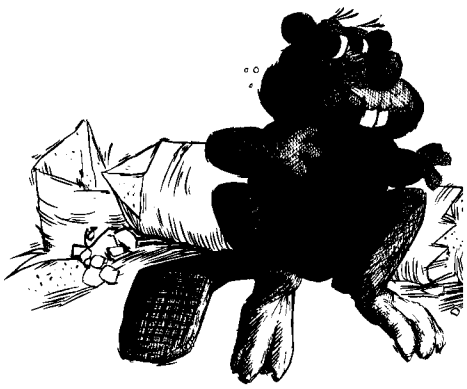
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT



A Look at Symmetric Binary B-trees

Nivio Ziviani
Frank Wm. Tompa

CS-80-51

November, 1980

A LOOK AT SYMMETRIC BINARY B-TREES

Nivio Ziviani
Frank Wm. Tompa

Department of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1
Canada

ABSTRACT

Symmetric binary B-trees have been proposed as an alternative for AVL trees and B-trees for representing dictionary information. The average costs to search, insert, and delete keys in symmetric binary B-trees are examined empirically, and the results are compared to similar experimental results for AVL trees. A generalization of the symmetric binary B-trees is also proposed. The results indicate that symmetric binary B-trees should be considered seriously when designing a representation for dictionaries.

Key phrases: Symmetric binary B-trees, 2-3-4 trees, dictionary representations, AVL trees, expected search time.

CR *Categories:* 4.34, 5.25

November 13, 1980

A LOOK AT SYMMETRIC BINARY B-TREES

Nivio Ziviani
Frank Wm. Tompa

Department of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1
Canada

1. INTRODUCTION

B-trees were introduced by Bayer and McCreight (1972) as a dictionary structure primarily for secondary store. A special case of B-trees, more appropriate for primary store, are called 2-3 trees, in which each node has either two or tree subtrees (Knuth, 1973). Such trees can be represented by binary trees as shown originally by Bayer (1971), and depicted in Figure 1. When seen as a binary B-tree, there is an inherent asymmetry in the sense that the left edges must be vertical (i.e., point to a node at the descendent level), whereas the right edges can be either vertical or horizontal. Removing the asymmetry of the binary B-trees leads to the symmetric binary B-trees, abbreviated as SBB trees (Bayer, 1972).

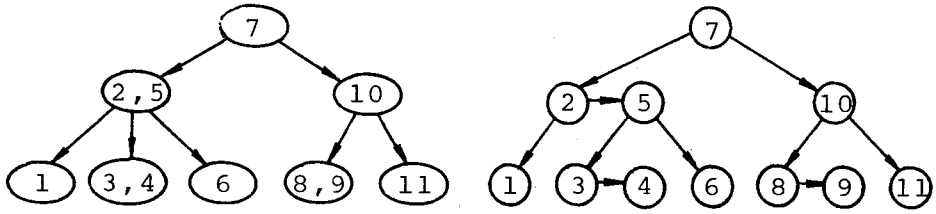


Figure 1 A 2-3 tree and the corresponding binary B-tree

Figure 2 shows a graphic representation of a SBB tree. SBB trees are binary trees with two kinds of edges, namely vertical edges and horizontal edges (called δ-edges and ρ-edges respectively, by Bayer), such that:

(i) all paths from the root to every leaf node have the same number of vertical edges, and

(ii) there are no two successive horizontal edges.

For SBB trees two kinds of heights need to be distinguished: the vertical height h, required for the uniform height constraint and calculated by counting only vertical edges plus one in any path from root to leaf, and the ordinary height k, required to determine the maximum number of key comparisons and calculated

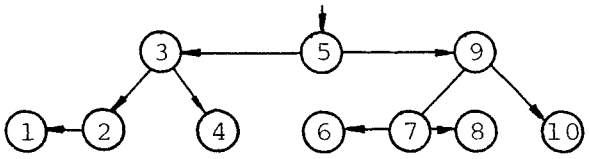


Figure 2 SBB tree of vertical height $h = 2$ and ordinary height $k = 4$

by counting all edges plus one in a maximal path from root to leaf. The ordinary height k is larger than the vertical height h whenever the tree has some horizontal edges. In particular, for a given SBB tree with n internal nodes, we have $h \leqslant k \leqslant 2h$, and $\log(n+1) \leqslant k \leqslant 2\log(n+2) - 2$ (Bayer, 1972).†

An SBB tree can also be seen as a binary representation for a 2-3-4 tree as defined by Guibas and Sedgewick (1978), in which "supernodes" may contain up to three keys and four sons. For example, such a "supernode" (with keys 3, 5, and 9 and sons containing keys 2, 4, 7, and 10) can be seen in the SBB tree of Figure 2.

SBB trees have been studied by several researchers. Bayer (1972) introduced the trees and the maintenance algorithms, and showed that the class of AVL trees is a proper subset. Wirth (1976) presented an implementation of the insertion algorithm using Pascal. Olivié (1980a, 1980b) presented a relationship between SBB trees and son-trees (Ottman and Six, 1976), and a new insertion algorithm which needs less restructurings per insertion and produces SBB trees with smaller height than the original algorithms. Ladner, Huddleston, and others (Ladner, 1980) have shown that rebalancing while building a tree of n nodes from the empty tree requires at most $O(n)$ time.

The objective of this paper is twofold:

(i) to present a performance evaluation of SBB trees. The average costs to search, insert, and delete keys in SBB trees are experimentally examined, and the results are compared to similar experimental results for AVL trees.

(ii) to present some ideas that permit a generalization of the concept of SBB trees, by permitting the number of successive horizontal edges to be greater than one, while not permitting unrestricted growth of the tree. This will be compared to similar work for AVL trees as presented by Foster (1973).

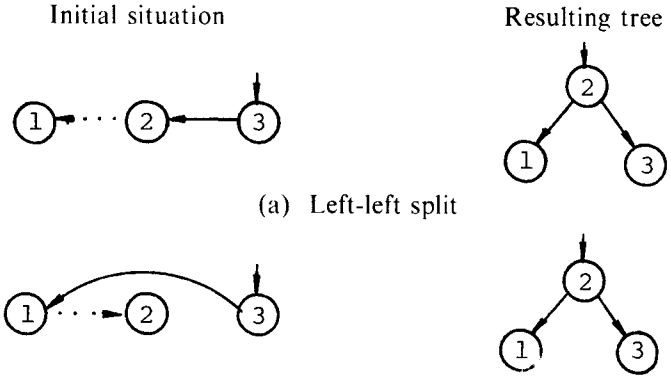
2. PERFORMANCE EVALUATION OF SYMMETRIC BINARY B-TREES

The algorithm to construct and maintain SBB trees uses local transformations on the path of insertion (deletion) to preserve the balance of the trees. The key to be inserted (deleted) is always inserted (deleted) after the lowest vertical pointer in the tree. Depending on the tree's status prior to insertion (deletion) two successive horizontal pointers may result, and a transformation may become necessary. If a transformation is performed, the number of vertical

† All logarithms are taken to base two.

pointers from the root to the new leaf may be altered, thus requiring further transformations to obtain a uniform height.

Figure 3 shows the transformations proposed by Bayer (1972). Symmetric transformations (i.e. right-right and right-left) also may occur. The results of a performance evaluation of the algorithm using these transformations can be found in Appendix B.



(a) Left-left split

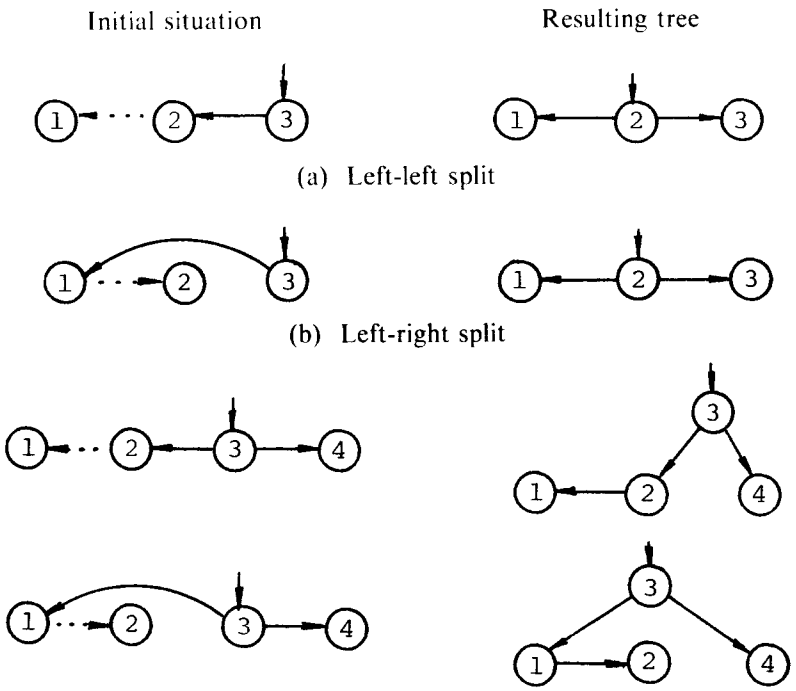
(b) Left-right split

Figure 3 The two transformations, called splits, as proposed by Bayer (1972)

A revised set of transformations has been proposed by Olivié (1980b). The insertion algorithm using the new transformations produces SBB trees with smaller height than does the original algorithm, and it needs less transformations to build the tree. Guibas and Sedgewick (1978) have also defined similar transformations, which have been adopted in the University of Washington's ESP text editor developed by Fisher, Ladner, Robertson, and Sandberg (Ladner, 1980). Figure 4 shows the new transformations. The left-left split and the left-right split require the modification of 3 and 5 pointers respectively, and the height-increase transformation requires only the modification of two bits. Symmetric transformations may also occur.

When a height-increase transformation occurs, the height of the transformed subtree is one more than the height of the original subtree, and thus the node rearrangement may cause other transformations along the search path up to the root. Usually the retreat along the search path terminates when either a vertical pointer is found or a split transformation is performed. As the height of the split subtree is the same as the height of the original subtree, at most one split transformation per insertion may be performed.

Both Bayer (1972) and Olivié (1980b) used two bits per node in their algorithms to indicate whether the right and left pointers are vertical or horizontal pointers. The University of Washington's text editor, however, uses only one bit per node: the information whether the left (right) pointer of a node is vertical or horizontal is stored in the left (right) son. Besides the fact that this requires less space at each node, the retreat along the search path to check for two successive horizontal pointers can be terminated earlier than when using the two bits algorithms, because the information about the type of pointer that leads to a node



(a) Left-left split

(b) Left-right split

(c) Height-increase

Figure 4 The new transformations for insertion into SBB trees as proposed by Olivié (1980b)

is available without the need to retreat to its father. The results in the next section were obtained using the set of transformations of Figure 4 and one bit per node.

2.1. Experimental Results

The performance evaluation of SBB trees was obtained by means of a sufficient number of repetitions (for different tree sizes) of the following experiment: a permutation of an ordered list is presented one key at a time to the procedure that inserts keys into an initially empty tree; this is followed by the presentation of another permutation to the procedure that deletes keys from the tree just constructed. In other words, there are n insertions followed by n deletions. Consequently, all observations of inserting (deleting) the i^{th} ($i \leqslant n$) node into a tree are independent events, which eliminates correlation in the simulation results. A similar type of design was used by Karlton et al. (1976) in an experimental study of AVL trees. In order to generate random permutations of an ordered list we used the algorithm presented by Durstenfeld (1964). After insertion of the n^{th} node into the tree the following values are tabulated:

(i) the average number of comparisons in an unsuccessful search (C_n');

(ii) the average number of comparisons in a successful search (C_n); and

(iii) the length of the longest path (i.e., the ordinary height).

Table I presents results for trees of various sizes. The unsuccessful search time C_n' is proportional to $\log n$. In fact, from Table I(a) it can be seen that

$$C_n' = \alpha \log n + \beta.$$

The values for α and β can be derived from the data in that table. The graph of Figure 5 shows the values of $\log n$ for trees of size 10, 50, 100, 500, 1000, 5000, and 10000 nodes represented along the x-axis, and the difference $(C_n' - \log n) \pm$ 95% confidence interval is represented along the y-axis. The values obtained for trees of size greater than approximately 50 nodes can be shown to be asymptotically independent of the number of nodes in the tree. (The number of sample trees for each tree size was chosen in such a way that the variance is approximately the same for all tree sizes.)

In order to obtain the bounds for α and β we considered the points in the graph corresponding to the trees of sizes 500, 1000, 5000, and 10000 nodes and performed a linear regression. Thus, the approximate value for C_n', to within the precision of the simulation, is

$$C_n' = (1.0186 \pm 0.0010) \log n + 0.0912 \pm 0.0114.$$

In addition, the following values are tabulated in order to estimate the cost of maintaining the properties of SBB trees:

on insertion:

(i) the percentage of insertions that caused a transformation to be performed (all four types of splits are considered separately), and

(ii) the number of nodes revisited to restore the tree property, counted from the father of the node inserted into the tree to the node at which the retreat terminated.

on deletion:

(iii) the percentage of deletions that caused a transformation to be performed (all four types of splits are considered separately), and

(iv) the number of nodes revisited to restore the tree property, counted from the node to be deleted from the tree to the node at which the retreat terminated. Sometimes the retreat terminates immediately at the node to be deleted; this is not counted as a retreat. For instance, if the node to be deleted is pointed at by a horizontal pointer, the only operation to perform is to replace that pointer by nil. (If the node to be deleted has two subtrees, it is first interchanged with the rightmost node of its left subtree before deletion.)

Table II presents the insertion and deletion results. These results are actually for trees of 10000 nodes only because they have shown to be asymptotically independent of the number of nodes in the tree. (In fact, for trees greater than approximately 50 nodes, the results approach these.)

n	C_n'	Variance
5	2.6667±0.0003	0.0000
10	3.5509±0.0032	0.0005
50	5.8549±0.0051	0.0030
100	6.8621±0.0053	0.0022
500	9.2234±0.0059	0.0014
1000	10.2435±0.0063	0.0010
5000	12.6056±0.0073	0.0010
10000	13.6273±0.0084	0.0009

(a) Expected unsuccessful search

n	C_n	Variance
5	2.2000±0.0003	0.0000
10	2.9057±0.0035	0.0005
50	4.9720±0.0051	0.0031
100	5.9307±0.0054	0.0023
500	8.2419±0.0059	0.0014
1000	9.2537±0.0062	0.0010
5000	11.6081±0.0073	0.0010
10000	12.6287±0.0083	0.0009

(b) Expected successful search

n	Longest Path	Variance
5	3.0000±0.0198	0.
10	4.0229±0.0222	0.0225
50	7.0089±0.0163	0.0311
100	8.0933±0.0330	0.0849
500	11.0267±0.0259	0.0261
1000	12.1400±0.0684	0.1216
5000	15.0143±0.0280	0.0143
10000	16.1800±0.1076	0.1506

(c) Expected worst case search

Table I SBB tree statistics (expected number of comparisons)

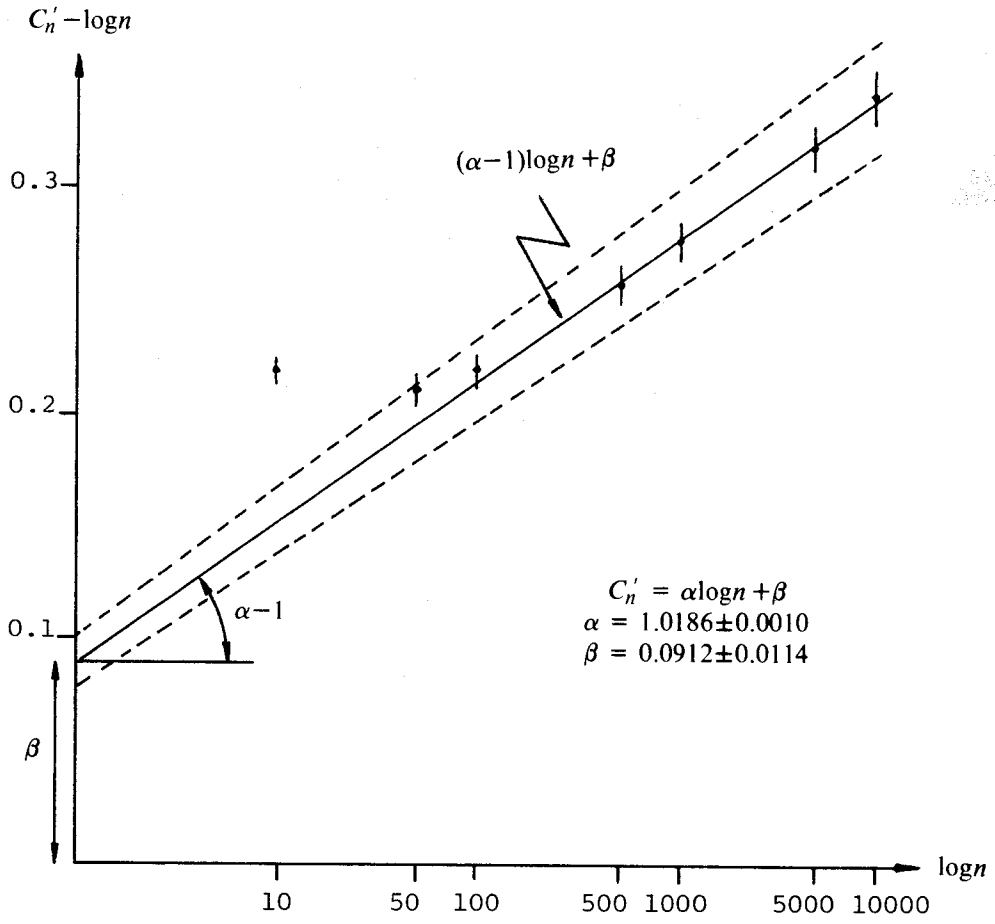


Figure 5 Representation of the C_n' for SBB trees of sizes 10, 50, 100, 500, 1000, 5000 and 10000 nodes. The straight line corresponds to a linear regression of the points corresponding to trees of 500, 1000, 5000, and 10000 nodes and the dashed lines correspond to choosing α and β at the extremes of their 95% confidence intervals

	Mean	Variance
Insertion:		
LL split	0.0967±0.0007	0.0000
LR split	0.0976±0.0008	0.0000
RR split	0.0970±0.0007	0.0000
RL split	0.0968±0.0009	0.0000
Total splits	0.3880±0.0011	0.0000
Height increase	0.5119±0.0007	0.0000
Nodes revisited	2.4109±0.0019	0.0001
Deletion:		
LL split	0.0585±0.0005	0.0000
LR split	0.0460±0.0005	0.0000
RR split	0.0585±0.0008	0.0000
RL split	0.0461±0.0006	0.0000
Total splits	0.2091±0.0011	0.0000
Height increase	0.0018±0.0001	0.0000
Ptr rearrangement	0.0583±0.0007	0.0000
Nodes revisited	0.8596±0.0014	0.0000

Table II Insertion and deletion statistics for SBB trees of 10000 nodes

2.2. Symmetric binary B-trees versus AVL trees

A binary search tree is AVL if the height of the left subtree and the height of the right subtree of any node in the tree differs by at most one (Adel'son-Vel'skii and Landis, 1962; Knuth, 1973). In Bayer (1972) it was shown that (i) the class of AVL trees is a proper subclass of SBB trees, and (ii) the upper bound on the number of comparisons in an unsuccessful search is $2\log(n+2) - 2$ for SBB trees, whereas it is well-known that the same upper bound for AVL trees is $1.44\log(n+2) - 0.328$. These two facts about SBB and AVL trees led Bayer (1972, p. 296-297), and also Wirth (1976, p.264), to conjecture that the number of comparisons for an unsuccessful search in an SBB tree is, on the average, larger than in the AVL case, but less work is required to maintain the SBB tree properties.

In order to compare the experimental results with similar results for AVL trees, the experiments done for SBB trees were repeated for AVL trees. The results for AVL trees coincide with the results of Karlton et al. (1976). The number of comparisons needed to insert the n^{th} item into an AVL tree is approximately $\alpha' \log n + \beta'$ † for large n , where $\alpha' = 1.0176 \pm 0.0022$, and $\beta' = 0.0513 \pm 0.0247$; recall that the same measure for SBB trees is approximately $\alpha \log n + \beta$ for large n, where $\alpha = 1.0186 \pm 0.0010$, and $\beta = 0.0912 \pm 0.0114$.

As expected, less work is required to maintain the SBB tree property than the AVL one. The number of splits per insertion is approximately 0.39 (against approximately 0.47 for AVL trees), and the number of nodes revisited per insertion is approximately 2.4 (against 2.8 for AVL trees). The height-increase transformation for SBB trees is equivalent to the modification of the balance field in the AVL case, and so this cost does not affect the comparison between them.

The number of splits per deletion is approximately 0.21 (against approximately 0.21 for AVL trees), and the number of nodes revisited per deletion is approximately 0.9 (against approximately 1.9 for AVL trees). Thus the number of transformations per deletion for AVL and SBB trees are roughly the same, and the number of nodes revisited per deletion is less for SBB trees.

3. GENERALIZATION OF SYMMETRIC BINARY B-TREES

Foster (1973) generalized the concept of AVL trees by allowing left and right subtrees to differ in height up to some constant $d \geqslant 1$. SBB trees can be similarly generalized as follows:

Let Δ be the maximum number of successive horizontal edges permitted in an SBB tree (Δ represents the quantity of imbalance permitted in a tree). An SBB[Δ] tree is any SBB tree that may contain up to Δ successive horizontal edges oriented in the same direction.

† The expected number of comparisons needed to insert the n^{th} item into an AVL tree has been reported to be approximately $\log n + 0.25$ for large n (Knuth, 1973, p.460). As a matter of fact in our experiments we found the coefficient of $\log n$ to be different from one. The appendix B shows the values for C_n' from which the above result was obtained in the same way the C_n' for SBB trees were obtained in the previous section.

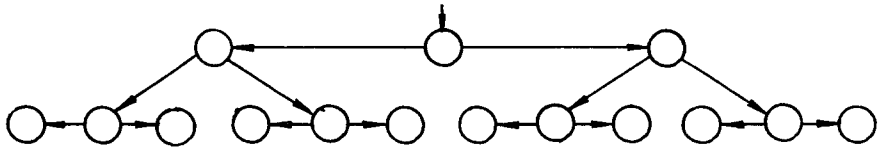
which gives

$$N_m(\Delta, h) = (\Delta + 1)(2^h - 1), \quad \text{for} \quad h > 0. \tag{4}$$

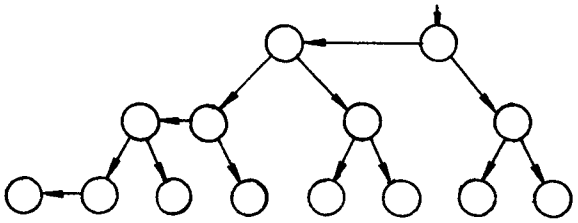
Again the proof proceeds by induction on h. (4) satisfies (3) for the basis step, $h = 0$. If it satisfies (3) for $h = i$, where $i \geqslant 0$, then

$$\begin{aligned} N_m(\Delta, i+1) &= (\Delta + 1)(2^i - 1) + (\Delta + 1)2^{i+1-1} \\ &= (\Delta + 1)2^{i+1} - (\Delta + 1) \\ &= (\Delta + 1)(2^{i+1} - 1) \end{aligned}$$

and thus satisfies (3) for $h = i+1$. Thus $N_m(\Delta,h) = (\Delta + 1)(2^h - 1)$ satisfies (3). Figure 9(b) shows a mintree of $N_m(1, 3) = 14$ nodes.



(a) SBB[2] maxtree of $h = 2$, $k = 4$, and $N_M(1, 2) = 15$ nodes



(b) SBB[1] mintree of $h = 3, k = 6$, and $N_m(1, 3) = 14$

Figure 9 Example of a maxtree and a mintree

Consequently, the upper and lower bounds for the ordinary height $k = (\Delta + 1)h$ of an SBB(Δ) tree of $N(\Delta,h)$ nodes is obtained taking logarithms in (2) and (4), as follows:

$$\frac{\Delta + 1}{\log(2\Delta + 2)} \log(N(\Delta,h) + 1) \leqslant k \leqslant (\Delta + 1)\log\left\lfloor \frac{N(\Delta,h)}{\Delta + 1} + 1 \right\rfloor.$$

3.2. Experimental results

Experiments similar to those reported in Section 2.1 can be done for SBB[Δ] trees using various values of Δ. In fact, using transformations that extend Bayer's original algorithms, experiments were carried on trees with Δ equal to 1, 2, 3, 4, and 5. Comparing the results to those for generalized AVL trees (Foster, 1973), it was discovered that SBB[Δ] trees do not perform as well on the average. For example:

- the average number of comparisons in a successful search (C_n) of an AVL[Δ] tree of 1000 nodes increases from 9.29 for Δ = 1 to 10.37 for Δ = 5. The same measures for SBB[Δ] trees increase from 9.38 for Δ = 1 to 10.78 for Δ = 5.
- in AVL[Δ] trees the average number of transformations per insertion drops from 0.47 for Δ = 1 to 0.045 for Δ = 5. The same measures for SBB[Δ] trees drop from 0.65 for Δ = 1 to 0.21 for Δ = 5.
- the average number of nodes revisited in updating an AVL[Δ] tree increases from 2.8 for Δ = 1 to 3.3 for Δ = 5. The same measures for SBB[Δ] trees increase from 3.3 for Δ = 1 to 4.6 for Δ = 5.

It must be remembered, however, that the performance of SBB trees depends very much on the types of transformations employed. Thus the use of transformations obtained by extending Olivié's algorithms, as described in Section 2, may improve the performance of SBB[Δ] trees sufficiently to compare favourably with AVL[Δ] trees. Such experiments remain to be done.

1. CONCLUSIONS

Symmetric binary B-trees have been shown to be reasonable structures for representing dictionary information. Among balanced trees, experimental results show that, on the average, SBB trees perform approximately as well as AVL trees and can be generalized analagously to allow various degrees of imbalance. The experimental results support the conjecture that SBB trees require less work than AVL trees to maintain balance, but this is at the expense of search time. In fact, the search time is only slightly longer and the maintenance time is in some areas significantly less. Thus as a practical structure, SBB trees should be considered as an option for representing dictionaries.

As for other balanced tree structures, what is most needed for SBB trees is an analytical performance study. Other researchers have begun the study of the expected behaviour for 2-3 trees (Yao, 1978) and for AVL trees (Brown, 1979; Mehlhorn, 1979), and the approaches used can be applied in a similar analysis of SBB trees, as started by Olivié (1980b). However, more research is required for a better understanding of all these structures. It is hoped that a deeper understanding of SBB trees will, in turn, yield insights into the behaviour of other balanced tree structures.

Acknowledgements

We wish to acknowledge the many fruitful discussions with Gaston Gonnet and with J. Howard Johnson, who helped particularly with the presentation of the experimental results. Financial support from the Brazilian Coordenação do Aperfeiçoamento de Pessoal de Nivel Superior (CAPES), the Universidade Federal de Minas Gerais, and the Canadian Natural Sciences and Engineering Research Council (NSERC) under grant A-9292 are also gratefully acknowledged.

REFERENCES

Adel'son-Vel'skii, G.M. and Landis, E.M. "An algorithm for the organization of information", *Doklady Akademia Nauk USSR 146*, 2 (1962), 263-266. English translation in *Soviet Math. Doklay 3* (1962), 1259-1263.

Bayer, R. "Binary B-trees for Virtual Memory", *Proc. 1971 ACM SIGFIDET Workshop*, San Diego (1971), 219-235.

Bayer, R. "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms", *Acta Informatica 1*, 4 (1972), 290-306.

Bayer, R. and McCreight, E. "Organization and Maintenance of Large Ordered Indexes", *Acta Informatica 1*, 3 (1972), 173-189.

Brown, M. "A Partial Analysis of Random Height-Balanced Trees", *SIAM Journal of Computing 8*, 1 (Feb. 1979), 33-41.

Durstenfeld, R. "Random Permutation", *Algorithm 235, CACM 7*, 7 (July 1964), 420.

Foster, C.C. "A Generalization of AVL Trees", *CACM 16*, 8 (1973), 513-517.

Guibas, L.J. and Sedgewick, R. "A Dichromatic Framework for Balanced Trees", *19th Annual Symposium on Foundations of Computer Science*, 1978.

Karlton, P.L., Fuller, S.H., Scroggs, R.E. and Kaehler, E.B. "Performance of Height-Balanced Trees", *CACM 19*, 1 (1976), 23-28.

Knuth, D.E. *The Art of Computer Programming*, Vol. 3 (Reading, Mass. : Addison-Wesley, 1973).

Ladner, R.E. Private communication, 1980.

Mehlhorn, K. "A Partial Analysis of Height-Balanced Trees under Random Insertions and Deletions", Report A 79/21, Universität des Saarlandes, West Germany, October 1979.

Olivié, H. "On the Relationship Between Son-Trees and Symmetric Binary B-trees", *Information Processing Letters 10*, 1 (February 1980a), 4-8.

Olivié, H. "Symmetric Binary B-trees Revisited", *Technical Report* 80-01, Interstedelijke Industriële Hogeschool Antwerpen-Mechelen, Antwerp, Belgium, (1980b).

Ottmann, Th. and Six, H.-W. "Eine neue Klasse von ausgeglichenen Binärbäumen", *Augewandte Informatik 9* (1976), 395-400.

Wirth N. *Algorithms + Data Structures = Programs* (New Jersey: Prentice-Hall, 1976).

Yao, A. "On Random 2-3 Trees", *Acta Informatica 9* (1978), 159-170.

APPENDICES

A. Experimental results for the original transformations

The results of a performance evaluation of the SBB insertion and deletion algorithms using the original transformations (Bayer, 1972) presented in Figure 3 can be found in Table A.I and Table A.II.

The performance of the algorithm using these transformations is clearly worse than the new algorithm using the transformations presented in Figure 4. The expected number of comparisons for an unsuccessful search in an SBB tree with 5000 nodes using the original algorithm is approximately 12.8, and approximately 12.6 using the improved algorithm.

A surprising result has occurred with the work required to maintain the SBB tree property under insertion. The number of splits for insertion is approximately 0.65 (against approximately 0.47 for AVL trees and approximately 0.39 for the new SBB algorithm), and the number of nodes revisited per insertion is approximately 3.3 (against 2.8 for AVL trees and 2.4 for the new SBB algorithm). In the absence of the discovery of the improved transformation algorithms, these results would seem to contradict the conjecture mentioned in Section 2.2.

The number of splits per deletion is approximately 0.22 (against approximately 0.21 for AVL trees and approximately 0.21 for the new SBB algorithm), and the number of nodes revisited per deletion is approximately 1.6 (against approximately 1.9 for AVL trees and approximately 0.86 for the new SBB algorithm).

n	C_n'	Variance
5	2.6667±0.0003	0.0000
10	3.5579±0.0047	0.0010
50	5.8823±0.0084	0.0046
100	6.9325±0.0106	0.0058
500	9.3380±0.0125	0.0065
1000	10.3634±0.0135	0.0047
5000	12.8179±0.0163	0.0062
10000	13.8600±0.0437	0.0065

(a) Expected unsuccessful search

n	C_n	Variance
5	2.2000±0.0003	0.0000
10	2.9137±0.0051	0.0012
50	4.9999±0.0086	0.0048
100	6.0018±0.0107	0.0059
500	8.3567±0.0125	0.0065
1000	9.3738±0.0135	0.0047
5000	11.8205±0.0163	0.0062
10000	12.8614±0.0437	0.0065

(b) Expected successful search

n	Longest Path	Variance
5	3.0000±0.0198	0.
10	4.1371±0.0510	0.1183
50	7.3000±0.0568	0.2100
100	8.7700±0.0646	0.2171
500	12.0125±0.0714	0.2123
1000	13.4000±0.1037	0.2800
5000	16.9556±0.1063	0.2647
10000	18.4615±0.2710	0.2485

(c) Expected worst case search

Table A.I SBB tree statistics (expected number of comparisons)

	Mean	Variance
Insertion:		
LL split	0.1636±0.0010	0.0000
LR split	0.1637±0.0009	0.0000
RR split	0.1633±0.0009	0.0000
RL split	0.1636±0.0009	0.0000
Total splits	0.6542±0.0009	0.0000
Nodes revisited	3.3022±0.0019	0.0001
Deletion:		
LL split	0.0519±0.0005	0.0000
LR split	0.0594±0.0007	0.0000
RR split	0.0526±0.0006	0.0000
RL split	0.0595±0.0007	0.0000
Total splits	0.2233±0.0011	0.0000
Nodes revisited	1.5984±0.0010	0.0000

Table A.II Insertion and deletion statistics for trees of 5000 nodes

B. Experimental results for AVL trees

The results of a performance evaluation of the AVL insertion and deletion algorithms are presented in Table B.I and Table B.II.

n	C_n'	Variance
5	2.6667±0.0003	0.0000
10	3.5507±0.0030	0.0005
50	5.8161±0.0044	0.0018
100	6.8228±0.0049	0.0013
500	9.1737±0.0054	0.0011
1000	10.1923±0.0056	0.0010
5000	12.5593±0.0067	0.0010
10000	13.5693±0.0079	0.0008

(a) Expected unsuccessful search

n	C_n	Variance
5	2.2000±0.0003	0.0000
10	2.9060±0.0033	0.0006
50	4.9324±0.0045	0.0018
100	5.8911±0.0049	0.0013
500	8.1920±0.0054	0.0011
1000	9.2025±0.0056	0.0010
5000	11.5618±0.0067	0.0010
10000	12.5706±0.0079	0.0008

(b) Expected successful search

n	Longest Path	Variance
5	3.0000±0.0198	0.
10	4.0000±0.0149	0.
50	6.9514±0.0226	0.0463
100	8.0000±0.0149	0.
500	10.9333±0.0401	0.0626
1000	12.0000±0.0247	0.
5000	14.8941±0.0658	0.0958
10000	16.0000±0.0582	0.

(c) Expected worst case search

Table B.I AVL tree statistics (expected number of comparisons)

	Mean	Variance
Insertion:		
Total rotations	0.4657±0.0015	0.0000
Nodes revisited	2.7816±0.0015	0.0000
Deletion:		
Total rotations	0.2142±0.0010	0.0000
Nodes revisited	1.9152±0.0012	0.0000

Table B.II Insertion and deletion satistics for AVL trees of 10000 nodes

C. Implementation of insertion and deletion algorithms for SBB trees

The SBB manipulation algorithms have been implemented in Pascal. A node has four fields, as follows:

KEY:	the key stored in the node
LEFT:	pointer to the left son
RIGHT:	pointer to the right son
BIT:	a Boolean variable to indicate whether the edge that points to the node is vertical or horizontal.

The parameters X and P for INSERT (DELETE) contain the key to be inserted into (deleted from) the tree and the pointer to the root of the subtree in which the insertion (deletion) must be performed. Whenever a node is raised to the next level during insertion, the parameter H for INSERT is set to the value HORIZONTAL to indicate that the pointer to that node has been made horizontal.

The five procedures LLSPLIT, LRSPLIT, RRSPLIT, RLSPLIT, and HTINCREASE are used in both INSERT and DELETE procedures in order to remove two successive horizontal pointers. The following five procedures are used locally in DELETE:

- LEFT_SHORT (RIGHT_SHORT) is called when a left (right) leaf node that is referenced by a vertical pointer is deleted. (The subtree is short in height after deletion.)
- LEFT_PTR_REARR(RIGHT_PTR_REARR) is used to rearrange the nodes near the one just deleted from the tree and is called from the procedure LEFT_SHORT(RIGHT_SHORT).
- If the node to be deleted has two subtrees the procedure DEL is called in order to interchange it with the rightmost node of its left subtree before deleting it.

The insertion and deletion procedures have been written as two procedures each. The outer procedures INSERT and DELETE are non-recursive and allow the shortcutting of the full recursion of the inner recursive procedures IINSERT and IDELETE.

```
{ Tree search, insertion and deletion in a SBB tree }

type inclination = (vertical,horizontal);
     ref = ↑node;
     node = record
              key: integer;
              left, right: ref;
              bit: inclination;
            end;

procedure llsplit (var p: ref; trans_type: inclination);
var p1: ref;
begin { trans_type=vertical => transf. increase height }
   p1 := p↑.left;   p↑.left := p1↑.right;   p1↑.right := p;
   p1↑.bit := p↑.bit;   p↑.bit := trans_type;
```

```
  if trans_type = vertical then p1↑.left↑.bit := vertical;
  p := p1
end;

procedure lrsplit (var p: ref; trans_type: inclination);
var p1, p2: ref;
begin { trans_type = vertical => transf. increase height }
  p1 := p↑.left;   p2 := p1↑.right;
  p2↑.bit := p↑.bit;   p↑.bit := trans_type;
  if trans_type = vertical then p1↑.bit := vertical;
  p1↑.right := p2↑.left;   p2↑.left := p1;
  p↑.left := p2↑.right;   p2↑.right := p;   p := p2
end;

procedure rrsplit (var p: ref; trans_type: inclination);
var p1: ref;
begin { trans_type = vertical => transf. increase height }
  p1 := p↑.right;   p↑.right := p1↑.left;   p1↑.left := p;
  p1↑.bit := p↑.bit;   p↑.bit := trans_type;
  if trans_type = vertical then p1↑.right↑.bit := vertical;
  p := p1
end;

procedure rlsplit (var p: ref; trans_type: inclination);
var p1, p2: ref;
begin { trans_type = vertical => transf. increase height }
  p1 := p↑.right;   p2 := p1↑.left;
  p2↑.bit := p↑.bit;   p↑.bit := trans_type;
  if trans_type = vertical then p1↑.bit := vertical;
  p1↑.left := p2↑.right;   p2↑.right := p1;
  p↑.right := p2↑.left;   p2↑.left := p;   p := p2
end;

procedure htincrease (var p: ref);
begin p↑.right↑.bit := vertical;
    p↑.left↑.bit := vertical
end;

procedure insert (x: integer; var p: ref; var h: inclination);
label 999;   { Shortcut full recursion of iinsert }

  procedure iinsert (x: integer; var p: ref; var h: inclination);
  var b: inclination;
  begin
    if p = nil
    then begin   { key is not in the tree;   insert it }
           new(p);   h := horizontal;
           p↑.key := x;   p↑.bit := horizontal;
           p↑.left := nil;   p↑.right := nil;
      end
    else
    if x < p↑.key
    then
    begin
      iinsert(x, p↑.left, h);
      if h = horizontal
      then if p↑.bit = horizontal then h := vertical else goto 999
      else
      begin b := vertical;
          if p↑.right ≠ nil
          then if p↑.right↑.bit = horizontal then b := horizontal;
          if b = horizontal
          then begin htincrease(p);   p↑.bit := horizontal;   h := horizontal end
          else if p↑.left↑.left ≠ nil
```

```
                then if p↑.left↑.left↑.bit = horizontal
                    then begin llsplit(p,horizontal);  goto 999 end
                    else begin lrsplit(p,horizontal);  goto 999 end
                else begin lrsplit(p,horizontal);  goto 999 end
          end
      end
      else
      if x > p↑.key
      then
      begin
          iinsert(x, p↑.right, h);
          if h = horizontal
          then if p↑.bit = horizontal then h := vertical else goto 999
          else
          begin b := vertical;
              if p↑.left ≠ nil
              then if p↑.left↑.bit = horizontal then b := horizontal;
              if b = horizontal
              then begin htincrease(p);  p↑.bit := horizontal;  h := horizontal end
              else if p↑.right↑.right ≠ nil
                  then if p↑.right↑.right↑.bit = horizontal
                      then begin rrsplit(p,horizontal);  goto 999 end
                      else begin rlsplit(p,horizontal);  goto 999 end
                  else begin rlsplit(p,horizontal);  goto 999 end
          end
      end
      else begin writeln(' Key ', x, ' is already in the tree');  goto 999 end
  end; {iinsert}

begin {insert}
  iinsert(x, p, h);
  999:  { Shortcut full recursion of iinsert }
end; {insert}

procedure delete (var x: integer; var p: ref);
label 1001;      { Shortcut full recursion of idelete }

  procedure left_ptr_rearr (var p: ref);
  var p1: ref;
  begin p1 := p↑.right;  p↑.right := p1↑.left↑.left;
      p1↑.left↑.left := p;  p↑.bit := horizontal;
      p1↑.bit := vertical;  p := p1
  end;

  procedure right_ptr_rearr (var p: ref);
  var p1: ref;
  begin p1 := p↑.left;  p↑.left := p1↑.right↑.right;
      p1↑.right↑.right := p;  p↑.bit := horizontal;
      p1↑.bit := vertical;  p := p1
  end;

  procedure idelete (var x: integer; var p: ref);
  var q: ref;

    procedure left_short (var p: ref);
    begin { The left leaf was deleted and the tree is left short in height }
      if p↑.left ≠ nil
      then if p↑.left↑.bit = horizontal
          then begin p↑.left↑.bit := vertical;  goto 1001 end;
      if p↑.right↑.bit = horizontal
      then
      begin left_ptr_rearr(p);
          if p↑.left↑.left↑.right ≠ nil
          then
```

```
      begin if p↑.left↑.left↑.right↑.bit = horizontal
          then begin if p↑.left↑.right ≠ nil
                  then if p↑.left↑.right↑.bit = horizontal
                      then
                      begin htincrease(p↑.left);
                          p↑.left↑.bit := horizontal;
                          goto 1001
                      end;
                  lrsplit(p↑.left,horizontal);
                  goto 1001
              end
          else goto 1001
      end
      else goto 1001
  end
  else
  begin p↑.right↑.bit := horizontal;
      if p↑.right↑.right ≠ nil
      then
      begin if p↑.right↑.right↑.bit = horizontal
          then begin rrsplit(p,vertical); goto 1001 end
          else if p↑.right↑.left↑.bit = horizontal
              then begin rlsplit(p,vertical); goto 1001 end
              else if p↑.bit = horizontal
                  then begin p↑.bit := vertical; goto 1001 end
      end
      else if p↑.right↑.left ≠ nil
          then begin rlsplit(p,vertical); goto 1001 end
  end
end; {left_short}

procedure right_short (var p: ref);
begin { The right leaf was deleted and the tree is right short in height }
  if p↑.right ≠ nil
  then if p↑.right↑.bit = horizontal
      then begin p↑.right↑.bit := vertical; goto 1001 end;
  if p↑.left↑.bit = horizontal
  then
  begin right_ptr_rearr(p);
      if p↑.right↑.right↑.left ≠ nil
      then
      begin if p↑.right↑.right↑.left↑.bit = horizontal
          then begin if p↑.right↑.left ≠ nil
                  then if p↑.right↑.left↑.bit = horizontal
                      then
                      begin htincrease(p↑.right);
                          p↑.right↑.bit := horizontal;
                          goto 1001
                      end;
                  rlsplit(p↑.right,horizontal);
                  goto 1001
              end
          else goto 1001
      end
      else goto 1001
  end
  else
  begin p↑.left↑.bit := horizontal;
      if p↑.left↑.left ≠ nil
      then
      begin if p↑.left↑.left↑.bit = horizontal
          then begin llsplit(p,vertical); goto 1001 end
          else if p↑.left↑.right↑.bit = horizontal
              then begin lrsplit(p,vertical); goto 1001 end
```

```
                else if p↑.bit = horizontal
                    then begin p↑.bit := vertical;  goto 1001 end
            end
            else if p↑.left↑.right ≠ nil
                then begin lrsplit(p,vertical);  goto 1001 end
        end
    end;  {right_short}

    procedure del (var r: ref);
     begin  { The node to be deleted is interchanged with rightmost node of left subtree }
       if r↑.right ≠ nil
       then begin del(r↑.right);  right_short( r ) end
       else
       begin
         q↑.key := r↑.key;   q := r;   r := r↑.left;
         if q↑.bit = horizontal
         then begin dispose(q);  goto 1001 end
         else begin dispose(q);
                  if r ≠ nil
                  then begin r↑.bit := vertical;  goto 1001 end
              end
       end
    end;

  begin  {idelete}
    if p = nil
    then begin writeln(' Key ', x, ' is not in the tree');  goto 1001 end
    else
    if x < p↑.key
    then begin idelete(x, p↑.left);  left_short( p ) end
    else
    if x > p↑.key
    then begin idelete(x, p↑.right);  right_short( p ) end
    else
    begin
      q := p;
      if q↑.right = nil
      then begin p := q↑.left;
                 if q↑.bit = horizontal
                 then begin dispose(q);  goto 1001 end
                 else begin dispose(q);
                            if p ≠ nil
                            then begin p↑.bit := vertical;  goto 1001 end
                      end
           end
      else
      if q↑.left = nil
      then begin p := q↑.right;
                 dispose(q);
                 if p ≠ nil
                 then begin p↑.bit := vertical;  goto 1001 end
           end
      else begin del(q↑.left);  left_short( p ) end
    end
  end;  {idelete}

begin  {delete}
  idelete(x, p);
  1001:   { Shortcut full recursion of idelete }
end;
```