Assignment #6

Due dates:

Tests (for Parts 1, 2, and 3): Monday, 1 April 2024, 8:00 pm
Code (for Parts 1, 2, and 3): Monday, 8 April 2024, 8:00 pm

- *For all programming questions below, write your solutions in the dialect of C++ used in class. You may use the following libraries, and no others:* `iostream`*,* `fstream`*,* `iomanip`*,* `cctype`*,* `string`*,* `vector`*,* `algorithm`*,* `cassert`*.*
- *For this assignment you have been provided with around a dozen* `.cc` *and* `.h` *files. There are only three files that you shouldn't modify. They are* `HashTable.h`*,* `HashTable.h` *and* `a6p3.h`*.*
- *You are free to add your own helper methods and fields to the* `SimpleHashTable` *and* `SmartHashTable` *classes. Just make sure that your changes are compatible with Marmoset and the* `HashTable` *class.*
- *For part 2, the provided test cases already achieve 100% on our solution. There is still a* `a6p2t` *project on Marmoset which you can use to check the correctness of your tests, but it is not worth any marks.*
- *You need to write your unit tests using GTest framework. Give your tests some meaningful names to understand where you should look in the case of test failures. You could use the Given-When-Then approach as a guide for unit test naming.*

Part I: `HashTable` - 30 %

Throughout this assignment we are going to be working with an abstract base class called `HashTable`. The `HashTable` class is abstract because it does not provide an implementation of its hash function. That's a job for its children. We've provided most of the implementation for you, but your first task will be to finish up the bits we skipped.

The `HashTable` class implements open hashing with chaining (check out page 154 of the slides `GraphsTreesHashTables` on Learn for a refresher). Besides the pure virtual `HashTable::hash` method, it's basically just an OOP version of the implementation from lectures. You're only going to have to make two changes:

1. First, implement `HashTable::remove()` (we skipped over this in class, but it's not hard).
2. Second, we want to read in `string`s from a text file later on, so we'd like you to implement an overload of `HashTable::insert()` which takes in an object of type `ifstream` (instead of a `string`) and adds its contents to the `HashTable`.

The precise requirements for these methods are given below.

Next, we want to be able to test our `HashTable` class to make sure that everything's working properly. Since `HashTable` is abstract, this means that we're going to have to implement a concrete child class that inherits from it (remember, you can't construct an instance of an abstract base class).

Since for now we just want to test our `HashTable` class, we aren't going to worry about hashing efficiency (that's a problem for part 2). Instead, we're just going to implement a class called `SimpleHashTable` which uses the simple (but inefficient) string hashing algorithm that returns the sum of the ASCII values of all the characters in a given `string` mod K (where K is the number of buckets, *i.e.,* the table size).

Here's the precise requirements for the methods you'll be implementing in this part:

- `void HashTable::remove(string word)` — Removes the given word from the `HashTable` (if it exists). If the word doesn't exist, then do nothing; do *not* trip an assertion.

- `void HashTable::insert(ifstream& file)` — Reads in `string` tokens from the given file until it is empty, and inserts those tokens into the `HashTable`. You may assume the given file has exactly one `string` token per line. Tokens may contain uppercase, lowercase, and non-alphabetical characters, but they will not contain whitespace characters. Taking tokenized input using `ifstream::operator>>` should be sufficient, but you may use `getline` if you wish. The contents of your `assgt6/InputFiles` directory will be uploaded to Marmoset when you submit your tests so that you can test this method.
- `SimpleHashTable::SimpleHashTable()` — Constructs a `SimpleHashTable` with a `HashTable::table` of size `HashTable::DefaultSize`.
- `SimpleHashTable::SimpleHashTable(int k)` — Constructs a `SimpleHashTable` with a `HashTable::table` of size `k`.
- `SimpleHashTable::~SimpleHashTable()` — Cleans up any heap-allocated memory associated with the given `SimpleHashTable`. (Hint: this one should be *really* simple.)
- `int SimpleHashTable::hash(string key) const` — Returns the sum of the ASCII values of the characters in the given `string` mod the size of `HashTable::table` (*i.e.,* the return value should be strictly greater than -1 and strictly less than the size of `HashTable::table`).

Part II: `SmartHashTable` - 20 %

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

In this part, you are to implement 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. Given 100,000 buckets and the contents of the provided file `twl-words.txt` (which contains 178,368 words), your hash function should be at least as good as this:

- no more than 25% empty buckets (that is, no more than 25,000 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()` which we have provided will tell you all this information and more.

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! If you use a hash function that you found on the web, put the URL of where you found it into your code as a comment.

Here are the specifics for the methods you have to implement in this part:

- `SmartHashTable::SmartHashTable()` — Constructs a `SmartHashTable` with a `HashTable::table` of size `HashTable::DefaultSize`.
- `SmartHashTable::SmartHashTable(int k)` — Constructs a `SmartHashTable` with a `HashTable::table` of size `k`.
- `SmartHashTable::~SmartHashTable()` — Cleans up any heap-allocated memory associated with the given `SmartHashTable` (hint: this one should be *really* simple).
- `int SmartHashTable::hash(string key) const` — Returns an integer strictly greater than -1 and strictly less than the size of `HashTable::table`. This hash function must meet all the criteria listed above.

Part III: Fun With Words - 50 %

Now that we've got our `HashTable` working, for your final assignment problem of this course let's have some fun with it. First, you're going to write two overloads for a function called `scrabbleValue`: one that takes a `string`, and another that takes a `char`. Their return values should be the Scrabble value of the given input. In the case of a `char`, its Scrabble value is given by the table below. In the case of a `string`, its Scrabble value is calculated by summing the Scrabble values of each of the letters it contains. The case of the letters is not significant: `"uxorious"` is worth the same as `"uXorIUOS"`. If `scrabbleValue` is given a non-alphabetical character (or a `string` which contains a non-alphabetical character) then you should print an error message to `stderr` and terminate the program via `exit` or `assert`.

Letter	*Scrabble value*	*Letter*	*Scrabble value*
`A,E,I,L,N,O,R,S,T,U`	1	`K`	5
`D,G`	2	`J,X`	8
`B,C,M,P`	3	`Q,Z`	10
`F,H,V,W,Y`	4		

Hint: You'll probably want to use `scrabbleValue(char c)` as a helper for `scrabbleValue(string s)`.

Now, at first, this last bit may seem harder than it really is. However, most of the hard work has already been done by you (in `scrabbleValue`, as well as parts I & II), by us (in the two functions `powerset` and `addChar` that have been provided to you in `a6p3.cc`), and by the C++ STL.

Your final task is to implement the function `bestScrabbleWord`, which takes a `string`s `letters` and returns the best Scrabble word (that is, the word with the largest Scrabble value) in the file `twl-words.txt` that can be made using those letters. TWL stands for Tournament Word List. This is the official word list used by North American Scrabble players. This list can be found on the web easily, and many web word games use this as their dictionary.[1] There are 178,368 words in this list (it's almost 2MB of plain text).

Now that sounds easy until you realize that (1) you need to compute the power set of the given letters and (2) for each member of the power set, you need to try all possible permutations of those letters. However, as we said, most of this work has already been done for you.

Have a look at the functions `powerset` and `addChar` that are provided to you in your `a6p3.cc` file. 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 in the dictionary, and for those that we find in there, we need to compute the Scrabble value. To do this, we'll first take each powerset vector element in turn. Suppose we're looking at element `i` of our powerset and we've saved it to a temporary `string` variable named `s`. There is a nifty function in the STL called `next_permutation` that can take in two iterators to the front and back of a container and return successively each possible permutation of the values. Since `s` is a `string`, and the `string` class provides `begin()/end()` iterators, this makes the job easy.

Each time it is called, the `next_permutation` function will cycle the given `string` into the lexicographically next-larger permutation (or back to the lexicographically smallest permutation if it is given the largest one). The return value of `next_permutation` is a boolean which will be true if the new permutation is lexicographically larger than the previous one, and false if it isn't (*i.e.,* we looped back to the start).

This means that to cycle through all permutations of a given `string` with `next_permutation`, our `string` needs to start 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:

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

```
string s = "aab";
// s should be one of the elements of the vector
// that was returned from your call to powerset
sort (s.begin(), s.end());
do {
    // Do cool stuff with the newly rearranged 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. Now, for a given input word, we just need to generate them all one at a time, and for each one, check if it's in the file `twl-words.txt`. If it is, we need to compute its Scrabble score. If its score is bigger than the current maximum for this input, remember it and the Scrabble value. But this is easy with our fancy new `HashTable` class!

To summarize, for this part you need to implement the following three functions:

- `int scrabbleValue(char c)` — Takes in a character and, if it is an uppercase or lowercase letter, returns its Scrabble value. If the given character is not an alphabetic character, then print an error message to `stderr` and abort via `exit` or `assert`. See `a6p3Test.cc` for an example of what this error message should look like.
- `int scrabbleValue(string s)` — Takes in a `string` and, if it contains only uppercase and lowercase letters, returns its Scrabble value. If the given `string` contains a non-alphabetic character, then print an error message to `stderr` containing the *first* such character to occur in the `string` and abort via `exit` or `assert`. See `a6p3Test.cc` for an example of what this error message should look like.
- `string bestScrabbleWord(string letters)` — Takes in a `string` of characters and returns the best Scrabble word that can be made with those letters. A Scrabble word is defined as any `string` token which (when converted to lowercase) occurs in the provided file `twl-words.txt`.
  - If there are multiple Scrabble words with the largest Scrabble value, return any of them.
  - The return value should preserve the case of the given letters (hint: think about the easiest way to do this).
  - If the given `string` contains a non-alphabetic character, then print an error message to `stderr` containing the *first* such character to occur in the `string` and abort via `exit` or `assert`. This error message should be identical to the one from `int scrabbleValue(string s)`. See `a6p3Test.cc` for an example of what this error message should look like.

Congratulations, you now have all of the important skills to write a popular word game for Facebook!