PREPRINT. In Proc. 2nd ann. International Symposium on Algorithms, pages 273-282. Springer Verlag, 1991.

Comparison-efficient and Write-optimal Searching and Sorting

Arne Andersson
Lund University
Lund, Sweden

Tony W. Lai
NTT Communication Science Labs
Kyoto, Japan

Abstract

We consider the problem of updating a binary search tree in $O(\log n)$ amortized time while using as few comparisons as possible. We show that a tree of height $\lceil \log(n+1) + 1/\sqrt{\log(n+1)} \rceil$ can be maintained in $O(\log n)$ amortized time such that the difference between the longest and shortest paths from the root to an external node is at most 2.

We also study the problem of sorting and searching in the *slow write* model of computation, where we have a constant size cache of fast memory and a large amount of memory with a much slower writing time than reading time. In such a model, it is important to sort using only $\Theta(n)$ writes into the slower memory. We say that such algorithms are *write optimal*, and we introduce a $O(n \log n)$ time, write-optimal sorting algorithm that requires only $n \log n + O(n)$ comparisons in the worst case. No previous sorting algorithm that performs $n \log n + o(n \log n)$ comparisons in the worst case had previously been shown to be write optimal.

The above results are based on a class of trees called k-stratum trees, which can be viewed as a generalization of stratified search trees.

1 Introduction

In this abstract, we introduce the *slow write* model of computation. In this model, we have a constant size cache of fast memory and a large amount of memory with a much slower writing time than reading time. We justify the

model by noting that certain types of memory such as flash memories and EEPROMs have much slower writing times than reading times.

We consider the problem of efficiently maintaining a comparison-based dictionary, while minimizing the number of comparisons and writes per update. In particular, we want updates and searches to require only $\log n + O(1)$ comparisons in the worst case; we say that a dictionary with this property is *comparison-efficient*. In addition, we want updates to require only a constant number of writes; a dictionary with this property is *write-optimal*.

In the area of comparison-efficient bounds, we show that a binary search tree of height at most $\lceil \log(n+1) + \epsilon(n) \rceil$ can be maintained in $O(\log n)$ amortized time, such that $\lim_{n\to\infty} \epsilon(n) = 0$. No such upper bound on the number of comparisons has been previously demonstrated. The exact value of $\epsilon(n)$ we obtain is $1/\sqrt{\log(n+1)}$.

In the area of write-optimal bounds, we show that a binary search tree of height at most $\lceil \log(n+1) + \epsilon \rceil$ can be maintained in $O(\log n)$ amortized time, such that the amortized number of writes is $O(1/\epsilon^3)$. Thus, by inserting n elements into such a structure, we obtain an $O(n \log n)$ time sorting algorithm that uses only $n \log n + O(n)$ comparisons and $\Theta(n)$ writes. Although there are both write-optimal sorting algorithms [5, 6, 7] and sorting algorithms with an optimal number of comparisons (with respect to the leading term) [3, 8], no previous solution has been shown to combine both properties.

The presented results are based on a new data structure called a k-stratum tree. They can be viewed as a generalization of stratified search trees [7].

Before proceeding further, we define some more terms. We define the length of a path in a tree to be the number of internal nodes on the path. We define the *height*, height(T), of a tree T to be the length of the longest path from the root to an external node. We define the *number of complete levels*, short(T), of a tree T to be the length of the shortest path from the root to an external node. The *incompleteness* of T is given by $\text{height}(T) - \text{short}(T)$. A tree T is *k-incomplete* if $\text{height}(T) - \text{short}(T) \leq k$. Thus, a complete tree is 0-incomplete, and a tree of minimal internal path length is 1-incomplete. The *weight*, $|T|$, of a tree T is the number of external nodes of T. A tree T is *perfectly balanced* if T is an external node, or T's subtrees are perfectly balanced and their weights differ by at most 1.

2 Comparison efficiency

To maintain an extremely well-balanced tree, we use the following basic ideas.

- Although the amortized cost of maintaining a binary search tree of minimal internal path length (1-incomplete tree) can be shown to be $\Omega(n)$ in general, it is possible to obtain a polylogarithmic amortized cost when the size of the tree is not close to a power of 2.
- By keeping two layers of 1-incomplete trees on top of each other, we achieve a 2-incomplete tree. By careful maintenance algorithms consisting of partial rebuilding within both layers, 3-way splitting and merging of subtrees, and occasional rebuilding of the entire tree, we guarantee that the sizes of the trees in both layers are favorable all the time. We also make the frequency of splitting, merging, and global rebuilding low enough to achieve a total amortized cost of $O(\log n)$ per update.

Note that we also achieve an incompleteness of 2; this incompleteness can be shown to be the smallest maintainable in $O(\log n)$ time in general, so we obtain matching upper and lower bounds on the incompleteness of a binary search tree.

Since we use 1-incomplete trees in two layers or strata, our tree is called a *2-stratum* tree. We give a formal definition of a 2-stratum tree below.

Definition 1 *Given two positive integers H_1 and H_2, a* 2-stratum tree *consists of a* topmost tree *whose external nodes are replaced by* leaf subtrees*, such that the following properties hold.*

1. *Let T_1 be the topmost tree. Then,* $\mathrm{short}(T_1) \geq H_1 - 1$ *and* $\mathrm{height}(T_1) \leq H_1$.
2. *Let T_2 be a leaf subtree. Then,* $\mathrm{short}(T_2) \geq H_2 - 1$ *and* $\mathrm{height}(T_2) \leq H_2$.
3. $H_1 + H_2 \leq \left\lceil \log(n+1) + \frac{1}{\sqrt{\log(n+1)}} \right\rceil$*, where n is the number of elements in the tree.*

We say that the topmost tree is the apex *of the tree and is in stratum 1, and the leaf subtrees are in stratum 2.*

The maintenance algorithms ensure a low height by utilizing the values of H_1 and H_2, which are changed only during global rebuildings of the tree. During most updates, H_1 and H_2 remain unchanged. When the tree is globally rebuilt, we make sure that $H_1 + H_2 \leq \left\lceil \log(n+1) + \frac{\epsilon}{2} \right\rceil$. Since the height of the tree is at most $H_1 + H_2$, it cannot increase between global rebuildings, so the only way the tree can become too tall is when a large number of deletions causes the number of stored elements to become too small. If this occurs, we make a new global rebuilding. This ensures that the height of the tree satisfies Definition 1 all the time.

To determine which stratum a node belongs in, we store one bit in each node.

The rest of the maintenance algorithms for a 2-stratum tree are similar to the algorithms for B-trees [2] in that updates are performed in the lowest layer, and leaf subtrees are split or merged when they become too large or too small. However, special care has to be taken due to the fact that both the topmost tree and the leaf subtrees are 1-incomplete. In order to keep a tree 1-incomplete at a low cost, we must ensure that its size is not close to a power of two. For this reason, the sizes of involved trees have to be carefully controlled when splitting and merging leaf subtrees. This is achieved by using three-way splitting and merging. The constants in the algorithms may seem complicated, but this is due to the above restrictions.

2.1 Construction

Recall that we want to maintain a tree of height $\lceil \log(n+1) + \epsilon(n) \rceil$, where $\epsilon(n) = \frac{1}{\sqrt{\log(n+1)}}$. Let $\delta(n) = \frac{4}{5}(1 - 2^{-\epsilon(n)/2})$. For brevity, in the following we refer to $\epsilon(n)$ as ϵ and $\delta(n)$ and δ.

We first present the global rebuilding algorithm that allows us to obtain a well-shaped 2-stratum tree for each value of n. Second, we present the algorithms for insertion and deletion.

In the following, for convenience, we assume that n is sufficiently large to make our formulas work.

To construct a tree of n elements (with weight $N = n+1$), we choose the weight N_1 of the topmost tree and construct a tree such that the minimum and maximum weights of the leaf subtrees are $\lfloor N/N_1 \rfloor$ and $\lceil N/N_1 \rceil$, respectively. From N_1, we also determine the values of H_1 and H_2. There are three

cases.

1. $2^{\lfloor \log N \rfloor} \leq N < \frac{(1+\delta)(2-\delta)}{2} 2^{\lfloor \log N \rfloor}$. We choose $N_1 = \left\lceil (1+\delta) 2^{\lfloor \log N \rfloor - 2\lceil \log\log N \rceil - 1} \right\rceil$, which implies that $H_1 = \lfloor \log N \rfloor - 2\lceil \log\log N \rceil$ and $H_2 = 2\lceil \log\log N \rceil + 1$.

2. $\frac{(1+\delta)(2-\delta)}{2} 2^{\lfloor \log N \rfloor} \leq N \leq (2 - \frac{5}{2}\delta) 2^{\lfloor \log N \rfloor}$. In this case, we choose $N_1 = \left\lfloor (2-\delta) 2^{\lfloor \log N \rfloor - 2\lceil \log\log N \rceil - 1} \right\rfloor$, which implies that $H_1 = \lfloor \log N \rfloor - 2\lceil \log\log N \rceil$ and $H_2 = 2\lceil \log\log N \rceil + 1$.

3. $(2 - \frac{5}{2}\delta) \cdot 2^{\lfloor \log N \rfloor} < N < 2 \cdot 2^{\lfloor \log N \rfloor}$. We choose $N_1 = \left\lceil (1+\delta) 2^{\lfloor \log N \rfloor - 2\lceil \log\log N \rceil} \right\rceil$, which implies that $H_1 = \lfloor \log N \rfloor - 2\lceil \log\log N \rceil + 1$ and $H_2 = 2\lceil \log\log N \rceil + 1$.

Lemma 1 *Immediately after a global rebuilding,* $H_1 + H_2 \leq \lceil \log(n+1) + \epsilon/2 \rceil$.

Proof: The proof follows in a straightforward manner from the construction algorithm described above. □

2.2 Insertion

Before proceeding further, we define some more notation. For a node p in stratum i, we define the weight $|p|$ to be the weight of the subtree in stratum i rooted at p. Similarly, we define the height $\text{height}(p)$ to be the height of the subtree in stratum i rooted at p, and $\text{short}(p)$ to be the number of complete levels of the subtree in stratum i rooted at the node p.

Initially, we perform an insertion in a subtree in the bottommost stratum, that is, stratum 2. To insert a node x into stratum i, first insert x into an appropriate subtree T_i in stratum i. If $\text{height}(T_i) \leq H_i$, then exit. Otherwise, determine the lowest ancestor p of the inserted node such that $\text{height}(p) > \left\lceil \left(1 + \frac{\log(\frac{1}{1-\delta/4})}{H_i}\right) \log |p| \right\rceil$. We consider two cases.

Case 1: Such a node p exists. Rebuild the subtree inside stratum i rooted at p to perfect balance.

Case 2: No such node p exists. The subtree T_i is too large. If $i = 1$, then globally rebuild the entire tree. Otherwise, redistribute nodes or split leaf subtrees as follows. Let T_i' denote T_i's closest leaf subtree. (As a measure of distance between leaf subtrees, we use the length of the path between their roots.) Two subcases occur.

Case 2.1: $|T_i| + |T_i'| < \left\lceil 3.5 \cdot 2^{H_i} \right\rceil$. Redistribute nodes between T_i and T_i' such that they have the same weight (within ± 1) and terminate.

Case 2.2: $|T_i| + |T_i'| \geq \left\lceil 3.5 \cdot 2^{H_i} \right\rceil$. Split T_i and T_i' into three leaf subtrees U_i, U_i', and U_i'' of equal weight (within ± 1). Splitting two leaf subtrees into three implies that some node x' is inserted into the apex, that is, into stratum 1. Insert x' into stratum 1 using this algorithm.

2.3 Deletion

We use a deletion algorithm similar to the insertion algorithm described above. Note that we may always transform a deletion of an internal node x into a deletion of a leaf by replacing x by its inorder successor y and deleting y. Hence, to perform a deletion, we first delete a node from a subtree from stratum 2.

To delete a node x from stratum i, first delete x from an appropriate subtree T_i in stratum i. If $\text{short}(T_i) \geq H_i - 1$, then exit. Otherwise, determine the lowest ancestor p of the deleted node such that $\text{short}(p) < \left\lfloor (1 - \frac{\log(\frac{1}{1-\delta/4})}{H_i}) \log |p| \right\rfloor$. Two cases occur.

Case 1: Such a node p exists. Rebuild the subtree inside stratum i rooted at p to perfect balance.

Case 2: No such node p exists. The subtree T_i is too small. If $i = 1$, then globally rebuild the tree. Otherwise, redistribute nodes or merge leaf subtrees as follows. Let T_i' and T_i'' denote T_i's two closest leaf subtrees. Two subcases occur.

Case 2.1: $|T_i| + |T_i'| + |T_i''| > \left\lceil \frac{11}{3} \cdot 2^{H_i} \right\rceil$. Redistribute nodes between T_i, T_i', and T_i'' such that they have the same weight (within ± 1) and terminate.

Case 2.2: $|T_i| + |T_i'| + |T_i''| \leq \left\lceil \frac{11}{3} \cdot 2^{H_i} \right\rceil$. Merge T_i, T_i', and T_i'' into two leaf subtrees U_i and U_i' of equal weight (within ± 1). Merging three leaf subtrees into two implies that some node x' is deleted from the apex, that is, from stratum 1. Delete x' from stratum 1 using this algorithm.

We also globally rebuild the entire tree whenever $H_1 + H_2 > \left\lceil \log(n+1) + \frac{1}{\sqrt{\log(n+1)}} \right\rceil$.

2.4 Analysis

From the description of the algorithms above it can be shown that the above algorithms correctly maintain a 2-stratum tree. We have the following lemma.

Lemma 2 *After each update, the tree satisfies Definition 1.*

It remains to show that the amortized cost of maintaining a 2-stratum tree is logarithmic.

From the description of the maintenance algorithms, the following technical lemmas can be shown. The cumbersome proof of Lemma 3 is omitted; details of the proofs can be found in [4].

Lemma 3 *The following is true for a 2-stratum tree.*

(a) When a partial rebuilding is made at a node p in a leaf subtree, at least $\left\lceil \frac{\epsilon |p|}{48 H_2} \right\rceil$ updates have been made below p since the last time p was involved in a partial or global rebuilding, split, merge, or redistribution.

(b) When an update in a leaf subtree v causes a split, merge, or redistribution, at least $\left\lceil \frac{\epsilon \cdot 2^{H_2}}{35} \right\rceil$ updates have been made since the last time v was involved in a split, merge, or redistribution.

(c) When a partial rebuilding is made at a node p in the topmost tree, at least $\frac{\epsilon^2}{840} H_1 |p|$ updates have been made below p since the last time p was involved in a partial or global rebuilding.

(d) When an update causes a global rebuilding, at least $\left\lceil \frac{\epsilon^2 \cdot 2^{H_1 + H_2}}{1225} \right\rceil$ updates have been made since the last global rebuilding.

Lemma 4 *The restructuring operations used in a 2-stratum tree have the following costs, including the depth first search performed to decide where to make the restructuring.*

Partial rebuilding at a node p, located in a leaf subtree: $O(|p|)$.

Split or merge of leaf subtrees: $O(2^{H_2})$.

Partial rebuilding at a node u, located in the topmost tree: $O(|u|)$.

Global rebuilding of the tree: $O(2^{H_1+H_2})$.

Proof: Immediate from the fact that a rebuilding, split, or merge takes linear time in the sizes of the subtrees involved, including the time of a depth first search. □

Theorem 1 *A 2-incomplete binary search tree of height* $\left\lceil \log(n+1) + \frac{1}{\sqrt{\log(n+1)}} \right\rceil$ *can be maintained with an amortized cost of* $O(\log n)$ *per update.*

Proof: The theorem can be proved by a straightforward amortized analysis. From Lemma 3 follows that the number of updates between rebuilding, split, or merge operations is large enough to cover the costs of these operations, given by Lemma 4. □

3 Write optimality

We generalize the 2-stratum trees above by allowing k strata, for $k \geq 3$. This allows us to reduce the amortized amount of restructuring, since expensive updates high in the tree are paid by many updates lower in the tree. For convenience, we ensure that, for each stratum i, the subtrees in stratum i have the same height and are of minimum height, but we do not impose any restriction on their incomplete levels. Note that the resulting k-stratum trees can be viewed as a generalization of stratified search trees [7].

Observe that if all subtrees in stratum i have height H and size at least $2^{H-\epsilon}$, then they contribute at most ϵ to the height (above $\log n$). Thus, to obtain a tree of nearly optimal height, we make the bottom stratum contribute at most $\epsilon/2$ to the height, the second bottommost stratum contribute at most $\epsilon/4$ to the height, and so forth. However, we can ensure only that apex has minimum height, so it contributes 1 to the height. Thus, the total height is less than $\log n + 1 + \epsilon$, or less than or equal to $\lceil \log(n+1) + \epsilon \rceil$.

3.1 Construction

We define $\log^{(0)} n = n$, and, for $i \geq 1$, we define $\log^{(i)} n = \log \log^{(i-1)} n$. We define $\log^* n$ to be the smallest integer such that $\log^{(\log^* n)} n \leq 1$.

Without loss of generality, assume that $\epsilon < 1$. Let $k(N)$ be a varying parameter, where $N = n + 1$ is the weight of the tree; we refer to $k(N)$ as k for brevity. Assume that $3 \leq k \leq \log^* N - \log^*(41/\epsilon) + 1$. It is possible to construct a k-stratum tree if $\log^* N < k + 3$, but this is not necessary for our purposes.

We construct the tree as follows. Let $H_3 = 2\lceil \log \log N \rceil + 1$, and, for $3 < i \leq k$, let $H_i = 2\lceil \log H_{i-1} \rceil + 1$. Let $N_k = N$, and, for $1 \leq i \leq k-1$, let N_i be the weight of the tree if the subtrees in strata $i+1, \ldots, k$ are removed. Let $\epsilon_i = \epsilon/2^{k+1-i}$ and $\delta_i = \frac{4}{5}(1 - 2^{-\epsilon_i})$, for $2 \leq i \leq k$. For $2 \leq i \leq k-1$, we choose $N_i = \lceil N_{i+1}/W_{i+1} \rceil$, where $W_{i+1} = \left\lceil (2 - 2\delta_{i+1}) 2^{H_{i+1}-1} \right\rceil + 1$; that is, we choose the weights of subtrees in stratum i to be either $W_i - 1$ or W_i. We also choose $N_1 = \lceil N_2/W_2 \rceil$, where $H_2 = \left\lfloor \frac{1}{2} \log N_2 \right\rfloor$ and $W_2 = \left\lceil (2 - 2\delta_2) 2^{H_2 - 1} \right\rceil + 1$. We construct the tree such that the weight of the apex is N_1; for $i = 2, \ldots, k$, there are N_{i-1} subtrees in stratum i, and each subtree has weight either $\lfloor N_i/N_{i-1} \rfloor$ or $\lceil N_i/N_{i-1} \rceil$; and, for $i = 1, \ldots, k$, each subtree in stratum i is perfectly balanced and has height H_i.

Lemma 5 *After the construction of a k-stratum tree, for $2 \leq i \leq k$, the weight of any subtree in stratum i is either $W_i - 1$ or W_i, where $W_i = \left\lceil (2 - 2\delta_i) 2^{H_i - 1} \right\rceil + 1$.*

3.2 Insertion

We use an insertion algorithm similar to that of the 2-stratum tree insertion algorithm. However, we handle apex updates differently, and we use a more

general multiway splitting scheme instead of a 3-way splitting scheme.

To insert a node into the apex, we perfectly rebalance the apex. To insert a node x into stratum i in a tree T, for $i > 1$, we apply the algorithm below. Note that insertions are performed in a bottom-up manner, so we first update stratum k.

1. Insert x into a subtree T_i in stratum i.
2. If height$(T_i) \leq H_i$, then exit.
3. Otherwise, find the lowest ancestor p of node x in T_i such that height$(p) > \left\lceil (1 + \frac{\gamma_i}{H_i}) \log |p| \right\rceil$, where $\gamma_i = \log(\frac{1}{1-\delta_i/8})$.
4. If such a node exists, rebalance the maximal subtree of T_i rooted at p.
5. Otherwise, no such node exists.
   (a) Locate the subtrees U_{i1}, ..., $U_{i,m-1}$ in stratum i closest to T_i, where $m = \left\lceil \frac{4}{3\delta_i} \right\rceil$.
   (b) If possible, redistribute nodes in T_i, U_{i1}, ..., $U_{i,m-1}$ to obtain T_i', U_{i1}', ..., $U_{i,m-1}'$, such that the following conditions hold.
      i. $|T_i'| = \left\lceil (1 - \frac{\delta_i}{4}) 2^{H_i} \right\rceil$.
      ii. For all j, if $|U_{ij}| \geq \left\lceil (1 - \frac{\delta_i}{4}) 2^{H_i} \right\rceil$, then $|U_{ij}'| = |U_{ij}|$.
      iii. For all j, if $|U_{ij}| < \left\lceil (1 - \frac{\delta_i}{4}) 2^{H_i} \right\rceil$, then $|U_{ij}| \leq |U_{ij}'| \leq \left\lceil (1 - \frac{\delta_i}{4}) 2^{H_i} \right\rceil$.
      iv. T_i', U_{i1}', ..., $U_{i,m-1}'$ are all perfectly balanced.
   (c) Otherwise, split T_i, U_{i1}, ldots, $U_{i,m-1}$ into $m + 1$ subtrees V_{i1}, ..., $V_{i,m+1}$ of equal weight (within ± 1); some node x_{i-1} must be inserted into stratum $i - 1$. Insert x_{i-1} using this algorithm.

3.3 Deletion

As in the case of the insertion algorithm, the deletion algorithm for k-stratum trees is similar to the deletion algorithm for 2-stratum trees. However, we do not use partial rebuilding, and we use general multiway merging.

To delete a node from the apex, we perfectly rebalance the apex. To delete a node x from stratum i in a tree T, for $i > 1$, we apply the algorithm below. Note that deletions are performed in a bottom-up manner, so we first update stratum k.

1. Delete x from a subtree T_i in stratum i.
2. If $|T_i| \geq \left\lceil (1 - \frac{5}{4}\delta_i)2^{H_i} \right\rceil$, then exit.
3. Otherwise, perform the following steps.
   (a) Locate the subtrees $U_{i1}, \ldots, U_{i,m-1}$ in stratum i that are closest to T_i, where $m = \left\lceil \frac{4}{3\delta_i} \right\rceil$.
   (b) If possible, redistribute nodes in $T_i, U_{i1}, \ldots, U_{i,m-1}$ to obtain $T_i', U_{i1}', \ldots, U_{i,m-1}'$, such that the following conditions hold.
      i. $|T_i'| = \left\lfloor (1 - \delta_i)2^{H_i} \right\rfloor$.
      ii. For all j, if $|U_{ij}| \leq \left\lfloor (1 - \delta_i)2^{H_i} \right\rfloor$, then $|U_{ij}'| = |U_{ij}|$.
      iii. For all j, if $|U_{ij}| > \left\lfloor (1 - \delta_i)2^{H_i} \right\rfloor$, then $|U_{ij}| \geq |U_{ij}'| \geq \left\lfloor (1 - \delta_i)2^{H_i} \right\rfloor$.
      iv. $T_i', U_{i1}', \ldots, U_{i,m-1}'$ are all perfectly balanced.
   (c) Otherwise, merge $T_i, U_{i1}, \ldots, U_{i,m-1}$ into $m - 1$ subtrees $V_{i1}, \ldots, V_{i,m-1}$ of equal weight (within ± 1); some node x_{i-1} must be deleted from stratum $i - 1$. Delete x_{i-1} using this algorithm.

We also rebuild the tree after $\overline{N}/2$ updates since the last global rebuilding, where $\overline{N}$ is the weight of the tree during the last rebuilding.

3.4 Analysis

The analysis is similar to the one for 2-stratum trees, although the details are more cumbersome. For brevity, we omit all the details.

Lemma 6 *The following is true for a k-stratum tree, for $k \geq 3$.*

(a) When a partial rebuilding is made at a node p in stratum i, for $i \geq 2$, at least $\frac{|p|}{40\sqrt{2}H_i} \cdot \frac{\epsilon}{2^{k+1-i}} \cdot \prod_{j=i+1}^{k}(\frac{\ln 2}{20\sqrt{2}} \cdot \frac{\epsilon}{2^{k+1-j}}2^{H_j})$ updates have been made below p since the last time p was involved in a partial or global rebuilding, split, merge, or redistribution.

(b) When an update in subtree v in stratum i causes a split, merge, or redistribution, for $i \geq 2$, at least $\prod_{j=i}^{k}(\frac{\ln 2}{20\sqrt{2}} \cdot \frac{\epsilon}{2^{k+1-j}}2^{H_j})$ updates have been made since the last time v caused a split, merge, or redistribution, or the last time the tree was globally rebuilt.

(c) When the apex is updated, at least $\prod_{j=2}^{k}(\frac{\ln 2}{20\sqrt{2}} \cdot \frac{\epsilon}{2^{k+1-j}}2^{H_j})$ updates have been made since the apex was last updated or the tree was globally rebuilt.

Lemma 7 *The restructuring operations used in a k-stratum tree have the following costs.*

Partial rebuilding at a node p, located in a subtree in stratum i: $O(|p|)$.

Split, merge, or redistribution of subtrees in stratum i: $O(\lceil \frac{4}{3\delta_i} \rceil 2^{H_i})$.

Updating of the apex: $O(2^{H_1}) = O(2^{H_2})$.

Global rebuilding of the tree: $O(N)$.

Theorem 2 *A binary search tree of height at most $\lceil \log(n+1) + \epsilon \rceil$ can be maintained with an amortized cost of $O(\log n)$ per update, for any constant $\epsilon > 0$. Furthermore, the amortized amount of restructuring performed is $O(1/\epsilon^3)$ per update.*

Corollary 1 *There exists a sorting algorithm that requires $O(n \log n)$ time, $n \log n + O(n)$ comparisons, and $\Theta(n)$ writes in the worst case.*

Proof: The existence follows immediately from Theorem 2. □

4 Comments

We have presented new algorithms for updating a dictionary in $O(\log n)$ amortized time such that all operations require only $\lceil \log(n+1) + \epsilon(n) \rceil$ comparisons in the worst case, where $\epsilon(n) = 1/\sqrt{\log(n+1)}$. Observe that $\lim_{n\to\infty} \epsilon(n) = 0$. This improves upon the best previously known bound of Andersson [1]. He showed that a bound of $\lceil \log(n+1) + \epsilon \rceil$ comparisons can achieved for constant $\epsilon > 0$. However, he obtained efficient worst-case algorithms, which suggests the open question of whether there are matching upper bounds for $O(\log n)$ worst-case time and $O(\log n)$ amortized time update algorithms.

We have also presented new algorithms for updating a dictionary in $O(\log n)$ such that all operations require only $\log n + O(1)$ comparisons (in the worst case) and $O(1)$ writes (amortized). This result implies that sorting can be performed in $O(n \log n)$ time, such that only $n \log n + O(n)$ comparisons and $\Theta(n)$ writes are performed. No analogous result had been shown previously.

It is interesting to compare the power of writes and data movements for sorting. Munro and Raman [5] showed that n elements can be sorted in $O(n \log n)$ expected time with $O(n)$ data movements, a substantially weaker result. Because Munro and Raman restrict the number of data movements, they also restrict the number of pointers a sorting algorithm can use, since an algorithm with $\Omega(n)$ pointers can sort elements indirectly via pointers, and thus avoid many data movements. In contrast, the slow write model of computation we have proposed does not allow such "loopholes," since pointer assignments require writes. Thus, we can obtain meaningful results while freely exploiting pointers.

References

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

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

[3] D. E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[4] T. W. Lai. *Efficient maintenance of binary search trees.* PhD thesis, University of Waterloo, 1990.

[5] J. I. Munro and V. Raman. Sorting with minimum data movement (preliminary draft). In *Proceedings of the 1st Annual Workshop on Algorithms and Data Structures*, pages 552–562, 1989.

[6] 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.

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

[8] I. Wegener. The worst case complexity of McDiarmid and Reed's variant of the bottom-up-heap sort in less than $n \log n + 1.1n$. In *Proceedings of the 8th Annual Symposium on Theoretical Aspects of Computer Science*, pages 137–147, 1991.