Project 3 Group Writeup: Part 1e

1)

- Equation to find parent(i)
    - parent(i) = (i – 1) / 4
- Equation to find child(i, j)
    - child(i, j) = 4i + j

2) When implementing the percolateDown algorithm in ArrayHeap, we decided to construct a helper method in which we did recursion to percolate down. This allowed us to only have our checking code in one location instead of copying and pasting the check code in four separate locations for each child in the heap. We designed our percolateDown method to check for the smallest child value that is less than the current parent value. If that child is found, our algorithm will then recurse through that child until the array heap fulfills the invariant of children's values being larger than the parent's value.

One of the challenges that came up was that our algorithm could be inefficient if it recurses through the first child value that is less than the parent value. This would true in cases where later children's values are also smaller than the first child's value since the array heap would then require multiple rounds of recursion. To approach this challenge, we decided to first find the smallest child value before recursing, which would ensure that we only recurse once through heap to properly percolate down.

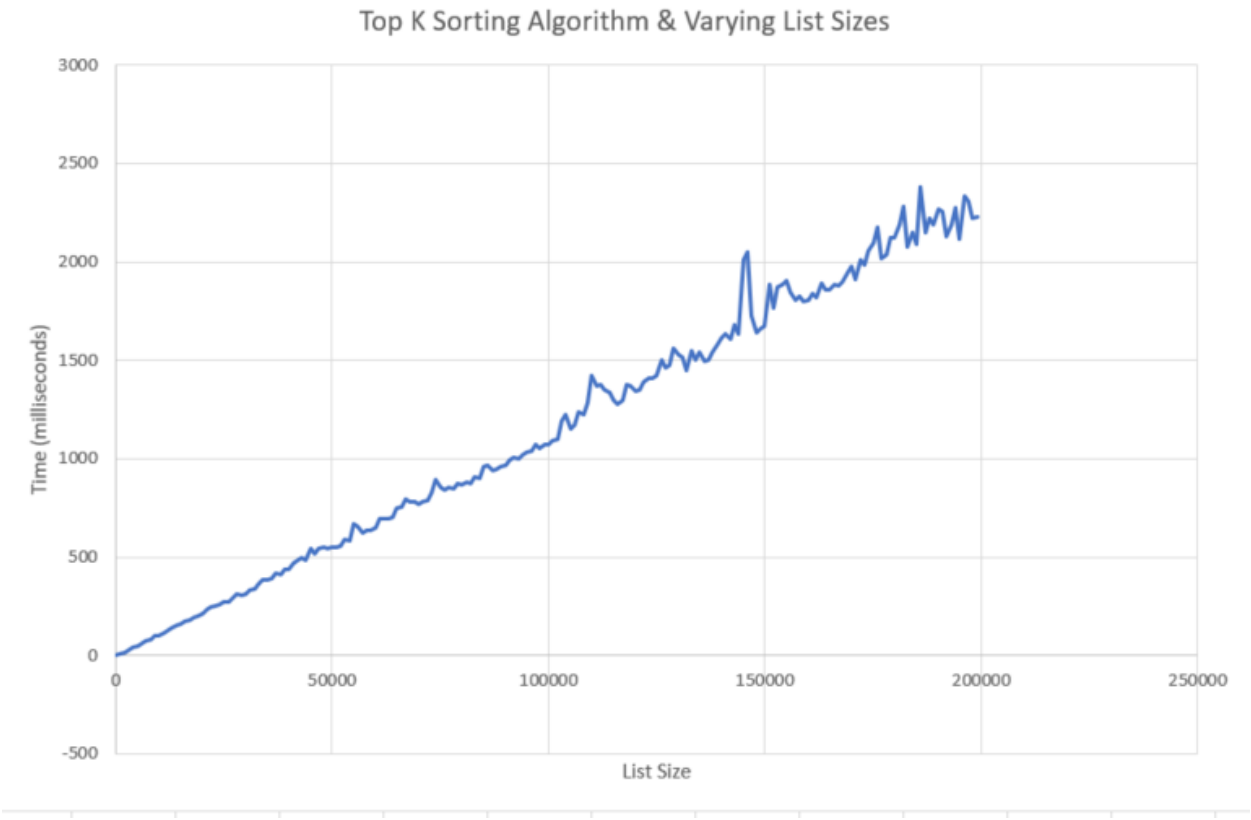
3)

Experiment 1:

1) Experiment 1 is testing out the top k sorting algorithm speed (in time) by creating lists that increase in size until a particular max value size. Each list is then tested by the top k sorting algorithm by always finding the top 500 values.

2) For our hypothesis, we believe that the larger the list becomes, the longer it will take for the experiment will take because there will be more values to go through. The time it takes to percolate down will always be, at worse, O(k) where k is the number of values that you are expecting. However, you still have to traverse through the entire list, which means it should take longer for larger lists.

3)



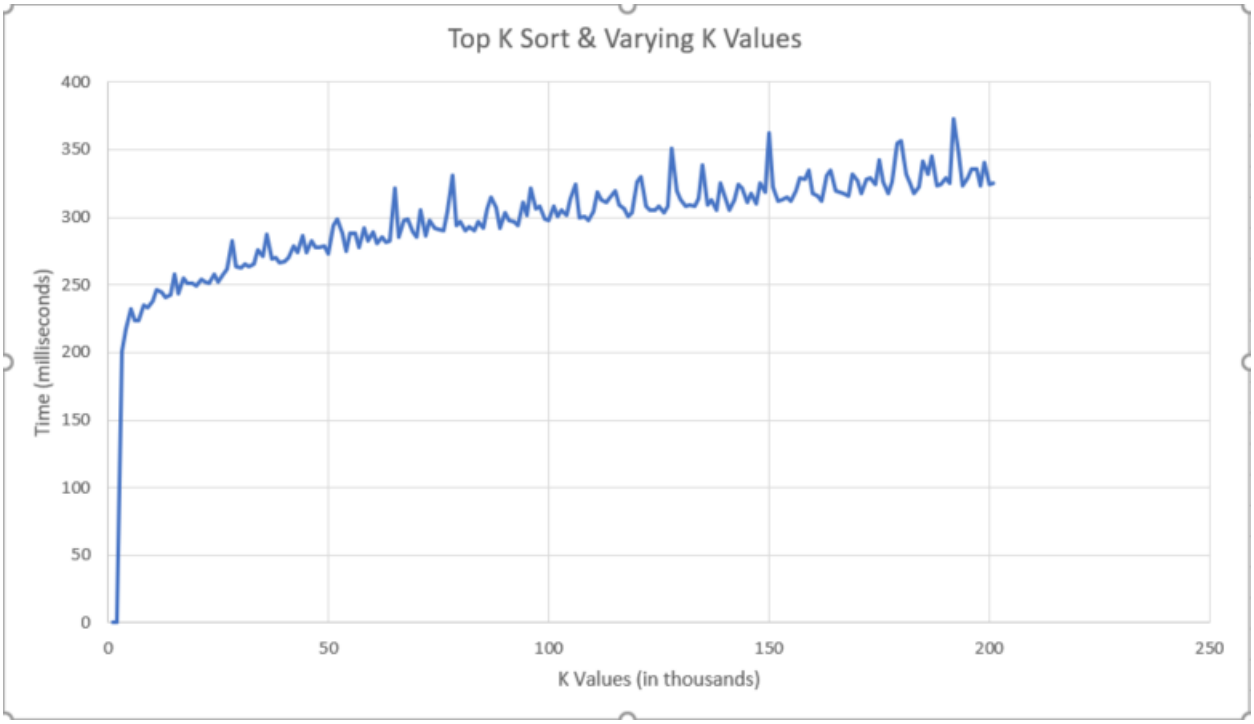
4) The results from the experiment supported our hypothesis. The times increase as the list size grows larger as we predicted. The larger the list, the more values our algorithm must traverse through which will take more time. The number of top values to be found remains constant at 500, which further supports our reasoning of how the list size is the difference maker in the recorded times. In other words our algorithm is nlog(k) where n is the size of the list and k is the top values to be found. K in this case is always constant while n is increasing which explains the increase in time.

Experiment 2:

1) Experiment two is testing how our top sort k algorithm performs while k increases as the list remains at a constant size.

2) For our hypothesis, we believe that the larger k is, the longer the resulting time will be. Our top k sorting algorithm runs O(nlog(k)) and as k increases then our runtime results should also increase steadily.

3)



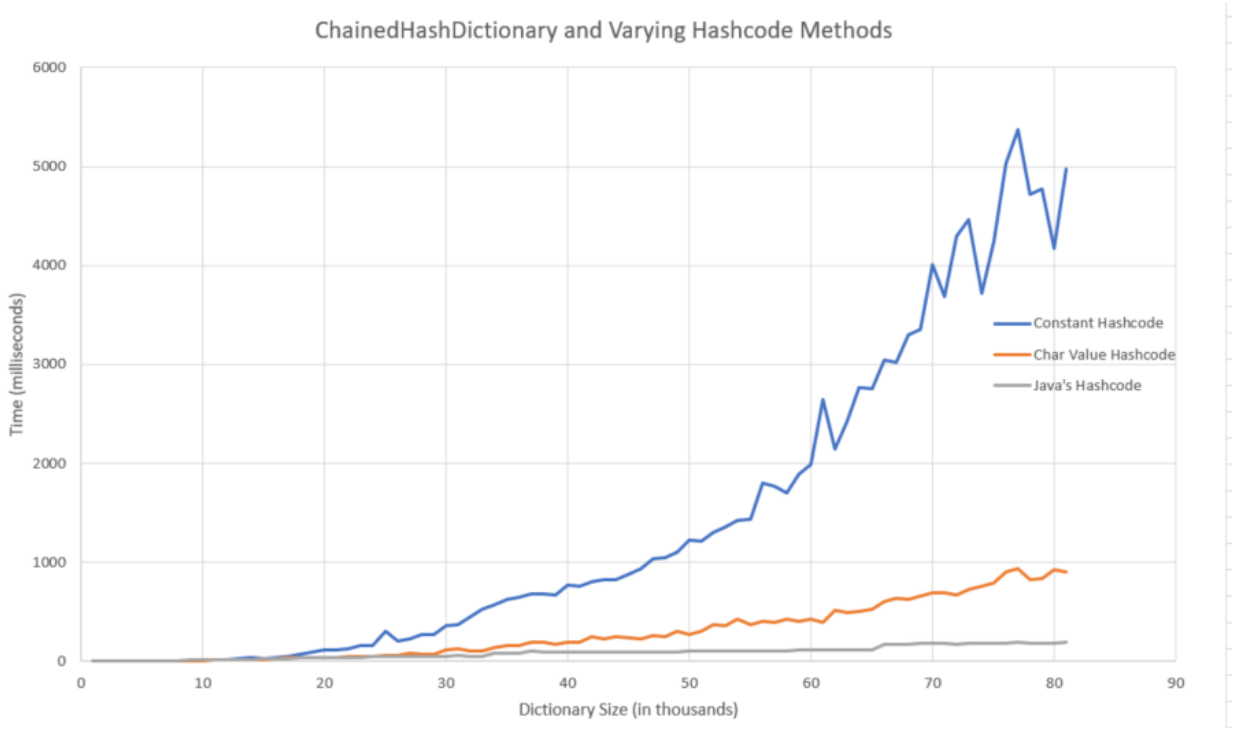
4) The results from experiment 2 support our hypothesis. There seems to be an initial spike due to k starting at 0 which returns an empty list. After k increases to 1000, and all k values after, there seems to be a consistent gradual increase in the time results.

Experiment 3:

1) Experiment 3 tests our ChainedHashDictionary with varied FakeString classes that had different hashcode methods. One hashcode method returned the same constant in order to make the chainedhashdictionary take a hideously long amount of time, the other one just returned the character integer value, and the last one basically used Java's list.hashcode() method to return the corresponding hashcode.

2) For our hypothesis, we believe that the test with the hashcode that basically does Java's list.hashcode() method will have the fastest time results since that it actually alters the character value by multiplying by 31 and adding the character value. This will reduce collisions within the chainedhashdictionary and therefore, have lesser result times. The next fastest test would be test 2 because having the hashcode be the character value will have less collisions than test 1 where its hashcode is returning the same constant.

3)



4) As we predicted, Java's hashcode method results were the most efficient out of the three hashcode methods. Again, this is most likely due to how the hashcode method adds in steps to alter the value returned which will reduce collisions. The next most efficient test was the char value hashcode. This hashcode method simply returns the char value. It does not have a algorithm to reduce algorithms like Java's hashcode method, but it will still have less collisions than the first test which returns the same constant. The constant returning hashcode method, over time, will be very inefficient due to the high number of collisions since the same constant is being returned from every call to that particular hashcode method.