

**LECTURE NOTES**  
**ON**  
**Data Structures and Problem Solving**  
**M.Tech I semester (IARE-R16)**

**Mr. RAJASEKHAR NENNURI**  
Assistant Professor



**COMPUTER SCIENCE AND ENGINEERING**  
**INSTITUTE OF AERONAUTICAL ENGINEERING**

DUNDIGAL, HYDERABAD - 500 043

# UNIT – I

## INTRODUCTION TO DATA STRUCTURES, SEARCHING AND SORTING

### Basic Concepts: Introduction to Data Structures:

A data structure is a way of storing data in a computer so that it can be used efficiently and it will allow the most efficient algorithm to be used. The choice of the data structure begins from the choice of an abstract data type (ADT). A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory space, as possible. Data structure introduction refers to a scheme for organizing data, or in other words it is an arrangement of data in computer's memory in such a way that it could make the data quickly available to the processor for required calculations.

A data structure should be seen as a logical concept that must address two fundamental concerns.

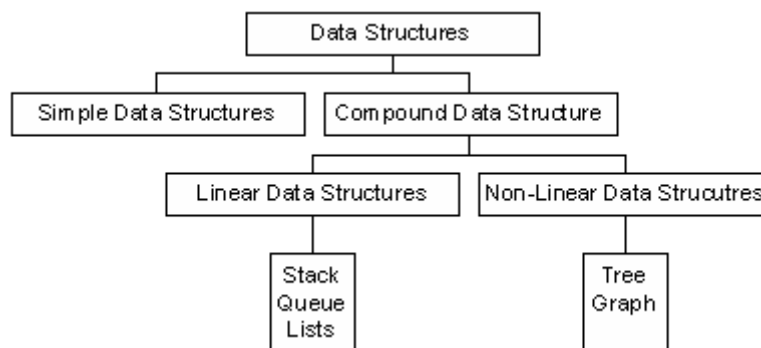
1. First, how the data will be stored, and
2. Second, what operations will be performed on it.

As data structure is a scheme for data organization so the functional definition of a data structure should be independent of its implementation. The functional definition of a data structure is known as ADT (Abstract Data Type) which is independent of implementation. The way in which the data is organized affects the performance of a program for different tasks. Computer programmers decide which data structures to use based on the nature of the data and the processes that need to be performed on that data. Some of the more commonly used data structures include lists, arrays, stacks, queues, heaps, trees, and graphs.

### Classification of Data Structures:

Data structures can be classified as

- Simple data structure
- Compound data structure
- Linear data structure
- Non linear data structure



[Fig 1.1 Classification of Data Structures]

**Simple Data Structure:**

Simple data structure can be constructed with the help of primitive data structure. A primitive data structure used to represent the standard data types of any one of the computer languages. Variables, arrays, pointers, structures, unions, etc. are examples of primitive data structures.

**Compound Data structure:**

Compound data structure can be constructed with the help of any one of the primitive data structure and it is having a specific functionality. It can be designed by user. It can be classified as

- Linear data structure
- Non-linear data structure

**Linear Data Structure:**

Linear data structures can be constructed as a continuous arrangement of data elements in the memory. It can be constructed by using array data type. In the linear Data Structures the relationship of adjacency is maintained between the data elements.

**Operations applied on linear data structure:**

The following list of operations applied on linear data structures

1. Add an element
  2. Delete an element
  3. Traverse
  4. Sort the list of elements
  5. Search for a data element
- For example Stack, Queue, Tables, List, and Linked Lists.

**Non-linear Data Structure:**

Non-linear data structure can be constructed as a collection of randomly distributed set of data item joined together by using a special pointer (tag). In non-linear Data structure the relationship of adjacency is not maintained between the data items.

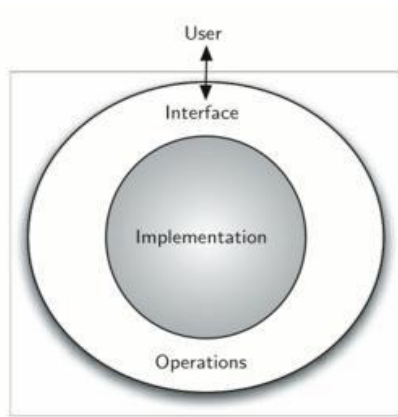
**Operations applied on non-linear data structures:**

The following list of operations applied on non-linear data structures.

1. Add elements
  2. Delete elements
  3. Display the elements
  4. Sort the list of elements
  5. Search for a data element
- For example Tree, Decision tree, Graph and Forest

**Abstract Data Type:**

An abstract data type, sometimes abbreviated ADT, is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what data is representing and not with how it will eventually be constructed. By providing this level of abstraction, we are creating an encapsulation around the data. The idea is that by encapsulating the details of the implementation, we are hiding them from the user's view. This is called information hiding. The implementation of an abstract data type, often referred to as a data structure, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types.



[Fig. 1.2: Abstract Data Type (ADT)]

## Algorithms:

### Structure and Properties of Algorithm:

An algorithm has the following structure

1. Input Step
2. Assignment Step
3. Decision Step
4. Repetitive Step
5. Output Step

An algorithm is endowed with the following properties:

1. **Finiteness:** An algorithm must terminate after a finite number of steps.
2. **Definiteness:** The steps of the algorithm must be precisely defined or unambiguously specified.
3. **Generality:** An algorithm must be generic enough to solve all problems of a particular class.
4. **Effectiveness:** the operations of the algorithm must be basic enough to be put down on pencil and paper. They should not be too complex to warrant writing another algorithm for the operation.
5. **Input-Output:** The algorithm must have certain initial and precise inputs, and outputs that may be generated both at its intermediate and final steps.

### Different Approaches to Design an Algorithm:

An algorithm does not enforce a language or mode for its expression but only demands adherence to its properties.

### Practical Algorithm Design Issues:

1. **To save time (Time Complexity):** A program that runs faster is a better program.
2. **To save space (Space Complexity):** A program that saves space over a competing program is

considerable desirable.

### Efficiency of Algorithms:

The performances of algorithms can be measured on the scales of **time** and **space**. The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical and the other is experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

**Time Complexity:** The time complexity of an algorithm or a program is a function of the running time of the algorithm or a program. In other words, it is the amount of computer time it needs to run to completion.

**Space Complexity:** The space complexity of an algorithm or program is a function of the space needed by the algorithm or program to run to completion.

The time complexity of an algorithm can be computed either by an **empirical** or **theoretical** approach. The **empirical** or **posteriori testing** approach calls for implementing the complete algorithms and executing them on a computer for various instances of the problem. The time taken by the execution of the programs for various instances of the problem are noted and compared. The algorithm whose implementation yields the least time is considered as the best among the candidate algorithmic solutions.

### Analyzing Algorithms

Suppose  $M$  is an algorithm, and suppose  $n$  is the size of the input data. Clearly the complexity  $f(n)$  of  $M$  increases as  $n$  increases. It is usually the rate of increase of  $f(n)$  with some standard functions. The most common computing times are

$O(1)$ ,  $O(\log_2 n)$ ,  $O(n)$ ,  $O(n \log_2 n)$ ,  $O(n^2)$ ,  $O(n^3)$ ,  $O(2^n)$

Example:

**Program Segment A**

```

-----
x = x + 2;
-----

```

**Program Segment B**

```

-----
for k = 1 to n do
    x = x + 2;
end;
-----

```

**Program Segment C**

```

-----
for j = 1 to n do
    for x = 1 to n do
        x = x + 2;
    end
end;
-----

```

**Total Frequency Count of Program Segment A**

Program Statements	Frequency Count
----- x = x + 2; -----	1
Total Frequency Count	1

**Total Frequency Count of Program Segment B**

Program Statements	Frequency Count
----- for k = 1 to n do x = x + 2; end; -----	(n+1) n n
Total Frequency Count	3n+1

**Total Frequency Count of Program Segment C**

Program Statements	Frequency Count
----- for j = 1 to n do for x = 1 to n do x = x + 2; end end; -----	(n+1) n(n+1) $n^2$ $n^2$ n
Total Frequency Count	$3n^2+3n+1$

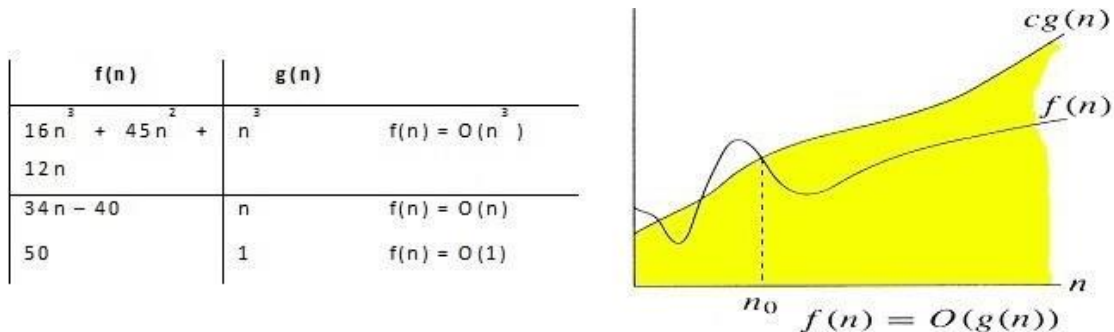
The total frequency counts of the program segments A, B and C given by 1,  $(3n+1)$  and  $(3n^2+3n+1)$  respectively are expressed as  $O(1)$ ,  $O(n)$  and  $O(n^2)$ . These are referred to as the time complexities of the program segments since they are indicative of the running times of the program segments. In a similar manner space complexities of a program can also be expressed in terms of mathematical notations,

which is nothing but the amount of memory they require for their execution.

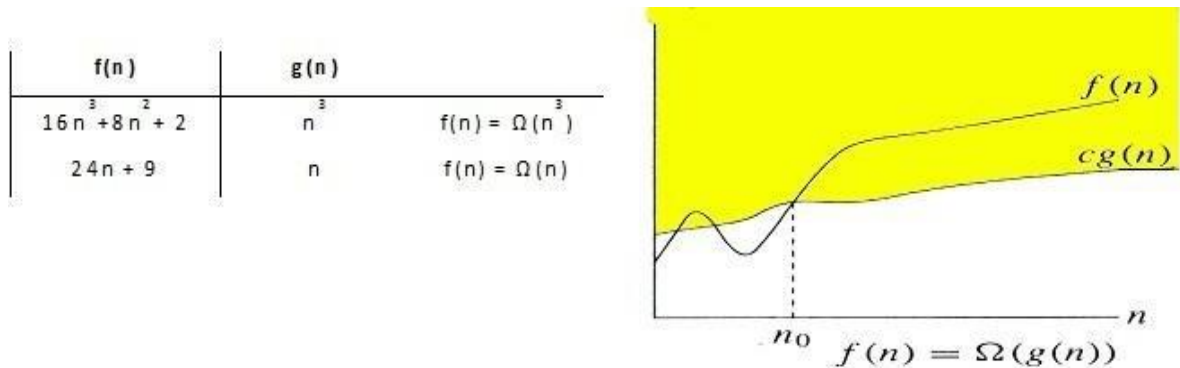
### Asymptotic Notations:

It is often used to describe how the size of the input data affects an algorithm's usage of computational resources. Running time of an algorithm is described as a function of input size  $n$  for large  $n$ .

**Big oh(O):** Definition:  $f(n) = O(g(n))$  (read as  $f$  of  $n$  is big oh of  $g$  of  $n$ ) if there exist a positive integer  $n_0$  and a positive number  $c$  such that  $|f(n)| \leq c|g(n)|$  for all  $n \geq n_0$ . Here  $g(n)$  is the upper bound of the function  $f(n)$ .

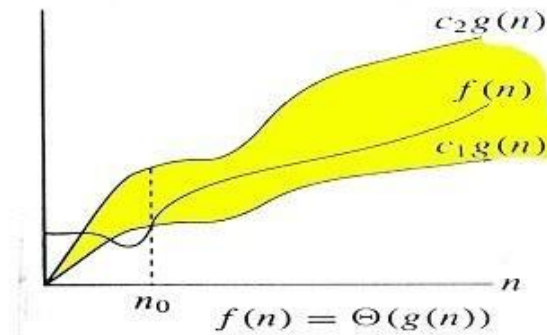


**Omega( $\Omega$ ):** Definition:  $f(n) = \Omega(g(n))$  (read as  $f$  of  $n$  is omega of  $g$  of  $n$ ), if there exists a positive integer  $n_0$  and a positive number  $c$  such that  $|f(n)| \geq c|g(n)|$  for all  $n \geq n_0$ . Here  $g(n)$  is the lower bound of the function  $f(n)$ .



**Theta( $\Theta$ ):** Definition:  $f(n) = \Theta(g(n))$  (read as  $f$  of  $n$  is theta of  $g$  of  $n$ ), if there exists a positive integer  $n_0$  and two positive constants  $c_1$  and  $c_2$  such that  $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$  for all  $n \geq n_0$ . The function  $g(n)$  is both an upper bound and a lower bound for the function  $f(n)$  for all values of  $n$ ,  $n \geq n_0$ .

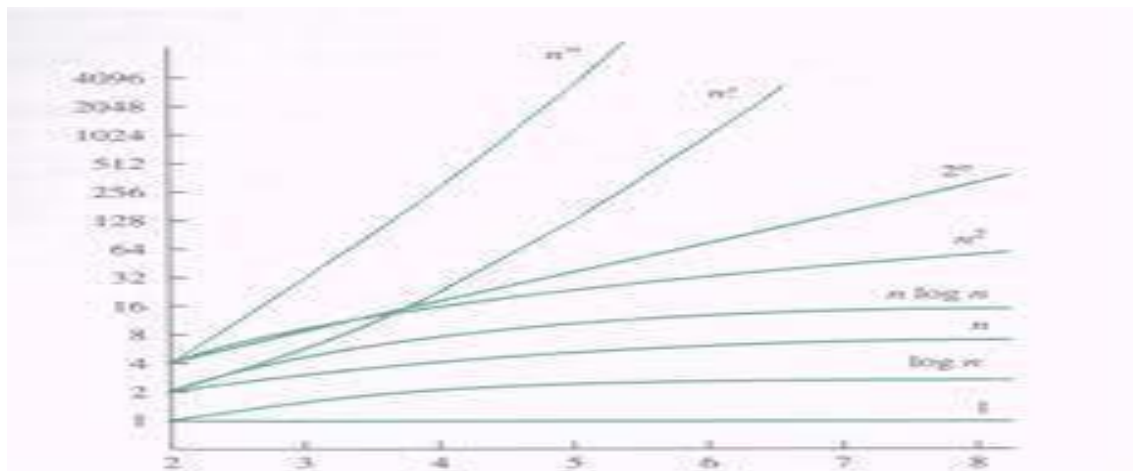
$f(n)$	$g(n)$	
$16n^3 + 30n^2 - 90$	$n^2$	$f(n) = \Theta(n^2)$
$7 \cdot 2^n + 30n$	$2^n$	$f(n) = \Theta(2^n)$



**Little oh(o):** Definition:  $f(n) = O(g(n))$  ( read as f of n is little oh of g of n), if  $f(n) = O(g(n))$  and  $f(n) \neq \Omega(g(n))$ .

**Time Complexity:**

**Time Complexities of various Algorithms:**



**Numerical Comparison of Different Algorithms:**

S.No.	$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$n^3$	$2^n$
1.	0	1	1	1	1	2
2.	1	2	2	4	8	4
3.	2	4	8	16	64	16
4.	3	8	24	64	512	256
5.	4	16	64	256	4096	65536

**Reasons for analyzing algorithms:**

1. To predict the resources that the algorithm requires



- Computational Time(CPU consumption).
  - Memory Space(RAM consumption).
  - Communication bandwidth consumption.
2. To predict the running time of an algorithm
- Total number of primitive operations executed.

### Recursive Algorithms:

**GCD Design:** Given two integers a and b, the greatest common divisor is recursively found using the formula

$$\text{gcd}(a,b) = \begin{cases} a & \text{if } b=0 \\ b & \text{if } a=0 \\ \text{gcd}(b, a \bmod b) & \end{cases}$$

Base case

General case

**Fibonacci Design:** To start a fibonacci series, we need to know the first two numbers.

$$\text{Fibonacci}(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ \text{Fibonacci}(n-1) + \text{fibonacci}(n-2) & \end{cases}$$

Base case

General case

### Difference between Recursion and Iteration:

1. A function is said to be recursive if it calls itself again and again within its body whereas iterative functions are loop based imperative functions.
2. Recursion uses stack whereas iteration does not use stack.
3. Recursion uses more memory than iteration as its concept is based on stacks.
4. Recursion is comparatively slower than iteration due to overhead condition of maintaining stacks.
5. Recursion makes code smaller and iteration makes code longer.
6. Iteration terminates when the loop-continuation condition fails whereas recursion terminates when a base case is recognized.
7. While using recursion multiple activation records are created on stack for each call whereas in iteration everything is done in one activation record.
8. Infinite recursion can crash the system whereas infinite looping uses CPU cycles repeatedly.
9. Recursion uses selection structure whereas iteration uses repetition structure.

### Types of Recursion:

Recursion is of two types depending on whether a function calls itself from within itself or whether two functions call one another mutually. The former is called **direct recursion** and the latter is called

**indirect recursion.** Thus there are two types of recursion:

- Direct Recursion
- Indirect Recursion

Recursion may be further categorized as:

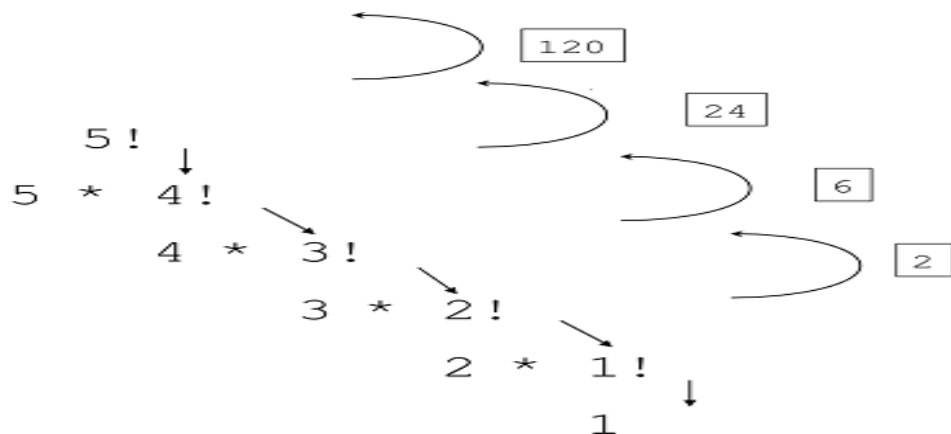
- Linear Recursion
- Binary Recursion
- Multiple Recursion

### Linear Recursion:

It is the most common type of Recursion in which function calls itself repeatedly until base condition [termination case] is reached. Once the base case is reached the results are return to the caller function. If a recursive function is called only once then it is called a linear recursion.

Example1: Finding the factorial of a number.

```
fact(int f)
{
    if (f == 1) return 1;
    return (f * fact(f - 1)); //called in function only once
}
int main()
{
    int fact;
    fact = fact(5);
    printf("Factorial is %d", fact);
    return 0;
}
```



## Binary Recursion:

Some recursive functions don't just have one call to themselves; they have two (or more). Functions with two recursive calls are referred to as binary recursive functions.

Example1: The Fibonacci function `fib` provides a classic example of binary recursion. The Fibonacci numbers can be defined by the rule:

$$\begin{aligned}\text{fib}(n) &= 0 \text{ if } n \text{ is } 0, \\ &= 1 \text{ if } n \text{ is } 1, \\ &= \text{fib}(n-1) + \text{fib}(n-2) \text{ otherwise}\end{aligned}$$

For example, the first seven Fibonacci numbers are

$$\text{Fib}(0) = 0$$

$$\text{Fib}(1) = 1$$

$$\text{Fib}(2) = \text{Fib}(1) + \text{Fib}(0) = 1$$

$$\text{Fib}(3) = \text{Fib}(2) + \text{Fib}(1) = 2$$

$$\text{Fib}(4) = \text{Fib}(3) + \text{Fib}(2) = 3$$

$$\text{Fib}(5) = \text{Fib}(4) + \text{Fib}(3) = 5$$

$$\text{Fib}(6) = \text{Fib}(5) + \text{Fib}(4) = 8$$

This leads to the following implementation in C:

```
int fib(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return fib(n - 1) + fib(n - 2);
}
```



```

{
    int i, total=1;

    for(i=0; i<b; i++) total *= a;

    return total;
}

```

### **Recursive algorithms for Factorial, GCD, Fibonacci Series and Towers of Hanoi:**

#### **Factorial(n)**

Input: integer  $n \geq 0$

Output:  $n!$

1. If  $n = 0$  then return (1)
2. else return  $\text{prod}(n, \text{factorial}(n - 1))$

#### **GCD(m, n)**

Input: integers  $m > 0, n \geq 0$

Output:  $\text{gcd}(m, n)$

1. If  $n = 0$  then return (m)
2. else return  $\text{gcd}(n, m \bmod n)$

Time-Complexity:  $O(\ln n)$

#### **Fibonacci(n)**

Input: integer  $n \geq 0$

Output: Fibonacci Series: 1 1 2 3 5 8 13.....

1. if  $n=1$  or  $n=2$
2. then  $\text{Fibonacci}(n)=1$
3. else  $\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$

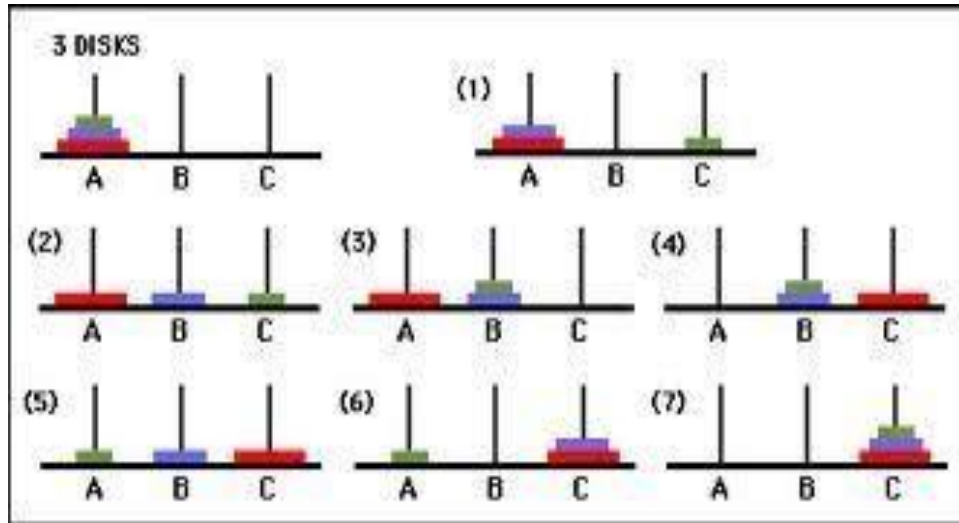
#### **Towers of Hanoi**

Input: The aim of the tower of Hanoi problem is to move the initial  $n$  different sized disks from needle A to needle C using a temporary needle B. The rule is that no larger disk is to be placed above the smaller disk in any of the needle while moving or at any time, and only the top of the disk is to be moved at a time from any needle to any needle.

Output:

1. If  $n=1$ , move the single disk from A to C and return,

2. If  $n > 1$ , move the top  $n-1$  disks from A to B using C as temporary.
3. Move the remaining disk from A to C.
4. Move the  $n-1$  disk disks from B to C, using A as temporary.



### Searching Techniques:

#### Linear Search:

1. Read search element.
2. Call function linear search function by passing N value, array and search element.
3. If  $a[i] == k$ , return  $i$  value, else return -1, returned value is stored in pos.
4. If  $pos == -1$  print element not found, else print  $pos+1$  value.

**Source Code:** (Recursive)

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

```
#include<conio.h>
```

```
void linear_search(int n,int a[20],int i,int k)
```

```
{
```

```
    if(i>=n)
```

```
    {
```

```
        printf("%d is not found",k);
```

```

        return;
    }
    if(a[i]==k)
    {
        printf("%d is found at %d",k,i+1);
        return;
    }
    else
        linear_search(n,a,i+1,k);
}

void main()
{
    int i,a[20],n,k;
    clrscr();
    printf("Enter no of elements:");
    scanf("%d",&n);
    printf("Enter elements:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("Enter search element:");
    scanf("%d",&k);
    linear_search(n,a,0,k);
    getch();
}

```

### **Input & Output:**

Enter no of elements:3

Enter elements:1 2 3

Enter search element:6

6 is not found

Enter no of elements:5

Enter elements:1 2 3 4 5

Enter search element:3

3 is found at position 3

### **Time Complexity of Linear Search:**

If input array is not sorted, then the solution is to use a sequential search.

Unsuccessful search:  $O(N)$

Successful Search: Worst case:  $O(N)$

Average case:  $O(N/2)$

### **Binary Search:**

1. Read search data.
2. Call `binary_search` function with values `N`, `array`, and `data`.
3. If `low` is less than `high`, making `mid` value as mean of `low` and `high`.
4. If `a[mid] == data`, make `flag=1` and break, else if `data` is less than `a[mid]` make `high=mid-1`, else `low=mid+1`.
5. If `flag == 1`, print data found at `mid+1`, else not found.

### **Source Code: (Recursive)**

```
#include<stdio.h>

#include<conio.h>

void binary_search(int a[20],int data,int low,int high)
{
    int mid;
    if(low<=high)
    {
        mid=(low+high)/2;
        if(a[mid]==data)
            printf("Data found at %d",mid+1);
        else
```



```

        if(a[mid]>data)

            binary_search(a,data,low,mid-1);

        else

            binary_search(a,data,mid+1,high);

    }

}

void main()

{

    int i,a[20],n,data;

    clrscr();

    printf("Enter no of elements:");

    scanf("%d",&n);

    printf("Enter elements:");

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

        scanf("%d",&a[i]);

    printf("Enter search element:");

    scanf("%d",&data);

    binary_search(a,data,0,n-1);

    getch();

}

```

### **Input & Output:**

Enter no of elements:3

Enter elements:1 2 3

Enter search element:25

Not found

Enter no of elements:3

Enter elements:1 2 3

Enter search element:3

Data found at 3

### **Time Complexity of Binary Search:**

Time Complexity for binary search is  $O(\log_2 N)$

### **Fibonacci Search:**

**Source Code:** (Recursive)

```
#include<stdio.h>

#include<conio.h>

void fib_search(int a[],int n,int search,int pos,int begin,int end)
{
    int fib[20]={0,1,1,2,3,5,8,13,21,34,55,89,144};
    if(end<=0)
    {
        printf("\nNot found");
        return;//data not found
    }
    else
    {
        pos=begin+fib[--end];
        if(a[pos]==search && pos<n)
        {
            printf("\n Found at %d",pos);
            return;//data found
        }
        if((pos>=n)|| (search<a[pos]))
            fib_search(a,n,search,pos,begin,end);
        else
        {
```

```

        begin=pos+1;

        end--;

        fib_search(a,n,search,pos,begin,end);

    }

}

void main()

{

    int n,i,a[20],search,pos=0,begin=0,k=0,end;

    int fib[20]={0,1,1,2,3,5,8,13,21,34,55,89,144};

    clrscr();

    printf("Enter the n:");

    scanf("%d",&n);

    printf("Enter elements to array:");

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

        scanf("%d",&a[i]);

    printf("Enter the search element:");

    scanf("%d",&search);

    while(fib[k]<n)

    {

        k++;

    }

    end=k;

    printf("Max.no of passes : %d",end);

    fib_search(a,n,search,pos,begin,end);

    getch();

}

```

**Input & Output:**

Enter the n:5

Enter elements to array:1 2 3 6 59

Enter the search element:56

Max no of passes required is : 5

Search element not found.....

### **Time Complexity of Fibonacci Search:**

Time complexity for Fibonacci search is  $O(\log_2 N)$

### **Sorting Techniques:**

Sorting in general refers to various methods of arranging or ordering things based on criterias (numerical, chronological, alphabetical, hierarchical etc.). There are many approaches to sorting data and each has its own merits and demerits.

### **Bubble Sort:**

Bubble Sort is probably one of the oldest, easiest, straight-forward, inefficient sorting algorithms. Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort. Although the algorithm is simple, most of the other sorting algorithms are more efficient for large lists. Bubble sort is not a stable sort which means that if two same elements are there in the list, they may not get their same order with respect to each other.

### **Bubble Sort Algorithm:**

Step 1: Repeat Steps 2 and 3 for  $i=1$  to 10

Step 2: Set  $j=1$

Step 3: Repeat while  $j \leq n$

(A) if  $a[i] < a[j]$

Then interchange  $a[i]$  and  $a[j]$

[End of if]

(B) Set  $j = j+1$

[End of Inner Loop]

[End of Step 1 Outer Loop]

Step 4: Exit

### **Source Code:**

```

#include<stdio.h>

#include<conio.h>

void main()

{

    int i,n,temp,j,arr[25];

    printf("Enter the number of elements in the Array: ");

    scanf("%d",&n);

    printf("\nEnter the elements:\n\n");

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

    {

        printf(" Array[%d] = ",i);

        scanf("%d",&arr[i]);

    }

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

    {

        for(j=0 ; j<n-i-1 ; j++)

        {

            if(arr[j]>arr[j+1]) //Swapping Condition is Checked

            {

                temp=arr[j];

                arr[j]=arr[j+1];

                arr[j+1]=temp;

            }

        }

    }

    printf("Array Elements after sorting: ");

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

    {

```

```

        printf("%5d",arr[i]);

    }

}

```

### output:

```
/*
```

Enter the number of elements in the Array: 10

Enter the elements:

Array[0] = 23

Array[1] = 65

Array[2] = 78

Array[3] = 22

Array[4] = 12

Array[5] = 09

Array[6] = 69

Array[7] = 34

Array[8] = 21

Array[9] = 56

Array Elements after sorting: 9 12 21 22 23 34 56 65 69 78

```
*/
```

### Step-by-step example:

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.

#### First Pass:

( **5** **1** 4 2 8 )      ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since  $5 > 1$ .

( 1 **5** **4** 2 8 )      ( 1 **4** **5** 2 8 ), Swap since  $5 > 4$

( 1 4 **5** **2** 8 )      ( 1 4 **2** **5** 8 ), Swap since  $5 > 2$

( 1 4 2 **5** **8** )

( 1 4 2 **5** **8** ), Now, since these elements are already in order ( $8 > 5$ ), algorithm does not swap them.

### Second Pass:

( 1 4 2 5 8 )      ( 1 4 2 5 8 )

( 1 4 2 5 8 )      ( 1 2 4 5 8 ), Swap since  $4 > 2$

( 1 2 4 5 8 )      ( 1 2 4 5 8 )

( 1 2 4 5 8 )      ( 1 2 4 5 8 )

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

### Third Pass:

( 1 2 4 5 8 )      ( 1 2 4 5 8 )

( 1 2 4 5 8 )      ( 1 2 4 5 8 )

( 1 2 4 5 8 )      ( 1 2 4 5 8 )

( 1 2 4 5 8 )      ( 1 2 4 5 8 )

### Time Complexity:

Worst Case Performance	$O(N^2)$
Best Case Performance	$O(N^2)$
Average Case Performance	$O(N^2)$

### Selection Sort:

The algorithm divides the input list into two parts: the sub-list of items already sorted, which is built up from left to right at the front (left) of the list, and the sub-list of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sub-list is empty and the unsorted sub-list is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sub-list, exchanging it with the leftmost unsorted element (putting it in sorted order), and moving the sub-list boundaries one element to the right.

Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has  $O(n^2)$  time complexity, making it inefficient on large lists.

### Selection Sort Algorithm and Pseudo Code:

```
/* a[0] to a[n-1] is the array to sort */
```

```
int i,j;
```

```
int iMin;
```

```
/* advance the position through the entire array */
```

```
/* (could do  $j < n-1$  because single element is also min element) */
```

```

for (j = 0; j < n-1; j++) {
    /* find the min element in the unsorted a[j .. n-1] */
    /* assume the min is the first element */
    iMin = j;
    /* test against elements after j to find the smallest */
    for ( i = j+1; i < n; i++) {
        /* if this element is less, then it is the new minimum */
        if (a[i] < a[iMin]) {
            /* found new minimum; remember its index */
            iMin = i;
        }
    }
    /* iMin is the index of the minimum element. Swap it with the current position */
    if ( iMin != j ) {
        swap(a[j], a[iMin]);
    }
}

```

#### **Source Code:**

```

# include <stdio.h>
# include <conio.h>

void main()
{
    int i,j,a[20],n,minindex,temp;
    clrscr();
    printf("\n Enter array size:");
    scanf("%d",&n);
    printf("\n Enter the elements:");
    for(i=0;i<n;i++)

```



```

        scanf("%d",&a[i]);

printf("\n The elements after sorting are:");

for(i=0;i<n;i++)
{
    minindex=i;
    for(j=i+1;j<n;j++)
    {
        if(a[j]<a[minindex])
            minindex=j;
    }
    temp=a[i];
    a[i]=a[minindex];
    a[minindex]=temp;
    printf("%5d",a[i]);
}

getch();
}

```

### **Output:**

Enter array size:6

Enter the elements:96 94 81 56 76 45

The elements after sorting are: 45 56 76 81 94 96

### **Step-by-step example:**

Here is an example of this sort algorithm sorting five elements:

64 25 12 22 11

11 **25** 12 22 64

11 **12** **25** 22 64

11 12 **22** **25** 64

**11** **12** **22** **25** **64**

### Time Complexity:

Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array. Selecting the lowest element requires scanning all  $n$  elements (this takes  $n - 1$  comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining  $n - 1$  elements and so on, for  $(n - 1) + (n - 2) + \dots + 2 + 1 = n(n - 1) / 2 \in O(n^2)$  comparisons. Each of these scans requires one swap for  $n - 1$  elements (the final element is already in place).

Worst Case Performance	$O(N^2)$
Best Case Performance	$O(N^2)$
Average Case Performance	$O(N^2)$

### Insertion Sort:

An algorithm consider the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted). Insertion sort is an example of an **incremental** algorithm. It builds the sorted sequence one number at a time. This is a suitable sorting technique in playing card games. Insertion sort provides several advantages:

- Simple implementation
- Efficient for (quite) small data sets
- Adaptive (i.e., efficient) for data sets that are already substantially sorted: the time complexity is  $O(n + d)$ , where  $d$  is the number of inversions
- More efficient in practice than most other simple quadratic (i.e.,  $O(n^2)$ ) algorithms such as selection sort or bubble sort; the best case (nearly sorted input) is  $O(n)$
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount  $O(1)$  of additional memory space
- Online; i.e., can sort a list as it receives it

### Source Code:

```
#include<stdio.h>

#include<conio.h>

void insertion_sort(int a[],int n)
{
    int temp,pos,i;
    for(i=0;i<n;i++)
    {
        temp=a[i];
```

```

        pos=i;
        while(pos>0&& a[pos-1]>temp)
        {
            a[pos]=a[pos-1];
            --pos;
        }
        a[pos]=temp;
    }
    printf("\nElements after sorting....\n");
    for(i=0;i<n;i++)
        printf("%5d",a[i]);
}

void main()
{
    int i,n,a[20];
    clrscr();
    printf("\nEnter the n:");
    scanf("%d",&n);
    printf("\nEnter the elements:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("\nElements before sorting:\n");
    for(i=0;i<n;i++)
        printf("%5d",a[i]);
    insertion_sort(a,n);
    getch();
}

```

**Output:**

Enter the n:5

Enter the elements: 1 3 6 25 0

Elements before sorting: 1 3 6 25 0

Elements after sorting: 0 1 3 6 25

**Step-by-step example:**


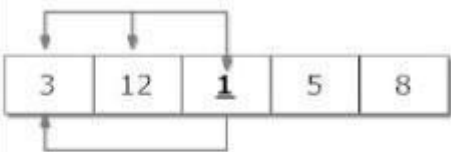
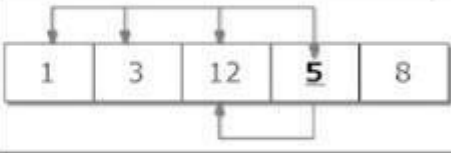
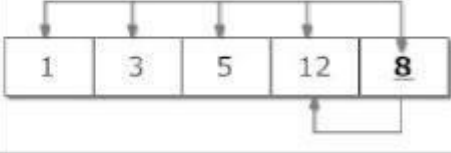

<b>Step 1</b>		Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.
<b>Step 2</b>		Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3.
<b>Step 3</b>		Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.
<b>Step 4</b>		Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.
		Sorted Array in Ascending Order

Figure: Sorting Array in Ascending Order Using Insertion Sort Algorithm

Suppose, you want to sort elements in ascending as in above figure. Then,

1. The second element of an array is compared with the elements that appear before it (only first element in this case). If the second element is smaller than first element, second element is inserted in the position of first element. After first step, first two elements of an array will be sorted.
2. The third element of an array is compared with the elements that appears before it (first and second element). If third element is smaller than first element, it is inserted in the position of first element. If third element is larger than first element but, smaller than second element, it is inserted in the position of second element. If third element is larger than both the elements, it is kept in the position as it is. After second step, first three elements of an array will be sorted.
3. Similarly, the fourth element of an array is compared with the elements that appear before it (first, second and third element) and the same procedure is applied and that element is inserted in the

proper position. After third step, first four elements of an array will be sorted.

If there are  $n$  elements to be sorted. Then, this procedure is repeated  $n-1$  times to get sorted list of array.

**Time Complexity:**

Worst Case Performance	$O(N^2)$
Best Case Performance(nearly)	$O(N)$
Average Case Performance	$O(N^2)$

**Output:**

Enter no of elements:5

Enter elements:1 65 0 32 66

Elements after sorting: 0 1 32 65 66

**Quick Sort :**

Quick sort is a divide and conquer algorithm. Quick sort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quick sort can then recursively sort the sub-lists.

The steps are:

1. Pick an element, called a **pivot**, from the list.
2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

The base case of the recursion is lists of size zero or one, which never need to be sorted.

**Quick sort**, or **partition-exchange sort**, is a sorting algorithm developed by Tony Hoare that, on average, makes  $O(n \log n)$  comparisons to sort  $n$  items. In the worst case, it makes  $O(n^2)$  comparisons, though this behavior is rare. Quick sort is often faster in practice than other  $O(n \log n)$  algorithms. It works by first of all by partitioning the array around a pivot value and then dealing with the 2 smaller partitions separately. Partitioning is the most complex part of quick sort. The simplest thing is to use the first value in the array,  $a[l]$  (or  $a[0]$  as  $l = 0$  to begin with) as the pivot. After the partitioning, all values to the left of the pivot are  $\leq$  pivot and all values to the right are  $>$  pivot. The same procedure for the two remaining sub lists is repeated and so on recursively until we have the entire list sorted.

**Advantages:**

- One of the fastest algorithms on average.
- Does not need additional memory (the sorting takes place in the array - this is called **in-place** processing).

**Disadvantages:** The worst-case complexity is  $O(N^2)$

**Source Code:**

```
#include<stdio.h>

#include<conio.h>

void quick_sort(int,int);
int partition(int,int);
void interchange(int,int);
int a[25];
void main()
{
    int i,n;
    clrscr();
    printf("Enter no of elements:");
    scanf("%d",&n);
    printf("Enter elements:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    quick_sort(0,n);
    printf("Elements after sorting:");
    for(i=0;i<n;i++)
        printf("%3d",a[i]);
    getch();
}

int partition(int low,int high)
{
    int pivot=a[low];
    int up=low,down=high;
    do
    {
```

```

        do
        {
            up=up+1;
        }while(a[up]<pivot);
        do
        {
            down=down-1;
        }while(a[down]>pivot);
        if(up<down)
            interchange(up,down);
    }while(up<down);
    a[low]=a[down];
    a[down]=pivot;
    return down;
}

void quick_sort(int low,int high)
{
    int pivotpos;
    if(low<high)
    {
        pivotpos=partition(low,high);
        quick_sort(low,pivotpos-1);
        quick_sort(pivotpos+1,high);
    }
}

void interchange(int i,int j)
{

```

```

    int temp;

    temp=a[i];

    a[i]=a[j];

    a[j]=temp;

}

```

### Output:

Enter no of elements:5

Enter elements:1 65 0 32 66

Elements after sorting: 0 1 32 65 66

### Step-by-step example:

1	2	3	4	5	6	7	8	9	10	11	12	13	Remarks
38	08	16	06	79	57	24	56	02	58	04	70	45	
Pivot	08	16	06	Up	57	24	56	02	58	Dn	70	45	Swap up and down
Pivot	08	16	06	04	57	24	56	02	58	79	70	45	
Pivot	08	16	06	04	Up	24	56	Dn	58	79	70	45	Swap up and down
Pivot	08	16	06	04	02	24	56	57	58	79	70	45	
Pivot	08	16	06	04	02	Dn	Up	57	58	79	70	45	Swap pivot and down
24	08	16	06	04	02	38	56	57	58	79	70	45	
Pivot	08	16	06	04	Dn	Up	56	57	58	79	70	45	Swap pivot and down
(02	08	16	06	04	24)	38	(56	57	58	79	70	45)	
Pivot	08	16	06	04	Up								
dn													
	Pivot	Up	06	Dn									Swap up and down
	Pivot	04	06	16									
	Pivot	04	Dn	Up									Swap pivot



													and down
	06	04	08										
	Pivot	Dn	Up										Swap pivot and down
	04	06											
(02	04	06	08	16	24	38)	(56	57	58	79	70	45)	
							Pivot	Up	58	79	70	Dn	Swap up and down
							Pivot	45	58	79	70	57	
							Pivot	Dn	Up	79	70	57	Swap pivot and down
							(45)	56	(58	79	70	57)	
									Pivot	Up	70	Dn	Swap up and down
									Pivot	57	70	79	
									Pivot	Dn	Up	79	Swap down and pivot
									(57)	58	(70	79)	
											Pivot	Up	Swap pivot and down
											Dn		
02	04	06	08	16	24	38	45	56	57	58	70	79	The array is sorted

### Time Complexity:

Worst Case Performance	$O(N^2)$
Best Case Performance(nearly)	$O(N \log_2 N)$
Average Case Performance	$O(N \log_2 N)$

## Merge Sort:

Merge sort is based on Divide and conquer method. It takes the list to be sorted and divide it in half to create two unsorted lists. The two unsorted lists are then sorted and merged to get a sorted list. The two unsorted lists are sorted by continually calling the merge-sort algorithm; we eventually get a list of size 1 which is already sorted. The two lists of size 1 are then merged.

### **Merge Sort Procedure:**

This is a divide and conquer algorithm.

This works as follows :

1. Divide the input which we have to sort into two parts in the middle. Call it the left part and right part.
2. Sort each of them separately. Note that here sort does not mean to sort it using some other method. We use the same function recursively.
3. Then merge the two sorted parts.

Input the total number of elements that are there in an array (number\_of\_elements). Input the array (array[number\_of\_elements]). Then call the function MergeSort() to sort the input array. MergeSort() function sorts the array in the range [left,right] i.e. from index left to index right inclusive. Merge() function merges the two sorted parts. Sorted parts will be from [left, mid] and [mid+1, right]. After merging output the sorted array.

### **MergeSort() function:**

It takes the array, left-most and right-most index of the array to be sorted as arguments. Middle index (mid) of the array is calculated as  $(\text{left} + \text{right})/2$ . Check if  $(\text{left} < \text{right})$  because we have to sort only when  $\text{left} < \text{right}$  because when  $\text{left} = \text{right}$  it is anyhow sorted. Sort the left part by calling MergeSort() function again over the left part MergeSort(array,left,mid) and the right part by recursive call of MergeSort function as MergeSort(array,mid + 1, right). Lastly merge the two arrays using the Merge function.

### **Merge() function:**

It takes the array, left-most , middle and right-most index of the array to be merged as arguments.

Finally copy back the sorted array to the original array.

### **Source Code:**

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

void merge(int a[],int start,int mid,int end)
{
    int b[20],k=0,i,j;
    i=start;
    j=mid+1;
    while((i<mid+1)&&(j<end+1))
    {
        if(a[i]<a[j])
```

```

        {
            b[k]=a[i];
            i++;
        }
    else
    {
        b[k]=a[j];
        j++;
    }
    k++;
}
while(i<mid+1)
{
    b[k]=a[i];
    k++;
    i++;
}
while(j<end+1)
{
    b[k]=a[j];
    j++;
    k++;
}
for(i=0;i<k;i++)
{
    a[start]=b[i];
    start++;
}

```

```

}

void merge_sort(int a[20],int start,int end)
{
    int mid;
    if(start<end)
    {
        mid=(start+end)/2;
        merge_sort(a,start,mid);
        merge_sort(a,mid+1,end);
        merge(a,start,mid,end);
    }
}

void main()
{
    int i,n,a[20],start=0,end;
    clrscr();
    printf("\nEnter the n:");
    scanf("%d",&n);
    printf("\nEnter the elements:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("\nElements before sort:\n");
    for(i=0;i<n;i++)
        printf("%5d",a[i]);
    end=n-1;
    merge_sort(a,start,end);
    printf("\nElements after sort:\n");
    for(i=0;i<n;i++)

```

```

printf("%5d",a[i]);

getch();

}

```

### Output:

Enter the n:5

Enter the elements:102 321 365 320 111

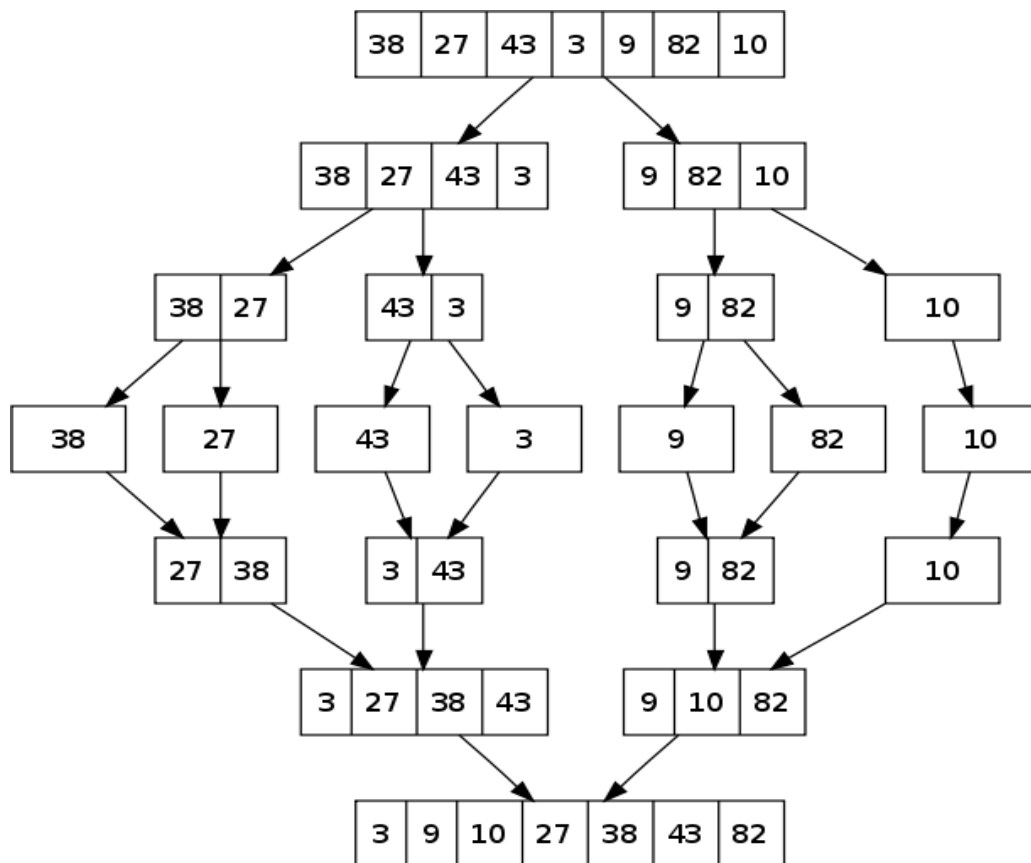
Elements before sort:

102 321 365 320 111

Elements after sort:

102 111 320 321 365

### Step-by-step example:



Merge Sort Example

### Time Complexity:

Worst Case Performance	$O(N \log_2 N)$
Best Case Performance(nearly)	$O(N \log_2 N)$
Average Case Performance	$O(N \log_2 N)$

### Comparison of Sorting Algorithms:

Time Complexity comparison of Sorting Algorithms:

Algorithm	Data Structure	Time Complexity		
		Best	Average	Worst
Quicksort	Array	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$
Mergesort	Array	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$
Bubble Sort	Array	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	Array	$O(n)$	$O(n^2)$	$O(n^2)$
Select Sort	Array	$O(n^2)$	$O(n^2)$	$O(n^2)$

Space Complexity comparison of Sorting Algorithms:

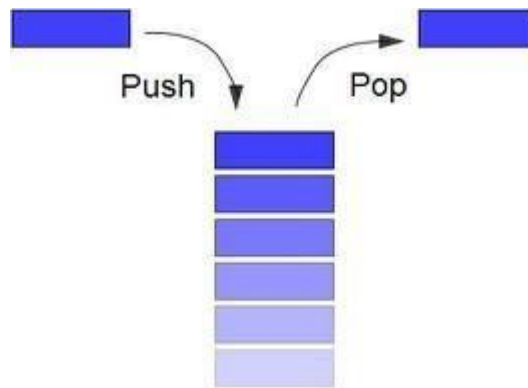
Algorithm	Data Structure	Worst Case Auxiliary Space Complexity
Quicksort	Array	$O(n)$
Mergesort	Array	$O(n)$
Bubble Sort	Array	$O(1)$
Insertion Sort	Array	$O(1)$
Select Sort	Array	$O(1)$
Bucket Sort	Array	$O(nk)$

## UNIT – II LINEAR DATA STRUCTURES

### Stacks Primitive Operations:

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. Push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.

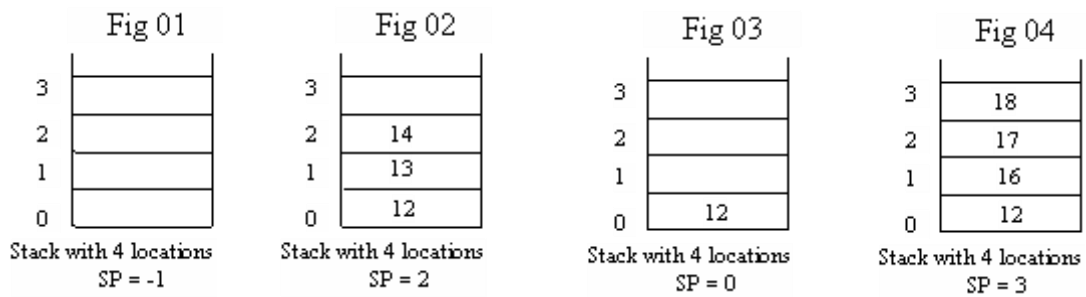
A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items or results in an empty stack, but, if the stack is empty, it goes into underflow state, which means no items are present in stack to be removed.



### Stack (ADT) Data Structure:

Stack is an Abstract data structure (ADT) works on the principle Last In First Out (LIFO). The last element add to the stack is the first element to be delete. Insertion and deletion can be takes place at one end called TOP. It looks like one side closed tube.

- The add operation of the stack is called push operation
- The delete operation is called as pop operation.
- Push operation on a full stack causes stack overflow.
- Pop operation on an empty stack causes stack underflow.
- SP is a pointer, which is used to access the top element of the stack.
- If you push elements that are added at the top of the stack;
- In the same way when we pop the elements, the element at the top of the stack is deleted.



### Operations of stack:

There are two operations applied on stack they are

1. push
2. pop.

While performing push & pop operations the following test must be conducted on the stack.

- 1) Stack is empty or not
- 2) Stack is full or not

#### Push:

Push operation is used to add new elements in to the stack. At the time of addition first check the stack is full or not. If the stack is full it generates an error message "stack overflow".

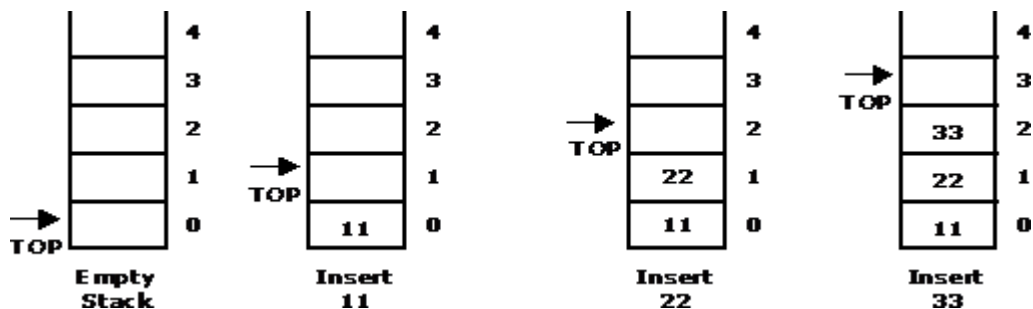
#### Pop:

Pop operation is used to delete elements from the stack. At the time of deletion first check the stack is empty or not. If the stack is empty it generates an error message "stack underflow".

### Representation of a Stack using Arrays:

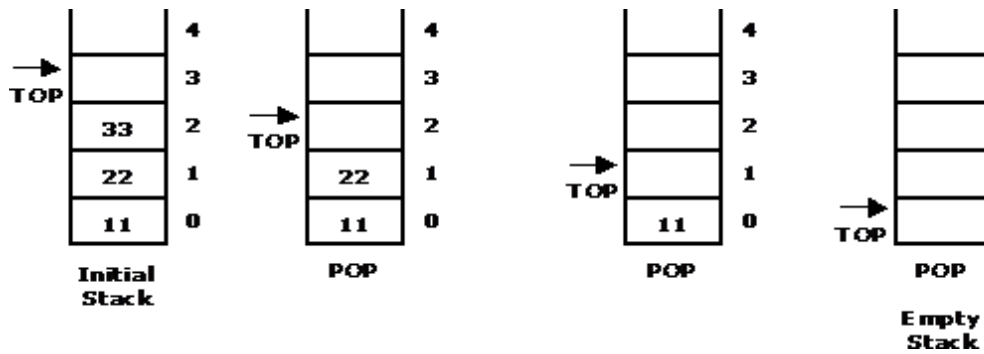
Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a stack overflow condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a stack underflow condition.

When a element is added to a stack, the operation is performed by push().



When an element is taken off from the stack, the operation is performed by pop().





### Source code for stack operations, using array:

**STACK:** Stack is a linear data structure which works under the principle of last in first out. Basic operations: push, pop, display.

1. **PUSH:** if (top==MAX), display **Stack overflow** else reading the data and making stack [top] =data and incrementing the top value by doing top++.
2. **POP:** if (top==0), display **Stack underflow** else printing the element at the top of the stack and decrementing the top value by doing the top--.
3. **DISPLAY:** IF (TOP==0), display **Stack is empty** else printing the elements in the stack from stack [0] to stack [top].

```
# include <stdio.h>

# include <conio.h>

# include <stdlib.h>

# define MAX 6

int stack[MAX];

int top = 0;

int menu()

{

    int ch;

    clrscr();

    printf("\n ... Stack operations using ARRAY... ");

    printf("\n -----*****-----\n");

    printf("\n 1. Push ");

    printf("\n 2. Pop ");

    printf("\n 3. Display");
```

```

        printf("\n 4. Quit ");

        printf("\n Enter your choice: ");

        scanf("%d", &ch);

        return ch;
    }

void display()
{
    int i;

    if(top == 0)
    {
        printf("\n\nStack empty..");

        return;
    }
    else
    {
        printf("\n\nElements in stack:");

        for(i = 0; i < top; i++)
            printf("\t%d", stack[i]);

    }
}

void pop()
{
    if(top == 0)
    {
        printf("\n\nStack Underflow..");

        return;
    }
    else

```

```

        printf("\n\npopped element is: %d ", stack[--top]);
    }
void push()
{
    int data;
    if(top == MAX)
    {
        printf("\n\nStack Overflow..");
        return;
    }
    else
    {
        printf("\n\nEnter data: ");
        scanf("%d", &data);
        stack[top] = data;
        top = top + 1;
        printf("\n\nData Pushed into the stack");
    }
}
void main()
{
    int ch;
    do
    {
        ch = menu();
        switch(ch)
        {
            case 1: push(); break;

```

```

        case 2: pop(); break;

        case 3: display(); break;

        case 4: exit(0);

    }

} while(1);

}

```

### Output:

Stack using arrays:

```
*****
```

1.Push

2.Pop

3.Display

4.Quit

Enter your choice:1

Enter charecter:2

charecter inserted successfully

Stack using arrays:

```
*****
```

1.Push

2.Pop

3.Display

4.Quit

Enter your choice:2

The popped element from stack is:2

Stack using arrays:

```
*****
```

1.Push

2.Pop

3.Display

4.Quit

Enter your choice:3

Stack is empty

Stack using arrays:

\*\*\*\*\*

1.Push

2.Pop

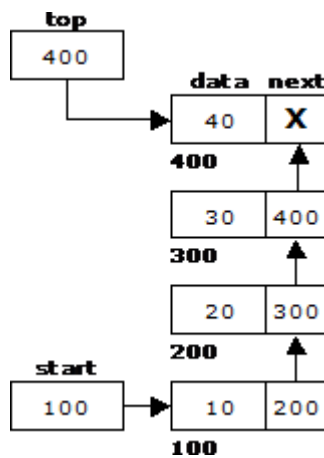
3.Display

4.Quit

Enter your choice:4

### Linked List Implementation of Stack:

We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using top pointer.



### Source code for stack operations, using linked list:

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

struct stack
{
    int data;
```

```

        struct stack *next;

};

typedef struct stack node;

node *start = NULL;

node *top = NULL;

node* getnode()
{
    node *temp;

    temp=(node *)malloc(sizeof(node));

    printf("\n enter data : ");

    scanf("%d",&temp -> data);

    temp -> next=NULL;

    return temp;
}

void push(node *newnode)
{
    node *temp;

    if(newnode==NULL)
    {
        printf ("\n stack overflow... ");

        return;
    }

    if(start == NULL)
    {
        start = newnode;

        top = newnode;
    }

    else
    {

```

```

        temp = start;

        while(temp -> next!= NULL)

            temp = temp -> next;

        temp -> next =newnode;

        top = newnode;

    }

    Printf ("\n\n\t data pushed into stack...");

}

void pop()
{
    node *temp;
    if(top == NULL)
    {
        printf("\n\n\t stack overflow...");
        return;
    }
    temp = start;
    if(start -> next == NULL)
    {
        printf("\n\n\t popped element is %d",top -> data);
        start = NULL;
        free(top);
        top = NULL;
    }
    else
    {
        while(temp -> next!=top)

```

```

        {
            temp = temp -> next;
        }
        temp -> next = NULL;
        printf("\n\n\t popped element is %d", top -> data);
        free(top);
        top = temp;
    }
}

void display()
{
    node *temp;
    if(top == NULL)
    {
        printf("\n\n\t stack is empty...");
    }
    else
    {
        temp = start;
        printf ("\n\n\t elements in the stack..\n");
        printf("%5d",temp -> data);
        while(temp!= top)
        {
            temp = temp -> next;
            printf("%5d",temp -> data);
        }
    }
}

```



```

int menu()
{
    int ch;

    printf("\n\t STACK OPERATIONS USING POINTERS : ");
    printf("\n ----*****----- ");
    printf("\n 1. Push ");
    printf("\n 2. Pop ");
    printf("\n 3. Display ");
    printf("\n 4. Quit");
    printf("\n enter your choice : ");
    scanf("%d",&ch);

    return ch;
}

void main()
{
    int ch;
    node *newnode;
    do
    {
        ch=menu();
        switch(ch)
        {
            case 1 : newnode=getnode();
                     push(newnode); break;
            case 2 : pop(); break;
            case 3 : display(); break;
            case 4 : return;
        }
    }
}

```

```
        }while(1);  
}
```

### Output:

#### STACK OPERATIONS USING POINTERS

-----\*\*\*\*\*-----

1. Push
2. Pop
3. Display
4. Quit

enter your choice : 1

enter data : 10

data pushed into Stack

#### STACK OPERATIONS USING POINTERS

-----\*\*\*\*\*-----

1. Push
2. Pop
3. Display
4. Quit

enter your choice : 1

enter data : 20

data pushed into Stack

#### STACK OPERATIONS USING POINTERS

-----\*\*\*\*\*-----

1. Push
2. Pop
3. Display
4. Quit

enter your choice : 1

enter data : 30

data pushed into Stack

#### STACK OPERATIONS USING POINTERS

-----\*\*\*\*\*-----

1. Push
2. Pop
3. Display
4. Quit

enter your choice : 3

elements in the stack..

10 20 30

#### STACK OPERATIONS USING POINTERS

-----\*\*\*\*\*-----

1. Push
2. Pop
3. Display
4. Quit

enter your choice : 2

popped element is 30

#### STACK OPERATIONS USING POINTERS

-----\*\*\*\*\*-----

1. Push
2. Pop
3. Display
4. Quit

enter your choice : 3

elements in the stack..

10 20

## STACK OPERATIONS USING POINTERS

-----\*\*\*\*\*-----

1. Push
2. Pop
3. Display
4. Quit

enter your choice : 2

popped element is 20

## STACK OPERATIONS USING POINTERS

-----\*\*\*\*\*-----

1. Push
2. Pop
3. Display
4. Quit

enter your choice : 3

elements in the stack..

10

## STACK OPERATIONS USING POINTERS

-----\*\*\*\*\*-----

1. Push
2. Pop
3. Display
4. Quit

enter your choice : 2

popped element is 10

## STACK OPERATIONS USING POINTERS

-----\*\*\*\*\*-----

1. Push

2. Pop

3. Display

4. Quit

enter your choice : 3

stack empty

### **Stack Applications:**

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.
4. In recursion, all intermediate arguments and return values are stored on the processor's stack.
5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.
6. Depth first search uses a stack data structure to find an element from a graph.

### **Factorial Calculation:**

#### **Source Code:**

```
#include<stdio.h>

#include<conio.h>

void main()
{
    int i,n,stack[20],top=0,f=1;

    printf("Enter n:");

    scanf("%d",&n);

    i=n;

    while(n>0)
    {
        stack[top]=n;

        top++;

        n--;
    }
}
```

```

while(top>0)
{
    f=f*stack[--top];
}
printf("factorial of %d is %d",i,f);
}

```

### Output:

Enter n:5

factorial of 5 is 120

### In-fix- to Postfix Transformation:

#### Procedure:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2.
  - a) If the scanned symbol is left parenthesis, push it onto the stack.
  - b) If the scanned symbol is an operand, then place directly in the postfix expression (output).
  - c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
  - d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Convert the following infix expression  $A + B * C - D / E * H$  into its equivalent postfix expression.

Symbol	Postfix string	Stack	Remarks
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	-	

-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

#### Source Code:

```
#include<stdio.h>

#include<string.h>

char opstack[50];
char infix[50];
char postfix[50];
int i,j,top=0;
void pop();
void push(char);
int lesspriority(char,char);
void main()
{
    char ch;
    printf("enter infix expression:\n");
    gets(infix);
    while((ch=infix[i++])!='\0')
    {
        switch(ch)
        {
            case ' ':break;
```

```

        case '(':
        case '+':
        case '-':
        case '*':
        case '/':
        case '^':
        case '%':    push(ch);
                    break;
        case ')':    pop();
                    break;
        default:    postfix[j]=ch;
                    j++;
    }
}
while(top>=0)
{
    postfix[j]=opstack[--top];
    j++;
}
postfix[--j]='\0';
printf("\n infix expression:%s",infix);
printf("\n postfix expression:%s",postfix);
}
int lesspriority(char op,char op_at_stack)
{
    int k;
    int pv1;
    int pv2;

```



```

char operators[]={'+', '-', '*', '/', '%', '^', '('};
int priority_value[]={0,0,1,1,2,3,4};
if(op_at_stack=='(')
    return 0;
for(k=0;k<6;k++)
{
    if(op==operators[k])
        pv1=priority_value[k];
}
for(k=0;k<6;k++)
{
    if(op_at_stack==operators[k])
        pv2=priority_value[k];
}
if(pv1<pv2)
    return 1;
else
    return 0;
}

void push(char op)
{
    if(top==0)
    {
        opstack[top]=op;
        top++;
    }
    else
    {

```

```

        if(op!='(')
        {
            while(lesspriority(op,opstack[top-1])==1 && top>0)
            {
                postfix[j]=opstack[--top];
                j++;
            }
            opstack[top]=op;
            top++;
        }
    }
    void pop()
    {
        While (opstack [--top]! ='(')
        {
            postfix[j]=opstack[top];
            j++;
        }
    }
}

```

### Output:

enter infix expression:

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

infix expression: (A+B)\*(C-D)

postfix expression: AB+CD-\*

### Evaluating Arithmetic Expressions:

#### Procedure:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.

Evaluate the postfix expression: 6 5 2 3 + 8 \* + 3 + \*

Symbol	Operand 1	Operand 2	Value	Stack	Remarks
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a „+“ is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a „*“ is seen, so 8 and 5 are popped as $8 * 5 = 40$ is pushed
+	5	40	45	6, 45	Next, a „+“ is seen, so 40 and 5 are popped and $40 + 5 = 45$ is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, „+“ pops 3 and 45 and pushes $45 + 3 = 48$ is pushed
*	6	48	288	<b>288</b>	Finally, a „*“ is seen and 48 and 6 are popped, the result $6 * 48 = 288$ is pushed

**Source Code:**

```
include<stdio.h>

#include<conio.h>

void evaluate_postfix(char a[20])
{
    int i=0,top=0,n1,n2,r,s[20];
    for(i=0;a[i]!='\0';i++)
    {
        if((a[i]>=48)&&(a[i]<=57))
```

```

        {
            s[top]=a[i]-48;
            top++;
        }
        else
        {
            n2=s[--top];
            n1=s[--top];
            switch(a[i])
            {
                case '+': r=n1+n2;break;
                case '-': r=n1-n2;break;
                case '*': r=n1*n2;break;
                case '/': r=n1/n2;break;
                case '^': r=n1^n2;break;
            }
            s[top]=r;
            top++;
        }
    }
    printf("RESULT IS %d",s[0]);
    getch();
}

void main()
{
    char a[20];
    printf("Enter the postfix expresion:");
    gets(a);

```

```

    evaluate_postfix(a);
}

```

### Output:

Enter the postfix expression: 23+5\*

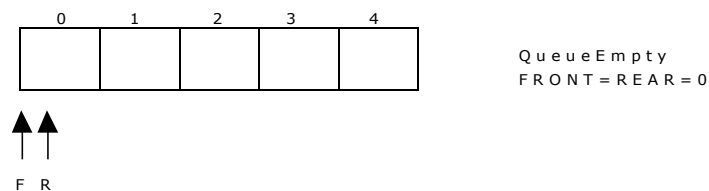
RESULT IS 25

### Basic Queue Operations:

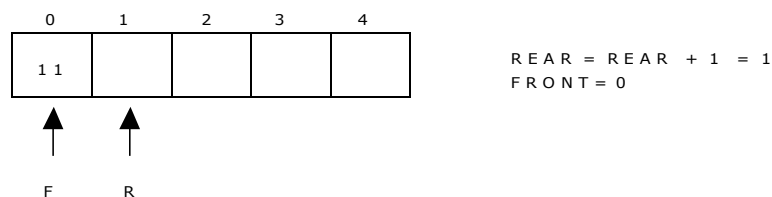
A queue is a data structure that is best described as "first in, first out". A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. A real world example of a queue is people waiting in line at the bank. As each person enters the bank, he or she is "enqueued" at the back of the line. When a teller becomes available, they are "dequeued" at the front of the line.

### Representation of a Queue using Array:

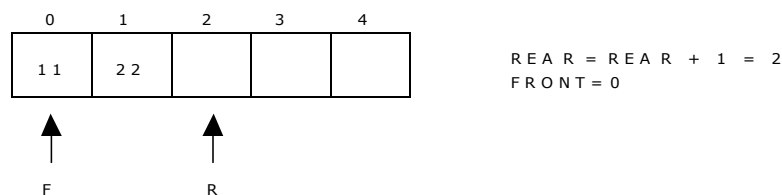
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



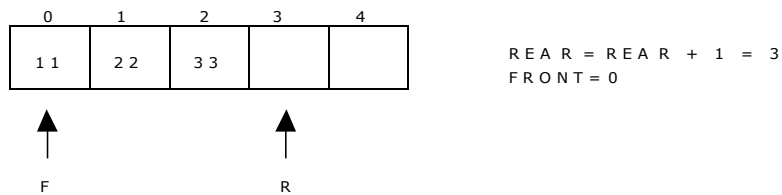
Now, insert 11 to the queue. Then queue status will be:



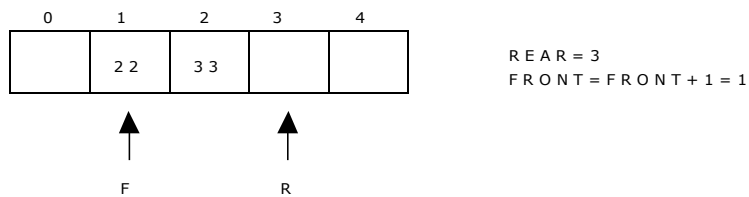
Next, insert 22 to the queue. Then the queue status is:



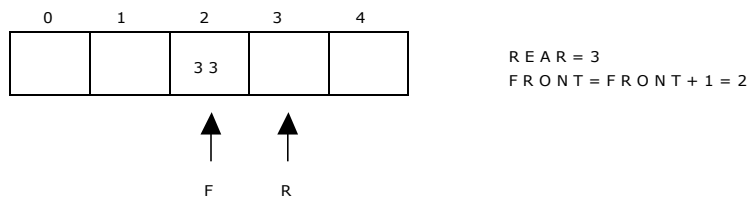
Again insert another element 33 to the queue. The status of the queue is:



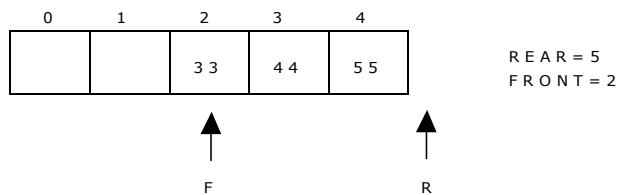
Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



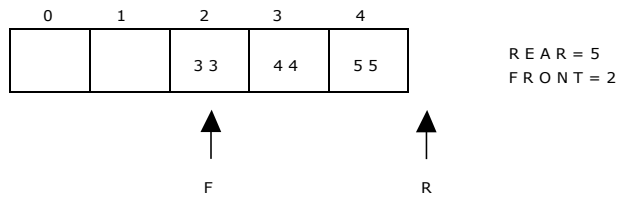
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



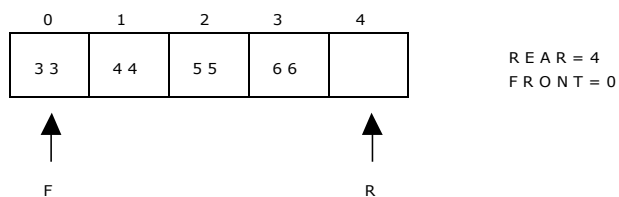
Now, insert new elements 44 and 55 into the queue. The queue status is:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To overcome this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue**.

### Procedure for Queue operations using array:

In order to create a queue we require a one dimensional array  $Q(1:n)$  and two variables *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and rear always points to the last element in the queue. Thus,  $front = rear$  if and only if there are no elements in the queue. The initial condition then is  $front = rear = 0$ .

The various queue operations to perform creation, deletion and display the elements in a queue are as follows:

1. insertQ(): inserts an element at the end of queue Q.
2. deleteQ(): deletes the first element of Q.
3. displayQ(): displays the elements in the queue.

### Source Code:

```
# include <stdio.h>

# include <conio.h>

# define MAX 6

int Q[MAX];

int front, rear;
```

```

void insertQ()
{
    int data;
    if(rear == MAX)
    {
        printf("\n Linear Queue is full");
        return;
    }
    else
    {
        printf("\n Enter data: ");
        scanf("%d", &data);
        Q[rear] = data;
        rear++;
        printf("\n Data Inserted in the Queue ");
    }
}

```

```

void deleteQ()
{
    if(rear == front)
    {
        printf("\n\n Queue is Empty..");
        return;
    }
    else
    {
        printf("\n Deleted element from Queue is %d", Q[front]);
        front++;
    }
}

```



```

    }
}

void displayQ()
{
    int i;
    if(front == rear)
    {
        printf("\n\n\t Queue is Empty");
        return;
    }
    else
    {
        printf("\n Elements in Queue are: ");
        for(i = front; i < rear; i++)
        {
            printf("%d\t", Q[i]);
        }
    }
}

int menu()
{
    int ch;
    clrscr();
    printf("\n \tQueue operations using ARRAY..");
    printf("\n -----*****-----\n");
    printf("\n 1. Insert ");
    printf("\n 2. Delete ");
    printf("\n 3. Display");

```

```

        printf("\n 4. Quit ");

        printf("\n Enter your choice: ");

        scanf("%d", &ch);

        return ch;
    }

void main()
{
    int ch;

    do
    {
        ch = menu();

        switch(ch)
        {
            case 1: insertQ(); break;
            case 2: deleteQ(); break;
            case 3: displayQ(); break;
            case 4: return;

        }

    } while(1);
}

```

### **Output:**

Queue using arrays:

\*\*\*\*\*

1.Insertion

2.Deletion

3.Display

4.QUIT

Enter your choice:1

Enter data:2

Data entered successfully

Queue using arrays:

\*\*\*\*\*

1.Insertion

2.Deletion

3.Display

4.QUIT

Enter your choice:1

Enter data:25

Data entered successfully

Queue using arrays:

\*\*\*\*\*

1.Insertion

2.Deletion

3.Display

4.QUIT

Enter your choice:2

The deleted element from the queue is:2

Queue using arrays:

\*\*\*\*\*

1.Insertion

2.Deletion

3.Display

4.QUIT

Enter your choice:1

Enter data:36

Data entered successfully

Queue using arrays:

\*\*\*\*\*

1.Insertion

2.Deletion

3.Display

4.QUIT

Enter your choice:1

Enter data:25

Data entered successfully

Queue using arrays:

\*\*\*\*\*

1.Insertion

2.Deletion

3.Display

4.QUIT

Enter your choice:2

The deleted element from the queue is:25

Queue using arrays:

\*\*\*\*\*

1.Insertion

2.Deletion

3.Display

4.QUIT

Enter your choice:3

The elements in the queue are: 36 25

Queue using arrays:

\*\*\*\*\*

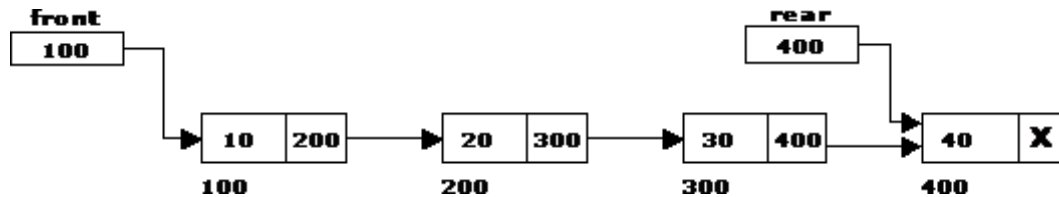
1.Insertion

2.Deletion

3.Display

4.QUIT

**Linked List Implementation of Queue:** We can represent a queue as a linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers front and rear for our linked queue implementation.



#### Source Code:

```
# include <stdio.h>

# include <stdlib.h>

# include <conio.h>

struct queue

{
    int data;

    struct queue *next;

};

typedef struct queue node;

node *front = NULL;

node *rear = NULL;

node* getnode()

{

    node *temp;

    temp = (node *) malloc(sizeof(node)) ;

    printf("\n Enter data ");

    scanf("%d", &temp -> data);

    temp -> next = NULL;

    return temp;
```

```

}

void insertQ()
{
    node *newnode;
    newnode = getnode();
    if(newnode == NULL)
    {
        printf("\n Queue Full");
        return;
    }
    if(front == NULL)
    {
        front = newnode;
        rear = newnode;
    }
    else
    {
        rear -> next = newnode;
        rear = newnode;
    }
    printf("\n\n\t Data Inserted into the Queue..");
}

void deleteQ()
{
    node *temp;
    if(front == NULL)
    {
        printf("\n\n\t Empty Queue..");
    }
}

```

```

        return;
    }
    temp = front;
    front = front -> next;
    printf("\n\n\t Deleted element from queue is %d ", temp -> data);
    free(temp);
}

```

```

void displayQ()
{
    node *temp;
    if(front == NULL)
    {
        printf("\n\n\t Empty Queue ");
    }
    else
    {
        temp = front;
        printf("\n\n\n\t Elements in the Queue are: ");
        while(temp != NULL )
        {
            printf("%5d ", temp -> data);
            temp = temp -> next;
        }
    }
}

char menu()
{

```

```

    char ch;

    printf("\n \t..Queue operations using pointers.. ");

    printf("\n\t -----*****-----\n");

    printf("\n 1. Insert ");

    printf("\n 2. Delete ");

    printf("\n 3. Display");

    printf("\n 4. Quit ");

    printf("\n Enter your choice: ");

    ch = getche();

    return ch;

}

void main()

{

    char ch;

    do

    {

        ch = menu();

        switch(ch)

        {

            case '1' : insertQ(); break;

            case '2' : deleteQ(); break;

            case '3' : displayQ(); break;

            case '4': return;

        }

    } while(ch != '4');

}

```

### **Output:**

QUEUE OPERATIONS USING POINTERS :



-----\*\*\*\*\*-----

1. Insert
2. Delete
3. Display
4. Quit

enter your choice : 1

enter data : 10

data inserted into Queue...

QUEUE OPERATIONS USING POINTERS :

-----\*\*\*\*\*-----

1. Insert
2. Delete
3. Display
4. Quit

enter your choice : 1

enter data : 20

data inserted into Queue...

QUEUE OPERATIONS USING POINTERS :

-----\*\*\*\*\*-----

1. Insert
2. Delete
3. Display
4. Quit

enter your choice : 1

enter data : 30

data inserted into Queue...

QUEUE OPERATIONS USING POINTERS :

-----\*\*\*\*\*-----

1. Insert
2. Delete
3. Display
4. Quit

enter your choice : 2

popped element is 10

QUEUE OPERATIONS USING POINTERS :

-----\*\*\*\*\*-----

1. Insert
2. Delete
3. Display
4. Quit

enter your choice : 3

Elements in the Queue...

20 30

QUEUE OPERATIONS USING POINTERS :

-----\*\*\*\*\*-----

1. Insert
2. Delete
3. Display
4. Quit

enter your choice : 2

popped element is 20

QUEUE OPERATIONS USING POINTERS :

-----\*\*\*\*\*-----

1. Insert
2. Delete
3. Display

4. Quit

enter your choice : 2

popped element is 30

QUEUE OPERATIONS USING POINTERS :

-----\*\*\*\*\*-----

1. Insert

2. Delete

3. Display

4. Quit

enter your choice : 3

QUEUE is EMPTY...

### **Applications of Queues:**

1. It is used to schedule the jobs to be processed by the CPU.
2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
3. Breadth first search uses a queue data structure to find an element from a graph.

### **Disadvantages of Linear Queue:**

There are two problems associated with linear queue. They are:

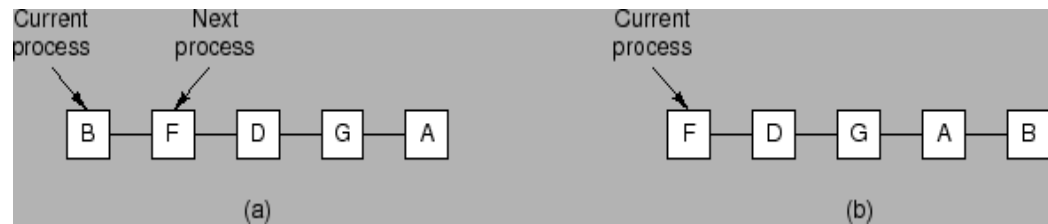
- ☐ Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.
- ☐ Signaling queue full: even if the queue is having vacant position.

### **Round Robin Algorithm:**

The round-robin (RR) scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but pre-emption is added to switch between processes. A small unit of time, called a time quantum or time slices, is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue. To implement RR scheduling

- ☐ We keep the ready queue as a FIFO queue of processes.
- ☐ New processes are added to the tail of the ready queue.
- ☐ The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
- ☐ The process may have a CPU burst of less than 1 time quantum.

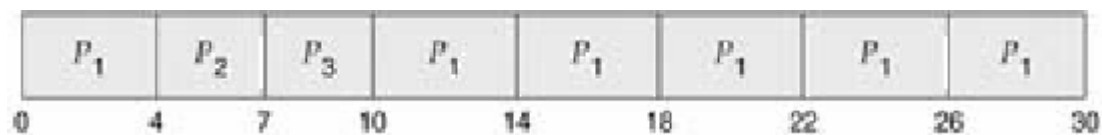
- In this case, the process itself will release the CPU voluntarily.
- The scheduler will then proceed to the next process in the ready queue.
- Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum,
  - The timer will go off and will cause an interrupt to the OS.
  - A context switch will be executed, and the process will be put at the tail of the ready queue.
  - The CPU scheduler will then select the next process in the ready queue.



The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds: (a time quantum of 4 milliseconds)

	Burst	Waiting	Turnaround
Process	Time	Time	Time
	24	6	30
	3	4	7
	3	7	10
Average	-	5.66	15.66

Using round-robin scheduling, we would schedule these processes according to the following chart:



**Source Code:**

```

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#define MAX 4

#define TS 5

struct process
{
    char pname[20];
    int bt;
    int wt;
}pq[MAX];
int f,r;
void readprocess()
{
    printf("\nEnter the process name nd execution time:\n");
    for(r=0;r<MAX;r++)
    {
        scanf("%s%d",pq[r].pname,&pq[r].bt);
        pq[r].wt=0;
    }
}
void calculatewt()
{
    int i,j,ctr=0;
    f=0;
    while(1)
    {

```

```

    if(pq[f].bt>0)
    {
        pq[f].bt=pq[f].bt-TS;
        j=f;
        for(i=0;i<MAX;i++)
        {
            j=(j+1)%MAX;
            if(j == f)
                break;
            if(pq[j].bt>0)
            {
                pq[j].wt+=TS;
            }
        }
    }
    f=(f+1)%MAX;
    ctr=0;
    for(i=0;i<MAX;i++)
    {
        if(pq[i].bt<=0)
            ctr++;
    }
    if(ctr==MAX)
        break;
}

}

```

```

void printwt()
{
    float twt=0;int i;
    for(i=0;i<MAX;i++)
    {
        printf("\n%s\t%d",pq[i].pname,pq[i].wt);
        twt+=pq[i].wt;
    }
    printf("\nAvgerage waiting time : %f",(twt/MAX));
}

void main()
{
    readprocess();
    calculatewt();
    printwt();
}

```

### Output:

Enter the process name nd execution time:

P1

20

P2

30

P3

15

P4

5

P1 35

P2 40

P3 35

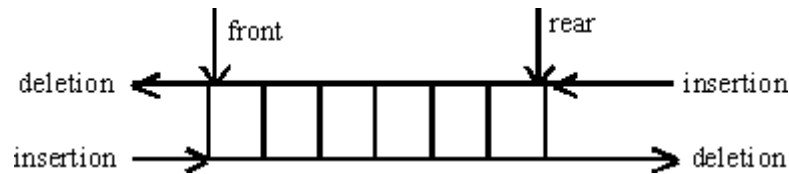
P4 15

Average waiting time : 31.250000

### DEQUEUE(Double Ended Queue):

A **double-ended queue** (**dequeue**, often abbreviated to **deque**, pronounced *deck*) generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail). It is also often called a **head-tail linked list**. Like an ordinary queue, a double-ended queue is a data structure it supports the following operations: enq\_front, enq\_back, deq\_front, deq\_back, and empty. Dequeue can behave like a queue by using only enq\_front and deq\_front, and behaves like a stack by using only enq\_front and deq\_rear.

The DeQueue is represented as follows.



DEQUEUE can be represented in two ways they are

- 1) Input restricted DEQUEUE(IRD)
- 2) output restricted DEQUEUE(ORD)

The output restricted DEQUEUE allows deletions from only one end and input restricted DEQUEUE allow insertions at only one end. The DEQUEUE can be constructed in two ways they are

- 1) Using array
- 2) Using linked list

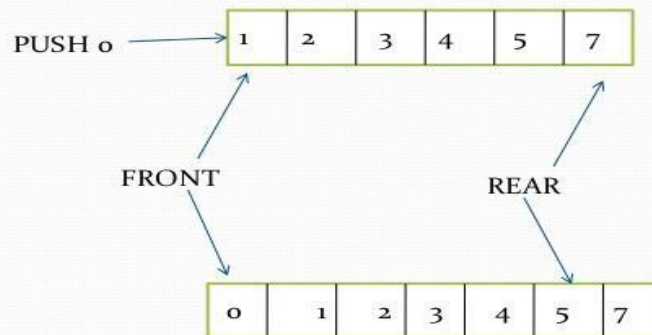
### Operations in DEQUEUE

1. Insert element at back
2. Insert element at front
3. Remove element at front
4. Remove element at back



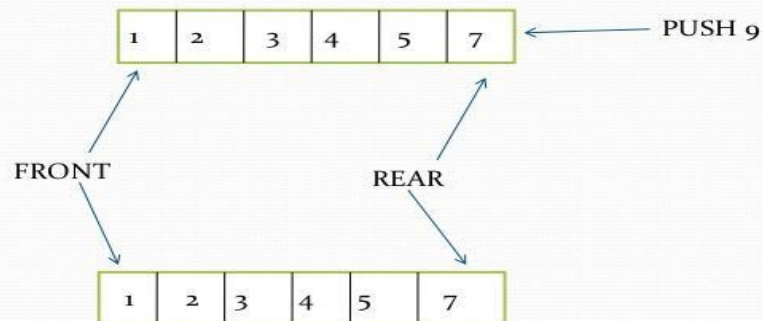
## Insert\_front

- `insert_front()` is a operation used to push an element into the front of the *Deque*.



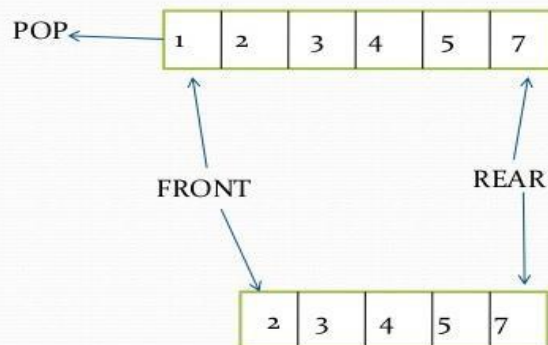
## Insert\_back

- `insert_back()` is a operation used to push an element at the back of a *Deque*.



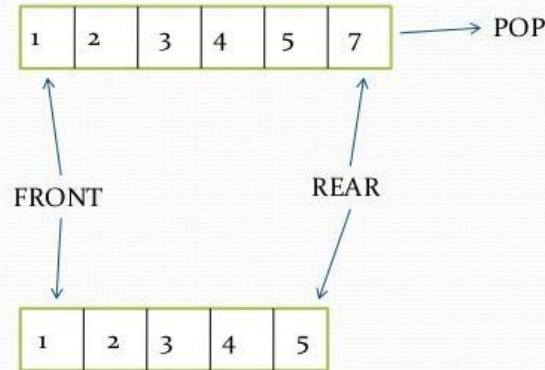
## Remove\_front

- `remove_front()` is a operation used to pop an element on front of the *Deque*.



## Remove\_back

- `remove_front()` is a operation used to pop an element on front of the *Deque*.



### Applications of DEQUE:

1. The A-Steal algorithm implements task scheduling for several processors (multiprocessor scheduling).
2. The processor gets the first element from the deque.
3. When one of the processor completes execution of its own threads it can steal a thread from another processor.
4. It gets the last element from the deque of another processor and executes it.

### Circular Queue:

Circular queue is a linear data structure. It follows FIFO principle. In circular queue the last node is connected back to the first node to make a circle.

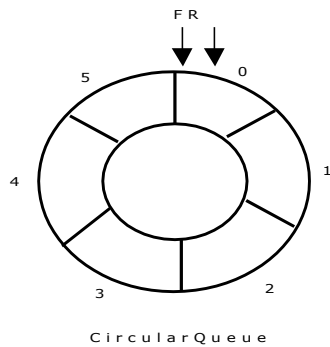
- Circular linked list follow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as “Ring buffer”.
- Items can inserted and deleted from a queue in  $O(1)$  time.

Circular Queue can be created in three ways they are

1. Using single linked list
2. Using double linked list
3. Using arrays

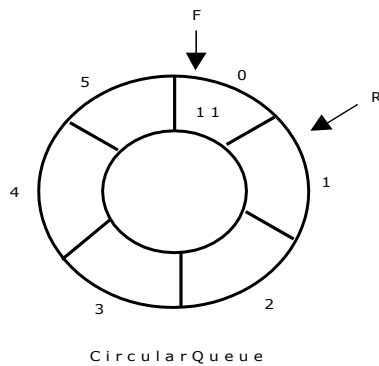
## Representation of Circular Queue:

Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



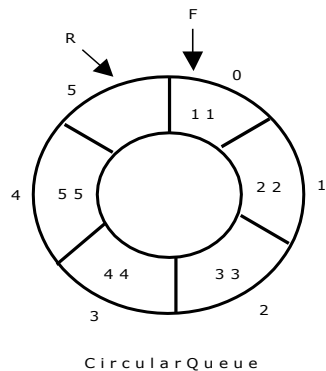
```
QueueEmpty  
MAX = 6  
FRONT = REAR = 0  
COUNT = 0
```

Now, insert 11 to the circular queue. Then circular queue status will be:



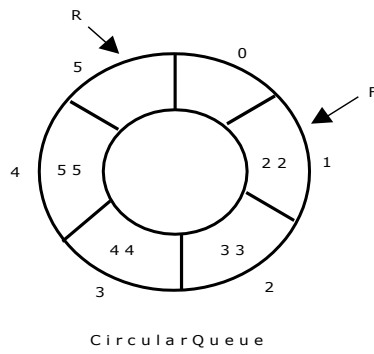
```
FRONT = 0  
REAR = (REAR + 1) % 6 = 1  
COUNT = 1
```

Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



```
FRONT = 0, REAR = 5  
REAR = REAR % 6 = 5  
COUNT = 5
```

Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:

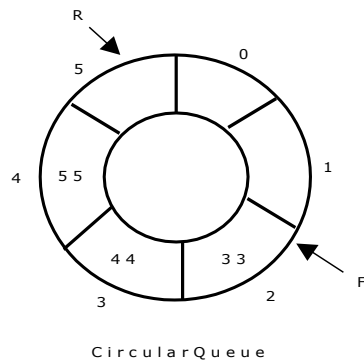


```

FRONT = (FRONT + 1) % 6 = 1
REAR = 5
COUNT = COUNT - 1 = 4

```

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:

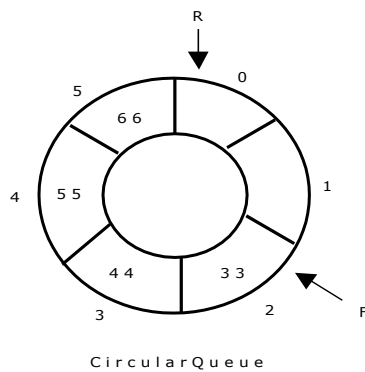


```

FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3

```

Again, insert another element 66 to the circular queue. The status of the circular queue is:

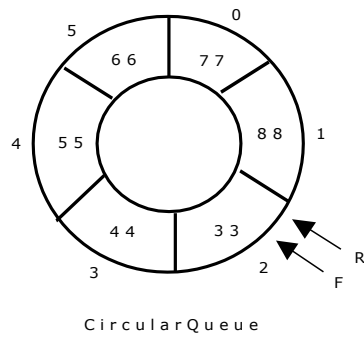


```

FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4

```

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



```
FRONT = 2, REAR = 2
REAR = REAR % 6 = 2
COUNT = 6
```

Now, if we insert an element to the circular queue, as  $COUNT = MAX$  we cannot add the element to circular queue. So, the circular queue is *full*.

### Source Code:

```
# include <stdio.h>

# include <conio.h>

# define MAX 6

int CQ[MAX];

int front = 0;

int rear = 0;

int count = 0;

void insertCQ()
{
    int data;

    if(count == MAX)
    {
        printf("\n Circular Queue is Full");
    }
    else
    {
        printf("\n Enter data: ");

        scanf("%d", &data);

        CQ[rear] = data;
```

```

        rear = (rear + 1) % MAX;

        count ++;

        printf("\n Data Inserted in the Circular Queue ");

    }

}

void deleteCQ()
{
    if(count == 0)
    {
        printf("\n\nCircular Queue is Empty..");
    }
    else
    {
        printf("\n Deleted element from Circular Queue is %d ", CQ[front]);
        front = (front + 1) % MAX;
        count --;
    }
}

void displayCQ()
{
    int i, j;
    if(count == 0)
    {
        printf("\n\n\t Circular Queue is Empty ");
    }
    else
    {
        printf("\n Elements in Circular Queue are: ");

```

```

        j = count;
        for(i = front; j != 0; j--)
        {
            printf("%d\t", CQ[i]);

            i = (i + 1) % MAX;
        }
    }
}

int menu()
{
    int ch;

    clrscr();

    printf("\n \t Circular Queue Operations using ARRAY..");
    printf("\n -----*****-----\n");

    printf("\n 1. Insert ");
    printf("\n 2. Delete ");
    printf("\n 3. Display");
    printf("\n 4. Quit ");

    printf("\n Enter Your Choice: ");

    scanf("%d", &ch);

    return ch;
}

void main()
{
    int ch;

    do
    {
        ch = menu();
    }
}

```

```

        switch(ch)
        {
            case 1: insertCQ(); break;
            case 2: deleteCQ(); break;
            case 3: displayCQ(); break;
            case 4: return;
            default: printf("\n Invalid Choice ");
        }
    } while(1);
}

```

### Output:

Circular Queue using arrays:

\*\*\*\*\*

1.Insertion

2.Deletion

3.Display

4.QUIT

Enter your choice:1

Enter data:3

Data entered successfully

Circular Queue using arrays:

\*\*\*\*\*

1.Insertion

2.Deletion

3.Display

4.QUIT

Enter your choice:1

Enter data:2



Data entered successfully

Circular Queue using arrays:

\*\*\*\*\*

1.Insertion

2.Deletion

3.Display

4.QUIT

Enter your choice:2

Data deleted at queue is:3

Circular Queue using arrays:

\*\*\*\*\*

1.Insertion

2.Deletion

3.Display

4.QUIT

Enter your choice:1

Enter data:36

Data entered successfully

Circular Queue using arrays:

\*\*\*\*\*

1.Insertion

2.Deletion

3.Display

4.QUIT

Enter your choice:1

Enter data:32

Data entered successfully

Circular Queue using arrays:

\*\*\*\*\*

1.Insertion

2.Deletion

3.Display

4.QUIT

Enter your choice:2

Data deleted at queue is:2

Circular Queue using arrays:

\*\*\*\*\*

1.Insertion

2.Deletion

3.Display

4.QUIT

Enter your choice:3

3 2

Circular Queue using arrays:

\*\*\*\*\*

1.Insertion

2.Deletion

3.Display

4.QUIT

Enter your choice:4

## UNIT – III LINKED LISTS

### Introduction to Linked List:

A **linked list** is a linear collection of data elements, called **nodes**, where the linear order is given by means of **pointers**. Each **node** is divided into two parts:

1. The first part contains the **information** of the element and
2. The second part contains the address of the next node (**link /next pointer field**) in the list.

The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

### Linked lists using dynamic variables:

1. In array implementation of the linked lists a fixed set of nodes represented by an array is established at the beginning of the execution
2. A pointer to a node is represented by the relative position of the node within the array.
3. In array implementation, it is not possible to determine the number of nodes required for the linked list. Therefore;
  - a. Less number of nodes can be allocated which means that the program will have overflow problem.
  - b. More number of nodes can be allocated which means that some amount of the memory storage will be wasted.
4. The solution to this problem is to allow nodes that are *dynamic*, rather than static.
5. When a node is required storage is reserved /allocated for it and when a node is no longer needed, the memory storage is released /freed.

### Allocating and freeing dynamic variables:

1. C library function **malloc()** is used for dynamically allocating a space to a pointer. Note that the **malloc()** is a library function in <stdlib.h> header file.
2. The following lines allocate an integer space from the memory pointed by the pointer p.

```
int *p;
```

```
p = (int *) malloc(sizeof(int));
```

Note that **sizeof()** is another library function that returns the number of bytes required for the operand. In this example, 4 bytes for the int.

### Advantages of linked lists:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of

a program.

2. Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.

#### **Disadvantages of linked lists:**

1. It consumes more space because every node requires an additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

#### **Types of Linked Lists:**

Basically we can put linked lists into the following four items:

1. Single Linked List.
2. Double Linked List.
3. Circular Linked List.
4. Circular Double Linked List.

A single linked list is one in which all nodes are linked together in some sequential manner. Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node.

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

#### **Comparison between array and linked list:**

<b>ARRAY</b>	<b>LINKED LIST</b>
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during	It is not necessary to specify the number of elements during declaration (i.e., memory is

compile time).	allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

### Applications of linked list:

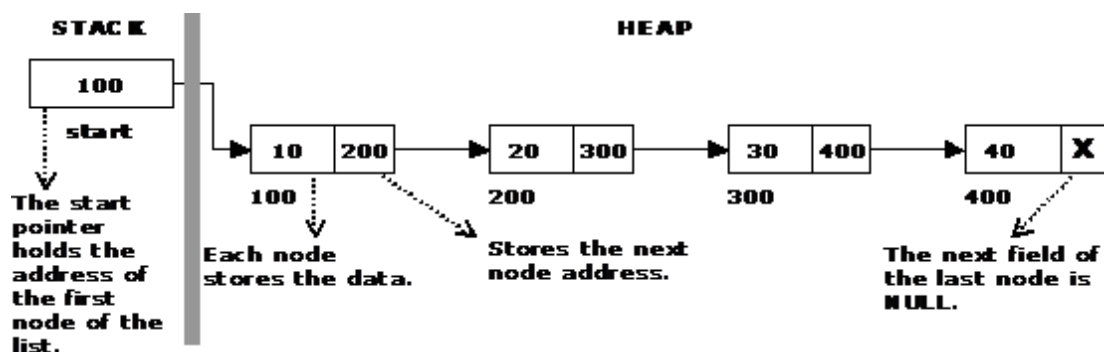
1. Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non zero coefficient and exponents. For example:  

$$P(x) = a_0 X^n + a_1 X^{n-1} + \dots + a_{n-1} X + a_n$$
2. Represent very large numbers and operations of the large number such as addition, multiplication and division.
3. Linked lists are to implement stack, queue, trees and graphs.
4. Implement the symbol table in compiler construction.

### Single Linked List:

The simplest kind of linked list is a singly-linked list, which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.

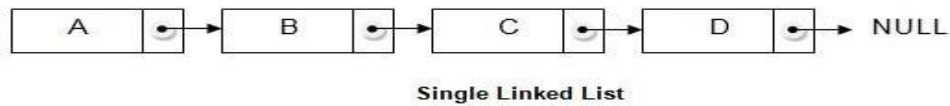
A singly linked list's node is divided into two parts. The first part holds or points to information about the node, and second part holds the address of next node. A singly linked list travels one way.



The beginning of the linked list is stored in a "**start**" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the **start** and following the next pointers.

The **start** pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that

it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.



### Implementation of Single Linked List:

1. Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.
2. Initialize the start pointer to be NULL.

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

typedef struct slinklist node;

node *start = NULL;
```

**node:**

data	next
------	------

**Empty list:**

<b>start</b> NULL
----------------------

### The basic operations in a single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

### Advantages of singly linked list:

1. Dynamic data structure.
2. We can perform deletion and insertion anywhere in the list.
3. We can merge two lists easily.

### Disadvantages of singly linked list:

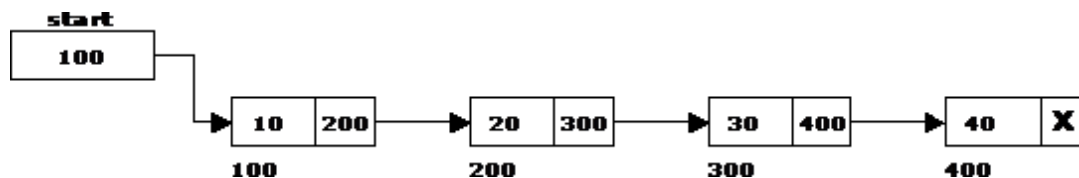
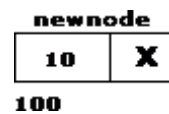
1. Backward traversing is not possible in singly linked list.

2. Insertion is easy but deletion take some additional time, because disadvantage of backward traversing.

### Creating a node for Single Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user, set next field to NULL and finally returns the address of the node.

```
node* getnode()
{
    node* newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode->data);
    newnode->next = NULL;
    return newnode;
}
```



### Insertion of a Node:

The new node can then be inserted at three different places namely:

1. Inserting a node at the beginning.
2. Inserting a node at the end.
3. Inserting a node at intermediate position.

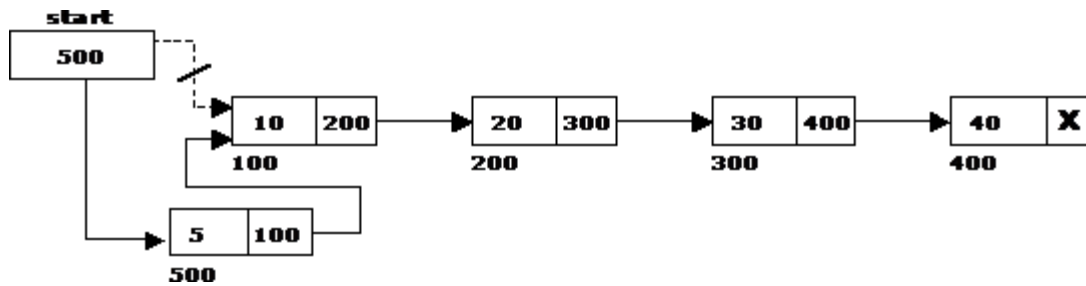
### Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

1. Get the new node using getnode().
2. If the list is empty then  $start = newnode$ .
3. If the list is not empty, follow the steps given below:

$newnode \rightarrow next = start;$

$start = newnode;$



### Inserting a node at the end:

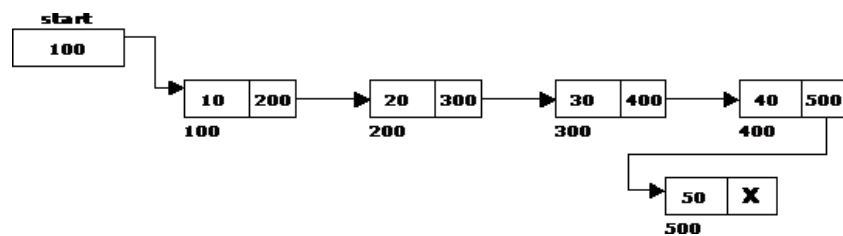
1. The following steps are followed to insert a new node at the end of the list:
2. Get the new node using `getnode()`
3. If the list is empty then  $start = newnode$ .
4. If the list is not empty follow the steps given below:

`temp = start;`

`while(temp -> next != NULL)`

1. `temp = temp -> next;`

`temp -> next = newnode;`



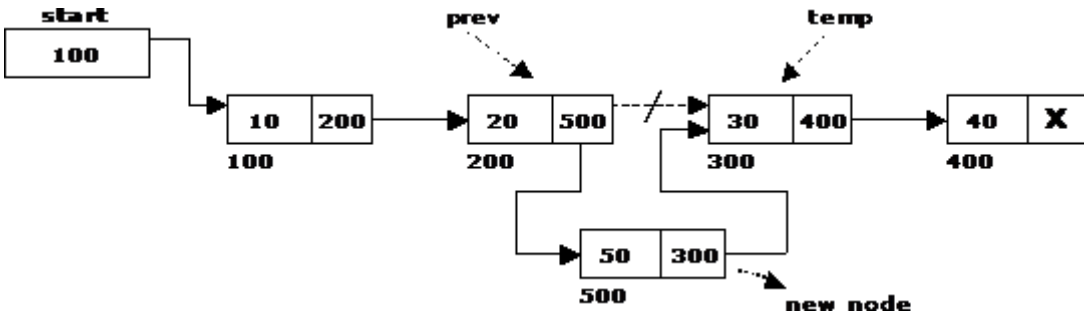
### Inserting a node at intermediate position:

1. The following steps are followed, to insert a new node in an intermediate position in the list:
2. Get the new node using `getnode()`.
3. Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by `countnode()` function.
4. Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
5. After reaching the specified position, follow the steps given below:



prev -> next = newnode;

newnode -> next = temp;



### Deletion of a node:

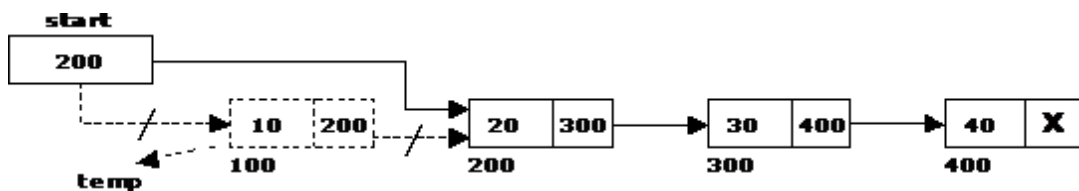
A node can be deleted from the list from three different places namely.

1. Deleting a node at the beginning.
2. Deleting a node at the end.
3. Deleting a node at intermediate position.

### Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:
  - i. temp = start;
  - ii. start = start -> next;
  - iii. free(temp);



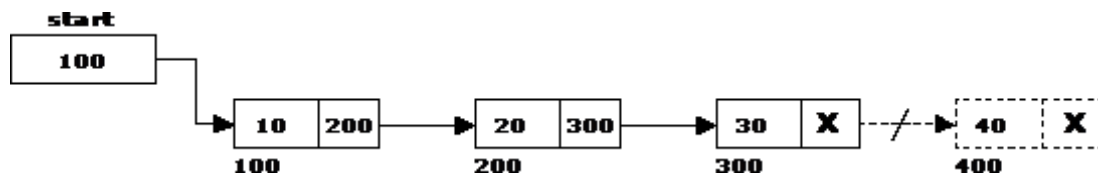
### Deleting a node at the end:

1. The following steps are followed to delete a node at the end of the list:
2. If list is empty then display 'Empty List' message.
3. If the list is not empty, follow the steps given below:

```

temp = prev = start;
while(temp -> next != NULL)
{
    1. prev = temp;
    2. temp = temp -> next;
}
prev -> next = NULL;
free(temp);

```



### Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

1. If list is empty then display 'Empty List' message
2. If the list is not empty, follow the steps given below.

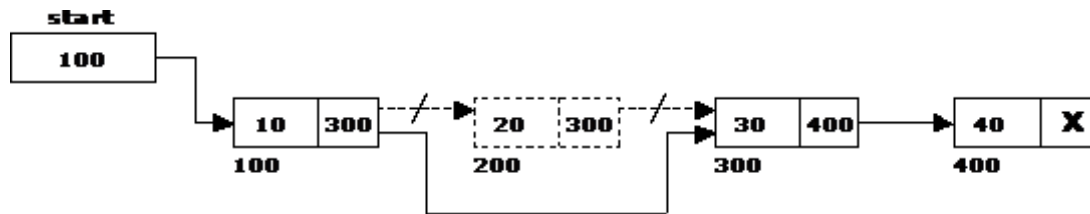
```

if(pos > 1 && pos < nodectr)
{
    temp = prev = start;
    ctr = 1;
    while(ctr < pos)
    {
        temp = temp -> next;
        ctr++;
    }
    prev -> next = temp -> next;
    free(temp);
}

```

```
printf("\n node deleted..");
```

```
}
```



### Traversal and displaying a list (Left to Right):

Traversing a list involves the following steps:

1. Assign the address of start pointer to a temp pointer.
2. Display the information from the data field of each node.

### Counting the Number of Nodes:

```
int countnode(node *st)
{
    if(st == NULL)
        return 0;
    else
        return(1 + countnode(st -> next));
}
```

### Source Code:

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

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

typedef struct linklist node;
node *start=NULL;
int menu()
```

```

{
    int ch;

    printf("\n\t *****IMPLIMENTATION OF SINGLE LINKED LIST*****");
    printf("\n\t -----\\n");
    printf("\n\t 1.Create list");
    printf("\n\t 2.Traverse the list(left to right)");
    printf("\n\t 3.Traverse the list(right to left)");
    printf("\n\t 4.Number of nodes");
    printf("\n\t 5.Insertion at Begining");
    printf("\n\t 6.Insertion at End");
    printf("\n\t 7.Insertion at Middle");
    printf("\n\t 8.Deletion at Beginning");
    printf("\n\t 9.Deletion at End");
    printf("\n\t 10.Deletion at Middle");
    printf("\n\t Enter your choice:");
    scanf("%d",&ch);
    return ch;
}

node* getnode()
{
    node *newnode;
    newnode=(node*)malloc(sizeof(node));
    printf("Enter data:\\n");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    return newnode;
}

int countnode(node*start)

```

```

{
    if(start==NULL)
        return 0;
    else
        return 1+countnode(start->next);
}

```

```

void createlist(int n)

```

```

{
    int i;
    node *newnode;
    node *temp;
    for(i=0;i<n;i++)
    {
        newnode=getnode();
        if(start==NULL)
        {
            start=newnode;
        }
        else
        {
            temp=start;
            while(temp->next!=NULL)
                temp=temp->next;
            temp->next=newnode;
        }
    }
}

```

```

void traverse()
{
    node *temp;

    temp=start;

    printf("The contents of the list(left to right)\n");
    if(start==NULL)
    {
        printf("\n Empty list");
        return;
    }
    else
    {
        while(temp!=NULL)
        {
            printf("%d\t",temp->data);
            temp=temp->next;
        }
        printf("X");
    }
}

void rev_traverse(node *start)
{
    if(start==NULL)
    {
        return;
    }
    else
    {

```

```

        rev_traverse(start->next);

        printf("%d\t",start->data);

    }

}

void insert_at_beg()
{
    node *newnode;

    newnode=getnode();

    if(start==NULL)
    {
        start=newnode;

    }

    newnode->next=start;

    start=newnode;

}

void insert_at_end()
{
    node *newnode,*temp;

    newnode=getnode();

    if(start==NULL)
    {
        start=newnode;

    }

    else

    {

        temp=start;

        while(temp->next!=NULL)

            temp=temp->next;
    }
}

```

```

        temp->next=newnode;

    }

}

void insert_at_mid()
{
    node *newnode,*pre,*temp;
    int pos,ctr=1,nodectr;
    printf("Enter position:");
    scanf("%d",&pos);
    nodectr=countnode(start);
    if(pos>1 && pos<nodectr)
    {

        newnode=getnode();
        temp=pre=start;
        while(ctr<pos)
        {
            pre=temp;
            temp=temp->next;
            ctr++;
        }
        pre->next=newnode;
        newnode->next=temp;

    }
    else
    {

```



```

        printf("\nNot a middle position");
    }
}

void del_at_beg()
{
    node *temp;
    if(start==NULL)
    {
        printf("List is empty");
        return;
    }
    else
    {
        temp=start;
        start=temp->next;
        free(temp);
        printf("Node is deleted");
    }
}

void del_at_end()
{
    node *pre,*temp;
    if(start==NULL)
    {
        printf("List is empty");
        return;
    }
}

```

```

else
{
    temp=start;
    while(temp->next!=NULL)
    {
        pre=temp;
        temp=temp->next;
    }
    pre->next=NULL;
    free(temp);
    printf("\Node deleted");
}
}

void del_at_mid()
{
    int pos,ctr=1,nodectr;
    node *temp,*pre;
    nodectr=countnode(start);
    if(start==NULL)
    {
        printf("List is empty");
        return;
    }
    else
    {
        printf("Enter position:");
        scanf("%d",&pos);
        if(pos>1 && pos<nodectr)

```

```

        {
            pre=temp=start;
            while(ctr<pos)
            {
                pre=temp;
                temp=temp->next;
                ctr++;
            }
            pre->next=temp->next;
            free(temp);
        }
    else
        printf("Not a mid position");
}

}

void main(void)
{
    int ch,n;
    clrscr();
    while(1)
    {
        ch=menu();
        switch(ch)
        {
            case 1:
                if(start==NULL)
                {
                    printf("Enter the number of nodes you want to create:");

```

```

        scanf("%d",&n);
        createlist(n);
        printf("List is created");
        break;
    }
    else
    {
        printf("List is already created:");
        break;
    }
case 2: traverse();
        break;
case 3:
        printf("The contents of the list(left to right):\n");
        rev_traverse(start);
        printf("X");
        break;
case 4: printf("Number of nodes:%d",countnode(start));
        break;
case 5: insert_at_beg();
        break;
case 6: insert_at_end();
        break;
case 7: insert_at_mid();
        break;
case 8: del_at_beg();
        break;
case 9: del_at_end();

```

```

                                break;
                    case 10:del_at_mid();
                                break;
                    case 11:exit(0);
                }
            }
        }
    }
}

```

### Output:

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Traverse the list(left to right)
- 3.Traverse the list(right to left)
- 4.Number of nodes
- 5.Insertion at Beginning
- 6.Insertion at End
- 7.Insertion at Middle
- 8.Deletion at Beginning
- 9.Deletion at End
- 10.Deletion at Middle

Enter your choice:1

Enter the number of nodes you want to create:5

Enter data:

1

Enter data:

2

Enter data:

3

Enter data:

4

Enter data:

5

List is created

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Traverse the list(left to right)
- 3.Traverse the list(right to left)
- 4.Number of nodes
- 5.Insertion at Beginning
- 6.Insertion at End
- 7.Insertion at Middle
- 8.Deletion at Beginning
- 9.Deletion at End
- 10.Deletion at Middle

Enter your choice:2

The contents of the list(left to right)

1    2    3    4    5    X

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Traverse the list(left to right)
- 3.Traverse the list(right to left)
- 4.Number of nodes
- 5.Insertion at Beginning
- 6.Insertion at End

- 7.Insertion at Middle
- 8.Deletion at Beginning
- 9.Deletion at End
- 10.Deletion at Middle

Enter your choice:3

The contents of the list(left to right):

5    4    3    2    1    X

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Traverse the list(left to right)
- 3.Traverse the list(right to left)
- 4.Number of nodes
- 5.Insertion at Beginning
- 6.Insertion at End
- 7.Insertion at Middle
- 8.Deletion at Beginning
- 9.Deletion at End
- 10.Deletion at Middle

Enter your choice:4

Number of nodes:5

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Traverse the list(left to right)
- 3.Traverse the list(right to left)
- 4.Number of nodes

- 5.Insertion at Beginning
- 6.Insertion at End
- 7.Insertion at Middle
- 8.Deletion at Beginning
- 9.Deletion at End
- 10.Deletion at Middle

Enter your choice:5

Enter data:

0

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Traverse the list(left to right)
- 3.Traverse the list(right to left)
- 4.Number of nodes
- 5.Insertion at Beginning
- 6.Insertion at End
- 7.Insertion at Middle
- 8.Deletion at Beginning
- 9.Deletion at End
- 10.Deletion at Middle

Enter your choice:2

The contents of the list(left to right)

0    1    2    3    4    5    X

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----



- 1.Create list
- 2.Traverse the list(left to right)
- 3.Traverse the list(right to left)
- 4.Number of nodes
- 5.Insertion at Beginning
- 6.Insertion at End
- 7.Insertion at Middle
- 8.Deletion at Beginning
- 9.Deletion at End
- 10.Deletion at Middle

Enter your choice:6

Enter data:

6

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Traverse the list(left to right)
- 3.Traverse the list(right to left)
- 4.Number of nodes
- 5.Insertion at Beginning
- 6.Insertion at End
- 7.Insertion at Middle
- 8.Deletion at Beginning
- 9.Deletion at End
- 10.Deletion at Middle

Enter your choice:2

The contents of the list(left to right)

0    1    2    3    4    5    6    X

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Traverse the list(left to right)
- 3.Traverse the list(right to left)
- 4.Number of nodes
- 5.Insertion at Beginning
- 6.Insertion at End
- 7.Insertion at Middle
- 8.Deletion at Beginning
- 9.Deletion at End
- 10.Deletion at Middle

Enter your choice:7

Enter position:4

Enter data:

7

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Traverse the list(left to right)
- 3.Traverse the list(right to left)
- 4.Number of nodes
- 5.Insertion at Beginning
- 6.Insertion at End
- 7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:2

The contents of the list(left to right)

0    1    2    7    3    4    5    6    X

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:8

Node is deleted

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

- 6.Insertion at End
- 7.Insertion at Middle
- 8.Deletion at Beginning
- 9.Deletion at End
- 10.Deletion at Middle

Enter your choice:2

The contents of the list(left to right)

1    2    7    3    4    5    6    X

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Traverse the list(left to right)
- 3.Traverse the list(right to left)
- 4.Number of nodes
- 5.Insertion at Beginning
- 6.Insertion at End
- 7.Insertion at Middle
- 8.Deletion at Beginning
- 9.Deletion at End
- 10.Deletion at Middle

Enter your choice:9

Node deleted

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Traverse the list(left to right)
- 3.Traverse the list(right to left)
- 4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:2

The contents of the list(left to right)

1    2    7    3    4    5    X

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

4.Number of nodes

5.Insertion at Beginning

6.Insertion at End

7.Insertion at Middle

8.Deletion at Beginning

9.Deletion at End

10.Deletion at Middle

Enter your choice:10

Enter position:3

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

1.Create list

2.Traverse the list(left to right)

3.Traverse the list(right to left)

- 4.Number of nodes
- 5.Insertion at Beginning
- 6.Insertion at End
- 7.Insertion at Middle
- 8.Deletion at Beginning
- 9.Deletion at End
- 10.Deletion at Middle

Enter your choice:2

The contents of the list(left to right)

1    2    3    4    5    X

\*\*\*\*\*IMPLIMENTATION OF LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Traverse the list(left to right)
- 3.Traverse the list(right to left)
- 4.Number of nodes
- 5.Insertion at Beginning
- 6.Insertion at End
- 7.Insertion at Middle
- 8.Deletion at Beginning
- 9.Deletion at End
- 10.Deletion at Middle

Enter your choice:1 1

**Merging two single linked lists into one list:**

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

```
#include<conio.h>
```

```
#include<stdlib.h>
```

```
struct slinkedlist
```

```

{
    int data;
    struct slinkedlist *next;
};

typedef struct slinkedlist node;
node *start=NULL;
int nodectr;
node *getnode()
{
    node *newnode;
    newnode=(node *)malloc(sizeof(node));
    printf("\n Enter data:");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    return newnode;
}
node * createlist(node * start,int n)
{
    node *temp,*newnode;
    int i;

    for(i=0;i<n;i++)
    {
        newnode=getnode();
        if(start==NULL)
        {
            start=newnode;
        }
    }
}

```

```

        else
        {
            temp=start;
            while(temp->next!=NULL)
            {
                temp=temp->next;
            }
            temp->next=newnode;
        }
    }
    return start;
}

void merge(node * start1,node *start2)
{
    node *temp;
    temp=start1;
    while(temp->next!=NULL)
        temp=temp->next;
    temp->next=start2;
    temp=start1;
    while(temp!=NULL)
    {
        printf("%5d",temp->data);
        temp=temp->next;
    }
}

int menu()
{

```



```

    int ch;

    printf("\nMerging two Single linked lists");

    printf("\n*****");

    printf("\n1.Create lists");

    printf("\n2.Mergelists");

    printf("\n3.Exit");

    printf("\n Enter your choice:");

    scanf("%d",&ch);

    return ch;

}

void main()

{

    int n,ch;

    node * start1=NULL;node * start2=NULL;

    do

    {

        ch=menu();

        switch (ch)

        {

            case 1:printf("Enter number of nodes you want:");

                    scanf("%d",&n);

                    start1=createlist(start,n);

                    printf("Enter number of nodes you want:");

                    scanf("%d",&n);

                    start2=createlist(start,n);

                    break;

            case 2:merge(start1,start2);

                    break;

```

```

        }
    }while(ch!=3);
    getch();
}

```

### Output:

Merging two Single linked lists

\*\*\*\*\*

1.Creat list 1

2.Create list 2

3.Mergelists

4.Exit

1. Enter your choice:1

Enter number of nodes you want:3

Enter data:1

Enter data:2

Enter data:3

2. Enter your choice:2

Enter number of nodes you want:

Enter data:4

Enter data:5

Enter data:6

Enter data:7

3. Enter your choice:3

1 2 3 4 5 6 7

Merging two Single linked lists

### Reversing a Single Linked List:

```

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

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

typedef struct node node;

node *start=NULL;

node * getnode()
{
    node * ptr;
    ptr=(node *)malloc(sizeof(node));
    printf("Enter data:\n");
    scanf("%d",&ptr->data);
    ptr->link=NULL;
    return ptr;
}

void create_ll(int N)
{
    node *ptr,*t;
    int i;
    printf("Enter the number of nodes you want:\n");
    scanf("%d",&N);
    for(i=0;i<N;i++)
    {
        ptr=getnode();
        if(start==NULL)

```

```

        {
            start=ptr;
        }
    else
    {
        t=start;
        while(t->link!=NULL)
        {
            t=t->link;
        }
        t->link=ptr;
    }
}

}

void reverse_ll(node *start)
{
    if(start==NULL)
    {
        return;
    }
    else
    {
        reverse_ll(start->link);
        printf("%d ",start->data);
    }
}

int menu()

```

```

{
    int ch;
    printf("1.Create list\n");
    printf("2.Reverse the list\n");
    printf("3.Exit\n");
    printf("Enter your choice:");
    scanf("%d",&ch);
    return ch;
}

void main()
{
    int n,ch;
    while(1)
    {
        ch=menu();
        switch(ch)
        {
            case 1:      create_ll(n);
                        printf("List created\n");
                        break;
            case 2:      printf("The list in reverse order:");
                        reverse_ll(start);
                        printf("\n");
                        break;
            case 3:      exit(0);
        }
    }
}

```

**Output:**

1.Create list

2.Reverse the list

3.Exit

Enter your choice:1

Enter the number of nodes you want:

3

Enter data:

12

Enter data:

56

Enter data:

33

List created

1.Create list

2.Reverse the list

3.Exit

Enter your choice:2

The list in reverse order:33 56 12

1.Create list

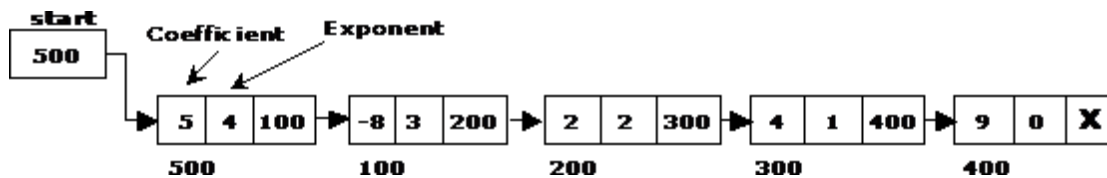
2.Reverse the list

3.Exit

Enter your choice:3

**Applications of Single Linked List to Represent Polynomial Expressions:**

The computer implementation requires implementing polynomials as a list of pairs of coefficient and exponent. Each of these pairs will constitute a structure, so a polynomial will be represented as a list of structures. A linked list structure that represents polynomials  $5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$  illustrates in figure ----



### Source Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<malloc.h>

struct link
{
    float coef;
    int expo;
    struct link *next;
};

typedef struct link node;

node *getnode()
{
    node *temp;
    temp=(node*)malloc(sizeof(node));
    printf("Enter coefficient:");
    fflush(stdin);
    scanf("%f",&temp->coef);
    printf("Enter exponent:");
    fflush(stdin);
    scanf("%d",&temp->expo);
    temp->next=NULL;
    return temp;
}
```

```

}
node *create_poly(node *p)
{
    char ch;
    node *temp,*newnode;
    while(1)
    {
        printf("Do you want polynomial node(y/n):\n");
        ch=getch();
        if(ch=='n')
            break;
        newnode=getnode();
        if(p==NULL)
            p=newnode;
        else
        {
            temp=p;
            while(temp->next!=NULL)
                temp=temp->next;
            temp->next=newnode;
        }
    }
    return p;
}

void display(node *p)
{
    node *t=p;
    while(t!=NULL)

```



```

        {
            printf("+%.f",t->coef);
            printf("x^%d",t->expo);
            t=t->next;
        }
    }
}

void main()
{
    node *poly1=NULL;
    Printf ("Enter First Polynomial..(in ascending order of exponent)\n");
    poly1=create_poly(poly1);
    printf("Polynomial 1:");
    display(poly1);
}

```

### Output:

Enter First Polynomial..(in ascending order of exponent)

Do you want polynomial node(y/n):Y

Enter coefficient:12

Enter exponent:3

Do you want polynomial node(y/n):Y

Enter coefficient:45

Enter exponent:4

Do you want polynomial node(y/n):N

Polynomial 1:+12x^3+45x^4

### Sparse Matrix Manipulation:

A **sparse matrix** is a matrix populated primarily with zeros as elements of the table.

#### Example of sparse matrix

```

[ 11 22 0 0 0 0 0 ]
[ 0 33 44 0 0 0 0 ]

```

```
[ 0 0 55 66 77 0 0 ]
[ 0 0 0 0 0 88 0 ]
[ 0 0 0 0 0 0 99 ]
```

The above sparse matrix contains only 9 nonzero elements of the 35, with 26 of those elements as zero. The basic data structure for a matrix is a two-dimensional array. Each entry in the array represents an element  $a_{ij}$  of the matrix and can be accessed by the two indices  $i$  and  $j$ . Traditionally,  $i$  indicates the row number (top-to-bottom), while  $j$  indicates the column number (left-to-right) of each element in the table. For an  $m \times n$  matrix, enough memory to store up to  $(m \times n)$  entries to represent the matrix is needed.

### Source Code:

```
#include<stdio.h>

void main()
{
    int matrix[20][20],m,n,total_elements,total_zeros=0,i,j;

    printf("Enter number of Rows and Columns:");

    scanf("%d %d",&m,&n);

    total_elements=m*n;

    printf("Enter data for Sparse matrix\n");

    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&matrix[i][j]);

            if(matrix[i][j]==0)
            {
                total_zeros++;
            }
        }
    }

    if(total_zeros>total_elements/2)
    {
        printf("Given Matrix is a Sparse matrix\n");
    }
}
```

```

printf("The Representation of Sparse matrix\n");
printf("Row\tColumn\tValue\n");
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        if(matrix[i][j]!=0)
        {
            printf(" %d \t %d \t %d\n",i,j,matrix[i][j]);
        }
    }
}
else
    Printf ("Given matrix is not a Sparse matrix...");
}

```

### Output:

Enter number of Rows and Columns:3

3

Enter data for Sparse matrix

2

0

0

0

0

3

0

10

0

Given Matrix is a Sparse matrix

The Representation of Sparse matrix

Row    Column   Value

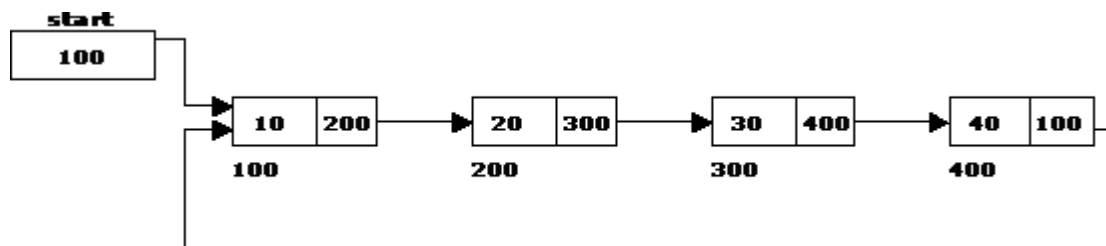
0      0      2

1      2      3

2 1 10

### Circular Linked List:

It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called *start* pointer always pointing to the first node of the list.



The basic operations in a circular single linked list are:

1. Creation.
2. Insertion.
3. Deletion.
4. Traversing.

### Creating a circular single Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

1. Get the new node using `getnode()`.  
`newnode = getnode();`
2. If the list is empty, assign new node as start.  
`start = newnode;`
3. If the list is not empty, follow the steps given below:  
`temp = start;`

```

while(temp -> next != NULL)

    temp = temp -> next;

temp -> next = newnode;

```

4. Repeat the above steps 'n' times.

```

newnode -> next = start;

```

### Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the circular list:

1. Get the new node using getnode().

```

newnode = getnode();

```

2. If the list is empty, assign new node as start.

```

start = newnode;

```

```

newnode -> next = start;

```

3. If the list is not empty, follow the steps given below:

```

last = start;

```

```

while(last -> next != start)

```

```

    last = last -> next;

```

```

newnode -> next = start;

```

```

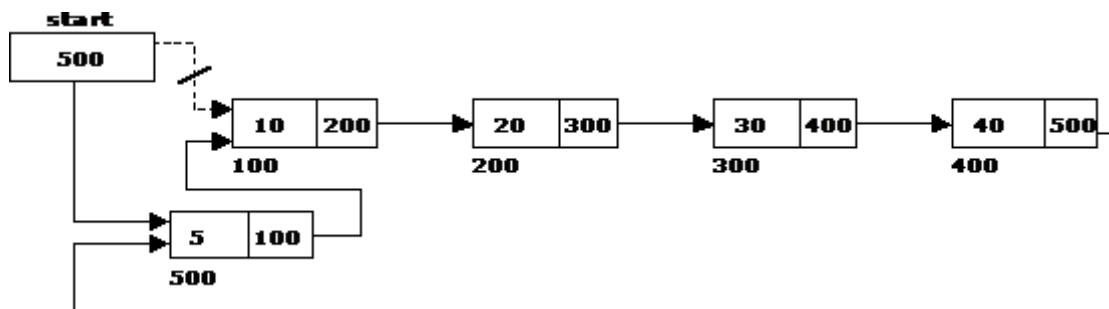
start = newnode;

```

```

last -> next = start;

```



### Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

1. Get the new node using getnode().

```
newnode = getnode();
```

2. If the list is empty, assign new node as start.

```
start = newnode;
```

```
newnode -> next = start;
```

3. If the list is not empty follow the steps given below:

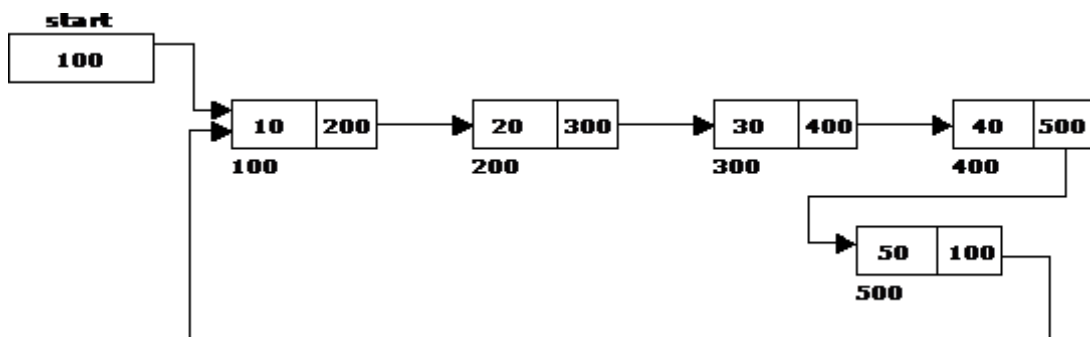
```
temp = start;
```

```
while(temp -> next != start)
```

```
temp = temp -> next;
```

```
temp -> next = newnode;
```

```
newnode -> next = start;
```



### Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If the list is empty, display a message 'Empty List'.
- If the list is not empty, follow the steps given below:

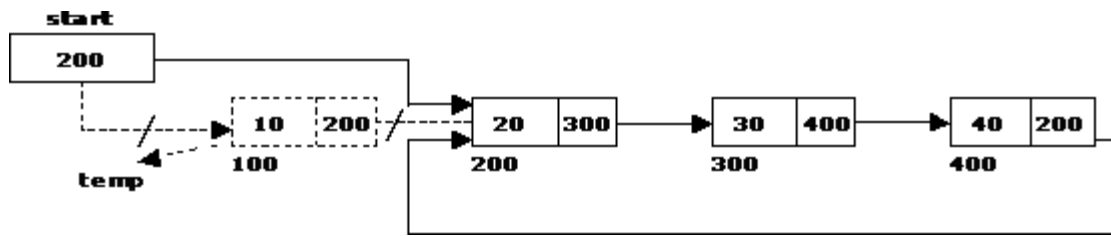
```
last = temp = start;
```

```
while(last -> next != start)
```

```
last = last -> next;
```

```
start = start -> next;
```

```
last -> next = start;
```



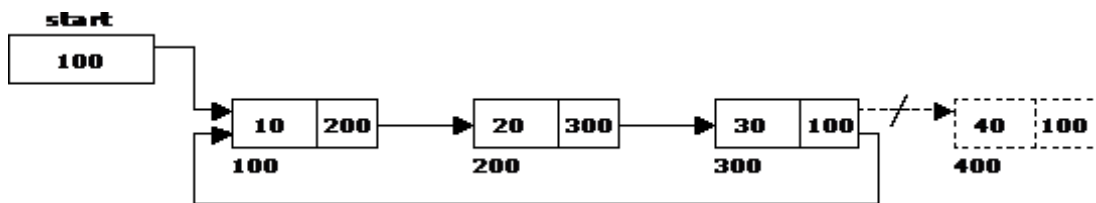
### Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

1. If the list is empty, display a message 'Empty List'.
2. If the list is not empty, follow the steps given below:

```
temp = start;
prev = start;
while(temp -> next != start)
{
    prev = temp;
    temp = temp -> next;
}
prev -> next = start;
```

3. After deleting the node, if the list is empty then start = NULL.



### Traversing a circular single linked list from left to right:

The following steps are followed, to traverse a list from left to right:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:

```
temp = start;
do
{
    temp = temp -> next;
} while(temp != start);
```

#### **Source Code:**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct clinklist
{
    int data;
    struct clinklist *next;
};
typedef struct clinklist node;
node *start=NULL;
int nodectr;
int menu()
{
    int ch;
    printf("\n\t *****IMPLIMENTATION OF CIRCULAR LINKED LIST*****");
    printf("\n\t -----\\n");
    printf("\n\t 1.Create list");
    printf("\n\t 2.Dispaly the contents");
```



```

    printf("\n\t 3.Number of nodes");
    printf("\n\t 4.Insertion at Begining");
    printf("\n\t 5.Insertion at End");
    printf("\n\t 6.Insertion at Middle");
    printf("\n\t 7.Deletion at Beginning");
    printf("\n\t 8.Deletion at End");
    printf("\n\t 9.Deletion at Middle");
    printf("\nEnter your choice:");
    scanf("%d",&ch);
    return ch;
}

node *getnode()
{
    node *newnode;
    newnode=(node *)malloc(sizeof(node));
    printf("Enter data:\n");
    scanf("%d",&newnode->data);
    newnode->next=NULL;
    return newnode;
}

void createlist(int n)
{
    int i;
    node *newnode;
    node *temp;
    nodectr=n;
    for(i=0;i<n;i++)
    {

```

```

        newnode=getnode();
        if(start==NULL)
        {
            start=newnode;
        }
        else
        {
            temp=start;
            while(temp->next!=NULL)
                temp=temp->next;
            temp->next=newnode;
        }
    }
    newnode->next=start;
}

void display()
{
    node *temp;
    temp=start;
    printf("The contents of the list:\n");
    if(start==NULL)
    {
        printf("\n Empty list");
        return;
    }
    else
    {
        do

```

```

        {

            printf("%d\t",temp->data);

            temp=temp->next;

        }while(temp!=start);

    }

    printf("X");
}

void insert_at_beg()
{
    node *newnode,*last;

    newnode=getnode();

    if(start==NULL)
    {
        start=newnode;
    }
    else
    {
        last=start;

        while(last->next!=start)

            last=last->next;

        newnode->next=start;

        start=newnode;

        last->next=start;
    }

    printf("\nNode is inserted ");

    nodectr++;
}

void insert_at_end()

```

```

{
    node *newnode,*temp;
    newnode=getnode();
    if(start==NULL)
    {
        start=newnode;
    }
    else
    {
        temp=start;
        while(temp->next!=start)
            temp=temp->next;
        temp->next=newnode;
        newnode->next=start;
    }
    printf("\nNode is inserted ");
    nodectr++;
}

void insert_at_mid()
{
    node *newnode,*pre,*temp;
    int pos,ctr=1;
    printf("Enter position:");
    scanf("%d",&pos);
    if(pos>1 && pos<nodectr)
    {

```

```

        newnode=getnode();
        temp=pre=start;
        while(ctr<pos)
        {
            pre=temp;
            temp=temp->next;
            ctr++;
        }
        pre->next=newnode;
        newnode->next=temp;
        printf("\nNode is inserted ");
        nodectr++;
    }
    else
    {
        printf("\nNot a middle position");
    }
}

void del_at_beg()
{
    node *temp,*last;
    if(start==NULL)
    {
        printf("List is empty");
        return;
    }
    else
    {

```

```

        last=temp=start;
        while(last->next!=start)
            last=last->next;

        start=temp->next;
        last->next=start;
        free(temp);
        nodectr--;
        printf("Node is deleted");
        if(nodectr==0)
            start=NULL;
    }
}

void del_at_end()
{
    node *temp,*pre;
    if(start==NULL)
    {
        printf("List is empty");
        return;
    }
    else
    {
        temp=start;
        pre=start;
        while(temp->next!=start)
        {
            pre=temp;
            temp=temp->next;

```

```

        }

        pre->next=NULL;

        free(temp);

        nodectr--;

        printf("Node is deleted");

        if(nodectr==0)

            start=NULL;

    }

}

void del_at_mid()

{

    int pos,ctr=0;

    node *temp,*pre;

    if(start==NULL)

    {

        printf("List is empty");

        return;

    }

    else

    {

        printf("Enter position:");

        scanf("%d",&pos);

        if(pos>1 && pos<nodectr)

        {

            pre=temp=start;

            while(ctr<pos-1)

            {

                pre=temp;

```

```

        temp=temp->next;

        ctr++;

    }

    pre->next=temp->next;

    free(temp);

    nodectr--;

    printf("\nNode deleted");

}

else

    printf("Not a mid position");

}

}

void main(void)

{

    int ch,n;

    clrscr();

    while(1)

    {

        ch=menu();

        switch(ch)

        {

            case 1:if(start==NULL)

                {

                    printf("Enter the number of nodes you want to create:");

                    scanf("%d",&n);

                    createlist(n);

                    printf("List is created");

                    break;

```



```

        }
        else

                printf("List is already created:");break;

        case 2: display();break;
        case 3: printf("Number of nodes:%d",nodectr);break;
        case 4: insert_at_beg();break;
        case 5: insert_at_end();break;
        case 6: insert_at_mid();break;
        case 7: del_at_beg();break;
        case 8: del_at_end();break;
        case 9: del_at_mid();break;
        case 10: exit(0);

    }
}
}

```

### Output:

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Dispaly the contents
- 3.Number of nodes
- 4.Insertion at Begining
- 5.Insertion at End
- 6.Insertion at Middle
- 7.Deletion at Beginning
- 8.Deletion at End

### 9.Deletion at Middle

Enter your choice:1

Enter the number of nodes you want to create:5

Enter data:

2

Enter data:

3

Enter data:

4

Enter data:

6

Enter data:

7

List is created

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:2

The contents of the list:

2    3    4    6    7    X

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Dispaly the contents
- 3.Number of nodes
- 4.Insertion at Begining
- 5.Insertion at End
- 6.Insertion at Middle
- 7.Deletion at Beginning
- 8.Deletion at End
- 9.Deletion at Middle

Enter your choice:3

Number of nodes:5

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Dispaly the contents
- 3.Number of nodes
- 4.Insertion at Begining
- 5.Insertion at End
- 6.Insertion at Middle
- 7.Deletion at Beginning
- 8.Deletion at End
- 9.Deletion at Middle

Enter your choice:4

Enter data:

1

Node is inserted

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Dispaly the contents
- 3.Number of nodes
- 4.Insertion at Begining
- 5.Insertion at End
- 6.Insertion at Middle
- 7.Deletion at Beginning
- 8.Deletion at End
- 9.Deletion at Middle

Enter your choice:2

The contents of the list:

1    2    3    4    6    7    X

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Dispaly the contents
- 3.Number of nodes
- 4.Insertion at Begining
- 5.Insertion at End
- 6.Insertion at Middle
- 7.Deletion at Beginning
- 8.Deletion at End
- 9.Deletion at Middle

Enter your choice:5

Enter data:

8

Node is inserted

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Dispaly the contents
- 3.Number of nodes
- 4.Insertion at Begining
- 5.Insertion at End
- 6.Insertion at Middle
- 7.Deletion at Beginning
- 8.Deletion at End
- 9.Deletion at Middle

Enter your choice:2

The contents of the list:

1    2    3    4    6    7    8    X

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Dispaly the contents
- 3.Number of nodes
- 4.Insertion at Begining
- 5.Insertion at End
- 6.Insertion at Middle
- 7.Deletion at Beginning
- 8.Deletion at End
- 9.Deletion at Middle

Enter your choice:6

Enter position:5

Enter data:

5

Node is inserted

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Dispaly the contents
- 3.Number of nodes
- 4.Insertion at Begining
- 5.Insertion at End
- 6.Insertion at Middle
- 7.Deletion at Beginning
- 8.Deletion at End
- 9.Deletion at Middle

Enter your choice:2

The contents of the list:

1    2    3    4    5    6    7    8    X

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Dispaly the contents
- 3.Number of nodes
- 4.Insertion at Begining
- 5.Insertion at End
- 6.Insertion at Middle
- 7.Deletion at Beginning
- 8.Deletion at End

9.Deletion at Middle

Enter your choice:7

Node is deleted

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:8

Node is deleted

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

1.Create list

2.Dispaly the contents

3.Number of nodes

4.Insertion at Begining

5.Insertion at End

6.Insertion at Middle

7.Deletion at Beginning

8.Deletion at End

9.Deletion at Middle

Enter your choice:9

Enter position:7

Not a mid position

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Dispaly the contents
- 3.Number of nodes
- 4.Insertion at Begining
- 5.Insertion at End
- 6.Insertion at Middle
- 7.Deletion at Beginning
- 8.Deletion at End
- 9.Deletion at Middle

Enter your choice:3

Number of nodes:6

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Dispaly the contents
- 3.Number of nodes
- 4.Insertion at Begining
- 5.Insertion at End
- 6.Insertion at Middle
- 7.Deletion at Beginning
- 8.Deletion at End
- 9.Deletion at Middle



Enter your choice:9

Enter position:3

Node deleted

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Dispaly the contents
- 3.Number of nodes
- 4.Insertion at Begining
- 5.Insertion at End
- 6.Insertion at Middle
- 7.Deletion at Beginning
- 8.Deletion at End
- 9.Deletion at Middle

Enter your choice:

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Dispaly the contents
- 3.Number of nodes
- 4.Insertion at Begining
- 5.Insertion at End
- 6.Insertion at Middle
- 7.Deletion at Beginning
- 8.Deletion at End
- 9.Deletion at Middle

Enter your choice:9

Enter position:7

Not a mid position

\*\*\*\*\*IMPLIMENTATION OF CIRCULAR LINKED LIST\*\*\*\*\*

-----

- 1.Create list
- 2.Dispaly the contents
- 3.Number of nodes
- 4.Insertion at Begining
- 5.Insertion at End
- 6.Insertion at Middle
- 7.Deletion at Beginning
- 8.Deletion at End
- 9.Deletion at Middle

Enter your choice:9

Enter position:3

Node deleted

### **Double Linked List:**

Doubly-linked list is a more sophisticated form of linked list data structure. Each node of the list contain two references (or links) – one to the previous node and other to the next node. The previous link of the first node and the next link of the last node points to NULL. In comparison to singly-linked list, doubly-linked list requires handling of more pointers but less information is required as one can use the previous links to observe the preceding element. It has a dynamic size, which can be determined only at run time.

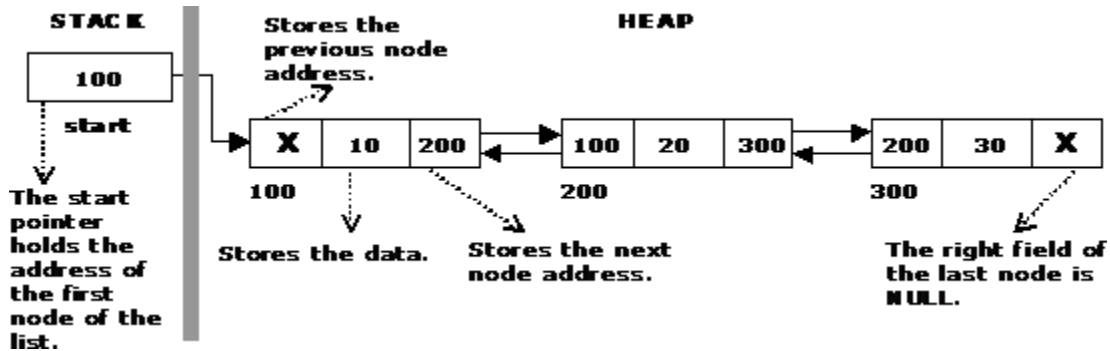
A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node contains three fields:

1. Left link.
2. Data.
3. Right link.

The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data. The basic operations in a double linked list are:

1. Creation.

2. Insertion.
3. Deletion.
4. Traversing.



The beginning of the double linked list is stored in a "start" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.

```
struct dlinklist
{
    struct dlinklist *left;
    int data;
    struct dlinklist *right;
};

typedef struct dlinklist node;
node *start = NULL;
```

**node:**

left	data	right
------	------	-------

**Empty list:**

start NULL
---------------

### Creating a node for Double Linked List:

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function.

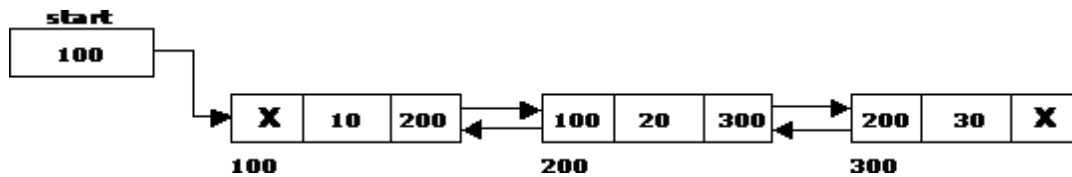
### Creating a Double Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

1. Get the new node using getnode().
 

newnode =getnode();
2. If the list is empty then  $start = newnode$ .

3. If the list is not empty, follow the steps given below:
  - i. The left field of the new node is made to point the previous node.
  - ii. The previous nodes right field must be assigned with address of the new node.
4. Repeat the above steps 'n' times.



### Inserting a node at the beginning:

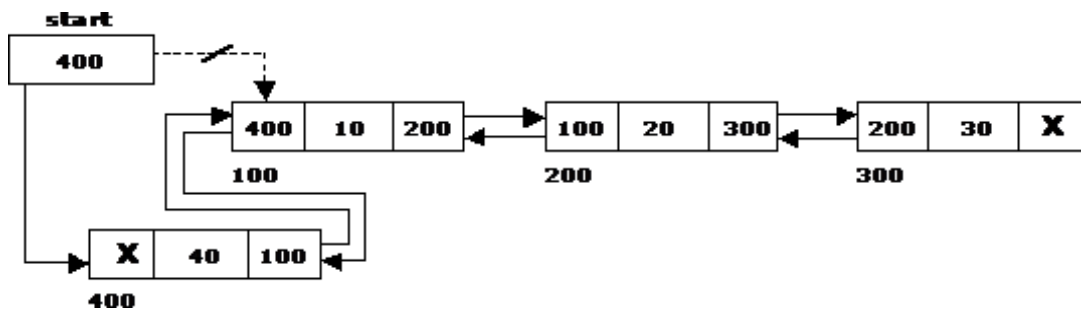
The following steps are to be followed to insert a new node at the beginning of the list:

1. Get the new node using `getnode()`.
2. If the list is empty then  $start = newnode$ .
3. If the list is not empty, follow the steps given below:

$newnode \rightarrow right = start;$

$start \rightarrow left = newnode;$

$start = newnode;$



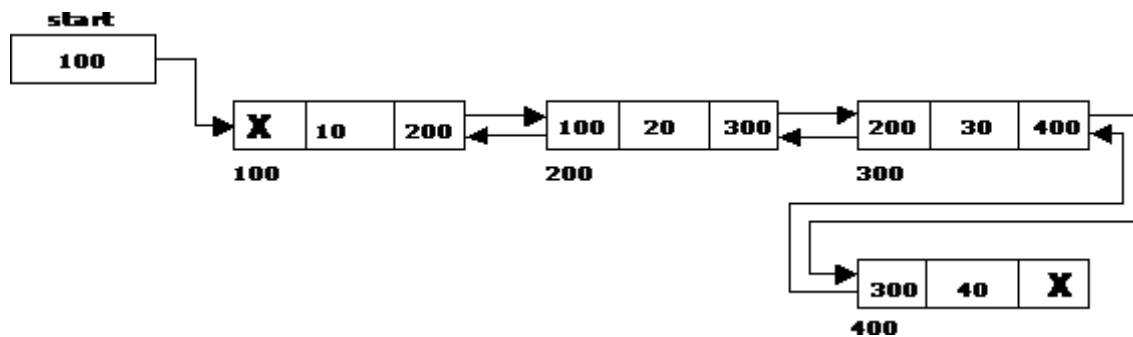
### Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

1. Get the new node using `getnode()`
2. If the list is empty then  $start = newnode$ .

- If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
    temp = temp -> right;
temp -> right = newnode;
newnode -> left = temp;
```



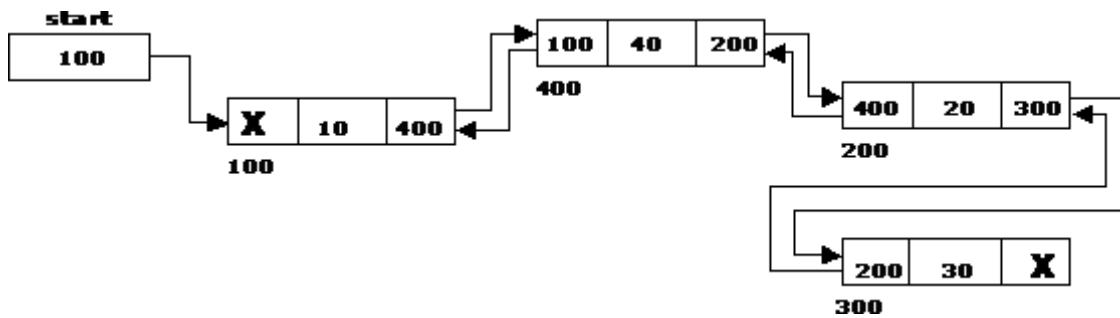
### Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using `getnode()`.  

```
newnode=getnode();
```
- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by `countnode()` function.
- Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
- After reaching the specified position, follow the steps given below:  

```
newnode -> left = temp;
newnode -> right = temp -> right;
temp -> right -> left = newnode;
temp -> right = newnode;
```

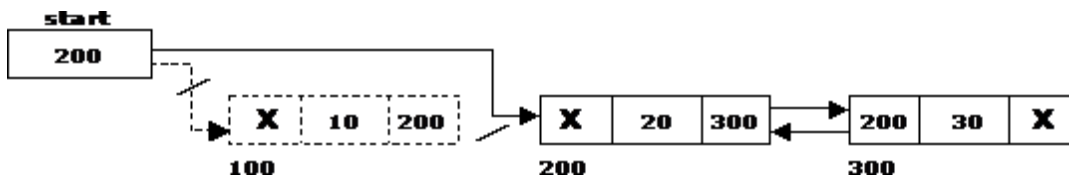


### Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> right;
start -> left = NULL;
free(temp);
```

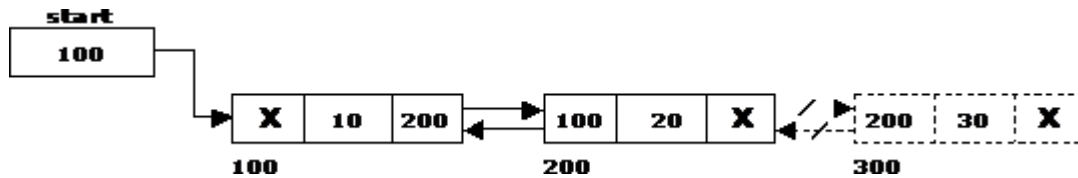


### Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

1. If list is empty then display 'Empty List' message
2. If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
{
    temp = temp -> right;
}
temp -> left -> right = NULL;
free(temp);
```



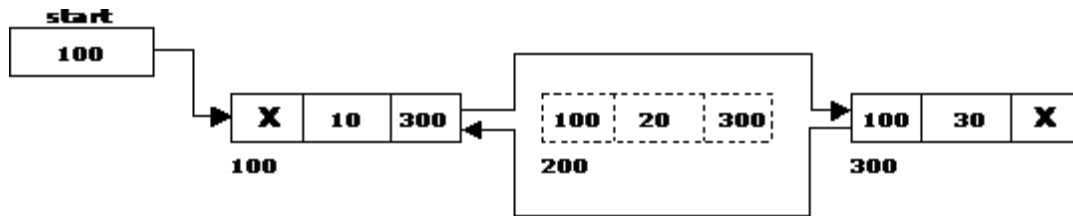
### Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two nodes).

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:
  - i. Get the position of the node to delete.
  - ii. Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
  - iii. Then perform the following steps:

```

if(pos > 1 && pos < nodectr)
{
    temp = start;
    i = 1;
    while(i < pos)
    {
        temp = temp -> right;
        i++;
    }
    temp -> right -> left = temp -> left;
    temp -> left -> right = temp -> right;
    free(temp);
    printf("\n node deleted..");
}
  
```



### Traversal and displaying a list (Left to Right):

The following steps are followed, to traverse a list from left to right:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:

```
temp = start;
while(temp != NULL)
{
    print temp -> data;
    temp = temp -> right;
}
```

### Traversal and displaying a list (Right to Left):

The following steps are followed, to traverse a list from right to left:

1. If list is empty then display 'Empty List' message.
2. If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
    temp = temp -> right;
while(temp != NULL)
{
    print temp -> data;
    temp = temp -> left;
}
```



### Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <conio.h>


struct dlinklist
{
    struct dlinklist *left;
    int data;
    struct dlinklist *right;
};

typedef struct dlinklist node;

node *start = NULL;

node* getnode()
{
    node * newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode -> data);
    newnode -> left = NULL;
    newnode -> right = NULL;
    return newnode;
}

int countnode(node *start)
{
    if(start == NULL)
        return 0;
    else
```

```

        return 1 + countnode(start -> right);
    }

int menu()
{
    int ch;

    clrscr();

    printf("\n 1.Create");
    printf("\n-----");

    printf("\n 2. Insert a node at beginning ");
    printf("\n 3. Insert a node at end");
    printf("\n 4. Insert a node at middle");
    printf("\n-----");

    printf("\n 5. Delete a node from beginning");
    printf("\n 6. Delete a node from Last");
    printf("\n 7. Delete a node from Middle");
    printf("\n-----");

    printf("\n 8. Traverse the list from Left to Right ");
    printf("\n 9. Traverse the list from Right to Left ");
    printf("\n-----");

    printf("\n 10.Count the Number of nodes in the list");
    printf("\n 11.Exit ");

    printf("\n\n Enter your choice: ");

    scanf("%d", &ch);

    return ch;
}

void createlist(int n)
{
    int i;

```

```

node *newnode;

node *temp;

for(i = 0; i < n; i++)
{
    newnode = getnode();
    if(start == NULL)
        start = newnode;
    else
    {
        temp = start;
        while(temp -> right)
            temp = temp -> right;
        temp -> right = newnode;
        newnode -> left = temp;
    }
}

void traverse_left_to_right()
{
    node *temp;
    temp = start;
    printf("\n The contents of List: ");
    if(start == NULL )
        printf("\n Empty List");
    else
        while(temp != NULL)
        {
            printf("\t %d ", temp -> data);

```

```

        temp = temp -> right;
    }
}

void traverse_right_to_left()
{
    node *temp;
    temp = start;
    printf("\n The contents of List: ");
    if(start == NULL)
        printf("\n Empty List");
    else
        while(temp -> right != NULL)
            temp = temp -> right;
    while(temp != NULL)
    {
        printf("\t%d", temp -> data);
        temp = temp -> left;
    }
}

void dll_insert_beg()
{
    node *newnode;
    newnode = getnode();
    if(start == NULL)
        start = newnode;
    else
    {
        newnode -> right = start;

```

```

        start -> left = newnode;

        start = newnode;

    }

}

void dll_insert_end()
{
    node *newnode, *temp;

    newnode = getnode();

    if(start == NULL)

        start = newnode;

    else

    {

        temp = start;

        while(temp -> right != NULL)

            temp = temp -> right;

        temp -> right = newnode;

        newnode -> left = temp;

    }

}

void dll_insert_mid()
{
    node *newnode, *temp;

    int pos, nodectr, ctr = 1;

    newnode = getnode();

    printf("\n Enter the position: ");

    scanf("%d", &pos);

    nodectr = countnode(start);

    if(pos - nodectr >= 2)

```

```

    {
        printf("\n Position is out of range..");
        return;
    }
    if(pos > 1 && pos < nodectr)
    {
        temp = start;
        while(ctr < pos - 1)
        {
            temp = temp -> right;
            ctr++;
        }
        newnode -> left = temp;
        newnode -> right = temp -> right;
        temp -> right -> left = newnode;
        temp -> right = newnode;
    }
    else
        printf("position %d of list is not a middle position ", pos);
}

void dll_delete_beg()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n Empty list");
        getch();
        return ;
    }

```

```

    }
else
{
    temp = start;
    start = start -> right;
    start -> left = NULL;
    free(temp);
}
}

void dll_delete_last()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n Empty list");
        getch();
        return ;
    }
else
{
    temp = start;
    while(temp -> right != NULL)
        temp = temp -> right;
    temp -> left -> right = NULL;
    free(temp);
    temp = NULL;
}
}

```

```

void dll_delete_mid()
{
    int i = 0, pos, nodectr;
    node *temp;
    if(start == NULL)
    {
        printf("\n Empty List");
        getch();
        return;
    }
    else
    {
        printf("\n Enter the position of the node to delete: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos > nodectr)
        {
            printf("\nthis node does not exist");
            getch();
            return;
        }
        if(pos > 1 && pos < nodectr)
        {
            temp = start;
            i = 1;
            while(i < pos)
            {
                temp = temp -> right;
            }
        }
    }
}

```



```

        i++;
    }
    temp -> right -> left = temp -> left;
    temp -> left -> right = temp -> right;
    free(temp);
    printf("\n node deleted..");
}
else
{
    printf("\n It is not a middle position..");
    getch();
}
}
}

void main(void)
{
    int ch, n;
    clrscr();
    while(1)
    {
        ch = menu();
        switch( ch)
        {
            case 1 : printf("\n Enter Number of nodes to create: ");
                    scanf("%d", &n);
                    createlist(n);
                    printf("\n List created..");
                    break;

```

```

        case 2 : dll_insert_beg();
                break;
        case 3 : dll_insert_end();
                break;
        case 4 : dll_insert_mid();
                break;
        case 5 : dll_delete_beg();
                break;
        case 6 : dll_delete_last();
                break;
        case 7 : dll_delete_mid();
                break;
        case 8 : traverse_left_to_right();
                break;
        case 9 : traverse_right_to_left();
                break;


        case 10 : printf("\n Number of nodes: %d", countnode(start));
                break;
        case 11: exit(0);
    }
    getch();
}
}

```



## UNIT – IV NON LINEAR DATA STRUCTURES

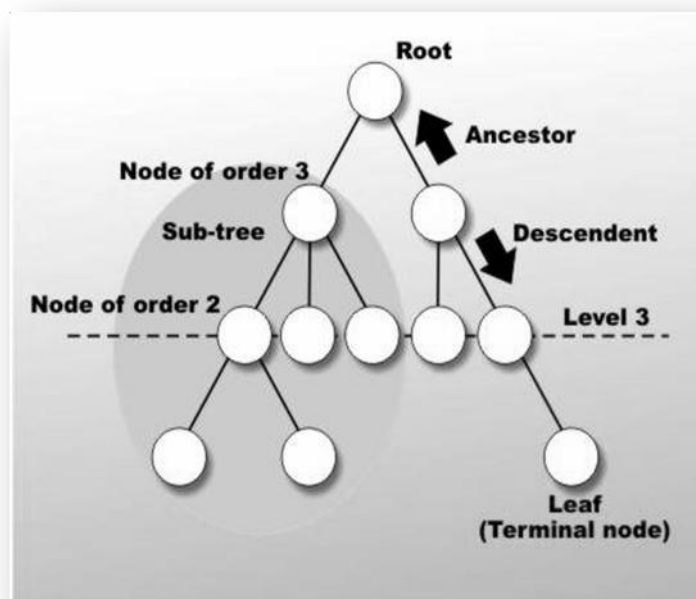
### Trees Basic Concepts:

A **tree** is a non-empty set one element of which is designated the root of the tree while the remaining elements are partitioned into non-empty sets each of which is a sub-tree of the root.

A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following

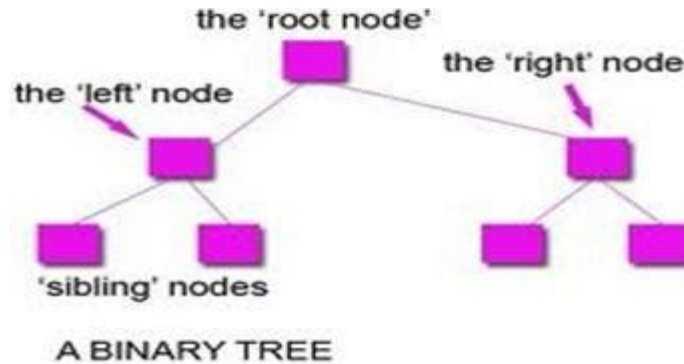
- If T is not empty, T has a special tree called the root that has no parent.
- Each node v of T different than the root has a unique parent node w; each node with parent w is a child of w.

Tree nodes have many useful properties. The **depth** of a node is the length of the path (or the number of edges) from the root to that node. The **height** of a node is the longest path from that node to its leaves. The height of a tree is the height of the root. A **leaf node** has no children -- its only path is up to its parent.



### Binary Tree:

In a binary tree, each node can have at most two children. A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**.



### Tree Terminology:

#### Leaf node

A node with no children is called a leaf (or external node). A node which is not a leaf is called an internal node.

**Path:** A sequence of nodes  $n_1, n_2, \dots, n_k$ , such that  $n_i$  is the parent of  $n_{i+1}$  for  $i = 1, 2, \dots, k - 1$ . The length of a path is 1 less than the number of nodes on the path. Thus there is a path of length zero from a node to itself.

**Siblings:** The children of the same parent are called siblings.

**Ancestor and Descendent** If there is a path from node A to node B, then A is called an ancestor of B and B is called a descendent of A.

**Subtree:** Any node of a tree, with all of its descendants is a subtree.

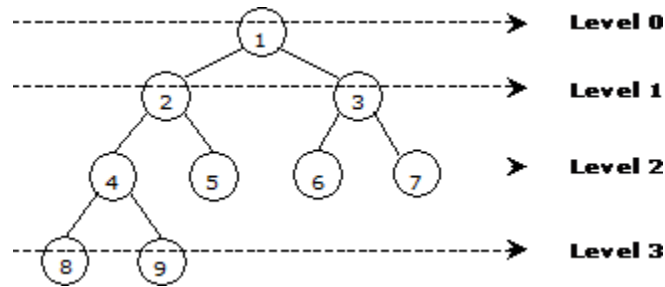
**Level:** The level of the node refers to its distance from the root. The root of the tree has level 0, and the level of any other node in the tree is one more than the level of its parent.

*The maximum number of nodes at any level is  $2^n$ .*

**Height:** The maximum level in a tree determines its height. The height of a node in a tree is the length of a longest path from the node to a leaf. The term depth is also used to denote height of the tree.

**Depth:** The depth of a node is the number of nodes along the path from the root to that node.

**Assigning level numbers and Numbering of nodes for a binary tree:** The nodes of a binary tree can be numbered in a natural way, level by level, left to right. The nodes of a complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent.



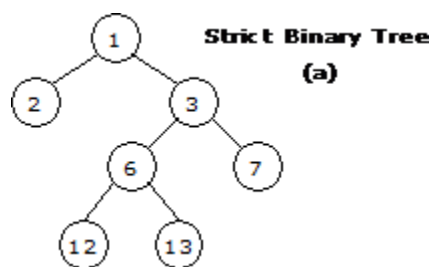
### Properties of Binary Trees:

Some of the important properties of a binary tree are as follows:

1. If  $h$  = height of a binary tree, then
  - a. Maximum number of leaves =  $2^h$
  - b. Maximum number of nodes =  $2^{h+1} - 1$
2. If a binary tree contains  $m$  nodes at level  $l$ , it contains at most  $2m$  nodes at level  $l + 1$ .
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most  $2^l$  node at level  $l$ .
4. The total number of edges in a full binary tree with  $n$  node is  $n - 1$ .

### Strictly Binary tree:

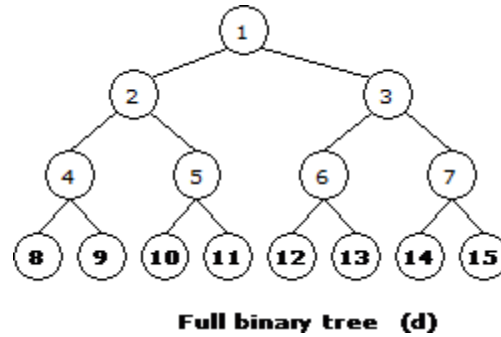
If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed a strictly binary tree. Thus the tree of figure 7.2.3(a) is strictly binary. A strictly binary tree with  $n$  leaves always contains  $2n - 1$  nodes.



### Full Binary Tree:

A full binary tree of height  $h$  has all its leaves at level  $h$ . Alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children.

A full binary tree with height  $h$  has  $2^{h+1} - 1$  nodes. A full binary tree of height  $h$  is a *strictly binary tree* all of whose leaves are at level  $h$ .

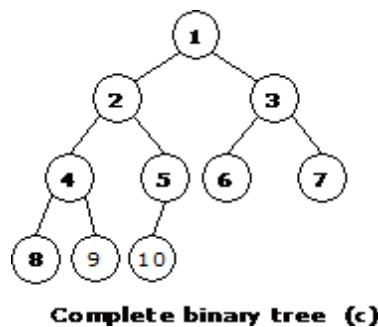


For example, a full binary tree of height 3 contains  $2^{3+1} - 1 = 15$  nodes.

### Complete Binary Tree:

A binary tree with  $n$  nodes is said to be **complete** if it contains all the first  $n$  nodes of the above numbering scheme.

A complete binary tree of height  $h$  looks like a full binary tree down to level  $h-1$ , and the level  $h$  is filled from left to right.



### Representation of Binary Trees:

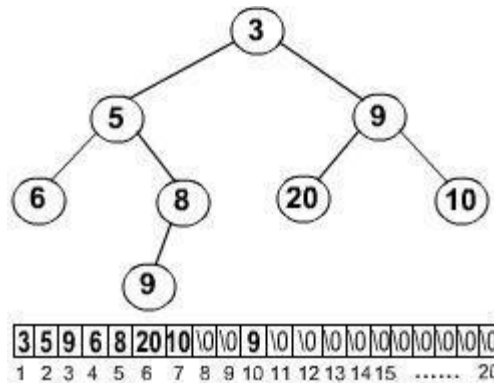
1. Array Representation of Binary Tree
2. Pointer-based.

### Array Representation of Binary Tree:

A single array can be used to represent a binary tree.

For these nodes are numbered / indexed according to a scheme giving 0 to root. Then all the nodes are numbered from left to right level by level from top to bottom. Empty nodes are also numbered. Then each node having an index  $i$  is put into the array as its  $i^{\text{th}}$  element.

In the figure shown below the nodes of binary tree are numbered according to the given scheme.



The figure shows how a binary tree is represented as an array. The root 3 is the 0<sup>th</sup> element while its leftchild 5 is the 1<sup>st</sup> element of the array. Node 6 does not have any child so its children i.e. 7<sup>th</sup> and 8<sup>th</sup> element of the array are shown as a Null value.

It is found that if  $n$  is the number or index of a node, then its left child occurs at  $(2n + 1)$ <sup>th</sup> position and right child at  $(2n + 2)$ <sup>th</sup> position of the array. If any node does not have any of its child, then null value is stored at the corresponding index of the array.

The following program implements the above binary tree in an array form. And then traverses the tree in inorder traversal.

```

struct node
{
    struct node * lc;
    int data;
    struct node * rc;
};
struct node * buildtree(int);/* builds the tree*/
void inorder(struct node *);/* Traverses the tree in inorder*/
int a[]={ 3,5,9,6,8,20,10,/0,/0,9,/0,/0,/0,/0,/0,/0,/0,/0,/0,/0};

void main( )
{
    struct node * root;
    root= buildtree(0);
    printf(-\n Inorder Traversal\|);
    inorder(root);
}

struct node * buildtree(int n);
{
    struct node * temp=NULL;
    if( a[n] != NULL)

```



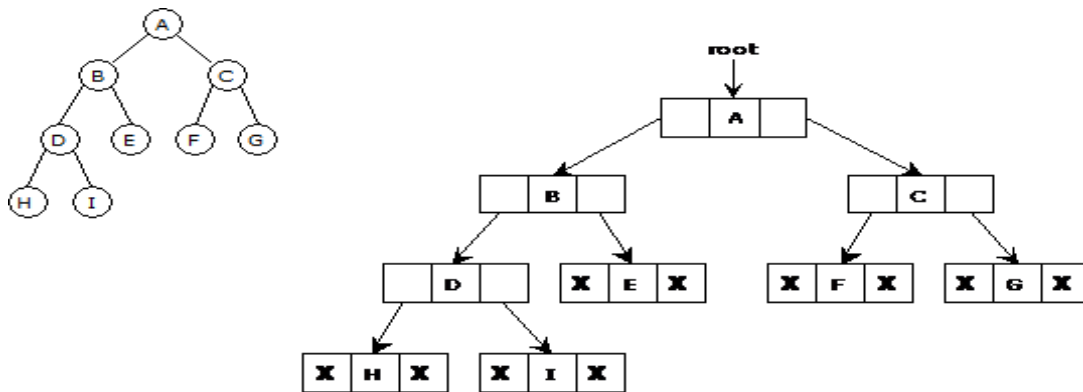
```

{
    temp = (struct node *) malloc(sizeof(struct node));
    temp->lc=buildtree(2n + 1);
    temp->data= a[n];
    temp->rc=buildtree(2n + 2);
}
return temp;
}

void inorder(struct node * root)
{
    if(root != NULL)
    {
        if(root!= NULL)
        {
            inorder(roo-> lc);
            printf(-%d\t\\,root->data);
            inorder(root->rc);
        }
    }
}

```

#### Linked Representation of Binary Tree (Pointer based):



Binary trees can be represented by links where each node contains the address of the left child and the right child. If any node has its left or right child empty then it will have in its respective link field, a null value. A leaf node has null value in both of its links.

The structure defining a node of binary tree in C is as follows.

```

struct node
{

```

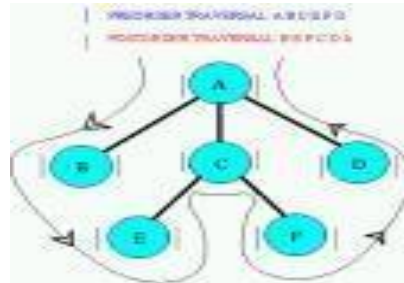
```

    struct node *lc ; /* points to the left child */
    int data; /* data field */
    struct node *rc; /* points to the right child */
}

```

### Binary Tree Traversals:

Traversal of a binary tree means to visit each node in the tree exactly once. The tree traversal is used in all t it.



In a linear list nodes are visited from first to last, but a tree being a non linear one we need definite rules. Th ways to traverse a tree. All of them differ only in the order in which they visit the nodes.

The three main methods of traversing a tree are:

- Inorder Traversal
- Preorder Traversal
- Postorder Traversal

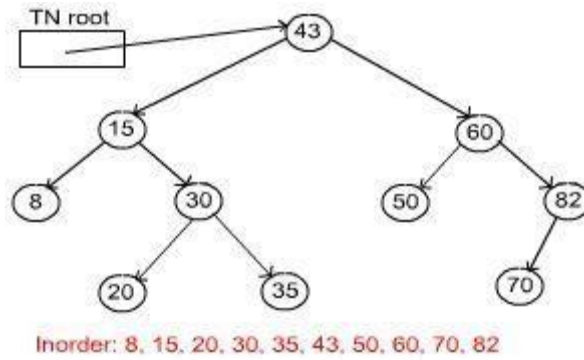
In all of them we do not require to do anything to traverse an empty tree. All the traversal methods are base functions since a binary tree is itself recursive as every child of a node in a binary tree is itself a binary tree.

### Inorder Traversal:

To traverse a non empty tree in inorder the following steps are followed recursively.

- Visit the Root
- Traverse the left subtree
- Traverse the right subtree

The inorder traversal of the tree shown below is as follows.



### Algorithm

The algorithm for inorder traversal is as follows.

```

struct node
{
    struct node * lc;
    int data;
    struct node * rc;
};

void inorder(struct node * root)
{
    if(root != NULL)
    {
        inorder(roo-> lc);
        printf("%d\t",root->data);
        inorder(root->rc);
    }
}
  
```

So the function calls itself recursively and carries on the traversal.

### Preorder Traversal:

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

### Postorder Traversal:

#### Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

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

```
#include <stdlib.h>
```

```
struct node
```

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

```
struct node* newNode(int data)
```

```
{  
    struct node* node = (struct node*)  
        malloc(sizeof(struct node));  
    node->data = data;  
    node->left = NULL;  
    node->right = NULL;  
    return(node);  
}
```

```
void printPostorder(struct node* node)
```

```
{  
    if (node == NULL)  
        return;
```

```

    printPostorder(node->left);
    printPostorder(node->right);
    printf("%d ", node->data);
}

void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

void printPreorder(struct node* node)
{
    if (node == NULL)
        return;

    printf("%d ", node->data);
    printPreorder(node->left);
    printPreorder(node->right);
}

int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);

```

```

    root->left->left = newNode(4);

    root->left->right = newNode(5);

    printf("\n Preorder traversal of binary tree is \n");
    printPreorder(root);

    printf("\n Inorder traversal of binary tree is \n");
    printInorder(root);

    printf("\n Postorder traversal of binary tree is \n");
    printPostorder(root);

    return 0;
}

```

```

/* program to construct tree using inorder and preorder traversals */

#include<stdio.h>

#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    char data;

    struct node* left;

    struct node* right;
};

int search(char arr[ ], int strt, int end, char value);

struct node* newNode(char data);

struct node* buildTree(char in[ ], char pre[ ], int inStrt, int inEnd)

```

```

{
    static int preIndex = 0;

    if(inStrt > inEnd)
        return NULL;
    struct node *tNode = newNode(pre[preIndex++]);
    if(inStrt == inEnd)
        return;
    int inIndex = search(in, inStrt, inEnd, tNode->data);
    tNode->left = buildTree(in, pre, inStrt, inIndex-1);
    tNode->right = buildTree(in, pre, inIndex+1, inEnd);
    return tNode;
}

int search(char arr[], int strt, int end, char value)
{
    int i;
    for(i = strt; i <= end; i++)
    {
        if(arr[i] == value)
            return i;
    }
}

struct node* newNode(char data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));

```

```

node->data = data;

node->left = NULL;

node->right = NULL;

return(node);
}

void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    printInorder(node->left);

    printf("%c ", node->data);

    printInorder(node->right);
}

int main()
{
    char in[] = {'D', 'B', 'E', 'A', 'F', 'C'};

    char pre[] = {'A', 'B', 'D', 'E', 'C', 'F'};

    int len = sizeof(in)/sizeof(in[0]);

    struct node *root = buildTree(in, pre, 0, len - 1);

    printf("\n Inorder traversal of the constructed tree is \n");

    printInorder(root);

    getchar();
}

```

Time Complexity:  $O(n^2)$ . Worst case occurs when tree is left skewed. Example Preorder and Inorder traversals for worst case are {A, B, C, D} and {D, C, B, A}.



## Threaded Binary Trees:

Inorder traversal of a Binary tree is either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

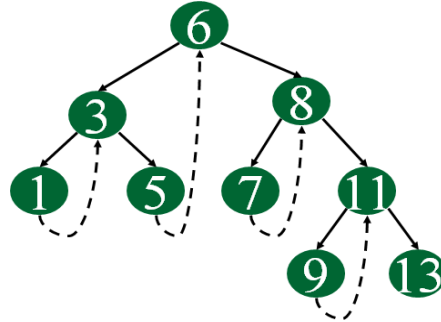
There are two types of threaded binary trees.

**Single Threaded:** Where a NULL right pointers is made to point to the inorder successor (if successor exists)

**Double Threaded:** Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



### C representation of a Threaded Node

Following is C representation of a single threaded node.

```
struct Node
{
    int data;

    Node *left, *right;

    bool rightThread;
}
```

Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

### Inorder Taversal using Threads

Following is C code for inorder traversal in a threaded binary tree.

```
// Utility function to find leftmost node in atree rooted with n
```

```
struct Node* leftMost(struct Node *n)
```

```
{
```

```
    if (n == NULL)
```

```
        return NULL;
```

```
    while (n->left != NULL)
```

```
        n = n->left;
```

```
    return n;
```

```
}
```

```
void inOrder(struct Node *root)
```

```
{
```

```
    struct Node *cur = leftmost(root);
```

```
    while (cur != NULL)
```

```
    {
```

```
        printf("%d ", cur->data);
```

```
        if (cur->rightThread)
```

```
            cur = cur->right;
```

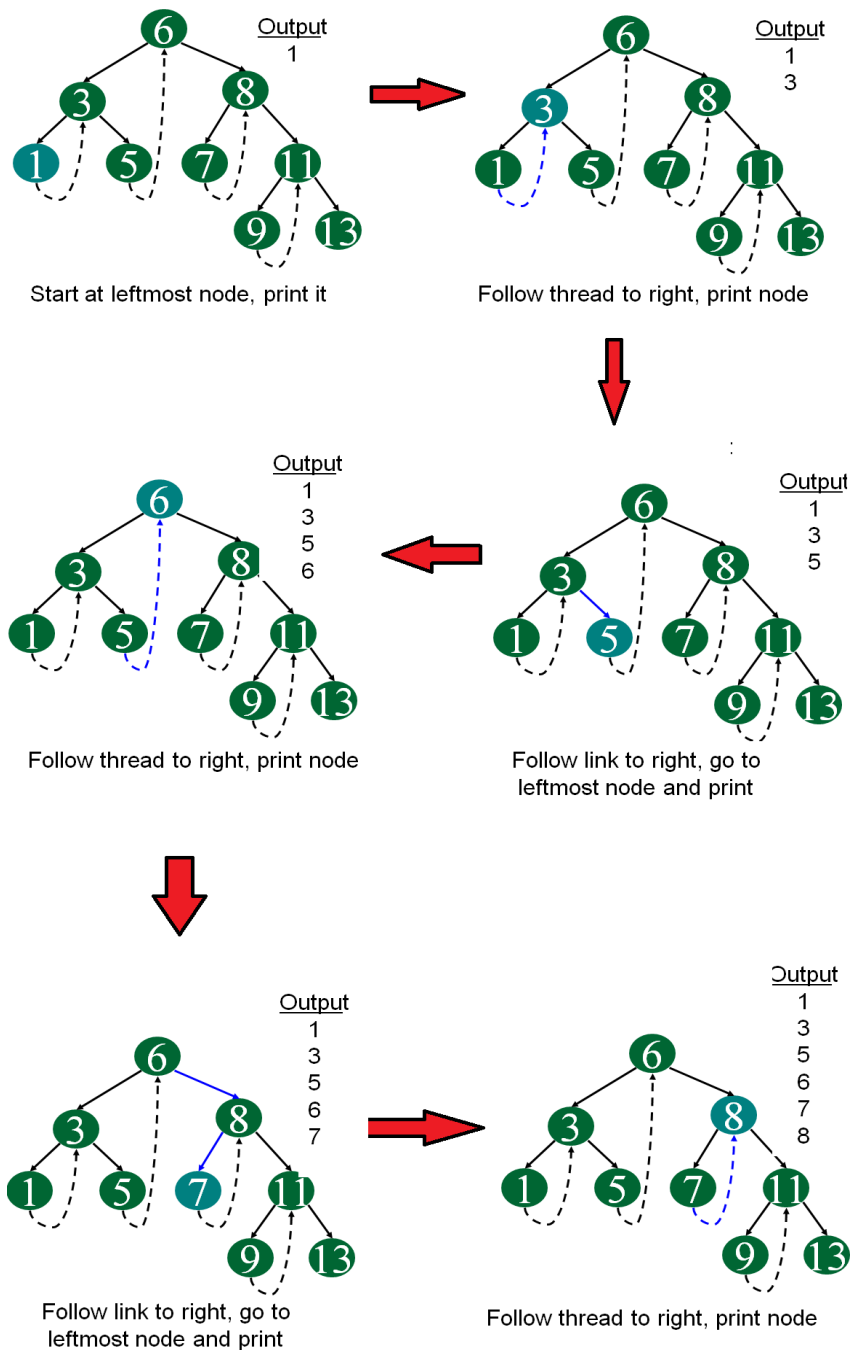
```
        else // Else go to the leftmost child in right subtree
```

```
            cur = leftmost(cur->right);
```

```
    }
```

```
}
```

Following diagram demonstrates inorder order traversal using threads.



**continue same way for remaining node.....**

## Priority Queues

Priority Queue is an extension of queue with following properties.

- 1) Every item has a priority associated with it.
- 2) An element with high priority is dequeued before an element with low priority.
- 3) If two elements have the same priority, they are served according to their order in the queue.

A typical priority queue supports following operations.

**insert(item, priority):** Inserts an item with given priority.

**getHighestPriority():** Returns the highest priority item.

**deleteHighestPriority():** Removes the highest priority item.

### Implementation priority queue

**Using Array:** A simple implementation is to use array of following structure.

```
struct item
{
    int item;
    int priority;
}
```

**insert()** operation can be implemented by adding an item at end of array in  $O(1)$  time.

**getHighestPriority()** operation can be implemented by linearly searching the highest priority item in array. This operation takes  $O(n)$  time.

**deleteHighestPriority()** operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.

We can also use Linked List, time complexity of all operations with linked list remains same as array. The advantage with linked list is **deleteHighestPriority()** can be more efficient as we don't have to move items.

### Applications of Priority Queue:

- 1) CPU Scheduling
- 2) Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
- 3) All queue applications where priority is involved.

### Application of Trees:

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system

```

/ <-- root
/  \
...  home
     /  \
    ugrad course
   /  /  |  \
... cs101 cs112 cs113
```

2. If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a

given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of  $O(\log_n)$  for search.

3) We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of  $O(\log_n)$  for insertion/deletion.

4) Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

The following are the common uses of tree.

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms

### **Basic Graph Concepts:**

Graph is a data structure that consists of following two components:

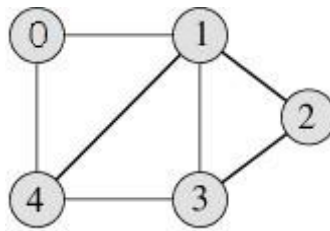
1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form  $(u, v)$  called as edge.

The pair is ordered because  $(u, v)$  is not same as  $(v, u)$  in case of directed graph (di-graph). The pair of form  $(u, v)$  indicates that there is an edge from vertex  $u$  to vertex  $v$ . The edges may contain weight/value/cost.

## Graph and its representations:

Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, facebook. For example, in facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale.

Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

### Adjacency Matrix:

Adjacency Matrix is a 2D array of size  $V \times V$  where  $V$  is the number of vertices in a graph. Let the 2D array be  $adj[][]$ , a slot  $adj[i][j] = 1$  indicates that there is an edge from vertex  $i$  to vertex  $j$ . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If  $adj[i][j] = w$ , then there is an edge from vertex  $i$  to vertex  $j$  with weight  $w$ .

The adjacency matrix for the above example graph is:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

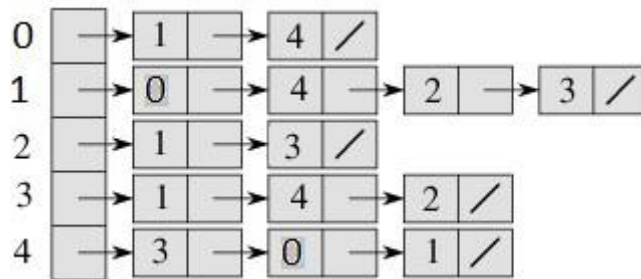
### Adjacency Matrix Representation of the above graph

*Pros:* Representation is easier to implement and follow. Removing an edge takes  $O(1)$  time. Queries like whether there is an edge from vertex  $u$  to vertex  $v$  are efficient and can be done  $O(1)$ .

*Cons:* Consumes more space  $O(V^2)$ . Even if the graph is sparse (contains less number of edges), it consumes the same space. Adding a vertex is  $O(V^2)$  time.

### Adjacency List:

An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be `array[]`. An entry `array[i]` represents the linked list of vertices adjacent to the  $i$ th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.

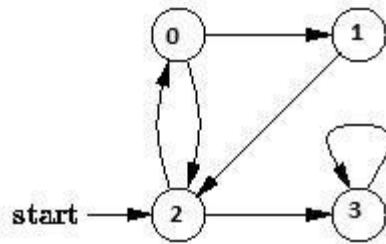


Adjacency List Representation of the above Graph

### Breadth First Traversal for a Graph

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. Breadth First Traversal of the following graph is 2, 0, 3, 1.



### Algorithm: Breadth-First Search Traversal

**BFS(V, E, s)**

```

for each  $u$  in  $V - \{s\}$ 
  do  $\text{color}[u] \leftarrow \text{WHITE}$ 
     $d[u] \leftarrow \text{infinity}$ 
     $\pi[u] \leftarrow \text{NIL}$ 
     $\text{color}[s] \leftarrow \text{GRAY}$ 

   $d[s] \leftarrow 0$ 
   $\pi[s] \leftarrow \text{NIL}$ 
   $Q \leftarrow \{\}$ 
   $\text{ENQUEUE}(Q, s)$ 
while  $Q$  is non-empty
  do  $u \leftarrow \text{DEQUEUE}(Q)$ 
    for each  $v$  adjacent to  $u$ 
      do if  $\text{color}[v] \leftarrow \text{WHITE}$ 
        then  $\text{color}[v] \leftarrow \text{GRAY}$ 
           $d[v] \leftarrow d[u] + 1$ 
           $\pi[v] \leftarrow u$ 
           $\text{ENQUEUE}(Q, v)$ 
     $\text{DEQUEUE}(Q)$ 
     $\text{color}[u] \leftarrow \text{BLACK}$ 
  
```



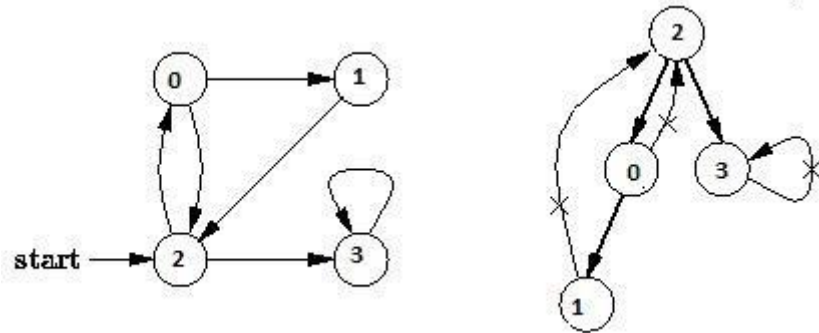
## Applications of Breadth First Traversal

- 1) Shortest Path and Minimum Spanning Tree for unweighted graph** In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.
- 2) Peer to Peer Networks.** In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.
- 3) Crawlers in Search Engines:** Crawlers build index using Bread First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.
- 4) Social Networking Websites:** In social networks, we can find people within a given distance  $k$  from a person using Breadth First Search till  $k$  levels.
- 5) GPS Navigation systems:** Breadth First Search is used to find all neighboring locations.
- 6) Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
- 7) In Garbage Collection:** Breadth First Search is used in copying garbage collection using Cheney's algorithm.
- 8) Cycle detection in undirected graph:** In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.
- 9) Ford–Fulkerson algorithm** In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to  $O(VE^2)$ .
- 10) To test if a graph is Bipartite** We can either use Breadth First or Depth First Traversal.
- 11) Path Finding** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
- 12) Finding all nodes within one connected component:** We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

## Depth First Traversal for a Graph

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. Depth First Traversal of the

following graph is 2, 0, 1, 3



### Algorithm Depth-First Search

The DFS forms a depth-first forest comprised of more than one depth-first trees. Each tree is made of edges  $(u, v)$  such that  $u$  is gray and  $v$  is white when edge  $(u, v)$  is explored. The following pseudocode for DFS uses a global timestamp time.

#### DFS (V, E)

```

for each vertex  $u$  in  $V[G]$ 
  do color[ $u$ ]  $\leftarrow$  WHITE
       $\pi[u] \leftarrow$  NIL
      time  $\leftarrow$  0
  for each vertex  $u$  in  $V[G]$ 
    do if color[ $u$ ]  $\leftarrow$  WHITE
      then DFS-Visit( $u$ )
  
```

#### DFS-Visit( $u$ )

```

  color[ $u$ ]  $\leftarrow$  GRAY
  time  $\leftarrow$  time + 1
  d[ $u$ ]  $\leftarrow$  time
  for each vertex  $v$  adjacent to  $u$ 
    do if color[ $v$ ]  $\leftarrow$  WHITE
      then  $\pi[v] \leftarrow u$ 
          DFS-Visit( $v$ )
  color[ $u$ ]  $\leftarrow$  BLACK
  time  $\leftarrow$  time + 1
  f[ $u$ ]  $\leftarrow$  time
  
```

### Applications of Depth First Search

Depth-first search (DFS) is an algorithm (or technique) for traversing a graph.

Following are the problems that use DFS as a building block.

1) For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

### 2) Detecting cycle in a graph

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See this for details)

### 3) Path Finding

We can specialize the DFS algorithm to find a path between two given vertices u and z.

- i) Call DFS(G, u) with u as the start vertex.
- ii) Use a stack S to keep track of the path between the start vertex and the current vertex.
- iii) As soon as destination vertex z is encountered, return the path as the contents of the stack

### 4) Topological Sorting

### 5) To test if a graph is bipartite

We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black! See this for details.

**6) Finding Strongly Connected Components of a graph** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See this for DFS based also for finding Strongly Connected Components)

C program to implement the Graph Traversal

- (a) Breadth first traversal
- (b) Depth first traversal

DFS search starts from root node then traversal into left child node and continues, if item found it stops otherwise it continues. The advantage of DFS is it requires less memory compare to Breadth First Search (BFS).

```
// DFS
```

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

```
int a[20][20],reach[20],n;
```

```
void dfs(int v)
```

```
{
    int i;
    reach[v]=1;
    for(i=1;i<=n;i++)
        if(a[v][i] && !reach[i])
        {
            printf("\n %d->%d",v,i);
            dfs(i);
        }
}
```

```

void main()
{
    int i,j,count=0;
    printf("\n Enter number of vertices:");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        reach[i]=0;
        for(j=1;j<=n;j++)
            a[i][j]=0;
    }
    printf("\n Enter the adjacency matrix:\n");
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            scanf("%d",&a[i][j]);
    dfs(1);
    printf("\n");
    for(i=1;i<=n;i++)
    {
        if(reach[i])
            count++;
    }
    if(count==n)
        printf("\n Graph is connected");
    else
        printf("\n Graph is not connected");
    }
}

```

BFS search starts from root node then traversal into next level of graph or tree and continues, if item found it stops otherwise it continues. The disadvantage of BFS is it requires more memory compare to Depth First Search (DFS).

//BFS

```

#include<stdio.h>
int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;
void bfs(int v)
{
    for(i=1;i<=n;i++)
        if(a[v][i] && !visited[i])
            q[++r]=i;
    if(f<=r)
    {
        visited[q[f]]=1;
        bfs(q[f++]);
    }
}
void main()
{

```

```

int v;
printf("\n Enter the number of vertices:");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
    q[i]=0;
    visited[i]=0;
}
printf("\n Enter graph data in matrix form:\n");
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        scanf("%d",&a[i][j]);
printf("\n Enter the starting vertex:");
scanf("%d",&v);
bfs(v);
printf("\n The node which are reachable are:\n");
for(i=1;i<=n;i++)
    if(visited[i])
        printf("%d\t",i);
    else
        printf("\n Bfs is not possible");
}

```

## UNIT – V BINARY TREES AND HASHING

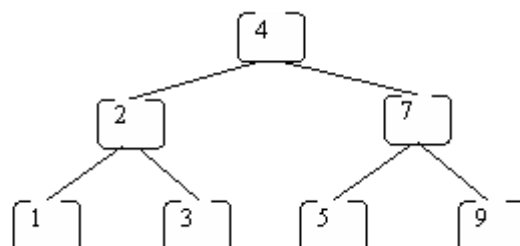
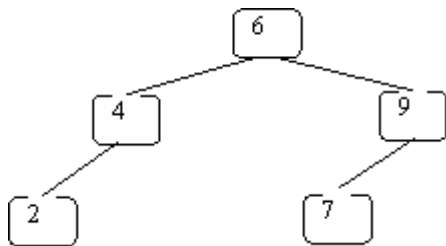
### Binary Search Trees:

An important special kind of binary tree is the **binary search tree (BST)**. In a BST, each node stores some information including a unique **key value**, and perhaps some associated data. A binary tree is a BST iff, for every node  $n$  in the tree:

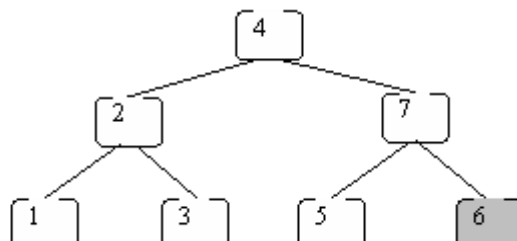
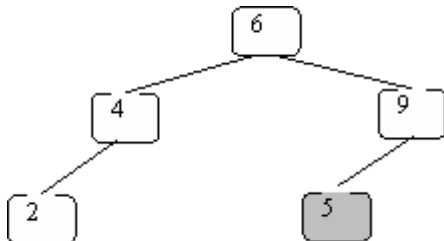
- All keys in  $n$ 's left subtree are less than the key in  $n$ , and
- All keys in  $n$ 's right subtree are greater than the key in  $n$ .

In other words, binary search trees are binary trees in which all values in the node's left subtree are less than node value all values in the node's right subtree are greater than node value.

Here are some BSTs in which each node just stores an integer key:



These are not BSTs:

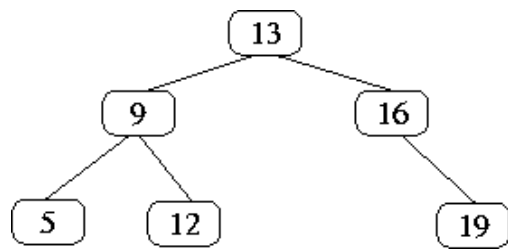


In the left one 5 is not greater than 6. In the right one 6 is not greater than 7.

The reason binary-search trees are important is that the following operations can be implemented efficiently using a BST:

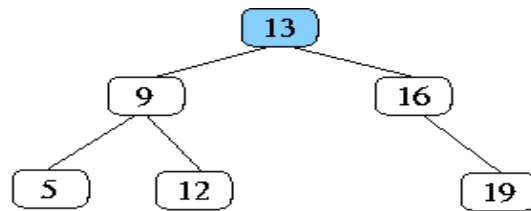
- insert a key value
- determine whether a key value is in the tree
- remove a key value from the tree
- print all of the key values in sorted order

Let's illustrate what happens using the following BST:

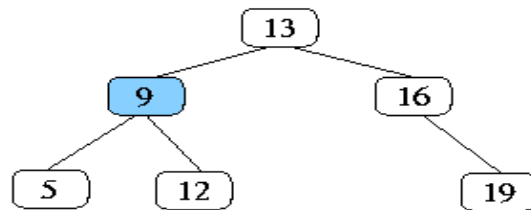


and searching for 12:

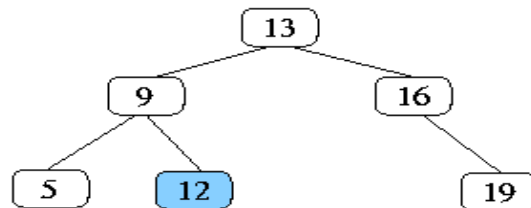
$12 < 13$  so go to  
left subtree



$12 > 9$  so go to  
right subtree.

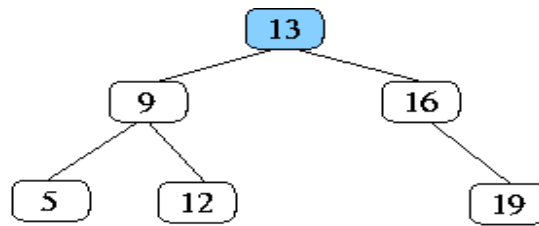


found!

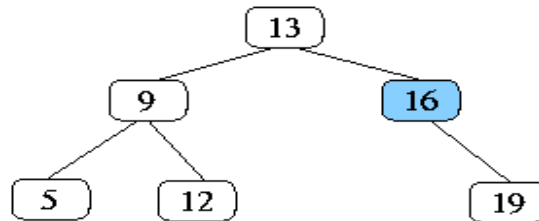


What if we search for 15:

15 > 13 so go to  
right subtree



15 < 16 so go to  
left subtree. It  
does not exist so  
search fails and it  
returns false



### Properties and Operations:

A BST is a binary tree of nodes ordered in the following way:

1. Each node contains one key (also unique)
2. The keys in the left subtree are < (less) than the key in its parent node
3. The keys in the right subtree > (greater) than the key in its parent node
4. Duplicate node keys are not allowed.

### Inserting a node

A naïve algorithm for inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root. We continue this process (each node is a root for some sub tree) until we find a null pointer (or leaf node) where we cannot go any further. We then insert the node as a left or right child of the leaf node based on node is less or greater than the leaf node. We note that a new node is always inserted as a leaf node. A recursive algorithm for inserting a node into a BST is as follows. Assume we insert a node N to tree T. if the tree is empty, then we return new node N as the tree. Otherwise, the problem of inserting is reduced to inserting the node N to left or right sub trees of T, depending on N is less or greater than T. A definition is as follows.

$\text{Insert}(N, T) = N$  if T is empty  
 $\quad = \text{insert}(N, T.\text{left})$  if  $N < T$   
 $\quad = \text{insert}(N, T.\text{right})$  if  $N > T$

### Searching for a node

Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node). A recursive definition of search is as follows. If the node is equal to root, then we return true. If the root is null, then we return false. Otherwise we recursively solve the problem for T.left or T.right, depending on  $N < T$  or  $N > T$ . A recursive definition is as follows.

Search should return a true or false, depending on the node is found or not.

$\text{Search}(N, T) = \text{false}$  if T is empty



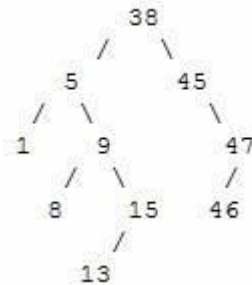
```

= true   if T = N
= search(N, T.left) if N < T
= search(N, T.right) if N > T

```

### Deleting a node

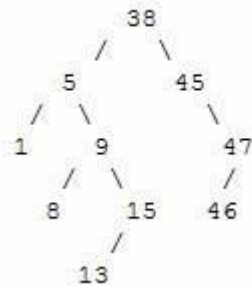
A BST is a connected structure. That is, all nodes in a tree are connected to some other node. For example, each node has a parent, unless node is the root. Therefore deleting a node could affect all sub trees of that node. For example, deleting node 5 from the tree could result in losing sub trees that are rooted at 1 and 9.



Hence we need to be careful about deleting nodes from a tree. The best way to deal with deletion seems to be considering special cases. What if the node to delete is a leaf node? What if the node is a node with just one child? What if the node is an internal node (with two children). The latter case is the hardest to resolve. But we will find a way to handle this situation as well.

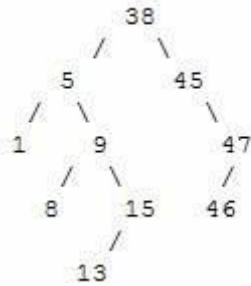
#### Case 1 : The node to delete is a leaf node

This is a very easy case. Just delete the node 46. We are done



#### Case 2 : The node to delete is a node with one child.

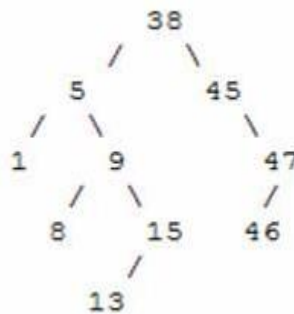
This is also not too bad. If the node to be deleted is a left child of the parent, then we connect the left pointer of the parent (of the deleted node) to the single child. Otherwise if the node to be deleted is a right child of the parent, then we connect the right pointer of the parent (of the deleted node) to single child.



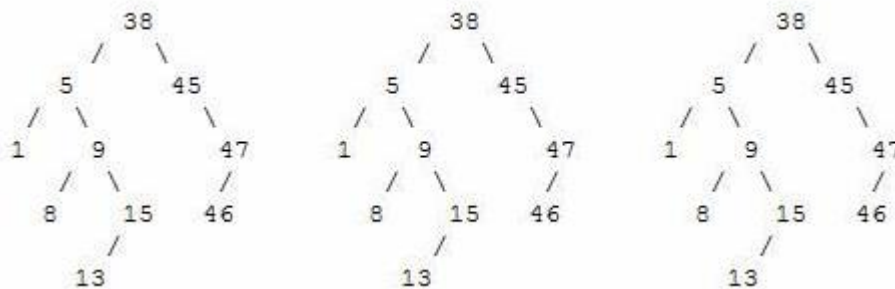
### Case 3: The node to delete is a node with two children

This is a difficult case as we need to deal with two sub trees. But we find an easy way to handle it. First we find a replacement node (from leaf node or nodes with one child) for the node to be deleted. We need to do this while maintaining the BST order property. Then we swap leaf node or node with one child with the node to be deleted (swap the data) and delete the leaf node or node with one child (case 1 or case 2)

Next problem is finding a replacement leaf node for the node to be deleted. We can easily find this as follows. If the node to be deleted is N, the find the largest node in the left sub tree of N or the smallest node in the right sub tree of N. These are two candidates that can replace the node to be deleted without losing the order property. For example, consider the following tree and suppose we need to delete the root 38.

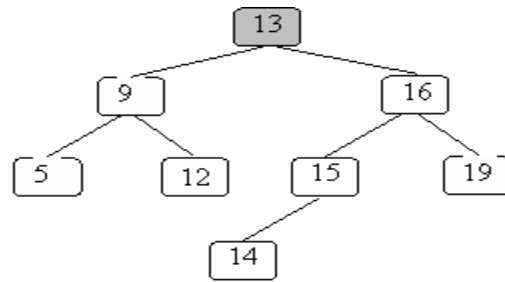


Then we find the largest node in the left sub tree (15) or smallest node in the right sub tree (45) and replace the root with that node and then delete that node. The following set of images demonstrates this process.

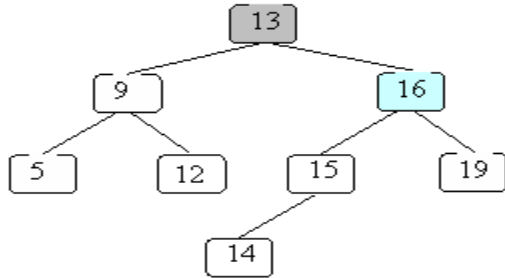


Let's see when we delete 13 from that tree.

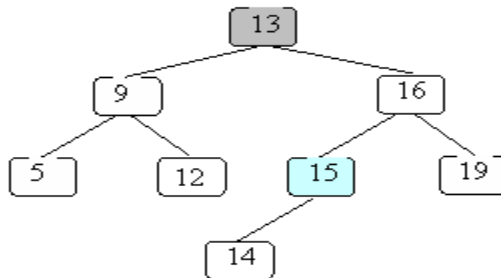
Original BST with  
13 located



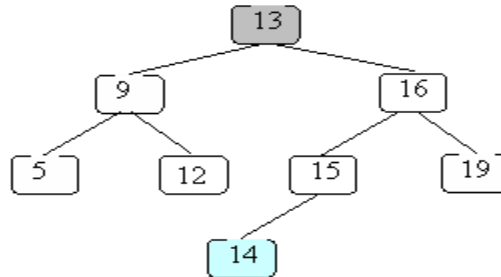
Step into right  
subtree.



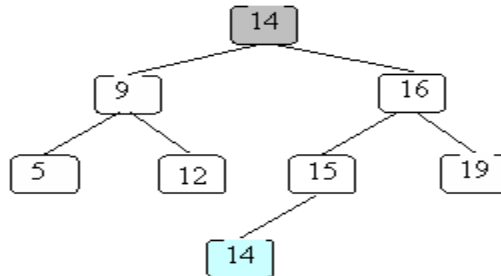
Go to left child.



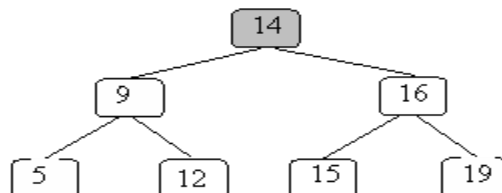
Continue to left  
child. This is last  
one.



Replace node to  
delete with far left  
child of right subtree.



Remove far left child  
of right subtree.



## Balanced Search Trees:

A **self-balancing** (or **height-balanced**) **binary search tree** is any node-based binary search tree that automatically keeps its height (maximal number of levels below the root) small in the face of arbitrary item insertions and deletions. The red–black tree, which is a type of self-balancing binary search tree, was called symmetric binary B-tree. Self-balancing binary search trees can be used in a natural way to construct and maintain ordered lists, such as priority queues. They can also be used for associative arrays; key-value pairs are simply inserted with an ordering based on the key alone. In this capacity, self-balancing BSTs have a number of advantages and disadvantages over their main competitor, hash tables. One advantage of self-balancing BSTs is that they allow fast (indeed, asymptotically optimal) enumeration of the items *in key order*, which hash tables do not provide. One disadvantage is that their lookup algorithms get more complicated when there may be multiple items with the same key. Self-balancing BSTs have better worst-case lookup performance than hash tables ( $O(\log n)$  compared to  $O(n)$ ), but have worse average-case performance ( $O(\log n)$  compared to  $O(1)$ ).

Self-balancing BSTs can be used to implement any algorithm that requires mutable ordered lists, to achieve optimal worst-case asymptotic performance. For example, if binary tree sort is implemented with a self-balanced BST, we have a very simple-to-describe yet asymptotically optimal  $O(n \log n)$  sorting algorithm. Similarly, many algorithms in computational geometry exploit variations on self-balancing BSTs to solve problems such as the line segment intersection problem and the point location problem efficiently. (For average-case performance, however, self-balanced BSTs may be less efficient than other solutions. Binary tree sort, in particular, is likely to be slower than merge sort, quicksort, or heapsort, because of the tree-balancing overhead as well as cache access patterns.)

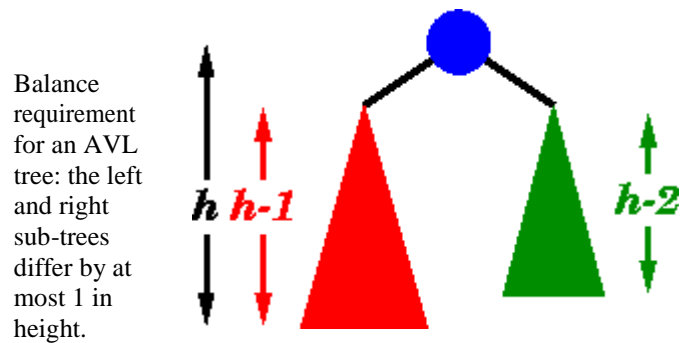
Self-balancing BSTs are flexible data structures, in that it's easy to extend them to efficiently record additional information or perform new operations. For example, one can record the number of nodes in each subtree having a certain property, allowing one to count the number of nodes in a certain key range with that property in  $O(\log n)$  time. These extensions can be used, for example, to optimize database queries or other list-processing algorithms.

### AVL Trees:

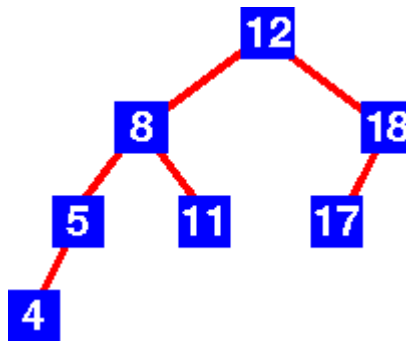
An **AVL tree** is another balanced binary search tree. Named after their inventors, **Adelson-Velskii** and **Landis**, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an  $O(\log n)$  search time. Addition and deletion operations also take  $O(\log n)$  time.

**Definition of an AVL tree:** An AVL tree is a binary search tree which has the following properties:

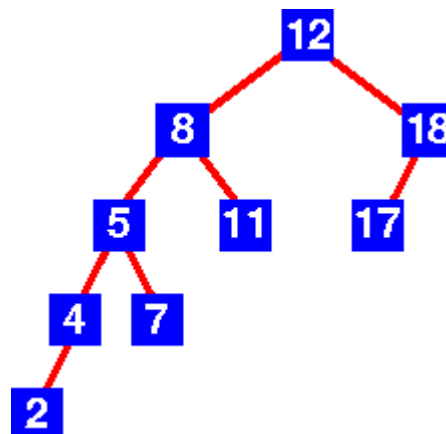
1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.



For example, here are some trees:



Yes this is an AVL tree. Examination shows that *each* left sub-tree has a height 1 greater than each right sub-tree.



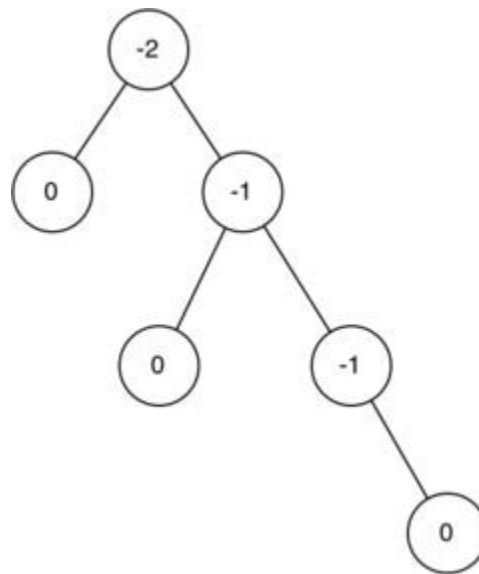
No this is not an AVL tree. Sub-tree with root 8 has height 4 and sub-tree with root 18 has height 2.

An AVL tree implements the Map abstract data type just like a regular binary search tree, the only difference is in how the tree performs. To implement our AVL tree we need to keep track of a **balance factor** for each node in the tree. We do this by looking at the heights of the left and

right subtrees for each node. More formally, we define the balance factor for a node as the difference between the height of the left subtree and the height of the right subtree.

$$\text{balanceFactor} = \text{height}(\text{leftSubTree}) - \text{height}(\text{rightSubTree})$$

Using the definition for balance factor given above we say that a subtree is left-heavy if the balance factor is greater than zero. If the balance factor is less than zero then the subtree is right heavy. If the balance factor is zero then the tree is perfectly in balance. For purposes of implementing an AVL tree, and gaining the benefit of having a balanced tree we will define a tree to be in balance if the balance factor is -1, 0, or 1. Once the balance factor of a node in a tree is outside this range we will need to have a procedure to bring the tree back into balance. Figure shows an example of an unbalanced, right-heavy tree and the balance factors of each node.



## Properties of AVL Trees

AVL trees are identical to standard binary search trees except that for every node in an AVL tree, the height of the left and right subtrees can differ by at most 1 (Weiss, 1993, p:108). AVL trees are HB-k trees (height balanced trees of order k) of order HB-1.

The following is the height differential formula:

$$|\text{Height}(T_L) - \text{Height}(T_R)| \leq k$$

When storing an AVL tree, a field must be added to each node with one of three values: 1, 0, or -1. A value of 1 in this field means that the left subtree has a height one more than the right subtree. A value of -1 denotes the opposite. A value of 0 indicates that the heights of both subtrees are the same. Updates of AVL trees require up to  $O(\log n)$  rotations, whereas updating red-black trees can be done using only one or two rotations (up to  $O(\log n)$  color changes). For this reason, they (AVL trees) are considered a bit obsolete by some.

## Sparse AVL trees

Sparse AVL trees are defined as AVL trees of height  $h$  with the fewest possible nodes. Figure 3 shows sparse AVL trees of heights 0, 1, 2, and 3.

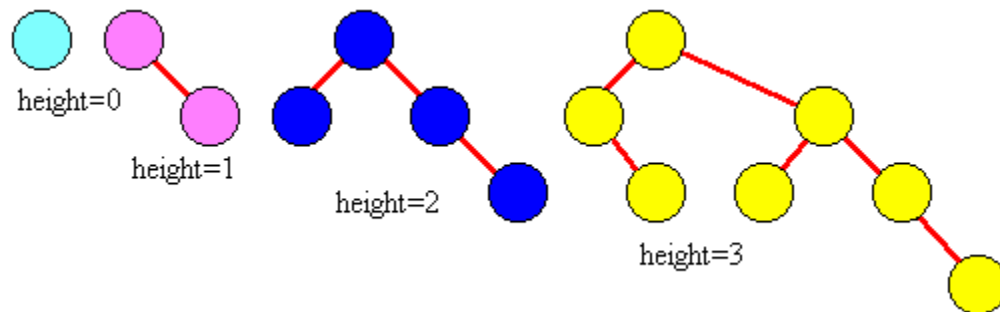


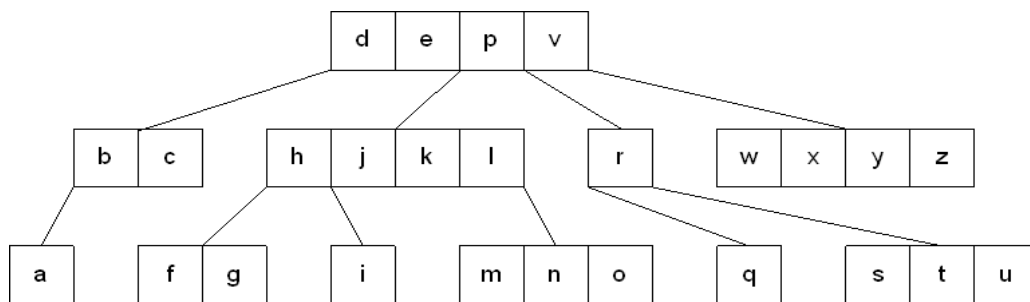
Figure Structure of an AVL tree

## Introduction to M-Way Search Trees:

A **multiway tree** is a tree that can have more than two children. A **multiway tree of order  $m$**  (or an  **$m$ -way tree**) is one in which a tree can have  $m$  children.

As with the other trees that have been studied, the nodes in an  $m$ -way tree will be made up of key fields, in this case  $m-1$  key fields, and pointers to children.

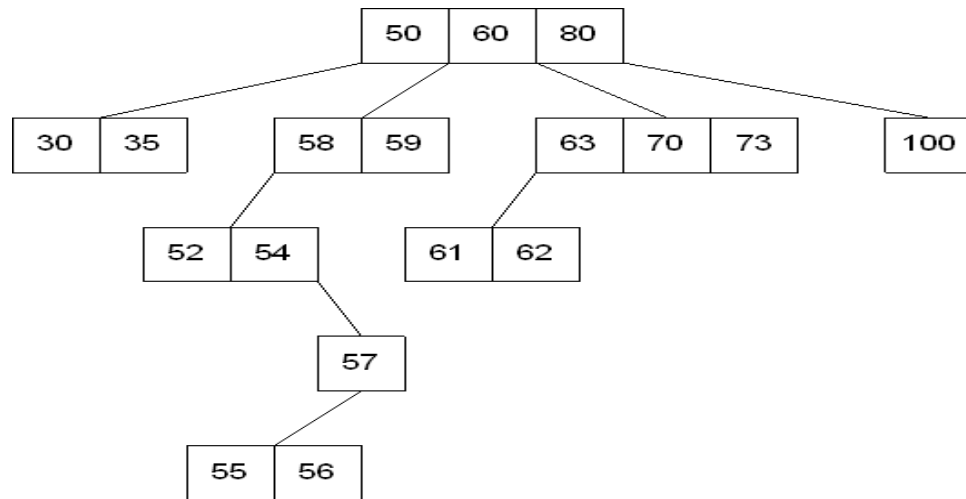
Multidway tree of order 5



To make the processing of  $m$ -way trees easier some type of order will be imposed on the keys within each node, resulting in a **multiway search tree of order  $m$**  (or an  **$m$ -way search tree**). By definition an  $m$ -way search tree is a  $m$ -way tree in which:

- Each node has  $m$  children and  $m-1$  key fields
- The keys in each node are in ascending order.
- The keys in the first  $i$  children are smaller than the  $i$ th key
- The keys in the last  $m-i$  children are larger than the  $i$ th key

#### 4-way search tree



M-way search trees give the same advantages to m-way trees that binary search trees gave to binary trees - they provide fast information retrieval and update. However, they also have the same problems that binary search trees had - they can become unbalanced, which means that the construction of the tree becomes of vital importance.

#### **B Trees:**

An extension of a multiway search tree of order  $m$  is a **B-tree of order  $m$** . This type of tree will be used when the data to be accessed/stored is located on secondary storage devices because they allow for large amounts of data to be stored in a node.

A B-tree of order  $m$  is a multiway search tree in which:

1. The root has at least two subtrees unless it is the only node in the tree.
2. Each nonroot and each nonleaf node have at most  $m$  nonempty children and at least  $m/2$  nonempty children.
3. The number of keys in each nonroot and each nonleaf node is one less than the number of its nonempty children.
4. All leaves are on the same level.

These restrictions make B-trees always at least half full, have few levels, and remain perfectly balanced.

#### **Searching a B-tree**

An algorithm for finding a key in B-tree is simple. Start at the root and determine which pointer to follow based on a comparison between the search value and key fields in the root node. Follow the appropriate pointer to a child node. Examine the key fields in the child node and continue to follow the appropriate pointers until the search value is found or a leaf node is reached that doesn't contain the desired search value.



## Insertion into a B-tree

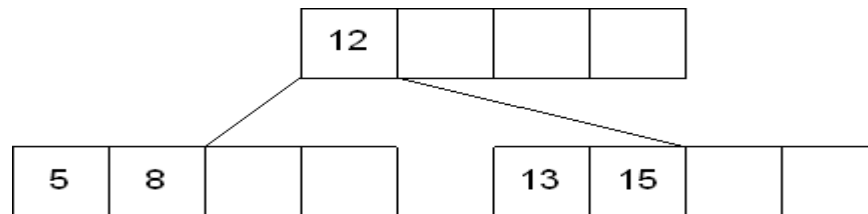
The condition that all leaves must be on the same level forces a characteristic behavior of B-trees, namely that B-trees are not allowed to grow at their leaves; instead they are forced to grow at the root.

When inserting into a B-tree, a value is inserted directly into a leaf. This leads to three common situations that can occur:

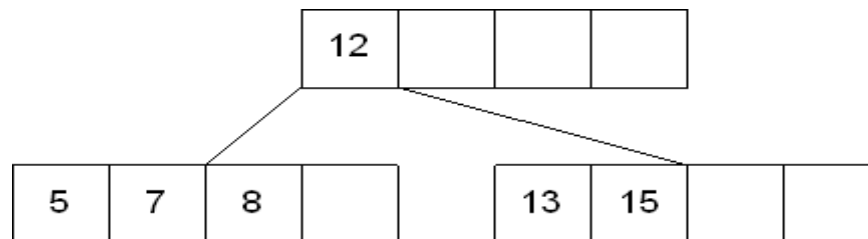
1. A key is placed into a leaf that still has room.
2. The leaf in which a key is to be placed is full.
3. The root of the B-tree is full.

### Case 1: A key is placed into a leaf that still has room

This is the easiest of the cases to solve because the value is simply inserted into the correct sorted position in the leaf node.



Inserting the number 7 results in:

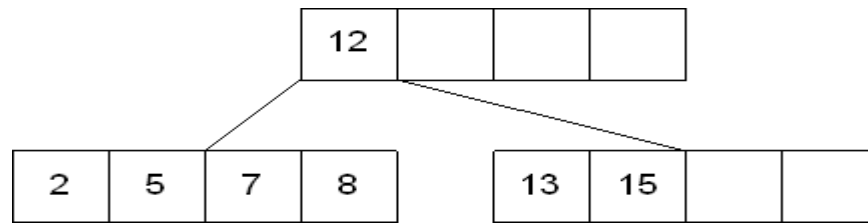


### Case 2: The leaf in which a key is to be placed is full

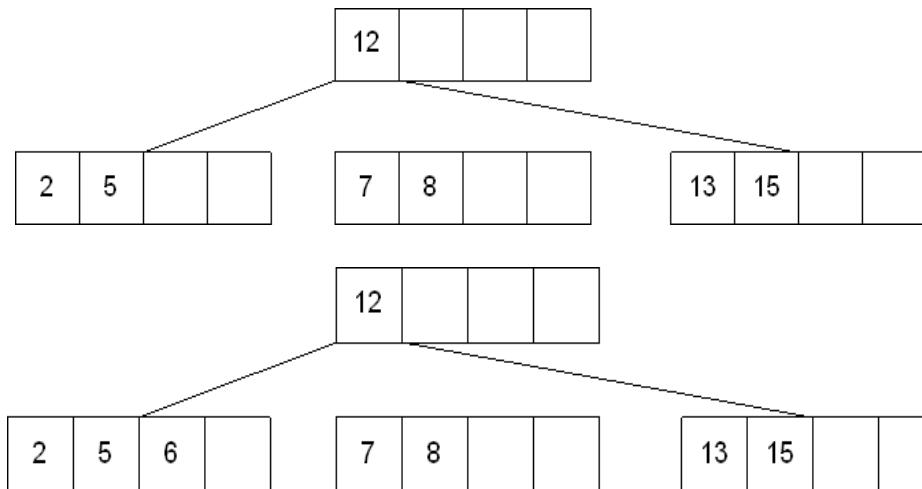
In this case, the leaf node where the value should be inserted is split in two, resulting in a new leaf node. Half of the keys will be moved from the full leaf to the new leaf. The new leaf is then incorporated into the B-tree.

The new leaf is incorporated by moving the middle value to the parent and a pointer to the new leaf is also added to the parent. This process continues up the tree until all of the values have "found" a location.

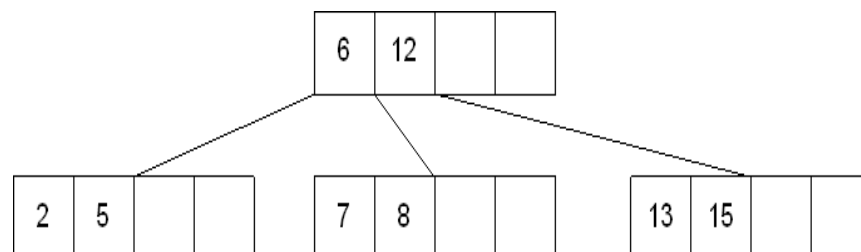
Insert 6 into the following B-tree:



results in a split of the first leaf node:



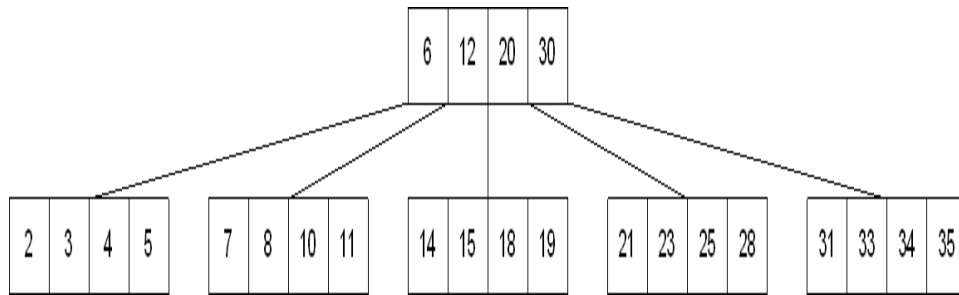
The new node needs to be incorporated into the tree - this is accomplished by taking the middle value and inserting it in the parent:



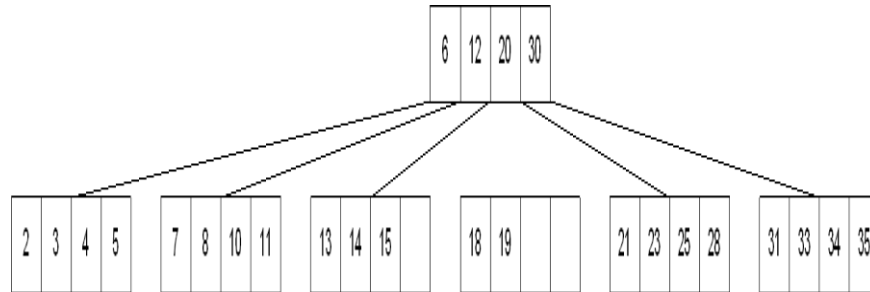
### Case 3: The root of the B-tree is full

The upward movement of values from case 2 means that it's possible that a value could move up to the root of the B-tree. If the root is full, the same basic process from case 2 will be applied and a new root will be created. This type of split results in 2 new nodes being added to the B-tree.

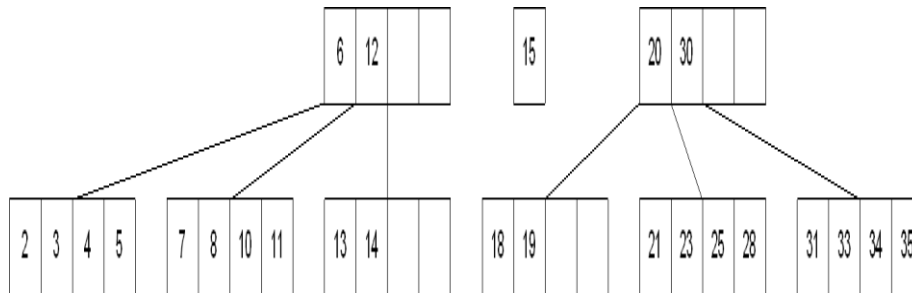
Inserting 13 into the following tree:



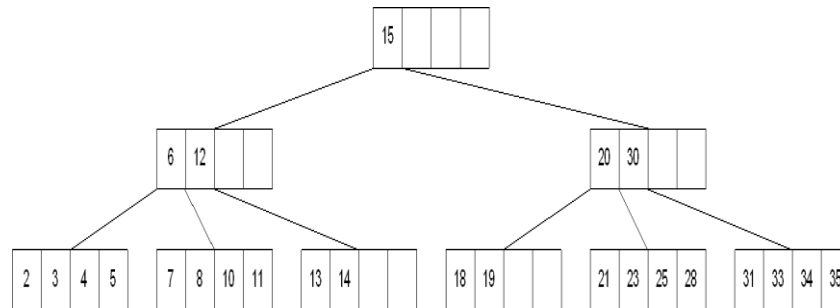
Results in:



The 15 needs to be moved to the root node but it is full. This means that the root needs to be divided:



The 15 is inserted into the parent, which means that it becomes the new root node:



## Deleting from a B-tree

As usual, this is the hardest of the processes to apply. The deletion process will basically be a

reversal of the insertion process - rather than splitting nodes, it's possible that nodes will be merged so that B-tree properties, namely the requirement that a node must be at least half full, can be maintained.

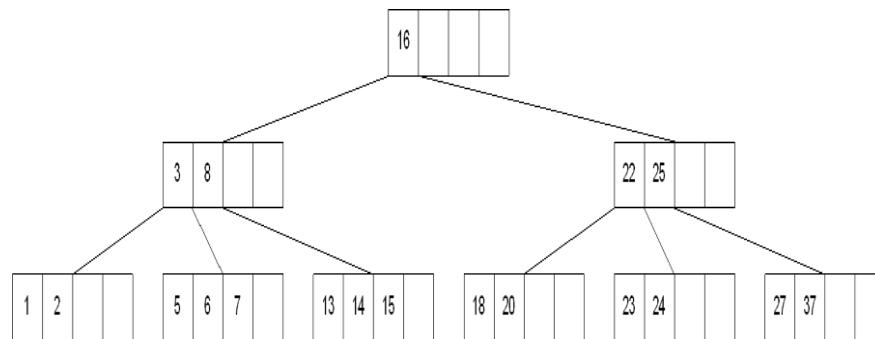
There are two main cases to be considered:

1. Deletion from a leaf
2. Deletion from a non-leaf

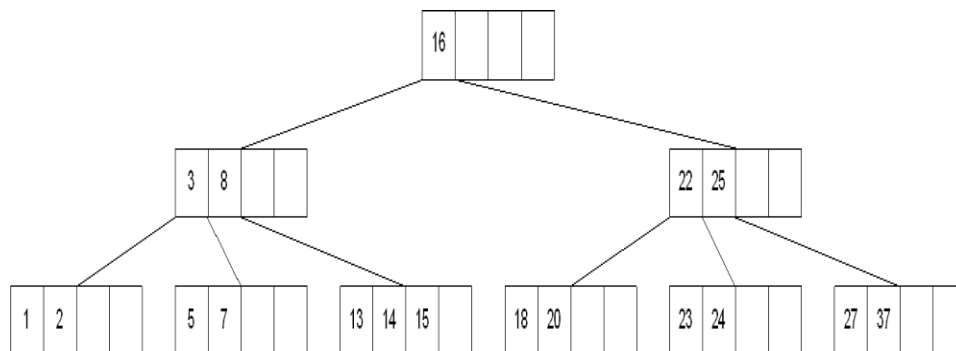
### Case 1: Deletion from a leaf

1a) If the leaf is at least half full after deleting the desired value, the remaining larger values are moved to "fill the gap".

Deleting 6 from the following tree:

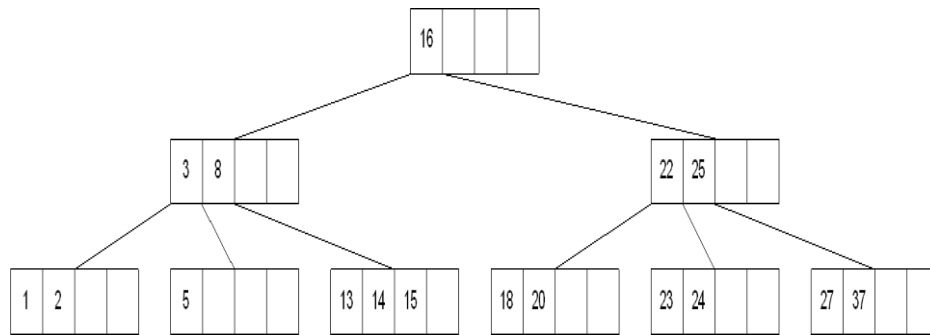


results in:

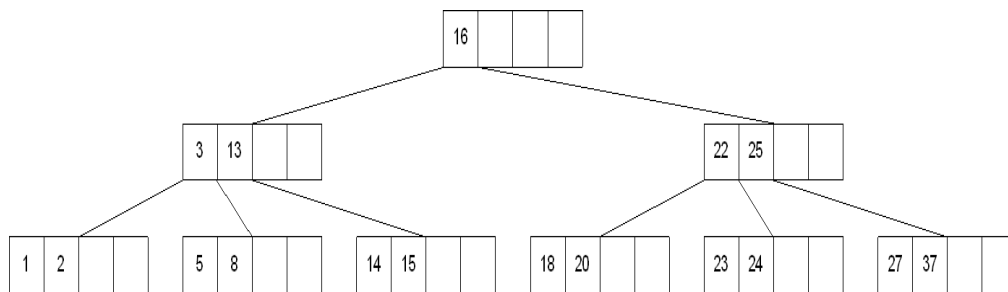


1b) If the leaf is less than half full after deleting the desired value (known as underflow), two things could happen:

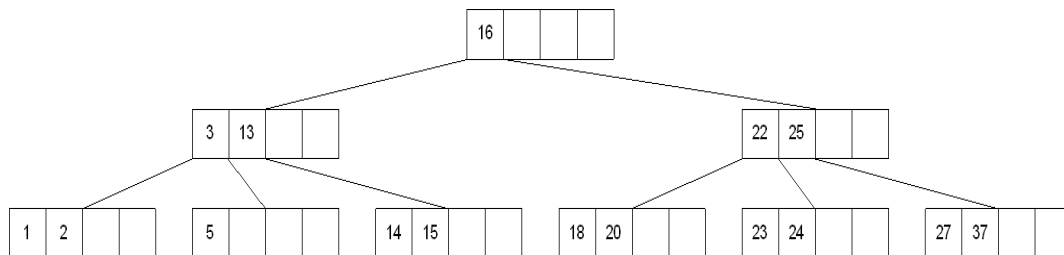
Deleting 7 from the tree above results in:



1b-1) If there is a left or right sibling with the number of keys exceeding the minimum requirement, all of the keys from the leaf and sibling will be redistributed between them by moving the separator key from the parent to the leaf and moving the middle key from the node and the sibling combined to the parent.



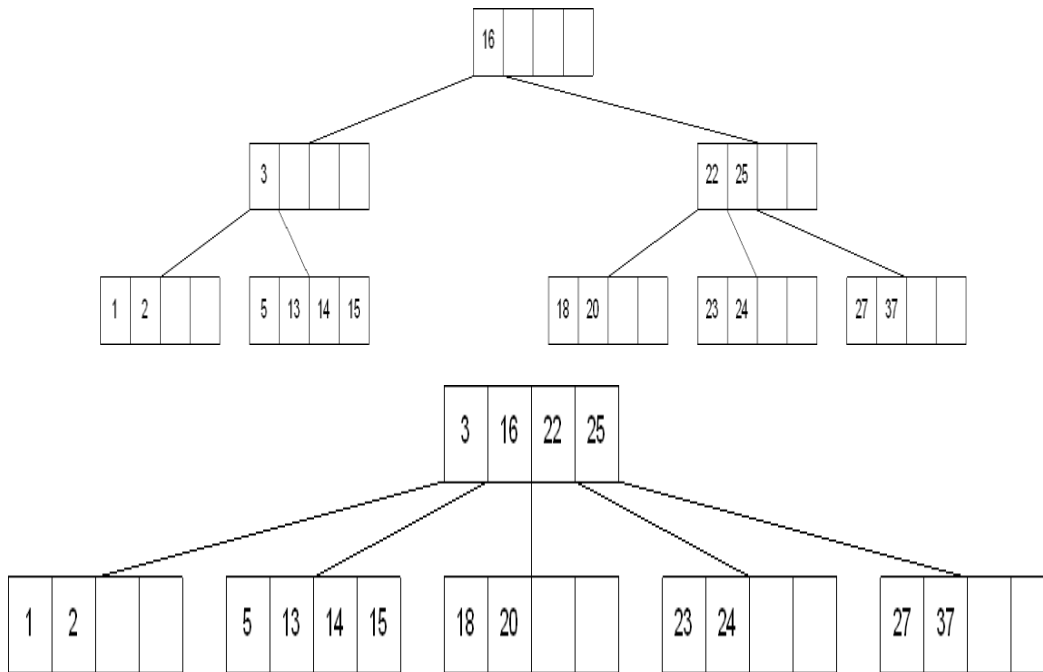
Now delete 8 from the tree:



1b-2) If the number of keys in the sibling does not exceed the minimum requirement, then the leaf and sibling are merged by putting the keys from the leaf, the sibling, and the separator from the parent into the leaf. The sibling node is discarded and the keys in the parent are moved to "fill the gap". It's possible that this will cause the parent to underflow. If that is the case, treat the parent as a leaf and continue repeating step 1b-2 until the minimum requirement is met or the root of the tree is reached.

**Special Case for 1b-2:** When merging nodes, if the parent is the root with only one key, the keys from the node, the sibling, and the only key of the root are placed into a node and this will

become the new root for the B-tree. Both the sibling and the old root will be discarded.

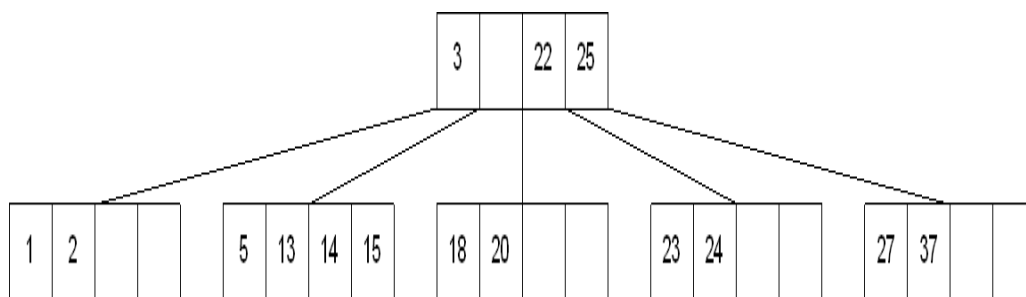


### Case 2: Deletion from a non-leaf

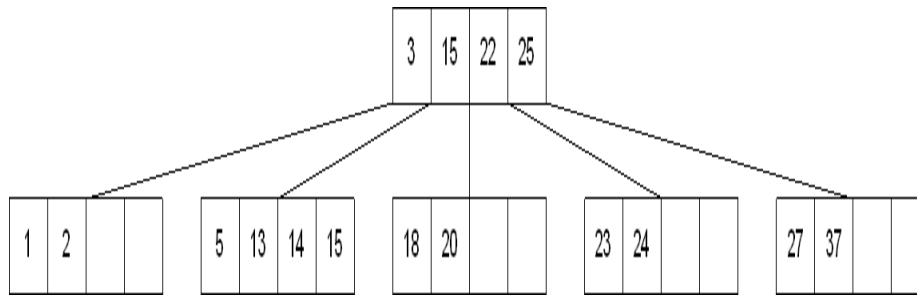
This case can lead to problems with tree reorganization but it will be solved in a manner similar to deletion from a binary search tree.

The key to be deleted will be replaced by its immediate predecessor (or successor) and then the predecessor (or successor) will be deleted since it can only be found in a leaf node.

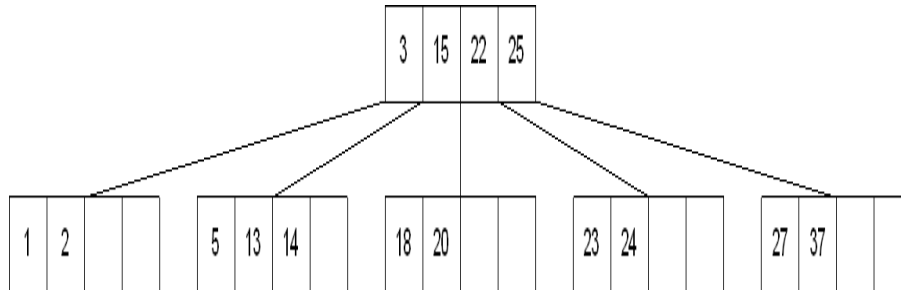
Deleting 16 from the tree above results in:



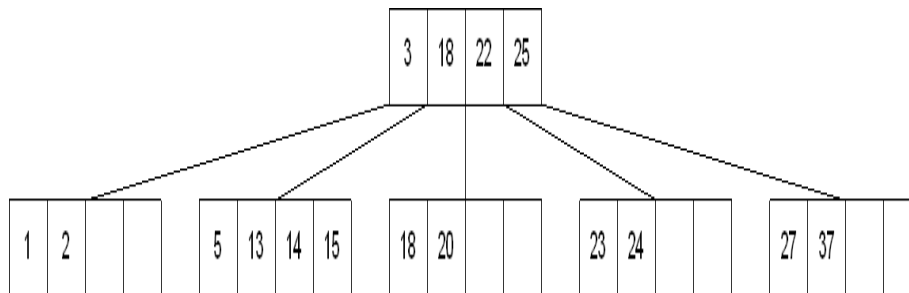
The "gap" is filled in with the immediate predecessor:



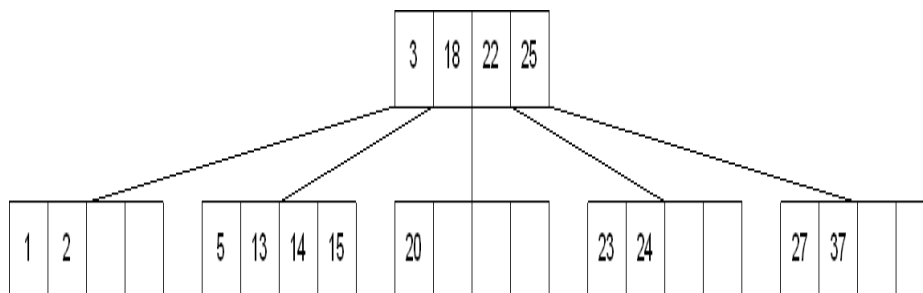
and then the immediate predecessor is deleted:



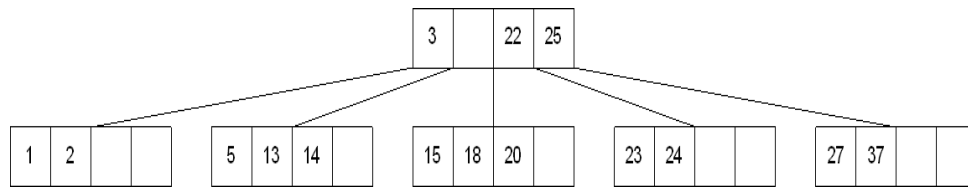
If the immediate successor had been chosen as the replacement:



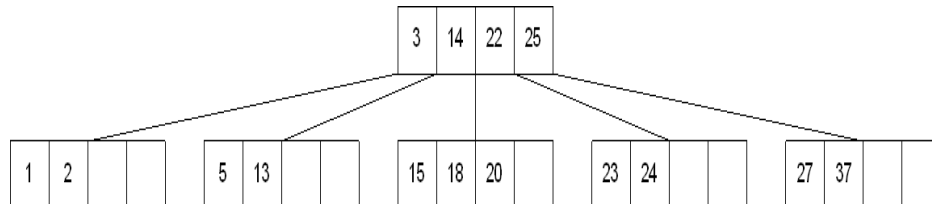
Deleting the successor results in:



The vales in the left sibling are combined with the separator key (18) and the remaining values. They are divided between the 2 nodes:



and then the middle value is moved to the parent:



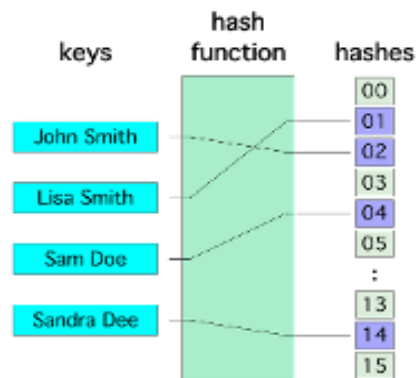
### Hashing and Collision:

Hashing is the technique used for performing almost constant time search in case of insertion, deletion and find operation. Taking a very simple example of it, an array with its index as key is the example of hash table. So each index (key) can be used for accessing the value in a constant search time. This mapping key must be simple to compute and must helping in identifying the associated value. Function which helps us in generating such kind of key-value mapping is known as Hash Function.

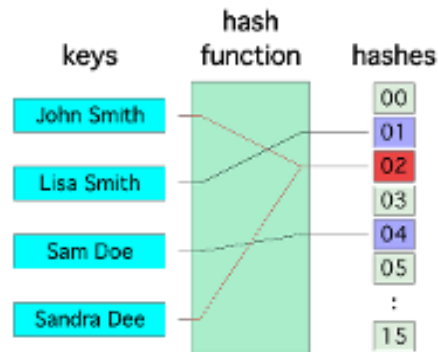
In a hashing system the keys are stored in an array which is called the Hash Table. A perfectly implemented hash table would always promise an average insert/delete/retrieval time of  $O(1)$ .

### Hashing Function:

A function which employs some algorithm to computes the key  $K$  for all the data elements in the set  $U$ , such that the key  $K$  which is of a fixed size. The same key  $K$  can be used to map data to a hash table and all the operations like insertion, deletion and searching should be possible. The values returned by a **hash function** are also referred to as **hash** values, **hash** codes, **hash** sums, or **hashes**.







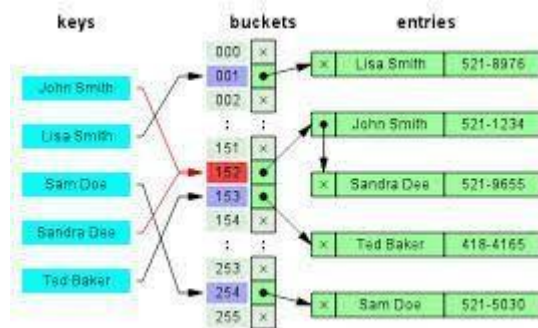
### Hash Collision:

A situation when the resultant hashes for two or more data elements in the data set  $U$ , maps to the same location in the has table, is called a hash collision. In such a situation two or more data elements would qualify to be stored / mapped to the same location in the hash table.

### Hash collision resolution techniques:

#### Open Hashing (Separate chaining):

Open Hashing, is a technique in which the data is not directly stored at the hash key index ( $k$ ) of the Hash table. Rather the data at the key index ( $k$ ) in the hash table is a pointer to the head of the data structure where the data is actually stored. In the most simple and common implementations the data structure adopted for storing the element is a linked-list.



n this technique when a data needs to be searched, it might become necessary (worst case) to traverse all the nodes in the linked list to retrieve the data.

*Note* that the order in which the data is stored in each of these linked lists (or other data structures) is completely based on implementation requirements. Some of the popular criteria are insertion order, frequency of access etc.

#### Closed hashing (open Addressing)

In this technique a hash table with pre-identified size is considered. All items are stored in the hash table itself. In addition to the data, each hash bucket also maintains the three states: EMPTY, OCCUPIED, DELETED. While inserting, if a collision occurs, alternative cells are tried until an empty

bucket is found. For which one of the following technique is adopted.

1. Liner Probing
2. Quadratic probing
3. Double hashing (in short in case of collision another hashing function is used with the key value as an input to identify where in the open addressing scheme the data should actually be stored.)

### A comparative analysis of Closed Hashing vs Open Hashing

Open Addressing	Closed Addressing
All elements would be stored in the Hash table itself. No additional data structure is needed.	Additional Data structure needs to be used to accommodate collision data.
In cases of collisions, a unique hash key must be obtained.	Simple and effective approach to collision resolution. Key may or may not be unique.
Determining size of the hash table, adequate enough for storing all the data is difficult.	Performance deterioration of closed addressing much slower as compared to Open addressing.
State needs be maintained for the data (additional work)	No state data needs to be maintained (easier to maintain)
Uses space efficiently	Expensive on space

### Applications of Hashing:

A **hash function** maps a variable length input string to fixed length output string -- its **hash value**, or **hash** for short. If the input is longer than the output, then some inputs must map to the same output -- a **hash collision**. Comparing the hash values for two inputs can give us one of two answers: the inputs are definitely not the same, or there is a possibility that they are the same. Hashing as we know it is used for performance improvement, error checking, and authentication. One example of a performance improvement is the common hash table, which uses a hash function to index into the correct bucket in the hash table, followed by comparing each element in the bucket to find a match. In error checking, hashes (checksums, message digests, etc.) are used to detect errors caused by either hardware or software. Examples are TCP checksums, ECC memory, and MD5 checksums on downloaded files. In this case, the hash provides additional assurance that the data we received is correct. Finally, hashes are used to authenticate messages. In this case, we are trying to protect the original input from tampering, and we select a hash that is strong enough to make malicious attack infeasible or unprofitable.

- ☐ Construct a *message authentication code* (MAC)
- ☐ Digital signature
- ☐ Make commitments, but reveal message later

- Timestamping
- Key updating: key is hashed at specific intervals resulting in new key

### 1. Define an algorithm?

Ans: An algorithm may be defined as a finite sequence of instructions each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.

The word “**Algorithm**” originates from the Arabic word “Algorism” which is according to the name of Arabic mathematician **AI Khwarizmi**. AI Khwarizmi is considered to be the first algorithm designer for adding two numbers with carry.

### 2. What are the properties of an algorithm?

Ans: An algorithm is endowed with the following properties:

- **Finiteness:** An algorithm must terminate after a finite number of steps.
- **Definiteness:** The steps of the algorithm must be precisely defined.
- **Generality:** An algorithm must be generic enough to solve all problems of a particular class.
- **Effectiveness:** The operations of the algorithm must be basic enough to be put down on pencil and paper. It should not be too complex.
- **Input – Output:** The algorithm must have certain initial and precise inputs, and outputs that may be generated both at its intermediate and final steps.

### 3. What are the approaches to write repetitive algorithms?

Ans: There are two approaches to write repetitive algorithms:

- Iteration
- Recursion

### 4. What is Recursion?

Ans:

- Recursion is a technique that solves a problem by solving a smaller problem of the same type.
- Recursion is a repetitive process in which an algorithm calls itself.
- A recursive function is a function invoking itself, either directly or indirectly. Recursion can be used as an alternative to iteration.
- Every recursive call must either solve a part of the problem or reduce the size of the problem.
- Recursion is a very important and powerful tool in problem solving and programming.
- Recursion is a programming technique that naturally implements the “divide-and-conquer” problem solving methodology.

### 5. What are the rules for designing a recursive algorithm?

Ans: Every recursive algorithm must have a base case. The rules for designing a recursive algorithm are

- First, determine the required parameters
- Second, determine the base case or stopping case.
- Then determine the general case.
- Combine the base case and the general cases into an algorithm.

### 6. How to trace a recursive function?

Ans: A stack is used to keep track of function calls. Whenever a new function is called, all its parameters and local variables are pushed onto the stack along with the memory address of the calling statement. For each function call, an activation record is created on the stack.

### 7. What are the difference between Recursion and Iteration?

Ans:

Sno.	Recursion	Iteration
1.	A Recursive function is a function invoking	Iterative functions are loop based imperative

	itself, either directly or indirectly.	functions.
2.	Recursion uses a stack.	Iteration does not use a stack.
3.	Recursion uses more memory as its concept is based on stacks.	Iteration uses less memory.
4.	Recursion is comparatively slower than iteration due to overhead condition of maintaining stacks.	Iteration is comparatively faster than Recursion.
5.	Recursion makes code smaller.	Iteration makes code longer.
6.	Recursion terminates when a base case is recognized.	Iteration terminates when the loop continuation condition fails.
7.	In recursion multiple activation records are created on stack for each call.	In iteration everything is done in one activation record.
8.	Infinite recursion can crash the system.	Infinite looping uses CPU cycles repeatedly.

### 9. What are the types of recursion?

Ans: Recursion is of two types depending on whether a function calls itself from within itself or whether two functions call one another mutually. The former is called direct recursion and the latter is called indirect recursion.

- ☐ **Direct recursion**
- ☐ **Indirect recursion**

Recursion may be further categorized as

- ☐ **Linear recursion:** performs a single recursive call. E.g. Find the sum of first n integers.
- ☐ **Binary recursion:** performs two recursive calls for each non base case. E.g. Find the Fibonacci series.
- ☐ **Multiple recursion:** performs many recursive calls.

### 10. What is Tail Recursion?

Ans: A function is said to be tail recursive when the recursive call is the last statement invoked in the evaluation of the body of the function. In tail recursion, the recursive call should be the last statement and there should be no earlier recursive calls whether direct or indirect.

### 11. What are the limitations of Recursion?

Ans:

- ☐ Recursion works best when the algorithm uses a data structure that naturally supports recursion. E.g. Trees.
- ☐ In other cases certain algorithms are naturally suited to recursion. E.g. Binary Search, Towers of Hanoi.
- ☐ On the other hand, not all looping algorithms can or should be implemented with recursion.
- ☐ Recursive solutions may involve extensive overhead (both time and memory) due to repetitive calls.

### 12. How to measure the efficiency of algorithms?

Ans: The performances of algorithms can be measured on the scales of

- ☐ **Time**
- ☐ **Space**

**Time Complexity:** The time complexity of an algorithm or a program is a function of the running time of the algorithm or a program. In other words it is the amount of computer time it needs to run to completion.

**Space Complexity:** The space complexity of an algorithm or a program is a function of the space needed by the algorithm or program to run to completion.

### 13. Why should we analyze algorithms?

Ans:

- ☐ To predict the resources that the algorithm requires.
  - Computational time (CPU consumption)
  - Memory space (RAM consumption)
  - Communication bandwidth consumption
- ☐ To predict the running time of an algorithm.
  - Total number of primitive operations executed.

**14. Explain the meaning of  $O(1)$ ,  $O(\log_2 n)$ ,  $O(n)$ ,  $O(n^2)$ ,  $O(n^3)$ ,  $O(2^n)$ ,  $O(k^n)$ ,  $O(n!)$ ?**

Ans:

Complexity	Notation	Description
Constant	$O(1)$	Constant number of operations, not depending on the input data size.
Logarithmic	$O(\log_2 n)$	Number of operations proportional to $\log_2(n)$ where $n$ is the size of the input data.
Linear	$O(n)$	Number of operations proportional to input data size.
Quadratic	$O(n^2)$	Number of operations proportional to the square of the size of the input data.
Cubic	$O(n^3)$	Number of operations proportional to the cube of the size of the input data.
Exponential	$O(2^n)$ $O(k^n)$ $O(n!)$	Exponential number of operations, fast growing.

Some of the commonly occurring time complexities in their ascending orders of magnitude are  $O(1) \leq O(\log_2 n) \leq O(n) \leq O(n \log_2 n) \leq O(n^2) \leq O(n^3) \leq O(2^n)$

**15. What do you mean by complexity of algorithms?**

Ans: Suppose  $M$  is an algorithm, and suppose  $n$  is the size of the input data. So the complexity  $f(n)$  of  $M$  increases as  $n$  increases. It is usually the rate of increase of  $f(n)$ . The running times of algorithms are not just dependent on the size of the input but also its nature.

- ☐ **Worse case time complexity:** The input instances for which the algorithm takes the maximum possible time is called the worst case and the time complexity in such a case is called worst case time complexity.
- ☐ **Best case time complexity:** The input instances for which the algorithm takes the minimum possible time is called the best case and the time complexity in such a case is called best case time complexity.
- ☐ **Average case time complexity:** All other input instances which are neither of the two are categorized as the average case and the time complexity in such a case is called average case time complexity.

e.g. To sequentially search for the first-occurring even number in the given list of numbers.

**Input 1:** -1, 3, 5, 7, 11, -13, 17, 21, -23, -29, 35, 37, 40 -- **worst case** in finding the even number

**Input 2:** 6, 17, 71, 21, 9, 3, 5, 64, -7, 23, 33, 37, 11, -97 -- **best case** in finding the even number

**Input 3:** 71, 21, 9, 3, 1, 5, -23, 11, 33, 36, 37, -5, -13, 22 -- **average case** in finding the even number

**16. What is the need of using asymptotic notation in the study of algorithm?**

Ans: The algorithm performance in comparison to alternate algorithm is best described by the order of growth of the running time of the algorithm. Asymptotic notation is introduced that describe the algorithm performance in a meaningful and impressive way. These notations describe the behavior of

time or space complexity for large characteristics. Some commonly used asymptotic notations are as follows: Big oh notation (**O**), Big Omega notation (**Ω**), Big Theta notation (**θ**), Little Oh notation (**o**).

**17. What are the commonly used asymptotic notations and also give their significance?**

Ans: Asymptotic notations are often used to describe how the size of the input data affects an algorithm's usage of computational resources. Commonly used asymptotic notations are

- **Big oh notation (O):**  $f(n) = O(g(n))$  if there exists a positive integer  $n_0$  and a positive number  $c$  such that  $|f(n)| \leq c|g(n)|$  for all  $n \geq n_0$ . Here  $g(n)$  is the upper bound of the function  $f(n)$ .
- **Big Omega notation (Ω):**  $f(n) = \Omega(g(n))$  if there exists a positive integer  $n_0$  and a positive number  $c$  such that  $|f(n)| \geq c|g(n)|$  for all  $n \geq n_0$ . Here  $g(n)$  is the lower bound of the function  $f(n)$ .
- **Big Theta notation (θ):**  $f(n) = \theta(g(n))$  if there exists a positive integer  $n_0$  and two positive constants  $c_1$  and  $c_2$  such that  $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$  for all  $n \geq n_0$ . Here  $g(n)$  is both an upper bound and lower bound for the function  $f(n)$  for all values of  $n$ .
- **Little Oh notation (o):**  $f(n) = o(g(n))$  if  $f(n) = O(g(n))$  and  $f(n) \neq \Omega(g(n))$ .

**18. Define Data Structure?**

Ans: A data structure is a way of organizing data that considers not only the items stored, but also their relationship to each other. In other words data structure is a representation of logical relationship existing between individual elements of data. Advance knowledge about the relationship between data items allows designing of efficient algorithms for the manipulation of data.

**19. List out any four applications of data structures?**

Ans:

- ☐ Compiler design
- ☐ Operating System
- ☐ Database Management system
- ☐ Network analysis

**20. What is a linear data structure?**

Ans: A data structure is said to be linear if its elements form a sequence or a linear list. E.g. Array, Stacks, Queues and Linked Lists organize data in linear order.

**21. What is a non linear data structure?**

Ans: A data structure is said to be non linear if its elements form a hierarchical classification where, data items appear at various levels. E.g. Trees and Graphs are widely used as non-linear data structures.

**22. What is meant by Searching?**

Ans: Searching is a process to find the location of the given data element in the data structure.

**23. What are the popular searching techniques?**

Ans: There are three popular search techniques. They are

- ☐ Linear or Sequential Search
- ☐ Binary Search
- ☐ Fibonacci Search

**24. What is the basic idea of Linear Search?**

Ans: In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

**25. What is the time complexity of Linear Search?**

Ans: The time complexity of linear search is  $O(n)$ .

**26. What is the basic idea of Binary Search?**

Ans: In binary search, we first compare the key with the item in the middle position of the array. If there's a match, we can return immediately. If the key is less than the middle key, then the item sought must lie in the lower half of the array. If it's greater, then the item sought must lie in the upper half of the array. This process is repeated on the lower (or upper) half of the array.

**27. What is the time complexity of Binary Search?**

Ans: The time complexity of binary search in a successful (or unsuccessful) search is  $O(\log_2 n)$ .

**28. What is the basic idea of Fibonacci Search?**

Ans: It is a search algorithm that applies to a sorted array. It makes use of divide-and-conquer approach that can greatly reduce the time needed in order to reach the target element. This algorithm is used for searching a sorted array by narrowing possible locations to progressively smaller intervals. These intervals are determined with the help of Fibonacci numbers.

**29. What is the time complexity of Fibonacci Search?**

Ans: The time complexity of Fibonacci Search in a successful (or unsuccessful) search is  $O(\log_2 n)$ . But it is much faster compared to the binary search.

**30. What is meant by Sorting?**

Ans: Sorting is a process of efficient arrangement of elements within a given data structure. In other words, sorting means arranging a list of elements in an order (ascending or descending).

**31. What is meant by Internal Sorting?**

Ans: If all elements to be sorted are present in the main memory then such sorting is called internal sorting.

**32. What is meant by External Sorting?**

Ans: If the elements to be sorted are present in the secondary memory then such sorting is called external sorting.

**33. What is the basic idea of Bubble Sort?**

Ans: Bubble sort is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. This process continues until no swaps are required.

**34. What is the time complexity of Bubble Sort?**

Ans: The bubble sort method of sorting an array of size  $n$  requires  $(n-1)$  passes and  $(n-1)$  comparisons in each pass. Thus the total number of comparisons is  $(n-1) * (n-1) = n^2 - 2n + 1$ , which is  $O(n^2)$ .

$$\begin{aligned} F(n) &= (n-1) + (n-2) + \dots + (n-k) + 3 + 2 + 1 \\ &= n(n-1)/2 \\ &= O(n^2) \end{aligned}$$

**35. What is the main idea behind Insertion Sort?**

Ans: In this technique, the list is divided into two parts: sorted and unsorted. In each pass, the first element of the unsorted sub-list is picked up and transferred to the sorted sub-list by inserting it at the appropriate place.

**36. What is the time complexity of Insertion Sort?**

Ans: The time complexity of Insertion sort is  $O(n^2)$  as an array of size  $n$  requires  $(n-1)$  passes to sort the data.



$$\begin{aligned}
 F(n) &= 1+2+3+\dots+(n-k)+\dots+(n-3)+(n-2)+(n-1) \\
 &= n(n-1)/2 \\
 &= O(n^2)
 \end{aligned}$$

### 37. What is the main idea behind Selection Sort?

Ans: In selection sort, first the smallest value in the array is identified and then it is placed in the first position. Then the second smallest value in the array is identified and then it is placed in the second position. This procedure is repeated until the entire array is sorted.

### 38. What is the time complexity of Selection Sort?

Ans: The time complexity of selection sort is  $O(n^2)$  as an array of size  $n$  requires  $(n-1)$  passes and  $(n-1)$  comparisons in each pass.

### 39. What is the purpose of Quick Sort?

Ans: Quick Sort is a very efficient sorting algorithm invented by C.A.R. Hoare. It has two phases:

- ☐ Partition phase
- ☐ Sort phase.

Quick sort works by **divide and conquer** strategy for solving problems. In quick sort, the array of items to be sorted is divided into two partitions. The first partition contains those elements less than some arbitrary chosen value (Pivot) taken from the set, and the second partition contains those elements greater than or equal to the chosen value. Once the array has been rearranged with respect to the pivot, the same partitioning procedure is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

### 40. What is the other name of Quick Sort?

Ans: The other name of Quick Sort is **Partition – exchange sort**.

### 41. What is the time complexity of Quick Sort?

Ans: The time complexity of quick sort algorithm is  $O(n \log_2 n)$ .

### 42. What is Bucket Sort?

Ans: Bucket sort is a sorting algorithm that works by inserting the elements of the sorting array into buckets, and then each bucket is sorted individually. The idea behind bucket sort is that if we know the range of our elements to be sorted, we can set up buckets for each possible element, and just toss elements into their corresponding buckets. We then empty the buckets in order, and the result is a sorted list. Initially we have to set up an array of empty buckets, and then put each object into its bucket. After this, we sort each bucket with elements, and then we pass through each bucket in order and gather all the elements into the original array.

### 43. What is the time complexity of Bucket Sort?

Ans: The time complexity of bucket sort is:  $O(n + m)$  where:

- ☐  $m$  is the range input values
- ☐  $n$  is the total number of values

in the array. It should only be used when the range of input values is small compared with the number of values. In other words, occasions when there are a lot of repeated values in the input. Bucket sort works by counting the number of instances of each input value throughout the array. It then reconstructs the array from this auxiliary data.

### 44. What is Heap Sort?

Ans: A heap sort algorithm works by first organizing the data to be sorted into a special type of binary tree called a heap. Any kind of data can be sorted either in ascending order or in descending order using heap trees.

**45. What is the time complexity of Heap Sort algorithm?**

Ans: The time complexity of Heap Sort algorithm is  $O(n \log_2 n)$ .

**46. What is Merge Sort?**

Ans: The mergesort algorithm is based on the classical divide-and-conquer paradigm. It operates as follows:

**DIVIDE:** Partition the  $n$ -element sequence to be sorted into two subsequences of  $n/2$  elements each.

**CONQUER:** Sort the two subsequences recursively using the merge sort.

**COMBINE:** Merge the two sorted subsequences of size  $n/2$  each to produce the sorted sequence consisting of  $n$  elements.

**47. What is the time complexity of Merge Sort?**

Ans: Merge Sort is a recursive sorting procedure that uses at most  $O(n \log n)$  comparisons.

**48. What is a stack?**

Ans: Stack is a data structure in which both insertion and deletion occur at one end only, called the top of the stack. Stack is maintained with a single pointer to the top of the list of elements. The other name of stack is Last-in -First-out list.

**49. Write operations that can be done on stack?**

Ans: There are two basic operations associated with stack

- ☐ PUSH: It is the term used to insert an element into a stack.
- ☐ POP: It is the term used to delete an element from a stack.

**50. Mention applications of stack?**

Ans:

- ☐ Stack is used in expression parsing
- ☐ Stack is used to evaluate postfix expression
- ☐ Stack is used by compilers to check for balancing of parenthesis
- ☐ Stack is used in recursion

**51. Define Infix, prefix and postfix notations?**

- ☐ Infix: In this form the arithmetic operator is placed in between the operands.  
e.g.  $(A + B) * (C * D)$
- ☐ Prefix: In this form the arithmetic operator is placed before the operands.  
e.g.  $* + A B - C D$
- ☐ Postfix: In this form the arithmetic operator is placed after the operands.  
e.g.  $A B + C D - *$

**52. Explain the usage of stack in recursive algorithm implementation.**

In recursive algorithms, stack data structures is used to store the return address when a recursive call is encountered and also to store the values of all the parameters essential to the current state of the procedure.

**53. What is the criterion for stack empty?**

Ans: When the top pointer contains NULL, then it indicates there are no elements in the stack or the stack is empty.

**54. What is the criterion for Stack Underflow?**

Ans: When the top pointer contains NULL, then this indicates there are no elements in the stack and this situation is called stack underflow.

**55. What is the criterion for Stack Overflow?**

Ans: When TOP = MAX that means the top pointer contains the address of the top most or MAX element, then this situation is called stack overflow.

**56. How to evaluate a postfix expression?**

Ans: The postfix expression is evaluated easily by the use of a stack.

- ☐ When a number is seen, it is pushed onto the stack.
- ☐ When an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.

**57. Define a Queue?**

Ans: Queue is an ordered collection of elements in which insertions and deletions are restricted to one end. The end from which elements are added and / or removed is referred to as the rear end and the end from which deletions are made is referred to as front end.

**58. What are the operations of Queue?**

Ans: There are three basic operations in Queue.

- ☐ Insert
- ☐ Delete
- ☐ Display

**59. What are the conditions that could be followed in a linked list implementations of queue?**

Ans:

**EMPTY QUEUE**  
FRONT == NULL  
**FULL QUEUE**  
NEWMODE == NULL

**60. State any two applications of Queue.**

Ans: Application of Queue:

1. Queues are useful in job scheduling algorithms in the operating system.
2. Queues are also useful for categorization of data.

**61. What are the problems associated with Linear Queue?**

Ans: There are two problems associated with linear queue. They are

- ☐ **Time Consuming:** Linear time to be spent in shifting the elements to the beginning of the queue.
- ☐ **Signaling Queue full:** Even if the queue is having vacant position.

**62. What is Circular Queue?**

Ans: Circular queue is a linear data structure. It follows FIFO principle.

- ☐ In circular queue the last node is connected back to the first node to make a circle.
- ☐ Elements are added at the rear end and the elements are deleted at front end of the queue
- ☐ Both the front and the rear pointers points to the beginning of the array.
- It is also called as “Ring buffer”.

**63. In how many ways Stacks and Queue can be implemented?**

Ans: There are two ways in which circular queue can be implemented.

- ☐ Linked lists
- ☐ Arrays

**64. What is priority queue?**

Ans: A priority queue is a collection of elements such that each element has been assigned a priority such that 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.
- ☐ Two elements with same priority are processed according to the order in which they were added to the queue.

**65. Give a prototype for Priority Queue?**

Ans: A prototype of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.

**66. Application of priority queues.**

Ans: 1. For scheduling purpose in operating system

2. Used for external sorting

3. Important for the implementation of greedy algorithm, which operate by repeatedly finding a minimum.

**67. What are the main properties of a binary heap?**

Ans: 1. Structure property

2. Heap order property

**68. Give an example of Dynamic Data Structure?**

Ans: Linked list is called a dynamic data structure where the amount of memory can be varied during its use.

**69. What is a Linked List?**

Ans: A linked list is an ordered collection of finite, homogeneous data elements called nodes where the linear order is maintained by means of links or pointers.

**70. What is a Node?**

Ans: An element in a linked list is a specially termed node, which consists of two fields:

- ☐ DATA: to store the actual information
- ☐ LINK: to point to the next node

**71. How to represent a Linked List in memory?**

Ans: There are two ways to represent a linked list in memory.

- ☐ Static representation using array
- ☐ Dynamic representation using free pool of storage.

**72. What are the common operations associated with Lists?**

Ans: Four common operations are associated with the list:

- ☐ Insertion
- ☐ Deletion
- ☐ Retrieval
- ☐ Traversal

**73. What do you mean by traversing a List?**

Ans: Traversing a list means going through the list, node by node, and processing each node. Three examples of list traversals are

- ☐ Counting the number of nodes
- ☐ Printing the contents of nodes
- ☐ Summing the values of one or more fields

**74. What are the types of Linked Lists?**

Ans: Basically Linked Lists can be categorized into four types:

- ☐ Single Linked List
- ☐ Double Linked List
- ☐ Circular Linked List
- ☐ Circular Double Linked List

**75. What is a Single Linked List?**

Ans: In a single linked list each node contains only one link which points to the subsequent node in the list. It is also called as linear linked list or one-way list.

**76. What is a Double Linked List?**

Ans: A double linked list is a two-way list because one can move in either direction, either from left to right or from right to left. This is accomplished by maintaining two link fields, which helps in accessing both the successor node and predecessor node.

**77. What is a Circular Linked List?**

Ans: A list can be implemented using a circularly linked list in which the last node's link points to the first node of the list.

**78. What is a Circular Double Linked List?**

Ans: A list can be implemented using a doubly linked list in which each node has a pointer to both its successor and its predecessor in the circular manner.

**79. What is a Multilinked List?**

Ans: A list can be implemented using a multilinked list to create two or more logical lists.

**80. What are the applications of a Linked List?**

Ans:

- ☐ Used to implement the symbol table in Compiler Construction
- ☐ Used to implement Stack, Queue, Trees and Graphs
- ☐ Used to represent and manipulate polynomials
- ☐ Used to represent sparse matrix
- ☐ Used as a way to represent deck of cards in a game

**81. What are the advantages of Linked Lists?**

Ans: Linked lists have many advantages:

- ☐ Linked lists are dynamic data structures i.e., they can grow or shrink during the execution of a program.
- ☐ Linked lists have efficient memory utilization.
- ☐ Insertion and Deletions are easier and efficient.

**82. What are the Disadvantages of Linked Lists?**

Ans:

- ☐ It consumes more space because every node requires an additional pointer to store address of the next node.
- ☐ Searching a particular element in list is difficult and also time consuming.

**83. Define tree traversal and mention the type of traversals.**

Ans: Visiting of each and every node in the tree exactly is called as tree traversal.

Three types of tree traversal

1. In order traversal
2. Pre order traversal
3. Post order traversal

**84. Define tree?**

Ans: Trees are non-linear data structure, which is used to store data items in a shorted sequence. It represents any hierarchical relationship between any data Item. It is a collection of nodes, which has a distinguish node called the root and zero or more non-empty sub trees  $T_1, T_2, \dots, T_k$ . each of which are connected by a directed edge from the root.

**85. What do you mean by a Sub tree?**

Ans: Any node of a tree, with all of its descendants is a sub tree.

**86. Define Height of tree?**

Ans: The height of  $n$  is the length of the longest path from root to a leaf. Thus all leaves have height zero. The height of a tree is equal to a height of a root.

**87. Define Depth of tree?**

Ans: For any node  $n$ , the depth of  $n$  is the length of the unique path from the root to node  $n$ . Thus for a root the depth is always zero.

**88. Define Degree of a node?**

Ans: It is the number of sub trees of a node in a given tree.

**89. Define Degree of a tree?**

Ans: It is the maximum degree of a node in a given tree.

**90. Define Terminal node or leaf node?**

Ans: Nodes with no children are known as leaves. A leaf will always have degree zero and is also called as terminal node.

**91. Define Non-terminal node?**

Ans: Any node except the root node whose degree is a non-zero value is called as a non terminal node. Non-terminal nodes are the intermediate nodes in traversing the given tree from its root node to the terminal node.

**92. Define Sibling?**

Ans: Nodes with the same parent are called siblings.

**93. Define Binary Tree?**

Ans: A Binary tree is a finite set of data items which is either empty or consists of a single item called root and two disjoin binary trees called left sub tree max degree of any node is two.

**94. What are the properties of Binary Tree?**

Ans: Some of the important properties of a binary tree are as follows:

- ☐ If  $h$  = height of a binary tree
  - ☐ Maximum number of leaves =  $2^h$
  - ☐ Maximum number of nodes =  $2^{h+1} - 1$
- ☐ If a binary tree contains  $m$  nodes at level  $l$ , then it contains at most  $2m$  nodes at level  $l+1$ .
- ☐ Since a binary tree can contain at most one node at level 0 (the root), it can contain at most  $2^l$  node at level  $l$ .

- The total number of edges in a full binary tree with  $n$  nodes is  $n - 1$ .

**95. What is the minimum and maximum height of a binary tree?**

Ans: The minimum and maximum height of a binary tree can be related to the number of nodes:

- $H_{\min} = \lceil \log_2 N \rceil + 1$
- ☐  $H_{\max} = N$

**96. What is the balance factor of a Binary Tree?**

Ans: The balance factor of a binary tree is the difference in height between its left and right sub trees.

- ☐  $B = H_L - H_R$

**97. What is a Complete Binary Tree?**

Ans: A complete binary tree has the maximum number of entries for its height; a tree is complete when the last level is full.

**98. What are the data structures used for Binary Trees?**

Ans:

- ☐ Arrays: Especially suited for complete and full binary trees.
- ☐ Pointer based.

**99. What are the applications of Binary Tree?**

Ans: Binary tree is used in data processing.

- File index schemes
- Hierarchical database management system

**100. What are the two approaches for Binary Tree Traversal?**

Ans: The two approaches to binary tree traversal are

- ☐ Depth first traversal
- ☐ Breadth first traversal

**101. What are the various operation performed in the binary search tree?**

- ☐ Insertion
- ☐ Deletion
- ☐ Find
- ☐ Find min
- ☐ Find max

**102. What are the three approaches for inserting data into general trees?**

Ans: The three approaches for inserting data into general trees are:

- ☐ FIFO
- ☐ LIFO
- ☐ Key Sequenced

**103. What is the criterion for deleting a node in a general tree?**

Ans: To delete a node in a general tree, we must ensure that it does not have a child.

**104. What is an ordered tree?**

Ans: In a directed tree if the ordering of the nodes at each level is prescribed then such a tree is called ordered tree.

**105. What is meant by traversing?**

Ans: Traversing a tree means processing it in such a way, that each node is visited only once.

**106. What are the two methods of binary tree implementation?**

Ans: Two methods to implement a binary tree are,

- ☐ Linear representation.
- ☐ Linked representation

**107. Define pre-order traversal.**

Ans: Pre-order traversal entails the following steps:

- ☐ Process the root node
- ☐ Process the left sub-tree
- ☐ Process the right sub-tree

**108. Define post-order traversal.**

Ans: Post order traversal entails the following steps:

- ☐ Process the left sub-tree
- ☐ Process the right sub-tree
- ☐ Process the root node

**109. Define in -order traversal.**

Ans: In-order traversal entails the following steps:

- ☐ Process the left sub-tree
- ☐ Process the root node
- ☐ Process the right sub-tree

**110. What are the disadvantages of sequential representation of tree over the linked representation?**

Ans: In sequential representation for storing the binary tree an array of some fixed size is available. There are chances that some nodes of the tree do not get stored in the array or there may be wastage of some space when the tree to be stored is very small. On the other hand in linked list representation the memory gets allocated dynamically hence there will not be any wastage of memory or shortage of space for storing the tree.

**111. What is meant by threaded binary tree?**

Ans: A threaded binary tree makes it possible to traverse the values in the binary tree via a linear traversal that is more rapid than a recursive in-order traversal. It is also possible to discover the parent of a node from a threaded binary tree, without explicit use of parent pointers or a stack, albeit slowly. This can be useful where stack space is limited, or where a stack of parent pointers is unavailable (for finding the parent pointer via DFS). This is possible, because if a node (k) has a right child (m) then m's left pointer must be either a child, or a thread back to k. In the case of a left child, that left child must also have a left child or a thread back to k, and so we can follow m's left children until we find a thread, pointing back to k. The situation is similar for when m is the left child of k.

**112. What is a Graph?**

Ans: A graph is a collection of nodes, called vertices, and a collection of line segments connecting pairs of nodes, called edges or arcs.

**113. Define a Cycle in a Graph?**

Ans: A cycle is a path of at least three vertices that starts and ends with the same vertex.

**114. Define Loop in a Graph?**

Ans: A loop is a special case of a cycle in which a single arc begins and ends with the same node.



**115. Define a Connected Graph?**

Ans: A graph is said to be connected if, for any two vertices, there is a path from one to other. A graph is disjointed if it is not connected.

**116. What are the various operations defined for a Graph?**

Ans: Six operations have been defined for a graph:

- ☐ Insert a vertex
- ☐ Delete a vertex
- ☐ Add an edge
- ☐ Delete an edge
- ☐ Find a vertex
- ☐ Traverse a graph

**117. What are the Graph Traversal techniques?**

Ans: There are two standard graph traversal:

- **Depth first traversal:** In this traversal, all of a node's descendents are processed before moving to an adjacent node.
- ☐ **Breadth first traversal:** In this traversal, all of the adjacent vertices are processed before processing the descendents of a vertex.

**118. What are the most common methods used to store a graph?**

Ans: The most common methods used to store a graph are:

- ☐ Adjacency matrix
- ☐ Adjacency List

**119. What is a Spanning Tree?**

Ans: A spanning tree contains all vertices in the graph.

**120. What do you mean by Minimum Spanning Tree (MST)?**

Ans: A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph i.e., any connected graph will have a spanning tree.

**121. What is a Forest?**

Ans: A Forest is a set of disjoint trees. If we remove the root node of a given tree then it becomes a forest.

**122. What is a multi-graph?**

Ans: A graph that has either self loop or parallel edges or both is called multi-graph.

