Binary Search Tree in C

Sept. 27, 2018 C (/blog/tag/c/?tag=c) C++ (/blog/tag/cpp/?tag=cpp) TREE (/blog/tag/tree/?tag=tree) BINARY TREE (/blog/tag/binary-tree/?tag=binary-tree) BINARY SEARCH TREE (/blog/tag/binary-search-tree/?tag=binary-search-tree) DATA STRUCUTRE (/blog/tag/data-strucutre/?tag=data-strucutre) 8408

(https://play.google.com/store/apps/details?id=com.blogsdope&pcampaignid=MKT-Other-global-all-co-prtnr-py-PartBadge-Mar2515-1)

Previous

1. Trees in Computer Science (https://www.codesdope.com/blog/article/trees-in-computer-science)
2. Binary Tree (https://www.codesdope.com/blog/article/binary-trees)
3. Binary Trees in C : Array Representation and Traversals (https://www.codesdope.com/blog/article/binary-trees-in-c-array-representation-and-travers)
4. Binary Tree in C: Linked Representation & Traversals (https://www.codesdope.com/blog/article/binary-tree-in-c-linked-

representation-traversals)

5. Binary Search Tree (https://www.codesdope.com/blog/article/binary-search-tree)

This post is about the coding implementation of ***BST*** in C and its **explanation**.
To learn about the concepts behind a binary search tree, the post Binary Search Tree (https://www.codesdope.com/blog/article/binary-search-tree) would be helpful.

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

struct node
{
    int data; //node will store an integer
    struct node *right_child; // right child
    struct node *left_child; // left child
};

struct node* search(struct node *root, int x)
{
    if(root==NULL || root->data==x) //if root->data is x then the element
        return root;
    else if(x>root->data) // x is greater, so we will search the right su
        return search(root->right_child, x);
    else //x is smaller than the data, so we will search the left subtree
        return search(root->left_child,x);
}

//function to find the minimum value in a node
struct node* find_minimum(struct node *root)
{
    if(root == NULL)
        return NULL;
    else if(root->left_child != NULL) // node with minimum value will hav
        return find_minimum(root->left_child); // left most element will
    return root;
}

//function to create a node
struct node* new_node(int x)
{
    struct node *p;
    p = malloc(sizeof(struct node));
    p->data = x;
    p->left_child = NULL;
    p->right_child = NULL;

    return p;
}

struct node* insert(struct node *root, int x)
{
    //searching for the place to insert
    if(root==NULL)
        return new_node(x);
    else if(x>root->data) // x is greater. Should be inserted to right
        root->right_child = insert(root->right_child, x);
    else // x is smaller should be inserted to left
        root->left_child = insert(root->left_child,x);
    return root;
}

// funnction to delete a node
struct node* delete(struct node *root, int x)
{
    //searching for the item to be deleted
    if(root==NULL)
```

```
        return NULL;
    if (x>root->data)
        root->right_child = delete(root->right_child, x);
    else if(x<root->data)
        root->left_child = delete(root->left_child, x);
    else
    {
        //No Children
        if(root->left_child==NULL && root->right_child==NULL)
        {
            free(root);
            return NULL;
        }

        //One Child
        else if(root->left_child==NULL || root->right_child==NULL)
        {
            struct node *temp;
            if(root->left_child==NULL)
                temp = root->right_child;
            else
                temp = root->left_child;
            free(root);
            return temp;
        }

        //Two Children
        else
        {
            struct node *temp = find_minimum(root->right_child);
            root->data = temp->data;
            root->right_child = delete(root->right_child, temp->data);
        }
    }
    return root;
}

void inorder(struct node *root)
{
    if(root!=NULL) // checking if the root is not null
    {
        inorder(root->left_child); // visiting left child
        printf(" %d ", root->data); // printing data at root
        inorder(root->right_child);// visiting right child
    }
}

int main()
{
    /*
                    20
                  /    \
                 /      \
                5        30
              /   \      /\
             /     \    /  \
            1      15 25   40
                  /           \
                 /             \
                9               45
```

```
          /   \           /
         /     \         /
        7      12       42
*/
struct node *root;
root = new_node(20);
insert(root,5);
insert(root,1);
insert(root,15);
insert(root,9);
insert(root,7);
insert(root,12);
insert(root,30);
insert(root,25);
insert(root,40);
insert(root, 45);
insert(root, 42);

inorder(root);
printf("\n");

root = delete(root, 1);
/*
                20
              /    \
             /      \
            5       30
              \     /\
               \   /  \
               15 25   40
             /            \
            /              \
           9                45
         /   \             /
        /     \           /
       7      12        42
*/

root = delete(root, 40);
/*
                20
              /    \
             /      \
            5       30
              \     /\
               \   /  \
               15 25  45
             /        /
            /        /
           9        42
         /   \
        /     \
       7      12
*/

root = delete(root, 45);
/*
                20
              /    \
             /      \
```

```
                5        30
                  \      /\
                   \    /  \
                   15 25  42
                  /
                 /
                9
              /   \
             /     \
            7      12
    */
    root = delete(root, 9);
    inorder(root);
    /*
                   20
                 /    \
                /      \
               5        30
                 \      /\
                  \    /  \
                  15 25  42
                 /
                /
               12
             /
            /
           7
    */
    printf("\n");

    return 0;
}
```

Explanation

The making of a node and traversals are explained in the post Binary Trees in C: Linked Representation & Traversals (https://www.codesdope.com/blog/article/binary-tree-in-c-linked-representation-traversals). Here, we will focus on the parts related to the binary search tree like inserting a node, deleting a node, searching, etc. Also, the concepts behind a binary search tree are explained in the post Binary Search Tree (https://www.codesdope.com/blog/article/binary-search-tree).

Search

```
struct node* search(struct node *root, int x)
{
     if(root==NULL || root->data==x)
          return root;
     else if(x>root->data)
          return search(root->right_child, x);
     else
          return search(root->left_child,x);
}
```

`search` is a function to find any element in the tree. To search an element we first visit the root and if the element is not found there, then we compare the element with the data of the root and if the element is greater, then it must lie on the right subtree (property of a BST – All elements greater than the data at the node are on the right subtree), otherwise on the left subtree. We repeat this process until the element is found or a null value is reached (the element is not in the tree).

`if(root==NULL || root->data==x)` → If the null value is reached then the element is not in the tree and if the data at the root is equal to x then the element is found.

`else if(x>root->data)` → The element is greater than the data at the root, so we will search in the right subtree – `search(root->right_child, x)`. Otherwise, on the left subtree – `search(root->left_child,x)`.

Inserting a new node

```
struct node* insert(struct node *root, int x)
{
     if(root==NULL)
          return new_node(x);
     else if(x>root->data)
          root->right_child = insert(root->right_child, x);
     else
          root->left_child = insert(root->left_child,x);
     return root;
}
```

Inserting a new node is similar to searching for an element in a tree. We first search for the element and if it is not found at the required place (where it should be) then we just insert a new node at that position. If the element to be inserted is greater than the data at the node, then we insert it in the right subtree – `root->right_child = insert(root->right_child, x)`. Suppose,

we have to insert a new node to the right child of a node 'X'. So, we will first create a new node which is returned by `insert(root->right_child, x)` and then make the right child of 'X' equal to that node – `root->right_child = insert(root->right_child, x)`. So after searching, if we reach to a null node, then we just insert a new node there – `if(root==NULL) → return new_node(x)`.

Thus, the entire `insert` function can be summed up as – If the current node is null then just return a new node. If the data at the current node is smaller than the data to be inserted, then we will change the right child of the current node with the right subtree obtained with the `insert` function. The insert function will either return a new node (in case of a null node) or the modified subtree itself (`return root`).

Delete

As discussed in Binary Search Tree (https://www.codesdope.com/blog/article/binary-search-tree), the code for the deletion is:

```
struct node* delete(struct node *root, int x)
{
    if(root==NULL)
        return NULL;
    if (x>root->data)
        root->right_child = delete(root->right_child, x);
    else if(x<root->data)
        root->left_child = delete(root->left_child, x);
    else
    {
        //No Children
        if(root->left_child==NULL && root->right_child==NULL)
        {
            free(root);
            return NULL;
        }

        //One Child
        else if(root->left_child==NULL || root->right_child==NULL)
        {
            struct node *temp;
            if(root->left_child==NULL)
                temp = root->right_child;
            else
                temp = root->left_child;
            free(root);
            return temp;
        }

        //Two Children
        else
        {
            struct node *temp = find_minimum(root->right_child);
            root->data = temp->data;
            root->right_child = delete(root->right_child, temp->data);
        }
    }
    return root;
}
```

If the tree has no children (`if(root->left_child==NULL && root->right_child==NULL)`) – Just delete the node – `free(root)`.

If only one child (`(root->left_child==NULL || root->right_child==NULL)`)– Make the parent of the node to be deleted point to the child. – `if(root->left_child==NULL)` – Only right child exists. We have to delete this node but we also have to point its parent to its child, so we are storing its child into a temporary variable – `temp = root->right_child` and then deleting the node – `free(root)`.

If two children – Find the minimum element of the right subtree – `find_minimum(root->right_child)`. Replace the data of the node to be deleted with the data of this node – `root->data = temp->data`. Delete node found by the minimum function – `delete(root->right_child, temp->data)`.

So, this post was all about the coding implementation of the binary search tree (https://www.codesdope.com/blog/article/binary-search-tree) in C. You can see the implementation of a BST in Java (https://www.codesdope.com/java-introduction) in the post – Binary Search Tree in Java (https://www.codesdope.com/blog/article/binary-search-tree-in-java/).

Amit Kumar (/blog/author/54322/?author=54322)

Developer and founder of CodesDope.

(https://www.facebook.com/codesdope) (https://www.twitter.com/codesdope) (https://www.linkedin.com/in/amit-kumar-66903395)