CS 251: Data Structures and Algorithms
Fall 2018
Project 2: Hash Tables
Max Points: 100
Submission on Vocareum due 11:59pm Friday, Sept. 28, 2018

Overview:

The purpose of this project is to develop a thorough understanding of hash tables and their applications. Hash tables are an important and frequently used data structure for which the lookup (search), insert, and delete operations take a small constant time on average. There are three parts in this project: Parts 1.1 and 2 deal with implementations of a hash table. Part 1.2 deals with an application of hash tables. There is value to be had in reading the entire assignment description before starting your work.

Part 1.1: Implement a Hash Table with Chaining (40 points)

You are required to write a Class that implements a hash table with separate chaining. This variation of the hash table maintains an array of linked lists. The hash table array is accessed using an index called the hash value, which is computed by the function

Hashvalue(hashcode) = (hashcode(key)) % capacity_of_hashtable

where the % symbol represents the modulus operator, and size_of_hash_table is the total number of entries in the table. The hashvalue(hashcode) function depends on the hash code, which is computed as follows using following pseudocode:

```
hashcode = 0
for i=0  to keylength-1
   hashcode = (37 * hashcode + key.charAt( i )) % capacity_of_hashtable;
```

The hashvalue is used to index the hash table array. Each table entry is a pointer to the head of an associated linked list in which the record information associated with the key is stored. To perform an insertion operation, compute the hashvalue for the key then insert the record at the head of the linked list. Similarly, for the lookup operation (search), a hashvalue is computed for the key, and the corresponding linked list is traversed until the requested key is found or the end of the linked list is reached. For more information on hash tables, please refer to lecture notes and the course text book.

You should name your Class *HashtableChaining* and implement the following functions:

- A constructor that accepts the size of the hash table as a parameter and initializes the table with empty linked lists at each index
- A function *insert(key, value)* that adds the key to the hash table. A <key, value> pair associates value/record with a key.
- A function *remove(key)* that removes the key from the table, if it is in the hash table.
- A function *contains(key)* that returns a Boolean indicating if the key is in the table.
- A function *size()* that returns the number of keys in the table.
- A function *hash(key)* that returns the hashvalue of the hashcode of a key.

- A function *rehash()* to resize and rehash the hashtable if the load factor goes above a threshold.

For Part 1.1 you may assume that all the input, i.e. keys, are strings and all values are integers. All input strings are combinations of lowercase and/or uppercase characters. All strings are delimited by a newline character. The initial size of the hash table will be provided, and it will be a prime number. The threshold for the load factor is 0.75, i.e. once the hash table reaches or exceeds 75% of its capacity, you must invoke rehash(). You should check if the hash table needs to be rehashed after every insertion. When you need to rehash the hash table, the new size must be the least prime number greater than twice the previous size. Remember, all keys must be re-hashed before being stored in the resized table.

Part 1.2: Application of a Hash Table (20 points)

Given an array of strings, write a function *mostFrequentStrings(String[] in)* using your implementation of Class *HashtableChaining* that computes the frequency of each string and returns an array of strings consisting of the 5 most frequent strings. You have to do this using the provided *Pair* class. Again, the strings will be combinations of lowercase and/or uppercase characters delimited by a newline character.

Part 2: Implement Quadratic Probing (an open addressing scheme) with Lazy Deletion (40 points)

You will implement a variant of the hash table that uses quadratic probing. After computing *hash(key)* this variant stores the value in the hash table array instead of in a linked list external to the array. To resolve the primary clustering problem use **quadratic probing**. With quadratic probing, rather than increasing the table index by i, as in linear probing, move i^2 spots from the point of collision, where *i* is the number of attempts to resolve the collision (see slide 32 of Week 3.1 Hashing1.pdf).

Lazy Deletion: When resolving collisions using quadratic probing one needs to remove keys from the table in a way that does not cause future search operations to fail. One strategy to do this is “lazy deletion,” which involves two steps, one that is immediate and one that comes later, the lazy step.

The first step replaces the key to be removed with an ‘inactive’ flag but does not reclaim the table cell for use by a new key. The use of the inactive flag prevents a future search from ending too soon. The search function is written to continue past any inactive cells until either finding the key or encountering a null cell, which is a cell that has never been used to hold a key since the time that the hash table was created.

The second step is to decide when to and with what key to replace the contents of the inactive cell. This can be implemented by following ways:

contains(key) method: The second step is to re-write the contains(key) function to keep track of the first inactive cell it encounters during quadratic probing and to swap the key of a successful search to the inactive table cell on the probe sequence. This completes the removal of a key. This reduces the search time for this key.

insert(key) method: The insert(key) method fills an inactive cell with a key (provided the key hashes to that cell) that requires only 1 probe for access but does not shorten the probe sequences for keys that collided earlier.

You should name your class *QuadraticProbing* and implement the following methods:

- A function *insert(key)* that inserts the key to the table. If the key maps to an inactive cell, you must make it active and replace the contents of the cell with this key.
- A function *remove(key)* that does a lazy deletion of the key from the table.
- A function *contains*(key) that returns a Boolean after checking if the key is in the table. You should swap the position of key with the first inactive cell encountered in the probe.
- A function *rehash()* to resize and rehash the hash table if the load factor goes above a threshold.
- A function *hash(key)* that returns the hashvalue of a key.
- A function *probe(key)* that returns the number of collisions using Quadratic Probing for that particular key. This value will depend on how the key is inserted/changed its position earlier (either by Insert() or contains() respectively).

The load factor for quadratic probing must be maintained at less than 0.4 (you should rehash when the hash table reaches or exceeds 40% full). rehash() should be implemented as in Part 1.1.

Note on testing

We plan to test the hash table implementations on a large number of random strings (on the order of thousands). All input strings are combinations of either lowercase or uppercase characters. All strings would be delimited by a newline character. A sample test case is provided for each of separate chaining and quadratic probing.