1.

For this assignment, you needed to implement a 4-heap – a heap where each child had four equations. This meant you needed to create two equations: a formula parent(i) to find the parent of some node i, and a formula child(i,j) to find the j-th child of some node i
.
What were those formula?
Parent(i) == (i-1)/4
Child(i, j) == 4*i +j

2.

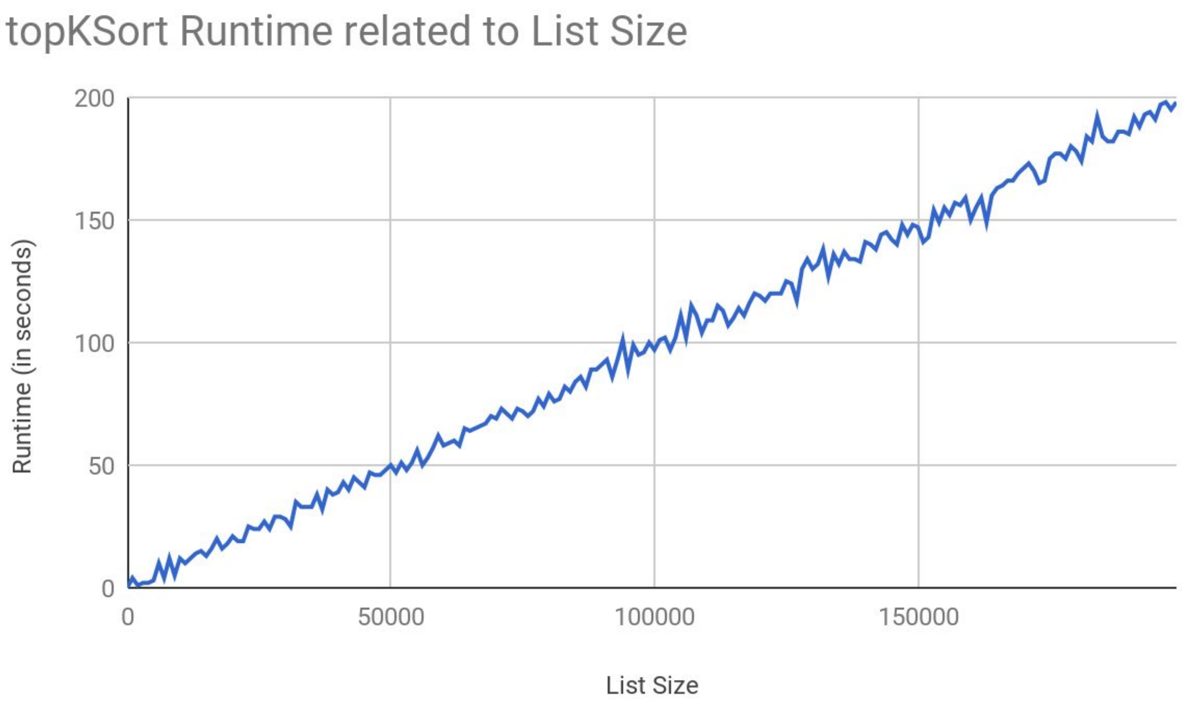
percolateDown algorithm in ArrayHeap, you probably noticed that manually checking each of the four children was tedious and redundant.
How did you refactor this redundancy? Were there any challenges you ran into along the way? If so, how did you handle those challenges?

I made a method in which I would find the child with the lowest value of the four, and then return that corresponding index for the child.

The issue was that I would sometimes check a child layer that went beyond what actually existed. In this case, I had to put in that the code only continued if the lowestChild wasn't null or was less than the length of the heap.

Experiment 1:

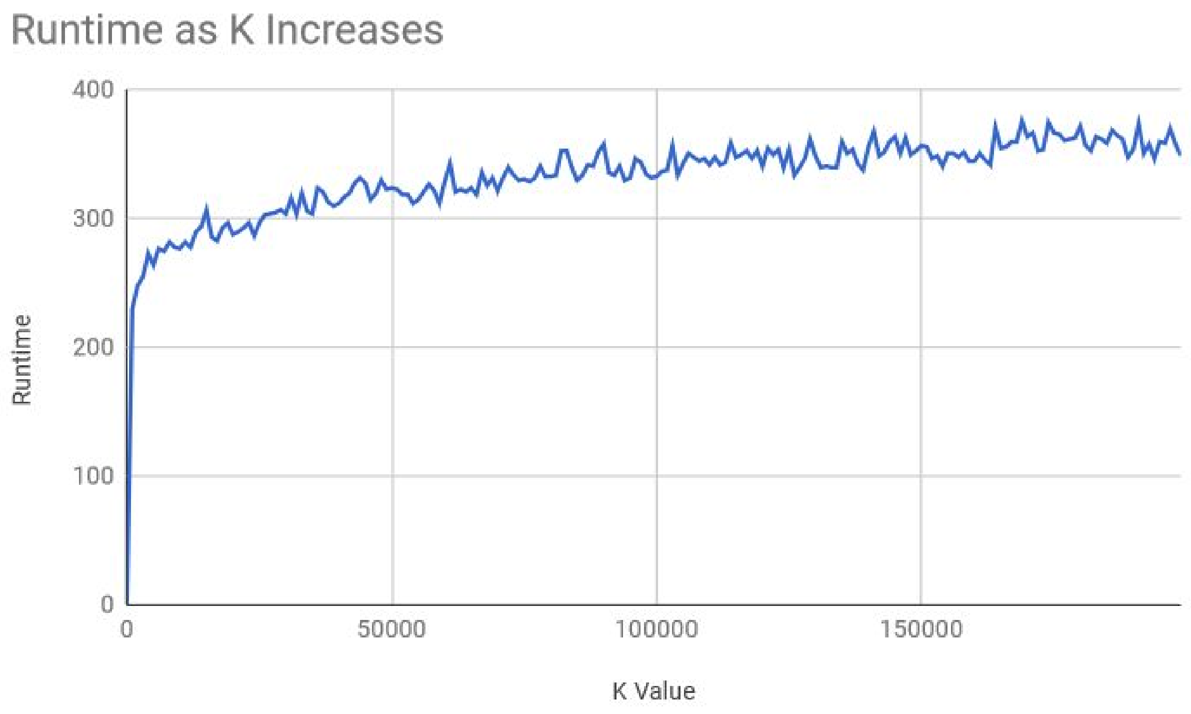
1. This experiment is testing how quickly topKsort functions with an increasing list size. It repeatedly tests topKsort with longer list sizes and inputs that into a CSV file
2. topKsort is supposed to approach n*log(k) asymptotic efficiency. As such, I think it will be above linear time (n) over the larger list sizes.



3.
4. topKSort seems generally very linear. This isn't my interpretation because it is missing any part of the log(k) nature that is multiplied by. This could potentially be because log(k) is dominated by n or that since the k-value is stagnant (at 500), it is very limited in the change it influences as the list size increases.

Experiment 2:

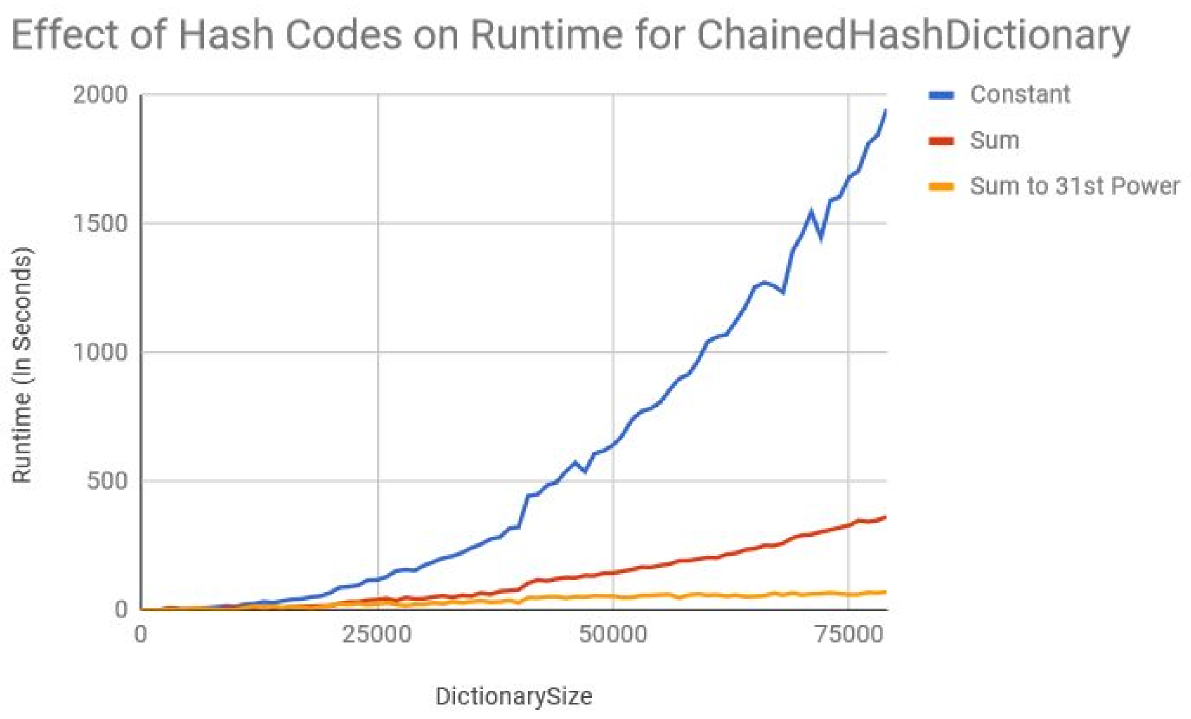
1. This experiment appears to have a constant list size, and tries to vary how large k is, and outputs the corresponding results.
2. I believe that as the k-value increases, the runtime will increase as well. However, since topK iterates through all of the values in the list to get the top K number of values, There will be a consistent time as we saw in the last graph corresponding to list size. The change will come from the log(k) runtime involved in nlog(k). Thus we will see a log(k) nature to this graph that is muffled by the n required to iterate through.



3.
4. I believe that the results match my hypothesis. I interpreted that, with a large list size, there would be a very consistent runtime overall. The change we see is generally logarithmic in nature due to the k value increasing over time. This represents the nlog(k) nature of topKsort.

Experiment 3:

1. Test 1,2, and 3 generate an array of random characters 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 differing implementation of hashCodes. FakeString1 adds the first 4 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. I believe that the Runtime will be poor for FakeString1 the most, as the hashcode will always return the first 4 characters added together. This will lead to them all being placed in the same slot and the ChainedHashDictionary resizing more often than normal 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. Yet as time progresses, it will have more collisions than FakeString3. This is because of the unlikely nature of numerics to the 31st power colliding. It’s statistically unlikely. While if characters are just added, they are more likely to collide.



3.

4. Test1 with the constant hashcode results in a worse runtime. However, I didn't expect it to be approaching n^2. This could potentially be because the chain keeps having to resize. Also, when adding a 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 actually is the value inside. If that single bucket it massive, it takes much longer to iterate. With Sum and sum to the 31st power, we see a very large decrease in collisions. Yet we also see that the sum collides more often than the sum to the 31st power as I had predicted.