User Account Management

Due: 23:59, Friday, April 08, 2016

In this assignment, you will implement a system for maintaining user account login information. The record system will use a (non-template) hash table class to store a set of `UserAccount`s. Collisions will be dealt with using separate chaining, and will be implemented as an *array of templated singly-linked lists*. You may find it helpful to refer to the postfix example already posted on the course website if you are not already familiar with linked list implementation by now.

Hash Table Class

Implement a hash table class called `HashTable` to store `UserAccount` objects. The items will be inserted based on a hash function applied to a string attribute. Your `Insert`, `Search`, `Retrieve`, and `Remove` methods must handle the situation where different items map to the same hash value (i.e. collision), and this should be handled structurally by inserting/finding/removing the value in the linked list at the given index. Please see the provided hashtable.h header file for the functional requirements of the `HashTable` class. You may add private member functions to hashtableprivate.h.

Hash Function

Insertions should be mapped to array indices using this hash function: $hash\_value$ modulo $array\_size$. To achieve this, your functions must first generate a hash value from a given string. This hash value should be calculated by assigning the values 1 to 26 to the letters 'a' to 'z' (regardless of letter case) and concatenating the binary values for each letter. The formula given below results in this value:

$$ch_0 * 32^{n-1} + ch_1 * 32^{n-2} + \cdots + ch_{n-2} * 32^1 + ch_{n-1} * 32^0$$

where ch_0 is the left-most character of the string and n is the size of the string. For example, the string "cat" has this value:

$$3 * 32^2 + 1 * 32 + 20 = 3124$$

Note that using the result of this calculation on large strings will cause overflow. You should therefore use Horner's method as shown in class to perform the calculation, *and apply the modulo operator after computing each expression in Horner's method*.

To find the "value" of a lower case character in position *i* in a string `str`, you can do this:

```
int asc = str[i] - 96; // ASCII a = 97
```

For the purposes of this assignment, you can assume that all strings will be lower case and will not contain any special characters, numerals, punctuation, or whitespace. i.e. your strings will consist of only the lower case characters 'a' to 'z'. The test driver which the TA uses for grading your submission will also only test using these lower case characters.

Linked List Class

Your hash table should deal with collisions using separate chaining implemented as an array of singly-linked lists. This means you must also implement a singly-linked list class (called `SLinkedList`). The linked lists' public methods are driven by the public methods of the `HashTable` class. You may refer to the postfix sample code posted to the course website earlier in the semester as a starting point for this class. Please refer to the provided slinkedlist.h header file for details about the linked list functional requirements.

Collisions

Dealing with collisions using separate chaining is straightforward since they are handled by the *structure* of the hash table (which is more complex than a hash table that implements open addressing since in this case it consists of an array of singly-linked lists).

It is important to determine which class is responsible for each part of the process of inserting, finding, or removing values. As an example, when inserting a string into the hash table, these steps should be performed:

1. Calculate which index the string maps to, by running the (private) hash function on the string
2. Determine whether or not the string has already been stored in the linked list at the hashed index
3. Either insert the string and return true, or return false without inserting the string

It is the responsibility of the `HashTable` class to compute the hash index (step 1). Steps 2 and 3 require in-depth knowledge of the contents of a singly-linked list and are therefore the responsibility of the `SLinkedList` class and should thus be handled by `SLinkedList` methods. The hash table's `Insert` method can therefore call the `Contains` and `Insert` method of the linked list at the appropriate index to perform the insertion and determine the result.

Vectors

A vector is a STL (Standard Template Library) template class that can be found in the `<vector>` header file. Below, you will find pretty much everything you need to know about vectors for the purposes of this assignment.

Declaring a Vector
Since vectors are a template class, then when a vector is declared, the type to be stored in it should be specified

```
vector<string> v; // creates a new vector of strings called v
```

Accessing Vector Elements
Individual elements of a vector can be accessed using indices like an array, or by using the vector's `at` method.

```
string s1 = v[2]; // assigns the third element of v to s1
string s2 = v.at(3); // assigns the fourth element of v to s2
```

Inserting Values in a Vector
There are numerous methods for inserting values in a vector. The simplest way is to insert a value at the *end* of a vector by calling its `push_back` method. Note that vectors increase size automatically as necessary.

```
v.push_back("ape"); // inserts the string ape at the end of v
```

Warning: inserting a value into a given position in a vector using its `insert` method is less efficient than using `push_back`.

Finding the Size of a Vector
The number of items stored in a vector can be found using its size method.

```
for (int i = 0; i < v.size(); i++) { ... } // iterates through the
                                                 // contents of v
```

Copying a Vector into Another Vector
The `insert` method can be used to insert the entire contents of a vector at the end of another vector. In the example given below, the contents of `v2` are inserted at the end of `v1`.

```
v1.insert(v1.end(), v2.begin(), v2.end());
```

Testing your `SLinkedList` and `HashTable` classes

Before you submit your assignment you should compile and run it with the a5simpledriver.cpp file from the course website. If you are unable to compile and run this program it is very likely that we will not be able to compile and run your submission and may not mark your assignment.

The test function is ***not** in any way intended to be a comprehensive test of your class* so successfully running the program straight from the website without modification does ***not** mean that you will get full marks for the assignment*. It is recommended to add your own code to specifically test `SLinkedList` and `HashTable` objects in detail.

Deliverables

Submit a ZIP file to CourSys, with the following contents:

- Title page with names and student information of contributing group members
- `slinkedlist.cpp`
- `hashtableprivate.h`
- `hashtable.cpp`

If you are unable to complete any of the methods, make sure that any calls to that method will still compile and run by writing a stub for that method that returns a default value of the appropriate type.