TREE
DATA
STRUCTURE

Content

6.1 Height of Binary Tree

```
int height(Node node)
{
        if (node == null)
                return 0;
        else
        {
                /* compute the height of each subtree */
                int l = height(node.left);
                int r = height(node.right);

                /* use the larger one */
                if (l > r)
                     return (l + 1);
                else
                        return (r + 1);
        }
}
```

6.2 Count Leaves in Binary Tree

```
int countLeaves(Node node)
{
    if ( node.left == null && node.right == null ) {
        return 1;
    }
    int left = countLeaves ( node.left ) ;
    int right = countLeaves ( node.right ) ;
    return ( left + right ) ;
}
```

6.3. Lowest Common Ancestor in a BST

```
static node lca(node root, int n1, int n2)
{
        while (root != null)
        {
                // If both n1 and n2 are smaller
                // than root, then LCA lies in left
                if (root.data > n1 && root.data > n2)
                        root = root.left;

                // If both n1 and n2 are greater
                // than root, then LCA lies in right
                else if (root.data < n1 && root.data < n2)
                        root = root.right;

                else break;
        }
        return root;
}
```

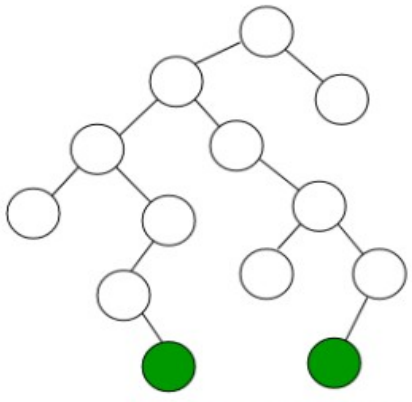
Note : Assume both are present

6.4. Determine if two trees are identical

```
boolean isIdentical ( Node root1, Node root2 )
{
        if ( root1 == null && root2 == null ) {
                return true ;
        }
        if ( root1== null ) {
                return false ;
        }
        if ( root2 == null ) {
                return false;
        }
        if ( root1.data != root2.data ) {
                return false;
        }
        return ( isIdentical ( root1.left, root2.left ) &&
                isIdentical ( root1.right, root2.right ) ) ;

}
```

6.5. Diameter of binary tree

```
int diameter ( Node node ){
        if ( node == null )
                return 0;
        int left = height (node.left ) ;
        int right = height (node.right ) ;
        int lDiameter = diameter ( node.left ) ;
        int rDiameter = diameter ( node.right ) ;
        //return max between lDiameter,rDiameter,(left+right+1)
        int max = left + right + 1 ;
        if ( lDiameter > max)
                max = lDiameter ;
        if ( rDiameter > max)
                max = rDiameter ;
        return max;
}
int height ( Node node ) {
        if ( node == null)
                return 0;
        int left = height ( node.left ) ;
        int right = height ( node.right ) ;
        if ( left > right ){
                return left + 1;
        }else
                return right + 1;
 }
```



Diameter 9 nodes not through root

6.6. Mirror tree

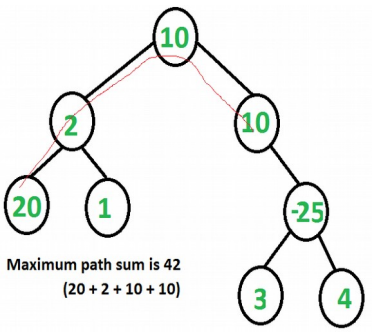
```
void mirror ( Node node)
{
        if ( node != null )
        {
                // swap left and right child of node
                Node temp = node.left ;
                node.left = node.right ;
                node.right = temp ;

                mirror ( node.left ) ;
                mirror ( node.right ) ;
        }
}
```

6.7. Left view of binary tree

```
int max_level = 0 ;
void leftViewUtil ( Node root, int level ) {
        if ( root != null ) {
                if ( max_level < level ) {
                        System.out.print ( root.data + " " ) ;
                        max_level = level;
                }
                leftViewUtil ( root.left, level + 1 ) ;
                leftViewUtil ( root.right, level + 1 ) ;
        }
}

void leftView ( Node root )
{
        leftViewUtil ( root, 1 ) ;
}
```

6.8. Maximum sum path

```
class Res{
        public int result;
}
class GfG {
        static int maxPathUtil ( Node root, Res res ) {
                if ( root == null ) {
                        return 0;
                }
                int l = maxPathUtil ( root.left, res ) ;
                int r = maxPathUtil ( root.right, res ) ;

                int res_single = Math.max ( Math.max ( l , r ) + root.data,
root.data);
                int res_top = Math.max ( res_single, l + r + root.data);
                if ( res.result < res_top ) {
                        res.result = res_top;
                }
                return res_single;
        }
                public static int maxPathSum ( Node root ){
                        Res res = new Res();
                        res.result = Integer.MIN_VALUE;
                        maxPathUtil ( root, res );
                        return res.result;
                 }

 }
```



6.9. Check for balanced tree

```
private int height(Node root){
      if(root == null){
         return 0;
      }
      int l = height ( root.left );
      int r = height ( root.right );
      if (l > r)
         return l + 1;
      else
         return r + 1;
}

boolean isBalanced (Node root)
{
         if (root == null ){
               return true;
         }
         if ( ! isBalanced ( root.left ) ) {
               return false;
         }
         if( ! isBalanced ( root.right ) ) {
               return false;
         }
         int lHeight = height ( root.left ) ;
         int rHeight = height ( root.right ) ;
         int r = Math.abs ( lHeight – rHeight ) ;
         if ( r > 1 ) {
               return false;
         }
         return true;
  }
```

6.10 Lowest common ancestor in a binary tree

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

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

        if ( left == null ) {
                return right;
        }
        if ( right == null ) {
                return left;
        }
        return root;
}
```

Note : Assume both are present

6.11. Level order traversal in spiral form

```
void printSpiral ( Node node)
{
        if(node!=null){
                Stack<Node> s1 = new Stack<>();
                Stack<Node> s2 = new Stack<>();

                s1.push(node);
                while(!s1.empty() || !s2.empty()){
                        while(!s1.empty()){
                                Node n = s1.peek();
                                s1.pop();
                                System.out.print(n.data+" ");
                                if(n.right!=null){
                                        s2.push(n.right);
                                }
                                if(n.left!=null){
                                        s2.push(n.left);
                                }
                        }
                        while(!s2.empty()){
                                Node n = s2.peek();
                                s2.pop();
                                System.out.print(n.data+" ");
                                if(n.left!=null){
                                        s1.push(n.left);
                                }
                                if(n.right!=null){
                                        s1.push(n.right);
                                }
                        }
                }
        }
}
```

6.12. Vertical traversal of binary tree

```
class Values { int min, max ;  }
class BinaryTree {
        Values val;
        void minMax ( Node node, Values min, Values max, int hd ) {
                if ( node == null ) {
                        return;
                }
                if ( hd < min.min ) {
                        min.min = hd;
                }else if ( hd > max.max ) {
                        max.max = hd;
                }
                minMax ( node.left, min, max, hd - 1 ) ;
                minMax ( node.right, min, max, hd + 1 ) ;
        }

        void printGivenLine(Node node,int line_no,int hd){
                if ( node == null ) {
                        return;
                }
                if ( line_no == hd ) {
                        System.out.print(node.data+" ");
                }
                printGivenLine(node.left,line_no,hd-1);
                printGivenLine(node.right,line_no,hd+1);
        }
        void verticalOrder(Node root) {
                val = new Values();
                minMax(root,val,val,0);
                for(int line_no = val.min ; line_no <= val.max ; line_no++){
                        printGivenLine(root,line_no,0);
                }
        }
}
```

Using HashMap (Order varies) nlogn: for order use queue level order

```
static void getVerticalOrder(Node node, hd, TreeMap < Integer,
Vector<Integer>> m ) {
        if ( node == null ) {
                return;
        }
        Vector < Integer > get = m.get ( hd ) ;
        if ( get == null ) {
                get = new Vector<>();
                get.add(node.data);
        }else
                get.add(node.data);

        m.put(hd,get);

        getVerticalOrder ( node.left, hd – 1, m);
        getVerticalOrder ( node.right, hd + 1 , m);
}

 static void verticalOrder (Node root)
{
        TreeMap<Integer,Vector<Integer>> m = new TreeMap<>();
        getVerticalOrder ( root, 0, m) ;

        //print map
        for(Map.Entry<Integer,Vector<Integer>> entry: m.entrySet()){
                Vector<Integer> temp = entry.getValue();
                Iterator value = temp.iterator();
                while(value.hasNext())
                        System.out.print(value.next()+" ");
        }
}
```

6.13. Bottom view of binary tree

```
public void bottomView ( Node root )
{
	if ( root != null ) {
		TreeMap<Integer,Integer> map = new TreeMap<>();
		LinkedList<Node> queue = new LinkedList<>();

		root.hd = 0;
		queue.add ( root ) ;
		while ( ! queue.isEmpty() ) {
			Node node = queue.poll () ;
			int hd = node.hd;
			map.put ( hd, node.data ) ;
			if ( node.left != null ) {
				node.left.hd = hd – 1 ;
				queue.add ( node.left ) ;
			}
			if ( node.right != null ) {
				node.right.hd = hd + 1 ;
				queue.add ( node.right ) ;
			}
		}
		Set<Entry<Integer,Integer>> set = map.entrySet();
		Iterator<Entry<Integer,Integer>> iterator = set.iterator();
		while ( iterator.hasNext() ) {
			Map.Entry<Integer,Integer> me = iterator.next();
			System.out.print ( me.getValue() + " " ) ;
		}
	}
}
```

6.14. Check for BST

```
boolean isBSTUtil ( Node node, int min, int max ) {
        if ( node == null) {
                return true;
        }
        if ( node.data < min || node.data > max ) {
                return false;
        }
        return ( isBSTUtil ( node.left, min, node.data – 1 ) &&
isBSTUtil(node.right, node.data + 1 , max ) ) ;
}


 int isBST(Node root)
 {
        if( isBSTUtil ( root, Integer.MIN_VALUE ,
Integer.MAX_VALUE)){
                return 1;
        }
        return 0;

 }
```

6.15. Connect nodes at same level

```
static void connect ( Node root )
{
	if ( root != null ) {
		LinkedList<Node> queue = new LinkedList<>();
		queue.add ( root ) ;
		queue.add(null);
		while(!queue.isEmpty()){
			Node q = queue.poll();
			if(q!=null){
				q.nextRight = queue.peek();
				if(q.left!=null){
					queue.add(q.left);
				}
				if(q.right!=null){
					queue.add(q.right);
				}
			}else if(!queue.isEmpty()){
				queue.add(null);
			}
		}
	}
}
```

6.16. Binary tree to DLL (Doubly linked list)

```
Node head = null;
Node prev = null;
void bToD ( Node root )
{
        if ( root == null ) {
                return;
        }
         bToD ( root.left ) ;
         if ( prev == null ) {
                head = root;
        }else{
                root.left = prev;
                prev.right = root;
        }
        prev = root;
        bToD ( root.right ) ;
}
Node bToDLL(Node root)
{
         bToD ( root ) ;
         return head;
}
```

6.17. Fixing two nodes of BST

```
Node prev, first, middle, last ;
public void correctBSTUtil ( Node node ) {
        if ( node != null ) {
                correctBSTUtil ( node.left ) ;
                if ( prev != null && node.data < prev.data ) {
                        if ( first == null ) {
                                first = prev;
                                middle = node;
                        }else
                                last = node;
                }
                prev = node;
                correctBSTUtil(node.right);
        }
}

public Node correctBST(Node root)
{
        prev = first =  middle = last = null;
        correctBSTUtil ( root ) ;
        if ( first != null && last != null ) {
                int temp = first.data;
                first.data = last.data;
                last.data = temp;
        }else if ( first != null ) {
                int temp = first.data;
                first.data = middle.data;
                middle.data = temp;
        }
        return root;
}
```

6.18. Find path of a node from root in a binary tree

```
private boolean findPath(Node root, int n, List<Integer> path)
{
        // base case
        if (root == null) {
                return false;
        }
        // Store this node . The node will be removed if
        // not in path from root to n.
        path.add(root.data);

        if (root.data == n) {
                return true;
        }

        if (root.left != null && findPath(root.left, n, path)) {
                return true;
        }

        if (root.right != null && findPath(root.right, n, path)) {
                return true;
        }

        // If not present in subtree rooted with root, remove root from
        // path[] and return false
        path.remove(path.size()-1);

        return false;
}
```