Information Processing Letters 17 (1983) 81–84
North-Holland

24 August 1983

LOG-LOGARITHMIC WORST-CASE RANGE QUERIES ARE POSSIBLE IN SPACE Θ(N)

Dan E. WILLARD

Bell Laboratories, Murray Hill, NJ 07974, U.S.A.

Communicated by K. Mehlhorn
Received 24 November 1982
Revised 17 February 1983

Let S denote a set of N records whose keys are distinct nonnegative integers less than some initially specified bound M. This paper introduces a new data structure, called the y-*fast trie*, which uses Θ(N) space and Θ(log log M) time for range queries on a random access machine. We will also define a simpler but less efficient structure, called the x-*fast trie*.

Keywords: Priority queues, stratified trees (often called 'Van Emde Boas trees'), sparse tables, special search algorithm of Fredmann, Komolós and Szemerédi

1. Introduction

Below we give a list of retrieval operations that we perform on a set of N positive integers, each $< M$:

(i) FIND(K): Determine whether or not the key K belongs to the set S, and return a pointer to K if it belongs to 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 of the set S with key value less than K.

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

For convenience, we will say that a data structure has a *worst-case overall-retrieval* complexity Θ[f(M)] iff retrieval operations (i) through (iii) have a worst-case runtime Θ[f(M)] and the worst-case runtime of SUBSET queries is proportional to f(M) plus the size of the retrieved subset.

Van Emde Boas, Kaas and Zijlstra [10] have presented a data structure, called a *stratified tree*, that has worst-case overall-retrieval complexity Θ(log log M) and uses Θ(M log log M) memory. Their work motivated much of our research effort. It indicated the need for more space-efficient data structures [8]. Van Emde Boas developed a modified tree data structure that uses Θ(M) space in [9]. Also, Knuth [5] offered a very clear summary of the paper by Van Emde Boas et al. [10].

Prior to this article, space-efficient modifications of stratified trees [10] were studied by Johnson [3], who showed how to attain a worst-case overall-retrieval complexity

$$\Theta\big[\log\log(\text{SUCCESSOR(K)} - \text{PREDECESSOR(K)})\big]$$

in $\Theta(N \cdot M^{\epsilon})$ space, and by Willard [11,12], who showed how to attain a time-complexity $\Theta(\sqrt{\log M})$ in Θ(N) space. Also, Willard [12] proved that all implementations of stratified trees use at least expected memory $\Omega(\sqrt{NM})$ when there exists an integer i such that $N = \Theta(M^{1-2^{-i}})$, and that their worst-case space always respects the lower bound $\Omega(N^{3/4}M^{1/4})$.

In this article we modify the stratified tree [9,10] with the proposal of Fredman et al. [1] to prove that a worst-case overall-retrieval complexity Θ(log log M) is possible in space Θ(N). Our results are the best known complexities for the worst-case in O(N) space, but [2,6,7,16] produced a log log N expected time under the uniform distribution for *even* unindexed files, and Willard [13,14] shows that the latter expected time generalizes to many nonuniform densities (even in the absence of an index or other information specify-

0020-0190/83/$3.00 © 1983, Elsevier Science Publishers B.V. (North-Holland)

ing the probability distribution).

Our data structure has a good expected insertion–deletion time, but it does not control the worst case of insertions and deletions. Therefore, no one of the results mentioned in this section is preferable to the others by all measures of complexity. In particular, the literature on stratified tree-like methods [3,9,10,11,12] has established several data structures with better insertion–deletion times than y-fast tries although less efficient combinations of retrieval time and space. This point leads to several open questions mentioned in Section 3, the most important of which is whether a worst-case complexity $\Theta(\log \log M)$ for insertions, deletions and retrievals simultaneously is possible in $\Theta(N)$ space.

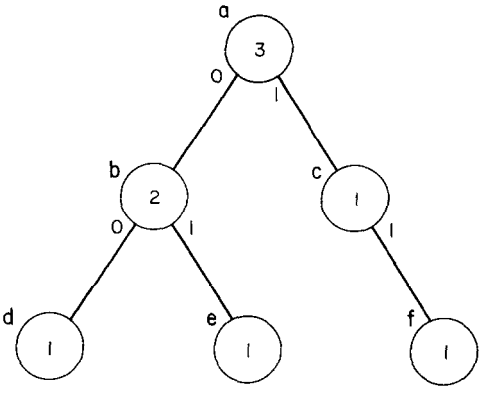

Fig. 1. An x-fast trie representing the set {0, 1, 3}. The quantity *inside a node is its* COUNT*-field. The only nodes with well-defined* DESCENDANT fields are the leaves (pointing to themselves) and c pointing to f.

2. Main result

For simplicity, we assume that M is an integer of the form $2^h - 1$. Our first data structure, the x-*fast trie*, will consist of a binary trie of height h where all records are stored at the leaf level (that is, at depth h). If v is a node at a height j, then all the leaves descending from v will have key values between the quantities $(i-1)2^j + 1$ and $i \cdot 2^j$, for some integer i. We will call i the *identifier* of v, and denote it as ID(v). The term COUNT(v) will denote the number of elements in the set S which correspond to leaves descending from v. If v is a node with no left (respectively right) son, then DESCENDANT(v) will be a pointer to the leaf with the smallest (respectively largest) key descending from it. The leaves of an x-fast trie will form a doubly-linked list with each leaf pointing to its left and right neighbor. To save memory space, a node v is stored in the x-fast trie only when COUNT(v) > 0. Fig. 1 illustrates an example of the main section of an x-fast trie.

The second part of an x-fast trie, its level-search structure (LSS), uses a concept recently introduced by Fredman, Komolós and Szemerédi [1]. They considered a computation model identical to that in the literature on stratified trees [1], and illustrated a data structure which uses $\Theta(n)$ space and executes FIND-queries in worst-case time $\Theta(1)$ for any set of n nonnegative integer keys each of which is $< m$. We will use $h + 1$ copies of this search structure. The *jth* copy, denoted LSS(j), will represent the set of trie-nodes of height j with their identifiers serving as keys. For instance, LSS(0) in Fig. 1 represents the trie's three leaves with the key-values 0, 1, and 3, respectively.

In our discussion, a node v of height j will be called an ancestor of the integer K iff $\lfloor K/2^j \rfloor =$ ID(v). The symbol BOTTOM(K) will denote the lowest ancestor of v stored in the trie. (We do not store nodes when COUNT(v) = 0.) Many aspects of our algorithm and data structure will have analogs from the work of Van Emde Boas, Kaas and Zijlstra [10]. For instance, they used analogs of descendant-pointers and also employed a binary search similar to that described in the next paragraph for finding BOTTOM(K). However, our method uses memory space much more economically. The next three lemmas are therefore significant.

Lemma 1. *The x-fast trie makes it possible to find the node* BOTTOM(K) *in worst-case time* $\Theta(\log \log M)$.

Proof. An algorithm which finds this node by making a binary search among the $h + 1$ different

[1] The assumption in [1] and in all the other articles we have cited is that all arithmetic operations on two nonnegative integers $\leq M$ can be performed in time $\Theta(1)$.

LSSs is shown in Fig. 2. Each probe of an LSS uses time $\Theta(1)$ by the basic result of [1]. Since our binary search makes $\lceil \log(h+1) \rceil$ probes and since $h = \log M$, the time of this algorithm is clearly $\Theta(\log \log M)$. □

A BINARY SEARCH OF THE TRIE-LEVELS FOR FINDING BOTTOM (K)

Step 1. Set $\ell = 0$ and $u = h$.

Step 2. Set $j = \lfloor \frac{1}{2}(\ell + u) \rfloor$.

Step 3. If searching LSS(j) indicates that the key K has a non-null ancestor at a height j in the trie, then set $u = j$, else set $\ell = j + 1$.

Step 4. If $\ell \neq u$, then go back to Step 2, else print the ancestor found in Step 3, since it is BOTTOM(K).

Fig. 2.

Lemma 2. *x-fast tries have a worst-case overall-retrieval complexity* $\Theta(\log \log M)$ *and never occupy more memory than* $O(N \cdot \log M)$.

Proof. *Time-complexity.* The first step of a search in an x-fast trie will be the $\Theta(\log \log M)$ algorithm for finding BOTTOM(K) in Lemma 1. Let v denote the node BOTTOM(K). The DESCENDANT-field of this node will be a pointer to K, its successor, or its predecessor. Since the leaves of an x-fast trie are ordered by key-value with each leaf pointing to its predecessor and successor, only constant time is needed to find these three elements after BOTTOM(K) is located. These three queries therefore run in time $\Theta(\log \log M)$. Subset queries run in time proportional to log log M plus the size of the retrieved subset, by similar reasoning.

Space-complexity. A trie of height log M with N leaves can have no more than $N \cdot \log M$ nodes. The main section of the x-fast trie will therefore consume this much space, and its LSSs also use space linear in $N \cdot \log M$ by the theorem of Fredman, Komolós and Szemerédi [1]. □

Van Emde Boas [9] has noted that the memory space of stratified trees can be reduced from $\Theta(M \log \log M)$ to $\Theta(M)$ by pruning the bottom of the data structure, and similar techniques have been used in other contexts [4,11,12]. Now we will apply this technique to x-fast tries, and develop a modified data structure, called the y-fast trie, which has the same retrieval complexity as x-fast tries but satisfies a better memory constraint $\Theta(N)$.

Define an L-*separator* of the set S, denoted S_L, as the subset of S including S's largest element, its smallest element, its (L + 1)st smallest element, its (2L + 1)st smallest element, etc. Let K_i denote the *i*th smallest element in S_L and let

$$S_i^* = \{k \in S \mid K_i \leqslant k < K_{i+1}\}.$$

Then a y-fast trie of order L will be defined as a two-part data structure whose top half is an x-fast trie representing the set S_L, whose bottom half is a forest of binary trees of height $\lceil \log L \rceil$ where the *i*th tree describes the set S_i^*, and which has the *i*th leaf in the top half of this data structure pointing to the *i*th tree.

Lemma 1 implies that the top half of the y-fast trie can certainly be searched in time $\Theta(\log \log M)$, and will occupy no more memory than $\Theta[(N/L) \times \log M]$. Each binary tree in the bottom half will have a search-time $\Theta(\log L)$, and the full forest of binary trees will use space $\Theta(N)$. Therefore, an arbitrary y-fast trie of order L will have a worst-case overall-retrieval complexity $\Theta(\log L + \log \log M)$ and will use $\Theta[N \cdot (1 + (\log M)/L)]$ space. Lemma 3 follows by applying these observations to the case $L = \log M$.

Lemma 3. *y-fast tries make possible a worst-case overall-retrieval complexity* $\Theta(\log \log M)$ *within memory space* $\Theta(N)$.

3. Open questions

Does Lemma 3 represent the best time possible for data structures in the space O(N)? Yao and Yao [16] have proven a lower bound log log N for unindexed files which may be partially relevant to this open question.

How does the answer to the question above change if the environment is dynamic? One problem is that y-fast tries do not have a good worst-case insertion–deletion time, although it should be possible to guarantee expected cost log log N. The best current worst-case result in the space O(N) is given in [11,12], and [3,8] describe alternatives in larger memory spaces. Fast tries have unusual implications for multi-dimensional retrieval [15], and many further questions remain there.

Acknowledgment

I would like to thank Eric Wolman for suggestions on presentation.

References

[1] M.L. Fredman, J. Komolós and E. Szemerédi, Storing a space table with O(1) worst-case access times, Proc. 23rd IEEE Symp. on Foundations of Computer Science (1982) pp. 165–169.

[2] G.H. Gonnet, L.D. Rogers and J.A. George, An algorithmic and complexity analysis of interpolation search, Acta Inform. 13 (1980) 39–52.

[3] D.B. Johnson, A priority queue in which initialization and queue operations take O(log log D) time, Rept. No. CS 81-13, Penn. State University, 1981.

[4] D.E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching (Addison-Wesley, Reading, MA, 1973).

[5] D.E. Knuth, Widely disseminated classroom notes on stratified trees, 1979.

[6] Y. Pearl, A. Itai and H. Avni, Interpolation search—A log log N search, Comm. ACM 21 (1978) 550–554.

[7] Y. Pearl and E.M. Reingold, Understanding the complexity of interpolation search, Inform. Process. Letters 6 (6) (1977) 219–222.

[8] P. Van Emde Boas, Preserving order in a forest in less than logarithmic time, Proc. 16th Ann. Symp. on the Foundations of Computer Science (1975) pp. 75–84.

[9] P. Van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Inform. Process. Lett. 6 (1977) 80–82.

[10] P. Van Emde Boas, R. Kaas and E. Zijlstra, Design and implementation of an efficient priority queue, Math. Systems Theory 10 (1977) 99–127.

[11] D.E. Willard, Two very fast trie data structures, 19th Ann. Allerton Conf. on Communication, Control and Computing (1981) pp. 355–363.

[12] D.E. Willard, New trie data structures which support very fast search operations of order $\sqrt{\log M}$, JCSS, to appear.

[13] D.E. Willard, Searching nonuniformly generated files in log log N runtime, SIAM J. Comput., to appear.

[14] D.E. Willard, A log log N search algorithm for nonuniform distribution, Proc. ORSA-TIMS Conf. on Applied Probability–Computer Science Interface Vol. II (Birkhäuser, Boston, 1981) pp. 3–14.

[15] D.E. Willard, A new time complexity for orthogonal range queries, 20th Allerton Conf. on Communications, Control, and Computing (1982) pp. 462–472.

[16] A.C. Yao and F.F. Yao, The complexity of searching an ordered random table, Proc. 17th Ann. Symp. on the Foundations of Computer Science (1975) pp. 173–177.