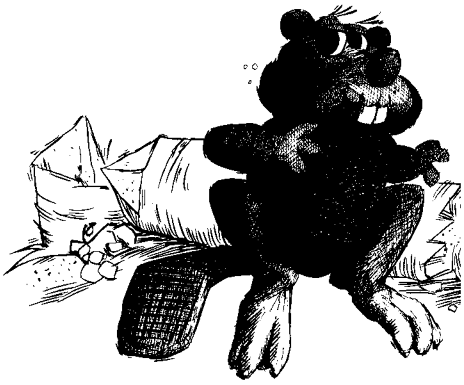


Updating Approximately Complete Trees

Tony W.H. Lai
and
Derick Wood

Data Structuring Group
Research Report CS-89-57

November, 1989

Updating Approximately Complete Trees[*]

Tony W. Lai[†] Derick Wood[†]

Abstract

We define a k-complete 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.* While 1-complete binary search trees have an amortized update cost of $\Theta(n)$, we demonstrate that 2-complete 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. We also prove that the amortized cost of updating an $f(n)$-complete binary search tree for any $f(n) = o(\log n)$ is $O(\log^2 n/f(n))$, and we explore approximately complete external search trees.

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 [5], height-balanced trees [1], and weight-balanced trees [8]. 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 advantageous to employ a slower updating algorithm that ensures the search cost is $\log n + O(1)$.

Gerasch [4] devised an insertion algorithm for minimum internal path length binary search trees, or *1-complete trees.* The use of 1-complete trees ensures that searches require $\lceil\log(n + 1)\rceil$ 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 algorithm for 1-complete trees can be devised, but the amortized cost of updating a 1-complete tree is still $\Theta(n)$.

*The work of the first author was supported under an NSERC Postgraduate Scholarship and that of the second under a Natural Sciences and Engineering Research Council of Canada Grant No. A-5692 and under an Information Technology Research Centre Grant. A preliminary version of this paper will appear in STACS '90. [6]

[†]Data Structuring Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.

We consider updating algorithms for binary search trees in which any two external nodes are no more that two levels apart; we refer to these trees as *2-complete.* 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 [12]. In particular, they are partial rebuilding schemes in the terminology of Overmars [9]: 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 [10] using a weight balance criterion, and by Andersson [3] 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-complete trees called *k-layering.* One interesting aspect of this scheme is that it requires no additional balance information 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 $O(\log^2 n)$.

By modifying the scheme of Overmars and van Leeuwen, we obtain another partial rebuilding scheme that updates $(f(n) + O(1))$-complete trees with an amortized cost of $O(log^2 n/f(n))$, for any $f(n) = o(\log n)$. In particular, this modified scheme also allows 2-complete trees to be updated with an amortized cost of $O(\log^2 n)$.

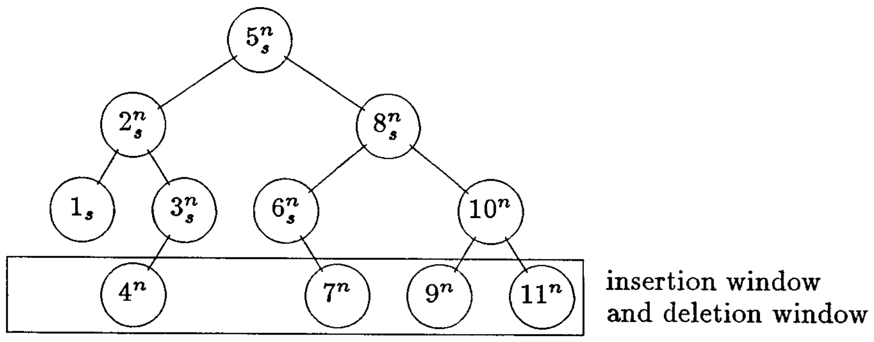
2 Definitions and notation

We define the *depth* of a node x of a tree to be the number of edges in the root-to-x path; the depth of the root is 0. The *nodes on level l* of a tree are the set of nodes of depth l in the tree. The *height*, $h(T)$, of a tree T is the number of internal nodes in the longest root-to-leaf path in the tree, so that a tree of one external node has height 0. A tree is *complete* if all external nodes are of the same depth.

In the following, we consider the class of *approximately complete trees.* A tree is *k-complete*, 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-complete tree is a complete tree, and a 1-complete tree is a tree of minimum internal path length.

A tree T is *perfectly balanced* if for each node p in T, the number 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 number of descendents of a node r in a tree T, including r itself.

To discuss Gerasch's insertion algorithm, we introduce some more terminology. A *slot-extension of T* is a tree T' such that:



^{n} indicates that the node has a node descendant in the deletion window.
$_s$ indicates that the node has a slot descendant in the insertion window.

Figure 1: Gerasch's scheme for updating 1-complete trees.

1. the nodes of T' are partitioned into T-*slots* and T-*nodes*;
2. T' is 0-complete;
3. if T is 0-complete, 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-complete binary search trees

Gerasch [4] showed that insertions can be performed in 1-complete 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.

An analogous deletion algorithm is a straightforward modification of this approach. We assume that only leaves are deleted; to delete an internal node x, we find the successor or predecessor y of x, move y's contents into x, and delete y instead. 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 leaf 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 its root-to-frontier 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.

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

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

Proof: It is sufficient to show that the amortized cost is $\Omega(n)$. Consider any complete binary tree of size no less than n. 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 of 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 [7] 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-complete trees, in the following sections, by presenting a novel scheme that allows updates in 2-complete trees in polylogarithmic amortized time, and by modifying Overmars and van Leeuwen's scheme to restrict the number of incomplete levels to a constant.

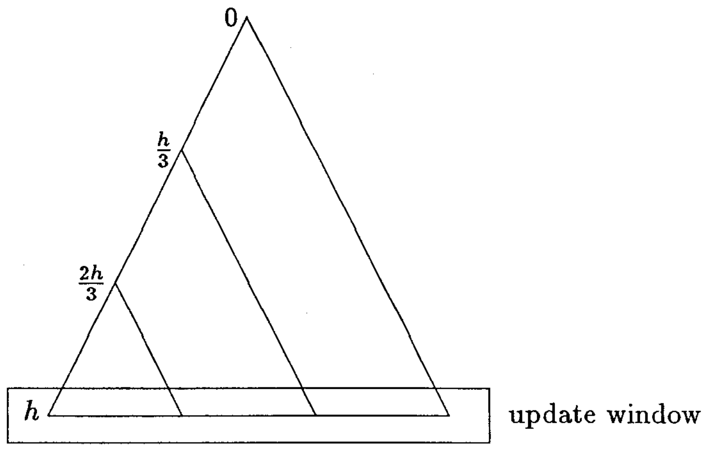


Figure 2: A schematic diagram of 3-layering.

4 2-complete trees and k-layering

4.1 An example: the 3-layering scheme

We propose a class of schemes for updating 2-complete 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 discuss the 3-layering scheme.

To maintain a 2-complete 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-complete by ensuring that all insertions and deletions take place inside the update window. Note that we assume that only leaves are deleted; to delete an internal node x, we find the successor or predecessor y of x, move y's contents into x, and delete y instead.

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. A schematic diagram is shown in Figure 2. Observe that a subtree rooted 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.

To compute the amortized cost, we first count the minimum number, $m(T)$, of insertions 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, it contains some subset of the leaves of T and their parents, which implies that 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 this 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 this is required only when we cannot update a subtree T rooted on level $h/3$. $\Omega(|T|) = \Omega(n^{2/3})$ updates must have occurred 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 positive integer k, the amortized cost of the k-layering scheme can be shown to be $O(kn^{1/k}) = O(n^{1/k})$.

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 $O(\log n)$ if we set $k = O(\log n)$. 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 T only when forced to, then there are pathological sequences of very high cost. To avoid this difficulty, we introduce a balance criterion to ensure that $\Omega(|T|/k)$ updates have occurred since the last time a proper supertree of T was reconstructed. In the next section, we discuss the k-layering scheme more formally to obtain an exact analysis.

4.2 The general scheme

In the previous section, 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 window as follows. If level $h-1$, that is, the lowest level, contains no less than $2^{h-1} - 1 + \frac{2}{9} \cdot (2^{h-1} + 2^h)$ nodes, then we position the window at levels $h-1$ and h; otherwise, we position it at levels $h-2$ and $h-1$. Suppose that we have placed the window at levels l and $l+1$. This way, we ensure that the number of nodes in the window is between $2^l - 1 + \lceil \frac{2}{9} \cdot (2^l + 2^{l+1}) \rceil$ and $2^{l+2} - 1 - \lceil \frac{2}{9} \cdot (2^l + 2^{l+1}) \rceil$, inclusive.

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 $L_1, \ldots, L_k$ of positive integers such that $l > L_1(l) > L_2(l) > \cdots > L_k(l) = 0$. Note that we require $l \geq k$. The functions $L_1, L_2, \ldots, L_k$ determine the levels at which we may reconstruct subtrees, and the constant ρ_i determines the balance criterion applied to subtrees on level L_i, for $i = 1, 2, \ldots, k$.

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; actually, if r' is on level l', then we define

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

Observe that we allow $\rho(T, r', l) < 0$ and $\rho(T, r', l) > 1$.

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 $i = 1, 2, \ldots, k$. This interval is smaller for subtrees rooted on higher levels, so that costly reconstruction high in the tree ensures that subsequent updates are inexpensive. More precisely, we reconstruct a subtree if it satisfies the following imbalance criterion.

1. Any update imbalances 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 imbalanced subtree rooted at a descendant r_{i-1} of r on level $L_{i-1}(l)$ such that $n(T, r_{i-1}) \notin [2^{l-L_{i-1}(l)} - 1 + \rho_{i-1} \cdot 3 \cdot 2^{l-L_{i-1}(l)}, 2^{l-L_{i-1}(l)+2} - 1 - \rho_{i-1} \cdot 3 \cdot 2^{l-L_{i-1}(l)}]$.

This 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 $\rho(T, r_i, l) \leq 1 - \rho_i$ and for all $j > i$, $\rho(T, r_j, l) > 1 - \rho_j$, and then reconstruct the subtree rooted at r_i. If we reconstruct the entire tree, then we reposition the update window.

The insertion and deletion algorithms are as follows. As in the discussion of the modified OvL scheme, we assume that only leaves are deleted, since the deletion of an internal node can be transformed into a deletion of leaf. For brevity, in the following we refer to $L_i(l)$ as L_i, for $i = 1, 2, \ldots, k$.

```
insert($T, x$)
    $i \leftarrow 1$
    insert $x$
    $r \leftarrow$ ancestor of $x$ on level $L_i$ in $T$
    while $i < k$ and $n(T,r) > 2^{l-L_i+2} - 1 - \rho_i \cdot 3 \cdot 2^{l-L_i}$
        $i \leftarrow i + 1$
        $r \leftarrow$ ancestor of $r$ on level $L_i$
    end
    reconstruct 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 $n(T,r) < 2^{l-L_i} - 1 + \rho_i \cdot 3 \cdot 2^{l-L_i}$
        $i \leftarrow i + 1$
        $r \leftarrow$ ancestor of $r$ on level $L_i$
    end
    reconstruct 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 [11]. In the next two sections, 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 $\rho_i = \frac{2(i-1)}{9(k-1)}$, for $i = 1, 2, \ldots, k$; and the amortized update cost is $O(\log^2 n)$ in the l-layering scheme if we choose $L_i = l - i$ and $\rho_i = \frac{2(i-1)}{9(l-1)}$, for $i = 1, 2, \ldots, l$.

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 bounds the number of nodes in the window in an updated subtree after reconstruction of one of its layered ancestors. The second 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_i rooted at node r_i at level L_i is the largest subtree of T reconstructed, where $i > 1$, then after the reconstruction, for any $j < i$ and any node r_j at level L_j, $n(T, r_j) \in [2^{l-L_j} - 1 + \lfloor \rho_i \cdot 3 \cdot 2^{l-L_j} \rfloor, 2^{l-L_j+2} - 1 - \lfloor \rho_i \cdot 3 \cdot 2^{l-L_j} \rfloor]$.*

Proof: Immediately after the reconstruction of T_i, we know that

$$n(T, r_i) \in \left[2^{l-L_i} - 1 + \rho_i \cdot 3 \cdot 2^{l-L_i}, 2^{l-L_i+2} - 1 - \rho_i \cdot 3 \cdot 2^{l-L_i}\right].$$

Since T_i is perfectly rebalanced, we know that

$$\begin{aligned} n(T, r_j) &\in \left[\lfloor 2^{l-L_j} - 1 + \rho_i \cdot 3 \cdot 2^{l-L_j} \rfloor, \lceil 2^{l-L_j+2} - 1 - \rho_i \cdot 3 \cdot 2^{l-L_j} \rceil\right] \\ &= \left[2^{l-L_j} - 1 + \lfloor \rho_i \cdot 3 \cdot 2^{l-L_j} \rfloor, 2^{l-L_j+2} - 1 - \lfloor \rho_i \cdot 3 \cdot 2^{l-L_j} \rfloor\right]. \end{aligned}$$

□

Lemma 5.2 *If a subtree T_i of T rooted at node r_i at 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} at 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

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

   which implies that

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

   Immediately after the last time some supertree of T_j of T_i rooted at r_j on level L_j was reconstructed, we know that

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

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

   $$\begin{aligned} &2^{l-L_{i-1}} - 1 + \lfloor \rho_i \cdot 3 \cdot 2^{l-L_{i-1}} \rfloor - \left[2^{l-L_{i-1}} - 2 + \lceil \rho_{i-1} \cdot 3 \cdot 2^{l-L_{i-1}} \rceil\right] \\ &\quad > \rho_i \cdot 3 \cdot 2^{l-L_{i-1}} - \rho_{i-1} \cdot 3 \cdot 2^{l-L_{i-1}} - 1 \end{aligned}$$

   updates.

2. A insertion has caused the reconstruction of T_i.

   Immediately before the reconstruction we know that

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

   which implies that

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

Immediately after the last time some supertree of T_j of T_i rooted at r_j on level L_j was reconstructed, we know that

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

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

$$\begin{aligned} &2^{l-L_{i-1}+2} - \lceil \rho_{i-1} \cdot 3 \cdot 2^{l-L_{i-1}} \rceil - \left[2^{l-L_{i-1}+2} - 1 - \lfloor \rho_i \cdot 3 \cdot 2^{l-L_{i-1}} \rfloor\right] \\ &\quad > \rho_i \cdot 3 \cdot 2^{l-L_{i-1}} - \rho_{i-1} \cdot 3 \cdot 2^{l-L_{i-1}} - 1 \end{aligned}$$

updates.

□

Theorem 5.3 *The amortized update cost of the k-layering scheme, for constant k, is $O(k^2 n^{1/k})$, for an appropriate choice of parameters ρ_1, ρ_2, …, ρ_k, L_1, L_2, …, L_k.*

Proof: Suppose that we have a reconstruction algorithm that requires cn time in the worst case. We choose $\rho_i = \frac{2}{9} \cdot \frac{i-1}{k-1}$ and $L_i = \lfloor \frac{k-i}{k} \cdot l \rfloor$, for $i = 1, 2, \ldots, k$. Let A be the amortized update cost.

$$\begin{aligned} A &< c \cdot 2^{l-L_1+2} + \sum_{i=2}^{k} \frac{c \cdot 2^{l-L_i+2}}{(\rho_{i-1} - \rho_i) \cdot 3 \cdot 2^{l-L_{i-1}} - 1} \\ &= c \cdot 2^{l-L_1+2} + \sum_{i=2}^{k} \frac{4c \cdot 2^{L_{i-1}-L_i}}{3(\rho_{i-1} - \rho_i) - 2^{L_{i-1}-l}} \\ &= O\left(2^{l/k} + \sum_{i=2}^{k} \frac{2^{l/k}}{\frac{2}{3} \cdot \frac{1}{k-1} - 2^{-(i-1)l/k}}\right). \end{aligned}$$

For sufficiently large l, we know that $2^{-(i-1)l/k} \ll \frac{2}{3} \cdot \frac{1}{k-1}$. Thus

$$A = O\left(2^{l/k} + \sum_{i=2}^{k} (k-1) \cdot 2^{l/k}\right).$$

Since $l \leq \log n$,

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

Observe that the amortized update cost increases by at most a constant factor if we determine $n(T, r)$ in time proportional to $n(T, r)$. □

This scheme is attractive because of its simplicity — it is simpler than Gerasch's scheme, yet it performs better. However, it still has super-polylogarithmic behavior. In the next section, we give a simple modification that achieves polylogarithmic behavior.

6 Level-layering

The update cost of our layering scheme can be greatly improved if we use all levels as layers; we refer to this as *l-layering* or *level-layering.* We choose the number of layers to be the number of levels l and, for $i = 1, 2, \ldots, l$, we choose $\rho_i = \frac{2}{9} \cdot \frac{i-1}{l-1}$ and $L_i = l - i$. To perform an amortized analysis, we first prove a lemma showing that at least two updates must occur between reconstructions of any subtree rooted at level L_i, for all $i > \lceil \log l \rceil + 3$.

Lemma 6.1 *If $i > \lceil \log l \rceil + 3$, then*

$$3(\rho_i - \rho_{i-1}) \cdot 2^{l-L_{i-1}} - 1 \geq 2.$$

Proof: It is sufficient to prove that

$$3(\rho_i - \rho_{i-1}) \cdot 2^{l-L_{i-1}} - 1 \geq 13/3$$

or
$$(\rho_i - \rho_{i-1}) \cdot 2^{l-L_{i-1}} \geq 16/9.$$

Recall that $l \leq \log n$ and $i \geq \lceil \log l \rceil + 4$; therefore,

$$\begin{aligned} \frac{2}{9} \cdot \frac{1}{l-1} \cdot 2^{l-(l-i+1)} &\geq \frac{2}{9} \cdot \frac{1}{l} \cdot 2^{i-1} \\ &\geq \frac{2}{9} \cdot \frac{1}{l} \cdot 2^{\lceil \log l \rceil + 3} \\ &\geq \frac{2}{9} \cdot \frac{1}{l} \cdot 8l \\ &\geq \frac{16}{9}. \end{aligned}$$

□

Theorem 6.2 *The amortized update cost of the level-layering scheme is $O(\log^2 n)$.*

Proof: Let A be the amortized update cost. Observe that the above lemma implies that for all $i > \lceil \log l \rceil + 3$,

$$3(\rho_i - \rho_{i-1})2^{l-L_{i-1}} - 1 \geq 2(\rho_i - \rho_{i-1})2^{l-L_{i-1}}.$$

Also, any subtree rooted on level L_i can be reconstructed at most once per update, for $i \leq \lceil \log l \rceil + 3$.

Let $\mathcal{L} = \lceil \log l \rceil + 3$. Recall that $l \leq \log n$; hence:

$$\begin{aligned} A &< c \sum_{i=1}^{\mathcal{L}} 2^{l-L_i+2} + \sum_{i=\mathcal{L}+1}^{l} \frac{c \cdot 2^{l-L_i+2}}{(\rho_i - \rho_{i-1}) \cdot 3 \cdot 2^{l-L_{i-1}} - 1} \\ &\leq c \cdot 2^{l-L_{\mathcal{L}}+3} + \sum_{i=\mathcal{L}+1}^{l} \frac{c \cdot 2^{l-L_i+2}}{2(\rho_i - \rho_{i-1}) \cdot 2^{l-L_{i-1}}} \end{aligned}$$

$$
\begin{aligned}
&\le c \cdot 2^{l-L_{\mathcal{L}}+3} + \sum_{i=\mathcal{L}+1}^{l} \frac{2c \cdot 2^{L_{i-1}-L_i}}{\rho_i - \rho_{i-1}} \\
&= c \cdot 2^{l-(l-\lceil \log l \rceil -3)+3} + \sum_{i=\mathcal{L}+1}^{l} \frac{2c \cdot 2}{2/9 \cdot \frac{1}{l-1}} \\
&< 128c \log n + \sum_{i=\mathcal{L}+1}^{l} 18c \log n \\
&= 128c \log n + (l - \lceil \log l \rceil - 3) \cdot 18c \log n \\
&= O(\log^2 n).
\end{aligned}
$$

□

Therefore, the use of the $O(\log n)$-layering scheme leads to an amortized update cost of $O(\log^2 n)$ for 2-complete trees.

7 Overmars and van Leeuwen's scheme

Overmars and van Leeuwen [10] devised a method for updating a binary search tree that has an amortized cost of $O(\frac{1}{\delta}\log_{2-\delta} n)$, for any constant $0 < \delta < 1$, while ensuring that the height of the tree is at most $\log_{2-\delta} n$. To bound the height, they place a weight balance constraint on each subtree: if a subtree T contains n nodes, then no proper subtree of T has more than $\lfloor n/(2-\delta) \rfloor$ nodes. Observe that their scheme requires that weight-balance information be kept at each node. To achieve the claimed time bound, Overmars and van Leeuwen require an $O(n)$ time perfect rebalancing algorithm.

Their update scheme works as follows. To insert a node x, insert it naively and perfectly rebalance the subtree rooted at the highest ancestor of x whose balance criterion has been violated. The deletion of a leaf is analogous, and the deletion of an internal node can always be transformed into a deletion of a leaf, as in the case of 1-complete trees. Stout and Warren [11] devised an algorithm to perfectly rebalance a binary search tree of n nodes in $O(n)$ time in the worst case.

In the Overmars-van Leeuwen scheme, hereafter referred to as the OvL scheme, the weight-balance constraint ensures that the height of the tree is at most $\log_{2-\delta} n + O(1)$, and the number of complete levels is at least $\log_{\frac{1}{1-1/(2-\delta)}} n + O(1) = \log_{\frac{2-\delta}{1-\delta}} n + O(1)$. Hence the number of incomplete levels is no more than $\log_{2-\delta} n - \log_{\frac{2-\delta}{1-\delta}} n + O(1) = g(\delta) \log n + O(1)$, where $g(\delta) = 1/\log(2-\delta) - 1/[\log(2-\delta) - \log(1-\delta)]$. Therefore, with an appropriate choice of δ, we can ensure that a tree has at most $f(n)$ incomplete levels for any function $f(n) = \Theta(\log n)$, since for any $c > 0$, there exists $\delta > 0$ such that $g(\delta) < c$. Unfortunately, the OvL scheme is insufficient to restrict the number of incomplete levels to $o(\log n)$.

We modify the OvL scheme by making δ dynamic. We do this by perfectly rebalancing the entire tree after every $n/2$ updates, and by choosing a new value of δ every time the entire tree is rebalanced. This increases the amortized update cost by at most a constant. To restrict the number of incomplete levels to $f(n) + O(1)$ some for some function $f(n) = o(\log n)$, we choose $\delta(n) = 2 - 2^{\frac{\log n}{\log n + f(n)/3}}$. Observe that for sufficiently large n, the height and number of complete levels change by at most a constant after $n/2$ updates.

We first prove that this choice of δ restricts the number of incomplete levels to no more than $f(n) + O(1)$. We then prove a bound on the resulting amortized update cost.

Lemma 7.1 *Choosing $\delta(n) = 2 - 2^{\frac{\log n}{\log n + f(n)/3}}$ implies that the height of a tree in this class is at most* $\log n + f(n)/3 + O(1)$.

Proof: The height of a tree is at most $\log_{2-\delta} n + O(1) = \log n / \log(2-\delta) + O(1)$. Observe that

$$2 - \delta = 2^{\frac{\log n}{\log n + f(n)/3}},$$

$$\log(2-\delta) = \frac{\log n}{\log n + f(n)/3},$$

and, hence

$$\frac{\log n}{\log(2-\delta)} = \log n + \frac{f(n)}{3}.$$

Thus the height of the tree is at most $\log n + f(n)/3 + O(1)$. □

Lemma 7.2 *If $f(n) \leq \log n$, then for sufficiently large n, the number of complete levels of a tree in this class is at least* $\log n - f(n)/2 + O(1)$.

Proof: The number of complete levels is at least $\log_{\frac{2-\delta}{1-\delta}} n + O(1)$. Observe that

$$\delta = 2 - 2^{\frac{\log n}{\log n + f(n)/3}} = 2 - 2^{1 - \frac{f(n)/3}{\log n + f(n)/3}}.$$

We claim that $-\log(1-\delta) \leq \frac{5}{6} \cdot \frac{f(n)}{\log n + f(n)/3}$. Let $z = \frac{f(n)/3}{\log n + f(n)/3}$; it is straightforward to show that $-\log(2^{1-z} - 1) \leq \frac{5}{2} z$ by observing that

$$-\log(2^{1-0} - 1) = 0 \leq \frac{5}{2} \cdot 0$$

and that since $0 \leq z \leq \frac{1}{4}$,

$$\frac{d}{dz}\left[-\log(2^{1-z} - 1)\right] = \frac{2^{1-z}}{2^{1-z} - 1}$$

$$
\begin{aligned}
&= \frac{1}{1-2^{z-1}} \\
&< \frac{d}{dz}\left(\frac{5}{2}z\right) \\
&= \frac{5}{2}.
\end{aligned}
$$

Therefore, we have

$$
\begin{aligned}
\log_{\frac{2-\delta}{1-\delta}} n &= \frac{\log n}{\log(2-\delta)-\log(1-\delta)} \\
&= \frac{\log n}{\frac{\log n}{\log n+f(n)/3} - \log(1-\delta)} \\
&\geq \frac{\log n}{\frac{\log n}{\log n+f(n)/3} + \frac{5/6\cdot f(n)}{\log n+f(n)/3}} \\
&= \frac{\log n[\log n + f(n)/3]}{\log n + \frac{5}{6}f(n)} \\
&= \frac{\log n[\log n + \frac{5}{6}f(n)] - \frac{f(n)}{2}\log n}{\log n + \frac{5}{6}f(n)} \\
&= \log n - \frac{\frac{f(n)}{2}\log n}{\log n + \frac{5}{6}f(n)} \\
&\geq \log n - f(n)/2.
\end{aligned}
$$

Thus, the number of complete levels is at least $\log n - f(n)/2 + O(1)$. □

Corollary 7.3 *The number of incomplete levels at most $f(n) + O(1)$.*

Lemma 7.4 *The amortized update cost is $O(\log^2 n/f(n))$.*

Proof: The amortized update cost of the OvL scheme is $O(\frac{1}{\delta}\log_{2-\delta} n)$, where $\delta = 2 - 2^{\frac{\log n}{\log n+f(n)/3}}$. We claim that $\delta \leq 2\ln 2 \cdot \frac{f(n)/3}{\log n+f(n)/3}$. Let $z = \frac{f(n)/3}{\log n+f(n)/3}$; it is straightforward to show that $2 - 2^{1-z} \leq (2\ln 2)z$ by observing that

$$2 - 2^{1-0} = 0 \leq (2\ln 2)\cdot 0$$

and that

$$\frac{d}{dz}\left[2 - 2^{1-z}\right] = 2^{1-z}\ln 2 \leq \frac{d}{dz}(2\ln 2)z = 2\ln 2.$$

The update cost is therefore $O(\frac{[\log n+f(n)/3]^2}{f(n)}) = O(\log^2 n/f(n))$. □

Theorem 7.5 *For any $f(n) = o(\log n)$, the amortized cost of the modified Overmars and van Leeuwen scheme for updating an $(f(n) + O(1))$-complete tree is $O(\log^2 n/f(n))$.*

Observe that because the number of incomplete levels is $f(n) + O(1)$, we cannot trivially obtain 2-complete trees by substituting $f(n) = 1$ in the above choice of δ. In the next section, we derive the choice of δ necessary to obtain 2-complete trees.

8 2-complete binary search trees and the OvL scheme

We now attempt to update a 2-complete binary search tree using the modified OvL scheme. To prove that we actually obtain a 2-complete tree, we first count the height and number of complete levels more carefully, since we can no longer ignore constant terms. We then determine choices of $\delta(n)$ that yield 2-complete trees. We finally analyze the amortized cost resulting from our choice of $\delta(n)$.

Lemma 8.1 *The height of a tree is at most* $1 + \lfloor \log_{2-\delta} n \rfloor$.

Proof: Immediate from the fact that no subtree of a tree of n nodes has more than $\lfloor \frac{1}{2-\delta} n \rfloor$ nodes. □

Lemma 8.2 *The number of complete levels of a tree is at least* $1 + \lceil \log_{\frac{2-\delta}{1-\delta}} n - \log_{\frac{2-\delta}{1-\delta}} (4-\delta) \rceil$.

Proof: Observe that any subtree of a tree of n nodes must contain at least $n - \lfloor \frac{1}{2-\delta} n \rfloor - 1 = \lceil \frac{1-\delta}{2-\delta} n \rceil - 1$ nodes. We determine a lower bound k on the depth of the highest node that has at most one child; the number of complete levels is then at least $1 + k$. If $\delta < \frac{1}{2}$, then the root of any subtree of 3 or more nodes must have two children. Thus it suffices to find the highest root of a subtree of 2 or fewer nodes. To obtain a lower bound, we underestimate the smallest number d_i of nodes in any subtree rooted at level i with some function e_i, and compute the highest level where $e_i \leq 2$.

Let $a = \frac{1-\delta}{2-\delta}$. Trivially $d_0 = n = e_0$, and by induction, for $i > 0$,

$$d_i \geq a d_{i-1} - 1 \geq a^i n - \sum_{j=0}^{i-1} a^j = e_i.$$

We solve $e_k \leq 2$ for k.

$$\begin{aligned} e_k &= a^k n - a^{k-1} - \cdots - a - 1 \\ &= a^k n - \frac{1-a^k}{1-a} \\ &\leq 2. \end{aligned}$$

This implies that

$$a^k n \le 2 + \frac{1}{1-a} - \frac{a^k}{1-a},$$

$$a^k(n + \frac{1}{1-a}) \le 2 + \frac{1}{1-a},$$

and

$$k + \log_a(n + \frac{1}{1-a}) \ge \log_a(2 + \frac{1}{1-a}).$$

We finally obtain

$$\begin{aligned} k &\ge \log_a(2 + \frac{1}{1-a}) - \log_a(n + \frac{1}{1-a}) \\ &\ge \log_{1/a}(n + \frac{1}{1-a}) - \log_{1/a}(2 + \frac{1}{1-a}) \\ &\ge \log_{1/a} n - \log_{1/a}(2 + \frac{1}{1-a}) \\ &\ge \log_{\frac{2-\delta}{1-\delta}} n - \log_{\frac{2-\delta}{1-\delta}}(4-\delta). \end{aligned}$$

Thus, the number of complete levels is at least $1 + \lceil \log_{\frac{2-\delta}{1-\delta}} n - \log_{\frac{2-\delta}{1-\delta}}(4-\delta) \rceil$.
□

Lemma 8.3 *If $\delta(n) = \frac{c}{\log n + c}$, where $0 < c < \ln 2$ is some constant, then there exists n_0 such that for any $n > n_0$, the tree is 2-complete.*

Proof: To ensure that the tree is 2-complete, it is sufficient to ensure that

$$\lfloor \log_{2-\delta} n \rfloor - \lceil \log_{\frac{2-\delta}{1-\delta}} n - \log_{\frac{2-\delta}{1-\delta}}(4-\delta) \rceil \le 2$$

or

$$\log_{2-\delta} n - \log_{\frac{2-\delta}{1-\delta}} n + \log_{\frac{2-\delta}{1-\delta}}(4-\delta) < 3.$$

Observe that $\log_{\frac{2-\delta}{1-\delta}}(4-\delta) < 2$ and $\log(\frac{2-\delta}{1-\delta}) > 1$. Now,

$$\frac{\log n}{\log(2-\delta)} - \frac{\log n}{\log(2-\delta) - \log(1-\delta)} < 3 - \log_{\frac{2-\delta}{1-\delta}}(4-\delta)$$

$$\frac{[\log(2-\delta) - \log(1-\delta)]\log n - \log(2-\delta)\log n}{\log(2-\delta)[\log(2-\delta) - \log(1-\delta)]} < 3 - \log_{\frac{2-\delta}{1-\delta}}(4-\delta)$$

$$\frac{-\log(1-\delta)\log n}{\log(2-\delta)[\log(2-\delta) - \log(1-\delta)]} < 3 - \log_{\frac{2-\delta}{1-\delta}}(4-\delta)$$

Hence,

$$-\log(1-\delta)\log n < \left[3 - \log_{\frac{2-\delta}{1-\delta}}(4-\delta)\right] \log\left(\frac{2-\delta}{1-\delta}\right) \log(2-\delta).$$

For any $0 < \alpha < 1$, there exists n_0 such that $\log(2-\delta) > \alpha$ for any $n > n_0$. It is straightforward to show that $-\log(1-\delta) \le \frac{1}{\ln 2} \cdot \frac{\delta}{1-\delta}$ for $\delta \ge 0$ by evaluating $-\log(1-\delta)$ and $\frac{1}{\ln 2}\cdot\frac{\delta}{1-\delta}$ at $\delta = 0$ and comparing their derivatives with respect to δ for all $0 < \delta < 1$. Hence, it is sufficient to choose δ such that

$$\frac{\delta}{1-\delta}\log n < [3 - \log_{\frac{2-\delta}{1-\delta}}(4-\delta)]\log\left(\frac{2-\delta}{1-\delta}\right)\cdot\alpha\ln 2.$$

We choose $\delta = \frac{c}{\log n + c}$, which implies $\frac{\delta}{1-\delta} = \frac{c}{\log n}$ and $\frac{\delta}{1-\delta}\log n = c$. Therefore, for any $c < \ln 2$, for sufficiently large n, we know that

$$\log_{2-\delta} n - \log_{\frac{2-\delta}{1-\delta}} n + \log_{\frac{2-\delta}{1-\delta}}(4-\delta) < 3.$$

Note that the above proof assumes that $\delta = \frac{c}{\log n + c}$ for the current value of n. In general, we may perform up to $n/2$ updates since the last time δ was chosen. For the above proof to hold, it is sufficient to ensure $\frac{\delta}{1-\delta}\log(\frac{3}{2}n) < \ln 2$, or $c < \frac{\ln 2}{1+\log(3/2)/\log n}$; if $c < \ln 2$, this is clearly satisfied for arbitrarily large values of n. □

Lemma 8.4 *The amortized update cost is $O(\log^2 n)$.*

Proof: The amortized update cost is $O(\frac{1}{\delta}\log_{2-\delta} n) = O(\frac{\log n + c}{c}\cdot\log n) = O(\log^2 n)$. □

Theorem 8.5 *The modified Overmars and van Leeuwen scheme for updating a 2-complete tree has an amortized cost of $O(\log^2 n)$.*

One disadvantage of the OvL scheme is that it requires weight-balance information to perform updates efficiently, so that we need extra storage for n integers, or $O(n\log n)$ bits. We can eliminate the extra storage requirement with the following modification due to Andersson [2]. Let n_0 be the number of elements immediately after the previous reconstruction of the entire tree. We reconstruct the entire tree after ϵn_0 updates, where $\epsilon < 1/3$ is some positive constant. We maintain two new invariants: the height of the tree cannot be more than $H = 1 + \lfloor\log_{2-\delta}(n_0 + \epsilon n_0)\rfloor$, and the number of complete levels must be at least $C = 1 + \lceil\log_{\frac{2-\delta}{1-\delta}}(n_0 - \epsilon n_0) - \log_{\frac{2-\delta}{1-\delta}}(4-\delta)\rceil$.

To insert a node x, we insert it naively; if x is below level $H-1$, then we proceed up the root-to-x path, reconstructing each subtree rooted at a weight-imbalanced node until the we reduce the height of the tree to H. The deletion of a leaf x is analogous, except that if x is above level C, then we perform reconstructions to increase the number of complete levels to C. Observe that we may count subtree sizes on the fly without increasing the update cost by more than a constant factor.

The height and complete level invariants imply that the tree is 2-complete for arbitrarily large values of n if we choose $c < (1-3\epsilon)\cdot\ln 2$ and $\delta = \frac{c}{\log n + c}$.

The amortized update cost of this scheme is $O(\log^2 n)$, since only weight-imbalanced subtrees are reconstructed, and a violation of the height or complete level invariants implies a violation of the weight balance invariant for some node.

9 External search trees

One problem with both the layering scheme and the modified OvL scheme is that only leaves can be deleted, making deletions difficult for some tree structures other than binary search trees, such as k-d trees. One solution to this problem is to use external search trees. In this section, we discuss the modifications required to use external search trees in both partial rebuilding schemes.

9.1 The layering scheme

The layering scheme for external search trees is identical to the scheme for internal search trees, except that we insert or delete two nodes (one internal and one external) for each update. Note that in the level-layering scheme, at least four updates must occur between reconstructions of any subtree rooted on level L_i, for any $i > \lceil \log l \rceil + 3$, so that inserting or deleting two nodes at once is valid. The amortized update time increases by a factor of 2, so we are able to update 2-complete trees with an amortized cost of $O(\log^2 n)$, as in the case of internal search trees.

We can simplify the layering scheme somewhat by redefining $n(T, r)$ to be the number of external nodes in the tree T that are descendants of 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 $h-1$ and h. Assuming that we have placed the window at levels l and $l+1$, this ensures that the number of external nodes in the window is in the interval $[\frac{1}{4} \cdot 2^{l+1}, \frac{3}{4} \cdot 2^{l+1}]$.

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: 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 $L_1(l)$ imbalances that subtree, as before. The analysis of this modified layering scheme is similar to the case of internal search trees, and the amortized update cost is $O(k^2 n^{1/k})$ if we choose $L_i(l) = \lfloor (1 - i/k) l \rfloor$ and $\rho_i = \frac{i-1}{4(k-1)}$ for $i = 1, 2, \ldots, k$. In our level-layering scheme, we choose the number of layers to be l, and, for $i = 1, 2, \ldots, l$, we choose $\rho_i = \frac{1}{4} \cdot \frac{i-1}{l-1}$ and $L_i(l) = l - i$. The resulting amortized update cost is $O(\log^2 n)$, as before.

9.2 The OvL scheme

In the OvL scheme for external search trees, we ensure that if any tree has n external nodes, then no proper subtree has more than $1 + \lfloor \frac{n-1}{2-\delta} \rfloor$ external nodes. As before, this results in a tree of height $\log_{2-\delta} n + O(1)$, and the number of complete internal levels is $\log_{\frac{2-\delta}{1-\delta}} n + O(1)$. Thus, for any $f(n) = o(\log n)$, if we choose $\delta = \frac{f(n) \ln 2}{\log n + f(n)}$, then we obtain a tree with at most $f(n) + O(1)$ incomplete levels; the amortized update cost is $O(\log^2 n / f(n))$, as in the case of internal search trees.

In particular, we can ensure that the height of the tree is at most $1 + \lfloor \log_{2-\delta}(n-1) \rfloor$, and the number of complete levels no less than $1 + \lceil \log_{\frac{2-\delta}{1-\delta}}(n-1) - \log_{\frac{2-\delta}{1-\delta}}(4-\delta) \rceil$. As in the case of internal search trees, if we choose $\delta = \frac{c}{\log n + c}$, where $0 < c < \ln 2$, then the number of incomplete levels is no more than 2. Thus we are able to update 2-complete external search trees in $O(\log^2 n)$ time.

9.3 Superlinear reconstruction algorithms

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. Both the modified OvL scheme and the level-layering scheme yield polylogarithmic amortized update costs when used with a $O(n \log^c n)$ worst-case time perfect rebalancing algorithm, if c is constant. In particular, it is straightforward to show that the modified OvL scheme allows $(f(n) + O(1))$-complete trees to be updated in $O(\log^{2+c} n / f(n))$ amortized time for any function $f(n) = o(\log n)$ and 2-complete trees to be updated in $O(\log^{2+c} n)$ time. The k-layering scheme can be shown to require $O(k^2 n^{1/k} \log^c n)$ amortized time to update 2-complete trees, and the level-layering scheme can be shown to require $O(\log^{2+c} n)$ amortized time.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Sov. Math. Dokl.*, 3:1259–1262, 1962.

[2] A. Andersson. Private communication.

[3] A. Andersson. Improving partial rebuilding by using simple balance criteria. In *Proceedings of the 1989 Workshop on Algorithms and Data Structures*, pages 393–402. Springer-Verlag, 1989.

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

[5] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.

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

[7] J. I. Munro. Private communication.

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

[9] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983.

[10] M. H. Overmars and J. van Leeuwen. Dynamic multi-dimensional data structures based on quad- and k-d trees. *Acta Informatica*, 17:267–285, 1982.

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

[12] J. van Leeuwen and D. Wood. Dynamization of decomposable searching problems. *Information Processing Letters*, 10:51–56, 1980.