PREPRINT. In Proc. Scandinavian Workshop on Algorithm Theory, pages 111-121. Springer Verlag, 1990.

Fast Updating of Well-Balanced Trees

Arne Andersson
Lund University
Sweden

Tony W. Lai
University of Waterloo
Canada

Abstract

We focus on the problem of maintaining binary search trees with an optimal and near-optimal number of incomplete levels. For a binary search tree with one incomplete level and a height of exactly $\lceil \log(n+1) \rceil$, we improve the amortized insertion cost to $O(\log^3 n)$. A tree with 2 incomplete levels and a near-optimal height of $\lceil \log(n+1) + \epsilon \rceil$ may be maintained with $O(\log^2 n)$ amortized restructuring work per update. The amount of restructuring work is decreased to $O(\log n)$ by increasing the number of incomplete levels to 4, while the height is still kept as low as $\lceil \log(n+1) + \epsilon \rceil$. This yields an improved amortized bound on the dictionary problem.

Trees of optimal and near-optimal height may be represented as a pointer-free structure in an array of size $O(n)$. In this way we obtain an array implementation of a dictionary with $O(\log n)$ search cost and $O(\log^2 n)$ update cost, allowing interpolation search to improve the expected search time.

1 Introduction

The binary search tree is a fundamental and well studied data structure, commonly used in computer applications to implement the abstract data type *dictionary*. In a comparison-based model of computation, the lower bound on the three basic operations *insert*, *delete* and *search* is $\lceil \log(n+1) \rceil$ comparisons per operation. This bound may be achieved by storing the set in a binary search tree of optimal height.

Definition 1 *A binary tree has optimal height if and only if the height of the tree is* $\lceil \log(n+1) \rceil$.

A special case of a tree of optimal height is an *optimally balanced* tree, as defined below.

Definition 2 *A binary tree is optimally balanced if and only if the difference in length between the longest and shortest paths is at most one.*

In the literature, there are a number of algorithms presented for maintenance of optimally balanced trees, all having an amortized cost of $\Theta(n)$ per update [7, 8, 9, 15]. Although this cost is high, it is regarded as affordable when updates are rare and when a low search cost is essential.

In this paper we improve the tradeoff between balance and maintenance cost for binary search trees. Starting with optimal balance, that is, at most one incomplete level and a height of $\lceil \log(n+1) \rceil$, we improve the amortized insertion time from linear to $O(\log^3 n)$. Ending with at most 4 incomplete levels and a near-optimal height of $\lceil \log(n+1) + \epsilon \rceil$, we achieve an optimal amortized update cost of $O(\log n)$ per insertion and deletion.

The paper is organized as follows. In Section 2, we show that a tree, defined by a simple balance criterion similar to the general balanced tree [3, 5], may be maintained with at most 2

incomplete levels and a height of $\lceil \log(n+1) + \epsilon \rceil$ at an amortized cost of $O(\log^2 n)$ per update. This result is analogous to the bound for 2-complete trees and weight-balanced trees, obtained by Lai and Wood [11, 12].

In Section 3, we turn to the problem of maintaining a tree with at most one incomplete level and an optimal height of exactly $\lceil \log(n+1) \rceil$. Based on the result in Section 2, we give an insertion algorithm with an amortized cost of $O(\log^3 n)$, provided that no deletions are made. Just as there is an efficient algorithm for insertions only into an optimally balanced tree, there is an efficient algorithm for deletions only.

In Section 4, we show that by allowing at most 4 incomplete levels, we can maintain a tree with near-optimal height $\lceil \log(n+1) + \epsilon \rceil$ at a cost of $O(\log n)$ per update. This result is achieved by introducing *dense trees* in which we replace the leaves of a tree of size $\Theta(n/\log n)$ by perfectly balanced subtrees of size $\Theta(\log n)$.

The algorithms presented in Sections 2 and 3 use partial rebuilding for maintenance. This fact, together with the low height, implies that a tree of optimal or near-optimal height may be represented as a pointer-free structure in an array of size $O(n)$. In this way we obtain an array implementation of a dictionary with $O(\log^2 n)$ update cost, allowing interpolation search to improve the expected search time.

The number of leaves in the subtree rooted at node v is called the *size* of v, denoted $|v|$. this implies that the size of a tree containing n elements is $n+1$. We assume that rebuilding the subtree rooted at v to perfect balance requires time proportional to the size of v. Linear algorithms for balancing a binary tree can be found in [7, 8, 15].

2 Maintaining Near-Optimal Balance

If we allow the height of a binary search tree to be an additive constant larger than the optimal height, we have a tree of *near-optimal* height. In this section we show that a near-optimal height of $\lceil \log(n+1) + \epsilon \rceil$ may be maintained at an amortized cost of $O(\log^2 n)$ per update. We also show how to bound the length of the shortest path to $\lfloor \log(n+1) - \epsilon \rfloor$. In this way, we are able to maintain a tree with at most two incomplete levels.

We use the simplest possible balance criterion, allowing the tree to take any shape as long as the height is at most $\lceil \log(n+1) + \epsilon \rceil$ for a given constant ϵ. As we will show, this simple balance criterion is sufficient to maintain the tree at an amortized cost of $O(\log^2 n)$ per update.

Theorem 1 *A binary search tree of height $\lceil \log(n+1) + \epsilon \rceil$ may be maintained with an amortized cost of $O\left(\frac{\log^2 n}{\epsilon}\right)$ per update, for any constant $\epsilon > 0$.*

Proof: We maintain the tree by partial rebuilding. The entire tree is rebuilt when the number of updates since the last global rebuilding equals half the size of the tree. The cost of each of these global rebuildings is amortized over $\Theta(n)$ updates, which gives an amortized cost of $O(1)$ per update.

We still have to show the amortized cost of the updates between these global rebuildings. To prove the cost of updating we define a potential function $\Phi(v)$ for each node v. The function is chosen in such a way that the decrease in potential during a rebuilding corresponds to the cost of the rebuilding. The increased potential during an update corresponds to the amortized cost of the update. By showing that the potential is always positive or zero, we prove that the amortized cost covers the cost of restructuring.

Let $\delta(v)$ denote the difference in sizes between v's two subtrees and n_0 denote the size of the entire tree at the latest global rebuilding. The cost of rebuilding the subtree rooted at v

to perfect balance is $R|v|$ for some constant R, where $|v|$ denotes the number of leaves in the subtree v. The potential $\Phi(v)$ is chosen as

$$\Phi(v) = \begin{cases} 0 & \text{if } v \text{ is perfectly balanced} \\ \frac{4R \log n_0}{\epsilon} \delta(v) & \text{otherwise.} \end{cases} \tag{1}$$

A single update changes the potential by $O\left(\frac{\log n_0}{\epsilon}\right)$ at a logarithmic number of nodes, which implies that the amortized cost per update is $O\left(\frac{\log^2 n}{\epsilon}\right)$.

Left to show is that there is a maintenance algorithm with a restructuring cost which is covered by the potential function Φ. When a single insertion or deletion causes the height of the tree to exceed $\lceil \log(n+1) + \epsilon \rceil$, we make a partial rebuilding at the lowest node v which satisfies

$$\text{height}(v) > \left\lceil \left(1 + \frac{\epsilon}{\log(n+1)}\right) \log |v| \right\rceil, \tag{2}$$

Clearly, at least one such node v exists, since Eq. (2) is satisfied by the root. Let v_1 denote v's highest child. Since v is the lowest node satisfying Eq. (2), we know that

$$\text{height}(v_1) \leq \left\lceil \left(1 + \frac{\epsilon}{\log(n+1)}\right) \log |v_1| \right\rceil \tag{3}$$

and

$$\text{height}(v) = \text{height}(v_1) + 1. \tag{4}$$

Combining Eqs. (2), (3), and (4) gives

$$\begin{aligned} \left\lceil \left(1 + \frac{\epsilon}{\log(n+1)}\right) \log |v| \right\rceil &< \text{height}(v) \\ &= \text{height}(v_1) + 1 \\ &\leq \left\lceil \left(1 + \frac{\epsilon}{\log(n+1)}\right) \log |v_1| \right\rceil + 1 \end{aligned} \tag{5}$$

which implies that

$$|v_1| > 2^{-1/\left(1 + \frac{\epsilon}{\log(n+1)}\right)} |v|. \tag{6}$$

The cost of a rebuilding at the node v is $R|v|$. This cost has to be covered by the decrease in v's potential. After the rebuilding the subtree v is perfectly balanced and $\Phi(v) = 0$. From Eq. (6) we get the potential immediately before the rebuilding.

$$\begin{aligned} \Phi(v) &= \frac{4R \log n_0}{\epsilon} \delta(v) \\ &= \frac{4R \log n_0}{\epsilon} \left(|v_1| - (|v| - |v_1|)\right) \\ &= \frac{4R \log n_0}{\epsilon} \left(2|v_1| - |v|\right) \\ &> \frac{4R \log n_0}{\epsilon} \left(2^{1 - 1/\left(1 + \frac{\epsilon}{\log(n+1)}\right)} - 1\right) |v| \\ &> \frac{2R \log n_0}{\epsilon} \frac{\epsilon |v|}{\log(n+1)} \\ &> R|v|. \end{aligned} \tag{7}$$

The fact that

$$2\left(2^{1 - 1/\left(1 + \frac{\epsilon}{\log n}\right)} - 1\right) > \frac{\epsilon}{\log n}, \quad \epsilon < \log n \tag{8}$$

is straightforward to show by the substitution

$$t = \frac{1}{1 + \frac{\epsilon}{\log n}}. \tag{9}$$

From Eq. (7) follows that the decrease in potential covers the restructuring cost, which completes the proof. □

As a matter of fact, the algorithm described above may be viewed as the maintenance algorithm for a general balanced tree or GB(c)-tree [3, 5], where we let the value of the tuning parameter c vary such that $c = 1 + \frac{\epsilon}{\log(n+1)}$.

In the same way as we may bound the longest path to be near-optimal, we may bound the shortest path to be close to maximum.

Theorem 2 *A binary search tree with a shortest path of at least* $\lfloor \log(n+1) - \epsilon \rfloor$ *may be maintained with an amortized cost of* $O\left(\frac{\log^2 n}{\epsilon}\right)$ *per update, for any constant* $0 < \epsilon \leq 1$.

Proof: The proof is similar to the proof of Theorem 1. Let $\mathrm{short}(v)$ denote the length of v's shortest path. Each time a path becomes too short, we make a partial rebuilding at the lowest node v which satisfies

$$\mathrm{short}(v) < \left\lfloor \left(1 - \frac{\epsilon}{\log(n+1)}\right) \log |v| \right\rfloor. \tag{10}$$

Let v_1 denote v's child on the shortest path. Since v is the lowest node satisfying Eq. (10) we know that

$$\mathrm{short}(v_1) \geq \left\lfloor \left(1 - \frac{\epsilon}{\log(n+1)}\right) \log |v_1| \right\rfloor \tag{11}$$

and

$$\mathrm{short}(v) = \mathrm{short}(v_1) + 1. \tag{12}$$

Combining Eqs. (10), (11), and (12) gives

$$\begin{aligned} \left\lfloor \left(1 - \frac{\epsilon}{\log(n+1)}\right) \log |v| \right\rfloor &> \mathrm{short}(v) \\ &= \mathrm{short}(v_1) + 1 \\ &\geq \left\lfloor \left(1 - \frac{\epsilon}{\log(n+1)}\right) \log |v_1| \right\rfloor + 1 \end{aligned} \tag{13}$$

which implies that

$$|v_1| < 2^{-1/\left(1 - \frac{\epsilon}{\log(n+1)}\right)} |v|. \tag{14}$$

As in the proof of Theorem 1, we may choose a potential function $\Phi(v)$ to be $\Theta\left(\frac{\log n}{\epsilon}\delta(v)\right)$, where $\delta(v)$ denotes the difference in sizes between v's two subtrees. From Eq. (14) follows that when v is to be rebuilt, the value of $\delta(v)$ will be large enough to make the potential cover the cost of the rebuilding. □

By combining the results of Theorem 1 and Theorem 2, we may restrict both the longest and the shortest path in the tree.

Theorem 3 *A binary search tree with a maximum height of* $\lceil \log(n+1) + \epsilon \rceil$ *and a shortest path of length* $\lfloor \log(n+1) - \epsilon \rfloor$ *may be maintained at a cost of* $O\left(\frac{\log^2 n}{\epsilon}\right)$ *per update.*

Proof: By combining the two algorithms described in the proofs of Theorem 1 and Theorem 2 we obtain the bounds stated above. □

3 Maintaining Optimal Balance

A consequence of Theorem 3 is that when the value of ϵ is small, the height of the tree will be $\lceil \log(n+1) \rceil$ and the shortest path will be of length $\lfloor \log(n+1) \rfloor$, except when n is close to a power of 2. Thus, the number of incomplete levels in the tree will be one in most cases and two at most.

This fact can be used to obtain an efficient algorithm for insertion into an optimally balanced tree. The cost of this algorithm is $O(\log^3 n)$, which is a significant improvement compared to previous linear algorithms [7, 8, 9, 15]. The idea is to vary the value of ϵ in Theorem 3 in such a way that the height of the tree is optimal at any time.

Theorem 4 *An optimally balanced binary search tree may be maintained with a restructuring work of $O(\log^3 n)$ per insertion, provided that no deletions are made.*

Proof: It is sufficient to prove the theorem for insertions when $2^{k-1} \leq n+1 \leq 2^k$ for any positive integer k. The case when $n \geq 0$ follows immediately. We maintain the tree as in the proof of Theorem 1 with the difference that we use a varying value of the tuning parameter ϵ. Each time $n+1 = 2^k - 2^{k-i+1}$, $i = 2, 3, 4 \ldots k$, we rebuild the tree and change the value of ϵ to $\log \frac{2^i}{2^i-1}$. In other words, the value of ϵ is set to $\log \frac{4}{3}$ when the lowest level is empty, $\log \frac{8}{7}$ when the level is half-filled, $\log \frac{16}{15}$ when it is $\frac{3}{4}$-filled, and so forth. Suppose that $n+1 < 2^k - 2^{k-i}$. Then, the height is given by

$$\begin{aligned}
\text{height}(T) &\leq \lceil \log(n+1) + \epsilon \rceil \\
&\leq \left\lceil \log\left(2^k - 2^{k-i}\right) + \log \frac{2^i}{2^i - 1} \right\rceil \\
&= \left\lceil \log \frac{\left(2^k - 2^{k-i}\right) 2^i}{2^i - 1} \right\rceil \\
&= k. \qquad (15)
\end{aligned}$$

The fact that $k = \lceil \log(n+1) \rceil$ proves that the height is optimal. Let ϵ_i be the value of ϵ after $n+1 = 2^k - 2^{k-i+1}$. The total cost for filling the lowest level is given by

$$\begin{aligned}
&O\left(\sum_{i=2}^{k} (\text{number of insertions for this value of } i) \cdot \frac{\log^2 n}{\epsilon_i} \right) \\
&= O\left(\sum_{i=2}^{k} \frac{n}{2^i} \cdot \frac{\log^2 n}{\epsilon_i} \right) \\
&= O\left(n \log^2 n \cdot \sum_{i=2}^{k} \frac{1}{2^i \log \frac{2^i}{2^i-1}} \right) \\
&= O\left(n \log^2 n \cdot \sum_{i=2}^{k} 1 \right) \\
&= O\left(n \log^3 n \right). \qquad (16)
\end{aligned}$$

This cost is amortized over $\Theta(n)$ updates, which gives a cost of $O(\log^3 n)$ per update. Thus, the proof is completed. □

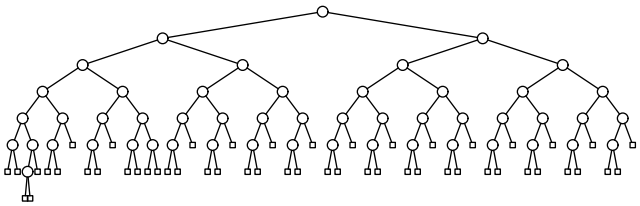

Figure 1: *A tree which requires rebuilding.*

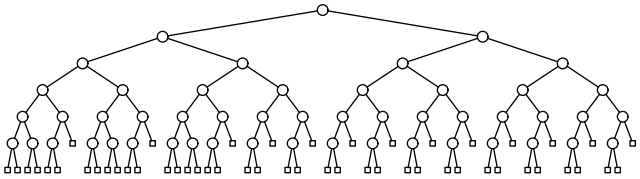

Figure 2: *The tree of Figure 1 after a partial rebuilding.*

Example: Figure 1 shows a tree which has become unbalanced by an insertion. The number of leaves in the tree is 53, the value of i is 3, thus $\epsilon = \log \frac{8}{7}$. A rebuilding is made at the lowest node v which satisfies

$$\text{height}(v) > \left\lceil \left(1 + \frac{\log\left(\frac{8}{7}\right)}{\log 53} \right) \log |v| \right\rceil . \tag{17}$$

Eq. (17) is satisfied by the left subtree of the root. A rebuilding is made at this node. The result of this rebuilding is illustrated in Figure 2. □

In the same way that we may handle insertions by bounding the longest path, we may handle deletions by bounding the shortest path.

Theorem 5 *An optimally balanced binary search tree may be maintained with a restructuring work of $O(\log^3 n)$ per deletion, provided that no insertions are made.*

Proof: The proof is analogous to the proof of Theorem 4. □

4 Dense Trees

Although the bounds for maintenance of trees with optimal and near-optimal height obtained in the preceding sections are low, there is still a gap between these bounds and the optimal of $O(\log n)$ per update. In this chapter we show how to overcome this gap by giving logarithmic algorithms for maintenance of trees of near-optimal height. In this way we obtain a near-optimal amortized solution to the dictionary problem.

The maintenance algorithms presented in this section differ from the algorithms in the previous sections in the way that we use a mixture of partial rebuilding and local restructuring. The basic idea is to maintain a tree of near-optimal height in which the leaves are replaced by

subtrees of size $\Theta(\log n)$. We refer to the tree as a *dense tree* consisting of a *topmost tree* and *leaf-subtrees*. Both the topmost tree and the leaf-subtrees are maintained by partial rebuilding. However, when a partial rebuilding is made in the topmost tree, the leaf-subtrees are not affected.

The advantage of dense trees is that most updates do not affect the topmost tree. Changes in the topmost tree occur only when a leaf-subtree is split, which results in an insertion into the topmost tree, or when two leaf-subtrees are joined with a subsequent deletion from the topmost tree. Provided that $\Theta(\log n)$ updates are required between two such operations, the amortized cost of updates in the topmost tree may be reduced by a factor of $\Theta(\log n)$. The price we pay for this improvement in time is that the number of incomplete levels is increased to 4. However, by making global rebuildings often and choosing the sizes of the leaf-subtrees carefully, the height may still be kept very low. The result is given in Theorem 6 below.

Theorem 6 *A binary search tree of height* $\lceil \log(n+1) + \epsilon \rceil$ *and at most 4 incomplete levels may be maintained at an amortized update cost of* $O\left(\frac{\log n}{\epsilon^2}\right)$, *where* $0 < \epsilon \leq 1/2$.

Proof: We use a dense tree, where the topmost tree T has a near-optimal height of $\lceil \log(|T|+1) + \epsilon/4 \rceil$ and the leaf-subtrees are perfectly balanced trees of size between $2^{\lceil \log\log n_0 \rceil}(1-\epsilon/4)$ and $2^{\lceil \log\log n_0 \rceil + 1} - 1$, where n_0 denotes the size of the tree the last time the entire tree was rebuilt.

To keep the value of n_0 up-to-date, we rebuild the entire tree completely after $\epsilon^2 n_0/32$ updates. At each global rebuilding we choose the size of each leaf-subtree to be $(2^{\lceil \log\log n_0 \rceil + 1} - 1)(1-\epsilon/4)$.

Updates are performed by adding or removing nodes in leaf-subtrees. When a leaf-subtree gets too small or too large, it is joined with another leaf-subtree or split. Only then an update is made in the topmost tree.

Theorem 1 implies that the amortized cost per update in the topmost tree is $O\left(\frac{\log^2 n}{\epsilon}\right)$. However, the way the sizes of the leaf-subtrees are restricted guarantees that each update in the topmost tree is preceded by $\Omega(\epsilon \log n)$ updates in the entire tree. Therefore, the cost of maintenance of the topmost tree is

$$\frac{O\left(\frac{\log^2 n}{\epsilon}\right)}{\Omega(\epsilon \log n)} = O\left(\frac{\log n}{\epsilon^2}\right). \tag{18}$$

The amortized cost of an update is the sum of the amortized cost of rebuilding the entire tree, the amortized cost of updating the topmost tree, and the amortized cost of updating the leaf-subtrees. This cost is

$$O(1) + O\left(\frac{\log n}{\epsilon^2}\right) + O(\log n) = O\left(\frac{\log n}{\epsilon^2}\right), \tag{19}$$

which proves the maintenance cost.

Next, we compute the maximum height of a dense tree. Due to the way the sizes of the leaf-subtrees are chosen, the size of the topmost tree immediately after a global rebuilding will be at most

$$\frac{n_0}{(2^{\lceil \log\log n_0 \rceil + 1} - 1)(1-\epsilon/4)}. \tag{20}$$

Every update in the topmost tree is preceded by at least $(2^{\lceil \log\log n_0 \rceil + 1} - 1)\epsilon/4$ updates in the entire tree. Thus, when the tree is rebuilt the next time at most

$$\frac{\epsilon^2 n_0/32}{(2^{\lceil \log\log n_0 \rceil + 1} - 1)\epsilon/4} = \frac{\epsilon n_0/8}{(2^{\lceil \log\log n_0 \rceil + 1} - 1)} \tag{21}$$

updates have been performed in the topmost tree. Thus, the size of the topmost tree is at most

$$\frac{n_0}{(2^{\lceil \log\log n_0\rceil+1}-1)(1-\epsilon/4)} + \frac{\epsilon n_0/8}{(2^{\lceil \log\log n_0\rceil+1}-1)} \leq \frac{(1+\epsilon/8)n_0}{(2^{\lceil \log\log n_0\rceil+1}-1)(1-\epsilon/4)}. \tag{22}$$

Since we make a global rebuilding after $\epsilon^2 n_0/2$ updates, we have

$$\begin{aligned} n &\geq (1-\epsilon^2/32)n_0 \\ n_0 &\leq \frac{n}{1-\epsilon^2/32}. \end{aligned} \tag{23}$$

Thus, from Eqs. (22) and (23) we get that the height of the topmost tree is at most

$$\begin{aligned} &\left\lceil \log\left(\frac{(1+\epsilon/8)n}{(2^{\lceil \log\log n_0\rceil+1}-1)(1-\epsilon/4)(1-\epsilon^2/32)}+1\right)+\frac{\epsilon}{4}\right\rceil \\ &= \left\lceil \log(n+2^{\lceil \log\log n_0\rceil+1}) + \log\left(1+\frac{\epsilon}{8}\right) - \log\left(1-\frac{\epsilon}{4}\right) - \log\left(1-\frac{\epsilon^2}{32}\right) - \lceil \log\log n_0\rceil - 1 + \frac{\epsilon}{4}\right\rceil \\ &\leq \lceil \log(n+1)+\epsilon\rceil - \lceil \log\log n_0\rceil - 1, \end{aligned} \tag{24}$$

provided that $\epsilon \leq 1/2$ and n is sufficiently large. The maximum size of a leaf-subtree is $2^{\lceil \log\log n_0\rceil+1}-1$; thus, its height is at most $\lceil \log\log n_0\rceil + 1$ since it is perfectly balanced. The maximum height of the entire tree is given by

$$\begin{aligned} \text{height}(T) &\leq \lceil \log(n+1)+\epsilon\rceil - \lceil \log\log n_0\rceil - 1 + \lceil \log\log n_0\rceil + 1 \\ &= \lceil \log(n+1)+\epsilon\rceil . \end{aligned} \tag{25}$$

In the same way that we restrict the height of the topmost tree, we may restrict its shortest path to have length at least

$$\lfloor \log(n+1)-\epsilon\rfloor - \lceil \log\log n_0\rceil - 1. \tag{26}$$

The shortest path of a leaf-subtree has a length of at least

$$\left\lfloor 2^{\lceil \log\log n_0\rceil}(1-\epsilon/4)\right\rfloor \geq \lceil \log\log n_0\rceil - 1. \tag{27}$$

By adding the two lengths together, we see that the length of the shortest path of the entire tree is at least

$$\lfloor \log(n+1)-\epsilon\rfloor - \lceil \log\log n_0\rceil - 1 + \lceil \log\log n_0\rceil - 1 = \lfloor \log(n+1)-\epsilon\rfloor - 2. \tag{28}$$

Thus, the maximum number of incomplete levels is given by

$$\lceil \log(n+1)+\epsilon\rceil - (\lfloor \log(n+1)-\epsilon\rfloor - 2) \leq 4. \tag{29}$$

The proof follows from Eqs. (19), (25), and (29). □

Thus, we have shown that an almost optimally balanced binary search tree may be efficiently maintained. An example of a dense tree is given in Figure 3.

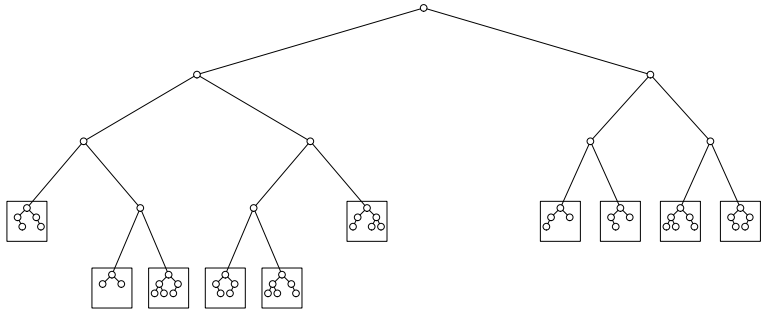

Figure 3: *A dense tree. The leaf-subtrees are marked by rectangles.*

5 Comments

We have presented improvements in the tradeoff between balance and maintenance cost for binary search trees.

Starting with one incomplete level, we have improved the amortized cost of insertions from $\Theta(n)$ to $O(\log^3 n)$. An analogous algorithm is also shown for deletions. These significant improvements make an optimally balanced tree much more attractive for the maintenance of a dynamic set than before.

Two incomplete levels may be maintained at an amortized cost of $O(\log^2 n)$ per update. This has also been shown by Lai and Wood [12] by introducing *level-layered trees* and a variant of weight-balanced trees. Compared to their results, the balance criteria used in this paper are simpler in the same sense that general balanced trees [3, 5] are simpler than other classes of trees, such as weight-balanced trees and AVL-trees.

Finally, four incomplete levels and a height of $\lceil \log(n+1) + \epsilon \rceil$ can be maintained with an optimal cost of $O(\log n)$ per update. This result allows us to improve the amortized upper bound on the dictionary problem.

Corollary 1 *For any value of ϵ, $\epsilon > 0$, there is a solution to the dictionary problem with the following costs:*

search: $\lceil \log(n+1) + \epsilon \rceil$ *comparisons in the worst case;*

update: $\Theta\left(\frac{\log n}{\epsilon^2}\right)$ *amortized.*

Thus, dense trees are superior (with respect to the search cost) to trees defined by weaker balance criteria [1, 6, 13, 14]. The only competitive tree presented so far is the ϵ-tree by Andersson [4, 5]. Compared to the ϵ-tree, however, the dense tree has simpler maintenance algorithms. A dense tree is also more balanced in the sense that we can guarantee at most four incomplete levels.

An interesting consequence of Theorem 1 is that we can store a binary search tree without any pointers in an array of size $O(N)$, where N is the maximum number of elements in the set, guaranteeing a search cost of at most $\lceil \log(n+1) + \epsilon \rceil$ and an amortized update cost of $O\left(\frac{\log^2 n}{\epsilon}\right)$. This is due to the fact that a partial rebuilding can be made in an implicitly stored tree, since

no pointer movements are required. Furthermore, the tree in Theorem 1 may be implemented without any balance information in the nodes in the same way as a general balanced tree.

Corollary 2 *For any value of $\epsilon > 0$, a set of n elements, $n \leq N$, may be maintained in an array of size $O(N)$ with the following costs:*

search: $\lceil \log(n+1) + \epsilon \rceil$ *comparisons in the worst case;*

update: $O\left(\frac{\log^2 n}{\epsilon}\right)$ *amortized.*

The ability to store the tree implicitly allows random access to a node at any level of the tree which makes *interpolation search* [10, 16] among the elements possible. If the elements are smoothly distributed, this will give an expected search cost of $O(\log \log n)$.

We are also able to improve the behaviour of a well-known sorting algorithm, namely *treesort* [2]. Using a dense tree we may sort n elements in $O(n \log n)$ time, using $n \log n + O(n)$ comparisons.

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] A. Andersson. Improving partial rebuilding by using simple balance criteria. In *Proceedings of the Workshop on Algorithms and Data Structures, WADS '89, Ottawa*, 1989.

[4] A. Andersson. Optimal bounds on the dictionary problem. In *Proc. Symposium on Optimal Algorithms, Varna*, 1989.

[5] A. Andersson. *Efficient Search Trees.* Ph. D. Thesis, Lund University, Sweden, 1990.

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

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

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

[9] T. E. Gerasch. An insertion algorithm for a minimal internal path length binary search tree. *Communications of the ACM*, 31:579–585, 1988.

[10] G. H. Gonnet. *Interpolation and Interpolation-Hash Searching.* Ph.D. Thesis, University of Waterloo, Canada, 1977.

[11] T. W. Lai and D. Wood. Updating approximately complete trees. Technical Report CS-89-57, University of Waterloo, 1989.

[12] T. W. Lai and D. Wood. Updating almost complete trees or one level makes all the difference. In *Proceedings of the 7th Symposium on Theoretical Aspects of Computer Science*, 1990.

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

[14] 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.

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

[16] A. C.-C. Yao and F. F. Yao. The complexity of searching an ordered random table. In *Proc. FOCS, Huston*, 1976.