

Assignment 3

CSE 503 – Summer 2024

Richard Douglas

Assignment Description

We were asked to edit code in a file called `tree.cpp` in order to accomplish the following tasks:

1. Implement `insert()` in `tree.cpp` and show the results of inserting 3, 2, 5, 6, 10, 4, 7, 8, 11, 15, 13 into an empty Binary search tree.
2. Start with the tree in problem 1 and implement `delete_node()` and do:
 1. delete 5, then
 2. delete 8.
3. Start with the tree in problem 2 and implement `search()` and do:
 1. Search 7
 2. Search 5

Logic Employed

Alright.... so the original `cpp` file implemented a `Node` class only. Some functions were suggested to complete the tree implementation... I chose to more closely follow the implementation given in our text book. It instead uses a tree class from which you can create tree objects. In the binary search tree class implementation in the text, they use a `struct` to implement `Node` objects within the `BST` class. I chose instead to keep the given `Node` class. However, the functions in my code for insertion, deletion and search are member functions of the `BST` class I created. This is a MAJOR difference from the insinuation the assignment makes.... So I hope there aren't automated tests that check for those functions - particularly because I have a correct (and fairly complete) implementation of a binary search tree.

That all said, the logic of a `BST` is as follows (split into sections):

NODE OBJECTS

Node objects contain a few data members – the data stored in the node, and pointers to the parent and child nodes (if they exist). Nodes can be created to accommodate a wide variety of types, but in this particular case, only ints. Pointers link the nodes of the tree to one another, allowing traversal of the tree (typically done through recursion; *sidenote: recursion call depth depends on tree depth, ideally a log function of the size of input.*

Two kinds of special nodes exist: *root*, and *leaves*. Only one root exists. It is the first node. leaves are simply nodes which do not have child nodes.

THE TREE CLASS

This class consists of several public member functions and a couple of public data members. The public members of this class reference the private members. As I understand it, this is in order to obfuscate the root node from functionality external to the class. The class also contains custom

constructor, copy constructor and destructor. I won't worry about pasting all of the code in here. Particularly not the definition of the BST class itself with all of its members.

Instead, below I have posted the operations we were assigned to implement, as written in my tree.cpp file:

INSERTION

```
void BST::insert(const int & x, Node * & t, Node * p) const
{
    /*
OVERALL LOGIC:
Recursively call insert until the place where 'x' fits into the tree is found
then, in the case where pointer passed into the function call points to NULL,
the new node is created and inserted as the left or right node pointed to.
The last case is the result of the dereferenced pointer being neither NULL value, greater than 'x',
or less than 'x'.
This implies we have found a node with a value equal to x, so we do NOTHING. Recursion stops,
and no node is created to avoid duplication in the tree.
*/

// empty tree check... For me it's helpful to keep in mind that the whole tree is a bunch of
subtrees...
if ( t == NULL)
{
    t = new Node( x );
    t->p = p;
}
// less than check
else if ( x < t->d() )
    insert( x, t->left, t );
// greater than check
else if ( t->d() < x )
    insert( x, t->right, t );
// implicit equality check... could be done explicitly, but it's unnecessary.
else
    cout << "tried to insert a duplicate" << x << ". no duplicates!" << endl; //do nothing
}
```

DELETION

```
void BST::delete_node(const int & x, Node * & t) const
{
    /*
    This one is actually a little funky.
    */
    if ( t == NULL ) // obvi
    {
        cout << x << " not found, so could not be deleted." << endl;
        return;
    }
    if ( x < t->d() ) // delete val lt node val. look left.
        delete_node( x, t->left);
    else if ( t->d() < x ) // delete val gt node val. look right.
        delete_node( x, t->right);
    else if ( t->left != NULL && t->right != NULL )
    {
        t->assign_val(findMin(t->right)->d()); // reassign current node val to the min of the right child
        tree (necessarily still gt current node val)
        delete_node(t->d(), t->right); // delete min node in right subtree. essentially, swapping the least
        val gt current node with current node val. slick.
    }
    else // only remaining case - delete value found and only one subtree exists from that node.
    {
        Node *old = t; // copy pointer to current node for later space freeing.
        t = (t->left != NULL) ? t->left : t->right; // if left pointer is not NULL, assign that object to current
        node. else, assign the right. i.i assign the only subtree to the current node.
        delete old; // free memory from old node (deleting it)
    }
}
```

SEARCH

```
bool BST::search( const int & x, Node *t) const
```

```

{
if ( t==NULL )
return false; // if an empty tree is found, false
else if ( x < t->d() )
return search( x, t->left ); // search val less than current node. look left.
else if ( t->d() < x )
return search( x, t->right ); // search val greater than current node. look right.
else
return true; // match found! total evaluation will always be true.
}

```

I ALSO implemented printing functions for the tree, with indentation. In main, two loops exist which attempt to add the desired values from an array of ints, to our tree. During the loops, sleep commands and “clear” system calls are made to effectively animate the creation of the BST. I use Linux... I think the function I made should work on windows or apple as well though... I think it should even work on old unix systems.

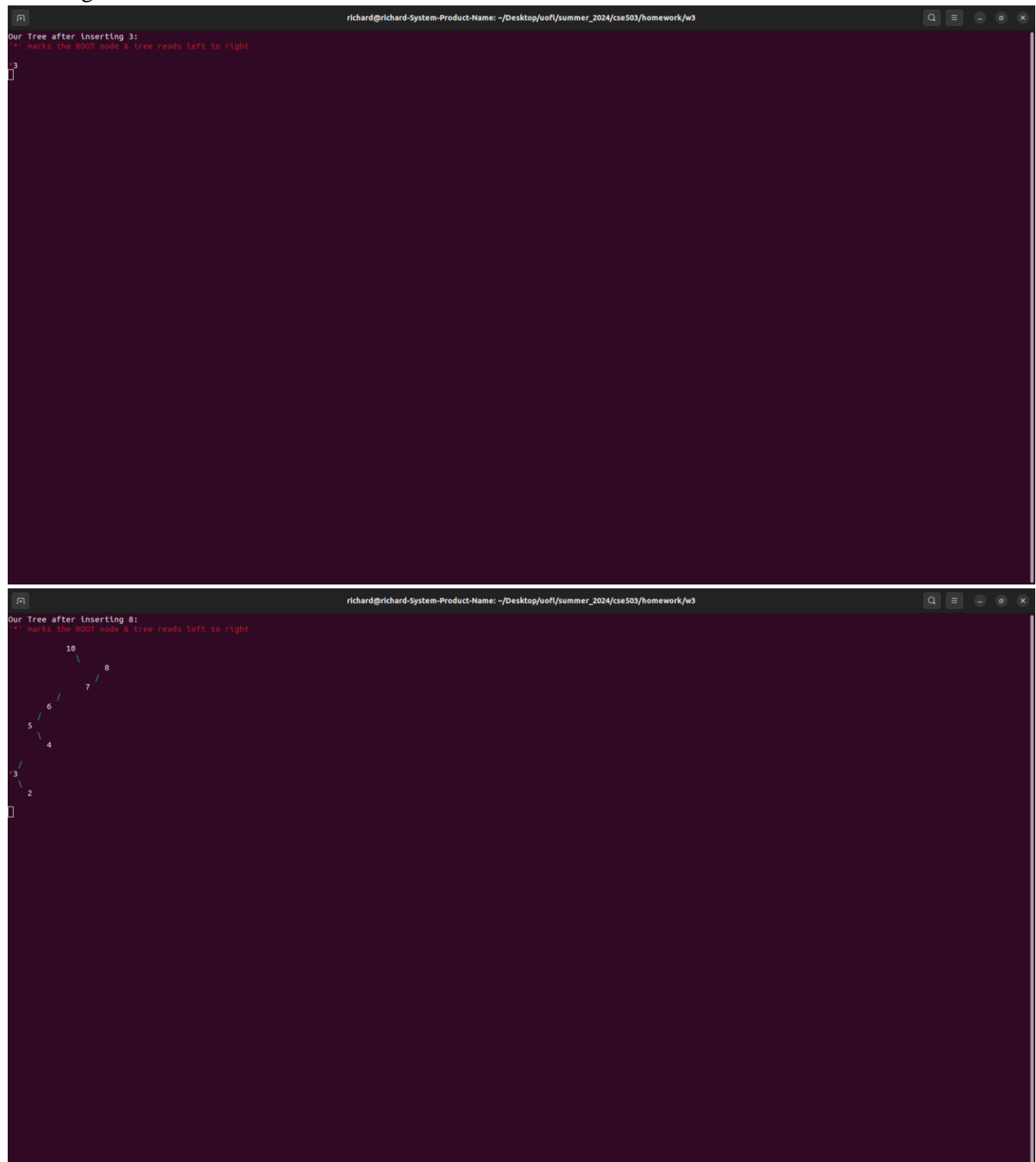
Finally, there is printed output in the terminal if the searched terms are found, to display the tree after deletions, and to show all of the nodes, with their respective pointers and data field.

I/O

The user is not prompted for input. Screenshots

The following is the output run in my terminal.

Building the tree:



```
richard@richard-System-Product-Name: ~/Desktop/uofl/summer_2024/cse503/homework/w3
Our Tree after inserting 3:
** marks the ROOT node & tree reads left to right
3
└─┘

richard@richard-System-Product-Name: ~/Desktop/uofl/summer_2024/cse503/homework/w3
Our Tree after inserting 8:
** marks the ROOT node & tree reads left to right
      10
     /  \
    6    8
   /  \  /
  5   4 7
 /  \
3   2
└─┘
```

Search results:

```
richard@richard-System-Product-Name: ~/Desktop/uofl/summer_2024/cse503/homework/w3
found 7
found 5
}
```

Node details

```
richard@richard-System-Product-Name: ~/Desktop/uofl/summer_2024/cse503/homework/w3
3
 2
 1
 0
 0001
Node ptr: 0x5f9e9c644eb0
Node val: 3
This is the root! No parent!
Right child ptr: 0x5f9e9c645320
Right child val: 6
Left child ptr: 0x5f9e9c6452f0
Left child val: 2
*****
Node ptr: 0x5f9e9c645320
Node val: 6
Parent node ptr: 0x5f9e9c644eb0
Parent node val: 3
Right child ptr: 0x5f9e9c645380
Right child val: 10
Left child ptr: 0x5f9e9c6453b0
Left child val: 4
*****
Node ptr: 0x5f9e9c645380
Node val: 10
Parent node ptr: 0x5f9e9c645350
Parent node val: 1703777109
Right child ptr: 0x5f9e9c645480
Right child val: 11
Left child ptr: 0x5f9e9c6453e0
Left child val: 7
*****
Node ptr: 0x5f9e9c645480
Node val: 11
Parent node ptr: 0x5f9e9c645380
Parent node val: 10
Right child ptr: 0x5f9e9c6454b0
Right child val: 15
No left child!
*****
Node ptr: 0x5f9e9c6454b0
Node val: 15
Parent node ptr: 0x5f9e9c645480
Parent node val: 11
No right child!
Left child ptr: 0x5f9e9c6454e0
Left child val: 13
```

Conclusions

This assignment was probably easier than I made it.... But I had a lot of fun and think it's decent. I now have a VERY solid understanding of binary search trees. Prior to this, I had done some challenges in Leet code related to trees and chose python as my preferred language. Writing data structures in C++

(C would have the same effect, I think) much more clearly demonstrates the logic, and the utility of pointers.