Advanced Programming final project: C++ part

Eleonora Donadini and Valeria Paolucci

March 2019

1 Introduction

For our final project, we were required to implement a template binary search tree (BST) in C++ language.
A BST is a hierarchical data structure where each node can have at most two children (left and right nodes) and each node stores a pair of a key and the associated value. The binary tree is ordered according to the keys: given a node N, all the nodes having keys smaller than the key of the node N can be found going left, while all the nodes with a key greater than the key of the node N can be reached going right.

2 Implementation details

We implemented a ***class BinTree***, templated on the type of the key and the type of the value associated with it. Among the private implementation details of the class, we have a unique pointer to the root node and we declare some functions (in all the cases, recursive and implemented outside the class) that are called through the public corresponding methods. Moreover, nested inside the class, we implemented a ***structure Node***, whose attributes are a const key-value pair (for which we exploited the ***std::pair***, a struct template that provides a way to store two heterogeneous objects as a single unit), two unique pointers to the left and right node respectively, and a raw pointer to the upper node. In the public interface, we have copy and move semantics, the classes Iterator and ConstIterator, and several methods (in most of the cases, implemented outside the BinTree class).

- The ***insert method*** is used to insert a new pair key-value. It inserts the first node (root node) if the tree is empty; otherwise, it calls ***insert_more private method***, which is a recursive one. If a key is already present in the tree, we chose to replace the value with the newest one;
- The ***clear method*** is used to clear the content of the tree. Thanks to the features of unique pointers, its implementation is very brief, thus avoiding us to use a recursive function;

- ***begin*** and ***cbegin*** return respectively an iterator and a const_iterator to the first node (i.e. the leftmost, having the smallest key);
- For ***end*** and ***cend*** we chose to return an iterator whose internal pointer points to nullptr;
- The ***balance method***, in our case, performs a non-in-place balancing of the tree. In order to do that, the nodes of the tree are stored, in order, in an auxiliary vector; then the tree is cleared and the private method balance is called to rebuild the tree in a balanced way, reading the values from the auxiliary vector;
- We implemented ***find method*** in both non-const and const versions. They are used to find a given key and return an iterator (or constiterator) to that node. If the tree is not empty, they both call the recursive private method recfind;
- We implement ***copy and move semantics*** for the tree;
- Since the tree must be traversed in order, we overrode the ***operator<<***, to be able to print in order the pair (key: value) of all the nodes in the tree.
- We also implemented the ***operator[]*** in the const and non-const versions;
- We added a ***function size***, useful to obtain the number of nodes of the tree.

3 Test

We used C++14 and we compiled our code with the flags -Wall -Wextra, taking care not to have any warning. We also checked that there were no memory leaks by means of valgrind.

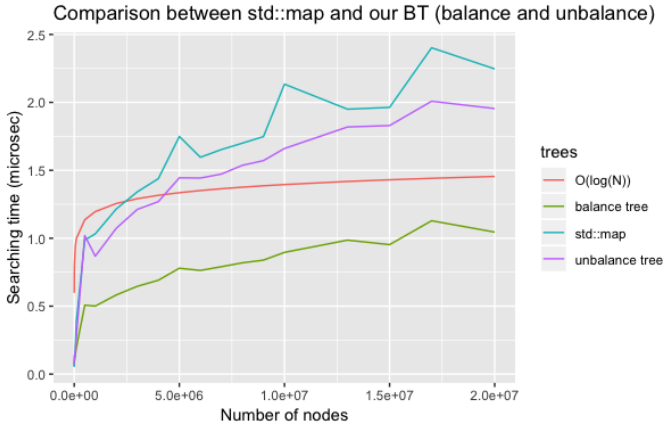
3.1 Testing the correctness

To check that our BinTree class is working properly, we wrote a Test.cc file.

3.2 Testing the performances

In this section we analyze the performance of the lookups using the function ***find*** before and after the tree is re-balanced. We also compare the results we get with the ***std::map*** where the ***std::map*** is an ordered associative container use to store elements represented by a key-value pairs with unique keys.
The search operations is expected to have a logarithmic complexity $O(\log(N))$.
To prove that we compile our Benchmark.cc file whit ***-O3*** level of optimization; the code first built each tree whose number of nodes is given by the user (specifying with the command line option -D the size of our data structures) and

after search a node inside the tree by a key. For this purpose we use the library chrono specifying the time interval to search is measured in microseconds, than we repeated the measures several times with different sizes and plotted the results in the following graph 3.2.
We can see that each trees presents expected behaviour moreover the balance tree presents better results and that due to the fact of the ordered implementation.



To compare the performance of our tree implementation before and after the balance we use the same approach of the previous case and we obtain the results shown in 3.2. The plot point out the better efficiency of the find() function of the balance tree.