CSCI 152, Performance and Data Structures, Exercise 4

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, layout and style. We apply high standards, that make you fit 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 **set** as *abstract datatype*. A set contains elements, but it does not distinguish how often an element occurs, and in which order the elements occur.

You have to write a simple implementation of set over `std::string`, using a `std::vector<std::string>`, and measure its performance. You will see that the asymptotic performance of the `std::vector` based implementation is unsatisfactory, so that you will understand why it is useful to look for better implementations.

Lab exercise 3 was intended as a preparation for the current exercise. In order to make the measurements, you can again use the classes `timer` and `timetable`. Explanations about their use were given in lab exercise 3. This time you have to submit the exercise in Moodle.

Download files **set.h**, **set.cpp** and **Makefile**. Class `set` is intended to represent a finite set of elements. The representation, using `std::vector<std::string>` and the interface are given in file **file.h**. The interface is straightforward.

The definition of class `set` contains the following code:

```
using const_iterator = std::vector< std::string > :: const_iterator;
const_iterator begin( ) const { return data. begin( ); }
const_iterator end( ) const { return data. end( ); }
```

It defines a `const_iterator` of `set` equal to `std::vector< std::string > :: const_iterator`. This code makes it possible to write for example:

```
for( auto p = s. begin( ); p != s. end( ); ++ p )
   (look at *p.)
```

or

```
for( const auto& str: s )
   (look at str.)
```

If you have time, think a few minutes about the question, why `set` should not have a `iterator`.

In order to make the task more interesting, we will require that `set` does not distinguish between upper case and lower case strings. This means that `set s; s. insert( "abc" ); s. insert( "ABC" );` will result in a set of one element.

1. Complete the function `set::equal` in file **set.cpp**. This function must not distinguish between upper and lower case letters. For example:

   ```
   std::cout << set::equal( "aStana", "AsTaNa" ) << "\n";
      // true
   std::cout << set::equal( "astana", "Almaty" ) << "\n";
      // false.
   std::cout << set::equal( "astana", "astana" ) << "\n";
      // true.
   ```

   Use the function `char tolower( char c )`, which is present in the library. Don't make lower case of the complete string. Use `tolower` on each character in the strings separately.

2. Implement `bool contains( const std::string& el ) const` in file **set.cpp**.

3. Implement `bool insert( const std::string& el )` in file **set.cpp**. `insert` must return `true` if `el` was inserted (which means that it was not present).

   It must return `false`, if the element was not inserted (which means that it was already present.)

   After that, complete `size_t insert( const set& s )`. It must return the number of elements inserted.

4. Write the print function
   `std::ostream& operator << ( std::ostream& out, const set& s )`
   in file **set.cpp**. The function must print the set in standard set notation, using `{` and `}`. The order of the elements is not important. Use `set::const_iterator`! This is a real requirement!

5. Implement `remove( const std::string& el )` in file **set.cpp**.

   It must return `true` if `el` was found and removed. Otherwise, it must return `false`.

The best way to implement `remove` is to first find `el`, then `std::swap` it with the `back()` of the `vector`, and then to use `pop_back()`.

Also, complete `size_t remove( const set& s )`. It must return the number of elements that were removed.

6. Complete the methods `size_t size( ) const` and `void clear( )` in file **set.h**.

7. Write `bool subset( const set& s1, const set& s2 )` in file **set.cpp**. It must return `true` if set `s1` is a subset of set `s2`.

8. Finally, it is time to do some measurements: Run the following code:

```
timetable tab( std::string( "set" ));
for( size_t s = 1000; s < 20000; s = 2 * s )
{
   set someset1;

   timer tt;
   size_t nr = 0;
   for( size_t k = 0; k != s; ++ k )
   {
      nr += someset1. insert( addnumber( "aa", rand( )));
      nr += someset1. insert( addnumber( "bb", rand( )));
   }

   auto someset2 = someset1;

   if( nr != someset1. size( ))
      throw std::runtime_error( "counting went wrong" );

   for( const auto& el : someset2 )
   {
      nr -= someset1. remove( el );
   }

   if( nr != 0 || someset1. size( ) != 0 )
      throw std::runtime_error( "counting went wrong" );

   tab. insert( s, tt. time( ));
}

std::cout << tab << "\n";
std::cout << "totaltime " << tab. totaltime( ) << "\n";
```