CS302

Assignment 7: Anagrams

Description

An anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once. For this assignment, you will be given a set of words (one word per line), and you will need to group all the anagrams together. For example (as shown in the sample output), if you have words `"eat"`, `"tea"`, `"tan"`, `"ate"`, `"nat"`, `"bat"` then you have the following groups of anagrams `"ate"`, `"eat"`, `"tea"` and `"nat"`, `"tan"` and `"bat"`. So you get 3 anagram groups, one of size 3, one of size 2, and one of size 1. You will need to implement your own custom hash map type

```
template <typename t1, typename t2>
class hashMap
{
public:
  hashMap();
  hashMap(const hashMap<t1, t2>&);
  const hashMap& operator=(const hashMap<t1, t2>&);
  ~hashMap();
  t2& operator[](t1);
private:
  int items;
  int tableSize;

  struct tableItem { t1 key; t2 value; };
  tableItem * table1;
  tableItem * table2;

  int h1(char) const;
  int h2(char) const;

  void resize();
};
```

Each member contains/performs the following

- `int items` - represents the amount of elements stored in the hash map structure
- `int tableSize` - represents the size of each table in the hash map structure

- `sruct tableItem{ t1 key; t2 value; }` - represents the data stored in each element of the hash map structure, every element is a (key, value) pair
- `tableItem * table1` - the first hash table
- `tableItem * table2` - the second hash table
- `int hashMap<t1, t2>::h1(char key) const` - is the first hash function, the hash function should be implemented as follows
  - If the key is larger than or equal to 'A', then return (key - 50) mod tableSize else return (key + 20) mod tableSize
- `int hashMap<t1, t2>::h2(char key) const` - is the second hash function, the hash function should be implemented as follows
  - If the key is larger than or equal to 'a', then return (key - 90) mod tableSize else return (key + 50) mod tableSize
- `void hashMap<t1, t2>::resize()` - doubles tableSize and allocates an array of size tableSize to table1 and table2, rehashes all the keys into the larger structure
- `hashMap<t1, t2>::hashMap()` - default constructor that sets items to 0, tableSize to 10, and allocates an array of type tableItem to table1 and table2, and sets the key and value of each element of table1 and table2 to t1() and t2() respectively
- `hashMap<t1, t2>::hashMap(const hashMap<t1, t2>& copy)` - copy constructor, performs a deep copy of the copy object into the `*this` object
- `const hashMap<t1, t2>& hashMap<t1, t2>::operator=(const hashMap<t1, t2>& rhs)` - assignment operator that performs a deep copy of rhs into the `*this` object, remember to first deallocate the `*this` object before performing the deep copy
- `hashMap<t1, t2>::~hashMap()` - destructor that deallocates table1 and table2
- `t2& hashMap<t1, t2>::operator[](t1 key)` - the hash maps search/insert function, this function receives a key (which will be of type `char` for this assignment) and finds an entry with the matching key or a vacant spot and returns the value field of the element in either table1 or table2, the steps for this function is described below
  1. call h1 and h2 and store them into two integer variables x and y
  2. Check if the load factor, if the ratio of items over tableSize is 0.5 or larger then call resize function
  3. Initialize a variable i to 0
  4. If table1 at index (x + i * y) % tableSize is a blank entry, assign key to this element's key, increment items by 1 and return this element's value field
  5. If table1 at index (x + i * y) % tableSize matches the key, then return this element's value field
  6. If table2 at index (y + i * x) % tableSize is a blank entry, assign key to this element's key, increment items by 1 and return this element's value field
  7. If table2 at index (y + i * x) % tableSize matches the key, then return this element's value field
  8. Update i by adding x or y (alternates each time) and go back to step 4

Contents of main

In main, you read an input filename, open a filestream and read the contents of the file and store them into a `vector<string> words` object (This was just to store the words from the file). Each group of anagrams would be stored into a `vector< vector<string> > anagrams` object, where each element of this structure

contains an array of strings that maintains all the words that are anagrams of each other. Each anagram set needs to mark its letters so a `vector< hashMap<char, bool> > lettersUsed` would be used to mark each letter used in each anagram. So if `lettersUsed[0]['a']` contains `true` then every word in anagram `anagrams[0]` all have one letter 'a' in each word. This is how I structured this, but it is up to you how you set up your main. So initially my `anagrams` vector would contain

```
anagrams[0] => {"eat"}
anagrams[1] => {"tea"}
anagrams[2] => {"tan"}
anagrams[3] => {"ate"}
anagrams[4] => {"nat"}
anagrams[5] => {"bat"}
```

And the hash map is initially set up like

```
lettersUsed[0]['e'] = true;
lettersUsed[0]['a'] = true;
lettersUsed[0]['t'] = true;

lettersUsed[1]['t'] = true;
lettersUsed[1]['e'] = true;
lettersUsed[1]['a'] = true;

//I think you get the idea
```

And when my program finishes, I have the following stored in the vector

```
anagrams[0] => {"eat", "tea", "ate"}
anagrams[1] => { }
anagrams[2] => {"tan", "nat"}
anagrams[3] => { }
anagrams[4] => { }
anagrams[5] => {"bat"}
```

Example Output

```
$ g++ main.cpp
$ ./a.out

Filename: words01.txt

Amount of groups: 3
Group 1 contains 3 words
Group 2 contains 2 words
Group 3 contains 1 words

$ ./a.out

Filename: words02.txt

Amount of groups: 6
Group 1 contains 5 words
Group 2 contains 4 words
Group 3 contains 3 words
Group 4 contains 1 words
```

```
Group 5 contains 1 words
Group 6 contains 1 words

$ ./a.out

Filename: words03.txt

Amount of groups: 4
Group 1 contains 6 words
Group 2 contains 4 words
Group 3 contains 3 words
Group 4 contains 1 words

$ ./a.out

Filename: words04.txt

Amount of groups: 18
Group 1 contains 1 words
Group 2 contains 1 words
Group 3 contains 1 words
Group 4 contains 1 words
Group 5 contains 1 words
Group 6 contains 1 words
Group 7 contains 1 words
Group 8 contains 1 words
Group 9 contains 1 words
Group 10 contains 1 words
Group 11 contains 1 words
Group 12 contains 1 words
Group 13 contains 1 words
Group 14 contains 1 words
Group 15 contains 1 words
Group 16 contains 1 words
Group 17 contains 1 words
Group 18 contains 1 words
```

Specifications

- Comment your code and your functions
- Make sure your code is memory leak free
- Output each groups in sorted order from largest to smallest
- There are extra lines in the output to make it easier to parse the output, code grade will skip any whitespace

Submission

Upload your source code and write up to the class site by the deadline, submit main.cpp and hashMap.hpp to code grade

References

- The image used in this write can be found at https://designlooter.com/jack-o-lantern-svg.html