

Efficiency of various forms of red-black trees

Hervé Brönnimann¹

Jyrki Katajainen^{2,*}

¹ *Department of Computer and Information Science, Polytechnic University
Six Metrotech, Brooklyn NY 11201, USA; hbr@poly.edu*

² *Department of Computing, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark; jyrki@diku.dk*

Abstract. Several state-of-the-art implementations of red-black trees are presented and evaluated. The techniques yield implementations of C++ standard-compatible `set` and `map` associative containers, which guarantee logarithmic worst-case performance, provide iterators with strict invalidation rules, and lower storage requirements to three, sometimes two, pointers per node. Lowering the storage requirements is important on modern platforms, where the efficiency of an implementation is more and more dependent on smaller storage and depends intricately on several factors, not all related to instruction count (such as efficient memory cache utilization). The code is entirely available online, including the various test drivers.

1. Introduction

Let \mathcal{U} be a universe of elements which can be compared. Elements of \mathcal{U} can be atomic or pairs, each consisting of a key and some satellite information associated with that key. A data structure D that stores a collection of elements drawn from \mathcal{U} and supports efficient insert, delete, and search operations is called an *ordered dictionary*. The collection stored may or may not contain duplicates. In an abstract setting, D maintains a dynamic collection of elements in sorted order. Ordered dictionaries are a central piece of any programmer's toolbox, and are provided as a facility in a number of high-level programming languages.

In C++ terminology [2, §23], an ordered dictionary is called an *associative container*. Each realization of the C++ standard library should provide four kinds of associative containers: `set` (elements atomic, no duplicates), `map` (elements pairs, no duplicates, ordering based on keys), `multiset` (elements atomic, duplicates allowed), and `multimap` (elements pairs, duplicates allowed, ordering based on keys). In the design of the C++ standard library, the main reason to support these four different dictionary variants is to allow

*Partially supported by the Danish Natural Science Research Council under contract 21-02-0501 (project "Practical data structures and algorithms").

a direct modification of satellite data, instead of forcing users to invoke a delete followed by an insert to make such an update. This feature is frequently employed in practical applications.

There are many data structures that could be used in the implementation of associative containers; almost any elementary textbook on data structures offers one. Most relevant alternatives are based on balanced search trees; hash tables cannot be used because of the element ordering. In his book [11, §13.6] Sedgwick gives a nice discussion of the possible data structures, and concludes by saying that red-black trees are the best suited for a general-purpose library implementation. Indeed, red-black trees are the structure of choice in many implementations of the C++ standard library. Actually, not many other data structures can fulfil the tight complexity requirements specified in the C++ standard [2, §23]. Also, it is known that the maximum height of a red-black tree storing n elements is bounded by $2\lg n + 2$ and in the average case the height is close to $\lg n$ [11, §13.4]. Since the height of a balanced tree storing n elements is at least $\lg(n + 1)$ [6, §6.2.3], there is not much room for improvements.

In this experimental study, we are interested in the efficiency of different modifications and extensions of red-black trees. One of the reasons for this study is the lack of literature on the many possible implementation choices. Many techniques are available (tricks-of-the-trade) but not always publicized, and the textbooks usually stick to an easier, less-efficient version of the data structure for ease of exposition. In comparison, industrial-strength implementations that are compatible with the specification given in the C++ standard are considerably more involved, and hence often only a single implementation is developed.

The purpose of this experimental study is to determine the range of various optimization techniques one can apply when implementing associative containers, and to evaluate their impact on the amount of storage required, as well the overall efficiency of dictionary operations. We based our implementation of red-black trees on the publically available SGI STL implementation [15] which is highly optimized—as confirmed by our preliminary experiments—and provides exception safety as discussed, for example, in [17, App. E]. The SGI STL implementation is a derivative of the HP STL implementation which is documented in detail in [9, §§13–14]. As in the SGI STL implementation, we specify a unique interface for red-black trees, which is then used for the realization of all four associative containers with appropriate adapters.

Our results are that it is possible to improve by as much as 20% in the runtime of typical operations upon the standard SGI STL implementation, by using compaction tricks. It is also possible to provide a set with logarithmic time random-access iterators. More surprisingly, only a very small portion of the code has to be changed due to our use of property maps to access

Operation	Effect	Cost
$++p$	returns an iterator to the successor	amortized $O(1)$
$--p$	returns an iterator to the predecessor	amortized $O(1)$
$D.begin()$	returns an iterator to the minimum element	$O(1)$
$D.end()$	returns an iterator to the one-past-the-end element; $--D.end()$ refers to the maximum element	$O(1)$
$D.insert(e)$	inserts an element	$O(\lg n)$
$D.insert(p,e)$	inserts an element with a hint	$O(\lg n)$, but amortized $O(1)$ if e is inserted just after p
$D.insert(S)$	inserts a collection of elements	$O(S \lg(n + S))$
$D.erase(e)$	deletes all elements equal to e ; let S denote the removed collection	$O(\lg n + S)$
$D.erase(p)$	deletes the element at the given position	amortized $O(1)$
$D.erase(p,q)$	deletes all elements between the two iterators; let S denote the removed collection	$O(\lg n + S)$
$D.find(e)$	returns an iterator to an element equal to e , if such exists	$O(\lg n)$
$D.lower_bound(e)$	returns an iterator to the smallest element not less than e , if such exists	$O(\lg n)$

Figure 1. Most important associative-container operations and their cost requirements. Here D is the container in question, p and q are iterators, e is an element, S is a collection of elements, $|S|$ denotes the cardinality of S , and n denotes the number of elements stored just prior to the operation in question.

pointers. We feel this is a useful technique in the implementation of linked data structures that deserves exposition.

We have the following requirements, which come from the C++ standard, for our data structure D :

- R_1 : D has to allow forward and backward iteration, starting at any position; a position is referred to by an *iterator*, which is a small object enough to be passed by value. Increasing or decreasing an iterator must take amortized constant time. Also, constant-time access must be provided for the first and last element stored in D .
- R_2 : D has to allow insertion, deletion of an existing element (passed by value or position), successful and unsuccessful searches (returning an iterator to the element or to the one-past-the-end element if unsuccessful). All these operations must have logarithmic cost, *cost* meaning the sum of element comparisons, element constructions, element destructions, and other operations on C++ built-in types, including memory allocation

and deallocation.

R₃: An iterator is *invalidated* if it points to a value which is either no longer in *D*, or differs from the one with which the iterator was created; insertion and deletion should not invalidate the existing iterators.

R₄: *D* can be constructed from a sequence of values (e.g. by repeated insertion) at logarithmic cost per element; however, if the sequence is already sorted, the construction must have linear cost.

The most important operations to be supported, together with their cost requirements, are summarized in Fig. 1. For a complete description of associative containers which we aim at implementing, we refer the reader to the C++ standard [2] or the SGI STL website [15].

The main purpose of this paper is to provide library users information about the utility of various optimization techniques described in computing literature. The optimization techniques considered are described in Section 2. One of the central themes of this paper is that smaller storage can lead to faster implementations, even if instruction count is higher due to the lack of direct access to pointers (because of compaction, encoding, or a combination of omission and reconstruction algorithms). Our implementation of different forms of red-black trees is high-lighted in Section 3. The experimental results are reported in Section 4. Finally, some conclusions are offered in Section 5.

The experiments discussed in this paper are carried out on a typical contemporary computer. The C++ programs and scripts used in our experiments are freely available (from <http://photon.poly.edu/~hbr/publi/rbtree.html>) so that, by downloading our code, users, who develop performance-critical software, can test the efficiency of the implementation alternatives developed in their own environment. Polished versions of the programs will also be made available via the CPH STL website [5].

2. Optimization techniques

In this section, we discuss a few optimization techniques that allow us to reduce either the amount of storage used by binary search trees (BSTs) or the number of instructions performed the BST operations. Some optimization techniques are appropriate for our purpose, and some are not, and we explain why.

Searching in a binary search tree. In a normal textbook description, the BST `find` algorithm relies on three-way element comparisons which have three outcomes less, equal, or greater. Hence, if only two-way element comparisons are available, as is the case in C++, `find` involves essentially two comparisons per node visited. There is a simple way to ensure that only one comparison per node is performed, even though more nodes may need to be visited. This search algorithm using only two-way comparisons was

discussed and experimentally evaluated by Andersson [1]. The same search algorithm was analysed by Vitter and Flajolet [18] a year earlier, so it seems that the technique has been part of computing folklore. In textbooks (see, for example, [10, 19]), this optimization is often only discussed in the context of binary search.

Essentially, the idea for searching for an element e in a BST is to simply test whether to go left at the current node x ($e < \text{element}[x]$) or not. If not, we may have equality but, if there is a right child, we go right anyway, and continue the search until we reach a null pointer. The only modification is that we remember the last node y on the search path for which the test failed and we visited the right child. Upon the end of the search, we do one more element comparison to see whether $\text{element}[y] < e$ (we already know that $\text{element}[y] \leq e$).

It is not hard to see that, if we followed a right pointer for a node y such that $e = \text{element}[y]$, the last node on the search path is either y itself (if $\text{right}[y]$ is null) or its successor (all the subsequent tests will fail and we basically follow the left-going chain from $\text{right}[y]$). Hence, the total increase in the search length over all successful searches, i.e. the number of pointers followed during the searches, is the total length of those successor paths, which is bounded by the total number of nodes in the tree, and there is no increase for unsuccessful searches. Thus, the average increase in the search length for a single search is at most one pointer.

Providing rank searches. We also have the option of providing rank searches by storing the subtree size in the node (see, for example, [8, §III.5.1]). Instead of providing a separate member function for rank searches, we simply upgrade the iterator class to provide random access, i.e. an iterator can advance (in logarithmic time) by an arbitrary number of positions. The fact that insertions may change the tree depending on whether the associative container is a `set/map` or `multiset/multimap` has implications. In `multiset/multimap` it is always safe to update the size fields on the way down the tree, and no traversal up is needed. In `set/map` a traversal back up to the root along the search path is necessary if the given element is not already present; if the element is present, we should not to update the size fields of the visited nodes. As an alternate design, this can be avoided by storing the size of one of the two subtrees along with a bit to remember which, and making sure on the way down that the subtree whose size is stored is the one which does *not* contain the key to be inserted (this is suggested for the randomized binary search trees by Martínez and Roura [7]).

Getting rid of parent pointers. Parent pointers are needed for traversals and various rebalancing algorithms that need to perform a second pass bottom-up after the first top-down scanning (e.g. insertion in red-black trees [4], or splaying a node after a find or insert operation). There are basically three techniques that help get rid of parent pointers in binary trees: recursion,

pointer inversion, and threading. In this section, we argue that only threading can be used to provide efficient traversal, insertion, and deletion.

Pointer inversion consists of modifying the fields in the tree to store the parent pointer during the traversal. The correct fields are restored on the way up. This technique is not appropriate when several traversals may be concurrently run, and does not work well with iterators.

Algorithms that need bottom-up access typically have a top-down pass first: they can be implemented using recursion (as in done, e.g. by Weiss [19]), at the cost of slower execution; indeed, an extra function call per level of the tree. The parent pointers are essentially stored in the recursion stack. Note that recursion is not always needed: for instance searching only uses left and right pointers, and therefore can remain non-recursive; also one can manage the recursion stack explicitly. Traversal can also be done using recursion, but an iterator needs to store the entire stack which makes it a heavy object. Moreover, should there be some structural change to the tree, even far away from the actual node pointed to by the iterator, the iterator would be invalidated.

Threading consists of encoding a pointer to the successor in leaves, by remarking that leaves have two pointers that are null. A special bit is needed to know if the right pointer is pointing to a right child or to the successor. Both pointer and bit can be maintained through insertions and deletions. Threading can be done symmetrically with the left pointer, to provide access to the predecessor. Even accessing the parent of a node x can be done in time proportional to the height of the tree by following all the right pointers from x to a node z , until the threading pointer of z necessarily leads to an ancestor y of x , and we know that both x and its parent are on the path from y to the predecessor z of y .

Threading does not suffer from the drawbacks of the previous two methods, it plays well with iterators (an iterator is essentially a node pointer). The only point where it fails us is when passing an iterator for deletion: since we must reattach the subtree of the node to be deleted back to the tree, we must somehow find an access to the parent of the node x to be deleted (by the technique above). Unfortunately, from then on, if rebalancing is needed after insertion, we do not have any other recourse than performing a search from the top, stacking all the nodes on the path for the bottom-up rebalancing, and even this is not possible if the tree implements a multiset (since in case of equality, we do not know on which side to continue the search). If only a set (no repetitions) is wanted, this strategy is possible but not effective: deletion is rendered inefficient by the rebalancing. If we do not want to perform any comparisons, we can access ancestors repeatedly in the fashion described above, but the complexity of a deletion would then become square-logarithmic. Another solution is to perform a search for the element to be deleted and store the ancestors in a stack. This is acceptable if

element comparisons are not too expensive. Note that with parent pointers, deletions do not have to perform element comparisons at all (but this is not a requirement of the C++ standard).

In conclusion, our only option for not storing the parent pointer is threading, which involves (slower) recursive functions, a considerably slower deletion method, and can only be made to work for sets and maps (without repetitions). Note that threading is still reasonably practical with treaps [12] and randomized binary search trees [7] when implemented using the join/split operations (which are top/down). With parent pointers, no recursive function is needed for any of the operations discussed in this paper and iterators are light (a single pointer).

Compacting color bits in pointers. Storing one or several bits in addition to the pointers leads to fatter nodes. On most modern machines, memory alignment considerations dictate that the extra bit(s) use actually a whole word of storage, making the structure extremely wasteful of memory. There are several tricks to avoid this waste: we discuss two.

One common trick used in implicit data structures is to encode a bit in the address of the children: the bit is zero if the address of the left child is less than the address of the right child, one otherwise. We can set the bit any way we want by permuting the two pointers (and exchanging the data they point to). This is transparent to the data structure, but may invalidate the iterators pointing to the two elements. Also, this trick encodes a bit only in internal nodes that have two children, but it cannot encode a bit in a leaf since both children pointers are null. For these reasons, this technique is not appropriate for this paper.

Another technique, which is appropriate for our purpose, is the use of the afore-mentioned memory alignment constraint. Since a node contains at least two pointers, we are guaranteed that its address is a multiple of four (actually, two would suffice for a single bit, and in practice many more bits are available per pointer). So we may store the bit in the least significant bits of the pointer, carefully cancelling them out when needing the pointer, and canceling all the high-order bits of the pointer when needing the bit encoded within.

3. Our implementations

As mentioned in the introduction, our starting point was the SGI STL implementation of a red-black-tree class template. In this implementation the insertion and deletion algorithms are based on those described in [4]. This red-black-tree class provides the actual functionality whereas appropriate wrapper classes provide the standard interface to associative containers `set`, `map`, `multiset`, and `multimap`. Our design goal was to let various versions

of red-black trees share most parts of the code. In this section, we describe our main programming techniques which made this recycling possible.

General approach. The basic data structure is implemented as a C++ class template

```
template <typename Key, typename Element, typename Extractor,
          typename Comparator, typename Allocator>
class rb_tree;
```

where `Key` is the type of keys and `Element` is the type of elements stored in the tree. The key and element types are the same for `set` and `multiset`, and the element type is a pair of `Key` and `Data` for `map` and `multimap`. A function object of type `Extractor` is stateless and its member function `operator()` is used to extract the key from the element (identity for `set` and `multiset`, and projection onto the first member of the pair for `map` and `multimap`; these are optimized away at compile time). A function object of type `Comparator` is used to compare two keys when ordering the keys inside the tree. We also allow the option to use a custom memory allocator, as required by the C++ standard. In order to accommodate `multiset` and `multimap`, the `rb_tree` class provides two insertion methods (`insert_unique` and `insert_equal`). Otherwise, it has the usual associative-container member functions. For testing purposes, we added a new member function `search_length` which performs a search and returns the number of element comparisons performed. For debugging purposes, we also added member functions to print the tree structurally and verify the invariants.

All in all, we have coded in this framework for several variants of red-black trees as well as for other forms of binary search trees. They are summarized, among with their features, in Table 1.

Property maps. Since we program in C++, we have the facility to use property maps for accessing the pointer. Instead of writing `x->right`, we can write `right[x]`, provided that `right` is a small object (typically with no data members) that has a member function `operator[]` taking a pointer (its *key type*) and returning another pointer (its *value type*). Property maps were introduced in the Boost Graph Library [14], and have been abstracted into a small Boost Property Map Library [13]. We borrowed the technique from there. This technique enables us to write much more legible code, in the manner of [4], as well as doing some type checking along the way. Property maps are used for accessing all the pointers of a node x (`parent[x]`, `left[x]`, `right[x]`) as well as its key (`key[x]`, whose value type is the key type), its color (`color[x]`, whose value type is boolean), etc.

Property maps are small objects which are typically optimized away at compile time. Hence, they have no disadvantage and offer extra type checking compared to direct pointer access. A similar effect was achieved in the SGI STL implementation [15] of red-black trees through static member functions.

Table 1. Data structures in our experiments. Node size assumes a word size of 32 bits.

Name	Description	Node size
<code>std::set</code>	C++ <code>set</code> , internally red-black tree.	16 bytes
<code>rb_tree</code>	Custom red-black trees, derived from the SGI STL <code>std::set</code> implementation, augmented to gather path length information.	16 bytes
<code>rb_cmp_tree</code>	Previous entry with color bit compacted in the parent pointer.	12 bytes
<code>rb_thr_tree</code>	Threaded red-black trees (* in the final version, these will have no parent pointers).	16 bytes
<code>rb_cmp_thr_tree</code>	Threaded red-black trees with the color bit compacted in the parent pointer.	12 bytes
<code>rb_aug_tree</code>	Same as <code>rb_tree</code> , augmented with subtree size information.	20 bytes
<code>rb_cmp_aug_tree</code>	Same as <code>rb_cp_tree</code> , augmented with size information.	16 bytes
<code>rb_cmp_aug_thr_tree</code>	Same as <code>rb_cmp_thr_tree</code> , augmented with size information.	16 bytes

Property maps appear to be more flexible, for instance they allow us to transparently compact the color bit into the parent pointer by redefining the color and parent property maps. No other changes are required.

We provide two general-purpose utilities, *compact_pointer_map* and *compact_bit_map*, which are standard lvalue property maps: they can be used on either side of an assignment. This is achieved by making the return type of the property map a *proxy object* instead of a value or reference. The proxy object is responsible for applying the correct mask and returning the proper type upon converting to an actual value (either pointer or bit), and for assigning only the relevant bits (leaving the bit or pointer portion unchanged) when making an assignment.

Conceivably, in an environment where no space is available to store the pointers within the key, and we want a set of keys and not of pointers to keys, the pointers may be stored not in the nodes themselves, but in some global tables. Using different property maps to access these tables again is transparent with respect to the BST code, and promotes code reuse.

The possibility of interchanging property maps to define the tree pointers leads us to formulate all the tree algorithms as function templates (with template parameters `Parent`, `Left`, `Right`, `Color`, etc. which are the types of the respective property maps). These algorithms are reused across the various implementations.

The header, a sentinel node. As is done in the SGI STL implementation, we use a special node, the *header*, which is the parent of the root; this makes

it the successor of the last iterator in the range (rightmost node) and the predecessor of the first iterator in the range (leftmost node). To have a uniform treatment of nodes in the predecessor and successor algorithms, the left child of the header points to the leftmost node, and the right child to the rightmost node. With parent pointers, we can store the root in the header's parent, and this can be taken advantage of in the algorithms. In this way, the iterators actually form a cyclical range with $n + 1$ positions, n being the number of elements stored, and the iterator to the one-past-the-end element is a pointer to the header node. When threading to avoid parent pointers, the root has to be maintained in a separate field, but it is even simpler to look for the successor (in the right node for leaves): we declare the header a leaf and keep the leftmost node the *right* child of the header, and the rightmost node the left child of the header.

4. Experimental results

In this section, we report preliminary experimental results for the trees of Table 1. In the final version of the paper, we will also include top-down threaded versions of our red-black trees, which will afford us a standard-compliant (in an expected sense) `set` implementation at a cost of only two pointers (8 bytes) per node! We will weigh those benefits against the impact on deletion discussed in Section 2. We also plan to include randomized binary search trees [7], splay trees [16], and treaps [12].

We offer the experimental results in the form of three tables detailing the running times for the following operations:

1. Insert a million elements in a certain order.
- 2.1. Traverse the data structure in order a hundred times.
- 2.2. Traverse the data structure in reverse order a hundred times.
3. Search for all the elements in the same order as inserted.
4. Gather the number of links followed by the search procedure of **3** divided by $n \ln n$ (this gives a multiplicative constant that should match the relevant asymptotic analysis of the internal path length), as well as the number of element comparisons performed (per node visited).
5. Perform a hundred thousand searches (first successful, or unsuccessful), picked at random without replacement from a pool of keys (first present, then not present).
6. Delete the hundred thousand elements of **5**.

We performed our experiments with a highly optimized pool memory allocator, which is taken from the Boost Pool Library [3]. The programs were compiled on a Mac OS X 10.4 platform with PowerPC processor, and a total of 1 GB of memory. The compiler was g++ 4.0 with all optimization options turned on.

As to the space occupancy of the data structures tested (see Table 1), we assume a setup in which a pointer has the same number of bytes as an integer representing the size (as indeed they should), and this number is 4 bytes. Because of alignment issues, a `bool` field with other pointer fields will require 4 bytes for storage as well.

The results for a random insertion order are presented in Table 2, for an increasing insertion order in Table 3, and for a decreasing insertion order in Table 4. As a rule, all the functionality gathering information about path length and number of element comparisons is given in separate functions in order to keep the search functions free of side effects. Thus the search times as quoted do not include the time taken to gather the statistics.

5. Discussion

We implemented several variants of binary search trees following best practice and using aggressive optimizations. Consequently, we verified that our implementations are more space-efficient and about as fast as—sometimes even faster than—the SGI STL implementation of red-black trees on which our implementations were based. Our code is freely available and will be released as part of the CPH STL [5].

The C++ standard [2, §23] requires that iterator operations `operator++` and `operator--` should take no more than amortized constant time. For our implementations of red-black trees this guarantee is met for a sequence of increments and a sequence of decrements, but not for a mixed sequence of increments and decrements. We are not aware of a standard library implementation of associative containers that would provide worst-case $O(1)$ -time iterator operations for a mixture of increments and decrements. Such a guarantee could be achieved by relying on leaf-oriented red-black trees or using threading. It would be interesting to know the performance implications of worst-case efficient iterator operations. Also, which approach provides the best performance in practice?

An economic way of implementing associative containers is to provide a sin-

Table 2. Random insertion order.

Data structure	1	2	3	4 (ptr.)	4 (comp.)	5 (succ.)	5 (unsucc.)	6
<code>std::set</code>	3.05s	0.3s	2.35s			0.25s	0.24s	0.18s
<code>rb_tree</code>	3.14s	0.28s	2.36s	1.47887	1.55125	0.25s	0.24s	0.17s
<code>rb_cmp_tree</code>	2.64s	0.26s	1.97s	1.47887	1.55125	0.22s	0.2s	0.16s
<code>rb_thr_tree</code>	3.16s	0.27s	2.36s	1.47887	1.55125	0.25s	0.25s	0.19s
<code>rb_aug_tree</code>	3.3s	0.29s	2.31s	1.47887	1.55125	0.26s	0.23s	0.38s
<code>rb_cmp_aug_tree</code>	3.38s	0.29s	2.34s	1.47887	1.55125	0.25s	0.24s	0.38s
<code>rb_cmp_aug_thr_tree</code>	3.43s	0.27s	2.41s	1.47887	1.55125	0.25s	0.25s	0.39s

Table 3. Increasing insertion order.

Data structure	1	2	3	4 (ptr.)	4 (comp.)	5 (succ.)	5 (unsucc.)	6
<code>std::set</code>	1.72s	0.11s	0.58s			0.1s	0.08s	0.13s
<code>rb_tree</code>	1.82s	0.09s	0.6s	1.47176	1.54414	0.1s	0.08s	0.14s
<code>rb_cmp_tree</code>	1.53s	0.06s	0.55s	1.47176	1.54414	0.07s	0.08s	0.1s
<code>rb_thr_tree</code>	1.87s	0.08s	0.62s	1.47176	1.54414	0.08s	0.09s	0.15s
<code>rb_aug_tree</code>	3.37s	0.28s	2.36s	1.47887	1.55125	0.24s	0.24s	0.38s
<code>rb_cmp_aug_tree</code>	2.25s	0.09s	0.58s	1.47176	1.54414	0.09s	0.08s	0.19s
<code>rb_cmp_aug_thr_tree</code>	2.33s	0.09s	0.62s	1.47176	1.54414	0.09s	0.09s	0.2s

Table 4. Decreasing insertion order.

Data structure	1	2	3	4 (ptr.)	4 (comp.)	5 (succ.)	5 (unsucc.)	6
<code>std::set</code>	1.62s	0.11s	0.57s			0.09s	0.09s	0.13s
<code>rb_tree</code>	1.66s	0.09s	0.58s	1.47176	1.54414	0.08s	0.1s	0.14s
<code>rb_cmp_tree</code>	1.48s	0.06s	0.54s	1.47176	1.54414	0.07s	0.07s	0.11s
<code>rb_thr_tree</code>	1.73s	0.08s	0.6s	1.47176	1.54414	0.09s	0.09s	0.15s
<code>rb_aug_tree</code>	2.11s	0.09s	0.58s	1.47176	1.54414	0.08s	0.09s	0.18s
<code>rb_cmp_aug_tree</code>	2.16s	0.09s	0.57s	1.47176	1.54414	0.08s	0.08s	0.19s
<code>rb_cmp_aug_thr_tree</code>	2.25s	0.09s	0.6s	1.47176	1.54414	0.09s	0.08s	0.2s

gle search-tree core and then implement `map`, `set`, `multiset`, and `multimap` as wrapper classes that all use the core—as done by the SGI STL implementation. However, asymptotically it would be better to implement `multiset/multimap` separately by collapsing duplicates and keeping all equal elements together. If all elements are equal, this implementation can provide `lower_bound` at a cost of $O(1)$, whereas the wrapper approach still requires logarithmic cost. Furthermore, the single-comparison-per-level optimization still works for the new implementation. It would be interesting to know how many duplicates there should be before the new approach wins the traditional approach.

References

- [1] A. Andersson, A note on searching in a binary search tree, *Software—Practice and Experience* **21** (1991), 1125–1128.
- [2] British Standards Institute, *The C++ Standard: Incorporating Technical Corrigendum 1*, 2nd Edition, John Wiley and Sons, Ltd. (2003).
- [3] S. Cleary, Boost pool library, Worldwide Web Document (2000–2001). Available at <http://www.boost.org/libs/pool/doc/>.
- [4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd Edition, The MIT Press (2001).
- [5] Department of Computing, University of Copenhagen, The CPH STL, Website accessible at <http://www.cphstl.dk/> (2000–2006).
- [6] D. E. Knuth, *Sorting and Searching, The Art of Computer Programming* **3**, 2nd Edition, Addison Wesley Longman (1998).
- [7] C. Martínez and S. Roura, Randomized binary search trees, *Journal of the ACM* **45**

- (1998), 288–323.
- [8] K. Mehlhorn, *Sorting and Searching, Data Structures and Algorithms 1*, Springer-Verlag (1984).
 - [9] P. Plauger, A. A. Stepanov, M. Lee, , and D. R. Musser, *The C++ Standard Template Library*, Prentice Hall PTR (2001).
 - [10] G. J. Rawlins, *Compared to What?: An Introduction to the Anaylsis of Algorithms*, W. H. Freeman (1991).
 - [11] R. Sedgewick, *Algorithms in C*, 3rd Edition, Addison-Wesley Publishing Company, Inc. (1998).
 - [12] R. Seidel and C. Aragon, Randomized search trees, *Algorithmica* **16** (1996), 464–497.
 - [13] J. Siek, Boost property map library, Worldwide Web Document (2000–2002). Available at http://www.boost.org/libs/property_map/property_map.html.
 - [14] J. G. Siek, L. Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*, Addison Wesley Professional (2002).
 - [15] Silicon Graphics, Inc., Standard template library programmer's guide, Website accessible at <http://www.sgi.com/tech/stl/> (1993–2004).
 - [16] D. D. Sleator and R. E. Tarjan, Self-adjusting binary search trees, *Journal of the ACM* **32** (1985), 652–686.
 - [17] B. Stroustrup, *The C++ Programming Language*, Special Edition, Addison Wesley Longman, Inc. (2000).
 - [18] J. S. Vitter and P. Flajolet, Average-case analysis of algorithms and data structures, *Algorithms and Complexity, Handbook of Theoretical Computer Science A* (1990), 431–524.
 - [19] M. L. Weiss, *Data Structures and Problem Solving using C++*, 2nd Edition, Addison Wesley Longman, Inc. (2000).