Aaron Williams

Assignment 3 (Homework Unit 4)

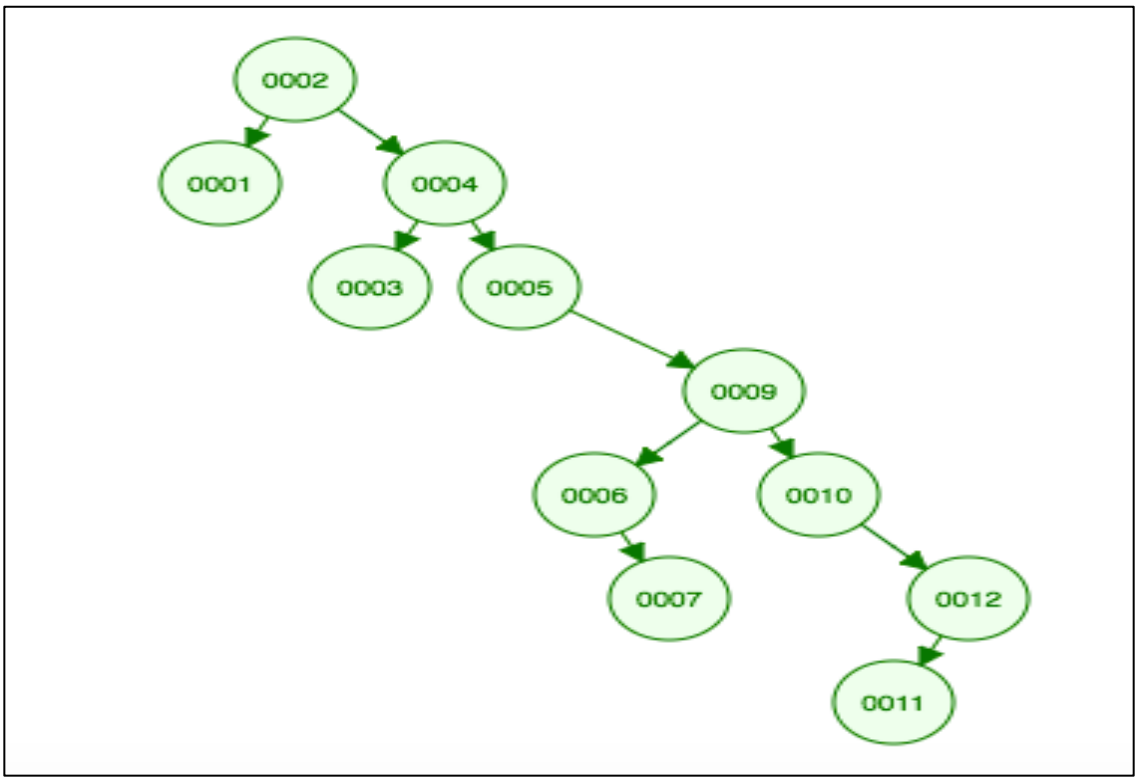
10 September 2017

Assignment Description

The purpose of this assignment is to begin with source code for a binary search tree (tree.cpp) and complete functions for inserting a node, deleting a node, and searching for a node while displaying relevant outputs as functions were built out.

Logic and Outputs

1) The first task was to implement the insert function and then display the results of inserting 2, 1, 4, 5, 9, 3, 6, 7, 10, 12, and 11 into the empty binary search tree. In theory, the binary tree should appear as the following:



The insert method function was created to check if the node to be inserted was less than or greater than the associated root/parent node. Following the principles of a binary tree, if the node value was less than the root/parent node, it was inserted to an appropriate position on the left. If the node

value was more than or equal to the root/parent node, it was inserted to an appropriate position on the right.

Insert Function:

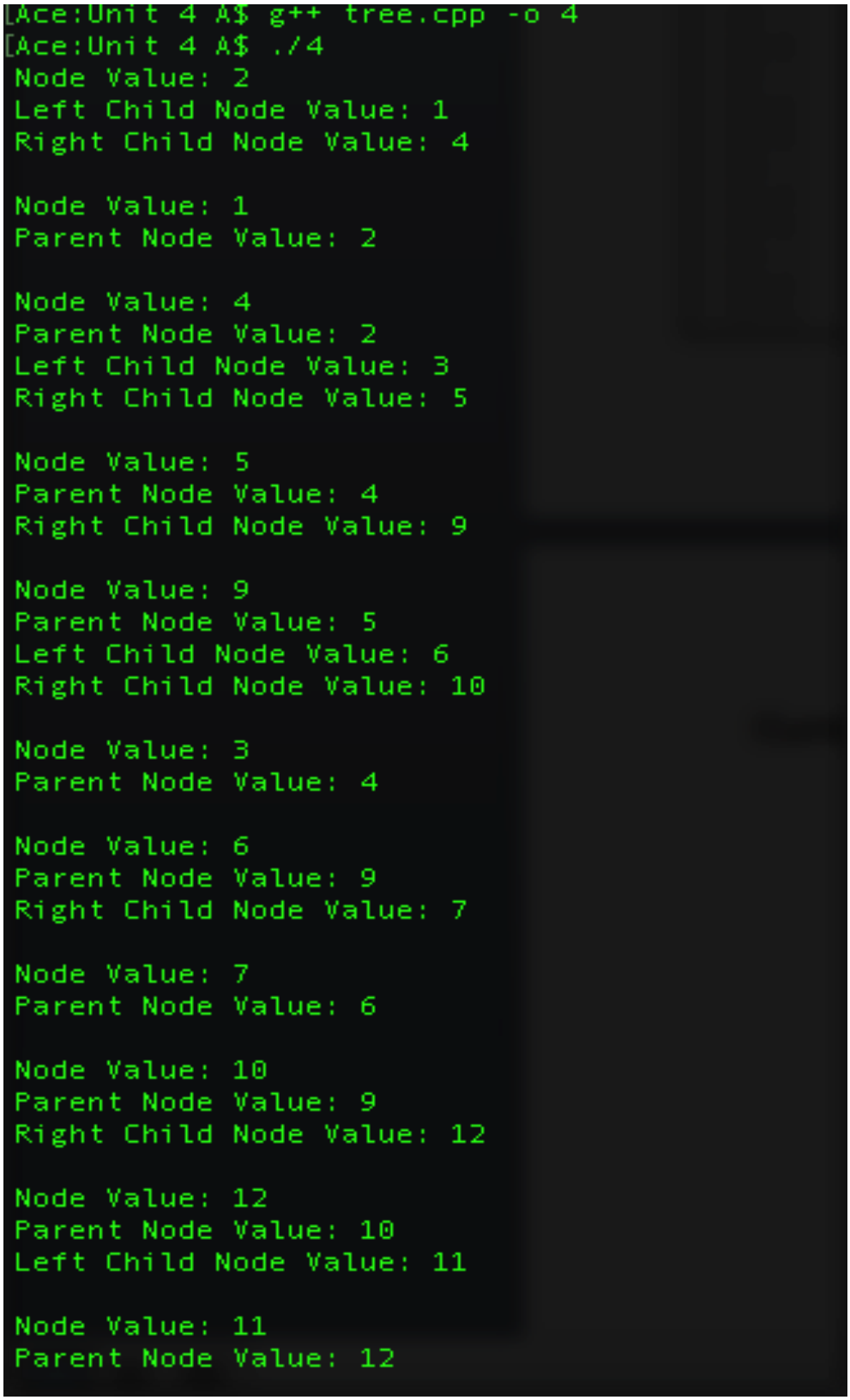
```
void insert(Node *insert_node, Node *tree_root) {
    //Your code here
    //Insert left if root value larger.
    if(insert_node->d() < tree_root->d()) {
        //Check if left child node exists.
        if(tree_root->left == NULL) {
            tree_root->left = insert_node;
            insert_node->p = tree_root;
        }
        else {
            insert(insert_node, tree_root->left);
        }
    }
    //Insert right if root value smaller or the same.
    if(insert_node->d() >= tree_root->d()) {
        //Check if right child node exists.
        if(tree_root->right == NULL) {
            tree_root->right = insert_node;
            insert_node->p = tree_root;
        }
        else {
            insert(insert_node, tree_root->right);
        }
    }
}
```

The nodes were created in the main function and inserted after their creation. Once the insertions were completed, all nodes were printed. In order to display the properties of each node, the print function was edited and implemented.

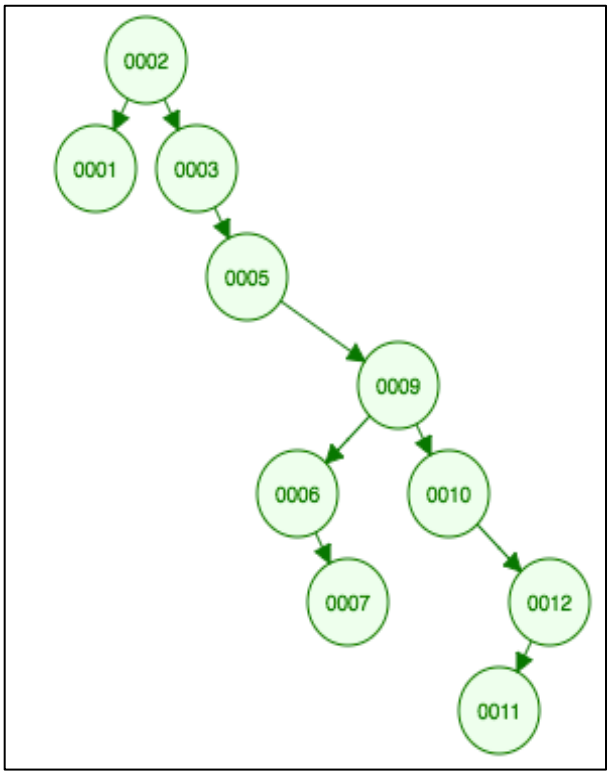
Updated Print Function:

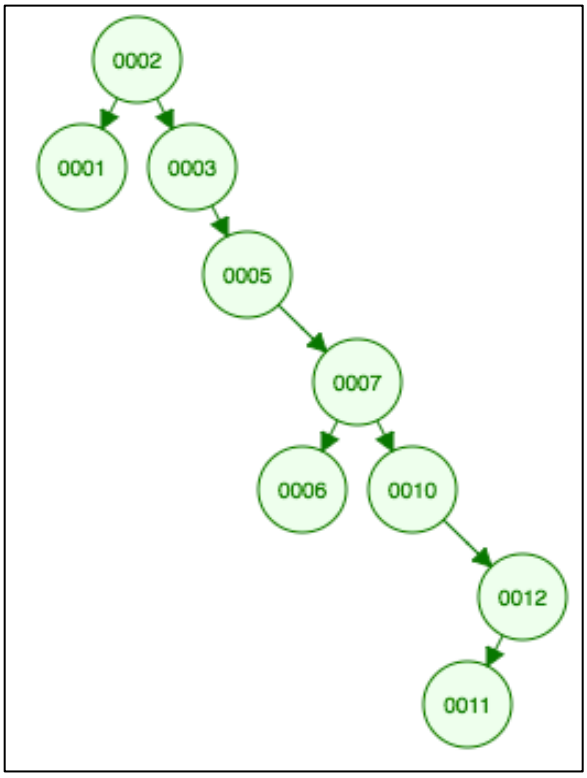
```
    void print() {
        cout << "Node Value: " << value << endl;
        if(p != NULL) {
            cout << "Parent Node Value: " << p->d() << endl;
        }
        if(left != NULL) {
            cout << "Left Child Node Value: " << left->d() << endl;
        }
        if(right != NULL) {
            cout << "Right Child Node Value: " << right->d() << endl;
        }
    }
```

Current Main Output:



2) The next task was to implement the delete_node function and show the results of deleting 4 then 9 from the binary search tree used in part 1. In theory, the binary tree should appear as the following (left picture shows after deletion of 4 and right picture shows after deletion of 9):





The delete_node function was created to check the value parameter and determine if it was smaller, larger or equal to the tree_root node parameter value. The function was set up recursively to call itself until the desired node to be deleted was found. In order to determine how to replace the deleted nodes, conditional statements were used to determine the structure of the nodes left subtree. Once the structure was determined for the subtree, relevant pointers were adjusted and the node was deleted.

Delete_node Function:

```
void delete_node(int value, Node *tree_root){
    if(tree_root == NULL) {
        cout << "Tree is empty" << endl;
    }
    if(value > tree_root->d()) {
        delete_node(value, tree_root->right);
    }
    if(value < tree_root->d()) {
        delete_node(value, tree_root->left);
    }
    if(value == tree_root->d()) {
        //No child nodes.
        if(tree_root->left == NULL && tree_root->right == NULL) {
            delete tree_root;
            cout << "Tree is now empty" << endl;
        }
        //One child node:
        //Right child node.
        if(tree_root->left == NULL && tree_root->right != NULL) {
            tree_root->right->p = tree_root->p;
            tree_root->p->right = tree_root->right;
            delete tree_root;
        }
        //Left child node.
        if(tree_root->left != NULL && tree_root->right == NULL) {
            tree_root->left->p = tree_root->p;
            tree_root->p->left = tree_root->left;
            delete tree_root;
        }
        //Two children nodes, focusing on left subtree.
        else {
            //Left child has both nodes.
            if(tree_root->left->left != NULL && tree_root->left->right != NULL) {
                while(tree_root->left->right->right != NULL) {
                    tree_root->left->p = tree_root->left->right->right;
                    tree_root->right->p = tree_root->left->right->right;
                    tree_root->left->right->right->left = tree_root->left;
                    tree_root->left->right->right->right = tree_root->right;
                    delete_node(tree_root->left->right->right->d(), tree_root->left->right->right);
                }

            }
            //Left child only has right node.
            else if(tree_root->left->left == NULL && tree_root->left->right != NULL) {
                tree_root->left->p = tree_root->left->right;
                tree_root->left->right->left = tree_root->left;
                tree_root->left->right->p = tree_root->p;
                tree_root->p->left = tree_root->left->right;
                tree_root->right->p = tree_root->left->right;
                tree_root->left->right->right = tree_root->right;
                tree_root->left->right = NULL;
                delete tree_root;
            }
            //Left child has no nodes or only left node.
            else {
                tree_root->left->p = tree_root->p;
                tree_root->p->right = tree_root->left;
                tree_root->left->right = tree_root->right;
                tree_root->right->p = tree_root->left;
                delete tree_root;
            }
        }
    }
}
```

Current Main Output:

```
Ace:Unit 4 A$ g++ tree.cpp -o 4
Ace:Unit 4 A$ ./4
Before deletion of 4
Node Value: 2
Left Child Node Value: 1
Right Child Node Value: 4

Node Value: 4
Parent Node Value: 2
Left Child Node Value: 3
Right Child Node Value: 5

Node Value: 3
Parent Node Value: 4

Node Value: 5
Parent Node Value: 4
Right Child Node Value: 9

After deletion of 4
Node Value: 2
Left Child Node Value: 1
Right Child Node Value: 3

Node Value: 3
Parent Node Value: 2
Right Child Node Value: 5

Node Value: 5
Parent Node Value: 3
Right Child Node Value: 9

Before deletion of 9
Node Value: 9
Parent Node Value: 5
Left Child Node Value: 6
Right Child Node Value: 10

Node Value: 6
Parent Node Value: 9
Right Child Node Value: 7

Node Value: 7
Parent Node Value: 6

Node Value: 10
Parent Node Value: 9
Right Child Node Value: 12

After deletion of 9
Node Value: 6
Parent Node Value: 7

Node Value: 7
Parent Node Value: 5
Left Child Node Value: 6
Right Child Node Value: 10

Node Value: 10
Parent Node Value: 7
Right Child Node Value: 12
```

3) The final task was to implement the search function and display the results of searching for 12 and then 4 from the binary search tree resulting from part 2. In theory, the node with value 12 should be found, but 4 should not. The search function determines if the value is larger than the tree_root node, and calls itself recursively until the node with the input value is found or can't be found.

Search Function:

```
130 void search(int value, Node *tree_root){
131     //Your code here
132     if(tree_root == NULL) {
133         cout << "Node " << value << " Can't Be Found" << endl;
134         return;
135     }
136     if(value > tree_root->d()) {
137         search(value, tree_root->right);
138     }
139     if(value < tree_root->d()) {
140         search(value, tree_root->left);
141     }
142     if(value == tree_root->d()) {
143         cout << "Node " << value << " Found" << endl;
144         tree_root->print();
145     }
146 }
```

Current Main Output: