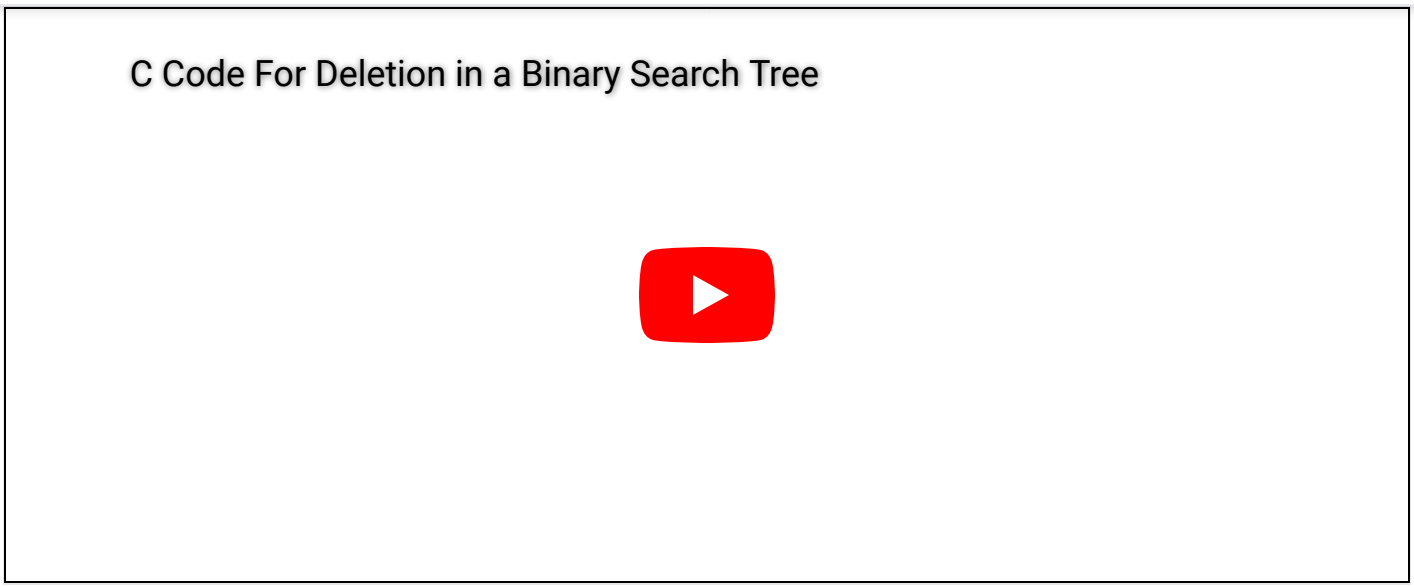


Show Course Contents

Overview Q&A Downloads Announcements

C Code For Deletion in a Binary Search Tree

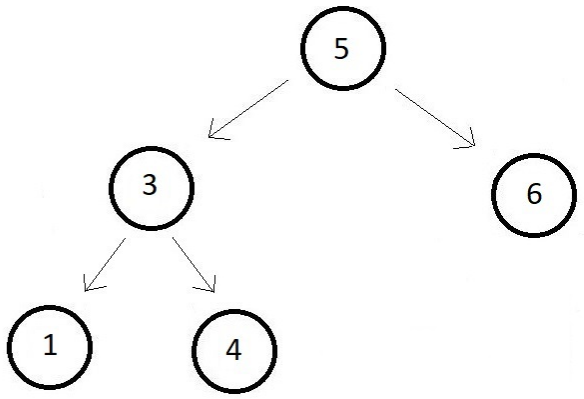
In the last lecture, we dealt with all the cases of deletion operations in a binary search tree. Despite the complexity of the cases, we made them all appear easy. We saw examples of all the cases. Today, we'll see how to program those deletion cases and overall, the implementation of the deletion operation in a binary search tree using C language.

So, let's just move on to our editors without any further ado. 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 *deleteNode* function, we could just copy everything we did in the last programming lecture where we learnt the *insertion* operation in a binary search tree. And this would save us a lot of time. And we would then have the InOrder and other traversal functions we did before at one place for our convenience.
2. You can always create a binary search tree of your own, and check if it is a BST

using the InOrder traversal, and see if it's in ascending order or not. We would rather go with the one we already had in the program. Now, let's see the *deleteNode*



Creating the *deleteNode* function:

3. So, the first thing we would like to know is whether deleting this new node is even possible or not. For that, create a struct Node pointer function *deleteNode* and pass the pointer to the root node, and the data of the node you want to delete as its parameters. We will call it *value*

4. So, the next thing you would do is see whether our *value* is greater than or less than the root node's data. If it's less than the root node's data, we'll simply recursively call the function to its left subtree, and pass the same *value* into it. And if it's greater, we'll do the same thing with the right subtree. So, this basically reduces our problem. We do this until we reach the node we want to delete, or to the NULL, signifying the node was not there in the tree.

5. So, the deletion strategy says, when you find the node you want to delete, if you remember what we learnt, we can substitute it with its InOrder predecessor or with its InOrder successor. And we prefer doing it here using the InOrder predecessor. So, the first thing you would do here is to find the Inorder predecessor of the root node you want to delete and store it in a struct node pointer *iPre*. We would have a separate function for this, named *inOrderPredecessor* which we'll see later. Suppose it returns the pointer to the InOrder Predecessor of the root node.

6. Having received the pointer to the InOrder Predecessor of the root node, you just copy the data of that predecessor node to our root node. And call the *deleteNode* function recursively to the left subtree of the root node, and with the value of the

InOrder predecessor's data. Now, observe this carefully. Why did we choose the left subtree? Because the InOrder predecessor of a node always lies on the left subtree of the node. And since you have replaced your root node with its InOrder predecessor, you have to now delete it from the left subtree.

7. And since these functions are recursive, we would need a base condition, and if you could realize, this recursive deletion of the replacing nodes must stop when it reaches a leaf node. So, the base conditions are when our given root is NULL, here we simply return NULL or both it's left and the right subtree is NULL, this is where the node we want to delete is the leaf node, so we'll simply free this node. And simply return the root node at the end.

```
struct node *deleteNode(struct node *root, int value){

    struct node* iPre;
    if (root == NULL){
        return NULL;
    }
    if (root->left==NULL&&root->right==NULL){
        free(root);
    }

    //searching for the node to be deleted
    if (value < root->data){
        deleteNode(root->left,value);
    }
    else if (value > root->data){
        deleteNode(root->right,value);
    }
    //deletion strategy when the node is found
    else{
        iPre = inOrderPredecessor(root);
        root->data = iPre->data;
        deleteNode(root->left, iPre->data);
    }
}
```

Code Snippet 1: Creating the *deleteNode* function

8. Now, this deleteNode function would not simply work alone. We would need an *inOrderPredecessor* function we have used in the program.

Creating the *inOrderPredecessor* function:

9. Create a struct node pointer function *inOrderpredecessor* and pass into it the pointer to the root node you want to find the InOrder predecessor of as its parameter. I hope you all would have realised the fact that an Inorder predecessor of a node is the rightmost node of its left subtree.

10. So, simply update root to its left subtree, and use a while loop to iterate through all the right subtree, until we reach a leaf node. This gives our InOrder predecessor of the given node. Here, return the root.

```
struct node *inOrderPredecessor(struct node* root){
    root = root->left;
    while (root->right!=NULL)
    {
        root = root->right;
    }
    return root;
}
```

Code Snippet 2: Creating the inOrderPredecessor 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
}
```

```c
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;
        }
    }

    struct node *inOrderPredecessor(struct node* root){
        root = root->left;
        while (root->right!=NULL)
        {
            root = root->right;
        }
        return root;
    }

    struct node *deleteNode(struct node *root, int value){

        struct node* iPre;
        if (root == NULL){
            return NULL;
        }
        if (root->left==NULL&&root->right==NULL){
            free(root);
            return NULL;
        }

        //searching for the node to be deleted
        if (value < root->data){
            root-> left = deleteNode(root->left,value);
        }
        else if (value > root->data){
```

```
        root-> right = deleteNode(root->right,value);
    }
    //deletion strategy when the node is found
    else{
        iPre = inOrderPredecessor(root);
        root->data = iPre->data;
        root->left = deleteNode(root->left, iPre->data);
    }
    return root;
}

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;

    inOrder(p);
    printf("\n");
    deleteNode(p, 3);
```

```
    inOrder(p);

    return 0;
}
```

Code Snippet 3: Implementing the *deleteNode* function in C

Now, let's verify this function by running the *deleteNode* function with the value being 3, and to see if the node really got deleted, we'll run the inOrder function before and after the deletion operation.

```
    inOrder(p);
    printf("\n");
    deleteNode(p, 3);
    inOrder(p);
```

Code Snippet 4: Using the *deleteNode* function

And the output we received was:

```
1 3 4 5 6
1 4 5 6
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 1: Output of the above code

So, yes, node 3 got deleted from the tree. And our tree didn't lose its identity, it remained a binary search tree. You can run the same function for all the other nodes to see if it works for all the cases. And try this for even bigger trees. Go through the last lecture again if you couldn't get hold of the basics of deletion. Next, we have our new data structure in the list, called the AVL trees.

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 get introduced to our next data structure, **AVL Trees**. Till then keep coding.

Previous

Next

CodeWithHarry