CS-TR-2222.1 APRIL, 1989
(REVISED JUNE, 1990)

CONCURRENT MAINTENANCE OF SKIP LISTS

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

Abstract

This paper describes a new approach to providing efficient concurrent access to a dynamic search structure. Previous approaches have attempted to solve this problem using search trees (either balanced or unbalanced). We describe methods for performing concurrent access and updates using *skip lists*. Skip lists are a probabilistic alternative to balanced trees that provide much of the simplicity of unbalanced trees, together with good worst-case expected performance. In this paper, we briefly review skip lists, describe simple methods for concurrent maintenance of sorted linked lists, formally prove the correctness of those methods, and show how they can be extended to provide simple and efficient algorithms for concurrent maintenance of skip lists.

CR Categories and Subject Descriptors: D.4.1 [**Process Management**]: Concurrency; 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. Introduction

Many papers have been written on implementing concurrent search structures using search trees, both balanced [Ell80a] [Ell80b] and unbalanced [KL80] [ML84] [Man84]. In concurrent search structures, locks are used to prevent concurrent threads from interfering with each other. A concurrency scheme must assure the integrity of the data structure, avoid deadlock and have a serializable schedule. Within those restrictions, we would like the algorithms to be as simple, efficient and concurrent as possible.

Performing concurrent deletion or rebalancing in trees is very complicated; most papers on concurrent tree maintenance have omitted describing techniques for one or the other of these problems, mainly due to the complexity of these algorithms.

In this paper we describe simple concurrent algorithms for access and update of skip lists. Skip lists [Pug89] are a new probabilistic data structure that can be used as a replacement for balanced trees. The concurrent algorithms for updating skip lists are much simpler than equivalent concurrent algorithms for updating balanced trees and allow more concurrency. The algorithms use only write locks; no exclusive locks or read locks are required.

The rest of this paper is organized as follows. In Section 2 we describe a simple concurrent updating scheme for sorted linked lists. In Section 3 we briefly review skip lists. In Section 4, we show how the concurrent updating scheme for linked lists can be adapted to skip lists and discuss the efficiency and allowable concurrency of our scheme. In Section 5, we discuss related work.

2. Concurrent Maintenance of Sorted Linked Lists

We now describe a method for performing concurrent updates of sorted linked lists. The elements of the list are keys with associated values. Each element is represented by a node. The forward pointer of a node points to the next node in the list and the nodes are kept in sorted order according their keys. The key of a node x is given by *x→key*, the value is given by *x→value* and the forward pointer is given by *x→forward*. The header of a list l is treated as a node and is given by *l→header*. For purposes of reasoning about our invariants, the header is assumed to have a key smaller than that of any node. The list is terminated by a special node NIL that has a key larger than that of any node.

We use the term thread to refer to a task or process operating concurrently with other threads. A thread obtains a lock on a field only when updating a field that other threads might be attempting to update. While searching for an element, no locks are needed. Only a single thread may hold a lock on a field, and by convention a thread only updates a field if it already holds a lock on the field. If a thread attempts to lock a field that is already locked, that thread is blocked until the lock can be obtained.

We assume that pointer and integer reads are atomic with respect to updates of that same pointer or integer. That is, reading a pointer always returns a valid pointer, not something midway between an old pointer value and a new pointer value. Without this assumption, read locks would be needed whenever any thread attempted to read a field.

2.1. Algorithms

The novel idea in our method is the use of pointer reversal. When deleting a node x, we update the predecessor of x to point to the successor of x, in the standard way. But we also update x itself so as to point back to its former predecessor. Other threads that were passing through x at the time of its deletion can follow this reversed pointed to get back into the current list, at the correct place to continue their work.

The algorithms are shown in Figures 1 and 2. The algorithms are actually very simple, but are annotated with a substantial amount of notation and invariants (explained in Section 2.2) to allow us to formally prove the correctness of the algorithms – for now, the reader can just ignore the annotations. The statement *lock(x, f)* acquires a lock on the field *f* of the record pointed to by x. The *Search* routine returns the value associated with the provided key or returns *not-found*. The result returned is guaranteed to have been true at some point during the execution of the search. *WeakSearch* and *StrongSearch* are utility routines. *WeakSearch* does not use any locks and *StrongSearch* returns a locked result.

```
Search(list, searchKey)
    (x, y) := WeakSearch(list, searchKey)
    if y→key = searchKey then return y→value
        else return not-found

WeakSearch(list, searchKey)
    x := list→header
    y := x→forward
    while y→key < searchKey do
        x := y
        y := x→forward
    -- wasTrue(self, x ➠ y) ∧ x→key < searchKey ≤ y→key ,
        ∴ wasTrue(self, y→key = searchKey ⇔ present(searchKey)) (via Theorem 3)
    -- wasTrue(self, y→key = searchKey ⇔ present(searchKey)),
        ∴ (y→key = searchKey ⇒ wasTrue(self, present(searchKey)))
            ∧ (y→key ≠ searchKey ⇒ wasTrue(self, ¬ present(searchKey)))
    return x, y

StrongSearch(list, searchKey)
    (x, y) := WeakSearch(list, searchKey) -- do as much work as possible without using locks
    lock(x, forward)
    y := x→forward -- the forward pointer of x may have changed since the completion of the weak search
    while x→key < searchKey do
        unlock(x, forward)
        x := y
        lock(x, forward)
        y := x→forward
    -- holdsLock(self, x, forward) ∧ x ➠ y ∧ x→key < searchKey ≤ y→key
    -- holdsLock(self, x, forward) ∧ ¬ deleting(self), ∴ ¬ beingDeleted(x) (via P4(x))
    -- ¬ beingDeleted(x) ∧ x ➠ y ∧ x→key < y→key , ∴ reachable(x) (via P1(x))
    -- x ➠ y ∧ x→key < searchKey ≤ y→key, ∴ y→key = searchKey ⇔ present(searchKey) (via Theorem 3)
    return x, y
```

FIGURE 1 - Concurrent search and lock utilities for sorted linked lists

When we delete a node y, we cannot immediately garbage collect y because other threads may have a pointer to y. Instead, *putOnGarbageQueue(y)* puts y onto a garbage queue. A node can be taken off the garbage queue any time after the completion of all searches/insertions/deletions that were in progress when the node was put on the queue. For the purposes of showing the correctness of the concurrent algorithms, we can let this be a no-op. This idea is similar to the garbage collection and maintenance threads used by other concurrent algorithms [KT80] [ML84] [Man84]. Algorithms for concurrent search structures that do not need delayed garbage collection force all threads to use read locks whenever they access a node.

2.2. Correctness

A formal definition of these routines is provided by their post-conditions (note that their post-conditions cannot be invalidated by a concurrent insertion/deletion). We show that the algorithms are correct and cannot be interfered with if the data structure invariants are maintained. We also show that no algorithm invalidates the data structure invariants. The proof of correctness of the algorithms is provided as annotations in Figures 1–2. Some of the notation we use in our proofs and invariants are shown below.

$\exists(x \parallel P)$	There exists a x such that P holds (where x is free in P).
$\forall(x \parallel R)$	For all x, R holds.
$\forall(x \parallel P \parallel R)$	For all x such that P holds, R holds.

```
Insert(list, searchKey, newValue)
    (x, y) := StrongSearch(list, searchKey)
    if y→key = searchKey then
        y→value := newValue -- update occurs now
    else
        y := newNode()
        -- Initially, we do not enforce invariants on y. This is safe, since y is not visible to other threads.
        y→key := searchKey
        y→value := newValue
        -- x→key < y→key ≤ x→forward→key
        y→forward := x→forward
        x→forward := y -- insertion occurs now.
        -- From now on, y is visible to other threads and invariants are enforced
        -- reachable(x) ∧ y→key < x→forward→key, ∴ P1(x)
        -- reachable(y) ∧ y→key ≤ y→forward→key, ∴ P1(y)
    unlock(x, forward)

Delete(list, searchKey)
    (x, y) := StrongSearch(list, searchKey)
    if x→key = searchKey then
        lock(y, forward)
        -- x→key < y→key ≤ y→forward→key
        -- deleting(self) is true only between the next two statements
        x→forward := y→forward -- deletion occurs now
        -- reachable(x) ∧ x→key ≤ x→forward→key, ∴ P1(x)
        -- beingDeleted(y) ∧ y→key ≤ y→forward→key, ∴ P1(y)
        -- reachable(y→forward), ∴ P2(y)
        -- reachable(x) ∧ Sorted, ∴ ∀(k || x→key < k < x→forward→key) || ¬ present(k))
        -- x→key < y→key ∧ x→forward = y→forward ∧ ∀(k || x→key < k < x→forward→key || ¬ present(k)),
            ∴∀(k || y→key < k < y→forward→key) || ¬ present(k))
            ∴P3(y)
        -- holdsLock(self, y, forward), ∴∀(t || holdsLock(t, y) ⇒ t = self) (via semantics of locks)
        -- ∀(t || holdsLock(t, y, forward) ⇒ t = self) ∧ deleting(self), ∴P4(y)
        y→forward := x
        -- ¬ reachable(y) ∧ y→key> y→forward→key, ∴ P1(y)
        unlock(y, forward)
        putOnGarbageQueue(y)
    unlock(x, forward)
```

FIGURE 2 – Concurrent insertion and deletion in a sorted linked list

$x \Rightarrow z$	True iff the forward pointer of x is equal to z. $x \Rightarrow z \Leftrightarrow x \to forward = z$
$x \Rightarrow^{+} z$	True iff we can get from x to z by following one or more forward pointers. $x \Rightarrow^{+} z \Leftrightarrow x \Rightarrow z \vee \exists(y \parallel x \Rightarrow y \wedge y \Rightarrow^{+} z)$
$reachable(x)$	True iff the node x is reachable from the header of the list. $reachable(x) \Leftrightarrow list \to header \Rightarrow^{+} x$
$present(searchKey)$	True iff *searchKey* is present in the list. $present(searchKey) \Leftrightarrow \exists(y \parallel y \to key = searchKey \wedge reachable(y))$
$downstream(searchKey, x)$	True iff *searchKey* is present at some point downstream from x. $downstream(searchKey, x) \Leftrightarrow \exists(y \parallel y \to key = searchKey \wedge x \Rightarrow^{+} y)$

self	Used in an annotation of the execution of an algorithm, refers to the thread executing the algorithm.
$holdsLock(t, x, f)$	True iff thread t holds a lock on field f of x.
$wasTrue(t, P)$	True iff P was true at some point since execution of thread t began. Note that is possible for both $wasTrue(t, P)$ and $wasTrue(t, \neg P)$ to be true at the same time.
$deleting(t)$	True iff execution of thread t is between the two marked statements of the deletion algorithm.
y_t	The local variable y of thread t.
$beingDeleted(x)$	True iff some thread is in the process of deleting x. $beingDeleted(x) \Leftrightarrow \exists(t \parallel deleting(t) \wedge y_t = x)$
$deleted(x)$	True iff x is neither reachable nor in the process of being deleted. $deleted(x) \Leftrightarrow \neg\, (reachable(x) \vee beingDeleted(x))$
$P, \therefore Q$	P is true, P implies Q, and therefore Q is true.

We enforce six invariants on our data structure:

$\forall(x \parallel P1(x)),\ P1(x) \equiv deleted(x) \Leftrightarrow x\rightarrow key > x\rightarrow forward\rightarrow key$
$\forall(x \parallel P2(x)),\ P2(x) \equiv \neg\, deleted(x) \Rightarrow reachable(x\rightarrow forward)$
$\forall(x \parallel P3(x)),\ P3(x) \equiv \forall(k \parallel beingDeleted(x) \wedge x\rightarrow key < k < x\rightarrow forward\rightarrow key \parallel \neg\, present(k)\ )$
$\forall(x \parallel P4(x)),\ P4(x) \equiv \forall(t \parallel holdsLock(t, x, forward) \wedge \neg\, deleting(t) \Rightarrow \neg\, beingDeleted(x)\ )$
$\forall(x \parallel R(x)),\ R(x) \equiv \forall(k \parallel k > x\rightarrow key \wedge present(k) \parallel downstream(k, x)\ )$
Sorted: $\forall(\ x, y \parallel reachable(x) \wedge reachable(y) \wedge x\rightarrow key < y\rightarrow key \parallel x \twoheadrightarrow^{+} y)$

The last two invariants are global invariants, so it is hard to verify directly that we are enforcing them. Fortunately, we can infer them from the first three invariants.

When we initially allocate a node, its fields are not yet initialized and our invariants obviously do not apply. The invariants are enforced as soon as the node is inserted into the list, which is the moment the node becomes visible to threads besides the one performing the insertion.

Three theorems we need for our proofs are given in this section.

THEOREM 1. $\forall(x \parallel P1(x)) \Rightarrow Sorted$

Proof: By induction on the number of pointers that need to be traversed between reachable nodes.

THEOREM 2: $\forall(x \parallel P1(x) \wedge P2(x) \wedge P3(x)) \Rightarrow \forall(x \parallel R(x))$

Proof: Given below.

		Why?
1.	$reachable(x) \wedge Sorted \Rightarrow R(x)$	*definition*
2.	$reachable(x) \Rightarrow R(x)$	1, *Theorem 1*
3.	$beingDeleted(x) \Rightarrow reachable(x\rightarrow forward)$	$P2(x)$
4.	$beingDeleted(x) \Rightarrow R(x\rightarrow forward)$	3, 2
5.	$beingDeleted(x) \Rightarrow \forall(k \parallel x\rightarrow key < k < x\rightarrow forward\rightarrow key \parallel \neg\, present(k)\ )$	$P3(x)$
6.	$\forall(k \parallel x\rightarrow key < k < x\rightarrow forward\rightarrow key \parallel \neg\, present(k)\ ) \wedge R(x\rightarrow forward) \Rightarrow R(x)$	*definition*
7.	$beingDeleted(x) \Rightarrow R(x)$	4, 5, 6
8.	$deleted(x) \Rightarrow x\rightarrow key > x\rightarrow forward\rightarrow key$	$P1(x)$
9.	$downstream(k, x\rightarrow forward) \Rightarrow downstream(k, x)$	*definition*
10.	$deleted(x) \wedge R(x\rightarrow forward) \Rightarrow R(x)$	8, 9
11.	$deleted(x) \Rightarrow R(x)$	*2, 7, 10, induction on # of forward pointers to reach undeleted node*
12.	$R(x)$	*2, 7, 11, case elimination*

THEOREM 3: $\forall(x \parallel P1(x) \wedge P2(x) \wedge R(x)) \wedge Sorted$

$$\Rightarrow \forall(\ x, k \parallel x\rightarrow key < k \leq x\rightarrow forward\rightarrow key \parallel present(k) \Leftrightarrow k = x\rightarrow forward\rightarrow key\ )$$

Proof: Given below. Why?

1.	$x \rightarrow key \leq x \rightarrow forward \rightarrow key \Rightarrow reachable(x \rightarrow forward)$	$P1(x)$, $P2(x)$
2.	$x \rightarrow key < k = x \rightarrow forward \rightarrow key \Rightarrow present(k)$	*1, definition*
3.	$reachable(x) \wedge downstream(k, x) \Rightarrow x \rightarrow key < k$	*Sorted*
4.	$reachable(x \rightarrow forward) \wedge k < x \rightarrow forward \rightarrow key \Rightarrow \neg\, downstream(k, x \rightarrow forward)$	*3*
5.	$k \neq x \rightarrow forward \rightarrow key \wedge \neg\, downstream(k, x \rightarrow forward) \Rightarrow \neg\, downstream(k, x)$	*definition*
6.	$x \rightarrow key < k \wedge \neg\, downstream(k, x) \Rightarrow \neg\, present(k)$	$R(x)$
7.	$x \rightarrow key < k < x \rightarrow forward \rightarrow key \Rightarrow \neg\, present(k)$	*1, 4, 5, 6*
8.	$x \rightarrow key < k \leq x \rightarrow forward \rightarrow key \Rightarrow (present(k) \Leftrightarrow k = x \rightarrow forward \rightarrow key)$	*2, 7*

***P1* is maintained**

When a node x is initially inserted into a list, $P1(x)$ is true (as noted in the insertion algorithm). The insertion and deletion algorithms are annotated to show that they maintain *P1* for all nodes they modify. If a node is not modified by an insertion or deletion operation, both it's reachable status and the direction of it's forward pointer (i.e., to a node with a larger key or a small key) remain unchanged, and therefore *P1* remains true.

***P2, P3* and *P4* are maintained**

The invariants *P2, P3* and *P4* are vacuously or immediately true for nodes not being deleted. The *Deletion* algorithm is annotated to show that these invariants are true for nodes that are being deleted.

2.3. Deadlock Avoidance, Serializability and Termination

The only time multiple locks are obtained is during execution of *Delete*. The second lock obtained by *Delete* is on the forward pointer of a node with a larger key than the key of the node who's forward pointer is locked first. This imposes a total order on locking order, so deadlock cannot occur.

Insertions and deletions happen precisely at the moments indicated in the *Insert* and *Delete* routines. A weak search is known to have been true at the moment a `y` := $x \rightarrow forward$ statement in *WeakSearch* is executed for the last time. This gives us a serializable schedule.

It is somewhat tricky to say something interesting about termination. A search might never terminate if the thread performing the search is slow and insertions continually happen between the node the search is currently at and the node containing the element the search is looking for. However, this is not a real problem and the search is making progress. The destination of a search is the smallest key currently in the list that is greater than or equal to the search key. Because of the invariant *R*, we know that at any point a search's destination is at some fixed (but unknown) distance k downstream. An insertion or deletion can increase by one the number of pointers a search needs to traverse. Therefore, if no new insertion/deletion threads are started, the number of pointers the search needs to traverse before completing is bounded by k (the current distance) plus the number of insertions and deletions currently in progress.

2.4. Duplicate keys

Note that the invariants and algorithms have been defined so that with a slight adjustment, everything works for lists containing duplicate keys. The insertion algorithm needs to be modified to always perform the insertion in order to insert duplicate keys, but other than that, the algorithms are unchanged. With a little extra work, it is possible to prove that operations on duplicate keys behave in *FIFO* order.

3. Skip Lists

S*kip lists* [Pug89] 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 that 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).

Each element is represented by a node in a skip list (Figure 3). 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

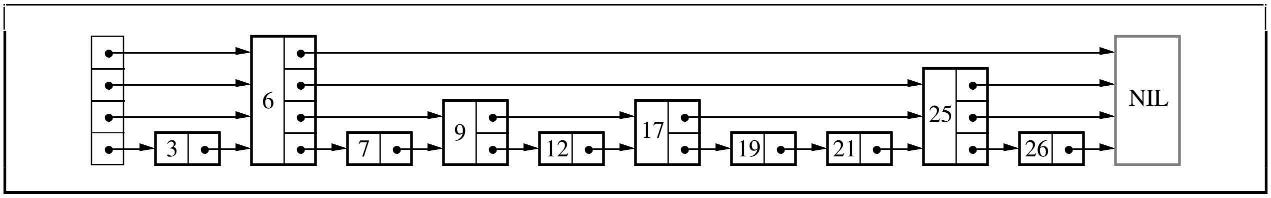

FIGURE 3 - A Skip List

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].

3.1. 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). 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.

```
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 4 - Skip list search algorithm

```
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 5 - Insertion and deletion algorithms

```
randomLevel()
    newLevel := 1
    while random() < p do
        newLevel := newLevel + 1
    return min (newLevel, MaxLevel)
```

FIGURE 6 - Generating a random level

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 4). 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. Figure 5 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.

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 6. 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.

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.

Duplicate keys

The algorithms can easily be adapted to allow duplicate keys by simply forcing the insertion algorithm to always perform the insertion. The algorithms will exhibit stack-like behavior with respect to duplicate keys.

3.2. 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. For the *Insert* and *Delete* operations, there is an additional cost proportional to the level of the element being inserted or deleted.

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 the probability of this is small enough to be ignored.

We have formally analyzed the performance of skip lists [Pug89] and summarize our results here. The expected number of comparisons required to perform a search is at most $L(n)/p + 1/(1–p) + 1$ and the variance of the number of comparisons is approximately $L(n)/p^2$. The probability distribution of the number of comparisons needed to perform a search is closely approximated by the number of coin flips needed to see $L(n)$ heads (where the probability of a coin being heads is p). This is a dominate cost measurement for searches, insertions and deletions, so the expected time for all three operations is $O(\log n)$.

3.3. Efficiency

We compared implementations of skip lists against implementations AVL trees, 2-3 trees and splay trees [Pug89]. 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 optimized splay tree implementations (using uniform query distributions).

4. Concurrent Maintenance of Skip Lists

The idea used to develop concurrent skip list algorithms is to recognize that the distribution of levels within a skip list effects only the performance of operations, not their correctness. To delete an element, we simply reduce the level of that element one step at a time, until it is level one, and then we delete it out of the level 1 linked list, which deletes the element. To change a level 4 element to a level 3 element, we simply delete the element from the linked list of level 4 forward pointers. If we think of a level 0 element

```
Search(list, searchKey)
    x := list→header
    for i := list→levelHint downto 1 do
              y := x→forward[i]
              while y→key < searchKey do
                    x := y
                    y := x→forward[i]
    -- wasTrue(self, y = x→forward[1]) ∧ x→key < searchKey ≤ y→key
    if y→key = searchKey then return y→value
         else return not-found

getLock(x, searchKey, i)
    -- x is at least level i ∧ x→key < searchKey
    -- since some time might have elapsed since we tried to advance at this level, see if we can advance further.
    y := x→forward[i]
    while y→key < searchKey do
         x := y
         y := x→forward[i]
    -- can't move any further, lock and double check result
    lock(x, forward[i])
    y := x→forward[i]
    while y→key < searchKey do
         unlock(x, forward[i])
         x := y
         lock(x, forward[i])
         y := x→forward[i]
    -- locked(x→forward[i]) ∧ x→key < searchKey ≤ key(x→forward[i]) ∧ x is reachable at level i
```

Figure 7 - Concurrent search and locking algorithms for skip lists

as an element that has no pointers and is not in the list, we can think of the process of deletion as reducing the level of an element down to 0. Insertion works similarly: we first insert the element in the level 1 linked list, then build up the level of the element as appropriate. Note that for the concurrent version of skip lists, we explicitly store the level of each element with the element.

Since the maximum level in the list is stored implicitly in the header, we don't have to store the maximum level in the list; we could have an operation first determine the maximum level i such that *list→header→forward*[i] is non–NIL, and start the search at that level. A slightly more efficient method is to store a *levelHint* with the list, which is considered to be a suggestion about what level to start a search at. It does not strictly correspond to the current maximum level of the list, but the *levelHint* will rarely be off by more than one or two. At the end of an update to the list, we check if *levelHint* is different from the maximum level of the list. If it is different and no other thread is updating it, we update it.

The algorithms are shown in Figures 7-9. The *Search* routine returns a result that is guaranteed to have been true at some point during the search. The *getLock* routine locks the forward pointer of the specified level immediately in front of the specified search key. The concurrent insertion and deletion algorithms for skip lists are somewhat more complicated. In addition to locking forward pointers, we also need to be able avoid situations such as two threads trying to delete the same element at the same time, or a thread trying to delete an element that is being inserted. A thread inserting or deleting an element locks the level of that element to prevent these situations from arising. An element is deleted from a skip list at the same moment it is deleted from the linked list of level 1 forward pointers (i.e., it is at this moment that a search will stop reporting the element as being in the list).

If the insert routine finds the element already in the list, it simply updates the value field. Note that an element cannot be deleted by another thread while we hold a lock on either the *level* of that element or on the level 1 forward pointer to that element.

Duplicate keys

The concurrent skip list algorithms also can be adapted to allow duplicate keys. However, in the straightforward implementation, an anomaly arises. If two elements with equal keys e_1 and e_2 are inserted concurrently, it is possible that e_1 occurs before e_2 in the level 1 forward pointers, and that e_2 occurs before e_1 in the level 2 forward pointers. Since e_1 and e_2 have the same key, this does not cause any major problems. However, the deletion routine needs to be written so that it searches using a *getLock* routine that searches for a particular node, and not for the first occurrence of a particular key.

4.1. Efficiency

The efficiency of concurrent skip lists rests on several factors:

1. The efficiency of (single-thread) skip lists
2. The overhead of the locking protocol (disregarding contention).
3. The overhead associated with making references to global shared memory, compared with the overhead of accessing local memory.
4. The amount of lock contention.

Rather than present numbers that would be particular to one benchmark, we only attempted to analyze aspects of our algorithms that should be consistent across implementations. As mentioned in Section 3.3, the efficiency of single-thread skip lists is similar to the efficiency of highly optimized balanced trees. The second factor is dependent on the architecture of the computer the algorithm will be implemented on. If there is no lock contention, an insertion needs an average of $1+1/(1-p)$ locks, and a deletion needs $1+2/(1-p)$ locks. Assuming lock contention does not dramatically increase the number of locks obtained, the locking protocol overhead should be small. The third factor is strongly dependent on the computer architecture, so we do not address it here.

Therefore, the primary question we need to address is the amount of lock contention (and whether or not lock contention has any effect on the number of locks obtained by a transaction). We ran simulations designed to test the amount of lock contention. In these simulations, we used busy-waiting, test-and-set semaphores for locking. At each step in the simulation, a random thread was advanced by one step; thus, some threads may be stalled for a long time. We felt that this simulated an appropriately hostile situation for fairness.

There are many possible scenarios for a simulation. One possibility would be to have n threads all attempt to insert elements into an initially empty data structure. This would lead to horrendous lock contention, as all n threads attempted to lock simultaneously the only level 1 forward pointer in the initial data structure and n–1 of the threads were blocked.

In our simulation, we started with a data structure containing m elements. Each of n threads inserts a random element, deletes an element and then repeats the entire process indefinitely. No two threads ever attempt to insert or delete the same element simultaneously. At any moment, the skip list would contain between m and $m+n$ elements (depending of the phase of each thread). We collected statistics only once the simulation had reached a steady state. We used a parameter of $p = 0.5$ for generating random levels.

Our results

Because of the design of our simulation, there was no contention for locks on the levels of elements. Because a task only attempts to obtain a lock on the *levelHint* of a list if it is was unlocked when it last checked, there was an insignificant amount of contention of locks on the *levelHint* of a list. This could be totally eliminated by writing the algorithm so that if it failed to get a lock initially, it would not wait for a lock. Therefore, we only report on contention for locks on forward pointers.

The results from our simulations are shown in Table 2. The percentage of lock requests blocked gives the percent of time a lock request for a forward pointer was forced to wait. The fourth column gives the average number of locks held on forward pointers as a percentage of the total number of forward pointers. The average number of locks obtained per transaction would be 3 without lock contention (2 for insertions, 4 for deletions).

These results show that we can achieve almost linear speed-up. For 1000 threads, we obtain a speed-up of 921. These results are in line with our expectations. The percentage of locks blocked seems proportional to the percentage of locks held, which is determined by the proportion of time the algorithms spend holding locks. This is in turn is determined by the proportion of concurrent writers to the number of elements in the data structure. The average number of locks per transaction is almost constant. Assuming the ratio of threads to elements remains constant, lock contention actually decreases as the number of threads increase. This is because the average amount of time a lock is held is constant, while the total time to perform a transaction increases as the number of elements increases. Therefore, as the size of the problem increases, the percentage of locks held decreases, which decreases the amount of lock contention. We also found that a thread had to wait an average of 5 ticks for a lock if it was blocked, which is approximately the same number of ticks a thread holds a lock for once it has obtained the lock. This suggests that the queue for a lock very rarely grew larger than one.

5. Related Work

There has been numerous papers on concurrency schemes for trees. Those for balanced trees [Ell80a] [Ell80b] tend to be very complicated, require exclusive locks and read locks, and allow at most $O(\log n)$ busy writers. Some of the concurrency schemes for unbalanced trees [KL80] [Man84] [ML84] allow $O(n)$ busy writers and are simpler than concurrent balanced tree schemes. However, certain input patterns can easily cause poor performance and the concurrency algorithms we have presented here for skip lists appear simpler and allow as much or more concurrency.

# of concurrent writers	# of elements: min–max	% of locks requests blocked	average % of locks held	average # of locks obtained per transaction	average # of steps per transaction
100	0–100	17%	25%	3.26	63
10	990–1000	0.09%	0.08%	3.00	70
100	900–1000	1%	1.4%	3.01	70
1000	0–1000	15%	20.5%	3.23	76

Table 2 – Results of simulations

It does not seem that skip lists are particularly well suited for disk-based data structures, so this work does not provide any direct competition for concurrent *B*-trees.

6. Conclusions

The concurrent skip list algorithms described in this paper provide an efficient and practical method of allowing concurrent access and updates to a search structure in shared memory. Since skip lists are roughly as fast or faster than balanced trees in a non-concurrent environment and contention does not significantly slow down concurrent skip lists, I conjecture that the concurrent skip list algorithms described in this paper are at least as efficient as any possible concurrent balanced tree implementation. It might be possible to design concurrent balanced tree algorithms that allowed $O(n)$ busy writers with high efficiency, but the complexity of such algorithms probably would make their implementation prohibitive.

References

[Ell80a] Ellis, C. Concurrent search and insertion in AVL trees, *IEEE Trans. on Comput.* C-29 (Sept. 1980) 811-817.

[Ell80b] Ellis, C. Concurrent search and insertion in 2-3 trees. *Acta Inf.* 14 (1980) 63-86.

[KL80] Kung, H.T. and Lehman, Q. Concurrent Manipulation of Binary Search Trees, *ACM Trans. on Database Systems,* Vol. 5, No. 3 (Sept. 1980), 354-382.

[Man84] Manber, U. Concurrent Maintenance of Binary Search Trees, *IEEE Transactions on Software Engineering,* Vol. SE-10, No. 6 (November 1984), 777-784.

[ML84] Manber, U. and Ladner, P. Concurrent Control In a Dynamic Search Structure, *ACM Trans. on Database Systems,* Vol. 9, No. 3 (Sept 1984), 439-455.

[Pug89] Pugh, W. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Algorithms and Data Structures: Workshop WADS '89,* Ottawa, Canada, August 1989, Springer-Verlag Lecture Notes in Computer Science 382, 437-449. (*revised version to appear in Comm. ACM*).

```
Insert(list, searchKey, newValue)
    local update[1..levelCap]
    x := list→header
    L := list→levelHint
    for i := L downto 1 do
        y := x→forward[i]
        while y→key < searchKey do
            x := y
            y := x→forward[i]
        update[i] := x

    x := getLock(x, searchKey, 1)
    if x→forward[1]→key = searchKey then
        x→forward[1]→value := newValue
        unlock(x, forward[1])
        return updated

    y := makeNode(randomLevel(), searchKey, value)
    lock(y, level)
    -- if L < y→level, add levels L+1..y→level to update, so that we can insert y into levels 1..y→level.
    -- This will allow the header of the list to be updated correctly.
    for i := L+1 to y→level do update[i] = header(list)
    -- insert y into levels 1..level(y)
    for i := 1 to y→level do
        -- y is effectively a level i–1 element
        if i ≠ 1 then x := getLock(update[i], searchKey, i)
        -- searchKey is not present at level i, ∴ x→key < searchKey < x→forward[i]→key
        y→forward[i] := x→forward[i]
        x→forward[i] := y
        unlock(x, forward[i])
    unlock(y, level)

    -- increase levelHint to correspond to maximum non–NIL level if needed
    L := list→levelHint
    if L < levelCap and list→header→forward[L+1] ≠ NIL
            and not locked(list, levelHint) then
        lock(list, levelHint)
        while list→levelHint < levelCap
                and list→header→forward[list→levelHint+1] ≠ NIL do
            list→levelHint := list→levelHint+1
        unlock(list, levelHint)

    return inserted
```

Figure 8 – Concurrent insertion algorithm for skip lists

```
Delete(list, searchKey)
    local update[1..levelCap]
    x := list→header
    L := list→levelHint
    for i := L downto 1 do
        y := x→forward[i]
        while y→key < searchKey do
            x := y
            y := x→forward[i]
        update[i] := x

    y := x
    repeat
        y := y→forward[i]
        if y→key > searchKey then return not-found
        lock(y, level)
        isGarbage := y→key > y→forward[i]→key
        if isGarbage then unlock(y, level)
    until y→key = searchKey and not isGarbage
    -- y is in the list and we have exclusive insert/delete rights
    for i := L+1 to y→level do update[i] := list→header
    for i := y→level downto 1 do
        -- loop invariant: y is effectively a level i element
        x := getLock(update[i], searchKey, i)
        -- x→forward[i] = y
        lock(y, forward[i])
        x→forward[i] := y→forward[i]
        y→forward[i] := x
        unlock(x, forward[i])
        unlock(y, forward[i])
    putOnGarbageQueue(y)
    unlock(y, level)

    -- decrease levelHint to correspond to maximum non–NIL level if needed
    L := list→levelHint
    if L > 1 and list→header→forward[L] = NIL and not locked(list, levelHint) then
        lock(list, levelHint)
        while list→levelHint > 1 and list→header→forward[list→levelHint] = NIL do
            list→levelHint := list→levelHint – 1
        unlock(list, levelHint)

    return deleted
```

Figure 9 – Concurrent deletion algorithm for skip lists