Binary Search Tree

ppulcini@sissa.it, macastel@sissa.it

2020-2-18

Contents

1 Introduction

The aim of the project is to implement a templated Binary Search Tree in c++. Then we compared its performance with class `map` provided by the standard library. The code is documented using Doxigen. The work is organized in two directories: in the *include* directory there is the templated classes and the implementation of the methods and in *test* there is the `main.cpp` used to test the code.

2 Code

The Binary Search Tree is composed of three templated classes: *bst*, *node*, *iterator*. We decided to create three different headers file for the three classes because we use different templates. Then all the method of the class *bst* are implemented in a separate file `methods.h`.

2.0.1 Node

Node is a `struct` because in this way all its members, variables and methods, are public for the bst tree to use. It is implemented outside the class *bst* because we decided to use a different template: it has just one template `T` , that is the type of the variable `elem` , since we use `std::pair<const TKey, Tvalue>` as type of the elements in the node. The key inside Node is set to constant in order to prevent any user to modify its value and thus alter the chosen inner structure of the tree, that in the default case is order with nodes having lower keys value on the left and higher values on the right. The Node object is constructed with those attributes:

- two `unique pointers` to the right and left children, to facilitate memory management: each node is the only responsible of its children, has the exclusive ownership on its children.
- a `raw pointer` to the parent that helps to navigate through the tree
- a variable `elem` of type `T`, where `T` is an `std::pair<const TKey, Tvalue>`

2.0.2 Iterator

This class implement the concept of forward iterators, which are a generalization of pointer and allows to iterate through the tree without exposing the underline data structure. It is templated with two templates:

- `node_t`that is the type of the node
- `C`that is the type of the pair passed (it could be `const` or not)

In this way we can implement only a templated class for Iterator and const Iterator, which differ in the type of the pair, a const Iterator if deferenced returns a const pair and so the user can not modifies the value, on the other hand Iterator allows the user to modify the value. In this class we overloaded the operators:

- `*()` which returns a `std::pair<>` (`const` or not for const Iterator and Iterator respectively) from the node
- `++()` which returns an iterator to the next node with respect to the order of the keys
- `++(int)`

2.0.3 BST

We implemented the binary search tree having three templates, respectively:

- `Tkey` for the key type
- `TValue` for the value type
- `TComparison` as the custom comparison operator between the nodes of the tree

Firstly, some aliases were created to facilitate make the usage more intuitive and significant.

The binary search tree has 2 members:

- `std::unique_ptr<node_type> root` which is a smart pointer to the uppermost node (the root) that represents the tree.
- `TComparison op` as the custom comparison operator between the nodes of the tree, which can be initialized via the tree constructor like this: `bst<int,int,std::greater<int>> mybstree{std::greater<int>{}};`

Other possible constructors are: empty `bst() noexcept = default` and with a pair `std::pair<const TKey,TValue>`

2.0.4 Methods

Here we very briefly discuss some of the most significant methods

2.0.4.1 Deep Copy methods

Since we want to perform a deep copy (element by element or better node by node) we implemented a recursive function `copyelement(tree.root)` that starting from the `root` insert recursively the found nodes into the target tree

2.0.4.2 Find

Find a given key. If the key is present, returns an iterator to the proper node, end() otherwise. We created a main function `_find` that return a pointer to the node, and two functions that wraps around

it an `iterator` or a `constant iterator` in order to avoid code repetitions.

2.0.4.3 Begin

Return an iterator to the left-most node accordingly to the in-order tree traversal procedure. Also in this case we built a main function `_begin` that returns a pointer to the right node and the other variation functions that wrap around it either an Iterator or a constant iterators.

2.0.4.4 End

Just a constant or non constant iterator to `nullptr`

2.0.4.5 Insert

The function returns a pair of an iterator (pointing to the node) and a bool. The bool is true if a new node has been allocated, false otherwise (i.e., the key was already present in the tree).

Same strategy as before, the private function `_insert` is called by two functions, depending if the value to look for is an `lvalue& x` and so we have to perform a copy or an `rvalue(&& x)` and so we can save up time by using a move.

The function `std::forward<T>(x)` is used to distinguish the cases.

2.0.4.6 Emplace

In order to insert a node starting from a couple of values, we just use `std::forward<T>(x)` to forward the `args` passed to `std::make_pair` and then perform the insertion of the result

2.0.4.7 Clear

Since root is a `smart pointer`, to empty the tree is enough to call `root.reset()`

2.0.4.8 Balance

We first iterate along the tree and pushes the values of the nodes to an `std::vector` . Secondly, with the help of a private recursive function we perform an ordered insertion.

```
1) Get the Middle of the array and make it root.
2) Recursively do same for left half and right half.
   a) Get the middle of left half and make it left child of the root created in step 1.
   b) Get the middle of right half and make it right child of the root created in step
```

2.0.4.9 Erase

Removes the element (if one exists) with the key equivalent to key. We covered four different case: if the node erased is the root all its left subtree is assigned to the leftmost child of root's children, if the node to erase is a left or right child and finally if it is a leaf.

2.0.4.10 Overloaded Operators

We overloaded the operators:

- << to print both the ordered (as requested) list of the trees and draw the tree
- ++ In order to be able to iterate across the tree (in one direction)
- `operator[]` to perform a lookup (if the key is found) or an insertion (if the key is not found)

As well as the move and copy assignment.

3 Benchmark

We compare our binary search tree with std::map from Standard Template Library. `int` was used for the keys keys.

For each N we averaged the time to find one element for all the possible positions in the balanced tree.

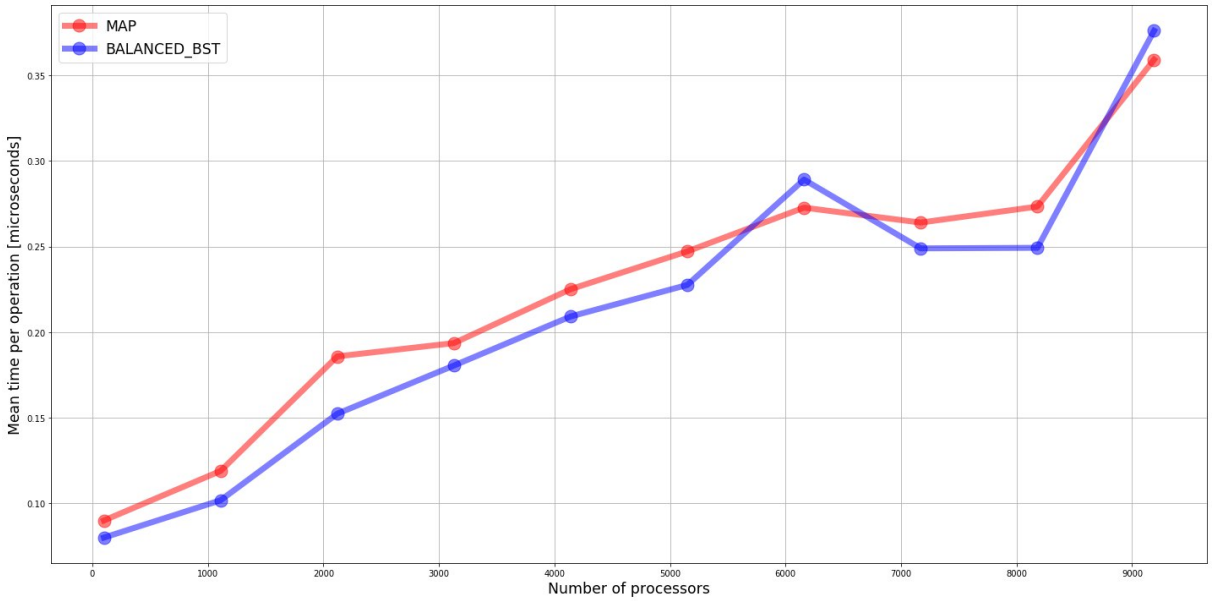

Figure 3.1: Benchmark: A lot of noise is present in this plot but we can observe that time goes as $O(\log 2N)$ and that out implementation (with `-O3` implementation) seems to keep up with the standard one