CSCI 152, Performance and Data Structures, Exercise 7

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.
- Untested code will be graded in the same way as unwritten code.
- Submitted code will be checked for correctness, readability, style and layout. We apply high quality standards.
- 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 submit identical solutions. You must write your code by yourself. See the syllabus for more rules. You may be invited for life grading, for discussion about the quality of your code, or its authorship.

Goal of this exercise is to change the hash-table based implementation of *set* into a *map*. The difference between set and map is that a set can only remember its elements, while a map can remember an associated value.
Mathematically, a map can be viewed a set of pairs with the property that:

$$(p, q) \in M, \quad (p, q') \in M \Rightarrow q = q'.$$

In order to change set into a map, we need to change the implementation of `buckets` from

```
std::vector< std::list< std::string >> buckets;
```

to

```
std::vector< std::list< std::pair< std::string, unsigned int >>> buckets;
```

Since `std::list< std::pair< std::string, unsigned int >` is hard to type and to read, and occurs quite often in the impementation, we will abbreviate it as follows:

```
using listofpairs = std::list< std::pair< std::string, unsigned int >> ;
std::vector< listofpairs > buckets;
```

Use this abbreviation whenever possible. It is part of the grading. Download files **map.h**, **map.cpp**, **main.cpp** and **Makefile**. Class `map` has the following fields:

```
class map
{
   size_t map_size;
   double max_load_factor;

   using listofpairs = std::list< std::pair< std::string, unsigned int >> ;
      // So that we don't have to type and read it all the time.

   std::vector< listofpairs > buckets;
};
```

As with **set**, the field `map_size` contains the total number of key/value pairs in the hash table. `max_load_factor` defines the maximal value that the *load factor* `map_size / buckets. size( )` can have. It is the average size of a `list` in the `vector`. If the load factor becomes bigger than the allowed maximum, the map must be rehashed.

As with **set**, strings are still compared without distinguishing between upper and lower case. You can reuse `equal` and `hash` from the previous task.

1. Study the functions

   ```
   static listofpairs :: iterator
      find( listofpairs& lst, const std::string& key );

   static listofpairs :: const_iterator
      find( const listofpairs & lst, const std::string& key );
   ```

   and make sure that you understand how they work. The functions try to find a pair in `lst` that has `key` as first element. If they find `key`, they return the iterator to the pair that contains it. If they don't find `key`, they return `lst.end( )`.

   The difference between the two versions of `find` is that the second version is **const**. They keyword `static` means that `find` can be called without class object (A `map` in this case.) In order to call `find` from outside of `map`, write `map::find`.

2. Write method `bool insert( const std::string& key, unsigned int val )` in file **map.cpp**. As before, `insert` must return `true`, when insertion takes place, which happens only when `key` was not present before. Do not forget to update `map_size` if `(key,val)` is inserted.

   First call `getbucket` to find the bucket where `key` belongs. After that use `find` to see if `key` already occurs in it.

   For the moment, ignore rehashing. We will come back to that later.

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 map& m )` and print the map.

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

   This function must return **true** if `key` was found and removed. Do not forget to update `map_size` when `key` is removed.

5. Complete method `bool contains_key( const std::string& key ) const` in file **map.cpp**.

6. Complete the two `at` methods in file **map.cpp**. As usual, two methods are needed, because one is **const** while the other one is not **const**. Using the non-**const** version of `at`, one can change the value of an existing key, but one can never create a new key using `at`. Both version of `at` throw an exception if `key` is not present.
   (Use `throw std::out_of_range( "at( ): string not found" );`) It must be possible to write:

   ```
      map german = { { "eins", 1 }, { "zwei", 4 }, { "drei", 3 } };
      german. at( "zwei" ) = 2; // Es tut mir leid.
   ```

   In order to test the case where the exception is thrown, write:

   ```
   try
   {
      german. at( "hundert" ) = 100;
   }
   catch( std::out_of_range err )
   {
      std::cout << "error: " << err. what( ) << "\n";
   }
   ```

   Don't be lazy at testing all cases, also the exception case. We, together with all the instructors and TAs hate it very much when students don't test carefully. Your future employer will also hate it.

7. Complete `unsigned int& operator[] ( const std::string& key );` This method differs from `at( )` in the fact that it adds `(key,0)` if `key` is not present. Because of this, it is possible to write

   ```
      german[ "vier" ] = german[ "zwei" ] + german[ "zwei" ];
         // Works in other languages as well.
   ```

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

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

std::vector< listofpairs > newbuckets{ newbucketsize };
```

For the rest, it is the same as `rehash` for `set`.

9. Complete method `void check_rehash( )`. It checks if a rehash is needed. If yes, it calls `rehash` with a proper `newbucketsize`. As with `ensure_capacity`, make sure that the bucket size is at least doubled in size.

10. Once you have finished `check_rehash`, you can add it in every method that potentially adds a key/value pair. These are

    ```
    bool insert( const std::string& key, unsigned int val );
    unsigned int& operator[]( const std::string& key );
    ```

    There is one problem that you need to **test very carefully**!
    In `operator[]( const std::string& key )`, the following can happen: `key` is not found, and a pair $(\text{key}, 0)$ is inserted into the hash table. After that, it is possible that a rehash takes place. When this happens, $(\text{key}, 0)$ is not any more at the place where it was inserted, and it has to be looked up again, because we need to return a reference to the inserted 0.

11. Using `main` as starting point, you can do some tests, and as always, we want you to be not lazy during testing. It is a habit that will hinder you later in your career.