Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Cameron	Student ID:	19035457
Other name(s):	Anthony		
Unit name:	Data Structures and Algorithms	Unit ID:	COMP1002
Lecturer / unit coordinator:	Valerie Maxville	Tutor:	lisa & Valerie
Date of submission:	27/05/2019	Which assignment?	(Leave blank if the unit has only one assignment.)

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature: Anthony Cameron Date of signature: 22/05/2019

(By submitting this form, you indicate that you agree with all the above text.)

Data Structures And Algorithms

Semester 1, 2019 v1.0

By Anthony Cameron
19035457

Table Of Contents

Chapter 1 Abstract

The purpose of this report is to explore ADT (Abstract Data Types) and compare the differences between multiple types of ADT implementation. The mission is to compare and report on multiple series of tree implementations such as Binary Search Tree, B-Tree and Two-Four-Tree comparing their efficiencies or time complexities for each method. Major differences in speeds include.

- Binary Search Trees are the slowest in time complexity for displaying data - has to navigate through the entire tree to find last value
- Binary Search Tree have the slowest insertion time for large data file because it need to check every node of a degenerate leaf to insert
- Binary Search Tree are non self balancing, high risk of degenerate trees changing time complexity from 0 log n to n^2
- B-Tree's have the fastest insertion time for large sample sizes however it slower than Binary Search Trees for small samples if the data doesn't cause a degenerate
- Two-Four-Tree has the slowest speeds in all fields as it has an extra step to split the tree by checking if full then splitting.

Background

Trees are data Structures that represent the hierarchal relationships between data. The choices of tree's were to compare the time complexities between a non balanced tree which is the Binary Search Tree and two other chosen self balancing trees, Two-Four-Tree and B-Tree. Two-Four-Tree and B-Tree share a lot of similarities such as splitting a child of a node when its full and pushes up the middle index followed by connecting the left and right children to the new node. However the major difference between Two-Four-Tree and B-Tree is that Two-Four-Trees always adds keys top down meaning it searches through leaf and splits any full nodes detected while B-Tree splits after it inserts into a full node building a tree up balancing itself, also known as building bottom up. Binary Search Trees difference is that it builds down wards and doesn't balance itself, which could become a problem when a leaf becomes degenerate or incomplete tree making time complexity and issue for functions that require searching the tree to insert, find or delete as it needs to navigate through everything.

Chapter 2 Methodology

The ADT's(Abstract Data Types) chosen are tested by profile mode which displays the size, height and balance. These statistics are use to determine the effectiveness for each tree. Each tree displays the overall speed it takes in nanosecond to complete reading the data and inserting, searching and deleting the values. B-Tree, Two-Four-Tree and Binary Search Tree each have their own test harness, which also describes the speeds in detail for each function and outputting the values stored within each node.

The chosen profile is tested by user input within command line :
Using the Following the format:

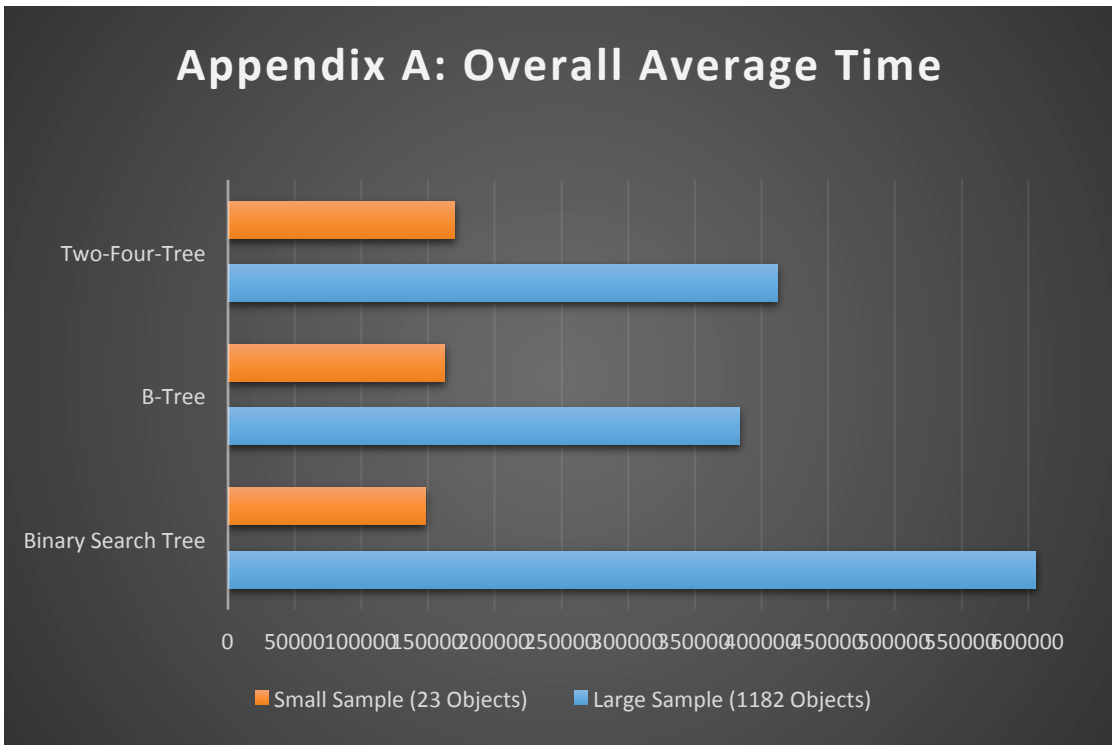
```
Usage: java TreeProfiler –n –y
        where
        n is –i :interactive testing environment
          or –p: profiling mode
        where y = Tree Type
          or –s: BinarySearchTree
          or –b: BTree
          or –f: 234Tree

while in profiler mode
 Usage: java TreeProfiler –n –y s [FILE.txt]
             where s: size
```

Files used for testing each tree were 2019.txt containing a small sample size of 30 and 20190218.txt containing 1500+. Test data is filtered by stocks class into valid objects, which changes the amount of values, inserted into each tree. Test results for each tree uses the same command line parameters and tested for both.

The reason for comparing B-Tree's and Two-Four-Tree is to find the difference of speeds in building a tree bottom up instead of top down and discover the differences in speed and height opposed to just implementing a Binary Search Tree. This narrows down the scope to which series of trees are preferable based on testing each set of data. Each test follows the procedure of running the above command line argument for the same file 2019.txt containing a small set of data. The Test is then repeated for each tree and documented afterwards. Data is then changed with the larger set and tested for each tree.

Chapter 3 Results



As seen in Appendix A. The Overall Average Time detailed above; the fastest algorithm is B-Tree for the largest data size, however Binary Search Trees are the fastest for sorting smaller sample sizes when processing the overall time in insertion time. Two-Four-Tree takes greater time to process in regards to small sample size whereas Binary Search Trees take longer to process. B-Trees have a smaller deviation window however Binary Search Trees have the greater deviation.

speeds	Binary Search Tree	B-Tree	Two-Four-Tree
Large sample	605616.3333333334	383478.3333333333	411767.3333333333
Small sample	147961.0	162409.33333333334	170058.33333333334

Appendix B

Appendix B shows a more accurate test result for each tree

Chapter 4 Conclusion And Future Work

Through the implementation of each tree, it has been discovered that the most efficient abstract data type are B-Trees. This is shown by its overall speeds in inserting data into a tree. B-Trees advantages become more apparent in larger test data while being slightly faster than Two-Four-Tree. However Binary Search Tress are overwhelmingly superior than the other trees when using small data sizes. Although Binary Search Trees are faster then the other trees, the effectiveness drops drastically when the data size increases as degenerate leafs become more common and the benefits are lost. B-Trees doesn't have the same problems as it self-balances keeping the process time more consistent when data sizes increases.

Further investigations could follow each tree's deletion method comparing the speeds. This method wasn't implemented in either Two-Four-Tree and B-Tree and may change the end result. Other improvements that could be included is testing different sets of data and comparing the time complexities based on randomised data, reversed data and sorted data.