**Program Assn.5 – Analysis**

**Functions (except Main):**

CreateTree – This function will execute the exact same number of statements every time it is called.

* Average: **f(n) = x** because it performs the same every time.
* Worst: **f(n) = x** because it performs the same every time.

GetFileName – This function contains a loop and its efficiency depends on the user’s input. Theoretically, the user could enter an invalid entry forever.

* Average: **f(n) = n** efficiency because the user could enter the correct file name the first time. I think this would happen on average. N is directly proportional to the number of times the user inputs the wrong file name, but it will always run at least 1 time.
* Worst: **f(n) = n!** (factorial) efficiency because the user could continue getting the file name wrong forever and the program never proceed. N is directly proportional to the number of times the user inputs the wrong file name.

IsEmpty – This function will only add 1 extra line of code if the binary tree is empty.

* Average: **f(n) = x** efficiency, because most of the time the binary tree will not be empty, therefore, this function will execute the same number of statements every time.
* Worst: **f(n) = x + 1** efficiency, because it n depends on if the binary tree is empty or not. The worst case would be if it is empty, it would perform exactly 1 more piece of code.

CreateNode – This function will execute the exact same number of statements every time it is called.

* Average: **f(n) = x** because it performs the same every time.
* Worst: **f(n) = x** because it performs the same every time.

InsertNode – This function must search the binary search tree with a loop to find the proper insertion point.

* Average: **f(n) = log n** because in a balanced tree, the algorithm will break the tree down into simpler parts while searching.
* Worst: **f(n) = n** efficiency because the tree would be unbalanced and the algorithm would be forced to search every node in the tree before inserting a new node.

FindNode – This function must also search the binary search tree with a loop.

* Average: **f(n) = log n** because in a balanced tree, the algorithm will break the tree down into simpler parts while searching.
* Worst: **f(n) = n** efficiency because the tree would be unbalanced and the algorithm would be forced to search every node in the tree before finding the correct node.

Menu – This function contains a loop and its efficiency depends on the user’s input. Theoretically, the user could enter an invalid entry forever.

* Average: **f(n) = n** efficiency because the user could enter the correct file name the first time. I think this would happen on average. N is directly proportional to the number of times the user inputs the wrong file name, but it will always run at least 1 time.
* Worst: **f(n) = n!** (factorial) efficiency because the user could continue getting the file name wrong forever and the program never proceed. N is directly proportional to the number of times the user inputs the wrong file name.

DeleteNode – This function is the most difficult to determine its efficiency. It contains two loops and many conditional statements. However, even though there are two loops, the efficiency is still proportional to its size.

* Average: **f(n) = log n** because in a balanced tree, the algorithm will break the tree down into simpler parts while searching for the correct node to delete.
* Worst: **f(n) = n** efficiency because the tree would be unbalanced and the algorithm would be forced to search every node in the tree before deleting a node.

InOrderDisplay This function will visit every node in a tree / subtree, depending on the root given.

* Average: **f(n) = n** because its efficiency is directly proportional to the number of nodes the root contains.
* Worst: **f(n) = n** because its efficiency is directly proportional to the number of nodes the root contains.

GetInteger - – This function contains a loop and its efficiency depends on the user’s input. Theoretically, the user could enter an invalid entry forever.

* Average: **f(n) = n** efficiency because the user could enter the correct file name the first time. I think this would happen on average. N is directly proportional to the number of times the user inputs the wrong file name, but it will always run at least 1 time.
* Worst: **f(n) = n!** (factorial) efficiency because the user could continue getting the file name wrong forever and the program never proceed. N is directly proportional to the number of times the user inputs the wrong file name.

FreeNodes – This function will visit every node in the binary search tree, recursively.

* Average: **f(n) = n** because its efficiency is directly proportional to the number of nodes the tree contains.
* Worst: **f(n) = n** because its efficiency is directly proportional to the number of nodes the tree contains.

DestoryTree – This function will execute the exact same number of statements every time it is called.

* Average: **f(n) = x** because it performs the same every time.
* Worst: **f(n) = x** because it performs the same every time.

**Main Function:** Given the dominant factor in the average cases and the unlikeliness of the worst case, I believe the **BigO** efficiency of this program is **O(n)**.

* Average: **f(n) = n** because it is the dominant factor across the board. The program mostly depends on the size of the binary tree.
* Worst: **f(n) = n!** because, theoretically, the program could go on forever if user inputs invalid entries.