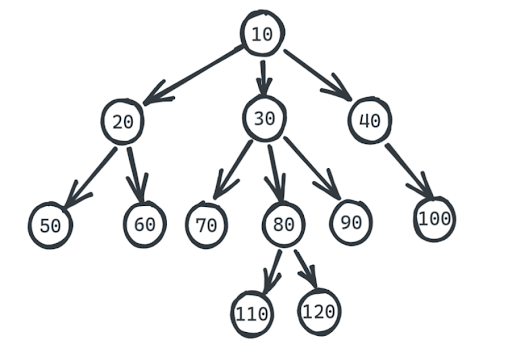
**1. Problem Discussion :**

1• You are given a partially written GenericTree class.

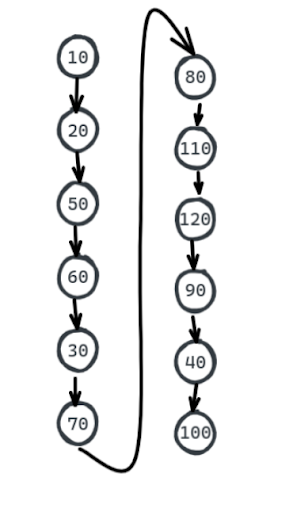
2• You are required to complete the body of linearize function. The function is expected to create a linear tree i.e. every node will have a single child only. For details check the question video.

**2. Approach:**

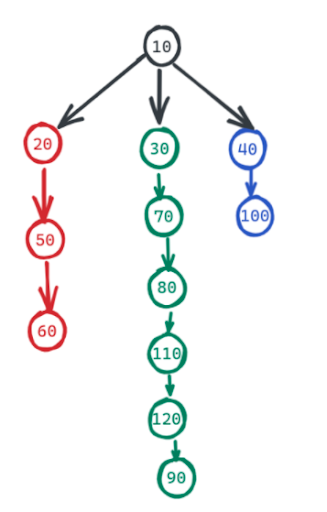
If we have a tree-like this:

****

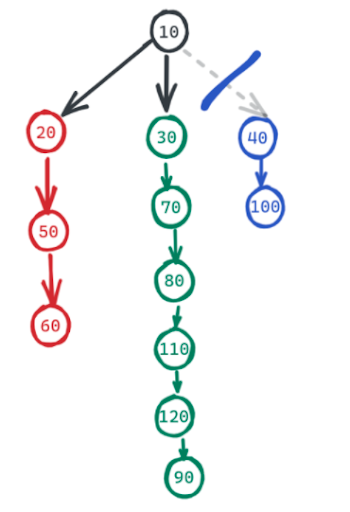
After linearizing it should become:

****

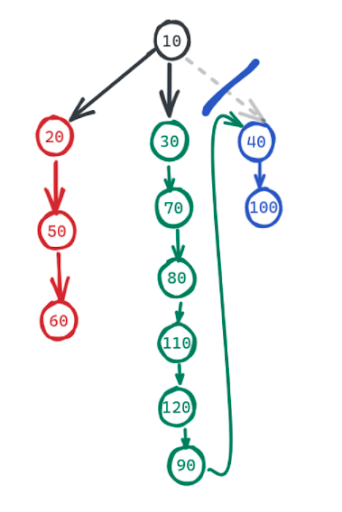
As we discussed previously, Generic trees problems can be solved with Recursion in many situations. Here too we will use a recursive approach. First, let's make an assumption for the Faith. We have Faith that 20, 30, and 40 i.e children of 10 know how to be linear. That is when I called linearize on root 10. it's children 20, 30 and 40 were already linearised. Look at the diagram below to understand better.

****

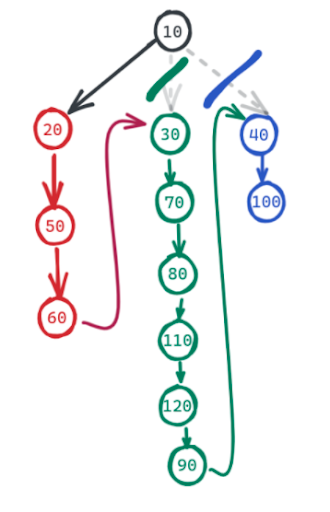
So this is our Faith. Now, what is our expectation? To linearize the root 10. And how do we do that? Look carefully. First, we will cut off 10's rightmost child. Hence there is no longer a connection between 10 and 40.

****

Now we will make 90's next as 40. What is 90? The second last child's tail.

****

Now we will keep doing it as long as 10 don't have a single child. Now we will disconnect 10 and 30, and make 60's next as 30.

****

Now 10 has only one child 20 and also we can see the tree is linearized. So, let's look at the corresponding pseudocode:

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 int size(Node node) {

int s = 0;

for (Node child : node.children) {

s += size(child);

}

s += 1;

return s;

}

public static int max(Node node) {

int m = Integer.MIN\_VALUE;

for (Node child : node.children) {

int cm = max(child);

m = Math.max(m, cm);

}

m = Math.max(m, node.data);

return m;

}

public static int height(Node node) {

int h = -1;

for (Node child : node.children) {

int ch = height(child);

h = Math.max(h, ch);

}

h += 1;

return h;

}

public static void traversals(Node node) {

System.out.println("Node Pre " + node.data);

for (Node child : node.children) {

System.out.println("Edge Pre " + node.data + "--" + child.data);

traversals(child);

System.out.println("Edge Post " + node.data + "--" + child.data);

}

System.out.println("Node Post " + node.data);

}

public static void levelOrderLinewiseZZ(Node node) {

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

stack.add(node);

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

int level = 0;

while (stack.size() > 0) {

node = stack.pop();

System.out.print(node.data + " ");

if (level % 2 == 0) {

for (int i = 0; i < node.children.size(); i++) {

Node child = node.children.get(i);

cstack.push(child);

}

} else {

for (int i = node.children.size() - 1; i >= 0; i--) {

Node child = node.children.get(i);

cstack.push(child);

}

}

if (stack.size() == 0) {

stack = cstack;

cstack = new Stack< >();

level++;

System.out.println();

}

}

}

public static void mirror(Node node) {

for (Node child : node.children) {

mirror(child);

}

Collections.reverse(node.children);

}

public static void removeLeaves(Node node) {

for (int i = node.children.size() - 1; i >= 0; i--) {

Node child = node.children.get(i);

if (child.children.size() == 0) {

node.children.remove(i);

}

}

for (Node child : node.children) {

removeLeaves(child);

}

}

private static Node getTail(Node node) {

while (node.children.size() == 1) {

node = node.children.get(0);

}

return node;

}

public static void linearize(Node node) {

for (Node child : node.children) {

linearize(child);

}

while (node.children.size() > 1) {

Node lc = node.children.remove(node.children.size() - 1);

Node sl = node.children.get(node.children.size() - 1);

Node slt = getTail(sl);

slt.children.add(lc);

}

}

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]);

}

Node root = construct(arr);

linearize(root);

display(root);

}

}

**3. Analysis:**

1• Time Complexity: O(n2)

We have visited every node to linearize it. Although the leaf nodes do not get linearized, still we have visited them and so, we have visited n nodes. Also, when we visit them and try to linearize them, we visit all the nodes after linearizing in order to find the tail and add the next node in the pre-order to its children's ArrayList. This happens inside the first loop of traversal. So, in a way- we have a nested loop where we are visiting almost n elements every time. So, the time complexity is O(n2).

2• Space Complexity: O(1)

The space complexity remains O(1) since we haven't used any extra space.