# Extending STL `maps` using LBSTs *

Leonor Frias §

**Abstract**

Associative containers are basic generic classes of the C++ standard library. Access to the elements can be done by key or iterator, but not by rank. This paper presents a new implementation of the `map` class, which extends the Standard with the ability to support efficient direct access by rank without using extra space. This is achieved using LBST trees. This document reports on the algorithmic engineering of this implementation as well as, experimental results that show its competitive performance compared to the widespread GCC library map implementation.

## 1 Introduction.

The use of standard libraries has become a practice of increasing importance in order to ease and fasten software development. The C++ programming language defines a standard library [5] whose algorithmic core constitutes the Standard Template Library (STL) [6].

The STL defines three different types of objects that cooperate between them. These are containers (to store collections of elements), iterators (to access and traverse elements in a container) and algorithms (to process the elements in a container). The C++ standard library defines both the functionality and the performance (using Big-O notation) of these objects.

Associative data structures (including `set`s, `map`s, `multiset`s and `multimap`s) are a particularly important kind of containers in the STL. The `map` class is a generic container that corresponds to the idea of a dictionary. Specifically, it is defined as a unique ordered associative container, that is, a container with efficient search, insertion and deletion of elements (pairs of key and value) based on unique keys that are also used to define an order among container elements. This order is internally kept to allow the user to efficiently navigate the structure through iterators. On a `map` with $n$ keys, search, insertion and deletion operations must be performed in $O(\log n)$ time; iterator traversal (either in increasing or decreasing order) must take amortized $O(1)$ time. For more details see [6, section 6.6].

The use of `map`s is illustrated in the following example that prints, in lexicographic order, how many times each word appears in the input:

```
string word;
map<string,int> M;
while (cin >> word) ++M[word];
for (map<string,int>::iterator it=M.begin();
     it!=M.end(); ++it)
    cout << it->first << " " << it->second << endl;
```

According to the Standard, access to the `map` elements can be done using direct access by key (as the instruction `++M[word]` in the example above) or using sequential bidirectional iterators that traverse the `map` in order (as in the `for` loop above). However, direct access by rank in the ordered sequence is not considered. This means that, for instance, getting the median element in a `map` of size $n$ takes $\Theta(n)$ time.

This paper presents a new implementation of the `map` class. While retaining compatibility with the C++ standard library, this new implementation offers efficient random access to the map elements. This functionality is achieved through the inclusion of random access iterators for `map`s, which allow the user to access the $i$-th element basically in the same way as he/she would access the $i$-th element of an array or a vector.

For instance, using random access iterators one can get the median element in `M` using the standard brackets operator, in the following way:

```
string median = M.begin()[M.size()/2].first;
```

Besides, it is possible to compute the difference between two random access iterators `p` and `q` with the usual subtraction: `p - q`. This could be useful, for instance, to calculate how many words are between two specific ones in the map `M` above. Additionally, an integer `k` can be added to an iterator (`p+=k`) to get the same result as repeating `k` times `++p`. Finally, it is possible to compare two iterators (`p<q`, `p<=q`, ...). Note that in all these operations the elements cardinal position in the sequential structure is used to get the result (instead of their keys).

In order to offer all these possibilities, we have implemented the `map` class using Logarithmic Binary

*This work has been supported by GRAMMARS project under grant CYCIT TIN2004-07925 and partially by AEDRI-II project under grant MCYT TIC2002-00190.

§Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya.

Search Trees (LBSTs) [10]. This variant of balanced binary search trees adds to existing implementations the ability to support direct access by rank in logarithmic time without using extra space. Moreover, its balancing criterion is fast to evaluate, and hence it is expected to be efficient in practice.

In this paper we report on the algorithmic and implementation issues of this new variant of `map`s, and experimentally show its competitive performance compared to the GCC implementation, which uses red-black trees as internal data structure. Besides, by contrast to other implementations, care has been taken to use the most appropriate algorithm upon input characteristics as required by the Standard. Altogether, equivalent or better performance has been obtained in main operations such as search, insertion or deletion by key, thus proving that more functionality can be offered without relevant drawbacks.

The remainder of this paper is organized as follows. First, in Section 2, related work is outlined. Then, in Section 3, the characteristics of the LBST implementation are presented. In Section 4, the design and results of the experiments are shown. Finally, Section 5 sets out the conclusions of this work. The appendix presents the details of the feasibility of the new method presented in Section 3 for one-way insertion and deletion operations.

## 2 Related work.

Due to the `map` cost requirements fixed by the C++ Standard, balanced binary search trees have been typically used for `map` implementation. Note that because elements must be kept in sorted order, hashing is not an option. Also, because the Standard fixes the costs requirements in the worst case scenario, randomized data structures like skip-lists can neither be used.

There are many different STL `map` implementations. However, none offers direct access by rank. Furthermore, a lot of them originally derive from the Hewlett-Packard implementation [11] that uses red-black trees: GCC [1], STL Port [2], SGI [3] and Microsoft [4].

This paper compares the performance of the LBST implementation against that of the GCC STL v.3 implementation [3]. The main reason for this selection is that this is a widespread, accessible and free implementation. An important characteristic of the GCC implementation is its use of generic operations for the four ordered associative containers included in the Standard (`map`, `set`, `multimap` and `multiset`), i.e., the same algorithm is used for each container when possible, even if this could cause unnecessary overhead. Another characteristic of the GCC implementation is that almost all algorithms are coded iteratively. Finally, it is not specialized for operations with sorted ranges as input and so, for these operations, violates the standard cost requirements.

Other approaches not using red-black trees have been proposed. For example, CPH STL [2] offers an implementation using B+ trees that is reported to be faster than SGI for search and traversal but equal or worse for insertion and deletion [4]. Another implementation uses AVL trees; in this case, the obtained results are equal or slightly worse compared to SGI's [7].

In any case, if these implementations were extended to support logarithmic access by rank, both extra space (to store subtrees sizes) and time in modifying operations (to keep size data updated) would be required. In [1, section 14.1] an extended red-black tree is analysed.

## 3 Implementation characteristics.

The final version of the `map` class is the result of successive refinements guided by experimental results. Specifically, three complete versions were developed. Although all of them are based on LBSTs, some changes in the structure were introduced successively. This section defines LBSTs and presents the evolution and characteristics of the three implementations.

**Logarithmic binary search trees.** LBSTs [10] are binary search trees that are balanced upon subtree sizes. Implementing the `map` class with LBSTs adds to existing implementations the ability to support direct access by rank (by means of random access iterators) in logarithmic time without using extra space. Furthermore, its balancing criterion is easier and faster to evaluate than that of other variants of BSTs that are also balanced upon subtree sizes, such as Weighted BSTs [9].

Before we go on, let us recall the definition of LBSTs. Let $\ell(n)$ be the number of bits required to encode $n$, that is, $\ell(0) = 0$, and $\ell(n) = 1 + \lfloor \log_2 n \rfloor$ for any $n \geq 1$. Given a BST $T$, denote by $|T|$ its number of nodes, and let $\ell(T)$ denote $\ell(|T|)$. Then, $T$ is an LBST if and only if (1) $T$ is an empty tree, or (2) $T$ is a non-empty tree with subtrees $L$ and $R$, such that $L$ and $R$ are LBSTs and $|(\ell(L) - \ell(R))| \leq 1$.

**First version.** The first version was a straightforward recursive implementation of the LBST algorithms. A basic LBST node contains (apart from a key and its related value) its subtree size and two pointers to its children. As in the GCC implementation, each node was enlarged to store its parent pointer, so as to support navigating the structure starting at any point. Empty trees were represented by a null pointer. Finally, a spe-

[1] http://gcc.gnu.org

[2] http://www.stlport.org

[3] http://www.sgi.com/tech/stl/

[4] http://www.microsoft.com/downloads/details.aspx?FamilyId=272BE09D-40BB-49FD-9CB0-4BFA122FA91B&displaylang=en

cial header node was introduced to mark the end of the sorted sequence (according to keys) of the map elements.

Unlike the GCC implementation, this first version was already compliant with the costs required by the Standard for some specialized operations with sorted ranges as input (this mainly includes constructors and multiple element insertions). For instance, given $n$ elements in sorted order, the GCC implementation builds a `map` in $\Theta(n \log n)$ time, while our implementation does so in $\Theta(n)$ time, as the C++ Standard requires. This was achieved implementing specific algorithms for these cases.

Besides, our algorithms may be more convenient than the ones in GCC for the "insertion with hint" operation. This standard operation is intended to reduce insertion time by starting the search at the hint, instead of at the tree root. Our implementation tries to minimize key comparisons assuming that the hint is close to the target key, whereas the GCC implementation ignores it unless the new element should go just before it. The experimental results showed that this approach reduces the cost of insertions by hint when the hint is good and comparisons are expensive. Anyway, the cost of our algorithm is never worse than $O(\log n)$.

Unfortunately, LBSTs exhibited a limitation: deleting a node from an iterator requires logarithmic cost, because every node from the deleted one to the root must be checked for balance, and its size field must be updated. Note that the Standard requires amortized constant cost for this operation. However, this requirement of cost is not critical, and was probably established arbitrarily assuming a red-black tree implementation. Observe that the cost of deleting by iterator (which is perhaps not a very common operation) is usually offset by the cost of previous insertions. Moreover, our implementation does offer an efficient range erase.

**Second version.** Experimental comparison of our first version against the GCC implementation proved that, on single element operations, GCC was faster. The analysis of the GCC code suggested that recursivity could be the reason of the weakness: it is well known that recursive implementations usually have worse performance on hardware than iterative ones.

Therefore, a second, iterative version was implemented. Besides, a unique dummy node for all empty trees was introduced. This dummy node reduced the number of possible cases and simplified the resulting code, thus making it both faster and easier to debug.

**Third version.** The experimental analysis of the second version showed that all operations achieved similar or better performance than GCC's, except for single element insertion, which was still around 30% slower. In order to understand the reason for this difference, observe that single element insertion and deletion have three main phases: (1) top-down search, (2) local insertion or deletion of the input key, and (3) down-top update of subtree sizes and rebalance through single or double rotations. The main difference between red-black trees and LBSTs is in the last phase, which can be inferred from the code and confirmed by profiling and experimental data. Specifically, while the red-black tree final update takes an amortized constant number of steps, this in LBSTs requires logarithmic time, because all levels, from the update point until the root, must be visited.

As the extra visited nodes are in fact the same that in the top-down search phase but in opposite order, the feasibility of suppressing the down-top traversal was considered. This involved new insertion and deletion algorithms to update the tree in just one (top-down) phase. These algorithms must update the tree and preserve the balancing criterion without knowing if the insertion or deletion will finally take place. The details of these new, sophisticated algorithms are in the Appendix.

The implementation of these algorithms, jointly with other minor changes, made up the third and final version. First, the LBST structure was required to be more flexible. The new LBST structure was based on an idea of [8] that consists on changing the meaning of the node's size field: rather than corresponding to its subtree size, it corresponds to the size of only one of the two children (which one is determined by the sign). Additionally, the total tree size is stored in the header node.

As a consequence of this change, the size of each subtree must be calculated at every step down the tree. However, this can be done in constant time. When starting at the root (the common case), we know the total size of the tree and the size of one of its children. Therefore, the child size is known directly, or otherwise trivially calculated. This process can be repeated at each step down the tree. On the other hand, for iterator operations that start anywhere in the tree, the size of the current subtree can be obtained visiting a constant number of tree levels on the average.

Finally, changing the meaning of the size field produced another benefit: a substantial reduction, in search schemes, of the number of visited nodes. In the third version, the only visited nodes are those whose key must be compared by the operator `<`. By contrast, in the first and second versions, some nodes were visited just to look up its size field. Altogether, these modifications decreased memory accesses, improved their locality and reduced the total time of operations.

## 4 Performance evaluation.

This section presents the experiments that were conducted to compare the three versions of the LBST implementation and the GCC implementation.

**Experimental setting.** An experiment is configured by means of input parameters such as: `map` implementation, key and value type (integer or string), `map` operation, input data characteristics and operation size. An experiment output is the time (in seconds) that takes the operation, not including experiment set up. In the case of single element operations, output is the amount of time of the complete sequence of operations.

All the experiments were performed on different widely available hardware architectures (x86, UltraSparc and Alpha) in order to obtain general results. Given that the results for every machine are similar in terms of relative performance (as compared to GCC), this paper only presents results for an x86 based machine: an Intel Pentium-4 with a 3.06 GHz hyperthreading CPU with 512 Kb cache and 900 Mb of main memory.

By default, the results assume integer keys. Results for other key types will only be mentioned when they present a different qualitative behavior.

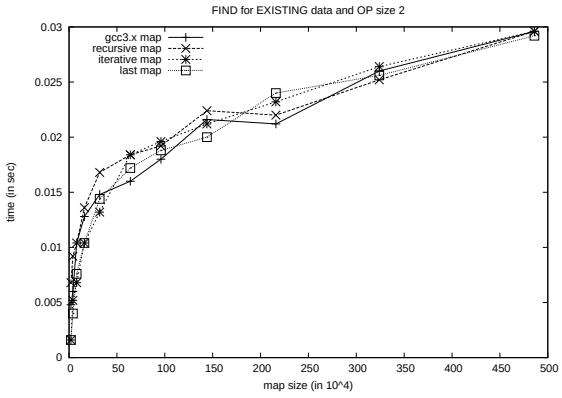
**Search.** The `map` class offers four kinds of search. However, they are very similar in terms of number of visited nodes and comparisons, so their experimental behavior is also similar. Therefore, results are only shown for `find` (classical search) and `equal_range` (range containing all elements whose key is the given one), in Figures 1(a) and 1(b), respectively.

Search operations neither use nor change balancing information, and so, their cost depends mainly on tree height. Given that the experimental results are almost identical for all implementations (see Figure 1(a)), the height of the trees can be considered equal from a practical point of view.

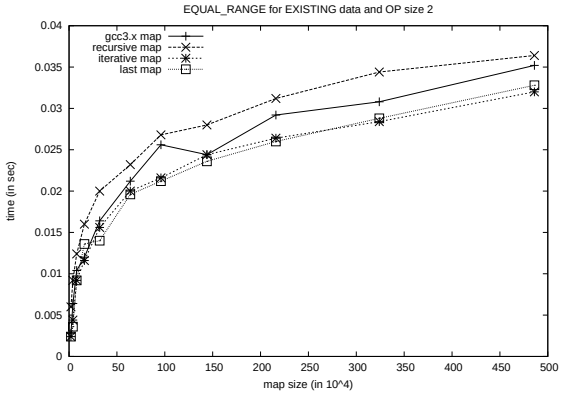
Nevertheless, for `equal_range`, our implementation is faster, specially for string keys, because it avoids the unnecessary overhead of the generic implementation in GCC. Specifically, `equal_range` is implemented in GCC calling both `lower_bound` and `upper_bound`, so our implementation performs almost half as key comparisons as GCC's.

**Simple insertion.** Results for random input data are shown in Figure 1(c). They show that our final implementation is competitive in performance and how our implementation has improved gradually.

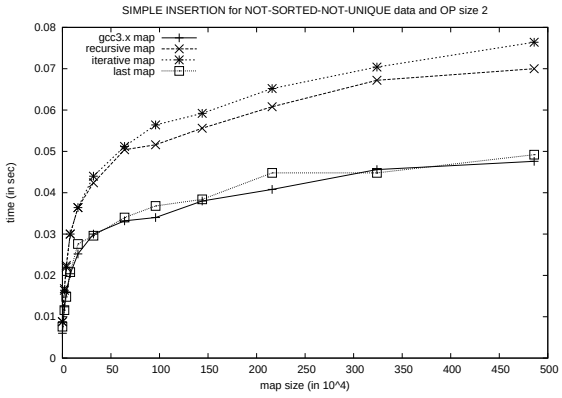
**Deletion by key.** Results for existing integer keys, non existing integer keys and existing string keys are shown in Figures 2(a), 2(b) and 2(c), respectively. They show that our third implementation is faster in general than GCC's; it is only slightly slower than GCC's for




(a) Classical search (`find`) of existing integer keys, fixed operation size of $2 * 10^4$

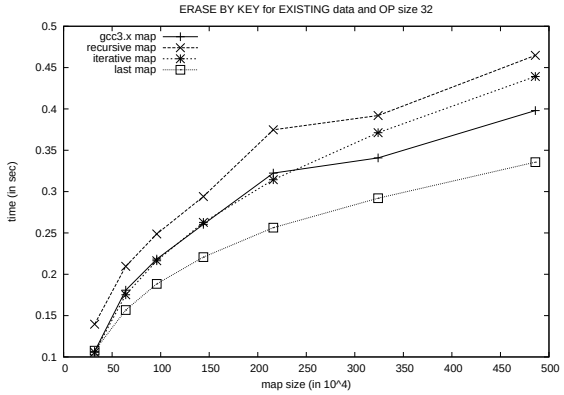



(b) `equal_range` of existing integer keys, fixed operation size of $2 * 10^4$

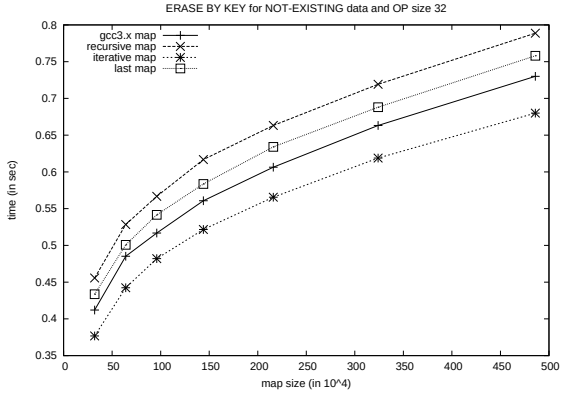



(c) Simple insertion for random data and fixed operation size of $2 * 10^4$

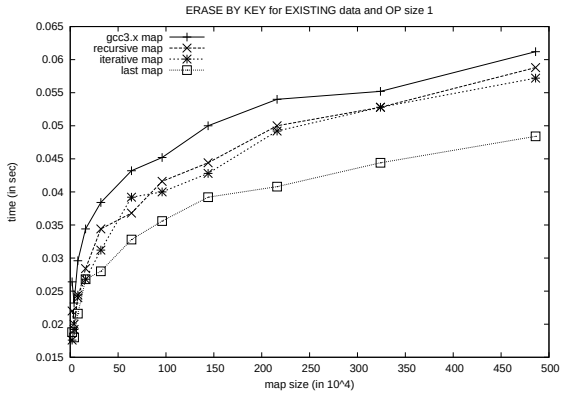
Figure 1: Experimental results for insertion and search by key ($x$ axis=operation size/$10^4$, $y$ axis= time in seconds).




(a) Erase of existing integer keys, fixed operation size of $32 * 10^4$

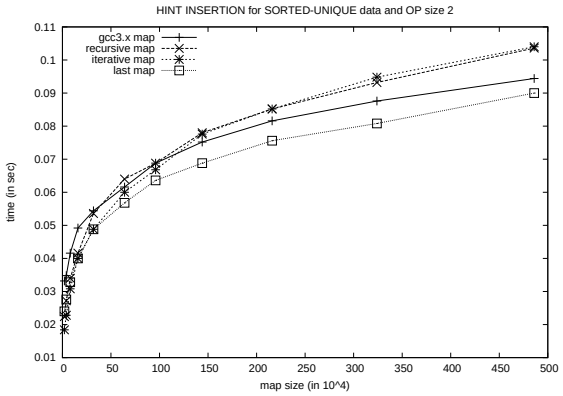



(b) Erase of not existing integer keys, fixed operation size of $32 * 10^4$

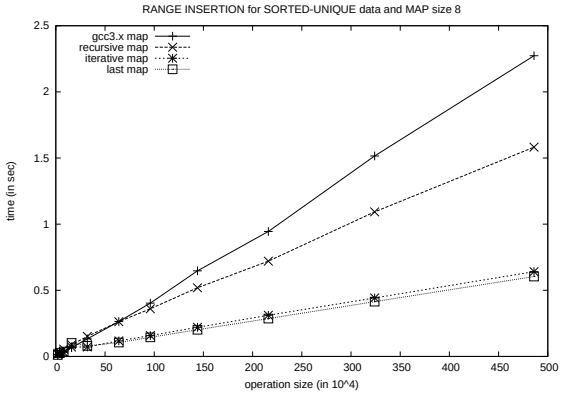



(c) Erase of existing string keys, fixed operation size of $1 * 10^4$

Figure 2: Experimental results for erase by key ($x$ axis=operation size/$10^4$, $y$ axis= time in seconds).




(a) Hint insertion for (string) sorted data and fixed operation size of $2 * 10^4$

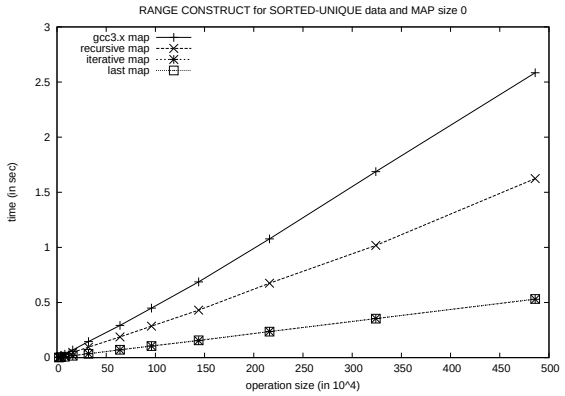



(b) Insertion from a sorted range and fixed `map` size of $8 * 10^4$

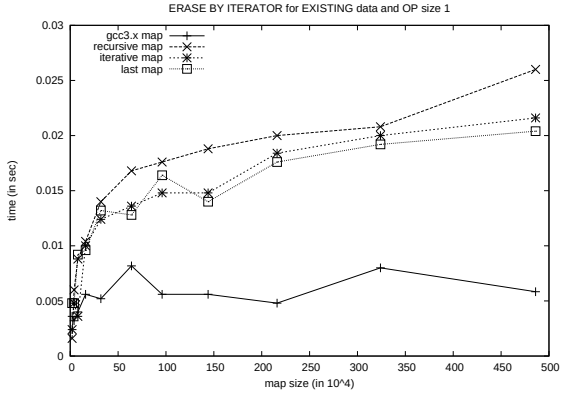
Figure 3: Experimental results for some non-basic `map` operations ($x$ axis=operation size/$10^4$, $y$ axis= time in seconds).

non existing integer keys due to rotations. In fact, it is substantially faster (more than 25%) than GCC's for successful erases of string keys because GCC erase by key is implemented through an `equal_range` operation, for which our implementation performs half as key comparisons as the GCC implementation (as commented previously, because `equal_range` GCC implementation is common to all associative containers).

**Insertion with hint.** In this experiment, each insert execution uses the previous inserted element as a hint. Results for sorted string input data are shown in figure 3(a). They show that our approach is slightly better than GCC's when the hint is reasonable. As expected, when the hint is not accurate, LBST results are slightly worse than GCC's.

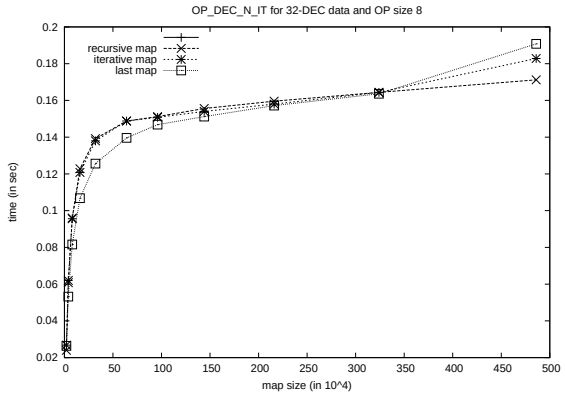



(a) Constructor from a sorted range

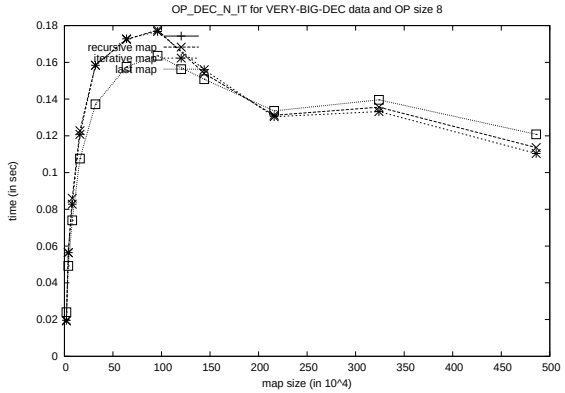



(b) Erase by iterator, fixed operation size of $1 * 10^4$

Figure 4: Experimental results for some non-basic `map` operations ($x$ axis=operation size/$10^4$, $y$ axis= time in seconds).




(a) 32-position decrement with fixed operation size of $8 * 10^4$




(b) Very big decrement relative to `map` size with fixed operation size of $8 * 10^4$

Figure 5: Experimental results for some random access iterator operations ($x$ axis=operation size/$10^4$, $y$ axis= time in seconds).

**Operations with sorted input.** Results for construction and insertion with sorted input are shown in Figures 4(a) and 3(b) respectively. Although the results for range insertions in LBSTs are only significantly better than GCC's when the input size is bigger than the current `map` size, the LBST constructor remarkably outperforms GCC's.

Besides, range insertion results suggest a possible change in the implementation that would take into account the range size to check whether it is sorted or not.

**Deletion operations through iterators.** Results for single deletion through an iterator are shown in Figure 4(b). They show modest results for the LBST implementation because, as explained in Section 3, the LBST data structure is worse fit than red-black trees for this operation. However, absolute times are not that high and are similar to those of deletions by key. On the other hand, results for deletion operation from a range defined by iterators are very similar for both implementations and show small absolute times.

**Random access iterators operations (rank operations).** Arbitrary iterator increment and decrement operations have been analyzed. Results for 32-position decrement and a bigger decrement are shown in Figures 5(a) and 5(b), respectively. As the GCC implementation does not support these operations, results are only shown for LBSTs.

First, when tuning our approach, we found that our general increment/decrement algorithm was already faster than the incremental approach from just 3 or

4 positions movement. Besides, observed times are similar for different increments/decrements, and not great differences in performance are observed between the versions of our implementation. This last result highlights that in the final version, the overhead cost of calculating subtree size from an arbitrary node is offset by the advantages of the new approach. Specifically, less nodes are accessed for looking up the size fields when traversing the tree downwards.

## 5 Conclusions.

In this paper, a new implementation of the `map` class of the standard C++ library using LBSTs has been presented. This implementation extends the Standard with the ability to support direct access by rank taking logarithmic time, and without using extra space. This was achieved by using random access iterators, which are an standard extension of bidirectional iterators, already required for the `map` class.

Our implementation meets the standard `map` cost requirements for almost all operations. The only relevant exception is the deletion through an iterator, which requires logarithmic cost due to the underlying LBST structure. On the other hand, some operations like construction from a sorted range do meet the Standard requirements. By contrast, the current GCC implementation (a widespread, accessible, free implementation based on red-black trees) does not.

The final implementation of the `map` class is the result of successive algorithmic refinements guided by experimental results. Three versions have been presented. The final version includes changes in the LBST structure to make it more suitable for implementing its operations. Specifically, a new algorithm for single element insertion and deletion was designed and its correctness exhaustively proved. The key point was to perform a unique top-down traversal, joining search and update phases. These changes decrease memory accesses and improve the use of cache memory.

Overall, the final implementation achieves a competitive performance compared to GCC's. In particular, main operations such as search, insertion or deletion by key show equivalent or better performance. This shows that the Standard could offer more functionality without relevant drawbacks.

Finally, this work could be extended in several ways. First, it could be widen for the rest of associative containers. Also, some operations could be specialized with respect to the key type. Finally, the efficiency could be further improved applying low-level techniques such as loop unrolling, or improving cache usage.

## 6 Acknowledgments.

I would like to thank Jordi Petit and Salvador Roura for their ideas and remarks, in short, for making possible this work.

## References

[1] T. H. Cormen, C. Leiserson, and R. Rivest. *Introduction to algorithms*. The MIT Press, Cambridge, 2 edition, 2001.

[2] DIKU (Performance Engineering Laboratory). CPH STL (Copenhaguen STL). http://www.cphstl.dk.

[3] GNU GCC. GCC C++ Standard Library. http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-3.3/index.html.

[4] J. Hansen and A. Henriksen. The (multi)?(map|set) of the Copenhagen STL. Technical report, DIKU (Performance Engineering Laboratory), 2001. http://www.cphstl.dk/Report/Map+Set/cphstl-report-2001-6.ps.

[5] International Standard ISO/IEC 14882. *Programming languages — C++*. American National Standard Institute, 1st edition, 1998.

[6] N. M. Josuttis. *The C++ Standard Library : A Tutorial and Reference*. Addison-Wesley, 1999.

[7] S. Lynge. Implementing the AVL-trees for the CPH STL. Technical report, DIKU (Performance Engineering Laboratory), 2004. http://www.cphstl.dk/Report/AVL-tree/cphstl-report-2004-1.ps.

[8] C. Martínez and S. Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998.

[9] J. Nievergelt and E. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):652–686, 1985.

[10] S. Roura. A new method for balancing binary search trees. In F. Orejas, P.G. Spirakis, and J. van Leeuwen, editors, *28th International Colloquium on Automata, Languages and Programming*, volume 2076 of *Lectures Notes in Computer Science*, pages 469–480. Springer–Verlag, Berlin, 2001.

[11] A. Stepanov and M. Lee. The standard template library. Technical Report HPL-95-11(R.1), HP, 1995. http://www.hpl.hp.com/techreports/95/HPL-95-11.html.

## Appendix A: Case analysis for single element insertion and erase

Changes introduced in the basic LBST structure aim that single element modifying operations require a unique tree top-down traversal merging search and update phases. However, we must proof that for each possible input LBST and modifying operation a correct algorithm exists, whether the modification takes eventually place or not.

On the one hand, for unconditional modifications (that is, always take place) such as erase from an iterator or erase the minimum or maximum from a tree, we can always get a correct algorithm without making any assumptions.

On the other hand, for conditional modifications such as, insertion or erase by key, which are the most frequent operations, tree coherence must be proved for every possible case with a case analysis.

The case analysis is presented for insertion/erase in left child (because right is analogous) and in case of being successful, the tree should be rebalanced (otherwise, the tree coherence is directly preserved in both cases).

Additionally, we will assume that all the nodes depicted exist and subtrees may be empty. Finally, size fields of visited nodes will be changed properly to indicate the opposite subtree size of the one being modified.

Lastly, notation:

- $\ell(T)$ *sat*: if and only if increasing $T$ size in one unit, provokes $\ell(T)$ also to be incremented.
- $\lambda- > \beta$: shows change in $\ell(T)$ value in case the operation is successful.

### A.1 Insertion by key.

There are two main cases:

#### Insertion in left branch of left child

There are three possible situations, showed in figure 6. They are the following:

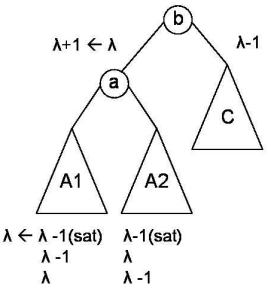



Figure 6: (Hypothetical) unbalancing insertion in left branch of left child ($A1$). Possible configurations.

1. $\ell(A1) = \lambda - 1(sat),\ \ell(A2) = \lambda - 1(sat)$
2. $\ell(A1) = \lambda - 1,\ \ell(A2) = \lambda$
3. $\ell(A1) = \lambda,\ \ell(A2) = \lambda - 1$

For the first and third cases a single right rotation suffices (see figure 7(a)). On the other hand, for the second case (see figure 7(b)) a double rotation is needed given that $\ell(A1) < \ell(A2)$ whether the insertion is successful or not (if $\ell(A1)$ reached $\lambda$, there would be a contradiction because then $\ell(a)$ before the insertion should be $\lambda + 1$).

#### Insertion in right branch of left child

There are three possible situations, showed in figure 8. They are the following:

1. $\ell(A1) = \lambda,\ \ell(A2) = \lambda - 1$
2. $\ell(A1) = \lambda - 1(sat),\ \ell(A2) = \lambda - 1(sat)$
3. $\ell(A1) = \lambda - 1,\ \ell(A2) = \lambda$

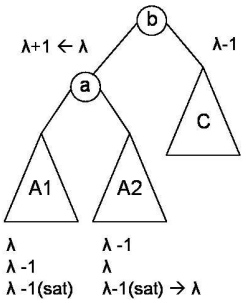


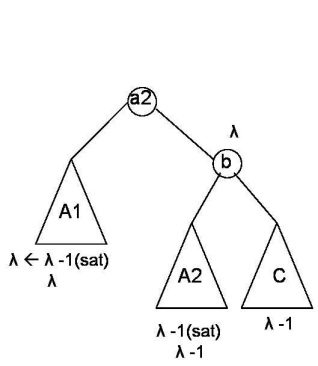

Figure 8: (Hypothetical) unbalancing insertion in right branch of left child ($A2$). Possible configurations.

For the first case, a single right rotation suffices (see figure 9(a)). For the second case on the other hand, a double rotation is needed, whether the insertion takes place in $A2L$ subtree or in $A2R$ (see figure 9(b)).

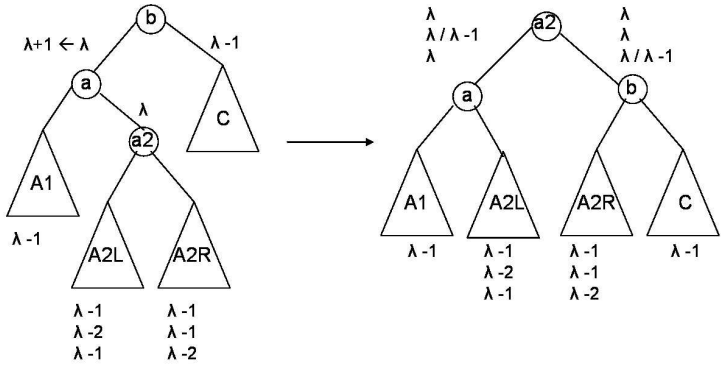
However, the third case is a bit more complicated: in figures 10 and 11 is shown the result of applying a double rotation to the initial tree, supposing that insertion takes place respectively in $A2L$ and $A2R$ subtrees.

For the first subcase we obtain a satisfactory result: as $A1$ is not saturated (otherwise $\ell(A)$ should be $\lambda + 1$ before the insertion), $\ell(a)$ cannot be greater than $\lambda$.

On the other hand, for the second subcase an incoherence may arise(marked with *) when $\ell(C) = \lambda - 1(sat)$. To solve it, $A2$ subtree must be firstly rotated to the left, as shown in figure 12, where insertion may take place either in left or right child of $A2R$.

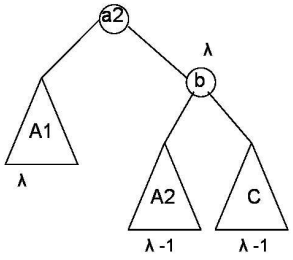



(a) Case 1 and 3: tree after one right rotation

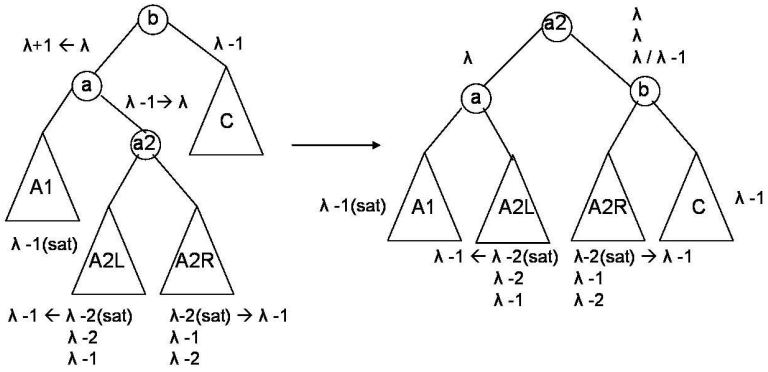



(b) Case 2: tree before and after the double rotation

Figure 7: (Hypothetical) unbalancing insertion in left branch of left child ($A1$).




(a) Case 1: tree after a single right rotation




(b) Case 2: tree before and after the double rotation, supposing insertion in $A2L$ or in $A2R$.

Figure 9: (Hypothetical) unbalancing insertion in right branch of left child ($A2$).

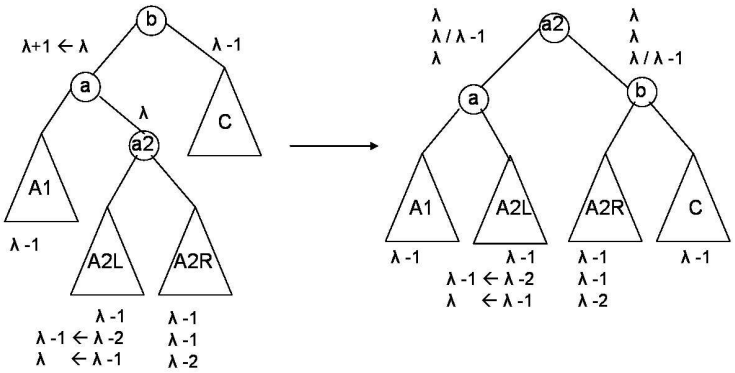



Figure 10: (Hypothetical) unbalancing insertion in right branch of left child ($A2$). Case 3: tree before and after the double rotation. Supposing insertion in $A2L$

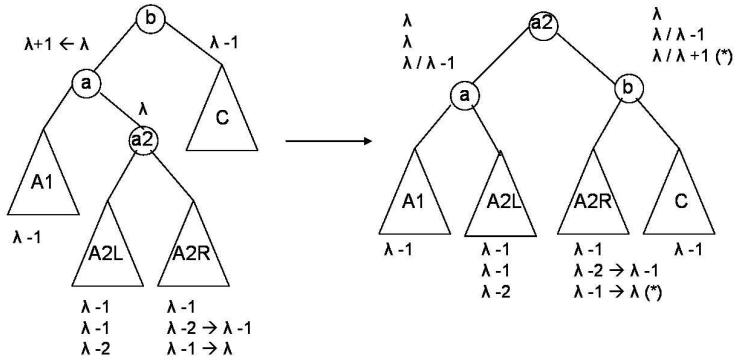



Figure 11: (Hypothetical) unbalancing insertion in right branch of left child ($A2$). Case 3: tree before and after the double rotation. Supposing insertion in $A2R$.

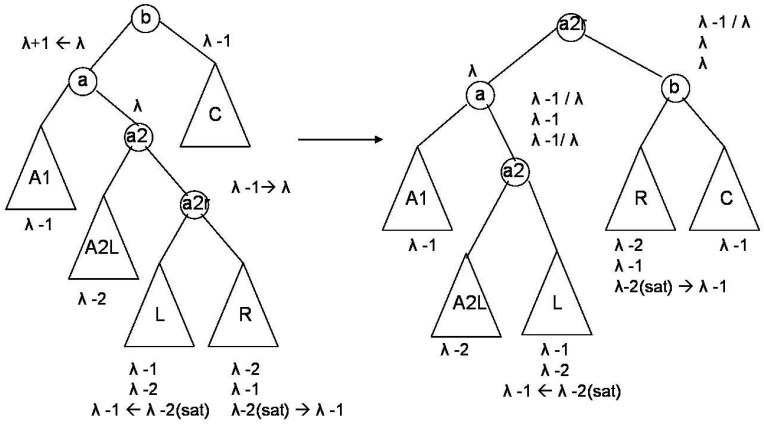
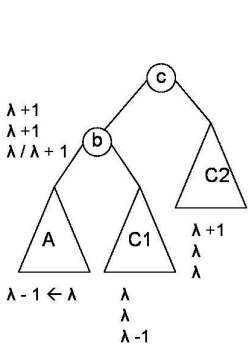
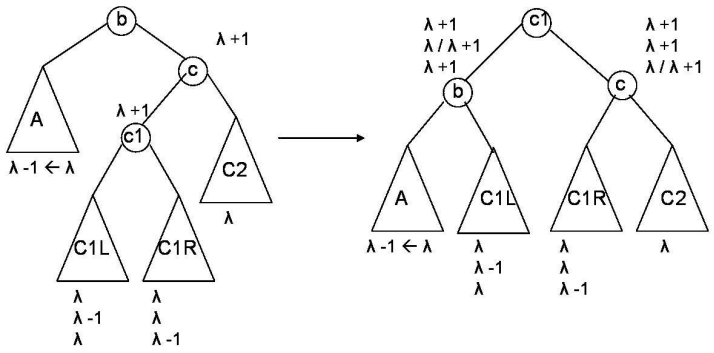



Figure 12: (Hypothetical) unbalancing insertion in right branch of left child ($A2$). Case 3*: tree before and after the triple rotation for * insertion case in $A2R$




(a) Cases 1,2,3: tree before and after a left rotation




(b) Case 4: tree before and after a double rotation.

Figure 13: Unbalancing erase in left child ($A$).

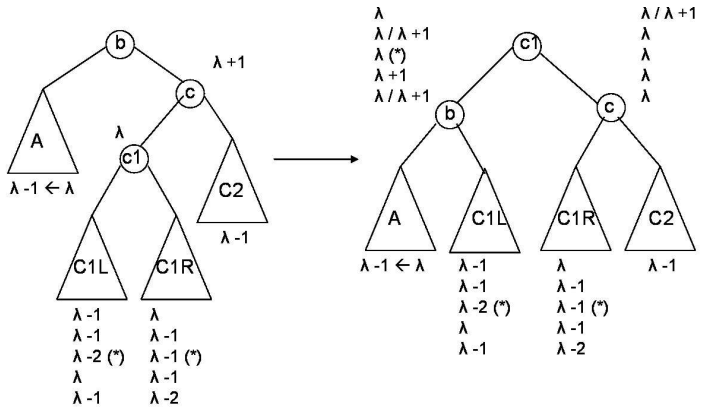
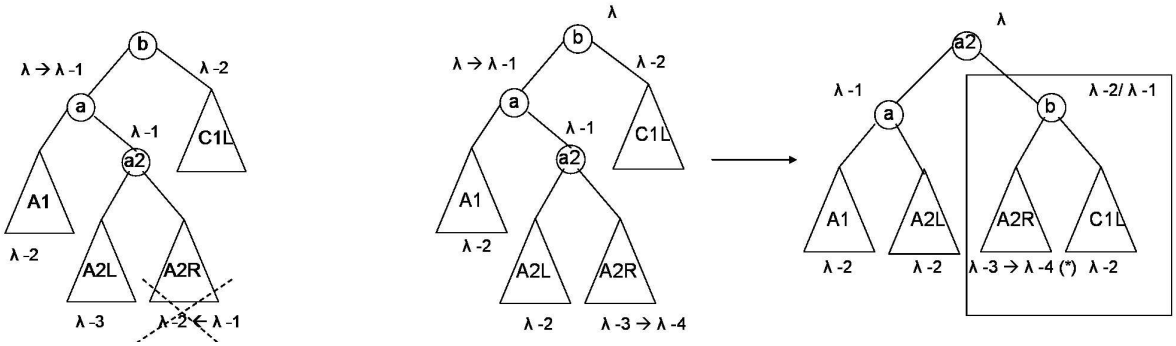



Figure 14: Unbalancing erase in left child ($A$). Case 5: tree before and after the double rotation.




(a) Initial impossible configuration, analogous to insertion's 3.*

(b) Case where $b$ increment is not enough. Inside the square is found the unbalanced subtree to be the starting point at the next erase step

Figure 15: Subtree $b$ analysis for case 5* of unbalancing erase in left child ($A$).

**A.2 Erase by key.** There are 5 basic possible configurations, shown in figure 16. They are the following:

1. $\ell(C1) = \lambda,\ \ell(C2) = \lambda + 1$
2. $\ell(C1) = \lambda,\ \ell(C2) = \lambda$
3. $\ell(C1) = \lambda - 1,\ \ell(C2) = \lambda$
4. $\ell(C1) = \lambda + 1,\ \ell(C2) = \lambda$
5. $\ell(C1) = \lambda,\ \ell(C2) = \lambda - 1$

For the three first cases a simple left rotation suffices (see figure 13(a)). On the other hand, for case 4 and 5, at least a double rotation is needed given that $\ell(C1) < l(C2)$. Specifically, for case 4 a double rotation suffices (see figure 13(b)), but it does not for case 5 (see figure 14). Specifically, an incoherence will arise (marked with *) if $\ell(C1L) = \lambda - 2$ and erase is not successful ($\ell(A) = \lambda$).

In order to solve it, size of right child of left child obtained after the double rotation (that is, new right child of $b$) must be incremented. In fact, subtree $b$ configuration is analogous to the initial configuration of an unbalancing insertion in left child, except for transition direction, and so share some similarities and differences. So, next, $b$ possible configurations, different from those of insertion, are discussed.

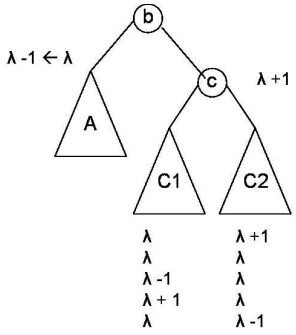



Figure 16: Unbalancing erase in left child ($A$). Possible configurations.

First of all, transition direction makes that insertion configuration 3.* is not possible for subtree $b$. On the other hand, there is a new configuration (marked with * in figure 15(b)) where $b$ increment is not enough. However, right subtree of the result of incrementing $b$ (so $b$ again) corresponds to the initial configuration of an unbalancing erase in left child. Therefore, this situation is solved starting next step at $b$, instead of starting at $b$ left child (the case for situations where $b$ increment is enough).