DSA Assignment-2

Problem 1: In a N-ary land, people really like the number ‘n’. You are their chief algorithms expert and you are given the task of designing an n-ary heap for these people. An n-ary heap is like a binary heap, but (with one possible exception) non-leaf nodes have ‘n’ children instead of 2 children. You have to design an n-ary heap and answer the following questions:

a) How can we represent an n-ary heap in an array?
b) Let’s say we have an n-ary heap which contains ‘d’ number of elements in total. Give the height of this heap in terms of ‘n’ and ‘d’.
c) Give an efficient implementation of EXTRACT-MAX and INSERT in an n-ary Max heap. Analyse the running times of both of the functions in terms of ‘d’ (total size) and n.
d) Give an efficient implementation of DECREASE-KEY(A, i, k) which will give an error if k > A[i], but otherwise sets A[i] = k and then updates the n-ary max-heap structure appropriately. Analyse its running time in terms of ‘d’ (size) and n.

Assume array based implementation of the heap (for parts b, c and d) and array is always sufficiently large to accommodate all elements, if total elements in heap are ‘d’ then array[d+1] can be assumed to be 0 or -1. Also please assume elements to be stored in heap to be positive.

Solution: (2.5 + 2.5 + 5 + 5) for full marks correct complexity should also be present with pseudo-code.

a) We just need to modify left, right and parent. We can get kth child of the ith node with **n*(i-1) + k+1** and the parent of ith node with **ceil((i-1)/n)** (when indexing is 1-based). If we have 0- based indexing then kth child of node i, will be given by **n*i + k** and parent of i will be **floor((i-1)/n)**.
b) It would be $\log_n(nd - d + 1)$. (derivation will be required)
c) Time complexity of **extract-max** is $(n*\log_n d)$

```
Int Extract-Max(int heap[]){
        Max -> heap[0]
        Heap[0] = heap[d-1]    // 0-based indexing
        Heap[d-1] = -1
        heapify(heap, 0)
        Return max;
}

heapify(int heap[], int i){
        Largest = i
        for(int k = 1; k<=n; k++){
                Child = n*i + k
                if(child <d && heap[child] > heap[largest]) {largest = child;}
        }
        if(largest != i){
```

```
                    Temp = heap[i];
                    Heap[i] = largest;
                    Heap[largest] = temp;
                    heapify(heap, largest);
                }
        }
```

Time complexity of **INSERT** would be $\log_n d$.

```
        insert(int heap[], int key){
                d++;
                Heap[d-1] = key;
                Current = d-1;
                while(current>0 && heap[current/n] <heap[current] )
                        swap(heap, current, current/n)
                        Current = current/n;

        }
```

d) Time Complexity of Decrease-Key will also be $(n*\log_n d)$

```
        Decrease-Key(int heap[], int i, int new_val){
                if( heap[i] < new_val) { //Error }
                Heap[i] = new_val;
                heapify(heap, i)
        }
```

Problem 2: Dumbledore's Luncheon (Stacks)

Dumbledore is hosting a luncheon at Hogwarts, and Ron is incharge of the sitting plans. Dumledore's dining table can be thought of as an infinite row of seats with Dumbledore at one end. Each of Dumbledore's guests represents a character of English alphabet (a to z). Dumbledore wants his guests to be seated such that the resulting string formed from the alphabets should be lexicographically smallest (See examples for reference). He didn't tell Ron the total number of guests coming, but he will ask Ron to close the gate once all the guests arrive (marking the end of incoming guests).

The rules of seating are as follows:

- Seats are allotted in sequential order, a guest who arrives later must be further from Dumbledore than a guest who arrives earlier.
- An incoming guest must be assigned a seat, as soon as he arrives. He will take a seat.
- Once a guest takes their seat, you can't ask them to move to some other seat. Even if the seat in front of him is empty, they will stay at their occupied seat only.
- But you can make a guest disappear using your spells, Dumbledore has allowed you to make at most 'k' of his guests disappear. He can handle 'k' guests not in the party but you can't remove more than 'k' guests.

- No seat should remain empty between any two guests or between guests and Dumbledore. Since the table has infinite seats, a series of empty seats would be present in the end.

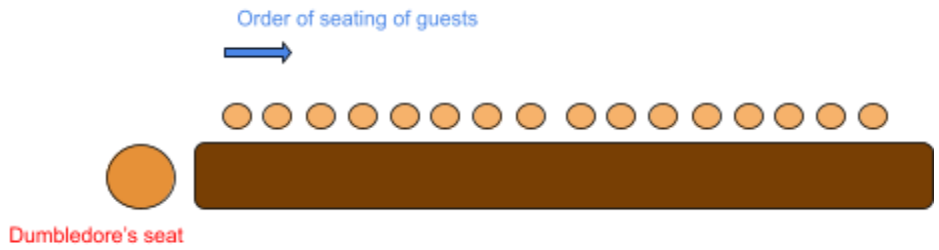
You are Ron, design a code that your magic wand will follow to keep track of which guest to assign which seat.

Assumptions:

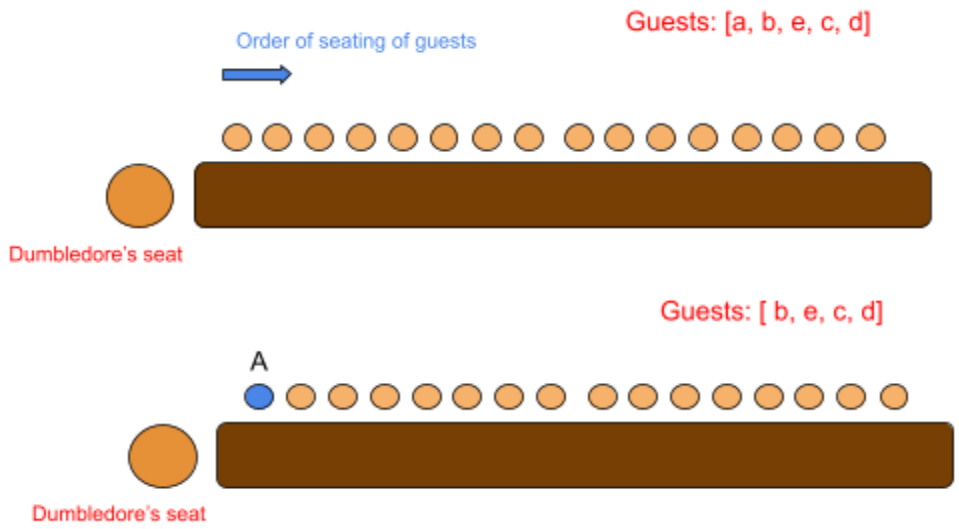
> You may assume that you have enough time when a guest comes so that you are able to make some existing guest disappear before assigning the new guest his seat.

> When Dumbledore asks you to close the gate, you can still make some guests disappear. Dumbledore will have a look at your arrangement after you close the gate, it should be lexicographically smallest by then.

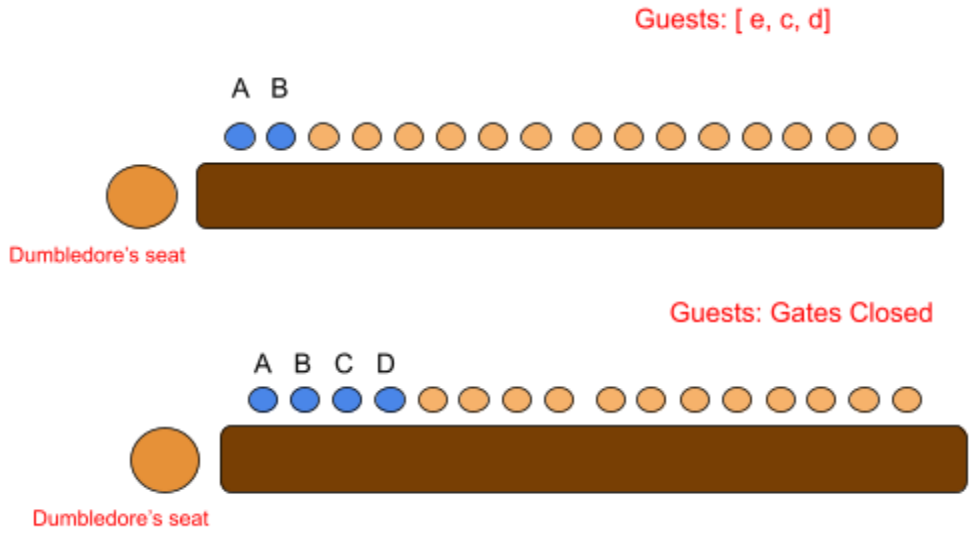
Marking Scheme: Total 15 points (5 + 5 + 5) (Algorithm explanation + Pseudo-code + runtime and data structure used)



Let's say incoming guests are [a, b, e, c, d] and K = 1.



We can make [a, b] sit and make ‘e’ disappear. [a, b, c, d] is the lexicographically smallest sequence possible.



Solution: Use a stack to maintain guests.
Full marks: Writing an O(n) solution using stacks or any other relevant data structure that gives the correct sequence as answer.
Partial marks: Instead of stack which allows pop() of last element in O(1), array or any similar structure is used such that worst case complexity is O(n) and overall complexity is $O(n^2)$ then 50% marks should be given provided answer is correct in terms sequence generated in the end.

5 marks are to be awarded if only the stack is mentioned as data structure and time complexity is stated as O(n) but the algorithm is absent or is incorrect.

Note that, if you ever try pop() from beginning, (stack.size()>1) it should be given 0 marks because in the question it was stated that you can’t change a guest’s seat once he is placed on the table and if you pop() from the beginning you are making each guest shift one seat to the front.

Pseudo Code:

```
placeGuests( int k){
        Stack<Guest> stack ;
        while(Incoming_Guests){
                New_guest -> new Guest;
                if(stack.isEmpty() || stack.peek().character <=  New_guest.character)
                        New_guest -> seated in the first empty seat
                        stack.push(New_guest);
                Else
                        while(!stack.isEmpty() && stack.peek().character > New_guest.character
&& k>0)
```

```
                        stack.peek() -> guest disappear
                        stack.pop()
                        K—
                stack.push(new_Guest)
                New_guest -> seated in the first empty seat
        }
        //Now Gates are closed
        while(stack.size()>0 && k>0){
                stack.peek() -> guest disappear
                stack.pop();
                k--;
              }
        // Now the stack contains the guest in the other they should be seated

}
```
(While placing each guest we want to place them as close to start as possible, we will keep disappearing guests if guest>current_guest)

Problem 3: In the country of WeirdLand, there is only one barber shop. In the shop there are 3 barbers, Alpha, Beta and Gamma, who are not always efficient as employees. All the customers are seated in a waiting area, each customer has a card which has a unique integral ID. For calling next person from the waiting area each barber has his own weird way:

Alpha: He will call either the person with least ID from the waiting area or the one with max ID. Min or Max is decided randomly by him (assume 50% probability for each min or max).
Beta: He will choose a ‘k’ and will call the kth smallest ID, (Kth ID when all IDs are arranged in ascending order). K is chosen randomly from the number of persons available in waiting area (1 <= K <= Total persons)
Gamma: He will always choose the median ID. If there are ‘n’ people in the waiting area, the median will be defined as (n+1)/2 th ID when all of them are arranged in ascending order. (For both odd and even ‘n’)

The Government has decided to digitise this shop and is asking for your suggestions on the data structure which should be ideal for this scenario.
Your task is to design a data structure which should support the query of each of the three barbers. The data structure should be highly efficient. You should properly explain which data structure you will use, what information will you store (and how), insertion and deletion of the IDs, query by each barber, and also discuss the time complexity of every operation (Insertion, deletion, and each query) (Write recurrence/ loop invariants and analyse worst case complexities).
Marking Scheme: Total 15 marks (10 + 5) (10 points for efficient implementation of all the required functions like, Insertion, Deletion, Queries and other Miscellaneous functions along

with their time complexities + 5 points for correct data structure which will help to execute all the queries efficiently)

Consider following Assumptions:
Each person will have a unique ID, IDs will be integral and can be represented using upto 10 digit integers.
The barbers will query sequentially, assuming no clash between two barbers choosing the same ID.
Once an ID is summoned by any barber, it should be removed from the waiting area.
Please come up with the most efficient solution using the data structures you have studied in class. More efficient solutions will fetch better marks. Avoid use of Inbuilt data structures.

Solution:

Marking: (All complexities should be worst case, not average case)
For Full marks: Each query should be executed in at most O(logN). Correct explanation for the time complexity analysis and correct data structure should be used. Correct data structure refers to data structures which allow each of these operations in O(logN) in the worst case.

For 50% marks: If at most one Query from Insert, delete, Alpha, beta, Gamma is in O(n) and all rest are executed in < O(n).

If only correct data structure (AVL Tree) is mentioned and other necessary functions are not implemented or queries not implemented efficiently, then 5 marks will be awarded.

Note: If someone tries to create an array of **size 10^10**, the solution should be given **0 marks** because creating arrays of this size isn't possible in JVM or any other language using current computational capabilities.
Same goes for solutions using Segment trees, Fenwick trees or BIT trees, they will be awarded 0 marks.

Solution and Explanation:
We can use a modified AVL tree which also stores the total number of elements in the subtree for each node. All the queries will have a complexity of O(logN) where 'n' is the number of IDs present.
For INSERT and DELETE operations: we can use insert and delete operations. Complexity O(logN) here N is the number of customers currently present.
For queries:

Alpha: It will use minnode and maxnode (both have complexities O(logN)) as per his requirement and then delete the popped customer from the Tree (Using delete() complexity O(logN). Total time complexity = O(logN)
Beta: It will use find_kth(), it also has a complexity of (O(logN)) and then delete(). Total time complexity = O(logN).
Gamma: It will first calculate the total number of customers using getSize(root) (Time complexity: O(1)) and then calculate the median index and then use find_kth() and then delete to achieve desired operation. Total time complexity = O(logN).

Pseudo codes for required functions:

```
Class node{
        Int id/data;
        Int height;
        Int size;
        Node left,right;
}
```

```
Void getSize(node root){
        If root -> null: return 0;
        Else return root.size;
}

// Please note that recursion should not used otherwise complexity would become O(n) and it should fetch 0 marks; (In this function)
```

```
Void getHeight(node root){
        If root->null: return 0;
        Return root.height;
}

Void getbalance(root){
        If root -> null: return 0;
        Else return getheight(root.left) - getHeight(root.right);
}
// Please note that recursion should not used otherwise complexity would become O(n) and it should fetch 0 marks; (In these two functions)
```

```
Void insert(node root, int val){
                if(root == null) {return new node(val);}
```

```
                if(root.data == val) {return root;}
                else if(root.data > val) {root.left = insert(root.left, val);}
                else {root.right = insert(root.right , val);}

                root.height = 1 + Math.max(getheight(root.left), getheight(root.right));
                root.size = 1 + getsize(root.left) + getsize(root.right);
                balance();
                Return root;
}

Void delete(node root, int val){
                if(root == null) {return root;}
                if(val < root.data){
                        root.left = delete(root.left, val);
                }
                else if(val > root.data){
                        root.right = delete(root.right, val);
                }
                else{

                        if(root.left == null && root.right==null){
                                root = null;
                                return root;
                        }
                        else if( root.left == null){
                                root = root.right;
                        }
                        else if (root.right == null){
                                root = root.left;
                        }
                        else{
                                node temp = minnode(root.right);

                                root.data = temp.data;
                                root.right = delete(root.right, temp.data);



                        }
                }
                root.height = 1 + Math.max(getheight(root.left), getheight(root.right));
                root.size = 1 + getsize(root.left) + getsize(root.right);
```

```
                balance();
}
```

```
 node minnode(node root)
        {
                if(root == null || root.left==null){return root;}
                return minnode(root.left);

        }
node maxnode(node root){
                if(root == null || root.right == null){return root;}
                return maxnode(root.right);
        }
```

```
node find_kth(node root, int k){
                if(root == null){
                        return root;
                }
                if(k <= getsize(root.left)){
                        return find_kth(root.left, k);
                }
                else if(k > getsize(root.left)+1){
                        return find_kth(root.right, k - getsize(root.left) -1);
                }
                return root;
        }
```

```
Node rightRotate(root){
                node root_left = root.left;
                node root_left_right = root_left.right;

                root_left.right = root;
                root.left = root_left_right;
                root.height = 1+Math.max(getheight(root.left), getheight(root.right)) ;
                root_left.height = 1 + Math.max(getheight(root_left.left),
getheight(root_left.right)) ;
                root.size = 1 + getsize(root.left) + getsize(root.right);
                root_left.size = 1 + getsize(root_left.left) + getsize(root_left.right);
                return root_left;
```

```
}
Node LeftRotate(root){
                node root_right = root.right;
                node root_right_left = root_right.left;

                root_right.left = root;
                root.right = root_right_left;

                root.height = 1 + Math.max(getheight(root.left), getheight(root.right));
                root.size = 1 + getsize(root.left) + getsize(root.right);

                root_right.height = 1 + Math.max(getheight(root_right.left),
getheight(root_right.right));
                root_right.size = 1 + getsize(root_right.left) + getsize(root_right.right);

                return root_right;
}
```

A Java based implementation of the same can be found on this link.

Problem 4 Binary Tree Problem:

There is a binary tree given to you. In which each of the node represents the student in the class. You are playing a game in which you need to team up with one of the students in class. Team can be formed in such a way that in a team both of the students can't have same parent in tree and should be at the same level in binary tree.

Input: we have given you some pair of students you need to find that is the team valid or not.

Output: 'Yes' if the team is valid or 'No' if the team is invalid.

(a) Optimal solution of O(n) for checking the team validity.

(b) How many tree traversal is required for the solution of the above problem.

Marking Scheme: Total 15 marks-> 10 for part (a)(Algorithm explanation + Pseudo-code + runtime and data structure used)+ 5 for part (b)(explanation is required)

Solution:

Ans (a)

Method:

(1.) Check for the level of the students first.

(2.) Then check if they have same parents.

Pseudo Code:

```
class Node
{
        int data;
       Node left, right;
        Node(int item)
           {
               data = item;
               left = right = null;
           }
}

class BinaryTree
{
    Node root;
//if both the student can't be in a team.
   boolean nonTeam(Node node, Node a, Node b)
    {
        // Base case
        if (node == null)
            return false;


        return ((node.left == a && node.right == b) || (node.left == b && node.right
== a) || nonTeam(node.left, a, b) ||nonTeam(node.right, a, b));

    }

 //return the level of the node in tree
    int level(Node node, Node ptr, int lev)
    {
        // base cases
        if (node == null)
            return 0;
        if (node == ptr)
            return lev;

        // Return level if Node is present in left subtree
```

```
        int l = level(node.left, ptr, lev + 1);
        if (l != 0)
            return l;

        // Else search in right subtree
        return level(node.right, ptr, lev + 1);
    }
 //if the team is possible or not
  boolean isTeam(Node node, Node a, Node b)
    {
        // 1. The two Nodes should be on the same level
        //       in the binary tree.
        /
/ 2. The two Nodes should not be siblings (means
        //    that they should not have the same parent
        //    Node).
        return ((level(node, a, 1) == level(node, b, 1)) &&
                (!nonTeam(node, a, b)));
    }


    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();
       //tree creation
        tree.root = new Node(1);
        //
        //
        Node Node1, Node2;
        Node1 = tree.root.left.left;
        Node2 = tree.root.right.right;
        if (tree.isTeam(tree.root, Node1, Node2))
            System.out.println("Yes");
        else
            System.out.println("No");
    }
}
```

Ans(b)

Time Complexity: O(n), three traversals are required. Two for calculating the level and one for checking that if the two students can't be in a team.

(number of traversals can be 1 or 2 too, depends on amount of information we want to store in each traversal)

Problem 5: Linked List Problem:

Suppose there is a list s1->s2->s3->…….->s_{n-1}->$s_{n.}$ given to you. You need to modify this list in such a way that it will be s1->s_n->s2->s_{n-1}->s3->s_{n-2}…….

Note: algorithm should be inplace without modifying the values of the nodes.

Marking Scheme: Total 10 marks (Algorithm explanation + Pseudo-code + runtime and data structure used)

Solution:

Method:

1) Break the list from middle into 2 lists.
2) Reverse the latter half of the list.
3) Now merge the lists so that the nodes alternate.

Pseudo code:

```
public class Solution {
  public  ListNode reverseList(ListNode root) {
                ListNode prev = null;

                ListNode curr = root;

                ListNode next = null;

                while (curr != null) {
                        // System.out.println(curr.val);
                        next = curr.next;
                        curr.next = prev;
                        prev = curr;
                        curr = next;
                }
```

```
            return prev;
        }
public ListNode reorderList(ListNode A) {


            if(A == null) return A;

            ListNode root = A;
            ListNode slow = root;
            ListNode fast = slow.next;
            while(fast!= null && fast.next != null) {
                slow = slow.next;
                fast = fast.next.next;
            }

            ListNode middleNode = slow.next;
            slow.next = null;

            ListNode r = reverseList(middleNode);

            ListNode temp = root;

            while(temp!= null && r!= null) {
                ListNode next1 = temp.next;
                temp.next = r;
                ListNode rNext = r.next;
                r.next = next1;
                r = rNext;
                temp = next1;
            }

            return root;
```

```
  }
}
```

Time complexity:

O(n) time is required for finding the middle node, O(n) time is required for reversing and O(n) is required for merging as well. Space complexity is O(1).

T(n) = n+n+n

=O(3n)=O(n)

Problem 6: x Number of players are participating in a sporting event.They are standing in a row. Each player has some match value associated with them. Now each player wants to make one friend on their right side having greater match value as compared to him and it should be the nearest player as well to him.
Your task is to print for each player their friend's index and his match value. If a player is not able to make such a friend print -1.

a) Find an Algorithm of Complexity O(n^2) to solve this problem. (5)
b) Find an efficient way to compute the friends in O(n) Complexity. (10)

Assume 1-based indexing.

Eg 1:- for the array [2, 5, 9, 3, 1, 12, 6, 8, 7] their are x = 9 players

Output(index of friend for each player):-[2, 3, 6, 6, 6, -1, 8, -1, -1]
For player at index1 the nearest friend is at index 2 having match value 5
For player at index2 the nearest friend is at index 3 having match value 9
For player at index3 the nearest friend is at index 6 having match value 12
For player at index4 the nearest friend is at index 6 having match value 12
For player at index5 the nearest friend is at index 6 having match value 12
For player at index6 there is no nearest friend found
For player at index7 the nearest friend is at index 8 having match value 8
For player at index 8 there is no nearest friend found
For player at index9 there is no nearest friend found

Marking Scheme:-Total 15 marks Part a:- 5 marks ; Part b:-10 marks
Rubrics:- For part a:-(2.5+2.5) (Algorithm Explanation(2.5)+Pseudo code(2.5))
For part b:-(5+2.5+2.5)(Algorithm Explanation with data Structure used(5)+Pseudo Code(2.5)+Runtime Analysis(2.5))

Solution :-

Marking Scheme:-
For part a:-(2.5+2.5) (Algorithm Explanation(2.5)+Pseudo code(2.5))
For part b:-(5+2.5+2.5)(Algorithm Explanation with data Structure used(5)+Pseudo Code(2.5)+Runtime Analysis or Time complexity Analysis(2.5))

Algorithm Explanation:-
Part a:-
Find the next greater match value for each player Using two loops.
Outer loop will be for iterating each player and inner loop will find the friend having greater match value.
Complexity :- $O(n^2)$

Pseudo Code:-

```
void printFriendIndex(int []arr){
    for(int i=0;i<arr.length;i++){
        System.out.print(arr[i]+" ");
    }
    return;
}

void makeFriend(int[] arr,int x){
    int []outputArr = new int[x];
    for(int i=0;i<x;i++){
            outputArr[i] = -1;
            for(int j=i+1;j<x;j++){
                if(arr[j]>arr[i]){
                    outputArr[i] = j+1;
                    break;
                }
            }
        }
```

```
            printFriendIndex(outputArr);
        }
```

Part B):-Approach:-We will preprocess all players on the right side of the current player to get the next greater match value player for the current player.
Method:-
Start processing each player's friend in reverse fashion i.e.. Starting from the rightmost player and going towards the leftmost.
2)For each current player check the stack:-

3)If the stack is empty which means no player with match value greater than the current player is present on the right of the current player.So simply get -1 as answer for this player.

Else the top most player in the stack might be the potential friend for the current player for this check the condition if the match value of topmost player in stack is greater than the current player.

4)If the condition is true we found our friend for current player

5) Else we will pop it out and again check the conditions in line 3.
6) After that we will push this player in the stack since it might be the potential friend for the remaining unprocessed players.
Do Similar iterations for other players on the left of the current player..
Time Complexity:-Since in each iteration we are pushing one player in the stack also we are popping some players out of the stack according to our condition.Also if a player is popped out from stack it will never again go in the stack.So in worst case n players move in stack and and n-1 players move out of the stack. So total computational time at max is n+n-1= 2n-1 = O(n) complexity.

Pseudo Code:-

```
        void printFriendIndex(int []arr){
          for(int i=0;i<arr.length;i++){
              System.out.print(arr[i]+" ");
          }
          Return; }

        void makeFriendEfficient(int[] arr,int x){
          Stack<Integer>stack = new Stack<Integer>();
```

```
        int[] outputArr = new int[x];
        for(int i=x-1;i>=0;i--){
            while(!stack.empty()&& arr[stack.peek()]<arr[i]){
                stack.pop();
            }
            if(stack.empty()){
                outputArr[i] = -1;
            }
            else{
                outputArr[i] = stack.peek()+1;
            }
            stack.push(i);
        }
        printFriendIndex(outputArr);
    }
```

Problem 7:-

Two monkeys Charlie and Leo are standing at two different nodes of a tree.
This tree is very weird, its root is situated at the highest point and grows in a downward direction.
Also it consists of several nodes and branches connecting two different nodes.But at max 2 branches can originate from a node.
Now ,there are some tasks which they have to perform using some algorithms.Since they are very lazy they want you to complete those tasks.
Task 1:- They want to move to the root.So they want to calculate the distance each of them have to travel in reaching to the root. (5)
Task2:-They want to meet with each other .So they want to compute the distance between each other. (10)

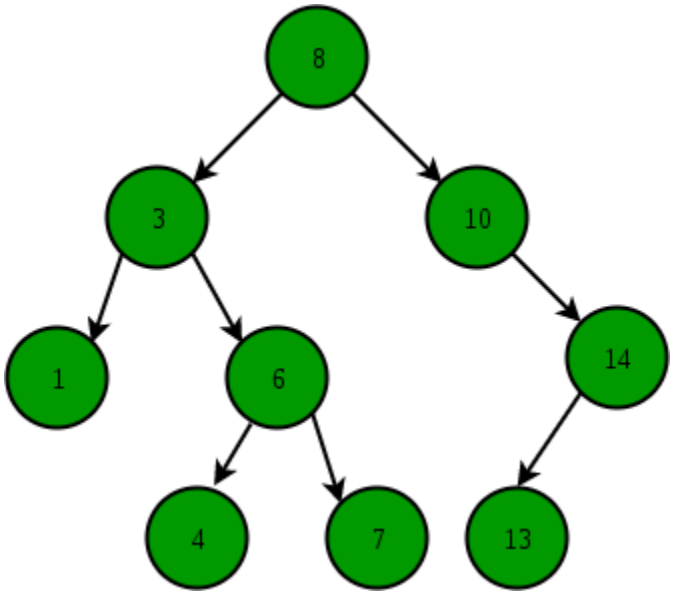
Can you help them in achieving these tasks?
Assumption:- You will be given the root Node of the tree and address of the nodes where charlie and leo are present

Marking Scheme:- Total Marks 15 marks (Task 1:- 5marks; Task 2:-10 marks)
Rubrics:-For Task 1:-(2+2+1) (Algorithm Explanation(2)+Pseudo code(2)+Time Complexity(1))
For Task 2:-(4+4+2)(Algorithm Explanation(4)+Pseudo Code(4)+Runtime Analysis(2))

Example of the tree:-

Input:-Here the root node is 8 and charlie is on node 1 and Leo is on node 7.
Output:- So the distance between root to charlie:-2
Distance between root to Leo:- 3

Distance between leo and Charlie:- 3



Solution:-

Marking Scheme:-
For Task 1:-(2+2+1) (Algorithm Explanation(2)+Pseudo code(2)+Time Complexity(1))
For Task 2:-(4+4+2)(Algorithm Explanation(4)+Pseudo Code(4)+Runtime Analysis(2))

For Task1:-
Algorithmic Explanation:-
We simply have to compute root to node path distance for each node.
Recursively or iteratively calculate the distance between the root node and Charlie and Leo nodes respectively.

Pseudo Code:-

```
public  int distance(Node root,Node n1){
        if(root==NULL)return -1;
```

```
    else{

        if(root==n1){
            return 0;
        }
        int left = distance(root.left,n1),right = distance(root.right,n1);
        if(left>=0)return left+1;
        else if(right>=0)return right+1;
        else return -1;
    }
}
```

Time Complexity:- In the worst Case we may need to traverse the whole tree but that will be the utmost once.So time complexity- O(n).

For Task 2:-

Algorithmic Explanation:-

1)We first have to find Lowest Common Ancestor(LCA) node(the common node from which both the nodes are reachable with shortest path) for both the monkeys and then

2)For calculating the distance between both monkeys we have to calculate

The distance = Distance of Charlie from root+Distance of leo from root-2*Distance of LCA from root

Distance of charlie and leo from root respectively is calculated in task 1 which can be directly used here.

Just calculate the distance of LCA from the root and find distance between charlie and leo from above formula.

Pseudo Code:-

```
public  Node LCA(Node root,Node p,Node q){
    if(root==null)return null;
    if(root==p || root==q){
        return root;
    }
    Node l = LCA(root.left,p,q),r = LCA(root.right,p,q)
    if(l!=null && r!=null)return root;
    else if(l!=null)return l;
    else{
        return r;
    }
}
```

```
public  distanceBetweenTwoNodes(Node root,Node p,Node q){
        Node lca = LCA(root,p,q)
      //distance function is from task 1
        int d1 = distance(root,p),d2 = distance(root,q),d3 = distance(root,lca)
        return d1+d2-2*d3

    }
```

Time Complexity:-

For Calculating LCA in worst Case:-We may have to traverse whole tree:-So complexity - n

For Calculating the Distance 3 times complexity-3n

So total time complexity = 3n+n = 4n = O(n) time complexity.