AP Exam - Binary Search Tree

Amadio Simone, Patrick Indri

February 17, 2019

1 Introduction

The aim of the project is to implement a binary search tree. Its performance should then be compared with `std::map`.
The tree is templated on key and value type. A third template allows user defined class to be used as key, if a compare structure is provided. As default, the tree uses `std::less` to compare two keys.

The code is organized in two folders: `include/`, which contains a header file, and `src/` which contains the implementation of the binary search tree and the `makefile`. The latter should be used to generate the executables to test and benchmark the implemented class. Please follow the instructions on the provided `README.md`.
`test.cc` is used to test the functionalities and features of our class, using `std::cout` to show what method is being called and verify its validity, verifying error handling as well. Furthermore, we tested the correct behaviour of our `BSTree` with user defined classes such as `CustomKey` and `CustomKey_Explicit`.

2 Classes

- `BSTNode`: binary search tree node. Located on a different namespace with respect to the tree, this class implements the idea of a generic node in a binary search tree. It is implemented in a different namespace so that it is not exposed to the user, who cannot in any way access or modify by chance this class and its methods.Unlike

the tree, the node is not templated on the compare function: this is another reason not to nest it in the BSTree class.

Every node is defined by its members: a content, which is an `std::pair` of templated key and value, two unique pointers to children nodes (namely "left" and "right") and one "parent" raw pointer. The key is implemented as a `const`, since it should not be modified in order to preserve the ordering of the tree and not corrupt it.

- `BSTree`: binary search tree. It is completely identified by its members: a unique pointer to `BSTNode` which represents the root of the tree, to which all of the other nodes are appended. We decided also to add the size of the tree as member, considering it can be useful.

- `Iterator`. Its only member is a pointer to `BSTNode`. Through the overloading of the operator ++, allows cycling through the nodes of the tree in key order. If deferenced, returns the content of the node it's pointing to.

- `ConstIterator`. Similar to the iterator, but can be used inside `const` functions.

- `Error`. Auxiliary structure used to throw and catch errors at run time.

- `CustomKey` and `CustomKey_explicit`. Auxiliary structures used to test tree behaviour on custom keys and with custom ordering.

3 Main methods

- `position_of()`. Auxiliary private function used both by the `insert()` and the `find()` methods. It takes as argument a key: if the key is found within the tree, it returns an iterator pointing to the node in which it's contained. If the key was not already present in the tree, it returns an iterator pointing to the to-be parent node.

- `find()`. It takes a key as argument and returns an iterator pointing to the node the key is contained in. If the key's not found, it returns `cend()`.

- `insert()`. Public method: it takes a key and a value and inserts a new node in the tree. Key and value can be passed both separately or contained within an `std::pair`. If the key chosen is already present, it does not override the previous content. The method returns a boolean which is `True` if a new node was appended, `False` if the key was already present.

- `print()`. It prints the content of all the elements of the trees using the overload of the operator «. The content is printed according to key ordering.

- `balance()`. Balance the tree. It creates a vine (linearized version of the tree, similar to a linked list) ordered by key. After inserting the median key, it recursively bisects the vine, inserting the median element of each bisection.

- `operator[]`. It takes a key as argument and returns the corresponding value. It's implemented in two different versions: `const` and non-`const`.

  In the `const` version, searching for a non-existent key throws an exception, since the tree is not supposed to be modified.

  In the non-`const` version, searching for a non-existent key provokes a new node to be created, containing the given key and the default value for the value type. This new insertion is advised to the user through a printed message.

- `operator==`. Returns `True` if the sizes of the trees and the content of every node are the same. It does not check the structure of the tree (insert-order dependent). If the same structure is needed, balancing the trees whose `==` returned `True` will achieve the aim.

Copy and move semantics for the `BSTree` were implemented as well. For more details on the implementation of the classes and their methods, please refer to the Doxygen documentation, provided in the `doc` folder.

4 Custom key ordering

With custom classes as key, an ordering must be provided by the user. Testing our code we verified that an user can provide such ordering in two different ways:

- Define a functor to be passed as the third template in defining `BSTree`.

- Overload the operator less than "<" with respect to the key class. Our code uses as default std::less, which, unless specified, exploit the "<" operator. In this case, there is no need to pass anything as a third template.

The user should be aware that if two keys are not greater (`<`) nor smaller (`>`) with respect to each other, in this class they will be considered as equal.

5 Benchmark

`benchmark.cc` provides the code we used to benchmark our `BSTree`.
We tested the effectiveness of our `find()` function and compared it with `std::map`. We tested both on a random unbalanced tree and on its balanced conversion. We chose double variables as keys. First, we populate a `std::vector` with an increasing sequence of doubles, in interval $[0, 1]$. Then, we randomly shuffle the vector: this procedure generates a random vector of doubles with no repetitions. We finally insert each element in

a `BSTree`. This procedure should result in randomly generated trees.

We used the `find()` function to search through all the keys in the data structures and sum up the time taken. For each dimension we took the average performance over 5 random trees, in order to minimise fluctuations. The results are shown in the following plots.

Fig. 5.1: On y axes, $log(time)$. Times in μs. Mean of 5 runs for each number of nodes.

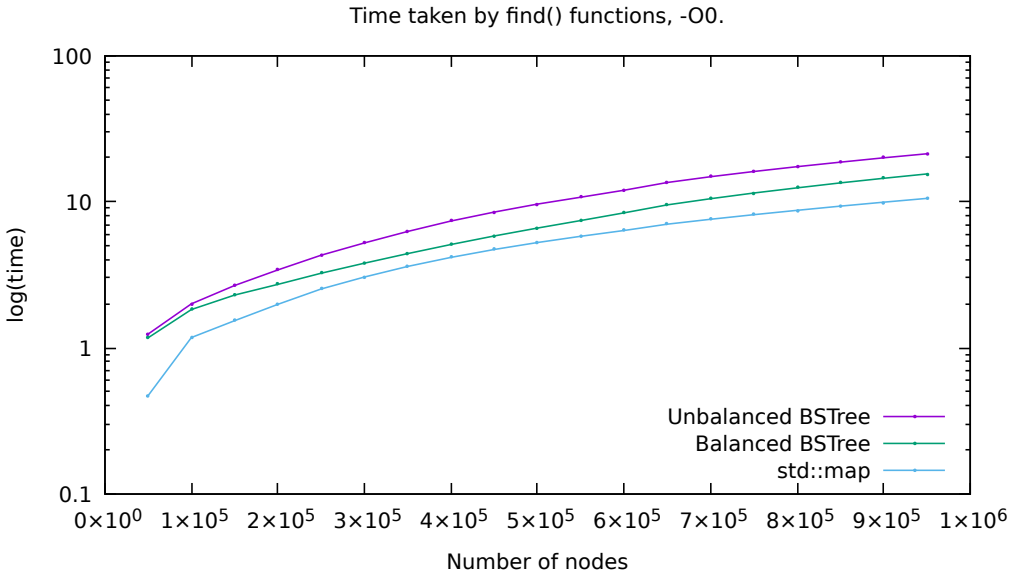
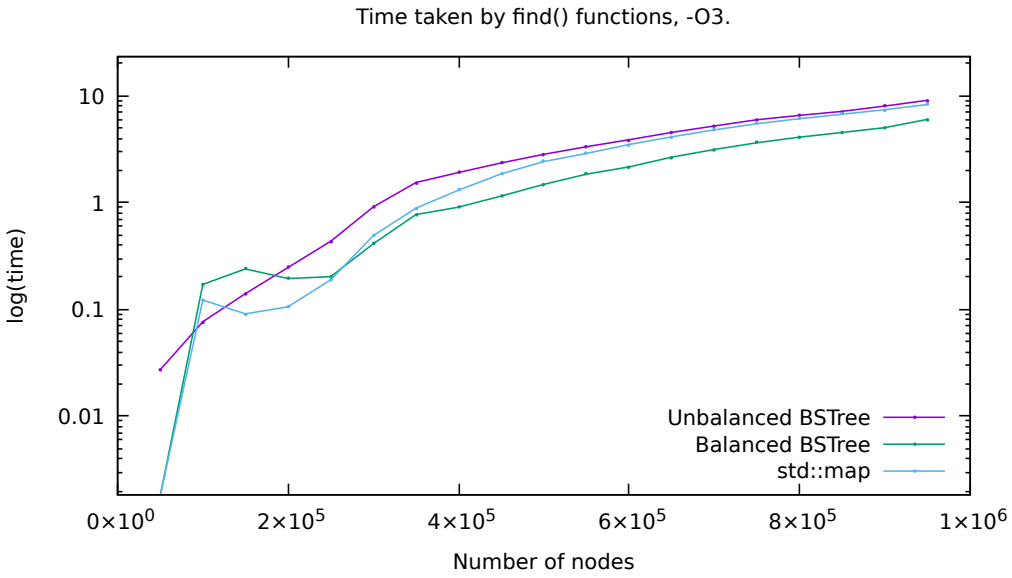


Fig. 5.2: On y axes, $log(time)$. Times in μs. Mean of 5 runs for each number of nodes.



We ran our test both on `-O0` and `-O3` optimisation levels. It seems that using `-O0` the `std::map` is faster than our tree, probably because it's implemented in an optimised fashion. Using `-O3` on the other hand, our code runs faster, probably because our simpler code can be more easily optimised.

We expect the balanced tree and `std::map` to have a $log(N)$ complexity for the find function, resulting in a linear behaviour in the plots. We can see that for a sufficient number of nodes, this hypothesis seems to hold.