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. We apply high quality standards during grading.
- 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 at least twice. The last submitted version counts. Use the earlier checks. If you submit only for the last check, you are taking unnecessary risk.
- If you need help, first your question on Piazza. Questions that do not contain code can be asked publicly. Questions that contain code must be posted as private question. Don't mail questions directly to an instructor, because individual instructors are not always available. Piazza is monitored almost full time, so you will get a quicker answer.
- 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 probably declaring it too early. C^{++} allows declarations of variables almost everywhere.
- Make sure to use `size_t` for all indexing. (Not `int`, nor `unsigned int`.) Don't write character contants by their ASCII codes. (Write 'a' instead of 97.)
- Do not implement anything in header files. In real life, short methods can be implemented in header files, while longer methods should be implemented in **.cpp** files. 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 for.
- Avoid assignments in constructors. 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 set ADT and with *binary search trees* (BST) as implementation of the set ADT.

A set stores elements in such a way that the same element cannot occur more than once. Sets are usually implemented as a *BST* or *hash table*. In this assignment, we will implement sets of `std::string` using BST. A BST keeps its elements ordered and arranged in a tree, so that the basic operations (insert, remove, lookup) can be performed in $\Theta(\, n.\log_2(n)\,)$ time. You will measure the performance and check that the observed performance agrees with the theoretical performance.

As we have done in the previous assignments, we will implement BST in such a way that it is easy to change the underlying type. You have to test the BST with `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, `"AST"` and `"ast"` come before both `"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"` wold come after `"ast"`.

Lab exercise 7 was intended as a preparation for the current assignment. In order to make the measurements, you must use the classes `timer` and `timetable` that were provided in exercise 7. Explanations of their use were given in exercise 7.

Download the files **set.h**, **set.cpp**, **main.cpp** and **Makefile**. Class `set` defines a binary search tree over a `valtype`, which is defined in class `treenode`. The representation and the interface are given in file **set.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 call this method after every operation, but during time measurements you should remove it, because checking the invariant 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 **set.cpp**. Most of the methods of class `set` 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 `valtype = int`. The order is defined by `valcmp = standard_cmp< valtype >` , 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::valtype& val );
treenode** find( treenode** n, const treenode::valtype& val );
```

in file **set.cpp**. The first version is used by method `bool contains( const valtype& val ) const`, while the second version will be used by `bool insert( const valtype& val )` and `bool remove( const valtype& val )`. 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 `valcmp`, you cannot simply call `==`, `<` and `>`. Instead write

```
treenode::valcmp cmp;
   // cmp can be called with two values of type treenode::valtype.
   // 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 valtype& val )`.

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

   If `val` is present, (as decided by `treenode::valcmp`), it must not change the current value 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 random values, so the trees will be reasonably balanced. Keeping trees balanced is hard, too hard for an assignment in the first year.

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

5. Implement `bool remove( const valtype& val )`.

   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 value from its left subtree, and put it in the place of the deleted value `val`. 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 `remove` must return `true` if `val` was found and removed. Otherwise, it must return `false`. Test carefully using `operator <<` and `checksorted( )`, and also with **valgrind**.

6. 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`.

7. 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. Because of this, we 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( )`.

8. 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 `set`.

   Also, complete the method `void clear( )`.

9. 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 time out during testing.

10. 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.

11. It remains to complete `set& operator = ( const set& 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`.

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 `set`. Change the `#if`s in `main( )` to select the tests for `std::string`. The code will work, but it will not work as we want, because it uses the standard order on strings, which distinguishes between upper and lower case. In order to solve this problem, you have to complete the next task:

12. Implement the method
    `int 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.
    ```

    Don't make a lower case copy of the strings. Compare the strings directly.

3 Performance Measurements

As in the lab exercise, you can now do some performance measurements. The code for doing this is already present in **main.cpp**. Use `g++ -O3 -flto`, and no valgrind. (You should use it during testing, however.) Make sure that there are no other processes on your computer. The performance should be $O(\, n.\log_2(n)\,)$. If you are not getting this performance, you are doing something wrong, and you will lose points. There are several possible causes:

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

2. 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.

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

Collect the measurements and include them in a comment in **main.cpp**.

As a final step, do some measurements to show that unbalanced trees have bad performance. In order to do this, change the value of the boolean, so that `values` will be sorted, and run the tests again. You will see that you need to decrease the final value for `s` very much. Find a final value such that the run time lies between 15 and 60 seconds. Again collect the measurements, and include them in a comment in **main.cpp**.

4 Submission

Submit your files **set.cpp** and **main.cpp**. Make sure that **main.cpp** contains the two measurement tables in comments. Make sure that the `#if`s in your `main( )` are set such, that your code compiles with `valtype = int`.