Course: CSC 115 (J)

Lab 09: Working with Hashtable

Objectives

You will be working on a basic hashtable that uses one of two provided hashing functions and resolves any collisions that occur when hashing using linear probing.

During lab 9, you will:

- Complete the *isFull*, *insert*, *find*, and *increaseCapacity* methods of a basic hashtable.
- Track the number of collisions that occur as items are added to the table using two different hash functions and table sizes.

Part I – Introduction to Hashtable

A hashtable is a data structure that uses a hash function to map identifying values (known as keys) to associated values, for example, mapping a name to a telephone number. Thus, a hash table implements an associative array. The hash function is used to transform the key into the index (the hash) of an array element (the slot or bucket) where the corresponding value is to be sought.

Ideally, the hash function should map each possible key to a unique slot index, but this ideal is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after it is created). Instead, most hash table designs assume that hash collisions—different keys that map to the same hash value—will occur and must be accommodated in some way.

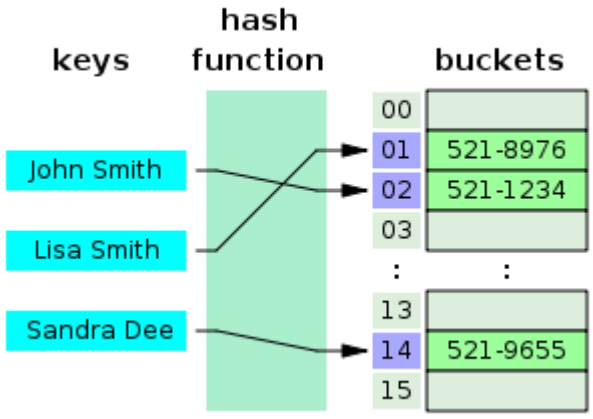

Figure 1: A small phone book as a hashtable

[For further details please visit http://en.wikipedia.org/wiki/Hash_table]

Part II – Completing a basic Hashtable

1. Download *KeyValuePair.java* from connex.
2. Download and complete *Hashtable.java*.
    - Start by inspecting the *KeyValuePair* class. The hashtable will store an array of *KeyValuePairs* that make up the table. Each *KeyValuePair* stores an Object, and an integer key that identifies that object.
    - The *Hashtable* class already includes some basic methods. Most importantly, it stores an array of *KeyValuePair* objects which serve as the table, along with integers storing the capacity of the hashtable (*tableSize*), and the number of items currently stored in the hashtable (*size*).
    - Spend a few moments reading through the constructor and the *toString()* method. Make sure you understand what they are doing before you begin writing code.
    - First, fill in the *isFull* method.
        - This method returns true if the hashtable is full, false otherwise.
    - Next, fill in the *insert* method.
        - It takes in a key and a value, and should insert a *KeyValuePair* containing them at the proper index in the table.
        - You will need to use one of the two provided hash functions to turn the key into an index in the table array. Start by using *hashFunction1*.
        - A collision occurs if a value hashes to a location in the table array that is already occupied. For this task, we will use linear probing to resolve such collisions. This means you should keep checking the next spot in the array until you find an empty place to put the data.
        - **Compile** and **run** your Hashtable to test the insert method. Make sure you understand the output it produces.
    - Fill in the *find* method.
        - It takes in a key, and should return the Object in the table that corresponds to that key, or null if the table does not contain any objects with that key.
        - First, use the same hash function you used for *insert* to transform the key into an index in the table array.
        - Check that spot to see if it is the data you are looking for. If not, keep checking the next index until you find the target of your search, or you hit an empty space, which means the target key is not in the table, and you should return a null.
        - Uncomment the next section of the main method and use it to **test** your *find* method. Make sure it works before moving on.

Part III – Tracking Collisions

- One way we can determine the efficiency of the hashtable is to count the number of collisions that occur as we insert items.
- Change your *insert* method so that it keeps track of every collision that occurs. You can use the *collisionCount* instance variable to store this number of collisions.
- Uncomment the next section of the *main* method and use it to **test** your collision count.
- Try changing your *insert* and find *method* to use *hashFunction2* instead of *hashFunction1*. Test the program again and compare the collision count. There should be a noticeable difference between the two. A hash function that better spreads the data will produce fewer collisions

Part IV – Changing the capacity of a Hashtable

- The efficiency of a hashtable is based on how full it is. As the load factor gets close to 1.0, many more collisions will occur. One strategy to deal with this is to expand the hashtable so that there will be fewer collisions.
- Complete the *increaseCapacity* method so that it increases the capacity of the hashtable based on the parameter being passed in.
    - Store a reference to the current table (we'll call it *tempTable*).
    - Create a new table array with the specified capacity.
    - Reset the *size*, *tableSize*, and *collisionCount* for this new table.
    - Grab each piece of data in the *tempTable* and insert it into the newly expanded table.
- Uncomment the next section of the *main* method and use it to **test** your *increaseCapacity* method. Make sure it works as expected.
- Test the two different hash functions with the bigger table size, and compare the number of collisions that occur.

Have fun