Homework Assignment 6

Due Date: 18:30, May 26, 2016

1 Problem 1

Binary search tree (BST) is a widely adopted data structure. Some kinds of self-balancing BST, such as red-black trees, are especially popular. This time, however, we are not going to implement red-black tree, or guys may start eating each other. You are asked to implement AVL tree in this homework instead, and learn more about iterators in the process.

1.1 Requirement

The spec of the implementation mimics the counterpart in STL, but we are only going to implement a subset of the functionality. The template class `AVLTree` in `AVLTree.hpp` has three template arguments: the type of key, the type of mapped datum and the type of a class with `operator()` that indicates an strict weak ordering that the container should follow. It's similar to `std::map` except that `std::map` has a fourth argument you don't have to care in this homework. Everytime `AVLTree` tries to locate a given key, it calls a functor, say `Comp`, which is an instance of the type specified in the third argument and determines where the key should appear. If `Comp(A, B)` is true, A should appear before B, i.e. it should appear in its left child if `B` is stored in current node, and vice versa. If both `Comp(A, B)` and `Comp(B, A)` are false, they are considered equivalent. The functions listed below are the functions in `AVLTree.hpp` you have to implement:

- Left rotate and right rotate
- Increment and decrement operators of iterators, in an in-order traversal manner
- The Find function

The places you should fill code are planted comments begin with `// TODO`. You can easily tell which part needs your work to accomplish the homework. Please note that the implementation simply sets the location of an iterator which points to an past-the-end element, e.g. returned from `end()`, to `nullptr` in this homework. This behavior is different from STL containers.

1.2 Notification

`AVLTree.hpp` uses a bunch of new features absent from C++98. Here lists significant ones:

- Trailing return type, e.g `auto foo::lol(void)->bar;` is equivalent to `foo::bar foo::lol(void);`.
- The keyword `auto` tells compiler to deduce the type of an variable from its initializer during compilation.
- Range-based for loop, e.g. `for(auto& It: lmao) It+=1;` is equivalent to `for(auto It=lmao.begin(); It!=lmao.end(); ++It) (*It)+=1;`

You can google it (recommended) or ask TA for more info.

1.3 Sample Program

Here is a sample program to test your implementation. If your code is able to be built along with `WordCounter.cpp` and it works properly, you can get at least part of the credits. Please note that TA may use other program to evalute your implementation. You can modify the code at will, but it should be compatible with original interface.

`WordCounter.cpp`

```
#include <iostream>
#include <string>
#include <cctype>
#include <functional>
#include <set>
#include "AVLTree.hpp"

using namespace std;

int main() {

        AVLTree<string, size_t> Counter;
        string Buffer;

        while( cin >> Buffer ) {

                string Word;

                for(char Char: Buffer) {

                        if( isalpha(Char) )
                                Word += tolower(Char);

                        }

                if( ! Word.empty() )
                        Counter[Word]++;

                }

        AVLTree<size_t, set<string>, std::greater<size_t>> Sorter;

        for(const auto& It: Counter)
                Sorter[It.second].insert(It.first);

        for(const auto& Rank: Sorter)
                for(const auto& Word: Rank.second)
                        cout << Word << ": " << Rank.first << endl;

        // TA will use Dump() to inspect the internal structure.
        Counter.Dump();

        return 0;

        }
```

1.4 Input

The input is a paragragh.

1.5 Output

The first part of output is the count of each word. The second part of the output is the internal structure of the AVL tree represented in a pre-order manner. You don't have to worry about the output format. The function `Dump` is provided by TA.

1.6 Sample Input

```
In each ascetic morality, man prays to one part of himself as a god and also finds it necessary
to diabolify the rest.
```

1.7 Sample Output

```
to:  2
a:  1
also:  1
and:  1
as:  1
ascetic:  1
diabolify:  1
each:  1
finds:  1
god:  1
himself:  1
in:  1
it:  1
man:  1
morality:  1
necessary:  1
of:  1
one:  1
part:  1
prays:  1
rest:  1
the:  1
((man:  1), ((each:  1), ((as:  1), ((also:  1), ((a:  1), (), ()), ((and:  1), (), ())),
((ascetic:  1), (), ((diabolify:  1), (), ())), ((himself:  1), ((god:  1), ((finds:  1),
(), ()), ()), ((in:  1), (), ((it:  1), (), ()))), ((one:  1), ((necessary:  1), ((morality:
1), (), ()), ((of:  1), (), ())), ((prays:  1), ((part:  1), (), ()), ((the:  1), ((rest:
1), (), ()), ((to:  2), (), ())))))
```

2 How to submit the assignment?

1. There won't be any demo taking place on 2016-05-26. Please submit a simple report as well as the code instead. The report shall not exceed two pages.
2. Name the source code of each problem and the report as following:
   - Problem 1: p1.hpp
   - Report: whatever name you like
3. Do not rename the files or put them into any directory. Upload them directly to the **e-Campus (E3)** system. You will get no credit if you don't follow the rule. Note that the penalty for late homework is **15% per day**, and late homework will not be accepted after 3 days past the due date. In addition, homework assignments must be individual work. If I detect what I consider to be intentional plagiarism in any assignment, the assignment will receive **zero credit.**