Micah Steinbock – 644014

Alex Peterson – 660863

CSC312 – Algorithms

Dr. Han

Project 3 Writeup

1. Parent (i) = (i – 1) / 4

   Child (i, j) = (i * 4) + j ← where j is range 1 – 4

   Or Child (i, j) = (i * 4) + (j + 1) ← where j is range 0 – 3

2. In order to reduce the redundancy of the percolateDown checks, we created a helper methods that analyzed the validity of the children as well as found the smallest child's index. The first helper method was used to check the validity of the children. It used a for loop from the index of the first child to the index of the last child and performed the Boolean check with the loop's counter (i) which was equivalent to the index of one of the children. We used the same principle for finding the smallest child's index, where we used a for loop to iterate over the children and perform the comparison. We did this because we noticed that the checks and comparisons were the same code, just with different indexes that were incremented by 1 from the previous, which made it really easy and clean to use a loop to perform these operations.

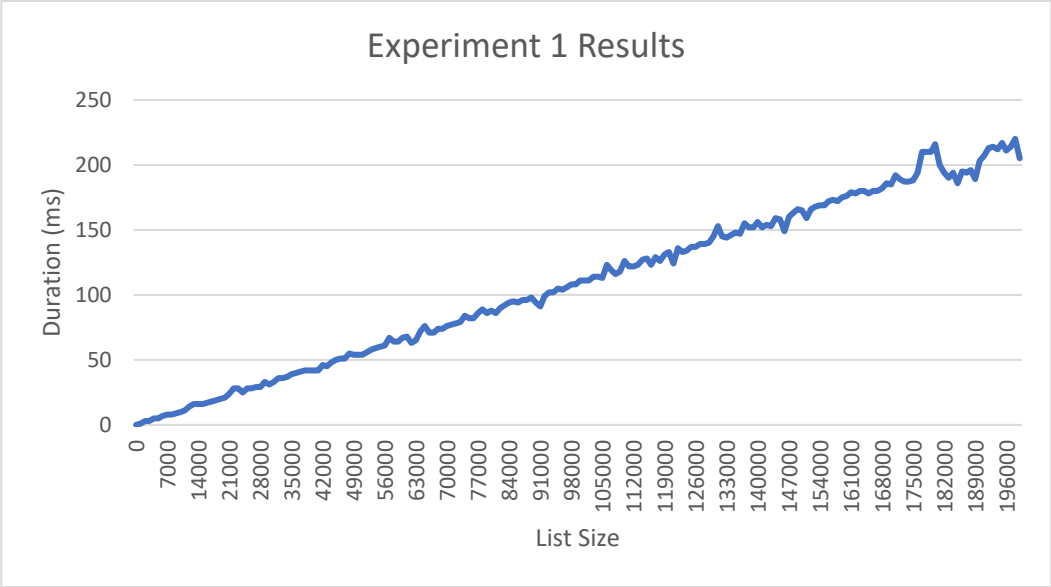
   We originally coded the method with all of the checks just in the percolateDown method and then split it up into helper methods with loops. When we first made the helper methods, we misread the original valid child comparison as having ands (&&) instead of ors (||). We reran our unit tests and discovered that they failed. We then analyzed the new code against the original and looked for our mistake. We then corrected our new logic and re-tested it against our tests and passed them all.

3. Experiment #1

   1. The test in experiment 1 is using our TopKSort method on a number of different lists of increasing size. We are analyzing how long it takes to use our TopKSort method on these lists.

   2. We believe that as the list size increases, the duration that it will take to run will increase at the same rate. This is because we loop over every element in the input list. As the list grows longer, we will have to loop over more elements in order to determine what the top k elements are. We then must sort these top k. So, the time complexity is O(n + logk), or just O(n).

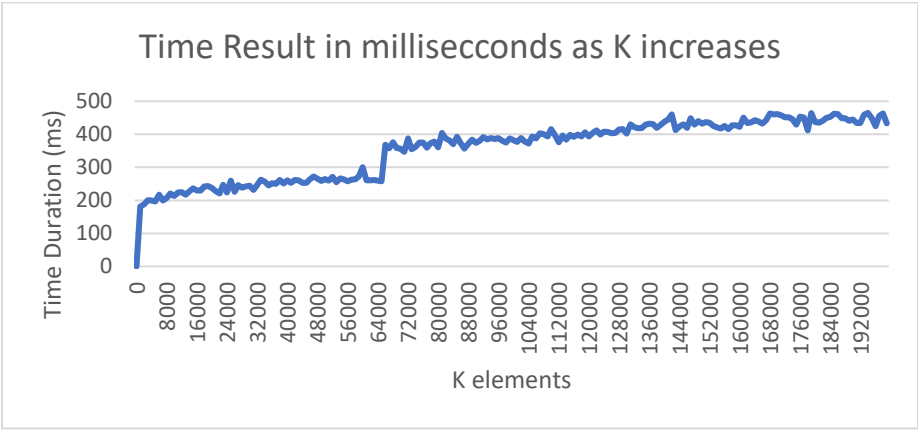
3. Below is the graph of the data for experiment 1:



4. The graph clearly demonstrates that the resulting time was linear. This is because, as we said in our prediction, we are iterating over every element in the list, meaning that it is roughly linear time complexity in order to sort the top k elements.

Experiment #2

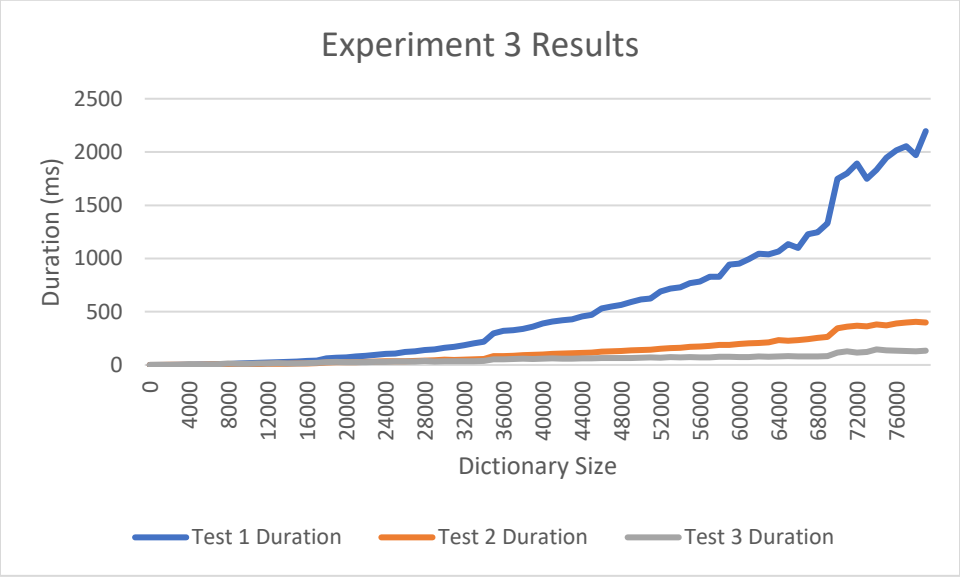
1. This experiment measures the time it takes in milliseconds to sort k elements of a list of increasing size (). As the number of elements to sort increased, so did the time it would take to be sorted. The test took the average time to sort he list. In this experiment, K was the changing variable.
2. When running this experiment, we expect the time it takes to be almost linearly proportional to the number of elements. We expect it to be closer to a linear runtime, than an O(n^2) runtime. We know that heap sort is normally O (n * log(n)) but if we are only sorting K elements, we anticipate the runtime to be O (n * log(k)). We know that resizing will have an impact on the time, but we are not sure at what point this will be greatly impactful (also considering .
3.



4. Our results were interesting to consider. It was relatively linear, but there was a large step up in time from 65,000 sorted elements and 66,000. We believe that this has to do with the resizing functionality of our heap. As mentioned, the result is relatively linear as we predicted. We believe we were correct because when considering O (n * log(k)), the dominating time complexity is O (n), so we did expect the runtime to be close to that.

Experiment #3

1. Experiment 3 consisted of 3 tests. Each test does basically the same thing. They each add elements into our ChainedHashDictionary class, but each test hashes the elements a different way. Test 1 simply returns the sum of the first 4 chars contained in the "String". Test 2 returns the sum of all of the chars in the "String". Test 3 is the most complicated because it loops over every char in the "String", multiplies the previous value by 31 (a prime number) and then adds the current char to the value.
2. We believe that test 1 is going to be the worst. This is because of the simplicity of the hash function will result in many different strings hashing to the same value. Test 2 is going to be much better than test 1 because it sums all of the chars in the String, making it a lot more likely that the hashes will be different. The third test will be very good, this is because it uses primes and multiplies as opposed to just summing the chars.
3. Below is the graph for Experiment 3 Results:



4. Our results match our hypothesis. As stated before, the later tests have more complicated hash functions, which increase the odds of unique hashes. What we didn't expect was that the first test would be that much more different than test 2. We expected the difference between test 2 and test 3 to be farther than the distance than test 1 to test 2. It also appears that very unique hashes don't matter all that much until sizes of around 16,000 at which point the first hash really become much slower than the other two.