Hash Tables

Submitted by:

Name: Oded Kesler
ID: 200973212
Username: odedkesler1

- Singular submission authorized by Dr. Amir Rubinstein.

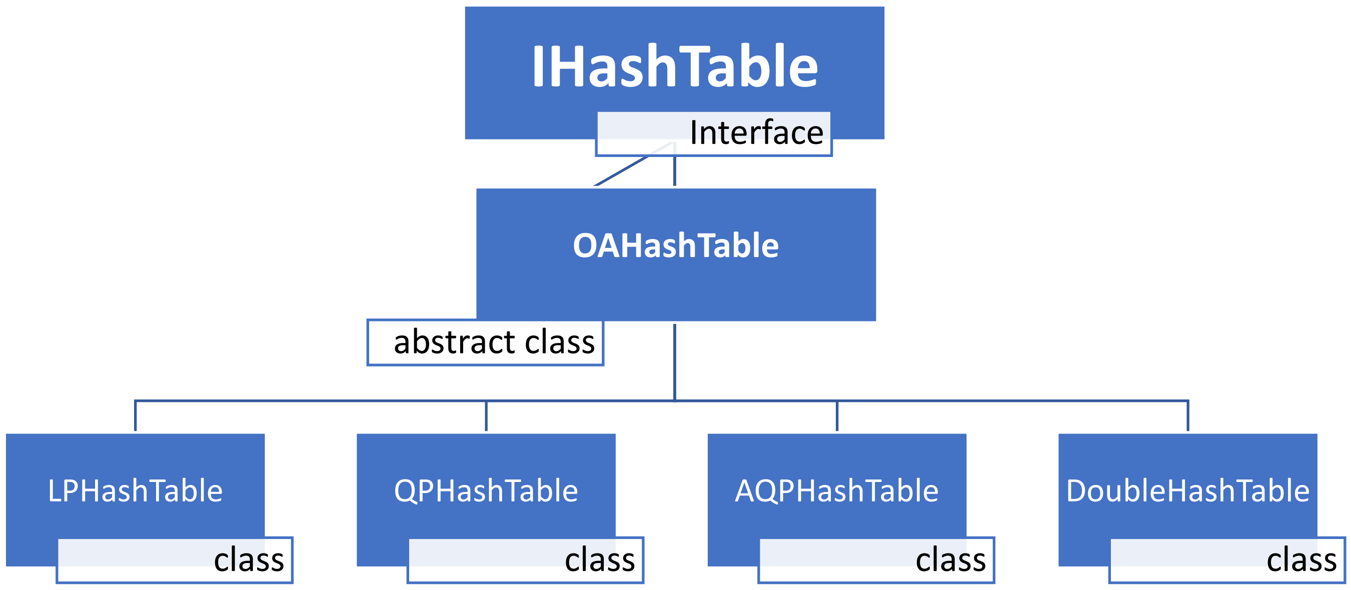
Introduction

In the following project, we’ve implemented different open addressing hash tables using four different probing techniques.
The abstract class **OAHashTable** implements the **IHashTable** interface:

- Find(long key)
- Delete(long key)
- Insert(HashTableElement hte)

In addition, the abstract class has an abstract function - Hash(long x, int i),
that each inheriting class implement, by applying a different probing technique.

Inheritance Scheme



IHashTable Implementation

- **Find(long key)** – Iterates the hash table, checking each cell if it contains the element that holds the input key. The order of iteration is implemented in each of the inheriting classes that applies a different probing technique (This applies for all other functions of the interface as well).
  While iterating thru the table:
  1. If we encounter a null cell, we return null since no such element exists in the table.
  2. If we encounter a cell that holds an element with the input value as its key, we return that element.
  3. Finally, we return null again if we iterated thru the whole table without finding the desired element.

- **Delete(long key) –** Iterates the hash table in the same manner as the Find() function.
  The main difference here lies in the returned values of the function:
  1. If we haven't found the element, either by encountering a null cell or by iterating the entire table, we return a **KeyDoesntExistException(key)**.
  2. If we do find the desired element, we mark it as deleted by inserting a place holder element to that cell, with a key and value of -1.

- **Insert(HashTableElement hte) –** In order to insert a new element to the hash table, we want to find a vacant cell. The vacant cell can either be the first cell that we encounter while iterating that is marked as deleted, or a null cell – whatever comes first.
  The breaking condition for stopping the iteration is only when we encounter a null cell,
  as we insert the input element to that index, or to the index of the first marked deleted cell if it exists.
  In each iteration we also check the key of the element at the probed index, to make sure we're not inserting a duplicate entry to the table – If such case occurs we return a **KeyAlreadyExistsException()**.
  A third case can occur while iterating the entire table without finding a vacant cell for insertion, and in that case we return **TableIsFullException(hte)**.

Hashing

When initialized, each inheriting class calls its abstract's class constructor,
followed by initializing its unique hash parameters by calling the **ModHash.GetFunc()** function.
Upon hashing, each class calculates the hash value for the given key, as the initial probed index, followed by a unique probing sequence (if needed) as such:

- **LPHashTable -** linearly iterates the table starting at the initial probed index.
- **QPHashTable -** iterates the table in a quadratic manner as such: **initial probed index + i*i**
- **AQPHashTable –** iterates the table in an alternating quadratic manner, to apply all step sizes: quadratic residues of the table size, and its non-quadratic residues.
  We optimized the way that we calculate the alternating sign of -1, by adding another field to this class name **direction –** For every iteration we switch it's sign by multiplying it by -1.
- **DoubleHashTable –** Determines its next step by calculating another hash function from the same universal family of functions that we used for the initial hash. It does that by holding another field that calls the **ModHash.GetFunc()** function but **passing it m-1 as an input instead of m.**
  The returned value of this function is then incremented by 1 to ensures that the next step size is:
  $1 \leq step\ size < m.$

Question no 3

3.1

For q = 6571:
Q1 = $|\{i^2 mod\ q|\ 0 \leq i < q\}| = 3286$
Q2 = $\left|\{(-1)^i * i^2 mod\ q|\ 0 \leq i < q\}\right| = 6571$

3.2

For **Quadratic Probing,**
on average, 70**%** of the times, the insertion sequence throws a “TableIsFullException” at the ~6570 insertion.
i.e., for most cases the table behaves as its full even though it has several vacant indexes left.

For **Alternating Quartic Probing,**
the series of insertions always work.
i.e., using this probing method we can completely fill the table.

As we’ve seen in section 3.1: Q1 = $|\{i^2 mod\ q|\ 0 \leq i < q\}| = 3286$
The size of Q1, states the number of possible steps in a probing sequence for a given hash.
For a hash table of size **q**,
using this implementation of quadratic probing only enables $\frac{q-1}{2}$ unique steps , excluding out the rest of the indexes from the probing sequence.
As the hash table gets full the chance of missing a vacant cell in the probing sequence grows.
Based on the results of this experiment, it happens in 70% of the times towards the final insertions (the load factor α gets closer to 1).

On the other hand,
the implementation of alternating quadratic probing,
results in **q** possible steps for a probing sequence (the size of Q2).
Meaning that for any hash, its probing sequence will cover all indexes of the hash table.
This in turn ensures that “TableIsFullException” will occur only when the table is truly full.

3.3

If the greatest common divisor of $(i, m) = 1$,
i is a quadratic residue modulo **m** if there is a solution to $x^2 \equiv i(mod\ m)$.
For an odd prime number there are $\frac{m-1}{2}$ quadratic residues, and this is what we’ve observed when we tried filling the hash table using **quadratic probing**.
Negating the quadratic residue of an odd prime that is congruent to 3 modulo 4, results in a non-quadratic residue.
This specific implementation of quadratic probing yields all quadratic residues of the size of the hash table, which is not a complete premutation of $[\boldsymbol{m}]$.
The alternating quadratic probing sequence, combined with a table size **m** which is $m = 6571 \equiv 3(mod\ 4)$, results in a complete permutation of $[\boldsymbol{m}]$.

Question no 4

4.1

$$\boldsymbol{p} = 1{,}000{,}000{,}007 \ | \, \boldsymbol{m} = 10{,}000{,}019 \ | \, \boldsymbol{n} = \left\lfloor \frac{m}{2} \right\rfloor$$

Class	**Running Time**
LPHashTable	$911 \times 10^{-3} sec$
QPHashTable	$821 \times 10^{-3} sec$
AQPHashTable	$815 \times 10^{-3} sec$
DoubleHashTable	$1030 \times 10^{-3} sec$

- $Average\ Running\ Time = 894 \times 10^{-3} sec$

In this segment we insert $\left\lfloor \frac{m}{2} \right\rfloor$ elements into a hash table of size **m.**
This means that the load factor $\alpha < 0.6$, and as we've seen in class this ensures that each insertion, for all probing schemes, is done in a constant time.
Therefore, the difference in running time between the classes is relatively small.

When compared to the average running time of all classes:
- **LPHashTable** has more or less the average running time.
- **AQPHashTable** & **QPHashTable** have slightly lower than average runtime.
  It runs faster than the other two classes since it has no linear clusters, and the probing computation is fast.
- **DoubleHashTable** has the highest running time, and in this case, it seems to be the cost of the computation of the additional function that determines the probing sequence.

4.2

$$\boldsymbol{p} = 1{,}000{,}000{,}007 \ | \, \boldsymbol{m} = 10{,}000{,}019 \ | \, \boldsymbol{n} = \left\lfloor \frac{19m}{20} \right\rfloor$$

Class	**Running Time**
LPHashTable	$8.3\ sec$
AQPHashTable	$3.5\ sec$
DoubleHashTable	$5.6\ sec$

- $Average\ Running\ Time = 5.8\ sec$

In this experiment we insert elements into the hash table until the load factor is $\alpha = 0.95$.
Reaching such a high load factor stresses out the limitations and/or advantages of the different probing schemes.
Therefore, in this experiment, we did not test the performance of **QPHashTable** since we've already shown how secondary clusters accumulate in this implementation. Which in turn can prevent us from hashing elements while there are still vacant cells in the table.

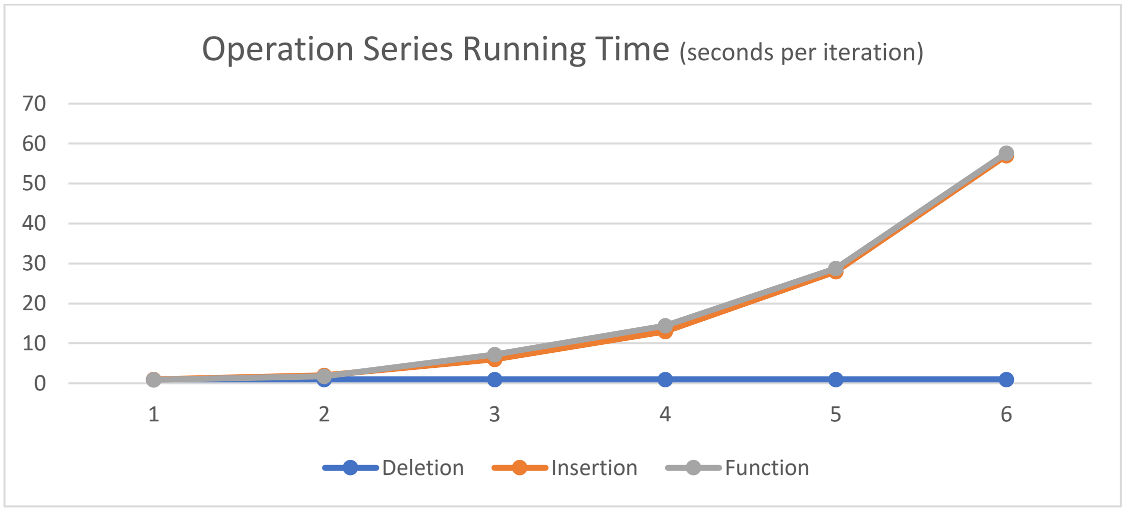
Based on these results, **LPHashTable** is performing poorly when compared to other classes.
It has 40% higher than average runtime which is probably caused due to primary/linear clustering.
The other two classes, **AQPHashTable** & **DoubleHashTable**, have a far better running time under these conditions.
And again, DoubleHashTable is slower than AQPHashTable due to the computation of the additional function that determines the probing sequence.

Question no 5

Iterations	Running Time Average
First 3 iterations	$4\ sec$
Last 3 iterations	$33.67\ sec$

- Results shown are the most consistent running time averages for this experiment, after running it 20 times.

The running time average for the last 3 iterations is longer by 8.5 than the first 3 iterations.
In order to fully understand the difference in the running time averages we kept track of the time it takes for each series of insertions and deletions:



As deletions takes a constant time of 1 second for each iteration, the time that takes to insert all elements grows significantly. Each series of insertions takes roughly twice as more time than the last iteration.
The running time for each iteration complies with the following function (grey trendline in graph):

$$Insertions(i) = 0.9 * 2^{i-1}$$

Hence, it seems that the solution that we've chose to handle with deleting elements from the table, without interfering in the probing sequences, **is not optimal**.
In this experiment the **growth of insertion time is exponential to the number of iterations**.

As we probe the table to insert a new element, the probing sequence will only end once we either found an empty cell, or we've probed the entire table (regardless to if we found a deleted cell or not).
Based on these results we can clearly see that when a table is getting filled with cells that are marked as deleted, and thus significantly reduce the number of empty cells in the hash table (null != deleted), the probing sequences length grows significantly and damages the desired amortized runtime complexity.