Jackson Campbell
Byron Hoy

JacksonHashMap Project: Documentation

During class, we were tasked with creating our own, small scale version of Java's HashMap class, and to at least give it a hashing method, a put method, a resize method, and lastly a containsKey method to catch collisions. Within JacksonHashMap, the following methods are contained and work as such:

simpleHash(E data):

- Hashes based off of the character length of the data input, as was instructed originally in class. Takes in the data to be input, appends it to an empty String variable, and returns the string's length for the key.

put(E data, Integer val):

- Takes the generic data to be input into the HashMap based off of the parameters set in the class initialization as the key, and then takes an integer value as the value to pair it with.
- Uses a system of two LinkedList arrays to handle the key-value placement, and checks for sizing and will dynamically resize if the capacity of lists exceeds the set size.
- Will check for a collision using containsKey(data) before placing the key and value in the map, and if a collision occurs, will replace the key with the new value at the current hashing index.
- Otherwise, simply places the key and value onto the map based on the returned hashing index value.

resize():

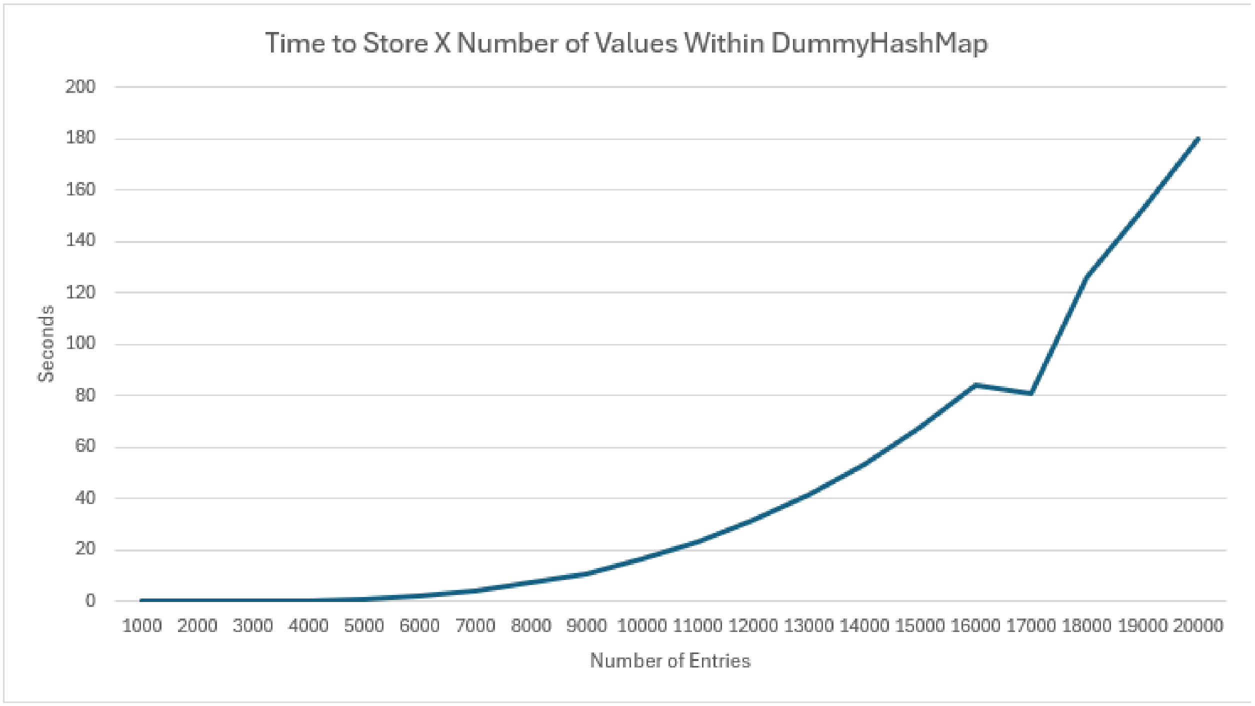
- Dynamically resizes the arrays if the capacity is reached. Within put, the size is checked during each placement, and will use this function if so.
- Takes the array's original set size, multiplies it by 2, and carries the already set values over to the new arrays.
- After resizing to temp arrays, it sets the original arrays to equal the temporary ones, thus becoming a permanent resize.

containsKey(E existingKey):

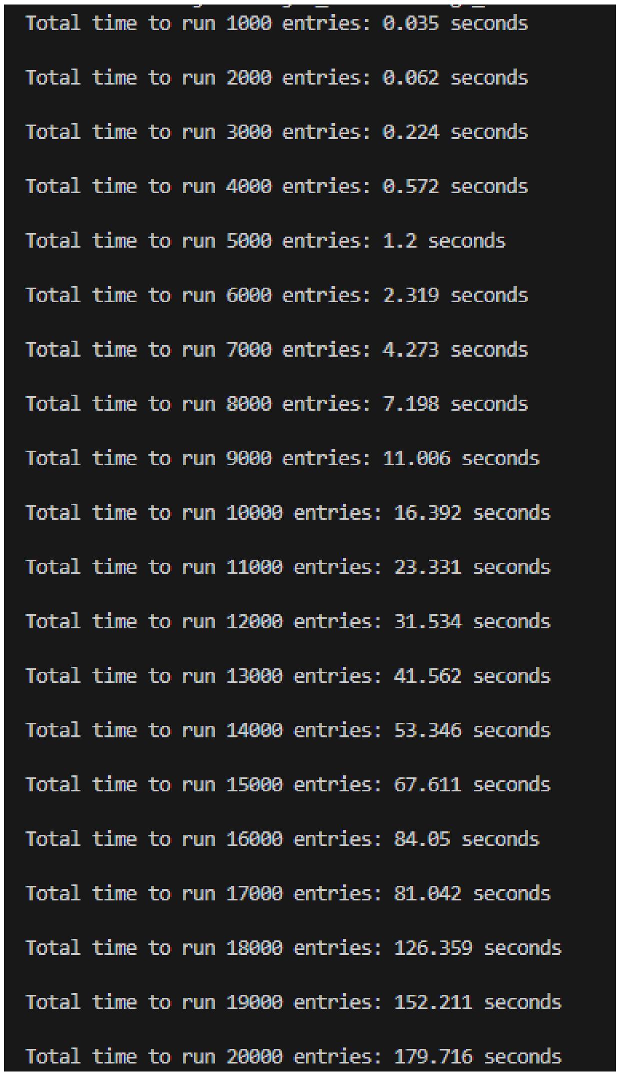
- Loops through the HashMap to see if a given key already exists based on the returned hashing index.
- If so, returns a true value. Otherwise, returns false.

With these methods, testing can now be done to see how fast the HashMap implementation runs when given a certain amount of inputs to process. For the data set to process, a String array of different fruits were created, and a random value was added to the end of its string when added in order to create unique values to place onto the map, and the value was randomized alongside the String. For example, an entry to the list could possibly be {Apple 5, 200} or {Cherry 1423, 298}.

Below is a graph testing with entry values ranging from 1,000 to 20,000, and displays the amount of time in seconds that it takes to map all values:



When viewing the graph, it can be seen that there is an exponential curve overall to the graph, albeit with a value of $m < 1$ as a coefficient it seems. However, the outlier in this data set is the plot from 16,000 to 17,000 entries, as the seconds decrease when increasing, even if not by much. During multiple tests, the same occurrence was reported at the same value area and is something that should have been explored further. Below is the output of the tester, with the given runtime for each test:



```
Total time to run 1000 entries: 0.035 seconds

Total time to run 2000 entries: 0.062 seconds

Total time to run 3000 entries: 0.224 seconds

Total time to run 4000 entries: 0.572 seconds

Total time to run 5000 entries: 1.2 seconds

Total time to run 6000 entries: 2.319 seconds

Total time to run 7000 entries: 4.273 seconds

Total time to run 8000 entries: 7.198 seconds

Total time to run 9000 entries: 11.006 seconds

Total time to run 10000 entries: 16.392 seconds

Total time to run 11000 entries: 23.331 seconds

Total time to run 12000 entries: 31.534 seconds

Total time to run 13000 entries: 41.562 seconds

Total time to run 14000 entries: 53.346 seconds

Total time to run 15000 entries: 67.611 seconds

Total time to run 16000 entries: 84.05 seconds

Total time to run 17000 entries: 81.042 seconds

Total time to run 18000 entries: 126.359 seconds

Total time to run 19000 entries: 152.211 seconds

Total time to run 20000 entries: 179.716 seconds
```

Here, it can be seen again that the jump from 16,000 entries to 17,000 entries decreases in time instead of steadily increasing as it should.

Overall, this program runs relatively slowly, and although it works properly, is incomplete in terms of speed and is not a viable option when mapping values quite yet. The speed of sorting and searching needs to be improved upon much more before the class itself should be used for large data sets. Small data sets are okay to use this class, as even at 1000 entries, the program runs in less than one tenth of a second still.