Skip Lists: A Probabilistic Alternative to Balanced Trees

Skip lists are a data structure that can be used in place of balanced trees. Skip lists use probabilistic balancing rather than strictly enforced balancing and as a result the algorithms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees.

William Pugh

Binary trees can be used for representing abstract data types such as dictionaries and ordered lists. They work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in order, produce degenerate data structures that give very poor performance. If it were possible to randomly permute the list of items to be inserted, trees would work well with high probability for any input sequence. In most cases queries must be answered on-line, so randomly permuting the input is impractical. *Balanced* tree algorithms re-arrange the tree as operations are performed to maintain certain balance conditions and assure good performance.

S*kip lists* are a probabilistic alternative to balanced trees. Skip lists are balanced by consulting a random number generator. Although skip lists have bad worst-case performance, no input sequence consistently produces the worst-case performance (much like quicksort when the pivot element is chosen randomly). It is very unlikely a skip list data structure will be significantly unbalanced (e.g., for a dictionary of more than 250 elements, the chance that a search will take more than 3 times the expected time is less than one in a million). Skip lists have balance properties similar to that of search trees built by random insertions, yet do not require insertions to be random.

Balancing a data structure probabilistically is easier than explicitly maintaining the balance. For many applications, skip lists are a more natural representation than trees, also leading to simpler algorithms. The simplicity of skip list algorithms makes them easier to implement and provides significant constant factor speed improvements over balanced tree and self-adjusting tree algorithms. Skip lists are also very space efficient. They can easily be configured to require an average of $1\ ^1/_3$ pointers per element (or even less) and do not require balance or priority information to be stored with each node.

SKIP LISTS

We might need to examine every node of the list when searching a linked list (*Figure* 1*a*). If the list is stored in sorted order and every other node of the list also has a pointer to the node two ahead it in the list (*Figure* 1*b*), we have to examine no more than $\lceil n/2 \rceil + 1$ nodes (where n is the length of the list). Also giving every fourth node a pointer four ahead (*Figure* 1*c*) requires that no more than $\lceil n/4 \rceil + 2$ nodes be examined. If every $(2^i)^{\text{th}}$ node has a pointer 2^i nodes ahead (*Figure* 1*d*), the number of nodes that must be examined can be reduced to $\lceil \log_2 n \rceil$ while only doubling the number of pointers. This data structure could be used for fast searching, but insertion and deletion would be impractical.

A node that has k forward pointers is called a *level* k node. If every $(2^i)^{\text{th}}$ node has a pointer 2^i nodes ahead, then levels of nodes are distributed in a simple pattern: 50% are level 1, 25% are level 2, 12.5% are level 3 and so on. What would happen if the levels of nodes were chosen randomly, but in the same proportions (e.g., as in Figure 1*e*)? A node's i^{th} forward pointer, instead of pointing 2^{i-1} nodes ahead, points to the next node of level i or higher. Insertions or deletions would require only local modifications; the level of a node, chosen randomly when the node is inserted, need never change. Some arrangements of levels would give poor execution times, but we will see that such arrangements are rare. Because these data structures are linked lists with extra pointers that skip over intermediate nodes, I named them *skip lists*.

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). The *Delete* operation deletes the specified key. It is easy to support additional operations such as "find the minimum key" or "find the next 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.

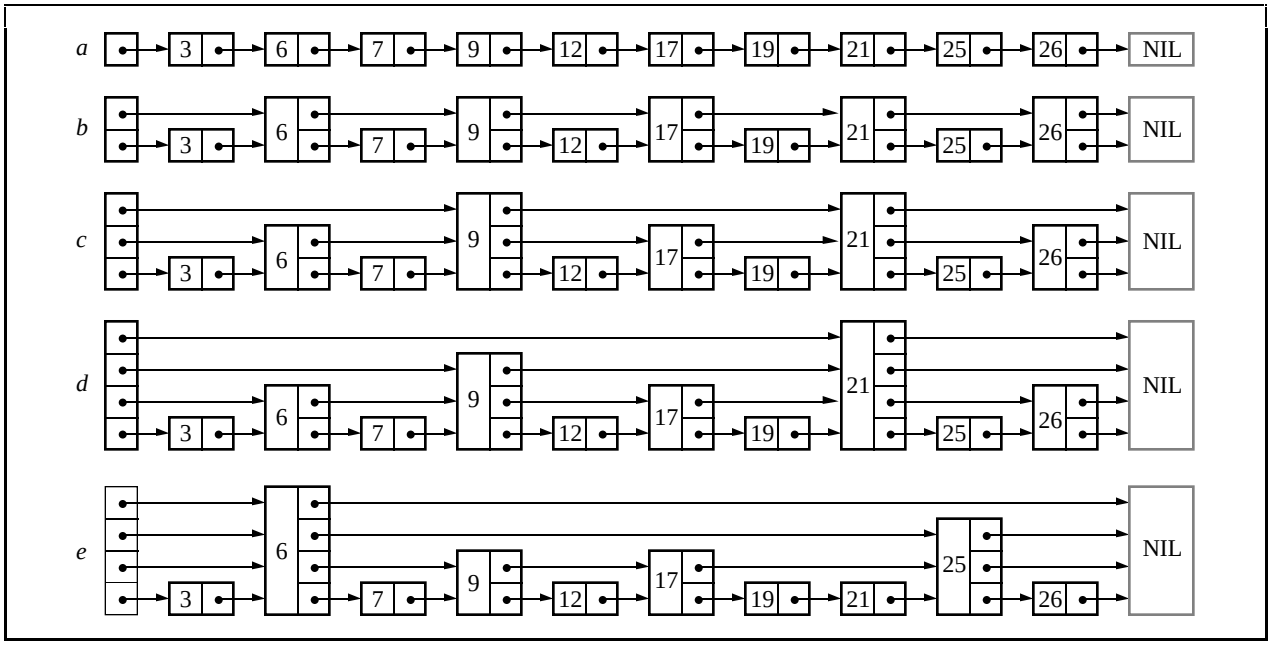

FIGURE 1 - Linked lists with additional pointers

Initialization

An element NIL is allocated and 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 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).

Insertion and Deletion Algorithms

To insert or delete a node, we simply search and splice, as shown in Figure 3. Figure 4 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.

```
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

Choosing 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 5. 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? If we use the standard algorithm and start our search at level 14, we will do a lot of useless work.

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 $\log_{1/p} n$.

There are a number of solutions to the problem of deciding how to handle the case where there is an element with an unusually large level in the list.

- *Don't worry, be happy.* Simply start a search at the highest level present in the list. As we will see in our analysis, 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 in this paper.

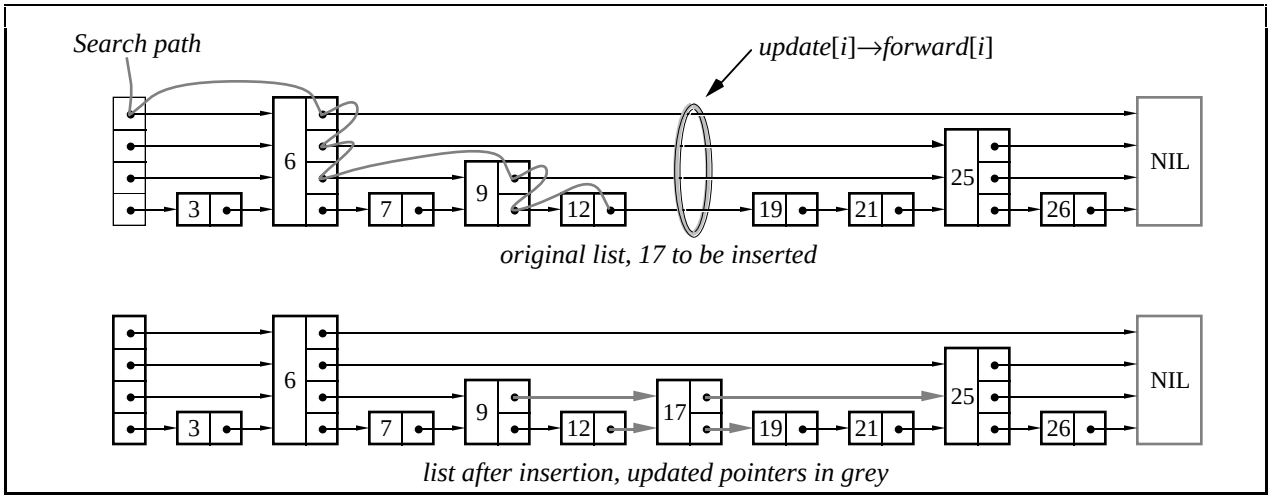

FIGURE 3 - Pictorial description of steps involved in performing an insertion

- *Use less than you are given.* Although an element may contain room for 14 pointers, we don't need to use all 14. We can choose to utilize only $L(n)$ levels. There are a number of ways to implement this, but they all complicate the algorithms and do not noticeably improve performance, so this approach is not recommended.
- *Fix the dice.* If we generate a random level that is more than one greater than the current maximum level in the list, we simply use one plus the current maximum level in the list as the level of the new node. In practice and intuitively, this change seems to work well. However, it totally destroys our ability to analyze the resulting algorithms, since the level of a node is no longer completely random. Programmers should probably feel free to implement this, purists should avoid it.

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 $MaxLevel = 16$ is appropriate for data structures containing up to 2^{16} elements.

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. For the *Insert* and *Delete* operations, there is an additional cost proportional to the level of the node being inserted or deleted. The time required to find an element is proportional to the length of the search path, which is determined by the pattern in which elements with different levels appear as we traverse the list.

Probabilistic Philosophy

The structure of a skip list is determined only by the number

```
randomLevel()
    lvl := 1
    -- random() that returns a random value in [0...1)
    while random() < p and lvl < MaxLevel do
        lvl := lvl + 1
    return lvl
```

FIGURE 5 - Algorithm to calculate a random level

```
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]
        -- x→key < searchKey ≤ x→forward[i]→key
        update[i] := x
    x := x→forward[1]
    if x→key = searchKey then x→value := newValue
    else
        lvl := randomLevel()
        if lvl > list→level then
            for i := list→level + 1 to lvl do
                update[i] := list→header
            list→level := lvl
        x := makeNode(lvl, searchKey, value)
        for i := 1 to level 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 4 - Skip List insertion and deletion algorithms

elements in the skip list and the results of consulting the random number generator. The sequence of operations that produced the current skip list does not matter. We assume an adversarial user does not have access to the levels of nodes; otherwise, he could create situations with worst-case running times by deleting all nodes that were not level 1.

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. More will be said about this later.

Analysis of expected search cost

We analyze the search path backwards, travelling up and to the left. Although the levels of nodes in the list are known and fixed when the search is performed, we act as if the level of a node is being determined only when it is observed while backtracking the search path.

At any particular point in the climb, we are at a situation similar to situation *a* in Figure 6 – we are at the i^{th} forward pointer of a node *x* and we have no knowledge about the levels of nodes to the left of *x* or about the level of *x*, other than that the level of *x* must be at least *i*. Assume the *x* is not the header (the is equivalent to assuming the list extends infinitely to the left). If the level of *x* is equal to *i*, then we are in situation *b*. If the level of *x* is greater than *i*, then we are in situation *c*. The probability that we are in situation *c* is *p*. Each time we are in situation *c*, we climb up a level. Let $C(k)$ = the expected cost (i.e, length) of a search path that climbs up *k* levels in an infinite list:

$C(0) = 0$
$C(k) = (1–p)$ (cost in situation *b*) $+ p$ (cost in situation *c*)

By substituting and simplifying, we get:

$C(k) = (1–p)\,(1 + C(k)) + p\,(1 + C(k–1))$
$C(k) = 1/p + C(k–1)$
$C(k) = k/p$

Our assumption that the list is infinite is a pessimistic assumption. When we bump into the header in our backwards climb, we simply climb up it, without performing any leftward movements. This gives us an upper bound of $(L(n)–1)/p$ on the expected length of the path that climbs from level 1 to level $L(n)$ in a list of *n* elements.

We use this analysis go up to level $L(n)$ and use a different analysis technique for the rest of the journey. The number of leftward movements remaining is bounded by the number of elements of level $L(n)$ or higher in the entire list, which has an expected value of $1/p$.

We also move upwards from level $L(n)$ to the maximum level in the list. The probability that the maximum level of the list is a greater than *k* is equal to $1–(1–p^k)^n$, which is at most np^k. We can calculate the expected maximum level is at most $L(n) + 1/(1–p)$. Putting our results together, we find

Total expected cost to climb out of a list of *n* elements
$\le L(n)/p + 1/(1–p)$

which is $O(\log n)$.

Number of comparisons

Our result is an analysis of the "length" of the search path. The number of comparisons required is one plus the length of the search path (a comparison is performed for each position in the search path, the "length" of the search path is the number of hops between positions in the search path).

Probabilistic Analysis

It is also possible to analyze the probability distribution of search costs. The probabilistic analysis is somewhat more complicated (see box). From the probabilistic analysis, we can calculate an upper bound on the probability that the actual cost of a search exceeds the expected cost by more than a specified ratio. Some results of this analysis are shown in Figure 8.

Choosing *p*

Table 1 gives the relative times and space requirements for different values of *p*. Decreasing *p* also increases the variabil-

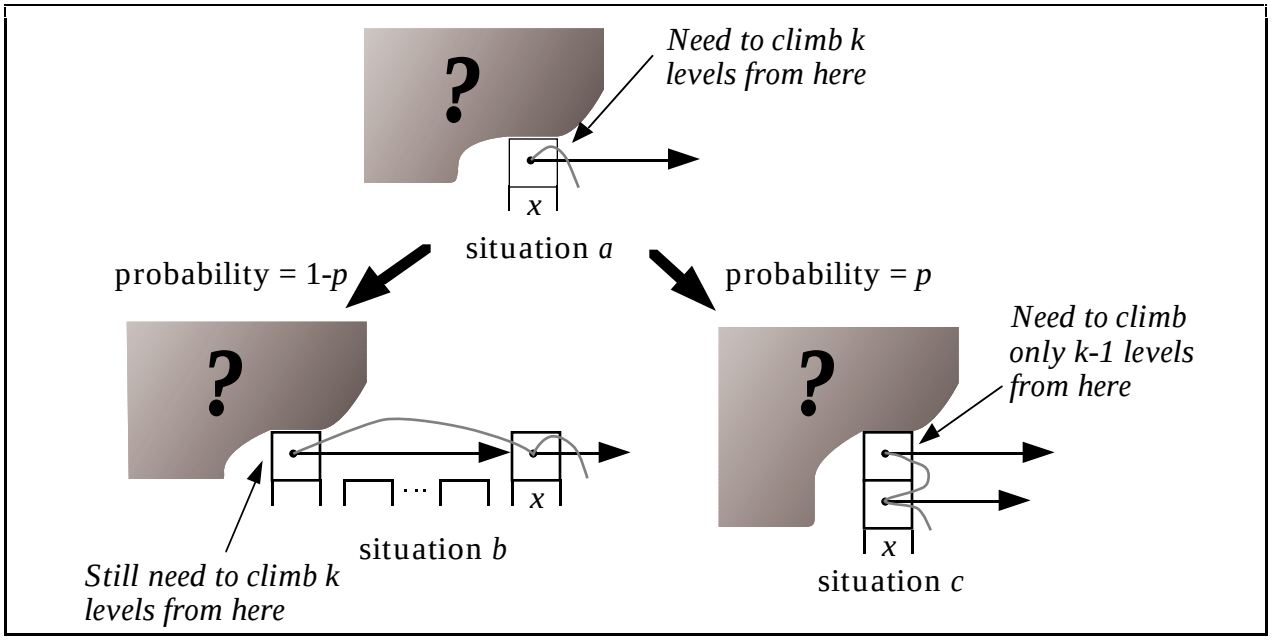

FIGURE 6 - Possible situations in backwards traversal of the search path

ity of running times. If $1/p$ is a power of 2, it will be easy to generate a random level from a stream of random bits (it requires an average of $(\log_2 1/p)/(1-p)$ random bits to generate a random level). Since some of the constant overheads are related to $L(n)$ (rather than $L(n)/p$), choosing $p = 1/4$ (rather than 1/2) slightly improves the constant factors of the speed of the algorithms as well. I suggest that a value of 1/4 be used for p unless the variability of running times is a primary concern, in which case p should be 1/2.

Sequences of operations

The expected total time for a sequence of operations is equal to the sum of the expected times of each of the operations in the sequence. Thus, the expected time for any sequence of m searches in a data structure that contains n elements is $O(m \log n)$. However, the pattern of searches affects the probability distribution of the actual time to perform the entire sequence of operations.

If we search for the same item twice in the same data structure, both searches will take exactly the same amount of time. Thus the variance of the total time will be four times the variance of a single search. If the search times for two elements are independent, the variance of the total time is equal to the sum of the variances of the individual searches. Searching for the same element over and over again maximizes the variance.

ALTERNATIVE DATA STRUCTURES

Balanced trees (e.g., AVL trees [Knu73] [Wir76]) and self-adjusting trees [ST85] can be used for the same problems as skip lists. All three techniques have performance bounds of the same order. A choice among these schemes involves several factors: the difficulty of implementing the algorithms, constant factors, type of bound (amortized, probabilistic or worst-case) and performance on a non-uniform distribution of queries.

Implementation difficulty

For most applications, implementers generally agree skip lists are significantly easier to implement than either balanced tree algorithms or self-adjusting tree algorithms.

p	Normalized search times (*i.e.*, normalized $L(n)/p$)	Avg. # of pointers per node (*i.e.*, $1/(1-p)$)
$1/2$	1	2
$1/e$	0.94...	1.58...
$1/4$	1	1.33...
$1/8$	1.33...	1.14...
$1/16$	2	1.07...

TABLE 1 – Relative search speed and space requirements, depending on the value of p.

Constant factors

Constant factors can make a significant difference in the practical application of an algorithm. This is particularly true for sub-linear algorithms. For example, assume that algorithms A and B both require $O(\log n)$ time to process a query, but that B is twice as fast as A: in the time algorithm A takes to process a query on a data set of size n, algorithm B can process a query on a data set of size n^2.

There are two important but qualitatively different contributions to the constant factors of an algorithm. First, the inherent complexity of the algorithm places a lower bound on any implementation. Self-adjusting trees are continuously rearranged as searches are performed; this imposes a significant overhead on any implementation of self-adjusting trees. Skip list algorithms seem to have very low inherent constant-factor overheads: the inner loop of the deletion algorithm for skip lists compiles to just six instructions on the 68020.

Second, if the algorithm is complex, programmers are deterred from implementing optimizations. For example, balanced tree algorithms are normally described using recursive insert and delete procedures, since that is the most simple and intuitive method of describing the algorithms. A recursive insert or delete procedure incurs a procedure call overhead. By using non-recursive insert and delete procedures, some of this overhead can be eliminated. However, the complexity of non-recursive algorithms for insertion and deletion in a balanced tree is intimidating and this complexity deters most programmers from eliminating recursion in these routines. Skip list al-

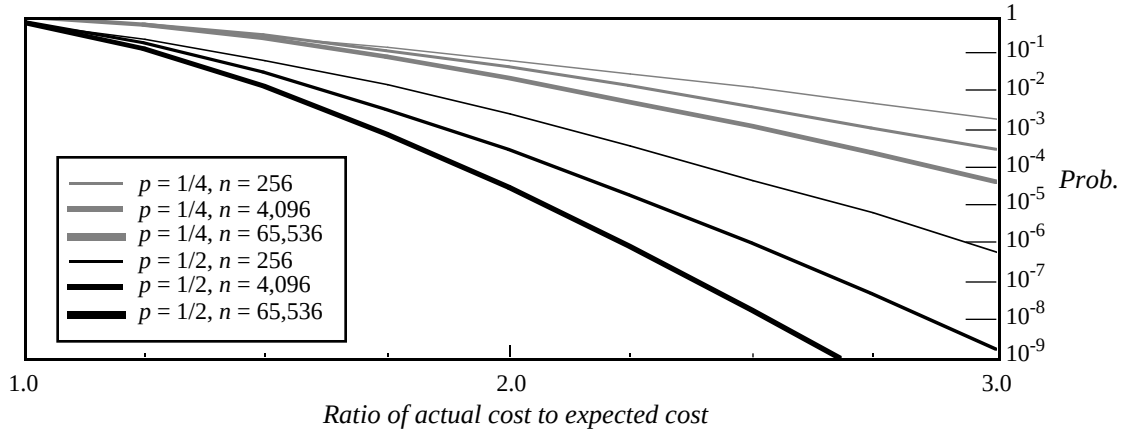

FIGURE 8 - This graph shows a plot of an upper bound on the probability of a search taking substantially longer than expected. The vertical axis show the probability that the length of the search path for a search exceeds the average length by more than the ratio on the horizontal axis. For example, for $p = 1/2$ and $n = 4096$, the probability that the search path will be more than three times the expected length is less than one in 200 million. This graph was calculated using our probabilistic upper bound.

Implementation	Search Time	Insertion Time	Deletion Time
Skip lists	0.051 msec (1.0)	0.065 msec (1.0)	0.059 msec (1.0)
non-recursive AVL trees	0.046 msec (0.91)	0.10 msec (1.55)	0.085 msec (1.46)
recursive 2–3 trees	0.054 msec (1.05)	0.21 msec (3.2)	0.21 msec (3.65)
Self–adjusting trees:			
top-down splaying	0.15 msec (3.0)	0.16 msec (2.5)	0.18 msec (3.1)
bottom-up splaying	0.49 msec (9.6)	0.51 msec (7.8)	0.53 msec (9.0)

Table 2 - Timings of implementations of different algorithms

gorithms are already non-recursive and they are simple enough that programmers are not deterred from performing optimizations.

Table 2 compares the performance of implementations of skip lists and four other techniques. All implementations were optimized for efficiency. The AVL tree algorithms were written by James Macropol of Contel and based on those in [Wir76]. The 2–3 tree algorithms are based on those presented in [AHU83]. Several other existing balanced tree packages were timed and found to be much slower than the results presented below. The self-adjusting tree algorithms are based on those presented in [ST85]. The times in this table reflect the CPU time on a Sun-3/60 to perform an operation in a data structure containing 2^{16} elements with integer keys. The values in parenthesis show the results relative to the skip list time The times for insertion and deletion do not include the time for memory management (e.g, in *C* programs, calls to *malloc* and *free*).

Note that skip lists perform more comparisons than other methods (the skip list algorithms presented here require an average of $L(n)/p + 1/(1–p) + 1$ comparisons). For tests using real numbers as keys, skip lists were slightly slower than the non-recursive AVL tree algorithms and search in a skip list was slightly slower than search in a 2–3 tree (insertion and deletion using the skip list algorithms was still faster than using the recursive 2–3 tree algorithms). If comparisons are very expensive, it is possible to change the algorithms so that we never compare the search key against the key of a node more than once during a search. For $p = 1/2$, this produces an upper bound on the expected number of comparisons of $7/2 + 3/2 \log_2 n$. This modification is discussed in [Pug89b].

Type of performance bound

These three classes of algorithm have different kinds of performance bounds. Balanced trees have worst-case time bounds, self-adjusting trees have amortized time bounds and skip lists have probabilistic time bounds. With self-adjusting trees, an individual operation can take $O(n)$ time, but the time bound always holds over a long sequence of operations. For skip lists, any operation or sequence of operations can take longer than expected, although the probability of any operation taking significantly longer than expected is negligible.

In certain real-time applications, we must be assured that an operation will complete within a certain time bound. For such applications, self-adjusting trees may be undesirable, since they can take significantly longer on an individual operation than expected (e.g., an individual search can take $O(n)$ time instead of $O(\log n)$ time). For real-time systems, skip lists may be usable if an adequate safety margin is provided: the chance that a search in a skip lists containing 1000 elements takes more than 5 times the expected time is about 1 in 10^{18}.

Non-uniform query distribution

Self-adjusting trees have the property that they adjust to non-uniform query distributions. Since skip lists are faster than self-adjusting trees by a significant constant factor when a uniform query distribution is encountered, self-adjusting trees are faster than skip lists only for highly skewed distributions. We could attempt to devise self-adjusting skip lists. However, there seems little practical motivation to tamper with the simplicity and fast performance of skip lists; in an application where highly skewed distributions are expected, either self-adjusting trees or a skip list augmented by a cache may be preferable [Pug90].

ADDITIONAL WORK ON SKIP LISTS

I have described a set of algorithms that allow multiple processors to concurrently update a skip list in shared memory [Pug89a]. This algorithms are much simpler than concurrent balanced tree algorithms. They allow an unlimited number of readers and *n* busy writers in a skip list of *n* elements with very little lock contention.

Using skip lists, it is easy to do most (all?) the sorts of operations you might wish to do with a balanced tree such as use search fingers, merge skip lists and allow ranking operations (e.g., determine the k^{th} element of a skip list) [Pug89b].

Tom Papadakis, Ian Munro and Patricio Poblette [PMP90] have done an exact analysis of the expected search time in a skip list. The upper bound described in this paper is close to their exact bound; the techniques they needed to use to derive an exact analysis are very complicated and sophisticated. Their exact analysis shows that for $p = 1/2$ and $p = 1/4$, the upper bound given in this paper on the expected cost of a search is not more than 2 comparisons more than the exact expected cost.

I have adapted idea of probabilistic balancing to some other problems arising both in data structures and in incremental computation [PT88]. We can generate the level of a node based on the result of applying a hash function to the element (as opposed to using a random number generator). This results in a scheme where for any set *S*, there is a unique data structure that represents *S* and with high probability the data structure is approximately balanced. If we combine this idea with an *applicative* (i.e., persistent) probabilistically balanced data structure and a scheme such as hashed-consing [All78] which allows constant-time structural equality tests of applicative data structures, we get a number of interesting properties, such as constant-time equality tests for the representations of sequences. This scheme also has a number of applications for incremental computation. Since skip lists are

somewhat awkward to make applicative, a probabilistically balanced *tree* scheme is used.

RELATED WORK

James Discroll pointed out that R. Sprugnoli suggested a method of randomly balancing search trees in 1981 [Spr81]. With Sprugnoli's approach, the state of the data structure is *not* independent of the sequence of operations which built it. This makes it much harder or impossible to formally analyze his algorithms. Sprugnoli gives empirical evidence that his algorithm has good expected performance, but no theoretical results.

A randomized data structure for ordered sets is described in [BLLSS86]. However, a search using that data structure requires $O(n^{1/2})$ expected time.

Cecilia Aragon and Raimund Seidel describe a probabilistically balanced search trees scheme [AC89]. They discuss how to adapt their data structure to non-uniform query distributions.

SOURCE CODE AVAILABILITY

Skip list source code libraries for both C and Pascal are available for anonymous ftp from `mimsy.umd.edu`.

CONCLUSIONS

From a theoretical point of view, there is no need for skip lists. Balanced trees can do everything that can be done with skip lists and have good worst-case time bounds (unlike skip lists). However, implementing balanced trees is an exacting task and as a result balanced tree algorithms are rarely implemented except as part of a programming assignment in a data structures class.

Skip lists are a simple data structure that can be used in place of balanced trees for most applications. Skip lists algorithms are very easy to implement, extend and modify. Skip lists are about as fast as highly optimized balanced tree algorithms and are substantially faster than casually implemented balanced tree algorithms.

ACKNOWLEDGEMENTS

Thanks to the referees for their helpful comments. Special thanks to all those people who supplied enthusiasm and encouragement during the years in which I struggled to get this work published, especially Alan Demers, Tim Teitelbaum and Doug McIlroy. This work was partially supported by an AT&T Bell Labs Fellowship and by NSF grant CCR–8908900.

REFERENCES

[AC89] Aragon, Cecilia and Raimund Seidel, Randomized Search Trees, *Proceedings of the 30th Ann. IEEE Symp on Foundations of Computer Science*, pp 540–545, October 1989.

[AHU83] Aho, A., Hopcroft, J. and Ullman, J. *Data Structures and Algorithms*, Addison-Wesley Publishing Company, 1983.

[All78] John Allen. *Anatomy of LISP*, McGraw Hill Book Company, NY, 1978.

[BLLSS86] Bentley, J., F. T. Leighton, M.F. Lepley, D. Stanat and J. M. Steele, *A Randomized Data Structure For Ordered Sets*, MIT/LCS Technical Memo 297, May 1986.

[Knu73] Knuth, D. "Sorting and Searching," *The Art of Computer Programming*, Vol. 3, Addison-Wesley Publishing Company, 1973.

[PMP90] Papadakis, Thomas, Ian Munro and Patricio Poblette, *Exact Analysis of Expected Search Cost in Skip Lists*, Tech Report # ????, Dept. of Computer Science, Univ. of Waterloo, January 1990.

[PT89] Pugh, W. and T. Teitelbaum, "Incremental Computation via Function Caching," *Proc. of the Sixteenth conference on the Principles of Programming Languages*, 1989.

[Pug89a] Pugh, W., *Concurrent Maintenance of Skip Lists*, Tech Report TR-CS-2222, Dept. of Computer Science, University of Maryland, College Park, 1989.

[Pug89b] Pugh, W., *Whatever you might want to do using Balanced Trees, you can do it faster and more simply using Skip Lists*, Tech Report CS–TR–2286, Dept. of Computer Science, University of Maryland, College Park, July 1989.

[Pug90] Pugh, W. Slow Optimally Balanced Search Strategies vs. Cached Fast Uniformly Balanced Search Strategies, to appear in *Information Processing Letters*.

[Spr81] Sprugnoli, R. Randomly Balanced Binary Trees, *Calcolo*, V17 (1981), pp 99-117.

[ST85] Sleator, D. and R. Tarjan "Self-Adjusting Binary Search Trees," *Journal of the ACM*, Vol 32, No. 3, July 1985, pp. 652-666.

[Wir76] Wirth, N. *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

PROBABILISTIC ANALYSIS

In addition to analyzing the expected performance of skip lists, we can also 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 a 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 $\leq_{prob}$). 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 $X \leq_{prob} Y$ 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
$X \leq_{prob} Y$ iff $\forall\ t$, Prob{ $X > t$ } $\leq$ Prob{ $Y > t$ }. □

For example, the graph in Figure 7shows the probability distribution of three random variables X, Y and Z. Since the probability distribution curve for X is completely under the curves for Y and Z, $X \leq_{prob} Y$ and $X \leq_{prob} Z$. Since the probability curves for Y and Z intersect, neither $Y \leq_{prob} Z$ nor $Z \leq_{prob} Y$. Since the expected value of a random variable X is simply the area under the curve Prob{ $X > t$ }, if $X \leq_{prob} Y$ then the average of X is less than or equal to the average of Y.

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. □

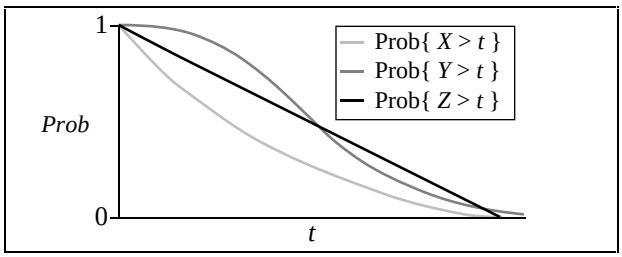

FIGURE 7 – Plots of three probability distributions

Probabilistic analysis of search cost

The number of leftward movements we need to make before we move up a level (in an infinite list) has a negative binomial distribution: it is the number of failures (situations *b*'s) we see before we see the first success (situation *c*) in a series of independent random trials, where the probability of success is p. Using the probabilistic notation introduced above:

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

We can sum the costs of climbing each level to get the total cost to climb up to level $L(n)$:

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:

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 an 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):

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

The expected value of our upper bound is equal to

$(L(n) - 1) + (L(n) - 1)(1 - p)/p + 1/p + p/(1–p) + 1$
$= L(n)/p + 1/(1–p)$,

which is the same as our previously calculated upper bound on the expected cost of a search. The variance of our upper bound is

$(L(n) - 1)(1–p)/p^2 + (1 - 1/np)/p + p/(1–p)^2$
$< (1–p)L(n)/p^2 + p/(1–p)^2 + (2p–1)/p^2$.

Figure 8 show a plot of an upper bound on the probability of an actual search taking substantially longer than average, based on our probabilistic upper bound.