JOURNAL OF COMPUTER AND SYSTEM SCIENCES 28, 379–394 (1984)

New Trie Data Structures Which Support Very Fast Search Operations

DAN E. WILLARD*

Bell Laboratories, Holmdel, New Jersey 07733

Received September 2, 1981; revised June 29, 1983

A new data structure called the q-fast trie is introduced. Given a set of N records whose keys are distinct nonnegative integers less than some initially specified bound M, a q-fast trie uses space $O(N)$ and time $O(\sqrt{\log M})$ for insertions, deletions, and all the retrieval operations commonly associated with binary trees. A simpler but less space efficient data structure called the p-fast trie is defined.

1. INTRODUCTION

Throughout this paper, S will denote a set of N records whose keys are distinct nonnegative integers, less than some initially specified bound M. We will measure complexity in terms of the time for a Random Access Machine to perform the following retrieval operations:

(i) MEMBER(K): Determine whether or not key K belongs to the set S;

(ii) SUCCESSOR(K): Find the least element in the set S with key value greater than K;

(iii) PREDECESSOR(K): Find the greatest member in the set S with key value less than K;

(iv) SUBSET(K_1, K_2): Find (and then produce) the list of the elements from the set S whose key values lie between K_1 and K_2.

For convenience, we will say that a data structure has an *overall worst-case retrieval complexity* $O[f(M, N)]$ iff the worst-case processing time on Random Access Machines for retrieval operations (i)–(iii) is $O[f(M, N)]$ and the worst-case runtime for SUBSET queries is proportional to $f(M, N)$ plus the size of the retrieved subset.[1] If the same data structure additionally supports insertions and deletions in worst-case processing time $O[f(M, N)]$ then this data structure will be said to have an overall *dynamic* worst-case retrival complexity $O[f(M, N)]$.

AVL trees [1, 7], 2–3 trees [2], and bounded-balance trees [11] are some examples

* *Present address*: Computer Science Department, SUNY, Albany, N.Y. 12222.

[1] Following Knuth [8], a function will be said to be $O(f)$ iff it is no more than proportional to f, $\Omega(f)$ iff it is at least proportional to f, and $\Theta(f)$ iff it is both $\Omega(f)$ and $O(f)$.

0022-0000/84 $3.00

of data structures which use space $O(N)$ and have an overall dynamic worst-case retrieval complexity $O(\log N)$. It is well known that better overall worst-case retrieval complexities are impossible when a Random Access Machine searches an ordered list of real numbers [7]. Nevertheless, some types of improvements do exist. Interpolation search [5, 12, 13, 25] will produce an expected retrieval time $O(\log \log N)$ when the keys are real numbers generated by the uniform probability distribution. Willard's modification of interpolation search [18, 19] will produce an expected retrieval time $O(\log \log N)$ when the keys are generated by a nonuniform distribution which is unknown to the search algorithm but differentiable. References [4, 14] showed that MEMBER queries can be executed more efficiently than the best results known for any predecessor query. The stratified trees of [15–17] will produce an overall dynamic worst-case retrieval complexity $O(\log \log M)$ when the keys are nonnegative integers less than some fixed bound M.

The latter data structure would be extremely useful if it did not have one serious disadvantage. Rather than use space $\Theta(N)$, similar to the other data structures cited in this section, the early versions of [15, 17] required space $\Theta(M \log \log M)$. The latter quantity is significantly more expensive than $\Theta(N)$ in applications where M is much larger than N.

The need for more space-efficient data structures was indicated in [15]. Van Emde Boas developed a modified tree data structure that uses space $\Theta(M)$ in [16]. To help motivate our research, Section 6 of this article will show that each of the implementations [9, 15, 16, 17] of a stratified tree (even in a static environment) uses significantly more space than $\Theta(N)$ in applications, where M is greater than N. These difficulties lead us to propose a new data structure in this article, called the *q-fast* trie, which uses space $\Theta(N)$ and time $O(\sqrt{\log M})$ for overall worst-case dynamic retrieval. Thus, by allowing a slight increase in asymptotic retrieval time, we develop a modified version of the concepts from [15–17] which has the desired memory space usage $\Theta(N)$.

Our results are likely to be useful because the time $O(\sqrt{\log M})$ is asymptotically less than the complexity $O(\log N)$ of AVL, 2–3, and bounded balance trees for virtually all values of the parameters M and N, *including* most cases where M is much greater than N. For instance, if $M = 2^{100}$ and $N = 2^{20}$ then $\sqrt{\log M} = 10$ and $\log N = 20$.

The notion of a q-fast trie was introduced by Willard in [21], and Johnson later observed that analogs to the first part of the trie transformation will reduce from $O(M)$ to $O(N \cdot M^{\varepsilon}/\varepsilon)$ the memory of the quite different data structures [6] with the process time $O((1/\varepsilon) \log \log$ (distance to K's closest neighbor)). Other quite recent work includes Karlson's independent observation [10] of an idea similar to q-fast tries, and Willard's proof [22] that a time $O(\log \log M)$ is indeed possible in the space $O(N)$ for the special case of static sets. No improvement of the retrieval time of q-fast tries is known without a sacrifice of either the memory space or the cost of insertions or deletions. It remains an open question whether such improvements are possible. Willard discusses the q-fast trie's implications for multi-dimensional retrieval in [23, 24].

This paper will be devided into five sections. Section 2 further motivates the research of this article by explaining the disadvantages of conventional tries. Section 3 introduces a simplified version of our proposed data structure, called the *p-fast* trie, and shows how it makes possible a combination of overall worst-case retrieval complexity $O(\sqrt{\log M})$ and worst-case memory space usage $O(N \sqrt{\log M}\ 2^{\sqrt{\log M}})$. Section 4 introduces the more sophisticated q-fast trie data structure, which reduces memory space occupancy to $O(N)$ without increasing asymptotic retrieval time. Section 5 shows that record insertion and deletion operations can be performed in worst-case time $O(\sqrt{\log M})$ under either the p- or q-fast trie data structures. Section 6 explains why these data structures are preferable to stratified trees.

2. The Disadvantage of Conventional Trees

As is suggested by its name, the new fast trie concept is a modified form of trie. According to Knuth [7], the latter data structure was first formally proposed by Fredkin [3]. We will say that a trie has *size*(h, b) if it has height h and a branching factor b.

All keys, in a conventional trie of size (h, b) can clearly be regarded as nonnegative integers less than b^h. In our discussion, the sequence $K_1, K_2, \ldots, K_h$ will denote the h-digit expansion of the integer-key K when it is written as a number in base b. In a trie of size (h, b), this notation clearly implies that the key K is stored at

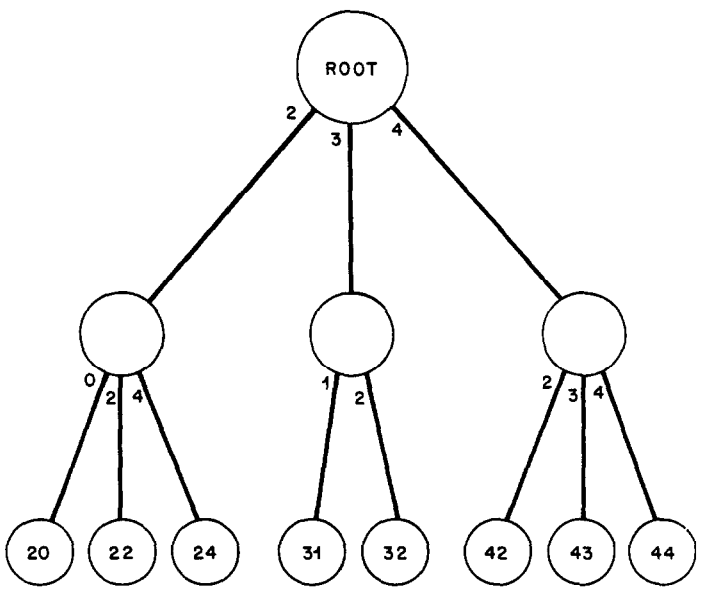

FIG. 1. An example of a trie with height $h = 2$ and branching factor $b = 5$. All keys at the leaf-level are written as two-digit numbers in base 5.

the K_hth child of the K_{h-1}th child of the $\cdots K_1$th child of the trie's root (as shown in Fig. 1). In our discussion, the symbol $\text{CHILD}_v(D)$ will denote the pointer, stored inside the internal node v, indicating the address of its Dth child. For instance in Fig. 1, $\text{CHILD}_{\text{root}}(2)$ would point to the root's *leftmost existing* child. Note that the root in Fig. 1 does not contain any children corresponding to the digits zero and one. By convention, we will therefore store element NULL in $\text{CHILD}_{\text{root}}(0)$ and $\text{CHILD}_{\text{root}}(1)$.

In most of our discussion, we will treat the retrieval complexity of a trie as a function of its size (h, b). A conventional topdown retrieval algorithm will clearly perform queries of types (i)–(iii) by visiting $\Theta(h)$ nodes of the trie and will perform type-(iv) queries by visiting a quantity of nodes proportional to h plus the size of $\text{SUBSET}(K_1, K_2)$. The retrieval time for a type (i) query is nevertheless very different from that of types (ii)–(iv) queries because the former needs only $\Theta(1)$ units of processing time to visit each node while the latter three queries require time $\Theta(\text{b})$ to visit a node, in the worst case.

The reason for the distinction is best illustrated with an example. Suppose that the set S consists of the single integer $b^h - 1$ and that it is represented by a trie of size (h, b), as illustrated in Fig. 2. Consider the complexity of a query that seeks to find the SUCCESSOR of the negative integer -1.

In order to determine that the integer key $b^h - 1$ is in fact this successor, our retrieval algorithm will need to verify that each of this key's ancestors has the NULL values stored in its first $b - 1$ CHILD fields. This retrieval algorithm will therefore require time $\Theta(b)$ to process each of these h ancestor nodes. Using this observation

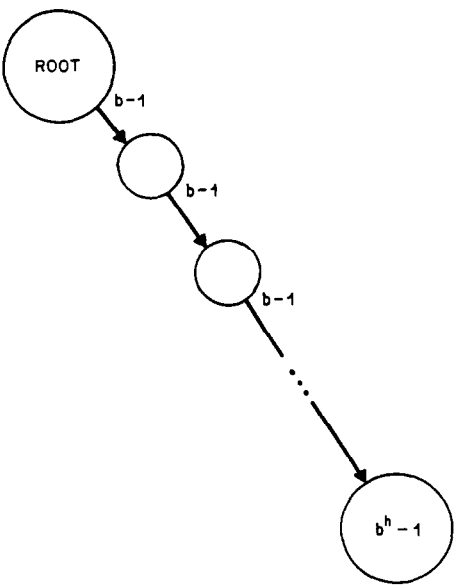

FIG. 2. A trie of size(h, b) whose only leaf corresponds to the integer $b^h - 1$.

and its analogs for PREDECESSOR and SUBSET queries, the overall worst-case retrieval complexity for conventional tries of size (h, b) is easily seen to be $\Theta(hb)$.

This article was motivated by the observation that it is possible to improve the performance of a conventional trie by slightly modifying this data structure. Our first proposal, the *p*-fast trie, has a worst-case overall-retrieval complexity $O(h + \log b)$, and uses the same amount of space asymptotically as a conventional trie. All our other results will follow from a series of elaborate applications of the concept of a *p*-fast trie. It should be stressed that a complexity of $O(h + \log b)$ is crucial to our analysis. Even a complexity $O(h \cdot \log b)$ would be insufficient for the asymptotic improvements that are discussed in the subsequent sections.

3. Definition and Analysis of *P*-Fast Tries

In this section, we define the data structure of a *p*-fast trie and then demonstrate its efficiency. Let S denote the set of integer keys stored in our trie. Our tries will always be structured so that their sets of internal nodes are as small as possible. That is, we store an internal node v in the trie T if and only if it is the ancestor of some elements of S. In the previous literature on tries, this compressed-storage representation has been used to conserve space. Our *p*-fast tries are somewhat unusual because their search algorithm will also need compression to guarantee its run-time efficiency, and indeed even its correctness (because the search algorithm would otherwise be misled by the trie's INNERTREE fields, as will become apparent later).

By definition, *p*-fast tries will be structured so that each internal node v contains the usual CHILD-pointer fields that have been traditionally associated with tries plus the following three new fields:

LOWKEY(v): This field will be a pointer to that leaf containing the smallest key descending from v.

HIGHKEY(v): This field will be a pointer to that leaf containing the largest key descending from v.

INNERTREE(v): This field will consist of a binary tree of worst-case height $O(\log b)$ that represents the set of digits D such that $\text{CHILD}_v(D) \neq \text{NULL}$.

For instance, in Fig. 1, LOWKEY(root) would be a pointer to the leaf 20, HIGHKEY(root) would be a pointer to the leaf 44, and INNERTREE(root) would represent the set $\{2, 3, 4\}$.

Also by definition, each leaf of a *p*-fast trie will be required to contain one pointer to the leaf which lies to its immediate left and another pointer to the leaf at its right. The sequence of keys thus completed will be called the trie's *ordered list*.

It is useful to introduce one further definition before we begin our analysis of the retrieval properties of *p*-fast tries. Define a CLOSEMATCH(K) to be a query that:

(i) returns the address of the record storing key K in trie T (if the trie actually contains K);

FIG. 3. *Algorithm* RETRIEVE(K).

Comment: Let $K_1, K_2, \ldots, K_h$ denote the expansion of key K as an h-digit number in base b. This algorithm will search a p-fast trie T of height h to find a key CLOSEMATCH(K).

Step 0: Set v equal to the root of trie T and i to 1.

Step 1: IF $\text{CHILD}_v(K_i) \neq \text{NULL}$ THEN $v \leftarrow \text{CHILD}_v(K_i)$ ELSE GO TO STEP 2;

IF $i = h$ THEN key has been found and algorithm should immediately terminate by returning the pointer v ELSE $i \leftarrow i + 1$ and GOTO the beginning of Step 1.

Step 2: Let D denote:

(i) the least digit greater than K_i satisfying $\text{CHILD}_v(D) \neq \text{NULL}$ (if such a digit exists);

(ii) and otherwise the greatest digit less than K_i satisfying $\text{CHILD}_v(D) \neq \text{NULL}$.

Calculate the value of D by performing the obvious binary search inside INNERTREE(v).

Step 3. IF $D > K_i$ THEN advance to the address $\text{LOWKEY}(\text{CHILD})_v(D))$ (where one suitable CLOSEMATCH key can be retrieved) ELSE retrieve CLOSEMATCH (K) from the address $\text{HIGHKEY}(\text{CHILD}_v(D))$.

END OF PROGRAM.

(ii) and otherwise retuns the address of either PREDECESSOR(K) or SUCCESSOR(K).

The first half of this section will prove that p-fast tries of size(h, b) have a worst-case complexity $\Theta(h + \log b)$ for CLOSEMATCH requests, and the second half will apply this result to demonstrate that these tries support an *overall* worst-case retrieval complexity $\Theta(\sqrt{\log M})$.

Our algorithm for performing CLOSEMATCH(K) queries is illustrated in Fig. 3. It consists of three steps. The first step will begin at the root of the p-fast trie and will perform a conventional topdown tree-walk that visits all of K's ancestors by following the obvious path indicated by the relevant CHILD-pointers. If the key K is actually stored in the trie T then this walk will eventually reach K, and our search algorithm will then terminate. If K is not stored in T then our topdown three-walk will eventually encounter a NULL pointer. In this latter case, our retrieval algorithm will complete its search by executing two further steps, which employ the novel characteristics of p-fast tries.

Step 2, the first of these new steps, will take advantage of the trie's special INNERTREE fields. In order to describe this procedure, we let v denote the last node that was visited by step 1, and $i - 1$ the depth of v. Our previous notation then implies that step 1 terminated because $\text{CHILD}_v(K_i) = \text{NULL}$. Let D denote:

(i) the least digit greater than K_i satisfying $\text{CHILD}_v(D) \neq \text{NULL}$ (if such a digit exists);

(ii) and otherwise the greatest digit less than K_i satisfying $\text{CHILD}_v(D) \neq \text{NULL}$.[2]

A search thorough INNERTREE(v) can determine the value of D in worst-case runtime $O(\log b)$. Step 2 will consist of such a search. Step 3 of our retrieval

[2] From the definition of p-fast tries, it can be easily proven that some digit must satisfy either condition (i) or (ii). This observation is important because otherwise D could be undefined.

algorithm for CLOSEMATCH will use a slightly different procedure in the two cases where D is (and is not) greater than K_i. In the first case, it will retrieve SUCCESSOR(K) by advancing to the position LOWKEY(CHILD)$_v(D)$). In the latter case, it will retrieve PREDECESSOR(K) by advancing to the analogous position HIGHKEY (CHILD$_v(D)$).

LEMMA 1. *The algorithm for* CLOSEMATCH *retrievals (formally defined in Fig. 3) will search a p-fast trie of size* (h, b) *in worst-case time* $O(h + \log b)$; *and this trie will use space* $O(hbN)$ *for representing a set of* N *records.*

Proof. It is easy to see that the algorithm's three steps have respective worst-case complexities $O(h)$, $O(\log b)$, and $O(1)$. Thus, CLOSEMATCH retrievals have a total worst-case complexity $O(h + \log b)$. The worst-case space of a p-fast trie of size (h, b) is clearly bounded by $O(hbN)$ (since each internal node occupies space $\Theta(b)$ and a trie of size(h, b) will require no more than hN internal nodes). Q.E.D.

LEMMA 2. *P-fast tries of size*(h, b) *have an overall worst-case retrieval complexity* $O(h + \log b)$.

Proof. Clearly a CLOSEMATCH retrieval supplies more information than a MEMBER retrieval. Therefore, MEMBER retrievals must have complexities at least as good as the measurement $O(h + \log b)$ for CLOSEMATCH operations. Next, we use the fact that each leaf, in a p-fast trie, contains pointers to its predecessor and successor in the trie's "ordered list." These pointers clearly assure that the complexity of SUCCESSOR and PREDECESSOR queries need exceed that of CLOSEMATCH queries by no more than an asymptotically inconsequential additive constant and that SUBSET queries can be performed in a complexity that exceeds CLOSEMATCH queries by no more than an amount proportional to the size of SUBSET(K_1, K_2). Hence, all these queries will respect the "overall" worst-case retrieval bound $O(h + \log b)$. Q.E.D.

THEOREM 1. *It is possible to obtain a combination of overall worst-case retrieval complexity* $O(\sqrt{\log M})$ *and worst-case memory occupancy* $O(N\sqrt{\log M}\, 2^{\sqrt{\log M}})$ *when a p-fast trie represents a set* S *of cardinality* N, *whose keys are nonnegative integers less than some specified bound* M.

Proof. These results follow by applying Lemmas 1 and 2 to a p-fast trie whose size has components $h = \lceil \sqrt{\log M} \rceil$ and $b = \lceil 2^{\sqrt{\log M}} \rceil$. Q.E.D.

Remark 1. The result above is a significant improvement over stratified trees [15–17] in applications where $M \gg N$ because its memory space is smaller than the counterparts $O(M \log \log M)$ and $O(M)$ for stratified trees. Van Emde Boas has mentioned that a retrieval complexity $O(\sqrt{\log M})$ is possible for an alternative data structure which has a branching factor 2^d for all nodes at depth d. Although [15] attributed space $\Theta(M)$ to this data structure, it is not difficult to see that the better upper bound $O(N2^{\sqrt{2\log M}})$ is possible when $M \gg N$. This quantity is not as good as

the space $O(N\sqrt{\log M}\, 2^{\sqrt{\log M}})$ of Theorem 1. The next section of this paper will show how to further reduce memory to $O(N)$ without increasing asymptotic retrieval time.

4. Definition and Analysis of Q-Fast Tries

In this section, we will propose a modified version of the fast trie concept which prunes the bottom of this tries to conserve memory. This pruning is desirable because the bottoms of fact tries have no effect on asymptotic retrieval time although they significantly increase memory costs.

The concept of pruning is not new. Knuth has mentioned the possibility of pruning conventional tries [8]. Van Emde Boas [16] has observed that this method will reduce the memory space of stratified trees from $\Theta(M \log\log M)$ to $\Theta(M)$. Pruning can also be profitably applied to many other data structures (e.g., polygon trees [20]). The contribution of this section consists of observing that pruning is exceptionally useful for fast tries. When $N \ll M$, this technique will make possible a retrieval complexity $O(\sqrt{\log M})$ in $O(N)$ space.

One preliminary definition must be introduced before we can define the new concept of q-fast trie. Let S denote an initial set, S^* an ordered set of keys: $0 = K_1^* < K_2^* \cdots < K_L^* < M$, S_i the subset $\{K \in S : K_i^* \leqslant K < K_{i+1}^*\}$ for $i < L$, and S_L the subset $\{K \in S : K \geqslant K_L^*\}$. Then the set S^* will be said to form a *c-partition* of set S iff each set S_i has a cardinality between c and $2c - 1$.

A *q-fast* trie of size(h, b, c) will be defined as a data structure which represents a set S by employing two substructures, called the upper and lower parts. Its upper section will be a p-fast trie T of size(h, b) that represents some set S^*, that forms a c-partition of S. Its lower part will be a forest of 2–3 trees whose ith tree T_i represents the set S_i. The leaf for key K_i^*, in trie T, will contain a pointer to the corresponding tree T_i. Also, the leaves in the forest of 2–3 trees will form an ordered list with each leaf containing one pointer to the leaf with the next highest key value and another

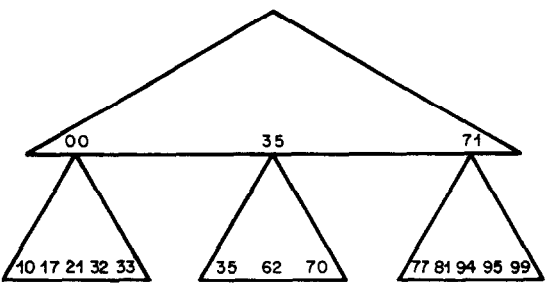

FIG. 4. An example of a Q-fast trie of size (2, 10, 3). The top triangle represents the upper section of this data structure, which is a P-fast trie. The bottom triangles are the forest of 2–3 trees, in its lower section. The numbers, in each triangle, are the values stored in the leaves.

pointer to the leaf with the next lowest key value. An example of a q-fast trie is illustrated in Fig. 4.

LEMMA 3. *The worst-case complexity for* CLOSEMATCH *retrievals in a q-fast trie of size* (h, b, c) *is* $O(h + \log b + \log c)$, *and this trie's worst-case memory usage will be* $O(N(1 + hb/c))$ *when it represents a set of* N *records.*

Proof. The natural algorithm for retrieving the CLOSEMATCH of Key K will consist of a two-part procedure, which first searches the upper part of the q-fast trie to find that Key K_i^* which is the predecessor of K in the set S^*, and then searches the particular tree T_i, which the lower part associates with Key K_i^*, to find one Key from set S that is a CLOSEMATCH for K. By Lemma 1, the first step of this procedure will have worst-case complexity $O(h + \log b)$. The logarithmic retrieval complexity of 2–3 trees [2] assures that the second step of this procedure will have complexity $O(\log c)$. Hence, CLOSEMATCH queries will have a worst-case complexity $O(h + \log b + \log c)$.

As a consequence of Lemma 1, the upper section of a q-fast trie will occupy worst-case memory $O(Nhb/c)$. Its lower section will occupy space $O(N)$, as a consequence of the linear-momory-space property of 2–3 trees [2]. Thus, q-fast tries of size(h, b, c) will clearly occupy memory space $O(N(1 + hb/c))$. Q.E.D.

LEMMA 4. *Q-fast tries of size*(h, b, c) *have an overall worst-case retrieval complexity* $O(h + \log b + \log c)$.

Proof. The proof of Lemma 4 is very similar to the proof of Lemma 2. In particular, Lemma 4 follows from Lemma 3 by the same reasoning that made Lemma 2 previously follow from Lemma 1. Q.E.D.

THEOREM 2. *It is possible to obtain a combination of overall worst-case retrieval complexity* $O(\sqrt{\log M})$ *and worst-case memory occupancy* $O(N)$ *when a q-fast trie represents a set* S *of cardinality* N, *whose keys are nonnegative integers less than some bound* M.

Proof. These results follow by applying Lemmas 3 and 4 to a q-fast trie whose size has components $h = \lceil \log M \rceil$, $b = 2^{\lceil \log M \rceil}$ and $c = hb$. Q.E.D.

5. DYNAMIC OPERATIONS ON FAST TRIES

In this section we will explain how to insert and delete records efficiently under the p- and q-fast trie data structures. In our discussion, we assume that one can allocate or deallocate the memory space of any internal node of a trie in $O(1)$ units of time. This assumption is reasonable, since all unused memory fields can be placed in a stack. We need this capability because the insertion and deletion of keys, in p-fast tries, can sometimes trigger the addition or removal of internal nodes. We will begin

FIG. 5. DELETE(K).

Comment: This algorithm will delete key K from the specified p-fast trie.

Step 1: Find the key K which should be deleted from the p-fast trie. Let K^- and K^+ denote the predecessor and successor of K, respectively. Change the trie's ordered list so that K^+ becomes the successor to K^- (thereby removing K from this list).

Step 2: For each ancestor v of key K DO:

IF LOWKEY(v) = K THEN LOWKEY(v) $\leftarrow K^+$

IF HIGHKEY(v) = K THEN HIGHKEY(v) $\leftarrow K^-$.

Step 3: DEALLOCATE the memory space of the record storing key K and of every ancestor v of K satisfying simultaneously the conditions LOWKEY(v) = K^+ and HIGHKEY(v) = K^-.

Step 4: Let v denote the highest node whose memory space was deallocated in Step 3 and f the parent of v. Delete the digit associated with v from INNERTREE(f) by using the AVL algorithm.

END OF PROGRAM.

our discussion by analyzing the p-fast trie data structure, and then we will turn our attention to q-fast tries.

The subtle aspect of the analysis of p-fast tries consists of proving a worst-case complexity $O(h + \log b)$ for performing all the adjustments on the INNERTREE fields, which are needed immediately after the insertion or deletion of a key. We will employ the AVL algorithm [1, 7] to assure logarithmic height for all INNERTREEs and a logarihmic complexity for inserting and deleting digits in them. Our algorithm for deleting records from p-fast tries of size(h, b) is illustrated in Fig. 5. The analysis of this procedure is straightforward: Its first three steps clearly consume time $O(h)$; its last step consumes worst-case time $O(\log b)$ because of our use of the AVL method. Hence, any record can be deleted from a size(h, b) trie in worst-case time $O(h + \log b)$. Insertions are performed by the approximate inverse of this algorithm and have the same complexity.

It is also fairly easy to see that insertions and deletions have a worst-case complexity $O(h + \log b + \log c)$ in q-fast tries of size(h, b, c). Figure 6 illustrates our algorithm for performing these operation. The first step of this procedure will search the the upper part of the q-fast trie to find that tree T_i, in the lower part, where the

FIG. 6. INSERT/DELETE(K, I)

Comment: We assume that I is an instruction either to insert a key K into or to delete it from a q-fast trie. This algorithm will perform this command.

Step 1: Note that the upper part of a q-fast trie is a p-fast trie. Apply the algorithm of Section 3 to find the predecessor K_i^* of key K in set S^*.

Step 2: Use the insertion and deletion algorithm from [2] either to add key K into tree T_i, or to remove it from this tree, as specified by instruction I.

Step 3: If Step 2 caused the tree T_i to grow and reach a cardinality 2c then split it into two equally sized trees. If the cardinality of this tree was reduced to $c - 1$ then merge it with one of its neighbors and split the resulting tree into two equally sized parts iff its cardinality is $2c$ or greater. The operations of split and merge should be performed with the algorithms from [2].

Step 4: Adjust set S^* to reflect the changes from Step 3 and also accordingly update the upper part of the q-fast trie by using the insertion and deletion algorithms for p-fast tries.

END OF PROGRAM.

record insertion or deletion should take place; this step will employ the PREDECESSOR search algorithm from Section 3 and therefore consume time $O(h + \log b)$. The second step will insert (or delete) the relevant record in tree T_i in worst-case time $O(\log c)$ by using the insertion/deletion algorithms of [2]. If the second step causes the size of tree T_i to exceed $2c - 1$ then the third will split it in half; similarly, if this tree shrinks to size below c then the third step will correct this imbalance by first merging it with one of its two immediate neighbors and then splitting the resulting tree in half if its size exceeds $2c - 1$. The tree splitting and merging operations of [2] assure that step 3's worst-case time will not exceed $O(\log c)$. Finally, the last step of the q-fast trie insertion and deletion algorithm will adjust the set S^* to reflect the changes made in the forest of 2–3 trees by the third step. More specifically, this procedure will insert a new key into S^* whenever step 3 has split a 2–3 tree, and it will delete a key whenever step 3 has merged two trees; it will then accordingly adjust the upper part of the q-fast trie in worst-case time $O(h + \log b)$, by employing our previously mentioned algorithm for insertion and deletion operations in p-fast tries. Thus, the analysis above shows that the procedure in Fig. 6 will perform any insertion or deletion operation in worst-case time $O(h + \log b + \log c)$. The following theorem is the main result of this paper:

THEOREM 3. *Let S denote a set of N nonnegative integer keys, less than some specified bound M. Then it is possible to provide:*

(A) *a combination of overall dynamic worst-case retrieval complexity $O(\sqrt{\log M})$ and worst-case space $O(N \sqrt{\log M}\ 2^{\sqrt{\log M}})$ with the p-fast trie data structure;*

(B) *and a combination of overall dynamic worst-case retrieval complexity $O(\sqrt{\log M})$ and worst-case space $O(N)$ with the q-fast trie data structure.*

Proof. The first result follows by applying Lemmas 1 and 2 and this section's analysis of the insertion and deletion complexities of p-fast tries to a data structure whose size has components $h = \lceil \sqrt{\log M} \rceil$ and $b = \lceil 2^{\sqrt{\log M}} \rceil$. The second result follow analogously by applying Lemmas 3 and 4 and this section's analysis of q-fast tries to a data structure whose size has components $h = \lceil \sqrt{\log M} \rceil$, $b = \lfloor 2^{\sqrt{\log M}} \rfloor$, and $c = hb$. Q.E.D.

6. COMPARISON WITH STRATIFIED TREES

Recall that the first section of this paper indicated that fast tries were motivated by the concept of stratified trees [15–17] and were designed to occupy significantly less memory space than these predecessors. In this section, we will discuss this topic in greater detail and explain how the space disadvantage of stratified trees arises. (Readers should bear in mind that stratified trees have a better time-complexity precisely because of their space disadvantage.) Although the discussion in this section

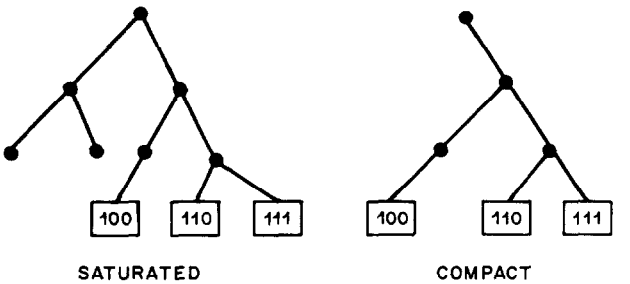

FIG. 7: The difference between a compact and saturated binary trie. Dots correspond to internal nodes and boxes to leaves in this figure. Note that the tries represent the set of binary numbers {100, 110, 111} in this example.

does not require the reader's familiarity with stratified trees [17], it may be helpful to have read this reference.

In our discussion, a trie will be said to be a *compact* representation of a set S iff it contains only those internal nodes v that are ancestors of some member of S, and a *saturated* representation iff it contains every possible internal node, regardless of whether or not a node's leaf-descendents include any member of the set S. Figure 7 illustrates the difference between compact and saturated tries.

The memory space $\Theta(M \log\log M)$ of stratified trees [15, 17] follows from the observation that they are a special form of saturated binary trie that has $\Theta(M)$ internal nodes and allocates $\log\log M$ space to each node. Note that a compact binary trie will need no more than $N \log M$ internal nodes to represent a set of N nonnegative integer keys which are bounded by M. It would therefore first appear that some modified form of the stratified tree concept could represent this set in no more than space $O[N(\log M)(\log\log M)]$. In this section, we will show that such a modification is impossible and that the most compressed possible stratified trees must occupy significantly more memory space.

In our discussion, we will employ some of the notation and terminology from [17]. For simplicity, we will assume that the upper bound M for the set S is a quantity of the form 2^{2^k}, for some integer k. Our saturated binary trie will therefore have height equal to 2^k. For any node v, we let *rank*(v) denote the largest integer i such that 2^i divides the height of v. Recall that all elements of the set S are represented as leaves throughout this paper. A node v will be defined to be *critical* with respect to the integer j iff the ancestors of S include at least two internal nodes which descend from v and lie precisely j levels below it. Finally, define a node v to be *active* iff either

(a) it is critical with respect to the integer $2^{\text{rank}(v)}$, or

(b) the ancestor of v, located above it by precisely $2^{\text{rank}(v)}$ levels, is critical with respect to $2^{\text{rank}(v)}$.

We will use the term type (a) active (resp. type (b) active) to describe the subset of active nodes that satisfy condition (a) (resp. (b).

The active internal nodes play a major role in the retrieval algorithm of [17]; each active internal node may be visited during the search of a stratified tree. Therefore memory space must be allocated for each active internal node; a lower bound on the space needed by any implementation of a stratified tree can thus be calculated by counting its number of active nodes. Our analysis will show that stratified trees occupy significatly more memory space than p-fast, q-fast and compact-binary tries, intuitively because internal nodes are frequently type (b) active without being the ancestors of any element of the set S.

THEOREM 4. *For simplicity, let M denote an integer of the form 2^{2^k}, S a set of N distinct nonnegative integer keys less than M, and $A(N, M)$ the number of active internal nodes in a stratified tree that represents the set S. Then $A(N, M)$ will respect the following lower bounds:*

(I) *The worst-case value of $A(N, M)$ will always be bounded below by*[3] $N^{3/4}M^{1/4}/2$.

(II) *If $N = M^{1-2^i}$, for some nonnegative integer i, and if the set S consists of N keys generated by distribution where all combinations of key values are equally likely, then the expected value of $A(N, M)$ will be bounded below by $\Omega(\sqrt{NM})$. (Here $(1 - e^{-1/2})^2$ is one feasible constant for the Ω-notation.)*

Proof of Assertion I. Let i denote the greatest integer such that $2^{2^{k-2^i}} \geqslant N/2$. Then it is possible to construct a set S whose stratified tree contains $N/2$ internal nodes at depth $2^k - 2^i$ which are critical with respect to the integer 2^{i-1}. Note that each descendant, located 2^{i-1} levels below such a critical node, must be type (b) active and that $2^{2^{i-1}}$ such descendants lie below each such critical node. Therefore, the total number of active nodes must be bounded below by $N \cdot 2^{2^{i-1}}/2$.

In order to complete our proof, we must show that $2^{2^{i-1}} > (M/N)^{1/4}$. First note that the definition of i (in the first sentence of this proof) implies $2^{2^{k-2^{i+1}}} < N/2$. This observation, together with the fact that $M = 2^{2^k}$, implies $2^{2^{i-1}} > (M/N)^{1/4}$. Hence, we may conclude from the last paragraph that the worst-case value of $A(N, M)$ is bounded below by $N^{3/4}M^{1/4}/2$. Q.E.D.

Proof of Assertion II. The assumption $M = 2^{2^k}$ and $N = M^{1-2^{-i}}$ clearly implies $N = 2^{2^k - 2^{k-i}}$, which in turn implies that precisely N nodes lie at a depth of $2^k - 2^{k-i}$ in a saturated binary trie. Employing this cardinality, it is easy to prove that under a distribution where all combinations of N keys are equally likely to belong to the set S, the probability that a node of depth $2^k - 2^{k-i}$ has elements of S descending simultaneously from both of its children is $\geqslant (1 - e^{-1/2})^2$. (We do not prove this fact because it is trivial and furthermore any fixed positive constant will justify the asymptote of Assertion II.) This observation implies that $(1 - e^{-1/2})^2 N$ is a lower bound on the expected number of nodes at the depth $2^k - 2^{k-i}$ which are critical with

[3] This lower bound is a highly conservative estimate for some values of N; and it is very close to $A(N, M)$'s actual worst-case value for a surprisingly broad spectrum of other values.

respect to 2^{k-i-1}. Each such critical node will be associated with $2^{2^{k-i-1}}$ distinct type (b) active nodes, located 2^{k-i-1} levels below it. Hence the total expected number of active nodes is bounded below by $(1-e^{-1/2})^2 N \cdot 2^{2^{k-i-1}}$, a quantity which equals $(1-e^{-1/2})^2 \sqrt{NM}$ by the assumptions $M = 2^{2^k}$ and $N = 2^{2^k - 2^{k-i}}$. Q.E.D.

Remark 2. A straightforward generalization of Assertion II will indicate the existence of three positive constants $C_1 < C_2$ and C_3 such that if N satisfies $C_1 M^{1+2^{-i}} < N < C_2 M^{1-2^{-i}}$ then $C_3 \sqrt{NM}$ is a lower bound on the expected number of active nodes. Although our discussion technically focused on the particular proposal of Van Emde Boas *et al.*]17], it is easy to apply similar reasoning to the related data structures in [9, 15, 16]. These difficulties in controlling memory costs are the reason we have proposed q-fast tries as an alternative.

7. Conjecture

We conjecture that the results in this article are optimal insofar as no data structure uses space $O(N)$ and improves upon a *dynamic* overall worst-case retrieval complexity $O(\sqrt{\log M})$. However, several methods come extremely close to improving upon our results by producing better asymptotic results for other cases. Thus interpolation search [5, 12, 13, 25] and its modification for nonuniform distributions [18, 19] will produce an expected runtime $O(\log \log N)$ on a data structure which uses precisely N units of space, and another paper by Willard [22] shows static data structures exist in $O(N)$ space with a worst-case retrieval complexity $O(\log \log M)$. The latter method is less efficient than fast tries in worst-case insert-delete costs.

Our conjecture does not claim that the coefficient in this paper are the best possible results. If the data structure of a p-fast trie is modified so that its INNERTREE fields are changed to stratified trees then the overall complexity of a size(h, b) trie would be reduced to $O(h + \log \log b)$. A similar modification of the q-fast trie data structure would produce a complexity $O(h + \log \log b + \log c)$. These modifications would not change the asymptotic results of Theorem 3 because the best values of h, b, and c would still require $O(\sqrt{\log M})$ time complexity for a data structure occupying $O(N)$ worst-case space. A serious disadvantage of such modified versions of p- and q-fast tries is that their runtime coefficients are typically worse than those of the tries in Sections 3 and 4 (because INNERTREEs of typical cardinalities are more efficiently represented as binary trees). Such modifications would therefore usually be impractical, but other modified data structures may produce better coefficients.

8. Conclusion

We have proved the existence of a data structure whose memory space is comparable to AVL, bounded-balance and 2–3 trees [1, 2, 7, 11], but which has a

better overall worst-case dynamic retrieval complexity when $\sqrt{\log M} < \log N$. Our data structure is intended for files stored in main memory and processed by a Random Access Machine.

ACKNOWLEDGMENTS

I would like to thank Y. Edmund Lien, Daniel D. Sleator, and Eric Wolman for their suggestions on presentation.

Note added in proof. Our time complexity $O(\sqrt{\log M})$ should not be confused with the complexity $O(1)$ which Atjai, Fredman, and Komlos recently published at the 1983 IEEE 24th FOCS symposium. The latter article assumes an unusually broad class of operations are executable in unit time, and even then the main result is valid only when M meets a stringent size constraint. (Atjai *et al.* technically require $M \geqslant 2^N$ although the lower bound can be reduced somewhat if the coefficient associated with $O(1)$ complexity is increased.) Atjai *et al.*'s main contribution is the implication that a proof of a lower bound matching our upper bound $O(\sqrt{\log M})$ probably requires $N \cong M^k$ for some fixed k.

REFERENCES

1. G. M. ADEL'SON-VEL'SKII AND Y. M. LANDIS, An algorithm for the organization of information, *Soviet Math. Dokl.* **3** (1962), 1259–1262.
2. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, "The Design and Analysis of Computer Algorithms," Addison–Wesley, Reading, Mass., 1974.
3. E. FREDKIN, "Trie Memory," *Comm. ACM* **3** (1962), 490–499.
4. M. L. FREDMAN, J. KOMOLÓS, AND E. SZEMERÉDI, Storing a space table with $O(1)$ worst case access times, *in* "Proceedings of the 23rd IEEE Symposium on the Foundations of Computer Science," pp. 165-169, 1982.
5. G. H. GONNET, L. D. ROGERS, AND J. A. GEORGE, An algorithmic and complexity analysis of interpolation search, *Acta Inform.* **13** (1980), 39–52.
6. D. B. JOHNSON, A priority queue in which initialization and queue operations take $O(\log \log D)$ time, *Math. System Theory* **15** (1982), 295–309.
7. D. E. KNUTH, "The Art of Computer Programming, Vol. 3; Sorting and Searching," Addison–Wesley, Reading, Mass., 1973.
8. D. E. KNUTH, Big omicron, big omega, and big theta, *SIGACT News* **8** (1976), 18–24.
9. D. E. KNUTH, Widely disseminated classroom notes on stratified trees, 1979.
10. R. KARLSON, Unpublished notes, University of Waterloo, 1982.
11. J. NIEVERGELT AND E. M. REINGOLD, Binary search trees of bounded balance, *SIAM J. Comput.* **2** (1973), 33–43.
12. Y. PEARL, A. ITAI, AND H. AVNI, Interpolation search—A log log N search, *Comm. ACM* **21** (1978), 550–554.
13. Y. PEARL AND E. M. REINGOLD, "Understanding the complexity of interpolation search, *Inform. Process. Lett.* **6** (1977), 219–222.
14. R. E. TARJAN AND A. C. YAO, Storing a sparse table, *Comm. ACM* **22** (1979), 606–611.
15. P. VAN EMDE BOAS, Preserving order in a forest in less than logarithmic time, *in* "Proceedings of the 16th Annual Symposium on the Foundations of Computer Science," pp. 75–84, 1975.
16. P. VAN EMDE BOAS, Preserving order in a forest in less than logarithmic time and linear space, *Inform. Process. Lett.* **6** (1977), 80–82.

17. P. VAN EMDE BOAS, R. KAAS, AND E. ZIJLSTRA, Design and implementation of an efficient priority queue, *Math. Systems Theory* **10** (1977), 99–127.
18. D. E. WILLARD, A log log N search algorithm for nonuniform distributions, *in* "Proceedings of the TIMS–ORSA Conference on Applied Probability–Computer Science Interface," Vol. II, pp. 1–9, 1981.
19. D. E. WILLARD, Searching nonuniformly generated files in log log N runtime, *SIAM J. Comput.*, in press.
20. D. E. WILLARD, Polygon retrieval, *SIAM J. Comput.* **11** (1981), 149–166.
21. D. E. WILLARD, Two very fast trie data structures, *in* "Proceeding of the 19th Annual Allerton Conference on Communications, Control, and Computing," pp. 335–363, 1981.
22. D. E. WILLARD, Log-logarithmic worst-case range queries are possible in space $O(N)$, *Inform. Process. Lett.* **17** (1983), 81–89.
23. D. E. WILLARD, A new time complexity for orthogonal range queries, *in* "Proceeding of the 20th Annual Allerton Conference on Communications, Control, and Computing," pp. 462–472, 1982.
24. D. E. WILLARD, New data structures for orthogonal range queries, *SIAM J. Comput.*, in press.
25. A. C. YAO AND F. F. YAO, The complexity of serarching an ordered random table, *in* "Proceedings of the 17th Annual Symposium on the Foundations of Computer Science," pp. 173–177, 1976.