HashTable_reemk_kestenberg1.md

Data Structures Project 2 - HashTable

- Maayan Kestenberg - *kestenberg1* [213056674]
- Re'em Kishnevsky - *reemk* [213057094]

Experiment 3

Section A

- Size of Q1 is 3286
- Size of Q2 is 6571

Section B

- Exceptions in `QPHashTable`: 83
- Exceptions in `AQPHashTable`: 0

The set of indices in the probe sequence of an element in `QPHashTable` is Q1 (from section A)with some constant added (the element's hash); therefore, in QP we check |Q1|=3286 indices (~halfthe table)when searching for a vacant place to insert the element in. Consequently, as the table gets filled,there is a non-negligible chance that the probe sequence will contain only occupied indices (up toa probability close to 1/2 when there is one free slot left), resulting in a `TableIsFullError`.

On the other hand, the set of indices in `AQPHashTable`'s probe sequences is Q2 (from section A) again with the element's hash added, which means that every such probe sequence contains 6571 indices - the whole table. Therefore, we only get `TableIsFull` errors when the whole table is full, and in our case, this never happens because we only insert m elements (the size of the table) and not a single element extra. That is why we got 0 errors.

Section C

Every prime number p>2 has (p+1)/2 quadratic residues mod p (including 0). The set Q1 is exactly the set of quadratic residues mod m=6751, which is prime. It is obviously a set of quadratic residues per definition because it is defined as the remainder of an integer i squared mod p. It contains ALL the quadratic residues because for every integer x>m, there exists some i<m for which x is equivalent to i mod m, which also means that x squared is equivalent to i squared mod m - and i squared mod m is a member of Q1. This is why Q1 has exactly (m+1)/2=(6751+1)/2=3286 elements, causing the probe sequence in QP to only contain half (rounded up) of the table's indices.For m=2, every integer is a quadratic residue mod m, so the probe sequence will contain all 2 indices. This is the only prime for which the phenomenon (the probe sequence only containing half the indices) doesn't occur, since for a any prime m>2 we have seen that it does occur.

Experiment 4

Section A

Class	Running Time
`LPHashTable`	0.751 sec
`QPHashTable`	0.739 sec
`AQPHashTable`	1.069 sec
`DoubleHashTable`	1.356 sec

In this case, the simpler the implementation (computationally), the better the performance. This is because collisions aren't very likely when inserting only half of the full capacity, which makes the simpler, easier to compute implementations better because the trade-off for less collisions isn't worth it. LPHashTable is the fastest because it's probe sequence indices are the easiest to compute, and it has minimal cache misses. QPHashTable follows by being the next most simple to compute and AQPHashTable comes after, possibly for being a little more complicated than QPHashTable. DoubleHashTable is the slowest because computing the two hash functions is the most computationally expensive amongst the implementations.

Section B

Class	Running Time
`LPHashTable`	11.123 sec
`AQPHashTable`	5.948 sec
`DoubleHashTable`	7.727 sec

We don't run `QPHashTable` because this time we add a number of elements that is bigger than half the table size, and as we saw in experiment 3, this can cause `QPHashTable` to throw `TableIsFull` exceptions - which, as instructed, are not supposed to be thrown in this experiment. The long running time of `LPHashTable` reveals it's limitation when the table becomes close to full. Since the search sequence of an element in linear probing is consecutive, clusters are formed in the table. As we insert more elements, those clusters get longer. While the table is getting close to being full (like in this experiment), different clusters begin to merge, forming even larger clusters and further lengthening the running time of additional inserts. This problem of merging clusters was not as apparent in the previous results because we inserted much less elements, so the length of clusters and their chances of merging together were lower.

`DoubleHashTable` outperforms `LPHashTable` because of it's use of a secondary hash function to determine the step between following probe indices; this way, `DoubleHashTable` minimizes collisions because even if two elements are hashed to the same cell, they are likely to have different search sequences (the step hash and base hash are independent).

Finally, we observe that `AQPHashTable` ran the fastest. that can be explained: `DoubleHashTable` uses a hash function to calculate the step of the search sequence, while `AQPHashTable` uses a simple calculation. The modulo operations of `DoubleHashTable`'s step hash function are more expensive than `AQPHashTable`'s calculation, so when using `DoubleHashTable` we trade the efficiency of our search sequence calculation for a more sophisticated search sequence that results in fewer collisions. This trade does not benefit us in the case that we fill up a large portion of the table (the above results are when filling 95% of the table) - because then, collisions are much more frequent, so we will be calculating a larger portion of the search sequence. Therefore it becomes more profitable to have an efficient way of calculating the search sequence (at the cost of more collisions) than to have a search sequence that is heavy to compute but avoids more collisions. On the other hand, `AQPHashTable` does not create large clusters and this is why it manages to perform well, unlike `LPHashTable`. `AQPHashTable` takes the middle ground between computational simplicity and collision minimization, which works best in this case.

Experiment 5

Iterations	Running Time
First 3 iterations	9.521 sec
Last 3 iterations	33.813 sec

The difference in running time stems from the fact that as we perform more iterations, the hash table is filled with more `deleted` marks due the elements we delete at each iteration. In the last 3 iterations, each iteration starts with significantly more cells in the hash table that are marked `deleted` than in the first 3 iterations. The more `deleted` cells in the table >>> the more cells that are expected to be scanned when inserting a new element >>> the more time required for insert operations, hence the longer running time. Another intuitive way of thinking about it is that each `deleted` mark is treated just like an ordinary taken cell when scanning the table to find a place to insert an element. And we already know from experiment 4 that the fuller the hash table, the longer it takes for each insert.