PREPRINT. In Proc. Workshop on Algorithms and Data Structures, pages 60-71. Springer Verlag, 1993.

Balanced Search Trees Made Simple

Arne Andersson*

Department of Computer Science, Lund University, Box 118, S-221 00 Lund, Sweden

Abstract. As a contribution to the recent debate on simple implementations of dictionaries, we present new maintenance algorithms for balanced trees. In terms of code simplicity, our algorithms compare favourably with those for deterministic and probabilistic skip lists.

1 Introduction

It is well known that there is a huge gap between theory and practice in computer programming. While companies producing computers or cars are anxious to use the best technology available – at least they try to convince their customers that they do so – the philosophy is often different in software engineering; it is enough to produce programs that "work." Efficient algorithms for sorting and searching, which are taught in introductory courses, are often replaced by poor methods, such as bubble sorting and linked lists. If your program turns out to require too much time or space you advice your customer to buy a new, heavier and faster, computer.

This situation strongly motivates an extensive search for simple and short-coded solutions to common computational tasks, for instance worst-case efficient maintenance of dictionaries. Due to its fundamental character, this problem is one of the most well-studied in algorithm design. Until recently, all solutions have been based on balanced search trees, such as AVL-trees [1], symmetric binary B-trees [6] (also denoted red-black trees [8]), SBB(k)-trees [4], weight-balanced trees [11], half-balanced trees [12], and k-neighbour trees [9]. However, none of them has become the structure of choice among programmers as they are all cumbersome to implement. To cite Munro, Papadakis, and Sedgewick [10], the traditional source code for a balanced search tree "contains numerous cases involving single and double rotations to the left and the right".

In the recent years, the search for simpler dictionary algorithms has taken a new, and quite successful, direction by the introduction of some "non-tree" structures. The first one, *the skip list*, introduced by Pugh [14] is a simple and elegant randomized data structure. A worst-case efficient variant, *the deterministic skip list*, was recently introduced by Munro, Papadakis, and Sedgewick [10].

The important feature of the deterministic skip list is its *code simplicity*. As pointed out by Munro et al., the source code for insertion into a deterministic skip list is simpler (or at least shorter) than any previously presented code for a

* Arne.Andersson@dna.lth.se

balanced search tree. In addition, the authors claimed that the code for deletion would be simple, although it was not presented.

In this article we demonstrate that *the binary B-tree*, introduced by Bayer [5], may be maintained by very simple algorithms. The simplicity is achieved from three observations:

– The large number of cases occurring in traditional balancing methods may be replaced by two simple operations, called *Skew* and *Split*. Making a *Skew* followed by a *Split* at each node on the traversed path is enough to maintain balance during updates.
– The representation of balance information as one bit per node creates a significant amount of book-keeping. One short integer ($o(\log \log n)$ bits) in each node makes the algorithms simpler.
– The deletion of an internal node from a binary search tree has always been cumbersome, even without balancing. We show how to simplify the deletion algorithm by the use of two global pointers.

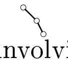
As a matter of fact, the coding of our algorithms is even shorter than the code for skip lists, both probabilistic and deterministic, are. In our opinion, it is also simpler and clearer. Hence, the binary search tree may compete very well with skip lists in terms of simplicity.

In Section 2 we present the new maintenance algorithms for binary B-trees, and in Section 3 we discuss their implementation. Section 4 contains a comparison between binary B-trees and deterministic skip lists. We also make a brief comparison with the probabilistic skip list. Finally, we summarize our results in Section 5.

2 Simple Algorithms

The binary B-tree, BB-tree, was introduced by Bayer in 1971 [5] as a binary representation of 2-3 trees [2]. Using the terminology in [4], we say that a node in a 2-3 tree is represented by a *pseudo-node* containing one or two binary nodes. Edges inside pseudo-nodes are *horizontal* and edges between pseudo-nodes are *vertical*. Only right-edges are allowed to be horizontal. In order to maintain balance during updates, we have to store balance information in the nodes. One bit, telling whether the incoming edge (or the outgoing right-edge) is horizontal or not, would be enough. However, in our implementation we chose to store an integer *level* in each node, corresponding to the vertical height of the node. Nodes at the bottom of the tree are on level 1.

Briefly, all representations of B-trees, including binary B-trees, red black trees, and deterministic skip lists, are maintained by two basic operations: joining and splitting of B-tree nodes. In a binary tree representation, this is performed by rotations and change of balance information. The reason why the algorithms become complicated is that a pseudo-node may take many different shapes, causing many special cases. For example, adding a new horizontal edge to a pseudo-node of shape or  may result in five different shapes, namely

 ,  ,  ,  , or  . These possibilities give rise to a large number of different cases, involving more or less complicated restructuring operations.

Fortunately, there is a simple rule of thumb that can be applied to reduce the number of cases:

Make sure that only right-edges are horizontal
before you check the size of the pseudo-node.

Using this rule, the five possible shapes in the above will reduce to two, namely  and  , only the last one will require splitting.

In order to apply our rule of thumb in a simple manner, we define two basic restructuring operations (p is a binary node):

Skew (p): Eliminate horizontal left-edges below p. This is performed by following the right path from p, making a right rotation whenever a horizontal left-edge is found.

Split (p): If the pseudo-node rooted at p is too large, split it by increasing the level of every second node. This is performed by following the right path from p, making left rotations.

These operations are simple to implement as short and elegant procedures. They also allow a conceptually simple description of the maintenance algorithms:

Insertion

1. Add a new node at level 1.
2. Follow the path from the new node to the root. At each binary node p perform the following:
   (a) *Skew* (p)
   (b) *Split* (p)

Deletion

1. Remove a node from level 1 (the problem of removing internal nodes is discussed in Section 3).
2. Follow the path from the removed node to the root. At each binary node p perform the following:
   (a) If a pseudo-node is missing below p, i. e. if one of p's children is two levels below p, decrease the level of p by one. If p's right-child belonged to the same pseudo-node as p, we decrease the level of that node too.
   (b) *Skew* (p)
   (c) *Split* (p)

The algorithms are illustrated in Figures 1 and 2.

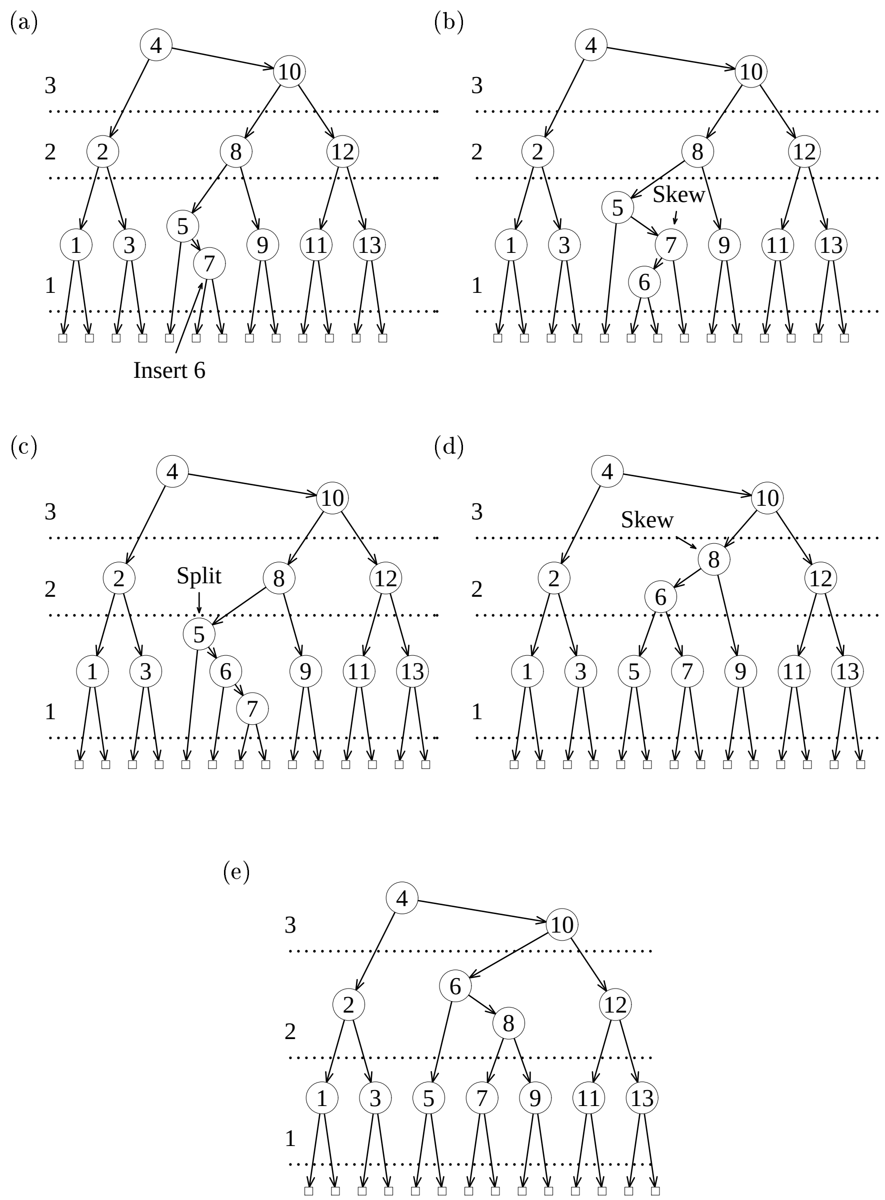


Fig. 1. *Example of insertion into a BB-tree. The levels are separated by horizontal lines.*

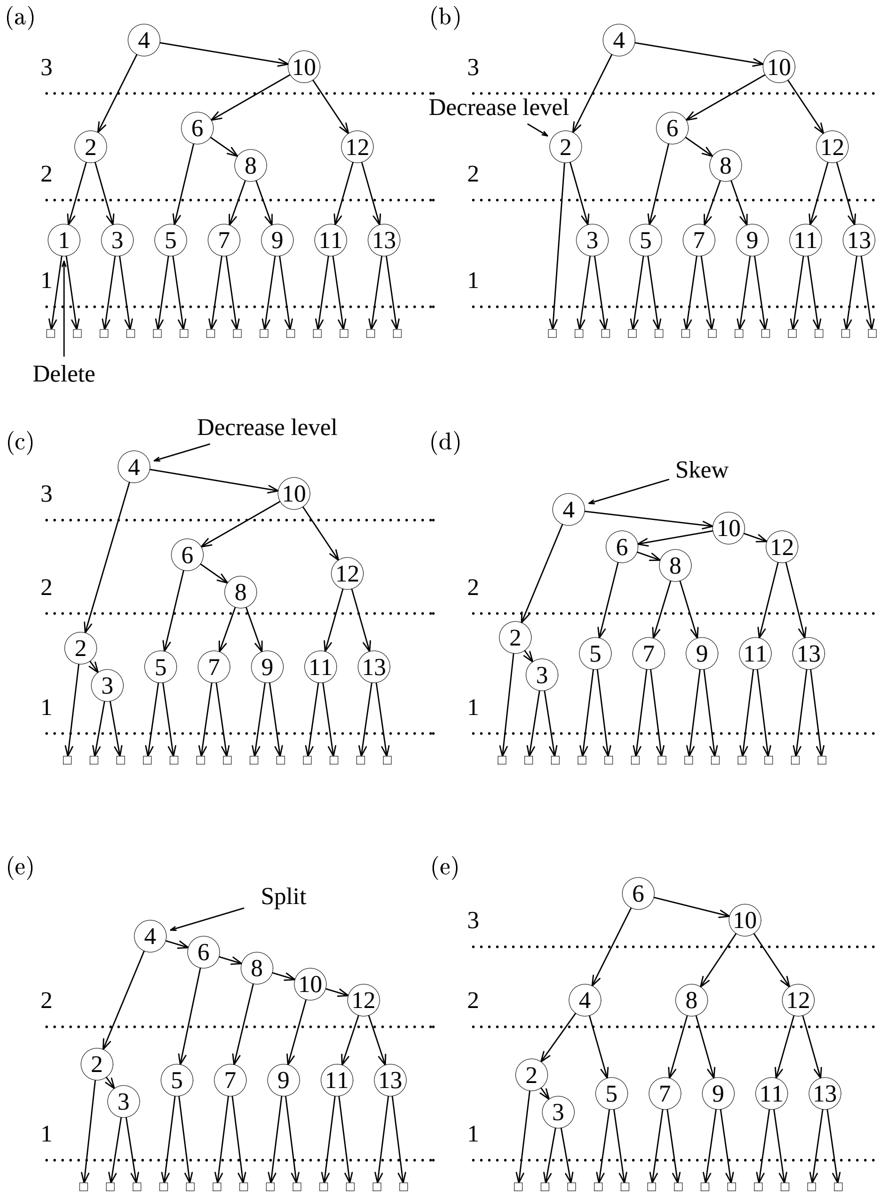


Fig. 2. *Example of deletion.*

3 Simple implementation

Below, we give the complete code for declarations and maintenance of a BB-tree.

We use the well-known technique of having a *sentinel* [15] at the bottom of the tree. In this way, we do not have to consider the existence of a node before examining its level. Each time we try to find the level of a node outside the tree, we examine the level of the sentinel, which is initialized to zero.

The declaration and initialization is straightforward. As a sentinel we use the global variable *bottom*, which has to be initialized. Note that more than one tree can share the sentinel. A pointer variable is initialized as an empty tree simply by making it point to the sentinel.

In our implementation, we use the following code for declarations of data types and global variables and for initialization:

```
type data = ...;
     Tree = ↑node;
     node = record
                 left, right: Tree;
                 level: integer;
                 key: data;
              end;
var bottom, deleted, last: Tree;
```

```
procedure InitGlobalVariables;
begin
  new (bottom);
  bottom↑.level := 0;
  bottom↑.left := bottom;
  bottom↑.right := bottom;
  deleted := bottom;
end;
```

The restructuring operations *Skew* and *Split* may be coded in several ways: they may be in-line coded into the insertion and deletion procedures, or they may be coded as separate procedures traversing a pseudo-node, making rotations whenever needed. The code given here is a third possibility where *Skew* and *Split* are coded as procedures operating on a single binary node. This is enough for the restructuring required during insertion, but during deletion we need three calls of *Skew* and two calls of *Split*. The fact that these calls are sufficient is not hard to show, we leave the details as an exercise.

In order to handle deletion of internal nodes without a lot of code, we use two global pointers *deleted* and *last*. These pointers are set during the top-down traversal in the following simple manner: At each node we make a binary comparison, if the key to be deleted is less than the node's value we turn left, otherwise we turn right (i.e. even if the searched element is present in the node we turn right). We let *last* point to each internal node on the path, and we let *deleted* point to each node where we turn right. When we reach the bottom of the tree, *deleted* will point to the node containing the element to be deleted (if it is present in the tree) and *last* will point to the node which is to be removed (which may be the same node). Then, we just move the element from *last* to *deleted* and remove *last*.

Altogether, insertion and deletion may be coded in the following way:

```
procedure Skew (var t: Tree);
var temp: Tree;
begin
  if t↑.left↑.level = t↑.level then
  begin { rotate right }
    temp := t;
    t := t↑.left;
    temp↑.left := t↑.right;
    t↑.right := temp;
  end;
end;

procedure Split (var t: Tree);
var temp: Tree;
begin
  if t↑.right↑.right↑.level = t↑.level then
  begin { rotate left }
    temp := t;
    t := t↑.right;
    temp↑.right := t↑.left;
    t↑.left := temp;
    t↑.level := t↑.level + 1;
  end;
end;

procedure Insert (var x: data;
    var t: Tree; var ok: boolean);
begin
  if t = bottom then begin
    new (t);
    t↑.key := x;
    t↑.left := bottom;
    t↑.right := bottom;
    t↑.level := 1;
    ok := true;
  end else begin
    if x < t↑.key then
      Insert (x, t↑.left, ok)
    else if x > t↑.key then
      Insert (x, t↑.right, ok)
    else ok := false;
    Skew (t);
    Split (t);
  end;
end;

procedure Delete (var x: data;
    var t: Tree; var ok: boolean);
begin
  ok := false;
  if t <> bottom then begin

{ 1: Search down the tree and }
{ set pointers last and deleted. }
    last := t;
    if x < t↑.key then
      Delete (x, t↑.left, ok)
    else begin
      deleted := t;
      Delete (x, t↑.right, ok);
    end;

{ 2: At the bottom of the tree we }
{ remove the element (if it is present). }
    if (t = last) and (deleted <> bottom)
    and (x = deleted↑.key) then
    begin
        deleted↑.key := t↑.key;
        deleted := bottom;
        t := t↑.right;
        dispose (last);
        ok := true;
    end

{ 3: On the way back, we rebalance. }
    else if (t↑.left↑.level < t↑.level-1)
    or (t↑.right↑.level < t↑.level-1) then
    begin
      t↑.level := t↑.level -1;
      if t↑.right↑.level > t↑.level then
        t↑.right↑.level := t↑.level;
      Skew (t);
      Skew (t↑.right);
      Skew (t↑.right↑.right);
      Split (t);
      Split (t↑.right);
    end;
  end;
end;
```

4 Discussion

Since the BB-tree has been known for a long time, an obvious question is whether our algorithms are actually simpler than known algorithms. A look in the original paper by Bayer [5] or in the textbook by Wirth [15] will show that this is the case.

First, by introducing a simple rule of thumb, we avoid all the various cases that occur in standard algorithms. Second, if we use just one bit in each node, we have to change the balance information in the two involved nodes after each rotation. This is not required in our case. Third, during deletion, the difference in level between a node and its child may become 0, 1, or 2. This is easily detected when the levels are stored explicitly in the nodes. If we use one bit in each node, we must use more bookkeeping, causing more (and quite tricky) code. Fourth, we have shown how to simplify the deletion of internal nodes. combining those observations, the implification becomes considerable.

We believe that our version of BB-trees is very suitable for classroom and textbook presentations. Since it fills less than one page, we can provide the student/reader with the entire code. Optimizing considerations, such as decreasing the number of bits used for balance information, are nice exercises for the interested student.

4.1 BB-trees versus deterministic skip lists

We start our comparison by examining the code for deterministic skip lists. Then, we discuss some other aspects.

Program code: Munro et al. presented a short and elegant source code for declaration, initialization, insertion, and search in a deterministic skip list. There also exist a preliminary version of the (quite complicated and long) deletion algorithm [13]. The code is given in C, but can easily be translated into Pascal. Doing so, we find that the total code for deterministic skip lists is about 40% longer than our code for BB-trees. The length was compared using the established method of counting lexicographical units (tokens) [7].

In addition, code for BB-trees appears to be simpler and cleaner. In our opinion, the occurrence of complicated pointer expressions, like $x\uparrow.d\uparrow.r\uparrow.r\uparrow.r\uparrow.key$ makes the code for deterministic skip lists harder to understand. The procedure for insertion into a DSL also contains a “hidden GOTO-statement” in the form of a **return** statement.

Recursion vs. non-recursion: A reader might argue that our comparison is "unfair" since we compare recursive code with non-recursive. To such an objection we answer

- Recursion is no disadvantage when aiming for simple and clear code.
- It does not seem likely that a recursive version of the DSL would be simpler than the BB-tree. At least, this remains to show.
- A stack of logarithmic size creates no problem in practice.
- The only possible drawback of recursion is that the execution time may become slightly longer. However, the subject treated here is to give simple and short algorithms, *not* to optimize code. If a minimal constant factor in execution time is very important, neither the binary B-tree nor the deterministic skip list is claimed to be the ideal choice.

Reserved key values: In the suggested implementation of deterministic skip lists two key values (*max* and *max + 1*) are used for special purposes. This has some drawbacks, which becomes evident when the DSL is to be used as a general-purpose abstract data type. If any of these keys are inserted or searched for, the program will fail. Furthermore, for some types of data, such as real numbers, it may be difficult to find suitable values for *max* and *max + 1*. Indeed, this "minor" problem may cause a great deal of confusion and irritation.

Note that a DSL does not necessarily require these reserved keys; they may be removed at the cost of more code.

Execution time: The purpose of this paper is not to minimize the execution time. However, for the sake of completeness we have run some experiments.

At the moment of testing, no code for deletion in a DSL was published, therefore we only compared the execution time for insertions and searches.

In our experiments, we used a Pascal version of the source code by Munro et al. [10]. For binary tree search we used the (non-recursive) algorithm discussed in [3].

The experiments were made in two environments: Sun-Pascal on a Sun SPARCstation 1 and Turbo-Pascal 6.0 on a Victor 2/86. We used two kinds of data: random real numbers and random strings. The strings where of type **packed array** [1..20] **of** *char* and consisted of randomly chosen capitol letters from the English alphabet. The dictionaries where of size 100 – 1000 on the PC and 100 – 10 000 on the Sun. (Trying to insert 2000 strings in a deterministic skip list, the PC run out of address space). In order to measure the cost of inserting/searching for n elements, the following experiment was made:

1. Start the clock.
2. Repeat ten times: Make n random insertions or search all n elements.
3. Stop the clock.
4. Repeat step 1 – 3 ten times, compute average time and standard deviation.

environment	input	n	insertion BB-tree	insertion DSL	search BB-tree	search DSL (1)	search DSL (2)
Turbo Pascal	real	100	1126	1587	752	1292	983
		200	1296	1825	856	1483	1100
		500	1548	2121	949	1741	1240
		1000	1724	2368	1039	1954	1368
	string	100	1230	1888	953	1539	1180
		200	1431	2154	1039	1784	1314
		500	1719	2520	1175	2067	1471
		1000	1922	2805	1259	2257	1594
Sun Pascal	real	100	109	58	13	15	15
		200	127	61	14	17	17
		500	152	66	16	20	20
		1000	174	70	17	21	22
		2000	198	72	19	24	25
		5000	221	85	23	30	30
		10000	241	89	26	32	33
	string	100	126	124	88	99	107
		200	150	130	92	103	112
		500	182	145	98	112	120
		1000	202	154	103	121	129
		2000	224	161	108	130	136
		5000	256	176	115	142	146
		10000	282	189	120	153	153

Table 1. *A comparison of execution time (in microseconds)*

The results of our experiment are given in Table 1. In all cases, the standard deviation was less than 6%.

From the table we conclude that searches are always faster in a BB-tree. For insertion we get varying results depending on environment. When using Turbo Pascal, it seems to be the case that the cost of recursion used by the BB-tree is compensated by the use of more comparisons and more calls to the memory allocation system used by the DSL. The only time when the DSL is faster is during insertions in the Sun Pascal environment.

We would like to point out that in many application element location is much more common that updates. In such cases, the BB-tree would probably consume less time than the DSL in any environment.

4.2 BB-trees versus the probabilistic skip list

So far, we have been concerned with worst-case efficient data structures. However, our binary tree implementation would also compete very well with the original, probabilistic, skip list. A demonstration program for this data structure has been announced by its inventor on the international network for anyone to fetch by anonymous ftp. Examining this source code, we find that the corresponding code for BB-trees is considerably shorter. In fact, a comparison of the number of tokens indicates that the total code of Pugh's program (including declaration, initialization, insertion, deletion, destroying a list, and search) is two to three times as long as the corresponding code for a BB-tree. The skip list code is also more "tricky". Of course, our comparison is not objective and therefore we leave it to the reader to judge.

It should be noted that also the probabilistic skip list uses a reserved key value with the same drawbacks as for the deterministic skip list.

5 Comments

The balanced binary search tree is not necessarily that complicated, although it took more than 30 years from its introduction to find out.

Finally, we hope that the search for simple maintenance algorithms for binary trees, skip lists, and other alternatives will help the mission of breaking the dominance of singly-linked lists and other poor data structures.

Acknowledgements

The comments of Thomas Papadakis and Kerstin Andersson has contributed to the presentation of this material.

References

1. G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Dokladi Akademia Nauk SSSR*, 146(2):1259–1262, 1962.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983. ISBN 0-201-00023-7.
3. A. Andersson. A note on searching in a binary search tree. *Software-Practice and Experience*, 21(10):1125–1128, 1991.
4. A. Andersson, Ch. Icking, R. Klein, and Th. Ottmann. Binary search trees of almost optimal height. *Acta Informatica*, 28:165–178, 1990.
5. R. Bayer. Binary B-trees for virtual memory. In *Proc. ACM SIGIFIDET Workshop on Data Description, Access and control*, pages 219–235, 1971.

6. R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.
7. S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. The Benjamin/Cummings Publishing Company Inc., 1986. ISBN 0-8053-2162-4.
8. L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th Ann. IEEE Symp. on Foundations of Computer Science*, pages 8–21, 1978.
9. H. A. Maurer, Th. Ottmann, and H. W. Six. Implementing dictionaries using binary trees of very small height. *Information Processing Letters*, 5(1):11–14, 1976.
10. J. I. Munro, Th. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proc. Symp. of Discrete Algorithms*, pages 367–375, 1992.
11. J. Nievergelt and E. M. Reingold. Binary trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.
12. H. J. Olivie. A new class of balanced search trees: Half-balanced binary search trees. *R. A. I. R. O. Informatique Theoretique*, 16:51–71, 1982.
13. Th. Papadakis. private communication.
14. W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Proc. Workshop on Algorithms and Data Structures, WADS '89, Ottawa*, pages 437–449, 1989.
15. N. Wirth. *Algorithms and Data Structures*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986. ISBN 0-13-022005-1.