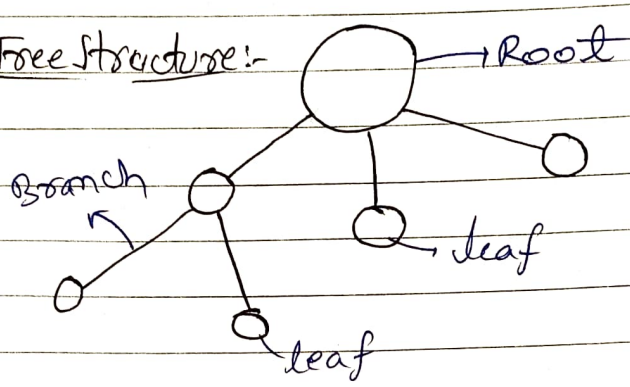
BINARY TREE

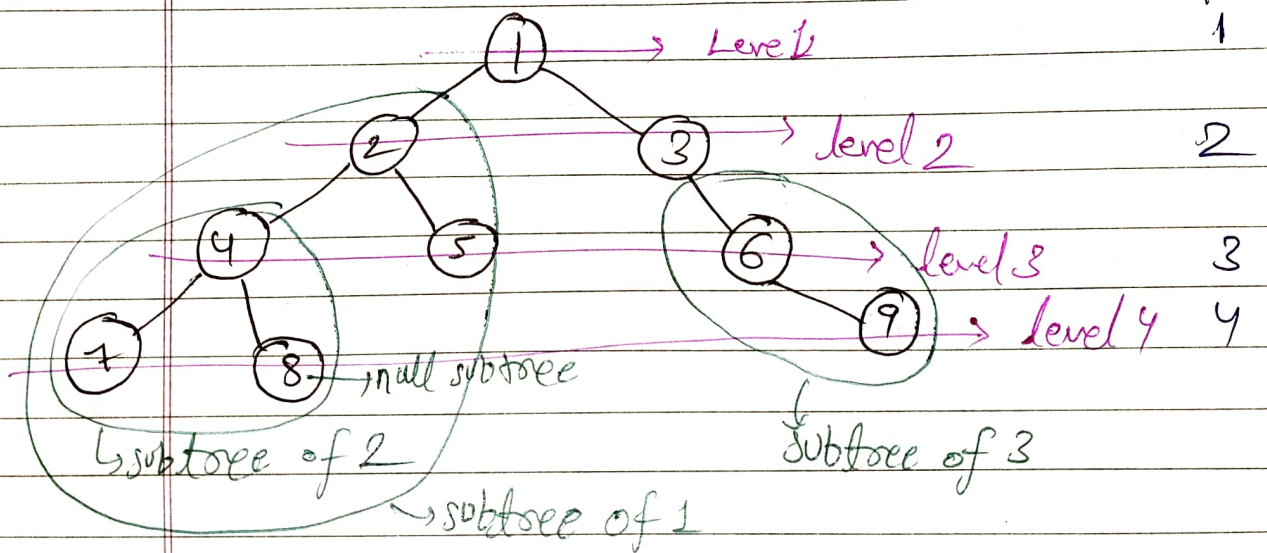
→ Hierarchial Data Structure

Ex- Family Tree Structure

Tree Structure:-



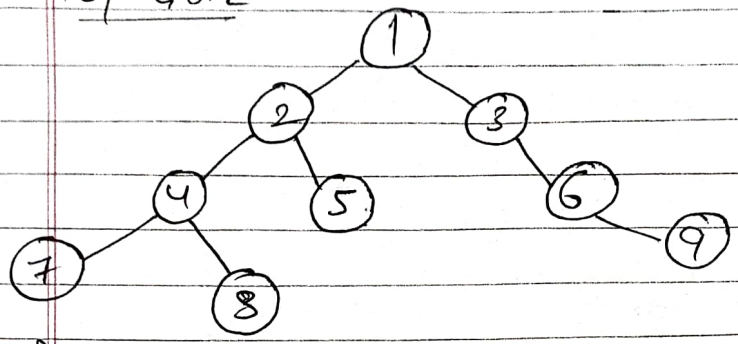
Binary Tree – At max 2 children



Depth
1
2
3
4

PAGE NO.
DATE: / /

Pop Quiz



a) Children of 4: 7, 8
b) No. of leaves: 4, (7, 8, 5, 9)
c) Parent of 6: 3
d) Level of 2: 2
e) Subtrees of 1 & 2:
f) Ancestors of 8: (2, 1, 4)

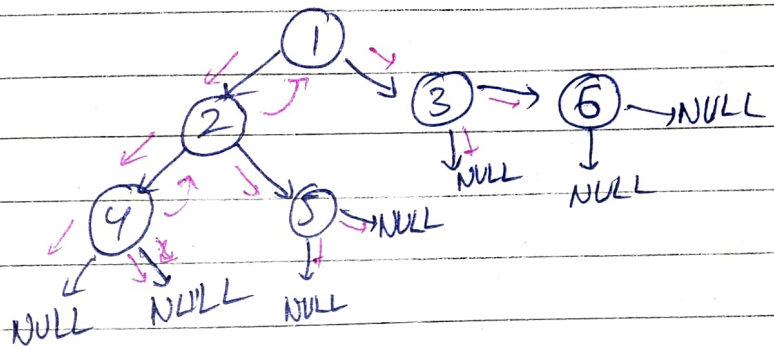
Build Tree Preorder

preorder sequence

1, 2, 4, -1, -1, 5, -1, -1, 3, -1, 6, -1, -1

null null

In preorder first node data is given then left subtree data is given then right subtree



PAGE NO.
DATE: / /

Code :-

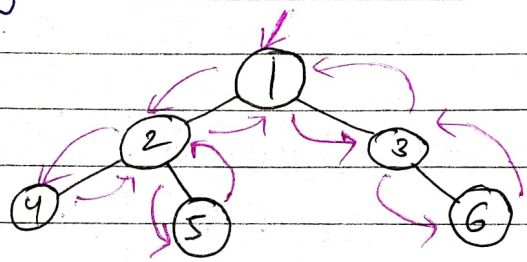
```
package Build_Tree_Preorders;

public class Solution {

Public class Solution {

    static class Node {
        int data;
        Node left;
        Node right;
        public Node (int data){
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }
    static class BinaryTree {
        static int index = -1;
        public static Node buildTree (int nodes[]){
            index++;
            if(nodes[index] == -1){
                return null;
            }
            Node newNode = new Node (nodes[index]);
            newNode.left = buildTree (nodes);
            newNode.right = buildTree (nodes);
            return newNode;
        }
    }
```

PAGE NO.
DATE: / /

```
psvm (String args[]){
    int [] nodes = {-1,2,4,-1,-1,5,-1,-1,3,-1,6,7,-1};
    Node rootNode = BinaryTrees.buildTree(nodes);
    Sout (rootNode.data);
}
}
```

Tree Traversal

a) Preorder
1) Print root
2) print left subtree
3) print right subtree



O(n) → Time complex

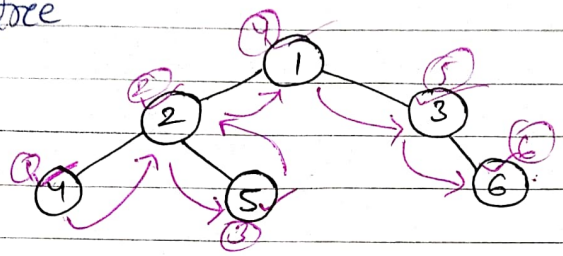
Output :- 1 2 4 5 3 6

It is called preorder because root gets accessed first.

Code-: In the same BinaryTree class

```
public static void preorder (Node root){
    if (root == null){
        return;
    }
    Sout (root.data + " ");
    preorder (root.left);
    preorder (root.right);
}
```

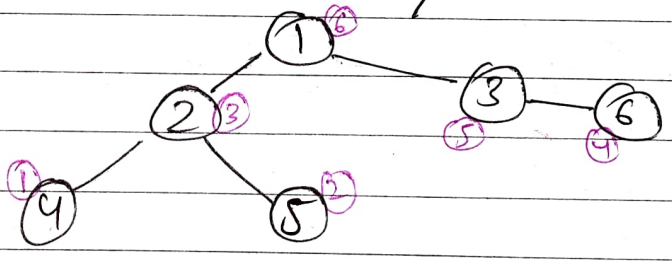
b) In order
1) Left Subtree
2) Root
3) Right Subtree




```
public static void inorder (Node root){
        if(root == null) return;
        inorder (root.left + " ");
        System.out.print (root.data+" ");
        inorder (root.right);
}
```

Depth First Search Approach

c) Post order
1) Left subtree
2) right subtree
3) root

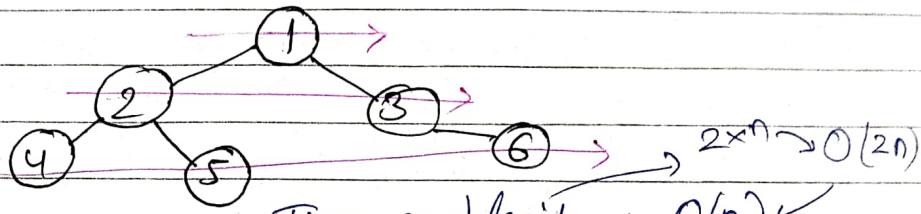


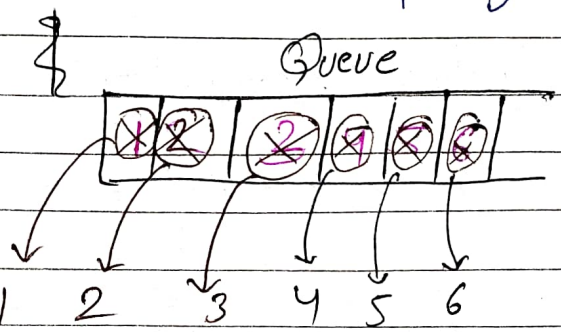

```
public static void postorder (Node root){
        if(root = null) return;
        postorder (root.left);
        postorder (root.right);
        Sout(root.data+" ");
}
```

PAGE NO.
DATE: / /

Level Order Traversal

Breadth First Search





1. Traverse through each level
2. ~~Push~~ Add the elements from left to right in Queue at that level
3. While removing from Queue add the nodes left and node's right in that Queue.
4. ~~When encounter null~~ when level gets change add null to Queue.
5. When null gets out of Queue print next line.

1 n
2 3 n
4 5 6 n

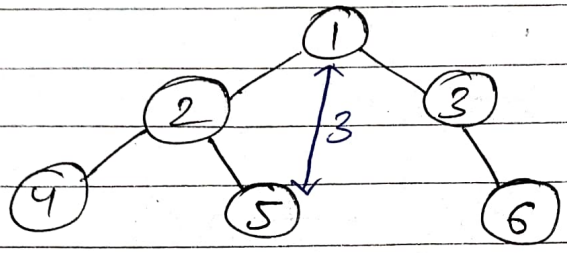
PAGE NO.
DATE: / /

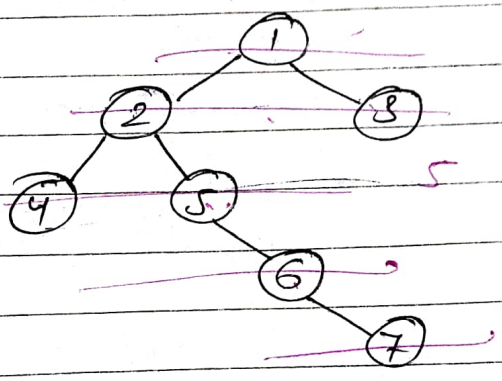
```
public static void levelorder(Node root){

    if(root == null) return;
                                    LinkedList<>();
    Queue<Node> q = new ~~ArrayDeque<>~~();

    q.add(root);
    q.add(null);

    while(!q.isEmpty()){

        Node currNode = q.remove();

        if(currNode == null){
            System.out.println();

            if(q.isEmpty()){
                break;

            }else{
                q.add(null);
            }
        else{
            System.out.print(currNode.data+" ");
            if(currNode.left != null){
                q.add(~~currNode~~)
                q.add(currNode.left);
            }
            if(currNode.right != null){
                q.add(currNode.right);
            }
        }
    }
}
```

ArrayDeque throws null pointer exception when adding null element.

Height of a tree

Maximum distance between root node and leaf node.





1. Height of null is 0.
2. Height of last node (leaf) is 1.
3. Calculate height of left leaf and right leaf.
4. return Max (left, right) and recursively calculate Max(" , ") till root node

Code:-

```
public static int height(Node root){

    if(root == null) return 0;

    int leftHeight = 1+height(root.left);
    int rightHeight = 1+height(root.right);

    return Math.max(leftHeight, rightHeight);

}
```

PAGE NO.
DATE: / /

Count of Nodes

→ Use your recursive mind

In the same BinaryTrees class

```
public static int countNodes(Node root){
    if(root == null) return 0;
    return countNode(root.left)+countNodes(
        root.right) +1;
}
```

Sum of Nodes

→ Use your recursive ~~mid~~ mind

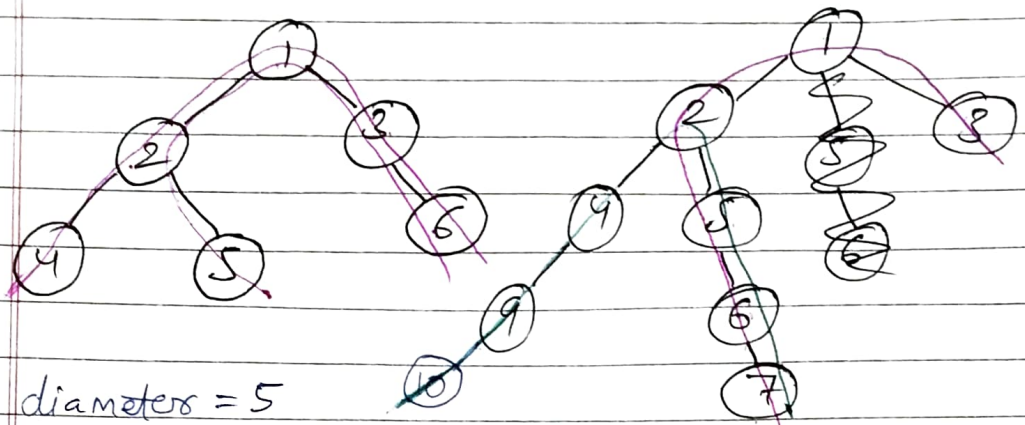
In the same BinaryTrees class

```
public static int sumNodes(Node root){
    if(root == null) return 0;
    return root.data + sumNodes(root.left)+
        sumNodes(root.right)
```

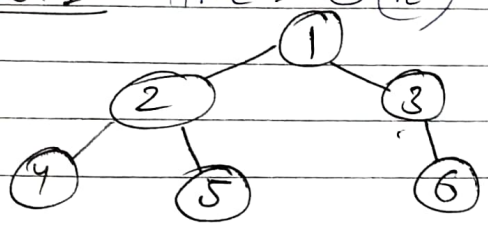
PAGE NO.
DATE: / /

Diameter of a Tree

No of nodes in the longest path between two leaves.



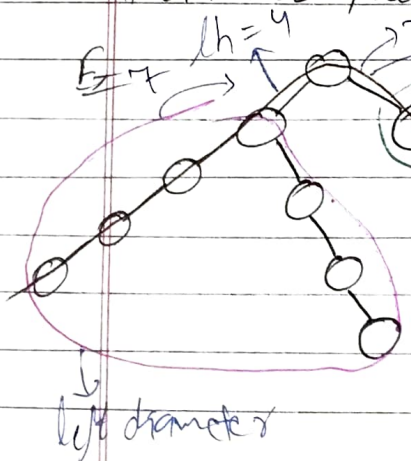
Approach 1:- Time : $O(n^2)$



// diameter path passes through root

diam = left height + right height + 1

// diameter path doesn't pass through root



root Diam = 4 + 1 + 1 = 6

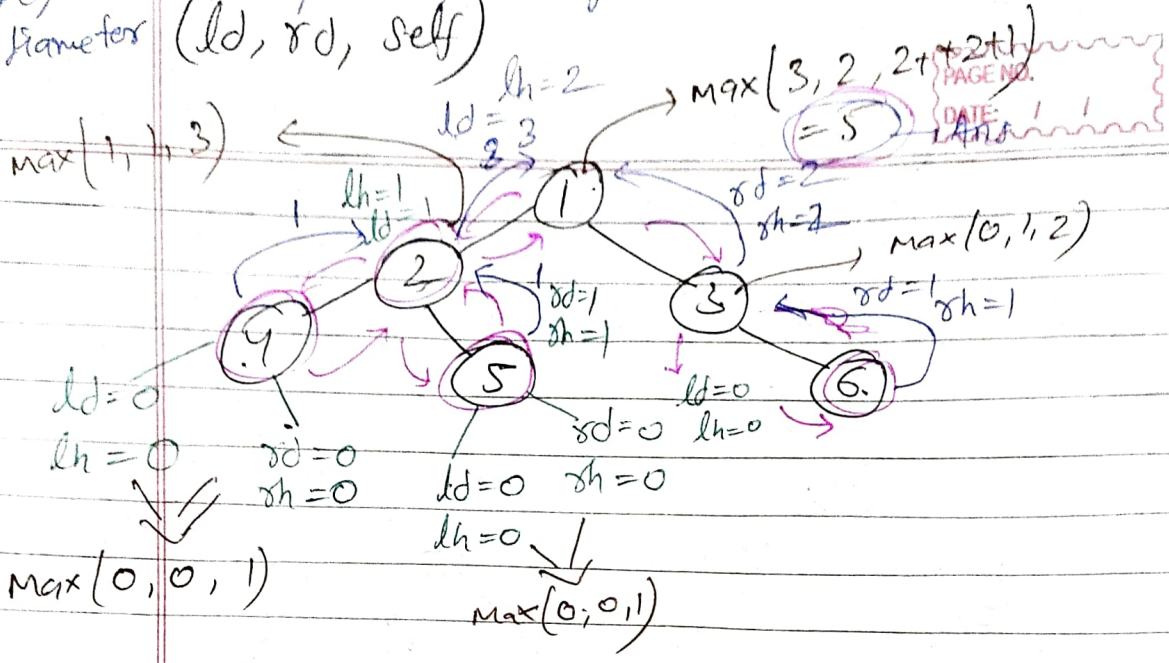
ldiam = 7

rdiam = 1

} Max.

left diameter ← right diameter → self diameter = lh + rh + 1

(ld, rd, self)



ldiam = diam (root.left)
rdiam = diam (root.right)

lh = height (root.left)
rh = height (root.right)

self diam = lh + rh + 1

return max (ldiam, rdiam, selfdiam)

Time complexity is order of $O(N^2)$ because first we are calculating diameter $O(N)$ then height of the same node $O(N) \Rightarrow O(N^2)$

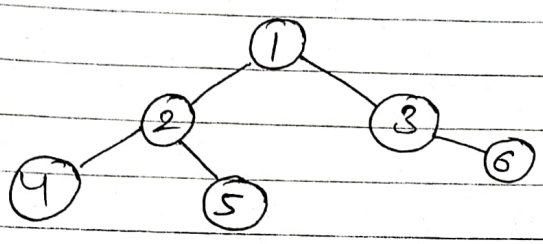
→ In the same ~~Binary Tree~~ class

Code:

```
public static int diameter(Node root) {
    if(root == null) return 0;
    int leftDiameter = diameter(root.left);
    int rightDiameter = diameter(root.right);
    int leftHeight = height(root.left);
    int rightHeight = height(root.right);
    int selfDiameter = leftHeight + rightHeight + 1;
    return Math.max(selfDiameter, Math.max(rightDiameter,
                    leftDiameter));
}
```

PAGE NO.
DATE: / /

Approach 2 -:




```
class Info{                    return Info
    int diam;
    int ht;
}
```

Code -:

```
static class Info{
    int diam;
    int ht;

    public Info(int diam, int ht){
        this.diam = diam;
        this.ht = ht;
    }
}
public static Info diameters(Node root){

    if(root == null) return new Info(0,0)

    Info leftInfo = diameter(root.left);

    Info rightInfo = diameters(root.right);
```

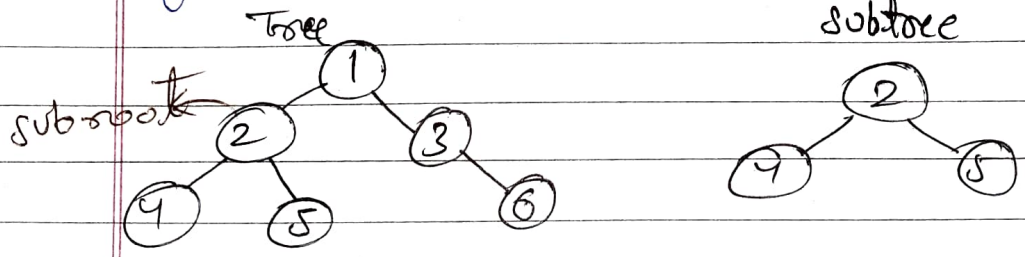
Math.max(PAGE NO. DATE: / /

```
int diam = Math.max(leftInfo.diam,
        rightInfo.diam), leftInfo.ht +
        rightInfo.ht + 1);

int ht = Math.max(leftInfo.ht, rightInfo.ht) + 1;

return new Info(diam, ht);
}
```

Time complexity: $O(N)$ since we visited each node only once and saved it's diameter & height in Info object.

Subtree of another Tree

Given the roots of two binary trees root and subroot, return true if there is a subtree of root with the same structure and node values of subroot and return false otherwise.



PAGE NO.
DATE: / /

1. find subroot in tree
2. check identical (given subtree, node subtree)

non-identical

1. node.data != subroot.data
2. node = null || subroot = null
3. left subtree → nonidentical
4. right subtree → nonidentical

} false else true

Code-:

```
public static boolean isSubtree (Node root, Node subroot){
    if( root == null) return false;
                ↳ initially or after reaching a leaf
    if( root.data == subroot.data) {
        if( isIdentical (root, subroot)){
            return true;
        }
    }
    return isSubtree (root.left, subroot) || isSubtree (root.right, subroot);
}
```

} checking

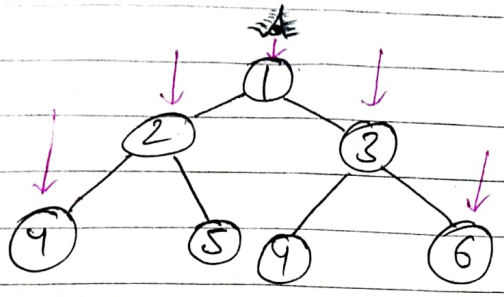
→ traversing & checking ←

```
public static boolean isIdentical (Node root, Node subroot){
    if( root == null && subroot == null) return true;
                ↳ identical when both null ↙
    if((root == null || subroot == null) || root.data != subroot.data){
        return false;
    else if( !isIdentical (root.left, subroot.left) ||
             !isIdentical (root.right, subroot.right)) return false;
    else return true;
```

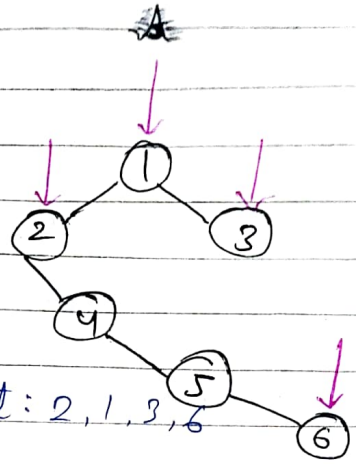
non identical

PAGE NO.
DATE: / /

Top View of a tree



Output: 4 2 1 3 6



Output: 2, 1, 3, 6

Hashmap (Brief)

Map (key, value)

Key unique

Countries	Population
"China"	125
"India"	100
"US"	50
"Nepal"	5
"Bangladesh"	5
String	Integer

Countries	Population

table gets formed in memory

java.util.Hashmap;

① `HashMap <String, Integer> Map = new HashMap<>();`

i) We can't create duplicate values, hashmap updates the existing one if we try.

ii) Add - O(1)
Remove - O(1)
Get - O(1)
} Implements hashing algorithm

PAGE NO.
DATE: / /

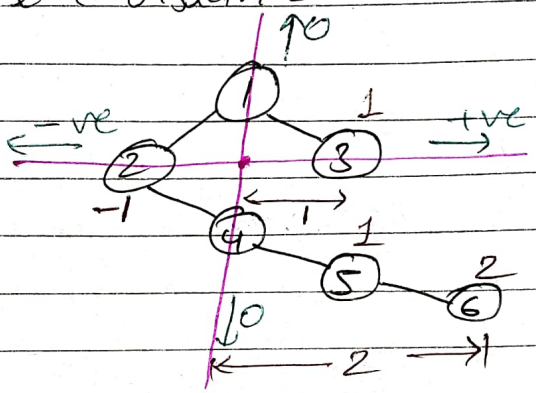
② Add

map.put(key, value)

③ Get

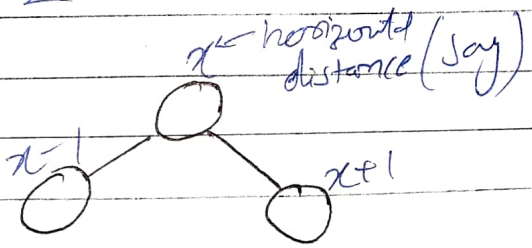
map.get(key)

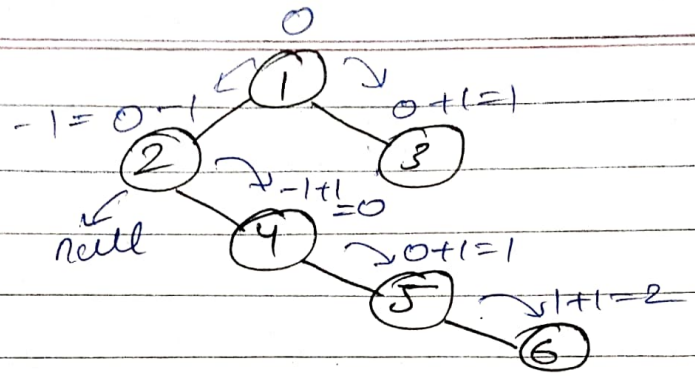
Horizontal distance :-



In top view if encounter of any horizontal distance is considered

	HD	node
repeats ignore	0	1
	-1	2
repeats ignore	1	3
	2	6





Approach :- Use Level order Traversal

Use map to track horizontal distances
Use map.containskey() to check if any HD already exists

map

HD (key)	node (value)
0	1
-1	2
1	3
2	6

Min = 0
Max = 0

```
static class Info{
    Node node;
    int hd;
    public Info(Node node, int hd){
        this.node = node;
        this.hd   = hd;
    }
}
```

PAGE NO.
DATE: / /

topView Code:-

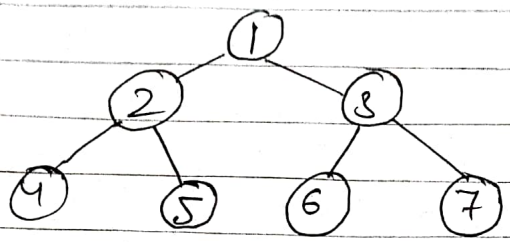
```
public static void topView(Node root){
    Queue<Info> q = new LinkedList<>();
    HashMap<Integer, Node> map = new HashMap<>();
    int min = 0, max = 0;          // → used in traversing map
    q.add(new Info(root, 0));
    q.add(null);

    while(!q.isEmpty()){
        Info curr = q.remove();
        if(curr == null){
            if(q.isEmpty()){
                break;
            } else{
                q.add(null);
            }
        } else{
            if(!map.containsKey(curr.hd)){   // if hd is not in map then only we add
                map.put(curr.hd, curr.node);
            }
            if(curr.node.left != null){      // checking for left node
                q.add(new Info(curr.node.left, curr.hd-1));   // just left hd
                min = Math.min(min, curr.hd-1);
            }
            if(curr.node.right != null){
                q.add(new Info(curr.node.right, curr.hd+1));  // just right hd
                max = Math.max(max, curr.hd+1);
            }
        }
    }
    for(int i=min; i<=max; i++){
        Sout(map.get(i).data + " ");
    }
}
```

same we do in levelorder

traversing map

PAGE NO.
DATE: / /

Kth Level of a Tree



Approach 1:- Iteratively:-

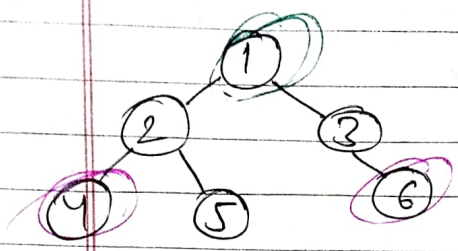
```
Publi static void kLevel (Node root, int k){
  if(root == null) return;

  Queue<Node> q = new LinkedList<>();
  int lvl = 1;
  q.add(root);
  q.add(null);

  While(!q.isEmpty()){
    Node currNode = q.remove();
    if(currNode == null){
      lvl++;
      if(q.isEmpty()){
        break;
      }else{
        q.add(null);
      }
    }else{
      if(lvl==k) Sout(currNode.data + " ");
      if(currNode.left != null) q.add(currNode.left);
      if(currNode.right != null) q.add(currNode.right);
    }
```

PAGE NO.
DATE: / /

Approach 2:- Recursively

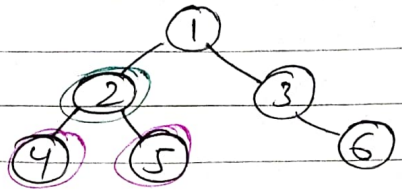
```
public static void kLevel(Node root, int k){
    if(root == null){
        return;
    }
    if(k == 2){
        if(root.left == null && root.right == null){
            return;
        }
        else if (root.left == null){
            Sout (root.right.data + " ");
            return;
        }
        else if (root.right == null){
            Sout (root.left.data + " ");
            return;
        } else{
            Sout (root.left.data + " " + root.right.data + "
            return;
        }
    }
    kLevel (root.left, k-1);
    kLevel (root.right, k-1);
}
```

Lowest Common Ancestor



n1 = 4 n = 2

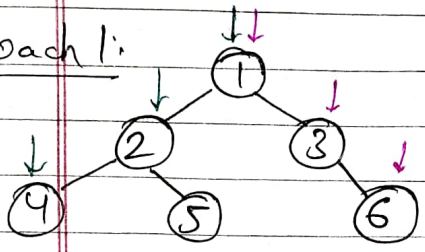
ans = 1



n1 = 4 n = 5

ans = 2

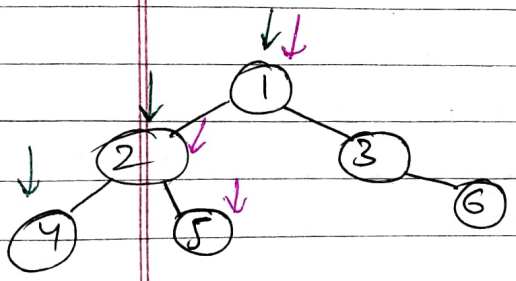
Approach 1:



path1 n1=4 | 1 | 2 | 4 |

path2 n2=6 | 1 | 3 | 6 |

lca = 1



path1 n1=4 | 1 | 2 | 4 |

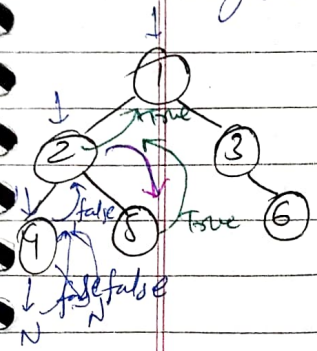
path2 n2=5 | 1 | 2 | 5 |

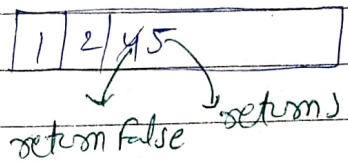
lca = 2

1. Root to node path
2. Last common node
3. Root to

① Root to Node path

getPath (root, node, ArrayList path)





② loop (i=0 to path1.length & path2.length)

Code:-

```
public static Node lca (Node root, int n1, int n2){
                        Node
ArrayList <Integer> path1 = new ArrayList <>();
ArrayList <Integer> path2 = new ArrayList<>();
                    Node

getPath(root, n1, path1);
getPath (root, n2, path2);
// last common ancestor
int i=0;
for(; i< path1.size() && i<path2.size(); i++){
        if( path1.get(i) != path2.get(i)){
                break;
        }
}
Node lca = path1.get (i-1); // last equal node i-1th

return lca;
}
```

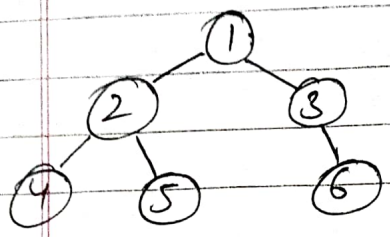
PAGE NO.
DATE: / /

```
public static boolean getPath(Node root, int n, ArrayList<Node> path){
    if(root == null) return false;
    path.add(root);

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

    boolean foundLeft = getPath(root.left, n, path);

    boolean foundRight = getPath(root.right, n, path);

    if(foundLeft || foundRight){
        return true;
    }
    path.remove(path.size()-1);
    return false;
}
```

Time complexity : O(n)

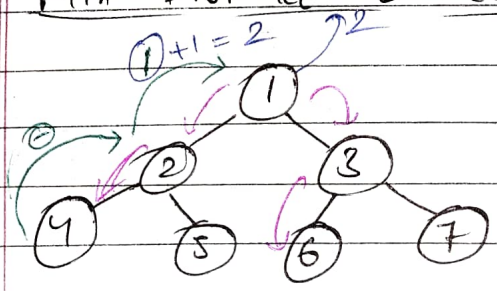
PAGE NO.
DATE: / /

Approach2: n1=4, n2=6




```
public static Node lac2(Node root, int n1, int n2){

    if(root.data == n1 || root.data == n2) return root;
      root == null ||
    lca2
    Node leftLca = lac2(root.left, n1, n2);

    Node rightLca = lca2(root.right, n1, n2);

    if(rightLca == null) return leftLca;
    if(leftLca == null) return rightLca;

    return root;
}
```

Q. Min Distance between nodes



n1=4 n2=6

① dist1 lca to n1
② dist2 lca to n2

dist1 + dist2 = min dist between n1 & n2.

```
if(root == null)
    return -1
if(root.data = n)
    return 0;
```

left n search
right n search

valid value (>-1) → value+1

```
public static int mmDist(Node root, int n1, int n2){
    Node lca = lca(root, n1, n2);
    lcaDist(root, n1);
    int dist1 = lcaDist(lca, n1);
    int dist2 = lcaDist(lca, n2);
    return dist1 + dist2;
}

public static int lcaDist(Node root, int n){
    if(root == null){ return -1; }
    if(root.data == n){ return 0; }
    int leftDist = lcaDist(root.left, n);
    int rightDist = lcaDist(root.right, n);
    if(leftDist == -1 && rightDist == -1){ return -
    else if(leftDist == -1) return rightDist+1;
    else return leftDist+1;
}
```