Binary Tree Documentation

By Steven Miller

Department of Computer Science and Engineering

This document lists all functions that are contained within the AVL Tree.

Note that helper functions associated with the following functions are not present in this document.

What I learned from doing this project is that planning is important. A lot of functions that are present within the AVL tree (regardless of this document) were written on the fly. Knowing how you're going to approach a problem in software design is important. If I could go back and start over, I would have planned out how I'm going to write this project a lot differently.

Pre-planning is worthless however without knowledge of the problem you're trying to solve. I did not know much about binary trees nor was I very good at understanding recursive functions before starting this project. Now, while I'm not perfect at it, I'm more comfortable with solving recursive problems.

The format of this document follows:

Function Name

Purpose

Time complexity

```
Node* priv_insert(Node*, std::string const&, std::string const&);
```

This function is responsible for inserting nodes into the tree and for balancing them

This function is has a time complexity of O(1), as all it does it reassign pointers and balance as necessary.
Note that balancing also has a time complexity of O(1).

```
Node* priv_remove(Node*, int const&);
```

This function is responsible for removing nodes from the tree. It has a time complexity of O(log(n)), as searching the tree in-order is necessary when the node has two children.

```
Node* priv_search(int const&, Node*);
```

This function is responsible for searching the tree for a specified integer value. It has a time complexity of O(log(n)), as it needs to search through half of the entire tree top down in the worst case.

```
std::queue<Node*> priv_search(std::queue<Binary_tree::Node*>,
std::string const&, Node*);
```

This function is responsible for searching the tree for a specified name(s). It has a time complexity of O(log(n)), as it needs to search through half of the entire tree top down in the worst case.

```
std::queue< Binary_tree::Node*> get_inorder(Binary_tree::Node*,
std::queue< Binary_tree::Node*>&);
```

This function is responsible for getting all nodes in the tree in-order. It has a time complexity of O(log(n)), as even though it goes through the entire tree, it does it by traversing through one-half of the entire tree, then the other.

```
std::queue< Binary_tree::Node*> get_preorder(Binary_tree::Node*,
std::queue< Binary_tree::Node*>&);
```

This function is responsible for getting all nodes in the tree pre-order. It has a time complexity of O(log(n)), as even though it goes through the entire tree, it does it by traversing through one-half of the entire tree, then the other.

```
std::queue< Binary_tree::Node*> get_postorder(Binary_tree::Node*,
std::queue< Binary_tree::Node*>&);
```

This function is responsible for getting all nodes in the tree post-order. It has a time complexity of O(log(n)), as even though it goes through the entire tree, it does it by traversing through one-half of the entire tree, then the other.

```
void print_level_count();
```

This function is responsible for the total height of the entire tree. It has a time complexity of O(log(n)), as it gets the height of the left subtree, then the right subtree. Both operations require traversing the whole tree.

```
void remove_N(int const&);
```

This function is responsible for removing the Nth node in the tree in-order. It has a time complexity of O(N). This is because it uses a queue to figure out which element to remove. It is possible that the last element in the tree is requested to be removed. Thus, traversing through all nodes in the tree is required.