CMPT 225: Assignment 3 Report

Program Specifics:

To perform this experiment I collected data samples of a BST and AVL tree at various sizes to demonstrate their varying efficiency as the number of keys increases. The operations were performed on tree sizes of 1000, 10000, 100000 and 1000000 keys. For each tree I performed 10000 operations in the same order for each of the tree instances. These operations were generated with the rand() function that was given a hard-coded initializer in order to generate the same sequence of the operations each time. The keys selected for each operations were chosen from a uniform distribution generator that varied from 0 to the maximum key for that tree.

To generate the initial keys that were inserted into the trees I used the following steps:

1. Start with initial key defined as MAX_KEY / KEY_GAP. This was 1 for the 1000 key tree.
2. For all of the keys that need to be inserted (1000, 10000, 100000 or 1000000):
    1. Generate a new key by adding the key gap to the original key and finding the remainder of the division between this value and the maximum key value.
    2. If this key is already in the trees then find the next key in the sequence.
    3. If the key is not already in the trees then insert it.

The sequence of keys that was inserted varied slightly based on the tree size. For example for the tree of 100000 keys the MAX_KEY was 1000000 and so the initial key was 115 instead of 1. The sequence of operations remained the same. The MAX_KEY tree size increased in order to ensure there were enough unique keys to insert into the tree. The MAX_KEY was always 10 times larger than the size of the tree.

To record information about the operations taking place I performed a set sequence of 10000 operations as described above and recorded information about each tree every 5 operations that took place. At every 5 operations I recorded the operation number, the current size of the tree, the height of the tree, the average node depth, the time of the operation being performed and (in the case of a search operation) the result (found or not found).

Plots:

To compare the results of these two trees I averaged the relevant data values over the 10000 operations and then compared how the two trees compared when they have different numbers of keys. This approach generates useful information that will effectively allow me to answer the question of if the balancing operations make the AVL faster or if they just slow the tree operations down overall. Understanding the runtime complexity of the various operations led to this experimental design. Since the balancing operations of O(1) they will generally take the same amount of time regardless of tree size. Meaning an AVL tree is expected to take extra time on inserts and remove operations for smaller trees. However, the search, remove and insert operations are proportional to the height of the tree in both the

AVL and BST which means a balanced tree will be more efficient. This efficiency is most evident in cases where the number of keys are large.

To generate the charts below I performed the following averaging operations using excel:
Height of Tree: I averaged the height of the tree over all 10000 operations performed.
Average Node Depth: I averaged the average node depth computed over all 10000 operations performed.
Average Operation Time: I averaged the operation time for each operation type within each tree size.

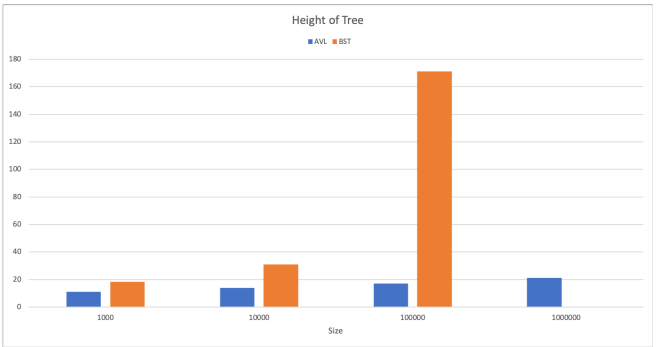


FIGURE 1: HEIGHT OF TREE

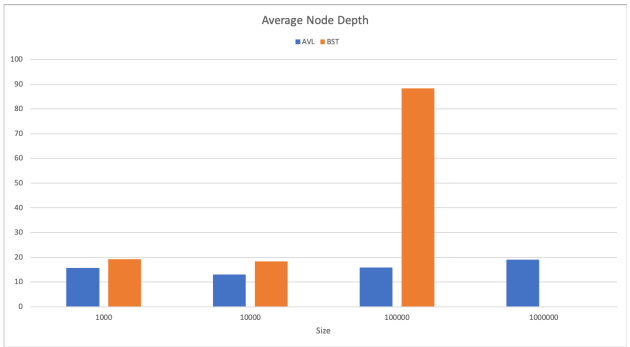


FIGURE 2: AVERAGE NODE DEPTH

Note: The BST height of tree is excluded in these graphics in order to allow a reasonable scale of the smaller tree values. The complete charts can be seen in Appendix of this report.

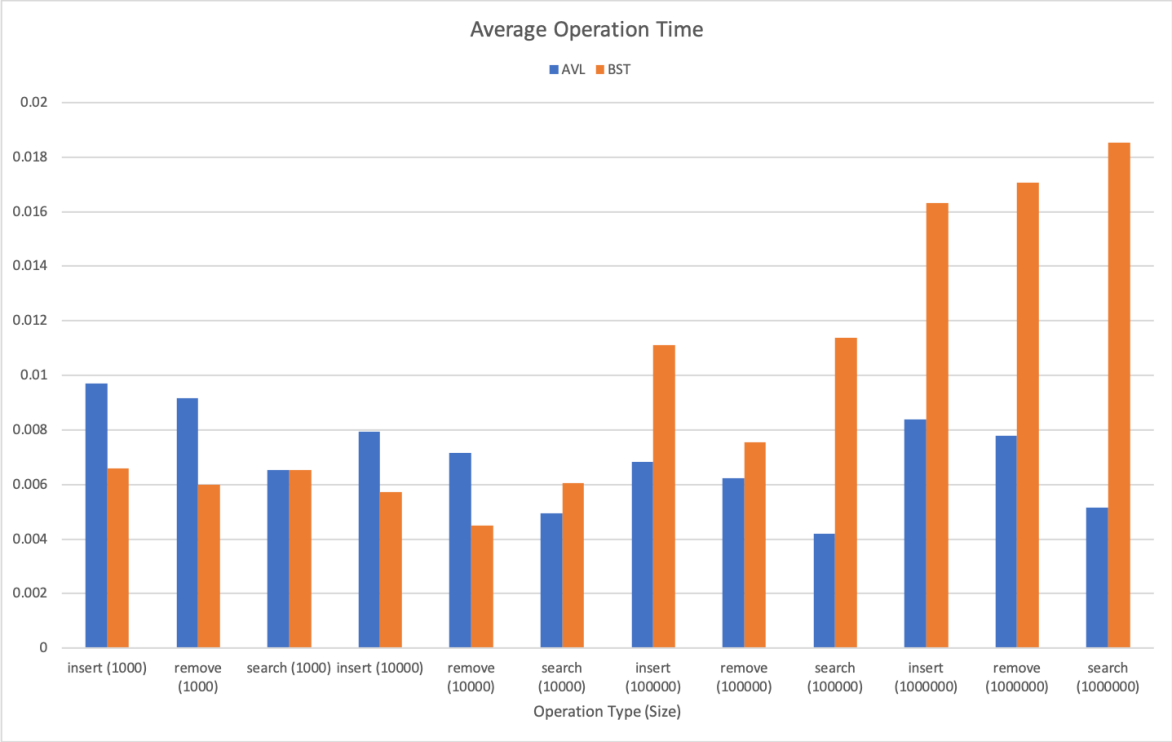


FIGURE 3: AVERAGE OPERATION TIME

Observations:

As seen Figure 1 it is very clear that the AVL tree is much shorter at every size of tree. At a tree of just 1000 keys the height of the AVL tree is nearly half of that BST tree (10.99 vs 18.31). While this is significant at such a small tree it does still not have much impact on the operation time. As seen in Figure 3 the average operation time for insertions and removals is greater for the AVL tree compared to the BST. This is a result of the constant time balancing operations that have to be performed as mentioned above.

As the number of keys increases the height of the BST tree quickly gets out of hand and becomes orders of magnitude larger than the AVL tree. At a tree size of 100000 keys the heights of AVL and BST trees are 17.02 and 171.18 respectively. This is significant as this difference begins to show itself in the operation times where the AVL tree is faster at all 3 operations and significantly faster (175%) at searches. This trend continues as the number of keys grows to 10000000 and the search speed is 260% faster and even insertions and removals are twice as fast (100% and 140% faster respectively to be exact).

The average node depth presents much of the same information as the height of the trees but provides one additional piece of information that explains the differences in the tree behaviours. The AVL tree in general had average node depths very close to the height of the trees. This makes sense since the trees are 'short and bushy'. In the BST case the depths of the nodes are around half the height of the tree indicating the trees are tall are scarce. The average node depth measure gives the best indication of the time it will take for an operation to complete since it shows the average number of levels you will have to traverse to access, insert or remove the key of interest.

The benefits of an AVL tree vs a BST are most clear when you observe their differences in operation speed. While the AVL may be slower at insertions and removals for tree sizes less than 100000 keys they are fasters at searches at every tree size. Choosing the correct tree for a specific application is important and based on the results presented some clear guidelines could be established such as:

- Are you mainly doing searches on the tree and few insertions and removals in comparison? **Choose BST**.
- Is your tree used for 10000 keys or fewer and the main operations are insertions and removals? **Choose BST**

Summary of Observations:

- The height of the BST grows significantly faster than the AVL tree.
- The average node depth of the trees indicate the speed of each operation and grows significantly faster with additional keys in the BST compared to the AVL tree.
- The average operation times are slower for insertions and removals at smaller tree sizes (<10000 keys) due to the constant time balancing operations needed.
- Search operations are always faster on an AVL tree and that speed advantage only grows as the number of keys increases.
- Choosing the correct tree for your application is based on the use of the tree. For small trees (<10000 keys) where insertions and removals are the most used operations, a BST is the best choice. In all other cases the AVL tree is the better choice.

List of files included:

Assignment3_DataAnalysis.xlsx — This excel file contains the data files for each experiment including the averaging formulas and generation of the plots contained in this report.

dataFiles — folder containing all of the outputted files from each trial. Each subfolder contains 3 files. 2 .csv files which report the operation number, size of tree, height of tree, average node depth, operation time and operation type. The 3rd file is a text file containing every operation performed on the tree specified by the operation number, operation type and the key used for the operation, and in the case of a search the result of the search where 0 is not found and 1 is found.

A3_Report.pdf — This report.

src —The source code used to generate the results used in this experiment.

AvlTree.h — AVL tree library. Contains functions for computing ipl() and height().
BinarySearchTree.h — BST tree library. Contains functions for computing ipl() and height().
dsexceptions.h — header file used by library to throw exceptions.
Makefile — file used to compile and clean the project directory.
main.cpp — main file used to generate experimental results.

Instructions for running program:

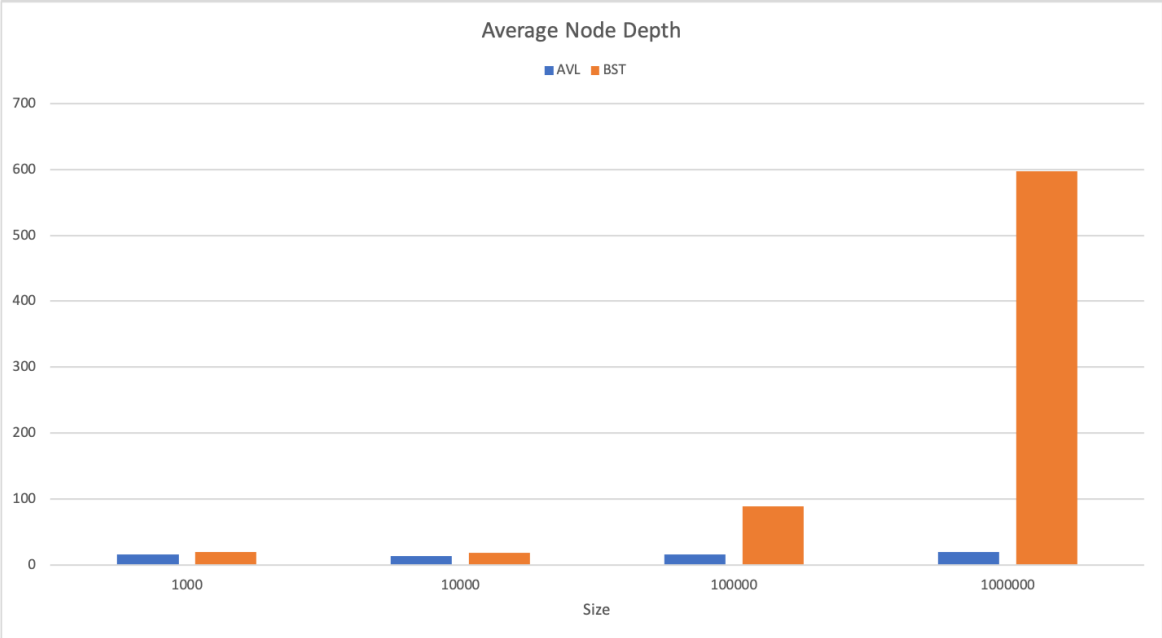
This program will generally run as it is expected to with no modifications. Using 'make all' will compile a version of the program that will produce results for a tree of 100000 keys.

Modifications to change experiment parameters:

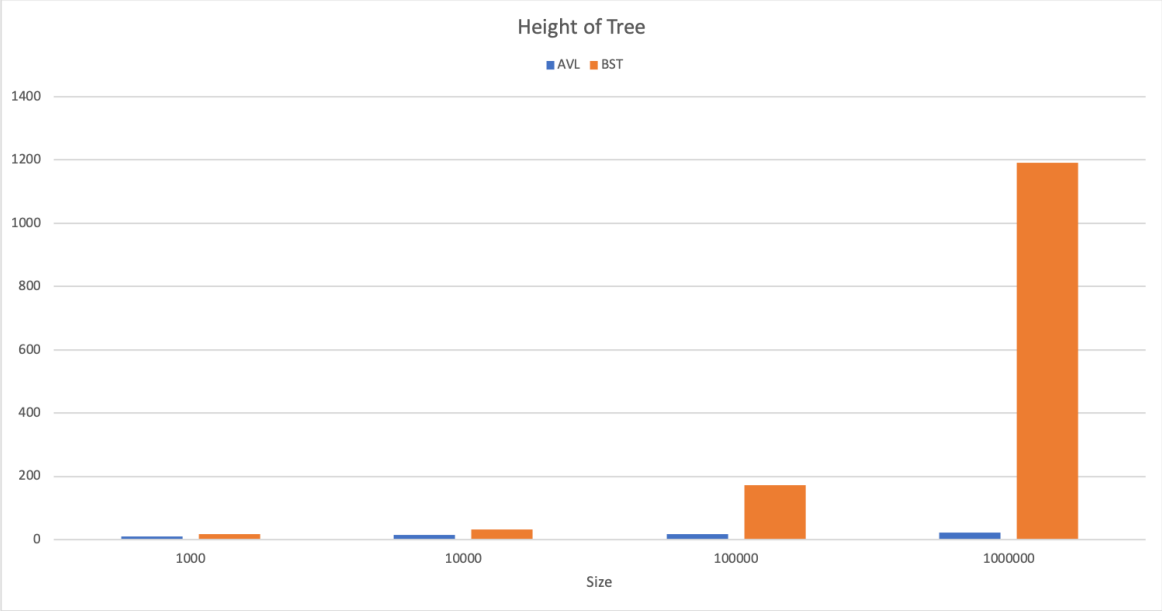
- To change the tree sizes, modify the constant defined on line 26, variable name: NUM_KEYs.
- To change the maximum key values, modify the constant on line 28, variable name: MAX_KEY.
- To change the number of operations performed, modify the constant on line 34, variable name: MAX_OPERATIONS.

Appendix:

Complete plots of height of binary tree and average node depth:



AVERAGE NODE DEPTH — COMPLETE DATA



HEIGHT OF TREE — COMPLETE DATA