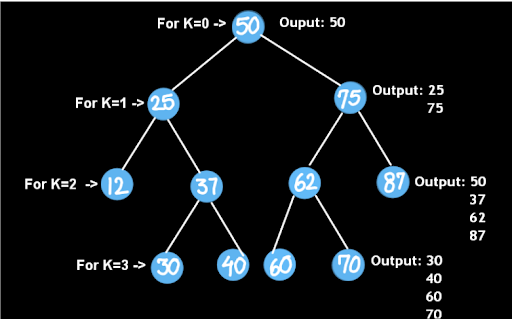
**1. Problem Discussion :**

In this problem you are given a partially written BinaryTree class. You are given a value k. All you have to do is to complete the body of the 'printKLevelsDown' function. The function is expected to print in different lines all nodes which are k level deep. Use preorder for printing. Input and Output is managed for you. Let's understand this using an example:

****

If you are finding it difficult to understand then we recommend you to watch the question video of this topic. I am sure that question will get much clearer once you watch it.

For more clarity of the question, watch the question video

Play Video

**2. Approach :**

The problem here deals with printing the nodes which are at a distance k away from the root node in a preorder fashion. We can easily solve this problem using a few recursive calls.

To achieve this we will use preorder traversal on the binary tree, as it is also specifically informed in the question.

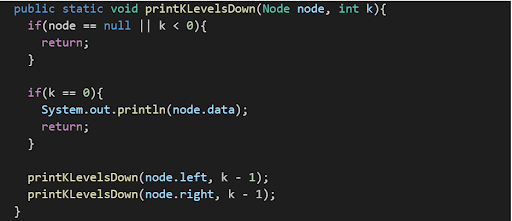
To start with, for every node we will check whether the K is 0 or has been reduced to 0, if yes then we would print the node and return, as now there is no point going to levels below K.

If not, then we would recursively call for the left subtree and right subtree which are a level down (K-1).

Moreover, our base case will handle cases when we hit a null node or if K becomes less than 0, then in this case we will simply return from function calls.

Passing K as a function argument helps us to check the distance of the current node from the root node without calling any helper function to calculate the distance between two nodes, this ensures that our algorithm works in linear time complexity.

Let's try to code this:

****

Yes, that looks perfect!!

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

public static class Node {

int data;

Node left;

Node right;

Node(int data, Node left, Node right) {

this.data = data;

this.left = left;

this.right = right;

}

}

public static class Pair {

Node node;

int state;

Pair(Node node, int state) {

this.node = node;

this.state = state;

}

}

public static Node construct(Integer[] arr) {

Node root = new Node(arr[0], null, null);

Pair rtp = new Pair(root, 1);

Stack< Pair> st = new Stack< >();

st.push(rtp);

int idx = 0;

while (st.size() > 0) {

Pair top = st.peek();

if (top.state == 1) {

idx++;

if (arr[idx] != null) {

top.node.left = new Node(arr[idx], null, null);

Pair lp = new Pair(top.node.left, 1);

st.push(lp);

} else {

top.node.left = null;

}

top.state++;

} else if (top.state == 2) {

idx++;

if (arr[idx] != null) {

top.node.right = new Node(arr[idx], null, null);

Pair rp = new Pair(top.node.right, 1);

st.push(rp);

} else {

top.node.right = null;

}

top.state++;

} else {

st.pop();

}

}

return root;

}

public static void display(Node node) {

if (node == null) {

return;

}

String str = "";

str += node.left == null ? "." : node.left.data + "";

str += " <- " + node.data + " -> ";

str += node.right == null ? "." : node.right.data + "";

System.out.println(str);

display(node.left);

display(node.right);

}

public static void printKLevelsDown(Node node, int k) {

if (node == null || k < 0) {

return;

}

if (k == 0) {

System.out.println(node.data);

return;

}

printKLevelsDown(node.left, k - 1);

printKLevelsDown(node.right, k - 1);

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int n = Integer.parseInt(br.readLine());

Integer[] arr = new Integer[n];

String[] values = br.readLine().split(" ");

for (int i = 0; i < n; i++) {

if (values[i].equals("n") == false) {

arr[i] = Integer.parseInt(values[i]);

} else {

arr[i] = null;

}

}

int k = Integer.parseInt(br.readLine());

Node root = construct(arr);

printKLevelsDown(root, k);

}

}

**3. Analysis**

Time Complexity:

O(n) The time complexity for the function is linear as tree traversal is involved which is linear in terms of time complexity.

Space Complexity:

O(logn) The space complexity for the function is proportional to the height of the tree due to the recursion stack.