VJTech Academy

Inspiring your Success…

Degree & Diploma Coaching classes.

Data Structure Using 'C'

Author,

Mr. Vishal Jadhav Sir

(BE Computer, 8 Years of IT Industry experience)

2022

Venue,
VJTech Computer Academy, Maharashtra
M. 9730087674, Email- vjtechacademy@gmail.com

mswordcoverpages.com

❖ Data Structure:

Definition:

- Data is a collection of numbers, alphabets and special symbols which are used to represent an information.
- The proper arrangement of data is known as data structure.
- The systematic representation of data in main memory is known as data structure.
- Data can be organized in different ways.
- The logical or mathematical model of a particular organization of data is called as data structure.
- A data structure is a way of storing data in a computer so that it can be used efficiently.

Need of Data Structure:

- Data structures are an important way of organizing information or data in a computer.
- It has a different ways of storing & organizing data in a computer.
- It helps to understand relationship of one data element with other.
- It helps to store data in logical manner.
- We need to store the data in such a way that may grow & shrink dynamically.
- If we organize the information in some proper manner then we can access it using efficient algorithm.
- Data structures are uses many efficient algorithms, & because of that it is possible to manage the huge amounts of data.

❖ Abstract Data Type(ADT):

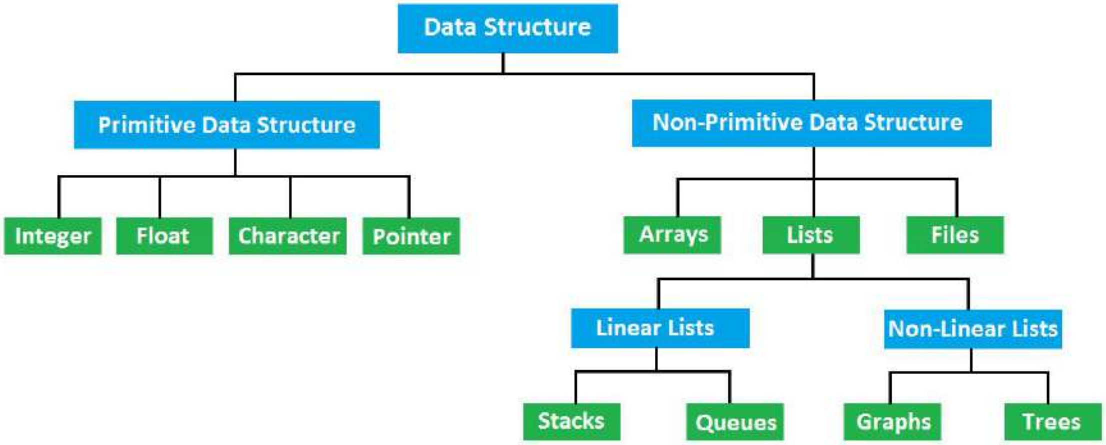
- ADT stands for Abstract Data Types.
- Abstract Data type (ADT) is a type for objects whose behavior is defined by a set of value and a set of operations.
- The definition of ADT mentions what operations are to be performed on the objects without giving the detail information about the how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called as “Abstract” because it will show only essential details and hide the unwanted details.

- The process of providing only the essentials details and hiding the unwanted details is known as abstraction.
- For example, we use int, float, char data types for storing values and performing the various operations. But we don't know how these operations are performed on the data.
- It means user only understand what are the data types and what it does but they don't know how it will manage data in memory and how it will do the various operations.
- ADT as a black box which hides the inner structure and design of the data type.

❖ Classification of data structure:

The classification of data structure mainly consists of :

1. Primitive data structure
2. Non-primitive data structure



Primitive data structure :

- The primitive data structures are known as basic data structures.
- All basic data types of any language are called as primitive data types.
- It defines how the data will be internally represented in, stored and retrieve from memory.
- Example.

    1. Integer
    2. Float
    3. Character
    4. Pointer

Non-Primitive data structure :

- All the data structures derived from primitive data structures are called as non-primitive data structures.
- The non-primitive data structures are highly developed complex data structures.
- The non-primitive data structure is responsible for organizing the group of homogeneous and heterogeneous data elements.
- Example.
  1. Arrays
  2. Lists

     I) Linear data structure

     a) Stack

     b) Queue

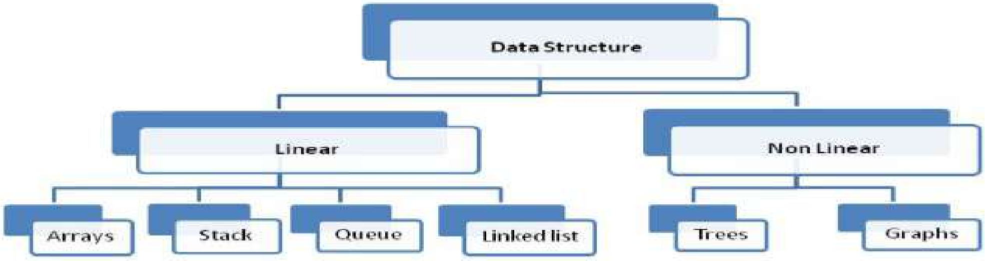
     II) Non-Linear data structure

     a) Tree

     b) Graph
  3. Files

❖ Linear and Non Linear Data Structure:

- Data structures are basically a way of storing and logically implementing the data elements.
- Primitive data structures which include the int, char, float, double and the non-primitive data structures can broadly be classified into two types:
  1. Linear Data Structure
  2. Non-Linear Data Structure



Linear Data Structure:

- In this type of data structure, all the data elements are stored in a particular sequence.
- All elements are arranged in linear fashion.
- The sequential organization of data elements is known as linear data structure.
- Examples: Array, Linked List, Stack, Queue, etc.

Non-Linear Data Structure:

- In this type of data structure, all the data elements do not form any sequence.
- All elements are arranged in non-linear fashion.
- The randomly organization of data elements is known as non-linear data structure.
- Examples: Tree, Graph, Table, etc.

❖ Operations on Data Structure:

The basic operations that are performed on data structures are as follows:

1. **Insertion:** It is adding a new data in data structure.
2. **Deletion:** It is removing a data from existing data structure
3. **Searching:** It is finding location of data within given data structure
4. **Sorting:** It is an arranging data in some logical order, it may be in ascending or descending order.
5. **Traversing:** It is an operation that access each and every element.
6. **Merging:** It is used to combine the data items of two data structure into single data structure.

❖ Algorithms Complexity:

Time complexity:-

- Time complexity of a program/algorithm is the amount of computer time that it needs to run to completion.
- How much time required for execution of given program/algorithm is know as time complexity.
- While calculating time complexity, we develop frequency count for all key statements which are important and basic instructions of an algorithm.
- How much time program controller visits a particular line, that count is known as frequency count.
- Example:

Sr.No	Statement	Frequency count
1.	sum=0;	1
2.	for(i=1;i<n;i++)	n+1
3.	sum=sum+i;	n
4.	printf("%d",sum);	1

So the time complexity is **2n+3** and if we represent this time complexity using Big-Oh notation then it will be **O(n)**.

Space complexity:-

- Space complexity of a program/algorithm is the amount of memory that it needs to run to completion.
- How much space is required for execution of given program/algorithm is known as space complexity.
- The space complexity consists of two parts:
    1. Fixed space part: It includes space for instructions, for simple variables, fixed size structured variables and constants.
    2. Variable space part: It consists of space needed by structured variables whose size depends on particular value of variables.

Space Complexity = Fixed part space + Variable part space

- **Example:**

```
sum=0;
for(i=0;i<n;i++);
{
     sum=sum+i;
}
printf("%d",sum);
```

Above example consists of three variables i,n,sum of integer type so fixed part space complexity is 6 byes and variable part is 0 bytes. So total space complexity is **6 + 0 = 6 bytes**

❖ Define Big 'O' Notation:

- This notation is used to represent time complexity of algorithm/program.
- Big O is a mathematical notation that represents time complexity of an algorithm.
- O stands for order of term and it is pronounced as **Big-Oh.**
- This notation calculate the upper bound value of given function.
- Suppose, if we have two functions i.e f(x) and g(x) then f(x) is said to be O(g(x)) if there exists two positive constants c and k such that **f(x) <= c(g(x)** for all x>=k

<u>Important Questions:</u>

1. Define Abstract Data Type. **2 Marks**
2. Explain time complexity and space complexity. (2M for time complexity and 2M for space complexity) **4 Marks**
3. Define data structure and give its classification (Definition - 1 Mark, Classification -1 Mark) **2 Marks**
4. Enlist various types of operation on data structure (Any 4 operations) **2 Marks**
5. Define primitive and non-primitive data structure. **4 Marks**
6. State the need of data structure. Write any four operations perform on data structure. **4 Marks**

❖ Sorting:

Definition:

- Sorting is an operation which we can perform on the data structure to arrange the elements in ascending and descending order.
- Sorting is the process where row data takes as input and generate the ordered data, it may be in ascending or descending order.
- Sorting is the operation of arranging data or elements in some given order.

Classification of sorting:

- The sorting is classified into two categories.

1. **Internal Sorting:** The sorting which is done on computer main memory is known as internal sorting. In this sorting technique, all the data is stored in main memory only and the data can be accessed randomly. Example: Bubble sort, Selection sort, Insertion sort, Quick sort, Radix sort, etc.
2. **External Sorting:** The sorting which is done on secondary memory is known as external sorting. If number of elements to be sorted is too large then external sorting is required. Example: Merge sort, etc.

❖ Sorting Techniques:

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Radix Sort
5. Quick Sort
6. Merge Sort
7. Shell Sort

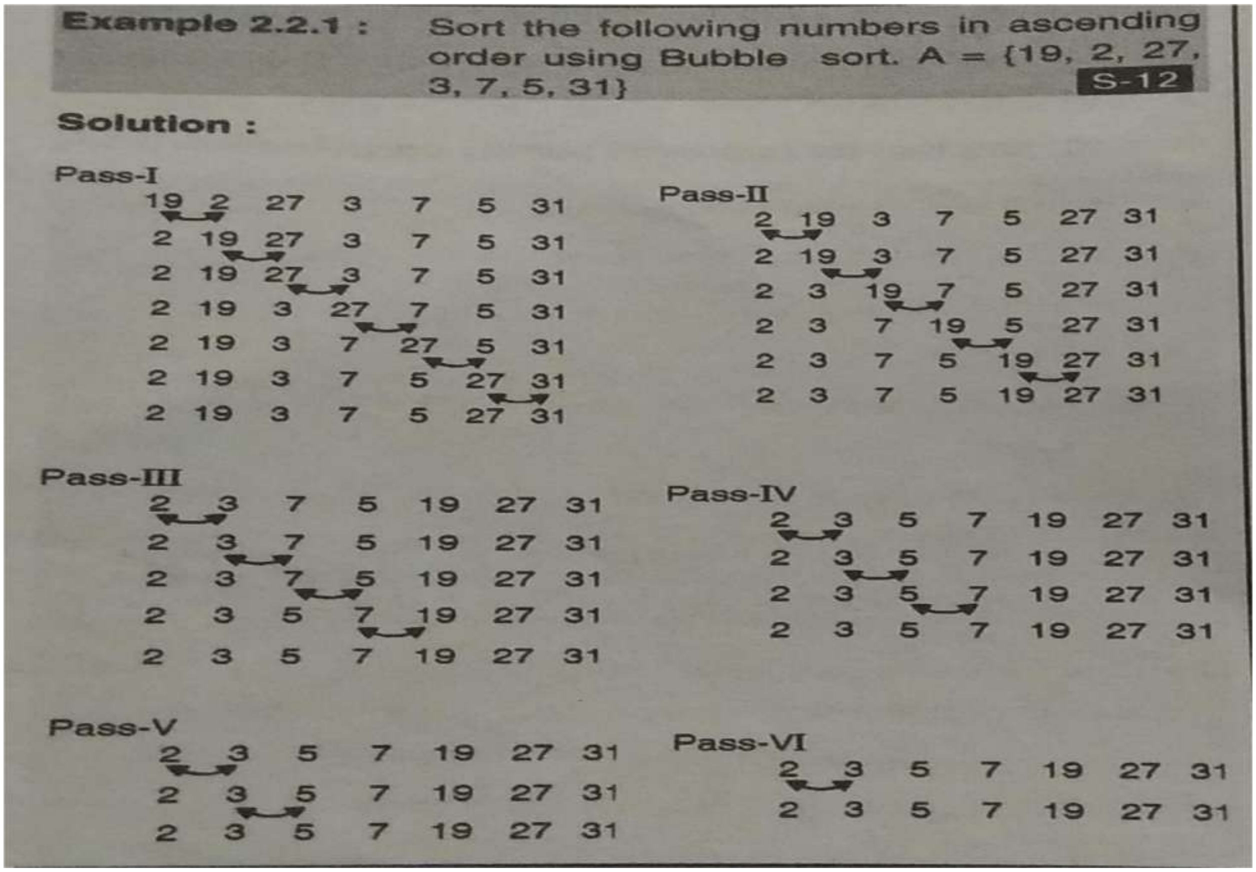
❖ Bubble Sort:

- Bubble sort is the one of the type of sorting algorithm. It is stable sorting algorithm.
- Following are the steps involved in the Bubble sort:

  1. Compares two adjacent elements.

  2. If current element is greater than next element then swap them.

  3. If current element is less than next element then no action required, move to the next element and repeat step 1.
- Sorting begins with 0th elements & it compares with 1st element in given list.
- If 0th element is greater than 1st element then swap them. Like this all elements are compared with next element & interchanged if required.
- At the end of 1st pass largest element is placed at last position.
- In 2nd pass again comparisons starts with 0th element & after completion of 2nd pass, 2nd largest element is placed at second last position.
- This process continues till list is in sorted order.
- In Bubble sort, total N-1 passes are required for sorting N elements.
- **Example:**



Example 2.2.1 : Sort the following numbers in ascending order using Bubble sort. A = {19, 2, 27, 3, 7, 5, 31} S-12

Solution :

Pass-I

```
19  2  27  3  7  5  31
 2 19  27  3  7  5  31
 2 19  27  3  7  5  31
 2 19   3 27  7  5  31
 2 19   3  7 27  5  31
 2 19   3  7  5 27  31
 2 19   3  7  5 27  31
```

Pass-II

```
2  19   3   7   5  27  31
2  19   3   7   5  27  31
2   3  19   7   5  27  31
2   3   7  19   5  27  31
2   3   7   5  19  27  31
2   3   7   5  19  27  31
```

Pass-III

```
2  3  7  5  19  27  31
2  3  7  5  19  27  31
2  3  7  5  19  27  31
2  3  5  7  19  27  31
2  3  5  7  19  27  31
```

Pass-IV

```
2  3  5  7  19  27  31
2  3  5  7  19  27  31
2  3  5  7  19  27  31
2  3  5  7  19  27  31
```

Pass-V

```
2  3  5  7  19  27  31
2  3  5  7  19  27  31
2  3  5  7  19  27  31
```

Pass-VI

```
2  3  5  7  19  27  31
2  3  5  7  19  27  31
```

- **Time complexity:**
  Best Case => **O(n2)**, Average Case => **O(n2)** Worst Case => **O(n2)**

- **Bubble Sort Algorithm:**
  Let a[20] is an integer array, N is size of array.

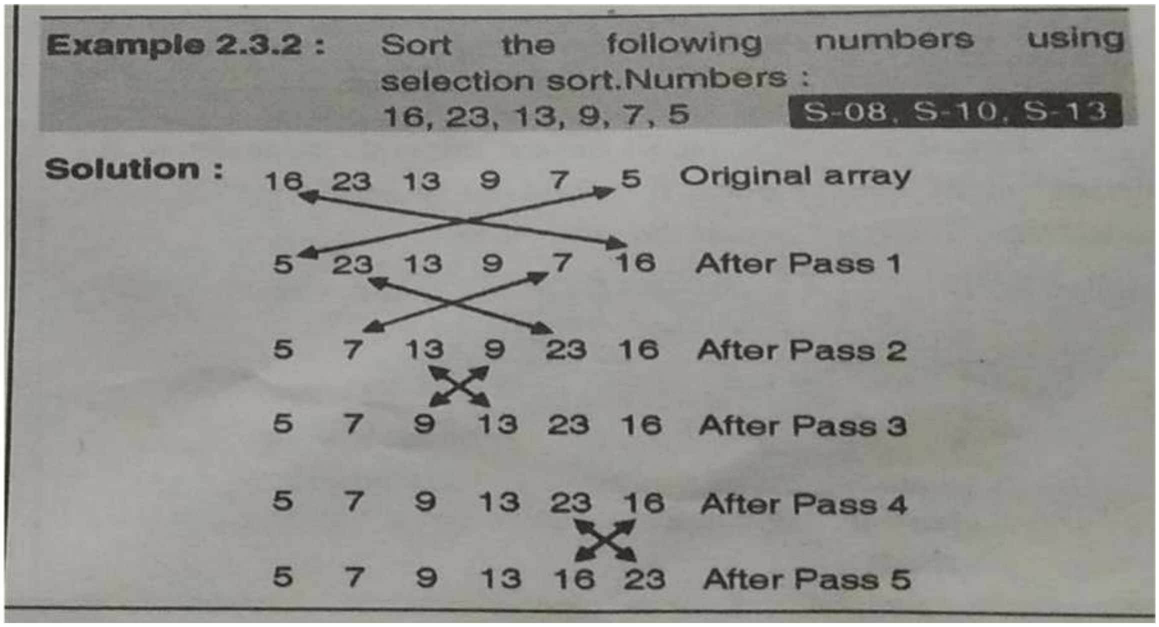
```
1) Repeat for i=1 to N-1 i.e N-1 passes.
2)     Repeat for j=0 to N-i
3)         if a[j]>a[j+1]
4)             Interchange a[j] with a[j+1].
5) Print sorted array.
```

- **Write a C Program to implement Bubble sort.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a[20],i,j,temp,N;
        clrscr();
        printf("Enter size of array: ");
        scanf("%d",&N);
        printf("\nEnter Array Elements:");

        for(i=0;i<N;i++)
        {
                scanf("%d",&a[i]);
        }
        for(i=1;i<N;i++)
        {
                for(j=0;j<N-i;j++)
                {
                        if(a[j]>a[j+1])
                        {
                                temp=a[j];
                                a[j]=a[j+1];
                                a[j+1]=temp;
                        }
                }
        }
        printf("\n\n\n Sorting Array Elements are: ");
        for(j=0;j<N;j++)
        {
                printf("  %d",a[j]);
        }
        getch();
}
```

❖ Selection Sort:

- Selection sort is the one of the type of sorting algorithm.
- Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending).
- In selection sort, the first element in the list is selected and it is compared repeatedly with remaining all the elements in the list.
- If any element is smaller than the selected element , then both are swapped.
- At the end of 1st pass smallest element is placed at first position.
- In 2nd pass, we select the element at second position in the list and it is compared with remaining all elements in the list.
- If any element is smaller than the selected element, then both are swapped.
- At the end of 2st pass second smallest element is placed at second position.
- This procedure is repeated till the entire list is sorted.
- In selection sort, total N-1 passes are required for sorting N elements.
- **Example:**



Example 2.3.2 : Sort the following numbers using selection sort.Numbers :
16, 23, 13, 9, 7, 5 S-08, S-10, S-13

Solution :

```
16  23  13   9   7   5   Original array
 5  23  13   9   7  16   After Pass 1
 5   7  13   9  23  16   After Pass 2
 5   7   9  13  23  16   After Pass 3
 5   7   9  13  23  16   After Pass 4
 5   7   9  13  16  23   After Pass 5
```

- **Time complexity:**

Best Case => **O(n2)**,
Average Case => **O(n2)**
Worst Case => **O(n2)**

- **Selection Sort Algorithm:**
  Let a[20] is an integer array, N is size of array.

```
1.  Repeat for i=1 to N-1 i.e N-1 passes.
2.      Set k=i;
3.      Repeat for j=i+1 to N-1
4.          if a[j]<a[k] then
5.              Set k=j
6.      if k!=i then
7.          Interchange a[k] with a[i].
8.  Print sorted array.
```

- **Write a C Program to implement Selection sort.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a[20],i,j,N,k,temp;
        clrscr();
        printf("Enter size of array: ");
        scanf("%d",&N);
        printf("\nEnter Array Elements:");

        for(i=0;i<N;i++)
        {
                scanf("%d",&a[i]);
        }
        for(i=1;i<N;i++)
        {
                k=i;
                for(j=i+1;j<N;j++)
                {
                        if(a[j]<a[k])
                        {
                                k=j;
                        }
                }
                if(k!=i)
                {
                        temp=a[i];
                        a[i]=a[k];
                        a[k]=temp;
                }
        }
        printf("\n\n\n Sorting Array Elements are: ");
        for(j=0;j<N;j++)
        {
                printf("  %d",a[j]);
        }
        getch();
}
```

VJTECH Academy

❖ Insertion Sort:

- Insertion sort is the one of the type of sorting algorithm. It is stable sorting algorithm.
- In insertion sort, sorting is done on the basis of shift and insert principle.
- In insertion sort, elements are inserted at right position.
- In first pass, 1st index element is compared with oth index element. If 0th index element is greater than 1st index element then store 1st index element into a temporary variable and shift oth index element to its right by one position. Then insert temporary variable value in oth position.
- In 2nd pass, compare 2nd index element with 1th index and then 0st index elements. If required perform shift and insert operations.
- In each pass fix one position and compare it with all elements placed before it.
- Continue this process till last position element is compared with all elements placed before it.
- In Selection sort, total N-1 passes are required for sorting N elements.
- **Example:**

  Refer your class notebook.

- **Time complexity:**
  Best Case => **O(n^2)**,
  Average Case => **O(n^2)**
  Worst Case => **O(n^2)**

- **Insertion Sort Algorithm:**
  Let a[20] is an integer array, N is size of array.

```
1. Repeat for i=1 to N-1 i.e N-1 passes.
2.     Set temp=a[i]
3.          Repeat for j=i-1; j>=0 && a[j]>temp ; j-- then
4.               Set a[j+1]=a[j];
5. Set a[j+1]=temp
6. Print sorted array.
```

- **Write a C Program to implement Insertion sort.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a[10],i,j,n,temp;
        clrscr();
        printf("\nEnter size of array:");
        scanf("%d",&n);
        printf("\nEnter array elements are:");
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
        }
        for(i=1;i<n;i++)
        {
                temp=a[i];
                for(j=i-1;j>=0 && a[j]>temp;j--)
                {
                        a[j+1]=a[j];
                }
                a[j+1]=temp;
        }
        printf("\n Sorted Array Elements are:");
        for(i=0;i<n;i++)
        {
                printf("%d\t",a[i]);
        }
        getch();
}
```

❖ Radix Sort:

- Radix sort is the one of the type of sorting algorithm It is stable sorting algorithm..
- In this method, ten buckets (0-9) are used to sort elements of an input list.
- All the elements are sorted according to their digit position from each element.
- In pass one each element is placed inside the bucket with respect its unit position digit.
- After placing all elements inside the buckets, read those from 0th bucket to 9th bucket.
- In pass 2, elements are placed in buckets with respect to 10th position digit from each element.
- In each pass one position is considered to arrange all the elements in bucket.
- At the end of each pass elements are collected from buckets and given as input to the next pass.
- Total number of passes required for sorting is equal to maximum number of digits present in the largest number from the input list.
- Last pass gives sorted list after reading all elements from 0 th bucket to 9th bucket.
- **Time complexity:**
  Best Case => **O(n)**
  Average Case => **O(n)**
  Worst Case => **O(n)**

- Number of passes required to sort the using the radix sort is equal to number of digits in the largest number in the list.

Range	Passes
0 to 9	1 pass
0 to 99	2 passes
0 to 999	3 passes

VJTECH Academy

a) Describe radix sort algorithm. Sort the following numbers in ascending order using radix sort.
12, 8, 25, 4, 66, 2, 98, 225.

Sorting of Given Numbers: -
Pass 1:

Bucket Elements	0	1	2	3	4	5	6	7	8	9
12			12							
8									8	
25						25				
4					4					
66							66			
2			2							
98									98	
225						225				

Output of Pass 1: 12 02, 04, 25, 225, 66, 8, 98

Pass 2:

Bucket Elements	0	1	2	3	4	5	6	7	8	9
12		12								
2	2									
4	4									
25			25							
225			225							
66							66			
8	8									
98										98

Output of Pass 2: 02, 04, 08, 12, 25, 225, 66, 98

Pass 3:

Bucket Elements	0	1	2	3	4	5	6	7	8	9
2	2									
4	4									
8	8									
12	12									
25	25									
225			225							
66	66									
98	98									

***Sorted list*: 2, 4, 8, 12, 25, 66, 98, 225**

❖ Quick Sort:

- Quick sort is the one of the type of sorting algorithm.
- Quick sort is the fastest internal sorting algorithm.
- It is an unstable sorting algorithm.
- Quick sort is an algorithm of the divide and conquer type.
- In quick sort, we select the pivot for partitioning the given elements. We will choose the first element of the lists as pivot.
- On the basis of pivot value, list is divided into left partition and right partition.
- **Time complexity:**
  Best Case => **O(n log n)**
  Average Case => **O(n log n)**
  Worst Case => **O(n^2)**

- **Quick Sort Algorithm:**

  Step 1: Use two index variables i & j with initial values of 1st index position & n-1 index position respectively.

  Step 2: Compare ith index with pivot element till it finds greater number than pivot element. Increment ith by 1 if ith element is less than pivot element

  Step 3: jth element is compared with pivot element till it finds a number less than pivot element Decrement jth by 1 if jth element is greater than pivot element.

  Step 4: After finding elements greater than & less than pivot elements interchange both the elements.

  Step 5: After interchange again increment & decrement current position of i& j, & compare each element with pivot element.

  Step 6: once all elements are compared with pivot element fix the final position of pivot element in the list.

  Step 7: Divide the total array into 2 parts without including fixed position element.

  Step 8: Each part then should be sorted with the same procedure as mentioned above till you get a sorted array.

- **Example:**

  Refer your class notebook.

VJTECH Academy

- **Advantages:**
  - Efficient average case as compared to any other sorting algorithm.
  - It is faster than other algorithm such as bubbles sort, selection sort and insertion sort.
- **Disadvantages:**
  - It is complex and massively recursive.
  - Sorting method is not stable.
  - It can not be used for the external sorting.
  - The worst-case complexity of quick sort is O(n^2), which is worse than the other sorting algorithms.

❖ Linear Search:

- Linear search is the one of the type of searching algorithm where key element is search sequentially in the given list.
- Linear search is also called as “sequential search”.
- Search key element is compared with each element in list
- In this method, searching the key element in given list is begin with the first element and end with the last element.
- Linear search is too slow to be used with large lists due to its O(n) average case performance.
- Efficient method for less no. of elements.
- Example:

0	1	2	3	4
3	21	11	91	19

- In above example, we are searching for element 91. To search 91 the item 91 is compared with element at A[0] then A[1] and so on.
- Until we find the key value or reach to the end of array.
- When item is found it displays the “Element found” message else displays “Element not found”.
- **Limitation:** Highly inefficient for large no of data.
- The time complexity of linear search is **O(n).**

- **Algorithm for Linear Search:**

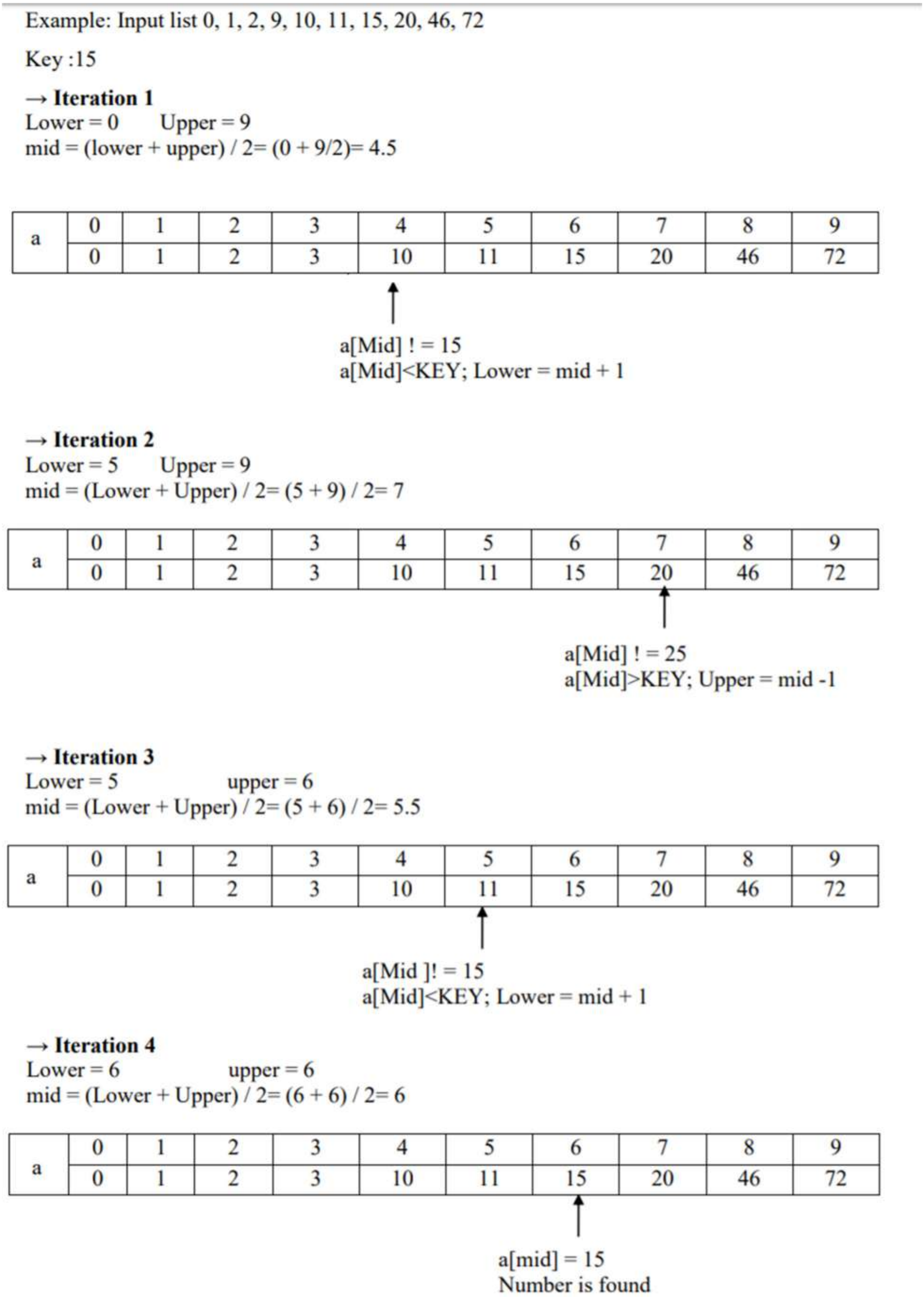
```
1. Start
2. Accept n values from user i.e. array elements.
3. Accept element to be searched from user i.e. key element.
4. Set i=0,flag=0
5. Compare A[i] with key
    If(A[i] is a key)
          Set flag=1 & go to step 7
     else
          Move to next data element i= i+1;
6. If (i<=n) go to step 5
7. If(flag=1) then
          print “Element is found”
    Else
          print “Element is not”
8. Stop.
```

- **Write a C Program to implement Linear search.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a[10],i,key,n,flag=0;
        clrscr();
        printf("\nEnter size of array:");
        scanf("%d",&n);
        printf("\nEnter array elements are:");
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
        }
        printf("\nEnter key element to be search:");
        scanf("%d",&key);
        for(i=0;i<n;i++)
        {
                for(key==a[i])
                {
                        flag=1;
                        break;
                }
        }
        if(flag==1)
        {
                printf(“\n Element is found”);
        }
        else
        {
                printf(“\n Element is not found”);
        }
        getch();
}
```

❖ Binary Search:

- Binary search can be performed only on sorted array.
- First find the lower index and upper index of an array and calculate mid index with the formula (lower + upper)/2.
- Compare the mid position element with the search key element.
- If both are equal then stop the search process. If both are not equal then divide list into two parts.
- If the search element is less than mid position element then change the upper index value and use first half of the list for further searching process.
- If the search element is greater than mid position element than change the lower index value and use second half of the list for further searching process.
- Again find lower and upper index value and calculate mid value.
- Repeat the process till element is found or lower index value is less than or equal to upper index value.
- The time complexity of binary search is **O(log n)**.



Example: Input list 0, 1, 2, 9, 10, 11, 15, 20, 46, 72

Key :15

→ **Iteration 1**

Lower = 0 Upper = 9

mid = (lower + upper) / 2= (0 + 9/2)= 4.5

a	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	10	11	15	20	46	72

a[Mid] ! = 15

a[Mid]<KEY; Lower = mid + 1

→ **Iteration 2**

Lower = 5 Upper = 9

mid = (Lower + Upper) / 2= (5 + 9) / 2= 7

a	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	10	11	15	20	46	72

a[Mid] ! = 25

a[Mid]>KEY; Upper = mid -1

→ **Iteration 3**

Lower = 5 upper = 6

mid = (Lower + Upper) / 2= (5 + 6) / 2= 5.5

a	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	10	11	15	20	46	72

a[Mid]! = 15

a[Mid]<KEY; Lower = mid + 1

→ **Iteration 4**

Lower = 6 upper = 6

mid = (Lower + Upper) / 2= (6 + 6) / 2= 6

a	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	10	11	15	20	46	72

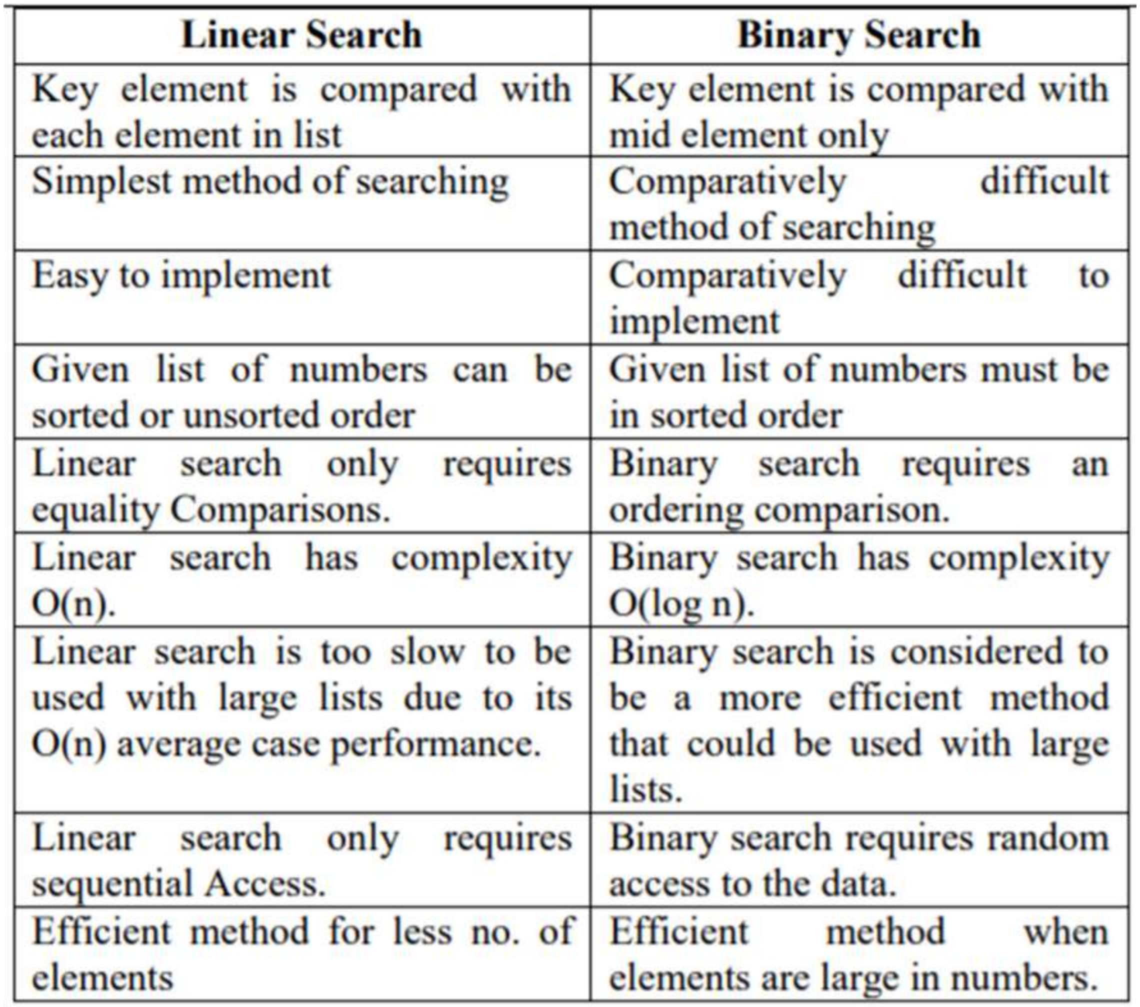
a[mid] = 15

Number is found

- **Write a C Program to implement Binary search.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a[10],i,j,n,mid,key,temp;
        clrscr();
        printf("\nEnter size of array:");
        scanf("%d",&n);
        printf("\nEnter array elements:");
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
        }
        for(i=1;i<n;i++)
        {
                for(j=0;j<n-i;j++)
                {
                        if(a[j]>a[j+1])
                        {
                                temp=a[j];
                                a[j]=a[j+1];
                                a[j+1]=temp;
                        }
                }
        }
        printf("\nEnter key element to be search:");
        scanf("%d",&key);
        i=0;
        j=n-1;
        mid=(i+j)/2;
        while(key!=a[mid] && i<=j)
        {
                if(key>a[mid])
                {
                        i=mid+1;
                }
                else
                {
                        j=mid-1;
                }
                mid=(i+j)/2;
        }

        if(i<=j)
                printf("\n Element is found!!!");
        else
                printf("\n Element is not found!!!");

        getch();
}
```

- **Differences between linear search and binary search:**



Linear Search	Binary Search
Key element is compared with each element in list	Key element is compared with mid element only
Simplest method of searching	Comparatively difficult method of searching
Easy to implement	Comparatively difficult to implement
Given list of numbers can be sorted or unsorted order	Given list of numbers must be in sorted order
Linear search only requires equality Comparisons.	Binary search requires an ordering comparison.
Linear search has complexity O(n).	Binary search has complexity O(log n).
Linear search is too slow to be used with large lists due to its O(n) average case performance.	Binary search is considered to be a more efficient method that could be used with large lists.
Linear search only requires sequential Access.	Binary search requires random access to the data.
Efficient method for less no. of elements	Efficient method when elements are large in numbers.

❖ Important Questions:

1. Differentiate between Binary Search and Sequential Search **4 Marks**
2. Find the position of element 29 using binary search method in an array 'A' given below : A = {11, 5, 21, 3, 29, 17, 2, 43} **4 Marks**
3. Write an algorithm for Selection sort. **4 Marks**
4. Explain the working of Radix Sort Method with an example **4 Marks**
5. Sort the following numbers in ascending order using Insertion sort. Given Numbers : 348, 14, 614, 5381, 47 and Write the output after each iteration . **4 Marks**

6. Sort the following numbers in ascending order using Bubble sort. Given Numbers : 348, 14, 614, 5381, 47 and Write the output after each iteration. **4 Marks**

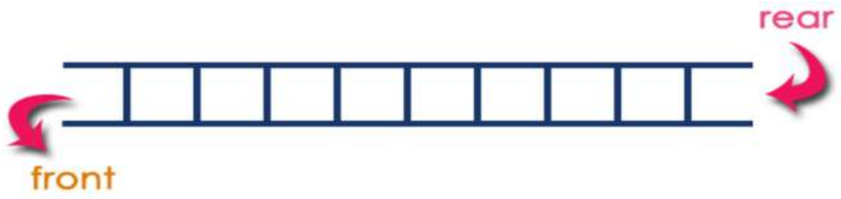
7. Write a program for sorting the array of 10 elements using the Bubble sort method.

8. Describe Binary search with an example. **4 Marks**

9. WAP to implement linear search for 10 elements in an array. **4 Marks**

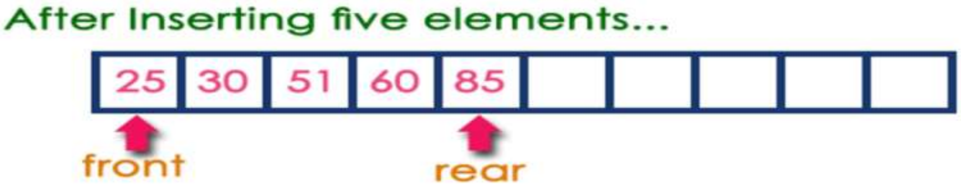
❖ Queue:

- Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends.
- Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO or LILO manner.
- In a queue data structure, adding and removing of elements are performed at two different positions.
- The insertion is performed at one end and deletion is performed at other end.
- In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'.
- In queue data structure, the insertion and deletion operations are performed based on FIFO (**First In First Out**) or LILO (**Last In Last Out**) manner.
- Below diagram shows the behavior of a stack:



- In the above figure, insertion and deletion operations are performed at two different end of the queue.
- **Rear** end is used for inserting an new element and **Front** end is used for deleting an element from the queue.
- **Front Variable:** While implementing the queue data structure, front variable is important which is an integer type. Front is a variable which hold the index of the element that is to be deleted. The initial value of front variable is -1.
- **Rear Variable:** While implementing the queue data structure, rear variable is important which is an integer type. Rear is a variable which hold the index of the element that has been inserted. The initial value of rear variable is -1.

- **Example:**
  If we want to create a queue by inserting 25,30,51,60 and 85. Then rear points to 85 element and front point to 25 element of array as mentioned in below:



❖ **Operations on a Queue:** The following operations are performed on the Queue.

1. **Initialize –** To initialize the queue, in this operation -1 value set to rear and front variables.
2. **Empty –** To check whether queue is empty or not.
3. **Full –** To check whether the queue is full or not.
4. **Insertion–** To insert the new element into the queue using rear end.
5. **Deletion –** To delete the element from the queue using front end.
6. **Print –** To display all elements of the queue.

Queue Overflow Condition: When queue is already full even if you are going to perform the insertion operation on the queue then Overflow condition will occurred.

Queue Underflow Condition: When queue is already full even if you are going to perform the deletion operation on the queue then Underflow condition will occurred.

❖ **Queue data structure can be implemented in two ways. They are as follows.**

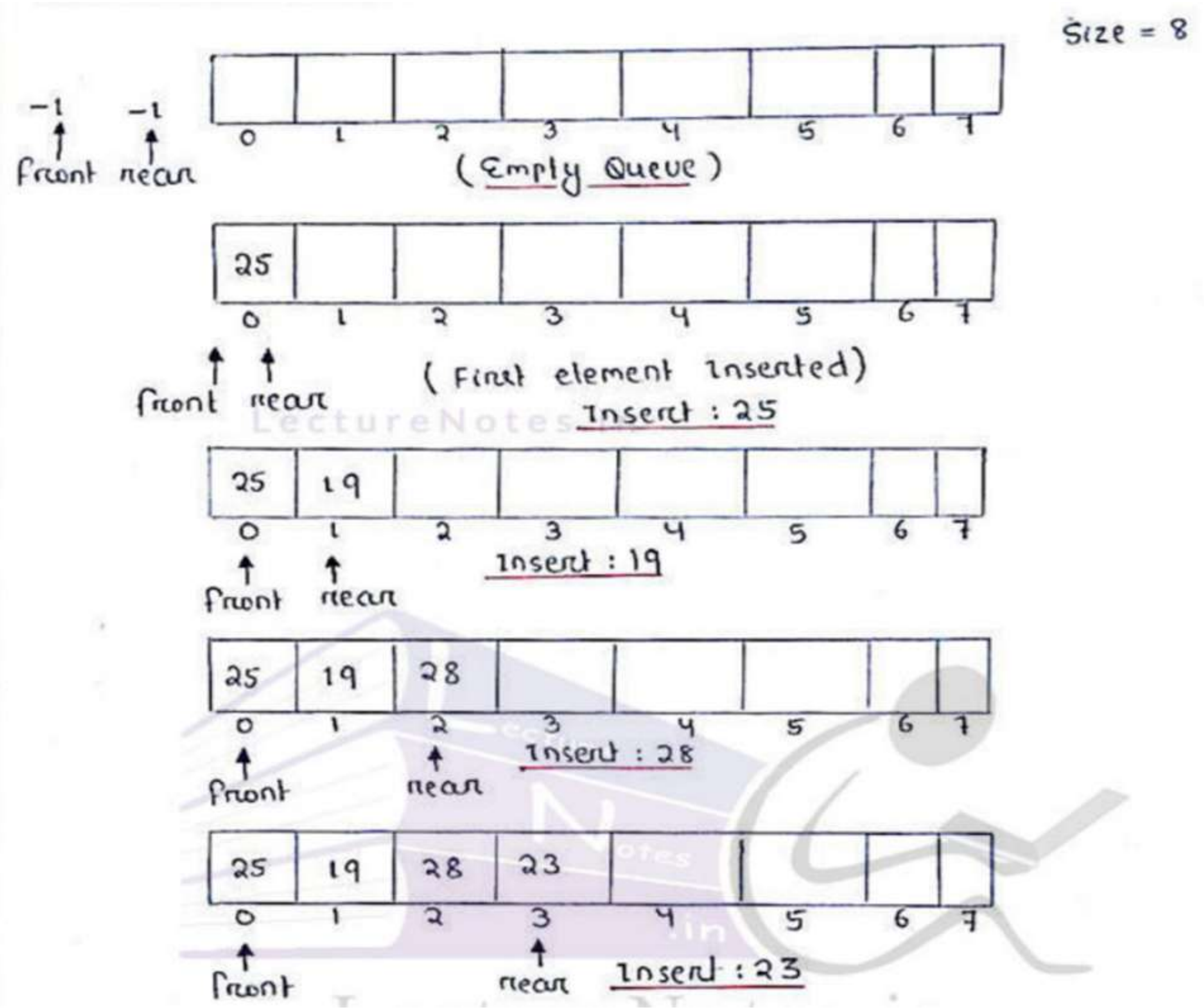
1. Using Array
2. Using Linked List

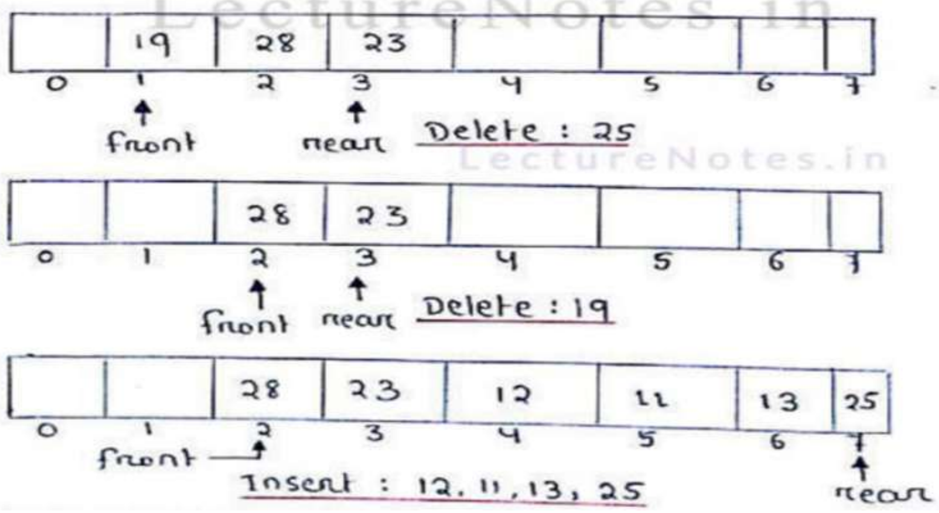
When a queue is implemented using arrays, that queue can organize only a limited number of elements. When a queue is implemented using a linked list, that queue can organize an unlimited number of elements.

❖ Queue Representation Using Array:

- A queue data structure can be implemented using one dimensional array.
- But queue implemented using array, can store only fixed number of data values.
- This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using FIFO or LILO manner with the help of variables rear and front.
- Initially, rear and front both are set to -1.
- Whenever we want to insert a value into the queue, increment the rear value by one and then insert at that position.
- Whenever we want to delete a value from the queue, then increment 'front' value by one and then display the value at 'front' position as deleted element.

- Some Insertion and Deletion operations on queue:





❖ Queue Operations using Array:

1. Initialize() : In this operation, we can initialize the queue. Here actually, we set -1 value to rear and front variables.

```
void initialize()

{

   rear=front=-1;

}
```

2. Empty() : In this operation, we can check whether queue is empty or not. If rear value is equal to -1 then it indicates queue is empty.

C Function:

```
int empty()
{
   if (rear==-1)

       return(1);

   else

       return(0);
}
```

3. Full() : In this operation, we can check whether queue is full or not. If rear value is equal to array size (MAX-1) then it indicates queue is full.

C Function:

```
int full()
{
   if (rear==MAX-1)

       return(1);
   else
       return(0);
}
```

4. Insertion() : In a queue, insertion() is a function used to insert an element into the queue. The new element is always inserted at rear end. Before inserting a new element into the queue first needs to be check whether queue is full or not if queue is not full then and then only we can insert new element otherwise we need to print “Queue is Full” message. Below steps are used to insert a new element into the queue.

Algorithm:

```
Step 1: Start
Step 2:   if(rear == SIZE-1)
                    print “queue is Full”
          otherwise
                    goto Step 2.
Step 3: If rear == -1 then
                i) Assign 0 to rear and front variable.
                ii) Insert new element into queue i.e data[rear]=x;
        Otherwise
                i) Increment rear variable by one i.e rear=rear+1
                ii) Insert new element into queue  i.e data[rear]=x;
Step 5: Stop
```

C Function:

```
void insertion()
{
        int x;
        printf("Enter data to be insert: ");
        scanf("%d",&x);
        if(rear==-1)
        {
                rear=front=0;
                data[rear]=x;
        }
        else
        {
                rear=rear+1;
            data[rear]=x;

   }
}
```

5. Deletion() : In a queue, deletion() is a function used to delete an element from the queue. The element is always deleted from front end. Before deleting a new element from the queue first needs to be check whether queue is empty or not if queue is not empty then and then only we can delete an element from the queue otherwise we need to print “Queue is Empty” message. Below steps are used to delete an element from the queue.

Algorithm:

```
Step 1: Start
Step 2: Delete front element from the queue i.e X=data[front]
Step 3: if(front == rear) then
                set -1 to rear and front variables.
        otherwise
                goto Step 4.
Step 4: Increment front by 1 i.e. front=front+1
Step 5: Print deleted value is X.
Step 6: Stop.
```

C Function:

```
void deletion()
{
        int x;
        x=data[front];
        if(front==rear)
        {
                rear=front=-1;
        }
        else
        {
                front=front+1;
        }
    printf("\nDeleted data= %d",x);
}
```

6. Print() : In this operation, We will display all the elements of the queue:

Algorithm:

```
Step 1: Start

Step 2:   if(front == -1)
                    print “Queue is Empty”
          otherwise
                    goto Step 3.

Step 3: Initialise variable 'i' with front.

Step 4: Display data[i] value and increment i value by one.

Step 4: Repeat above step 4 until i value less than or equals to rear.

Step 5: Stop.
```

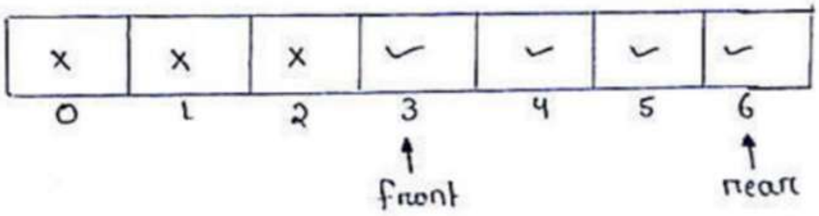
C Function:

```
void print()
{
        int i;
        if(front==-1)
        {
                printf("\nQueue is Empty!!!");
        }
        else
        {
                printf("\nQueue data: ");
                for(i=front;i<=rear;i++)
                {
                        printf("%d\t",data[i]);
                }
        }
}
```

==

Assignment : What are the limitations of a queue ?

Sol^{n} Consider the queue as given bellow.



Here, 3-elements have been deleted. Now 4-elements remain in the queue. rear is pointing to the last element.

Size of the queue = 7

Now if we will go for inserting an element in the queue then overflow will arise, because here rear = Size - 1.

But there exist free memory locations i.e. in the index position 0, 1, 2. Even having free memory locations, no element can be inserted in the queue.

This is the limitation having in a linear queue.

Assignment : What are the applications of a queue ?

Sol^{n} The applications of queue are :

<1> Round robin technique for processor Scheduling is implemented using queues.

<2> All types of customer services.

<3> Printing jobs in a printer (shared one) connected to the LAN (Local Area Network).

LectureNotes.in

/* Write a Queue Program which perform all operations*/

```
#include<stdio.h>
#include<conio.h>
#define MAX 10

int data[MAX],rear, front;

void initialize();
int empty();
int full();
void insertion();
void deletion();
void print();

void main()
{
   int ch;
   clrscr();
   initialize();
   do
   {
      printf("\n ***QUEUE OPERATIONS");
      printf("\n 1. Insertion");
      printf("\n 2. Deletion");
      printf("\n 3. Print");
      printf("\n 4. EXIT");
      printf("\n Enter your choice: ");
      scanf("%d", &ch);

            switch (ch)
      {
         case 1: if(full()==0)
                       {
                                    insertion();
                       }
                       else
                       {
                                    printf(“\nQueue is Full!!!”);
                       }
                break;
         case 2: if(empty()==0)
                       {
                                    deletion();
                       }
                       else
                       {
                                    printf(“\nQueue is Empty!!!”);
                       }
                break;
            case 3: print();
                              break;
            case 4: break;
            default:
                    printf("Invalid choice");
        }
    } while(ch!=4);
    getch();
}

void initialize()
{
    rear=front=-1;
}

int full()
{
    if(rear==MAX-1)
        return(1);
    else
        return(0);
}

int empty()
{
    if(rear==-1)
        return(1);
    else
        return(0);
}

void insertion()
{
                int x;
                printf("Enter data to be insert: ");
                scanf("%d",&x);
                if(rear==-1)
                {
                                rear=front=0;
                                data[rear]=x;
                }
                else
                {
                                rear=rear+1;
                        data[rear]=x;

    }
}
```

```
void deletion()
{
        int x;
        x=data[front];
        if(front==rear)
        {
                rear=front=-1;
        }
        else
        {
                front=front+1;
        }
   printf("\nDeleted data= %d",x);
}

void print()
{
        int i;
        if(front==-1)
        {
                printf("\nQueue is Empty!!!");
        }
        else
        {
                printf("\nQueue data: ");
                for(i=front;i<=rear;i++)
                {
                        printf("%d\t",data[i]);
                }
        }
}
```

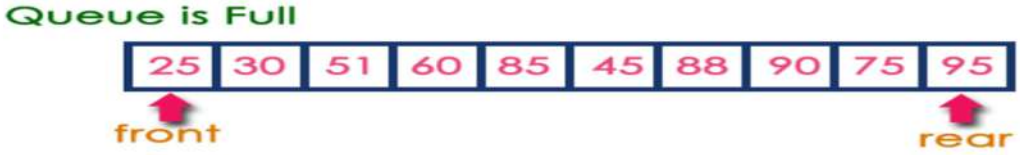
❖ Applications of Queue:

1. Queue is useful in CPU scheduling, Disk Scheduling. When multiple processes require CPU at the same time, various CPU scheduling algorithms are used which are implemented using Queue data structure.
2. When data is transferred asynchronously between two processes.Queue is used for synchronization.
3. Queue used in Breadth First search algorithm of a Graph
4. Keyboard buffer.
5. Round robin Scheduling.
6. Job scheduling.
7. In real life, Call Center phone systems will use Queues, to hold people calling them in an order.
8. To solve real life problems like railway reservation counter, movie ticket counter , college admission system peoples are present in queue.

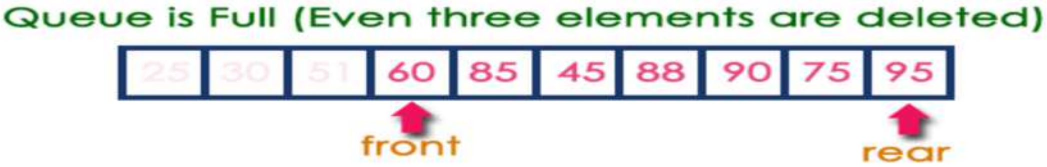
❖ Circular Queue:

- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.
- In a normal Queue Data Structure, we can insert elements until queue becomes full. But once if queue becomes full, we can not insert the next element until all the elements are deleted from the queue. For example consider the queue below:

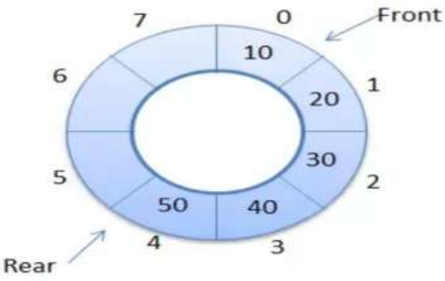
After inserting all the elements into the queue.



Now consider the following situation after deleting three elements from the queue.



- This situation also says that Queue is Full and we can not insert the new element because, 'rear' is still at last position.
- In above situation, even though we have empty positions in the queue we can not make use of them to insert new element.
- This is the major problem in normal queue data structure. To overcome this problem we use circular queue data structure.
- Graphical representation of a circular queue is as follows...



- In circular queue we can insert the new element into the queue until the all cells are not filled.

- When rear or front reached to the end of the queue then it will again come back to the first index by using the below formula:

I = (I + 1) % MAX (Suppose size of array is 5)
----
0 = (-1 + 1) % 5
1 = (0 + 1) % 5
2 = (1 + 1) % 5
3 = (2 + 1) % 5
4 = (3 + 1) % 5
0 = (4 + 1) % 5
1 = (0 + 1) % 5

- We can use above formula for incrementing/decrementing rear and front variable value in circular queue.

Operations on a Circular Queue: The following operations are performed on the circular Queue.

1. **Initialize** – To initialize the circular queue, in this operation -1 value set to rear and front variables.
2. **Empty –** To check whether circular queue is empty or not.
3. **Full –** To check whether the circular queue is full or not.
4. **Insertion–** To insert the new element into the circular queue using rear end.
5. **Deletion –** To delete the element from the circular queue using front end.
6. **Print –** To display all elements of the circular queue.

Queue Operations using Array:

1. Initialize() : In this operation, we can initialize the circular queue. Here actually, we set -1 value to rear and front variables.

```
void initialize()
{
   rear=front=-1;
}
```

2. Empty() : In this operation, we can check whether circular queue is empty or not. If rear value is equal to -1 then it indicates queue is empty.

C Function:

```
int empty()
{
   if (rear==-1)

      return(1);

   else

      return(0);
}
```

3. Full() : In this operation, we can check whether circular queue is full or not. If (rear+1)%MAX value is equal to front value then it indicates queue is full.

C Function:

```
int full()
{
   if ((rear+1)%MAX==front)

      return(1);
   else
      return(0);
}
```

4. Insertion() : In a circular queue, insertion() is a function used to insert an element into the circular queue. The new element is always inserted at rear end. Before inserting a new element into the circular queue first needs to be check whether circular queue is full or not if circular queue is not full then and then only we can insert new element otherwise we need to print “Circular Queue is Full” message. Below steps are used to insert a new element into the queue.

Algorithm:

```
Step 1: Start
Step 2:   if((rear+1)%MAX == front)
                    print “queue is Full”
          otherwise
                    goto Step 2.
Step 3: If rear == -1 then
               i) Assign 0 to rear and front variable.
               ii) Insert new element into queue i.e data[rear]=x;
        Otherwise
               i) Increment rear variable i.e  rear = ( rear + 1 ) % MAX
               ii) Insert new element into queue  i.e data[rear]=x;
Step 5: Stop
```

C Function:

```
void insertion()
{
        int x;
        printf("Enter data to be insert: ");
        scanf("%d",&x);
        if(rear==-1)
        {
                rear=front=0;
                data[rear]=x;
        }
        else
        {
                rear=(rear+1)%MAX;
             data[rear]=x;

   }
}
```

5. Deletion() : In a circular queue, deletion() is a function used to delete an element from the circular queue. The element is always deleted from front end. Before deleting a new element from the circular queue first needs to be check whether circular queue is empty or not if circular queue is not empty then and then only we can delete an element from the circular queue otherwise we need to print “Circular Queue is Empty” message. Below steps are used to delete an element from the queue.

Algorithm:

```
Step 1: Start
Step 2: Delete front element from the queue i.e X=data[front]
Step 3: if(front == rear) then
                set -1 to rear and front variables.
        otherwise
                goto Step 4.
Step 4: Increment front i.e. front = ( front + 1 ) % MAX
Step 5: Print deleted value is X.
Step 6: Stop.
```

C Function:

```c
void deletion()
{
        int x;
        x=data[front];
        if(front==rear)
        {
                rear=front=-1;
        }
        else
        {
                front=(front+1)%MAX;
        }
    printf("\nDeleted data= %d",x);
}
```

6. Print() : In this operation, We will display all the elements of the circular queue:

C Function:

```
void print()
{
        int i;
        if(front==-1)
        {
                printf("\nQueue is Empty!!!");
        }
        else
        {
                i=front;
                printf("\nQueue data: ");
                do
                {
                        printf("%d\t",data[i]);
                        i=(i+1)%MAX;
                }while(i!=rear);
                printf("%d\t",data[i]);
        }
}
```

/* Write a Circular Queue Program which perform all operations*/

```
#include<stdio.h>
#include<conio.h>
#define MAX 10

int data[MAX],rear, front;

void initialize();
int empty();
int full();
void insertion();
void deletion();
void print();

void main()
{
   int ch;
   clrscr();
   initialize();
   do
   {
      printf("\n ***Circular QUEUE OPERATIONS");
      printf("\n 1. Insertion");
```

```
        printf("\n 2. Deletion");
        printf("\n 3. Print");
        printf("\n 4. EXIT");
        printf("\n Enter your choice: ");
        scanf("%d", &ch);

                switch (ch)
        {
           case 1: if(full()==0)
                                {
                                                        insertion();
                                }
                               else
                                {
                                                        printf("\nQueue is Full!!!");
                                }
                   break;
           case 2: if(empty()==0)
                                {
                                                        deletion();
                                }
                               else
                                {
                                                        printf("\nQueue is Empty!!!");
                                }
                   break;

           case 3: print();
                          break;
           case 4: break;
           default:
                   printf("Invalid choice");
        }
    } while(ch!=4);
    getch();
}

void initialize()
{
    rear=front=-1;
}

int full()
{
    if((rear+1)%MAX==front)
        return(1);
    else
        return(0);
}
```

```
int empty()
{
   if(rear==-1)
      return(1);
   else
      return(0);
}


void insertion()
{
        int x;
        printf("Enter data to be insert: ");
        scanf("%d",&x);
        if(rear==-1)
        {
                rear=front=0;
                data[rear]=x;
        }
        else
        {
                rear=(rear+1)%MAX;
            data[rear]=x;

   }
}

void deletion()
{
        int x;
        x=data[front];
        if(front==rear)
        {
                rear=front=-1;
        }
        else
        {
                front=(front+1)%MAX;
        }
    printf("\nDeleted data= %d",x);
}

void print()
{
        int i;
        if(front==-1)
        {
                printf("\nQueue is Empty!!!");
        }
        else
        {
                i=front;
                printf("\nQueue data: ");
                do
                {
                        printf("%d\t",data[i]);
                        i=(i+1)%MAX;
                }while(i!=rear);
                printf("%d\t",data[i]);
        }
}
```

❖ Priority Queue:

- Priority queue is a one of the types of queue.
- Priority Queue is a linear data structure.
- The rules for processing the elements of priority queue are:

  1. Elements are processed based on priority.

  2. Highest priority elements are processed first before the lowest priority elements.

  3. If two elements have the same priority, they are processed in FIFO manner.

Example of priority queue:

- Hospital waiting room where emergency patient admitted first before the normal patient.
- Operating system scheduler
- Routing
- Graph algorithms like Dijkstra’s shortest path algorithm, Prim’s Minimum Spanning Tree, etc

Operations on a Priority Queue: The following operations are performed on the Priority Queue.

1. Initialize – To initialize the priority queue, in this operation -1 value set to rear and front variables.

2. Empty – To check whether priority queue is empty or not.

3. Full – To check whether the priority queue is full or not.

4. Insertion– To insert the new element into the priority queue using rear end.

5. Deletion – To delete the element from the priority queue using front end on the basis of priority.

6. Print – To display all elements of the priority queue.

Queue Operations using Array:

1. Initialize() : In this operation, we can initialize the priority queue. Here actually, we set -1 value to rear and front variables.

```
void initialize()
{
   rear=front=-1;
}
```

2. Empty() : In this operation, we can check whether priority queue is empty or not. If rear value is equal to -1 then it indicates queue is empty.

C Function:

```
int empty()
{
   if (rear==-1)

      return(1);

   else

      return(0);
}
```

3. Full() : In this operation, we can check whether priority queue is full or not. If (rear+1)%MAX value is equal to front value then it indicates queue is full.

C Function:

```
int full()
{
   if ((rear+1)%MAX==front)

      return(1);
   else
      return(0);
}
```

VJTECH Academy

4. Insertion() : In a priority queue, insertion() is a function used to insert an element into the priority queue. The new element is always inserted at rear end. Before inserting a new element into the priority queue first needs to be check whether priority queue is full or not if priority queue is not full then and then only we can insert new element otherwise we need to print “priority Queue is Full” message. Below steps are used to insert a new element into the queue.

Algorithm:

```
Step 1: Start
Step 2:   if((rear+1)%MAX == front)
                  print “queue is Full”
          otherwise
                  goto Step 2.
Step 3: If rear == -1 then
                i) Assign 0 to rear and front variable.
                ii) Insert new element into queue i.e data[rear]=x;
        Otherwise
                i) Increment rear variable i.e  rear = ( rear + 1 ) % MAX
                ii) Insert new element into queue  i.e data[rear]=x;
Step 5: Stop
```

C Function:

```c
void insertion()
{
        int x;
        printf("Enter data to be insert: ");
        scanf("%d",&x);
        if(rear==-1)
        {
                rear=front=0;
                data[rear]=x;
        }
        else
        {
                rear=(rear+1)%MAX;
            data[rear]=x;

    }
}
```

5. Deletion() : In a priority queue, deletion() is a function used to delete an element from the priority queue. The element is always deleted from front end. Before deleting a new element from the priority queue first needs to be check whether priority queue is empty or not if priority queue is not empty then and then only we can delete an element from the priority queue otherwise we need to print “priority Queue is Empty” message. Below steps are used to delete an element from the queue.

Algorithm:

```
Step 1: Start
Step 2: Delete front element from the queue i.e X=data[front]
Step 3: if(front == rear) then
                set -1 to rear and front variables.
        otherwise
                goto Step 4.
Step 4: Increment front i.e. front = ( front + 1 ) % MAX
Step 5: Print deleted value is X.
Step 6: Stop.
```

C Function:

```c
void deletion()
{
        int x;
        x=data[front];
        if(front==rear)
        {
                rear=front=-1;
        }
        else
        {
                front=(front+1)%MAX;
        }
    printf("\nDeleted data= %d",x);
}
```

6. Print() : In this operation, We will display all the elements of the priority queue:

C Function:

```
void print()
{
        int i;
        if(front==-1)
        {
                printf("\nQueue is Empty!!!");
        }
        else
        {
                i=front;
                printf("\nQueue data: ");
                do
                {
                        printf("%d\t",data[i]);
                        i=(i+1)%MAX;
                }while(i!=rear);
                printf("%d\t",data[i]);
        }
}
```

/* Write a priority Queue Program which perform all operations*/

```
#include<stdio.h>
#include<conio.h>
#define MAX 10

int data[MAX],rear, front;

void initialize();
int empty();
int full();
void insertion();
void deletion();
void print();

void main()
{
  int ch;
  clrscr();
  initialize();
  do
  {
    printf("\n ***Circular QUEUE OPERATIONS");
    printf("\n 1. Insertion");
    printf("\n 2. Deletion");
    printf("\n 3. Print");
    printf("\n 4. EXIT");
    printf("\n Enter your choice: ");
    scanf("%d", &ch);
```

VJTECH Academy

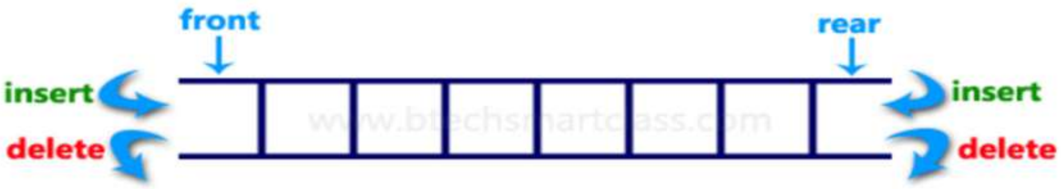
```
            switch (ch)
      {
         case 1: if(full()==0)
                        {
                                        insertion();
                        }
                       else
                        {
                                        printf("\nQueue is Full!!!");
                        }
                 break;
         case 2: if(empty()==0)
                        {
                                        deletion();
                        }
                       else
                        {
                                        printf("\nQueue is Empty!!!");
                        }
                 break;

         case 3: print();
                     break;
         case 4: break;
         default:
                printf("Invalid choice");
      }
   } while(ch!=4);
   getch();
}

void initialize()
{
   rear=front=-1;
}

int full()
{
   if((rear+1)%MAX==front)
      return(1);
   else
      return(0);
}

int empty()
{
   if(rear==-1)
      return(1);
   else
```

```
    return(0);
}


void insertion()
{
        int x;
        printf("Enter data to be insert: ");
        scanf("%d",&x);
        if(rear==-1)
        {
                rear=front=0;
                data[rear]=x;
        }
        else
        {
                rear=(rear+1)%MAX;
            data[rear]=x;

  }
}

void deletion()
{
        int x;
        x=data[front];
        if(front==rear)
        {
                rear=front=-1;
        }
        else
        {
                front=(front+1)%MAX;
        }
   printf("\nDeleted data= %d",x);
}

void print()
{
        int i;
        if(front==-1)
        {
                printf("\nQueue is Empty!!!");
        }
        else
        {
                i=front;
                printf("\nQueue data: ");
                do
                {
```

```
                                        printf("%d\t",data[i]);
                                        i=(i+1)%MAX;
                              }while(i!=rear);
                              printf("%d\t",data[i]);
                 }
}
```

❖ <u>Double Ended Queue:</u>

- Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear).
- We can insert at both front and rear positions and can delete from both front and rear positions.



<u>Operations on a Double ended Queue:</u> The following operations are performed on the DQueue.

1. Initialize – To initialize the DQueue, in this operation -1 value set to rear and front variables.

2. Empty – To check whether DQueue is empty or not.

3. Full – To check whether the DQueue is full or not.

4. Insertion at Rear – To insert the new element into the DQueue at rear end.

5. Insertion at Front – To insert the new element into the DQueue at front end.

6. Deletion from Front– To delete the element from the DQueue at front end.

7. Deletion from Rear– To delete the element from the DQueue at rear end.

6. Print – To display all elements of the Dqueue

Double Ended Queue can be represented in TWO ways, those are as follows...

1. Input Restricted Double Ended Queue

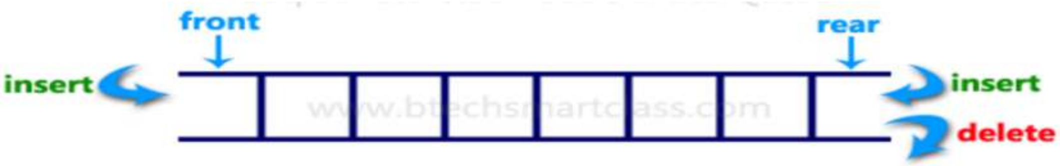
2. Output Restricted Double Ended Queue

1. Input Restricted Double Ended Queue:

i) Insertion at Rear end

ii) Deletion from Rear end

iii) Deletion from Front end

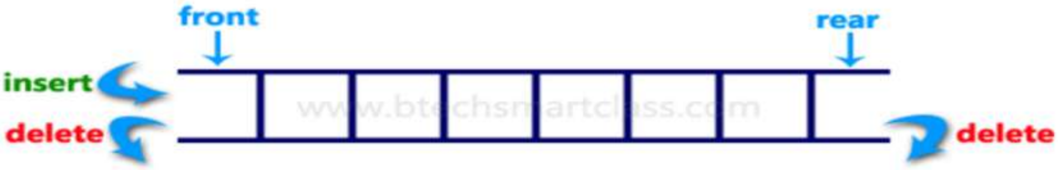


2. Output Restricted Double Ended Queue:

i) Insertion at Front end

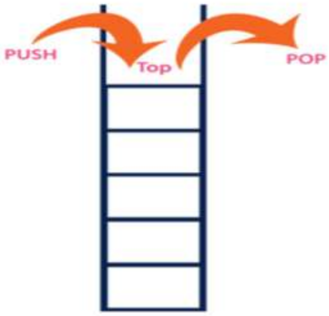
ii) Deletion from Front end

iii) Insertion at Rear end



❖ Stack:

- Stack is a linear data structure in which the operations are performed based on LIFO or FILO principle.
- A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO or FILO principle.
- In a stack, adding and removing of elements is performed at single position which is known as "Top".
- That means, new element is added at the top of the stack and an element is removed from the top of the stack.
- In stack, the insertion and deletion operations are performed based on LIFO (**Last In First Out** and **First In Last Out**) manner.
- Below diagram shows the behavior of a stack:



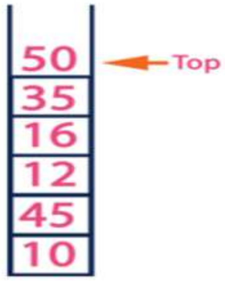
- In a stack, the insertion operation is performed using a function called "push" and deletion operation is performed using a function called "pop".
- In the above figure, PUSH and POP operations are performed at top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e at Top)
- **Top Variable:** While implementing the stack data structure, Top variable is important. It is an integer variable which holds the index of the topmost elements of the stack. The initial value of Top variable is

  -1.
- **General Example of stack:**

  1. Plates placed on the counter of the cafeteria.
  2. Books on the table one after another.

- **Example:**

  If we want to create a stack by inserting 10,45,12,16,35 and 50. Then 10 becomes the bottom most element and 50 is the top most element. Top is at 50 as shown in the image below:



❖ **Operations on a Stack:** The following operations are performed on the stack.

1. Initialize – To initialize the stack, in this operation -1 value set to Top variable.

2. Empty – To check whether stack is empty or not.

3. Full – To check whether the stack is full or not.

4. Push – To insert the element onto the stack is known as Push operation.

5. Pop – To delete the element from the stack is known as Pop operation.

6. Print – To display all elements of the stack.

❖ **Stack data structure can be implemented in two ways. They are as follows.**

1. Using Array/Static Representation

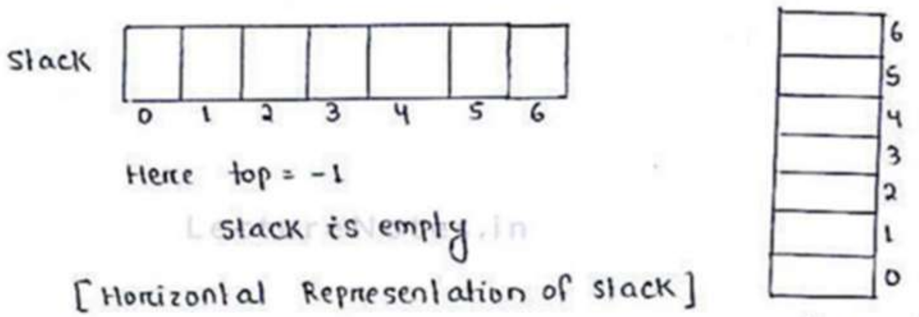
2. Using Linked List/ Dynamic Representation

When a stack is implemented using arrays, that stack can organize only a limited number of elements. When a stack is implemented using a linked list, that stacks can organize an unlimited number of elements.

❖ Stack Representation Using Array

- A stack data structure can be implemented using one dimensional array.
- But stack implemented using array, can store only fixed number of data values.
- This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using LIFO principle with the help of a variable 'top'.
- Initially, top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert.
- Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

→ Consider the array named "stack" having size = 7 bellow.



Here top = -1

stack is empty

[Horizontal Representation of stack]

top = -1

stack is empty.

[vertical Representation]

<u>Stack Overflow Condition</u>: When stack is already full even if you are going to perform the push opeartion on the stack then Overflow condition will occurred.

<u>Stack Underflow Condition</u>: When stack is already Empty even if you are going to perform the Pop opeartion on the stack then Underflow condition will occurred.

Stack Operations using Array:

1. Initialize() : In this operation, we can initialize the stack. Here actually, we set -1 value to Top variable.

```
void initialize()

{

    top=-1;

}
```

2. Empty() : In this operation, we can check whether stack is empty or not. If top value is equal to -1 then it indicates stack is empty.

C Function:

```
int empty()
{
    if (top==-1)

        return(1);

    else

        return(0);
}
```

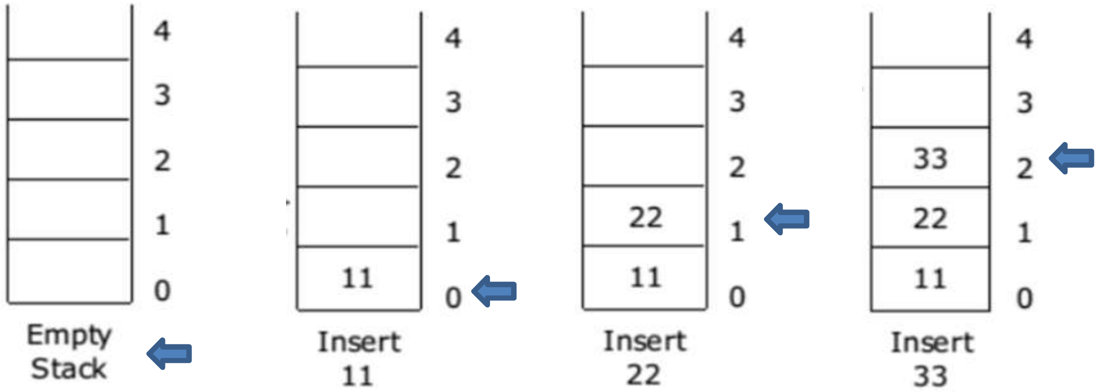
3. Full() : In this operation, we can check whether stack is full or not. If top value is equal to array size (SIZE-1) then it indicates stack is full.

C Function:

```
int full()
{
    if (top==MAX-1)

        return(1);

    else

        return(0);
}
```

VJTECH Academy

4. Push() : In a stack, push() is a function used to insert an element into the stack. The new element is always inserted at top position. Below steps are used to push a new element onto the stack.



a) Stack is empty
i.e top=-1

b) Insert 11
i.e top=0

c) Insert 22
i.e top=1

d) Insert 33
i.e top=2

Algorithm:

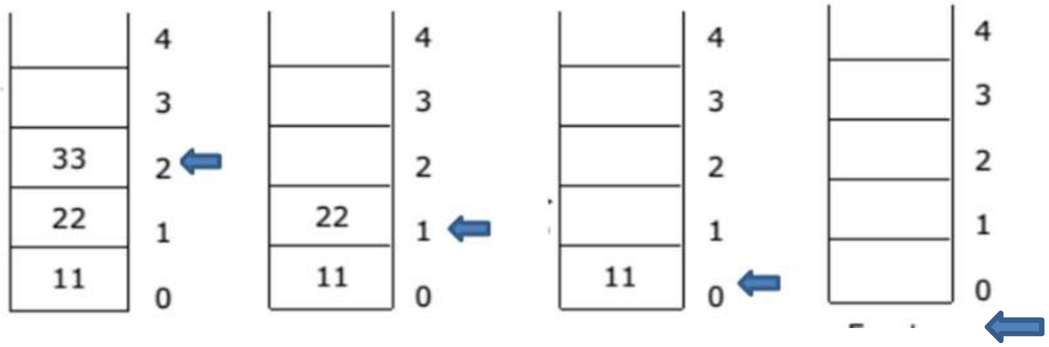
```
        Step 1: Start
        Step 2:   if(top == SIZE-1)
                        print “Stack is Full”
                  otherwise
                        goto Step 2.
        Step 3: Increment top variable i.e top=top+1
        Step 4: Push new element onto stack i.e data[top]=x;
        Step 5: Stop
```

C Function:

```
void push()
{
    int x;
    printf("Enter data to be insert: ");
    scanf("%d",&x);
    top=top+1;
    data[top]=x;
}
```

VJTECH Academy

5. Pop() : In a stack, pop() is a function used to delete an element from the stack. The element is always deleted from top position. Below steps are used to pop an element from the stack.



a) Initially stack i.e top=2

b) delete 33 i.e top=1

c) delete 22 i.e top=0

d) delete 11 i.e top=-1

Algorithm:

```
Step 1: Start
Step 2:   if(top == -1)
                        print “Stack is Empty”
            otherwise
                        goto Step 2.
Step 3: delete top value from stack i.e.  X=data[top]
Step 4: decrement top by 1 i.e. top=top-1
Step 5: Print deleted value is X.
Step 5: Stop.
```

C Function:

```
void pop()
{
      int x;
      x=data[top];
      top=top-1;
      printf("\nData popped= %d",x);
}
```

2. Print() : In this operation, We will display all the contents of the stack:

Algorithm:

```
Step 1: Start

Step 2:   if(top == -1)

                    print "Stack is Empty"

        otherwise
                    goto Step 2.

Step 3: Initialise variable 'i' with top. Display data[i] value and decrement i value by one.

Step 4: Repeat above step 3 until i value becomes '0'.

Step 5: Stop.
```

C Function:

```c
void print()
{
        int i;
        if(top==-1)
        {
                printf("\nStack is Empty!!!");
        }
        else
        {
                printf("\nStack data: ");
                for(i=top;i>=0;i--)
                {
                        printf("%d\t",data[i]);
                }
        }
}
```

VJTECH Academy

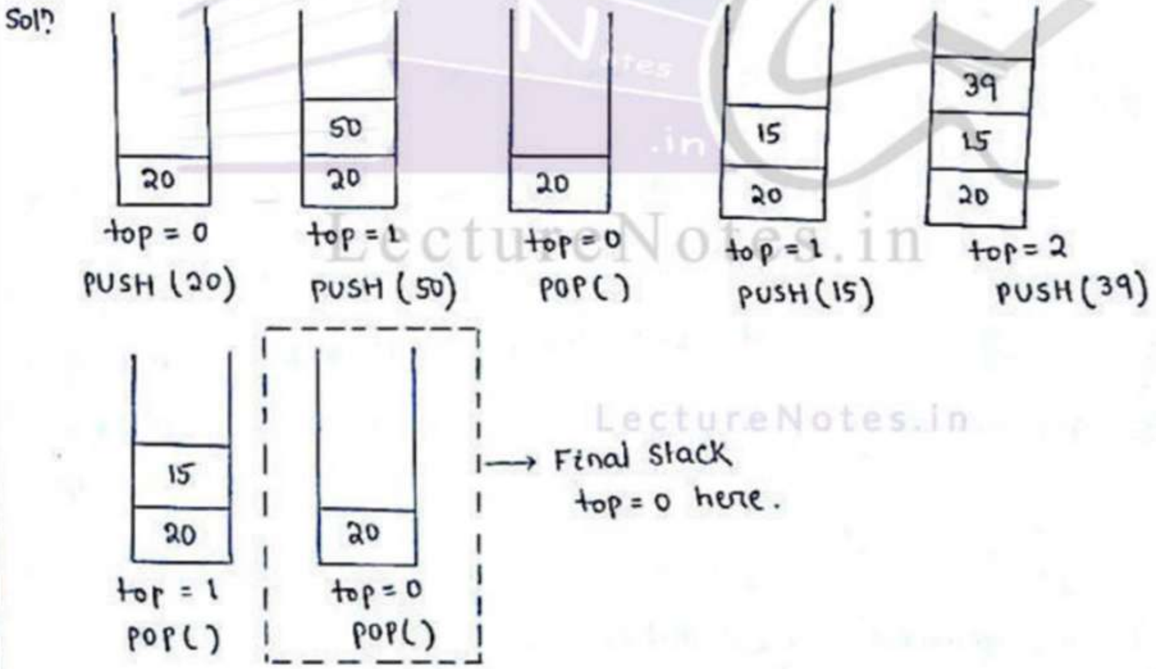
==

Assignment : Consider the following operations in a stack

1. PUSH (20) 2. PUSH (50) 3. POP() 4. PUSH (15) 5. PUSH(39)
6. POP() 7. POP()

After these operations find out the elements in the stack and if top = -1 initially then what will be the top value?

Soln



→ Final Stack
top = 0 here.

==

LectureNotes.in

/* Write a Stack Program which perform all operations*/

```
#include<stdio.h>
#include<conio.h>
#define MAX 10

int data[MAX],top;

void initialize();
int empty();
int full();
void push();
void pop();
void print();

void main()
{
    int ch;
    clrscr();
    initialize();
    do
    {
        printf("\n ***STACK OPERATIONS");
        printf("\n 1. PUSH");
        printf("\n 2. POP");
        printf("\n 3. Print");
        printf("\n 4. EXIT");
        printf("\n Enter your choice: ");
        scanf("%d", &ch);

        switch (ch)
        {
            case 1: if(full()==0)
                     {
                              push();
                     }
                    else
                     {
                              printf(“\nStack is Full!!!”);
                     }
                     break;
            case 2: if(empty()==0)
                     {
                              pop();
                     }
                    else
                     {
```

```
                              printf(“\nStack is Empty!!!”);
                     }
                     break;

            case 3: print();
                     break;
            case 4: break;
            default:
                    printf("Invalid choice");
        }
    } while(ch!=4);
    getch();
}

void initialize()
{
    top=-1;
}

int full()
{
    if (top==MAX-1)
        return(1);
    else
        return(0);
}

int empty()
{
    if (top==-1)
        return(1);
    else
        return(0);
}

void push()
{
    int x;
    printf("Enter data to be insert: ");
    scanf("%d",&x);
    top=top+1;
    data[top]=x;
}
```

```
void pop()
{
        int x;
        x=data[top];
        top=top-1;
       printf("\nData popped= %d",x);
}

void print()
{
        int i;
        if(top==-1)
        {
                printf("\nStack is Empty!!!");
        }
        else
        {
                printf("\nStack data: ");
                for(i=top;i>=0;i--)
                {
                        printf("%d\t",data[i]);
                }
        }
}
```

==

❖ <u>Applications of STACK:</u>

1. Expression Evaluation.
2. Expression Conversion.
    a) Infix to Postfix
    b) Infix to Prefix
    c) Postfix to Infix
    d) Prefix to Infix
3. Simulation of recursion.
4. Function call.
5. Reversing the list.
6. Parsing.

==

❖ What is an Expression?

- Expression is a collection of operands and operators.
- Operator is a symbol which indicate operation to be perform like arithmetic operation or logical operation or conditional operation etc.
- Operands are the values on which operation to be perform.

Expression Types:

1. Infix Expression

2. Postfix Expression

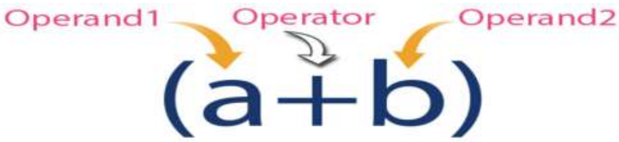
3. Prefix Expression

1. Infix Expression: In infix expression, operator is used in between operands.

Syntax:

Operand1 Operator Operand2

Example:

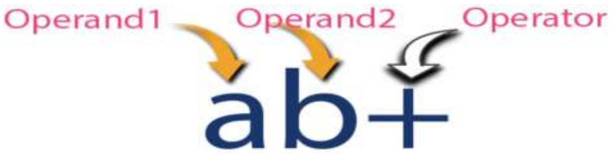


2. Postfix Expression: In postfix expression, operator is used after operands.

Syntax:

Operand1 Operand2 Operator

Example:



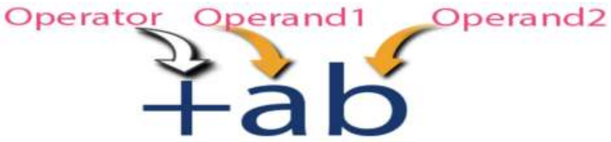
VJTECH Academy

3. Prefix Expression: In prefix expression, operator is used before operands.

Syntax:

Operator Operand1 Operand2

Example:



Any expression can be represented using the above three different types of expressions. And we can convert an expression from one form to another form like Infix to Postfix, Infix to Prefix and vice versa.

==

❖ Infix to Postfix Conversion using Stack Data Structure:

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps.

Step 1: Start

Step 2: Read all the symbols one by one from left to right in the given Infix Expression.

Step 3: If the reading symbol is operand, then directly print in the result (Output).

Step 4: If the reading symbol is left parenthesis '(', then Push it on to the Stack.

Step 5: If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.

Step 6: If reading symbol is operator whose precedence is lower or equal than current stack operator then pop operators from stack and print in the result.

Step 7: Repeat step 3 to Step 6 untill input expression End.

Step 8: Stop

VJTECH Academy

Example:

Convert A+(B*C) into Postfix form.

Input Expression	Stack	Output
A	Empty	A
+	+	A
(	+ (	A
B	+ (	A B
*	+ (*	A B
C	+ (*	A B C
)	+	A B C *
End	NULL	A B C * +

Postfix Expression: ABC*+

===

❖ Infix to Prefix Conversion using Stack Data Structure:

To convert Infix Expression into Prefix Expression using a stack data structure, We can use the following steps.

Given input Infix Expression: A+(B*C)

Step 1: Reverse the given infix expression.

)C*B(+A

Step 2: Make every ‘(‘ as ‘)’ and every ‘)’ as ‘(‘.

(C*B)+A

Step 3: Convert Step 2 expression into Postfix form.

Input Expression	Stack	Output
(	(	Empty
C	(	C
*	(*	C
B	(*	C B
)	NULL	C B *
+	+	C B *
A	+	C B * A
End	NULL	C B * A +

Step 4: Reverse the postfix expression.

Prefix Expression: + A * B C

==

❖ **Postfix Expression Evaluation:** Below steps are required to evaluate postfix expression.

Step 1: Scan postfix expression from left to right

Step 2: If reading symbol is operand then PUSH it on to stack.

Step 3: If reading symbol is operator then POP two values from stack and perform the operations between them and again push result back onto stack.

Note: First pop value become second operand and second pop value become first operand in expression.

Step 4: Repeat above all steps until input expression End.

==

❖ **Prefix Expression Evaluation:** Below steps are required to evaluate prefix expression.

Step 1: Scan prefix expression from right to left.

Step 2: If reading symbol is operand then PUSH it on to stack.

Step 3: If reading symbol is operator then POP two values from stack and perform the operations between them and push result back onto stack.

Note: First pop value become first operand and second pop value become second operand in expression.

Step 4: Repeat above all steps until input expression End.

==

❖ **Prefix to Infix Conversion**: Refers examples given in your notebook.

❖ **Postfix to Infox Conversion:** Refers examples given in your notebook.

==

❖ Difference between Array and Linked List:

Array	Linked List
Array is a collection of similar type of elements.	Linked list is collection nodes which are connected.
Array size is fixed	Linked list is variable size
Array size cannot be dynamically grow and shrink	Linked list size dynamically grows and shrinks.
Array elements are stored in contiguous memory location.	Linked list stored in random memory location.
Difficult to perform insertion and deletion operation on array.	Easy to perform insertion and deletion operation on linked list.
Memory utilization is poor	Memory utilization is better than array.
Array elements are accessed sequentially or directly by using the array index or subscript.	Linked list elements are accessed sequentially.
Memory Should be allocated at Compile-Time.	In Linked list memory is allocated at Run-Time.
Example: draw array of 4 size	Example: draw linked list of 4 nodes.

❖ Linked List:

Definition:

- Linked list is used to represent linear data structure.
- Linked list is a collection of nodes, every node consists of two fields.
  1. Data field - It contains information of data element.
  2. Next field - It is a pointer variable which contains address of next node.

Data	Next

Fig. Single Node structure

- There are four types of linked list:
  1. Singly Linked List
  2. Circular Linked List
  3. Doubly Linked List
  4. Doubly Circular Linked List.

Advantages of Linked Lists:

1. Linked list allocates the memory dynamically when required.
2. Insertion and deletion operations can be easily implemented.
3. Stacks and queues can be easily executed.
4. Linked List reduces the access time.

Disadvantages of Linked Lists

1. The memory is wasted as pointers require extra memory for storage.
2. No element can be accessed randomly, it has to access each node sequentially.
3. Reverse Traversing is difficult in linked list.

Applications of Linked Lists:

1. Linked lists are used to implement stacks, queues, graphs, etc.
2. Linked lists is used to implement the hash tables.
3. Linked list is used to represent the sparse matrix.
4. For manipulation of polynomial, we use linked list.

❖ **Basic Terms of Linked List:**

1. **Node:** A linked list is a linear data structure where each element is a separate object. Each element is called as a node of a linked list. Every node consists of two fields, first is data and second is address of next node.

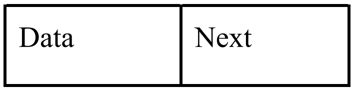


Fig. Single Node structure

2. **Information Field/ Data field:** It is used to store the data inside the node.
3. **Address/Next Pointer:** It is used to store the address of next node.
4. **Null Pointer: :** It is used to specify the end of the list. The last element of list contains NULL pointer to specify end of list.
5. **Empty List:** A linked list is said to be empty if head node contains NULL pointer.
   i.e. head=NULL.

❖ Basic Linked List Operations:

The basic operations that can be performed on a list are:

1. **Creation:** This operation is used to create a node in the linked list.
2. **Insertion:** This operation is used to insert a new node in the linked list.
   - Inserting node at the beginning of the list.
   - Inserting node at the middle of the list.
   - Inserting node at the end of the list.
3. **Deletion**: This operation is used to delete node from the linked list.
   - Deleting node from the beginning of the list.
   - Deleting node from the middle of the list.
   - Deleting node from the end of the list.
4. **Traversing:** It is a process of going through all the nodes of a linked list from one end to the other end.
5. **Display:** This operation is used to print all nodes information field.
6. **Searching:** To search a specific element in given linked list.
7. **Count**: To count number of nodes present in list.
8. **Concatenation:** Concatenating two linked lists.
9. **Sorting:** Sorting of an elements stored in a linked list.
10. **Separating** a linked list in two linked lists.

❖ Singly Linked List:

- Singly linked list is one of the type of linked list.
- Singly linked list is a collection of nodes and every node consists of two fields.

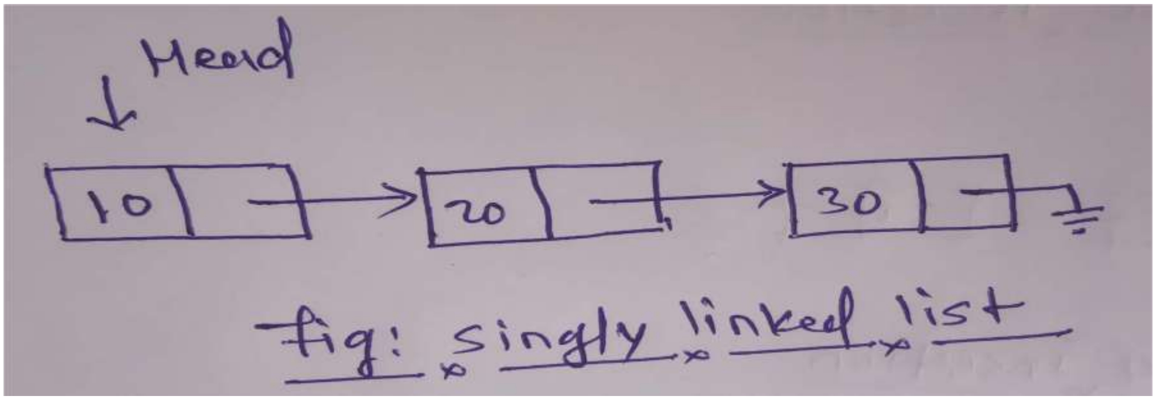
  Data field - It contains information of data element.

  Next field - It is a pointer variable which contains address of next node.

Data	Next

Fig. Node structure

- In singly linked list, we can move only in forward direction.
- We cannot move in backward direction.
- Example:



- **Advantages of Singly linked list:**

  1. Accessing the elements in forward direction is easier.

  2. Easy to perform insertion and deletion operation on SLL.

- **Disadvantages of Singly linked list:**

  1. We cannot access the node in backward direction.

  2. Accessing of node is a time consuming.

VJTECH Academy

- Below structure in ‘C’ can be used to define the node.

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

- Below steps are used to create a single node of SLL.

  1. node *p;
  2. p=(node*)malloc(sizeof(node));
  3. p->data=x;
  4. p->next=NULL;

VJTECH Academy

❖ Creating a Singly Linked List:

- To create a linked list, we will create a node one by one and add them to the end of the linked list.
- All these nodes are created by using the structure, pointer and dynamic memory allocation function malloc.
- Below structure in ‘C’ can be used to define the node.

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

- Below steps are used to create a single node of SLL.

  1. node *p;
  2. p=(node*)malloc(sizeof(node));
  3. p->data=x;
  4. p->next=NULL;

- **C function for creation of SLL.**

```
node* create(int n)
{
          node *head,*p;
           int x,i;
           printf(“\nEnter data:”);
           scanf(“%d”,&x);
           head=(node*)malloc(sizeof(node));
           head->data=x;
           head->next=NULL;
           p=head;
           for(i=2;i<=n;i++)
           {
                      printf(“\nEnter data:”);
                      scanf(“%d”,&x);
                      p->next=(node*)malloc(sizeof(node));
                      p=p->next;
                      p->data=x;
                      p->next=NULL;
           }
           return(head);
}
```

❖ Traversing a Singly Linked List:

- To visit each and every nodes of a linked list at least once is known as traversing operation.
- Traversing the linked list is always start from the first node and end with the last node.
- To display all the contents of the linked list, we need to traverse it from the first node to last node.
- **Algorithm:**

```
1.  Start
2.  Check if the linked list is empty or not.
    if head = = NULL then

         Display “Linked List is Empty” and goto step 5.

    else

         Goto step 3.

3. Set the head node to the temporary node p.

   p=head;

4. Traverse till the last node.

   while(p != NULL) then
        Display p->data;
        p=p->next;
5. Stop
```

- **C function for traversing/printing the SLL.**

```
void print(node *head)
{
        node *p;
        if(head==NULL)
        {
                printf(“\n Linked List is Empty!!!”);
         }
         else
          {
                  p=head;
                  printf(“\nSLL Nodes:”);
                  while(p!=NULL)
                   {
                           printf(“%d->”,p-data);
                           p=p->next;
                    }
            }
}
```

VJTECH Academy

❖ Counting no of nodes in a Singly Linked List:

- In this operation, we can count the no of nodes in the linked list.
- **Algorithm:**

```
1.  Start
2.  Set count variable to 0.
3.  Check if the linked list is empty or not.
        if head = = NULL then

            Display "Linked List is Empty" and goto step 7.

        else

            goto step 4.

 4. Set the head node to the temporary node p.

    p=head;

 5.  Traverse till the last node.

     while(p != NULL) then
         Increment the count variable by 1 i.e count++;
          p=p->next;
6. Display the total count of nodes.
7. Stop
```

- **C function for counting no of nodes in the SLL.**

```
void count(node *head)
{
        node *p;
        int count=0;
        if(head==NULL)
        {
                 printf("\n Linked List is Empty!!!");
        }
        else
         {
                 p=head;
                 printf("\nSLL Nodes:");
                 while(p!=NULL)
                  {
                          count++;
                          p=p->next;
                  }
                  printf("\n No of nodes in SLL=%d",count);
            }
}
```

❖ Searching a node in the linked list:

- Searching is the process of checking the given key element in the linked list.
- We will start the searching of a node from the first node and end up when element is found or end of the linked list.
- In this operation, we will match given key element with the every node of the linked list.
- **Algorithm:**

```
1.  Start
2.  Set flag variable to 0.
3.  Read key element from user.
4.  Check if the linked list is empty or not.
        if head = = NULL then

            Display "Linked List is Empty" and goto step 7.

        else

            goto step 4.

4. Set the head node to the temporary node p.

      p=head;

5.  while(p ! = NULL) then

        if(key = = p->data) then

            Set flag to 1 and goto step 6

        Otherwise

            p=->next.

  6. if flag = = 1 then
        Display "Element is found"
     else
        Display " Element is not found"
  7. Stop
```

- **C function for searching:**

```
void count(node *head)
{
        node *p;
        int key,flag=0;
        printf("\n Enter key element to be search:");
        scanf("%d"& key);
        if(head = = NULL)
        {
                 printf("\n Linked List is Empty!!!");
         }
         else
          {
                   p=head;
                   printf("\nSLL Nodes:");
                   while(p!=NULL)
                    {
                              if(key = = p->data)
                               {
                                        flag=1;
                                        break;
                                }
                              p=p->next;
                     }
                     if(flag = =1)
                          printf("\n Element is found!!!");
                     else
                          printf("\n Element is not found!!!");
            }
}
```

❖ Insertion:

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

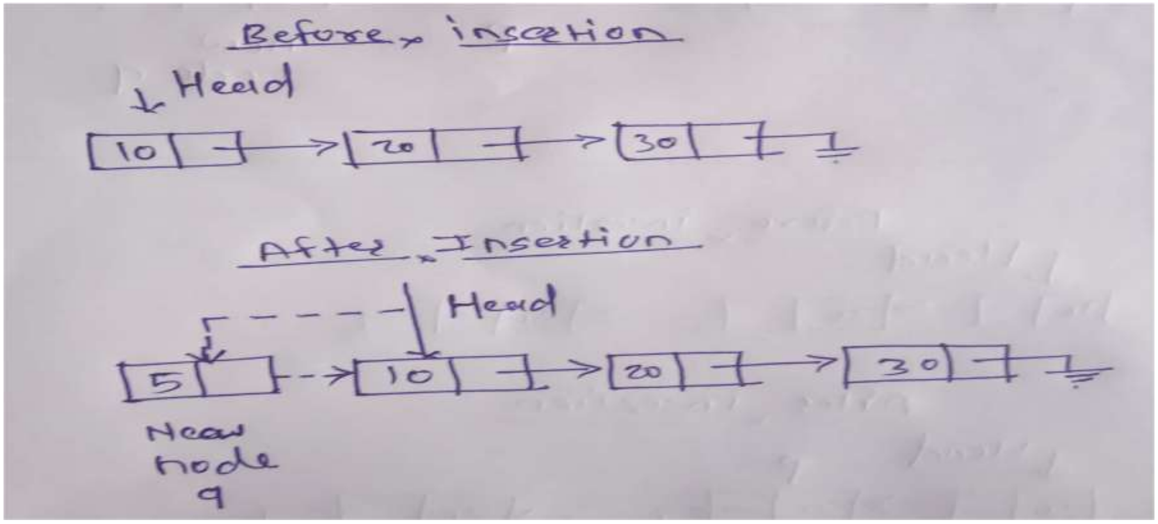
1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Middle of the list

Inserting At Beginning of the list:

- In this operation, new node can be created and added at the beginning of a list.
- New node points to existing first node and after that make new node as head node.
- Algorithm:

```
1.  Start
2.  Allocates memory for the new node q.
      q=(node*)malloc(sizeof(node));
3.  Assign data to the data field and NULL to the next field of the new node q.
       q->data=x;
       q->next=NULL;
4.  Check whether linked list is Empty.
      if (head = = NULL) then
                 Set head = q and goto step 6.
5.  Otherwise, set q→next = head and head = q.
6.  Stop
```

- Example:



VJTECH Academy

- C function for inserting a node at beginning of the list.

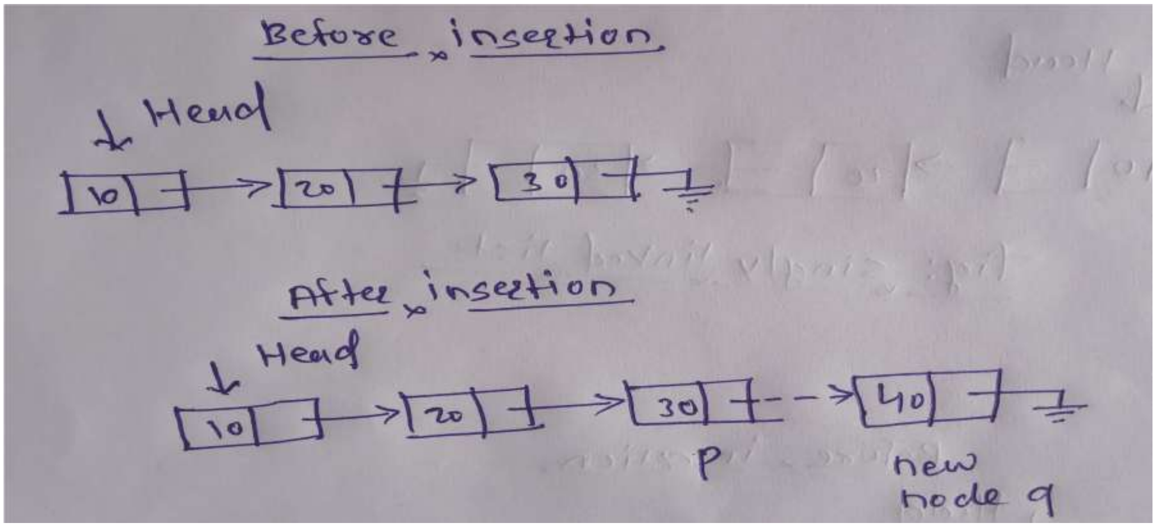
```
node* insert_at_begin(node *head)
{
        node *q;
        int x;
        printf("\n Enter data:");
        scanf("%d",&x);
        q=(node*)malloc(sizeof(node));
        q->data=x;
        q->next=NULL;
        if(head = = NULL)
        {
                head=q;
                return(head);
           }
          else
          {
                q->next=head;
                head=q;
                return(head);
           }
}
```

Inserting At End of the list:

- In this operation, new node can be created and added at the end of a list.
- Algorithm:

```
1.  Start
2.  Allocates memory for the new node q.
        q=(node*)malloc(sizeof(node));
3.  Assign data to the data field and NULL to the next field of the new node q.
        q->data=x;
        q->next=NULL;
4.  Check whether linked list is Empty
    if (head is 'NULL') then
            Set head=q & goto step 7
5.  Otherwise initialize p with head and Keep moving the p to its next node until it reaches to the
    last node in the list
        p=head
        while(p->next!=NULL)
            p=p->next;
6.  Make link between last node and new node.
        p->next=q;
7.  Stop
```

- Example:



- C function for inserting a node at end of the list.

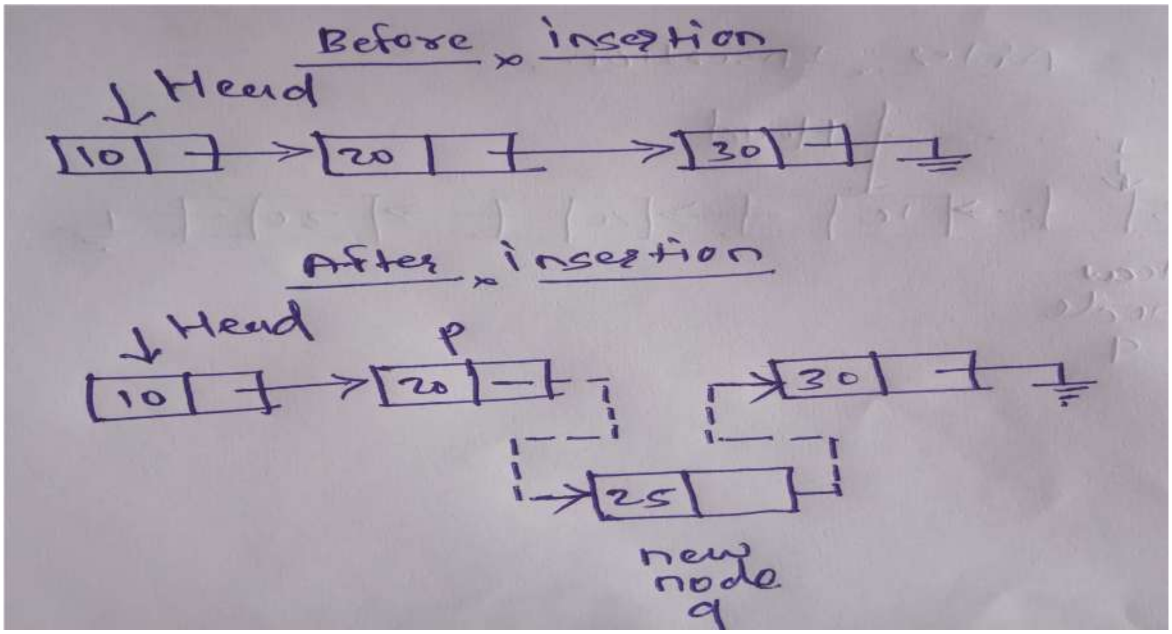
```
node* insert_at_end(node *head)
{
    node *q;
    int x;
    printf("\n Enter data:");
    scanf("%d",&x);
    q=(node*)malloc(sizeof(node));
    q->data=x;
    q->next=NULL;
    if(head = = NULL)
    {
        head=q;
        return(head);
     }
     else
     {
        p=head;
        while(p->next!=NULL)
        {
             p=->next;
        }
        p->next=q;
        return(head);
     }
}
```

Inserting at Middle of the linked list:

- In this operation, new node can be created and added in the middle of the linked list on the basis of the given location.
- Algorithm:

```
1.  Start
2.  Allocates memory for the new node q.
        q=(node*)malloc(sizeof(node));
3.  Assign data to the data field and NULL to the next field of the new node q.
        q->data=x;
        q->next=NULL;
4.  Check whether linked list is Empty
        if (head is 'NULL') then
            Set head=q & goto step 7
5.  Otherwise initialize p with head and keep moving the p to its next node until it reaches to the
    node after which we want to insert the new node q.
        p=head
        for(i=1;i<loc-1;i++)
            p=p->next;
6. Set q->next=p->next and p->next=q;
7. Stop
```

- Example:



- C function for inserting a node at middle of the list.

```
node* insert_at_middle(node *head)
{
        node *q;
        int x,i,loc;
        printf("\n Enter data:");
        scanf("%d",&x);
        printf("\n Enter location for insertion:");
        scanf("%d",&loc);
        q=(node*)malloc(sizeof(node));
        q->data=x;
        q->next=NULL;
        if(head = = NULL)
        {
                head=q;
                return(head);
          }
          else
          {
                p=head;
                for(i=1;i<loc-1;i++)
                {
                        p=->next;
                }
                q->next=p->next;
                p->next=q;
                return(head);
          }
}
```

❖ Deletion:

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

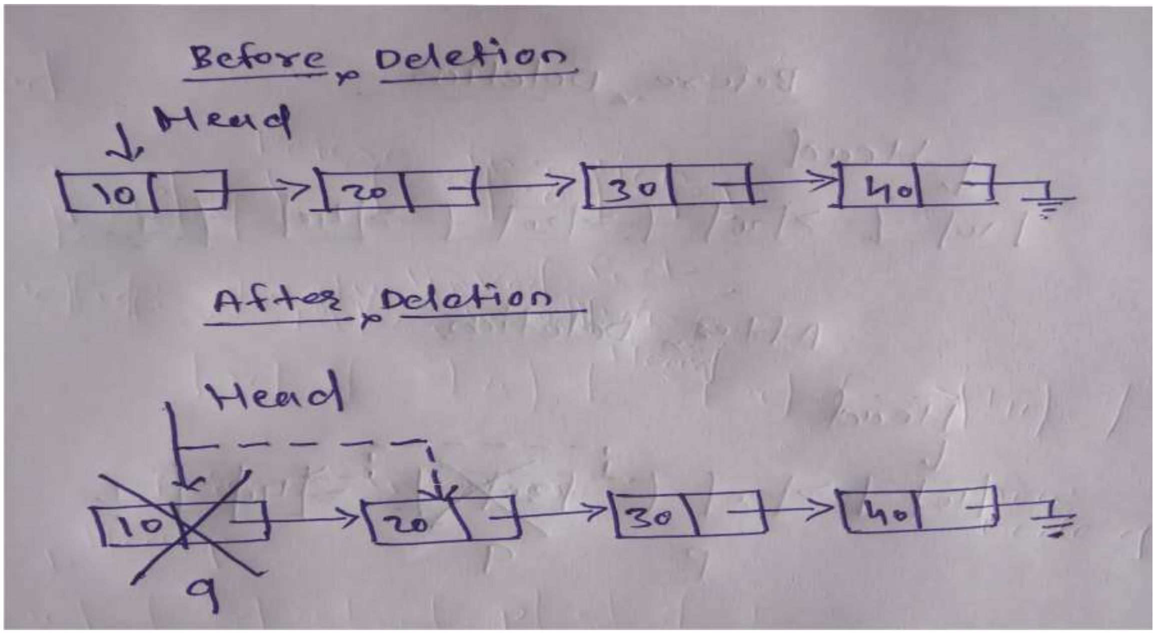
4. Deleting At Beginning of the list
5. Deleting At End of the list
6. Deleting At Middle of the list

Deleting at Beginning of the list:

- In this operation, first node is deleted from the linked list.
- Algorithm:

```
1.  Start
2.  Create temporary node pointer variable q.
3.  Check whether linked list is Empty.
      if (head = = NULL) then
            Display 'List is Empty!!! Deletion is not possible' and goto step 7.
4.  Assign address of first node to q i.e q=head.
5.  Move head node to next node i.e head=head->next.
6.  Delete q node i.e free(q).
7.  Stop
```

- Example:



- C function for deleting a node at beginning of the list.

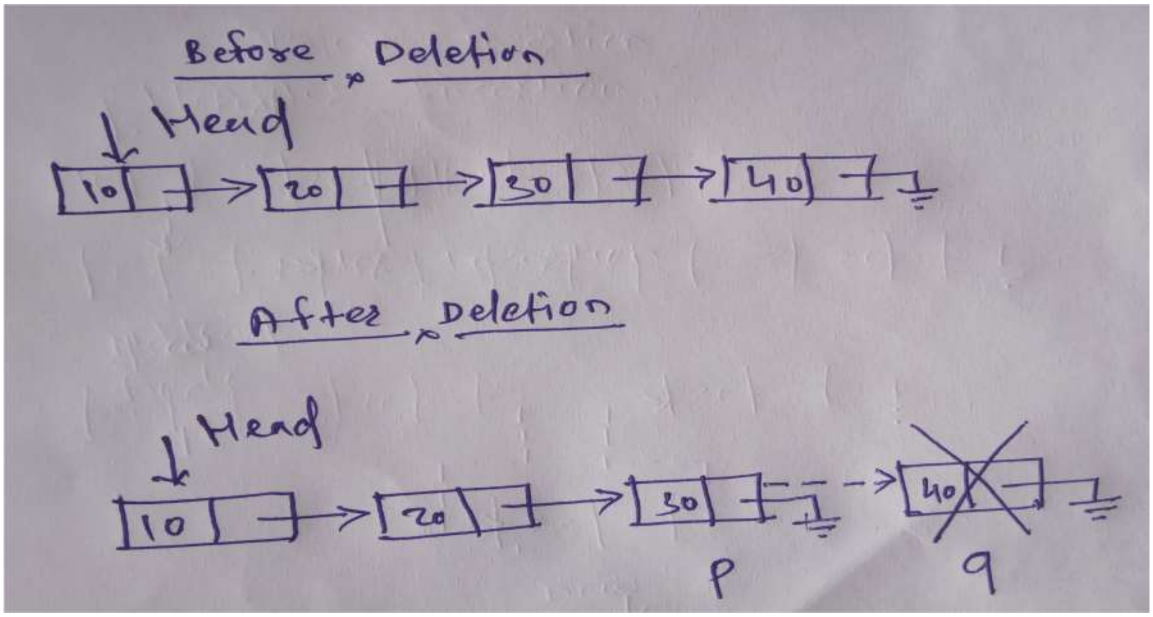
```
node* delete_at_begin(node *head)
{
        node *q;
        if(head = = NULL)
        {
                printf(“\n Linked List is Empty!!! Deletion is not possible”);
        }
        else
        {
                q=head;
                head=head->next;
                free(q);
                return(head);
        }
}
```

Deleting at End of the list:

- In this operation, last node is deleted from the linked list.
- Algorithm:

```
1.  Start
2.  Create temporary node pointer variable q & p.
3.  Check whether linked list is Empty.
    i.e if (head = = NULL) then
            Display 'List is Empty!!! Deletion is not possible' and goto step 8.
4.  Otherwise initialize p with head and keep moving the p to its next node until it reaches to the
second last node in the list
    i.e p=head
        while(p->next->next!=NULL)
            p=p->next;
5.  Assign last node address in q i.e q=p->next;
6.  Assign NULL value to next field of second last node p  i.e p->next=NULL.
7.  Delete q node i.e free(q).
8.  Stop
```

- Example:



- C function for deleting a node at end of the list.

```
node* delete_at_end(node *head)
{
        node *q,*p;
        if(head = = NULL)
        {
                printf(“\n Linked List is Empty!!! Deletion is not possible”);
        }
        else
        {
                p=head;
                while(p->next->next!=NULL)
                 {
                         p=p->next;
                }

                q=p->next;
                p->next=NULL;
                free(q);
                return(head);
        }
}
```

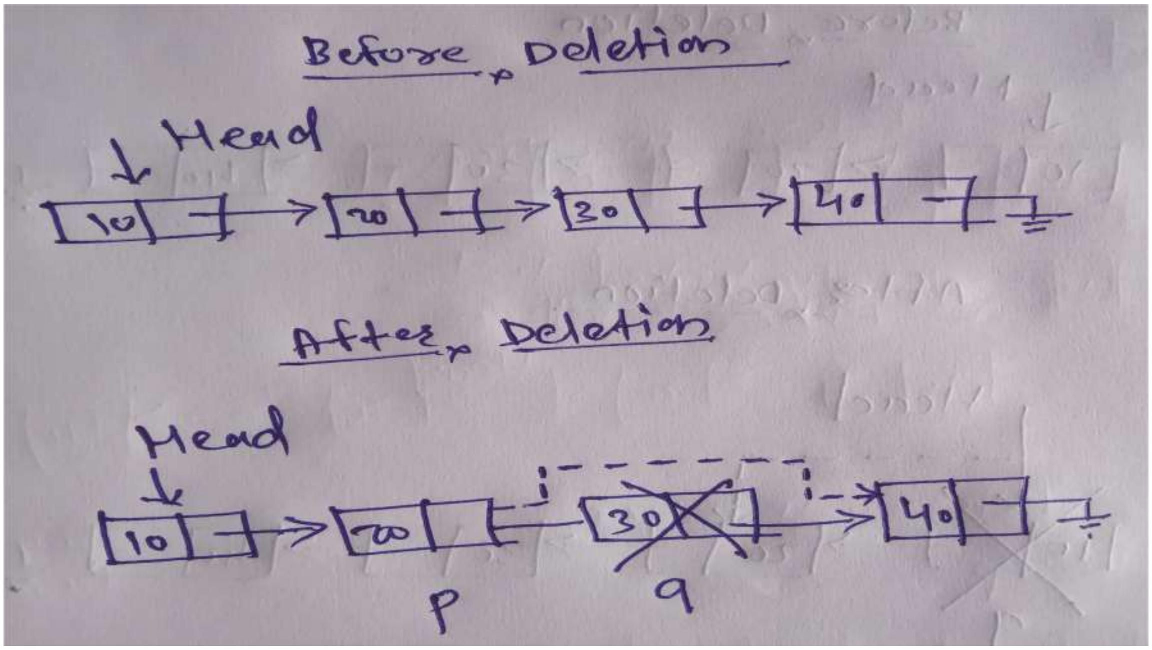
VJTECH Academy

Deleting at Middle of the linked list:

- In this operation, node is deleted from the middle of the linked list on the basis of the given location.
- Algorithm:

```
1.   Start
2.   Create temporary node pointer variable q & p.
3.   Check whether linked list is Empty.
     i.e if (head = = NULL) then
            Display 'List is Empty!!! Deletion is not possible' and goto step 8.
4. Assign address of first node to  p and traverse the list up to previous node of node to be deleted.
     i.e p=head
        for(i=1;i<loc-1;i++)
             p=p->next;
5. Mark the node to be deleted q i.e q=p->next.
6. Assign value of  next field of q node to next field of p node.
   i.e p->next=q->next;
7. Delete q node i.e free(q).
8. Stop
```

- Example:



- C function for deleting a node at middle of the list.

```
node* delete_at_middle(node *head)
{
        node *q,*p;
        int i,loc;
        if(head = = NULL)
        {
                printf("\n Linked List is Empty!!! Deletion is not possible");
         }
         else
         {
                p=head;
                for(i=1;i<loc-1;i++)
                {
                          p=->next;
                }
                q=p->next;
                p->next=q->next;
                free(q);
                return(head);
           }
}
```

❖ Singly Linked Program:

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

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

node* create(int n);
void print(node *p);
void count(node *p);
void search(node *p);

node* insert_at_Begin(node *head);
node* insert_at_Middle(node *head);
node* insert_at_End(node *head);

node* delete_at_Begin(node *head);
node* delete_at_Middle(node *head);
node* delete_at_End(node *head);

void main()
{
        node *Head;
        int no,op;
        clrscr();
        Head=NULL;
        do
        {
                printf("\n************SLL OPERATIONS*************");
                printf("\n1.Create\n2.Print\n3.Count\n4.Search");
                printf("\n5.Insert at Begin\n6.Insert at Middle\n7.Insert at End");
                printf("\n8.Delete from Begin\n9.Delete from Middle\n10.Delete from
End\n11.Exit");
                printf("\nEnter your choice:");
                scanf("%d",&op);
                switch(op)
                {
```

VJTECH Academy

```
case 1: printf("\nEnter size of linked list:");
          scanf("%d",&no);
          Head=create(no);
          getch();
          clrscr();
          break;
case 2: print(Head);
          getch();
          clrscr();
          break;
case 3: count(Head);
          getch();
          clrscr();
          break;
case 4: search(Head);
          getch();
          clrscr();
          break;
case 5: Head=insert_at_Begin(Head);
          getch();
          clrscr();
          break;
case 6: Head=insert_at_Middle(Head);
          getch();
          clrscr();
          break;
case 7: Head=insert_at_End(Head);
          getch();
          clrscr();
          break;
case 8: Head=delete_at_Begin(Head);
          getch();
          clrscr();
          break;
case 9: Head=delete_at_Middle(Head);
          getch();
          clrscr();
          break;
case 10: Head=delete_at_End(Head);
          getch();
          clrscr();
```

```
                        break;
                case 11: printf("\nGood Bye..Thanks for using our Application!!!");
                        break;
                default: printf("\nPlease select valid option!!!");
                         getch();
                         clrscr();
                }
        }while(op!=11);
        getch();
}
node* create(int n)
{
        node *head,*p;
        int i,x;
        printf("\nEnter data:");
        scanf("%d",&x);

        head=(node*)malloc(sizeof(node));
        head->data=x;
        head->next=NULL;

        p=head;

        for(i=2;i<=n;i++)
        {
                printf("\nEnter data:");
                scanf("%d",&x);
                p->next=(node*)malloc(sizeof(node));
                p=p->next;
                p->data=x;
                p->next=NULL;
        }
        printf("\nLinked list created successfully:");
        return(head);
}
```

```
void print(node *p)
{
        printf("\nSLL Nodes:");
        while(p!=NULL)
        {
                printf("%d->",p->data);
                p=p->next;
        }
}
void count(node *p)
{
        int count=0;
        while(p!=NULL)
        {
                count++;
                p=p->next;
        }
        printf("\nNo of Nodes:%d",count);
}
void search(node *p)
{
        int key,flag=0;
        printf("\nEnter key element to be search:");
        scanf("%d",&key);
        while(p!=NULL)
        {
                if(key==p->data)
                {
                flag=1;
                break;
                }
                p=p->next;
        }
        if(flag==1)
                printf("\nElement is found!!!");
        else
                printf("\nElement is not found!!!");
}
```

```
node* insert_at_Begin(node *head)
{
        node *q;
        int x;
        printf("\nEnter data:");
        scanf("%d",&x);
        q=(node*)malloc(sizeof(node));
        q->data=x;
        q->next=NULL;
        if(head==NULL)
        {
                head=q;
                return(head);
        }
        else
        {
                q->next=head;
                head=q;
                printf("\nNode inserted at Begin successfully!!!");
                return(head);
        }
}

node* insert_at_End(node *head)
{
        node *q,*p;
        int x;
        printf("\nEnter data:");
        scanf("%d",&x);
        q=(node*)malloc(sizeof(node));
        q->data=x;
        q->next=NULL;
        if(head==NULL)
        {
                head=q;
                return(head);
        }
        else
        {
                p=head;
                while(p->next!=NULL)
```

```
                    {
                    p=p->next;
                    }
                    p->next=q;
                    printf("\nNode inserted at end successfully!!!");
                    return(head);
          }
}
node* insert_at_Middle(node *head)
{
          node *q,*p;
          int x,loc,i;
          if(head==NULL)
          {
                    printf("\nLinked list is already Empty!!! you can not insert node at middle!!!");
                    return(NULL);
          }
          else
          {
                    printf("\nEnter data:");
                    scanf("%d",&x);
                    q=(node*)malloc(sizeof(node));
                    q->data=x;
                    q->next=NULL;
                    printf("\nEnter location for insertion:");
                    scanf("%d",&loc);
                    p=head;
                    for(i=1;i<loc-1;i++)
                    {
                    p=p->next;
                    }
                    q->next=p->next;
                    p->next=q;
                    printf("\nNode inserted at middle successfully!!!");
                    return(head);
          }
}
```

```
node* delete_at_Begin(node *head)
{
        node *q;
        if(head==NULL)
        {
                printf("\nLinked list is already Empty!!! you can not delete node!!!");
                return(NULL);
        }
        else
        {
                q=head;
                head=head->next;
                free(q);
                printf("\nNode deleted from Begin successfully!!!");
                return(head);
        }
}

node* delete_at_End(node *head)
{
        node *q,*p;
        if(head==NULL)
        {
                printf("\nLinked list is already Empty!!! you can not delete node!!!");
                return(NULL);
        }
        else
        {
                p=head;
                while(p->next->next!=NULL)
                {
                p=p->next;
                }
                q=p->next;
                p->next=NULL;
                free(q);
                printf("\nNode deleted from end successfully!!!");
                return(head);
        }
}
```

```
node* delete_at_Middle(node *head)
{
        node *q,*p;
        int i,loc;
        if(head==NULL)
        {
                printf("\nLinked list is already Empty!!! you can not delete node!!!");
                return(NULL);
        }
        else
        {
                printf("\nEnter location of node for deletion:");
                scanf("%d",&loc);
                p=head;
                for(i=1;i<loc-1;i++)
                {
                p=p->next;
                }
                q=p->next;
                p->next=q->next;
                free(q);
                printf("\nNode deleted from middle successfully!!!");
                return(head);
        }
}
```

❖ Circular Linked List:

- Circular linked list is one of the type of linked list.
- Circular linked list is a collection of nodes where last node is connected to first node.
- In circular linked list, next field of last node store the address of first node.
- Circular linked list is a collection of nodes and every node consists of two fields.

  Data field - It contains information of data element.

  Next - It is a pointer variable which contains address of next node.

Data	Next

Fig. Node structure

- In circular linked list, we can move in forward with circular direction.
- Example:

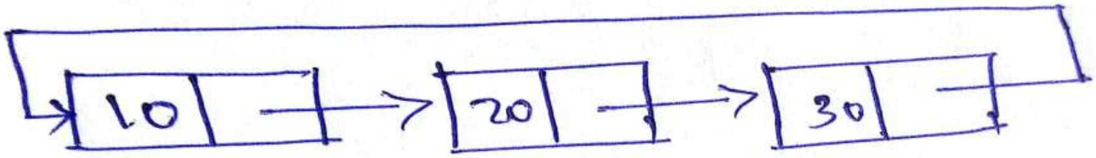


fig: circular Linked List

- Below structure in 'C' can be used to define the node.

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

- Below steps are used to create a single node of CLL.

  1. node *p;

  2. p=(node*)malloc(sizeof(node));

  3. p->data=x;

  4. p->next=NULL;

VJTECH Academy

❖ Creating a Circular Linked List:

- To create a linked list, we will create a node one by one and add them to the end of the linked list.
- In CLL, next field of last node store the address of first node.
- All these nodes are created by using the structure, pointer and dynamic memory allocation function malloc.
- Below structure in 'C' can be used to define the node.

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

- Below steps are used to create a single node of CLL.

```
1. node *p;
2. p=(node*)malloc(sizeof(node));
3. p->data=x;
4. p->next=NULL;
```

- **C function for creation of CLL.**

```
node* create(int n)
{
        node *head,*p;
         int x,i;
         printf("\nEnter data:");
         scanf("%d",&x);
         head=(node*)malloc(sizeof(node));
         head->data=x;
         head->next=head;
         p=head;
         for(i=2;i<=n;i++)
         {
                  printf("\nEnter data:");
                  scanf("%d",&x);
                  p->next=(node*)malloc(sizeof(node));
                  p=p->next;
                  p->data=x;
                  p->next=head;
         }
         return(head);
}
```

❖ Traversing a Circular Linked List:

- To visit each and every nodes of a linked list at least once is known as traversing operation.
- Traversing the linked list is always start from the first node and end with the last node.
- To display all the contents of the linked list, we need to traverse it from the first node to last node.
- **Algorithm:**

```
1.   Start
2.   Check if the linked list is empty or not.
       if head = = NULL then

            Display “Linked List is Empty” and goto step 5.

     else

           Goto step 3.

 3.  Set the head node to the temporary node.

         p=head;

4.  Traverse till the last node.

     do
      {
            Display p->data;
            p=p->next;
      }while(p!=head);
5. Stop
```

- **C function for traversing/printing the SLL.**

```
void print(node *head)
{
        node *p;
        if(head==NULL)
        {
                printf(“\n Linked List is Empty!!!”);
         }
         else
          {
                  p=head;
                  printf(“\nCLL Nodes:”);
                  do
                   {
                              printf(“%d->”,p-data);
                              p=p->next;
                    }while(p!=head);
           }
}
```

❖ Counting no of nodes in a Circular Linked List:

- In this operation, we can count the no of nodes in the linked list.
- **Algorithm:**

```
1.  Start
2.  Set count variable to 0.
3.  Check if the linked list is empty or not.
        if head = = NULL then

            Display "Linked List is Empty" and goto step 7.

        else

            goto step 4.

4. Set the head node to the temporary node.

    p=head;

5.  Traverse till the last node and increment count variable by one.

    do
    {
        Increment the count variable by 1 i.e count++;
        p=p->next;
    }while(p!=head);
6. Display the total count of nodes.
7. Stop
```

- **C function for counting no of nodes in the SLL.**

```
void count(node *head)
{
        node *p;
        int count=0;
        if(head==NULL)
        {
                printf("\n Linked List is Empty!!!");
        }
        else
         {
                p=head;
                printf("\nSLL Nodes:");
                do
                 {
                        count++;
                        p=p->next;
                 }while(p!=head);
                 printf("\n No of nodes in CLL=%d",count);
          }
}
```

VJTECH Academy

❖ Searching a node in the Circular linked list:

- Searching is the process of checking the given key element in the linked list.
- We will start the searching of a node from the first node and end up when element is found or end of the linked list.
- In this operation, we will match given key element with the every node of the linked list.
- **Algorithm:**

```
1.  Start
2.  Set flag variable to 0.
3.  Read key element from user.
4.  Check if the linked list is empty or not.
       if head = = NULL then
             Display "Linked List is Empty" and goto step 9.
       else
             goto step 5.
5. Set the head node to the temporary node.

        p=head;

6.  Check key value with the node data.

       i.e   if (key = = p->data) then
                     set flag to 1 and go to step 8.
                else
                      p=p->next.
7. Repeat step 6 until p!=head otherwise goto step 8

8. if flag = = 1 then

          Display "Element is found"
     else
          Display " Element is not found"
 9. Stop
```

VJTECH Academy

- **C function for searching:**

```
void count(node *head)
{
        node *p;
        int key,flag=0;
        printf("\n Enter key element to be search:");
        scanf("%d"& key);
        if(head = = NULL)
        {
                printf("\n Linked List is Empty!!!");
         }
         else
          {
                  p=head;
                  printf("\nCLL Nodes:");
                  do
                   {
                             if(key = = p->data)
                              {
                                      flag=1;
                                      break;
                               }
                             p=p->next;
                    }while(p!=head);
                    if(flag = =1)
                         printf("\n Element is found!!!");
                    else
                         printf("\n Element is not found!!!");
            }
}
```

❖ **Insertion:**

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

7. Inserting At Beginning of the list
8. Inserting At End of the list
9. Inserting At Middle of the list

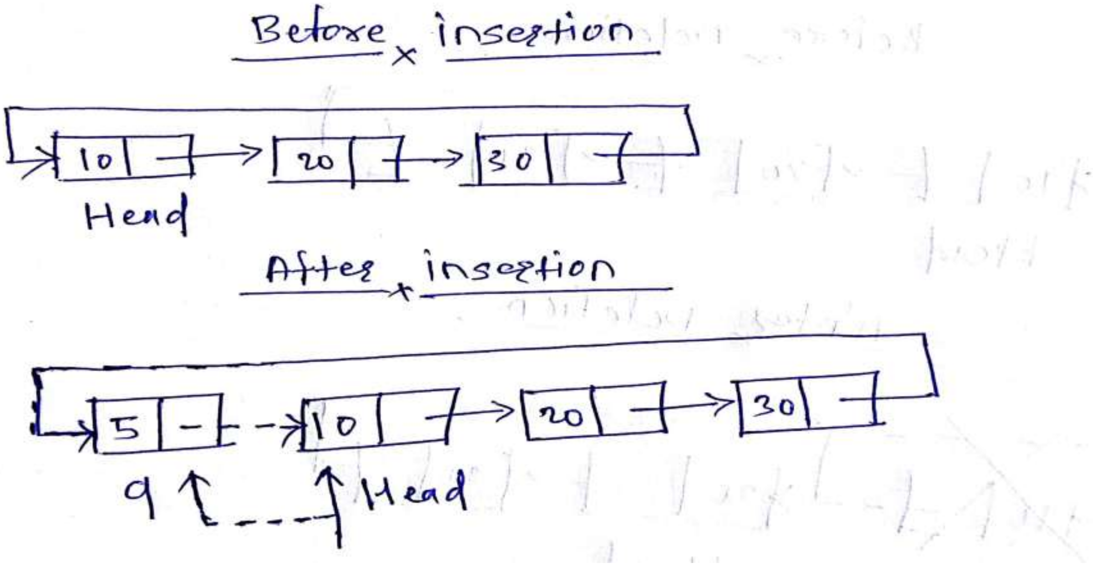
VJTECH Academy

Inserting at Beginning of the list:

- In this operation, new node can be created and added at the beginning of a list.
- New node points to existing first node and after that make new node as head node.
- Algorithm:

```
1.  Start
2.  Allocates memory for the new node q.
    i.e q=(node*)malloc(sizeof(node));
3. Assign data to the data field and address of itself  to the next field of the new node q.
    i.e q->data=x;
        q->next=q;
4.  Check whether linked list is Empty.
    i.e if (head = = NULL) then
              Set head = q and goto step 8.
5.  Set the head node to the temporary node p.

    i.e p=head

6. Keep moving the 'p' to its next node until it reaches to the last node
   i.e  while(p->next!=NULL)
        {
              p=p->next;
        }
7. Set q->next=head, p->next=q and head=q.
8. Stop
```

- Example:



- C function for inserting a node at beginning of the list.

```
node* insert_at_begin(node *head)
{
        node *q,*p;
        int x;
        printf(“\n Enter data:”);
        scanf(“%d”,&x);
        q=(node*)malloc(sizeof(node));
        q->data=x;
        q->next=q;
        if(head = = NULL)
        {
                head=q;
                return(head);
           }
          else
          {
                p=head;
                while(p->next!=NULL)
                 {
                         p=p->next;
                  }
                q->next=head;
                 p->next=q;
                head=q;
                return(head);
           }
}
```

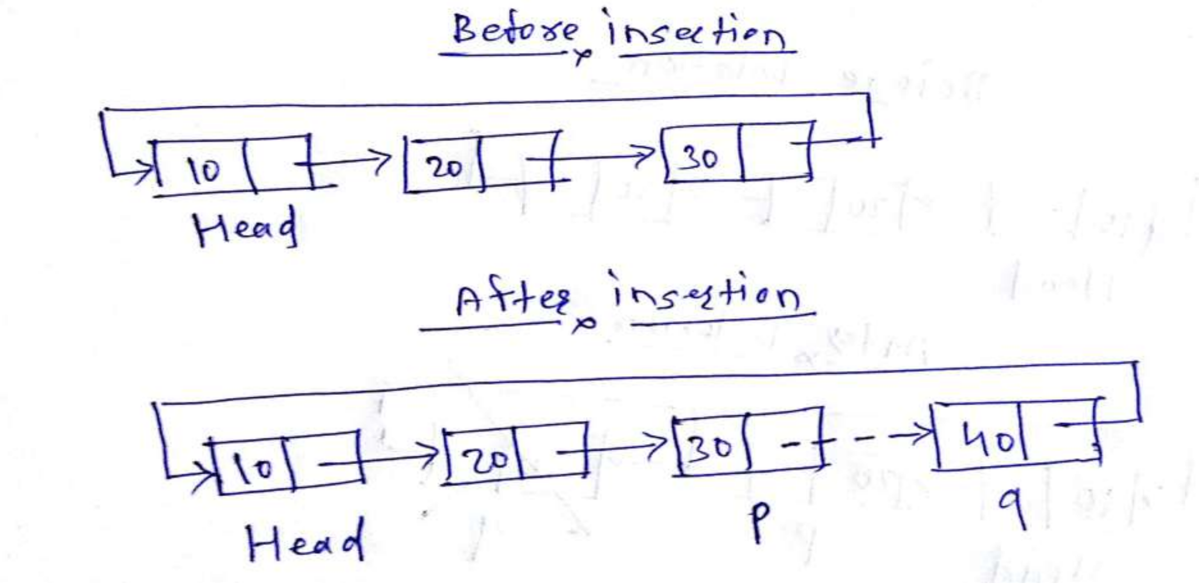
VJTECH Academy

Inserting at End of the list:

- In this operation, new node can be created and added at the end of a list.
- Algorithm:
-

```
1.   Start
2.   Allocates memory for the new node q.
     i.e q=(node*)malloc(sizeof(node));
3.  Assign data to the data field and address of itself  to the next field of the new node q.
     i.e q->data=x;
         q->next=q;
4.    Check whether linked list is Empty.
     i.e if (head = = NULL) then
                 Set head = q and goto step 8.
5.   Set the head node to the temporary node p.

     i.e p=head

6.  Keep moving the 'p' to its next node until it reaches to the last node
     i.e  while(p->next!=NULL)
         {
                 p=p->next;
          }
7. Set p->next=q, q->next=head.
8. Stop
```

- Example:



- C function for inserting a node at end of the list.

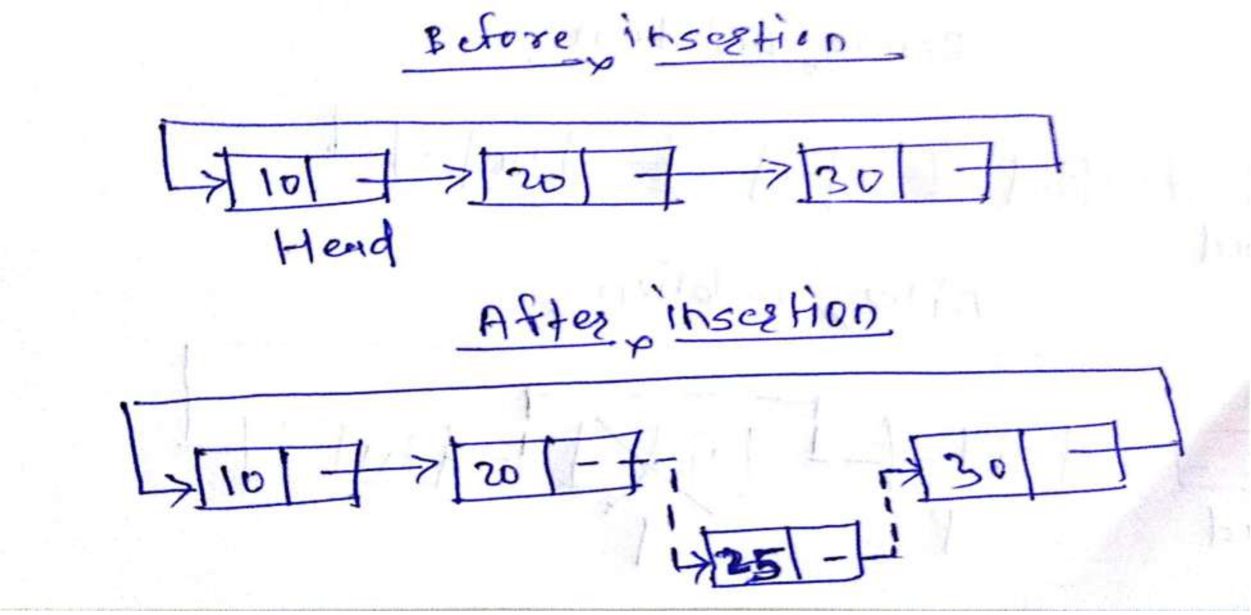
```
node* insert_at_end(node *head)
{
        node *q,*p;
        int x;
        printf("\n Enter data:");
        scanf("%d",&x);
        q=(node*)malloc(sizeof(node));
        q->data=x;
        q->next=qL;
        if(head = = NULL)
        {
                head=q;
                return(head);
           }
          else
          {
                p=head;
                while(p->next!=NULL)
                {
                          p=->next;
                }
                p->next=q;
                q->next=head;
                return(head);
           }
}
```

Inserting at Middle of the linked list:

- In this operation, new node can be created and added in the middle of the linked list on the basis of the given location.
- Algorithm:
-

```
1.   Start
2.   Allocates memory for the new node q.
     i.e q=(node*)malloc(sizeof(node));
3.  Assign data to the data field and address of itself  to the next field of the new node q.
     i.e q->data=x;
         q->next=q;
4.   Check whether linked list is Empty.
     i.e if (head = = NULL) then
                 Set head = q and goto step 8.
5.   Set the head node to the temporary node p.

     i.e p=head

6.  Keep moving the p node  to its next node until it reaches to the node after which we
want to insert the new Node q
         for(i=1;i<loc-1;i++)
                p=p->next;
7. Set q->next=p->next and p->next=q;
8. Stop
```

- Example:



- C function for inserting a node at middle of the list.

```
node* insert_at_middle(node *head)
{
        node *q,*p;
        int x,i,loc;
        printf("\n Enter data:");
        scanf("%d",&x);
        printf("\n Enter location for insertion:");
        scanf("%d",&loc);
        q=(node*)malloc(sizeof(node));
        q->data=x;
        q->next=q;
        if(head = = NULL)
        {
                head=q;
                return(head);
          }
          else
          {
                p=head;
                for(i=1;i<loc-1;i++)
                {
                          p=->next;
                }
                q->next=p->next;
                p->next=q;
                return(head);
          }
}
```

❖ Deletion:

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

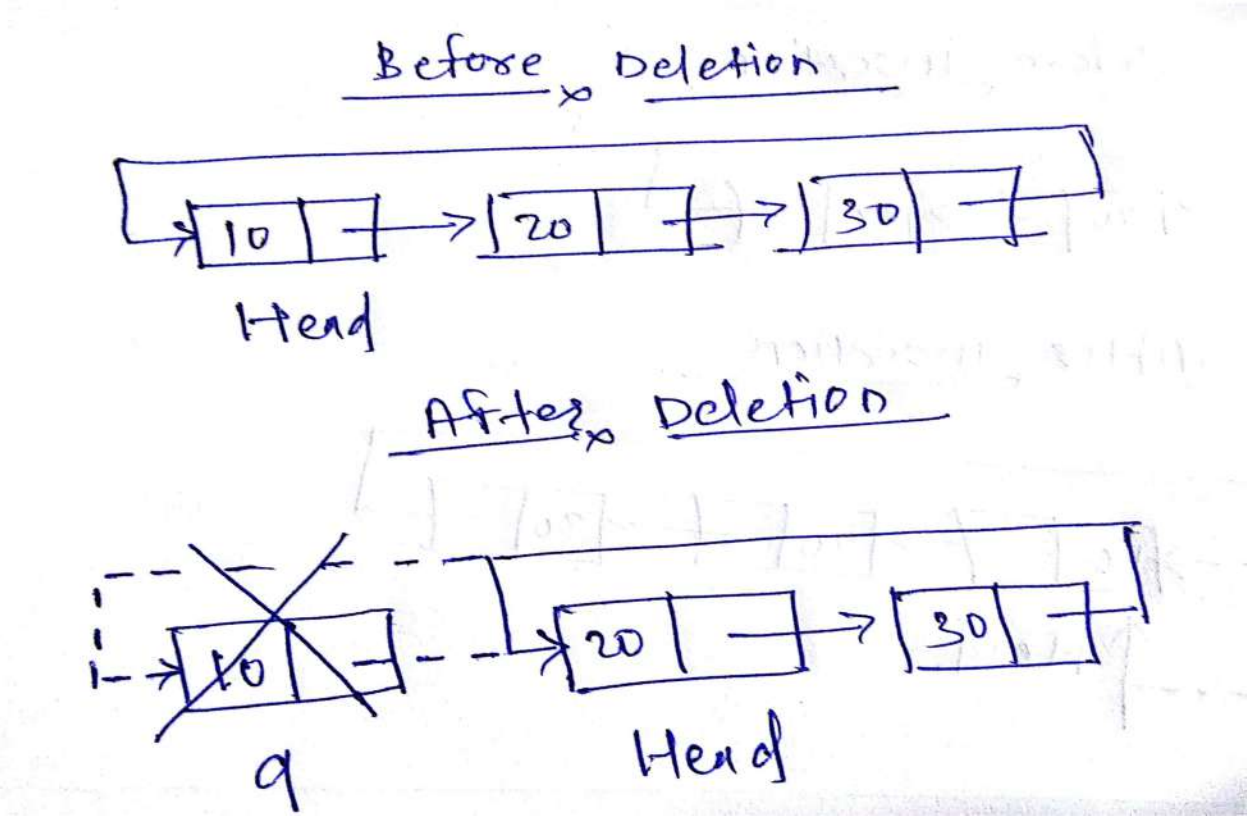
1. Deleting At Beginning of the list
2. Deleting At End of the list
3. Deleting At Middle of the list

Deleting at Beginning of the list:

- In this operation, first node is deleted from the linked list.
- Algorithm:

```
1.  Start
2.  Create temporary node pointer variable q & p.
4.  Check whether linked list is Empty.
    i.e if (head = = NULL) then
          Display 'List is Empty!!! Deletion is not possible' and goto step 7.
4.  Assign address of first node head to p i.e  p=head.
5.  Move the p node to next node until it reaches to the last node.
    i.e while(p->next!=NULL)
          {
                p=p->next;
          }
6.  Set q=head, head=head->next,  p->next=head and delete q node.
7.  Stop
```

- Example:



- C function for deleting a node at beginning of the list.

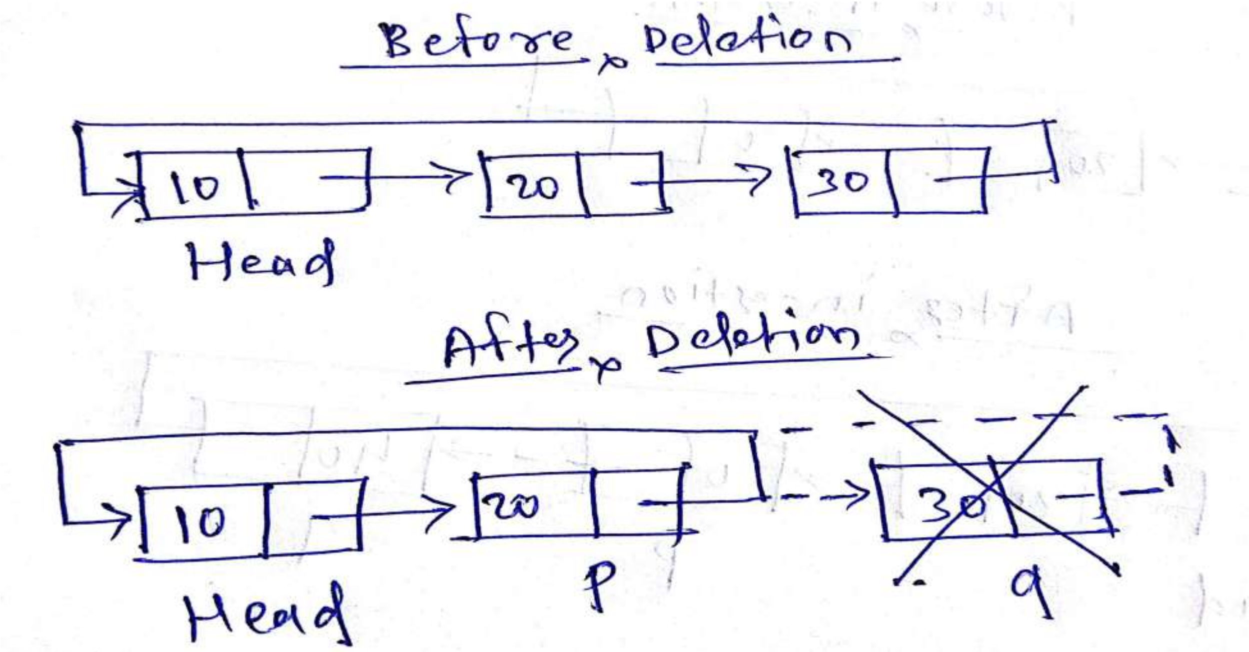
```
node* delete_at_begin(node *head)
{
        node *q,*p;
        if(head = = NULL)
        {
                printf("\n Linked List is Empty!!! Deletion is not possible");
        }
        else
        {
                p=head;
                while(p->next!=NULL)
                 {
                          p=p->next;
                 }
                q=head;
                head=head->next;
                 p->next=head;
                free(q);
                return(head);
        }
}
```

Deleting at End of the list:

- In this operation, last node is deleted from the linked list.
- Algorithm:

```
        1.   Start
        2.   Create temporary node pointer variable q & p.
        3.   Check whether linked list is Empty.
             i.e if (head = = NULL) then
                         Display 'List is Empty!!! Deletion is not possible' and goto step 8.
        4.   Otherwise initialize p with head and keep moving the p to its next node until it reaches to the
second last node in the list
             i.e p=head
                 while(p->next->next!=head)
                        p=p->next;
        5.   Assign last node address in q i.e q=p->next;
        6.   Assign head to next field of second last node p  i.e p->next=head.
        7.   Delete q node i.e free(q).
        8.   Stop
```

- Example:



- C function for deleting a node at end of the list.

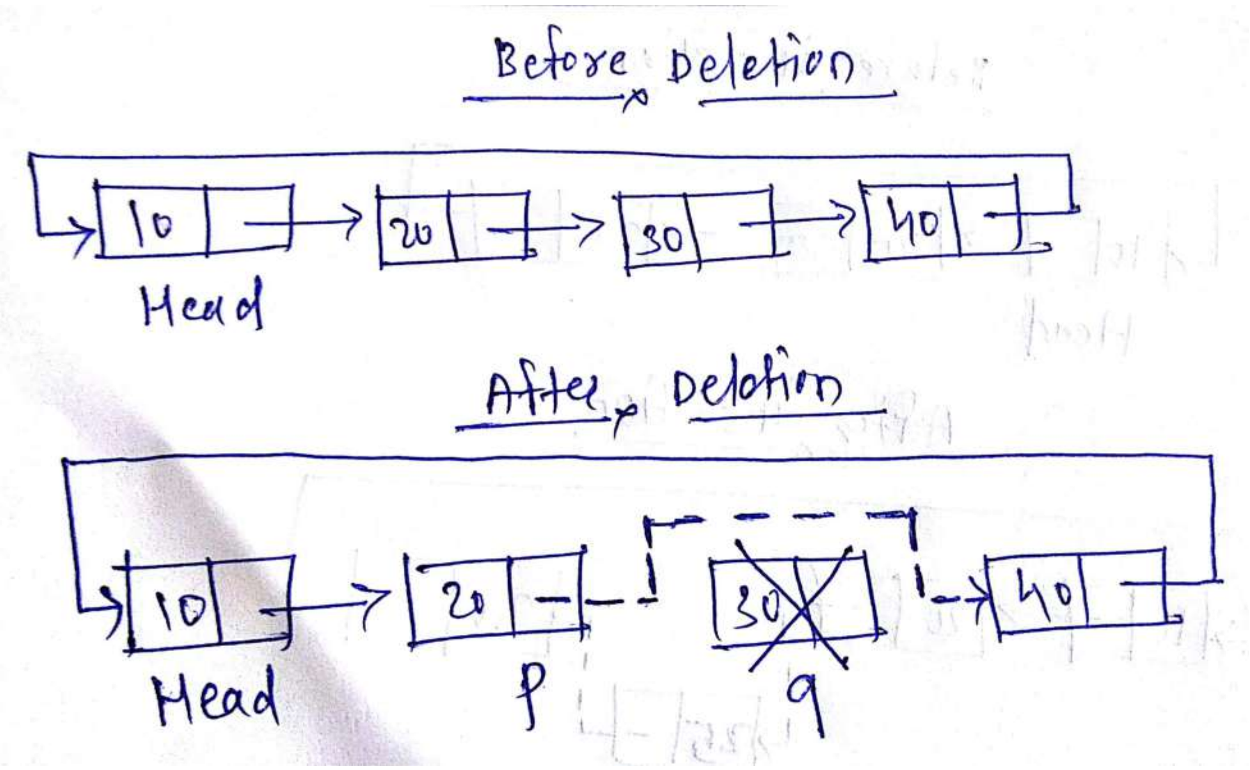
```
node* delete_at_end(node *head)
{
        node *q,*p;
        if(head = = NULL)
        {
                printf(“\n Linked List is Empty!!! Deletion is not possible”);
        }
        else
        {
                p=head;
                while(p->next->next!=head)
                 {
                         p=p->next;
                }

                q=p->next;
                p->next=head;
                free(q);
                return(head);
        }
}
```

Deleting at Middle of the linked list:

- In this operation, node is deleted from the middle of the linked list on the basis of the given location.
- Algorithm:

```
1. Start
2. Create temporary node pointer variable q & p.
3. Check whether linked list is Empty.
   i.e if (head = = NULL) then
       a) Display 'List is Empty!!! Deletion is not possible' and goto step 8.
4. Assign address of first node to  p and traverse the list up to previous node of node to be deleted.
   i.e p=head
       for(i=1;i<loc-1;i++)
           p=p->next;
5. Mark the node to be deleted q i.e q=p->next.
6. Assign value of  next field of q node to next field of p node.
   i.e p->next=q->next;
7. Delete q node i.e free(q).
8. Stop
```

- Example:



- C function for deleting a node at middle of the list.

```
node* delete_at_middle(node *head)
{
        node *q,*p;
        int i,loc;
        if(head = = NULL)
        {
                printf("\n Linked List is Empty!!! Deletion is not possible");
         }
         else
         {
                p=head;
                for(i=1;i<loc-1;i++)
                {
                          p=->next;
                }
                q=p->next;
                p->next=q->next;
                free(q);
                return(head);
          }
}
```

❖ Circular Linked Program:

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

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

node* create(int n);
void print(node *head);
void count(node *head);
void search(node *head);

node* insert_at_Begin(node *head);
node* insert_at_Middle(node *head);
node* insert_at_End(node *head);

node* delete_at_Begin(node *head);
node* delete_at_Middle(node *head);
node* delete_at_End(node *head);
```

```
void main()
{
        node *Head;
        int no,op;
        clrscr();
        Head=NULL;
        do
        {
                printf("\n************CLL OPERATIONS*************");
                printf("\n1.Create\n2.Print\n3.Count\n4.Search");
                printf("\n5.Insert at Begin\n6.Insert at Middle\n7.Insert at End");
                printf("\n8.Delete from Begin\n9.Delete from Middle\n10.Delete from End\n11.Exit");
                printf("\nEnter your choice:");
                scanf("%d",&op);
                switch(op)
                {
                        case 1: printf("\nEnter size of linked list:");
                                scanf("%d",&no);
                                Head=create(no);
                                getch();
                                clrscr();
                                break;
                        case 2: print(Head);
                                getch();
                                clrscr();
                                break;
                        case 3: count(Head);
                                getch();
                                clrscr();
                                break;
                        case 4: search(Head);
                                getch();
                                clrscr();
                                break;
                        case 5: Head=insert_at_Begin(Head);
                                getch();
                                clrscr();
                                break;
                        case 6: Head=insert_at_Middle(Head);
                                getch();
                                clrscr();
                                break;
                        case 7: Head=insert_at_End(Head);
                                getch();
                                clrscr();
                                break;
                        case 8: Head=delete_at_Begin(Head);
                                getch();
                                clrscr();
                                break;
                        case 9: Head=delete_at_Middle(Head);
                                getch();
                                clrscr();
                                break;
                        case 10: Head=delete_at_End(Head);
```

```
                                getch();
                                clrscr();
                                break;
                        case 11: printf("\nGood Bye..Thanks for using our Application!!!");
                                break;
                        default: printf("\nPlease select valid option!!!");
                                 getch();
                                 clrscr();
                }
        }while(op!=11);
        getch();
}
node* create(int n)
{
        node *head,*p;
        int i,x;
        printf("\nEnter data:");
        scanf("%d",&x);

        head=(node*)malloc(sizeof(node));
        head->data=x;
        head->next=head;

        p=head;

        for(i=2;i<=n;i++)
        {
                printf("\nEnter data:");
                scanf("%d",&x);
                p->next=(node*)malloc(sizeof(node));
                p=p->next;
                p->data=x;
                p->next=head;
        }
        printf("\nLinked list created successfully:");
        return(head);
}
void print(node *head)
{
        node *p;
        p=head;
        printf("\nCLL Nodes:");
        do
        {
                printf("%d->",p->data);
                p=p->next;
        }while(p!=head);
}
```

```
void count(node *head)
{
        int count=0;
        node *p;
        p=head;
        do
        {
                count++;
                p=p->next;
        }while(p!=head);
        printf("\nNo of Nodes:%d",count);
}

void search(node *head)
{
        int key,flag=0;
        node *p;
        printf("\nEnter key element to be search:");
        scanf("%d",&key);
        p=head;
        do
        {
                if(key==p->data)
                {
                        flag=1;
                        break;
                }
                p=p->next;
        }while(p!=head);
        if(flag==1)
        {
                printf("\nElement is found!!!");
        }
        else
        {
                printf("\nElement is not found!!!");
        }
}
```

```
node* insert_at_Begin(node *head)
{
        node *q,*p;
        int x;
        printf("\nEnter data:");
        scanf("%d",&x);
        q=(node*)malloc(sizeof(node));
        q->data=x;
        q->next=q;
        if(head==NULL)
        {
                head=q;
                return(head);
        }
        else
        {
                p=head;
                while(p->next!=head)
                {
                        p=p->next;
                }
                p->next=q;
                q->next=head;
                head=q;
                printf("\nNode inserted at Begin successfully!!!");
                return(head);
        }
}

node* insert_at_End(node *head)
{
        node *q,*p;
        int x;
        printf("\nEnter data:");
        scanf("%d",&x);
        q=(node*)malloc(sizeof(node));
        q->data=x;
        q->next=q;
        if(head==NULL)
        {
                head=q;
                return(head);
        }
        else
        {
                p=head;
                while(p->next!=head)
                {
                        p=p->next;
                }
                p->next=q;
                q->next=head;
                printf("\nNode inserted at end successfully!!!");
                return(head);
```

```
    }
}
node* insert_at_Middle(node *head)
{
    node *q,*p;
    int x,loc,i;
    printf("\nEnter data:");
    scanf("%d",&x);
    q=(node*)malloc(sizeof(node));
    q->data=x;
    q->next=q;
    if(head==NULL)
    {
        head=q;
        printf("\nNode inserted in CLL as first node because list was empty");
        return(head);
    }
    else
    {

        printf("\nEnter location for insertion:");
        scanf("%d",&loc);
        p=head;
        for(i=1;i<loc-1;i++)
        {
            p=p->next;
        }
        q->next=p->next;
        p->next=q;
        printf("\nNode inserted at middle successfully!!!");
        return(head);
    }
}
node* delete_at_Begin(node *head)
{
    node *q,*p;
    if(head==NULL)
    {
        printf("\nLinked list is already Empty!!! you can not delete node!!!");
        return(NULL);
    }
    else
    {
        p=head;
        while(p->next!=head)
        {
            p=p->next;
        }
        q=head;
        head=head->next;
        p->next=head;
        free(q);
        printf("\nNode deleted from Begin successfully!!!");
        return(head);
    }
```

```
}

node* delete_at_End(node *head)
{
        node *q,*p;
        if(head==NULL)
        {
                printf("\nLinked list is already Empty!!! you can not delete node!!!");
                return(NULL);
        }
        else
        {
                p=head;
                while(p->next->next!=head)
                {
                        p=p->next;
                }
                q=p->next;
                p->next=head;
                free(q);
                printf("\nNode deleted from end successfully!!!");
                return(head);
        }
}
node* delete_at_Middle(node *head)
{
        node *q,*p;
        int i,loc;
        if(head==NULL)
        {
                printf("\nLinked list is already Empty!!! you can not delete node!!!");
                return(NULL);
        }
        else
        {
                printf("\nEnter location of node for deletion:");
                scanf("%d",&loc);
                p=head;
                for(i=1;i<loc-1;i++)
                {
                        p=p->next;
                }
                q=p->next;
                p->next=q->next;
                free(q);
                printf("\nNode deleted from middle successfully!!!");
                return(head);
        }
}
```

❖ Trees:

Basic Terminology:

1. **Tree:**
   - Tree is a non-linear data structure.
   - Tree is a collection of nodes and edges but it does not contain cycle.
   - Example:

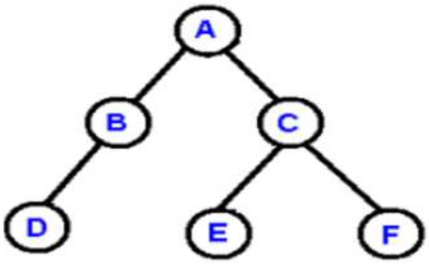


Figure: Tree (a)

   - **Types of tree:**
     I. General tree
     II. Binary tree
     III. Binary Search tree
     IV. Expression tree

2. **Degree of Node:**
   - The maximum number of child nodes of any node is known as degree of node.
   - Degree of node is the number of nodes connected to a particular node.
   - Example: Degree of each node of above given tree (a) is given below.

Nodes	Degree
A, C	2
B	1
D, E, F	0

3. **Degree of Tree:**
   - The maximum number of child nodes of root node is known as degree of tree.
   - Degree of tree is the number of nodes connected to a root node.
   - Example: For above given tree (a), degree of tree is 2 as no of child nodes of root node A is 2.
   - **In-degree of Node:** No of incoming edges to that particular node is known as In-degree of Node.
   - **Out-degree of Node:** No of outgoing edges from that particular node is known as Out-degree of Node.

4. Root Node:

- It is a special node of a tree and all tree referenced through it.
- The root node which don't have parent node.
- In above figure tree (a), node A is root node.

5. Parent Node:

- The immediate predecessor of a node is known as parent node.
- In above given tree (a), node C is the parent node of E and F.

6. Child Node:

- All immediate successors of nodes are its children node.
- In above given tree (a), node E and F are the child nodes of C.

7. Leaf Node:

- The node which has no children's is known as Leaf node.
- The node which has 0 degree is known as Leaf node.
- Leaf node is known as terminal node.
- In above given tree (a) node D, E and F is Leaf node.

8. Levels:

- Level of a node represents the generation of a node.
- If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- Example:

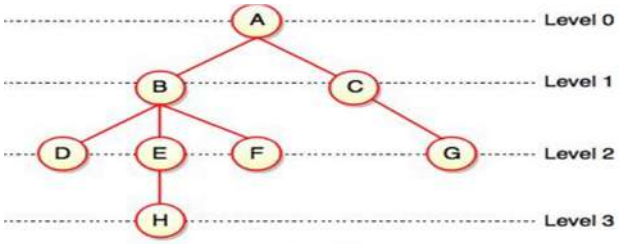


Fig. Levels of Tree

9. Sibling:

- The nodes which have the same parent node are known as Sibling node.
- In above given tree (a), node B and C both are sibling node.

10. Path:

- The sequence of consecutive edges from source node to destination node is called a path.
- The length of path means no of edges present on that path.
- In above given tree (a) path A-B-D , A-C-E and A-C-F
- The length of A-B-D path is 2 because 2 edges are present on that path.

11. Ancestor Nodes:

- Ancestors of a node are all nodes along path from root to that a particular node.
- In above given tree (a), node B & A are ancestor of D

12. Descendant Nodes:

- All the nodes that are reachable from the root node or parent node are the descendant nodes of that parent node or root node.

13. Depth/Height of tree:

- Maximum number of levels in a tree is known as depth/Height of a tree.
- Example:

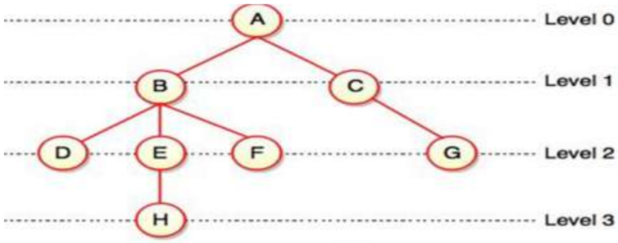


Fig. Levels of Tree

In above given tree, depth/height of tree is 3

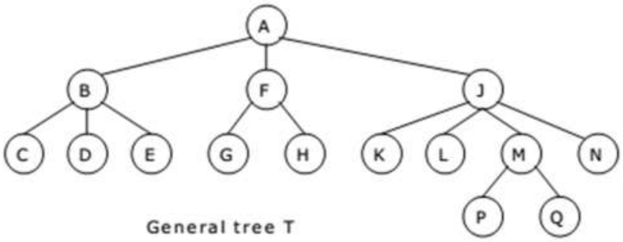
❖ Types of Trees:

There are three types of tree which are listed below:

1. General Tree
2. Binary Tree
3. Binary Search Tree
4. Expression Tree

1. General Tree:

- General tree is one of the types of tree.
- Every nodes of the general tree which has infinite number of children.
- In general tree, root has in-degree 0 and maximum out-degree N.
- In general tree, each node have in-degree one and maximum out-degree N.
- Sub-trees of general tree are not ordered.
- Example:



2. Binary Tree:

- Binary tree is one of the types of tree.
- Binary tree is a tree in which every nodes of tree have maximum two children.
- In binary tree, root has in-degree 0 and maximum out-degree 2.
- In binary tree, each node have in-degree one and maximum out-degree 2.
- Sub-trees of binary tree are ordered.
- **There are five types of Binary tree:**
  1. Full Binary Tree
  2. Complete Binary Tree
  3. Skewed Binary Tree
  4. Extended Binary Tree

VJTECH Academy

- Example of Binary tree:

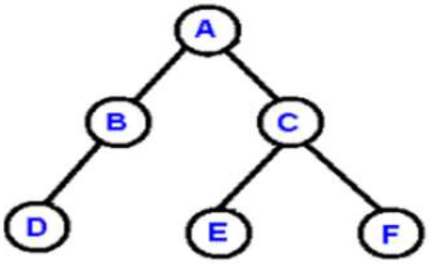


Fig. Binary Tree

- **Applications of Binary Tree:**

  I) To create expression tree.

  II) To create Binary search tree.

  III) To represent hierarchical data into the memory

3. Binary Search Tree:

- Binary tree is said to be binary search tree when all nodes satisfy below rules:
  1. All key value should be distinct / unique
  2. The value of left child should be less than its parent node value.
  3. The value of right child should be greater than its parent node value.
- Example of binary search tree:

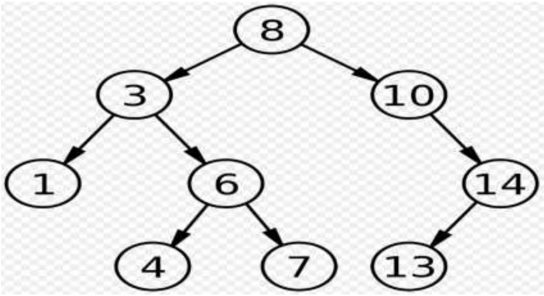


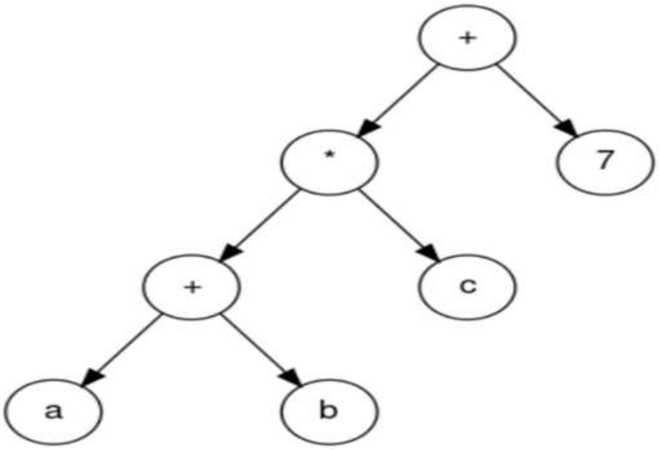
Fig. Binary Search Tree

4. Expression Tree:

- Expression trees are a special kind of binary tree used to evaluate certain expressions.
- When expression is represented by using binary tree then that tree is known as Expression tree.
- In Expression tree, leaf nodes are used to represent operand and intermediate nodes are used to represent operators.

VJTECH Academy

- Example of Expression Tree:



Above tree represent Expression ((a + b) * c + 7)

❖ Tree Traversals Technique:

- To visit each and every nodes of a tree at least once is known as Tree Traversing.
- There are three different Tree traversals technique given below:

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

1. Preorder traversal:-

- In this traversal method first visit root element, then left sub tree and then right sub tree.
- **Procedure:-**

  Step 1: Visit root node

  Step 2: Visit left sub tree in preorder

  Step 3: Visit right sub tree in preorder

2. Inorder traversal:-

- In this traversal method first visit left element, then root element and then the right element.
- **Procedure:-**

  Step 1: Visit left sub tree in inorder

  Step 2: Visit root node

  Step 3: Visit right sub tree in inorder

3. Postorder traversal:-

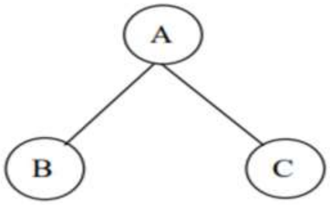
- In this traversal first visit left sub tree, then right sub tree and then the root element.
- **Procedure:-**

  Step 1: Visit left sub tree in postorder

  Step 2: Visit right sub tree in postorder

  Step 3: Visit root node

Example:



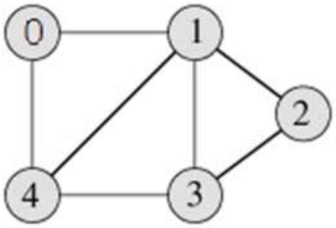
Preorder: A, B, C
Inorder: B, A, C
Postorder: B, C, A

VJTECH Academy

❖ Graphs:

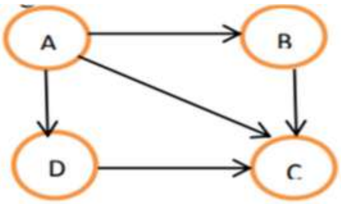
Basic Terminology:

1. **Graph:**
   - Graph is a non-linear data structure.
   - Graph is a collection of vertices and edges but it contains cycle.
   - Example:

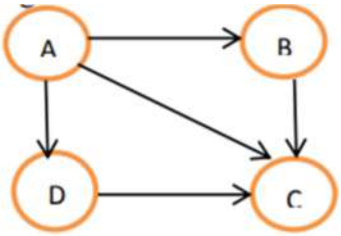


2. **Degree:**
   - The total number of edges linked/connected to the vertex is called as **degree**
   - **In-degree: -** No of total incoming edges to that particular vertex is known as In-degree.
   - **Out-degree: -** No of total outgoing edges from that particular vertex is known as Out-degree.
   - **Source Vertex: -** The vertex which has only outgoing edges and no incoming edges is known as Source Vertex.
   - **Sink Vertex: -** The vertex which has only incoming edges and no outgoing edges is known as Sink vertex.
   - **Pendant Vertex: -** The vertex which has only one incoming edge and no outgoing edges is known as Pendant vertex.
   - **Isolated Vertex: -** The vertex which has no incoming and no outgoing edges is known as Isolated Vertex.

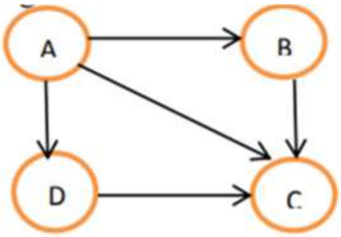
3. **Successor:** If there is a directed edge from vertex A to vertex B then vertex B is said to a successor of vertex A.



4. **Predecessor:** If there is a directed edge from vertex A to vertex B then vertex A is said to a Predecessor of vertex B.



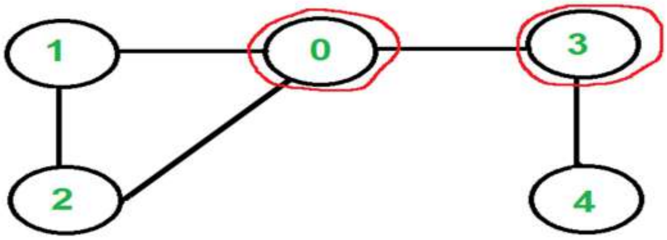
5. Path: The sequence of successive edges from source vertex to destination vertex is known as Path.



For the above graph paths can be A-B-C, A-B, and A-C, etc.

6. Articulation point:

- A vertex in an undirected connected graph is an articulation point if removing it disconnects the graph.
- On removing the vertex the graph gets disconnected, then that vertex is called the articulation point.
- Example:



Articulation points are 0 and 3

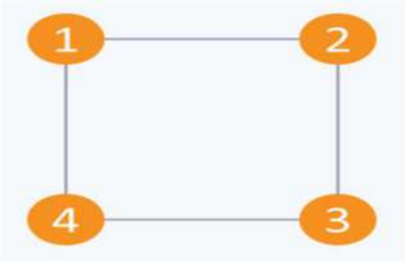
7. **Adjacent Vertex:** Two vertices are called adjacent if there is an edge between them.

❖ Types of Graph:

1. Undirected Graph
2. Directed Graph
3. Complete Graph
4. Weighted Graph

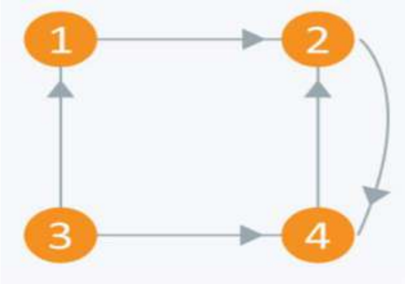
1. Undirected Graph:

- A graph with only undirected edges is said to be undirected graph.
- When edges of graph do not represent any direction then that graph is known as undirected graph.
- For this graph, all the edges are bi-directional.
- Example:



2. Directed Graph:

- A graph with only directed edges is said to be directed graph.
- When edges of graph represent direction then that graph is known as directed graph.
- For this graph, all the edges are uni-directional.
- Example:



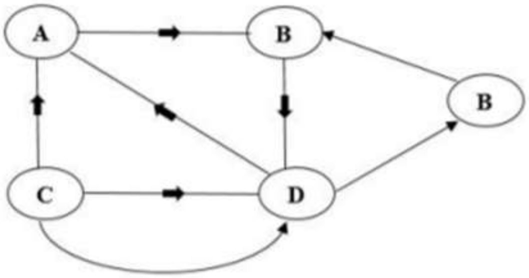
❖ Representation of Graph:

- Graph is a collection of vertices and edges but it contains cycle. It is a non-linear data structure.
- Graph data structure is represented using following representations:

  1. Adjacency matrix
  2. Adjacency list

1. Adjacency matrix:

- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.
- It is also called as bit matrix as it contains only two values i.e. 1 and 0.
- Value 1 indicates that there is an edge from vertex i to vertex j.
- Value 0 indicates that there is no edge from vertex i to vertex j.
- The adjacency matrix for the below example graph is:



Adjacency matrix for the above graph.

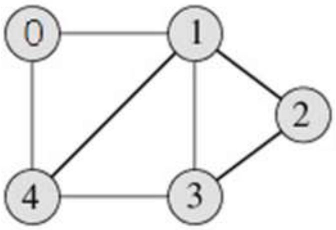
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	0	1	0
C	1	0	0	1	0
D	1	0	0	0	1
E	0	1	0	0	0

2. Adjacency List:

- In this representation, every vertex of graph contains list of its adjacent vertices.
- An array of linked lists is used.
- Size of the array is equal to number of vertices.

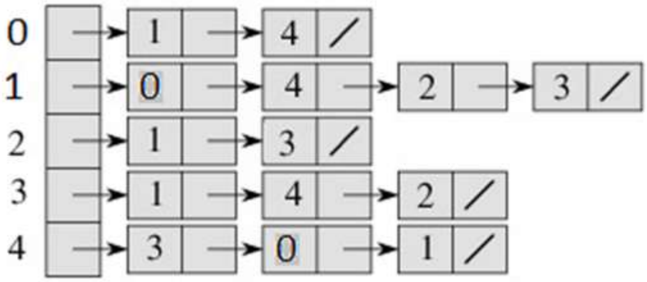
VJTECH Academy

- Suppose we have array[]. An entry array[i] represents the linked list of vertices adjacent to the ith vertex.
- This representation can also be used to represent a weighted graph.
- The weights of edges can be stored in nodes of linked lists.
- Adjacency list contains two columns as vertex name and adjacent vertices. It show who all are adjacent vertices of each vertex in a graph.
- The adjacency list for the below example graph is:



Vertex	Adjacent vertices
0	1,4
1	0,2,3,4
2	1,3
3	1,2,4
4	0,1,3

Following is adjacency list representation of the above graph.



VJTECH Academy

❖ Applications of Graph:

1. To represent road map
2. To represent circuit or networks
3. To represent program flow analysis
4. To represent transport network
5. To represent social network
6. Neural networks

1. Social Network Graphs: to tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.

2. Transportation networks: In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

3. Neural networks: Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 1011 neurons and close to 1015 synapses.

4. Utility graphs: The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.

5. Network packet traffic graphs: Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.

6. Graphs in compilers: Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes.

VJTech Academy
VJ
Inspiring Your Success...