PREPRINT. Journal of Algorithms 29:256-276, 1998.

# Approximate indexed lists

Arne Andersson* Ola Petersson†

**Abstract**

Let the position of a list element in a list be the number of elements preceding it plus one. An *indexed list* supports the following operations on a list: Insert; delete; return the position of an element; and return the element at a certain position. The order in which the elements appear in the list is completely determined by where the insertions take place; we do not require the presence of any keys that induce the ordering.

We consider *approximate indexed lists,* and show that a tiny relaxation in precision of the query operations allows a considerable improvement in time complexity. The new data structure has applications in two other problems; namely, *list labeling* and *subset rank.*

## 1 Introduction

An *indexed list* [5] is a list abstract data type that supports the following operations:

*Insert*$(x, y)$**:** Insert list element $y$ immediately after list element $x$, which may be a list header;

*Delete*$(x)$**:** Delete list element $x$;

*Pos*$(x)$**:** Return the position of list element $x$, that is, one plus the number of list elements preceding $x$;

*Rep*$(i)$**:** Return the list element at position $i$.

In the first three operations, a reference to the parameter $x$ is provided, and so no searches are needed. It is important to observe that the order in which the elements appear in the list is completely determined by where the insertions take place. A data structure that implements indexed lists does not rely on the presence of any keys that induce the ordering.

Note that operations like successor, predecessor, return first, and return last can easily be supported in constant time by augmenting any solution to the indexed list problem, by threading the list elements.

Using a balanced tree scheme it is easy to devise a data structure for indexed lists which supports all the above operations in $O(\log n)$ worst-case time, where $n$ denotes the number of elements currently in the list. Dietz [5] gave a more efficient solution in which all operations run in $O(\log n / \log \log n)$ amortized time. This is optimal due to a lower bound of Fredman and Saks [10].

*Department of Computer Science, Lund University, Box 118, 221 00 Lund, Sweden. arne@dna.lth.se

†Prodigy, 44 South Broadway, White Plains, NY 10601, USA. ola@prodigy.net

In this paper we consider *approximate indexed lists,* which support the same updates as indexed lists, while answers to queries only approximate the exact answers. We show that by slightly relaxing the precision, the time complexity can be improved significantly. The new data structure has two interesting applications: in *list labeling* and in *subset rank;* both of which are discussed in detail below.

For further eading on approximate data structures, we refer to Matias, Vitter, and Young [13]

### 1.1 Approximate indexed lists

Let $\epsilon$ be a small positive constant. (By using the well-known technique of global rebuilding, where the entire data structure is reconstructed at periodic intervals, we may let $\epsilon$ vary with $n$.) An $\epsilon$*-approximate indexed list* is a list abstract data type that supports the following queries:

*Approx-Pos*$(x)$**:** Return an integer value such that

$$Pos(x) \leq \textit{Approx-Pos}(x) \leq (1+\epsilon)Pos(x).$$

*Approx-Rep*$(i)$**:** Return an element $x$ such that

$$Pos(x) \leq i \leq (1+\epsilon)Pos(x).$$

The choice of a biased approximation is justified below. To make the approximate solution mimic the exact version as closely as possible, and to increase its applicability, we require the following additional conditions to be satisfied:

*Consistency:* $\textit{Approx-Rep}(\textit{Approx-Pos}(x)) = x$;

*Monotonicity:* For any elements $x$ and $y$, $\textit{Approx-Pos}(x) < \textit{Approx-Pos}(y)$ if and only if $Pos(x) < Pos(y)$;

The reason why the approximation is slightly biased is now evident: since we restrict the answers of *Approx-Pos* to be integers, then by the pigeonhole principle the monotonicity condition implies that $\textit{Approx-Pos}(x)$ cannot be less than $Pos(x)$.

As for consistency, note that we can *not* hope to achieve

$$\textit{Approx-Pos}(\textit{Approx-Rep}(i)) = i, \tag{1}$$

since, unless the answers are exact, the domain for *Approx-Rep* is strictly larger that the domain for *Approx-Pos*. (Intuitively, the $n$ elements will be spread among more than $n$ positions, and so some positions will be empty.) However, our solution guarantees that, *if* there exists an element $x$ such that $\textit{Approx-Pos}(x) = i$ then Equality (1) does hold.

We devise a data structure for $\epsilon$-approximate indexed lists that obeys the consistency and monotonicity conditions, and supports queries in constant worst-case time and updates in $O(1/\epsilon^2)$ amortized time. Thus, if $\epsilon$ is regarded as a small constant then all operations run in $O(1)$ time.

### 1.2 Application: List labeling

In a *labeled list* [4, 6, 14], each element has an integer label, such that if the list is traversed from head to tail, the labels increase monotonically. The updates are the same as for indexed lists. The query operation $x$) simply returns the label of list element $x$.

A solution to the list labeling problem allows us to dynamically answer queries of the form "which comes first in the list, $x$ or $y$?" in constant time by comparing their labels. This, in turn, can be used to support ancestor queries, that is, "is $x$ an ancestor of $y$?", in rooted trees under insertions and deletions, as shown by Dietz [4]. In the same article, Dietz showed how a solution to the latter problem can be applied to implement context trees. Another application arises in the implementation of fully persistent data structures [8]. More recently, list labeling has found an application in an algorithm for graph drawing [2].

In order to understand why list labeling is nontrivial, consider an insertion. If the labels were allowed to occupy any number of bits, the new element could be assigned a label which equals the mean of the labels of its two neighbors. If many insertions occur in a small region of the list, this scheme could result in labels of linear size (in the number of bits) which are too costly to manipulate. Assuming that words of $O(\log n)$ bits can be read in constant time, the challenge is to maintain a label space of size polynomial in the number of list elements $n$, since then each label would fit in a constant number of words. It is easy to see that if an adversary repeatedly inserts a new element at the point in the list where the gap between the labels of two consecutive elements is the smallest, after a logarithmic number of insertions some elements need to be relabeled to accommodate the new element. To keep the update cost low, we should try to perform as few relabelings as possible.

An ideal solution would support updates in constant (amortized) time, that is, on average only a constant number of list elements should be relabeled during an update. However, Dietz, Seiferas, and Zhang [6] proved a lower bound of $\Omega(\log n)$ on the amortized time per update. Moreover, Dietz [4] described a tree-based data structure which achieved this bound. Itai, Konheim, and Rodeh [11] showed that the label space can be reduced to $2n$ at the price of an increase to $O(\log^2 n)$ amortized insertion time; they failed to prove the same time bound for deletions.

As mentioned above, the reason why the labels should be integers of size polynomial in $n$ is that then they can be stored and read in constant time on a RAM. In the light of this it is reasonable to drop the requirement that every element must have an explicit label stored with it at every time, and only require that its label can be *computed* in constant time. The lower bound by Dietz, Seiferas and Zhang [6] counts the number of labelings and relabelings needed to maintain explicit labels and does not apply to this relaxed, natural model. However, most prior work has assumed the relaxed model. (In some applications, though, explicit labels are required; e.g., when the labels are used as addresses into some other data structure.)

In the relaxed model, Tsakalidis [14] showed that $O(1)$ amortized time per update suffices. This was later improved by Dietz and Sleator [7], who gave a simpler data structure which achieves the same bound, as well as a quite sophisticated solution which attains constant time in the *worst* case.

When studying the list labeling problem there seems to be a tradeoff between label space and update time: the bigger the gap in label space between consecutive list elements, the more insertions can be accommodated at low cost. Of the results mentioned above, the ones achieving constant update time use super-linear label space [7, 14], and the one that uses linear label space requires super-constant update time [11].

**Results.** By the monotonicity condition, the answers of the *Approx-Pos* operations can be used as labels. Therefore, we can solve the list labeling problem with label space $\{1, 2, \ldots, (1+\epsilon)n\}$, supporting retrieval of a label in $O(1)$ time and updates in $O(1/\epsilon^2)$ amortized time. This improves significantly upon the tradeoff between label space and update time. Moreover, if explicit labels are required, we improve upon the results by Itai, Konheim, and Rodeh [11] (cf. Corollary 8).

### 1.3 Application: Subset rank

*Subset rank* is a fundamental data structuring problem. Consider the representation of a subset $S$ of $\{1, 2, \ldots, n\}$ under the operations:

*Insert*$(i)$**:** Insert element $i$ into $S$;

*Delete*$(i)$**:** Delete element $i$ from $S$;

*Rank*$(i)$**:** Return the number of elements in $S$ that are smaller than $i$.

Fredman and Saks proved an $\Omega(\log n/\log\log n)$ lower bound on the amortized time on at least one of the operations. A matching upper bound can be derived in a straightforward way from an algorithm by Dietz [5].

**Results.** If one is satisfied with approximate answers to rank queries, our data structure can be augmented to support all operations in $O(\log\log n)$ amortized time, beating the lower bound for the exact problem. Moreover, using the augmented data structure in a rather obvious way, we can approximate the number of inversions in a permutation of $\{1, 2, \ldots, n\}$ in time $O(n \log\log n)$. This does not go below any known lower bound, but it beats the best known upper bound for the exact problem, which is $O(n \log n/\log\log n)$.

## 2 Main theorem and overview of proof

The main result of the paper is:

**Theorem 1** *For any positive values of $\epsilon$ and $n$, there is a data structure for $\epsilon$-approximate indexed lists that satisfies the consistency and monotonicity conditions, and which supports queries in constant worst-case time and updates in*

$$O\left(\min\left\{\frac{1}{\epsilon^2}, \frac{\log^2(\epsilon n+1)}{\epsilon}, n\right\}\right)$$

*amortized time.*

As the complexity of our algorithm depends on two parameters, $n$ and $\epsilon$, som care must be taken when expressing the time complexity. The reason that we include $\epsilon$ in the bound is to allow for a "varying constant" by letting the value of $\epsilon$ depend on $n$.

In the following sections, we give a constructive proof of this theorem by developing a suitable data structure. The data structure is arranged in layers, where the number of layers might change dynamically depending on the relation between $n$ and $\epsilon$. The maximum number of layers is three.

To give an intuitive explanation of the data structure, assume for now that there are three layers. The list is implemented as a list of sublists, each of which is also a list of (even smaller) sublists. The size of a sublist is logarithmic in the size of the list 'above.' The maintenance cost decreases exponentially with the number of layers, while the query cost is only linear in the number of layers. The sublists in the third layer are short enough to be implemented very efficiently by means of a global lookup table.

The small variation in size of the sublists is the key property which makes it possible to estimate how many list elements precede a certain sublist. We get a good estimation by counting the number of preceding sublists and multiplying by the maximum sublist size. Combining this estimation with the approximate position within the sublist, we can approximate the position of a specified element.

## 3 Overview of the technical construction

To avoid confusion, throughout the paper, we will use *size* for the number of elements in a list, and *length* for the number of entries in an array. As a matter of fact, as will be seen below, a list of size $n$ will be sometime be implemented by means of an array of length greater than $n$.

Although we allow $\epsilon$ to vary with $n$, we will in the following proofs assume that $\epsilon$ does not change. When the value of $\epsilon$ is to be changed, we will need to reconstruct the entire data structure. If we change $\epsilon$ every $n$th update, the amortized cost for this rebuilding is $O(1)$ per update.

Our solution will divide the list into $\epsilon$-approximate sublists, which themselves are subdivided, etc. We introduce some convenient notation that simplifies the technical developments:

We say that an $\epsilon$-approximate indexed list is *normal* if and only if it has the following features:

1. The consistency and monotonicity conditions hold;
2. Queries take constant worst-case time;
3. Given a sequence of $n$ elements, we can construct an $\epsilon$-approximate indexed list (that obeys the same order) in $O(n)$ time;
4. A list of size $n$ can be emptied and deallocated in $O(n)$ time;
5. If there are more than one list present; given a pointer to a list element, we can compute in $O(1)$ time which list it belongs to.

Our solution will indeed have all the above features, and when nothing else is stated it is therefore assumed that all lists are normal.

The last two features are added because they allow us to merge and split sublists in time linear in the number of elements involved.

The min-expression in Theorem 1 consists of three parts. We start by proving the last, and least intersting, one.

**Lemma 2** *An 0-approximate list of size $n$ can be maintained in $O(n)$ space at $O(n)$ worst-case cost per update.*

**Proof.** We just store the elements in an array of length exactly $n$, in which each entry holds the list element as well as the index of the entry. The exact position of an element can then be computed in constant time by returning the index of the array entry pointed at. *Rep*$(i)$ simply returns the element in entry $i$ of the array. The array is rebuilt completely after every update. □

Next, we prove the second part of Theorem 1.

**Lemma 3** *An $\epsilon$-approximate list of size $n$ can be maintained in $O(n)$ space at $O\left(\frac{\log^2(\epsilon n+1)}{\epsilon}\right)$ amortized cost per update.*

**Sketch of proof.** This proof is rather technical; the detailed proof is given in Section 4.

We store our elements in an array of length roughly $(1+\epsilon)n$ where the empty positions are spread evenly. The array is organized as an implicit tree which is kept well-balanced by a pratial rebuilding technique, similar to the maintenance of general balanced trees [1]. □

In the proofs of Lemmas 2 and 3, the employed data structure is just a simple array. In order to prove the first, and most interesting, part of Theorem 1, we use a more elaborate construction, described in the proofs of the following lemmas.

**Lemma 4** *The amortized update cost in an $\epsilon$-approximate list of size $n$ equals $O\left(1/\epsilon^2\right)$ plus the cost of making an update in an $\epsilon/5$-approximate list of size $\Theta(\log^2 n)$.*

**Sketch of proof.** The complete proof is given in Section 5. We use a two-level data structure, where the upper level is a list of size roughly $n/\log^2 n$. each element in the upper level list refers to a lower level list of size roughly $\log^2 n$. The upper level is implemented using Lemma 3. □

**Lemma 5** *An $\epsilon$-approximate list of size $n$ can be maintained at $O\left(1/\epsilon^2\right)$ amortized cost plus the cost of making an update in an $\epsilon/25$-approximate list of size $\Theta\left((\log\log n)^2\right)$.*

**Proof.** We have from Lemma 4 that an $\epsilon/5$-approximate list of size $\Theta(\log^2 n)$ can be maintained at $O\left(1/\epsilon^2\right)$ amortized cost plus the cost of making an update in an $\epsilon/25$-approximate list of size $\Theta\left((\log\log n)^2\right)$. Using this complexity, applying Lemma 4 once more to construct an $\epsilon$-approximate list of size $n$, finishes the proof. □

**Lemma 6** *A set of 0-approximate lists, each being of size $\Theta((\log\log n)^2)$ while the total size of all lists is $n$, can be maintained in $O(n)$ space with $O(1)$ amortized cost per update.*

**Sketch of proof.** The complete proof is given in Section 6. The basic idea is to use a global set of precomputed data structures. Each list is stored in an unsorted array. Since a list is very small, the total number of configurations of such an array (i.e. the number of elements in the list and their relative order) is also rather small. By means of a global *configuration graph* we keep track of all possible configurations. When an insertion or deletion occurs in a list, we change status by following an edge in the configuration graph. In this way, using precomputed solutions, updates and queries can be performed efficiently. □

In the following proofs we will, w.l.o.g., assume that $\epsilon < 1$.

## 4 Proof of Lemma 3

For simplicity, put $\xi = \epsilon/4$. Since we assume $\epsilon < 1$, we assume $\xi < 1/4$.

The $n$ list elements are stored in an array that is rebuilt periodically. The length of the array is $n_0(1+\xi/2)$, where $n_0$ denotes the number of elements at the last rebuilding. Each array entry holds its index, a pointer to a header of the array, and (possibly) a list element. The pointer guarantees that given an element, we can find the header of its list in $O(1)$ time. (When we maintain several lists this turns out to be crucial in order to identify which list a particular element belongs to.) Rebuilding of the array occurs after every $\xi n_0/5$ updates, and its length is adjusted according to the new value of $n_0$. Hence,

$$n_0(1-\xi/5) \leq n \leq n_0(1+\xi/5).$$

There is an implicit hierarchical partition of the array: On the zeroth level of the partition, the array consists of one segment, on the first level it consists of two segments of equal length, and in general, on the $i$:th level it is divided into $2^i$ segments of equal length. If the length of the array is not a power of 2, the segments share entries. For example, in an array of length 13, each segment on level 2 contains 13/4 entries. This means that the leftmost segment contains the three first entries and 1/4:th of the fourth entry, the next segment contains the rest of the fourth entry, entries 5 and 6, and half of the seventh entry, etc.

The subdivision stops at level $\log(\xi n_0)$. (Without loss of generality, $\log(\xi n_0)$ can be assumed to be a positive integer; this can be achieved by decreasing $\xi$ by a factor less than 2.) Hence, each segment on the lowest level contains

$$\frac{n_0(1+\xi/2)}{2^{\log(\xi n_0)}} = \frac{1+\xi/2}{\xi}$$

entries. Since we assume that $\xi < 1/4$, the above is greater than 3. This ensures in particular that each segment on the lowest level contains at least one whole entry.

The main idea of the hierarchical partition is to spread the empty positions evenly in the array. In order to to this, we maintain the following:

**Invariant** *Each segment on the lowest level contains at most one empty entry, which, if it exists, is located in the middle of the segment.*

A segment above the lowest level is said to satisfy the invariant if all lowest-level segments below it do. Initially, and after each global rebuilding, the elements are evenly distributed in such a way that every second segment on the lowest level is full. Due to the chosen length of the array, initially $\xi n_0/2$ array entries are empty, and so $\xi n_0/2$ lowest-level segments are full and $\xi n_0/2$ are non-full. This ensures that it is conceivable to maintain the invariant during $\xi n_0/5$ updates.

### 4.1 Updates

When making an insertion into a segment containing an empty entry, the appropriate elements in the segment are shifted one step to make room for the new element. Similarly, after deleting an element from a full segment, some of the remaining elements are shifted so that the empty entry is located in the middle of the segment. Hence, such updates take time linear in the length of a segment, that is, $O(1/\xi)$.

When attempting to insert an element into a full segment, or delete an element from a segment in which there already is an empty entry, some redistribution of elements is needed before the update can be accomplished, or the invariant will be violated. In either case, we traverse the implicit hierarchy by visiting the segments containing the update entry bottom-up, starting at the lowest level, $\log(\xi n_0)$. At each level we compare the sizes (that is, the number of elements) of the two sub-segments (segments on the next lower level). The size of a segment is computed by simply scanning it. The traversal stops when reaching a segment $S$ where the difference in size between its two sub-segments is at least $|S|\xi/(5\log(\xi n_0))$, where $|S|$ denotes the size of $S$. The elements in $S$ plus the one which was to be inserted, or minus the one which was to be deleted are then redistributed evenly in $S$: The difference in size between $S$'s two sub-segments becomes at most one, the same applies to the sub-segments' sub-segments, and so on.

To prove the correctness of this algorithm we need to show that (1) it restores the invariant in $S$; and (2) there always exists a segment $S$ that satisfies the termination condition.

The first property is easily verified. Consider first an insertion. At the found segment $S$ we know that the difference in size between its sub-segments, the inserted element uncounted, is at least $|S|\xi/(5\log(\xi n_0))$. Since this is a positive number, and as the difference in size between two segments on the same level must be an integer, it is at least one. Therefore, the smaller of the two must contain at least one empty entry. Hence, by shifting the elements of $S$ appropriately the new element can be accommodated, and the invariant gets restored in $S$ and all its descendants. The argument for deletion is very similar.

The second property is shown by contradiction. Consider first the case when an insertion would result in an overfull lowest-level segment. Suppose that during the bottom-up traversal no segment is found that satisfies the termination condition. Let $S$ be any of those segments and denote by $S_\ell$ and $S_s$ the larger and smaller of $S$'s sub-segments, respectively. Then,

$$\frac{|S|\xi}{5\log(\xi n_0)} > |S_\ell| - |S_s| = |S_\ell| - (|S| - |S_\ell|) = 2|S_\ell| - |S|.$$

Consequently,

$$\frac{|S|}{|S_\ell|} > \frac{2}{1 + \frac{\xi}{5\log(\xi n_0)}}.$$

Hence, when ascending from one level to the next, the number of elements increases by more than a factor of $2/(1 + \frac{\xi}{5\log(\xi n_0)})$. Since the segment in which the update was to be performed was full, it contained $(1 + \xi/2)/\xi$ elements. Hence, the total number of elements in the array will exceed

$$\begin{aligned}
\frac{1+\xi/2}{\xi}\left(\frac{2}{1+\frac{\xi}{5\log(\xi n_0)}}\right)^{\log(\xi n_0)} &= \frac{1+\xi/2}{\xi}\frac{\xi n_0}{\left(1+\frac{\xi}{5\log(\xi n_0)}\right)^{\log(\xi n_0)}} \\
&> n_0\frac{1+\xi/2}{e^{\xi/5}} \\
&> n_0\frac{1+\xi/2}{1+\xi/4} \\
&> n_0(1+\xi/5),
\end{aligned}$$

where the last two inequalities hold for any $\xi < 1$. This contradicts that the number of elements in the array is at most $n_0(1+\xi/5)$, and we are done.

Similarly, assume that a segment would underflow due to a deletion, and suppose that no segment on the bottom-up traversal satisfies the termination condition. Pick $S$, $S_\ell$, and $S_r$ as above. Then

$$\frac{|S|\xi}{5\log(\xi n_0)} > |S_\ell| - |S_s| = (|S| - |S_s|) - |S_s| = |S| - 2|S_s|,$$

and so

$$\frac{|S|}{|S_s|} < \frac{2}{1 - \frac{\xi}{5\log(\xi n_0)}}.$$

Hence, when ascending from one level to the next, the number of elements increases by less than a factor of $2/(1 - \frac{\xi}{5\log(\xi n_0)})$. The underflowing segment contains $(1+\xi/2)/\xi - 1$ elements; and, consequently, the total number of elements is less than

$$\left(\frac{1+\xi/2}{\xi} - 1\right)\left(\frac{2}{1 - \frac{\xi}{5\log(\xi n_0)}}\right)^{\log(\xi n_0)} \quad = \quad n_0 \frac{1-\xi/2}{\left(1 - \frac{\xi}{5\log(\xi n_0)}\right)^{\log(\xi n_0)}}.$$

The denominator of the right side grows monotonically with $\log(\xi n_0)$, which is a positive integer. Hence, the entire expression attains its maximum for $\log(\xi n_0) = 1$. Thus, the total number of elements is less than

$$n_0 \frac{1-\xi/2}{1-\xi/5} < n_0(1-\xi/5).$$

This contradicts that the total number of elements is at least $n_0(1-\xi/5)$.

Having proven correctness of the updating algorithms, we now turn to their time requirements. Updates that invoke global rebuilding have an amortized cost of $O(1/\xi)$. Updates that do not require any redistribution have a worst-case cost of $O(1/\xi)$ (shifting within the segment). Consider an update that invokes a redistribution in a segment $S$, with sub-segments $S_1$ and $S_2$. We note four facts.

- Immediately before the redistribution, $||S_1| - |S_2|| \geq |S|\xi/(5\log(\xi n_0))$. (Recall that the latter is a positive integer.)
- Immediately after the previous redistribution involving $S$, $||S_1| - |S_2|| \leq 1$.
- At least one update has been made in $S$ since the previous redistribution; namely, the one that invoked the current redistribution.
- Each update increases $||S_1| - |S_2||$ by at most one.

It follows that the number of updates that have been made in $S$ since the previous redistribution is at least

$$\max\left\{\frac{|S|\xi}{5\log(\xi n_0)} - 1, 1\right\} = \Omega\left(\frac{|S|\xi}{\log(\xi n)}\right).$$

The cost of the update, including counting and redistribution, is $\Theta(|S|)$. Hence, we may use a traditional argument in amortized analysis: Paying a deposit of $\Theta(\log(\xi n)/\xi)$ at each segment affected by an update is enough to cover the cost. The number of such segments is $\Theta(\log(\xi n))$, giving a total amortized cost of $O(\log^2(\xi n)/\xi)$.

### 4.2 Queries

Consider a query *Approx-Pos*$(x)$, where $x$ resides in the $i$:th entry of the array. We claim that

$$Pos(x) \leq i \leq (1+4\xi)Pos(x) \tag{2}$$

The lower bound is obvious since there are at most $i-1$ elements preceding $x$ in the array. To verify the upper bound, we distinguish two cases. If $i < \lceil (1+\xi/2)/2\xi \rceil$, then $x$ is located in the first half of the first segment, and so all entries preceding $x$ are filled, and $i = Pos(x)$. Thus, Inequality (2) hold trivially in this case.

Consider now the second case. We derive a lower bound on $Pos(x)$ in terms of $i$. As each lowest-level segment contains at most one empty entry, the number of empty entries preceding $x$ is at most $\lceil (i\xi)/(1+\xi/2) \rceil$. Hence,

$$Pos(x) \geq i - \left\lceil \frac{i\xi}{1+\xi/2} \right\rceil \tag{3}$$

Using the fact that $i \geq \lceil (1+\xi/2)/2\xi \rceil \geq 1/2\xi$, we have

$$\left\lceil \frac{i\xi}{1+\xi/2} \right\rceil \leq \frac{i\xi}{1+\xi/2} + 1 \leq \frac{i\xi}{1+\xi/2} + 2i\xi \leq \frac{4i\xi}{1+4\xi},$$

where the last inequality holds for sufficiently small $\xi$. Using this upper bound in Inequality (3) leads to the required lower bound on $Pos(x)$, and Inequality (2) follows. Returning $i$ as an approximate position of $x$ is thus sufficient.

Similarly, *Approx-Rep*$(i)$ is answered by simply returning the element $x$ in array entry $i$, or, if that entry is empty, the element in the next entry. Repeating the above calculations shows that if $i < \lceil (1+\xi/2)/2\xi \rceil$, then $Pos(x) = i$; and if $i \geq \lceil (1+\xi/2)/2\xi \rceil$, then

$$Pos(x) \leq i \leq (1+4\xi)Pos(x),$$

and so element $x$ satisfies the conditions.

As an element $x$ resides in array entry *Approx-Pos*$(x)$, it is obvious that the consistency and monotonicity conditions hold. The other features of a normal approximate indexed list are easily verified.

The proof of Lemma 3 is now complete.

## 5 Proof of Lemma 4

In this section we show how to decrease the update time by using a list of sublists. The advantage of this scheme is that only a few updates will escalate from the sublists to the top list, while most updates take place in the sublists which are shorter and support faster operations. In some sense, we reduce the problem from that of maintaining lists of size $n$ to that of maintaining sublists of size $\Theta(\log^2 n)$. The cost of the reduction is $O(1/\epsilon^2)$. The drawback is that we lose a little bit of accuracy in the approximation; only a small factor, though.

First, we note that if $\epsilon < C/\log^2 n)$, for some positive constant $C$, Lemma 3 gives a cost of $O(1/\epsilon^2)$ without using any sublists at all. Hence, in the following we can assume that $\epsilon \geq 40/\log^2 n$. We put $\xi = \epsilon/5$. Since we assume $40/\log^2 n \leq \epsilon < 1$, we assume $8/\log^2 n \leq \xi < 1/6$.

We divide the list into sublists. The list of sublists, referred to as the 'top' list, is stored in an $\xi$-approximate indexed list of Lemma 3, each array entry being empty or containing a bidirectional pointer to a header of a sublist.

The sublists are of size close to $m = \lceil \log^2 n \rceil$. Initially, and after each rebuilding of the entire data structure, each sublist contains $m$ or $m+1$ elements, and, by increasing or decreasing the number of sublists dynamically, the size of any sublist is guaranteed to stay between $L = \lceil m(1-\xi) \rceil$ and $U = \lfloor m(1+\xi) \rfloor$. We make no assumptions about the update time in the sublists—only that they are normal $\xi$-approximate indexed lists.

When an update would cause a sublist to underflow or overflow we restore the condition by merging and splitting sublists, as follows. We examine the violating sublist's $2/\xi - 1$ nearest neighbors. The total number of elements in these sublists, including the violating sublist, is at least $2L/\xi$ and at most $2U/\xi$. If this number is at most $2m/\xi$, we merge the sublists and create $2/\xi - 1$ new sublists and distribute the elements evenly among those. On the other hand, if the number of elements exceeds $2m/\xi$, we distribute them evenly among $2/\xi + 1$ sublists. Even though a merge followed by a split destroys and creates many sublists, the *number* of sublists only changes by one. By reusing the pointers in the top list the merging and splitting gives rise to exactly one update in the top list.

After the redistribution, the created sublists will be of suitable size. Examining the occurring cases, the minimum size of each new sublist is found to be at least

$$\begin{aligned}
\min\left\{\left\lfloor \frac{2m/\xi+1}{2/\xi+1}\right\rfloor, \left\lfloor \frac{2L/\xi}{2/\xi-1}\right\rfloor\right\} &\geq \min\left\{\frac{2m}{2+\xi}, \frac{2L}{2-\xi}\right\} - 1 \\
&\geq \min\left\{\frac{2m}{2+\xi}, \frac{2L}{2-\xi}\right\} - \frac{\xi m}{6} \\
&\geq m\left(1 - \frac{5\xi}{6}\right),
\end{aligned}$$

where the second inequality holds because $\epsilon \geq 40/\log^2 n$, which implies $\xi \geq 6/m$, and the last inequality holds for any $\xi \leq 1/2$. Similarly, the maximum size is at most

$$\begin{aligned}
\max\left\{\left\lceil \frac{2U/\xi}{2/\xi+1}\right\rceil, \left\lceil \frac{2m/\xi}{2/\xi-1}\right\rceil\right\} &\leq \max\left\{\frac{2U}{2+\xi}, \frac{2m}{2-\xi}\right\} + 1 \\
&\leq \max\left\{\frac{2U}{2+\xi}, \frac{2m}{2-\xi}\right\} + \frac{\xi m}{6} \\
&\leq m\left(1 + \frac{5\xi}{6}\right),
\end{aligned}$$

where, again, the second inequality holds because $\xi \geq 6/m$; and the last inequality holds for any $\xi \leq 1/2$.

It is important to observe that, after a split/merge has taken place, the number of updates required to make any of the involved sublists overflow or underflow is at least $\xi m/6 = \Omega(\xi m)$.

(Recall that in order for this to happen, the size of a sublist must have shrunk/grown to $L-1$ or $U+1$.) Moreover, the same applies initially, because $m \geq m(1-5\xi/6)$ and $m+1 \leq m(1+5\xi/6)$.

Consider now the cost of an update in one of the sublists. Ignoring the cost of inserting in the sublist itself, there are two terms contributing to this quantity. First, consider the cost of merging and splitting to control the sizes of the sublists. The actual cost of a merge is linear in the number of elements involved, that is, $\Theta(m/\xi)$. However, the number of updates required for any of the newly created sublists to violate the size condition is $\Omega(m\xi)$. Consequently, the amortized cost of a merge is $O\left(1/\xi^2\right)$ per update. Second, each merge/split generates one update in the top list, which takes $O(\log^2(\xi n/m)/\xi)$ time, by Lemma 3. Since at most every $\Theta(m\xi)$:th update causes such an update, the amortized cost per update in this list is $O\left(\log^2(\xi n/m)/(\xi^2 m)\right)$. Summing the above gives that the total amortized cost per update, ignoring the cost of updates in the sublists, is

$$O\left(\frac{1}{\xi^2}+\frac{\log^2(\xi n/m)}{\xi^2 m}\right)=O\left(\frac{1}{\xi^2}\right)$$

by the choice of $m$.

Answers to queries are computed in constant time in a straightforward way. Consider a query *Approx-Pos*$(x)$. First, apply *Approx-Pos* to compute an approximate position $P$ of $x$'s sublist in the top list. (This involves first finding the header of $x$'s sublist in constant time.) Second, compute an approximate position $p$ of $x$ within its sublist. We answer the query by $i=(P-1)U+p$. To verify that this is a good enough approximation, we need to show

$$Pos(x) \leq i \leq (1+5\xi)Pos(x). \tag{4}$$

Denote by $R$ the exact position of $x$'s sublist in the top list. Similarly, denote by $r$ the exact position of $x$ within its sublist. By definition of $U$, $Pos(x) \leq (R-1)U+r$. Since both the top lists and the sublists are $\xi$-approximate,

$$\begin{array}{rcccl} R & \leq & P & \leq & (1+\xi)R, \\ r & \leq & p & \leq & (1+\xi)r. \end{array}$$

Using the left inequalities, it follows that

$$i=(P-1)U+p \geq (R-1)U+r,$$

and so $i \geq Pos(x)$, verifying the left part of Inequality (4).

To show the right part of the inequality, we distinguish two cases. First, if $R=1$, then for sufficiently small $\xi$, $P=1$. This implies that $Pos(x)=r$ and $i=p$ and so, in this case $i \leq (1+\xi)Pos(x)$.

Second, if $R>1$, then using the upper bounds on $P$ and $p$, elementary calculations yield

$$\begin{aligned} i &\leq [(1+\xi)R-1]U+(1+\xi)r \\ &\leq (1+2\xi)(R-1)U+(1+\xi)r \\ &\leq (1+5\xi)[(R-1)L+r] \\ &\leq (1+5\xi)Pos(x). \end{aligned}$$

To answer *Approx-Rep*$(i)$, we essentially perform *Approx-Pos* backwards. Compute $P = \lfloor (i-1)/U + 1 \rfloor$ and perform *Approx-Rep* $(P)$ in the top list, which returns the sublist that we are interested in. Then, compute the remainder $p = i - (P-1)U$ and perform *Approx-Rep* $(p)$ in the found sublist. This returns an element $x$.

In order to prove the desired properties of the returned element $x$, we consider two cases. First, suppose there exists an element $y$ such that *Approx-Pos*$(y) = i$. Then it can easily be verified that indices $P$ and $p$ computed by *Approx-Rep*$(i)$ and *Approx-Pos*$(y)$ will be the same. Hence $x = y$. This shows that, in this case, the element returned by *Approx-Rep* meets the approximation requirements. Moreover, the consistency condition is satisfied.

Second, if *Approx-Pos*$(y) \neq i$ for all elements $y$, then as the element in the next entry gets picked, *Approx-Pos*$(x) > i$. Thus,

$$Pos(x) \leq i \leq \textit{Approx-Pos}(x) \leq (1 + 5\xi) Pos(x),$$

Hence, $x$ meets the requirements in this case as well.

Regarding the space requirements, we note that the list of sublist requires sublinear space while each sublist is assumed to be normal and, hence, use linear space. It is easily seen that the other features of a normal approximate indexed list are satisfied.

The proof of Lemma 4 is now complete.

## 6 Proof of Lemma 6: maintaining short lists

Short sublists can be implemented to support updates in $O(1)$ amortized time, and *exact* answers to queries in $O(1)$ worst-case time:

Recall that the total number of elments stored in all sublists is $n$. The elements are distributed in $\Theta(n/(\log\log n)^2)$ 0-approximate indexed sublists of size $\Theta((\log\log n)^2)$. To prove the present lemma, we need not consider the problem of merging and splitting. We only need to ensure that a sublist can be created and destroyed in linear time (in the number of elements).

Let $U$ be the maximum size of a sublist. Each sublist is implemented in an array of length $U$. Updates are handled in a straightforward manner: At an insertion the new element is placed in the first vacant location; at a deletion the element is simply removed and its location becomes vacant. Let a *configuration* of a sublist be a sequence of at most $2U + 1$ numbers; the first number tells the number of elements presently in the sublist, next comes the sequence of array indices that would be visited by an ordered traversal of the sublist. Then comes the index of the first vacant location (if any). Finally comes $U$ numbers, where the $i$:th is the position of the element in the $i$:th array entry, or 0 if the entry is empty.

How many different configurations can there be? This can be determined by computing how many values the numbers can take. First, note that, the last $U$ numbers are redundant, and so do not give rise to any new configurations. Hence, we can disregard those in the calculation. The first number in a configuration can take $O(U)$ different values and the rest of the sequence can take $U!$ different values. In total, the number of possible configurations is $O(U!U)$.

Given the array and its configuration, both *Rep*$(i)$ and *Pos*$(x)$ can be trivially computed in constant time.

An update takes one configuration to another, and storing the appropriate information about configurations we can support operations efficiently. We use a global *configuration graph,* where the nodes correspond to possible configurations and two nodes are connected by a directed edge whenever there exists an update that takes the first configuration to the other. The graph can be constructed as follows:

- Generate all possible configurations in lexicographic order. Each configuration requires $O(U)$ time and space, and the number of configurations is $O(U!U)$. In total this construction takes $O(U!U^2)$ time and space. This is $o(n)$.
- For each configuration, compute the $O(U)$ possible configurations that can occur as a result of one update. Each new configuration can be generated in $O(U)$ time, and, using binary search, its corresponding node can be found in the sorted set of all configurations in $O(U \log(U!U)) = O(U^2 \log U)$ time. The outgoing edges from a node are stored as pointers in an array of length $U$. The $i$:th entry holds a pointer corresponding to an update in the $i$:th entry of the array in which the sublist is stored. The total number of edges to generate is $O(U!U^2)$, and so the total cost is $O(U!U^4 \log U) = o(n)$.

Each sublist keeps a pointer into the configuration graph. At an update we follow the appropriate edge in the configuration graph in constant time.

The configuration graph uses $o(n)$ space while each short list use linear space in ist size. Hence, the total space taken by these lists is $O(n)$. All other properties of normal approximate indexed lists are easily verified.

The proof of Lemma 6 is now complete.

## 7 Applications

### 7.1 List labeling

The monotonicity condition gives immediately

**Corollary 7** *For any non-increasing positive function $\epsilon$ on $n$, there is a data structure for list labeling which maintains label space $\{1, 2, \ldots, (1+\epsilon)n\}$, and which supports retrieval of a label in constant worst-case time, and updates in*

$$O\left(\min\left\{\frac{1}{\epsilon^2}, \frac{\log^2(\epsilon n)}{\epsilon}, n\right\}\right)$$

*amortized time.*

As mentioned in the introduction, this improves significantly upon the tradeoff between label space and update time.

From the proof of Lemma 3 it is easy to deduce:

**Corollary 8** *For any non-increasing positive function $\epsilon$ on $n$, there is a data structure for list labeling with explicit labels which maintains label space $\{1, 2, \ldots, (1+\epsilon)n\}$, and which supports retrieval of a label in constant worst-case time, and updates in*

$$O\left(\frac{\log^2(\epsilon n)}{\epsilon}\right)$$

*amortized time.*

This improves upon the results by Itai, Konheim, and Rodeh [11] in two respects: the label space is reduced from $2n$ to $(1+\epsilon)n$; moreover, a bound on the time required by deletions is provided. We would like to point out that the data structure of Lemma 3 is similar to that of Itai, Konheim, and Rodeh, in that both make use a sparse array representation. The main reason why we obtain better bounds is due to the sharing of array entries of consecutive segments.

### 7.2 Approximate subset rank and inversions

Recall the definition of the subset rank problem given in the introduction. To support insertions, deletions, and approximate rank queries on a subset $S$ of $\{1, 2, \ldots, n\}$, we will augment our data structure for indexed lists with a van Emde Boas data structure (VEB) [9]. Among other operations, a VEB supports insertions, deletions, and searches (i.e., return the predecessor of an element, which itself may or may not be present in the data structure) in $O(\log\log n)$ time. The elements of the set $S$ are represented both in a VEB and in an $\epsilon$-approximate indexed list, and each VEB element points to its corresponding indexed list element. The operations are implemented as follows.

*Insert*$(i)$**:** Find $i$'s predecessor $j$ in the VEB. Follow the pointer to the indexed list and insert $i$ after $j$. Also, insert $i$ in the VEB and set the pointer between the two $i$'s.

*Delete*$(i)$**:** Find $i$ in the VEB. Follow the pointer to the indexed list and delete $i$. Also, delete $i$ from the VEB.

*Approx-Rank*$(i)$**:** Find $i$ in the VEB. Follow the pointer to the indexed list and return *Approx-Pos*$(i)-1$.

By rebuilding our data structure periodically, we can allow ourselves to vary the value of $\epsilon$ such that $0.5/\sqrt{\log\log n} < \epsilon < 1/\sqrt{\log\log n}$ is always satisfied. Then, all operations run in $O(\log\log n)$ amortized time, by Theorem 1. Moreover, the approximation of the rank is very good:

$$\mathit{Rank}(i) \leq \mathit{Approx\text{-}Rank}(i) \leq \left(1 + \frac{1}{\sqrt{\log\log n}}\right) \mathit{Rank}(i).$$

This should be contrasted to the result of Fredman and Saks [10], which states that in order to compute the exact ranks, at least one of the three operations requires $\Omega(\log n/\log\log n)$ amortized time.

We conclude by pointing out a straightforward application of the above result. Recall that the number of inversions in a permutation $\pi$ of $\{1, 2, \ldots, n\}$ is defined as [12]

$$\mathit{Inv}(\pi) = |\{(k, l) \mid k < l \text{ and } \pi(k) > \pi(l)\}|.$$

$\mathit{Inv}(\pi)$ can be computed by inserting the elements $\pi(n), \pi(n-1), \ldots, \pi(1)$ into an initially set $S$, and accumulate the ranks of the most recently inserted element.

The best known upper bound for computing $\mathit{Inv}(\pi)$ is $O(n\log n/\log\log n)$, which follows by combining the outlined algorithm with a data structure of Dietz [5]. Whether *Inv* can be computed faster, maybe even in linear time, is an open problem.

Diaconis and Graham [3] showed that

$$Inv(\pi) \leq D(\pi) \leq 2Inv(\pi),$$

where $D(\pi) = \sum_{i=1}^{n} |\pi(i) - i|$. Hence, since $D$ is trivially computable in linear time, so is a fairly good approximation of $Inv$.

Again, varying $\epsilon$ such that $0.5/\sqrt{\log\log n} < \epsilon < 1/\sqrt{\log\log n}$ and using the above combined data structure, it follows immediately that in $O(n \log\log n)$ time we can compute a number $I$, such that

$$Inv(\pi) \leq I \leq \left(1 + \frac{1}{\sqrt{\log\log n}}\right) Inv(\pi).$$

This improves significantly on the accuracy at the price of slightly longer running time.

## Acknowledgements

We would like to thank Joseph Wang for spotting a weakness in a preliminary version of this article. This work was partially supported by TFR, the Swedish Research Council for Engineering Sciences. The work by the second author was carried out while visiting Zvi Galil at Columbia University.

The anonnymous referees made several comments which improved the presentation of this article.

## References

[1] A. Andersson. Improving partial rebuilding by using simple balance criteria, In *Proc. Workshop on Algorithms and Data Structures*, pages 393–402. LNCS 382, Springer Verlag, 1989.

[2] T. Biedl and G. Kant. A better heuristic for orthogonal graph drawing. In *Proc. 2nd European Symp. on Algorithms*, pages 24–35. LNCS 855, Springer-Verlag, 1994.

[3] P. Diaconis and R.L. Graham. Spearman's footrule as a measure of disarray. *J. Royal Statistical Society Series B*, 39:262–268, 1977.

[4] P.F. Dietz. Maintaining order in a linked list. In *Proc. 14th ACM Symp. on the Theory of Computing*, pages 122–127, 1982.

[5] P.F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proc. 1989 Workshop on Algorithms and Data Structures*, pages 39–46. LNCS 382, Springer-Verlag, 1989.

[6] P.F. Dietz, J.I. Seiferas, and J. Zhang. A tight lower bound for on-line monotonic list labeling. In *Proc. 4th Scandinavian Workshop on Algorithm Theory*, pages 131–142. LNCS 824, Springer-Verlag, 1994.

[7] P.F. Dietz and D.D. Sleator. Two algorithms for maintaining order in a list. *J. Comput. System Sci.* To appear. Prel. version in *Proc. 19th ACM Symp. on the Theory of Computing,* pages 365–372, 1987.

[8] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. *J. Comput. System Sci.*, 38:86–124, 1989.

[9] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.

[10] M.L. Fredman and M.E. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st ACM Symp. on the Theory of Computing*, pages 345–354, 1989.

[11] A. Itai, A.G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Proc. 8th Internat. Colloquium on Automata, Languages and Programming*, pages 417–431. LNCS 115, Springer-Verlag, 1981.

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

[13] Y. Matias, J. S. Vitter, and N. E. Young. Approximate data structures with applications. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 187–194, 1994.

[14] A.K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21:101–112, 1984.