Binary Tree Implementation

Maria Grazia Berni

Introduction

The goal of this project is to implement a binary search tree, without self-balancing mechanism. The code is organized in two many folders: the include folder contains the implementation files, the test folder has a test file with a functional test, and a file with the benchmark. Also a folder with a documentation is present, and another folder in which the main results of the benchmark are stored.

Brief Description of the Implementation

The Binary Search Tree class, *bst*, is a template class with three arguments : the Key type K, the value type V and the function used to perform the comparisons, *Compare*, which has default value $std :: less < K >$. The nodes of the tree are represented using an internal private class *Node*, and a unique pointer to a *Node* type is used to store the root position in the class *bst*. I choose to implement *Node* as an internal private class because its implementation and many of its methods, for instance the *successor* function, reflect the logic of the tree and they have no sense out of this context.

The Node class

The class *Node* contains the following attributes :

- an std::pair containing the key and the value to store in an instance of Node.
- a unique pointer to the left child of the current node
- a unique pointer to the right child of the current node
- a pointer to the parent of the current node

I choose to use smart pointers because in this context it simplify a lot the memory management. There are three constructor in the class: the default constructor, a constructor to construct a node storing an std::pair and initializing to nullptr the left child, right child and the parent, and a constructor that in addition to this latter, sets the parent node. An explicit copy constructor is used to support the copy semantic of the *bst* class, and performs a recursive copy of the node. The copy assignment is not implemented because it does not make sense in this context, in fact the copy and move semantic of the class Node is a support to the copy and move semantic of the entire *bst* class, in fact there are no move or copy operations involving only part of the nodes. The *Node* class is equipped with functions to set the left or right child, to delete them, a function which returns the Node with the maximum key in the tree, and one that returns the Node with the minimum key. Moreover, a function successor, is useful to perform an in-order visit of the tree, and even to delete elements of the tree.

The Iterator class

The *bst* class has two forward iterators: $Iterator$ and $Const\_Iterator$. The *iterator* class is templated with two argument: $Node$ and KV, where this last argument takes the value $std :: pair < const\ K, V >$ in case of Iterator, and $const\ std :: pair < cont\ K, V >$ in case of Constant Iterator. In the *bst* class, $Iterator$ and $Constant\_Iterator$ are defined within the scope of the class itself. Iterator class takes as data member a reference to an instance of the $Node$ class, *current_node*, and an explicit constructor is initialized with this value. As required, Iterator and Const_Iterator define the reference operator, the pointer operator, an increment and a post increment operators, the equality and inequality operators.

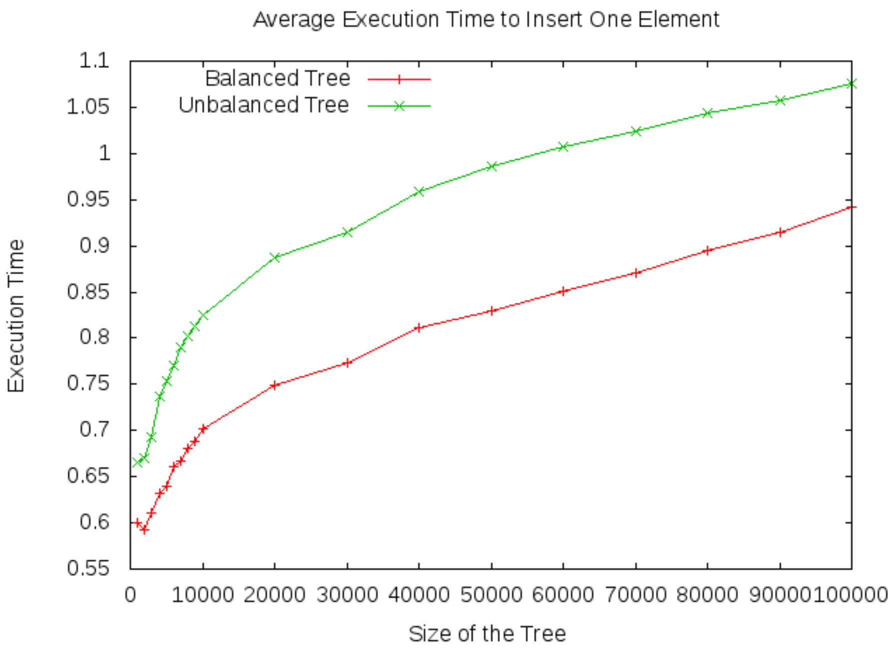
Description of bst methods

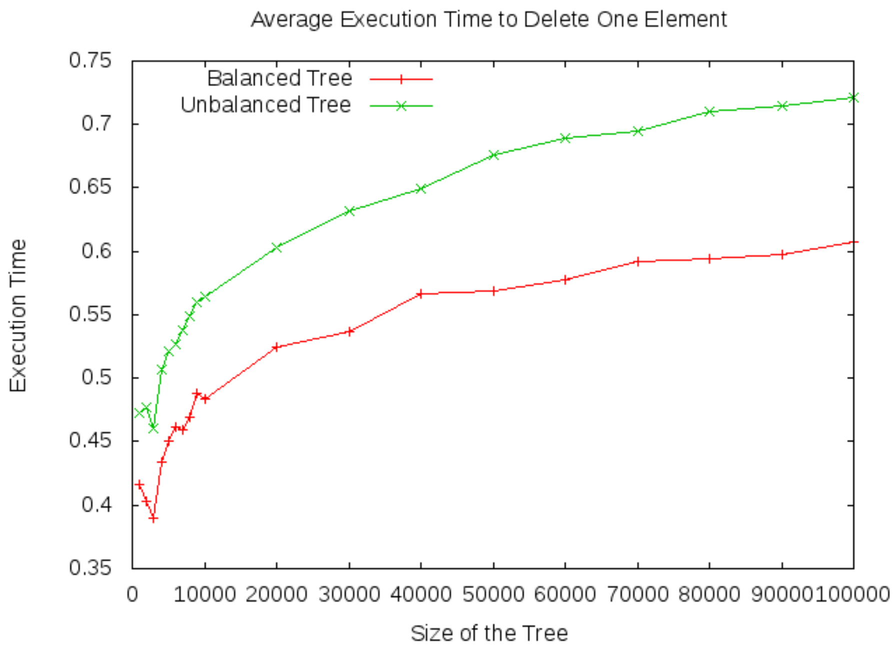
The *bst* class performs the copy and move behavior. An overloaded method to insert new values is implemented: one version to insert a key-value pair from an l-value and another from an r-value. The insert method is implemented in a trivial way, comparing the keys respect to the total order function and inserting the element in the right place. After that the function returns an std::pair with an Iterator initialized with the node in which the insertion has been done, or to the node containing the key in case this was already present in the tree, and a bool which is true or false depending on whether the insert operation was successful or not. An emplace method uses the insert method and construct an std:.pair<K,V> in place. The return value is the same as the one of the insert function. The find methods return an Iterator or a Constant Iterator initialized to the node where the key has been found, to the end of the tree (nullptr) otherwise. To delete a node containing a certain Key, I implemented the erase function which uses four auxiliary functions to perform the cancellation: del_leaf in case the node to delete is a leaf, *del_root* to deal with the cancellation of the root and *del_node_with_one_child* in case the node to delete has only one child and *del_node_with_two_children* in case it has two. This has been done to simplify the erase function and remove some if statements. The balance method does not balance the actual tree, but erase it and, using a sorted vector and the auxiliary recursive function *balanced_tree*, rebuild a new balanced tree.

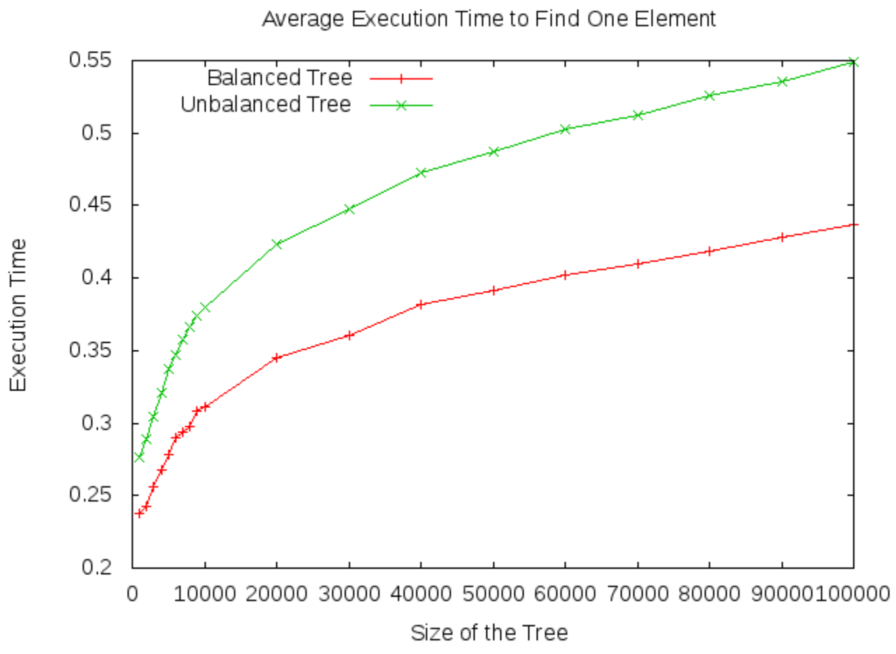
Benchmark

The benchmark measures the execution times of the insert, erase and find operations, with respect to the size of the tree. The tree used in the benchmark has been built in a random way, so it is to be expected that it will not be too unbalanced.

The insert operation was always successful in this benchmark, because it has been executed using elements that were not present in the tree, so it can be considered as an upper bound time. Moreover, to be precise taking times, the insert has been executed many times. So, in order to be consistent with the statement "taking time at constant size", have been inserted in the tree a number of elements equal to 1/8 of the number of leaves that the tree would have in case it was balanced. Since the actual tree is not too unbalanced, in this case there is no substantial change in the number of levels. The same is valid for the erase function. Here the inserted element will be deleted. In this case the erase is always successful, but it's always deleting elements stored in the leaves, so, while the "true deletion" is faster respect to the deletion when dealing with internal nodes, the pre-operation of finding the element takes always time $\Theta(logn)$ (in case of balanced tree), so even in this case, this times has to be considered an upper bound. Finally, the find operation search for elements that have the 50% of probability of being in the tree at any level, and the 50% of probability of not being present in the tree, so it's as general as possible. The very same times are taken in the case of balanced tree. Here some plot of the performances:







This operations have a $\mathcal{O}(log_2(n))$ complexity in case of balanced tree. The balanced tree is in fact performing better, but even the unbalanced tree has a logarithmic complexity because it has been constructed in a random way.