RAIRO. INFORMATIQUE THÉORIQUE ET APPLICATIONS

TONY W. LAI
DERICK WOOD

Updating approximately complete trees

RAIRO. Informatique théorique et applications, tome 28, n° 5 (1994), p. 431-446

<http://www.numdam.org/item?id=ITA_1994__28_5_431_0>

Informatique théorique et Applications/Theoretical Informatics and Applications
(vol. 28, n° 5, 1994, p. 431 à 446)

UPDATING APPROXIMATELY COMPLETE TREES (*)

by Tony W. LAI ([1]) and Derick WOOD ([2])

Communicated by C. CHOFFRUT

Abstract. – *We define a k-incomplete binary search tree to be a tree in which any two external nodes are no more than k levels apart; we say that it is* **approximately complete**. *Whereas we show that 1-incomplete binary search trees have an amortized update cost of $\Theta(n)$, we demonstrate that 2-incomplete binary search trees have an amortized update cost of $O(\log^2 n)$. Thus, they are an attractive alternative for those situations that require fast retrieval (that is, $\log n + O(1)$ comparisons) and have few updates.*

Résumé. – *Nous définissons un arbre de recherche k-incomplet comme un arbre dans lequel les hauteurs de deux nœuds quelconques ne diffèrent pas plus de k : nous disons qu'il est* **approximativement complet**. *Alors que nous montrons que les arbres de recherche 1-incomplets ont un coût amorti de mise à jour en $\Theta(n)$, nous prouvons que les arbres binaires de recherche 2-incomplets ont un coût amorti de mise à jour en $O(\log^2 n)$. Ainsi, ils représentent une alternative attrayante dans les situations qui exigent une récupération des données rapide (c'est-à-dire $\log n + O(1)$ comparaisons) et peu de mises à jour.*

1. INTRODUCTION

Many kinds of binary search trees have been devised to guarantee that the worst-case search and update cost is $O(\log n)$; for example, red-black trees [6], height-balanced trees [1], and weight-balanced trees [11]. However, none of these data structures ensure that the worst-case search cost is $\log n + O(1)$. If searches are performed much more frequently than updates, it may be

(*) Received June 6, 1991, revised March 4, 1992, accepted July 28, 1994.

The work of the first author was supported under an NSERC Postgraduate Scholarship while he was studying at the University of Waterloo and that of the second was supported under a Natural Sciences and Engineering Research Council of Canada Grant No. A-5692 and under an Information Technology Research Centre Grant while he was at the University of Waterloo. A preliminary version of this paper appeared in STACS'90 [9].

([1]) Department of Computer Science, University of Toronto, Toronto, Ontario, Canada.

([2]) Department of Computer Science, University of Western Ontario, London, Ontario, Canada.

Informatique théorique et Applications/Theoretical Informatics and Applications
0988-3754/94/05/$ 4.00/

advantageous to employ a slower updating algorithm that ensures the search cost is $\log n + O(1)$.

Gerasch [5] devised an insertion algorithm for minimum internal path length binary search trees, or **1-incomplete trees**. The use of 1-incomplete trees ensures that searches make $\lceil \log (n+1) \rceil$ comparisons in the worst case, but the worst-case and amortized cost of his insertion algorithm is $\Theta(n)$. A deletion algorithm analogous to Gerasch's insertion algorithm can be devised, but the amortized cost of updating a 1-incomplete tree is still $\Theta(n)$.

We consider updating algorithms for binary search trees in which any two external nodes are no more that two levels apart; we call them **2-incomplete trees**. Such trees have two advantages: their worst-case search cost is $1 + \lceil \log (n+1) \rceil$, and their amortized update cost is $O(\log^2 n)$.

The schemes we propose are types of dynamization [16]. In particular, they are partial rebuilding schemes in the terminology of Overmars [12]; we have a balance criterion and we reconstruct subtrees that become unbalanced with respect to our criterion. Other partial rebuilding schemes have been devised by Overmars and van Leeuwen [13] using a weight balance criterion, and by Andersson [2] using a height balance criterion. However, both schemes ensure only that the height of a tree is $O(\log n)$ rather than $1+\lceil \log (n+1) \rceil$.

We propose a simple, novel technique for updating 2-incomplete trees called **k-layering**. One interesting aspect of this scheme is that it requires no additional balance informatin and that it needs to compute only subtree sizes. We also discuss a variant of k-layering called **level-layering** that achieves an amortized update cost of $O(\log^2 n)$. Lai and Wood [8] also achieved the same amortized bound for updating 2-incomplete trees by modifying Overmars and van Leeuwen's weight-balance scheme, but the resulting scheme is considerably more complicated than k-layering or level-layering.

Recently, Andersson and Lai devised algorihms for updating 4-incomplete trees in $O(\log n)$ amortized time [3] and algorithms for updating 2-incomplete trees in $O(\log n)$ amortized time [4, 7]. However, the algorithms are complex and are basically of only theoretical interest.

2. DEFINITIONS AND NOTATION

Recall that an (extended) binary tree consists of internal nodes with two children and external nodes with no children. We define the **level** of a node x of a tree to be the number of edges in the root-to-x path; the level of the root is 0. The **height,** $h(T)$, of a tree T is the maximum level of any

external node in T. A tree T is **complete** if all external nodes in T are on the same level.

In the following, we consider the class of **approximately complete trees**. A tree is **k-incomplete**, or in the set $AC[k]$, if every two external nodes are no more than k levels apart. Clearly $AC[k_1] \subset AC[k_2]$ if and only if $k_1 < k_2$. Observe that a 0-incomplete tree is a complete tree, and a 1-incomplete tree is a tree of minimum internal path length.

A tree T is **perfectly balanced** if for each node p in T, the numbers of nodes in p's left and right subtrees differ by at most one. We use $|T|$ to denote the number of nodes in T and use $n(T, r)$ to denote the size of the subtree rooted at r.

To discuss Gerasch's insertion algorithm, we introduce some more terminology. Intuitively, to obtain a **slot-extension** of a tree T, we add the minimum positive number of specially marked nodes necessary to obtain a complete tree containing T. More precisely, a slot-extension of T is a tree T' such that:

1. The nodes of T' are partitioned into **T-nodes** and **T-slots**. The T-nodes are the nodes of T and the T-slots are the additional, specially marked nodes;
2. T' is 0-incomplete or complete;
3. If T is 0-incomplete, then $h(T') = h(T) + 1$; otherwise, $h(T') = h(T)$; and
4. If all T-slots of T' are removed, then $T' = T$.

Observe that the slot-extension of T is unique. When discussing Gerasch's algorithm, we always refer to the slot-extension T' of T; we refer to T-nodes of T' as nodes of T and to T-slots of T' as slots of T.

3. 1-INCOMPLETE BINARY SEARCH TREES

Gerasch [5] showed that insertions can be performed in 1-incomplete binary search trees in $O(n)$ time in the worst case. For a binary tree of height h, his scheme maintains a window at level h if the tree is complete, otherwise at level $h - 1$; we refer to it as the **insertion window**. Gerasch's scheme uses one bit for each node to indicate whether the node has a slot descendant in the insertion window. To insert some value x, first search for x in the tree; then insert x by sliding data values in the subtree rooted at the lowest node on its root-to-frontier path that has a slot descendant in the insertion window. Finally, adjust the flags of the binary tree appropriately and move the insertion window if the tree has become complete.

We can obtain an analogous deletion algorithm in a straightforward manner. We consider only the case where a node without internal children is deleted, since we can easily transform the deletion of any internal node into a similar deletion. For a tree of height h, we position the **deletion window** at level $h-1$. We maintain an additional bit for each node to indicate whether the node has a node descendant in the deletion window, as shown in Figure 1. To delete a node with value x, first search for x in the tree; then delete x by sliding data values in the subtree rooted at the lowest node on the root-to-x path that has a node descendant in the deletion window. Finally, adjust the flags of the binary tree appropriately and move the deletion window if necessary. This deletion algorithm requires $O(n)$ time in the worst case.

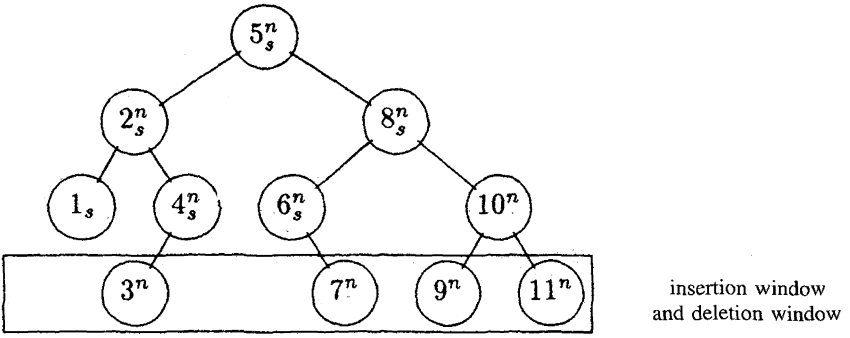

Figure 1. – Gerasch's scheme for updating 1-incomplete trees. The superscript n of a node label indicates that the node has a node descendant in the deletion window. The subscript s of a node label indicates that the node has a slot descendant in the insertion window.

These two algorithms give us a fully dynamic 1-incomplete binary search tree structure. Unfortunately, not only is the worst-case update cost of any 1-incomplete binary search tree $\Theta(n)$, but also the amortized update cost is $\Theta(n)$.

THEOREM 3.1: *The amortized update cost of a 1-incomplete binary search tree is* $\Theta(n)$.

Proof: If is sufficient to show that the amortized update cost is $\Omega(n)$, for infinitely many n. Consider a complete binary searh tree of size $n = 2^k - 1$, for some arbitrary integer $k \geq 1$. If we delete the maximum element and insert an element smaller than any element in the tree, then all elements in the tree must be moved to maintain the total order of the binary search tree. This pair of operations requires $\Theta(n)$ time and can be repeated indefinitely. Therefore, the amortized update cost is $\Omega(n)$. □

The simplicity of Gerasch's scheme is seductive, yet its cost is prohibitive in almost all situations. Ian Munro [10] suggested that by allowing more incomplete levels, the resulting elasticity could lead to polylogarithmic update costs. The key restriction is that the number of incomplete levels is fixed for all n. This is in contrast to the scheme of Overmars and van Leeuwen, who allow a number of incomplete levels proportional to $\log n$. In their scheme, the larger the value of n, the more incomplete levels there are. We partially validate Munro's conjecture for 2-incomplete trees, by presenting a novel scheme that supports updates in 2-incomplete trees in polylogarithmic amortized time.

4. 2-INCOMPLETE TREES AND k-LAYERING

We propose a class of schemes for updating 2-incomplete trees called **k-layering schemes**. In the k-layering scheme, where k is some positive integer, we allow subtree reconstructions on only k distinct levels of the tree. As an example of k-layering, we first describe the 3-layering scheme, before discussing the general scheme.

4.1. An example: the 3-layering scheme

To maintain a 2-incomplete tree, we maintain a two-level **update window** at the bottom of the tree and ensure that the tree is complete if the nodes in the window are excluded. Hence, we guarantee that the tree is 2-incomplete by ensuring that all insertions and deletions take place inside the update window. Note that we assume that only nodes without internal children are deleted, as before.

In the 3-layering scheme, we allow subtrees to be reconstructed only if they are rooted on levels 0, $h/3$, or $2h/3$, where h is the height of the tree. (In general, $h/3$ and $2h/3$ are not integers, so we actually allow reconstructions on levels 0, $\lfloor h/3 \rfloor$, and $\lfloor 2h/3 \rfloor$.) A schematic diagram is shown in Figure 2. Observe that a subtree rotted on level 0 is of size n, a subtree rooted on level $h/3$ is of size approximately $n^{2/3}$, and a subtree rooted on level $2h/3$ is of size approximately $n^{1/3}$.

We claim that updates in the 3-layering scheme have an amortized cost of $O(n^{1/3})$, assuming that we have an $O(n)$ worst-case time perfect rebalancing algorithm. To show this, we consider the amortized cost of reconstructions at levels $2h/3$, $h/3$, and 0. To simplify the analysis, we assume that the update window is positioned on levels $h-1$ and h, and that only insertions are performed.

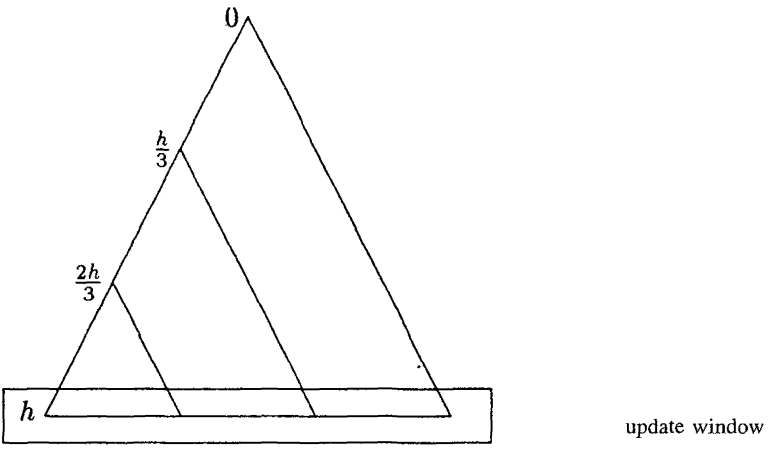

Figure 2. – A schematic diagram of 3-layering.

To compute the amortized cost, we first count the minimum number, $m(T)$, of insertions that we can perform in a subtree T before being forced to reconstruct a proper supertree of T. Observe that $m(T)$ is approximately the number of nodes of T that can lie in the update window. Because the update window is two-leveled and is placed at the bottom of T, the number of nodes of T that can lie in the update window is $\Theta(|T|)$. Therefore, $m(T) = \Omega(|T|)$.

The amortized cost of reconstructing a subtree rooted on level $2h/3$ is $O(n^{1/3})$, since the size of the subtree is $O(n^{1/3})$, and we may have to perform a reconstruction after each update. The cost of reconstructing a subtree rooted on level $h/3$ is $O(n^{2/3})$, but it is necessary only when we cannot update a subtree rooted on level $2h/3$. Since a subtree on level $2h/3$ allows $\approx n^{1/3}$ insertions before it is too large or $\approx n^{1/3}$ deletions before it is too small, $\Omega(n^{1/3})$ updates must have occurred previously, so the amortized cost is again $O(n^{1/3})$. Finally, the cost of reconstructing a subtree rooted on level 0 is $O(n)$, but it is required only when we cannot update a subtree T rooted on level $h/3$. $\Omega(|T|) = \Omega(n^{2/3})$ updates must have occured previously, which implies that the amortized cost of reconstructing the entire tree is $O(n^{1/3})$.

An update simultaneously affects subtrees rooted on levels $2h/3$, $h/3$, and 0, which implies that the total amortized cost is the sum of amortized costs at each level, which is $O(n^{1/3}) + O(n^{1/3}) + O(n^{1/3}) = O(n^{1/3})$. In general, for any fixed, positive integer k, the amortized cost of the k-layering scheme that we use is $O(kn^{1/k}) = O(n^{1/k})$.

From the above analysis, it appears that the cost of the k-layering scheme is $O(kn^{1/k})$, so that we have an amortized cost of $O(\log n)$ when we set k to be $O(\log n)$. However, our time bound is actually a factor of k too low. The problem is that if we reconstruct a proper supertree of some subtree S only when we are forced to, then we may have the pathological situation, depicted in Figure 3. Suppose we allow subtrees rooted on any level to be rebuilt, and we have a tree T of height h that has a complete left subtree T_L of height $h-1$ and a complete right subtree of height $h-3$. If we delete the largest element of T_L and insert an element less than any element in T, then the smallest subtree we must reconstruct is T_L. If we reconstruct T_L and not T, then we can perform the above pair of updates repeatedly to obtain an amortized update time of $\Omega(n)$. To avoid this difficulty, we introduce a balance criterion to ensure that $\Omega(|S|/k)$ updates have occurred since the last time a proper supertree of S was reconstructed. In Section 4.2, we discuss the k-layering scheme more formally to obtain an exact analysis.

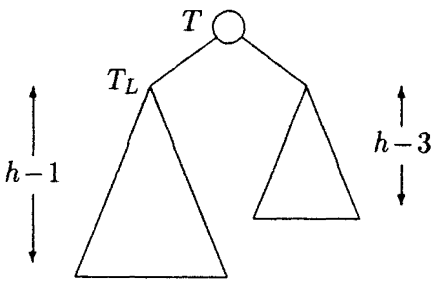

Figure 3. – A pathological situation.

4.2. The general scheme

In Section 4.1, we neglected the problem of choosing the location of the update window. For a binary tree of height h, we choose the levels of the update window as follows. If the size of the tree is no less than $2^{h-1}-1+\frac{2}{9}\cdot(2^{h-1}+2^h)$, then we position the window at levels $h-1$ and h; otherwise, we position it a levels $h-2$ and $h-1$. Suppose that the have placed the window at levels l and $l+1$. Since the size of the tree is at least 2^{h-1} and at most 2^h-1, it is possible to show that the size of the tree is no less than $2^l-1-\lceil\frac{2}{9}\cdot(2^l+2^{l+1})\rceil$ and no greater than $2^{l+2}-1-\lceil\frac{2}{9}\cdot(2^l+2^{l+1})\rceil$. This way, we ensure that the update window is neither too full nor too empty. We note that our update algorithms only depend on the fact that the size of the

tree is initially between $2^l - 1 + \lceil \varepsilon \cdot (2^l + 2^{l+1}) \rceil$ and $2^{l+2} - 1 - \lceil \varepsilon \cdot (2^l + 2^{l+1}) \rceil$, for some constant $\varepsilon > 0$. However, we choose ε to be as large as possible to minimize the amortized update time, and it can be proved that ε is at most $\frac{2}{9}$.

In the k-layering scheme, where k is some positive integer constant, we choose constants $\rho_1 = 0 < \rho_2 < \cdots < \rho_k = 2/9$ and functions $L_1, \cdots, L_k$ such that $l > L_1(l) > L_2(l) > \cdots > L_k(l) = 0$. Note that we require $l \geq k$. The functions $L_1, L_2, \cdots, L_k$ determine the levels on which we may reconstruct subtrees, and the constant ρ_i determines the balance criterion applied to subtrees on level L_i, for $i = 1, 2, \cdots, k$.

We define the **update window density** $\rho(T, r', l)$ of a subtree T' of T rooted at node r' to be the proportion of T''s nodes in the update window to the maximum possible number of nodes; more properly, if r' is on level l', then we define

$$\rho(T, r', l) = \frac{n(T, r') - (2^{l-l'} - 1)}{2^{l+2-l'} - 1 - (2^{l-l'} - 1)}.$$

For example, in Figure 4, we have $l = 2$, $n(T, y) = 6$, $n(T, z) = 3$, and $n(T, x) = 10$, which implies that $\rho(T, y, l) = \frac{5}{6}$, $\rho(T, z, l) = \frac{1}{3}$, and $\rho(T, x, l) = \frac{7}{12}$. Note that if the update window is empty and we perform a deletion above the window, then we get $\rho(T, r', l) < 0$. Similarly, if the update window is full and we perform an insertion below the window, then we get $\rho(T, r', l) > 1$.

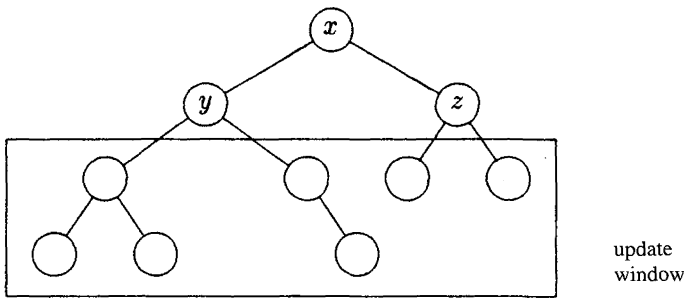

Figure 4.

Our imbalance criterion is: *For any subtree T' rooted on level $L_i(l)$, the update window density of T' is restricted to the interval $[\rho_i, 1 - \rho_i]$, for $i = 1, 2, \cdots k$.* This interval is smaller for subtrees rooted on levels higher in the tree, so that costly reconstruction high in the tree ensures that subsequent

updates are inexpensive. To delay reconstruction as much as possible, we rebuild a subtree T' only when we cannot handle an update in a proper subtree of T'. More precisely, we reconstruct a subtree if it satisfies the following imbalance criterion.

1. Any update destroys the balance of the subtree rooted at level $L_1(l)$ in which the update takes place.

2. A subtree rooted at node r on level $L_i(l)$, where $i > 1$, is imbalanced if there exists some unbalenced subtree rooted at a descendant r_{i-1} of r on level $L_{i-1}(l)$ such that $\rho(T, r_{i-1}, l) \notin [\rho_{i-1}, 1 - \rho_{i-1}]$, or, equivalently, $n(T, r_{i-1}) \notin [2^{l-L_{i-1}(l)} - 1 + \rho_{i-1} \cdot 3 \cdot 2^{l-L_{i-1}(l)}, 2^{l-L_{i-1}(l)+2} - 1 - \rho_{i-1} \cdot$

3. $2^{l-L_{i-1}(l)}]$.

We emphasize that this imbalance criterion is used solely to reduce the overall update time. *During updates, we directly ensure 2-incompleteness by examining path lenghts.*

The imbalance criterion yields straightforward insertion and deletion algorithms. As an example, suppose we want to insert some key x into T. We first insert x naively. Let r_i be the ancestor of depth L_i of x. We determine the highest node r_i such that $\rho(T, r_i, l) \leqq 1 - \rho_i$ and, for all $j < i$, $\rho(T, r_j, l) > 1 - \rho_j$, and then reconstruct the subtree rooted at r_i, since r_i is imbalanced. If we reconstruct the entire tree, then we reposition the update window.

The insertion and deletion algorithms are as follows. In the deletion algorithm, we assume that only nodes without internal children are deleted, as before. For brevity, in the following we refer to $L_i(l)$ as L_i, for $i = 1, 2, \cdots, k$.

insert (T, x);
  $i \leftarrow 1$;
  insert x;
  $r \leftarrow$ ancestor of x on level L_i in T;
  while $i < k$ and $n(T, r) > 2^{l-L_i+2} - 1 - \rho_i \cdot 3 \cdot 2^{l-L_i}$ do
    $i \leftarrow i + 1$;
    $r \leftarrow$ ancestor of r on level L_i
  end;
  reconstruct the subtree of T rooted at r;
  if $i = k$, then reposition update window
end insert

delete (T, x);
  $i \leftarrow 1$;
  $r \leftarrow$ ancestor of x on level L_i in T;
  delete x;
  while $i < k$ and $n(T, r) < 2^{l-L_i} - 1 + \rho_i \cdot 3 \cdot 2^{l-L_i}$ do
    $i \leftarrow i + 1$;
    $r \leftarrow$ ancestor of r on level L_i
  end;
  reconstruct the subtree of T rooted at r;
  if $i = k$, then reposition update window
end delete

To reconstruct a tree T, we use a perfect rebalancing algorithm, such as Stout and Warren's algorithm [15]. To determine $n(T, r)$, we can use brute force without increasing the running time of our algorithms by more than a constant factor, since the total time for counting the size of a subtree is proportional to the time taken to reconstruct the subtree.

In Section 5, we prove that the amortized update cost is $O(k^2 n^{1/k})$ in the k-layering scheme, for any positive integer k, if we choose $L_i = \lfloor (1 - i/k) l \rfloor$ and $\rho_i = \dfrac{2(i-1)}{9(l-1)}$, for $i = 1, 2, \ldots, k$. By choosing L_i and ρ_i to be evenly distributed in the intervals $[0, l)$ and $\left[0, \dfrac{2}{9}\right]$, respectively, we minimize the amortized cost of our update algorithms. A consequence of the above result is that if we allow reconstructions on every level and choose $L_i = l - i$ and $\rho_i = \dfrac{2(i-1)}{9(l-1)}$, for $i = 1, 2, \ldots, l$, then the amortized update cost is $O(\log^2 n)$. We call this latter scheme the **level-layering scheme**.

Observe that with the above choice of ρ_i and L_i, only the parameters k and l need be kept in the k-layering scheme, since ρ_i and L_i can be computed from i, k, and l in constant time. Similarly, only l need be kept in the l-layering scheme.

5. ANALYSIS

To perform an amortized analysis we first prove two technical lemmas. The first lemma bounds the number of nodes in the window in an updated subtree after reconstruction of one of its layered ancestors. The second

lemma bounds the number of updates that must have occurred since the last reconstruction of a layered subtree.

LEMMA 5.1: *If, after some update, the tree T_j rooted at node r_j at level L_j is the largest subtree of T reconstructed, where $j > 1$, then after the reconstruction, for any $k < j$ and any node r_k at level L_k,*

$$n(T, r_k) \in [2^{l-L_k} - 1 + \lfloor \rho_j \cdot 3 \cdot 2^{l-L_k} \rfloor, 2^{l-L_k+2} - 1 - \lfloor \rho_j \cdot 3 \cdot 2^{l-L_k} \rfloor].$$

Proof: Trivial. □

LEMMA 5.2: *If a subtree T_i of T rooted at node r_i on level L_i is reconstructed, where $i > 1$, then at least $(\rho_i - \rho_{i-1}) \cdot 3 \cdot 2^{l-L_{i-1}} - 1$ updates must have occurred since the last time a supertree of T_i was reconstructed.*

Proof: If T_i is reconstructed, then there must have been an update performed in some subtree T_{i-1} of T_i rooted at node r_{i-1} on level L_{i-1}. There are two cases to consider.

1. A deletion has caused the reconstruction of T_i.

Immediately before the reconstruction we know that

$$n(T, r_{i-1}) < 2^{l-L_{i-1}} - 1 + \rho_{i-1} \cdot 3 \cdot 2^{l-L_{i-1}}.$$

Lemma 5.1 implies that immediately after the last time some supertree T_j of T_i rooted at r_j on level L_j was reconstructed, we have

$$n(T, r_{i-1}) \geqq 2^{l-L_{i-1}} - 1 + \lfloor \rho_j \cdot 3 \cdot 2^{l-L_{i-1}} \rfloor.$$

Since $j \geqq i$, we must have performed at least

$$2^{l-L_{i-1}} - 1 + \lfloor \rho_i \cdot 3 \cdot 2^{l-L_{i-1}} \rfloor - [2^{l-l_{i-1}} - 1 + \rho_{i-1} \cdot 3 \cdot 2^{l-L_{i-1}}] > \rho_i \cdot 3 \cdot 2^{l-l_{i-1}} - \rho_{i-1} \cdot 3 \cdot 2^{l-L_{i-1}} - 1$$

updates.

2. An insertion has caused the reconstruction of T_i.

Immediately before the reconstruction we know that

$$n(T, r_{i-1}) > 2^{l-L_{i-1}+2} - 1 - \rho_{i-1} \cdot 3 \cdot 2^{l-L_{i-1}}.$$

Lemma 5.1 implies that immediately after the last time some supertree T_j of T_i rooted at r_j on level L_j was reconstructed, we have

$$n(T, r_{i-1}) \leqq 2^{l-L_{i-1}+2} - 1 - \lfloor \rho_j \cdot 3 \cdot 2^{l-L_{i-1}} \rfloor.$$

Since $j \geqq i$, we must have performed at least

$$2^{l-L_{i-1}+2} - 1 + \rho_{i-1} \cdot 3 \cdot 2^{l-L_{i-1}} - [2^{l-l_{i-1}+2} - 1 - \lfloor \rho_i \cdot 3 \cdot 2^{l-L_{i-1}} \rfloor]$$
$$> \rho_i \cdot 3.2^{l-l_{i-1}} - \rho_{i-1} \cdot 3 \cdot 2^{l-L_{i-1}} - 1$$

updates. □

THEOREM 5.3: *For any $1 \leqq k \leqq l$, the amortized update cost of the k-layering scheme is $O(k^2\, n^{1/k})$, for an appropriate choice of parameters ρ_1, $\rho_2, \cdots, \rho_k$, $L_1, L_2, \cdots, L_k$. In particular, if reconstructions are allowed on every level and $L_i = l - i$ and $\rho_i = \dfrac{2(i-1)}{9(l-1)}$, for $i = 1, 2, \cdots, l$, then the amortized update cost is $O(\log^2 n)$.*

Proof: Suppose that we have a reconstruction algorithm that requires cn time in the worst case. We choose $\rho_i = \dfrac{2}{9} \cdot \dfrac{i-1}{k-1}$ and $L_i = \lfloor \dfrac{k-i}{k} \cdot l \rfloor$, for $i = 1, 2, \cdots, k$. Let A be the amortized update cost. For any i, the time to reconstruct a subtree rooted on level L_i is at most $c \cdot 2^{l-L_i+2}$. Also, for any $i > 1$, Lemma 5.2 implies that at least $\max(1, (\rho_i - \rho_{i-1}) \cdot 3 \cdot 2^{l-L_{i-1}} - 1)$ updates must be performed between reconstructions of a subtree rooted on level L_i. Moreover, observe that

$$\max(1, (\rho_i - \rho_{i-1}) \cdot 3 \cdot 2^{l-L_{i-1}} - 1) \geqq \frac{3}{2}(\rho_i - \rho_{i-1}) \cdot 2^{l-L_{i-1}}$$

Since an update affects k layers simultaneously, the amortized cost is

$$A \leqq c \cdot 2^{l-L_1+2} + \sum_{i=2}^{k} \frac{c \cdot 2^{l-L_i+2}}{\frac{3}{2}(\rho_{i-1} - \rho_i) \cdot 2^{l-L_{i-1}}}$$

$$= 4c \cdot 2^{l-L_1} + \sum_{i=2}^{k} \frac{4c \cdot 2^{L_{i-1}-L_i}}{\frac{3}{2}(\rho_{i-1} - \rho_i)}$$

$$\leqq 4c \cdot 2^{l/k+1} + \sum_{i=2}^{k} \frac{4c \cdot 2^{l/k+1}}{\frac{3}{2} \cdot \frac{2}{9} \cdot \frac{1}{k-1}}$$

$$= 8c \cdot 2^{l/k} + \sum_{i=2}^{k} [24c \cdot (k-1) \cdot 2^{l/k}]$$

$$= 8c \cdot 2^{l/k} + 24c \cdot (k-1)^2 \cdot 2^{l/k}.$$

Since $l \leqq \log n$,

$$A = O(n^{1/k} + k^2\, n^{1/k}) = O(k^2\, n^{1/k}).$$

When $k = l$, we have $A = O(\log^2 n)$. □

The level-layering scheme is attractive because of its simplicity – it is simpler than Gerasch's scheme, yet it performs better.

6. AN EMPIRICAL COMPARISON OF LEVEL-LAYERED AND AVL TREES

To assess the practicality of level-layered trees, we conducted simulations of level-layered trees and AVL trees on a MIPS M/2000 running RISC/os version 4.51. We coded the algorithms in *C* and compiled them using *gcc*, version 1.37.1. Since we were interested only in the update time, we were interested only in the update time, we performed 50,000 insertions into an empty tree. We performed two sets of simulations: we inserted keys in sorted order and in random order. To implement AVL trees, we used the update algorithms of Reingold and Hansen [14]. To implement level-layered trees, we used Stout and Warren's perfect rebalancing algorithm [15], and we counted subtree sizes by converting subtrees into paths of right children.

The results of our simulations are shown in Tables I and II. We measured the average CPU time per insertion (in milliseconds) and the average number of pointer dereferences per insertion or, in other words, the average number of pointers followed per insertion. Note that the CPU time for random insertions includes the time required to generate pseudorandom numbers.

TABLE I

A comparison of sequential insertions.

Number of insertions	AVL trees		Level-layered trees	
	CPU time	Dereferences	CPU time	Dereferences
10,000	0.016	50.7	0.23	1179.4
20,000	0.017	52.7	0.28	1437.9
30,000	0.018	53.8	0.33	1639.1
40,000	0.018	54.7	0.36	1760.2
50,000	0.020	55.4	0.42	1907.5

TABLE II

A comparison of random insertions.

Number of insertions	AVL trees		Level-layered trees	
	CPU time	Dereferences	CPU time	Dereferences
10,000	0.020	47.4	0.041	131.2
20,000	0.023	49.4	0.050	134.6
30,000	0.025	50.6	0.068	165.9
40,000	0.027	51.4	0.062	138.6
50,000	0.028	52.1	0.060	133.2

Our simulations show that, for random insertions, level-layered trees take at most three times more CPU time and four times more pointer dereferences than AVL trees, and, for sequential insertions, level-layered trees take at most 21 times more CPU time and 35 times more pointer dereferences than AVL trees. Note that the CPU time for random insertions in an AVL tree is significantly more than the CPU time for sequential insertions, whereas the number of pointer dereferences is less. We speculate that this phenomenon is caused by a poor hit ratio in the processor cache. Also, note that the average number of pointer dereferences for 30,000 random insertions in a lavel-layered tree is higher than the number of dereferences for 50,000 random insertions; this anomaly is probably caused by a global rebuilding between 20,000 and 30,000 insertions. While the cost of insertions in level-layered trees is much more than the cost of insertions in AVL trees for sequential insertions, the likelihood of sequential insertions is arguably low, since a different algorithm would be employed if the insertions were known in advance to be sequential. Thus, level-layered trees may be a practical alternative to other balanced trees if the updates are infrequent and real-time update performance is not critical.

7. EXTERNAL-SEARCH TREES

One problem with the layering scheme is that only nodes with external children can be deleted, making deletions difficult for tree structures that are not binary search trees, such as k-d trees. One solution to this problem is to use **external-search trees** (that is, we associate keys with external nodes, and separating vlues with internal nodes), rather than use internal-search trees.

In the layering scheme for external-search trees, we maintain a two-level update window as before. If we place the window on levels l and $l+1$, then we ensure that all external nodes are on levels $l-1$, l, or $l+1$, which guarantees that the tree is 2-incomplete. We redefine $n(T, r)$ to be the number of external nodes in the subtree of tree T rooted at r. For a binary tree of height h, we choose the level of the update window as follows. If level h (that is, the lowest level) contains no fewer than $\frac{1}{2} \cdot 2^h$ external nodes, then we position the window at levels h and $h+1$; otherwise, we position it at levels $h-1$ and h. Assuming that we have placed the window at levels l and $l+1$, we ensure that the number of external nodes in the window is in the interval $\left[\frac{1}{4} \cdot 2^{l+1}, \frac{3}{4} \cdot 2^{l+1}\right]$.

In the k-layering scheme, where k is constant, we choose constants $\rho_1 = 0 < \rho_2 < \cdots < \rho_k = 1/4$ and functions $l > L_1(l) > L_2(l) > \cdots > L_k(l) = 0$. Our imbalance criterion is now: *A subtree rooted at node r on level $L_{i-1}(l)$, where $i > 1$, is imbalanced if there exists some imbalanced subtree rooted at a descendant r_{i-1} of r on level $L_i(l)$ such that* $n(T, r) < \rho_{i-1} \cdot 2^{l-L_{i-1}(l)+1}$ *or* $n(T, r) > (1 - \rho_{i-1}) \cdot 2^{l-L_{i-1}(l)+1}$. An update to a subtree rooted on level $L_1(l)$ unbalances that subtree, as before. The analysis of this modified layering scheme is similar to the analysis for internal-search trees, and the amortized update cost is $O(k^2 n^{1/k})$ when we choose $L_i(l) = \lfloor (1 - i/k) l \rfloor$ and $\rho_i = \dfrac{i-1}{4(k-1)}$, for $i = 1, 2, \ldots, k$. In the level-layering scheme, we choose the number of layers to be l, and, for $i = 1, 2, \ldots, l$, we choose $\rho_i = \dfrac{1}{4} \cdot \dfrac{i-1}{l-1}$ and $L_i(l) = l - i$. The resulting amortized update cost is $O(\log^2 n)$, as before.

For tree structures other than binary search trees, we may need to use a perfect rebalancing algorithm that requires $\omega(n)$ time in the worst case. The level-layering scheme yields polylogarithmic amortized update cost when used with an $O(\log^c n)$ worst-case time perfect rebalancing algorithm, when c is constant. In particular, the k-layering scheme can be shown to take $O(k^2 n^{1/k} \log^c n)$ amortized time to update 2-incomplete trees, and the level-layering scheme can be shown to take $O(\log^{2+c} n)$ amortized time.

REFERENCES

1. G. M. Adel'son-Vel'skii and E. M. Landis, An algorithm for the organization of information, *Sov. Math. Dokl.*, 1962 *3*, pp. 1259-1262.
2. A. Andersson, Improving partial rebuilding by using simple balance criteria, In *Proceedings of the 1989 Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science,* 1989, *447*, Springer-Verlag, pp. 393-402.
3. A. Andersson, *Efficient Search Trees,* PhD thesis, Lund University, Sweden, 1990.
4. A. Andersson and T. W. Lai, Comparison-efficient and write-optimal searching and sorting, In *Proceedings of the 2nd Annual International Symposium on Algorithms, Lecture Notes in Computer Science,* 1991, *557,* Springer-Verlag, pp. 273-282.
5. T. E. Gerasch, An insertion algorithm for a minimal internal path length binary search tree, *Communications of the ACM,* 1988, *31*, pp. 579-585.
6. L. C. Guibas and R. Sedgewick A dichromatic framework for balanced trees, In *Proceedings of the 19th Annual IEEE Sympossium on Foundations of Computer Science,* 1978, pp. 8-21.
7. T. W. H. Lai, *Efficient Maintenance of Binary Search Trees,* Ph. D thesis, University of Waterloo, 1990.
8. T. W. H. Lai and D. Wood, *Updating approximately complete trees,* Technical Report CS-89-57, University of Waterloo, 1989.

9. T. W. H. 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, Lecture Notes in Computer Science,* 1990, *415*, Springer-Verlag, pp. 188-194.
10. J. I. MUNRO, Private communication.
11. J. NIEVERGELT and E. M. REINGOLD, Binary search trees of bounded balance, *SIAM Journal on Computing,* 1973, *2*, pp. 33-43.
12. M. H. OVERMARS, The Design of Dynamic Data Structures, volume 156 of *Lecture Notes in Computer Science,* 1983, Springer-Verlag.
13. M. H. OVERMARS and J. VAN LEEUWEN, Dynamic multi-dimensional datta structures based on quad- and k-d trees, *Acta Informatica,* 1982, *17*, pp. 267-285.
14. E. M. REINGOLD and W. J. HANSEN, *Data Structures in Pascal,* Little, Brown and Company, 1986.
15. Q. F. STOUT and B. L. WARREN, Tree rebalancing in optimal time and space, *Communications of the ACM,* 1986, *29,* pp. 902-908.
16. J. VAN LEEUWEN and D. WOOD, Dynamization of decomposable searching problem, *Information Processing Letters*, 1980, *10,* pp. 51-56.