CSCI 1321 (Principles of Algorithm Design II), Fall 2001

Homework X[1]

Assigned: December 7, 2001.

Due: December 17, 2001, at 5pm. *Not accepted late.*

Credit: 20 extra-credit points.

> *Note:* The HTML version of this document may contain hyperlinks. In this version, hyperlinks are represented by showing both the link text, formatted like this, and the full URL as a footnote.

Contents

1 Before getting started **1**
- 1.1 Reading . . . 1
- 1.2 C++ programming tip . . . 1

2 Problem statement **2**

3 Details **2**
- 3.1 Hash tables . . . 2
- 3.2 Operations . . . 2
- 3.3 A hash function . . . 3
- 3.4 Resizing the hash table . . . 4
- 3.5 Suggestions for implementation . . . 4

4 What files do I need? **4**

5 What to turn in **5**

1 Before getting started

1.1 Reading

Read chapter 12, sections 2 and 3.

1.2 C++ programming tip

Default function arguments can specify values for function arguments that usually have the same value. For example, the function `int foo(int x, int y = 3)` can be called using either `foo(4,5)` or `foo(4)`. In the former case, the parameter values are `x = 4` and `y = 5`. In the latter case, `y`'s parameter value is `3`. See also the textbook pp. 60–62.

[1] © 2001 Jeffrey D. Oldham (`oldham@cs.stanford.edu`) and Berna L. Massingill (`bmassing@cs.trinity.edu`). All rights reserved. This document may not be redistributed in any form without the express permission of at least one of the authors.

2 Problem statement

You are to implement a simple class for hash tables, similar in functionality to the STL hash table[2] You may solve the problem in any way you like, except that you may not use the STL `map` or `hash_map` classes. I strongly recommend making use of one of the data structures we have studied and/or implemented in this course.

3 Details

3.1 Hash tables

A *hash table* is a data structure supporting insertion, removal, and querying of elements in expected constant time by using a hash function. A *hash function* converts an element into a number specifying where the element should be stored. You are to implement an *open-chained* hash table using strings as keys. As discussed in class, such a table consists (conceptually) of an array of lists, with one array element for each possible output of the hash function. (Notice that while we can describe this table as an array of lists, we can actually implement it using any convenient data structure, such as the STL `vector` class.) For simplicity, your hash table need only store keys, rather than key-value pairs as hash tables usually do. (Such a table could still be useful; for example, we could use it to store a mathematical set in such a way that it would be quick to add elements, remove elements, or determine whether a particular element is in the set.)

3.2 Operations

The `hashTable` class should support the operations listed in the following table. These operations are similar to but not identical to those provided by the STL hash tables.

Prototype	**Example use**	**Explanation**
`hashTable(void)`	`hashTable h;`	create a hash table with no entries.
`void insert(const string & key);`	`h.insert("hello");`	add the given key to table if not already present.
`bool query(const string & key) const;`	`bool b = h.query("goodbye");`	return true if and only if `key` is in table.
`void remove(const string & key);`	`h.remove("whatever")`	remove `key` from table, if present.
`friend ostream& operator<<(ostream & out, const hashTable & h);`	`cout << h;`	print hash table's contents, one per line, in no specific order.
`void printInfo(ostream & out) const`	`printInfo(cout);`	print the table plus internal info; see description below.

Notes:

- The `insert()` function ensures that the given key is in the hash table. If a matching key is already in the hash table, the given key is not inserted. Thus, all keys in the hash table are always unique.

[2] `http://www.sgi.com/tech/stl/hash_set.html`

- The `remove()` function ensures that the given key is not in the table. Notice that since keys are unique, we need not worry about removing multiple occurrences of a key, and if the key is not in the table, this function should do nothing.

- The `printInfo()` function should print the following information: the size of the table's main array/vector, the number of keys currently stored in the table, and a list of keys stored in the table together with their hash codes (where a key's *hash code* is the value produced by applying the hash function to the key). The purpose of this function is to provide, in an easy-to-implement format, information about the internal arrangement of the table, in particular how often it happens that two keys have the same hash code — something we hope does not happen often. Here is example output, for a table containing keys "data", "which", "bye", and "hello":

```
Table size:  16
Number of keys stored:  4
Contents:
data (4)
which (7)
bye (12)
hello (15)
```

- The output operator should print only a list of keys stored in the table, in some reasonable format. Here is example output for a table containing keys "data", "which", "bye", and "hello":

```
data
which
bye
hello
```

3.3 A hash function

For the hash table described above, we need a hash function that converts a string to a position in the hash table's main array or vector. There are many possibilities for defining such a function; this section describes (and the starter code implements) one that has proved to work well in practice.

Conceptually, the hash function you will use is a composition of three simpler functions:

1. The first function interprets the string as a base-256 number using the string's ASCII values, with the digits in reverse order. For example, the string "Bob" would be given value $98 + 256 * (111 + 256 * 66)$. (The values 66, 111, and 98 correspond to characters "B", "o", and "b" respectively.)

2. The second function takes the resulting (very large) integer and converts it to a fraction in a way that spreads out the fractions fairly evenly over the range $0 \leq f < 1$. This is done by first multiplying the integer by an irrational number and then dropping the digits to the left of the decimal point. Donald E. Knuth recommends using the golden ratio $(\sqrt{5} - 1)/2$; see [Knu98, p. 517]. (This reference probably also explains why multiplication by an irrational number produces a good distribution of results, a topic that is interesting but beyond the scope of this course.)

3. The third function takes the resulting fraction and maps it to an integer in the range $\{0, 1, 2, \ldots, M-1\}$, where M is the length of the hash table's main array or vector, by multiplying by M and dropping the fractional part of the result.

In practice, converting the string into a base-256 number can yield a very large number that can overflow the largest integer that most computers can store, so we will combine the first two steps: After converting each string's character into its ASCII code, we multiply by the golden ratio. After each multiplication or addition, we drop the integral portion of the result. This will ensure that the numbers will always have reasonable size.

3.4 Resizing the hash table

A good hash function spreads out a hash table's contents, but if any significant number of table positions have too many entries, using the hash table will take too much time. Conversely, if only a few table positions have any entries at all, the table is probably bigger than it needs to be. Thus, if the hash table becomes too full, we should make the table bigger; if it becomes too close to empty, we should make it smaller. Specifically, if M is the size of the main array/vector, and n is the number of actual entries (keys stored), we should do the following: If $2n > M$, double the size of the main array/vector. If $n < M/8$, halve the size of the main array/vector.

When resizing the hash table, every key in the hash table must be rehashed, since the hash function's values depend on the table's size. One implementation strategy is to insert the keys into a newly created vector. Another strategy is to always have two vectors in the hash table object, plus a variable indicating which vector currently contains the keys. To resize, one can then clear the other vector, resize it to the new size, insert into it all the keys from the vector currently containing the keys, and mark it as the vector currently containing the keys.

3.5 Suggestions for implementation

You will likely find it easiest to implement your hash table class using STL containers and functions. I suggest representing the table as a `vector` of `vector`s of `string`s. Useful `vector` functions include the following:

- `resize(`*`size`*`)`, which changes the number of items a vector can store,
- `erase(`*`vector::iterator`*`)`, which removes the item pointed to by the iterator, and
- `clear()`, which removes all the items in the vector.

Note that you can use the STL `find` function[3] to search for an element in a `vector`; this function returns an iterator and so could be used in conjunction with `erase()` above.

You may want to initially implement your class without resizing, and add that functionality later.

4 What files do I need?

I have provided a skeleton for your hash table class in file `hashTable.h`[4], including do-nothing implementations of all the required functions and an almost-complete implementation of the hash function described above. Places where you need to add or change code are indicated by comments

[3] `http://www.sgi.com/tech/stl/find.html`

[4] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW0X/Problems/hashTable.h`

of the form `ADD YOUR CODE HERE` or `REPLACE THE FOLLOWING LINE WITH YOUR CODE`. I have also provided a program you can use to test your hash table class, `test-hashTable.cpp`[5]. How the test program works should be fairly obvious if you compile it and execute it; it will compile with the skeleton hash table class, though it doesn't do anything very interesting.

5 What to turn in

Submit your source code (`hashTable.h` only) as described in the Guidelines for Programming Assignments[6]. For this assignment use a subject line of "cs1321 hw0x".

References

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, second edition, 1998.

[5] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW0X/Problems/test-hashTable.cpp`
[6] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Notes/pgmguidelines/index.html`