Alcom-FT Technical Report Series
ALCOMFT-TR-01-167

A New Method for Balancing Binary Search Trees *

Salvador Roura

Departament de LSI
Universitat Politècnica de Catalunya
E-08028 Barcelona, Catalonia, Spain

June 6, 2001

Abstract

A new balancing method for binary search trees is presented, which achieves logarithmic worst-case cost on searches and updates. The method uses the sizes of the subtrees as balancing information; therefore operations by rank are efficiently performed without any changes in the data structure. Compared to weighted binary search trees [7], which also achieve logarithmic worst-case cost by making use of the sizes of the subtrees, the operations involved with our method are likely to be less costly in most real situations.

1 Introduction

The binary search tree (BST) data structure is fundamental to computer science. Since BSTs perform poorly when they are skewed, many variants of balanced BSTs have been devised so far. Weighted BSTs [7] achieve logarithmic worst-case cost by using the sizes of the subtrees as balancing information. Other variants, like AVL trees [1] and red-black trees [4], use information different from the sizes of the subtrees; thus rank operations are not efficiently supported unless an additional field is included at every node. The same comment applies to splay trees [9] and general balanced trees [2], which achieve logarithmic amortised costs without storing any structural information at the nodes. Other variants of balanced trees make use of the sizes of the subtrees but do not guarantee logarithmic worst-case cost; for instance, randomised BSTs [6].

This paper presents a new balancing method for BSTs, which, like weighted BSTs, achieves logarithmic worst-case cost by using the sizes of the subtrees as balancing information. So let us first briefly recall weighted BSTs. Suppose that L and R are the subtrees of a weighted BST, with x and y leaves respectively, and assume w.l.o.g. that $x \leq y$. The balancing property of weighted BSTs states that $y < (1+\sqrt{2})x$, or alternatively, that $2y^2 < (x+y)^2$, which is anyway an expensive property to check. This seems to be the main reason not to use weighted BSTs as default balancing method: "However, it appears that the bookkeeping required for maintaining weight balance takes more time than Algorithm A[1] ..." [5, page 476]. As shown

*This research was partially supported by the IST Programme of the EU IST-1999-14186 (ALCOM-FT), by TIC2000-1970-CE, and by the project DGES PB98-0926 (AEDRI).

[1] Insertion in AVL trees.

in the next sections, the method introduced in this paper is likely to be more efficient than weighted BSTs in most practical situations.

The next sections are organised as follows. Section 2 introduces the main definitions, including the concept of Logarithmic BST (LBST), and proves that the height of an LBST is always logarithmical w.r.t. the size of the tree. Section 3 presents the insertion and deletion algorithms for LBSTs. The former is implemented in Sect. 4, where some empirical evidence that LBSTs are faster than weighted BSTS is provided. Section 5 ends the paper with some further comments.

2 Basic Definitions

Definition 2.1 *For positive n, let $\ell(n)$ be defined as*

$$\ell(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1 + \lfloor \log_2 n \rfloor, & \text{if } n \geq 1 \end{cases} .$$

Note that, except for $n = 0$, $\ell(n)$ is the largest position with a bit equal to one in the binary representation of n; in other words, $\ell(n)$ is the unique integer such that $2^{\ell(n)-1} \leq n \leq 2^{\ell(n)} - 1$.

Given a BST T, let $|T|$ denote the number of keys in T, and let $\ell(T) = \ell(|T|)$. We call our trees Logarithmic BSTs, since their fundamental property is that, at every node, the discrete logarithm of the size of the left subtree and the discrete logarithm of the size of the right subtree differ at most in one unit.

Definition 2.2 (Logarithmic BST) *A BST T is an LBST if and only if*

- *T is an empty tree,*
- *or T is a non-empty tree with subtrees L and R, such that L and R are LBSTs and $-1 \leq \ell(L) - \ell(R) \leq 1$.*

Let us consider the case where T is non-empty. Let $\lambda = \ell(L)$, and assume w.l.o.g. that $|L| \leq |R|$. Then $\ell(R)$ is either λ or $\lambda + 1$. We analyse both cases separately. Suppose first that $\ell(R) = \lambda$:

- If $\lambda > 0$, from $2^{\lambda-1} \leq |L|, |R| \leq 2^{\lambda} - 1$ we deduce that $2^{\lambda} + 1 \leq |T| \leq 2^{\lambda+1} - 1$. Therefore $\ell(T) = \lambda + 1$.
- If $\lambda = 0$, we have $|T| = 1$ and $\ell(T) = \lambda + 1$ as well.

Suppose now that $\ell(R) = \lambda + 1$:

- If $\lambda > 0$, the fact that $2^{\lambda-1} \leq |L| \leq 2^{\lambda} - 1$ and $2^{\lambda} \leq |R| \leq 2^{\lambda+1} - 1$ implies $3 \cdot 2^{\lambda-1} + 1 \leq |T| \leq 3 \cdot 2^{\lambda} - 1$. In this case $\ell(T)$ equals either $\lambda + 1$ or $\lambda + 2$.
- If $\lambda = 0$, then $|T| = 2$ and $\ell(T) = \lambda + 2$.

These two cases are summarised in Fig. 1, using the symbolism that we will keep for the rest of the paper. Every node is labeled with the name of the subtree rooted at that node. For every combination of the weights and every subtree S, the several possibilities for $\ell(S)$ are shown, unless $\ell(S)$ is the same in all situations (in that case it is shown just once). For

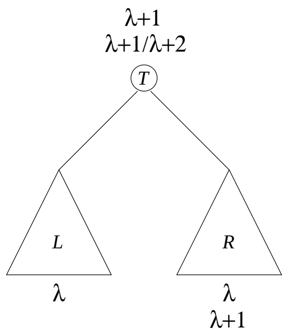

Figure 1: Cases for an LBST with right subtree larger than left subtree

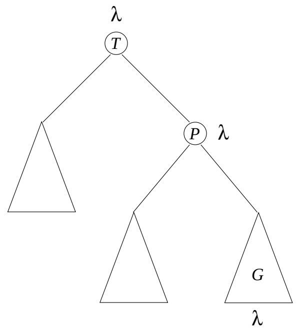

Figure 2: Impossible case for an LBST

example, in Fig. 1 we have that $\ell(L) = \lambda$ and $\ell(R) = \lambda$ imply $\ell(T) = \lambda + 1$, whilst $\ell(L) = \lambda$ and $\ell(R) = \lambda + 1$ imply $\ell(T) = \lambda + 1$ or $\ell(T) = \lambda + 2$.

It is not difficult to prove that the height of an LBST T is always $\Theta(\log |T|)$. Let $\lambda = \ell(T)$. It is enough to notice that every grandchild G of T must satisfy $\ell(G) \leq \lambda - 1$. Otherwise we would have the situation of Fig. 2. Since both the brother of P and the brother of G would include at least $2^{\lambda-2}$ nodes, and G would include at least $2^{\lambda-1}$ nodes, T would have at least $2^{\lambda} + 2$ nodes, which is a contradiction. Therefore, every path from the root of T to a leaf visits at most $2(\lambda + 1)$ nodes, where $\lambda = \log_2 |T| + \mathcal{O}(1)$.

Theorem 2.3 below states that the constant 2 is in fact asymptotically tight, i.e., that the worst-case height of an LBST T is $\sim 2\log_2 |T|$. But first we need to introduce two functions. For every $N \geq 0$, define $I(N) = 4 \cdot 2^N + 2N$, and $J(N) = 6 \cdot 2^N + 2N + 1$. Observe that $4 = I(0) < J(0) < I(1) < J(1) < \cdots$ Hence, for every $n \geq 4$, there is a unique N such that either $I(N) \leq n < J(N)$ or $J(N) \leq n < I(N + 1)$.

Theorem 2.3 *Let $H(n)$ be the maximum height of an LBST with n keys. Then $H(0) = 0$, $H(1) = 1$, $H(2) = H(3) = 2$, and for every $n \geq 4$,*

$$H(n) = \begin{cases} 2N + 3, & \text{if } I(N) \leq n < J(N) \\ 2N + 4, & \text{if } J(N) \leq n < I(N + 1) \end{cases} .$$

(The proof is by induction on N.)

We thus know that, if $I(N) \leq n < I(N+1)$, $H(n) \leq 2(N+2)$. But $N+2 \leq \log_2 I(N) \leq \log_2 n$, and we can conclude that the height of an LBST with $n \geq 2$ keys is never larger than $2\log_2 n$.

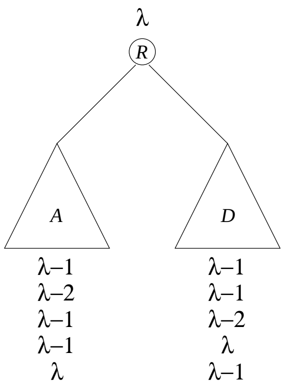

Figure 3: Five possible cases for an LBST R with $\ell(R) = \lambda$

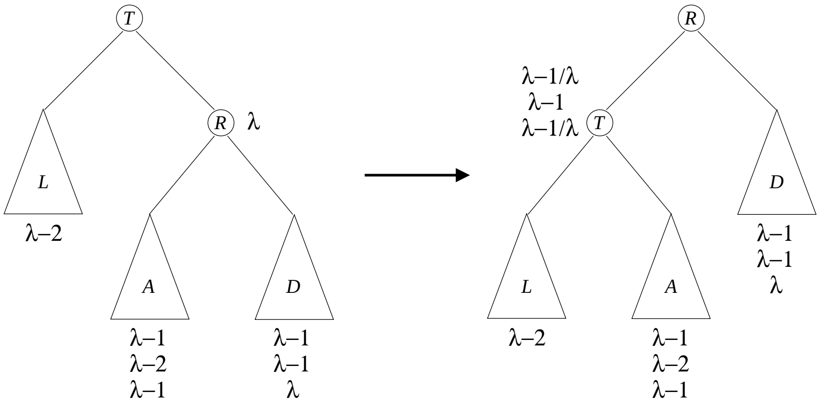

Figure 4: Cases of Fig. 3 for which a single rotation suffices

Note that the constant 2 is asymptotically tight, i.e., $H(n) \sim 2\log_2 n$. Recall that the worst-case height of a weighted BST with n keys is also $\sim 2\log_2 n$.

3 The Insertion and Deletion Algorithms

In the insertion and deletion algorithms, we will make use of the following algorithm, which obtains an LBST from a BST such that its left subtree L and its right subtree R are LBSTs, where $\ell(L) = \lambda - 2$ and $\ell(R) = \lambda$. Let A and D be the left and right subtrees of R, respectively. Figure 3 includes the five possible combinations for $\ell(A)$ and $\ell(D)$, provided that $\ell(R) = \lambda$.

As shown in Fig. 4, a single rotation suffices for the first, second and fourth cases of Fig. 3. For instance, consider the first case, where $\ell(A) = \ell(D) = \lambda - 1$. After the rotation, the left subtree of R, labeled T in the figure, is such that $\lambda - 1 \leq \ell(T) \leq \lambda$, and thus $\ell(T)$ differs in at most one unit with $\ell(D)$.

Figure 5 proves that, for the third case of Fig. 3, a double rotation suffices. Let B and C be the left and right subtrees of A, respectively (notice that $\ell(A) \geq 1$ implies that A is never empty). As in Fig. 3, there are five possible combinations for $\ell(B)$ and $\ell(C)$. For each one and after two rotations, the first rotation between A and R, the second between A and T,

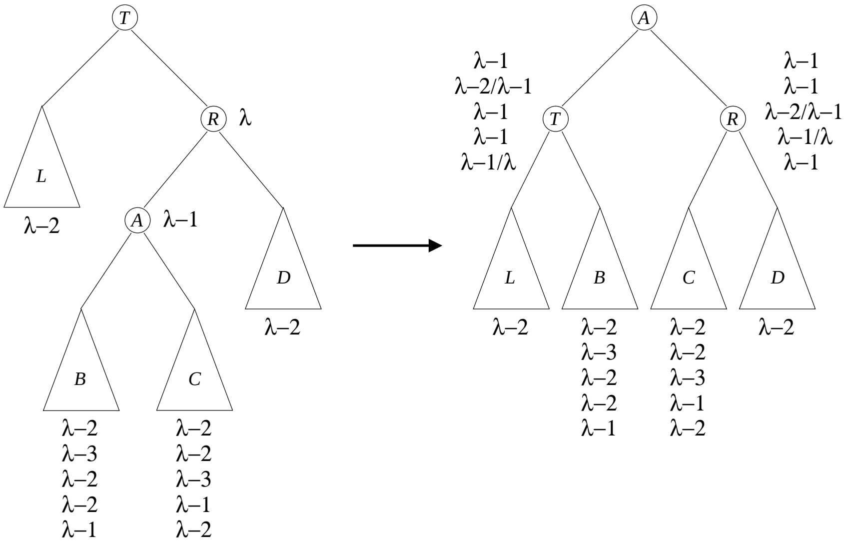

Figure 5: Third case of Fig. 3; a double rotation suffices

the balancing property is reestablished.

The fifth and last case of Fig. 3 also requires a double rotation (see Fig. 6), but this case is slightly different from the case in Fig. 5. Indeed, only three of the five combinations for $\ell(B)$ and $\ell(C)$ are possible here, since $\ell(B) = \lambda - 1$ and $\ell(C) = \lambda$ (or $\ell(B) = \lambda$ and $\ell(C) = \lambda - 1$) together with $\ell(D) = \lambda - 1$ would imply $\ell(R) = \lambda + 1$, which is against the hypotheses.

We are now ready to present the insertion algorithm of a new key x into a given LBST T, which follows the traditional approach of balanced trees:

- If T is empty, return a BST with x as only key.
- Otherwise, let L and R be the left and right subtrees of T, respectively.
  - If x is smaller than the root of T, recursively insert x into L; if afterwards $\ell(L) = \ell(R) + 2$, perform a local update.
  - If x is larger than the root of T, recursively insert x into R; if afterwards $\ell(R) = \ell(L) + 2$, perform a local update.

The local updates mentioned above, meant to reestablish the balancing property of LBSTs, are those included in Figs. 4, 5 and 6. Note that, in fact, only the second and third cases of Fig. 3 are possible here, because $|R|$ must be exactly $2^{\lambda-1}$ after the recursive insertion.

We now consider how to delete a key x from a given LBST T. The deletion algorithm also uses the local updates included in Figs. 4, 5 and 6:

- If T is empty, x is not in T; hence no updates are needed.
- If T has x as unique key, return the empty tree.
- Otherwise, let L and R be the left and right subtrees of T, respectively.
  - If x is smaller than the root of T, recursively delete x from L; if afterwards $\ell(R) = \ell(L) + 2$, perform a local update.
  - If x is larger than the root of T, recursively delete x from R; if afterwards $\ell(L) = \ell(R) + 2$, perform a local update.
  - If x is equal to the root of T, remove it.

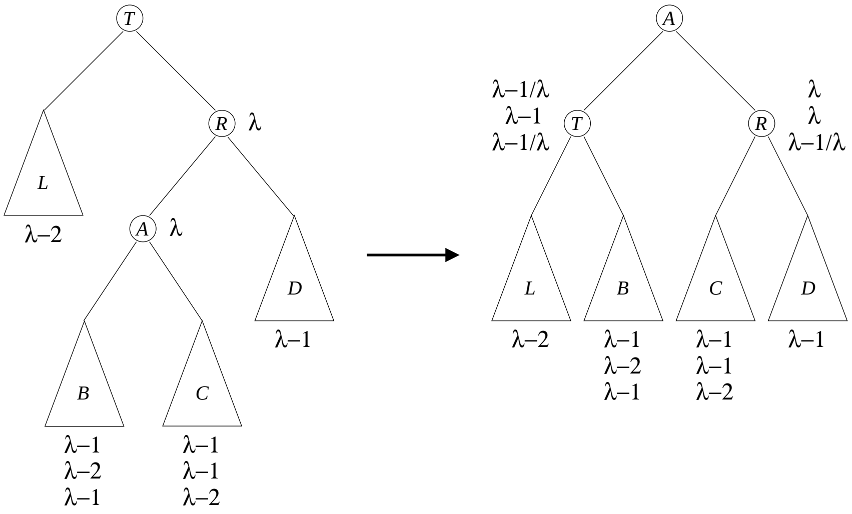

Figure 6: Fifth case of Fig. 3; a double rotation suffices

The removal of the root of T can be done in several ways. For instance, we can replace the root of T by the minimum of the keys in R when $|L| \le |R|$, or by the maximum of the keys in L when $|L| > |R|$. The algorithm to extract the minimum of the keys from a non-empty LBST T is quite simple:

- If the left subtree of T is empty, the minimum key is the root of T, and the right subtree of T is the remaining tree.
- Otherwise, recursively extract the minimum from the left subtree of T; afterwards perform a local update if necessary.

Once again, the local updates required here are identical to those of the insertion algorithm and deletion algorithm. The algorithm to extract the maximum of the keys of a non-empty LBST is symmetrical.

4 Implementing LBSTs

The *C* code presented in this paper implements the insertion algorithm for LBSTs. This code has been written to emphasise the simplicity of the algorithms, so faster programmes could be obtained at the price of obscuring the code.

As shown in Fig. 7, an LBST is identified with a pointer to its root node. Every node contains a key, two pointers to its children and a counter of type `size` with the number of keys in the subtree rooted at the node. We assume that keys and counters are long integers, and that empty trees are equal to `null`, which is a pointer to a node with no key and size 0. The call `singleton(x)` returns an LBST with `x` as only key.

Figure 7 also includes some fundamental functions. Given two sizes `a` and `b`, the call `smaller_ell(a, b)` tells us whether $\ell(\mathtt{a}) < \ell(\mathtt{b})$ or not. Let $\lambda = \ell(\mathtt{b})$. If $\mathtt{a} \ge \mathtt{b}$, we trivially have $\ell(\mathtt{a}) \ge \lambda$. Otherwise, we perform a logical "and" of `a` and `b`, shift the result one bit to the left, and compare the final result (let us call it α) against `b`. Assume that $\lambda \ge 1$. If $\ell(\mathtt{a}) = \lambda$, we have $\ell(\alpha) = \lambda + 1$; hence $\alpha > \mathtt{b}$ and the function returns `FALSE`. If $\ell(\mathtt{a}) < \lambda$,

Table 1: Empirical average search cost, height, and number of single and double rotations of LBSTs built under random and sorted insertions

	Random order				Increasing order			
# keys	A.S.C.	Height	# S.R.	# D.R.	A.S.C.	H.	# S.R.	# D.R.
15625	13.254	17.60	3415.78	3413.68	12.952	15	15611	0
31250	14.276	18.97	6829.06	6822.12	13.952	16	31235	0
62500	15.290	20.15	13636.65	13660.82	14.952	17	62484	0
125000	16.316	21.45	27336.78	27282.39	15.952	18	124983	0
250000	17.340	22.8	54678.0	54634.3	16.952	19	249982	0
500000	18.366	23.9	109496.4	109073.1	17.951	20	499981	0
1000000	19.376	25.3	218454.2	218615.9	18.951	21	999980	0
2000000	20.379	26.3	436917.6	437033.4	19.951	22	1999979	0

then α is at most $2(\mathtt{b} - 2^{\lambda-1})$. This happens when $\ell(\mathtt{a}) = 2^{\lambda-1} - 1$, i.e., when the digital representation of `a` includes as many bits equal to one as possible. Since $\mathtt{b} < 2^{\lambda}$, we have $\alpha \leq 2\mathtt{b} - 2^{\lambda} < \mathtt{b}$, and the function returns `TRUE`, es expected. The function always returns `FALSE` for the special case $\lambda = 0$.

The call `rot_left(t)` returns the result of rotating `t` to its left, updating conveniently the fields `b->s` and `t->s`. The call `inc_left(t)` returns the result of balancing `t` by means of the rotations in Figs. 4, 5 and 6, assuming that $\ell(\mathtt{t\text{->}r}) = \ell(\mathtt{t\text{->}l}) + 2$. Notice that one call to the function `smaller_ell()` suffices to discriminate the first, second and fourth cases from the third and fifth cases of Fig. 3. The functions `rot_right()` and `inc_right()` are easily obtained from the functions `rot_left()` and `inc_left()`.

Given two sizes `A` and `B` such that $\ell(\mathtt{A}) \leq \ell(\mathtt{B}) + 1$, we use the macro `balanced(A, B)` to know whether $\ell(\mathtt{B}) \leq \ell(\mathtt{A}) + 1$ or not. Note that the logical instructions in this macro and in the function `smaller_ell()` are usually fast in most computers.

The left side of Table 1 shows the empirical average search cost and height of an LBST produced after n random insertions into an initially empty tree, for several values of n. Table 1 also includes the total number of single and double rotations that take place during the construction of the LBST. The first four rows are averaged over 100 executions; the last four rows are averaged over 10 executions. The right side of Table 1 includes the same measures, this time for LBSTs built in increasing order. We define the average search cost as the internal path length divided by the number of keys.

It is not difficult to prove that any LBST obtained after inserting the keys in increasing order is (almost) perfectly balanced (and hence the results for sorted insertions given in Table 1). An exact analysis of random LBSTs is much harder. However, the empirical results provided in Table 1 indicate that the average search cost is $\sim \beta \cdot \log_2 n$ for some constant β very close to 1, which can be regarded as optimal for practical purposes. A similar result holds for other variants of balanced BSTs.

A single insertion into an LBST T may require up to $\Theta(\log |T|)$ rotations. However, less than n rotations are enough to build in increasing order an LBST with n keys, and, from Table 1, the total number of rotations under random insertions also seems to be $\Theta(n)$. The next theorem states that this is not a coincidence.

```
typedef long key, size;
typedef struct node *lbst;
typedef struct { key k; lbst l, r; size s; } node;

int smaller_ell(size a, size b)
{ if (a >= b) return FALSE;
  return ((a&b)<<1) < b;
}

lbst rot_left(lbst t)
{ lbst b = t->r; t->r = b->l; b->l = t;
  b->s = t->s; t->s = 1 + t->l->s + t->r->s;
  return b;
}

lbst inc_left(lbst t)
{ if (smaller_ell(t->r->r->s,t->r->l->s)) t->r = rot_right(t->r);
  return rot_left(t);
}

#define balanced(A, B) !(smaller_ell(A, (B)>>1))

lbst Insert(key x, lbst t)
{ if (t == null) return singleton(x);
  t->s++;
  if (x < t->k)
    { t->l = Insert(x, t->l);
      if (!balanced(t->r->s, t->l->s)) t = inc_right(t);
    }
   else
    { t->r = Insert(x, t->r);
      if (!balanced(t->l->s, t->r->s)) t = inc_left(t);
    }
  return t;
}
```

Figure 7: Insertion algorithm in C

Table 2: Empirical times (in seconds) to build LBSTs, proper weighted BSTs, relaxed weighted BSTs, AVL trees and red-black tres in random order

# keys	LBSTs	WBSTs	3WBSTs	AVLs	RBTs
15625	0.01424	0.01711	0.01437	0.01791	0.02811
31250	0.04909	0.05457	0.04933	0.05543	0.07658
62500	0.14171	0.15370	0.14291	0.15323	0.19947
125000	0.36025	-	0.36261	0.40145	0.48025
250000	0.8667	-	0.8694	0.9547	1.1331
500000	2.0340	-	2.0344	2.2339	2.5683
1000000	4.6638	-	4.6783	5.1646	5.7938
2000000	10.6002	-	10.6326	11.6718	13.0526

Theorem 4.1 *The total number of rotations required to build an LBST T from an empty tree is $\mathcal{O}(|T|)$.*

(The theorem can be proved by means of the potential method. Note that the same property is true for weighted BSTs [3].)

Tables 2 and 3 include some empirical results about the time efficiency of our algorithms. The tests consisted in the construction of BSTs with n keys, for five different balancing strategies and several values of n. Two limiting situations were considered, namely when keys are inserted at random (Table 2), and when keys are inserted in increasing order (Table 3). The times, expressed in seconds, were obtained with a PC[2], and averaged over 1000 executions for the first four rows, and over 100 executions for the last four rows.

The code for LBSTs was the one provided in this paper, with the function `smaller_ell()` replaced by a macro for efficiency. The code used for weighted BSTs was the same except for the balancing condition, which was "$2y^2 < (x+y)^2$", where x and y are respectively the number of leaves of the "small" subtree and of the "large" subtree (this condition was only checked after an insertion into the "large" subtree). The computation of $2y^2$ and $(x+y)^2$ caused an overflow for large values of n; hence the empty fields in Tables 2 and 3. A relaxed variant of weighted BSTs was also implemented, with "$y < 3x$" as balancing condition (we call these trees 3WBSTs). The property for 3WBSTs is cheaper to check than the one for proper weighted BSTS, but it can degrade somehow the tree, since the worst-case height becomes $\sim \ln 2/\ln(4/3) \cdot \log_2 n \simeq 2.40942 \log_2 n$. The code for AVL trees and the code for red-black trees were taken from [10, page 153] and from [8, page 554] respectively. Both codes were slightly modified, to make them comparable with the code of the rest of balancing strategies.

Under random insertions, LBSTs and 3WBSTs performed similarly, and faster than WBSTs. Since the trees obtained are very well balanced in all the cases, the crucial factor was the high cost of evaluating the balancing property of WBSTs. For sorted insertions, both WBSTs and 3WBSTs were about 30 percent slower than LBSTs; note that 3WBSTs built in increasing order are not perfectly balanced. In general, red-black trees achieved the worst times, while AVL trees turned out to be about 10 percent slower than LBSTs. The implemen-

[2] Pentium(r) II Processor, 128 MB of RAM, DJGPP *C* compiler.

Table 3: Empirical times (in seconds) to build LBSTs, proper weighted BSTs, relaxed weighted BSTs, AVL trees and red-black tres in increasing order

# keys	LBSTs	WBSTs	3WBSTs	AVLs	RBTs
15625	0.01125	0.01467	0.01474	0.01253	0.03040
31250	0.02651	0.03447	0.03431	0.03018	0.06751
62500	0.05939	0.07635	0.07672	0.06719	0.14485
125000	0.12968	0.16485	0.16570	0.14448	0.31201
250000	0.2746	-	0.3541	0.3072	0.6568
500000	0.5844	-	0.7600	0.6559	1.3985
1000000	1.2266	-	1.6241	1.3908	2.9786
2000000	2.5781	-	3.4708	2.9299	6.3003

tation of AVL trees was the only one without a counter field with the sizes of the subtrees. If rank operations were needed, this extra field should be updated conveniently during the insertions, which would increase the insertion time of AVL trees. Note that the time of sorted insertions was much smaller than the time of random insertions. This was probably due to the high memory locality of the former, which induced an efficient use of the cache memory.

5 Final Remarks

Other operations for BSTs, like joins, splits, unions, intersections, set subtractions, and so on, can be easily and efficiently implemented using the ideas in this paper. Moreover, since a counter field is kept at each node, rank operations are efficiently performed without any further modification of our data structure.

There are several variants of LBSTs that may be considered. First, it is possible to use the number of leaves instead of the number of keys as balancing information. The algorithms obtained perform similarly to the ones presented in this paper. On the other hand, we could relax the condition in Definition 2.2 to be $-k \leq \ell(L) - \ell(R) \leq k$ for some constant $k \geq 1$; alternatively, we could define $\ell(n) = 1 + \lfloor \log_b n \rfloor$ for some base $b \neq 2$, or combine several of these possibilities. In general, LBSTs with large k (or with large b) perform less rotations than plain LBSTs, since its balancing condition is less astringent. On the other hand, its worst-case height increases as k (or b) increases.

Finally, it must be said that the experimental results presented in this paper are only an indication that LBSTs can be a practical alternative to traditional balancing strategies. Nevertheless, there are many factors that should be considered in our election: time (and space) efficiency, algorithm and code complexity, variety of supported operations (rank operations, set operations, etc.), ease of obtaining non-recursive versions to increase efficiency, average and worst-case cost (measured as number of visited nodes, rotations, etc.), and so on.

Acknowledgments

The comments of Josep Díaz, Rolf Fagerberg and Conrado Martínez improved the presentation of this work.

References

[1] G.M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Dokladi Akademia Nauk SSSR*, 146(2):263–266, 1962. English translation in Soviet Math. Doklay 3, 1259-1263, 1962.

[2] A. Andersson. General balanced trees. *Journal of Algorithms*, 30:1–18, 1999.

[3] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *TCS: Theoretical Computer Science*, 11:303–320, 1980.

[4] L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. of the 19th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 8–21, October 1978.

[5] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA, 2nd edition, 1998.

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

[7] J. Nievergelt and E. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[8] R. Sedgewick. *Algorithms in C*. Addison-Wesley, 3rd edition, 1998.

[9] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[10] M.A. Weiss. *Data Structures & Algorithm Analysis in C++*. Addison-Wesley, 2nd edition, 1999.