C Code For Searching in a BST

In the last lecture, we saw one important application of binary search trees. It is its search operation that allows us to search any key in the binary search tree in *logn* best case time complexity. We saw how searching is done by reducing the search space after every comparison, and how searching time is proportional to the height of the tree. Today, we'll see the programming of this search function in C and automate the searching process.

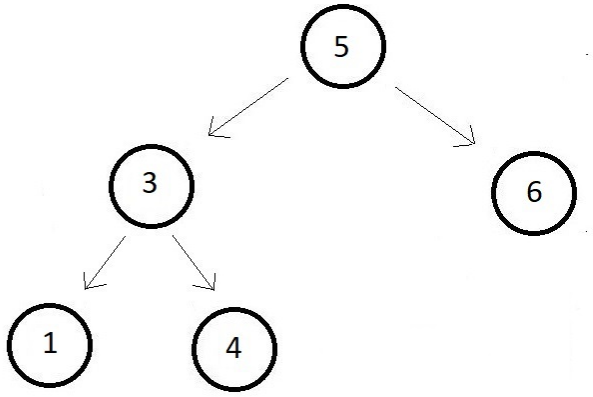
We did see the pseudocode of the search function in the last class. Follow it while we understand the procedure. I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. First of all, we'll anyway have to create a binary search tree to be able to use it to search some keys. And since we created everything the last time, we'll just copy everything we have done till the last programming lecture. It covered everything from creating a node to building a tree. It also features all the traversal methods

and the isBST function. We copied everything so as to keep everything in one place, and to avoid repeating things in the course, and this has also saved us a lot of time.

2. Let's create a binary search tree I've illustrated below using the createNode function. Creating a binary search tree should be an issue anymore. And if you are still not sure about creating a binary search tree, follow the previous lectures.



3. We'll do each of the other operations we usually do. But today our focus is on the search operation. Let's now just create a function to search a key in the binary search tree we created.

Creating the function search:

4. Create a struct Node pointer function *search* and pass into it the pointer to the root and the key you want to search as two of its parameters.
5. First of all, check if the root is NULL. If the root is NULL, we haven't found our key, and we'll simply return NULL.
6. Now, check if the node we are currently at has the data element equal to the key we were searching for. If it is the case, we have found the key, and we'll simply return the pointer to the current root.
7. But if we still haven't found the key, it is probably in the left subtree of the root, or in the right subtree of the root. To judge which side to proceed with, we'll check if the current root is less than or greater than the data element of the root. If it is less than the root, the key probability lies on the left subtree, and hence we would reduce our search space and recursively start searching in the left subtree. Otherwise, we would start searching in the right subtree. Anyways, we are

reducing our search space after each comparison.

```
struct node * search(struct node* root, int 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);
    }
}
```

Code Snippet 1: Creating the search function

Recursion in the above function is pretty simple this time, unlike the isBST function we did early. Here, we simply decide every time which subtree to pursue next. And recursively go with that.

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 * search(struct node* root, int 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);
    }
}

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;

    struct node* n = search(p, 10);
    if(n!=NULL){
    printf("Found: %d", n->data);
    }
    else{
        printf("Element not found");
    }
    return 0;
}
```

Code Snippet 2: Implementing the search function

Now, make sure you store the value returned by the search function in a struct node pointer, say n. We'll see if our function actually works, and it reverts the pointer to the node it found, or a NULL if it didn't find the key. Let's search 10, in the above function.

```
    struct node* n = search(p, 10);
    if(n!=NULL){
    printf("Found: %d", n->data);
    }
    else{
```

```
        printf("Element not found");
    }
```

Code Snippet 3: Using the search function

And the output we received was:

```
Element not found
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 1: Output of the above code

And since 10 was not there in the binary search tree we created, the function *search* returned a NULL. Let's make it work for something which is there. Let's use the function to find 3 in the above BST.

```
    struct node* n = search(p, 3);
    if(n!=NULL){
    printf("Found: %d", n->data);
    }
    else{
        printf("Element not found");
    }
```

Code Snippet 4: Using the search function again

And the output we received was:

```
Found: 3
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 2: Output of the above code

And yes, it says, our function search found the key 3 in the above binary search tree. It was simple to digest I believe. We had already learned the concepts in the last lecture. Searching shouldn't be a deal, since binary search trees were structured so as to make searching look easy. :)

So, this was all we had to do to automate the searching in a binary search tree. And

that would be all. Create your own binary search trees, and test the *search* function. Please let me know if it has any flaws. It is now time to move on to our next operation which is something similar to what we did today but with a different approach.

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 search iteratively in a binary search tree, and write its code in C. Till then keep coding.

Previous Next