Iterative Search in a Binary Search Tree

In the last lecture, we learned how to program the *search* function to be able to search keys in a binary search tree. The function returns the pointer to the node if it finds the key, otherwise returns NULL. We saw how easy binary search trees make the process of searching and offers a best-case runtime complexity of *logn*.

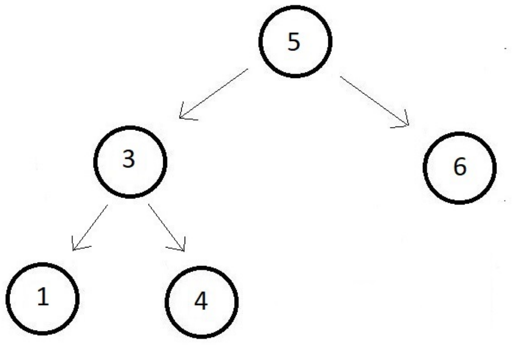
The *search* function we studied was a **recursive** function, calling the function recursively on the left or the right subtree. But today, we'll see the concept of searching **iteratively** in a binary search tree and its programming in C.

In the recursive approach, you start with the root and compare your key with the root node's data, and if it's smaller than the key, you select the whole left subtree and start searching in it thinking of it as another smaller tree. But in an iterative approach, we have the tree intact the whole time. We just make a pointer run from the root to the place we expect our key to be. Let's directly see the code for iterative search.

I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. There is nothing much to explain in the code. Although for the people coming right here, we have just copied everything we had done till the last programming lecture which dealt with the search function. It had covered everything from creating a node, to building a tree. It also featured all the traversal methods, and all the other functions. We did this to save ourselves some time.
2. Let's create a binary search tree I've illustrated below using the createNode function. And without doing any further ado, we would move to creating the *searchIter*



Creating the function searchIter:

3. Create a struct Node pointer function *searchIter* and pass into it the pointer to the root and the key you want to search as two of its parameters.
4. First of all, check if the root's data itself is the key we were looking for. If the root data is equal to the key, we have found the key, and we'll return the pointer to the root.
5. If we couldn't find the key on the root, we'll further see if our key is greater than or smaller than the root data. If it is smaller than the root data, we will make the root node pointer now point to the left child of the root, using the arrow operator. And if the key is greater than the root data, we'll make the root node pointer point to the right child of the root
6. And we'll use a while loop to run steps 4 and 5 iteratively until our root becomes NULL, or we find the key and return from the function.
7. And in the end, if we couldn't find the key after iterating through the tree, or the root we received in the starting itself was a NULL, we would return NULL.

```
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;
}
```

Code Snippet 1: Creating the searchIter function

Iteration is more intuitive because you are able to see what's exactly happening and how things end. Even so, recursion is considered a powerful tool.

Here is the whole source code:

```c
#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 memory in
the heap
    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;
}

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 = searchIter(p, 6);
    if(n!=NULL){
    printf("Found: %d", n->data);
    }
    else{
        printf("Element not found");
    }
    return 0;
}
```

Code Snippet 2: Implementing the searchIter function

Now, make sure you store the value returned by the searchIter 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. We did all this time as well.

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

Code Snippet 3: Using the searchIter 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 *searchIter* returned a NULL. Let’s see if it works for something else. Let’s use the function to find 6 in the above BST.

```
    struct node* n = searchIter(p, 6);
    if(n!=NULL){
    printf("Found: %d", n->data);
    }
    else{
        printf("Element not found");
    }
```

Code Snippet 4: Using the searchIter function again

And the output we received was:

```
Found: 6
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 2: Output of the above code

Ah yes. The function is working all good. Although we changed our approach from recursive to iterative, things remained pretty much the same. We still stop our searching after reaching the NULL. So, more or less the base case and the flow remain the same. And it did feel more intuitive.

So, this was all we had in searching. We saw both approaches. I encourage you to experiment with both methods. Having well-versed in both methods, it is now time to move on to our next operation that is the insertion in a binary search tree. Furthermore, we will see the deletion of nodes in a binary search tree. Things are pretty interesting on the other side. Stay connected.

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 inserting nodes in a binary search tree. Till then keep coding.