CSCI 152, Performance and Data Structures, Exercise 5

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 *binary search trees* (BST). **set** as *abstract datatype*. A set contains elements, but it does not distinguish how often an element occurs. A BST keeps the elements ordered, and arranged in a tree, so that the basic operations (insert, remove, lookup) can be performed in $O(n. \log_2(n))$ time.

In this task, you have to complete an implementation of BST over `std::string`, using a `std::vector<std::string>`, and measure its performance. If everything goes well, you will see that the asymptotic performance of **set**, implemented with BST has much better performance than the implementation of **set** implemented with `std::vector`.

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

Download files **set.h**, **set.cpp**, **main.cpp** and **Makefile**. Class `set` is intended to represent an ordered set of `std::strings`. The representation, and the interface are given in file **set.h**. The interface is similar to the vector based implementation, but `const_iterator` was removed, because it has no efficient implementation with the current implementation of BST.

As in the previous exercise, strings are compared without distinguishing between upper and lower case. You have already implemented a function
`bool equal( const std::string& , const std::string& )`. Since a BST

stores its strings in an ordered fashion, you will need to write an additional function `bool before( const std::string& , const std::string& )` that determines the order in the BST.

It is rather tricky to preserve the invariants of BST. In order to help you, we provided `operator <<` and `checksorted( )`. `operator <<` prints the BST in such a way that you can see the structure. Method `checksorted( )` checks that the BST is correctly sorted. During testing, you should call this function after every operation.

We also provide copy constructor, destructor and assignment for `set`, so you don't need to worry about writing those. You should still understand how they work.

1. Write a function `size_t log_base2( size_t )` that computes $\log_2(t)$ rounded down to a natural number.

   ```
   std::cout << log_base2(1) << "\n";       // prints 0
   std::cout << log_base2(15) << "\n";      // prints 3.
   std::ocut << log_base2(16) << "\n";      // prints 4.
   ```

   $\log_2(0)$ is undefined in mathematical sense, but `log_base2(0)` should return 0.

2. Complete the function `before( const std::string& s1, const std::string& s2 )` in file **set.cpp**, which returns **true** if string `s1` comes before `s2` in the alphabetic, lexicographic order, when the distinction between upper case and lower case is ignored. For example:

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

   Use the function `char tolower( char c )`, which is present in the library. Don't convert the complete string to lower case! We will reject solutions that do this. Use `tolower` on each character in the strings separately.

3. Implement the two functions

   ```
   const treenode* find( const treenode* n, const std::string& el );
   treenode** find( treenode** n, const std::string& el );
   ```

   in file **set.cpp**. The first version is needed for lookup, the second version is needed for insertion and removal. Note that the second version never returns `nullptr`, even when `el` is not found.

4. Implement `bool insert( const std::string& el )` in file **set.cpp**. This method must return `true` if `el` was inserted (which means that it was not present). It must return `false`, if `el` was not inserted (which means that it was already present.)

   Function `insert` must use the second version of `find( )` that you wrote for part 3.

   You don't need to worry about keeping the BSTs balanced. The tests below use random strings, so the trees will be reasonably balanced.

5. Write the method `contains( const std::string& el ) const` of class `set`. This is easy, because all you have to do, is call the function `find` that you wrote already.

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

   This function is a bit tricky. First use `find( )` of part 3 (second version) to find `el` if it occurs. If the `treenode` containing `el` not have two descendants, it can be easily deleted. Otherwise, the right descendant of the `treenode` must be inserted in the left descendant. (It was explained during the lectures.) In order to do this, you need to complete the function

   ```
   void rightinsert( treenode** into, treenode* n );
   ```

   that inserts `n` at the right most position in `into`.

   Method `remove` must return `true` if `el` was found and removed. Otherwise, it must return `false`. Test carefully using `operator <<` and `checksorted( )`.

7. Complete the methods `size_t size( ) const` and `void clear( )` in file **set.h**. First complete the function `size( const treenode* n )` outside of class `set`, and then complete class method `size( ) const`. In order to call the outside method from inside the class, you have to write `::size( )`.

8. Complete the method `bool isempty( ) const` that returns **true** if the BST is empty. **This method must work in constant time!**. Therefore, writing `isempty( ) const { return size( ) == 0; }` is not an acceptable solution! It will be rejected.

9. Write a method `size_t height( ) const` that returns the height of the tree. First complete the function `height( )` outside of class `set`, and then complete the class method.

10. As in the previous lab exercises, you can now do measurements. The code for doing this is present in **main.cpp**. The performance should be $O(n.\log_2(n))$, which is much better than the performance of the vector based implementation of the previous exercise 4.

    If you are not getting this performance, there are several possible causes:

(a) You have an $O(n)$ operation hidden in the loop. Check if `isempty()` does not use `size( )`.

(b) The BST is balanced badly. Compare `height( )` with `log_base2( size( ))`. (Note that this comparison destroys the performance, so comment it out during the final test.) If the numbers are very different, then the tree is badly balanced.

(c) You still call `checksorted( )` in the loop. This function is $O(n)$, so if you repeat it n times, you get $O(n^2)$.