## 1 Implementation

The header file `bst.hpp` contains three templated entities:

- `bst` class that implements a binary search tree by managing the root node and providing the interface for its manipulation. Its templates specify the types of the keys, the values and the type of the function object for key ordering
- `node` struct which implements a tree node; it references a parent node and manages two children, and it contains a data member defined by its template type
- `node_iterator` class that implements a forward iterator for accessing the tree's nodes data with an in-order traversal

Pointers to the managed nodes are wrapped in `std::unique_ptr`. Naked pointers to nodes are handled freely in `bst` and `node_iterator`, but never deleted explicitly or exposed; `std::unique_ptr::reset` is the only action that deallocates a node.
Only the second field of a node's `std::pair` can be modified from outside its `bst`, by dereferencing a non-const iterator, abstracting data access from the internal structure, as is the philosophy of the STL.
The use of managed pointers, however, results in recursive deletions when a `bst`'s root is deleted, as each node resets the pointers to its children.
This might cause a stack overflow; the `clear` method has been implemented so that the children are released in a post-order traversal.
`node::parent` member was added to allow a simple implementation of the iterator increment, as well as the node methods `leftmost` and `first_right_ancestor`. The `depth` methods were added for debugging and benchmarking purposes, although this reveals features of the underlying data structure.
`bst::key_comp` is analogous to the `std::map`'s method of the same name; it returns a copy of the tree's comparator and it was added to easily benchmark the comparisons performed.
Methods that allocate dynamic memory, whether directly or undirectly, were not marked `noexcept`. As there is no guarantee on the comparator's behavior, `noexcept` cannot be used on `bst` constructors (except the move one) and in the methods that use it.
Throughout the `bst` implementation, the focus has been on code reuse; the main cases susceptible of code duplication were

- finding either the node holding a key, or the parent candidate of a key; the two operations are provided by a single private method `find_parent_candidate`, which returns a pair with a node pointer and an enum identifying whether it contains the key, or which child would hold it if inserted

- methods that overload on forwarding references, such as `operator[]`, for which a private method forcing type deduction through an additional template parameter was provided
- const and non-const accessors, which presented two forms:
  - a new object is constructed from a managed resource, such as in `find`; in this case a private const method returning a non-const object was provided. The mutable accessor can invoke freely the private one, while the const accessor promotes the constness of the returned value
  - direct access to a member, such as in the node's `leftmost` and `first_right_ancestor` methods; as a possible return value is a pointer to the node itself, its constness cannot be demoted as in the previous case. A private static method was provided, templated on the node type, propagating the constness of the starting node passed as parameter. This comes at the cost of a second template instantiation, with the alternative being the implementation of the const version and the use of `const_cast` in the mutable one.

The balance operation performs a tree traversal through an iterator and moves the pairs to a `std::vector`; due to the traversal, the elements are sorted. Since the keys are constant, the move might only be beneficial for the values.
The `bst` is then cleared; recursively, if the vector is not empty, the middle element (chosen at offset $\lfloor size/2 \rfloor$) is moved from the vector to the emplaced in the tree. The recursive calls use the reduced vector ranges [begin, mid) and (mid, end).
Since the elements in each remaining range of the vector differ at most by 1, the resulting `bst` is balanced and its depth is $O(\log_2(size))$.
The post-order visit in `bst::clear` method duplicates the content of `node::first_right_ancestor`, since it must act before traveling to the current node's parent.

## 2 Worst case lookup

The worst case for a lookup is reaching a leaf node of maximum depth, while having to compare for both less and greater than each node's key found along the way. Thus performing $2 \times depth$ comparisons when the key being searched is greater than or equal to the righmost node and that node is the deepest.
Other than the bare computational cost of a key comparison, there is the

cost of accessing the nodes' children, which might be scattered on different cache lines, as well as possibly dynamic memory allocated by the key and required by the comparison (for example, a large `std::string`).
These costs vary depending on the actual interaction with the memory hierarchy, rather than being constant.
It is anyway expected, for the relation between the worst lookup execution and the bst's depth, to resemble a linear dependency.
Additionally, for a balanced bst and the self-balancing `std::map`, this linear dependency would translate to $O(\log_2(size))$ due to the depth being bounded logarithmically.

## 3 Benchmark for randomly distributed keys

The program `profile.x` measures the insertion and repeated lookups of uniformly distributed random keys in a `std::map` or a `bst`. In the latter case, a parameters whether to balance the tree and repeat the lookups.
The experiment aims to measure the behavior of different container types without enforcing a degenerate case for the unbalanced `bst`.
The default keys are `std::array<std::size_t, 1>`, their size can be controlled at compile time with the macro `KEY_SIZE`. Only the last element in the key is random.
The keys are generated from a uniform distribution; the seeds for the insertion and lookup distribution differ and since the range spans all the positive integers representable by `std::size_t`, very few hits are registered.
The number of comparisons per lookup are traced with a custom comparator class.
The program was compiled with `gcc` 9.0.3 and flags `-O3 -march=native -DNDEBUG`.
The runs were performed on a Linux machine with a Intel Core i7-7700HQ 2.8GHz (3.8 GHz TurboBoost), 6MB L3 cache and 16BG RAM.
Time values are the average of 10 runs for each container and size; sizes vary from $2^{12}$ to $2^{24}$, increasing by a factor of 4.

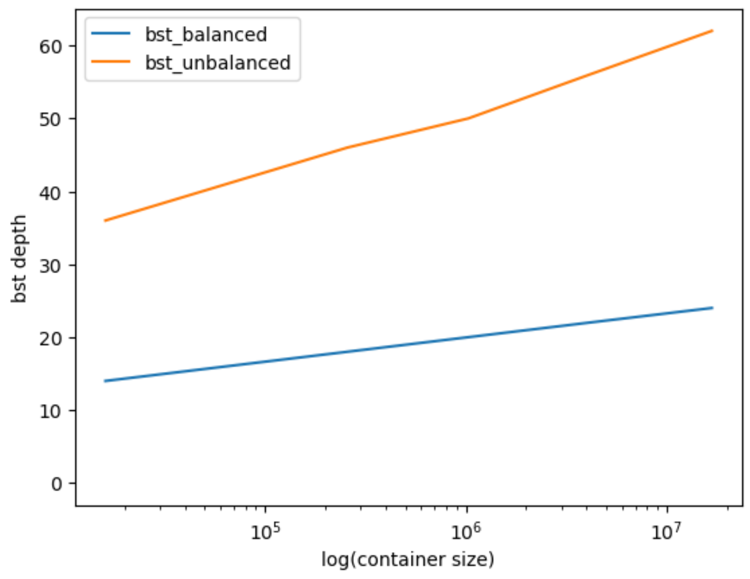

Figure 1: Depth of `bst` before and after balancing

With uniformly distributed keys, the depth of an unbalanced `bst` still roughly follows a logarithmic curve and its depth is roughly 2.55 times that of the balanced one. As is expected, balancing the tree yields a perfectly logarithmic depth.

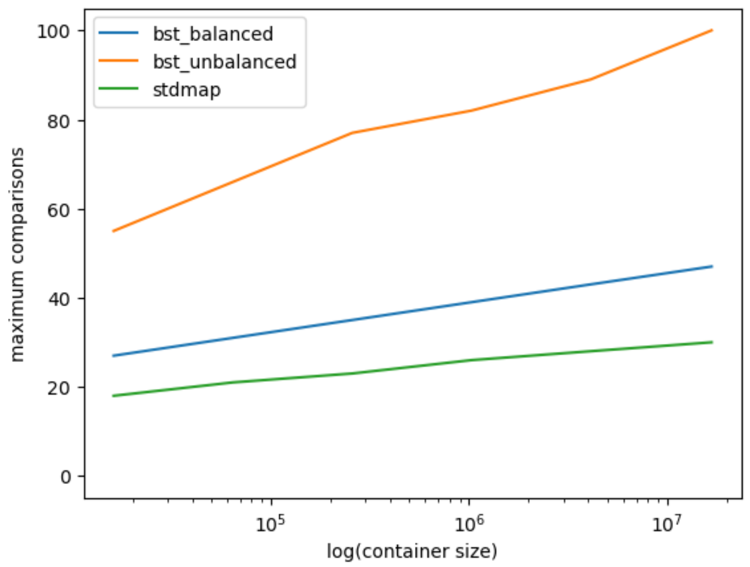

Figure 2: Maximum number of comparisons for a single lookup

The behavior of the maximum number of comparisons in a single lookup reflect the differences in depth between the two `bst`. Somewhat surprisingly in this metric, the self-balanced `std::map` consistently outperforms a balanced `bst`.
Variations of the experiment using the same seed for insertion and lookups and different sizes, give consistent results.
As the theoretical worst case for lookup comparisons is reached for the balanced `bst`, this suggests that the red-black tree is denser at the center, compared to the full or nearly full tree resulting from the balancing.

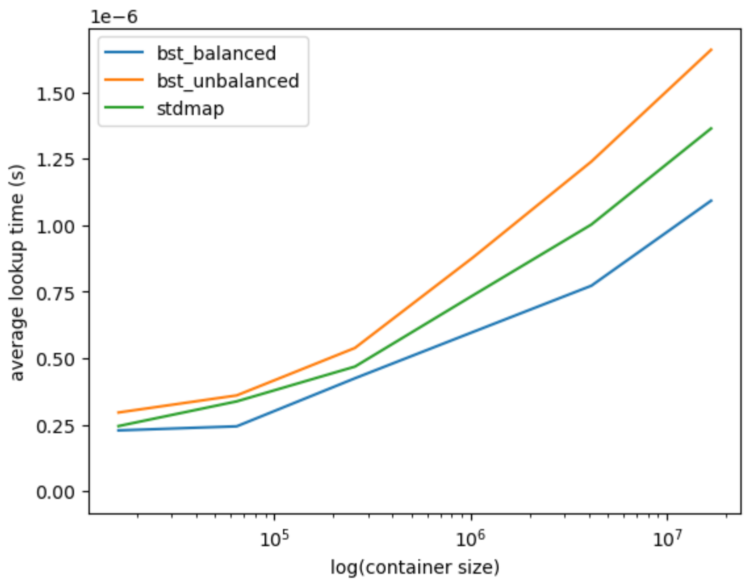

Figure 3: Average time for a single lookup

The average time for a single lookup is not perfectly logarithmic in the container size, rather resembling a parabola.
This is possibly due to the key's distribution, but is an unexpected result and possibly a hint of a defect in the experiment's design.
As expected, the lookup for unbalanced `bst` is the slowest, although not as much as the difference in depths would imply for the worst case, compared to the balanced trees.
The `std::map` is consistently slower than the balanced `bst`, again possibly due to the structure taken by the self-balancing tree and the key's distribution.
Overall, the difference on the single lookup time is under a microsecond at the biggest tested size: 1.66$\mu$s for the unbalanced tree, 1.36$\mu$s for `std::map` and 1.09$\mu$s for `bst`.
The unbalanced tree, on average, is 1.52 times slower than the balanced one and `std::map` 1.25 times slower.