CSCI 152, Performance and Data Structures, Assignment 3

Rules for Assignments

- You are expected to read the complete assignment and to complete every part of it. 'I did not see this part' is never a valid excuse for not completing a part of the assignment.
- Submitted code will be checked for correctness, correct memory management, readability, style, and layout. We use **valgrind** for memory checks.
- Assignments are graded with the help of autograders. Make sure that your submitted code can be compiled. If you don't know how to complete a task, insert an empty function, so that your code compiles in all cases.
- In order to make it possible to adopt your code to our automatic checkers, and to give more feedback, we check your code twice. The last submitted version counts. Always submit for the first check. If you submit only for the last check, you are taking unnecessary risk.
- The starter code contains a **main.cpp** file containing rudimentary tests. These tests are not sufficient. You have to create additional tests that include at least all functions that are included in the assignment. In the end, you have to submit a **main.cpp** file with the tests that you created. It is possible that your tests will be inspected.
- If you need help, you can either **(1)** come to a lab session, or **(2)** post your question on Piazza in a private question (with visibility set to 'all instructors'.) Do not mail directly to an instructor.
- Don't wait until the last moment with starting to work on the assignment.
- You must write all submitted code by yourself! Even allowing others to see your code, or looking at others' code is already academic misconduct.

C^{++} Coding Rules

- Avoid unitialized variables. If you don't have a value to initialize a variable, you are almost certainly declaring it too early. C^{++} allows declarations of variables almost everywhere. Declare variables where you need them for the first time.
- Make sure to use `size_t` for all indexing. (Not `int`, nor `unsigned int`.)
- Don't write character constants by their ASCII codes. (Never write 97 instead of 'a'.)
- Do not implement anything in header files. In real life, short methods should be implemented in header files, while longer methods should be implemented in **.cpp** files. Because of the way that we test, this is not possible in assignments. We test with our own header files, so everything that you write in a header file, will be ignored.
- You are not allowed to use any STL containers or library functions, unless explicitly mentioned in the assignment. If in doubt, ask in Piazza.
- Don't use `printf`, `malloc`, `free`, or `memcpy`. Don't use `#define`, except in include guards.
- Don't use `0` or `NULL` to represent the null-pointer. The only correct representation of the null-pointer is `nullptr`. It is OK to write `if(p)` if you want to test that `p` is not the null-pointer.
- Avoid `break` and `continue`. They are just `goto` in disguise. In nearly all cases, they can be avoided by changing the condition of the **while** loop, which always results in more readable code.
- Don't `throw` and `catch` an exception inside the same function. That is not what exceptions are intended for.
- Avoid assignments in constructor bodies. Use member initializers wherever possible.
- Don't write: `if(b) return true; else false;` Just write `return b;`

Goal of this exercise is to make you familiar with the map (also called *dictionary*) ADT and with *binary search trees* (BST) as a possible implementation of map.

A map implements a table of key/value pairs in such a way that for each key, there is at most one value, and this value can be found efficiently, if it exists. Maps are usually implemented as as a *BST* or as a *hash table*. In this assignment, we use BST. A BST keeps its key/value pairs ordered and arranged in a tree, so that the basic operations (insert, erase, lookup) can be performed in $\Theta(\, n. \log_2(n)\,)$ time.

As we have done in the previous assignments, we will implement BST in such a way that it is easy to change the types of key and of value. You have to test the BST with keytype `int`, and also with `std::string`, where we will treat strings as *case insensitive*. That means that `"bst"` and `"BST"` will be treated as the same string. Moreover, both `"AST"` and `"ast"` come before `"bst"` and `"BST"` in the order that the BST will be using. If we would not be ignoring case, then `"BST"` would come before `"ast"` and `"bst"` would come after `"ast"`.

Download the files **map.h**, **map.cpp**, **main.cpp** and **Makefile**. Class `map` defines a binary search tree over a `keytype`, which is defined in class `treenode`. The representation and the interface are given in file **map.h**, together with the declarations of some helper functions.

Since the invariants of BST are tricky, we included `operator <<` and `checksorted( )`. The `operator <<` prints the BST in such a way that you can see its structure. Method `checksorted( )` checks that the BST is correctly sorted. During testing, you should frequently call this method, but you should remove it in your final submission, because checking that the tree is sorted requires a complete pass through the tree, which has $\Theta(n)$, while the basic operations are $\Theta(\, \log(n)\,)$.

All your implementations must be in the file **map.cpp**. Most of the methods of class `map` are implemented by means of helper functions outside of the class. Often these helper functions have the same name as their corresponding class methods. When calling a helper function from inside the class, you must preceed the call by `::`, so that the compiler will not confuse it with the class method.

1 Implementation of BST with int

We first implement BST using `keytype = int`. The order is defined by `keycmp = standard_cmp< keytype >` , which uses the standard order on `int`. In the second part, we will be using `std::string` with a non-standard, case insensitive order.

1. Write the function `unsigned int log_base2( size_t )` that computes $\log_2(t)$, rounded down to the nearest natural number.

```
std::cout << log_base2(1) << "\n";    // must print 0.
std::cout << log_base2(2) << "\n";    // must print 1.
std::cout << log_base2(15) << "\n";   // must print 3.
std::ocut << log_base2(16) << "\n";   // must print 4.
```

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

2. Complete the two helper functions

```
const treenode* find( const treenode* n, const treenode::keytype& key );
treenode** find( treenode** n, const treenode::keytype& key );
```

in file **map.cpp**. The first version is used by method `bool contains( const keytype& key ) const`, while the second version will be used by `bool insert( const keytype& key, const valtype& val )` and `bool erase( const keytype& key )`. Note that the second version never returns `nullptr`, even when `val` is not present in the tree. In that case, it returns a pointer to a null pointer.

Both functions must not be recursive, but iterative. Since the order used for sorting is defined by `keycmp`, you cannot simply call `==, <` and `>` on `key`. Instead write

```
treenode::keycmp cmp;
   // cmp can be called with two values of type treenode::keytype.
   // It returns a negative int if the first value is smaller than the second.
   // It returns a positive int if the first value is greater than the second.
   // It returns 0 if the two values are equal.
```

Note that the specification does not guarantee that the returned value is -1, 0, or 1. It can be different integers.

3. Write the method `bool insert( const keytype& key, const valtype& val )`.

If `key` is not present, `insert` must insert `key/val` (without changing `key`), and return `true`.

If `key` is present, (as decided by `treenode::keycmp`), it must not change the existing `key/val` pair, and return `false`.

Method `insert` must use the second version of `find( )` that you wrote in part 2.

You don't need to worry about keeping the BST balanced. The tests will use keys that are shuffled, so the trees will be reasonably balanced. Keeping a BST balanced is hard, too hard for an assignment in the first year.

4. Write the method `contains( const keytype& key ) const`. This method must use the first version of `find( )` that you wrote before in part 2.

5. Write the method `const map::valtype& at( const keytype& key ) const`. This method also uses the first version of `find( )`. If `key` is not present, this method must throw `std::out_of_range`.

6. Next, write `map::valtype& map::at( const keytype& key )`. This method has to call the second version of `find( )`, even though we never create a new node. The reason is that the first version returns `const treenode*`, which makes it impossible to get **non-const** access to the value. If the key is not present, this method must throw `std::out_of_range`.

7. Write `map::valtype& map::operator [] ( const keytype& key )`. This method is similar to the **non-const** version of `at`, but instead of throwing an exception, it creates a value, when `key` is not present. This is done by the default constructor of —valtype—.

8. Implement `bool erase( const keytype& key )`.

   This function is complicated. Carefully read the lecture slides. First use the second version of `find( )` to find the `treenode` of `val`, if it exists. If this `treenode` does not have two subtrees, it can be easily removed.

   Otherwise, you need to extract the rightmost key/value pair from its left subtree, and put it in the place of the deleted key/value pair. The easiest way to do this, is by using an additional function

   ```
   treenode* extractrightmost( treenode** from );
   ```

   This function keeps on walking into the right subtree of `*from` until it reaches a node without right subtree. It will extract this node from the tree, and replace it by its left child, if there is one. It returns the extracted node without deleting it. Function `extractrightmost` is given, so you can use it.

   Once you have extracted the rightmost node of `*from`, you can copy its value into the node that contained `val`, and delete `*from`.

   Method `erase` must return `true` if `val` was found and removed. Otherwise, it must return `false`. Test carefully using `operator <<` and `checksorted( )`, and also with **valgrind**.

9. Complete the helper function `size_t size( const treenode* n )`, that returns the size of the tree starting at `n`.

   Also complete method `size_t size( ) const`. The call of the helper function must have form `::size( )`, because otherwise the compiler will confuse it with the class method `size`.

10. Complete the helper function `size_t height( const treenode* n )`, which returns the height of tree below `n`. Note that, because `height()` returns `size_t`, it is impossible to return -1 when the tree is empty. We therefore redefine the height as the number of nodes in the longest path from the root to a leaf in the tree. It is always one more than the height as defined in the lectures. The empty tree has height zero, and a tree consisting of one node, has height one.

After that, complete method `size_t height( ) const`. Again, if you call the helper function, the call must have form `::height( )`.

11. Complete the helper function `deallocate( treenode* n )`, which must delete all `treenode`s that are reachable from `n`.

    When you are finished, you can complete the destructor of class `map`.

    Also complete the method `void clear( )`.

12. Complete the method `bool empty( ) const`, that returns `true` if the BST is empty. **This method must work in constant time!**. Therefore, you cannot implement it by checking that the size is zero. Such solutions will likely cause a time out during testing.

13. We still need a copy constructor and an assignment operator. First complete the helper function `treenode* makecopy( const treenode* n )`. After that, complete the copy constructor.

14. Now you can also complete `map& operator = ( const map& other )`. First check for self assignment. If there is no self assignment, then deallocate the current tree, and replace it by a copy of `other`.

15. For containers, copying can be expensive, so it natural to create the moving operators. Implement the moving copy-constructor `map( map&& other )`. **The moving copy-constructor must work in constant time!**.

16. Also create the moving assignment operator `map& operator = ( map&& other );` Erase the current tree, and copy the pointer from `other`. Make sure that `other` is left in empty state. Note that moving assignment to itself is UB, hence you don't need to check for it.

2 Implementation of BST using string

In order to try out BST for `std::string`, change `int` into `std::string` in class `treenode`. Don't change anything in class `map`. Change the `#if`s in **main.cpp** to select the tests for `std::string`. The code will compile, but it will not work as we want, because it uses the standard order on strings, which distinguishes between upper and lower case. If one first calls `insert( "abc", "xyz" )`, and after that, calls `at( "ABC" )`, the call to `.at` will throw an exception. In order to solve this problem, you have to complete the next task:

17. Implement the method

```
int case_insensitive_comp::operator( ) (
          const std::string& s1,
          const std::string& s2 ) const
```

This method must compare the strings `s1` and `s2`, ignoring their case. It must return a negative number if `s1` comes before `s2`, it must return 0 if `s1` equals `s2`, and return a positive number if `s1` comes after `s2`.

For example

```
case_insensitive_cmp cmp;
std::cout << cmp( "aA", "Aa" ) << "\n";   // Prints 0
std::cout << cmp( "Xy", "xyz" ) << "\n";  // Prints a negative integer.
std::cout << cmp( "Shymkent", "Aqtau" ) << "\n";  // Positive.
```

It is not allowed to make lower case copies of the strings, in order to compare them with standard comparison. Compare the strings directly. We will look at your implementation!

3 Submission

Submit your files **map.cpp** and **main.cpp**. Make sure that your submitted **main.cpp** compiles with

```
using keytype = std::string;
using keycmp = case_insensitive_cmp;
using valtype = std::string;
```

Also try out the other test functions. Your code must work for all combinations of `keytype,keycmp` and `valtype`. Make sure that your functions in **map.cpp** do not create debugging output, and remove all calls to `checksorted` because they degrade performance. Create your own tests. Our diligent TAs will look at your **main** file and check if you created tests of your own.