Index - 4

Binary Tree

1) Inorder Traversal

i. Recursive

```
ArrayList<Integer> inOrder(Node root)
   {
       ArrayList<Integer> ans = new ArrayList<>();
       traverse(root, ans);
       return  ans;
   }
   void traverse(Node root,ArrayList<Integer> ans){
       if(root==null)
           return;

       traverse(root.left ,ans);
       ans.add(root.data);
       traverse(root.right, ans);
   }
```

ii. Iterative

```
public List<Integer> inorderTraversal(TreeNode root) {
        Stack<TreeNode> s = new Stack<>();
        List<Integer> ans = new ArrayList<>();
        TreeNode cur = root;

        while(cur!=null || !s.isEmpty()){
            while(cur!=null){
                s.push(cur);
                cur = cur.left;
            }
            cur = s.pop();
            ans.add(cur.val);
            cur = cur.right;
        }
        return ans;
    }
```

T.C = O(n)
S.C = O(n)

2) PreOrder Traversal

i. Recursive

```
ArrayList<Integer> preorder(Node root)
{
        ArrayList<Integer> ans = new ArrayList<>();
        traverse(root, ans);
        return ans;
}
void traverse(Node root, ArrayList<Integer> ans)
{
        if(root==null)
                return;

        ans.add(root.data)
        traverse(root.left, ans);
        traverse(root.right, ans);
}
```

ii. Iterative

```
 public List<Integer> preorderTraversal(TreeNode root) {

        Stack<TreeNode> st = new Stack<>();
        List<Integer> answer = new ArrayList<>();
        TreeNode cur = root;
        while(cur!=null || !st.isEmpty())
        {
            while(cur!=null){
                answer.add(cur.val);
```

```
            if(cur.right!=null)
                st.push(cur.right);
            cur = cur.left;
        }
        if(!st.isEmpty())
            cur = st.pop();
    }
    return answer;
}
```

T.C = O(n)
S.C = O(n)

3) Postorder Traversal

```
ArrayList<Integer> postOrder(Node root)
    {
        ArrayList<Integer> ans = new ArrayList<>();
        traverse(root ,ans);
        return ans;
    }
    void traverse(Node root,ArrayList<Integer> ans)
    {
        if(root==null)
            return;
        traverse(root.left ,ans);
        traverse(root.right ,ans);
        ans.add(root.data);
    }
```

T.C = O(n)
S.C = O(n)

4) Left View of a BinaryTree

i. Recursive Level order

```
static int max = 0;
    ArrayList<Integer> leftView(Node root)
    {
        ArrayList<Integer> ans = new ArrayList<>();
        levelOrder(root,1 ,ans);
        return ans;
    }
    void levelOrder(Node root, int level,ArrayList<Integer> ans){
        if(root==null)
            return;
        if(max<level){
            ans.add(root.data);
            max = level;
        }
        levelOrder(root.left, level+1 ,ans);
```

```
        levelOrder(root.right ,level+1, ans);
    }
```

ii. Iterative Level order

```
ArrayList<Integer> leftView(Node root)
    {
        ArrayList<Integer> ans = new ArrayList<>();
        if(root==null)
            return ans;
        Queue<Node> q = new LinkedList<>();
        q.add(root);

        while(!q.isEmpty()){
            int size = q.size();

            for(int i=0;i<size ;i++){
                Node cur = q.remove();
                if(i==0)
                    ans.add(cur.data);

                if(cur.left!=null)
                    q.add(cur.left);
                if(cur.right!=null)
                    q.add(cur.right);
            }
        }
        return ans;
    }
```

5) Bottom View of a BinaryTree

Here Node has 4 attributes i. Data, ii. Left, iii. Right, iv. Hd

hd Represents the horizontal distance from the root

```
ArrayList<Integer> bottomView(Node root)
{
        if(root==null)
                return new ArrayList<>();

        Queue<Node> q = new LinkedList<>();
        Map<Integer, Integer> map = new TreeMap<>();

        root.hd = 0;
        q.add(root);
        int hd = 0;


        while(!q.isEmpty())
        {
```

```
            Node cur = q.remove();
            hd = cur.hd;

            map.put(hd, cur.data);

            if(cur.left!=null)
            {       cur.left.hd = hd-1;
                    q.add(cur.left);
            }
            if(cur.right!=null)
            {
                    cur.right.hd = hd+1;
                    q.add(cur.right);
            }
        }

        return new ArrayList<>(map.values());
}
```

T.C = O(n)
S.C = O(n)

Note:
map.values() =======> returns all the values present in the map
map.getSet() =======> returns all the keys present in the map

6) Top View of Binary Tree

```
static class TreeNode{
      Node node;
      int hd;
      TreeNode(Node root, int h){
         node = root;
         hd = h;
      }
   }
   static ArrayList<Integer> topView(Node root)
   {
      if(root==null)
         return new ArrayList<>();

      Queue<TreeNode> q = new LinkedList<>();
      Map<Integer,Integer> map = new TreeMap<>();
      q.add(new TreeNode(root,0));

      while(!q.isEmpty()){
         TreeNode cur = q.remove();

         if(!map.containsKey(cur.hd))
            map.put(cur.hd,  cur.node.data);
```

```
            if(cur.node.left!=null)
                q.add(new TreeNode(cur.node.left, cur.hd-1));

            if(cur.node.right!=null)
                q.add(new TreeNode(cur.node.right ,cur.hd+1));

        }

        return new ArrayList<>(map.values());
    }
```

T.C = O(n)
S.C = O(n)

7) Level Order Traversal of a BinaryTree

```
 static ArrayList <Integer> levelOrder(Node node)
    {
        Queue<Node> q = new LinkedList<>();
        ArrayList<Integer> a = new ArrayList<>();
        if(node==null)
            return a;
        q.add(node);

        while(!q.isEmpty()){
            Node cur = q.remove();
            if(cur.left!=null)
                q.add(cur.left);
            if(cur.right!=null)
                q.add(cur.right);
            a.add(cur.data);
        }
        return a;
    }
```

T.C = O(n)
S.C = O(n)

8) Spiral Level Order Traversal of a Binary Tree

i. It can be just solved by the above approach using a boolean variable and two different type of for loops
This is using Queue

ii. Using 2 Stacks it can be solved by following process

Push one level onto on stack and other level onto the other stack. While pushing take care of spiral pattern

```
ArrayList<Integer> findSpiral(Node root)
    {
        Stack<Node> s1 = new Stack<>();
        Stack<Node> s2 = new Stack<>();
        ArrayList<Integer> ans = new ArrayList<>();
        if(root==null)
            return ans;

        s1.push(root);

        while(!s1.isEmpty() || !s2.isEmpty()){
            while(!s1.isEmpty()){
                Node cur = s1.pop();
                if(cur.right!=null)
                    s2.push(cur.right);
                if(cur.left!=null)
                    s2.push(cur.left);
                ans.add(cur.data);
            }
            while(!s2.isEmpty()){
                Node cur = s2.pop();
                if(cur.left!=null)
                    s1.push(cur.left);
                if(cur.right!=null)
                    s1.push(cur.right);
                ans.add(cur.data);
            }
        }
        return ans;
    }
```

T.C = O(n)
S.C = O(n)

9) Height of a Binary Tree

```
 int height(Node node)
    {
        if(node==null)
            return 0;

        return 1 + Math.max(height(node.left),height(node.right));
    }
```

T.C = O(n)
S.C = O(n)

10) Diameter of a Binary Tree

Now the Diameter of a BT is the number of nodes in the longest path between any two nodes.

There is possibility that root is included or not. If included then leftHeight+rightHeight+1, else one of leftDiameter or rightDiameter

So diameter = Max(leftHeight+rightHeight+1, leftDiameter, rightDiameter)

Thus find the height also.

i. Two different functions for height and diameter

```
int diameter(Node root)
{
	if(root==null)
		return 0;

	int lHeight = height(root.left);
	int rHeight = height(root.right);

	int lDiameter = diameter(root.left);
	int rDiameter = diameter(root.right);

	int curNodeIncluded = lHeight + rHeight + 1;

	return  Math.max(curNodeIncluded, Math.max(lDiameter, rDiameter));
}
```

T.C = O(n^2)   Since we are finding height and diameter of traversing tree n^2 times
S.C = O(1)

ii. Calculate the height in the same function call using pointer

```
class Heights{
	int h;
}

int diameter(Node root)
{
	Height height = new Height();
	return findDiameter(root, height);
}

int findDiameter(Node root, Height height)
{
	if(root==null)
	{
		height.h = 0;
		return 0;
```

```
        }

        Height lHeight = new Height();
        Height rHeight = new Height();

        int lDiameter = findDiameter(root.left, lHeight);
        int rDiameter = findDiameter(root.right, rHeight);

        height.h = Math.max(lHeight.h, rHeight.h) +1;
        return Math.max( lHeight.h+ rHeight.h+1, Math.max( lDiameter, rDiameter);
}
```

T.C = O(n)
S.C = O(1)

11) Transform to Sum Tree

```
 public void toSumTree(Node root){
        sumTree(root);
   }
   public int sumTree(Node root){
       if(root==null)
          return 0;

        int cur = root.data;
        root.data = sumTree(root.left)+sumTree(root.right);

        return root.data+cur;
   }
```

12) Check for Height Balanced Tree

i. Calculate height separately

T.C = O(n^2)
S.C = O(1)

ii. Calculate height in same recursive call

```
class Height{
       int h;
}

boolean isBalanced(Node root)
   {
       Height height = new Height();
       return check(root, height);
   }
   boolean check(Node root,Height height){
       if(root==null){
          height.h = 0;
```

```
            return true;
        }
        Height lh = new Height(),rh = new Height();

        boolean l = check(root.left, lh);
        boolean r = check(root.right, rh);

        height.h = Math.max(lh.h ,rh.h)+1;

        if(l&&r&&(Math.abs(lh.h-rh.h)<=1))
            return true;

        return false;
    }
```

T.C = O(n)
S.C = O(1)

13) Determine if Two Trees are Identical

```
boolean isIdentical(Node root1, Node root2)
{
          if(root1==null && root2==null)
                return true;
             if(root1==null || root2==null)
                return false;

            return (root1.data==root2.data) && isIdentical(root1.left, root2.left) &&
                            isIdentical(root1.right root2.right);
}
```

14) LCA in a Binary Tree

i. **Find the paths of both the nodes and then traverse the paths and return the first node that is last common in the paths**

```
Node findAncestors(Node root, int n1, int n2)
{
        List<Node> path1 = new ArrayList<>();
        List<Node> path2 = new ArrayList<>();

        boolean p1 = findPath(root, n1, path1);
        boolean p2 = findPath(root, n2, path2);

        // if any one of the value is not in the tree
        if(!p1 || !p2)
                return null;

        int i=0;
```

```
	while(i<path1.size() && i<path2.size())
	{	if(path1.get(i)!=path2.get(i))
			break;
		i++;
	}
	return path1.get(i-1);
}

boolean findPath(Node root, int n, List<Integer> path)
{
	if(root==null)
		return false;

	path.add(root);
	if(root.data==n)
		return true;

	if(findPath(root.left, n, path) || findPath(root.right, n, path))
		return true;

	path.remove(path.size()-1);
	return false;
}
```

T.C = O(n) + O(n)
S.C = O(n) + O(n)

ii. Single traversal if both values are always present in the tree

```
Node lca(Node root, int n1, int n2)
{
	if(root==null)
		return root;

	if(root.data==n1 || root.data==n2)
		return root;

	Node l = lca(root.left, n1, n2);
	Node r = lca(root.right, n1, n2);

	if(l!=null && r!=null)
		return root;

	return (l!=null)?l:r;
}
```

iii. Single traversal if both values may or may not be present in the tree

```
boolean v1,v2;
```

```
Node findAncestor(Node root, int n1, int n2)
{
	v1 = false;
	v2 = false;
	Node ans = lca(root, n1, n2)
	if(v1 && v2)
		return null;
	return ans ;
}
Node lca(Node root, int n1, int n2)
{
	if(root==null)
		return root;
	Node temp = null;

	if(root.data==n1)
	{
		temp = root;
		v1 = true;
	}
	if(root.data==n2)
	{
		temp = root;
		v2 = true;
	}

	Node l = lca(root.left, n1, n2);
	Node r = lca(root.right, n1, n2);

	if(temp!=null)
		return temp;

	if(l!=null && r!=null)
		return root;

	return l!=null?l:r;
}
```

T.C = O(n)
S.C = O(1)

15) Symmetric Tree

i. Iterative Approach

If a tree is mirror of itself then the nodes at each level has to be palindrome.

So first left and right child of root.
Do until queue not empty

- **Remove two elements**
- **If any one of them is null then return false**

- **If both of them is null then continue**
- **If the values are not equal then return false**
- **Now push left node of n1 and right node n2, then right node of n2 and left node of n2.**
- **Continue above till queue is not empty**

```
boolean isSymmetric(Node root)
{
	Queue<Node> q = new LinkedList<>();
	q.add(root.left);
	q.add(root.right);

	while(!q.isEmpty())
	{
		Node n1 = q.remove();
		Node n2 = q.remove();

		if(n1==null && n2==null)
			continue;

		if(n1==null || n2==null)
			return false;

		if(n1.data!=n2.data)
			return false;

		q.add(n1.left);
		q.add(n2.right);
		q.add(n1.right);
		q.add(n2.left);
	}
	return true;
}
```

T.C = O(n)
S.C = O(n)

ii. Recursive Approach

```
boolean isSymmetric(Node root)
{
	return mirror(root, root);
}
boolean mirror(Node root1, Node root2)
{
	if(root1==null && root2==null)
		return true;
	if(root1==null || root2==null)
		return false;
	if(root1.data!=root2.data)
```

```
                return false;

        return mirror(root1.left, root2.right) && mirror(root1.right, root2.right);
}
```

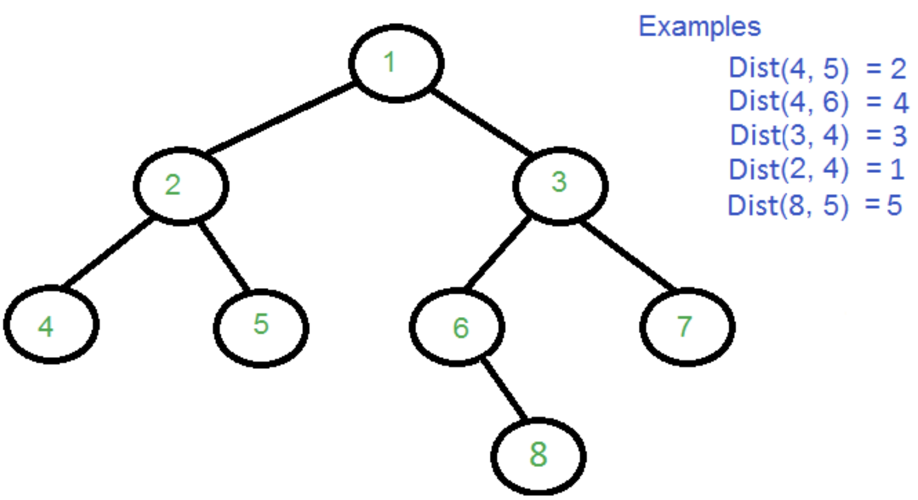
T.C = O(n)
S.C = O(1)

16) Maximum Path Sum

```
 int res;
    public int maxPathSum(TreeNode root) {
        if(root==null)
            return 0;
        res = Integer.MIN_VALUE;
        findPath(root);
        return res;
    }
    int findPath(TreeNode root){
        if(root==null)
            return 0;

        int l = findPath(root.left);
        int r = findPath(root.right);

        int curMax = Math.max(Math.max(l ,r)+root.val,Math.max(l+r+root.val, root.val));
        res = Math.max(res, curMax);
        return Math.max(Math.max(l ,r)+root.val, root.val);

 }
```

T.C = O(n)
S.C = O(1)

17) Min distance between two given nodes of a Binary Tree



i. Here Min path can be calculated by using LCA.
First Find LCA of both nodes
Now find the number of edges from lca a to node1 and to node2. Finally return the sum of their answer.

```
int findMinPath(Node root, int a, int b)
{
	Node lca = findLca(root, a, b);

	int path1 = findPath1(lca, a, 0);
	int path2 = findPath2(lca, b, 0);

	return path1+path2;
}

Node findLCA(Node root, int a, int b)
{
	if(root==null)
		return root;
	if(root.data==a || root.data==b)
		return root;

	Node l = findLca(root.left, a, b);
	Node r = findLca(root.right, a, b);

	if(l!=null && r!=null)
		return root;

	return l!=null?l:r;
}

int findPath(Node root, int n, int dist)
{
	if(root==null)
		return -1;

	if(root.data==n)
		return dist;

	int l = findPath(root.left, n, dist+1);
	if(l!=-1)
		return l;

	return findPath(root.right, n, dist+1);
}
```

Here we need to traverse the tree first to find lca then traverse again to find path from lca to node.

ii. Using the Formula

Distance between 2 node = (root to node1 path) + (root to node2 path) - 2*(root to lca path)

Also root to node path is noting but the no. of levels between them.
It can be found in lca function only. So time complexity will be better.

Also is one of the two nodes itself is lca then find the path.

```
int d1, d2, ans;
int minPath(Node root, int a, int b)
{
	d1 = -1;  // Initialise level of a to -1;
	d2 = -1; //     ‘’        ‘’      ‘’  b to -1;
	ans = 0; // Final ans

	Node lca = findLca(root, a, b, 1); // last parameter is level

	if(d1!=-1 && d2!=-1)
		return ans;


	// if d1 is lca
	if(d1!=-1)
		return findPath(lca, b, 0);    // passing 0 as dist because edges = n-1

	//d2 is lca
	if(d2!=-1)
		return findPath(lca, a, 0);

	return -1;
}

Node findLca(Node root, int a, int b, int lvl)
{
	if(root==null)
		return root;

	if(root.data==a)
	{
		d1 = lvl;
		return root;
	}
	if(root.data==b)
	{
```

```
            d2 = lvl;
            return root;
        }

        Node l = findLca(root.left, a, b, lvl+1);
        Node r = findLca(root.right, a, b, lvl+1);

        if(l!=null && r!=null)
        {
            ans = d1 + d2 - 2*lvl;
            return root;
        }

        return l!=null?l:r;
}

int findPath(Node root, int n, int dist)
{
        if(root==null)
            return -1;

        if(root.data==n)
            return dist;

        int l = findPath(root.left, n, dist+1);
        if(l!=-1)
            return l;

        return findPath(root.right, n, dist+1);
}
```

Here the time complexity is less than the previous one.

18) Flatten Binary Tree to Linked List

```
public void flatten(TreeNode root) {
      if(root==null)
         return;
      if(root.left==null && root.right==null)
         return;

      if(root.left!=null){
         flatten(root.left);
         TreeNode temp = root.right;

         root.right = root.left;
         root.left = null;

         TreeNode t = root.right;

         while(t.right!=null)
```

```
            t = t.right;
        t.right = temp;
    }
    flatten(root.right);
}
```

T.C = O(n)
S.C = O(1)

19) Construct Tree from Inorder & Preorder

i. **Use take the first element from pre and construct node for it. Now find it's position in inorder and call the buildTree from start to pos-1 and pos+1 to end.**
   Here if we do linear search to find the position in inorder traversal then

   T.C = O(n^2)
   S.C = O(1)

ii. Use HashMap to access elements quickly

```
HashMap<Integer,Integer> map;
    int pIndex;
   public  Node buildTree(int inorder[], int preorder[], int n)
   {
       map = new HashMap<>();
       pIndex = 0;

       for(int i=0;i<inorder.length ;i++)
           map.put(inorder[i],i);

       return constructTree(inorder, preorder, 0,n-1);
   }
    Node constructTree(int in[],int pre[],int start, int end){
       if(start>end)
           return null;

       int cur = pre[pIndex];
       pIndex++;
       Node node = new Node(cur);
       int pos = map.get(cur);
       if(start==end)
           return node;

       node.left = constructTree(in, pre, start ,pos-1);
       node.right = constructTree(in ,pre, pos+1, end);
       return node;
   }
```

T.C = O(n)
S.C = O(n)

20) Construct Tree from Postorder and Inorder

```
 Map<Integer,Integer> map;
  int pIndex;
   Node buildTree(int in[], int post[], int n) {

       map = new HashMap<>();
       pIndex = n-1;
       for(int i=0;i<n;i++)
           map.put(in[i],i);

       return constructTree(in ,post, 0,n-1);
   }
   Node constructTree(int in[],int post[],int start ,int end){
       if(start>end)
           return null;

       int cur = post[pIndex];
       pIndex--;
       Node node = new Node(cur);
       if(start==end)
           return node;

       int pos = map.get(cur);


       node.right = constructTree(in, post, pos+1, end);
       node.left = constructTree(in, post, start, pos-1);

       return node;
   }
```

T.C = O(n)
S.C = O(n)