# Advanced Programming - Final Project

Nicola Meneghini, Niccolò Rossi, Lorenzo Fresco

February 17, 2019

**Abstract**

The purpose of this project was to implement a Binary Search Tree using the C++ programming language and to test the performance of the lookups against `std::map`.

## 1 Introduction

A Binary Search Tree (BST) is a data structure such that, starting from the "root", each node has up to two child nodes (left and right child). Each child can either be a "leaf" (a terminal node) or itself the root of a sub-tree; nodes hold a "value" and are ordered according to "keys", such that at every point the left sub-tree contains only nodes with keys less than the parent node and the right sub-tree contains only nodes with keys greater than the parent node. A BST allows for a faster items search than other containers (such as a linked list). Specifically, this is the case when the tree is balanced, i.e. when there is roughly the same number of descendents on both sides of the "root", and the same definition recursively applies to its subtrees.

These are the basic rules we kept in mind while writing the code, which we placed into two different folders: `include`, which contains the header file `BST.h` and the `BST.cpp` file, and `src`, where we placed two files which are aimed to verify the correctness of the binary search tree implementation. Specifically with `test.cpp` we tested the call to the functions, while in `benchmark.cpp` we tested the time taken to find each key in the tree.

In what follows we describe what are the classes and functions we used throughout the code. We also here point out that a wide use of smart pointers was made, as they allocate and free memory automatically.

## 2 Classes

- `BST`: the actual Binary Search Tree implementation. It is templated both on the key and the value but also a third template was added in order to compare keys between different nodes. The BST has three private members: the `node` structure described below, the comparison operator and a unique pointer that points to the "root" Node of the tree to which all the other Nodes will be appended. They were made private because they deal with the fundamental tree structure and it should not be a concern of the user.

- `Node`: the building block of the tree. This structure is the implementation of the concept of a generic node in a Binary Search Tree. It is templated only on the key and value types. Every node is defined by its members: the `std::pair` key-value pair, two `std::unique_ptr` pointing to two children (one left, one right) and a raw pointer to the parent Node.
- `Iterator` An iterator used to identify an element of the sequence. Through the overload of the `operator++` it allows us to travers the tree in key order.
- `ConstIterator` Similar to the `Iterator`, but can be used as inside `const` functions.

# 3 Principal Member Functions

- `Node` The Node `struct` has only one constructor and a default destructor.
- `Iterator`: this class contains the constructor as member functions. Besides that we overloaded the operator ++ allows a tree traversal in key order (i.e. for each iterator the following element in the sequence is the one identified by the following key).
- `BST` the main methods implemented for this class are:
  - `insert` which inserts a new Node in the tree.
  - `balance` which takes a BST as argument and then returns a new balanced tree. It makes use of a `std::vector` to store the elements of the tree in order and then starting from this ordered vector it creates a balanced BST.
  - Copy and move semantics were added to the class.
  - `find` which takes a key value as argument and returns an iterator to the node with that key. This function is the one used in the benchmarking phase of our project.
  - `begin` and `end` which return respectevely the first and the last element of the tree. A `const` version of the two was also provided.
  - overload of the operator `<<` so that it possible to print the couples (key, value) in key order.

# 4 Benchmark

Basically the aim of the tree is to find its elements in relatively short time, namely $O(log(N))$, with N the size of the tree. To check if this was the case, we timed the the function find on always growing trees. We then compared this time on the same tree sizes but after they were balanced. What we obtained is a great improve of performances, which remarkably became comparable to the STL function `std::map` as we can see in Fig:1 We obtained similar results for different key and value types.

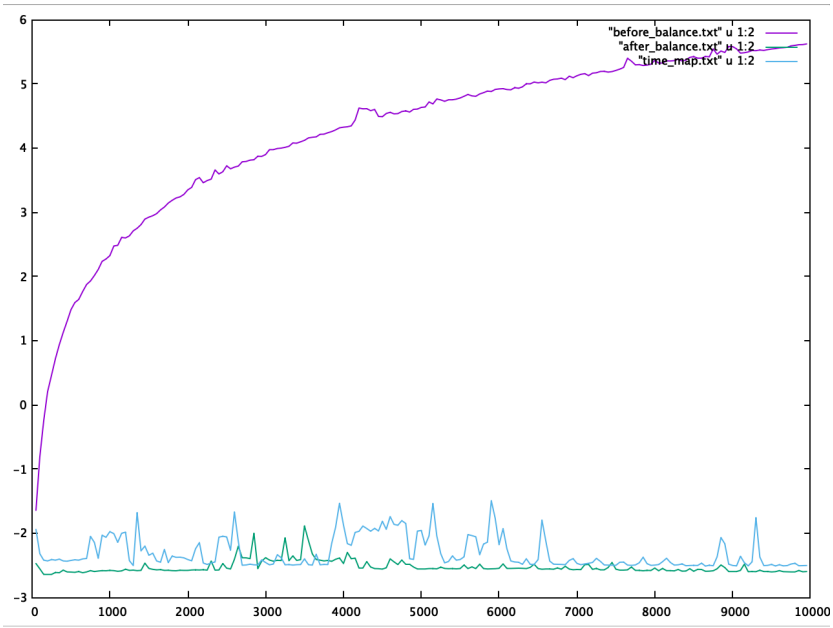


Figure 1: Comparison of the transformed time between the unbalanced tree, balanced tree and `std::map`

# 5 Result discussion

What we obtained is the desired time performance, meaning that the tree implementation is highly optimized. Finally note that also there are no memory leaks as checked with Valgrind.