***Explanation:***

Dear reader, we recommend you watch the [binary search video](https://www.youtube.com/embed/j6BgMBIOCLQ)and the [binary tree constructor video](https://www.youtube.com/embed/XV1ADVV6FbQ)first before moving on with construction of a BST. We will approach this problem like the way we have been approaching the recursion based problems in binary tree but, we want you to try to think of the code side-by-side as we do the high-level thinking because we have done the same.

So, let's get started.

We are given a sorted array as shown in the figure (see fig-2). We have to construct a BST from it. Remember the **Binary Search Tree Property** that we have talked about a few times?

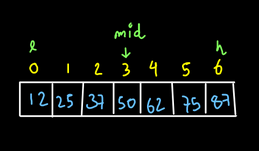
**Binary Search Tree Property:** A tree is said to be a binary search tree if for all the nodes, the left subtree has all the elements smaller in value as compared to the parent node and the right subtree has all the nodes that are greater in value as compared to the parent node.



We will start as we start with the binary search algorithm. We will first find the mid of the array. The low of the array will be the index of the first element of the array i.e. 0. The high will be the index of the last element in the array i.e. n-1 i.e. 6, where n is the size of the array. Mid can be calculated as :

**mid = (low + high)/2**

So, we get the low, mid and high as shown in the figure:

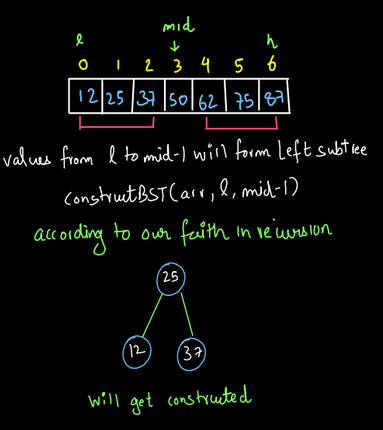


So, we know that the node currently at mid is going to be our first node i.e. the root node. But, we will work in a slightly different way. Let us see our high-level thinking and try to understand it.

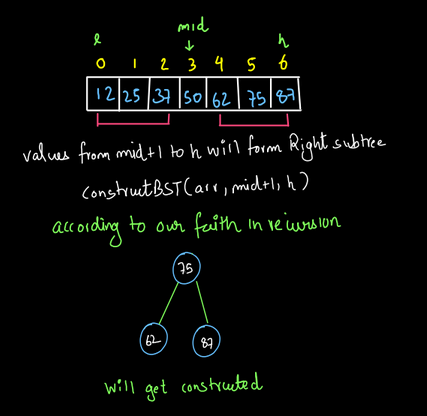
**High-Level Thinking:**

**Expectation:** We expect that when we pass the array as a parameter to the constructBST (array, low, high) function, we will get a BST constructed.

**Faith and Relation:** We have a faith that if we want to construct the left subtree of the BST, then recursion will do it for us if we pass the elements which are to be kept in the left sub-tree. Similarly, if we pass the elements to be kept in the right subtree, we will get the right sub-tree constructed.**Why are we having this kind of faith?** Let's have a look at the diagram given below:



We know that arr[mid] should be the data in the root node and all the elements of the array before it i.e. all the elements from low to mid-1 will be in the left sub-tree. Now, according to our faith in recursion, when we pass these elements which will form the left sub-tree, recursion will be able to construct our left sub-tree. Please dear reader, we request you to have complete faith on recursion here and don't think how this tree will be constructed for now. We will do the low level analysis, to understand the same. Similarly, according to our recursion faith, we will be able to construct the right subtree as well.



We will add one more thing to the faith now. When we get the sub-trees constructed, not only they get constructed but the recursive function also returns the root of these sub-trees to us. So, the left sub-tree returns the node (25) that we have stored in the node left-child.

**Node lchild = constructBST(arr,l,mid-1);**

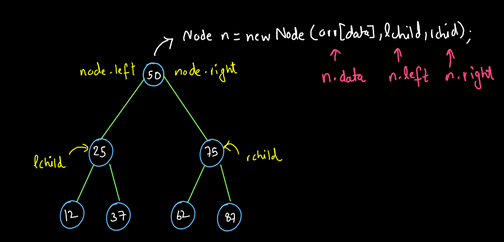
Same will be the case for the right sub-tree. It will return its root i.e. node (75) that we have stored in node right child i.e.

**Node rchild = constructBST(arr,mid+1,h);**

Now that we have the left child and the right child of the root node, we can construct the root node and make its left child as lchild and right child as rchild.

**Node n= new Node (arr[data],lchild,rchild);**

So now the root node is also constructed and its left child and right child have the left and right subtrees which were constructed according to the recursion faith.



So, our tree got constructed completely. Again, please don't think at this moment how the tree got constructed by the recursion call. Just have faith in recursion.

Now that we have done the high-level thinking and simultaneously, we have thought about the code also, let's write the code based on this high-level thinking.

**2. Java Code (Based On High-Level Thinking)**

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 Node construct(int[] arr, int lo, int hi) {

int mid = (lo + hi) / 2;

Node lchild = construct(arr, lo, mid - 1);

Node rchild = construct(arr, mid + 1, hi);

Node node = new Node(arr[mid], lchild, rchild);

return node;

}

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 main(String[] args) throws Exception {

int[] arr = {12, 25, 37, 50, 62, 75, 87};

Node root = construct(arr, 0, arr.length - 1);

display(root);

}

}

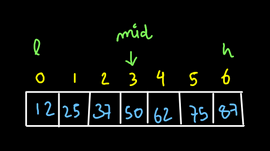
If you have any doubts till here, you may refer to the [solution video](https://www.youtube.com/embed/yGrs-VuPCYg) to clear all your doubts. Now, let us do the low-level analysis of this code so that we get to know the base case and also, how our faith is working.

**3. Low-Level Analysis:**

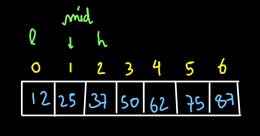
Dear reader, we recommend you keep the code by your side while doing the low-level analysis of this code. So, initially low is at index 0 and high is at index 6 in the given array:



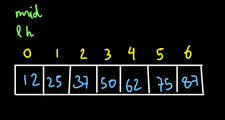
(fig-7)So mid will be calculated as (l+h)/2 and it will be at index 3.



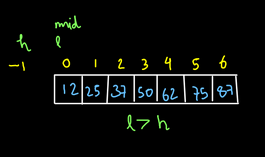
Now according to the code, after calculation of mid, we make a call for the left sub-tree by passing the elements from low to mid-1 in the function.. Now the low is =0 and high=2. So, mid is 1



Again, a call will be made for the left subtree abinary\_search\_tree\_constructor\_nd we will pass the values from low to mid-1. Here, the only value from low to mid -1 is index 0. So now, the low and high, both will be at index 0 and the mid will also be at index 0.

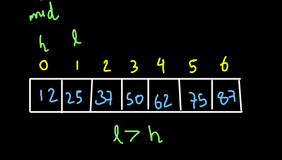


Now, when the call is made to the left sub-tree, low will remain at its place again and high will be mid-1=0-1=-1. So, low became greater than high.



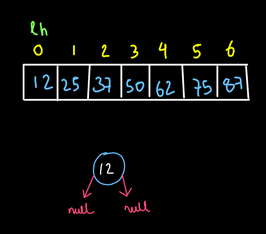
So, this should not happen. This will be our base case. So, when low becomes greater than high, it means that we have reached a leaf node. So, its left child will be null. So, let us return null from here. The returned null is stored in the Node variable lchild. So for now, lchild=null.

Now we are back to the previous state when l=h=0. Here, we have already made a call for the left child. So now, a call will be made for the right child and high will stay at its place but low will become mid+1=1.

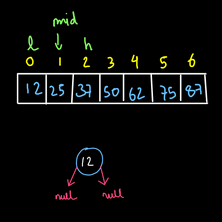


Again, we will return null since low became greater than high. This was the call for the right child. Means now we have rchild=null.

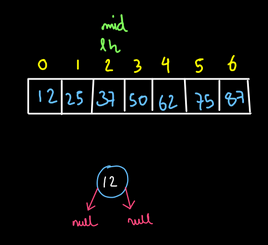
Again, we are back to the state where l=h=mid=0. Now, we have stored both its left and right child. So, we have to just create a new node and set its data as the arr[mid] and the left and right to the left child (lchild) and right child (rchild) respectively. So, we do the same.



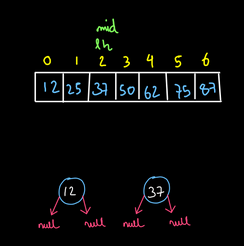
Since all the lines of code have been executed for the case where l=h=0, we will now move back to the previous state in recursion i.e. low=0 and high=2.



The left call for this case was already made. So, now we make the right call i.e. low becomes mid+1. So now, low, high and mid will all be at index 2.



Now, we will call for the left child here. So, high will become mid-1=1 while low is at index 2. Since low became greater than high, we will return null and since this was the left call therefore lchild=null. Similarly when we call for the right subtree l=mid+1 i.e. low becomes 3 while high is at index 2. So, we will return null again. After this when we will be back to the stage when low and high both were at index 2 (see fig-15), both left and right calls have been made. So, we will create a node and assign the data as arr[mid] and its left and right child will be null.



Now, we come back to the stage when l=0 and high=2. For this stage, we have made both the left and right calls. So now, we will make a node with data=arr[mid] and the lchild will become its left and the rchild will become its right (see fig-17).

As you can see, the left subtree of our BST to be constructed is constructed completely. This shows the code is working and the base case is also very simple.

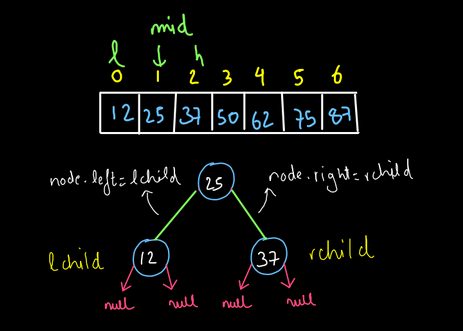
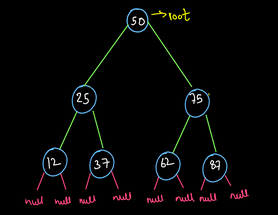


fig-17)So dear reader, we request you to follow this procedure and try to get the complete BST. Your BST should look like this:



We hope that you understood the entire procedure and the base case too. So, let's write the complete code.

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 Node construct(int[] arr, int lo, int hi) {

if (lo > hi)

{

return null;

}

int mid = (lo + hi) / 2;

Node lchild = construct(arr, lo, mid - 1);

Node rchild = construct(arr, mid + 1, hi);

Node node = new Node(arr[mid], lchild, rchild);

return node;

}

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 main(String[] args) throws Exception {

int[] arr = {12, 25, 37, 50, 62, 75, 87};

Node root = construct(arr, 0, arr.length - 1);

display(root);

}

}

**Note:** the display function used in this code is directly taken from Binary Trees display code. You may refer to the [BINARY TREE DISPLAY VIDEO](https://www.youtube.com/embed/sYU6AnSJyjo)to understand this code.

Hope that you have understood the above code completely. If you still have any doubts regarding the procedure, you may refer to the [complete solution video](https://www.youtube.com/embed/yGrs-VuPCYg)to clear all your doubts. Now let us discuss the time and space complexity of this approach.

**4. Analysis**

**TIME COMPLEXITY- O(n)**

The time complexity of this approach is O(n)

**Space Complexity: O(h)**

The space complexity will be O(1) because the array was given input to us and we will not consider its space and we have not used any extra space to create a tree.