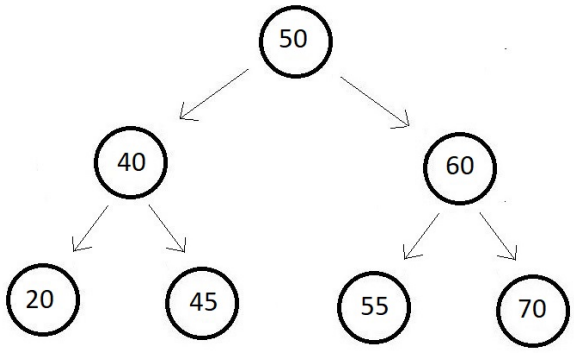
Searching in a Binary Search Trees (Search Operation)

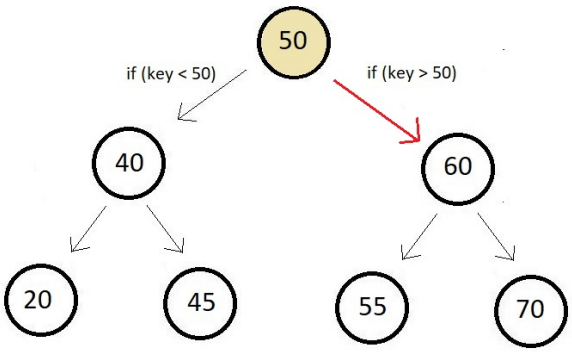
In the last tutorial, we saw the implementation of the *isBST* function in C. We thoroughly discussed the programming. We saw the modifications we had to make in the *inOrder* function to check if the InOrder traversal is in ascending order or not, and in fact, to check if a binary tree is a binary search tree or not. And today, we are going to look after our first binary search tree operation, the **search operation**. We'll see if any given key lies in the binary search tree or not.

Now you might be wondering what exactly is the significance of learning about binary search trees since binary trees were equally good. But let me tell you that one of the major applications of using a binary search tree, is to be able to search some key in the tree in **logn** time complexity in the best case where n is the number of nodes. Each time you compare a node with your key, you divide the search space to its half. But let's not proceed without an example.

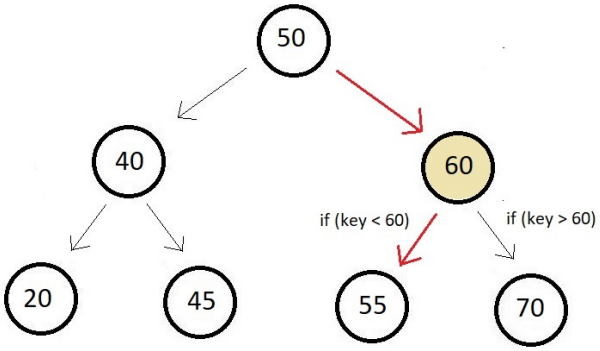
For our purpose of understanding, we will first examine a sample binary search tree. Suppose we have a Binary Search Tree illustrated below.



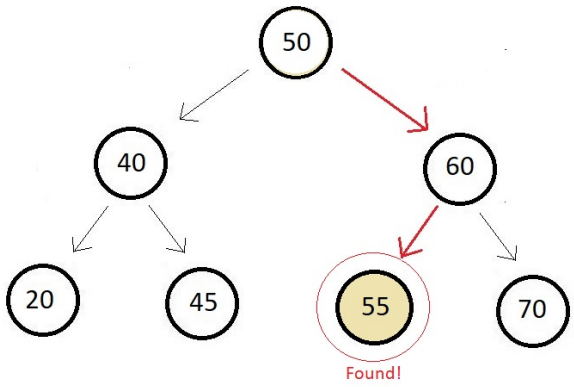
And let's say the key we want to search in this binary search tree is 55. Let's start our search. So, we'll first compare our key with the root node itself, which is 50.



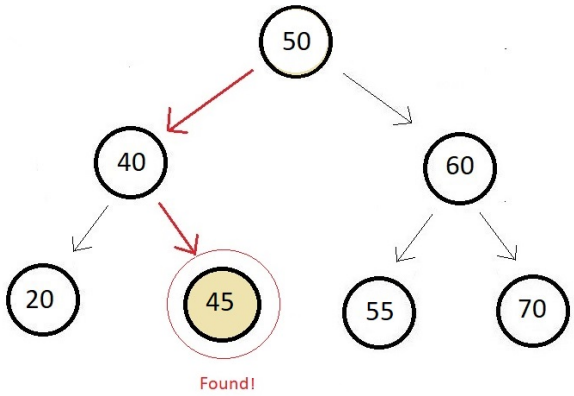
But since 50 is less than 55, which side should we proceed with? Left, or Right? Of course, right. Since all the elements in the right subtree of a node are greater than that node, we'll move to the right. The first element we check our key with is 60.



Now, since our key is smaller than 60, we'll move to the left of the current node. The Left subtree of 60 contains only one element and since that is equal to our key, we revert the positive result that yes, **the key was found**. Had this leaf node been not equal to the key, and since there are no subtrees further, we would have stopped here itself with negative results, saying the key was not found.



Let's see for one more key which is 45. Now, I'll directly illustrate the path we followed and whether it was found or not.



And yes, key 45 was found too. The path we followed is colored red. We went to 50 and found it smaller so moved to its left. Then we found 40, and since our key was greater than that, we moved to its right, and the leaf node we found was equal to 45 only.

Let's analyze the time complexity of the searching algorithm we have discussed above.

Time Complexity of the Search Operation in a Binary Search Tree:

If you remember, we had studied an algorithm called the Binary Search. We could use that algorithm only if the array we were searching for some key in was sorted. And that algorithm had the time complexity of **O(logn)** where n was the length of the array**.** Because we were always dividing our search space into half on the basis of whether our key was smaller or greater than the *mid*. And as you might have guessed, searching in a binary search tree is very much similar to that.

Searching in a binary search tree holds **O(logn)** time complexity in the best case where n is the number of nodes making it incredibly easier to search an element in a binary search tree, and even operations like inserting get relatively easier.

Let's calculate exactly what happens. If you could see the above examples, the algorithm took the number of comparisons equal to the height of the binary search tree, because at each comparison we stepped down the depth by 1. So, the time complexity $T \propto h$, that is, our time complexity is proportional to the height of the tree. Therefore, the time complexity becomes **O(h)**.

Now, if you remember, the height of a tree ranges from logn to n, that is

$$(\log n) \leq h \leq n$$

So, the best-case time complexity is O(logn) and the worst-case time complexity is O(n).

Our next aim would be to automate the searching process. The way we compared the nodes and decided to move to the left or to the right needs to be programmed. So today I'll just brief you all about the way we program this search operation and will implement them in C in the next lecture. We will also write pseudocode for better understanding.

Pseudocode for searching in a Binary Search Tree:

1. There will be a struct node pointer function *search* which will take the pointer to the root node and the key you want to search in the tree. And before you do anything, just check if the root node is not NULL. If it is, return NULL here itself. Otherwise, proceed further.
2. Now, check if the node you are at is the one you were looking for. If it is, return that node. And that would be it. But if that is not the one, just see if that key is greater than or less than that node. If it is less, then return recursively to the left subtree, otherwise to the right subtree. And that is all.

```
Node * search(node* root, key){
  if(root==NULL){
    return NULL;
  }
  if(key==root->data){
    return root;
  }
  else if(key<root->data){
    return search(root->left, key);
  }
  else{
    return search(root->right, key);
  }
}
```

Figure 1: Pseudocode for the search function

We'll see the programming segment in the next lecture in detail. We will make use of our IDEs. And program our own and run them to actually search for a key in a binary search tree.

I appreciate your support throughout. I hope you enjoyed the tutorial. If you genuinely appreciate my work, please let your friends know about this course too. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial where we'll learn to build a program which will search keys in a binary search tree following the algorithm we learnt today. Till then keep coding.