Optimal Algorithms for List Indexing and Subset Rank

Paul F. Dietz

Technical Report 291
June 1989

Optimal Algorithms for List Indexing and Subset Rank

Paul F. Dietz
Department of Computer Science
University of Rochester
Rochester, NY 14627

dietz@cs.rochester.edu

Abstract

Fredman and Saks [1] have proved a $\Omega(\log n / \log \log n)$ amortized time lower bound for two problems, List Indexing and Subset Rank, in the cell probe model with logarithmic word size. This paper gives algorithms for both problems that achieve the lower bound on a RAM with logarithmic word size.

1 Introduction

The List Indexing problem[1] is that of performing the following operations on a linked list:

Insert(x, y) Insert a new record y immediately after record x.

Delete(x) Delete record x from the list.

Index(i) Return the ith element in the list.

[1] Fredman and Saks called this the List Representation problem.

Position(x) Return the position in the list of record x. That is, $Position = Index^{-1}$.

A list admitting these operations is called an *indexed list.* (Note that records are not stored in some predefined sorted order, since they do not have associated keys; order is determined entirely by the arguments to the *Insert* operations.)

Fredman and Saks [1] have proved a lower bound of $\Omega(\log n / \log\log n)$ amortized time per operation for indexed lists in the cell probe model of computation [5]. The obvious upper bound, using balanced trees, is $O(\log n)$ time per operation (n the length of the list) [2].

Fredman and Saks also proved the same lower bound for a related problem, Subset Rank. This it the problem of maintaining a subset $S \subseteq \{1, \ldots, n\}$ under the operations *Insert*, *Delete* and *Rank* (given $i \in S$, return the number of element in S that are less than or equal to i).

This paper describes an algorithm for indexed lists that achieves the lower bound, in an amortized sense. The algorithm requires that one be able to manipulate integers with $O(\log n)$ bits in constant time, and makes use of the addressing capabilities of the RAM model. I assume that words with $O(\log n)$ bits can be manipulated in constant time.

Section 2 describes the Partial Sum problem, its efficient solution on the RAM and the application to the Subset rank problem. Section 3 gives the main algorithm for indexed lists.

2 The Partial Sum Problem

2.1 The Partial Sum Problem on Short Lists

The data structure will make use of an efficient algorithm for the following problem, which is a restricted version of a problem described by Fredman [2], Yao [6] and Fredman and Saks [1].

Definition 1 *Let* $A[1], \ldots, A[b]$ *be integers. The* Partial Sum problem *is that of performing two kinds of operations: the update* $add(i, \delta)$*, which implements* $A[i] \leftarrow A[i] + \delta$ *(where* $|\delta| = O(\log^{O(1)} n)$*), and the query* $sum(j)$*, which returns* $\sum_{i \leq j} A[i]$.

I now show how the Partial Sum problem can be solved in $O(1)$ amortized time per operation on a RAM if b is $O(\log^{\epsilon} n)$, where $\log n$ is the word size of the machine and ϵ is a positive

[2] All logarithms in this paper are base two.

constant less than one[3]. I assume I can perform a linear (in n) amount of precomputation.

The idea is to represent the partial sums in two parts: an array B which contains an "old" version of the partial sums, and an array C which contains a record of recent updates. Every b updates, the recent changes stored in C are expunged and B is brought up to date. In more detail,

```
var B[1..b]: integer;
var C[1..b]: −log^c n.. log^c n;
var count: integer ← 0;

proc add(i, δ)
        count ← count + 1;
        C[i] ← C[i] + δ;
        if count = b then
          count ← 0;
          for j ← 1 to b do
              B[j] ← sum(j);
          C[1..b] ← 0
end add
```

The arrays satisfy this equation:

$$\sum_{i=1}^{j} A[i] \;=\; B[j] \;+ \sum_{k=1}^{j} C[k]. \tag{1}$$

The algorithm is made efficient by representing C by a string of $\Omega(\log^c n \log\log n)$ bits Updates to the bits string can be made in constant time by table lookup.

Note that the sum in the right side of equation 1 is a function only of C and j. We can precompute this function for all possible values of its arguments and store them in a table. Precomputation can be done in sublinear time; lookup takes constant time. Therefore, we can perform $sum(j)$ queries in constant time. Note that if we are not allowed precomputation the algorithm can still be made to run in $O(\log n/\log\log n)$ time per operation. Precompute the table during the first $2^{\log n/\log\log n}$ operations. During this time represent the set using a balanced tree. After the table is ready, copy it into the optimal data structure. This will cause only $O(1)$ extra amortized work per operation.

It remains to show that updates take constant amortized time. When C overflows, after b updates, we spend $\Theta(b)$ time updating B (C can be zeroed in constant time). Therefore, each

[3]Actually, b could be as large as $c\log n/\log\log n$ for sufficiently small positive constant c.

of the last b updates gets charged a constant amount of work.

We can get rid of the need for amortization by updating the $B[j]$'s incrementally. The updates performed on C can be implemented by table lookup (indexed by C and *count*).

```
proc add(i, δ)
    count ← count + 1;
    C[i] ← C[i] + δ;
    B[count] ← B[count] + C[count];
    if count < b then
      C[count + 1] ← C[count] + C[count + 1];
    end;
    C[count] ← 0;
    if count = b then count ← 0
end add
```

We conclude:

Theorem 2 *The Partial Sum problem on lists of length $O(\log^\epsilon n)$ can be solved in constant time per operation on a RAM with logarithmic word size.*

2.2 Partial Sums on Large Lists; Subset Rank

This section extends the algorithm of the previous section to longer lists. Let m be the length of the list. We store the list at the leaves of a nearly complete tree of branching factor $b = \Theta(\log^\epsilon n)$. The tree has height $\Theta(\log_b m)$. If $m = n$, the tree has height $\Theta(\log n / \log\log n)$. At each internal node we store the sum of the leaves in the subtree rooted at that node. Call this the weight of the node. We also store, at each internal node, the partial sums of the weights of its children, using the algorithm described in the previous section. To compute the sum $A[1] + \ldots + A[i]$, we add the weight of $A[i]$ to the weights of the left siblings of the nodes on the path from the ith leaf back to the root. This can be done in time proportional to the height of the tree.

Theorem 3 *The Partial Sum problem on lists of length m can be solved in $\Theta(\log m / \log\log n)$ time per operation on a RAM with logarithmic word size.*

If the $A[i]$ are restricted to zero/one values, then this is the Subset Rank problem.

Corollary 4 *The Subset Rank problem can be solved in* $\Theta(\log n / \log\log n)$ *time on a RAM with logarithmic word size.*

3 An Optimal Algorithm for List Indexing

This section describes the algorithm for List Indexing. Briefly, the idea is to represent the list with a tree of branching factor roughly $b \in O(\log^{\epsilon} n)$. At each internal node, we use the fast algorithm for the Partial Sum problem to efficiently compute the number of leaves in the subtrees rooted at the first j children of the node. These partial sums allow us to find the rank of a list element or to find the ith list element in time proportional to the height of the tree.

3.1 Notation

The algorithm will make use of rooted, ordered trees. If x is a node, define

$h(x)$ The height of x.

$p(x)$ The parent of x (**null** if x is the root).

$w(x)$ The number of leaves in the subtree rooted at x, called the *weight* of x.

$i(x)$ If x is not the root, $i(x)$ is the position of x in the list of the children of $p(x)$.

For each node x, with children $x_1, \ldots, x_k$, define $w^+(x,0), \ldots, w^+(x,k)$ as follows:

$$w^+(x,i) = \begin{cases} 0 & \text{if } i = 0 \\ w^+(x,i-1) + w(x_i) & \text{otherwise} \end{cases}$$

That is, $w^+(x,i)$ is the sum of the weights of the leftmost i children of x. If x is not the root, define $w^*(x) = w^+(p(x), i(x) - 1)$ and $w^+(x) = w^+(p(x), i(x))$.

If x is an internal node and $j \in \{1, \ldots, w(x)\}$, define $s(x,j)$ to be the leftmost child y of x such that $w^+(y) \geq j$.

3.2 Balanced Trees

The algorithm makes use a kind of weight balanced B–tree. Let c_1 and ϵ be positive constants, $\epsilon < 1$.

Definition 5 *A WBB–tree is a rooted, ordered tree having the following properties:*

- *The elements of the list are at the leaves of the tree, in order from left to right.*
- *The leaves of the tree are at the same depth.*
- *Let* $b = \max(4, \lceil c_1 \log^\epsilon N \rceil)$ *be the* branching factor *of the tree. Let* N *be a value chosen so that*

$$1/2 < N/w(root) < 2. \tag{2}$$

Define the fullness *of a node* x *to be the quantity*

$$full(x) = w(x)/b^{h(x)} \tag{3}$$

For every internal node x *except the root,*

$$1/2 < full(x) < 2 \tag{4}$$

and, $full(root) < 2$.

WBB–trees are closely related to ordinary B–trees [3]. The weight balancing condition has been added so that the *Index* operation can be efficiently implemented. The branching factor of $\Theta(\log^\epsilon n)$ was chosen so that the following theorem holds.

Theorem 6 *The height of a WBB–tree is* $\Theta(\log n / \log\log n)$, *and every node has* $O(\log^\epsilon n)$ *children.*

Proof: Immediate. ∎

I now show how to maintain the balance conditions under insertion/deletion of leaves.

The values of N and b are only changed when equation 2 is violated. When that happens, N is set to $w(root)$ and the entire tree is reconstructed. This is done in such as way as to make nodes at the same height in the tree have similar weights and to make their weights be close to $b^{h(x)}$ (except for the root, which may have weight as small as $2b^{h(x)-1}$.)

Reconstruction takes $\Theta(n)$ time ($O(N)$ time to recompute the lookup table for the Partial Sum problems at the nodes of the tree [see below] and $\Theta(N)$ time to reconstruct the tree itself). Since $\Omega(N)$ updates occur between changes to N, each update is charged only $O(1)$ amortized work.

When a new leaf is added, an internal node of height 1 acquires an extra child. Find the highest ancestor z of the leaf that now violates equation 4, if any exist. The subtree rooted at z is deleted and replaced by two subtrees of half size.

When a leaf is deleted, one again finds the highest ancestor z of the leaf which now violates equation 4. z cannot be the root, and it cannot be the only child of its parent. Let y be a neighboring sibling of z. If $full(y) \leq 1$ we merge the subtrees rooted at y and z and form a new subtree of height $h(z)$ with $w(y) + w(z)$ leaves. Otherwise, merge the subtrees and evenly split their leaves into two subtrees of height $h(z)$. Again, when subtrees are reconstructed we attempt to make nodes of height n have weight about b^h.

One can show that an insertion or deletion in a WBB–tree takes $O(\log n / \log\log n)$ amortized time. We argued before that changes in N charge each update $O(1)$ work. The time spent rebalancing subtrees is easily analyzed by means of a potential function [4] Φ_1 defined by

$$\Phi_1 \;=\; c_2 \sum_{x \text{ a node}} |w(x) - b^{h(x)}| \tag{5}$$

where c_2 is some positive constant. An insertion or deletion causes Φ_1 to increase by $O(\log n / \log\log n)$. Rebalancing reduces Φ_1 by $\Theta(w(z))$ (proof omitted), so if c_2 is large enough the time spent rebalancing is less than the reduction in potential.

3.3 Position Queries; Representation of w^*

Recall that $w^*(x)$ (x not the root) is the sum of the weights of the siblings of x that occur to the left of x. One can easily compute the position of a leaf y as follows. Let $y = x_0, \ldots, x_h$ be the path in the tree from y to the root x_h. Then,

$$Position(y) \;=\; 1 + \sum_{i=0}^{h-1} w^*(x_i). \tag{6}$$

We can compute $w^*(x)$ in constant time if, at each node, we use the fast algorithm for the Partial Sum problem (section 2). When an insertion (deletion) is performed in the subtree rooted at some node x, this causes $w(x)$ to increase (decrease) by 1, which is less than b.

Insertions and deletions in the tree can also cause subtrees to be reconstructed. When a new node is allocated in this way, the representation for its Partial Sum problem is reset (the array B is brought up to date). This increases the cost of an update by only a constant factor. When the number of children of a node changes, the same thing happens. This adds $\Theta(b)$ to the cost of each update, which is not significant.

3.4 Index Queries

To perform *Index* queries, we compute $s(x, j)$ (the child of x containing the jth leaf beneath x). We store at each internal node x an array $S[1..b]$ of pointers to children of x. $S[i]$ will be a child of x that is "close to" $s(x, w(x)i/b)$. More specifically, $S[i]$ is a child of x such that

$$\frac{w(x)(i-1)}{b} \leq w^+(S[i]) \pm O(b) \leq \frac{w(x)i}{b}. \tag{7}$$

Using S, we can efficiently compute $s(x, j)$:

1. Let y be $S[\lceil jb/w(x) \rceil]$.
2. Starting at y search linearly through the children of x to find the leftmost child z with $w^+(z) \leq j$.

This takes $O(1)$ time if $h(x) \geq 2$. We can compute $Index(i)$ as follows:

1. Initially, let x be the root.
2. While x is not a leaf,
   (a) Let y be $s(x, i)$.
   (b) Set x to y and i to $i - w^*(y)$.
3. Return x.

All iterations of the inner loop except the last take $O(1)$ time; the last takes $O(b)$ time. Since b is $\Theta(\log^\epsilon n)$ and the height of the tree is $\Theta(\log n / \log\log n)$, *Index* can be computed in $\Theta(\log n / \log\log n)$ time.

It remains to show how S changes during updates and that the inequality in equation 7 is maintained. S is changed only when the array B of old partial sums at x is updated. At that time, we set $S[i]$ to $s(x, w(x)i/b)$. This satisfies equation 7. At most b updates are performed beneath x before S is updated, so equation 7 remains valid.

4 Summary

I have given optimal algorithms for List Indexing and Subset Rank on the RAM model with logarithmic word size. The algorithms described here use a tree structure that is perhaps overly

complicated. For example, there is really no need for the leaves of the tree to be at the same depth. It would also be desirable to eliminate amortization.

5 Acknowledgements

I would like to thank Michael Fredman for telling me about the List Indexing problem, and Rajeev Raman for spotting some errors in a earlier draft of this paper.

References

[1] Michael Fredman and Michael Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st ACM STOC*, pages 345–354, May 1989.

[2] Michael L. Fredman. The complexity of maintaining an array and computing its partial sums. *Journal of the ACM*, 29(1):250–260, January 1982.

[3] Kurt Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms.* Springer-Verlag, New York, 1984.

[4] Robert E. Tarjan. Amortized computational complexity. *SIAM J. on Alg. and Disc. Meth.*, 6(2):306–318, 1985.

[5] Andrew C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, July 1981.

[6] Andrew C. Yao. On the complexity of maintaining partial sums. *SIAM J. On Computing*, 14(2):277–288, May 1985.