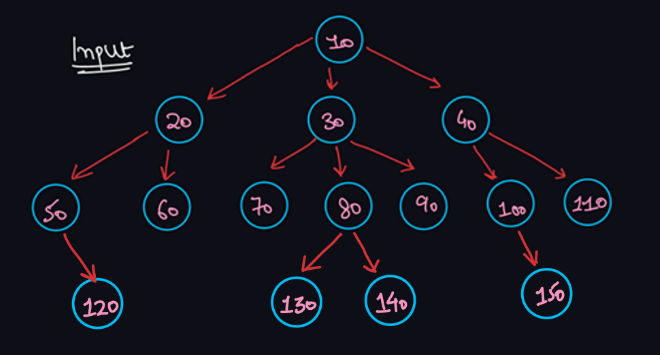
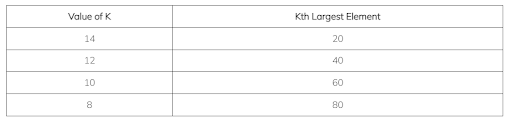
**1. Problem Statement:**

1. You are given a partially written GenericTree class. 2. You are given a number k. You are required to find and print the kth largest value in the tree. 3. Input and Output is managed for you.

Note: If you have not tried enough to come up with logic, then we recommend you to first spend an hour or so doing it, else read only the logic used, take it as a hint and try the problem again with the same logic.

****

****

**2. Approach:**

Before reading the solution, if you do not know how to find the ceil and floor of a number in a generic tree, then please solve this problem first. Try to think, how can we apply the concept of ceil (or floor) in this problem. If we want the kth largest element, then what we can do is first find the largest element of the entire tree by calling ceilAndFloor() with data = Integer.MAX\_VALUE and the current floor = Integer.MIN\_VALUE initially. We will get the largest element in this floor variable. Now, we will take the floor of the previous largest element to get the largest element among the smaller elements. Thus, finding the floor of the largest element will give us the 2nd largest element, floor of 2nd largest element will give us 3rd largest element, and so on. Floor of Mth Largest Element = (M + 1)th Largest Element Hence, we will call for ceilAndFloor() function, (which we have already coded in previous problem), for k times, each time updating data as the previous floor and making the current floor back to Integer.MIN\_VALUE.

**3. Pseudo Code:**

Declare a static integer variable floor in the global scope (inside the Main class). Declare a local integer variable data and initialize it to Integer.MAX\_VALUE. Also, update floor as Integer.MIN\_VALUE. Run a for loop for k times: Call the ceilAndFloor(node, data) function. Update data as the current floor. Make the floor back to Integer.MIN\_VALUE (for the next iteration). Return the kth largest element stored in data.

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;

}

static int floor;

public static void ceilAndFloor(Node node, int data) {

if (node.data < data) {

if (node.data > floor) {

floor = node.data;

}

}

for (Node child : node.children) {

ceilAndFloor(child, data);

}

}

public static int kthLargest(Node node, int k) {

int data = Integer.MAX\_VALUE;

floor = Integer.MIN\_VALUE;

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

ceilAndFloor(node, data);

data = floor;

floor = Integer.MIN\_VALUE;

}

return data;

}

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

Node root = construct(arr);

int kthLargest = kthLargest(root, k);

System.out.println(kthLargest);

}

}

This code is written and explained by our team in the solution video. Do check it out to understand the concept completely.

For more clarity of the question, watch the question video

Play Video

**4. Analysis:**

Time Complexity:

This approach is a naive algorithm. We are traversing the entire tree k times, by calling ceilAndFloor() for k times. Hence the total time complexity will be O(n \* k) where n = number of nodes in the tree.

Space Complexity:

We are not using any auxiliary data structure, hence O(1) extra space is used. However, since we are using recursion, the recursion call stack may have O(d) space where d = maximum depth of the tree.

**5. Follow Up:**

Similar to Kth Largest Element, try to find Kth Smallest Element in the generic tree, by replacing the logic of floor with ceil. The approach discussed has very poor time complexity. Once, you will learn about heap/priority queue, then come back to this problem, and try to come up with a more optimized solution. Hope that you liked the article on Kth Largest Element in Generic Tree. Subscribe to Pepcoding's youtube channel for more such amazing video content on Data Structures & Algorithms. You can suggest any improvements to the article on our telegram channel, or on the youtube channel's comment section.