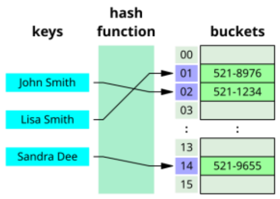

Hash Table Lab

The aim of this lab is to develop a data structure that can achieve a constant-time search through a container class, or at least get immeasurably close to constant time O(1). You will first generate data for 10,000 employees, each of which has a unique employee number between 0 and 9,999 and a name. Once written out to a text file, your main program will read in those employees into a hash table and give the client to search for an employee with a particular name. Once found, the program will let you know what their employee number is. We will time the search to see if it is independent from the size.

- As a refresher for file input and output, create a method that will generate 10,000 data elements and write them into a file. Each data element describes an Employee which will consist of a 4 digit employee-number and an employee name: a randomly generated word from 3-8 characters long. Each name will start with a consonant and alternate between random consonants and random vowels.

- Creating an Employee object would be a good design decision:

```
public class Employee implements Comparable
{     //only needs to be Comparable if using sorted containers
      private int employeeNumber;      //unique for each instance
      private String name;
      /*now define the following methods...
        constructors, get methods
        override String toString()     //employeeNumber+" "+name
        override int hashCode()
        override boolean equals
        int compareTo                  //will compare by name */
}
```

Write the 10,000 Employees into a file called "hashData.txt" with each line containing the Employee's unique number and randomly generated name, separated by a space:

```
0 rigec
1 pab
2 cojehap
…
9998 qove
9999 wudaqe
```

Once the file is created, you will not need to create it again. Now we can write the program that will read in these employees from the file into a Hash Table, then allow the client to search for a particular employee to retrieve their number. If done well, our search-time will be O(1).

Write a program that will then read in the file into a hashed container class. Your objective is to create and implement a data structure that conserves space and limits run-time, even though these two things often work against each other. The idea is that a user will type in an employee name to search for, and the program will then return the four-digit employee-number that is associated with that name. So, when reading the data into the hashed container class, use the employee-name to produce a hash-code, which will then be used to direct where the data is stored in the container class. If more that one item yields the same hash-code, find a place for these colliding data items to go. You may implement any of these time-proven methods:

- An array with linear probing to resolve collisions.
- A 2-dimensional array where each column is a bucket to store unsorted collisions.
- An array of linked lists where each "chain" stores unsorted collisions.
- An array of binary tree roots where each tree stores sorted collisions.

Special note: you are not allowed to use a `java.util.HashSet` nor a `java.util.HashMap`, since the point of this lab is to make our own. That would be equivalent of using a `java.util.ArrayList` to make MyArrayList.

Or, of you are particularly creative, develop your own container class. Again, the idea is to get close to a O(1) search for any item in the data structure. Realistically, there will likely be some collisions. If you are resolving collisions through chaining with an array of 1,000 chains, this will result in averaging 10 collisions per chain. You need to make sure that your hash codes are evenly distributed across the entire range of your array indexes. The bigger your array is, the less collisions you should have and therefore a faster search, but at the expense of more memory. Consider the analogy of a communities' parking lot, where each resident gets an assigned parking space (the exact index of their hash code) and two visitor spaces. Residents are more likely to park close to where they want to go, but the parking lot needs to take up more space.

After the data has been read in and stored in the hash table, create a means for a user to type in a name to search for. If the program can find an employee with that name, it will report back the accompanying employee-number, and quickly. Report how much time was taken to do the search. `System.currentTimeMillis()` will return a long of the number of milliseconds that have elapsed since January 1, 1970. If you record the current millis before the search and again after, take the difference of times and divide by 1000.0 to get the time used for the search. Now let's push the computer: make your data file store 10 million employees and see how much time it takes to search in your hash table.