# Efficient Concatenable Ordered Lists

William Pugh[†]

TR 87-831
April 1987

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

[†] AT&T Bell Laboratories Scholar

# Efficient Concatenable Ordered Lists

William Pugh[†]
Dept. Computer Science
Cornell University

**Abstract**

A new approach for providing an efficient implementation of concatenable ordered lists is discussed. The structures described have an equivalence to search trees. In balanced search trees the tree is continually modified to maintain certain balance properties; with our structure the tree is guaranteed to be structured randomly and with very high probability is relatively balanced. We thus avoid the overhead associated with explicitly maintaining the balance. Because of this property, the structures described are referred to as *guaranteed-random trees.*

## Introduction

We give a representation for concatenable ordered lists and efficient algorithms for operations that determine the relative order of two elements, insert and delete elements, and append and split lists. The algorithms described require $O(L(n))$ worst-case time to perform a comparison, $O(n^{1/L(n)})$ expected time to perform an insertion or deletion, and $O(L(n)\, n^{1/L(n)})$ expected time to append or split lists, where $L(n)$ is any function of n picked by the user, and can be a constant function. Notice that if $L(n) = (\ln n)/c$, then $n^{1/L(n)} = e^{c}$. The algorithms are of unique theoretical interest both because the order of the algorithms can be tuned as desired, and because of the use they make of randomness. The algorithms are also of practical interest: they are simple to implement, involve low constant factors, and require linear space; an implementation is given in the appendix. Several extensions are suggested that can reduce the time required to insert and delete elements, depending on the application.

The best previous solution for this problem is by Huddleston and Mehlhorn[4] using weak $B$ -trees which gives $O(1)$ time for insertion and deletion, and $O(\log n)$

[†] AT&T Bell Laboratories Scholar

time for all other operations.

Our algorithms are probabilistic, and hence we analyse the probability distribution of the expected running times instead of the worst case time. Very briefly, the following results are obtained:

- For any sequence of operations, the actual running time averaged over many executions is equal to the expected time.
- If L(n) ≥4, the probability that the actual running time of an *append* or *split* operation will more than 4 times the expected time is less than one in a thousand.

The technique presented has roughly the same computational power as balanced search trees, although it uses a totally different approach. We discuss how the approach described here could be used for any problem that uses balanced search trees, and the advantages of our methods.

## Objectives and related work

We wish to be able to perform efficiently the following operations on a collection of lists and their elements:

Insert($x$,$y$) - Insert $x$ immediately to the left of $y$.

Delete($x$) - Delete $x$ from the list containing it.

ContainingList($x$) - Return the list containing $x$.

Compare($x$,$y$) - Determine if $x$ and $y$ are in the same list, and if they are, determine if $x$ occurs to the left of $y$.

Split($x$) - Split the list containing $x$ into two lists, one containing all the elements to the left of $x$, and the other containing $x$ and all elements to the right of it, and return the two resulting lists.

Append(*L1* ,*L2* ) - Append *L1* to *L2* (joining the right end of *L1* to the left end of *L2* and return the resulting list).

We assume elements are not intrinsically ordered, and are ordered only by

their position in a list. Operations are done on-line; each operation must complete before the next operation is known. There are numerous motivating examples for an efficient implementation of concatenable ordered lists. Our original motivation is that as discussed in [2], one can use an ordered list to determine ancestor information in trees, with each ancestor check requiring two comparisons, and cutting or grafting trees requiring the splitting and appending of lists. For our application, we needed to perform many more ancestor checks than graft and cut operations, and so were willing to have the graft and cut operation be slightly slower in order to speed up the ancestor check operation.

Tsakalidis[7] explored a similar problem, but did not examine the operations of splitting or appending lists, or of handling multiple lists. He gives algorithms using indexed $BB[\alpha]$-trees that require $O(1)$ time for comparisons and amortized $O(1)$ time for insert and delete. However, a fundamental restriction of his algorithm would seem to require $O(n/\log n)$ time to perform a split or append operation. In [3], Dietz and Sleator report on methods that are simpler than those in [7] for maintaining an ordered list, but once again do not consider the problem for splitting or appending lists.

Sleator and Tarjan[6] present a method using self-adjusting binary trees that takes $O(\log n)$ amortized time for all operations.

As mentioned before, Huddleston and Mehlhorn[4] present a method using weak $B$ -trees which gives $O(1)$ time for insertion and deletion, and $O(\log n)$ time for all other operations.

In [1], concatenable queues are discussed that are similar to concatenable ordered lists except that they assume that elements are keyed, and allow the queue to be searched for a particular element is in the queue. Concatenable queues are implemented using 2-3 trees, and all operations are $O(\log n)$ time.

## Data Structures, Invariants and Algorithms

When an element is created, it is assigned a *level* that is an integer in the range 1 to *L(n)*. Choosing a *level* to assign to an element is discussed in the section on time analysis, but for now the reader should simply be aware that elements with low *level* numbers are more common than elements with higher *level* numbers.

Lists are doubly linked, and are terminated on both ends with special terminator elements that are considered to be *level* $\infty$. We maintain pointers from a list to its left and right terminator, and from each terminator to the list it terminates.

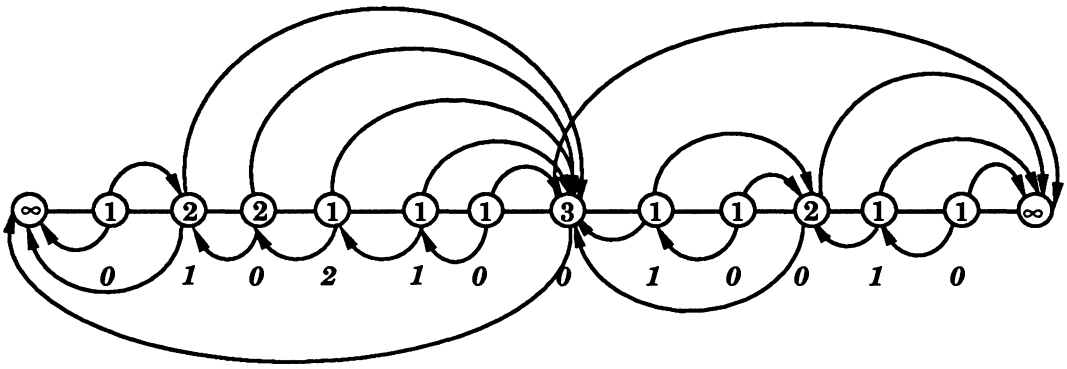
In addition, for each element $x$ other than terminators, we maintain the following information:

*right_parent* - a pointer to the first element to the right of $x$ that has a greater *level* number than $x$.

*left_sibling* - a pointer to the first element to the left of $x$ that has the same or a greater *level* number than $x$.

*right_siblings* - a count of the number of elements between an element and its *right_parent* that have the same *level* number as $x$

For the moment, we avoid assigning any higher-level or intuitive meaning to the terms *parent* and *sibling*; their meaning is exactly as defined above. The figure below shows this information for a sample list. The elements are shown in order, with the *level* of each element inscribed. The *right_parent* pointers are shown above the elements, and the *left_sibling* pointers are shown below them. The values for *right_siblings* are shown below the elements.




**Comparing two elements**

The compare operation, given two elements, determines if they are in the same list, and if they are, determines which is to the left of the other. To compare two elements $p$ and $q$, we travel from each element to the right by following *right_parent* pointers until we come to a terminator element, and save the path traveled. If different terminator elements are reached, $p$ and $q$ are in different lists. Otherwise, let $(p_i, p_{i-1}, \ldots, p_2, p_1, p_0)$ be the path traveled from $p$, where $p_i = p$ and $p_0$ is the terminator element, and $p_{k-1}$ is the *right_parent* of $p_k$. Similarly let $(q_j, q_{j-1}, \ldots, q_2, q_1, q_0)$ be the path traveled from $q$.

Let $k$ be the smallest integer such that $k \le i$, $k \le j$, and $p_k \neq q_k$ (assume $k$ exists). If $p_k$ and $q_k$ have the same *level* number no element with a level greater than their *level* can exist between them. Assume, without loss of generality, that *right_siblings*$(p_k)$ > *right_siblings*$(q_k)$. Then $p_k$ is to the left of $q_k$. Element $p$ must be to the left of $p_k$, or must be $p_k$, and similarly for $q$ and $q_k$. Since the path from $q$ can not pass over $p_k$, $p_k$ must be to the left of $q$. Therefore the elements appear in the order $p \le p_k < q \le q_k$, where $\le$ and $<$ represent order in the list.

If $p_k$ and $q_k$ have different *level* numbers, assume without loss of generality that *level*$(p_k)$ > *level*$(q_k)$. Then $p_k$ must be to the left of $q_k$, and as above, $p \le p_k < q \le q_k$

If no such $k$ exists, then one list is a suffix of the other. If the paths are the

same, $p$ and $q$ are the same element. Assume, without loss of generality, that the path from $q$ is a proper suffix of the path from $p$. Then $p$ is to the left of $q$.

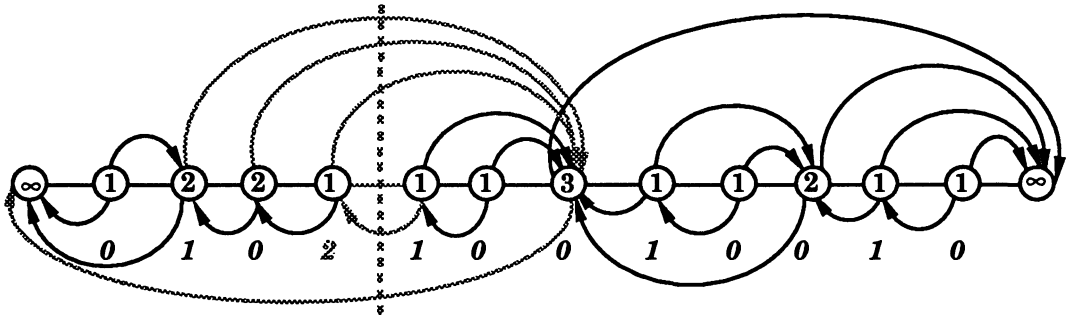
**Determining the list containing an element**

To determine which list an element is contained in, we travel down the *right_parent* chain from the element until we come to a terminator, and return the list associated with the terminator.

**Splitting a list or appending two lists**

To split a list between two elements, we first create two new list terminator elements, and insert them into the doubly linked list. We must then find all pointers that cross over the split and make them point to the appropriate terminator elements and also update the necessary *right_siblings* counters.

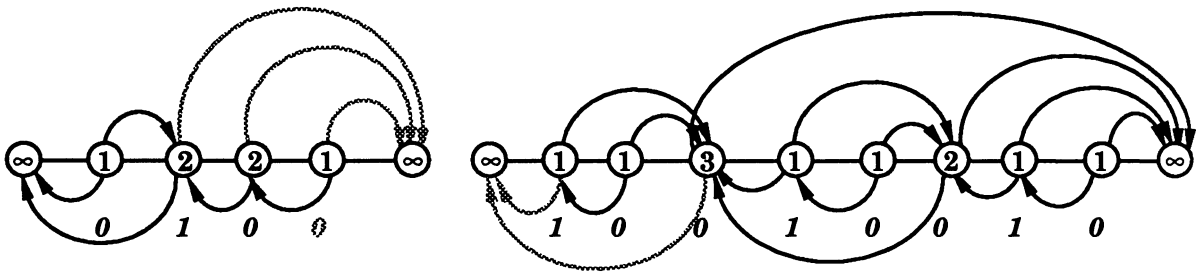
The figure below shows what is involved in performing a sample split. The dashed grey line shows where the list is being split, and the grey arcs and numbers are invalid and need to be updated.




Looking to the left of the split, all the *level* 1 elements we encounter before we find a *level* 2 (or greater) element must have their *right_parent* pointer changed to point to the new terminator element for the left list, and also possibly have their *right_siblings* field updated. Once we have encountered an element of *level* 2, we can ignore the elements with *levels* less than it and update all the *level* 2 elements we find until we encounter a *level* 3 element, and so on. The list of elements that must be updated are exactly those elements that are found by following the *left_sibling* chain from the element immediately to the left of the split. Their

*right_parent* and *left_sibling* pointers must be updated to point to the newly created terminator element, and their *right_sibling* counters must be updated to the correct values.

Looking to the right of the split, the first *level* 1 element must have its *left_sibling* pointer updated. After updating an element of *level i*, we move to the right until we find the first element with a *level* greater than *i*, and update that element. The elements that need to be updated are exactly those elements found by following the *right_parent* pointer from the element immediately to the right of the split. After splitting the above list we have the following data structure:




Appending two lists is the inverse of splitting a list; we need to update all pointers to the list terminators being removed, and update the appropriate right_siblings counters. Consider appending a pair of lists that had just be split at the same point, and you can see that the elements that need to be updated are the same. When we append two lists, we must move outwards in both directions simultaneously as we update pointers; a detailed implementation of both split and append is given in the appendix.

**Inserting or deleting an element**

Inserting an element could be performed by splitting the list and appending a singleton list between them. We can, however, achieve better performance by combining the operations that would be performed, and eliminating the unnecessary ones. In particular, we can perform a single pass outwards from the insertion location, updating pointers as required, and we can stop when we reach an element that has a greater *level* than the element being inserted. For example, when inserting an element of *level* 1, there is no need to update information for any *level*

2 or greater elements unless they are immediately adjacent (the only pointers they might have to a *level* 1 element are their pointers to immediately adjacent elements). Deleting an element is similar to inserting an element, and involves updating the same values.

## Time Analysis

*Compare and ContainingList operations* - The length of the path travelled can not be longer than $L(n)$, since each time we traverse a *right_parent* pointer, we move to an element with a higher *level* number. Therefore, the worst-case time either of these operations is $O(L(n))$.

*Split and Append operations* - The time required to split a list or append two lists is determined by the length of the *left_sibling* chain plus the length of the *right_parent* chain. The maximum length of the *right_parent* chain is $L(n)$. The length of the *left_sibling* chain of an element of *level* 1 is determined by the number of elements of *level* 1 we encounter while moving to the left before we find an element of *level* 2, plus the number of elements of *level* 2 we encounter before we encounter an element of *level* 3, and so on. This is in turn determined by the patterns in which elements with different *levels* appear as we traverse the list, which is in turn determined by the method we use to assign *levels* to elements.

One method of assigning *levels* to elements is to make the *level* of an element dependent on the position of the element in the list containing it. However, any position-dependent scheme for assigning a *level* to elements has the major problem that any modification to a list would require that we update the *levels* of many elements. We shall consider a scheme that never requires us to change the *level* of an element.

To simplify our analysis we assume that for all $i$, the number of elements of *level* $i$ we expect to encounter before encountering an element with a *level* larger than $i$ is the same, and will be represented as $E(n)$. To achieve this value of $E(n)$, we will assign *levels* to elements randomly, with a probability distribution such

that the average length of one level of a *left_sibling* chain is $E(n)$. Using a function *random* that returns a random number in the range [0..1), we calculate this as:

*level* $\leftarrow 1$
**while** *random* $< p$ **and** *level* $< L(n)$ **do** *level* $\leftarrow$ *level*+1;
*(Where p is a value we calculate as a function of E(n) to give us the desired results.)*

In the section on probability analysis, we derive equations for $p$, $E(n)$ and the expected length of a *left_sibling* chain in terms of $n$: $p = 1/(E(n)+1)$, $n = E(n)(E(n)+1)^{L(n)-1}$, or $E(n) \cong n^{1/L(n)}-1$, and that the expected value of a *left_sibling* chain is $O(L(n)(E(n)+1))$. As a result, the expected time required to split a list or append two lists is $O(L(n)n^{1/L(n)})$.

*Insert and Delete operations* - To insert or delete an element of *level* $k$ requires $O(kE(n))$ expected time. The expected time to insert or delete an element with an unknown *level* is the sum over $k$ ranging from 1 to $L(n)$ of the probability that element is level $k$ or greater times the additional work that level $k$ element requires compared to a level $k$-1 element. This is $O(E(n)(1+1/(E(n)+1) + 1/(E(n)+1)^2 + ... + 1/(E(n)+1)^{L(n)-1}))$. If $E(n)$ is greater than or equal to 1 the time is bounded by $2E(n)$, so inserting an element has $O(E(n))$ expected running time. Note that as a special case, inserting or deleting an element at the left end of a list requires $O(1+1/(E(n)+1) + 1/(E(n)+1)^2 + ... + 1/(E(n)+1)^{L(n)-1})$, or $O(1)$ expected time.

**When $E(n)$ changes as $n$ changes**

We must know $E(n)$ when we create an element so that we can assign a *level* to it. If $L(n) \neq (\log n)/c$, $E(n)$ will change as $n$ changes. If we have a good idea of the maximum value of $n$, we can pick a value for *nMax* in advance and pre-compute $E(nMax)$. If the value of $n$ will not be known in advance, we can simply reassign a *level* to all elements every time $n$ doubles or halves. We can renumber

the *levels* of all elements and restore all the invariants in $O(n)$ time, so the cost of renumbering all elements can be amortized as a constant time overhead for the creation or destruction of an element. We renumber the *levels* of all the elements of a list simply by removing element one by one from the right end of the list, assigning the element a new *level*, and inserting it at the left end of a new list. All elements must be assigned a level using the same value of $E(n)$, otherwise we loose the property that the level of an element is completely independent of the sequence of operations that have occurred.

## Probability Analysis

Because the *level* of an element is never affected by an operation on it, nor is the ordering of items in the lists affected by the *levels* of elements, the distribution of *levels* in a list of elements is always completely random. We assume that the user does not have access to the *levels* of elements. If he did, an adversarial user could create situations with worst-case running times by going through a list and deleting all elements that were not *level* 1. A user without access to the levels of elements might do this by chance, but as we shall show below, the probability of this is small enough to be ignored.

If the distribution of *levels* has been computed as above using $p$, then E(n) can be computed as:

$$E(n) = 0\,p + (1+E(n))(1-p)$$

Solving for $p$ in terms of $E(n)$ we get:

$$p = 1/(E(n)+1)$$

Since we only want to have $E(n)$ elements of *level* $L(n)$ in an entire list, there is a relationship between $E(n)$, $p$, $L(n)$ and $n$. In particular:

$$(\#\ of\ elements\ of\ level\ L(n)) = E(n)$$

$$n\,p^{L(n)-1} = E(n)$$

Substituting for $p$ we get:

$$n/(E(n)+1)^{L(n)-1} = E(n),\quad n = E(n)(E(n)+1)^{L(n)-1},$$

$$or\ E(n) = n^{1/L(n)}-1$$

The probability that we will find exactly $k$ elements of *level* $i$ before finding

an element of a *level* greater than $i$ is $p(1-p)^k$. The probability that we will find $k$ or more elements of *level* $i$ before finding an element of a *level* greater than $i$ is $(1-p)^k$.

To calculate the length of the *left_sibling* chain, we will assume that the element we are tracing the *left_sibling* chain from is near the right end of the list, which produces the longest expected chain. Because we are near the right end of the list, we assume the element is to the right of the rightmost element of *level L(n)* in the list. The expected length of the *left_sibling* chain is then equal to the number of elements of *level L(n)* plus the length of the chain we have to traverse to reach an element of *level L(n)*. Let $E_k(n)$ be the expected number of elements we have to traverse to climb up $k$ levels:

$$E_1(n) = E(n)$$
$$E_k(n) = p\,(E_{k-1}(n)+1) + (1-p)\,(1+E_k(n))$$

Solving for $E_k(n)$ we get:

$$E_k(n) = k\,E(n) + k$$

The expected length of the *left_sibling* chain is then:

*(# elements of level L(n)) + (length of chain to reach an element of level L(n))*

which is $E(n) + E_{L(n)-1}(n)+L(n)-1$, or $L(n)\,(E(n)+1)$

Let $f(k,l)$ be the probability that we have to traverse $k$ or more elements of a left_sibling chain to climb up $l$ *levels.* Imagine that during this traversal, each time we get to a new element, we flip a weighted coin and for each head we get before we get a tail the *level* of the element reached is one greater than the *level* of the element we are coming from. We will slightly overcharge ourselves and count the number of coins flipped instead of the number of elements traversed. With this model, $f(k,l)$ is the probability that we have to flip $k$ or more weighted coins to get $l$ heads, where the probability of getting a head on a single flip is $p$. The equations for $f(k,l)$ are:

$$f(l,l) = 1$$
$$f(k,l) = p\,f(k-1,l-1) + (1-p)\,f(k-1,l)$$

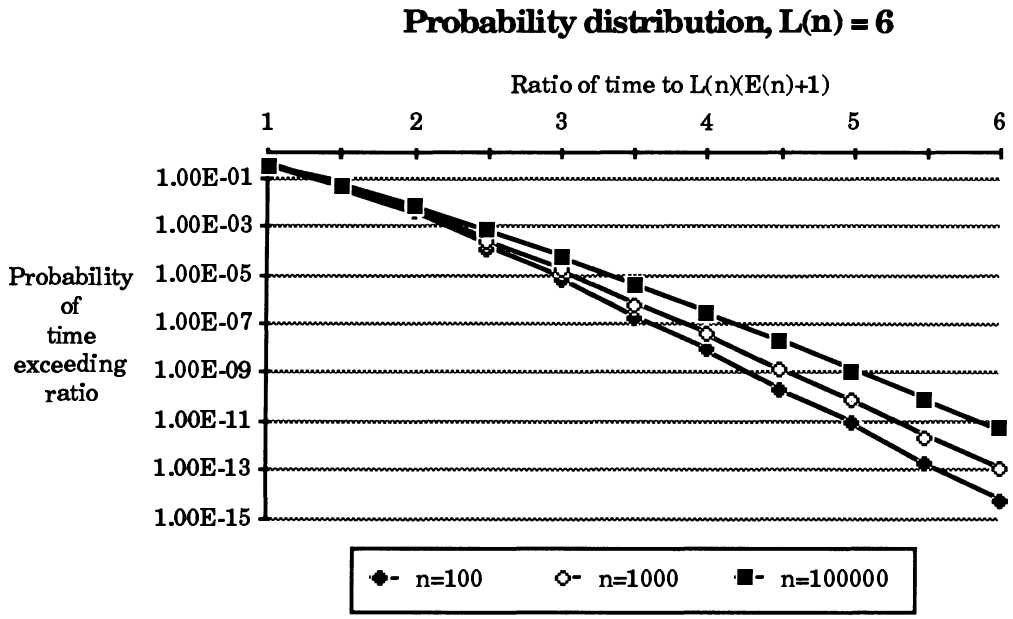
Solving for $f(k,l)$ gives us:

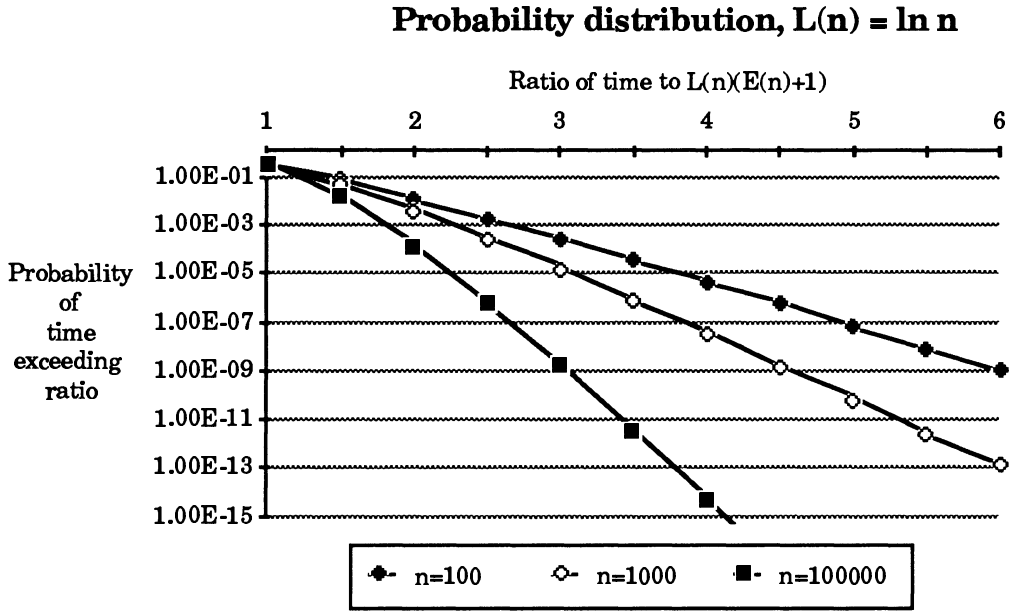
$$f(k,l) = (1-p)^{k-l} \sum_{i=0}^{k-1} p^i \frac{(k-l-1+i)!}{(k-l-1)!\, i!}$$

The probability that the *left_sibling* chain is $k$ elements or longer is equal to the sum, as $j$ ranges from $0$ to $k$, of the probability that exactly $j$ elements of level $L(n)$ exist times the probability that we will have to traverse $k-j$ or more elements to reach an element of level $L(n)$:

$$\sum_{j=0}^{k} \left( p^{(L(n)-1)j} (1-p^{L(n)-1})^{k-j} \frac{k!}{(k-j)!\, j!} \right) f(k-j, L(n)-1)$$

This formula is a bit too complicated to get an intuitive feel for, so the graphs below show the plot of the function.






## Reducing the time to insert or delete an element

The time to insert or delete a *level* 1 element is dominated by the time required to renumber the *right_siblings* counters of all the *level* 1 elements between it and the first element to the left with a *level* greater than 1. The time required to insert or delete a *level* 1 element can be reduced to a $O(1)$ cost by using a sparse sibling numbering for *level* 1 elements. The numbering may fill up and require us to renumber a sequence of *level* 1 elements. However, if $E(n) < log(MaxInteger/E(n))$ , this can only happen at most once every $E(n)$ operations, so this adds only a constant factor to the expected time of an insert or delete operation when $E(n)$ is bounded as above. The expected time to insert or delete an element then becomes $O(1+E(n)(1/(E(n)+1)) + 2E(n)(1/(E(n)+1))^2 + ...)$, which is $O(1)$ .

If we know we are going to be inserting an entire sequence of elements at one location, there is another method. We can build a list of $m$ elements in $O(m)$ time by inserting them one by one at the left end of a new list. We can then perform an operation to insert a list that is similar to the inserting a single element. The time

required to insert a list is $O(l\ E(n))$, where $l$ is the highest *level* of any element in the list. This allows us to insert a list $m$ long in $O(m + E(n)\ log(m)/log(E(n))\ )$ expected time.

A third method is that if we are willing to have the cost of the compare operation rise to $O(L(n) + n^{1/L(n)})$, we can reduce the expected time to insert or delete an element to $O(1)$ by eliminating the *right_siblings* counters.

## Use as search trees

The structures described in this paper have a equivalence to $E(n)$-ary search trees, where height of the tree is $L(n)$, and the height of an element in the tree is the level of the element. By maintaining some additional pointers, we can at slight additional constant overhead use the structures presented here to perform any operation that a search tree might be used for, with a worst-case time of $O(L(n))$ for any operation that climbs to the root of the tree from an element, and expected $O(L(n)\ n^{1/L(n)})$ time for any operation that searches down from the root searching for an element.

To search for elements, elements must be ordered by a *key* . We define the *key* of a left terminator to be less than any *key*, and the *key* of a right terminator to be greater than any *key* . Since elements are keyed, we now longer need to support the compare operation, and eliminate the *right_sibling* counters. To allow us to search a list for an element, we must redefine the left siblings of element $x$. For each element $x$ (*including* the right terminator) we maintain:

*left_sibling(x)[1..level(x)]* - *left_sibling(x)[k]* points to the first element to the left of element $x$ that is *level* $k$ or higher.

This allows us to search for an element with key $v$ in list $L$ as follows:

```
k ← L(n);
x ← right_terminator(L);
while k > 0 do
        while key(left_sibling(x)[k]) > v  do
                x ← left_sibling(x)[k];
        k ← k-1;
```

When this terminates, $x$ will either point to an element with key $v$, or if it does not exist, will point to the element with the next higher key. This search will take $O(L(n)E(n))$ expected time and maintaining the necessary pointers will require additional storage space for each element equal to $1/(E(n)+1) + 1/(E(n)+1)^2 + \ldots + 1/(E(n)+1)^{L(n)-1}$, which is $O(1)$. Maintaining these additional pointers will not increase our running times.

If we wish, we can start searching for a object from a starting point, or *finger* , as described in [4]. To do this, we set $x$ to be the element to start searching from and initialize the search as follows before executing the main loop of the search described above:

```
while key(x)<v or key(left_sibling(x)[level(x)]) >v  do
        x ← right_parent(x);
k ← level(x);
```

This gives us an expected time for the search of $O((1+ \log m / \log E(n)) E(n))$ , where $m$ is the distance between the element being searched for and $x$.

To optimize $L(n)$ for use in search trees, we wish to minimize $L(n)(E(n)+1)$. We assume $L(n)$ is of the form $(\ln n)/c$. The derivative of $((\ln n)/c)(e^c)$ is $(1-1/c)e^c$. Setting this equal to zero, we find that the equation for $L(n)$ that gives the best time results is $\ln n$ .

## Conclusion

If a search tree is built using random insertions, it is almost certain to be reasonably well-balanced[5]. The problem is that insertions are rarely random and common operation sequences, such as inserting all the elements in order, produce degenerate trees and unacceptable execution times. The standard solution to this problem has been to constantly reorganize a tree as changes are made to it to insure that it remains relatively balanced.

In our solution, the structure that we build is totally oblivious to the order in which updates are applied to it, and it is impossible for any common or predictable sequence of operations to consistently create a degenerate structure. Because the structure is guaranteed to be random we refer to the structures described in this paper as *guaranteed-random trees*. With *guaranteed-random trees*, we can avoid the overhead of maintaining the balance properties of a tree, while providing a virtual certainty that the tree is reasonably balanced. It is not clear if it would be possible to design a method using fixed-arity tree nodes that would have this property, but we feel quite happy with our solution of having tree nodes with an unbounded arity. We have only begun to study this area, and hope that this technique many prove useful in solving many different problems.

In addition to being more efficient and easier to understand than balanced trees, this approach may have advantages for real-time and interactive applications. Some balanced tree schemes only give good performance using amortized time; occasionally an operation might require $O(n)$ time instead of $O(\log n)$ time. With our approach, it is virtually impossible for an operation to require more than four or five times the expected time, and so we can guarantee good response in a real time or interactive environment.

## Acknowledgements

We would like to thank Tim Teitelbaum, Dexter Kozen, John Gilbert, Carla Marceau and Brad Vander Zander for their comments on earlier drafts of this paper.

## References

[1] Aho, Hopcroft and Ullman, *The Design and Structure of Computer Algorithms,* Addison-Wesley, 1974, pp. 155-156.

[2] P. Dietz, *Maintaining order in a linked list,* 14th annual ACM Symposium on the Theory of Computing, 1982, pp. 122-127.

[3] P. Dietz and D. Sleator, *Two Algorithms forMaintaining Order in a List,* 19th annual ACM Symposium on the Theory of Computing, 1987, *to appear.*

[4] S. Huddleston and K. Mehlhorn, *A New Data Structure for Representing Sorted Lists*, Acta Informatica, 17, 1982, pp. 157-184

[5] D. Knuth, *Sorting and Searching*, The Art of Computer Programming, Vol. 3, Addison-Wesley, 1973, p. 427.

[6] D. Sleator and R. Tarjan, *Self-adjusting binary trees,* Proc. Fifteenth Annual ACM Symposium on the Theory of Computing, 1983, pp. 235-245.

[7] A. Tsakalidis, *Maintaining Order in a Generalized Linked List,* Acta Informatica 21 (1984), pp. 101-112.

## Appendix

The following is an implementation of the algorithms described in the paper. It is assumed that an element is assigned a level when it is created, and the routine for creating an element is not given here.

```
Compare(p,q)
if (p=q) return(same_element );
i ← 0;
p[0]← p;
while not is_terminator(p) do
    p←right_parent(p);
    i←i+1;
    p[i]← p;
j ← 0;
q[0]← q;
while not is_terminator(q) do
    q ← right_parent(q);
    j ← j+1;
    q[j] ← q;
if p≠q then return(not_in_same_list);
k ← 0;
while (p[i-k]=q[j-k] & i > k & j > k)    do
    k←k+1;
if  p[i-k]=q[j-k] then
    if  i > k
        then return(p_is_to_the_left)
        else return(q_is_to_the_left)
if  level(p[i-k]) > level(q[j-k])
    or level(p[i-k]) = level(q[j-k])
        and right_siblings(p[i-k])
            > right_siblings(q[j-k])
    then return(p_is_to_the_left)
    else return(q_is_to_the_left);
```

```
ContainingList(x)
while not(is_terminator(x) do
    x ← right_parent(x);
return(listOf(x));
```

```
Split(x)
r ← x;
l ← left(x);
newRightTerminator
    ← newTerminator();
newLeftTerminator
    ← newTerminator();
rightList ← newList();
leftTerminator(rightList)
    ← newLeftTerminator;
listOf(newLeftTerminator)
    ← rightList;
right(l) ← newRightTerminator;
left(newRightTerminator)  ← l;
left(r) ← newLeftTerminator;
right(newLeftTerminator)  ← r;
s ← 0;
k ← 0;
while  not is_terminator(l) do
    right_parent(l)
        ← newRightTerminator;
    right_siblings(l)  ← s;
    if  level(left_sibling(l)) > level(l)
        then s ← 0
        else s ← s+1;
    l ← left_sibling(l);
while not is_terminator(r) do
    left_sibling(r)
        ← newLeftTerminator;
    r ← right_parent(r);
rightTerminator(listOf(r))
    ← newRightTerminator;
listOf(newRightTerminator)
    ← listOf(r);
listOf(r) ← rightList;
```

Append(L1,L2)

```
l ← left(rightTerminator(L1));
r ← right(leftTerminator(L2));
s ← 0;
k ← 0;
while not is_terminator(l) do
    while level(r) ≤ level(l) do
        if level(r)=level(l) then
            s ← right_siblings(r)+1;
            left_sibling(r) ← l;
        else s ← 0;
        r ← right_parent(r);
    right_parent(l) ← r;
    right_siblings(l) ← s;
    s ← s+1;
    l ← left_sibling(l);
while not is_terminator(r) do
    left_sibling(r) ← l;
    r ← right_parent(r);
free(rightTerminator(L1));
free(leftTerminator(L2));
free(L2);
rightTerminator(L1) ← r;
```

Delete(x)

```
l ← left(x);
r ← right(x);
s ← 0;
k ← 0;
while level(l) < level(n) do
    while level(r) ≤ level(l) do
        if level(r)=level(l) then
            s ← right_siblings(r)+1;
            left_sibling(r) ← l;
        else s ← 0;
        r ← right_parent(r);
    right_parent(l) ← r;
    right_siblings(l) ← s;
    s ← s+1;
    l ← left_sibling(l);
while level(r) ≤ level(x) do
    left_sibling(r) ← l;
    if level(r) = level(l) then
        s ← right_siblings(r)+1;
    r ← right_parent(r);
while level(l) = level(x) do
    right_siblings(l) ← s;
    s ← s+1;
    l ← left_sibling(l);
```

Insert(x,y)

```
r ← y;
l ← left(r);
right_siblings(x) ← 0;
while level(r) ≤ level(x) do
        if level(r)=level(x) then
            s ← right_siblings(r)+1;
            left_sibling(r) ← x;
        else s ← 0;
        r ← right_parent(r);
right_parent(x) ← r;
s ← 0;
k ← 0;
while level(l) < level(x) do
    right_siblings(l) ← s;
    right_parent(l) ← x;
    if level(left_sibling(l)) > level(l)
        then s ← 0
        else s ← s+1;
    l ← left_sibling(l);
left_sibling(x) ← l;
s ← right_siblings(x)+1;
while level(l)=level(x) do
    right_siblings(l) ← s;
    s ← s+1;
    l ← left_sibling(l);
```