Assignment #6

Due date: Friday, 1 April 2016, 5:59 pm (no foolin')

- *Write your solutions in the dialect of C++ used in class.*
- *Store your set of functions in a set of files named `a6p1.cc`, `a6p2.cc`, ...*
- *You may define your own `main` function, but we will use our own `main` to test your code.*

Part I

This part is pretty easy. Write a function called `scrabbleValue` that takes a `string` and returns an `int` that is the Scrabble value of the word. The Scrabble value is calculated by summing the values of each of the letters it contains, according to the table below. Case of the letters is not significant: `"uxorious"` is worth the same as `"uXorIUOS"`. If the string contains a non-letter as a character, just ignore it.

You should overload the `scrabbleValue` function to have a second version that takes a single `char` and returns its value (this will be called by the other one as a helper function). You might also want to use the `tolower` function that takes a `char` and returns its lower case equivalent `char` if you pass it an upper case letter (otherwise it gives you the same `char` back); you will need to `#include<cctype>` to be able to use `tolower`.

Letter	*Scrabble value*	*Letter*	*Scrabble value*
EAIONRTLSU	1	K	5
DG	2	JX	8
BCMP	3	QZ	10
FHVWY	4		

Part II

This part is pretty easy too. We are going to explore hash tables and inheritance, and we've already implemented most of hash table class (with `string`s as the element type) in the file `a6p2.cc`, which you can download from the web page. It's pretty much exactly the implementation shown in class except for a couple of things. First, as in class, we didn't implement the `remove` method; you are to provide the implementation for this (if asked to remove something that isn't in the table, just do nothing and return.) Second, you may notice that in the `HashTable` class, the `hash` function itself is "pure virtual" (i.e., abstract, not implemented, defined to be "`=0`"), thus the `HashTable` class itself is abstract. We did this to provide a way of having different child classes that each implement different hash functions.

To complete this question, implement a second class called `SimpleHashTable` that inherits from `HashTable`. It should define and declare two `public` constructors that echo the ones in `HashTable`, plus a `public` destructor, and a `private` implementation of `hash` that has the same signature as the one declared in `HashTable`. The destructor should have a null body, as the hard work is already being done in `HashTable`. For `hash`, use the simple algorithm I showed in class, where you sum the ASCII values (i.e., as `char`s) for the string, and then take the result mod `K` (where `K` is the number of buckets, i.e., the table size). Yeah, this hash function has a terrible "spread", we know. You're going to find a better one later on.

You may notice a couple of other functions I defined in the `HashTable` class: `print` and `report`. The first might be useful to you when you're debugging. The second is how we will evaluate the effectiveness of your hash function; have a look through it.

To populate the table with data, you can try out your own (small) test datasets, and trace through the results to make sure your program does what you think it does. However, we've also prepared a monster word list for you, which we will be testing your program against. North American Scrabble players use an official list of words known as TWL, for Tournament Word List. This list can be found on the web easily, and many web word games uses this as their dictionary.[1] There are 178,368

[1] We used the version of the list with the officially designated "rude" words removed; obviously we haven't had checked the results, so apologies in advance if you see any word you find offensive.

words in our list, and you can download it from the course webpage as a file named `twl-words.txt` (it's almost 2MB of plain text).

To summarize, all that we require you to do for this part is to implement `HashTable::remove()`, and to declare and define the `SimpleHashTable` class as described above. Experimenting with different table sizes, different word lists, etc. is up to you. We will write our own main program that uses your code.

Note that you will need to put `#include<algorithm>` at the beginning of your solution, as my implementation of `report` uses the `sort` procedure that is defined there.

Part III

This part is harder and requires some thinking and some experimentation, although the amount of code you will have to write is small. Note that Part IV can be completed without doing this part, in case you run out of time. Obviously, tho, you will lose marks if you decide not to do it.

In this part, you are to create a second child class of `HashTable` called `SmartHashTable`. This will be very similar to its inheritance sibling `SimpleHashTable`, except that you are to find and implement a much better hash function!

What does "better" mean? Here's a concrete target to aim for: If you have 100,000 buckets and use the TWL world list, then aim for results that are at least as good as this: no more than 25% empty buckets, maximum overflow list size of no more than 25, and a median overflow list size of no more than 10. (The method `HashTable::report()` will tell you this information.) If you can't get this good, do the best you can. You are free to look around the web for ideas. You are not allowed to share these ideas with each other, but you are allowed to discuss how good your results are with your friends. Don't forget that we use plagiarism detection tools on your code!

To be clear, for this part you are handing in a file called `a6p3.cc` that includes your definition of `SmartHashTable` together with the code for `SimpleHashTable` and `HashTable`.

Part IV

At first, this question may seem harder than it really is. However, most of the hard work has already been done by you (in parts I-III), me (in the two functions `pset` and `addChar` that you can download from the webpage), and the C++ STL. It will require you to look at some code and think about it a bit.

Your job is to write a program (like you did in assignment #1, way back at the beginning of term) that takes one command-line argument, call it `wordlistFileName`, and then reads a sequence of tokens from the standard input (`cin`) until `eof`. If the user fails to provide a single command line argument, you should print the following error message and abort:

```
Error, no word list file name provided.
```

Your program should first check that `wordlistFileName` can be opened as an input file stream (like we did at the beginning of term); if it can't, print the following error message and abort:

```
Error, couldn't open word list file.
```

Assuming all goes well, read in the data from the word list, for which you should use the TWL list from the previous parts. You may assume that the input has one word per line, which you read in and add to your `SmartHashTable` (or `SimpleHashTable` if you skipped Part III). Once you've built the table, you should then read in one token at a time from the standard input (`cin`). For each token, you are to find the maximum valued Scrabble word that can be made from those letters.[2]

Now that sounds easy, until you realize that (1) you need to compute the power set of those letters and (2) for each member of power set, you need to try all possible combinations of those letters. However, we've done most of the work for you already.

Have a look at the functions `powerset` and `addChar` I wrote that can be found in the handout file `a6p4.cc` on the web page. If you pass a character string to `powerset`, you will get back a vector of the power set of those letters. Actually, it's a power multi-set, as we do not delete duplicates. So, if you pass in `"aab"` to `powerset`, you will get back a vector whose elements (in no particular order) are `"aab"`, `"ab"`, `"aa"`, `"a"`, `"ab"`, `"a"`, `"b"`, and `""`. That's step (1) done.

Now we want to look at each element in the vector, consider all of the possible permutations of those letters, look up each one of the hash table, and for those that we find in there, we compute the Scrabble value. To do this, we'll first take each vector element in turn. Suppose we're looking at element `i` and we've saved it to a temporary variable named `s`. There is a

[2]Don't worry about the fact that in a standard Scrabble set that there is only one Z and Q, for example. And don't worry about the blank tiles.

really nifty algorithm in the STL that called `next_permutation` that can take a pair of iterators and return successively each possible permutation of the values. Since `s` is a `string`, and strings provide `begin()/end()` iterators, this makes the job easy. However, the `next_permutation` function assumes that the input has already been sorted; lucky for us, the STL also provides a sort procedure that works by passing in a `begin()/end()` iterator pair. Note that you need to `#include<algorithm>` to use `sort` and `next_permutation`.

So try this out:

```
string s = "aab";
// Well, s will be actually be one of the element of the vector
// that was returned from your call to powerset
sort (s.begin(), s.end());
do {
    // do cool stuff with the newly re-arranged s
} while (next_permutation (s.begin(), s.end()));
```

At this point, we're now able to find all possible combinations of all possible subsets of a set of characters. For the given input word, generate them all one at a time. For each one, check if it's in the hash table you set up from the word list. If it is, compute its Scrabble score. If it's bigger than the current maximum for this input word, remember it and the Scrabble value. For each word you read in, print the highest scoring word and the Scrabble value (in the case of a tie, just print one of the words).

Suppose that you had only three tokens in your input stream, `aba` and `kuqijz`, `zz`. Then your output should match the below precisely (assume there are no leading or trailing spaces, and that there is a single space after the colon):

```
aba: aba has score of 5
kuqijz: quiz has score of 22
zz: no matches
```

Congratulations, you now have all of the important skills to write a popular word game for Facebook! I want 10% of whatever you make![3]

[3] Yeah, right.