Name: Pavanikumari

Roll no: 320022720007

1st semester

Sub: DS and Algorithms

1.) Define minimal spanning tree and explain with an example?

A) Minimal spanning tree: A minimum spanning tree is a special kind of tree that minimizes the lengths (or "weights") of the edges of the tree. An example is a cable company wanting to lay line to multiple neighborhoods; by minimizing the amount of cable laid, the cable company will save money. A tree has one path joins any two vertices. A spanning tree of a graph is a tree that: · contains all the original graph's vertices. · Reaches out to (spans) all vertices. · Is acyclic. In other words, the graph doesn't have any nodes which loop back to itself.

We found three spanning trees off one complete graph. A complete undirected graph can have maximum nn-2 number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3, hence 33-2 = 3 spanning trees are possible.

General Properties of Spanning Tree:

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G -

· A connected graph G can have more than one spanning tree.

· All possible spanning trees of graph G, have the same number of edges and vertices.

- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

2.) Explain quick sort with an example?

QUICK SORT: Quick Sort is a sorting algorithm, which is commonly used in computer science. Quick Sort is a divide and conquer algorithm. It creates two empty arrays to hold elements less than the pivot value and elements greater than the pivot value, and then recursively sort the sub arrays. There are two basic operations in the algorithm, swapping items in place and partitioning a section of the array.

Important characteristics of Quick Sort:

- Quick Sort is useful for sorting arrays.
- In efficient implementations Quick Sort is not a stable sort, meaning that the relative order of equal sort items is not preserved.
- Overall time complexity of Quick Sort is O(nLogn). In the worst case, it makes O(n2) comparisons, though this behavior is rare.
- The space complexity of Quick Sort is O(nLogn). It is an in-place sort (i.e. it doesn't require any extra storage)

Example:

```
#include<studio.h>
```

```
#include<conio.h>
void quicksort(int array[], int firstIndex, int lastIndex)
{
int pivotIndex, temp, index1, index2; if(firstIndex < lastIndex)
{
pivotIndex = firstIndex;
 index1 = firstIndex;
 index2 = lastIndex;
while(index1 < index2)
 {
while(array[index1] <= array[pivotIndex] && index1 < lastIndex)
{ ndex1++;  while(array[index2]>array[pivotIndex])
{ ndex2--;  }
 if(index1<index2)
{ etmp = array[index1];
 array[index1] = array[index2]; array[index2] = temp;  } }
temp = array[pivotIndex];
 array[pivotIndex] = array[index2];
 array[index2] = temp; quicksort(array, firstIndex, index2-1);
quicksort(array, index2+1, lastIndex);  } }
int main() {
 int array[100],n,i: printf("Enter the number of element you want to sor
t ");:
 scanf("%d",&n);
 printf("Enter Elements in the list  :
```

");
for(i = 0; i < n; i++) {
scanf("%d",&array[i]); }
quicksort(array,0,n-1); printf("Sorted elements: "); for(i=0;i<n;i++){
printf(" %d",array[i]); getch(); return 0;
}
Output: Sorted elements: 4 6 8 10 30

3.) Define double linked list, explain implantation of stacks using linked list?
Double linked List: Doubly linked list is a type of linked list in which each node apart from storing its data has two links. The first link points to the previous node in the list and the second link points to the next node in the list. The first node of the list has its previous link pointing to NULL similarly the last node of the list has its next node pointing to NULL
Implementation of double link list: ·
add_front: Adds a new node in the beginning of list
· add_after: Adds a new node after another node · add_before: Adds a new node before another node · add_end: Adds a new node in the end of list · delete: Removes the node ·
forward_traverse: Traverse the list in forward direction ·
backward_traverse: Traverse the list in backward direction Operations of Double linked list: In a double linked list, we perform the following operations... 1. Insertion 2. Deletion 3. Display 1.) Insertion: In a double linked list, the insertion operation can be performed in three ways as

follows... 1. Inserting At Beginning of the list 2. Inserting At End of the list 3. Inserting At Specific location in the list. 2.) Deletion: In a double linked list, the deletion operation can be performed in three ways as follows...

1. Deleting from Beginning of the list

2. Deleting from End of the list 3. Deleting a Specific Node

3.) Displaying: we can use the following steps to display the elements of a double linked list... · Step 1 - Check whether list is Empty (head == NULL) · Step 2 - If it is Empty, then display 'List is Empty!!!' and terminate the function.

· Step 3 - If it is not Empty, then define a Node pointer 'temp' and initialize with head. · Step 4 - Display 'NULL <--- '. · Step 5 - Keep displaying temp → data with an arrow (<====>) until temp reaches to the last node · Step 6 - Finally, display temp → data with arrow pointing to NULL (temp → data ---> NULL).

```
public class DoublyLinkedList
{
Node head; class Node {
int data; Node prev;
Node next; Node(int d) {
data = d; } }
public void insertAtFront(int data) {
Node newNode = new Node(data);
newNode.next = head; newNode.prev = null; if (head != null)
head.prev = newNode; head = newNode;
```

```
}
 public void insertAfter(Node prev_node, int data) { if(prev_node ==
null) {
System.out.println("previous node cannot be null"); return;  }
Node new_node = new Node(data); new_node.next = prev_node.next;
prev_node.next = new_node; new_node.prev = prev_node;
 if (new_node.next != null) new_node.next.prev = new_node;  }
 void insertEnd(int data) {
Node new_node = new Node; Node temp = head; new_node.next = null; if
(head == null) {
new_node.prev = null; head = new_node; return;  }while (temp.next !=
null)
temp = temp.next; temp.next = new_node;  //Assign prev of newNode to
temp new_node.prev = temp;
} void deleteNode(Node del_node{ if(head == null || del_node == null)
{
return;  }
// if del_node is the head node, point the head pointer to the next of
del_node
if (head == del_node) {
 head = del_node.next;
if (del_node.next != null) {
del_node.next.prev = del_node.prev;
if (del_node.prev != null) {
del_node.prev.next = del_node.next;  }
```

```
}
 public void printlist(Node node) {
 Node last = null; while (node != null) {
System.out.print(node.data + "->"); last = node; node = node.next;  }
System.out.println();  }public static void main(String[] args)
{DoublyLinkedList doubly_ll = new DoublyLinkedList();
doubly_ll.insertEnd(5); doubly_ll.insertFront(1); doubly_ll.insertFront(6);
doubly_ll.insertEnd(9); doubly_ll.insertAfter(doubly_ll.head, 11);
doubly_ll.insertAfter(doubly_ll.head.next, 11);
doubly_ll.printlist(doubly_ll.head);
doubly_ll.deleteNode(doubly_ll.head.next.next.next.next.next);
doubly_ll.printlist(doubly_ll.head);  }  }
```