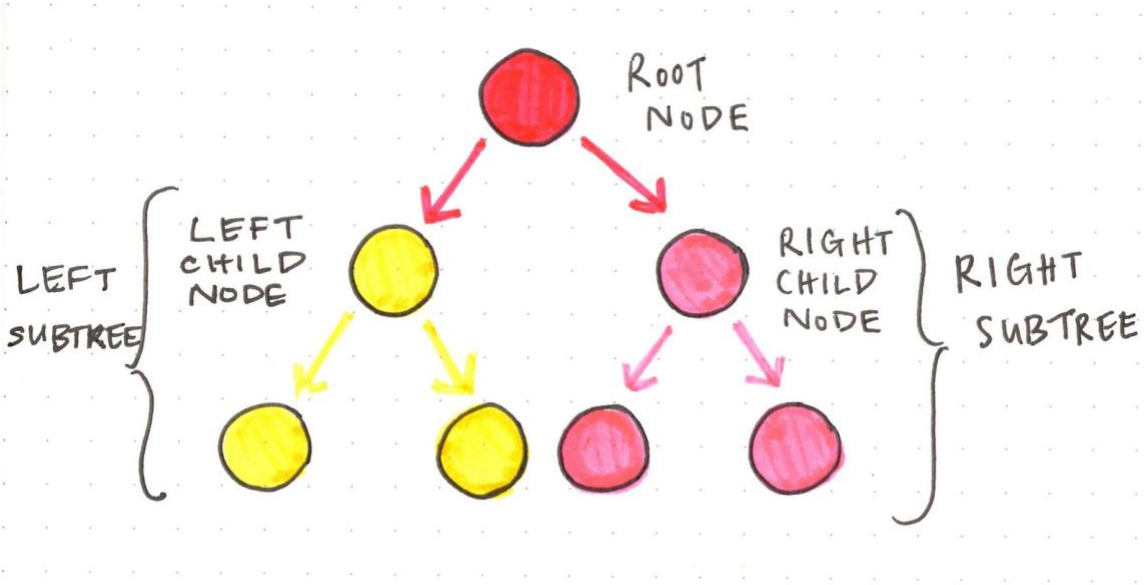
AVL TREE VS BINARY SEARCH TREE

Assignment Two



MSTRAH001

Rahul Mistri

CSC2001F

The Experiment

Aim:

The aim of this experiment is to determine the efficiency (by using the best, worst and average cases) of insertion and searching of the AVL(Adelso-Velskii and Landis) Tree against that of the Binary Search Tree.

Method:

Four classes will be created to conduct this experiment, the description will follow later in the OO Design. These classes will help create both data structures independent of one another and will undergo a series of tests. More specifically, these data structures will be tested with sets of data to determine the best, worst and average case performance scenario. These sets of data are created by use of scripting and a CSV file containing 500 records. The script will create 500 subsets of the data which will be fed as input in both data structures for insertion and searching. The results will be recorded, graphed and analysed according.

OO Design:

In this section of the document, I will discuss only the relevant fields and methods from the classes used in this experiment. For a descripting of all fields and methods, please consult the Javadocs for this assignment.

Power.java – this class serves as the blueprint for the objects that populate the AVL tree and binary search tree. This class implements the comparable interface to determine placement in the tree data structures.

- ***Fields***
    - *Date: String*
    - *Power: String*
    - *Voltage: String*

- ***Methods***
    - *Power(String s) – this method served as the main constructor, from String s the Date, Power and Voltage fields were extracted*
    - *toString() – to return a String combining all the data*
    - *getDate() – return the date*
    - *compareTo() – this class implemented the comparable interface, and helped compare two power recordings according to the date recorded.*

PowerAVLApp.java - this class had 2 inner classes, one for the AVL tree and one for the node used in the AVL tree. It had search methods and was tested for efficiency.

- ***Fields***
    - *tree: AVLTree*
- ***Methods***
    - *PowerAVLApp(String filename) – this method serves as the main constructor, and constructs an AVL tree from the data found in filename*
    - *printCounterToFile() – prints a counter variable to the file*
    - *printDateTime(String dateTime) – prints the details of the power object recorded at dateTime*
    - *printAllDateTimes() – prints all power objects*
    - *printAllDateTimes(String keyfile) – prints all power objects corresponding to the list of keys*
- ***AVLTree (inner class)***
    - ***Fields***
        - *root: Node – serves as the starting node of the tree*
    - ***Methods***
        - *leftRotate(Node y) – left rotate node rooted at y*
        - *rightRotate(Node y) – right rotate node rooted at y*

        - *insert(Power key) – inserts a node into the tree and performs rotations to keep the tree balanced*
        - *findnode(String dT) – finds a specific node in the tree given the date time*
- ***Node (inner class of AVLTree)***
    - ***Fields***
        - *Power data*
        - *Node leftChild*
        - *Node rightChild*
    - ***Methods***
        - *Node() - used to create a node object for storing in the tree*
        - *toString() – returned the toString of data*

PowerBSTApp – this class had 2 inner classes, one for the binary search tree and one for the node used in the binary search tree. It too had search methods and was tested for efficiency.

- ***Fields***
    - *tree: BST*
- ***Methods***
    - *main() – this method would either return a specific Power toString if a specific date was given at runtime, or the toStrings of all Power objects if no parameter was passed*
    - *printDateTime(String dT) – this method displays the*
- ***BST (inner class)***
    - ***Fields***
        - *mainRoot: Node*
    - ***Methods***
        - *BST() – this constructor initialized the tree*
        - *InsertRec(Node root, Power data) – this recursive method added nodes to the ends of the tree, and was sorted according to the dateTime reference recorded*
        - *PrintTreeInOrder(Node root) – this recursively printed the tree structure from earliest power recordings to latest recordings*

        - *PrintDateTime(Power temp, Node node, int opCount) – this recursive method would print the required node's Power data and the corresponding amount of comparisons it took to find it*

- ***Node (inner class of BST)***
    - ***Fields***
        - *Power data*
        - *Node leftChild*
        - *Node rightChild*
    - ***Methods***
        - *Node() - used to create a node object for storing in the tree*
        - *toString() – returned the toString of data*

ExtractData.java – This class served as an auxiliary class to help aggregate the results from the testing done by the bash scripts.

- ***Fields***
    - *arrmin[] : int[500] – stores the minimum operation count per subset*
    - *arrmax[] : int[500] – stores the maximum operation count per subset*
    - *arravg[] : double[500] – stores the average operation count per subset*
- ***Methods***
    - *ExtractData(int preFname, int n) - for the nth test, this would store the data in the temporary array*
    - *getMax() – find the minimum in the temp array and store it in the arrmin*
    - *getMin() – find the maximum in the temp array and store it in the arrmax*
    - *getAvg() – find the average of the temp array and store it in the arravg*
    - *printToFile() – prints the arrays to a file, each nth line corresponding to the nth subset*

****NB: Both the Binary Search Tree and AVL Tree was implemented by using code from GeeksForGeeks and was adapted accordingly for the assignment**

Testing and Results

Part two testing was used to test if the AVLTree was working and the following 3 known data keys and 1 unknown were used, thereafter all the data was displayed.

```
16/12/2006/17:37:00
16/12/2006/22:51:00
17/12/2006/00:06:00
18/12/2006/00:06:00
```

The following output was displayed, with the search count below.

```
rahulmistri@mistriii:~/WorkRepo/Assignment 2/bin$ java PowerAVLApp 16/12/2006/17:37:00
16/12/2006/17:37:00 5.268 232.910
5
rahulmistri@mistriii:~/WorkRepo/Assignment 2/bin$ java PowerAVLApp 16/12/2006/22:51:00
16/12/2006/22:51:00 2.414 239.970
7
rahulmistri@mistriii:~/WorkRepo/Assignment 2/bin$ java PowerAVLApp 17/12/2006/00:06:00
17/12/2006/00:06:00 2.858 241.140
9
rahulmistri@mistriii:~/WorkRepo/Assignment 2/bin$ java PowerAVLApp 18/12/2006/00:06:00
Date/time not found
10
```

Thereafter all data was displayed, below are the first and last 10 outputs.

```
16/12/2006/17:24:00 4.216 234.840
16/12/2006/17:25:00 5.360 233.630
16/12/2006/17:26:00 5.374 233.290
16/12/2006/17:27:00 5.388 233.740
16/12/2006/17:28:00 3.666 235.680
16/12/2006/17:29:00 3.520 235.020
16/12/2006/17:30:00 3.702 235.090
16/12/2006/17:31:00 3.700 235.220
16/12/2006/17:32:00 3.668 233.990
16/12/2006/17:33:00 3.662 233.860
```

…

```
17/12/2006/01:34:00 2.358 241.540
17/12/2006/01:35:00 3.954 239.840
17/12/2006/01:36:00 3.746 240.360
17/12/2006/01:37:00 3.944 239.790
17/12/2006/01:38:00 3.680 239.550
17/12/2006/01:39:00 1.670 242.210
17/12/2006/01:40:00 3.214 241.920
17/12/2006/01:41:00 4.500 240.420
17/12/2006/01:42:00 3.800 241.780
17/12/2006/01:43:00 2.664 243.310
```

Part Four

Part 4 testing was used to test if the Binary Search Tree was working and the following 3 known data keys and 1 unknown were used, thereafter all the data was displayed.

```
16/12/2006/17:37:00
16/12/2006/22:51:00
17/12/2006/00:06:00
18/12/2006/00:06:00
```

The following output was displayed, with the search count below.

```
rahulmistri@mistriii:~/WorkRepo/Assignment 2/bin$ java PowerBSTApp 16/12/2006/17:37:00
16/12/2006/17:37:00 5.268 232.910
2
rahulmistri@mistriii:~/WorkRepo/Assignment 2/bin$ java PowerBSTApp 16/12/2006/22:51:00
16/12/2006/22:51:00 2.414 239.970
8
rahulmistri@mistriii:~/WorkRepo/Assignment 2/bin$ java PowerBSTApp 17/12/2006/00:06:00
17/12/2006/00:06:00 2.858 241.140
11
rahulmistri@mistriii:~/WorkRepo/Assignment 2/bin$ java PowerBSTApp 18/12/2006/00:06:00
Date/time not found
11
```

Thereafter all data was displayed, below are the first and last 10 outputs.

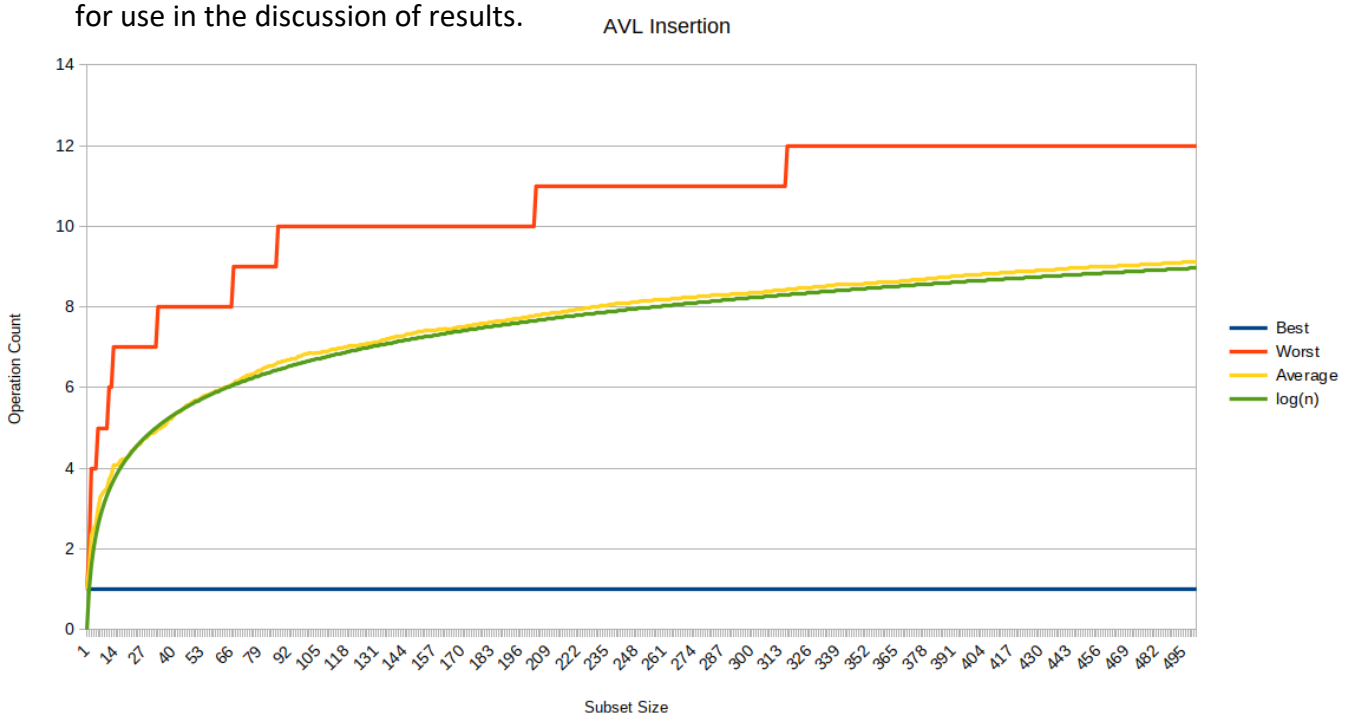
```
16/12/2006/17:24:00 4.216 234.840
16/12/2006/17:25:00 5.360 233.630
16/12/2006/17:26:00 5.374 233.290
16/12/2006/17:27:00 5.388 233.740
16/12/2006/17:28:00 3.666 235.680
16/12/2006/17:29:00 3.520 235.020
16/12/2006/17:30:00 3.702 235.090
16/12/2006/17:31:00 3.700 235.220
16/12/2006/17:32:00 3.668 233.990
16/12/2006/17:33:00 3.662 233.860
```

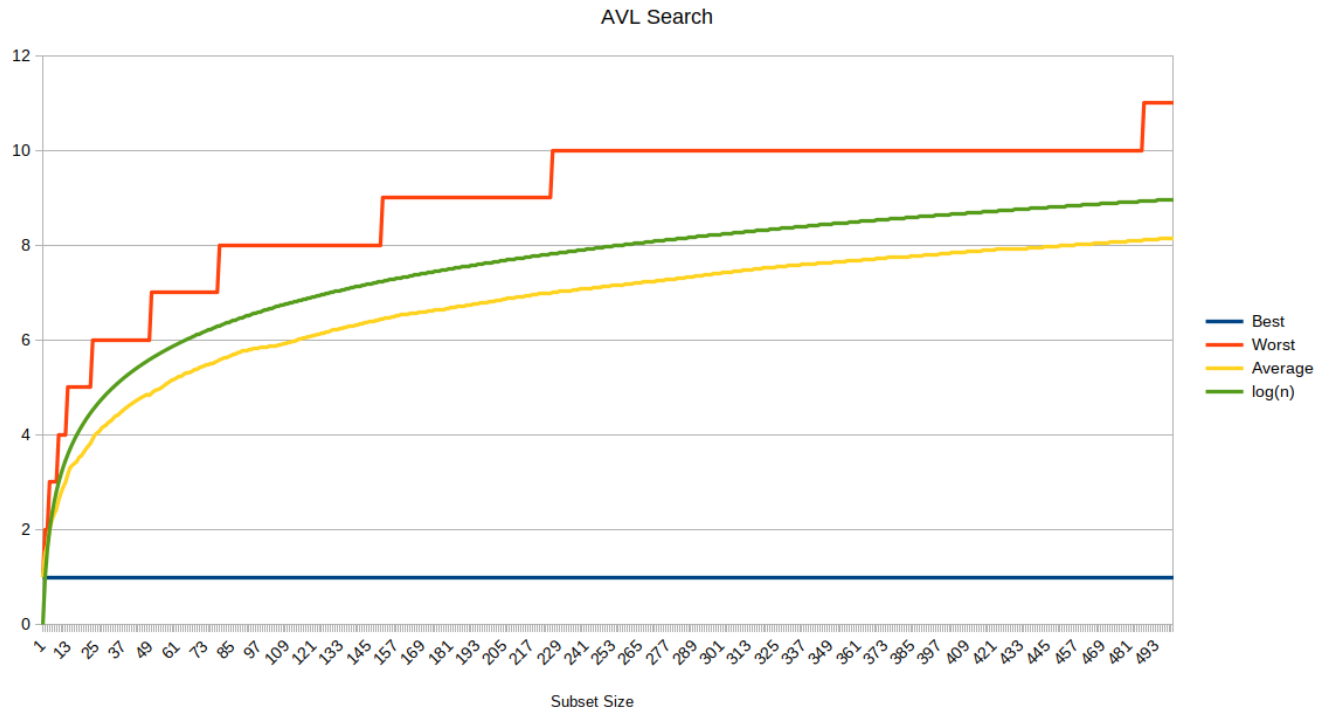
…

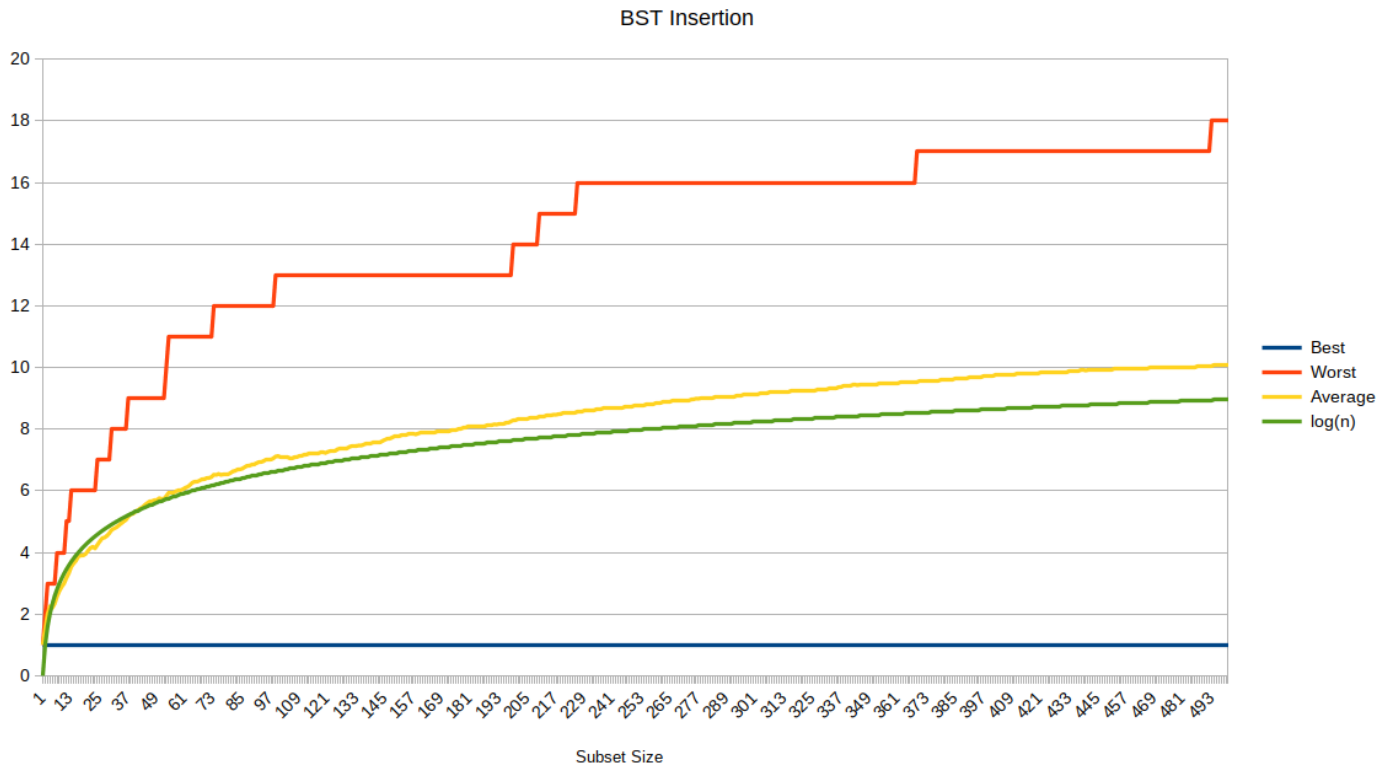
```
17/12/2006/01:34:00 2.358 241.540
17/12/2006/01:35:00 3.954 239.840
17/12/2006/01:36:00 3.746 240.360
17/12/2006/01:37:00 3.944 239.790
17/12/2006/01:38:00 3.680 239.550
17/12/2006/01:39:00 1.670 242.210
17/12/2006/01:40:00 3.214 241.920
17/12/2006/01:41:00 4.500 240.420
17/12/2006/01:42:00 3.800 241.780
17/12/2006/01:43:00 2.664 243.310
```

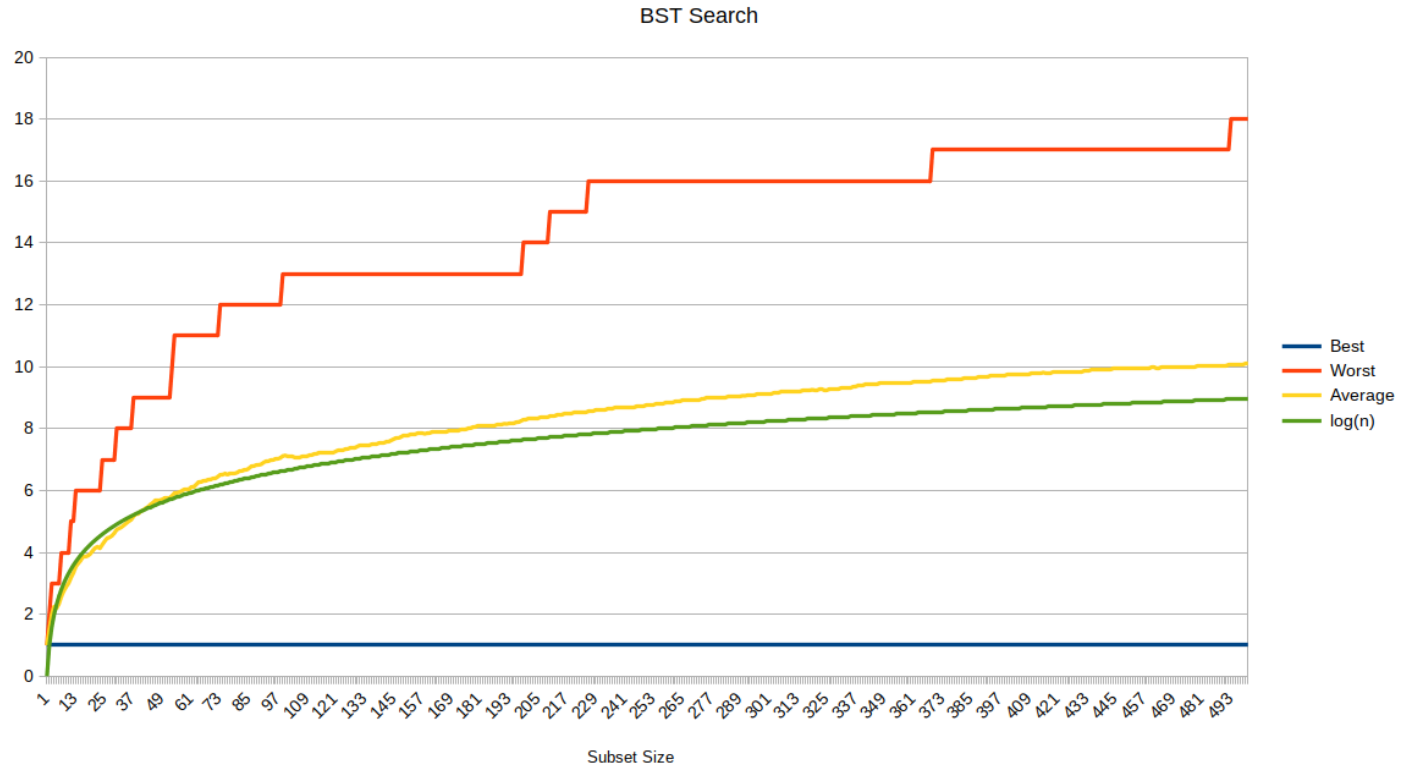
Part 5

Using bash scripting, FileGeneration.sh would generate 500 subsets of the csv file of data, where the nth subset contained the first n lines from the file. AVLTesting.sh and BSTTesting.sh would run and test each of the 500 subsets using the respective data structures. The ExtractData was used to obtain the data and formed the following graphs. Included in the graphs are the log(n) for use in the discussion of results.






BST Insertion
20
18
16
14
12
10
8
6
4
2
0
1 13 25 37 49 61 73 85 97 109 121 133 145 157 169 181 193 205 217 229 241 253 265 277 289 301 313 325 337 349 361 373 385 397 409 421 433 445 457 469 481 493
Subset Size
Best
Worst
Average
log(n)


BST Search
20
18
16
14
12
10
8
6
4
2
0
1 13 25 37 49 61 73 85 97 109 121 133 145 157 169 181 193 205 217 229 241 253 265 277 289 301 313 325 337 349 361 373 385 397 409 421 433 445 457 469 481 493
Subset Size
Best
Worst
Average
log(n)

Discussion of Results

The first two graphs depict the insertion and search operations for the AVL tree. Both have identical best case insertions and searches since they refer to either insertion of the first element or searching for the element that is the root node. This is an O(1) operation. The AVL tree has a slightly higher worst case for insertion than it does for searching, and this is due to the comparisons made during rotations. It is only a slight difference because as the tree grows in size, the less common it is for a rotation to be needed as the tree is always self-balancing and as it grows, there are more leaf nodes for insertions to take place on without disbalancing the tree. It follows a similar pattern to that of a logarithm relationship and hence the worst case for insertion and searching is O(log n). On average, both insertion and searching are very similar again, with insertion being more expensive due to rotations once again. From the graph it's seen that it follows the O(log n) relationship very closely.

The last two graphs depict the insertion and search operations for the Binary Search tree. It is interesting to note that for both insertion and searching, the best, worst and average cases are identical because no extra comparisons are made at all as there are no rotations. Insertion is basically searching for where the node ought to be and hence they are identical in graphs. The best cases are both 1 once again since they refer to the inserting or searching for the root node. On average, they are a similar to the log n relationship as shown in the graph, and this makes sense logically since you are always either going down a left or right subtree, hence it is O(log n). The worst case is also not too high, but is roughly double the average case, and for larger values of n-size subsets, it could be expensive. Regardless, it still follows an O(log n) pattern even in the worst case.

The results get more interesting when comparing the AVL tree with the Binary Search tree. It's clear to see that although their average cases are extremely close, it is their respective worst cases that differ greatly. AVL trees are self-balancing, and so to reach any leaf node of a tree of height n would take approximately log n operations. Although binary trees are similar, it is not a hard and fast property that is maintained. For large values of n, the implementation efforts of an AVL tree are well worth taking to reduce operation costs for insertion and searching.

Part 6

Results

	Insertion			Search		
	Best	**Worst**	**Average**	**Best**	**Worst**	**Average**
AVL	1	9	8.68	1	9	8.68
BST	1	500	250	1	500	250

*this data came from a single sorted subset of size 500

Discussion

From the above table, its clear to see that that AVL trees are vastly superior to Binary Search Trees when working with sorted data. AVL trees average value is extremely close to O(log n) whereas the binary search tree degenerated into a linked list of O(n) and traversed through all elements to insert and search, hence their identical values in searching and insertion.

Binary Search Trees are hence better suited for smaller values of n-size sets that are unsorted. As soon as data is sorted, or large values of n-size sets are used, AVL trees are the much better choice.

Creativity

Learning from previous mistakes in assignment 1, I made use of a combination of bash scripting and java programming to derive my test results. In order to be methodical, work was split up into separate bash tasks like generating files and running tests. All 500 tests were run using bash scripting and it was extremely efficient. Instead of opening up all data into excel and coding extraction in excel for best, worst and average case, I created a java program to read in all 2000 files of test data and to extract the best, worst and average cases. This enabled me to only have to import 4 text files of data which were ready to use and graph.

Git Usage Log

```
: commit eaa7f8f72f0ccdb14fb5644eddf4118b37c7d985
1: Author: Rahul Mistri <mstrah001@myuct.ac.za>
2: Date: Fri Mar 22 13:30:27 2019 +0200
3:
4: All files ready for submission
5:
6: commit 0795e632abd42bdfafbc728244f5f8b1f44d1093
7: Author: Rahul Mistri <mstrah001@myuct.ac.za>
8: Date: Fri Mar 22 12:19:21 2019 +0200
9:
...
103: Author: Rahul Mistri <mstrah001@myuct.ac.za>
104: Date: Thu Mar 21 17:00:25 2019 +0200
105:
106: Brought over Power.java from previous assignment
107:
108: commit e1bc785dbda9f48a1212a96a6836171ad52c7be5
109: Author: Rahul Mistri <mstrah001@myuct.ac.za>
110: Date: Thu Mar 21 12:44:32 2019 +0200
111:
112: Added .gitignore file to exclude class files
```

Please Note:

When executing the programs, please do not move anything around between directories as all have been placed in their correct place.