# Advanced Programming Exam – Report

This report illustrates the implementation of a binary search tree, done via the use of three classes. The class **bst** is the main one, and it relies on the struct **Node** and the class **_iterator**.

## Struct Node

It is templated on the type T of the value contained in the node and has four member variables: *value* of type T, *left* and *right* which are unique pointers to the left and right child (initialised to nullptr unless otherwise specified), and a raw pointer *parent* which points to the parent node. The struct has a default constructor and destructor, as well as two custom constructors, which create a new node starting from a pointer and either an rvalue reference or a const lvalue reference to T. There is finally a copy constructor, called by the class bst to perform a deep copy.

## Class _iterator

Used to iterate through the tree, the class is templated on the type of node, on cmp which represents the type of comparison between nodes, and on T, the value contained in the nodes, which allows to define both iterators and const_iterators from the same class.

It has two private variables, a pointer *current* pointing to the current node, and a comparison operator *op*. The class has default constructor and destructor, and a custom constructor initialising *current* to a given pointer *p*, as well as copy semantics. In addition I have implemented the reference and pointer operators, the comparison operators ==, !=, and the prefix increment and postfix increment operators. There are also a number of utility functions which allow for easier navigation of the tree.

## Class bst

The class has three templates: the type of the keys, the type of the values and the comparison type, defaulted to std::less<int>. Each tree has the private variables *root,* a unique pointer to the root node, and *op*, the comparison operator.

The class has a default constructor and destructor, and three custom constructors. The first takes a pair key-value, while the second accepts both a pair and a comparison type. The third on the other hand builds a tree from a given vector of pairs by calling the public function **build_from_vector**. I have implemented copy semantics, which perform a deep copy, and move semantics, as well as the following public member functions.

- **Insert**: Insert a new node in the tree if the key is not already present, otherwise it leaves the value unchanged. There are two versions so that it can take as input either a const lvalue reference or an rvalue reference to a std::pair.
- **Emplace**: implemented using a variadic template, it takes as input either a std::pair or a key and a value in order to insert a new node.
- **Clear**: deletes the content of the tree and releases the memory. It resets *root* to nullptr, which in turn resets the children and ultimately empties the tree.
- **Begin**: overloaded function, it returns either an iterator or a const_iterator pointing to the first element.
- **Cbegin**: returns a const_iterator pointing to the first element.
- **End**: overloaded function, it returns either an iterator or a const_iterator pointing to one past the last element.
- **Cend**: returns a const_iterator pointing to one past the last element.
- **Find**: Overloaded to return either an iterator or a const_iterator, it starts from the root node and moves down by comparing the current node's key with the given one. It stops if the key is found, returning an iterator to the node, otherwise it returns **end**.
- **Balance:** it balances the tree. It saves the node values in a std::vector in an ordered way, then **clear**s the tree and calls **build_from_vector** to rebuild the tree in a balanced way.
- **Build_from_vector:** given a vector of pairs and a starting and ending point, it finds the median element, inserts it in the tree and then recursively calls itself on the two subvectors to build the whole tree.

- **Erase:** searches for the node with the given key, and if found erases it. If the node is a leaf node simply its parent's left or right pointer are reset to nullptr, otherwise the tree, minus the node to delete, is copied into a vector, then the tree is **clear**ed and rebuilt from said vector.

- **Operator []:** Returns a reference to the value that is mapped to the given key, performing an insertion with defaulted value if the given key is not found. The operator is overloaded in order to accept either a const lvalue reference or an rvalue reference to key type.

- **Operator <<:** friend function which prints the tree to an output stream.

- **Get_root:** utility function that returns a pointer to the root node.

- **Count_nodes:** Utility function that counts the number of nodes in the tree.

## Benchmark

In order to benchmark the efficiency of the code, I compared the time it takes to look for keys through the **find** function of the binary search tree with the time needed to find keys in a std::map. In particular, given a size the program benchmark.cpp creates a balanced bst and a std::map with the given number of node, then performs a find operation for all keys on both the bst and the map. The time is measured through the std::chrono high resolution clock. The program also compared the time it takes to build the tree with the time it takes to build the map.

The program was run with different number of nodes, ranging between 5'000 and 50'000. The results concerning the **find** function are reported in the following figure: as expected, the map's times are better, although not by much. On average std::map outperforms the bst tree by 2 to 3 times.

On the other hand, the building times of the tree are on average more than 1000x slower than those of the std::map. For example, for 10'000 nodes the building takes 9.12 seconds against 0.008 seconds, whereas for 50'000 nodes it takes more than 4 minutes to build and balance the tree against 0.06 seconds of std::map. This is probably due to the fact that the std::map rebalances itself after every insertion, whereas bst does not and therefore, before balancing it, produces a highly unbalanced tree, thus achieving linear complexity in the number of nodes. Being there such a huge disparity, plotting all the times for a comparison would have been meaningless.

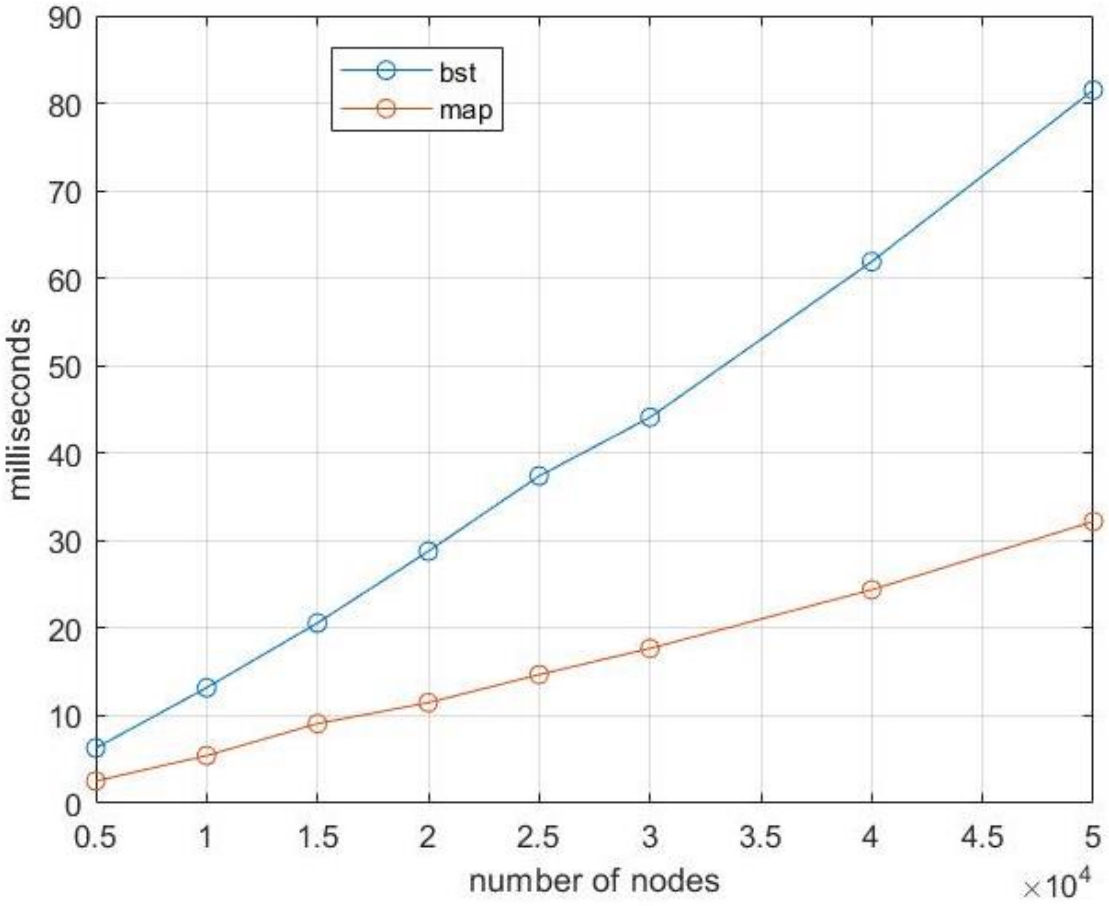


**Fig. 1**: Comparison of **find** times between bst and std::map