Dai Hua Chen and Anastasia Erofeeva
CSE 373
February 25, 2018

Project 3 Part 1e: Group Writeup

Formulas

Find parent of some node i: parent(i) = (i - 1) / 4

Find j-th child of some node i: child(i, j) = 4i + j

Refactoring percolateDown Redundancy

To refactor this redundancy, instead of comparing the parent index with all four of its children, we created a new private method called getSmallestChildIndex, which compares all the children of the given parent index and returns the index of the smallest child. This design decision helped us reduce the number of times needed to compare a parent node to its children nodes and then determining which node it should swap with as we would only be comparing the parent node to its smallest child. One major challenge we ran into was getting the index of the smallest child when the parent index either didn't have any children, or had less than 4 children. We ended up overcoming this challenge by drawing several examples on paper to help visualize the scenario and also creating a new toString method inside the array heap and printing out the array heap during the jUnit tests to help us debug.

Experiments

1. **Summaries**

   Experiment 1 tests the amount of time it takes to return the top 500 elements in sorted order from lists of various sizes. The sizes of the lists range from 0 to 200,000 in 1,000 step increments.

   Experiment 2 tests the amount of time it takes to return the top K elements from a list in sorted order 10 times. The K values range from 0 to 199,000 in 1,000 step increments.

Experiment 3 tests the ChainedHashDictionary with three different i implementations of hashcodes. Since the client defines the getHashCode() method for the dictionary, the variations in getting hashcode results in variations in performance of the dictionary.

2. **Predictions**

   For Experiment 1, we predict that the more items there are in the input list, the more time it will take to return the top 500 sorted items from that list.

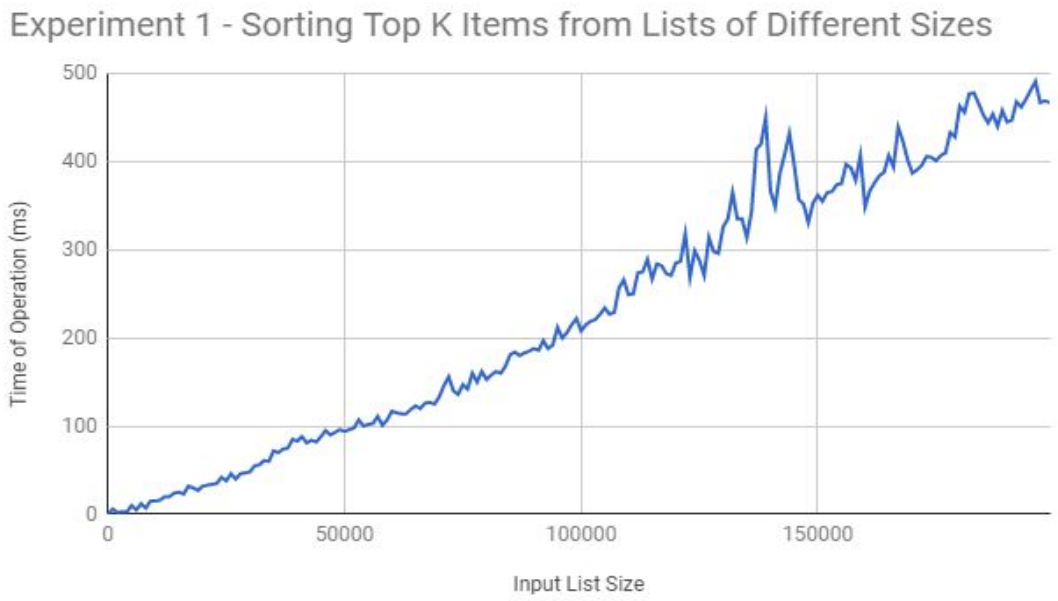
   For Experiment 2, we predict that the larger the K value, the more time it will take to return the top K sorted items from a list, because a bigger K value indicates that more values will need to be sorted. Sorry, sounds good, thanks haha

   For Experiment 2, we predict that the runtime for each K value should be relatively the same as we are inputting the same list size into our searcher method. The way our method works is by adding every element in the input list into an ArrayHeap, which sorts everything, therefore the runtime should be roughly the same.

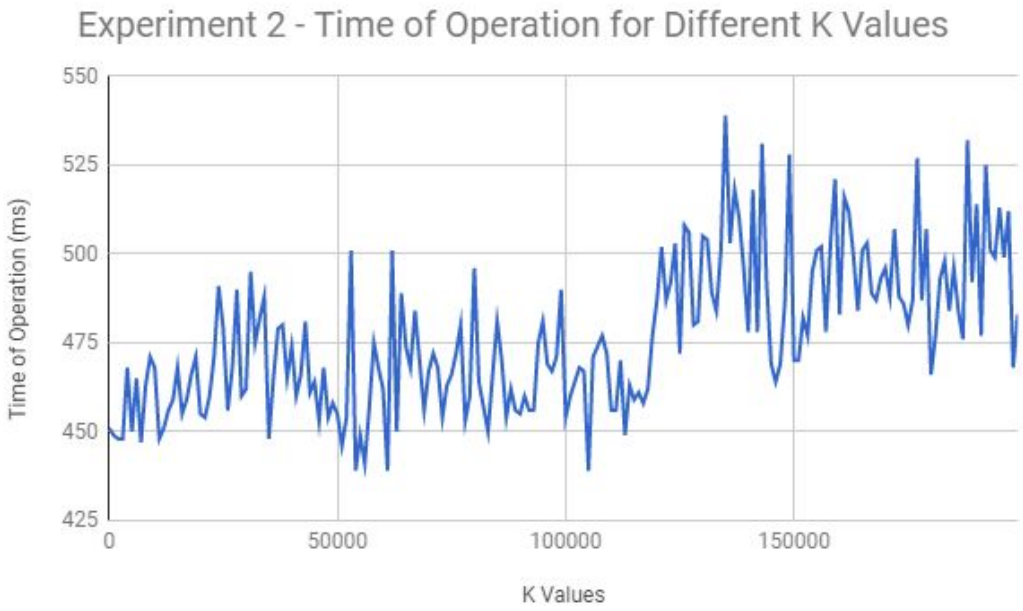
   For Experiment 3, we predict that FakeString 3 with the variation of hashCode() similar to Java's list.hashCode() method would exhibit the best performance with the least runtime. FakeString 1 would have the highest runtime as it is only adding up the values of the first 4 characters of each String, whereas the second hashcode function adds up the values of all characters, adding more variation to the hashCode function. FakeString 2 has the most variation to its values as it multiplies each value by 31 before incrementing.

3. **Results**

   For Experiment 1, we correctly predicted that the larger the input list size, the longer the time of operation will be for returning the top K sorted items from the list. The plot below shows that the time of operation grows almost linearly as the input list size increases. This makes sense, because the topKSort method has to traverse through all of the elements in the list, which requires more time the bigger the list is.



For Experiment 2, although there are many variations in our data, the average of all the data points are about the same. However there is a slight increase overall as the value of K increases. This is most likely due to the constant time of operation it takes to add elements into a DoubleLinkedList (constant time because it is double-linked). As the values of K are smaller, most elements of the ArrayHeap is returned and not used and K elements are returned and added into a LinkedList as output. As K increases, the size of the list increases and, therefore, the number of items to be added to the list as well.



For Experiment 3, we correctly predicted that FakeString 1 would have the highest runtime as the hashCode function is declared very ideally for the dictionary. However, to our surprise, the hashCode function for FakeString 2 had a lower runtime than FakeString 3. We think this is because initially there might not be a lot of characters in our dictionary, which would result in a very large dictionary with only a few elements inside as the FakeString 3 getHashCode() function has a much wider range of values than FakeString 2. Therefore, it would take longer to traverse through the dictionary constructed by Fakestring 3 than FakeString 2.