Assignment 2: Binary Search & AVL Trees

by
Calvin Nyoni

Object Orientated design:

Binary Search Tree OO 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, 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 (class name "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 load shedding data/textfile. It would then for each line in the textfile 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 which were created. For each BinaryNode created, when inserted to the BinarySearchTree data structure it was compared to other BinaryNodes using item/loadShedding ID comparison 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.

AVL Tree OO approach:
The object oriented approach for this set of programmes included a loadShedding class, a Node class, a AVLTree class, and the LSAVLTApp. The loadShedding class consisted of loadShedding objects, which represented a specific period of loadShedding, 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 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 searched 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 load shedding data/textfile. It would then for each line in the textfile 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 which were created. For each Node created, when inserted to the AVLTree data structure it was compared to other Nodes using element/loadShedding ID comparison 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 branch of the AVL tree - until it found a corresponding node (or not) and return its item.

Experiment objective and method:

The objective of the experiment is to analyse the difference in efficiency between Binary Search Trees, and **A**delson-**V**elsky and **L**andis 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 ***n***. These samples were all subsets of the large/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 sample of size ***n*** was reached. ***n*** took on the values 250, 500, 750, ... 2500.

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 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 ***n*** 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 ***n*** was conducted to determine insertion efficiency.