Advanced programming course
Final project
Binary search tree

Lorenzo Basile and Arianna Tasciotti

February 2020

Contents

1 Introduction

In this report we present our C++ 14 implementation of a binary search tree (bst) and, in the last section, we report the results obtained by benchmarking our data structure against `std::map`, a standard container whose behaviour is replicated by our bst.

2 Program structure

Our implementation of binary search tree relies on the interplay of three classes: `bst`, `node` and `_iterator`, defined in different header files.
`bst` is templated on a comparison operator `cmp_op` (the default is `std::less<key_type>`), the type of the keys `key_type` and the type of the values `value_type`.
`node` is templated on `T`, the type of data stored in the nodes of the tree (in our case, an `std::pair<const key_type, value_type>`).
`_iterator` is templated on `node_t` and `pair_type`, respectively the type of the node (in our case, `node<std::pair<const key_type, value_type>>`) and, once again, the type of data stored in the node. This last template parameter would not be strictly necessary to implement an iterator class but it is useful to define both an iterator and a const iterator using the same `_iterator` class.

2.1 Class `bst`

This class has two private member variables: `op`, which is an object of type `cmp_op`, and `root`, which is a `std::unique_ptr` to the root node of the tree, or to `nullptr` if the tree is empty.
We provided this class with a default constructor and destructor and with copy and move semantics. The copy constructor and overloaded `operator=` for copy semantics were implemented making sure to perform a deep copy, by means of a recursive call of the copy constructor of the class `node` (2.2).

2.1.1 Private functions

We decided to implement some private utility functions to make easier the design of the public ones and to avoid code duplication.

The function `_insert()` takes a forwarding reference and returns a `std::pair <iterator,bool>`. Since this method is called by the public functions `insert()` and `emplace()`, its input is by design a `const` lvalue or rvalue reference to `std::pair<const key_type, value_type>`, which is then used to find the right place where the pair should be inserted in the tree (if an insertion is actually required) and forwarded to the proper node constructor.

If the key is not present in the tree, a new node is created with the given key and value and the function returns a pair made of an iterator pointing to the newly allocated node and `true`; otherwise the function returns a pair made of an iterator pointing to the node already containing the given key and `false`.

The function `_find()` takes a `const` lvalue reference to `key_type` and looks for a node which contains this key. If such node is present in the tree, the function returns a pointer to this node, otherwise `nullptr`. The search is performed in a binary fashion: starting from root node, the given key is compared with the key of the node (according to a certain comparison operator `op`); the node against which the key is compared is then updated according to the result of the previous comparison until the key is found or a leaf node has been reached.

The functions `leftmost()` and `findmin()` are used to find the leftmost descendant of the right child of a given node (which is passed as a pointer to node).
`leftmost()` returns a pointer to the parent of the input node if this node has no right child, otherwise it calls `findmin()` and returns its result, which is a pointer to the actual leftmost descendant of the right child.

2.1.2 Public functions

The function `insert()` performs the insertion of a `std::pair` in the tree (if the key is not already present), and it is provided in two versions: one takes a `const` lvalue reference to a pair, the other a rvalue reference to a pair. The two functions return the result of `_insert()`, invoked on the correct input type.

The function `emplace()` is implemented using a variadic template and it is used to insert a node in the tree, taking in input either a `std::pair` or a key and a value. It returns the result of `_insert()`, invoked forwarding the input parameters.

The function `clear()` is used to empty out the tree, releasing the memory occupied by its nodes. This is performed by resetting the root to `nullptr`.

The functions `cbegin()` and `cend()` are used to traverse the tree in order (from left to right) and they return a const iterator pointing to respectively the leftmost node and one node after the rightmost.
`begin()` and `end()` are provided in two overloaded versions which return an iterator or a const iterator pointing to the same positions pointed by `cbegin()` and `cend()`.

The function `find()` searches for a node containing a given key and it is provided in two versions: one returns an iterator pointing to that node (if it exists) or to `end()` (if it does not exist) while the other returns a const iterator pointing to the same positions.
The search is performed by calling the function `_find()` (section 2.1.1).

The function `balance()` is used to balance the tree: we consider the tree balanced if, for each node, the number of right descendants and the number of left descendants are equal or differ by one. `balance()` stores the nodes of the tree in a `std::vector` through the function `vectorize()` and calls `clear()`.
The vector containing the nodes is then reordered using the function `reorder()` which finds the median of the input vector and emplaces it back in the returned vector; then, it calls itself recursively on the right and left subvectors.
Finally, `balance()` reinserts the nodes in the tree reading their values from the re-ordered vector.

The two versions of subscripting operator (`operator[]`) take as input a `const` lvalue reference or a rvalue reference to `key_type` and look for a node whose key is equal to the input one: if such node is not found, a pair containing the given key and the default value for the `value_type` is inserted in the tree. Finally, these functions return a reference to the value whose key is equal to the one given as input or to the newly inserted value.

The put to operator (`operator <<`) is a `friend` function of the `bst` class: it is used to print the tree to an output stream (for example to standard output using `std::cout`). It takes as arguments a reference to an output stream object and a tree and returns the same output stream object to allow concatenated call. The tree is accessed in order using the const iterator.

The function `erase()` takes as input a `const` lvalue reference to a key and removes the node containing this key, if it exists, without removing its descendants.
At first, `_find()` is called and if it returns `nullptr`, this means that the key is not present and no node has to be erased, otherwise it returns a pointer to the node to be erased. The right child of this node (if present) is attached to its grandparent in the same way the node to be erased was attached to it and the left child is attached to the leftmost node of the right subtree. Otherwise, if there is no right child, the left child is directly attached to the grandparent node. Finally, the pointer to the node to be erased is used to `delete` the node.

In `bst.hpp` there are other two functions: `size()`, which simply returns the number of nodes of the bst and which is used to reserve the right vector capacity in `vectorize()`,

and `unbalanced()`.
`unbalanced()` returns true if the tree is unbalanced (according to the criterion described in section 2.1.2) and false otherwise. It would be reasonable to call such a function in `balance()` to avoid balancing an already balanced tree but, due to its implementation which requires counting right and left descendants at each split, it turns out to be much slower than `balance()` itself and therefore it is only useful to have an idea of how nodes are actually inserted in the tree and to test the correctness of the function `balance()`.

2.2 Class `node`

The class `node` (actually, a struct) is templated on `T`, the type of data stored in the node and has four member variables: a (raw) pointer to the parent node, two `std::unique_ptr`, one pointing to the left child and one to the right child, and a variable of type `T`, the actual value stored in the node.
We provided this struct with three constructors and a default destructor: one constructor takes a `const` lvalue reference to `T` (the data to store in the node being constructed) and a pointer to node (the parent of the node to construct), another one a rvalue reference to `T` and a pointer to node and both initialize the new node with these values (paying attention to the constructor of `T` to be called) and with `nullptr` as left and right children; the last one is a copy constructor, used for deep copy and called by the copy constructor of class `bst`.

This struct has two functions, both used by the `unbalanced()` function of class `bst`: `num_nodes()` and `unbalanced_node()`.
`num_nodes()` returns a `std::pair<int,int>` containing the number of right and left descendants of a given node; `unbalanced_node()` returns a `std::pair<bool,const node*>`, set to `true,“pointer_to_unbalanced_node”` if there is unbalance on a node of the subtree on whose root the method has been called, and to `false,{}` if the subtree is balanced.

2.3 Class `_iterator`

The class `_iterator` is templated on `pair_type`, the type of data stored in the node, and on `node_t`, the type of node. It has one member variable which is a raw pointer to the current node of type `node_t`.
We provided this class with a constructor which initializes the raw pointer to the current node, a default destructor and copy semantics.
Furthermore, we implemented a dereference operator which returns a reference to the data stored in the current node, a pre-increment operator which allows to traverse the tree from left to right and boolean operators `==` and `!=`.

3 Benchmark

In order to assess the performances of our code, we measured the time needed to access elements of type `std::pair<int,double>` stored in the tree through the `find()` function on a randomly initialized `bst`, on the same tree after balancing and on a `std::map` initialized in the same way as our tree.
In particular, fixed a size of the tree N, we inserted in a `bst` and in a `std::map` nodes with keys equal to all values in the set $\{0, 1, ..., N-1\}$ and random `double` values between 0 and 10 in a random order.
Using `std::chrono`, we measured the average time (over all the keys) needed to access the elements stored in the trees via the `find()` function. Then, the test was repeated on the same `bst` after having balanced it through `balance()`. In all these tests, we made sure that `find()` was actually performed by using its returned value to perform a sum.
We ran our benchmarking program [1] with 9 different sizes N: $\{10^4, 2\times10^4, 5\times10^4, 10^5, 2\times10^5, 5\times10^5, 10^6, 2\times10^6, 5\times10^6\}$ and for each size we performed 10 runs to average results and to compute their standard deviations, by means of a Python script.[2]
The results are reported in microseconds against a logarithmic x axis and we represented the standard deviations as vertical bars. We used shaded regions to make the graph more interpretable, since some bars could overlap with each other.
In figure 1 we report the results obtained running the program compiled without optimizations, while in figure 2 we show the results of the same runs with optimized code (compiled with the `-O3` flag). The compiler used is g++, version 9.2.0.
From these graphs we can see that, as we could expect, optimizing the code improves significantly performances, with a maximum time decrease (for balanced `bst` and $N = 5 \times 10^6$) of over 5 times.
Moreover, the expected logarithmic growth in time is more or less reflected in the plots, since all curves are almost linear (on a log scale), with the only partial exception of the curve obtained for the balanced `bst` with `-O3` optimization, which stays almost flat for all values of N.
In figure 1 we can see that `std::map` is faster than our unbalanced `bst` but it is (surprisingly) slower than the `bst` after balancing.
This behaviour is confirmed by the plot in figure 2 where the balanced `bst` becomes dramatically faster than the other two containers and `std::map` becomes even slower than the unbalanced `bst`; probably this result can be explained by the low complexity of our data structure if compared to `std::map`, which makes the optimization easier to the compiler.

[1] File "benchmark.cpp" in folder "Benchmark".
[2] File "script.py" in folder "Benchmark".

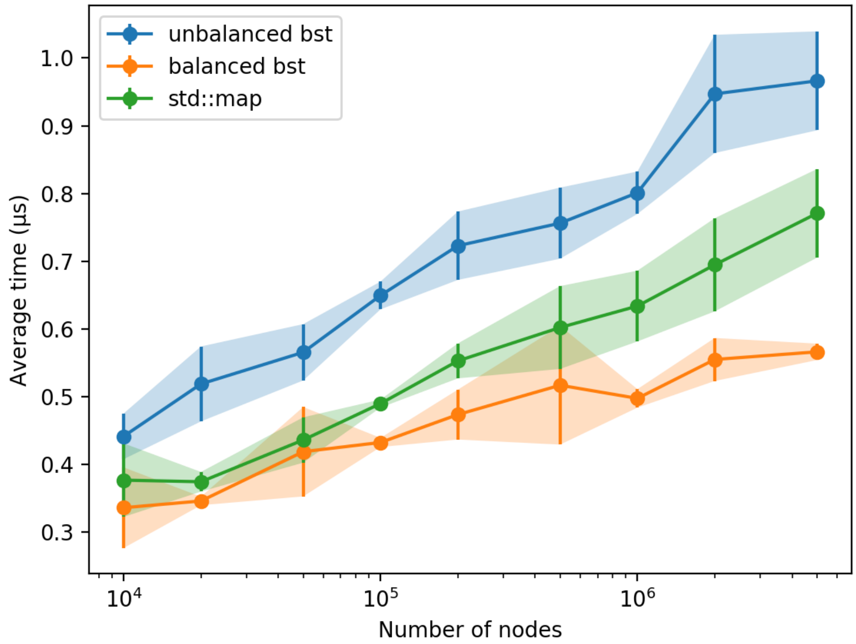

Figure 1: Average times for `find()` without optimizations for unbalanced `bst`, balanced `bst` and `std::map`.

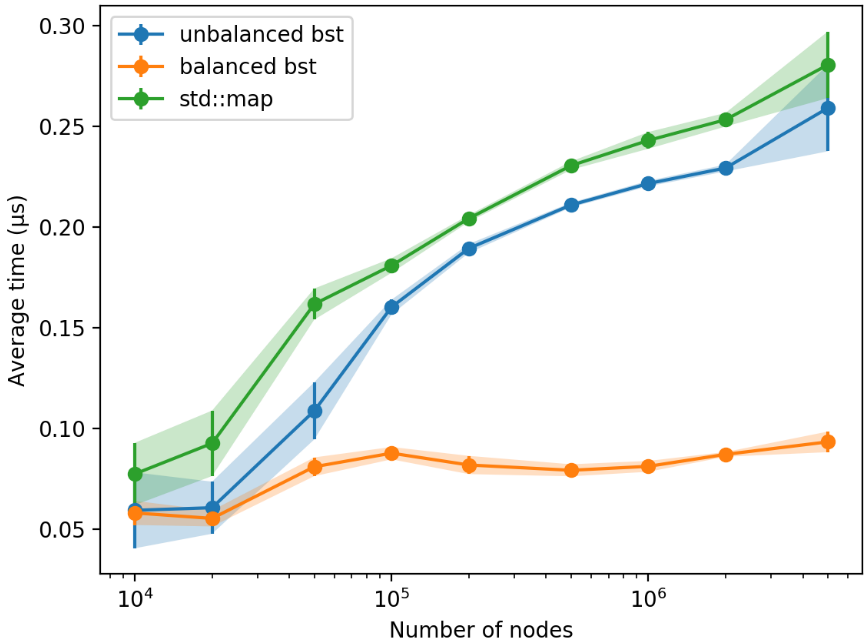

Figure 2: Average times for `find()` with optimizations for unbalanced `bst`, balanced `bst` and `std::map`.