

# Data Structure & Algorithms

Prepared by Bal Krishna Subedi

## What is data structure?

- Data structure is a way of organizing all data items and establishing relationship among those data items.
- Data structures are the building blocks of a program.
  - Data structure mainly specifies the following four things:
    - Organization of data.
    - Accessing methods
    - Degree of associativity
    - Processing alternatives for information

To develop a program or an algorithm, we should select an appropriate data structure for that algorithm. Therefore algorithm and its associated data structures form a program.

### Algorithm + Data structure = Program

A **static data structure** is one whose capacity is fixed at creation. For example, array. A **dynamic data structure** is one whose capacity is variable, so it can expand or contract at any time. For example, linked list, binary tree etc.

## Abstract Data Types (ADTs)

An **abstract data type** is a data type whose representation is hidden from, and of no concern to the application code. For example, when writing application code, we don't care how strings are represented: we just declare variables of type *String*, and manipulate them by using string operations.

Once an abstract data type has been designed, the programmer responsible for implementing that type is concerned only with choosing a suitable data structure and coding up the methods.

### Classification of data structure:

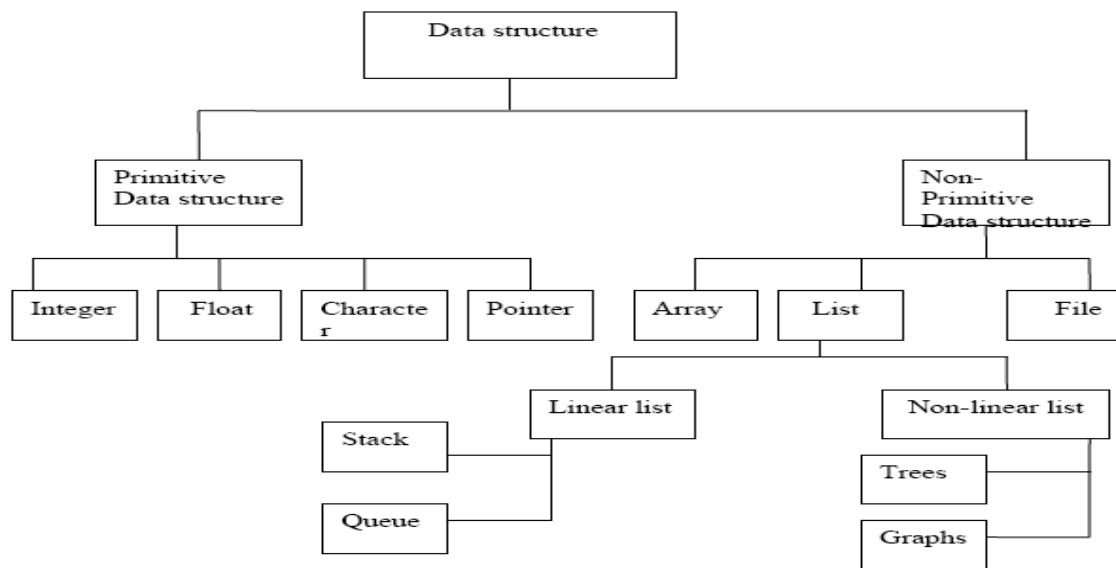


Fig:- Classification of data structure

## Array

- Array is a group of same type of variables that have common name
- Each item in the group is called an *element* of the array
- Each element is distinguished from another by an *index*
- All elements are stored contiguously in memory
- The elements of the array can be of any valid type- integers, characters, floating-point types or user-defined types

### **Types of Array:**

#### **1). One dimensional array:**

The elements of the array can be represented either as a single column or as a single row.

#### **Declaring one-dimensional array:**

```
data_type array_name[size];
```

Following are some valid array declarations:

```
int age[15];
float weight[50];
int marks[100];
```

Following are some invalid array declarations in c:

```
int value[0];
int marks[0.5];
int number[-5];
```

## **Array Initialization (1-D):**

The general format of array initialization is:

```
data_type array_name[size]={element1,element2,.....,element n};
```

**for example:**

```
int age[5]={22,33,43,24,55};  
int weight[]={55,6,77,5,45,88,96,11,44,32};  
float a[]={2,3.5,7.9,-5.9,-8};  
char section[4]={'A','B','C','D'};
```

**Example 1: A program to read n numbers and to find the sum and average of those numbers.**

```
#include<stdio.h>  
void main()  
{  
    int a[100], i, n, sum=0;  
    float avg;  
    printf("Enter number of elements");  
    scanf("%d",&n);  
    printf("Enter %d numbers",n);  
    for(i=0;i<n;i++)  
    {  
        scanf("%d",&a[i]);  
        sum=sum+a[i];  
        //sum+=a[i];  
    }  
    avg=sum/n;  
    printf("sum=%d\n Average=%f", sum, avg);  
}
```

**Some common operations performed in one-dimensional array are:**

- Creating of an array
- Inserting new element at required position
- Deletion of any element
- Modification of any element
- Traversing of an array
- Merging of arrays

## **Two-Dimensional array:**

When we declare two dimensional array, the first subscript written is for the number of rows and the second one is for the column.

### **Declaration of 2-D array:**

```
Return_type array_name[row_size][column_size];
```

#### **Example;**

```
int a[3][4];
```

```
float b[10][10];
```

int this first example, 3 represents number of rows and 4 represents number of columns.

- Think, two-dimensional arrays as tables/matrices arranged in rows and columns
- Use first subscript to specify row no and the second subscript to specify column no.

	Column 0	Column 1	Column 2	Column 3
Row 0	a[ 0 ][ 0 ]	a[ 0 ][ 1 ]	a[ 0 ][ 2 ]	a[ 0 ][ 3 ]
Row 1	a[ 1 ][ 0 ]	a[ 1 ][ 1 ]	a[ 1 ][ 2 ]	a[ 1 ][ 3 ]
Row 2	a[ 2 ][ 0 ]	a[ 2 ][ 1 ]	a[ 2 ][ 2 ]	a[ 2 ][ 3 ]

↑                    ↑                    ↑                    ↑

Array name                    Row subscript                    Column subscript

## **Array Initialization (2-D):**

The general format of array initialization is:

```
data_type array_name[row_size][col_size]={element1,element2,.....,element n};
```

**for example:**

```
int a[2][3]={33,44,23,56,77,87};
```

**or**

```
int a[2][3]={ {33,44,23},  
             {56, 77, 87} };
```

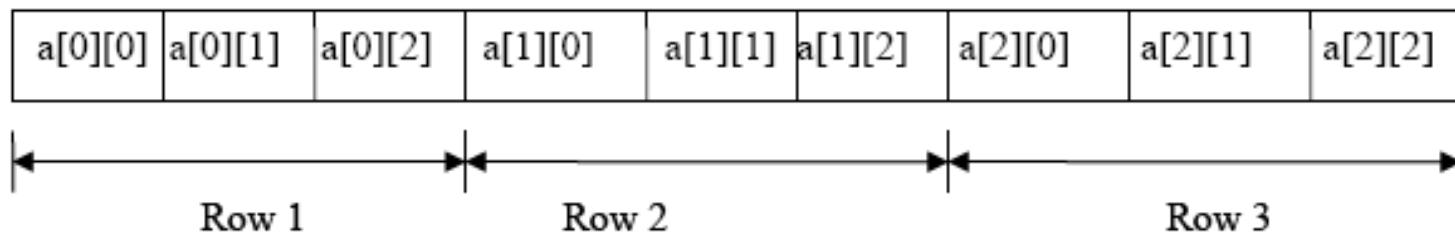
## **Implementation of a two-dimensional array:**

A two dimensional array can be implemented in a programming language in two ways:

- Row-major implementation
- Column-major implementation

### **Row-major implementation:**

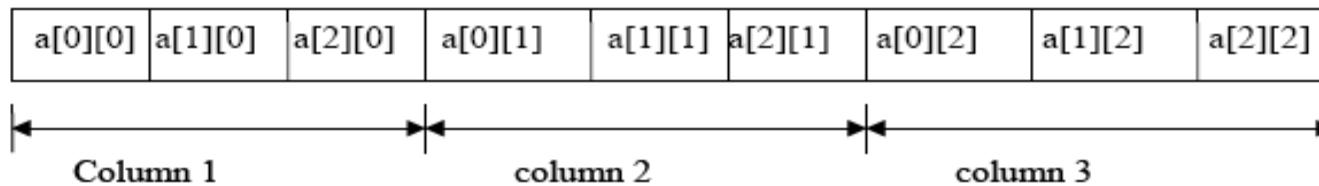
Row-major implementation is a linearization technique in which elements of array are reader from the keyboard row-wise i.e. the complete first row is stored, and then the complete second row is stored and so on. For example, an array  $a[3][3]$  is stored in the memory as shown in fig below:



### **Column-major implementation:**

In column major implementation memory allocation is done column by column i.e. at first the elements of the complete first column is stored, and then elements of

complete second column is stored, and so on. For example an array  $a[3][3]$  is stored in the memory as shown in the fig below:



### **Multi-Dimensional array**

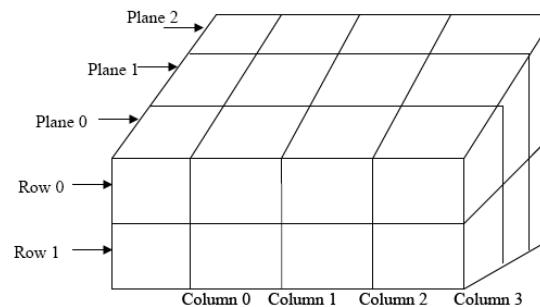
C also allows arrays with more than two dimensions. For example, a three dimensional array may be declared by

```
int a[3][2][4];
```

Here, the first subscript specifies a plane number, the second subscript a row number and the third a column number.

However C does allow an arbitrary number of dimensions. For example, a six-dimensional array may be declared by

```
int b[3][4][6][8][9][2];
```



### **Show that an array is an ADT:**

Let A be an array of type T and has n elements then it satisfied the following operations:

- CREATE(A): Create an array A
- INSERT(A,X): Insert an element X into an array A in any location
- DELETE(A,X): Delete an element X from an array A
- MODIFY(A,X,Y): modify element X by Y of an array A
- TRAVELS(A): Access all elements of an array A
- MERGE(A,B): Merging elements of A and B into a third array C

Thus by using a one-dimensional array we can perform above operations thus an array acts as an ADT.

## **Structure:**

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name.

An array is a data structure in which all the members are of the same data type. Structure is another data structure in which the individual elements can differ in type. Thus, a single structure might contain integer elements, floating-point elements and character elements. The individual structure elements are referred to as *members*.

Defining a structure: A structure is defined as

```
struct structure_name  
{  
    member 1;  
    member 2;  
    .....  
    member n;  
};
```

We can define a structure to hold the information of a student as follows:

```
struct Student  
{  
    char name[2];  
    int roll;  
    char sec;  
    float marks;  
};
```

### **Structure variable declaration:**

```
struct Student s1, s2, s3;
```

We can combine both template declaration and structure variable declaration in one statement.

Eg,

```
struct Student
{
    char name[2];
    int roll;
    char sec;
    float marks;
} s1, s2, s3;
```

### Accessing members of a structure:

There are two types of operators to access members of a structure. Which are:

- Member operator (dot operator or period operator (.))
- Structure pointer operator (->).

### Structure initialization:

Like any data type, a structure variable can be initialized as follows:

```
struct Student
{
    char name[20];
    int roll;
    char sec;
    float marks;
};
```

```
struct Student s1={"Raju", 22, 'A', 55.5};
```

The s1 is a structure variable of type Student, whose members are assigned initial values. The first member (name[20]) is assigned the string "Raju", the second member (roll) is assigned the integer value 22, the third member (sec) is assigned the character 'A', and the fourth member (marks) is assigned the float value 55.5.

**Example: program illustrates the structure in which read member elements of structure and display them.**

```
#include<stdio.h>
void main()
{
    struct Student
    {
        char name[20];
        int roll;
        char sec;
        float marks;
    };
    struct Student s1;
    clrscr();
    printf("Enter the name of a student");
    gets(s1.name);
    printf("Enter the roll number of a student");
    scanf("%d",&s1.roll);
    printf("Enter the section of a student");
    scanf("%c",&s1.sec);
    printf("Enter the marks obtained by the student");
    scanf("%f",&s1.marks);
    //displaying the records
    printf("Name=%s\n Roll number =%d\n Section=%c\n Obtained marks=%f",s1.name,
    s1.roll, s1.sec, s1.marks);
}
```

# Thank You All

Any Questions?

# Lecture 2(DSA)

Prepared by Bal Krishna Subedi

## Structures within Structures:

Structures within structures mean nesting of structures. Study the following example and understand the concepts.

**Example: the following example shows a structure definition having another structure as a member. In this example, person and students are two structures. Person is used as a member of student.(person within Student)**

```
#include<stdio.h>
struct Psrson
{
    char name[20];
    int age;
};
struct Student
{
    int roll;
    char sec;
    struct Person p;
};
```

**Equivalent form of nested structure is:**

```
struct Student
{
    int roll;
    char sec;
    struct Person
    {
        char name[20];
        int age;
    }p;
};
```

```
void main()
{
    struct Student s;
    printf("Enter the name of a student");
    gets(s.p.name);
    printf("Enter age");
    scanf("%d",&s.p.age);
    printf("Enter the roll number of a student");
    scanf("%d",&s.roll);
    printf("Enter the section of a student");
    scanf("%c",&s.sec);
    //displaying the records
    printf("Name=%s\n Roll number =%d\n Age=%d\n Section=%c\n",s.p.name, s.roll,
    s.p.age, s.sec);
}
```

## Passing entire structures to functions:

```
#include<stdio.h>
void display(struct student);
struct student
{
    char name[20];
    int age;
    int roll;
    char sec;
};
void main()
{
    struct student s;
    int i;
    printf("Enter the name of a student");
    gets(s.name);
    printf("Enter age");
    scanf("%d",&s.age);
    printf("Enter the roll number of a student");
    scanf("%d",&s.roll);
    printf("Enter the section of a student");
    scanf("%c",&s.sec);
    display(s); //function call
```

```
}

void sisplay(struct student st)
{
    //displaying the records
    printf("Name=%s\n Roll number =%d\n Age=%d\n Section=%c\n",st.name, st.roll,
    st.age, st.sec);
}
```

# Pointers

A pointer is a variable that holds address (memory location) of another variable rather than actual value. Also, a pointer is a variable that points to or references a memory location in which data is stored. Each memory cell in the computer has an address that can be used to access that location. So, a pointer variable points to a memory location and we can access and change the contents of this memory location via the pointer. Pointers are used frequently in C, as they have a number of useful applications. In particular, pointers provide a way to return multiple data items from a function via function arguments.

## Pointer Declaration

Pointer variables, like all other variables, must be declared before they may be used in a C program. We use asterisk (\*) to do so. Its general form is:

*data-type \*ptrvar;*

## **For example,**

```
int* ptr;  
  
float *q;  
  
char *r;
```

This statement declares the variable *ptr* as a pointer to *int*, that is, *ptr* can hold address of an integer variable.

## **Reasons for using pointer:**

- A pointer enables us to access a variable that is defined outside the function.
- Pointers are used in dynamic memory allocation.
- They are used to pass array to functions.
- They produce compact, efficient and powerful code with high execution speed.
- The pointers are more efficient in handling the data table.
- They use array of pointers in character strings result in saving of data storage space in memory. Sorting strings using pointer is very efficient.
- With the help of pointer, variables can be swapped without physically moving them.
- Pointers are closely associated with arrays and therefore provides an alternate way to access individual array elements.

## **Pointer initialization:**

Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement as follows:

```
int marks;  
int *marks_pointer;  
Marks_pointer=&marks;
```

## **Passing (call) by Value and Passing (call) by Reference**

Arguments can generally be passed to functions in one of the two ways:

- Sending the values of the arguments (pass by value)
- Sending the addresses of the arguments (pass by reference)

**Pass by value:** In this method, the value of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function. With this method the changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function. The following program illustrates ‘call by value’.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b;
    void swap(int, int );
    clrscr();
    a = 10;
    b = 20;
    swap(a,b);
    printf("a = %d\tb = %d",a,b);
    getch();
}
```

```
void swap(int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
    printf("x = %d\ty = %d\n",x,y);
}
```

The output of the above program would be

x = 20 y = 10  
a = 10 b = 20

Note that values of **a** and **b** are unchanged even after exchanging the values of x and y.

**Pass by reference:** In this method, the addresses of actual arguments in the calling function are copied into formal arguments of the called function. This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them. The following program illustrates this fact.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b;
    void swap(int*, int*);
    clrscr();
    a = 10;
    b = 20;
    swap(&a,&b);
    printf("a = %d\b = %d",a,b);
    getch();
}
void swap(int *x, int *y)
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
    printf("x = %d\b y = %d\n",*x,*y);
}
```

The output of the above program would be

$x = 20$   $y = 10$   
 $a = 20$   $b = 10$

**Note:** We can use call by reference to return multiple values from the function.

## Pointers and Arrays

An array name by itself is an address, or pointer. A pointer variable can take different addresses as values. In contrast, an array name is an address, or pointer, that is fixed.

## Pointers and One-dimensional Arrays

In case of one dimensional array, an array name is really a pointer to the first element in the array. Therefore, if  $x$  is a one-dimensional array, then the address of the first array element can be expressed as either  $\&x[0]$  or simply  $x$ . Moreover, the address of the

second array element can be expressed as either  $\&x[1]$  or as  $(x+1)$ , and so on. In general, the address of array element  $(x+i)$  can be expressed as either  $\&x[i]$  or as  $(x+i)$ . Thus we have two different ways to write the address of any array element: we can write the actual array element, preceded by an ampersand; or we can write an expression in which the subscript is added to the array name.

Since,  $\&x[i]$  and  $(x+i)$  both represent the address of the  $i$ th element of  $x$ , it would seem reasonable that  $x[i]$  and  $*(x+i)$  both represent the contents of that address, i.e., the value of the  $i$ th element of  $x$ . The two terms are interchangeable. Hence, either term can be used in any particular situation. The choice depends upon your individual preferences. For example,

```
/* Program to read n numbers in an array and display their sum and average */
#include<stdio.h>
#include<conio.h>
#define SIZE 100
void main()
{
    float a[SIZE],sum=0,avg;
    int n,i;
    clrscr();
    printf("How many numbers?");
    scanf("%d",&n);
    printf("Enter numbers:\n");
    for(i=0;i<n;i++)
    {
        scanf("%f",(a+i)); // scanf("%f",&a[i]);
        sum=sum+*(a+i); //sum=sum+a[i];
    }
    avg=sum/n;
    printf("Sum=%f\n",sum);
    printf("Average=%f",avg);
    getch();
}
```

```
/* using pointer write a program to add two  $3 \times 2$  matrices and print the result in matrix form */
#include<stdio.h>
#include<conio.h>
#define ROW 3
#define COL 2
void main()
{
    int a[ROW][COL],b[ROW][COL],i,j,sum;
    clrscr();
    printf("Enter elements of first matrix:\n");
    for(i=0;i<ROW;i++)
    {
        for(j=0;j<COL;j++)
            scanf("%d", (*(a+i)+j));
        printf("\n");
    }
}
```

```
printf("Enter elements of second matrix:\n");
for(i=0;i<ROW;i++)
{
    for(j=0;j<COL;j++)
        scanf("%d", (*(b+i)+j));
    printf("\n");
}
printf("Addition matrix is:\n");
for(i=0;i<ROW;i++)
{
    for(j=0;j<COL;j++)
    {
        sum = *(*(a+i)+j)+*(*(b+i)+j);
        printf("%d\t",sum);
    }
    printf("\n");
}
getch();
}
```

```

/*Sum of two matrix using dynamic memory allocation*/
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void read(int**, int, int);
void write(int**, int, int);
void main()
{
    int **a;
    int **b;
    int **s;
    int r,c,i,j;
    clrscr();
    printf("Enter no of row and columns of a matrix\n");
    scanf("%d%d",&r,&c);
    for(i=0;i<r;i++)
    {
        *(a+i)=(int*)malloc(sizeof(int)*c);
        *(b+i)=(int*)malloc(sizeof(int)*c);
        *(s+i)=(int*)malloc(sizeof(int)*c);
    }
    printf("Enter elements of first matrix");
    read(a,,r,c);
    printf("Enter elements of Second matrix");
    read(b,,r,c);
    for(i=0;i<r;i++)
    {
        for(j=0;j<c;j++)
        {
            *((s+i)+j)=*(*(a+i)+j)+*(*(b+i)+j);
        }
    }
    printf("Matrix A is:\n\n");
    write(a,r,c);
    printf("Matrix B is:\n\n");
}

```

```

        write(b,r,c);
        printf("Sum of matrix A and B is:\n\n");
        write(s,r,c);
        getch();
    }
void read(int **x,,int r,int c)
{
    int i,j;
    for(i=0;i<r;i++)
    {
        for(j=0;j<c;j++)
        {
            scanf("%d",*(x+i)+j);
        }
    }
}
void write(int**y,int r,int c)
{
    int i,j;
    for(i=0;i<r;i++)
    {
        for(j=0;j<c;j++)
        {
            printf("%d\t",*(y+i)+j));
        }
        printf("\n");
    }
}

```

# Thanks You

# DSA(Lecture#3)

Prepared By Bal Krishna Subedi

# The stack

## a. concept and definition

- primitive operations
- Stack as an ADT
- Implementing PUSH and POP operation
- Testing for overflow and underflow conditions

## b. The infix, postfix and prefix

- Concept and definition
- Evaluating the postfix operation
- Converting from infix to postfix

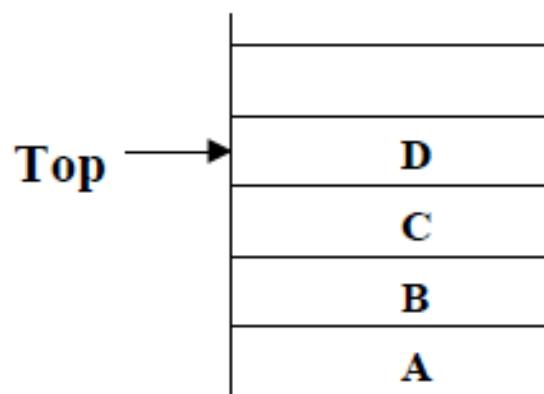
## c. Recursion

- Concept and definition
- Implementation of:
  - ✓ Multiplication of natural numbers
  - ✓ Factorial
  - ✓ Fibonacci sequences
  - ✓ The tower of Hanoi

## Introduction to Stack

*A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the top of the stack.* The deletion and insertion in a stack is done from top of the stack.

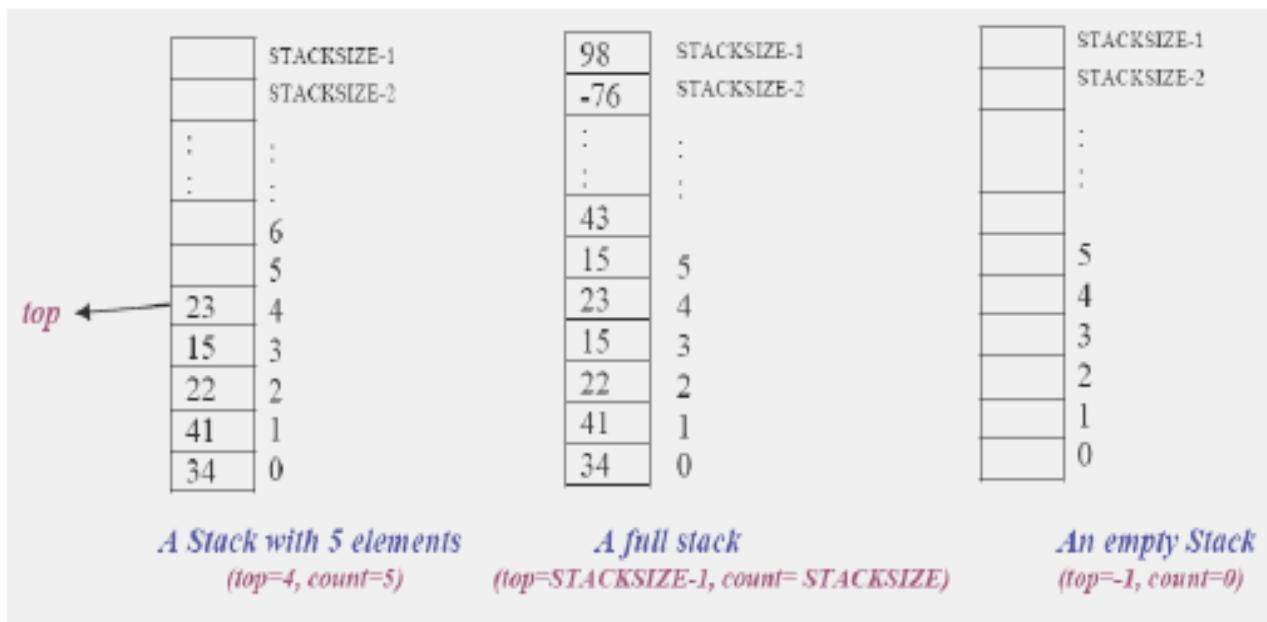
The following fig shows the stack containing items:



(Fig: A stack containing elements or items)

Intuitively, a stack is like a pile of plates where we can only (conveniently) remove a plate from the top and can only add a new plate on the top.

In computer science we commonly place numbers on a stack, or perhaps place records on the stack



## Applications of Stack:

Stack is used directly and indirectly in the following fields:

- To evaluate the expressions (postfix, prefix)
- To keep the page-visited history in a Web browser
- To perform the undo sequence in a text editor
- Used in recursion
- To pass the parameters between the functions in a C program
- Can be used as an auxiliary data structure for implementing algorithms
- Can be used as a component of other data structures

## Stack Operations:

The following operations can be performed on a stack:

❖ **PUSH** operation: The push operation is used to add (or push or insert) elements in a stack

- ▶ When we add an item to a stack, we say that we **push** it onto the stack
- ▶ The last item put into the stack is at the top

	STACKSIZE-1
	STACKSIZE-2
:	
:	
6	
5	
23	
4	
15	
3	
22	
2	
41	
1	
34	0

*Before PUSH  
(top=4, count=5)*

	STACKSIZE-1
	STACKSIZE-2
:	
:	
15	
5	
23	
4	
15	
3	
22	
2	
41	
1	
34	0

*After PUSH  
(top=5, count= 6)*

❖ **POP** operation: The pop operation is used to remove or delete the top element from the stack.

- ▶ When we remove an item, we say that we **pop** it from the stack

- When an item popped, it is always the top item which is removed

	STACKSIZE-1	STACKSIZE-2
:	:	:
:	:	:
6		5
5		4
<b>23</b>		
15	15	3
3	22	2
<b>22</b>	41	1
41	34	0
34		

*Before POP  
(top=4, count=5)*

	STACKSIZE-1	STACKSIZE-2
:	:	:
:	:	:
5		4
4		3
<b>15</b>	22	2
3	41	1
<b>22</b>	34	0
41		
34		

*After POP  
(top=3 count=4)*

The **PUSH** and the **POP** operations are the *basic or primitive* operations on a stack. Some others operations are:

- **CreateEmptyStack** operation: This operation is used to create an empty stack.
- **IsFull** operation: The **isfull** operation is used to check whether the stack is full or not ( i.e. stack overflow)
- **IsEmpty** operation: The **isempty** operation is used to check whether the stack is empty or not. (i. e. stack underflow)
- **Top operations:** This operation returns the current item at the top of the stack, it doesn't remove it

## The Stack ADT:

A stack of elements of type  $T$  is a finite sequence of elements of  $T$  together with the operations

- **CreateEmptyStack(S):** Create or make stack  $S$  be an empty stack
- **Push(S, x):** Insert  $x$  at one end of the stack, called its **top**
- **Top(S):** If stack  $S$  is not empty; then retrieve the element at its **top**
- **Pop(S):** If stack  $S$  is not empty; then delete the element at its **top**
- **IsFull(S):** Determine if  $S$  is full or not. Return **true** if  $S$  is full stack; return **false** otherwise
- **IsEmpty(S):** Determine if  $S$  is empty or not. Return **true** if  $S$  is an empty stack; return **false** otherwise.

## **Implementation of Stack:**

Stack can be implemented in two ways:

1. Array Implementation of stack (or static implementation)
2. Linked list implementation of stack (or dynamic)

## **Array (static) implementation of a stack:**

It is one of two ways to implement a stack that uses a one dimensional array to store the data. In this implementation top is an integer value (an index of an array) that indicates the top position of a stack. Each time data is added or removed, top is incremented or decremented accordingly, to keep track of current top of the stack. By convention, in C implementation the empty stack is indicated by setting the value of top to -1( $\text{top}=-1$ ).

```
#define MAX 10  
  
struct stack  
{  
    int items[MAX]; //Declaring an array to store items  
    int top;        //Top of a stack  
};  
typedef struct stack st;
```

## Creating Empty stack :

The value of top=-1 indicates the empty stack in C implementation.

```
/*Function to create an empty stack*/  
void create_empty_stack(st *s)  
{  
    s->top=-1;  
}
```

## Stack Empty or Underflow:

This is the situation when the stack contains no element. At this point the top of stack is present at the bottom of the stack. In array implementation of stack, conventionally top=-1 indicates the empty.

The following function return 1 if the stack is empty, 0 otherwise.

```
int isempty(st *s)  
{  
    if(s->top== -1)  
        return 1;  
    else  
        return 0;  
}
```

### **Stack Full or Overflow:**

This is the situation when the stack becomes full, and no more elements can be pushed onto the stack. At this point the stack top is present at the highest location (MAXSIZE-1) of the stack. The following function returns true (1) if stack is full false (0) otherwise.

```
int isfull(st *s)
{
    if(s->top==MAX-1)
        return 1;
    else
        return 0;
}
```

### **Algorithm for PUSH and POP operations on Stack**

Let Stack[MAXSIZE] be an array to implement the stack. The variable top denotes the top of the stack.

#### **i) Algorithm for PUSH (inserting an item into the stack) operation:**

This algorithm adds or inserts an item at the top of the stack

1. [Check for stack overflow?]  
if  $top=MAXSIZE-1$  then  
    print "Stack Overflow" and Exit  
else  
    Set  $top=top+1$  [Increase top by 1]

*Set Stack[top]:= item [Inserts item in new top position]*

2. *Exit*

**ii) Algorithm for POP (removing an item from the stack) operation**

This algorithm deletes the top element of the stack and assign it to a variable *item*

1. *[Check for the stack Underflow]*

*If top<0 then*

*Print "Stack Underflow" and Exit*

*else*

*[Remove the top element]*

*Set item=Stack [top]*

*[Decrement top by 1]*

*Set top=top-1*

*Return the deleted item from the stack*

2. *Exit*

## The PUSH and POP functions

The C function for **push** operation

```
void push(st *s, int element)
{
    if(isfull(s)) /* Checking Overflow condition */
        printf("\n \n The stack is overflow: Stack Full!!\n");
    else
        s->items[++(s->top)]=element; /* First increase top by 1 and store element at top position*/
}
```

## OR

Alternatively we can define the push function as give below:

```
void push()
{
    int item;
    if(top == MAXSIZE - 1) //Checking stack overflow
        printf("\n The Stack Is Full");
    else
    {
        printf("Enter the element to be inserted");
        scanf("%d",&item); //reading an item
        top= top+1;           //increase top by 1
        stack[top] = item;    //storing the item at the top of
                               //the stack
    }
}
```

## The C function for POP operation

```
void pop(stack *s)
{
    if(isempty(s))
        printf("\n\nstack Underflow: Empty Stack!!!");
    else
        printf("\nthe deleted item is %d:\t",s->items[s->top-1]);/*deletes top element and
decrease top by 1 */
}
```

## OR

Alternatively we can define the push function as give below:

```
void pop()
{
    int item;
    if(top <0) //Checking Stack Underflow
        printf("The stack is Empty");
    else
    {
        item = stack[top]; //Storing top element to item variable
        top = top-1; //Decrease top by 1
        printf("The popped item is=%d",item); //Displaying the deleted item
    }
}
```

## Infix, Prefix and Postfix Notation

One of the applications of the stack is to evaluate the expression. We can represent the expression following three types of notation:

- Infix
- Prefix
- Postfix

- ❖ **Infix expression:** It is an ordinary mathematical notation of expression where operator is written in between the operands. Example: A+B. Here '+' is an *operator* and *A* and *B* are called *operands*
- ❖ **Prefix notation:** In prefix notation the operator precedes the two operands. That is the operator is written before the operands. It is also called polish notation. Example: +AB
- ❖ **Postfix notation:** In postfix notation the operators are written after the operands so it is called the postfix notation (post mean after). In this notation the operator follows the two operands. Example: AB+

### Examples:

A + B (Infix)

+ AB (Prefix)

AB + (Postfix)

- Both prefix and postfix are parenthesis free expressions. For example

(A + B) \* C                  Infix form

\* + A B C                  Prefix form

A B + C \*                  Postfix form

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

## Converting an Infix Expression to Postfix

First convert the sub-expression to postfix that is to be evaluated first and repeat this process. You substitute intermediate postfix sub-expression by any variable whenever necessary that makes it easy to convert.

- ❖ Remember, to convert an infix expression to its postfix equivalent, we first convert the innermost parenthesis to postfix, resulting as a new operand
- In this fashion parenthesis can be successively eliminated until the entire expression is converted
- ❖ The last pair of parenthesis to be opened within a group of parenthesis encloses the first expression within the group to be transformed
- This last in, first-out behavior suggests the use of a stack

## Precedence rule:

While converting infix to postfix you have to consider the **precedence rule**, and the precedence rules are as follows

1. Exponentiation ( the expression  $A\$B$  is A raised to the B power, so that  $3\$2=9$ )
2. Multiplication/Division
3. Addition/Subtraction

When un-parenthesized operators of the same precedence are scanned, the order is assumed to be left to right except in the case of exponentiation, where the order is assumed to be from right to left.

- $A+B+C$  means  $(A+B)+C$
- $A\$B\$C$  means  $A\$(B\$C)$

By using parenthesis we can override the default precedence.

Consider an example that illustrate the converting of infix to postfix expression,  $A + (B^* C)$ .

Use the following **rule** to convert it in postfix:

1. Parenthesis for emphasis
2. Convert the multiplication
3. Convert the addition
4. Post-fix form

**Illustration:**

$A + (B * C)$ . Infix form

$A + (B * C)$       Parenthesis for emphasis

$A + (BC^*)$       Convert the multiplication

$A (BC^*) +$       Convert the addition

$ABC^*+$  Post-fix form

### Consider an example:

$(A + B) * ((C - D) + E) / F$	Infix form
$(AB+) * ((C - D) + E) / F$	
$(AB+) * ((CD-) + E) / F$	
$(AB+) * (CD-E+) / F$	
$(AB+CD-E+*) / F$	
$AB+CD-E+*F/$	Postfix form

### Examples

<b><i>Infix</i></b>	<b><i>Postfix</i></b>
$A + B$	$AB +$
$A + B - C$	$AB + C -$
$(A + B) * (C - D)$	$AB + CD - *$
$A \$ B * C - D + E / F / (G + H)$	$AB \$ C * D - EF / GH + / +$
$((A + B) * C - (D - E)) \$ (F + G)$	$AB + C * DE - - FG + \$$
$A - B / (C * D \$ E)$	$ABCDE \$ * / -$

## **Algorithm to convert infix to postfix notation**

Let here two stacks opstack and poststack are used and otos & ptos represents the opstack top and poststack top respectively.

1. Scan one character at a time of an infix expression from left to right
2. opstack=the empty stack
3. Repeat till there is data in infix expression
  - 3.1 if scanned character is '(' then push it to opstack
  - 3.2 if scanned character is operand then push it to poststack
  - 3.3 if scanned character is operator then
    - if(opstack!= -1)  
while(precedence (opstack[otos])>precedence(scan character)) then  
pop and push it into poststack
    - otherwise  
push into opstack
  - 3.4 if scanned character is ')' then  
pop and push into poststack until '(' is not found and ignore both symbols
4. pop and push into poststack until opstack is not empty.
5. return

### Trace of Conversion Algorithm

The following tracing of the algorithm illustrates the algorithm. Consider an infix expression

$((A-(B+C))^D\$(E+F))$

Scan character	Poststack	opstack
(	.....	(
(	.....	((
A	A	((
-	A	(( -
(	A	(( -(
B	AB	(( -(
+	AB	(( -( +
C	ABC	(( -( +
)	ABC+	(( -
)	ABC+-	(
*	ABC+-	(*
D	ABC+-D	(*
)	ABC+-D*	.....
\$	ABC+-D*	\$
(	ABC+-D*	\$()
E	ABC+-D*E	\$()
+	ABC+-D*E	\$(+
F	ABC+-D*EF	\$(+
)	ABC+-D*EF+	\$
.....	ABC+-D*EF+\$ (postfix)	.....

## Converting an Infix expression to Prefix expression

The precedence rule for converting from an expression from infix to prefix are identical.

Only changes from postfix conversion is that the operator is placed before the operands rather than after them. The prefix of

A+B-C is -+ABC.

A+B-C (infix)

=(+AB)-C

=-+ABC (prefix)

*Example Consider an example:*

A \$ B \* C - D + E / F / (G + H)    infix form

= A \$ B \* C - D + E / F / (+GH)

=\$AB\* C - D + E / F /(+GH)

=\*\$ABC-D+E/F/(+GH)

=\*\$ABC-D+(/EF)/(+GH)

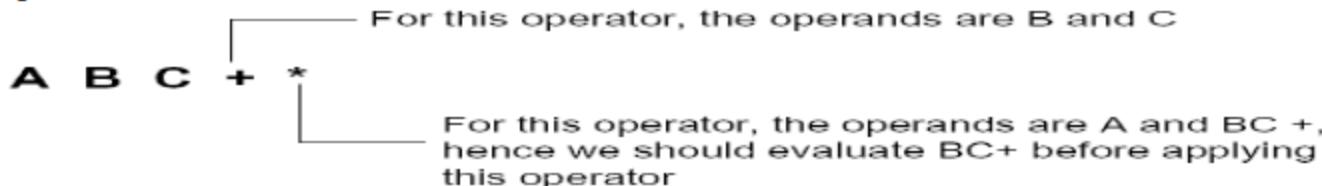
=\*\$ABC-D+//EF+GH

= (-\*\$ABCD) + (//EF+GH)

=+-\$ABCD//EF+GH    which is in prefix form.

## Evaluating the Postfix expression

Each operator in a postfix expression refers to the previous two operands in the expression.



To evaluate the postfix expression we use the following procedure:

Each time we read an operand we push it onto a stack. When we reach an operator, its operands will be the top two elements on the stack. We can then pop these two elements perform the indicated operation on them and push the result on the stack so that it will be available for use as an operand of the next operator.

### **Consider an example**

3 4 5 \* +  
=3 20 +  
=23 (answer)

**Evaluating the given postfix expression:**

**6 2 3 + - 3 8 2 / + \* 2 \$ 3 +**

**=6 5 - 3 8 2 / + \* 2 \$ 3 +**

**=1 3 8 2 / + \* 2 \$ 3 +**

**=1 3 4 + \* 2 \$ 3 +**

**=1 7 \* 2 \$ 3 +**

**=7 2 \$ 3 +**

**=49 3 +**

**= 52**

## Algorithm to evaluate the postfix expression

Here we use only one stack called vstack(value stack).

1. Scan one character at a time from left to right of given postfix expression
  - 1.1 if scanned symbol is operand then  
read its corresponding value and push it into vstack
  - 1.2 if scanned symbol is operator then
    - pop and place into op2
    - op and place into op1
    - compute result according to given operator and push result into vstack
2. pop and display which is required value of the given postfix expression
3. return

## Trace of Evaluation:

Consider an example to evaluate the postfix expression tracing the algorithm

**ABC+\*CBA-+\***  
**123+\*321-+\***

Scanned character	value	Op2	Op1	Result	vstack
A	1	.....	.....	.....	1
B	2	.....	.....	.....	1 2
C	3	.....	.....	.....	1 2 3
+	.....	3	2	5	1 5
*	.....	5	1	5	5
C	3	.....	.....	.....	5 3
B	2	...	.....		5 3 2
A	1	.....	.....	.....	5 3 2 1
-	.....	1	2	1	5 3 1
+	.....	1	3	4	5 4
*	.....	4	5	20	20

Its final value is 20.

## Evaluating the Prefix Expression

To evaluate the prefix expression we use two stacks and some time it is called two stack algorithms. One stack is used to store operators and another is used to store the operands. Consider an example for this

+ 5 \*3 2 prefix expression

= +5 6

= 11

**Illustration:** Evaluate the given prefix expression

/ + 5 3 - 4 2      prefix equivalent to  $(5+3)/(4-2)$  infix notation

= / 8 - 4 2

= / 8 2

= 4

## **Recursion:**

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a previous result.

In order to solve a problem recursively, two conditions must be satisfied. First, the problem must be written in a recursive form, and second, the problem statement must include a stopping condition.

## **Example:**

```
/*calculation of the factorial of an integer number using recursive function*/
#include<stdio.h>
#include<conio.h>
void main()
{
    int n;
    long int facto;
    long int factorial(int n);
    printf("Enter value of n:");
    scanf("%d",&n);
    facto=factorial(n);
    printf("%d! = %ld",n,facto);
    getch();
}
long int factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

## Let's trace the evaluation of factorial(5):

Factorial(5)=  
5\*Factorial(4)=  
5\*(4\*Factorial(3))=  
5\*(4\*(3\*Factorial(2)))=  
5\*(4\*(3\*(2\*Factorial(1))))=  
5\*(4\*(3\*(2\*(1\*Factorial(0)))))=  
5\*(4\*(3\*(2\*(1\*1))))=  
5\*(4\*(3\*(2\*1)))=  
5\*(4\*(3\*2))=  
5\*(4\*6)=  
5\*24=  
120

### **Example:**

```
/*calculation of the factorial of an integer number without using recursive function*/
#include<stdio.h>
#include<conio.h>
void main()
{
    int n;
    long int facto;
    long int factorial(int n);
    printf("Enter value of n:");
    scanf("%d",&n);
    facto=factorial(n);
    printf("%d! = %ld",n,facto);
    getch();
}
```

```

long int factorial(int n)
{
    long int facto=1;
    int i;
    if(n==0)
        return 1;
    else {
        for(i=1;i<=n;i++)
            facto=facto*i;
        return facto;
    }
}

/* Program to find sum of first n natural numbers using recursion*/
#include<stdio.h>
#include<conio.h>
void main()
{
    int n;
    int sum_natural(int );
    printf("n = ");
    scanf("%d",&n);
    printf("Sum of first %d natural numbers = %d",n,sum_natural(n));
    getch();
}
int sum_natural(int n)
{
    if(n == 1)
        return 1;
    else
        return n + sum_natural(n-1);
}

```

## **Tower of Hanoi problem:**

### **Initial state:**

- There are three poles named as origin, intermediate and destination.
- $n$  number of different-sized disks having hole at the center is stacked around the origin pole in decreasing order.
- The disks are numbered as 1, 2, 3, 4, ..... $n$ .

### **Objective:**

- Transfer all disks from origin pole to destination pole using intermediate pole for temporary storage.

### **Conditions:**

- Move only one disk at a time.
- Each disk must always be placed around one of the pole.
- Never place larger disk on top of smaller disk.

**Algorithm:** - To move a tower of  $n$  disks from *source* to *dest* (where  $n$  is positive integer):

1. If  $n == 1$ :
  - 1.1. Move a single disk from *source* to *dest*.
2. If  $n > 1$ :
  - 2.1. Let *temp* be the remaining pole other than *source* and *dest*.
  - 2.2. Move a tower of  $(n - 1)$  disks form *source* to *temp*.
  - 2.3. Move a single disk from *source* to *dest*.
  - 2.4. Move a tower of  $(n - 1)$  disks form *temp* to *dest*.
3. Terminate.

**Example: Recursive solution of tower of Hanoi:**

```
#include <stdio.h>
#include <conio.h>
void TOH(int, char, char, char); //Function prototype
void main()
{
    int n;
    printf("Enter number of disks");
    scanf("%d",&n);
    TOH(n,'O','D','I');
    getch();
}

void TOH(int n, char A, char B, char C)
{
    if(n>0)
    {
        TOH(n-1, A, C, B);
        printf("Move disk %d from %c to%c\n", n, A, B);
        TOH(n-1, C, B, A);
    }
}
```

### **Advantages of Recursion:**

- The code may be much easier to write.
- To solve some problems which are naturally recursive such as tower of Hanoi.

### **Disadvantages of Recursion:**

- Recursive functions are generally slower than non-recursive functions.
- May require a lot of memory to hold intermediate results on the system stack.
- It is difficult to think recursively so one must be very careful when writing recursive functions.

# Thanks You

# DSA lecture#4

Prepared By bal Krishna Subedi

## Queues

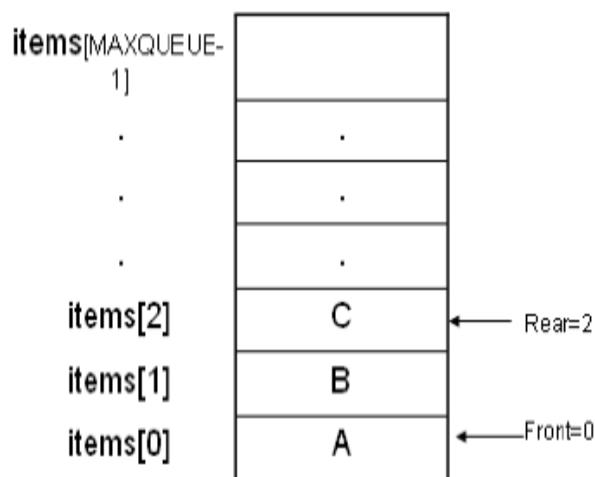
- a) Concept and definition
- b) Queue as ADT
- c) Implementation of insert and delete operation of
  - Linear queue
  - Circular queue
- d) Concept of priority queue

### What is a queue?

> A **Queue** is an ordered collection of items from which items may be deleted at one end (called the *front* of the queue) and into which items may be inserted at the other end (the *rear* of the queue).

- > The first element inserted into the queue is the first element to be removed. For this reason a queue is sometimes called a *fifo* (first-in first-out) list as opposed to the stack, which is a *lifo* (last-in first-out).

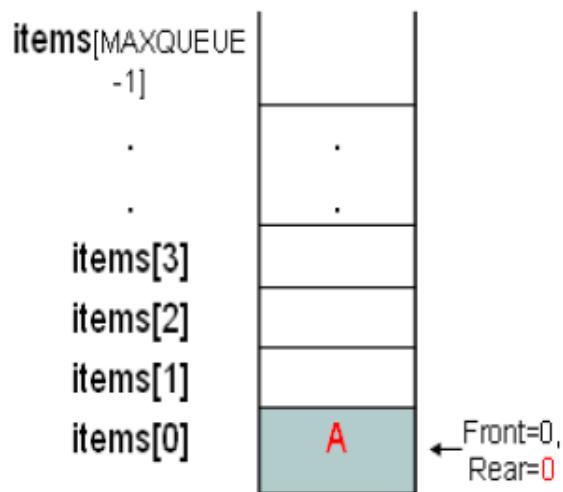
### Example:



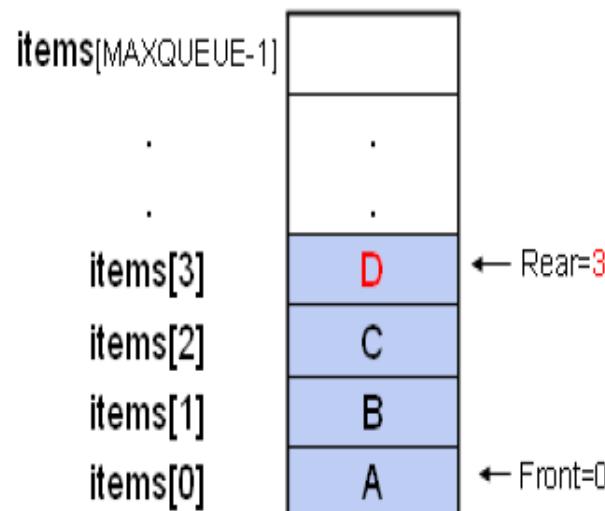
## Operations on queue:

- ***MakeEmpty(q):*** To make q as an empty queue
- ***Enqueue(q, x):*** To insert an item x at the rear of the queue, this is also called by names add, insert.
- ***Dequeue(q):*** To delete an item from the front of the queue q. this is also known as Delete, Remove.
- ***IsFull(q):*** To check whether the queue q is full.
- ***IsEmpty(q):*** To check whether the queue q is empty
- ***Traverse (q):*** To read entire queue that is display the content of the queue.

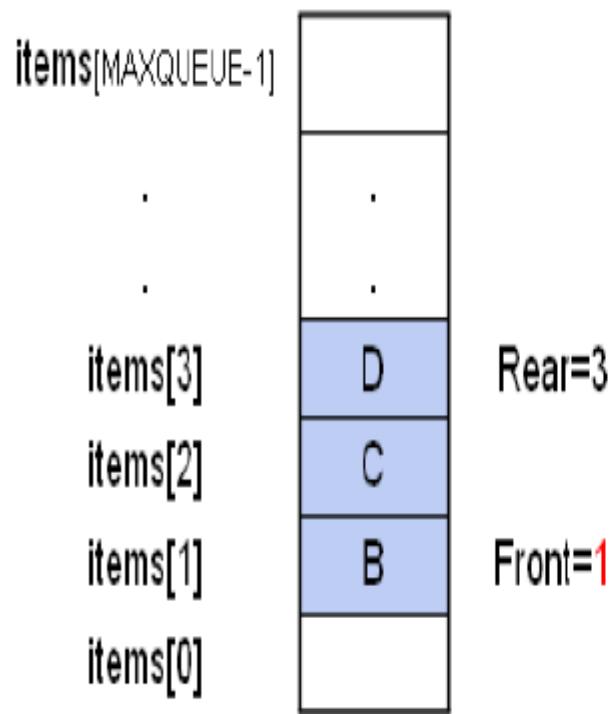
**Enqueue(A):**



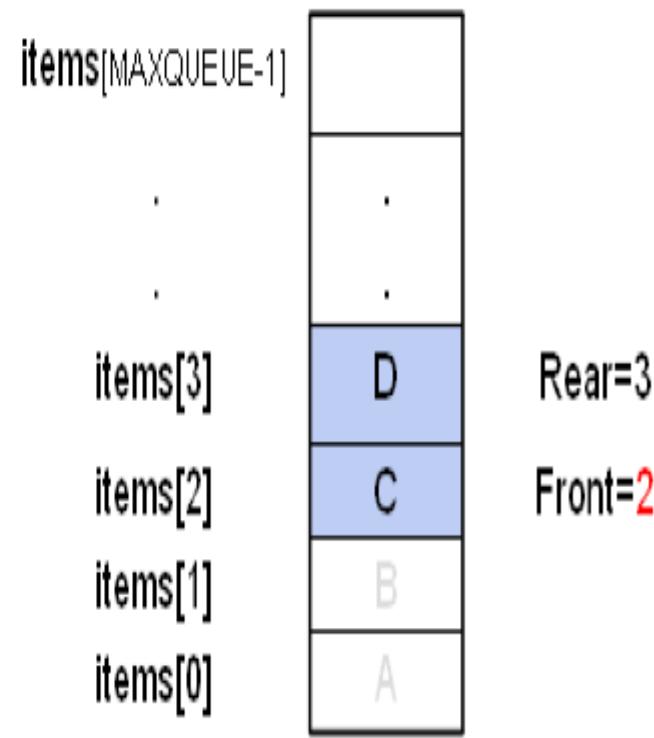
**Enqueue(B,C,D):**



Dequeue(A):



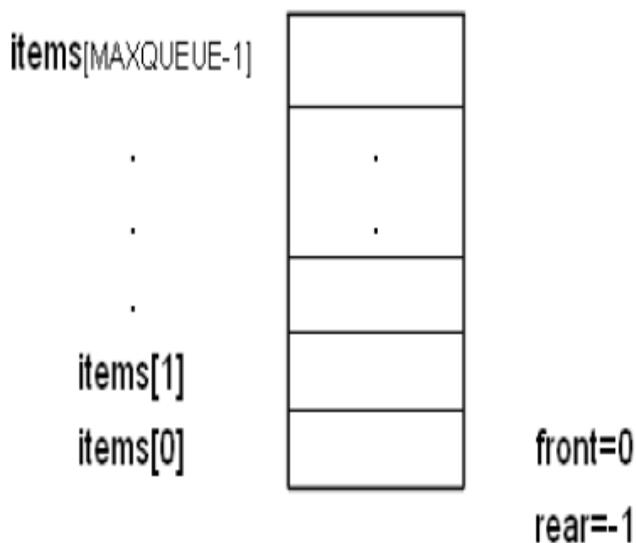
Dequeue(B):



## Initialization of queue:

- The queue is initialized by having the *rear* set to -1, and *front* set to 0. Let us assume that maximum number of the element we have in a queue is MAXQUEUE elements as shown below.

```
rear=-1  
front=0
```



## Applications of queue:

- Task waiting for the printing
- Time sharing system for use of CPU
- For access to disk storage
- Task scheduling in operating system

## The Queue as a ADT:

A queue  $q$  of type  $T$  is a finite sequence of elements with the operations

- ***MakeEmpty(q)***: To make  $q$  as an empty queue
- ***IsEmpty(q)***: To check whether the queue  $q$  is empty. Return true if  $q$  is empty, return false otherwise.
- ***IsFull(q)***: To check whether the queue  $q$  is full. Return true if  $q$  is full, return false otherwise.
- ***Enqueue(q, x)***: To insert an item  $x$  at the rear of the queue, if and only if  $q$  is not full.
- ***Dequeue(q)***: To delete an item from the front of the queue  $q$ . if and only if  $q$  is not empty.
- ***Traverse (q)***: To read entire queue that is display the content of the queue.

## **Implementation of queue:**

There are two techniques for implementing the queue:

- Array implementation of queue(static memory allocation)
- Linked list implementation of queue(dynamic memory allocation)

## **Array implementation of queue:**

In array implementation of queue, an array is used to store the data elements. Array implementation is also further classified into two types

✓ ***Linear array implementation:***

A linear array with two indices always increasing that is rear and front. Linear array implementation is also called linear queue

✓ ***Circular array implementation:***

This is also called circular queue.

## Linear queue:

### Algorithm for insertion (or Enqueue ) and deletion (Dequeue) in queue:

#### Algorithm for insertion an item in queue:

```
1. Initialize front=0 and rear=-1  
   if rear>=MAXSIZE-1  
     print "queue overflow" and return  
   else  
     set rear=rear+1  
     queue[rear]=item  
2. end
```

#### Algorithm to delete an element from the queue:

```
1. if rear<front  
   print "queue is empty" and return  
   else  
     item=queue[front++]  
2. end
```

## Declaration of a Queue:

```
# define MAXQUEUE 50 /* size of the queue items*/
struct queue
{
    int front;
    int rear;
    int items[MAXQUEUE];
};

typedef struct queue qt;
```

## Defining the operations of linear queue:

- The MakeEmpty function:

```
void makeEmpty(qt *q)
{
    q->rear=-1;
    q->front=0;
}
```

- The IsEmpty function:

```
int IsEmpty(qt *q)
{
    if(q->rear < q->front)
        return 1;
    else
        return 0;
}
```

- The Isfull function:

```
int IsFull(qt *q)
{
    if(q->rear == MAXQUEUEZIZE - 1)
        return 1;
    else
        return 0;
}
```

- The Enqueue function:

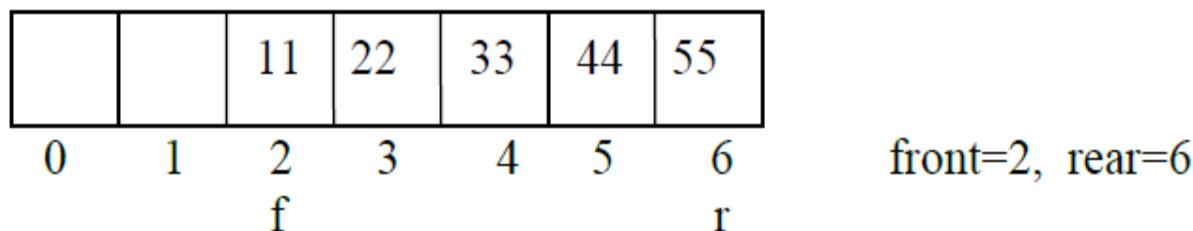
```
void Enqueue(qt *q, int newitem)
{
    if(IsFull(q))
    {
        printf("queue is full");
        exit(1);
    }
    else
    {
        q->rear++;
        q->items[q->rear]=newitem;
    }
}
```

- The Dequeue function:

```
int Dequeue(qt *q)
{
    if(IsEmpty(q))
    {
        printf("queue is Empty");
        exit(1);
    }
    else
    {
        return(q->items[q->front]);
        q->front++;
    }
}
```

### Problems with Linear queue implementation:

- Both rear and front indices are increased but never decreased.
- As items are removed from the queue, the storage space at the beginning of the array is discarded and never used again. Wastage of the space is the main problem with linear queue which is illustrated by the following example.



This queue is considered full, even though the space at beginning is vacant.

```

/*Array implementation of linear queue*/
#include<stdio.h>
#include<conio.h>
#define SIZE 20
struct queue
{
    int item[SIZE];
    int rear;
    int front;
};
typedef struct queue qu;
void insert(qu* );
void delet(qu* );
void display(qu* );
void main()
{
    int ch;
    qu *q;
    q->rear=-1;
    q->front=0;
    clrscr();
    printf("Menu for program:\n");
    printf("1:insert\n2:delete\n3:display\n4:exit\n");
    do
    {
        printf("Enter your choice\n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                insert(q);
                break;
            case 2:
                delet(q);
                break;
            case 3:
                display(q);
                break;
            case 4:
                exit(1);
                break;
            default:
                printf("Your choice is wrong\n");
        }
    }while(ch<5);
    getch();
}

```

```
void insert(qu *q)
{
    int d;
    printf("Enter data to be inserted\n");
    scanf("%d",&d);

    if(q->rear==SIZE-1)
    {
        printf("Queue is full\n");
    }
    else
    {
        q->rear++;
        q->item[q->rear]=d;
    }
}
```

```
void delet(qu *q)
{
    int d;
    if(q->rear<q->front)
    {
        printf("Queue is empty\n");
    }
    else
    {
        d=q->item[q->front];
        q->front++;
        printf("Deleted item is:");
        printf("%d\n",d);
    }
}
```

```
void display(qu *q)
{
    int i;
    if(q->rear<q->front)
    {
        printf("Queue is empty\n");
    }
    else
    {
        for(i=q->front;i<=q->rear;i++)
        {
            printf("%d\t",q->item[i]);
        }
    }
}
```

## Circular queue:

A circular queue is one in which the insertion of a new element is done at very first location of the queue if the last location of the queue is full.

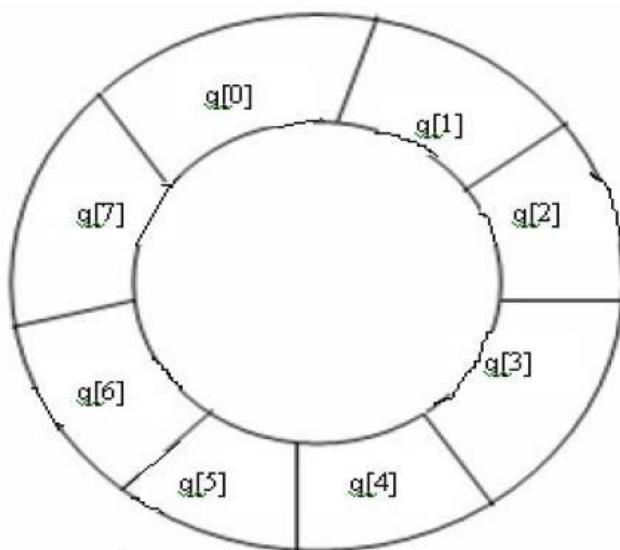


Fig:- Circular queue

A circular queue overcomes the problem of unutilized space in linear queue implementation as array.

In circular queue we sacrifice one element of the array thus to insert  $n$  elements in a circular queue we need an array of size  $n+1$ . (or we can insert one less than the size of the array in circular queue).

## Initialization of Circular queue:

rear=front=MAXSIZE-1

## Algorithms for inserting an element in a circular queue:

This algorithm is assume that rear and front are initially set to MAXSIZE-1.

1. if (front==(rear+1)%MAXSIZE)  
          print Queue is full and exit
- else  
          rear=(rear+1)%MAXSIZE; [increment rear by 1]
2. cqueue[rear]=item;
3. end

## Algorithms for deleting an element from a circular queue:

This algorithm is assume that rear and front are initially set to MAXSIZE-1.

1. if (rear==front) [checking empty condition]  
          print Queue is empty and exit
2. front=(front+1)%MAXSIZE; [increment front by 1]
3. item=cqueue[front];
4. return item;
5. end.

## Declaration of a Circular Queue:

```
# define MAXSIZE 50 /* size of the circular queue items*/  
struct cqueue  
{  
    int front;  
    int rear;  
    int items[MAXSIZE];  
};  
typedef struct cqueue cq;
```

## Operations of a circular queue:

### ◆ The MakeEmpty function:

```
void makeEmpty(cq *q)  
{  
    q->rear=MAXSIZE-1;  
    q->front=MAXSIZE-1;  
}
```

### ◆ The IsEmpty function:

```
int IsEmpty(cq *q)  
{  
    if(q->rear<q->front)  
        return 1;  
    else  
        return 0;  
}
```

◆ **The Isfull function:**

```
int IsFull(cq *q)
{
    if(q->front==(q->rear+1)%MAXDIZE)
        return 1;
    else
        return 0;
}
```

◆ **The Enqueue function:**

```
void Enqueue(cq *q, int newitem)
{
    if(IsFull(q))
    {
        printf("queue is full");
        exit(1);
    }
    else
    {
        q->rear=(q->rear+1)%MAXDIZE;
        q->items[q->rear]=newitem;
    }
}
```

◆ *The Dequeue function:*

```
int Dequeue(cq *q)
{
    if(IsEmpty(q))
    {
        printf("queue is Empty");
        exit(1);
    }
    else
    {
        q->front=(q->front+1)%MAXSIZE;
        return(q->items[q->front]);
    }
}
```

**/\*implementation of circular queue with specifying one cell \*/**

```
#include<stdio.h>
#include<conio.h>
#define SIZE 20
struct cqueue
{
    int item[SIZE];
    int rear;
    int front;
};
typedef struct cqueue qu;
void insert(qu* );
void delet(qu* );
void display(qu* );
void main()
{
    int ch;
    qu *q;
    q->rear=SIZE-1;
    q->front=SIZE-1;
    clrscr();
    printf("Menu for program:\n");
    printf("1:insert\n2:delete\n3:display\n4:exit\n");
    do
    {
        printf("Enter your choice\n");
        scanf("%d",&ch);
```

```

switch(ch)
{
    case 1:
        insert(q);
        break;
    case 2:
        delet(q);
        break;
    case 3:
        display(q);
        break;
    case 4:
        exit(1);
        break;
    default:
        printf("Your choice is wrong\n");
        break;
}
}while(ch<5);
getch();
}

/**********insert function*******/
void insert(qu *q)
{
    int d;
    if((q->rear+1)%SIZE==q->front)
        printf("Queue is full\n");
}

```

```

else
{
    q->rear=(q->rear+1)%SIZE;
    printf ("Enter data to be inserted\n");
    scanf("%d",&d);
    q->item[q->rear]=d;
}

} **** delete function ****
void delet(qu *q)
{
    if(q->rear==q->front)
        printf("Queue is empty\n");
    else
    {
        q->front=(q->front+1)%SIZE;
        printf("Deleted item is:");
        printf("%d\n",q->item[q->front]);
    }
}
***** All functions ****

```

```
void display(qu *q)
{
    int i;
    if(q->rear==q->front)
        printf("Queue is empty\n");
    else
    {
        printf("Items of queue are:\n");
        for(i=(q->front+1)%SIZE;i!=q->rear;i=(i+1)%SIZE)
        {
            printf("%d\t",q->item[i]);
        }
        printf("%d\t",q->item[q->rear]);
    }
}
```

*/\*implementation of circular queue without specifying one cell by using a count variable \*/*

```
#include<stdio.h>
#include<conio.h>
#define SIZE 20
struct cqueue
{
    int item[SIZE];
    int rear;
    int front;
};
int count=0;
typedef struct cqueue qu;
void insert(qu* );
void delet(qu* );
void display(qu* );
void main()
{
```

```
int ch;
qu *q;
q->rear=SIZE-1;
q->front=SIZE-1;
clrscr();
printf("Menu for program:\n");
printf("1:insert\n2:delete\n3:display\n4:exit\n");
do
{
    printf("Enter youer choice\n");
    scanf("%d",&ch);
    switch(ch)
    {
        case 1:
            insert(q);
            break;
        case 2:
            delet(q);
            break;
        case 3:
            display(q);
            break;
        case 4:
            exit(1);
            break;
        default:
            printf("Your choice is wrong\n");
            break;
    }
}while(ch<5);
getch();
}
```

```
*****insert function*****
void insert(qu *q)
{
    int d;
    if(count==SIZE)
        printf("Queue is full\n");
    else
    {
        q->rear=(q->rear+1)%SIZE;
        printf ("Enter data to be inserted\n");
        scanf("%d",&d);
        q->item[q->rear]=d;
        count++;
    }

}
*****delete function*****
void delet(qu *q)
{
    if(count==0)
        printf("Queue is empty\n");
    else
    {
```

```

        q->front=(q->front+1)%SIZE;
        printf("Deleted item is:");
        printf("%d\n",q->item[q->front]);
        count--;
    }
}

*****display function*****
void display(qu *q)
{
    int i;
    if(q->rear==q->front)
        printf("Queue is empty\n");
    else
    {
        printf("Items of queue are:\n");
        for(i=(q->front+1)%SIZE; i!=q->rear; i=(i +1)%SIZE)
        {
            printf("%d\t",q->item[i]);
        }
        printf("%d\t",q->item[q->rear]);
    }
}

```

## Priority queue:

A priority queue is a collection of elements such that each element has been assigned a priority and the order in which elements are deleted and processed comes from the following rules:.

- ◆ An element of higher priority is processed before any element of lower priority.
- ◆ If two elements has same priority then they are processed according to the order in which they were added to the queue.

The best application of priority queue is observed in CPU scheduling.

- ✓ The jobs which have higher priority are processed first.
- ✓ If the priority of two jobs is same this jobs are processed according to their position in queue.
- ✓ A short job is given higher priority over the longer one.

## Types of priority queues:

- ◆ *Ascending priority queue(min priority queue):*

An **ascending priority queue** is a collection of items into which items can be inserted arbitrarily but from which only the smallest item can be removed.

- ◆ *Descending priority queue(max priority queue):*

An **descending priority queue** is a collection of items into which items can be inserted arbitrarily but from which only the largest item can be removed.

## **Priority QUEUE Operations:**

- ◆ **Insertion :**

The insertion in Priority queues is **the same as** in non-priority queues.

- ◆ **Deletion :**

Deletion requires a search for the element of highest priority and deletes the element with highest priority. The following methods can be used for deletion/removal from a given Priority Queue:

- ✓ An empty indicator replaces deleted elements.
- ✓ After each deletion elements can be moved up in the array decrementing the rear.
- ✓ The array in the queue can be maintained as an ordered circular array

## Priority Queue Declaration:

*Queue data type of Priority Queue is the same as the Non-priority Queue.*

```
#define MAXQUEUE 10 /* size of the queue items*/
struct pqueue
{
    int front;
    int rear;
    int items[MAXQUEUE];
};
struct pqueue *pq;
```

## The priority queue ADT:

A ascending priority queue of elements of type T is a finite sequence of elements of T together with the operations:

- ◆ **MakeEmpty(p):** Create an empty priority queue p
- ◆ **Empty(p):** Determine if the priority queue p is empty or not
- ◆ **Insert(p,x):** Add element x on the priority queue p
- ◆ **DeleteMin(p):** If the priority queue p is not empty, remove the minimum element of the quque and return it.
- ◆ **FindMin(p):** Retrieve the minimum element of the priority queue p.

## Array implementation of priority queue:

### ◆ Unordered array implementation:

- ✓ To insert an item, insert it at the rear end of the queue.
- ✓ To delete an item, find the position of the minimum element and
  - ✗ Either mark it as deleted (lazy deletion) or
  - ✗ shift all elements past the deleted element by one position and then decrement rear.

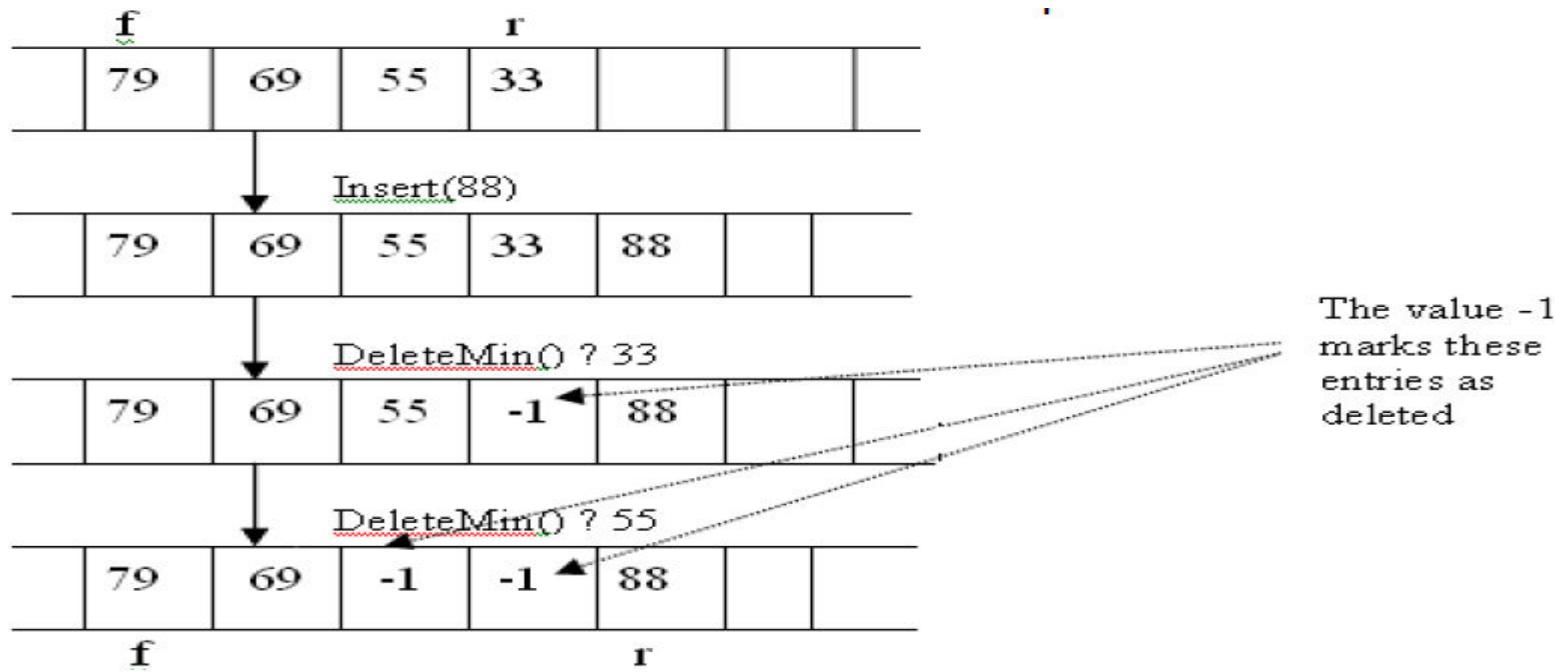


Fig Illustration of unordered array implementation

◆ **Ordered array implementation:**

- ✓ Set the front as the position of the smallest element and the rear as the position of the largest element.
- ✓ To insert an element, locate the proper position of the new element and shift preceding or succeeding elements by one position.
- ✓ To delete the minimum element, increment the front position.

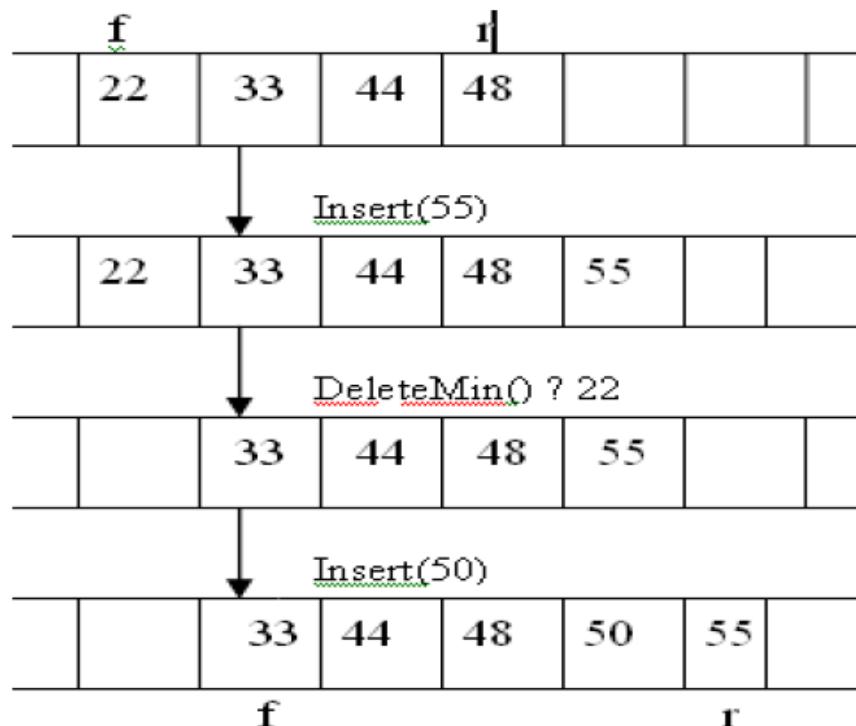


Fig Illustration of ordered array implementation

## **Application of Priority queue:**

In a time-sharing computer system, a large number of tasks may be waiting for the CPU, some of these tasks have higher priority than others. The set of tasks waiting for the CPU forms a priority queue.

**/\*implementation of ascending priority queue \*/**

```
#include<stdio.h>
#include<conio.h>
#define SIZE 20
struct cqueue
{
    int item[SIZE];
    int rear;
    int front;
};
typedef struct queue pq;
void insert(pq* );
void delet(pq* );
void display(pq* );
```

```

void main()
{
    int ch;
    pq *q;
    q->rear=-1;
    q->front=0;
    clrscr();
    printf("Menu for program:\n");
    printf("1:insert\n2:delete\n3:display\n4:exit\n");
    do
    {
        printf("Enter your choice\n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                insert(q);
                break;
            case 2:
                delet(q);
                break;
            case 3:
                display(q);
                break;
            case 4:
                exit(1);
                break;
            default:
                printf("Your choice is wrong\n");
                break;
        }
    }while(ch<5);
    getch();
}

```

```

void insert(pq *q)
{
    int d;
    if(q->rear==SIZE-1)
        printf("Queue is full\n");
    else
    {
        printf ("Enter data to be inserted\n");
        scanf("%d",&d);
        q->rear++;
        q->item[q->rear]=d;
    }
}

*****delete function*****
void delet(pq *q)
{
    int i, temp=0, x;
    x=q->item[q->front];
    if(q->rear<q->front)
    {
        printf("Queue is empty\n");
        return 0;
    }
    else
    {

```

```

        for(i=q->front+1; i<q->rear; i++)
        {
            if(x>q->item[i])
            {
                temp=i;
                x=q->item[i];
            }
        }
        for(i=temp;i< q->rear-1;i++)
        {
            q->item[i]=q->item[i+1];
        }
        q->rear--;
        return x;
    }
}

*****display function*****
void display(pq *q)
{
    int i;
    if(q->rear < q->front)
        printf("Queue is empty\n");
    else
    {
        printf("Items of queue are:\n");
        for(i=(q->front i<=q->rear;i++)
        {
            printf("%d\t",q->item[i]);
        }
    }
}

```

# Thanks You

# DSA(Lecture#5)

Prepared By Bal Krishna Subedi

## Linked List:

- a) Concept and definition
  - b) Inserting and deleting nodes
  - c) Linked implementation of a stack (PUSH / POP)
  - d) Linked implementation of a queue (insert / delete)
  - e) Circular linked list
    - ◆ Stack as a circular list (PUSH / POP)
    - ◆ Queue as a circular list (Insert / delete)
  - f) Doubly linked list (insert / delete)
-

## Self referential structure:

It is sometimes desirable to include within a structure one member that is a pointer to the parent structure type. Hence, a structure which contains a reference to itself is called self-referential structure. In general terms, this can be expressed as:

```
struct node
{
    member 1;
    member 2;
    .....
    struct node *name;
};
```

**For example,**

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

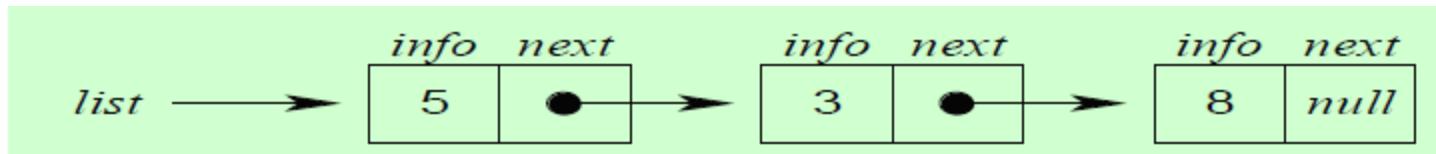
This is a structure of type node. The structure contains two members: a *info* integer member, and a pointer to a structure of the same type (i.e., a pointer to a structure of type node), called next. Therefore this is a ***self-referential*** structure.

## Linked List:

A linked list is a collection of nodes, where each node consists of two parts:

- ◆ **info:** the actual element to be stored in the list. It is also called data field.
- ◆ **link:** one or two links that points to next and previous node in the list. It is also called next or pointer field.

## **Illustration:**



- ◆ The nodes in a linked list are not stored contiguously in the memory
- ◆ You don't have to shift any element in the list.
- ◆ Memory for each node can be allocated dynamically whenever the need arises.
- ◆ The size of a linked list can grow or shrink dynamically

## Operations on linked list:

The basic operations to be performed on the linked list are as follows:

- ◆ **Creation:** This operation is used to create a linked list
- ◆ **Insertion:** This operation is used to insert a new node in a linked list in a specified position. A new node may be inserted
  - ✓ At the beginning of the linked list
  - ✓ At the end of the linked list
  - ✓ At the specified position in a linked list
- ◆ **Deletion:** The deletion operation is used to delete a node from the linked list. A node may be deleted from
  - ✓ The beginning of the linked list
  - ✓ the end of the linked list
  - ✓ the specified position in the linked list.
- ◆ **Traversing:** The list traversing is a process of going through all the nodes of the linked list from one end to the other end. The traversing may be either forward or backward.
- ◆ **Searching or find:** This operation is used to find an element in a linked list. If the desired element is found then we say operation is successful otherwise unsuccessful.
- ◆ **Concatenation:** It is the process of appending second list to the end of the first list.

## Types of Linked List:

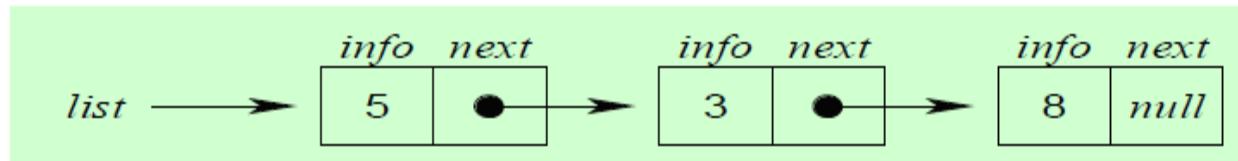
basically we can put linked list into the following four types:

- ◆ Singly linked list
- ◆ doubly linked list
- ◆ circular linked list
- ◆ circular doubly linked list

### Singly linked list:

A singly linked list is a dynamic data structure which may grow or shrink, and growing and shrinking depends on the operation made. In this type of linked list each node contains two fields one is info field which is used to store the data items and another is link field that is used to point the next node in the list. The last node has a NULL pointer.

The following example is a singly linked list that contains three elements 5, 3, 8.



## **Representation of singly linked list:**

We can create a structure for the singly linked list the each node has two members, one is **info** that is used to store the data items and another is **next** field that store the address of next node in the list.

We can define a node as follows:

```
struct Node
{
    int info;
    struct Node *next;
};
```

```
typedef struct Node NodeType;
```

NodeType \*head; //head is a pointer type structure variable

This type of structure is called self-referential structure.

- ◆ *The NULL value of the next field of the linked list indicates the last node and we define macro for NULL and set it to 0 as below:*

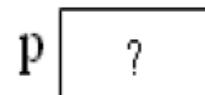
```
#define NULL 0
```

## Creating a Node:

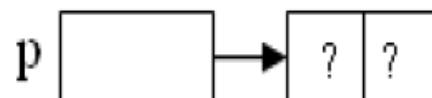
- ◆ To create a new node, we use the **malloc** function to dynamically allocate memory for the new node.
- ◆ After creating the node, we can store the new item in the node using a pointer to that node.

The following steps clearly shows the steps required to create a node and storing an item.

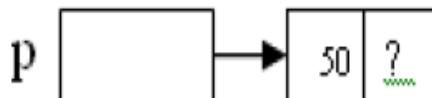
Nodetype \*p;



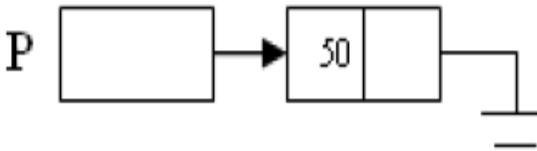
P=(NodeType\*)malloc(sizeof(Nodetype));



p->info=50;



p->next=NULL;



*Note that p is not a node; instead it is a pointer to a node.*

## The getNode function:

we can define a function `getNode()` to allocate the memory for a node dynamically. It is user-defined function that return a pointer to the newly created node.

```
Nodetype *getNode()
{
    NodeType *p;
    p=(NodeType*)malloc(sizeof(NodeType));
    return(p);
}
```

## Creating the empty list:

```
void createEmptyList(NodeType *head)
{
    head=NULL;
}
```

## Inserting Nodes:

To insert an element or a node in a linked list, the following three things to be done:

- ◆ Allocating a node
- ◆ Assigning a data to info field of the node
- ◆ Adjusting a pointer and a new node may be inserted
- ◆ At the beginning of the linked list
- ◆ At the end of the linked list
- ◆ At the specified position in a linked list

Insertion requires obtaining a new node and changing two links

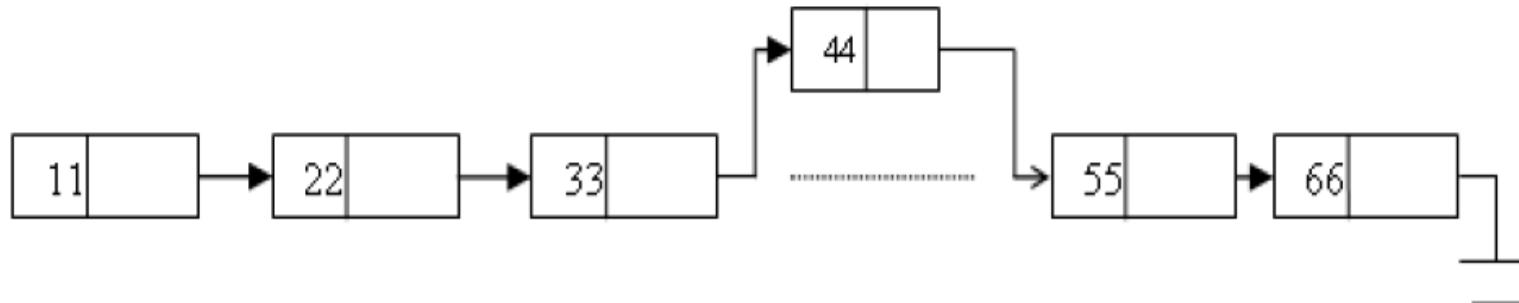


fig:- Inserting the new node with 44 between 33 and 55.

## An algorithm to insert a node at the beginning of the singly linked list:

let \*head be the pointer to first node in the current list

1. Create a new node using malloc function

*NewNode=(NodeType\*)malloc(sizeof(NodeType));*

2. Assign data to the info field of new node

*NewNode->info=newItem;*

3. Set next of new node to head

*NewNode->next=head;*

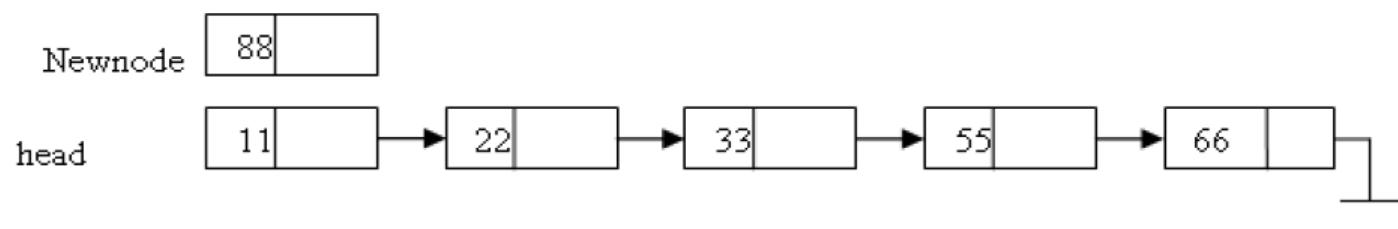
4. Set the head pointer to the new node

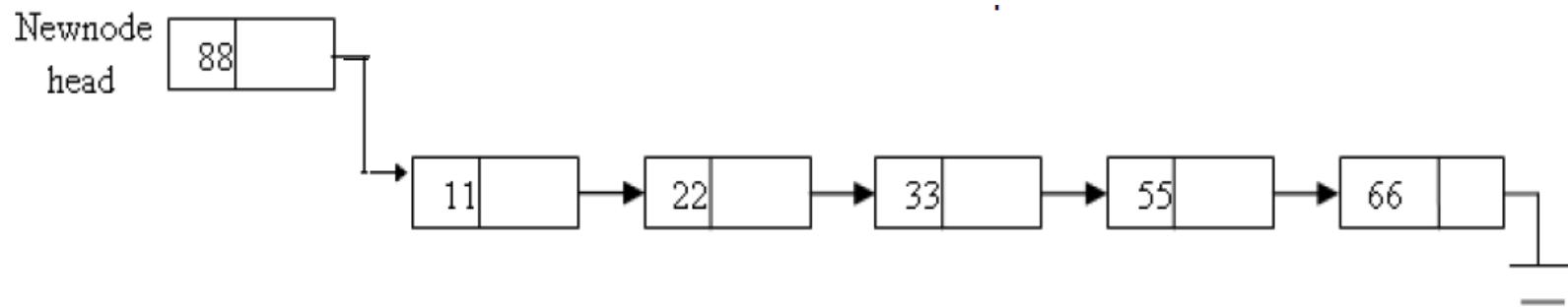
*head=NewNode;*

5. End

## **The C function to insert a node at the beginning of the singly linked list:**

```
void InsertAtBeg(int newItem)
{
    NodeType *NewNode;
    NewNode=getNode();
    NewNode->info=newItem;
    NewNode->next=head;
    head=NewNode;
}
```





### An algorithm to insert a node at the end of the singly linked list:

let `*head` be the pointer to first node in the current list

1. Create a new node using malloc function  

$$\text{NewNode} = (\text{NodeType}^*)\text{malloc}(\text{sizeof}(\text{NodeType}));$$
2. Assign data to the info field of new node  

$$\text{NewNode-}>\text{info} = \text{newItem};$$
3. Set next of new node to NULL  

$$\text{NewNode-}>\text{next} = \text{NULL};$$
4. if (`head == NULL`)then  
 Set `head = NewNode`.and exit.
5. Set `temp = head`;
6. while(`temp->next != NULL`)  

$$\text{temp} = \text{temp-}>\text{next}; \text{//increment temp}$$
7. Set `temp->next = NewNode`;
8. End

## **The C function to insert a node at the end of the linked list:**

```
void InsertAtEnd(int newItem)
{
    NodeType *NewNode;
    NewNode=getNode();
    NewNode->info=newItem;
    NewNode->next=NULL;
    if(head==NULL)
    {
        head=NewNode;
    }
    else
    {
        temp=head;
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        temp->next=NewNode;
    }
}
```

## An algorithm to insert a node after the given node in singly linked list:

let \*head be the pointer to first node in the current list and \*p be the pointer to the node after which we want to insert a new node.

1. Create a new node using malloc function

*NewNode=(NodeType\*)malloc(sizeof(NodeType));*

2. Assign data to the info field of new node

*NewNode->info=newItem;*

3. Set next of new node to next of p

*NewNode->next=p->next;*

4. Set next of p to NewNode

*p->next =NewNode..*

5. End

## The C function to insert a node after the given node in singly linked list:

```
void InsertAfterNode(NodeType *p int newItem)
{
    NodeType *NewNode;
    NewNode=getNode();
    NewNode->info=newItem;
    if(p==NULL)
    {
        printf("Void insertion");
        exit(1);
    }
    else
    {
        NewNode->next=p->next;
        p->next =NewNode..
    }
}
```

## An algorithm to insert a node at the specified position in a singly linked list:

let \*head be the pointer to first node in the current list

1. Create a new node using malloc function

*NewNode=(NodeType\*)malloc(sizeof(NodeType));*

2. Assign data to the info field of new node

*NewNode->info=newItem;*

3. Enter position of a node at which you want to insert a new node. Let this position is pos.

4. Set temp=head;

5. if (head ==NULL)then

*printf(“void insertion”); and exit(1).*

6. for(i=1; i<pos-1; i++)

*temp=temp->next;*

7. Set *NewNode->next=temp->next;*

*set temp->next =NewNode..*

8. End

## The C function to insert a node at the specified position in a singly linked list:

```
void InsertAtPos(int newItem)
{
    NodeType *NewNode;
    int pos , i ;
    printf(" Enter position of a node at which you want to insert a new node");
    scanf("%d",&pos);
    if(head==NULL)
    {
        printf("void insertion");
        exit(1).
    }
    else
    {
        temp=head;
```

```
for(i=1; i<pos-1; i++)
{
    temp=temp->next;
}
NewNode=getNode();
NewNode->info=newItem;
NewNode->next=temp->next;
temp->next =NewNode;
}
}
```

## **Deleting Nodes:**

A node may be deleted:

- ◆ From the beginning of the linked list
- ◆ from the end of the linked list
- ◆ from the specified position in a linked list

## **Deleting first node of the linked list:**

### **An algorithm to deleting the first node of the singly linked list:**

let \*head be the pointer to first node in the current list

1. If(`head==NULL`) then  
    print “Void deletion” and exit
2. Store the address of first node in a temporary variable **temp**.  
    `temp=head;`
3. Set head to next of head.  
    `head=head->next;`
4. Free the memory reserved by temp variable.  
    `free(temp);`
5. End

## The C function to deleting the first node of the singly linked list:

```
void deleteBeg()
{
    NodeType *temp;
    if(head==NULL)
    {
        printf("Empty list");
        exit(1);
    }
    else
    {
        temp=head;
        printf("Deleted item is %d" , head->info);
        head=head->next;
        free(temp);
    }
}
```

## Deleting the last node of the linked list:

### An algorithm to deleting the last node of the singly linked list:

let \*head be the pointer to first node in the current list

1. If(`head==NULL`) then //if list is empty  
    print “Void deletion” and exit
2. else if(`head->next==NULL`) then //if list has only one node  
    Set `temp=head;`  
    print deleted item as,  
    `printf(“%d”,head->info);`  
    `head=NULL;`  
    `free(temp);`
3. else  
    set `temp=head;`  
    **`while`**(`temp->next->next!=NULL`)  
        set `temp=temp->next;`  
    End of **`while`**  
    `free(temp->next);`  
    Set `temp->next=NULL;`
4. End

## The C function to deleting the last node of the singly linked list:

let \*head be the pointer to first node in the current list

```
void deleteEnd()
{
    NodeType *temp;
    if(head==NULL)
    {
        printf("Empty list");
        return;
    }
    else if(head->next==NULL)
    {
        temp=head;
        head=NULL;
        printf("Deleted item is %d", temp->info);
        free(temp);
    }
    else
    {
        temp=head;
        while(temp->next->next!=NULL)
        {
            temp=temp->next;
        }
        printf("deleted item is %d" , temp->next->info);
        free(temp->next);
        temp->next=NULL;
    }
}
```

## An algorithm to delete a node after the given node in singly linked list:

let \*head be the pointer to first node in the current list and \*p be the pointer to the node after which we want to delete a new node.

1. *if*(*p*==NULL or *p*->next==NULL) *then*  
*print* “deletion not possible and exit
2. set *q*=*p*->next
3. *Set p->next=q->next;*
4. **free**(*q*)
5. End

## The C function to delete a node after the given node in singly linked list:

let \*p be the pointer to the node after which we want to delete a new node.

```
void deleteAfterNode(NodeType *p)
{
    NodeType *q;
    if(p==NULL || p->next==NULL )
    {
        printf("Void insertion");
        exit(1);
    }
    else
    {
        q=p->next;
        p->next=q->next;
        free(q);
    }
}
```

## An algorithm to delete a node at the specified position in a singly linked list:

let \*head be the pointer to first node in the current list

1. *Read position of a node which to be deleted, let it be pos.*
2. if head==NULL  
    print “void deletion” and exit
3. Enter position of a node at which you want to delete a new node. Let this position is pos.
4. Set temp=head  
    declare a pointer of a structure let it be \*p
5. if (head ==NULL)then  
    print “void ideletion” and exit  
    otherwise;
6. for(i=1; i<pos-1; i++)  
    temp=temp->next;
7. *print deleted item is temp->next->info*
8. *Set p=temp->next;*
9. *Set temp->next =temp->next->next;*
10. free(p);
11. End

## The C function to delete a node at the specified position in a singly linked list

```
void deleteAtSpecificPos()
{
    NodeType *temp *p;
    int pos, i;
    if(head==NULL)
    {
        printf("Empty list");
        return;
    }
    else
    {
        printf("Enter position of a node which you want to delete");
        scanf("%d", &pos);
        temp=head;
        for(i=1; i<pos-1; i++)
        {
            temp=temp->next;
        }
        p=temp->next;
        printf("Deleted item is %d", p->info);
        temp->next =p->next;
        free(p);
    }
}
```

## Searching an item in a linked list:

To search an item from a given linked list we need to find the node that contain this data item. If we find such a node then searching is successful otherwise searching unsuccessful.

*let \*head be the pointer to first node in the current list*

```
void searchItem()
{
    NodeType *temp;
    int key;
    if(head==NULL)
    {
        printf("empty list");
        exit(1);
    }
    else
    {
        printf("Enter searched item");
        scanf("%d", &key);
        temp=head;
        while(temp!=NULL)
        {
            if(temp->info==key)
            {
                printf("Search successful");
                break;
            }
            temp=temp->next;
        }
        if(temp==NULL)
        printf("Unsuccessful search");
    }
}
```

## Complete program:

```
*****Various operations on singly linked list*****  
#include<stdio.h>  
#include<conio.h>  
#include<malloc.h> //for malloc function  
#include<process.h> //fpr exit function  
struct node  
{  
    int info;  
    struct node *next;  
};  
typedef struct node NodeType;  
NodeType *head;  
head=NULL;  
void insert_atfirst(int);  
void insert_givenposition(int);  
void insert_atend(int);  
void delet_first();  
void delet_last();  
void delet_nthnode();  
void info_sum();  
void count_nodes();  
void main()  
{  
    int choice;  
    int item;  
    clrscr();  
    do  
    {
```

```

        printf("\n manu for program:\n");
        printf("1. insert first \n2.insert at given position \n3 insert at last \n 4:Delete firs
node\n 5:delete last node\n6:delete nth node\n7:count nodes\n8Display items\n10:exit\n");
        printf("enter your choice\n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                printf("Enter item to be inserted");
                scanf("%d", &item)
                insert_atfirst(item);
                break;
            case 2:
                printf("Enter item to be inserted");
                scanf("%d", &item)
                insert_givenposition(item);
                break;
            case 3:
                printf("Enter item to be inserted");
                scanf("%d", &item)
                insert_atend();
                break;
            case 4:
                delet_first();
                break;
            case 5:
                delet_last();
                break;
            case 6:
                delet_nthnode();
                break;
            case 7:
                info_sum();
                break;
            case 8:

```

```
        count_nodes();
        break;
    case 9:
        exit(1);
        break;
    default:
        printf("invalid choice\n");
        break;
    }
}while(choice<10);
getch();
}
```

```
*****function definitions*****  
void insert_atfirst(int item)
{
    NodeType *nnode;
    nnode=(NodeType*)malloc(sizeof(NodeType));
    nnode->info=item;
    nnode->next=head;
    head=nnode;
}
```

```
void insert_givenposition(int item)
{
    NodeType *nnode;
    NodeType *temp;
    temp=head;
    int p,i;
    nnode=( NodeType *)malloc(sizeof(NodeType));
    nnode->info=item;
    if (head==NULL)
    {
        nnode->next=NULL;
        head=nnode;
    }
    else
    {
        printf("Enter Position of a node at which you want to insert an new node\n");
        scanf("%d",&p);
        for(i=1;i<p-1;i++)
        {
            temp=temp->next;
        }
        nnode->next=temp->next;
        temp->next=nnode;
    }
}
```

```
void insert_atend(int item)
{
    NodeType *nnode;
    NodeType *temp;
    temp=head;
    nnode=( NodeType *)malloc(sizeof(NodeType));
    nnode->info=item;

    if(head==NULL)
    {
        nnode->next=NULL;
        head=nnode;
    }
    else
    {
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        nnode->next=NULL;
        temp->next=nnode;
    }
}
```

```

void delet_first()
{
    NodeType *temp;
    if(head==NULL)
    {
        printf("Void deletion\n");
        return;
    }
    else
    {
        temp=head;
        head=head->next;
        free(temp);
    }
}

void delet_last()
{
    NodeType *hold,*temp;
    if(head==NULL)
    {
        printf("Void deletion\n");
        return;
    }
    else if(head->next==NULL)
    {
        hold=head;
        head=NULL;
        free(hold);
    }
    else
    {
        temp=head;
        while(temp->next->next!=NULL)
        {
            temp=temp->next;
        }
        hold=temp->next;
        temp->next=NULL;
        free(hold);
    }
}

```

```
void delet_nthnode()
{
    NodeType *hold,*temp;
    int pos, i;
    if(head==NULL)
    {
        printf("Void deletion\n");
        return;
    }
    else
    {
        temp=head;
        printf("Enter position of node which node is to be deleted\n");
        scanf("%d",&pos);
        for(i=1;i<pos-1;i++)
        {
            temp=temp->next;
        }
        hold=temp->next;
        temp->next=hold->next;
        free(hold);
    }
}
```

```
void info_sum()
{
    NodeType *temp;
    temp=head;
    while(temp!=NULL)
    {
        printf("%d\t",temp->info);
        temp=temp->next;
    }
}
void count_nodes()
{
    int cnt=0;
    NodeType *temp;
    temp=head;
    while(temp!=NULL)
    {
        cnt++;
        temp=temp->next;
    }
    printf("total nodes=%d",cnt);
}
```

## Linked list implementation of Stack:

### Push function:

let \*top be the top of the stack or pointer to the first node of the list.

```
void push(item)
{
    NodeType *nnode;
    int data;
    nnode=( NodeType *)malloc(sizeof(NodeType));
    if(top==0)
    {
        nnode->info=item;
        nnode->next=NULL;
        top=nnode;
    }
    else
    {
        nnode->info=item;
        nnode->next=top;
        top=nnode;
    }
}
```

## Pop function:

let \*top be the top of the stack or pointer to the first node of the list.

```
void pop()
{
    NodeType *temp;
    if(top==0)
    {
        printf("Stack contain no elements:\n");
        return;
    }
    else
    {
        temp=top;
        top=top->next;
        printf("\ndeleted item is %d\t",temp->info);
        free(temp);
    }
}
```

## A Complete C program for linked list implementation of stack:

```
*****Linked list implementation of stack*****
```

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<process.h>
struct node
{
    int info;
    struct node *next;
};
typedef struct node NodeType;
NodeType *top;
top=0;
void push(int);
void pop();
void display();
void main()
{
    int choice, item;
    clrscr();
```

```
do
{
    printf("\n1.Push \n2.Pop \n3.Display\n4:Exit\n");
    printf("enter ur choice\n");
    scanf("%d",&choice);
    switch(choice)
    {
        case 1:
            printf("\nEnter the data:\n");
            scanf("%d",&item);
            push(item);
            break;
        case 2:
            pop();
            break;
        case 3:
            display();
            break;
        case 4:
            exit(1);
            break;
        default:
            printf("invalid choice\n");
            break;
    }
}while(choice<5);
getch();
}
```

```
*****push function*****
void push(int item)
{
    NodeType *nnode;
    int data;
    nnode=( NodeType *)malloc(sizeof(NodeType));
    if(top==0)
    {
        nnode->info=item;
        nnode->next=NULL;
        top=nnode;
    }
    else
    {
        nnode->info=item;
        nnode->next=top;
        top=nnode;
    }
}
*****pop function*****
void pop()
{
    NodeType *temp;
    if(top==0)
    {
        printf("Stack contain no elements:\n");
        return;
    }
}
```

```

        }
    else
    {
        temp=top;
        top=top->next;
        printf("\ndeleted item is %d\t",temp->info);
        free(temp);
    }
}

*****display function*****
void display()
{
    NodeType *temp;
    if(top==0)
    {
        printf("Stack is empty\n");
        return;
    }
    else
    {
        temp=top;
        printf("Stack items are:\n");
        while(temp!=0)
        {
            printf("%d\t",temp->info);
            temp=temp->next;
        }
    }
}

```

# Thank You

Any Queries?

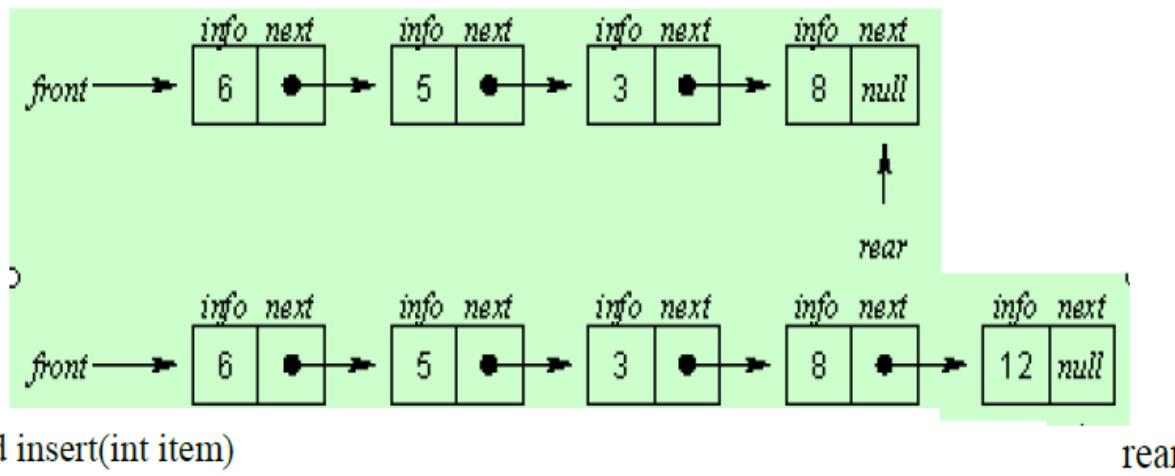
# DSA(Lecture#6)

Prepared By  
Bal Krishna Subedi

## Linked list implementation of queue:

### Insert function:

let \*rear and \*front are pointers to the first node of the list initially and insertion of node in linked list done at the rear part and deletion of node from the linked list done from front part.



```
void insert(int item)
{
    NodeType *nnode;
    nnode=( NodeType * )malloc(sizeof(NodeType));

    if(rear==0)
    {
        nnode->info=item;
        nnode->next=NULL;
```

```

        rear=front=nnode;
    }
else
{
    nnode->info=item;
    nnode->next=NULL;
    rear->next=nnode;
    rear=nnode;
}
}

```

**Delete function:**

let \*rear and \*front are pointers to the first node of the list initially and insertion of node in linked list done at the rear part and deletion of node from the linked list done from front part.

```

void delet()
{
    NodeType *temp;
    if(front==0)
    {
        printf("Queue contain no elements:\n");
        return;
    }
    else if(front->next==NULL)
    {
        temp=front;
        rear=front=NULL;
        printf("\nDeleted item is %d\n",temp->info);
        free(temp);
    }
}

```

```
    else
    {
        temp=front;
        front=front->next;
        printf("\nDeleted item is %d\n",temp->info);
        free(temp);
    }
}
```

### A Complete C program for linked list implementation of queue:

```
*****Linked list implementation of queue*****
```

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<process.h>
struct node
{
    int info;
    struct node *next;
};
typedef struct node NodeType;
NodeType *rear,*front;
rear=front=0;
void insert(int);
```

```

void delet();
void display();
void main()
{
    int choice, item;
    clrscr();
    do
    {
        printf("\n1.Insert \n2.Delet \n3.Display\n4:Exit\n");
        printf("enter ur choice\n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                printf("\nEnter the data:\n");
                scanf("%d",&item);
                insert(item);
                break;
            case 2:
                delet();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(1);
                break;
            default:
                printf("invalid choice\n");
                break;
        }
    }while(choice<5);
    getch();
}

```

```
*****insert function*****
void insert(int item)
{
    NodeType *nnode;
    nnode=( NodeType *)malloc(sizeof(NodeType));

    if(rear==0)
    {
        nnode->info=item;
        nnode->next=NULL;
        rear=front=nnode;
    }
    else
    {
        nnode->info=item;
        nnode->next=NULL;
        rear->next=nnode;
        rear=nnode;
    }
}
```

```
void delet()
{
    NodeType *temp;
    if(front==0)
    {
        printf("Queue contain no elements:\n");
        return;
    }
    else if(front->next==NULL)
    {
        temp=front;
        rear=front=NULL;
        printf("\nDeleted item is %d\n",temp->info);
        free(temp);
    }
    else
    {
        temp=front;
        front=front->next;
        printf("\nDeleted item is %d\n",temp->info);
        free(temp);
    }
}
```

```

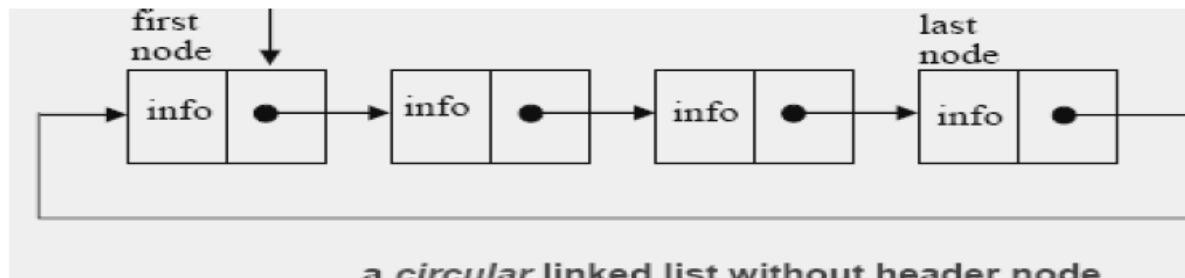
void display()
{
    NodeType *temp;
    temp=front;
    printf("\nqueue items are:\t");
    while(temp!=NULL)
    {
        printf("%d\t",temp->info);
        temp=temp->next;
    }
}

```

### Circular Linked list:

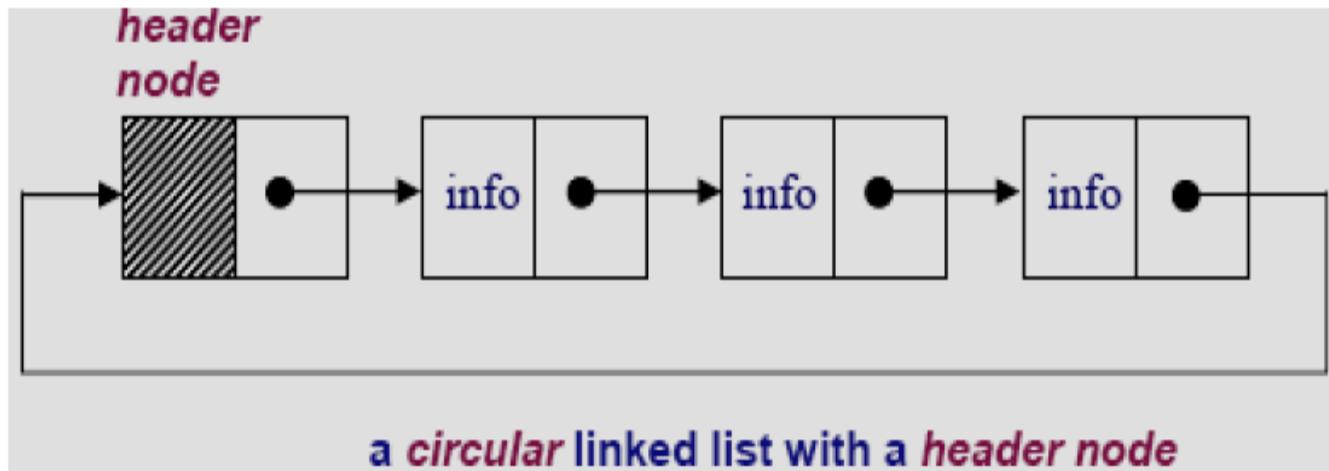
A circular linked list is a list where the link field of last node points to the very first node of the list .

Circular linked lists can be used to help the traverse the same list again and again if needed. A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node.

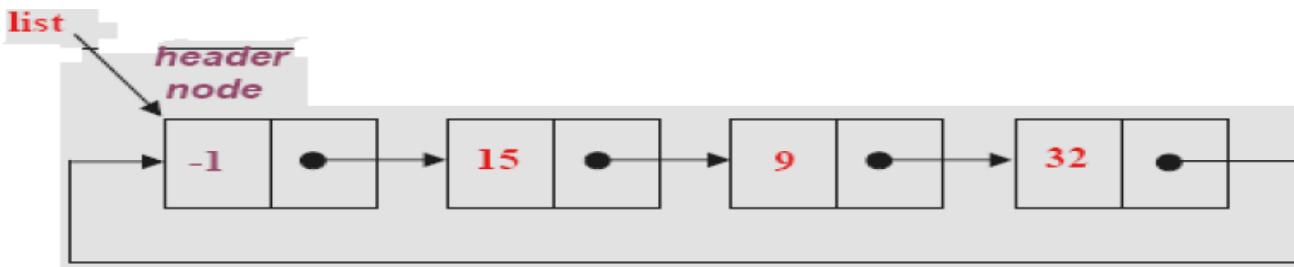


In a circular linked list there are two methods to know if a node is the first node or not.

- ◆ Either a external pointer, *list*, points the first node or
  - ◆ A ***header node*** is placed as the first node of the circular list.
- The header node can be separated from the others by either heaving a ***sentinel value*** as the info part or having a dedicated ***flag*** variable to specify if the node is a header node or not.



### **CIRCULAR LIST with header node**



## C representation of circular linked list:

we declare the structure for the circular linked list in the same way as declared it for the linear linked list.

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

typedef struct node NodeType;
NodeType *start=NULL;
NodeType *last=NULL;
```

## Algorithms to insert a node in a circular linked list:

### Algorithm to insert a node at the beginning of a circular linked list:

1. Create a new node as

```
newnode=(NodeType*)malloc(sizeof(NodeType));
```

2. if start==NULL then

```
    set newnode->info=item
    set newnode->next=newnode
    set start=newnode
    set last newnode
```

end if

```
3. else
    set newnode->info=item
    set newnode->next=start
    set start=newnode
    set last->next=newnode
end else
4. End
```

### **Algorithm to insert a node at the end of a circular linked list:**

```
1. Create a new node as
    newnode=(NodeType*)malloc(sizeof(NodeType));
2. if start==NULL then
    set newnode->info=item
    set newnode->next=newnode
    set start=newnode
    set last=newnode
end if
3. else
    set newnode->info=item
    set last->next=newnode
    set last=newnode
    set last->next=start
end else
4. End
```

## C function to insert a node at the beginning of a circular linked list:

```
void InsertAtBeg(int Item)
{
    NodeType *newnode;
    newnode=(NodeType*)malloc(sizeof(NodeType));
    if(start==NULL)
    {
        newnode->info=item;
        newnode->next=newnode;
        start=newnode;
        last=newnode;
    }
    else
    {
        newnode->info=item;
        last->next=newnode;
        last=newnode;
        last->next=start;
    }
}
```

## C function to insert a node at the end of a circular linked list:

```
void InsertAtEnd(int Item)
{
    NodeType *newnode;
    newnode=(NodeType*)malloc(sizeof(NodeType));
    if(start==NULL)
    {
        newnode->info=item;
        newnode->next=newnode;
        start=newnode;
        last=newnode;
    }
    else
    {
        newnode->info=item;
        last->next=newnode;
        last=newnode;
        last->next=start;
    }
}
```

## Algorithms to delete a node from a circular linked list:

### Algorithm to delete a node from the beginning of a circular linked list:

```
1. if start==NULL then  
    “empty list” and exit  
2. else  
    set temp=start  
    set start=start->next  
    print the deleted element=temp->info  
    set last->next=start;  
    free(temp)  
end else  
3. End
```

## Algorithm to delete a node from the end of a circular linked list:

1. if start==NULL then  
    “empty list” and exit
2. else if start==last  
    set temp=start  
    print deleted element=temp->info  
    free(temp)  
    start=last=NULL
3. else  
    set temp=start  
    while( temp->next!=last)  
        set temp=temp->next  
    end while  
    set hold=temp->next  
    set last=temp  
    set last->next=start  
    print the deleted element=hold->info  
    free(hold)
4. End

## C function to delete a node from the beginning of a circular linked list:

```
void DeleteFirst()
{
    if(start==NULL)
    {
        printf("Empty list");
        exit(1);
    }
    else
    {
        temp=start;
        start=start->next;
        printf(" the deleted element=%d", temp->info);
        last->next=start;
        free(temp)

    }
}
```

## C function to delete a node from the end of a circular linked list:

```
void DeleteLast()
{
    if(start==NULL)
    {
        printf("Empty list");
        exit(1);
    }
    else if(start==last) //for only one node
    {
        temp=start;
        printf("deleted element=%d", temp->info);
        free(temp);
        start=last=NULL;
    }
    else
    {
        temp=start;
        while( temp->next!=last)
            temp=temp->next;
        hold=temp->next;
        last=temp;
        last->next=start;
        printf("the deleted element=%d", hold->info);
        free(hold);
    }
}
```

## Stack as a circular List:

To implement a stack in a circular linked list, let pstack be a pointer to the last node of a circular list. Actually there is no any end of a list but for convention let us assume that the first node(rightmost node of a list) is the top of the stack.  
An empty stack is represented by a null list.

*The structure for the circular linked list implementation of stack is:*

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

typedef struct node NodeType;
NodeType *pstack=NULL;
```

C function to check whether the list is empty or not as follows:

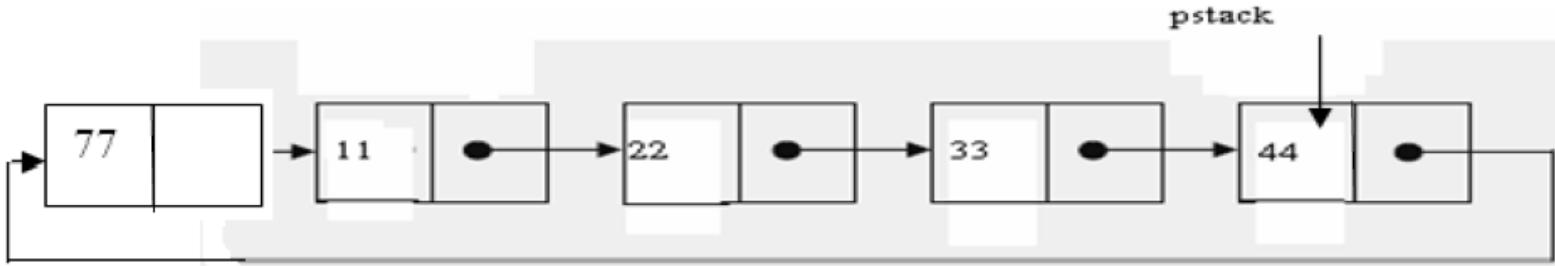
```
int IsEmpty()
{
    if(pstack==NULL)
        return(1);
    else
        return(0);
}
```

## PUSH function:

```
void PUSH(int item)
{
    NodeType newnode;
    newnode=(NodeType*)malloc(sizeof(NodeType));
    newnode->info=item;
    if(pstack==NULL)
    {
        pstack=newnode;
        pstack->next=pstack;
    }
    else
    {
        newnode->next=pstack->next;
        pstack->next=newnode;
    }
}
```



fig: circular linked list



### **POP function:**

```

void POP()
{
    NodeType *temp;
    if(pstack==NULL)
    {
        printf("Stack underflow\n");
        exit(1);
    }
    else if(pstack->next==pstack) //for only one node
    {
        printf("poped item=%d", pstack->info);
        pstack=NULL;
    }
    else
    {
        temp=pstack->next;
        pstack->next=temp->next;
        printf("poped item=%d", temp->info);
        free(temp);
    }
}

```

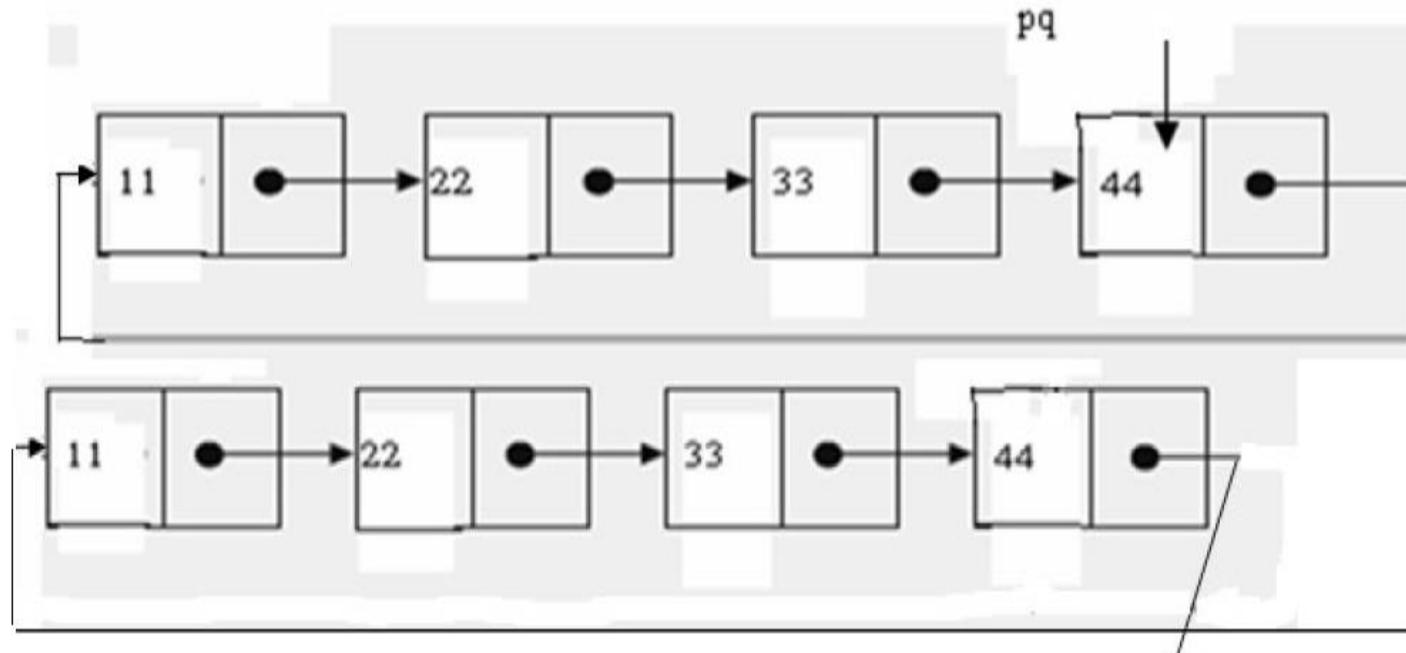
## Queue as a circular List:

It is easier to represent a queue as a circular list than as a linear list. As a linear list a queue is specified by two pointers, one to the front of the list and the other to its rear. However, by using a circular list, a queue may be specified by a single pointer q to that list. node(q) is the rear of the queue and the following node is its front.

### Insertion function:

```
void insert(int item)
{
    NodeType *nnode;
    nnode=( NodeType *)malloc(sizeof(NodeType));
    nnode->info=item;
    if(pq==NULL)
        pq=nnode;
    else
    {
        nnode->next=pq->next;
        pq->next=nnode;
        pq=nnode;
    }
}
```

## Deletion function:



```
void delete(int item)
{
    NodeType *temp;
    if(pq==NULL)
    {
        printf("void deletion\n");
        exit(1);
    }
    else if(pq->next==pq) //for only one node
    {
        printf("poped item=%d", pq->info);
        pq=NULL;
    }
}
```

```

else
{
    temp=pq->next;
    pq->next=temp->next;
    printf("poped item=%d", temp->info);
    free(temp);
}
}

```

## **Doubly Linked List:**

A linked list in which all nodes are linked together by multiple number of links ie each node contains three fields (two pointer fields and one data field) rather than two fields is called doubly linked list.

It provides bidirectional traversal.

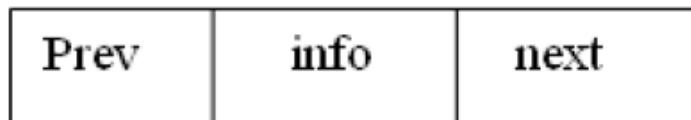


Fig: A node in doubly linked list

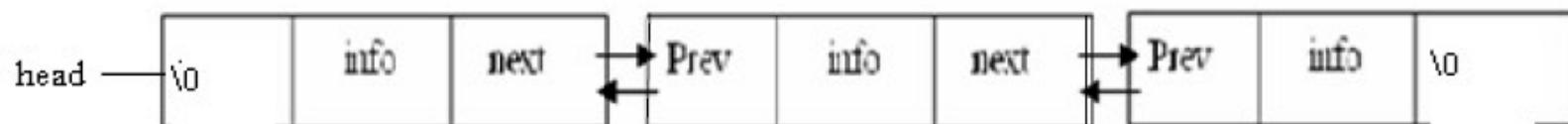


fig: A doubly linked list with three nodes

## C representation of doubly linked list:

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

typedef struct node NodeType;
NodeType *head=NULL;
```

## Algorithms to insert a node in a doubly linked list:

### Algorithm to insert a node at the beginning of a doubly linked list:

1. Allocate memory for the new node as,  
newnode=(NodeType\*)malloc(sizeof(NodeType))
2. Assign value to info field of a new node  
set newnode->info=item
3. set newnode->prev=newnode->next=NULL
4. set newnode->next=head
5. set head->prev=newnode
6. set head=newnode
7. End

### C function to insert a node at the beginning of a doubly linked list:

```
void InsertAtBeg(int Item)
{
    NodeType *newnode;
    newnode=(NodeType*)malloc(sizeof(NodeType));
    newnode->info=item;
    newnode->prev=newnode->next=NULL;
    newnode->next=head;
    head->prev=newnode;
    head=newnode;
}
```

### Algorithm to insert a node at the end of a doubly linked list:

1. Allocate memory for the new node as,  
newnode=(NodeType\*)malloc(sizeof(NodeType))
2. Assign value to info field of a new node  
set newnode->info=item
3. set newnode->next=NULL
4. if head==NULL  
    set newnode->prev=NULL;  
    set head=newnode;
5. if head!=NULL  
    set temp=head  
    while(temp->next!=NULL)  
        temp=temp->next;  
    end while  
    set temp->next=newnode;  
    set newnode->prev=temp
6. End

### Algorithm to delete a node from beginning of a doubly linked list:

```
1. if head==NULL then  
    print "empty list" and exit  
2. else  
    set hold=head  
    set head=head->next  
    set head->prev=NULL;  
    free(hold)  
3. End
```

### Algorithm to delete a node from end of a doubly linked list:

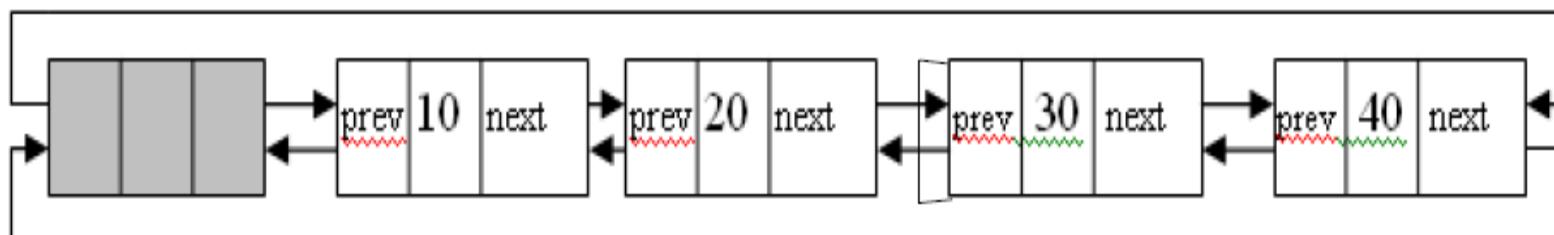
```
1. if head==NULL then  
    print "empty list" and exit  
2. else if(head->next==NULL) then  
    set hold=head  
    set head=NULL  
    free(hold)  
3. else  
    set temp=head;  
    while(temp->next->next !=NULL)  
        temp=temp->next  
    end while  
    set hold=temp->next  
    set temp->next=NULL  
    free(hold)  
4. End
```

## Circular Doubly Linked List:

A circular doubly linked list is one which has the successor and predecessor pointer in circular manner.

It is a doubly linked list where the next link of last node points to the first node and previous link of first node points to last node of the list.

The main objective of considering circular doubly linked list is to simplify the insertion and deletion operations performed on doubly linked list.



head node

Fig: A circular doubly linked list

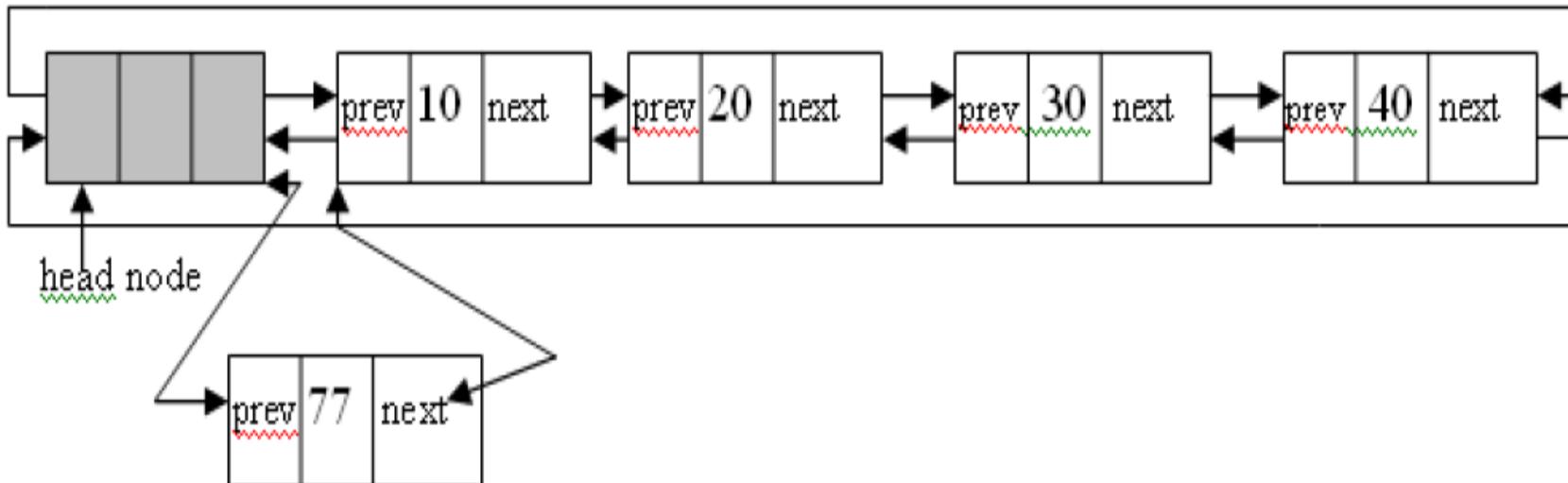
## C representation of doubly circular linked list:

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

typedef struct node NodeType;
NodeType *head=NULL;
```

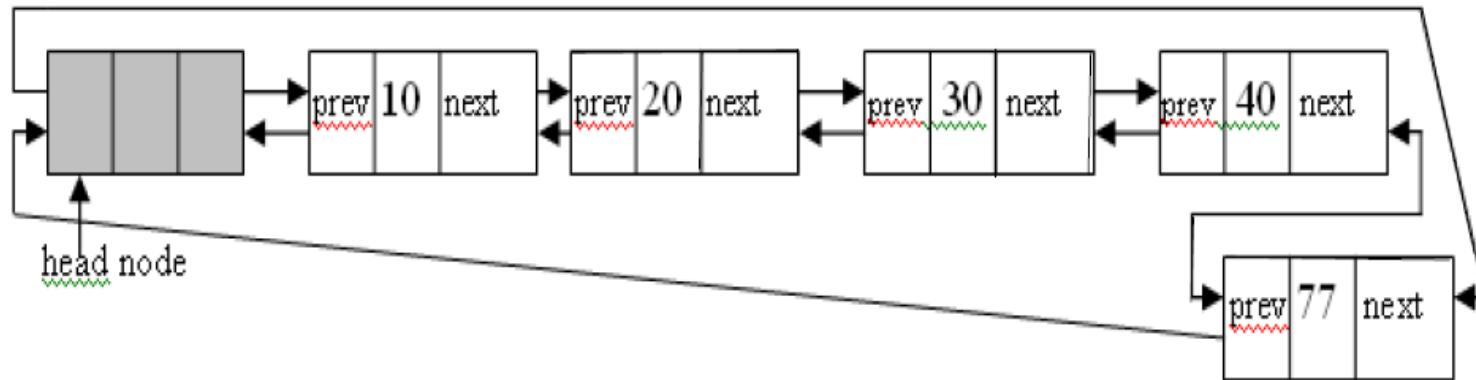
## Algorithm to insert a node at the beginning of a circular doubly linked list:

1. Allocate memory for the new node as,  
newnode=(NodeType\*)malloc(sizeof(NodeType))
2. Assign value to info field of a new node  
set newnode->info=item
3. set temp=head->next
4. set head->next=newnode
5. set newnode->prev=head
6. set newnode->next=temp
7. set temp->prev=newnode
8. End



### Algorithm to insert a node at the end of a circular doubly linked list:

1. Allocate memory for the new node as,  
`newnode=(NodeType*)malloc(sizeof(NodeType))`
2. Assign value to info field of a new node  
`set newnode->info=item`
3. set `temp=head->prev`
4. set `temp->next=newnode`
5. set `newnode->prev=temp`
6. set `newnode->next=head`
7. set `head->prev=newnode`
8. End



**Algorithm to delete a node from the beginning of a circular doubly linked list:**

1. if head->next==NULL then  
print “empty list” and exit
2. else  
    set temp=head->next;  
    set head->next=temp->next  
    set temp->next=head  
    free(temp)
3. End

**Algorithm to delete a node from the end of a circular doubly linked list:**

1. if head->next==NULL then  
print “empty list” and exit
2. else  
    set temp=head->prev;  
    set head->left=temp->left  
    free(temp)
3. End

# Thanks You

Any Queries



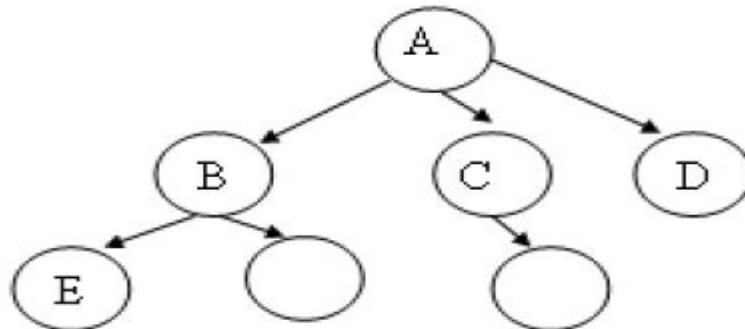
# DSA(Lecture#7)

Prepared by Bal Krishna Subedi

## Tree:

A tree is an abstract model of a hierarchical structure that consists of nodes with a parent-child relationship.

- Tree is a sequence of nodes.
- There is a starting node known as root node.
- Every node other than the root has a parent node.
- Nodes may have any number of children.



A has 3 children, B, C, D  
A is parent of B

## Some key terms:

### Degree of a node:

The degree of a node is the number of children of that node.  
In above tree the degree of node A is 3.

### Degree of a Tree:

The degree of a tree is the maximum degree of nodes in a given tree.  
In the above tree the node A has maximum degree, thus the degree of the tree is 3.

### Path:

It is the sequence of consecutive edges from source node to destination node.  
There is a single unique path from the root to any node.

### Height of a node:

The height of a node is the maximum path length from that node to a leaf node. A leaf node has a height of 0.

### Height of a tree:

The height of a tree is the height of the root.

### Depth of a node:

Depth of a node is the path length from the root to that node. The root node has a depth of 0.

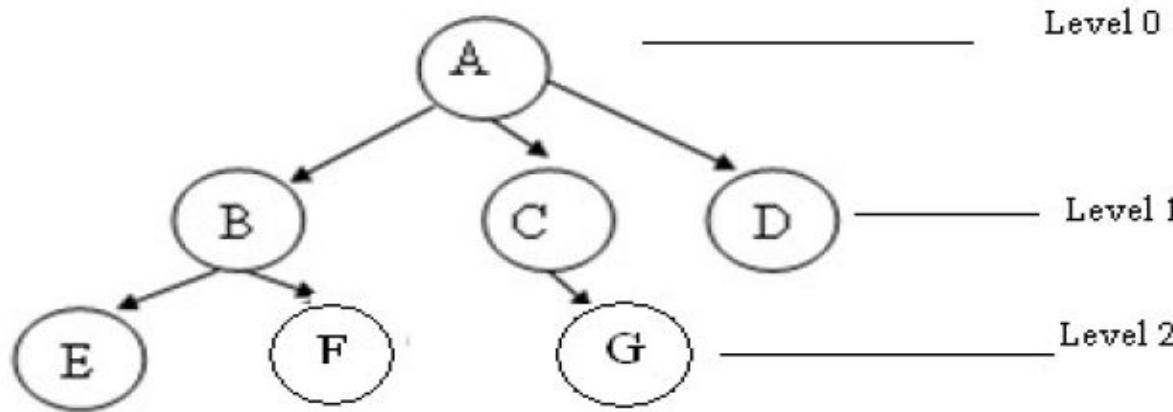
### Depth of a tree:

Depth of a tree is the maximum level of any leaf in the tree.  
This is equal to the longest path from the root to any leaf.

### Level of a node:

the level of a node is 0, if it is root; otherwise it is one more than its parent.

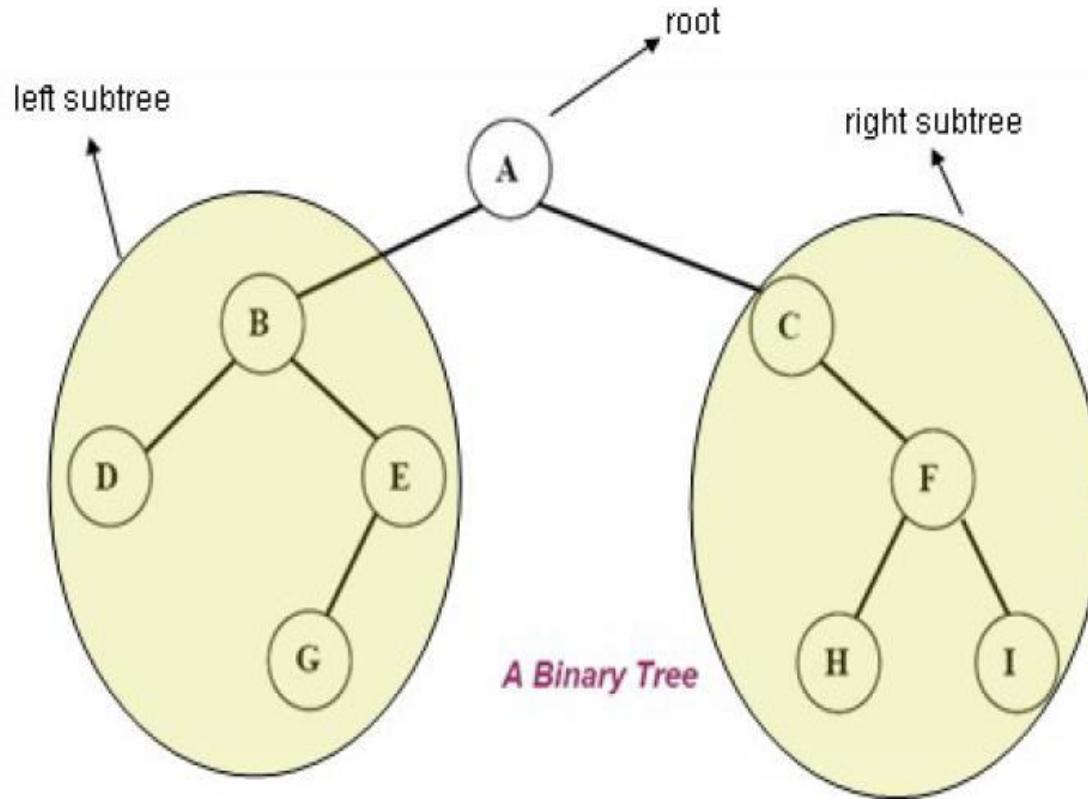
## Illustration:



- ✓ A is the root node
- ✓ B is the parent of E and F
- ✓ D is the sibling of B and C
- ✓ E and F are children of B
- ✓ E, F, G, D are external nodes or leaves
- ✓ A, B, C are internal nodes
- ✓ Depth of F is 2
- ✓ the height of tree is 2
- ✓ the degree of node A is 3
- ✓ The degree of tree is 3

two subsets are themselves binary trees called the *left* and *right sub-trees* of the original tree. A left or right sub tree can be empty.

Each element of a binary tree is called a *node* of the tree. The following figure shows a binary tree with 9 nodes where A is the root.



- A **binary tree** consists of a **header**, plus a number of **nodes** connected by **links** in a hierarchical data structure:

## Binary tree properties:

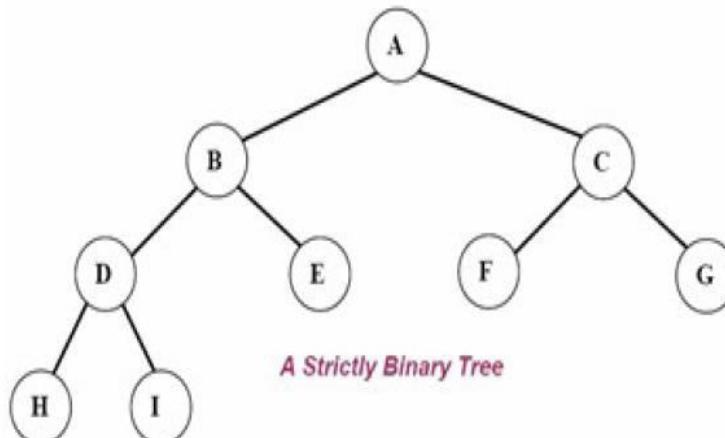
- ✓ If a binary tree contains  $m$  nodes at level 1, it contains at most  $2m$  nodes at level  $l+1$ .
- ✓ Since a binary tree can contain at most 1 node at level 0 (the root), it contains at most  $2^l$  nodes at level  $l$ .

## Types of binary tree

- ✓ Complete binary tree
- ✓ Strictly binary tree
- ✓ Almost complete binary tree

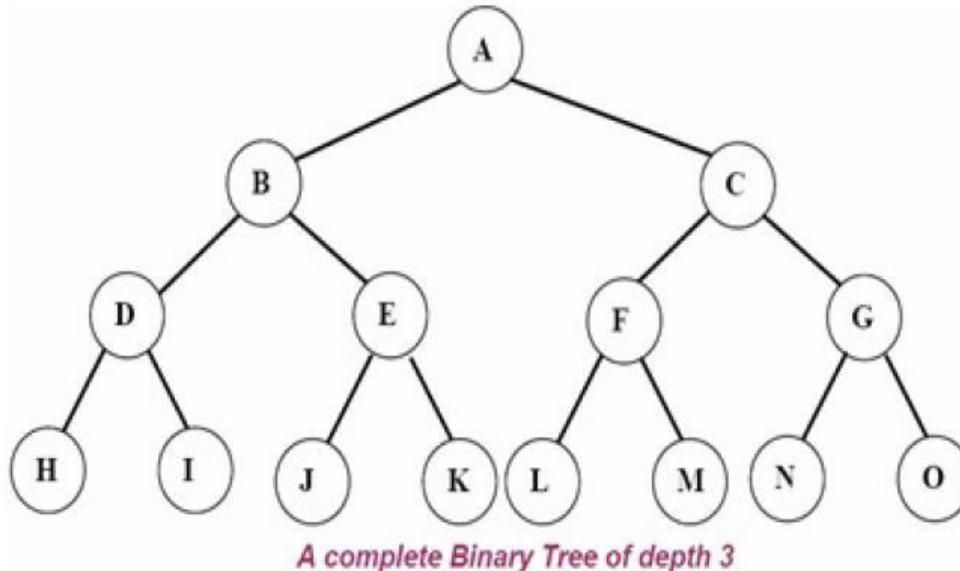
### Strictly binary tree:

If every non-leaf node in a binary tree has nonempty left and right sub-trees, then such a tree is called a *strictly binary tree*.



## Complete binary tree:

A *complete binary tree* of depth  $d$  is called strictly binary tree if all of whose leaves are at level  $d$ . A complete binary tree with depth  $d$  has  $2^d$  leaves and  $2^d - 1$  non-leaf nodes(internal)



## Almost complete binary tree:

A binary tree of depth  $d$  is an almost complete binary tree if:

- ✓ Any node  $nd$  at level less than  $d-1$  has two sons.
- ✓ For any node  $nd$  in the tree with a right descendant at level  $d$ ,  $nd$  must have a left son and every left descendant of  $nd$  is either a leaf at level  $d$  or has two sons.

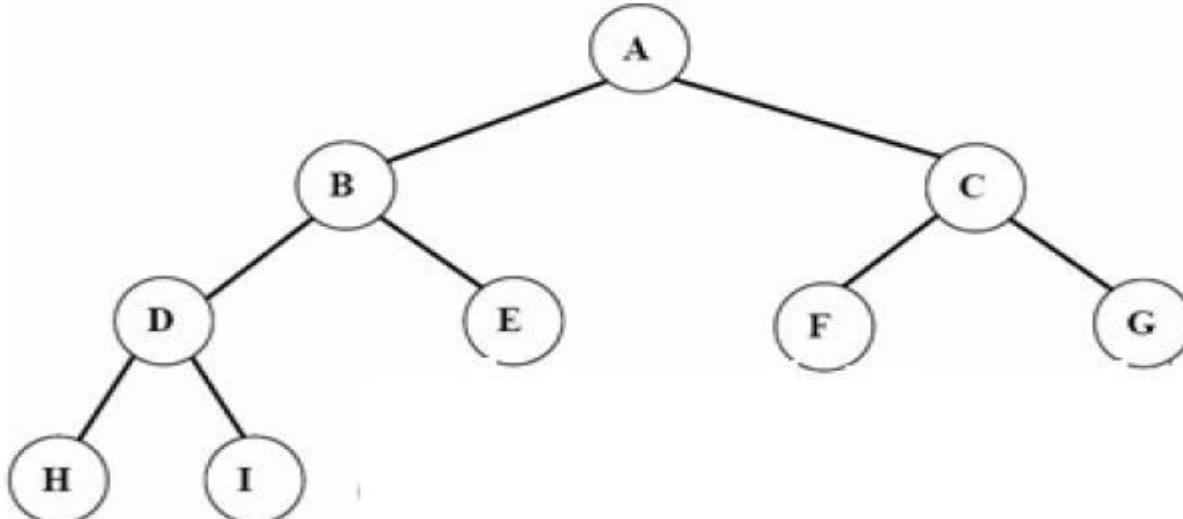


Fig Almost complete binary tree.

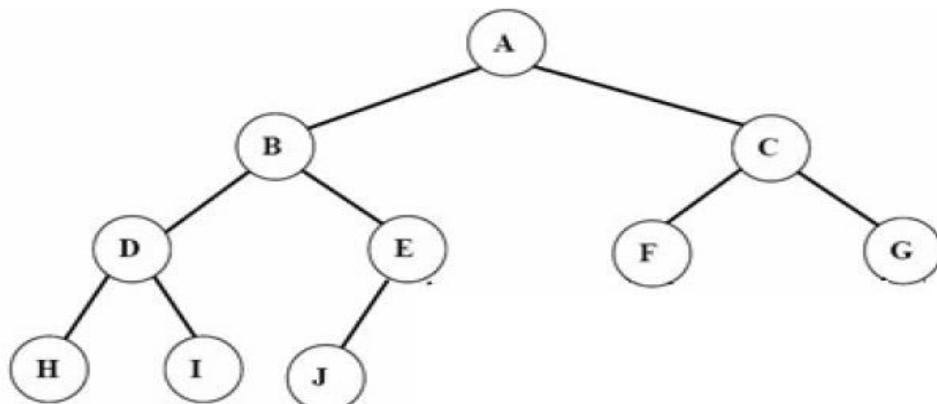


Fig Almost complete binary tree but not strictly binary tree.  
Since node E has a left son but not a right son.

## Operations on Binary tree:

- ✓ **father(n,T):**Return the parent node of the node n in tree T. If n is the root, NULL is returned.
- ✓ **LeftChild(n,T):**Return the left child of node n in tree T. Return NULL if n does not have a left child.
- ✓ **RightChild(n,T):**Return the right child of node n in tree T. Return NULL if n does not have a right child.
- ✓ **Info(n,T):** Return information stored in node n of tree T (ie. Content of a node).
- ✓ **Sibling(n,T):** return the sibling node of node n in tree T. Return NULL if n has no sibling.
- ✓ **Root(T):** Return root node of a tree if and only if the tree is nonempty.
- ✓ **Size(T):** Return the number of nodes in tree T
- ✓ **MakeEmpty(T):** Create an empty tree T
- ✓ **SetLeft(S,T):** Attach the tree S as the left sub-tree of tree T
- ✓ **SetRight(S,T):** Attach the tree S as the right sub-tree of tree T.
- ✓ **Preorder(T):** Traverses all the nodes of tree T in preorder.
- ✓ **postorder(T):** Traverses all the nodes of tree T in postorder
- ✓ **Inorder(T):** Traverses all the nodes of tree T in inorder.

## C representation for Binary tree:

```
struct bnode
{
    int info;
    struct bnode *left;
    struct bnode *right;
};

struct bnode *root=NULL;
```

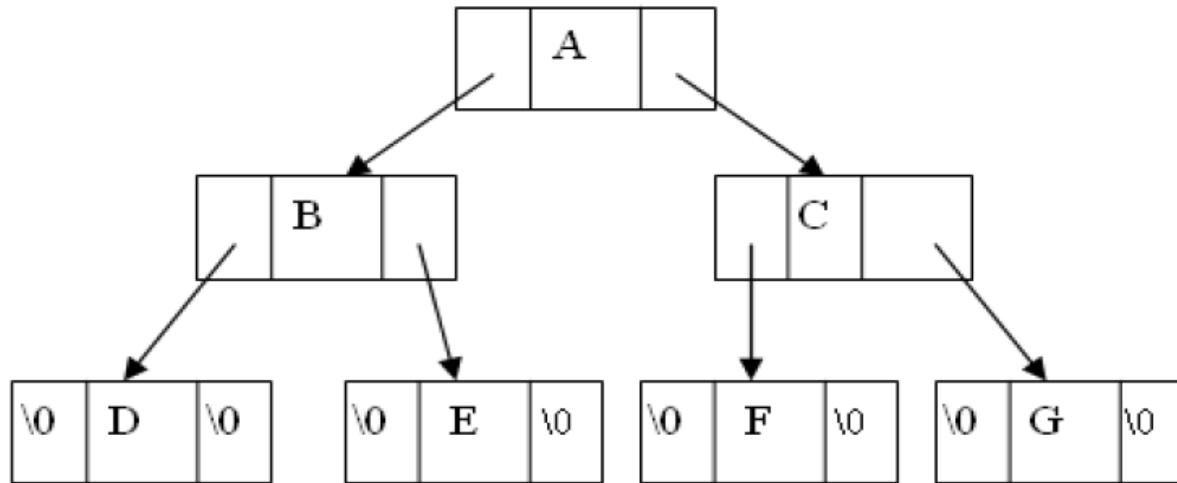


Fig: Structure of Binary tree

## Tree traversal:

The tree traversal is a way in which each node in the tree is visited exactly once in a symmetric manner.

There are three popular methods of traversal

- ✓ Pre-order traversal
- ✓ In-order traversal
- ✓ Post-order traversal

## Pre-order traversal:

The pre-order traversal of a nonempty binary tree is defined as follows:

- ✓ Visit the root node
- ✓ Traverse the left sub-tree in pre-order
- ✓ Traverse the right sub-tree in pre-order

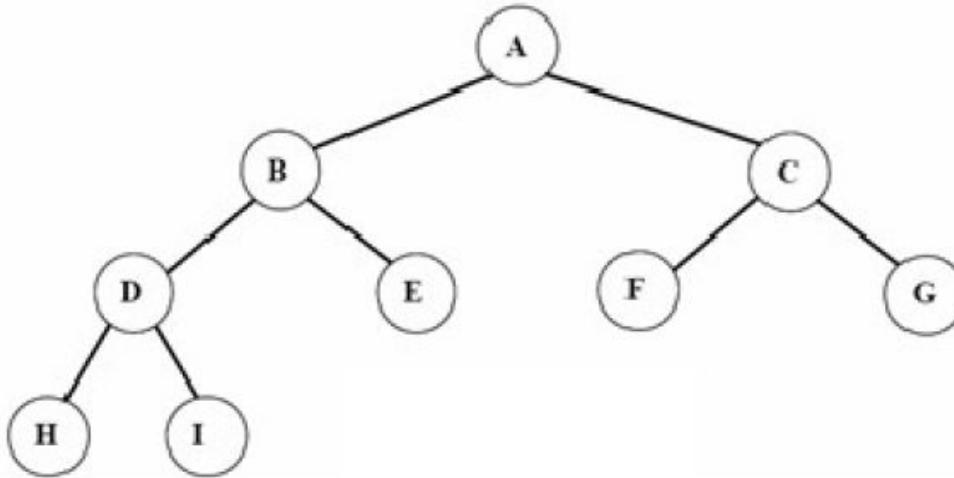


fig Binary tree

The preorder traversal output of the given tree is: A B D H I E C F G

The preorder is also known as depth first order.

### **C function for preorder traversing:**

```
void preorder(struct bnode *root)
{
    if(root!=NULL)
    {
        printf("%c", root->info);
        preorder(root->left);
        preorder(root->right);
    }
}
```

## In-order traversal:

The inorder traversal of a nonempty binary tree is defined as follows:

- ✓ Traverse the left sub-tree in inorder
- ✓ Visit the root node
- ✓ Traverse the right sub-tree in inorder

The inorder traversal output of the given tree is: H D I B E A F C G

## C function for inorder traversing:

```
void inorder(struct bnode *root)
{
    if(root!=NULL)
    {
        inorder(root->left);
        printf("%c", root->info);
        inorder(root->right);
    }
}
```

## **Post-order traversal:**

The post-order traversal of a nonempty binary tree is defined as follows:

- ✓ Traverse the left sub-tree in post-order
- ✓ Traverse the right sub-tree in post-order
- ✓ Visit the root node

The post-order traversal output of the given tree is: H I D E B F G C A

## **C function for post-order traversing:**

```
void post-order(struct bnode *root)
{
    if(root!=NULL)
    {
        post-order(root->left);
        post-order(root->right);
        printf("%c", root->info);
    }
}
```

## **Binary search tree(BST):**

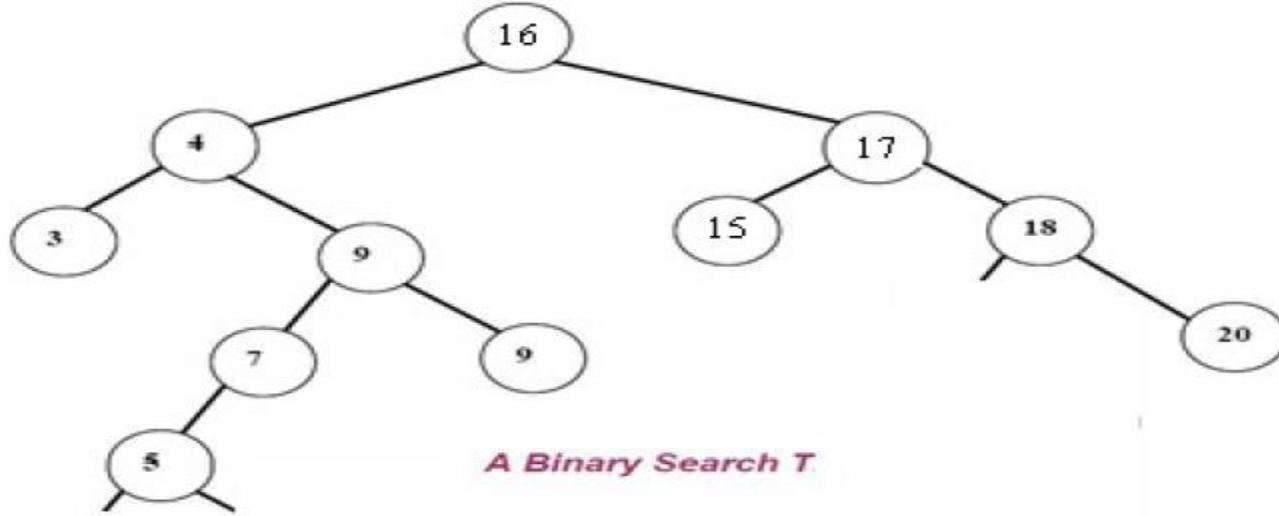
A binary search tree (BST) is a binary tree that is either empty or in which every node contains a key (value) and satisfies the following conditions:

- ✓ All keys in the left sub-tree of the root are smaller than the key in the root node
- ✓ All keys in the right sub-tree of the root are greater than the key in the root node
- ✓ The left and right sub-trees of the root are again binary search trees

Given the following sequence of numbers,

14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

The following binary search tree can be constructed:



## Operations on Binary search tree(BST):

Following operations can be done in BST:

- ✓ **Search(k, T):** Search for key k in the tree T. If k is found in some node of tree then return true otherwise return false.
- ✓ **Insert(k, T):** Insert a new node with value k in the info field in the tree T such that the property of BST is maintained.
- ✓ **Delete(k, T):** Delete a node with value k in the info field from the tree T such that the property of BST is maintained.
- ✓ **FindMin(T), FindMax(T):** Find minimum and maximum element from the given nonempty BST.

## Searching through the BST:

• **Problem:** Search for a given target value in a BST.

• **Idea:** Compare the target value with the element in the root node.

- ✓ If the target value is **equal**, the search is successful.
- ✓ If target value is **less**, search the left subtree.
- ✓ If target value is **greater**, search the right subtree.
- ✓ If the subtree is **empty**, the search is unsuccessful.

## BST search algorithm:

To find which if any node of a BST contains an element equal to *target*:

1. Set *curr* to the BST's root.

2. Repeat:

    2.1. If *curr* is null:

        2.1.1. Terminate with answer *none*.

    2.2. Otherwise, if *target* is equal to *curr*'s element:

        2.2.1. Terminate with answer *curr*.

    2.3. Otherwise, if *target* is less than *curr*'s element:

        2.3.1. Set *curr* to *curr*'s left child.

    2.4. Otherwise, if *target* is greater than *curr*'s element:

        2.4.1. Set *curr* to *curr*'s right child.

2. end

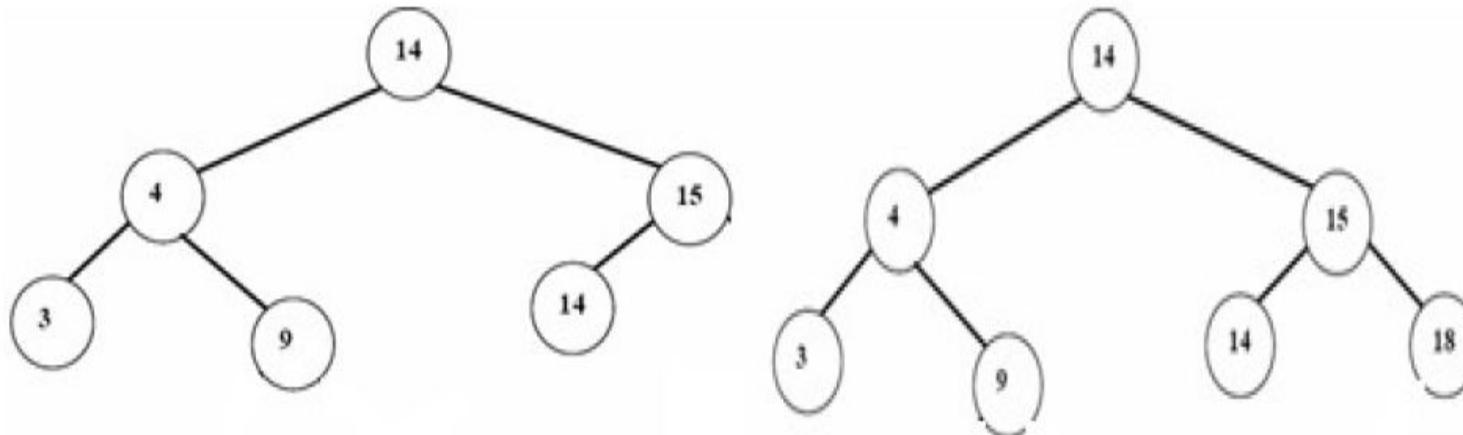
## C function for BST searching:

```
void BinSearch(struct bnode *root , int key)
{
    if(root == NULL)
    {
        printf("The number does not exist");
        exit(1);
    }
    else if (key == root->info)
    {
        printf("The searched item is found");
    }
    else if(key < root->info)
        return BinSearch(root->left, key);
    else
        return BinSearch(root->right, key);
}
```

## Insertion of a node in BST:

To insert a new item in a tree, we must first verify that its key is different from those of existing elements. To do this a search is carried out. If the search is unsuccessful, then item is inserted.

- **Idea:** To insert a new element into a BST, proceed as if searching for that element. If the element is not already present, the search will lead to a null link. Replace that null link by a link to a leaf node containing the new element.



---

insert(18)

## BST insertion algorithm:

To insert the element *elem* into a BST:

1. Set *parent* to null, and set *curr* to the BST's root.
2. Repeat:
  - 2.1. If *curr* is null:
    - 2.1.1. Replace the null link from which *curr* was taken (either the BST's root or *parent*'s left child or *parent*'s right child) by a link to a newly-created leaf node with element *elem*.
    - 2.1.2. Terminate.
  - 2.2. Otherwise, if *elem* is equal to *curr*'s element:
    - 2.2.1. Terminate.
  - 2.3. Otherwise, if *elem* is less than *curr*'s element:
    - 2.3.1. Set *parent* to *curr*, and set *curr* to *curr*'s left child.
  - 2.4. Otherwise, if *elem* is greater than *curr*'s element:
    - 2.4.1. Set *parent* to *curr*, and set *curr* to *curr*'s right child.
3. End



## C function for BST insertion:

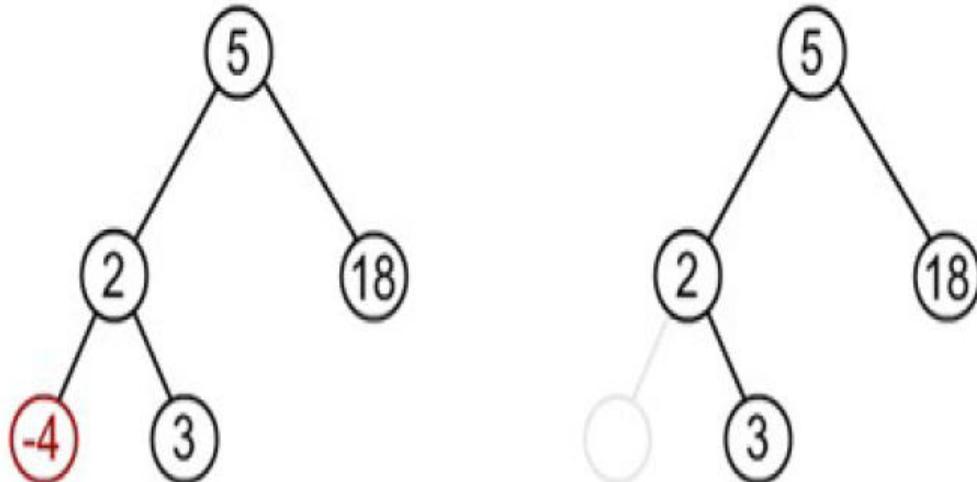
```
void insert(struct bnode *root, int item)
{
    if(root=NULL)
    {
        root=(struct bnode*)malloc (sizeof(struct bnode));
        root->left=root->right=NULL;
        root->info=item;
    }
    else
    {
        if(item<root->info)
            root->left=insert(root->left, item);
        else
            root->right=insert(root->right, item);
    }
}
```

## Deleting a node from the BST:

While deleting a node from BST, there may be three cases:

### *1. The node to be deleted may be a leaf node:*

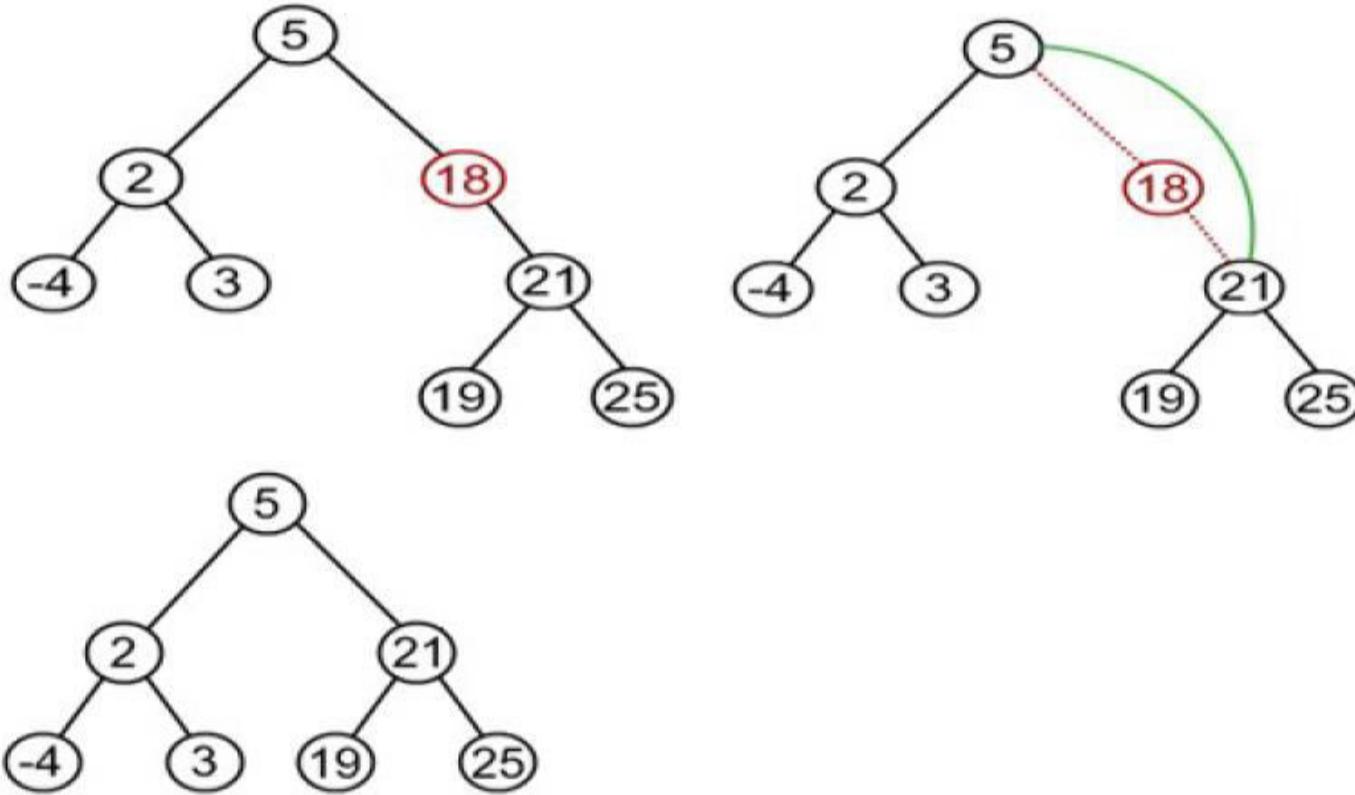
In this case simply delete a node and set null pointer to its parents those side at which this deleted node exist.



Suppose node to be deleted is -4

## 2. The node to be deleted has one child:

In this case the child of the node to be deleted is appended to its parent node.  
Suppose node to be deleted is 18

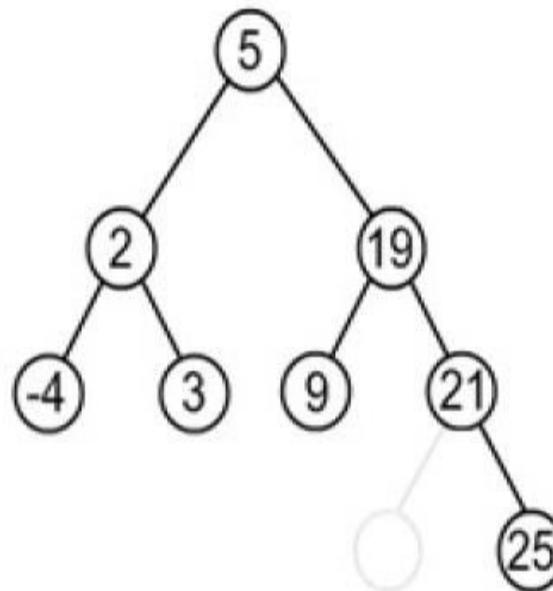
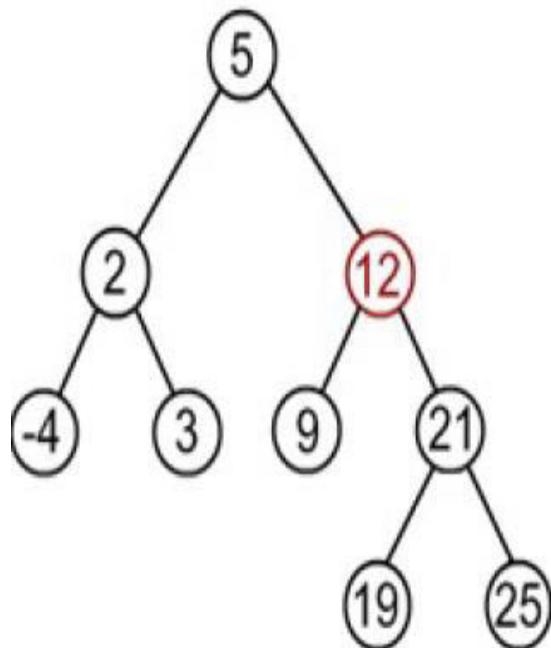


### 3. the node to be deleted has two children:

In this case node to be deleted is replaced by its in-order successor node.

OR

If the node to be deleted is either replaced by its right sub-trees leftmost node or its left sub-trees rightmost node.



Suppose node to be deleted is 12

Find minimum element in the right sub-tree of the node to be removed. In current example it is 19.

## General algorithm to delete a node from a BST:

1. start
2. if a node to be deleted is a leaf node at left side then simply delete and set null pointer to it's parent's left pointer.
3. If a node to be deleted is a leaf node at right side then simply delete and set null pointer to it's parent's right pointer
4. if a node to be deleted has one child then connect it's child pointer with it's parent pointer and delete it from the tree
5. if a node to be deleted has two children then replace the node being deleted either by
  - a. right most node of it's left sub-tree or
  - b. left most node of it's right sub-tree.
6. End

## The deleteBST function:

```
struct bnode *delete(struct bnode *root, int item)
{
    struct bnode *temp;
    if(root==NULL)
    {
        printf("Empty tree");
        return;
    }
```

```

else if(item<root->info)
    root->left=delete(root->left, item);
else if(item>root->info)
    root->right=delete(root->right, item);
else if(root->left!=NULL &&root->right!=NULL) //node has two child
{
    temp=find_min(root->right);
    root->info=temp->info;
    root->right=delete(root->right, root->info);

}
else
{
    temp=root;
    if(root->left==NULL)
        root=root->right;
    else if(root->right==NULL)
        root=root->left;
    free(temp);
}
return(temp);
}
*****find minimum element function*****
struct bnode *find_min(struct bnode *root)
{
    if(root==NULL)
        return0;
    else if(root->left==NULL)
        return root;
    else
        return(find_min(root->left));
}

```

## **Huffman algorithm:**

*Huffman algorithm is a method for building an extended binary tree with a minimum weighted path length from a set of given weights.*

- This is a method for the construction of minimum redundancy codes .
- Applicable to many forms of data transmission

*Our example: text files*

-1951, David Huffman found the “most efficient method of representing numbers, letters, and other symbols using binary code”. Now standard method used for data compression.

In Huffman Algorithm, a set of nodes assigned with values if fed to the algorithm. Initially 2 nodes are considered and their sum forms their parent node. When a new element is considered, it can be added to the tree. Its value and the previously calculated sum of the tree are used to form the new node which in turn becomes their parent.

Let us take any four characters and their frequencies, and *sort* this list by increasing frequency

Since to represent 4 characters the 2 bit is sufficient thus take initially two bits for each character this is called fixed length character.

<u>character</u>	<u>frequencies</u>
E:	10
T:	07
O:	05
A:	03

Now sort these characters according to their frequencies in non-decreasing order.

<u>character</u>	<u>frequencies</u>	<u>code</u>
A:	03	00
O:	05	01
T:	07	10
E:	10	11

Here before using Huffman algorithm the total number of bits required is

$$nb = 3*2 + 5*2 + 7*2 + 10*2 = 06 + 10 + 14 + 20 = 50 \text{ bits}$$

A: 3

O: 5

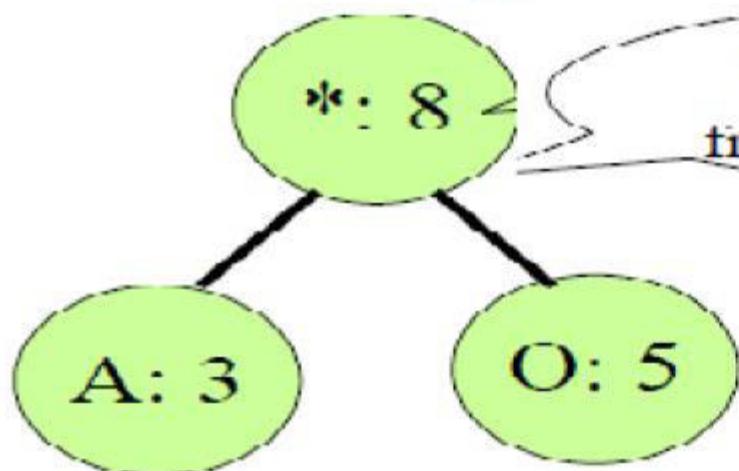
T: 7

E: 10

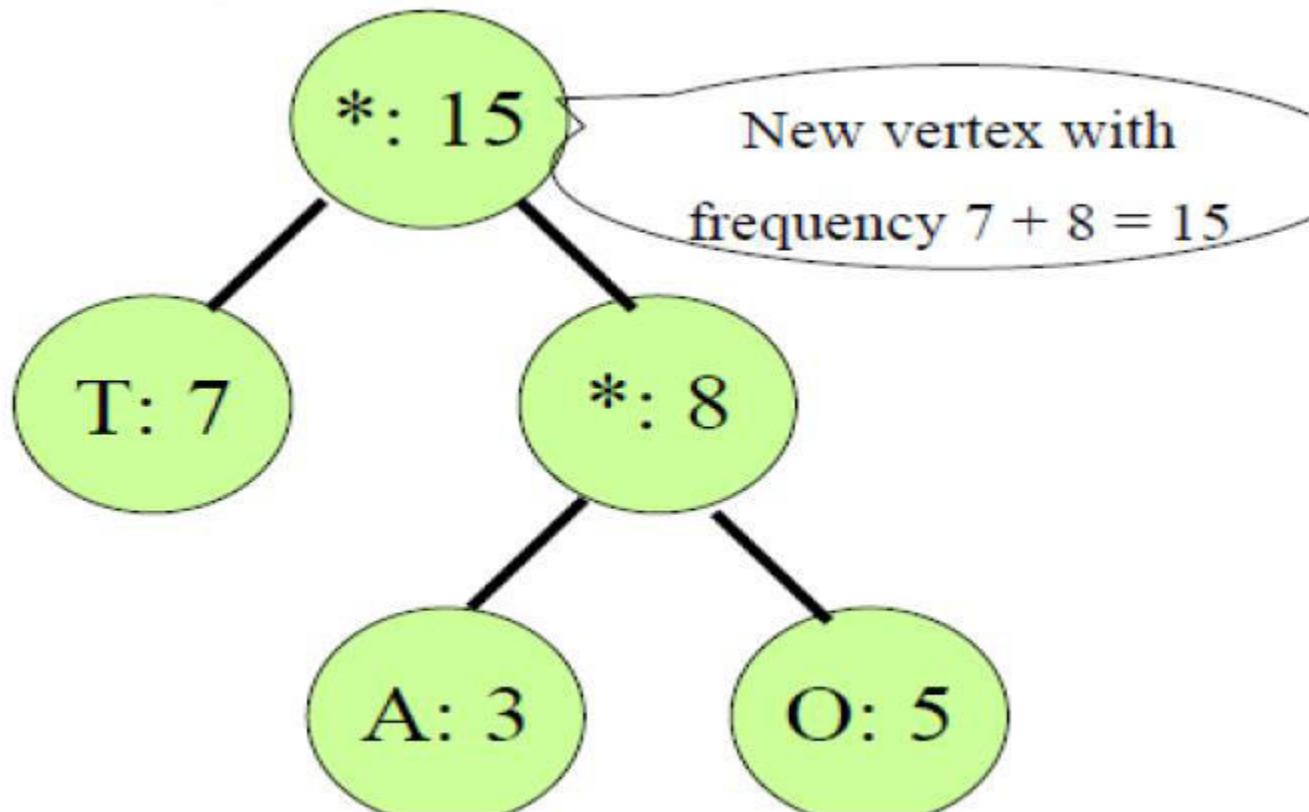
T: 7

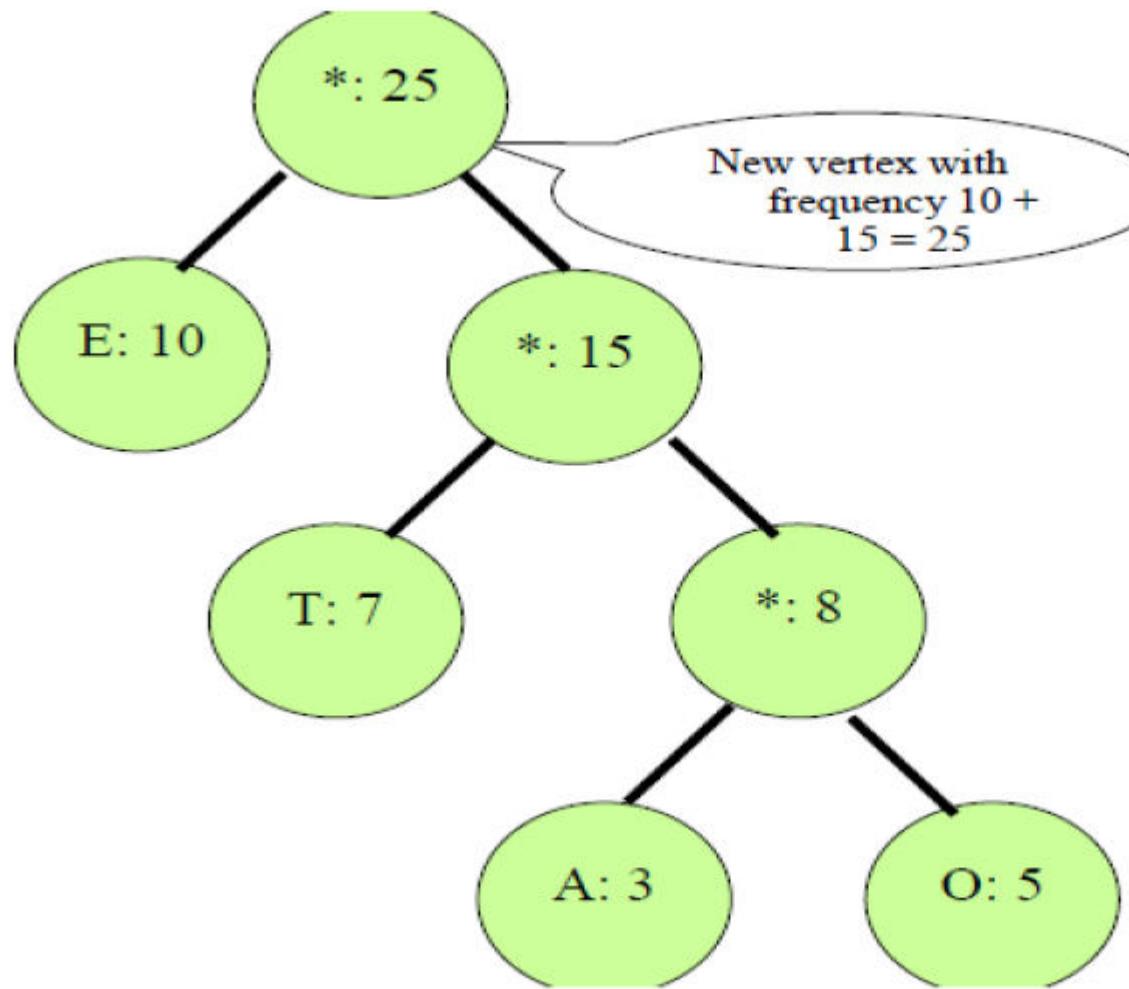
\*: 8

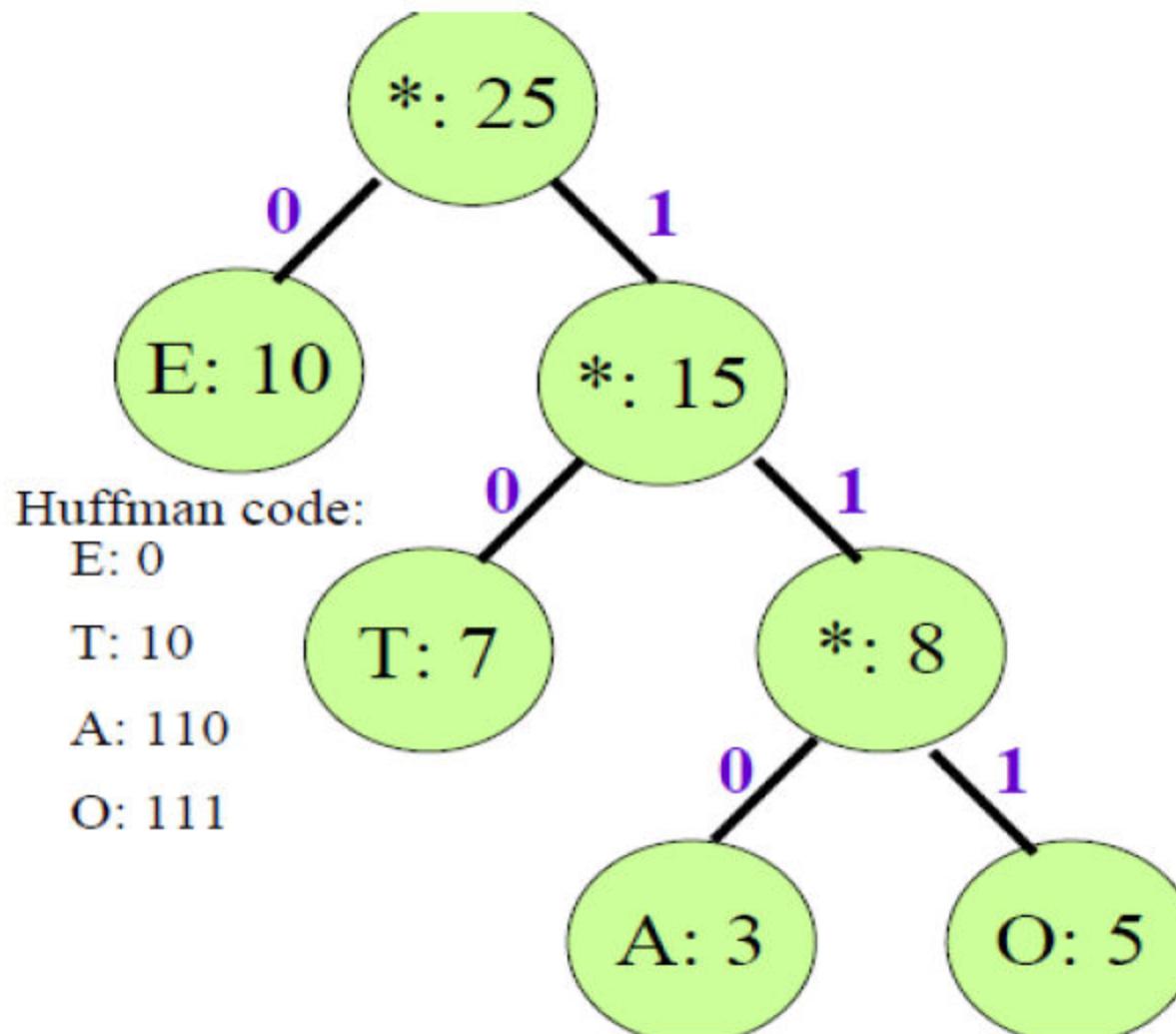
E: 10



New vertex with  
frequency  $3 + 5 = 8$







Left branch is 0  
Right branch is 1

Now from variable length code we get following code sequence.

<u>character</u>	<u>frequencies</u>	<u>code</u>
A:	03	110
O:	05	111
T:	07	10
E:	10	0

Thus after using Huffman algorithm the total number of bits required is

$$nb = 3*3 + 5*3 + 7*2 + 10*1 = 09 + 15 + 14 + 10 = 48 \text{ bits}$$

$$(50-48)/50*100\% = 4\%$$

Since in this small example we save about 4% space by using Huffman algorithm. If we take large example with a lot of characters and their frequencies we can save a lot of space.

# Thanks You

Any Queies

# DSA(lecture#8)

Prepared By Bal Krishna Subedi

## Sorting

Sorting is among the most basic problems in algorithm design. We are given a sequence of items, each associated with a given key value. The problem is to permute the items so that they are in increasing (or decreasing) order by key. Sorting is important because it is often the first step in more complex algorithms. Sorting algorithms are usually divided into two classes, internal sorting algorithms, which assume that data is stored in an array in main memory, and external sorting algorithm, which assume that data is stored on disk or some other device that is best accessed sequentially. We will only consider internal sorting. Sorting algorithms often have additional properties that are of interest, depending on the application. Here are two important properties.

In brief the sorting is a process of arranging the items in a list in some order that is either ascending or descending order.

## Bubble Sort:

The basic idea of this sort is to pass through the array sequentially several times. Each pass consists of comparing each element in the array with its successor (for example  $a[i]$  with  $a[i + 1]$ ) and interchanging the two elements if they are not in the proper order. For example, consider the following array:

| Initially:

25	57	48	37	12	92	86	33
0	1	2	3	4	5	6	7

| After Pass 1:

25	48	37	12	57	86	33	92
0	1	2	3	4	5	6	7

| After Pass 2:

25	37	12	48	57	33	86	92
0	1	2	3	4	5	6	7

| After Pass 3:

25	12	37	48	33	57	86	92
0	1	2	3	4	5	6	7

**After Pass 4:**

12	25	37	33	48	57	86	92
0	1	2	3	4	5	6	7

**After Pass 5:**

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

**After Pass 6:**

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

**After Pass 7:**

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

Here, we notice that after each pass, an element is placed in its proper order and is not considered in succeeding passes. Furthermore, we need  $n - 1$  passes to sort  $n$  elements.

**Algorithm**

BubbleSort(A, n)

```

{
    for(i = 0; i < n-1; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if(A[j] > A[j+1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
  
```

### **Time Complexity:**

Inner loop executes for  $(n-1)$  times when  $i=0$ ,  $(n-2)$  times when  $i=1$  and so on:

$$\begin{aligned}\text{Time complexity} &= (n-1) + (n-2) + (n-3) + \dots + 2 + 1 \\ &= O(n^2)\end{aligned}$$

There is no best-case linear time complexity for this algorithm.

### **Space Complexity:**

Since no extra space besides 3 variables is needed for sorting

$$\text{Space complexity} = O(n)$$

## Selection Sort:

**Idea:** Find the least (or greatest) value in the array, swap it into the leftmost(or rightmost) component (where it belongs), and then forget the leftmost component. Do this repeatedly. Let  $a[n]$  be a linear array of  $n$  elements. The selection sort works as follows:

**pass 1:** Find the location loc of the smallest element int the list of  $n$  elements  $a[0], a[1], a[2], a[3], \dots, a[n-1]$  and then interchange  $a[loc]$  and  $a[0]$ .

**Pass 2:** Find the location loc of the smallest element int the sub-list of  $n-1$  elements  $a[1], a[2], a[3], \dots, a[n-1]$  and then interchange  $a[loc]$  and  $a[1]$  such that  $a[0], a[1]$  are sorted.

..... and so on.

Then we will get the sorted list  $a[0] \leq a[1] \leq a[2] \leq a[3] \leq \dots \leq a[n-1]$ .

## **Algorithm:**

```
SelectionSort(A)
{
    for( i = 0;i < n ;i++)
    {
        least=A[i];
        p=i;
        for ( j = i + 1;j < n ;j++)
        {
            if (A[j] < A[i])
                least= A[j]; p=j;
        }
        swap(A[i],A[p]);
    }
}
```

## Time Complexity:

Inner loop executes for  $(n-1)$  times when  $i=0$ ,  $(n-2)$  times when  $i=1$  and so on:

$$\begin{aligned}\text{Time complexity} &= (n-1) + (n-2) + (n-3) + \dots + 2 + 1 \\ &= O(n^2)\end{aligned}$$

There is no best-case linear time complexity for this algorithm, but number of swap operations is reduced greatly.

## Space Complexity:

Since no extra space besides 5 variables is needed for sorting

$$\text{Space complexity} = O(n)$$

## Insertion Sort:

**Idea:** like sorting a hand of playing cards start with an empty left hand and the cards facing down on the table. Remove one card at a time from the table, and insert it into the correct position in the left hand. Compare it with each of the cards already in the hand, from right to left. The cards held in the left hand are sorted

Suppose an array  $a[n]$  with  $n$  elements. The insertion sort works as follows:

**pass 1:**  $a[0]$  by itself is trivially sorted.

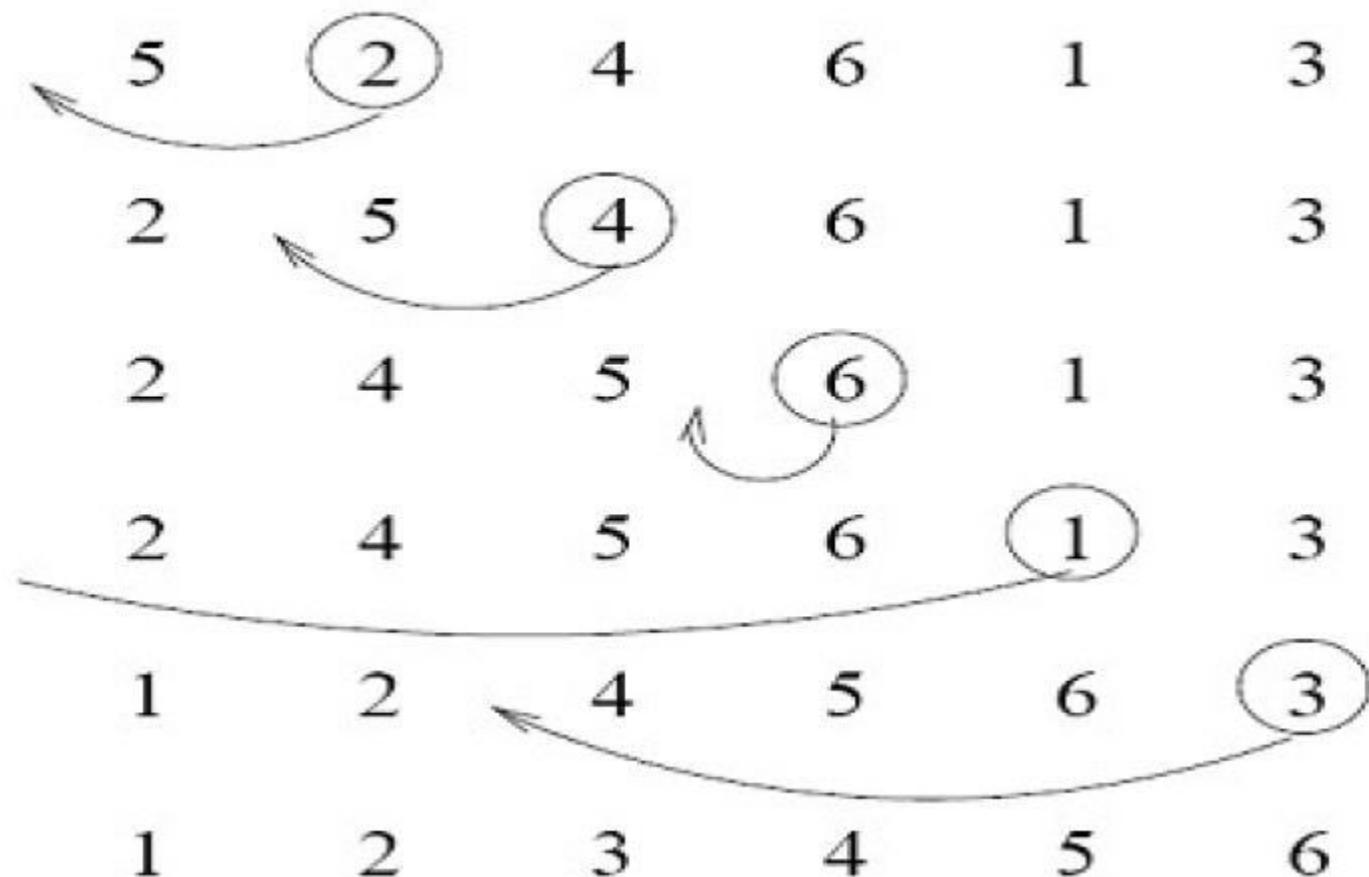
**Pass 2:**  $a[1]$  is inserted either before or after  $a[0]$  so that  $a[0], a[1]$  is sorted.

**Pass 3:**  $a[2]$  is inserted into its proper place in  $a[0], a[1]$  that is before  $a[0]$ , between  $a[0]$  and  $a[1]$ , or after  $a[1]$  so that  $a[0], a[1], a[2]$  is sorted.

.....

**pass N:**  $a[n-1]$  is inserted into its proper place in  $a[0], a[1], a[2], \dots, a[n-2]$  so that  $a[0], a[1], a[2], \dots, a[n-1]$  is sorted with  $n$  elements.

**Example:**



**Algorithm:**

```
InsertionSort(A)
{
    for (i=1; i<n; i++)
    {
        key = A[ i ]
        for(j=i; j>0 && A[j] >key, j--)
        {
            A[j + 1] = A[j]
        }
        A[j + 1] = key
    }
}
```

**Time Complexity:****Worst Case Analysis:**

Array elements are in reverse sorted order

Inner loop executes for 1 times when  $i=1$ , 2 times when  $i=2\dots$  and  $n-1$  times when  $i=n-1$ :

$$\begin{aligned}\text{Time complexity} &= 1 + 2 + 3 + \dots + (n-2) + (n-1) \\ &= O(n^2)\end{aligned}$$

**Best case Analysis:**

Array elements are already sorted

Inner loop executes for 1 times when  $i=1$ , 1 times when  $i=2\dots$  and 1 times when  $i=n-1$ :

$$\begin{aligned}\text{Time complexity} &= 1 + 2 + 3 + \dots + 1 + 1 \\ &= O(n)\end{aligned}$$

**Space Complexity:**

Since no extra space besides 5 variables is needed for sorting

$$\text{Space complexity} = O(n)$$

## Quick Sort:

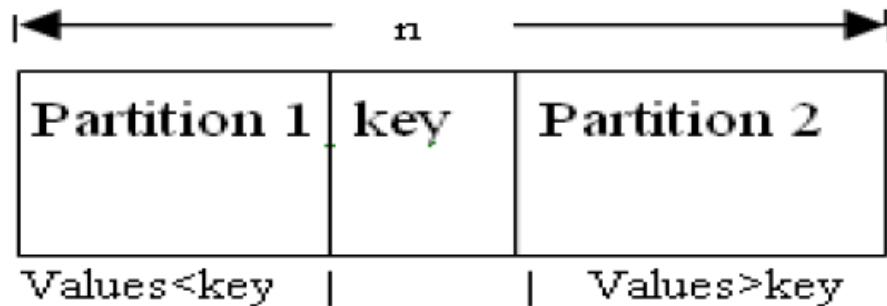
Quick sort developed by C.A.R Hoare is an unstable sorting. In practice this is the fastest sorting method. It possesses very good average case complexity among all the sorting algorithms. This algorithm is based on the divide and conquer paradigm. The main idea behind this sorting is partitioning of the elements.

### *Steps for Quick Sort:*

**Divide:** partition the array into two nonempty sub arrays.

**Conquer:** two sub arrays are sorted recursively.

**Combine:** two sub arrays are already sorted in place so no need to combine.



**Example:  $a[] = \{5, 3, 2, 6, 4, 1, 3, 7\}$**

5	3	2	6	4	1	3	7
x						y	

5	3	2	6	4	1	3	7
			x			y	

(swap x & y)

5	3	2	3	4	1	6	7
					y	x	

(swap y and pivot)

1	3	2	3	4	5	6	7
					p		

(1 3 2 3 4) 5 (5 7)

and continue this process for each sub-arrays and finally we get a sorted array.

**Algorithm:**

```
QuickSort(A,l,r)
{
    f(l<r)
    {
        p = Partition(A,l,r);
        QuickSort(A,l,p-1);
        QuickSort(A,p+1,r);
    }
}

Partition(A,l,r)
{
    x =l;
    y =r ;
    p = A[l];
    while(x<y)
    {
        while(A[x] <= p)
            x++;
        while(A[y] >=p)
            y--;
        if(x<y)
            swap(A[x],A[y]);
    }
    A[l] = A[y];
    A[y] = p;
    return y;      //return position of pivot
}
```

## Time Complexity:

### Best Case:

Divides the array into two partitions of equal size, therefore

$T(n) = 2T(n/2) + O(n)$  , Solving this recurrence we get,

$$T(n)=O(n\log n)$$

### Worst case:

when one partition contains the  $n-1$  elements and another partition contains only one element.  
Therefore its recurrence relation is:

$T(n) = T(n-1) + O(n)$ , Solving this recurrence we get

$$T(n)=O(n^2)$$

### Average case:

Good and bad splits are randomly distributed across throughout the tree

$T_1(n)= 2T'(n/2) + O(n)$  Balanced

$T'(n)= T(n-1) + O(n)$  Unbalanced

Solving:

$$\begin{aligned}B(n) &= 2(B(n/2 - 1) + \Theta(n/2)) + \Theta(n) \\&= 2B(n/2 - 1) + \Theta(n) \\&= O(n\log n) \\&\Rightarrow T(n)=O(n\log n)\end{aligned}$$

# Merge Sort

To sort an array  $A[1 \dots r]$ :

- **Divide**

- Divide the  $n$ -element sequence to be sorted into two sub-sequences of  $n/2$  elements

- **Conquer**

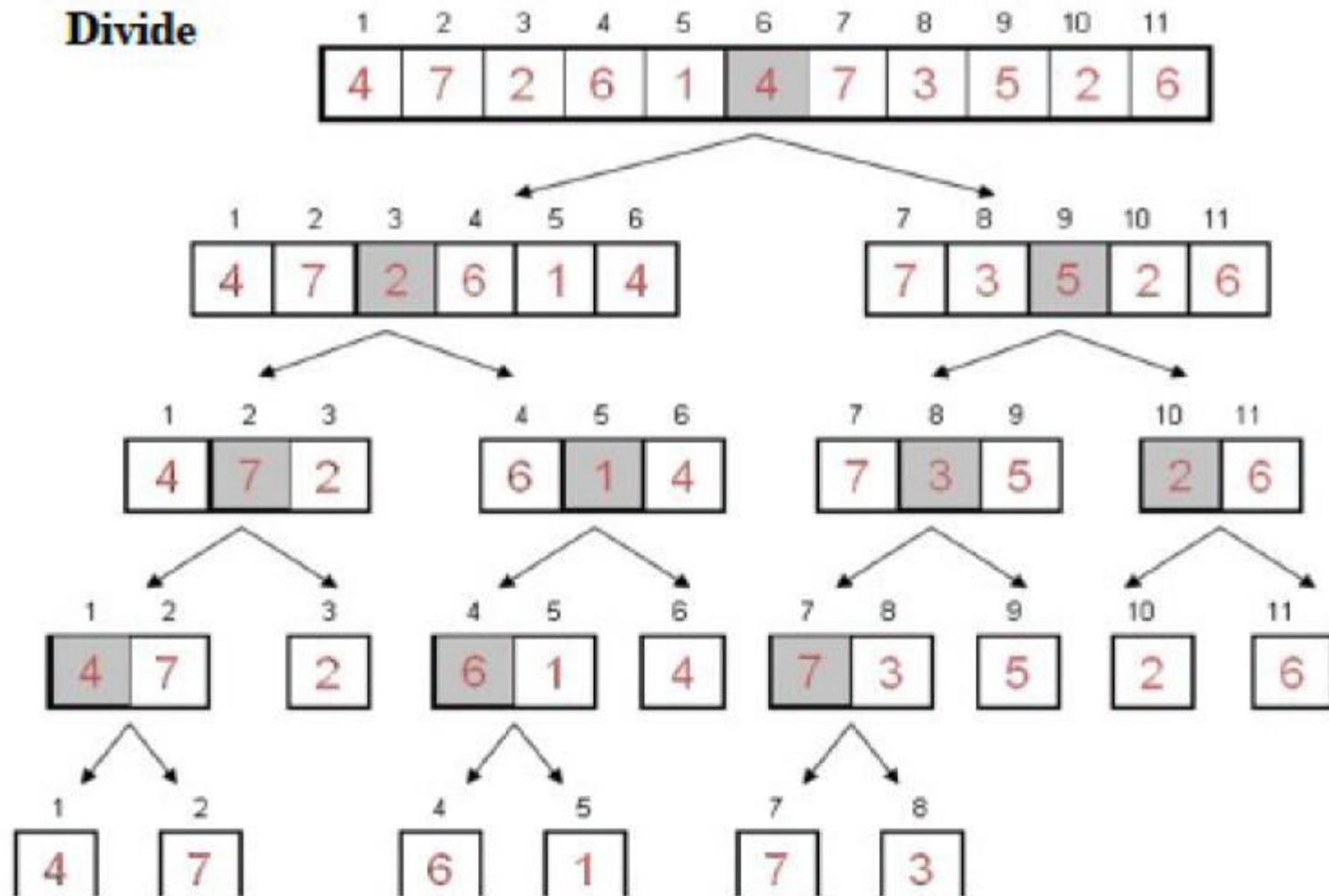
- Sort the sub-sequences recursively using merge sort. When the size of the sequences is 1 there is nothing more to do

- **Combine**

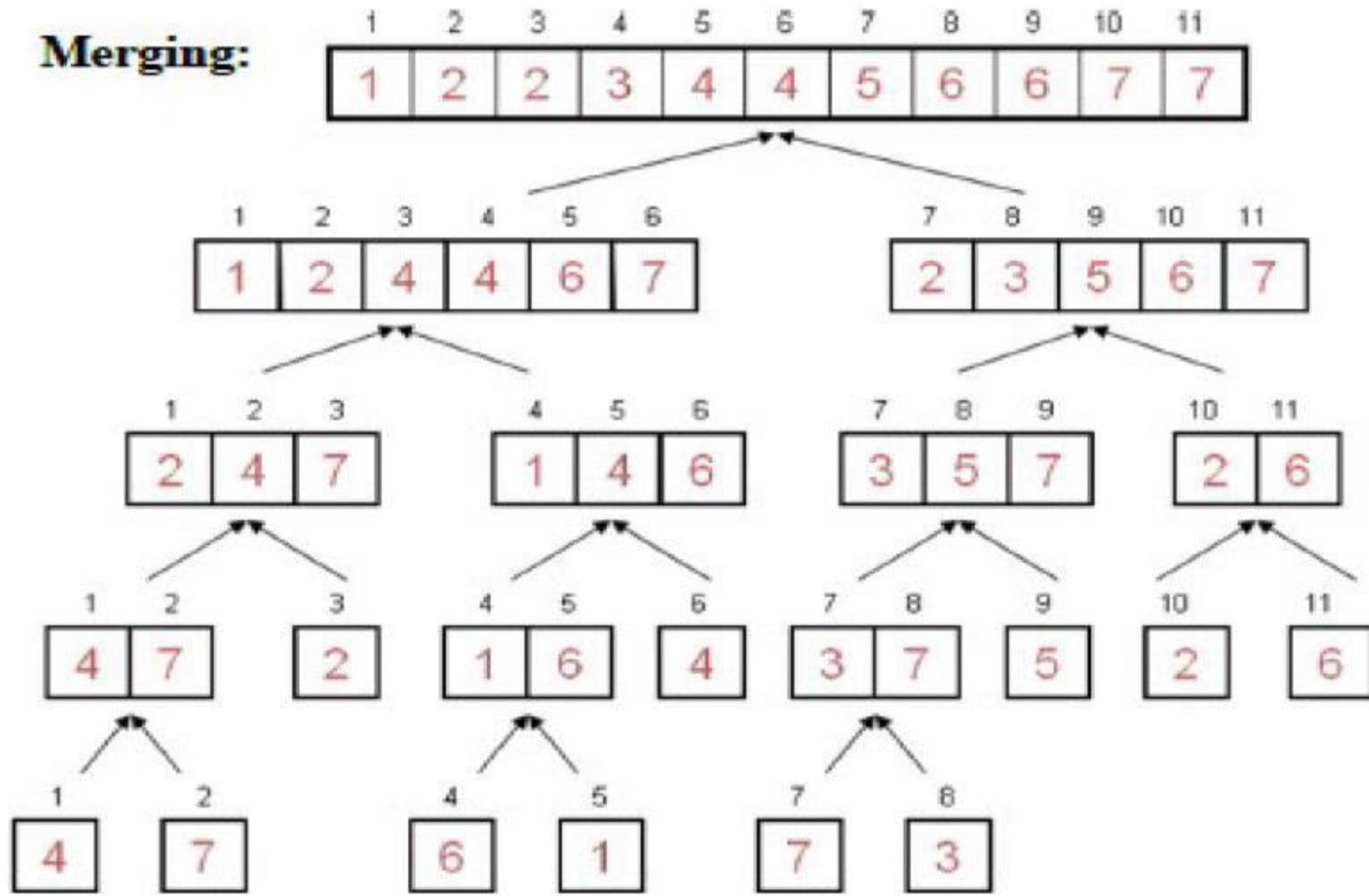
- Merge the two sorted sub-sequences

Example:  $a[] = \{4, 7, 2, 6, 1, 4, 7, 3, 5, 2, 6\}$

## Divide



## Merging:



**Algorithm:**

```
MergeSort(A, l, r)
{
    If ( l < r)                                //Check for base case
    {
        m = ⌊(l + r)/2⌋                      //Divide
        MergeSort(A, l, m)                     //Conquer
        MergeSort(A, m + 1, r)                 //Conquer
        Merge(A, l, m+1, r)                   //Combine
    }
}
```

```
Merge(A,B,l,m,r)
{
    x=l, y=m;
    k=l;
    while(x<m && y< r)
    {
        if(A[x] < A[y])
        {
            B[k]= A[x];
            k++; x++;
        }
        else
        {
            B[k] = A[y];
            k++; y++;
        }
    }
}
```

```
    }
    while(x<m)
    {
        A[k] = A[x];
        k++; x++;
    }
    while(y<r)
    {
        A[k] = A[y];
        k++; y++;
    }
    for(i=1;i<= r; i++)
    {
        A[i] = B[i]
    }
}
```

## Time Complexity:

Recurrence Relation for Merge sort:

$$T(n) = 1 \text{ if } n=1$$

$$T(n) = 2 T(n/2) + O(n) \text{ if } n>1$$

Solving this recurrence we get

$$T(n) = O(n\log n)$$

## Space Complexity:

It uses one extra array and some extra variables during sorting, therefore

$$\text{Space Complexity} = 2n + c = O(n)$$

## Sorting Comparison:

Sort	Worst Case	Average Case	Best Case	Comments
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	(*Unstable)
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	Requires Memory
Heap Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	*Large constants
Quick Sort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	*Small constants

# Thanks You

Any Queries?

# DSA Lecture#9

Prepared By Bal Krishna Subedi

# Searching

## Introduction:

Searching is a process of finding an element within the list of elements stored in any order or randomly. Searching is divided into two categories Linear and Binary search.

## Sequential Search:

In linear search, access each element of an array one by one sequentially and see whether it is desired element or not. A search will be unsuccessful if all the elements are accessed and the desired element is not found.

In brief, Simply search for the given element left to right and return the index of the element, if found. Otherwise return “Not Found”.

### **Algorithm:**

```
LinearSearch(A, n,key)
{
    for(i=0;i<n;i++)
    {
        if(A[i] == key)
            return i;
    }
    return -1;//-1 indicates unsuccessful search
}
```

### **Analysis:**

**Time complexity = O(n)**

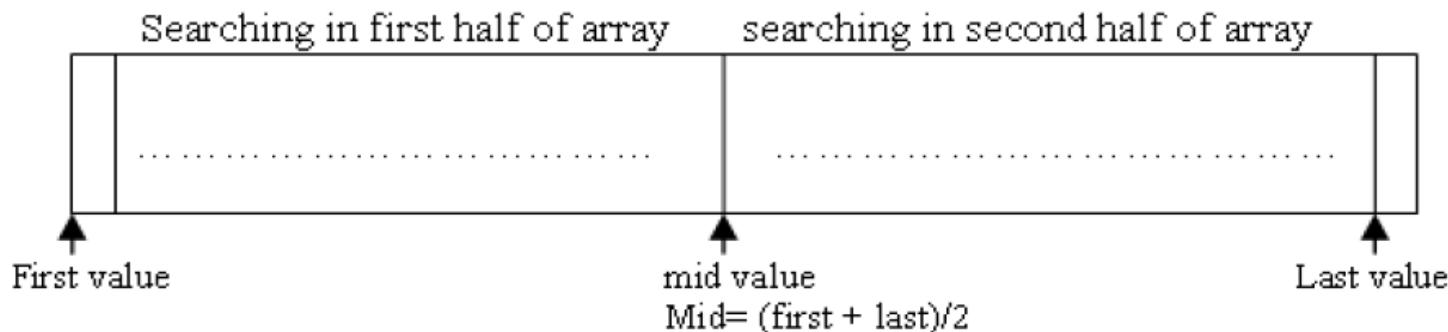
## Binary Search:

Binary search is an extremely efficient algorithm. This search technique searches the given item in minimum possible comparisons. To do this binary search, first we need to sort the array elements. The logic behind this technique is given below:

- ✓ First find the middle element of the array
- ✓ compare the middle element with an item.
- ✓ There are three cases:
  - ✗ If it is a desired element then search is successful
  - ✗ If it is less than desired item then search only the first half of the array.
  - ✗ If it is greater than the desired element, search in the second half of the array.

Repeat the same process until element is found or exhausts in the search area.

In this algorithm every time we are reducing the search area.



## **Running example:**

Take input array  $a[] = \{2, 5, 7, 9, 18, 45, 53, 59, 67, 72, 88, 95, 101, 104\}$

For key = 2

low	high	mid	
0	13	6	key < A[6]
0	5	2	key < A[2]
0	1	0	

Terminating condition, since  $A[mid] == 2$ , return 1(successful).

For key = 103

low	high	mid	
0	13	6	key > A[6]
7	13	10	key > A[10]
11	13	12	key > A[12]
13	13	-	

Terminating condition  $high == low$ , since  $A[0] != 103$ , return 0(unsuccessful).

For key = 67

low	high	mid	
0	13	6	key > A[6]
7	13	10	key < A[10]
7	9	8	

Terminating condition, since  $A[mid] = 67$ , return 9(successful).

**Algorithm:**

```
BinarySearch(A,l,r, key)
{
    if(l== r) //only one element
    {
        if(key == A[l])
            return l+1; //index starts from 0
        else
            return 0;
    }
    else
    {
        m = (l + r) / 2 ; //integer division
        if(key == A[m])
            return m+1;
        else if (key < A[m])
            return BinarySearch(l, m-1, key) ;
        else
            return BinarySearch(m+1, r, key) ;
    }
}
```

**Efficiency:**

From the above algorithm we can say that the running time of the algorithm is

$$T(n) = T(n/2) + O(1)$$

$$= O(\log n) \text{ (verify).}$$

In the best case output is obtained at one run i.e.  $O(1)$  time if the key is at middle.

In the worst case the output is at the end of the array so running time is  $O(\log n)$  time.

In the average case also running time is  $O(\log n)$ .

For unsuccessful search best, worst and average time complexity is  $O(\log n)$ .

## Hashing:

It is an efficient searching technique in which key is placed in direct accessible address for rapid search.

Hashing provides the direct access of records from the file no matter where the record is in the file. Due to which it reduces the unnecessary comparisons. This technique uses a hashing function say  $h$  which maps the key with the corresponding key address or location.

*A function that transforms a key into a table index is called a hash function.*

A common hash function is

$$h(x) = x \bmod \text{SIZE}$$

if key=27 and SIZE=10 then

hash address= $27 \% 10 = 7$

## Hash-table principles:

- Illustration:

key	value
$k_1$	$v_1$
$k_2$	$v_2$
$k_3$	$v_3$
$k_4$	$v_4$
$\vdots$	$\vdots$
$k_n$	$v_n$

collision

hash table  
(array of buckets)



hashing  
(translating keys  
to bucket indices)

## Hash collision:

If two or more than two records trying to insert in a single index of a hash table then such a situation is called hash collision.

Some popular methods for minimizing collision are:

- ✓ Linear probing
- ✓ Quadratic probing
- ✓ Rehashing
- ✓ Chaining
- ✓ Hashing using buckets etc

But here we need only first two methods for minimizing collision

## Linear probing:

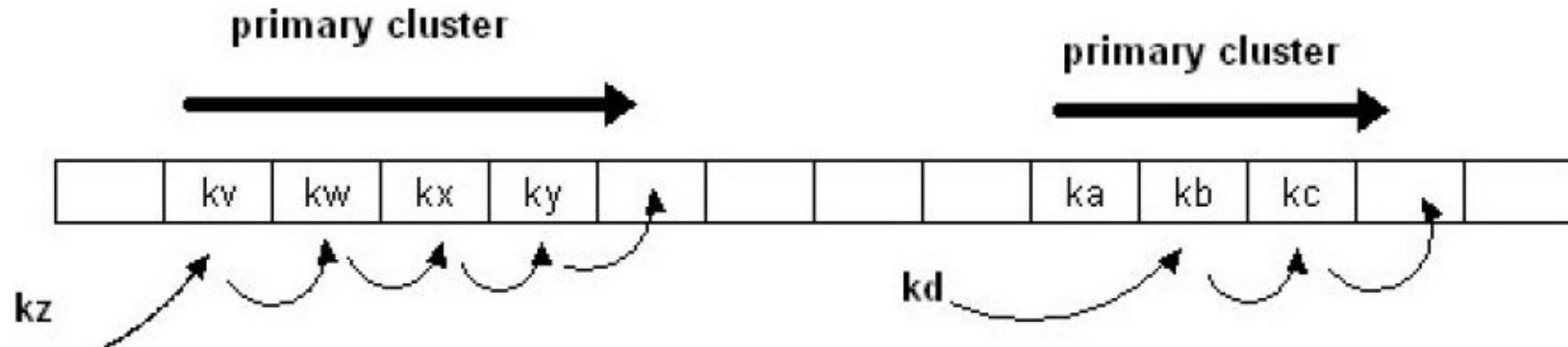
A hash-table in which a collision is resolved by putting the item in the next empty place within the occupied array space.

It starts with a location where the collision occurred and does a sequential search through a hash table for the desired empty location.

Hence this method searches in straight line, and it is therefore called linear probing.

*Disadvantage:*

Clustering problem



*Example:*

*Insert keys {89, 18, 49, 58, 69} with the hash function  
 $h(x)=x \bmod 10$  using linear probing.*

solution:

when  $x=89$ :

$$h(89)=89\%10=9$$

insert key 89 in hash-table in location 9

when  $x=18$ :

$$h(18)=18\%10=8$$

insert key 18 in hash-table in location 8

when  $x=49$ :

$$h(49)=49\%10=9 \text{ (Collision occur)}$$

so insert key 49 in hash-table in next possible vacant location of 9 is 0

when  $x=58$ :

$$h(58)=58\%10=8 \text{ (Collision occur)}$$

insert key 58 in hash-table in next possible vacant location of 8 is 1  
(since 9, 0 already contains values).

when  $x=69$ :

$$h(69)=69\%10=9 \text{ (Collision occur)}$$

insert key 69 in hash-table in next possible vacant location of 9 is 2  
(since 0, 1 already contains values).

□	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	89

Fig Hash-table for above keys using linear probing

## Quadratic Probing:

Quadratic probing is a collision resolution method that eliminates the primary clustering problem take place in a linear probing.

When collision occur then the quadratic probing works as follows:

$$(\text{Hash value} + 1^2) \% \text{table size}$$

if there is again collision occur then there exist rehashing.

$$(\text{hash value} + 2^2) \% \text{table size}$$

if there is again collision occur then there exist rehashing.

$$(\text{hash value} + 3^2) \% \text{table size}$$

in general in ith collision

$$h_i(x) = (\text{hash value} + i^2) \% \text{size}$$

**Example:**

**Insert keys {89, 18, 49, 58, 69} with the hash-table size 10 using quadratic probing.**

solution:

when x=89:

$$h(89)=89\%10=9$$

insert key 89 in hash-table in location 9

when x=18:

$$h(18)=18\%10=8$$

insert key 18 in hash-table in location 8

when x=49:

$$h(49)=49\%10=9 \text{ (Collision occur)}$$

so use following hash function,

$$h1(49)=(49 + 1)\%10=0$$

hence insert key 49 in hash-table in location 0

when x=58:

$$h(58)=58\%10=8 \text{ (Collision occur)}$$

so use following hash function,

$$h1(58)=(58 + 1)\%10=9$$

again collision occur use again the following hash function ,

$$h2(58)=(58+ 22)\%10=2$$

insert key 58 in hash-table in location 2

when  $x=69$ :

$h(89)=69\%10=9$  (Collision occur)

so use following hash function,

$h1(69)=(69 + 1)\%10=0$

again collision occur use again the following hash function ,

$h2(69)=(69+ 22)\%10=3$

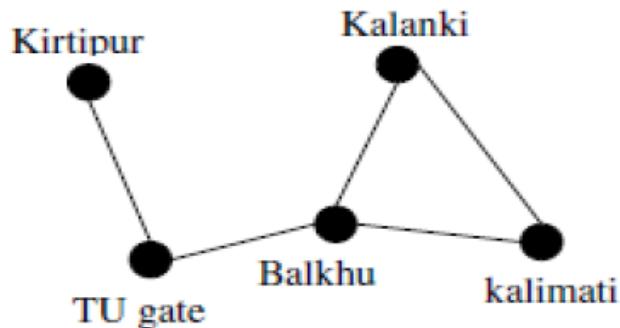
insert key 69 in hash-table in location 3

0	49
1	
2	58
3	69
4	
5	
6	
7	
8	18
9	89

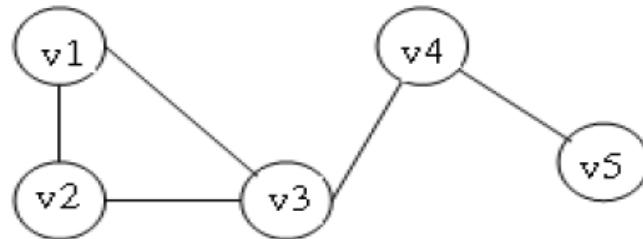
fig:Hash table for above keys using quadratic probing

## Graph:

A Graph is a pair  $G = (V, E)$  where  $V$  denotes a set of vertices and  $E$  denotes the set of edges connecting two vertices. Many natural problems can be explained using graph for example modeling road network, electronic circuits, etc. The example below shows the road network.



Let us take a graph:



$$V(G) = \{v_1, v_2, v_3, v_4, v_5\}$$

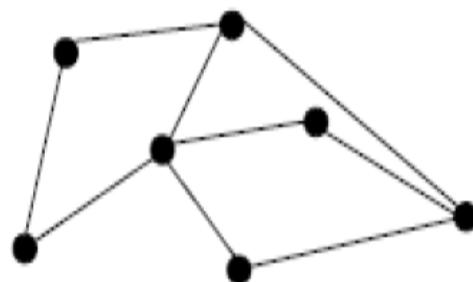
$$E(G) = \{(v_1, v_2), (v_2, v_3), (v_1, v_3), (v_3, v_4), (v_4, v_5)\}$$

## Types of Graph:

### Simple Graph:

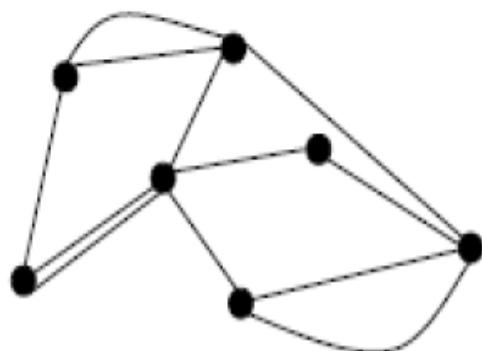
We define a simple graph as 2 – tuple consists of a non empty set of vertices V and a set of unordered pairs of distinct elements of vertices called edges. We can represent graph as  $G = (V, E)$ . This kind of graph has no loops and can be used for modeling networks that do not have connection to themselves but have both ways connection when two vertices are connected

but no two vertices have more than one connection. The figure below is an example of simple graph.



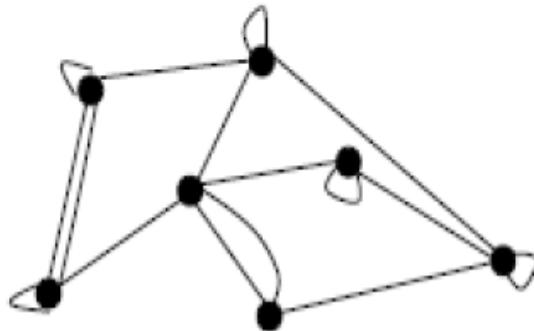
## Multigraph:

A multigraph  $G = (V, E)$  consists of a set of vertices  $V$ , a set of edges  $E$ , and a function  $f$  from  $E$  to  $\{\{u, v\} | u, v \in V, u \neq v\}$ . The edges  $e_1$  and  $e_2$  are called multiple or parallel edges if  $f(e_1) = f(e_2)$ . In this representation of graph also loops are not allowed. Since simple graph has single edges every simple graph is a multigraph. The figure below is an example of a multigraph.



### Pseudograph:

A pseudograph  $G = (V, E)$  consists of a set of vertices  $V$ , a set of edges  $E$ , and a function  $f$  from  $E$  to  $\{\{u, v\} | u, v \in V\}$ . An edge is a loop if  $f(e) = \{u, u\} = \{u\}$  for some  $u \in V$ . The figure below is an example of a multigraph



## Directed Graph:

A directed graph  $(V, E)$  consists of a set  $V$  of vertices, a set  $E$  of edges that are ordered pairs of elements of  $V$ . The below figure is a directed graph. In this graph loop is allowed but

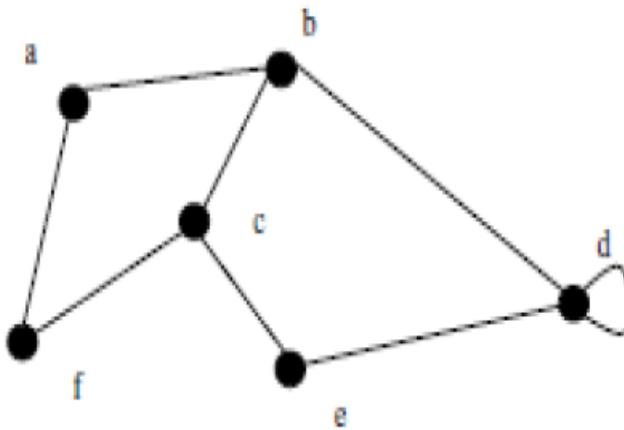
## Terminologies:

Two vertices  $u, v$  are adjacent vertices of a graph if  $\{u, v\}$  is an edge.

The edge  $e$  is called incident with the vertices  $u$  and  $v$  if  $e = \{u, v\}$ . This edge is also said to connect  $u$  and  $v$ , where  $u$  and  $v$  are end points of the edge.

**Degree of a vertex** in an undirected graph is the number of edges incident with it, except a loop at a vertex. Loop in a vertex counts twice to the degree. Degree of a vertex  $v$  is denoted by  $\deg(v)$ . A vertex of degree zero is called isolated vertex and a vertex with degree one is called pendant vertex.

**Example:** Find the degrees of the vertices in the following graph.



**Solution:**

$$\deg(a) = \deg(f) = \deg(e) = 2 ; \deg(b) = \deg(c) = 3; \deg(d) = 4$$

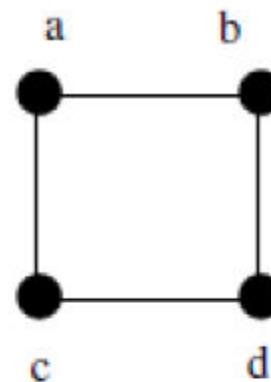
## Representation of Graph

Generally graph can be represented in two ways namely adjacency lists(Linked list representation) and adjacency matrix(matrix).

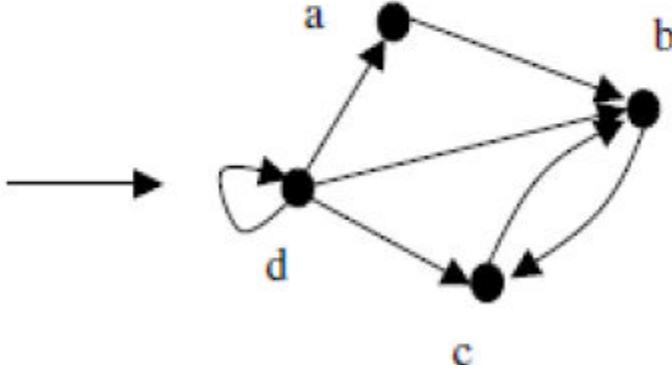
### Adjacency List:

This type of representation is suitable for the undirected graphs without multiple edges, and directed graphs. This representation looks as in the tables below.

Edge List for Simple Graph	
Vertex	Adjacent Vertices
a	b, c
b	a, d
c	a, d
d	b, c



Edge List for Directed Graph	
Initial Vertex	End Vertices
a	b
b	c
c	b
d	a, b, c, d



If we try to apply the algorithms of graph using the representation of graphs by lists of edges, or adjacency lists it can be tedious and time taking if there are high number of edges. For the sake of the computation, the graphs with many edges can be represented in other ways. In this class we discuss two ways of representing graphs in form of matrix.

### Adjacency Matrix:

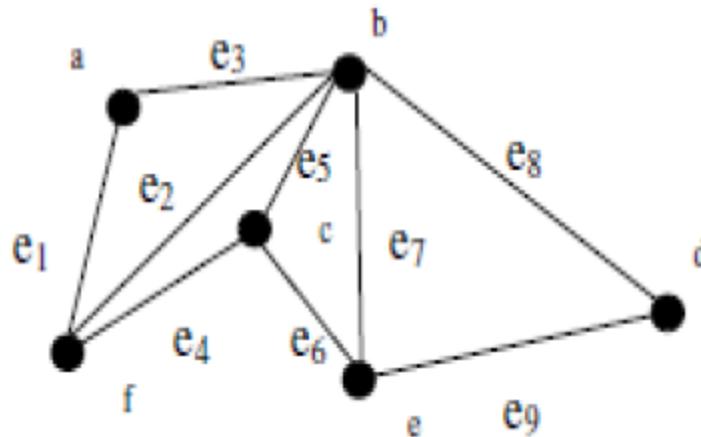
**Given a simple graph  $G = (V, E)$  with  $|V| = n$ . assume that the vertices of the graph are** listed in some arbitrary order like  $v_1, v_2, \dots, v_n$ . The adjacency matrix  $A$  of  $G$ , with respect to the order of the vertices is  $n$ -by- $n$  zero-one matrix ( $A = [a_{ij}]$ ) with the condition,

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$

Since there are  $n$  vertices and we may order vertices in any order there are  $n!$  possible order of the vertices. The adjacency matrix depends on the order of the vertices, hence there are  $n!$  possible adjacency matrices for a graph with  $n$  vertices.

In case of the directed graph we can extend the same concept as in undirected graph as dictated by the relation

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$

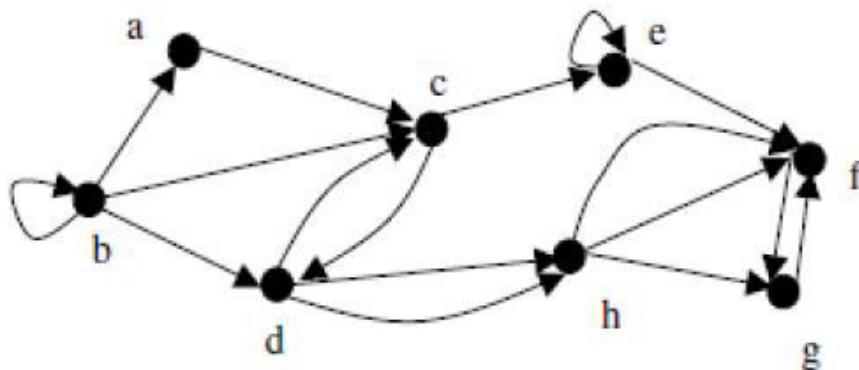


**Solution:** Let the order of the vertices be a, b, c, d, e, f

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Adjacency Matrix

Let us take a directed graph



**Solution:**

Let the order of the vertices be a, b, c, d, e, f, g

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & p \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 1 & 0 \end{bmatrix}$$

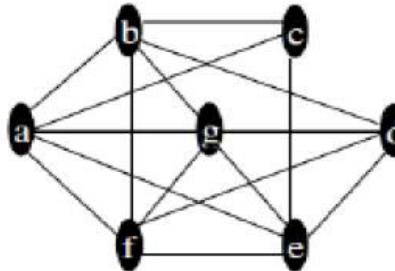
## Graph Traversals

### **Breadth-first search:**

This is one of the simplest methods of graph searching. Choose some vertex arbitrarily as a root. Add all the vertices and edges that are incident in the root. The new vertices added will become the vertices at the level 1 of the BFS tree. Form the set of the added vertices of level 1, find other vertices, such that they are connected by edges at level 1 vertices. Follow the above step until all the vertices are added.

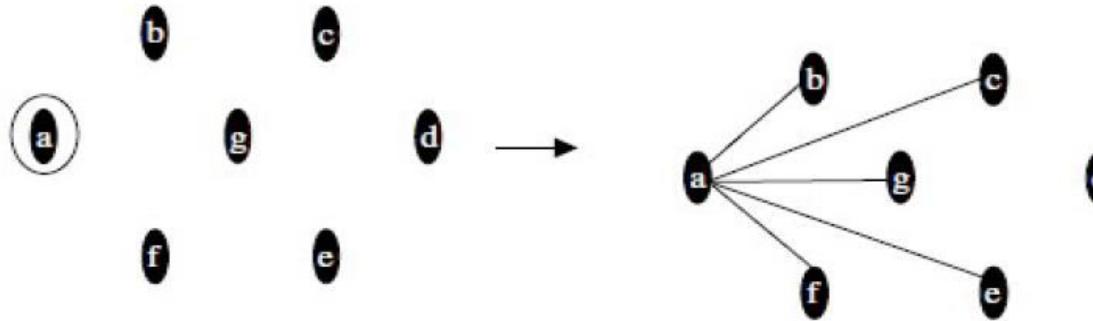
**Example:**

Use breadth first search to find a BFS tree of the following graph

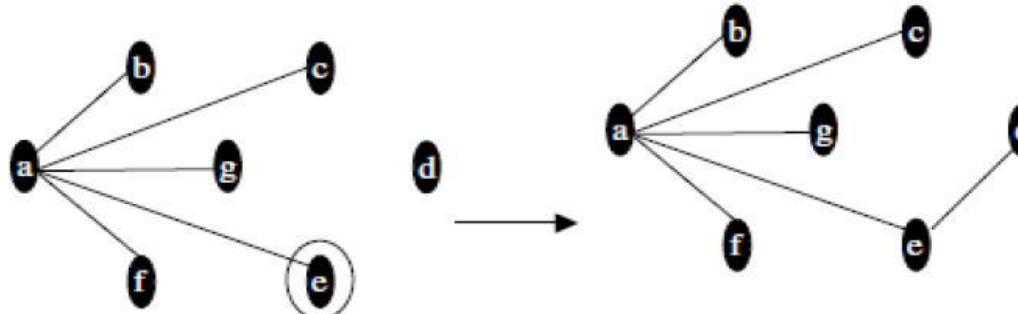


**Solution:**

Choose a as initial vertex then we have



Order the vertices of level 1 i.e. {b, c, g, e, f}. Say order be {e, f, g, b, c}.



**Algorithm:**

BFS(G,s) //s is start vertex

{

T = {s};

    L =  $\Phi$ ; //an empty queue

Enqueue(L,s);

    while (L  $\neq \Phi$ )

{

v = dequeue(L);

for each neighbor w to v

            if ( w  $\notin$  L and w  $\notin$  T )

{

enqueue( L,w);

T = T U {w}; //put edge {v,w} also

}

}

}

## **Analysis**

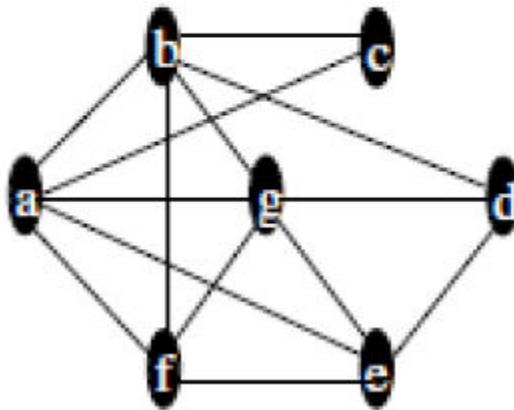
From the algorithm above all the vertices are put once in the queue and they are accessed. For each accessed vertex from the queue their adjacent vertices are looked for and this can be done in  $O(n)$  time(for the worst case the graph is complete). This computation for all the possible vertices that may be in the queue i.e.  $n$ , produce complexity of an algorithm as  $O(n^2)$ .

## **Depth First Search:**

This is another technique that can be used to search the graph. Choose a vertex as a root and form a path by starting at a root vertex by successively adding vertices and edges. This process is continued until no possible path can be formed. If the path contains all the vertices then the tree consisting this path is DFS tree. Otherwise, we must add other edges and vertices. For this move back from the last vertex that is met in the previous path and find whether it is possible to find new path starting from the vertex just met. If there is such a path continue the process above. If this cannot be done, move back to another vertex and repeat the process. The whole process is continued until all the vertices are met. This method of search is also called **backtracking**.

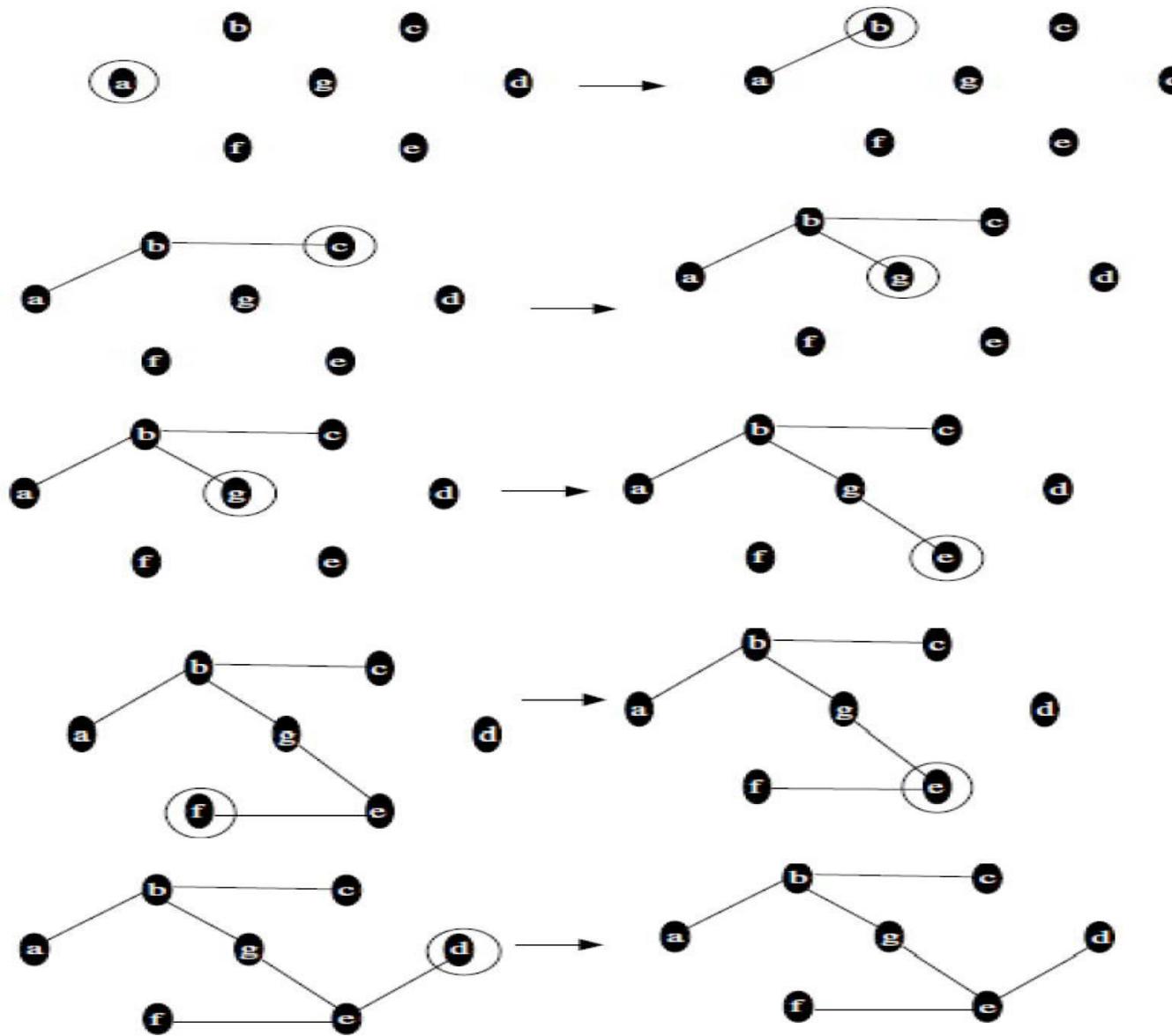
**Example:**

Use depth first search to find a spanning tree of the following graph.



**Solution:**

Choose a as initial vertex then we have



### **Algorithm:**

```
DFS(G,s)
{
    T = {s};
    Traverse(s);
}
Traverse(v)
{
    for each w adjacent to v and not yet in T
    {
        T = T U {w}; //put edge {v,w} also
        Traverse (w);
    }
}
```

### **Analysis:**

The complexity of the algorithm is greatly affected by **Traverse** function we can write its running time in terms of the relation  $T(n) = T(n-1) + O(n)$ , here  $O(n)$  is for each vertex at most all the vertices are checked (for loop). At each recursive call a vertex is decreased. Solving this we can find that the complexity of an algorithm is  $O(n^2)$ .

## Minimum Spanning Trees

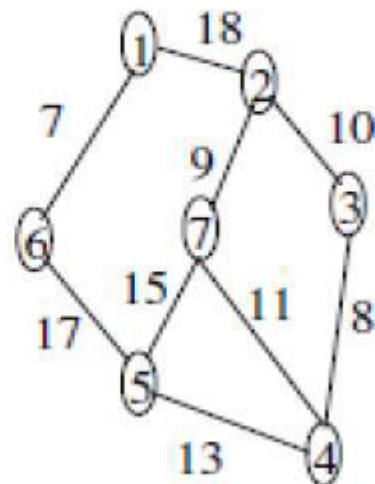
A minimum spanning tree in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges. In this part we study one algorithm that is used to construct the minimum spanning tree from the given connected weighted graph.

### Kruskal's Algorithm:

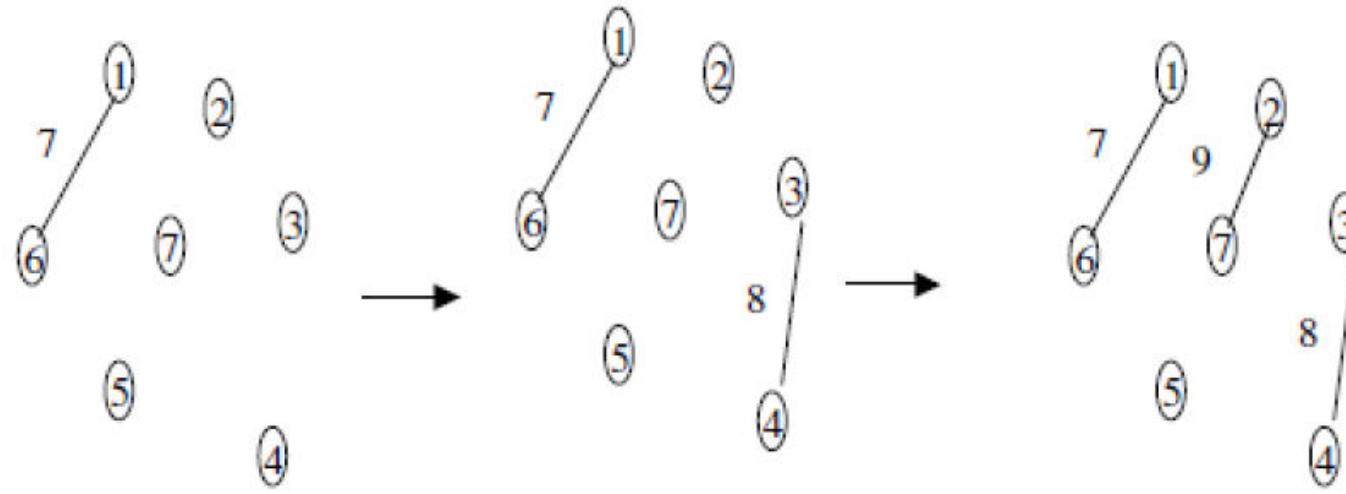
The problem of finding MST can be solved by using Kruskal's algorithm. The idea behind this algorithm is that you put the set of edges form the given graph  $G = (V,E)$  in nondecreasing order of their weights. The selection of each edge in sequence then guarantees that the total cost that would from will be the minimum. Note that we have  $G$  as a graph,  $V$  as a set of  $n$  vertices and  $E$  as set of edges of graph  $G$ .

**Example:**

Find the MST and its weight of the graph.

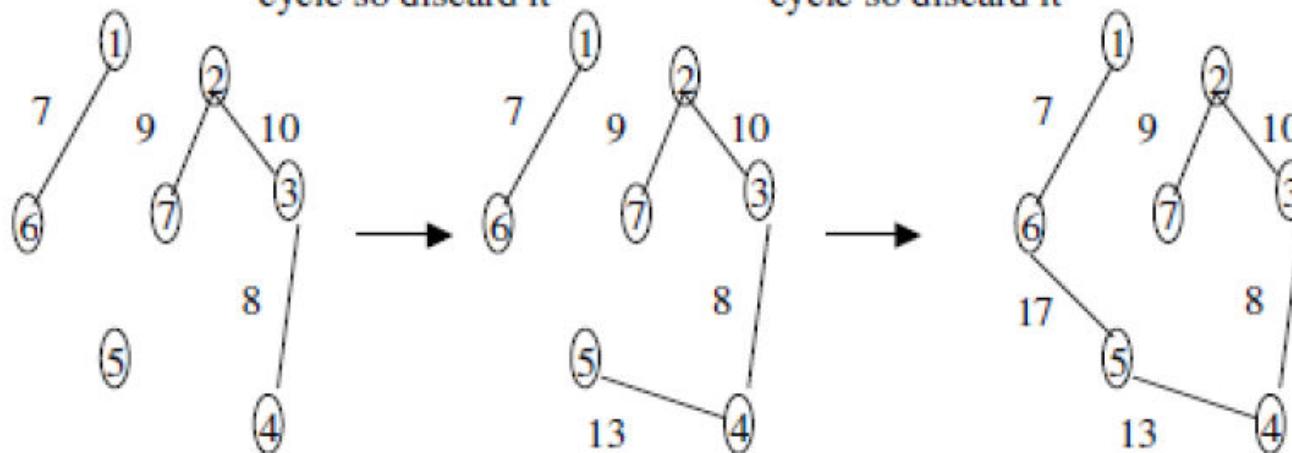


### Solution:



Edge with weight 11 forms cycle so discard it

Edge with weight 15 forms cycle so discard it



The total weight of MST is 64.

**Algorithm:**

*KruskalMST(G)*

{

*T = {V} // forest of n nodes*

*S = set of edges sorted in nondecreasing order of weight*

*while(|T| < n-1 and E != Ø)*

{

*Select (u,v) from S in order*

*Remove (u,v) from E*

*if((u,v) does not create a cycle in T))*

*T = T È {(u,v)}*

}

}

**Analysis:**

In the above algorithm the  $n$  tree forest at the beginning takes  $(V)$  time, the creation of set  $S$  takes  $O(E \log E)$  time and while loop execute  $O(n)$  times and the steps inside the loop take almost linear time (see disjoint set operations; find and union). So the total time taken is  $O(E \log E)$

# Thanks You

Any Queries?

# [Graph Theory]

Discrete Structure

Tribhuvan University

Kathmandu, Nepal

## Graphs

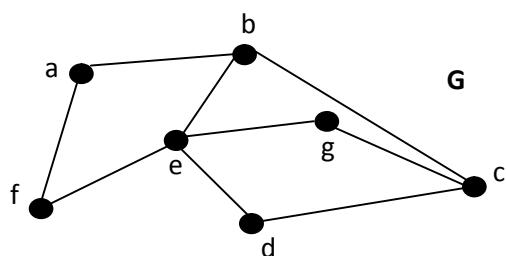
Graph is a discrete structure consisting of **vertices** and **edges** that connect these vertices. Problems in almost every conceivable discipline can be solved using graph model. Graph was introduced in eighteenth century by the great Swiss mathematician Leonhard Euler. Graphs are used to solve many problems in many fields. For example,

- Graphs can be used to determine whether a circuit can be implemented on a planar circuit board.
- Graphs can be used to distinguish two chemical compounds with the same molecular formula but different structures.
- Graphs can be used to study the structure of the WWW (World Wide Web).
- Graph model of computer network can be used to determine whether two computers are connected by a communication link.
- Graphs with weights assigned to their edges can be used to solve problems such as finding the shortest path between two cities in a transportation network.
- Graphs can also be used to schedule exams and assign channels to television stations.

## Types of graphs

### Simple Graph

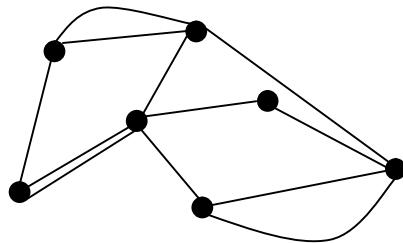
We define a simple graph as 2 – tuple consists of a non empty set of vertices V and a set of unordered pairs of distinct elements of vertices called edges. We can represent graph as  $G = (V, E)$ . A simple graph  $G = (V, E)$  consists of V, a nonempty set of vertices, and E, a set of unordered pairs of distinct elements of V called edges. This kind of graph has undirected edges, no loops and no multiple edges. The figure below is an example of simple graph.



In the above graph  $G = (V, E)$ , the set of vertices,  $V(G)$  or  $V = \{a, b, c, d, e, f, g\}$  and the set of edges  $E(G)$  or  $E = \{\{a, b\}, \{a, f\}, \{b, c\}, \{b, e\}, \{c, d\}, \{c, g\}, \{d, e\}, \{e, g\}, \{e, f\}\}$ .

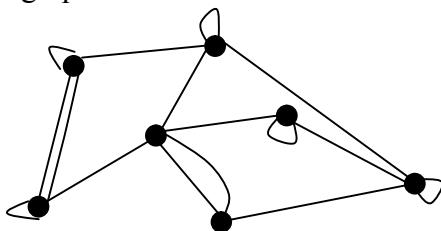
## Multigraph

A multigraph  $G = (V, E)$  consists of a set of vertices  $V$ , a set of edges  $E$ , and a function  $f$  from  $E$  to  $\{\{u, v\} \mid u, v \in V, u \neq v\}$ . The edges  $e1$  and  $e2$  are called multiple or parallel edges if  $f(e1) = f(e2)$ . This type of graph can have multiple edges between the same two vertices. This kind of graph has undirected edges, and no loops. Since simple graph has single edge, every simple graph is a multigraph. The figure below is an example of a multigraph.



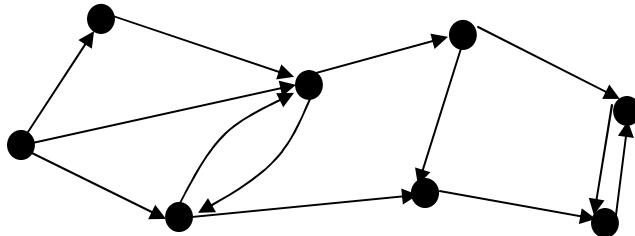
## Pseudograph

A pseudograph  $G = (V, E)$  consists of a set of vertices  $V$ , a set of edges  $E$ , and a function  $f$  from  $E$  to  $\{\{u, v\} \mid u, v \in V\}$ . An edge is a loop if  $f(e) = \{u, u\} = \{u\}$  for some  $u \in V$ . The figure below is an example of a pseudograph.



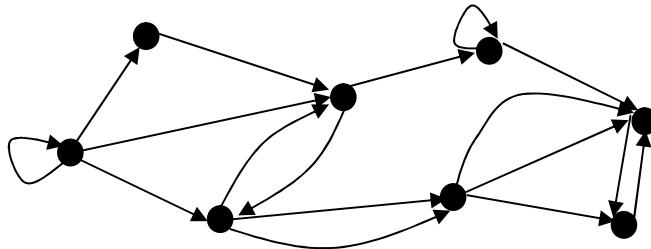
## Directed Graph

A directed graph  $(V, E)$  consists of a set  $V$  of vertices, a set  $E$  of edges that are ordered pairs of elements of  $V$ . The below figure is a directed graph. In this graph loop is allowed but no two vertices can have multiple edges in same direction.



## Directed Multigraph

A directed multigraph  $G = (V, E)$  consists of a set of vertices  $V$ , a set of edges  $E$ , and a function  $f$  from  $E$  to  $\{(u, v) | u, v \in V\}$ . The edges  $e1$  and  $e2$  are called multiple edges if  $f(e1) = f(e2)$ . The figure below is an example of a directed multigraph.

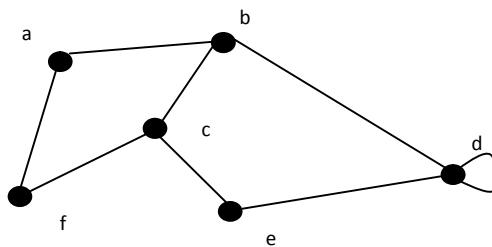


## Graph Terminologies

Two vertices  $u, v$  in an undirected graph  $G$  are called **adjacent** or **neighbors** if  $\{u, v\}$  is an edge. The edge  $e$  is called **incident with** the vertices  $u$  and  $v$  if  $e = \{u, v\}$ . This edge is also said to connect  $u$  and  $v$ , where  $u$  and  $v$  are end points of the edge.

The **degree of a vertex** in an undirected graph is the number of edges incident with it, except a loop at a vertex. Loop in a vertex counts twice to the degree. Degree of a vertex  $v$  is denoted by  $\deg(v)$ . A vertex of degree zero is called **isolated** vertex and a vertex with degree one is called **pendant** vertex.

**Example:** Find the degrees of the vertices in the following graph.



**Solution:**

$$\deg(a) = \deg(f) = \deg(e) = 2 ; \deg(b) = \deg(c) = 3; \deg(d) = 4$$

## Theorem 1: The Handshaking Theorem

Let  $G = (V, E)$  be an undirected graph with  $e$  edges. Then  $2e = \sum_{v \in V} \deg(v)$

**Theorem 2:**

An undirected graph has an even number of vertices of odd degree.

**Proof:**

Take two sets of vertices,  $V_1$ , a set of vertices with even degree, and  $V_2$ , a set of vertices with odd degree. In an undirected graph  $G = (V, E)$  we have

$$2e = \sum_{v \in V} \deg(v) = \sum_{v \in V_1} \deg(v) + \sum_{v \in V_2} \deg(v)$$

From the equality above we can say the left part is even i.e.  $2e$  is even, the sum of  $\deg(v)$  for  $v \in V_1$  is even since every vertex has even degree. So for the left hand to be even sum of  $\deg(v)$  for  $v \in V_2$  must be even. Since all the vertices in the set  $V_2$  have odd degree the number of such vertices must be even for the sum to be even. Hence proved.

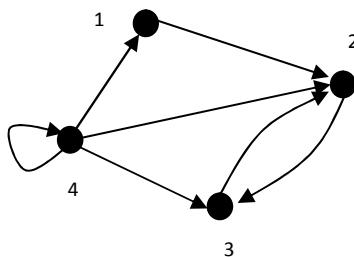
**Other Terminologies**

Let  $(u, v)$  be an edge representing edge of a directed graph  $G$ ,  $u$  is called adjacent to  $v$  and  $v$  is called adjacent from  $u$ . The vertex  $u$  is called ***initial*** vertex and the vertex  $v$  is called ***terminal*** or ***end*** vertex. Loop has same initial and terminal vertex.

In directed graph the ***in-degree*** of a vertex  $v$ , denoted by  $\deg^-(v)$ , is the number of edges that have  $v$  as their terminal vertex. The ***out-degree*** of a vertex  $v$ , denoted by  $\deg^+(v)$ , is the number of edges that have  $v$  as their initial vertex. Loop at a vertex adds up both in-degree and out-degree to one more than calculated in-degree and out-degree.

**Example:**

Find the in-degree and out-degree of each vertex in the following graph



**Solution:**

In-degrees of a graph are  $\deg^-(1) = \deg^-(4) = 1$ ;  $\deg^-(2) = 3$ ;  $\deg^-(3) = 2$  and the out-degrees of a graph are  $\deg^+(1) = \deg^+(2) = \deg^+(3) = 1$ ;  $\deg^+(4) = 4$

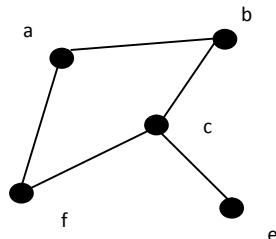
**Walk**

A **walk** is an alternating sequence of vertices and edges, beginning and ending with a vertex, where each vertex is incident to both the edge that precedes it and the edge that follows it in the sequence, and where the vertices that precede and follow an edge are the end vertices of that edge. A walk is **closed** if its first and last vertices are the same, and **open** if they are different. An open walk is called a **path**.

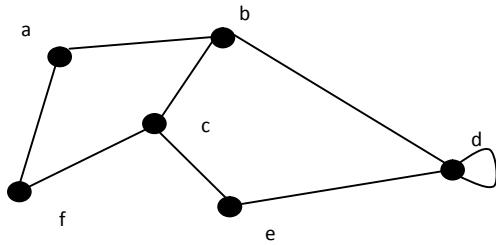
**Theorem 3:**

Let  $G(V, E)$  be a graph with directed edges. Then  $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$ .

A **subgraph** of a graph  $G = (V, E)$  is a graph  $H = (W, F)$  where  $W \subseteq V$  and  $F \subseteq E$ .

**Example:**

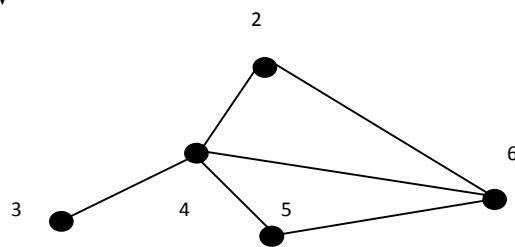
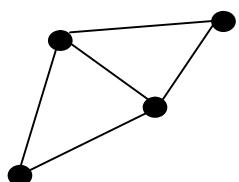
*is a subgraph of*

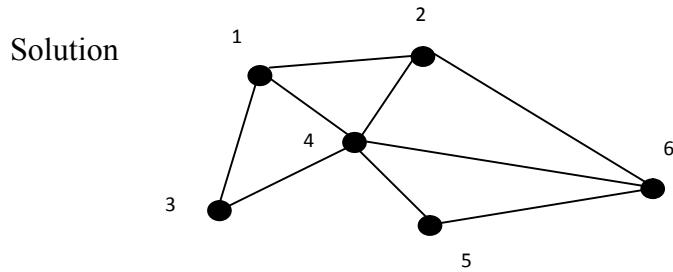


The union of two simple graphs  $G_1 = (V_1, E_1)$  and  $G_2 = (V_2, E_2)$  is the simple graph with vertex set  $V_1 \cup V_2$  and the edge set  $E_1 \cup E_2$ . The union of  $G_1$  and  $G_2$  is denoted by  $G_1 \cup G_2$ .

**Example:**

Find the union of two graphs given below





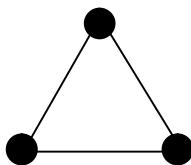
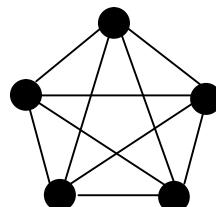
## Complete Graphs

The complete graph of  $n$  vertices, denoted by  $K_n$ , is the simple graph that contains exactly one edge between each pair of distinct vertices.

### Example:

What are  $K_1$ ,  $K_3$ , and  $K_5$ ?

### Solution:

 $K_1$  $K_3$  $K_5$ 

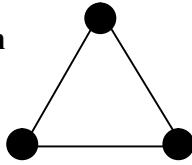
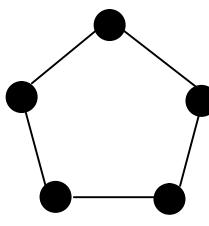
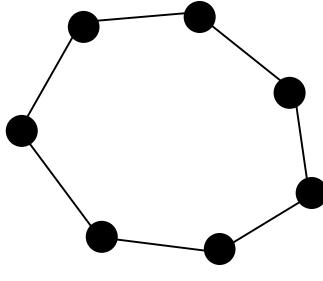
## Cycles

The cycle  $C_n$ ,  $n \geq 3$ , consists of  $n$  vertices  $v_1, v_2, \dots, v_n$  and edges  $\{v_1, v_2\}, \{v_2, v_3\}, \dots, \{v_{n-1}, v_n\}$ , and  $\{v_n, v_1\}$ .

### Example:

What are  $C_3$ ,  $C_5$ , and  $C_7$ ?

### Solution

 $C_3$  $C_5$  $C_7$

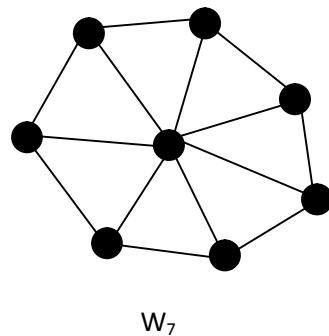
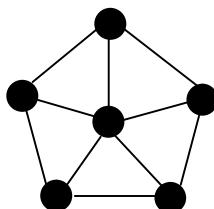
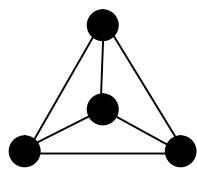
## Wheels

The wheel  $W_n$ , for  $n \geq 3$ , is an union of  $C_n$  and additional vertex where the new vertex is connected by each vertex of the cycle.

### Example:

What are  $W_3$ ,  $W_5$ , and  $W_7$ ?

### Solution:



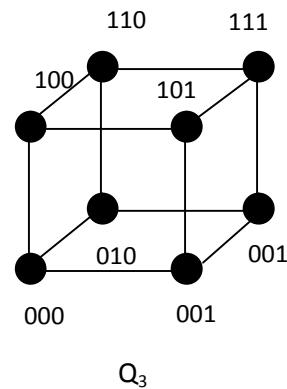
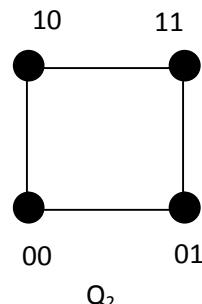
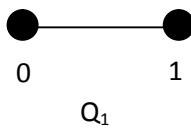
## $n$ - Cubes

The  $n$ -dimensional cube, or  $n$ -cube, denoted by  $Q_n$ , is the graph that has vertices representing the  $2^n$  bit strings of length  $n$ . Two vertices are adjacent if and only if the bit strings that they represent differ in exactly one bit position.

### Example:

What are  $Q_1$ ,  $Q_2$ , and  $Q_3$ ?

### Solution:



## Bipartite Graphs

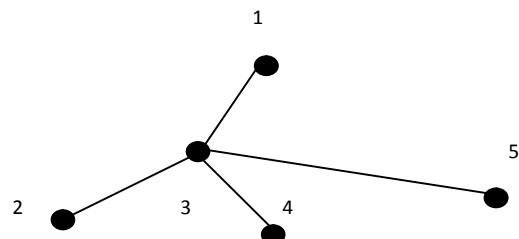
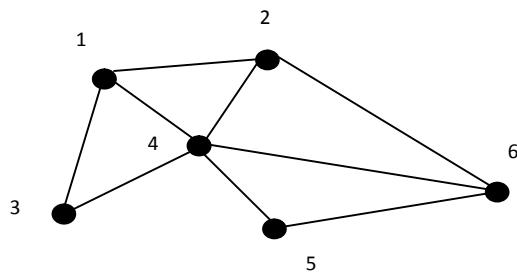
A simple graph  $G$  is bipartite if its vertex set  $V$  can be partitioned into two disjoint subsets  $V_1$  and  $V_2$  such that every edge in the graph connects a vertex from the set  $V_1$  to the vertex of the set

$V_2$ . No two vertices of the same set are connected by an edge. For example,  $C_6$  is bipartite but  $K_3$  is not.

A graph is bipartite if and only if it is possible to color the vertices of the graph with at most two colors such that no two adjacent vertices have the same color.

A graph is bipartite if and only if it is not possible to start at a vertex and return to this vertex by traversing an odd number of distinct edges.

Q. Are the graphs below bipartite



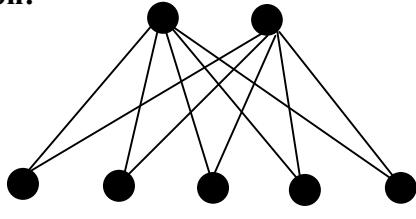
### Complete Bipartite Graphs

The complete bipartite graph  $K_{m,n}$  is the graph where the vertex set is partitioned into two subsets of  $m$  and  $n$  vertices, respectively. In this graph there is an edge between two vertices if and only if two vertices are in different subsets of vertices.

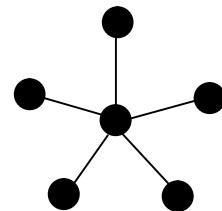
#### Example:

Sketch  $K_{2,5}$ ,  $K_{1,5}$ .

#### Solution:



$K_{2,5}$



$K_{1,5}$

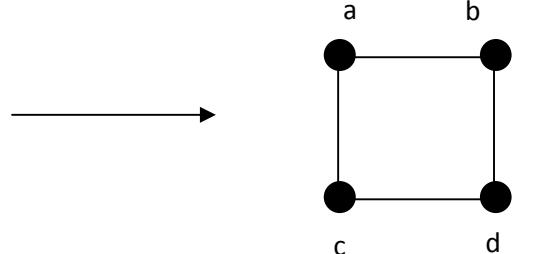
## Graph Representations

Graph can be represented in many ways. One of the ways of representing a graph without multiple edges is by listing its edges. Some other ways are described below:

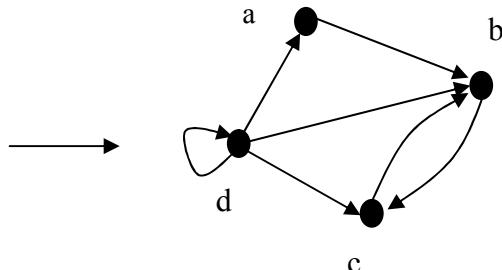
### Adjacency List

This type of representation is suitable for the undirected graphs without multiple edges, and directed graphs. This representation looks as in the tables below.

Edge List for Simple Graph	
Vertex	Adjacent Vertices
a	b, c
b	a, d
c	a, d
d	b, c



Edge List for Directed Graph	
Initial Vertex	End Vertices
a	B
b	C
c	B
d	a, b, c, d



If we try to apply the algorithms of graph using the representation of graphs by lists of edges, or adjacency lists it can be tedious and time taking if there are high numbers of edges. For the sake of the computation, the graphs with many edges can be represented in other ways. Here we discuss two ways of representing graphs in form of matrix.

## Adjacency Matrix

Given a simple graph  $G = (V, E)$  with  $|V| = n$ . Assume that the vertices of the graph are listed in some arbitrary order like  $v_1, v_2, \dots, v_n$ . The adjacency matrix  $A$  of  $G$ , with respect to the order of the vertices is  $n$ -by- $n$  zero-one matrix ( $A = [a_{ij}]$ ) with the condition,

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of } G \\ 0 & \text{otherwise} \end{cases}$$

Since there are  $n$  vertices and we may order vertices in any order there are  $n!$  possible order of the vertices. The adjacency matrix depends on the order of the vertices, hence there are  $n!$  possible adjacency matrices for a graph with  $n$  vertices.

Adjacency matrix for undirected graph is symmetric, in case of the pseudograph or multigraph the representation is similar but the matrix here is not zero-one matrix rather the  $(i, j)^{\text{th}}$  entry of the matrix contains the number of edges appearing between that pair of vertices.

In case of the directed graph we can extend the same concept as in undirected graph as dictated by the relation

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$

The main difference is that the matrix may not be symmetric.

If the number of edges is few then the adjacency matrix becomes sparse. Sometimes it will be beneficial to represent graph with adjacency list in such a condition.

## Incidence Matrix

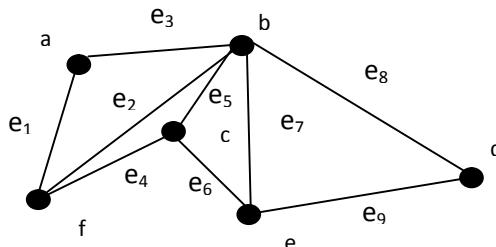
This is another way of representing graph. Given an undirected graph  $G = (V, E)$ . Assume that the vertices of the graph are  $v_1, v_2, \dots, v_n$  and the edges of the graph are  $e_1, e_2, \dots, e_m$ . The incidence matrix of a graph with respect to the above ordering of  $V$  and  $E$  is  $n$ -by- $m$  matrix  $M = [m_{ij}]$ , where

$$m_{ij} = \begin{cases} 1 & \text{when edge } e_j \text{ is incident with } v_i, \\ 0 & \text{otherwise.} \end{cases}$$

When the graph is not simple then also the graph can be represented by using incidence matrix where multiple edges corresponds to two different columns with exactly same entries. Loops are represented with column with only one entry.

### Example 1:

Represent the following graph using adjacency matrix and incidence matrix.



### Solution:

Let the order of the vertices be a, b, c, d, e, f and edges order be e1, e2, e3, e4, e5, e6, e7, e8, e9.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

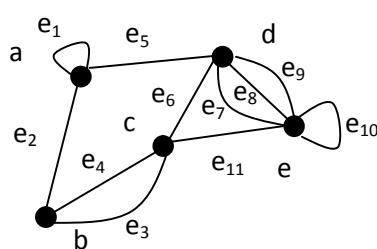
Adjacency Matrix

$$\begin{array}{l} a \quad e_1 \quad e_2 \quad e_3 \quad e_4 \quad e_5 \quad e_6 \quad e_7 \quad e_8 \quad e_9 \\ b \quad [1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0] \\ c \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \\ d \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \\ e \quad 0 \quad 1 \quad 1 \\ f \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \\ 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \end{array}$$

Incidence Matrix

### Example 2:

Represent the following graph using adjacency matrix and incidence matrix.



**Solution:**

Let the order of the vertices be a, b, c, d, e and edges order be e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11.

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 2 & 0 & 0 \\ 0 & 2 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 3 & 1 \end{bmatrix}$$

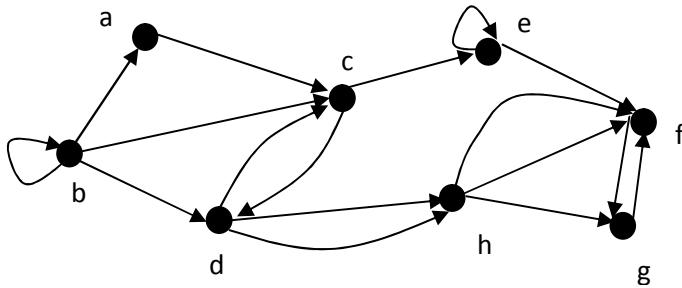
Adjacency Matrix

$$\begin{array}{ll} & \begin{matrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & e_9 & e_{10} & e_{11} \end{matrix} \\ a & \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \\ b & \\ c & \\ d & \\ e & \end{array}$$

Incidence Matrix

**Example 3:**

Represent the following directed graph using adjacency matrix.

**Solution:**

Let the order of the vertices be a, b, c, d, e, f, g and h. The adjacency matrix is

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 1 & 0 \end{bmatrix}$$

## Isomorphism of Graphs

The two simple graphs  $G_1 = (V_1, E_1)$  and  $G_2 = (V_2, E_2)$  are isomorphic if there is a one-to-one and onto function  $f$  from  $V_1$  to  $V_2$  with the property that  $a$  and  $b$  are adjacent in  $G_1$  if and only if  $f(a)$  and  $f(b)$  are adjacent in  $G_2$ , for all  $a$  and  $b$  in  $V_1$ . Such a function  $f$  is called isomorphism.

In other words, two simple graphs are called isomorphic if there is a one-to-one correspondence between vertices of the two graphs that preserves the adjacency relationship. Also, isomorphism of simple graphs is an equivalence relationship.

Example: Show that the graphs  $G = (V, E)$  and  $H = (W, F)$  displayed below are isomorphic.



### Solution

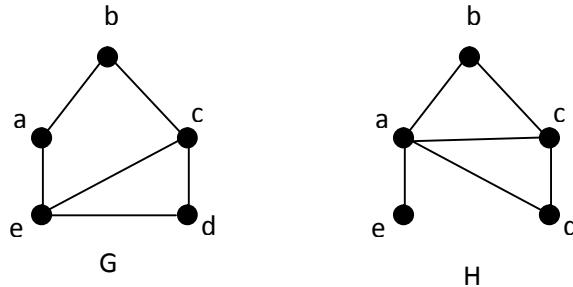
The function  $f$  with  $f(a) = p$ ,  $f(b) = s$ ,  $f(c) = r$ , and  $f(d) = q$  is a one-to-one correspondence between  $V$  and  $W$ . Also, this correspondence preserves adjacency because adjacent vertices in  $G$  are  $a$  and  $b$ ,  $a$  and  $c$ ,  $b$  and  $d$ , and  $c$  and  $d$ , and each of the pairs  $f(a) = p$  and  $f(b) = s$ ,  $f(a) = p$  and  $f(c) = r$ ,  $f(b) = s$ , and  $f(d) = q$ , and  $f(c) = r$  and  $f(d) = q$  are adjacent in  $H$ .

Determining whether two graphs are isomorphic or not is a difficult task since there are  $n!$  possible one-to-one correspondence between the vertex sets of two simple graphs with  $n$  vertices if  $n$  is large.

However we can show that two simple graphs are not isomorphic by showing that they do not share a property that isomorphic simple graphs must both have. Such a property is called invariant. These properties are described as follows:

1. Isomorphic simple graphs must have the same number of vertices (one-to-one correspondence between vertices of two graphs is required).
2. Isomorphic simple graphs also must have the same number of edges (due to adjacency preservation).
3. The degrees of the vertices in isomorphic simple graphs must be same because the number of edges from the vertex is determined by degree.

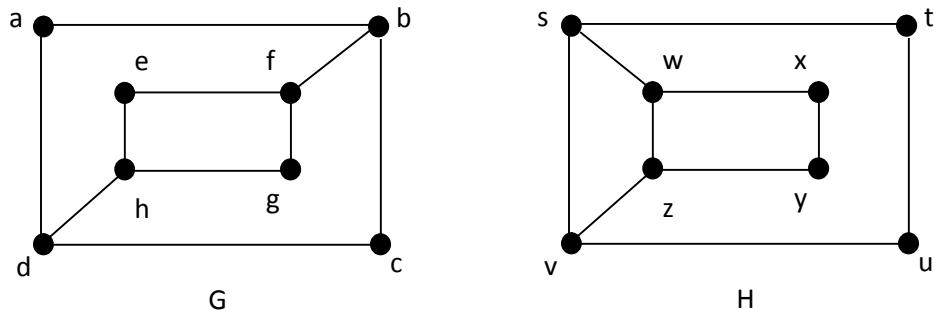
**Example:** Show that the graphs displayed below are not isomorphic.



**Solution:**

Both graphs have five vertices and six edges. However, H has a vertex of degree one, namely e, whereas G has no vertices of degree one. Hence G and H are not isomorphic.

When these invariants are the same, it does not necessarily mean that the two graphs are isomorphic. There are no useful sets of invariants currently known that can be used to determine whether simple graphs are isomorphic.



**Example:**

Determine whether the graphs shown below are isomorphic.

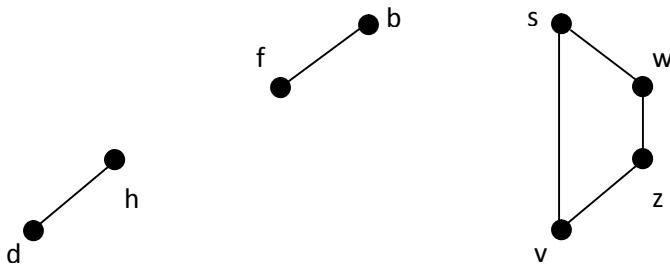
**Solution:**

These both graphs have eight vertices and ten edges. Also, they both have four vertices of degree two and four vertices of degree three.

However, G and H are not isomorphic. Since,  $\deg(a) = 2$  in G, a must correspond to either t, u, x, or y in H because these are the vertices of degree two in H. Here, each of these four vertices in H is adjacent to another vertex of degree two in H, which is not true for a in G.

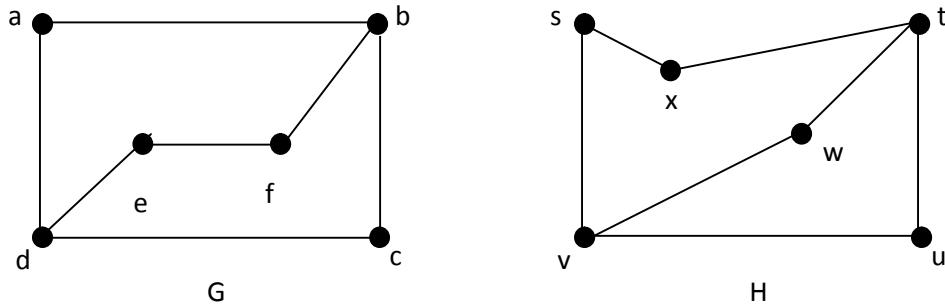
Another way to see that two graphs are not isomorphic is to note that the subgraphs formed by

connecting the edges from the vertex with same degree in both the graphs are not isomorphic. For example, subgraphs of G and H made up of vertices of degree three and the edges connecting them in the above figure are not isomorphic. Hence, G and H are not isomorphic. The figure below shows subgraphs of G and H made up of vertices of degree three.



To show isomorphism of graphs, we can also use adjacency matrix. For this we should show that adjacency matrices of the two graphs are same. For this, we may need to arrange vertices in the adjacency matrix.

**Example:** Determine whether the graphs displayed below are isomorphic.



**Solution:**

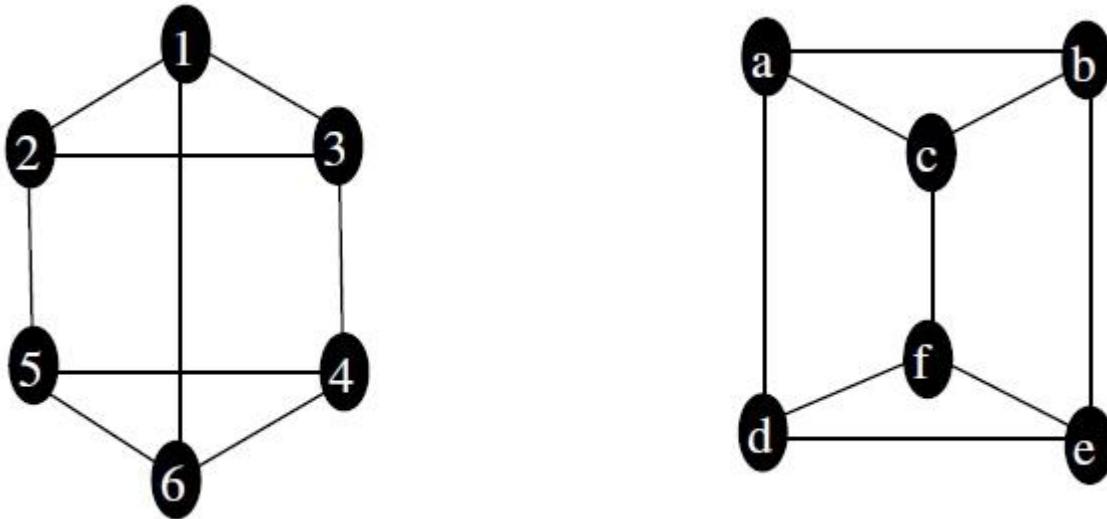
Both graphs have six vertices and seven edges. Both have four vertices of degree two and two vertices of degree three. Also, subgraphs are isomorphic. Here we cannot say that these graphs are isomorphic or not. For this we can use adjacency matrix with the order of vertices a, b, c, d, e, f and w, t, u, v, s, x. If we draw the adjacency matrices with the above mentioned orders of vertices, we can see similar two matrices. Hence these graphs are isomorphic.

**Note:**

If both adjacency matrices are not same, we cannot say that the two graphs are not isomorphic because another correspondence of the vertices may be same.

**Example**

Determine whether the given two graphs are isomorphic or not?



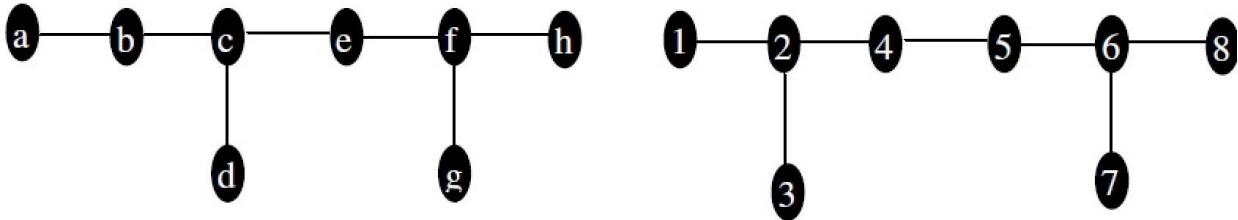
Solution:

In the above two graphs number of vertices in both graphs is same (i.e. 6), number of edges equal to 9 in both the graphs and all the vertices in both the graphs have degree 3.

Since the invariants agree in both the graphs, we can try out to find the function that is isomorphism. Take the sequence of vertices from the first graph as 1, 2, 3, 4, 5, and 6. Now define  $f(1) = c$ ,  $f(2) = a$  here there is adjacency preservation since we have  $\{1, 2\}$  as an edge in the first graph where as  $\{f(1), f(2)\} = \{c, a\}$  is an edge in the second graph. Similarly we can assign  $f(3) = b$ ,  $f(4) = e$ ,  $f(5) = d$ ,  $f(6) = f$ . Since we found one to one correspondence between vertices of two graphs persevering the adjacency, the above two graphs are isomorphic. We can note that the adjacency matrices of two isomorphic graphs in which the vertices are ordered in terms of function i.e. in our example 1, 2, 3, 4, 5, and 6 for the first graph and c, a, b, e, d, and f in the second graph are same. In the above two graphs note that the number of circuits of same length in both the graphs is same.

**Example:**

Determine whether the given two graphs are isomorphic or not?

**Solution:**

In the above two graphs number of vertices in both graphs is 8, the number of edges equal to 7 in both the graphs, in both graphs two vertices have degree 3, 4 vertices have degree 1 and the remaining 2 vertices have degree 2. Since the invariants agree in both the graphs, we can continue to get the function such that it is isomorphism. However, in case of first graph the subgraph containing the vertex c (degree 3), with vertices a, b, c, d, and e is not isomorphic with any of the subgraph formed by connecting edges with vertex 2 or 6 (both of degree 3). Hence the two above graphs are not isomorphic.

**Graph Connectivity**

Many problems can be modeled with paths formed by travelling along the edges of the graphs. For example, problem of determining whether a message can be sent between two computers using intermediate links, problems of efficiently planning routes for mail delivery, garbage pickup, diagnostics in computer networks, and so on.

**Path**

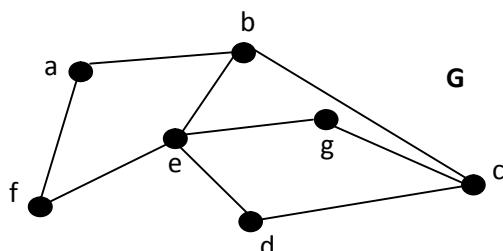
Informally, a path is a sequence of edges that begins at a vertex of a graph and travels along edges of the graph, always connecting pairs of adjacent vertices. A formal definition is given below.

In an undirected graph  $G$ , a path of length  $n$ , where  $n$  is a nonnegative integer, from  $a$  to  $b$  in  $G$  is a sequence of  $n$  edges  $e_1, e_2, \dots, e_n$  of  $G$  such that  $f(e_1) = \{x_0, x_1\}$ ,  $f(e_2) = \{x_1, x_2\}$ , ...,  $f(e_n) = \{x_{n-1}, x_n\}$ , where  $a = x_0$  and  $b = x_n$ . When the graph is simple, we denote this path by a vertex sequence  $x_0, x_1, \dots, x_n$ . If  $a = b$  and the path length is greater than zero, then the path is called circuit or cycle. The path or circuit is said to pass through

the vertices  $x_1, x_2, \dots, x_{n-1}$  or traverse the edges  $e_1, e_2, \dots, e_n$ . A path or circuit is simple if it does not contain the same edge more than once.

When it is not necessary to distinguish between multiple edges, we will denote a path  $e_1, e_2, \dots, e_n$ , where  $f(e_i) = \{x_{i-1}, x_i\}$  for  $i = 1, 2, \dots, n$  by its vertex sequence  $x_0, x_1, \dots, x_n$ . A path of length zero consists of a single vertex.

Example: In the simple graph  $G$  shown below  $a, b, e, g, c$  is a simple path of length 4 since  $\{a, b\}$ ,  $\{b, e\}$ ,  $\{e, g\}$ , and  $\{g, c\}$  are all edges. However,  $a, b, g, c$  is not a path since  $\{b, g\}$  is not an edge. Note that  $a, b, e, f, a$  is a circuit of length 4 since  $\{a, b\}$ ,  $\{b, e\}$ ,  $\{e, f\}$ ,  $\{f, a\}$  are edges, and this path begins and ends at  $b$ . The path  $a, b, e, f, a, b$ , which is of length 5, is not simple since it contains the  $\{a, b\}$  twice.

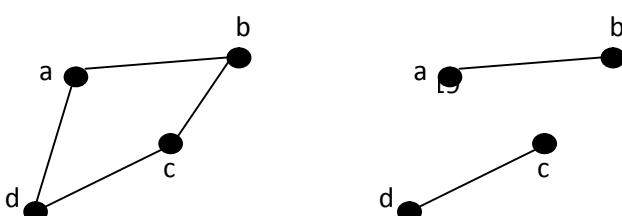


Let  $n$  be a nonnegative integer and  $G$  a directed multigraph. A path of length  $n$ , where  $n$  is a nonnegative integer, from  $a$  to  $b$  in  $G$  is a sequence of  $n$  edges  $e_1, e_2, \dots, e_n$  of  $G$  such that  $f(e_1) = (x_0, x_1)$ ,  $f(e_2) = (x_1, x_2)$ , ...,  $f(e_n) = (x_{n-1}, x_n)$ , where  $a = x_0$  and  $b = x_n$ . When there is no multiple edges in the directed graph, this path is denoted by its vertex sequence  $x_0, x_1, \dots, x_n$ . If  $a = b$  and the path length is greater than zero, then the path is called circuit or cycle. A path or circuit is simple if it does not contain the same edge more than once.

Note that the terminal vertex of an edge in a path is the initial vertex of the next edge in the path. When it is not necessary to distinguish between multiple edges, we will denote a path  $e_1, e_2, \dots, e_n$ , where  $f(e_i) = (x_{i-1}, x_i)$  for  $i = 1, 2, \dots, n$  by its vertex sequence  $x_0, x_1, \dots, x_n$ .

## Connectedness in Undirected Graphs

An undirected graph is called connected if there is a path between every pair of distinct vertices



of the graph. The first graph given below is connected whereas the second graph is not. A graph that is not connected is the union of more than one connected graphs that do not share the common vertex. These disjoint connected subgraphs are called **connected component** of a graph.

**Theorem:**

There is a simple path between every pair of distinct vertices of a connected undirected graph.

**Proof:**

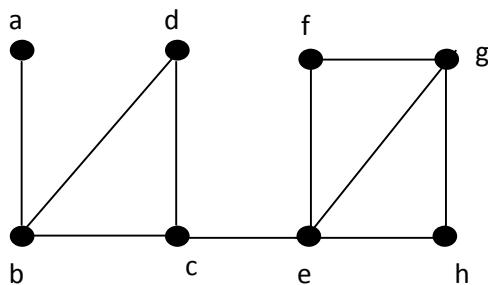
Suppose  $a$  and  $b$  are two distinct vertices of the connected undirected graph  $G = (V, E)$ . We know that  $G$  is connected so by definition there is at least one path between  $a$  and  $b$ . Let  $x_0, x_1, \dots, x_n$ , where  $x_0 = a$  and  $x_n = b$ , be the vertex sequence of a path of a least length. Now if this path of the least length is not simple then we have  $x_i = x_j$ , for some  $i$  and  $j$  with  $0 \leq i < j$ . This implies that there is a path from  $a$  to  $b$  of shorter length with the vertex sequence  $x_0, x_1, \dots, x_i, \dots, x_{j+1}, \dots, x_n$  obtained by removing the edges corresponding to the vertex sequence  $x_{i+1}, \dots, x_j$ . This shows that there is a simple path.

**Cut vertices (articulation points)** are those vertices in the graph whose removal along with the edges incident on them produces subgraph with more connected components than in the original graph.

**Cut edge (bridge)** is an edge whose removal produces a graph with more connected components than in the original graph. A **cut set** of a graph is a set of edges such that the removal of these edges produces a subgraph with more connected components than in the original graph, but no proper subset of this set of edges has this property.

**Example:**

Find the cut vertices and cut edges in the graph given below.

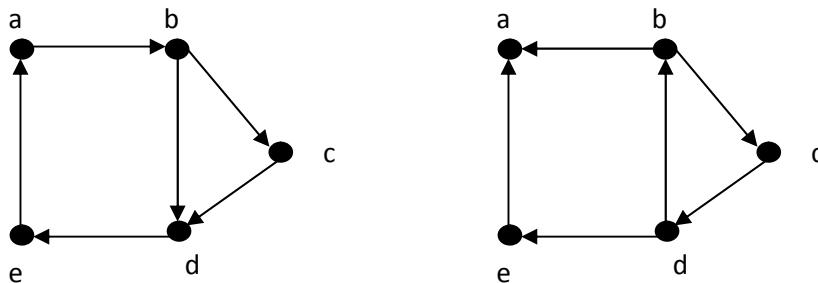


**Solution:**

The cut vertices are b, c, and e. the removal of one of these vertices and its adjacent edges disconnects the graph. The cut edges are {a, b} and {c, e}. Removing either one of these edges disconnects the graph.

**Connectedness in Directed Graphs**

A directed graph is **strongly connected** if there is a path from a to b and from b to a whenever a and b are vertices in the graph. A directed graph is **weakly connected** if there is a path between every two vertices in the underlying undirected graph. The subgraphs of a directed graph G that are strongly connected but not contained in larger strongly connected subgraphs are called **strongly connected components** or **strong components** of G.

**Example:**

Are the directed graphs shown above strongly connected? Are they weakly connected?

**Solution:**

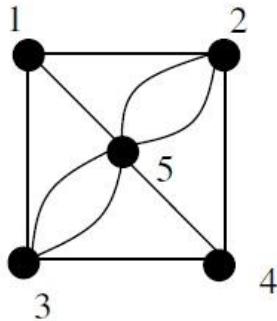
The first graph is strongly connected because there is a path between any two vertices in this directed graph. Hence it is also weakly connected. The second graph is not strongly connected. There is no directed path from a to b in this graph. However, this graph is weakly connected, since there is a path between any two vertices in the underlying undirected graph of this graph.

**Euler Paths and Circuits**

An Euler circuit in a graph G is a simple circuit containing every edge of G. An Euler path in G is a simple path containing every edge of G.

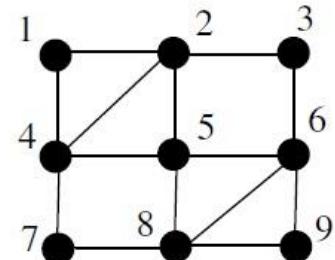
**Example:**

Find the Euler path or circuit in the following graphs



No Euler circuit exist, but the Euler path is  
1,2,5,1,3,5,4,3,5,2,4

The Euler circuit is 1,2,3,6,9,8,7,4,5,8,6,5,2,4,1



## Necessary and Sufficient Conditions for Euler Circuits and Paths

### Theorem 5:

A connected multigraph has an Euler circuit if and only if each of its vertices has even degree.

### Proof:

Take a connected multigraph  $G = (V, E)$  where  $V$  and  $E$  are finite. We can prove the theorem in two parts.

First we prove that if a connected multigraph has an Euler circuit, then all the vertices have even degree. For this, take a vertex  $v$ , where the Euler circuit begins. There is some edge that is incident to  $v$  and some other vertex say  $u$  then we have an edge  $\{v, u\}$ . This edge  $\{v, u\}$  contributes one to the degree of  $v$  and  $u$  both. Again there must be some edge other than  $\{v, u\}$  that is incident to  $u$  and some other vertex. In this case the total degree of the vertex  $u$  becomes even, so whenever in the circuit the vertex is met the degree of that vertex is even since every time entering and leaving the vertex gives even degree to all the vertices other than the initial vertex. However since the circuit must terminate in the vertex  $v$  and the edge that is terminating the circuit contributes one to the degree of the initial vertex  $v$  the total degree of the vertex  $v$  is also even. Now we have every time the vertex is entered and left it gives even degree and the initial vertex also gives even degree, we can conclude that if a graph has Euler circuit, then all the vertices have even degree. Now we try to prove that if all the vertices in the connected multigraph have even degree, the there exist Euler circuit. For this, take a connected multigraph  $G$  with all the vertices having even degree. To make a circuit start at arbitrary vertex, say  $a$  of  $G$ .

now start from the vertex  $a = x_0$  and arbitrarily choose other vertex  $x_1$  to form and edge  $\{x_0, x_1\}$ .

Continue building the simple path  $\{x_0, x_1\}, \{x_1, x_2\}, \dots, \{x_{n-1}, x_n\}$ . This path terminates since it has a finite number of edges. It begins at  $a$  with an edge  $\{a, x\}$  and terminate at  $a$  with some edge  $\{y, a\}$ . This is correct since every vertex has even degree in the graph we are considering, if an edge left some vertex then there must be an edge entering that vertex to make its degree even. Now we have shown that there exists simple circuit in the graph with all the vertices of even degree. If this circuit has all the edges of the graph in it, then the simple circuit is itself an Euler circuit. If all the edges are not in the circuit, then we have next possibility. Now, consider the subgraph, say  $H$  that is formed by removing all the edges that are already in the simple circuit formed above and by removing the isolated vertices after edges are removed. Since the original graph  $G$  is connected, there must be at least one vertex of  $H$  that is common with the circuit we have formed. Let  $w$  be such a vertex. Every vertex in  $H$  has even degree since it is a subgraph of original graph. In case of  $w$ , while forming the circuit pairs of incident edges are used up. So the degree of  $w$  is again even. Beginning at  $w$  we can build a simple circuit as described above. We can continue this process until all edges have been used. Now if we combine the formed circuit in a way that it makes use of common vertex to make a circuit then we can say that the circuit is an Euler circuit. Hence if every vertices of a connected graph has an even degree then it has an Euler circuit. This concludes the proof.

### **Theorem 6:**

A connected multigraph has an Euler path but not Euler circuit if and only if it has exactly two vertices of odd degree.

### **Proof:**

This fact can be proved if we can prove that first, if the connected multigraph has Euler path exactly two vertices have odd degree and second if the connected multigraph has exactly two vertices of odd degree, then it has Euler path.

Now, if the graph (in this proof graph means connected multigraph) has an Euler path say from  $a$  to  $z$  but not Euler circuit, then it must pass through every edge exactly once. In this scenario the first edge in the path contributes one to the degree of vertex  $a$ , and at all other time when other edges pass through vertex  $a$  it contributes twice to the degree of  $a$ , hence we can say that degree

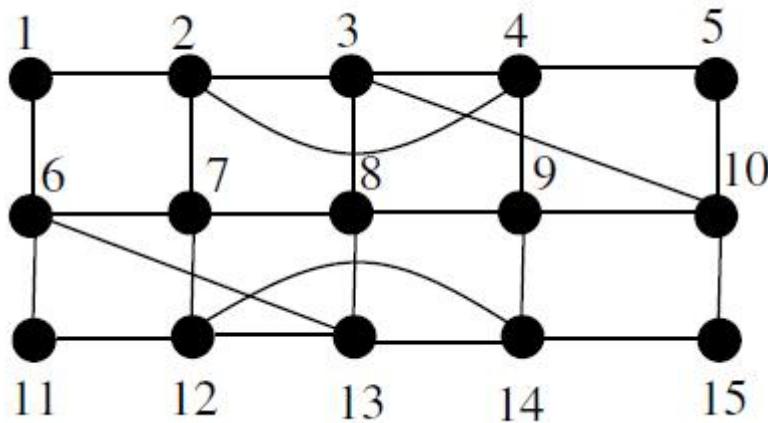
of  $a$  is odd. Similarly the last edge in the path coming to  $z$  contributes one to the degree of  $z$ , all the other edges contributes two one for entering and one for leaving. Here also the degree of last vertex,  $z$  is odd. All the other vertices other than  $a$  and  $z$  must have even degree since the edges in those vertices enter and leave the vertex contributing two to the degree every time the vertices are met. Hence if there is an Euler path but not an Euler circuit, exactly two vertices of the graph have odd degree.

Secondly, if exactly two vertices of a graph have odd degree and let's consider they are  $a$  and  $z$ . Now, consider another graph that adds an edge  $\{a, z\}$  to the original graph, then the newly formed graph will have every vertex of even degree. So there exists Euler circuit in the new graph and the removal of the new edge gives us the Euler path in the original graph. Hence if exactly two vertices of the graph have an odd degree, then the graph has an Euler path but not Euler circuit.

This concludes the proof.

### Example:

Find Euler path or circuit?



### Solution:

In the above graph, all the vertices have even degree, hence there is an Euler circuit. The Euler circuit is 1,2,3,4,5,10,15,14,13,12,11,6,13,8,9,14,12,7,8,3,10,9,4,2,7,6,1

## Hamilton paths

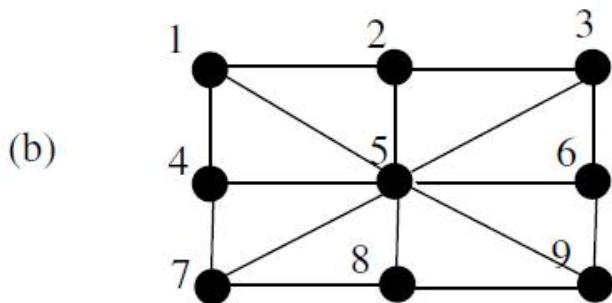
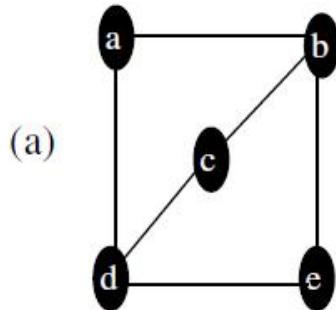
A path  $x_0, x_1, \dots, x_{n-1}, x_n$  in the graph  $G = (V, E)$  is called Hamilton path if  $V = \{x_0, x_1, \dots, x_{n-1}, x_n\}$  and  $x_i \neq x_j$  for  $0 \leq i < j \leq n$ . A circuit  $x_0, x_1, \dots, x_{n-1}, x_n, x_0$ , with  $n > 1$ , in a graph  $G = (V, E)$  is a Hamilton circuit if  $x_0, x_1, \dots, x_{n-1}, x_n$  is a Hamilton path.

A graph with a vertex of degree one cannot have a Hamilton circuit.

If a vertex in a graph has degree 2, then both edges incident with this vertex must be part of Hamilton cycle.

### Example:

Find Hamilton circuit from the following graph if exists? What about Hamilton path?



### Solution:

In graph (a) there is no Hamilton circuit since the node c has degree 2 and both the edges from it must be in Hamilton circuit, which is not possible. One of the Hamilton path in the graph (a) is a, b, c, d, e.

In graph (b) we can find Hamilton circuit, the circuit can be 1,2,3,5,6, 9,8,7,4,1. Since there is circuit we can have path also.

## Theorem 7: Dirac's Theorem

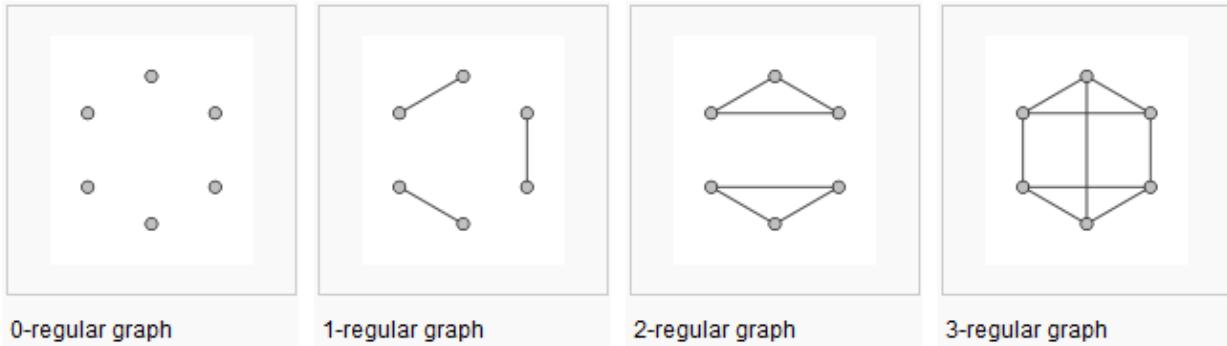
If  $G$  is a simple graph with  $n$  vertices with  $n \geq 3$  such that the degree of every vertex in  $G$  is at least  $\frac{n}{2}$ , then  $G$  has a Hamilton circuit.

## Theorem 8: Ore's Theorem

If  $G$  is a simple graph with  $n$  vertices with  $n \geq 3$  such that  $\deg(u) + \deg(v) \geq n$  for every pair of nonadjacent vertices  $u$  and  $v$  in  $G$ , then  $G$  has a Hamilton path.

## Regular Graphs

In graph theory, a regular graph is a graph where each vertex has the same number of neighbors; i.e. every vertex has the same degree. A regular graph with vertices of degree  $k$  is called a  $k$ -regular graph or regular graph of degree  $k$ . Regular graphs of degree at most 2 are easy to classify: A 0-regular graph consists of disconnected vertices, a 1-regular graph consists of disconnected edges, and a 2-regular graph consists of disconnected cycles. A 3-regular graph is known as a cubic graph.

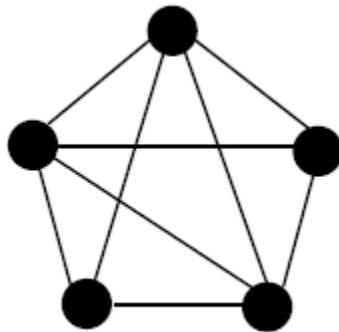


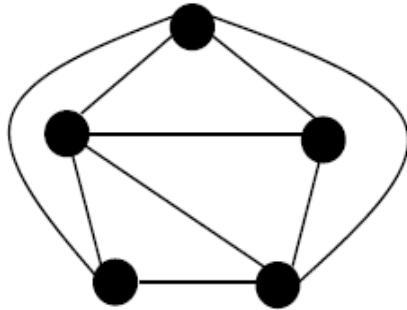
## Planar graphs

A graph is called a planar if it can be drawn in the plane without any edges crossing. Such drawing is called a planar representation of the graph.

### Example:

Draw the graph below as planar representation of the graph.



**Solution****Theorem 9: Euler's Formula**

Let  $G$  be a connected planar simple graph with  $e$  edges and  $v$  vertices. Let  $r$  be the number of regions in a planar representation of  $G$ . Then  $r = e - v + 2$ .

**Example:**

Suppose that a connected planar graph has 30 edges. If a planar representation of this graph divides the plane into 20 regions, how many vertices does this graph have?

**Solution:**

We have,  $r = 20$ ,  $e = 30$ , so by Euler's formula we have  $v = e - r + 2 = 30 - 20 + 2 = 12$ . So the number of vertices is 12.

**Corollary 1:**

If  $G$  is a connected planar simple graph with  $e$  edges and  $v$  vertices where  $v \geq 3$ , then  $e \leq 3v - 6$ .

**Corollary 2:**

If  $G$  is a connected planar simple graph, then  $G$  has a vertex of degree not exceeding five.

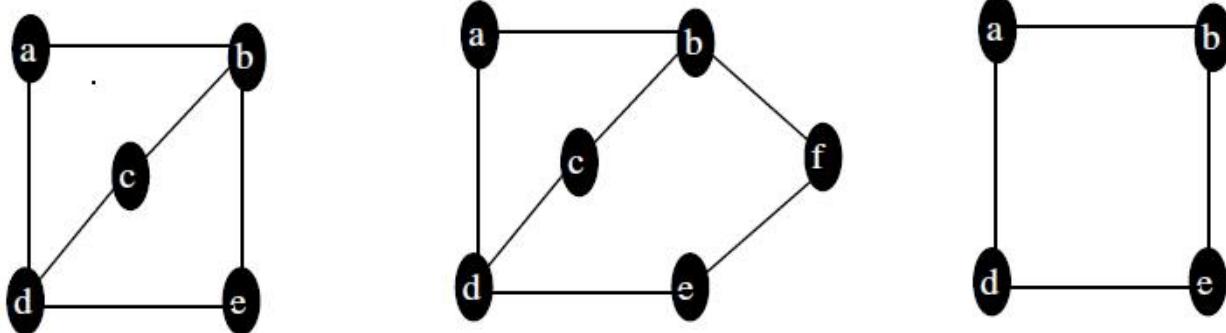
**Corollary 3:**

If a connected planar simple graph has  $e$  edges and  $v$  vertices with  $v \geq 3$  and no circuits of length three, then  $e \leq 2v - 4$ .

If a graph is planar, so will be any graph obtained by removing an edge  $\{u, v\}$  and adding a new vertex  $w$  together with edges  $\{u, w\}$  and  $\{w, v\}$ . Such an operation is called an elementary subdivision. The graphs  $G_1 = (V_1, E_1)$  and  $G_2 = (V_2, E_2)$  are called homeomorphic if they can be obtained from the same graph by a sequence of elementary subdivisions.

**Example:**

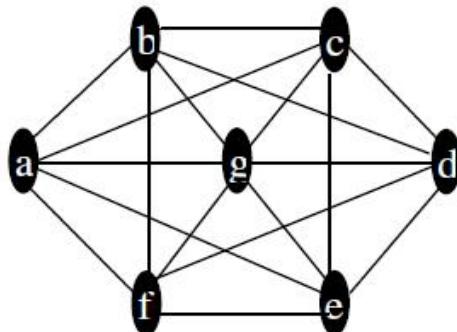
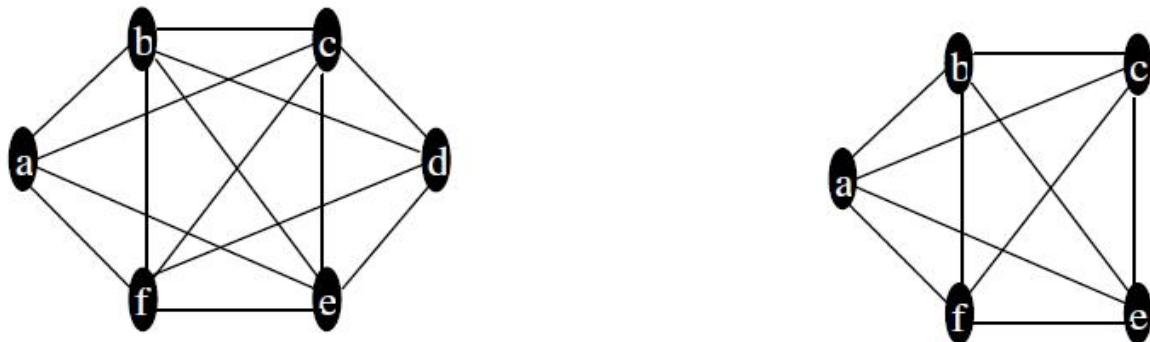
The below example graphs are homeomorphic to the third graph.

**Theorem 10: Kuratowski's Theorem (Planarity Testing Algorithm)**

A graph is nonplanar if and only if it contains a subgraph homeomorphic to  $K_{3,3}$  or  $K_5$ .

**Example:**

Determine whether the following graph is planar or not?

**Solution:**

Above we saw that the graph is homeomorphic to  $K_5$ , the given graph is not planar.

## A Shortest – Path Algorithm

There are several algorithms that find a shortest path between two vertices in a weighted graph. Graphs that have a number assigned to each edge are called **weighted graphs**.

### Algorithm (Dijkstra's Algorithm)

**Procedure** *Dijkstra(G : weighted connected simple graph, with all weighted positive)*

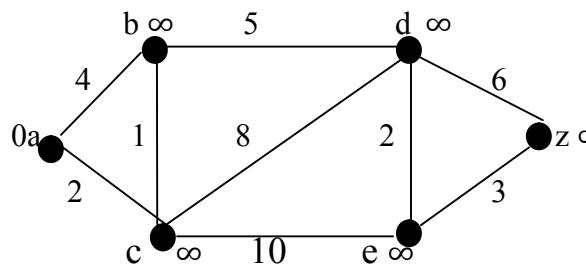
- *G has vertices  $a = v_0, v_1, \dots, v_n = z$  and weights  $w(v_i, v_j)$  where  $w(v_i, v_j) = \infty$  if  $(v_i, v_j)$  is not an edge in G*
- **for**  $i = 1$  **to**  $n$ 
  - $L(v_i) = \infty$
- $L(a) = 0$
- $S = \emptyset$
- *{the labels are now initialized so that the label of a is 0 and all others label are  $\infty$ , and S is the empty set}*
- **while**  $z \notin S$
- **begin**
  - $u = a$  vertex not in  $S$  with  $L(u)$  minimal
  - $S = S \cup \{u\}$
  - for** all vertices  $v$  not in  $S$ 
    - if**  $L(u) + w(u, v) < L(v)$  **then**  $L(v) = L(u) + w(u, v)$
    - (this adds a vertex to S with minimum label and updates the labels of vertices not in S)*
- **end** *{ $L(z) = \text{length of the shortest path from } a \text{ to } z$ }*

### Algorithm Tracing

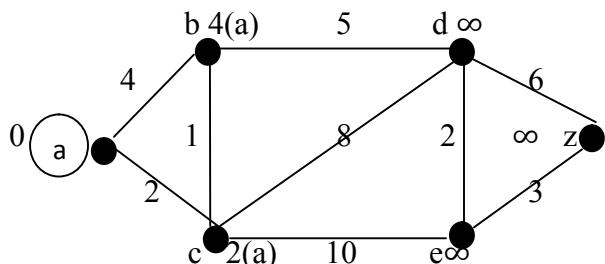
Use Dijkstra's algorithm to find the shortest path from the vertices  $a$  to  $z$  in following weighted graph given in figure (a).

### Solution

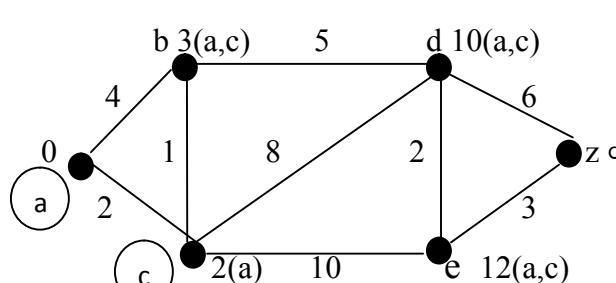
The steps used by Dijkstra's algorithm to find the shortest path between  $a$  and  $z$  are shown below.



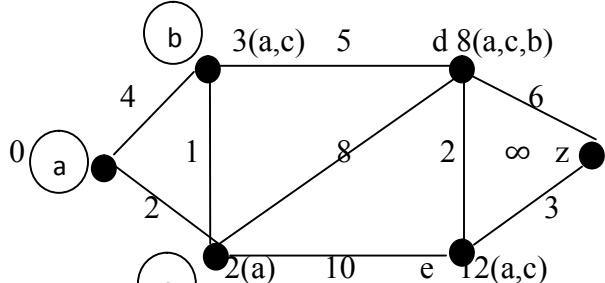
(a)



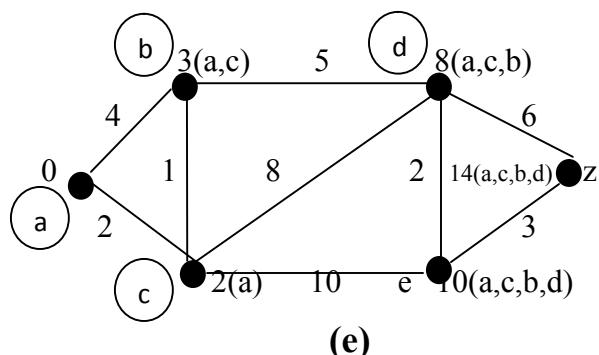
(b)



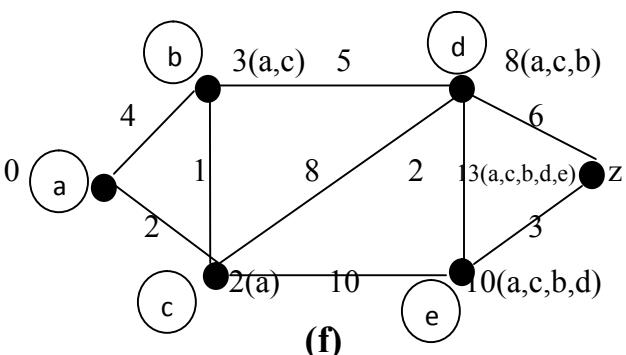
(c)



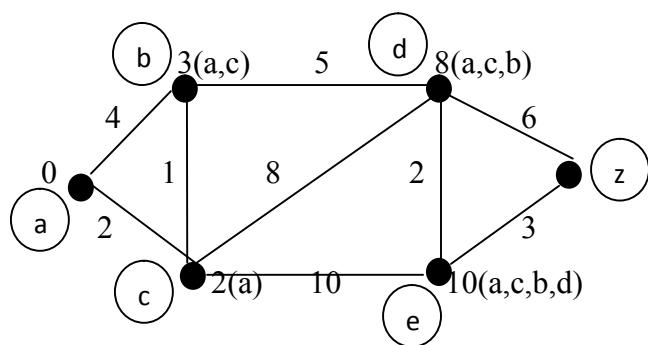
(d)



(e)



(f)



(g)

## Graph coloring

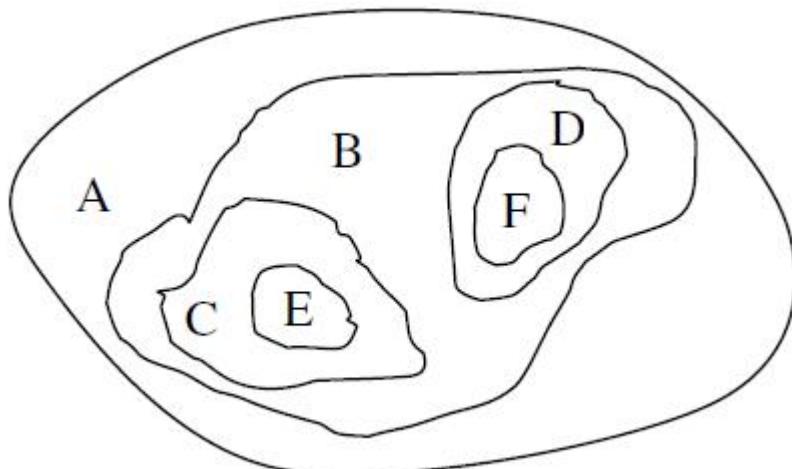
A coloring of a simple graph is the assignment of a color to each vertex of the graph so that no two adjacent vertices are assigned the same color. A chromatic number of a graph is the least number of colors needed for a coloring of this graph.

### Theorem 11: The Four Color Theorem

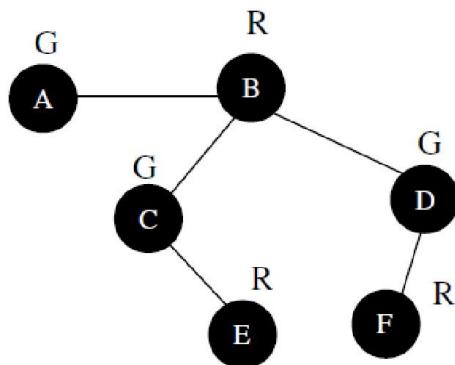
The chromatic number of a planar graph is no greater than four. To show the chromatic number of a graph as  $k$  we must show that the graph can be colored using  $k$  colors and the given graph cannot be colored using fewer than  $k$  colors.

#### Example:

Construct the dual graph for the map shown. Then find the number of colors needed to color the map so that no two adjacent regions have the same color.



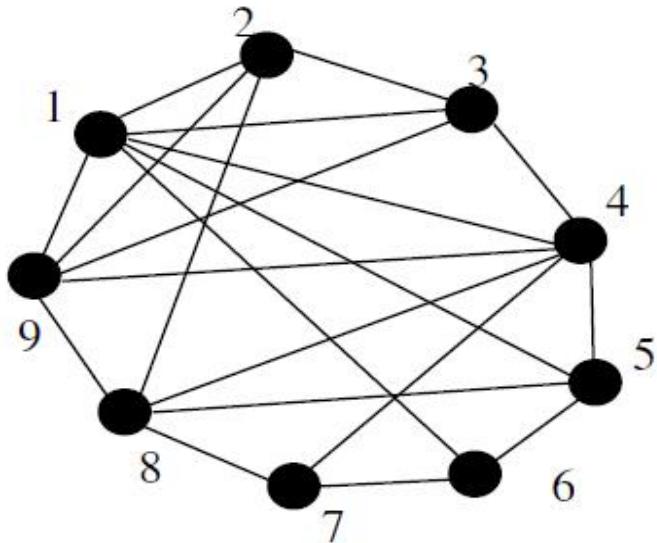
#### Solution



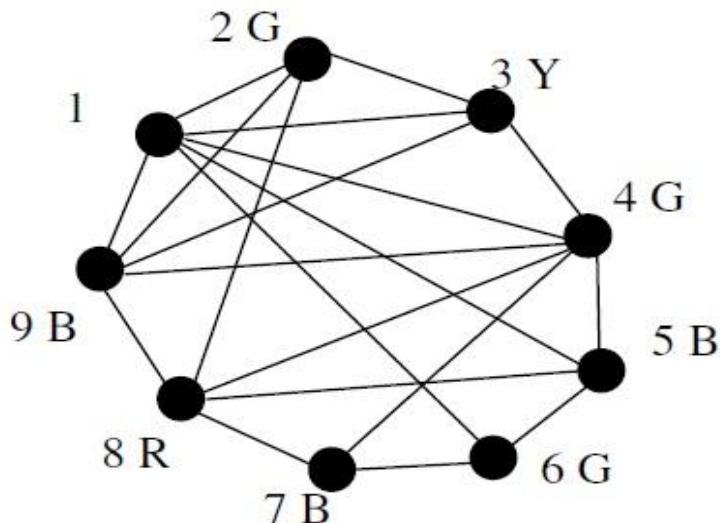
We can color the graph with at most 2 colors as shown in the graph. Take R and G as red and green

**Example:**

Find the chromatic number of the graph below.

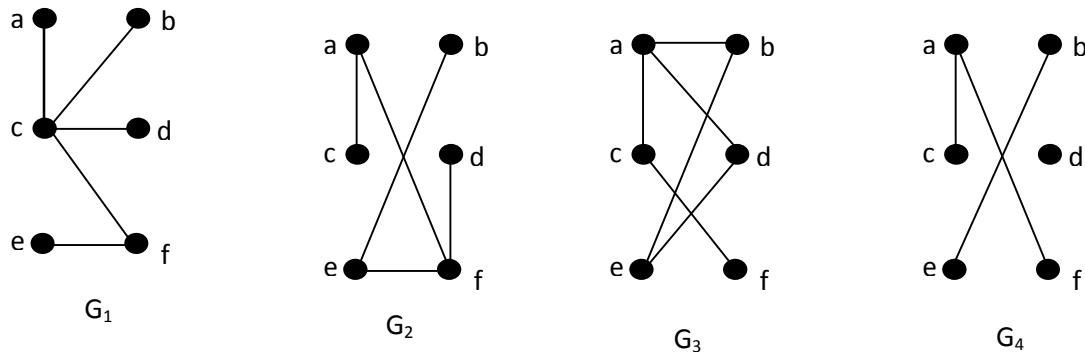
**Solution:**

Lets start with vertex 1, it has adjacent vertices as 2, 3, 4, 5, 6, 9 so using only 2 colors would suffice for the graph having the edges from 1 to its adjacent vertices. However since 9 has its adjacent vertices as 1, 2, 3, 4 we cannot just color the above graph with 2 colors because at least 1, 2 and 9 must have different colors. Trying with 3 colors we found that at least 1, 2, 3, and 9 must have different colors. So trying with four colors we can color the graph. Hence the chromatic number of the above graph is 4. Possible coloring is shown in the figure below.



## Trees

A tree is a connected undirected simple graph with no simple circuits. A tree is a particular type of graph. For example, family trees are graphs that represent genealogical charts. Family trees have vertices to represents the members of a family and edges to represents parent-child relationship. Trees are particularly useful in computer science in a wide range of algorithm including searching and sorting.



**Q. Which of the graphs shown below are trees?**

An undirected graph having no simple circuit and is not connected is called forest. The forest has each of its connected components as tree. For example, the figure  $G_4$  above displays a forest.

### Theorem 1:

An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

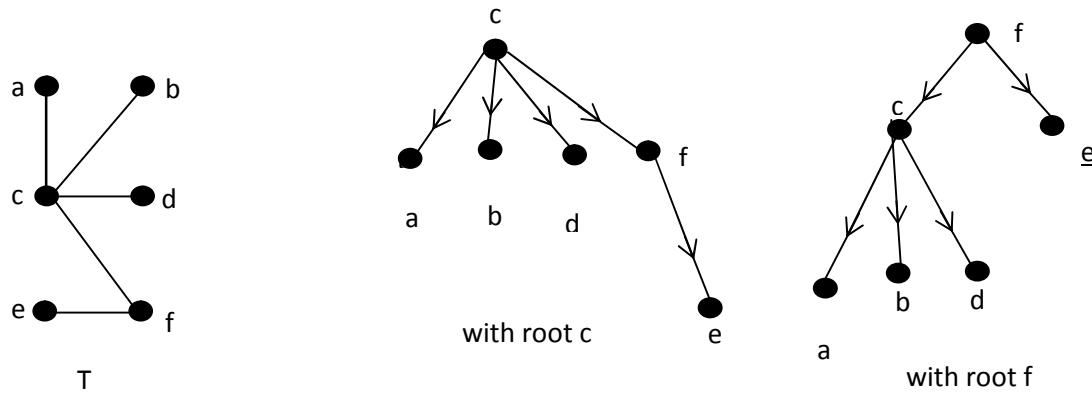
#### Proof:

Assume that  $T$  is a tree. Since  $T$  is a tree it is a connected simple graph with no simple circuits. Let  $x$  and  $y$  be two vertices of  $T$ . we know that every connected graph has a simple path between every pair of vertices. So there is a simple path from  $x$  to  $y$ . This path must be unique because, if the path between  $x$  and  $y$  is not unique then there is another path between  $x$  and  $y$  that uses edges different from the path between  $x$  and  $y$  for first path, then reversing the path i.e. going from  $x$  to  $y$  from the first path and going from  $y$  to  $x$  through the second path forms a circuit. This is a contradiction that  $T$  is a tree; hence there is a unique simple path between any two vertices of a tree.

Again assume that there is a unique simple path between any two vertices of a graph, say T. Since there is a path between any two vertices of a graph, the graph is connected. Now, we can show that the graph T cannot have simple circuit. Had there been a simple circuit, there would be two simple paths between two vertices, say x and y, and the two simple path between x and y would create a simple circuit where first path goes from x to y and the second path goes from y to x. This violates our assumption that the path is unique. Hence, a graph with a unique simple path between any two vertices is a tree.

In many applications of trees, a particular vertex of a tree is designated as the root. A rooted tree is a tree in which one vertex has been designated as the root and every edge is directed away from the root.

We can change an un-rooted tree in to a rooted by choosing any vertex as the root. The tree in which root is defined produces a directed graph as shown below:



Take a rooted tree  $T$ . if  $v$  is the vertex in  $T$  other than root, then the ***parent*** of  $v$  is a vertex  $u$  in  $T$  such that there is a directed edge from  $u$  to  $v$ . In this scenario  $v$  is called ***child*** of  $u$ . Vertices with same parents are called ***siblings***. All the vertices that appear in the path from root to some vertex  $v$  in  $T$ , including root are called ***ancestors*** of  $v$ . The ***descendents*** of a vertex  $v$  are those vertices that have  $v$  as their ancestor. All the vertices that have children are called ***internal vertices*** (root is also an internal vertex if the tree has more than one vertices).

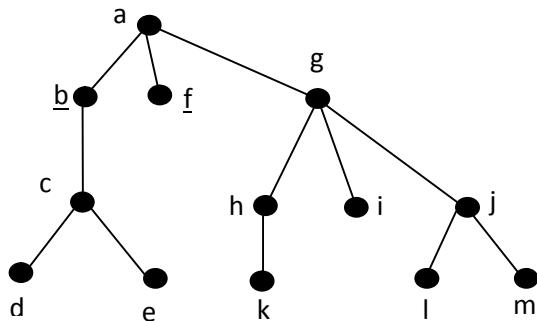
A **subtree** of a rooted tree  $T$ , with root  $a$ , is a **subgraph** of the tree consisting of  $a$  and all of its descendants and all the edges incident to these descendants. Here all the vertices must be in  $T$  also.

A  **$m$ -ary** tree is a rooted tree in which every internal vertex has no more than  $m$  children. It is called **full  $m$ -ary** tree if every internal vertex has exactly  $m$  children.

**Example:** Binary tree i.e. 2-ary tree.

An **ordered rooted** tree is a rooted tree where the children of each internal vertex are ordered. For e.g. in ordered binary tree (also called just a binary tree) if an internal vertex has two children then the first child is called **left child** and the second child is called **right child**.

Q. In the rooted tree given below, find the parent of  $c$ , the children of  $g$ , the sibling of  $h$ , all ancestors of  $e$ , all descendants of  $b$ , all internal vertices, and all leaves. What is the sub tree rooted at  $g$ ?



A rooted tree is called an  $m$ -ary tree if every internal vertex has no more than  $m$  children. The tree is called a full  $m$ -ary tree if every internal vertex has exactly  $m$  children. An  $m$ -ary tree with  $m = 2$  is called a binary tree.

An ordered rooted tree is a rooted tree where the children of each internal vertex are ordered from left to right.

In an ordered binary tree (usually called just a binary tree), if an internal vertex has two children the first child is called the left child and the second child is called the right child. The tree rooted at the left child of a vertex is called the left sub tree of this vertex, and the tree rooted at the right child of a vertex is called the right sub tree of the vertex.

## Properties of Trees

**Theorem:** A tree with  $n$  vertices has  $n-1$  edges.

**Proof:** Here, we use mathematical induction to prove this theorem.

**Basis step:** When  $n=1$ , a tree with  $n=1$  vertex has no edges. It follows that the theorem is true for  $n=1$ .

**Inductive hypothesis:** Assume that the tree with  $k$  vertices has  $k-1$  edges, where  $k$  is a positive integer.

**Inductive step:** Suppose that a tree  $T$  has  $k+1$  vertices and that  $v$  is a leaf of  $T$ . Removing the vertex  $v$  and the associated edge from  $T$  produces a tree  $T_1$  with  $k$  vertices, since the resulting graph is still connected and has no simple circuits. By the induction hypothesis,  $T_1$  has  $k-1$  edges. Hence,  $T$  has  $k$  edges since it has one more edge than  $T_1$ , the edge connecting  $v$  to its parent.

**Theorem:** A full  $m$ -ary tree with  $i$  internal vertices contain  $n = mi + 1$  vertices.

The level (depth) of a vertex  $v$  in a rooted tree is the length of the unique path from the root to vertex. The level of the root is zero. The height of the rooted tree is the length of the longest path from the root to any vertex. A rooted  $m$ -ary tree of height  $h$  is balanced if all leaves are at levels  $h$  or  $h - 1$ .

## Application of Trees

We can solve different problems using trees. For example,

- How should items in a list be stored so that an item can be easily located? To solve these problems, we use the concept of binary search trees.
- What series of decisions should be made to find an object with a certain property in a collection of objects of a certain type? To solve these problems, we use the concept of decision trees.
- How should a set of characters be efficiently coded by bit strings? To solve these problems, we use the concept of prefix codes.

## Application of Graphs

- Graphs are used to represent networks of communication, data organization, computational devices, flow of computation.
- Graphs can be used to show the link structures of web sites. The vertices are the web pages available at the website and a directed edge from page A to page B exists if and only if A contains a link to B.
- Graph theory is also used to study molecules in chemistry and physics.
- In chemistry a graph makes a natural model for a molecule, where vertices represent atoms and edges bonds. This approach is especially used in computer processing of molecular structures, ranging from chemical editors to database searching.

## Binary Search Trees

A binary search tree (BST) is a binary tree in which each child of a vertex is designated as a right or left child, no vertex has more than one right child or left child and each vertex is labeled with a key, which is one of the items. Furthermore vertices are assigned keys so that the key of a vertex is both larger than the keys of all vertices in its left subtree and smaller than the keys of all vertices in its right subtree.

We use a recursive procedure to form a binary search tree for a list of items. We start with a tree containing just one vertex, namely, the root. The first item in the list is assigned as the key of the root. To add a new item, first compare it with the keys of vertices already in the tree, starting at the root and moving to the left if the item is less than the key of the respective vertex if this vertex has a left child or moving to the right if the item is greater than the key of the respective vertex if this vertex has a right child. When the item is less than the respective vertex and this vertex has no left child, then a new vertex with this item as its key is inserted as a new left child. Similarly, when the item is greater than the respective vertex and this vertex has no right child, then a new vertex with this item as its key is inserted as a new right child.

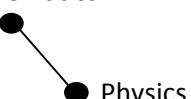
**Example:** Form a BST for the words mathematics, physics, geography, zoology, metrology, geology, psychology, and chemistry.

Mathematics



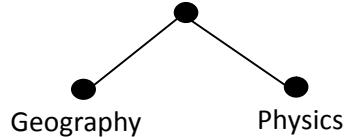
(a)

Mathematics



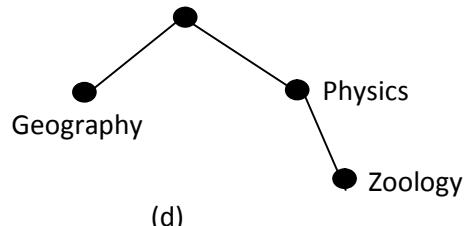
(b)

Mathematics



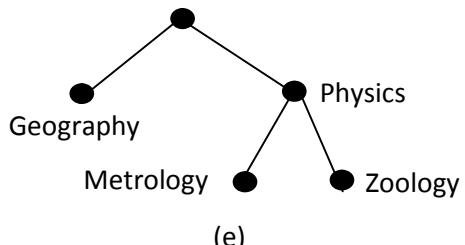
(c)

Mathematics



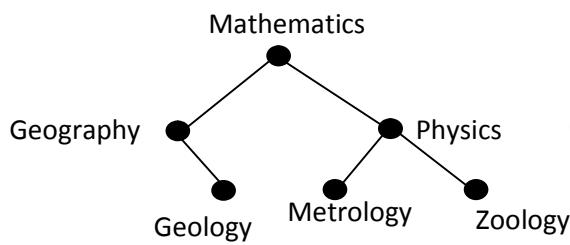
(d)

Mathematics

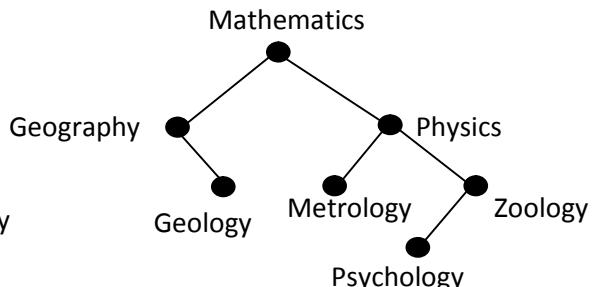


(e)

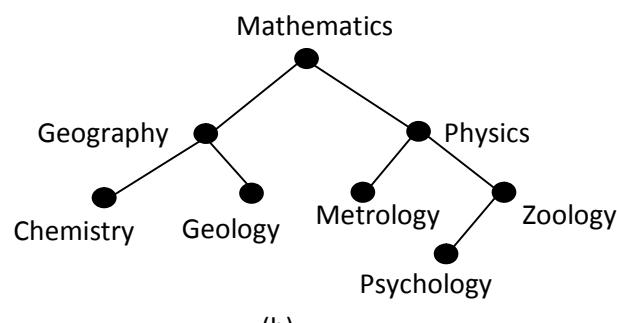
**Solution:**



(f)



(g)



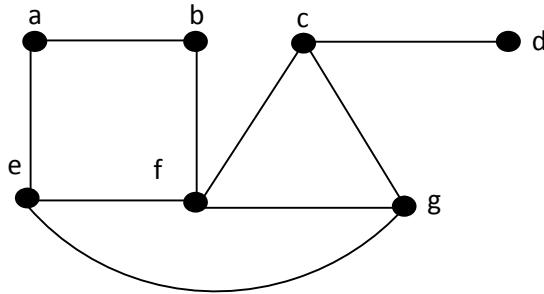
(h)

Q. Consider a BST whose elements are abbreviated names of chemical elements. Starting with an empty BST, show the effect of successively adding the following elements: H, C, N, O, Al, Si, Fe, Na, P, S, Ni, and Ca.

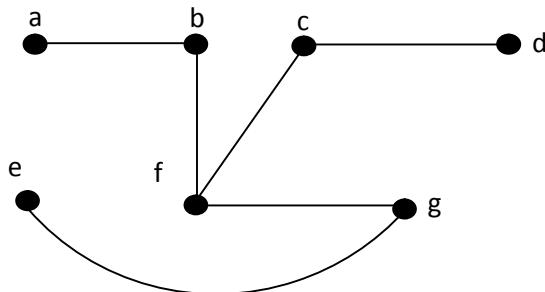
## Spanning Trees

Let  $G$  be a simple connected graph. A spanning tree of  $G$  is a subgraph of  $G$  that is a tree containing every vertex of  $G$ .

**Example:** Find a spanning tree of the simple graph  $G$  shown below.



**Solution:** The above graph is connected but is not a tree because it contains simple circuits. By removing the edges  $\{a, e\}$ ,  $\{e, f\}$  and  $\{c, g\}$ , we obtain a simple graph with no simple circuits. This subgraph is a spanning tree. The figure below shows this spanning tree. This tree is not the only spanning tree of the graph. The graph can also have other spanning trees.



**Theorem:** A simple graph is connected if and only if it has a spanning tree.

**Proof:** First, suppose that a simple graph  $G$  has a spanning tree  $T$ . Then,  $T$  contains every vertex of  $G$ . Furthermore, there is a path in  $T$  between any two of its vertices. Since  $T$  is a subgraph of  $G$ , there is a path in  $G$  between any two of its vertices. Hence,  $G$  is connected.

Now, suppose that  $G$  is connected. If  $G$  is not a tree, it must contain a simple circuit. Remove an edge from one of these simple circuits. The resulting subgraph has one fewer edge but still contains all the vertices of  $G$  and is connected. If this subgraph is not a tree, it has a simple

circuit; so as before, remove an edge that is in a simple circuit. Repeat this process until no simple circuits remain. A tree is produced since the graph stays connected as edges are removed. And, this tree is a spanning tree since it contains every vertex of G.

*Note: Spanning trees are important in data networking.*

## Minimum Spanning Trees

A minimum spanning tree in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

Kruskal's algorithm for constructing minimum spanning trees: To carry out Kruskal's algorithm, choose an edge in the graph with minimum weight. Successively add edges with minimum weight that do not form a simple circuit with those edges already chosen. Stop after n-1 edges have been selected.

### Algorithm:

*procedure Kruskal (G:weighted connected undirected graph with n vertices)*

*T := empty graph*

*for i := 1 to n - 1*

*begin*

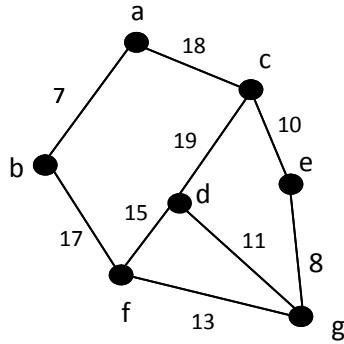
*e := any edge in graph G with smallest weight that edges not form a simple*

*circuit when added to T*

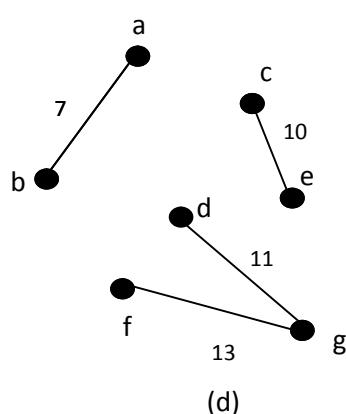
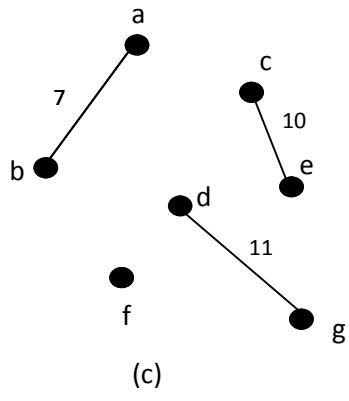
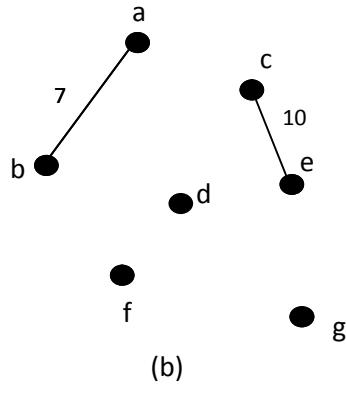
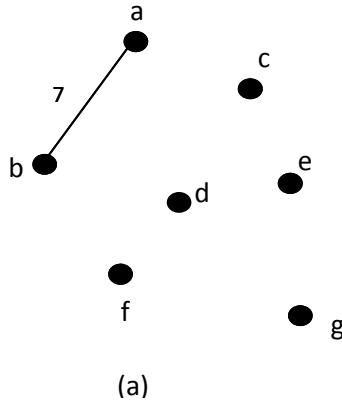
*T := T with e added*

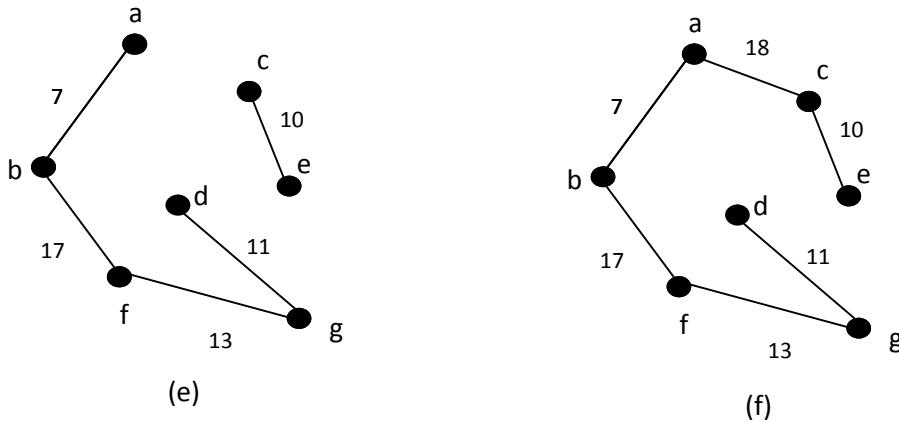
*end {T is a minimum spanning tree of G}*

Example: Find the minimum spanning tree of the following graph using Kruskal's algorithm.



**Solution:**





## Prim's Algorithm

**Working principle:** This is a greedy algorithm that chooses optimal solution at a particular instance. Choose an edge of the smallest edge, put it into the spanning tree. Successively add to the tree edges of minimum weight that are incident to the vertex already in the tree and not forming a simple circuit. Stop when  $n-1$  edges are added.

### Algorithm:

*Tree prim( $G$ : connected weighted undirected graph with  $n$  vertices)*

{

$T$  = a minimum weight edge.

for  $i = 1$  to  $n-2$

{

$e$  = an edge of minimum weight incident to a vertex in  $T$  and not forming a simple circuit in  $T$  if added to  $T$

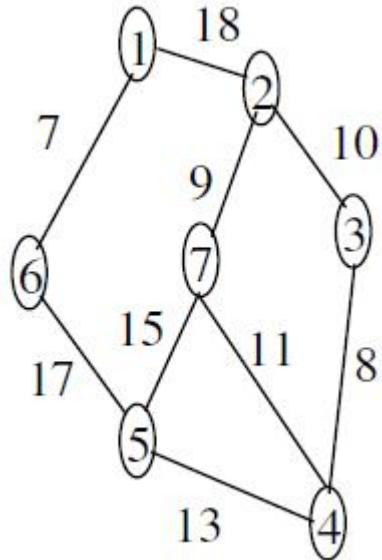
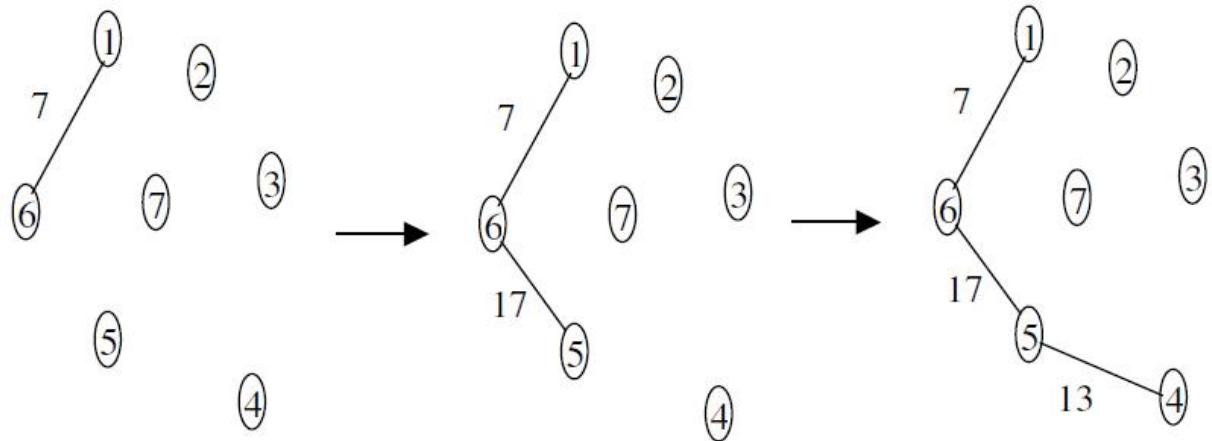
$T = T$  with  $e$  added

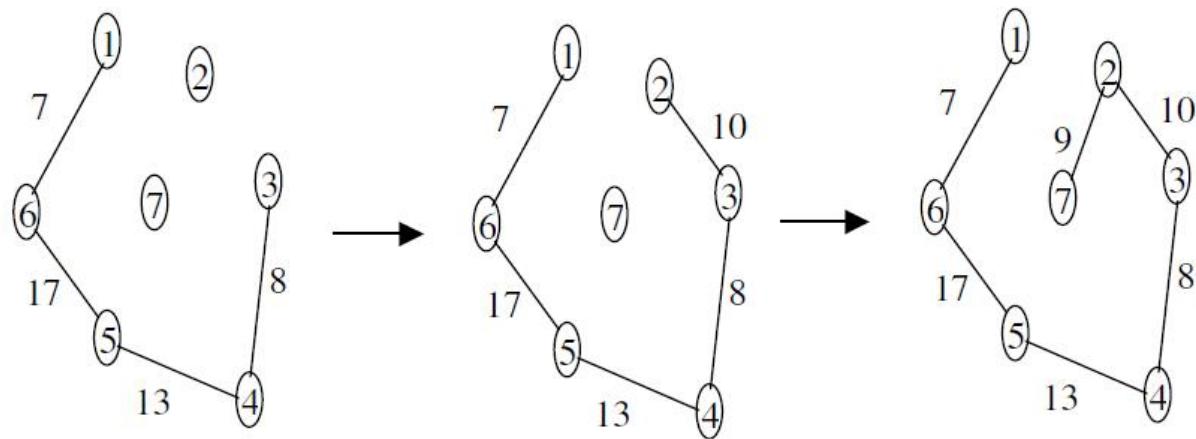
}

}

**Example:**

Find the minimum spanning tree of the following graph using Prim's algorithm.

**Solution**



## Bibliography

- Discrete Mathematics and its Applications by Kenneth H Rosen
- Notes from Samujjwal Bhandari of CDSCSIT, TU
- Notes from Nabaraj Poudel of Patan Multiple Campus, TU