Report: Binary Search Tree

The aim of this project is to implement a template binary search tree, which mainly allows to create a binary tree, store and search data in it and balance the tree. After implementing the class, we are interested in comparing the performance of our tree with and without balancing. Moreover, we want to compare the tree with the associative container std::map.

Struct Node

Since a binary tree is a data structure based on nodes linked to each other, our class BST depends on a templated data structure Node. This structure represents a single node of the tree, thus it has as attributes an std::pair that stores key and value of the node and two pointers to Node, left and right. Key and value of the node are templated, since in our tree we want to manage different types of nodes. The value can be both numeric and alphabetic, instead the type of the key must be of the numeric type to allow the sorting of the nodes.

Class BST

Private members

The class BST has some private variables: a pointer to the first node of the tree, a pointer to the last node and pointer to a constant, called head, that points to the first node. The choice to declare const the head node is due to the fact that this variable is used to keep the tree alive even when all its nodes are deleted, so head pointer points to a node that has the only utility to keep track of the first node of the tree, the root node, and it is never changed.
Moreover, we declare as private two integers which keep track of the current number of nodes and height of the tree.
We declared as private also some helper functions that facilitate the definition of the public methods.

Public members

We decided to define two constructors: the default constructor, which creates an empty tree, and a customized constructor that allows to create a tree setting the value and the key of the first node.
We implemented copy and move semantic both with specific constructors and with functions that overload the operator "=" for copy and move.
Since the pointers to the first and the last node are private member, we need functions to return their current value: begin and end. We were also expected to implement two functions which return constant pointers to these nodes: we did this with a constant casting since the pointers root and tail are non-constant class members and the const_cast is the only type of casting that allows to turn them into constant pointers.

Other relevant methods that are directly connected with tree management are **insert, find**, **balance** and **clear**. All of these functions make use of private recursive methods:

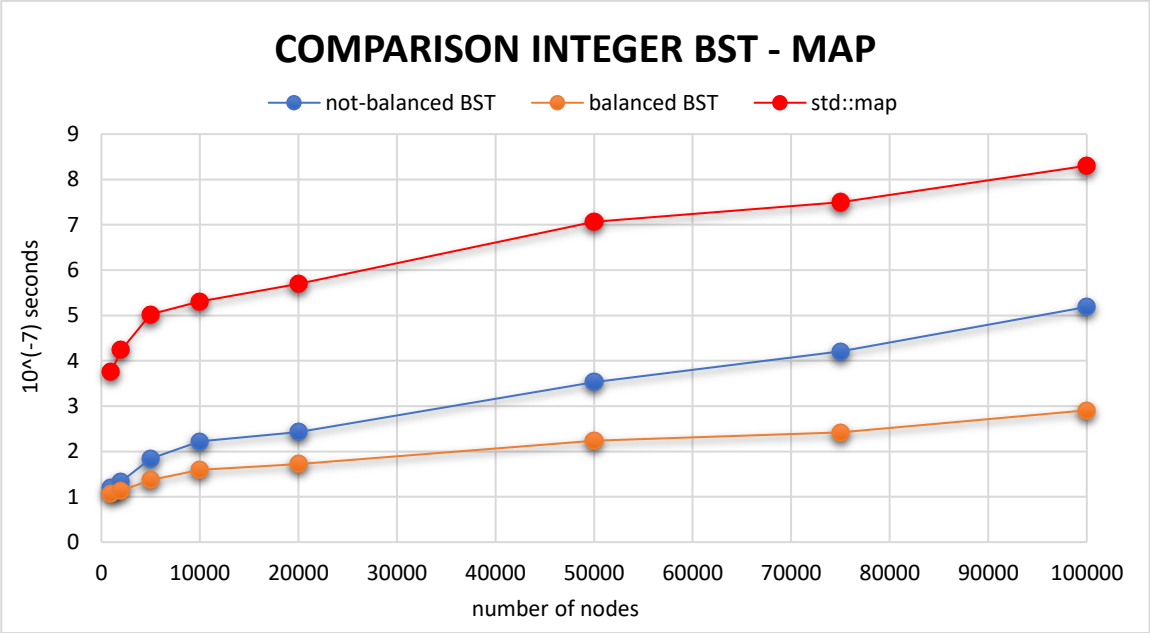
- **Insert** make use of a function that takes a value and a key and put them into a new node placed in the tree respecting the tree ordering rules. Moreover, the function increments by one the private variable number of nodes and also height of the tree if necessary.
- The function **find** takes as argument a key and returns a pointer to the node in which the key is found. If the key is not found returns a pointer to the last node and print a warning message.
- The function **balance** make use of two private functions: the first creates an std::vector of nodes ordered with respect to the key. The second function rebuild the tree starting from the node in the middle of the ordered vector as root. In this way, the nodes are equally distributed between the right and the left side of the tree (differing for one node if the number of nodes is odd).
- The **clear** method uses the private recursive method clear that deletes all the nodes that are present in the tree.

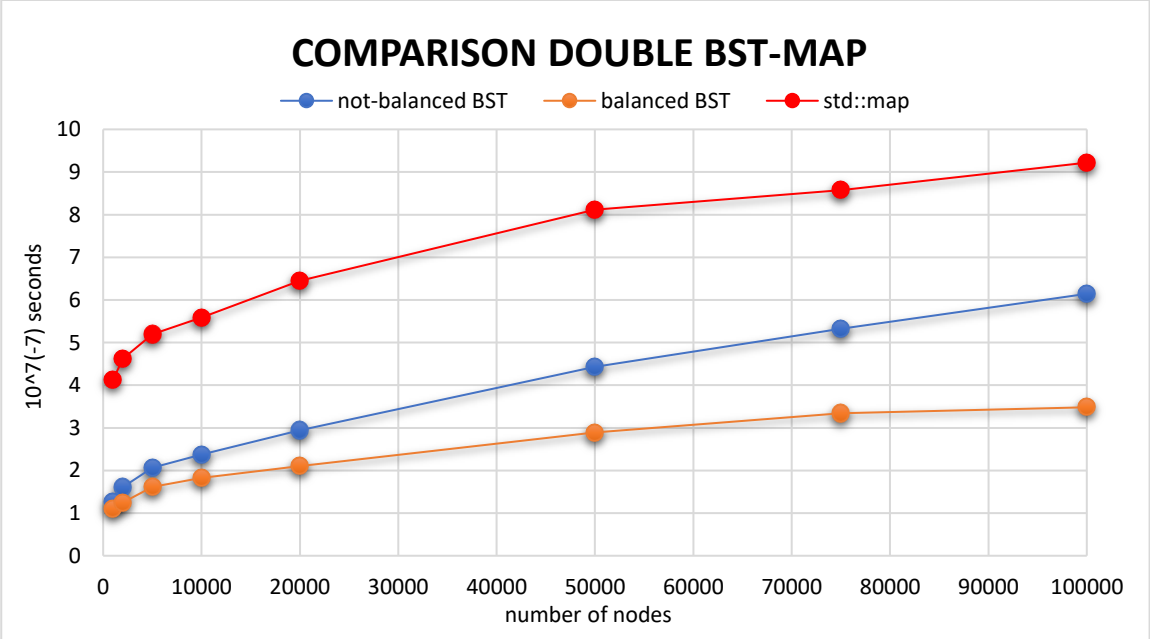
Performance test

To test the lookups performance of our implementation of a binary search tree we have implemented a **performanceTest** method that measures the mean time to find a single node, before and after the tree is re-balanced, comparing also the results with a std::map.

In this method we create an std::vector of random values, that will be our keys, and we fill the BST and the map with these numbers. After that, we swap the value of each element with that of some other randomly picked element, using the random_shuffle function. Next, we search every key in our BST, not-balanced and balanced, and in the std::map, measuring the time that it takes to complete these operations and obtaining the mean time to find a single node. We have done this for integer keys and double keys, with the aim of verifying whether there is a difference between the two cases.

We got the following results, presented in the following two graphs:





The time complexity in a binary search tree and in a std::map have to be O(log(N)), from these graphs it can be noticed that this behaviour is respected in both cases.
An expected result is that the balanced tree takes less time than the not-balanced tree, moreover, an unexpected result is that our BST implementation works better than the map, even if the tree is not balanced.

In the end, it can be noticed that there is not a big difference, in terms of times, between the case of integer keys and double keys.