PREPRINT. In Proc. Symposium on Optimal Algorithms, pages 106-114. Springer Verlag, 1989.

Optimal Bounds on the Dictionary Problem

Arne Andersson
Department of Computer Science
Lund University
Lund, Sweden

Abstract

A new data structure for the dictionary problem is presented. Updates are performed in $\Theta(\log n)$ time in the worst case and the number of comparisons per operation is $\lceil \log n + 1 + \epsilon \rceil$, where ϵ is an arbitrary positive constant.

1 Introduction

One of the fundamental and most studied problems in computer science is the *dictionary problem*, that is the problem of how to maintain a set of data during the operations search, insert and delete. It is well known that in a comparison-based model the lower bound on these 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.

An efficient dynamic data structure for the dictionary problem should have a worst case cost of $\Theta(\log n)$ per operation. The first efficient solution 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 data structures for the dictionary problem with good worst case performance are symmetric binary B-trees [3] and trees of bounded balance [5], which require $2 \log n$ comparisons. A close approximation to the optimal bound was given by Mauer et. al. [4]. 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 [6] and by Andersson's generalized symmetric binary B-trees [2]. For all these structures there is a tradeoff between the number of comparisons and the maintenance cost in such a way that a lower value of ϵ corresponds to a higher cost for restructuring during updates.

Although a bound of $(1+\epsilon) \log n$ is sufficiently low for practical purposes there is still a gap between this and the optimal bound. Recently, Andersson [2] showed that there exists a data structure, which is based on the generalized symmetric binary B-tree, requiring at the most

$$\left\lfloor \left(1 + \frac{1}{\log \log n}\right) \log n \right\rfloor + 1 = \log n + o(\log n) \qquad (1)$$

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

Here we show that the difference between the upper and lower bound on the dictionary problem is only a 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. We may choose ϵ in such a way that mostly we make only one, and never more than two extra comparisons compared to the lower bound.

The paper is organized in the following way: In section 2 we give a short description of the k–neighbour tree [4], on which our result is based. We show how to improve the insertion algorithm by adding some information to the nodes in the tree. In section 3 we introduce the ϵ-tree. This data structure is a k-neighbour tree in which we let the value of the tuning parameter k change as the number of stored elements changes. The ϵ-tree can be maintained with an amortized cost of $\Theta\left(\frac{\log n}{\epsilon}\right)$ per update. Using two ϵ-trees we improve the amortized bound into a worst case bound.

We use the same terminology regarding trees as used in [4]. The height of a tree containing n elements is denoted H_n and the number of comparisons required to search among n elements is denoted C_n. log denotes the logarithm to the power of 2 and ln denotes the natural logarithm.

2 k–neighbour trees

The k-neighbour trees presented by Mauer et. al. [4] are unary-binary trees where there are at least k binary nodes between two unary nodes on the same level. We have the following definition:

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

1. *All leaves have the same depth.*
2. *If a node v has only one child then*
   (a) *v has at least one right neighbour*
   (b) *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.*

An example of a k-neighbour tree is given in Figure 1. The tree is 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 child.

The tree has a maximum height of

$$H_n \leq \left\lfloor \frac{\log n}{\log(2 - \frac{1}{k+1})} + 1 \right\rfloor \tag{3}$$

Figure 1: *A 3-neighbour tree.*

Although the elements are not ordered in the same way as in an ordinary binary search tree there is an efficient search algorithm such that

$$C_n = H_n + 1 \tag{4}$$

Mauer e. al. gave maintenance algorithms with $\Theta(\log n)$ worst case performance. Here we are concerned only with the insertion algorithm, which works as follows:

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 distance $\leq k$ which has only one child. Make both p and q binary by moving all leaves between them one step.
>
> Case 2.2: p has no neighbour q as described above. Create a new unary node p' and let its only child be the leftmost child of p. Then
>
> > Case 2.2.1: p has no parent. Create a new root of the tree to be the common parent of p and p'.
> >
> > Case 2.2.2: p has a father. Insert p' as a new child of p's father and call INSERT(p's parent).

The largest amount of restructuring work is made during case 2.1 when a number of nodes are moved between p and q. The worst case cost for this is $\Theta(k + \log n)$. Since the algorithm terminates in case 2.1 this work is performed only once. The search for a unary

neighbour of p in case 2 takes $\Theta(k)$ time. Since case 2.2 may occur on each level of the tree we get a worst case cost of $\Theta(k \log n)$ for insertion.
In order to obtain our result we have to improve the complexity of the insertion algorithm for k-neighbour trees. This can be made by adding some information to the nodes, as shown in Lemma 1 below.

Lemma 1 *An insertion into a k-neighbour tree requires $\Theta(\log n + k)$ time in the worst case.*

Proof: To each internal node we add a boolean variable telling whether the node has a unary descendant at the distance of $\lfloor \log(k+1) \rfloor$ or not. From the definition of k-neighbour trees follows that each node has at the most one unary descendant at this distance.

Insertions are performed as in the algorithm described in section 2 with the following modification: Instead of making an explicit search for a unary neighbour of p in case 2 we check the ancestor at the distance $\lfloor \log(k+1) \rfloor$ as well as its left and right neighbours. If any of the three nodes has a unary descendant q at the same level as p we proceed with case 2.1 even if the distance between p and q is greater than k. Using this algorithm, the time spent locating the node q in case 2 is $O(1)$ at each level and the total time is $O(\log n)$.

Thus the dominating cost is the one for moving nodes in case 2.1. This cost is depending on the longest possible distance nodes are moved between p and q. Since the ancestors of p and q at distance $\lfloor \log(k+1) \rfloor$ are neighbours (or maybe even the same node) the distance between p and q is less than

$$2 \cdot 2^{\lfloor \log(k+1) \rfloor} \leq 4k + 2 = \Theta(k)$$

Thus the cost for insertion is $\Theta(\log n + k)$ in the worst case and the proof is completed. □

An example of the insertion algorithm is given in Figure 2.

3 ϵ-trees

The result in Lemma 1 allows us to let k take a value of $\Theta(\log n)$, still having a logarithmic cost per insertion. However, the nature of k-neighbour trees does not allow us to change the value of k dynamically. This problem is solved by changing k at repeated intervals.

In this way we achieve a tree with a maximum height of $\lceil \log n + \epsilon \rceil$. The resulting data structure, called the *ϵ-tree* due to its low height, is defined below.

Definition 2 *An ϵ-tree is an k-neighbour tree with the following modifications*

1. *The stored elements are of two types: present and deleted.*
2. *$k \geq \left\lceil \frac{3 \log n}{\epsilon} \right\rceil$, n is the number of present elements.*
3. *The number of deleted elements is less than $\frac{\epsilon}{3} n$*

where ϵ is an arbitrary constant greater than zero.

Figure 2: *A insertion in a 3-neighbour tree. (a) A new leaf is inserted below p. The ancestor at a distance of* $\lfloor \log(k+1) \rfloor$ *and its neighbours (gray-coloured) are examined. One of them has a unary descendant q, which is a neighbour of p.*
(b) The tree after moving nodes between p and q.

Queries are performed in the following way:

Insertion: Insert the element into the tree and mark it as present.

Deletion: Locate the element and mark it as deleted.

Search: Locate the element. If the element is found and marked as present the search is successful.

In order to satisfy the definition we rebuild the tree repeatedly. When the number of updates since the latest rebuilding exceeds $\frac{\epsilon}{3}n$ all deleted elements are removed and the tree is rebuilt. In this way the number of inserted 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

$$n_0 \geq \left(1 - \frac{\epsilon}{3}\right) n \tag{5}$$

When the tree is rebuilt we set

$$k = \left\lceil \frac{3}{\epsilon} \log \left(\frac{n}{1 - \frac{\epsilon}{3}} \right) \right\rceil \tag{6}$$

Equations (5) and (6) gives that

$$k = \left\lceil \frac{3}{\epsilon} \log \left(\frac{n_0}{1 - \frac{\epsilon}{3}} \right) \right\rceil \geq \left\lceil \frac{3 \log n}{\epsilon} \right\rceil \tag{7}$$

In Lemma 2 and 3 below we analyze the height and maintenance cost of an ϵ-tree.

Lemma 2 *The following is true for an ϵ-tree:*

$$H_n \leq \lceil \log n + \epsilon \rceil$$

Proof: The height of the tree depends on k and the total number of elements (present and deleted), denoted N. From equation (3) we have

$$H_n \leq \left\lfloor \frac{\log N}{\log\left(2 - \frac{1}{k+1}\right)} + 1 \right\rfloor \tag{8}$$

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 $N < (1 + \frac{\epsilon}{3})n$. This together with the fact that $k \geq \left\lceil \frac{3 \log n}{\epsilon} \right\rceil$ gives that

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

$$= \left\lfloor \frac{\log n}{\log\left(2 - \frac{1}{\left\lceil \frac{3 \log n}{\epsilon} \right\rceil + 1}\right)} + \frac{\log\left(1 + \frac{\epsilon}{3}\right)}{\log\left(2 - \frac{1}{\left\lceil \frac{3 \log n}{\epsilon} \right\rceil + 1}\right)} + 1 \right\rfloor \tag{9}$$

For small values of ϵ we have that

$$\log\left(1 + \frac{\epsilon}{3}\right) < \frac{\epsilon}{3 \ln 2} < 0.5\epsilon \tag{10}$$

and

$$\log\left(2 - \frac{1}{\left\lceil \frac{3 \log n}{\epsilon} \right\rceil + 1}\right) \approx 1 \tag{11}$$

which implies that

$$\frac{\log\left(1 + \frac{\epsilon}{3}\right)}{\log\left(2 - \frac{1}{\left\lceil \frac{3 \log n}{\epsilon} \right\rceil + 1}\right)} < 0.6\epsilon < \left(1 - \frac{1}{6 \ln 2}\right)\epsilon \tag{12}$$

Combining equation (9) and (12) gives that

$$H_n \leq \left\lceil \frac{\log n}{\log\left(2 - \frac{1}{\left\lceil \frac{3 \log n}{\epsilon} \right\rceil + 1}\right)} + \left(1 - \frac{1}{6 \ln 2}\right)\epsilon \right\rceil \tag{13}$$

The proof follows from the inequality

$$\frac{\log n}{\log\left(2 - \frac{1}{\frac{3}{\epsilon} \log n + 1}\right)} + \left(1 - \frac{1}{6 \ln 2}\right)\epsilon < \log n + \epsilon \quad , \quad 2 < n < \infty \tag{14}$$

The inequality (14) is not obvious but can be shown by the substitution

$$t = 2 - \frac{1}{\frac{3 \log n}{\epsilon} + 1} \tag{15}$$

This gives

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

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

$$\log t - \frac{t-1}{\left(1-\frac{1}{2\ln 2}\right)t-1+\frac{1}{\ln 2}} > 0 \quad , \quad 1<t<2 \tag{16}$$

By setting

$$f(t) = \log t - \frac{t-1}{\left(1-\frac{1}{2\ln 2}\right)t-1+\frac{1}{\ln 2}} \tag{17}$$

the inequality (16) is equivalent to

$$f(t) \geq 0 \quad , \quad 1 \leq t \leq 2 \tag{18}$$

Derivation gives that

$$f'(t) = \frac{(4(\ln 2)^2 - 4\ln 2 + 1)t^2 + (12\ln 2 - 10(\ln 2)^2 - 4)t + 4(\ln 2)^2 - 8\ln 2 + 4}{((2\ln 2 - 1)t - 2\ln 2 + 2)^2 t \ln 2} \tag{19}$$

Both $f(t)$ and $f'(t)$ are continuous in the interval $1 \leq t \leq 2$. The equation

$$f'(t) = 0 \tag{20}$$

has two solutions, namely

$$t_1 = \frac{2(\ln 2)^2 - 4\ln 2 + 2}{4(\ln 2)^2 - 4\ln 2 + 1} \approx 1.262 \tag{21}$$

and

$$t_2 = 2 \tag{22}$$

Inspection of $f'(t)$ gives that

$$f'(t) > 0 \quad , \quad 1 \leq t < t_1 \tag{23}$$

$$f'(t) < 0 \quad , \quad t_1 < t < 2 \tag{24}$$

Thus $f(t)$ has a local maximum at t_1. This together with the fact that $f(1) = f(2) = 0$ gives that

$$f(t) \geq 0 \quad , \quad 1 \leq t \leq 2 \tag{25}$$

which completes the proof. □

Lemma 3 *The amortized cost per update in an ϵ-tree is $\Theta\left(\frac{\log n}{\epsilon}\right)$.*

Proof: From Lemma 1 and equation (7) we get the cost for a single update to be

$$\Theta(\log n + k) = \Theta\left(\log n + \frac{3\log n}{\epsilon}\right) = \Theta\left(\frac{\log n}{\epsilon}\right) \tag{26}$$

The rebuilding of a tree requires $\Theta(n)$ time. This is amortized over $\Theta(\epsilon n)$ updates which implies that the amortized cost per update is $\Theta\left(\frac{1}{\epsilon}\right)$. Thus the dominating cost is the one given in equation (26) which completes the proof. □

The result of Lemma 2 and 3 allows us to achieve a very low upper bound on the dictionary problem at a logarithmic amortized cost as shown in Theorem 1 below.

Theorem 1 *For any value of $\epsilon, \epsilon > 0$, there is a data structure such that*

$$C_n \leq \lceil \log n + 1 + \epsilon \rceil$$

and the amortized cost per update is

$$\Theta\left(\frac{\log n}{\epsilon}\right)$$

Proof: The proof follows from equation (4) and Lemma 2 and 3. □

Finally, we show how to improve the result of Theorem 1 into a worst case result. BY keeping two ϵ-trees we do not have to stop making queries while we are rebuilding a tree since there will always be one tree in which the queries can be performed.

Theorem 2 *For any value of $\epsilon, \epsilon > 0$, there is a data structure such that*

$$C_n \leq \lceil \log n + 1 + \epsilon \rceil$$

and the worst case cost per update is

$$\Theta\left(\frac{\log n}{\epsilon}\right)$$

Proof: We use a data structure consisting of two ϵ-trees. In each node of the trees we store the number of present descendants. The data structure is maintained in the following way:

1. When one of the trees is rebuilt the rebuilding work is distributed over the updates in such a way that the maximum time spent per update is $\Theta(\log n)$.
2. The two trees are not being rebuilt at the same time.
3. Updates are performed in the following way:
   (a) The update is made in a tree which is not being rebuilt. From the stored information about the number of present descendants we compute the cardinality of the inserted/deleted element.

(b) If the other tree is not being rebuilt an update is also made in that tree. For this second update we use the cardinality of the element and therefore no comparisons are required.

(c) If the other tree is being rebuilt we store the update in a queue to be performed at the end of the rebuilding. The cardinality of the element is stored in the queue to avoid comparisons when the update is to be made in the tree.

A rebuilding consist of two phases: construction of a new tree and performance of the queued updates. The first step requires $\Theta(n)$ time and the second one requires $\Theta(n \log n)$ time. Therefore, the entire construction can be distributed over a linear number of updates at a worst case cost of $\Theta(\log n)$ per update. Although each update is performed in both trees we make element-comparisons only the first time. The rest of the proof is similar to the proof of Theorem 1. □

4 Comments

We have shown that there exist a data structure such that we can perform the dictionary queries insert, search and delete in logarithmic time, making at the most two comparisons more than the optimal number.

The method to improve complexity by using a "varying constant" (k in the ϵ-tree) is not restricted to search trees and it might also be used to improve complexity in other applications.

References

[1] G. M. Adelson-Velski and E. M. Landis. An algorithm for the organization of information. *Dokladi Akademia Nauk SSSR*, 146(2), 162.

[2] A. Andersson. Binary search trees of almost optimal height. *tech. report, Department of Computer Science, Lund University*, 1988.

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

[4] H. A. Mauer, Th. Ottman, and H. W. Six. Implementing dictionaries using binary trees of very small height. *Information Processing Letters*, 5(1), 1976.

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

[6] J. van Leeuwen and M. H. Overmars. Stratified balanced search trees. *Acta Informatica*, 18, 1983.