inheritance and polymorphism.
- A class may contain a constructor and destructor that allows initialization and termination “cleanup” automatically.

Like a structure a class is only a type definition. It does not have any memory associated with it. An object is an instantiation of a class. Objects are variables and have memory associated with them. Consider the following program:

```
/* This program declares a simple class. It uses constructors and
destructors. The age and names are declared private. The constructor
is used to initialize the class while the destructor is used to say
bye. */

#include    "string.h"
#include    "stdio.h"

class person
{
    private:
        int    age;
        char    name[80];
    public:
        person(int a, char n[80]);
        ~person();
        void    display ();
};

person::person (int a, char n[80])
{
    age = a;
    strcpy(name, n);
}

person::~~person()
{
    printf ("Mr. %s says Bye\n", name);
}

void person::display()
{
    printf(" %s is %d years old\n", name, age);
}

void main()
{
    person    a(26,  "bill"), b(32, "fred"),*p;

    a.display();
    b.display();

    p = new person (16, "anna");
    p->display();
    delete p;

    //    a.age = 5;          not legal.
}
```

The class definition:

```
class person
{
    private:
        int    age;
        char   name[80];
    public:
        person(int a, char n[80]);
        ~person();
        void   display ();
};
```

is the prototype of the class. It declares the members of the class and there access restrictions. In this class there are 5 members. Two data members declared as private and 3 function members declared as public.

Since the data members are declared as private only the 3 member functions declared inside of the class can access the private elements.

The line:

```
//    a.age = 5;           not legal.
```

Is illegal because age is a private member of the object and only members within the class can access it. This is why it is commented out.

The line:

```
    person    (int a, char n[80]);
```

is the declaration of the constructor. A constructor is a member function that is executed when the object is instantiated (created). This function is used to initialize the object before it is used. Note that this function is executed automatically when the object is declared and the responsibility is not left to the programmer to initialize the object.

The line:

```
    ~person();
```

is the destructor. The destructor is executed automatically when the object is destroyed. This function is used to execute any code that must be run before the object is destroyed. For example if the object has allocated dynamic memory then this function can return the dynamic memory to the operating system. And like the constructor this function executed automatically taking away the responsibility from the programmer to execute it.

The line:

```
void display ();
```

is a member function to display the data. Since the data is private a member function must display any data that we want to be displayed.

The function declaration:

```
person::person (int a, char n[80])
{
    age = a;
    strcpy(name, n);
}
```

declares the constructor. This function received data that is used to initialize the private elements of the object. Note that name is an array and therefore one must use the strcpy() function or one will simply copy the address of the array.

The

```
person::person( ... )
```

is the scope resolution operator. It says that even though person is being declared out side of the class it belongs to class person. The actual operator is :: and it ties the function to the class.

The output will be



```
C:\ Select "C:\Documents and Settings\Fernando\Desktop\Debug\programa...
Mr. bill says Hi
Mr. fred says Hi
  bill is 26 years old
  fred is 32 years old
Mr. anna says Hi
  anna is 16 years old
Mr. anna says Bye
Mr. fred says Bye
Mr. bill says Bye
```

Notice the execution of the constructor and destructors is automatic.

10.2 More on Classes

The C++ language also has some new features that enhance the readability of your code. In the next program pass by reference (without using pointers) and function and operator overloading are shown as well as friend functions.

Consider the following program:

```
#include "iostream.h"
#include "string.h"

class Person
{
private:
    int      age;
    char     *name;
public:
    Person(int a, char *s);
    ~Person( );
    void     display( );
    void     assign(int a, char *s);
    void     operator = (Person &other_object);
    friend   void show_older(Person &a, Person &b);
};

Person::Person( int a, char *s)
{
    cout << "Creating " << s << "\n";
    age = a;
    name = new char[80];
    strcpy (name,s);
}

Person::~~Person()
{
    cout << "Deleting " << name << "\n";
    delete name;
    name = NULL;
}

void
Person::assign( int a, char *s)
{
    age = a;
    if (name == NULL)
        strcpy (name,s);
}

void
Person::display()
{
    cout << name << " is " << age << " years old \n";
}
```

```

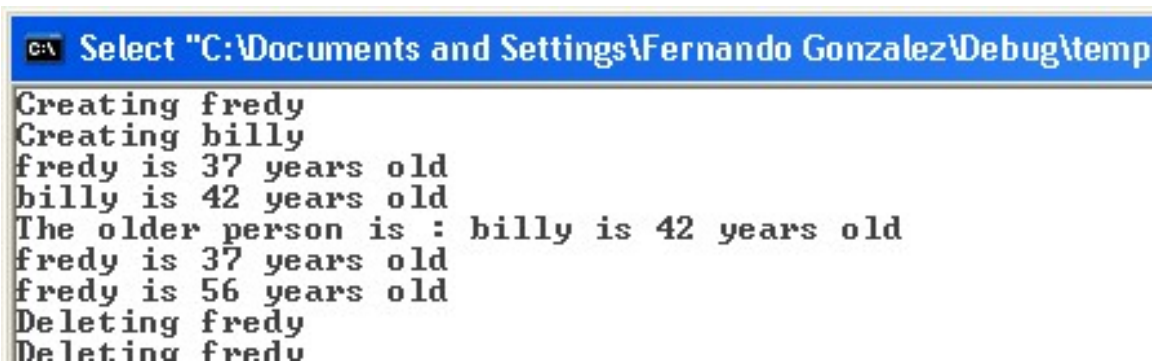
void
Person:: operator = (Person &other_object)
{
    age = other_object.age;
    strcpy(name,other_object.name);
}

void
show_older(Person &a, Person &b)    // I am a friend of
Person
{                                  // so I can access private
data
    cout << "The older person is : ";
    if (a.age > b.age)
        a.display();
    else
        b.display();
}

void
main()
{
    Person fred(37 ,"fredy"), bill(42, "billy");
    fred.display();
    bill.display();
    show_older(bill,fred);
    bill = fred;           // Our "=", not the usual one.
    bill.display();
    bill.assign(56, "new guy");
    bill.display();
}

```

The output is:



```

C:\ Select "C:\Documents and Settings\Fernando Gonzalez\Debug\temp"
Creating fredy
Creating billy
fredy is 37 years old
billy is 42 years old
The older person is : billy is 42 years old
fredy is 37 years old
fredy is 56 years old
Deleting fredy
Deleting fredy

```

The line

```
name = new char[80];
```

in the constructor ask the operating system for memory to implements its string. The constructor accepts an integer and a pointer to a string (an array). The constructor creates a new array by asking the operating system to give it sufficient memory to implement the array (dynamic memory allocation). Then each component in the input string is copied to the newly allocated string.

The line

```
delete name;
```

in the destructor returns the memory back to the operating system.

The member function called assign allows the user to assign values to the object after instantiation. Notice it does not get new memory since the object already has memory.

The line

```
void display( );
```

declares a member function called display(). It is the only way to view the private elements of the object.

The member function

```
void  
Person:: operator = (Person &other_object)  
{  
    age = other_object.age;  
    strcpy(name,other_object.name);  
}
```

demonstrates the use of operator overloading and the new pass by reference in C++.

First the pass by reference. By putting a & sign in front of an argument in the declaration of a function it tells the compiler that that variable is to be passed by reference. The compiler then puts in the necessary pointers to get it to work. The programmer write the code as though it is passed by value (no pointers stuff) and the compiler fixes it automatically.

For example both programs do the same thing:

<pre>void bill(int *x) { *x = *x + 5; } main() { int y = 1; bill (&y); printf("y is %d\n", y); }</pre>	<pre>void bill(int &x) { x = x + 5; } main() { int y = 1; bill (y); printf("y is %d\n", y); }</pre>
---	--

The output for both is:

```
y is 6
```

Note in the right version the, we write the code as though it were pass by value. In reality the compiler puts in all of the pointer stuff and produces machine code equivalent to the program on the left (with pointers).

The next new feature is the operator overload. The line

```
void
Person::operator = (Person &other_object)
```

Indicated the function is to be called “=”. There are several things happening here.

First the function is called “=” not “bill”, “copy”, etc but a symbol. Next the name of “=” is already used. Finally the way we call the function also changes.

When we call the function we use:

```
bill = fred; // Our "=", not the
usual one.
```

Not

```
= (bill, fred);
```

as we for other functions as in

```
copy (bill, fred);
```

This makes the code much nicer to read. It looks very mathematical.

Next since we have functions = for integers, double, char, etc. we are basically using an already existing name. In the line

```
int    x, y;  
x = y
```

the = is a function that copies the data in the right parameter (the y) to the left parameter (the x). If x and y were type double then the = function will be a different function that accepts doubles. We are using operator overloading. Operator overloading like function overloading is the act of giving more than one function the same name. The compiler distinguishes between them by the argument list. For example

```
int  bill(int x)
{
    return x + 5;
}

double  bill(double x)
{
    return x + 5.5;
}

main()
{
    int      a = 1;
    double   b = 1.5;

    bill(a);
    bill(b);
    printf("a is %d and b is %lf\n",a,b);
}
```

The output is

```
a is 6 and b is 7.0
```

In this program

```
bill (a)
```

calles the top bill function and

```
bill (b);
```

calls the bottom one. The compiler knows which one to call by the argument passed to it.

So the line

```
bill = fred;           // Our "=", not the usual one.
```

In our program calls the = function that we wrote for the class. The compiler knows to call this function because of the type of bill and fred being object of type Person.

The member function called operator = is used to allow the assignment of objects. If one does not supply this function, assignment of objects may not be possible or the compiler may generate a default function. If a default function is used it will only copy the contents of the source object to the destination object. If the source object contains a pointer then the default assignment function will only copy the pointer itself and not the contents of the memory that the pointer points to. This could lead to problems. It is advised that if you plan to use an operator with an object that you specify the member function and do not rely on the default.

The line

```
friend void show_older(Person &a, Person &b);
```

in the class declaration says that while show_older() is not a member function of the class it does have access to its private elements. Now the function:

```
void
show_older(Person &a, Person &b)    // I am a friend of
Person
{                                   // so I can access private
data
    cout << "The older person is : ";
    if (a.age > b.age)
        a.display();
    else
        b.display();
}
```

receives 2 objects and can access its private elements. From the code you can see that object a and b are treated the same. This shows that this is a function involves a and b but is not part of either. In the function call:

```
show_older(bill, fred);
```

bill and fred are treated the same as well. If it not were for friend functions this is how it will have been written:

```
bill.show_older(fred);
```

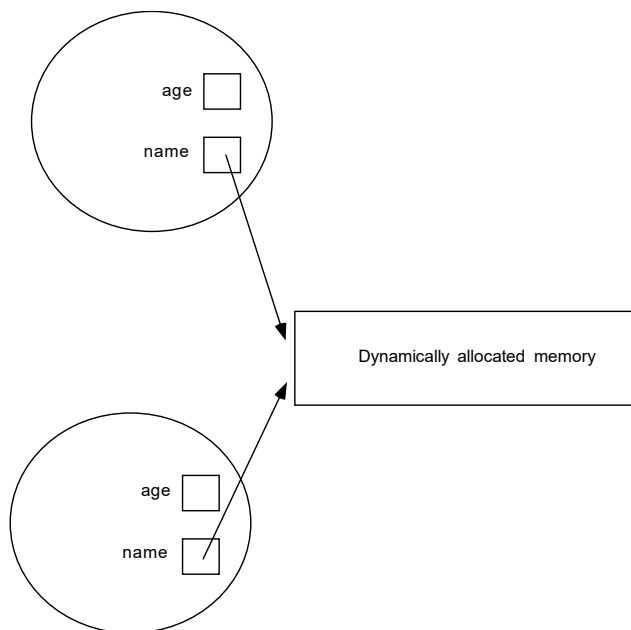
or

```
fred.show_older(bill);
```

This indicates the function corresponds to bill or fred when it corresponds to neither.

10.3 Passing objects

In both functions that have an object being passed, the object is passed by reference. Passing by value could lead to the following situation. An object is declared in the main function. The constructor allocates memory dynamically to this object. The object is passed to the function as pass by value. The function creates a new object but its constructor is not executed. Instead a simple byte by byte copy is made. The function you created to copy objects is not called either. When the pointer variable called name is copied it only copies the pointer itself not the contents of the memory that name points to. This means the name variable in the new object points to the same memory as the old object.



This is pretty bad but it gets worse. When the function terminates it deletes all of its local variables. Included is the object it created to receive the argument passed to it. When this object is destroyed its destructor is called. The destructor then returns the dynamically allocated memory pointed to by name. After the function is called the object that was passed has lost its memory even though name still points to it. Accessing name at this point may crash the system. To fix this it is best to always pass objects by reference.

Example: In the following program, Name is a pointer to a character. Its constructor allocates memory dynamically while its destructor returns this memory. It even has a function called “operator =” to perform copying correctly.

```
// PassingObjectsTest.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "string.h";

class A
{
public:
    char *Name;

    void show()
    {
        printf("Showing \"%s\" \n", Name);
    }

    void Change(const char n[80])
    {
        printf("Changed \"%s\" to \"%s\" \n", Name,n);
        strcpy_s(Name, 80, n);
    }

    A(const char n[80])
    {
        Name = new char[80];

        strcpy_s(Name, 80,n);
        printf("Constructor: Creating \"%s\" \n", Name);
    }

    ~A()
    {
        printf("Destructor: Destroying \"%s\" \n", Name);
        delete Name;
    }

    void operator = (A &other_object)
    {
        printf("operator = : \n", );
    }

};

void funct_PassByValue(A obj)
{
    printf("Just entered into funct_PassByValue \n");
    obj.show();
    obj.Change("Bill");
    obj.show();
    printf("About to leave funct_PassByValue \n");
}
```

```
void funct_PassByReference(A &obj)
{
    printf("Just entered into funct_PassByReference \n");
    obj.show();
    obj.Change("Fred");
    obj.show();
    printf("About to leave funct_PassByReference \n");
}

void funct_PassByPointer(A *obj)
{
    printf("Just entered into funct_PassByPointer \n");
    obj->show();
    obj->Change("Anna");
    obj->show();
    printf("About to leave funct_PassByPointer \n");
}

void MyMain()
{
    printf("Creating local objects Obj1, Obj2, and Obj3 :\n");
    A obj1("Obj_ByValue");
    A obj2("Obj_ByRef");
    A obj3("Obj_ByPointer");

    printf("\nGoing to call funct_PassByValue(Obj1) \n");
    funct_PassByValue(obj1);
    printf("returned from funct_PassByValue(Obj1) \n");
    obj1.show();

    printf("\nGoing to call funct_PassByReference(Obj2) \n");
    funct_PassByReference(obj2);
    printf("returned from funct_PassByReference(Obj2) \n");
    obj2.show();

    printf("\nGoing to call funct_PassByPointer(Obj3) \n");
    funct_PassByPointer(&obj3);
    printf("returned from funct_PassByPointer(Obj3) \n");
    obj3.show();

    printf("\nLeaving MyMain\n");
}

int main()
{
    MyMain();

    getchar();

    return 0;
}
```

Executing this program generally crashes the system however this one time it did not and its output was captured. Note in the pass-by-value case, the pointer Name points to

memory that has since been returned. It appears the operating system reused that memory since it now points to garbage.

```

Creating local objects Obj1, Obj2, and Obj3 :
Creating local objects Obj1, Obj2, and Obj3 :
Constructor: Creating "Obj_ByValue"
Constructor: Creating "Obj_ByRef"
Constructor: Creating "Obj_ByPointer"

Going to call funct_PassByValue(Obj1)
Just entered into funct_PassByValue
Showing "Obj_ByValue"
Changed "Obj_ByValue" to "Bill"
Showing "Bill"
About to leave funct_PassByValue
Destructor: Destroying "Bill"
returned from funct_PassByValue(Obj1)
Showing "|||||||||||||||||||||||||||||||||||||||||| 1"

Going to call funct_PassByReference(Obj2)
Just entered into funct_PassByReference
Showing "Obj_ByRef"
Changed "Obj_ByRef" to "Fred"
Showing "Fred"
About to leave funct_PassByReference
returned from funct_PassByReference(Obj2)
Showing "Fred"

Going to call funct_PassByPointer(Obj3)
Just entered into funct_PassByPointer
Showing "Obj_ByPointer"
Changed "Obj_ByPointer" to "Anna"
Showing "Anna"
About to leave funct_PassByPointer
returned from funct_PassByPointer(Obj3)
Showing "Anna"

Leaving MyMain
Destructor: Destroying "Anna"
Destructor: Destroying "Fred"
Destructor: Destroying "|||||||||||||||||||||||||||||||||||||||||| 1"

```

Example:

In the next example, the memory is not allocated dynamically but rather it's an array declared as part of the object. In this case, the byte-by-byte copy included the data in the

array memory. So the compiler produced code to copy the data correctly. That is, the array itself is also copied to an object that is passed by value. However there is still a problem. Its constructor is not called but its destructor is executed at the end of the function execution. Therefore you have a situation where an objects destructor is called but not its corresponding constructor. This may pose some problems and as a programmer one must be aware of such consequences. Furthermore, not all compilers will copy the data in the array. Some may choose only to copy the pointer.

```
// PassingObjectsTest.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "string.h";

class A
{
public:
    char    Name[80];

    void show()
    {
        printf("Showing \"%s\" \n", Name);
    }

    void Change(const char n[80])
    {
        printf("Changed \"%s\" to \"%s\" \n", Name,n);
        strcpy_s(Name, 80, n);
    }

    A(const char n[80])
    {
        strcpy_s(Name, 80,n);
        printf("Constructor: Creating \"%s\" \n", Name);
    }

    ~A()
    {
        printf("Destructor: Destroying \"%s\" \n", Name);
    }

    void operator = (A &other_object)
    {
        printf("operator = : \n", );
    }

};

void funct_PassByValue(A obj)
{
    printf("Just entered into funct_PassByValue \n");
    obj.show();
    obj.Change("Bill");
}
```



```
        obj.show();
        printf("About to leave funct_PassByValue \n");
    }

void funct_PassByReference(A &obj)
{
    printf("Just entered into funct_PassByReference \n");
    obj.show();
    obj.Change("Fred");
    obj.show();
    printf("About to leave funct_PassByReference \n");
}

void funct_PassByPointer(A *obj)
{
    printf("Just entered into funct_PassByPointer \n");
    obj->show();
    obj->Change("Anna");
    obj->show();
    printf("About to leave funct_PassByPointer \n");
}

void MyMain()
{
    printf("Creating local objects Obj1, Obj2, and Obj3 :\n");
    A obj1("Obj_ByValue");
    A obj2("Obj_ByRef");
    A obj3("Obj_ByPointer");

    printf("\nGoing to call funct_PassByValue(Obj1) \n");
    funct_PassByValue(obj1);
    printf("returned from funct_PassByValue(Obj1) \n");
    obj1.show();

    printf("\nGoing to call funct_PassByReference(Obj2) \n");
    funct_PassByReference(obj2);
    printf("returned from funct_PassByReference(Obj2) \n");
    obj2.show();

    printf("\nGoing to call funct_PassByPointer(Obj3) \n");
    funct_PassByPointer(&obj3);
    printf("returned from funct_PassByPointer(Obj3) \n");
    obj3.show();

    printf("\nLeaving MyMain\n");
}

int main()
{
    MyMain();

    getchar();

    return 0;
}
```

The output is:

```
Creating local objects Obj1, Obj2, and Obj3 :
Constructor: Creating "Obj_ByValue"
Constructor: Creating "Obj_ByRef"
Constructor: Creating "Obj_ByPointer"

Going to call funct_PassByValue(Obj1)
Just entered into funct_PassByValue
Showing "Obj_ByValue"
Changed "Obj_ByValue" to "Bill"
Showing "Bill"
About to leave funct_PassByValue
Destructor: Destroying "Bill"
returned from funct_PassByValue(Obj1)
Showing "Obj_ByValue"

Going to call funct_PassByReference(Obj2)
Just entered into funct_PassByReference
Showing "Obj_ByRef"
Changed "Obj_ByRef" to "Fred"
Showing "Fred"
About to leave funct_PassByReference
returned from funct_PassByReference(Obj2)
Showing "Fred"

Going to call funct_PassByPointer(Obj3)
Just entered into funct_PassByPointer
Showing "Obj_ByPointer"
Changed "Obj_ByPointer" to "Anna"
Showing "Anna"
About to leave funct_PassByPointer
returned from funct_PassByPointer(Obj3)
Showing "Anna"

Leaving MyMain
Destructor: Destroying "Anna"
Destructor: Destroying "Fred"
Destructor: Destroying "Obj_ByValue"
```

10.4 Inheritance

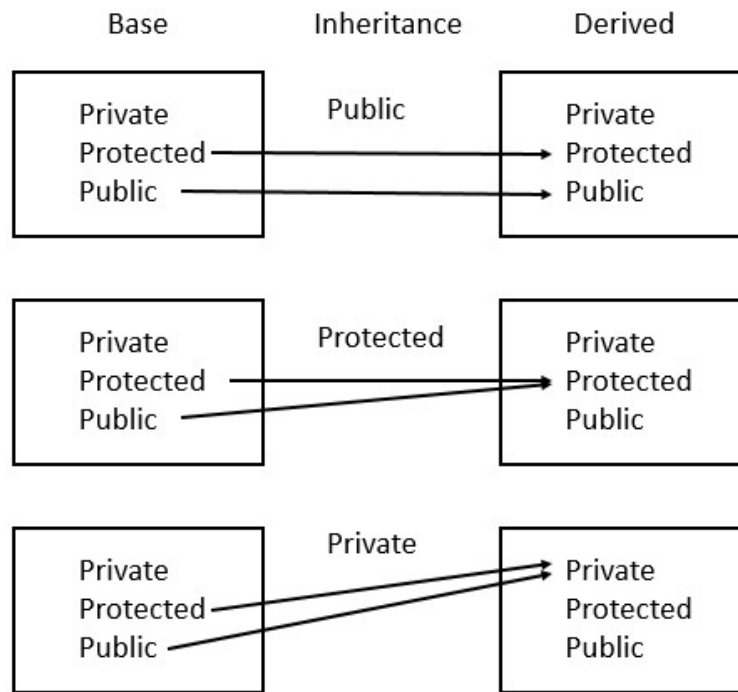
Now consider that you want to make a class to represent an employee. Since an employee is a person that works at your company we will create a class Employee that inherits class Person so that we do not need to recreate all that code. Consider the code:

```

Class Employee : public Person
{
}

```

The following is a summary of the types of inheritance that can be used.



10.5 Polymorphism

Polymorphism allows a pointer that is declared to point to a base class to also point to any derived class.

Consider that your company has 3 types of people, employees, contractors, and suppliers and you wish to produce a report of all associated people in the company. Assume you created class Employee, class Contractor and class Supplier all derived from class Person. Assume your data is stored in an array declared as follows:

```
Class Person *People[1000];
```

Then the code to produce the report may look like the following:

```

int i;
For (i = 0; i < NumberOfPeople; i++)
    People[i]->display();

```

Now if the company were to add a new type of person, say clients, with class Client also derived from class Person then the report function can still be generated without any code modification. If the code to generate the report come across an object of class Client it simply calls its display function just like it does with the other object types. This allows your code to be extended without being modified, a characteristic of good programming.

Polymorphism implements dynamic binding, that is, the program does not determine the function to call until run time. In a standard function like

```
x = sin(y);
```

the compiler knows that this function call goes to function sin(.). The address of function sin(.) is hardcoded into that spot in the program. On the other hand with dynamic binding as in

```
x = ObjPnt->display();
```

the compiler does not know which function will be called. It depends on what type of class ObjPnt points to at the time. The binding of the function is performed dynamically at run time.

Chapter 11 Complexity Analysis of Algorithms

11.1 Introduction

The big-O notation is a measure of the order of functions. This is a measure of how the function grows as the input grows. For the purpose of algorithms the function is a measure of the amount of time an algorithm will take to execute. The growth function tells how the required time to execute will grow as the number of input items grows.

Big-O notation and definition:

A function $t(n)$ is said to be of order of $f(n)$, written as $O(f(n))$, IFF there exist a constant C and n_0 such that $t(n) \leq cf(n) \forall n \geq n_0$

Example:

$$\begin{aligned} t(n) &= (n+1)^2 \\ &= n^2 + 2n + 1 \\ &\leq n^2 + 2n + n^2 \forall n \geq 2 \\ &\leq 4n^2 \forall n \geq 2 \Rightarrow O(n^2) \end{aligned}$$

Another example:

$$\begin{aligned} t(n) &= 7n^4 + 3n^3 + 5n^2 \\ &\leq 7n^4 + 3n^4 + 5n^4 \\ &= 15n^4 \Rightarrow O(n^4) \end{aligned}$$

Rule of sums of orders:

if $t_1(n) = O(f_1(n))$ and $t_2(n) = O(f_2(n))$ then
 $t_1(n) + t_2(n) = O(\max(f_1(n), f_2(n)))$

Proof:

$$\begin{aligned} t_1(n) &\leq k_1 f_1(n) \\ t_2(n) &\leq k_2 f_2(n) \\ t_1(n) + t_2(n) &\leq k_1 f_1(n) + k_2 f_2(n) \\ &\leq k(f_1(n) + f_2(n)) \\ &\leq k(2 \max(f_1(n), f_2(n))) \\ &= O(\max(f_1(n), f_2(n))) \end{aligned}$$

Rule of products of orders:

if $t_1(n) = O(f_1(n))$ and $t_2(n) = O(f_2(n))$ then
 $t_1(n)t_2(n) = O(f_1(n)f_2(n))$

Proof:

$$\begin{aligned} t_1(n) &\leq k_1 f_1(n) \\ t_2(n) &\leq k_2 f_2(n) \\ t_1(n)t_2(n) &\leq k_1 f_1(n)k_2 f_2(n) \\ &\leq k f_1(n)f_2(n) \\ &= O(f_1(n)f_2(n)) \end{aligned}$$

The purpose of the big-O notation is to measure the growth of a function. Note that the constants of the growth function are ignored. This is because regardless of the size of the constant, a function with a higher order of growth will eventually be greater than a function with a growth function of less order but with a much larger constant.

Example: Find the time complexity of the following function.

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

let $t(n)$ be the amount of time it takes $\text{fac}(n)$ to return a result.
then

$$t(n) = \begin{cases} d & \text{if } n \leq 1 \\ c + t(n-1) & \text{if } n > 1 \end{cases}$$

$$\begin{aligned} t(n) &= t(n-1) + c \\ &= t(n-2) + c + c \\ &= t(n-3) + 3*c \\ &\vdots \\ &= t(1) + (n-1)*c \\ &= d + (n-1)*c \\ &= O(n) \end{aligned}$$

So the factorial function $\text{fac}(n)$ has linear time complexity. Or in other words, the amount of time to execute $\text{fac}(n)$ grows linearly with n .

Example: find the time complexity of the following double for loop.

```
for (r = 0; r < n; r++)
    for (c = 0; c < m; c++)
        { ??? }
```

Assume the ??? statement takes duration of K.

let $t(n)$ be the amount of time it takes to complete.
then

$$\begin{aligned} t(n) &= \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} K = \sum_{i=0}^{n-1} K((m-1) - 0 + 1) = K \sum_{i=1}^{n-1} m = Km \sum_{i=0}^{n-1} 1 \\ &= Km((n-1) - 0 + 1) = kmn = O(mn) \end{aligned}$$

$$\text{recall: } \sum_{i=1}^n i = \frac{n(n+1)}{2} \text{ and } \sum_{i=m}^n K = k(n-m+1)$$

So the time to execute the double for loop is proportional to the square of n.

Example: find the time complexity of the following more complex double for loop.
Notice the dependency between the two for loops.

```
for (I = 1; I <= n - 1; I++)
    for (j = n; j >= I+1; j--)
        { ??? }
```

Assume the ??? statement takes duration of K.

let $t(n)$ be the amount of time it takes to complete.
then

$$\begin{aligned} t(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n K = \sum_{i=1}^{n-1} K(n - (i+1) + 1) = K \sum_{i=1}^{n-1} (n - i) = K \left(\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i \right) \\ &= K(n(n-1) - \sum_{i=1}^{n-1} i) = Kn(n-1) - K \frac{(n-1)n}{2} \\ &= Kn^2 - Kn - \frac{Kn^2}{2} + \frac{Kn}{2} = O(n^2) \end{aligned}$$

So the time to execute the double for loop is proportional to the square of n.

Chapter 12 Lists

12.1 Introduction

12.2 The General List Abstract Data Type

The list is a general ADT where data is stored in a linear list or a sequence. The following functions are included in the ADT:

- **Insert** any at a given location
- **Remove** at a given location
- **Insert** at the **front**
- **Remove** at the **front**
- **Insert** at the **rear**
- **Remove** at the **rear**
- Check to if the list is **empty**
- Find the **next** component in the list
- Find the **previous** component in the list

From the general list ADT we can get some special cases.

The Stack ADT is a special case of the List ADT where the data can only be inserted and removed from one end of the list. The functions are called Push, Pop and empty. Push inserts and pop removes data from the list. This list is accessed in a last-in-first-out (LIFO) order.

The Queue ADT is also a special case of the List ADT. In a queue the data is inserted into one end and removes from the other end. It accesses the data in a first-in-first-out (FIFO) order. The functions are called enqueue for inserting and dequeue for removing.

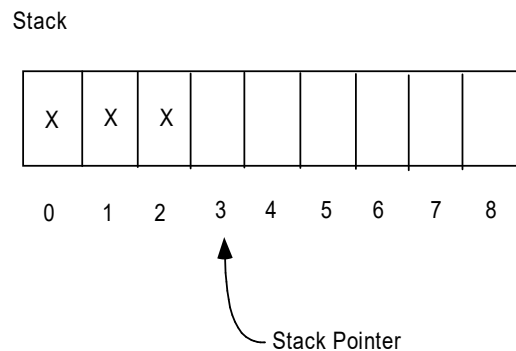
Depending on the implementation the time complexity of these functions varies.

12.4 Array implementation of the general list

The most straightforward way to implement a list is with a simple array. This may not yield the best performance however. The insertions at either end of the list is fast but to insert into the middle of the list requires all of the data from the place you wish to insert to the end of the list to be shifted one space to the back. The same is true to remove. A shift is necessary to fill the gap. Since insertions and removals at either end are fast, simple arrays may be used to implement stacks and queues.

The Stack

To implement a stack we need to insert and remove from one end of the list. The end of the list may be chosen as to avoid any shifting in the list.



In this case the stack pointer points to the next available spot. To insert into the stack we place the data in the location pointed to by the stack pointer then increment the stack pointer. To remove we first decrement the stack pointer then we take the data from the location pointed to by the stack pointer. The list is empty when the stack pointer is 0 and is full when the stack pointer is N (in this example 9 since there are 9 spots in the stack).

```
#define      N      100

class  Stack
{
private:
    int      array[N],
    int      stack_pointer,
public:

    Stack( )
    {
        stack_pointer = 0;
    }

    int      empty( )
    {
        return stack_pointer == 0;
    }

    int      full( )
    {
        return stack_pointer >= N;
    }
}
```

```

    }

void push( int data)
{
    if (stack_pointer < N)
        stack[stack_pointer++] = data;
}

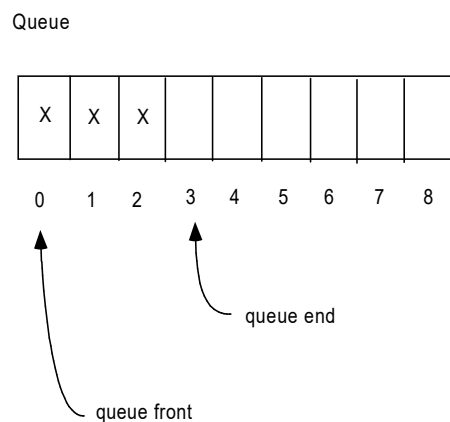
int pop(      )
{
    if (stack_pointer != 0)
        return stack[--stack_pointer];
    else
        return -1;
}
}

```

This example assumes a stack size of 100, the data to store being integer and returns -1 if the stack is empty. If -1 is a valid data in the stack then the function can not distinguish between a -1 in the stack and an empty stack. In this case return a number that will never be stored in the stack. You may also have a flag or a message.

The Queue

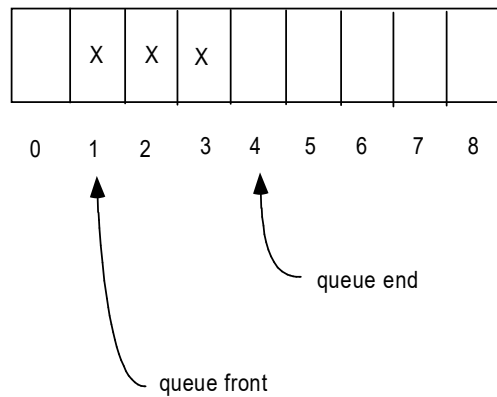
To implement a queue we need to insert at one end and remove from the other end of the list.



In this case the front pointer points to the occupied spot at the front of the list and the end pointer points to the next available spot at the end of the list. To enqueue (insert rear) we place the data at the position pointed to by the end pointer then increment the pointer. To remove we take the data pointed to by the front pointer then increment the pointer. There is a problem however. Consider inserting an item into the queue then removing an item.

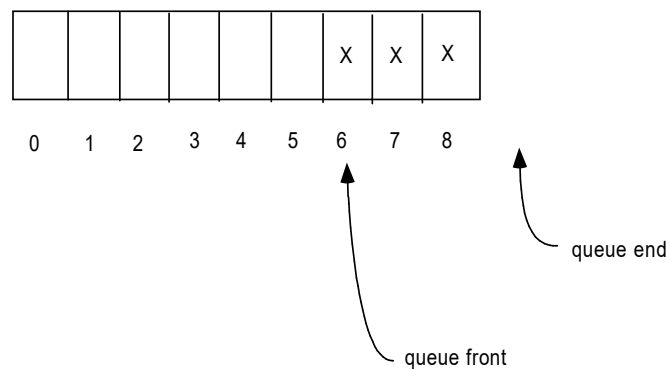
The above queue will look like:

Queue

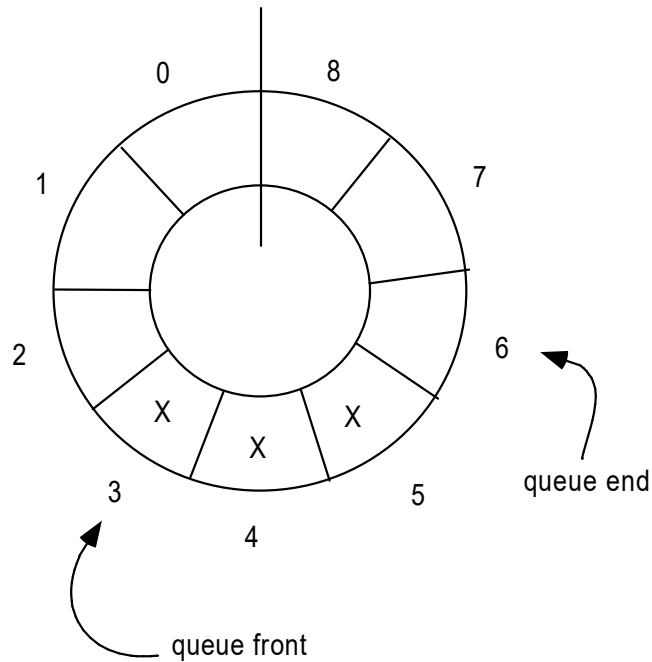


Now imagine repeating this 5 more times. The above queue will look like:

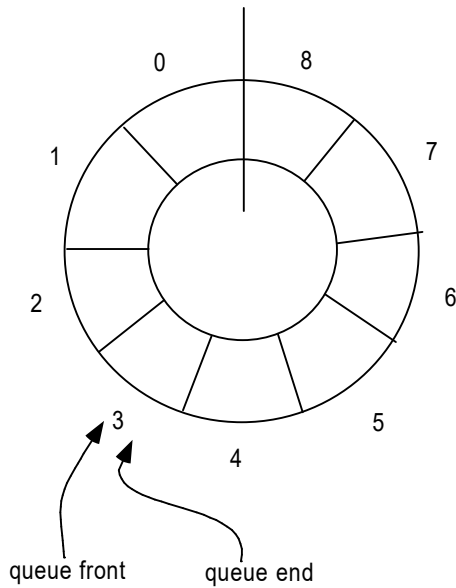
Queue



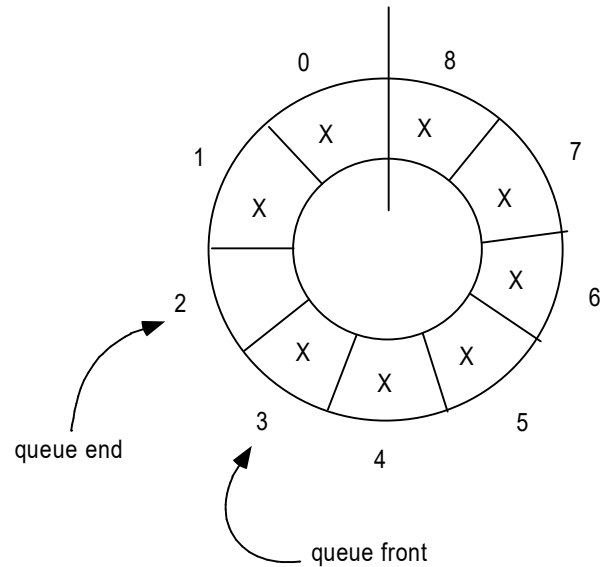
Now if I want to insert a new item I can't. The queue tends to shift with usage and at this point the queue has shifted all the way to the right. One solution is to shift the queue all the way to the left every time the queue shifts itself to the right. This is a $O(n)$ operation that is not necessary. Instead a better solution is to think of the array as being a circular array. This is a simple array where we assume the two ends are tied together. After spot 8 comes spot 0 and before spot 0 is spot 8.



With this array the queue may shift itself as much as it wants. It simply keeps going around in circles. To insert and remove is the same as before with the exception that we wrap around. We use the modulus function (%) in C to wrap around. As before the queue is empty when front pointer and the end pointer are equal. Notice however that if insert 9 items into the queue such that it is full then the front pointer and the end pointer will be equal. This state is the same as the state when the queue is empty. So how do we distinguish between the two? One solution is to have a flag that tells whether the condition means the queue is empty or full. A simpler solution is to simply not let the queue get full. Consider the queue as full when there are $N-1$ items stored in the queue as opposed to N . In this case the end pointer will be just before the front pointer.



Empty condition



Full condition

```

#define      N      100

void enqueue(  int    array[N],
               int    queue_front,
               int    queue_rear,
               int    data)
{
    if ( (queue_end + 1) % N != queue_front )
    {
        queue[queue_end] = data;
        queue_end = (queue_end + 1) % N;
    }
}

int deque(    int    array[N],
              int    queue_front,
              int    queue_rear)
{
    if ( (queue_end != queue_front) )
    {
        hold = queue[queue_front];
        queue_front = (queue_front + 1) % N;
    }
}

```

```

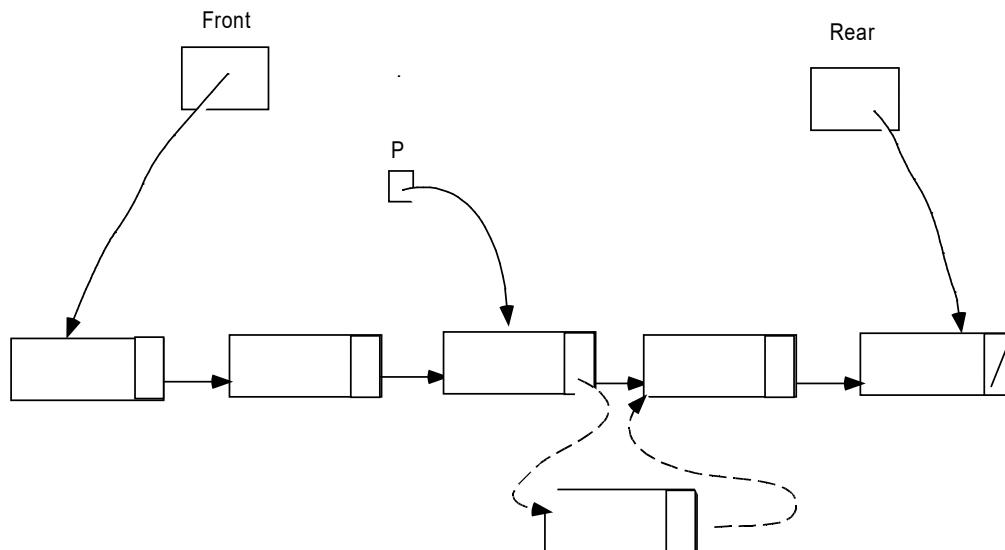
else
    return -1;
}

```

12.5 Linked list implementation of general list

With linked list we get some flexibility. We can insert and remove an item from anywhere in the list with an algorithm whose time complexity is proportional to 1, $O(1)$. There is no need to shift items to make space for a new item.

To insert into the middle of the list we need a pointer to point to the node that is before the location to insert. This way we have direct access to the node that is before the new node and do not have to traverse the list to find it.



The structure of the node is:

```

struct node
{
    int    data;
    node  *next;
}

```

Note, the data type is integer for simplicity. In reality it will be a more complex type.

Insert into List

We want to insert a new node after P.

There are 4 cases:

1. The list is **empty**.
2. We want to insert at the **front** of the list.
3. We want to insert at the **end** of the list.
4. We want to insert **between** two other nodes.

Since p points to the node before the new node, to insert a node at the front of the list p must be NULL.

```

void insert(    node    **f,
               node    **r,
               node    *p,
               int      data)
{
    node    *q;

    q = new node;
    q->data = data;

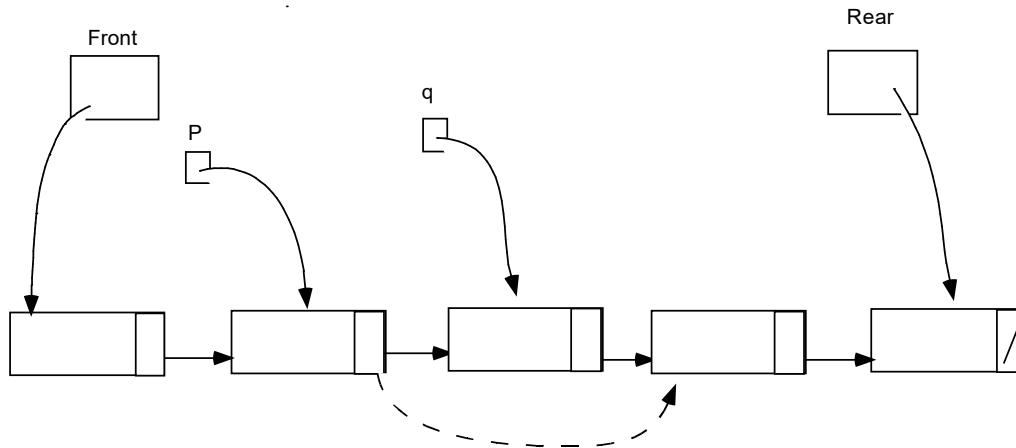
    if (*f == NULL) // case 1
    {
        *f = *r = q;
        q->next = NULL;
    }
    else if (p == NULL) // case 2
    {
        q->next = *f;
        *f = q;
    }
    else if (p == *r) // case 3
    {
        p->next = q;
        q->next = NULL;
        *r = q;
    }
    else // case 4
    {
        q->next = p->next;
        p->next = q;
    }
}

```

Remove from List

The remove function also has 4 case. P points to the node before the node to be removed.
The four cases:

1. We want to remove the **only** node.
2. We want to remove the **first** node.
3. We want to remove the **last** node.
4. We want to remove a node **between** two nodes.



```

int remove(    node **f,
               node **r,
               node *p)
{
    int    returnval;
    node   *q;

    // make q point to the node to delete.
    if (p != NULL)
        q = p->next;
    else
        q = *f;

    returnval = q->data;

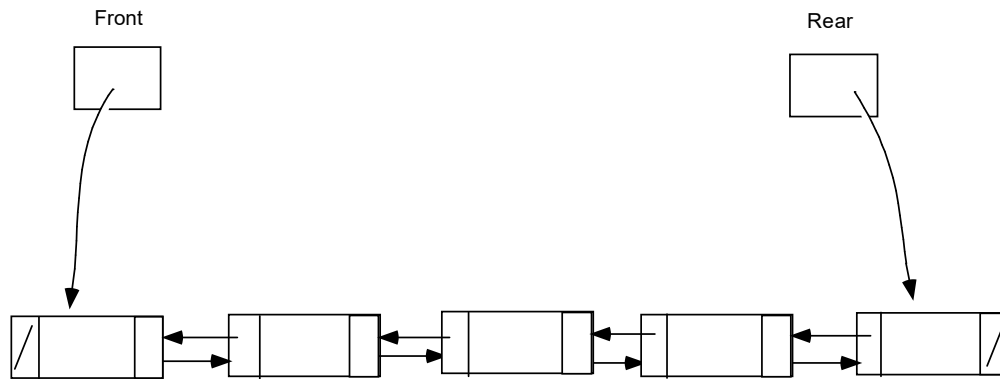
    if (*f == *r) // case 1
        *f = *r = NULL;
    else if (p == NULL) // case 2
        *f = (*f)->next;
    else if (p->next == *r) // case 3
    {
        *r = p;
        p->next = NULL;
    }
    else // case 4
        p->next = q->next;

    delete q;
    return returnval;
}

```

12.6 Doubly linked list

The search algorithm may require to move backwards in the list as well as forward. With single linked list this will require a step whose time complexity is proportional to the number of nodes in the list. A solution is to put 2 links into the list, One in each direction.



Now the search algorithm can move in either direction. The cost is about 2 bytes per node in memory. Now the remove function can have p point directly to the node to remove. For the insert function p still points to the node before the one to insert. Also the removal from the rear of the list can take $O(1)$ time as opposed to $O(n)$ time complexity.

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

node  *front, *rear;

void insert(    node  **f,
               node  **r,
               node  *p,
               int    data)
{
    node  *q;

    q = new node;
    q->data = data;

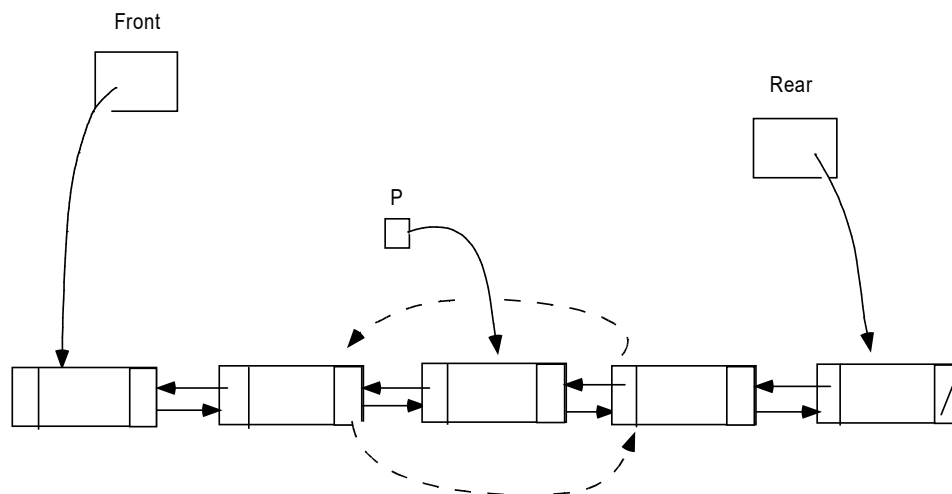
    if (*f == NULL) // case 1 list is empty
    {
        *f = *r = q;
        p->prev = q->next = NULL;
    }
```

```

else if (p == NULL) // case 2 insert at front
{
    q->next = *f;
    q->prev = NULL;
    (*f)->prev = q;
    *f = q;
}
else if (p == *r) // case 3 insert at rear
{
    p->next = q;
    q->prev = p;
    q->next = NULL;
    *r = q;
}
else // case 4 insert in the middle
{
    q->next = p->next;
    q->prev = p;
    p->next->prev = q;
    p->next = q;
}
}

```

Note p points to the node to remove. The problem still has the same 4 cases.



```

int remove(    node **f,
               node **r,
               node *p)
{
    int    returnval;

```

```

returnval = p->data;

if (*f == *r) // case 1 remove only node
    *f = *r = NULL;
else if (p == *f) // case 2 remove first node
    {
        *f = (*f)->next;
        (*f)->prev = NULL;
    }
else if (p == *r) // case 3 remove last node
    {
        *r = p->prev;
        (*r)->next = NULL;
    }
else // case 4
    {
        p->prev->next = p->next;
        p->next->prev = p->prev;
    }

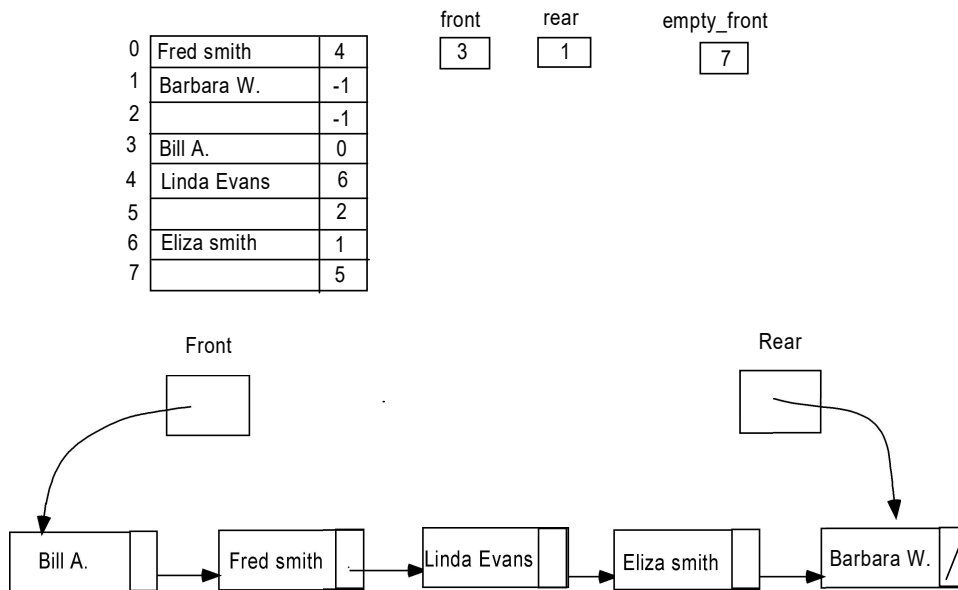
delete p;
return returnval;
}

```

12.7 Linked List in Arrays

Some of the older but still used programming languages and most assembly languages, especially for small micro-controllers, do not provide a facility to allocate memory dynamically. Yet the need to implement linked list may still exist. Linked list can be implemented using simple arrays. The limitation of having to know the maximum size of the list before hand as is the case for array implementation of list still exist. However for inserting and removing from the middle array implementation of linked list still gives $O(1)$ algorithms like regular linked list.

The top diagram is an array implementation of linked list. The bottom is the linked list that is in the array. Note the empty nodes form them selves a linked list so that getting a new node is actually removing the first node in the empty list.



The data structures is as follows:

```

struct node
{
    char    data[80];
    int     next;
}

node  list[10],
      front,
      rear,
      empty_front;

```

The insert and remove functions are much the same but the linking is different. where you have `p->next` you replace it with `list[p].next` and where you have `NULL` you replace it with `-1` and finally the new or `malloc` function is replaced by `get_new_node(list,ef)`. The function `get_new_node` is simple a pop function for the empty list.

Since `p` points to the node before the new node, to insert a node at the front of the list `p` must be `NULL` (`-1` in this case).

```

void insert(    node  list[10],
               int    *ef
               int    *f,
               int    *r,
               int    p,
               int    data)
{
    int    q;

    q = get_new_node(list,ef); // q = new node;
    list[q].data = data // q->data = data;

    if (*f == -1) // case 1 list is empty
    {
        *f = *r = q;
        list[q].next = -1 // q->next = NULL;
    }
    else if (p == -1) // case 2 insert at the front
    {
        list[q].next = *f; // q->next = *f;
        *f = q;
    }
    else if (p == *r) // case 3 insert at the rear
    {
        list[p].next = q; // p->next = q;
    }
}

```

```
        list[q].next = -1; // q->next = NULL;
        *r = q;
    }
else // case 4 insert in the middle
    {
        list[q].next = list[p].next; // q->next = p->next;
        list[p].next = q; // p->next = q;
    }
}

int get_new_node(    node    list[10],
                   int      *ef)
{
    if (*ef == -1)
        return -1;
    else
    {
        p = *f;
        *f = list[p].next; // *f = p->next;
        list[p].next = -1; // p->next = NULL;
        return p;
    }
}
```

12.8 Comparisons between Different Implementation Methods

So far we have seen several ways to implement general list. The following is a table of all of the methods and their time complexities.

	linear array	single L.L. using pointers	double L.L. using pointers	single L.L. using arrays	double L.L. using arrays
empty	O(1)	O(1)	O(1)	O(1)	O(1)
next	O(1)	O(1)	O(1)	O(1)	O(1)
prev	O(1)	O(n)	O(1)	O(n)	O(1)
insert	O(n)	O(1)	O(1)	O(1)	O(1)
remove	O(n)	O(1)	O(1)	O(1)	O(1)
insert front	O(1)	O(1)	O(1)	O(1)	O(1)
remove front	O(1)	O(1)	O(1)	O(1)	O(1)
insert rear	O(1)	O(1)	O(1)	O(1)	O(1)
remove rear	O(1)	O(n)	O(1)	O(n)	O(1)
push	O(1)	O(1)	O(1)	O(1)	O(1)
pop	O(1)	O(1)	O(1)	O(1)	O(1)
enqueue	O(1)	O(1)	O(1)	O(1)	O(1)
dequeue	O(1)	O(1)	O(1)	O(1)	O(1)

You can see that to implement a simple stack or queue the simple array implementation is best in terms of time complexity. Its time complexity is O(1) for all functions except for insert and remove from the middle of the list which neither the stack or the queue will use. If you are using a machine that does not support pointers or dynamic memory allocation, then to implements a general list the array implementation of pointers is better than using simple arrays.

Chapter 13 Searching Algorithms

13.1 Introduction

This section deals with searching through a collection of data to find a particular item. The simplest method is to use a simple array and to store the data in any order. To find an entry one simply starts with the first component of the list and searches one by one until it finds the entry or searches the whole list to the end.

Algorithm:

```
int find(          datatype    data[N],
          int      last,
          datatype entry)
{
    data[last] = entry;
    while (data[I] != entry)
        I++;
    if (I == last)
        return -1;
    else
        return I;
}
```

On average, the algorithm will have to search half of the list before it finds the entry. So the time complexity of the algorithm is proportional to the size of the list, $O(n)$. note the $\frac{1}{2}$ constant is eaten by the big-O notation. $O(n)$ simply means that if the n double in size so will the time to find an entry. This is very poor. Imagine trying to find a name in a telephone book and having to search through all 3 million names in Orlando. What about the telephone book for the United States ???

A better way, in fact the best way is to use a binary search. Here the data is maintained sorted. The idea is to eliminate $\frac{1}{2}$ of the data with every comparison. Each loop starts by comparing the entry to the data item that is located in the middle of the list. Depending on the comparison the entry to find will lie on one of the two halves. The half that does not contain the entry is discarded. The remaining half is then taken as the new list and the steps are repeated.

```
int BinaryFind(      datatype    data[N],
                   int      front,
                   int      end,
                   datatype    entry)
{
    if (end - front < 2)
    {
        if (data[front] == entry)
            return front;
    }
}
```

```

        else if data[end] == entry)
            return end;
        else
            return -1;
    }
else
    {
        middle = (end + front) / 2;
        if (data[middle] < entry)
            return BinaryFind(data, middle + 1, end, entry);
        else
            return BinaryFind(data, front, middle, entry);
    }
}

```

The following is the same algorithm but implemented without recursion.

```

int BinaryFind(    datatype    data[N],
                 int    last,
                 datatype    entry)
{
    int    front, end;

    front = 0;
    end = last - 1;
    while (end - front > 1)
    {
        middle = (end + front) / 2;
        if (data[middle] < entry)
            front = middle + 1;
        else
            end = middle;
    }

    if (data[front] == entry)
        return front;
    else if data[end] == entry)
        return end;
    else
        return -1;
}

```

It can be seen that since with every iteration the amount of data to search is halved. So if the size of the array doubles then it will require only 1 extra comparison (iteration) to find

the entry. To see how many iterations, or also referred as comparisons, it takes to find the entry we must start with 2 and keep doubling it until we get to the size of the array.

For array of 100 elements, we get 1,2,4,8,16,32,64,128. This results in 6 comparisons since $\log_2 128 = 7$. So this algorithm finds an entry in $O(\log n)$ time. Note that the base of the log is not important in the big-O notation because to change a base is equivalent to multiplying the log by a constant.

Recall: $y = \log_a x \Leftrightarrow a^y = x$

also $\log x^y = y \log x$

So $\log_b x = \log_b a^y = \log_b a^{\log_a x} = (\log_a x)(\log_b a)$ and $\log_b a$ is simply a constant.

As n grows the difference between the linear and the binary search becomes significant.

n	log n
10	3
1,000	10
1,000,000	20
1,000,000,000	30

Imagine, if it takes 1 second to perform a compare operation then it will take 30 seconds to search a list of one billion numbers using a binary search versus 16 years using a linear search.

You have seen an algorithm to implement a binary search on an array. It is possible that when the list is divided into 2 parts and when the middle entry is compared to the search entry that the middle entry equals the search entry. One possible improvement to the algorithm is to make the algorithm check not only to see if the search entry is greater or less than the middle entry but also to check if it is equal to the middle entry. If it is equal to the middle entry then the algorithm can stop right there since it has found the entry. Intuitively this algorithm is an improvement because as the area in the list to search becomes smaller the probability of hitting the entry increases. The improvement will not be orders of magnitude better. It is still $O(\log n)$. It may only be 2 or 3 times faster. The possibility of doubling the speed of an algorithm is still worth the time to investigate the algorithm even though the time complexity for the improved algorithm is still $O(\log n)$. To analyze an algorithm we use a comparison tree.

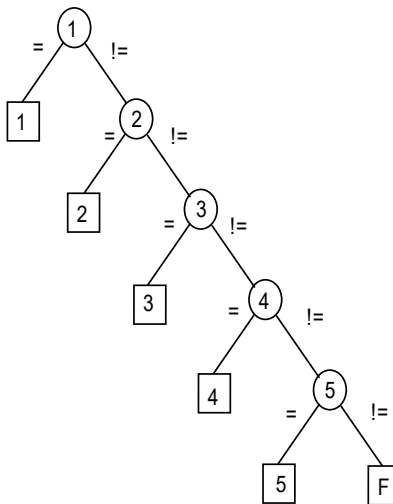
13.2 Comparison Trees

Comparison trees is a useful tool to analyze the running time of certain algorithms. Note that to search a list the running time depends on the number being searched. Is searching a list using the linear search algorithm and the number we are searching for is the first number in the list then that search algorithm is very fast. To find an estimate of the average running time we take the average of the running time for searching each number in the list. To do this we can not simply look at the algorithm. We must analyze the different path the algorithm takes for each number we search.

The comparison tree of an algorithm is obtained by tracing through the action of the algorithm, representing each comparison by a vertex of the tree. The branches of the trees represent the possible outcomes of the comparison. A leaf is an external vertex of the tree and is where a branch terminates. Consider the following list of 5 entries. Note the items do not have to be sorted to perform a linear search.

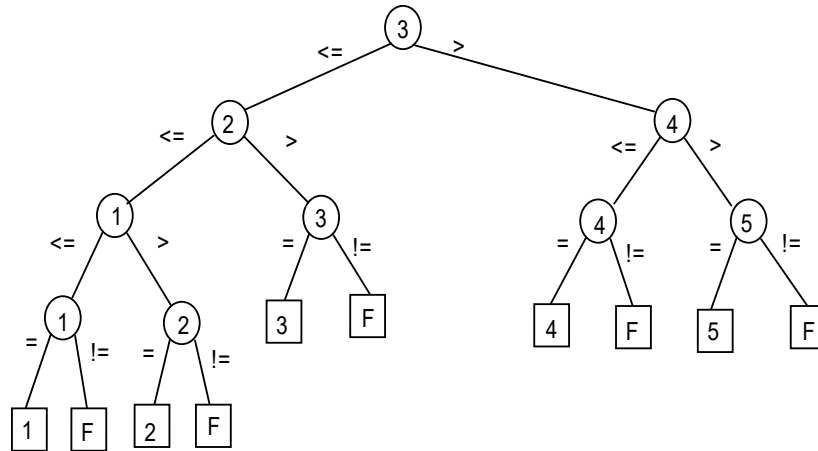
1	2	3	4	5
---	---	---	---	---

The following is a comparison tree for the linear search algorithm on this list.

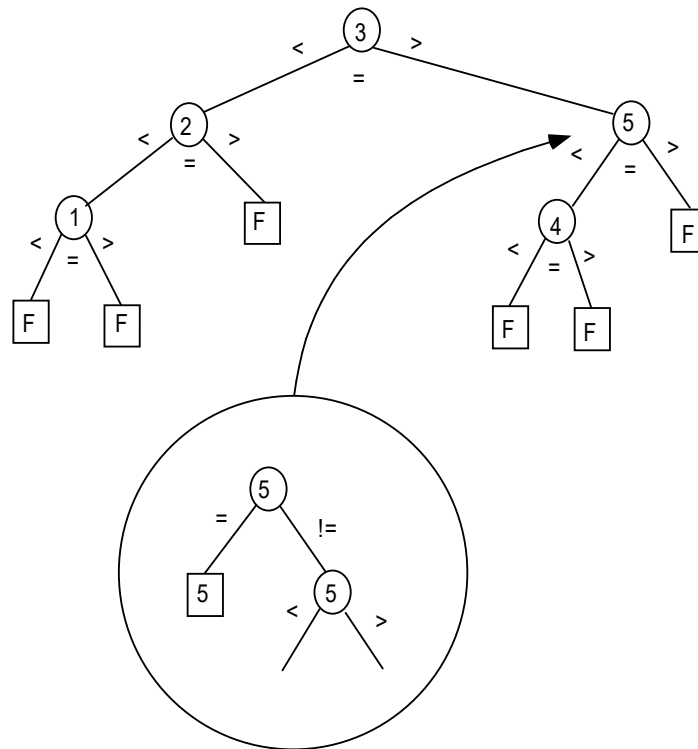


The circles represent comparisons. The squares represent leafs and a square with an F in it represents a failure (entry not found). Note to search for 4 the 4 will be compared to 1 then to 2 then to 3 then finally to 4. Vertices 1,2,3, and 4 will be traversed.

Now for the binary search algorithm we presented earlier, the comparison tree for the 5 entry list is as follows.



Note this algorithm does not check to see if the middle entry is the entry we are searching for. To find the 3 the algorithm traverses the vertices 3,2,3. Three comparisons are made. The comparison tree for the algorithm which checks the middle entry for a match follows.



Note the tree is drawn in a compact way. Each circle actually represents 2 comparisons. See the insert. One to see if the entry is equal to or not, and the other to see if it is not equal then is it less than or greater than the key. So you must remember to multiply the

length of the path by 2. To find the 4, 3 vertices are traversed so $3*2 = 6$ comparisons are required. In reality only 5 comparisons are required since the check for equality is performed first. If the comparison is equal then the check for less than or greater than is not performed. To find the 3, 1 vertex is traversed so 1 comparison is required.

To analyze an algorithm the average length of the paths are calculated. For every search, the algorithm will travel through one path. Starting at the root and going to a leaf. If we sum the length of all of the paths then divide it by the number of paths then we will get the average path length. The sum of all of the paths is referred to as the **external path length**. For the first algorithm, the linear search we get $1+2+3+4+5+5 = 20$. Note there are two paths with length 5; one to find the 5 and the other if not found. The average path length is then $20/6 = 3.33$. We are assuming that the probability to fail is the same as the probability to find a particular number. That is assuming 17% of the searches fail. Since the probability that an item is in the list is application dependent we want to calculate the average path length for success and a separate one for failure.

$$\text{Success } \frac{1+2+3+4+5}{5} = 3$$

$$\text{For failure } \frac{5}{1} = 5$$

For the lazy binary search (the one that does not check the middle entry for a hit) we get 2 paths of length 4 and 3 paths of length 3 for successes and the same for failure. So for

$$\text{Success } (2*4) + (3*3) = 17 \text{ and } \frac{17}{5} = 3.4 \text{ and for}$$

$$\text{Failure } (2*4) + (3*3) = 17 \text{ and } \frac{17}{5} = 3.4 .$$

Now for the second binary search algorithm we have

$$\text{Success } \frac{1+2+3+2+3}{5} * 2 - 1 = 3.4 \text{ and}$$

$$\text{Failure } \frac{3+3+2+2+3+3}{6} * 2 = 5.33 .$$

Note, for the success part we used the sum of all of the paths that go from the root to a vertex. This is referred to as the **internal path length**.

We calculated the average number of comparisons for several algorithms with 5 elements. Now we wish to compute the average number of comparison for the general case with n elements. So we need to determine the external and internal path length of a comparison tree given only the number of elements in the list. To do this we need some

theorems relating these path lengths to the number of vertices in the tree. We introduce 2-trees.

A 2-tree is a binary tree where each vertex has exactly 0 or 2 children. The vertices that have 0 children are the leaves.

For 2-trees the following lemmas exist.

- Lemma 6.1: The number of vertices on each level is at most 2 times the number on the level above.
- Lemma 6.2: The number of vertices on level t is at most 2^t for $t \geq 0$.
- Theorem 6.3: Let the external path length be E , the internal path length be I and q be the number of vertices that are not leaves then $E = I + 2q$.

Note: the root is considered to be level 0.

We can see that the comparison tree for both binary search algorithms are 2-trees.

Analysis of the lazy binary search algorithm.

We can see that all of the leaves in the lazy algorithm lie on either the last or the second to the last level. This means that the height of the tree is very close to the average path length. So we can use the height of the tree as the average path length which is the average number of comparisons made by the algorithm. Our goal is to find a formula that relates the number of entries in the list, n , to the height of the tree, t .

By lemma 6.2 and by the fact that the 2-tree has $2n$ leaves for the lazy algorithm, the height of the tree, t , is the smallest integer t such that $2^t \geq 2n$. To simplify the computation lets increase the number of entries to search, n such that $2^t = 2n$. Now since $2^t = 2n \Leftrightarrow t = \log_2 2n$ this means that the average number of comparisons made to search a list of n entries is $\log_2 2n$ using the lazy binary search algorithm. To further simplify the equation we can use the fact that $\log xy = \log x + \log y$ so

$$t = \log_2 2n = \log_2 n + \log_2 2 = \log_2 n + 1.$$

We know that the average number of comparisons for success is the same for failure and for the combine success and failure.

Analysis of the more complex binary search algorithm.

To analyze for failures: The binary search algorithm that checks the middle entry for a possible hit we have $n+1$ failures leaves. So by similar argument we get

$2^t = n + 1 \Leftrightarrow t = \log_2(n + 1)$. But since each vertex is actually 2 comparisons we get $t = 2 \log_2(n + 1)$. This is about 2 times the amount of comparisons for the lazy algorithm.

For the success case we use theorem 6.3. Again the goal is to relate the number of entries, n , to the height of the internal path length. Since there is an internal vertex for each entry in the list, the tree has n internal vertices. We saw from the failure case that the height of the tree is $\log_2(n + 1)$. Since all of the paths have about the same length we can approximate the external path length by multiplying the height of the tree by the number of paths. So the external path length is about $(n + 1) \log_2(n + 1)$. Using theorem 6.3 we have the internal path length is given by $(n + 1) \log_2(n + 1) - 2n$. Note that the lemma 6.2 assumes that the root is level 0. This means that the height of the tree is actually 1 less. So a tree with only a root is considered to have a height of 0. So to find the average number of comparisons we:

1. Divide the internal path length by n
2. Add 1 to compensate for the root being level 0
3. Double this quantity since each vertex is actually 2 comparisons and
4. Subtract 1 since the last comparison being equal means the second comparison is not performed.

We start with the internal path length, I .

$$I = (n + 1) \log_2(n + 1) - 2n$$

We divide the internal path length by n and get.

$$\frac{n + 1}{n} \log_2(n + 1) - 2$$

We then add 1 to compensate for the root being level 0 and get.

$$\frac{n + 1}{n} \log_2(n + 1) - 1$$

We then double this quantity since each vertex is actually 2 comparisons to get.

$$2 \frac{n + 1}{n} \log_2(n + 1) - 2$$

And finally we subtract 1 since the last comparison being equal means the second comparison is not performed to get.

$$\frac{2(n + 1)}{n} \log_2(n + 1) - 3$$

So the number of comparisons performed by the binary search algorithm that checks the middle entry for a hit is $\frac{2(n+1)}{n} \log_2(n+1) - 3$ for successes. As n grows we can

approximate the number of comparisons to $2 \log_2 n - 3$.

Summary:

Lazy binary search algorithm

Success $\log_2 n + 1$

Failure $\log_2 n + 1$

Other binary search algorithm

Success $2 \log_2 n - 3$

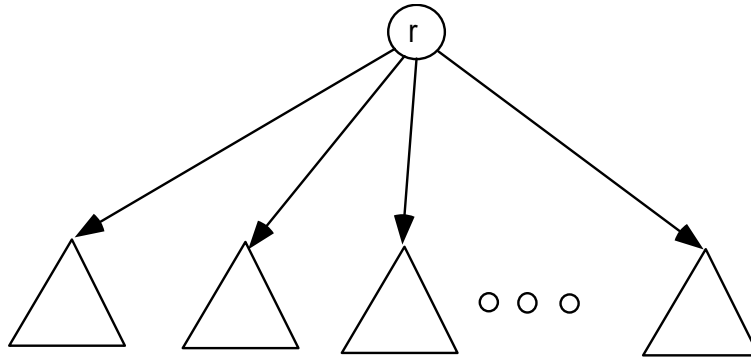
Failure $2 \log_2(n + 1)$

We can see that even for success this algorithm has about 2 times the number of comparisons that the lazy algorithm.

Chapter 14 Trees

14.1 Introduction

A tree can be defined recursively. A tree is a collection of nodes. It consists of a node r called the root and several possibly zero sub trees each whose roots are connected to the node r by a directed edge. The root of each sub tree is said to be a child of r and r is the parent of each sub tree root. A node with no children is called a leaf.



A path from node n_1 to node n_k is defined as a sequence of nodes n_1, n_2, \dots, n_k such that n_i is the parent of n_{i+1} . The length of the path is the number of nodes namely $k-1$.

For any node n_i the depth of n_i is the length of the unique path from the root to n_i . The root is at depth 0.

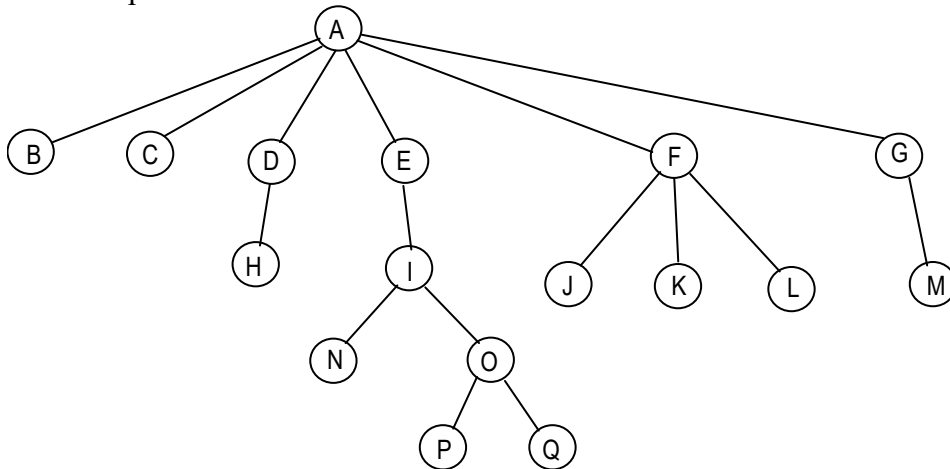
The height of n_i is the length of the longest path from n_i to a leaf. The leaves are at height 0 and the height of the tree is the height of the root.

14.2 Implementation of trees

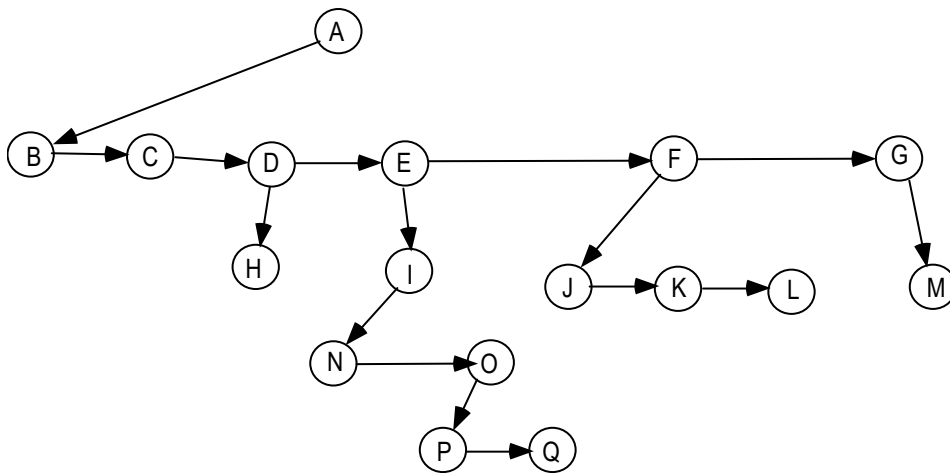
One way to implement a tree is to have a link to each sub tree in the node besides the data. But since trees can have any number of children it is more feasible to keep the children of each node in a linked list. The structure is:

```
struct tree_node
{
    datatype          data;
    tree_node         *first_child;
    tree_node         *next_sibling;
}
```

For example this is a tree.



This is the first/next sibling representation of the tree above.



Note, the tree is conceptually the way it is in the first figure. So the path from A to I is still A, E, I and is of length 3. The path is not A, B, C, D, E, I as the second figure may indicate. The second figure is only the way we implement the tree and not the conceptual tree.

14.3 Binary trees

A binary tree is a tree where no node can have more than 2 children. Since there are only at most 2 children we can have a direct link to each child. We can use

```

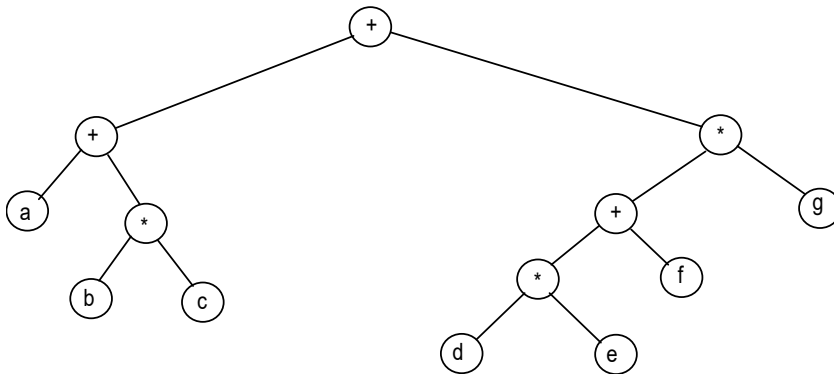
struct binary_tree_node
{
    datatype                data;
    binary_tree_node *Lchild;
    binary_tree_node *Rchild;
}
  
```

}

14.4 Expression trees

In an expression tree the leaves are operands and the roots are operators.

Example. An expression tree for $(a + b * c) + ((d * e + f) * g)$



To evaluate this expression we apply the operator of the root to the values obtained by recursively evaluating the left and right subtrees.

We can produce the infix expression by recursively producing the left expression then print out the operator in the root then produce the right expression. This traversal will print the expression in infix notation. It is considered to be an inorder traversal.

An alternative traversal strategy is to recursively print the left subtree the right subtree then the root. This will produce a postfix expression and the traversal is called a postorder traversal. The expression will be

$a\ b\ c\ * +\ d\ e\ * f + g\ * +$

Another alternative traversal strategy is to recursively print the root then the left subtree then the right subtree. This will produce a prefix expression and the traversal is called a preorder traversal. The expression will be

$++\ a\ * b\ c\ * +\ * d\ e\ f\ g$

```

PostOrderPrint(    binary_tree_node    *T)
{
    PostOrderPrint(T->Lchild);
    PostOrderPrint(T->Rchild);
    printf( "%d ",T->data);
}

```

```

InOrderPrint(    binary_tree_node    *T)

```

```

    {
    InOrderPrint(T->Lchild);
    printf( "%d ",T->data);
    InOrderPrint(T->Rchild);
    }

PreOrderPrint(binary_tree_node *T)
    {
    printf( "%d ",T->data);
    PreOrderPrint(T->Lchild);
    PreOrderPrint(T->Rchild);
    }

```

We can convert a postfix expression to an expression tree.

The algorithm:

Get the next symbol

If the symbol is an operand

 Create a 1 node tree.

 Push the pointer to this tree onto the stack.

Else if the symbol is an operator

 t1 = pop

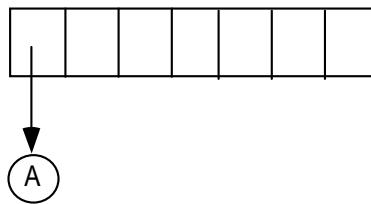
 t2 = pop

 Form a new tree with the operator as the root and t1 and t2 as its children.

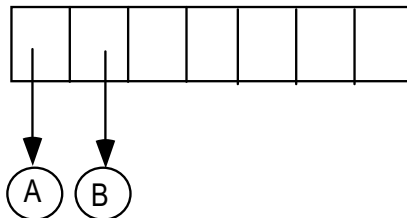
 Push the pointer to this tree onto the stack.

Example: build the binary tree for the postfix expression: A B C + *

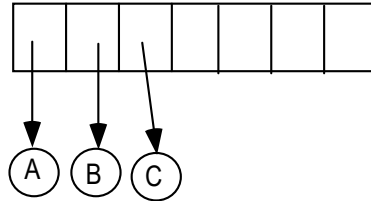
Read the A.



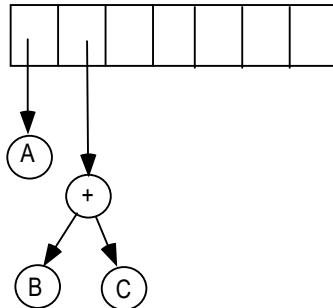
Read the B.



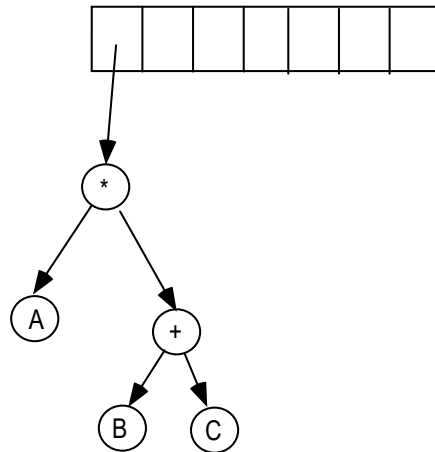
Read the C.



Read the +



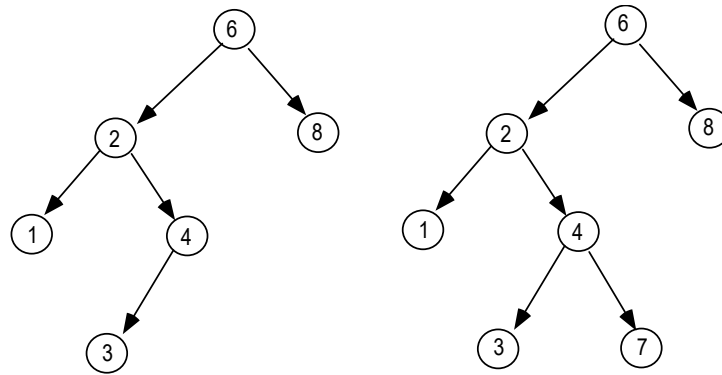
Read the *



14.5 Binary Search Trees

An important application of binary trees is in searching. A binary search tree has the following property. For every node X in the tree all of the values in the left subtree are smaller than the value of X and all of the values in the right subtree have values greater than the value of X .

The right is a binary tree but not a binary search tree. Note the 7 in the left subtree of node 6 is greater than 6. The tree on the left is a binary search tree.



Binary search trees allows us to have a data structure that permits searching for an item in $O(\log n)$ time like a binary search using a sorted array while also permitting insertions and removals in $O(\log n)$ time as in the linked list case. Recall that while a binary search in a sorted array takes $O(\log n)$ time, the insert and remove functions take $O(n)$ time. And the linked list allows $O(1)$ time for insert and remove but a linear search is required to find an item in the list.

Some of the functions that can be done to binary search trees are

- `make_empty()`
- `find()`
- `find_min()`
- `find_max()`
- `insert()`
- `remove()`

All of these routines take $O(\log n)$ time

To make a tree empty we simply traverse the tree in postorder. We need to traverse in postorder so that the root is not deleted until both children are deleted.

```

void make_empty( binary_tree_node *T)
{
    if (T != NULL)
    {
        make_empty(T->lchild);
        make_empty(T->rchild);
        delete T;
    }
}

```

To find an item in the tree we compare the root to see if the root is the item. If so we return the root. If not we search either the left or the right subtree depending whether the value we are searching is less than or greater than the root. If the element is not in the tree

then the algorithm will eventually go to a leaf and pass NULL to a recursive call. So if the function receives a NULL then the item is not found.

```
binary_tree_node *find(    binary_tree_node    *T,
                          datatype              data)
{
    if (T == NULL)
        return NULL;
    else if (data < T->data )
        return find(T->lchild,data);
    else if (data > T->data)
        return find(T->rchild,data);
    else
        return T;
}
```

To find the minimum or maximum element in a binary search tree we simply find the minimum of the left child for the minimum or the maximum of the right child for the maximum. This can be done very easily with out recursion as well.

```
binary_tree_node *find_min(    binary_tree_node    *T)
{
    if (T == NULL)
        return NULL;
    else if (T->lchild == NULL)
        return T;
    else
        return find_min(t->Lchild);
}
```

The insert function first searches for the position to insert. This is done in the second part of the algorithm. Once the position is found, a node is created and attached to the tree. Note the tree T is passed by reference so the new node created is passed back through the argument list to the variable that was passed in the previous recursion. This may be the call to insert with (*T)->Rchild for example. If this is the case, the pointer to the newly created node will be copied into (*T)->Rchild and therefore attached to the tree.

```
void insert(    binary_tree_node    **T,
               datatype              data)
{
    if (*T == NULL)
    {
        *T = new binary_tree_node;
        if (*T == NULL)
            printf("out of space\n");
        else

```

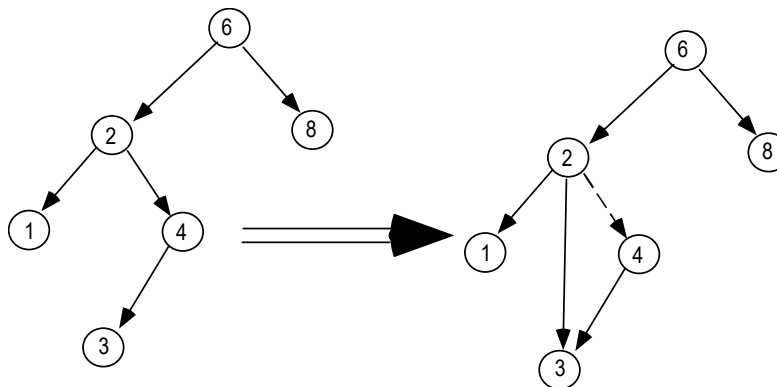


```

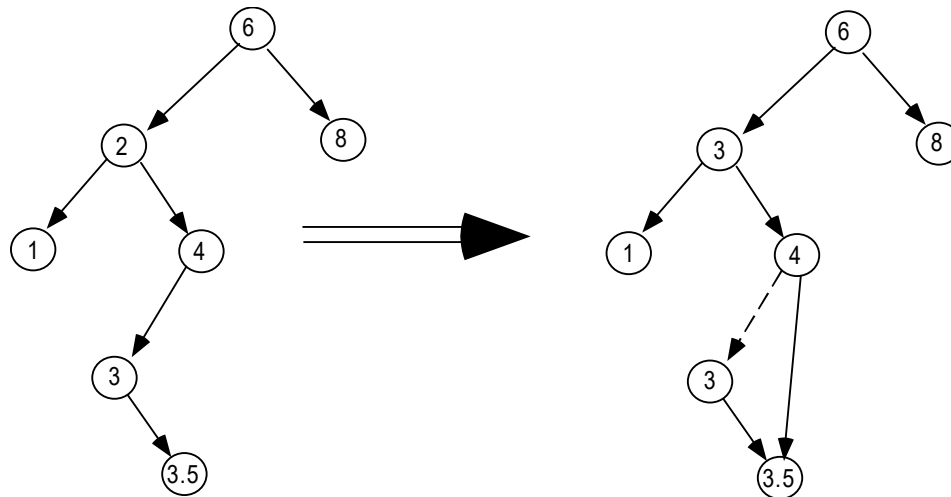
    {
        (*T)->data = data;
        (*T)->Lchild = NULL;
        (*T)->Rchild = NULL;
    }
}
else if (data < (*T)->data)
    insert( &(amp; (*T)->lchild,data ) );
else if (data > (*T)->data)
    insert( &(amp; (*T)->rchild,data ) );
// else data is already in the tree so do nothing
}

```

The remove function is the most difficult one because if you simply remove a node from the tree its children will be disconnected and lost. If the node is a leaf then one can remove it by simply deleting it and NULLing the pointer to it in its parent node. If the node has 1 child then the child will take the place of the node being deleted. So the parent will skip the node being deleted and point to its child directly. In the following example the 4 is being deleted. Its parent, the 2, is modified so that the 3 becomes its child instead of its grandchild.



If the node has 2 children then the node's data will be swapped with the data of the node that is next sequentially. This is the maximum of the left subtree or the minimum of the right subtree. And the node that got swapped in will be deleted. This node will only have 1 or no children since it is a minimum or maximum. In the following example the 2 is being deleted. Note the 2 is replaced with the next node in sequence, the 3, then the 3 is deleted.



```

void remove( binary_tree_node  **T,
             datatype          data)
{
    binary_tree_node  *temp;

    if ( (*T) == NULL)
        printf("Not found\n");

    else if (data < (*T)->data)
        remove( &(amp; (*T)->lchild ),data);

    else if (data > (*T)->data)
        remove( &(amp; (*T)->rchild ),data)

    else if ( (*T)->lchild != NULL && (*T)->rchild != NULL)
        { // The case with two children
          temp = find_min( (*T)->rchild);
          (*T)->data = temp->data;
          remove( &(amp; (*T)->rchild ), (*T)->data);
        }

    else
        { // the case with either 1 child or no children
          temp = *T;

          if ( (*T)->lchild == NULL) // only a right child
              *T = (*T)->rchild;
          else if ( (*T)->rchild == NULL) // only a left child
              *T = (*T)->lchild;

          delete temp;
        }
}

```

}

14.6 Average case analysis

Let $D(n)$ be the internal path length of some tree T with n nodes. Recall that the internal path length is the sum of the depth of all nodes in a tree. And the depth is the distance from the root to the node where the depth of the root is 0. Note, a leaf has only 1 node at level 0 so $D(1) = 0$. An n node tree has i nodes in its left subtree and $n - i - 1$ nodes in its right subtree, plus a root. $D(i)$ is the internal path length of the left sub tree with respect to its root. In the main tree all of these path lengths are 1 level deeper. The same holds for the right subtree. So we get

$$D(n) = D(i) + D(n - i - 1) + n - 1$$

Since there are $n-1$ nodes besides the root we have $n-1$ paths from the root to a node. The $n-1$ term is added to $D(n)$ to make each path 1 level longer since $D(i)$ and $D(n - i - 1)$ are in terms of their own root.

Note, all subtree sizes are equally likely, so we can have either $D(0)$ and $D(n - 1)$ or $D(1)$ and $D(n - 2)$ or $D(2)$ and $D(n - 3)$ or ... or $D(n - 1)$ and $D(0)$. That is i can be 0 or 1 or 2 or ... or $n-1$ with equal probability. Since all subtree sizes are equally likely the average value of both $D(i)$ and $D(n - i - 1)$ is

$$\frac{1}{n} \sum_{j=0}^{n-1} D(j)$$

This yields

$$D(n) = \frac{2}{n} \left(\sum_{j=0}^{n-1} D(j) \right) + n - 1$$

This is the same recurrence we will see in the section under sorting, specifically in the section under quick sort algorithm, which we showed to yield $O(n \log n)$.

So in this case this yields an average of

$$D(n) = O(n \log n)$$

so the expected depth of any node is $O(\log n)$.

This is true if all binary search trees are equally likely as was assumed above. This is not quite true however. Since the remove function always replaces the deleting node with the next largest node then deleting the next largest node and since the next largest node is in the right subtree it follows that the right subtree tends to become shorter than the left. After many insertions and removals the left subtree becomes taller than the right.

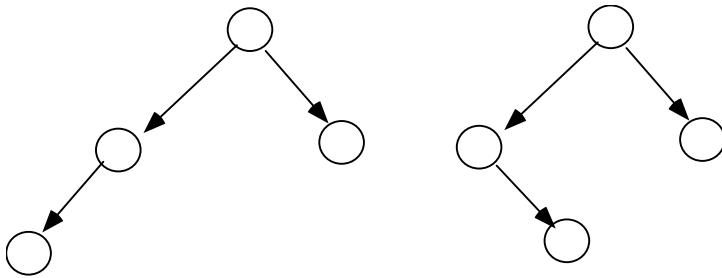
To fix this bias we can alternate between replacing the deleting node with the smallest node in the right subtree and the largest node in the left subtree. This will remove the bias but this bias does not seem to effect small trees.

We may also have the situation where the list if items being inserted into the tree is presorted then the tree will effectively become a linked list. All of the nodes will have only left or right children and searches will take $O(n)$ time. To satisfy this problem we may insist in a balancing condition.

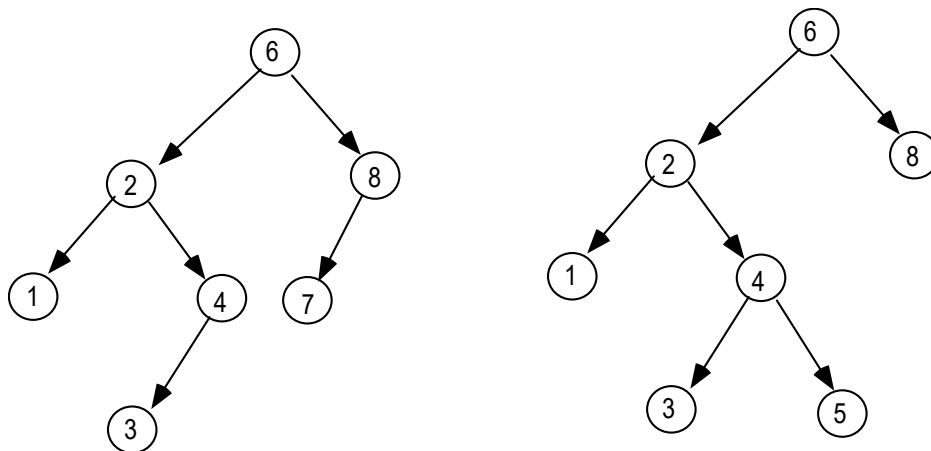
Chapter 15 AVL Trees

15.1 Introduction

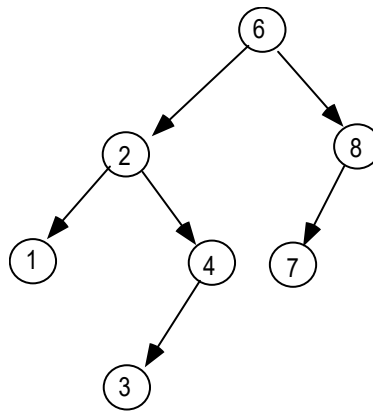
The AVL (Adelson-Velskii and Landis) tree is a binary search tree with a balancing condition. It insures that the depth of the tree is $O(\log n)$. One simple condition is to require that each node have subtrees of equal height. This condition is too rigid since only perfectly balanced trees of size $2^k - 1$ will satisfy the condition. Imagine a tree with 4 nodes. One subtree will get 2 nodes and the other will only get 1. There is no way to balance this tree.



The condition for AVL trees is more relaxed. For every node in the tree, the height of the left subtree and right subtree can differ by at most 1. The height of an empty tree is considered to be -1. The tree on the left is an AVL tree but the tree on the right is not.



With AVL trees, the insertion algorithm must maintain the tree balanced in order for the tree to continue to be considered an AVL tree. For example if we add 6.5 to the following tree the tree will become unbalanced at node 8.

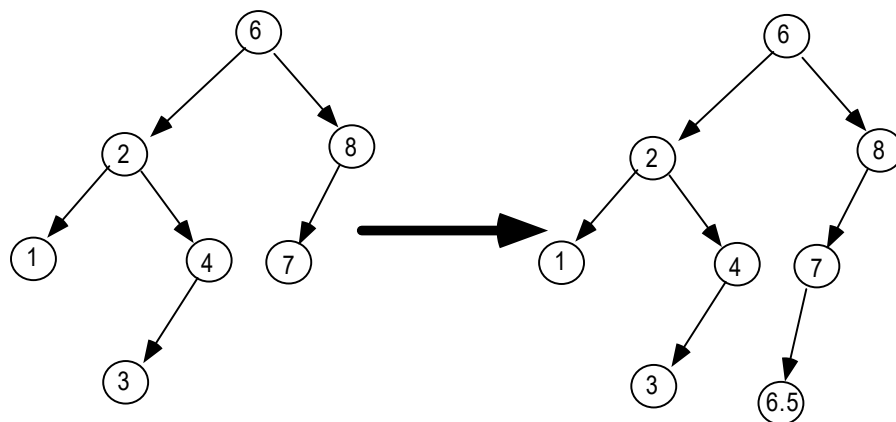


In each node, the height information is stored. After the 6.5 is inserted as a left child of the 7 the tree must be rebalanced by performing what is known as a rotation.

15.2 Single Rotation

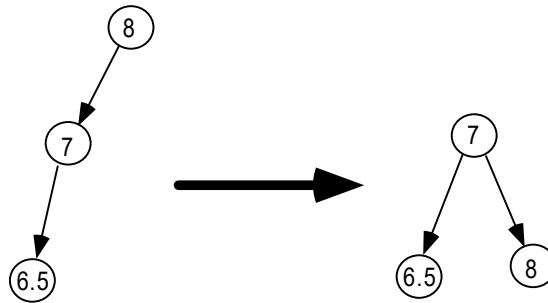
A single rotation is a way to rebalance a tree. A balanced tree has the middle number or element as the root. A tree becomes unbalanced when after adding a new element the root is no longer the middle number. Balancing a tree is the task of putting the middle element as the root. If a tree is maintained balance and only became unbalanced as a result of the most recent insertion then the middle number is only 1 node away from the root. A single rotation or a double rotation, explained later, will rebalance the tree. So a single rotation is the task of making the root's left child or right child the new root and the old root becomes a left or right child of the new root.

For example, suppose we added 6.5 to the tree above.

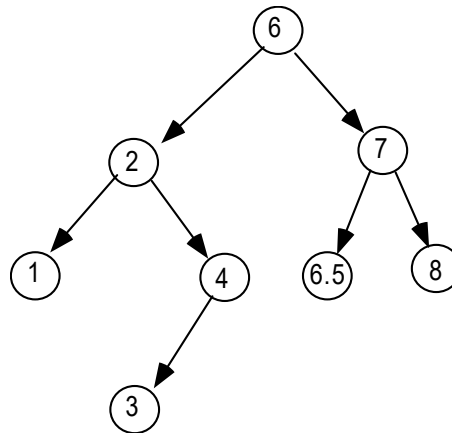


Now the tree became unbalanced at the 8. This is because the 8 used to be the middle number when compared to 7 and 8. But when the 6.5 was added the 8 is no longer the

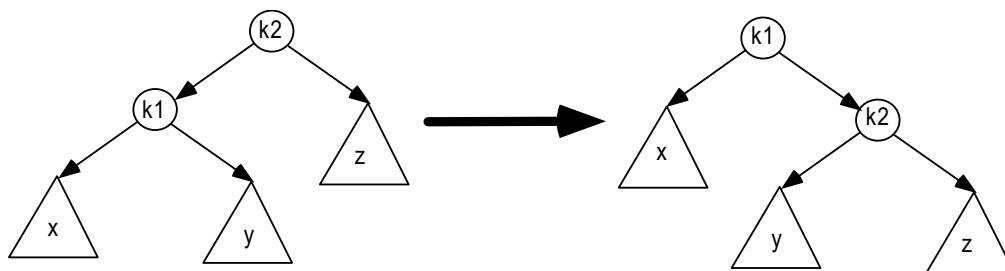
middle number. Instead the 7 is the middle number of 6.5, 7 and 8 and should therefore become the new root. So the single rotation will make the 7 the new root.



and the tree becomes:



Notice in the general case, a single rotation is:

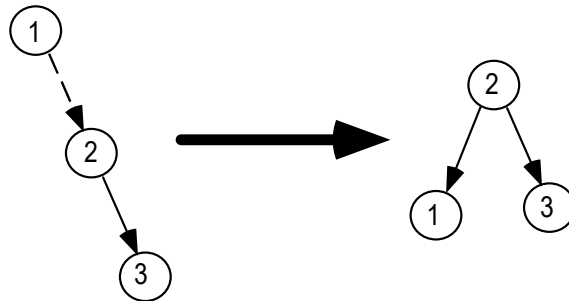


Notice that in both trees:

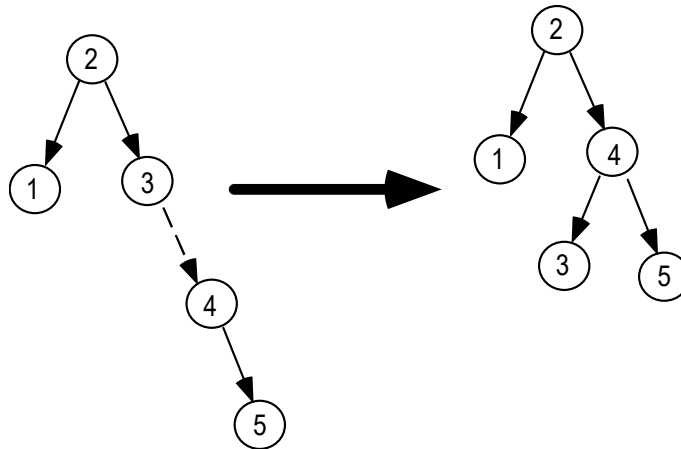
1. $k1 < k2$.
2. elements in $x < k1$.
3. elements in $z > k2$.
4. elements in $y > k1$ and $< k2$.

So the trees are both binary search trees and they represent the same list of data. Notice also that the subtree y is the right child of k_1 before the rotation and became the left child of k_2 after the rotation. This is valid since y is between k_1 and k_2 . This was needed since the right child of k_1 became k_2 and therefore the subtree y had to be moved. Since the left child of k_2 was k_1 and k_1 became the new root, k_2 is left without a left child and therefore this is where the subtree y is attached.

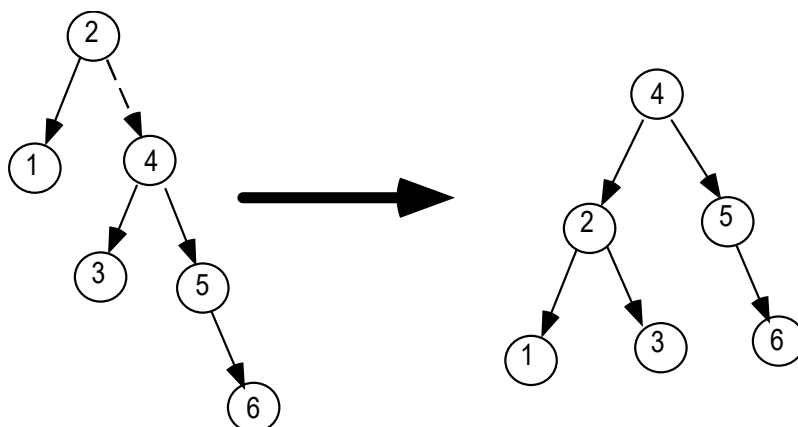
Example Lets insert the numbers 1 through 7 in sequential order. The first problem occurs when we insert the 3. The AVL property is violated at the root. We fix this by doing a single rotation between the root and its right child.



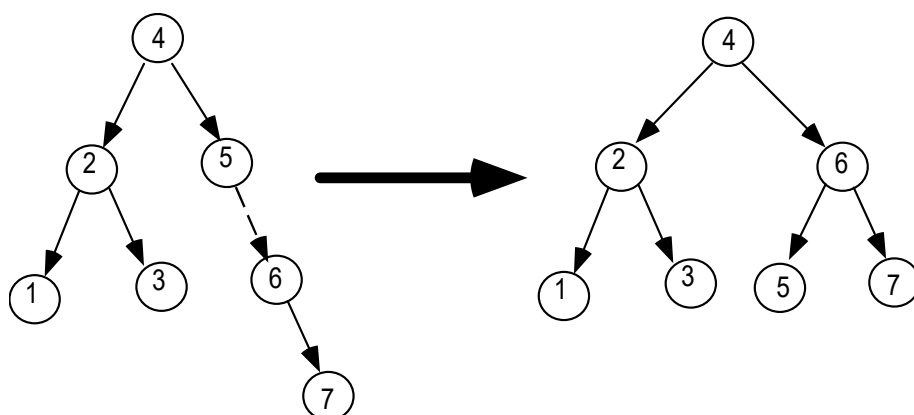
The dashed line indicates the two nodes that are being rotate. We then insert the 4 which creates no problem. Then the 5 is inserted creating a violation at node 3.



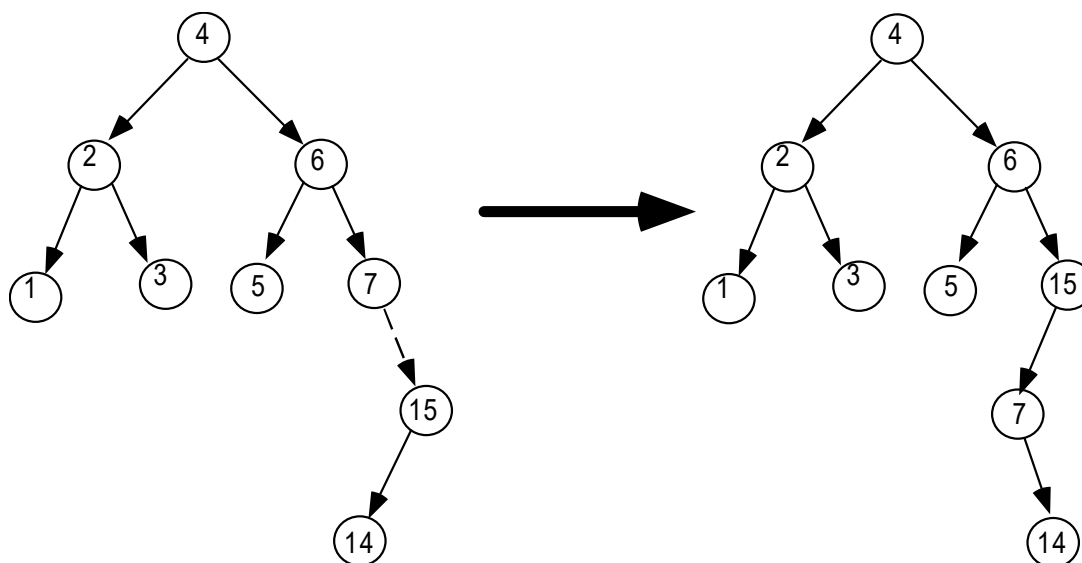
Next we insert 6 which causes a balance problem at the root. We perform a single rotation between the root and 4.



Next we insert the 7 which causes a rotation between the 5 and 6.



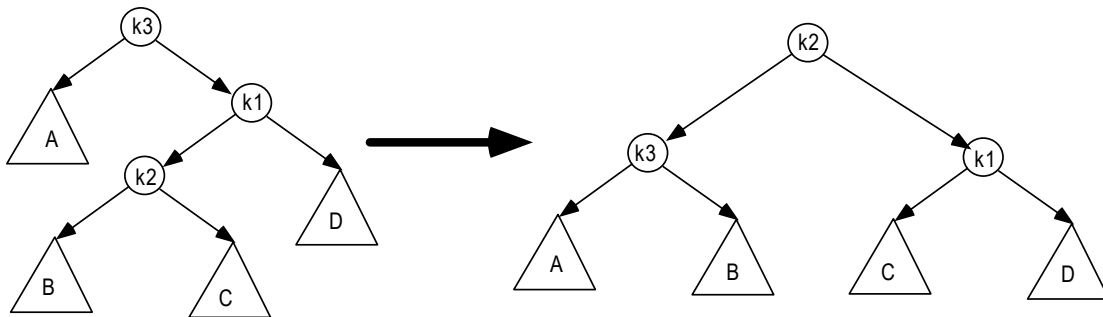
Now continuing our example suppose we insert the numbers 8 through 15 in reverse order. Inserting the 15 caused no problem but inserting the 14 caused an imbalance at node 7. We can try to fix it with a single rotation.



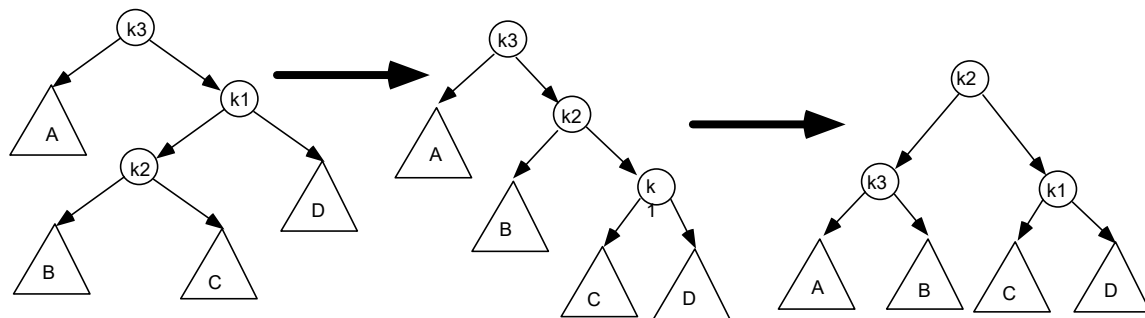
Notice that the problem was not fixed by the single rotation. This is because the node that must replace the 7 is the 14 not the 15 and the 14 is not a child of the 7 but rather a grandchild. If you start at the node that is out of balance and go along the path that goes from that node to the node inserted that caused the imbalance the first 3 nodes in the path are involved in the single rotation. The middle node is the one that becomes the new root. In this case the path is 7,15,14 and the middle node is the 14. Since the 14 is not a child of the 7 we need to move the 14 to where the 15 is at by doing a single rotation. Then do another rotation to move the 14 to the location where the 7 is at. This is called a double rotation.

15.3 Double Rotation

A double rotation is a rotation that involves 4 trees instead of 3.

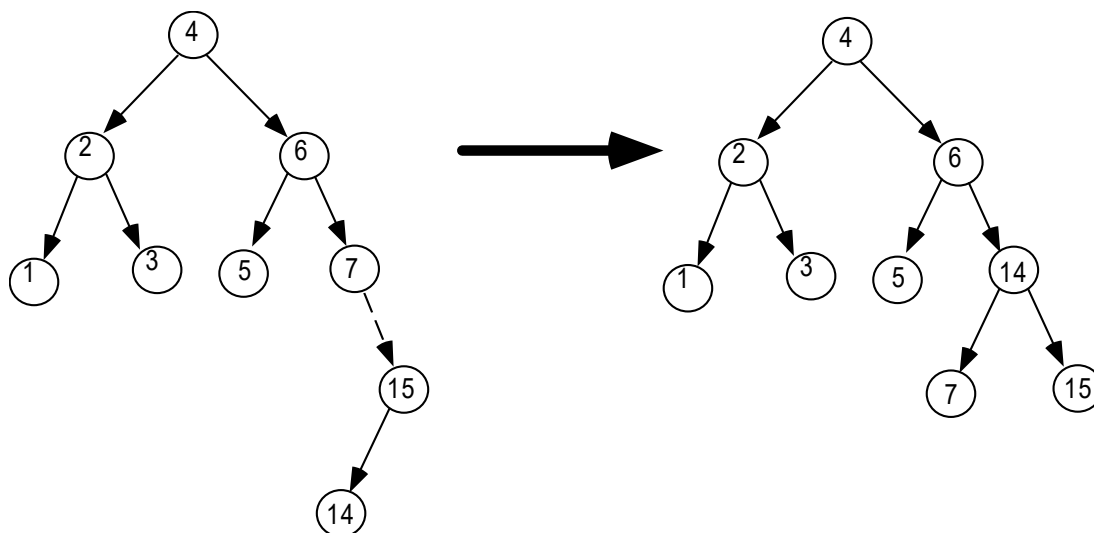


Note the node k2 is a grandchild of the root k3 not a child as in the single rotation case. A double rotation is effectively a single rotation between k1 and k2 and another single rotation between k2 and k3.

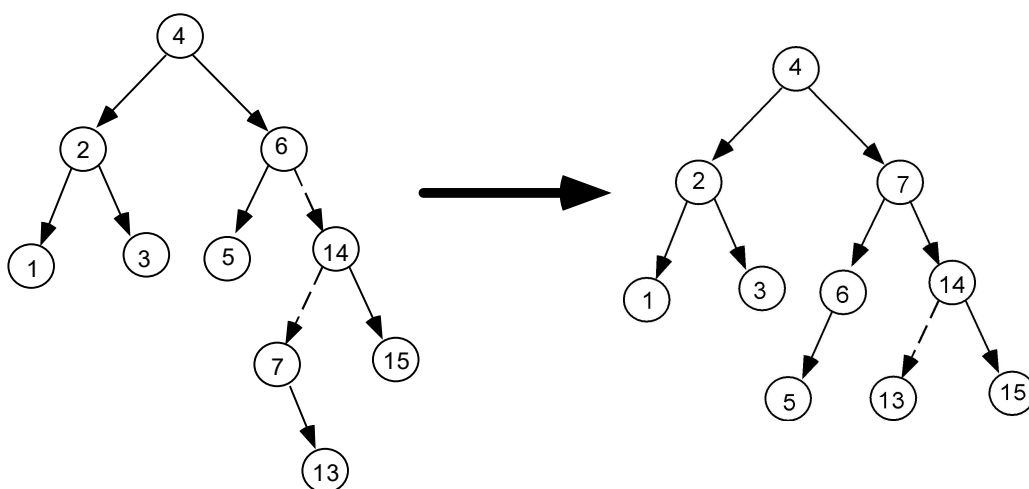


The situation where a double rotation needs to be performed instead of a single rotation is when the number inserted is between the first 2 nodes in the path from the unbalanced node towards the inserted node.

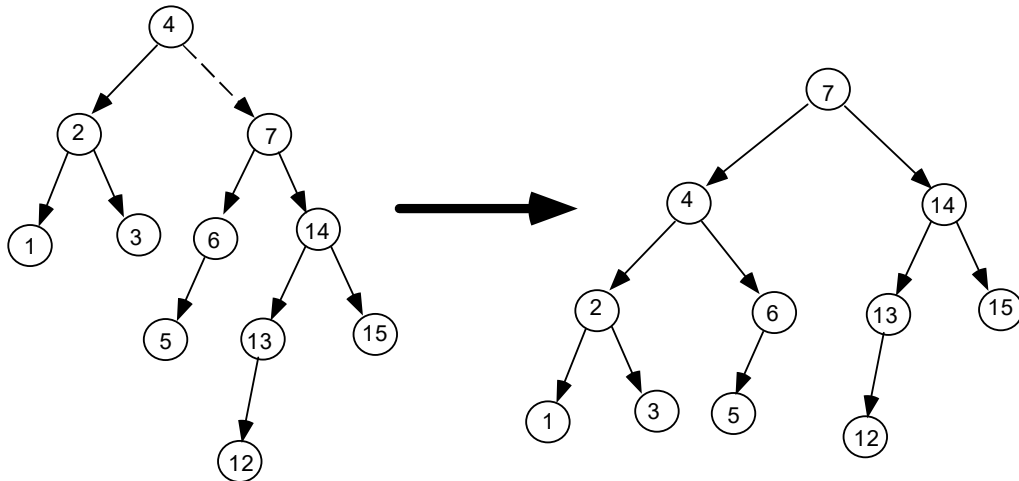
Continuing our example we just inserted the 14 and a single rotation did not fix the problem. This is because the inserted number 14 is between the 7 and 15. So a double rotation is needed instead.



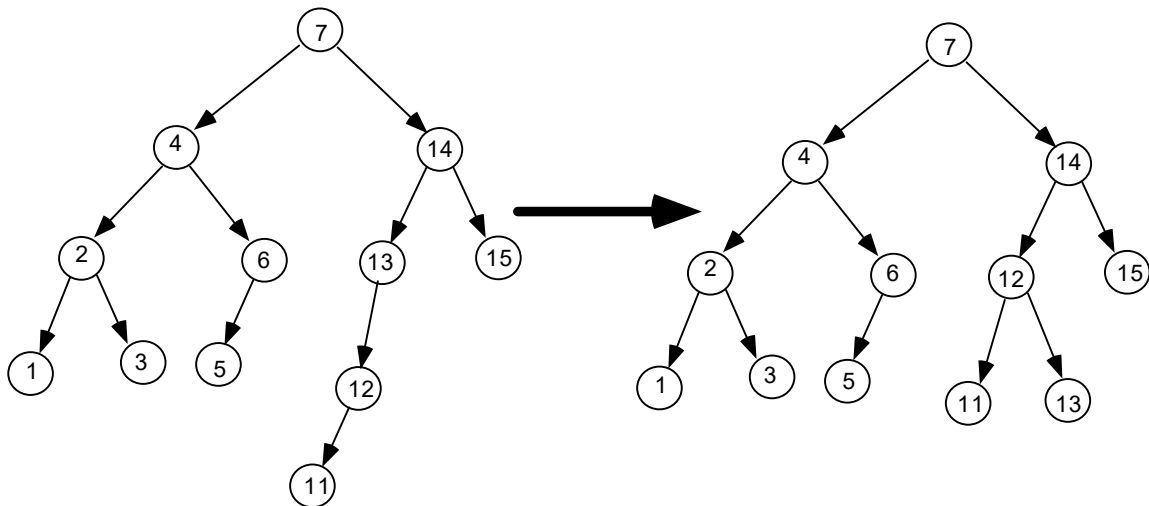
Next we insert the 13. The node 14 becomes out of balanced. This also requires a double rotation since 13 is between 7 and 14.



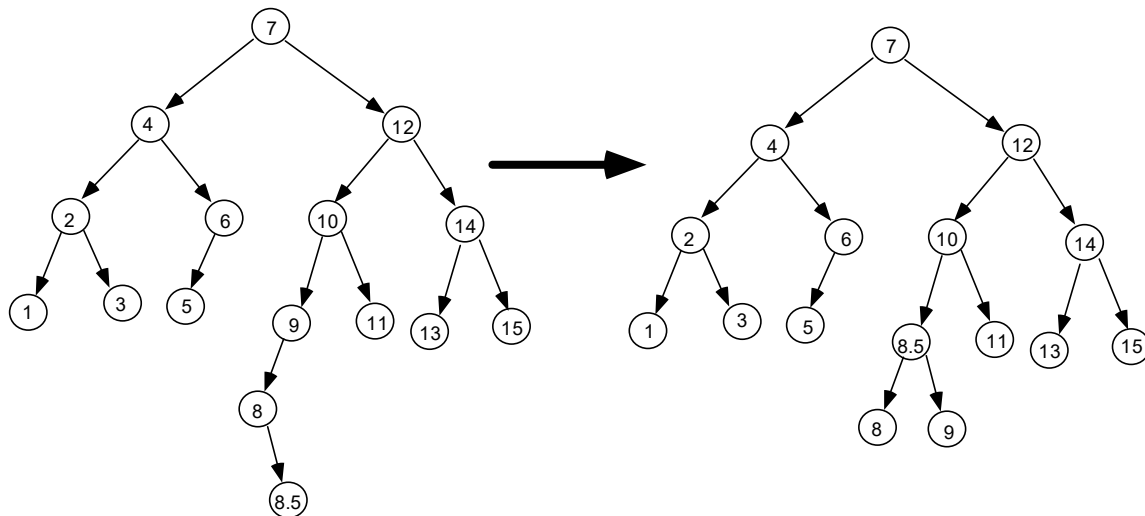
Now we insert the 12 and the root becomes unbalanced. Since 12 is not between 4 and 7 a single rotation will do.



Inserting the 11 will require a single rotation.

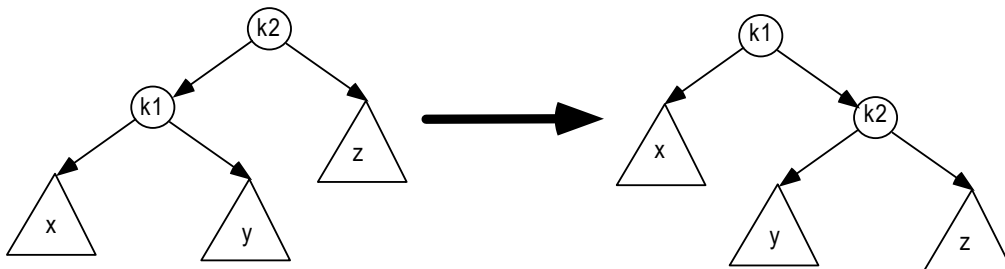


To insert 10 a single rotation is required and the same is true for the subsequent insertion of 9. We insert 8 without a rotation, creating an almost perfectly balanced tree. Finally we insert a node with 8.5 and the tree is again out of balanced at the 9. This requires a double rotation since 8.5 is between 8 and 9.



The algorithms:

The following algorithm performs a single rotation to the left. It performs the rotation between the node T and its left child as in the diagram.



```
void S_Rotate_Left(binary_tree_node **T)
{
    binary_tree_node *k1,*k2;

    k2 = *T;
    k1 = k2->Lchild;

    k2->Lchild = k1->Rchild; // k1->Rchild is Y in the diagram.
    k1->Rchild = k2;

    k2->height = max( node_height(k2->Lchild) , node_height(k2->Rchild) ) + 1;
    k1->height = max( node_height(k1->Lchild), k2->height) + 1;

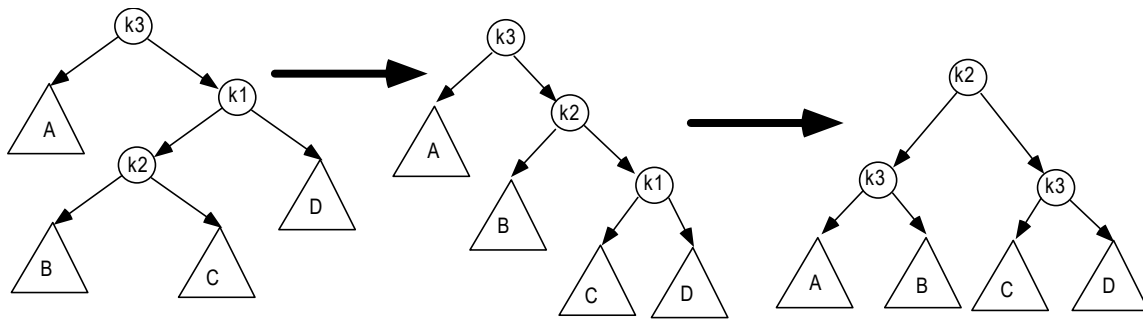
    *T = k1;
}
```

Where the function `node_height` exists only because its argument may be null. If this `T->height` were used instead of `node_height(T)` then a check to see if T is null will have to

be done also. Also note that the height of an empty tree, $T == \text{NULL}$, is -1 and the height of a leaf, a tree with only 1 node, is 0.

```
int    node_height( binary_tree_node    *T)
{
    if (T != NULL)
        return T->height;
    else
        return -1;
}
```

The following algorithm performs a double rotate right. It performs the double rotate by performing two single rotates as in the diagram.



```
void D_Rotate_Right(binary_tree_node    **k3)
{
    {
        S_Rotate_Left( &(amp; (*k3)->Rchild ) );
        S_Rotate_Right (k3);
    }
}
```

The functions to perform a single rotate to the right and a double rotate to the left are not shown since they are very similar.

The following function inserts a node into an AVL search tree.

```
void insert (    binary_tree_node    **T,
               datatype                data)
{
    if (*T == NULL)
    {
        *T = new binary_tree_node;
        (*T)->Lchild = NULL;
        (*T)->Rchild = NULL;
        (*T)->data = data;
    }
}
```

```

else if (data < (*T)->data)
{
    insert ( &(*T)->Lchild, data);
    if (node_height( (*T)->Lchild) - node_height( (*T)->Rchild) == 2)
        if (data < (*T)->Lchild->data)
            S_Rotate_Left( T );
        else
            D_Rotate_Left ( T );
    else
        calculate_height( *T );
}
else if (data > (*T)->data)
{
    insert ( &(*T)->Rchild, data);
    if (node_height( (*T)->Rchild) - node_height( (*T)->Lchild) == 2)
        if (data > (*T)->Rchild->data)
            S_Rotate_Right( T );
        else
            D_Rotate_Right ( T );
    else
        calculate_height( *T );
}
}

```

Where calculate_height is:

```

void calculate_height( binary_tree_node    *T)
{
    T->height = max( node_height(T->Lchild), node_height(T->Rchild) ) + 1;
}

```

15.4 Lazy Deletions

Note that to delete a node will require a more complicated algorithm. Even if the tree were not to be maintained balance it will still be complicated to remove a node and maintain the tree in a binary search tree organization. So what is done is what is called lazy deletions. Here the node is not removed but rather simply marked as deleted. The data in the node or the key used for comparison remains in the node since this key tells the algorithms which subtree to consider.

Chapter 16 Sorting Algorithms

16.1 Introduction

16.2 Insertion Sort

This algorithm is one of the simplest sorting algorithms. It consists of moving the elements from the unordered list to another list that is maintained in sorted order. You start with an empty list and move the first element of the unordered list to the first element of the sorted list. Then you move the next element of the unordered list into the sorted list inserting it into the proper location. This continues until all of the elements in the unordered list are transferred to the sorted one. Intuitively one may search the sorted list for the proper position using a binary search. Then insert by shifting all elements to the right by one position. A better way is to start at the right of the sorted list and move to the left until the proper position is found. Each time shifting the element in the array to the right so that when the proper position is found the space to insert is created. Furthermore only one list is used. The left part of the list is maintained sorted while the right part is unsorted. Elements are moved from the right part of the list to the proper position in the left part. As the left side grows the right side shrinks.

Algorithm:

```
InsertionSort( int    A[ ],
               int    n)
{
    int    p,t,j;

    A[ 0 ] = min_val( );
    for ( p = 2; p <= n; p++)
    {
        t = A[ p ];
        for (j = p; t < A[ j - 1 ]; j--)
            A[ j ] = A[ j - 1 ];
        A[ j ] = t;
    }
}
```

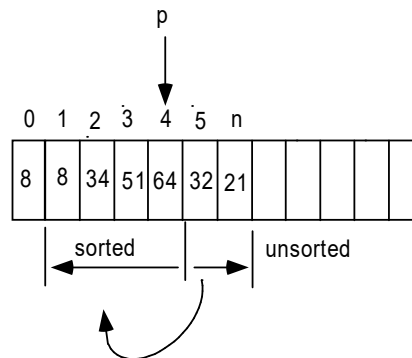
Example:

Let $n = 6$ and $A =$ the array with $\{34, 8, 64, 51, 32, 21\}$

then p goes from 2 to 6. The table below shows the array after each iteration.

original	34 8 64 51 32 21	positions moved
After $p = 2$	8 34 64 51 32 21	1
After $p = 3$	8 34 64 51 32 21	0
After $p = 4$	8 34 51 64 32 21	1
After $p = 5$	8 32 34 51 64 21	3
After $p = 6$	8 21 32 34 51 64	4

The vertical line indicates that the part to the left is sorted and the part to the right is not yet sorted.



The above is a picture of the array after $p = 4$. The next step is to make $p = 5$, get the element at position p ($=5$) and insert it into the left (sorted) side in the proper position. This is done by swapping the 32 with the 64, the 32 with the 51 and finally the 32 with the 34. Note the 8 at position 0 is there only so that the inner loop does not run passed the end of the array. It is not part of the data.

Analysis of insertion sort

the outer loop goes from $p = 2$ to n and for each p the inner loop goes from p down to 1 (in the worst case). so the worst case time complexity is

$$\sum_{p=2}^n p = 2 + 3 + 4 + \dots + n = O(n^2)$$

In the best case if the input is already sorted then the inner loop will only execute 1 time for every outer loop. This makes the inner loop execute in constant time so the time

complexity is $\sum_{p=2}^n k = k(n - 2 + 1) = O(n)$

The worst case of $O(n^2)$ is poor but the best case of $O(n)$ is excellent. So where does the average case lie? Since there is such a large difference between the worst and the best case, it is necessary to analyze the sorting algorithm more closely.

An inversion is a pair of numbers (I , j) having the property that $I < j$ but $A[I] > A[j]$. In the previous example there were 9 inversions: (34,8), (34,32), (34,21), (64,51), (64,32), (64,21), (51,32), (51,21), and (32,21). Notice that the insertion algorithm has a swap for each inversion. This is because swapping 2 adjacent elements that are out of place removes exactly 1 inversion. Since there is $O(n)$ other work involved in the algorithm the algorithm takes $O(I + n)$ time where I is the number of inversions in the unsorted array.

Theorem: The average number of inversions in an array of n distinct numbers is $\frac{n(n-1)}{4}$.

Proof:

Since the unsorted array is assumed to be in random order each pair is as likely to be out of order than it is to be in order. Since there are $\frac{n(n-1)}{2}$ total number of pairs in the list

and halve of them are inversions we get that there are $\frac{n(n-1)}{2} \cdot \frac{1}{2} = \frac{n(n-1)}{4}$ inversions in

the list. The $\frac{n(n-1)}{2}$ term comes from purely combinatorics. There are n numbers and for each number there are n-1 other numbers that can be pair up with that number. Since each pair is counter twice, the total is divided by 2.

This theorem implies that the **average** time complexity of the insertion sort algorithm is $O(n^2)$. As a matter of fact the theorem shows that any algorithm like the bubble sort and the selection sort that only perform adjacent changes is going to have the average time complexity be $O(n^2)$.

This lower bound shows us that in order for a sorting algorithm to run in subquadratic, or less than $O(n^2)$ time, it must do comparisons and in particular, exchanges between elements that are far apart. For a sorting algorithm to be efficient it must eliminate more than just one inversion per exchange.

16.3 Shell Sort

Shell sort is one of the first algorithms to break the quadratic time barrier, $O(n^2)$. It works by comparing elements that are distant. The goal is to compare and possibly swap elements that are far apart. Every time you swap an element that is say k elements apart you eliminate k inversions. This will allow the algorithm to eliminate more than 1 inversion per exchange. The idea behind this sorting algorithm is to perform an insertion sort on only certain elements in the array at once. The elements are chosen to be far apart from each other. This is where increments come into play. For a particular increment, h , all of the elements that are h elements apart from each other are sorted using a insertion sort algorithm. There may be several sub arrays with the elements being h elements apart from each other. After the array is sorted with increment h the array is said to be h -sorted and we have $A[i] \leq A[i + h]$ for all i .

For example, for an array with 13 elements, if $h = 5$ then there are 5 sub arrays. Each one is sorted individually, see the figure below. The 5 sub-arrays are the last 5 in the list of 7. The first 2 are only parts of the last 2.

Shell sort uses an increment sequence. Any sequence will do as long as the last one is 1. The shell sort with an increment of 1 is equivalent to the insertion sort. But since the list will be partially sorted the insertion sort will, if the sequence was chosen wisely, only take an average time near its lower bound of $O(n)$. The larger the increment the more inversions are eliminated per exchange. But if the increment is too large there will be very few elements in the sub array and the algorithm will be wasting time. Which sequence of increments to choose is not an easy task.

This algorithm uses the sequence of $\frac{n}{2}, \frac{n}{4}, \dots, \frac{n}{n} = 1$. It turns out that this sequence is quite poor but still performs better than simply using an increment of 1 (or insertion sort).

The algorithm:

```

ShellSort(    int    A[ ],
             int    n)
{
    int    t,
           incr,
           I,
           J;

    for ( incr = n / 2; incr > 0; incr = incr / 2;
          for ( I = incr + 1; I <= n; I++)
          {
              t = A[ I ];
              for ( J = I; J > incr; J = J - incr)
                  if ( t < A[ J - incr ] )
                      A[ J ] = A[ J - incr ];
                  else
                      break;
              A[ J ] = t;
          }
    }

```

The algorithm cycles through the inner loop once for each increment. The next loop moves I from a position of increment + 1 to n. The inside of this loop is the insertion sort sorting only the elements that are increment elements apart. It also only sorts to the left of i. This is why for I = 6 in the example the elements at position 11 is not considered until when I is 11. So the sub-array is only partially sorted then I = 6. the algorithm finishes sorting this sub-array when I = 11.

incr = 5, n = 13

81	94	11	96	12	35	17	95	28	58	41	75	15
----	----	----	----	----	----	----	----	----	----	----	----	----

81					35							
35					81							

i = 6, j = 6

	94					17						
	17					94						

i = 7, j = 7

		11					95					
		11					95					

i = 8, j = 8

			96					28				
			28					96				

i = 9, j = 9

				12					58			
				12					58			

i = 10, j = 10

35						81					41	
35						41					81	

i = 11, j = 1,6

	17						94					75
	17						75					94

i = 12, j = 2,7

		11						95				15
		11						15				95

i = 13, j = 3,8

Example.

original	81 94 11 96 12 35 17 95 28 58 41 75 15
After 5 sort	35 17 11 28 12 41 75 15 96 58 81 94 95
After 3 sort	28 12 11 35 15 41 58 17 94 75 81 96 95
After 1 sort	11 12 15 17 28 35 41 58 75 81 94 95 96

Analysis of shell sort

Worst-case analysis

The running time for the average case for this algorithm depends on the choice of the increment sequence. For this sequence the average time complexity is a long-standing open problem. We will show the worst case time analysis.

Theorem: The worst case running time for shell sort using shell's sequence is $O(n^2)$.

Proof:

The first part of the proof is to show that the running time is at least $O(n^2)$. The second part is to show that it is no more than $O(n^2)$.

To show the lower bound on the worst case imagine the following input list. The list has a length of n where n is a power of 2. That is $\exists k \text{ s.t. } 2^k = n$. The $n/2$ smallest numbers are in the even positions and the other $n/2$ number (large) are in the odd positions. Since the length is a power of 2 all of the increments are even except for the last increments which is 1. When we come to the last pass the $n/2$ largest numbers are still in the even positions and the $n/2$ smallest numbers are still in the odd positions. The i th smallest number for $i \leq \frac{n}{2}$ is in position $2i$. The last pass then must move the i th position to the

correct spot by moving it i spaces. So to move the $\frac{n}{2}$ smallest numbers to the correct

places requires $\sum_{i=1}^{\frac{n}{2}} i = O(n^2)$ moves. So the last pass alone takes $O(n^2)$ time.

The next step is to show that the running time is not more than $O(n^2)$. Let h_i be the increment i . Recall Shells increments are $h_i = \frac{n}{2}, h_{i-1} = \frac{n}{4}, \dots, h_1 = 1$. Since each time the increment is divided by 2 there are $\log_2 n$ increments. Since each pass uses 1 increment and each pass is effectively an insertion sort using part of the list, we have $\log_2 n$ insertion sorts. Since each insertion sort takes $O(n^2)$ time, intuitively one may think that the time for the whole Shell sort takes $O(n^2 \log_n n)$ time. But we can find a tighter bound. The upper bound on the worst case is actually $O(n^2)$.

Note that a pass with increment h_k consists of h_k insertion sorts each with about

$\frac{n}{h_k}$ elements. See note #1. Since the insertion sort is quadratic the total cost of a pass is

$O\left(h_k \left(\frac{n}{h_k}\right)^2\right) = O\left(\frac{n^2}{h_k}\right)$. Summing over all passes gives a total running time of

$O\left(\sum_{i=1}^t \frac{n^2}{h_i}\right) = O\left(n^2 \sum_{i=1}^t \frac{1}{h_i}\right)$. But because the increments form a geometric series with

common ration of 2, the largest term in the series is $h_1 = 1$, see note #2, therefore

$$\sum_{i=1}^t \frac{1}{h_i} < 2 \text{ and the total bound is } O(n^2).$$

Note #1:

Assume $n = 8$. For $h_1 = \frac{n}{2} = 4$ we have $h_1 = 4$ insertion sorts each with $\frac{n}{h_1} = \frac{8}{4} = 2$

elements. For $h_2 = \frac{n}{4} = 2$ we have $h_2 = 2$ insertion sorts each with $\frac{n}{h_2} = \frac{8}{2} = 4$ elements.

And for $h_3 = \frac{n}{8} = 1$, we have $h_3 = 1$ insertion sort with $\frac{n}{h_3} = \frac{8}{1} = 8$ elements.

Note #2:

Since $h_i = \frac{n}{2}, h_{i-1} = \frac{n}{4}, \dots, h_1 = 1$ we have $\sum_{i=1}^t \frac{1}{h_i} = \frac{2}{n} + \frac{4}{n} + \dots + \frac{n}{n} = 1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{\left(\frac{n}{2}\right)} < 2$

16.4 Selection Sort

In the previous 2 sorting algorithms the elements was moved or swapped several times during the algorithm. Up to now the only criteria used for selecting a good sorting algorithm was based on the number of comparisons. The time it takes to swap the two elements was a constant which is eaten by the big-O notation. However if the data to swap is very large, then the amount of work done in swapping these elements can be large. The selection sort algorithm is not necessarily a good algorithm in the sense of having few comparisons but in this algorithm the elements are only moved once if at all. This algorithm is suitable for list where the elements represent very large amounts of data and swapping two elements represents a lot of work.

The algorithm, like the insertion sort, divides the list into the sorted part on the left and the unsorted part on the right. But in order to not move an element more than once, in each pass the algorithm searches the unsorted list for the smallest element and then swaps it with the element that is just to the right of the sorted list. Note that the elements in the sorted list are the smallest elements in the list. That is all of the elements in the unsorted list are larger or equal to the ones in the sorted part of the list. For this reason once an element is swapped into the sorted list it will no longer be moved.

The algorithm:

```

SelectionSort (int    A[],
               int    n)
{
    int    p, f, t, min;

    for (p = 0; p < n - 1; p++)
    {
        min = p;
        for (f = p + 1; f < n; f++)
            if (A[f] < A[min])
                min = f;
        if (p != min)
        { // swap A[min] with A[p]
            t = A[p];
            A[p] = A[min];
            A[min] = t;
        }
    }
}

```

Analysis

The worst case and average running time of the selection sort algorithm is $O(n^2)$. Note that the algorithm is totally deterministic. That is it does the same thing regardless of the data. So the average and the worst case running time are the same.

Proof:

The first time through the inner loop ($p = 0$), the inner loop searches the whole list for the smallest number. This requires $n - 1$ comparisons. The next time through the inner loop ($p = 1$) the algorithm searches $n - 1$ elements for the smallest number requiring $n - 2$ comparisons and so on. So the running time of this algorithm is

$$(n - 1) + (n - 2) + \dots + 1 = \frac{n(n - 1)}{2} = \frac{1}{2}n^2 + O(n) = O(n^2)$$

The average running time for insertion sort was found to be $O(n^2)$ or more precisely

$\frac{1}{4}n^2 + O(n) = O(n^2)$. This is about a half of the number of comparisons required for the selection sort.

The advantage of this algorithm is the small number of swaps performed. Since elements are never moved more than once, in the worst case where every element must be moved

this will only require $n - 1$ moves. Compare this with $\frac{1}{4}n^2 + O(n) = O(n^2)$ average moves that is required for the insertion sort algorithm.

16.5 Merge Sort

The merge sort algorithm runs in $O(n \log n)$ worst case running time and the number of comparisons is nearly optimal. Merge sort and the next algorithm called Quick sort are both of divide-and-conquer type algorithms. Divide-and-conquer means the list to be sorted is divided into sub lists and the sub lists are sorted individually then merged together. Since up to now we have seen sorting algorithms that take $O(n^2)$ running time, it is faster to sort several small list than one large one. When the number of elements double the amount of work can quadruple. For example for $n = 100$, $100^2 = 10,000$ but $50^2 + 50^2 = 5,000$ and $4(25^2) = 2,500$. So dividing the list into smaller list then merging the list together greatly reduces the number of comparisons.

The basic merge sort algorithm divides the list into 2 parts of nearly equal sizes. Performs a merge sort recursively of each halve then merges the two halves together to form a completely sorted list. Each of these halves is then sorted using the same merge sort algorithm so they are too split into 2 parts. And so on until there is only 1 element in the list. At this point the merge sort simply returns the list with one element as the sorted list since any list with only one element is sorted. Most of the work is done in the merging process. However because these two list are sorted, the merge operation can be done in one pass through the list if the output goes to a third list.

The basic merging algorithm takes two input arrays a and b an output array c and three counters, ap, bp, and cp, which are initially set to the beginning of their perspective arrays. The smaller of a[ap] and b[bp] is copied to the next entry in c and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to c.

Since every comparison adds 1 element to c except for the last one which adds 2, the algorithm requires $n-1$ comparisons to merge the two lists. The time to merge two sorted lists is therefore clearly linear.

The algorithm

```
MergeSort(    int    A[ ],
              int    n)
{
    int    *TA = new int[n + 1];
    Msort(A,TA,1,n);
    delete TA;
```

```

    }

Msort(    int    A[ ],
         int    TA[ ],
         int    left,
         int    right)
{
    int    center;

    if (left < right)
    {
        center = (left + right) / 2;
        Msort ( A, TA, left, center);
        Msort( A, TA, Center + 1, right);
        Merge( A, TA, left, center + 1, right);
    }
}

Merge(    int    A[ ],
         int    TA[ ],
         int    left,
         int    right
         int    right_end )
{
    int    left_end = right - 1;
    int    temp = left;
    int    n = right_end - left + 1;

    while (left <= left_end && right <= right_end)
        if (A[left] <= A[right])
            TA[temp++] = A[left++];
        else
            TA[temp++] = A[right++]

    while (left <= left_end)
        TA[temp++] = A[left++];

    while (right <= right_end)
        TA[temp++] = A[right++];

    for ( I = 1; I <= n; I++,right_end--)
        A[right_end] = TA[right_end];
}

```

Analysis

The running time of merge sort is

$$T(1) = 1$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Note since the function is recursive, the running time function is also analyzed recursively. For $T(n)$ the algorithm has 2 recursive calls to itself each with half of the number of elements. So we have $2T(n/2)$. Then we have to merge the two lists which takes time proportional to n . So we add n .

We then divide each side by n to get

$$\frac{T(n)}{n} = \frac{2T\left(\frac{n}{2}\right)}{n} + \frac{n}{n} = \frac{T\left(\frac{n}{2}\right)}{\frac{n}{2}} + 1$$

This equation is valid for any n power of 2. So we have

$$\frac{T(n)}{n} = \frac{T\left(\frac{n}{2}\right)}{\frac{n}{2}} + 1$$

$$\frac{T\left(\frac{n}{2}\right)}{\frac{n}{2}} = \frac{T\left(\frac{n}{4}\right)}{\frac{n}{4}} + 1$$

$$\frac{T\left(\frac{n}{4}\right)}{\frac{n}{4}} = \frac{T\left(\frac{n}{8}\right)}{\frac{n}{8}} + 1$$

and so on until we get

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

Now add up all of the $\log n$ equations. This means we add all of the terms on the left-hand side and set the result equal to the sum of the terms on the right hand side. Observe

that the term $\frac{T\left(\frac{n}{2}\right)}{\frac{n}{2}}$ appears on both side of the equation and so they cancel. In fact almost all of the terms cancel. This is called telescoping a sum. The final result is

$$\frac{T(n)}{n} = \frac{T(1)}{1} + \log n \approx 1 + \log n$$

Divide by n gives

$$T(n) = n \log n + n = O(n \log n)$$

16.6 QuickSort

Quick sort is the fastest sorting algorithm used in practice. It has best and average time $O(n \log n)$. However it has worst case time $O(n^2)$. This worst case is unlikely however. The basic algorithm:

1. if the number of elements in the set S is 0 or 1 then do nothing and return.
2. Pick any element in the set $v \in S$. Call it the pivot.
3. Partition $S - \{v\}$ such that $S_1 = \{x \in S - \{v\} : x \leq v\}$ and $S_2 = \{x \in S - \{v\} : x \geq v\}$.
4. return $Qsort(S_1), v, Qsort(S_2)$

Choosing the pivot

1. Choose the first element of the array as the pivot. This works if the elements in the list are random. However it may lead to poor results if they are not. If the list is sorted then the pivot will allways be the smallest element. Using this pivot value leads to partitioning all of the elements into one sub list. Since the list is not being partitioned the time complexity goes to $O(n^2)$.
2. Choose an element at random. This is ideal since on average the partitions will be of equal length, but random number generators are very time consuming and therefore this method is better but still too slow.
3. Choose the median of the list. This is a good pivot but like random numbers, computing the median of the list will require $O(n)$ work.
4. A good way is to choose the median of the first middle and last elements.

Partitioning strategy

First swap the pivot with the last element to get the pivot out of the way.

Goal: We want to get all of the elements that are smaller than the pivot to the left of the array and all of the elements greater to the right.

The basic algorithm. Assume all elements are distinct:

1. Set I to the left of the array and J to the right - 1.
2. While $I < J$ move I to the right skipping over elements that are smaller than the pivot.
3. While $J > I$ move j to the left skipping over elements that are greater than the pivot.
4. If $I < J$ then swap the $A[I]$ with $A[j]$.
5. Swap $A[I]$ with the pivot $A[J-1]$.

What if an element is equal to the pivot? Do we skip or swap. To analyze this imagine a list where all of the elements are the same. This is not so uncommon. If you skip equal elements then J will go all the way to the left and I to the right. When the element at I is swapped with the pivot then the left partition, the small elements contains all of the elements but the pivot and the right partition is empty. Since what allows the algorithm to take $O(n \log n)$ time is the fact that the list is partition into smaller list, if all of the elements go to one partition and very few or none to the other then this will not hold and the algorithm will take $O(n^2)$ time.

On the other hand if we do not skip elements that are equal to the pivot then I and J will swap every element and they will cross each other in the middle of the list. This results in many unnessasary swaps but the list is partitioned evenly so the algorithm will take less than $O(n^2)$ time.

Sorting a list with all of the elements are equal is actually common. If you are sorting a list with 100,000 elements of which 5,000 are equal then the algorithm will eventually sort the list of 5,000 equal elements.

Cutoff limits

For list smaller than 20 elements the insertion sort algorithm work faster. So instead of partitioning until the partitions have only 0 or 1 ellement we stop partitioning when the list size goes below a limit of about 20 elements. At this point we sort the small list with an insertion sort algorithm or we leave the list unsorted. If the list is left unsorted then after running the Quick sort algorithm we must run the insertion sort algorithm to finish sorting the list. The insertion sort algorithm run very fast when the list is sorted. recal that to sort a list that is already sorted the insertion sort algorithm takes $O(n)$ time. This is far less than any other sorting algorithm sorting sorted data.

Using a cutoff in this manner saves about 15% of execution time.

The algorithm:

```
void QuickSort(    int    A[ ],
                  int    n)
{
    Qsort(A,1,n);
    InsertionSort(A,n);
}
```

```

Qsort( int    A[ ],
      int    L,
      int    R)

{
    int    pivot, I, J;

    if (R - L < 20)
        return;

    pivot = median3(A,L,R);
    I = L;
    J = R - 1;

    for ( ; ; )
    {
        while (A[++I] < pivot);
        while (A[--J] > pivot);
        if (I < J)
            swap(A[I],A[J]);
        else
            break;
    }

    swap( A[I],A[R - 1]);
    Qsort(A,L,I - 1);
    Qsort(A,I + 1,R);
}

int    median3(    int    a[ ],
                  int    L,
                  int    R)

{
    C = (L + R) / 2;

    if ( A[L] > A[C] )
        swap( A[L], A[C] );

    if ( A[L] > A[R] )
        swap( A[L], A[R] );

    if ( A[C] > A[R] )
        swap( A[C], A[R] );

    // now A[L] <= A[C] <= A[R]

```

```

swap( A[C], A[R - 1]);

return A[C];
}

```

Notice this algorithm put the pivot in the second to the last position. Since the last position is already greater than the pivot there is no need to reconsider this element in the partitioning step.

```

inline swap ( int & a,
              int & b)
{
    int    t;

    t = a;
    a = b;
    b = t;
}

```

The swap function is not actually a function. The inline command tells the compiler to place a copy of the code every place the function is called instead of placing a function call. This saves execution time since the operands and the return address do not need to be pushed on to the stack. Since the swap function is executed very often it is worth putting it inline.

Analysis

Let the number of comparisons be represented by $T(n)$ where n is the number of elements in the list.

$$T(n) = T(i) + T(n - i + 1) + Cn$$

Note since we do not know how many elements are in each partition we put I in the first and the rest of the elements $n - I + 1$ in the second partition. The Cn term is the number of comparisons needed to partition the list.

Worst case:

All of the elements excluding the pivot continuously fall into one of the two partitions. Then we have

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

We can telescope

$$T(n-1) = T(n-2) + C(n-1)$$

·
·
·

$$T(2) = T(1) + C2$$

canceling terms and summing we get

$$T(n) = T(1) + C \sum_{i=2}^n i = O(n^2)$$

Best case:

The partition is always partitions the list into to equally sized list.

$$T(n) = 2T\left(\frac{n}{2}\right) + Cn$$

to telescope we use

$$\frac{T(n)}{n} = \frac{T\left(\frac{n}{2}\right)}{\frac{n}{2}} + C$$

After telescoping we get

$$T(n) = Cn \log n + n = O(n \log n)$$

Average case:

Assume each partition size is equally likely. So each size has a probability of $\frac{1}{n}$.

$$T(i) = \frac{1}{n} \sum_{k=0}^{n-1} T(k) \text{ on average}$$

So now we get

$$T(n) = \frac{2}{n} \sum_{K=0}^{n-1} T(k) + Cn$$

$$\Rightarrow nT(n) = 2 \sum_{k=0}^{n-1} T(k) + Cn^2$$

$$\Rightarrow (n-1)T(n-1) = 2 \sum_{k=0}^{n-2} T(k) + C(n-1)^2$$

subtracting we get

$$\Rightarrow nT(n) - (n-1)T(n-1) = 2T(n-1) + 2Cn - C$$

$$\Rightarrow nT(n) \approx (n+1)T(n-1) + 2Cn$$

and we divide to telescope

$$\Rightarrow \frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2C}{n+1}$$

We telescope and get

$$\Rightarrow \frac{T(n)}{n+1} = \frac{T(1)}{2} + 2C \sum_{k=3}^{n+1} \frac{1}{k} \Rightarrow O(\log n)$$

And so

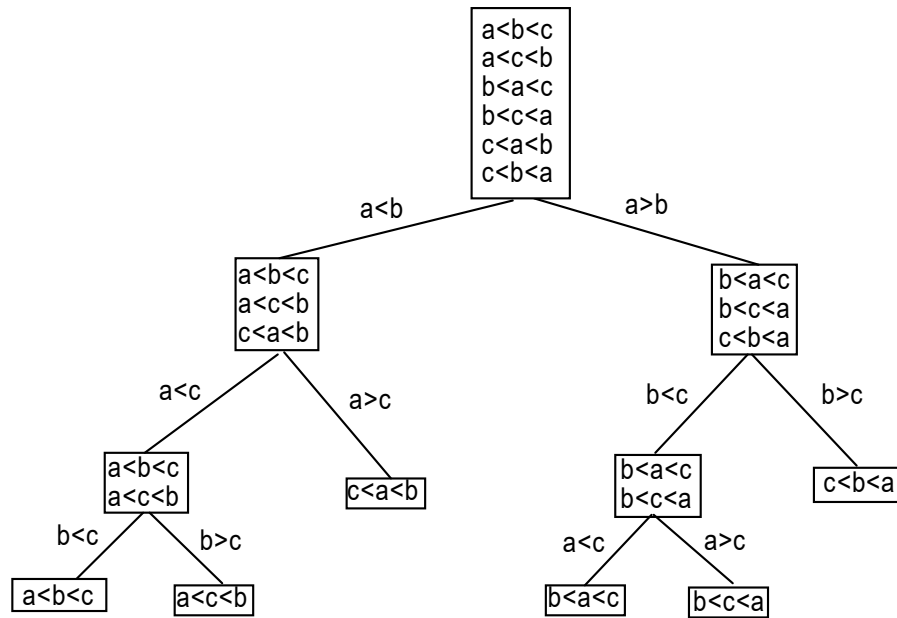
$$T(n) = O(n \log n)$$

A general lower bound for sorting.

In this section we prove that any algorithm for sorting that uses only comparisons requires $\Omega(n \log n)$ comparisons in worst case.

Note $\Omega(n \log n)$ means that the lower bound is in the order of $n \log n$. So omega is like big-O but we can eliminate constants by making the function smaller instead of bigger. This is used for lower bounds instead of upper bounds where we use the big-O notation.

Suppose we are going to sort 3 numbers a, b, and c. The comparison tree follows:



From the decision tree we know

1. If T is a binary decision tree of depth d then T has at most 2^d leaves and
2. A binary tree with L leaves must have a depth of at least $\lceil \log L \rceil$.

We can see that the decision tree for sorting has $n!$ leaves. Since there are $n!$ different combinations with n numbers and each combination is a leaf we have $n!$ leaves. To see why we have $n!$ combinations suppose you are going to choose 1 combination. First you will choose the first number. You have n choices. Then you need to choose the second number. Now you only have $n-1$ choices since one number is missing. So you have so far $n(n-1)$ choices. To choose the third number you have $n-2$ choices and so on. Up until you have $n-1$ numbers and to choose the last one you have $n-n = 0$ or no choices since there is only one left.

So since the tree has $n!$ leaves then by lemma 2, the depth of the tree must be at least $\log n!$. Since the depth of the tree represents the number of comparisons this means to sort a list of n elements requires at least $\log n!$ comparisons.

But to simplify:

$$\begin{aligned}
 \log n! &= \log(n(n-1)(n-2)\dots(1)) \\
 &= \log n + \log(n-1) + \log(n-2) + \dots + \log 2 + \log 1 \\
 &= \log n + \log(n-1) + \log(n-2) + \dots + \log\left(n - \frac{n}{2}\right) + \dots + \log 2 + \log 1
 \end{aligned}$$

If we eliminate the $\log\left(n - \frac{n}{2} - 1\right) + \dots + \log 1$ term then the right side of the equation becomes smaller.

so we have $\log n! \geq \log n + \log(n-1) + \dots + \log\left(\frac{n}{2}\right)$

And if we make all of the terms $\log \frac{n}{2}$ then the right becomes even smaller and we get

$$\begin{aligned}\log n! &\geq \log \frac{n}{2} + \dots + \log \frac{n}{2} = \frac{n}{2} \log \frac{n}{2} = \frac{n}{2} (\log n - \log 2) \\ &= \frac{n}{2} \log n - \frac{n}{2} = \Omega(n \log n)\end{aligned}$$

So $\Omega(n \log n)$ is a general lower bound for any sorting algorithm that only uses comparisons. This shows that merge sort and quick sort are optimal to within a constant factor.

Chapter 17 Hast Tables

17.1 Introduction

The hash table ADT is used for insertions, deletions and finds. It performs these operations in constant time. It has the advantage that to find an entry only requires constant time $O(1)$. The best we can do so far is with a binary search and it takes $O(\log n)$ time. However the list is not in sorted order so we can not find the smallest or largest element and we can not print the list in sorted order like we can with a sorted list.

The hash table is an array where the entries are inserted into the array by using a key extracted from the entry itself. The key is the bucket number or component number and tells where the entry is stored or will be stored in the array. So instead of searching for an entry in the array we simply extract the key which tells where the location where the entry is stored in the array.

The function that extracts the key is called the hash function.

Example, suppose that our hash table has 10 components or buckets. Also suppose that Bill hashes to 3 and Fred to 5. That is

$\text{hash}(\text{"Bill"}) \rightarrow 3$ and $\text{hash}(\text{"Fred"}) \rightarrow 5$
then the hash table will look like

0	
1	
2	
3	Bill
4	
5	Fred
6	
7	
8	
9	

Now if we want to find Bill we simply look in the array in bucket number $\text{hash}(\text{"Bill"})$. No need to search. The hash function is a mapping between the entries and the bucket numbers.

What if I want to add Bob and $\text{Hash}(\text{"Bob"}) = 5$. This is known as a collision. When two or more entries are hashed into the same bucket a collision occurs.

We will investigate two areas. The first is the hashing function and the second some collision strategies.

Hashing functions

If the entries are random numbers then the number modulus the array size will be a good hashing function provided that the numbers are as large as the number of buckets in the array.

Let the array size be N . then this function will be $\text{hash}(x) = x \bmod N$. But what if the numbers are not random. suppose that you have a table with 50 buckets and you are going to use the person's age as the hashing key. Then if the entire table is used by a kinder garden teacher to store the 100 students records all of the entries will hash to 4 and 5. The other 48 buckets will be empty and bucket number 4 and 5 will have about 50 entries each. This will result in a very inefficient algorithm. This is a result of the data not being random. Another problem is if the array size, N , is 10,000 and the number that is used for the key is random but in a range of integers from 0 to 7 then only the first 7 buckets will be used.

So to use this hashing function the data must be random and cover the whole range of bucket numbers (0 to $N-1$).

The next hashing function is to be used with strings of characters like a person's name. Here we simply add the ASCII codes of all of the characters. so

$\text{hash}(x) = \text{sum of ASCII codes of all of the characters.}$

This has the following problems. Suppose that the table has 10,000 buckets and we want to use all of them evenly to get an efficient algorithm. Since there are 127 ASCII codes and an each name has an average of 8 characters then we will get numbers in the range of 0 to $127 * 8 = 1016$. Only the first 1016 buckets will be used. This works well with small tables.

The next hash function is to multiply the character number by each other. Since there are 27 letters in the alphabet including the space, if we get the first 3 letters we have $27^3 = 17,576$. This definitely covers all of the 10,000 buckets. However a search through the English dictionary reveals that there are only 2,851 combinations of letters used in the first three letters of every word. This means that only about 3000 of the 10,000 buckets will be used. This again is as a result of the data not being random. The first three letters in a word are not random. Combination like "aaa" and "azz" are not found.

The last hash function that is presented is a good function and is commonly used. Here the hash function is

$$\text{hash}(A[]) = \left(\sum_{i=1}^{\text{length}} A[\text{length} - i] * 32^{i-1} \right) \bmod N$$

Where length is the length of the string A[].

For a 3-letter word this will be

$$\text{hash}(A[]) = (A[2] + A[1] * 32 + A[0] * 32^2) \bmod N$$

We use the constant 32 instead of 27 for 27 letters in the alphabet because $2^5 = 32$ and we can multiply by shifting the number by 5 bits to the left.

The idea is that we use all of the letters in the string not just the first 3. If there are too many letters in the string and the amount of calculations is too large one can simple select a few of the letters to participate. For example if the records to store are named and address then one might take a few letters from the name one or two from the street name and one from the zip code.

To implement we use Horner's rule

Another way to implement

$$k_1 + 27k_2 + 27^2 k_3$$

is by

$$(27k_3 + k_2)27 + k_1.$$

This can be extended to n degree.

The algorithm:

```
int hash(char A[ ],
        int n)
{
    char *p = A;
    int returnval = 0;

    while (*p != NULL)
        returnval = (returnval << 5) + *(p++);

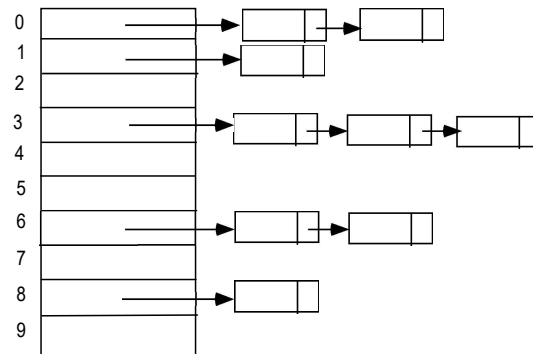
    return returnval % n;
}
```

Collision Strategies

What do we do when several entries hash to the same bucket? There are two strategies, open and closed hashing.

17.2 Open Hashing

Here we create a linked list for each bucket. If several entries hash to the same bucket all of these entries are inserted into the linked list at that bucket.



To find an entry first find the bucket where it is at by using the hash function then do a linear search in the list for the entry. Since these linked list are very short, a linear search is efficient.

The load factor is defined as

$L = \text{number of elements in the table} / \text{table size};$

so L starts at 0 and increases as the table fills. One should not have an L greater than 1. If L is 1 and the hashing function distributes the entries evenly then the linked list will have an average length of 1 node.

It is also good to make the table size, N , prime. This will help the hashing function distribute the buckets more evenly.

The other collision strategy is closed hashing.

17.3 Closed Hashing

This is an alternative to open hashing and its main advantage is the fact that linked lists are not used. This runs faster since dynamic memory allocation is slow.

In closed hashing when a collision occurs alternate buckets are tried until an empty one is found. The buckets $h_1(x), h_2(x), h_3(x), \dots$ where $h_i(x) = (\text{hash}(x) + f(i)) \bmod N$ and $f(0) = 0$ are tried in succession until one is found empty.

For closed hashing the loading factor $L \leq \frac{1}{2}$ for the algorithm to work efficiently.

We will look at three probing functions $f(x)$. The first is linear probing.

Linear Probing:

Here $f(i) = i$

In a collision the buckets are tried sequentially until an empty one is found.

The problem with this method is that clusters tend to form. Then to find an empty bucket requires a long search.

It can be shown that for linear probing the number of probes are:

$\frac{1}{2} \left(1 + \frac{1}{(1-L)^2} \right)$ for insertions and unsuccessful searches, and

$\frac{1}{2} \left(1 + \frac{1}{(1-L)} \right)$ for successful searches.

The derivations of these equations are complex.

To see how significant is the effect of clustering we will compare it to the situation where clustering does not exist. This will be equivalent to the ideal case where the buckets are probed randomly. It is easy to derive the formulas for this case. If clustering were not a problem then the formula for unsuccessful searches as well as for insertions will be:

$$\frac{1}{1-L}.$$

Since the fraction of empty cells is $1-L$, the number of cells we expect to probe is

$\frac{1}{1-L}$. This can also be seen by noting that when the table is empty $L = 0$ and

$\frac{1}{1-L} = 1$ and when $L = 0.5$, the table is half full on would expect that with random

probing if you choose a bucket randomly you will have a 50% chance of it being empty.

So you will need an average of 2 probes to find an empty bucket and $\frac{1}{1-L} = 2$ for $L =$

0.5.

Now we need to derive the formula for the cost of a successful search. The number of probes required for a successful search is equal to the number required for an unsuccessful search when the item was inserted. An insertion is done as a result of an unsuccessful search. So we can use the formula for an unsuccessful search to compute the formula for a successful search.

But L goes from 0 when the table is first used and is empty to its current value that is ≤ 1 . So the early insertions are cheaper and should bring the average down. For example say we inserted item a when $L = .2$ and now $L = .5$, to compute the cost of searching for item a we need to compute the cost of an unsuccessful search but for $L = .2$ not the current value of $.5$. Therefore to compute the cost of a successful search we need to take the average cost of an unsuccessful search from when the table was empty, ($L = 0$), to its current value.

$$I(L) = \frac{1}{L} \int_0^L \frac{1}{1-L} dL = \frac{1}{L} \ln \frac{1}{1-L}$$

So $I(L)$ is the cost of a successful search for the current value of L . These formulas are clearly better than for those with clustering. Note for $L = .75$ the linear probing will take an average cost of 8.5 probes for an insertion while only 4 probes are expected if clustering is not a problem. For $L = .9$ is even worse. With clustering 50 probes are expected while only 10 are expected with out clustering. The solution to using linear probing is to maintain $L < .5$. Note for $L = .5$ linear probing will require 2.5 probes for an insertion and only 1.5 for a successful search.

Quadratic Probing:

This is a method that eliminates the clustering problem. Here we use a quadratic probing function $f(i) = i^2$. So if a collision occur the next bucket is tested. If this collides again then the bucket that is 4 buckets away is tested followed by the bucket that is 9 buckets away if that one collides and so on. Here the loading factor L is even more critical than for linear probing. While linear probing runs inefficiently when L approaches 1, with quadratic probing there is no guarantee that a bucket will be found if the table is more than half full and this is only if the table size is prime. If the table size is not a prime number, then even with the table being less than half full the function may not find an empty bucket. For example if the table size is 16 then only alternate locations will be at distances 1, 4, and 9 away. Say we hashed to location 0. Then the alternate locations will be

$$f(1) = 1 \bmod 16 = 1,$$

$$f(2) = 4 \bmod 16 = 4,$$

$$f(3) = 9 \bmod 16 = 9,$$

$$f(4) = 16 \bmod 16 = 0,$$

$$f(5) = 25 \bmod 16 = 9,$$

$$f(6) = 36 \bmod 16 = 4$$

and so on. Suppose that locations 0, 1, 4 and 9 are the only buckets that are used, then the probing function will not find an empty bucket even though the table is only 25% full.

Theorem: If quadratic probing is used and the table size is prime, then a new element can always be inserted if the table is at least half empty.

Proof: Let N be the table size and suppose N is a prime number greater than 3. We need to show that the first $N/2$ alternate locations are distinct. Two of these locations are $hash(x) + f(i) \bmod N$ and $hash(x) + f(j) \bmod N$

where $0 \leq i \leq \frac{N}{2}$ and $0 \leq j \leq \frac{N}{2}$ and $i \neq j$.

Suppose for the sake of contradiction that these two locations are the same but $i \neq j$. Then

$$hash(x) + i^2 = hash(x) + j^2 \bmod N$$

$$i^2 = j^2 \bmod N$$

$$i^2 - j^2 = 0 \bmod N$$

$$(i - j)(i + j) = 0 \bmod N$$

Since N is prime it follows that either $i - j$ or $i + j$ is 0. Since $i \neq j \Rightarrow i - j \neq 0$ and

since $0 \leq i, j \leq \frac{N}{2} \Rightarrow i + j \neq 0$. So this is a contradiction and the first $N/2$ alternate

locations are distinct. This means that any element has $N/2$ alternate locations that it can be inserted. If at most $N/2$ locations are full this means that between the $N/2$ alternate locations and the location the function hashes to, an empty location can always be found. Note that if N is not prime then there can exist 3 numbers a , b and k such that $i - j = a$, $i + j = b$ and $ab = kN$ then $(i - j)(i + j) \bmod N = 0$.

Now if N is prime then for $k = 1$ we have $ab = N$ and this is not possible by the definition of prime. For other k if $ab = kN$ then $a = k$, $b = N$. But this implies

that $i + j = b = N$ and since $0 \leq i, j \leq \frac{N}{2}$, $i \neq j \Rightarrow i + j < N$. So this is a

contradiction.

Although quadratic probing eliminates primary clustering, elements that hash to the same location will probe the same alternate locations. This is known as secondary clustering. Simulations show this generally causes less than an extra .5 probes per search. The following technique eliminates this but is at a cost of extra multiplication.

Double Hashing:

For this method we use a second hash function to determine the alternate location. So $f(i) = i \cdot hash_2(x)$. This formula provides alternate locations at distances of $hash_2(x)$, $2hash_2(x)$, $3hash_2(x)$ and so on. One has to be careful with the choice of the second hashing function. $hash_2(x) = R - (x \bmod R)$ with R being prime and smaller than N and N being prime makes a good choice. Here N must be prime. If N is not prime than you may not find an empty bucket. For example suppose $N = 10$ and $hash_2(x) = R - (x \bmod R) = 5$ then there will be at only 1 alternative location.

Quadratic probing is generally better since it is much faster and is only worse than double probing by the fact that it has the secondary-clustering problem. Secondary clustering

does not usually cause enough problems to warrant having the extra multiplication required by the second hashing method.

Chapter 18 Heaps

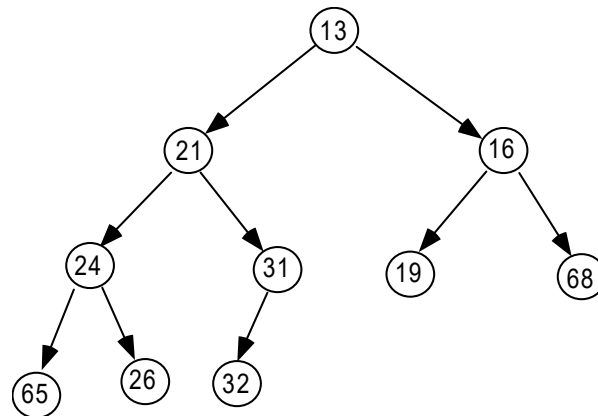
18.1 Introduction

A heap also known as a priority queue is a data structure that does the following two operations:

1. insert inserts an element and
2. delete minimum deletes the minimum element from the queue.

There are several ways to implement priority queues. One way is to store the elements in a linked list maintaining the list in sorted order. This makes finding the minimum $O(1)$ but insertions will take $O(n)$ since the element will have to be inserted in the right spot to maintain the list sorted. We could also use a binary search tree. To insert and remove will take $O(\log n)$ time. This is an improvement over simply using a simple list. Using binary search trees however is a bit of an overkill. If the list will never be traversed in order then there is no need to maintain all of the ordering that is maintained with binary search trees. A heap is similar to a binary search tree but does not have as much order. This makes the algorithms run faster since there is less structure to the tree. There is still a better way to implement priority queues.

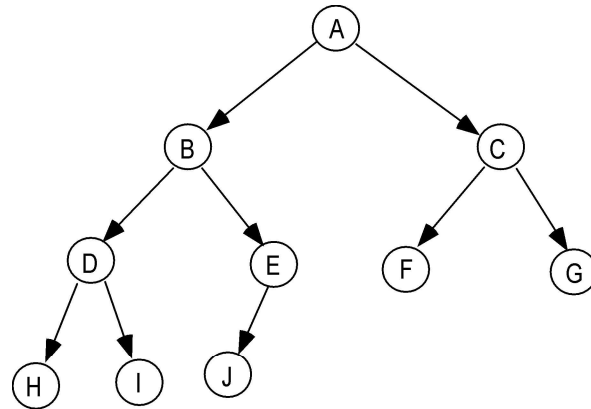
A heap is a binary tree where for each root all of the descendent have larger values. That is, both of its children have larger or equal values than itself. For example the following is a heap.



18.2 Structural Property

A heap is a binary tree that is completely filled except with the exception of the bottom level, which is filled from left to right. Such a tree is known as a complete binary tree. A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes. This implies that the height of the tree is $\log n$ which is clearly $O(\log n)$.

A complete binary tree can be represented in an array. The following complete binary tree



is implemented in the following array.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	A	B	C	D	E	F	G	H	I	J			

For an element in position i ,

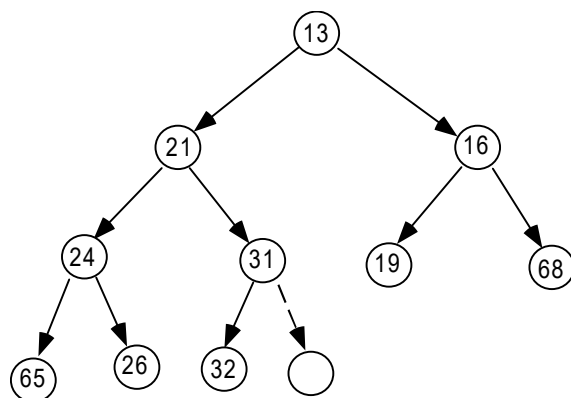
- its left child is at position $2i$,
- the right child is in the cell after at $2i + 1$ and
- the parent is at position $\frac{i}{2}$.

This implementation avoids the need to use pointers which are very time consuming.

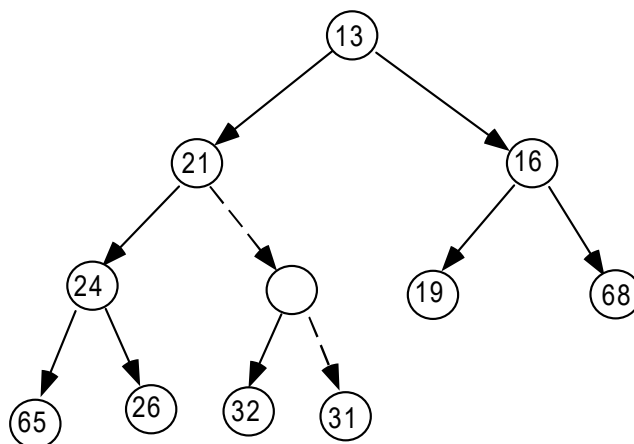
18.3 Insert Operation

To insert an element in to the heap we create a hole in the next available location, since otherwise the tree will not be complete. If the element can be placed in the hole without violating the heap property then we do so and are done. Otherwise we slide the parent of the hole into the hole which causes the hole to bubble up towards the root. We continue this process until we can place the element into the hole. This process is called percolate up. The new element is percolated up the heap until the correct location is found.

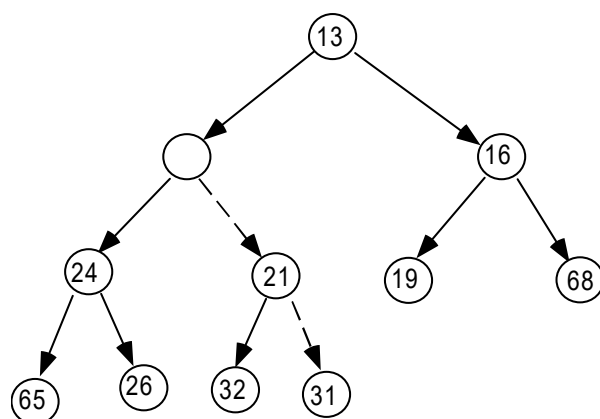
Example: Insert the number 14. We first insert a new empty node at the end of the heap.



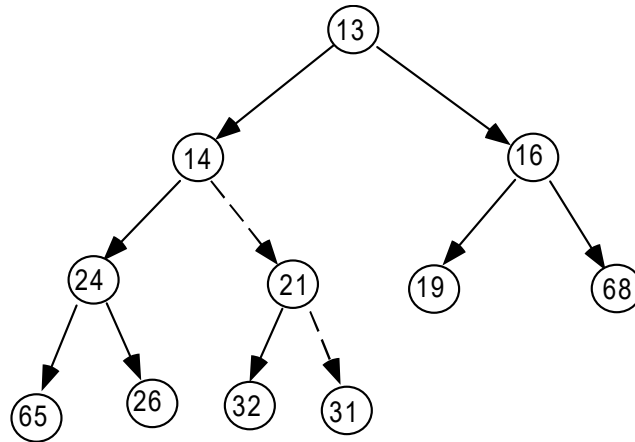
Then we check to see if we can insert the 14 into this hole. Since $14 < 31$ we can not so we slide its root, the 31, down into the hole.



Next we see if the 14 can go into the hole. Again it can not since the 21 is larger so we slide the 21 into the hole.



Now we see if the 14 can go into the hole. This time since the 14 is greater than the hole's root, the 13, we can insert the 14 into the hole. Finally we get



which maintains the heap property.

```

void insert(    int    table[ ],
              int    data)
{
    int    i;

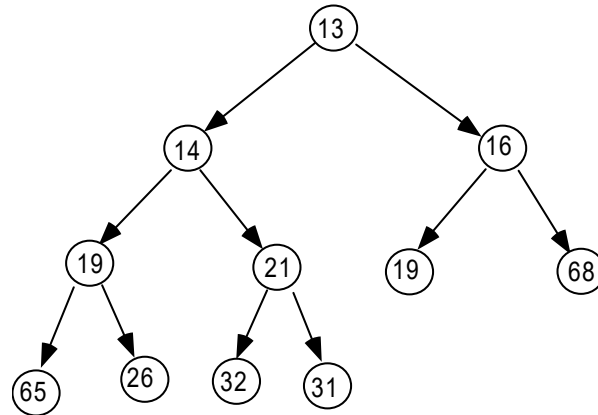
    if ( table_full(table) )
    {
        printf("The table is full \n");
        return;
    }
    else
    {
        i = ++table[0];
        while ( i > 1 && table[i / 2] > data )
        {
            table[i] = table[i / 2];
            i = i / 2;
        }
        table[i] = data;
    }
}
  
```

18.4 Remove Operation

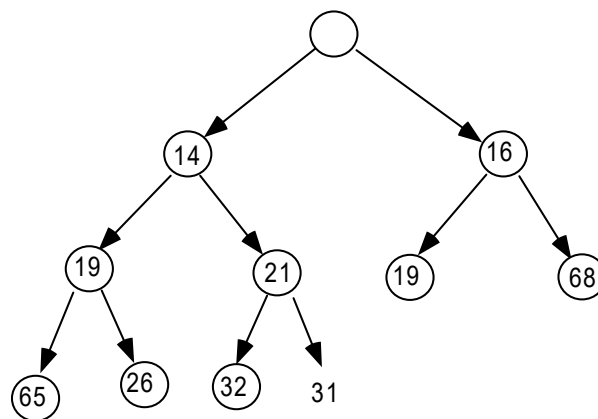
To remove the minimum we simply take the element in the root. This however leaves a hole at the root. Since the heap now becomes one smaller it follows that the last element in the heap must move somewhere in the heap. If the last element can be placed in the hole we are done. Otherwise we slide the smaller of the holes children in the hole thus

pushing the hole down one level. We repeat this step until the last element can be pushed into the hole. This strategy is called percolate down.

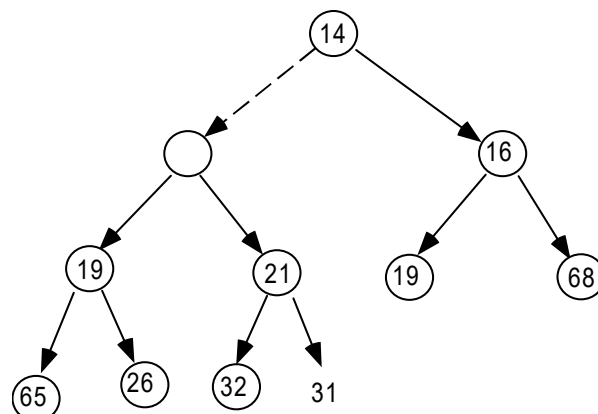
Example: remove the minimum. The minimum is at the root and is 13.



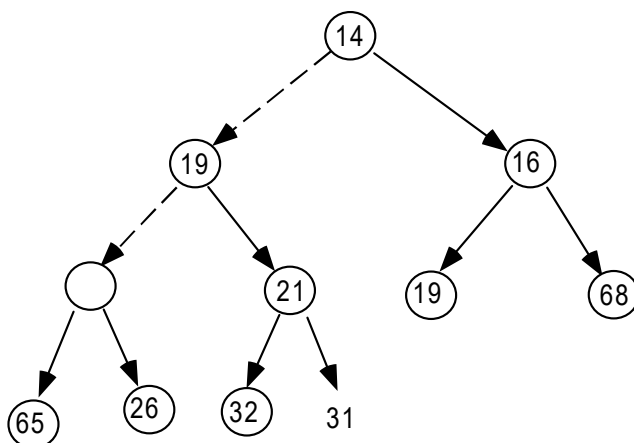
Once the 13 is removed we have a hole at the root. Also the last position in the heap must be moved into the heap. This corresponds to the 31 that lost its spot in the heap.



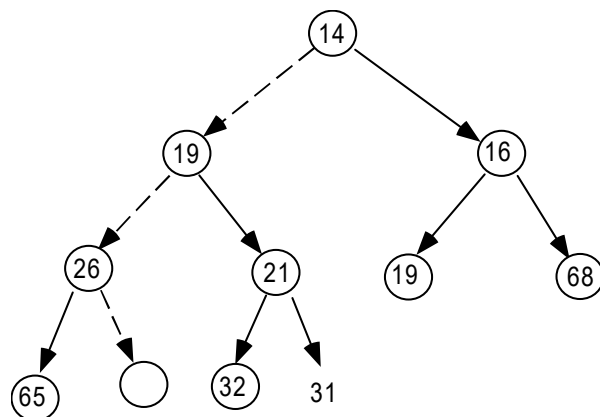
Since the 31 can not be moved into the hole we move the smaller child of the hole, the 14, into the hole.



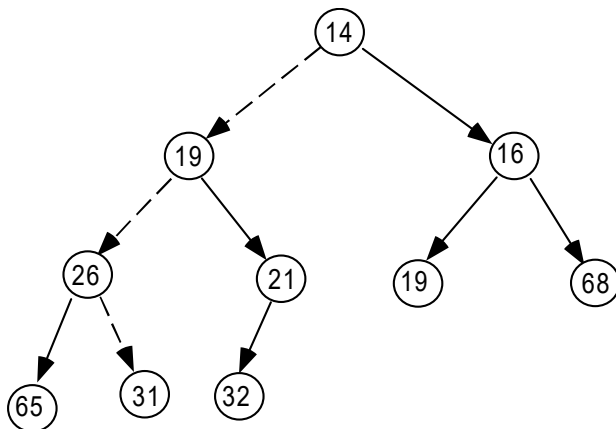
We repeat this again placing the smaller child of the hole, the 19, into the hole.



Next since the 32 still can not be placed into the hole we repeat by placing the 26 into the hole.



Now we can place the 31 into the hole without violating the heap property.



The algorithm:

```

int  remove_min( int    table[ ])
{
    int    i,
           child,
           last_element,
           size,
           hold;

    if (empty ( table ) )
    {
        printf("The heap is empty\n");
        return 0;
    }
    else
    {
        hold = table[1];
        size = table[0]--;
        last_element = table[ size ];
        for (i = 1; i *2 <= size; i = child)
        {
            // find smaller child
            child = i * 2;
            if (child != size)
                if (table[ child + 1 ] < table[ child ])
                    child++;

            // percolate one level
            if ( last_element > table[ child ] )
                table[ i ] = table[ child ];
            else
                break;
        }
        table[ i ] = last_element;
        return hold;
    }
}

```

Chapter 19 Graph Algorithms

19.1 Introduction

The following are some basic graph terminology.

Definitions

A *graph* $G = (V, E)$ consists of a set of *vertices*, V , and *edges*, E . An edge is a pair (v, w) where $v, w \in V$.

A vertex v is *adjacent* to w IFF $(v, w) \in E$.

A *path* in a graph a sequence of vertices w_1, w_2, \dots, w_n such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < n$.

The *length* of a path is the number of edges in the path or $n - 1$ for the above path.

A *simple* path is a path where all vertices are distinct except for that the first and last could be the same.

A *cycle* is a simple path of length at least 1 such that $w_1 = w_n$. For directed graphs the path need not be simple.

A directed graph is *acyclic* if it has no cycles.

An undirected graph is called *connected* if there is a path from vertex to every other vertex.

A directed graph with this property is called *strongly connected*.

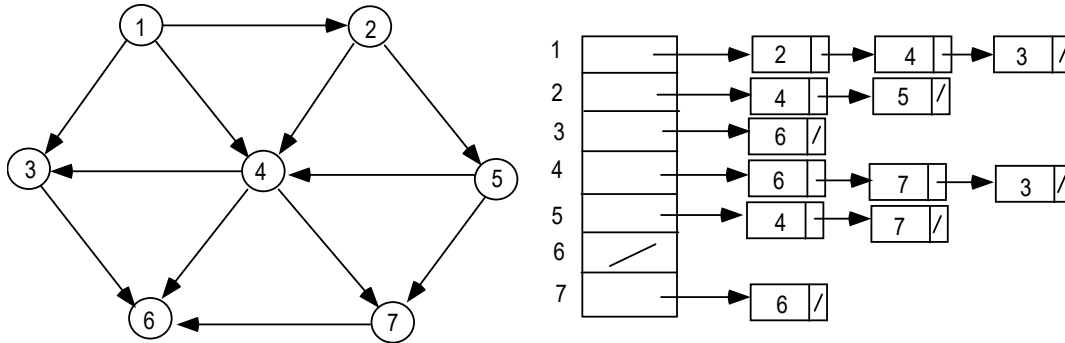
A *complete* graph is a graph that has an edge between every pair of vertices.

Representation Of Graphs

Adjacency matrix representation We can simply use a 2-dimensional array. For each edge (v, w) in the graph we set $a[v][w]$ equal to 1 or to a weight if one is associated with that edge. This uses a lot of space if the graph is not dense. Note the space requirement is $\Theta(|V|^2)$

Adjacency List representation. We have an array of pointers with a components for each vertex. The component points to a linked list of vertex numbers in which there exist an edge.

Consider the following directed graph on the left. Its representation is on the right.



Using this data structure one can find the vertices that are adjacent to a particular vertex.

19.2 Topological Sort

The topological sort of a directed graph is simply the way the graph is presented. The graph itself is not changed. The display of a sorted graph shows all of the edges going in a left to right general direction. Note that for undirected graphs the topological sort does not make sense. The leftmost vertex is the one with no edges going in to it. All of the other vertices have edges going in to it only from vertices to the left of it.

The algorithm starts by finding a vertex with no edges going in to it. We will refer to the number of edges going into a vertex as its in-degree. It processes this vertex by marking it with a number representing the order of the vertex in the sorted display. That is 1 for the first vertex 2 for the second and so on. Next the algorithm subtracts one from all of the vertices that are adjacent to the one being processed. This lowers their in-degree by one. This then causes another vertex to have an in-degree of 0 and thus become the next vertex to process.

Algorithm:

```
List Graph[MaxVertex]; // The graph
```

```
Void TopSort( Graph G)
```

```
{
  int    i,v,w, indegree[N+1];
  node   *p;

  compute indegree;

  for (i = 1; i < N; i++)
  {
    v = find_vertex_with_in_degree_0 ( indegree);
    if (v = NOT_A_VERTEX)
    {
```

```

        cout >> "The graph has a cycle\n";
        return;
    }
    Top_number[v] = i; // Record the vertex's ordering

    p = G[v];        // P is the linked list of adjacent vertices.
    While (p != NULL) // for each w adjacent to v
    {
        w = p->AdjVertex;
        indegree[w]--;
        p = p->next;
    }
}
}

```

Note this algorithm requires $O(|V|^2)$. The for loop runs $|V|$ times and the function `find_vertex_with_in_degree_0` runs another $|V|$ times. Since they are nested that yields a running time of $O(|V|^2)$.

Since the only vertices that may have an in-degree of 0 are the ones that are adjacent to the vertex being processed, it is only necessary to check those vertices in the function `find_vertex_with_in_degree_0`. The next algorithm does this by enqueueing the vertices that have an in-degree of 0 into a queue. It does this when it traverses all of the vertices adjacent to itself. Then to find the next vertex with in-degree 0 we simply dequeue from the queue. This eliminates the $O(|V|)$ involved with the function `find_vertex_with_in_degree_0`.

Void TopSort(Graph G) // A more efficient algorithm

```

{
    int    i,v,w, indegree[N+1];
    node   *p;
    Queue  Q;

    compute indegree table;

    i = 0;

    for (v = 1; v < N; v++)
        if (indegree[v] == 0)
            Q.enqueue(v);

    While ( !Q.empty() )
    {
        v = Q.dequeue();
    }
}

```

```

    Top_number[v] = i++;
    p = G[v];
    While (p != NULL)    // for each w adjacent to v
    {
        w = p->AdjVertex;
        if (--indegree[w] == 0)
            Q.enqueue(w);
        p = p->next;
    }

    if (i <= N)
        cout << "The graph has a cycle\n";
}

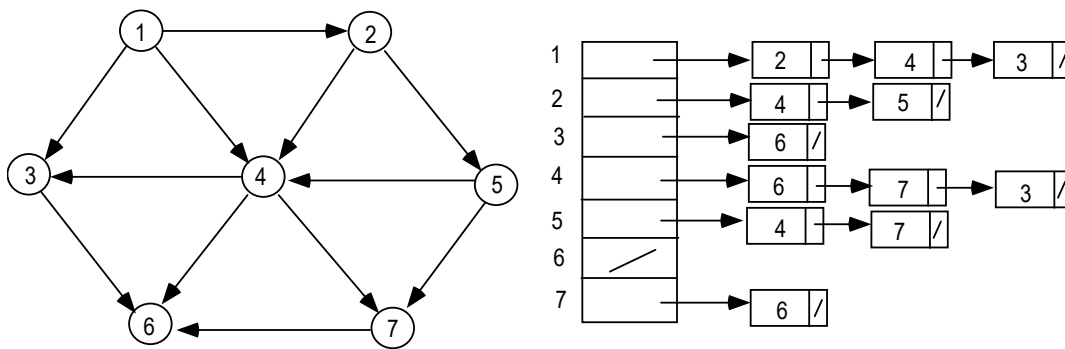
```

This now takes $O(|V| + |E|)$. The first for loop and the first while loop require $|V|$ cycles and the second while loop requires a total of $|E|$ cycles for all of the times it is executed.

Below is an example.

Example:

Consider the following graph and its representation.



The indegree array will look like

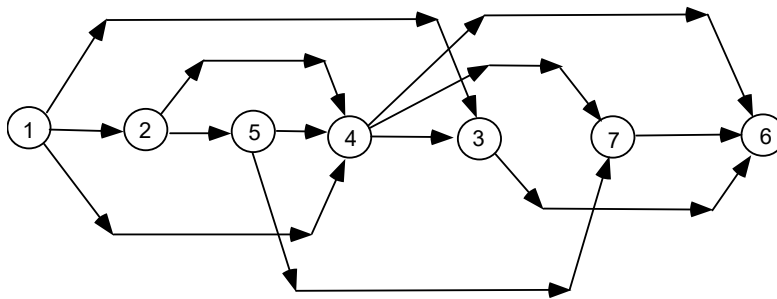
Vertex number	1	2	3	4	5	6	7
In-degree	0	1	2	3	1	3	2

This means for example vertex #3 has an in-degree of 2 or it has 2 edges going into it.

After execution the Top_number array will look like

Vertex number	1	2	3	4	5	6	7
Topological order	1	2	5	4	3	7	6

The graph may be displayed as



Notice all of the arrows are in a left to right direction.

19.3 Shortest Path Algorithms

In order to find the shortest path from a vertex to another vertex, the algorithms must compute the shortest path from a vertex to all other vertices. Therefore, the algorithm returns the shortest path from the specified vertex to all other vertices. The first algorithm finds the shortest path of an unweighted graph. This is a special case of the weighted graph problem.

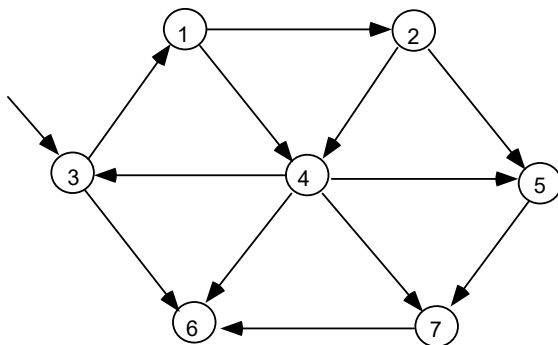
Unweighted Shortest path

This algorithm finds the shortest path from a specified vertex to all other vertices. All edges have the same weight. The algorithm works in layers. It starts by labeling all of the vertices adjacent to the specified vertex as being a distance of 1 away. Next for each vertex that has a distance of 1 it labels all of the vertices that are adjacent to this vertex as having a distance of 2. Next for each vertex with distance 2 it labels all of the vertices that are adjacent too this vertex as having a distance of 3. This process is repeated until all of the vertices are labeled. Note if a vertex is adjacent to a vertex being processed and this vertex is already labeled we do not relabel. Relabeling will in fact give it a larger distance. Also the algorithm uses a queue to determine which vertex to process next. Every time a vertex is labeled is in inserted into the rear of the queue. The next vertex to process is the one at the front of the queue.

To have a computer perform the algorithm we need to put the data pertaining to the labeling and the path into a table. The table must keep track of not only the distance but also the actual shortest path. The table will have a column for the distance and one for the path. The path is simply the vertex in the path just before reaching the current vertex. The algorithm uses a queue to determine the next vertex to process so we do not need a field to mark the vertex as processed. However for visually running the algorithm we will put a dot in the vertex to remind us that the vertex has been processed.

Example: Consider the following graph with V3 being the specified vertex.

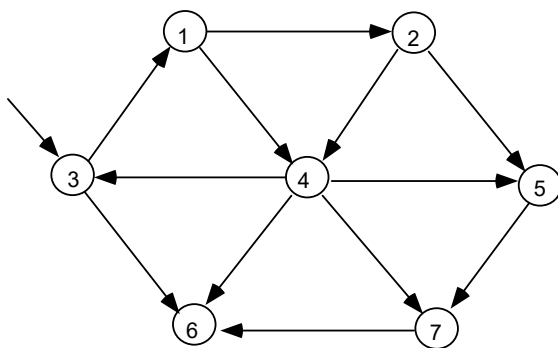
--	--	--	--	--



Vertex	Dist.	Path
V1		
V2		
V3		
V4		
V5		
V6		
V7		

We start with the specified vertex as the only vertex in the queue and initialize all distances as Inf which may simply be -1.

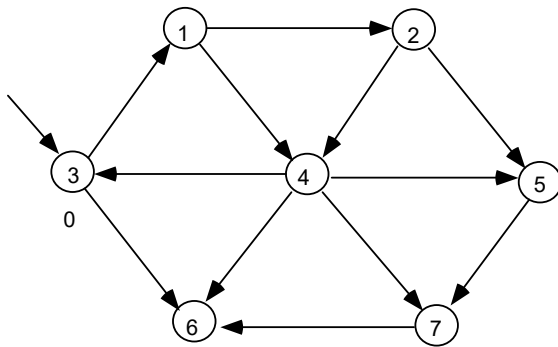
V3				
----	--	--	--	--



Vertex	Dist.	Path
V1	Inf	
V2	Inf	
V3	Inf	
V4	Inf	
V5	Inf	
V6	Inf	
V7	Inf	

We start the loop by removing the next vertex from the queue and labeling it with a distance of 0.

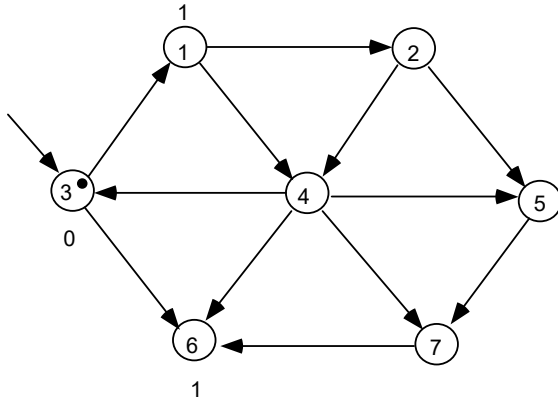
--	--	--	--	--



Vertex	Dist.	Path
V1	Inf	
V2	Inf	
V3	0	
V4	Inf	
V5	Inf	
V6	Inf	
V7	Inf	

Next we process V3 by the following: for all vertices adjacent to V3 and that are not already labeled we label them as having a distance of 1 and insert it into the rear of the queue. V1 and V6 are adjacent so they are labeled as having a distance of 1. Now we are done processing V3 so it is marked as being done. The dot in the vertex indicates the vertex has been processed.

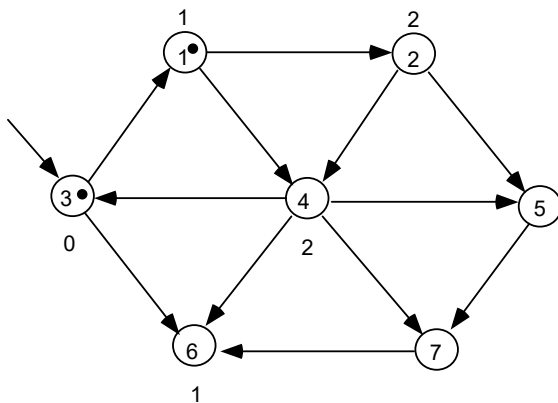
V1	V6			
----	----	--	--	--



Vertex	Dist.	Path
V1	1	V3
V2	Inf	
V3	0	
V4	Inf	
V5	Inf	
V6	1	V3
V7	Inf	

Next we remove the next vertex from the queue and process it. We process V1. For all vertices adjacent to V1 and that are not already labeled we label them as having a distance of 2 and insert it into the queue. V2 and V4 are adjacent so they are labeled as having a distance of 2. Now we are done processing V1 so it is marked as being done.

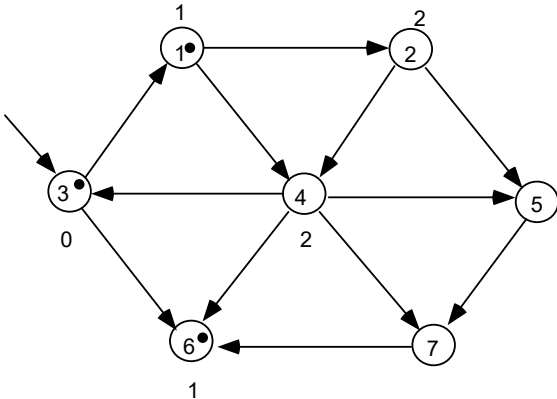
V6	V2	V4		
----	----	----	--	--



Vertex	Dist.	Path
V1	1	V3
V2	2	V1
V3	0	
V4	2	V1
V5	Inf	
V6	1	V3
V7	Inf	

We next remove V6 and process it. No vertices are adjacent to V6 so we are done with V6 and it is marked as processed.

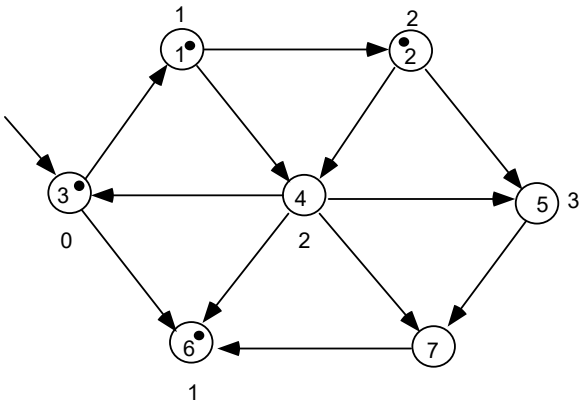
V2	V4			
----	----	--	--	--



Vertex	Dist.	Path
V1	1	V3
V2	2	V1
V3	0	
V4	2	V1
V5	Inf	
V6	1	V3
V7	Inf	

Now we are done with the layer that includes all of the vertices of distance 1 and move on to process the next layer. This layer includes all of the vertices that have a distance of 2. This includes V2 and V4. We remove the next vertex from the queue and process it. We process V2 by labeling all of the vertices adjacent to V2 that are not labeled, that is V5, as having a distance of 3 and mark V2 as done.

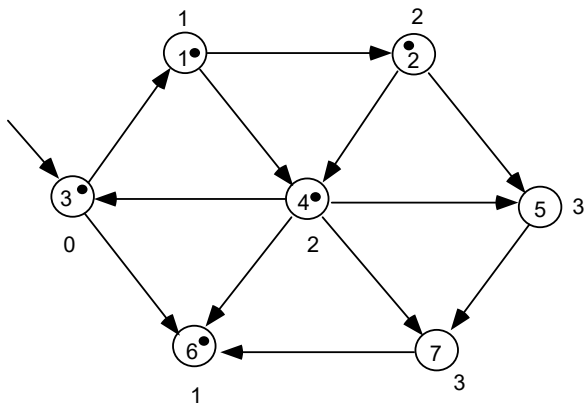
V4	V5			
----	----	--	--	--



Vertex	Dist.	Path
V1	1	V3
V2	2	V1
V3	0	
V4	2	V1
V5	3	V2
V6	1	V3
V7	Inf	

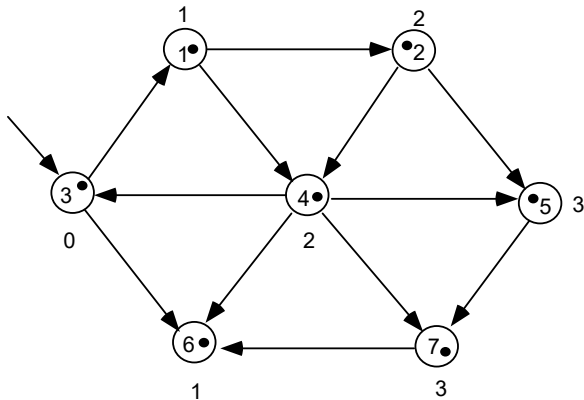
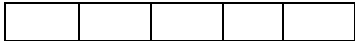
Next we process V4 by labeling all of the vertices that have not been processed or labeled with a distance of 3. Only V7 qualifies so it is labeled with a distance or 3 and V4 is marked as being processed.

V5	V7			
----	----	--	--	--



Vertex	Dist.	Path
V1	1	V3
V2	2	V1
V3	0	
V4	2	V1
V5	3	V2
V6	1	V3
V7	3	V4

We are now done with the layer of distance 2 and move on to process the layer with vertices having a distance of 3. We have V5 and V7. We remove V5 from the queue. The only vertex adjacent to V5 is V7 and it already has a label so we are done with V5. V7 only has V6 adjacent to it and it is already processed so we are done with V7 as well. Both V5 and V7 are marked it as being processed.



Vertex	Dist.	Path
V1	1	V3
V2	2	V1
V3	0	
V4	2	V1
V5	3	V2
V6	1	V3
V7	3	V4

Now there are no more vertices in the queue and the algorithm ends. Note all vertices have been processed. The table indicates the actual path. For example the shortest path from the specified vertex, V3 to say V5 is V3 V1 V2 V5. The paths are stored in reversed order. That is we can see from looking at the V5 entry that V5 came from V2. From V2 we see that V2 came from V1. And from V1 we see that V1 came from V3. We can use a recursive algorithm or a stack to display the path.

The algorithm:

Unweighted (Vertex S, Graph G);

```

{
  Queue Q;
  Table T;
  Vertex W, V, *p; // int in the range of 1 to N

  T[S].dist = 0;
  Set all others to Inf. // Maybe -1 for inf.
  Q.insert( S );

  While ( ! Q.empty( ) )
  {
    V = Q.remove();
    p = G[V];
    While (p != NULL) // for each w adjacent to v
    {
      W = p->AdjVertex;
      If ( T[W].dist == Inf) // Does not already have a path from the start
      {
        T[W].dist = T[V].dist + 1;
        T[W].path = V;
        Q.insert ( W );
      }
      p = p->next;
    }
  }
}

```

19.4 Weighted Shortest Path (Dijkstra's Algorithm)

If the graph is weighted we use Dijkstra's algorithm. This algorithm is a good example of a greedy algorithm. Greedy algorithms solve a problem in stages always doing what appears to be best at each stage. For example consider an algorithm to return change using the minimum number of coins in US currency. A greedy algorithm will first give as many quarters as it can then dimes then nickels and finally pennies. This will result in the change with the minimum number of coins. Greedy algorithms do not always work however. What if we add a 12 cents coin. Then to return change of 15 cents the algorithm will first return a 12 coin followed by three pennies. The change with the minimum number of coins is a dime and a nickel.

This algorithm differs from the previous unweighted shortest path in that it is possible that after you find a path, at some later time or layer, you find a shorter path, perhaps one with more edge but a smaller total edge weight sum. This cause the algorithm to need to reevaluate vertices that it has already given a path to. This is a greedy algorithm since it

gives vertices the smallest possible path as it moves to later layers, however, to fix the problem we saw with the coins, the algorithm may change its mind and give a new path that is shorter (less total weight). We need to change the queue to a heap since the order we remove from the queue now matters. But this is not a normal heap. Since we may update an existing path with a better one, if the vertex that got updated is in the heap it will need to percolate up given its smaller updated value. This means we need to perform percolate operations on random nodes even when we did not insert a new one. Without the queue we also need to mark the vertices that have already been processed. We cannot rely on the queue for that as we did before.

Algorithm on next page

```

struct    tablenode
{
    bool        known;
    double      dist;
    int         path
}

tablenode Table[N];

atruct    graphLLnode
{
    int         AdjVertex;
    double      Weight;
    graphLLnode *next;
}

struct    graphnode
{
    char        Name [80];
    graphLLnode *link;
}

graphnode Graph[N];

void Dijkstras( graphnode    G[ ],
               int          s,
               tablenode     T[ ])
{
    int         v,w;
    graphLLnode *p;

    init_table ( s, G, T);

    for ( ; ; )
    {
        v = SpecialHeap.remove() // smallest unmarked v

        if (v == NOT_A_VERTEX)
            Break;

        T[v].known = true;

        p = G[v].link;
        while (p != NULL)    // for each w adjacent to v
        {
            w = p-> AdjVertex;
            if ( T[v].dist + p->Weight < T[w].dist )

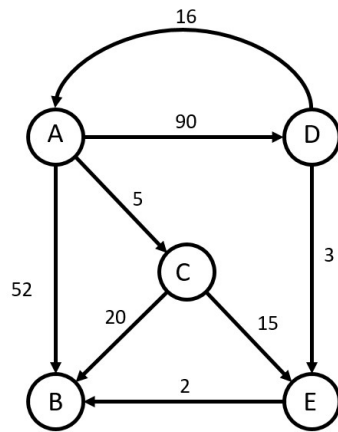
```

```

    {
        T[w].dist = T[v].dist + p->Weight;
        T[w].path = v;
        SpecialHeap.insert(w);
    }
    p = p->next;
}
}

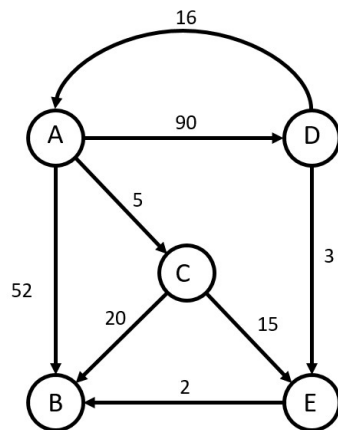
```

Example: Consider the shortest path from vertex A.



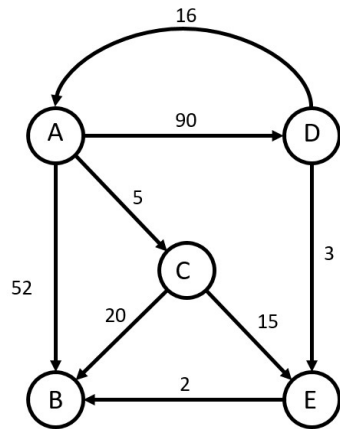
Vertex	Distance	Path
A		
B		
C		
D		
E		

We start with having the distance to A as 0 and the rest as Inf:



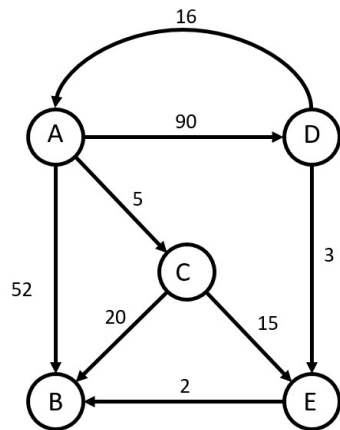
Vertex	Distance	Path
A	0	
B	Inf	
C	Inf	
D	Inf	
E	Inf	

Next we find the smallest unmarked vertex, that is A, traverse its list of adjacent vertices (B, C, D) and add or update each one with a path. We mark A as processed.



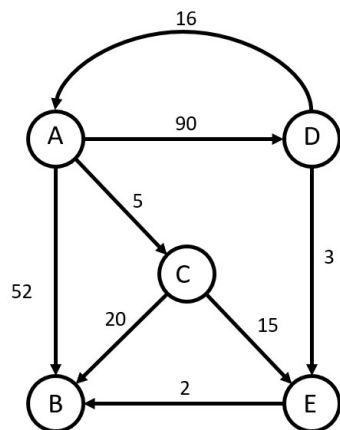
Vertex	Distance	Path
A *	0	
B	52	A
C	5	A
D	90	A
E	Inf	

Then we find the the smallest unmarked vertex again. That is C, traverse its list of adjanect vertices (B, E) and add or update each one with a path. Mark C.



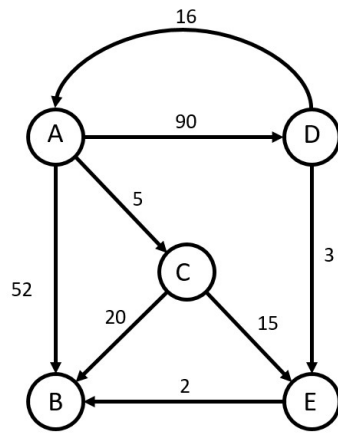
Vertex	Distance	Path
A *	0	
B	52, 25	A, B
C *	5	A
D	90	A
E	20	C

Then we find the the smallest unmarked vertex again. That is E, traverse its list of adjanect vertices (B) and add or update each one with a path. Mark E.



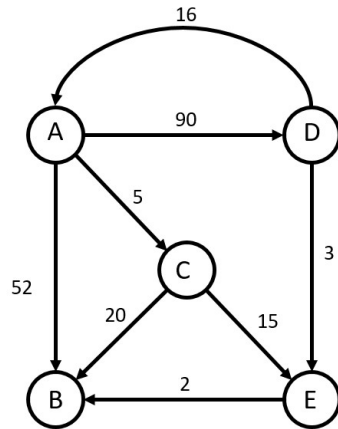
Vertex	Distance	Path
A *	0	
B	52, 25, 22	A, C, E
C *	5	A
D	90	A
E *	20	C

Then we find the the smallest unmarked vertex again. That is B, traverse its list of adjanect vertices (nothing) and add or update each one with a path. Mark B.



Vertex	Distance	Path
A *	0	
B *	52, 25, 22	A, C, E
C *	5	A
D	90	A
E *	20	C

Then we find the the smallest unmarked vertex again. That is D, traverse its list of adjanect vertices (A, E) and add or update each one with a path. Mark D.



Vertex	Distance	Path
A *	0	
B *	52, 25, 22	A, C, E
C *	5	A
D *	90	A
E *	20	C

There are no unmarked vertices so the search returns NOT_A_VERTEX which just may be -1 or something like that.

Note the algorithm changed its mind 2 times for the path to B. First it found a path of weight 52 via A, then it found a better path of 25 via C then finally it found the best path of 22 via E.

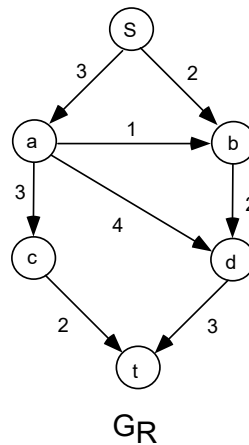
19.5 Network Flow Problems

Assume you are given a directed graph with weighted edges, if the weights are considered to be a flow capacity then the graph represents a flow network. For example the graph may represent a system of water pipes where the flow capacity represents the maximum amount of water that can pass through the pipe. The graph needs to have a vertex, s , that is the source and a second vertex, t , that is the sink. Flow then goes from

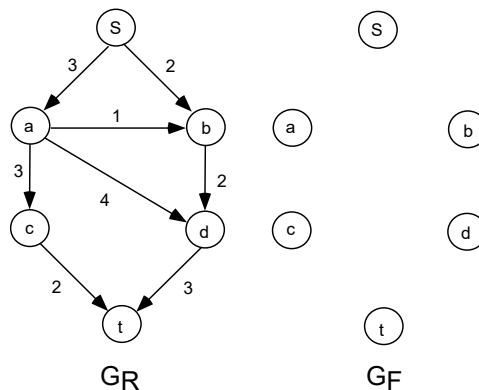
the source to the sink. The rest of vertices must have the amount of flow entering the vertex equal to the amount of flow exiting. The problem is to determine the maximum amount of flow that passes from the source to the sink.

The solution is to have two graphs, a residual graph and a flow graph. Initially the residual graph has the initial graph and the flow graph has all of the vertices but no edges. In each stage a path is found in the residual graph and added to the flow graph. The path is then subtracted from the residual graph. The path must have the same amount of flow on all its edges.

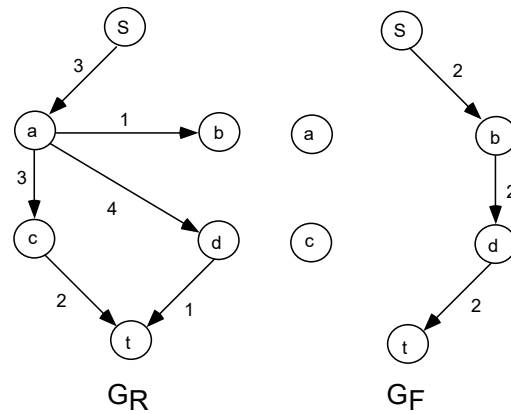
Example consider the following graph.



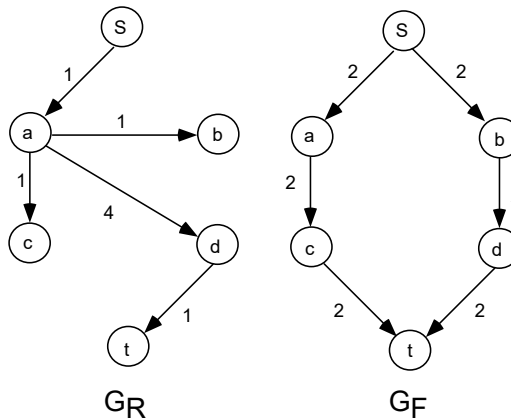
The initial condition of the algorithm will be.



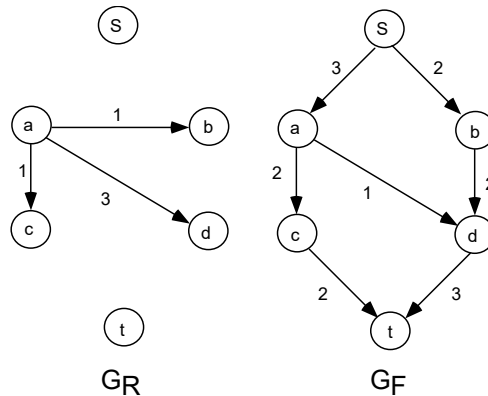
The first step is to choose a path from s to t. We choose s, b, d, t with a flow of 2 units.



Next we choose s, a, c, t with a flow of 2 units.



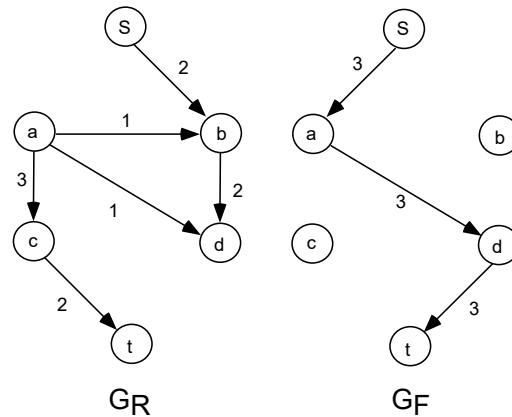
Finally we choose the only path left s, a, d, t , with 1 unit of flow.



now we stop since there are no more paths from s to t . The $\text{indegree}(t) = \text{outdegree}(s) = 5$ Units of flow. Also for all other vertices the $\text{indegree} = \text{outdegree}$.

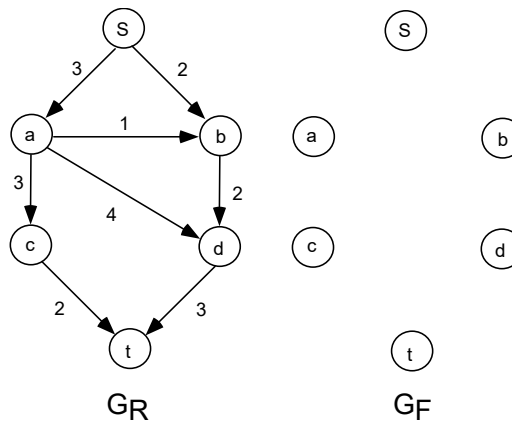
Notice that the algorithm has a choice as to which path to choose next. This type of algorithm is called non-deterministic.

What if we had chosen the path s, a, d, t , to begin with. We will get

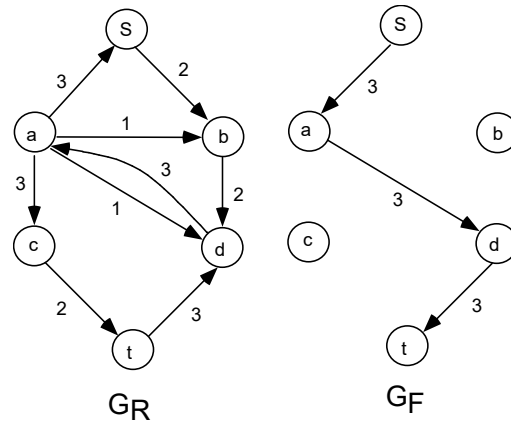


Now the algorithm stops since there are no paths from s to t in the residual graph. This was obviously a poor choice of paths. This is an example of a greedy algorithm that does not work. An improved algorithm is to allow the algorithm to change its mind and undo a flow it had previously done. We do this by putting an edge in the reverse direction is that we are subtracting. This allows the algorithm to use this path and effectively undo some of the flow.

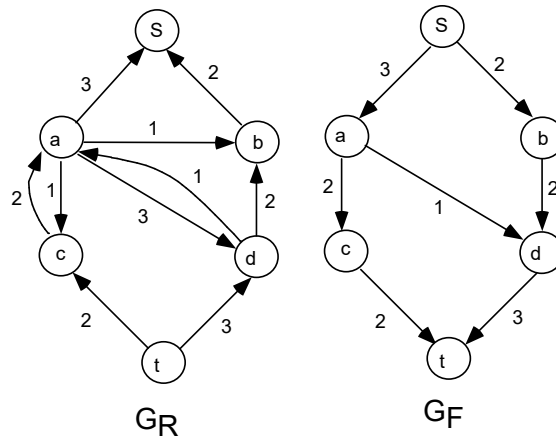
For example we start with



Next we choose the path s, a, d, t , with 3 units of flow.

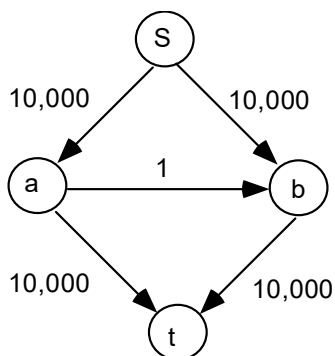


Notice the reverse path. This path simply allows the algorithm to undo some of its previously flow. Next we choose the only path remaining s, b, d, a, c, t with 2 units of flow. Notice that this path requires going from d to a . We are actually changing the flow from a to d from 3 units back to only 1 unit.

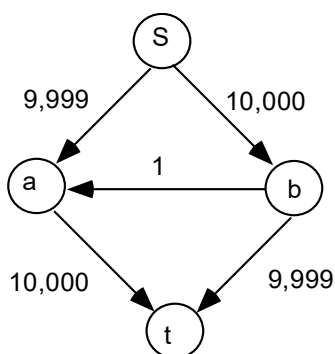


At this point there are no more paths and the algorithm stops. Notice we got the same result as with the other algorithm when we choose the paths correctly. Yet we choose the path that used to yield only 3 units as the maximum flow.

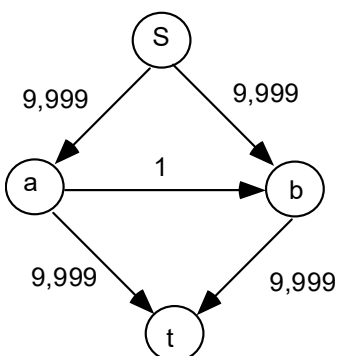
Since we can now choose the paths in any sequence the best choice is the path with the largest flow. Choosing a path sequence that is random for example may yield a very slow algorithm although the resulting maximum flow will be correct. For example consider the following graph.



If we choose the path s, a, b, t, with 1 unit of flow we will get



Notice that we only added a path of 1 unit of flow to the flow graph with this stage. If we then choose the path s, b, a, t we get



In the last two stages we only reduced the residual graph by 2 units. We will need 20,000 stages to complete the problem.

If we always choose the path with the largest flow the problem will be solved in only 2 stages. We can use Dijkstra's algorithm to select the path with the largest flow.

19.6 The Time Complexity.

If we choose the path sequence in random order, then we can use the unweighted shortest path algorithm to find the next path. Recall this algorithm takes $O(|E|)$. If we further assume that the weights are integers then with each stage we subtract at least 1 unit of flow to the residual graph. If the maximum flow is f then the time complexity will be $O(|E|f)$.

However we saw that this method of choosing the path sequence was not good. The path with the largest flow should be chosen. We need to use Dijkstra's algorithm instead. This algorithm takes $O(|E|\log|V|)$ time. Since the network flow algorithm takes

$O(|E|\log cap_{\max})$ with out considering the selection of the path, once we include the time to execute Dijkstra' algorithm we have a time complexity of $O(|E|^2 \log|V|\log cap_{\max})$.

Where cap_{\max} is the weight of the edge with the largest weight. If this weight is not related to the number of edges or vertices then it becomes a constant and the time complexity becomes $O(|E|^2 \log|V|)$.

19.7 Minimum Spanning Tree Prim's Algorithm

A minimum spanning tree is a graph with the minimum number of edges, or minimum sum of weights of all its edges, such that the tree is connected. It can be a directed or undirected graph. Some example applications include wiring a building. We do not care about the length of wire that the current must pass through however we do care about the amount of wire we use to wire the building. Prim's algorithm is best used for directed graphs. Its basically Dijkstra's algorithm. Kruskal's algorithm works for undirected graphs.

Prim's algorithm works by adding edges with the smallest cost to the spanning tree one by one until all vertices are connected. We are interested in the minimum distance from a vertex not connected to the tree to the spanning tree. When considering how to connect a vertex, V , to the spanning tree, we do not care to what vertex V is connected to as long as it is connected to some vertex in the spanning tree. We only consider the weight from V to any vertex in the spanning tree. Prim's algorithm is the same as Dijkstra's algorithm with the exception that we only consider the distance of the vertex to the vertex being processes as opposed to the distance from the vertex to the initially selected vertex.

Algorithm:

```
Void Prims ( int S,
            Graph G)
{
    Table T;
    int P,V,W;
    init_table ( S, G, T);
```



```

For ( ; ; )
{
    V = smallest unmarked vertex

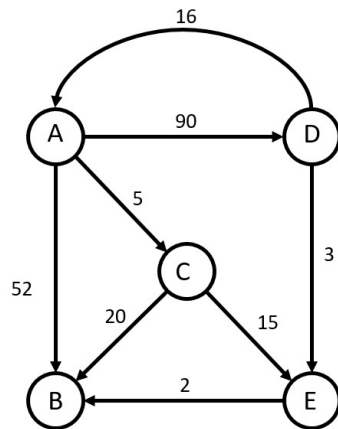
    If (V == NOT_A_VERTEX)
        Break;

    T[V].Known = TRUE;

    P = G[V];
    While (P != NULL) // for each w adjacent to v
    {
        W = P->AdjVertex;
        If ( P->Weight < T[W].dist )
        {
            T[W].dist = P->Weight;
            T[W].path = V;
        }
        P = P->next;
    }
}

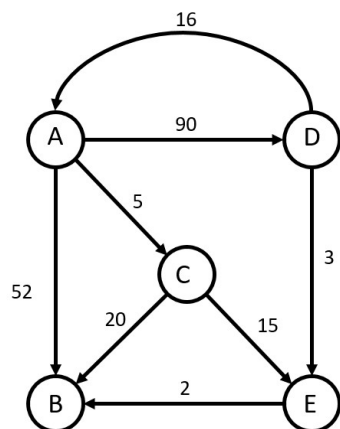
```

Example: Start from A.



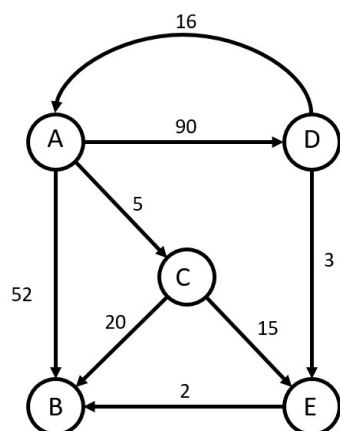
Vertex	Distance	Path
A	0	
B	Inf	
C	Inf	
D	Inf	
E	Inf	

Process A. Mark A as processed and process each adjacent vertex.



Vertex	Distance	Path
A *	0	
B	52	A
C	5	A
D	90	A
E	inf	

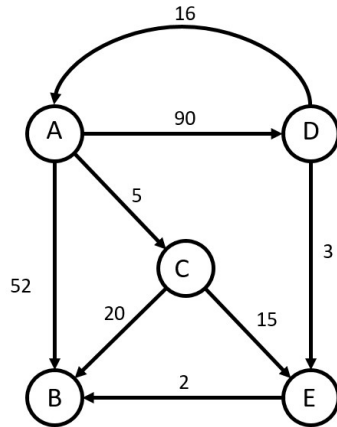
Next process C. Mark C as processed and process each adjacent vertex.



Vertex	Distance	Path
A *	0	
B	52,20	A,C
C *	5	A
D	90	A
E	15	C

Notice that B changed where it comes from. It used to come from A at a cost of 52 and now it comes from C at a cost of 20. Also note the cost is not $5 + 20$ for 25 but just 20 since we do not care where the vertex connects to the spanning tree so long as it is connected.

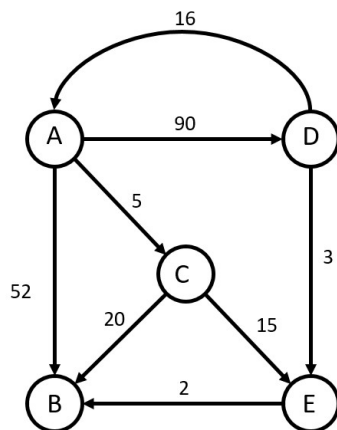
Next process E. Mark E as processed and process each adjacent vertex.



Vertex	Distance	Path
A *	0	
B	52,20,2	A,C,E
C *	5	A
D	90	A
E *	15	C

Notice that B changed where it comes from. It used to come from C at a cost of 20 and now it comes from E at a cost of 2.

Next process B. Mark B as processed and process each adjacent vertex. Nothing changed. Then process D. Mark D as processed.

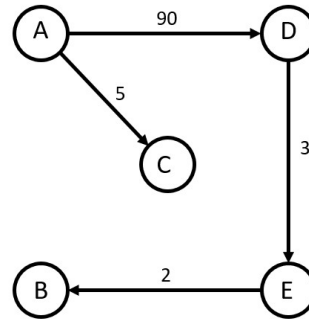


Vertex	Distance	Path
A *	0	
B *	52,20,2	A,C,E
C *	5	A
D *	90	A
E *	15,3	C,D

Notice that E changed where it comes from. It used to come from C at a cost of 15 and now it comes from D at a cost of 3. Basically, once you have D connected, you can get to E with an additional cost of 3 as opposed to 15.

Once done, the spanning tree can be extracted from the table by adding all the vertices that take each vertex from the tree to itself. For example, the vertex going from D to E at a cost of 3 is added to the spanning tree since the table says we need that edge to get to E from D. So the spanning tree then includes the following edges:

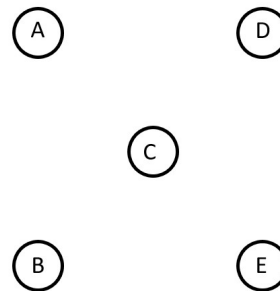
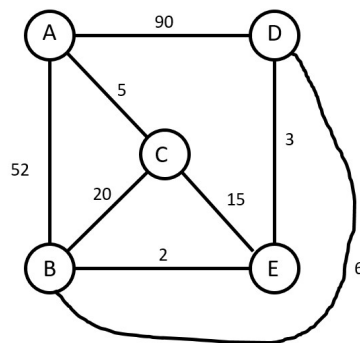
$E \rightarrow B$ at a cost of 2,
 $A \rightarrow C$ at a cost of 5
 $A \rightarrow D$ at a cost of 90
 $D \rightarrow E$ at a cost of 3



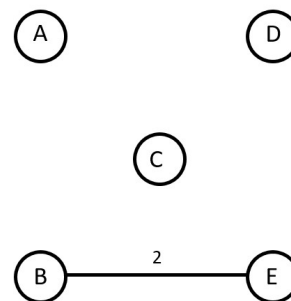
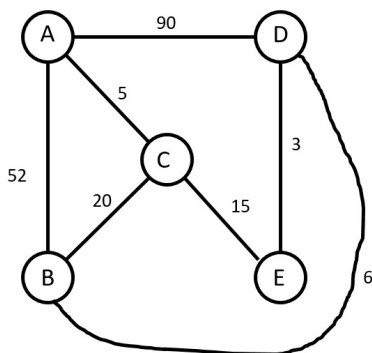
19.8 Minimum Spanning Tree Kruskal's Algorithm

This algorithm works best for undirected graphs. It works by adding the lowest weighted edges one at a time, omitting the edges that create a cycle, until the tree is connected.

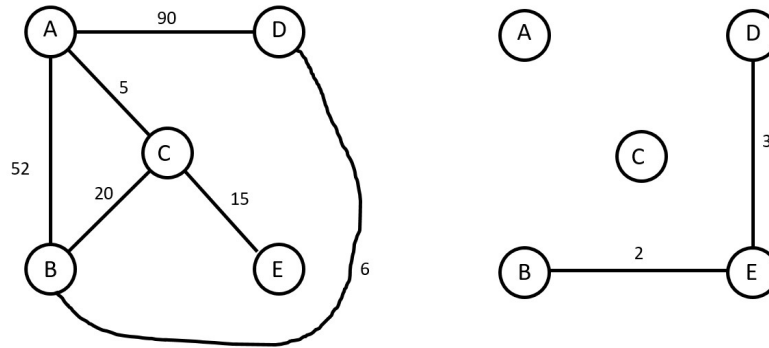
Example: This is the same graph as in the previous example except that its not directed.



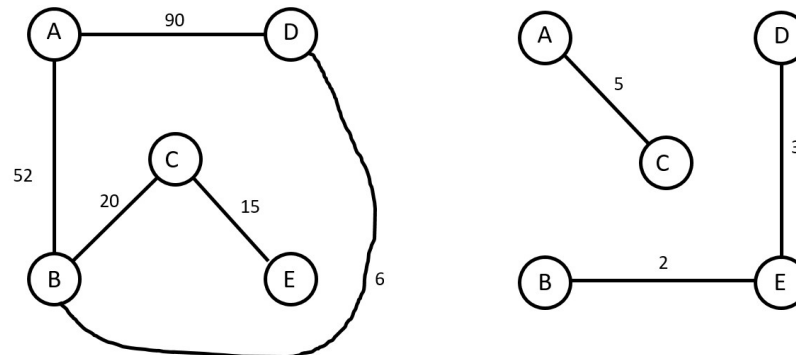
We start with the 2:



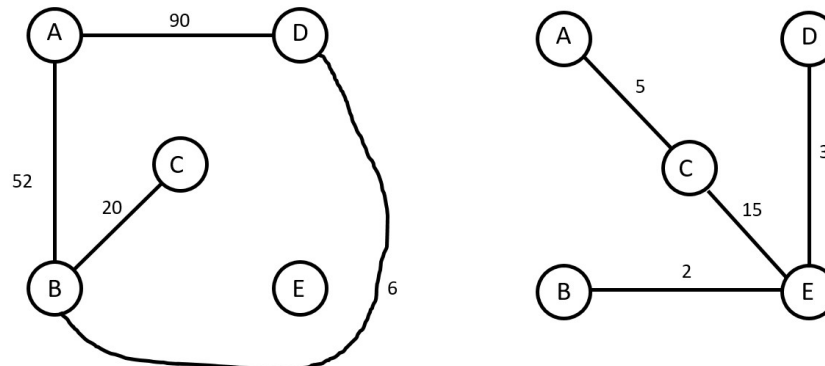
Next we move the 3:



Next move the 5:



Next the 6 is the smallest but the 6 goes between B and D and will form a cycle if we add so we skip and move to the next smallest, the 15.



At this time if we add the next smallest, the 20, it will form a cycle. In fact, adding any of the remaining edges will form a cycle since the tree is now fully connected.

19.9 Euler Circuits

An Euler path is a path that visits every edge of a graph exactly once. If the path starts and ends at the same vertex, then its an Euler Circuit.

Can we tell if a graph has an Euler path? Consider the following:

Suppose the graph has an Euler path. Then for every vertex, v , other than the starting and ending vertex, the path enters v the same number of times as it exists v . If the path enters and exists v , s times, then there are $2s$ number of edges connected to v . Therefore, all

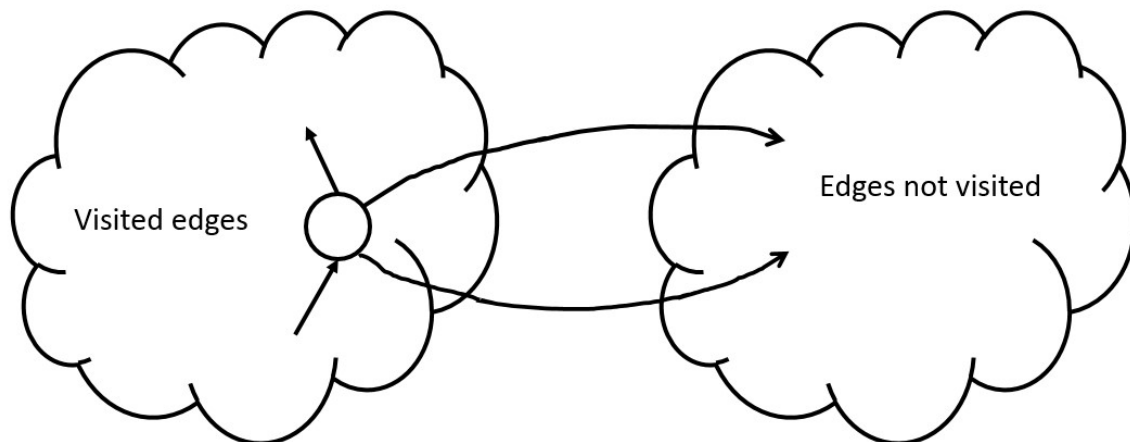
vertices other than the end points must have an even number of edges connected to them. These are called even vertices. If the path starts at one vertex and ends at another, then the starting and ending vertex will have an odd number of edges connected to them. These are called odd vertices. So we have two conclusions:

1. If a graph G has an Euler path, then it must have exactly two odd vertices.
2. If the number of odd vertices in G is anything other than 2, then G cannot have an Euler path.
3. If a graph G has an Euler circuit, then all of its vertices must be even vertices.
4. If the number of odd vertices in G is anything other than 0, then G cannot have an Euler circuit.

We know these are necessary conditions but are they sufficient conditions? That is, if the graph has exactly 2 odd vertices, does that mean it must have an Euler path? And if the graph has no odd vertices does that mean it must have an Euler circuit? Euler says yes, provided the graph is connected.

This is not a formal proof but a rationale for why an Euler path or circuit must exist if the graph has exactly 0 to 2 odd vertices.

Consider you are going for a walk along the edges of the graph and you will never walk along an edge you already walked on. And consider that you cannot stop unless you get to a vertex that has no un-walked path leading out. And let's consider the Euler circuit case, where there are no odd vertices. Then, since all vertices have an even number of edges connected to it, if you enter a vertex then there is at least one un-walked edge leading out and you must take it. You should be able to see that you will only stop when you get to the starting vertex. Then by contradiction, if no Euler circuit exists, then there must be edges that you did not walk on that you could not reach. If this is the case, then the edge that did not get walked on forms a sub graph with other edges that did not get walked on either. Since the vertex connecting the edge that did not get walked on is even, it must have a second edge that did not get walked on either. That edge is also in the subgraph of un-walked edges.

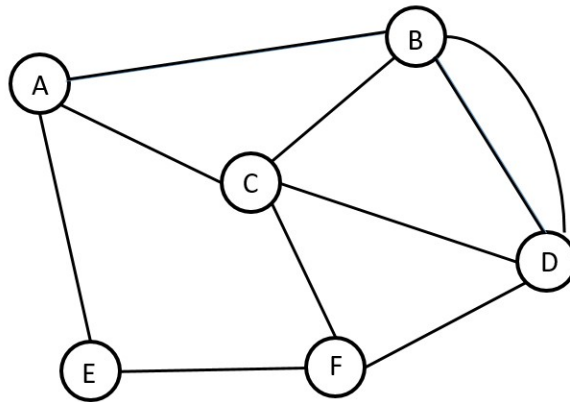


Then repeat the same path and when you get to that vertex take the un-walked on path to the unreachable subgraph. For the same reason you cannot stop before returning to the initial vertex, you eventually will return to the vertex you left from. Then we see that that part of the graph is in fact reachable and this is a contradiction as we said that part was not reachable since no Euler circuit exists. Therefore, we see that a Euler circuit must exist.

Next Fleury's algorithm will find the Euler path or circuit if it exists.

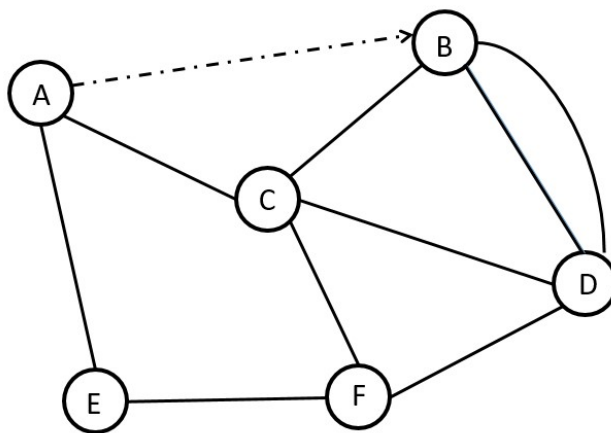
Start at the starting vertex (must be odd for an Euler path).
While you can take an edge without dicconnecting the graph
Take the edge and remove the it from the graph.

Example: Consider the following graph.

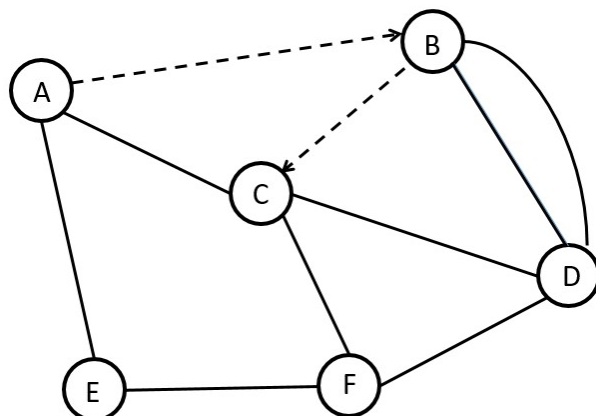


Note vertex A and F are odd and the rest are even so it has an Euler path. We need to start at either A or F. Lets start at A.

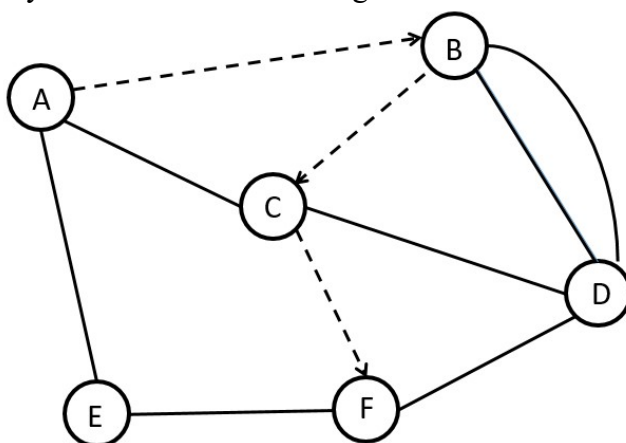
Step 1 Starting at A take any path, say to B and remove that edge. The dashed line is the path so far.



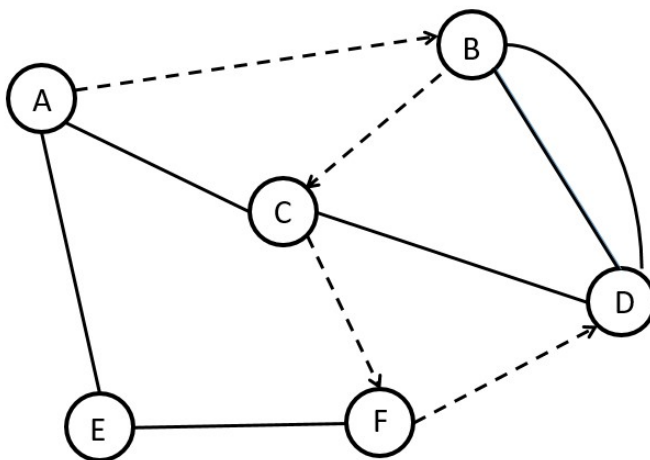
Next take any path, say to C and remove that edge.



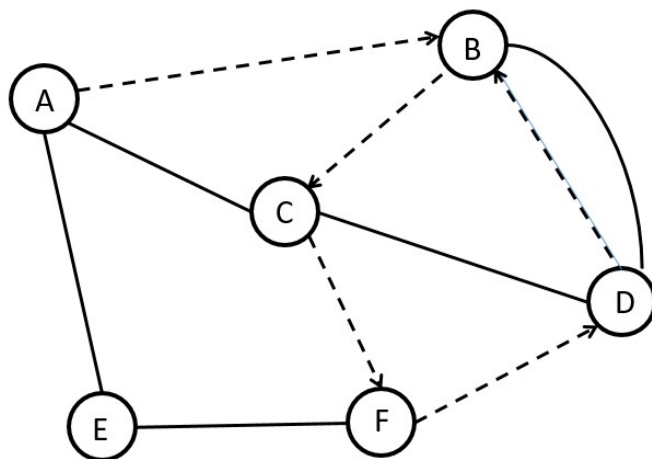
Next take any path, say to F and remove that edge.



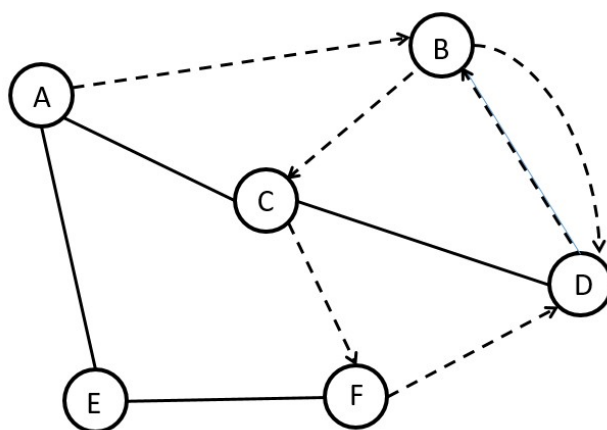
Next take any path, say to D and remove that edge.



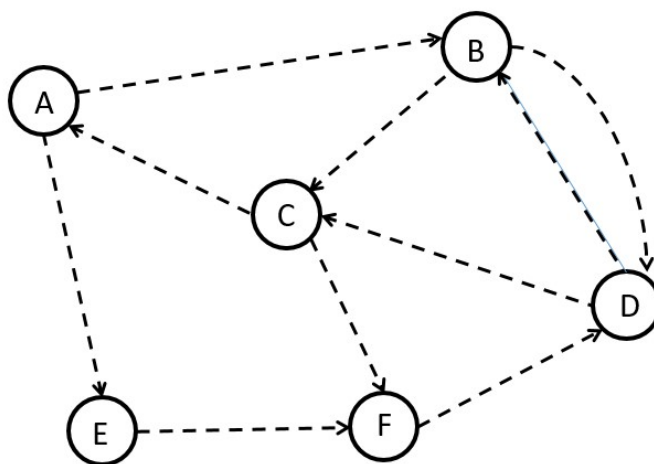
Now we must be careful. If we take D to C we disconnect the graph and get stuck when we work our way to F. We will not be able to jump to D or B to finish their path. The edge D-C is called a bridge in this graph as it ties together the two subgraphs. So let's take D-B instead.



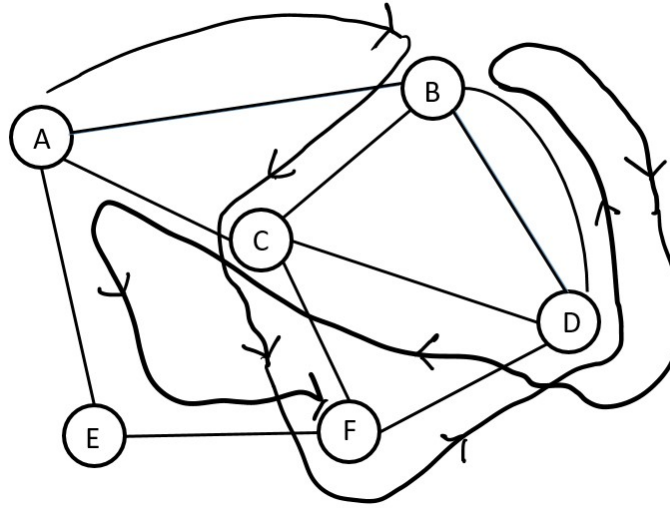
Then go to D.



Next we go to C, then A then E the finally we reach the end point F.



The path is therefore,



The algorithm ran in $O(|E|)$.

19.10 Introduction to NP-Complete

The study of algorithms, Theory of Algorithms, is a fundamental area of computer science. This section is just an introduction of this topic. In this topic we will show how the level of difficulty of a problem can be categorized and placed into sets with problems of similar difficulty. It will be shown that variations of a problem can be quite more difficult to solve than the original problem.

The Euler circuit problem is to find a simple path in a graph that visits every edge. The path must visit every edge exactly once since the path needs to be simple. This algorithm runs in linear time $O(|E|)$.

The Hamiltonian circuit problem is to find a simple path that visits every vertex. The path must visit every vertex exactly once. This problem seems to be a small variation of the Euler circuit problem and yet there is no algorithm that runs in linear time. In fact there are no known algorithms that are guaranteed to run in polynomial time.

The single source unweighted shortest path problem is solvable in linear time but the corresponding longest simple path problem is not. This problem is like the Hamiltonian problem in the sense that there are no known algorithms that is guaranteed to run in polynomial time.

There are harder problems like the Towers of Hanoi problem. This problem is known to take a fast computer millions of years with just a few disks. Adding a single disk increases the duration exponentially.

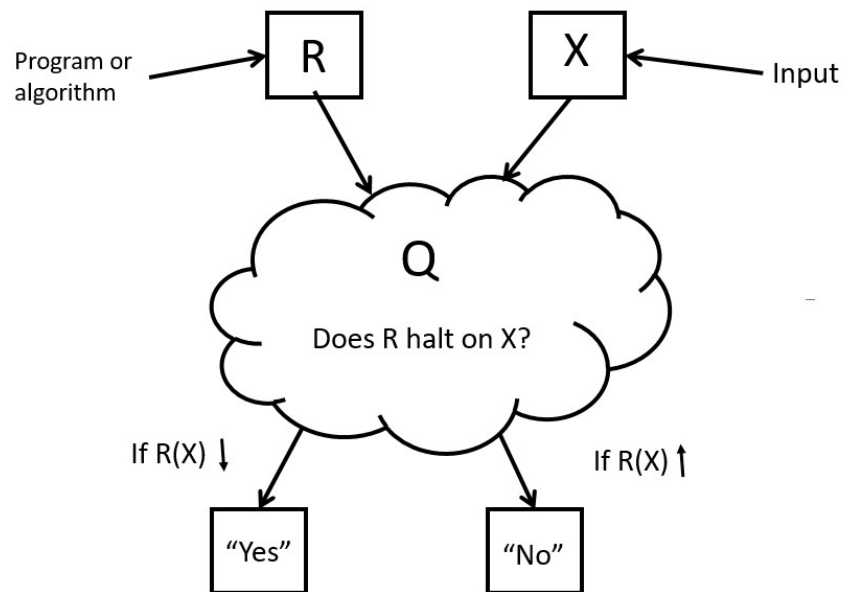
Then there are some problems that are truly hard. These problems are so hard that they are impossible to solve. These problems are called undecidable problems. One particular example is the Halting problem.

19.11 The Halting Problem

Consider adding an extra feature to your compiler. This feature is capable of checking to see if the code it is compiling will go into an infinite loop when given a certain input. We will show that such a feature is impossible. Consider what will happen when we give the feature its own code to check.

Let $Q(R, X)$ be the Halting problem. Q runs program R with X as R 's input, $R(X)$ and returns "YES" if $R(X)$ halts and "NO" if $R(X)$ goes into an infinite loop. That is:

$$Q(R, X) = \begin{cases} R(X) \downarrow \xrightarrow{\text{then}} \text{"YES"} \\ R(X) \uparrow \xrightarrow{\text{then}} \text{"NO"} \end{cases}$$



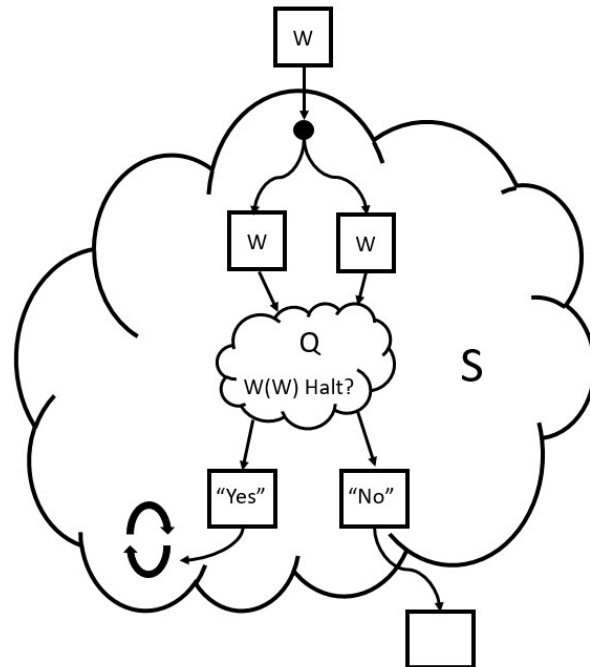
Now let $S(W)$ be constructed such that it first makes a copy of the input W and uses W as both the program and the input to the program. So the program Q inside of S returns YES if $W(W)$ halts and returns NO if $W(W)$ goes into an infinite loop.

Given Q , S can be realized very easily. Consider the code below.

```

1.  int    S (char W[ ])
2.      {
3.      ans = Q(W,W);
4.      if (ans == YES)
5.          while (true);
6.      else
7.          return NO;
8.      }

```

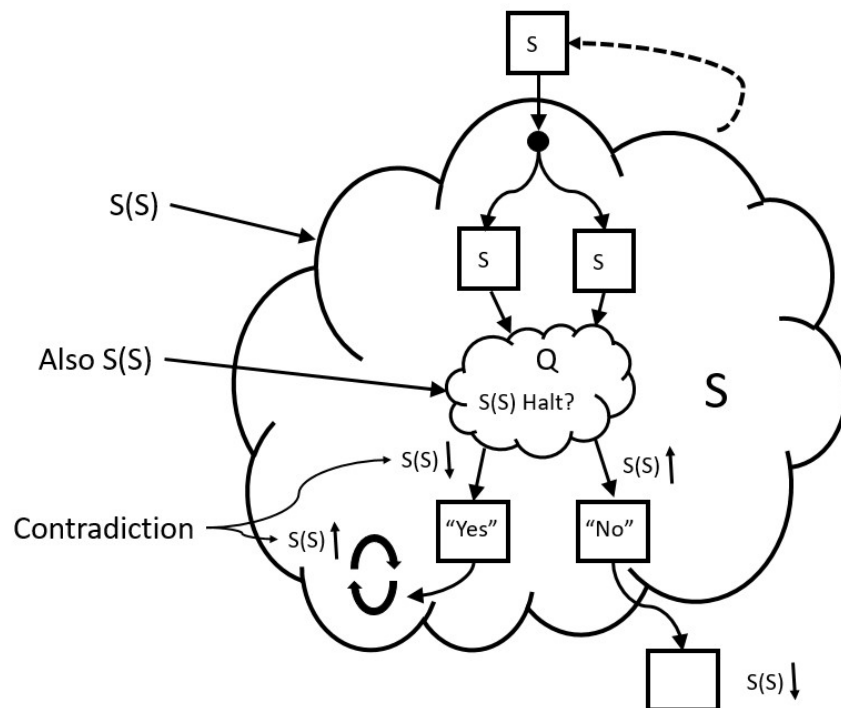


Now lets run program S but use program S source code for its input as in:

```

void main ( )
{
    return S(S);
}

```



What do you think the output of this program should be?

If $S(S)$ halts then $Q(S,S)$ will return YES and therefore $S(S)$ will go into an infinite loop by line 5 in S above. But this means that $S(S)$ does not halt? This is a contradiction. Also if $S(S)$ goes into an infinite loop then $Q(S,S)$ will return NO so by line 7, $S(S)$ returns NO. But we just assumed that $S(S)$ goes into an infinite loop. So $S(S)$ must halt and go into an infinite loop at the same time. This is a contradiction so the program S cannot exist. As we can see above we created program S quite easily given program Q . Therefore its program Q that cannot exist.

Problem Classification:

Deterministic Algorithm: A deterministic algorithm is one that performs that same regardless of its input. It does not look at the input to guide the algorithm. The bubble sort algorithm is an example of a deterministic algorithm.

Non-deterministic Algorithm: On the other hand, a non-deterministic algorithm makes choices based on its input and changes the flow of the algorithm accordingly. The selection sort is a good example of a non-deterministic algorithm.

Magic Computer: If we assume that we have an ultimately intelligent computer that whenever it is given a decision it makes an optimal choice, then having this computer solve a problem using a non-deterministic algorithm is simply a matter of verify its output. For example, if I use this computer to solve the shortest path problem then at each vertex the computer simply tells me where to go next. It will drive the algorithm straight to the shortest path. In fact, this is equivalent to verifying that it is a path given the shortest path.

The set P: The set of all deterministic problems that can be solved in polynomial time is called the set P .

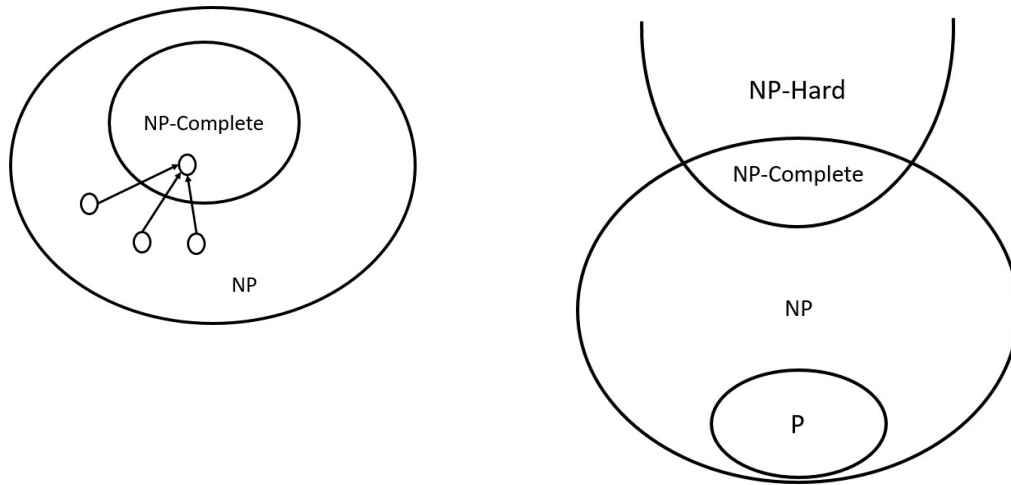
The set NP: The set of all non-deterministic problems that can be solved in polynomial time is called the set NP . By assuming that we have the computer mention above we rule out the possibility that a problem is not considered to be in NP just because we can not find it.

Polynomial time: A problem is solvable in polynomial time if its running time is $O(n^p)$ for some integer p .

Note P is a subset of NP .

The set NP-Hard: If there is a problem such that every problem in the set NP can be reduced to it in polynomial time, then this problem is in the set NP -Hard. The halting problem is in this set.

The set NP-Complete: The set NP-Complete is a subset of the NP set that are also NP-Hard. So a problem is in NP-complete if any problem in the set NP can be reduced to it in polynomial time.



Reduces To: By “reduces to” we mean that a problem A can be reduced to a problem B if we can find an algorithm that runs in polynomial time that convert the solution to B into the solution of A. So if we solve B then we can solve A with a conversion that runs in polynomial time.

A $\xrightarrow{\text{Reduces to}}$ B If we solve A \Rightarrow We can solve B

Therefore, if we were to some day find a solution to a problem in the set NP-complete we would have solved all of the problems in the set NP.

Prove a problem is in NP-Complete: To prove a problem is in the set NP-Complete, we simply find a known problem in the set NP-Complete and find an algorithm that can convert the solution of the problem of the problem in question to that known problem in the set NP-Complete. This conversion must run in polynomial time.

(an example will be great here)

This classification is useful to know so we do not spend too much time searching for a solution to a problem that is too hard.