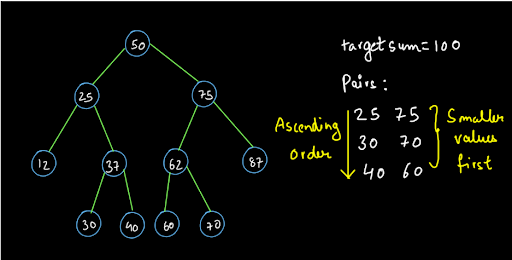
**1. Problem Discussion :**

****

So, you already know the basic approach. Now, try to solve this using some other approach. If you remember, the time complexity of the previous approach that we have studied is O(n x h) where h is the height of the tree which is in average case O(log2n) which makes the time complexity as O(nlog2n) and the space complexity is O(h) which on average case is also O(log2n). Try to make the approach more efficient. Hint: Try to maintain the same space complexity and reduce the average case time complexity to O(n).

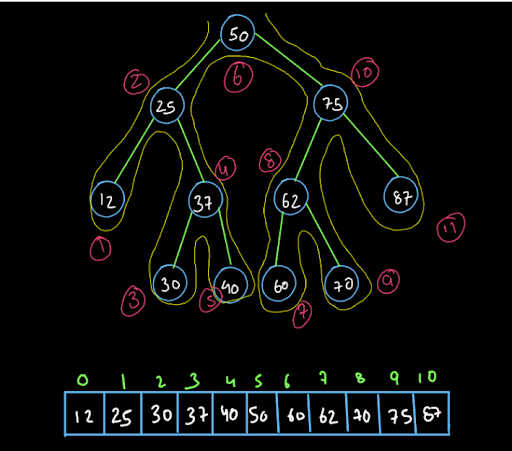
**2. Approach :**

ArrayList Filling Approach

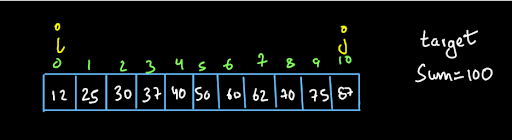
Time Complexity: O(n) Space Complexity: O(n) Let us just recap a little bit about the previous method. We were traversing the tree and we were doing the work in the inorder of the tree. We calculated a complement value for every value in the tree and searched for it. If we found the complement value also, then we printed the pair.

Now, instead of searching for a complement value we will just traverse the tree in inorder and we will fill the values of the nodes that appear during the inorder traversal into an arraylist. Why are we doing this? You must know the property of inorder traversal of a binary search tree. This property states that when we visit a tree in its inorder traversal, we get all the values in a sorted order. So, storing the values in an arraylist by traversing the tree in inorder will give us an arraylist which will contain all the node values of the tree in a sorted order. You may refer to the diagram.

We have traced the Euler path and also mentioned the sequence in which the nodes appear in the inorder on the nodes itself. After the complete traversal, we will get an arraylist as shown in the diagram.

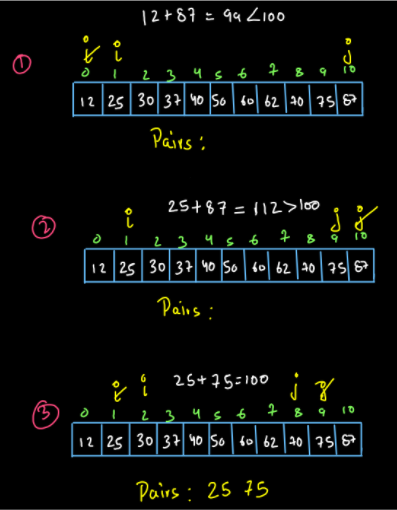
****

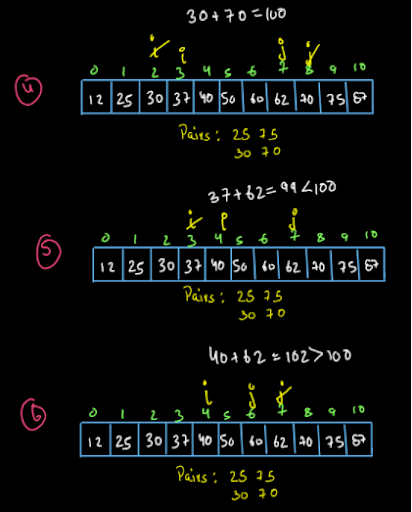
Okay, so we discussed that we filled the array list while traversing the tree in the inorder traversal. But, we have still not discussed the reason for filling the values in the arraylist. Well, the reason is simple. We have to find the target sum pairs from the given BST. It is easier and more efficient to find the target sum pair from a sorted array as compared to a BST. Why? Think about this!!! So, let's now discuss the method to find the target sum pairs from this sorted arraylist. Have a look at the diagram given below:

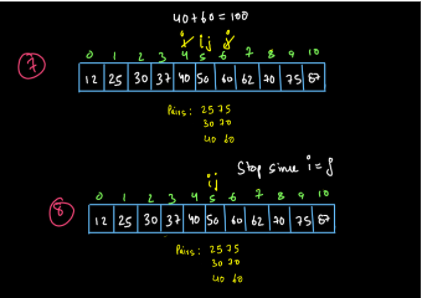
****

We have kept a variable 'i' at the beginning of the arraylist i.e. i=0 a and j is at size-1. From here we will follow this algorithm till i< j: If the sum is less than the target sum (100), increment the value of 'i' . This is because the smaller value is so small that we are not able to complete the sum. If the sum is greater than the target sum (100), decrement the value of 'j'. This is because the larger number (arr[j]) is probably too large that the sum exceeds the required sum or the target sum. If the sum is equal to the target sum, print the smaller value with a space and then print the larger value and leave a line. Also, increment i and decrement j. We have to stop when i becomes greater than or equal to j. We cannot have i=j as we have to print only unique pairs. We cannot take a node twice. For example to find the target sum 100, we cannot take a node with value 50 twice in our sum.

We would have printed all the pairs till then. Have a look at the diagrams given below:

****

****

****

So, dear reader, did you understand this algorithm? You may refer to the solution video to understand the concept till here. Now that we have understood the procedure, let us code the algorithm:

**3. Code Implementation :**

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 tnf(Node node, ArrayList< Integer> list )

{

if (node == null)

{

return;

}

tnf(node.left, list);

list.add(node.data);

tnf(node.right, list);

}

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

Node root = construct(arr);

// write your code here

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

tnf(root, list);

int li = 0;

int ri = list.size() - 1;

while (li < ri)

{

if (list.get(li) + list.get(ri) > data)

{

ri--;

}

else if (list.get(li) + list.get(ri) < data)

{

li++;

}

else

{

System.out.println(list.get(li) + " " + list.get(ri));

li++;

ri--;

}

}

}

}

**4. Analysis**

Time Complexity (ArrayList Filling Approach):

The time complexity will be O(n) as filling the arraylist by traversing the tree in inorder takes O(n) time and finding the target sum pairs also takes O(n) time (single traversal in the arraylist). So, the time complexity is O(n) + O(n)=O(n).

Space Complexity (ArrayList Filling Approach):

The space complexity of the above approach is O(n) as we have used an arraylist filled with all the nodes (node's data actually) to solve our problem. So, dear reader, we reduced the time complexity from O(n x h) to O(n) but we increased the space complexity from O(h) to O(n). Is there any method by which we can have O(h) space complexity and O(n) time complexity?

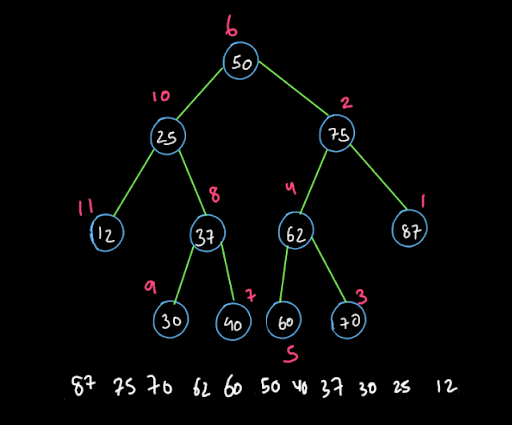
**5. Approach :**

Reverse Inorder Approach

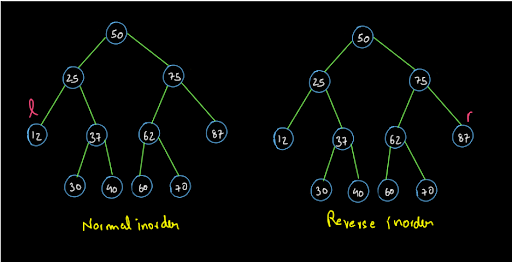
Time Complexity: O(n) Space Complexity: O(h)

From the above approach we have understood one thing that if we can keep one variable on the left end where the values are in increasing order and the other on the right end where we have the largest value, it is easy for us to find the target sum pairs. The problem is that when we store these values and then apply this method, the space complexity becomes O(n). So, if we can get a method to do this without storing the elements, the space complexity will remain O(h) (due to recursion) and the time complexity will become O(n). We know that inorder traversal gives us the values in an increasing order. What if we perform the reverse inorder traversal? The inorder traversal follows the LNR (left-node-right) property. If we apply the reverse inorder traversal i.e. we perform RNL (right-node-left) then we will get the values in a decreasing order.

We recommend you watch the ITERATIVE TREE TRAVERSALS video to understand how we can iteratively perform the inorder traversal of a tree.

****

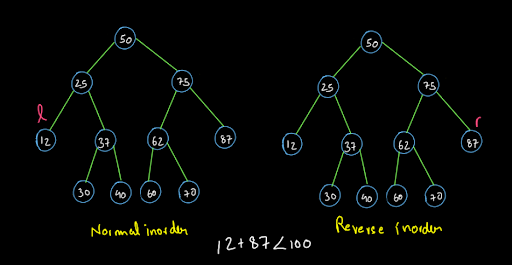
So, this procedure is very simple if we know the iterative inorder traversals of a tree. Have a look at the diagram given below:

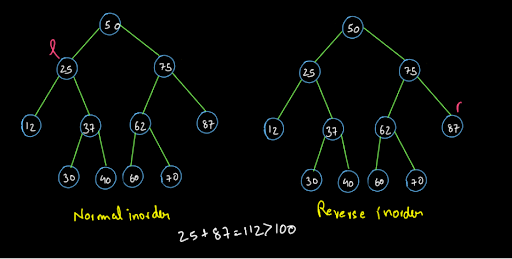
****

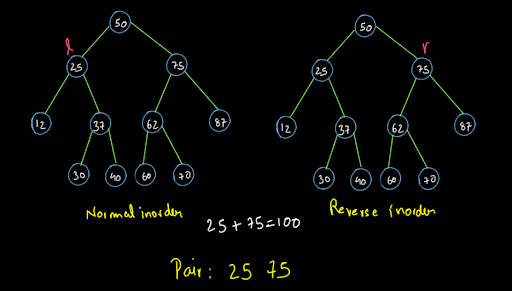
We are performing inorder and reverse inorder on the same tree. We have denoted two trees for the ease of your understanding. Currently, the left which is taken by using the normal inorder is at the lowest element and right which is taken from the reverse inorder is at the highest element. So, we will follow these rules:

1• If left.data + right.data < target sum, we will move forward and take left to the next node in the normal preorder traversal. 2• If left.data + right.data > target sum, we will move forward and take right to the next node in the reverse preorder traversal. 3• If left.data + right..data = target sum, we will print the left.data space separated with right.data, leave a line and make both left and right move to the next node in the normal and reverse inorder respectively.

For instance, if the target sum is 100 then the first three steps are shown below. These three steps involve all the three cases discussed above in the same order:

****

****

****

So, dear reader, we hope that you got what we have to do. You can trace the Euler path for the normal and reverse inorder side-by-side and solve this problem completely yourself now. You know how we solve all the questions in recursion based on a faith? In the same way, we have faith in you that you have studied the iterative inorder traversal method and can write the code for it and reverse iterative inorder yourself. Hence we are not explaining that part here. If you want, you can watch the ITERATIVE TREE TRAVERSALS video and then write the code for the same.

Now, let us write the complete code for this approach:

**6. Code Implementation :**

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 class ITPair

{

Node node;

int state = 0;

ITPair() {};

ITPair(Node node, int state)

{

this.node = node;

this.state = state;

}

}

public static void bestApproach(Node node, int tar)

{

Stack< ITPair> ls = new Stack< >();

Stack< ITPair> rs = new Stack< >();

ls.push(new ITPair(node, 0));

rs.push(new ITPair(node, 0));

Node left = getNextFromNormalInorder(ls);

Node right = getNextFromReverseInorder(rs);

while (left.data < right.data)

{

if (left.data + right.data < tar)

{

left = getNextFromNormalInorder(ls);

}

else if (left.data + right.data > tar)

{

right = getNextFromReverseInorder(rs);

}

else

{

System.out.println(left.data + " " + right.data);

left = getNextFromNormalInorder(ls);

right = getNextFromReverseInorder(rs);

}

}

}

public static Node getNextFromNormalInorder(Stack< ITPair> st)

{

while (st.size() > 0)

{

ITPair top = st.peek();

if (top.state == 0)

{

if (top.node.left != null)

{

st.push(new ITPair(top.node.left, 0));

}

top.state++;

}

else if (top.state == 1)

{

if (top.node.right != null)

{

st.push(new ITPair(top.node.right, 0));

}

top.state++;

return top.node;

}

else

{

st.pop();

}

}

return null;

}

public static Node getNextFromReverseInorder(Stack< ITPair> st)

{

while (st.size() > 0)

{

ITPair top = st.peek();

if (top.state == 0)

{

if (top.node.right != null)

{

st.push(new ITPair(top.node.right, 0));

}

top.state++;

}

else if (top.state == 1)

{

if (top.node.left != null)

{

st.push(new ITPair(top.node.left, 0));

}

top.state++;

return top.node;

}

else

{

st.pop();

}

}

return null;

}

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

Node root = construct(arr);

// write your code here

bestApproach(root, data);

}

}

**7. Analysis**

Time Complexity (Reverse Inorder Approach):

The time complexity of this method is O(n) as we are traversing the tree once for the normal and reverse inorder. So, the traversal for normal inorder is O(n) and for reverse inorder is also O(n). Therefore the time complexity becomes O(n)+O(n)=O(n).

Space Complexity (Reverse Inorder Approach):

The space complexity for this method is O(h) where h is the height of the tree. This is because of recursion stack space only and we have not used any extra space.

So, dear reader, we hope that you understood both the above methods completely. If you have any doubts regarding anything explained above, you may refer to the complete solution video to clear all of them. With this, we have completed the topic of binary search trees also.