lab_hash Hellish Hash Tables

Due: Sunday, November 15 at 11:59 PM
Doxygen for lab_hash

Assignment Description

In this lab you will be implementing functions on hash tables with two different collision resolution strategies — separate chaining and linear probing. These hash tables serve an implementation of the dictionary abstract data type. In the second part of the lab you will use the dictionary to solve some problems.

Note that there are a LOT of files in this lab — don't get too intimidated as you won't be modifying the vast majority of them.

Notes About `list` Iterators

When you are working with the Separate Chaining Hash Table, you will need to iterate over the linked list of a given bucket. Since the hash tables are templatized, however, this causes us a slight headache syntactically in C++. To define a `list` iterator on a given bucket, you will need to declare it as follows:

```C++
typename list< pair<K,V> >::iterator it = table[i].begin();
```

Alternatively you can just use `auto` and have the compiler worry about the type:

```C++
auto it = table[i].begin()
```

Also note that if you do not insert your data at the head of the bucket, you may end up with your results being in a different order than ours. This is fine, but if you want to `diff` with the solution output (in the `soln`) folder, you will want to ensure that you always insert your data at the head of the `list` in each respective bucket.

If you use the `list::erase()` function, be advised that if you erase the element pointed to by an iterator that the parameter iterator is no longer valid. For instance:

```C++
auto it = table[i].begin();
table[i].erase(it);
it++;
```

is invalid because `it` is invalidated after the call to `erase()`. **So, if you are looping with an iterator, remember a `break` statement after you call `erase()`!**

Checking Out The Code

You can get the code as you usually do: running `svn up` in your `cs225` directory.

There is also a separate `data` directory which must be downloaded separately. Do so by running

```
make data
```

from your `lab_hash` directory.

Here is the list of all of the files in this lab. (YOU WILL BE FINE. IT IS OK!) There are a **lot** of files, but you only will need to modify five of them. They have been marked in the above table, and the spec guides you through which functions you have to implement. Don't panic!

In your directory there should be two folders as well: `data` and `soln`. The `data` folder contains text files used in the applications later, and the `soln` folder contains sample output to `diff` your code's output against.

If you want to speed up compile time on a `make all`, try using `make -jN all`, where `N` is an integer (say 4 for instance).

`remove` and `resizeTable` on a Separate Chaining Hash Table

Open your `schashtable.cpp`. In this file, the above two functions have not been implemented—your job is to implement them.

`remove`

- Given a key, remove it from the hash table.
- If the given key is not in the hash table, do nothing.
- You may find the Doxygen for `remove` helpful.

`resizeTable`

- This is called when the load factor for our table is ≥ 0.7.
- It should resize the internal array for the hash table. Use the return value of `findPrime` with a parameter of double the current size to set the size. See other calls to `resize` for reference.
- Here is the Doxygen for `resizeTable`.

`insert` on a Linear Probing Hash Table

Open your `lphashtable.cpp`. In this file, you will be implementing the `insert` function.

- `insert`, given a `key` and a `value`, should insert the `(key, value)` pair into the hash table.
- Remember the collision handling strategy for linear probing! (To maintain compatibility with our outputs, you should probe by moving forwards through the internal array, not backwards).
- You do not need to concern yourself with duplicate keys. When in client code and using our hash tables, the proper procedure for updating a key is to first remove the key, then re-insert the key with the new data value.
- Here is the Doxygen for `insert`.

You MUST handle collisions in your `insert` function, or your hash table will be broken!

Test Your Code Using `charcount`

You can read the Doxygen for the `CharFreq` class here. It has already been implemented for you.

Type the following into the terminal:

```
TERMINAL
make charcount
```

This will make the `charcount` executable. This file takes two arguments: the first is the file to analyse, and the second is the frequency for which to return characters. For example, a run on `data/sherlock_holmes.txt` for characters with frequency greater or equal to 10000 might look like:

```
TERMINAL
$ ./charcount data/sherlock_holmes.txt 10000 schash
Finding chars in data/sherlock_holmes.txt with frequency >= 10000 usin
54459   e
40038   t
35944   a
34570   o
30921   i
29449   h
29436   n
27805   s
25298   r
18981   d
17548   l
13495   u
12060   m
11432   w
10847   c
```

Note

You should verify that your solution outputs the right values for both kinds of hash tables!! You can test each one by changing the third command line argument to `"schash"` or `"lphash"`, respectively.

Implement the `WordFreq` Class

Now we will write our own application for a hash table! Open `word_counter.cpp`. You will be writing the `getWords` function. This function, given an integer `threshold`, should return a `vector` of (`string`, `int`) pairs. Each of these pairs is a string and its frequency in the file. Your vector should only contain those pairs whose frequencies are greater than or equal to threshold. Look at `char_counter.cpp` if you are stuck. You may find the `TextFile` class defined in `textfile.h` helpful.

Helpful Doxygen links: Doxygen for `WordFreq`, Doxygen for `TextFile`.

Test Using the `wordcount` Executable

When you're done with the above, type the following into your terminal:

```
TERMINAL
make wordcount
```

This will build the `wordcount` executable, which uses the `WordFreq` class to determine word frequencies in a file. It, like `charcount`, takes two arguments: the first is the path to the file to be analyzed, and the second is the threshold for which to grab words. If a word occurs >= `threshold` times, it will be printed to the console.

For example, a run of `wordcount` on `data/metamorphoses.txt` with a frequency 1000 might look like:

```
TERMINAL
$ ./wordcount data/metamorphoses.txt 1000 lchash
Finding words in data/metamorphoses.txt with frequency >= 1000 using L
10473   the
5753    of
4739    and
3078    to
2246    a
1989    in
1706    was
1572    his
1548    that
1500    her
1456    with
1306    is
1241    he
```

```
1185    by
```

You should verify your solution works with both kinds of hash tables.

Implement the `AnagramFinder` Class

You can read the Doxygen for `AnagramFinder` here.

Finding anagrams is another clever use of hash tables. Open `anagram_finder.cpp`. You will be writing the `checkWord` function, which simply returns a boolean value. It should return `true` if the string `test` is an anagram of `word`, and `false` otherwise.

An anagram of a given word is a word which has the same letters. For example, “retinas” and “stainer” are anagrams of each other. Think carefully about what it means for a word to be an anagram of another. Is there something we can use a hash table to keep track of? Does counting letter frequency help you?

Test Using the `anagramtest` Executable

When you’re done with the above, type the following into your terminal:

```
make anagramtest
```

This will make the `anagramtest` executable, which you can use to test your `AnagramFinder` class. It can be run with or without arguments. Without arguments will test a very simplistic example—with arguments is more interesting.

`anagramtest` can also take two arguments: the first is the path to the file to be analyzed, and the second is the word to find anagrams for. For example, a run of `anagramtest` on `data/words.txt` looking for anagrams of `retinas` might look like:

```
$ ./anagramtest data/words.txt retinas schash
Checking file data/words.txt for anagrams of retinas using SCHashTable
anestri is an anagram of retinas
asterin is an anagram of retinas
eranist is an anagram of retinas
nastier is an anagram of retinas
ratines is an anagram of retinas
resiant is an anagram of retinas
restain is an anagram of retinas
retains is an anagram of retinas
retinas is an anagram of retinas
retsina is an anagram of retinas
sainter is an anagram of retinas
stainer is an anagram of retinas
starnie is an anagram of retinas
stearin is an anagram of retinas
```

You should verify your solution works with both kinds of hash tables to verify their correctness as well.

Implement the `LogfileParser` Class

This task is a bit more abstract, but is highly applicable for companies. Given a logfile with the following content:

```
[customer_name]:    url    date
```

We want to answer the following questions (quickly!):

- What unique URLs have been visited?
- Given a customer and a product, have they visited that product? If so, when was the most recent time they did so?

In order to do so, you will have to parse the logfile and do something with hash tables. The parsing occurs in the constructor for the `LogfileParser` and the querying happens in the other two functions. You should design your constructor so that you can run the two helper functions as quickly as possible (that is, I should be able to do n queries on your `LogfileParser` in $\mathcal{O}(1)$ time per query after the constructor has finished).

Read the Doxygen for `LogfileParser` here.

Hint

You should not need to define a new hash function for this problem! Think about how to construct a key based on what data you have.

You are safe to assume that customers have distinct names, and that URLs always follow the pattern `/product/XX/` where `XX` is an integer. Products (and thus URLs) have distinct integer IDs.

Test Your `LogfileParser` with `lfparse`

When you are done with the above, type the following into your terminal:

```
make lfparse
```

TERMINAL

This will build the `lfparse` executable, which will test your `LogfileParser` class. It takes a single argument: the data file to be processed. For example, a run of `lfparse` on `data/log2.txt` might look like:

```
$ ./lfparse data/log2.txt
Parsing logfile: data/log2.txt...
```

TERMINAL

```
Number of unique URLs: 10
Printing unique URLs:
        /product/3/
        /product/2/
        /product/9/
        /product/6/
        /product/7/
        /product/4/
        /product/5/
        /product/1/
        /product/8/
        /product/0/
Running sample visited queries...
        chase visited /product/0/ on Mon Apr 11 15:05:56 2011
        chase visited /product/1/ on Sat Apr  9 15:54:36 2011
        chase visited /product/2/ on Tue Apr 12 15:52:58 2011
        chase visited /product/3/ on Tue Apr 12 15:48:33 2011
        chase visited /product/4/ on Tue Apr 12 14:21:47 2011
        chase visited /product/5/ on Sat Apr  9 14:30:12 2011
        chase visited /product/6/ on Mon Apr 11 16:07:27 2011
        chase visited /product/7/ on Tue Apr 12 15:20:48 2011
        chase visited /product/8/ on Mon Apr 11 14:31:36 2011
        chase visited /product/9/ on Tue Apr 12 15:04:12 2011
```

You should verify your solution works for both kinds of hash tables to verify their correctness as well.

`genlog` More Logfiles!

If you would like, you can also generate more random logfiles to throw at your `lfparse` executable. To do so, type the following into your terminal:

TERMINAL

```
make genlog
```

`genlog` takes three arguments: `filename`, `products`, and `lines` (in that order).

- `filename` is the path to the output file. **If the output file exists, it will be overwritten!**
- `products` is the number of unique products there are.
- `lines` is the length of the logfile in lines to generate.

Don't accidentally delete your `data/log.txt` or `data/log2.txt`!

Committing Your Code

When you are finished with the lab, be sure to `commit` your code back to subversion—this lab will be autograded.

Grading Information

The following files (and ONLY those files!!) are used for grading this lab:

- `lphashtable.cpp`
- `schashtable.cpp`
- `word_counter.cpp`
- `anagram_finder.cpp`
- `logfile_parser.cpp`

If you modify any other files, they will not be grabbed for grading and you may end up with a "stupid zero."