Hash Map User Manual

By: Jacob Attia

1. What is this?

This project shows how a very small hash map works. A hash map is a box that stores many items and lets you check quickly if an item is inside.

- We use a very simple hash rule: the length of the text (String.length()).
- Items that hash to the same value sit in a short list.
- The map grows when it gets too full.

2. What files are here?

File Name	Job
JacobHashMap.java	The main code for the map
JacobHashMapTest.java	A short test that adds a few words and prints results
HashMapExperiment.java	Makes lots of random words, times how long inserts take, and saves numbers to results.csv

3. How does each file work?

JacobHashMap.java

- Holds an array of LinkedLists. Each spot is called a bucket.
- dumbHash: length % bucketCount chooses the bucket.
- add(item): puts the item in the right bucket. If the map is 75 % full, it doubles the bucket count.
- contains(item): checks if the item is in its bucket.
- resize(newSize): makes a bigger (or smaller) bucket array and moves everything.

JacobHashMapTest.java

- Makes a map.
- Adds eight fruit names.

- Prints: does it have "banana"? does it have "kiwi"?
- Shows the bucket count before and after a manual resize to 32.

HashMapExperiment.java

- Builds maps of different sizes (10 k, 20 k, … 160 k words).
- Each word is 5–15 random letters.
- Tracks how long the insert loop takes.
- Writes one line per run to results.csv: size, timeInNanoseconds, bucketCount

4. Running the code and what you will see

4.1 Compile everything

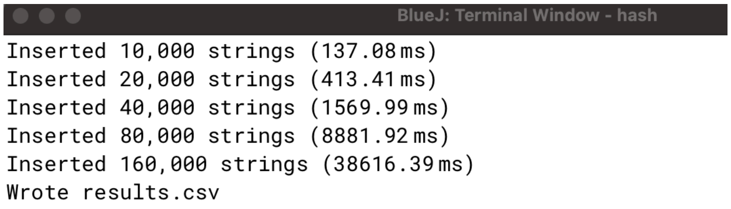
4.2 Quick check of basic functions

- java JacobHashMapTest
    - You should see:
      Contains 'banana'? true
      Contains 'kiwi'?   false
      Initial capacity: 16
      Capacity after manual resize: 32

- banana was added so it is found.
- kiwi was not added so it is not found.
- The map doubled its bucket array from 8 to 16 when it got 75 % full, then you resized it to 32 by hand.

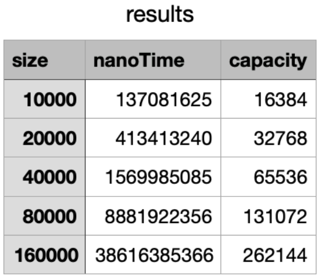
4.3 Performance experiment

- java HashMapExperiment
    - This program makes new maps of different sizes, fills them with random words, times each fill, and saves the numbers to results.csv. Console example:



```
Inserted 10,000 strings (137.08 ms)
Inserted 20,000 strings (413.41 ms)
Inserted 40,000 strings (1569.99 ms)
Inserted 80,000 strings (8881.92 ms)
Inserted 160,000 strings (38616.39 ms)
Wrote results.csv
```

    -
    - Results.csv



results

size	nanoTime	capacity
10000	137081625	16384
20000	413413240	32768
40000	1569985085	65536
80000	8881922356	131072
160000	38616385366	262144

-

5. Conclusion

This mini-project shows how a hash map works at its core:

- Buckets hold items that hash to the same spot.
- A hash rule turns a key into a bucket number. We used key length so you can see collisions easily.
- When the map gets too full, it grows and every key is placed into a new bucket.

By running the simple test you see that look-ups are quick. By running the experiment you see that insert time grows roughly in a straight line and that bucket size doubles at certain points.

Use this code as a starting point to try better hash rules, add a remove() method, or compare with Java’s built-in HashSet. Each change will teach you a bit more about how real hash maps balance speed and memory.