
# DATA STRUCTURE IMPORTANT QUESTIONS LIST WITH ANSWERS

**Author:**

***"Prof. Vishal Jadhav"***

***[BE in Computer Engineering and Having 8.5 years of IT industry experience.***
***VJTech Academy, Maharashtra Contact No: +91-9730087674 / 7743909870,***
***Email id: vjtechacademy@gmail.com)]***

❖ **UNIT-I : Introduction on to Data Structures:** **Total Marks: 06**

1. **Write any four applications of data structure.**

**Ans:**

- Data structure can be linear or non-linear.
- Linear data structures are those in which data is stored sequential manner. Examples – Stack, Queue.
- Non-Linear data structures are those in which data is stored randomly. Examples – tree, graph, table.
- **Stack:** Evaluation of expressions, Backtracking, Runtime memory management, Arrangement of books in a library.
- **Queue:** Disk scheduling, CPU scheduling, File IO, Data transmission.
- **Linked List:** Representation of sparse matrices, Non-contiguous data storage, Implementation of non-binary tree or other data structures, Dynamic memory management, Equalizing parenthesis, Symbol tables
- **Graph:** Computer networking, Problem solutions involving 'Depth-First' search or 'Breadth-First' search algorithms, Representation of matrices, Study of molecular interactions in Chemistry
- **Tree:** Representation of data lists, quickly accessible data storage, Representation of hierarchal data, Routing of algorithms.

2. **State the need of data structure. Write any four operations perform on data structure.**

**Ans: Need of data structure:**

- Data structures are an important way of organizing information or data in a computer.
- It has a different way of storing & organizing data in a computer.
- It helps to store data in logical manner.
- It allows collection of data to grow & shrink dynamically over time & to organize the information so that one can access it using efficient algorithms.
- Specific data structures are essential ingredients of many efficient algorithms, & they make possible management of huge amount of data, such as large collections of databases.

**Operations perform on data structure:**

- Insertion
- Deletion
- Searching
- Sorting
- Traversing
- Merging

3. **Define Abstract Data Type**

**Ans:**

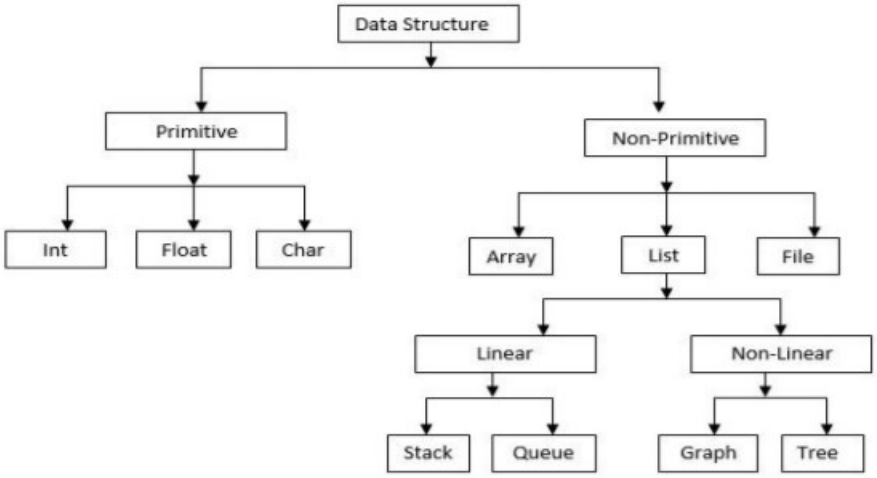
- Abstract Data type (ADT) is a type for objects whose behavior is defined by a set of value and a set of operations.
- The definition of ADT only mentions that what operations are to be performed but don't know how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called "abstract" because it gives an implementation independent view.
- The process of providing only the essentials details and hiding the unnecessary details is known as abstraction.
- The user of data type need not know that how data type is implemented, for example, we have been using int, float, char data types only with the knowledge with values that

can take and operations that can be performed on them without any idea of how these types are implemented.

- So, a user only needs to know what a data type can do but don't know how it will do it.
- We can think of ADT as a black box which hides the inner structure and design of the data type. Now we'll define three ADTs namely List ADT, Stack ADT, Queue ADT.

4. **Describe with example classification of data structure.**

**Ans: Classification of 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

**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

5. **Differentiate between Linear and Non-linear data structures w.r.t. any 2 parameters.**

| Sr. No. | Linear Data Structure | Non Linear Data Structure |
|---|---|---|
| 1. | Data Elements are organized sequentially | Data elements are linked to other data elements through pointers |
| 2. | Easy to implement | Complicated to implement. |
| 3. | Data elements in a linear data structure are traversed one after the other | Data elements cannot be traversed at one run |
| 4. | Some commonly used linear data structures are arrays, linked lists, stacks and queues. | Data structures like multidimensional arrays, trees and graphs are some examples of widely used nonlinear data structures |

6. **Define primitive and non-primitive data structure.**

**Ans:**

**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

**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

7. **List any four operations on data structure.**

**Ans: Operations perform on data structure:**

- Insertion
- Deletion
- Searching
- Sorting
- Traversing
- Merging

8. **Explain the term: Space and time 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 known 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:**

```
1. sum=0;
2. for(i=0;i<n;i++);
3. {
4.       sum=sum+i;
5. }
6. 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**

9. **Define recursion. Write any two advantages of recursion. Write 'C' program to calculate the factorial of number using recursion.**

**Ans:**

**Definition:** Recursion is the process of calling function by itself. A recursive function body contains function call statement that calls itself repeatedly.

**Advantages:**

- The main benefit of a recursive approach to algorithm design is that it allows programmers to take advantage of the repetitive structure present in many problems.
- Complex case analysis and nested loops can be avoided.
- Recursion can lead to more readable and efficient algorithm descriptions.
- Recursion is also a useful way for defining objects that have a repeated similar structural form.
- Using recursion, the length of the program can be reduced.

**Program:**

```
#include <stdio.h>
int fact(int n);
int main()
{
int N,result;
printf("\nEnter Value of N:");
scanf("%d",&N);
result=fact(N);
printf("\nFactorial=%d",result);
return 0;
}
int fact(int n)
{
        if(n==0)
                return(1);
        else
                return(n*fact(n-1));
}
```

10. **Define algorithm. How it is analyzed?**

**Ans:**

**Algorithm:**

An algorithm is a step by step set of instructions designed to perform a specific task.

**There are different types of time complexities which can be analyzed for an algorithm:**

**Best Case Time Complexity:**
It is measure of minimum time that algorithm will require for input of size "n".
Running time of many algorithms varies not only for inputs of different sizes but also input of same size. For example in running time of some sorting algorithms, sorting will depend on ordering of input data. Therefore if input data of "n" items is presented in sorted order, operations performed by algorithm will take least time.

**Worst Case Time Complexity:**
It is measure of maximum time that algorithm will require for input of size "n" Therefore if various algorithms for sorting are taken into account & say "n" input data items are supplied in reverse order for any sorting algorithm, then algorithm will require $n^2$ operations to perform sort which will correspond to worst case time complexity of algorithm.

**Average Case Time Complexity:**
The time that an algorithm will require to execute typical input data of size "n" is known as average case time complexity. We can say that value that is obtained by averaging running time of an algorithm for all possible inputs of size "n" can determine average case time complexity. Computation of exact time taken by algorithm for its execution is very difficult. Thus work done by algorithm for execution of input of size "n" defines time analysis as function f(n) of input data items.

11. **Define Big 'O' Notation and State limitations of the Big 'O' notation.**

**Definition:**

- 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 calculates 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

**Limitations:**

1. Many algorithms are too hard to analyze mathematically.
2. There may not be sufficient information to calculate the behavior of the algorithm in the average case.
3. Big-Oh analysis only tells us how the algorithm grows with the size of the problem, not how efficient it is, as it does not consider programming effort.

12. **Implement C Program for performing following operations on Array: Insertion, Display.**

```
#include <stdio.h>
int main()
{
   int a[10],N,x,i,Loc;
   printf("\nEnter Size of Array:");
   scanf("%d",&N);
   printf("\nEnter Array Elements:");
   for(i=0;i<N;i++)
   {
      scanf("%d",&a[i]);
   }
   printf("\nEnter element for insertion:");
   scanf("%d",&x);
   printf("\nEnter location for insertion:");
   scanf("%d",&Loc);
   for(i=N-1;i>=Loc-1;i--)
   {
      a[i+1]=a[i];
   }
   a[Loc-1]=x;
   N++;
   printf("\nYour Array Elements after insertion:");
   for(i=0;i<N;i++)
   {
      printf("%d\t",a[i]);
   }
   return 0;
}
```

13. **Implement C Program for performing following operations on Array: deletion, Display.**

```
#include <stdio.h>
int main()
{
   int a[10],N,i,Loc;
   printf("\nEnter Size of Array:");
   scanf("%d",&N);
   printf("\nEnter Array Elements:");
   for(i=0;i<N;i++)
   {
      scanf("%d",&a[i]);
   }
   printf("\nEnter location for deletion:");
   scanf("%d",&Loc);
   for(i=Loc-1;i<N-1;i++)
   {
      a[i]=a[i+1];
   }
   N--;
   printf("\nYour Array Elements after deletion:");
   for(i=0;i<N;i++)
   {
      printf("%d\t",a[i]);
   }
   return 0;
}
```


# DATA STRUCTURE IMPORTANT QUESTIONS LIST WITH ANSWERS

**Author:**

***"Prof. Vishal Jadhav"***

***[BE in Computer Engineering and Having 8.5 years of IT industry experience.***
***VJTech Academy, Maharashtra Contact No: +91-9730087674 / 7743909870***
***Email id: vjtechacademy@gmail.com)]***

## ❖ UNIT-II : Searching Sorting: Total Marks: 12

1. **Define internal and external sorting.**

   **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.

   **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.

2. **Define searching and give its type.**

   **Searching:** It is the process of finding a data element in the given data structure is known as searching.

   **Types:**

   1. Linear search
   2. Binary search

3. **Define sorting and give its type.**

   **Sorting:** It is a process of arranging collection of items in ascending order or in descending order.

   **The sorting is classified into two categories:**

   1. Internal Sorting.
   2. External Sorting.

4. **List any four sorting techniques.**

   1. Bubble sort
   2. Selection sort
   3. Insertion sort
   4. Radix sort
   5. Shell sort
   6. Quick sort

7. Merge sort

5. **Write an algorithm for Selection sort.**

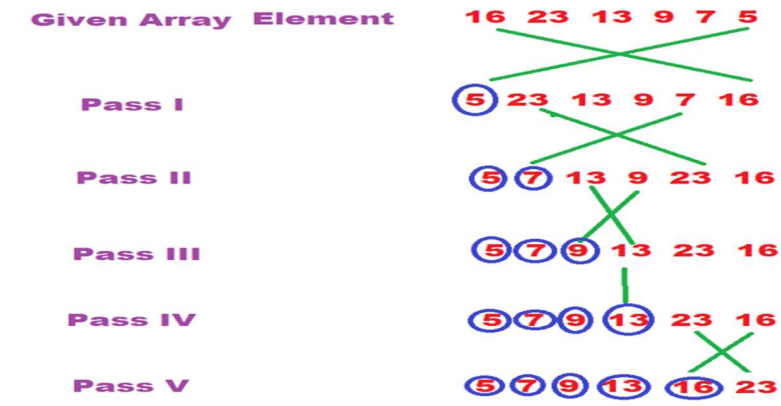
- In selection sort, the first element in the list is selected and it is compared repeatedly with remaining all the elements in the list.
- If any element is smaller than the selected element, then both are swapped.
- At the end of 1st pass smallest element is placed at first position.
- In 2nd pass, 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.

6. **Describe working of selection sort method with suitable example**

- Selection sort is the one of the types 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:**

**Selection Sort** {16,23,13,9,7,5}




7. **Differentiate between Binary Search and Sequential Search.**

**Ans:**

| Linear Search | Binary Search |
|---|---|
| Linear search compares each element with search element, while searching an element in the list. | Binary search computes mid element. It compares mid element with search element, while searching element in the list. |
| Linear search works on both sorted and unsorted input list | Binary search works only on sorted list. |
| Linear search take more time to search as it compares with each element. | Binary search requires less time to search as it compares only mid element with search element. If they are not equal then it reduces the list for comparison. |
| linear search has complexity O(n) | Binary search has complexity O(log n) |
| Linear search supports sequential access of elements. | Binary search supports random access of elements. |
| Good technique for small set of elements | Binary search is recommended when number of elements are large |

8. **Describe working of radix sort along with example.**

**Radix Sorting: -**

- 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 0th bucket to 9th bucket.

**Example:**

**Sort given elements using Radix Sort : 132, 543, 783, 63, 7, 49, 898**

**Given elements:** 132 543 783 063 007 049 898

PASS-I :

| Bucket No / Elements | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 132 | | | 132 | | | | | | | |
| 543 | | | | 543 | | | | | | |
| 783 | | | | 783 | | | | | | |
| 063 | | | | 063 | | | | | | |
| 007 | | | | | | | | 007 | | |
| 049 | | | | | | | | | | 049 |
| 898 | | | | | | | | | 898 | |

**Output of pass-I :** 132 543 783 063 007 898 049

PASS-II :

| Bucket No / Elements | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 132 | | | | 132 | | | | | | |
| 543 | | | | | 543 | | | | | |
| 783 | | | | | | | | | 783 | |
| 063 | | | | | | | 063 | | | |
| 007 | 007 | | | | | | | | | |
| 898 | | | | | | | | | | 898 |
| 049 | | | | | 049 | | | | | |

**Output of Pass-II :** 007 132 543 049 063 783 898

PASS-III

| Bucket No / Elements | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 007 | 007 | | | | | | | | | |
| 132 | | 132 | | | | | | | | |
| 543 | | | | | | 543 | | | | |
| 049 | 049 | | | | | | | | | |
| 063 | 063 | | | | | | | | | |
| 783 | | | | | | | | 783 | | |
| 898 | | | | | | | | | 898 | |

**Sorted Elements:** 7 49 63 132 543 783 898

9. **Describe quick sort. State its advantages and disadvantages.**

- 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.

**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:**

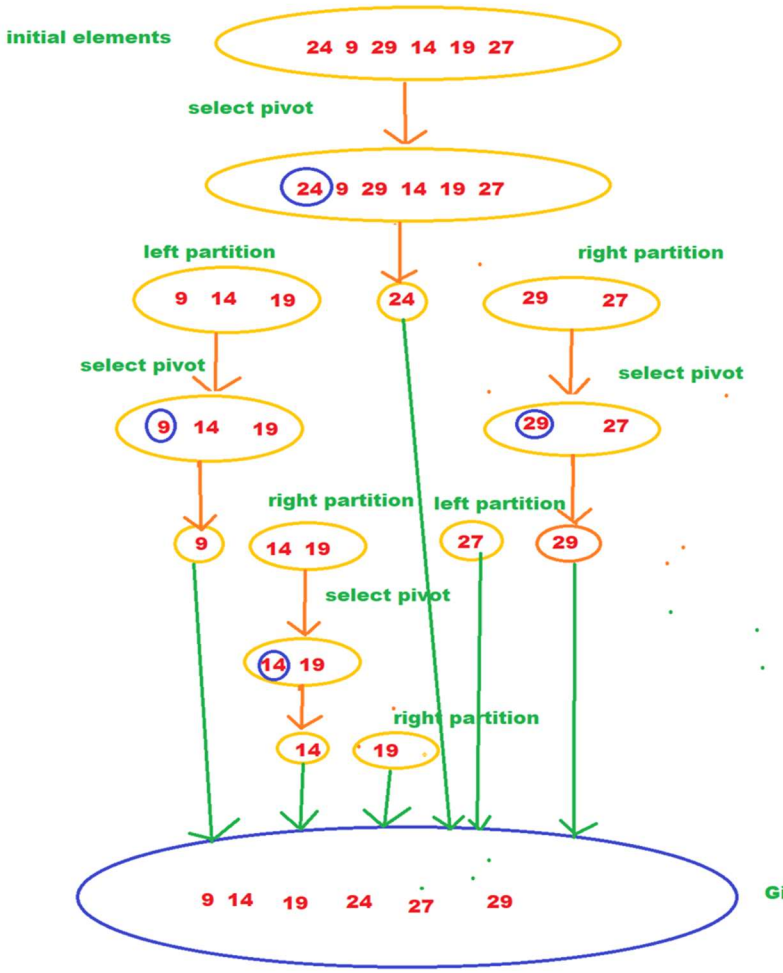
- It is complex and massively recursive.
- The worst-case complexity of quick sort is O(n^2), which is worse than the O (n log n) worst-case complexity of algorithms like merge sort, heapsort, binary tree sort, etc.

**Example:**

**Quick Sort:** Given elements = {24 9 29 14 19 27 }

Given elements ={24 9 29 14 19 27}

Divide and conque

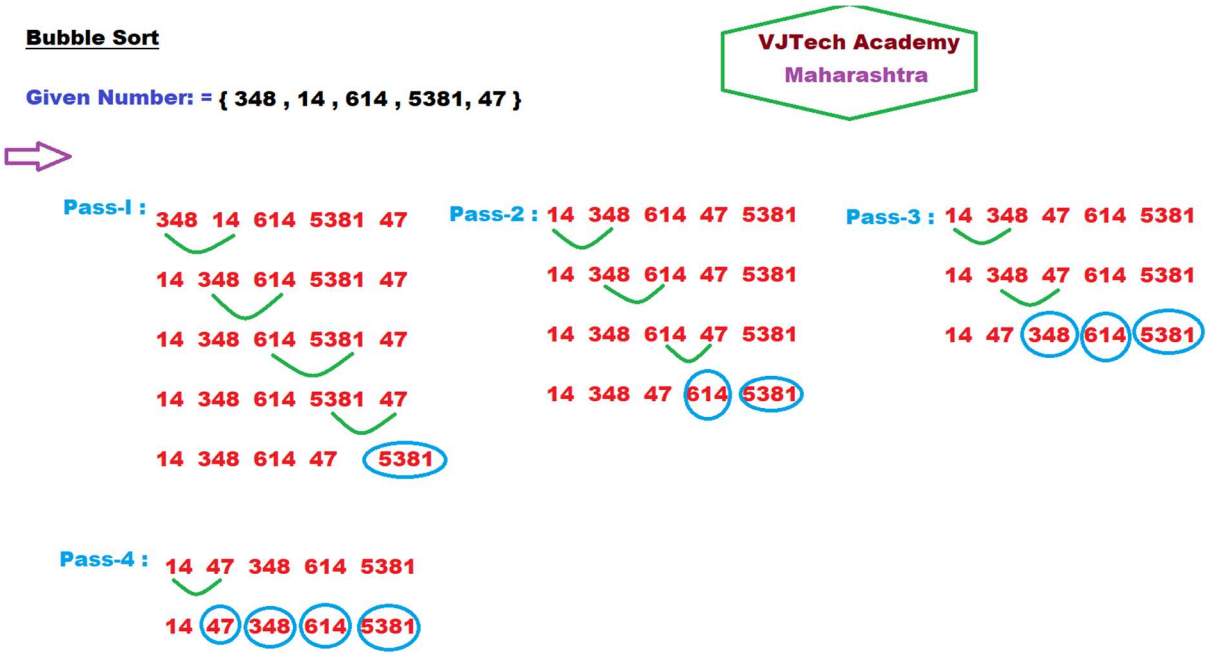



Given list sorted in ascending order

10. **Write an algorithm to implement 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.

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



12. **Sort number in ascending order using bubble sort. 19, 2, 27, 3, 7, 5, 31.**

**Bubble Sort**

VJTech Academy
Maharashtra

**Given Number: ={19, 2, 27, 3, 7, 5, 31}**

**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-2 :**
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-3 :**
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-4 :**
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-5 :**
2 3 5 7 19 27 31
2 3 5 7 19 27 31
2 3 5 7 19 27 31

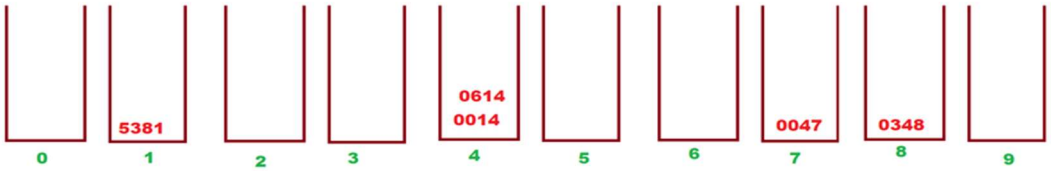
**Pass-6 :**
2 3 5 7 19 27 31
2 3 5 7 19 27 31

13. **With explanation sort the given no. in ascending order using radix sort. Given Numbers : 348, 14, 614, 5381, 47**

**Radix Sort :** Given Elements={ 348, 14, 614, 5381, 47}

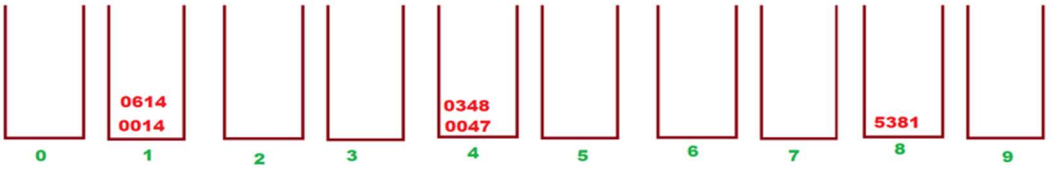
Given elements: 0348 0014 0614 5381 0047

Pass-I :



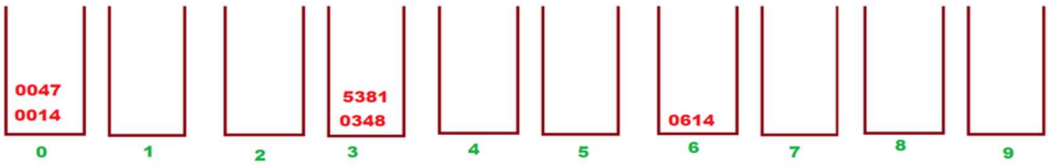

Merged List: 5381 0014 0614 0047 0348

Pass-II




Merged List: 0014 0614 0047 0348 5381

Pass-III




Merged list: 0014 0047 0348 5381 0614

Pass-III


Merged list: 0014 0047 0348 0614 5381

Sorted Elements={14 47 348 614 5381}

14. **Perform radix sort on the following list to arrange all array elements in ascending order : 132, 543, 783, 63, 7, 49, 898**

**Sort given elements using Radix Sort : 132, 543, 783, 63, 7, 49, 898**

**Given elements:** 132 543 783 063 007 049 898

**PASS-I :**

| Bucket No / Elements | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 132 | | | 132 | | | | | | | |
| 543 | | | | 543 | | | | | | |
| 783 | | | | 783 | | | | | | |
| 063 | | | | 063 | | | | | | |
| 007 | | | | | | | | 007 | | |
| 049 | | | | | | | | | | 049 |
| 898 | | | | | | | | | 898 | |

**Output of pass-I :** 132 543 783 063 007 898 049

**PASS-II :**

| Bucket No / Elements | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 132 | | | | 132 | | | | | | |
| 543 | | | | | 543 | | | | | |
| 783 | | | | | | | | | 783 | |
| 063 | | | | | | | 063 | | | |
| 007 | 007 | | | | | | | | | |
| 898 | | | | | | | | | | 898 |
| 049 | | | | | 049 | | | | | |

**Output of Pass-II :** 007 132 543 049 063 783 898

**PASS-III**

| Bucket No / Elements | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 007 | 007 | | | | | | | | | |
| 132 | | 132 | | | | | | | | |
| 543 | | | | | | 543 | | | | |
| 049 | 049 | | | | | | | | | |
| 063 | 063 | | | | | | | | | |
| 783 | | | | | | | | 783 | | |
| 898 | | | | | | | | | 898 | |

**Sorted Elements:** 7 49 63 132 543 783 898

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

**Insertion Sort : Given Elements={ 348, 14, 614, 5381, 47 };**

**initial elements**

| 348 | 14 614 5381 47 |
|---|---|
| Sorted Elements | Unsorted Elements |

**Pass-I :** Insert 14

| 14 348 | 614 5381 47 |
|---|---|
| Sorted Elements | Unsorted Elements |

**Pass-II :** Insert 614

| 14 348 614 | 5381 47 |
|---|---|
| Sorted Elements | Unsorted Elements |

**Pass-III :** Insert 5381

| 14 348 614 5381 | 47 |
|---|---|
| Sorted Elements | Unsorted Elements |

**Pass-IV :** Insert 47

| 14 47 348 614 5381 | Empty |
|---|---|
| Sorted Elements | Unsorted Elements |

**Sorted Elements={ 14 47 348 614 5381 }**

16. **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}**

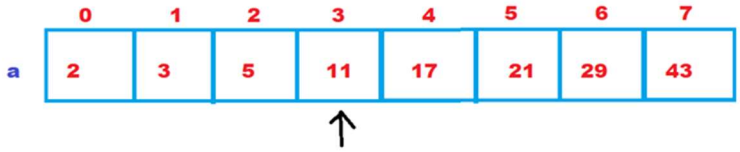
**Example-1:** Given elements={ 11, 5, 21, 3, 29, 17, 2, 43} key=29

⇨

**STEP-1 :** Given element list is not in sorted order so we have to sort it.

Lower=0 Upper=7 key= 29

mid=(Lower+Upper)/2 = (0+7)/2 = 3



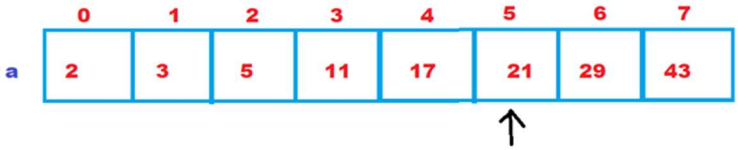

key!=a[mid]

key>a[mid]; Lower=(mid+1) = (3+1) = 4

**STEP-2**

Lower= 4 Upper=7 key= 29

mid=(Lower+Upper)/2 = (4+7)/2 = 5



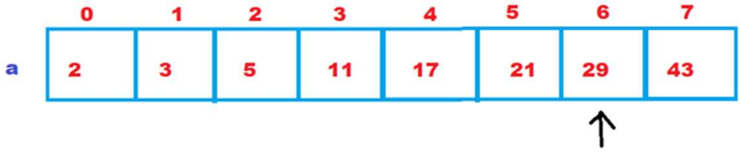

key!=a[mid]

key>a[mid]; Lower=(mid+1) = (5+1) = 6

**STEP- 3**

Lower= 6 Upper=7 key= 29

mid=(Lower+Upper)/2 = (6+7)/2 = 6




key==a[mid] Element is found

**Key element 29 is found at position 6.**

**Write a 'C' program to perform bubble sort on array of size N.**

```
#include <stdio.h>
int main()
{
   int a[20],N,i,j,temp;
   printf("\nHow much elements wants to store on 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("\nYour Sorted Array Elements:");
   for(i=0;i<N;i++)
   {
      printf("%d  ",a[i]);
   }
   return 0;
}
```

17. **Write a program for Binary search**

```
#include <stdio.h>
int main()
{
   int a[10],N,i,j,mid,key,flag=0;
   printf("\nEnter Size of Array:");
   scanf("%d",&N);
   printf("\nEnter Array Elements:");
   for(i=0;i<N;i++)
   {
      scanf("%d",&a[i]);
   }
   printf("\nEnter key element for searching:");
   scanf("%d",&key);
   i=0;
   j=N-1;
   mid=(i+j)/2;
   while(key!=a[mid]&&i<=j)
   {
      if(key<a[mid])
      {
         j=mid-1;
      }
      else
      {
         i=mid+1;
      }
      mid=(i+j)/2;
   }
   if(i<=j)
   {
      printf("\nElement is Found");
   }
   else
   {
      printf("\nElement is not Found");
   }
   return 0;
}
```

18. **Write a program for selection sort.**

```
#include <stdio.h>
int main()
{
   int a[20],N,i,k,j,temp;
   printf("\nHow much elements wants to store on array:");
   scanf("%d",&N);
   printf("\nEnter array elements:");
   for(i=0;i<N;i++)
   {
      scanf("%d",&a[i]);
   }
   for(i=0;i<N-1;i++)
   {
      k=i;
      for(j=i+1;j<N;j++)
      {
         if(a[j]<a[k])
         {
            k=j;
         }
      }
      if(i!=k)
      {
         temp=a[i];
         a[i]=a[k];
         a[k]=temp;
      }
   }
   printf("\nYour Sorted Array Elements:");
   for(i=0;i<N;i++)
   {
      printf("%d  ",a[i]);
   }
   return 0;
}
```

19. **Write a program to search an element in an array. Display position of element.**

```
#include <stdio.h>
int main()
{
   int a[10],N,i,key,flag=0;
   printf("\nEnter Size of Array:");
   scanf("%d",&N);
   printf("\nEnter Array Elements:");
   for(i=0;i<N;i++)
   {
      scanf("%d",&a[i]);
   }
   printf("\nEnter key element for searching:");
   scanf("%d",&key);
   for(i=0;i<N;i++)
   {
      if(key==a[i])
      {
         flag=1;
         break;
      }
   }
   if(flag==1)
   {
      printf("\nElement is Found at %d position",i+1);
   }
   else
   {
      printf("\nElement is not Found");
   }
   return 0;
}
```

20. **Describe working of binary search method. Give stepwise procedure to search 65 in the following list : List : 23, 12, 5, 29, 10, 65, 55, 70**

- 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.

**Searching Element in Given List:**

List: 23, 12, 5, 29, 10, 65, 55, 70

Pre-condition for Binary search is array elements must be in ascending order.

The given list is not sorted.

**Sorted List A**= {5, 10, 12, 23, 29, 55, 65, 70}

**Search element (k)** = 65

i. Low=0 high=7

Mid= (0+7)/2 = 3

A[mid] =a [3] =23

65>23

k>a [mid]

ii. Low=mid+1 High=7 Mid= (4+7)/2=5

A[mid] =a [5] =29

65>29

iii. Low=6 high=7

Mid= (6+7)/2 = 6

A[mid] =a [6] =65

A[mid] =k

Therefore key element is found at 6th position, no. of comparison required = 3.

- Search is successful

21. **Write a program for linear search. Find position of element 30 using linear search algorithm in given sequence. 10 5 20 25 8 30 40**

**Program:**

```
#include <stdio.h>
int main()
{
   int a[10],N,i,key,flag=0;
   printf("\nEnter Size of Array:");
   scanf("%d",&N);
   printf("\nEnter Array Elements:");
   for(i=0;i<N;i++)
   {
      scanf("%d",&a[i]);
   }
   printf("\nEnter key element for searching:");
   scanf("%d",&key);
   for(i=0;i<N;i++)
   {
      if(key==a[i])
      {
         flag=1;
         break;
      }
   }
   if(flag==1)
   {
      printf("\nElement is Found");
   }
   else
   {
      printf("\nElement is not Found");
   }
   return 0;
}
```

**Steps to find position of element 30:**

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| array A | 10 | 5 | 20 | 25 | 8 | 30 | 40 |

**Search 30**

**step 1:** compare element at position 0 i.e 10 with 30

10==30 NO, increment position by 1

**step 2:** compare Element at position 1 i.e 05 with 30

5==30 NO, increment position by 1

**step 3:** compare Element at position 2 i.e 20 with 30

20==30 NO, increment position by 1

**step 4:** compare Element at position 3 i.e 25 with 30

25==30 NO, increment position by 1

**step 5:**compare Element at position 4 i.e 8 with 30

8==30 NO, increment position by 1

**step 6:** compare Element at position 5 i.e 30 with 30

30==30 YES , SEARCH IS SUCCESSFUL AND ELEMENT IS PRESENT AT POSITION 5


# DATA STRUCTURE IMPORTANT QUESTIONS LIST WITH ANSWERS

**Author:**

*"Prof. Vishal Jadhav"*

*[BE in Computer Engineering and Having 4.7 years of IT industry experience.*
*VJTech Academy, Awasari(KH), Contact No: +91-9730087674,*
*Email id: vjtechacademy@gmail.com)]*

❖ **UNIT-III : Stacks and Queues:** **Total Marks: 20**

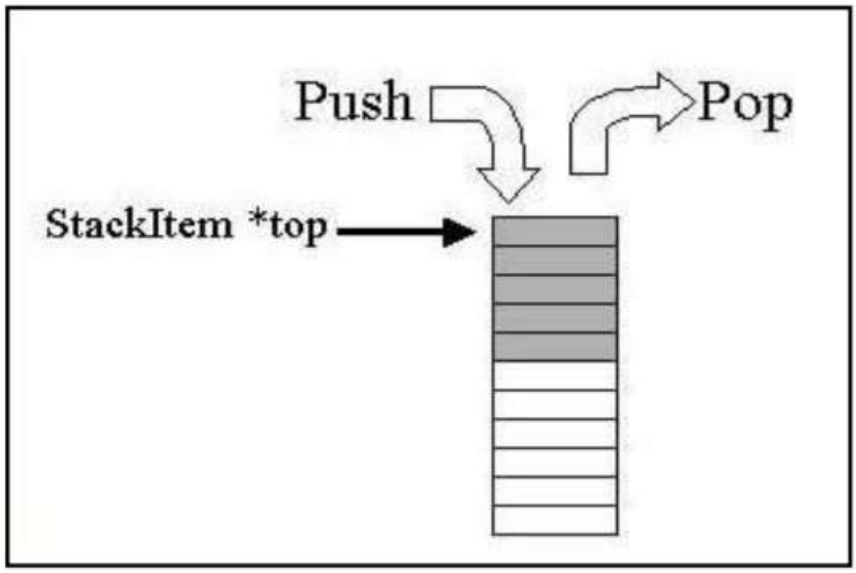
1. **Describe stack as an abstract data type**

**Ans:**

Stack is a linear data structure in which data elements are stored in a specific sequence. It works if LIFO manner i.e. Last In First Out. It has one pointer called as top. Data elements are push and pop using top pointer inside stack. Stack as an abstract data type contains stack elements and operations performed on it.

**Stack elements are:** Array in which data elements are stored and stack top pointer to perform operations on stack.

**Diagram:**




**Stack operations include:**

- **Initialize**: set stack to empty
- Checking whether **stack is empty** or not
- Checking if **stack is overflow/full** or not
- Insert a new element. This operation is called as **push.**
- Delete an element from stack. This operation is called as **pop**.

2. **Describe Queue as an abstract data type**

**Ans: Try to write answer as per question 1.**

3) **Queue as an Abstract Data Type:** Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. 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)** principle.

3. **Distinguish between stack and queue**

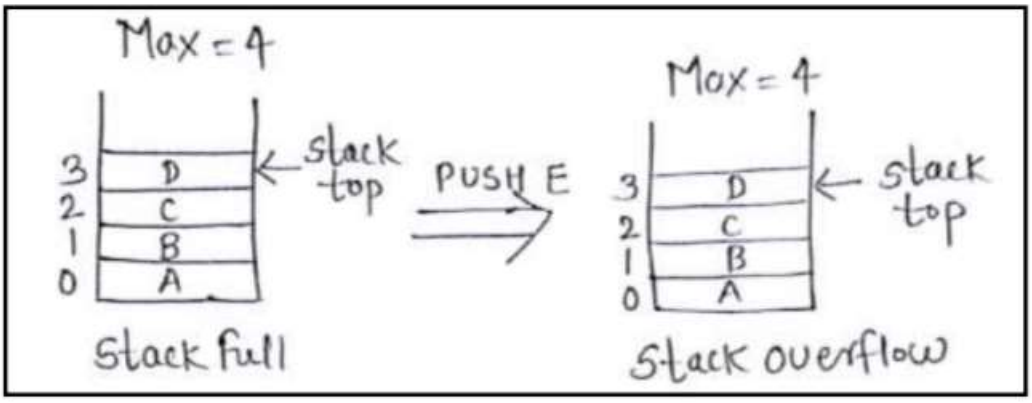
**Ans:**

| Stack | Queue |
|---|---|
| 1. In Stack insertion and deletion operations are performed at same end. | 1. In Queue insertion and deletion operations are performed at different end. |
| 2.In stack the element which is inserted last is first to delete so it is called Last In First Out | 2.In Queue the element which is inserted first is first to delete so it is called First In First Out |
| 3.In stack only one pointer is used called as Top | 3.In Queue two pointers are used called as front and rear |
| 4.In Stack Memory is not wasted | 4. In Queue memory can be wasted/ unusable in case of linear queue. |
| 5. Stack of books is an example of stack | 5. Students standing in a line at fees counter is an example of queue |
| 6.Application:Recursion,Polish notation | 6. Application: In computer system for organizing processes. In mobile device for sending receiving messages. |

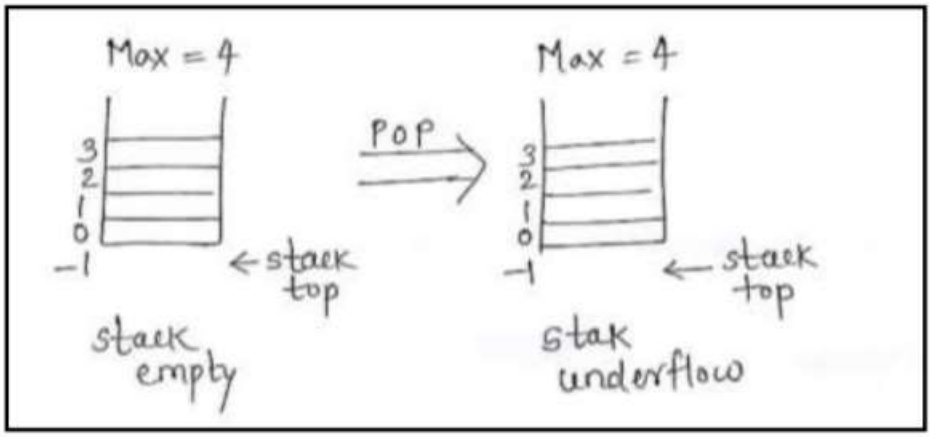
4. **Describe following terms with suitable diagram: i) Stack overflow ii) Stack underflow.**

**Ans:**

**Stack overflow**: When stack contains maximum number of elements i.e. stack is full and push operation is called to insert a new element then stack is said to be in overflow state.




**Stack underflow:** When there is no elements in a stack i.e. stack is empty and pop operation is called then stack is said to be in underflow state



5. **Describe following terms with suitable diagram: i) Queue overflow ii) Queue underflow.**

**Ans: Try to write answer as per question 4.**

**Queue Overflow:** Inserting an element in a queue which is already full is known as Queue Overflow condition (Rear = Max-1)

**Queue Underflow:** Deleting an element from queue which is already empty is known as Queue Underflow condition (Front == Rear=-1)

6. **Give any two applications of stack.**

**Ans: This question answer you people already know.**

**Applications for Stack:-**

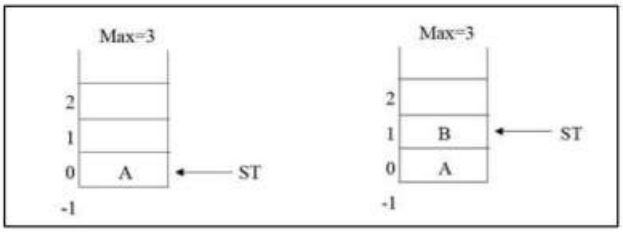
1. Recursion
2. Polish Notation Conversion
3. Expression Evaluation
4. Reversal of List

7. **Explain four primitive operations on stack.**

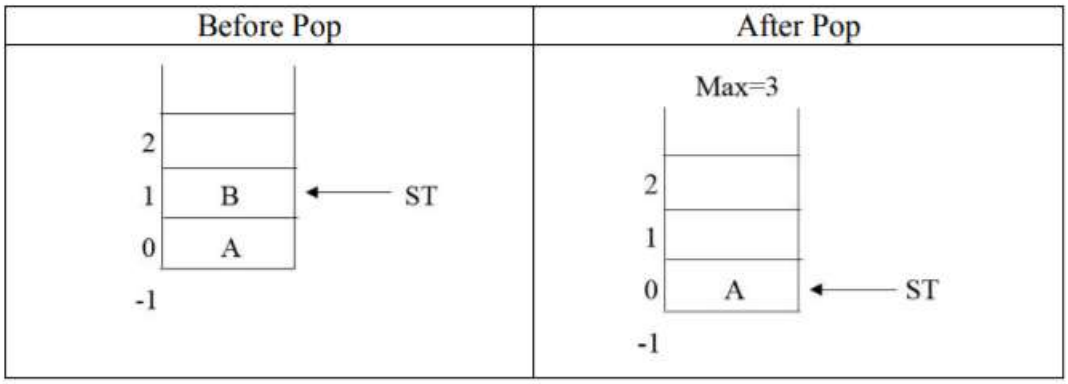
**Ans:**

1. push() : This operation is used to insert an element in a stack. Inserting an element in stack requires increment of stack top by one position and then store data element in it.

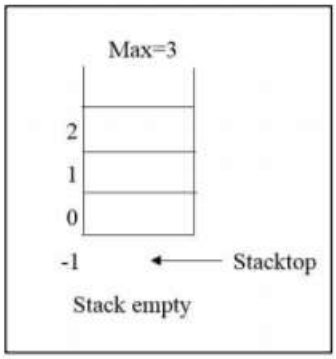
Push B



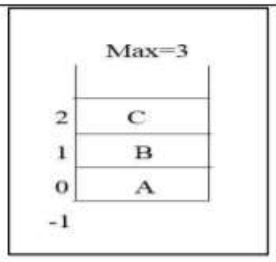

2. pop(): This operation is used to remove an element from stack. Removing an element from stack requires decrement of stack top by one position.



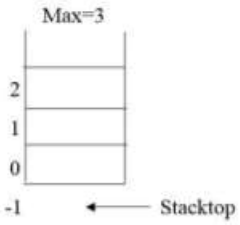

3. isempty(): This operation is used to check whether stack is empty or not. It checks stack top position. If stack top is -1 then stack is empty.



4. isfull(): This operation is used to check whether stack is full or not. It checks stack top position. If stack top is maximum size -1 then stack is full.



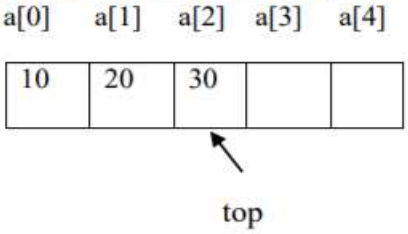

5. initialize(): This operation is used to initialize stack top to –1 value.



8. **Show the memory representation of Stack using array with the help of a diagram.**

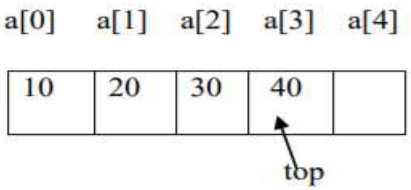
**Ans:**

1. **Definition:** Stack is a linear data structure which follows Last-In First- Out (LIFO) principle where, elements are added and deleted from only one end called as stack top
2. **Array representation of a Stack:**

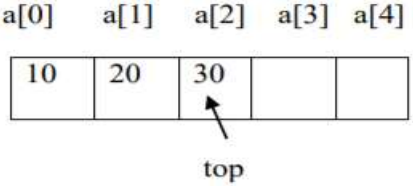



3. **PUSH Operation:**
   a. If (top==max-1)
      then display "Stack overflow"
   b. else
      do top=top+1
   c. a[top]=data
4. **POP Operation:**
   a. If (top==- 1)
      then display "Stack underflow"
   b. else
      do Data= a[top]
         Top=top-1

5. **Example:**
   a. PUSH 40




   b. POP



9. **Describe push and pop operations on stack. Also write algorithm for push and pop operations.**

push() : This operation is used to insert an element in a stack. Inserting an element in stack requires increment of stack top by one position and then store data element in it.

**Push algorithm: - Max is maximum size of stack.**

**Step 1:** [Check for stack full/ overflow]
If stack_top is equal to max-1 then
Display output as "Stack Overflow" and return to calling function
Otherwise
Go to step 2

**Step 2:** [Increment stack_top]
Increment stack top pointer by one.
stack_top=stack_top +1;

**Step 3:** [Insert element]
stack [stack_top] = item;

**Step 4:** return to calling function

pop(): This operation is used to remove an element from stack. Removing an element from stack requires decrement of stack top by one position.

**Pop Algorithm:**

**Algorithm:**

**Step 1:** [Check for stack empty/ underflow]
If stack top is equal to -1 then write "Stack Underflow"
return

**Step 2:** [Copy data]
item=stack[top];

**Step 3 :** [decrement top]
top = top-1;

**Step 4 :** return

10. **Recursion is one of the application of stack-YES/NO ? Explain it for calculating the factorial of a number 5 using recursion**

**Ans:** Yes, Recursion is one of the application of stack

1. Recursion is calling a function from itself repeatedly. A function call to the recursive function is written inside the body of a function.
2. In the recursive call each time a function executes the same number of statements repeatedly. Each function contain local variables.
3. When a recursive function is called, before executing the same function again, the local variables are saved in the data structure stack. This way in each execution local variables values are copied to the stack.
4. When the recursive function terminates one by one each element is removed from the stack and we get the result.

**Example:** Factorial of number, Tower of Hanoi. Fibonacci Series

```
int factorial (int no)
{
If(no==1)
 Return 1;
Else
Fact=fact*factorial(no-1);
}
```

Fact= fact* factorial (no-1); This statement gives recursive call to the function.

Each time when factorial function is called stack stores the value of current local variables and then next variable. If factorial of 5 then first time stack will have 5 then in 2nd call 4 … till the last call stack gets the element. Once it is empty all variable values are pop & result is calculated.

11. **Convert the following infix expression to its postfix form using stack A + B – C*D/E + F.**

| OPERATOR | PRECEDENCE |
|---|---|
| Exponentiation ($ or ↑ or ^) | Highest |
| *, / | Next highest |
| +, - | Lowest |

**###Convert Infix Expression to Postfix Expression##**

**Question:** **Infix Expression: A + B – C*D/E + F**

| Input Expression | Stack | Output Expression |
|---|---|---|
| A+B–C*D/E+F | empty | - |
| +B-C*D/E+F | empty | A |
| B-C*D/E+F | + | A |
| -C*D/E+F | + | AB |
| C*D/E+F | - | AB+ |
| *D/E+F | - | AB+C |
| D/E+F | - * | AB+C |
| /E+F | - * | AB+CD |
| E+F | - / | AB+CD* |
| +F | - / | AB+CD*E |
| F | + | AB+CD*E/- |
| empty | + | AB+CD*E/-F |
| empty | empty | AB+CD*E/-F+ |

**Postfix Expression AB+CD*E/-F+**

12. **Convert following infix expression into a postfix expression. Show all steps. (P + (Q * R – (S/T ↑ U) * V) * W)**

| Input Expression | Stack | Output Expression |
|---|---|---|
| (P+(Q*R–(S/T^U)*V)*W) | EMPTY | - |
| P+(Q*R-(S/T^U)*V)*W) | ( | - |
| +(Q*R-(S/T^U)*V)*W) | ( | P |
| (Q*R-(S/T^U)*V)*W) | (+ | P |
| Q*R-(S/T^U)*V)*W) | (+( | P |
| *R-(S/T^U)*V)*W) | (+( | PQ |
| R-(S/T^U)*V)*W) | (+(* | PQ |
| -(S/T^U)*V)*W) | (+(* | PQR |
| (S/T^U)*V)*W) | (+(- | PQR* |
| S/T^U)*V)*W) | (+(-( | PQR* |
| /T^U)*V)*W) | (+(-( | PQR*S |
| T^U)*V)*W) | (+(-(/ | PQR*S |
| ^U)*V)*W) | (+(-(/ | PQR*ST |
| U)*V)*W) | (+(-(/^ | PQR*ST |
| )*V)*W) | (+(-(/^ | PQR*STU |
| *V)*W) | (+(- | PQR*STU^/ |
| V)*W) | (+(-* | PQR*STU^/ |
| )*W) | (+(-* | PQR*STU^/V |
| *W) | (+ | PQR*STU^/V*- |
| W) | (+* | PQR*STU^/V*- |
| ) | (+* | PQR*STU^/V*-W |
| EMPTY | EMPTY | PQR*STU^/V*-W*+ |

13. **Convert the given infix expression in to prefix expression. Write all steps for conversion (a * b + c/d) * (e + f g).**

**Question:** **Infix Expression:** **(a*b+c/d)*(e+f*g)**

**Step-1: Reverse the given infix expression**

**)g*f+e(*)d/c+b*a(**

**Step-2: Make every '(' as ')' and every ')' as '('**

**(g*f+e)*(d/c+b*a)**

**Step-3: Convert expression to postfix form**

| Input Expression | Stack | Output Expression |
|---|---|---|
| ( | ( | empty |
| g | ( | g |
| * | (* | g |
| f | (* | gf |
| + | (+ | gf* |
| e | (+ | gf*e |
| ) | empty | gf*e+ |
| * | * | gf*e+ |
| ( | *( | gf*e+ |
| d | *( | gf*e+d |
| / | *(/ | gf*e+d |
| c | *(/ | gf*e+dc |
| + | *(+ | gf*e+dc/ |
| b | *(+ | gf*e+dc/b |
| * | *(+* | gf*e+dc/b |
| a | *(+* | gf*e+dc/ba |
| ) | * | gf*e+dc/ba*+ |
| empty | empty | gf*e+dc/ba*+* |

**Step-4 : Reverse postfix expression**

**prefix expression : *+*ab/cd+e*fg**

14. **Convert the following infix expression to its prefix form using stack A + B ^ C*(D/E) – F%G.**

**Show diagrammatically each step of conversion.**

| *, / , % |
|---|
| +, - |

**Question:** **Infix Expression: A+B^C*(D/E)-F%G**

⇨

**Step-1: Reverse the given infix expression**

G%F-)E/D(*C^B+A

**Step-2: Make every '(' as ')' and every ')' as '('**

G%F-(E/D)*C^B+A

**Step-3: Convert expression to postfix form**

| Input Expression | Stack | Output Expression |
|---|---|---|
| G | EMPTY | G |
| % | % | G |
| F | % | GF |
| - | - | GF% |
| ( | -( | GF% |
| E | -( | GF%E |
| / | -(/ | GF%E |
| D | -(/ | GF%ED |
| ) | - | GF%ED/ |
| * | -* | GF%ED/ |
| C | -* | GF%ED/C |
| ^ | -*^ | GF%ED/C |
| B | -*^ | GF%ED/CB |
| + | + | GF%ED/CB^*- |
| A | + | GF%ED/CB^*-A |
| EMPTY | EMPTY | GF%ED/CB^*-A+ |

**Step-4 : Reverse postfix expression**

**prefix expression: +A-*^BC/DE%FG**

15. **Translate below expression into prefix:**

$(x + y) * (p - q).$

**After that evaluate the above prefix expression using below values.**

$x = 2, y = 3, p = 6, q = 2.$

**Step-1: Reverse the given expression.**

)q-p(*)y+x(

**Step-2: Make every '(' as ')' and every ')' as '('.**

(q-p)*(y+x)

**Step-3: Convert expression to postfix form.**

| Input Expression | Stack | Output Expression |
|---|---|---|
| (q-p)*(y+x) | empty | |
| q-p)*(y+x) | ( | |
| -p)*(y+x) | ( | q |
| p)*(y+x) | ( - | q |
| )*(y+x) | ( - | qp |
| *(y+x) | empty | qp- |
| (y+x) | * | qp- |
| y+x) | * ( | qp- |
| +x) | * ( | qp-y |
| x) | * ( + | qp-y |
| ) | * ( + | qp-yx |
| empty | * | qp-yx+ |
| empty | empty | qp-yx+* |

**Step-4: Reverse the postfix expression.**

**Prefix Form : *+xy-pq**

| Prefix Expression | Stack | Comment |
|---|---|---|
| *+xy-pq | | Initially stack empty |
| *+xy-p | 2 | push 2 |
| *+xy- | 6<br>2 | push 6 |
| *+xy | 4 | push 6-2 = 4 |
| *+x | 3<br>4 | push 3 |
| *+ | 2<br>3<br>4 | push 2 |
| * | 5<br>4 | push 2+3 = 5 |
| empty | 20 | push 5*4=20 |

16. **Evaluate the following prefix expression: - * + 4 3 2 5 Show diagrammatically each step of evaluation using stack.**

**Prefix Expression :** - * + 4 3 2 5 **Evaluation**

| Input Expression | Stack | Comments |
|---|---|---|
| - * + 4 3 2 5 | 5 | Push 5 |
| - * + 4 3 2 | 2<br>5 | Push 2 |
| - * + 4 | 3<br>2<br>5 | Push 3 |
| - * + | 4<br>3<br>2<br>5 | Push 4 |
| - * | 7<br>2<br>5 | 4+3 = 7<br>Push 7 |
| - | 14<br>5 | 7*2=14<br>Push 14 |
| empty | 9 | 14-5 = 9<br>Push 9 |

**Step-1:** Scan the the expression right to left

**Step-2:** If reading character is operand then push it onto the stack.

**Step-3:** If reading character is operator then pop two values from stack and perform operation and push result back onto the stack.

**Note:** first pop value become first operand and second pop value become second operand.

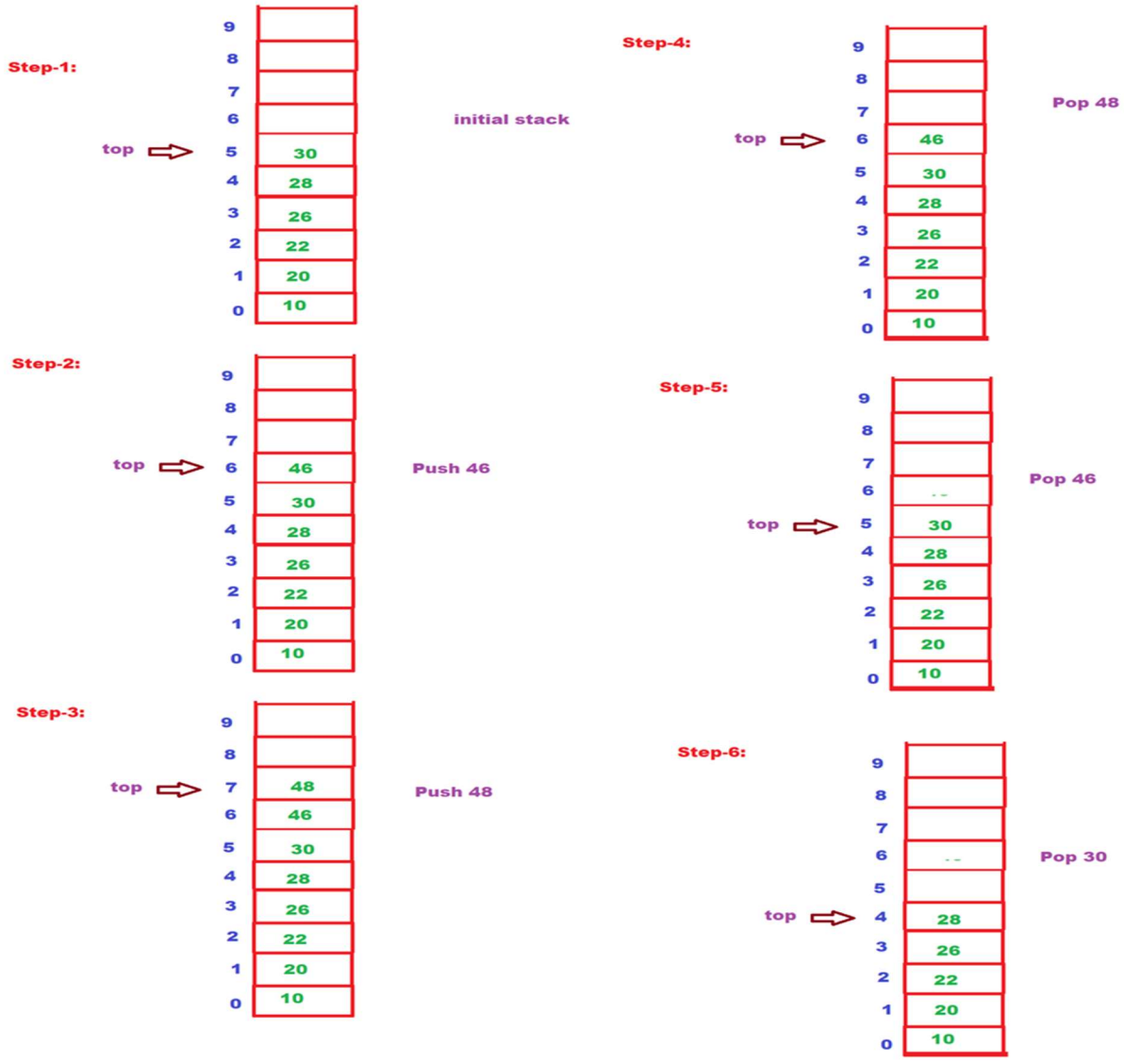
**Step 4:** Repeat above all process till input expression end

**Final Result**

17. **Show the effect of PUSH and POP operation on to the stack of size 10. The stack contains 10, 20, 22, 26, 28, and 30, with 30 being at top of the stack. Show diagrammatically the effect of**

**1. PUSH 46**
**2. PUSH 48**
**3. POP**
**4. POP**
**5. POP**

**Sketch the final structure of Stack after performing the above said operations. Sketch the final structure of Stack after performing the above said operations.**



18. **Write an algorithm for performing push operation on stack.**

**Ans:**

Step 1: Start

Step 2: if(top == SIZE-1)

print “Stack is Full” and goto step 5.

otherwise

goto Step 3.

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

## 19. Show the effect of INSERT and DELETE operations on to the linear queue of size 5.

**Show the effect of INSERT and DELETE operations on to the linear queue of size 5. Initially queue is empty. Show diagrammatically the effect of -**

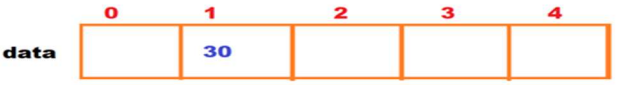
1. **INSERT (10)**
2. **INSERT (30)**
3. **DELETE**
4. **INSERT (50)**
5. **INSERT(45)**
6. **DELETE**

> **Initially queue is empty ( rear,front=-1 )**


**Step-1 : Insert 10 (front=0 , rear=0)**


**Step-2 : Insert 30 (front=0 , rear= 1)**

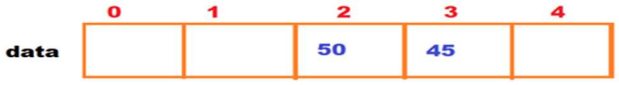

**Step-3 : Delete (front=1 , rear= 1)**




**Step-4 : Insert 50 (front=1, rear=2)**


**Step-5 : Insert 45 (front=1, rear= 3)**


**Step-6 : Delete (front= 2, rear= 3)**

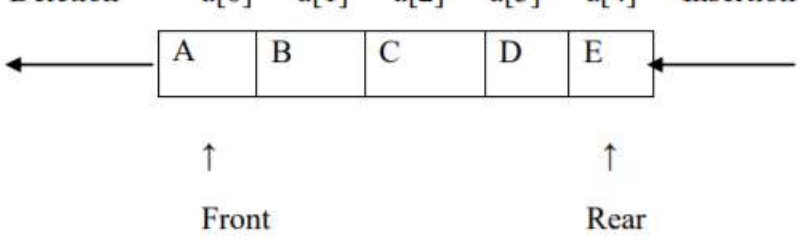


20. **Draw the diagram of queue to represent front and rear pointers of queue.**

**Ans:**

**Front: -** This end is used for deleting an element from a queue. Initially front end is set to --1. Front end is incremented by one when a new element has to be deleted from queue.

**Rear**: - This end is used for inserting an element in a queue. Initially rear end is set to -1.rear end is incremented by one when a new element has to be inserted in queue.




21. **Write algorithm to insert an element into a linear queue.**

**Step 1:** Start
**Step 2:** if(rear == SIZE-1)
print "queue is Full"
otherwise
goto Step 3.

**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 4:** Stop

22. **Explain concept of priority queue with example and its advantages.**

Ans:

A priority Queue is a collection of elements where each element is assigned a priority and the order in which elements are added into the queue.
The rules for processing the elements of priority queue are:

1) An element with higher priority is processed before any element of lower priority.
2) Two elements with the same priority are processed according to the order in which they are added to the queue (FCFS).

**Advantages:**

1) Preferences to the higher priority process are added at the beginning. High priority process executes first.
2) Keep the list sorted in increasing order.

23. **Write a 'C' program to insert an element in a queue.**

```
#include <stdio.h>
#define MAX 5
int data[MAX],rear,front;

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

int main()
{
   int ch;
   initialize();
   do
   {
      printf("\n****Queue MENU****");
      printf("\n1. Insertion");
      printf("\n2. Deletion");
      printf("\n3. Print");
      printf("\nEnter Your Choice:");
      scanf("%d",&ch);
      switch(ch)
      {
         case 1: insertion();
```

```
            break;
        case 2: deletion();
            break;
        case 3: print();
            break;
        case 4: printf("\nThanks for using our software");
            break;
        default:printf("\nInvalid choice");
      }
   }while(ch!=4);
   return 0;
}
void initialize()
{
   rear=front=-1;
}
int empty()
{
   if(rear==-1)
      return(1);
   else
      return(0);
}
int full()
{
   if(rear==MAX-1)
      return(1);
   else
      return(0);
}
void insertion()
{
   int x;
   if(full()==0)
   {
      printf("\nEnter Data:");
      scanf("%d",&x);
      if(rear==-1)
      {
         rear=front=0;
         data[rear]=x;
      }
      else
      {
```

```
        rear=rear+1;
        data[rear]=x;
      }
      printf("\nData inserted Successfully");
   }
   else
   {
      printf("\nQueue is FULL");
   }
}
void deletion()
{
   int x;
   if(empty()==0)
   {
      x=data[front];
      if(front==rear)
      {
         rear=front=-1;
      }
      else
      {
         front=front+1;
      }
      printf("\nDeleted Data: %d",x);
   }
   else
   {
      printf("\nQueue is EMPTY");
   }
}
void print()
{
   int i;
   printf("\nQueue Data:");
   for(i=front;i<=rear;i++)
   {
      printf("%d\t",data[i]);
   }
}
```

24. **Write a program for insert and delete operation perform on queue. State any two application of queue.**

```
#include <stdio.h>
#define MAX 5
int data[MAX],rear,front;

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

int main()
{
   int ch;
   initialize();
   do
   {
      printf("\n****Queue MENU****");
      printf("\n1. Insertion");
      printf("\n2. Deletion");
      printf("\n3. Print");
      printf("\nEnter Your Choice:");
      scanf("%d",&ch);
      switch(ch)
      {
         case 1: insertion();
               break;
         case 2: deletion();
               break;
         case 3: print();
               break;
         case 4: printf("\nThanks for using our software");
               break;
         default:printf("\nInvalid choice");
      }
   }while(ch!=4);
   return 0;
}
void initialize()
{
```

```
   rear=front=-1;
}
int empty()
{
   if(rear==-1)
      return(1);
   else
      return(0);
}
int full()
{
   if(rear==MAX-1)
      return(1);
   else
      return(0);
}
void insertion()
{
   int x;
   if(full()==0)
   {
      printf("\nEnter Data:");
      scanf("%d",&x);
      if(rear==-1)
      {
         rear=front=0;
         data[rear]=x;
      }
      else
      {
         rear=rear+1;
         data[rear]=x;
      }
      printf("\nData inserted Successfully");
   }
   else
   {
      printf("\nQueue is FULL");
   }
}
void deletion()
{
   int x;
   if(empty()==0)
```

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

**Application of queue:**

1) Simulation
2) CPU scheduling in multiprogramming and time sharing systems.
3) Queue is in round robin scheduling algorithm
4) Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
5) In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
6) Handling of interrupts in real-time systems.

25. **Write a code delete an element in queue.**

**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: Check queue is Empty or not. If empty then display message “Queue is empty” and goto step 7.
  Step 3: Delete front element from the queue i.e X=data[front]
  Step 4: if(front == rear) then
  set -1 to rear and front variables.
  otherwise
  goto Step 5.

  Step 5: Increment front by 1 i.e. front=front+1

  Step 6: Print deleted value is X.

  Step 7: Stop.

  **C Function:**

```
void deletion()
{
   int x;
   if(empty()==0)
   {
      x=data[front];
      if(front==rear)
      {
         rear=front=-1;
      }
      else
      {
         front=front+1;
      }
```

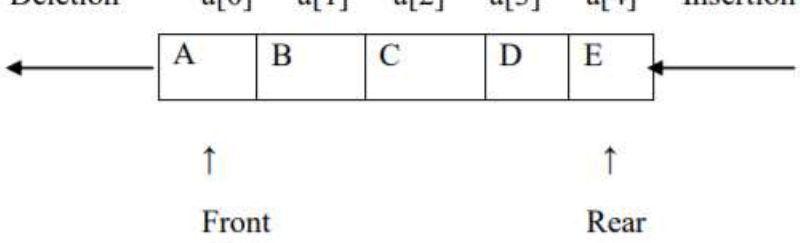
```
    printf("\nDeleted Data: %d",x);
  }
  else
  {
    printf("\nQueue is EMPTY");
  }
}
```

26. **Define the terms FRONT and REAR of queue. Enlist any four operations of queue.**

**Ans:**

**Front: -** This end is used for deleting an element from a queue. Initially front end is set to --1. Front end is incremented by one when a new element has to be deleted from queue.

**Rear**: - This end is used for inserting an element in a queue. Initially rear end is set to -1.rear end is incremented by one when a new element has to be inserted in queue.



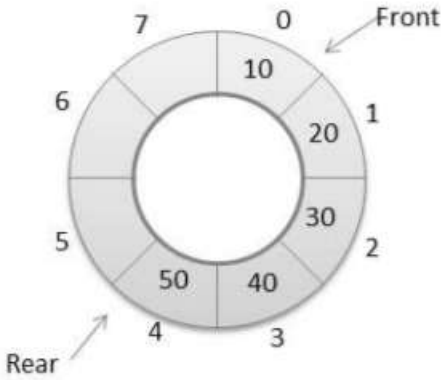

**Operations on end of queue:-**

- Insert
- Delete
- Search
- Traverse
- Display

27. **Draw and explain circular queue.**

**Ans:**

- A queue, in which the last node is connected back to the first node to form a cycle, is called as circular queue.
- It has rear pointer to insert an element and front pointer to delete an element. It works in FIFO manner where first inserted element is deleted first.




**Advantage of circular queue over linear queue:**

1. In linear queue, before insertion of element 'queue full' condition is checked. If rear pointer points to the maximum-1 position then queue is full even though front pointer is greater than 0. If front pointer is greater than 0 that means space is available at the beginning in which new element can be stored. If rear pointer indicates queue full then insertion cannot be done. So it leads to wastage of space.

2. Circular queue has advantage of utilization of space over linear queue. Circular queue is full only when there is no empty position in a queue. Before inserting an element in circular queue front and rear both the pointers are checked. So if it indicates any empty space anywhere in a queue then insertion takes place.


# DATA STRUCTURE IMPORTANT QUESTIONS LIST WITH ANSWERS

**Author:**

*"Prof. Vishal Jadhav"*

*[BE in Computer Engineering and Having 8.5 years of IT industry experience.*
*VJTech Academy, Awasari(KH), Contact No: +91-9730087674 / 7743909870*
*Email id: vjtechacademy@gmail.com)]*

## ❖ UNIT-IV : Linked List: Total Marks: 16

1. **Describe any two terms from the following: Node, Null pointer, empty list with respect to linear linked list with diagram.**

**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.

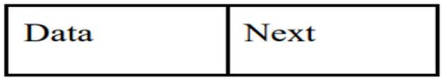



**Fig. Single Node structure**

**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.

**Empty list:**

- A linked list is said to be empty if head node contains NULL pointer. i.e. head=NULL.

2. **Describe any two terms from the following: information Field/ Data field, Address, Next Pointer with respect to circular linked list with diagram.**

**information Field/ Data field:**

- It is used to store the data inside the node.

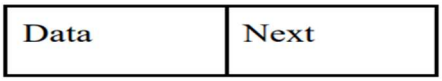



**Fig. Single Node structure**

**Next Pointer:**

- It is used to store the address of next node.

**Address:**

- In CLL, every node consists of two fields, first is data and second is address.
- Address field contain the memory address of next node.

3. **List and define operations of linked list.**

**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.

4. **Write algorithm for 'search' operation in an unsorted 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=p->next
```

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

5. **WAP to search an element in a linked list.**

```
void search(node *p)
{
   int key,flag=0;
   printf("\nEnter Key Node for Searching:");
   scanf("%d",&key);
   while(p!=NULL)
   {
      if(key==p->data)
      {
         flag=1;
         break;
      }
      p=p->next;
   }
   if(flag==1)
   {
      printf("\nNode is present in given SLL");
   }
   else
   {
      printf("\nNode is not present in given SLL");
   }
}
```

6. **Explain insertion at the beginning and at end operations on linked list with example.**

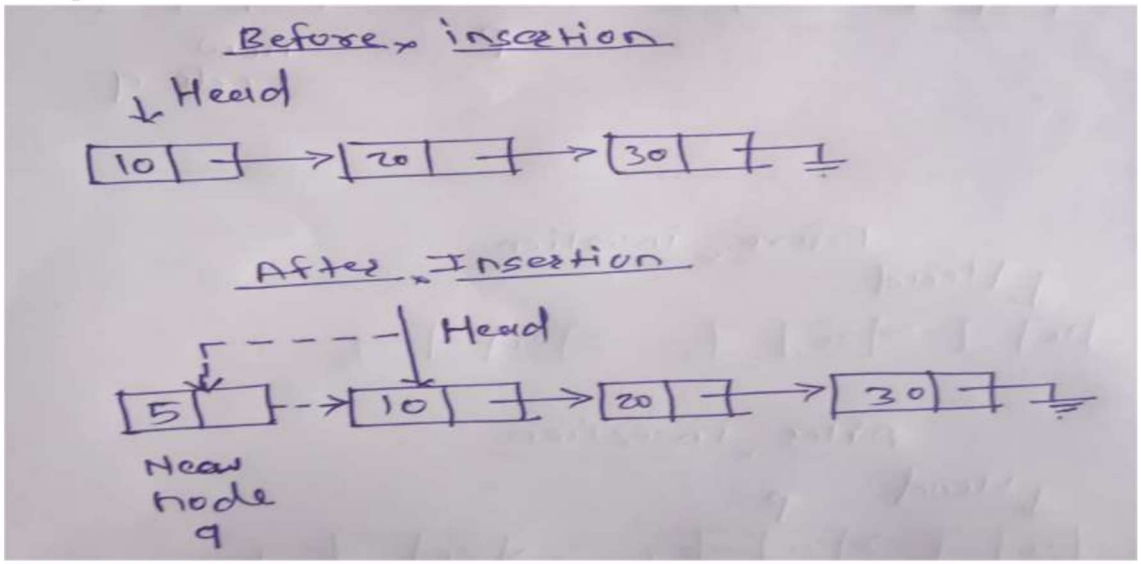
- In a single linked list, the insertion operation can be performed in three ways.
- They are as follows.

  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
- In this operation, first we create new node q and assign value to data and next field.
- Assign head to next field of q node.
- At last, make q node as head node.

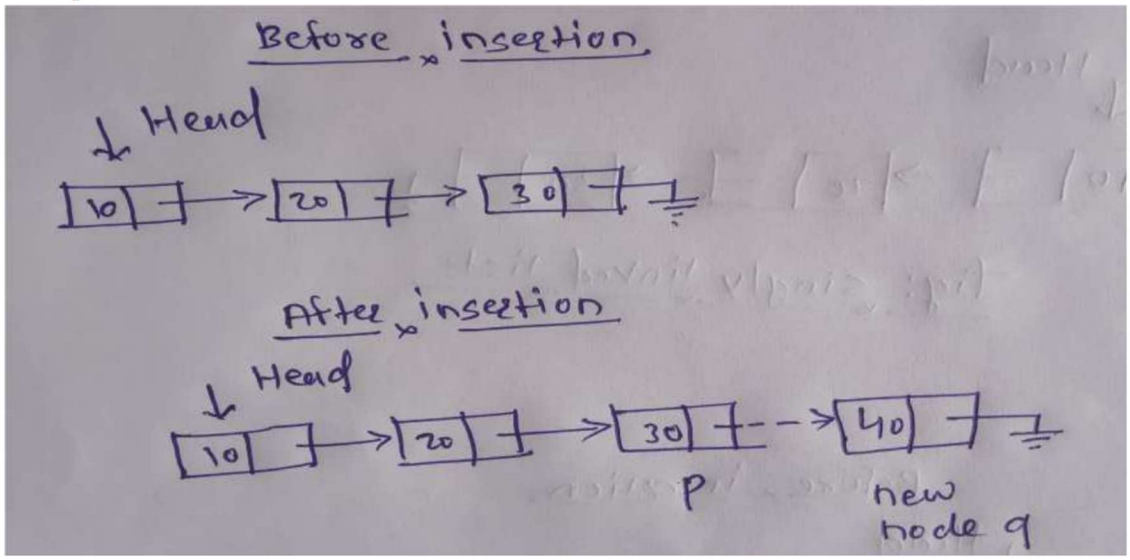
Example:




**Inserting At End of the list:**

- In this operation, new node can be created and added at the end of a list.
- In this operation, first we create new node q and assign value to data and next field.
- Then we assign head to pointer p and traverse pointer p to last node.
- At last make the link between p and q.

Example:




7. **Write a 'C' program to insert new node at the end of linear linked list.**

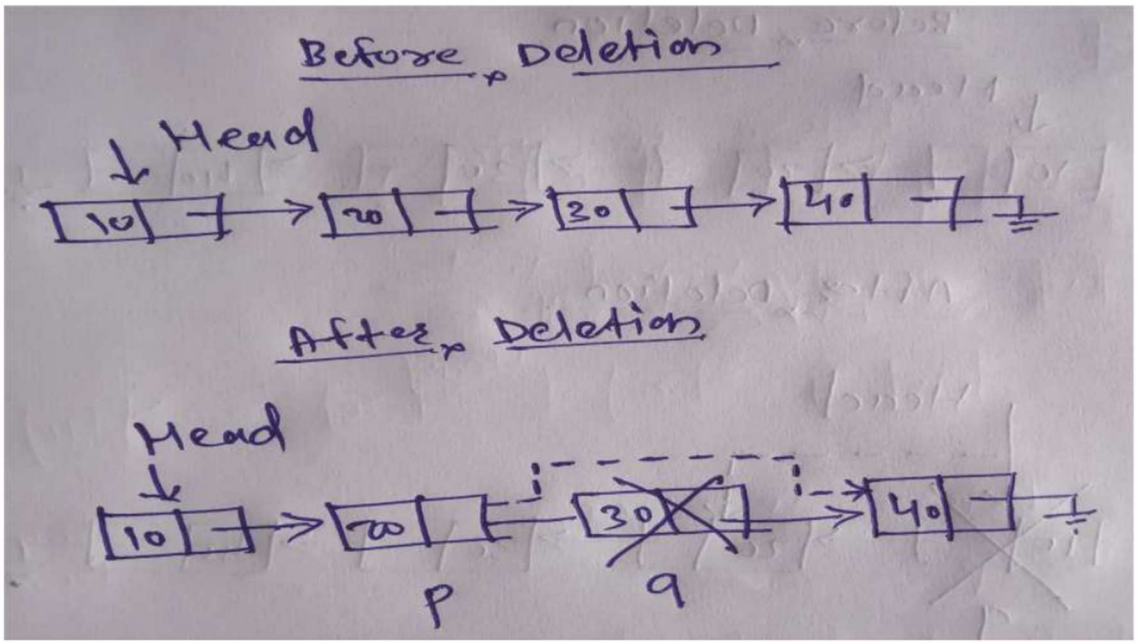
```
node* Insert_At_End(node *head)
{
   if(head==NULL)
   {
      printf("\nLinked list is Empty hence unable to insert node at end");
   }
   else
   {
      node *q,*p;
      int x;
      printf("\nEnter Data for insertion:");
      scanf("%d",&x);
      q=(node*)malloc(sizeof(node));
      q->data=x;
      q->next=NULL;
      p=head;
      while(p->next!=NULL)
      {
         p=p->next;
      }
      //insert node at end
      p->next=q;
      return(head);
   }
}
```

## 8. Write algorithm to delete an intermediate node from a Singly 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:



9. **Write an algorithm to insert a new node as the last of a singly linked list. Give example.**

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

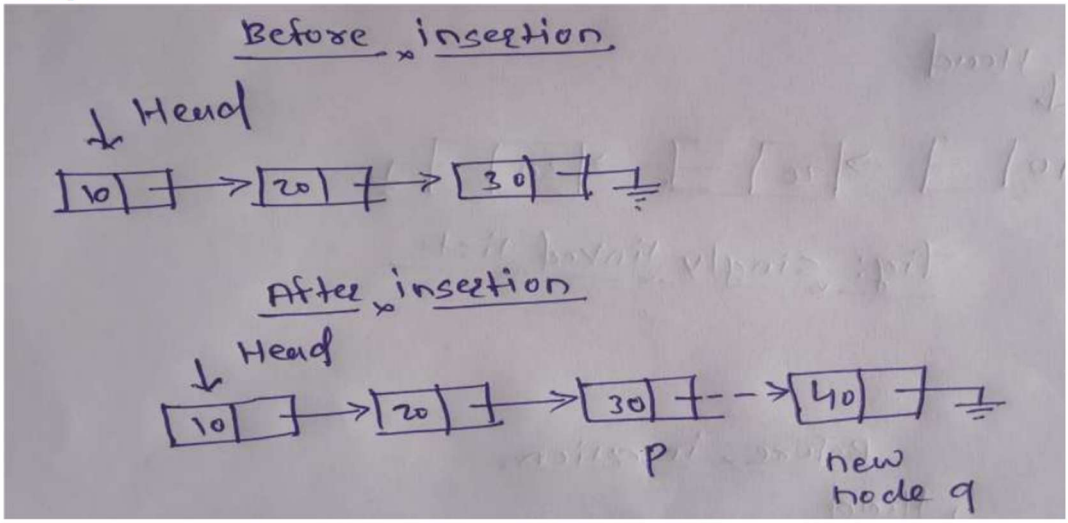
   print "Linked list is empty" and goto step 7
5. Otherwise initialize p with head and keep moving the p to its next node until it reaches to the last node in the list

   p=head

   while(p->next !=NULL)

   p=p->next;
6. Make link between last node and new node. p->next=q;
7. Stop

- Example:



10. **Write an algorithm to traverse 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
```

11. **Write an algorithm to count no of nodes in the 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
```

12. Create a Singly Linked List using data fields 10, 20, 30, 40, 50 and show procedure step-by step with the help of diagram from start to end.
13. Create a Singly Linked List using data fields 10, 20, 30, 40, 50. Search a node 40 from the SLL and show procedure step-by-step with the help of diagram from start to end.

14. **Describe advantage of circular linked list over linear linked list with example.**

**Advantages of Circular Linked Lists:**

- It is possible to traverse from the last node back to the first i.e. the head node.
- The starting node does not matter as we can traverse each and every node
- The previous node can be easily identified.
- No requirement for a NULL assignment in the code. The circular list never points to a NULL pointer unless fully deallocated.
- Circular linked list also performs all regular functions of a singly linked list.
- Circular linked lists are advantageous for end operations since beginning and end coincide.

15. **With example, describe how circular linked list works when a node is deleted from beginning of list**

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

- **Algorithm:**

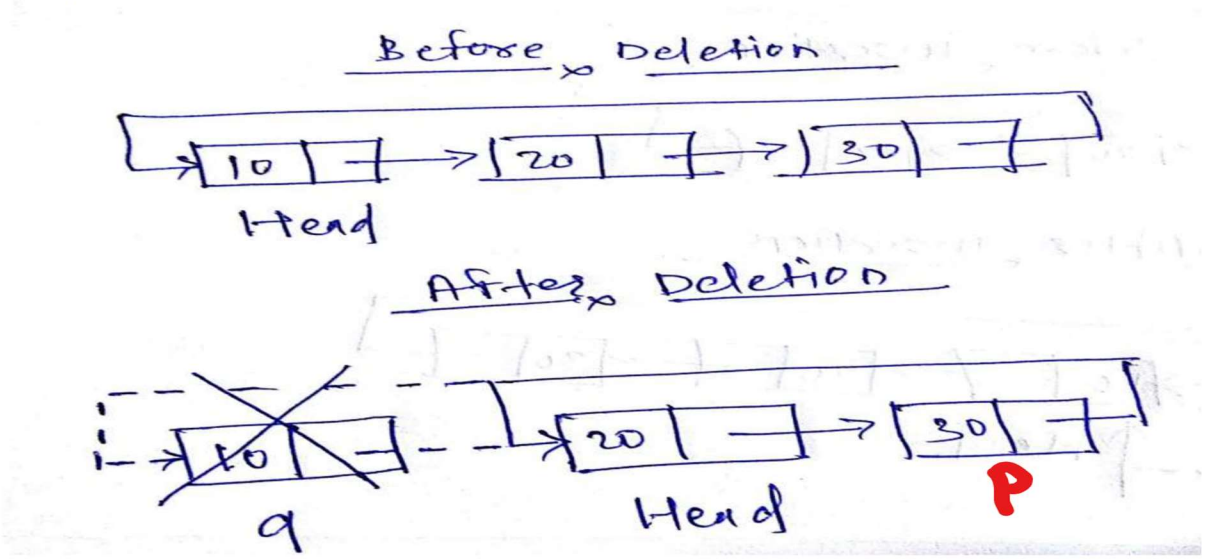
1. Start
2. Create temporary node pointer variable q & p.
3. Check whether linked list is Empty. i.e if (head = = NULL) then

   Display 'List is Empty,Deletion is not possible' and goto step 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!=head)
{
    p=p->next;
}
```

6. Set q=head, head=head->next, p->next=head and delete q node.
7. Stop

- Example:



## 16. 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. |

## 17. Define 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

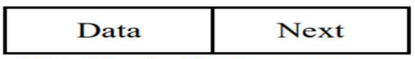



**Fig. Single Node structure**

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


# DATA STRUCTURE IMPORTANT QUESTIONS LIST WITH ANSWERS

**Author:**

*"Prof. Vishal Jadhav"*

*[BE in Computer Engineering and Having 8.5 years of IT industry experience.*
*VJTech Academy, Maharashtra Contact No: +91-9730087674 / 7743909870*
*Email id: vjtechacademy@gmail.com)]*

## ❖ UNIT-V: Trees and Graphs: Total Marks:16s

1. **Differentiate between tree and graph w.r.t. any 4 parameters.**

| Tree | Graph |
|---|---|
| Tree is a non-linear data structure | Graph is a non-linear data structure |
| Tree is a collection of nodes and edges | Graph is a collection of vertices and edges |
| Tree does not form any cycle. | Graph can form cycle. |
| Tree are always directed | Graph can be directed or undirected |
| We traverse a tree using in-order, pre-order, or post-order traversal methods. | For graph traversal, we use Breadth-First Search (BFS), and Depth-First Search (DFS). |
| If there are n nodes then there would be n-1 number of edges | Each node can have any number of edges. |

2. **Learn all basic terms of tree.**

**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:

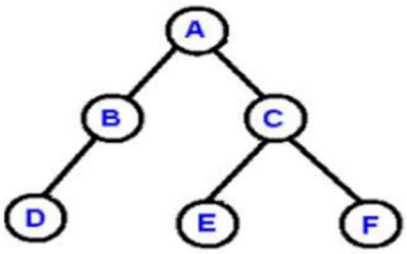



**Figure: Tree (a)**

   - **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:

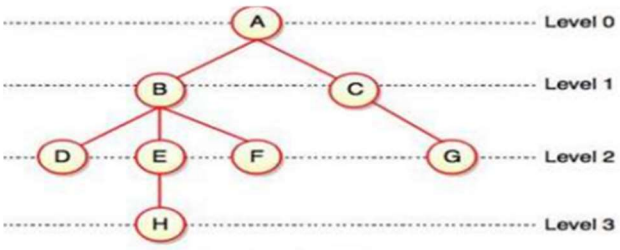



Fig. Levels of Tree

**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:

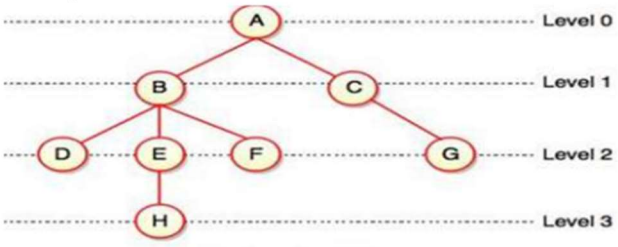



Fig. Levels of Tree

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

3. **Describe any two types of trees from the following: General tree, binary tree, and binary search tree.**

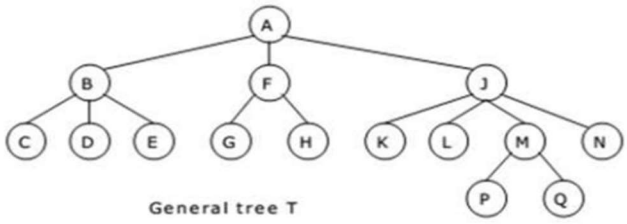
❖ **Types of Trees:**

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:



General tree T

**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

### 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:

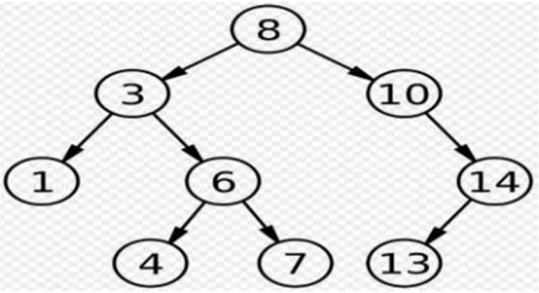
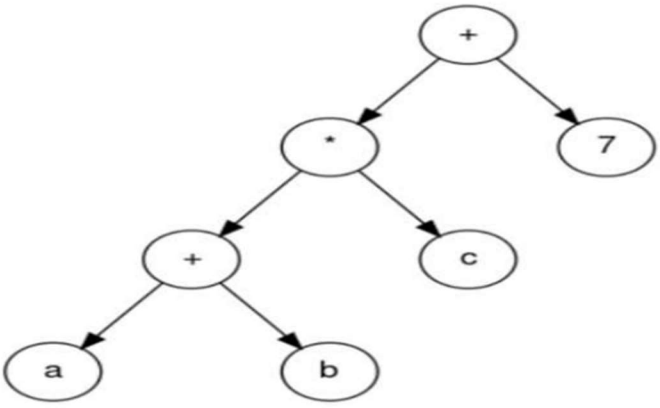



**Fig. Binary Search Tree**

### 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.

- Example of Expression Tree:




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

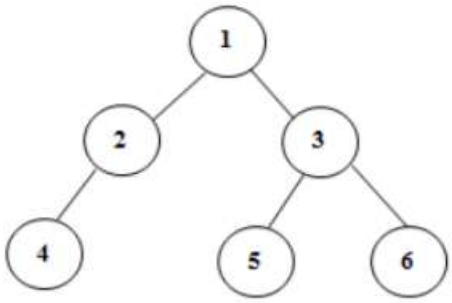
4. **Construct the binary search tree using following elements: 35,15,40,7,10,100,28,82,53,25,3. Show diagrammatically each step of construction of BST.**

5. **Construct Binary Search Tree: 1, 22, 27, 14, 31, 40, 43, 44, 10, 20, 35.**

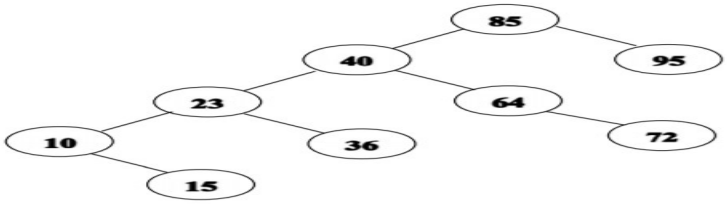
6. **Draw binary search tree:**

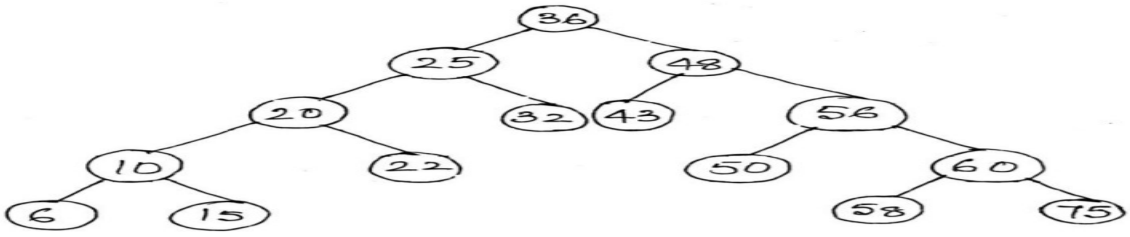
   35, 15, 40, 7, 10, 100, 28, 82, 53, 25, 3

7. From the given tree complete six answers:
   1. Degree of tree - 2
   2. Degree of node 3 - 2
   3. Level of node 5 - 2
   4. Indegree of node 3 - 1
   5. Outdegree of node 3 - 2
   6. Height of tree - 2

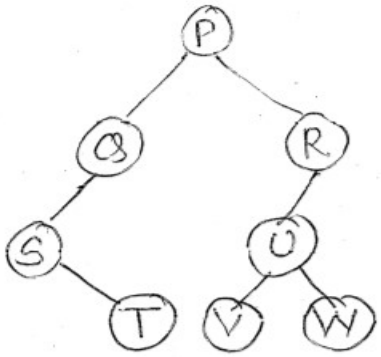


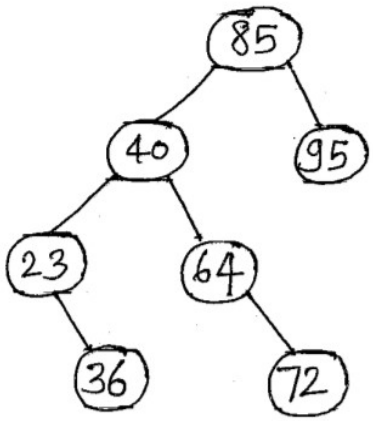

8. **Define the term tree traversal. Traverse the following tree in Inorder, Preorder and Postorder.**



9. **Define binary tree. Traverse the following tree in inorder, preorder and postorder.**

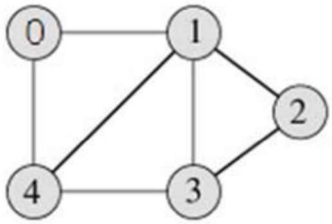


10. **Learn all basic terms of graph.**

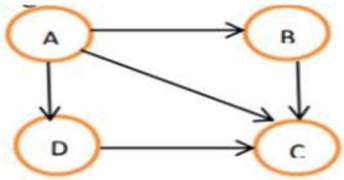
**Basic Terminology:**

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

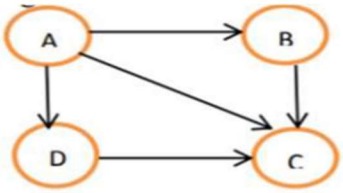



2. **Degree:**
   - The total number of edges linked/connected to the vertex is called as **degree**
   - **In-degree: -** No of total incoming edges to that particular vertex is known as In-degree.
   - **Out-degree: -** No of total outgoing edges from that particular vertex is known as Out-degree.
   - **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.

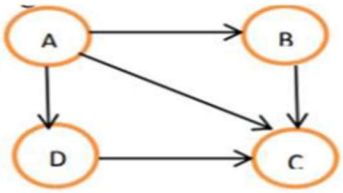
3. **Successor:** If there is a directed edge from vertex A to vertex B then vertex B is said to a successor of vertex A.




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.



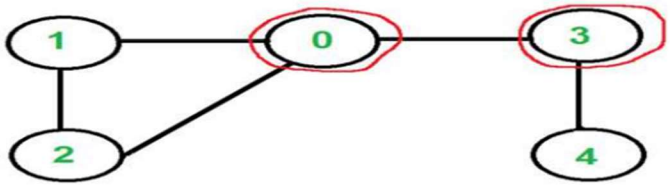

**5. Path:** The sequence of successive edges from source vertex to destination vertex is known as Path.




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.

11. **Give any two applications of Graph.**

❖ **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.

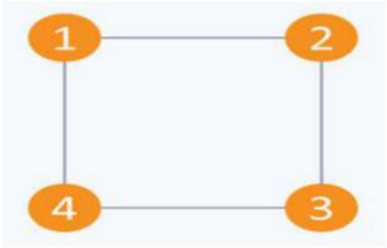
12. **Describe given two types of graphs: Directed and undirected graph.**

❖ **Types of Graph:**

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

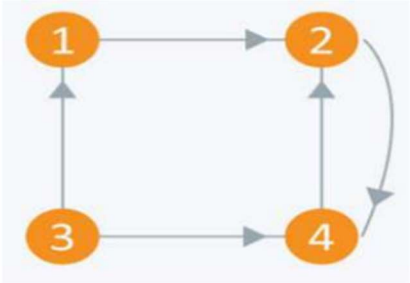
**1. Undirected Graph:**

- 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:



13. **For the following directed graph:**
    **i) Give adjacency matrix representation.**
    **ii) Give adjacency list representation.**

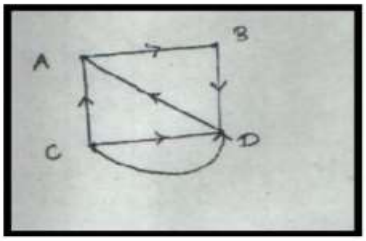



**Fig. A**

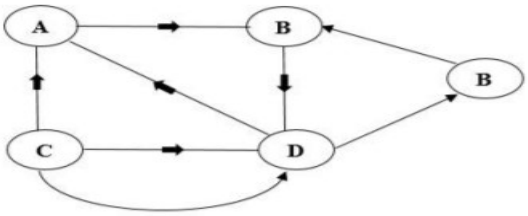



**Fig. B**

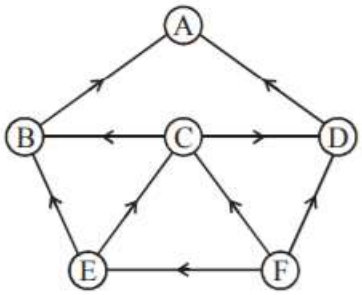
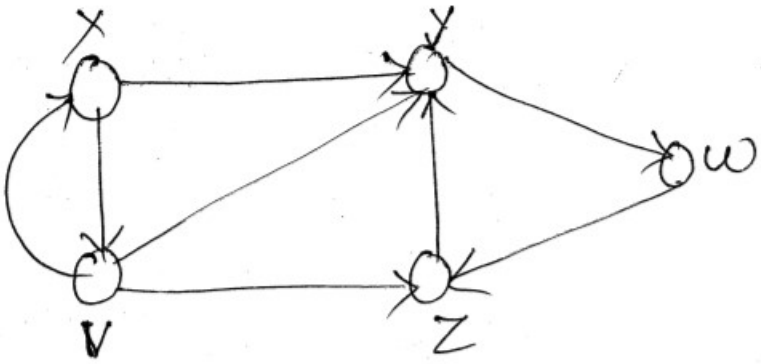
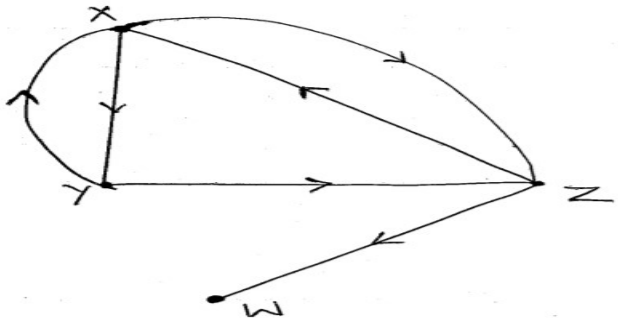



**Fig. C**

14. **Consider the graph 'G' in following figure :**
    **i) Find all simple path from X to Z.**
    **ii) Find indegree and outdegree of nodes Y and Z.**
    **iii) Find adjacency matrix A for the above graph.**
    **iv) Give adjacency list representation of above graph.**





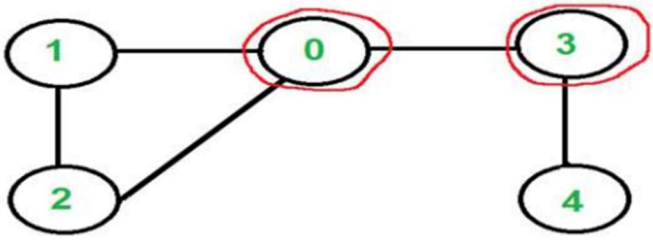

15. **Explain below terms:**

i) Source ii) Isolated node iii) Sink iv) Articulation point

- **Source Vertex: -** The vertex which has only outgoing edges and no incoming edges is known as Source Vertex.
- **Isolated Vertex: -** The vertex which has no incoming and no outgoing edges is known as Isolated Vertex.
- **Sink Vertex: -** The vertex which has only incoming edges and no outgoing edges is known as Sink vertex.

**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**