

ECE 220 Computer Systems & Programming

Lecture 23-24: C++ Examples



BST (Binary Search Tree)

- Create a binary search tree and perform:
 - Insertion
 - Search a node
 - Count no. of nodes
 - Find height of the tree
 - Traverse the tree (inorder) and
store the nodes in a vector
 - Delete tree
 - Delete a particular node!

BST (class)

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

```
class bst{  
    public:  
        bst();  
        ~bst();  
        void insert(int data);  
        node *search(int data);  
        void inorder();  
        int countnodes();  
        int getHeight();  
  
    private:  
        void delete_tree(node *root);  
        void insert(int data, node *root);  
        node *search(int data, node *root);  
        void inorder(node *root, vector<int> &v);  
        int countnodes(node *root);  
        int getHeight(node *root);  
        node *root;  
};
```

BST (main)

```
int main() {  
    cout<<"build a binary search tree"<<endl;  
    bst tree1;  
    tree1.insert(30);  
    tree1.insert(20);  
    tree1.insert(10);  
    tree1.insert(15);  
    tree1.insert(50);  
    tree1.insert(12);  
}
```

BST (main cont.)

```
cout<<"total number of nodes in this tree: "<<tree1.countnodes()<<endl;

//searching a node;

int node_data=6;
cout<<"Searching a node with data_val:"<<node_data<<endl;

node *x;
x=tree1.search(node_data) ;

if (x)
    cout<<"Node Found. The address is: "<<x<<endl;
else
    cout<<"Node does not exist"<<endl;
```

BST (main cont.)

```
//Calculating the height of the Tree

cout<<"The height of the tree is:"<<tree1.getHeight()<<endl;

//Inorder Traversing of the Tree
tree1.inorder();

    return 0;
}
```

BST (constructor & insertion)

Constructor:

```
bst::bst() {  
    root = NULL;  
}
```

```
void bst::insert(int data) {  
    if (root == NULL) {  
        root = new node;  
        root->data = data;  
        root->left = root->right = NULL;  
        return;  
    }  
    else  
        insert(data, root);  
}
```

BST (insertion)

```
void bst::insert(int data){
    if(root == NULL){
        root = new node;
        root->data = data;
        root->left = root->right = NULL;
        return;
    }
    else
        insert(data, root);
}
```

```
void bst::insert(int data, node *root){
    if(data == root->data)
        return; //data already exists in BST
    else if(data < root->data){
        if(root->left == NULL){
            root->left = new node;
            root->left->data = data;
            root->left->left = root->left->right = NULL;
            return;
        }
        else
            return insert(data, root->left);
    }
    else{
        if(root->right == NULL){
            root->right = new node;
            root->right->data = data;
            root->right->left = root->right->right = NULL;
            return;
        }
        else
            return insert(data, root->right);
    }
}
```

BST (search)

```
node * bst::search(int data){  
    if (root == NULL || root->data == data)  
        return root;  
    else  
        return search(data, root);  
}
```

```
node * bst::search(int data, node *root){  
    if(root == NULL)  
        return NULL;  
    if(data == root->data)  
        return root;  
    else if(data < root->data)  
        return search(data, root->left);  
    else  
        return search(data, root->right);  
}
```

BST (countnodes)

```
int bst::countnodes() {  
    if (root == NULL)  
        return 0;  
    else  
        return countnodes(root) ;  
}
```

```
int bst::countnodes(node *root) {  
    if (root == NULL)  
        return 0;  
    else  
        return 1+countnodes (root->left)+countnodes (root->right) ;  
}
```

BST (getHeight)

```
int bst::getHeight() {  
    if (root == NULL)  
        return 0;  
    else  
        return getHeight(root);  
}
```

```
int bst::getHeight(node *root) {  
    int lh, rh;  
  
    // base case: Reached to NULL  
    if (root == NULL)  
        return -1;  
  
    // recursive case: Calculate the height of the left-subtree and  
    // the right-subtree, and take the bigger one.  
  
    else {  
        lh = getHeight(root->left);  
        rh = getHeight(root->right);  
        if (lh > rh)  
            return lh + 1;  
        else  
            return rh + 1;  
    }  
}
```

BST (inorder traversing)

```
void bst::inorder() {  
    if (root == NULL)  
        cout<<"empty tree"<<endl;  
    else{  
        cout<<"inorder traversal of this binary search tree"<<endl;  
        vector<int> v;  
        inorder(root, v);  
    }  
}
```

```
void bst::inorder(node *root, vector<int> &v) {  
    if (root != NULL) {  
        inorder(root->left, v);  
        v.push_back(root->data);  
        inorder(root->right, v);  
    }  
}
```

BST (inorder traversing)

```
void bst::inorder(node *root, vector<int> &v) {  
    if (root != NULL) {  
        inorder(root->left, v);  
        v.push_back(root->data);  
        inorder(root->right, v);  
    }  
}
```

BST (inorder traversing)

```
void bst::inorder() {
    if (root == NULL)
        cout<<"empty tree"<<endl;
    else{
        cout<<"inorder traversal of this binary search tree"<<endl;
        vector<int> v;
        inorder(root, v);

        //Traversing vector using iterator
        /*
            for(auto it=v.begin(); it!=v.end(); it++){
                cout<<"node:"<<*it<<endl;
            }
        */

        //Traversing vector without using iterator
        for(int it=0; it<size(v); it++){
            cout<<"node:"<<v[it]<<endl;
        }
    }
}
```

BST (~bst) – delete BST

```
bst::~~bst() {  
    cout<<"delete a binary search tree "<<endl;  
    delete_tree(root);  
}
```

```
void bst::delete_tree(node *root) {  
    if (root != NULL) {  
        delete_tree(root->left);  
        delete_tree(root->right);  
        cout<<"node deleted: "<<root->data<<endl;  
        delete root;  
    }  
}
```

Delete a Node from BST (in C)

```
// C program to implement optimized delete in BST.
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int key;
    struct Node *left, *right;
};

// A utility function to create a new BST node
Node* newNode(int item)
{
    Node* temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}
```

```
/* A utility function to insert a new node with given key in
 * BST */
Node* insert(Node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL)
        return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
```

```
// Driver Code
int main()
{
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);

    printf("Original BST: ");
    inorder(root);

    printf("\n\nDelete a Leaf Node: 20\n");
    root = deleteNode(root, 20);
    printf("Modified BST tree after deleting Leaf Node:\n");
    inorder(root);

    printf("\n\nDelete Node with single child: 70\n");
    root = deleteNode(root, 70);
    printf("Modified BST tree after deleting single child Node:\n");
    inorder(root);

    printf("\n\nDelete Node with both child: 50\n");
    root = deleteNode(root, 50);
    printf("Modified BST tree after deleting both child Node:\n");
    inorder(root);

    return 0;
}
```

Ref: <https://www.geeksforgeeks.org/deletion-in-binary-search-tree/>

```
/* Given a binary search tree and a key, this function
deletes the key and returns the new root */
```

```
Node* deleteNode(Node* root, int k)
{
    // Base case
    if (root == NULL)
        return root;

    // Recursive calls for ancestors of
    // node to be deleted
    if (root->key > k) {
        root->left = deleteNode(root->left, k);
        return root;
    }
    else if (root->key < k) {
        root->right = deleteNode(root->right, k);
        return root;
    }

    // We reach here when root is the node
    // to be deleted.

    // If one of the children is empty
    if (root->left == NULL) {
        Node* temp = root->right;
        delete root;
        return temp;
    }
    else if (root->right == NULL) {
        Node* temp = root->left;
        delete root;
        return temp;
    }
}
```

```
// If both children exist
```

```
else {
    Node* succParent = root;
    // Find successor
    Node* succ = root->right;
    while (succ->left != NULL) {
        succParent = succ;
        succ = succ->left;
    }
    // Delete successor. Since successor
    // is always left child of its parent
    // we can safely make successor's right
    // right child as left of its parent.
    // If there is no succ, then assign
    // succ->right to succParent->right
    if (succParent != root)
        succParent->left = succ->right;
    else
        succParent->right = succ->right;
    // Copy Successor Data to root
    root->key = succ->key;
    // Delete Successor and return root
    delete succ;
    return root;
}

void inorder(Node* root)
{
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}
```