Document version: 1.1 (2015-11-15)

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:	ONG	Student ID:	19287368
Other name(s):	MING HANG		
Unit name:	Data Structures and Algorithms	Unit ID:	COMP1002
Lecturer / unit coordinator:	Valerie Maxville	Tutor:	Valerie Maxville
Date of submission:	21/05/2019	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: ONG MING HANG Date of signature: 21/05/2019

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

Table of Contents

Abstract

The purpose of this report is to test the implementation of self-balancing tree ADT that maintains data in sorted order whilst ensuring the overall balance of the entire structure. A review of how these algorithm provides an insights into storage management and how it maintains its efficiency indefinitely by using the statistics to measure the size, height, balance and average time to insert or delete of the tree to analyze the time complexity and its efficiency. In this report, we will be assessing binary search tree, block tree and 234 tree.

Key findings include:

- Binary Search Tree has the worst time complexity out of all the trees when it comes to inserting large amount of data in partially sorted order.
- Block Tree (B-Tree) and 234-Tree has about the same average time (*in nanoseconds*) when it comes to inserting large sample of data in partially sorted order.
- Binary Search Tree has the best time complexity when it comes to inserting randomly sorted data.
- Block Tree (B-Tree) maintains a very consistent low height, this is due to its property of being extremely storage efficient.
- 234-Tree is shown to have a lot more half-filled nodes throughout the tree and a height that is higher than what Block Tree has due to its splitting behaviour.

The analysis shows that each of the trees is suited for different applications due to its unique property, it indicates that not all scenarios suits all trees. The result also shows that depending on the tree implementation, the speed of its functionality vastly differs on the design of the entire program. The report shows that the current implementation of each tree may not be the most solid interpretation of each concept but it showcase the ideas behind how they would work. The method of deletion for B-Tree and 234-Tree was not implemented due to its complexity and logic, to actually implement the deletion method would take up too much time just to get it working properly, for the meantime, the method isn't implemented and it would be favourable if there is a proper implementation for deletion in both B-Tree and 234-Tree.

Background

The trees being assessed for this report are **Binary Search Tree**, **Block Tree** and **234 Tree**. **Binary Search Tree** is considered as the most simplest tree ADT implementation and its simplicity has its advantages and disadvantages depending on the scenarios it receives. The structure of this tree consist of a network of nodes, each node consist of two pointers and a value. Every single node (except for leaf node) has at least a pointer to a single child, which it may be greater or lesser than its parent. Every tree has a single root that signifies the balance of the entire Binary structure, it is the pinnacle of the entire structure connecting each nodes to one another. Everytime a node is inserted, it will proceed to check the conditions to look for a spot to place the new node in, usually there are only two condition, if its lesser, go left else go right if its greater. There is also another condition that each key must be unique, if the key to be inserted already exist within the current tree, it will be effectively discarded, else insert as it is. The time complexity for accessibility is usually (*O(log 2 n)*) due to the fact it has to always check the conditions on each iteration.

Block Tree (B-Tree) has a structure that is vastly different from the likes of Binary Search Tree, for BST it has two pointers to two children while B-Tree has multiple pointers depending on the order (minimum number of keys per node) it has been given and the fact that each node has an array of keys which stores a reference to value, the leaves for B-Tree are also at the same depth. The height of both trees provides an insight to storage efficiency, usually the height of BST is large depending on the data is has been given, for instance ascending, descending or randomized data. No matter what type of data has been fed to B-Tree, it will be automatically sorted and balances itself to maintain an efficient structure. B-Tree is built for storage efficiency while BST is built for simplicity, the time complexity between the two trees will obviously differs depending on the data it has been fed but the height of the B-Tree will always remains the same, maintaining a low and consistent height. This is due to one of its behaviour that BST doesn't possess, a splitting function to balance itself. B-Tree will only split if the node is full, else leave it as it is, the node represents a small portion of data which maintains the primary structure through its balancing feature, which BST doesn't have. The sequential access time complexity for both tree is very different, due to the speed of accessing an array index (*O(1)*), accessibility is vastly superior for B-Tree.

The choice of both tree described above was required for the assignments specification, the additional tree ADT implementation will be **234-Tree**, which are B-Trees of order 4. Like how a normal B-Tree behave, 234-Tree can search, insert and delete with a time complexity of (*O(log n)*) time. Every node inside 234-Tree is a 2, 3, or 4 node which holds 1, 2 and 3 elements at most, respectively. The difference that puts a boundary between the two would be the splitting behaviour, B-Tree will only split when needed while 234-Tree splits when it encounters a full node on its way down the tree even when it isn't the right node to be inserted, thus the height will be slightly higher than B-Tree but still lower than BST.

Methodology

In the actual program, profiling mode allows the user to enter command-line arguments. There will be a method that processes those arguments to be used for profiling each trees depending on the choice that the user makes. The usage details will be displayed if the user does not enter sufficient command-line arguments, as displayed below:

```
Usage: java TreeProfiler m [t d f], square brackets are exclusive to
profiling mode
    where m is the type of which mode to use
    m is one of the modes
        -i - interactive testing environment
        -p - profiling mode
    t is one of the trees
        BST - Binary Search Tree
        BT - Block Tree
        TTFT - Two Three Four Tree
    d is the data size
        Note: Only BT requires data size, e.g. order
    f is the input file
        e.g. stock.txt, half.txt, small.txt
```

If the user's input matches the requirements, it will proceed to display the necessary informations regarding the profile of each tree. The statistics that is given are:

- **Size:** Number of elements in tree
- **Height:** Based on longest path
- **Balance:** A percentage measuring from 0-100% indicates how complete the tree structure is
- **Average time insertion per node:** How long would each node takes to insert measured in nanoseconds when given a file input

To actually test each tree on how they would handle large amount of data, large sample data are retrieved from https://www.asxhistoricaldata.com which contains EOD (end-of-day) stock data updated day by day. The size of each file is perfect for the testing environment of the trees, as it is consistent and not overly complicated. Since each of the ticker inside the file are unique, it would be perfect to test how well each tree adapts to the given data, measuring the time it takes to execute several functions, the number of elements it contains, the height of the tree and the overall balance of the tree.

To retrieve the statistic of each profile, each of them works differently for each tree. To get the size, it is automatically given by each tree as each inserts increments the size thus no complex function is made. Getting the height is the same for both B-Tree and 234-Tree but different for BST, for B-Tree and 234-Tree, the height is only increased if a split function is made on the root node, therefore a new root is made. For BST, it recursively traverse the tree until it gets the longest pathway from the root to the lowest leaf. The method to calculate the overall output balance would depend on the theory of ideal heights and calculating average time in nanoseconds will use Java built-in functions, the calculation for these methods is displayed below:

Ideal height:

For B-Tree and 234-Tree: $\log_{\texttt{max children size}}(\texttt{size})$

For Binary Search Tree: $\log_2(\texttt{size})$

Java implementation:

For B-Tree and 234-Tree ideal height: $\frac{Math.log(size)}{Math.log(maxChildSize)}$

For Binary Search Tree ideal height: $\frac{Math.log(size)}{Math.log(2)}$

Calculating average time in nanoseconds:

```
long startTime = System.nanoTime();
//...code to be measured...//
long estimatedTime = (System.nanoTime()- startTime)/tree.getSize();
```

Balance percentage: $\frac{Ideal\ height}{Actual\ Height} \times 100$

Output Examples:

Binary Search Tree commands: java TreeProfiler -p BST 0 stock.txt

```
Output:
...Profiling mode activated...
Tree size: 1552
Tree height: 1525
Tree balance: 0.66%
Average time insertion per node: 58741 nanoseconds
```

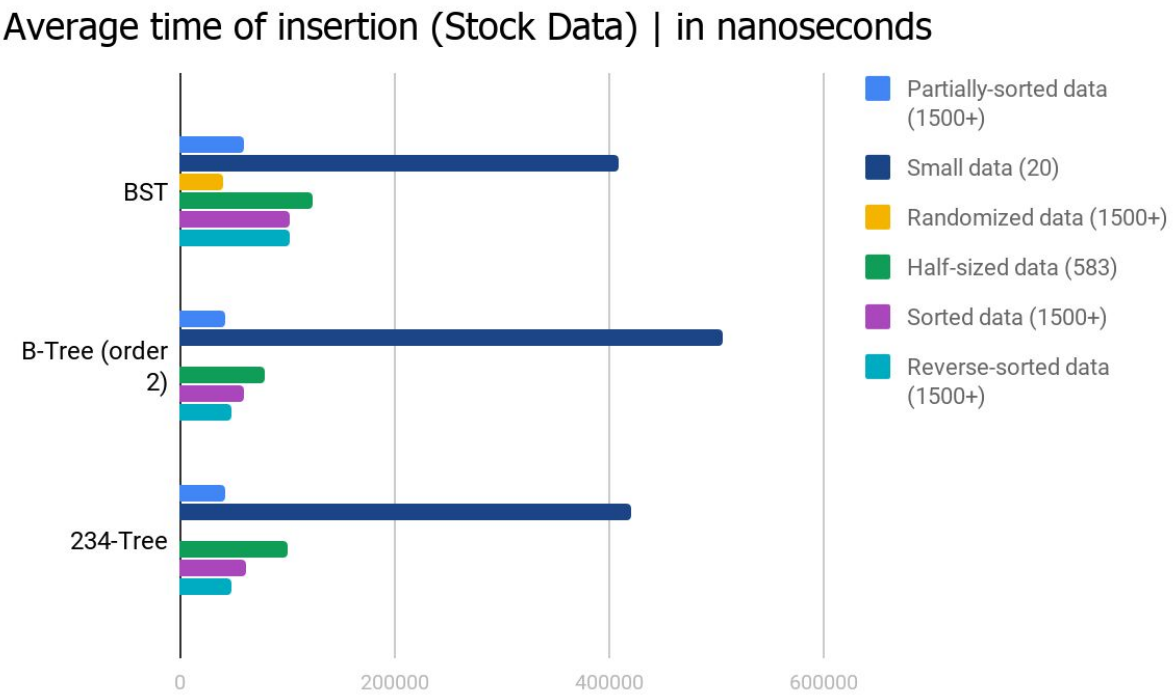
Block Tree commands: java TreeProfiler -p BT 2 stock.txt

```
Output:
...Profiling mode activated...
Tree size: 1552
Tree height: 7
Tree balance: 57.14%
Average time insertion per node: 46060 nanoseconds
Profiling mode ends!
```

Two-Three-Four Tree commands: java TreeProfiler -p TTFT 0 stock.txt

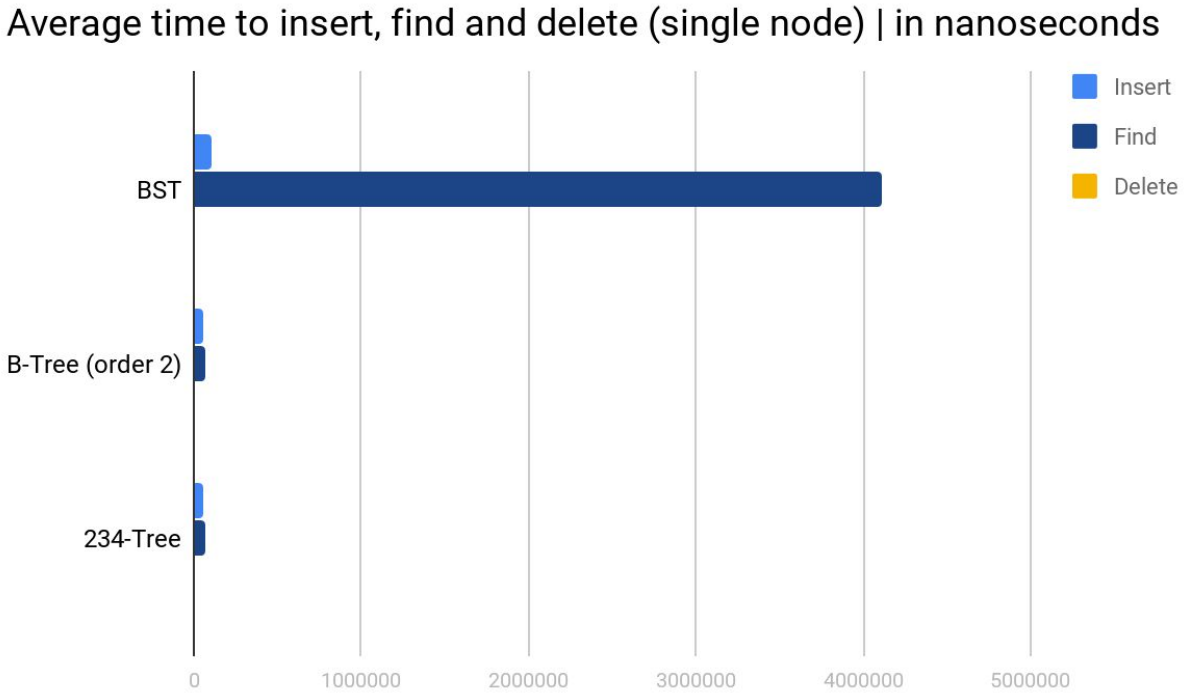
```
Output:
...Profiling mode activated...
Tree size: 1552
Tree height: 10
Tree balance: 40.00%
Average time insertion per node: 49440 nanoseconds
Profiling mode ends!
```

Results



Trees	Partially-sorted data (1500+)	Small data (20)	Randomized data (1500+)	Half-sized data (583)	Sorted data (1500+)	Reverse-sorted data (1500+)
BST	59688	407823	40343	124047	102728	102094
B-Tree (order 2)	41165	504882	N/A	79696	59557	48566
234-Tree	41482	420480	N/A	100959	61727	48110

Note: The B-Tree and 234-Tree program throws an AIOB (ArrayIndexOutOfBoundsException) when given randomized data, the cause is currently unknown and it will remained the same.



Trees	Insert	Find	Delete
BST	100708	4116666	9842
B-Tree (order 2)	52572	62081	N/A
234-Tree	57683	64633	N/A

Note 1: The complexity of implementing deletion for B-Tree and 234-Tree is too difficult to implement within the scope of this report.
Note 2: All of the insert, find and delete function are tested using test harnesses.

Conclusion

The assessed tree ADT results shows that each tree performs differently depending on the data it has been given, when it comes to the average time for inserting data into the trees, BST performs the slowest of them all by almost 32% but it performs relatively fast when given a randomized sets of data, unfortunately B-Tree and 234-Tree can't be tested on randomized sets of data due to issues caused by the program. It is discovered that somehow small amount of data causes the program to take longer to execute in comparison to large sets of data by a fraction of 10, the cause is currently unknown. When given half-sized stock data, sorted data and reversed-sorted data, B-Tree and 234-Tree performs about the same, BST performs just a little slower than the rest. At the same time, testing insertion, finding and deletion on a single specified node shows that insertion takes 2 time longer for BST, in comparison to B-Tree and 234-Tree. Finding takes BST 66 times longer to execute in comparison to B-Tree and 234-Tree, most likely due to how it traverses the tree. Deletion can only be tested for BST and it performs relatively fast, however, deletion implementation for B-Tree and 234-Tree is on another different scale in terms of complexity, so it isn't implemented for the program. The assessed tree ADT which are implemented in a fashion that wouldn't be considered optimized due to the design of the entire program which puts limitation on time complexity. The program itself requires improvements in every area if possible, the key areas to improve is how the structure of the code is written, implementing inheritance to add stability to the entire program which might also increase the flow of the program whilst improving time complexity. It must be remembered that this implementation and analysis is limited; the functions in each of the tree are just basic operation on how they operate, it seems that to further improve the design, a greater depth of understanding and evaluating these advanced tree ADT is required to utilize the concept to its utmost potential. The problem of not being able to implement deletion for B-Tree and 234-Tree limits what B-Tree in general are capable of, it is also one of its prime self-balancing feature. At this point, the program will work as it is, if there were more time allocated for this project and report, it could be further improved. This report highlights the importance of tree ADT and understanding how they would work in terms of real hardware concepts like RAM.