Tight Worst-Case Bounds on the Dictionary Problem*

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

*A preliminary version published in Proceedings of International Symposium of Optimal Algorithms, 1989, Lecture Notes in Computer Science 401, pp. 106–114, Springer Verlag

Abstract

An improved worst-case bound on the dictionary problem in a comparison-based model of computation is presented. Dictionary operation can be performed at a worst-case cost of $O(\log n)$, using only $\lceil \log(n+1) + \epsilon \rceil$ comparisons, where ϵ is an arbitrary positive constant. We feel that this fact leads to a deeper understanding of of this well-studied and fundamental problem.

We also give two applications. First, we improve the upper bound on on-line sorting: for each value of δ, $\delta > 0$, there is a worst-case efficient on-line sorting algorithm using less than $\sum_{n=1}^{N} \lceil \log n \rceil + \delta N + \log N$ comparisons when sorting N elements. By worst-case efficient on-line we mean that the worst-case cost of adding an element is $\Theta(\log n)$, n is the number of elements added so far. Second, we show how to maintain a binary search tree with an optimal height of exactly $\lceil \log(n+1) \rceil$ at an amortized cost of $O(\log^2 n)$ per insertion.

1 Introduction

Searching and sorting are fundamental problems in algorithm theory and several studies of their exact bounds have been made. In this paper we present some facts which hopefully will help in creating a deeper understanding of the nature of these problems.

One of the main issues in information processing is how to store and retrieve data efficiently. In its basic form this task is denoted the *dictionary problem.* Our aim is to maintain a set of data where each element contains an identifying key and some additional data. The following operations on the set are called *the dictionary operations.*

Retrieve $(x,\ S)$**:** Locate the element x in the set S and retrieve the information in x.

Insert $(x,\ S)$**:** Add the element x to the set S.

Delete $(x,\ S)$**:** Remove x from S.

It is well known that in a comparison-based model the lower bound on the dictionary operations is $\lceil \log(n+1) \rceil$ comparisons both in the average and in the worst case. This bound can be achieved by storing the set in an array or in a perfectly balanced binary search tree. However, for both these data structures the overhead cost per update is high, $\Theta(n)$ in the worst case. In order to be regarded as efficient, a solution to the dictionary problem should have a worst case cost of $\Theta(\log n)$ per operation.

The first worst case-efficient solution to the dictionary problem was presented by Adelson-Velski and Landis [1]. Their data structure, the AVL-tree, requires $1.44 \log n$ comparisons per operation and allows updates in logarithmic worst case time. Other examples of efficient solutions to the dictionary problem with good worst case performance are symmetric binary B-trees [5] and trees of bounded balance [10], which require $2 \log n$ comparisons. A close approximation to the optimal bound was given by Maurer et. al. [9]. They presented the k-neighbour tree which requires $\lfloor (1+\epsilon) \log n + 2 \rfloor$ comparisons where ϵ is an arbitrary positive constant. Similar results are obtained by van Leeuven's and Overmars's stratified balanced trees [12].

Recently, Andersson et al. [2] showed that there exists a data structure, which is based on a generalized symmetric binary B-tree, requiring at the most

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

comparisons per search and $\Theta(\log n)$ worst-case cost per update. This bound is optimal in the leading term but the $o(\log n)$-term is quite large.

Here we decrease the gap between the upper and lower bound on the dictionary problem to an arbitrary small additive constant. We present the ϵ-*tree*, which require

$$\lceil \log(n+1) + \epsilon \rceil \tag{2}$$

comparisons per search or update for an arbitrary small value of ϵ. Thus, we may choose ϵ in such a way that for most values of n we need exactly the same number of comparisons as given by the lower bound, and we need never more than one extra comparison.

The dictionary problem is related to another well-studied problem, the *sorting problem*, and a dictionary can be viewed as an algorithm for *on-line sorting.* By efficient on-line we mean that the worst-case cost of adding an element is $\Theta(\log n)$, n is the number of elements added so far. For the general (off-line) sorting problem, the upper and lower bounds are tight except for lower order terms: the lower bound on sorting n elements is $\log n! \approx n \log n - 1.443n$ comparisons while the upper bound is $n \log n - 1.329n + o(n)$, achieved by the Ford-Johnson algorithm [6, 7]. For certain values of n the $\Theta(n)$ term can be somewhat decreased [8]. This leaves a gap of $\Theta(n)$ between the upper and lower bound. For on-line sorting the gap has remained even wider, but in this paper we shrink the gap to δn for an arbitrary small value of δ. Thus we make the gap for on-line sorting smaller than the gap for off-line sorting.

The results presented in this paper are achieved by making careful modifications of the k-neighbour trees. Therefore, we start by a short presentation of this data structure. The ϵ-tree, a node-oriented version of the k-neighbour tree with a varying value of the tuning parameter k, is introduced in Section 3, and the height of such a structure is shown to be $\lceil \log(n+1) + \epsilon \rceil$. Maintenance algorithms with a worst case cost of $O(\log n)$ are given in Section 4. In Section 5 we give a new upper bound on on-line sorting. Finally, in Section 6 we show how to maintain a tree with an *optimal* height of (i.e. exactly $\lceil \log(n+1) \rceil$) efficiently during insertions.

We use log to denote the logarithm to the power of 2 and ln to denote the natural logarithm. The number of leaves of a tree T is denoted $|T|$ and the number of edges on the longest root-to-leaf path, the height of T, is denoted $\mathrm{h}(T)$. By the *vertical distance* between two nodes we mean the difference between their heights. By the *horizontal distance* between two nodes on the same level we mean the difference in the cardinality of the two nodes when all nodes on that level are enumerated from left to right.

2 k-neighbour trees

The k-neighbour trees presented by Maurer, Ottmann, and Six [9] are unary-binary trees with at least k binary nodes between two unary nodes on the same level.

Definition 1 *A unary-binary tree is called a k-neighbour tree iff*

- *all leaves have the same depth;*
- *if a node v has only one child then*
  1. *v has at least one right neighbour*
  2. *if v has at least k right neighbours, then the k nearest right neighbours have two children, otherwise all the right neighbours have two children.*

Maurer et. al. introduced the tree as a leaf-oriented search tree which represents an ordered set in the following way:

1. All elements are stored in the leaves in sorted order from left to right.
2. Each internal node contains the value of its smallest descendant.

This way of labeling nodes is somewhat different from the ordinary labeling in a binary search tree, since the labels do not occur in sorted order during an inorder traversal. In spite of this, there is an efficient search algorithm such that the number of comparisons equals $\mathrm{h}(T)+1$. The height of a k-neighbour is given by

$$\mathrm{h}(T) \leq \left\lfloor \frac{\log |T|}{\log(2-\frac{1}{k+1})} + 1 \right\rfloor \tag{3}$$

Thus, a maximum number of comparisons of $\lfloor (1+\epsilon)\log n + 2 \rfloor$ may be achieved by choosing k large enough,

$$k > \frac{1}{2 - 2^{1/(1+\epsilon)}} - 1 \tag{4}$$

3 ϵ-trees

The new data structure presented in this paper is based on the k-neighbour tree. First, the number of comparisons per operation is decreased by one by changing the tree from being leaf-oriented to node-oriented. Second, letting the tuning parameter k take a varying value, we decrease the height of the tree. With these changes, the data structure can not be maintained using the algorithms presented by Maurer et al. Fortunately, the insertion algorithm may be modifyed in such a way that the cost per insertion is still logarithmic. Deletions are taken care of by marking elements as deleted and removing them at periodic intervals. The resulting data structure, called an *ϵ-tree* due to its low height, is defined below.

Definition 2 *For a given constant ϵ, $0 < \epsilon < 1$, an ϵ-tree is a k-neighbour tree with the following modifications:*

1. *Each binary node contains one element, while unary nodes and leaves are empty.*
2. *The elements occurs in sorted order during an inorder traversal of the tree.*
3. *The stored elements are of two types: present and deleted.*
4. *$k \geq \left\lceil \frac{3\log n}{\epsilon} \right\rceil$, where n is the number of present elements.*
5. *The number of deleted elements is less than $\frac{\epsilon}{3}n$.*

Since no elements are stored in leaves, the maximum number of comparisons required to locate an element in an ϵ-tree T equals $\mathrm{h}(T)$. Using the inequality in Lemma 1 we state the height of an ϵ-tree in Lemma 2.

Lemma 1

$$\frac{\log n}{\log\left(2 - \frac{1}{\frac{3}{\epsilon}\log n + 1}\right)} < \log n + \frac{\epsilon}{6\ln 2}\ , \quad 2 < n < \infty \tag{5}$$

Proof: The proof is shown by the substitution

$$t = 2 - \frac{1}{\frac{3\log n}{\epsilon} + 1} \tag{6}$$

This gives

$$\frac{\frac{1}{2-t}-1}{3\log t}\epsilon < \frac{\epsilon}{3}\left(\frac{1}{2-t}-1\right)+\frac{\epsilon}{6\ln 2} \quad , \quad 1<t<2 \tag{7}$$

$$\frac{\frac{1}{2-t}-1}{\log t} < \frac{1}{2-t}-1+\frac{1}{2\ln 2} \quad , \quad 1<t<2 \tag{8}$$

which implies that

$$\log t - \frac{t-1}{\left(1-\frac{1}{2\ln 2}\right)t-1+\frac{1}{\ln 2}} > 0 \quad , \quad 1<t<2 \tag{9}$$

By setting

$$f(t) = \log t - \frac{t-1}{\left(1-\frac{1}{2\ln 2}\right)t-1+\frac{1}{\ln 2}} \tag{10}$$

the inequality (9) is equivalent to

$$f(t) \geq 0 \quad , \quad 1 \leq t \leq 2 \tag{11}$$

The value of $f(t)$ is 0 for $t=1$ and $t=2$. By differentiating $f(t)$ it is straightforward to show that the function has a local maximum in the interval $1 \leq t \leq 2$ and therefore it is nonnegative within this interval. This completes the proof. □

Lemma 2 *The following is true for an ϵ-tree T containing n present elements:*

$$\mathrm{h}(T) \leq \lceil \log(n+1)+\epsilon \rceil \tag{12}$$

Proof: From Eq. (3) we have

$$\mathrm{h}(T) \leq \left\lfloor \frac{\log|T|}{\log\left(2-\frac{1}{k+1}\right)}+1 \right\rfloor \tag{13}$$

From the definition of the ϵ-tree we know that the number of deleted elements is less than $\frac{\epsilon}{3}n$, which implies that $|T| \leq (1+\frac{\epsilon}{3})(n+1)$. This, together with the fact that $k \geq \left\lceil \frac{3\log n}{\epsilon} \right\rceil$, gives

$$\mathrm{h}(T) \leq \left\lfloor \frac{\log\left(\left(1+\frac{\epsilon}{3}\right)(n+1)\right)}{\log\left(2-\frac{1}{\lceil\frac{3\log n}{\epsilon}\rceil+1}\right)}+1 \right\rfloor$$

$$= \left\lfloor \frac{\log(n+1)}{\log\left(2-\frac{1}{\lceil\frac{3\log n}{\epsilon}\rceil+1}\right)} + \frac{\log\left(1+\frac{\epsilon}{3}\right)}{\log\left(2-\frac{1}{\lceil\frac{3\log n}{\epsilon}\rceil+1}\right)}+1 \right\rfloor \tag{14}$$

From Lemma 1 follows

$$\mathrm{h}(T) \leq \left\lfloor \log(n+1) + \frac{\epsilon}{6\ln 2} + \frac{\log\left(1+\frac{\epsilon}{3}\right)}{\log\left(2 - \frac{1}{\left\lceil\frac{3\log n}{\epsilon}\right\rceil+1}\right)} + 1 \right\rfloor \tag{15}$$

For $\epsilon \leq 10$ and $n \geq 10$ we have that

$$\frac{\log\left(1+\frac{\epsilon}{3}\right)}{\log\left(2 - \frac{1}{\left\lceil\frac{3\log n}{\epsilon}\right\rceil+1}\right)} < 0.5\epsilon \tag{16}$$

Combining Eqs. (15) and (16) gives

$$\begin{aligned} \mathrm{h}(T) &\leq \left\lfloor \log(n+1) + \left(\frac{1}{2} + \frac{1}{6\ln 2}\right)\epsilon + 1 \right\rfloor \\ &\leq \lceil \log(n+1) + \epsilon \rceil \end{aligned} \tag{17}$$

which completes the proof. □

4 Maintenance Algorithms

In order to maintain ϵ-trees efficiently, we modify the insertion algorithm for k-neighbour trees in two ways. First, we adjust the algorithm to work for a node-oriented tree. Second, the fact that $k = \Theta(\log n)$ increases the cost of the original algorithm to $\Theta(\log^2 n)$ per update, and the algorithm has to be modifyed in order to decrease this cost.

The original insertion algorithm for k-neighbour trees, given by Maurer et. al, works in the following way:

1. Follow a search path down the tree and locate the node p below which a new leaf is to be inserted.
2. Create a new leaf as a child of p and call the procedure INSERT(p).

The recursive procedure INSERT(p) works as follows:

Case 1: p has two children; the insertion is completed.

Case 2: p has three children; two subcases occur:

Case 2.1: p has a neighbour q at a horizontal distance $\leq k$ which has only one child; move all children of the nodes between p and q one step in the same direction, making both p and q become binary.

Case 2.2: p does not have any neighbour q as described above; create a new unary node p' and let the only child of p' be the leftmost child of p. Then

Case 2.2.1: p is the root of the tree; create a new root of the tree which becomes the common parent of p and p'.

Case 2.2.2: p has a parent; insert p' as a new child of p's parent and call INSERT(p's parent).

Example: Figure 1 illustrates the insertion algorithm for a 3-neighbour tree. (a) An insertion of Q makes the node P ternary. Since there is no unary node at a distance k from P case 2.2 occurs. (b) A new unary node is created and the node N on the level above becomes ternary. Again, case 2.2 occurs. (c) The node J has a unary neighbour (A) which is close enough. Case 2.1 occurs and subtrees are moved between J and A. (d) The insertion is completed. □

In order to make this algorithm work for a node-oriented k-neighbour tree, we modify it in the following way:

- Instead of placing the inserted element in a new leaf below p, it is placed in p and the new leaf is left empty.
- When a node becomes ternary, it contains two elements.
- When subtrees are moved between p and q in Case 2.1 all elements between p and q, on the same level or above, are shifted one step in inorder or reverse inorder. After this shift, both p and q contain one element each. Shifting elements also preserves the binary search tree ordering of the nodes.
- When the node p is split in Case 2.2, we remove the smallest element from p and add it to p's parent (which may become ternary).

Example: The insertion algorithm for a node-oriented 3-neighbour tree is illustrated in Figure 2. The example is analogous to the example in Figure 1. In the first step (a), the insertion of Q results in a ternary node PQ. In the last step from (c) to (d), the elements B, F, I, and M are shifted in reverse inorder. □

The fact that the insertion algorithm for k-neighbour trees can be modifyed to handle a node-oriented tree is not enough to maintain an ϵ-tree efficiently. The cost of the algorithm as presented by Maurer et. al. is $\Theta(k \log n)$ per insertion. Since $k = \Theta(\log n)$ in a ϵ-tree, the cost per update with this algorithm would be $\Theta(\log^2 n)$. The critical point of the algorithm is the search for unary neighbours in Case 2. This case may occur at $\Theta(\log n)$ levels of the tree, each time requiring $\Theta(k)$ time. Fortunately, this cost can be reduced by adding some information to the nodes, as shown in Lemma 3—4 below.

Lemma 3 *Let p and q be two nodes at the same level in a k-neighbour tree and let p_1, p_2 and q_1 be three nodes $\lfloor \log(k+1) \rfloor$ levels lower, such that p_1 and p_2 are descendants of p and q_1 is a descendant of q. Let dist(p,q) denote the horizontal distance between the nodes p and q. Then, the following is true:*

1. *if p_1 is unary then p_2 is binary.*
2. *if dist$(p,q) \leq 2$ then dist$(p_1,q_1) \leq 6k$.*
3. *if dist$(p,q) > 2$ then dist$(p_1,q_1) > k$.*

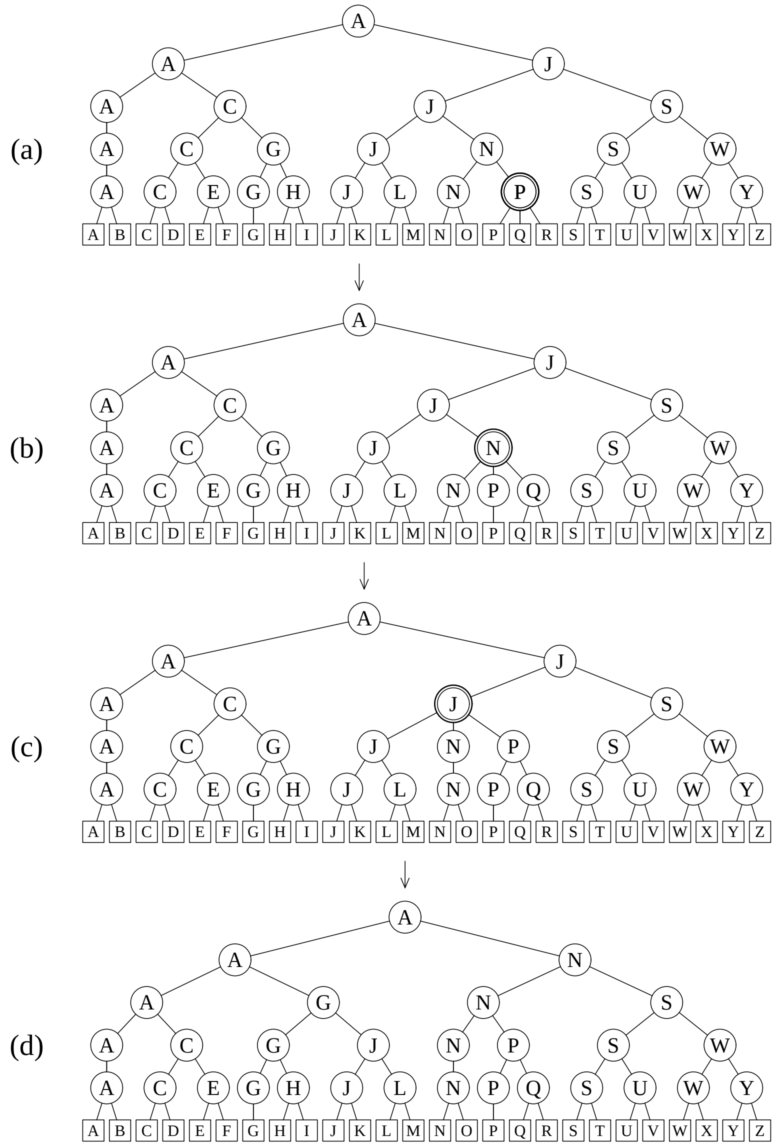

Figure 1: *Insertion into a 3-neighbour tree.*

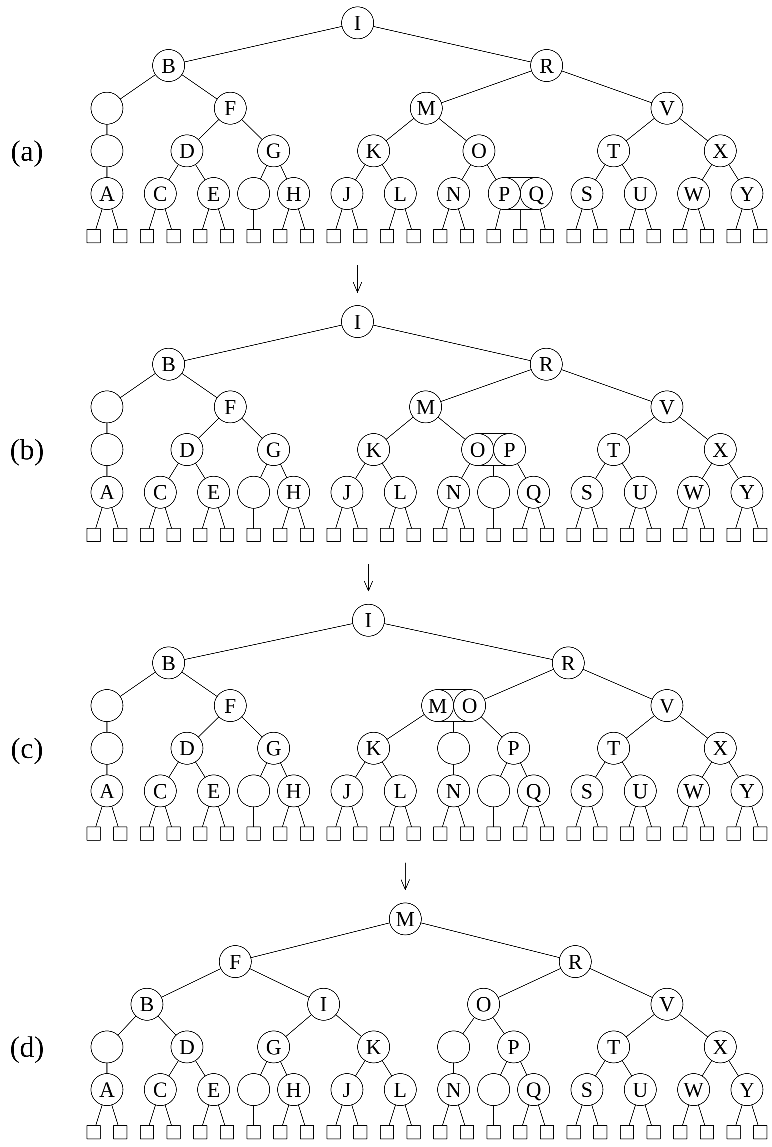

Figure 2: *Insertion into a node-oriented 3-neighbour tree.*

Proof: The lemma follows in a straightforward way from the definition of k-neighbour trees. □

Lemma 4 *A node oriented k-neighbour tree may be maintained at a worst-case cost of* $\Theta(\log n + k)$ *per insertion.*

Proof: Since the only part of the insertion algorithm that costs more than $\Theta(\log n + k)$ is the search for a unary neighbour in Case 2, we concentrate on this part.

We add one bit to each internal node specifying whether the node has a unary descendant at the vertical distance of $\lfloor \log(k+1) \rfloor$ or not. From Lemma 3 we have that each node has at most one unary descendant at this vertical distance.

Insertions are performed using the algorithm described above with the following modification: Instead of making an explicit search for a unary neighbour of p in Case 2, we examine the stored bit in at most five nodes: p's ancestor at the vertical distance $\lfloor \log(k+1) \rfloor$ and the two nearest left and right neighbours (if any) of this ancestor. In order to examine these nodes in constant time we maintain five "trailing pointers" while traversing the search path. If any of these nodes has a unary descendant q at the same level as p we proceed to Case 2.1 regardless if the horizontal distance between p and q is greater than k or not. Using this algorithm, the time spent locating the node q in Case 2 is $O(1)$ at each level and the total time spent searching for unary nodes is $O(\log n)$.

With this modification of the algorithm, the dominating cost is the one of moving nodes in Case 2.1. This cost depends on the horizontal distance over which nodes are moved between p and q. From Lemma 3 we have that the distance between these two nodes is at most $6k$. Thus, the cost of moving nodes is $\Theta(\log n + k)$ in the worst case and the proof is completed. □

The algorithm described in Lemma 4 does not behave in the same way as the algorithm for a node-oriented k-neighbour tree described above, since nodes may be shifted over distances longer than k. This is illustrated in Example 3 below. The difference between the two algorithms may be eliminated by adding some further information to each node, specifying the position at which a possible unary descendant is located. In this way we may avoid shifting nodes longer than k steps.

Example: The algorithm given in Lemma 4 is illustrated in Figure 3. To look for unary neighbours of the node PQ in Figure 3a, we examine PQ's ancestor at a distance of 2, M, and its three neighbours. Since F has a unary descendant, nodes are moved between this unary node and P, and the algorithm terminates. □

Altogether, the insertion algorithm for a node-oriented k-neighbour tree may be described as below.

1. Follow a search path down the tree until a node p is found on the lowest level.

2. Create a new leaf as a child of p. Add the inserted element to p and call INSERT(p).

The recursive procedure INSERT(p) works as follows:

Case 1: p is binary and contains one element; the algorithm terminates.

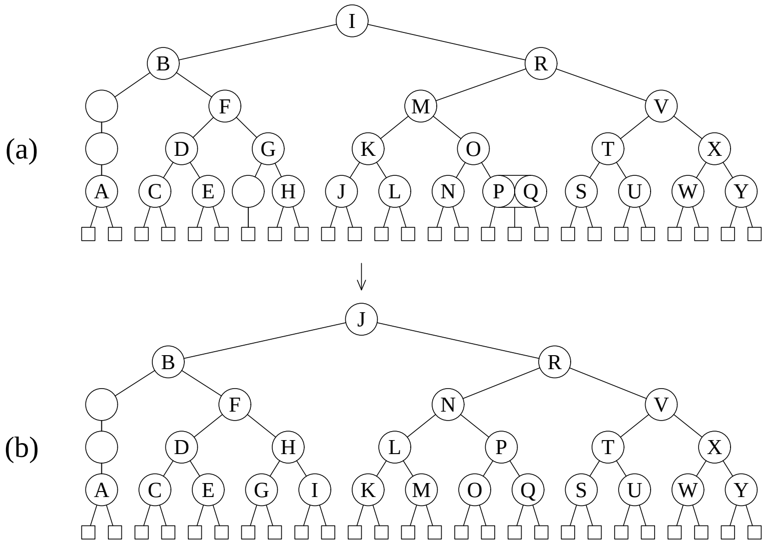

Figure 3: *Insertion into an ϵ-tree.*

Case 2: p is ternary and contains two elements; We examine p's ancestor at the distance $\lfloor \log(k+1) \rfloor$ as well as the two nearest neighbours to the left and right of this ancestor to see whether any of these five nodes has a unary descendant q at the same level as p. Two subcases occur:

Case 2.1: such a node q exists; move all children of the nodes between p and q one step in the same direction, making both p and q become binary. Shift all elements between p and q, on the same level or above, one step in inorder or reverse inorder. After this p and q contain one element each and the inorder-ordering of elements is preserved.

Case 2.2: such a node q does not exist; create a new unary node p' and let the only child of p' be the leftmost child of p. Then

Case 2.2.1: p is the root of the tree; create a new root of the tree which becomes the common parent of p and p'. Place the smallest of p's two elements in the root.

Case 2.2.2: p has a parent; insert p' as a new child of p's parent, place the smallest of p's two elements in the parent, and call INSERT(p's parent).

Now, having advised an insertion algorithm, we give the maintenance algorithms for ϵ-trees. Queries are performed in the following way:

Retrieve: If the element is found in the tree and marked as present the search is successful.

Insert: Insert the element into the tree, using the algorithm described below, and mark it as present.

Delete: Locate the element and mark it as deleted without removing it.

In order to remove deleted elements and change the value of k we rebuild the entire tree periodically. When the number of deletions since the latest rebuilding exceeds $\frac{\epsilon}{3}n$, all elements which are marked as deleted are removed and the tree is rebuilt. In this way the number of deleted elements is always less than $\frac{\epsilon}{3}n$. Let n_0 denote the value of n at the latest rebuilding of the tree. We have

$$n_0 \geq \left(1 - \frac{\epsilon}{3}\right) n \tag{18}$$

When the tree is rebuilt we set

$$k = \left\lceil \frac{3}{\epsilon} \log\left(\frac{n}{1-\frac{\epsilon}{3}}\right) \right\rceil \tag{19}$$

Eqs. (18) and (19) give that

$$k = \left\lceil \frac{3}{\epsilon} \log\left(\frac{n_0}{1-\frac{\epsilon}{3}}\right) \right\rceil \geq \left\lceil \frac{3\log n}{\epsilon} \right\rceil \tag{20}$$

Thus, rebuilding after every $\frac{\epsilon}{3}n$ deletions and chosing an appropriate value of k each time will make the tree satisfy Definition 2.

Lemma 5 *The amortized cost per update in an ϵ-tree is $\Theta\left(\frac{\log n}{\epsilon}\right)$.*

Proof: From Lemma 4 we know that the cost per insertion is $O(\log n + k)$ and from the definition of ϵ-trees we know that $k = \Theta\left(\frac{\log n}{\epsilon}\right)$. Thus, the worst-case cost for a single insertion is

$$\Theta\left(\log n + \Theta\left(\frac{\log n}{\epsilon}\right)\right) = \Theta\left(\frac{\log n}{\epsilon}\right) \tag{21}$$

Deletion causes no restructuring work, since the deleted element is only marked as deleted without being removed.

Rebuilding of a tree requires $\Theta(n)$ time. This is amortized over $\Theta(\epsilon n)$ deletions which implies that the amortized cost per deletion is $\Theta\left(\frac{1}{\epsilon}\right)$. Thus, the dominating cost is the one given in Eq. (21), which completes the proof. □

The amortized result obtained in Lemma 5 may be transformed into a worst-case result by maintaining two trees simultaneously instead of one. In this way the cost of changing the value of k (including a global rebuilding) may be distributed in such a way that the worst-case cost per update becomes logarithmic. Providing that the two trees are not being rebuilt at the same time, there will always be one tree in which queries can be performed. By storing ranks of elements we only need to perform element-comparisons in one of the trees; in the second tree we search by the rank. Having to make element-comparisons in both trees would have doubled the number of comparisons. This technique, also used in [2], is a refinement of the method of *global rebuilding*, introduced by Overmars [11].

Theorem 1 *There is a data structure supporting search and updates at a cost of $\lceil \log(n+1) + \epsilon \rceil$ comparisons and $\Theta(\log n)$ time in the worst case.*

Proof: We use two ϵ-trees and a queue of updates to be performed. In each node of the trees we store the number of present descendants. This stored information allows us to compute the rank of elements. The data structure is maintained in the following way:

1. After $\frac{n}{2}$ updates the value of k is changed and each tree is rebuilt. Rebuildings are scheduled in such a way that the two trees are not being rebuilt at the same time.
2. Rebuilding work is distributed over the updates in such a way that the time spent per update is $O(\log n)$.
3. Updates are performed in the following way:
   (a) An update is always made in a tree which is not being reconstructed. Using the stored information on the number of present descendants of each node, we compute the rank of the inserted/deleted element.
   (b) If the other tree is not being rebuilt the insertion/deletion is made also in that tree. No element-comparisons are required for this update, since we use the rank to locate the element.
   (c) If the other tree is being rebuilt we store the update in a queue to be performed when the rebuilding is completed.

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

Although each update is performed in both trees, element-comparisons are made only the first time. Thus, the number of comparisons equals the height of the tree, which completes the proof. □

5 On-Line Sorting

The data structure presented above allows us to present tight bounds on on-line sorting. The trivial lower bound on the number of comparisons required by on-line sorting of N elements is $\sum_{n=1}^{N} \lceil \log N \rceil$. Below, we give a new upper bound, leaving a gap of only $\delta N + \log N$ for arbitrary small δ. We use n to denote the number of inserted elements at a given moment.

Theorem 2 *For each value of δ, $\delta > 0$, there is an on-line sorting algorithm with $\Theta(\log n)$ worst case insertion cost, using less than $\sum_{n=1}^{N} \lceil \log n \rceil + \delta N + \log N$ comparisons.*

Proof: W. l. o. g. we assume that N is a power of two. We perform our on-line sorting by using the data structure described in Theorem 1 with $\epsilon = \log \frac{2}{2-\delta}$. From Theorem 1 we know that insertion of the nth element requires at most $\lceil \log n + \epsilon \rceil$ comparisons. Thus, the total number of comparisons required to insert N elements is given by

$$\sum_{n=1}^{N} \lceil \log n + \epsilon \rceil = \sum_{k=0}^{\log N - 1} \sum_{n=2^k}^{2^{k+1}} \lceil \log n + \epsilon \rceil \tag{22}$$

The term $\lceil \log n + \epsilon \rceil$ equals either $\lceil \log n \rceil$ or $\lceil \log n + 1 \rceil$, depending on the value of n. The constant 1 is added only when

$$\lceil \log n + \epsilon \rceil = \lceil \log 2^k \rceil + 1 \tag{23}$$

For $2^k < n \leq 2^k + 1$ this occurs when

$$\begin{aligned} \log n + \epsilon &\geq \log 2^k + 1 \\ 2^\epsilon n &\geq 2^{k+1} \\ n &\geq \frac{2^{k+1}}{2^\epsilon} \end{aligned} \tag{24}$$

That is, for $2^k < n < (2 - \frac{2}{2^\epsilon})2^k$ we use $\lceil \log n \rceil$ comparisons and for $(2 - \frac{2}{2^\epsilon})2^k \leq n \leq 2^{k+1}$ we may need $\lceil \log n \rceil + 1$ comparisons. This implies that the number of comparisons is given by

$$\begin{aligned} & \sum_{k=1}^{\log N - 1} \left(\sum_{n=2^k}^{\frac{2^{k+1}}{2^\epsilon} - 1} \lceil \log n \rceil + \sum_{\frac{2^{k+1}}{2^\epsilon}}^{2^{k+1}} \lceil \log n \rceil + 1 \right) \\ = & \sum_{n=1}^{N} \lceil \log n \rceil + \sum_{k=1}^{\log N - 1} \sum_{\frac{2^{k+1}}{2^\epsilon}}^{2^{k+1}} 1 \\ < & \sum_{n=1}^{N} \lceil \log n \rceil + \left(2 - \frac{2}{2^\epsilon}\right) N + \log N \\ = & \sum_{n=1}^{N} \lceil \log n \rceil + \delta N + \log N \end{aligned} \tag{25}$$

This completes the proof. □

6 An Improved Algorithm for Insertions Into a Tree of Optimal Height

Although a bound of $\lceil \log(n+1) + \epsilon \rceil$ comparisons per operation may seem low enough, we may perform even better in the case when updates are not to common and only insertions are made. As will be shown below, a binary search tree with an optimal height of exactly $\lceil \log(n+1) \rceil$ may be maintained at an amortized cost of $O(\log^2 n)$ per insertion. This improves the result by Andersson and Lai [3], where algorithms for an optimally balanced binary search tree (i.e. a binary search tree where the difference between the length of longest and shortest path is at most one) was given at a cost of $O(\log^3 n)$, per insertion. The technique (and the proof) is identical to the one used in [3] with the difference that we use an ϵ-tree instead of partial rebuilding. Note, however, that although we maintain optimal height, we do not maintain an optimally balanced tree as in [3].

Theorem 3 *A binary search tree with a height of exactly* $\lceil \log(n+1) \rceil$ *may be maintained with an amortized cost of* $O(\log^2 n)$ *per insertion, provided that no deletions are made.*

Proof: The proof is identical to the proof of Theorem 4 in [3] with the difference that we use an ϵ-tree instead of partial rebuilding. The basic idea is to use a varying value of ϵ, such that when $n+1 = 2^k - 2^{k-i+1}$, $i = 2, 3, 4 \ldots k$, we change the value of ϵ to $\log \frac{2^i}{2^i - 1}$. Following the analysis in [3] we get a height of exactly $\lceil \log(n+1) \rceil$ and the amortized cost per update is $O(\log^2 n)$. $\square$

7 Comments

Searching and sorting are fundamental problems in algorithm theory and several studies of their exact bounds have been made. Introducing the ϵ-tree, we have achieved (almost) tight comparison-based worst-case bounds on dictionary operations. The result is mainly of theoretical interest, and only in cases when updates are rare or when comparisons are expensive the effort of implementing an ϵ-tree is worthwhile.

References

[1] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Dokladi Akademia Nauk SSSR*, 146(2):1259–1262, 1962.

[2] A. Andersson, Ch. Icking, R. Klein, and Th. Ottmann. Binary search trees of almost optimal height. *Acta Inormatica*, 28:165–178, 1990.

[3] A. Andersson and T. W. Lai. Fast updating of well-balanced trees. In *Proc. Scandinavian Workshop on Algorithm Theory, SWAT '90, Bergen*, pages 111–121. Springer Verlag, 1990. Lecture Notes in Computer Science 447.

[4] A. Andersson and T. W. Lai. Comparison-efficient write-optimal searching and sorting. In *Proc. 2nd ann. International Symposium on Algorithms, December 16-18, 1991, Academia Sinica, Taipei, Taiwan*, pages 273–282. Springer Verlag, 1991. Lecture Notes in Computer Science 557.

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

[6] L. R. Ford and S. M. Johnson. A tournament problem. *Amer. Math. Monthly*, 66(5):387–389, 1959.

[7] F. K. Hwang and S. Lin. An analysis of Fords and Johnson's sorting algorithm. In *3rd Annual Princeton Conference on Information Sciences and Systems*, pages 292–296. Princeton University, Princeton, N. Y., 1969.

[8] G. K. Manacher, T. D. Bui, and T. Mai. Optimum combinations of sorting and merging. *Journal of the ACM*, 36(2):290–334, 1989.

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

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

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

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