Binary Search Trees of Almost Optimal Height

Arne Andersson*
Department of Computer Science
Lund University
Box 118
S-221 00 Lund
Sweden

Christian Icking Rolf Klein
Universität- Gesamthochschule Essen
FB 6 / Praktische softwareorientierte Informatik
Schützenbahn 70
4300 Essen 1
Federal Republic of Germany

Thomas Ottmann†
Institut für Informatik
Universität Freiburg
Rheinst. 10-12
D-7800 Freiburg
Federal Republic of Germany

*This work was partially supported by a grant from the National Swedish Board for Technical Development.

†This work was partially supported by a grant from the Deutsche Forschungsgemeinschaft, grant no. Ot 64/5-2.

Abstract

First we present a generalization of symmetric binary B-trees, SBB(k)-trees. The obtained structure has a height of only $\lceil (1+\frac{1}{k})\log(n+1)\rceil$ [1], where k may be chosen to be any positive integer. The maintenance algorithms require only a constant number of rotations per updating operation in the worst case. These properties together with the fact that the structure is relatively simple to implement makes it a useful alternative to other search trees in practical applications.

Then, by using an SBB(k)-tree with a varying k we achieve a structure with a logarithmic amortized cost per update and a height of $\log n + o(\log n)$. This result is an improvement of the upper bound on the height of a dynamic binary search tree. By maintaining two trees simultaneously the amortized cost is transformed into a worst-case cost. Thus, we have improved the worst-case complexity of the dictionary problem.

1 Introduction

The *binary search tree* is a simple data structure well suited for storing an ordered set of elements. The original binary search tree allows insertion or search of an element in $\Theta(\log n)$ average-case time [2, 7]. However, each operation mentioned requires linear time in the worst case. This bad behaviour is due to the fact that the height of the tree is not controlled by the updating algorithms and therefore a skew tree may occur.

To improve the worst-case behaviour of the binary search tree different types of *balanced search trees* have been presented [1, 3, 11, 13, 15]. Those trees have a height of $\Theta(\log n)$ and efficient maintenance algorithms which allow insertion, deletion and search to be performed in $\Theta(\log n)$ worst-case time.

One way to obtain a balanced binary search tree is by using binary representations of B-trees. The B-tree introduced by Bayer and McCreight [4] is often referred to as an a-b-tree where a and b are two integers, $a < b$. The data structure is a multiway tree where each node except the root has a degree between a and b. All leaves in a B-tree have the same depth. The *symmetric binary B-tree*, or *SBB-tree*, introduced by Bayer [3] is a binary representation of a 2-4-tree where each node in the B-tree is represented by a small binary tree. The structure has several properties that makes it a good

[1] log denotes the logarithm to the base 2 and n denotes the number of elements stored in the tree.

alternative to other binary search trees. It can be represented with only one extra bit per stored element for balance information, it has relatively simple updating algorithms and it can be maintained with only a constant number of restructurings per update. The last property is crucial in some applications where restructuring is expensive, such as when implementing priority search trees [12]. However, the maximal height of an SBB-tree is $2 \log n$. Thus, in applications where fast search is essential the symmetric binary B-tree is outperformed by the AVL-tree which has a maximal height of $1.44 \log n$ [1], and the k-neighbour tree with a maximal height of $(1+\epsilon) \log n$, where ϵ is an arbitrary small constant [11].

A question which arises from studying SBB-trees is whether there are any useful binary representations of B-trees that are not 2-4-trees. Bayer [3] defined a generalized symmetric binary B-tree but so far this generalization has not shown any particular advantage that makes it useful. In this paper we present a new generalization of the SBB-tree that represents a-b-trees where $a = 2^k$ and $b = 2^{k+1}$ for any positive integer k. This generalized structure, denoted an *SBB(k)-tree* (in [9] it is referred to as a *red-h-black tree*), inherits the good properties of the SBB-tree with respect to space efficiency, need of only a constant number of restructurings and simplicity of the algorithms. Moreover, it has the advantage of being of low height, $\left\lceil (1 + \frac{1}{k}) \log(n+1) \right\rceil$. These properties make SBB(k)-trees a good alternative to other search trees for practical applications.

The fact that our generalized structure requires few restructurings per update also gives us the possibility to present an improved upper bound on the height of a dynamic binary search tree. This is obtained by using a modified version of an SBB(k)-tree where we use a varying value of k. We obtain a binary search tree with a logarithmic cost for updates and a height of $\log n + o(\log n)$, which is better than for any other dynamic binary search tree.

2 Data Structure

A B-tree may be represented as a binary tree where each node in the B-tree corresponds to a number of binary nodes which together form a *pseudo-node*. Each pseudo-node is in itself a small binary tree. The pseudo-node containing the root of the entire tree is called the *pseudo-root*. The number of binary nodes in a pseudo-node is the *size* of the pseudo-node. The *maximum* and *minimum height* of a pseudo-node is the longest and shortest distance

from its topmost node to a node outside the pseudo-node, respectively. A pseudo-node is *optimally balanced* if the difference between its maximum and minimum heights is at most 1.

Using the terminology above, which is different from that used by Bayer[2], the *symmetric binary B-tree* [3], or *SBB-tree*, may be defined as a representation of a 2-4-tree where each pseudo-node is optimally balanced. An example of an SBB-tree is given in Figure 1.

The generalized SBB-tree presented here is different from the generalization suggested by Bayer. In his structure, a pseudo-node can take any shape as long as its maximum height is less than or equal to some constant. In our structure, the SBB(k)-tree, each pseudo-node is required to be an optimally balanced tree.

A formal definition of SBB(k)-trees is given below.

Definition 1 *For a given integer constant k, $k \geq 1$, a generalized symmetric binary B-tree, or SBB(k)-tree, is a binary tree which satisfies the following balance criteria:*

- *The pseudo-root is optimally balanced and has a maximum height $\leq k+1$;*
- *All other pseudo-nodes have a minimum height $\geq k$ and a maximum height $\leq k+1$.*
- *The number of pseudo-nodes on the path from the root to a leaf is the same for all leaves.*

From the definition follows that all pseudo-nodes except the pseudo-root have a size between $2^k - 1$ and $2^{k+1} - 1$. The a-b-tree corresponding to an SBB(k)-tree has $a = 2^k$ and $b = 2^{k+1}$. For $k = 1$ we get a symmetric binary B-tree.

To keep an SBB(k)-tree balanced we have to be able to identify pseudo-nodes and determine their maximum and minimum heights. One bit stored in each binary node tells whether the node is the topmost node of a pseudo-node or not. From this information the minimum and maximum heights of a pseudo-node P can be computed by explicit examination of the binary nodes in P. If this examination is considered to be too time-consuming,

[2]Instead of the term pseudo-node Bayer uses two types of edges, edges within a pseudo-node are *horizontal* and edges between pseudo-nodes are *vertical*. The definitions are equivalent, but the notation of pseudo-nodes will simplify our presentation.

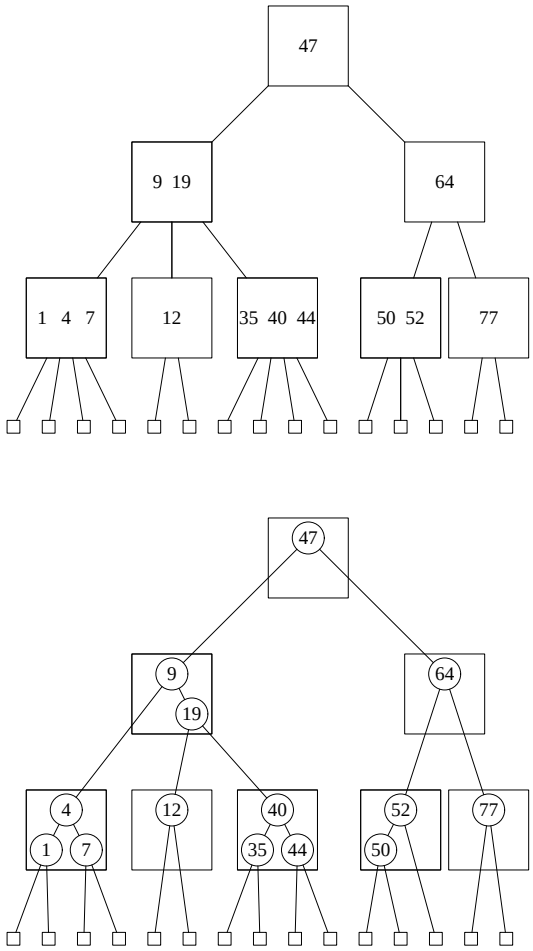

Figure 1: *A 2-4-tree and a corresponding SBB-tree.*

the balance information may be stored in another somewhat more space-requiring way. In this case two integers stored in each binary node in P give the minimum and maximum distances from the binary node to a node outside P.

A nice property of the SBB(k)-tree is that the constant k may be chosen in a way which brings its height arbitrary close to the optimal height of a binary search tree, which is shown below.

Theorem 1 *Let T be an SBB(k) tree. Then*

$$height(T) \leq \left\lceil \left(1 + \frac{1}{k}\right) \log(n+1) \right\rceil . \tag{1}$$

Proof: From the definition above we know that the B-tree corresponding to an SBB(k)-tree is an a-b-tree where $a = 2^k$. The height of this B-tree is the same as the number of pseudo-nodes on a path from the root to a leaf in the SBB(k)-tree. From the analysis of B-trees [4, 7] we know the height of an a-b-tree:

$$\begin{aligned} \text{height}(a\text{-}b\text{-tree}) &\leq 1 + \left\lfloor \log_a \frac{n+1}{2} \right\rfloor \\ &= 1 + \left\lfloor \frac{\log(n+1) - 1}{\log 2^k} \right\rfloor \end{aligned} \tag{2}$$

To compute the height of an SBB(k)-tree we use the fact that the shortest distance from the root to a leaf is at most $\lfloor \log(n+1) \rfloor$. Since each pseudo-node is optimally balanced, the difference between the depths of two leaves is at most one per level in the corresponding B-tree. Thus, the height of an SBB(k)-tree T is at most the sum of the minimum depth of a leaf and the maximum height of the corresponding a-b-tree, that is

$$\begin{aligned} \text{height}(T) &\leq \lfloor \log(n+1) \rfloor + \text{height}(a\text{-}b\text{-tree}) \\ &\leq \lfloor \log(n+1) \rfloor + 1 + \left\lfloor \frac{\log(n+1) - 1}{\log 2^k} \right\rfloor \\ &\leq \left\lfloor \left(1 + \frac{1}{k}\right) \log(n+1) + 1 - \frac{1}{k} \right\rfloor \\ &\leq \left\lceil \left(1 + \frac{1}{k}\right) \log(n+1) \right\rceil \end{aligned} \tag{3}$$

which completes the proof. □

From Theorem 1 we conclude that a large value of k keeps the height of the tree close to optimal. However, the amount of restructuring work required per update increases as k increases.

3 Maintenance Algorithms

The algorithms for insertion and deletion in an SBB(k)-tree are closely related to those for the same operations in a B-tree. In order to describe the algorithms we define some basic operations below.

3.1 Basic Operations

Let P and S denote two pseudo-nodes. The updating algorithms for an SBB(k)-tree use the following operations:

- *Balance P:* P is rebuilt to become optimally balanced.
- *Split P:* P is split into two pseudo-nodes and its topmost node is inserted into the parent pseudo-node.
- *Join P and S:* P and S are joined, the common parent of their respective topmost nodes is deleted from the parent pseudo-node and becomes the topmost node of the new pseudo-node.
- *Co-balance P and S:* P and S are joined, the new pseudo-node is balanced and split into P and S.
- *Single rotation.*

The *balancing* operation is performed when insertion or removal of a binary node makes a pseudo-node unbalanced, that is, when its maximum and minimum heights differ by two.

The *split* is used when a pseudo-node becomes too large, that is, when insertion of a binary node increases its size to 2^{k+1}. In this case, before the binary node was added the pseudo-node had its largest possible size with a maximum and minimum height of $k+1$. Adding the binary node gave a maximum height of $k+2$ and thus, the pseudo-node is still optimally balanced. Splitting it will result in two pseudo-nodes with minimum heights of k and maximum heights of k and $k+1$, respectively (Figure 2b). Thus, both of the new pseudo-nodes satisfy the definition of an SBB(k)-tree and

no balancing of pseudo-nodes is required. The topmost node of the old pseudo-node will be inserted into the parent pseudo-node.

When the *pseudo-root* is split a new pseudo-root of size one is created. In this case the number of pseudo-nodes on each path from the root to a leaf will increase by 1, which corresponds to increasing the height of a B-tree.

The *join* and *co-balancing* operations are performed when a pseudo-node becomes too small. This occurs when removal of a binary node decreases the size of the pseudo-node to $2^k - 2$. Before the deletion the pseudo-node had its smallest possible size with a minimum and maximum height of k. Thus, after the removal the pseudo-node has a minimum height of $k-1$ and a maximum height of k and is optimally balanced. Since the pseudo-node is too small, we have to either join it with another pseudo-node or add nodes to it.

When two pseudo-nodes are joined or co-balanced, their topmost nodes have to be siblings. In some cases a *single rotation* is needed to obtain this configuration.

The *join* is performed when the pseudo-node's sibling has its smallest possible size. That is, when both its maximum and its minimum heights are k. The resulting joined pseudo-node will have a minimum height of k and a maximum height of $k+1$ (Figure 3d) and it will satisfy the definition of an SBB(k)-tree. If the pseudo-root contains only one binary node it will disappear when its two children are joined. In this case the joined pseudo-node will become the new pseudo-root and the number of pseudo-nodes on each path from the root to a leaf will decrease by 1, which corresponds to decreasing the height of a B-tree.

Co-balancing is performed when the sibling of the pseudo-node is too large to allow joining. The effect of this operation is that binary nodes are moved from the larger of the two pseudo-nodes to the smaller one. After the co-balancing both pseudo-nodes will satisfy the definition of an SBB(k)-tree.

The split and join operations are performed merely by changing the stored balance information in three binary nodes and do not change the tree structure at all. Thus, both these operations are very simple to perform in contrast to the corresponding operations for an ordinary B-tree. The cost of balancing or co-balancing is proportional to the size of a pseudo-node, which is depending of k.

3.2 Insertion and Deletion

The good properties of the maintenance algorithms for an SBB(k)-tree are due to the fact that a split or a join operation do not change the tree structure, and therefore do not require any restructuring. The other operations (balancing, co-balancing, and single rotation) leads to immediate termination of the updating algorithms. These two facts give that an SBB(k)-tree can be updated with only a constant amount of restructuring work. Note that only rotations and rebalancing are regarded as restructuring work, while changing stored balance information is not, since the shape of the tree is not affected.

We present the algorithms below. The analysis is given in section 3.3. In the following we let $\max(P)$ and $\min(P)$ denote the maximum and minimum heights of the pseudo-node P. P_{top} denotes P's topmost node.

Insertion:

When a new key is to be inserted into an SBB(k)-tree we follow a search path down the tree until a leaf is reached. A new node is created at the bottom of the tree and the key is placed there. This may violate one of the criteria given in the definition of the SBB(k)-tree by making a pseudo-node P either unbalanced or too large. We have the following cases:

Case 1: $\max(P) - \min(P) = 2$. P is not optimally balanced. We balance P and terminate.

Case 2: $\max(P) = k + 2$. P is too large. We split P. One binary node is added to the parent pseudo-node. Therefore we proceed to the parent pseudo-node to look for possible violations of the balance criteria.

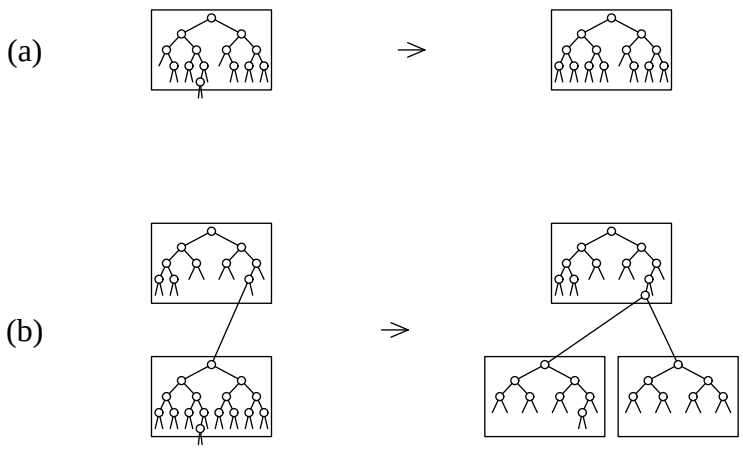

Figure 2: *Insertion into an SBB(3)-tree. (a) Case 1: The pseudo-node is balanced. (b) Case 2: the pseudo-node is split. In this example the splitting results in an imbalance in the parent pseudo-node.*

Deletion:

When a key is to be deleted from an SBB(k)-tree we follow a search path down the tree until the binary node containing the key is found. If the node is at the bottom of the tree it is removed and replaced by a leaf, otherwise it is replaced by its predecessor (or successor), which is found at the bottom of the tree. The effect will be that one node is deleted from a pseudo-node P at the lowest level of the tree, which may make P either unbalanced or too small. As for insertion, the two cases cannot occur at the same time.

Case 1: $\max(P) - \min(P) = 2$. P is not optimally balanced. We balance P and terminate.

Case 2: $\max(P) = k-1$. P is too small and has to be joined or co-balanced with another pseudo-node. (Note that this case will not occur at the pseudo-root since its minimum height is not restricted.) In order to join or co-balance P with another pseudo-node S, P_{top} and S_{top} have to be siblings. If there is no such pseudo-node S we obtain this by a single rotation. After this step two subcases occur:

Case 2a: $\max(S) = k+1$. S is large enough to give away a node. We co-balance P and S and terminate.

Case 2b: $\max(S) = k$. S has its smallest possible size. We join P and S and proceed to the parent pseudo-node to check for possible violations of the balance criteria.

The correctness of the algorithms for insertion and deletion follows from the fact that

1. when an element is inserted into or deleted from an SBB(k)-tree, no other violation of the balance criteria than the cases treated by the algorithms above may occur;
2. both algorithms will terminate when the topmost pseudo-node is reached;
3. all terminating states of the algorithms represent an SBB(k)-tree.

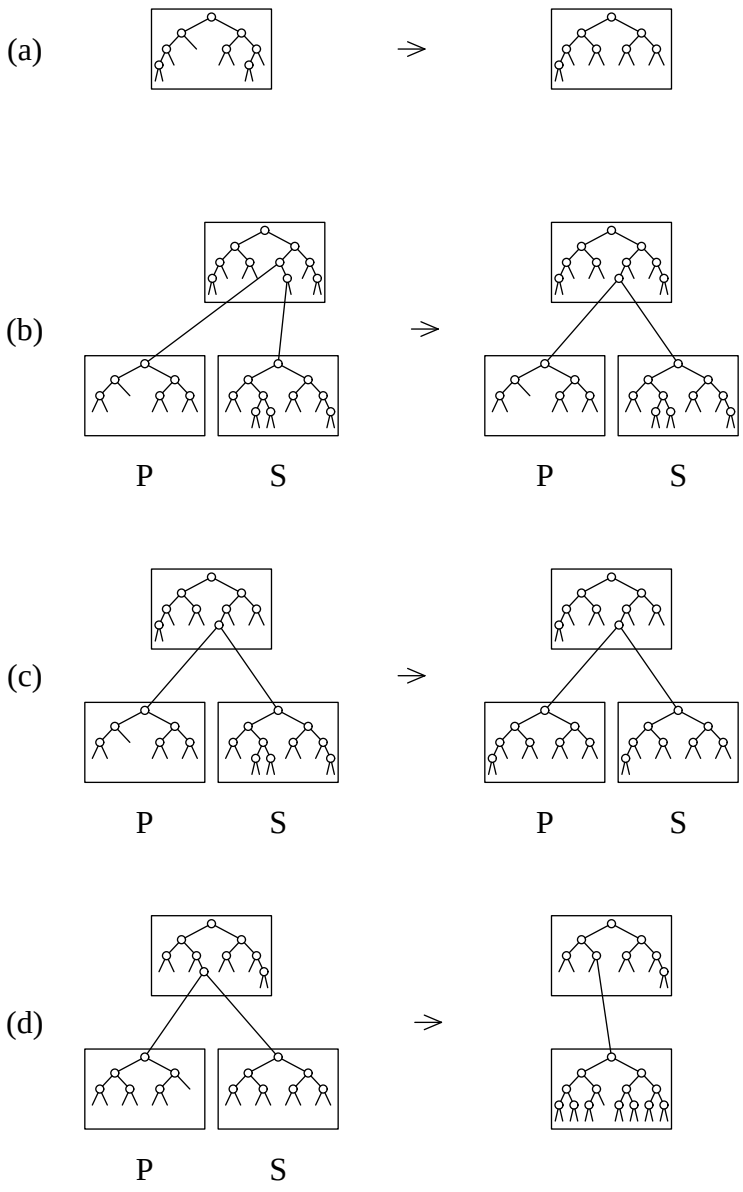

Figure 3: *Deletion in an SBB(3)-tree. (a) Case 1: The pseudo-node is balanced. (b) A rotation is performed in Case 2 to make P's and S's topmost nodes become siblings. (c) Case 2a: P and S are co-balanced. (d) Case 2b: P and S are joined.*

3.3 Analysis

The maintenance algorithms given above use few restructurings, as shown below.

Theorem 2 *An SBB(k)-tree can be updated in* $\Theta(\log n)$ *time and with a constant amount of restructuring work in the worst case.*

Proof: In order to prove the constant amount of restructuring work we take a closer look at the maintenance algorithms presented above. Both algorithms work bottom-up, starting at the lowest pseudo-node on the visited path.

Insertion: The two cases which may occur are described in the algorithm above. Rebalancing is required only in Case 1. Having balanced the pseudo-node, it satisfies the definition of an SBB(k)-tree and the algorithm terminates. Thus, at most one balancing may be performed per insertion, which proves a constant amount of restructuring work.

Deletion: As for insertion, the balance criteria are satisfied as soon as a balancing or a co-balancing has been performed. The only case a restructuring operation does not lead to immediate termination is when the single rotation in Case 2 is followed by a join. After the join we continue to the parent pseudo-node to look for a violation of the balance criteria. However, a rotation is required only when the parent pseudo-node of P and S has its bottom nodes at two different levels (Figure 3b). In this case, removal of one node from that pseudo-node cannot cause any violation of the balance criteria. Thus, the algorithm will terminate also when a rotation in Case 2 is followed by a join. From this follows that the amount of restructuring required by a deletion is at most one single rotation plus one co-balancing.

From the description above follows that both algorithms work along a path down the tree, spending constant time at each level, which proves the overall logarithmic cost per update.

Thus, we have shown that the restructuring work is constant, and that the cost per update is logarithmic, which completes the proof. □

Even if the amount of restructuring work is constant it might be of some interest to determine how much is actually required. The exact amount depends on the algorithm used to balance pseudo-nodes. There are several

algorithms for balancing binary search trees which may be used for this purpose [5, 6, 10, 19].

As an example we can study the algorithm given by Stout and Warren [19]. This algorithm is not the most efficient one but it uses rotations which makes the balancing cost comparable with the cost required for other classes of search trees. The Stout-Warren algorithm works in two steps:

1. A right-degenerated tree is produced by repeated right rotations.

2. A balanced tree is produced from the degenerated one by repeated left rotations.

Let P be an unbalanced pseudo-node and P' the same pseudo-node after rebalancing. Let $\mathrm{ll}(P)$ and $\mathrm{rl}(P)$ denote the length of P's leftmost and rightmost path, respectively. The number of rotations required by the Stout-Warren algorithm is $2|P| - \mathrm{rl}(P) - \mathrm{rl}(P')$. This number may be slightly reduced by two modifications. In the first step of the algorithm we can make P left- or right-degenerated, depending on which one of $\mathrm{ll}(P)$ and $\mathrm{rl}(P)$ is largest. In step 2 we can keep as many nodes as possible on the degenerated path, that is $\lceil \log(|P|+1) \rceil$ nodes. With these modifications the number of rotations required to balance the pseudo-node P is

$$2|P| - \mathrm{Max}\,(\mathrm{ll}(P), \mathrm{rl}(P)) - \lceil \log(|P|+1) \rceil\,. \tag{4}$$

In Theorem 3 below we give the maximum number of rotations required for updates in an SBB(k)-tree when this modified balancing algorithm is used. Note that in the proof of Theorem 2 we claimed that only one rotation is required. In this case we also use rotations for the rebalancing operations. This is not a contradiction, since rebalancing can be made without rotations.

Theorem 3 *An SBB(k)-tree can be maintained with* $2^{k+2} - 2k - 4$ *rotations per insertion and* $3 \cdot 2^{k+1} - 2k - 7$ *rotations per deletion in the worst case.*

Proof: An examination of the worst possible cases gives the following:

Insertion: The worst case occurs when a pseudo-node of maximum size, $2^{k+1} - 1$, is rebalanced. In such a pseudo-node there is one path of length k and one path of length $k+2$, the rest of the paths have a length of $k+1$. Thus, at least one of the right- and leftmost paths

has a length of $k+1$ before rebalancing. From Eq. (4) the number of rotations is at most

$$\begin{aligned} & 2(2^{k+1}-1)-(k+1)-\left\lceil \log 2^{k+1} \right\rceil \\ = \; & 2^{k+2}-2k-4. \end{aligned} \tag{5}$$

Deletion: The worst case occurs in Case 2 when a single rotation is followed by co-balancing of two pseudo-nodes of maximum size. The number of nodes involved in the co-balancing is at most $3 \cdot 2^k - 2$. The maximum height of the left- and rightmost path before the balancing is at least $k+2$. From Eq. (4) we get the number of rotations for a co-balancing to be

$$\begin{aligned} & 2(3 \cdot 2^k - 2)-(k+2)-\left\lceil \log(3 \cdot 2^k - 1) \right\rceil \\ = \; & 3 \cdot 2^{k+1} - 2k - 8. \end{aligned} \tag{6}$$

Adding the single rotation, we get a total sum of

$$3 \cdot 2^{k+1} - 2k - 7. \tag{7}$$

The proof follows from Eqs. (5) and (7). □

Thus, the SBB(k)-tree combines a low height with a low cost of rebalancing. As an example, $k=2$ gives a height of $1.5\log(n+1)$, 8 rotations for insertion and 13 for deletion. Which value of k to choose in a particular case depends on the expected size of the tree, the ratio of searches to updates, and the cost of comparisons compared to the cost of restructuring.

Note that the number of rotations required for an SBB(1)-tree is the same as that required by the algorithms for the SBB-tree by Guibas, Sedgewick and Tarjan [8, 18, 20]. Their algorithms may be regarded as special cases of our algorithms for an SBB(1)-tree. However, the algorithms are not identical. In some cases our modified Stout-Warren algorithm makes a rotation which is not necessary (Case 2a in the deletion algorithm when $|S| = 3$). Thus the average number of rotations may be improved by optimizing the Stout-Warren algorithm further. For $k > 1$ it may be possible to improve the worst-case number of rotations using a better rebalancing algorithm.

4 Improved Height of the SBB(k)-tree

In the preceding sections we have shown that the SBB(k)-tree has a height of $\lceil (1+\epsilon)\log(n+1)\rceil$ for any positive value of ϵ. By varying the value of k in such a way that the maintenance cost remains logarithmic, the height may be reduced to $\log n + o(\log n)$, which is optimal in the leading term. Unfortunately, continuous variation of k cannot be afforded since the entire tree has to be reconstructed for each new value. This problem is circumvented by changing k only at preset intervals.

We start by presenting an amortized result.

Theorem 4 *There is a binary search tree T for which*

$$height(T) \leq \left\lceil \log(n+1) + \frac{\log(n+1)}{\log\log(n+1)} \right\rceil \tag{8}$$

and the amortized cost per update is $\Theta(\log n)$.

Proof: We use a modified SBB(k)-tree. Associated with the tree we have a counter which is increased by one each time an element is inserted or deleted. When the value of the counter equals half the number of elements in the tree, k is set to $\lceil \log\log 2n \rceil$ and the entire tree is reconstructed in linear time. Then, $\Theta(n)$ updates have been made since the last total rebalancing, which gives an amortized cost of $O(1)$ per update for changing the value of k.

Let N denote the size of the tree when the latest total restructuring was made. From the fact that a total rebalancing is made after every $\frac{n}{2}$th update we know that the size of the tree is bound to be between $\frac{2}{3}N$ and $2N$. This gives

$$\lceil \log\log(n+1) \rceil \leq k \leq \lceil \log\log 3n \rceil. \tag{9}$$

From Theorem 1 we know that

$$\begin{aligned} \text{height}(T) &\leq \left\lceil \left(1+\frac{1}{k}\right)\log(n+1) \right\rceil \\ &\leq \left\lceil \left(1+\frac{1}{\log\log(n+1)}\right)\log(n+1) \right\rceil . \end{aligned} \tag{10}$$

From Theorem 3 we know that the maximal amount of restructuring work required per update is $O(2^k)$. Since $k = O(\log\log n)$ we have a restructuring cost of

$$O(2^{\log\log n}) = O(\log n). \tag{11}$$

The proof follows from Eqs. (10) and (11). □

5 An Efficient Worst-Case Solution to the Dictionary Problem

The amortized result obtained in Theorem 4 may be transformed into a worst-case result by maintaining two trees simultaneously instead of one. In this way the cost of changing the value of k (including a global rebuilding) may be distributed in such a way that the worst-case cost per update becomes logarithmic. Providing that the two trees are not being rebuilt at the same time, there will always be one tree in which queries can be performed.

Our technique is an improvement of the method of *global rebuilding* introduced by Overmars [17]. Using the rank of elements we avoid making extra element-comparisons when updating two trees.

Theorem 5 *There is a data structure supporting search and updates at a cost of*

$$\left\lceil \log(n+1) + \frac{\log(n+1)}{\log\log(n+1)} \right\rceil$$

comparisons and $\Theta(\log n)$ *time in the worst case.*

Proof: We use a data structure consisting of two SBB(k)-trees where $k \approx \log\log n$, as described in the proof of Theorem 4. In each node we store the number of descendants. The data structure is maintained in the following way:

1. After $\frac{n}{2}$ updates the value of k is changed and each tree is rebuilt. Rebuildings are scheduled in such a way that the two trees are not being rebuilt at the same time.

2. Rebuilding work is distributed over the updates in such a way that the time spent per update is $O(\log n)$.

3. Updates are performed in the following way:

   (a) An update is always made in a tree which is not being reconstructed. From the number of descendants, which is stored in each node, we compute the rank of the inserted/deleted element.

(b) If the other tree is not being rebuilt the insertion/deletion is made also in that tree. No comparisons are required for this update, since we use the rank to locate the element.

(c) If the other tree is being rebuilt we store the update in a queue to be performed when the rebuilding is completed.

Rebuilding of each of the trees includes construction of a new tree and performance of the queued updates. The first step requires $O(n)$ time and the second one $O(n \log n)$ time. Therefore, the entire procedure may be distributed over a linear number of updates at a worst-case cost of $O(\log n)$ per update.

Although each update is performed in both trees, element-comparisons are made only the first time. Thus, the number of comparisons equals the height of the tree. The rest of the proof is similar to the proof of Theorem 4. □

6 Comments

We have presented a generalization of the symmetric binary B-tree, the SBB(k)-tree. The new tree is of low height and has efficient updating algorithms requiring only a constant number of restructurings per operation. Instead of using only primitive operations such as rotations, the algorithms for maintenance of SBB(k)-trees are based on rebalancing of larger structures. In this way we avoid several special cases, which makes the presentation clearer. The technique of representing B-tree nodes as large binary trees is also used by van Leeuwen and Overmars [21] to obtain $\log n$-maintainable subclasses of some search trees, for example AVL-trees and α-balanced trees. While their results mainly are of theoretical interest, the SBB(k)-tree is an alternative to other search trees well worth to be considered in practical applications.

Introducing SBB(k)-trees gives an answer to an open problem formulated by Olivie [14]. He presented a class of balanced trees called *half-balanced trees*, which in fact is the same class as the symmetric binary B-trees, and showed that trees of this class can be updated with a constant number of rotations per operation. The question was whether there are other classes of balanced trees with this property. Clearly, for any value of k larger than 1 the SBB(k)-trees form a new class with this property. A more detailed

study of tree structures requiring a constant number of rotations per update can be found in [16].

Apart from the constant amount of restructuring work per update there are other properties of the SBB(k)-trees which may be derived from the SBB-trees. For instance, there are updating algorithms for SBB-trees which use top-down balancing [8]. The idea is to perform all split and join operations which may be required on the way down the tree during the search. Such algorithms are useful in a parallel environment since one operation on the tree may be started before the previous one is completed. Another example is the ability to join and split SBB-trees [8].

Based on the SBB(k)-tree we have also presented a data structure which may be efficiently maintained in the worst case with a number of comparisons per dictionary operation which is optimal in the leading term. Thus, we have improved the upper bound on the worst-case complexity of the dictionary problem.

Acknowledgements

We would like to thank Dr. Svante Carlsson and the referees for valuable comments on this paper.

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 Mass, 1983.

[3] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[4] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[5] H. Chang and S. S. Iynegar. Efficient algorithms to globally balance a binary search tree. *Communications of the ACM*, 27(7):695–702, 1984.

[6] A. C. Day. Balancing a binary tree. *Computer Journal*, 19(4):360–361, 1976.

[7] G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1983. ISBN 0-201-0023-7.

[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] C. Icking, R. Klein, and T. Ottmann. Priority search trees in secondary memory. In *Graphtheoretic Concepts in Computer Science (WG '87), Staffelstein, LNCS 314*, pages 84–93, 1987.

[10] W. A. Martin and D. N. Ness. Optimizing binary trees grown with a sorting algorithm. *Communications of the ACM*, 15(2):88–93, 1972.

[11] H. A. Mauer, T. Ottmann, and H. W. Six. Implementing dictionaries using binary trees of very small height. *Information Processing Letters*, 5(1):11–14, 1976.

[12] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

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

[14] H. J. Olivie. *A Study of Balanced Binary Trees and Balanced One-Two-Trees*. Ph. D. Thesis, Dept of Mathematics, University of Antwerp, 1980.

[15] 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.

[16] Th. Ottmann and D. Wood. Updating binary trees with constant linkage cost. To appear in Proc. Scandinavian Workshop on Algorithm Theory, SWAT '90, Bergen, 1990.

[17] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer Verlag, 1983. ISBN 3-540-12330-X.

[18] N. Sarmak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[19] Q. F. Stout and B. L. Warren. Tree rebalancing in optimal time and space. *Communications of the ACM*, 29(9):902–908, 1986.

[20] R. E. Tarjan. Updating a balanced search tree in $O(1)$ rotations. *Information Processing Letters*, 16:253–257, 1983.

[21] J. van Leeuwen and M. H. Overmars. Stratified balanced search trees. *Acta Informatica*, 18:345–359, 1983.