Batandwa Mguts
MGTBAT001

Introduction

The goal of this assignment is to compare the Binary Search Tree with a traditional unsorted array data structure. The programming is done in Java using a real world application to check if a student is on a pre-approved list of access to campus during the lockdown.

The OOP Design

The application has two entry points, AccessArrayApp and AccessBSTApp. Both of these entry points initialise the common main entry point in the Application class and pass it an object that implements the SimpleCollection interface. AccessArrayApp passes in a SimpleArrayCollection instance whereas AccessBSTApp passes in a BinarySearchTree instance. Both the SimpleArrayCollection and BinarySearchTree are defined in the data structures package. The BinarySearchTree class uses some code written by Hussein Suleman which is kept in the forks package inside the data structures package.

Once the Application instance is run, it loads the student data from the oklist.txt file into the given SimpleCollection. The reading of the file is done by the StudentDataReaderService, which returns an array of students. That array is then passed to the given SimpleCollection to store. SimpleArrayCollection stores the array as it is, whereas BinarySearchTree loads the data as nodes. For easy unit testing, the opening of the oklist.txt file is done by the FileSystemService which opens the file into an InputStream.

The StudentDataReaderService depends on the FileSystemService to do its work, but passing the FileSystemService to the reader everytime we need to read data would make the code hard to follow. Dependency injection using a static dependency manager class was used to solve this. The Locator class has a static Locator instance that allows the programmer to register objects that other parts of the code will need later. So the Application instance registers the FileSystemService with the Locator and the StudentDataReaderService gets it from there.

End-to-end tests

The two application entry points were thoroughly tested through unit tests. To verify that the whole integration works, the whole application was tested like how an end user would use it.

Invocation	Output
java AccessArrayApp JCBROR008	Rorisang Jacobs
java AccessArrayApp MLLNOA014	Noah Maluleke

java AccessArrayApp SHBCAL017	Caleb Shabangu
java AccessArrayApp bad1	Access denied!
java AccessArrayApp bad2	Access denied!
java AccessArrayApp bad3	Access denied!
java AccessBSTApp JCBROR008	Rorisang Jacobs
java AccessBSTApp MLLNOA014	Noah Maluleke
java AccessBSTApp SHBCAL017	Caleb Shabangu
java AccessBSTApp bad1	Access denied!
java AccessBSTApp bad2	Access denied!
java AccessBSTApp bad3	Access denied!
java AccessArrayApp	MLLNOA014, Noah Maluleke WTBJAY001, Jayden Witbooi KHZOMA010, Omaatla Khoza MLTLUK019, Luke Malatji NKNTHA021, Thato Nkuna …. …. MSXROR015, Rorisang Mosia DNLAYA006, Ayabonga Daniels CHKOFE015, Ofentse Chauke MNGREA015, Reatlegile Moeng SHBCAL017, Caleb Shabangu
java AccessBSTApp	MLLNOA014, Noah Maluleke WTBJAY001, Jayden Witbooi KHZOMA010, Omaatla Khoza MLTLUK019, Luke Malatji NKNTHA021, Thato Nkuna …. …. MSXROR015, Rorisang Mosia DNLAYA006, Ayabonga Daniels CHKOFE015, Ofentse Chauke MNGREA015, Reatlegile Moeng SHBCAL017, Caleb Shabangu

Carrying out the experiment

To test the performance of the two data structures, instrumentation tests were done. The Instrumentation.java class runs the instrumentation tests. The instrumentation is performed as follows:

Batandwa Mguts
MGTBAT001

1. Choose 10 random sets of students where the first set has 500 students, the second 1000 students, the third 1500 students, and so on.
2. For each set do:
    a. Load the data into the SimpleArrayCollection data structure then for each student in the set search for that student in the data structure and record the opCount. The opCout is the number of comparison (<, =, >) operations performed by the data structure while searching for the item. We clear the opCount when we search for the next item. Once all the opCounts are collected the case scenarios are computed as follows
        i. Best Case -> the lowest opCount
        ii. Average Case -> the sum of the opCounts divided by the cardinality of the set
        iii. Worst Case -> the highest opCount
    b. Repeat *a* but now use the BinarySearchTree data structure.

Once the above instrumentation was done, the Instrumentation class prints out the results. Here is one of trials we got:

```
BST Instrumentation Logs For Given Cardinality
-----------------------------------------------
n, worstCase, averageCase, bestCase
500, 37, 20, 1
1000, 41, 23, 1
1500, 45, 23, 1
2000, 57, 27, 1
2500, 45, 24, 1
3000, 49, 24, 1
3500, 49, 25, 1
4000, 51, 26, 1
4500, 49, 25, 1
5000, 51, 26, 1

Array Instrumentation Logs For Given Cardinality
-----------------------------------------------
n, worstCase, averageCase, bestCase
500, 500, 243, 1
1000, 999, 465, 1
1500, 1500, 680, 1
2000, 2000, 887, 1
2500, 2500, 1063, 1
3000, 3000, 1242, 1
3500, 3499, 1401, 1
4000, 4000, 1565, 1
4500, 4500, 1702, 1
5000, 5000, 1865, 1
```

Note that n is the cardinality of the data set.

Batandwa Mguts
MGTBAT001

Conclusion from above data

The simple conclusion we make about the BinarySearchTree compared to the SimpleArrayCollection is that it is more efficient in finding items. For the worstCase scenario at 5000 entries, for example, the BST implementation performed 98% faster than the array implementation.

Both implementations have the same best case scenario which is the case when the item being searched for is the first item in the array or the root node in the BST. What is interesting about the BST is that using different tree traversal algorithms can alter the best case. The preOrder algorithm is what gives the best case of 1.

Now thinking about it though, this was an unfair experiment. When searching for an item, the BST is traversing a sorted collection whereas the array had its data unsorted. The array collection had no efficient way of searching for items because its data was not sorted. Had its data been sorted, it would have been able to perform binary searching which would have improved its searching efficiency by a great margin..

Miscellaneous

Git Usage

```
> git log | (ln=0; while read l; do echo $ln\: $l; ln=$((ln+1)); done) | (head -10; echo ...; tail
-10)

0: commit f464e8068641a50db5387127e79c6e8d9dd76f52
1: Author: Batandwa Mgutsi <B.mgutsi@gmail.com>
2: Date: Thu Apr 8 00:10:07 2021 +0200
3:
4: Ceil the averageCase scenario instead of just casting it to int to get more accurate
values
5:
6: commit e5ba6fa5443eacce2cd43041b1623526b9e35d39
7: Author: Batandwa Mgutsi <B.mgutsi@gmail.com>
8: Date: Wed Apr 7 23:59:48 2021 +0200
9:
...
101: Author: Batandwa Mgutsi <B.mgutsi@gmail.com>
102: Date: Mon Apr 5 11:09:58 2021 +0200
103:
104: Adds locator
105:
106: commit f5909e8ef276b999a192774a533a44d67b26ccae
107: Author: Batandwa Mgutsi <b.mgutsi@gmail.com>
108: Date: Mon Apr 5 10:00:05 2021 +0200
109:
110: Initial commit
```

Batandwa Mguts
MGTBAT001