

Writeup for third project of
CMSC 420: “Data Structures”
Section 0201, Fall 2018

Theme: Hash Tables

On-time deadline: Friday, 11-02, 11:59pm (midnight)

Late deadline (30% penalty): Sunday, 11-04, 11:59pm (midnight)

1 Overview

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

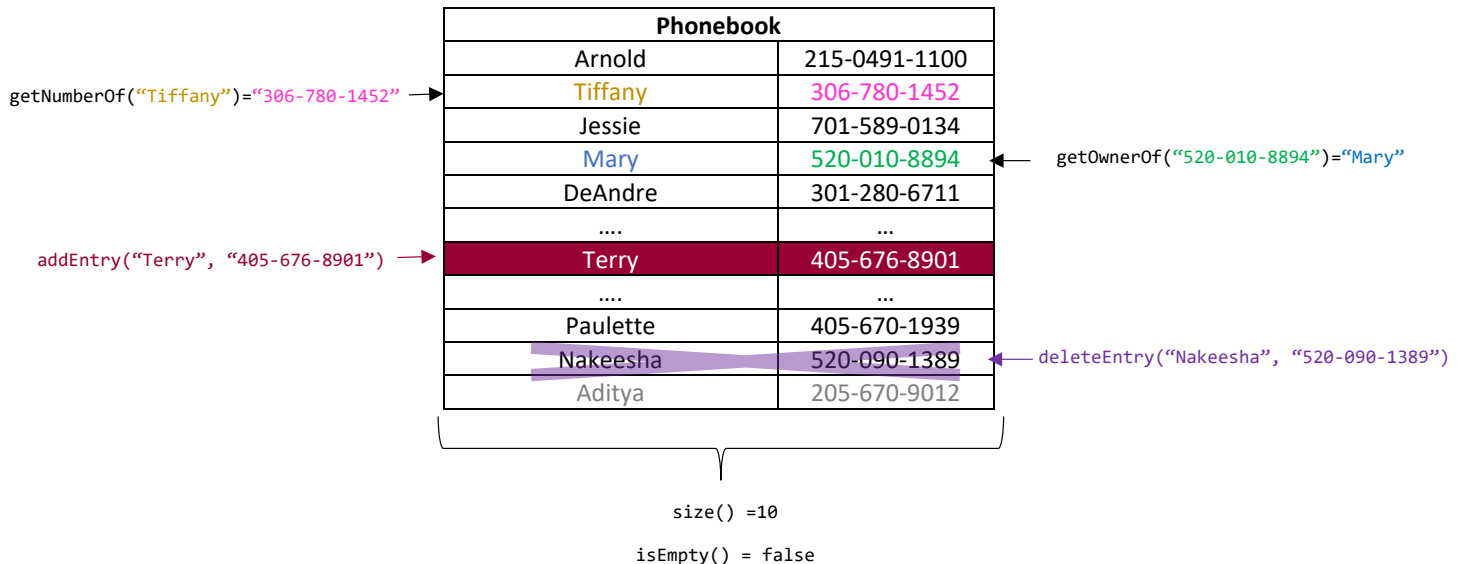


Figure 1: A high-level view of how your phonebook is supposed to work.

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)!

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 to “tune” 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 “discontinuous”.

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. Assuming that a table employs LP, then the following **memory allocation function** $m_{lp}(k, i)$, where $i \geq 1$, denotes the i^{th} hash table cell probed to find an empty cell in the table, 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: $m_{lp}(22, 1) = h(22) + (1 - 1) \bmod 11 = 22 \bmod 11 = 0$, $m_{lp}(22, 2) = \dots = 1$ 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.

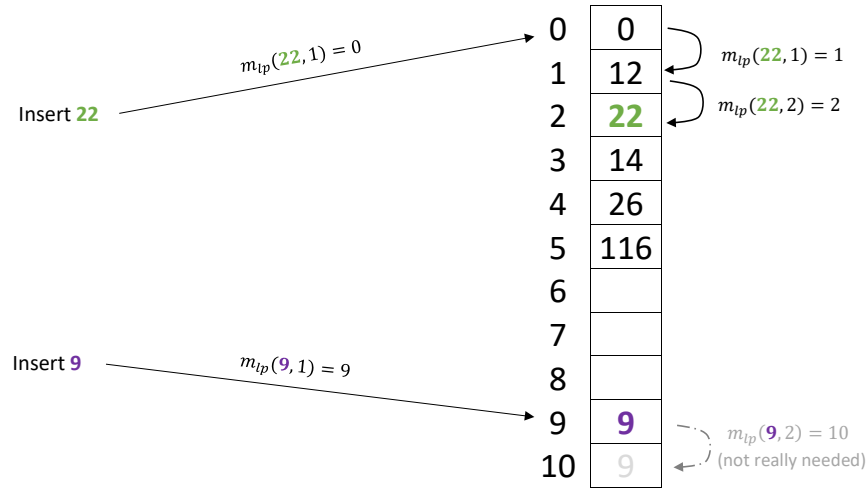


Figure 2: Examples of various memory allocations for two integer keys to be inserted into an LPHT.

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

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

which will lead into the following memory addresses being probed:

$$\underbrace{h(k) \bmod M, (h(k) + 1^2) \bmod M}_{\text{First two addresses probed same as in LP}}, (h(k) + 2^2) \bmod M, (h(k) + 3^2) \bmod M, \dots$$

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 memory allocations shown in Figure 3.

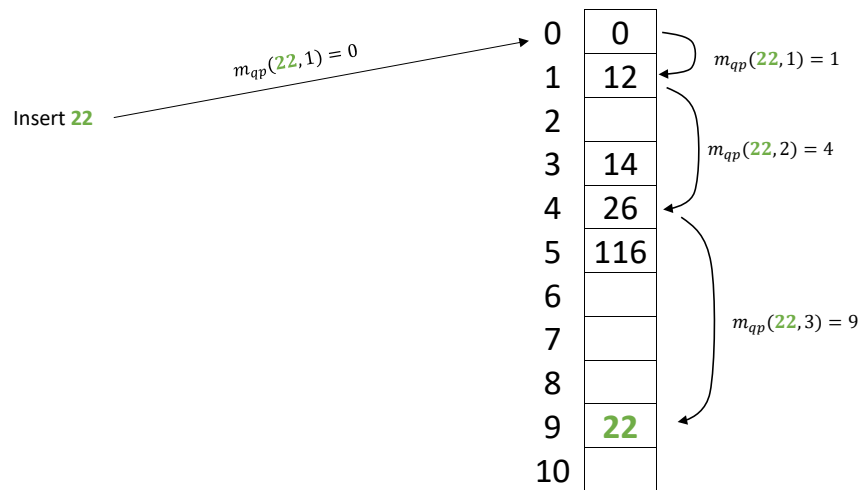


Figure 3: Examples of the memory addresses probed by $m_{qp}(22, i)$ for $i = 1, 2, 3$. Notice how the first two addresses probed are **exactly the same** as those probed by $m_{lp}(\cdot)$; this is natural, since $0^2 = 0$ and $1^2 = 1$. For practice, go ahead and insert 22 **after** you have inserted 9!

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?*

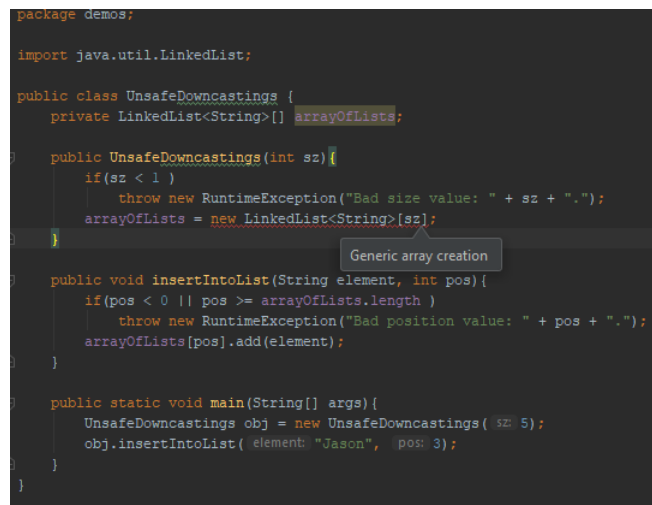
A: For insertions, you should follow the approach that we have discussed in class: **the first insertion that takes place after your hash table is at 50% capacity or more should first trigger a resizing of the hash table to the largest prime number smaller than twice your current size, and then insert 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 $\approx 57.1\%$ 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).

For successful deletions, we will follow the **symmetric** approach: **if a deletion is requested for a hash table with a capacity M at most 50% (not $< 50\%$ but $\leq 50\%$!), we will find the smallest prime larger than $\frac{M}{2}$, allocate a new array of that size and then re-insert all the elements except for the one to be deleted.** For example, if our hash table size is $M = 19$, we have 9 elements inside it and we request a deletion, we will resize to $M' = 11$, the smallest prime greater than $\frac{19}{2} = 9.5$, and then we will re-insert all elements of the table except for the one that we want to delete. Please make sure that you understand how this simple deletion scheme differs from the "normal" deletion scheme for Openly Addressed Hash Tables that we have discussed in lecture! Also, please make sure that the deletion that triggers the resizing is **successful**, i.e make sure that the key is **in** your table **before** you perform the resizing.

It is important that you stick **exactly** to these guidelines, 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.

- 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:** Because *creating a raw array over generic types* in Java is a pain. As you can see in figure 4, in Java it is **not** possible to create a raw array over generics.



```
package demos;

import java.util.LinkedList;

public class UnsafeDowncastings {
    private LinkedList<String>[] arrayOfLists;

    public UnsafeDowncastings(int sz){
        if(sz < 1)
            throw new RuntimeException("Bad size value: " + sz + ".");
        arrayOfLists = new LinkedList<String>[sz];
    }

    public void insertIntoList(String element, int pos){
        if(pos < 0 || pos >= arrayOfLists.length)
            throw new RuntimeException("Bad position value: " + pos + ".");
        arrayOfLists[pos].add(element);
    }

    public static void main(String[] args){
        UnsafeDowncastings obj = new UnsafeDowncastings( 5);
        obj.insertIntoList( element: "Jason", pos: 3);
    }
}
```

The screenshot shows a Java IDE with a code editor. The code defines a class `UnsafeDowncastings` with a private field `arrayOfLists` of type `LinkedList<String>[]`. The constructor `UnsafeDowncastings(int sz)` attempts to create an array of `LinkedList<String>` objects. A red squiggly line under the array creation `new LinkedList<String>[sz]` indicates a compile-time error. A tooltip labeled "Generic array creation" points to this line, indicating that Java does not allow the creation of arrays of generic types.

Figure 4: Creating an array of generic types leads to a **compile-time** error.

Instead, one would need to declare a raw array over `Objects` and then hope that the downcastings involved will not be unsafe. Unfortunately, `java.util.LinkedList` is a type that implements the interface `Iterable` (and so are *all* `java.util.Lists`), an interface that is **not** implemented by `java.lang.Object`. This in turn means that the downcasting of *any* `Object` reference to a `LinkedList`

reference is **inherently unsafe**, since the line of code that is making the downcasting might request access to the `iterator()` method available to a `LinkedList` instance, but **unavailable** to an `Object` instance. This is further demonstrated by Figure 5:

```
package demos;

import java.util.LinkedList;

public class UnsafeDowncastings {
    private Object[] arrayOfLists;

    public UnsafeDowncastings(int sz){
        if(sz < 1 )
            throw new RuntimeException("Bad size value: " + sz + ".");
        arrayOfLists = ((LinkedList<String>[])new Object[sz]);
    }

    public void insertIntoList(String element, int pos){
        if(pos < 0 || pos >= arrayOfLists.length )
            throw new RuntimeException("Bad position value: " + pos + ".");
        ((LinkedList<String>)arrayOfLists[pos]).add(element);
    }

    public static void main(String[] args){
        UnsafeDowncastings obj = new UnsafeDowncastings( 5);
        obj.insertIntoList( element: "Jason", pos: 3);
    }
}
```

Figure 5: An example of an unchecked and unsafe downcast. This small snippet of code has been included for you under the package `demos` on our Git repo, for your experimentation. The code, as shown, throws an instance of `java.lang.ClassCastException`.

An alternative would be to drop the raw array and use a `java.util.ArrayList`, but this causes another problem: the fact that *the way that ArrayLists are internally resized is implementation-dependent and completely transparent* to the user. This means that you might have your hash table internally resizing to a way other than those that we have discussed in class, and this can affect the quality of your hashing. For example, if we use an `ArrayList` which **doubles** itself whenever its capacity is at 50%, this would re-insert the elements in the resized `ArrayList` in their current order instead of using our hash function! In order to allow for a project where it is **your** responsibility to maintain the size of the hash table **appropriately**, for all given hash tables, we need to stick to raw arrays, and this means getting entirely **rid** of generics.