Two Simplified Algorithms for Maintaining Order in a List

Michael A. Bender[1*], Richard Cole[2**], Erik D. Demaine[3***], Martin Farach-Colton[4†], and Jack Zito[1]

[1] Dept of Computer Science, SUNY Stony Brook, Stony Brook, NY 11794, USA. bender@cs.sunysb.edu. JackZito@JackZito.com.

[2] Courant Institute, New York University, 251 Mercer Street, New York, NY 10012, USA. cole@cs.nyu.edu.

[3] MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139, USA. edemaine@mit.edu.

[4] Google Inc., 2400 Bayshore Parkway, Mountain View, CA 94043, USA, and Department of Computer Science, Rutgers University, Piscataway, NJ 08855, USA. farach@cs.rutgers.edu.

Abstract. In the *Order-Maintenance Problem*, the objective is to maintain a total order subject to insertions, deletions, and precedence queries. Known optimal solutions, due to Dietz and Sleator, are complicated. We present new algorithms that match the bounds of Dietz and Sleator. Our solutions are simple, and we present experimental evidence that suggests that they are superior in practice.

1 Introduction

The *Order-Maintenance Problem* is to maintain a total order subject to the following operations:

1. Insert (X, Y): Insert a new element Y immediately after element X in the total order.
2. Delete (X): Remove an element X from the total order.
3. Order (X, Y): Determine whether X precedes Y in the total order.

We call any such data structure an *order data structure*.

1.1 History

The first order data structure was published by Dietz [4]. It supports insertions and deletions in $O(\log n)$ amortized time and queries in $O(1)$ time. Tsakalidis improved the update bound to $O(\log^* n)$ and then to $O(1)$ amortized time [10].

* Supported in part by HRL Laboratories, NSF Grant EIA-0112849, and Sandia National Laboratories.

** Supported in part by NSF Grants CCR-9800085 and CCR-0105678.

*** Supported in part by NSF Grant EIA-0112849.

† Supported in part by NSF Grant CCR-9820879.

The fastest known order data structures was presented in a classic paper by Dietz and Sleator [3]. They proposed two data structures: one supports insertions and deletions in $O(1)$ amortized time and queries in $O(1)$ worst-case time; the other is a complicated data structure that supports all operations in $O(1)$ worst-case time.

Special cases of the order-maintenance problem include the *online list labeling problem* [1, 5–7] also called the *file maintenance problem* [11–14]. In online list labeling, we maintain a mapping from a dynamic set of n elements to the integers in the range from 1 to u (*tags*), such that the order of the elements matches the order of the corresponding tags. Any solution to the online list labeling problem yields an order data structure. However, the reverse is not true: there is an $\Omega(\log n)$ lower bound on the list labeling problem [5, 6]. In file maintenance, we require that $u = O(n)$, since it is the problem of maintaining a dynamic file stored sequentially in memory with gaps for insertions.

1.2 Results

$O(1)$ Amortized Order Data Structure: The drawback of Dietz and Sleator's amortized $O(1)$ solution is its proof. It relies on a complicated potential-function argument, which is not just unintuitive, but counterintuitive. This data structure assigns tags to elements *à la* online list labeling[5]. If we use more bits for the tags (say, $3 \log n$ rather than $2 \log n$ bits per tag), the amortized number of relabels *should* decrease. However, the proof of the Dietz and Sleator algorithm gives a paradoxical *increasing* upper bound on the amortized number of relabels. This was cited in their paper as a shortcoming of their analysis.

We present a similar list labeling-based algorithm that has several advantages. First, the proof is straightforward. Second, it yields a sensible tradeoff between tag size and relabeling. We verify experimentally that this tradeoff is essentially tight for our algorithm.

Furthermore, our analysis yields a counter-intuitive insight. Intuitively, it seems advantageous to assign the tag of a new element to be the average of its neighbors. Otherwise, an adversary could insert in the smaller half, thus inducing more relabelings. Indeed, Dietz and Sleator's proof relies on this average-tag choice. Our proof is independent of the tag-choice strategy; thus, the bound holds no matter how the tag is chosen. Perhaps our bounds are simply tighter for bad choices of tags. However, we confirm the insensitivity to tag selection experimentally.

In summary, we present a simple amortized solution along with a simple proof. This proof elucidates the tradeoffs and quirks of the algorithm, and we verify these experimentally. Furthermore, we show experimentally that our solution dominates Dietz and Sleator's in terms of the bit/relabel tradeoff.

[5] They use a standard trick of indirection to overcome the list labeling lower bound. We use the same trick.

$O(1)$ Worst-Case Order Data Structure: The Dietz and Sleator worst-case constant-time solution relies on an algorithm by Willard [11–14] for the *file maintenance problem.* Instead, we give a direct deamortization of the simple amortized algorithm above. Due to space considerations, we defer the discussion of this algorithm to the full paper.

$O(\log^2 n)$ Worst-Case File Maintenance: This result matches the bounds of the afore-mentioned Willard result. However, a complete description of our result takes 5 pages (See Section 5), rather than Willard's 54.

2 Preliminaries

2.1 Indirection: Shaving a Log

Below, we give a solution to the order-maintenance problem that takes $O(\log n)$ amortized time to update and $O(1)$ worst-case time for queries. The standard technique of having a two-level data structure, where the bottom level has $\Theta(\log n)$ elements and the top level has $\Theta(n/\log n)$ elements can be applied to any data structure with such a complexity to improve the update time to $O(1)$ amortized time, while keeping $O(1)$ worst case time queries.

We omit the details, which can be found, e.g., in [3].

2.2 Connection to Weight-Balanced Trees

Our order maintenance algorithms rely on a close connection between list labeling and weight-balanced trees. Suppose we have a list where each element has been assigned an integer tag from $[0, u)$. The bits of the tags implicitly form a binary tree, which is the *trie* of the bit strings. We call this trie the *virtual tree.*

The leaves of the virtual tree correspond to the tags, with each root-to-leaf path specifying the tag's binary representation. The depth of the virtual tree is $\log u$. As we insert new elements into the list, we assign them new tags, effectively inserting new leaves into the virtual tree. When we run out of tags, we redistribute tags within some subrange of the list, and this redistribution corresponds to a rebalancing operation in the implicit tree.

The next two lemmas give the algorithmic connection between this tree viewpoint and the list-labeling viewpoint:

Lemma 1. *Any strategy for maintaining n tags in the universe $[0, u)$, $u = n^c$ for some constant c, with amortized relabel cost $f(n)$ yields a strategy for maintaining balanced binary trees of height $\log u$ with amortized insertion time of $O(f(n))$.*

If the converse were true, we would immediately have an $O(1)$ amortized solution to the order maintenance problem (e.g., by using red-black trees plus the indirection noted above). However, updates in balanced tree structures might change many root-to-leaf paths. For example, a root rotation in red-black tree changes the path label of every leaf. Therefore, the cost of modifying (i.e., splitting/merging/rotating) a node v in the virtual tree is proportional to the *number of descendants* of v, which we refer to as the *weight* of v.

The *weight cost* of an operation on a tree is the running time of the operation, plus the sum of the weights of all modified nodes. For example, the red-black tree has $O(\log n)$ weight-cost searches and $O(n)$ weight-cost insertions and deletions. This definition extends naturally to *amortized weight cost*, and we note that red-black trees have $O(n)$ amortized weight-cost insertion and deletions as well.

Theorem 1. *Any balanced-tree structure with (amortized) weight cost $f(n)$ for insertions, maximum degree d, and depth h yields a strategy for list labeling with (amortized) cost $O(f(n))$ and tags from universe $[0, d^h)$.*

We require a balanced-tree structure where the *weight costs* of all operations are small. Fortunately, a few such trees exist, such as BB[α] trees [8], skip lists [9], and weight-balanced B-trees [2]. All these trees are *weight balanced*, in the sense that, for every non-leaf node, the weights of its children are within constant factors of each other.

These tree, when combined with indirection, give simple constant-time amortized solutions to the order-maintenance problem. The advantage of our solution below, as well as Dietz and Sleator's, is that it does not maintain the tree explicitly.

2.3 Roadmap

In Section 3, we show an amortized order-maintenance data structure that does not store the weight-balanced tree explicitly. The algorithm is similar to Dietz and Sleator's original algorithm, but our analysis is simple enough to teach to undergraduates, and it elucidates the performance issues. These performance issues are further explored and experimentally verified in Section 4. In Section 5, we give an $O(\log^2 n)$ algorithm for file-maintenance.

3 The Amortized Order-Maintenance Algorithm

Our algorithm does not use an explicit tree, but our analysis does. Let u, the tag universe size, be a power of two, and consider the complete binary tree in which the leaves represent the u tags. At any state of the algorithm, n of these leaves are occupied, namely, the leaves that correspond to the tags used to label list elements.

When we insert an element f between elements e and g, we are free to choose any tag between the tags of e and g. If e and g occupy adjacent leaves (their tags differ by 1), then we relabel a sublist of the linked list to make room for the new element.

Each internal node of the tree corresponds to a (possibly empty) sublist of the linked list, namely those linked elements that occupy leaves below the node.

Each leaf has $\log u$ such *enclosing tag ranges*. The *density* of a node is the fraction of its descendant leaves that are occupied.

When we insert f between e and g, if we need to relabel, we examine the enclosing tag ranges of e until we find the smallest enclosing tag range with low-enough density. We calculate the density by traversing the elements within a certain tag range (instead of walking up the implicit tree).

It remains to define when an enclosing region is considered "sparse enough." We define the *overflow threshold* as follows. Let T be a constant between 1 and 2. For a range of size 2^0, that is, a leaf of the tree, the overflow threshold τ_0 is 1. Otherwise, for ranges of size 2^i, $\tau_i = \tau_{i-1}/T = T^{-i}$. We say that a range is *in overflow* if its density is above the overflow threshold.

We summarize the algorithm as follow: relabel the smallest enclosing range that is not overflowing. Note that larger enclosing ranges may be in overflow. We lazily stop at the smallest non-overflowing range.

This algorithm is similar to Dietz and Sleator's. Whereas we scan the list in geometrically increasing tag ranges, they scan the list in geometrically increasing number of list elements. Whereas we have a stopping condition that is a function of the density and size of our subrange, their stopping condition is that the density drops by some constant between sublists. We therefore call our algorithm the *Tag-range Relabeling Algorithm*, and Dietz and Sleator's the *List-range Relabeling Algorithm*.

We now show that the labels assigned to list elements are logarithmic in length, and that the amortized cost of an insertion is $O(\log n)$.

Lemma 2. *The Tag-range Relabeling Algorithm uses $O(\log n)$ bits per tag.*

Proof. Note that u is a function of n, but n may change wildly. Thus, we pick a u that works for any value from $n/2$ to $2n$. If at any point we have too few or too many list elements, we rebuild the data structure for the new value of n. This rebuilding introduces only a constant amortized overhead.

The root range must never be allowed to overflow. It is the subrange of last recourse. That is, we should always be able to relabel at the root. The density at the root is at most $2n/u$, because we rebuild if we get more than $2n$ elements. So we set $\tau_{\log u} = T^{-\log u} = u^{-\log T} = 2n/u$. Thus, $u = (2n)^{1/1-\log c}$, so we use $\log n/(1-\log T)$ bits. □

We conclude that the worst-case query complexity is $O(1)$ since tags fit in a constant number of machine words.

Lemma 3. *The amortized cost of insertion in the Tag-range Relabeling Algorithm is $O(\log n)$.*

Proof. When we relabel a range of tags of size 2^i, the resultant density throughout the range becomes uniformly no more than τ_i, because we only relabel ranges that are not overflowing. The time complexity for this relabeling is $O(2^i \tau_i) = O((2/T)^i)$.

Before we relabel this range again, one of the two child subranges must overflow. After the relabeling, the density of the child subranges is at most τ_i

and when they next overflow their density will be at least τ_{i-1}. This requires at least $(2 - T/2)(2/T)^i$ insertions, thus yielding an amortized insertion cost of $2 - T/2 = O(1)$.

Finally, by the previous lemma, each insertion has $O(\log n)$ enclosing subranges. Because the amortized cost *per enclosing subrange* is $O(1)$, the total amortized cost is $O(\log n)$. $\square$

Using one level of indirection, we obtain

Theorem 2. *The order-maintenance problem can be solved in $O(1)$ amortized insertion and deletion time and $O(1)$ worst-case query time.*

4 Experiments

4.1 Choosing Tags

Surprisingly, the method for picking the label of the new element does not play a rôle in the proof. Consider two cases. The natural choice for a label is the average of the neighbors, as in Dietz and Sleator. A seemingly worse choice for the label is one more than the predecessor's label, which is vulnerable to adversarial relabeling. We refer to these strategies as *Average Inserts* and *Consecutive Inserts*, respectively.

Our proof suggests that the two schemes have equal performance. But it may be that our bound is simply tighter for the Consecutive Inserts case, and that Average Inserts is in fact a constant factor faster. To determine the constant factors, we performed the following experiment. We inserted n elements, each at the front. Thus, the Consecutive Inserts algorithm generates a relabeling at every other step.

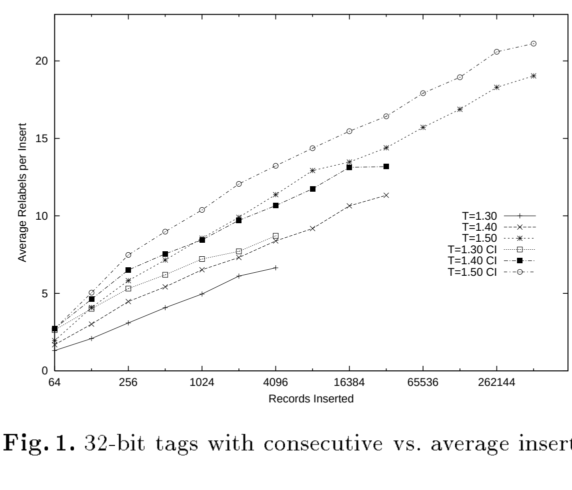

Fig. 1. 32-bit tags with consecutive vs. average inserts.

It turns out that the total number of relabelings is almost the same for both schemes, with only an additive-constant overhead. See Figure 1. What appears to be happening is the following: Consecutive inserts trigger relabels after every insert, but the number of elements relabeled is small. On the other hand, average inserts have fewer triggers, and the number of elements relabeled is larger. With Consecutive Inserts, the average number of relabels per element remains only a small additive amount higher.

4.2 Space/Time Tradeoff

Note that the number of bits needed for an order maintenance structure that hold n elements is $\log n/(1 - \log T)$, where T is between 1 and 2. On a real machine, the number of bits is fixed, and thus there is a tradeoff between n and T. Recall that the amortized cost of insertions is $(2 - T/2)(\log n)/(1 - \log T)$. Thus, as we require our machine word to serve for more and more elements, we require a larger T, the cost of an insertion increases. While this is not surprising, Dietz and Sleator's proof had a paradoxical, opposite tradeoff in the bounds.

In our proof, we concentrate on the duration of time that the root range does not overflow, and when it does, we increase our word size and rebuild the structure. This, of course, does not work on a real computer. So for each T, the order data structure should have some maximum capacity, after which the root overflows, and at some point after which we start thrashing with relabels. In the next experiment, we measured how close to theoretical overflow we get thrashing, for a variety of choices of T, and for Dietz and Sleator's List-range algorithm.

Figure 2 shows relabels per insertion as we insert more and more elements into a list with a variety of choices of T, as well as for Dietz and Sleator (labeled DNS in the figure). The theoretical capacity given by each choice of T is indicated with an arrow.

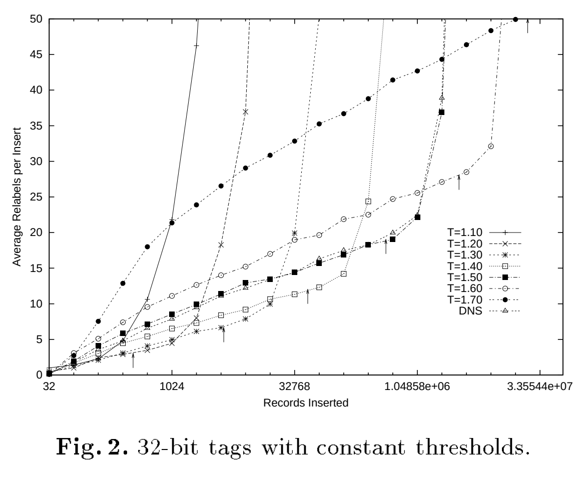

Fig. 2. 32-bit tags with constant thresholds.

In Figure 2, the y-axis represents the average number of relabels per inserted element. The x-axis represents the number of elements inserted. For this graph, lower points are better, representing fewer relabels per element inserted. Note that for any given threshold T, the algorithm thrashes soon after the "rated" maximum limit for that T has been exceeded. It is interesting to note that a threshold of $T = 1.5$ is comparable to Dietz and Sleator's algorithm for "worst case" inserts. While Dietz and Sleator's analysis suggests that the capacity of their structure is approximately 64,000, our structure with $T = 1.5$ is rated to approximately 430,000 elements. Thus Dietz and Sleator's analysis has substantial slack.

4.3 Variable *T*

Finally, note that although standard RAM models assume $O(\log n)$ word size, computers have fixed word sizes. The choice of T depends on n, the number of inserted elements. For any particular n, we would like T to be as small as possible, to reduce the cost of insertions. One final note about our algorithm is that there is nothing in our analysis that requires T to be fixed. At any step, we can calculate the smallest T that keep the root range from overflowing, and we use that T to compute the τ's. We call this algorithm the *Variable Threshold* Algorithm. Note that this algorithm does not produce theoretical improvements over the fixed-threshold algorithm, because they assume incomparable machine models. However, it seems to perform well, as Figure 3 shows.

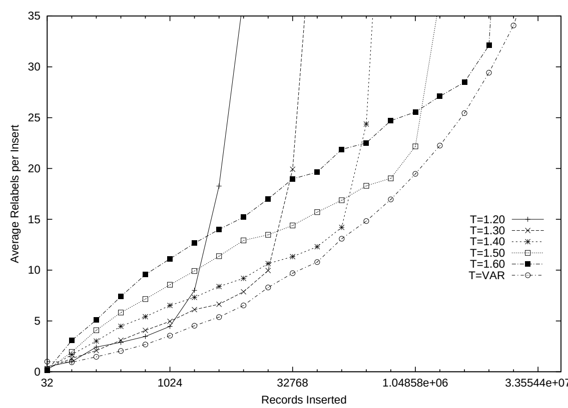

Fig. 3. 32-bit tags with fixed versus variable thresholds.

5 File Maintenance

We begin our description by showing how to store n items in an array of size $O(n)$. Four operations are supported:

- Insert(e, f): Given a pointer to already stored item e, insert f immediately after e.
- Delete(e): Given a pointer to item e, delete it.
- Scan-right(k, e): Given a pointer to item e, scan the next k items in the list starting at item e.
- Scan-left(k, e): Analogous to scan-right.

Our solution stores the items in list order in the array. Note that the order-operation is trivially implemented by index comparison. The insert and delete operations require $O(\log^2 n)$ worst case time and the scan requires $O(k)$ time. Willard gave a data structure with this performance. Unfortunately, his solution is quite involved.' (The journal version requires some 54 pages).

As in Willard's data structure, the array is implicitly recursively partitioned into an $O(\log n)$ level balanced binary tree. The leaves correspond to length $a \log n$ intervals of the array for a suitable constant a; we call such intervals *cells*. As is standard, the nodes in the tree are assigned levels, incrementing from the leaf level and with the leaves having level zero. The subinterval corresponding to a node at level i comprises 2^i cells, and is called an i-interval. It is convenient to extend tree terminology (e.g., parent, child, sibling) to the i-intervals.

The key invariant below ensures there is always room to insert an item.

Invariant 1.
(i) An i-interval stores at most $(a \log n - 2i) \cdot 2^i$ items.
(ii) The number of items stored in sibling i-intervals differ by at most $2 \cdot 2^i$ items.

The basic update operation is the following: when the number of items stored by two sibling intervals differ by $2 \cdot 2^i$ items, 2^i items are redistributed from the fuller to the less full sibling, one item per cell. This redistribution is performed over the next 2^i insertions and deletions occurring on these two intervals, $O(\log n)$ operations being performed per insertion or deletion. The process is called an $(i+1)$-rebalancing. The work performed during a single insertion or deletion on a single rebalancing operation is called a *phase* of the rebalancing. As each insertion or deletion lies within $O(\log n)$ intervals, the $O(\log^2 n)$ time bound follows immediately.

In fact, Invariant 1(ii) might not be maintained during the course of a rebalancing, so we modify it as follows:

Invariant 1(ii).
(ii) The number of items stored in sibling i-intervals differ by at most $2 \cdot 2^i$ items, except possibly when a task is rebalancing an ancestral $(j+1)$-interval, $j \geq i$, and the task is moving items into or out of the i-intervals in question.

Since only 2^i insertions or deletions are performed during the course of a rebalancing, clearly Invariant 1(ii) holds after the rebalancing, given that it held at the start. Justification of Invariant 1(i) is deferred. In order to detect when the condition for rebalancing holds, with each node of the tree, we record its imbalance, namely how many more items its left subinterval holds than its right subinterval. The imbalance is stored in the cell immediately to the left of the middle of the interval.

The main part of the implementation lies in the rebalancing procedure itself. The challenge is to show that the rebalancings at different levels of the tree do not interfere.

WLOG, we consider a rebalancing of an $(i+1)$-interval whose left half is fuller. The rebalancing scans the $(i+1)$-interval from right to left, shifting items to the right, with the net effect of adding one item to each cell in the right half and removing one item from each cell in the left half. Ignoring the effect of further insertions and deletions, it is clear that the balance of any subinterval of the $(i+1)$-interval is unchanged.

In order to control the interaction of the various rebalancings, we need to store $O(i)$ auxiliary data with an i-rebalancing and $O(1)$ data with each cell.

To understand the data stored with a rebalancing process, it is helpful to consider the arrangement of data in an $(i+1)$-interval that is being rebalanced: at the right end, there will be data that has already been copied over to the new density, in the middle there will be a gap with empty cells, and at the left end will be the data that has yet to be moved. Finally, there may be two partially filled cells, one at either end of the gap. The algorithm is slightly simplified by requiring that at the end of its phase, the cell at the front (left end) of the gap be empty (in the case of a gap less than one cell in length, we consider the cell contents to be to the rear of the gap). We need to record how the data is to be distributed in the gap, and what density of data is to be moved from the left end (this is non-trivial owing to the interaction with rebalancing procedures associated with subintervals of the $(i+1)$-interval, as will become clear).

More specifically, the left end, comprising the as-yet-not-moved data, is partitioned into up to $i+2$ maximal h-intervals, $h \leq i$. For each such interval, we record an integral number of items to be copied to or from each cell in the interval (it is the same number for every cell). This data is stored as a linked list, called the *increment list*. At the start of a rebalancing there will be two i-intervals, one with count 1 and the other with count -1.

Similarly, the gap is partitioned into up to $2i$ maximal h-intervals, $h < i$. For each interval, we record how many items are to be moved into it. The partially filled cell at the rear of the gap, if any, forms a one-cell interval within the gap. This data is also kept as a linked list, called the *gap list*.

Invariant 2. The number of items to be moved into the gap, as recorded in the list of gap intervals, equals the change to the number of items in the “left end”, as recorded in the list of intervals for the left end.

Both lists are kept in interval order, to facilitate combining of intervals as needed. When the contents of a cell are moved the natural changes are made to the lists. One detail remains to be specified: as the right end of the gap is filled, the interval at the right end of the gap will be split in two: the associated count is simply divided as evenly as possible. This has the effect of somewhat smoothing the local fluctuations in density within the $(i+1)$-interval.

In order to avoid unnecessary rebalancings at $(h+1)$-subintervals due to the gap created in the $(i+1)$-interval by its rebalancing, we do not alter the imbalance value for the $(h+1)$-subinterval due to the data movements of the

$(i+1)$-rebalancing. Of course, if the gap of the rebalancing includes the whole $(h+1)$-subinterval, as the subinterval begins to be filled, its imbalance is reset to $-1, 0$, or 1, as appropriate.

Thus we understand Invariant 1(ii) as applying to the imbalances apparently recording the relative number of items in a pair of subintervals as opposed to the actual number which may differ due to a rebalancing of a larger interval containing these two subintervals.

The only difficulty with multiple rebalancing operations occurs when two gaps meet. There are just two possibilities: either they cross or they combine, as we detail. The task rebalancing the larger subinterval is named *Big* and the task for the smaller subinterval, contained in *Big*'s subinterval, is named *Small*.

WLOG, suppose that *Big* is moving items from left to right.

Case 1. *Small* moves items from left to right.
Case 1a. The gap for *Small* is to the left of the gap for *Big* when they meet (due to *Big*'s progress).
Then the rebalancing for *Small* is terminated, and *Big*'s process takes over Small's task by combining *Small*'s list with its own, as follows. Basically, the collections of lists are merged so as to record the desired incremental changes to intervals that have not had items moved, and to record the desired size for intervals in the gap. This may entail the splitting or joining of some intervals in order to combine information of equal sized intervals and to then produce maximal intervals once more.
Case 1b. The gap for *Big* overlaps the right end of *Small*'s interval or is just to the right of the right end of *Small*'s interval when *Small* is initiated, so in the latter case *Small*'s gap meets *Big*'s gap as *Small* is initiated.
Small is terminated, and its lists combined with those for *Big* as in Case 1a.
Case 1c. The gap for *Big* is to the left of the gap for *Small* when they meet (due to *Small*'s progress). This is handled as in Case 1a.

Each of the above cases uses $O(\log n)$ time.

Case 2. *Small* moves items from right to left. Details left for the full paper.

To ensure an $(i+1)$-rebalancing completes within 2^i phases, in each phase the contents of one cell are moved from the front end to the rear end of the rebalancing's gap. To ensure a smooth interaction with the processing of Cases 1 and 2 above, operations are ordered as follows. In a phase, first, adjacent gaps moving in opposite directions, if any, are merged; then the contents of a cell are moved across the gap; finally, adjacent gaps moving in the same direction, if any, are merged.

Lemma 4. *Invariant 2 is unaffected by the processing of Cases 1 and 2 above and by insertions and deletions.*

Lemma 5. *Invariant 1(ii) holds.*

Proof. A rebalancing of an $(i+1)$-interval is completed within 2^i insertions and/or deletions to the interval, and thus the difference between the number of items in the two halves remains bounded by $2 \cdot 2^i$. The only reason it would not finish its task is through its being terminated and its work taken over by a $(j+1)$-rebalancing, for some $j > i$. But whenever the $(i+1)$-rebalancing would have performed a phase, the $(j+1)$ rebalancing would also perform a phase, and hence the $(j+1)$-rebalancing will finish the processing of the $(i+1)$-interval at least as soon as the $(i+1)$-rebalancing would have. Consequently Invariant 1(ii) is maintained.

Lemma 6. *Invariant 1(i) holds.*

Proof. Left to the full version.

References

1. A. Andersson and O. Petersson. Approximate indexed lists. *Journal of Algorithms*, 29(2):256–276, 1998.
2. L. Arge and J. Vitter. Optimal dynamic interval management in external memory. In *FOCS*, 1996.
3. P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *STOC*, 1987.
4. P.F. Dietz. Maintaining order in a linked list. In *STOC*, 1982.
5. P.F. Dietz, J. I. Seiferas, and J. Zhang. A tight lower bound for on-line monotonic list labeling. In *SWAT*, 1994.
6. P.F. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In *SWAT*, 1990.
7. A. Itai, A. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *ICALP*, 1981.
8. J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIComp*, 2:33–43, 1973.
9. W. Pugh. Skip lists: a probabilistic alternative to balanced trees. In *WADS*, 1989.
10. A.K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–112, May 1984.
11. D. Willard. Inserting and deleting records in blocked sequential files. Technical Report TM81-45193-5, Bell Laboratories, 1981.
12. D. Willard. Maintaining dense sequential files in a dynamic environment. In *STOC*, 1982.
13. D. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *SIGMOD*, 1986.
14. D. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, 1992.

A Final Thought

Dietz and Sleator is quite influential
With its tags and its proofs by potential
But to teach it in class
Is a pain in the —
So our new result is preferential.