UMIACS–TR–89–72.1 July, 1989
CS–TR–2286.1 (Revised June, 1990)

A Skip List Cookbook

William Pugh
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland, College Park

Abstract

Skip lists are a probabilistic data structure that seem likely to supplant balanced trees as the implementation method of choice for many applications. Skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space. The original paper on skip lists only presented algorithms for search, insertion and deletion. In this paper, we show that skip lists are as versatile as balanced trees. We describe and analyze algorithms to use search fingers, merge, split and concatenate skip lists, and implement linear list operations using skip lists. The skip list algorithms for these actions are faster and simpler than their balanced tree cousins. The merge algorithm for skip lists we describe has better asymptotic time complexity than any previously described merge algorithm for balanced trees.

CR Categories and Subject Descriptors: E.1 [**Data Structures**]: Lists; F.1.2 [**Models of Computation**]: Probabilistic computation; F.2.2 [**Nonnumerical Algorithms and Problems**]: Sorting and searching; G.3 [**Probability and Statistics**] Probabilistic Algorithms.

1. Overview

Skip lists [1] are a probabilistic list-based data structure that are a simple and efficient substitute for balanced trees. Probabilistic balancing also can be used for tree based data structures [2] [3]. Previously, we showed how skip lists could be used to implement a dictionary abstract data type (i.e., implement search, insertion and deletion). In this paper, we show that skip lists are as versatile as balanced trees. We describe and analyze algorithms to:

- use search fingers so that searching for an element k away from the last element searched for takes $O(\log k)$ expected time,
- merge, split and concatenate skip lists, and
- implement linear list operations using skip lists (e.g., "insert this after the k^{th} element of the list").

These operations have been described for balanced trees [4] [5] [6] [7] [8] [9]. However, the skip list versions of these algorithms are simpler and at least as fast, and often cannot be easily adapted directly from the balanced trees algorithms. The analysis techniques required for the skip list versions are radically different from the techniques used to analyze their balanced tree cousins.

The merge algorithm we describe has better asymptotic time complexity than any previously described merge algorithm (such as Brown and Tarjan's [7]). This claim may seem ludicrous, since the $O(m + m \log n/m)$ upper bound of Brown and Tarjan was proven to be a lower bound. But that lower bound only holds for the worst-case input (uniformly distributed merges). Our algorithm is optimal for *all* inputs. If two data structures simply need to be concatenated, our algorithm runs in $O(\log n)$ expected time, while that of Brown and Tarjan runs in $O(m + \log n)$ time. Of course, there are algorithms that concatenate two balanced trees in $O(\log n)$ time. However, our algorithm optimally handles both uniformly distributed merges and concatenation, as well as everything in between. Our strategy for merging skip lists can be applied to balanced trees, providing merge algorithms for balanced trees that are optimal for all inputs (although the algorithms would be prohibitively complicated).

We also describe and analyze variations that noticeably simplify or improve skip list algorithms.

2. A Review of Skip Lists

An understanding of skip lists is crucial to understanding the algorithms presented in this paper. To understand the analysis of the algorithms presented here, a reader also must be familiar with the probabilistic techniques and terminology developed for the analysis of skip lists. To make this paper self contained, a condensed review of skip lists is presented in this section. Other papers on skip lists also compare the constant factors of skip lists and balanced trees [1] and describe simple and efficient methods for performing concurrent maintenance of skip lists [10].

Each element is represented by a node in a skip list (Figure 1). Each node has a height or level, which corresponds to the number of forward pointers the node has. A node's i^{th} forward pointer points to the next node of level i or higher. When a new element is inserted into the list, a node with a random level is inserted to represent the element. Random levels are generated with a simple pattern: 50% are level 1, 25% are level 2, 12.5% are level 3 and so on. It should be fairly clear how to perform efficient searches, insertions and deletions in this data structure. Insertions or deletions would require only local modifications. Some arrangements of levels would give poor execution times, but we will see that such arrangements are rare. The expected cost of a search, insertion or deletion is $O(\log n)$. More details and intuitions about skip lists are described elsewhere [Pug89].

2.1. Skip List Algorithms

This section gives algorithms to search for, insert and delete elements in a dictionary or symbol table. The *Search* operation returns the contents of the value associated with the desired key or *failure* if the key is not present. The *Insert* operation associates a specified key with a new value (inserting the key if it had not already been present).

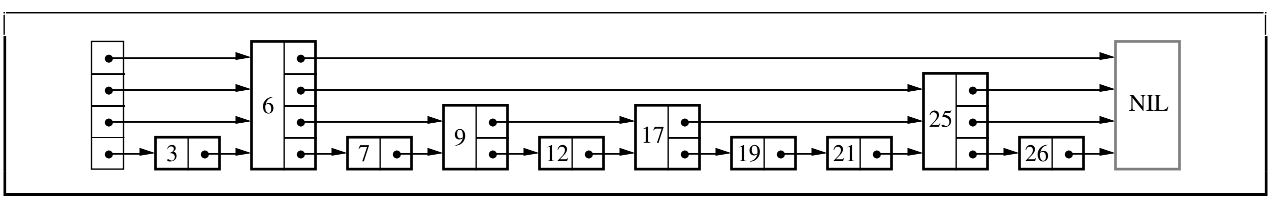

FIGURE 1 - A Skip List

The *Delete* operation deletes the specified key.

Each element is represented by a node, the level of which is chosen randomly when the node is inserted without regard for the number of elements in the data structure. A *level i* node has *i forward* pointers, indexed 1 through *i*. We do not need to store the level of a node in the node. Levels are capped at some appropriate constant *MaxLevel*. The *level* of a list is the maximum level currently in the list (or 1 if the list is empty). The *header* of a list has forward pointers at levels one through *MaxLevel*. The forward pointers of the header at levels higher than the current maximum level of the list point to NIL.

Initialization

An element NIL is given a key greater than any legal key. All levels of all skip lists are terminated with NIL. A new list is initialized so that the *level* of the list is equal to 1 and all forward pointers of the list's header point to NIL.

Search Algorithm

We search for an element by traversing forward pointers that do not overshoot the node containing the element being searched for (Figure 2). When no more progress can be made at the current level of forward pointers, the search moves down to the next level. When we can make no more progress at level 1, we must be immediately in front of the node that contains the desired element (if it is in the list).

```
Search(list, searchKey)
    x := list→header
    -- loop invariant: x→key < searchKey
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
    -- x→key < searchKey ≤ x→forward[1]→key
    x := x→forward[1]
    if x→key = searchKey then return x→value
        else return failure
```

FIGURE 2 - Skip list search algorithm

Insertion and Deletion Algorithms

To insert or delete a node, we simply search and splice. Figure 3 gives algorithms for insertion and deletion. A vector *update* is maintained so that when the search is complete (and we are ready to perform the splice), *update*[*i*] contains a pointer to the rightmost node of level *i* or higher that is to the left of the location of the insertion/deletion. If an insertion generates a node with a level greater than the previous maximum level of the list, we update the maximum level of the list and initialize the appropriate portions of the update vector. After each deletion, we check if we have deleted the maximum element of the list and if so, decrease the maximum level of the list.

```
Insert(list, searchKey, newValue)
    local update[1..MaxLevel]
    x := list→header
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
        update[i] := x
    x := x→forward[1]
    if x→key = searchKey then x→value := newValue
    else
        newLevel := randomLevel()
        if newLevel > list→level then
            for i := list→level + 1 to newLevel do
                update[i] := list→header
            list→level := newLevel
        x := makeNode(newLevel , searchKey, value)
        for i := 1 to newLevel do
            x→forward[i] := update[i]→forward[i]
            update[i]→forward[i] := x

Delete(list, searchKey)
    local update[1..MaxLevel]
    x := list→header
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
          x := x→forward[i]
        update[i] := x
    x := x→forward[1]
    if x→key = searchKey then
        for i := 1 to list→level do
          if update[i]→forward[i] ≠ x then break
          update[i]→forward[i] := x→forward[i]
        free(x)
        while list→level > 1 and
            list→header→forward[list→level] = NIL do
          list→level := list→level – 1
```

FIGURE 3 - Insertion and deletion algorithms

Generating a Random Level

Initially, we discussed a probability distribution where half of the nodes that have level *i* pointers also have level *i*+1 pointers. To get away from magic constants, we say that a fraction *p* of the nodes with level *i* pointers also have level *i*+1 pointers. (for our original discussion, p = 1/2). Levels are generated randomly by an algorithm equivalent to the one in Figure 4. Levels are generated without reference to the number of elements in the list.

At what level do we start a search? Defining *L*(*n*)

In a skip list of 16 elements generated with $p = 1/2$, we might happen to have 9 elements of level 1, 3 elements of level 2, 3 elements of level 3, and 1 element of level 14 (this would be very unlikely, but it could happen). How should we handle this? Where should we start the search? Our analysis suggests that ideally we would start a search at the level L where we expect $1/p$ nodes. This happens when $L = \log_{1/p} n$. Since we will be referring frequently to this formula, we will use $L(n)$ to denote $\log_{1/p} n$. If we use the standard algorithm and start our search at level 14, we will do much useless work. However, the probability that the maximum level in a list of n elements is significantly larger than $L(n)$ is very small. Starting a search at the maximum level in the list does not add more than a small constant to the expected search time. This is the approach used in the algorithms described here.

```
randomLevel()
    lvl := 1
    while random() < p and lvl < MaxLevel do
        lvl := lvl + 1
    return lvl
```

FIGURE 4 - Generating a random level

Determining *MaxLevel*

Since we can safely cap levels at $L(n)$, we should choose $MaxLevel = L(N)$ (where N is an upper bound on the number of elements in a skip list). If $p = 1/2$, using $MaxLevel = 32$ is appropriate for data structures containing up to 2^{32} elements.

2.2 Analysis of Skip List Algorithms

The time required to execute the *Search, Delete* and *Insert* operations is dominated by the time required to search for the appropriate element. The time required to find an element is proportional to the length of a search path, which is determined by the pattern in which elements with different levels appear as we traverse the list.

Probabilistic Philosophy and Assumptions

We assume an adversarial user does not have access to the levels of elements; otherwise, he 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, the probability of this is small enough to be ignored. Note that the probabilities of poor running times for successive operations on the same data structure are NOT independent; two successive searches for the same element will both take exactly the same time.

Probabilistic Analysis

Besides analyzing the expected performance of skip lists, we also can analyze the probabilistic performance of skip lists. This will allow us to calculate the probability that an operation takes longer than a specified time. This analysis is based on the same ideas as our analysis of the expected cost, so that analysis should be understood first.

A *random variable* has a fixed but unpredictable value and a predictable probability distribution and average. If X is a random variable, Prob{ $X = t$ } denotes the probability that X equals t and Prob{ $X > t$ } denotes the probability that X is greater than t. For example, if X is the number obtained by throwing an unbiased die, Prob{ $X > 3$ } = 1/2.

It is often preferable to find simple upper bounds on values whose exact value is difficult to calculate. To discuss upper bounds on random variables, we need to define a partial ordering and equality on the probability distributions of non-negative random variables.

Definitions ($=_{prob}$ and $\le_{prob}$). Let X and Y be non-negative independent random variables (typically, X and Y would denote the time to execute algorithms A_X and A_Y). We define $X \le_{prob} Y$ to be true if and only if for any value t, the probability that X exceeds t is less than the probability that Y exceeds t. More formally:

$X =_{prob} Y$ iff $\forall\, t$, Prob{ $X > t$ } = Prob{ $Y > t$ } and

$X \le_{prob} Y$ iff $\forall\, t$, Prob{ $X > t$ } ≤ Prob{ $Y > t$ }. □

We make use of two probability distributions:

Definition (binomial distributions — $B(t, p)$). Let t be a non-negative integer and p be a probability. The term $B(t, p)$ denotes a random variable equal to the number of successes seen in a series of t independent random trials where the probability of a success in a trial is p. The average and variance of $B(t, p)$ are tp and $tp(1 - p)$ respectively. □

Definition (*negative binomial distributions — $NB(s, p)$*). Let s be a non-negative integer and p be a probability. The term $NB(s, p)$ denotes a random variable equal to the number of failures seen before the s^{th} success in a series of random independent trials where the probability of a success in a trial is p. The average and variance of $NB(s, p)$ are $s(1–p)/p$ and $s(1–p)/p^2$ respectively.□

Probabilistic Analysis of Search Times

We analyze the search path in a backward direction, starting at the element immediately to the left of the element searched for and travelling up and to the left. The length of this path is one less than the number of comparisons we need to perform. We first examine the number of pointers we have to backtrack to climb from level 1 (of the element immediately before the element searched for) up to level $L(n)$. We assume we have no knowledge of the levels of elements in the list, and that we always reach level $L(n)$ before we reach the header of the list (i.e., the list extends infinitely to the left).

At any particular point in the climb, we are at the i^{th} forward pointer of an element x, and we have no knowledge about the levels of elements to the left of x or about the level of x, other than that the level of x must be at least i. The probability that the level of x is greater than i is p. We can analyze the time to traverse the search path by considering this backward climb to be a series of random independent trials with a success corresponding to a move up a level, and a failure corresponding to a move left. The number of leftward movements in the path climbing to level $L(n)$ is the number of failures seen before seeing the $(L(n) – 1)^{\text{th}}$ success in a series of random trials, which is a negative binomial distribution. The number of upward movements is exactly $(L(n) – 1)$. This gives us:

Cost to climb to level $L(n)$ in an infinite list $=_{prob} (L(n) – 1) + NB(L(n) – 1, p)$

Our assumption that the list is infinite is a pessimistic assumption. When we bump into the header in our backward climb, we simply climb up it, without performing any leftward movements. This gives us:

Cost to climb to level $L(n)$ in a list of n elements $\leq_{prob}$ Cost to climb to level $L(n)$ in an infinite list

$\therefore$ Cost to climb to level $L(n)$ in a list of n elements $\leq_{prob} (L(n) – 1) + NB(L(n) – 1, p)$

Once we have climbed to level $L(n)$, the number of leftward movements is bounded by the number of elements of level $L(n)$ or greater in a list of n elements. The number of elements of level $L(n)$ or greater in a list of n elements is a random variable of the form $B(n, 1/np)$.

Let M be a random variable corresponding to the maximum level in a list of n elements. The probability that the level of a node is greater than k is p^k, so Prob$\{ M > k \} = 1–(1–p^k)^n < np^k$. Since $np^k = p^{k-L(n)}$ and Prob$\{ NB(1, 1–p) + 1 > i\} = p^i$, we get a probabilistic upper bound of $M \leq_{prob} L(n) + NB(1, 1 – p) + 1$. Note that the average of $L(n) + NB(1, 1 – p) + 1$ is $L(n) + 1/(1–p)$.

This gives a probabilistic upper bound on the cost once we have reached level $L(n)$ of $B(n, 1/np) + (L(n) + NB(1, 1 – p) + 1) – L(n)$. Combining our results to get a probabilistic upper bound on the total length of the search path (i.e., cost of the entire search):

cost to climb out of a list of n elements $\leq_{prob} L(n) + NB(L(n) – 1, p) + B(n, 1/np) + NB(1, 1 – p)$

The number of comparisons is one more than the length of the search path:

number of comparisons $\leq_{prob} L(n) + NB(L(n) – 1, p) + B(n, 1/np) + NB(1, 1 – p) + 1$

The expected value of our upper bound is equal to $L(n)/p + 1/(1–p) + 1$, and the variance of our upper bound is $(L(n) – 1)(1–p)/p^2 + (1 – 1/np)/p + p/(1–p)^2$.

2.3. Efficiency

We compared [1] implementations of skip lists against implementations AVL trees, 2-3 trees and splay trees. We found that skip lists had roughly the same efficiency as highly optimized, non-recursive balanced tree implementations (insertions and deletions were slightly faster in skip lists), and that skip lists were significantly faster (by a factor of 2–3) than straightforward, recursive balanced tree implementations or highly optimized splay tree implementations (for uniform query distributions).

3. Skip List Extensions

This section describes our new results. We first describe algorithms for using search fingers, merging, splitting and concatenation. Linear list operations, described next, allow us to implement operations such as “insert this after the k^{th} element of the list.” The algorithms given allow only linear list operations, but it is easy to combine the linear list versions with the standard skip list algorithms to provide skip lists that allow operations both by key and by position.

We next return to the original algorithms and look for ways to improve them. We discuss how to reduce the number of comparisons when comparisons are expensive, mention the changes needed to allow duplicate elements and examine more precisely the cost to search for the k^{th} element of a skip list containing n elements. We also describe a new probability distribution for the levels of elements that provides both low space costs and a low search time variance.

3.1 Using Search Fingers

We can maintain a *finger* into a search structure so that the expected time to search for an element is $O(\log k)$, where k is the distance between the previous element searched for and the new element [6] [8] [9]. A *finger* is associated with each list and defined so that *finger*(*list*)[k] is the rightmost element of level k that is left of the last element examined. This is identical to the *update* vector that is used for the insertion and deletion operations, except that the finger is maintained from one operation to the next. An algorithm to perform a search using search fingers is shown in Figure 6. The overhead of using fingers could result in an overall slow down if searches do not have a locality of reference.

Theorem 1. The number of comparisons required to locate an element k away from the location last examined is bounded by

$$2L(k) + 2NB(1, 1 - p) + NB(L(k) - 1, p) + B(k, 1/kp) + 3$$

which has an average of $L(k) + L(k)/p + 2p/(1-p) + 4$ and a variance of $L(k)(1-p)/p^2 + (1 - 1/kp)/p + 4p/(1-p)^2$.

Proof: Determining if we have to move forward or backward requires 1 comparison. Let i be the position of the tip of the finger (i.e., the position of *finger*(*list*)[1]). Let j be the position of the element being searched for. If $i < j$, let *lvl* be the maximum level among elements $(i + 1) .. (j - 1)$. Otherwise (if $j \leq i$) let *lvl* be the maximum level among elements $j..i$. The finger search will climb up to level $lvl + 1$ before starting the decent phase of the

```
SearchWithFinger(list, searchKey)
    lvl := 2
    if list→finger[1]→key < searchKey then
        -- move forward, find the largest lvl s.t. list→finger[lvl]→forward[lvl]→key < searchKey
        while lvl ≤ list→level and list→finger[lvl]→forward[lvl]→key < searchKey do lvl := lvl + 1
        lvl := lvl – 1
        x := list→finger[lvl]
    else
        -- move backward, find the smallest lvl s.t. list→finger[lvl]→key < searchKey
        while lvl ≤ list→level and list→finger[lvl]→key ≥ searchKey do lvl := lvl + 1
        if lvl > list→level then
            lvl := list→level
            x := list→header
        else x := list→finger[lvl]

    for i := lvl downto 1 do
        while x→forward[i]→key < searchKey do x := x→forward[i]
        list→finger[i] := x

    x := x→forward[1]
    if x→key = searchKey then return x→value
        else return failure
```

FIGURE 6 – Search using finger

search. This will require lvl comparisons. Let $k = |i + 1 - j|$ (i.e., let k be the number of elements lvl is the maximal level over). The maximum level among those k elements (lvl) is probabilistically bounded by $L(k) + 1 + NB(1, 1 - p)$. The length of the (backwards) search path is bounded by $lvl - 1$ upward movements and $NB(L(k) - 1, p) + B(k, 1/kp)$ leftward movements.□

3.2 Optimal Merging of Skip Lists

The merge algorithm in Figure 7 merges two skip lists. The algorithm moves as many elements as possible from one input list to the output list, moves as many as possible from the other list and repeats this process until all elements have been moved to the output list. If an element appears in both lists, the value field from *list2* is used. Let k be the number of elements moved in a single merge step. The number of comparisons required works out to be the same as is required to search using a search finger. This gives an average of $L(k) + L(k)/p + 2p/(1 - p) + 3$ comparisons, which is $O(\log k)$.

We need to introduce some new terminology to describe the total cost to perform a merge, which consists of many merge steps. Let n be the size of *list1* and m be the size of *list2*. Assume w.l.g. that $m \le n$. We use $S_1...S_j$ to describe how elements are taken from *list1* and $T_1...T_k$ to describe how elements are taken from *list2*, as shown in Figure 8. If we first take elements from *list1*, then we take S_1 elements from *list1*, T_1 elements from *list2*, S_2 elements from *list1*, and so on. Otherwise, we take T_1 elements from *list2*, S_1 elements from *list1*, T_2 elements from *list2*, and so on. Note that j and k can differ by at most one, all of $S_1...S_j$ and $T_1...T_k$ are required to be positive integers, $n = S_1 + S_2 + \cdots + S_j$ and $m = T_1 + T_2 + \cdots + T_k$.

Our merge algorithm runs in expected time

$$\mathrm{O}\left(j + \sum_{i=1}^{j} \log S_i + k + \sum_{i=1}^{j} \log T_i \right)$$

Brown and Tarjan's algorithm for merging balanced trees [7] inserts the elements from the smaller list into the larger list one at a time, using a search finger to reduce the amount of work required. This algorithm runs in time

$$\mathrm{O}\left(j + \sum_{i=1}^{j} \log S_i + \sum_{i=1}^{k} T_i \right) = \mathrm{O}\left(j + \sum_{i=1}^{j} \log S_i + m \right)$$

The worst case situation for both of these algorithms is a uniformly distributed merge, in which $k = m$, all of the T_i's are equal to 1 and all of the S_i's are equal to n/m. For this situation, both algorithms run in $O(m + m \log n/m)$ expected time. Brown and Tarjan showed that this is a lower bound, and therefore optimal. But that lower bound holds only for uniformly distributed merges. Our algorithm is optimal for *all* inputs. If the data structures simply need to be concatenated, our algorithm runs in $O(\log n)$ expected time, while Brown and Tarjan's algorithm requires $O(m + \log n)$ time. There are specialized algorithms that concatenate balanced trees in $O(\log n)$ time. However, our algorithm optimally handles both uniformly distributed merges and concatenation, as well as everything in between.

Lower bounds and optimality

If sequences are represented in a way such that moving k elements down the list takes $\Omega(\log k)$ time and we must visit each of the locations in the input lists where the merge switches from one input list to the other, as well as the beginnings and ends of the lists, then the time to merge two sequences is

$$\Omega\left(j + \sum_{i=1}^{j} \log S_i + k + \sum_{i=1}^{k} \log T_i \right)$$

Since this is the same as the performance of our algorithm, our algorithm is optimal for all inputs.

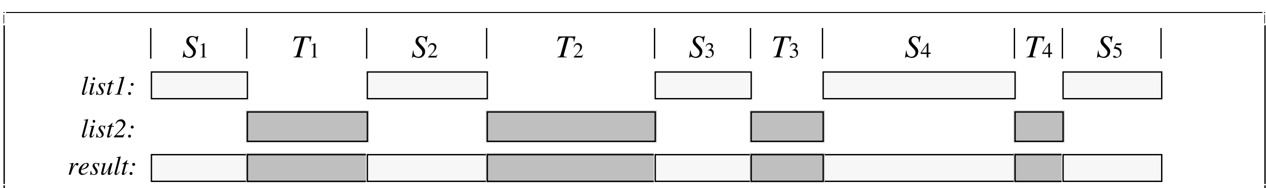

FIGURE 8 – Example showing how $S_1...S_5$ and $T_1...T_4$ describe the merge process.

Our assumptions seem incontestable. Our attempts to show optimally based on information theoretic terms failed, since is it possible to tailor a merge algorithm to use very few comparisons on any specific distribution. For example, it is possible to write a merge algorithm take requires only $2m$ comparisons for a uniformly distributed merge of two lists of size m and n ($m \le n$). However, such an algorithm would still require $\Omega(m \log n/m)$ work using almost any conceivable implementation and would require many more comparisons for a merge that did not have a perfectly uniform distribution.

Optimal merging of balanced trees

The strategy we have described for merging skip lists can be applied to merging balanced trees to give the same time bounds. In a merge step, as many elements as appropriate (k elements) are split off from the appropriate input trees and concatenated to the right side of the output tree, taking $O(\log k)$ time. However, merging balanced trees this way is considerably more complex than the merge algorithm we have presented for skip lists.

Constant Factor Speed Comparisons

Brown and Tarjan [7] presented an analysis of the constant factors for their merge algorithm on a hypothetical machine [11]. To produce directly comparable results, we designed a new merge algorithm for skip lists that used the same strategy they used: insert the elements from the smaller list one at a time into the larger list using a search finger. The resulting algorithm is 26 statements long. We calculated a probabilistic upper bound on the number of steps taken by our algorithm on a uniformly distributed merge of lists of m and n elements ($m \le n$) , which for p = 0.25 has an average of

$$m(17 \log_2 n/m + 42) + 6$$

steps. Their algorithm (which is over 100 statements long in full detail [12]) runs in

$$m(15 \log_2 n/m + 118.5) + 43.5$$

steps. These results are in line with our expectations: the portion of the algorithms that search for the place to insert the next element run at similar speeds. However, tree merging incurs a large overhead for set–up and rebalancing

```
Split(list1, splitKey) -- Remove from list1 all elements with keys ≥ splitKey and return them in a new list
list2 := newList()
list2→level := list1→level
x := list1→header
for i := list1→level downto 1 do
    while x→forward[i]→key < splitKey do x := x→forward[i]
    list2→header→forward[i] := x→forward[i]
    x→forward[i] := NIL

while list1→header→forward[list1→level] = NIL and list1→level > 1 do
    list1→level := list1→level – 1
while list2→header→forward[list→level] = NIL and list2→level > 1 do
    list2→level := list2→level – 1

return list2

Concatenate(list1, list2) --Appends list2 to the end of list1, assumes last key in list1 ≤ first key in list2
    if list1→level < list2→level then
        for i := list1→level+1 to list2→level do list1→header→forward[i] := NIL
        list1→level := list2→level

    x := list1→header
    for i := list1→level downto 1 do
        while x→forward[i] ≠ NIL do x := x→forward[i]
        if i ≤ list2→level then x→forward[i] := list2→header→forward[i]
    free(list2)
```

FIGURE 9– Concatenating and splitting skip lists

that makes skip list merging faster for most applications.

3.3 Splitting and Concatenation

The algorithms for splitting and concatenating are shown in Figure 9. They are fast (requiring $O(\log n)$ expected time) and very simple.

3.4 Linear List Operations

We now describe a method for implementing linear list operations [5] using skip lists (e.g., give me the k^{th} element of this list). These operations are sometimes refereed to as rank operations. The search, insertion and deletion algorithms are shown in Figures 10-11. Each element x has an index $pos(x)$. We use this value in our invariants but do not store it. The index of the header is zero, the index of the first element is one and so on. Associated with each forward pointer is a measurement, *fDistance*, of the distance traversed by that pointer:

$$x\to fDistance[i] = pos(x\to forward[i]) - pos(x).$$

```
SearchByPosition(list, k)
    if k < 1 or k > size(list) then return bad-index
    x := list→header
    pos := 0
    -- loop invariant: pos = pos(x)
    for i := list→level downto 1 do
        while pos + x→fDistance[i] ≤ k do
            pos := pos + x→fDistance[i]
            x := x→forward[i]
    return x→value
```

FIGURE 11 – Searching for the k^{th} element of a list

Note that the distance traversed by a level 1 pointer is always 1, so some storage economy is possible here at the cost of a slight increase in the complexity of the algorithms.

3.5 Skip List Variations

In this section, we present some variations on the skip list algorithms and analyses presented previously.

Reducing comparisons

If comparisons are expensive (*e.g.*, involve expensive comparisons of reals or strings) we can reduce the number of comparisons by assuring that we never compare the search key against the key of an element more than once. This optimization results in an inner loop like that shown in Figure 13. This optimization saves an average of $(L(n) + p/(1-p)^2)p$ comparisons.

```
alreadyChecked := NIL
for i := list→level downto 1 do
    while x→forward[i] ≠ alreadyChecked
            and x→forward[i]→key < searchKey do
        x := x→forward[i]
    alreadyChecked := x→forward[i]
    -- if performing insertion or deletion
    update[i] := x
```

FIGURE 12 – Inner loop optimized for expensive comparisons

Theorem 2. The number of comparisons required by the inner loop in Figure 12 is bounded by

$$B(L(n) + NB(1, 1 - p), 1 - p) + NB(L(n) - 1, p) + B(n, 1/np) + 1.$$

Proof: Consider the situation where we find we can make no more progress at the current level and need to move down from level i to level $i - 1$ (where i is not the maximum level in the list). Let y be the element who's key we just compared against the search key. The level of y is greater than or equal to i. If the level of y is greater than i, this optimization allows us to avoid comparing keys. The probability that the level of y is more than i is p. There are $L(n) + NB(1, 1 - p) + 1$ comparisons associated with upwards movements (in the backwards search path) and we might save a comparison for all except the first. □

Duplicate elements

It is easy to modify skip list algorithms so that they allow duplicate entries. The insertion algorithm simply needs to be modified so as to always perform the insertion. The search and deletion algorithms are unchanged. With these changes, the skip lists will react to duplicate elements with a stack-like discipline (e.g., the element found by a search is the one most recently inserted and a deletion removes the most recently inserted version of an element).

Precision Analysis

In previous analyses of skip lists, we made worst-case assumptions about the element being searched for. Here, we more precisely examine the cost to search for the k^{th} element of a skip list containing n elements.

Theorem 3. The number of comparisons required to search for, insert, or delete the k^{th} element of a skip list containing n elements is bounded by

$$L(n) + NB(L(k) - 1, p) + B(k, 1/kp) + NB(1, 1-p) + 1$$

which has an average of $L(n) + L(k)\,(1-p)/p + 1/(1-p) + 1$

Proof: The number of upward movements in the search path is bounded by $L(n) + NB(1, 1-p)$. The number of leftward movements to climb out of a list of k elements is $NB(L(k) - 1, p) + B(k, 1/kp)$. The number of comparisons is one greater than the length of the search path. □

Non-standard probability distributions

In previous descriptions of skip lists, we assigned a random level to an element using a negative binomial distribution equal to $1 + NB(1, 1-p)$. Consider a probability distribution in which the probability of an element being level i or greater is p^{i+k-1} (all elements are at least level 1). The original probability distribution corresponds to $k = 0$. This is equivalent to the standard distribution except that the level 2 through k+1 forward pointers have been removed and all higher level forward pointers have dropped down a level. For this probability distribution, $L(n) = \log_{1/p} n - k$ This distribution requires an average of $1+p^{k+1}/(1-p)$ pointers per element.

Theorem 4. Using this distribution, the number of comparisons required to perform a search is bounded by

$$\log_{1/p} n - k - 1 + NB(1, p^{k+1}) + NB(\log_{1/p} n - k - 2, p) + B(n, 1/np) + \mathrm{NB}(1, 1-p) + 2,$$

which has an average of $(\log_{1/p} n - k - 1 + 1/p^k)/p$ and a variance of $((\log_{1/p} n - k - 2)(1-p) + p + 1/p^{2k} - p/p^k + 1/n)/p^2$.

Proof: The number of leftward movements at level 1 is $NB(1, p^{k+1})$ (since the probability that an element is at least level 2 is 1/4). For i such that $2 \leq i < \log_{1/p} n - k$, the number of leftward movements at level i is $NB(1, p)$. All of the other costs are similar to the standard case. □

The motivation for using this distribution is shown in Table 2. Previously, to reduce the space costs of a skip list we would decrease p. Now that we have added this additional parameter to the probability distribution, increasing k is often a better way to decrease space costs (particularly if n is large).

4. Conclusions

The algorithms presented here put skip lists on a par with balanced trees for versatility, and in almost all cases the skip list algorithms are substantially simpler and at least as fast. This is partially because using a list-based structure to represent an ordered collection seems more natural than using a tree-based structure, and partially because probabilistic balancing seems simpler and more efficient than strictly enforced balancing.

Several researchers have reported to the author difficulties in developing and analyzing the sorts of extensions and variations described in this paper. Much of the reason for this difficulty may be that in order to develop and

Distribution/strategy	*p*	*k*	*Average of search cost*	*Variance of search cost*	*Average # pointers/element*
baseline distribution	1/2	0	$2 \lg n + 3$	$2 \lg n - 4/n$	2
old strategy: to reduce space	1/4	0	$2 \lg n + 7/3$	$6 \lg n - 68/9 - 16/n$	$1\,{}^1/_3$
costs, reduce p	1/8	0	$2\,{}^2/_3 \lg n + 15/7$	$18\,{}^2/_3 \lg n - 2344/49 - 64/n$	$1\,{}^1/_7$
new strategy: to reduce space	1/2	1	$2 \lg n + 3$	$2 \lg n + 10 - 4/n$	$1\,{}^1/_2$
costs, increase k	1/2	2	$2 \lg n + 5$	$2 \lg n + 52 - 4/n$	$1\,{}^1/_4$
mixed strategy:	1/4	1/2	$2 \lg n + 13/3$	$6 \lg n + 274/9 - 16/n$	$1\,{}^1/_6$

Table 2 – Average and variance of search costs and space utilizations using different probability distributions for the levels of elements

analyze these sorts of extensions, you need an intuition about skip lists that is as strong as the intuition that has developed for balanced tree structures over the past 27 years. A main goal of this paper is to help the research community develop that intuition.

References

[1] William Pugh, Skip Lists: A Probabilistic Alternative to Balanced Trees, *Proceedings of the Workshop on Algorithms and Data Structures,* Ottawa Canada, August 1989 (*to appear in Comm. ACM*).

[2] William Pugh and Tim Teitelbaum, Incremental Computation via Function Caching, *PROC of the 16th ACM Conf on the Principles of Programming Languages,* pp 315-328

[3] Cecilia Aragon and Raimund Seidel, Randomized Search Trees, *FOCS* 89, pp 540–545.

[4] Clark A. Crane, *Linear Lists and priority queues as balanced binary trees.* Ph.D. Thesis, STAN-CS-72-259, Computer Sci. Department, Stanford U., Stanford, CA, Feb. 1972.

[5] Donald Knuth. Sorting and Searching, *The Art of Computer Programming,* Vol. 3, 1973.

[6] L. Guibas, E. McCreight, M Plass and J. Roberts, A new representation for linear lists. *9th ACM Symposium on Theory of Computing,* Bolder CO, 1977, pp 49-60.

[7] Mark Brown and Robert Tarjan, A Fast Merging Algorithm, *J. of the ACM,* Vol. 26, No. 2, April 1979, pp. 211-226.

[8] Mark Brown and Robert Tarjan, Design and Analysis of a Data Structure for Representing Sorted Lists, *SIAM J. Comput.* 9, 594-614 (1980).

[9] S. Huddleston and Kurt Mehlhorn, A new data structure for represented sorted lists, *Acta Informatic*, 17, 157–184.

[10] William Pugh, *Concurrent Maintenance of Skip Lists,* TR–2222, Dept. of Computer Science, University of Maryland, College Park, April 1989

[11] Donald Knuth, Structured programming with **goto** statements. *Computing Surveys* 6, 4 (Dec 1974) 261-301.

[12] Mark Brown and Robert Tarjan, *A Fast Merging Algorithm,* STAN–CS–77–625, Computer Sci. Dept., Stanford U., Stanford, Calif, August 1977

```
-- create a new list containing the elements of list1 and list2 (destroys list1 and list2)
-- if an element appears in both lists, use the value field from list2

merge(list1, list2)

    local update[1..levelCap]
    unflipped := true -- false if list1 and list2 are interchanged

    -- initialize new list
    list := newList()
    list→level := max(list1→level, list2→level)
    for i := 1 to list→level do update[i] := list→header

    while list1→header→forward[1] ≠ NIL and list2→header→forward[1] ≠ NIL do

        key1, key2 := list1→header→forward[1]→key, list2→header→forward[1]→key

        -- assume w.l.g. that key1 ≤ key2 (i.e., if key1 > key2, exchange list1 and list2; key1 and key2)
        if key1 > key2 then list1, list2, key1, key2, unflipped := list2, list1, key2, key1, not unflipped

        -- merge step: remove from list1 elements with keys ≤ key2 and put them on the output list

        ---------- for all lvl s.t. list1→header→forward[lvl]→key ≤ key2, connect output list to list1
        lvl := 1
        repeat
            update[lvl]→forward[lvl] := list1→header→forward[lvl]
            lvl := lvl + 1
            until lvl > list→level or list1→header→forward[lvl]→key > key2
        lvl := lvl – 1

        ---------- for each level attached to output list, find endpoint at that level (i.e., last element with key ≤ key2)
        x := list1→header→forward[lvl]
        for i := lvl downto 1 do
            while x→forward[i]→key ≤ key2 do x := x→forward[i]
            -- x→key ≤ key2 < x→forward[i]→key
            update[i] := x
            list1→header→forward[i] := x→forward[i]

        -- x = last element moved to output list
        -- if the element at the front of list2 is a duplicate of an element already moved to output list, eliminate it
        if key2 = x→key then
            if unflipped then x→value := list2→header→forward[1]→value
            y := list2→header→forward[1]
            for i := 1 to y→level do list2→header→forward[i]:= y→forward[i]
            free(y)
        -- end of main while loop

    if list2→header→forward[1] = NIL then leftOver := list1 else leftOver := list2
    for i = 1 to leftOver→level do update[i]→forward[i] := leftOver→header→forward[i]

    -- the following two lines are necessary because we may have eliminated some duplicate elements
    for i = leftOver→level + 1 to list→level do update[i]→forward[i] := NIL
    while list→header→forward[list→level] = NIL and list→level > 1 do
        list→level := list→level – 1

    free(list1)
    free(list2)
```

FIGURE 7 – Merge algorithm

```
InsertByPosition(list, k, value) -- insert a new element immediately after position k
    if k < 0 or k > list→size then bad-index

    lvl := randomLevel()
    y := makeElement(lvl, value)
    if lvl > list→level then
        for i := list→level + 1 to lvl do
            list→header→forward[i] := NIL
            list→header→fDistance[i] := list→size + 1
        list→level := lvl

    x := list→header
    pos := 0 -- pos = pos(x)
    for i := list→level downto 1 do
        while pos + x→fDistance[i] ≤ k do
            pos := pos + x→fDistance[i]
            x := x→forward[i]
        if i > lvl then x→fDistance[i] := x→fDistance[i] + 1
        else
            z := x→forward[i] -- insert y between x and z
            y→forward[i] := z
            x→forward[i] := y
            -- new pos(z) = pos + old x→fDistance[i] + 1
            -- new pos(y) = k + 1
            y→fDistance[i] := pos + x→fDistance[i] – k    -- new y→fDistance = new pos(z) – new pos(y)
            x→fDistance[i] := k + 1 – pos                 -- new x→fDistance = new pos(y) – new pos(x)
    list→size := list→size + 1


DeleteByPosition(list, k) -- delete the k^th element
    local update[1..levelCap]
    if k < 1 or k > list→size then bad-index

    x := list→header
    pos := 0 -- pos = pos(x)
    for i := list→level downto 1 do
        while pos + x→fDistance[i] < k do
            pos := pos + x→fDistance[i]
            x := x→forward[i]
        update[i] := x

    x := x→forward[1]
    for i := 1 to list→level do
        if update[i]→forward[i] = x then
            update[i]→forward[i] := x→forward[i]
            update[i]→fDistance[i]:= update[i]→fDistance[i] + x→fDistance[i] – 1
        else
            update[i]→fDistance[i] := update[i]→fDistance[i] – 1

    list→size := list→size – 1
    while list→header→forward[list→level] = NIL and list→level > 1 do
        list→level := list→level – 1
```

FIGURE 10 – Insertion and Deletion by position