1. Data Structure(DS):

   A data structure is a way to represent the relationship among different available data items appearing in a group and along with set of operations which can be performed on that group so we can say

   Data Structure = Related Data + Allowed Operations on them.

   Frequent operations include storage, searching, insertion, deletion and sorting.

2. Applications of DS:

   Data structures are used in any program or software.

   They are used in the areas of

   - Compiler Design
   - Operating System
   - DBMS
   - Graphics
   - Simulation
   - Numerical Analysis
   - Artificial Intelligence

3. Advantages of a Linked list over an array:

   Insertion of an element.

   Deletion of an element.

   linked list has dynamic size whereas for array it is fixed size.

4. Syntax in C to create a node in the singly linked list:

```
struct node{

    int data;

    struct node *next;

};

struct node *head,*ptr;

ptr =(struct node *)malloc(sizeof(struct node));
```

5. Doubly linked list allows element two way traversal. On other hand doubly linked list can be used to implement stacks as well as heaps and binary trees. Singly linked list is

preferred when we need to save memory and searching is not required as pointer of single index is stored.

6. The difference between an Array and Stack:

   In an array, you have a list of elements and you can access any of them at any time. But in a stack, there's no random-access operation; there are only Push, Peek and Pop, all of which deal exclusively with the element on the top of the stack. A stack is data-structure which has a last in first out policy.

7. The minimum number of Queues needed to implement the priority queue:

   Two Queues are needed. One stores data elements and another stores priorities.

8. The different types of traversal techniques in a tree:

   Pre order traversal.(NLR)

   In order traversal.(LNR)

   Post order traversal.(LRN)

9. It is said that searching a node in a binary search tree is efficient than that of a simple binary tree because

   Binary Search Tree allows for fast retrieval of elements stored in the tree as each node key is thoroughly compared with the root node, which discards half of the tree.

10. The applications of Graph DS:
    a. Graph theory is used to study molecules in chemistry and physics.
    b. Graph data structures are used in building a networks. In providing LAN connections between nodes and also in VPN. Specially spanning tree is used in these networks.
    c. Recommendations on e-commerce websites: The “Recommendations for you” section on various e-commerce websites uses graph theory to recommend items of similar type to user’s choice
11. Can we apply Binary search algorithm to a sorted Linked list?

    Yes, Binary search is possible on the linked list if the list is ordered and you know the count of elements in list. But While sorting the list, you can access a single element at a time through a pointer to that node i.e. either a previous node or next node.

12. When can you tell that a Memory Leak will occur?

Memory leak occurs when programmers create a memory in heap and forget to delete it.
Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.

To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

13. To see if a binary tree is a binary search tree, check:

    If a node is a left child, then its key and the keys of the nodes in its right subtree are less than its parent's key.

    If a node is a right child, then its key and the keys of the nodes in its left subtree are greater than its parent's key.

14. Which data structure is ideal to perform recursion operation and why?

    Stack. Because of its LIFO (Last In First Out) property it remembers its 'caller' so knows whom to return when the function has to return. Recursion makes use of system stack for storing the return addresses of the function calls.
    Every recursive function has its equivalent iterative (non-recursive) function.

15. What are some of the most important applications of a Stack?

    Expression Handling − Infix to Postfix or Infix to Prefix Conversion − ...
    Backtracking Procedure − Backtracking is one of the algorithm designing technique. ...
    Another great use of stack is during the function call and return process.

16. Consider the below expression: a + b / c - d

    At first, compiler scans the expression to evaluate the expression b / c, then scans the expression again to add a to it. The result is then subtracted with d after another scan.

    This repeated scanning makes the process very inefficient and time consuming. It will be much easier if the expression is converted to prefix (or postfix) before evaluation.

    The corresponding expression in prefix form is: -+a/bcd. The prefix expressions can be easily evaluated using a stack.

17. Sorting a stack using a temporary stack:

    Create a temporary stack say tempStack.

While input stack is NOT empty do this:

Pop an element from input stack call it temp

while temporary stack is NOT empty and top of temporary stack is greater than temp, pop from temporary stack and push it to input stack.

push temp in temporary stack.

The sorted numbers are in tempStack.

18. Program to reverse a queue

```
public class Queue_reverse {

   static Queue<Integer> queue;

   // Utility function to print the queue
   static void Print()
   {
      while (!queue.isEmpty()) {
         System.out.print( queue.peek() + ", ");
         queue.remove();
      }
   }

   // Function to reverse the queue
   static void reversequeue()
   {
      Stack<Integer> stack = new Stack<>();
      while (!queue.isEmpty()) {
         stack.add(queue.peek());
         queue.remove();
      }
      while (!stack.isEmpty()) {
         queue.add(stack.peek());
         stack.pop();
      }
   }


   public static void main(String args[])
   {
```

```
        queue = new LinkedList<Integer>();
                for (int i = 0; i<10; i++)
                {
                        queue.add(i);

         }
         reversequeue();
       }
    }
```

19. Program to reverse first k elements of a queue:

```
import java.util.*;
public class que {

        static void reverseQueueFirstKElements(int k,
Queue<Integer> queue)
   {
      if (queue.isEmpty() == true
         || k > queue.size())
         return;
      if (k <= 0)
         return;

      Stack<Integer> stack = new Stack<Integer>();

      for (int i = 0; i < k; i++) {
         stack.push(queue.peek());
         queue.remove();
      }

      // Enqueue the contents of stack
      // at the back of the queue
      while (!stack.empty()) {
         queue.add(stack.peek());
         stack.pop();
      }

      for (int i = 0; i < queue.size() - k; i++) {
         queue.add(queue.peek());
         queue.remove();
      }
   }

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                 Queue<Integer> queue = new LinkedList<Integer>();
```

```
                queue.add(10);
                queue.add(20);
                queue.add(30);
                queue.add(40);
                queue.add(50);
        queue.add(60);
        queue.add(70);
                reverseQueueFirstKElements(3, queue);
                for(Integer q : queue)
                {
                        System.out.println(q);
                }


        }

    }
```

20. Program to return the nth node from the end in a linked list

```
class Node {
   int data;
   Node next;
   Node(int d)
   {
      data = d;
      next = null;
   }
}


void printNthFromLast(int n)
{
   int len = 0;
   Node temp = head;


   while (temp != null) {
      temp = temp.next;
      len++;
   }

   // check if value of n is not more than length of
   // the linked list
   if (len < n)
      return;

   temp = head;
```

```
        // 2) get the (len-n+1)th node from the beginning
        for (int i = 1; i < len - n + 1; i++)
            temp = temp.next;

        System.out.println(temp.data);
    }

    /* Inserts a new Node at front of the list. */
    public void push(int new_data)
    {
        /* 1 & 2: Allocate the Node &
                  Put in the data*/
        Node new_node = new Node(new_data);

        /* 3. Make next of new Node as head */
        new_node.next = head;

        /* 4. Move the head to point to new Node */
        head = new_node;
    }

    public static void main(String[] args)
    {
        LinkedList llist = new LinkedList();
        llist.push(20);
        llist.push(4);
        llist.push(15);
        llist.push(35);

        llist.printNthFromLast(4);
    }
}
```

21. Reverse a linked list

```
class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d)
        {
```

```
        data = d;
        next = null;
    }
  }

  /* Function to reverse the linked list */
  Node reverse(Node node)
  {
    Node prev = null;
    Node current = node;
    Node next = null;
    while (current != null) {
      next = current.next;
      current.next = prev;
      prev = current;
      current = next;
    }
    node = prev;
    return node;
  }

  // prints content of double linked list
  void printList(Node node)
  {
    while (node != null) {
      System.out.print(node.data + " ");
      node = node.next;
    }
  }

  public static void main(String[] args)
  {
    LinkedList list = new LinkedList();
    list.head = new Node(85);
    list.head.next = new Node(15);
    list.head.next.next = new Node(4);
    list.head.next.next.next = new Node(20);

    System.out.println("Given Linked list");
    list.printList(head);
    head = list.reverse(head);
    System.out.println("");
    System.out.println("Reversed linked list ");
    list.printList(head);
  }
}
```

22. Replace each element of the array by its rank in the array

```
import java.util.*;

class GFG {

    // Function to assign rank to
    // array elements
    static void changeArr(int[] input)
    {
        // Copy input array into newArray
int newArray[] = Arrays.copyOfRange(input, 0, input.length);

        // Sort newArray[] in ascending order
        Arrays.sort(newArray);
        int i;

        // Map to store the rank of
        // the array element
        Map<Integer, Integer> ranks
            = new HashMap<>();

        int rank = 1;

        for (int index = 0;
             index < newArray.length;
             index++) {

            int element = newArray[index];

            // Update rank of element
            if (ranks.get(element) == null) {

                ranks.put(element, rank);
                rank++;
            }
        }

        // Assign ranks to elements
        for (int index = 0;
             index < input.length;
             index++) {

            int element = input[index];
            input[index]
                = ranks.get(input[index]);
```

```
        }
    }

    public static void main(String[] args)
    {
        // Given array arr[]
        int[] arr = { 100, 2, 70, 2 };

        // Function Call
        changeArr(arr);

        // Print the array elements

            System.out.println(Arrays.toString(arr));
    }
}
```

23. Check if a given graph is a tree or not

An undirected graph is tree if it has following properties.
1) There is no cycle.
2) The graph is connected.

For an undirected graph we can either use BFS or DFS to detect above two properties.

24. Find out the Kth smallest element in an unsorted array

```
public int kthElement(int[] array , int k)

{   Arrays.sort(array);

    return array[k-1];

 }
```

25. How to find the shortest path between two vertices?

Algorithm:

- Input the graph.
- Input the source and destination nodes.
- Find the paths between the source and the destination nodes.
- Find the number of edges in all the paths and return the path having the minimum number of edges.