Assignment 2 Report: AVL, and Binary Search Trees

Calvin Nyoni

April 16, 2020

1 Object Orientated Design

1.1 Binary Search Tree Object Orientated Approach

The object oriented approach for this set of programmes included a loadShedding class, a BinaryNode class, a BinarySearchTree class, and the LSBSTApp. The loadShedding class consisted of loadShedding objects, which represented a specific period of loadShedding. They were recognised by the stage, date on which they occur, and the starting time. The loadShedding objects also stored the areas affected in an array. These loadShedding objects were then 'stored'/referenced to in BinaryNode objects as the 'item' variable. These BinaryNodes then had links/reference variables to other BinaryNodes. The BinaryNodes were then stored in a binary search tree data structure (classname: "BinarySearchTree"). The BinaryNodes were sorted in the BinarySearchTree data structure by their respective items. This sorting involved comparing each BinaryNode's item's (loadShedding object) ID to another to determine which was smaller than, or greater than, or equal to.

If/when the LSBSTApp was invoked the LSBSTApp would open the loadshedding data/textfile. It would then for each line in the text file create a corresponding loadShedding object with an ID storing the stage, date, and time while the corresponding stages were stored in an array. These loadShedding objects were then referenced to as items in their respective BinaryNodes. For each BinaryNode created, when inserted in to the BinarySearchTree, data structure, it was compared to other BinaryNodes using the item/loadShedding's ID to find an appropriate 'spot' in the binary search tree. When searching the BinarySearchTree the function 'search' in the tree would compare the key or search query given to each BinaryNode's item/loadShedding object until it found a corresponding node (or not) and return its item.

1.2 AVL Tree OO approach

The object oriented approach for this set of programmes included a loadShedding class, a Nodeclass, a AVLTreeclass, and the LSAVLTApp. The loadShedding class consisted of loadShedding objects, which represented a specific period of loadShedding. They were recognised by the stage, date on which they occur, and the starting time. The loadShedding objects also stored the areas affected

in an array. These loadShedding objects were then 'stored'/referenced to in the Node objects as the 'element' variable. These Nodes then had links/reference variables to other Nodes. The Nodes were then stored in a Adelson-Velsky-Landis Tree which is a binary search tree data structure, but which is continuously balanced - the class name for this balanced binary search tree was 'AVLTree'. The Nodes were sorted in the AVLTree data structure by their respective items. This sorting involved comparing each Node's element's (loadShedding object) ID to another to determine which was smaller than, or greater than, or equal to.

If/when the LSAVLTApp was invoked the LSAVLTApp would open the loadshedding data/textfile. It would then for each line in the text file create a corresponding loadShedding object with an ID storing the stage, date, and time while the corresponding stages were stored in an array. These loadShedding objects were then referenced to as elements in their respective Nodes. For each Node created, when inserted to the AVLTree data structure it was compared to other Nodes using the element/loadShedding's ID to find an appropriate 'spot' of in the binary search tree. Following insertion appropriate rotations would occur if the AVLTree became unbalanced. When searching the AVLTree the function 'get' would compare the element or search query given to each Node's element/loadShedding object - in a given branchof the AVL tree - until it found a corresponding node (or not) and return its item.

2 Experiment objective and methodology:

The objective of the experiment is to analyse the difference in efficiency between Binary Search Trees, and Adelson-Velsky and Landis Trees. This analysis examined differences in efficiency when searching and inserting for both data structures. This analysis also aims to determine efficiency among the structures with varying data sizes. The method of this experiment began with generating 10 random samples of varying size η. Where $\eta \in \{250, 500, 750, \ldots 2500\}$. These samples were all subsets of the larger/complete load shedding data set. The lines were obtained using a random function which generated numbers corresponding to lines in the file. Random lines were generated until the desired sample size η was reached.

Once the samples were generated from the data, the samples were then inputted in the various data structures. Testing was then conducted using a variable counter which then counted the number of times the computationally costly compareTo method was invoked during search, and insertion.For searching every single element of each sample size η was used as a search query to truly obtain the best, worst, and average case for each sample. For insertion a simple test using a fraction of the elements in each sample size η was conducted to determine efficiency during insertion.

3 Experimental results:

3.1 Searching:

Results for the binary search tree were as expected. Best case, it only took one operation for the search query to be obtained. The only case for which this occurs is when the element/item being searched for in the binary search tree appears in the tree's root node. The best case is not illustrative of the efficiency of the data structure as it rarely occurs thus not being representative of the true algorithmic complexity. Average, the results indicate that average searches using binary search trees model a logarithmic function or $\emptyset(\log n)$. In practical terms, this means that on average when searching for a node/item in a binary search tree the number of operations taken to find the item/node will be less than the number of nodes in the tree. The difference between the number of operations conducted when searching, and the number of nodes in the tree (size) increases as the size of tree increases thus the benefit of using this data structure improves as effeciency is 'virtually' preserved. Worst, the results are similar to those of the average case in that they are $\emptyset(\log n)$ in complexity. Again in practical terms, this means the worst case performance will still result in a search taking less than the number of nodes in the tree in terms operations.