Irwin Frimpong – November 19th, 2018 – CS 150 Project 2

Introduction

Managing large amounts of data is an inevitable aspect of modern-day computing. Therefore, it is imperative to find ways in which these large sets of data can not only managed but managed in the most efficient way possible. In this project, one was tasked with implementing a container data structure that made use of an ArrayList and a Red Black Tree to store the elements. This was done in means of determining which of the two data structures would be more efficient in carrying out operations involving adding and search for elements.

Considering the constraint of not having duplicates in the Array List as well as maintaining a sorted Array List after each insertion, the Array List would have to be shifted after each insertion, depending on the index of the Array List that it is inserted in. This will cause its insertion operation to take on an O(N) for worst case runtime if elements are to be added to the front or middle of the Array List. For finding the Kth smallest element, because of indexing, the computer can linearly retrieve elements in an Array List when giving a specific index of an element to retrieve; giving the FindKth operation on the ArrayList a runtime of O(1).

On the other hand, since, a Red Black Tree is a balanced binary search tree, this balance is ensured by the rules of insertions involving rotations and recoloring to ensure that the number of black nodes on each path to a null node is consistent. As a result of such, all the operation performed (ie: Insertion and FindKth) will have an O(lgN).

In the scenario where there are more insertions than getKth's, I hypothesize that the ArrayList would run significantly slower than the Red-Black-Tree as a result of the shifting. Further, because the insertion numbers are also randomly generated there is a high chance of shifting happening most likely for each insertion. On the other hand, with the scenario having more findKth than add, I still believe that the Red-Black-Tree will surpass the ArrayList because even though there are fewer additions, the time in which the Array List would add and then performs shifts would be greater than the time the RBT to perform its insertions. This is because the O(N) runtime that the ArrayList uses to perform still surpasses the O(LgN) of the tree. Further, although the Array List has a constant runtime for FindKth, when we take into consideration the cumulative time for both FindKth and inserting to the Array List, I hypothesize that it should still be greater than cumulative time for the actions performed on the RBT.

Irwin Frimpong – November 19th, 2018 – CS 150 Project 2

Approach

The initial design of my program consisted of five classes: IndexSet, ListIndexSet, TreeIndexSet IndexedRedBlackTree, and Experiment Controller. These classes will be discussed in detail in the following paragraphs.

IndexSet class acts an interface for Elements of comparable type E that will be used for the abstract method of add, getKth, and getSize methods in this class. This interface will then be implemented by the ListIndexSet and the TreeIndexSet class.

The ListIndexSet Class is a generic class and as mentioned earlier, implements the IndexSet interface giving it access to the methods defined in that interface. The ListIndexSet is where the ArrayList data structure, used for experimentation, is housed. This including methods add, getKth, getSize, and getArray. Since the ArrayList had to remain sorted after each insertion as well not contain duplicates, the most efficient way to override the add method was to implement a binary search instead of implementing a linear search which would have been inefficient. More details on how the methods were implemented will be discussed in the following section of this report.

The TreeIndexSet Class is another generic class that implements the IndexSet Interface, and thus overrides the methods acquired from the interface. An instance of the IndexedRedBlackTree is created which is the RBT that the operations will be performed on. This class is essential for it houses the RedBlackTree data structure that will be used for experimentation.

The IndexedRedBlackTree class, another generic class, houses the methods that ensure the proper functioning of a Red-Black-Tree. Most of this class was written by Mark Allen Weis, slight changes were made in the add and rotate methods to allow the nodes in the tree to keep track of its sizes; this being imperative for the FindKth operations.

ExperimentController is one of the essential classes in this package, for it is where most of the computation for the experimentation takes place. We'll first discuss the instance variables defined in the class and their significance. The time_250K_arr and time_250K_tree variables are used to store the total time it would take to perform the set of operations in the randomly generated file containing 250,000 elements for the 5 trails ran. The case is the same for the variables: time_500k_arr, time_500K_tree, time_750K_arr , time_750K_tree, time_1M_arr, and

time_1M_tree. The time_arr and time_arr variables are used in the ReadingFile(String y) method to keep track of the cumulative time used by each data structure to process all the operations in a given input file. The methods contained in the ExperimentController include: run() , readingFile(String y) , generate25(int num) , generate50(int num), generate75(int num). The details of this methods will be discussed in the following section.

Methods

ListIndexSet:

Considering the constraints on the insertions in the ArrayList data structure, it was mentioned earlier that a binary search insertion was implemented in the add method to ensure that the addition was as efficient as possible. This was done by making use of the java collection binary search method which is passed the list and the element "e" that one would like to be inserted. The results of the binary search are stored in the variable `index` which would then be used to determine whether or not the elements already exist or if it needs to be inserted. If the former is true, the Boolean value of false is returned by the method; if the latter is true, the element is inserted in the negated value of the index -1 place in the ArrayList. [1]A Boolean value of true is returned by the method.

The getKth method is passed an int parameter k, which would be used in determining the kth smallest element of the Array List. In the method, K is first analyzed to see if it falls within the size of the list, hence the if statement (k >listSet.size()). If such is true a null is returned by the method, otherwise, the element in the (k-1) index is returned from the Array List.

The getSize() of the ListIndexSet returns the size of the Array List at any given moment.

TreeIndexSet

Since an instance the IndexedRedBlackTree(IRBT) is created in this class, the add method calls on the instance of IRBT's add method and is passed the element "e". If the add is successful, a Boolean true value is returned by the method, otherwise a Boolean false is returned.

[1] API, J. (2018, June 23). Class Collections. Retrieved from https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#binarySearch(java.util.List,%20T)

The getKth method like that in ListIndexSet is passed in a value, that value is then passed into the findKth method of the instance of IRBT. The value returned by the findKth method is further returned by the getKth method of this class.

The getSize method essentially returns the size of the right subtree of the header node. This made possible by the getter methods that I had created in the IRBT class.

IndexedRedBlackTree

To allow for the sizes of nodes to be in the RBT to be updated as insertion are made, a new Node is created which will essentially serve as our iterative node called itr. This node is set to the header's right child done with the line: `RedBlackNode<AnyType> itr = header.right;` . Now to increment every ancestor of that newly added node, a search is implemented to find that node, and every node that the itr node lands on before landing on the new node, their sizes are incremented by 1. The size of the newly inserted node which in this case is `current` is also updated by calling the `resetSize(current)` method which adds the size of the node's right and left subtrees and adds 1.

The `findKth(int k, RedBlackNode<AnyType> t) method` is passed in the parameters k and node, which in this case is header's right node. The first thing that is checked is to see if the node provided is null, if such is the case an `IllegalArgumentException()` is thrown, otherwise the size of the of the left subtree is assigned to the variable leftSize if it exists; if not, the variable is assigned a value of 0. Following this, a series of if statements are used to determine whether the kth smallest values are in the left subtree or in the right subtree, or if it is header.right node. Of course, before all these are done, the value of k is checked to see if it is larger than the size of the tree; if such is true, null is returned. [2]

The rotateWithLeftChild and rotateWithRightChild methods are called to perform rotation on the RBT when there are zigzag or zigzag cases in the RBT. The sizes of the nodes rotated in the tree are updated by making use of the `resetSize()` method.

[2] Weiss, Mark Allen. Data Structures & Problem-Solving Using Java. Pearson Education, 2010. Chapter 19: Binary Search Trees

Getter methods are created to retrieve the size of a node, the header node, and well as the right and left children of a node. This was done because these parameters have private access in the class and cannot be accessed outside of the class.

Experiment Controller:

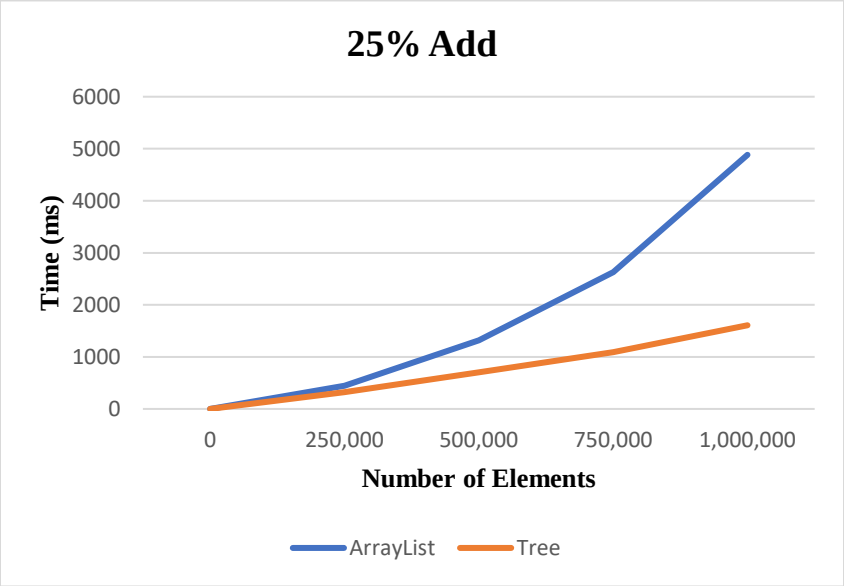
The `run()` method in the experiment controller contains the algorithm that is used for generating the data for the for the trials conducted on the varying sizes of data ranging from 250,000 to 1,000,000. Two for loops are employed in this method, the inner for loop is used in generating the five trails per number of elements; the seconds for loop acts as a select to determine the number of elements used in the experimentation. If the outer loop, whose iteration variable is j is equal to zero, the 250,000 data set is used for the experiment; as the value of j increases the number of elements also increases up until the million. The method was written in such as way that one can run an experiment on using each random generation method each time the program is run by commenting/ uncommenting methods that are either used or not used in each run of the program. The global variables `time_250k_tree`, `time_250K_arr …` are used to store the time each data structure takes to process the elements for each of the trials conducted. These values are averaged and are used for the graphical representations of runtimes displayed in the following section.

The `readingFile(String y)` which takes in a string parameter y is used to set the file name for the scanner. I created the scanner in this method because I wanted to be able to recycle the scanner for each of the trails that I run in my `run()` method without rewriting the scanner's implementation. In the method I instantiate both the ListIndexSet and the TreeIndexSet classes, these two classes hold the ArrayList and RBT data structures respectively. Since, we are analyzing the runtimes of both the ArrayList and the Tree, two print writers, copying the return values of each add and findKth to the output file corresponding to the data type (`output_list`,`output_tree`). Considering that in our input files, functions can either be an add or a FindKth, if statements are employed to deal with each scenario while also keeping track of the time it takes to perform each operation on the data structures. This method also gives the user the convenience of directly calling this method in the main method with an input file he/she may want to be analyzed, without having to drastically alter the program.

The `generate25()`, `generate50()`, `generate75()` employ the same algorithms with differing numbers of add and get Kth's specific to the method. All these methods take in an int parameter

`num` which essentially the number of operations to be done in the randomly generated input. This `num` value is then used for computing the number of add and getKth's; these values are stored in a variable called `num_add` and `num_kth`. For the purposes of our random generation of values, an instance of the random class is created. The `current_size` variable is used to keep track of the of the number of adds performed as they happen in the algorithm to ensure that the random generation of getKth values are properly bonded. To get the alteration of add's and getKth's I create a variable `add` which is set to 1 and `num_kth` which is set to 2. So essentially, I have a random function bounded between 1 and 2, so for each iteration of the for loop, either a 1 or 2 is generated which is then tested against the conditional statements following; these statements not only check to see the type of operation to be performed but also if the required number of that specific operation had already been met. If the aforementioned is the case, the next set of operation that will be added to the file will automatically be of the other type. I decided to first do five adds every time the random generation runs, that was a design choice, it could have been done by randomizing the adds/getKth for the first five runs. Nonetheless, every time an add is performed, I decrement the `num_add` variable by one as well as increment the `current_size` by one.

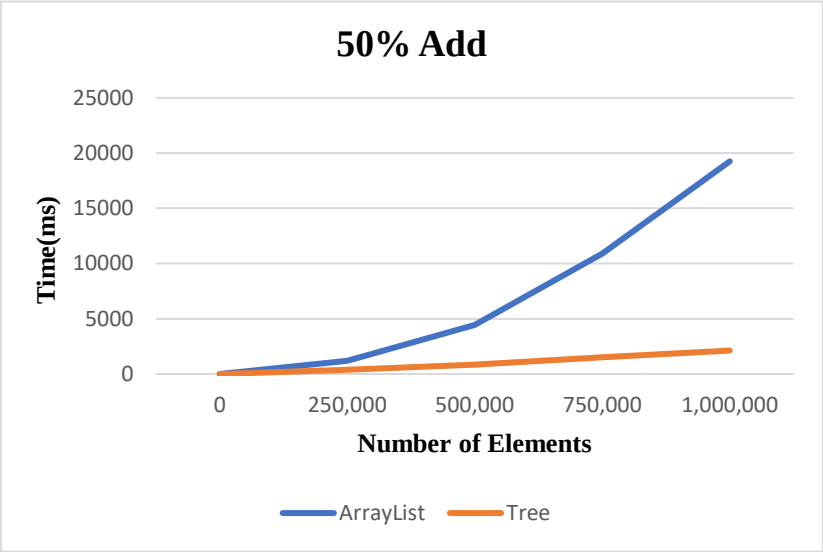
Data and Analysis



Graph 1: Graphical Depiction of 25% Add

Number of Elements	**Runtimes ArrayList**	**Runtimes Tree**
0	0	0
250,000	444.4	321
500,000	1317.2	701.8
750,000	2630.6	1092.6
1,000,000	4883.4	1607.4

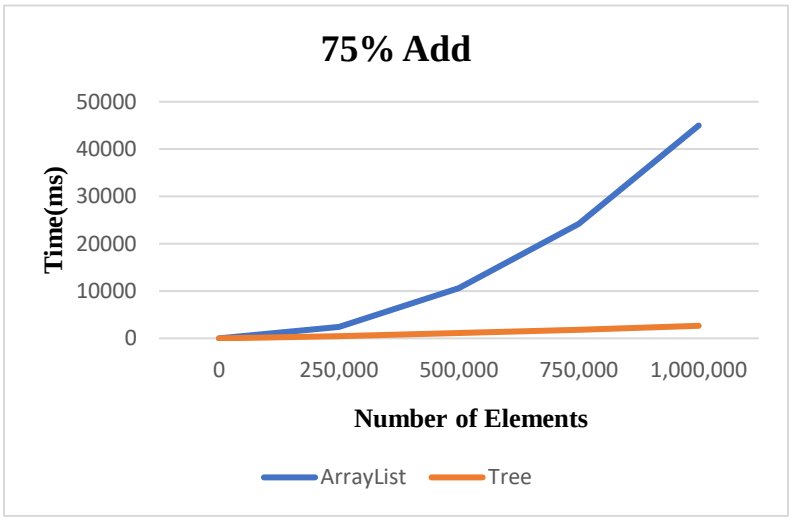
Figure 1: Table of Runtimes 25% Add



Graph 2: Graphical Depiction of 50% Add

Number of Elements	Runtimes ArrayList	Runtimes Tree
0	0	0
250,000	1190.6	386
500,000	4434.8	849
750,000	10905.4	1510.2
1,000,000	19252.4	2125

Figure 2: Table of Runtimes 50% Add



Graph 3: Graphical Depiction of 75% Add

Number of Elements	Runtimes ArrayList	Runtimes Tree
0	0	0
250,000	2403.8	453.4
500,000	10618.2	1108.8
750,000	24198.4	1816.4
1,000,000	44983	2656.4

Figure 3: Table of Runtimes 75% Add

Data Analysis:

Data was generated using the three different random generation methods, the first one being 25% add, the second 50% add and the second 75% add. Each of these methods were run on elements ranging from 250,000 to a million to identify a correlation between runtimes and the number of elements while also taking into consideration the amount of add and getKth operations performed on each data set.

Irwin Frimpong – November 19th, 2018 – CS 150 Project 2

For 25% add, as seen in Figure 1, the Array List and Tree have runtimes that are about 123 ms apart for the data set that had 250,000 elements. The ArrayList had a runtime of 444.4 ms while the Tree had a runtime of 321 ms. The runtime differences do significantly increase as the number of elements increases from 250,000 to a million.; The runtime differences go from 615.4 ms for 500,000 elements to 3,276 ms for the 1,000,000 elements.

For the 50% add and 75% add the same pattern is depicted in the data shown in Figure 2 and 3. The ArrayList in 50% add for the millionth number of elements, clocking in a runtime of 19252.4 ms while its RBT counterpart had runtimes of 2125 ms, giving these two data structures a runtime difference of 17127.4 ms. Moreover, for 75% add for the millionth number of elements similar patterns can be seen in the runtime differences; the array list with a runtime of 44983 ms while its tree counterpart had a runtime of 2656.4, giving it a difference of 42326.6 ms.

In analyzing the data, it is also important that we compare the runtimes according to the number of adds performed. According to the data depicted in Figure 1-3, it is evident that as the percentages of adds increase, irrespective of the number of elements, the runtimes for both data structures increases; however, the runtimes of the ArrayList increases faster than of the Red-Black-Tree. For specificity, the runtimes for the ArrayList for 25%,50%, and 75% add for 250,000 elements are reported as 444.4ms, 1190.6 ms, and 2403.8 respectively. On the other hand, the RBT has runtimes of 321 ms,385 ms, and 453.4 ms respectively. There is a drastic increase in runtimes as the number of adds increases and this is because of the shifting being done on the ArrayList with every addition made, which is highly expensive considering that such process is done with a theoretical runtime of O(N). With the RBT on the other hand, although rotations are performed with the some of its insertions to remain balanced and preserve its RBT properties, insertions are done theoretically with a runtime of O(LgN); this is significantly faster than the ArrayList's O(N).

Conclusively, the same pattern is exhibited in all the Graphs 1-3, since the RBT's theoretical runtime is O(lgN) its runtime graph for all the add methods grows significantly slower than that of the ArrayList.

Conclusion

Based on the results obtained from experimentation, the Red-Black-Tree is more efficient when it came to 25%, 50%, and 75% add methods. Its runtimes were significantly smaller than that of the Array List on each set of trails run. This further affirms my hypothesis, in that the ArrayList did take a significant amount of time to performs its operations in each scenario of adds, considering its larger theoretical runtime of O(n).

Therefore, if it was a choice between an Array List or a Red Black Tree data structure, it would be beneficial to implement a RBT, for one is guaranteed guaranteed efficiency.

References

1. API, J. (2018, June 23). Class Collections. Retrieved from https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#binarySearch(java.util.List,%20T)
2. Weiss, Mark Allen. Data Structures & Problem-Solving Using Java. Pearson Education, 2010. Chapter 19: Binary Search Trees