Coms 262 Project 3A - Hash Tables Spring 2018

Introduction

When we store data in a list, we typically place it in whatever spot is convenient. For example, if we are storing the name `John Doe`, we might write

```
list.pushBack("John Doe");
```

or if we want to place it in a specific position, we might write

```
list.insert(pos, "John Doe");
```

In either case, if we go back later to find the name `John Doe`, we probably have to search the list for it because we no longer know exactly where it is. Wouldn't it be great if we could know ahead of time exactly where `John Doe` is stored. That would be possible if the list were designed in such a way that there was only one place where `John Doe` could be stored. That is exactly what a *hash table* does.

Hash Tables

Let's assume that we are storing names and e-mail addresses, so that when we look up `John Doe` in the list, we will get back his e-mail address (which is `john@hsc.edu`). A simplistic method would be to use the first letter of the person's name and interpret it as a position. So for `John Doe`, we would use the `J`, which has an ASCII value of `74`. We would store `John Doe` (and his e-mail address) in position `74` of the list. An obvious problem with this scheme is that there are many names that begin with the letter `J`. We cannot store all of them in position `74`. Furthermore, `63` is the ASCII value of the question mark `'?'`. It is unlikely that anyone's name begins with a question mark, so that position would never be used.

On the other hand, if we used the entire name `John Doe`, and added up the ASCII values of all the characters (including the blank), then we would have a number that would probably not match any other name. In the case of John Doe, that number is

$$74 + 111 + 104 + 110 + 32 + 68 + 111 + 101 = 711.$$

What if the list has only 250 positions? Then we divide by 250 and use the remainder. In other words, we use $711 \bmod 250 = 211$. The *hash function* h is the function that computes this. In our example,

$$\begin{aligned} h(\texttt{"John Doe"}) &= (\texttt{'J'} + \texttt{'o'} + \texttt{'h'} + \texttt{'n'} + \texttt{' '} + \texttt{'D'} + \texttt{'o'} + \texttt{'e'}) \bmod 250 \\ &= 711 \bmod 250 \\ &= 211. \end{aligned}$$

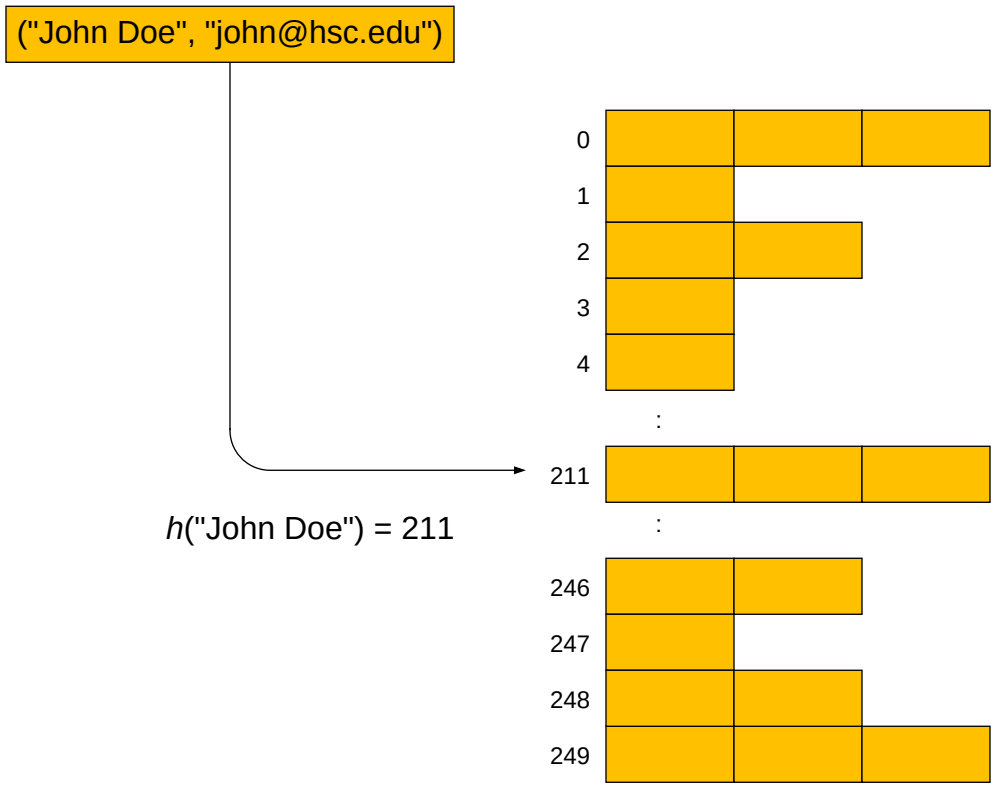
In this example, the name `John Doe` is called the *key*. It is used to determine the location. The rest of the data (the e-mail address) is called the *value*. Each position in the hash table holds a key and a value. Later, if we want to look up John Doe's e-mail address, we simply repeat the calculation and get the index 211. Then we know that John Doe's e-mail address is stored in position 211 of the hash table.

Collisions

It is not hard to imagine that two different names n_1 and n_2 could produce the same index. That is, $h(n_1) = h(n_2)$ even though $n_1 \neq n_2$. In other words, h is not a one-to-one function. Because of this possibility, at each index of the array, rather than store a single record we will store a *list* of records. Each such list is called a *bucket*. The bucket will contain all records whose keys produce that index.

Obviously, it is best to avoid collisions. To this end, we would like our hash function to distribute the indexes as evenly as possible. Ideally, the keys would produce numbers with a uniform distribution. An easy way to achieve that is to use the numerical value of the key (e.g., 711) as the seed in the built-in random number generator `rand()`. Thus, in the above example, we would first compute 711, then pass `711` to `srand()` to initialize the seed, then use `rand()` to get a random number, and finally, mod it by the size of the array. It so happens that `John Doe` produces the random integer `11716`. Had the name been `John Roe`, the random integer would have been `18076`. Had the name been `John Coe`, the random integer would have been `27645`. You get the idea: any change at all in the name will produce a completely different index, and the indexes produced will be distributed evenly over the array. If the array size were 250, then these integers would give us indexes `216`, `76`, and `145`. Note that they are well distributed.

The data structure looks like this.



Resizing the Hash Table

We will define the *size* of the hash table to be the number of records stored in the table and define the *capacity* of the table to be the number of buckets.

Clearly, the larger the hash table, the less likely will be a collision, and the smaller the hash table, the more likely a collision. Making the hash table too small will result in too many collisions. Making the array too large will result in wasted memory. The rule of thumb is that the size the hash table should be no greater than 3/4 the number of buckets and no less than 1/4 the number of buckets. So if the hash table has 400 buckets, then it will function most efficiently when its size is between 100 and 300 records.

We will follow these rules: Once the table size reaches 3/4 of its capacity, we will double its capacity. As records are deleted, once the table size drops as low as 1/4 of its capacity, we will halve its capacity. Both of these operations will require a `resize()` function. However, under no circumstances should the number of buckets be less than 8.

We must be aware that when the table is resized, the hash function will produce new indexes for each element. In the earlier example, after using the `rand()` function, the three records were stored in locations `216`, `76`, and `145`. If the size is halved to `125`, then those same three records will be stored in location 11716 mod 125 = `91`, 18076 mod 125 = `76`, and 27645 mod 125 = `20`. Thus, when we copy records from the old memory to the new memory, we must use the hash function to determine each record's location. A similar phenomenon occurs when the capacity is doubled. The details are discussed below.

The `HashtableEntry` Class

The `HashtableEntry` template class is an extremely simple class. A `HashtableEntry` object consists of a key and a value. The key is a string and the value is an object of type `T`. Note that this is a template class. See the document `HashtableEntry Class.pdf`.

The `Hashtable` Class

The `Hashtable` class is described in detail in the document `Hashtable Class.pdf`. Read the document carefully. Note a few things.

- The `m_bucket` data member has data type

  `ArrayList<LinkedList<HashtableEntry<T>>>`.

  That is, it is an array list of linked lists of `HashtableEntry` objects. An array list is used so that buckets can be accessed quickly. A sequential search is done to locate the `HashtableEntry` object within the bucket, which is acceptable because the buckets will always be very small, seldom having more than three members, even when the table is filled to 3/4 capacity. Therefore, a sequential search of the buckets will be very fast.

- The default capacity of the hash table is 8 buckets. When the hash table is emptied, the empty will still contain 8 buckets, although each bucket will be empty.

The `State` Class

I have compiled data about the 50 states of the United States in a file named `State Data.txt`. The data include the state abbreviation, name, capital city, population, etc. Create a `State` class that contains the data for a single state. This is a very simple class. The details are in the document `State Class.pdf`.

The Test Programs

I have written a two simple test programs named `HashtableEntryTest.cpp` and `HashtableTest.cpp`. `HashtableEntryTest.cpp` creates objects of type `Hashtable-Entry<State>`, using the `State` class. The program creates hashtable entries from some fictitious states (Atopia and Krasnovia) and then performs basic operations on those entries. For each state, the state's two-letter abbreviation serves as the key. The data about the state (including its two-letter abbreviation) constitute the value of the entry. Use this program to test your `HashtableEntry` class.

`HashtableTest.cpp` begins by creating an empty hash table with capacity 8. Then it stores 12 records in the table, which should quadruple the capacity to 32 (once when the size reaches $6 = 3/4$ of 8 and again with the size reaches $12 = 3/4$ of 16). After storing the records, the structure of the table is displayed. Then the records themselves are displayed in a nicely formatted style (according to the `display()` function defined in the `State` class). Then the twelve states are retrieved individuall and displayed.

After an attempt to retrieve a non-existent state, all but 2 of the records are deleted, causing the capacity to be halved twice (once when the size is $8 = 1/4$ of and again when the size is $4 = 1/4$ of 16, but not when the size is $2 = 1/4$ of 8). After deleting those records, the program displays the structure of the table and then attempts to retrieve one of the deleted records, an attempt which should fail. Use the test program to test your `Hashtable` class.

When you are finished testing your work, turn in the files `arraylist.h`, `linkedlist.h`, `linkedlistnode.h`, `hashtable.h`, `hashtableentry.h`, `state.h`, and `state.cpp`. Your work is due by 5:00 pm Monday, March 19. (On Monday, March 19, Project 3B will be assigned. You want to be sure that Project 3A works.)