SPACE-TIME TRADE OFF IN IMPLEMENTING CERTAIN SET OPERATIONS

Jayadev MISRA *
Department of Computer Science, University of Texas, Austin, TX 78712, U.S.A.

Received 11 June 1978; revised version received 10 October 1978

Algorithms, complexity, space-time trade off

1. Introduction

Let S be a set of integers that initially is $\{1, 2, ..., n\}$. Consider the following operations on S.

remove(i) : : i is removed from S.
next(i) : : returns the next larger number j, $j > i$, $j \in S$, if such a j exists.

We consider algorithms that trade off time for space in implementing these operations. The following notations are used in this paper: log denotes the base 2 logarithm. $\log^t(n)$ denotes log log ··· log n, where log is applied t times. By convention, $\log^0(n) = n$. $G(n)$ is defined as the smallest k such that $\log^k(n) \leqslant 1$, for any positive integer n. $\alpha(n)$ is related to the inverse of Ackermann's function; an exact definition may be found in [2]. We note that $\alpha(n)$ grows slower than $G(n)$.

We distinguish between two types of problems.

Type 1. **next**(i) is applied only if $i \in S$.

Type 2. **next**(i) may be applied irrespective of whether $i \in S$ or not.

We use a logarithmic cost measure in stating the time and space complexities. We assume that the time needed to index into an array of size n is O(log n) and storage (number of bits) required to address any one of potentially n items is O(log n).

We exhibit algorithms whose time and storage requirements are summarized in Table 1, for any t, $1 \leqslant t \leqslant G(n)$. Columns 2, 4 labelled 'variation' refer to slight variations of the algorithms represented in the preceding columns. Variations reduce the storage requirement to a linear function of n, for constant t. Limiting behaviors of these algorithms are shown in Table 2. This table is derived from Table 1 by setting $t = 1$ and $t = G(n)$ in turn. Algorithms corresponding to $t = 1$ are known. $t = G(n)$ is particularly interesting since the time requirements are 'almost' optimal and storage requirements are almost linear.

These operations were found necessary in a recent algorithm [1], which used a sieve technique to find all prime numbers between 2 and n. The algorithm executes at most n **remove** and **next** operations of type 1. A linked list was used to implement S. This has a logarithmic cost of O(log n) time per operation using O(n log n) bits of storage, since each link can potentially address any one of n items. Using the methods of this paper, for any constant t, an algorithm that requires O(log n) time per operation can still be constructed that uses storage of O($n \log^t(n)$) bits.

2. Solving the type 1 problem

Each operation can be done in O(log n) time by maintaining S in a doubly linked list as follows. Let BACK and FRONT be two arrays of length n each. For any $i \in S$, FRONT[i] denotes the next larger element and BACK[i] denotes the next smaller element in S; if no such elements exist, the corresponding values are $n + 1$ or 0. Initially, BACK[i] = $i - 1$, $i > 0$ and FRONT[i] = $i + 1$, $i \leqslant n$. Operation **next** is implemented by returning FRONT[i] if it is less than

* Work partially supported by NSF Grant MCS77-09812.

Table 1
Time and storage requirements (logarithmic) of the algorithms, for any t, $1 \leqslant t \leqslant G(n)$

	Type 1	Type 1 (Variation of preceding algorithm)	Type 2	Type 2 (Variation of preceding algorithm)
time	$O(t \log n)$ per operation	$O((t-1) \log n + \log n \log n \log^{t-1}(n))$ per operation	$O(tn \log n\, \alpha(n))$ per $O(n)$ operations	$O(n \log n \log^{t-1}(n) + tn \log n\, \alpha(n))$ per $O(n)$ operations
storage	$O(n \log^t n + tn)$	$O(tn)$	$O(n \log^t(n) + tn)$	$O(tn)$

$n+1$. Operation **remove**$[i]$ links BACK$[i]$ to FRONT$[i]$ properly.

The storage requirement of $O(n \log n)$ bits can be drastically reduced with a slight increase in time requirements as shown below; the amount of trade off depends on t, $1 \leqslant t \leqslant G(n)$. The algorithm is easily explained when $t = 2$. Assume that $n = 2^{2^k}$, for some $k > 0$. Divide the set $\{1, 2, ..., n\}$ into contiguous blocks of length $\log n$. There are thus $n/\log n$ blocks. Element i belongs to the $\lceil i/\log n \rceil$-th block; note that this block number can be computed for any i by properly shifting the bit representation of i, since $\log n$ is a power of 2. A block is *active* if it has some element x, $x \in S$; a block is *inactive* otherwise. Initially all the blocks are active.

At any point in the algorithm, an element $i \in S$ is linked to its predecessor and successor in its *own* block. If an element does not have a predecessor or successor in its own block, the corresponding link field is null. Similarly, all the *active* blocks themselves are doubly linked. Furthermore, each active block b holds a pointer, **first**(b), to the smallest i, $i \in S$ in that block.

Initialization of this data structure can be done in $O(n \log n)$ time; each element i is linked to $(i-1)$ and $(i+1)$ if these are in the same block and each block is linked to its preceding and succeeding blocks. Operation **next**(i), $i \in S$ may be implemented as follows: determine whether i has a successor j in its own block. If so, j is the answer; otherwise, find the successor block b of the parent block of i. (The parent block of i must be active.) If b is null then i does not have a next element, else **first**(b) is **next**(i). Operation **remove**(i) may be implemented as follows: remove i from the doubly-linked list in its own block. If both predecessor and successor of i were null, then i was the only element in S in its block; hence its removal makes its block inactive. Then remove the block from the doubly-linked list of blocks. Furthermore, first (b) needs to be updated properly if the removed

Table 2
Limiting behaviors of the algorithms represented in Table 1. $t = 1$, corresponds to known algorithms.

		Type 1	Type 1 (Variation)	Type 2	Type 2 (Variation)
$t = 1$	Time	$O(\log n)$ per operation	$O(n \log n)$ per operation	$O(n \log n\, \alpha(n))$ per $O(n)$ operations	$O(n^2 \log n)$ per $O(n)$ operations
	Storage	$O(n \log n)$	$O(n)$	$O(n \log n)$	$O(n)$
	Known algorithm	Doubly linked list manipulation	Linear search (without links)	Disjoint set union	Linear search (without links)
$t = G(n)$	Time	$O(\log n\, G(n))$ per operation	$O(\log n\, G(n))$ per operation	$O(n \log n\, G(n)\, \alpha(n))$ per $O(n)$ operations	$O(n \log n\, G(n)\, \alpha(n))$ per $O(n)$ operations
	Storage	$O(n\, G(n))$	$O(n\, G(n))$	$O(n\, G(n))$	$O(n\, G(n))$

element is the first one belonging to S in the block. Operation **next** may involve finding the next element within the block and the next of the parent block. Similarly **remove** may delete an element from a doubly-linked list within a block and also delete the parent block from the list of active blocks. Each application of **next** or **remove** requires $O(\log n)$ step for each level. Storage required is $O(n \log\log n)$ for links within the blocks and $O((n/\log n)\log(n/\log n))$ for linking the blocks. $O((n/\log n)\log\log n)$ bits are required to maintain **first** for all the blocks combined. Hence the total storage is

$$O(n \log\log n) + O\left(\frac{n}{\log n} \log\left(\frac{n}{\log n}\right)\right)$$

$$+ O\left(\frac{n}{\log n} \log\log n\right) = O(n \log\log n) + O(n).$$

It is straightforward to generalize this idea of splitting into blocks to more than two levels. For instance with $t = 3$, we will split each block of size $\log n$ into smaller blocks each of size $\log\log n$.

We now describe the algorithm for any arbitrary t, $1 \leqslant t \leqslant G(n)$. We associate a tree structure having $(t+1)$ levels with the set $\{1, 2, ..., n\}$. The root is at level 0 and it corresponds to the entire set $\{1, 2, ..., n\}$. A node at the ith level, $0 \leqslant i < t$, corresponds to a contiguous block of $\log^i(n)$ unique elements; different nodes at the same level correspond to blocks of disjoint sets of elements. Hence the entire set $\{1, 2, ..., n\}$ is represented at the ith level by $n/\log^i(n)$ nodes, $0 \leqslant i < t$. A block of length $\log^i(n)$ corresponding to a node at ith level is split into smaller blocks of length $\log^{i+1}(n)$ which correspond to sons of this node at $(i+1)$st level, $i+1 < t$. There are n nodes at the tth level, each corresponding to one individual element.

A node is *active* if there is an element $x \in S$, in the block corresponding to it. Initially all nodes are active. As a result of **remove** operation some nodes may become inactive. Inactive nodes never become active again. Clearly a tth level node is active if the element corresponding to it is in S; an ith level node, $0 \leqslant i < t$, is active if it has a son which is active.

It is important to note that no explicit pointers are required to maintain the tree structure. An array can be maintained corresponding to every level; given the index of a node at $(i+1)$st level, the index of its father node at ith level can be computed easily by arithmetic manipulations. We next superimpose a doubly linked list structure on each level – all the active *brother* nodes in a level are doubly linked. Furthermore, every active internal nodes has a pointer to its first active son.

The amount of storage used is $O(n \log^t n)$ for doubly linking all the tth level nodes since they have $\log^{t-1}(n)$ brothers each. A node at the ith level, $i < t$, has $\log^{i-1}(n)/\log^i(n)$ brothers; number of nodes at the ith level is $n/\log^i(n)$. Hence number of bits for doubly linking nodes at ith level is $O(n/\log^i(n)) \log(\log^{i-1}(n)/\log^i(n)) = O(n) - O(n \log^{i+1}(n)/\log^i(n)) = O(n)$, $1 \leqslant i < t$. Additional data kept with each node requires no more than $O(n)$ storage per level for all nodes combined. Hence total amount of storage is bounded by $O(n \log^t(n)) + O(tn)$. The time required for processing each instruction is $O(t \log n)$, since at worst, we may have to go up the tree up to the first level (and then possibly climb down all the way to find the next element, for instance).

This technique can be applied even when n is not a power of 2, without 'padding' with dummy elements. Let $\lg(n)$ denote the next power of 2 greater than or equal to $\log n$; $\lg^0(n) = n$ and $\lg^{i+1}(n) = \lg(\lg^i(n))$. For any t, $1 \leqslant t \leqslant G(n)$, we have the tree structure consisting of $(t+1)$ levels as before. All nodes at the ith level, $0 \leqslant i < t$, except possibly the rightmost node, correspond to blocks of length $\lg^i(n)$; the last node corresponds to the remaining $n - \lfloor n/\lg^i(n) \rfloor \cdot \lg^i(n)$ elements, if $\lg^i(n)$ does not evenly divide n. Thus, number of nodes at the ith level is $\lceil n/\lg^i(n) \rceil$ and number of brothers of each ith level node is at most $\lceil \lg^{i-1}(n)/\lg^i(n) \rceil$. Hence time and storage requirements are of the same order as in the previous analysis. Since $\lg^{i-1}(n)/\lg^i(n)$, for any $i > 1$, is a power of 2, index of the father of any ith level node j, can be computed by simple manipulation of the bit representation of j.

3. Solving the type 2 problem

We show that this problem is reducible to the disjoint set union problem [2]; hence n applications of **remove**, **next** require no more than $O(n \log n\, \alpha(n))$ steps using $O(n \log n)$ bits of storage. The technique of the last section can then be used to obtain the space-time tradeoff.

Given $S \subseteq \{1, 2, ..., n\}$, we define an equivalence relation $\equiv$ on the set $\{1, 2, ..., n\}$ as follows. Let SUCC(i) denote the smallest j, $j > i$ and $j \in S$; SUCC is undefined if there is no such j. Define an equivalence relation $\equiv$ as follows; $i_1 \equiv i_2$ if and only if SUCC(i_1) = SUCC(i_2) or SUCC is undefined for both. Initially, each element belongs to a separate equivalence class. Effect of **remove**(i) is to adjoin two equivalence classes; namely the class corresponding to $(i-1)$ and the one corresponding to i. Effect of **next**(i) is to find the unique SUCC associated with the equivalence class of i – data which can be maintained with each equivalence class.

For any t, $1 \leqslant t \leqslant G(n)$, we associate a tree structure of $t+1$ levels with the set $\{1, 2, ..., n\}$. We define *active* nodes as given earlier. We redefine SUCC for any node b, SUCC(b) is the first active brother to the right * of b, if such a node exists. Two brother nodes will be called *equivalent* if both have the same SUCC or SUCC is undefined for both. Data structure for the disjoint set union algorithm is maintained corresponding to the set of *brother nodes* at every level. It is thus possible to find the SUCC of any node using the disjoint set union algorithm. Furthermore, a pointer to the first active son of every internal node is maintained with every node in the data structure.

Operation **next**(i) may be implemented as follows. Find the successor SUCC of the (tth level) node corresponding to i. If it has no successor, repeat this step on the father node and successive ancestor nodes.

Ultimately, either no successor is found (**next** is undefined) or a successor node b of an ancestor is found. Then the first active son of b is found and this step is repeated until an active terminal node corresponding to an element in S is found.

Operation **remove**(i) is implemented as follows. Whenever a node b is made inactive, the equivalence class corresponding to b and that corresponding to the brother node to the left of b (if one exists) get merged. The pointer of b's father to the first active son may need to be modified. This process continues up the tree as long as the removed node was the only active son of its father.

Disjoint set union algorithm requires $O(r \log r)$ bits of storage for r elements. Every node at the ith level has $\log^i(n)/\log^{i+1}(n)$ sons, $i+1 < t$, which is the set size for the disjoint set union algorithm at $(i+1)$th level corresponding to the sons of this node. Since number of nodes at the ith level is $n/\log^i(n)$, total storage requirement for $(i+1)$th level is

$$O\left(\frac{n}{\log^i(n)} \cdot \frac{\log^i(n)}{\log^{i+1}(n)} \log\left(\frac{\log^i(n)}{\log^{i+1}(n)}\right)\right) = O(n).$$

A $(t-1)$th level node corresponds to a block of size $\log^{t-1}(n)$ and hence has $\log^{t-1}(n)$ sons. Thus storage requirement for the tth level is

$$O\left(\frac{n}{\log^{t-1}(n)} \cdot \log^{t-1}(n) \cdot \log(\log^{t-1}(n))\right)$$
$$= O(n \log^t(n)).$$

Total storage requirement is $O(n \log^t(n)) + O(tn)$.

Every **next** and **remove** operation involves at most $O(t)$ disjoint set union operations on sets of size less than n. Total time required for $O(n)$ operation is bounded by $O(tn \log n\, \alpha(n))$.

4. A variation

A slight variation of the data structure results in a different space-time trade off. In the type 1 problem we do not doubly link the terminal (tth level) nodes. A linear search is performed at tth level to find the next active brother of an element. Rest of the levels are doubly linked as given before. Thus the total storage requirement is $O(tn)$ for all the levels combined, since only $O(n)$ storage (instead of $O(n \log^t(n))$ is needed for the tth level. However linear search at the lowest level may look at as many as $\log^{t-1}(n)$ elements, resulting in a cost of $O(\log n \log^{t-1}(n))$ for that level. Costs for other levels is $O((t-1)\log n)$; hence the total cost is $O(\log n \log^{t-1}(n)) + O((t-1)\log n)$. A similar variation for the type 2 problem results in a storage complexity of $O(tn)$ and time complexity of $O(n \log n \log^{t-1}(n)) + O(tn \log n\, \alpha(n))$.

Acknowledgement

I am indebted to Professor S. Rao Kosaraju of Johns Hopkins University who made many helpful

* A node n_2 is to the right of n_1 in the same level if the elements in the block corresponding to n_1 are all smaller than those corresponding to n_2.

suggestions. Earlier work with Professor David Gries on implementing a sieve algorithm for finding prime numbers led to this problem. Professors C.L. Liu of the University of Illinois and Jim Bitner of the University of Texas made helpful comments on improving the style of the paper. I am grateful to a referee who pointed out reference [4], which discusses a related problem.

References

[1] D. Gries and J. Misra, A linear sieve algorithm for finding prime numbers, CACM. To appear.

[2] R.E. Tarjan, Efficiency of a good but not linear set union algorithm, JACM 22 (2) (1975).

[3] P. Van Emde Boas, Preserving order in a forest in less than logarithmic time, Proc. 16th Ann. Symp. Foundations Comput. Sci., October 13–15, 1975, University of California at Berkeley.

[4] P. Van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Information Processing Letters 6 (3) (1977) 80–82.