

Empirical Analysis of Linear Probing with Dynamic Resizing

Problem statement :

To study linear probing and measure its performance under different conditions and determine the breakeven point between efficient and degraded performance. Our aim is to create a combined algorithm with resize logic and measure the performance of the new algorithm under different array sizes and load factors.

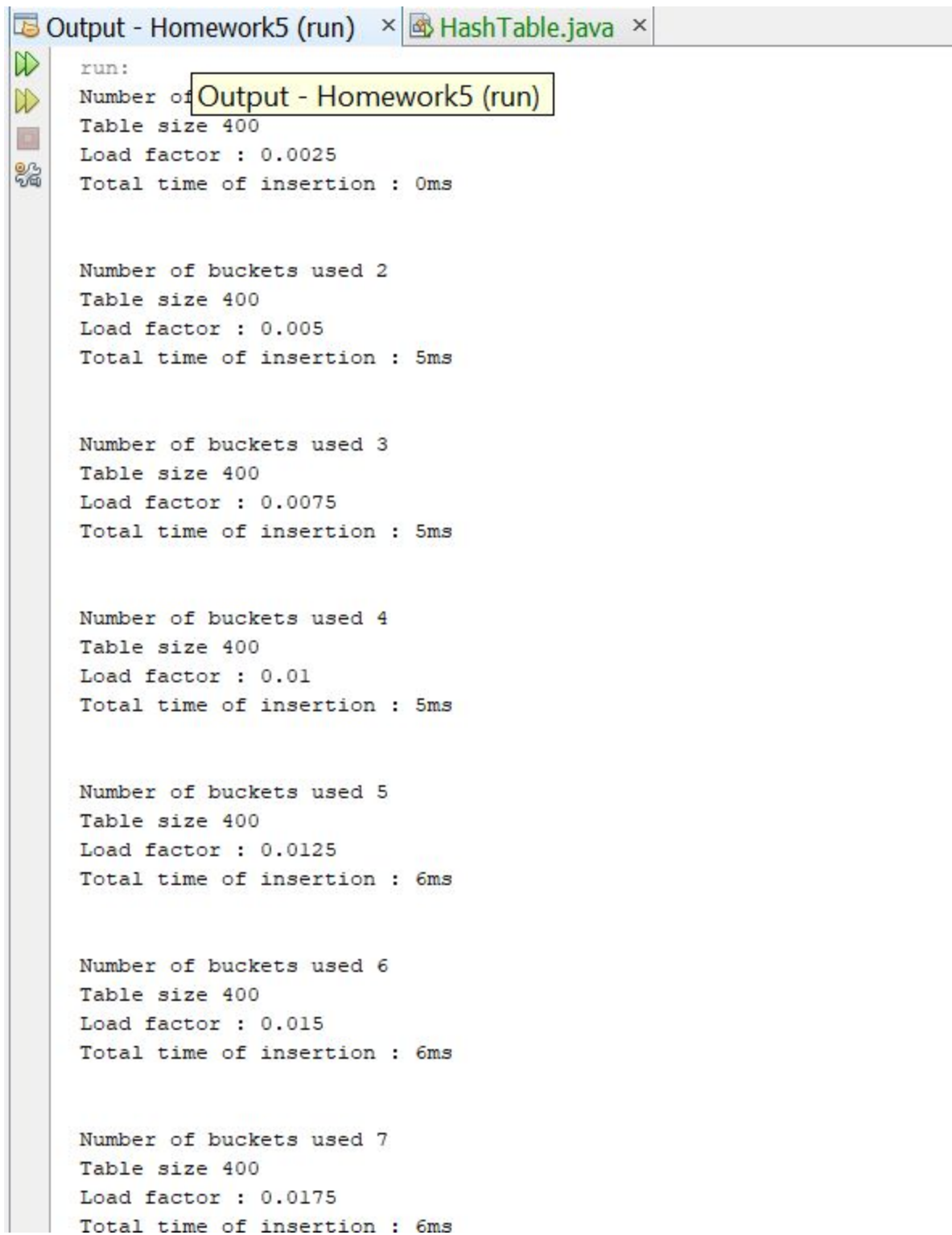
- For linear probing, the formula to calculate the associated index for a key is calculated as $hi(X) = (\text{Hash}(X) + i) \bmod \text{tablesize}$, where i is a whole number.
- For any load factor less than 1, linear probing finds an empty slot to store the key value pair.
- When collision occurs in linear probing, the next immediate available index is assigned to the incoming data.
- To avoid the clusters, we will rehash the hashtable and double its capacity in order to minimize the load and observe the performance of the insert operation

Observations:

- After a point, as the load factor approaches the value 1 (~ 1) the time taken to insert an element in a hashtable increases significantly since collision happens.

- As we increase the capacity of the hashtable by double, the insertion time decreases as there are no clusters/collisions and its easier to find a dedicated spot for the incoming key value pair.

Screenshot of outputs:



```
run:
Number of buckets used 2
Table size 400
Load factor : 0.0025
Total time of insertion : 0ms

Number of buckets used 2
Table size 400
Load factor : 0.005
Total time of insertion : 5ms

Number of buckets used 3
Table size 400
Load factor : 0.0075
Total time of insertion : 5ms

Number of buckets used 4
Table size 400
Load factor : 0.01
Total time of insertion : 5ms

Number of buckets used 5
Table size 400
Load factor : 0.0125
Total time of insertion : 6ms

Number of buckets used 6
Table size 400
Load factor : 0.015
Total time of insertion : 6ms

Number of buckets used 7
Table size 400
Load factor : 0.0175
Total time of insertion : 6ms
```



```
Output - Homework5 (run) × HashTable.java ×
▶▶▶ Table size 400
▶▶▶ Load factor : 0.4925
▶▶▶ Total time of insertion : 140ms

▶▶▶ Number of buckets used 198
▶▶▶ Table size 400
▶▶▶ Load factor : 0.495
▶▶▶ Total time of insertion : 140ms

▶▶▶ Number of buckets used 199
▶▶▶ Table size 400
▶▶▶ Load factor : 0.4975
▶▶▶ Total time of insertion : 140ms

▶▶▶ Number of buckets used 200
▶▶▶ Table size 400
▶▶▶ Load factor : 0.5
▶▶▶ Total time of insertion : 140ms

▶▶▶ Doubling the hashtable capacity :
▶▶▶ Number of buckets used 201
▶▶▶ Table size 800
▶▶▶ Load factor : 0.25125
▶▶▶ Total time of insertion : 140ms

▶▶▶ Number of buckets used 202
▶▶▶ Table size 800
▶▶▶ Load factor : 0.2525
▶▶▶ Total time of insertion : 140ms

▶▶▶ Number of buckets used 203
▶▶▶ Table size 800
▶▶▶ Load factor : 0.25375
▶▶▶ Total time of insertion : 140ms
```

As we can see from the above image, when we increase the capacity of the hashtable, the load factor decreases and clusters are reduced which makes the insertions without collisions in less time.