Employee Database – Part 1

A ***table*** has two important features: it's a collection of pairs of information, and each pair consists of a key and an associated value.

Access to table elements is generally by key: we ask for the value that's associated with a given key. There can't be any more than one value associated with a given key. For example, if the key "mike" was associated with the value 27, it would not also be associated with the value 45.

One observation that you may have come up with is that if the keys in the table are integers, you can use a normal array as a table with O(1) (constant time) access. Simply store the values in an array and access them immediately using array indexing. For example, if the table were an array of 100 elements that stores the entries `(5, "Clancy")`, `(17, "Wei")`, `(42, "Waliany")`, and `(83, "Hale")`, the value associated with 5 is just `table[5]`, and we can access that value in constant time.

Easy, right? Unfortunately, this only works with integer keys (which is what we will be working with in this lab). However, ***hashing*** is a technique for extending this constant-time access to non-integer keys when accessing a table's array elements.

Complete the following problems to check your understanding before proceeding:

1. Given the following keys:

   ```
   10 100 32 45 58 126 1 29 200 400 15
   ```

   Devise a hash function that, when used with an empty table of size 11, produces three or fewer collisions when the eleven keys are added to the table. Remember that hash codes should be easy and fast to compute (they shouldn't be complicated or expensive functions).

2. What is the run-time (Big-O) of retrieving the value associated with a specific key, given a hash table where all keys have unique hash codes?

3. What is the run-time (Big-O) of retrieving the value associated with a specific key, given a hash table where all the keys happen to have the same hash code, given an open hashing collision resolution technique?

Employee Database

Write a class **EmployeeDatabase.java** that will function as a hash table (map) of <`int`, Employee> pairs, where the `int` is a unique five-digit employee ID (we will keep this as an `int`, rather than wrap it into a class, for the sake of simplicity).

Your EmployeeDatabase class should have at least one field, an array of Entry objects. Entry is an inner class that encapsulates the data that will be held at a specific location (a hash code or hash index) in the hash table.

The Entry class should be private and should have `int ID` and `Employee employee` fields, along with applicable constructor, overridden `toString` method, etc.

The Employee class should (at least) have a `name` (obviously in a real program this class would include all relevant details about this employee – salary, benefits, etc.).

Your hash table will use a ***closed hashing*** approach, where collision resolutions take place within the table (array) itself. You will need to implement the following collision resolution schemes:

1. Linear probing
2. Quadratic probing

```
/* there are more collision resolution techniques, e.g. pseudo-random probing or double
hashing, that are better than the above, and may be useful to know */
```

First, make a copy of EmployeeDatabase and name it EmployeeDatabaseQuadratic then rename the original EmployeeDatabaseLinear. In EmployeeDatabaseLinear write a method `int hashCode(int key)` that will return the hash code for the given key (in this case, the key will be the five-digit employee ID number). Include an algorithm that you think is a good hash function. A good hash function aims for an even distribution, and should use the maximum amount of storage while producing the least amount of collisions. Because the keys for this table are (conveniently!) integers, a decent hash function shouldn't be too hard to devise.

Next, complete the methods below. When faced with a hash code collision, you should implement the collision resolution schemes mentioned previously.

- `put(key, value)`
- `get(key)`

In a **Tester.java** (linear probe) class with a `main` method, thoroughly test your map. Add some <key, value> pairs, attempt to retrieve a value for a key that does / doesn't exist, generate some collisions to ensure collision resolution works the way you think it should, etc. Then implement the following methods for EmployeeDatabaseQuadratic and create a separate **Tester.java** (quadratic probe).

- `int hashCode(int key)`
- `put(key, value)`
- `get(key)`