CS241
Project 5
Hash Tables and Quicksort

In this project, you will write code to implement both a hash table and the quicksort algorithm. These are not particularly hard to implement --- especially because you can get the quicksort code from the book --- so we will explore some useful C++ Standard Template Library classes that will help you implement this project and be useful in the future as well.

The project is based around reading in a text file and counting how often each word occurs (similar to the movie reviews project). However, we will only store a word and its frequency for this project (nothing involving word scores), and we won't use a struct to do it.

Pairs

C++ (as well as other languages) often have a pair class that is commonly used when you need to store two objects of different types in one structure. It comes up frequently with the map ADT (as we are using in this project), to store a relationship between a key and its value.

In C++, you can use the pair class by doing `#include <pair>`. There is reference material at http://en.cppreference.com/w/cpp/utility/pair, but I'll walk you through the basics.

Our pairs will be a `string` (the key) and an `int` (the value), since we are storing an association between a word and the number of times it appears in a document.

Here are some common operations:

- Make a new pair:
  ```
  pair<string, int> mypair("word", 5);  // makes a new pair of "word" and 5
  ```

- Once you have a pair, you can access the elements by using `.first` and `.second`:
  ```
  cout << mypair.first;     // prints word
  cout << mypair.second;    // prints 5
  mypair.second++;          // changes the frequency
  ```

You can think of a pair as a `struct` that always has two pieces that are always named `first` and `second`. It's useful to save time and not have to make a new `struct` every time you want this kind of class.

Lists

C++ has a built-in doubly-linked list class called (unsurprisingly) `list` (`#include <list>`). We will be implementing a hash table using chaining, so we'll use a table that stores linked lists of `pair<string, int>`. The reference material is at http://en.cppreference.com/w/cpp/container/list, but I'll give you the highlights. To introduce what a `list` can do, we'll simplify things and use a `list` of `int`s:

Common operations:

- Make a new, empty list:
  ```
  list<int> mylist;
  ```

- Lists have `push_back` and `push_front`, just like C++ vectors do. They do analogous things: `push_back` is an append operation (adds to the back/tail of the list), and `push_front` is a prepend operation (adds to the front/head of the list).
  ```
  mylist.push_back(5);
  mylist.push_back(7);
  mylist.push_front(3);           // mylist is now 3 → 5 → 7
  ```

- Because `list` is a doubly-linked list, we don't have random access to individual elements, so we can't use square brackets with indices to access elements. Instead, we can use a for loop to iterate through the list:

  ```
  for (int item : mylist) {
    cout << item;
  }
  ```

This is a useful idiom that you will use frequently to iterate over a list. However, what if we want to change items within the list as we're iterating? This won't work:

```
for (int item : mylist) {
  item++;
}
```

The issue is that `item` is a new variable that is just a copy of the actual data from the internal linked list node. However, C++ has a slick way of giving us access directly to the internal data of the list: we can use a *reference variable*:

```
for (int & item : mylist) {
  item++;
}
```

Notice the ampersand in the for loop. This is similar to how pass-by-reference works in functions; it tells C++ that `item` should not be a copy of the internal integer from the linked list node, it should refer directly to the contents of the list. That way, when we do a ++ operation on `item`, we're directly modifying the contents of the list, rather than a copy.

- Get the number of elements in a list: `mylist.size()`

In this project, as we mentioned above, we will need to store lists of pairs, however. So here's some sample code that covers the common operations:

```
list<pair<string, int>> mylist;         // make a linked list that holds pairs of strings and ints
pair<string, int> mypair("word", 5);    // make a new pair
mylist.push_back(mypair);               // put the pair at the end of the list
mylist.push_back(pair<string, int>("word2", 10));          // another way to append to the list

for (pair<string, int> & p : mylist)    // iterate over the list with a reference variable
{
  cout << p.first << p.second;
  if (p.first == "word")                // look for a specific word...
    p.second = 15;                      // ...and change its frequency
}
```

How this project should work

The skeleton code does a lot of the grunt work for you of opening files and splitting lines, etc. Fill in the parts directed. Here's the order I suggest:

1. Do the vector stuff first, because it's easier. Write code to insert/update frequencies in the freqVec vector first. This vector is initially unsorted, so put new pairs at the back of the vector (use push_back) like normal.

2. Then write a function to print the vector contents --- follow the sample output for how it should look.

3. Write a function to quicksort the vector.

4. Now write a binary search function to search the vector. This is needed at the end of the program when you're looking for the frequency of a word. Don't use linear search!

5. Now switch to the hash table. Write the Hashtable.cpp functions, starting with the hash function itself (because you need to call hash() inside of get()/put()). Notice how the hash table itself is a vector<list<pair<string, int>>>. That is a vector that stores linked lists, where each linked list holds pairs of strings and ints. This should make sense. The vector is the hash table itself. Each entry in the vector is a linked list (because we're using chaining). Each linked list holds pairs of strings and ints where each string is a word and the int is the frequency that the word appears in the text file.

   The hash function is described in the .cpp file. The put function should insert or update a frequency in the table, and the get function should retrieve a frequency given a word, or return -1 if the word isn't in the table.

6. After you finish the hash table you may want to write some quick test cases in main.

7. Then go add in the code in main to insert words and frequencies into the freqHt table correctly.

8. Done!

Other information

- As usual, your output should match mine as closely as possible. We're all using the same hash function so the order of everything should match 100%.

- If you get errors like this:

```
./Hashtable.h:22:33: error: a space is required between consecutive right angle brackets
(use '> >')
```

-
  ```
      vector<list<pair<string, int>>> table;  // the table itself
  ```

  that means you need to put spaces between the angle brackets. The reason for this is without the spaces, C++ thinks that two angle brackets together are one symbol (like in `cin >> x`). Older versions of C++ require this, newer versions don't. Consider getting a newer version of your C++ compiler!

- The constructor for the hash table doesn't need to do anything except set the size variable, and (here's the important part) initialize the table itself to have that number of slots in it. You can do this with the resize function. So your constructor should set size=sz, then call table.resize() to set up the table with the appropriate number of slots.

- The get & put functions in the hash table should NOT iterate over the table. The whole point of a hash table is you get constant time access to the correct slot in the table, calculated from the hash function. So the first thing you should do inside get & put is call hash on the key value; this will give you the slot in the table that you should be accessing.

- On the other hand, print() is allowed to iterate over the table, b/c it needs to print the whole thing.