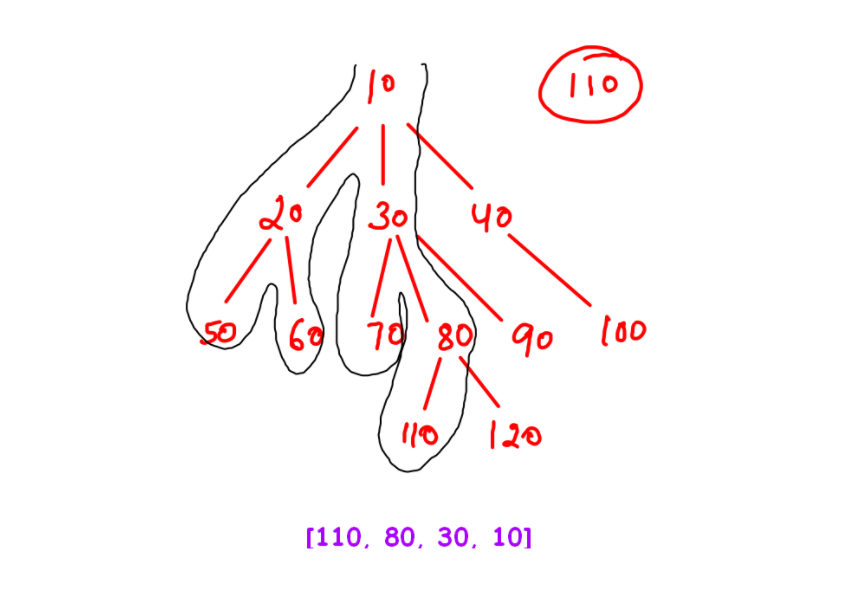
**1. Problem Discussion :**

You are given an input data of a tree and a value of a node. You are required to complete the body of the nodeToRootPath function. The function is expected to return an Arraylist containing the path from element to root, if the element with data is found. Say, you are given a tree as shown in figure 1 and a value 110. You have to return the path from 110 to root i.e. [110, 80, 30, 10].

****

**2. Approach :**

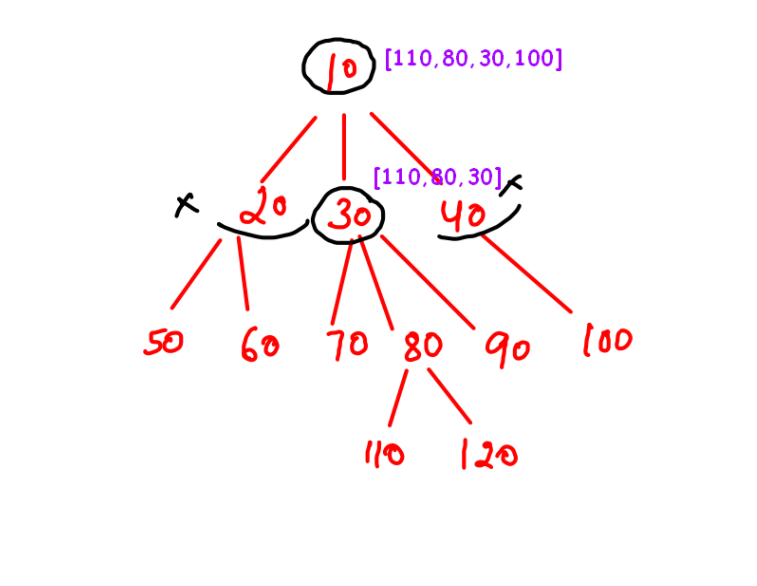
We go through 3 steps of High Level Thinking.

Set Expectation :

We expect that if we input a generic tree and a value of node, then the output should be an ArrayList which returns the path from the given node to the root of the generic tree.

Build Faith :

We must have faith that if our code can give us a path till the root of generic tree, then it can definitely give us a path from the given value to the root of the sub tree.

****

In figure 2, the path from 110 to the subtree with a root 30 is [110, 80, 30]. You just need to believe. Don't focus on "HOW" that will happen.

Expectation meets Faith :

Now to meet our expectation from the faith we add the root of the generic tree i.e. 10 to the path [110,80,30] which will give us an output of [110, 80, 30, 10].

Since this question is similar to the previous question, we are confident that you'll be able to figure out its code. If you are not able to then there is no need to get disheartened, just check the code given below and its discussion.

**3. CODE:**

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

private static class Node {

int data;

ArrayList< Node> children = new ArrayList< >();

}

public static void display(Node node) {

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

for (Node child : node.children) {

str += child.data + ", ";

}

str += ".";

System.out.println(str);

for (Node child : node.children) {

display(child);

}

}

public static Node construct(int[] arr) {

Node root = null;

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

for (int i = 0; i < arr.length; i++) {

if (arr[i] == -1) {

st.pop();

} else {

Node t = new Node();

t.data = arr[i];

if (st.size() > 0) {

st.peek().children.add(t);

} else {

root = t;

}

st.push(t);

}

}

return root;

}

public static ArrayList< Integer> nodeToRootPath(Node node, int data) {

if (node.data == data) { //1

ArrayList < Integer> list = new ArrayList< >();

list.add(node.data);

return list;

}

for (Node child : node.children) {

ArrayList< Integer> ptc = nodeToRootPath(child, data); //2

if (ptc.size() > 0) { //3

ptc.add(node.data);

return ptc;

}

}

return new ArrayList< >(); //4

}

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

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

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

int[] arr = new int[n];

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

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

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

}

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

Node root = construct(arr);

ArrayList< Integer> path = nodeToRootPath(root, data);

System.out.println(path);

// display(root);

}

}

CODE DISCUSSION

● Base Case: If the current node data is equal to the given data, then we simply add that node data to a new list and return it.

● If the current node data is not equal to the given data then we check whether any of the children of the current node can find the required path by recursively calling the function. This answer is received in an ArrayList "ptc"(path till child).

● If size of ptc>0 i.e. a path is found, then we add our node to ptc and return this path. In our example, if we find the path till 30 i.e. [110, 80, 30] then we add our parent node 10 to this path and return it.

● If a case arises that neither the data was equal to the current node nor it was found in the families of its children, then that value doesn't exist and an empty list is returned.

For example, had 150 been the given value, then neither is it equal to 10 nor is it found in the families of 20, 30 or 40, so an empty list is returned.

**4. 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 given tree due to the recursion stack. With this we conclude our question. We want you to check out the solution video for this question to understand it even better.