Individual Writeup Project 3

Summary:

For this assignment, we need to implement a 4-heap that a heap contains four children. For my understanding, it is to create basic functions “parent” and “child” based on the index and node position. Parent(index) = (index - 1) / NUM_CHILDREN (4) and first child = index * NUM_CHILDREN (4) +1. Since we need to apply the percolate down into ArrayHeap, I created a method that I would find the position of the lowest value of the four (minChild). When the minChild wasn’t null, inserting an element should be checked if the child is smaller than the parent (let the child compare the parent: compareTo() <= 0).

Experiment 1

1. Summary:
    - The first experiment is testing the runtime reflection of topKsort with an increment list size. It repeatedly tests topKsort with large list sizes and inputs, and finally, it is saved as a CSV file.
2. Prediction:
    - The topKsort approaches the O(n*log(k)) asymptotic efficiency. However, I assume it might be a linear graph over the larger list sizes because int K only equals 500 what is not too big to impact the entire runtime.

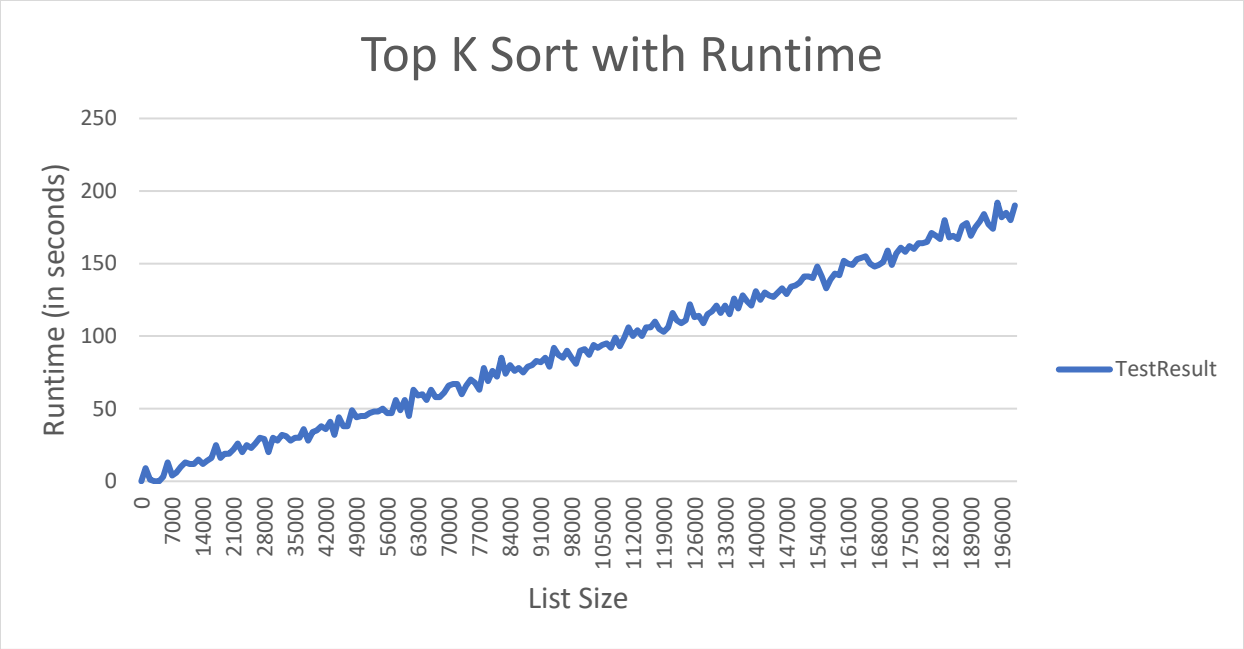

Figure – 1

3. Comparison:

- After running the experiment1, we can find the topKsort seems like a generally linear graph (Figure-1). It matches my hypothesis.

Result:

- The log(k) part cannot found apparently in the graph that is multiplied by because log(k) potentially is dominated by n or the final k value is too small (int k = 500). It is very limited in the change it influences as the list size increases.

Experiment 2

1. Summary:

- The experiment 2 presents how the large value of k contains an increment constant list size in runtime and their outputs.

2. Prediction:

- I assume when the value of k increases, the runtime will increase as well. Besides, there will be a consistent runtime corresponding to a large k value because the top K value iterates through all of the values to get the top K number of values. The change will come

from the log(k) runtime which is part of n*log(k). Therefore, a log(k) in this graph might be covered by n required to iterate through.

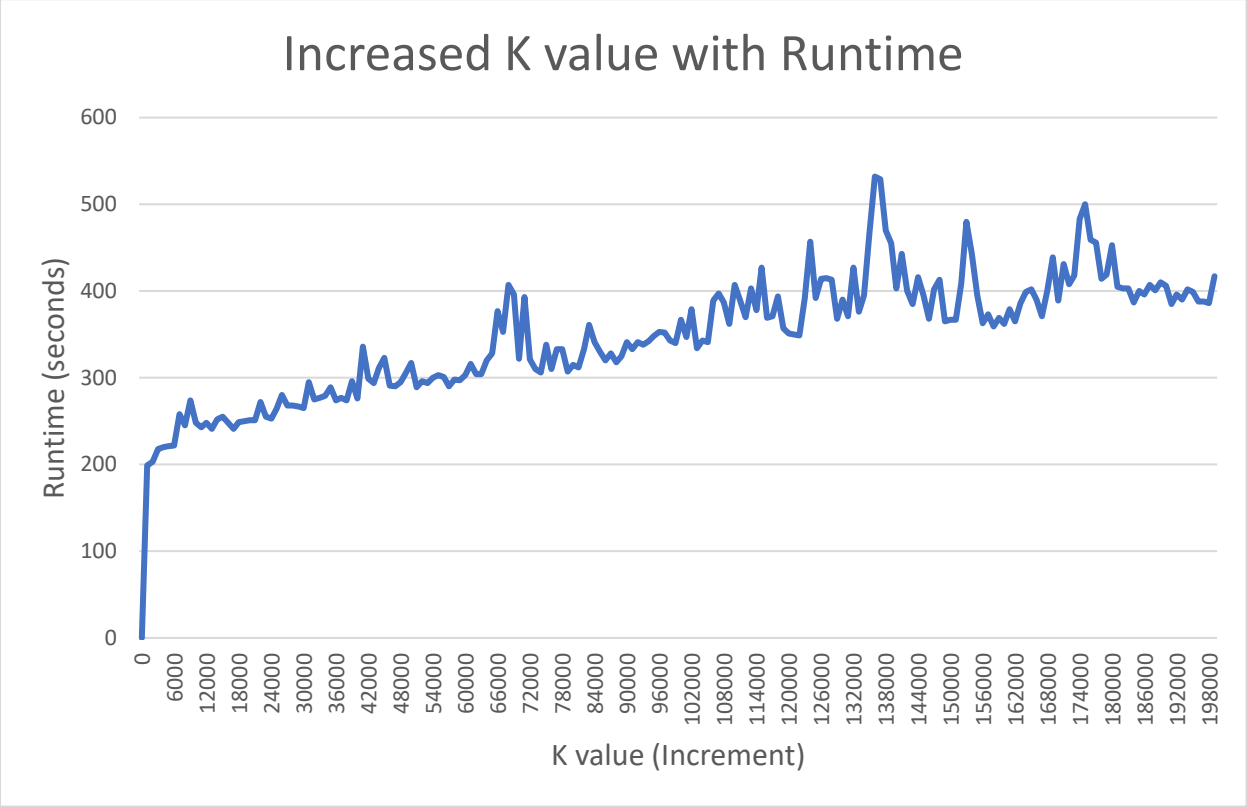

Figure – 2

3. Comparison:

- After running the experiment 2 (Figure-2), It's like my hypothesis that is a very consistent runtime overall, corresponding to the values K and the total number. However, there're some points are up-and-down distinctly because of the memory.

Result:

- The change we see is generally logarithmic due to the k value increasing over time. This represents the n*log(k) nature of topKsort.

Experiment 3

1. Summary:

- The experiment 3 is testing an array of random characters for each test (1,2 and 3) and uses a for-each to put them all into a chained hash dictionary. The difference is in the

FakeString class that each one uses, which provides a differing implementation of hashCode(). FakeString1 adds the first four characters and returns them. The FakeString2 returns the sum of the numeric values of the all the characters and returns that. FakeString3 returns the sum of the characters to the power of 31. After inputting these random characters into the dictionary, a runtime is outputted.

2. Prediction:

- I think the runtime will be inefficient for FakeString1(got more collisions), as the hashCode method always returns the first four characters added together. It will lead to them all being placed in the same slot and the ChainedHashDictionary resizing more often than usual because all values are in the same bucket. It will also be more difficult, and thus slower to find the correct value since all values are in the same bucket. FakeString2 will have a generally more even spread initially. As time progresses, it will have more collisions than FakeString3 because of the unlikely nature of numeric to the 31st power colliding. It's statistically unlikely. While if characters are just added, they are more likely to collide.

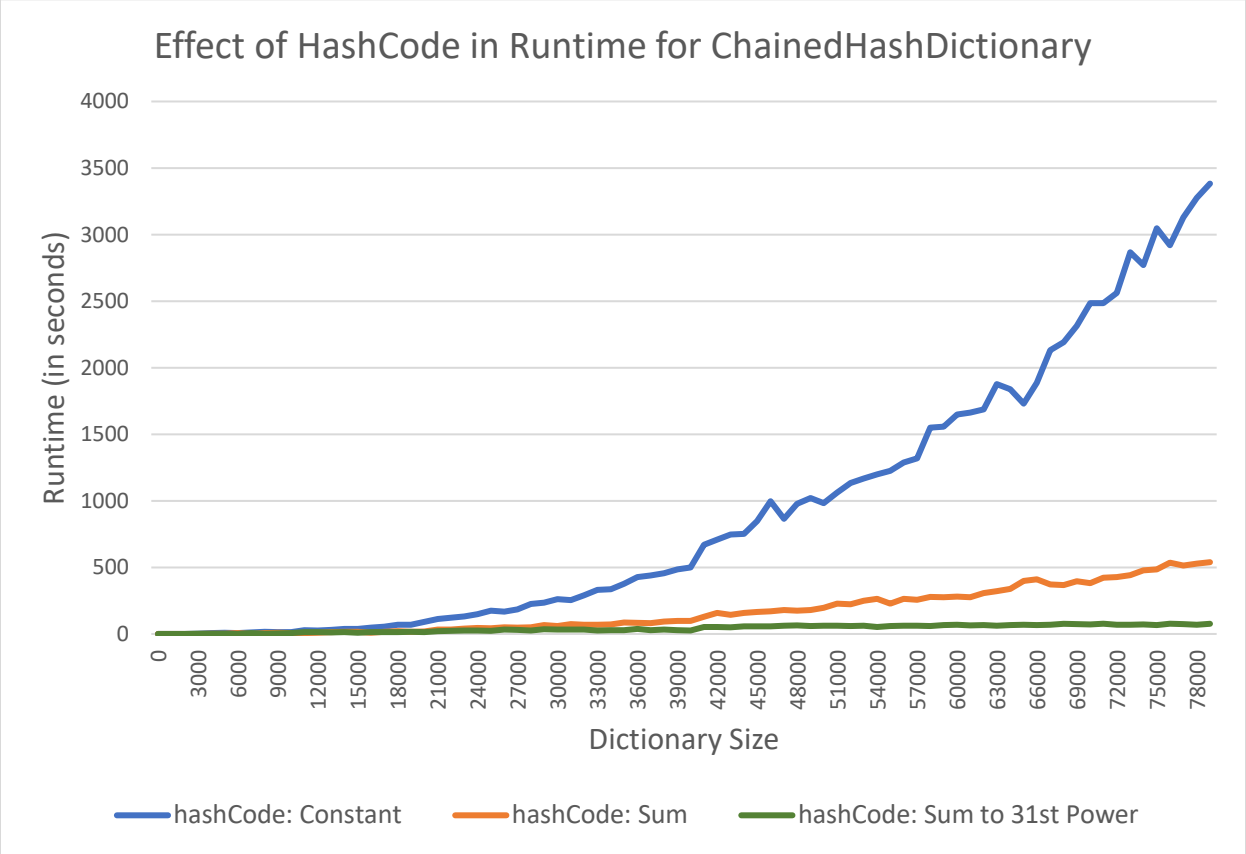

Figure – 3

3. Comparison:

- Test1: It is with the constant hashcode results in a worse runtime. It causes more collisions and the chain keeps having to resize.
- Test 2: When adding value, we check if the key is contained, which calls ArrayDictionary for the bucket and has to iterate over the bucket to figure out if there is the value inside. If that single bucket it massive, it takes much longer to iterate. Test 2 has a generally more even spread initially because of the HashCode to sum.
- Test 3: With Sum and sum to the 31st power, we see a very large decrease in collisions. 31 is a large enough prime number that the number of buckets is unlikely to be divisible by it.

Result:

- Test 1 got the worst runtime result. It is approaching n^2. We see that the sum collides (test 2) more often than the sum to the 31st power as I had predicted. 31 is an odd prime number. Prime numbers are chosen to distribute data among hash buckets best. It increases the runtime because it reduces the collisions.

Questions about the project:

1. What are your name and your partner's name?

My name is Emily Ding. It is a solo project.

2. How was the project? Did you enjoy it?

The project is interesting. The big challenge to me is to think about the 4-heap in the ArrayHeap. It's not a regular heap (2 children). So, first of all, I draw a tree to see the ordering. It helps me to know the relationship between the parent and children. Second, I try to write the pseudocode. This is a good practice for me to think about those data structures and their implementations.

3. Do you have any suggestions for improving the project?

For the future student, there can be more step in the assignment: try to write the tree and formulas. Those help-methods (position of parent, the position of the child, and their relationships) can help students write the algorithms easily.

Questions about your partner

1. How did you feel about pair programming this assignment?

I don't have a partner to do pair programming for this project. I do all the project 3 by myself.

2. How was your partnership?

For some personal reason, my partner and I split the project. I do all the project 3 by myself.