Jonathan Barton
Mitchell Orsucci
CSE 373 Project 3
Winter 2018

1. **Finding Parents and Children in a 4-Heap**

The formula to find the parent of a given node i was : **parent(i) = (i-1)/4,**
The formula to find the jth parent of a given node i: **child(i, j) = (4*i) + j + 1**

2. **Checking Children for percolateDown()**

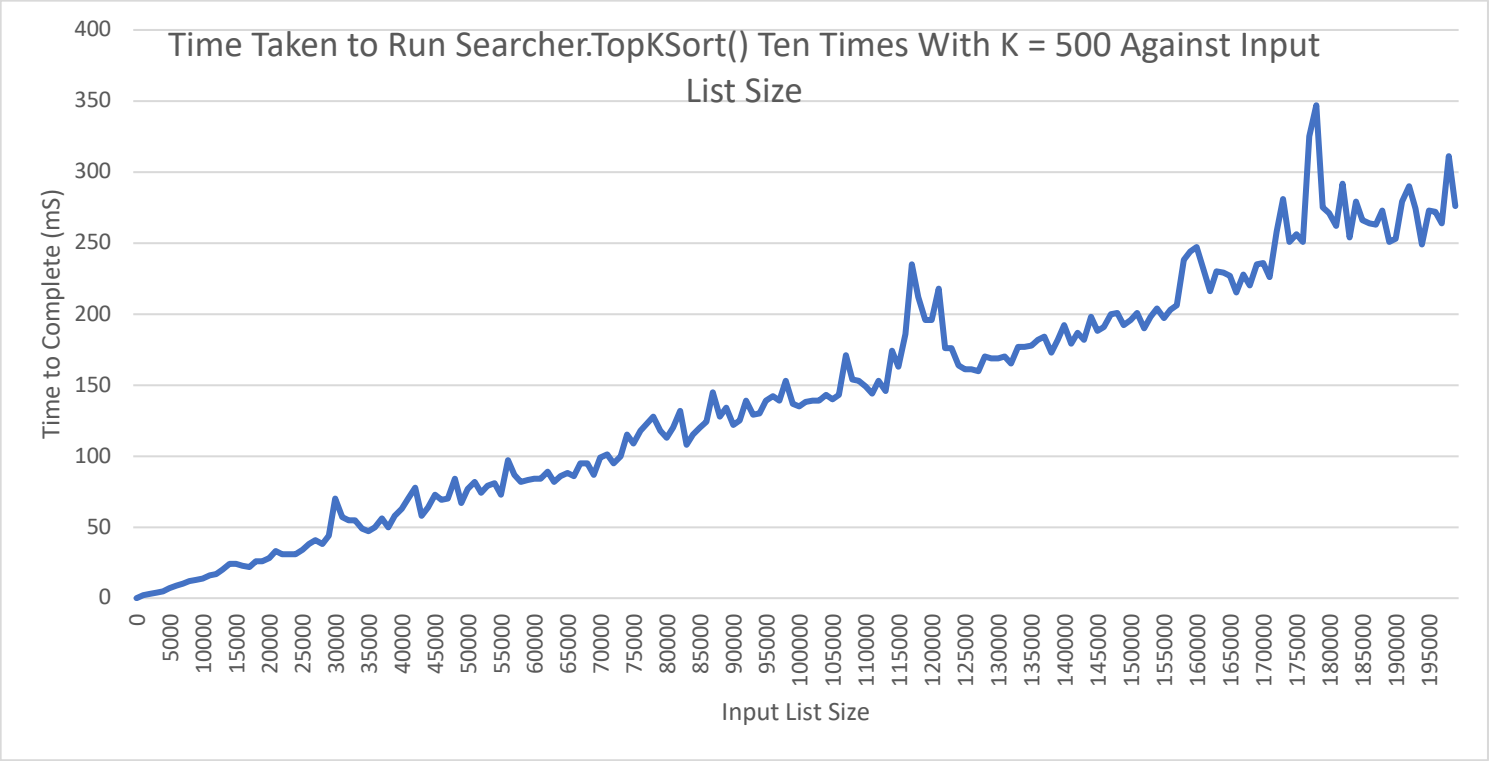
Instead of checking each child against every other child, I simply looped through the list of children one time and saved the 0th index as the current smallest child, then moved on to the next child, compared it to the current smallest child, and replaced that if it was smaller. This means that I only need to make three comparisons to end up with the actual smallest child. Unless the children are sorted as you place them in the list then there isn't much that you can do to improve the speed, and sorting the children would introduce a great deal of complexity to the binary tree.

3. **Experimental Data**

    a. **Experiment 1**
        i. This experiment measures the amount of time required to run Sorted.TopKSort() 10 times with K = 500 on lists varying from size 0 to size 200000 in integer increments. This test is run five times and then the results of each test are averaged against each other.

        ii. As our implementation of TopKSort() runs in O(n * log(k)) time, we expect to see our results for duration of the test increase in a linear fashion since the factor ‘log(k)’ is the same for each test and n increases in a linearly.

iii.



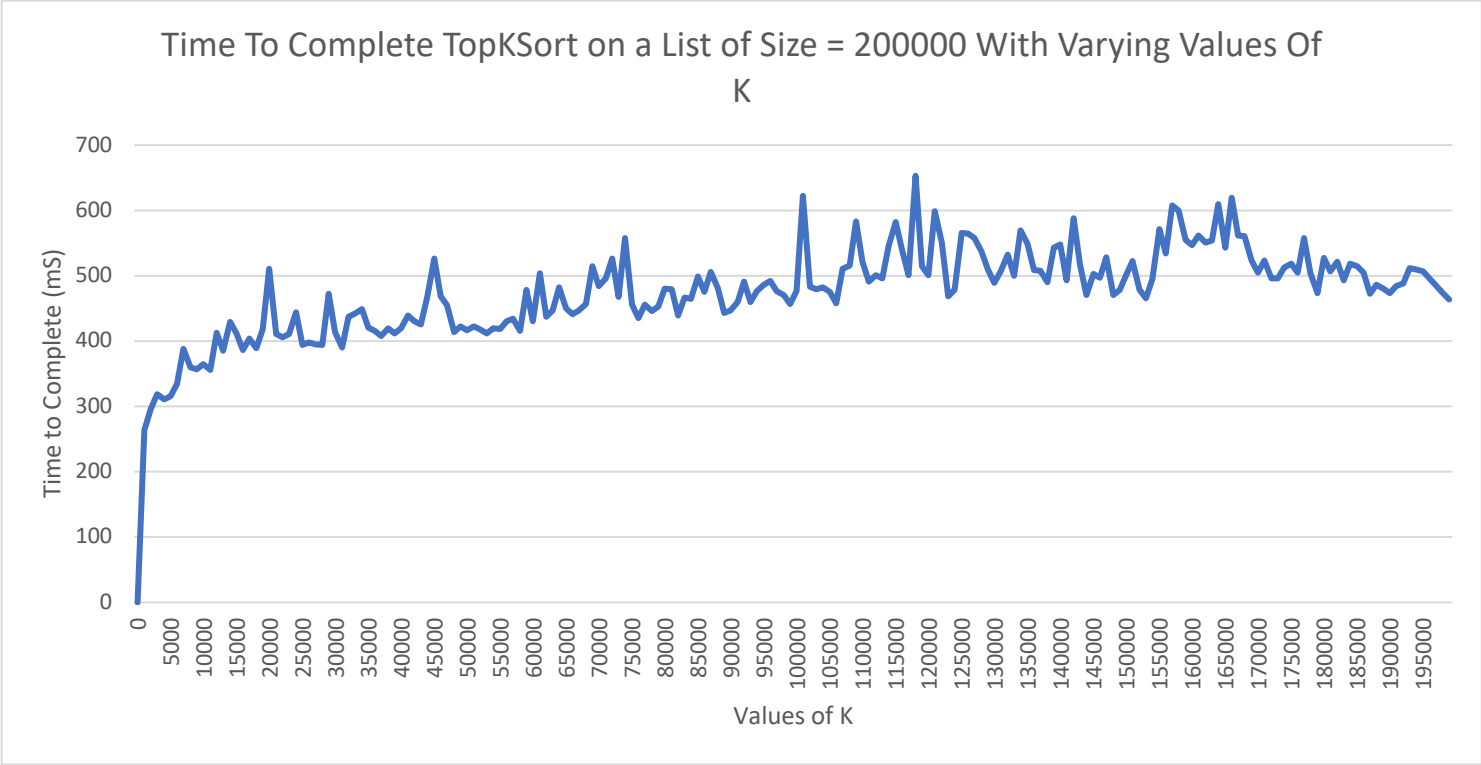
iv. According to our test results, we see that time to run the test is mostly linear with respect to the input list size as we expected. There are some non-linear spiked in the graph but we suspect that this has more to do with memory performance than anything else.

b. Experiment 2

i. This experiment builds a list of integers from zero to 200000 inclusive and runs Searcher.TopKSort() 10 times on the list with varying values of K. The values of K tested are between 0 and 200000 with increments of 1000. The amount of time it takes for TopKSort() to complete this call is the measured metric. This test runs 5 times completely and averages the results of all the tests together.

ii. As the efficiency of TopKSort() relates to n * log(k), and n is the same for each test, we expect that the amount of time it takes to complete the test given different values of k will be logarithmic in nature.

iii.



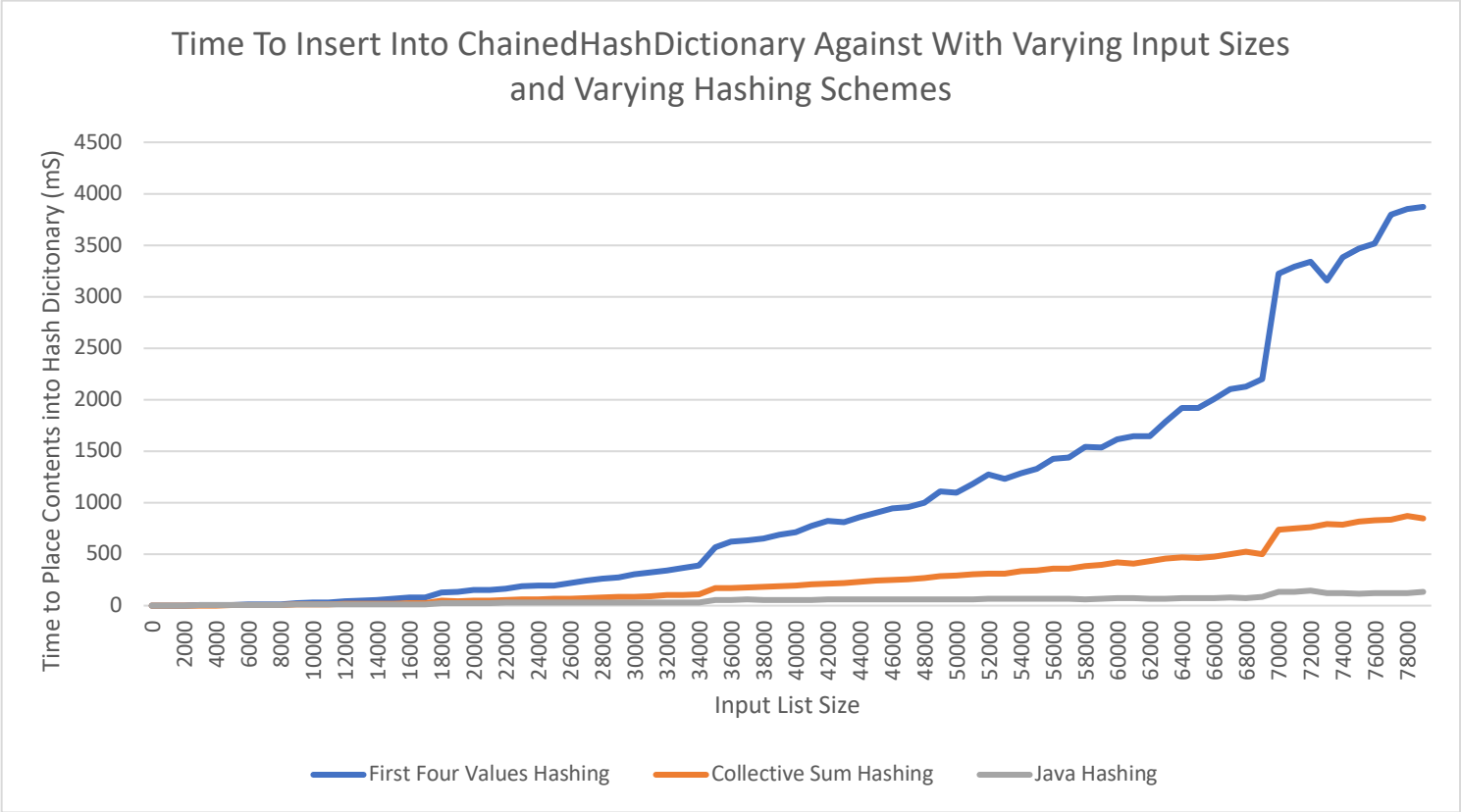
iv. We can see from the plotted results that there is indeed a logarithmic nature to the relationship between the amount of time to run the test and the increasing sizes of K.

c. Experiment 3

i. This experiment creates arrays of random characters and then tests the amount of time it takes to insert all of these characters into our Chained Hash Dictionary. The test runs on input list sizes from 0 to 80000 in steps of 1000. Additionally, this test is run with three different hashing implementations for the collections of characters. The first scheme uses the value of the first four characters in the array. The second scheme uses the total sum of each character in the array. The third scheme uses the implementation that Java uses. Results are averaged out over five runs.

ii. Given that the performance of the ChainedArrayDictionary ultimately relies on the implemented hashing scheme, we expect that the third test will run fastest, the second test will be next fastest, and the first test will be slowest as it will cause more collisions to occur. With more collisions, there are more items occupying the same bucket, and getting from the double linked list at the bucket will take more time.

i.



iv. From the results, we can clearly see that the Java Hashing implementation results in the fewest collisions in the ChainedHashDictionary. Additionally, the Collective Sum Hashing approach operates with fewer collusions than the First Four Values hashing scheme.