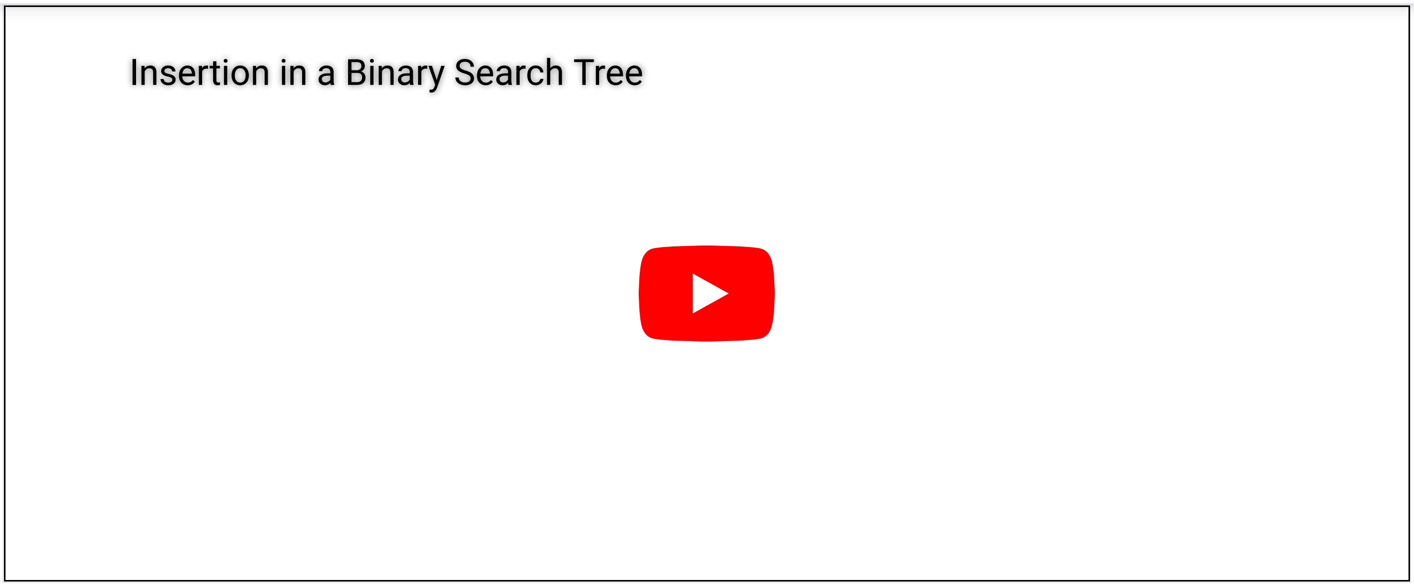
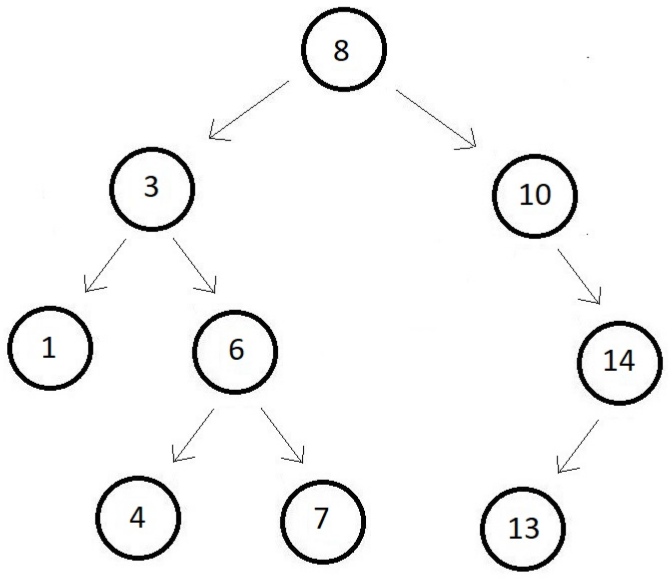


Insertion in a Binary Search Tree

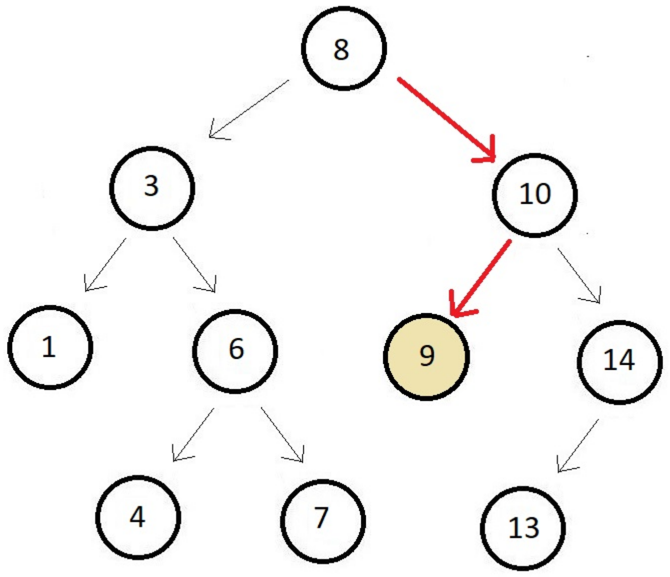
In the last lecture, we learned how to program the search function which will search for a key the user gives in the binary search tree, and revert its position if it is present in the tree, else return NULL. Today we'll learn our second operation in binary search trees which is the insertion in a binary search tree. Inserting in a binary search tree is really not a big deal, but yet very important.
Your half the job is already done if you have diligently followed the search operation. I am saying this because in the algorithm we will follow to insert an element, there are a lot of things in common to the algorithm we learn to search an element. We know a few facts about binary search trees. And I'll only mention the ones we will focus on, while we learn the insertion operation.

1. There are no duplicates in a binary search tree. So, if you could search the element you are being asked to insert, you would return that the number already exists.
2. Now, you would follow what we did in the search operation. Here is an example

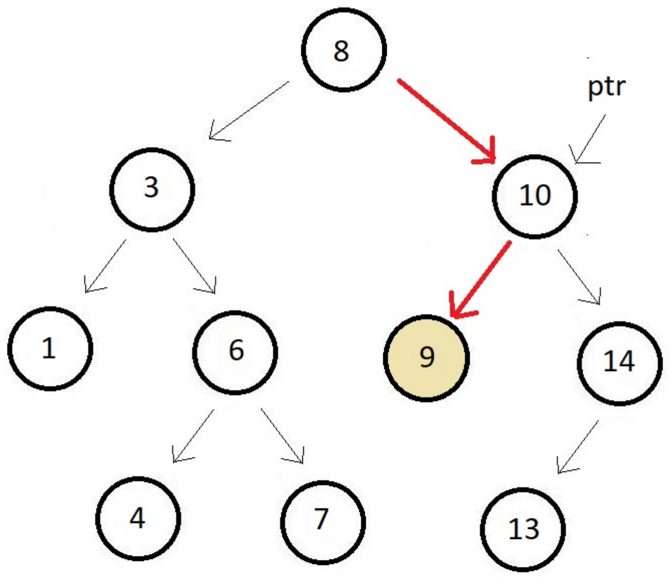
binary search tree, and the element we want to insert is 9.



Now, you would simply start from the root node, and see if the element you want to insert is greater than or less than. And since 9 is greater than 8, we move to the right of the root. And then the root is the element 10, and since this time 9 is less than 10, we move to the left of it. And since there are no elements to its left, we simply insert element 9 there.



This was one simple case, but things become more complex when you have to insert your element at some internal position and not at the leaf.
Now, before you insert a node, the first thing you would do is to create that node and allocate memory to it in heap using malloc. Then you would initialize the node with the data given, and both the right and the left member of the node should be marked NULL.
And another important thing to see here is the pointer you would follow the correct position with. In the above example, to be able to insert at that position, the pointer must be at node 10.



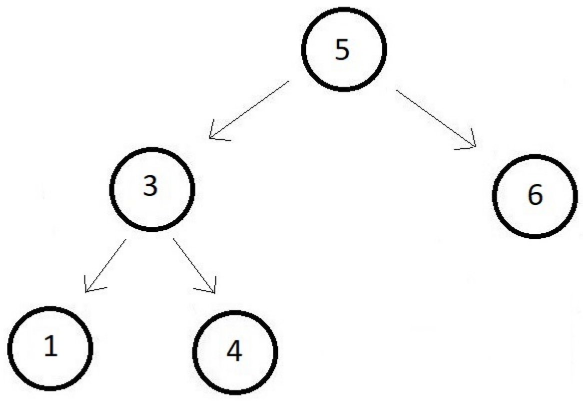
And then you check whether going to the left side is good, or the right. Here you came to the left, but had it been right, we would have updated our pointer *ptr* further and maintained a second pointer to the previous root. We'll see all this via our program. That would actually make things simpler to understand. I have attached the source code below. Follow it as we proceed.

Understanding the source code below:

1. Now, since our main focus would be to create the *insert* function, we could just copy everything we did in the last programming lecture where we learned the

iterative search operation. And this would save us a lot of time. And we would have the createNode and other functions we did before at one place for our exigency.

2. Create a binary search tree of your choice, we would rather go with the one we already had in the program. Now, let's see the *insert*



Creating the *insert* function:

3. So, the first thing we would like to know is whether inserting this new node is even possible or not. For that, create a void function *insert* and pass the pointer to the root node, and the data of the node you want to insert as its parameters. We will call it
4. Now, we would use two struct pointers to traverse through the array. One of them would be our *root* which we would traverse through the nodes, and the other one would be *prev* which stores the pointer to the previous root. SO, just create a struct Node pointer *prev* to maintain the node you were previously at, at some point in time.
5. Run a while loop that is for until we reach some leaf, and couldn't traverse further. So, run that loop until the root becomes NULL. And inside that loop, make the *prev* equal to the current root since we would definitely move further because this root is not a NULL. We would either move to the left of this root or to the right of this root. But before that check, if this root itself is not equal to the node we are trying to insert. That is, write an if condition to see if there are any duplicates here. If there is, return from the function here itself.
6. Further in the loop, check if the element you want to insert is less than the

current root. If it is, update the root to the left element of the struct root. And if it isn't, update the root to the right element of the struct root. And since we have already stored this root in the *prev* node, there isn't any issue updating.

7. And finally, you will have a *prev* node as the outcome at the end after this loop finishes. Now, the only procedure left now is to link these nodes together, that is the *prev* node, the *new* node*,* and the node *next* to the *prev*
8. Now, before you insert, make sure you create that new struct node using malloc, or simply the *createNode* Fill in the *key* data into this *new* node. Now, simply check if the *prev->data* is less than the key or greater than the key. If it is less, insert our new node to the left of *prev,* else to right of *prev.* And that would be it. We are done inserting our new node.

```
void insert(struct node *root, int key){
   struct node *prev = NULL;
   while(root!=NULL){
       prev = root;
       if(key==root->data){
           printf("Cannot insert %d, already in BST", key);
           return;
       }
       else if(key<root->data){
           root = root->left;
       }
       else{
           root = root->right;
       }
   }
   struct node* new = createNode(key);
   if(key<prev->data){
       prev->left = new;
   }
   else{
       prev->right = new;
   }
}
```

Code Snippet 1: Creating the *insert* function

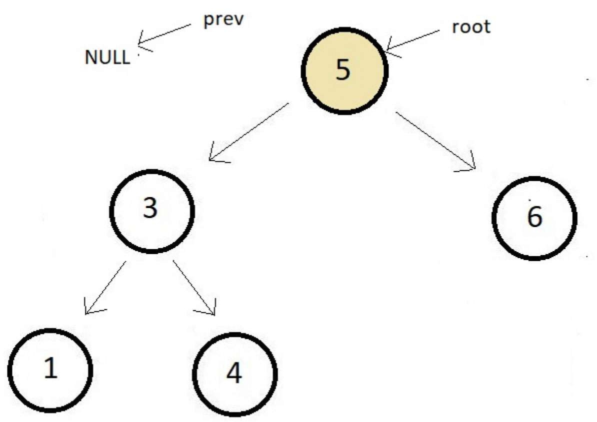
Here is the whole source code:

```
#include<stdio.h>
#include<malloc.h>

struct node{
    int data;
    struct node* left;
    struct node* right;
};

struct node* createNode(int data){
    struct node *n; // creating a node pointer
    n = (struct node *) malloc(sizeof(struct node)); // Allocating m
    n->data = data; // Setting the data
    n->left = NULL; // Setting the left and right children to NULL
    n->right = NULL; // Setting the left and right children to NULL
    return n; // Finally returning the created node
}

void preOrder(struct  node* root){
    if(root!=NULL){
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}

void postOrder(struct  node* root){
    if(root!=NULL){
        postOrder(root->left);
        postOrder(root->right);
        printf("%d ", root->data);
    }
}

void inOrder(struct  node* root){
    if(root!=NULL){
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);
    }
}
```

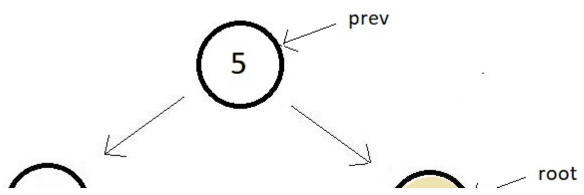
```
int isBST(struct  node* root){
    static struct node *prev = NULL;
    if(root!=NULL){
        if(!isBST(root->left)){
            return 0;
        }
        if(prev!=NULL && root->data <= prev->data){
            return 0;
        }
        prev = root;
        return isBST(root->right);
    }
    else{
        return 1;
    }
}

struct node * searchIter(struct node* root, int key){
    while(root!=NULL){
        if(key == root->data){
            return root;
        }
        else if(key<root->data){
            root = root->left;
        }
        else{
            root = root->right;
        }
    }
    return NULL;
}

void insert(struct node *root, int key){
   struct node *prev = NULL;
   while(root!=NULL){
       prev = root;
       if(key==root->data){
           printf("Cannot insert %d, already in BST", key);
           return;
       }
       else if(key<root->data){
           root = root->left;
       }
       else{
           root = root->right;
       }
   }
   struct node* new = createNode(key);
   if(key<prev->data){
       prev->left = new;
   }
   else{
       prev->right = new;
   }

}

int main(){

    // Constructing the root node - Using Function (Recommended)
    struct node *p = createNode(5);
    struct node *p1 = createNode(3);
    struct node *p2 = createNode(6);
    struct node *p3 = createNode(1);
    struct node *p4 = createNode(4);
    // Finally The tree looks like this:
    //      5
    //     / \
    //    3   6
    //   / \
    //  1   4

    // Linking the root node with left and right children
    p->left = p1;
    p->right = p2;
    p1->left = p3;
    p1->right = p4;

    insert(p, 16);
    printf("%d", p->right->right->data);
    return 0;
}
```

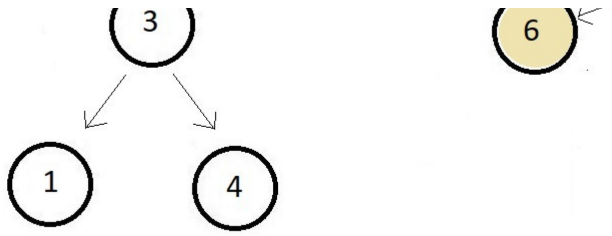
Code Snippet 2: Implementing the *insert* function in C

Now, suppose we want to insert element 7 in the above BST we had in the program. Now, let's first dry run that ourselves. And then we'll verify the position of the inserted node via the program. So, firstly, *prev* is NULL. And we start driving in the loop.

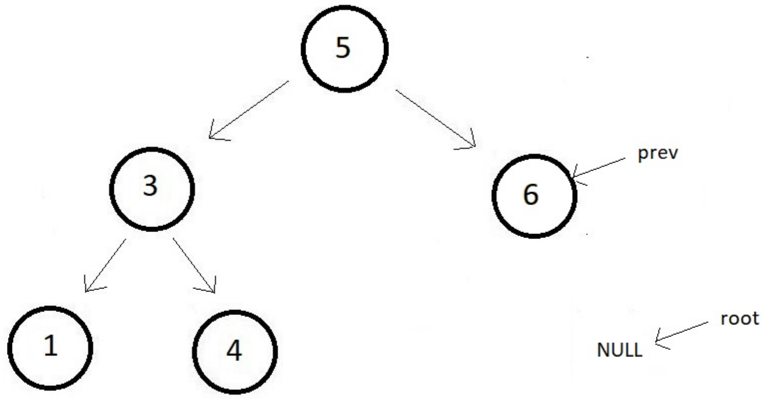


Now, first, we'll make our *prev* point to the root node, and since 7 is greater than 5, we move to the right of the root.

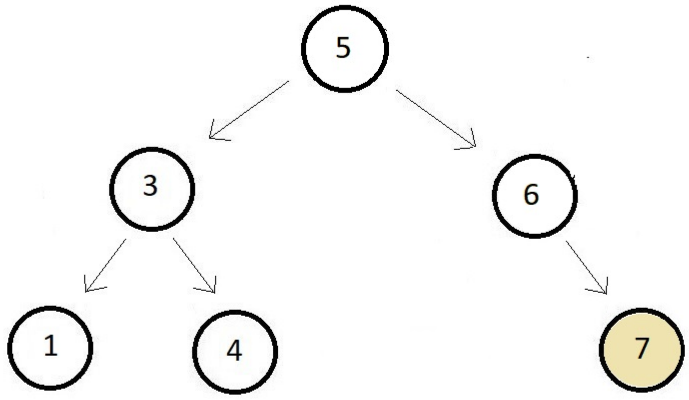




Now, we'll again update our *prev* with root, and since root is still smaller than 7, we'll move to the root's right which is a NULL. So, our loop ends here, and we get node 6 as our *prev* node.



And since 6 is smaller than 7, we'll insert node 7 to the right of node 6. And hence our element 7 gets placed at the position relative to the root as root -> right -> right.



Now, let's verify this by running the *insert* function, and print the data at the root -> right -> right position.

```
insert(p, 7);
printf("%d", p->right->right->data);
```

Code Snippet 3: Using the *insert* function

And the output we received was:

```
7
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 1: Output of the above code

Heh, so that got inserted with this ease. And that is all we had in the insertion operation. It is possible to learn this on your own if you practice. Next, we have the deletion function. Having learnt the insertion and the search operations would make learning deletion a cakewalk. You just have to be patient with me.

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 the process of deleting nodes in a binary search tree. Till then keep coding.

Previous Next

CodeWithHarry