The formula to find the index of the parent node is (i-1)/4. The formula to find the children where i is the current node a j refers to which child is 4i + j.

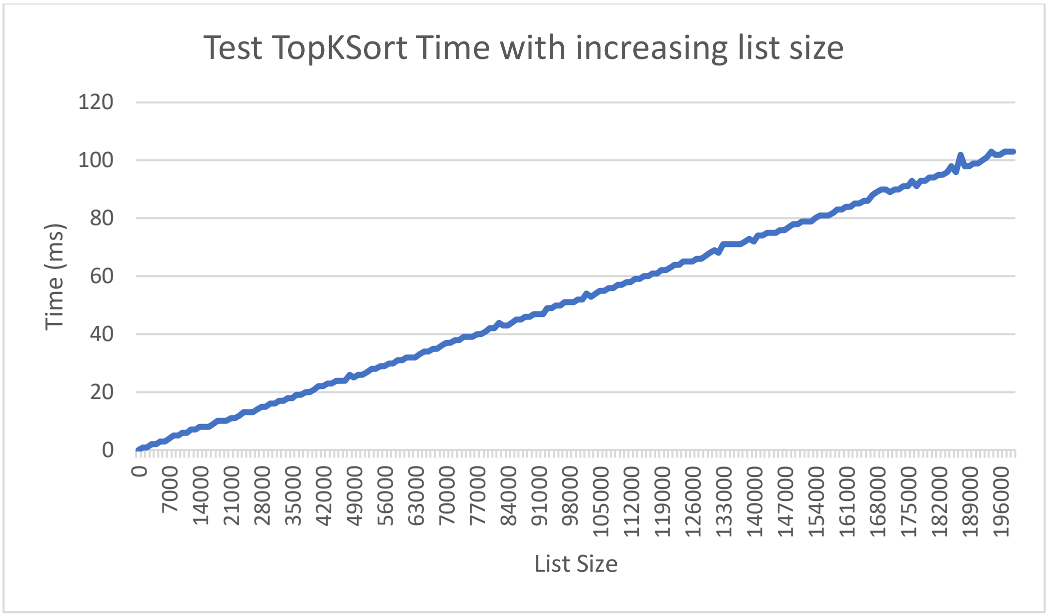
To get the child with the lowest value for a node we implemented a for loop and used the compareTo method. We set the first child to the min value then compared it to each of the other nodes using a for loop. If a node is less than the min value, then that node becomes the min value and that is used going forward as the min value. This is how we get the overall min value. To get the next row of children for percolate down we use 4i + j and then repeat the compare process.

Experiment 1

When my partner and I looked at the codes from experiment 1, we believe that experiment 1 is constructing a double-linked list with the minimum size of 0, the maximum size of 200000 with increments of a 1000. In the test part, we believe that the test codes are testing the topKsort function. The test code will iterate ten times, and each time the TopKSort function will return a list consisted of 500 largest values from the double-linked list. The purpose of the test is to see how efficient is the TopKSort function when the input list size is increasing.

Hypothesis: we believe that as list size increases the TopKSort time will increase as a factor of list size.

In experiment 1, the K is 500. Log(500) is 2.6989, thus K does not generate a large effect. Therefore, the time is more closely linked the list size which is our hypothesis.



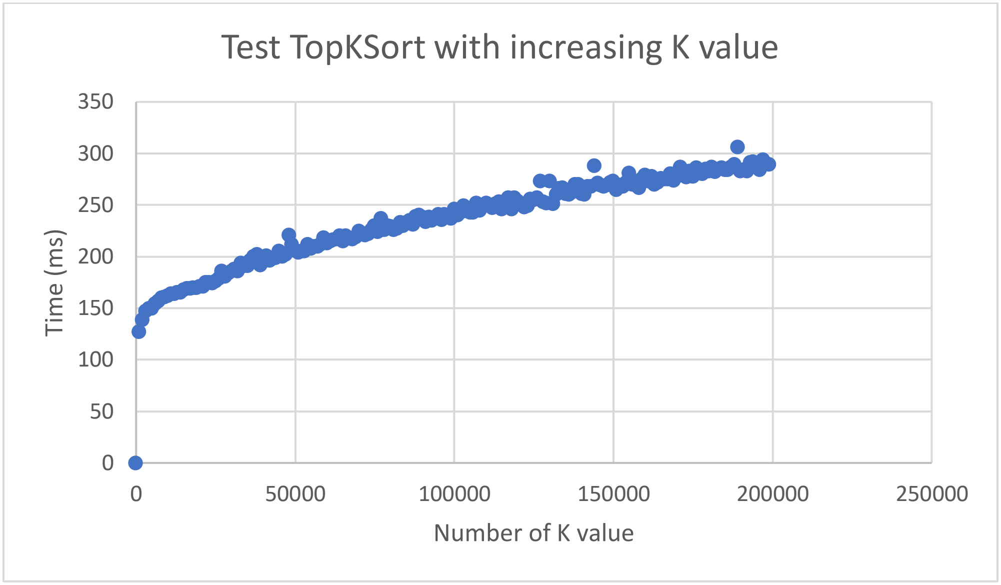
The result adheres to our hypothesis. We can see the pattern from the plot. As list size increases, the time it takes sort largest K value goes up as well. We can observe a linear relationship between the graph; it is apparently a relatively straight line. This straight line is not surprising after all. The runtime of TopKSort is N*log(K). In the most situation, log(K) is constant compared to increasing N. List Size has a more significant effect compared to K. Based on those, the result from experiment one is reasonable.

Experiment 2

When my partner and I look at those codes from experiment 2, we believe that experiment 2 is constructing a double-linked list with the constant size of 200000 and K value that starts at 0 and makes increments of a 1000. In the test part, we believe that the test codes are testing the TopKSort function. The test code will iterate ten times, and each time the TopKSort function will return a list consisted of K largest values from the double-linked list with increasing K value. The purpose of the test is to see how efficient is the TopKSort function when K value is increasing.

Hypothesis: we believe that as K value increases the TopKSort time will increase as a factor of log(K).

In experiment 2, the K is increasing at rate of 1000. Since the list size is 200000 and is not changing at all. So the list size will not have significant effect on run time. Therefore, the time is more closely linked with K value which is our hypothesis.



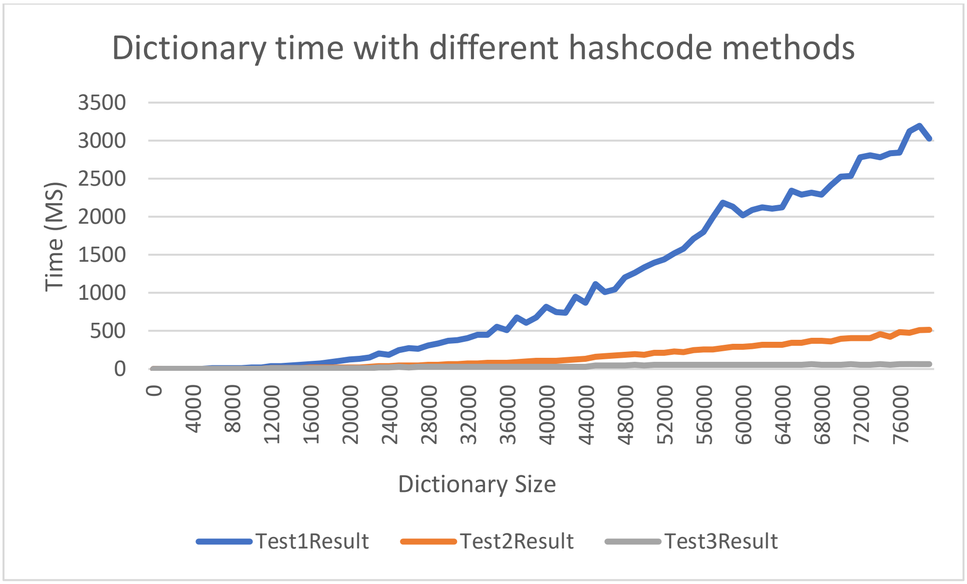
The result adheres to our hypothesis. We can see the pattern from the plot. As K value increases, the time it takes sort largest K value goes up as well. We can observe a log relationship between the graph; it is apparently a relatively log graph. This log graph is not surprising after all. The runtime of TopKSort is N*log(K). Since the list size is constant the graph only increase with a factor of logK (only K is increasing N stays constant).

Experiment 3

When my partner and I look at the codes from the experiment 3, we noticed the codes consist of three tests. The essence of those three tests are largely the same. The main goal is to put an entire character array and entire list that consisted of character arrays into ChainedHashDictionary. Although the main methodology is largely the same, other aspects are different. For these three tests, they use different fakeStrings1, fakeString2, fakeString3. The main difference of those three are the calculation methods for hashcodes.

Hypothesis: the time of test 3 is faster than the time of test2, and the time of test 2 is faster than test 1.

Explanation: We looked at the hashcodes for these three fakestring cases. We concluded that test 3's hashcode methology is the best most effective dealing with collision. This hashcode methology multiply 31 each time and add character value. The second best is the test 2’s hashcode methodology, which is adding all of the characters value together. The worst hashcode methodology is the test 1, which is just add first four characters value together.



This graph adheres with the hypothesis. The test 3 most effectively deal with collision, and its runtime is the fastest among three. It represents an almost O(1) time. The test 2 result is also pretty fast, but it is not as efficient as the test 3. The worst situation is test 1, because the hashcodes method only add the first four characters value, which is not ideal for dealing with collisions. This is because the when adding only the first 4 characters together, the number of hashcodes is much lower than when adding all the letter values up. This increases the amount of collisions that happen.