University of Cape Town

CSC2001F 2022

Assignment 1

Author: NJMLUN002 [1438523]

Date: 10-03-2022

Comparison of the Binary Search Tree and Traditional Array Data Structure

Abstract

An experiment was conducted to compare the performance of Binary Search Tree data structure with an unsorted array data structure. These data structures were implemented in two Java programs, namely the *VaccineArrayApp* and *VaccineBSTApp*, to obtain daily vaccination numbers for a list of countries from a dataset with 9919 randomised entries. Each data structure item stored 3 values for an entry: the country name, date of data recording and the number of vaccinations on the date. The program which implemented the binary search tree data structure used recursive *insert* and *search* to add data and search for data in the tree, respectively. The program implementing the unsorted array data structure used iterative methods to add data and search for data. The performance of the data structures was tested using instrumentation whereby an *opCount* static instance variable stored the number of comparisons completed in each program. The higher number of comparisons when using the array data structure compared to when using the binary search tree indicated that the binary search tree was more efficient than the unsorted array data structure. This was expected as theory showed that the search algorithm for an unsorted array has a worst case time complexity of O(nlogn) whereas the binary search tree has a worst case time complexity of O(n). The size n of the dataset was varied to measure the number of comparison operations in the best, average and worst case. The experiment used ten values of n that were spaced approximately equally, i.e. n = 991, 1982, 2973, 3964, 4955, 5946, 6937, 7928, 8919, and 9919. For each subset of n entries, a query list with one item was used to determine the number of comparison operations and the best case, worst case and average of these operations.

Introduction

Data structures are programmatic entinties used for storing data so that it can be used in the most efficient manner. There are various types of data structures including linked lists, AVL trees, B+ trees and Red-Black trees. An experiment was conducted to compare the efficiency of the Binary Search Tree (BST) data structure and the unsorted array data strucure as implemented in two Java programs. The VaccineArrayApp program was used to implement the unsort array data structure while the VaccineBSTApp was used to implement the BST data structure.

The efficiency of a data structure is measured by the speed at which algorithms perform specific tasks and their usage of memory. In general, data structure algorithms include search, sort, insert, update and delete algorithms. This experiment focused on the comparison between the insert and search algorithms of the binary search tree and unsorted array data structure.

Programmatic instrumentation was used to compare the insert and search algorithms of the two data structures. The best case, worst case and average case of each algorithm was determined using the number of comparisons of keys in each data structure. The best case involves the least number of comparisons to be performed to insert an item into the data structure or to search for an item. This generally involves inserting an item into the first unoccupied position in the data structure or searching for an item and getting a hit after the first comparison. On the contrary, the worst case involves inserting an item or searching for an item in the last occupied position in the data structure. The worst case

and average cases are typically of more concern since these are more important to the user and the programmer.

This report includes the description of the VaccineArrayApp and VaccineBSTApp in terms of the classes that were used in the OO design. Following the OO design description is the method used for comparing the binary search tree and unsorted array which includes the test values of the dataset subsets that were used. The results follow the method and show the output for each test value and the best, average and worst cases for both applications. The discussion of the results and the conclusion complete the report of the experiment and a statement of creativity and the summary of statitistic from the git log are found at the end of the paper.

VaccineArrayApp OO Design

Classes:

- Country
- CountryRegister
- VaccineArrayApp

The Country class has 4 instance variables including the name of the country, the date on which the vaccination information was recorded and the number of vaccinations on that date. The fourth instance variable is the key associated with this combination of name and date. The Country object has the primary compareTo() method which overrides the compareTo() method of the Comparable interface. This compareTo() method is used in all classes to compare the name and date of each data item to sort the data in a lexicographical manner.

The CountryRegister is an ancestor class of the Country class and takes in a file to create an unsorted array data structure of Country objects using each line in the data file. The CountryRegister class uses a two-dimensional array to collect the data in the file. A date is stored in the first index of each array and every country with information that was recorded on that date is stored in the indices that follow. The Country and CountryRegister classes are found in the models package.

The VaccineArrayApp contains the main method of the application. It uses the CountryRegister class's generateData() method and subsequently the addDate() and addData() methods to insert data into the unsorted two-dimensional array. When the user is prompted for a query list of countries, the CountryRegister uses the findCountry() method to search for the queried country and date.

VaccineBSTApp OO Design

Classes:

- Country
- BinaryTreeNodeInterface
- BinaryTreeNode
- BinarySearchTreeInterface
- BTQueue

- BinarySearchTree
- VaccinationsBST
- VaccineBSTApp

The Country class is used again for the VaccineBSTApp to store the country name, date and number of vaccinations. The compareTo() method in this class is used to compare and sort data in a lexicographical way.

The BinaryTreeNode class is a parametrised (or generic) class which defines a binary tree node which with a right child, a left child and holds a data item of the Country data type. The methods in the BinaryTreeNode are implemented from the BinaryTreeNodeInterface interface for concreteness.

The BinarySearchTree class is also a parametrised class which uses the Country class as a parameter and implements methods in the BinarySearchTreeInterface for concreteness. This class uses the compareTo() method in the Country class to guide the insert and search methods to be used in the VaccineBSTApp program. The BinarySearchTree class creates the binary search tree data structure which stores the Country-typed data items. It uses the BinaryTreeNode class to create its binary tree nodes which hold the left and right children.

The BinarySearchTreeInterface interface requires that the BinarySearchTree class has traversal methods (i.e. the in-order, pre-order, post-order and level-order traversal algorithms of binary trees). To perform the level-order traversal, the BinarySearchTree uses the BTQueue class to create a queue to store the nodes in each level so that the nodes of the binary search tree can be visited one level at a time.

The VaccinationsBST class takes in a file and generates the binary search tree using the BinarySearchTree class's insert method. The VaccinationsBST is used in the VaccineBSTApp, which contains the application's main method, to create the BST data structure. The VaccineBSTApp implements the BinarySearchTree's find() method to search for the user query list.

Experimental Method

An experiment was conducted with the VaccineArraApp and VaccineBSTApp to demonstrate the speed difference for searching between a BST and a traditional unsorted array. Instrumentation was used to count the number of comparison ($<$, $>$, $=$) operations performed in the program. For each application, a static instance int variable called opCount was incremented for each comparison operation.

The size n of the data set was varied and the opCount was recorded for the insert and search algorithms for each data structure. For each subset, the best and worst cases were determined by using one input. For the best case, the query matched the first item in the subset while the for the worst case, the query matched the last item in the subset. The experiment used 10 subsets that were spaced equally apart.
The size of the subsets that were used were:

n = 991, 1982, 2973, 3964, 4955, 5946, 6937, 7928, 8919, and 9919.

The data structures had n items inserted into them and one item was searched each time. Each subset was randomised. Finally, the same number of queries was performed on the same subset for both data structures.

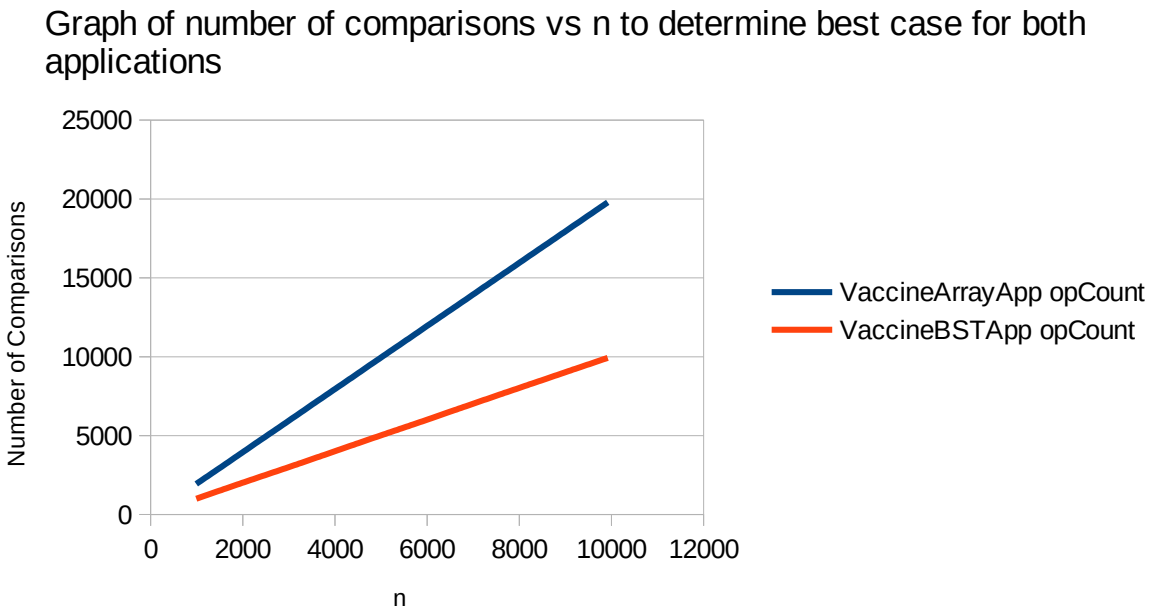
Results

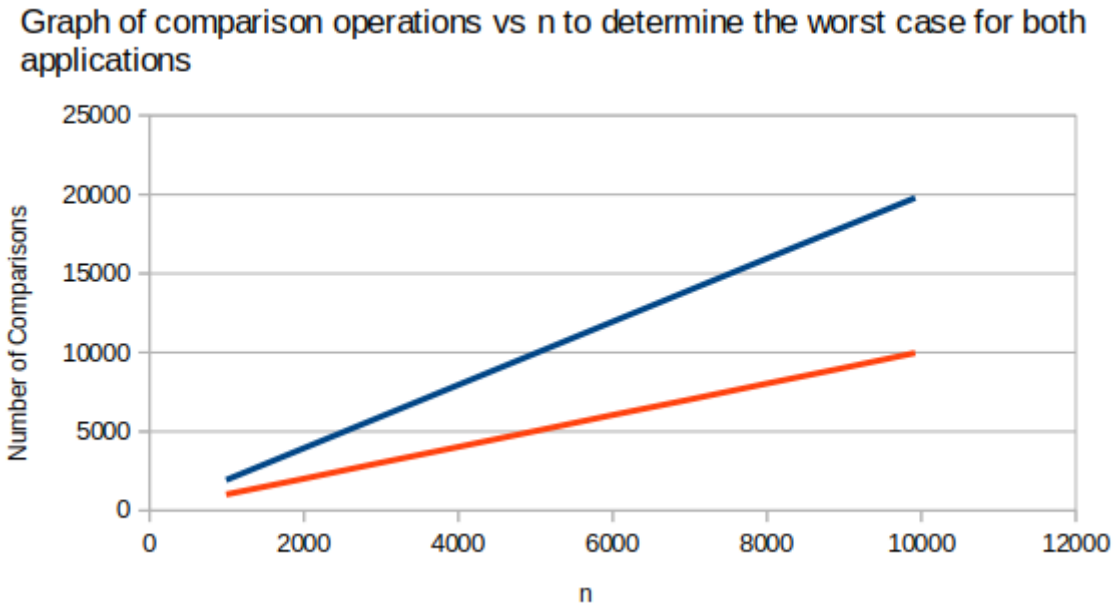
Table 1: Showing the instrumentation results when the size of each subset was varied for both applications to determine the best case

n	VaccineArrayApp opCount	VaccineBSTApp opCount
991	1933	1001
1982	3915	1999
2973	5897	2979
3964	7879	3976
4955	9861	4975
5946	11843	5956
6937	13825	6962
7928	15807	7952
8919	17789	8939
9919	19789	9930

Table 2: Showing the instrumentation results when the size of each subset was varied for both applications to determine the worst case

n	VaccineArrayApp opCount	VaccineBSTApp opCount
991	1933	1019
1982	3915	1999
2973	5897	3004
3964	7879	3992
4955	9861	4989
5946	11843	5993
6937	13825	6965
7928	15807	7965
8919	17789	8954
9919	19789	9972





Discussion

The results show a linear relationship between the number of data items n in the data structure and the number of comparison operations performed by each application. The graphs also show that the VaccineArrayApp which uses an array data structure performs more comparison operations than the VaccineBSTApp which uses a BST data structure.

The results of the best case are as expected since the best case is O(1) for both data structures, i.e. in the case where the data structure is empty and the data item is inserted into the first position and when the searched item is in the first position, this gives the best time complexity. The similarity in the best case is shown be the small gap between the two straight lines when the data subset is relatively small. As the value of n increases, the gap between the results increases as the discrepancy between the efficiency of the data structures increases. The binary search tree is more efficient for larger values of n.

Similarly, in the worst case, the graph of the number of comparisons vs the size of the dataset shows that for smaller datasets, the data structures perform the same. However, as the size of the dataset increased, the binary search tree performed more efficiently than the array data structure.

Conclusion

An experiment was conducted to compare the performance of the binary search tree to the array data structure using two Java programs. The experiment used a dataset of 9919 items which was then divided into smaller subsets of size n in the range [991, 1982, 2973, 3964, 4955, 5946, 6937, 7928, 8919, 9919]. The best and worst cases of the experiment were determined by using one query for each subset size, respectively for both program data structures, and the number of comparison operations were determined programmatically using instrumentation. The binary search tree data structure was more efficient than the array data structure for large values of n. For small values of n, the discrepancy between the performance of the data structures was small as these data structures performed a similar number of comparisons.

Statement of Creativity

The VaccineArrayApp's CountryRegister can be used for the full dataset of vaccinations. It can also produce the list of vaccinations for each unique country. With over ninety-three thousand entries, the class can isolate the 235 unique countries and the total vaccinations on the last date of data collection.

The VaccineBSTApp can use the in-order, pre-order, post-order and level-order traversals to print out all the data for the user.

git log

```
0: commit eae509e517858fa4d60da72bd421e2c583b1aa9b
1: Author: NJMLUN002 <njmlun002@myuct.ac.za>
2: Date: Thu Mar 10 15:42:42 2022 +0200
3:
4: changed infinite while loop in VaccineBSTApp
5:
6: commit 7846e072f2b395a82608fd3f492bfa4acad82823
7: Author: NJMLUN002 <njmlun002@myuct.ac.za>
8: Date: Thu Mar 10 15:38:36 2022 +0200
9:
...
12: commit c7379d12846711d5e381138477108efa91300c00
13: Author: NJMLUN002 <njmlun002@myuct.ac.za>
14: Date: Thu Mar 10 12:19:01 2022 +0200
15:
16: final submission, git first attempt. Apologies for not logging all changes. Will do better
next assignment
```