Advanced programming: C++ project

Alessia Paoletti, Michela Venturini

January 31, 2019

Abstract

The project consists in the implementation of a *Binary Search Tree* using C++11 and testing its performances compared to `std::map` concerning the access to the elements.

1 Implementation

The implementation is a simple *not autobalancing Binary Search Tree*. The binary tree is ordered according to the keys and it is templated on the key, on the type of the value and on the operation used to compare different keys. The program is implemented following `pair programming` and `incremental` developement approaches. We start from the implementation of simple classes and methods and then we refine the intermediate results.
We organize the code in an unique header file `BST.h` containing all the classes' and the methods' declarations and implementations. Two `main.cc` are used to test the correctness of the code and the performances.

2 Structs and Classes

Class BST The class BST contains all the other classes. Its private members are:

1. the struct `Node` which is the unit of the BST and we will present later its implementation
2. `std::unique_ptr<Node> root` that represents the root of the BST

The public members are:

1. The class `Iterator`
2. The class `ConstIterator`
3. The object `C compare` that is initialized by default to the function object `std::greater<K>` templated on the type of the key.

The class implements also the public move constructor, copy constructor, move assignment and copy assignment, the non-const and const versions of the operator [], that allows the access to a node of a tree using the key and return the respective value, and a operator $\ll$ overloading to print the tree in a standard format. Both the copy constructor and the copy assigment use a private method copy() that act a deep copy from a BST to another one, by passing it a poiter to the root. Other methods will be specified below.

Struct Node This struct represents the structure of nodes in our Binary Search Tree and it is composed of:

1. `std::pair<const K, V> pair` that represent the node itsef - its key and associated value.
2. `std::unique_ptr<Node> left` and `std::unique_ptr<Node> right` that represent the left and right children of the node.
3. `Node* const parent` that is a pointer to the parent of the current node; the parent is defined as:
   - the node to which the current node is appended to, for lefthand nodes.
   - the parent of the node to which the current node is appended to, for righthand nodes.

   This choice allows to find easily the successor of the node and simplify the implementation of the class Iterator.

Moreover the class implements a custom contructor taking as argument a key, a value and the pointer to the parent of the node.

Class Iterator and ConstIterator The classes `Iterator` and `ConstIterator` implement the `Iterator design pattern` that allows to scan the BST without knowing its implementation. The class `Iterator` inherhits from `public std::iterator<std::forward_iterator_tag,std::pair<const K, V>>` and `ConstIterator` inherits from `Iterator`. As far as concern the class Iterator, its member is a private pointer to a node and we overload the following operators `==` , `!=` , `++ prefix`, `++ posfix`, `* and ->` . The class ConstIterator, instead, only overload the operators * and `->` that return a const reference to a `const std::pair<K,V>` and a pointer to a `const std::pair<K,V>` respectively, by calling the operators inherited from `Iterator`. This implementation allows to avoid as possible code duplication.

3 Methods

- `Iterator find(const K key) const noexcept` is a public method of the class BST that helped by the private method `findHelper()` allows to find a Node through the key and returns an iterator to it.

- `Iterator begin() const noexcept` returns an itereator to the begin of the tree (node with minimum value of the key into the tree).
- `Iterator end() const noexcept` returns an itereator to the end of the tree (node with maximum value of the key into the tree). The end of the tree is implemented as an iterator that points to `nullptr`.
- `ConstIterator cbegin() const noexcept` returns a constant itereator to the begin of the tree (node with minimum value of the key into the tree). In practice it is a pointer to a const Node such that i can have access to it but i cannot change its value.
- `Iterator end() const noexcept` returns a constant itereator to the end of the tree (node with maximum value of the key into the tree).
- `bool add(const K key, const V val)` is a public method of the class BST that, helped with the private insert(), adds new nodes to the tree.
- void clear() noexcept is a public method of the class BST that delete the entire tree.
- void balance() is a public method tha balance a tree and it calls the private methods `storeBST`,that stores the nodes of the old tree as std::pair in a vector, `isBalanced` that returns a boolean that indicates if the tree is balanced, `height` that returns the height of a subtree, `balanceHelper` that build the new balanced tree by using the vector of std::pair previously built up.

4 Testing

To test the program we implement a `test.cc` with different tests for all the funcnion; we use also intermediate tests by printing partial results inside the methods and we check the memory leaks by running *valgrind*.

5 Performances

In this section we present performances of the method `find` that we implemented in `BST` and the relative methd in `std::map`. Both the methods search a node inside the tree by a key. For this purpose we use the library `chrono` and in particular its method `std::chrono::steady_clock::now()` which returns the current time. Moreover to fill the `BST` and `std::map` whith random values we use the methot `rand` that allows to generate just over 2 billion positive integers.

Performance of `find` on unbalanced and balanced tree In this section we present performances measured on our implementation of BST. To do this we:

- write a `main.cc` which first build an unbalance BST filling it with random keys and values and measure the time to find a value, and then balance it and measure the time to find a value into it.

- write a bash script that for a list of values (indicating the size of the tree) compile the main, define a preprocessor value `NUM` used inside main to indicate the size of the tree and run the same executable five times to obtain some statistics.

The results in 1 show that find() in both balanced and non balanced BST cases follows a O(log(N)) behaiviour, but with better results with balanced BST. This is due to the implementation of the data structure which is ordered and to the implementation of the method that exploits this property and at each step of the search it half the BST and it performs in average log(N) comparisons and at most O(N).

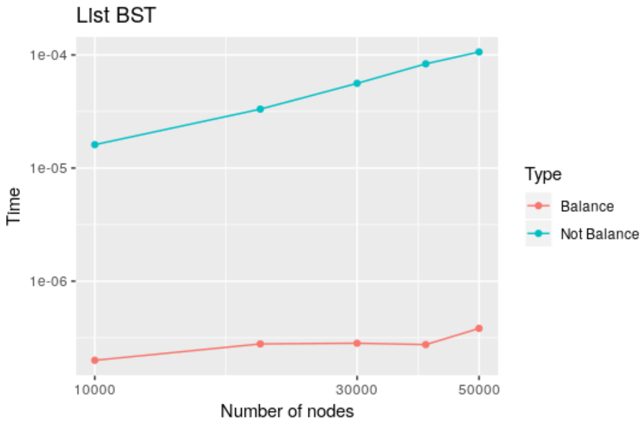
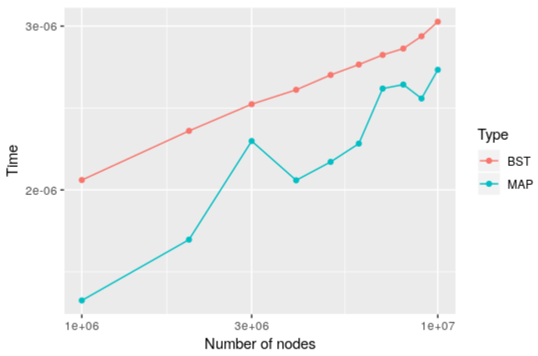


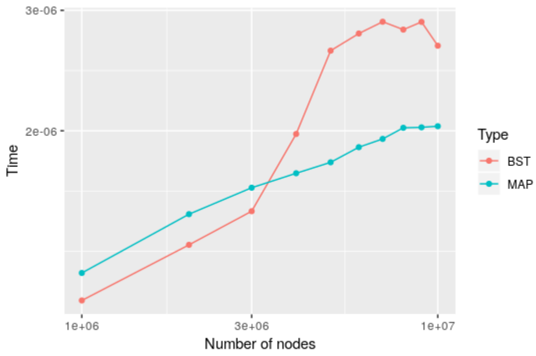
Figure 1: Time to find and element - BST

Comparison between `find` on balanced tree and `std::map` To compare our BST implementation and `std::map` we use the same approach of the proevious case ab we obtain the results shown in 2a. The plot point out the better efficiency of the `std::map` method `find()`, which performs three oder of magnitute better than our implementation.

Comparison between `find` on balanced tree and `std::map` using -O3 GCC compiler optimization The last part of this section regards the comparison of the former case but by use -O3 GCC compiler optimization; This type of optimization allows different types of optimizations shown in [1]. The results we obtain are shown in 2b. Also in this case our BST implementation it's slower but from the coparison between the not optimized and the optimized version, emerges that the former is faster than the optimized version. Moreover in this last case the performances seem not to follow a logaritmic behaiviours with repect to the number of nodes.



(a) Not using -O3 GCC comiler optimization



(b) Using -O3 GCC comiler optimization

Figure 2: Time to find element - BST vs. std::map

References

[1] `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`