The GatorAVL Class

```
/* Balancing, Height, And Related Functions */
```

```
Student* RotateLeft(Student*& node);
Student* RotateRight(Student*& node);
```

Computational Complexity

Both of these methods manipulate a constant number of pointers regardless of the size of the AVL tree or the size of the data inside of the tree. Therefore the **best, average, and worst cases** of these two methods is O(1) or constant time.

```
Student* RotateLeftRight(Student*& node);
Student* RotateRightLeft(Student*& node);
```

Computational Complexity

Similarly, to the other rotation methods, these use the functionality of the above, in a manner that does not depend on the size of the list. In each of the above methods two of the basic rotation methods for left and right rotation are called which both have constant running times. Therefore, the **best, average and worse case** time complexity of these methods is also constant time or O(1).

```
short Height(Student* node);
```

Computational Complexity

This method only depends on the size of the data and not the complexity or state of it. It recursively finds the height going down the left and right subtree of each child of a node. Let n be the number of nodes in the subtree where *node* is root. Then the *height* function will pass through every node of the tree to return the result of the height. Therefore, the **best, average, and worst case** performance of this function is O(n).

Things That I Could Have Done Better

When I wrote the height function, I originally thought of making a Boolean value inside of each of the nodes that would determine whether a node is part of a path down which change has been made. There are Boolean values in the *Student* class that are to perform such task. That way the complexity of the heigh function would be O(log n) where n is the number of nodes in a given subtree when it is first called. Then after the first call, all of the nodes in the path where a change has occurred would have an updated height and subsequent calls would then be O(1).

```
bool IsBalanced(Student* node);
```

Computational Complexity

This function calls the *Height* function for the left and right node that is passed in. Let n be the number of nodes in the sub-tree, then *Height* called on the left and right node will both access approximately (1/2)n nodes since *Height* is O(n). This makes the **best, average, and worst case** for the height to be O(n) as well.

```
void Update(Student*& node);
```
Computational Complexity
Letting n be the total number of nodes of the AVL tree. This function only goes down the path where a change occurred. The function recursively calls until the nodes is null or does not have a marker on it. Let n be the number of nodes in the AVL tree, then this function would be O (log n) since it only goes down one branch of the tree for the **best, average and worse case**.

```
void BalanceRecursive(Student*& node);
```
Computational Complexity
Let n be the number of nodes in the tree. The *BalanceRecursive* function, where the first node passed to it is the root, does a postorder traversal. Therefor it makes n accesses. For each of these accesses it calls the function *IsBalanced*, this function is O(m) where m is the number of nodes in the subtree at a given node, and calls other functions that determine where a change has occurred at a given node (which are all O(1). Since m is usually equal to (n/2^(depth)). We know that this function is O(n^2) in the **best, average, and worst case.**

```
void Balance();
```
Computational Complexity
The balance function calls *BalanceRecusive* with the node being root, and then calls the update function. The update function is O(log n) and the balance recursive function is O(n^2) therefore. This function is O(n^2) where n is the number of nodes in the tree in the **best, average, and worst case.**

```
/* Misc And Helpers */
```

```
short Max(short n1, short n2);
```
Computational Complexity
This function takes to values and returns the higher of the two with if statements. The function is constant time in the **best, average, and worst cases.**

```
short AbsoluteValue(short n1);
```
Computational Complexity
This function takes a short and if it is negative returns the negative of the short, if it is positive then it will return the number. Therefore, the function is constant time in the **best, average, and worst case.**

```
bool Right(Student* node);
```

```
bool Left(Student* node);
```

Computational Complexity

Both of these methods check if the node in the direction indicated by the function name is not null and has a marker. This function is O(1) is the **best, average, and worst cases.**

```
bool LeftLeft(Student* node);
bool LeftRight(Student* node);
bool RightLeft(Student* node);
bool RightRight(Student* node);
```

Computational Complexity

These methods make use of the *Left* and *Right* methods. All of these methods work similarly to the following example. Take *LeftRight* for example. It would call *Left* at the current node, then it would call *Right* of the left child of the node. Since *Left* and *Right* are O(1) this would make all of the above also O(1) in **best, average, and worst cases.**

```
void PrintInorderHelper(Student* node, std::string& toPrint);
void PrintPreorderHelper(Student* node, std::string& toPrint);
void PrintPostorderHelper(Student* node, std::string& toPrint);
```

Computational Complexity

These functions all perform these traversals. In each traversal they add the names of the node and recursively call themselves according to their type of traversal. All the actions done inside of each iteration of the function (without recursion) is O(1). The functions hits all of the nodes of the tree. Let n be the number of nodes in the tree. Then the above functions are O(n) in there **best, average, and worst cases.**

```
void PrintChildren(Student* node);
```

Computational Complexity

This method prints the left and right children of the node and it is used for debugging. If the node is null, then it would print that the respective child is null. This methods is O(1) in the **best, average, and worst cases.**

```
void PrintIDofName(Student* node, const std::string& name, std::string
& toPrint);
```

Computational Complexity

This function does a preorder traversal while checking if each node has the same name as *name*. It then adds their name and other formatting to *toPrint.* This function is therefore O(n) where n is the length of the tree.

```
void FindInOrderNthID(Student* node, unsigned short& n, int& id);
```

Computational Complexity

This function in the **best, average, and worst cases** performs at O(n) where n is the size of the list. That is the *n*th node in the list by the Inorder traversal. If *n* is larger than the length of the list, then it would stop once it traverses all of the nodes in the list.

```
void MarkAll(Student* node);
```

Computational Complexity
This method performs at O(n) where n is the length of the list in the **best, average, and worst cases.**

```
void MarkRight(Student* node);
void MarkLeft(Student* node);
```

Computational Complexity
These functions only get evoked in a special case in the Remove function. When the new root after removal does not have a left or right node. This would mark all the nodes to the left if the root does not have a right node, and vice versa. Let n be the number of elements in the tree. Since this function only goes down only one branch in the tree, then the **best, average, and worst case** performance is O(log n).

```
td::string IdTo8Digit(int id);
```

Computational Complexity
This function no matter the input performs in constant time. It converts an ID to a string and then appends zeros to the front of the string until it is 8 characters long. That is this function is O(1) in the **best, average, and worst case.**

```
/* Functionality */
```

```
void InsertNameID(const std::string& name, int id);
```

Computational Complexity
Let n be the length of the tree. The insertion process itself is O(log n) since this function only travels on one branch of the whole tree. However, it calls the balance function at the end, therefore the time complexity is O(n^2) in the **best, average, and worst case.**

```
void RemoveID(int id);
```

Computational Complexity
Let n be the length of the tree. Finding the node to remove is O(log n). If the nodes has two children then finding the Inorder successor is also O(log n). This is because both of the processes go through only one branch of the AVL tree. However, this function also calls balance at the end which is O(n^2). Therefore, the computational complexity is O(n^2) for the **best, average, and worst case.**

Thing That I Could Have Done Better
Prior to implementing this function, I was not sure of all of the cases that could occur in my program. Because of this, I did not have the time to compartmentalize code. This is one thing I would definitely change if I were to do this again. For example, when I find the inorder successor, I do it inside of this function when I should be doing it in another function that is called by this one.

```
void SearchID(int id);
```

Computational Complexity
Let n be the length of the tree. This function is then O(log n) since it only travels down one branch of the tree for the **best, average, and worst case**.

```
void SearchName(const std::string& name);
```

Computational Complexity
Let n be the number of nodes in the tree. This function uses a preorder traversal thus the **best, average, and worst case** complexity is O(n).

```
void PrintInorder();
void PrintPreorder();
void PrintPostorder();
void PrintLevelCount();
```

Computational Complexity
Similar to the above. Let n be the number of nodes in the tree. Each of these traversals visits each node. Therefore, the **best, average, and worst case** complexity is O(n)

```
void RemoveInorder(unsigned short n);
```

Computational Complexity
Let n be the number of nodes in the tree and let *n* signify the short input into the function. This function calls the *FindInOrderNthID* which is O(n). But since is calls the function that removes a node, which is O(n^2). The **best, average, and worst case** complexity if O(n^2).

What I Learned

When I implemented this class, I did not really have a grasp at how the rotations will in the end interact with a function that balances. Towards the end however, I feel like I understand how the inner workings of an AVL tree work together to keep the BST self-balancing. Finally, I understood how references to pointers work in C++, and how they can be used to switch to pointers in a cleaner fashion.