Two Algorithms for Maintaining Order in a List

Daniel D. Sleator * and *Paul F. Dietz*

15 September 1988

CMU-CS-88-113

*Partial support provided by DARPA, ARPA order 4976, amendment 19, monitored by the Air Force Avionics Laboratory under contract F33615-87-C-1499, and by the National Science Foundation under grant CCR-8658139. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Air Force Avionics Laboratory, the National Science Foundation or the US Government.

Two Algorithms for Maintaining Order in a List [1]

Paul F. Dietz [2]
Department of Computer Science
University of Rochester
Rochester, NY 14627

Daniel D. Sleator [3]
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The *order maintenance problem* is that of maintaining a list under a sequence of *Insert* and *Delete* operations, while answering *Order* queries (determine which of two records comes first in the list). We give two new algorithms for this problem. The first algorithm matches the $O(1)$ amortized time per operation of the best previously known algorithm, and is much simpler. The second algorithm permits all operations to be performed in $O(1)$ *worst-case* time.

---

[1] A preliminary version of this paper was presented at the 19th Annual ACM Symposium on Theory of Computing, New York, May 25–27, 1987.

[2] This author was employed by Schlumberger-Doll Research, Ridgefield, Connecticut, at the time this work was done.

[3] Partial support provided by DARPA, ARPA order 4976, amendment 19, monitored by the Air Force Avionics Laboratory under contract F33615-87-C-1499, and by the National Science Foundation under grant CCR-8658139.

1. Introduction

A variety of important problems in the design of data structures arise from the need to store and manipulate ordered sets of records. These problems can be formulated as a collection of operation types, each with an anticipated frequency of occurrence. A data structure is desired which performs efficiently on the operations which occur with high frequency.

The most versatile data structure for solving these problems is the binary search tree [13]. In some circumstances the commonly used operations do not require the full generality of binary search trees, and faster methods are possible. In this paper, we consider one of these special problems.

The *order maintenance problem* is that of maintaining a non-empty list L of records under a sequence of the following three types of operations:

Insert(x, y): Insert record y after record x in the list. The record y must not already be in the list.

Delete(x): Delete record x from the list.

Order(x, y): Return true if x is before y in the list, otherwise return false.

In addition to being a very natural problem to consider, the order maintenance problem has several applications, the most compelling of which is the *ancestor query problem*. In this problem a rooted tree is maintained under the operations of insertion and deletion of nodes, while ancestor queries are being performed. These queries are of the form "is x an ancestor of y", where x and y are nodes in the tree. Dietz [2] showed how a data structure to solve the order maintenance problem can be used to solve the ancestor query problem, and how an ancestor query data structure can be used to implement context trees [15]. Another application of an efficient ancestor query data structure is in the construction of fully persistent data structures [5].

Search trees can be used to solve the order maintenance problem as follows. The records of the list are stored in the internal nodes of the search tree in symmetric order. Insertion and deletion are done in the ordinary fashion. An *Order*(x, y) query is performed by finding the least common ancestor of x and y (by walking up the tree from x and y), and determining which of x or y is in the left or right subtree of this common ancestor. By using balanced, or self-adjusting search trees [10], these operations on a list of n records can all be performed in $O(\log n)$ time.

Search trees allow other operations – such as concatenating and splitting – to be performed in $O(\log n)$ time. If concatenating and splitting are needed, but rarely compared to order queries, then Pugh's *skip lists* [8] are a viable alternative. This data structure allows the user to choose a trade-off point between the cost of doing the order queries and the other operations.

How can we take advantage of the specific requirements of the order maintenance problem to obtain a more efficient data structure? One approach is to maintain a linked list of records, and to keep in each record an associated number called the *label*. The labels

are in monotonically increasing order from the beginning of the list to the end. An *Order* query can then be performed by comparing the labels of the two records in question. When a record is inserted into the list it is given a label between those of its two neighbors. This method is extremely efficient if the insertions are very uniformly distributed throughout the list. If many insertions occur in the same region of the list, then many bits will be required to represent the labels so as to ensure that all labels are different. In the worst case, the penalty for having to manipulate and store such large numbers is severe, giving an algorithm which takes $O(n)$ time per operation.

A solution to this problem is to renumber the records in the vicinity of the inserted record. Dietz [2] gave an algorithm for performing insertions in which each insertion causes $O(\log n)$ renumberings to take place in the amortized sense[4]. (Here n is the number of records in the list.) In order to determine which records to renumber, Dietz's method made use of a data structure (in addition to the linked list of records) called the *overlying tree*.

Tarjan [11] observed that this algorithm for maintaining order in a list can be used as a component of another algorithm for the same problem that runs in constant amortized time per operation. This result is obtained by breaking the list into sublists of size $O(\log n)$, and using Dietz's algorithm to maintain the list of sublists. (This details of this method are given at the end of Section 2.) A similar result was independently obtained by Tsakalidis [14].

Leiserson [7] also devised (but did not publish) a method to solve the order maintenance problem. His algorithm did away with the explicit storage of an overlying tree. Instead, the bits of the numbers were constrained to correspond to paths in a hypothetical 2-4 tree which would represent the current list. He took advantage of the properties of 2's complement arithmetic to efficiently perform the renumbering. Leiserson's algorithm renumbers a sublist of records following the point of insertion, and uses wrapping modulo M to make the renumbering procedure independent of where the insertion takes place.

This paper presents two new algorithms for the order maintenance problem. The first is distinguished by its simplicity. It attains the same time bounds previously obtained by Dietz, Tsakalidis, and Lieserson for renumbering a list ($O(\log n)$ amortized renumberings per insertion), but is much simpler. Our algorithm resembles Leiserson's algorithm in that it does not make use of any data structures besides the list and the labels, and it uses the technique of wrapping modulo M. When an insertion occurs in a congested region of the list, a contiguous sublist of records (starting from the point of insertion) is renumbered as uniformly as possible. The criterion for determining which sublist to renumber is a simple test. Like the other methods, our algorithm can also be used in the list-of-lists scheme mentioned above to allow all operations to be performed in constant amortized time.

Our second algorithm achieves constant *worst-case* time for all operations. This answers an open question in [5] and permits updates to be performed on fully persistent search trees in $O(\log n)$ worst case time and $O(1)$ worst case space.

We describe both of our algorithms assuming that there is a fixed upper limit N on how

[4]The techniques of amortized analysis are described in [9] and [12].

big the list will ever grow. We also assume a model of computation in which numbers of $O(\log N)$ bits can be manipulated in constant time. These realistic assumptions simplify the description and analysis of the algorithms.

2. A Simple $O(\log n)$ Amortized Time Algorithm

The data structure consists of a circularly linked list of records. Each record r is labeled with an integer $v(r)$. The successor of a record is denoted $s(r)$. The set of integers available for labeling the records is $\{0, 1, 2, \ldots, M-1\}$. This set will be called the *arena*.

Let n denote the size of the list at any time. Our algorithm works under the assumption that the arena size, M, has been chosen so that

$$M > n^2 \tag{1}$$

always holds. Therefore, the list is restricted to contain at most

$$N = \left\lceil M^{1/2} \right\rceil - 1$$

records.

The list initially contains one record (which is never deleted) called the base and denoted b. The base represents the end and the beginning of the list. The initial label of the base is arbitrary. For convenience we will use the following definitions:

$$v_b(r) = (v(r) - v(b)) \bmod M$$

$$v_b^*(r) = \begin{cases} M & \text{if } s(r) = b \\ v_b(s(r)) & \text{otherwise} \end{cases}$$

The following invariant will always be maintained:

$$v_b(r) < v_b^*(r) \text{ for all records } r. \tag{2}$$

The algorithms for *Order* and *Delete* are trivial: *Order*(x, y) is done by comparing $v_b(x)$ and $v_b(y)$, if $v_b(x) < v_b(y)$ return true, otherwise return false. *Delete*(x) is done simply by removing x from the list.

Before we describe the insertion algorithm, we define some notation. Let *Insert*(x, y) be the insertion operation being performed. Let v_0 be the label of record x, let v_1 be the label of record $s(x)$, let v_2 be the label of $s^2(x) = s(s(x))$, etc. Let $w_i = (v_i - v_0) \bmod M$ for $0 \le i < n$ (n denotes the number of records in the list before the insertion). Note that $w_0 = 0$, and $w_1 = 1$, and define $w_n = M$. The insertion algorithm first walks down the list until $w_j > j^2$:

```
j ← 1;
while w_j ≤ j² do
        j ← j + 1
end
```

Relabel the $j-1$ records $s^1(x), \ldots s^{j-1}(x)$ with the labels

$$v(s^k(x)) = \left(\left\lfloor w_j \frac{k}{j} \right\rfloor + v_0 \right) \bmod M. \tag{3}$$

The j gaps created between adjacent labels will differ in size by at most 1. It is now the case that $v_b(x) + 2 \leq v_b^*(x)$, and we can insert y between x and $s(x)$, setting $v(y)$ to be

$$v_b(y) = \left\lfloor \frac{v_b(x) + v_b^*(s(x))}{2} \right\rfloor.$$

This completes the insertion.

This algorithm can be implemented easily and efficiently. If M is chosen so that $\log M$ is the word size of the machine[5] (or a small multiple thereof) then the sum of two numbers modulo M is computed automatically as a result of adding the numbers.

The following lemmas show that the insertion algorithm terminates, and relabels records in a way that satisfies the invariant 2.

Lemma 1 *When inserting a record y after a record x, (a) the search part of the insertion algorithm terminates, (b) upon termination $w_j \geq 2j$, and (c) if $j > 1$ upon termination then $w_{j-1} < n^2$.*

Proof: (a) Note that $w_n = M > n^2$, so the loop must terminate.
(b) If the loop terminates with $j = 1$ then $j \geq 2 = 2j$. If $j > 1$ then $j > j^2 \geq 2j$.
(c) Since the loop did not terminate in the previous iteration, $w_{j-1} \leq (j-1)^2 < j^2 \leq n^2$.
∎

We will analyze this algorithm by the technique described in [9] and [12]. The potential function we shall use is:

$$\Phi = \sum_{k=0}^{n-1} \max(0, c \log \frac{2n}{g_k}) \tag{4}$$

Here g_k denotes the size of the gap between successive labels, that is, $g_k = w_{k+1} - w_k$, $0 \leq k \leq j-1$. The positive constant c will be chosen later.

Lemma 2 *For fixed $w_j > j^2$, the value of*

$$\Phi' = \sum_{k=0}^{j-1} \max(0, c \log \frac{2n}{g_k}) \tag{5}$$

is minimized when $g_k = 2k+1$, $0 \leq k < j-1$.

[5] All logarithms in this paper are binary.

Proof: For the moment, we will assume the w_i's (and therefore the g_i's) can take on real values, rather than just integer values. We will show Φ' achieves its minimum value when $w_i = i^2$, $0 < i < j$.

Note that the function

$$\phi(g_i) = \max(0, c \log \frac{2n}{g_i})$$

is concave. This implies that if $g_i > g_{i+1}$, setting $w_{i+1} = \frac{1}{2}(w_i + w_{i+2})$ (and therefore $g_i = g_{i+1}$) cannot increase Φ'. Therefore, for any arrangement of gaps that can be found by the algorithm (that is, for which $w_i \leq i^2$ for $i = 0, \ldots, j-1$ and $w_j \geq j^2 + 1$) there is another such arrangement with equal j and w_j in which $g_i \leq g_{i+1}$ for which Φ' is no larger.

Similar reasoning permits one to further constrain the arrangement so that $g_i < g_{i+1}$ iff $w_{i+1} = (i+1)^2$, since if $w_{i+1} < (i+1)^2$ one could increase w_{i+1}, stopping only when the gaps were equal or w_{i+1} achieved its maximum value.

So, suppose the w_i satisfy these constraints and achieve the minimum possible value of Φ'. We first show that $w_{j-1} = (j-1)^2$. Assume not. Then, $g_{j-2} = g_{j-1} > 2j$. A simple argument by induction shows that this implies that all the gaps must be equal, so $w_{j-k} < (j-1)^2 - 2j(k-1)$, and therefore $w_0 < 0$, a contradiction. Similar argument demonstrate that $w_i = i^2$, $i = 1, \ldots, j-2$. This proves the lemma. ∎

Theorem 3 *The amortized time to do Insert on a list containing n records is $O(\log n)$, and the amortized (and worst-case) time to do Delete or Order is $O(1)$.*

Proof: We assume that n is initially one and is always at least one, and that $M \geq 2$. This implies that Φ is initially zero. Thus, to prove the first part of the theorem it suffices to show that c can be chosen so that for an insertion:

$$\text{(amortized work)} \equiv \text{(work done)} + \text{(change in potential)} \leq c \log n + O(1).$$

We define the work done during an insertion to be the number of records whose labels change. The actual running time of the algorithm is proportional to this number.

Suppose that the insertion does not cause any of the old records to be relabeled. The work done is one (the new record is labeled). To evaluate the change in the potential, we separate it into two components. First, an existing gap is split in two and a new term introduced into equation 4. The increase in Φ is maximized when the gap being is split is of size 2, in which case Φ increases by $c(1 + \log n)$. In addition, n is increased by one. This causes every term in equation (4) to increase by at most $cn^{-1} \log e$, so Φ increases by an additional $c(1 + n^{-1}) \log e$. The total increase in Φ from the actual insertion is therefore at most

$$c \log n + (\log e + 1)c + (\log e)cn^{-1} = c \log n + O(1).$$

In an insertion in which some relabeling is done, the final step is just like that described above. Therefore to complete the proof it suffices to show that the amortized work of relabeling is constant. Consider an insertion in which $j-1$ of the records already in the

list are relabeled, $j > 1$. We shall choose c such that the potential decreases by at least $j - O(1)$.

Let i, j, and w be defined as in the insertion algorithm. Let Φ_0 denote the potential before the insertion. We divide the relabelling process into three phases.

1. The first phase relabels $x_1, \ldots, x_{j-1}$ so that the first $j-1$ gaps are j, and the final gap is $w_j - j(j-1)$. The potential at this point will be denoted Φ_1.

2. The second phase adjusts all the gaps so they are equal. The potential at this point will be denoted Φ_2.

3. The final phase adjusts the gaps g_k (for $0 \leq k < j$) so that they are all integers and the largest and the smallest differ by at most 1. The multiset of gaps resulting is exactly that produced by the insertion algorithm, and thus the resulting potential, Φ_3, is the same as that after the algorithm finishes relabeling.

The decrease in the potential caused by phase 1 is minimized when the contribution to Φ_0 from $g_0, \ldots, g_{j-1}$ is minimized. By lemma 2 this occurs when $g_i = 2i + 1$, $i = 0, \ldots, j-2$. Note that in this case the term corresponding to the last gap increases by at most $c \log \frac{2j}{j+1} < c$ (the gap goes from $2j$ to $j+1$). So,

$$\Phi_1 - \Phi_0 \leq c + \sum_{i=0}^{j-2} c \log \frac{(2i+1)}{j}.$$

Writing the sum of logarithms as the logarithm of a product gives

$$\Phi_1 - \Phi_0 \leq c + c \log \frac{(2(j-1))!}{2^{j-1}(j-1)!j^{j-1}}.$$

We now use Stirling's approximation to the factorial function

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + O(n^{-1}))$$

and simplify to get

$$\Phi_1 - \Phi_0 \leq -cj \log \frac{e}{2} + O(1).$$

Because of the concavity of ϕ, phase 2 cannot increase Φ; i.e.,

$$\Phi_2 - \Phi_1 \leq 0.$$

We now must bound the effect of phase 3 on the potential. Let $D = w_j$ be the total space used by the j modified gaps. In the uniform arrangement of gaps produced by phase 2, there are j gaps of size D/j. In the discrete arrangement produced by phase 3, there are $j \cdot f$ gaps of size $\lceil D/j \rceil$, and $j \cdot (1-f)$ gaps of size $\lfloor D/j \rfloor$, where $f = D/j - \lfloor D/j \rfloor$ is the fractional part of the rational number D/j.

$$\Phi_3 - \Phi_2 = (1-f)jc \log \frac{2n}{\lfloor D/j \rfloor} + f \cdot jc \log \frac{2n}{\lceil D/j \rceil} - jc \log \frac{2n}{D/j}. \tag{6}$$

Defining $x = j/D$, and substituting into (6), we obtain

$$\frac{1}{jc}(\Phi_3 - \Phi_2) = -(1-f)\log(1 - x \cdot f) \; - \; f \cdot \log(1 + x \cdot (1-f)). \tag{7}$$

The partial derivative of the righthand side of equation 7 with respect to x is

$$xf \cdot (1-f)(1 - x \cdot f)^{-1}(1 + x \cdot (1-f))^{-1} \cdot \log e$$

which is nonnegative for $0 \leq f \leq 1$ and $0 < x < 1$. Therefore, since $0 < x < j^{-1}$,

$$\frac{1}{jc}(\Phi_3 - \Phi_2) \leq -(1-f)\log(1 - f \cdot j^{-1}) \; - \; f \cdot \log(1 + (1-f) \cdot j^{-1}). \tag{8}$$

Using the inequality

$$\log\,(1+\delta) \geq \frac{\delta}{1+\delta}\,\log e$$

(for $\delta > -1$), we get

$$\begin{aligned} \frac{1}{jc \log e}(\Phi_3 - \Phi_2) &\leq j^{-2} f \cdot (1-f)(1 - f \cdot j^{-1})^{-1}(1 + (1-f) \cdot j^{-1})^{-1} \\ &\leq f \cdot (1-f) j^{-1}(j-f)^{-1} \\ &= O(j^{-2}). \end{aligned} \tag{9}$$

Therefore,

$$\Phi_3 - \Phi_2 = O(j^{-1}). \tag{10}$$

Putting the three bounds together yields

$$\Phi_3 - \Phi_0 \; \leq \; -jc \log \frac{e}{2} \; + \; O(1).$$

By choosing $c \geq (\log(e/2))^{-1} = 2.25889\cdots$, we get $\Phi_3 - \Phi_0 \leq -j + O(1)$, which is the desired result.

The bound on the amortized time of *Delete* follows by observing that a deletion only causes the potential to decrease. Thus the amortized time of a deletion is at most the worst-case time of a deletion, which is a constant. An order query takes constant worst-case time and does not change the potential, therefore its amortized time is also constant. ∎

Note that the arena size may at any time be increased without violating (1), and that increasing M during the execution of the algorithm cannot cause the potential function in Theorem 3 to increase.

The algorithm could be improved slightly by changing the relabeling scheme so that the first gap is twice the size of the other gaps, or, equivalently, inserting the new record into the first gap before relabeling is performed.

The algorithm can be modified to avoid modular arithmetic. Increase M so that $M > 2n^2$. When renumbering, first determine if $v(x) < M/2$. If so, use the algorithm as

described. Otherwise, replace x with y and renumber in the opposite direction, starting at y. Insert x before y when this renumbering is complete.

For completeness, we briefly outline how this algorithm can be used to obtain an algorithm which uses only $O(1)$ amortized time per operation. The list is represented as a list of sublists. Inside each sublist records are assigned monotonically increasing labels. When a record is inserted into a sublist its label is the average of the labels of its neighbors, or the label of its predecessor plus n if the record in inserted at the end of the sublist.

The algorithm tries to make the length of each sublist about $\log(k+1)$, where k is the number of items in the list. When the algorithm inserts into (or deletes from) a sublist, if the length σ of the sublist is at least $2\log(k+1)$, the list is split into $\lfloor \sigma / \log(k+1) \rfloor$ sublists of size at least $\log(k+1)$, and each sublist is inserted into the list of sublists. By using a potential function of the form

$$\Phi = c \sum_{i=1}^{s} |length(L_i) - \log(k+1)|$$

(where $L_1, \ldots, L_s$ are the sublists) one can show that insertions and deletions take $O(1)$ amortized time. $Order(x, y)$ is implemented by determining the order of the sublists containing x and y (if they are in different sublists), or by comparing the labels of x and y (if they are in the same sublist); this takes constant time.

3. A Real-Time Algorithm

Our real-time algorithm makes use of a technique of Willard [16,17] for performing insertions and deletions in an ordered dense sequential file in time $O(\log^2 n)$ per operation. To apply Willard's algorithm to the order maintenance problem we implement $Order(x, y)$ by comparing the addresses of x and y in the file.

For simplicity, we will mostly ignore deletions and restrict the list to contain at most N records. Our basic idea is to extend the two level scheme of Tarjan and Tsakalidis to four levels, so that Willard's algorithm manipulates a file with $O(N/\log^3 N)$ elements, each a sublist of $O(\log^3 N)$ elements. We need two of these levels because an insertion in Willard's algorithm requires $O(\log^2 N)$ time. We need the third level because of Theorem 5.

The data structure is a tree of height 4. The leaves are all at depth 4 (height 0), while the root has is at depth 0 (height 4). The leaves of this tree are the records in the list. The root has $O(N/\log^3 N)$ children. Nodes at height $h = 1, 2, 3$, called *internal nodes*, are the roots of subtrees with $\leq \log^h N$ leaves.

The algorithm assigns real numbers to the children of each node. For each node, the numbers of its children increase monotonically from left to right. The numbers assigned to children of the root are the positions in the file assigned by Willard's algorithm. Children of internal nodes are numbered using the following simple algorithm:

> When a new child is inserted between two other children, its number is the average of their numbers. When inserting at the beginning (at the end) of a

list of children, the new child's number is one less than (one greater than) the number of its successor (predecessor).

This algorithm requires no renumbering. We will show (Lemma 4) that the internal nodes have $O(\log N)$ children, so this algorithm uses numbers with $O(\log N)$ significant bits.

To answer a query $Order(x, y)$, the algorithm finds the lowest common ancestor of leaves x and y. It determines the order of x and y by comparing the numbers of the two relevant children of this least common ancestor. Under the assumption that numbers with $O(\log N)$ bits can be compared in constant time, *Order* can be computed in constant time.

Whenever a node x of height h is the root of a subtree with more than $\log^h N$ leaves, the algorithm splits x into two new nodes, called *overflow nodes*, which are inserted before and after x, as children of x's parent. (This process is analogous to the process of inserting into a b-tree, except that a node splits when the number of leaves in the subtree rooted there gets too large.)

Because we can not afford to immediately spend the $O(\log N)$ time that may be required to split the node, the children of x are gradually moved into the overflow nodes. Over each of the next $O(\log N)$ insertions that insert into the subtrees rooted at x or its overflow nodes, we move $O(1)$ children of x to the overflow nodes. Children are moved to the overflow node which is the root of the subtree with fewest leaves. Incremental movement can be done fast enough that the overflow nodes cannot themselves split before all of x's children are moved. When x has no more children it can be deleted. Note that order queries are not hindered by nodes that are in the process of splitting.

At any time, we can partition the internal nodes of the tree into three classes: *overflow nodes*, which are being copied into, *splitting nodes*, which are being emptied, and *quiescent nodes*, which are all other internal nodes. The splitting strategy causes the following lemma to hold:

Lemma 4 *Every internal node has $O(\log N)$ children.*

Proof: No internal node is the root of a subtree with more than $log^h N$ leaves, h the height of the node. At most one internal node at each height in the tree was not created by splitting. When a node is created by splitting it is the root of a subtree with at least $(\log^h N - \log^{h-1} N)/2$ leaves (because a node begins splitting when it is the root of a subtree with $\log^h N$ nodes, and children are copied to the overflow node which is the root of the subtree with the fewest leaves). If k is large enough then there cannot be enough time for an overflow node to begin splitting before the node from which its children are being moved is emptied and deleted (at which point the overflow nodes become quiescent). Therefore, any internal node x of height $h = 1, 2, 3$ with c children has at least $(c-1)/3$ children that are the roots of subtrees with $\Theta(\log^{h-1} N)$ leaves, so x has $O(\log N)$ children. ∎

When splitting a depth 1 node, we must insert the overflow nodes into Willard's data structure. Again, we cannot spend the $O(\log^2 N)$ time required to do this immediately.

This case is complicated because queries cannot be done in a partially modified Willard structure. Therefore we must arrange the data structure so that when the split begins the two overflow nodes have already been inserted into the Willard structure. Depth 1 overflow nodes are created and inserted by a process that runs in the background independent of the insertions that are taking place. At each insertion, a constant number of steps are taken in this process, which is outlined below:

1. Find the child of the root x that (a) has not yet had overflow nodes allocated, (b) is the root of a subtree with more than $(2/3)\log^3 N$ leaves, and (c) has the maximum number of leaves of any such child. If there are no such children, wait until one appears.

2. Create two overflow nodes for x.

3. Begin executing Willard's insertion algorithm to insert the two overflow nodes before and after x. This algorithm executes over the next $O(\log^2 N)$ insertions. When done, go back to 1.

To keep this process from interfering with the order queries we must keep both old and new versions of every field in the Willard structure that changes during an insertion. A time stamp is used so that a query can use the appropriate version of a node. When an insertion is finished, the time is incremented so that the new version of the structure is activated. Since only two versions are needed at any time, this technique multiplies space by a constant factor.

We also need to maintain a priority queue to be able to determine where to put the overflow nodes. It is easy to make such a data structure that works in constant time per operation, because depth 1 nodes are inserted when they have exactly $\lfloor (2/3)\log^3 N \rfloor + 1$ leaves (so they are inserted at the bottom of the priority queue) and the number of leaves below a node increases by exactly one for each insertion beneath that node. The number of leaves beneath a node does not decrease until that node begins to split, at which time it has been removed from the priority queue.

We can show that, if we execute Willard's insertion algorithm sufficiently quickly, no depth 1 node will begin splitting before its overflow nodes are ready. This is a consequence of the following theorem:

Theorem 5 *Let $x_1, \ldots, x_n$ be n real valued variables, all initially zero. Repeatedly perform the following procedure:*

1. *Find an i, $1 \leq i \leq n$, such that $x_i = \max_j \{x_j\}$. Set x_i to zero.*
2. *Pick n nonnegative reals $a_1, \ldots, a_n$ such that $\sum_{i=1}^{n} a_i = 1$.*
3. *For $i = 1, \ldots, n$, set x_i to $x_i + a_i$.*

No x_i will ever exceed $H_{n-1} + 1$, where $H_k = \sum_{i=1}^{k} i^{-1}$, the kth harmonic number.

Proof: Observe that the sum of the elements of x is always less than n: it is initially zero, and one iteration of the procedure can increase it from S to at most $1 + \frac{n-1}{n}S$, which is also less than n.

Let $x^{(1)}, \ldots, x^{(m)}$ be a sequence of vectors produced by repeated application of the procedure, where $x^{(1)}$ is the initial zero vector. We will assume m is at least n; if not, pad the beginning of the sequence with additional zero vectors and the following argument will still apply. We will show that the largest element in $x^{(m)}$ is less than $H_{n-1} + 1$. Since m was chosen arbitrarily, this will prove the theorem.

For each $k = 1, \ldots, n$, let π_k be a permutation on $1, \ldots, n$ such that for $i = 1, \ldots, n-1$,

$$x_{\pi_k(i)}^{(m-k+1)} \geq x_{\pi_k(i+1)}^{(m-k+1)}.$$

For $k = 1, \ldots, n$, define

$$S_k = \sum_{i=1}^{k} x_{\pi_k(i)}^{(m-k+1)},$$

that is, the sum of the k largest elements of $x^{(m-k+1)}$. We can show

$$S_k \leq \frac{k}{k+1} S_{k+1} + 1 \quad (k = 1, \ldots, n-1).$$

This is because after the largest element of $x^{(m-k)}$ has been set to zero in step 1, the sum of its k largest elements is at most $\frac{k}{k+1}S_{k+1}$, and S_k is maximized by concentrating all the nonzero a_j's chosen in step 2 on these elements.

By induction,

$$S_1 \leq H_{k-1} + \frac{S_k}{k} \quad (k = 2, \ldots, n).$$

Since S_n is less than n, S_1, the largest value in $x^{(m)}$, is less than $H_{n-1} + 1$.

∎

Note that this bound is tight, since one can make the x_i's exceed any value less than 1 in a finite number of steps, then add H_{n-1} to one of them in $n-1$ more steps by adding $1/(n-1)$ to $n-1$ nonzero elements, then $1/(n-2)$ to $n-2$ nonzero elements, etc.

Theorem 5 implies that at most $O(\log^3 N)$ leaves will be inserted beneath any depth one node before its overflow nodes are ready. To see why this is so, define the *fullness* of a depth one node to be zero if its overflow nodes are ready and

$$\max(0, \text{number of leaves} - \lfloor (2/3)\log^3 N \rfloor)$$

otherwise. During one insertion by Willard's algorithm, the fullness of depth one nodes increases by a total of at most $O(\log^2 N)$. Since the overflow node creation process always picks the node with the highest fullness, by Theorem 5 the fullness of any depth one node is at most $O(H_{N-1}\log^2 N)$, or $O(\log^3 N)$. The constant can be made arbitrarily small (specifically, less than $\frac{1}{3}$) by executing Willard's algorithm with sufficient speed.

We briefly outline how to modify the algorithm so that N can be variable. Initially, N is some power of 2. When the number of elements in the data structure reaches $(1-\epsilon)N$

for some constant $\epsilon > 0$, begin copying the data structure to a larger copy in which N is doubled. Copying is done by performing *Insert* operations using the previous, fixed N algorithm. A constant amount of work in this copying process is done on each of the next $O(N)$ insertions, with a constant chosen to be large enough so that copying is complete before the old data structure overflows. The same technique can be used to handle deletions: when an item is deleted, mark it. Concurrently, make a fresh copy of the data structure, discarding marked leaves. This is reminiscent of incremental garbage collection [1].

The algorithm is presented in detail in the appendix.

4. Remarks

Our first algorithm is remarkably simple compared to previous algorithms for solving this problem. It is also probably the best algorithm to use in practice. One of us [3] has adapted the idea behind the algorithm to solve the problem of maintaining a sparse sorted table in $O(\log^2 n)$ amortized time per insertion, where the table occupies space linear in the number of data items. This simplifies the algorithm of Itai, Konheim and Rodeh [6], which achieves the same amortized time bound.

The technique we used to develop the second algorithm is a first step towards a more general means for converting algorithms efficient in the amortized sense to algorithms efficient in the worst case. We have applied Theorem 5 and the ideas of Section 3 to construct a new data structure for search trees in which a constant number of changes are made to the data structure per update. A search tree with this property is required to make search trees fully persistent. Such a search tree was constructed by an entirely different means in [5].

Our second algorithm is perhaps more complex than necessary. The first step toward creating a simpler algorithm is to devise a method of doing insertions into the list that does $O(\log n)$ renumberings. This would obviate the use of of Willard's technique (which is in itself rather complex), and would reduce the number of levels required from four to three.

Acknowledgements

We thank Charles Leiserson and Bob Tarjan for several interesting and enlightening discussions, and Joe Wald for his helpful comments on early drafts of this paper.

References

[1] Baker, H. G. Jr. List Processing in Real Time on a Serial Computer. *C.ACM* 21(4), April 1978, pages 280-294.

[2] Dietz, P. F. Maintaining Order in a Linked List. *Proc. 14th ACM STOC*, May 1982, pages 122-127.

[3] Dietz, P. F. A Simple Algorithm for Padded Lists. March 1988. In preparation.

[4] Dietz, P. F. and Sleator, D. D. Two Algorithms for Maintaining Order in a Linked List. *Proc. 19th ACM STOC*, May, 1987.

[5] Driscoll, J. R., Sarnak, N., Sleator, D. D., Tarjan, R. E. Making Data Structures Persistent. *Proc. 18th ACM STOC*, May 1986. To appear in *JCSS*.

[6] Itai, A., Konheim, A. G., Rodeh, M. A Sparse Table Implementation of Sorted Lists. IBM Research Report RC 9146, Nov. 1981. See section 9.

[7] Leiserson, C. Personal communication, 1982.

[8] Pugh, W. Efficient Concatenable Ordered Lists. TR 87-831, Department of Computer Science, Cornell University, Ithaca, NY, 1987.

[9] Sleator, D. D., Tarjan, R., E. Amortized Efficiency of List Update and Paging Rules. *C.ACM* 28(2), February 1985, pages 202-206.

[10] Sleator, D. D., Tarjan, R., E. Self-Adjusting Binary Search Trees. *J.ACM* 32(3), July 1985, pages 652-686.

[11] Tarjan, R. E. Personal communication.

[12] Tarjan, R. E. Amortized Computational Complexity. *SIAM J. Alg. Disc. Meth.* 2(6), April 1985, pages 306-318.

[13] Tarjan, R. E. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[14] Tsakalidis, A. K. Maintaining Order in a Generalized Linked List. *Acta Informatica*, 21(1), 1984, pages 101-112.

[15] Wegbreit, B. Retrieval from Context Trees. *Info. Proc. Lett.* 3(4), March 1975, pages 119-120.

[16] Willard, D. E. Maintaining Dense Sequential Files in a Dynamic Environment. *Proc. 14th ACM STOC*, May 1982, pages 114-121.

[17] Willard, D. E. Good Worst-Case Algorithms for Inserting and Deleting Records in Dense Sequential Files. *ACM SIGMOD 86*, May 1986, pages 251-260.

Appendix: Details of the Real Time Algorithm

The real time algorithm manipulates a tree with three kinds of nodes. The leaves represent the elements being inserted into the list; we assume that pointers to the leaves are passed to the algorithm. The root of the tree includes a copy of Willard's data structure, the details of which are unimportant. All other nodes are internal nodes of height 1, 2 or 3.

Nodes are implemented as data structures with the following fields:

nleaves(x): The number of leaves in the subtree rooted at the node x.

children(x): A doubly linked list of children of x, in order from left to right. For leaves this field is null.

index(x): A real number. For any node y, the *index* fields of nodes in the list *children*(y) increase monotonically. For children of the root, the *index* field contains the number assigned to the children of the root by Willard's algorithm, and is implemented using a time-stamp (this detail is omitted below).

splitting?(x): A boolean value. This field is true if x is an internal node other than the root that is being split into two new nodes.

depth(x), *parent*(x): have the obvious meanings. The depth of the root is zero.

There is a priority queue of depth one nodes, *RPQueue*. A node is inserted into this queue when it has $\lfloor (2/3)\log_2^3 N \rfloor + 1$ children, and is removed when its overflow nodes have been inserted.

```
- Return true if x occurs before y in the list
proc Order(x, y)
   x_0 ← x; y_0 ← y;
   for i ← 1 to 4 do
      x_i ← parent(x_{i-1}); y_i ← parent(y_{i-1});
      if x_i = y_i then return index(x_{i-1}) < index(y_{i-1})
      end
   end
end Order
```

Figure 1: Real-Time Algorithm: Order

```
- Insert y after x in the list
proc Insert(x, y)
   x_0 ← x;
   for i ← 1 to 3 do
      x_i ← parent(x_{i-1});
      nleaves(x_i) ← nleaves(x_i) + 1;
      if nleaves(x_i) > (log_2 N)^i ∧ ¬splitting?(x_i)
         then SetUpSplit(x_i) end;
   end;
   if nleaves(x_3) = ⌊(2/3) log_2^3 N⌋ + 1 ∧ ¬splitting?(x_3) then
      Insert x_3 into RPQueue
   else if x_3 is already in RPQueue then
      Move x_3 up one position in RPQueue
   end;
   nleaves(y) ← 1;
   PlaceAfter(x, y);
   for i ← 1 to 3 do
      IncrementalSplit(x_i);
      if x_i ≠ first(children(parent(x_i)))
            then IncrementalSplit(pred(x_i)) end;
      if x_i ≠ last(children(parent(x_i)))
            then IncrementalSplit(next(x_i)) end
   end;
   Do O(1) steps of ReservationProcess
end Insert;
```

Figure 2: Real-Time Algorithm: Insert

```
- Make y a child of parent(x), placing it just after
-  x in the list of children. Assume parent(x) is an
-  internal node other than the root.
proc PlaceAfter(x, y)
   p ← parent(x);
   parent(y) ← p;
   if x = first(children(p)) then index(y) ← index(x) + 1;
      else index(y) ← (index(x) + index(next(x)))/2
   end;
   Insert y after x in children(p)
end PlaceAfter

- Make y a child of parent(x), placing it just before
-  x in the list of children. Assume parent(x) is an
-  internal node other than the root.
proc PlaceBefore(x, y)
   p ← parent(x);
   parent(y) ← p;
   if x = last(children(p)) then index(y) ← index(x) − 1;
      else index(y) ← (index(x) + index(pred(x)))/2
   end;
   Insert y before x in children(p)
end PlaceBefore
```

Figure 3: Real-Time Algorithm: PlaceBefore and PlaceAfter

```
- Prepare an internal node of depth 2 or 3 to split.
- Create two overflow nodes and insert beneath parent(x).
proc SetUpSplit(x)
   if parent(x) ≠ root then
      Create two new nodes, y and z;
      nleaves(y) ← 0;
      nleaves(z) ← 0;
      depth(y) ← depth(x);
      depth(z) ← depth(x);
      PlaceBefore(x, y);
      PlaceAfter(x, z)
   end;
   splitting?(x) ← true
end SetUpSplit
```

Figure 4: Real-Time Algorithm: Prepare to Split a Node

- Copy k children from beneath a splitting node.
- k is a sufficiently large constant.

proc $IncrementalSplit(x)$
  if $\neg splitting?(x)$ **then return end;**
  for $i \leftarrow 1$ **to** k **do**
    if x has no children **then**
      Delete x from $children(parent(x))$;
      return
    end;
    if $nleaves(first(children(x))) \leq nleaves(last(children(x)))$ **then**
      $y \leftarrow first(children(x))$;
      Remove y from $children(x)$;
      $nleaves(x) \leftarrow nleaves(x) - nleaves(y)$;
      $nleaves(pred(x)) \leftarrow nleaves(pred(x)) + nleaves(y)$;
      $PlaceAfter(last(children(pred(x))), y)$
    else
      $y \leftarrow last(children(x))$;
      Remove y from $children(x)$;
      $nleaves(x) \leftarrow nleaves(x) - nleaves(y)$;
      $nleaves(next(x)) \leftarrow nleaves(next(x)) + nleaves(y)$;
      $PlaceBefore(first(children(next(x))), y)$
    end
  end
end $IncrementalSplit$

Figure 5: Real-Time Algorithm: Program for Splitting

```
– The process that guarantees that depth 1 overflow nodes
–  are available when neeed.
process ReservationProcess
  while true do
    Wait until RPQueue is not empty;
    Remove the node from the priority queue that has
     the largest number of leaves, breaking ties arbitrarily;
    Create two new nodes, y and z;
    nleaves(y) ← 0;
    nleaves(z) ← 0;
    depth(y) ← 1;
    depth(z) ← 1;
    PlaceBefore(x, y);
    PlaceAfter(x, z);
    Use Willard's algorithm to insert y and z before and after x;
    Activate new time-stamped indices produced in the previous step;
    splitting?(x) ← true
  end
end ReservationProcess
```

Figure 6: Real-Time Algorithm: Depth 1 Insertion Process