Jordy Kafwe Kioni
KFWJOR001

Assignment 1 - Binary Search Trees

OO Design

The **Vaccine** class is the data class. That is, its sole purpose is to hold the data read from the vaccinations CSV file and make it accessible to all the other classes that need this data. The data it stores are the country, date and number of vaccinations. **Vaccine** objects are what the 2 data structures (**VaccineArray** and **BinarySearchTree** classes) store. The class also implements the **Comparable** interface which is used to compare items when performing the search operations.

VaccineArray class wraps around a traditional fixed-size array and abstracts all the operations that need to be performed on the array. That is, it abstracts away inserting data and searching through the array for data. The operation count values are also counted in this class.

VaccineArrayApp class ties the **Vaccine** and **VaccineArray** classes together to provide a user interface and also to allow for experimenting on the array data structure. The user interface part accepts user input to allow the user to query the data and returns all the results to the console. The experimenting part writes operation count values for both search and insert operations of every data item to a text file. There is no user interface for the experiment part as it is run from a script.

BinarySearchTree class – from Vula – provides all the operations that are needed to perform on a binary search tree. The operation count values are also counted in this class.

VaccineBSTApp class does everything that **VaccineArrayApp** does but just using a binary search tree – it uses the **BinarySearchTree** classs for this. That is, it provides a user interface and a way to do the experiment. There is no user interface for the experiment part as it is run from a script.

The goal of the experiment

The goal of the experiment is to compare the efficiency of storing and retrieving data in a binary search tree to a fixed-size unsorted array data structure – both implemented in Java. Especially how efficient each data structure is at inserting and searching for data as the amount of data increases. The experiment does this by counting the operations taken to insert and retrieve data items.

How the experiment was executed

The experiment was executed by using 10 subsets (starting at 992 and going up in increments of 10 percent of the 9919 entries). That is the second subset would have 1982 entries (20 percent of 9919) and so forth.

The Python script (**experiment.py**) creates a subset and then the test input, but the data is only inserted once into the data structures in the Java programs for every subset. This is because the insert operations for items in the same subset (and in the same order) will take the same number of key comparisons every time – so there is no reason to do it every time. The operation count values for insert are written to 2 different files - one for the binary search tree and another for the array.

The script then searches for every one of the n country+date entries in the subset using the 2 instrumented applications and writes the number of operations it took to search for each item to a file – one file for the binary search tree and another for the array.
After all the operations counts in all subsets (for both insert and search) has been obtained, a second script (**analysis.py**) is used to calculate the best, average and worst

cases (for both search and insert) and then writes these to 2 CSV files – a file each for search and insert.

Test Values

I ran this command (changing the file names and also the class to execute depending on the test) to do the input and output redirection to obtain these test values.

```
java -cp bin VaccineArrayApp < analysis/part1/testinput1 >
analysis/part1/arrayoutput1
```

All the files (input and output) can be found in the "analysis/part1/" directory in my submission. Both the VaccineArrayApp VaccineBSTApp gave out the same output for all the test inputs

Test Number	Input	Output

1	`2022-01-30`
	`South Africa`
		`Enter the date:`
		`Enter the list of countries`
		`(end with an empty line):`
		`Results:`
		`South Africa = 57785`

2	`2022-02-13`
	`Peru`
	`Maldives`
	`South West Africa`
		`Enter the date:`
		`Enter the list of countries`
		`(end with an empty line):`
		`Results:`
		`Peru = 97908`
		`Maldives = 2541`
		`South West Africa = <Not Found>`

3	`2022-02-03`	`Enter the date:`

```
Jamaica
Scotland
North Bhutan
Jordan
Liechtenstein
```

```
Enter the list of countries
(end with an empty line):
Results:
Jamaica = 2551
Scotland = 7592
North Bhutan = <Not Found>
Jordan = 14966
Liechtenstein = 84
```

Results

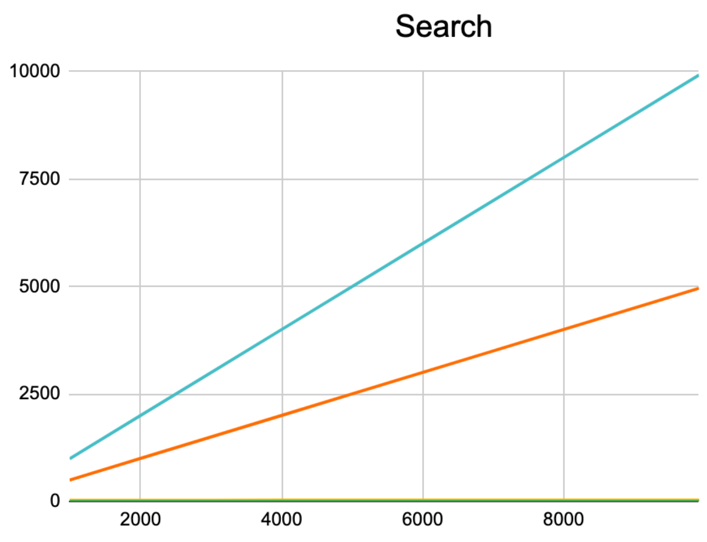
Array

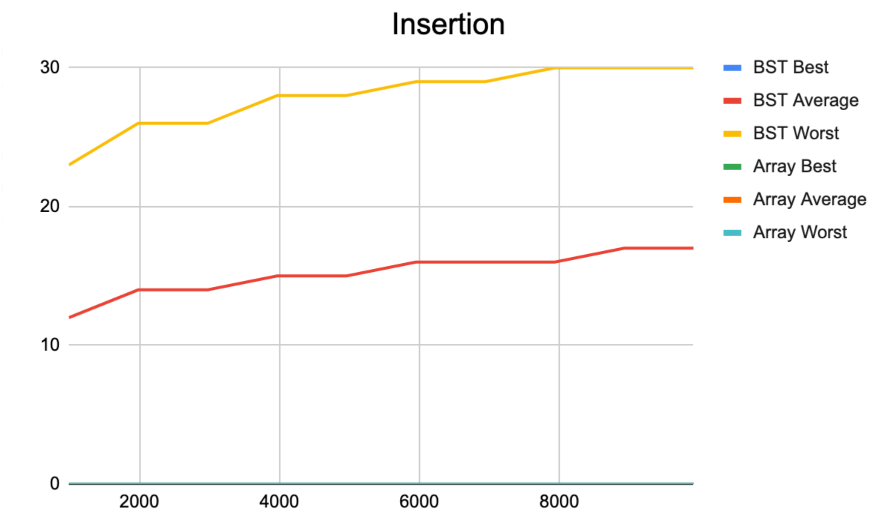
	Insert			Search		
n	Best	Average	Worst	Best	Average	Worst
992	0	0	0	1	497	992
1984	0	0	0	1	993	1984
2976	0	0	0	1	1489	2976
3968	0	0	0	1	1985	3968
4960	0	0	0	1	2481	4960
5952	0	0	0	1	2977	5952
6944	0	0	0	1	3473	6944
7936	0	0	0	1	3969	7936
8928	0	0	0	1	4465	8928
9919	0	0	0	1	4960	9919

Binary Search Tree

	Insert			Search		
n	Best	Average	Worst	Best	Average	Worst

992	0	12	23	1	13	24
1984	0	14	26	1	15	27
2976	0	14	26	1	15	27
3968	0	15	28	1	16	29
4960	0	15	28	1	16	29
5952	0	16	29	1	17	30
6944	0	16	29	1	17	30
7936	0	16	30	1	17	31
8928	0	17	30	1	18	31
9919	0	17	30	1	18	31





Discussion of results

In the worst case, the array has to search through all the items in the array. Which means that in the largest subset (of 9919) it has to make 9919 key comparisons to find the data item. A binary search tree only takes 31 comparisons to find the data item in its worst case in the largest subset. Clearly the binary search tree is better at retrieving data because it needs to make exponentially less key comparisons (9919 vs 31) than an array. This trend carries out throughout all the worst and average cases for all subsets.

It is more efficient to insert items in an array because it always take 0 comparisons but a binary search tree is not much worse because in its worst case on the largest subset it takes 30 comparisons to insert 9919 items.

In conclusion, the binary search tree is clearly the better data structure for retrieving a data item in a large dataset – even though there is a negligible increase in insertion cost.

Creativity

- I managed to code my application to only insert once. This cut the experiment time from hours – like most of my peers had – to generating all the operation counts for all subsets (both BST and Array) in under 1 minute.

- I also made use of Java command line arguments in my script to determine if the app is being run by the user or if it is an experiment being executed by the script

- My **analysis.py** script summarises all the data into best, average, worst (for both BST and array) and saves into a CSV file. This makes it easy to open the data in spreadsheet software which tabulates it and also makes it easy to generate graphs.

Git Usage log

```
0: a26bf2f add docstrings
1: b23986e add analysis script
2: b4d569a add javadoc comments
```

```
3: 745e246 conduct experiment in array app class
4: e58172b conduct experiment in BST app class
5: cb01e8f add array opcount file write
6: a3d8cf3 add experiment script
7: 5502ab2 add makefile
8: 04b2728 change method access modifiers
9: 3ef1797 add find result by country
...
18: 5e0c0b3 read from csv file
19: 49ef5cf add VaccineArray constructor
20: fb11cca add VaccineArrayApp class
21: cb44979 add find method
22: 074833a add VaccineArray class
23: 82bb469 change vaccinations field to int
24: 78c77cb add Vaccine compareTo method
25: 3ab346a add Vaccine getters
26: d3f0a45 create vaccine class
27: f206219 first commit
```