Advance Programming Project

Buscaroli Elena, Valeriani Lucrezia

DSSC 2020-2021

1 Binary Search Tree - C++ script

The tasks we were asked to solve, were to implement a binary search tree, using C++14, and to benchmark our output code against `std::map` and `std::unordered_map` to evaluate its performance.

The implementation of our binary search tree is mainly based on `node<T>` struct, `bst<value_type,key_type,cmp_op>` and `_iterator<node,T>` classes. We decided to generate two different header files, `bst.hpp` and `iterator.hpp`, containing `node` and `bst`, and `_iterator`, respectively. The two header files, are then included in the `main.cpp` file, containing instead the `main()` with the calls to the implemented functions.

1.1 Struct `node`

First of all, we implemented the `node<T>` struct. This is a struct templated on `T`, and has four public attributes: a raw pointer pointing to the parent node, two `std::unique_ptr<node>` unique pointers, one pointing to the left and the other to the right nodes, and the attribute `value`, of type `T`, storing the value of the node.

This struct presents five different constructors. Two of them, given a const lvalue reference or a rvalue reference to an object of type `T`, and a raw pointer to the parent node, set by default to nullptr, will initialize a new node having as `value` the input one, with the left and right `std::unique_ptr` to nullptr and the parent raw pointer initialized to the input one, if given, or to nullptr, if not provided.

Two constructors, given a const lvalue or rvalue reference to a `T` object, and raw pointers to left, right and parent nodes, with the parent one set by default to nullptr, will initialize a node with attributes set to the input values.

Eventually, we defined a copy constructor which, given a `const std::unique_ptr` reference to a node and a raw pointer to the parent node, set to nullptr as

default, will perform a deep copy of the node, by resetting the left and right unique pointers, if present, to point to the left and right nodes of the input one.

We use as deconstructor the default one.

1.2 Class `bst`

Then, we implemented the `bst<value_type,key_type,cmp_op>` class. The class is templated on the type of the values, the type of the keys, that will be both used in each `std::pair<key_type,value_type>` associated to the nodes, and the comparison operator, set as default to `std::less<key_type>`.

The class presents two private attributes: the `std::unique_ptr` pointing to the root node of the tree and the comparison operator `cmp_op op`.

The class has four private functions, that will be called by public functions and have been implemented to avoid code duplication. The first one is `_find(const key_type& x)`. The purpose of this function is to check, given a `const` lvalue reference to an object of type `key_type`, if a key with the input value is present in the tree. The function will return a raw pointer to the found node, if present, or a nullptr, if the key was not present in the tree.

The second private function is `_insert(O&& x)`, a function templated on `O`, which, given a reference to an object of type `O`, will check if such object is already present in the tree, by means of `_find()`, and if not present, it will insert it in the right position of the tree. The function will return a `std::pair<iterator,bool>` where the iterator will be pointing to the inserted node, if not present, or to the already existing node, otherwise, and the bool will be `true`, if the object was inserted by the function, or `false`, otherwise.

The third private function is `_balance(std::vector<pair_type>& nodes, int start, int end)`. Giving as input a vector containing all the tree nodes and two integers corresponding to the starting and final vector indexes. Then, the function calls itself recursively on the first and second halves of the vector to balance the tree.

The last private function is `_inorder(node* x)` that, given as input a raw pointer to a node, will find and return a raw pointer to the left-most node in the (sub-)tree rooted in the input node.

Regarding the public member functions, we decided to use the default constructor and deconstructor, as well as for the move constructor and assignment. We implemented the copy constructor and assignment `operator=` to perform a deep copy.

Then, we implemented the `insert()` function, which aims at inserting a `std::pair`

in the tree and returning a pair `<iterator,bool>`. We defined two versions of this function, taking as input a `const` lvalue or a rvalue references to a `std::pair` object. The function simply calls the `_insert()` function on the given input and returns the pair returned by `_insert()`.

The `emplace()` function, templated on `Types`, performs the insertion of a pair in the tree taking as input either a `std::pair` or a key and value, by calling the function `insert()` forwarding the input parameters with `std::forward<Types>`, and returning the object returned by `insert()`.

The `clear()` function clears the content of the tree, by resetting the tree root node to nullptr.

The `begin()` and `end()` functions, each provided in two versions, return an iterator or a const iterator to the left-most and one past the right-most nodes, respectively. The `cbegin()` and `cend()` functions search and return a const iterator to the left-most and one past the right-most nodes in the tree.

The function `find(const key_type& x)` is implemented in two versions, returning an iterator or const iterator to the node having the key equal to the input one, if present, or to nullptr. Both versions call `_find()` function on the input key, that will return a raw pointer to the found node or to nullptr, if the key is present or not. Then each version will return an iterator or const iterator to the raw pointer returned by `_find()`.

The function `balance()` generates a vector containing all the nodes in the tree, calls `clear()` to clear the content of the tree and calls `_balance()` on the nodes vector, with start and end as 0 and the number of nodes, respectively, to generate a balanced tree.

The `erase(const key_type& x)` function erases the node in the tree having key `x`, if present, by releasing its ownership on the other nodes. First of all, the function calls `_find()` to check if the key is present or not in the tree. If not, the function returns.
Otherwise, if the key is present, we might face three different situations: the node to be erased has no left and right nodes, hence it is a leaf; it might have only one left or right node; it has both left and right nodes. In all the situations, we need to check if the node to be erased is the root and, if not, release the node parent's ownership, and in the two latter cases, we also need to reset the ownership of the node to left and/or right nodes. When the node to be erased has two children nodes, the function will call `_inorder()` to find the first in-order successor, and then calls itself to erase the successor node, after having stored its value.

As last thing, we overload two operators. The first one is the friend put-to operator `std::ostream operator<<`, that prints the key of the nodes ordered, from the left-most to the right most-one, using const iterators.

The other implemented operator is the subscripting `::operator[]` that searches the input key, given as either a const lvalue or rvalue reference, and returns the associated value. If the key is not present in the tree, it inserts a node with that key and as value a random `value_type`. This operator is overloaded to return a lvalue reference to the value mapped by the key.

1.3 Class `_iterator`

The class `_iterator<node_t,T>` is templated on `node_t`, the type of the struct node, and on `T`, the type of the data stored in the node. It has a unique private attribute `node_t* current`, a raw pointer to the current node.

The class is provided with five public member functions. The first one is the constructor which initializes the raw pointer to the current node, and as deconstructor we used the default.
We overload the dereference operator `*` to return a reference to the pair stored by the node object and the reference operator `->` to return a raw pointer pointing to the pair stored by the node.
Next, we overload the pre-increment operator `++`, which allows the traverse of the tree in key order, so from the left-most to the right-most node. So, it is useful for printing out the tree.
As last thing, the two friend `bool` operators `==` and `!=` are implemented to compare two iterators.

1.4 Benchmark

In order to asses the performance of our binary search tree (bst) implementation, we measured the time of execution of the function `find(key_type& x)`, that searches if a node with the given key `k` is present or not in the tree. The performance of the `find()` function is measured on our unbalanced and balanced bst and on the `std::map` and `std::unordered_map` containers, part of the standard libraries. The functions used for the benchmark of our code are contained in the `test.cpp` script.

As first step, we create, with the function `generate_tree()`, trees having as many nodes as the input size; the key of the nodes are number in set $\{0, ..., size-1\}$ and the value is equal to the key.
The size used are $1 \times 10^2, 2 \times 10^2, 4 \times 10^2, 6 \times 10^2, 8 \times 10^2, 1 \times 10^3, 2 \times 10^3, 4 \times 10^3, 6 \times 10^3, 8 \times 10^3, 1 \times 10^4, 2 \times 10^4, 4 \times 10^4, 6 \times 10^4, 8 \times 10^4, 1 \times 10^5, 2 \times 10^5, 4 \times 10^5, 6 \times 10^5, 8 \times 10^5, 1 \times 10^6, 2 \times 10^6, 4 \times 10^6, 6 \times 10^6, 8 \times 10^6$.

Then, we apply to each tree the `find()` function; we start to measure the time with `std::chrono::high_resolution_clock::now()` before the call of the function. The `find()` is called for all the keys in the tree, so `size` times; we stop the time after the last key was find. Lastly, we calculate the final time by dividing the obtained elapsed time by the size of the tree, so the number

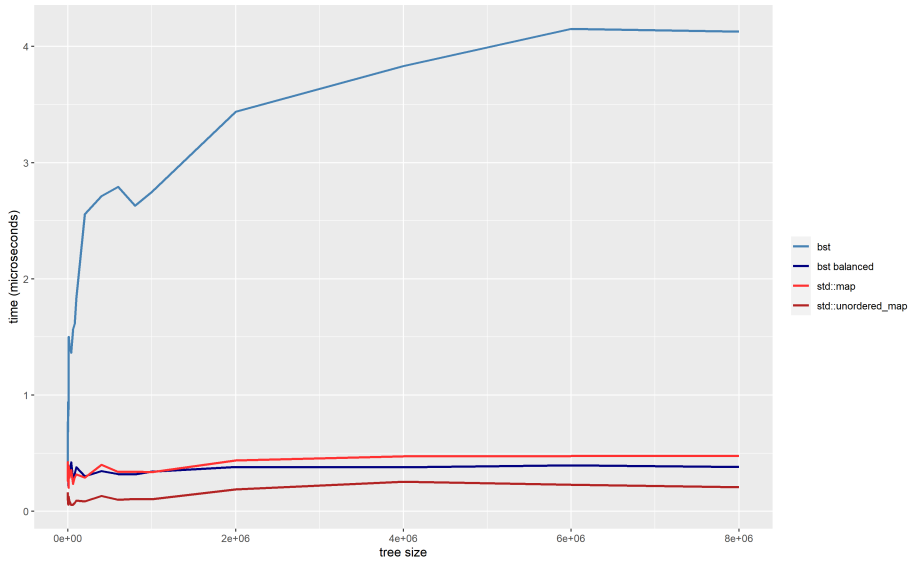

Figure 1: Comparison of the time, in microseconds, varying the number of nodes in the tree, taken by our code to find a key in unbalanced and balanced trees, with the time taken by `std::map` and `std::unordered_map`.

of nodes; the timing is reported in microseconds. Those results are obtained for each size and for each of the four containers. The procedure was performed four times in order to obtain more robust results by averaging the timing. The obtained results are reported and plotted in Figure 1.

From Figure 1 we can observe that the highest time is obtained by the bst in the unbalanced version; the `std::map` and the balanced bst show similar timing and the best performing one is the `std::unordered_map`, as expected, since it is know that the key are not internally ordered but stored in an hash table, so the access is constant in time. The similar result between bst balanced and `std::map` can be explained by the fact that the tree implemented in `std::map` is a red-and-black tree, a balanced tree type.

2 Reverse Dictionary - `Python` script

The solution for the problem of reversing a `Python` dictionary is reported here:

```
def reverse_dict(d):
    s = {s for a in d.values() for s in a}
    rd = {k:[j for j in d.keys() if k in d.get(j,False)] for k in s}
    return rd
```

In order to solve this problem we propose a two step solution. In the first step a

set `s` containing the values of each key, each reported only once, is created using a set comprehension. This set contains all the new keys. Then each element `k` in the set `s` becomes a key of the `rd` dictionary; the value `j` for each key is determined through a list comprehension. In fact each value is a list containing all the old keys that have as associated value the new key `k`. We exploit the dictionary comprehension with a nested list comprehension in order to obtain a faster code.