CSCI 152, Performance and Data Structures, Exercise 6

Rules for Exercises:

- You are expected to read the complete exercise, and to make every part of it. 'I did not see this part', will never be a valid excuse for not having made some part of the exercise.
- Submitted code will be checked for correctness, readability, style and layout. We apply high standards, that make you suitable for the real world.
- You are allowed to cooperate with other students during lab sessions and to ask for advice for lab assistants. It is, however, not allowed to share code, or to hand in identical solutions. You must write your code by yourself. See the syllabus for more rules. You may be invited for life grading, for discussion of the quality of your code, or its authorship.

Goal of this exercise is to make you familiar with implementation of set using a *hash table*.

A set contains elements, but it does not distinguish how often an element occurs. A hash table uses a *hash function* to compute an index in a `vector` where the string will be stored. Because there are more strings than possible indices in the `vector`, it will happen that two different strings have the same index. In order to deal with this, we `list` to store the strings that have the same index in the `vector`. In theory, a hash table should be more efficient than a binary search tree, because computation of the hash function can be done in constant time, and the size of the `list`s will be bounded by a constant.

In this task, you have to complete an implementation of set over `std::string` using a hash table. If everything goes well, you will see that the asymptotic performance of **set**, implemented with a hash table has has a somewhat better performance than the implementation of **set** implemented with BST.

Lab exercise 6 is a continuation of exercises 3, 4 and 5. In order to make the measurements, reuse the classes `timer` and `timetable` from the previous exercises. Explanations about their use were given in lab exercise 3.

Download files **set.h**, **set.cpp**, **main.cpp** and **Makefile**. Set, using hash map, is implemented as follows:

```
class set
{
```

```
   size_t set_size;
   double max_load_factor;
   std::vector< std::list< std::string >> buckets;

};
```

Field `set_size` contains the number of string in the hash table. `max_load_factor` defines the maximal value that the *load factor* `set_size / buckets. size( )` can have. The *load factor* is the average size of the `list` in the `vector`. If the load factor becomes bigger than the allowed maximum, the set must be rehashed.

As in previous exercises, strings are compared without distinguishing between upper and lower case. You have already implemented a function `bool equal( const std::string& , const std::string& )` that is case insensitive. The hash function must agree with `equal`. Strings for which `equal` returns true must have the same hash value.

1. Complete the function `hash( const std::string& s )` that computes a hash value for `s`. This function *must agree with* `equal`, which means that it must not distinguish between upper and lower case characters.

   So, `hash( "NurSultan" ) == hash( "nursultan" )` must return **true**.

2. Complete the method `bool simp_insert( const std::string& s )` in file **set.cpp**. As usual, `simp_insert` returns `true`, when insertion takes place. It is called 'simp', because it does not rehash. Do not forget to update `set_size` when `s` is inserted.

3. Implement the method `std::ostream& print( std::ostream& out ) const`. When you are finished, you can remove `#if 0` and `#endif` around `std::ostream& operator << ( std::ostream& out, const set& s )`.

4. Implement `bool remove( const std::string& s )` in file **set.cpp**.

   The function must return **true** if `s` was found and removed. Do not forget to update `set_size` when `s` is removed.

5. Complete the method `void clear( )` in file **set.cpp**.

6. Complete the method `void rehash( size_t newbucketsize )`. It should start with

   ```
   if( newbucketsize < 4 )
      newbucketsize = 4;

   std::vector< std::list< std::string >> newbuckets =
      std::vector< std::list< std::string >> ( newbucketsize );
   ```

7. Complete method `bool set::insert( const std::string& s )`. First call `simp_insert( const std::string& s )`, and after that, decide if a

rehash is necessary. As with `ensure_capacity`, make sure that the bucket size is at least doubled in size.

8. As in the previous lab exercises, you can now do some measurements. The code for doing this is present in **main.cpp**. The performance should be $O(n)$, and it should be better than the performance of the BST-based implementation of **set**.

   Make two measurements: One for `set someset1` (default `max_load_factor` of 3.0), and one using `set someset1( 4, 25.0 )` (`max_load_factor` is 25.0).