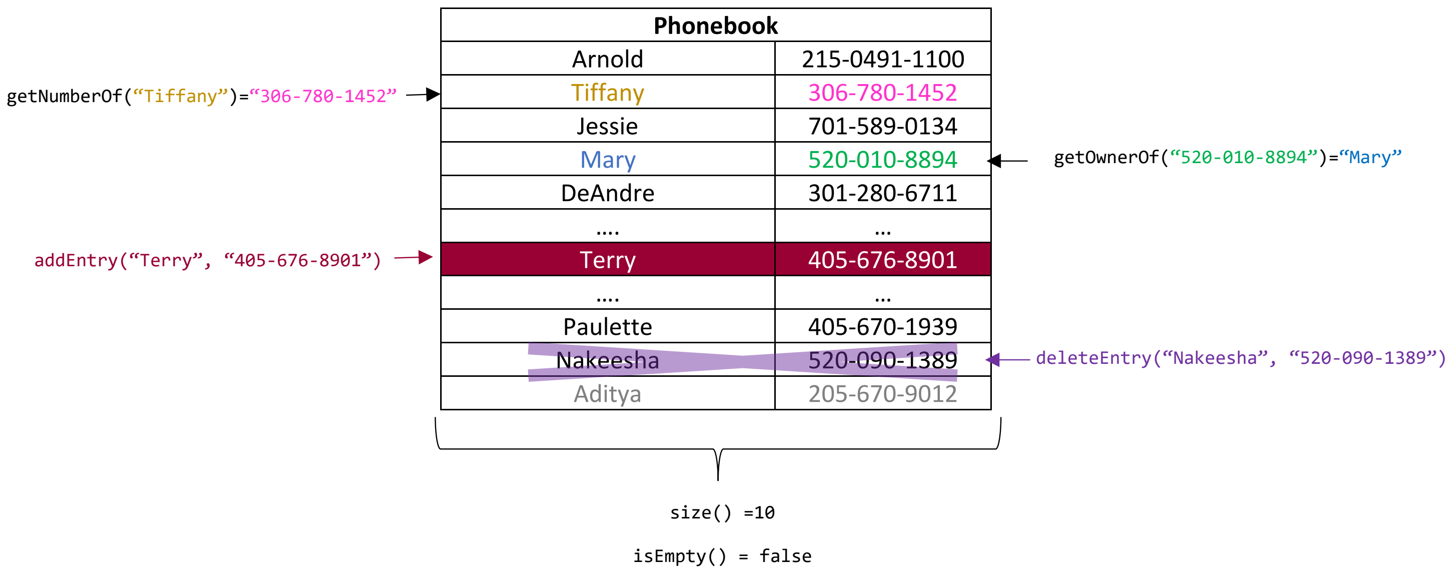
Writeup for third project of CMSC 420: “Data Structures” Section 0101, Spring 2019

Theme: Hash Tables

On-time deadline: Monday, 03-25, 11:59pm (midnight)
Late deadline (30% penalty): Wednesday, 03-27, 11:59pm (midnight)

1 Overview

In this project, you will have to implement an abstraction over a *phonebook*; A collection of pairs of type $< Full\_Name, Phone\_Number >$. Your phonebook will support **both** name-based search **and** phone-based search. See figure 1 for a pictorial view of the project.



Phonebook	
Arnold	215-0491-1100
Tiffany	306-780-1452
Jessie	701-589-0134
Mary	520-010-8894
DeAndre	301-280-6711
….	…
Terry	405-676-8901
….	…
Paulette	405-670-1939
~~Nakeesha~~	~~520-090-1389~~
Aditya	205-670-9012

getNumberOf(“Tiffany”)=“306-780-1452”
getOwnerOf(“520-010-8894”)=“Mary”
addEntry(“Terry”, “405-676-8901”)
deleteEntry(“Nakeesha”, “520-090-1389”)
size() =10
isEmpty() = false

Figure 1: A high-level view of how your phonebook is supposed to work.

To make both types of searches efficient, your phonebook will internally maintain **a pair of hash tables** from `Strings` to `Strings`: One will have the person’s name as a key and the phone number as a value, and the other one will have the phone number as a value and the name as a key. In your simple phonebook,

entry uniqueness is guaranteed: Every person has **exactly** one phone number, and every phone number is associated with **exactly one** person.

The hash tables maintained by the phonebook will be one of three types:

- One that uses *Separate Chaining* as a collision resolution technique (class `SeparateChainingHashTable`).
- One that uses *Linear Probing* as a collision resolution technique (class `LinearProbingHashTable`).
- One that uses *Quadratic Probing* as a collision resolution technique (class `QuadraticProbingHashTable`).

The central class of the project is `PhoneBook`. What is interesting about `PhoneBook` is that **it has been implemented for you**! However, the methods of `PhoneBook` depend on methods of the interface `HashTable`, which is extended by the three classes mentioned in the list above. What *you* will need to do is complete the implementation of these three classes so that their methods can support the methods of `PhoneBook`. The Release Tests **primarily** test methods of `PhoneBook` (approx. 90% of their code), while a smaller number of tests check if you are implementing basic hash table functionality correctly (e.g resizings, see below).

The various methods of `PhoneBook` will have to run in *amortized constant time* (except for `size()` and `isEmpty()`, which should run in *constant* time). Therefore, all of the instances of `HashTable` that you implement need to offer **amortized constant insertions, searches and deletions**. We **will** be checking your source code after submission to make sure you are **not** implementing the methods **inefficiently** (e.g logarithmic complexity, linear complexity, or even worse)! In practice, the only way you can do this is by **not** consulting the hash function **at all** for your operations; just looping over the entire table until you either find the element (`remove`, `containsKey`) or you find an empty position (`put`). While this would indeed allow you to pass the tests, we will be **checking your submission** to make sure you consult the hash function. Implementing all operations as mentioned above would constrain them to be linear time, which is **unacceptable** for both the project (i.e no credit for this project) and Computer Science **as a whole**. That said, **not all** of the methods you implement make use of the hash function, which means that their complexity parameters will necessarily be different.

2 Getting Started

You should first pull the starter code from our GitHub repo. After that, you should study the JavaDocs and source code of `PhoneBook` to understand how your methods can be used (and, therefore, tested!). The functions you have to implement are under the package `hashes`.

We include a package called `utils` with three classes: `PrimeGenerator`, `KVPair` and `KVPairList`. These are helpful utilities which you will quite possibly end up using in your project and we encourage you to read their documentation. We include some unit test libraries so that you can get ideas for implementing your own `HashTable` and `PhoneBook` tests.

You should fill in the `public` methods of `SeparateChainingHashTable`, `LinearProbingHashTable` and `QuadraticProbingHashTable`. You will notice that the `private` data fields that your `public` methods will operate on have been **given** to you. You should **NOT** edit these private fields, since they provide the **basic infrastructure necessary** for your various operations to attain the **efficiency** required by this project!

3 Quadratic Probing

In lecture we saw that Linear Probing is susceptible to the "clustering" phenomenon, where various different collision chains end up "crowding" next to each other and even "overlapping". This causes several collisions for even wildly different hash codes when compared to the ones that started the chains that have crowded each other. We also saw that tuning Linear Probing such that its "jump" is changed from 1 to some other number, e.g 2 or 3, does **not** solve the clustering problem: instead, the clusters become discontiguous.

This begs the question: *what if, instead of having a static offset to Linear Probing, we were to* **increase** *the "step" that the algorithm takes* ***every time it encounters a collision****?* One studied solution that implements this idea is **quadratic probing (QP)**. To explain how QP works, we will first mathematically formalize how Linear Probing (**LP**) works.

Suppose that our hash function is $h(k)$, where k is some input key. Let also $i \geq 1$ be an integer that denotes the i^{th} probe that we have had to endure during our search for an empty cell in the table. Assuming that our hash table employs LP, then the following **memory allocation function** $m_{lp}(k, i)$, returns the *actual cell index* of the i^{th} probe:

$$m_{lp}(k, i) = (h(k) + (i - 1)) \bmod M$$

This means that LP will probe the following memory addresses in the original hash table:

$$h(k) \bmod M, (h(k) + 1) \bmod M, (h(k) + 2) \bmod M, (h(k) + 3) \bmod M, \dots$$

which fits intuition. For example, in the LPHT shown in Figure 2, if we wanted to insert the key 22, we would have the sequential memory allocations: $m_{lp}(22, 1) = h(22) + (1 - 1) \bmod 11 = 22 \bmod 11 = 0, m_{lp}(22, 2) = \cdots = 1$ and $m_{lp}(22, 3) = 3$. On the other hand, if we wanted to insert the key 9, we would only need the single allocation $m_{lp}(9, 1) = 9$, since cell 9 is empty. Of course, we could also compute $m_{lp}(9, 2) = 10$ or $m_{lp}(9, 3) = 0$, but there is no reason to, since m_{lp} gave us an empty address in the first probe.

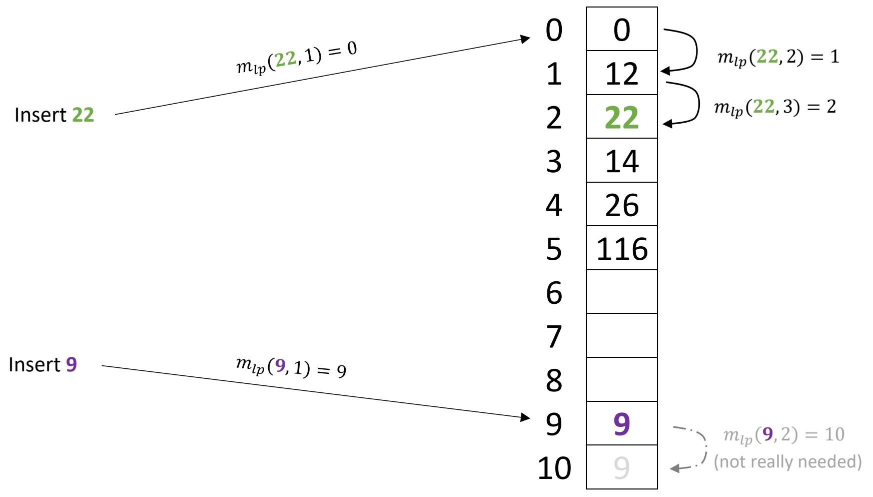


Figure 2: Examples of various memory allocations for two integer keys to be inserted into an LPHT.

QP, in its simplest form (which is the one you will implement in this project), employs the following memory allocation function m_{qp}:

$$m_{qp}(k, i) = \left(\ h(k) + (i - 1) + (i - 1)^2\ \right) \bmod M$$

which will lead into the following memory addresses being probed:

$$h(k) \bmod M, (h(k)+2)) \bmod M, (h(k)+6) \bmod M, (h(k)+12) \bmod M, \ldots$$

Note that the offset is **always** computed from the address that $h(k)$ probed. For example, if the table of Figure 2 were a QPHT instead of an LPHT, we would have the single memory allocations shown in Figure 3.

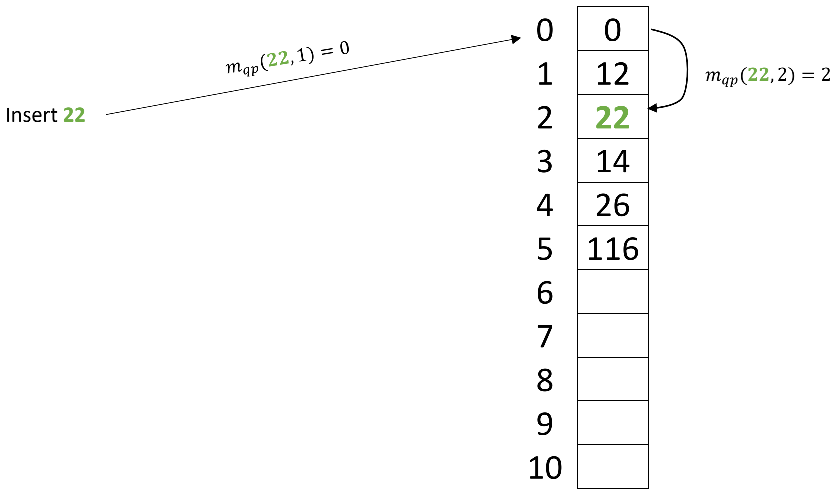


Figure 3: Example of insertion speed-up achieved by quadratic probing; $m_{qp}(22, 2)$ attains a bigger "jump" than $m_{lp}(22, 2)$ and reduces the number of probes required to insert 22 by 1. For practice, try inserting 11 into either one of the hash tables **after** you insert 22, and look at the speed-up attained by QP! ☺

The entire idea behind QP is that if a key collides with another, we need to *try harder to make it not collide in the immediate future*. By increasing the quadratic "step" every time that a collision happens, the algorithm hopes to disperse keys that collide more **aggressively**. Feel free to read the relevant Wikipedia article or scour the web for additional resources on how QP improves upon LP. It doesn't improve **universally**, though. For one, it doesn't display the **cache locality** that Linear Probing displays, especially for keys that collide a lot (so the value of i is large for them). Also, the hash function is just a tiny bit more **expensive** to compute, since there's another summand and a squaring involved.

4 FAQs

Q: *Why is it that* `SeparateChainingHashTable` *has two methods (*`enlarge()`, `shrink()`*) which are* **not** *part of the interface* `HashTable`*?*

A: Because enlarging or reducing the number of entries in a hash table implemented with Separate Chaining as its collision resolution strategy is a process that never *has* to happen **automatically** in order for its operations to work (particularly, insertions). Enlarging the hash table can lead to better *efficiency* of **insertions**, while reducing its size can lead to better storage tradeoffs after numerous **deletions** have happened. This means that changing the Separately Chained hash table's *capacity* is an issue that should be left with the caller to decide. Maybe the caller decides to enlarge when the capacity is at 70%; if so, the caller must **explicitly** make the call to `enlarge()` (similarly for `shrink()`). On the other hand, an openly addressed hash table will need to **internally mutate** the table in order to not just allow for better performance and storage trade-offs but, in the case of insertions, to even allow for the operation to **complete**!

Q: *Does this mean that I* **don't** *need such methods for my Openly Addressed Hash Tables, that is,* `LinearProbingHashTable` *and* `QuadraticProbingHashTable`*?*

A: You will **absolutely** need such methods, but they have no business being `public` methods. Method stubs for both `enlarge()` and `shrink()` have been provided for you in the project starter code, and you should keep them `private` after you implement them. It is the expectation that you will be calling these methods from within your `put()` and `remove()` method bodies.

Q: *How should these methods be implemented?*

A: For insertions, you should follow the approach that we have discussed in class: **the first insertion that takes place after your hash table is at 50% capacity or more should first trigger a resizing of the hash table to the largest prime number smaller than twice your current size, and then insert the new key.** For example, if the current hash table capacity is 7, the 4th insertion will cause the hash table to have a count of 4, which is ≈ 57.1% of the hash table size. We will **not** resize after the 4th insertion though, because we don't know whether a 5th one will come yet, and it is possible that we would be resizing "for nothing". **If** a 5th insertion is requested, we will **first** resize to 13, the largest prime number smaller than $2 \cdot 7 = 14$, and **then** we will insert the 5th element. Note that the element might be hashed to an address different than the one it would have been hashed to if we had **not** resized, since the hash code will be "modded" by a **new** hash table size.

It is important that you stick **exactly** to this guideline, because `HashTable` instances expose a `public` method called `capacity()` which checks for the actual hash table **size**, i.e the number of cells of the internal 1D array, whether they are **occupied or not**. This means that we can **test** for the return value of that method, and we will be expecting that you follow these guidelines to a tee. Feel free to use the class `utils.PrimeGenerator` to get the appropriate prime numbers for free. Check the file `StudentTests.java` for some examples of how we can test for the return value of `capacity()`.

Q: For an Openly Addressed Hash Table that is very sparse, doesn't it make sense to truncate its size to the first prime greater than its current number of elements, such that we save space?

A: Yes, in practice, you should. However, in this project, we assume that our hash tables are kept at a reasonable load factor so that you don't ever encounter significant sparsity.

Q: *Why did you implement your own linked list over* `KVPair` *instances (*`KVPairList`*) instead of just instantiating the* `private` *data field* `table` *of* `SeparateChainingHashTable` *with a* `java.util.LinkedList <KVPair>`*? Surely that is easier to do instead of writing your own list for the project and then testing it!*

A: For the exact same reason mentioned in the writeup for Project 1. If you remember, in `AdjacencyListGraph`, there was some information about why we were using an array of instances of a custom linked list (`NeighborList`) instead of an array of instances of any one of the various lists that Java provides.