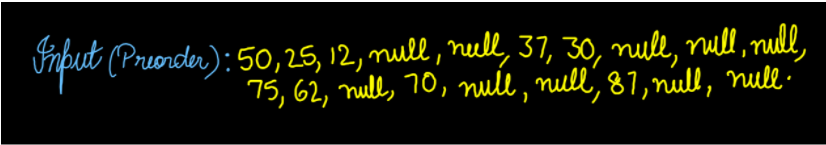
**1. Problem Discussion:**

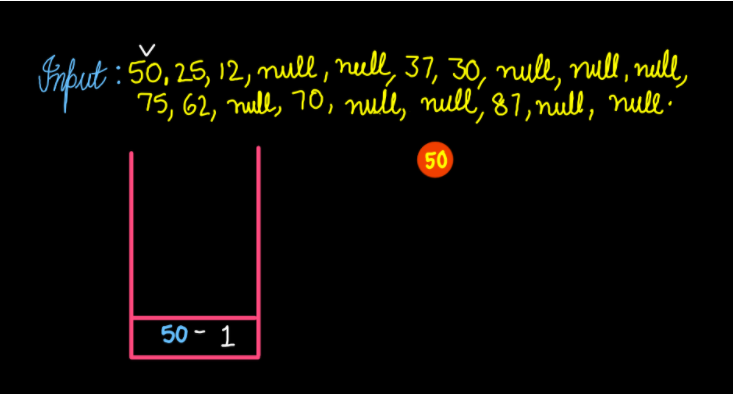
Here you will be provided with the preorder of the binary tree in an array form. All you need to do is to construct a binary tree using this information. Sounds easy!

**2. Approach :**

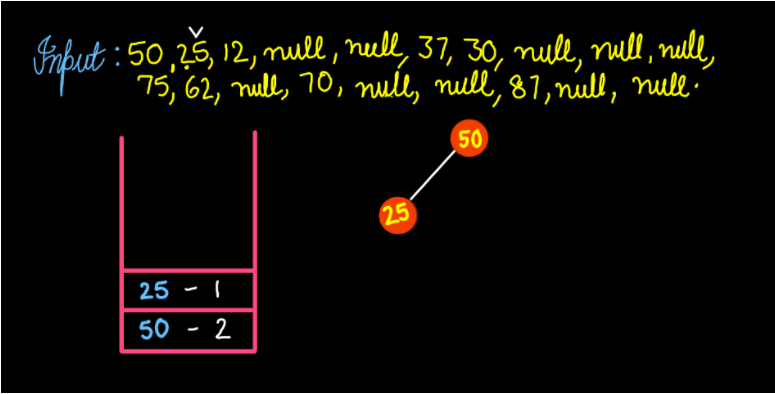
In order to solve this we use a similar trick which we used while solving the Iterative Preorder and Postorder of Generic Tree. Let's discuss it: The algorithm deals with taking a stack and initially we will be pushing (root, 1). To achieve this, we create a class that will be storing the node along with its state (integer data type). The state represents the Euler's tree state at any instance of time. If state == 1 of the node at the top of the stack then it means that the next element of the array will represent the left of the top node. And therefore we will place it at the left of this node. After that we increment the value of state to 2 and also push the new node into the stack with stage initialized to 1. In addition, when the state value is 2, it indicates that the next element of the array is the right of the node at top. And therefore we will place it at the right of this node. After that we increment the value of state to 3 and also push the new node into the stack with stage initialized to 1. Furthermore, when state == 3 this means that it is time to pop the node out of the stack as at this point we have dealt with both right and left child and placed them at their correct position. Now we will iterate over the stack till the stack becomes empty and for every iteration we will consider the state of the top of the stack node and its state will decide the action to be taken as mentioned above. That's all we need to do. Let's take an example and understand this in a better way. Consider the below given preorder as the input.

****

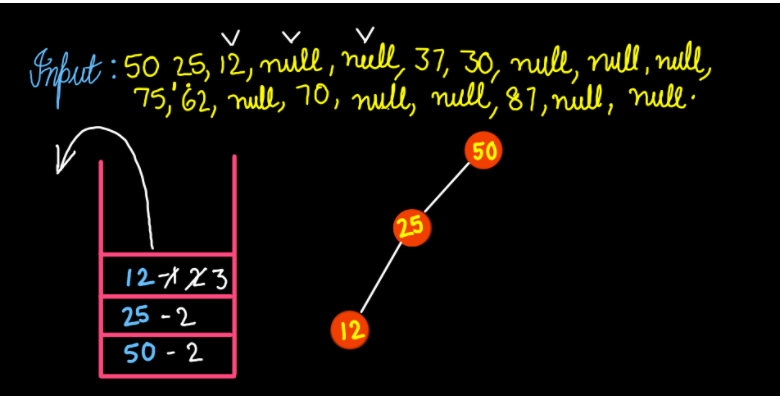
Initially the first node (50) will be pushed into the stack with corresponding value of stage to be 1.

****

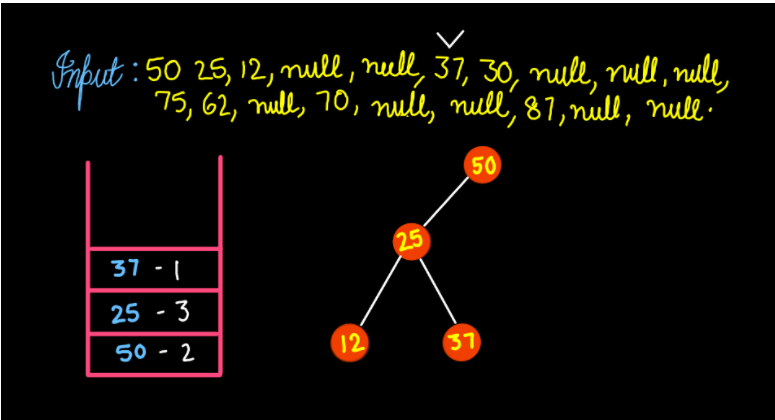
Since the stage of the top most (50) element of the stack is 1, this implies that the next element (25) of the input array will be the left value of the node. So we push this into the stack with stage value 1. And increment the stage value of 50, making it 2.

****

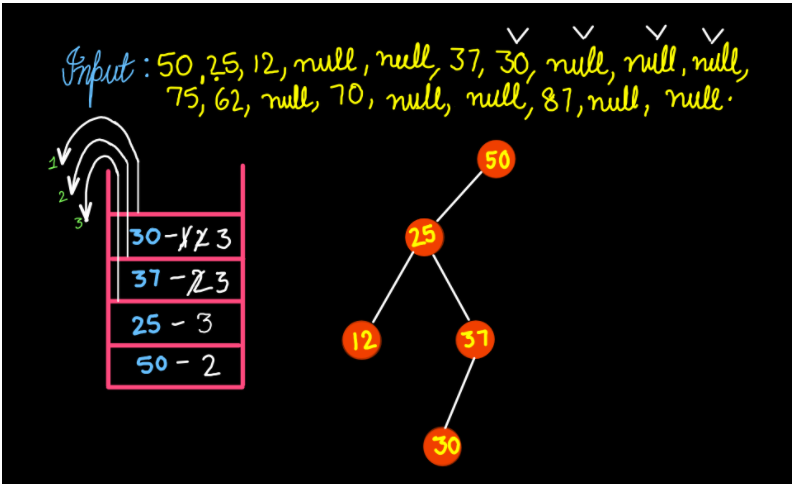
Now, the stage of top most element (25) of stack is 1, this implies that the next element (12) of the input array will be the left value of the node (25). So we push this into the stack with stage value 1. And increment the stage value of 25, making it 2.

****

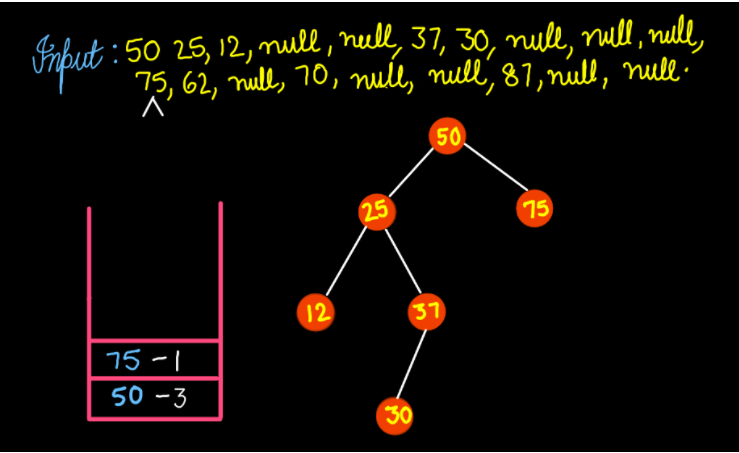
Now, the stage of top most element (12) of stack is 1, this implies that the next element null will be the left of the node (12). Since it is a null we only increment the stage value of 12, making it 2 and we move to the next element. Now, the stage of top most element (12) of stack is 2, this implies that the next element null will be the right of the node (12). Since it is a null we only increment the stage value of 12, making it 3. As the stage of topmost element (12) becomes 3, 12 pops out of the stack and we move to the next element. Now, the stage of top most element (25) of stack is 2, this implies that the next element (37) of the input array will be the right value of the node (25). So we push this into the stack with stage value 1. And increment the stage value of 25, making it 3.

****

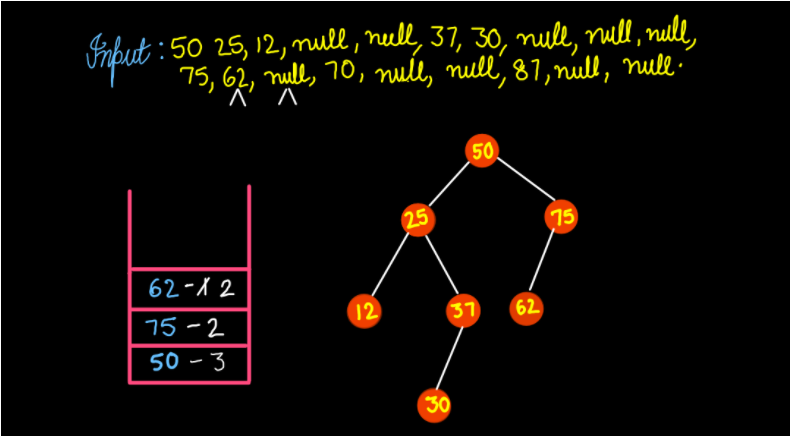
Now, the stage of top most element (37) of stack is 1, this implies that the next element (30) of the input array will be the left value of the node (37). So we push this into the stack with stage value 1. And increment the stage value of 37, making it 2.

****

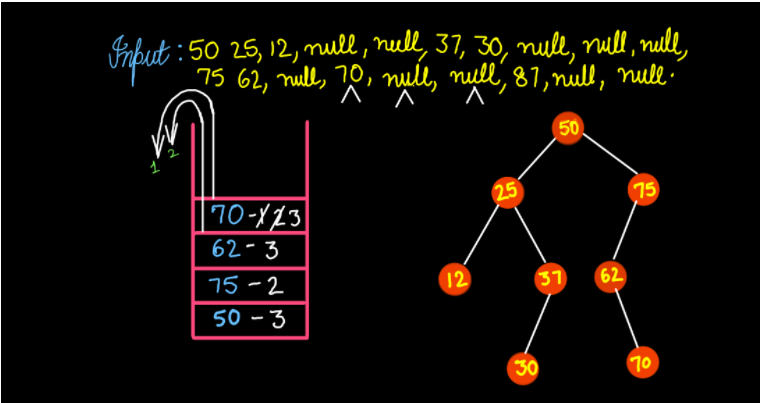
Now, the stage of top most element (30) of stack is 1, this implies that the next element null will be the left of the node (30). Since it is a null we only increment the stage value of 30, making it 2 and we move to the next element. The stage of top most element (30) of stack is 2, this implies that the next element null will be the right of the node (30). Since it is a null we only increment the stage value of 30, making it 3. As the stage of topmost element (30) becomes 3, 30 pops out of the stack. Now, the stage of the top most element (37) of stack is 2, this implies that the next element null will be the right value of the node (37).. And increment the stage value of 37, making it 3. As the stage of topmost element (37) again becomes 3, 37 pops out of the stack. The stage of the top most element (25) of stack is 3, this implies that 25 also pops out of the stack. Now, the stage of the top most (50) element of the stack is 2, this implies that the next element (75) of the input array will be the right value of the node. So we push this into the stack with stage value 1. And increment the stage value of 50, making it 3.

****

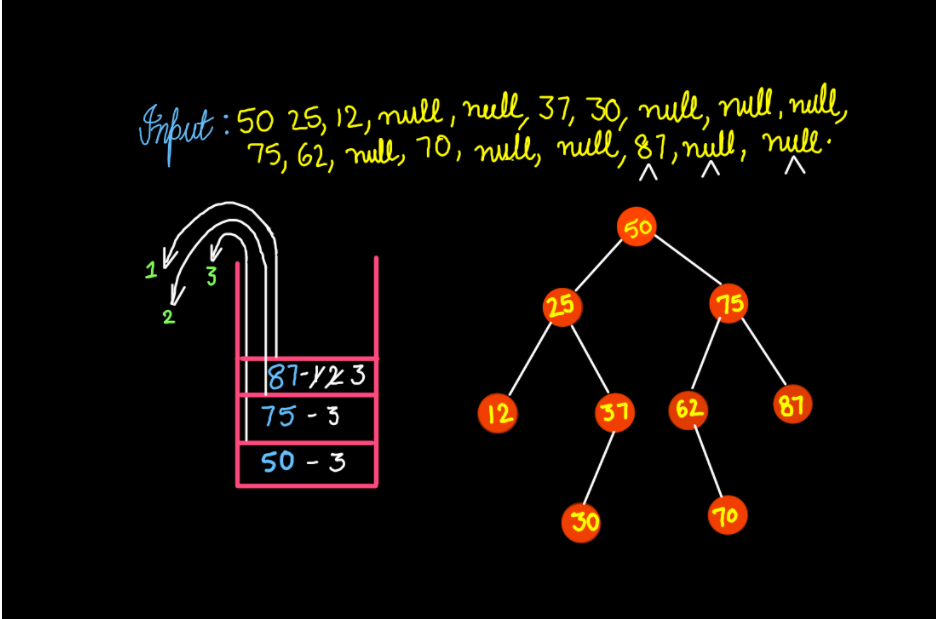
Now, the stage of top most element (75) of stack is 1, this implies that the next element (62) of the input array will be the left value of the node (75). So we push this into the stack with stage value 1. And increment the stage value of 75, making it 2.

****

Now, the stage of top most element (62) of stack is 1, this implies that the next element null will be the left of the node (62). Since it is a null we only increment the stage value of 62, making it 2 and we move to the next element. The stage of the top most element (62) of stack is 2, this implies that the next element 70 of the array will be the right of the node (62). So we push this into the stack with stage value 1. And increment the stage value of 62, making it 3.

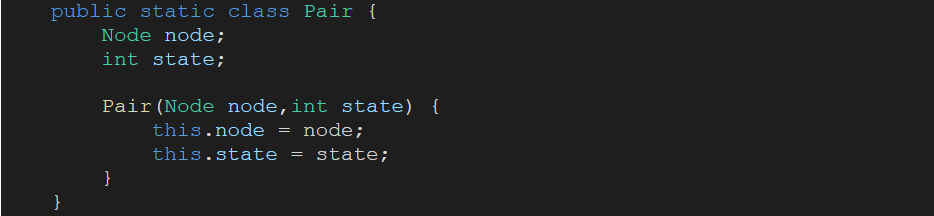
****

Now, the stage of top most element (70) of stack is 1, this implies that the next element null will be the left of the node (70). Since it is a null we only increment the stage value of 70, making it 2 The stage of top most element (70) of stack is 2, this implies that the next element null will be the right of the node (70). Since it is a null we only increment the stage value of 70, making it 3. As the stage of the topmost element (30) becomes 3, 70 pop out of the stack. Now, the stage of top most element (62) of stack is 2, this implies that the next element null will be the right value of the node (62). And increment the stage value of 62, making it 3. As the stage of topmost element (62) also becomes 3, 62 pops out of the stack. The stage of top most element (75) of stack is 2, this implies that the next element (87) of the input array will be the right value of the node (75). So we push this into the stack with stage value 1. And increment the stage value of 75, making it 3.

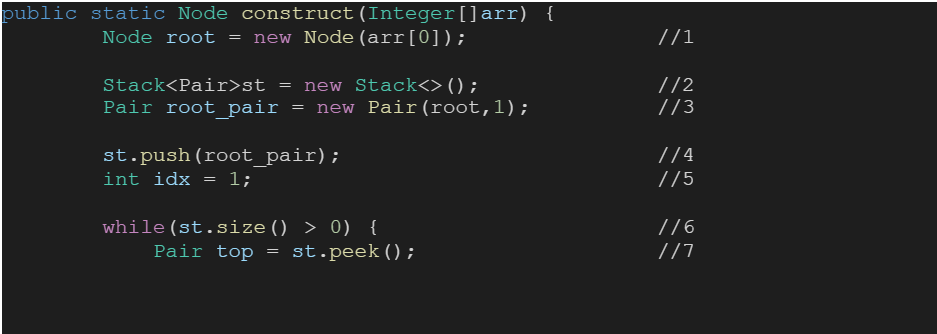
****

Now, the stage of top most element (87) of stack is 1, this implies that the next element null will be the left of the node (87). Since it is a null we only increment the stage value of 87, making it 2. Now, the stage of top most element (87) of stack is 2, this implies that the next element null will be the right of the node (87). Since it is a null we only increment the stage value of 87, making it 3. As the stage of the topmost element (87) becomes 3, 87 pops out of the stack. The stage of the top most element (75) of the stack is 3, this implies that it also pops out of the stack. Also, the stage of the top most element (50) of the stack is 3, this implies that 50 also pops out of the stack.

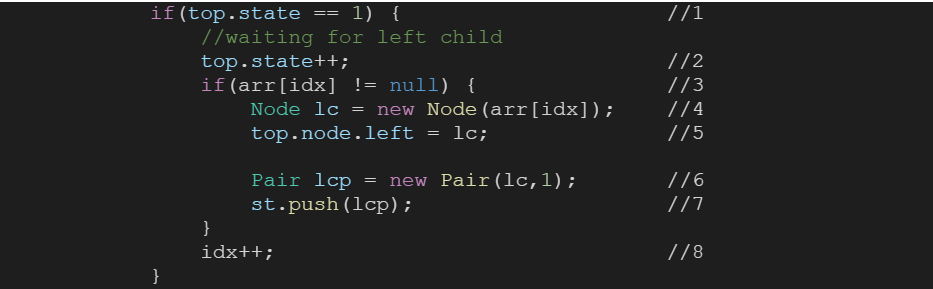
Let's try to code this

****

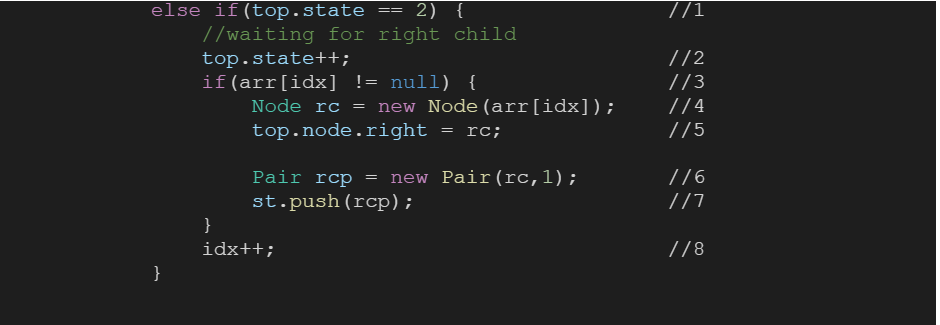
In the above code snippet, a class Pair has been created. It has a Node and int as two of its properties. Then we have defined a constructor with Node and state as its parameters.

****

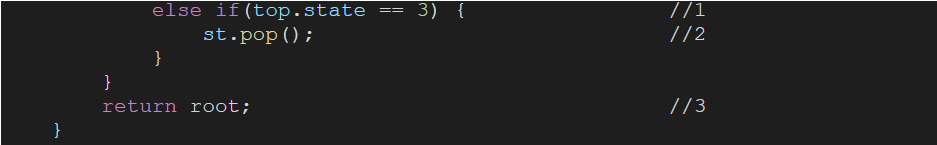
1• A node is defined as the root which stores the value of an array at index 0. Since the array is arranged in preorder, the first value is supposed to be the root's data. 2• Then a stack, of type Pair, is created to carry out a whole and heavy mission. 3• A new pair is defined, root is stored as node and 1 as its stage. 4• Then the pair is pushed in the stack before the real operation starts. 5• An integer named idx is defined and initialized with 1, in order to access the input array. 6• Now we run a while loop, until the stack empties out. 7• To start with, we store the top of the stack in a variable top of type pair obviously so that we can access it's information further.

****

1• Moving further we first check, if the stage of Pair top equals to 1. If it is true then further statements get executed otherwise we check the next condition. 2• As soon as control enters the if condition, we increment the top.state. 3• Then we check whether arr(idx) equals null or not. 4• If it doesn't equal null then we firstly define a new Node lc (left child) and pass arr(idx) as it's data value. 5• After that we set the Node lc as the left of Node node. 6• Furthermore, we define a pair lcp (left child pair) and set lc as it's Node and 1 as its stage. 7• Then we push this pair into the stack. 8• And finally increment idx.

****

1• Moving further, if the initial 'if' condition wasn't true then we now check if the stage of Pair top equals to 2. If it is true then further statements get executed otherwise we skip this and jump to the next part. 2• As soon as control enters the if condition, we increment the top.state. 3• Then we check whether arr(idx) equals null or not. 4• If it doesn't equal null then we firstly define a new Node rc (right child) and pass arr(idx) as it's data value. 5• After that we set the Node rc as the right of Node node. 6• Furthermore, we define a pair rcp (right child pair) and set rc as it's Node and 1 as its stage. 7• Then we push this pair into the stack. 8• And finally increment idx.

****

1• If none of the above mentioned conditions was true then we check if stage of top equals 3. 2• If the condition is true then we pop out the top of the stack. 3• After coming out of the while loop we return the root.

ConsoleJava

import java.util.\*;

public class Main {

public static class Node {

int data;

Node left;

Node right;

Node() {

}

Node(int data) {

this.data = data;

}

}

public static class Pair {

Node node;

int state;

Pair() {

}

Pair(Node node, int state) {

this.node = node;

this.state = state;

}

}

public static Node construct(Integer[]arr) {

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

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

Pair root\_pair = new Pair(root, 1);

st.push(root\_pair);

int idx = 1;

while (st.size() > 0) {

Pair top = st.peek();

if (top.state == 1) {

//waiting for left child

top.state++;

if (arr[idx] != null) {

Node lc = new Node(arr[idx]);

top.node.left = lc;

Pair lcp = new Pair(lc, 1);

st.push(lcp);

}

idx++;

}

else if (top.state == 2) {

//waiting for right child

top.state++;

if (arr[idx] != null) {

Node rc = new Node(arr[idx]);

top.node.right = rc;

Pair rcp = new Pair(rc, 1);

st.push(rcp);

}

idx++;

}

else if (top.state == 3) {

st.pop();

}

}

return root;

}

public static void display(Node node) {

if (node == null) {

return;

}

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

String left = (node.left == null) ? "." : "" + node.left.data;

String right = (node.right == null) ? "." : "" + node.right.data;

str = left + str + right;

System.out.println(str);

display(node.left);

display(node.right);

}

public static void main(String[]args) {

Integer[]arr = {50, 25, 12, null, null, 37, 30, null, null, null, 75, 62, null, 70, null, null, 87, null, null};

Node root = construct(arr);

}

}

**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. Also each element is pushed and popped out only once.

Space Complexity:

O(n) The space complexity for the function is proportional to the space used by stack, which can be O(n) at max.//doubt We hope that this article was helpful. If somehow you are finding it difficult to understand this then we advise you to watch the video lecture.