Advanced Programming Exam Report

By *Edoardo Alessandroni*, *Alberto Luvisutto* e *Andrea Mecchina*. All the project was implemented, debugged and tested by the whole of us.

Introduction

This report explains the implementation and the benchmarking of a binary seach tree implemented in `C++`. The implementation of the container makes use of the iterators to move across the tree. This implementation is compared with `std::map` and `std::unordered_maps`. All the codes are commented in Doxygen style.

The main purpose of our implementation is to write a correct `C++` code rather than a extremely performant one.

The code compiled `-Wall -Wextra` does not produce any warning. We also used Valgrind to ensure that the code has no memory leaks.

Implementation

The binary search tree consists of a templated class `bst` containing the nested struct `node` and the nested class `__iterator`, both templated. The `include` folder contains the following header files:

- `bst.hpp`, which contains the hierarchical structure of the `bst` class and the signatures of all its methods, together with the inline implementations of the shortest of them,
- `bst_functions.hpp`, which contains the non-inline implmentation of the `bst` methods which require a greater amount of coding,
- `node.hpp`, which contains the structure and the implementation of the node struct methods,
- `iterator.hpp`, which contains the structure and the implementation of the iterator class methods.

Each functions has its own Doxygen style comment documentation.

All the requested methods are implemented; the only other methods are `order`, an iterative private method called by the `balance` method and `op_eq`, returning a boolean according to the values of two input keys.

We used `std::unique_ptr` for the children nodes and raw pointers for the parent node. We chose smart pointer in order to automatically handle the objects' lifetime and their dynamic memory allocation.

For the same reason, the `bst` data member `head` is a unique pointer.

The iterator class also contains the overloading of the operators `*` and `->` in order to make user's life easier.

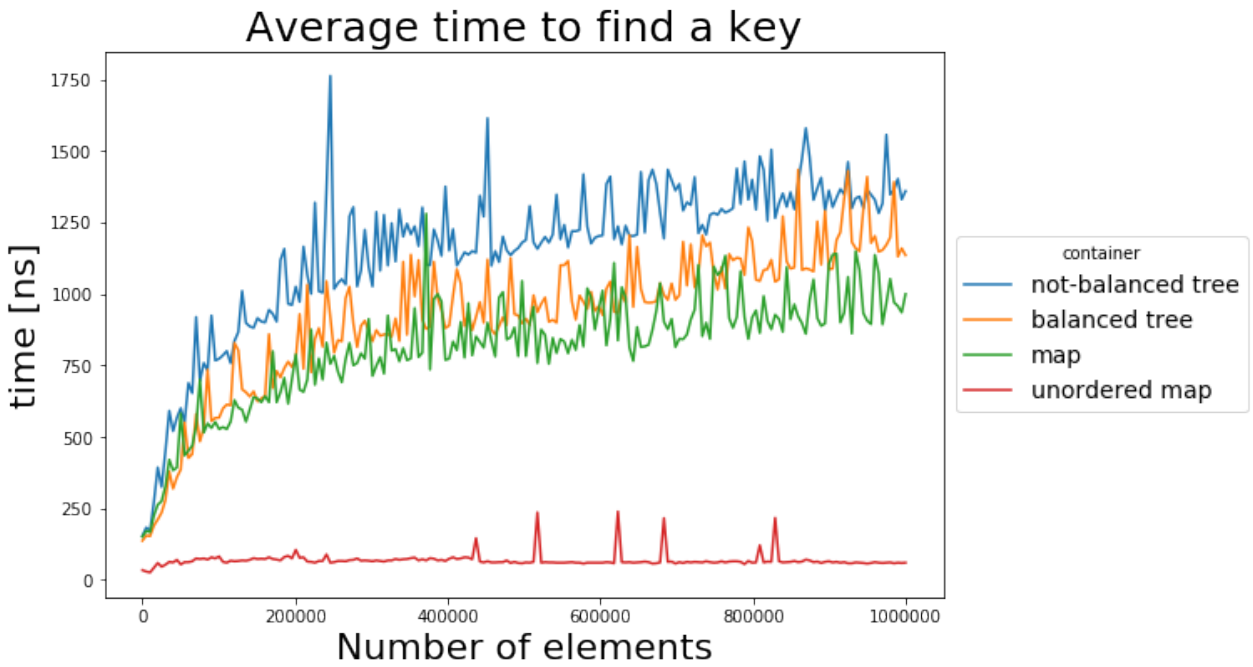
An additional iterator template is used to avoid code duplication while distinguishing between constant and non constant iterators.

Benchmark

Since the main advantage of a binary search tree is the logaritmic dependence on the number of elements, when searching a given key, some benchmarks have been performed timing a call to the `find` method.

In particular, the structures compared are `std::map`, `std::unordered_map`, our `bst` tree and a balanced version of it. We chose to average every measure on 200 keys to find. The measures have been taken varying the number of elements per container in a range from 5000 to 1000000.

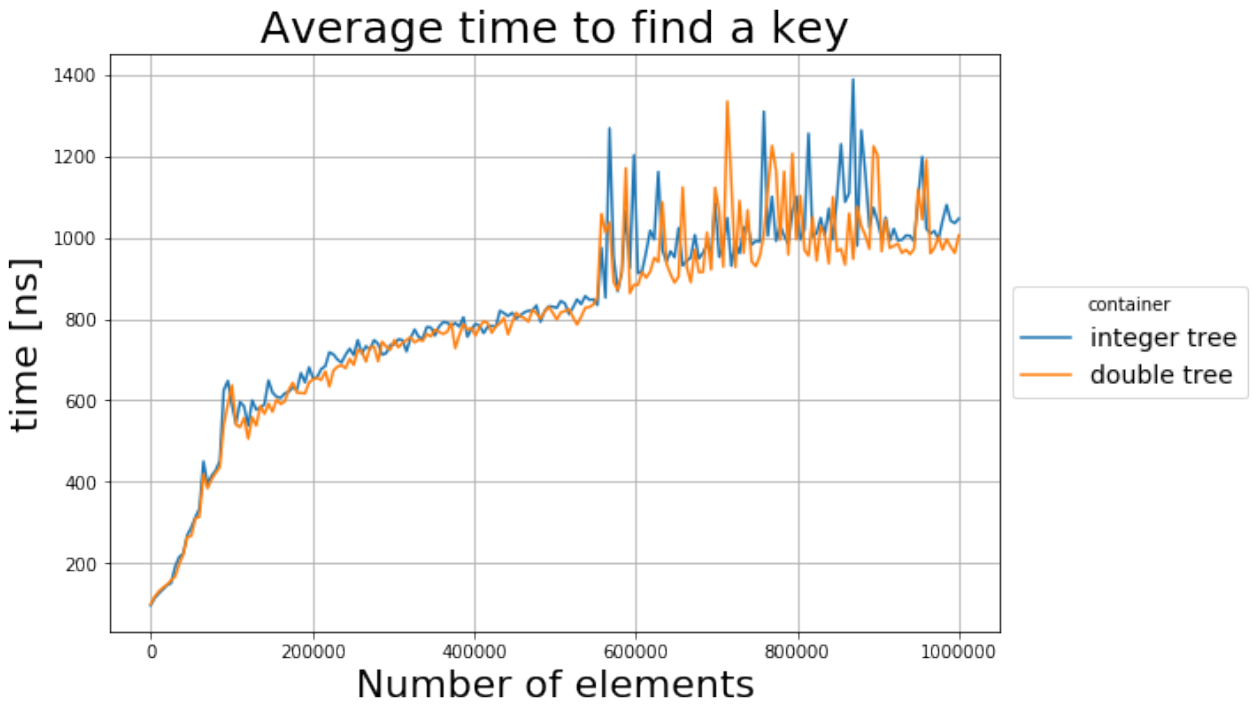
The following plot compares the results among the different containers.



It shows that our balanced tree performs better than its non balanced version, but slightly worse than `std::map` and much worse than `std::unordered_map`.

A logarithmic dependence on the number of elements in `bst` containers is clearly visible.

We also tested the search time varying the type of the key, using integers and doubles. The following plot shows the results.



No relevant differences have been observed.