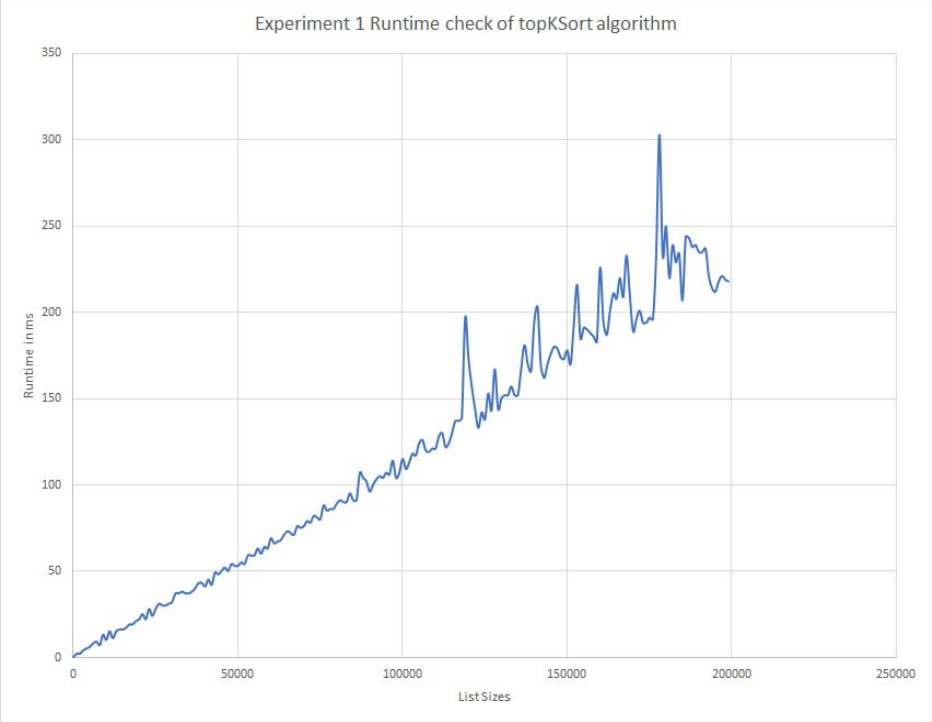
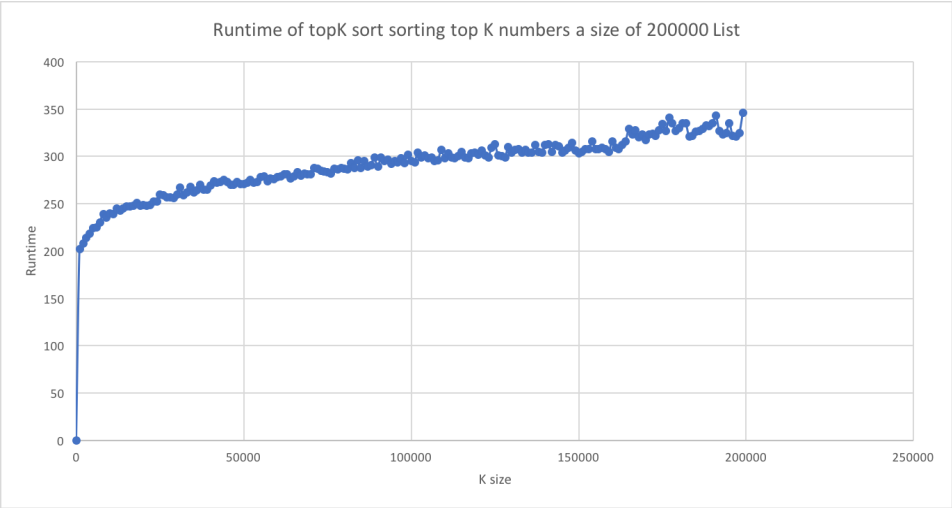
1.
    a. Parent(i) = (i-1)/4
    b. Child(i,j) = 4 * i + j
2. To reduce redundancy, we introduced a private helper method to find the min value index for us. We first set the parent index to be the min index value, loop through its children, update min index if we find a smaller value in its children. Then we return that min index to our precolateDowm algorithm. We call this helper method each time when we need to find a index to percolate down our current value.
3.
    a. Experiment 1:
        i. This test test the average time of our topSort function running ten times on sorting top 500 numbers for the lists which size range 0 to 200000 with step of 1000.
        ii. The outcome of the result will be 200 time data with a runtime of n(log base 4, 500) where n range from 0 to 200000 with step of 1000. The runtime will be linear since the size of the list increases.
        iii.



        iv. The graph of execution basically matches our prediction. The graph tends to be linear. While the list size increases, the runtime will increases, but the increments are the same since our Ksort takes n * log4(500) where log4(500) is a constant. We assume those spikes are resulted from machine performance.
    b. Experiment 2:

i. This test test the average time of our topSort function running ten times on sorting top K numbers range from 0 to 200000 with step of 1000 with list size 200000

ii. The outcome of the result will consist of 200 rows of data with a runtime of 20000 * log 4 (K) where K ranges from 0 to 200000 with step of 1000. Therefore the graph will be in the form of log.

iii. 

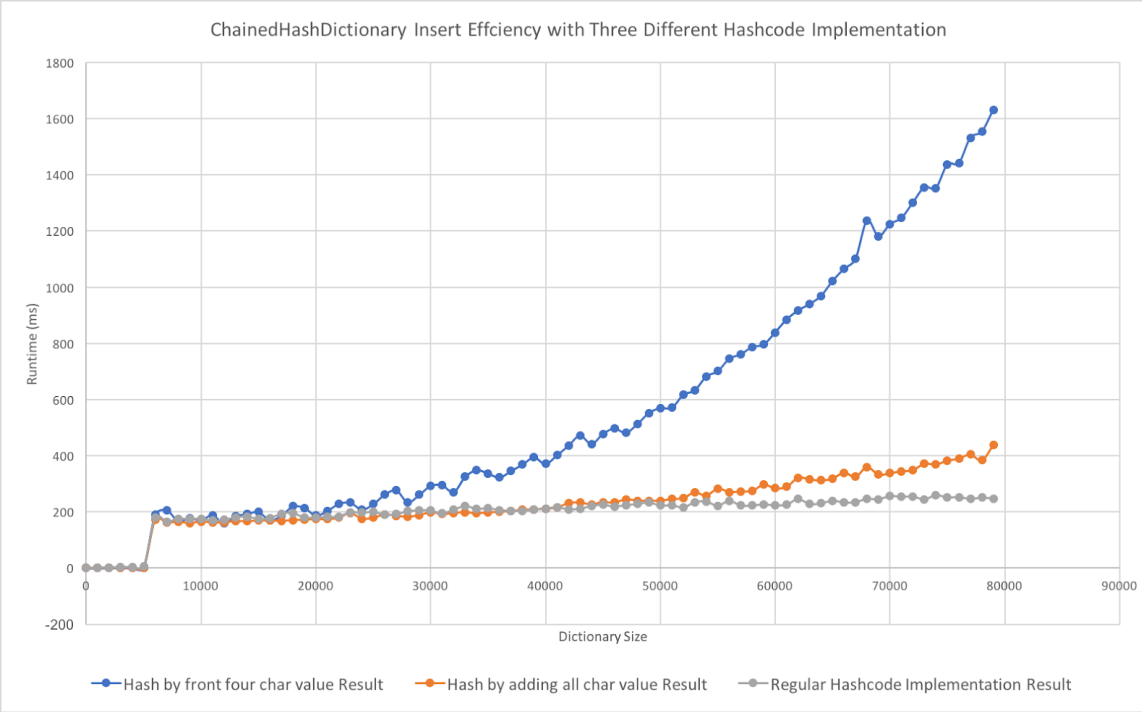
iv. The graph of execution basically matches our prediction. The graph tends to be logarithm.

c. Experiment 3:

i. This test test the average time of our ChainedHashDictionary insert efficiency with four different hashcode strategy by creating dictionaries with sizes range ranges from 0 to 80000 with step of 1000 and insert a list of random character arrays with 200 char with list size range from 0 to 80000 with step of 1000.

ii. According to the hashcode implementation, first test will have a highest run time because of high collision rate of the hashcode method, the last one will have least runtime because of the lowest colllition rate hashcode implementation.

iii.



iv. The graph of execution matches out prediction: hashing by adding first 4 characters take most of the time because the rate of collision is the highest. Hashing by all char value result in way less collision and use prime number 31 as hash method take least rate of collision.