Kossine

DATA STRUCTURES AND ALGORITHMS

By : Jude Miranda

Neebal Learning Pvt. Ltd.

contact@kossine.com

+1 (408) 663-7998 (USA/Europe) | +91 98213 09516 (Asia)

www.kossine.com

INDEX

1. Introduction

1.1 Data type

A method of interpreting bit pattern is called ***data type***. Every programming language supports certain data types such as *integer, real, char* etc. Each data type supports the primitive operations such as addition, subtraction etc., on that model.

`Integer`	:	The numeric data type consisting of whole numbers, either negative or positive, i.e., -122, +122
`real`	:	Consists of decimal numbers.
`char`	:	Data enclosed in single or double quotes.

For example, 11111110
The above representation will be interpreted as 126 if the variable is signed char, whereas, if the variable is unsigned char, then it will be interpreted as 254.

The difference between the *mathematical data type* and *computer data type* is that mathematical data type can store value without any limit but computer data type cannot store values more than its limit. For example, a variable of type *int* can store only from –32768 to +32767, whereas, the mathematical integer can store without any range restriction.

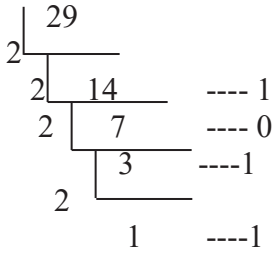
1.2 Number System

Numbers are represented in different formats:

1. Binary
2. Decimal
3. Octal
4. Hexadecimal

1.2.1 Binary Number System

All data in computers are transformed or coded into strings of two symbols. These two symbols are 0 and 1. They are known as *binary digits.* In this system, each bit represents the power of 2 beginning with 2^0 from rightmost where the power increases towards left. Under this representation, any string of bits of length n represents unique non-negative integers between 0 and 2^n-1.
The binary representation of $(29)_{10}$ is $(11101)_2$



Negative Integers :

There are two widely used methods to represent negative numbers.

1. ***One's complement:*** In this pattern, the negative number is represented by changing each bit in its absolute value to the opposite bit setting. For example, 11101 represents 29 and 00010 represents –29. The leftmost bit is no longer used to represent the power of 2 but for the sign. If the leftmost bit begins with 0 then it represents positive and 1 represents negative number. In this representation, we get a positive as well as a negative zero. When all the bits are 1 then it represents a negative zero and when all the bits are 0 then it represents 1.

2. ***Two's complement:*** In this representation, 1 is added with one's complement. 00010 represents –29 in one's complement and 00011 represents –29 in two's complement. The advantage of two's complements is that we avoid the negative representation of 0 and make use of the representation for one more value. Hence, any string of bit length n can represent negative integers from -2^{n-1} to $2^{n-1}-1$.

Real Numbers:

In general, the real numbers are represented in a scientific form. It has two components; mantissa and exponent. The mantissa is represented by the first three bytes and the exponent is represented by the last byte. The base is usually fixed as 10.

The binary representation of 3.53 could be represented as 353×10^{-2}.

000000000000000101100001**11111110**

The highlighted bit string represents the exponent.

1.2.2 Decimal Number System

The decimal number system uses ten symbols 0,1,2,3,4,5,6,7,8,9, hence its base is ten.
In general, we use the decimal representation of numbers.

```
int num=29;

printf("%d",num); → 29
```

1.2.3 Octal Number system

The octal number system uses 8 symbols, namely, 0,1,2,3,4,5,6,7 thus having 8 as its base.

The octal equivalent of $(90)_{10}$ is $(132)_8$

```
8 | 90
8 | 11    ---- 2
     1    ---- 3
```

```
int num=90;
printf("%o", num); → 132
```

1.2.4 Hexadecimal Number system

The hexadecimal representation uses the following16 symbols:

0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

thus its base is 16.

The hexadecimal representation of $(284)_{10}$ is $(11C)_{16}$

```
   | 284
16 |________
    16 | 17        ----C (equivalent of 12)
       |_____
          1        ---- 1
```

```
int num=284
printf("%x", num);   →11c
printf("%X",num);    →11C
```

1.3 Data Structure

A ***data structure*** is a data object that may be a composition of primary and secondary data types, with a relationship among the other objects and among the individual element that composes the object.
The study of data structures, therefore, involves two complementary goals.

- The first goal is to identify and develop useful mathematical entities and operations and to determine what classes of problems can be solved by using these entities and operations. This stage is known as defining an Abstract Data Type.

- The second goal is to determine representations for those abstract entities and to implement the abstract operations on these concrete representations. This stage is known as implementation.

The formal **Definition of Data structure** is as follows:

A ***data structure*** is a set of domains $\mathcal{D}$, a designated domain $d \varepsilon \mathcal{D}$, a set of functions $\mathcal{F}$ and a set of axioms $\mathcal{A}$. The triple $(\mathcal{D},\mathcal{F},\mathcal{A})$ denotes the data structure d which will usually be abbreviated by writing d.

Here, designated domain means specified area, functions is to operate on that data and axioms mean the semantic of the operations.

d=Integer
D={Integer,Boolean}
F={ADD,SUB, DIV..}
A= ADD(x,y) a function to add x and y.

Depending on the organization of the elements of the object, data structure is classified into two:

1. ***Linear :*** Elements are stored in a sequential order. eg. Arrays, linked list, stacks and queues.
2. **Non-linear data structures:** Elements of this data structure are stored in a non-linear order. eg Trees and graphs.

1.4 Abstract Data type (ADT)

An abstract data type is a mathematical model together with various operations defined on the model. With an ADT the users are not concerned with how the task is done, rather, they need to be concerned only with what it can do.
It consists of two parts:

1. Declaration of data
2. Declaration of operations

Declaration of Data

Declaration of abstract data types is generalization of primitive data types (integer, real, etc.)

It is a collection of values. It consists of two parts:

1. Definition clause
2. Conditional clause

The keyword *abstract typedef* is used to introduce the value definition and the condition is used to specify any condition. The definition clause is required whereas the conditional clause is not necessary for every ADT.

Declaration of Operations

Declaration of operation are procedures that are generalizations of primitive operations (+,-, etc) such as creation, initialization, manipulation, and etc., on that model.
The operator is defined as an abstract function with three parts:

1. Header
2. Precondition (Optional)
3. Post condition

The Operator definition of the ADT of STACK includes the operations of push, pop, isempty, and etc.

```
Abstract Pop(s,elt)
STACK(eltype) s;
Precondition: empty(s)==FALSE
Postcondition s=sub(s,1,len-1);
```

Here the first two lines indicate the header. The precondition specifies the necessary condition to proceed further. The post condition specifies what operation does. In a post condition, the name of the function is used to denote the result of the operation.

Sequence as a Data

A sequence is simply an ordered set of elements. A sequence, S, is sometimes written as the enumeration of its elements, such as;

S=<s0,s1,….sn-1>

If S contains n elements, S is said to be of length n. We assume the existence of the length function len such that len(S) is the length of the sequence. We also assume first(s) that returns first value of the list and last(s) that returns the last value of the sequence. Two sequences are

equal if each element of the first is equal to the second. A sub sequence is a contagious portion of a sequence. If S is the sequence, the function sub(S,I,j) refers to the subsequence of S starting at position I in S, consisting of j consecutive elements.

The primitive operations may be defined for that ADT are given in the following

1) ***Create()***
   Creating a new variable of the newly defined data type.
2) ***Assign ()***
   Assigning a new value to the instance of the data type
3) ***Operations()***
   This operation creates a new set of values out of the given two sets. The operation may be union or intersection.
4) ***Find()***
   Searches an entity 'X' in the variable. If yes returns 1 else returns 0
5) ***Cardinality()***
   Returns the number of members in the set.

1.5 Implementation of a data structure

The *implementation of a data structure* is a process of defining an abstract data type in terms of directly executable function using a programming language. At the first stage, the data structure needs to be defined and then it needs to be coded. Each data structure is built up from the basic data types of the underlying programming language using the available data structuring facility such as arrays, structures, files etc.

Implementation of a data structure ***d*** is a mapping from ***d*** to a set of other data structure ***e***. This mapping requires two concepts:

1. Every object of ***d*** is to be represented by the objects of ***e***.
2. Every function of ***d*** must be written using the function of the implementing data structure ***e***.

It is important to recognize the limitation of a particular implementation. For example, one cannot store an integer value beyond the range of the integer in the system.

Also, the efficiency of such implementation needs to be considered. The efficiency, in general, is determined by time and space. If a particular application is heavily dependent on manipulating highlevel data structure, the speed of the program and the memory required to run the application needs to be considered.

There are two kinds of implementations:

1. **Hardware Implementation**
   Interpreting the string of bits in the desired fashion and to perform the required operations. The hardware implementation is already defined.

2. **Software Implementation**
   The software implementation includes a specification of how an object of the new data type is represented by an object of previously existing data types. For

example, string is a data structure, which is been represented with another data type character.

Suppose the computer contains an instruction;

MOVE(source, dest, length)

This instruction copies a character string of *length* bytes from an address specified by *source* to an address specified by *dest.* In this case, the MOVE function copies a fixed length, array copies to another array. If we want to copy a varying length array, we need to know the length of the array, which can be accomplished by the code.

```
MOVE(source,dest,1)
      {
            for(I=0;I<dest;I++)
            {
                  MOVE(SOURCE[I], DEST[I],1)
            }
                  }
```

1.6 Linear list

A ***linear list*** is a data structure whose instances are of the form (e1,e2,e3…….en), where *n* is the finite natural number. e^i is the elements of the list and *n* is the length of the list. If `n=0`, then the list is empty, when n>0, the e1 is the first element and en is the last element; where s is the size of each element.

The following number of operations may also be performed on a linear list:

1. *Store* data on the list
2. *Retrieve* data from the list
3. Computing the *length* of a list
4. *Search*ing for a given element in a list
5. *Insert*ing a new element at a specified position “i”, `0<=i<=n-1` where “n” is the length of the list
6. *Deletion* of an element at a given position, “ I” in a list of “n” elements
7. *Reversing* the list
8. *Sorting* the list.

1.7 Data representation Methods

In Linear list, data can be represented in the following categories:

1. Formula Based
2. Linked Representation (Pointer Based)
3. Indirect addressing
4. Simulator Pointer

1.7.1 Formula Based: Arrays

Formula based representation uses a mathematical formula to determine where (i.e., the memory address) to store each element of a list. It uses an array to represent the instance of the object. Each position of the array is called as cell or node . Each node is large enough to

hold one of the elements that make up an instance of the data object. In some cases, one array may represent one instance while in other cases it represents several instances.

For example, in an array the location of the i^{th} element is at the `i-1` position

location(i) = i-1

```
int arr[10]={ 3,4,8,2,1,9,5,6,7,11};
printf("%d", arr[3]); →2;
```

Where 2 is stored at the 4^{th} position. When the 4^{th} positional value is to be manipulated we get it through the subscript by reducing one from its position

location(4^{th} element) = 4-1;

1.7.2 Linked Representation : Linked List

In a linked representation the elements of a list may be stored in any arbitrary set of memory locations. Each element has an explicit pointer (or link) that tells us the location (i.e., Address) of the next element. There are three categories of linked list

1. Single linked list
2. Double linked list
3. Circular linked list

1.7.3 Indirect Addressing: Array representation of linked list

Indirect addressing is a combination of formula based method and linked representation. In indirect addressing, the list elements may be stored in any arbitrary set of locations. A table is maintained to store the address of i^{th} element. The elements may themselves be stored in dynamically allocated nodes or in an array of nodes.

In this method, we retain the advantage of using formula based method, wherein, elements can be accessed by an index along with the advantage of linked list, which is not moving the elements physically, during the operation of sorting and searching.

1.8 STRINGS

A string is an array of character stored in an adjacent memory location terminated by the NULL character. The null character indicates the end of the string. A string constant is a set of characters included in double quote marks. The NULL character is automatically appended to the end of the character and is denoted by '\0'.

Implementation of Strings:

String as an ADT

```
Abstract typedef  <char> STRING;

Abstract int length(s)
STRING s;
```

Postcondition: return the length of s;

```
Abstract  STRING concat(s1,s2)
STRING s1,s2;
```

Postcondition: s3=s1+s2;
Return s3;

```
Abstract  strcpy(s1,s2)
STRING s1,s2;
```
Precondition: if the s1 is not empty
Postcondition: Copy s1 into s1.

```
Abstract strrev(s)
STRING s;
Postcondtion: len=strlen(s)
              Swap the content of str[I] and str[len]
              until I and len becomes equql.
              I=I+1;
              len=len-1;
```

String Functions in C

```
Int length(char str[])
{
     int len=0;
     for(len=0;str[ len ]!='\0';len++); /*increase the value of len by one till the
'\0'character*/
      return len;
}

int strcpy(char str1[],char str2[])
{
     int I=0,len;
     len=strlen(str1);          /* Get the length of str1*/
     if(len==0)
     printf("Empty string");
     else
     for(I=0;I<len;I++)          /*Copy character by character from str1 to str2.  till
'len' */
      {
           str2[I]=str1[I]
      }
}

void strrev(char str[])
{
   int I;len=strlen(str);                    /*Find the length of 'str'*/
   for(I=0,len=len-1 ; I>=len;I++,len--)/* Until len and I become greater than or
equal swap the  content.*/
   {     char temp;
         temp=str[I];
         str[I]=str[len];
         str[len]=temp;
   }
}
```

1.9 Algorithm

A finite sequence of unambiguous instructions to solve a problem is called an *algorithm*.

Algorithm has the following five characteristics:

1. **Input:** An algorithm begins with instructions to accept *inputs*. These inputs are processed by subsequent instructions in the algorithm.

2. **Definiteness:** The processing rules specified in the algorithm must be precise and nambiguous. In other words, the instruction must not be vague.
3. **Effectiveness:** Each instruction must be sufficiently basic (division, multiplication, addition etc.) so that a person carries it out in finite time, with paper and pencil.
4. **Finiteness:** The total time to carry out all the steps in the algorithm must be finite. Even if it contains repetitive statements, the repetitions must be finite.
5. **Output:** An algorithm must produce one or more outputs.

Note: The difference between algorithm and a program is that a program does not necessarily satisfy condition 4. For example, operating system is a program that never stops.

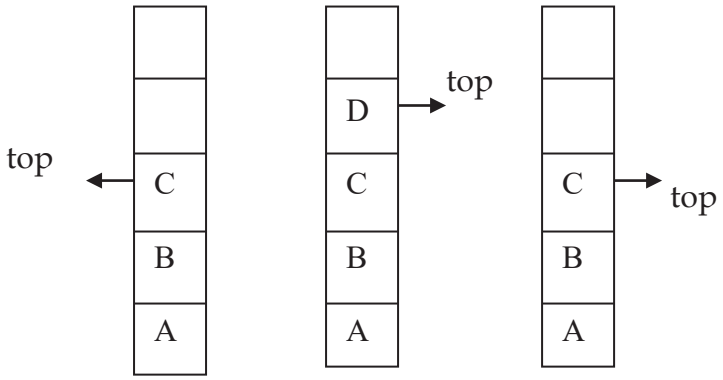
Questions

1. Define structure.
2. Explain different kinds of data structures?
3. Explain in brief different types of data representation.
4. What is an algorithm? List the characteristics of good algorithms.
5. Define Abstract data type?
6. What do you mean by implementation of data structure?
7. Name the two components of data structure?
8. What is the difference between data structure and data type?
9. What is a string? Implement string as a Data structure.

2. Stack

Definition: A ***stack*** is an ordered list in which insertion and deletion are done at one end, where the end is called as ***top***. The last element to be inserted is the first one to be deleted. Hence, it is called as Last In First Out (LIFO) list.

Special names are given to the two changes that can be made to a stack. When an element is inserted in a stack, the concept is called as *push*, and when an element is removed from the stack, the concept is called as *pop*. Trying to pop out an empty stack is called as *underflow* and trying to push an element in a full stack is called as *overflow*.



2.1 Stack as an Abstract Data Type (ADT)

The following operations make a stack an ADT

```
Abstract  typedef  STACK(etype)
```

```
Abstract  bool  isempty(s)
```
if the stack is empty, returns TRUE(1) else return FALSE `(0)`

```
Abstract bool isfull(s)
```
If the stack is full, it returns 1 else0;

```
Abstract  push(s, V)          //Push the value on the top
Type(V)=etype.                // v  is the type of data type of stack
```
It places the value **v** at the top of the stack

```
Abstract etype pop(s)  //Returns the element at the top of the stack
```
If the stack is not empty returns the top element of the stack.

Abstract etype poptop(s)

If the stack is not empty, returns the top element of the stack.

2.2 Algorithm for STACK

EMPTY(S) : To check whether the stack is empty
S: stack variable contains an array 'arr' and pointer to the 'top'

If(top==-1)
Return 1
Else
Return 0

PUSH(S, V) : To store a value in the stack
S : stack variable contains an array 'arr' of size 'MAX'and the top pointer 'top'
V : Value to be inserted

If(top>=MAX-1)
Print "overflow"
Else
Top=Top+1; //increment the top element of the stack
Arr[top]=V //Storing 'V' in the array.

POP(S) : //To remove an element from the top

S: stack variable contains an array 'arr' and pointer to the 'top'
If(top=-1)
Print "empty"
Return 0
Else
V= arr[top]
Top=top-1
Return v

MAIN()

STEP 1: Display Choice 1. Push 2.Pop 3. exit
STEP 2: Get the choice
STEP 3: If the choice is
1. goto step 3
2. goto step 4
3. goto step 5

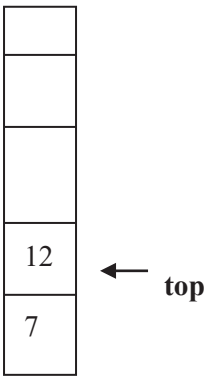
STEP 4: Take a value from the user and push in the stack then goto step 1
STEP 5: Pop the stack and print the value then goto step 1
STEP 6: stop.

2.3 Array implementation of Stack

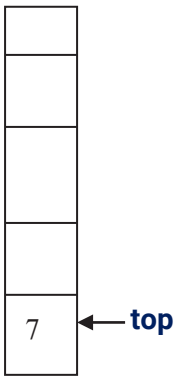
Functions of stack.

1. During the initialization of the stack, 'top' is initialized to –1
2. On each addition, top is pre-incremented by one
3. On each removal, top is post-decremented by one
4. top=SIZE-1 denotes that the stack is full. Insertion on a full stack causes overflow.
5. top=-1 denotes that the stack is empty. Removal from the empty stack causes underflow.

```
#include<stdio.h>
#define SIZE 10                    // Maximum  number of elements in every stack can be 10
struct stack
{
   int top;
   int arr[SIZE];
};

int isempty(struct stack *p)
{
  return (p->top==-1);   // if the condition is true then 1 is returned  else 0 is returned

}
int isfull(struct stack *p)
{
  return (p->top==SIZE-1)   //if the condition is true then 1 is returned else 0 is returned
}

void push(struct stack *p, int elem)
{
 if(p->top==MAX-1)        /* p->top==MAX-1 indicates that the  stack is full*/
  printf("Stack overflows");
 else
  p->arr[++p->top]=elem;     /*Increasing the 'top' by 1 and storing the value at 'top' position*/
}

int pop(struct stack *p)
{
  if(p->top==-1)          /* p-top==-1 indicates empty stack*/
  {
  printf("stack is empty");
  return 0;
  }
  else
  return( p->arr[p->top--]);/* Removing element from 'top' of the array and reducing 'top' by 1*/
}
void main()
{
      struct stack s;          //Creating a stack variable.
      s.top=-1;                //Initializing the top to –1
      push(&s,7);              //To push the value 7 at the 0th position of the array.
      push(&s,12);             //To push the value 12 at the 1st position
      printf("%d\t",pop(&s));   → 12  //Removing the last inserted value in the stack.


}
```



After pushing two elements



After calling the pop function

2.4 Applications of stacks

The concept of stacks can be applied in many applications . The following are the some of the applications

1. **Parsing**
2. **Reversing**
3. **Postponment**
4. **BackTracking**

. Balancing Symbols:

Compilers check the programs for syntax errors. Lack of one symbol such as a missing parenthesis or comment starter will cause the compiler to spill out hundred lines of errors. A useful tool in this situation is using a stack to check whether the opening and closing symbols are corresponding or not.

`(a+b)+(c-d)` → valid
`((a+b)+(c-d)` → Error: one closing brace is missing
`((A+B)+[c-d])` → Valid: Opening and immediate closing braces correspond
`((A+B)+[c-d]}` → Error: The last closing brace does not correspond with the first opening parenthesis.

Make an empty stack. Read characters until the end of file. If the character is an opening symbol then push in the stack. If it is a closing symbol, pop the stack. If the stack is empty or the popped out characters is not corresponding character reports an error. Finally, after reading everything if the stack is not empty reports an error.

2. Converting Infix to postfix

A stack can be used to convert an infix expression to a postfix or prefix expression. When an operand is read, it can be immediately placed on the postfix array. When an operator is read, including an opening parenthesis, we push in the stack by considering the precedence of the operators. When a closing parenthesis is read, we pop out all the operators from the stack till we encounter the opening parenthesis and store it in the postfix array. After reading all the characters, if any operator remains in the stack then we pop and store on the postfix array.

Note: Give an example.

3. Recursion

When a function is called, all the important information that has to be saved, such as register values and the return address, is saved in the top of the pile. Then the control is transferred to the new function, which is free to replace the register with its values. The pile where the information is stored is called as stack or stack frame. Running out of the stack space is always a fatal error.
Recursion implicitly uses stack in which activation records of recursive calls are pushed. These activation records are popped when a call to a function returns to its caller.

5. Stack Machines

A stack machine uses last in first out stacks
A stack machine obtain their argument from a stack and place their result on the stack.

It consists of two stores

1. **Program Store :** is organized as an array and is read only
2. **Data Store :**The data store is organized as a stack.

It also has four registers

1. **Instruction Register(IR) :** It contains the instruction
2. **Stack top pointer**: Contains the address of the top element of the stack.
3. **Program Counter:** Address of the next instruction that is to be interpreted.
4. **Activation Record Pointer:** Contains the base address of the activation record of the procedure which is being interpreted.

Also it has set of instruction sets.

2.4.1.1 Algorithm for balancing symbols

Let 's' be the stack variable and 'exp' be an infix expression to be validated
Read a character from 'exp'

```
While('exp' is not null)
{
  if (character is an opening parenthesis)

    push in the stack
    else (if it is a closing brace)
    pop the stack

Read the next character
}
if(stack is empty)
print valid expression
else
print invalid expression.
```

2.4.1.2 C procedure for balancing symbols

/* Program to check whether the given expression has proper balancing symbols*/
Note : Define structure, push and pop functions

```
main()
{
 struct stack s;
 char str[20];
 int i,c;
 s.top=-1;
 scanf("%s",str);
 printf("%s",str);

 for(i=0;str[i]!='\0';i++)
 {
  if(str[i]=='(')
  {
   push(&s,str[i]);
  }
```

```
  if(str[i]==')')
  {
   c=pop(&s);
  }
 }
 if(s.top==-1)          /*if the empty stack is empty then all the opening braces have equal number
of closing braces*/
  printf("VALID EXPRESSION");
 else
 printf("INVALID EXPRESSION");
return 0;
}
```

2.4.2 Infix, prefix and postfix

Infix: An infix expression is a single letter, or an operator, preceded by one infix string and followed by another Infix string.

A
A+B
(A+B)+ (C-D)

Prefix: A prefix expression is a single letter, or an operator, followed by two prefix strings. Every prefix string longer than a single variable contains an operator, first operand and second operand.

A
+AB
++AB-CD

Postfix: A postfix expression is a single letter or an operator, preceded by two postfix strings. Every postfix string longer than a single variable contains first and second operands followed by an operator.

A
AB+
AB+CD-+

Prefix and postfix notions are methods of writing mathematical expressions without parenthesis. Time to evaluate a postfix and prefix expression is O(N), were N is the number of elements in the array.

Infix	Prefix	Postfix
A+B	+AB	AB+
A+B-C	-+ABC	AB+C-
(A+B)*C-D	-*+ABCD	AB+C*D-

*** Points to be remembered while converting infix expression to postfix or prefix.**

1. First always evaluate the higher precedence operators and make the expression into single operand.
2. If the required output expression is postfix then put the operators after the two operands, if prefix, place the operator before the operands.

3. If there are many brackets, always evaluate the innermost bracket
4. If there are operators with same precedence, evaluate from left to right.

Note: ^ can also be written as $.

Infix : `((A+B)*C-(D-E)) $(F+G)`

Postfix :
```
((AB+)*C-(DE-))$(FG+)
((AB+C*) -(DE-)) $ (FG+)
(AB+C*DE-) $ (FG+)
 AB+C*DE-FG+$
```

Prefix:
```
((+AB) *C-(-DE))$ (+FG)
((*+ABC)-(-DE)) $ (+FG)
(-*+ABC-DE) $ (+FG)
 $-*ABC-DE+FG
```

2.4.2.1 Algorithm to convert infix to postfix

Precedence()

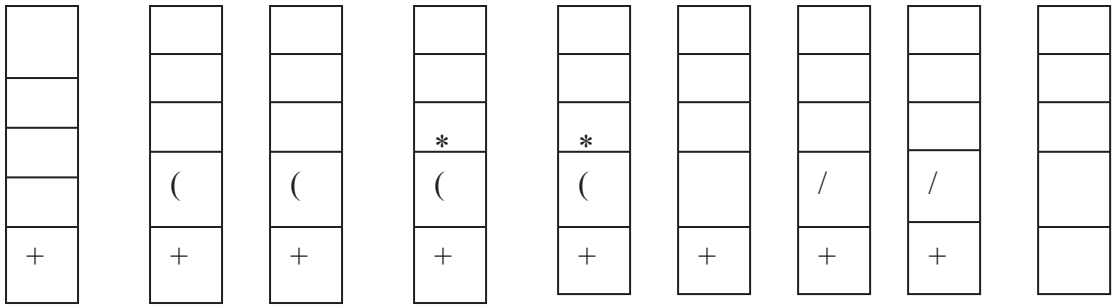
Operator	precedence
(	0
+ -	1
* / %	2
^	3
)	4

Let 's' be a stack, 'exp' be an infix expression

While(character of 'exp'is not '\0')
{ **if**(character is an operand)
store it in the postfix array 'post'
else
{
if(1. the stack is empty or
2. the character has higher precedence than the stacktop operator
3. the character is '('
4.character and stacktop character is '^'
5.AND the character is not)
{
push the character in the stack
}
else if(character is ')')
{
pop all the characters from the stack until opening '('
}
else if(the character is lesser or equl precedence with the stack top operator)
{
1.pop all the characters from the stack until the current character

becomes higher precedence to the stack top operator or the stack
becomes empty.
2. push the current character in the stack.
}
}
}
while(the stack is not empty)
{
pop the stack and store all the characters in the 'post' array
}

Diagrammatic representation of the above algorithm for the expression a+(b*c)/d



Output Array :

a	a	ab	ab	abc	abc*	abc*	abc*d	abc*d/+

2.4.2.2 C procedure for Converting infix to postfix

Note: Define stack, push,pop and stacktop functions

```
int higher(char op1,char op2)
{    int r1=preced(op1);
     int r2=preced(op2);
     if(r1>r2)
     return 1;
     else
     return 0;
}

/*Returns the precedence of the operator */
int preced(char op)
{   int r;
   switch(op)
   {
   case '(':
   r=0;
   break;
   case '+':
   case '-':
    r=1;break;
   case '*':
   case '/':
    r=2;break;
```

```
    case '^':
    r=3;break;
    case ')':
    r=4;
    }
    return r;
}
void main()
{   struct stack s;
    char  str[30],post[30];
    int i,j=0;
    s.top=-1;
    printf("Enter the infix expression");
    scanf("%s",str);
    for(i=0;str[i]!='\0';i++)
    {    if(isalpha(str[i]))       //if the character is an alphabet store it in the postfix array
         {  post[j++]=str[i];
         }
         else if(str[i]!=')')
         {     while(s.top!=-1&&!higher(str[i],poptop(&s))&&str[i]!='('&&
                     poptop(&s)!='^'&&str[I]!='^')
           {
                post[j++]=pop(&s);
           }
           push(&s,str[i]);
         }
         else
         {    char ch;
            while( (ch=pop(&s))!='(')
            {
              post[j++]=ch;
            }
         }
   }
    while(s.top!=-1)
    {    post[j++]=pop(&s);
    }
    post[j]='\0';
    printf("%s",post);
 }
```

2.5 Algorithm to evaluate postfix expression

Let 's' be a stack variable and 'exp' be a postfix expression
Read a character from 'exp'
While(the character is not '\0')
{
  if(the character is an operand)
  push in the stack
  else
  {
   pop two operands from the stack
   make the corresponding operation of the character operator
   push the result in the stack
  }
}

} pop the stack to print the final result.

Note: To evaluate prefix expression, reverse the prefix expression and apply the same algorithm**.**

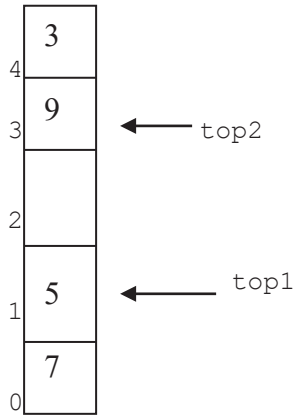
2.6 Maintaining two stacks in one array

Functions:

1. There are two top variables, maintaining top1 and top2.
2. During Initializations, top1 is initialized to –1 and top2 is initialized to MAX
3. Both stacks grow toward each other in the array.
4. Push1 inserts a value with the incremented value of top1.
5. Push2 inserts a value with the decremented value of top2.
6. Pop1 pops up the value with the post decremented value of top1.
7. Pop2 pops up the value with the post incremented value of top2.
8. if top1+1 = top2, then in both the stacks, value cannot be inserted.
9. if top1=-1, then value cannot be removed from stack1.
10. if top2 =Max, then value cannot be removed from stack2.

```
#include<stdio.h>
#define MAX 10
struct stack
{     int arr[MAX];
      int top1,top2;
};
void push1(struct stack *s,int v)
{  if((s->top1==s->top2)
    printf("overflow");
    else
    s->arr[++s->top1]=v;
}
            int pop1(struct stack *s)
            {
             if(s->top1==-1)
             {
              printf("underflow");
              return 0;
             }
             else
             {
              return s->arr[s->top1--];
             }
            }
            void push2(struct stack *s,int v)
            {
             if((s->top2==s->top1))
             {
              printf("overflow");
             }
             else
             {
              s->arr[--s->top2]=v;
             }
            }
            int pop2(struct stack *s)
            {
             if(s->top2==MAX)
             {
```

```
  printf("underflow");
  return 0;
 }
 else
 {
  return s->arr[s->top2++];
 }
}
int main()
{

 struct stack s;
 s.top1=-1;
 s.top2=MAX;
 push1(&s,7);
 push1(&s,5);
 push2(&s,3);
 push2(&s,9);
 printf("%d",pop1(&s)); -->5
 printf("%d",pop2(&s)); -->9
 return 0;
}
```



Use of Multiple Stacks in one array

In certain applications, each data item may be having related elements or objects. To establish these links, one has to go for interlinked data structures. One method of achieving this is by multiple stacks.

Questions

1. What is a stack? Define stack as ADT.
2. List the applications of stacks
3. Write an algorithm for stacks using arrays?
4. Can we define multiple stacks in one array? If yes, what is the need for it.
5. Show diagrammatically how the stack is used to evaluate the following expression? (a+b-c*d)/(e+f)
6. Convert the following infix expression into prefix and postfix
   - I) (A-B)*C-D/E+G-H
   - II) (A^B^C)*D/(F-G)
   - III) (A*(B/C)/D+(E+G)^H)
   - IV) (A+B)+(C*D)*(E+F)^(G/H)
7. Convert the following prefix and postfix expression into infix
   - i) ^ - * +ABC-DE+FG
   - ii) AB^C*D-EF/GH+/+
   - iii) –A/B*C$DE
8. Write an algorithm to convert an infix expression to postfix expression
9. Write an algorithm to evaluate a postfix expression.

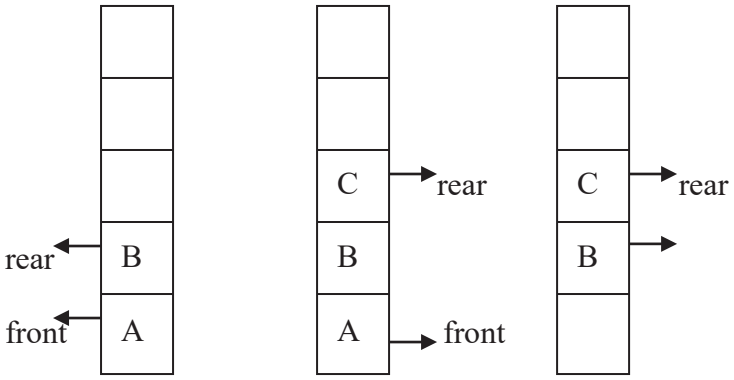
Programs

1. Write a C function to define stacks using arrays.
2. Write a C function to convert and to display the binary form of an integer.
3. Write a C function to test whether a string of opening and closing parenthesis is well formed or not.

4. Write an algorithm to determine if the input character string is of the form
   x C y
   where x is a string consisting of letters ‘A’ and ‘B’ and y is the reverse of x. (that is, if x=”ABBABA”, y must be equal “ABBABA”). At each point you may read only the next character of the string.
5. Write a C function to copy one stack into another.
   i) The stack is implemented using array
   ii) The stack is implemented using linked list
6. Write a C function to evaluate a postfix expression.
7. Design a method for keeping two stacks within a single linear array $[spacesize] so that neither stack overflows until all of the memory is used and an entire stack is never shifted to a different location within the array. Write C routines push1, push2,pop1 and pop2 to manipulate the two stacks(Hint: The two stacks grow towards each other)
8. The Bashemin Parking garage contains a single lane that holds up ten cars. There is only a single entrance / exit to the garage at one end of the lane. If the customer arrives to pick up a car that is not nearest the exit, all cars blocking its path are moved out, then the customer’s car is driven out and the other cars are restored in the same order that they were in originally. Write a program that processes a group of input lines . Each input line contains an ‘A’ for arrival or a ‘D’ for departure, and a license plate number. Cars are assumed to arrive and depart in the order specified by the input. The program should print a message whenever a car arrives or departs. When a car arrives, the message should specify whether or not there is room for the car in the garage. If there is no room, the car leaves without entering the garage. When a car departs the message should include the number of times that the car was moved out of the garage to allow other cars to depart.
9. Write a program to convert fully parenthesized infix expression into postfix.

3. Queues

Definition **:** Queue is a linear list in which insertion takes place at one end called as *rear* and deletion takes place at the other end called as *front*. Hence, in this list, the first one inserted is the first one to be deleted. Therefore, it is called as First in first out (FIFO) list.



3.1 Queue as ADT

```
Abstract typedef QUEUE( etype, front,rear)
Typeof(rear,front)=int;

Abstract empty(q)
QUEUE(etype,front,rear) q;
```
Postcondtion: if the front==rear return 0 else return 1;

```
Abstract insert(q)
QUEUE(etype,front,rear) q;
```
Precondition: `if rear!=MAXSIZE-1`
Postcondition: Insert the value at the rear th position

```
Abstract etype remove(q)
QUEUE(etype, front,rear) q;
```
Precondition: if the queue is not empty
Postcondition: Return the front value.

Abstract etype full(q)

QUEUE(etype, front,rear) q;
Postcondition: if the rear!=MAXSIZE-1 then return 0 else return 1;

3.2 Array Implementation of Queue

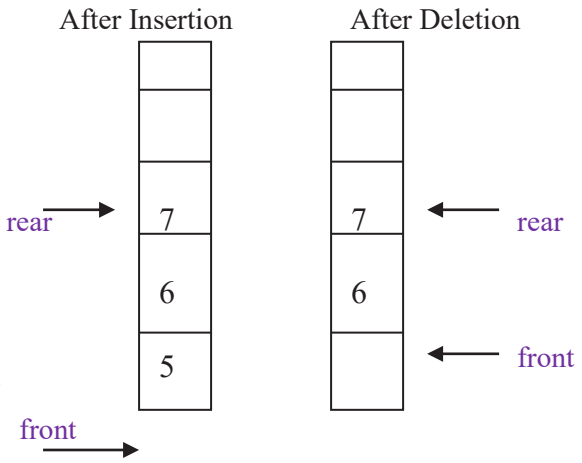
3.2.1 Linear Queue

Functions of Linear Queue:

1. During the initialization of the queue, front and rear are set to –1.
2. On each addition, rear is incremented by one.
3. One each removal, front is incremented by one.
4. Addition cannot be done if queue is full and removal cannot be done if queue is empty.

5. If the queue is non-empty, then “front” denotes minimum, one less than rear.
6. The length of the queue is rear-front.
7. If the condition front==rear is true, then the queue is empty.

```
#include<stdio.h>
#define MAX 10          /*Maximum size of the queue is 10*/

struct queue
{
   int arr[MAX];
   int rear,front;
};

void insertq(struct queue *p, int val)
{    if(p->rear==MAX-1)
     {      printf("The queue is full");
     }
     else
     {     p->arr[++p->rear]=val;
     }
}

int removeq(struct queue *p)
{
   if(p->rear==p->front)
   { printf(“Queue is empty”)
     exit(0);
   }
   else
   { return p->arr[++p->front];
   }
}

void main()
{
   struct queue  q;
   int ch,no;
   q.front=-1;
   q.rear=-1;
   insertq(&q,5);
   insertq(&q,6);
   insertq(&q,7);
   printf(“%d”,removeq(&q));  →7
}
```



Drawback of Linear Queue

With this version of “removeq()” function, it is quite possible that “insertq()” function does not add an item to the queue, even if there is a room, because of queue-full condition. Once the rear reaches MAXSIZE-1, there is no provision given to change the value pointer of the array. To overcome this problem, one may shift the “Elements” array towards left whenever “rear” is seen to be equal to “MAX-1”. But such a solution is certainly time consuming, especially if, the number of elements in a queue is large at time when “queue_full” condition is fulfilled. To overcome this drawback, circular queues are implemented.

3.2.2 Circular Queue

A more efficient queue representation using arrays is achieved through *circular queue*. In this concept, the array is assumed to be circular. The index of the array increases in clockwise direction except for the index MAXSIZE-1. "rear" points to the element which is at the rear end of the queue and "front" points to anti-clockwise neighbor of the beginning of the queue. In this concept, we always sacrifice one space, which means that one space always remains empty.

Functions of Circular Queue

1. Initially, "front" and "rear" be initialized to zero.
2. When an element is to be inserted, the front is incremented by one and then on that rear position the value is placed.
3. When an element is to be removed, the front is incremented by one and on that front position the value is removed.

4. If the front value and the rear value go beyond the MAXSIZE-1, we take the modulus(%) of MAXSIZE and then the rear and front is brought between 0 and MAXSIZE-1.

```
front=front%MAXSIZE
rear=  rear%MAXSIZE
```

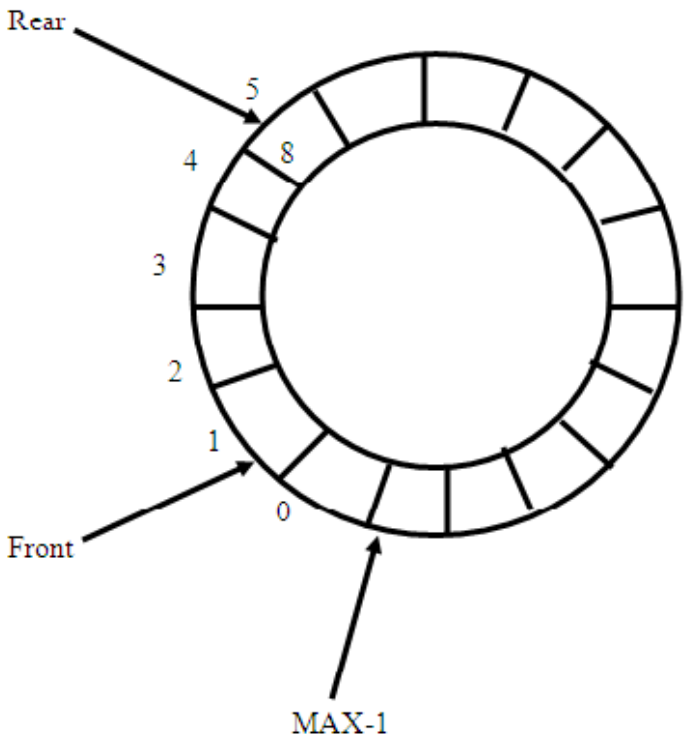
5. If "front" and "rear" are equal, we consider the queue to be empty.

```
front=rear
```

6. In order to get the full condition, we always sacrifice one space. *i.e.,* If

```
front+1=rear
```

then it is considered that the queue is full.



Array Implementation of Circular Queue

```
#include<stdio.h>
#define MAX 5

struct queue
{     int arr[MAX];
      int rear, front;
};

int isempty(struct queue *p)
{
 if(p->front==p->rear)
    return 1;
 else
   return 0;
}

void insertq(struct queue *p, int v)
{
 int t;
 t=(p->rear+1)%MAX;
 if(t==p->front)
 {   printf("overflow");
 }
 else
 { p->rear=t;
  p->arr[p->rear]=v;
 }
}

int removeq(struct queue *p)
{
    if(isempty(p))
    { printf("queue underflow");
      exit(0);

    }
    else
    { p->front=(p->front+1)%MAX;
      return (p->arr[p->front]);
    }
}

void main()
{
  struct queue q;
  char ch;
  int no;
  q.rear=q.front=0;
  insertq(&q,7)
  insertq(&q,10);
  insertq(&q,12);
  insertq(&q,15);
  insertq(&q,8);
  printf("%d", removeq(&q)); → 7;
```

```
}
```

3.2.3 Deque

Definition : A deque (deck) is a queue in which the insertion and deletion can be done in both the ends, namely, left and right. In this concept, one array is used to maintain two queues. Both the queues grow toward each other.

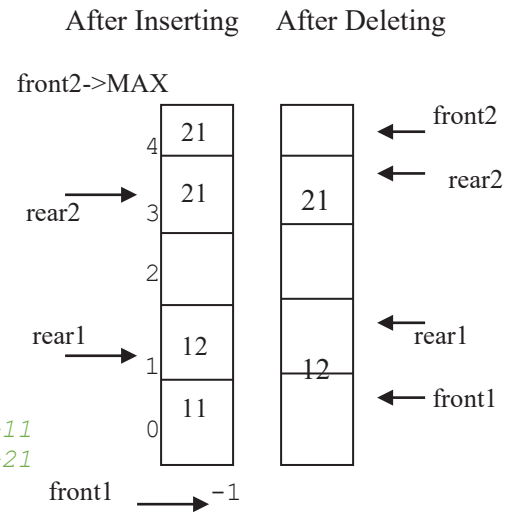
Functions of deque:

1. Initially, front1 and rear1 are initialized to –1 and front2 and rear2 initialized to MAX.
2. *Insertleft* inserts a value on the array using the incremented rear1. *Insertright* inserts using decremented rear2.
3. *Removeleft* removes a value from the left side of the array using incremented value of front1. Similarly *removeright* removes a value from the array using decremented value of front2.
4. If rear1+1 =rear2, then no insertion operation can be done on any of the queue.
5. If rear1=-1 then no deletion can be done from the left queue. Similarly, if rear2=MAX-1, then no deletion can be done from the right side queue.

```
#include<stdio.h>
#define MAX 5

struct deque
{
      int arr[MAX];
      int rearleft,rearright,frontleft,frontright;
};
void insertleft(struct deque *p,int v)
{
      if(p->rearleft+1==p->rearright)
      {printf("DEQUE OVERFLOWS");
      }
      else
      {     p->arr[++p->rearleft] = v;
      }
}
void insertright(struct deque *p,int v)
{
      if(p->rearleft+1==p->rearright)
      {           printf("DEQUE OVERFLOWS");
      }
      else
      {           p->arr[--p->rearright] = v;
      }
}
int removeleft(struct deque *p)
{
      if(p->frontleft==p->rearleft)
      {
            printf("DEQUE UNDERFLOWS");
            return 0;
      }
      else
      {
            return p->arr[++p->frontleft];
      }
```

```
}
int removeright(struct deque *p)
{
      if(p->frontright==p->rearright)
      {
            printf("DEQUE UNDERFLOWS");
            return 0;
      }
      else
      {
            return p->arr[--p->frontright];
      }
}
```

```
main()
{
      struct deque q;
      int no,ch;
      q.rearleft=-1;
      q.frontright=MAX;
      q.rearright=MAX;
      q.frontleft=-1;
      insertleft(&q,11);
      insertleft(&q,12);
      insertright(&q,21);
      insertright(&q,22);
      printf("%d",removeleft(&q));   ->11
      printf("%d",removeright(&q)); ->21
      return 0;
}
```



3.3 Priority Queue

A *priority queue* is a data structure in which operation on the elements do not happen by the natural order, FIFO, but happens on the priority of the element. The elements of the priority queue need not be an integer or a character that can be compared directly, but it may be a complex structure that is ordered on several fields.

The priority queues are classified into two:

1. Ascending priority queue
2. Descending priority queue

3.3.1 Ascending Priority Queue

Ascending Priority queue is a collection of items into which items can be inserted arbitrarily and from which only the smallest can be removed. The two functions of ascending priority queue is *insert()* and *removemin()*. A linear queue is considered as an ascending priority queue because the element that is inserted initially gets deleted first. On the basis of time, it is considered as ascending priority queue.

3.3.2 Descending Priority Queue

A descending priority queue is a queue in which items can be inserted arbitrarily, but only the maximum value can be deleted. The two functions of descending priority queue is *insert* and

removemax(). For example, stack is called as descending priority queue, because the last inserted value is removed first. In this case, the priority is been determined by the time of insertion.

3.3.3 Draw back of using an array in priority Queues

Whenever an element is to be removed, there are two processes involved, shown as follows:

1. The smallest or the largest element of the queue needs to be searched. Therefore the average search we make, in order to, search the elements is n/2.
2. After removing the elements, the array has to be shifted so as to fill the empty space. In this case, if the array is very large much time is consumed to shift the elements.

Hence, in ***priority queues,*** the execution time is been considerably increased. This draw back is overcome using linked list.

3.4 Applications of Queues

1. Spooling: When a job is submitted to the printer, they are arranged in the order of arrival. Thus, jobs sent to a line printer are arranged in the order of arrival.

2. **Networking:** In network of computers, disk may be attached only to one computer. That machine is known as a server. The users can access the disk in the first come first serve basis.

3. Topological Sort: If a task numbered 'i' must be finished before the task 'j', 'j' must be finished before 'k' and so on, and if all the precedence of the task comes as an input, then a diagram needs to be built in the runtime to sort the task in ascending order of their precedence. Such an ordering is called as *topological sorting* of the tasks. To sort the task, queues have been used.

4. Calls to larger companies are placed on queue when all operators are busy.

Questions

1. Write a short note on
   a).Linear queue
   b).Circular queue
   c).Priority queue
   d).Deque

2. Define ADT for all the above queues.
3. Define and implement linear queue using arrays.
4. Give a data structure definition to implement circular queue using arrays.
5. Write a C procedure to implement Deque
6. What is priority queue? Explain the drawbacks while implementing them using arrays.
7. Explain the applications of queues.

4. Recursion

Definition : A construct operating on itself is called as recursion. In C, if a function calls itself, then it is called as recursive function. In recursive function, a problem is reduced into a smaller instance of the same problem.

4.1 Design of Recursive Function

A recursive definition of *f(n)* must meet the following requirements:

1. The definition must include the **base component,** in which *f(n)* is defined directly(non recursively) for one or more values of n (that the base covers the case n≤k ,for some constant k).
2. In the **recursive component,** all occurrences of *f* have a parameter smaller than *n,* so that repeated applications of the recursive component reduce all occurrences of *f* to the base component.

```
int fact(int n)
{
      if(n==0)
     return 1;

      else
      return n*fact(n-1);
}
```

Base Component

Recursive Component

3. **Design Rule:** Assume that all recursive calls work. While designing recursive programs, it is not necessary to know the details of the bookkeeping arrangements. Rather, it is extremely difficult to track down the actual sequence of recursive calls and bookkeeping arrangements.

4. **Never duplicate** work by solving the same instance of a problem in separate recursive calls.

4.2 Types of recursion

Recursive functions are classified into two:

1. **Direct recursion**
   If a function calls itself, then it is called as direct recursion. This means that a code of function *f(n)* contains a call of *f(n).*
   f(n) → *f(n)*

2. **Indirect recursion**
   If a function calls another function that directly or indirectly calls the former, then it is called as indirect recursion. This means that the code of function *f(n)* contains a statement that calls *g(n)* and *g(n)* invokes *f(n)* again.
   f(n)→g(n)→f(n)

4.3 Analysis on a function call

When a function calls an another function, the calling function is suspended from execution and the control is given to the called function. The calling function remains suspended until the control returns from the called function. The values of the local variables and parameters of the calling function remain unchanged until the control returns except the reference variables.

For every call a new set of parameters and local variables are created. To store the details of local variables and address where the value to be return are stored in a stackframe and then bushed into the system stack. When a function concludes the stackframe is popped out and the return value is stored then the calling function resumes the work from where it was suspended.
A stack frame contains four different elements.

1. Passing arguments
2. Allocating and initializing local variables
3. Returning values and Transferring control to the function

4.3.1 Passing Arguments: When a function is called, the parameter gets its value assigned. Hence, new memory is created within the area of the memory. All changes made to the parameters are reflected only within the block.

4.3.2 Allocating and initializing local variables: After arguments are passed, local variables are created. These local variables include all those declared in the function as well as the variable that would be created during the program execution.

4.3.3. Transferring control to the function: After executing a function, it returns a value to the called function. In this case, the function should know where the value is to be returned. Hence, along with the arguments of the function, implicit addresses are passed to the called function so that the return value may be stored in a specified location.

When a function returns a value, three actions are performed:

1. The return address is retrieved and stored in a safe location.
2. The function's data area, such as local variable, arguments, is freed.
3. The return value is placed in a secure location from where the calling program can retrieve it.

After these procedures the calling function resumes its works.

4.4 Stacks and recursion

Every language that supports recursive function uses a stack of activation records so as to record the values of all the variables belonging to each active function of a program. This is known as stack frame. When a program P is called, a new activation record for P is placed on the stack irrespective of any other activation record placed on the stack. When P returns, its activation record must be on the top of the stack. Since P cannot return until all functions, it is called returned to P. Thus we may pop the activation record for this call of P to cause control so as to return to the point at which P was called.

4.5 Advantages and disadvantages of recursion

4.5.1 Advantages

1. In recursion, problems are decomposed into smaller sub-problems. Sub- problems are resolved and finally the solutions of the sub-problems are combined and given the original answer. Hence, logic becomes simple.
2. The code becomes precise so it becomes easy to debug or modify them.

4.5.2 Disadvantages

1. Due to repeated overhead of the calling functions, Recursive programs are time consuming.
2. Each time a recursive function calls itself, a new memory for data is created for the local variables, parameters and for the return address. Thus it consumes more space.
3. A string of return address forms a stack, hence, the most recent address to be added is the first to be removed. Otherwise each call will use the same area of local variable and destroy the values in the storage space for the outer calls and the outer calls cannot be completed successfully.
4. Since recursion implicitly uses stack data structure, results once obtained are popped and not remembered thereafter. Hence *fib(2)* is called so many times. If some other data structure could have been used, then, the old value could be preserved.
5. Many programming languages do not support recursive functions.

4.6 Tail recursion

If a function calls itself at the last line of execution, it is called as *tail recursion*.(Eg. The last line of tower of Hanoi program is the call to itself).

4.6.1 Removal of Tail recursion

In the tower of Hanoi problem, the last statement is a call to itself. To remove the tail recursion, the values of the actual parameters 'to' and 'aux' in the last recursive call to tower are the values of 'from' and 'aux' in the previous call. These values are to be swapped before branching to the beginning of the function after eliminating recursion.

```
void move(int n, char from, char to, char aux)
{
   char  temp;
    there:
if(n==1)
{
   printf("Move disk from %c to %c\n", from, to);
}
else
{   move(n-1, from, aux, to);
    printf("Move the disk from %c to %c\n",from, to);
    n=n-1;
    temp=aux;
    aux=from;
    from=temp;
    goto there;
}}
```

4.7 Divide and conquer

The divide-and conquer is an approach where a large problem is divided into smaller problems. To solve a large instance, we

(1) Divide the instance into two or more smaller instances,
(2) Solve each of these smaller problems
(3) Combine the solutions of these smaller problems to obtain the solution to the original instance, the smaller instances are often instances of the original problem and may be solved using the divide-and conquer strategy recursively.

Eg. i) Quick Sort
ii) Merge Sort
iii)Binary Search

4.7.1 Binary Search

In an ordered array, when an element is to be searched, we compare the elements in linear order. This approach is called as linear search.
In the binary search approach the array is divided into half. If the middle value is equal to the value to be searched, then we say that the value is found. If the value is greater than the middle value, then we search the value in the second half, if less, we search the value in the first half. The above procedure is continued till the lower subscript becomes greater than the higher subscript or the higher subscript become less than zero.

4.7.2 Algorithm for binary search

```
int bsearch(int a[], int val, int low, int high)
{

        if(low>high or high<0)
    Value is not found
        Else
        Take the average of low and high call as mid
        If ( a[mid]==val)
       Value is found
        Else
    If(a[mid]>val)
    Assign mid-1 to high
        Else
    Assign mo\id+1 to low
    Call the function again with the new low or high value
}
```

4.7.3 C function for binary search.

```
include<stdio.h>

int bsearch(int a[],int val, int low,int high)
{
    int mid;
```

```
    if(low>high||high<0)   /*if the low value is greater than high or
high value is less than 0 then the value is not present*/
    {return -1;
    }
    else
    {  mid=(low+high)/2;
       if(a[mid]==val)
       {  return mid;
       }
       if(a[mid]<val)
       low=mid+1;
       else
       high= mid;
       return bsearch(a,x,low,high);
    }
}


void main()
{     int arr[10]={1,2,3,4,5,6,7,8,9,10};
      int no,i;
      printf("Enter the number to be searched");
      scanf("%d",&no);
      i=bin(arr,no,0,9);
      if(i==-1)
      { printf("Not available");
      }
      else
      {
        printf("Available at the %d position", i);
      }
}
```

4.8 Backtracking

Backtracking algorithms attempt to complete a search for a solution, by constructing partial solutions, always ensuring that the partial solutions remain consistent with the requirements of the problem. The algorithm then attempts to extend a partial solution toward completion, but when an inconsistency with the requirements of the problem occurs, the algorithm backs up (**backtracks**) by removing the most recently constructed part of the solution by trying another possibility.

Using Backtracking we solve

1. N – Queen's Problem

4.8.1 N- Queen's Problem

In this program, we try to find all ways to place 'n' queen in a nxn chess board. We represent the problem in an array 'x'. x[i] represents the column of the i^{th} queen on the i^{th} row.

When we place the queen on the k^{th} row, the following conditions need to be satisfied:

1. No two queens can be placed at the same row
   i ≠ k

2. No two queens can be placed at the same column
   x[i]≠ x[k]
3. No two queens can be placed at the diagonal
   | i-k | = | x[i]-x[k] |

AddQueen() is a function which tests whether a queen can be placed at x[k] th column at the kth row, by checking the above conditions.

Recursive N Queen:

```
#include<stdio.h>

/*It returns true if the queen can be placed in the kth row */
int AddQueen(int Q[], int k)
{         int i;
      for(i=0;i<k;i++)
      {
            if(Q[i]==Q[k]||(abs(Q[i]-Q[k])==abs(i-k)))
            return 0;
      }
       return 1;
}
void NQueen(int Q[],int k,int n)
{
int i,j;
for(i=0;i<n;i++)
{     Q[k]=i;
   if(AddQueen(Q,k))
   {
       if(k==n-1)
       {
      for(j=0;j<n;j++)
      printf("%d",Q[j]);
      printf("\n");
       }
       else
        NQueen(Q,k+1,n);
   }
}
}

void main( )
{     int n,Q[20]={0}, i;
      printf("Enter the number of Queens");
      scanf("%d",&n);
      NQueen(Q,0,n);
 }
```

Tower of Hanoi

In this problem, there are n disks, arranged in a stand in the decreasing order of size where smaller disks are placed on the larger ones. The program gives the recursive solution to shift the n disks from one peg to another peg with the following conditions:

1. Only one disk can be moved at a time
2. A larger disk should never be placed upon the smaller ones.

3. Only one auxiliary stand can be used for the intermediate storage of disks.

The recursive algorithm to this problem

Assume that there exist three stands: source, destination and auxiliary

`Case 1:` If there is only one disk, then shift from source to Destination.

`Case 2:` If there is more than one disk,

1. Shift n-1 disks to the auxiliary using destination stand as the intermediate
2. Then shift the nth disk to destination
3. Shift n-1 disk from auxiliary to the destination-using source as the intermediate stand.

The total number of shifts for n disks is 2^n-1.

4.9 Comparison of iteration and recursion

	Iteration	Recursion
Initialization	The loop variable is initialized	The variable is initialised in the form of arguments
Condition	The condition is used to determine whether further looping is necessary	The argument values are used to determine whether further recursive calls are required.
Computation	The necessary computation is performed within the block	The computation is performed using the local variables and the parameters at current depth.
Updation	The decision parameter is updated and a transfer is made to the next iteration.	The update is done so that the variable can be used for further recursive calls.
Efficiency	It does not return value. Therefore, it is easier to store the values. However, if the problem itself is recursive in nature, an iterative solution will have to stimulate **stack**.	Additional amount of work must be done to save the return address So that the correct statement is executed. The function saves formal parameters and local variables on to the stack upon entry and restores them on completion. For every function call a stack is created. Therefore, it consumes much memory and turns out to be less efficient.

4.10 Converting a recursive algorithm to a non-recursive algorithm

Common Initialization:

1. Declare a stack that will hold the activation records consisting of all local variables, parameters called by value and label to specify where the function is

recursively called. The stack and stack top pointers are declared as global variables. Initialize the stack as empty.

2. To enable each recursive call to start at the beginning of the original function, its first executable statement, after the stack initialization block, is associated with a label.

For Every recursive call:

3. Push all the local variable and parameters called by value into the stack.
4. Push an integer 'i' into the stack, if this is the i^{th} place, from where the function is called recursively.
5. Set the formal parameters called by value to the values given in the new call to the recursive function.
6. Replace the call to the recursive function with a 'goto' to the statement that is the first statement after the stack initialization block. A label is associated to this statement in rule 2.
7. Make a new statement label Li(if this is the i^{th} place where the function is called recursively) and attach the label, to the first statement after the call to the same function

Before returning to Calling function

8. If the stack is empty, then the recursion is finished, make a normal return.
9. Otherwise pop the stack to restore the values of all local variables and parameters called by value.
10. Pop the integer "i" from the stack and use this to go to the statement labeled Li.

Questions

1. Define recursion? Explain recursion with an example.
2. Compare recursion with iteration?
3. Explain the efficiency of recursion? Or advantages and disadvantages of recursion?
4. What is a tail recursion? Explain with an example?
5. Write a short note on tower of Hanoi problem?
6. Prove that tower of Hanoi makes 2^n-1 moves to shift n disks.
7. What is Ackerman's function? What would be the output if n=2,m=2;
8. Explain divide and conquer approach? How is recursion used in this approach? Explain with an example.
9. Define backtracking? Give an example.
10. How can recursion be eliminated?
11. How will you eliminate the tail recursion in tower of Hanoi?
12. Explain the relationship between stack and recursion?

Programs:

Using recursion:

1. Numbers
   - i) Find the factorial of a number
   - ii) Multiply a and b
   - iii) Add a and b by succession
   - iv) Find GCD : Elucid's Algo /Brute Force Algorithm/ Dikstra's Algorithms
   - v) Fibonacci of number of n
   - vi) Prime Number

vii) Power function

2. Arrays
   i) Find minimum and maximum of an array
   ii) Sum of the array
3. Strings
   i) Reverse a string
   ii) Palindrome
   iii) Finding the length of the string
   iv) Compare two String
   v) Finding the substring
   vi) Converting string of numerals into number
   vii) Converting prefix to postfix/infix
   viii) Converting postfix to prefix/infix
4. BackTracking
   i) Eight queens problem
5. Divide and Conquer
   i) Tower of Hanoi iii) Sort an array using merge sort and quick sort
   ii) Binary search

5. Linked List

Definition : Suppose L is the linear list consisting of $e_1,e_2,e_3,\ldots..e_n$. In linked list representation for this list, each element e_i is represented in a separate node. Each node is stored in random memory location. Every node has two parts: ***data field*** and the ***link (address) field***. The *data field* stores data and the *link field* is used to represent next element in the linear list. The e_n node, where n is the number of nodes, has no link due to which its link field is set to NULL. The list with no nodes are called as ***empty list*** or ***Null list***
Linked list is classified into three:

1. Single Linked list
2. Double Linked list
3. Circular Linked list

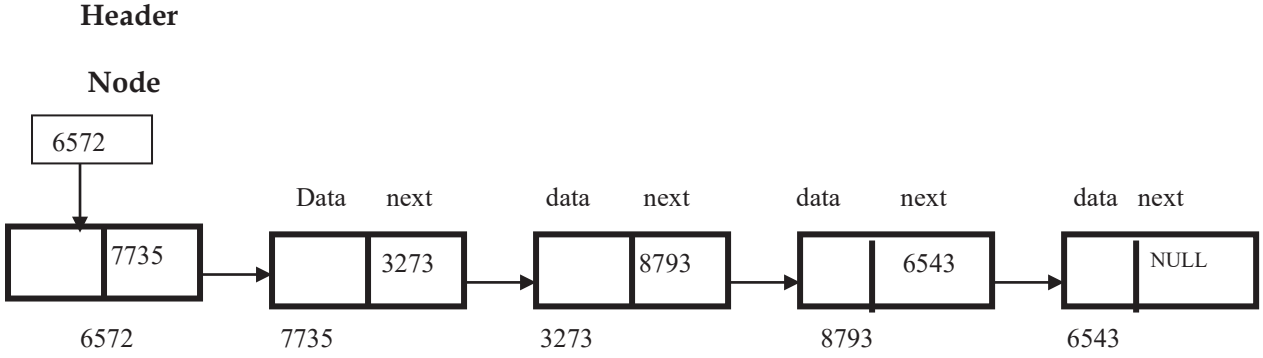
Requirements to Implement linked list using pointers

When a linked list is to be represented using pointers, it requires three facilities:

1. *Acquiring memory for storage during run-time*: In C programming language, memory can be acquired through *malloc()*. The *malloc()* function dynamically allocates memory from the system until the system has enough memory to acquire. It allocates memory from the *heap*.
2. *Dividing the memory into two parts like data part and link part*: The free() function returns the chunk of memory to the system which can be acquired through the concept of *self-referential structure.*
3. *Freeing memory,* during execution of program, once the need for the acquired memory is over.

5.1 Single Linked List

A single linked list consists of series of nodes, which are not necessarily adjacent in memory. Each node contains an element and a pointer to a structure containing the address of its successor. We call this as *next* pointer. The last cell's *next* pointer points to NULL. The Ansi C specifies that NULL is zero. Since the first node links to the second and the second to the third…and the last has the NULL, this list is called as ***singled linked*** list or a ***chain***.



Single Linked list

Header node

1. Header node contains the pointer to the current or the last node of the list. It is identical to type of the list of nodes
2. During the traversal of the list, the value of the header node is assigned to the current. Hence it becomes easy for traversal.

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

/* Defining the node*/
struct node
{
   int data;
   struct node *next;
};
```

ADD()

Function: This function adds the 'newnode' at the beginning. The last node inserted will always be connected with the front.

Algorithm:

1. Create a 'newnode'
2. Assign the 'val' to 'newnode->data';
3. Transfer the content of 'front' to 'newnode->next'.
4. Assign 'newnode' to 'front'.

```
void add(struct node **front, int val)
{    struct node *newnode;
      newnode=(struct node*)malloc(sizeof(struct node));
      newnode->data =val;
      newnode->next=*front;
     *front=newnode;
}
```

Display()

```
/*    Function: Displays the content of each node.
```

Algortihm:

1. Assign the front to cur.
2. while(cur is not NULL)
3. print cur->data
4. Assign cur to cur->next

```
*/
void  display (struct node **front)
{
      struct node *cur;
      cur=*front;
      while(cur!=NULL)
      {
     printf("%d->",cur->data);
     cur=cur->next;
      }
```

```
}
```

---

del()

Function: Frees the node that contains the value 'val' from the list and stores the address of cur->next in previous node.

Algorithm:

1. Search for the node and the previous node that contains the value 'val'
2. If the cur is NULL print value 'val' is not front
3. if not, assign 'cur->next' to 'prev->next'
4. free the 'cur'

```
void del(struct node **front,int val)
{
      struct node *curr,*prev;
      curr= prev=*front;
      while(curr->data!=val&&curr!=NULL)
      {
            prev=curr;
            curr=curr->next;
      }
      if(curr==NULL)
      printf("Value not found");
      else
      if(prev==curr)
      {
            *front=curr->next;
      }
      else
      {     prev->next=curr->next;
            free(curr);
      }
}
```

SEARCH()

Function: It searches for the given value 'val' in the list. If fount, it returns TRUE else FALSE.

Algorithm:

1. Assign front to cur
2. while(cur->data!=val and cur!=NULL)
   { cur =cur->next
   }
3. if cur is NOT NULL, return TRUE else FALSE

```
int search( struct node **front, int val)
{
   struct node *cur;
     cur=*front;
   while(cur!=NULL)
   {
       cur=cur->next;
   }
   return  cur!=NULL; }
```

count()

Function: To count the number of nodes connected in the list.

C code:

```
int count(struct node **front)
{
   int c=0;
   struct node *cur;
   cur=*front;
   while(cur!=NULL)
   {  c++;
     cur=cur->next;
   }
   return c;
}
```

Copy()

Function: The contents of one list (front1) are copied into another list (front2). Every time a newnode is created, until the first list is entirely read.

Algorithm

1. Assign the address of front1 to cur1 to traverse
2. while(cur1 is NOT NULL)
   {
   create a newnode
   copies the cur->data to newnode->data
   if (front2==NULL)
   Assign newnode to front
   Newnode->next =NULL
   Else
   Assign the address of the newnode to the last node in the list
   }
   assign cur=cur->next

}

C code:

```
void copy(struct node **front1,struct node **front2)
{
  struct node *newnode,*cur1=*front1,*cur2;
  while(cur!=NULL)
  {         newnode=(struct node*)malloc(sizeof(struct node));
          if(*front2==NULL)
          {
            *front2=newnode;
            cur2=*front2;
          }
```

```
        else
        {
          cur2->next=newnode;
          cur2=newnode; }
        newnode->next=NULL;
        newnode->data=cur->data;
        cur=cur->next;
       }
   }
```

```
void main()
{
  struct node *front1=NULL,*front2=NULL;
   int no,val;
  /* Depend on th requirement call the function*/
 }
```

5.1.1 Stacks using Linked List

Concept:

1. The initial value of front is NULL.
2. If the front is NULL, then it is considered that the stack is empty.
3. Whenever a node is added, it is always added at the beginning of the list. This is to say, that the address of the last inserted node is always stored at the 'front'.
4. Since the address of the last inserted node is stored at the 'front', we free the node whose address is been stored at the front.

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

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

PUSH()

```
/* Algorithm and function are same as the add function above*/
void push(struct stack **front, int val)
{
  struct stack *newnode=(struct stack*)malloc(sizeof(struct stack));
  newnode->next=*front;
  *front=newnode;
  newnode->data=val;
}
/*
```

POP()

Function: Frees the node which is connected with the front and assigns the address of 'next' to 'front'. If the front is 'NULL' then it is considered that the stack is empty.

Algorithm:

1. If the front is NULL, then stack is empty
2. If not, Assign the address of front to cur
3. Assign front=cur->next
4. Assign val=cur->data
5. free (cur)
6. Return val

C code:

```
*/
int pop(struct stack **front)
{  struct stack *cur;
   if(*front==NULL)
   {
     printf("The stack is empty");
     return 0;
   }
   else
   {  int val;
       cur=*front;
       val=cur->data;
      *front=cur->next;
       free(cur);
       return val;
   }
}

void main()
{
struct stack *front=NULL;
push(&front,7);
push(&front,9);
push(&front,11);
printf("%d\n",pop(&front));
printf("%d",pop(&front));

}
```

Output:

```
11
9
```

5.1.1.1 Difference between Stacks using linked list and arrays

1. Each stack is able to grow and shrink to any size. When a stack needs a node, it can obtain it where and when required.
2. No space is pre-allocated to any single stack and no stack uses space that is not required.
3. The initial total amount of space required for a stack is less than the run time.

5.1.2 Queues using Linked List

Explanation:

1. In order to maintain the beginning address and the end address, two pointer variables have been declared: 'front' and 'rear'. Both of them are initialized to NULL.
2. If the front is NULL, then it is considered that the queue is empty.

3. ‘front’ stores the address of first inserted node and ‘rear’ stores the address of the last inserted node.
4. A node to be inserted ‘rear->next’ stores the ‘newnode’ and ‘rear’ stores the address of ‘newnode’.
5. The address of the node that is been stored in the ‘front’ is always removed.
6. ‘rear’ and ‘front’ are combined in a struct queue for the sake of convenience.

```
#include<stdio.h>

struct node              /*definition of the node with two members*/
{
  int data;
  struct node *next;
};

struct queue              /* combining the pointers front and rear as one
{                         struct variable*/
    struct node *rear,*front;
};

/* Insert()

void insert(struct queue *p, int val)
{ struct node *newnode;
  newnode=(struct node*)malloc(sizeof(struct node));
  newnode->data=val;
  newnode->next=NULL;
  if(p->rear==NULL)          /* rear is NULL then make rear and front to newnode*/
  {  p->front=p->rear=newnode;
  }
  else
  {  p->rear->next=newnode; /* Pass rear->next to newnode->next
                            assign newnode to rear*/
    p->rear=newnode;
  }
}
int removeq(struct queue *p)
{  int val;
  struct node *cur=NULL;
  if(p->front==NULL)
  {  printf("Queue is empty");
     return 0;
  }
  else
  { val=p->front->data;
    cur=p->front;
    p->front=p->front->next;
    if(p->front==NULL)
    {p->rear=NULL;
    }
    free(cur);
    return val;
  }
}
```

```
void main()
{
   struct  queue s;
   int val,ch;
   s.rear=NULL;
   s.front=NULL;
   insert(&q,7);
   insert(&q,10);
   insert(&q,11);
   printf("%d",removeq(&q));→7
}
```

5.1.3 Priority Queues using Linked List

Explanation:

1. Initial value of 'front' is set to NULL.
2. If the 'front' is NULL, the queue is empty.
3. newnodes are added in the list depending on the value. Hence, a search is been made for the appropriate position.
4. If the newnode->data has lesser value than the cur node then it is added before the current node. If not, it is added after the current node.
5. 'front' stores the lowest value. Hence, the node whose address is been stored in the front is always removed.

```
Struct node
{
   int data;
   struct node *next;
};
void insertq(struct node **front,int val)
{
  struct node *cur,*prev,*newnode;
  cur=prev=*front;
  newnode=(struct node*)malloc(sizeof(struct node));
  newnode->data=val;
  if(*front==NULL)
  {    *front=newnode;
       newnode->next=NULL;
  }
  else
  {  while(cur->data<newnode->data&&cur!=NULL)
     { prev=cur;
       cur=cur->next;
     }
     if(*front==cur)
     {  newnode->next=*front;
       *front=newnode;
     }
     else
     {    newnode->next=prev->next;
          prev->next=newnode;
     }
  }
}

int removeq(struct queue *p)
```

```
{  int val;
  struct node *cur=NULL;
  if(p->front==NULL)
  {  printf("Queue is empty");
     return 0;
  }
  else
  { val=p->front->data;
    cur=p->front;
    p->front=p->front->next;
    if(p->front==NULL)
    {p->rear=NULL;
    }
    free(cur);
    return val;
  }
}
main()
{
     struct node *front;
     insertq(&front,21);
     insertq(&front, 7);
     insert(&front, 11);
     insert(&front,8);
     printf("%d", removeq(&front));
}
output: 7
```

5.1.3.1 Advantage of defining Priority queues using Linked list

In the array implementation of priority queues, when an element is to be removed, there are two procedures involved:

1.Searching for the element
2. Shifting the array

Whereas the priority queue implementation of linked list requires an average of n/2 nodes examination for insertion, since a node is inserted before its lowest value; but for the removal of the node it requires only one node examination, since while inserting we insert either in ascending or descending order.

Also, in linked list, there is no shifting involved because while insertion or deletion only the pointer is redirected. Hence, there is no empty space left out. Whereas, in arrays it is impossible to maintain a queue without an empty space, in an unordered list, the examination of nodes for insertion is one where the examination of nodes for deletion is ‘n’ because it traverses (visits once) the entire list once. Therefore, the examination of nodes becomes ‘n’

5.1.4 Representation and operations on Polynomial Equations

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

struct node
{
  float coeff;
```

```
  int exp;
  struct  node *next;
};
```

*/*NOTE: In a polynomial equation, every term has two components, co-efficient and power. Hence, every node consists of coeff, pow and the next pointer*/*

Function: This function creates a 'newnode' and adds in the list depending on the 'pow' of the 'newnode'.

INSERT()

Function: It creates a polynomial expression. It takes the header node of the list and the newnode to be connected with the list, as a parameter. When a node is to be connected, it inserts the node in an appropriate place after comparing the power.

Algorithm:

1. Let 'front' be the header node of the polynomial expression and initialized to NULL.
2. If the 'front' is NULL assign the newnode to front and newnode->next to NULL.
3. If the 'front' is not NULL, search for the node which has less than or equal power to the newnode.
4. if node in the list has same power as of the newnode then add the coefficients.
5. If not, Assign the previous node->next of the lower powered node to newnode->next.
6. Assign the newnode to previous->next.

C Code:

```
void insert(struct node **front, float coeff, int pow)
 {
  struct node *curr,*prev,*newnode;
  curr = prev = *front;
   if(*front == NULL)
    {
     *front = newnode;
     newnode->next = NULL;
    }
   else
  {
    while(curr->pow > newnode->pow && curr!= NULL)
     {
      prev = curr;
      curr = curr->next;
     }
      if(curr->pow == newnode->pow&&curr!=NULL)
       {
        curr->pow = newnode->pow;
        curr->coeff = curr->coeff + newnode->coeff;
       }
     else if(*front == curr)
       {
     newnode->next = *front;
     *front = newnode;
```

```
        }
       else
        {
      newnode->next = prev->next;
      prev->next = newnode;
        }
    }
  }
```

```
void display(struct node *front)
{
       while(front!= NULL)
       {
        printf("%dx^%d",front->coeff,front->pow);
        front = front->next;
         if(front!=NULL)
         printf("+");
       }
}
```

addpoly()

Function: This function adds two polynomial equations.

Algorithm:

1. Let 'front1' and 'front2' be the header nodes of two polynomial equations. Let 'front3' be the header node of the resultant polynomial expression.
2. Assign cur1=*front1 cur2=*front2.
3. while(cur1 or cur2 is NOT NULL).
   {
   - create a newnode;
   - If cur1->pow is greater than cur2->pow, then copy the contents of cur1 to newnode. Move cur1 to the next position.
   - If cur2->pow is greater than cur1->pow, then copy the contents of cur2 to newnode. Move cur2 to the next position.
   - If both are equal, add the coeffients and store the newnode->coeff. Copy one of the pow into newnode->pow. Move both the pointers to the next postion.
   - Connect the newnode with the front.

   }

```
void addpoly(struct node *front1, struct node *front2,struct node

**front3)
{
  float coeff;
  int pow;
  struct node *cur1,*cur2,*cur3,*newnode;
  cur1=front1;
  cur2=front2;
  while(cur1!=NULL||cur2!=NULL)
  {
       newnode=(struct node*)malloc(sizeof(struct node));

        if(cur1->pow>cur2->pow&&cur1!=NULL)
```

```
        {  pow=cur1->pow;
           coeff=cur1->coeff;
           cur1=cur1->next;
         }
        else if(cur1->pow<cur2->pow&&cur2!=NULL)
        {  pow=cur2->pow;
           coeff=cur2->coeff;
           cur2=cur2->next;
        }
       else
       {
           pow=cur1->pow;
           coeff=cur1->coeff+cur2->coeff;
           cur1=cur1->next;
           cur2=cur2->next;
       }
       insert(front3,coeff,pow);
  }
}
```

PLOYMUL()

Function: It multiplies two polynomial equations. It takes every node of the first equation and multiples it with all the nodes of the second equation.

Algorithm:

1. 'front1' and 'front2' are the two header nodes, consists of two polynomial equations to be multiplied and front3 is the header node where the result is to be stored.
2. Assign cur1=front1, cur2=front2
3. If the 'front1' is NULL, assign 'front2' to 'front3'
4. If the front2 is NULL, assign 'front1' to 'front3'
5. while(cur1 is NOT NULL)
   {
   assign cur2=front2
   while(cur2 is NOT NULL)
   { create a newnode;
   assign newnode->coeff=cur1->coeff+cur2->coeff;
   assign newnode->pow=cur1->pow*cur2->pow;
   insert the newnode at the appropriate place
   move cur2 to the next node
   }
   move 'cur1' to the next node
   }

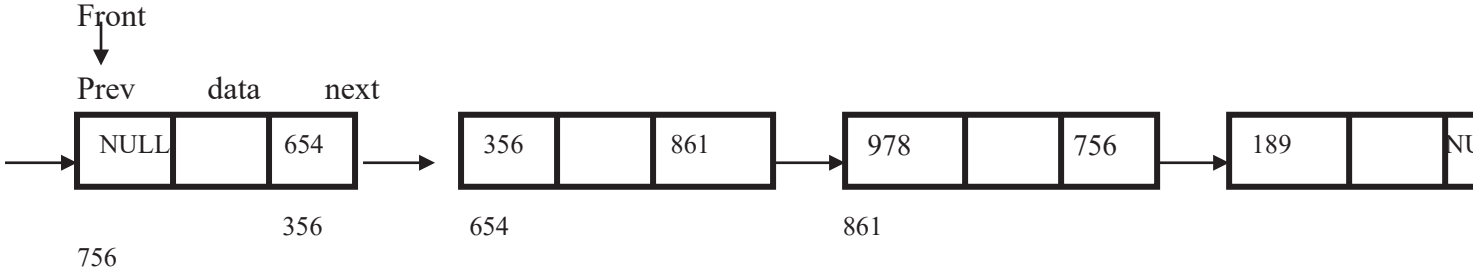
void polymul(struct node *front1,struct node *front2,struct node **front3)

```
{
 struct node *curr1,*curr2,*newnode,*curr3;
 float coeff;
 int pow;
 curr1 = front1;
 curr2 = front2;
 if(front1==NULL || front2 == NULL)
  {
   *front3 = NULL;
```

```
    }
    else
     {
  while(curr1 !=NULL)
   {
      curr2 = front2;

     while(curr2!=NULL)
      {
       coeff = curr1->coeff * curr2->coeff;
       pow = curr1->pow + curr2->pow;

       insert(front3,coeff,pow);
       curr2 = curr2->next;
      }
      curr1 = curr1->next;
    }
    }
  }
}

void main()
{ struct node *first=NULL,*sec=NULL,*tot=NULL,*pro=NULL;
  create(&first,1.5,3);
  create(&first,3,2);
  create(&first,4,1);
  create(&first,-5,0);
  display(first);
  create(&sec,1.5,3);
  create(&sec,3,2);
  create(&sec,4,1);
  create(&sec,5,0);
  display(sec);
  polyadd(first,sec,&tot);
  polymul(first, sec,&pro);
  display(&tot);
  display(&pro);
}
```

5.2 Double linked list

A double linked list is an ordered sequence of nodes in which each node has two pointers: *prev* and *next*. The *prev* pointer points to the node on the left side node and the *next* pointer points to the right side (next) node.




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

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

ADDBEG()

Function: It adds the 'newnode' at the beginning of the list. Hence, in this concept the content of the 'front' is always transferred 'newnode->next' and the 'front' header node always stores the address of the 'newnode'. Here newnode->prev is always NULL.

Algorithm

1. Let 'front' be the header node of the list that stores the address of the first node.
2. Create a newnode.
3. Assign the value 'val' at 'newnode->data' and newnode->prev=NULL
4. If the 'front' is NULL assign the newnode to front
5. If not, assign
6. newnode->next=front
   front->prev=newnode
   front=newnode;

C code:

```
void addbeg(struct node **front, int val)
{
             struct node *newnode;
             newnode=(struct node*) malloc(sizeof(struct node));
              newnode->data=val;



if(*front==NULL)
      {
*front = newnode;
            newnode->next=NULL;
            newnode->prev=NULL;
      }
      else
      {     (*front)->prev=newnode;
            newnode->next=*front;
            *front=newnode;
            newnode->prev=NULL;
      }
}
```

Note: In case if it is mentioned to write a function to 'insert' a node, the above function can be written.

If it is specified to insert a node after a node, then it cannot be written.

```
void display(struct node **front)
{
      struct node *cur;
      cur=*front;
```

```
        while(cur!=NULL)
        {
                printf("%d ",cur->data);
                cur=cur->next;
        }
}
```

deletenode()

Function: It frees the node whose value is passed in the parameter. First it searches for the 'node' with the value 'val'. Then makes proper links between previous and next node of the 'current' node thus freeing the current node.

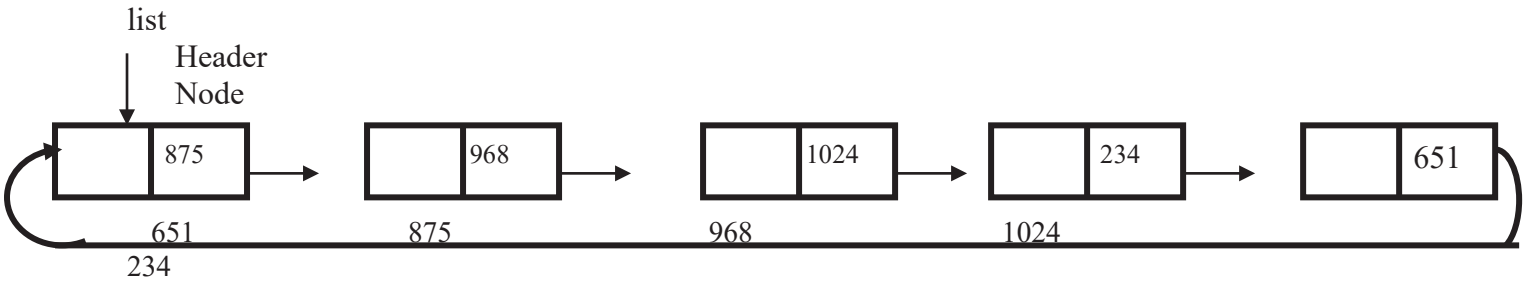
Algorithm:

1. Assign the header node 'front' to 'cur'
2. If the first node itself stores the 'val' then front=front->next
3. If not, search for the node that contains 'val' and store the address in 'cur'
4. Make links between previous and next node of 'cur'
5. De-allocate the memory of 'cur' (free cur)

```
void deletenode(struct node **front,int val)
{
        struct node *curr;
        curr=*front;
        if(*front==curr)
        {
        *front=curr->next;
        }


        else
        {
        while(curr!=NULL&&curr->data!=val)
        {
                curr=curr->next;
        }
        curr->prev->next=curr->next;
        curr->next->prev=curr->prev;
        }
        free(curr);
}

void main()
{
   struct node *front=NULL;
   addbeg(&front,7);
   addbeg(&front,8);
   addbeg(&front,9);
   deletenode(&front,8);
   display(&front)
}
```

output: 9,7

5.3 Circular Linked List



Circular list

The doubly linked list LeftEnd ->left is a pointer to the right most node(i.e .,it equals to the right end) and the RightEnd->right is a pointer to the left most node.

5.3.1 stack using circular list

Function:

1. During initialization 'front' has been set to NULL.
2. If the front is NULL then the stack is considered as empty.
3. 'front' stores the address of the first inserted node.
4. 'front->next' stores the address of the last inserted node.
5. Pop function frees the node that is been stored in front->next
6. During removal front->next and front has same value then it is considered as last node. Hence, the front is set to NULL.

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

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

void push(struct node **front, int val)
{
  struct node *newnode;
  newnode=(struct node*)malloc(sizeof(struct node));
  newnode->data=val;
  if(*front==NULL)
  {   *front=newnode;
       newnode->next=newnode;
  }
  else
  { newnode->next=(*front)->next;
   (*front)->next=newnode;
  }
}

int pop(struct node **front)
{ struct node *cur;
```

```
  int val;
  if(*front==NULL)
  {  printf("Stack is empty");
     return 0;
  }
  else
  {
   cur=(*front)->next;
   if(*front==cur)
   {  *front=NULL;
   }
   (*front)->next=cur->next;
    val=cur->data;
    free(cur);
    return val;

}
}

void main()
{
  struct node *front=NULL;
  push(&front, 7);
  push(&front, 8);
  push(&front, 9);
  push(&front,10);
  printf("%d",pop(&front));

}
```

output: *10*

5.3.2 Circular queues using Linked List

Explanation:

1. Initial value of the header node 'front' is set to NULL
2. If the front is NULL, the queue is considered as empty.
3. 'front' always stores the address of the last node.
4. 'front->next' always stores the address of first inserted node.
5. front->next node is always removed first.
6. If 'front' and 'front->next' is same, then only one node existing in the list.

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

struct node
          7. { int data;
  struct node *next;
};
void insertq(struct node **front, int val)
{
    struct node *newnode;
    newnode=(struct node*)malloc(sizeof(struct node));
    newnode->data=val;
    if(*front ==NULL)
    {  *front=newnode;
     newnode->next=newnode;
```

```
    }
    else
    { newnode->next=(*front)->next;
      (*front)->next=newnode;
      *front=newnode;
    }
}
int removeq(struct node **front)
{  int val;
   struct node *cur;
   if(*front==NULL)
   { printf("Queue is empty");
     return 0;
   }
    else
   {
     cur=(*front)->next;
    (*front)->next=cur->next;
     val=cur->data;
     if(cur==(*front)->next)
     {
       *front=NULL;
     }
     free(cur);
     return val;
    }
}
void main()
{

struct node *front=NULL;
insert(&q,6);
insert(&q,7);
insert(&q,8);
insert(&q,9);
printf("%d",removeq(&q));
}
```

output: *6*

5.4 Array (Cursor) Implementation of Linked list

Many languages such as BASIC and Fortran do not support pointers. If linked list is required and pointers are not available, then the alternative implementation used is *cursor* implementation of linked list. Here, a global array of structure is used to implement. Every node consists of two blocks: 'data' and 'next' where 'data' stores the data and next stores the subscript of its next data member.

1. A global variable 'avail' is used to determine out the availability of nodes.
2. 'avail' is initialized to zero. If avail=0 then the list is empty.
3. If avail= -1 then the list is full.
4. Function init() is used to initialize the 'next' blocks to its successor. The last node's next block is initialized to –1.
5. insert function requires two parameters , a value and position after which the value to be inserted.

6. remove() function removes a value that is been stored after the given position in the parameter.

```
#include<stdio.h>

struct node
{   int data,next;
};
struct node n[10];
int avail=0;
void init()
{    int i;
   for(i=0;i<10;i++)
   {      n[i].next=i+1;
   }
   n[9].next=-1;
}
void insert(int p,int val)
{
    if(avail==-1)
    {
       printf("You can not insert");
    }

else
    { int q;
      q=avail;
      n[q].data=val;
      avail=n[q].next;
      n[q].next=n[p].next;
      n[p].next=q;
    }
}
int remove(int p)
{   int v,q;
   if(p==-1||n[p].next==-1)
   {
     printf("Can not remove");
     exit(0);
   }
   else
   {   q=n[p].next;
       v=n[q].data;
      n[p].next=n[q].next;
      avail=q;
   }
   return v;
}
void main()
{
   int ch,pos,val;
   init();
   do
   {
      printf("Enter your choice");
      scanf("%d", &ch);
```

```
        switch(ch)
        {
       case 1:
       printf("Enter position and value ");
       scanf("%d%d",&pos,&val);
       insert(pos,val);
       break;
       case 2:
       printf("Enter the poistion after which the value to be
removed");
       scanf("%d",&pos);
       printf("The value removed %d",rem(pos));
       break;
        }
    }while(ch==1||ch==2);
    }
```

Comparison of Arrays and Linked List

	Arrays	Linked list
Memory Allocation	Memory is allocated during the compile time. Hence it is called as ***static memory allocation.*** Memory is allocated in ***adjacent memory location***	Memory is allocated during the runtime. Hence it is called as ***dynamic memory allocation***. Memory is allocated in ***random address*** and they are linked with the next pointer.
Flexibility	The initial size of the array is the maximum size of the array. Hence the memory is ***fixed***.	The initial size of the list is zero and a linked list is a dynamic data structure. The number of nodes in a list may vary dramatically as the elements are inserted and removed. The size is ***dynamic***.
Execution	Execution is faster because memory is allocated during the compile time.	Execution is slower because memory is allocated during the runtime.
Insertion & Deletion	***Insertion and deletion*** becomes a tedious process because of shifting of elements.	Insertion and deletion is easier because we need to only change the next pointers.
Size	The total size of the array is the product of its data type and the number of elements in the array.	In linked representation, every node requires two fields: data field and the link field. So, the extra memory is been utilized in linked representation compared to arrays representation.

Note:

This disadvantage of extra memory allocation is easily justifiable when we need to store large data. In formula based representation (or arrays), the memory is allocated during the compile time or run time only once. Hence, either more elements cannot be inserted or memory is been wasted due to excess amount of memory allocation.

Common Errors

Memory access Violation or segment Violation: Certain times a program may crash, because an uninitalised pointer variable points to a random base address. Therefore, while accessing the particular node, since it contains a random base address, it goes on moving unending.

Memory allocation: Whenever a new value is to be stored, a *newnode* must be created using malloc. If not, the old value will be replaced by the previous value.

Pointer declaration: Declaring a pointer to the node does not create a memory to store values rather it creates a memory to store an address of a node. Hence, memory must be created using *malloc.*

Last Node: In a single and doubly linked list, the last node's next block must be initialized to NULL.

Doubly linked list

A doubly linked list consists of three fields: a data field that contains the information and the left that contains the prev nodes' address whereas the right contains the next nodes' address.

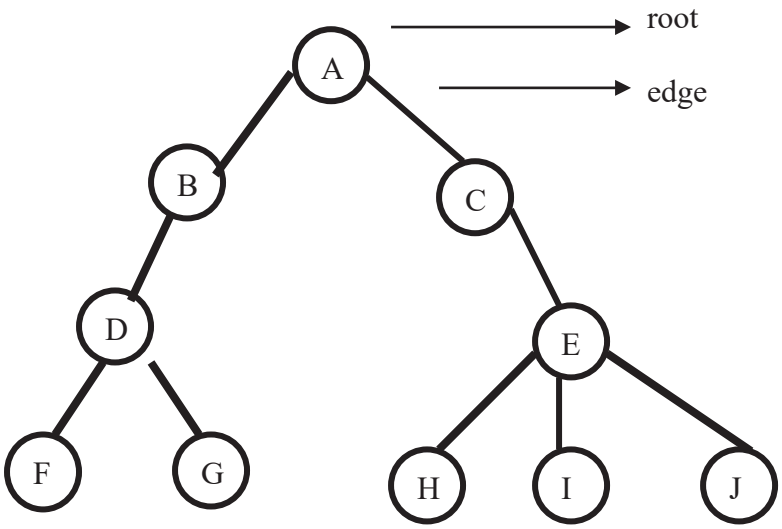
1. In a single linked list, if it is necessary to go behind after searching, it is advisable to store the previous address in a temporary variable.

1. After removing the last node from the list, the front should be made as NULL.
2. Address of a variable can only be stored by the pointer variable of the same data type.
3. Address of a single pointer must be stored in a double pointer of the same type.

6. Trees

Definition: A ***tree*** is a finite non-empty set of nodes where specially designated node is called as the root of the tree and the remaining nodes are partitioned into n disjoint sets $T_1,T_2…..T_n$ and each of them in turn are trees. The trees $T_1,T_2…..T_n$ are the sub-trees of the root.

Terminologies of Tree



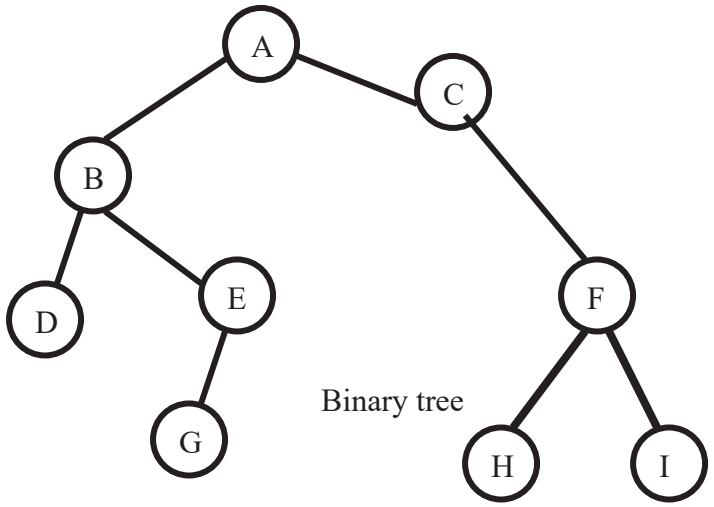
- Each element of a tree is called as ***node*** of the tree.
- The topmost node (A) is called as the *root*.
- Each node may have arbitrary number of nodes (children) connected to it. Every child is a *sub-tree*. ' B' and 'C' are the children of 'A'.
- Each sub-tree is connected by a directed *edge* from root.
- Nodes with same parent are called as *sibling*. 'B' and 'C' are siblings.
- The number of sub-trees connected to a node is called as the *degree*. If a node has 'n' children, the node is at the *degree* 'n'. The degree of 'C' is 1 and the degree of 'E' is 3. Number of edges leaves from a node is known as *outdegree* of the node. Number of edges reaches the node is known as *indegree* of the node.
- 'C' is the *ancestor* of 'E' and 'E' is the *descendent* of 'C'.
- Nodes with no children are called as ***leaves*** or *terminal nodes*. F,G,H,I,J are the leaves.
- Going from the *leaves* to the root is called as *climbing* the tree.
- Going from the *root* to the *leaves* is called as *descending* the tree.
- A *path* from node n_1 to n_k is defined as a sequence of nodes $n_1,n_2,n_3…..n_k$ where n_i is the parent of n_i+1 for $1 \leq i < k$

- The *length* of the path is the number of edges on the path (k-1). There is a path of length zero from every node to itself.
- The *depth* of n_i is the length of the unique path from the root to n_i
- The height of n_i is the longest path from n_i to leaf. Thus all the leaves are at height zero.
- The depth of the tree is equal to the depth of the deepest leaf. This is always equal to the height of the tree.
- Non-leaves are called as *internal* nodes and leaves are called as *external nodes*. If there are 'n' internal nodes, there will be 'n+1' external nodes available in a tree.

6.2 Binary Tree:

Definition: A ***binary tree*** is a tree in which a node can have at most two children(sub-trees). It consists of a root and two ***subtrees*** T_L and T_R, Where T_L and T_R are the ***left*** and ***right subtrees*** of the original tree. The *root*, has no predecessor, and all others have exactly one predecessor, and each node may have 0, 1 or 2 successors. Each successor could be possibly empty or contain another set of trees.

Recursive Definition: A **binary tree** is either empty (represented by a null pointer), or consists of a root and two disjoint binary trees called the left and right sub-tree and each sub-tree is a **binary tree** by itself.



Binary tree

Binary Tree as ADT

```
Abstract typedef NODE(etype, left, right)
Type(left,right)= NODE pointer.

Abstract bool Empty(T):
```
If the tree T is empty return 1 else return 0;

`Abstract Insert(T, V)`
If 'V' is greater that T->data then insert it on the right side else on the left side.

`Abstract Inorder(T)`
Visit all the left subtrees and parent and then the right subtrees.

`Abstract Postorder(T)`
Visit all the left subtrees and right subtrees and then the parent

`Abstract Preorder (T)`
Visit parent, left and then the right subtrees.

`Abstract etype MAX(T)`
Returns the largest element in the tree.

`Abstract etype MIN(T)`
Return the smallest element in the tree.

`Abstract etype Delete(T,V) :`
Deletes 'V' form the tree

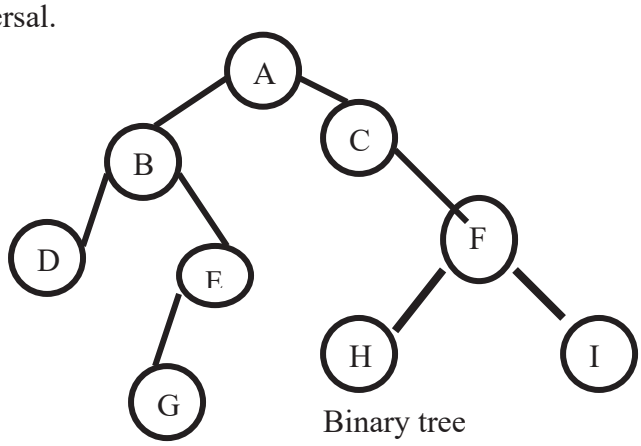
`Abstract copy( T1,T2)`
Copies the elements of T1 into T2.

6.3 Traversal Technique

Traversing is a process of visiting every node in the tree exactly once in some linear sequence.
There are two fundamentally different kinds of binary tree traversal ***depth first*** and ***breadth first*** traversal.
In *depth-first* traversal a node is visited three times once it way down the tree, second time coming up from the left child and third time coming up from the right child. To implement we need *stacks*. There are three *depth-first* traversal.

1.Inorder traversal 2.Preorder Traversal 3.Postorder Traversal

In *breadth-first* traversal, it does not follow the branches of the tree. To implement we need the *queue* data structure. There is only one kind of *breadth-first* traversal ie) ***level order*** traversal.



Binary tree

6.3.1 Inorder Traversal

If a tree "T" is traversed in an "inorder" fashion then "T" is traversed in the following order

1. **L**eft sub-tree is traversed
2. **P**arent node is visited
3. **R**ight sub-tree is visited.

It is also known as ***symmetric*** order traversal.

Inorder Traversal : `DBGEACHFI`

6.3.2 Preorder Traversal

In preorder traversal, the root node is traversed first then the left-sub tree and finally the right sub tree. (work at a node is performed before its children are processed.). The order as follows

1. **P**arent node is traversed
2. **L**eft-sub tree is traversed
3. **R**ight sub-tree is traversed

It is also known as ***depth-first order*** traversal.

Preorder Traversal : `ABDEGCFHI`

6.3.3 Postorder Traversal

In post-order traversal work at a node is performed after its children are evaluated. The left sub-tree is traversed first then the right sub-tree and then the root is visited. The order as follows

1. **L**eft sub-tree is traversed
2. **R**ight sub-tree is traversed
3. **P**arent node is traversed

Postorder Traversal : `DGEBHIFCA`

NOTE*:** If the left and right is interchanged in the above definitions, the result is called as ***Converse of definition. (Ex. converse preorder is parent, right, left).

6.3.4 Level-order Traversal

Data in all the nodes of k^{th} level is printed before k+1 level. Data is printed from left to right.

Levelorder Traversal: `ABCDEFGHI`

6.4 Representation of a binary Tree

1. Linked Array Node Representation
2. Linked Dynamic Node Representation
3. Sequential Array Representation

6.4.1 Linked Array Node Representation

In this representations, a node contains 4 fields, ***data*** field that contains the information, ***left*** stores the subscript of the left child and ***right*** stores the subscript of the right child and ***father*** field stores the subscript of the parent's subscript.

```
struct
{
      int data;
      int left,right, father
}node[100];
```

If a node does not have a left or right child, is indicated through null value –1. To be more efficient, in order to know whether the node is left child or right child, a flag variable is used. If the flag variable is 0 (FALSE) then it is considered as the left child and if 1 (True) then it is considered as right child. Root is indicated through –1.

6.4.2 Linked Dynamic Node Representation

In this representation, a node contains three fields: data field that contains the information and left and right pointer which contains the address of the left and right child node's address respectively. If a node does not have a left child and right child , it is indicated through NULL pointer. In this representation, the memory for a node is allocated during the runtime. Here the father field is not necessary because the tree is always traversed in downward fashion.

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

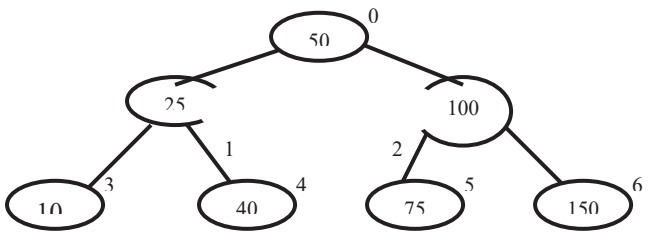
6.4.3 Sequential Representation

An array can be used to represent a binary tree

1. Representing a complete and almost Binary Tree through an array

A complete and almost complete binary tree can be represented through a sequential rray. Since a tree of ***n*** nodes can be numbered through 0 to n-1.

- if p is the position of a node then its left sub tree is at the 2p+1 position and it's parent is at p/2 position
- If p is the position of the node then its right sub tree is at 2p+2 position and its parent is at p/2-1position.

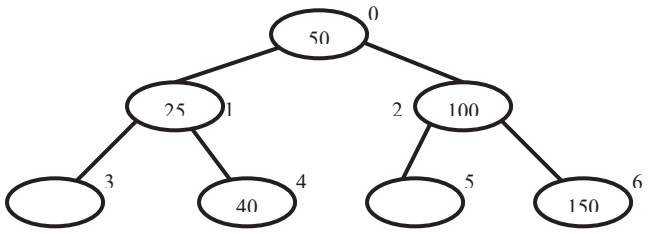


0	1	2	3	4	5	6
50	25	100	10	40	75	150

2. If a binary tree is not a complete binary tree, then arrays of node's have to be created where a node will have two fields: data field and flag field. Data field stores the data and flag field stores whether the selected node is used or unused. If it is used it stores 1 else 0.

```
struct node
{
            int data;
            int used;
};
```

The left and the right sub-tree of a node is recognized through the expression 2p+1 and right sub-tree is recognized through 2p+2. When a value to be inserted, if the node is unused then it is used, if not, the search process begins for an unused node. If value to be inserted is greater, the search is made on the right side and otherwise on the left side.



0	1	2	3	4	5	6
1	1	2	0	1	0	1
50	25	100		40		150

6.5 Binary Search Tree(BST)

A "binary search tree" (BST) or "ordered binary tree" is a type of binary tree where the tree may be empty or the nodes are arranged in order. A non-empty binary search tree satisfies the following conditions:

1. Every element has a key (or value) and no two elements have the same key. Therefore all the keys are distinct.
2. The keys in the left sub-tree are smaller than the key in the root.
3. The keys in the right sub-tree are larger than the key in the root.
4. `The left and right sub-trees are also binary search trees.

Note :

- ***Predecessor*** of the node X is the maximum value in the left sub-tree (left sub-tree rightmost). If a node that does not have a child then its first left ancestor.
- ***Successor*** of X is the minimum value of the right sub-tree (right sub-tree leftmost).

C implementation of Binary Search Tree

```
struct node
{
      int data;
      struct node *left, *right;
```

```
};

void insert(struct node **root, int val)
{
      if(*root==NULL)
      {
            struct node *newnode;
            newnode=(struct node*)malloc(sizeof(struct node));
            newnode->data=val;
            newnode->left=NULL;
            newnode->right=NULL;
            *root=newnode;
      }
      else
      {     if((*root)->data>val)
            insert(&(*root)->left,val);
            else if((*root)->data<val)
            insert(&(*root)->right,val);
            else
            printf("%d is a duplicate",,val);
      }
}

void inorder(struct node *root)   //L-P-R
{
      if(root==NULL)
      return;
      else
      {     inorder(root->left);
            printf("%d\n",root->data);
            inorder(root->right);
      }
}
//Recursive
void preorder(struct node *root)   //P-L-R
{
      if(root==NULL)
      return;
      else
      {     printf("%d\n",root->data);
            preorder(root->left);
            preorder(root->right);
      }
}

//Recursive
void postorder(struct node *root)   //L-R-P
{
      if(root==NULL)
      return;
      else
      {
            postorder(root->left);
            postorder(root->right);
            printf("%d\n",root->data);
      }
}
```

```
int max(struct node *root)
{
      while(root->right!=NULL)
      root=root->right;
      return root->data;
}

int min(struct node *root)
{
      if(root->left==NULL)
      return root->data;
      else
      return min(root->left);
}

int search( struct node *root,int val)
{
      if(root==NULL)
      return 0;
      else
      if(root->data > val)
      return search(root->left,val);
      else
      if(root->data <val)
      return search(root->right,val);
      return 1;
}
```

Deleting a node in a binary Tree.

To delete a node form a binary search tree, we must first locate it. Then the node to be deleted may satisfy one of the following cases.

1. **The node to be deleted has no children:** When the deleted node has no children, all we need to do is set the delete node's parent to null, recycle its memory, and return.

2. **The node to be deleted has only a right subtree:** If there is only a left subtree, then we can simply attach the right subtree to the delete node's parent, and recycle its memory.

3. **The node to be deleted has only a left subtree:** If there is only a left subtree, then we attach the left subtree to the deleted node's parent and recycle its memory.

4. **The node to be deleted has two subtrees:** While it is possible to delete a node from the middle of a tree, the result tends to create very unbalance trees. Rather than simply delete the node, therefore, we try to maintain the existing structure as much as possible by finding data to take the deleted data's place. This can be done in one of two ways: (1) we can find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data or (2) we can find the smallest node on the deleted node's right sub-tree and move its data to replace the deleted nodes data. Regardless of which logic we use. We will be exchanging data with a leaf or a leaf alike node that can then be deleted.

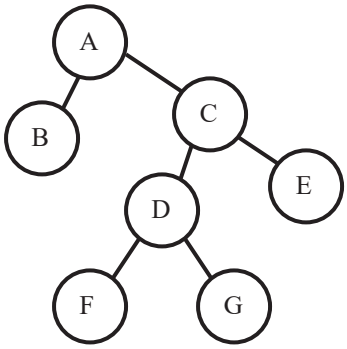
5. **The node to be deleted is linked with the header:** Find the non-empty subtree and place it in the header node.

```
void del(struct node **root, int val)
{
      struct node *temp, *prev, *succ;
      temp=*root;
      if(*root==NULL)
      printf("NO NODES EXISTS");
      else
      {
            while(temp->data!=val && temp!=NULL) //searching
            {
                  prev=temp;
                  if(temp->data > val)
                  temp=temp->left;
                  else
                  temp=temp->right;
            }

            if(temp->left!=NULL && temp->right!=NULL)//4th Possibility
            {

                  succ=temp->right;
                  while(succ->left!=NULL)
                  {
                        prev=succ;
                        succ=succ->left;
                  }
                  temp->data=succ->data;
                  temp=succ;
            }



            if(temp->left!=NULL    &&    temp->right==NULL)    //    3rd
      Possibility
            {

                  if(prev->left==temp)
                  prev->left=temp->left;
                  else
                  prev->right=temp->left;
            }
            if(temp->left==NULL     &&     temp->right!=NULL)     //2nd
      Possibility
            {

                  if(prev->left==temp)
                  prev->left=temp->right;
                  else
                  prev->right=temp->right;
            }
```

```
            if(temp->left==NULL     &&     temp->right==NULL)     //1st
      Possibility
            {
                  if(prev->left==temp)
                  prev->left=NULL;
                  else
                  prev->right=NULL;
            }

            if(temp==*root) //5th Possibility
            {
                  if(temp->left!=NULL)
                  *root=temp->left;
                  else
                  *root=temp->right;
            }
      free(temp);
      }
}
```

6.6 Strictly Binary Tree

If every node in a binary tree has either no sub-trees or two sub-trees then, the tree is termed as *strictly binary* tree.



6.7 Complete Binary Tree

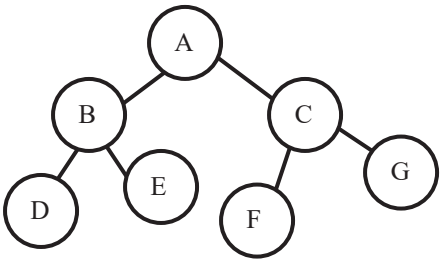
A ***complete binary tree*** is a tree in which

i) It is a strictly binary tree. Hence all the non-leaf nodes will have 2 children.

ii) All the leaves are at the same depth d.

iii) The total number of nodes(t_n) in a complete binary tree at the depth d is equal to the sum of the number of nodes at each level between 0 and d.

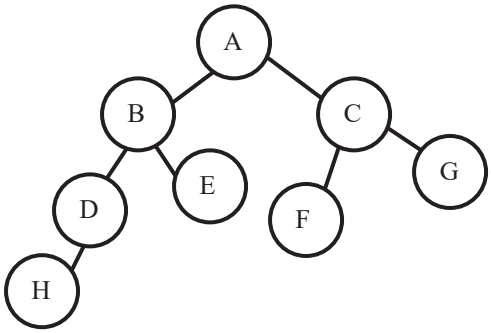
$t_n=2^0+2^1+2^2+\ldots\ldots\ldots\ldots\ldots+2^d=\sum 2^{j-1}$ where $0<=j<=d$



6.8 Almost Complete Binary Tree

A binary tree of depth d is almost complete binary tree iff

1. A node can have either left children alone or it should have right and left together or it may be empty.
2. If nd is the leaf node at the depth d then all the leaves at depth d-1 are at the same level.
3. For every node the height of the left sub-tree may be one greater than the right sub-tree, or it may be equal.



6.9 Difference between binary tree and tree

1. A binary tree can be empty whereas a tree cannot be.
2. Each element in a binary tree can have atmost two sub-trees or it may be empty whereas in tree a node can have any number of sub-trees but it can't be empty.
3. The sub- tree of each element in a binary tree is ordered. That is we distinguish between the left and the right sub-tree. The sub-trees of trees are unordered.

6.10 Application of Binary Search Tree

1. To find the duplicates at the given list of numbers.
   Binary Tree is a useful when we want to determine whether duplicate numbers are in the list. Compare to the linear search, the binary tree search makes less comparison. If the number being searched is greater than the value at the node position then we traverse on the right side , if it is lesser then we search on the left side else we conclude it is a duplicate.

2. ***Sorting:*** Construct a binary search tree whose keys are the unordered elements to be sorted Perform an inorder traversal. We get sorted list.

3. ***Searching:*** It is easy to search for an element from a binary search tree. If the key X is greater than K travel on the right sub-tree else left sub-tree, where X is the element being searched and K the value of the key at the node. The running time for the search operation is O(h) where h is the height of the tree. If the tree is balanced, the height is logn.

Expression Tree

An expression tree is a binary tree in which all the internal nodes contain the operator and the leaf node stores operands. It satisfies the following properties.

1. Each leaf is an operand
2. The root and internal nodes are operators.
3. Sub trees are sub expression with the root being an operator.

The root stores the operator that will be evaluated at last. Reading the tree in inorder strategy will produce infix form , preorder strategy will give prefix form and postorder will give postfix form of the expression.

```
#define OPERATOR 1
#define OPERAND  2

      struct node
      {
            int type;
            union
            {     char op;
                  float num;
            }data;
            struct node *left,*right;
      };
```

Evaluating an expression tree.

In an expression tree, the node may contain operand or operator.

```
float  operation(float op1,flaot op2,  char op)
{
      switch(op)
      {
            case '+': return op1+op2;
            case '-':   return op1-op2;
            case '*': return op1*op2;
            case '/' :return op1/op2;
      }
}


      float evalbintree(struct node *root)
      {
            if(root->type==OPERAND)
            { return root->data.num;
            }
            else
            {
                  float op1,op2;
                  op1=evalbintree(root->left);
```

```
        op2=evalbintree(root->right);
        return operation(op1,op2,root->data.op);
    }
```

Hetrogeneous Tree:

In a binary tree, often, nodes may not contain same type of data. For an example, in a expression tree, the non-leaf nodes contain the operator, a character and leaves contain operand an integer. In this case we use union, in order to represent the data block of the node. This kind of tree is called as ***heterogeneous*** trees.

6.11 Threaded Binary Tree

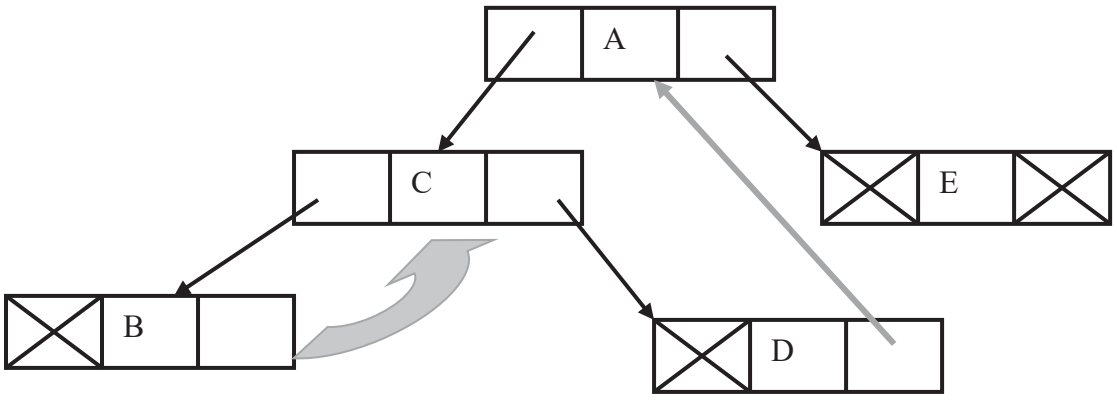
Thread: In a binary tree, an empty node that points to left and/or right sub-trees stores NULL. But in a threaded binary tree, the *leftchild* of the empty node (n) stores the inorder predecessors of 'n' similarly, if the *rightchild* of the node 'n' is empty then it stores the *inorder* successor of 'n' .In a threaded binary tree every node contains two flag variables *rightflag* and *leftflag* respectively. The *rightflag* stores TRUE if the *rightchild* points to its inorder successor else it stores FALSE. Similarly the *leftflag* stores TRUE if the *leftchild* points to its inorder predecessor else it stores FALSE. In this case, In this case there is no longer a need for the stack. Since the last node visited during a traversal of a left subtree points directly to its inorder successor. Such a pointer is called as ***thread***.

Every node in a threaded binary tree must contain the flag variables . Hence a structure of a node in such trees contains two more fields in addition to the three fields "data", left" and " right".

```
struct node
{
   int data;
   int leftflag,rightflag;
   struct node *left, *right;
};
```

Right-in -Threaded binary tree:

The right node of a tree contains its inorder successor instead of NULL and the right most node in the tree has no inorder successor, such trees are called as rightin-threaded binary tree. To implement right – in threaded binary tree in dynamic implementation, we need to have a flag filed that contains true if the node has any inorder successor, else it contains false.



right-in threaded binary tree

Left-in threaded Tree: The left node of a tree contains its inorder predecessor instead of NULL.
In-threaded Binary Tree: If a binary tree in both left-in and right-in threaded then that binary tree is called as In-threaded binary tree.

Advantages of Threaded Binary Tree

Threaded trees are used for efficient inorder traversal. Normally in many applications we may have to traverse the binary tree quite often. In the binary tree approach, when a tree is traversed through recursion, a stack is created. Hence considerable space is utilized by the stack for every call and time is being consumed for every call. Where as in the threaded binary tree, the right node stores the address of its successor, we can traverse the tree using iteration. Hence the drawback of recursion is rectified in threaded binary tree.

6.12 Huffman Tree

A *Huffman tree* is a binary tree that finds the minimum length bit string which can be used to encode a string of symbols. It minimizes the sum of f(i)D(i) over all leaves 'i', where f(i) is the frequency of weight of leaf 'i' and D(i) is the length of the path from the root to leaf 'i'. The Huffman code is very popular in data compression algorithms. For example when a code to be transmitted through networks, the overall length of the transmission is shorter if Huffman code is used.

Properties of Huffman Tree

i) A Huffman tree is a strictly binary tree that is in which all the node are either empty or will have two sub-trees
ii) Smaller frequencies are further away from the root
iii) The two smallest frequencies are siblings.

Huffman Encoding

Huffman's scheme uses a table of frequency of occurrence for each symbol (or character) in the input. This table may be derived from the input itself or from data which is representative of the input. For instance, the frequency of occurrence of letters in normal English might be derived from processing a large number of text documents and then used for encoding all text documents. Once we have established the weight of each character, we build a tree based on those values.

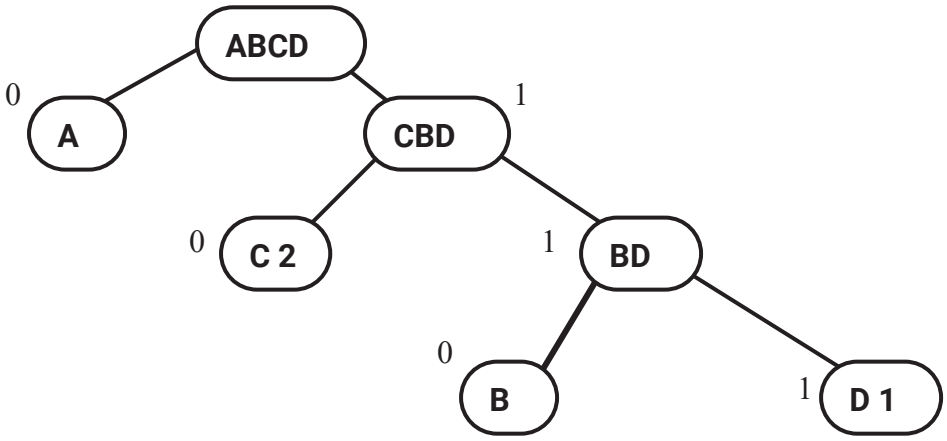
a. First we organize the entire character set into a row, ordered according to frequency from highest to lowest. Each character is now a node at the leaf level of a tree.
b. Next we find the two nodes with the smallest combined frequency weights and join them to form a third node, resulting in a simple two level tree. The weight of the new node is the combined weights of the original two nodes. This node, one level up from the leaves, is eligible to be combined with other nodes. Remember the sum of the weights of the two nodes chosen must be smaller than the combination any other possible choices.
c. We repeat step 2 until all of the nodes, on every level, are combined into a single tree. If none of the higher values are adjacent to the lower value, we can rearrange the nodes for clarity.

Once the tree is complete, we use it to assign codes to each character. First, we assign a

bit value to each branch. Starting from the root, we assign 0 to the left branch and 1 to the

right branch becomes 0 and which becomes 1 is left to the designer—as long as the assignments are consistent throughout the tree

A	B	C	D
3	1	2	1



Decoding Huffman-encoded Data

The decoding procedure is deceptively simple. Starting with the first bit in the stream, one then uses successive bits form the stream to determine whether to go left or right in the decoding tree. When we reach a leaf of the tree, we've decoded a character, so we place that character onto the (uncompressed) output stream. The next bit in the input stream is the first bit of the next character.

Code of A 0 Code of C 10 Code of B 110 Code of D 111

Huffman Algorithm

```
/*   initialize the set of root nodes  */
rootnodes =  the empty ascending priority queue;


/*     construct a node for each symbol    */
for  ( i = 0 ;  i < n; i++)   {
          p  =  maketree(frequency [i]);
          position [i] = p; /* a pointer to the leaf cibtaububg */
                              /* the ith symbol  */
          pqinsert(rootnodes, p);
}  /* end for */

while  (rootnodes contains more than one item) {
     p1 = pqminelete(rootnodes);
     p2 = pqminelete(rootnodes);
     /*  combine p1 and p2 as branches of a single tree */
     p = maketree(info(p1)+info(p2));
     setleft(p, p1);
     setright(p, p2);
     pqinsert(rootnodes, p);
}  /* end while */
```

```
/* the tree is now constructed; use it to find codes */

root = pqmindelete(rootnodes);
for (i = 0; i < n; i++) {
    p = position [i];
    code[i] = the null bit string;
while (p ! = root) {
    /* travel up the tree */
    if (inleft(p))
        code[i] = 0 followed by code[i];
    else
        code[i] = 1 followed by code[i];
    p = father(p) ;
} /* end while*/

} /* end for */
```

Game Tree:

A game tree is a tree that represents the movement of position between two players eg. *Chess, Tic-tac-toe.* Starting at any position, it is possible to construct a tree of the possible position that may result from each move.

Min-max Strategy:

Properties of a Binary Tree

1. **A tree with 'n' nodes will have n-1 edges**

Proof:

When n=1, then the number of edges is 0.

i.e. n-1= 1-1=0.

Hence the base is proved.

When n =n, then

It must have a root and the root has ***k*** children where (k>0).

Since a node can have only one parent therefore k edges have been connected with the root and the total number of nodes is k+1.

When n=k+1

= n-1;

=(k+1)-1;

=k;

Let n_i be the number of nodes connected to the i^{th} child for I=0 to I=k-1 . Hence each child is assumed to be a tree, by induction , the total number of edges with the i^{th} child

$=n_i-1$

Hence total number of edges in the tree of i^{th} child

$= \sum(n_i-1)$ for(I=0 to k-1)

The original tree contains k edges for its k children from the root

Therefore the total number of edges

$= \sum(n_i-1) + k \quad for(I=0 \text{ to } k-1)$
$= \sum(n_i) - k + k \quad for(I=0 \text{ to } k-1)$
$= n-1;$

Thus the said property is proved for any tree.

2. In a tree every node has exactly only one parent except the root.

3. There is exactly only one path connecting any two nodes in a tree.

4. The maximum number of nodes at level I of a binary tree is 2^{i-1}, I>=1

Proof:
If root is the only one node in the tree, then the level is 1 (I=1)

When the level I=1,number of nodes in the tree is 1.

Ie) $1= 2^0=2^{i-1};$
Hence by induction the base is proved.

By assumption, when the level is j where $1<=j<I$, the maximum number of nodes is 2^{j-1}

By induction hypothesis , the maximum number of nodes at level I-1 is
$i-1= 2^{i-2}$
Since each node in a binary tree has maximum degree 2 , the maximum number of nodes at level I must be 2 times the maximum number of nodes at level I-1;
$I=2^{i-2}+2^{i-2}$
$= 2^{i-1}$
Hence proved.

5. Maximum number of nodes in a binary tree of depth k is 2^k-1, k>=1

By the previous proof, the maximum number of nodes at the level I is 2^{i-1}
Maximum number of nodes in a binary tree of depth k is the sum of the maximum number of nodes at every level I where $1<=I<k$
Ie) $=\sum 2^{i-1}$ for I=1 to k
$=2^k-1$

Note: A complete binary tree is a tree where all the nodes are at the same level. A complete binary tree of height h must contain 2^h-1 elements.

6. The minimum height of the binary tree with 'n' nodes is $\log_2(n+1)$

Proof:
Since there must be an element at each level, the height cannot exceed 'n'
Also from the previous proof, the maximum number of nodes at the height h is 2^h-1

$2^h-1 \leq n$
$2^h \leq n+1$
$h \leq \log_2(n+1)$

Thus proved.

7. For any non-empty binary tree T, if n_0 is the number of leaves(terminal nodes) and n_2 is the number of nodes at degree 2, then $n_0=n_2+1$
   Proof:

   Since all the nodes in binary tree(T) are of degree ≤ 2
   Let 'n' be the total number of nodes
   Let n_1 be the number of nodes of degree 1.
   Let n_2 be the number of nodes of degree 2
   And n_0 be the number of leaves
   Hence
   $n= n_{0+} n_{1+} n_2$ ..... (1)

   By the previous proof, if 'n'is the total number of nodes then it has n-1 edges .
   Total number of edges in a tree (T) is the total number of edges with single degree and total number of edges with 2 degrees
   $n-1 = n_{1+} 2n_2$
   $n= n_1+ 2n_2 +1$ ……..(2)
   Form (1) and (2) we get

   $n_1+ 2n_2 +1= n_0+ n_1+ n_2$
   From rearranging the terms, we get
   $n_2 = n_0-1$
   Thus proved

8. A binary tree with 'n' internal nodes will have 'n+1' external nodes

Proof: If the tree is empty ie) n=0 then the tree does not have any internal nodes but it has one external node, thus the base is proved.

Let us assume that a tree has m internal nodes.
Let k be the number of nodes on the left sub-tree
Therefore the right subtree contains m-(k+1) nodes
By induction the number of external nodes in the left sub tree is k+1
The number of external nodes on the right sub-tree is ((m-k-1+1)
ie) m-k
Thus the total number of external nodes of the entire binary tree becomes
=k+1+m-k
=m+1;
Thus proved.

The proof is based on Induction

File System

A file is a collection of data stored in a auxiliary storage device. The file structure allows us to access the information when and where is required.

Files are classfied into two
1. Permanent Files
2. Temporary Files

Permanent Files

7. Analysis of Algorithm

Number of algorithms can be designed to solve a particular problem. The analysis of algorithm provides information that gives us a general idea of how long an algorithm will take for a given problem set. Thus, we select a best algorithm for the given set of inputs. There are many factors that affect the running time of the program. Among these are the algorithm itself, the input data, robustness of a program (how well it deals with the erroneous inputs) and the computer system used to run the program. The performance of a computer is determined by

- The hardware
  - Procoessor used (type and speed)
  - Memory available (Cache and Ram)
  - Disk available
- The programming language in which the algorithm is spcieified
- Operating system
- Robustness of a program :

In the rapid growth of technologies the result of such analysis are not likely to be applicable to the next generation. Hence we use the following two approaches to determine the performance of an algorithm

1. Space Complexity.
2. Time Complexity:

7.1 Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. We are interested in the space complexity of a program for the following reasons.

- If the program is to run on a multi user system, then we may need to specify the amount of memory to be allocated to the program.
- To know in advance whether or not sufficient memory is available to run the program.

7.1.1 Components of space complexity

The space needed by a program has the following components.

i) Instruction space
ii) Data space
iii) Environment stack space

i) Instruction space:

Instruction space is the space required to store the compiled version of the program instruction. It depends on the following factors.

- The computer that compiles the program.
  For example,
  a + b + b * c + (a + b –c) / (c + b) + 4.

All computers perform exactly the same arithmetic operations, but each needs a different amount of space.

- The computer uses options in effect at the time of compilation. Some computers use optimization mode.
  For example, some computers might use the knowledge that
  a+ b + b * c = b * c + (a + b)
  and generate the shorter and more efficient code. This use of the optimization mode will generally increase the time needed to compile the program.
- The configuration of target computer also affects code size. If the computer has floating point hardware, then the floating-point operations will translate into one machine instruction per operation. If this hardware is not installed then code to simulate floating-point computations will be given.

ii) Data space

Data space is the space needed to store all constants and variable values. Data space has three components.

a. Space needed by constants, and simple variables.
b. Space needed by component variable such the array. This category includes space needed by structure and dynamically allocated memory.
c. Space needed for temporary variables generated during the evaluation of expressions.

iii) Environment stack space:

The environment stack is used to save the information needed to resume execution of partially completed functions. Each time a function is invoked the following data are saved on the environment stack.

a. Return address.
b. The value of all local variables and value formal parameters in the function being invoked. The amount of stack space needed by recursive function is called the recursion stack space.
c. The binding of all reference and const reference parameters

7. 2 Time Complexity.

The time complexity of a program is the amount of computer time it needs to run to completion. The time taken by a program is the sum of compilation time and run time. Here we analyse only the run time.

We are interested in analyzing the time complexity for the following reasons

- Some computer systems require the user to provide an upper limit on the amount of time the program will run for.
- The program we are developing might need to provide a satisfactory real time response.
- To find an alternate solution

Requirement of time :

i) The time required to access a variable from the memory and the time required

to store the variable.

$T_{access} + T_{store}$

ii) The time required to perform an elementary operations on integers such as addition (T_+) ,Subtraction(T-) ,multiplication(T*) division(T/), comparison(T>) .

iii) The time required to call a function and the time required to return from the function.

$T_{call} + T_{return}$

iv) The time required to pass an integer argument to a function or procedure is the same as the time required to store an integer in memory.

v) The time required to access an array element

Y=arr[I];

Time required to access the first base address arr, time to access I , and to give the value arr[I].

vi) Time required to allocate memory during the runtime using *malloc()* and time required to de-allocate the memory by *free()*.

We use the following approaches to estimate the time.

7.2.1 Operation Count:

We estimate the time complexity of a program by selecting one or more operations. Such as add, multiply and compare and to determine how many of each is done. The success of this method depends on our ability to identify the operations that contribute most of time complexity.

For example, in sequential search, to search the element at the last location of the list, we make 'n' comparisons. Here comparison is the major operation.

Drawback: Here we focus on certain key operation and we ignore all others.

7.2.2 Step Count:

In step-count method, we attempt to account for the time spent in all steps of the program. For example in the sequential search, we execute 2n+3 statements to find the element at the last location.

```
int sequential(int arr[], int val, int n)
{     int I=0;
     while(I<n&&arr[I]!=val)
     {     I++;
      }
     if(I==n)
     return -1;
     else
     return I;}
```

Drawback: This method overcomes the deficiency of operation count by including all the steps. However the notion itself is inexact because,

1. The magnitude of each statement is not taken into consideration. For example,

x=y and x= a+b*c/d-e are considered as one statement. But both take different amount of time to execute.

2. It is not necessary to know the exact number of steps to determine the run time.

Performance Analysis:

In an algorithm we come across the following order of loops. The order is arranged in increasing order of time requirements.

Function	Name
1	constant
logn	logarithmic
n	linear
nlogn	n log n
n^2	quadratic
n^3	cubic
2^n	exponential
n!	factorial

7.2.3 Asymptotic notations

The asymptotic notation describes the behavior of the time or space complexities for the large instance of characteristics. In the following discussion the function f(n) denotes the time and space complexities of a program measured as a function of the instance characteristic n.

7.2.3.1 Big Oh Notation (O)

The big O notation provides the upper bound for the function *f*.

Definition [Big oh] f(n)=O(g(n)) (read as "f of n is big oh of g of n") if positive constants c and n_0 exist such that $f(n) \leq cg(n)$ for all n, $n \geq n_0$.

The definition states that the function f is at most c times the function g except possibly when n is smaller than n_0. Here c is some positive constant.

Linear Function:

Consider $f(n) = 3n+2$.

When n is at least 2,

$$=3n+2$$
$$\leq 3n+n$$
$$\leq 4n.$$

So $f(n) = O(n)$.

Thus f(n) is bound from above by a linear function.

Quadratic Function:

Suppose that $f(n) = 10n^2 + 4n + 2$.

We see that for $n \geq 2$,

$f(n) \leq 10n^2 + 5n$. Now we note that for $n \geq 5$, $5n \leq n^2$.

Hence for $n \geq n_0 = 5$,

$f(n) \leq 10n^2 + n^2$

$\leq 11n^2$.

Therefore, $f(n) = O(n^2)$.

Exponential Function

$f(n) = 6 * 2^n + n^2$.

Observe that for $n \geq 4$, $n^2 \leq 2^n$.

$f(n) \leq 6 * 2^n + 2^n$

$= 7 * 2^n$ for $n \geq 4$.

Therefore, $= O(2^n)$.

Constant Function:

When f(n) is a constant, as in f(n) = 9or f(n) = 2033, we write f(n)=O(1).

7.2.3.2 Omega Notation (Ω)

The omega notation, which is the lower bound analog of the big oh notation, permits us to bound the value of *f* from below.

Definition [*Omega*] f(n)= Ω(g(n)), (read as "f of n is omega of g of n") if positive constants c and n_0 exist such that $f(n) \geq cg(n)$ for all n, $n \geq n_0$.

When we write f(n)= Ω(g(n)), we are saying that *f* is at least c times the function *g* except possible when n is smaller than n_0. Here c is some positive constant.

Example:

$f(n) = 3n + 2 > 3n$ for all n.

$f(n) > 3n$,

f(n)= Ω(n).

7.2.3.3 Theta Notation (Θ)

The theta notation is used when the function *f* can be bounded both from above and below by the same function *g*.

Definition [theta] *f*(n) = Θ (g(n)) (read as " f of n is theta of g of n") iff positive constants c_1 and c_2 and an n) exist such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$

When we write *f*(n) = Θ (g(n)), we are saying that *f* lies between c_1 times the function *g* and c_2 times the function *g* except possibly when n is smaller than n_0 > here c1 and c2 are positive constants. Thus *g* is in both lower and upper bound.

Example: From example

$3n + 2 = \Theta(n)$;

$3n + 3 = \Theta(n)$;

$10n^2 + 4n + 2 = \Theta(n^2)$;

$1000n^2 + 100n - 6 = \Theta(n^2)$;

$6 * 2^n + n^2 = \Theta(2^n)$.

$10 * \log_2 n + 4 = \Theta(\log_2 n)$

7.2.3.4 Little Oh (O)

Definition [Little oh] f(n) = o(g(n)) (read as " f of n is little oh of g of n") iff f(n) = O(g(n)) and f(n) ≠ Ω (g(n)).

Example: $3n + 2 = o(n^2)$ as $3n + 2 = O(n^2)$ and $3n + 2 \neq \Omega(n^2)$ however, $3n + 2 \neq o(n)$. Similarly, $10n^2 + 4n+2 = o(n^3)$, but is not $o(n^2)$.

7.2.3 Recurrence Relations:

Recursion is a process where a construct operates on itself. Recursive functions are function, where a function calls itself. In other words, we solve a problem in terms of itself.

Recurrence relations can be directly derived from a recursive algorithm, but we can not determine how efficient the algorithm is. Hence we need to establish a closed form by removing the recursive nature. This is done by series of repeated substitutions until we can see the pattern that develops.

7.2.4 Cases to Consider

When analyzing an algorithm we come across three cases.

1. Best case.
2. Worst case.
3. Average case.

1. ***Best case:*** The best case for an algorithm is the input that requires the algorithm to take the shortest time. If we are looking at the searching algorithm, the best-case would be if the values we are searching for was the value stored at the first location produced by the algorithm.
2. ***Worst case:*** Worst-case analysis requires that we identify the input values that cause an algorithm to do the most work. For searching algorithms the worst case is the one where the value is found at the last time. The worst case for sequential searching is N.
3. ***Average case***: The average case analysis process is divided into 3 steps.
    1. Determining the number of different groups into which all the input set can be divided.
    2. Determine the probability that the input will come from each of these groups.
    3. Determine how long the algorithm will run for each of these.

    When all of this has been done, the average case time is given by the following formula.

$$A(n) = \sum_{i=1}^{m} p_i * t_i$$

where n is the size of the input, m is the number of groups, p_i is the probability that the input will be from group i, t_i is the time that the algorithm takes for the input from group i.

In case if all the input takes different probabilities, we calculate the average case by the given formula.

$$A(n) = 1/n \sum_{i=1}^{n} t_i$$

Mathematical Review:

exponents

$$X^A X^B = X^{A+B}$$

$$\frac{X^A}{X^B} = X^{A-B}$$

$$(X^A)^B = X^{AB}$$

$$X^N + X^N = 2X^N$$

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

Logarithms

$X^A = B$ if and only if $\log_x B = A$

$$\log_a B = \frac{\log_c B}{\log_c A}; \quad C > 0$$

$$\text{LogAB} = \log A + \log B$$

Series

$$\sum_{i=0}^{N} 2i = 2^{N+1} - 1$$

$$\sum_{i=0}^{N} Ai = \frac{A^{N+1} - 1}{A - 1}$$

In the latter formula, if $0 < A < 1$, then

$$\sum_{i=0}^{N} Ai \leq \frac{A^{N+1} - 1}{A - 1}$$

and as N tends to ∞, the sum approaches 1/(1-A). These are the “Geometric series” formulas.

Another type of common series in analysis is the arithmetic series. Any such series can be evaluated from the basic formula.

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

$$\sum_{i=0}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$$

$$\sum_{i=0}^{N} i^3 = \frac{N^2(N+1)^2}{4} \approx \frac{N^4}{4}$$

$$\sum_{i=0}^{N} i = \frac{N^{k+1}}{|k+1|} \quad k \neq -1$$

$$H_N = \sum_{i=0}^{N} \frac{1}{i} \neq \log_e N$$

$$\sum_{i=0}^{N} f(n) = Nf(N)$$

$$\sum_{i=n_0}^{N} f(i) = \sum_{i=1}^{N} f(i) - \sum_{i=1}^{n_0-1} f(i)$$

Exercises:

1. Show that the following equalities are correct, using the definitions of O, Ω, Θ, and o only.
   a. $5n^2 - 6n = \Theta(n^2)$
   b. $n! = O(n^n)$
   c. $2n^2 2^n + n\log n = \Theta(n^2 2^n)$
   d. $\sum_{i=0}^{n} i^2 = \Theta(n^3)$.
   e. $\sum_{i=0}^{n} i^3 = \Theta(n^4)$.

2.Show that the following equalities are incorrect:	3. Put the following recurrence relations into closed form.
a. $10n^2 + 9 = O(n)$. b. $n^2\log n = \Theta(n^2)$. **c.** $n^2 / \log n = \Theta(n^2)$. d. $n^3 2^n + 6n^2 3^n = O(n^3 2^n)$.	$T(n) = 3T(n-1) - 15$ $T(1) = 8$

8. Searching

Searching is a process by which we search for a key in the given list. If the key is found in the list, then the search is successful else it is unsuccessful. The following methods are used to search

1. Sequential search
2. Binary search.
3. Hashing
4. Tree search

8.1 Sequential search:

Sequential search compares the elements to be searched one at a time from the first element. It terminates in two conditions

1. When it finds the element
2. if the list ends.

If the search terminates because of the first condition, it is a successful search or its called as unsuccessful search.
The sequential search is used whenever the list is not ordered. Generally, you will use the technique only for small lists or lists that are not searched often

Algorithm :

```
            seqSearch      (List, n, key, index)
Locate the target in an unordered list of size elements.

      Pre   List must contain at least one element
            n is number of  element in the list
            key  contains the data to be located
            index  will store the index of the key

      Post    if found matching index stored in 'index' & return  true
              if not found - last stored in 'index' & return  false

1   looker = 1
2   loop (looker <  last AND target not equal list [looker])
      1    looker  =  looker  + 1
3  index  =  looker
4  if  ( target equal list [ looker])
      return  true
5    else
      return false
```

Program

```
#include <stdio.h>
#define size 25

int  sequential(int a[],int val,int n, int *index)
{
int i=0;
while(i<n&&a[i]!=val)
```

```
{    i++;
}

if(a[i]==val)
{    *index=i;
     return 1;
}
else
{  *index=NULL;
   return 0;
}}

void main()
{
int a[size],n,ch,val,i,p;
printf("\nEnter the no of elements:");
scanf("%d",&n);
printf("\nEnter the elements:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
printf("\nEnter the value:");
scanf("%d",&val);
sequential(a,val,n,&p);
printf("The element is found at %d position",p);

}
```

Analysis of sequential search.

Best Case : If the element to be searched is at the first position then the number of comparisons made to search the target key is 1.

F(n) =1;

Worst Case: If the element is there at the last position of the list, then all the elements need to be compared till the 'n' th element. Hence worst case is 'n'

F(n)=n;

Average case : To find the average comparison done for a successful search in sequential search, we would simply add all the number of comparisons and divide it by n, where n is the number of items in the list.

ie) f(n) = (1+2+3+……n)/n;
= n(n+1)/2 n;
= (n+1)/2.

Note: For an unsuccessful search, the total number of comparison is 'n', because all the elements are compared.

8.1.1 Variation on sequential searches

There are four useful variations on the sequential search algorithm:

(1) The sentinel search.
(2) The probability search
(3) The ordered list search.
(4) Indexed sequential search

8.1.1.1 Sentinel search

This is done by adding the key to be searched at the end of the array . Its known as sentinel entry. The search is terminated when the key is encountered in the list. If the key is found before the end of the list, then the search is said to be successful. If it is found at the end of the list, then it is said to be unsuccessful.

The use of placing the sentinel is that we don't need to make two comparison in the loop. We remove the condition of comparing the index with the end because the key is available in the List.

Function for sentinel search:

```
int  sentinel(int a[],int val,int n, int *index)
{
int i=0;
a[n]=val; /*sentinel value is placed*/

while(a[i]!=val)
{
    i++;
}
if(i<n)
{
    *index=i;
    return 1;
}
else
{  *index=NULL;
   return 0;
}
}
```

8.1.1.2 Probability Search

In probability search the most probable element is brought at the beginning. When a key is found, it is swapped with the previous key . Thus if the key is accessed very often, it is brought at the beginning. Thus the most probable key is brought at the beginning.

```
int  probability(int a[],int val,int n, int *index)
{
int i=0;
while(a[i]!=val&&i<n)
{
    i++;
}


if(a[i]==val)
{   if(i!=0)
    {  int temp;
       temp=a[i];
       a[i]=a[i-1];
```

```
        a[i-1]=temp;
     }
     *index=i;
     return 1;
}
else
{ *index=NULL;
   return 0;
}
}
```

8.1.1.3 Linear Order Search.

When the array is sorted, to search an element, its compared with the elements of the array . When we encounter an element whose value is greater or equal to the element to be searched, then the search is terminated. If we terminate the procedure due to the equal value to the key then, the search is said to be successful, else unsuccessful.

```
int linearorder(int a[],int val,int n, int *index)
{                   /*The array must be sorted*/
int i=0;
while(a[i]<val&&i<n)
{
    i++;
}
if(a[i]==val)
{   *index=i;
    return 1;
}
else
{ *index=NULL;
   return 0;
}
}
```

8.1.1.4 Indexed Sequential Search

In indexed sequential search, there is an auxiliary table (key table) is created along with the array. The key table stores the key and pointer to the key with constant distance. Once a key is to be searched, the key is compared with the key table. Through the key table, we find the possible upper limit and the lower limit of the key in the original array. Then a sequential search is made on the array from the upper limit to the lower limits. If the element is find found between the range, then it is said the search is successful else unsuccessful.

```
int indexed(int arr[],int key[][2],int n,int nk,int val,int *index)
{  int i,begin,end;
   for(i=0;i<nk&&key[i][0]<val;i++);   /* Finding the range where the key lies*/
   begin=key[i-1][1];
   end=key[i][1];
   for(i=begin;i<end&&arr[i]!=val&&i<n;i++); /* Making a sequential search from the
                                                beginning range till the end*/
   if(arr[i]==val)
   {
     *index=i;
     return 1;
   }
```

```
    else
    {
      *index=NULL;
      return 0;
    }
}

void main()
{
int a[size],n,ch,val,i,j,p,key[10][2];
printf("\nEnter the no of elements:");
scanf("%d",&n);
printf("\nEnter the elements:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
for(j=0,i=0;i<n;i+=3,j++)
{
  key[j][0]=a[i];
  key[j][1]=i;
}
printf("\nEnter the value:");
scanf("%d",&val);
indexed(a,key,n,j,val,&p);
printf("The element is found at %d position",p);
}
```

Advantages:

1. Searching is performed in smaller index
2. In Linked list, it reduces the overhead associated with accessing of all the nodes from the memory.
3. Deletion from a indexed sequential table can be made more easily, by introducing flag variable.

Disadvantages:

1. Insertion into a indexed sequential table is more difficult, since there may not be a room between two already existing table entries, thus necessitating a shift in a larger number of tables.

Caution: Since we store the key to be searched at the end of the array, we need to make sure that it is not accessed in further use.

8.2 Binary Search:

Binary search is a searching method in which the target key is always compared with the middle element of the remaining list. If the middle element is not the target key then it is compared with the target key. If the target key is higher than the middle element, then the target key is searched from next to the middle element till the end. If it is less, then it is searched from the beginning till the previous element of the middle element. This procedure is continued. At each iteration, we shall reduce the size of the part of this list by half. The target key which is been searched will be found between low and high.

- This is a very efficient method of searching, because it eliminates half of the element for the further consideration. Because of the basic principles of elimination, the method is also known as ***dichotomous search***.

- This method can only be used with the sorted list.
- Binary search is not good for linked lists, because it requires jumping back forth from one end of the list to the middle. This action is easy within an array but slow for a linked list. Hence in general, we use binary search in contiguous memory location.

Algorithm:

```
BinarySearch ( list, low, high, key )
list        the elements to  be searched
key         the value being searched for
low         the lower range of the array
high        the  higher range of  the array

1. Loop(low ≤ high)
      1. mid = (low + high)/ 2
      2. if(list[mid]=val)
            1. return true
      3.if(list[mid]>val)
            1. high=mid-1
      4.else
            1.low=mid+1
2. return false
```

Non-recursive binary search Program

```
int  bsearch(int arr[],int low,int high, int val)
{           while(low<=high)
            {
            int mid=(low+high)/2;
            if(arr[mid]==val)
            return 1;
            else  if(arr[mid]>val)
                  high=mid-1;
            else
                  low=mid+1;
            }
            return 0;
  }
}
```

Recursive Binary search Program:

```
int  bsearch(int arr[],int low,int high, int val)
{
     if(low>high)
     return 0;
      else
     {
            int mid=(low+high)/2;
            if(arr[mid]==val)
            return 1;
            else  if(arr[mid]>val)
            return bsearch(arr,low,mid-1);
            else
            return bsearch(arr,mid+1,high);
     }
}
```

Binary search analysis:

Best Case:

If the key to be searched remains at the middle, then the number of search is 1.

Worst case:

In this algorithm, every iteration we make search on only half of the range from the previous iteration. Hence the searching limit is reduced half, every time, we conclude that the loop is logarithmic.

$f(n)= O(\log_2 (n))$

Average Case:

There are N possible locations for the target. Hence the probability of an element is 1 / N. If we consider the binary tree that represents this search process, One comparisons is made to find an element at the root and the number of node at the root is also 1.

Two comparisons are done to find the element that are in the nodes on level 2,. The number of nodes at level two is 2.

Three comparisons are done to find the elements that are in the node on level 3. Number of node at level 3 is 4.

In general i comparison are done to find the elements that are the nodes on level i. Number of nodes at i^{th} level are 2^{i-1} nodes on level

The total number of comparison done for every possible case by adding, for every level, the product of the number of nodes on each level and the number of comparisons for that level. This gives an average case of analysis.

If k is the level of the tree then,

$$A(N)= \frac{1}{N} \Sigma i*2^{i-1}$$

$$A(N) = \frac{1}{N} * \frac{1}{2} * [(k – 1)2^{k+1} + 2]$$

$$A(N) = \frac{1}{N} [(k – 1) 2^k + 1]$$

$$A(N) = [\frac{k2^k – N}{N}]$$

$$A(N) = \frac{K2^k}{N} – 1$$

$$A (N) \approx k –1$$

$$A (N) \approx \log (N + 1) – 1$$

$$A(N) \approx O(\log(N))$$

8.2.1 Variations on Binary search

8.2.1.1 Interpolation Search

Another technique for searching an ordered array is called **interpolation search.** If the keys are uniformly distributed between k(0) and k(n - 1), the method may be even more efficient than binary search.

```
mid = low + (high – low ) * ((key – k(low))/ (k(high) – k(low)))
```

If key is lower than k(mid), reset high to mid – 1; if higher, reset low to mid + 1. Repeat the process until the key has been found or low > high.

Analysis on interpolation search:

Average Case:
Interpolation search requires an average of $\log_2(\log_2 n)$ comparison and rarely requires much more, compared with binary search's $\log_2 n$.

Worst Case:
If the keys are not uniformly distributed, interpolation search can have very poor average

behavior. In the worst case, the value of mid can consistently equal low + 1 or high – 1, in

which case interpolation search degenerates into sequential search.

Note: Even though the average case of interpolation search is superior than the binary search, the worst case become inconsistent because it may become a linear search where as binary search remains consistent that is O(log N)

8.2.1.2 Robust interpolation search

It attempts to remedy the poor practical behavior of interpolation search while non-uniform key distributions. This is done by establishing a value *gap* so that *mid-low* and *high –mid* are always greater than *gap*.

Initially, gap is set to *sqrt(high – low + 1).*
Probe is set to *low + (high – low) * ((key – k (low))/(k(high) – k(low))),*
mid is set equal to *min(high – gap, max(probe, low + gap)).*

The expected number of comparison for robust interpolation search for a random distribution of keys is O(log log n). this is superior to binary search.

8.3 Hashing

Hashing is a method that uses a hash function *f* to associate the key *k* of a record *r with* a pointer through a table, called hash table. If *r* is the record and *k* is the key then *r* is stored at *f(k)* . If the element is found then the search is successful else the new record is inserted at the *f(k)* position.

Eg. If the key is 7253, the record can be stored at 7253%10 th location (3)

8.3.1 Hash Functions:

8.3.1.1 Direct method or Linear addressing function

In **direct hashing**, the key is the address without any algorithmic manipulation. The data structure must therefore contain an element for every possible key.

(Eg). Imagine that a small organization has fewer than 100 employees. Each employee is assigned an employee number between 1 and 100. In this case, if we create an array of 100 employee records, the employee number can be directly used as the address of any individual record.

8.3.1.2 Subtraction method

Some time we have keys that are consecutive but do not start from one. For example, a company may have only 100 employees, but the employee numbers start from 1000 and go to 1100. in this case, we use a very simple hashing function that subtracts 1000 from the key to determine the address. The beauty of this example is that it is simple and that it also guarantees that there will be no collisions. Its limitations are the same as direct hashing: it can only be used for small lists in which the keys map to a densely filled list.

8.3.1.3 Modulo-division method

Also known as **division-remainder**, the **modulo-division** method divides the key by the array size and uses the remainder as the address. This gives us the simple hashing algorithm shown below when list size is the number of elements in the array.

Address = key MODULUS Size

How to choose the 'size'

- *If Size is the power of 2 ie) Size=2^K then key %2^k extracts only bottom k bits of binary representation of x. Here many numbers will have bottom k bits same. Hence probability for collision is more.*
- *Where as if Size is the prime number then the division does not depend on the bottom k bits , it depends on all the bits. Hence the probability of two keys having same pointer is less. Therefore all the keys will be spread out.*

8.3.1.5 Midsquare Method

In **midsquare hashing,** the key is squared and the address selected from the middle of the squared number. First assume that M is the power of 2, and W is the wordsize of the computer in bits (16, 32). Shift the x^2 to the right by w-k bits. Here we extract k bits from

the middle of the square of the key. By definition the right shift inserts zeros on the left . Hence the result always falls between 0 to M-1.

Advantage over Division Method: Since integer division is usually slower than the multiplication, by using this method we can potentially increase the run time.

Multiplication Method;

Multiply the key with the carefully chosen constant and then extract the middle k bits.
We should choose a constant that has neither a trailing zeros nor trailing zeros.

8.3.1.4 Digit-Extraction method

Using **digit extraction,** selected digits are extracted from the key and used as the address.

379452	⟹	394
121267	⟹	112
378845	⟹	388
160252	⟹	102
045128	⟹	051

8.3.1.6 Folding Methods

There are two **folding methods** that are used, fold shift and fold boundary.
In **fold shift,** the key value is divided into parts whose size matches the size of the required address. Then the left and right parts are shifted and added with the middle part. For example, imagine we want to map social security numbers into three- digit addresses. We divide the nine-digit social security number into three-digit numbers, which are then added.

In **fold boundary**, the left and right numbers are folded on a fixed boundary between them and the center number.

123	321
456	456
789	987
1368	**1**764

Here 1 is discarded.

8.3.1.7 Length Dependent Method:

In this method the length of the key is used along with some portion of the key. For example if the key is 12345, then 123 is taken from the key and added with the length of the key ie) 5. Thus , the pointer id 128

8.3.1.8 Pseudorandom Method:

A random number is generated using the key. Using the modulo-division method a pointer is generated to the key. The common formula we use to generate the pointer is

P=(a* key +c)%Tablesize
Where a is a coefficient and c is a constant.

Characterestics of a good hash functions;

1. Avoids collisions
2. Tends to spread the key evenly in the array
3. Easy to compute

8.3.2 Collision:

A **collision** is the event that occurs when a hashing algorithm produces an address for an insertion key and that address is already occupied. The address produced by the hashing algorithm is known as the home address. The memory that contains all of the home addresses is known as the **prime area.** When two keys collide at a home address, we must resolve the collision by placing one of the keys and its data in another location. Each calculation of an address and test for successive place, is known as a **probe**.

8.3.3 Clustering:

Some hashing algorithms tend to cause data to group within the list. This tendency of data to build up unevenly across a hashed list is known as **clustering**.

Two distinct types of clusters have been identified by computer scientists.

The first, **primary clustering** occurs when data become clustered around a home address. Two keys that hash into different values compete with each other in successive rehashes, is called ***primary clustering***. For example, when we insert 94 with the hash key function n%10, then it will be placed on 4^{th} location. If we insert 114, 124,134 successively then the location of 94 is examined. Primary clustering is easy to identity.
The primary clustering can be alleviated through the random probing and quadratic probing rehashing and offset key method.

Secondary clustering occurs when data become grouped along a collision path throughout a list. It is not as easy to identify. In secondary clustering, the data are widely distributed across the whole list so that the list appears to be well distributed. If the data all lie along a well-traveled collision path, however, the time to locate a requested element of data can become large.
The secondary clustering can be alleviated through a second hashing function which is independent of the first. This variation of open addressing is known as double hashing.

8.3.4 Collision Resolution Techniques:

The collision resolution technique is an attempt to place the record elsewhere in the table when collision occurs. There are two broad techniques are used.

1. Open Addressing
2. Chaining
3. Bucket Hashing
4. Rehashing
5. Dynamic Hashing
6. Extendible Hashing

8.3.4.1 Open Addressing

The first collision resolution method, **open addressing,** resolves collisions in the home area. When a collision occurs, from the homes address an unoccupied location is searched, where the new data can be placed. We discuss four different methods

1. linear probe
2. quadratic probe
3. double hashing
4. key offset.

8.3.4.1.1 Linear probe

In a **linear probe,** when data cannot be stored in the home address, we resolve the collision by adding one to the current address. The expected number of probes using linear probing for insertion and unsuccessful search is roughly

$$\frac{1}{2}(1+1/(1-\lambda)^2)$$

For a successful search is $\frac{1}{2}(1+1/(1-\lambda))$

Where λ is known as load factor.

Advantage:

1. They are quite simple to implement.
2. Data tend to remain near the home address.

Disadvantage:

1. It tend to produce primary clustering
2. It complicates the search algorithm when the data is deleted.

Note: The **load factor** of a hashed list is the number of elements in the list divided by the number of physical elements allocated for the list expressed as a percentage.

$\lambda = k/n*100$; where n is the table size and k is the number of elements in the table.

8.3.4.1.2 Quadratic probe

In the quadratic probe, the increment is the collision probe number squared. Thus, for the first probe, we add 1^2; for the second collision probe, we add 2^2; for the third collision probe we add 3^2; and so forth until we either find an empty element or we exhaust the possible elements.

address= h(key)+ square(j) where j is the number of time h(key) is been accessed for the particular address.

Disadvantage:

1. A potential disadvantage of the quadratics probe is the time required to square the probe number.
2. It is not possible to generate a new address for every element in the list. Only 42 probe will generate unique address. The solution to this problem is to use a prime number as a list size. If so at least half of the list is reachable.

8.3.4.1.3 Double Hashing

The last open addressing methods are collectively known as double hashing.

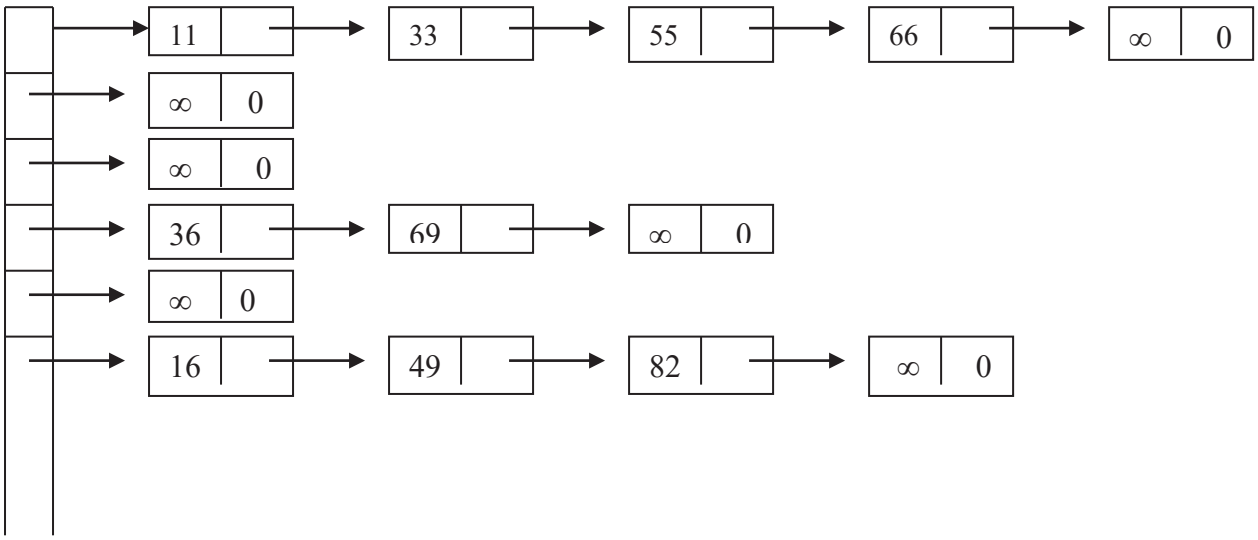
8.3.4.1.4 Key offset

Key offset is a double hashing method that produces different collision paths for different keys. Whereas the Pseudorandom number generator produces a new address as a function of the previous address, key offset calculates the new address as a function of the old address and the key. One of the simplest version adds the quotient of the key divided by the list size to the address to determine the next collision resolution.

```
Offset= key/tablesize;
address= (offset+old address)%tablesize +1
```

8.3.4.2. Chaining

Chaining resolution techniques are achieved through linked lists. There are 'n' number of header pointers are created, where 'n' is the maximum number of pointers can be produced by the hash function. Every key from the same pointers are linked through the next block of the node. The collision occurs when one element is stored in the prime area. While the overflow area can be any data structure it is typically implemented a
s a linked list in dynamic memory.



∞ denotes large key

The number of comparison for a successful search is

$$S(n)= 1+ \lambda /2$$

For an unsuccessful search or for insertion

$$U(n)= (1+ \lambda)/2$$ where λ is load factor.

8.3.4 .4 Bucket Hashing

Another approach to handling the problem of collision is to hash to **buckets**, nodes that accommodate multiple data occurrences. Because a bucket can hold multiple pieces of data, collisions are postponed until the bucket is full. Let us consider that the bucket can hold data about three employees. Under this assumption, there would not be a collision until we tried to add a fourth employee to an address. There are two problems with this concept.
First, it uses significantly more space because many of the buckets will be empty or partially empty at any given time.

Second it does not completely resolve the collision problem. At some point, a collision will occur and need to be resolved. When it does, a typical approach is to use a linear probe, assuming that the next element will have some empty space.

8.3.4.5 Rehashing:

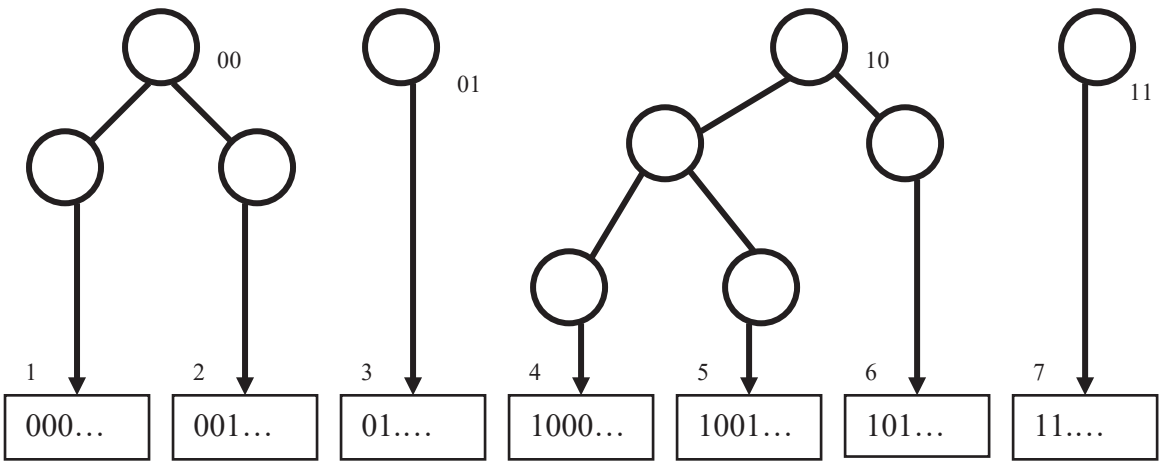
If the table gets too full, the running time for the operations will start taking too long and Inserts might fail for open addressing hashing with quadratic resolution. This can happen if there are too many removals intermixed with insertions. A solution then, is to build another table that is about twice as big (with an association new hash function) and scan down the entire original hash table, computing the new hash value for each (non-deleted) element and inserting it in the new table.

This entire operation is called rehashing. This is obviously a very expensive operation; the running time is O(N), since there are N elements to rehash and the table size is roughly 2N, but it is actually not all that bad, because it happens very infrequently.

Rehashing frees the programmer from worrying about the table size and is important because hash tables cannot be made arbitrarily large in complex programs.

8.3.5 Dynamic hashing

Initially 'm' buckets are created and a hash table of size m is allocated where buckets stores the different combination of binary. The hash functions produces one of the combination. There the Key is to be inserted in the bucket, if the bucket is empty, if not the bucket is splitted . Depends on the i + 1(I is the pointer given by the hash function) the binary number of the key, it is inserted either the left side of the root or the right of the root. If 0 then inserted on the left sub-tree, else on the right sub-tree.



A dynamic hashing configuration

8.3.6 Extendible hashing

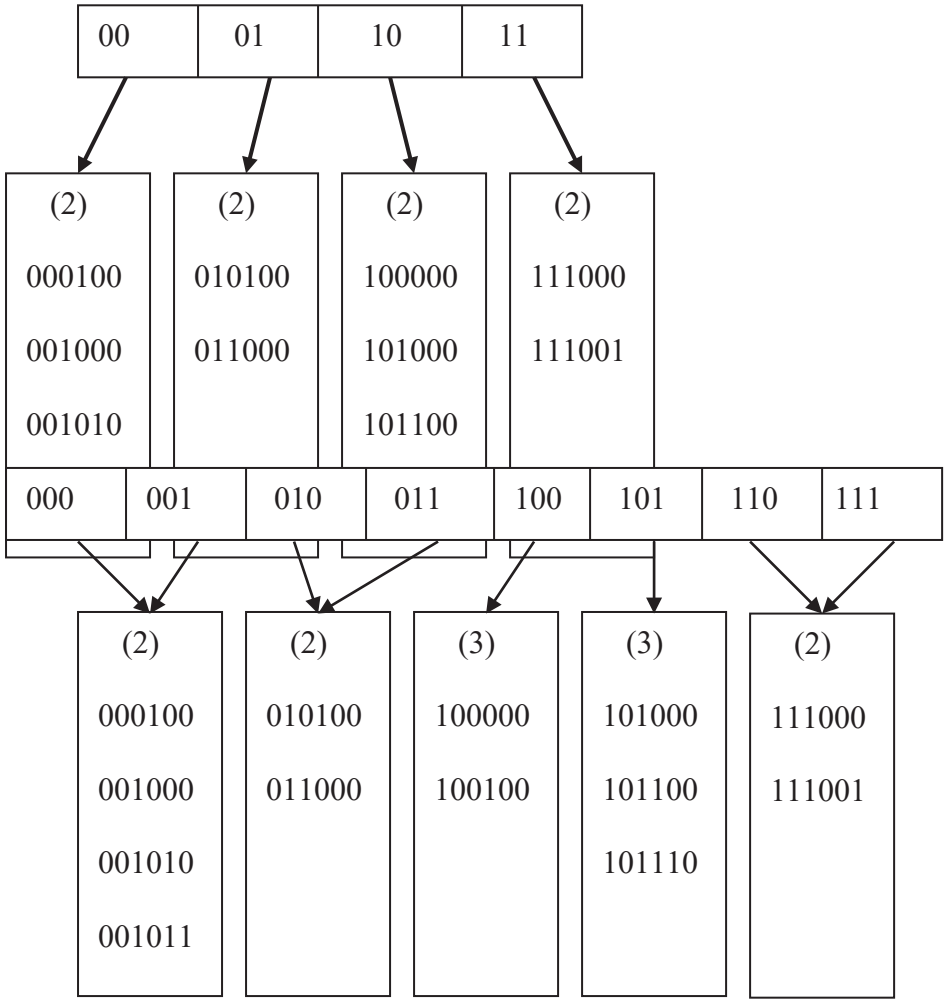
8.3.4.3 Key offset:

Key offset is a double hashing method that produces different collision paths for different keys. Whereas the Pseudorandom number generator produces a new address as a function of the previous address, key offset calculates the new address as a function of the old address and the key. One of the simplest version adds the quotient of the key divided by the list size to the address to determine the next collision resolution.

```
Offset= key/tablesize;
address= (offset+old address)%tablesize +1
```

In extendible hashing, the root tree is created with the possible combination of the leading bits. When a key is to be inserted, first the leading bits are compared. Depend on the leading bits; it is placed on the bucket connected with it. Let us suppose, for the moment, that our data consists of several six-bit integers. The root of the "tree" contains four pointers determined by the leading two bits of the data. Each leaf has up to M = 4 elements. It happens that in each leaf the first two bits are identical; the number in parentheses indicates this. Suppose that we want to insert the key 100100. This would go into the third leaf, but as the third leaf is already full, there is no room. We thus split this leaf into two leaves, which are now determined by the first three bits. This requires increasing this directory size to 3. This very simple strategy provides quick access times for Insert and Find operations on large databases. There are a few important details we have not considered.

Figure: Extendible hashing: original data



First, it is possible that several directory splits will be required if the elements in a leaf agree in more than D + 1 leading bits. For instance, starting at the original example, with D = 2, if 111010, 111011, and finally 111100 are inserted, the directory size must be increased to 4 to distinguish between the five keys. This is an easy detail to take care of, but must not be forgotten. Second, there is the possibility of duplicate keys; if there are more than M duplicated, then this algorithm does not work at all.

The expected number of leaves is (N/M) $\log_2$ e. The more surprising result is that the expected size of the directory (in other words, 2^d) is $O(N^{1+1/M}/M)$. If M is very small, then the directory can get unduly large.

8.3.8 Uses of Hashing

1. Compilers uses hash tables to keep a track of declared variables in source code. This data structure is known as symbol table
2. A hash table is used for any graph theory problem where the nodes have real names instead of numbers.
3. A third common use is in playing games. As program searches through different lines of play, it keeps a track of position it has seen by computing a hash function based on position
4. Hashing is used on on-line spelling checkers.

8.4. Search Trees

In search trees, we will see the following aspects

1. Binary search trees
2. Balanced Trees: AVL Tree
3. Multiway-search tree: B & B+ trees
4. Tries

8.4.1 Binary Search Tree:

A binary search tree is a binary tree with the following properties

1. All the item in the left sub-tree are less than the root
2. All the item in the right subtree are greater than or equal to the root
3. Each subtree is itself is a binary search tree.

Note: For insertion, deletion, Traversal techniques refer chapter 5 from data structures notes.

Efficiency of Binary Search Tree Operations

Best Case and worst case: If elements are inserted into the tree by the foregoing insertion algorithm. If the records are inserted in sorted (or reverse) order, the resulting tree contains all null left (or right) links, so that the tree search reduces to a sequential search ie O(n).

Average Case: If the records are presented in random order, balanced trees result more often than not, so that on the average, search time remains O(log n). For proof, refer Binary search average case.

8.4.2 Balanced Trees

Trees With a worst- Case height of O(logn) are called height **balanced trees** . One of the more popular balanced trees, known as an **AVL trees,** was introduced in 1962 by Adelson –Velskii and Landis.

AVL –Trees:

Defnition: An empty binary tree is an AVL tree. If T is a nonempty binary tree with TL and TR as its left and right subtress, then T is an AVL tree iff (1) TL and TR are AVL trees and (2) | hL – hR | < 1 where HL and HL are the heights of TL and TR, respectively .

1. The height of an AVL tree with *n* Elements / nodes is O(logn) .
2. For every value of n, n > 0**,** there exits an AVL tree.
3. An *n*-element AVL searched in O(height) =O(logn) time.
4. A new element can be inserted into an *n* element AVL search tree so that the result is an *n* + 1 element AVL tree and such an insertion can be done in O(log*n*)time .
5. An element can be deleted from an *n* element search tree so that the result is an *n*-1 element AVL tree and such a deletion can be done in O(log*n*) time.

Note: Known as perfectly balanced tree.

8.4.2.1 Rotations:

Insertion and deletion can cause imbalance in an AVL-Tree. In that circumstance, we use two kinds of rotations to make the tree balance.

i). Single Rotation
ii).Double Rotation

i) Single Rotation

Insertion of a node at the left of left or on the right of right causes imbalance, then we use single rotation

if Left of left causes imbalance

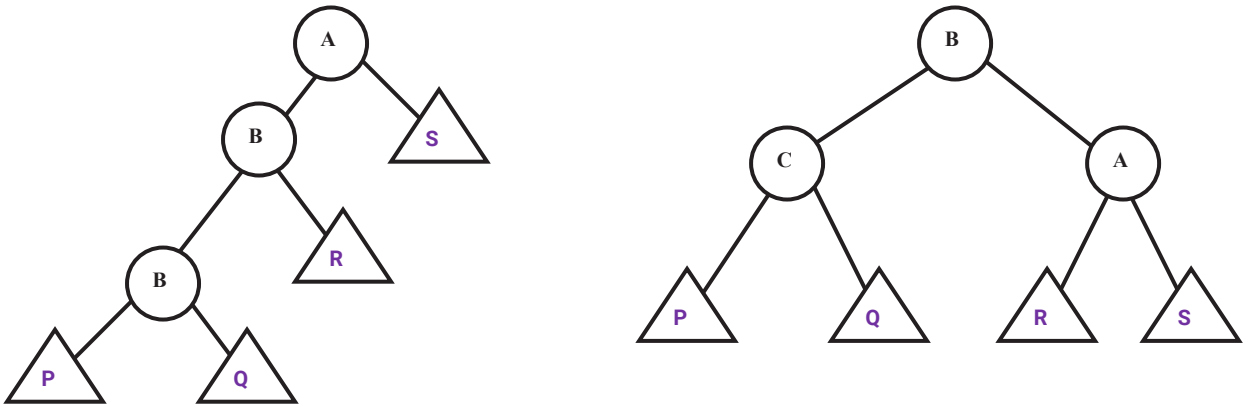
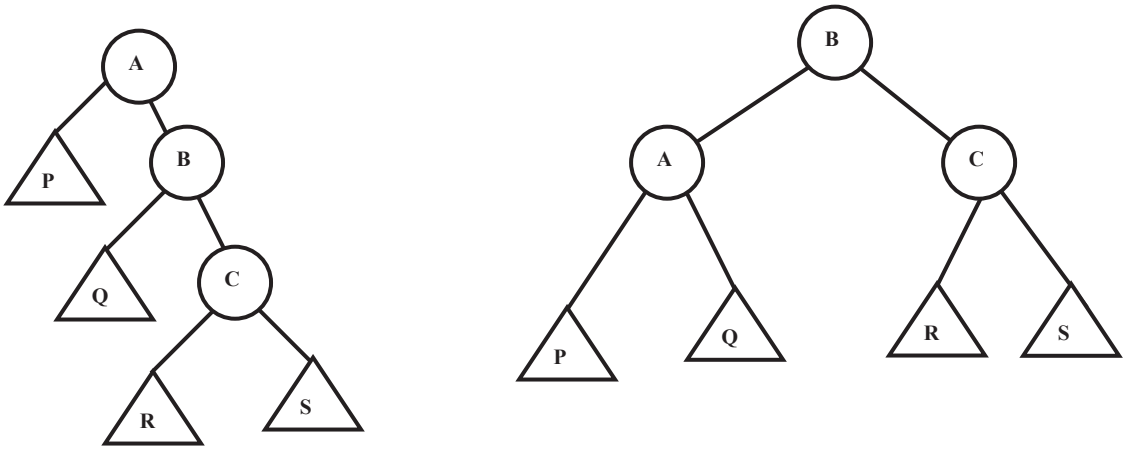


Figure: AVL Property destroyed by insertion of 6, then fixed by a single rotation

```
Algorithm rotatRight (ref  root < tree pointer >)
This algorithm exchanges pointers to rotate the tree right.
      Pre   root points to tree to be rotated.
      Post: Node rotated and root updated.
1   tempPtr       = root - > left
2   root - > left             =  tempPtr - > right
3   tempPtr - > right   = root
4   root          = tempPtr
5   return root
end rotateRight
```

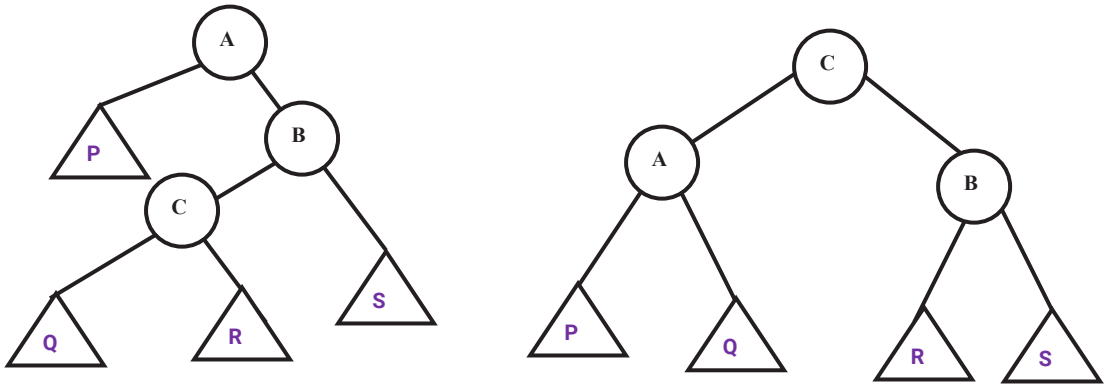
2. If right of right causes imbalance




```
Algorithm rotateLeft (ref root < tree pointer>)
This algorithm exchanges pointers to rotate the tree left.
      Pre   root points to tree to be rotated.
      Post  Node rotated and root updated.
1     tempPtr     =root - >right
2     root -> right     =tempPtr ->left
3     tempPtr ->left    =  root
4     root   =    tempPtr
5     return root.
6     end   rotateLeft
```

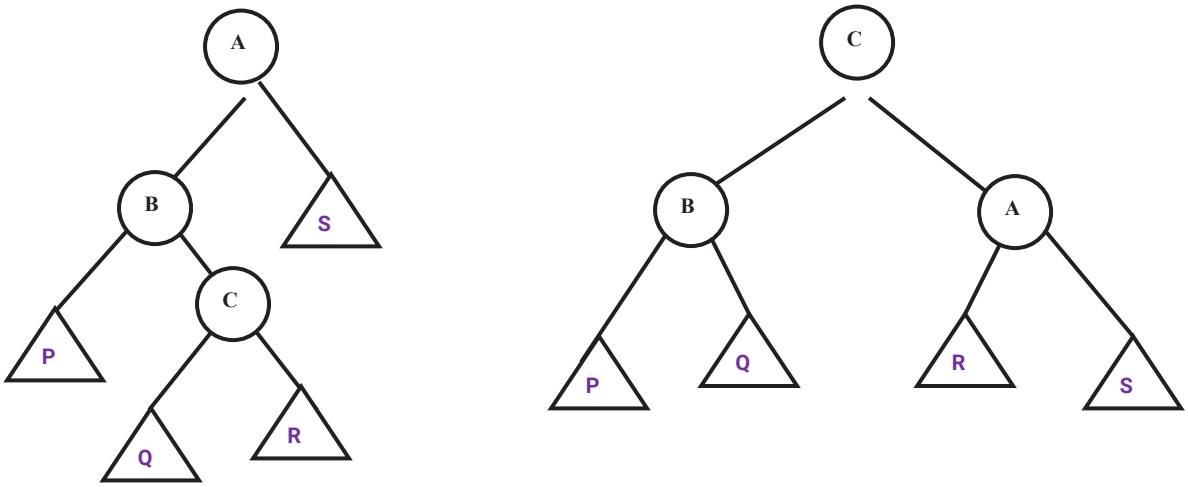
ii) Double Rotation

1. Left-right double rotation




```
Double RotateLeft ( root `
      Pre :       root p          the true to be rotated
      Post: Node rotated .        . updated
            1     root  ft    =   single rotateright (root->left)
            2 returns   singlerotatedLeft  ( root);
```

2. Right-left double rotation




```
Double Rotateright   < root >
Pre :              root points to the tree to be rotated
Post:              Node rotated and root updated
      1     root  right    =    Singlerotateleft(root-> right)
      2.     return      singlerotateright   (  root  )
```

8.4.2 Insertion in an AVL tree

We can insert a new mode in an AVL tree by finding its appropriate position . But insertion of a new node involves certain overheads since the tree may become unbalanced due to the increase in its heights .

If the new node is inserted as a child of any non-leaf node then there will be no effect on its balance, as the height of the trees does not increase .
If the new node is inserted as a child of leaf node then there is possibility that the tree may become unbalanced .
If the new node is inserted as a child of leaf node of sub-tree of shorter height then there will be no effect on the balance of an AVL tree.
If the height of the left and the right sub –tree is same then the insertion of a new node on any of the leaf does not disturb the balance of an AVL tree .In this case even if the height of a sub tree increases by one there will be no effect on the balance of the tree
If the new node is inserted as a child of the leaf node of taller sub –tree (left or right) then the AVL tree becomes unbalanced and the tree no longer remains an AVL tree.

Insertion Algorithm for AVL Tree:

1: Find the place to insert the new element by following a path from the root as in a search for an element with the same key. During this process, keep track of the most recently seen node with balance factor –1 or 1. Let this node be A. If an element with the same key is found the insert fails and the remaining steps are not performed.

2: If there is no node A, then make another pass from the root, updating balance factors. Terminate following this pass.

3: If bf (A) = 1 and the new node was inserted in the right sub-tree of A or if bf (A) = -1 and the insertion took place in the left sub-tree, then the new balance factor of A is zero. In this case update balance factors on the path from the new node and terminate.

4: Classify the imbalance at A and perform the appropriate rotation. Change balance factors as required by the rotation as well as those of nodes on the path from the new sub-tree root to the newly inserted node.

8.4.3 AVL Tree delete Algorithm

Like a Binary search tree, the deletion happens in only in leaf node or leaf like node.

1: Search for the node to be deleted.
2: If the left and right child of the node to be deleted is empty, then assign null to its parent.
3: If the left alone is null or right alone is empty then assign the non-empty child to it parent.
4: If both the children are non-empty search for a successor and replace the value and delete the successor.
5: The node deleted is on the right side of a parent, then rebalance the tree, by rotating on the left.
6: The node deleted is on the left side of a parent, then rebalance the tree by rotating on the right.

8.4.4 Splay tree

A **splay tree** is a self-balancing binary search tree with the additional property that recently accessed elements are quick to access again by keeping on the root. It performs basic operations such as insertion, look-up and removal in O(log(n)) amortized time. For many non-uniform sequences of operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. The splay tree was invented by Daniel Sleator and Robert Tarjan.

The splay operation

When a node *x* is accessed, a splay operation is performed on *x* to move it to the root. To perform a splay operation we carry out a sequence of *splay steps*, each of which moves *x* closer to the root.

The three types of splay steps are:

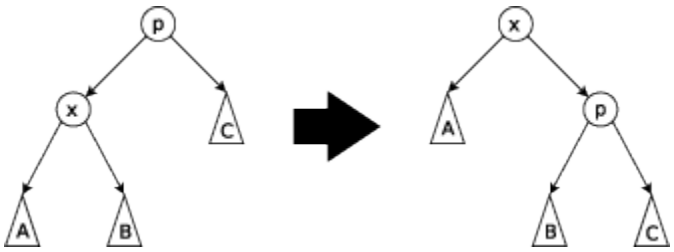
Note *:In the following graphical representation of rotation, the following convention is used.*

x *node to be accessed*

p *parent*

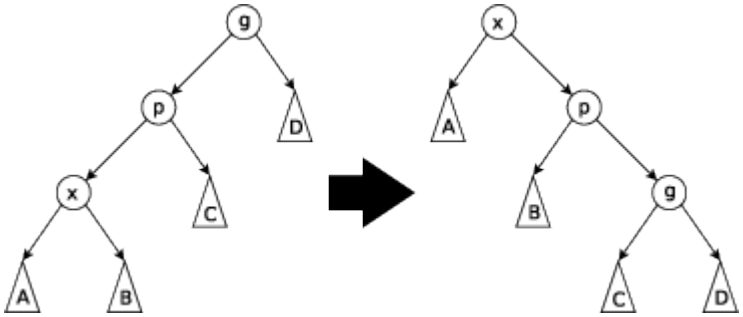
g *grandparent*

Zig Step: x doesn't have a grand parent ie) *p* is the root. The tree is rotated on the edge between *x* and *p*. Zig steps exist to deal with the parity issue and will be done only as the last step in a splay operation and only when *x* has odd depth at the beginning of the operation.

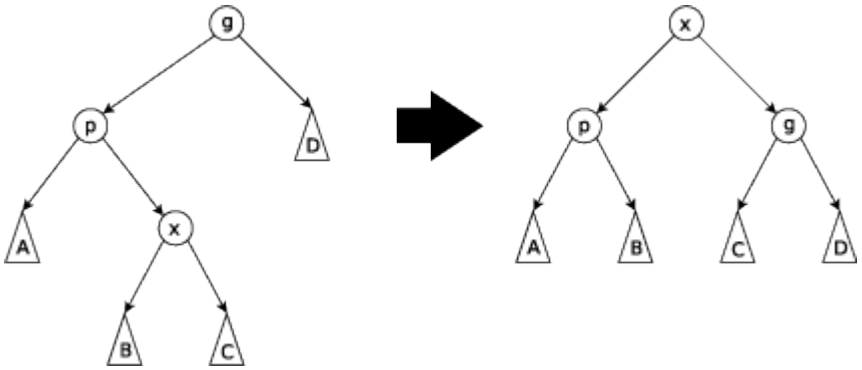


Zig-zig Step: x is at Left of Left or Right of Right of g.

This step is done when *p* is not the root and *x* and *p* are either both right children or are both left children. The picture below shows the case where *x* and *p* are both left children. The tree is rotated on the edge joining *p* with *its* parent *g*, then rotated on the edge joining *x* with *p*. Note that zig-zig steps are the only thing that differentiate splay trees from the *rotate to root* method introduced by Allen and Munro prior to the introduction of splay trees.



Zig-zag Step: x is at Left of Right and Right of Left of g. The tree is rotated on the edge between *x* and *p*, then rotated on the edge between *x* and its new parent *g*.



Advantages and disadvantages

Good performance for a splay tree depends on the fact that it is self-balancing, and indeed self optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly. This is an advantage for nearly all practical applications, and is particularly useful for implementing caches and garbage collection algorithms; however it is important to note that for uniform access, a splay tree's performance will be considerably (although not asymptotically) worse than a somewhat balanced simple binary search tree.

Splay trees also have the advantage of being considerably simpler to implement than other self-balancing binary search trees, such as red-black trees or AVL trees, while their average-case performance is just as efficient. Also, splay trees do not need to store any bookkeeping data, thus minimizing memory requirements. However, these other data structures provide worst-case time guarantees, and can be more efficient in practice for uniform access.

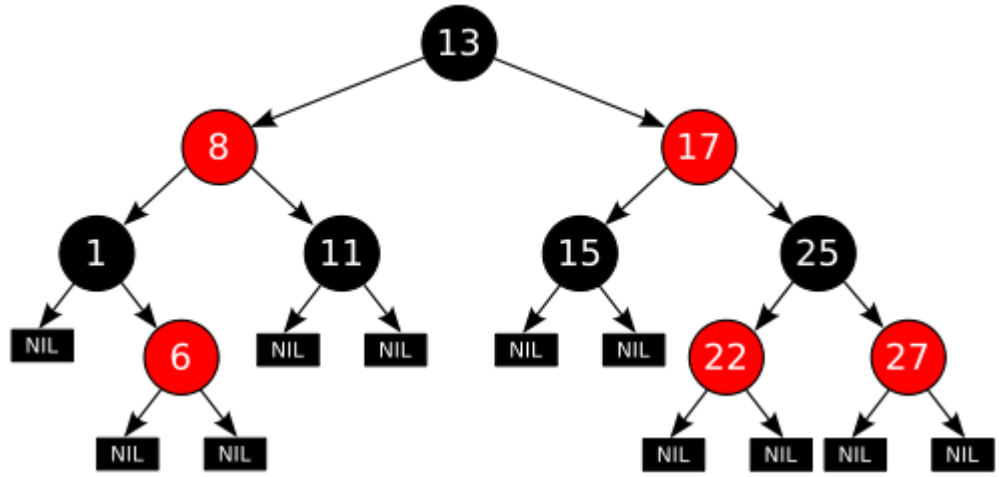
One worst case issue with the basic splay tree algorithm is that of sequentially accessing all the elements of the tree in the sorted order. This leaves the tree completely unbalanced (this takes n accesses - each a O(log *n*) operation). Re-accessing the first item triggers an operation that takes O(n) operations to rebalance the tree before returning the first item. This is a significant delay for that final operation, although the amortized performance over the entire sequence is actually O(log *n*). However, recent research shows that randomly rebalancing the tree can avoid this unbalancing effect and give similar performance to the other self-balancing algorithms.It is possible to create a persistent version of splay trees which allows access to both the previous and new versions after an update. This requires amortized O(log *n*) space per update.

Contrary to other types of self balancing trees, splay trees work well with nodes containing identical keys. Even with identical keys, performance remains amortized O(log *n*). All tree operations preserve the order of the identical nodes within the tree, which is a property similar to stable sorting algorithms. A carefully designed find operation can return the left most or right most node of a given key.

8.4.4 Red –Black Tree

A red-black tree is a special type of binary tree, used in computer science to organize pieces of comparable data, such as numbers.

In red-black trees, the leaf nodes are not relevant and do not contain data. These leaves need not be explicit in computer memory — a null child pointer can encode the fact that this child is a leaf — but it simplifies some algorithms for operating on red-black trees if the leaves really are explicit nodes. To save memory, sometimes a single *sentinel* node performs the role of all leaf nodes; all references from internal nodes to leaf nodes then point to the sentinel node.



An example of a red-black tree

A red-black tree is a binary search tree where each node has a *color* attribute, the value of which is either *red* or *black*. In addition to the ordinary requirements imposed on binary search trees, the following additional requirements of any valid red-black tree apply:

1. A node is either red or black.
2. The root is black. (This rule is used in some definitions and not others. Since the root can always be changed from red to black but not necessarily vice-versa this rule has little effect on analysis.)
3. All leaves are black, even when the parent is black (The leaves are the *null* children.)
4. Both children of every red node are black.
5. Every simple path from a node to a descendant leaf contains the same number of black nodes, either counting or not counting the null black nodes. (Counting or not counting the null black nodes does not affect the structure as long as the choice is used consistently.)
6. the longest path from the root to a leaf is no more than twice as long as the shortest path from the root to a leaf in that tree. The result is that the tree is roughly balanced.

7. The shortest possible path has all black nodes, and the longest possible path alternates between red and black nodes

8.4.4 M-Way Search Trees

An **m-way tree** is a search tree in which each node can have from zero to m sub-trees, where m is defined as the order of the tree, given a nonempty multiway tree, we can identify the following properties.

- Each node has 0 to m sub-trees
- Given a node with k < m subtrees, the node contains k subtree pointer, some of which may be null, and k – 1 data entries.
- The key value in the first subtree are all less than the key in the first entry; the key values in the other subtrees are all greater than or equal to the key in their parent entry.
- The keys of the data entries are ordered, $key_1 \leq key_2 \leq \ldots\ldots \leq key_k$.

8.4.4.1 B- Tree:

A B-tree is an M-way search tree with the following properties.

- The root is either a leaf or it has 2 ……. m Sub trees.
- All internal nodes have at least m/2 non-null sub-trees and at most m non-null sub-trees.
- All leaf nodes are at the same level that is the tree is perfectly balanced.
- The Key values in first sub-trees are all greater than or equal to the key in their parent entry.
- All leaf nodes have at least [m/2] – 1 and at most m – 1 entry.

8.4.4.1.1 Insertion in B-Tree

Like s binary search tree, B-tree insertion takes place at a leaf node.

1. Search for the leaf node where the data to be inserted.
2. If the node has less than m-1 entries then the new data are simply inserted in sequence.
3. If the node is full, the insertion causes overflow. Therefore split the node into two nodes. Bring the median data to its parent left entries of the median to copied into the left sub-tree and right entries copied into right sub-tree.

While splitting, we come across the following circumstances.

1. The new key is less than the median key.
2. The new key is the median key.
3. The new key is greater than the median key.

If the new key is less than or equal to the median key the new data belong to the left or original node, if the new key is greater than the median key, the data belong to the new node.

8.4.4.2 Deletion in B-Tree:

The deletion must always happen at the leaf node.

1. Search for the entry to be deleted.
2. If found continue else terminate.
3. If it is a leaf simply delete it.
4. If it is an internal node find a successor from its sub-tree and replace it. There are two data can substitute, either the immediate predecessor or immediate successor.
5. After deleting if the node has less than the minimum entries known as underflow then continue else terminate.

8.4.4.2.1 Balancing or combining

When underflow happens in a node it is handled by two concepts.

1. Balancing (splitting).
2. Combine (merging).

1. **Balancing:** It shifts data among nodes to reestablish the integrity of the tree. Because it does not change the structure of the tree. We balance a tree by rotating an entry from one sibling to another through parent. We determine the direction of the rotation depends on the number of entries in a left or right sub-tree.

2. **Combine:** Combine joins the data from an under-flowed entry, a minimal sibling and a parent node. The result is one node with maximum entries and an empty node that must be recycled. It makes no difference which sub-tree has under- flowed, we combine all the nodes into the left sub-tree. Once this has been done we recycle the right sub-tree.

Deletion in B-Tree:

Analysis:

If M is the ordrer of a B-tree, then

- Minimum number of keys in a B-Tree of order M ≥2 and height h≥0 is $2[M/2]^h-1$
- The depth of the tree is at most $\log_{M/2} N$.
- We make O(log N) search to determine the node to insert or to delete
- To insert a value in a node maximum search can be made is O(M)
- The worst case running time for deletion and insertion of a value is $O(M\log_M N)$

Advantages:

The real use of B-tree lies in Database systems where the tree is kept on a physical disk instead of main memory. It is not possible to maintain the entire structure in a primary storage(RAM). But if it is maintained in the secondary storage, is significantly slower than RAM, because the system often spends more time in retrieving the data than actually processing the data.

8.4.4.3 B-TREE VARIATION

B*Tree

When a B-tree is used to store a large number of entries, the space requirements can become excessive. This is because up to 50% of the entries can be empty. The first variation, the **B*tree**, addresses the space usage for large trees. Rather than each node containing a minimum of one-half the maximum entries, the minimum is set at two thirds.

In B*tree, when a node overflows, instead of being split immediately, the data are redistributed among its siblings, delaying the creation of a new node. Splitting occurs only when all the siblings are full. Furthermore, when the nodes are split, data from two
Full siblings are divided among the two full nodes and a new node with the result that all three nodes are two –thirds full.

B+Tree

In large file system, data need to be processed both randomly and sequentially. In these situation, the most popular file organization methods use the B-tree to process the data randomly. However, a lot of processing time is taken up moving up and down the tree structure when the data need to be processed sequentially. This has led to the second B-tree variation, the **B+tree.**

There are two differences between the B-tree and the B+tree.

1. Each data entry must be represented at the leaf level. This is true even though there may be internal nodes with the same keys. Because the internal nodes are used only for searching, they generally do not contain data.
2. Each leaf node has one additional pointer, which is used to move to the next leaf node in sequence. However, process the data sequentially, we simply locate the leftmost entry and then process the data as though we were processing a linked list in which each node is an array.

8.4 Tries:

A digital search tree contain m pointers, corresponding to the m possible symbols in each position of the key. Thus, if the keys were numeric, there would be 10 pointers in a node, and if strictly alphabetic, there would be 26. A pointer in a node is associated with a particular symbol value based on its position in the node; the first pointer corresponds to the lower symbol value, the second pointer to the second lowest, and so forth. The number of nodes that must be accessed to find a particular key is log mn. A digital search tree implemented in this way is called a trie.

A trie is useful when the set of keys is dense, so that most of the pointers in each node are used. When the key set is sparse, a trie wastes a large amount of space with large nodes that are mostly empty.

9. Heaps

Defn: A priority queue(heap) is a data structure that allows to operate on the priority of the elements with the DelMin or DelMax operation. It is a binary tree with the following two properties

1. Structured property
2. Heap order property

Structured Property

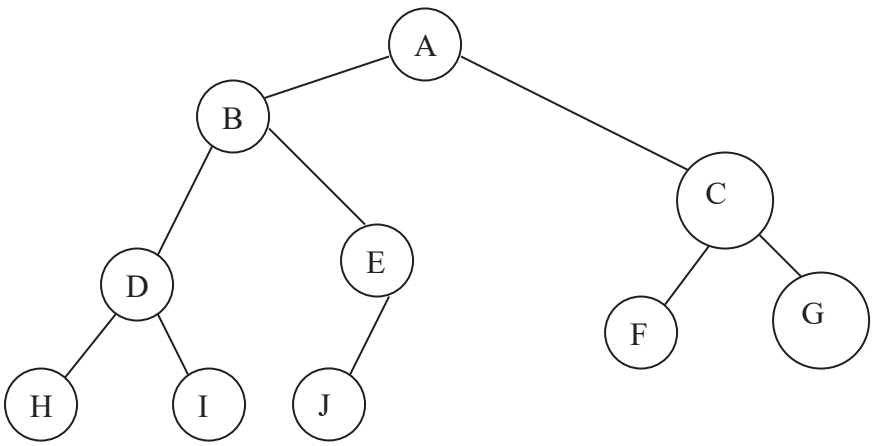
- A heap is complete binary tree or almost complete binary tree. The elements are stored in the heap from left to right.
- Since it is a complete binary tree height of the tree is 0 (logN).
- If i is the node then 2i is the position of its left child 2i+1 is the position of its right child $i/_2$ is the parent

Heap order property.

The heap can be represented either for an ascending priority Queue or for descending priority Queue,

- If ascending priority Queue then parent stores lower value than its children with the DelMin Operation.. if descending priority queue then the parents stores the higher value than the children.

- Every node is a heap.



-1	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10			

9.1 Basic Heap Operations

Insert

To insert an element X into the heap, we create a hole in the next available locations, otherwise the tree will not be complete. If X can be placed in the hole without violating the heap order, then we do so. Otherwise we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root. We continue this process until X can be placed in the hole. This general strategy is known as ***percolate up*** or ***reheap*** up. The new element is percolated up the heap until the correct location is found.
If the element to be inserted is the new minimum, it will be pushed all the way to the top. At some point, 'i' will be 1 and we will want to break out of the while loop. We could do this with an explicit test, but we have chosen to put a very small value in position 0 in order to make the while loop terminate. This value must be guaranteed to be smaller than any element in the heap; it is known as a sentinel.

DeleteMin

DeleteMin are handled in a similar manner as insertions. Finding the minimum is easy; the hard part is removing it. When the minimum is removed, a hole is created at the root. Since the heap now becomes one smaller, it follows that the last element X in the heap must move somewhere in the heap. If X can be placed in the hole, then we are done. This unlikely, so we slide the smaller of the hole's children into the hole, thus pushing the hole down one level. We repeat this step until X can be placed in the hole. Shifting the hole down wards as known as ***percolate down*** or ***reheap*** down.

DeleteMax

It deletes the max element in the heap. The max element is stored at the root. Rest of the procedure is same as above.

```
struct heap
{
int arr[size];
int pos;
};

void insert(struct heap *p,int val)
{
int i;
if(p->pos==size-1)
printf("\nHeap is full");
else
{
for(i=++p->pos;p->arr[i/2]>val;i=i/2)
p->arr[i]=p->arr[i/2];
p->arr[i]=val;
}
}


int deleteMin(struct heap *p)
{
int temp,val,i,child;
if(p->pos==0)
{
```

```
printf("\nHeap is empty");
val=' ';
}
else
{
temp=p->arr[p->pos--];
val=p->arr[1];
for(i=1;2*i<p->pos;i=child)
{
child=2*i;
if(p->arr[child]>p->arr[child+1]) /* To find whether the left or right child is greater*/
child++;
if(p->arr[child]<temp)   /*To compare the value to be settled with the higher child*/
p->arr[i]=p->arr[child];
else
break;
}
p->arr[i]=temp;
}
return val;
}
void display(struct heap p)
{
int i;
if(p.pos==0)
printf("\nHeap is empty");
else
for(i=1;i<(p.pos+1);i++)
printf("%d  ",p.arr[i]);
}

void main()
{
int ch,n,val;
struct heap h;
h.arr[0]=-1;
h.pos=0;
clrscr();
do
{
printf("\nEnter the choice:\n1) insert\n2) delete\n3) display\n4) exit\n");
scanf("%d",&ch);
switch(ch)
{
case 1:
      printf("\nEnter the value:");
      scanf("%d",&val);
      insert(&h,val);
       break;
case 2:n=deleteMin(&h);
       if(n!=' ')
       printf("\nThe removed value is:%d",n);
       break;
case 3:display(h);
       break;
}}
while(ch!=4);
getch();
}
```

9.2 Applications of Heap

Three common applications of heaps are

1. Selection algorithm
2. Event Simulation
3. Sorting

3.2.1 Selection Algorithms

Given the problem to determine the kth element in an unsorted list, we have two solutions.

- We could first sort the list and select the element at location k, or we could create a heap and delete k – 1 elements from it, leaving the desired element at the top.
- We return to the original problem and find the *k*th largest element. We use the idea from the abobe algorithm . At any point in time we will maintain a set S of the k largest elements. After the first k elements are read, when a new element is read it is compared with the kth largest element, which we denote by S_k. Notice that S_k is the smallest element in S. If the new element is larger, then it replaces S_k in S. S will then have a new smallest element, which may or may not be the newly added element. At the end of the input, we find the smallest element in S and return it as the answer.

3.2.2 Event Simulation

We described an important queuing problem. Recall that we have a system, such as a bank, where customers arrive and wait on a line until one of k tellers is available. Customer arrival is governed by a probability distribution function, as is the service time. We are interested in statistics such as how long on average a customer has to wait or how long the line might be.

A simulation consists of processing events. The two events here are (a) a customer arriving and (b) a customer departing, thus freeing up a teller. We can use the probability functions to generate an input stream consisting of ordered pairs of arrival time and service time for each customer, sorted by arrival time. We do not need to use the exact time of day. Rather, can use a quantum unit, which we will refer to as a tick.

3.2.3 Sorting

An unordered array can be sorted using heaps. In involves two procedure that is constructing the array as a heap (descending). Then copy the root value at the last location of the array and then reconstruct the heap. Continue the procedure until there is only one element in the tree.

10. Sorting

Defn: Sorting is a process by which the records are arranged in an some order depending on the key. The order may be descending, ascending or alphabetical.

The following factors are determines the efficiency of a sorting algorithm

- **Data Sensitivity :** In this aspect we analyze whether a particular sorting method reduces the execution time if the keys are partially sorted or sorted.
- **Stability:** This aspect analyses, whether the all the non-distinct elements are arranged in the same order after sorting as they were before.
- **Memory :** A list can be sorted either through a *comparative* method or distributive method. In comparative method we compare the elements within the array where as in *distributive* method we distribute the elements with certain principles. After we merge them. In this case we require an extra memory space. It may be same as the list size or more.
- **Time:** This aspect is given through the Best case, When the array
- is sorted , Worst case, when the list is in reverse order and the average case, if the elements are stored in random.

Sorting methods are classified based on their functions

1. Exchange Sort: Bubble Sort and quick Sort
2. Insertion Sort : Insertion Sort and Shell sort.
3. Selection Sort: Straight selection sort, heap sort and binary tree sort
4. External Sort: Merge Sort and radix Sort, bucket sort

10.1 Exchange Sorts

10.1.1 Bubble Sort:

Concept:

- Interchange the adjacent keys if the j^{th} key is greater than the $(j+1)^{th}$ key. Begin j from 0 until j and (n-i) are equal. Where i is the i^{th} pass. This completes one pass
- Continue the above procedure n-1 times

Note: For the i^{th} pass, the i^{th} largest element is settled at the i^{th} last location. Therefore we iterate the inner loop n-i times.

The unsorted array is:

	3	5	1	7	2	6	4
Pass 1:	3	1	5	2	6	4	**7**
Pass 2:	1	3	2	5	4	**6**	**7**
Pass 3:	1	2	3	4	**5**	**6**	**7**
Pass 4:	1	2	3	**4**	**5**	**6**	**7**
Pass 5:	1	2	**3**	**4**	**5**	**6**	**7**
Pass 6:	1	**2**	**3**	**4**	**5**	**6**	**7**

Algorithm

1. Establish an array with 'n' elements
2. Let i = 1
3. Let j = 0
4. Swap A [j] with A [j + 1] if the first is greater than the second
5. Increment j by 1
6. Repeat 4 and 5 n - i times
7. increment i by 1
8. Repeat step 3 to 7 n – 1 times
9. end

```
void bubble(int arr[],int n)
{
int i,j,temp;

for(i=1;i<n;i++)
{
      for(j=0;j<n-i;j++)
      {     if(arr[j]>arr[j+1])
            {     temp=arr[j];
                  arr[j]=arr[j+1];
                  arr[j+1]=temp;
            }
      }
}
}
```

Analysis of Bubble Sort:

Number of Passes: n-1.

Bubble sort makes (n-i) comparison for the i^{th} pass. Whether the array is sorted or unsorted, the total number of passes is always n-1 and for every pass there is always n-i comparison where i is the i^{th} pass. Therefore the best case , worst case an average case are same.

$$\mathbf{F(n)} = \sum_{I=1}^{n-1} i = O(n^2)$$

Variations on Bubble Sort

10.1.1.1 Modified Bubble Sort

In the above version of the bubble sort, even after the array is sorted at the 3rd pass, we continued the iteration. This drawback is removed in the modified bubble sort where if there is no swapping occurred in a pass, it concludes that the array is sorted and the process is terminated. This concept is obtained through a flag variable.

The unsorted array is: 3 5 1 7 2 6 4

Pass 1:	3	1	5	2	6	4	7
Pass 2:	1	3	2	5	4	6	7
Pass 3:	1	2	3	4	5	6	7
Pass 4:	1	2	3	4	5	6	7

```
void Modifybubble(int arr[],int n)
{
   int i,j,temp,x=n,flag=1;
   for(i=1;i<n&&flag;i++)
   {         flag=0;
         for(j=0;j<n-i;j++)
         {
               if(arr[j]>arr[j+1])
               {     temp=arr[j];
                     arr[j]=arr[j+1];
                     arr[j+1]=temp;
                     flag=1;
               }
         }
   }
}
```

Analysis of Modified Bubble Sort:

Best Case: The best case occurs if the array is sorted. Since the array is sorted , the condition is not satisfied in the first iteration Hence the flag remain 0. Hence it does not come for the second iteration.

Worst Case : If the array is in the reverse order, there will be always swapping until the smallest element comes into the first location. Hence there will be n-1 passes. It is same as Bubble Sort. Therefore

$$T(n) = O(n^2)$$

Average Case: The average case is also $O(n^2)$. Where the constant value reduces.

10.1.1. 2 Shaker Sort

In the bubble sort, the largest value is settled at the last location at the first pass and the second largest at the second pass. Where as in shaker sort, in the first pass, the largest value and the smallest value is settled at the first and last position . In the second pass, the second largest and the second smallest are settled and so on. The procedure is continued till we reach the middle.

```
void shaker (int arr[], int n)
{     int i, j, temp, flag = 1,x;
      for (i = 0; flag; i++)
      {     flag = 0;
            for (j = i; j < n -1-i; j ++)
            if (arr[j+1]<arr[j])
            {     temp = arr [j];
                  arr[j] = arr[j+1];
```

```
            arr[j+1] = temp;
            flag =1;
        }
        if (flag)
        { flag = 0;
        for (j=n-2-i;j>=i+1;j--)
        {   if (arr[j-1]>arr[j])
            {     temp = arr[j];
                  arr[j] =arr[j-1];
                  arr[j-1] = temp;
                  flag =1;
            }
        }
        }
        }
        }
```

The unsorted array is:

	7	6	5	4	3	2	1
Pass 0:	1	6	5	4	3	2	7
Pass 1:	1	2	5	4	3	6	7
Pass 2:	1	2	3	4	5	6	7
Pass 3:	1	2	3	4	5	6	7

Analysis of shaker Sort:

In this method, from the remaining elements, largest element and smallest elements are settled at the same time. Hence the number of passes are reduced .Whereas in the best case and in the worst case, the number of swapping remains same as the modifies version of bubble sort where as in the average case, the swapping is reduced slightly than the modified buuble sort. Even though average case is $O(n^2)$, the constant which needed to be multiplied with n^2 is lesser than modified bubble sort.

10.1.2 Quick Sort

Concept: Take the first element and settle it, in its location in the sorted array. The element that is to be settled is known as pivot. Once the pivot is settled, all the elements at the left side must be smaller than the pivot and right side must be larger than the pivot. Sort the left side and the right side of the pivot using the same concept.

1. Take the first value
2. Start comparing the elements from second location. If the current element is greater than the pivot then stop. Store the location in ‘i’ .
3. Start comparing from the last location. If the current element is smaller than the pivot then stop. Store the location in ‘j’
4. If ‘i’ is less than j then swap the i^{th} element with the j^{th} element and continue 2 and 3. If not;
5. Swap the pivot with the j^{th} element. Thus complete one pass
6. Divide the array into two, left side of the pivot and right side of the pivot. Sort them using the same concept.

The unsorted array is:

	4	2	6	3	5	7
Pass 1:	(3	2)	**4**	(6	5	7)
Pass 2:	(2)	**3**	4	(6	5	7)
Pass 3:	2	3	4	(5)	**6**	(7)

/*It settles the pivot at its location and returns the location of the pivot*/

```
int partition(int arr[],int low,int high)
{     int i,j,temp;
      i=low+1;
      j=high;
      x=arr[low];
      while(i<j)
      {     i++;
            while(arr[i]<x)
            i++;
            while(arr[j]>x)
            j--;
            if(i<j)
            {      temp=arr[i];
                  arr[i]=arr[j];
                  arr[j]=temp;
            }
      else
      break;
 }
  arr[low]=arr[j];
  arr[j]=x;
 return j;
}
```

Recursive QuickSort:

1. Establish an array with low and higher subscripts
2. Pivot = a [low]
3. Partition the array by settling the Pivot in its location
4. Sort the left array of the Pivot using Quick Sort
5. Sort the right array of the Pivot using Quick Sort
6. Repeat 3 – 6 until low becomes grater than higher.

```
void quick(int arr[],int low,int high)
{
      int j;
      if(low<high)
      {       int x;
            j=partition(arr,low,high);
            quick(arr,low,j-1);
            quick(arr,j+1,high);
      }
}
```

Non-Recursive QuickSort:

```
void quick(int arr[], int low, int high)
{
      struct
      { int lo, up;
      } stack [20],item;
      int top = -1, pos;
      top ++;
      stack [top].lo= low;
      stack [top].up= high;

      while (top!=-1)
      {
      low = stack [top].lo;
      high = stack [top].up;
      top --;
      if (low <high)
      {     pos=partition (arr, low, high);
            top ++;
            stack [top].lo = low;
            stack [top].up = pos -1;
            top ++;
            stack [top]. lo = pos +1;
            stack [top] .up = high;
       }
      }}
```

Analysis of Quick Sort:

Worst Case: If the array is sorted or on the reverse order, the pivot will be settled either at the leftmost or on the rightmost respectively. Hence for every settlement of the pivot totally n-i comparison will be made. where I is the location of the pivot .
Therefore time required to sort the array can be calculated through the recursive relations

$T(N) = T(N-1) + CN$

By expanding the recurrence relation

$T(N-1) = T(N-2) + C(N-1)$

……

Adding up all these equations

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

$$= O(N^2)$$

Best Case : If the pivot is always comes at the middle, then the best case happens. In this case two sub arrays are created with the same size. Hence we derive the recursive relation

$T(N) = 2T(N/2)+CN;$

Divide both sides by N

$T(N) /N= T(N/2)/(N/2)+c$

By expanding the recurrence relation, we get

T(N)/N = T(1)/1 + ClogN

Which yields

T(N) = CNlogN+N
=O(NlogN)

Average Case: The average case occurs if the elements are randomly distributed.

T(N) = O(NlogN)

10.2 Insertion Sort:

10.2.1 Straight Insertion Sort

There is always two arrays exists at the left hand side of the element to be settled and the right and side of the element to be settled. Left side is sorted and the right side is unsorted. If the element to be settled is lesser than the existing sorted array, then it is inserted on a right position.
Start the procedure from the second element. Sort the left side. The highlighted element indicates that the element at that position was inserted .

The unsorted array is:

	6	2	7	4	3	5	1
Pass 1:	2	**6**	7	4	3	5	1
Pass 2:	2	6	**7**	4	3	5	1
Pass 3:	2	4	6	**7**	3	5	1
Pass 4:	2	3	4	6	**7**	5	1
Pass 5:	2	3	4	5	6	**7**	1
Pass 6:	1	2	3	4	5	6	**7**

Algorithm

1. Establish an array with ‘n’ elements
2. Let I = 2
3. Insert A[i] into sorted array A [i] to A [i –1]
4. increment i by 1
5. Repeat 3 and 4 n – 1 times
6. Stop

```
void insertion(int arr[],int n)
{
int i,y,j,x=n;
for(i=1;i<n;i++)
{
      y=arr[i];
      for(j=i;j>0&&y<arr[j-1];j--)
      arr[j]=arr[j-1];
      arr[j]=y;
}
}
```

Analysis on Insertion Sort:

Best Case : if the input is presorted, the running time is O(N), because the inner for loop always fails immediately.

Worst Case : If the input is in reverse order, for every i^{th} iteration, the inner for loop makes i+1 comparisons to settle the elements. Here the I starts from 0 to n-1. Hence the total number of comparison made to sort the array is

$$\sum_{i-2}^{N} i = 2 + 3 + 4 + \ldots\ldots + N = (N^2-N)/2 = \Theta(N^2)$$

Average Case: if the input is almost sorted, insertion sort will run quickly. Because of this wide variation, it is worth analyzing the average-case behavior of this algorithm.
It turns out that the average case is

$= N^2/4$
$= \Theta(N^2)$

12345
analysus refer bruno 513

Shell Sort:

Shell sort creates a sub-list with an interval. The interval is initially taken as half the number of elements. Elements with the same interval are connected and created as a sub-list. Each sub list is sorted with the insertion sort. The interval in reduced into half of the current interval and the same procedure is followed until interval becomes 0.

Algorithm for Shell Sort:

ShellSort(List,N)

List : The list to be sorted
N: Number of elements in the list

1. Dist = N/2
2. Loop(Dist>=1)
   {
   1. start=Dist
   2. Loop(start<=N)
      1. Temp=List[start];
      2. index=start
      3. Loop(index>=start &&List[index-Dist]>temp)
         1. List[index]=List[index-Dist]
         2. index=index-Dist
      4..List[index]=temp
   3. start=start+1

3.Dist=Dist/2
4. End

Analysis of Shell Sort

The choice of the 'dist' can have a major effect on the order of shell sort. Number of different options are given here.

Worst Case:

1. Using Shell's diminishing increment ie) N/2 is $\Theta(N^2)$
2. Using Hibbard's increment ie) 1, 3, 7, …..2^k-1 is $\Theta(N^{3/2})$
3. Using Sedgwick increment sequence { 1,5, 19,41, 109 } is $O(N^{4/3})$

Average Case:

1. Using Shell's diminishing increment ie) N/2 is $O(N^2)$
2. Using Hibbard's increment ie) 1, 3, 7, …..2^k-1 is $\Theta(N^{5/4})$
3. Using Sedgwick increment sequence { 1,5, 19,41, 109 } is $O(N^{7/6})$

10. 3 Selection Sort:

10.3.1 Straight Selection Sort

Concept: To settle the smallest key at the i^{th} location from the table where i begins from 0 to n-1..

1. A search is made to locate the smallest key in the table from the i+1 location
2. to n.
3. Smallest Key is interchanged with the i^{th} location
4. Location is moved to the next i=i+1;
5. Procedure is continued until 'i' and n becomes equal.

The unsorted array is:

	2	6	4	5	1	3	7
Pass 1:	**1**	6	4	5	2	3	7
Pass 2:	1	**2**	4	5	6	3	7
Pass 3:	1	2	**3**	5	6	4	7
Pass 4:	1	2	3	**4**	6	5	7
Pass 5:	1	2	3	4	**5**	6	7
Pass 6:	1	2	3	4	5	**6**	7

Algorithm:

```
void selection(int arr[],int n)
{
      int i,j,x,temp,pos;

      for(i=0;i<n-1;i++)
      {     pos=i;
            for(j=i+1;j<n;j++)
            {         if(arr[j]<arr[pos])
                  pos=j;
            }
            temp=arr[pos];
```

```
            arr[pos]=arr[i];
            arr[i]=temp;
        }
}
```

Analysis of straight selection sort

Whether the array is sorted (Best case), unsorted(Average case) or in the reverse order (worst case) the number of passes is n-1. Every pass has (n-i) comparisons. Therefore the total number of comparisons are the sum of the comparisons

$$T(n) = \sum(n-i) \text{ for(I=1 to n-1)}$$
$$= ½\ n(n-1)$$
$$= O(n^2)$$

- Time required for all the three cases will be $O(n^2)$.
- The total number of interchanges is n-1

Since all the three cases require equal amount of time to sort the array, it is concluded that this sorting method is not data sensitive.

10.3.2 Heap Sort

Concept: The heap sort is an improved version of the selection sort in which the largest element (at the root) is selected and exchanged with the last element in the unsorted list.

This sorting method involves two procedures.

1. Construction of heap: where the array elements are arranged as a descending heap. Every node is considered as a heap and the construction begins from the n/2 th location till 0^{th} location.
2. After construction, the max element at the root is swapped with the last location. After swapping, if the last element, which is shifted to the root, does not fit into the location, then a suitable value is searched from either from the left or the right child. The traversal goes on until the last element find its suitable place. This procedure is known as reconstruction.
3. The above procedure is continued until there is one element in the array.

Algorithm for Heap Sort:

```
Heapsort( List, N)
List : The list to be sorted
N    : Number of elements in the List

   1. index=N/2
   2. Loop( index>=0)
         1. Construct the heap from index till N
         2. index=index+1
   3. index=N-1
   4.  Loop(index>0)
         1. swap(List[index], List[0]
         2. construct the heap from 0 to N
         3. index=index-1
   5. Stop
```

Note: Write algorithm for construction.

The unsorted array is:
3 6 4 1 7 2 8

COSTRUCTION of Heap
3 6 4 1 7 2 8
3 6 8 1 7 2 4
3 7 8 1 6 2 4
8 7 4 1 6 2 3

RECONSTRUCTION of HEAP
3 7 4 1 6 2 8
2 6 4 1 3 7 8
2 3 4 1 6 7 8
1 3 2 4 6 7 8
2 1 3 4 6 7 8
1 2 3 4 6 7 8

```
void construct(int arr[],int i,int n)
{
      int temp,child;
      for(temp=arr[i];(2*i+1)<n;i=child)
      {
            child=2*i+1;
            if(arr[child]<arr[child+1]&&child<n-1)
            child++;
            if(temp<arr[child])
            arr[i]=arr[child];
            else
            break;
      }
      arr[i]=temp;
}


void heapsort(int arr[], int n)
{
int i,temp,j,x;
                        /*construction of heap*/
for(i=n/2;i>=0;i--)
{  construct(arr,i,n);
}

for(i=n-1;i>0;i--)      /*Reconstruction of heap*/
{
temp=arr[0];
arr[0]=arr[i];
arr[i]=temp;
construct(arr, 0,i);
}
```

Analysis of Heap Sort:

Best Case =Worst Case =Average Case= O(NlogN)

10.3.3 Binary Tree Sort

Concept: Store the array in a binary search tree. Traverse the tree in-order and store it in the original array.

10.4 External Sorting

10.4.1 Merge Sort

Concept: The array is divided into half, until there is only one element in the array. Then each sub-sequent sub arrays are merged and formed an array of size 2. These arrays are merged again to form an array of size 4. This procedure is continued until the size of the array becomes same as the original size of the array.

The unsorted array is:
9 8 7 6 5 4 3 2 1

8	9	7	6	5	4	3	2	1
7	**8**	**9**	6	5	4	3	2	1
7	8	9	**5**	**6**	4	3	2	1
5	**6**	**7**	**8**	**9**	4	3	2	1
5	6	7	8	9	**3**	**4**	2	1
5	6	7	8	9	3	4	**1**	**2**
5	6	7	8	9	**1**	**2**	**3**	**4**
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

```
void merge(int arr[], int l1,int u1,int l2,int u2)
{
int temp[50],n=0,i,j;
i=l1;
j=l2;
while(i<=u1&&j<=u2)
temp[n++]=arr[i]<arr[j]?arr[i++]:arr[j++];
while(i<=u1)
temp[n++]=arr[i++];
while(j<=u2)
temp[n++]=arr[j++];
n=0;
for(i=l1;i<=u2;i++)
arr[i]=temp[n++];
}
```

Recursive merge sort:

```
void mergesort(int arr[], int low ,int high)
{
int mid;
if(low<high)
{
mid=(low+high)/2;
mergesort(arr,low,mid);
mergesort(arr,mid+1,high);
merge(arr,low,mid,mid+1,high);
}
}
```

Non-recursive Merge sort

```
void mergesort(int arr[], int n)
{

      int temp [10], i, j, k, lb1, lb2, ub1, ub2,size=1;
      while (size < n)
      {
            lb1 = 0; k=0;
            while (lb1+size<n)
            {     lb2 = lb1+ size ;
                  ub1 = lb2 -1;
                  (ub1 +size <n)?(ub2=ub1+size): (ub2=n-1);
                  i = lb1 ; j =lb2;
                  while (i <=ub1 && j <= ub2)
                  temp[k++]=arr[i]< arr [j]?arr[i ++]:arr[j++];
                  while (i <=ub1)
                  temp [k++] = arr[i++];
                  while (j <=ub2)
                  temp [k++] = arr[j++];
                  lb1 = ub2 + 1;
             }
             for (i=0; i <= ub2; i++)
             arr[i]=temp[i];
             size = 2 * size;
             }
        }
```

Analysis of MergeSort:

Since merge sort goes on dividing into half, this can be proved in the recurrence relationship. When there is one element in the list, time required to sort the array is 1.

$T(1) = 1$

If there are more than one element, then we go on dividing into half and then we merge. Time required to merge is linear. Therefore

$T(N) = 2T(N/2) + N$

Substituting

$T(N/2) = 2\ T(N/4) + N/2$

$T(N) = 4T(N/4) + 2N$

By expanding the recurrence

T(N)=NT(1) + NlogN

T(N) = NlogN +N
T(N) = O(NlogN)

The division and merging happens whether the array is sorted or unsorted. Therefore, the best case, worst case and the average case for mergesort is O(NlogN)

10.4.2 Bucket Sort:

Concept: Let all the elements in the list are within the range of the array (n). Then create a counter array to find the elements which are repeated and store it in the counter array. Arrange the elements in the array, depends on the counter array.

```
void bucket(int arr[],int n )
{
int i, j, count [size] = {0};
for (i = 0; i < n; i ++) /*find the number of repeated elements*/
{count [arr [ i ] ] ++;
}
for (i = 0, j = 0; i < n; i ++ )   /* To copy back */
     while (count [ i ] -- > 0)
     arr [ j ++] = i;
}
```

The unsorted array is: 5 3 7 5 2 3 1 8 4 9
The sorted array is: 1 2 3 3 4 5 5 7 8 9

Analysis of Bucket Sort:

To count the number of repeated elements , we traverse the entire list once. The time to traverse the list is linear. Therefore the traversing time is O(n).

In order to copy back, we use two loops. Therefore here we can not judge that it is quadratic. Because the maximum number of iteration is 2n. Because if all the elements are stored in the same bucket then the inner loop iterates n times. Since all the other elements are stored in the same bucket other buckets remains empty.. Hence the inner loop does not iterates for other buckets.

Time required to sort the array is time required to count O(N) and time required to copy back O(N).

T(N) = O(N)+ O(N) where the first
= O(N).

Hence the time required to sort the array using Bucket sort for all the cases is O(N).

Drawback: The drawback of this method is, it can sort only the elements are within the range of the size of the array. For elements which are randomly distributed, it require more time and more space.

10.4.3 Radix Sort

In this sort we will create a set of buckets and distribute the elements based on a key digit. After distributing all the elements, we collect them from the first bucket. Then the same procedure is followed for the further key digits. Initially the key digit is the least significant digit, then tens, hundreds and so on.

The unsorted array is:

423 115 342 387 17 882 931 431 283 95

Pass 1: 93**1** 43**1** 34**2** 88**2** 42**3** 28**3** 11**5** **9**5 387 **1**7

Pass 2: 1**1**5 **1**7 4**2**3 9**3**1 4**3**1 3**4**2 8**8**2 2**8**3 3**8**7 **9**5

Pass 3: **0**17 **0**95 **1**15 **2**83 **3**42 **3**87 **4**23 **4**31 **8**82 **9**31

```
void radix(int arr[],int n)
{
int k,data,i,j,d,f=0,exp=1,front[10],rear[10];
struct
{
      int data,next;
}node[size];
for(i=0;i<n;i++)
{     node[i].data=arr[i];
      node[i].next=i+1;
}
node[n-1].next=-1;
for(d=1;d<=5;d++)
{     for(i=0;i<10;i++)
      front[i]=rear[i]=-1;
      while(f!=-1)
      {     data=(node[f].data/exp)%10;
            if(rear[data]==-1)
            front[data]=rear[data]=f;
            else
            {     node[rear[data]].next=f;
                  rear[data]=f;
            }
            f=node[f].next;
      }
      exp*=10;
      for(i=0;i<10&&front[i]==-1;i++);
      f=front[i];
      j=i;
      while(j<10)
      {     for(k=j+1;k<10&&front[k]==-1;k++);
            if(k<10)
            node[rear[j]].next=front[k];
```

```
            else
            node[rear[j]].next=-1;
            j=k;
        }
}for(i=0;i<n;i++)
      {     arr[i]=node[f].data;
            f=node[f].next;
      }
}
```

Analysis of Radix sort:

Each key is looked at m number of times where 'm' is the maximum number of digits. Therefore the total time required in m*n where n is the number of elements.
When we compare m and n, n is much larger than the m. Because we may have 2000 elements but the digits may be 10 or 20. Therefore we conclude that the time required to sort the array is

$$T(n) = m*n$$
$$=O(n)$$

All the cases requires same time. Even though the time complexity is O(n), it requires extra memory space of minimum size of the list.

11. Graph

Defn: A graph G consists of a set V, whose members are called as vertices of G, together with a set of pairs of distinct edges (E) Hence graph G=(V,E). If *e=(v,w)* an edge with vertices *v*, *w*, then *v* and *w* are said to lie on *e* and *e* is said to be incident with *v* and *w*.

Terminology:

- A *graph* is a data structure that consists of set of *nodes* (vertices) and set of *edges* (arcs).
- If the pair of nodes that makes up the arcs are ordered pair then its known as *directed graphs* or *digraphs* .
- A node n is an incident of the *arc* x, only if n is one of the pair of x.
- A degree of a node is number of edges connected to it. The number of edges that has n as head is known as *indegree.* The number of arcs that has n as tail is known as *out degree.*
- A number associated with an edge is known as *weight*. A graph that has a weight is known as *weighted graph* or *network.*
- In an undirected graph, if there is a path from every vertex to other vertex is known as *connected*. If it is a directed graph then is known as *strongly connected*.
- A *complete graph* is a graph in which there is an edge between every pair of vertices. The maximum number of edges in an undirected graph of n vertices is n(n-1)/2. An n vertices graph with exactly n(n-1)/2 edges is said to be *complete*. In case of directed graph on n vertices the maximum number of edges in n(n-1).
- A node n is adjacent to m if there is an edge from m to n.
- A *path length* k from a node *a* to *b* is defined as a sequence of k+1 nodes $n_1,n_2,..n_{k+1}$ such that n_1=a, n_{k+1} =b.
- A path from a node to itself is called a *cycle*. If a graph contains a cycle, it is *cyclic*. Otherwise *acyclic.* A directed acyclic graph is known as *dag*.
- A sub graph (v,e) of a graph or digraph (V,E) is one that has subset of vertices and edges of the full graph.

11.1 Representation of Graph

A graph can be represented through adjacency matrix or through adjacency list

11.1.1 Adjacency Matrix

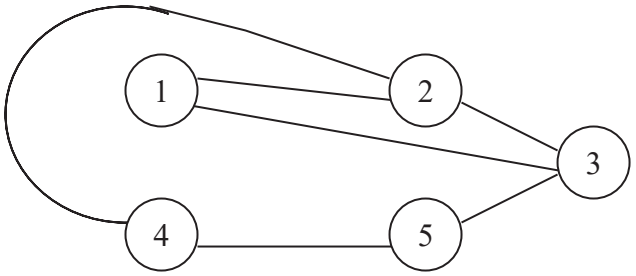
An adjacency matrix *adj* for a graph G, who has V vertices and E edges, is represented in a two dimensional array of size N x N. Each location of [i,j] of this array will store 1, if there is an edge from node v_i to v_j exists . If not it stores 0.

$$adj[i][j] = \begin{cases} 1 \text{ if } v_iv_j \in E \\ 0 \text{ if } v_ivj \notin E \end{cases}$$

For a given digraph G= (V,E), the adjacency matrix depends upon the ordering of elements of V. For different ordering of V, we get different adjacency matrices of the same graph G. However any one of the adjacency matrices of G can be obtained from another adjacency matrix by simply interchanging the rows and columns.

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

Adjacency matrix

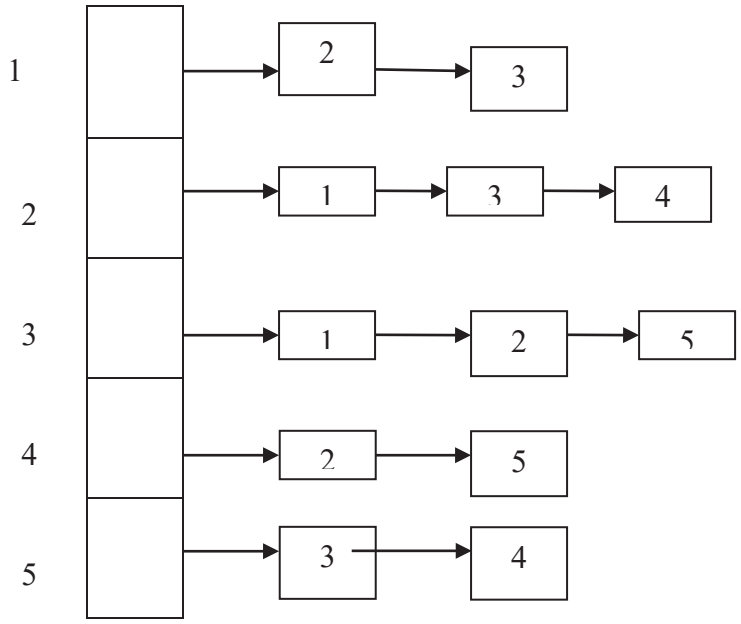


List

- If a graph has only nodes and does not have any edges then, the adjacency matrix has all its elements 0. This matrix is known as *Null matrix*.
- If there are loops at each node but no other edges in the graph, then the adjacency matrix is known as *identity* or the *unit matrix.*
- Since the elements of the adjacency matrix are either 0 or 1, it is also called as *bit matrix* or *Boolean matrix*.

11.1.2 Adjacency List

An adjacency list *adj* for a graph G=(V,E) with |v| =N will be stored in a one dimensional array of size N, with each location being a pointer to a linked list. There will be one list for each node and that list will have one entry for each adjacent node. For a weighted graphs and digraphs, the adjacency list entries would have an additional field to hold the weight of the edge.



Adjacency list for the above graph.

Advantages and disadvantages

1. An adjacency matrix gives us the ability to access the information quickly. Whereas adjacency list uses space that is proportional to the number of edges in the graph., but the time to access the edge information may be greater.
2. In adjacency matrix, if the graph is not complete then there will be many empty locations, whereas in adjacency list we create a node only where there is a relation between two vertices. Therefore if there are many vertices, then it is better to use adjacency list. If less vertices adjacency matrix is a better option.

11.2 Path Matrix or Transitive Closure

A path matrix is a matrix that simply shows the presence or absence of at least one path between a pair of nodes and also the presence or absence of a cycle at any node.

Let G= {V,E) be a simple diagraph which contains N nodes that are assumed to be ordered. A N xN matrix whose elements are given by

$$\text{Path}[i][j] = \begin{cases} 1 & \text{if there is a path from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

is called as *path matrix or reachability matrix*.

The path matrix 'path' of a graph G is obtained by adding the adjacency matrix of path length from 1 to n.

The original adjacency matrix is itself gives all the pairs of nodes connected with 1 edge. This is known as matrix of path length 1. To obtain matrix of path length 2 we need to multiply *adj* matrix with itself with the Boolean matrix multiplication

```
Adj2= adj1&&adj1.
```

To obtain path length 3

```
Adj3=adj1&&adj2
```

To obtain path length 4

```
Adj4 = aadj1&&adj3
```

Hence to obtain path length *n*

```
Adjn=adj1&&adj(n-1).
```

The path matrix of a simple digraph is

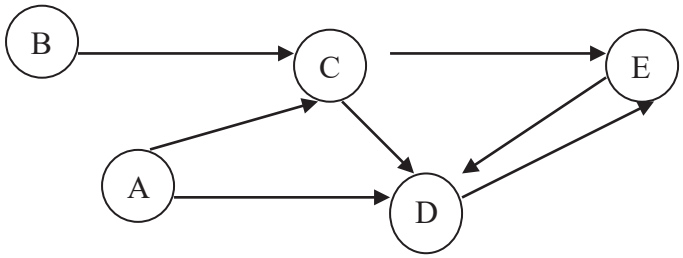
Path[i][j] = adj1[i][j] || adj2[i][j]+ adj3[i][j]+…….adjn[i][j]

```
#define MAX 5
void boolMultiply (int a[] [MAX], int b[][MAX], int c[] [MAX])
{ int i,j,k;
      for(i=0; i< MAX; i++)
      {
            for(j=0;j<MAX;j++)
```

```
            {
                  c[i][j]=0;
                  for(k=0; k< MAX; k++)
                  c[i][j]= c[i][j] || (a[i][k] &&b[k][j]);
            }
      }
}


void transclose (int adj [][MAX], int path[][MAX])
{     int i,j,k;
      int adjprod[MAX][MAX],newprod[MAX][MAX];
      for(i=0; i< MAX; i++)
            for(j=0; j< MAX; j++)
            adjprod[i][j]=path[i][j]=adj[i][j];

      for(i=1; i< MAX; i++)
      {
            boolMultiply(adjprod,adj, newprod);

            for(j=0; j< MAX; j++)
            {     for(k=0; k< MAX; k++)
                  path[j][k]=path[j][k] ||newprod[j][k];
            }
            for(j=0; j< MAX; j++)
            {
                  for(k=0; k< MAX; k++)
                  adjprod[j][k]=newprod[j][k];
            }
      }
}
void main()
{
      int adj[MAX][MAX]={ {0,0,1,1,0},
                        {0,0,1,0,0},
                        {0,0,0,1,1},
                        {0,0,0,0,1},
                        {0,0,0,1,0}
                  };
      int  path[MAX][MAX];
      int i,j;
      transclose(adj,path);
      for(i=0;i<MAX;i++)
      {
        for(j=0;j<MAX;j++)
        {
          printf("%d\t", path[i][j]);
        }
        printf("\n");  }}
```

Efficiency Consideration: The efficiency of b*oolMultiply* function is $O(n^3)$. In the *transclose* function *boolMultiply* is embedded in a loop which is repeated n-1 times, so the efficiency of *transitive* closure procedure is $0(n^4)$.

Give an adjacency & Path matrix for the given graph below.



		Adj1 0 0 1 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0
Adj1 0 0 1 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0	Adj1 0 0 1 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0	Adj2 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 0 0 0 0 0 1
Adj1 0 0 1 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0	Adj2 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 0 0 0 0 0 1	Adj3 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0
Adj1 0 0 1 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0	Adj3 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0	Adj4 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 0 0 0 0 0 1
Adj1 0 0 1 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0	Adj4 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 0 0 0 0 0 1	Adj5 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0
Path==Adj1||Adj2||Adj3 ||Adj4||Adj5		0 0 1 1 1 0 0 1 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1

11.2.1 Warshall Algorithm.

The previous method to calculate the transitive closure is inefficient, because the run time is $O(n^4)$. Therefore, Warshall has designed an algorithm that reduces the runtime to $O(n^3)$.

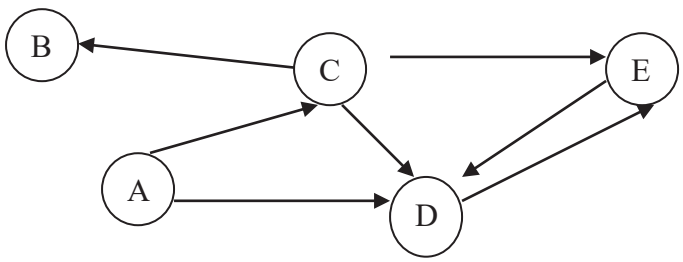
In order to find the presence of a path between node v_i to v_k, Find whether there is a path existing between v_i and v_j where j<k. If yes, check whether any path exists between v_{j+1} and v_k. if yes, then conclude that there is path exists between v_i and v_k. If not, no path exist between v_i and v_k through v_j.

```
void warshall (int adj[] [MAX], int path[][MAX])
{
  int i,j,k;
      for(i=0; i< MAX; i++)
      for(j=0; j< MAX; j++)
            path[i][j]=adj[i][j];

      for(j=0; j< MAX; j++)
      {
            for(i=0; i< MAX; i++)
            {     if (path[i][j]==1)
                  {     for(k=0;k<MAX;k++)
                        path[i][k] = path[i][k] || path[j][k];
                  }
            }
      }
}
```

11.3 Graph Traversal:

Visiting all the nodes of a graph once is called as traversal. A graph can be traversed in two methods

1. Breadth First Traversal (BFS)
2. Depth First Traversal (DFS)



11.3.2 Breadth First Traversal

In breadth first search, we go evenly in many directions. Initially we visit the starting node and then on the first pass, we visit all the nodes directly connected to it. In the second we visit all the unvisited nodes connected with the staring node with two edges. The third pass visits all the unvisited nodes connected with the staring node with three edges and so on. This procedure is continued until all the nodes are visited.

To implement the BFS algorithm, we use a queue. Whenever a node is visited, we place it in the queue. Then we remove the front element from the queue and visit all the node connected with the node. This procedure is continued until the rear and front becomes equal.

BFS Traversal: ACDBE

```
BFS(G, V)

G is the graph represented as an adjacency matrix or list
V is the initial node from where the process begins

   1. Visit  V
   2. insertQueue (V)
   3. Loop( queue is non empty)
         1. x= removequeue()
         2. Loop(till the x the row)
            1. if( a node is not visited)
            2. insertqueue(v)
    4. end
```

```
void bfs(int adj[MAX][MAX],int start)
{
      int i;
      int visited[MAX]={0};
      int q[30],first=-1,last=-1;
      visited[start]=1;
      q[++last]=start;
      while(first!=last)
      {
            start=q[++first];
            printf("%d ",start+1);
            for(i=0;i<MAX;i++)
            if(adj[start][i]!=0 && visited[i]==0)
            {
                  q[++last]=i;
                  visited[i]=1;
            }
      }
}
```

11.3.2 Depth First Traversal

In depth first traversal we will go as far as possible in one path until we reach the dead end. In an undirected graph, a node is a dead end if all of the nodes adjacent to it is already been visited. In directed graph, we have two dead ends, ie) a node that does not have a successor and a node that is already been visited. Once we reach the dead end we come one step behind

and visit all the nodes connected to it. This process of stepping back is continued until we come back to the first node.

In order to implement this algorithm we use a stack. Whenever we visit a node, we place it in the stack. Once we reach the dead end, we pop the stack and traverse all the unvisited node connected to it.

DFS Traversal : A C B D E

Algorithm:

```
DFS( G, V)
G is the graph represented in adjacency matrix
V is the vertex from where the traversal begins

      1. Vist(V)
      2. Loop(compare all the node at v^th row)
         1.if any node from v is unvisited
         2. DFS(G, I)
      3. Stop.
```

Recursive DFS

```
void dfs (int adj [] [MAX], int visited [], int start )
{
            int i;
            printf("%c-",start +64);
            visited[start] = 1;
            for (i=0; i<MAX; i++)
            if (adj[start][i] &&  visited [i]==0 )
            dfs(adj, visited,i );
}
```

Non- Recursive DFS

```
void dfs (int adj [] [MAX], int visited [], int start )
{               int stack[10];
            int top=-1;
            int i;
            printf("%c-",start +65);
            visited[start] = 1;
            stack[++top]=start;
            while(top!=-1)
            {
                start=stack[top--];
                for (i=0; i<MAX; i++)
                {
                  if (adj[start][i] &&  visited [i]==0 )
                  {  stack[++top]=i;
                     printf("%c-",i +65);
                     visited[i] = 1;
                  }
                }
      }
```

```
}
void main()
{     int adj[MAX][MAX]={ {0,0,1,1,0},
                          {0,0,0,0,0},
                          {0,1,0,1,1},
                          {0,0,0,0,1},
                          {0,0,0,1,0}
                        };
      int  visited[MAX]={0};
      dfs(adj,visited,0);

}
```

DFS Traversal: ACBDE

11.3.3 Applications of DFS:

1. Traversing Directed Graphs
2. Traversing Undirected Graphs
3. Euler's Circuits / Tour
4. Finding the strong components of a graph
5. Biconnectivity

11.3.3.3 Euler's Circuits

Visiting all the edges only once in a sequential manner without a break or without using a traversed edge is known as *Euler's circuits or Euler's Tour/path.* This concept is depicted in the form of a famous puzzle that is to reconstruct a figure without lifting the pen and without using the same line. The mathematician Euler solved this problem.

Necessary Conditions for a Euler's path

- It must end on its starting vertex if the graph is connected and each vertex has even degree.
- If exactly two vertices has an odd degree, the tour need not end with starting vertex. It may start from one odd degree vertex end it with other
- If there are more than two odd degree vertexes, the Euler's path is not possible.

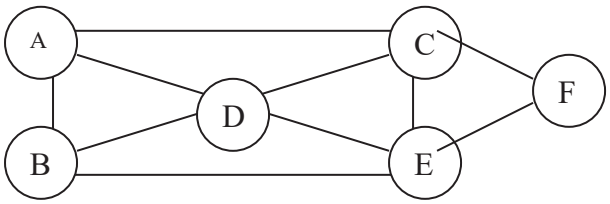
To carry out the procedure,

1. Find a path that is strongly connected of a graph G from the first vertex by making a depth first search
2. Select the first vertex from the path and try to form a cyclic path from the remaining untraversed edges from the vertx. This gives an another circuit. Splice the path into the original path.
3. The second step is continued until all the edges are traversed

To implement this algorithm, we use an adjacency list. To avoid repetitious scanning of adjacency list, a pointer to the last edge is scanned. When a path is spliced in , a search for a new vertex from which the next depth first search must begin at the start of the splice point.

The total search time of the algorithm is O(|E|)

The total running time is O(|E| + |V|) where E is the total number of edges and V is the total number of vertex.



ACDA ——————— 1

CEFC ——————— 2

EBDE ——————— 3

Splicing 2 in 1 instead of C we get

ACEFCDA ——————— 4

Splicing 3 in 4 instead of E, we get

ACEBDEFCDA ——————— 5

Connect the odd degree vertex B

ACEBDEFCDAB

This is the Euler's path for the above given graph.

11.3.3.4 Finding Strong Components

By performing two depth-first searches, we can test whether a directed graph is strongly connected, and if it is not, we can actually produce the subsets of vertices that are strongly connected to themselves.

First, a depth-first search is performed on the input graph G. the vertices of G are numbered by a postorder traversal of the depth-first spanning forest, and then all edges in G are reversed, forming G_r.

11.3.3.5 Biconectivity:

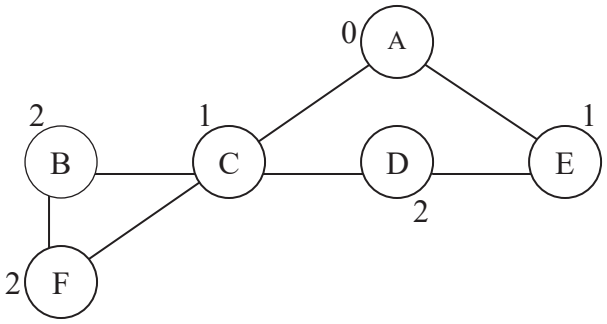
A connected undirected graph is biconnected, if there are no vertices whose removal disconnects the rest of the graph. This concept is used in the network of computers where when one computer goes down, the entire system does not get affected by that. If a graph is not biconnected, the vertex whose removal causes the disconnection of the graph is known as *articulation point*. Depth first search provides a linear time algorithm to find all the articulation points in a graph.

11.4 Spanning Tree:

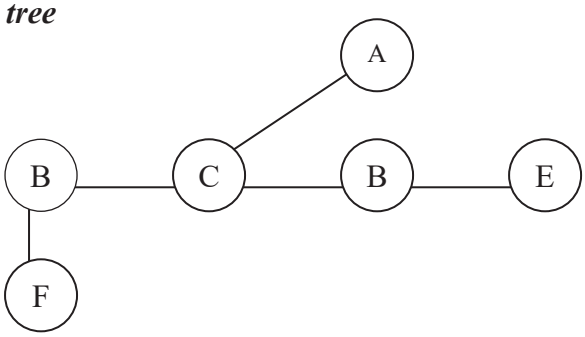
A spanning tree of a graph is an undirected tree consisting of only those edges necessary to connect all the nodes in the original graph. A spanning tree has the properties that for any pair of nodes there exists only one path between them, and the insertion of any edge to a spanning tree forms a unique cycle.

A spanning tree for a graph can be generated through

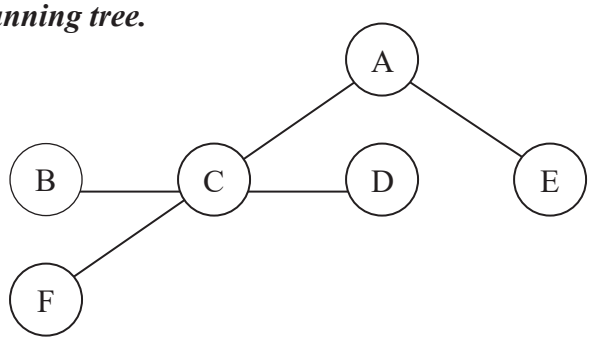
a) If Depth first search algorithm is used to form the edges of the tree, It is referred as a ***depth first spanning tree.***
b) If a breadth first search is used to form the edges of the tree then ***breadth first spanning tree***.



DFS spanning tree



BFS spanning tree.



When determining the cost of a spanning tree of a weighted graph, the cost is simply the sum of the weights of the tree's edges. A minimal cost-spanning tree is formed when the edges are picked to minimize the total cost.

11.4.1 Minimum – Cost Spanning Trees

The minimum spanning tree (MST) of a weighted connected graph is a sub-graph that contains all of the nodes of the original and a subset of the edges so that the subgraph is connected and the total of the edge weights is the smallest possible. If the original graph is not connected, the processes below can be used on each of the separate components to produce a spanning tree for each one.

The problem is to select n –1 edges from a weighted graph G such a way that the selected edges form a least-cost spanning tree of G. We can formulate at least three different greedy strategies to select these n – 1 edges.

These strategies result in three greedy algorithms for the minimum-cost spanning tree problem: *Kruskal's* algorithm, *Prim's algorithm* and sollin's algorithm. We will see the first two.

11.4.1.1 Kruskal's Algorithm:

The Method: Kruskal's algorithm selects the n-1 edges one at a time using the greedy criterion:

- ***From the remaining edges, select a least-cost edges that does not form a cycle with the already selected edges.***

Kruskal's algorithm has up to *e* stages where *e* is the number of edges in the network. The *e* edges are considered in order of increasing cost, one edge per stage. When an edge is considered, it is rejected, if it forms a cycle when it is added to the set of already selected edges. Otherwise, it is accepted.

Algorithm Kruskal (cost, N, preced)

Cost	**:**	Cost Matrix
N	**:**	Number of vertex in the graph
Preced	**:**	array that stores the previous node to the node in the minimum spanning tree.
Post Condition:		The final cost is returned.

1. sort the edges in nondecreasing order by weight
2. initialize partition structure
3. MinCost=0
4. edgeCount = 1
5. includedCount = 0
6. while (edgeCount ≤ E and includedCount ≤ N - 1)

    1. parent1 = FindRoot (edge [edgeCount] . start)
    2. parent2 = FindRoot (edge [edgeCount] . end)
    3. if parent1 ≠ parent2 then

        1. *add edge [edgeCount] to spanning tree*
        2. *includedCount = includedCount + 1*
        3. *union (parent1, Parent2)*
        4. *MinCost=MinCost+ Cost[edge[edgecount].start,edge[edgecount]. end].*

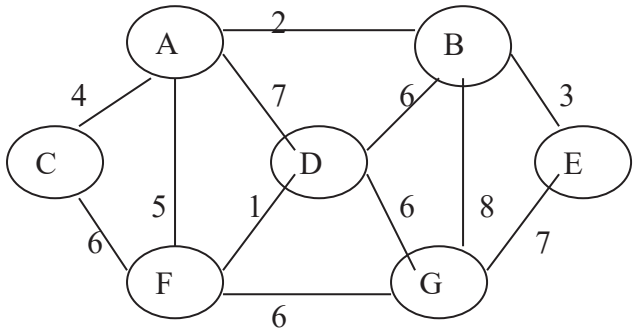
4. edgeCount = edgeCount + 1

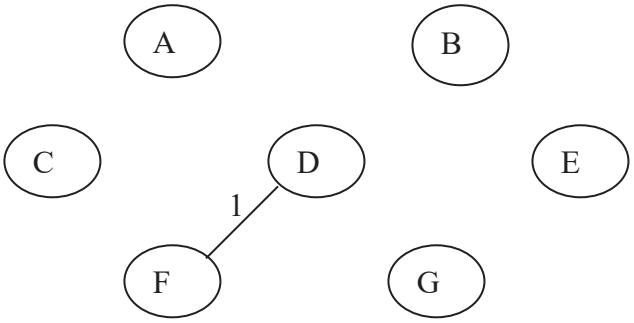
7. retrun MinCost.

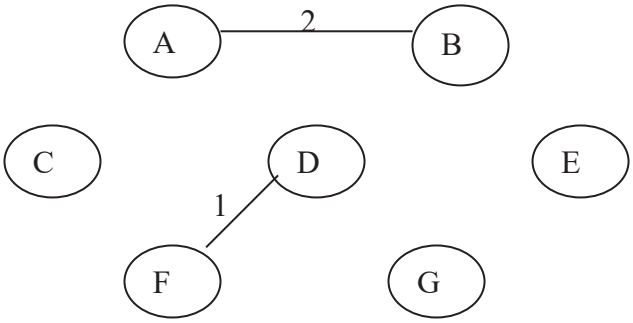
Program

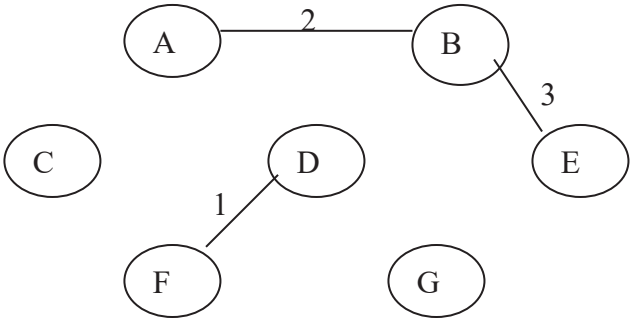
```
# define IN 100
# define MAX 7
int kruskal(int weight[][MAX])
{
        int  k,nodecount,mincost=0,parent [MAX] = {0};
         struct edge
         {       int v1, v2, wt;
         } e[20];
         int i,j,size = 0;
         for (i=0; i<MAX-1 ; i++ )    //Sort the edges
         {
                 for(j=i+1; j<MAX; j++ )
                 {
                         if(weight[i][j]!=IN)
                         {   e[size].v1 = i;
                            e[size].v2 =j;
                            e[size].wt = weight [i] [j];
                            size++;
                         }
                  }
         }
        for (i = 0; i < size - 1; i++)
        {       for (j=i+1;j<size; j++)
                {
                         if(e[j].wt<e[i].wt)
                         {   struct edge temp = e[i];
                                 e[i] =e[j];
                                 e[j] = temp;
                         }
                }
        }
        nodecount = 1;
        for (k = 0; nodecount < MAX ; k++)
        { i = e[k].v1 ; j = e[k].v2;
        while (parent[i]>0)
        i = parent[i];
        while (parent[j]>0)
        j = parent[i];
        if (i!=j)
            {parent[j] = i;
             printf("%c-%c\t%d\n", e[k].v1+65, e[k].v2+65,e[k].wt);
                 mincost=mincost+e[k].wt;
                 nodecount++;
                 }
              }
```

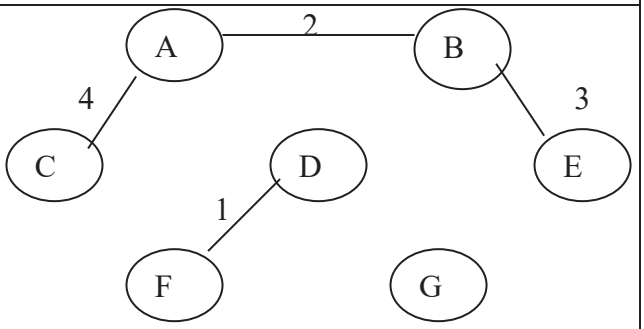
```
        return mincost;
}
```

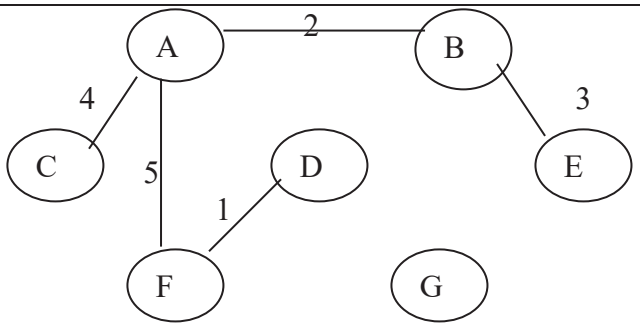


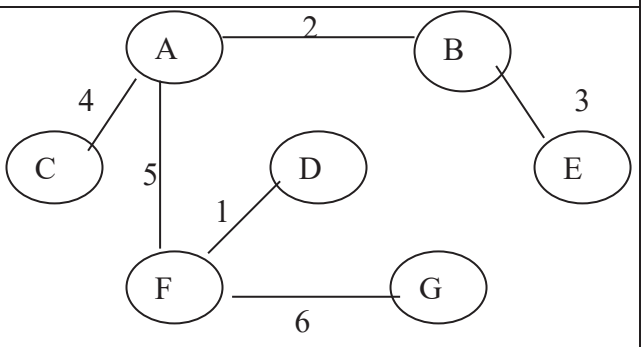












Minimum Cost =21

11.4.1.2 The Prim's algorithm

This method builds up a minimum cost spanning tree by picking an edge one at a time using the following greedy criterion.

From the selected edges, choose a least cost edge whose addition to the already chosen edges forms a tree.

Algorithm Prim (cost, N, preced)

Cost : Cost Matrix
N : Number of vertex in the graph
Preced: array that stores the previous node to the node in the minimum spanning tree.
Post Condition: The final cost is returned.

1. Initialize the distance[] to infinity
2. Select a starting node 'current'
3. distance[current]=0;
4. Mincost=0;
5. While(all the nodes are not selected)
    1. Consider the smallest distance INFINITY 'small=INFINITY'
    2. for(I=1;I<=N;I++)
        1. Mindistance = Find(cost[current][i])
    3. Current=MindistanceVertex
    4. Mincost=Mincost+Mindistance
6. Return Mincost

Program

```
int prim (int weight [][MAX], int preced[] )
{     int perm[MAX]={0},distance[max];
      int current, i, k, smalldist, newdist,minimumcost=0;
      perm[0]=1; current = 0;

      for(i=0;i<MAX;i++)
      distance[i]=IN;
      distance[current]=0;
      preced[current]=0;
      while ( allvisited(perm)==0)
      {     smalldist=IN;
            for (i=0; i<MAX; i++)
            {           if (perm[i]==0)
                  {     newdist=weight[current][i];
                         if (newdist<distance[i] )
                         { distance[i]=newdist;
                           preced[i]=current;
                         }
                        if (distance[i] < smalldist)
                        {  smalldist = distance[i];
                           k= i;
                        }
                  }
            }
            current = k;
```

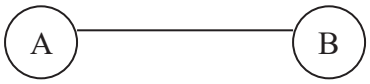
```
            perm[current] = 1;
            minimumcost=minimumcost+distance[current];

        }

return minimumcost;
}
void main()
{
      int cost[MAX][MAX]= { {IN,2,4,7,IN,5,IN},
                            {2,IN,IN,6,3,IN,8},
                            {4,IN,IN,IN,IN,6,IN},
                            {7,6,IN,IN,IN,1,6},
                            {IN,3,IN,IN,IN,IN,7},
                            {5,IN,6,1,IN,IN,6},
                            {IN,8,IN,6,7,6,IN}
                           };
      int min,i,preced[MAX]={0}, distance[MAX];
      min=prim(cost,preced);
      printf("Minimum Cost \t%d\n\n",min);
      for(i=0;i<MAX;i++)
      printf("%c\n",preced[i]+65);

}
```

Solution:

From A

~~0~~	2	4	7	IN	5	IN
A	B	C	D	E	F	G

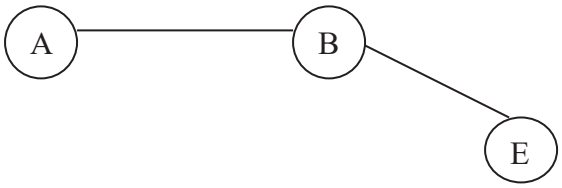
Cost=0



From B

~~0~~	~~2~~	4	6	3	5	8
A	B	C	D	E	F	G

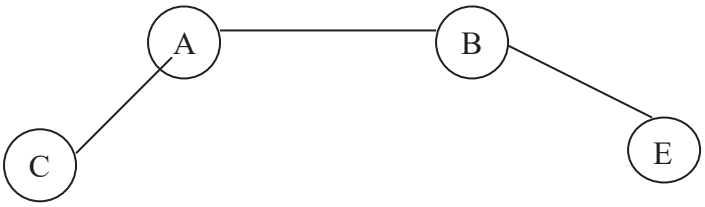
Cost =2



From E

0	~~2~~	4	6	~~3~~	5	7
A	B	C	D	E	F	G

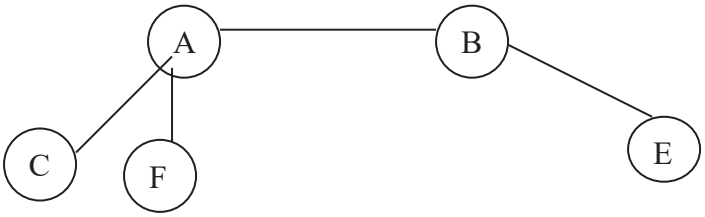
Cost =2 +3 =5



From C

0	2	4	6	3	5	7
A	B	C	D	E	F	G

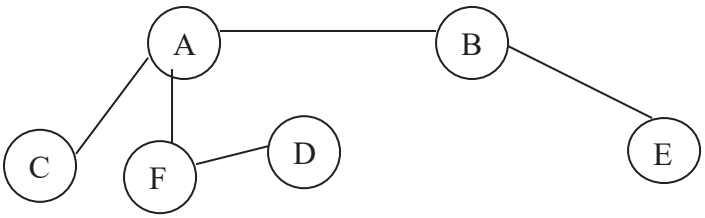
Cost =5+4 =9



From F

0	2	4	1	3	5	6
A	B	C	D	E	F	G

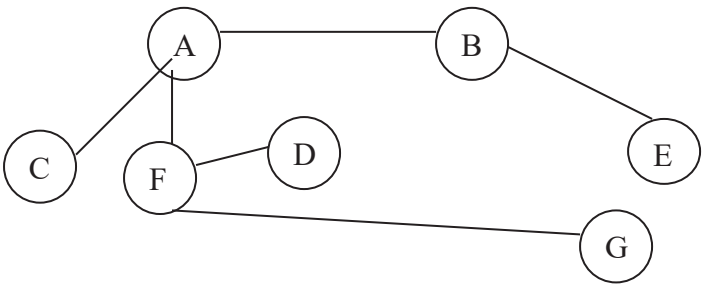
Cost =9+5 =14



From D

0	2	4	1	3	5	6
A	B	C	D	E	F	G

Cost = 14+1=15



Cost =15+6.

The Minimum Cost is 21.

11.5 Single Source Shortest Paths

A graph can be used to find a shortest path between two destinations. In order to find a shortest path we use ***Dijkstra's algorithm.*** It uses the following greedy criterion

From the remaining vertices, select a vertex that has a shortest path from the initial vertex.

Algorithm ShortestPath (cost, start, preced, distance)

Cost : Cost Matrix

start : Staring vertex in the graph from where the distance needs to be calculated.

Preced: array that stores the previous node to the node in the minimum spanning tree.

Distance: Stores the shortest distance between 'start' node and the i^{th} node

Post Condition: The distance array is returned with the shortest distance to each node.

1. Initialize the distance[] to infinity
2. Initialize the selected array to FALSE selected[]=0
3. current=start
4. distance[current]=0
5. selected[current]=TRUE
6. While(all the nodes are not selected)
   1. Consider the smallest distance INFINITY 'small=INFINITY'
   2. currentdistance= distance[current]
   3. for(I=1;I<=N;I++)
      1. distance[I]=currentdistance+ cost[current][I]
      2.Mindistance = Find(distance[I])
   4. Current=MindistanceVertex
   5. selected[current]=TRUE

7.end

Program ShortestPath:

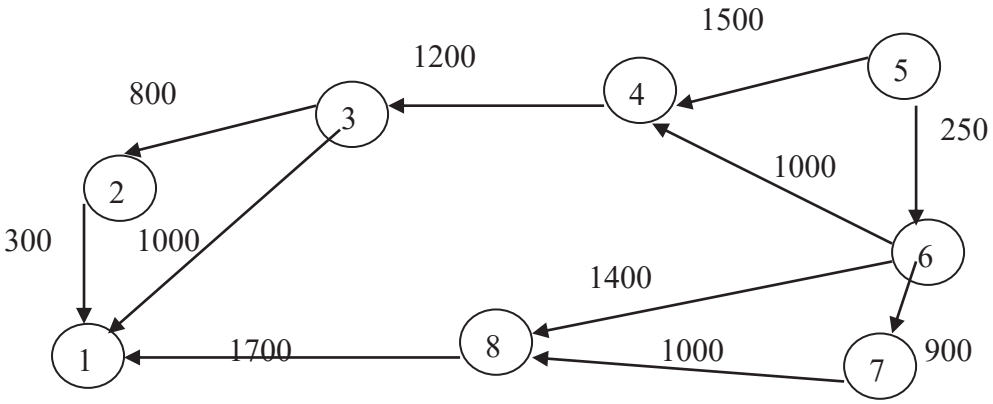
```
int allselected (int selected[])
{int i;
for (i=0; i<MAX; i++)
if (selected[i]==0)
return 0;
return 1;}


void shortpath (int cost[][ MAX], int preced [], int distance[] )
{
int selected[MAX] = {0};
int current=0, i, k, dc, smalldist, newdist;
for(i=0;i<MAX;i++)
```

```
                        {4,IN,IN,IN,IN,6,IN},
                        {7,6,IN,IN,IN,1,6},
                        {IN,3,IN,IN,IN,IN,7},
                        {5,IN,6,1,IN,IN,6},
                        {IN,8,IN,6,7,6,IN}
                       };
int i,preced[MAX]={0}, distance[MAX];
shortpath(cost,preced,distance);
for(i=0;i<MAX;i++)
printf("%d\n",distance[i]); }
```

Time Complexity: Any shortest path algorithm must examine each edge in the graph atleast once, since any of the edges could be in a shortest path. Hence minimum possible time for such an algorithm would be Ω(|E|) where E is the total number of edges. Since cost adjacency matrix is used to represent the graph , it takes $O(n^2)$ time to determine which edges are in G.

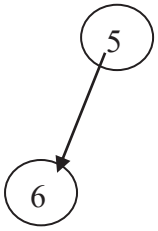
Problem:

Find a shortest Path from Node 5 to all other nodes



From 5
Distance[I] = 0+ ..

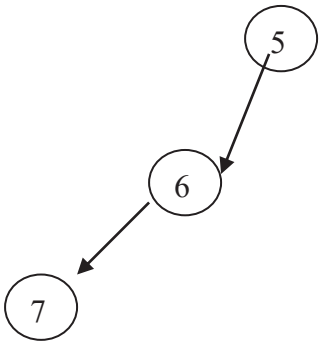
Ά	ά	ά	1500	0	250	ά	ά
1	2	3	4	5	6	7	8



From 6: Path to 6 ={5, 6 } = 250

Distance [I]=250+

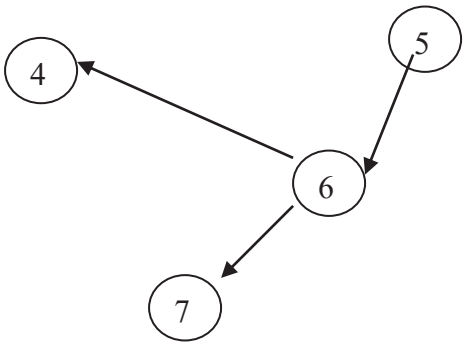
ά	ά	ά	1250	0	250	1150	1650
1	2	3	4	5	6	7	8



From 7 path to 7 ={ 5,6,7} =1150

Distance [I]=1150+

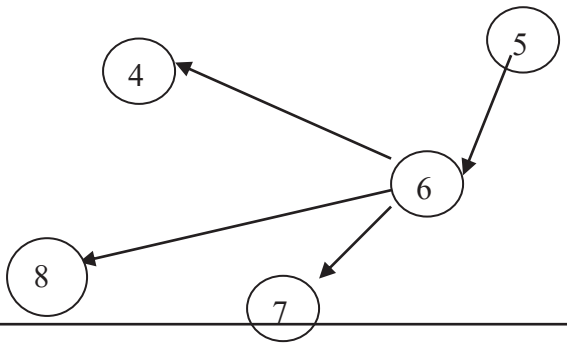
ά	ά	ά	1250	0	250	1150	1650
1	2	3	4	5	6	7	8



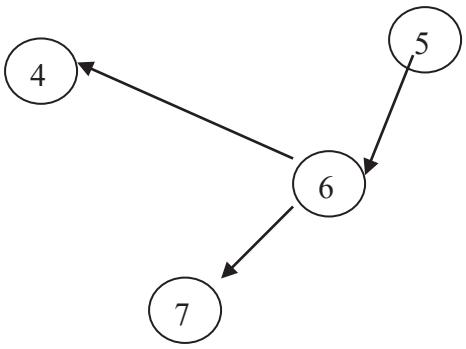
From 4 path to 4 ={ 5,6,4} = 1250

Distance [I]=1250+

ά	ά	2450	1250	0	250	1150	1650
1	2	3	4	5	6	7	8



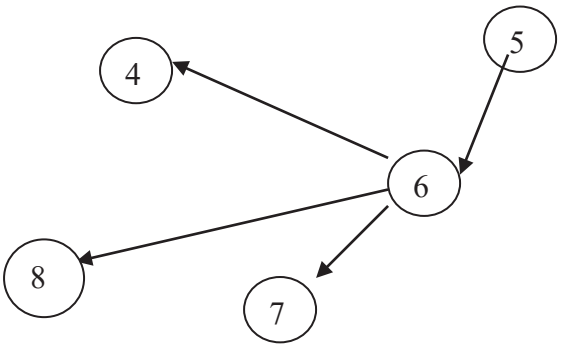
ά	ά	ά	1250	0	250	1150	1650
1	2	3	4	5	6	7	8



From 4 path to 4 ={ 5,6,4} = 1250

Distance [I]=1250+

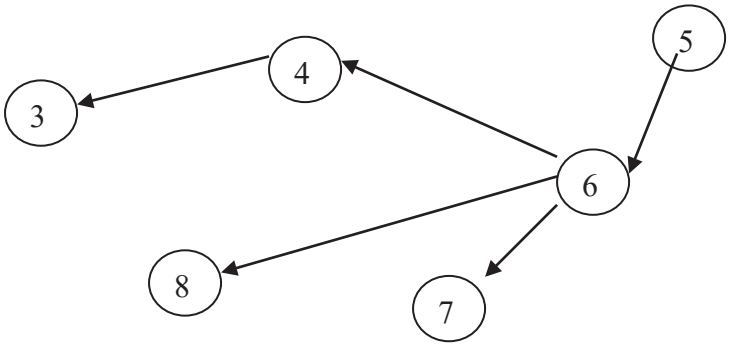
ά	ά	2450	1250	0	250	1150	1650
1	2	3	4	5	6	7	8



From 8 path to 8 ={ 5,6,8} = 1650

Distance [I]=1650+

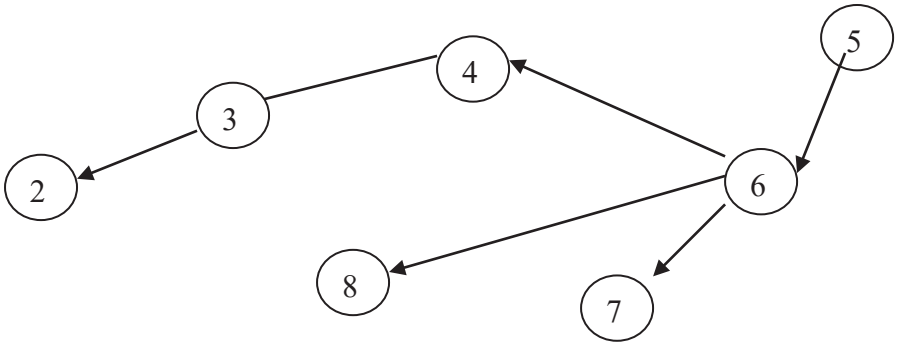
3350	Ά	2450	1250	0	250	1150	1650
1	2	3	4	5	6	7	8



From 3 path to 4 ={ 5,6,4,3} = 2450

Distance [I]=2450+

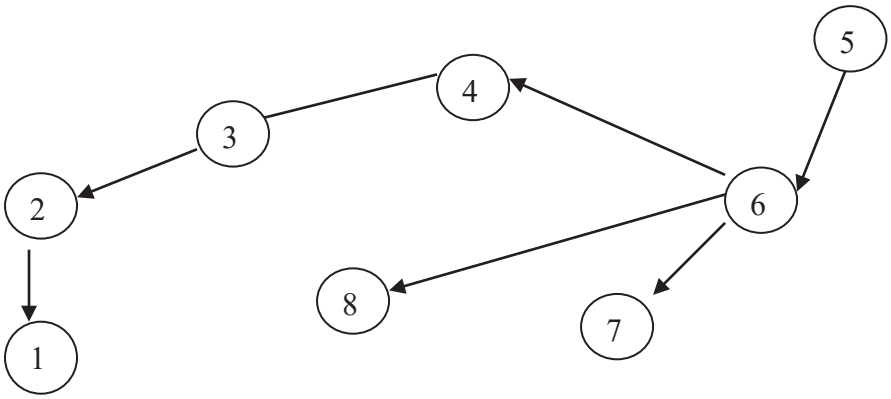
3350	3250	~~2450~~	1250	~~0~~	250	~~1150~~	~~1650~~
1	2	3	4	5	6	7	8



Path to 2 ={ 5,6,4,3,2}= 3250

From 2

3350	~~3250~~	~~2450~~	1250	~~0~~	250	~~1150~~	~~1650~~
1	**2**	**3**					



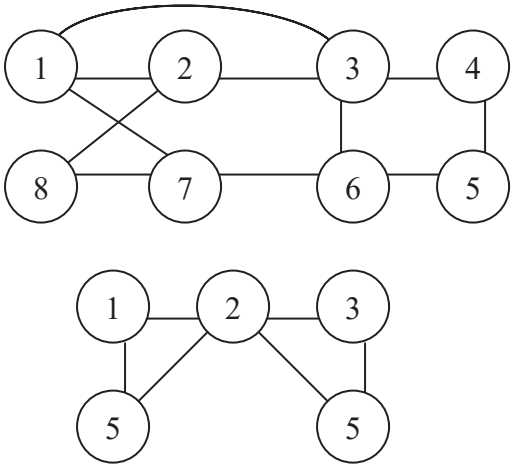
Path to 1 ={ 5,6,8,1}= 3350

Note: If you were asked to find a path between two specific nodes start from one until you cancel the target node continue the procedure.

In the Program, instead of while(! Allselected(selected)) use the condition while(current!=target). The function should receive start and target.

HAMINLTONIAN CYCLES

In *Euler* circuits we visited all the EDGES once in sequential order without a break. Whereas Hamiltonian cycle is a trip path along *n* edges of G which visits every NODE once and returns to its starting position. In other words if a Hamiltonian cycle begins at some vertex V_1 E G and the vertices of G are visited in the order $V_1, V_2, \ldots$. Vn +1 then the edges (Vi, Vi + 1) are in E, $1 \leq i \leq n$ and the vi are distinct except for v1 and Vn+1 which are equal.



In the above two graphs, first one contains a Hamiltonian cycle (Path :1 2 8 7 6 5 4 3 1)

The backtracking solution vector ($x_1 \ldots\ldots, x_n$) is defined so that x_i represents the i^{th} visited vertex of the proposed cycles. Now all we need to do is determine how to compute the set of possible vertices for x_k if $x_1, \ldots.., x_{k-1}$ have already been chosen. If k=1 then X(1) can be any one of the n vertices. In order to avoid the printing of the same cycle n times we require that X(1) – 1 If 1 < k < n then n X(k) can be

This procedure is started by first initializing the adjacency matrix GRAPH (1:n, 1:n), then setting X(2:n) ← 0, X(1) ← 1 and then executing call HANILTONIAN(2).

Recall from section 5.8 the traveling salesperson problem which asked for a “tour” which has minimum cost, this tour is a Hamiltonian cycle. For the simple case of a graph all of whose edge costs are identical, procedure HAMILTONIAN will find a minimum cost tour if a tour exists. If the common edge cost is c, the cost of a tour is *cn* since there are n edges in a Hamiltonian cycle.