CHRIS KATEERA
KTRKUD001
ASSIGNMENT 2
April 29, 2020

CSC2001 DATA STRUCTURES & JAVA

UNIVERSITY OF CAPE TOWN
IYUNIVESITHI YASEKAPA • UNIVERSITEIT VAN KAAPSTAD

COMPARISON OF OPERATIONS IN A BST TREE AND AN AVL TREE DATA STRUCTURE

INTRODUCTION

From a traditional array we arrived at a binary search tree that is more efficient at storing and retrieving data. We now explore an additional data structure type called the AVL tree, to compare its storage and searching operations' efficiency with a Binary Search Tree. An AVL Tree is a Binary Search Tree with an additional condition: a balancing criterion which states that difference between height of left sub-tree and right sub-tree of any node can't be more than 1. This condition drastically reduces the height of the AVL Tree compared to that of the BST Tree. Since the worst-case operations of a binary tree are proportional to the height of the tree – which is related to the size (number of nodes) of the tree, this increases the efficiency of the insertion and searching operations in an AVL Tree. For the AVL Tree, balancing is retained by rotations after each operation. We experiment to prove this.

In this experiment the aim is to compare the Binary Search Tree with an AVL Tree data structure in terms of time complexity for various key operations such as insertion and searching of keys relating to the load shedding raw data we have been supplied with. Both BST and AVL being binary trees, their insertion and searching algorithms' time complexity is proportional to the height of the tree. However, The AVL tree height does not grow as fast as the BST height because of the balancing process. In the worst case of sorted data input, the BST can be linear while the AVL Tree will be balanced. The balancing operation (single and double rotations) are constant O(1) time operations because they only change the pointers of the children tree nodes but do not move actual data. Because of this, they do not affect the time as the data set grows in our asymptotic analysis but have a constant time offset so they can be considered negligible. The experiment intends to demonstrate this. Below is a guide you through the procedure.

THE EXPERIMENT

In order to do this experiment, I initially had to use the same dataset as a control for the experiment – the Cape Town Load shedding data. I made two applications: the AVLApp and the LSBSTApp. I read input from a file into my storage data structures and I implemented a lookup algorithm for the AVLApp program and the LSBSTApp program. After testing and making sure they were working, I then instrumented both programs. This meant I could count the number of steps it took for me to find a key I am searching for. This entailed having a variable increment on each key comparison. I then ran this with the original dataset, then made subsets of differently sized data (using a bash script) so that I could plot and get an indication of the correlation between key comparisons taken to find the same key on both programs and the data size. The variation of the comparisons and the data size gave an indication of the asymptotic growth of the comparisons with increasing data size. I then plotted my findings and discussed the results.

PROGRAM DESIGN

I created a package with two new classes: the AVLApp and the LSBSTApp. I also added the BinarySearchTree classes to the same package: BinarySearchTree, BinaryTree, BTQueue, BTQueueNode, AVLTree and an updated version of the BinaryTreeNode including the height variable. The AVLApp and the LSBSTApp inserted and stored each line of the load shedding data as the data in the tree nodes, each according to its insertion algorithm, AVL insert and BST insert respectively. The LSBSTApp uses insertion and lookup methods contained in the BinarySearchTree class. The AVLApp uses insertion and lookup methods contained in the AVLTree class. There are insertion and finding variables located in LSBSTApp and AVLApp which are incremented, accessed and used in the respective comparison counting. AVLTree inherits attributes and methods from the BinaryTree. I implemented OOP by making use of separate classes with working methods and variables – some of which I then re-used without compromising security. Each object was stored in the tree and was retrieved when it was required.

BST Test Results

```
------------START-OF-OUTPUT-----------------
Value of BST insertCount is:2976
Value of BST height is: 206
Arguments: No arguments passed
1_10_00 15
1_10_02 16
1_10_04 1
1_10_06 2
1_10_08 3
1_10_10 4
```

…

```
8_9_06 6, 14, 2, 10, 7, 15, 3, 11
8_9_08 7, 15, 3, 11, 8, 16, 4, 12
8_9_10 8, 16, 4, 12, 9, 1, 5, 13
8_9_12 9, 1, 5, 13, 10, 2, 6, 14
8_9_14 10, 2, 6, 14, 11, 3, 7, 15
8_9_16 11, 3, 7, 15, 12, 4, 8, 16
8_9_18 12, 4, 8, 16, 13, 5, 9, 1
8_9_20 13, 5, 9, 1, 14, 6, 10, 2
8_9_22 14, 6, 10, 2, 15, 7, 11, 3
2976
------------END-OF-OUTPUT-----------------
```

First 10 and last 10 lines printed when PrintAllAreas is invoked - when no parameters are passed for BST

```
------------START-OF-OUTPUT-----------------
Value of BST insertCount is:2976
Value of BST height is: 206
Arguments: 2 29 00
2_29_00 4,12
Value of BST findCount is:32
------------END-OF-OUTPUT-----------------
------------START-OF-OUTPUT-----------------
Value of BST insertCount is:2976
Value of BST height is: 206
Arguments: 4 17 10
4_17_10 6, 14, 2, 10
Value of BST findCount is:68
------------END-OF-OUTPUT-----------------
------------START-OF-OUTPUT-----------------
Value of BST insertCount is:2976
Value of BST height is: 206
Arguments: 8 10 22
8_10_22 14, 6, 10, 2, 15, 7, 11, 3
Value of BST findCount is:188
------------END-OF-OUTPUT-----------------
------------START-OF-OUTPUT-----------------
Value of BST insertCount is:2976
Value of BST height is: 206
Arguments: 3 5 7
Areas not found.
------------END-OF-OUTPUT-----------------
------------START-OF-OUTPUT-----------------
Value of BST insertCount is:2976
Value of BST height is: 206
Arguments: Invalid number of arguments
------------END-OF-OUTPUT-----------------
------------START-OF-OUTPUT-----------------
Value of BST insertCount is:2976
Value of BST height is: 206
Arguments: invalid text input
Areas not found.
------------END-OF-OUTPUT-----------------
```

Test Results when various parameters are passed from the bash script. – for BST

AVL Test Results

```
------------START-OF-OUTPUT------------------
Value of AVL insertCount is:2976
Value of AVL height is: 12
Arguments: No arguments passed
1_10_00 15
1_10_02 16
1_10_04 1
1_10_06 2
1_10_08 3
1_10_10 4
```

…

```
8_9_06 6, 14, 2, 10, 7, 15, 3, 11
8_9_08 7, 15, 3, 11, 8, 16, 4, 12
8_9_10 8, 16, 4, 12, 9, 1, 5, 13
8_9_12 9, 1, 5, 13, 10, 2, 6, 14
8_9_14 10, 2, 6, 14, 11, 3, 7, 15
8_9_16 11, 3, 7, 15, 12, 4, 8, 16
8_9_18 12, 4, 8, 16, 13, 5, 9, 1
8_9_20 13, 5, 9, 1, 14, 6, 10, 2
8_9_22 14, 6, 10, 2, 15, 7, 11, 3
2976
------------END-OF-OUTPUT-----------------
```

First 10 and last 10 lines printed when PrintAllAreas is invoked - when no parameters are passed for AVL

```
------------START-OF-OUTPUT------------------
Value of AVL insertCount is:2976
Value of AVL height is: 12
Arguments: 2 29 00
2_29_00 4,12
Value of AVL findCount is:9
------------END-OF-OUTPUT-----------------
------------START-OF-OUTPUT------------------
Value of AVL insertCount is:2976
Value of AVL height is: 12
Arguments: 4 17 10
4_17_10 6, 14, 2, 10
Value of AVL findCount is:12
------------END-OF-OUTPUT-----------------
------------START-OF-OUTPUT------------------
Value of AVL insertCount is:2976
Value of AVL height is: 12
Arguments: 8 10 22
8_10_22 14, 6, 10, 2, 15, 7, 11, 3
Value of AVL findCount is:12
------------END-OF-OUTPUT-----------------
------------START-OF-OUTPUT------------------
Value of AVL insertCount is:2976
Value of AVL height is: 12
Arguments: 3 5 7
Areas not found
------------END-OF-OUTPUT-----------------
------------START-OF-OUTPUT------------------
Value of AVL insertCount is:2976
Value of AVL height is: 12
Arguments: Invalid number of arguments
------------END-OF-OUTPUT-----------------
------------START-OF-OUTPUT------------------
Value of AVL insertCount is:2976
Value of AVL height is: 12
Arguments: invalid text input
Areas not found
------------END-OF-OUTPUT-----------------
```

Test Results when various parameters are passed from the bash script – for AVL.

```
Notes on scripts used:

1.Assignment2_AVL/pt1to4script.sh : produces part 1 to part 4 output bst_part1_2.txt and
                                    avl_part3_4.txt
2.Assignment2_AVL/subset.sh       : produces the 10 subset data sets
3.Assignment2_AVL/pt5avl.sh       : automates the searching of each data line in each of the
                                    datasets for AVL
4.Assignment2_AVL/pt5bst.sh       : automates the searching of each data line in each of the
                                    datasets for BST
5. Assignment2_AVL/Makefile       : Regular Makefile with targets bst and avl with optional
                                    parameters that can be passed for searching.
```

FINAL RESULTS & DISCUSSION

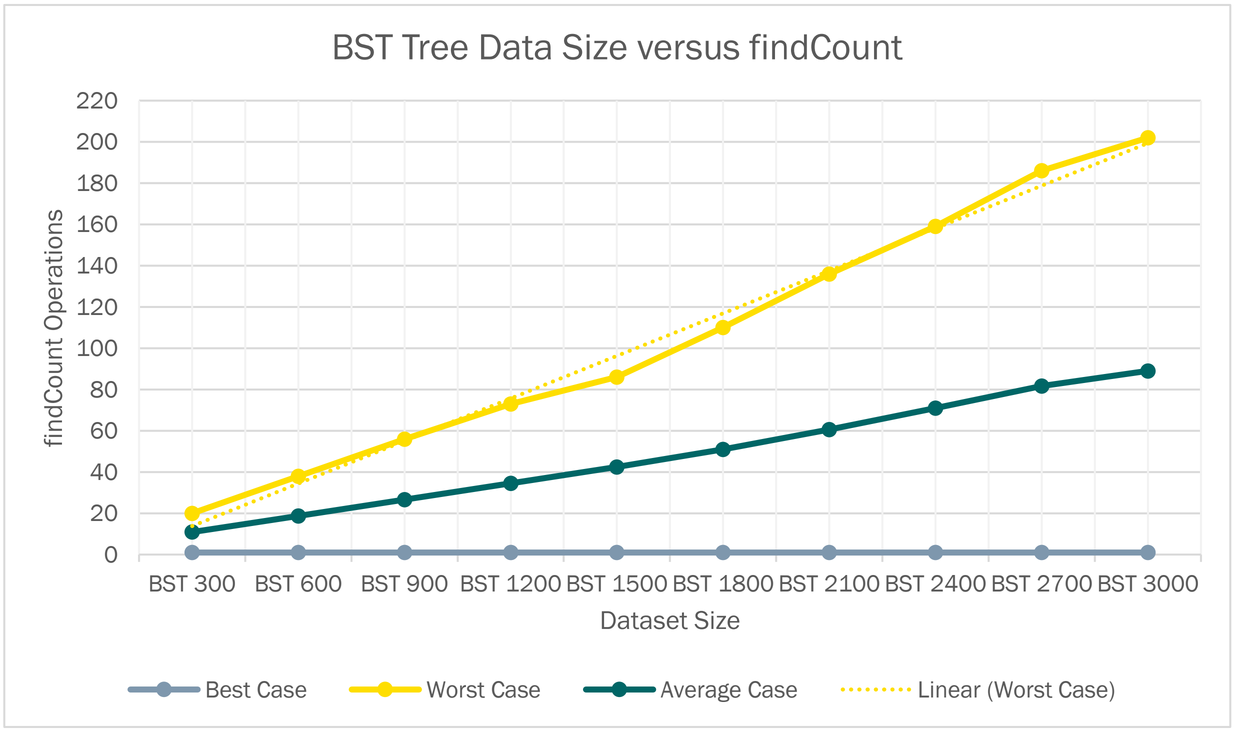

Figure 1 BST Tree Graph showing best case, worst case, average case

As seen in Figure 1, the number of findCount operations is growing approximately linearly with the increasing data size. There is a positive correlation between the comparison operations and the data size. This is because the height of the tree is growing as the number of elements are increasing. This in turn increases the time it takes to find a key – in the worst-case the algorithm must linearly go through all elements till the last one to find it. Therefore, the BST algorithm's worst-case scenario has a time complexity of O(n). The average-case has a logarithmic time complexity of O(logn). The best case is 1 with the key element at the tree root.

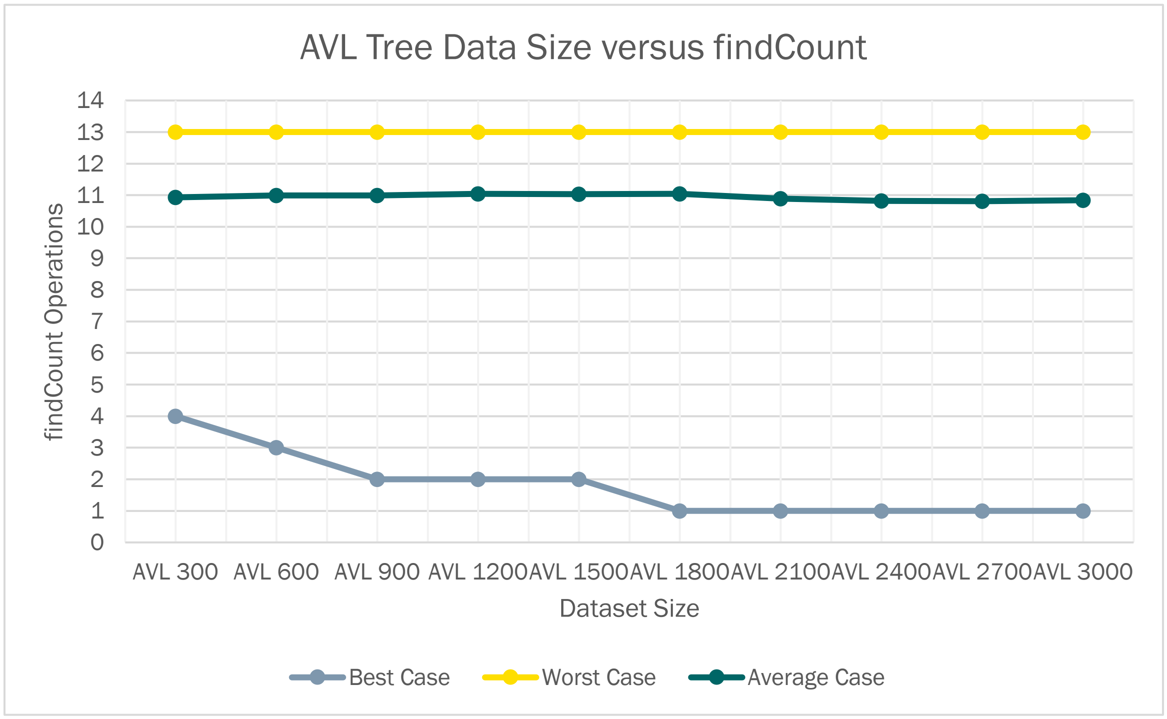

Figure 2 AVL Tree Graph showing best case, worst case, average case

As seen in Figure 2, The worst-case scenario when the tree is of its maximum height of 12, it goes through 13 findCount steps (including the root node). The average case is roughly 11 with the best case being 1: when the searched item is the root node. As expected, the worst case slowly grows at a rate of O(logn) and its not visible on the graph but with even larger data sets, that is the general trend line. This is because each time an AVL Tree is built, it is balanced which reduces the height of the tree. If n is the data size and number of nodes, since an AVL is a binary tree, n is an exponential function of the height and inversely, the height is a log function of the number of nodes. Therefore, its worst-case and average case asymptotic time complexity which is proportional to the height then becomes O(logn) - which is more efficient than BST's O(n).

TREE BUILDING

In the worst case, in the construction of a BST, the algorithm inserts an element one after another linearly. BST insert is O(n) but when the elements are being inserted linearly, the comparisons go through each previously added elements and thus deteriorates to O(n^2).

On the other hand, for the balanced AVL, insertion of 1 element takes O(log(n)) and inserting n items takes O(nlog(n)). This follows the fact that rotations take constant time O(1).

PRACTICAL APPLICATIONS

Since AVL Trees are quicker at searching, a good practical application would be one that there are few insertion and deletion processes, and a short search time is required. An example would be a shopping mall self-service kiosk which has a set number of shops with their respective information. Usually the shops rarely change year on year – few insertion and deletions. But customers need information about shops immediately when typing in the shop name in near realtime. An AVL implementation would be more suitable. However, other data structures such as B+ Trees used in Mysql databases are more optimized for insertion, deletion and retrieval of information.

CREATIVITY

1. I used bash scripting to automate the 10 susbset creation in a file called susbset.sh.
2. I created a Makefile that accepts parameters in a string called params that would then be sent to my program. E.g. make run params="2 4 6"
3. I created a bash script to automate the testing and lookup of the subset values then printing the results to a file via stderr redirection.
4. I counted the insertion comparisons when inserting to a Binary Search Tree and the AVL Tree.
5. I used a bash script to do the testing of part1 to part4 and stored the data in a file.
6. I discussed the of building of a BST vs AVL in terms of time complexity.
7. I discussed practical applications of AVL trees.

GIT SUMMARY STATISTICS

```
0: commit 4eb34b12447a733644b866dbd76d221310625730
1: Author: Chris Kay <kckateera@gmail.com>
2: Date: Thu Apr 30 08:51:39 2020 +0200
3:
4: Accurate AVL Data
5:
6: commit e5f04f5cff13a3a72596a03b7a74dd7564764eb5
7: Author: Chris Kay <kckateera@gmail.com>
8: Date: Thu Apr 30 07:18:51 2020 +0200
9:
...
163: commit b18002cdd4a324e475a9547e9f467c6e07eb5234
164: Author: Chris Kay <kckateera@gmail.com>
165: Date: Wed Apr 29 15:41:12 2020 +0200
166:
167: Packages renaming
168:
169: commit 3a28d52c531ae8f20325a2d98989c378aa6e03fb
170: Author: Chris Kay <kckateera@gmail.com>
171: Date: Wed Apr 29 15:29:33 2020 +0200
172:
```

Git summary statistics

CONCLUSION

The experiment was a success and rigorously compared the Binary Search Tree to an AVL Tree data structure. The results proved the AVL Tree has a faster worst-case asymptotic time complexity of O(logn) compared to the BST with a slower worst-case O(n) time complexity in both the insertion and searching operations. In conclusion, it is better to use an AVL Tree for applications requiring more of searching operations as it is more efficient.