Report on Exam Assignment
Advanced Programming course

Giulio Dondi

January 11, 2021

Section 1 - Binary Search Tree

The assignment involves the implementation of a *Binary Search Tree*, an ordered data structure composed of nodes containing a key and an associated value, where any node may be linked to at most two children nodes to the left and right. The tree starts from a root node, and every node must respect a general ordering rule: given some operator to compare the node keys, the key contained in the node must be greater than all the keys in the nodes to its right and smaller than all the keys to its left.

A tree can be described by its *size*, i.e. the total number of nodes it contains, and the *depth*, i.e. the maximum number of nodes to be traversed in order to reach the bottom. We could define an "ideal" binary tree as a tree in the shape of a triangle: every node is connected to two children except for those at the bottom depth layer which all have zero children. Such a tree has size $2^d - 1$ where d is the depth.
The shape of a tree depends not only on the incoming keys but also in the order they are added in the tree: as a result is its possible to build trees that are *balanced* with respect to the root node or *unbalanced (skewed)* to one side. We define the notion of balance: a tree is balanced if the children of every node are the sub-roots of a balanced sub-tree, and the depth of the left and right subtrees differ by at most 1.

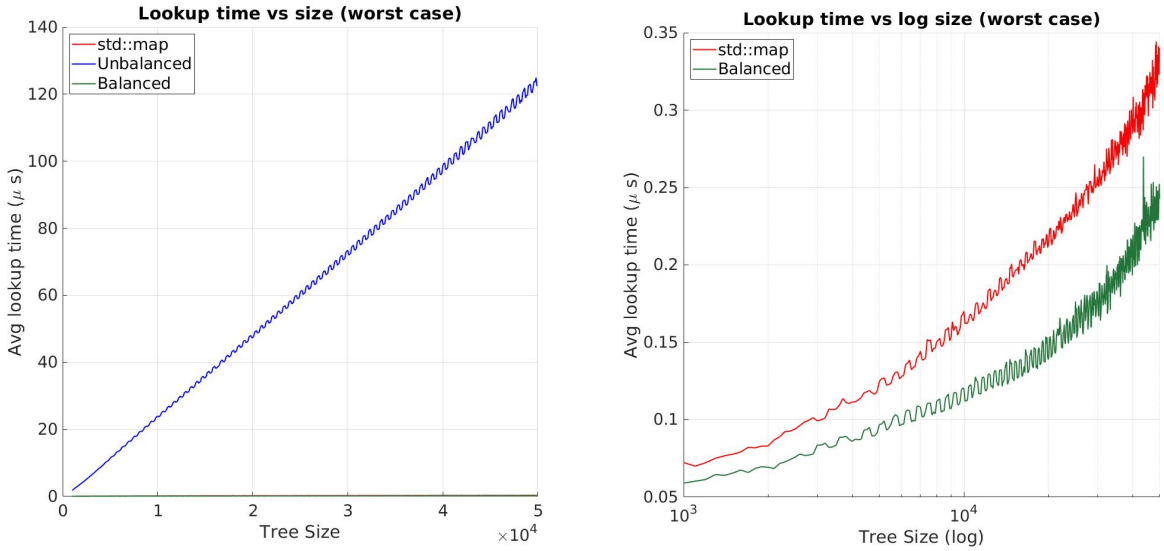
The binary tree has been implemented in `C++` as a class `bst` which is templated on 3 types: the type of key K, the type of data contaned T, and the comparison operator CMP. A separate class `bst_node` was created for the tree nodes, this class is templated only on K and T. Internally, the key and data types are merged as a single type `std::pair< const K , T >` in order to let the user change the value of a node but not its key, since then the ordering of the tree may be broken.
The nodes also contain pointers to the left and right children as well as the previous node: the children nodes use smart pointers in order to retain ownership of the memory locations used by the children; since each node is a child of its previous node the pointer to the previous node is a simple one in order not to have an ownership conflict.
The tree class only contains a smart pointer to the root node and a tree size variable, conversely the previous pointer of the root node is set to `null`. The following public member functions have been implemented:

- **Constructor:** The default constructor initialised the root to null and the size to zero
- **Move:** Default move constructor and assignment are used;
- **Copy:** The copy assignment takes the input tree class given as argument and moves it onto the currrent tree. The copy constructor performs a deep copy by allocating a new root node using the data from the corresponding node, and then recursively repeating this for the left and right children nodes;
- **Clear:** Zeroes the tree size and resets the smart pointer to the root node which, in turn, destroys every node in the tree;

- **Iterators :** Objects that point to a certain node in the tree and are able to traverse the tree in the correct order, as well as access the key and value contained through de-referencing. Iterators are templated on three types: key and value type, as well as the assembled pair of key and type which may be passed as constant or not. In addition, there are methods that return iterators to the beginning and end of the tree;
- **Size:** Returns the size of the tree;
- **Depth:** Calculates recursively the depth of the tree by obtaining the depth of left and right sub-trees at each node, and finally finding the maximum;
- **Insert:** Inserts a given a pair of key and value in the tree, by navigating down the tree to the right place and constructing a new node if the key does not exist. An iterator to the node in question is returned, as well as a flag that signals if a new node has been allocated;
- **Emplace:** Similar to insert, except the input arguments are separate key and value, as opposed to a single pair;
- **Find :** Given a key value, navigates to its correct position and returns an iterator to the corresponding node. If the node does not exist, an iterator to the end of the tree is given;
- **Subscription operator[] :** Given a key value, searches the tree for it and returns a reference to the corresponding value, inserting a new node with default value in the right place if it doesn't exist. This lets the user change the value corresponding to a given key;
- **Put-to >> operator :** Outputs the tree in the correct order by using iterators, printing `( key , value )` of each node;
- **Balancing:** The recursive procedure to determine if the tree is balanced (described previously) is implemented. Before attempting to balance an unbalanced tree, it is copied locally so that the operation may be reverted in case of exceptions. Balancing happens in several steps.
  First, the tree is de-constructed and a vector of smart pointers to each node in the right oder is built. At each step, every node is "detached" from its parent node, meaning ownership of the node is transferred from the parent node to this vector, in such a way to always have a smart pointer owning an node in case of problems.
  Then, a new tree is built: given a vector of nodes the node in the middle is labelled as the root, the nodes to the left will form a sub-tree to be attached to the left of the root and same for the right nodes. A recursive routine initialises a new tree in this way.
- **Comparison operator:** A method to return a copy of the comparison operator used is provided. This is similar to the `key_comp` method found in `std::map` and lets the user implement custom comparator objects that, for instance, provide diagnostics on how often they are used;

Section 2 - Benchmark

The performance of the implementation of the bynary tree was tested against `std::map`, namely the average time to find a given element. An important difference between the two is that `std::map` is a *self-balancing* binary tree, meaning that the resulting map will always be balanced no matter the order the elements are inserted in. This is obviously not true for this implementation.
For this reason two cases are considered: the *worst case* where we insert as keys integers in both this tree and `std::map` in order from 0 up to a tree size N and the *shuffled case* where we take the same set of elements but scramble them using a random number generator before inserting. In both cases, a full lookup of all the keys is performed, taking the keys in scrambled order . The time is measured and averaged over the number of keys, for `std::map`, this tree unbalanced and the same tree after balancing.
Th data type chosen is `<int,int>` and `std::less` as the comparator. However, a custom comparator wrapper was built that also contains a running counter of how many times it is invoked by the tree or the map. The number of comparisons made to look for all the keys can be extracted by sampling this counter before and after the lookup.
We considered trees from size 1000 up to size 50000 incrementing in steps of 100, for each case three runs were performed and the times averaged.



(a) (b)

Figure 1: Performance of the binary tree implementation against std::map in the worst case scenario. 1a shows the linear scaling of the unbalanced tree while the blaanced and std::map curves are barely visible at the bottom. 1b is a semilog plot of only the balanced tree vs. std::map.

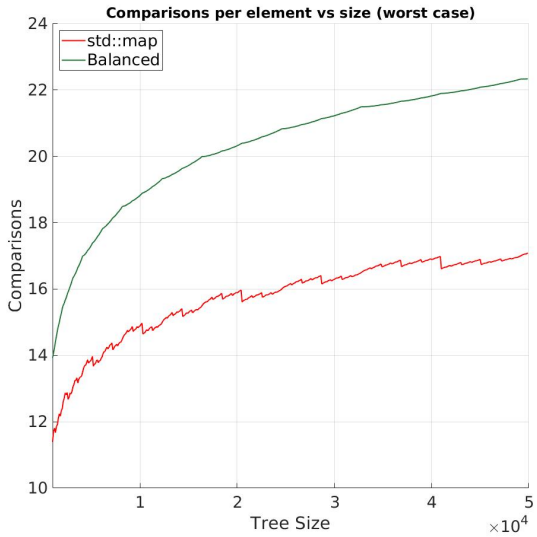
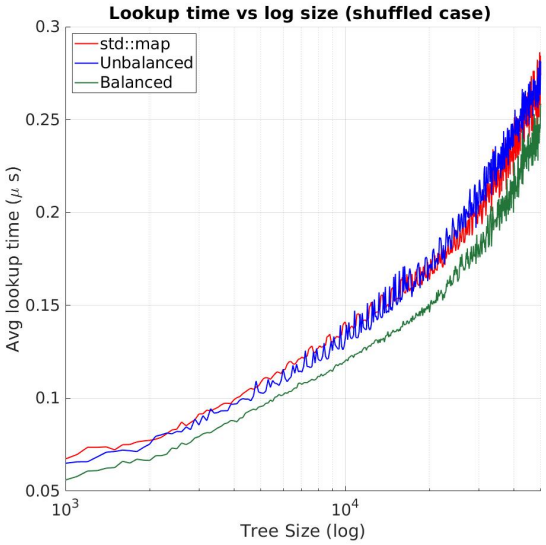


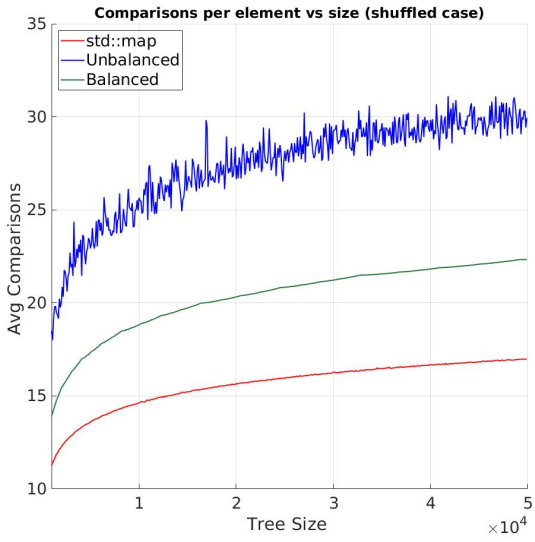
Figure 2: Number of comparisons per element performed during the lookup. Only the balanced tree and the std::map curves are shown.

The plots in figure 1 display the lookup times per element in the worst-case scenario. IN 1a the effects of a completely skewed tree are clear, as the time scales linearly for the unbalanced tree while the other two are barely visible near the x-axis. 1b is a semi-log plot which compares the balanced tree versus `std::map` in this case: this implementation of the binary tree is actually superior to the map with respect to this metric. Given a tree of size N, the number of steps to take to descend down the tree all the way to the bottom scales as $log_2 N$, we then expect the same kind of scaling for the average number of comparisons per element, this was tested by using the counter in the comparator and figure 2 confirms this prediction.
By the same token one would expect the average time per element to follow the same scaling, and thus one would expect to see straight lines in a semi-logarithmic plot. Figure 1b shows that this is only somewhat true up to about

$N = 5000$ and then it no longer seems to be valid at all, either for this implementation or for the map. This is perhaps due to the branch-prediction logic of the compiler, given that the lookup is essentially a series of nested if clauses in a loop: The compiler branch optimisations are apparently only efficient up to a certain size and fro mthere on, a penalty is paid that creates a time overhead.
Finally, from 2 it is clear that, although the balanced binary tree beats `std::map` in average lookup time, the map performs only about 75% the number of comparisons to find an element.



(a) Semilog plot of the std::map, unbalanced and balanced lookup times in the shuffled case.



(b)

Figure 3: Number of comparisons per element performed during the lookup in the shuffled case.

The same analysis is repeated in figure 3a for the tree and map initialised with shuffled integers. This time the unbalanced tree performs comparably to `std::map`, while the tree after balancing performs better once again, although by a lesser margin this time. Looking at figure 3b, the number of comparisons per elements once again scales logarithmically as expected. A large difference can be seen between the balanced and unbalanced trees, although `std::map` is once again more efficient in the number of comparisons.

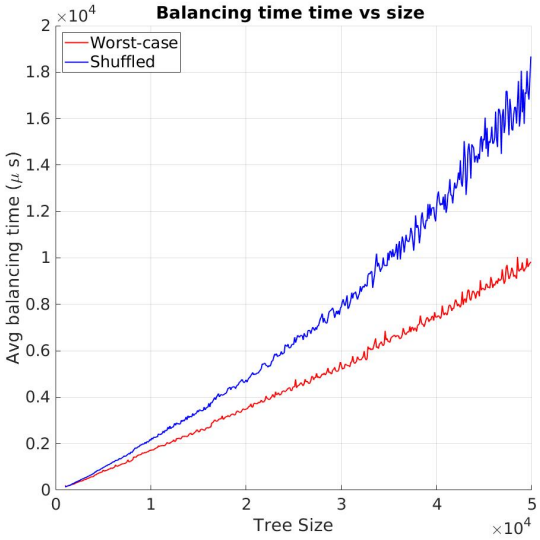


Figure 4: Balancing time for the tree implementation in the worst and shuffled cases

Finally, figure 4 displays the scaling of the time necessary to balance a tree in the worst case and the shuffled case. In the worst case the tree shape is self-similar and therefore is it expected that the time depends only on the number of elements to be handled, as evidenced. The shuffled case has a much less predictable structure and therefore the performance of the balancing algorithm worsens as the size grows.