LOCK-FREE LINKED LISTS AND SKIP LISTS

MIKHAIL FOMITCHEV

A thesis submitted to the Faculty of Graduate Studies in
partial fulfilment of the requirements
for the degree of
Master of Science

Graduate Programme in Computer Science
York University
North York, Ontario

November, 2003

Abstract

Lock-free shared data structures implement distributed objects without the use of mutual exclusions, thus providing robustness and reliability. We present new implementations of lock-free linked list and lock-free skip list dictionary data structures for shared-memory systems. We give a detailed proof of correctness for both of them and present an amortized performance analysis for our linked lists. To the best of our knowledge, our implementation of the lock-free skip lists is the first that does not use the universal constructions. We also show that our linked lists implementation has a better amortized performance than prior lock-free implementations of this data structure. Our algorithms use the single word C&S synchronization primitive.

Acknowledgements

First, I want to thank my supervisor Eric Ruppert. Without his guidance and support, this thesis probably would not have been completed, or at the very least, it would be in a much poorer shape. I am very grateful for his vast contributions to this thesis and for the many things that I have learnt from him during my M. Sc. studies.

I would also like to thank the professors I have been taking courses with during my graduate studies in York University. Particularly, I would like to thank Patrick Dymond, my co-supervisor. The insight into the amortized analysis techniques that I have obtained from him was critical to the success of this work.

I am grateful to all members of my examining committee for their time and for their helpful comments on my thesis.

I would like to express my gratitude to the Ontario Graduate Scholarship program for the scholarship I was awarded in 2002-2003 academic year.

Last, but not least, I would like to thank Computer Science Department of York University and Computational Physics Department of St. Petersburg State University for the excellent educational opportunities that I have been offered during my studies.

Table of Contents

1 INTRODUCTION 1

2 RELATED WORK 6

3 LINKED LISTS 8

3.1 High-level description and motivation 8
3.1.1 Common problems 8
3.1.2 Mark bits and successor fields 9
3.1.3 Back_links 10
3.1.4 Flagging successor fields 15

3.2 Algorithms 16
3.2.1 The data structure and notation 16
3.2.2 Pseudo-code 17

3.3 Correctness 22
3.3.1 Invariants 22
3.3.2 Critical steps of a node deletion 30
3.3.3 The SearchFrom routine 31
3.3.4 Linearization points and correctness 33

3.4 Performance analysis 39
3.4.1 Billing extra work to the successful C&S's 40
3.4.2 Introduction to Billing 40
3.4.3 More on Billing 44
3.4.4 Mapping C&S operations. 49
3.4.5 Mapping back_link pointer traversals 54
3.4.6 Mapping next_node and curr_node pointers updates 55
3.4.7 Putting everything together 58

4 SKIP LISTS 61

4.1 High-level description 61
4.1.1 The skip list data structure 61
4.1.2 Lock-free skip list design 62
4.1.3 Implementation issues 64

4.2 Algorithms 67
4.2.1 Pseudo code 68

4.3 Correctness 73
4.3.1 Invariants for the levels of the skip list 73
4.3.2 Critical steps of a node deletion 80

4.3.3 Vertical structure of the skip list 81
4.3.4 The SearchRight and SearchToHeight_SL routines 84
4.3.5 The Insert_Node and Insert_SL routines 90
4.3.6 Proving invariants for the towers of the skip list 93
4.3.7 A stronger SearchToLevel_SL postcondition 93
4.3.8 Linearization points and correctness 100
4.3.9 The heights of the towers 106

4.4 Lock-Freedom and performance 111
4.4.1 Lock-freedom 111
4.4.2 Fine tuning performance and space requirements 114

5 MEMORY MANAGEMENT 116

5.1 Existing approaches 116
5.1.1 Reference counting 116
5.1.2 Garbage collectors 117
5.1.3 Deferred freeing 117

5.2 The new approach 117

6 CONCLUSION 119

REFERENCES 121

1 Introduction

One of the standard ways to implement data structures in distributed systems is to use *mutual exclusion*. With mutual exclusion, *locks* are used to resolve conflicts between processes attempting to access data structure simultaneously. Although this approach is widely used and it is easy to implement data structures this way, it has a major weakness – when one process is in the *critical section* (i.e. when it is modifying the data structure), all the other processes must wait before they are permitted to access it. Thus, any delay of a process while in the critical section (due to a page fault, memory access latency, etc.) becomes a bottleneck, which can cause serious performance problems. When process failures are possible, this becomes particularly important, because the entire system can stop making progress if a process fails in the critical section.

By contrast, an implementation of a shared-memory object is *lock-free* (or *non-blocking*), if for any possible execution, the system as a whole is always making progress, i.e. starting from any point of time during an execution, after a finite number of steps taken by one of the processes, some process is guaranteed to complete its operation. An implementation is *wait-free* if any non-faulty process will make progress, i.e. starting from any state of the system each process is guaranteed to complete its operation after a finite number of its own steps.

Lock-freedom is a desirable property, because if an implementation is lock-free, individual delays or failures of the processes do not block the progress of other processes in the system. Thus a lock-free data structure is more fault-tolerant and more resilient to scheduling decisions, compared to a data structure implemented using mutual exclusion. Lock-free data structures also have the potential to have better performance, because several processes are allowed to modify a data structure at the same time. Indeed, practical testing of recent implementations of lock-free linked lists [Har01, Mic02-1] has shown that they perform better than their counterparts that use locks.

When an object is lock-free and not wait-free, *starvation* of the processes trying to perform operations on the object is possible. A process P experiences starvation when it takes steps, but due to the other processes operating on the object, P cannot complete its operation indefinitely. If an object is wait-free, processes performing operations on it do not starve. The wait-freedom property, although desirable for some applications, usually entails significant overheads compared to lock-freedom. In this thesis we focus on the lock-free designs.

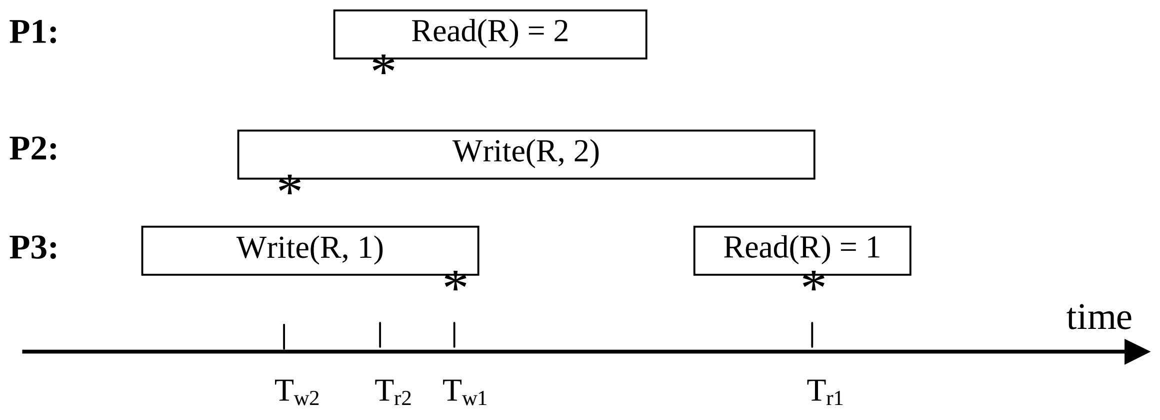
A system is called *asynchronous* when processes in the system run at arbitrary varying speeds, i.e. the scheduling of each process is independent from the scheduling of other processes.

Our model is an *asynchronous shared-memory* distributed system of several processes, where an arbitrary number of *process failures* are allowed. The failures that are allowed are *halting failures:* a failed process can stop taking steps indefinitely, but does not exhibit Byzantine (malicious) behaviour.

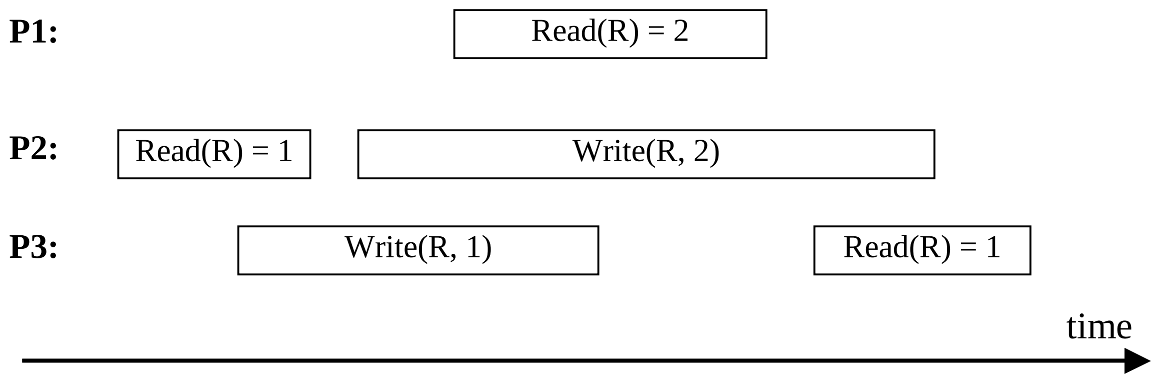
To define the correctness of our implementations we use the notion of *linearizability* [HW90]. An execution of concurrent operations on a shared-memory object is said to be *linearizable* if each operation performed on this object by the processes can be viewed as happening instantaneously at some time between its invocation and response. This time is called the *linearization point* of the operation.

Responses to the operations have to be same as if the operations were executed atomically at their linearization points. For example, if R is a read-write register initially containing 0, the execution showed in Figure 1(a) is linearizable. We can choose the linearized the operations as follows: Write(R, 2) at time T_{w2}, the Read that returns 2 at time T_{r2}, Write(R, 1) at T_{w1}, and the Read that returns 1 at T_{r1}.

The execution showed in Figure 1(b), on the other hand, is not linearizable: let T_{w1}, T_{w2}, and T_{r2}, T_{r1}, and T_{r1}' be the linearization points of Write(R, 1), Write(R, 2), the Read that returns 2, the Read performed by P2 that returns 1, and the Read performed by P3 that returns 1 respectively. Write(R, 1) must be linearized before the Read performed by P2, because the Read returns 1. Process P2 starts Write(R, 2) after it completes the Read operation, therefore Write(R, 2) has to be linearized after Write(R, 1), i.e. $T_{w1} < T_{w2}$. The Read performed by P1 returns 2, so Write(R, 2) must be linearized before that Read. Then Write(R, 2) is linearized before the Read performed by P3, because P3's read starts after P1's read finishes. Therefore $T_{w2} < T_{r1}$. So, $T_{w1} < T_{w2} < T_{r1}$, but then the read performed by P3 must return 2 – a contradiction.



(a) Linearizable execution.



(b) An execution that is not linearizable.

Figure 1: Linearizability.
(R is a read-write register initially containing 0.)

An implementation of an object is *linearizable* [HW90] if all its possible executions are linearizable. Linearizability is a commonly desired property for shared-memory objects, because it allows a user to abstract away and ignore the details of the object behaviour when several operations by different processes overlap in time. With linearizability, all operations performed on the object can be viewed as a sequence of atomic steps, and therefore in a deterministic system an execution can be defined by the schedule in which the processes take steps. In our model this schedule is chosen by the adversary. Linearizability is one of the properties that we require our implementations to possess.

Herlihy showed [Her91, Her93] that multi-writer multi-reader registers alone are not sufficient to implement arbitrary lock-free data structures. He showed that in order to be able to design arbitrary lock-free data structures, one needs a *universal synchronization primitive*. Our algorithms use a single-word *COMPARE&*SWAP object (sometimes called a COMPARE&SWAP register), which is one of the universal synchronization primitives. This object implements an atomic single-word COMPARE&SWAP operation, denoted C&S (see Figure 2), which accepts three arguments: address, old_value, and new_value. If the value stored at the address is equal to old, it updates it to new and returns the old value stored in the register. Otherwise, it just returns this value stored at the address. The C&S object also allows processes to perform simple reads and writes.

```
C&S(a: Address; old, new: Word): Word          {ATOMIC}
if (a^ == old)
    a^ = new
    return old
return a^
```

Figure 2: C&S operation.

The single-word C&S object was first introduced in the IBM System/370 architecture in the late 1970's. Today this primitive or its equivalents are commonly supported by many popular parallel architectures (SPARC v.9, Alpha AXP, PowerPC, etc.). Some implementations of lock-free data structures use a stronger double-word C&S operation, denoted DC&S. This operation executes two different single-word C&S's at once, and if one of these C&S's fails, the other fails as well, i.e. an operation can either succeed on both addresses, or fail on both. DC&S was supported on some Motorola machines of the 68000 processor family in the 1980s, but proved to be inefficient. Since then no processor architectures support this operation. In the future when we say C&S operation, we will mean a single-word C&S, unless stated otherwise.

Herlihy [Her91, Her93] introduced the first universal method of designing lock-free data structures using C&S. Many others followed, but all of them suffer from several major flaws, such as inefficiency, low parallelism, excessive copying, and generally high overhead, which often make them impractical. To achieve adequate performance, original algorithms, specific to a particular data structure are usually required. The design of such algorithms is a challenging task even for very simple data structures. For example, no one knows yet how to implement efficient lock-free doubly-linked lists or BSTs.

We present a new design of a lock-free singly-linked list using C&S which implements dictionary operations with a better order of complexity than all prior implementations. (The constant factors of the complexity of our implementation are quite reasonable). We give a detailed proof of correctness and a performance analysis of our data structure.

We also give a new implementation of a lock-free skip list using C&S along with a detailed proof of correctness. A skip list [Pug90] is a dictionary data structure, providing randomized algorithms for searches, insertions, and deletions. The worst-case cost of operations on the skip list is O(n), but the expected cost of any operation is O(log(n)) (where the expectation is taken over random choices made by the algorithms). Here n is the number of elements in the skip-list. We talk about skip lists extensively in Chapter 4. Our implementation of lock-free skip lists is the first such implementation that does not use the universal constructions. It is worth mentioning, that the most recent implementations of lock-free linked lists [Har01, Mic02-1] were evaluated by their authors by doing practical testing rather than a performance analysis. We believe that there exists a certain lack of theoretical development in this area, and we hope that our work partially addresses this problem.

When analyzing performance of a lock-free data structure, one generally cannot evaluate the worst-case cost of individual operations, because the lock-freedom property allows individual operations to take arbitrarily many steps, as long as the system as a whole is making progress. On the other hand, performing an *amortized analysis* seems to be a natural thing to do, because it evaluates the performance of a system as a whole.

We evaluate the performance of our data structure and the prior lock-free linked list implementations by performing an amortized analysis, and we measure the performance as a function of *contention*.

Point contention is the number of processes running concurrently at a given point of time. We define *contention of operation S* denoted c(S) to be the maximum point contention during the execution of S.

The amortized analysis of our data structure relies on a fairly complex technique of billing part of the cost of each operation to the successful C&S's that are performed by operations that are running concurrently. Generally speaking, if a process P has to perform some extra work because of a modification performed by another concurrent process P′, then the cost of that extra work is billed to P′. The amortized cost of an operation S, denoted $\hat{t}(S)$, is equal to the actual cost of S plus the total cost billed to S from other operations minus the total cost billed from S to other operations. We prove that $\hat{t}(S) = O(n(S) + c(S))$, where n(S) is the number of elements in the data structure when operation S is invoked and c(S) is the contention of operation S. We then show that for any execution E, the cost of the entire execution, denoted t(E), is $O\left[\sum_{S\in E}(n(S)+c(S))\right]$, where the sum is taken over all operations invoked during E, and that the average cost of an operation in the execution $\bar{t}_E(S) \in O\left[\frac{\sum_{S\in E}(n(S)+c(S))}{m}\right]$ where m is the total number of operations invoked during E. If convenient, one can also use the following two bounds that are less tight: $\bar{t}_E(S) \in O(\max(n) + \max(c))$ (and hence

$\bar{t}_E(S) \in O(m)$), where max(n) is the maximum value of n(S) during E, and max(c) is the maximum value of c(S) for all S in E.

The rest of the thesis is organized as follows. Chapter 2 gives an outline of related work done in the area. Chapter 3 presents our linked list data structure: Section 3.1 gives a high-level description and motivation for our implementation, Section 3.2 contains the algorithms themselves, Section 3.3 presents the invariants and some properties of our data structure, as well as the proof of correctness, and Section 3.4 is devoted to the performance analysis. Chapter 4 presents our skip list data structure, and is organized similarly to the third chapter, except that in Section 4.4 we only prove the lock-freedom property, without giving an analysis. Chapter 5 describes several existing memory management schemes that can be used with our data structure, and outlines a new scheme, which we currently are investigating. Chapter 6 contains some concluding remarks.

2 Related Work

A general method for creating lock-free implementations of any shared data structure using C&S operations was presented by Herlihy [Her91, Her93]. Although universal, his methods are highly inefficient for most of the real-life applications (including linked lists and skip lists), because they essentially employ C&S to change one shared global pointer to the data structure, and thus each operation must copy the entire data structure, and when several processes are attempting to perform operations on the data structure concurrently, only one can succeed.

An efficient lock-free implementation of singly-linked lists was first presented by Valois [Val95]. The main idea of his approach was to maintain auxiliary nodes in between normal nodes of the linked list and to use the fields of these nodes to perform insertions and deletions. One of the major problems was to delete these auxiliary nodes once the corresponding “real” nodes were deleted. Valois partially solved this problem by introducing back_link pointers to the nodes (back_link is basically a pointer to a node’s predecessor). Still, in the worst-case execution, arbitrarily large chains of auxiliary nodes could be created, which hampered the performance of the data structure. Also, because of the auxiliary nodes, algorithms for the basic operations on the list were fairly complicated. Valois claims [Val95] that his linked list implementation can be used to design skip lists, but he does not explain how exactly this can be done. Implementing skip lists, as we will see, involves several challenges that do not arise with the linked lists and are not trivial to deal with. This is especially true when an underlying linked lists design is complicated, as Valois’s design. Also, in Valois’s linked list implementation processes used rather complicated cursors consisting of three pointers to traverse the linked list. It was not clear how these cursors could be maintained when processes change levels performing the searches in a skip list.

Another implementation of linked lists was given by Harris [Har01]. His algorithms are simpler than Valois’s and his experimental results show that generally they also perform better, but, as we will show later, in the worst case they perform worse. His main idea was to introduce *mark bits*, and to perform deletions in two steps: first mark the node by setting its mark bit to 1 (*logical deletion*), and then delete it from the list by updating the right pointer of the preceding node (*physical deletion*). The right pointer of each node was replaced with a composite field (let us call it *successor field*), consisting of the normal right pointer and a mark bit. All C&S operations performed by Harris’s algorithms were applied to a successor field, and none of them modified successor fields with a mark bit set to 1. Although these algorithms performed better than Valois’s during practical testing, there was no theoretical analysis presented. In Subsection 3.1.2 we present an example, which illustrates that Harris’s approach does not always result in a good performance.

Yet another implementation of a lock-free linked list was proposed by Michael [Mic02-1], as part of his lock-free hash table design. His implementation was based on Harris’s ideas and uses the same design. The same example, illustrating the possibility of bad performance, which we give for Harris’s algorithms in Subsection 3.1.2, applies to Michael’s algorithms as well. However, Michael’s algorithms, unlike Harris’s, are compatible with efficient memory management techniques, such as IBM freelists [IBM83, Tre86] and the safe memory reclamation method (SMR) [Mic02-2].

Another contribution in the area of the lock-free data structures which is worth mentioning, is the implementation of lock-free *extensible* hash tables proposed by Shalev and Shavit [SS03]. They use a linked list to store all the elements of the table with each bucket storing a pointer to an appropriate section of the list.

No implementations of lock-free skip lists have been published. It is interesting, however, that a lot of data structures similar to skip-lists appear in peer-to-peer distributed applications (e.g. Chord [SMKKB00]).

Our design is built using several new ideas, as well as some of the ideas from Harris's and Valois's approaches. An amortized cost of an operation S performed by our algorithms is O(n(S) + c(S)), where n(S) is the number of elements in the data structure when S is invoked and c(S) is the contention of S. Based on this, we show that for any execution E, the average cost of an operation in the execution $\bar{t}_E(S) \in O\left[\frac{\sum_{S \in E}(n(S)+c(S))}{m}\right] \subset O(\max(n) + \max(c)) \subset O(m)$ where m is the total number of operations invoked during E, max(n) is the maximum value of n(S) during E and max(c) is the maximum value of c(S) during E . To compare, the average costs of operations in Valois's implementation can be $\Theta(m)$ (even when $m \notin O(n)$). Obviously, m is always at least n, and the difference can be quite significant. The average costs of operations in Harris's implementation can be $\Omega(\max(n)\max(c))$, which is also strictly worse than in our implementation.

3 Linked Lists

In this chapter we present our implementation of a lock-free sorted linked list data structure. We start by giving a high-level description of our data structure and the motivation behind our implementation, then we present our algorithms, prove their correctness, and finally present the performance analysis of our data structure.

3.1 High-level description and motivation

In this section we will describe some common difficulties one encounters when designing a lock-free linked list, and explain how our data structure overcomes them.

3.1.1 Common problems

One of the major problems one runs into when designing a lock-free linked list is illustrated below. Suppose we use a straightforward approach and perform insertions and deletions by using C&S on the right pointers of the nodes. Consider the example illustrated in Figure 3. Initially the list contains nodes A, B and D. Node B is deleted from the list by switching A.right from B to D, and at the same time node C is inserted by switching B.right from D to C. The pre-change pointers are represented with solid lines, and post-change pointers are represented with dashed lines. The resulting list contains only nodes A and D, so B was successfully deleted, but C was not successfully inserted.

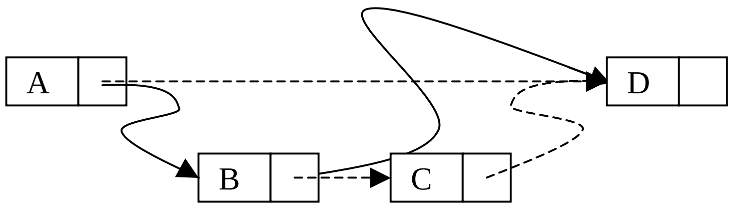


Figure 3: Concurrent deletion of B and insertion of C

A similar problem arises when we try to delete two adjacent nodes concurrently as shown in Figure 4. Initially the list contains nodes A, B, C, and D. Node B is deleted by switching A.right from B to C, node C is deleted by switching B.right from C to D. As a result of these changes, node B is successfully deleted from the list, but node C remains in the list.

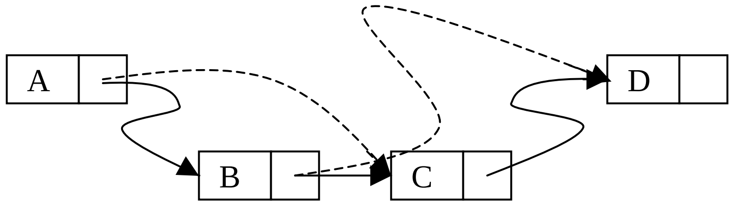


Figure 4: Concurrent deletion of B and C

The root of these problems lies in the fact that we cannot easily control both the right pointer of the node we are deleting and the right pointer of its preceding node at the same time if we use a simple C&S primitive. (It can be done with a stronger double-compare-and-swap primitive [Gre99].)

3.1.2 Mark bits and successor fields

To counter these problems one can use the technique of Harris's implementation [Har01]. Instead of applying C&S's to the right pointer of the node, apply it to the *successor field*, which consists of a right pointer and a *mark bit*. Normally, the mark bit of a node is 0. To perform a deletion of the node, use two C&S operations: the first *marks* the successor field of the node by setting its mark bit to 1, and the second removes the node from the list, as illustrated in Figure 5, where marked successor fields are crossed. A node is *logically deleted* after the first step, and *physically deleted* after the second step. A node is called *marked* if and only if its successor field is marked. All of the C&S's performed by the algorithms will only modify unmarked successor fields, so once the successor field of a node is marked, it never changes, and thus when the physical deletion of a node is performed, its right pointer (which is part of the successor field) is fixed. This solves the problems described above.

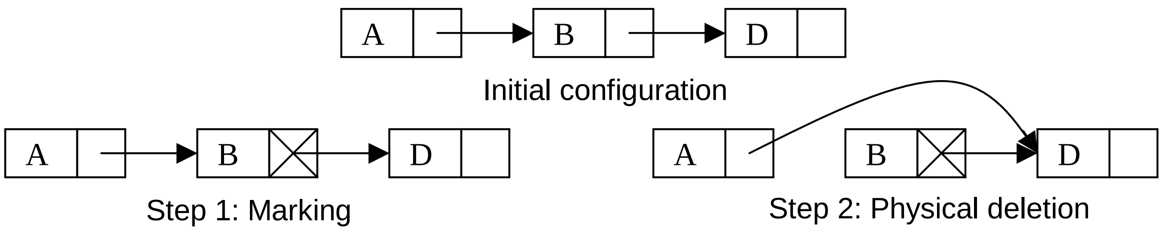


Figure 5: Two-step deletion of a node.

We use a three-step deletion procedure and 2 bits, not 1, to reflect the status of the node. The reason is that, although a two-step deletion solves some problems, it also introduces performance-related problems. Suppose there is a process P_1 that is performing an insertion of node C between nodes B and D, and at the same time process P_2 is performing a two-step deletion of node B. Then, P_1 is trying to update the right pointer of B from D to C, and P_2 is trying to mark B (see Figure 6).

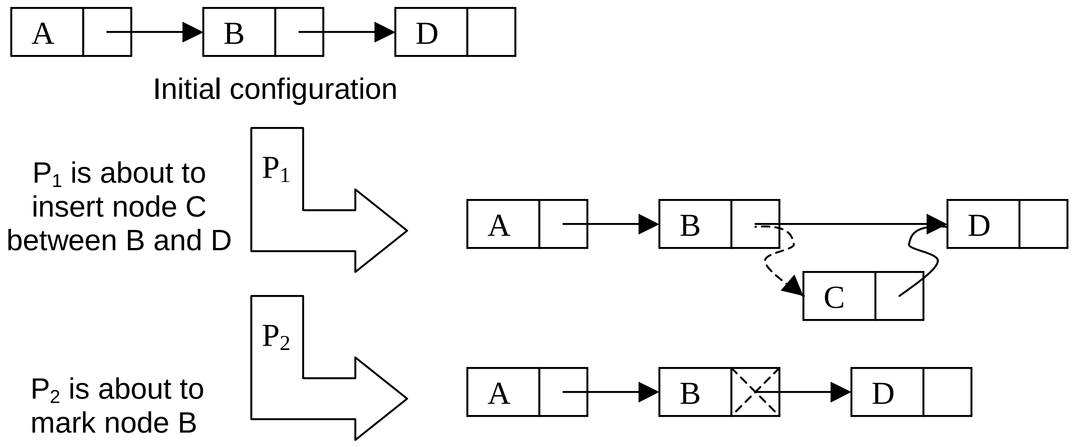

A
B
D
Initial configuration
P_1 is about to insert node C between B and D
P_1
A
B
D
C
P_2
P_2 is about to mark node B
A
B
D

Figure 6: Conflict of two processes.

Suppose that P_2 performs the change before P_1 does. Then P_1's C&S will fail and P_1 will have 2 options:

1. Start the Insert operation all over again, or
2. Try to recover from the failure.

The first approach is an easier way to deal with the problem, and this is basically how Harris's [Har01] algorithm works. But this requires P_1 to start the search for the right place to insert C all over again, from the beginning of the list. This results in the average cost of an operation for some executions to be $\Omega(\max(n)\max(c))$, where max(n) is the maximum number of elements in the list during the execution, and max(c) is the maximum contention during the execution. The following example execution yields this bound.

First processes insert n elements with keys $k_1, k_2, \ldots , k_m$ into the list. Then $q - 1$ processes $P_1, P_2, \ldots , P_{q-1}$ attempt to insert the keys $k_{m+1}, k_{m+2}, \ldots ,k_{m+q-1}$ such that $k_m < k_{m+1}, k_{m+2}, \ldots , k_{m+q-1}$, i.e. they are trying to insert keys greater than the largest key in the list. Concurrently, the remaining one process P_q deletes keys $k_m, k_{m-1}, \ldots , k_1$ (in that order). Suppose P_q always performs deletions just before any of the inserting processes perform their C&S's, but after all of them locate the position they wish to insert the key into. In other words, the execution goes as follows: processes $P_1, P_2, \ldots , P_{q-1}$ start their searches, find the appropriate insertion position, and then P_q performs a deletion. Consequently, processes $P_1, P_2, \ldots , P_{q-1}$ fail their respective insertion C&S's, and thus they have to start their searches all over again from the head of the list, and the cycle repeats. Since processes $P_1, P_2, \ldots , P_{q-1}$ are attempting to insert the keys at the end of the list, they have to search through the whole list to locate the appropriate insertion position. Before the first deletion the length of the list is n, and with each deletion it decreases by 1, so the total work done by the system throughout the sequence of operations is at least $\Omega(q(m + (m - 1) + \ldots + 1) = \Omega(qm^2)$. The total number of operations performed is $2m + q - 1$: m initial insertions, m deletions by P_q, and $q - 1$ insertions by $P_1, P_2, \ldots , P_{q-1}$, so the average cost of an operation in this execution is $\Omega(qm^2/(2m + q - 1))$. If we choose $m > q$, then $\Omega(qm^2/(2m + q - 1)) = \Omega(qm)$. Note that in the described execution the maximum number of elements in the list max(n) is m and the maximum contention max(c) is q. Therefore, this execution yields the desired bound.

Our algorithms do allow a process to recover from the failure and achieve an average cost of $O\left[\frac{\sum_{S\in E}(n(S)+c(S))}{m}\right] \subset O(\max(n) + \max(c))$ per operation. The recovery procedure is described below.

3.1.3 Back_links

The first idea is to introduce back_link pointers, similar to the ones proposed by Valois [Val95], and use them to replace mark bits, so that now the successor field consists of a right pointer and a back_link. If the node has a null back_link pointer, it is not marked, otherwise it is. In the marking stage of a deletion, a process would set the back_link of the node to be deleted to point to the preceding node, using C&S on the node's successor field.

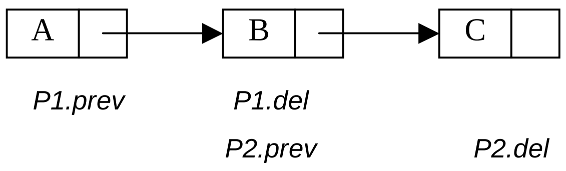
For example, when P2 marks node B (Figure 6), it would set B's back_link to point to node A. Then, if another process (P1 in our example) fails an insertion because B has been marked, it would traverse the back_link to the preceding node and start another search from there, instead of starting it from the head of the list. Thus, if everything goes as described, the cost of recovery would be reduced from O(n) to O(1). However this approach does not always work. There can be three problems that can lead to increases in the recovery cost:

1. Before the physical deletion of B takes place, new nodes can be inserted between A and B, so that after P1 fails its insertion of C and follows B's back_link to A, it will have to search through all of these new nodes before it can attempt to insert C again.
2. Nodes that have back_links pointing to them (such as A) can get deleted as well, so that instead of following one back_link from B to A, P1 might have to follow a chain of such back_links.
3. Back_links might be set to point to nodes that are already marked or even physically deleted. In the case illustrated in Figure 6, it is possible that right before P2 marks B by setting its back_link to point to node A, A itself can get marked and physically deleted by another process P3. Again, as a result process P1 will have to traverse a chain of back_links to recover from the failure. This might look similar to the 2nd problem described previously, but it is actually different: here back_links are added at the right end of the chain, at its beginning, and the chain is growing *towards the right*, whereas in the second problem back_links were added at the left end of the chain, at its end, and the chain was growing *towards the left*.

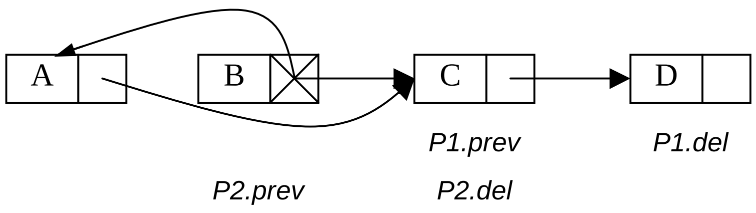
The 1st and the 2nd problems do not actually increase the order of the amortized cost of operations. We will present a formal proof of this later, but the reason for this is that if back_link chains cannot grow *towards the right*, each process will traverse a given back_link no more than once, and the cost of traversing back_links and newly inserted nodes can be billed to respective deletions and insertions. Then it can be shown that each process can be billed no more than O(c) for each C&S operation it performs, so that the amortized cost of any operation is O(n + c).

The third problem, however, leads to the possibility of an execution with high amortized cost of operations and high memory requirements. Even when the maximum number of elements in the list throughout an execution does not exceed four, and the number of processes does no exceed three, the amortized cost of each operation and the amount of memory required are Ω(m), where m is the total number of operations invoked. (Thus, to guarantee good performance and reasonable memory requirements, we would need to rebuild our data structure each time it performs a certain number of operations.) An example of such an execution is presented below. We first make the processes build a long chain of back_links, and then make one of them repeatedly fail insertions and traverse that chain.

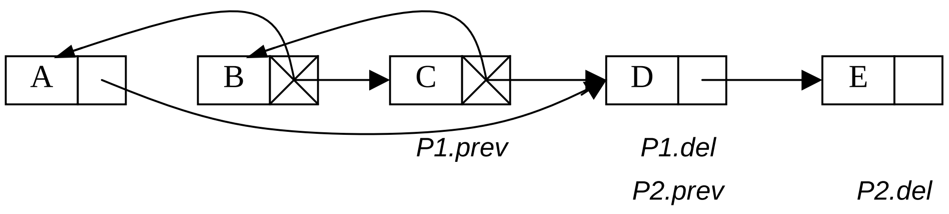
Suppose the list initially contains three nodes: A, B, C. Process P1 begins by performing a deletion of node B, and process P2 begins by performing a deletion of node C. They both locate their respective nodes at approximately the same time and are prepared to perform C&S on their successor fields: P1 is poised to mark node B by setting its back_link to point to A, and P2 is poised to mark node C by setting its back_link to point to B (Figure 7(a)). Then, suppose P2 gets delayed, while P1 marks and physically deletes node B, then inserts node D, and then starts its next operation, which is a deletion of node D. Before P2 takes any steps, P1 locates node D and is ready to mark it (see Figure 7(b)). Then P1 gets delayed, while P2 marks node C by setting its back_link to node B. Then P2 tries to physically delete node C by switching the right pointer of B from C to D, but it fails because B was marked, so it physically deletes C by switching A's right pointer from C to D. (We do not focus here on how exactly P2 locates node A). Then P2 inserts node E and starts its next operation, which is a deletion of node E. Before P1 takes any more steps, P2 locates node E and is ready to mark it (see Figure 7(c)). Notice that the system configuration in (c) looks like configuration in (b), except that P1 and P2 are reversed and the chain of back_links is one longer. By performing steps similar to the step from (b) to (c), we can make this chain arbitrarily long, with the maximum number of elements in the list never exceeding 3. It takes 1 insertion and 1 deletion to increase the length of the chain by 1, and there are 3 nodes in the list in the initial configuration, so to get a chain of length k, we would need to perform 2k + 3 operations: k + 3 insertions and k deletions.



(a) Initial configuration



(b) P1 deletes B, inserts D and starts deletion of D.



(c) P2 deletes C, inserts E, and starts deletion of E.

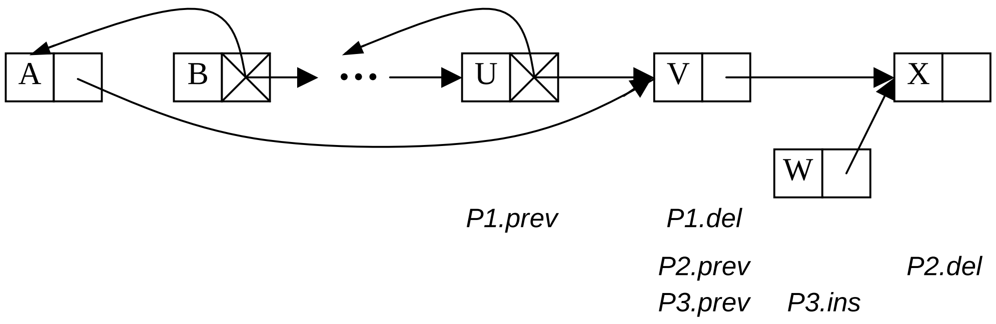
Figure 7: Building a chain of back_links.

Suppose processes P1 and P2 perform 2k + 3 operations to create a chain of back_links of length k, as discussed above (see Figure 8(a)). The chain consists of nodes

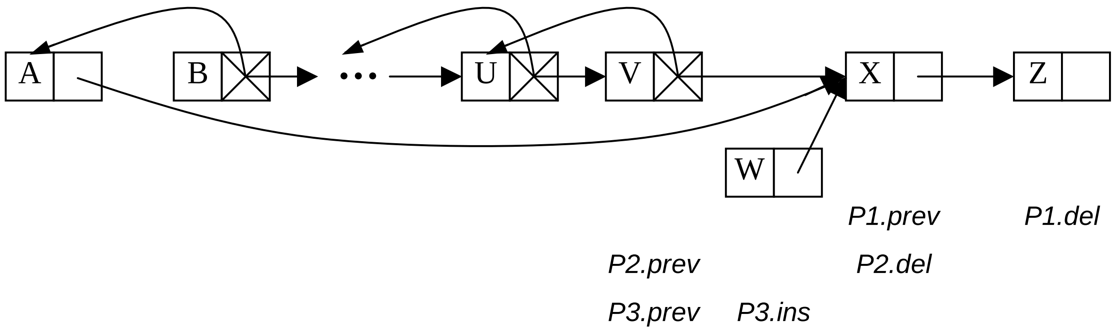
B, …, U. Process P1 is in the middle of the deletion of node V: it is about to mark V and then to try to physically delete it by switching U’s right pointer. Process P2 is in the middle of the deletion of node X: it is ready to mark X and then try to physically delete it by switching V’s right pointer. From this configuration process P3 starts the insertion of node W, locates the place to insert it, and is ready to switch V’s right pointer from X to W.

Then P2 and P3 are delayed, while P1 marks node V and physically deletes it by switching the right pointer of A from V to X. Then P1 inserts node Z, starts a deletion of node Z, and becomes delayed just when it is ready to mark Z. Then P3 tries to switch the right pointer of V from X to W, but fails because V is marked, so it has to traverse a chain of back_links to A, performing $\Omega(k)$ steps (see Figure 8(b)). Then P3 inserts node W by switching the right pointer of A from X to W, and then starts its next operation, which is a deletion of node W (see Figure 8(c)). In the next step, P3 deletes W and starts an insertion of node Y (see Figure 8(d)). Notice that the system configuration is now similar to the configuration in Figure 8(a) with the roles of P1 and P2 interchanged. The only difference is that the length of the back_links chain was increased by 1 by adding node V to it, and is now $k + 1$.

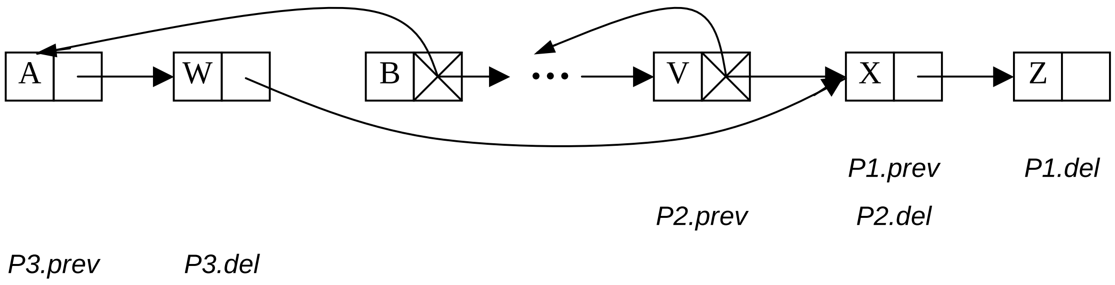
The total work done by the system in going from Figure 8(a) to Figure 8(d) is $\Omega(k)$, the number of operations performed is 4: the deletion of V, the insertion of W, the deletion of W, and the insertion of Z. Notice that at any time the number of non-deleted elements in the list does not exceed 4. Now let us consider an execution, which first creates a chain of back_links of length k and then repeats the sequence of steps shown in Figure 8 k times. The total number of operations performed by the processes would be $m = 2k + 3 + 4k = O(k)$, and the total amount of work would be $\Omega(k^2) = \Omega(m^2)$. Thus the amortized cost of each operation would be $\Omega(m)$. Notice that the whole chain of back_links of length k needs to be stored in memory after it is created, because it is constantly traversed by P3 performing insertions, so the amount of memory required by the data structure is $\Omega(k) = \Omega(m)$.



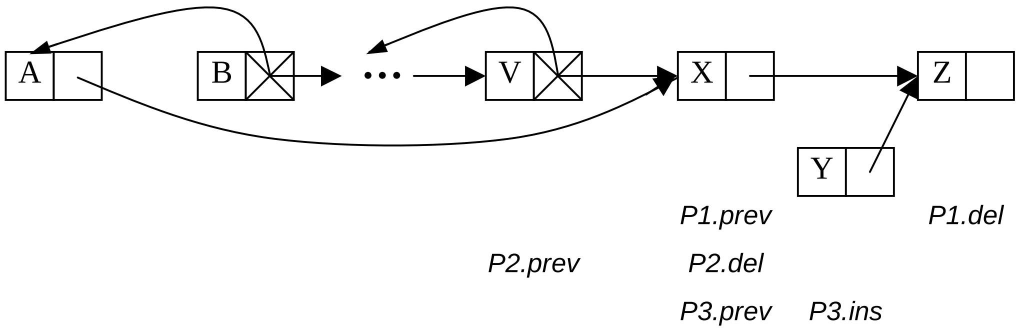
(a) P1 is deleting V, P2 is deleting X, P3 is inserting W.



(b) Step 1: P1 deletes V, inserts Z, and starts the deletion of Z,
P3 fails the insertion of W and follows a chain of back_links to A.



(c) Step 2: P3 inserts node W between A and X,
then starts the deletion of W.



(d) Step 3: P3 deletes W and starts the insertion of Y.

Figure 8: Traversing the chain of back_links.

3.1.4 Flagging successor fields

The simple introduction of back_links does not yield a data structure with good performance. The main problem is that long chains of back_link pointers can be traversed by the same process several times, which happens because these chains can grow towards the right, i.e. back_links can be set to marked nodes, and thus nodes can be linked to the beginning of the chain. We show there is a way to stop that from happening, thereby obtaining a data structure with better performance.

One of the ways to do so is to ensure that *whenever a back_link is set, it is pointing to an unmarked node*. We do this by introducing one more bit to reflect the status of the node – the *flag bit*. The flag bit, like the mark bit, is part of the successor field. While a node is flagged, its successor field is fixed and cannot be marked or otherwise changed until the flag is removed. So before marking a node, a process flags the predecessor node, thus ensuring that when the back_link is set to point to the predecessor, it will not be pointing to a marked node. The flag can be thought of as a warning that a deletion of the next node is in progress.

With introduction of the flag bits, back_link does not need to be a part of the successor field, and we make it a separate field. In our construction the successor field consists of a right pointer, a mark bit, and a flag bit. Note that we could make back_link part of the successor field, similarly to the construction in the previous subsection. The successor field would then consist of a right pointer, a back_link pointer, and a flag bit. However, this would impose a tighter upper bound on the possible size of the pointers in the data structure, because the size of the successor field cannot be greater than the size of the C&S registers.

The three-step deletion procedure is illustrated in Figure 9 and is explained below.

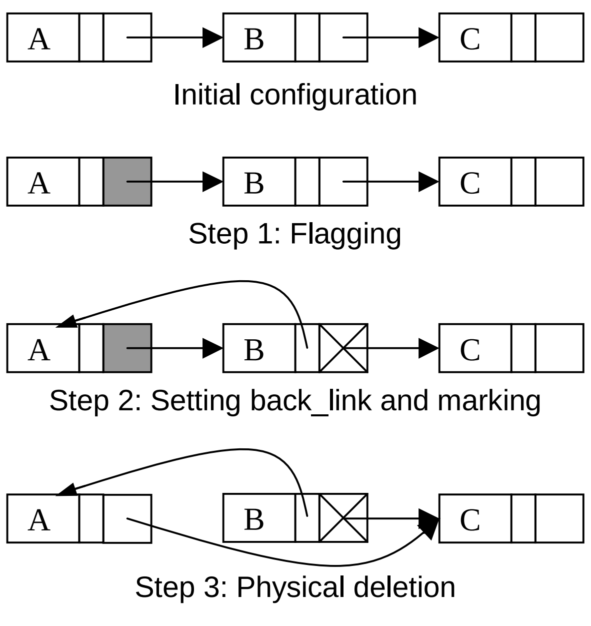

A
B
C
Initial configuration
Step 1: Flagging
Step 2: Setting back_link and marking
Step 3: Physical deletion

Figure 9: Three-step deletion of a node.

Shaded boxes denote flagged fields, and crossed boxes denote marked fields. Initially nodes A, B, and C are in the list and node B is about to get deleted (Figure 9, Initial configuration). Deletion consists of three steps. The first step is flagging the predecessor node A by applying C&S to its successor field (Figure 9, Step 1). The second step is setting B’s back_link to point to its predecessor A and then marking B by applying C&S to its successor field (Figure 9, Step 2). The third and final step is a physical deletion of node B by applying C&S to A’s successor field. This C&S switches right pointer of A from B to C and sets the flag bit of A to 0 at the same time.

By introducing flag bits we solve the problem of chains of back_links growing towards the right, because a flagged node cannot be marked, so a back_link can never be set to point to a marked node. However, we also create another problem: if a process wants to update a flagged node’s successor field, it will be unable to do so until the flag is removed. We solve this problem by allowing processes to help one another with deletions: if a process cannot complete its operation because of the flagged node, it will try to complete the corresponding deletion, thus removing the flag, and then continue with its own operation.

3.2 Algorithms

In the previous section we gave a high-level description of the deletion procedure in our data structure. In this section we provide a more detailed description of the implementation of deletions, insertions, and searches in our data structure, and we present the pseudocode for them.

3.2.1 The data structure and notation

Each node in our data structure has the following fields:

- key
- element
- back_link
- succ, which has three parts:
    - right
    - mark
    - flag

Key is the key of the node. All the nodes in the list are ordered by their keys. We assume that all the keys are distinct. (A simpler version of the data structure, that allows duplicate keys, can be obtained by removing from the Insert routine the lines that check for duplicate keys.) Element is the rest of the data to be stored in the node. Back_link is set by the deletions and used by the insertions as discussed above. If a node is not marked (or about to get marked) as deleted, its back_link is null. Succ is a successor field, which itself consists of three fields: right, mark, and flag. Right is a pointer to the next node in the list. Mark and flag are both 1-bit fields, which reflect the state of the node. If both of these fields are 0, the node is in a normal state. If flag = 1, then the node is flagged, which means that its successor node is about to get deleted from the list. If mark = 1, it means that the node is marked as deleted. Both flag and mark cannot be equal to 1 at the same time.

For simplicity, we will often omit “succ”, when refering to distinct parts of successor field of the node, i.e. we will use node.right instead of node.succ.right, node.mark instead of node.succ.mark, and node.flag instead of node.succ.flag.

In addition, there are shared variables *head* and *tail* that point to the head and to the tail nodes of the list respectively. These variables are never modified. The head node has a key of -∞, and the tail node has a key of +∞. The head and tail nodes have no elements. When the list is initialized, the head and tail nodes are the only nodes in the list, head.succ = (tail, 0, 0), and tail.succ = (null, 0, 0).

3.2.2 Pseudo-code

We present our algorithms in pseudo-code. Figures 10-17 show various routines used by our data structure. Figures 10, 12, and 13 show the pseudo code for the three *major routines*: Search, Insert, and Delete; the rest of the figures show auxiliary routines. The variables of the Node type can be either node pointers, or one of the special return values (DUPLICATE_KEY, NO_SUCH_KEY).

```
Search (Key k): Node
(curr_node, next_node) = SearchFrom(k, head)
if (curr_node.key = k)
    return curr_node
else
    return NO_SUCH_KEY
```

Figure 10: The Search routine searches for a node with the supplied key.

```
SearchFrom (Key k, Node *curr_node): (Node, Node)
next_node = curr_node.right
while (next_node.key <= k)
    while (next_node.mark == 1 && (curr_node.mark == 0 || curr_node.right != next_node))
        if (curr_node.right == next_node)
            HelpMarked(curr_node, next_node)
        next_node = curr_node.right
    if (next_node.key <= k)
        curr_node = next_node
        next_node = curr_node.right
return (curr_node, next_node)
```

Figure 11: The SearchFrom routine finds two consecutive nodes such that. the first has the key less or equal than k, and the second has the key strictly greater than k.

Insert (Key k, Elem e): Node

```
(prev_node, next_node) = SearchFrom(k, head)
if (prev_node.key == k)
    return DUPLICATE_KEY
newNode = new node(key = k, elem = e)
loop
    prev_succ = prev_node.succ
    if (prev_succ.flag == 1)
        HelpFlagged(prev_node, prev_succ.right)
    else
        newNode.succ = (next_node, 0, 0)
        result = c&s(prev_node.succ, (next_node, 0, 0),  (newNode, 0, 0))
        if (result == (newNode, 0, 0))   // SUCCESS
            return newNode
        else                             // FAILURE
            if (result == (*, 0, 1))            // failure due to flagging
                HelpFlagged(prev_node, result.right)
            while (prev_node.mark == 1)         // possibly a failure due to marking
                prev_node = prev_node.back_link
    (prev_node, next_node) = SearchFrom(k, prev_node)
    if (prev_node.key == k)
        free newNode
        return DUPLICATE_KEY
end loop
```

Figure 12: The Insert routine attempts to insert a new node into the list.

Delete(Key k): Node

```
(prev_node, del_node) = SearchFrom($k - \varepsilon$, head)
if (del_node.key != k)      // k is not found in the list
    return NO_SUCH_KEY
(prev_node, result) = TryFlag(prev_node, del_node)
if (prev_node != null)
    HelpFlagged(prev_node, del_node)
if (result == false)
    return NO_SUCH_KEY
return del_node
```

Figure 13: The Delete routine attempts to delete a node with the supplied key

**HelpMarked(Node *prev_node,
Node *del_node)**

```
next_node = del_node.right
c&s(prev_node.succ, (del_node, 0, 1), (next_node, 0, 0))
```

Figure 14: The HelpMarked routine attempts to physically delete the marked node del_node.

```
HelpFlagged(Node *prev_node, Node *del_node)
del_node.back_link = prev_node
if (del_node.mark == 0)
    TryMark(del_node)
HelpMarked(prev_node, del_node)
```

Figure 15: The HelpFlagged routine attempts to mark and physically delete the successor of the flagged node prev_node.

```
TryMark(Node del_node)
repeat
    next_node = del_node.right
    result = c&s(del_node.succ, (next_node, 0, 0), (next_node, 1, 0))
    if (result == (*, 0, 1))          // failure due to flagging
        HelpFlagged(del_node, result.right)
until (del_node.mark == 1)
```

Figure 16: The TryMark routine attempts to mark the node del_node.

```
TryFlag (Node *prev_node, Node *target_node): (Node, result)
loop
    if (prev_node.succ == (target_node, 0, 1))    // predecessor is already flagged
        return (prev_node, false)
    result = c&s(prev_node.succ, target_node, 0, 0), (target_node, 0, 1))
    if (result == (target_node, 0, 0))           // c&s was successful
        return (prev_node, true)
    /* Failure */
    if (result == (target_node, 0, 1))           // failure due to flagging
        return (prev_node, false)
    while (prev_node.mark == 1)                  // possibly failure due to marking
        prev_node = prev_node.back_link
    (prev_node, del_node) = SearchFrom(target_node.key – $\varepsilon$, prev_node)
    if (del_node != target_node)     // target_node got deleted
        return (null, false)
end loop
```

Figure 17: The TryFlag routine attempts to flag the predecessor of prev_node.

Let us start with an overview of the SearchFrom routine (Figure 11), which is used to perform the searches in our data structure. This routine takes a key and a node as its arguments. It traverses the list starting from the specified node, looking for the first node with a key strictly greater than the specified key. It returns the pointers to two consecutive nodes n1 and n2, that satisfy the following condition at some point of time during the execution of SearchFrom: n1.right = n2 and n1.key ≤ k < n2.key. SearchFrom also deletes the marked nodes along its way by calling the HelpMarked routine (line 5).

The Search routine (Figure 10) calls SearchFrom in its first line, then uses the first of the two nodes returned to determine if there is a node with key k in the list.

Now, before we move on to other routines, consider a hypothetical SearchFrom2 routine, which would be the same as SearchFrom, except that "less or equal" (≤) in lines 2 and 7 would be replaced with "strictly less" (<). Then SearchFrom2(k, n) would execute the same as SearchFrom(k – ε, n), where ε is an extremely small number (smaller than the difference between any two keys in the list). To avoid writing a separate routine, we use SearchFrom(k – ε, n) in our pseudocode to denote SearchFrom2(k, n). The keys of the two nodes that SearchFrom(k – ε, head) returns satisfy the following: n1.key < k ≤ n2.key (not n1.key ≤ k < n2.key, as would be the case with SearchFrom(k, n)).

The Insert routine (Figure 12) first calls SearchFrom to find where to insert the new key. SearchFrom returns a pair of node pointers prev_node and next_node such that prev_node.key ≤ k < next_node.key. Insert compares the key of prev_node to the new key it is trying to insert, and if they are not equal (remember that we require all the keys to be unique), it creates a new node and enters the loop in lines 5-23, from which it can only exit if it successfully inserts the new node or another process inserts a node with the same key (lines 20-22). In each iteration of the loop, the executing process will attempt to insert the node between prev_node and next_node by performing a C&S in line 11. If the C&S succeeds, Insert returns, otherwise it detects the reason for the failure, recovers from it, and enters the next iteration. The reason for the failure can only be the change of prev_node's successor pointer. There are several possible ways in which the successor pointer can change: it can get flagged, marked, redirected to another node, or any two of the above, except that it cannot be both marked and flagged. If prev_node got flagged, it means that another process was performing a deletion of the successor node. In this case Insert calls the HelpFlagged routine (lines 15-16), which helps to complete that deletion and remove the flag from prev_node. If prev_node got marked, Insert traverses the back_link pointers until it finds an unmarked node and then sets prev_node to point to it (lines 17-18). In any case, Insert performs another search in line 19, starting from prev_node, and updates its prev_node and next_node pointers. Then Insert enters the next iteration of the loop.

To obtain a simpler version of the data structure that would allow duplicate keys, it is enough to remove lines 2-3 and lines 20-22 from the Insert routine.

The Delete routine (Figure 13) performs a three-step deletion of the node, as discussed in Section 3.1. If the deletion is successful, Delete returns a pointer to the deleted node, otherwise it returns NO_SUCH_KEY. A successful deletion is linearized when the marking is performed. Delete starts by calling SearchFrom(k – ε, head) to find a node del_node and its predecessor prev_node such that prev_node.key < k ≤ del_node.key.

After SearchFrom returns, Delete checks if del_node's key is equal to k. If not, it reports that key k was not found in the list. Otherwise it calls the TryFlag routine. We will examine this routine in more detail later, for now let us say that it repeatedly attempts to flag del_node's predecessor, until it is successful or del_node gets deleted. TryFlag returns two values: a node pointer prev_node and a boolean result value. There can be three ways the TryFlag routine can return. If TryFlag flags del_node's predecessor itself, it returns a pointer to the predecessor and result = true. If TryFlag detects that another process flagged del_node's predecessor (which means that another process is performing a deletion of del_node), it returns a pointer to the predecessor and result = false. If TryFlag detects that del_node got deleted from the list, it returns null and result = false.

If prev_node returned by TryFlag is not null, Delete proceeds by calling the HelpFlagged routine, which sets the back_link of del_node pointing to prev_node, then ensures that del_node gets marked and physically deleted. If TryFlag returned result =

true, Delete then returns a pointer to the deleted node in line 9 (i.e. reports success). Otherwise, if result = false, it means that either del_node got deleted, or another process flagged del_node’s predecessor (and is going to report success). In this case Delete returns NO_SUCH_KEY.

Note that if several deletions of del_node are executed concurrently by different processes, the one that successfully flags del_node’s predecessor will report success, and the other processes will report failure. So, even though deletions are linearized at the moment of marking, the process that performs the marking does not necessarily report success.

The HelpMarked routine (Figure 14) accepts two arguments – the marked node that it helps to delete (del_node) and its flagged predecessor (prev_node). HelpMarked physically deletes del_node by swinging prev_node’s successor pointer to del_node’s successor.

The HelpFlagged routine (Figure 15) accepts two arguments – the node that it helps to delete (del_node) and its predecessor, which is flagged (prev_node). HelpFlagged starts by setting a back_link of del_node to point to prev_node. We will show later that although each back_link can be set multiple times by different processes, it is actually always set to the same value, i.e. once it is set the first time, it never changes. HelpFlagged tries to mark del_node by calling the TryMark routine, which does not exit until del_node gets marked. Finally, HelpFlagged calls HelpMarked, which physically deletes del_node.

The TryMark routine (Figure 16) accepts as its one argument the node that it should mark (del_node). TryMark does not return until it marks del_node or another process does. When the routine is called, it enters the loop in lines 1-6. In each iteration of the loop it attempts to mark del_node by performing C&S in line 3. If it is successful, it exits the loop and returns. The reason for the failure can be marking of del_node by another process, flagging of del_node, or a change of del_node’s right pointer. In case of marking, the routine exits the loop and returns. In case of flagging (lines 4-5), the routine calls HelpFlagged to remove the flag from del_node and enters the next iteration of the loop. In case of a change of del_node.right, the routine just enters the next iteration of the loop, where it updates next_node in line 2.

The TryFlag routine (Figure 17) accepts two arguments – target_node (the node whose predecessor it should flag), and prev_node (the node that was target_node’s predecessor when the process last checked). Before TryFlag completes, the target_node’s predecessor gets flagged, or target_node gets deleted. In the first case TryFlag returns target_node’s predecessor and result = true (if it flagged the predecessor itself), or result = false (if another process did). In the second case, it returns (null, false). When the routine is called, it enters the loop in lines 1-14. In each iteration of the loop it attempts to flag prev_node by performing a C&S in line 4. If it is successful, it returns (prev_node, true) (lines 5-6). If it fails due to another process flagging prev_node, and prev_node is still target_node’s predecessor, it returns (prev_node, false) (lines 7-8). If TryFlag detects that prev_node got marked, it follows the chain of back_links from prev_node and updates prev_node to be the unmarked node at the (left) end of the chain (lines 9-10). In line 11 TryFlag performs a search to find a new predecessor of target_node. If it detects that target_node got deleted, it returns (null, false) (lines 12-13). Then TryFlag enters the next iteration of the loop.

3.3 Correctness

In this section we will give a proof of correctness of our linked list implementation.

3.3.1 Invariants

To prove the correctness of our algorithms we will first show that there are several invariants that hold throughout the execution. To state the invariants we must first give a few definitions. We have already used the notions of logically and physically deleted nodes, when we talked about Harris's algorithms. We use similar classification when we define the types of nodes in our data structure. We classify nodes used by our algorithms into four types: preinserted, regular, logically deleted, and physically deleted. The definitions of these are given below.

Def 1: A *preinserted* node is a node that was created, but has not yet been inserted into the list. More precisely, it is a node referred to by newNode pointer in the Insert routine after line 4 is executed, but before the C&S in line 11 is successfully executed.

Def 2: A node is *regular* if it is unmarked, and it is not a preinserted node.

Def 3: A node is *logically deleted* if it is marked and has a regular node linked to it, i.e. n is logically deleted if n.mark = 1 and there exists a regular node m such that m.right = n.

Def 4: A node is *physically deleted* if it is marked and there is no regular node linked to it, i.e. n is physically deleted if n.mark = 1 and there is no regular node m such that m.right = n.

Notice that this classification is complete: if a node is not marked, it is either preinserted, or regular, and if a node is marked, it is either logically or physically deleted. Also note that head and tail nodes are always regular. After we state the invariants we will give alternative, equivalent definitions of the notions of the physically deleted and logically deleted nodes.

In our algorithms preinserted nodes are only the nodes referred to by the newNode pointer in the Insert routine, and only before the C&S in line 11 of that routine is executed successfully. For all of the other node pointers in our algorithms it is true that if the referred node is unmarked, it is a regular node. The preinserted nodes can only be modified by the process that created them, and we do not think of them as part of our data structure. In the future, when we refer to the nodes of the list we will mean the nodes of one of the other three types (regular, logically deleted, or physically deleted), unless stated otherwise.

Algorithm invariants given below apply to all the nodes of the list (preinserted nodes are not considered).

Inv 1: Keys are strictly sorted: For any two nodes n1, n2, if n1.right = n2, then n1.key < n2.key.

Inv 2: The union of regular and logically deleted nodes forms a linked list structure with head being the first node and tail being the last node, i.e. if n is a regular or a logically deleted node and n ≠ head, then there is exactly one regular or logically deleted node m such that m.right = n. Node m is called n's predecessor. If n ≠ tail, then m′ = n.right is a regular or a logically deleted node, which is called n's successor (it follows from Inv 1 that m′ ≠ n). The head node has no predecessors, and the tail node has no successors: tail.right = null. The head and tail nodes are always unmarked.

Inv 3: For any logically deleted node, its predecessor is flagged (and unmarked), and its successor is not marked, i.e if n is logically deleted, and m is a node of the list such that m is not physically deleted and m.right = n, then m.succ = (n, 0, 1) and (n.right).mark = 0.

Inv 4: For any logically deleted node, its back_link is pointing to its predecessor, i.e if n is logically deleted, and m is a node of the list such that m is not physically deleted and m.right = n, then n.back_link = m.

Inv 5: No node can be both marked and flagged at the same time, i.e. there is no node n such that n.succ = (*, 1, 1).

Perhaps a more intuitive way to define a notion of a physically deleted node would be to say that a node is physically deleted if there is no way to get to it from a regular node by following only right pointers.

Def 3′: A node is *logically deleted* if it is marked, and one can get to it from some regular node by following a chain of right pointers, i.e n is logically deleted if n.mark = 1 and there exists a set of nodes m_1, m_2, …, m_k such that m1 is a regular node and m_1.right = m_2, m_2.right = m_3,…, m_{k-1}.right = m_k.

Def 4′: A node is *physically deleted* if it is marked, and one cannot get to it starting from a regular node by following a chain of right pointers, i.e n is physically deleted if n.mark = 1 and there is no set of nodes m_1, m_2, …, m_k, such that m_1 is a regular node and m_1.right = m_2, m_2.right = m_3,…, m_{k-1}.right = m_k.

Definitions 3 and 3′, as well as definitions 4 and 4′, are equivalent. The equivalence follows from Inv 3. (To show that Def 3 follows from Def 3′, apply an induction on the length of the chain m_1, m_2, …, m_k to prove that if m1 is regular, then all the nodes in this chain are either regular or satisfy the property stated in Def 3.)

Now we will prove the invariants. We will start by proving Inv 5, as it is the easiest one. Then we will prove invariants 1-3 by proving that they are preserved by all the C&S's performed by our algorithms. Finally, we will prove Inv 4. Along the way we will prove several useful propositions and lemmas. A few of these propositions and lemmas will be used in the analysis section, and so they will include some claims that are not required to prove the invariants themselves.

Theorem 1: Invariant 5 always holds.

Proof:

The invariant obviously holds for an empty list. When a new node is created, its successor pointer is set to be unflagged and unmarked (line 10 in Insert), and it does not change until the node is inserted into the list. The successor pointers of the nodes that are part of the list can only be modified by one of the four C&S operations: line 11 in Insert, line 2 in HelpMarked, line 3 in TryMark, and line 4 in TryFlag. None of them makes the successor field both marked and flagged, and therefore invariant 5 always holds. ■

Proposition 2: Once a node is marked, its successor field never changes.

Proof:

It is easy to see that none of the four C&S's can change a marked successor field. ■

Proposition 3 (weak SearchFrom postconditions): Suppose SearchFrom(k, n) is executed, and suppose n.key ≤ k. Let (n1, n2) be the pair of nodes SearchFrom returns.

Then n1.key ≤ k < n2.key, and there exists a point of time during the execution of the routine such that at that time n1.right = n2 and SearchFrom does not try to perform any C&S's on n1.succ after that time.

Proof:

The first of the two nodes returned in line 10 (n1 = curr_node) is either equal to n or was set in line 8. In the first case, since n.key ≤ k, n1.key ≤ k. In the second case, since line 8 is only executed if condition in line 7 is true, n1.key ≤ k as well. Since the loop in lines 2-9 exits only when the condition in line 2 is false, we can conclude that n2.key > k. Let T1 be the moment of time when variable next_node is last assigned a value. That can be in line 1, 6, or 9. In any case, next_node is assigned a value of curr_node.right, and SearchFrom performs no C&S's on n1 after that moment. Also, at T1 curr_node = n1, because the value of curr_node can only be changed in line 8, and if it is changed, next_node is changed as well. So at time T1, n2 = next_node = n1.right. ■

We are going to prove that invariants 1-3 always hold by proving that they are preserved by all the C&S's performed by our algorithms. We start by proving this for the C&S in the Insert routine.

Proposition 4: The C&S in line 11 of the Insert routine preserves invariants 1-3.

Proof:

The C&S is *result = c&s(prev_node.succ, (next_node, 0, 0), (newNode, 0, 0)).*
A successful execution of this C&S swings the right pointer of prev_node from next_node to newNode.

Before this C&S is executed, there are no other nodes linked to newNode, because newNode was created by this invocation of Insert in line 4, and only this invocation can insert newNode into the data structure, and then it is poised to exit in line 12 without any further attempts to perform a C&S. Therefore, before the C&S is executed, newNode is a preinserted node. Also, it is not marked or flagged. After the execution of this C&S there is exactly one node linked to newNode, and newNode is still not marked, thus it becomes a regular (unmarked) node of the list. Let us prove the proposition separately for each of the three invariants.

1. Since newNode.right is set pointing to next_node in line 10, to prove that the keys remain sorted after the execution of this C&S, it is sufficient to prove that prev_node.key < newNode.key < next_node.key. Prev_node and next_node were returned either by SearchFrom in line 1, if this is the first iteration of the loop, or by SearchFrom in line 19. In either case we know from Proposition 3 that prev_node.key ≤ k < next_node.key. Since in lines 2-3 and in lines 20-22 we ensure that prev_node.key ≠ k, and since newNode.key = k (line 4), this yields what we have to prove.
2. To prove that Inv 2 is preserved, it is sufficient to prove that after the execution of this C&S it holds for newNode and next_node. After the C&S there is exactly one node of the list linked to newNode, and that is prev_node, which is a regular node, so Inv 2 holds for newNode. Since Inv 2 held for next_node before the execution of the C&S, the only regular or logically deleted node linked to next_node was prev_node. Thus after this C&S the only regular or logically deleted node linked to next_node is newNode, and thus Inv 2 holds for next_node as well.

3. Prev_node was not flagged before the C&S, so since Inv 3 held, next_node was not marked before the C&S. Thus, next_node is not marked immediately after the C&S is executed. Prev_node and newNode are also not marked after the C&S, so since the third invariant held before the C&S, and none of the successors or predecessors of the other nodes changed, the invariant holds after the C&S as well. ■

Before we are ready to prove that the other C&S's preserve invariants 1-3, we must prove a couple of the auxiliary propositions about the HelpMarked routine.

Proposition 5: If the HelpMarked routine is invoked with parameters prev_node and del_node, then del_node was marked at some point before this invocation.

Proof:

The HelpMarked routine can only be called from SearchFrom in line 5, or from HelpFlagged in line 4. If HelpMarked(prev_node, del_node) was called from SearchFrom in line 5, then the condition in line 3 had to be true, so del_node was marked at that moment. If HelpMarked(prev_node, del_node from HelpFlagged in line 4, then prior to this call either TryMark(del_node) was called in line 3 or del_node was already marked. It is easy to see by looking at the code of TryMark routine, that the routine does not exit until del_node is marked. So in any case, before HelpMarked(prev_node, del_node) is invoked, del_node is marked. ■

Lemma 6 (Physical deletion): Suppose HelpMarked(prev_node, del_node) successfully executes a C&S in line 2. Then if invariant 2 held before this C&S, del_node is physically deleted immediately after the C&S.

Proof:

By Proposition 5 del_node is marked before the execution of the C&S, and by Proposition 2 it stays marked. By Inv 2 there cannot be more than one regular node linked to del_node before the C&S is executed. Thus the only non-marked node linked to del_node immediately before the C&S is prev_node. After the execution of this C&S, there are no more regular nodes linked to del_node, del_node is marked, and thus del_node becomes physically deleted. ■

The following two propositions prove that the C&S's performed by the HelpMarked and TryFlag routines preserve invariants 1-3.

Proposition 7: The C&S in line 2 of the HelpMarked routine preserves invariants 1-3.

Proof:

The C&S is *c&s(prev_node.succ, (del_node, 0, 1), (next_node, 0, 0)).*

A successful execution of this C&S swings the right pointer of prev_node from del_node to next_node and removes the flag from prev_node. Just before a successful C&S, prev_node is unmarked and linked to del_node. Del_node is marked, and thus del_node was a logically deleted node before the C&S. By Lemma 6, if the C&S is executed successfully, del_node becomes physically deleted. Also, since by Proposition 5 del_node was marked before HelpMarked was called, marked successor fields do not change, and del_node.right = next_node when line 1 of HelpMarked is executed, it follows that del_node.right = next_node immediately before the C&S is executed. Now let us prove the proposition separately for each of the 3 invariants.

1. Before the C&S prev_node.right = del_node and del_node.right = next_node. Since Inv 1 held before the C&S, prev_node.key < next_node.key. Thus, after the C&S is executed, Inv 1 still holds.
2. Del_node becomes physically deleted after the execution of this C&S, and prev_node becomes linked to next_node. Since the structure of the rest of the list does not change, Inv 2 holds after the execution of this C&S.
3. Del_node becomes physically deleted, so there is no condition imposed on its predecessor. Before the C&S, del_node is a logically deleted node and del_node.right = next_node, thus by Inv 3, next_node is not marked. Prev_node is also not marked immediately after the C&S, and thus Inv 3 holds. ■

Proposition 8: The C&S in line 4 of the TryFlag routine preserves invariants 1-3.

Proof:

The C&S is *c&s(prev_node.succ, (target_node, 0, 0), (target_node, 0, 1)).*

A successful execution of this C&S flags prev_node. The flag field of any node is not relevant to invariants 1 or 2. Flagging a node cannot make invariant 3 false, thus the invariants will hold after the execution of this C&S. ■

To prove that Inv 1-3 always hold, we now only need to show that the C&S in the TryMark routine does not violate these invariants. In particular, we need to prove that this C&S does not violate Inv 3. Thus, we need to show that when a node gets marked, its predecessor is flagged. The next lemma will help us to prove this.

Lemma 9 (HelpFlagged is invoked only if a flagged node is detected): Suppose process P invokes the HelpFlagged routine with parameters prev_node = n and del_node = m during the execution M of some major routine (Search, Insert, or Delete). Then there exists time T during the execution M and before the invocation of HelpFlagged, when n.succ = (m, 0, 1). Also, P does not try to perform any C&S's on n between T and the moment it calls HelpFlagged.

Proof:

HelpFlagged can be called in line 6 of the Delete routine, in line 8 or 16 of the Insert routine, or in line 5 of the TryMark routine.

Suppose it was called from the Delete routine. Then TryFlag, called in line 4 must have returned a non-null prev_node, which means that TryFlag returned in line 3, 6, or 8. If TryFlag returned in line 3, then the moment when it executed the previous line (line 2) is a valid moment for T. If TryFlag returned in line 6 or 8, then the moment just after it last tried to perform a C&S in line 4 is a valid moment for T.

Suppose HelpFlagged was called from line 8 of the Insert routine. Then n = prev_node and m = prev_succ.right. At the moment when Insert executed line 6 for the last time before calling HelpFlagged, n.right = prev_succ.right = m and n.flag = prev_succ.flag = 1. Since nodes cannot be both marked and flagged at the same time, this is a valid moment for T.

If Insert called HelpFlagged in line 16, then n = prev_node and m = result.right. When Insert tried to perform the C&S in line 11 for the last time before calling HelpFlagged, n.succ = result = (m, 0, 1), and this is a valid moment for T.

Finally, suppose HelpFlagged was called in line 5 of the TryMark routine. This case is very similar to the previous case: when TryMark tried to perform the C&S in line 3 for the last time, n.succ = (m, 0, 1), and this is a valid moment for T. ■

TryMark is the only routine that marks nodes. The only place the TryMark routine is called is line 3 of the HelpFlagged routine. In the previous lemma we showed that if HelpFlagged was invoked, prev_node was flagged at some point of time before the invocation. Now we will prove that prev_node stays flagged at least until its successor gets marked.

Lemma 10 (Predecessor is still flagged when a node gets marked): Suppose the C&S in line 3 of the TryMark routine successfully marks del_node. Let v be the first parameter of the HelpFlagged routine that called this TryMark routine. Then starting from some time before HelpFlagged was invoked, and until the C&S in TryMark is performed, v.succ = (del_node, 0, 1).

Proof:

Let C1 be the C&S in question. From Lemma 9 it follows that at some point of time T before the HelpFlagged(v, del_node) that called this TryMark was invoked, v.succ = (del_node, 0, 1). Let us prove that v.succ does not change before C1 is performed.

Suppose there was a C&S C2 that changed it (let us take the first such C&S after T if there are several of them). The only C&S that can change a flagged successor field is the C&S in the HelpMarked routine. Since immediately before C2, v.succ = (del_node, 0, 1), and C2 was successful, HelpMarked had to be called with parameters v and del_node. By Proposition 5, del_node had been marked before the HelpMarked routine was called. Marked nodes never become unmarked (Proposition 2), so del_node is still marked when C1 is performed. But C1 would fail if del_node was marked, and we know it is successful – a contradiction. ■

Proposition 11: The C&S in line 3 of the TryMark routine preserves invariants 1-3.

Proof:

The C&S is *result = c&s(del_node.succ, (next_node, 0, 0), (next_node, 1, 0)).*

A successful execution of this C&S marks del_node. Suppose Inv 1-3 hold before the C&S. By Lemma 10 when this C&S is executed, there exists a node v such that v.succ = (del_node, 0, 1), i.e. there is an unmarked node linked to del_node. Thus, immediately after this C&S is successfully executed del_node becomes logically deleted. Also, since the C&S succeeded, del_node was not flagged, and thus, by Inv 3, next_node was not marked before the C&S. Now let us prove that all three invariants hold after the C&S.

1. No right pointers are changed, so Inv 1 holds.
2. Immediately after the C&S del_node becomes logically deleted. Del_node's successor next_node was not marked before the C&S, and thus it was a regular node immediately before the C&S, and it remains a regular node immediately

after the C&S as well. The status of other nodes is not affected by this C&S, and no right pointers change, so Inv 2 holds.

3. By Lemma 10 when C&S is executed, prev_node.succ = (del_node, 0, 1), so Inv 3 is satisfied for del_node. Also, next_node was not marked before the C&S, and therefore it is not marked immediately after the C&S, so the invariant is satisfied for del_node's successor. Thus, Inv 3 holds. ■

Theorem 12: Invariants 1-3 always hold.

Proof:

Initially the list contains no keys and all the invariants obviously hold. Our algorithms modify the data structure only by performing C&S operations or by setting back_links in line 1 of HelpFlagged routine. Notice that setting a back_link cannot violate any of the invariants 1-3. There are 4 different types of C&S's performed by our algorithms: the C&S in the Insert routine, the C&S in TryFlag routine, the C&S in TryMark routine, and the C&S in HelpMarked routine. By Propositions 4, 7, 8, and 11 none of these four C&S operations can violate invariants 1-3, so they always hold. ■

We proved that invariants 1, 2, 3, and 5 always hold, and the only invariant left is invariant 4. Before we will be ready to prove it, we need to prove several auxiliary claims.

Proposition 13: Once a node is physically deleted, it remains physically deleted forever.

Proof:

A marked node can never become unmarked, so a node can stop being physically deleted only if the right pointer of an unmarked node is set pointing to it. Out of the 5 possible modifications of our data structure only 2 change right pointers: the C&S in the Insert routine and the C&S in the HelpMarked routine.

The C&S in the Insert routine is *result = c&s(prev_node.succ, (next_node, 0, 0), (newNode, 0, 0)).* First note that newNode is obviously not marked when this C&S is performed. Also note that since the C&S is successful, prev_node was not flagged, and thus by Inv 3, next_node was not marked, when this C&S was performed. Therefore, no physically deleted nodes can be reinserted into the list by this C&S.

The C&S in the HelpMarked routine is *c&s(prev_node.succ, (del_node, 0, 1), (next_node, 0, 0)).* In order to prove that this C&S cannot reinsert physically deleted nodes into the list, it is sufficient to show that if it is successful, then immediately before it is performed, next_node is not a physically deleted node. Immediately before the C&S del_node is a logically deleted node, because by Proposition 5 it is marked, and, since the C&S is successful, its predecessor is not marked. Therefore, by Inv 3, next_node is not marked, and thus it cannot be a physically deleted node. ■

Lemma 14: For any node m, after some flagged node n is linked to it, m will never have a regular predecessor other than n. When n's successor field changes, m becomes physically deleted.

Proof:

By Inv 2 the node m cannot have two regular predecessors, so before it can have a regular predecessor other than n, n must have its successor field changed. The only C&S that changes successor fields of flagged nodes is the C&S in the HelpMarked routine. By Lemma 6 we know that if this C&S successfully changes n's successor field, m becomes physically deleted, and by Proposition 13 it remains physically deleted forever, so there can be no regular nodes linked to it after that. ■

Proposition 15: Once a back_link is set, it never changes.

Proof:

Suppose process P set the back_link of del_node to prev_node, and process P′ set the back_link of del_node to prev_node′. Proving that prev_node = prev_node′ proves the proposition. Since back_links are set only in the HelpFlagged routine, we can conclude that at some point HelpFlagged(prev_node, del_node) was called by P, and at some point HelpFlagged(prev_node′, del_node) was called by P′. By Lemma 9 this means that at some moment of time prev_node.succ = (del_node, 0, 1), and at some other moment of time prev_node′.succ = (del_node, 0, 1). Then by Lemma 14 prev_node = prev_node′. ■

We are now ready to prove that Inv 4 always holds.

Theorem 16: Inv 4 always holds.

Proof:

This invariant obviously holds in the beginning when the list is empty. Let us prove that if it holds before a modification, it will hold afterwards as well. There are five types of modifications that can be performed by our algorithms: four types of (successful) C&S's and setting the back_link in line 1 of HelpFlagged routine. We will prove that none of these modifications can violate Inv 4.

- The C&S in the Insert routine is
  result = c&s(prev_node.succ, (next_node, 0, 0), (newNode, 0, 0)).
  Immediately before the C&S prev_node is not flagged, and thus by Inv 3, next_node is not marked. Also, when the C&S is performed, newNode is not marked. Thus this C&S preserves Inv 4.
- The C&S in the HelpMarked routine is
  c&s(prev_node.succ, (del_node, 0, 1), (next_node, 0, 0)).
  By Proposition 5 del_node was marked before HelpMarked was called, and thus from Proposition 2 it follows that next_node is del_node's successor immediately before the C&S. Also, from Proposition 5 it follows that immediately before the C&S del_node is a logically deleted node. Therefore, by Inv 3 next_node is not marked immediately before the C&S. By Lemma 6 del_node becomes physically deleted after this C&S, and since next_node is not marked, Inv 4 holds.
- The C&S in the TryFlag routine is
  result = c&s(prev_node.succ, (target_node, 0, 0), (target_node, 0, 1)).
  This C&S flags the successor field of prev_node, which cannot violate Inv 4.
- The C&S in the TryMark routine is
  result = c&s(del_node.succ, (next_node, 0, 0), (next_node, 1, 0)).

Before the C&S, del_node is a regular node, and after the C&S it becomes a logically deleted node (since by Lemma 10 it has an unmarked predecessor), so we must show that it has a correct back_link.

Let v be the predecessor of del_node at the time of the C&S. Lemma 10 implies that TryMark was called from the HelpFlagged routine with the first parameter prev_node = v. Therefore, when that HelpFlagged executed the first line, del_node's back_link was set to v. From Proposition 15 we know that back_links do not change, and thus when C&S was executed, del_node's back_link was pointing to its predecessor. Thus, Inv 4 holds after the C&S.

- Back_links are set only in line 1 of the HelpFlagged routine:
  del_node.backlink = prev_node.

Let us prove by contradiction that this modification does not violate the invariant. Suppose it did. Then del_node was a logically deleted node when this modification was performed. Since the invariant held prior to the modification, and del_node was logically deleted, it means that del_node's back_link was set to a correct node before this modification. But by Proposition 15 once back_link is set, it never changes, so this modification could not violate the invariant. ■

3.3.2 Critical steps of a node deletion

Just after a node is inserted into the list, it is a regular node. We show that the deletion of a node must proceed in three steps.

Def 5: There are three types of successful C&S's that are said to be the *critical steps of the deletion* of del_node. These are:

1. The C&S that flags del_node's predecessor.
2. The C&S that marks del_node.
3. The C&S that unflags del_node's predecessor and physically deletes del_node.

Proposition 17 (Critical steps of a deletion): The three C&S steps described in Definition 5 can be successfully performed only once for each particular node del_node and only in the order they are listed.

Proof:

First, just by looking at the C&S's that are performed by our algorithms, it is easy to see that step 1 can be performed only by the C&S in line 4 of the TryFlag routine, step 2 can be performed only by the C&S in line 3 of the TryMark routine, and step 3 can be performed only by the C&S in line 2 of the HelpMarked routine. Let us show that each of these steps can be performed only once for each particular node del_node.

After step 1 takes place in line 4 of TryFlag, prev_node.succ = (del_node, 0, 1). To perform this step again, the C&S in line 4 of the TryFlag routine must succeed, which means that there must exist a node prev_node′ such that prev_node′.succ = (del_node, 0, 0). By Inv 2, two nodes cannot be linked to del_node at the same time, and prev_node's successor does not change as long as prev_node is flagged. Thus, prev_node would have to be unflagged before prev_node′ is linked to del_node. But, by Lemma 14, changing the successor field of prev_node physically deletes del_node and del_node would then stay physically deleted forever by Proposition 13, and thus the unmarked node prev_node′ cannot be linked to del_node. Thus, step 1 cannot be performed more than once.

Immediately before step 2 is performed, del_node is unmarked. After step 2 takes place, del_node becomes marked, and by Proposition 2 it stays marked forever. Thus, this step cannot be performed more than once.

Immediately before step 3 is performed del_node is not physically deleted, since there exists a regular node prev_node linked to del_node: prev_node.succ = (del_node, 0, 1). By Lemma 6, after step 3 takes place, del_node becomes physically deleted. By Proposition 13, it stays physically deleted forever. Thus, step 3 cannot be performed more than once.

Now let us show that these steps can only be performed in the order they are listed in Definition 5. By Lemma 10, del_node's predecessor prev_node gets flagged before del_node gets marked, thus step 2 can be performed only after step 1 is performed. Step 3 can be performed only in the HelpMarked routine, and by Proposition 5 HelpMarked can be invoked only after a node is marked. Thus step 3 can be performed only after step 2 is performed. ■

A given invocation of the Delete routine starts a deletion operation. However, it is not necessary that all three critical steps of a deletion operation are performed by the Delete routine that started it: other processes may *help* to perform some of the critical steps. Also, while a process is executing a Delete routine, it may apply some changes to the data structure, such as helping other deletions, which are unrelated to the deletion operation that was started by the routine. So, there is a difference between the notions of “the Delete routine” and “the deletion operation”, although there is a one-to-one correspondence between the two. A similar thing can be said about the Insert routine vs. the insertion operation and the Search routine vs. the search operation.

3.3.3 The SearchFrom routine

In this subsection we investigate the properties of the SearchFrom routine, and prove a few useful propositions concerning it. We start by proving a utility proposition about marked nodes.

Proposition 18: If n1 is a marked node and n.1right = n2, then either n2 is unmarked, or n1 got marked before n2.

Proof:

Marked pointers never change, so when n1 was marked, it had to be pointing at n2. After n1 was marked, it became a logically deleted node, and by the third invariant its successor had to be unmarked at that moment. So, when n1 got marked, n2 was unmarked. Thus, n2 is either unmarked, or got marked after n1. ■

The following proposition states an important property of the SearchFrom routine, which we will use to prove the SearchFrom postconditions. We will also use this proposition later, when we perform a performance analysis of our data structure.

Proposition 19 (SearchFrom property): Suppose the SearchFrom routine is executed. Let n_1, n_2, …, n_s be the sequence of nodes curr_node points to during the

SearchFrom execution. Suppose curr_node gets set to n_i at time T_i for $1 \le i \le s$, and suppose some node n_j is not marked or was not in the list at time $T_j' \le T_j$. Then nodes n_j, …, n_s were either not in the list or were in the list and unmarked (i.e. regular) at T_j'.

Proof:

Suppose there are some nodes in the sequence n_j, …, n_s which were in the list and marked at T_j'. Let n_k be the first such node. We know that $k \ne j$, because n_j was not marked or not in the list at T_j'. Therefore, n_k is not the first node in the sequence n_j, …, n_s and node n_{k-1} also belongs to this sequence. We shall show that SearchFrom could not move its curr_node pointer from n_{k-1} to n_k.

Since n_{k-1} and n_k are two consecutive nodes in the sequence, there had to be a moment T during the execution of SearchFrom, when curr_node = n_{k-1} and next_node was assigned a value n_k. At that moment, SearchFrom could be executing line 1, line 6, or line 9. In any case, at that moment n_k = next_node = curr_node.right = n_{k-1}.right. Since n_k was marked at moment T_j', it is marked at time T as well, because $T_j' \le T_j \le T_{k-1} < T$. Let us show that n_{k-1} cannot be marked at T. If n_{k-1} was marked at this moment, then it must have got marked after T_j' (because n_k is the first node in the sequence n_j, …, n_s that was marked before T_j'), which means that n_k got marked before n_{k-1}, contradicting Proposition 18. So, n_{k-1} has to be unmarked at time T.

By the third invariant at time T, n_{k-1}.succ = (n_k, 0, 1), and by Lemma 14, once n_{k-1}'s successor field changes, n_k becomes physically deleted. Let us take the first moment T′ after time T when SearchFrom executes line 3. Note that SearchFrom performs no curr_node or next_node pointer updates between T and T′, so curr_node = n_{k-1} and next_node = n_k at T′.

If n_{k-1}.succ changed between T and T′, then by Lemma 14 n_k became physically deleted between T and T′. Also, when n_k got physically deleted, n_{k-1}.right was set to n_k's successor, and after that by Proposition 13 n_{k-1}.right could never be set back to n_k. So, at time T′ n_{k-1}.right != n_k, and therefore SearchFrom will enter the while loop in lines 3-6, and update its next_pointer in line 6. After that SearchFrom has no local pointer set to n_k, and n_k is physically deleted, so there is no way SearchFrom can ever set its curr_node pointer to n_k – a contradiction.

If n_{k-1}.succ is still equal to (n_k, 0, 1) at time T′, then SearchFrom will enter the while loop in lines 3-6. In lines 4-5 it will make sure that next_node gets physically deleted (if n_k.right doesn't change, n_k will be physically deleted by HelpMarked in line 5, and if it does, then n_k gets physically deleted by Lemma 14), and then in line 6 it will update its next_node pointer. As in the previous case, SearchFrom will never be able to set its curr_node pointer to n_k after that – a contradiction. ■

The following proposition states the postconditions of the SearchFrom routine. It is an extension of Proposition 3.

Proposition 20 (SearchFrom postconditions): Suppose SearchFrom(k, n) is executed, and suppose n.key ≤ k. Let (n1, n2) be the pair of nodes SearchFrom returns. Then the following conditions are true.

- n1.key ≤ k < n2.key.

- There exists a point of time during the execution of SearchFrom when n1.right = n2, and after that time SearchFrom does not call any C&S operations on n1.succ.
- For any time T before or when SearchFrom is invoked the following is true: if n.mark = 0 at T, then there exists a point of time T′ between T and the moment SearchFrom returns, when n1.right = n2 and n1.mark = 0.

Proof:

The first two statements were proved in Proposition 3. Let us prove the third statement. Let T1 be the time, specified by Proposition 3. If n1 is not marked at time T1, then T1 is a valid time for T′, and we are done with the proof. Suppose n1 got marked at some time T2 < T1. Since curr_node = n when SearchFrom is invoked and curr_node = n1 just before SearchFrom returns, it follows from Proposition 19 that n1 was either not in the list (i.e. preinserted or not created), or was unmarked at T. Therefore, T < T2 < T1. Since successor fields of marked nodes do not change, and we know that at time T1 n1.right = n2, we can conclude that at time T2 n1.right = n2 as well, so just before T2 we have n1.mark = 0 and n1.right = n2. This is a valid moment for T′. ■

3.3.4 Linearization points and correctness

Our data structure allows three types of dictionary operations: searches, insertions, and deletions, which are executed by invoking the Search, Insert, and Delete routines respectively. There is a one-to-one correspondence between each operation performed by our data structure and each execution of one of the major routines (i.e. Search, Insert, or Delete). An execution of an operation starts with the invocation of the corresponding major routine and ends when that routine returns. (Also, the linearization point of an operation is the same as the linearization point of the corresponding execution of a major routine).

Consider any concurrent execution of our algorithms. In this subsection we show how to assign linearization points to operations performed during an execution and prove that our algorithms implement the dictionary operations. We say that the set of elements currently stored in the dictionary is the set of the elements of the regular nodes of our data structure. We show that this set is modified only at the linearization points of the insertions and deletions according to the specifications of these operations. We also prove that if an operation completes, it returns a correct result, according to its point of linearization.

We will start by assigning linearization points to the deletions, but before we do this, we need to prove the following proposition for the TryFlag routine.

Lemma 21 (TryFlag invariant): Suppose TryFlag(n, m) is called. Then at any time during its execution, prev_node.key < m.key and target_node = m.

Proof:

The value of target_node never changes, so target_node = m. Notice that TryFlag can only be called in line 4 of the Delete routine, so n and m were returned by SearchFrom in line 1 of Delete, and therefore n.key < m.key by Proposition 20. So, at the beginning of the execution of TryFlag, prev_node.key < m.key. The value of prev_node can be changed only in line 10 or 11. In the first case the key of prev_node decreases (follows from invariants 1 and 4), and in the second case prev_node.key ≤

target_node.key – ε < m.key by Proposition 20, so prev_node.key is always strictly less than m.key. ■

In the following theorem we assign linearization points to the executions of the Delete routines. We will linearize each unsuccessful deletion at some moment when no regular node has the key it searches for. We will linearize each successful deletion at the moment when a (regular) node with the key it searches for gets marked. Our algorithms are designed in such a way that a deletion of some node n reports success only if it performs the first critical step of n's deletion (flagging n's predecessor). A successful deletion does not necessarily perform the second step (marking) itself, another process may do it. We also define a mapping that will help us prove that an element can be deleted from the dictionary only by executing an appropriate deletion.

Theorem 22 (Delete correctness): If an execution of the Delete(k) routine returns NO_SUCH_KEY (indicating an unsuccessful deletion), then for this execution we can choose a linearization point, at which there was no regular node with key k in the data structure. If an execution of the Delete(k) routine returns a pointer to a node (indicating a successful deletion), then for this execution we can choose a linearization point, at which this node became marked.

Furthermore, there exists a mapping σ of all marked nodes to the successful Delete executions and non-terminated Delete executions such that

1. σ is injective.
2. For any successful execution D of the Delete routine, D = σ(n) if and only if the node D returns is n.
3. At any time T, if node n is marked at T, then σ(n) is linearized at the moment when n got marked.

Proof:

Let P be the process that is executing the deletion. We choose the linearization point for the execution depending on how it runs.

Case 1: Suppose Delete returns NO_SUCH_KEY in line 3. This is a case when Delete could not find a node with key k in the list. Let n1 and n2 be the values of prev_node and del_node SearchFrom returns in line 1. Since head is an unmarked node, by Proposition 20 there was a moment T′ during the execution of that SearchFrom when n1.right = n2, n1.mark = 0, and n1.key < k ≤ n2.key. We linearize Delete at T′. Since n2.key ≠ k (line 2), it follows that n1.key < k < n2.key. At the linearization point node n1 is unmarked, thus n1 is a regular node of the list, and since n1.right = n2, n2 is either a regular or a logically deleted node. Since the union of regular and logically deleted nodes is a sorted linked list, and since at the linearization point the value k is between the keys of the two consecutive nodes of this list, we can conclude that there is no regular node with key k in our data structure at the linearization point of the deletion.

Case 2: Suppose the routine returns NO_SUCH_KEY in line 8. This is a case when Delete found a node with the required key, but still failed to delete it, and we must linearize Delete at a moment when there was no regular node with key k in the data structure. Let (n1, r) be the values that were returned by the TryFlag routine in line 4, and let n2 be the value of del_node. Since Delete returns in line 8, n2.key = k and r = false. We will prove that there was a point of time T during Delete's execution, when n2 is a logically deleted node. We will linearize Delete at that moment, and then we will prove that this is a valid linearization, i.e. that at this moment there is no regular node with key k in our data structure.

First notice that as in Case 1, there was some moment of time T′ during the execution of the deletion, when n2 had a regular node linked to it, i.e. n2 was either a regular or a logically deleted node. Suppose n2 was a regular node at T′. Since TryFlag returned result = false, it had to return in line 3, 8, or 13.

Case 2(a): TryFlag returns in line 13. This is a case when n2 gets deleted before TryFlag can flag its predecessor. Let us examine the last SearchFrom in line 11 which TryFlag executed before returning in line 13. Let (n3, n4) be the values of prev_node and del_node that search returned, and let n3′ be the value of prev_node immediately before that search was executed. At moment T2, when line 9 in TryFlag was executed for the last time before that SearchFrom, n3′ = prev_node was not marked. Also, by Lemma 21, n3′.key < target_node.key = n2.key (and thus, n3′.key < n2.key – ε). Therefore, by Proposition 20, there is some moment T″ during the execution of TryFlag (between T2 and the moment when SearchFrom returned), when n3.right = n4 and n3.mark = 0. Also, by Proposition 20, n3.key ≤ n2.key – ε < n4.key, or, equivalently, n3.key < n2.key ≤ n4.key. By invariants 1 and 2, all the regular and logically deleted nodes form a linked list with strictly ordered keys. Since n3 and n4 are in this list at T″ (because n3 is not marked and n3.right = n4), n3.key < n2.key ≤ n4.key, and n4 ≠ n2 (line 12), n2 is not a regular or a logically deleted node at T″. Since T″ > T′, node n2 is physically deleted at T″. Thus, by Proposition 17, there was a moment T between T′ and T″, when n2 was a logically deleted node

Case 2(b): TryFlag returns in line 3 or 8. This is the case when some other process Q flags the predecessor of n2. Q is also executing a Delete(k) routine and will report success (or die before finishing its deletion). Process P’s Delete ensures that all critical steps of the deletion are performed and reports failure. Let us prove that in this case there also exists a moment T during the execution of P’s deletion, when n2 is a logically deleted node.

TryFlag returned in line 3 or 8, so n1 ≠ null, and thus before Delete returns in line 8, it calls HelpFlagged(n1, n2) in line 6. If n2 is not marked, HelpFlagged calls TryMark(n2), which does not exit until n2 is marked. So n2 is a marked node at some point of time T″ during the execution of the TryMark routine. Therefore, by Proposition 17, there exists a moment T between T′ and T″, when n2 is logically deleted.

So, regardless of how TryFlag returned, there is a time T during the execution of Delete when n2 is a logically deleted node, and n2.key = k. So by Inv 1 and 2 there are no regular nodes with key k in the list, and thus we can choose T as the linearization point for the deletion.

Case 3: Suppose the Delete routine returns del_node in line 9. As in the previous case, let (n1, r) be the values that were returned by the TryFlag routine in line 4, and let n2 be the value of del_node. This is the case when the deletion is successful, and it must be linearized at the moment when n2 got marked.

We know that Delete returns in line 9, so n2.key = k and r = true, which means that the TryFlag routine returned in line 6. This means that the C&S in line 4 of the TryFlag routine was successfully executed flagging node prev_node = n1. At that moment n1 was the predecessor of n2. Flagging is the first critical step of the deletion, so by Proposition

17, n2 was not marked at that point of time. Before returning in line 9, Delete called HelpFlagged(n1, n2) in line 6, which, as we reasoned before, does not exit until n2 is marked. Therefore n2 gets marked during the execution of the Delete routine. We linearize Delete at this moment.

Case 4: Suppose the execution of the routine is non-terminated. Suppose Delete has already called the TryFlag routine in line 4, and that TryFlag routine performed a successful C&S in line 4. Let n2 be the value of target_node immediately before that C&S. If n2 is marked, then we linearize this execution of Delete at time T when n2 got marked By Proposition 17, marking of a node is performed after flagging of the node's predecessor, so T is after the deletion started. Therefore, T is a valid linearization point. In any other case we do not linearize this execution.

Finally, let us construct a mapping σ of all marked nodes to successful and non-terminated Delete executions, which has the properties described in the proposition. By Proposition 17, before a node gets marked, its predecessor gets flagged. Nodes can only get flagged in the TryFlag routine, which can be called only from the Delete routine. Let us define σ so that it maps a marked node n to the execution of the Delete routine that flagged n's predecessor, i.e. the Delete execution that performed the first critical step of n's deletion. If a Delete execution flags n's predecessor, then it is poised to return node n in line 9, so at any point of time this execution is either successful or non-terminated.

Each Delete invocation flags at most one node, so ψ is injective (property 1 proved).

If D is a successful execution of the Delete routine, and D returns n, then, as we have showed in Case 3 above, D has flagged n's predecessor, and thus D = σ(n). On the other hand, if D is successful and D = σ(n), then by the definition of σ, the TryFlag routine called by D has flagged n's predecessor by performing a successful C&S in line 4. Since that C&S has flagged n's predecessor, TryFlag's local variable target_node is equal to n immediately before the C&S. By Lemma 21, target_node never changes its value, so when TryFlag was called by D, target_node = n, and thus D's local variable del_node = n. Therefore the node returned by D in line 9 is n (property 2 proved).

Finally, if at time T node n is marked, then σ(n) is defined, and by property 2, execution D = σ(n) returns n. From the way we assign linearization points to the executions of the Delete routines, it follows that D is linearized at the moment n got marked (property 3 proved). ■

The next three propositions are not directly concerned with correctness, but we prove them because we will need them later when we do the performance analysis. Proposition 23 below proves that all three critical steps of the deletion of a node are performed during the execution of the respective Delete routine.

Proposition 23: Each of the three critical steps of n's deletion is performed during the execution of the Delete routine σ(n).

Proof:

As we showed above, if the Delete routine σ(n) returns n, then the first step of n's deletion was performed during the execution of the TryFlag routine it called in line 4. After that, HelpFlag is called in line 6, and it ensures that the second critical step of n's deletion is performed (lines 2-3 of HelpFlagged). Then HelpFlagged calls HelpMarked.

Let us show that before HelpMarked returns, the third step of n's deletion is performed, i.e. n = del_node becomes physically deleted. Successful execution of the C&S in line 2 of HelpMarked physically deletes n. Suppose this C&S fails. Then prev_node.succ was changed after prev_node was flagged, but by Lemma 14 this means that del_node = n was physically deleted. ■

In the following theorem we assign linearization points to the executions of the Insert routines. We will linearize successful insertions at the moment when they insert the node. We will linearized unsuccessful insertions at the moment when there is a regular node in the list that contains the key they are trying to insert. We also define a mapping that will help us to prove that an element can be inserted into the dictionary only by executing an appropriate insertion.

Theorem 24 (Insert correctness): If an execution of the Insert(k, e) routine returns DUPLICATE_KEY (indicating an unsuccessful insertion), then for this execution we can choose a linearization point, at which there was a regular node with key k in the data structure. If an execution of the Insert(k, e) routine returns a pointer to a node (indicating a successful insertion), then the node's key is equal to k, and for this execution we can choose a linearization point, at which this node gets inserted, becoming a regular node.

Furthermore, there exists a mapping ψ of all regular, logically deleted and physically deleted nodes of the data structure to the successful Insert executions and non-terminated Insert executions such that

1. ψ is injective.
2. For any successful execution I of the Insert routine, I = ψ(n) if and only if I returns n.
3. At any time T, if node n is regular, logically deleted, or physically deleted at T, then ψ(n) is linearized in the moment when n got inserted.

Proof:

We choose the linearization point depending on how the Insert routine runs.

Case 1: Suppose the routine returns DUPLICATE_KEY in line 3. Let (n1, n2) be a pair of nodes SearchFrom returns in line 1. The head node is always unmarked, so by Proposition 20 there exists a moment of time during the execution of SearchFrom in line 1, when n1 is unmarked. We linearize Insert at that point of time. At that moment n1 is a regular node, and n1.key = k (line 2).

Case 2: Suppose the routine returns DUPLICATE_KEY in line 22. Let (n1, n2) be a pair of nodes SearchFrom returns when it is executed for the last time in line 19. First notice that n1.key = k (line 20), so to prove the theorem for this case, it is sufficient to choose a linearization point at which n1 is a regular node.

Let n1′ be the value of prev_node immediately before the search in line 19 is executed for the last time. We will show that there exists a moment during the execution of Insert, when n1′ is not marked. Let T1 be the moment when Insert executes line 6 for the last time before returning. If the condition in line 7 was true, then n1′ was not marked at T1 (Invariant 5). If the condition in line 7 was false, the C&S in line 11 was executed and failed, and then Insert entered the loop in lines 17-18. When line 17 was executed for the last time, prev_node = n1′ and it is not marked.

So there exists a moment of time T during the execution of Insert, when n1′ was not marked. Since the first argument of SearchFrom in line 19 is n1′, by Proposition 20 there was a moment of time between T and the completion of SearchFrom (i.e. during the execution of Insert), when n1 was not marked. This is where we linearize Insert. At that moment n1 is a regular node.

Case 3: Suppose the routine returns newNode in line 13. Let n1 be the last node prev_node was pointing to. Insert returns in line 13 only if the condition in line 12 is true, i.e. if the C&S in line 11 is executed successfully. When this C&S is successfully executed, newNode becomes a regular node, because newNode is an unmarked node and has a predecessor n1, which is a regular node of the list. (C&S can only succeed if prev_node is unmarked.) Also notice that newNode.key = k (line 4). We linearize the Insert routine at the moment of the successful execution of the C&S.

Case 4: Suppose the execution of the routine is non-terminated. If the routine has performed a successful C&S in line 11, then we linearize it at the moment when that C&S was performed. This is non-ambiguous, because once Insert performs a successful C&S in line 11, it is poised to return in line 13, so each execution of Insert performs at most one successful C&S in line 11. If the routine has not performed a successful C&S, we do not linearize it.

Finally, let us construct a mapping ψ of all regular, logically deleted, and physically deleted nodes to successful and non-terminated Insert executions, which has the properties described in the proposition. In order for a new node n to be added to the data structure, a preinserted node must be created by the execution of the Insert routine, and then the C&S in line 11 of that Insert routine must be performed successfully. Let us define ψ so that it maps each node to the Insert execution that successfully performed that C&S. If the Insert execution $\psi(n)$ successfully performs the C&S in line 11, then it is poised to return n in line 13, so at any point of time this Insert execution is either successful or non-terminated.

Each Insert invocation executes at most one successful C&S in line 11, so ψ is injective (property 1 proved).

If I is a successful execution of the Insert routine, and I returns n, then, as we have showed above, I has successfully performed the C&S in line 11 that inserted n, and thus I = $\psi(n)$. On the other hand, if I is successful and I = $\psi(n)$, then by the definition of ψ, I has performed a successful C&S in line 11 that inserted n, and therefore the node it returns is n (property 2 proved).

Finally, if at time T node n is regular, logically deleted, or physically deleted, then $\psi(n)$ is defined, and by property 2, execution I = $\psi(n)$ returns n. From the way we assign linearization points to the executions of the Insert routines, it follows that I is linearized at the moment when it successfully performed the C&S in line 11, i.e. when n got inserted (property 3 proved). ■

In the following theorem we assign a linearization points to the executions of the Search routines. This is the last theorem we need to prove the correctness of our implementation.

Theorem 25 (Search correctness): If an execution of the Search(k) routine returns NO_SUCH_KEY, then we can choose a linearization point, at which there was no regular node with key k in the data structure. If an execution of the Search(k) routine returns a pointer to a node, then the key of this node is k, and for this execution we can choose a linearization point, at which this node was a regular node.

Proof:

In the first line the Search routine calls SearchFrom. The head node is always an unmarked node, so by Proposition 20 the nodes returned by SearchFrom(k, head) satisfy the following condition at some point of time during the execution of SearchFrom: curr_node.key ≤ k < next_node.key, curr_node.right = next_node, and curr_node.mark = 0. We linearize the Search routine at this point of time. At that moment curr_node is a regular node, because it is not marked (and it is obviously not a preinserted node).

Case 1: Suppose the Search execution returns a pointer to a node in line 3. Then curr_node's key is k (line 2), and, as we showed, it is regular at the linearization point.

Case 2: Suppose the Search execution returns NO_SUCH_KEY in line 5. Then curr_node.key ≠ k. In this case at the linearization point curr_node and next_node are the two consecutive nodes in the list of regular and logically deleted nodes, and curr_node.key < k < next_node.key, which means that there is no regular node with key k in the data structure. ■

Note that if an execution of the Search(k) routine is non-terminated, we do not linearize it.

It follows from Theorems 22, 24, and 25, that our data structure correctly implements the three dictionary operations. The set of the elements currently stored in the dictionary is the set of the elements of the regular nodes of our data structure. An element of a node n is added to the dictionary at the moment when the insertion $\psi(n)$ is linearized, and is removed from the dictionary at the moment when the deletion $\sigma(n)$ is linearized.

3.4 Performance analysis

In this section we present a performance analysis of our algorithms. We express the performance in terms of contention.

Recall that *point contention* is the number of processes running concurrently at a given point of time, and the *contention of operation S* denoted c(S) is the maximum point contention during the execution of operation S.

Our amortized analysis relies on a fairly complex technique of billing part of the cost of each operation to the successful C&S's that are performed by operations that are running concurrently. The amortized cost of an operation S, denoted $\hat{t}(S)$, is equal to the actual cost of S (which is equal to the cost of the execution of the major routine (including all subroutine calls), corresponding to S) plus the total cost billed to S from other operations minus the total cost billed from S to other operations. We prove that $\hat{t}(S)$ = O(n(S) + c(S)), where n(S) is the number of elements in the data structure when operation S is invoked and c(S) is the contention of operation S.

Let E be an entire execution. The cost of an execution E, denoted t(E), is equal to the sum of the costs of all the steps performed by the processes in the system during E. Note that it follows from our bound on the amortized cost of an operation, that t(E) = $\sum_{S\in E}(t(S)) = \sum_{S\in E}(\hat{t}(S)) = O\left[\sum_{S\in E}(n(S)+c(S))\right]$, where the sum is taken over all operations invoked during E. Let max(n) be the maximum value of n(S), and max(c) be the maximum value of c(S) for all S in E. Let m(S) be the number of operations invoked before S and m be the total number of operations invoked during E. Finally, let $\bar{t}_E(S)$ be the average cost of an operation in execution E. Note that $\bar{t}_E(S) = \frac{t(E)}{m}$. From our bound on the amortized cost of an operation it follows that $\bar{t}_E(S) \in O\left[\frac{\sum_{S\in E}(n(S)+c(S))}{m}\right]$. Since $m \geq \max(n)$ and $m \geq \max(c)$, one can simplify this formula in a following way, getting a less tight bound: $\bar{t}_E(S) \in O\left[\frac{\sum_{S\in E}(n(S)+c(S))}{m}\right] \subset O(\max(n) + \max(c)) \subset O(m)$.

3.4.1 Billing extra work to the successful C&S's

If two processes are executing operations concurrently, their operations on the data structure may interfere, and as a result, one of the processes may need to perform some extra work. We will bill such extra work to the successful C&S's that caused this work to be done. For example, suppose process P is executing a major routine M. If P follows a right pointer into a node that was inserted after P invoked M, the cost of the right pointer traversal is billed to the C&S that inserted that node. If P has to traverse a back_link of a node, the cost of the traversal is billed to the C&S that marked that node. If P fails a C&S, the cost of the failure is billed to the C&S that caused this failure. For every successful C&S we will then define an operation this C&S is part of. Then we will prove that any successful C&S that is part of operation S can be billed no more than c(S) – 1 times, and that the total cost billed to a C&S is O(c(S)). Finally, we will show that since each operation performs a constant number of successful C&S's, the amortized cost of operation S is O(n + c(S)), where the O(n) term comes from the cost of performing an operation without an interference from other processes.

3.4.2 Introduction to Billing

First let us show that when we calculate the cost of our algorithms it is only essential to calculate the number of C&S attempts, the number of back_link pointer traversals, and the number of next_node/curr_node pointer updates by searches. We will show that counting these steps gives an accurate picture of the required time (up to a constant factor).

Proposition 26: Suppose process P is executing a routine M of our data structure. Then the total work done by P during M's execution is O(1 + b + c + u + u′), where b is the number of back_link pointer traversals performed by P during the execution of M (i.e. line 10 in TryFlag, line 18 in Insert), c is the number of C&S's (both successful and unsuccessful) performed by P during the execution of M, u is the number of times P updates a next_node pointer of a SearchFrom routine (line 6 in SearchFrom) during the execution of M, and u′ is the number of times P updates a curr_node pointer of a SearchFrom routine (line 8 in SearchFrom) during the execution of M.

Proof:

We will start by proving auxiliary propositions for each of the routines used by the data structure. We express the performance of the routines using the notation of the proposition: each routine R has its own values of b_R, c_R, u_R, and u'_R which accordingly reflect the number of back_link traversals, C&S's, next_node updates, and curr_node updates performed during the execution of the routine (including all the subroutine calls).

Case 1: M is an execution of HelpMarked, TryMark or HelpFlagged.

Let us first show that the cost of executing a HelpMarked, TryMark or HelpFlagged routine M is $O(c_M)$.

The cost of the HelpMarked routine is O(1), and there is 1 C&S. Lines 1 and 2 in the HelpFlagged routine, and lines 1, 2, 4, and 6 in the TryMark routine cost O(1) to execute. We assign the cost of executing lines 1 and 2 of HelpFlagged to the C&S in the HelpMarked routine called in line 4 of HelpFlagged – this results in adding an additional cost of at most O(1) to each of the C&S's in HelpMarked. We assign the cost of lines 1, 2, 4, and 6 of TryMark to the C&S in line 3 of TryMark. As a result, all the work done in the HelpFlagged, TryMark, and HelpMarked routines will be billed to C&S's, and each C&S will be billed for no more than O(1) steps. Thus the cost of HelpMarked, TryMark or HelpFlagged routine M is $O(c_M)$.

Case 2: M is an execution of SearchFrom.

Let us show that the cost of executing the SearchFrom routine M is $O(1 + c_M + u_M + u'_M)$.

Since execution of HelpMarked costs $O(c_{HM})$ (where c_{HM} is the number of C&S performed by HelpMarked), line 5 does not influence the correctness of the claim and we can ignore it. Line 1 costs O(1). Each iteration of the loop in lines 2-9 costs O(1) + cost of loop in lines 3-6, and in each iteration there is at least one update of next_node pointer in line 6 or curr_node pointer in line 8. Each iteration of the loop in lines 3-6 (ignoring line 5) costs O(1), and there is one update of next_node pointer in each iteration. Therefore, the cost of M is $O(1 + c_M + u_M + u'_M)$.

Case 3: M is an execution of TryFlag.

Let us show that in this case M costs $O(1 + b_M + c_M + u_M + u'_M)$.

The SearchFrom routine called in line 11 costs $O(1 + c_{SF} + u_{SF} + u'_{SF}) = O(1) + O(c_{SF} + u_{SF} + u'_{SF})$. The second addend does not influence the correctness of the claim, so we only need to account for the O(1) term in the analysis. Each iteration of the loop in lines 9-10 has one back_link pointer traversal and costs O(1), so we can ignore lines 9-10 as well. What is left of each iteration of the loop in lines 1-14 costs O(1) and has one C&S attempt. So, the claim for TryFlag holds.

Case 4: M is an execution of Delete.

Let us show that in this case M costs $O(1 + b_M + c_M + u_M + u'_M)$.

By the same argument as in the proof of the claim for TryFlag, we only need to account for the O(1) term in the cost of SearchFrom in line 1. TryFlag in line 4 costs $O(1 + b_{TF} + c_{TF} + u_{TF} + u'_{TF})$, HelpFlagged in line 6 costs $O(c_{HF})$, the rest of the routine costs O(1), thus the claim holds.

Case 5: M is an execution of Insert.

Let us show in this case M costs $O(1 + b_M + c_M + u_M + u'_M)$.

The cost of lines 2-4 is O(1). We only need to account for the O(1) term in the cost of SearchFrom in lines 1 and 19. Lines 8 and 16 are already accounted for in the analysis.

Each iteration of the loop in lines 17-18 costs O(1) and has one back_link pointer traversal, therefore lines 17-18 are accounted for as well. Then what is left of each iteration of the loop in lines 5-23 costs O(1). Note that if HelpFlagged is called in line 8, it will call HelpMarked, which will execute a C&S. Thus, each iteration of the loop in lines 5-23 does one C&S (either in line 8 or in line 11), and has an unaccounted cost of O(1), which we can bill to this C&S. So, the claim holds for the Insert routine

Case 6: M is an execution of Search.

Finally, we show that an execution M of the Search routine costs $O(1 + c_M + u_M + u'_M)$.

SearchFrom in line 1 costs $O(1 + c_{SF} + u_{SF} + u'_{SF})$, the rest of the routine costs O(1), thus the claim holds. ■

When we perform the amortized analysis of our algorithms, we will use the above proposition to focus on the lines of the code that are performance-critical and can change the order of complexity of operations.

Now let us examine the interaction of the processes with one another. A process P completes a particular operation using the minimal number of steps if no operations by other processes run concurrently, so that P does not have to help other operations to complete, and none of the other processes interfere with P's work. Speaking more specifically about our algorithms, P makes the minimal number of steps, if the searches it performs never have to call HelpMark routine, and none of the C&S's it performs fail.

For each operation S that is executed, we will define a different execution of that operation, which will be used to measure the complexity of the actual execution of S.

Consider the system configuration C right before process P invokes operation S. Let us examine all the deletions that are in progress in C. If a deletion performed exactly one critical step, we roll it back by unflagging the node it flagged. If a deletion performed exactly two critical steps, we complete it by performing a third critical step for it, i.e. we switch the predecessor's right pointer to the successor and unflag the predecessor. If a deletion performed zero, or three critical steps, we do not do anything. The configuration that results from all these changes is denoted $\overline{C}$. Now let P invoke S and execute it from $\overline{C}$ without any other processes taking steps. P's steps are called a *solo execution* of operation S from configuration C, and the cost of this execution is called the *solo cost* of S from C.

Note that if all operations executed by the data structure were performed instantaneously at their respective linearization points, the execution of any operation would have the same steps as its solo execution.

Proposition 27: If node n is a regular node in configuration C, then n remains a regular node in $\overline{C}$. If n is a logically deleted node in C, then n is a physically deleted node in $\overline{C}$.

Proof:

To create $\overline{C}$ from C, we complete all the deletions that performed at least two critical steps. This physically deletes from the list all the logically deleted nodes. It does not change the status of the regular nodes. Unflagging the flagged nodes does not change the status of the regular nodes either. Therefore, all the nodes that were regular in C will remain regular in $\overline{C}$, and the nodes, which were logically deleted in will become physically deleted in $\overline{C}$. ■

Proposition 28: If the number of keys in the list in configuration C is n, then the solo cost of S from C is O(n + 1)

Proof:

In $\overline{C}$ there are no flagged or marked nodes in the list. Therefore, P will never have to help other deletions, while executing S solo. We now show that P will also never fail a C&S in the solo execution of S. If P is performing an insertion, a C&S in line 11 does not fail, because the search performed in line 1 returns adjacent nodes, and since no other processes are performing operations, the nodes stay adjacent until the C&S. Also, since there are no deletions in progress, prev_node.flag = prev_node.mark = 0. If P is performing a deletion, the first C&S (line 4 in TryFlag) does not fail for the same reasons the C&S in Insert does not fail. It is easy to see that the second C&S (line 3 in TryMark) also does not fail. The third C&S (line 2 in HelpMarked) does not fail because prev_node was flagged by the first C&S. So, P will also never fail a C&S. Therefore, P will attempt to perform a maximum of three C&S operations, and all of them will be successful. Since P will never fail a C&S, it will never traverse any back_link pointers. Also, P will never perform any updates of the next_node pointer in line 6 of SearchFrom, because there are no marked nodes in the list. Finally, since the deletions are linearized at the point of marking, by Proposition 27, there will be exactly n regular nodes in the list and no logically deleted nodes, when P starts the solo execution of S. Therefore, P will perform no more than n updates of the curr_node pointer in line 8 of SearchFrom. Then, by Proposition 26, the cost of the solo execution of S from C is O(n + 1). ■

When many processes are working together, a change performed by one of them may cause C&S's of other processes to fail and processes may also need to do some extra work to help other processes complete their deletions (i.e. HelpMarked, HelpFlagged routines). In our amortized analysis we will bill the cost of such extra work to the C&S that performed the change that caused this work to be done. For example, if a process tries to insert some node after node n, but the C&S in the Insert routine fails because n is marked, we will bill the cost of this failed C&S and the cost of recovering from the failure to the C&S that performed the marking. We will design a function that will map all the "extra" steps taken by the actual execution of an operation to the successful C&S's executed by the operations running concurrently. We will use this function in our amortized analysis, so that if the actual cost of an operation exceeds its solo cost, the difference between the two is billed to the successful C&S's of the operations running concurrently. Then we will calculate the amortized cost of each successful C&S, which is the cost of the C&S itself (1 step) plus the total cost billed to it. Finally the amortized cost of the operation will be comprised of the solo cost of the routine corresponding to this operation and the sum of the amortized costs of the successful C&S's that are *part of this operation*. We explain this in more detail in the next paragraph.

Our data structure implements three types of operations: searches, insertions, and deletions. As far as the billing is concerned, searches are the most straightforward: if P performs a search operation, its amortized cost is equal to its solo cost (all the rest is billed to the successful C&S's of other operations). If P performs an insertion, the amortized cost of the operation is comprised of the solo cost of the insertion and the cost that is billed to the C&S that P successfully executes in the Insert routine. With the Delete routine it is more complicated, because processes can help one another with deletions. As we showed in Proposition 17, for each deletion there are three successful C&S's that are considered its critical steps. Processes can help each other with deletions by calling HelpFlagged and HelpMarked routines, so the process that invokes the deletion will not necessarily execute all three of these C&S's. Nevertheless, these C&S's are considered to be part of the delete operation and their amortized cost is added to the cost of the delete operation. The amortized cost of the delete operation is comprised of the amortized cost of these three successful C&S's and the solo cost of the deletion itself.

3.4.3 More on Billing

In this section we will introduce two mapping functions that will help us in our amortized analysis. The first function β will define most of our billing scheme, the second mapping function γ will be used to prove that the amortized cost of any operation S is O(n + c(S)). Before we define these functions, we need to prove a few auxiliary propositions that will be useful in showing that a C&S can fail only if another process running concurrently performed a successful C&S on the same field.

Proposition 29: If SearchFrom calls HelpMarked(n, m) in line 5, then at the time when SearchFrom executes line 4 for the last time before HelpMarked is called, n.succ = (m, 0, 1).

Proof:

Let T be the moment when SeachFrom executes line 4 for the last time before calling HelpMarked. At that moment n.right = m. If we prove that at that moment it is also true that n.mark = 0 and m.mark = 1, then it would follow from the third invariant that n.succ = (m, 0, 1) at T. First notice that when SearchFrom executed line 3, m was marked, so it is still marked at T. Let us show by contradiction that n cannot be marked at moment T. Suppose n is marked at time T. We know that n.right = m and m is marked at time T. Therefore, by Proposition 18, n got marked before m, but when SearchFrom executed line 3, it first saw that m is marked and then that n is not marked – a contradiction. So, n is not marked at moment T, and thus n.succ = (m, 0, 1). ■

Proposition 30: If the C&S in line 2 of the HelpMarked(n, m) routine fails, then m is already physically deleted at this point.

Proof:

Let us first show that there was a moment of time T before that C&S was executed, when n.succ = (m, 0, 1). HelpMarked can be called from SearchFrom in line 5 or from HelpFlagged in line 4. If it was called from SearchFrom, then by Proposition 29, when SearchFrom executed line 4 for the last time, n.succ = (m, 0, 1). If it was called from HelpFlagged, then by Lemma 9 before HelpFlagged was invoked, n.succ = (m, 0, 1). So, there was a time T before the C&S when n.succ = (m, 0, 1). Since the C&S fails, n.succ must have changed and, by Lemma 14, when n's successor field changed, m became physically deleted. ■

In the next proposition we examine a moment when line 6 in the SearchFrom routine is executed (i.e. a next_node pointer update is performed). We will bill next_node pointer updates to the C&S's that perform insertions and the C&S's that perform physical deletions and we will use this proposition to prove that no more than O(c(S)) next_node pointer updates can be billed to a successful C&S.

Proposition 31: Suppose SearchFrom executes line 6 at time T′. Let T be the time when SearchFrom updated its next_node pointer for the last time before T′. Let n be the value of curr_node at T′, m be the value of next_node just before T′, and m′ be the value of next_node just after T′. Then either m got physically deleted between T and T′, or m′ got inserted between T and T′.

Proof:

Let us first show that m is not physically deleted at T. Just before T′ curr_node = n and next_node = m. So, just after T, when next_node was updated for the last time, curr_node = n, next_node = m, and n.right = m. Let T″ be the time when SearchFrom last executes line 3 before T′ (to be more precise, it is the moment, when SearchFrom checks the last condition in line 3 before it proceeds, assuming a short-circuit evaluation of the logical expressions). Obviously, $T < T'' < T'$. At T″ m is marked and either n is unmarked, or n.right ≠ m.

If n is not marked at T″, it is not marked at T either, and since n.right = m at T, m is not physically deleted at T. If n.right ≠ m at T″, then n.right changed between T and T″. Since successor pointers of marked nodes never change, it follows that n was unmarked at T, and since n.right = m at T, m is not physically deleted at T.

So, m was not physically deleted at T. Lets return to time T″. We know that m is marked and either n is unmarked, or n.right ≠ m.

If n.right = m at T″, then by Inv 3, n is flagged at T″. Let us show that in this case m will get physically deleted by the time SearchFrom gets to line 6. By Lemma 14, if n's successor field changes, m becomes physically deleted. On the other hand, if n's successor field does not change, then the condition in line 4 would be true, and SearchFrom will call HelpMark in line 5, which will physically delete m.

If n.right ≠ m at T″, then either m already got physically deleted, or a new node got inserted after time T between nodes n and m. If m is still not physically deleted at T′, then it means that there is still at least one such new node between n and m, and since m′ = n.right, m′ will be one of these new nodes. It was inserted into the list between T and T′. ■

Now we are ready to define the function β, which maps the set of steps of the entire execution to itself. We will use this function to define our billing scheme, i.e. if some step performed by a process is mapped by β to a C&S C performed by another process, then the cost of this step is billed to C.

Suppose process P is executing operation S. As we reasoned in the previous section, P may fail some C&S operations, help deletions, and perform other kinds of extra work, which it would not need to do if it was executing S solo. This extra work is caused by the successful C&S's of the operations executing concurrently with S. We want the mapping β to map each extra step performed by P to a C&S that caused it. Steps of operation S will be mapped only to C&S's performed during the execution of S. Since we are aiming at the O(n + c(S)) complexity for our data structure operations, we want to ensure that no more than c(S) steps are mapped to each of the C&S's.

By Proposition 26, our analysis must only account for the C&S's, back_link pointer traversals, next_node pointer updates in line 6 of the SearchFrom routine, and curr_node pointer updates in line 8 of the SearchFrom routine. Therefore, we define β only for these steps.

Def 6: Let Q be the set of steps in the entire execution E that are C&S's, back_link pointer traversals, updates to next_node in line 6 of SearchFrom, or updates to curr_node in line 8 of SearchFrom. Function β maps Q to itself. If some operation S performs step s $\in$ Q, β maps this step either to itself, or to a successful C&S that is part of another operation as described below.

- C&S's:

  Suppose a C&S C on the successor field of node n was executed.

  Our algorithms execute four types of C&S's:

  1. Insertion C&S (line 11 in Insert)
  2. Flagging C&S (line 4 in TryFlag)
  3. Marking C&S (line 3 in TryMark)
  4. Physical Deletion C&S (line 2 in HelpMarked)

  If C is successful, then we map it to itself. If C fails, and it is not of the fourth type, we map it to the C&S that last modified n.succ. If C of the fourth type fails, we map it to the C&S that physically deleted the node that C was trying to delete. (Note that by Proposition 30 such a C&S exists).
- Back_link pointer traversals:

  A back_link pointer traversal from node n to node m is mapped to the C&S that marked node n.
- Next_node pointer updates in line 6 of the SearchFrom routine:

  Suppose just before the update next_node = m, and just after the update next_node = m′. If m is physically deleted when the update is performed, we map the update to the C&S that performed the physical deletion of m. (Note that even though this C&S could be performed by HelpMarked called from this SearchFrom routine in line 5, it is part of another operation.) Otherwise we map the update to the C&S that inserted m′. Proposition 31 will ensure that the C&S we map the update to was performed during the execution of S.
- Curr_node pointer updates in line 8 of the SearchFrom routine:

  Suppose the update sets curr_node pointer to node n. If n was inserted into the list after operation S was invoked, then the update is mapped to the C&S that inserted n (type 1). Otherwise, the update is mapped to itself.

The function β defines most of the billing scheme. It maps all the extra steps taken by operations to the successful C&S's that caused these steps to be taken. However, it does not redistribute the cost of the successful C&S's themselves between the operations. In the previous subsection we reasoned that processes help one another, and even if a C&S C is performed by process P executing operation S, C may be part of some other operation S′. Thus, it would be logical to bill the cost of C to S′, and not S. The second mapping we use in our analysis – mapping γ – takes care of this issue. It maps the range of β to the set of steps of the solo executions of the operations performed during E. The complete billing scheme will be defined by $\gamma \circ \beta$ – the composition of γ and β. Later we shall prove that γ is injective, i.e. no two steps in range(β) are mapped to the same step in the set of solo executions. We will also show that β maps no more than c(S) steps to each successful C&S. These two facts together with Proposition 28 will help us prove later that the amortized cost of any operation is O(n + c(S)).

The following lemma proves that certain preconditions hold when SearchFrom is called from the Insert routine in line 19. These preconditions will help us define mapping γ, and they will also be used later in the analysis.

Lemma 32: Suppose at time T Insert is about to call the SearchFrom routine in line 19. Let n be the value of prev_node at that moment. Then there exists time T′ < T during the execution of Insert such that n is not marked at T′, and Insert does not traverse any back_link pointers and does not call any SearchFrom routines between T′ and T.

Proof:

Let T″ be the moment when Insert executes line 6 for last time before the search. Let us examine the moment when Insert executes the next line (line 7). Suppose the condition in line 7 was true. In this case T″ satisfies the conditions for time T′ set by the lemma, because by the fifth invariant n.mark = 0 at T″, and Insert does not traverse any back_link pointers and does not call any SearchFrom routines between T″ and T.

Suppose the condition in line 7 is false. Then Insert will execute the loop in lines 17-18. At the moment when it last executes line 17, prev_node = n and prev_node.mark = 0. This is a valid moment for T′. ■

Lemma 33 is analogous to Lemma 32, except that it applies to TryFlag instead of Insert.

Lemma 33: Suppose at time T TryFlag is about to execute the search in line 11. Let n be the value of prev_node at that moment. Then there exists a time T′ < T during the execution of TryFlag such that n is not marked at T′, and TryFlag does not traverse any back_link pointers and does not call any SearchFrom routines between T′ and T.

Proof:

Before TryFlag calls SearchFrom in line 11, it has to execute the loop in lines 9-10. At the moment when it last executes line 9, prev_node = n and prev_node.mark = 0. This is a valid moment for T′. ■

The following proposition will be used to define mapping γ.

Proposition 34: Suppose operation S performs a curr_node pointer update u in line 8 of a SearchFrom routine it called, which sets a curr_node pointer to node n. If β maps u to itself, then

1. node n is not marked when S is invoked, and
2. there is a curr_node update u′ during the solo execution of S, which sets a curr_node pointer to node n as well.

Proof:

We will start by proving the first claim. Let T be the time when update u occurs in a SearchFrom routine. Suppose this SearchFrom started from node m. If we prove that there was a time T′ during the execution of S, when m was not marked, then by Proposition 19, n was not marked at T′ either, and thus n was not marked when S is invoked.

SearchFrom can be called from line 1 in Search, from line 1 or line 19 in Insert, from line 1 of Delete, or from line 11 in TryFlag. If SearchFrom was called from line 1 in Search, from line 1 in Insert, or from line 1 in Delete, the first claim holds, because the head node is always unmarked. If SearchFrom was called from line 19 in Insert or from line 11 in TryFlag, then Lemma 32 or 33 applies, and thus the first claim holds in this case as well.

Now let us prove the second claim. Since β maps u to itself, n was in the list and, as we showed, was not marked when S was invoked, i.e. n was a regular node. Therefore, by Proposition 27, n will be a regular node when the solo execution of S starts. Suppose the SearchFrom that set curr_node to n was invoked with the first parameter k. This is a key value, which does not change throughout the execution of S, and it is the same in the solo execution of S. Therefore, if SearchFrom called by S updated curr_node to n in line 8, it means the condition in line 7 was true and n.key ≤ k. By invariants 1 and 2, all the regular and logically deleted nodes of the data structure are arranged into a linked list in a sorted order. Therefore, since SearchFrom traverses the nodes one by one following the right pointers, and since n.key ≤ k, at some point during the solo execution of S curr_node pointer will be set to n. ■

We are now going to define mapping γ. Mapping γ is going to operate on the range of mapping β, applied to the steps of the execution. Recall, that range(β) has only successful C&S's and curr_node pointer updates by SearchFrom.

Def 7: Let E be an entire execution. Function γ maps the range of mapping β, applied to E, to the set of steps of solo executions of all the operations in E as described below.

- C&S's:

Each successful C&S that is part of operation S is mapped to the C&S of the same type in the solo execution of S.

- Curr_node pointer updates by SearchFrom routines in line 8:

Suppose the update u performed by S sets curr_node pointer to node n. Then u is mapped to the update u′ performed by the solo execution of S that sets curr_node pointer to n. Since the only curr_node updates in range(β) are the updates that are mapped by β to themselves, such a u′ exists by Proposition 34.

In the next four subsections we will show that β does not map more than c(S) steps to any successful C&S performed by operation S. Then we will prove that the amortized cost of any operation S is O(n + c(S)).

3.4.4 Mapping C&S operations.

In this subsection we prove that if a successful C&S is part of operation S, no more than O(c(S)) failed C&S's can be mapped to it by β. We start by proving an auxiliary lemma about the HelpFlagged routine.

Lemma 35: Suppose process P is executing the HelpFlagged(n, m) routine. Then the following holds

1. When P completes HelpFlagged(n, m), m is physically deleted. In fact, m is guaranteed to be physically deleted after the C&S in the HelpMarked(n, m) routine called by HelpFlagged is performed.
2. If P tries to mark some node m′ during the execution of HelpFlagged(n, m), then m′ gets marked at some time T during the execution, and n is flagged at T.

Proof:

Let us start with the first claim. HelpFlagged calls HelpMarked, which tries to execute a C&S C to physically delete m. By Lemma 9, at some point before HelpFlagged was called, n.succ = (m, 0, 1). If this is still true when C is executed, then C succeeds and m gets physically deleted. Otherwise n.succ was changed, and by Lemma 14 any change to n.succ physically deletes m.

Let us now prove the second claim. HelpFlagged can attempt to mark nodes by calling (possibly, recursively), the TryMark routine in line 3. First note that if HelpFlagged attempts to mark node m′, then m′ will get marked at some time T during the execution of HelpFlagged, because it is not marked before TryMark(m′) is called (line 2 in HelpFlagged), and TryMark(m′) does not exit until m′ gets marked. Second, by the third invariant, m′'s predecessor is flagged at T. If m′'s predecessor is n, then the claim holds. Otherwise, the claim follows from Lemma 14 and from the fact that HelpFlagged recursively calls itself only if the node it is trying to mark is flagged. ■

Lemmas 36 and 37 below will help us prove that a failed C&S performed by an execution M of a major routine is mapped by β to a C&S that was performed during M (Proposition 38), and that one execution of a major routine cannot have more than one failed C&S mapped to the same successful C&S (Proposition 39). Then we will use these facts to prove that if a successful C&S is part of operation S, no more than O(c(S)) failed C&S's can be mapped to it by β.

Lemma 36: Suppose process P is performing an execution M of some major routine, and it fails a C&S C of the form
C&S(n.succ, (old_right, old_mark, old_flag), (new_right, new_mark, new_flag)). Suppose, C is of type 1, 2, or 3. Then there exist moments of time T1, T2, and T3 during M, such that at moment T1 n.right = old_right, at moment T2 n.mark = old_mark, and at moment T3 n.flag = old_flag. Furthermore, P does not try to perform any C&S's on n.succ between T1 and C, P does not try to perform C&S's of types 1, 2, or 3 on n.succ between T2 or T3 and C, and P does not traverse any back_link pointers between T2 and C.

Proof:

Roughly speaking, the reason this lemma holds is that processes do not “blindly” perform the C&S’s. Before P attempts a C&S on n.succ, it makes sure the old values it gives to the C&S accurately reflect the values of n.succ’s fields at some time after P last attempted a C&S. Therefore, the only way P can fail this C&S is if another process concurrent with P did successful C&S on n.succ after P’s last attempt to perform a C&S. We will prove the lemma separately for each possible type of C.

Case 1: C is of type 1 (insertion).

The C&S is c&s(prev_node.succ, (next_node, 0, 0), (newNode, 0, 0)).

Here, n is the node prev_node was pointing to, when C failed. Let m be the node next_node was pointing to when C failed. C is of type one, so it was performed by the Insert routine. Let us examine the flow of execution of that Insert up until the moment when it performed C. When Insert last executed line 7, the condition there had to be false. Therefore, when it executed the previous line (line 6), n.flag = 0, and this is a valid moment for T3. Let us examine the last SearchFrom routine that Insert called before C. This SearchFrom was called in line 1 or in line 19, and Insert did not perform any C&S’s on n.succ after that until C. By Proposition 20 there exists a moment during the execution of that SearchFrom, when n.right = m and SearchFrom does not perform any C&S’s on n after that moment – this is a valid moment for T1. Now we only have to find a valid moment for T2.

If the last SearchFrom which Insert called was in line 1, then, since head is always an unmarked node, there exists a moment during the execution of that SearchFrom, when n.mark = 0 by Proposition 20. Since SearchFrom does not perform C&S’s of the first three types and does not traverse any back_links, this is a valid moment for T2.

Suppose the last SearchFrom was called in line 19. Let n′ be the value of prev_node immediately before the execution of line 19. We shall prove that there was a time T before SearchFrom was called, when n′ was not marked, such that Insert does not perform any C&S’s of types 1-3 on n and does not traverse any back_link pointers between T and C. It then follows from Proposition 20, that there exists a moment between T and the completion of SearchFrom, when n.mark = 0, and that would be a valid moment for T2. Let T′ be the time when Insert executes line 6 for last time before the search. Let us look at the moment when Insert executes the next line (line 7) after T′. Suppose the condition in line 7 was true. We will show that in this case T′ satisfies the conditions for moment T we outlined above. First, notice that Insert does not traverse any back_link pointers between T′ and C, and by the fifth invariant n′.mark = 0 at T′. Second, since the condition in line 7 is true, HelpFlagged in line 8 will be executed. HelpFlagged may perform several C&S’s of type 3 and 4 on n′’s successor and the nodes following it and then a C&S of type 4 on n′. By Lemma 35, all the nodes on which HelpFlagged attempts to perform C&S’s of type 3 get marked while n′ is flagged (and not marked), and therefore none of them is n, because, by Proposition 20, n cannot get marked before n′. Thus, HelpFlagged will perform no C&S’s of types 1-3 on n. The SearchFrom which is then called in line 19 can only perform C&S’s of type 4, so it does not perform C&S’s of types 1-3 on n either. Thus, T′ indeed satisfies the necessary conditions for moment T in the case when the condition in line 7 is true.

Suppose the condition in line 7 is false. Then Insert will execute the loop in lines 17-18. When it last executes line 17, prev_node = n′ and prev_node.mark = 0. This is a valid moment for T, because no C&S's of types 1-3 will be performed on n and no back_link pointers will be traversed between this point of time and C.

So, there exists a moment T with the properties outlined above.

Case 2: C is of type 2 (flagging).

The C&S is c&s(prev_node.succ, (target_node, 0, 0), (target_node, 0, 1)).

C is of type two, so it was performed by the TryFlag routine, and M is an invocation of a Delete routine. Here, n is the node prev_node was pointing to when C failed. Let m be the node target_node was pointing to when C failed. Let us examine the flow of execution of the TryFlag routine up until the moment when it performed C.

Let us look at the last SearchFrom routine called by M before C (which can be a search in line 1 of Delete or in line 11 of TryFlag). That SearchFrom returned nodes n and m (if a search in line 11 of TryFlag returns del_node ≠ target_node, TryFlag exits in line 13). So, by Proposition 20 there was a moment during the execution of that SearchFrom, such that at that moment n.right = m and after it SearchFrom does not perform any C&S's on n. This is a valid moment for T1. Now let us examine the moment T3′ when TryFlag checks the condition in line 2 for the last time before performing C. At that moment prev_node = n. If n is not flagged at T3′, then this is a valid moment for T3. Suppose it is flagged. T3′ is the last time when TryFlag executes line 2 before performing C, so the condition in line 2 must be false, and thus n.right ≠ m at T3′. At moment T1 n.right = m, so n.right had to change between T1 and T3′. Immediately after that change n is not flagged and this is a valid moment for T3.

Now we only have to find a valid moment for T2. If the last SearchFrom called by M before C was in line 1 of Delete, then, since the head node is always unmarked, there exists a moment during the execution of that SearchFrom, when n.mark = 0 by Proposition 20. Since SearchFrom does not perform C&S's of the first three types and does not traverse any back_links, this is a valid moment for T2.

Suppose the last SearchFrom before C was called in line 11 of TryFlag. Let n′ be the value of prev_node before the execution of line 11. Before calling SearchFrom, TryFlag executed the loop in lines 9-10. Let T be the moment when it executed line 9 for the last time. At that moment n′ was not marked. Then by Proposition 20, there exists a moment between T and the return of SearchFrom, when n was not marked. This is a valid moment for T2.

Case 3: C is of type 3 (marking).

The C&S is c&s(del_node.succ, (next_node, 0, 0), (next_node, 1, 0)).

Here, n is the node del_node was pointing to when C failed. Let m be the node next_node was pointing to when C failed. C was performed by the TryMark routine. The moment when TryMark sets next_node in line 2 for the last time before C, is a valid moment for T1. Let us examine the moment when TryMark executes line 4 for the last time before C. If n is not flagged at that moment, then it is a valid moment for T3. Suppose n is flagged at that moment. Then HelpFlagged will be called, and by the time it returns, n's successor will be physically deleted (Lemma 35). Therefore, a successful C&S of type four will be performed on n at some point of time after line 4 in TryMark is executed and before HelpFlagged returns in line 5. After that C&S, n is not flagged, and this is a valid moment for T3, because P does not perform C&S's of the first three types on n during the execution of HelpFlagged. Now we only have to find a valid moment for T2. If TryMark iterates at least once before executing C, then the moment it executes line

6 for the last time before C is a valid moment for T2. Suppose C is performed during the first iteration of TryMark. In this case, notice that TryMark can was called from HelpFlagged, and before that call was made, HelpFlagged checked that n was not marked – this is a valid moment for T2. ■

The next lemma is a simpler analogue of Lemma 36 and is concerned with the C&S's of the fourth type.

Lemma 37: Suppose process P is performing an execution M of some major routine, and it fails a C&S C of type four
C&S(n.succ, (old_right, 0, 1), (new_right, new_mark, new_flag)). Then there exists a time T during M, such that n.succ = (old_right, 0, 1) at T, and P does not try to perform any C&S's on n.succ between T and C.

Proof:

C is of type four, so it was performed in the following line of the HelpMarked routine: c&s(prev_node.succ, (del_node, 0, 1), (next_node, 0, 0)).

Let n be the node prev_node was pointing to when C failed. Let m be the node del_node was pointing to when C failed. HelpMarked can be called from SearchFrom or from HelpFlagged.

Suppose HelpMarked was called from SearchFrom. Then by Proposition 29 at the moment when SearchFrom executed line 4 for the last time before calling HelpMarked, n.succ = (m, 0, 1), and thus this moment is valid for T.

Suppose HelpMarked was called from HelpFlagged. Notice HelpFlagged calls HelpMarked with the same arguments it was called with. Therefore, by Lemma 9 there exists a moment T′ during M, when n.succ = (m, 0, 1), and P performs no C&S's on n between T′ and the moment it calls HelpFlagged. Since HelpFlagged itself performs no C&S's on n before C, T′ is a valid moment for T. ■

Proposition 38: Suppose process P fails a C&S C on n.succ during the execution M of some major routine and β(C) = C′. Then C′ was performed during M.

Proof:

Case 1: C is of type 1, 2, or 3

In this case Lemma 36 applies. Let T be the earliest of T1, T2, T3 defined in Lemma 36. Since C failed, at least one successful C&S had to be performed on n.succ between T and C. By the definition of β, C′ is the last such C&S, and thus it was performed between T and C as well. Since both of these two moments belong to the period of execution M, C′ was performed during M.

Case 2: C is of type 4

In this case Lemma 37 applies. Since C failed, at least one successful C&S had to be performed on n.succ between T and C. At T n.flag = 1, and the only C&S's that can change flagged successor fields, are the C&S's of the fourth type. Thus, there had to be a successful C&S of type four performed on n.succ between T and C. By the definition of β, C′ is the last such C&S, and thus it was performed between T and C as well. Since both of these two moments belong to the period of execution M, C′ was performed during M. ■

Proposition 39: Suppose process P fails a C&S C on n.succ during an execution M of some major routine and β(C) = C′. Then there are no other C&S's performed by P that were mapped to C′ by β.

Proof:

Suppose there exist two failed C&S's C1 and C2 performed by P, which are both mapped by β to the same C&S C′. We will prove that this is impossible by contradiction. Let C1 be before C2, and let C2 be of the form
C&S(n.succ, (old_right, old_mark, old_flag), (new_right, new_mark, new_flag)).

Case 1: C1 and C2 are both of types 1-3 (see Figure 18).

In Figures 16-18, circles denote successful C&S's, crossed circles denote failed C&S's.

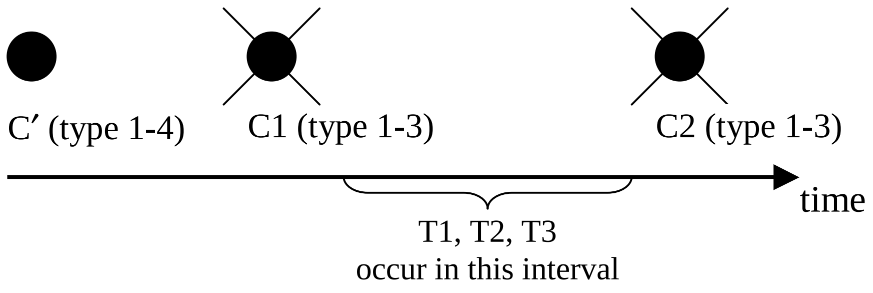


Figure 18: C1 and C2 both of types 1-3.

Let T1, T2, and T3 be the times defined in Lemma 36 for C2. Since both C1 and C2 are of types 1-3, T1, T2, and T3 are between C1 and C2 by Lemma 36. Since C2 failed, n.succ had to change between the earliest of these three moments and C2. So there had to be a successful C&S on n.succ between C1 and C2, and therefore β(C2) must be between C1 and C2, but β(C2) = C′, which is before C1 – a contradiction.

Case 2: C1 is of type 4 and C2 is of type 1, 2, or 3 (see Figure 19).

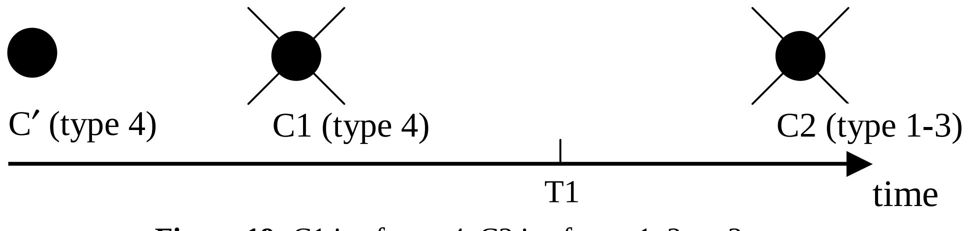


Figure 19: C1 is of type 4, C2 is of type 1, 2, or 3.

Let T1 be the time defined in Lemma 36 for C2. Since β(C1) = C′ and C1 is of type 4, C′ is of type 4 as well. Since C2 is of type 1-3, it is mapped to the latest successful C&S on n.succ before it, and thus there were no successful C&S's performed on n.succ between C′ and C2. So immediately before C2, n is not marked or flagged, i.e. n.mark = 0 = old_mark, n.flag = 0 = old.flag. Also, from Lemma 36 it follows that moment T1 is between C1 and C2, and therefore n.right = old_right immediately before C2. So, immediately before C2, n.succ = (old_right, old_mark, old_flag), and C2 must be successful – a contradiction.

Case 3: C2 is of type 4 (see Figure 20).

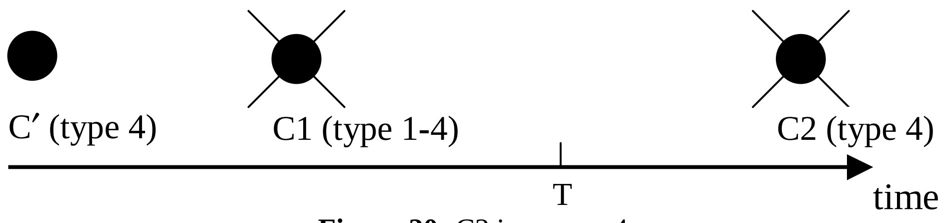


Figure 20: C2 is of type 4.

Let T be the time defined in Lemma 37 for C2. By Lemma 37, T is between C1 and C2. Since C2 fails, there must be a successful C&S on n.succ between T and C2. Let us take the first such C&S C″. Immediately before it is performed, n is flagged, because it was flagged at moment T. Therefore, since C″ successfully changes a flagged successor field, it must be of the fourth type. But then C2 must be mapped to C″, and not to C′ – a contradiction. ■

The following proposition will be used to bound the contribution of failed C&S's to the amortized cost of a successful C&S.

Proposition 40: If a successful C&S C′ is part of an operation S, then β maps no more than c(S) failed C&S's to C′.

Proof:

Suppose C′ was performed by process Q while it was executing a major routine M′. By Proposition 39 each process P can map at most one failed C&S to C′. We know that if a C&S is part of operation S, then it was performed during the execution of S (for insertions it is obvious, and for deletions it follows from Proposition 23). Thus, to prove the proposition, it would be sufficient to prove that if a C&S performed by P is mapped to C′, then P was executing some operation when C′ was performed. This follows directly from Proposition 38. ■

3.4.5 Mapping back_link pointer traversals

Back_link pointer traversals from node n are mapped to the C&S that marked n. In this subsection we will prove that if a marking C&S (type 3) is part of an operation S, then no more than c(S) back_link traversals are mapped to it by β. We start by proving that back_link chains cannot grow towards the right, i.e. that back_links are never set to marked nodes.

Proposition 41: Back_links cannot be set to marked nodes. I.e. if n2.back_link = n1, then either n1 is not marked, or n1 got marked later than n2.

Proof:

By the third invariant, when n2 got marked, n1 was flagged, and therefore not marked. So, either n1 is not marked, or it was marked later than n2. ■

Proposition 42: Suppose process P traverses a back_link from node n at time T during the execution M of a major routine. Then n got marked during M, and P has never traversed n's back_link before.

Proof:

Back_link pointers can be traversed in line 18 of the Insert routine and in line 10 of the TryFlag routine.

Suppose the back_link was traversed in line 18 of the Insert routine. Since line 18 was executed, the C&S C that the Insert performed when it last executed line 11 failed. Let n′ be the value of prev_node when C failed. Since Insert entered the loop in lines 17-18, n′ was marked when Insert first executed line 17 after failing C, so it is marked at time T as well. The C&S C is of type 1, so Lemma 36 applies. By that lemma, there was a time T2 during the execution of Insert, when n′ was not marked, and Insert does not traverse any back_links between T2 and C. This means that n′ was marked between T2 and T, and since node n was reached by following back_links from n′, n got marked after n′, by Proposition 41. On the other hand, n is marked at time T, so n got marked between T2 and C. First, notice that this means that n was marked during the execution of Insert. Second, this means that Insert never traversed n's back_link pointer before T, because it only traversed it once (at time T) between C and T, it did not traverse it between T2 and C by Lemma 36, and it could not traverse it before T2, because n got marked after T2.

The proof for the case when back_link was traversed during the execution of TryFlag goes exactly the same way. (The C&S in line 4 is of type 2, so Lemma 36 applies). ■

The following proposition will be used to bound the contribution of back_link traversals to the amortized cost of a successful C&S.

Proposition 43: If a successful C&S C′ is part of an operation S, then β maps no more then c(S) back_link pointer traversals to C′.

Proof:

Suppose C′ was performed by process Q while Q was executing a major routine M′. From Proposition 42 it follows that any process P can map at most one back_link traversal to C′, and that P can only map a back_link traversal to C′ if it was executing some operation when C′ was performed. Since C′ is part of operation S, it was performed during the execution of S by Proposition 23, and therefore no more than c(S) back_link traversals can be mapped to C′. ■

3.4.6 Mapping next_node and curr_node pointers updates

In this subsection we prove that if a successful C&S is part of operation S, then no more than c(S) next_node pointer updates and no more than c(S) curr_node pointer updates are mapped to it. We start by proving this for the next_node pointer updates.

Proposition 44: Suppose process P updates a next_node pointer from node m to node m′ in line 6 of the SearchFrom routine at time T′ during the execution M of a major routine. If this update is mapped by β to a C&S C, then β does not map any of M's earlier next_node pointer updates to C, and C was performed during M.

Proof:

By the definition of β, if m is physically deleted when the update is performed, then C is the C&S that performed the physical deletion. Otherwise C is the C&S that inserted m′. From Proposition 31 it follows that C was performed between T and T′, where T is the time when SearchFrom last updated its next_node pointer. It is then easy to see that both claims of the proposition hold. ■

Proposition 45: If a successful C&S C′ is part of an operation S, then β maps no more then c(S) next_node pointer updates to C′.

Proof:

From Proposition 44 it follows that for any process P, at most one of P's next_node pointer updates can be mapped to C′, and that this can happen only if P was executing some operation when C′ was performed. Since C′ is part of operation S, it was performed during the execution of S, and therefore no more than c(S) next_node pointer updates can be mapped to C′. ■

The following proposition proves an important property of the curr_node pointer updates, which we will use to prove that no more than c(S) curr_node pointer updates can be mapped to a single successful C&S.

Proposition 46: Process P cannot set curr_node pointer in line 8 of a SearchFrom routine to the same node n more than once during a single execution M of a major routine.

Proof:

During the execution of a single SearchFrom routine, the curr_node pointer moves through the list strictly towards the right, because it is only updated in line 8, and it is set to a node that was curr_node's successor at some point. Therefore, since the keys of the nodes are strictly ordered, the key of curr_node strictly increases, and thus curr_node cannot be set to the same node more than once during the execution of a single SearchFrom routine.

If M is an invocation of the Search routine, the proposition holds, because Search invokes SearchFrom only once. If M is an invocation of the Insert routine, then SearchFrom can be called by M in lines 1 and 19 of the Insert routine. If M is an invocation of the Delete routine, then SearchFrom can be called by M in lines 1 of the Delete routine or in line 11 of the TryFlag routine. We will examine the case when M is an invocation of the Insert routine. The proof for the other case is similar.

Suppose two invocations of the SearchFrom routine, SF1 and SF2, called by the Insert routine, entered node n at times T1 and T2 respectively, where T1 < T2 (see Figure 21). Suppose SF1 started from node n_1, and the first of the two nodes returned by SF1 was $n_1′$, and SF2 started from node n_2 and the first node returned by SF2 was $n_2′$. As we reasoned, the key of curr_node strictly increases during executions SF1 and SF2, so n_1.key < n.key ≤ $n_1′$.key and n_2.key < n.key ≤ $n_2′$.key (n_1 and n_2 are different from n because SF1 and SF2 assigned value n to their curr_node pointers in line 8). Therefore, n_2.key < n.key ≤ $n_1′$.key. After SF1 returns, Insert assigns prev_node = $n_1′$, and then before Insert calls SF2, it somehow changes prev_node to n_2. Executing a SearchFrom routine only increases the key of prev_node, so the only way for Insert to decrease the key of prev_node is to traverse a chain of back_links. Therefore, at some time between T1 and T2, Insert traversed a chain of the nodes m_1, …, m_k following their back_link pointers, and for some $j < k$, m_{j+1}.key < n.key ≤ m_j.key. We will prove that m_1 got marked after T1, which will lead us to a contradiction.

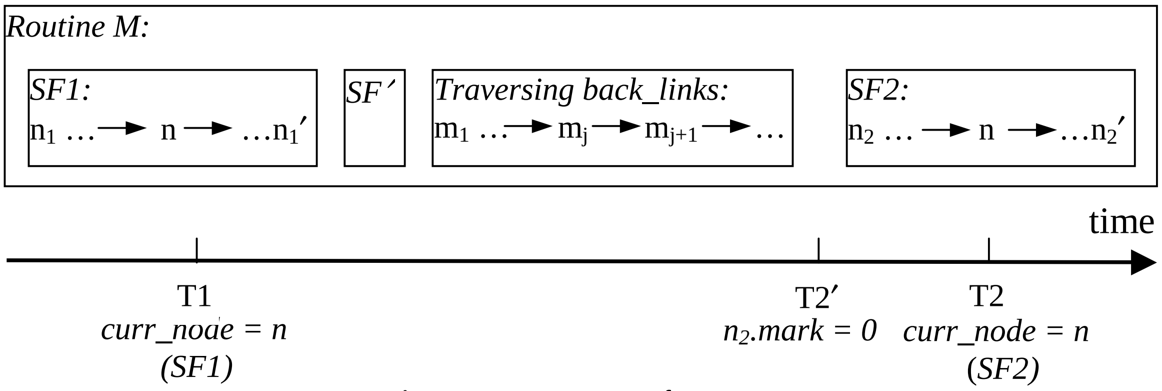


Figure 21: Execution of routine M.

Routine SF1 traversed node n at time T1, and therefore n was inserted into the list before T1.

Routine SF2 started from node n_2. By Lemma 32 there was time T2′, when n_2 was not marked, and Insert does not call any SearchFrom routines and does not traverse any back_link pointers between T2′ and the time it calls SF2. Therefore, T1 < T2′ < T2. Since SF2 traversed node n, by Proposition 19, n was not marked or was not in the list at T2′. We know that n was inserted before T1, and therefore at T2′ n is in the list and unmarked, i.e. it is a regular node. Hence, n is also a regular node at T1.

Since Insert started to traverse a chain of back_links from m_1, node m_1 was returned by some invocation SF′ of the SearchFrom routine as a first parameter. Invocation SF′ can be the same as SF1 or a separate invocation called between SF1 and SF2. We will now prove that in both cases node m_1 got marked after T_1.

Case 1: SF′ ≠ SF1.

Then by Lemma 32 there was time T′ when the node SF′ starts from was not marked, and Insert does not call any SearchFrom routines between T′ and the time it starts SF′. Therefore, T1 < T′. Since SF′ assigned curr_node = m_1 at some point, it follows from Proposition 19 that m_1 was either unmarked or not in the list at T′. Since T1 < T′, m_1 got marked after T1.

Case 2: SF′ = SF1.

We know that n.key ≤ m_1.key, so either n = m_1, or SF1 assigns curr_node = n before it assigns curr_node = m_1. When SF1 assigns curr_node = n at T1, n is not marked. Then it follows from Proposition 19 that at T1 m_1 is either unmarked or not in the list. Thus, in this case it is also true that m_1 got marked before T1.

So, we proved that m_1 got marked after T1. It then follows from Proposition 41 that nodes m_1, m_2, …, m_k all got marked after T1. Let us examine a moment T3 when m_j got marked. We know that T3 > T1. Remember that at time T2′ node n is a regular node, and Insert does not traverse any back_link pointers between T2′ and the time it calls SF2.

Therefore, T2′ is after the moment Insert traversed a back_link from m_j to m_{j+1}, so it is after time T3, when m_j got marked. So, T1 < T3 < T2′. At T2′ n is a regular node, so it is a regular node at T3 as well. We know that m_j's back_link is pointing to node m_{j+1}, so by the third invariant m_{j+1}.succ = (m_j, 0, 1) immediately before T3. On the other hand m_{j+1}.key < n.key ≤ m_j.key, and immediately before T3 nodes n, m_j, and m_{j+1} are regular nodes of the list. Also, n ≠ m_j, because m_j is marked at T3 and n is not. So, by the second invariant immediately before T3 node n must be between nodes m_j and m_{j+1} – a contradiction.

The proof for the case when M is a Delete routine is identical, except that Lemma 33 should be used instead of Lemma 32. ■

Proposition 47: If a successful C&S C′ is part of an operation S, then β maps no more than c(S) curr_node pointer updates to C′.

Proof:

From the definition of the mapping function β it follows that process P can map a curr_node pointer update to C′ only if P was executing some major routine M when C′ was performed. Furthermore, by Proposition 46, M can set curr_node pointer in line 6 of a SearchFrom routine to the same node n at most once during its execution. Since C′ is part of operation S, it was performed during the execution of S, and therefore no more than c(S) curr_node pointer updates can be mapped to C′. ■

3.4.7 Putting everything together

In this final subsection of the chapter we will prove that the amortized cost of each operation S performed by our data structure is O(n + c(S)), where n is the n is the number of elements in the dictionary (equal to the number of regular nodes in the list) when S is invoked, and c(S) is the contention of operation S. We will then give the bounds for the average cost of an operation in an execution.

Function β defines the billing scheme, i.e. if steps s and s′ are such that β(s) = s′, then the cost of s is billed to s′. From the definition of β it follows that either s = s′, or s′ is a successful C&S. In the next theorem we prove that any successful C&S can be billed for no more than O(c(S)) steps.

Theorem 48: The total number of steps billed by β to a successful C&S that is part of operation S is O(c(S)).

Proof:

From the definition of β, it follows that the steps that can be billed to a successful C&S are the failed C&S's, back_link pointer traversals, updates to next_node in line 6 of SearchFrom, and updates to curr_node in line 8 of SearchFrom. By Propositions 40, 43, 45, and 47, there are no more than O(c(S)) such steps billed to each successful C&S. ■

The next theorem states the amortized cost of the operations of our data structure.

Theorem 49: For any execution E of operations on our data structure, the total cost of all the steps performed by the processes during E, denoted t(E), is $O\left[\sum_{S\in E}(n(S)+c(S))\right]$

where the sum is taken over all operations invoked during E, and n(S) is the number of elements in the dictionary when operation S is invoked.

Proof:

Let Q be the set of steps in the entire execution E that are C&S's, back_link pointer traversals, updates to next_node in line 6 of SearchFrom, or updates to curr_node in line 8 of SearchFrom.

From Proposition 26, it follows that there exists a constant $C_1 > 0$, such that for any execution M of a major routine, the actual cost of M, denoted t(M), is less than $C_1(1 + b + c + u + u')$, where b, c, u, and u′ are the number of back_link traversals, C&S's, next_node pointer updates in line 6 of SearchFrom, and curr_node pointer updates in line 8 of SearchFrom respectively, performed during M. When we calculate the cost of an execution E, we will account only for the steps that belong to set Q, and we will assume that each such step costs C_1. Let t′ denote the cost function calculated this way. We will prove an upper bound on t′, and since $t(M) < C_1(1 + b + c + u + u') = t'(M)$ for any execution of a major routine M, that upper bound will apply to t as well.

To prove the bound on the amortized cost of the operations, we will construct a scheme of billing the cost of steps that are in set Q to the operations. We will first use mapping β to bill the cost of individual steps in Q to one another. This will give us the amortized cost of each step in Q. Then we will use mapping γ to bill the amortized costs of these individual steps to the operations performed during E.

Let $s \in Q$ be an individual step. For each step $s \in Q$, we define the amortized cost $\hat{t}'(s)$ to be the total cost t′ of the steps mapped to s by β. Since β maps Q to itself, this will not change the total cost of the steps in Q, i.e. $\sum_{s\in Q} t'(s) = \sum_{s\in Q} \hat{t}'(s)$. By the definition of β, β maps each step in Q either to itself or to a successful C&S. The cost t′(s) of each step $s \in Q$ is C_1. Therefore, if $s \notin \mathrm{range}(\beta)$, $\hat{t}'(s) = 0$, if $s \in \mathrm{range}(\beta)$, and s is not a successful C&S, then $\hat{t}'(s) = C_1$, and if s is a successful C&S that is part of operation S, then it follows from Theorem 48 that the $\hat{t}'(s) = O(c(S))$.

We will now use mapping γ to bill the amortized costs of all the steps in range(β) (the amortized cost of the rest of the steps is 0) to steps of the solo executions of operations performed during E. For each operation S executed during E, we define the amortized cost $\hat{t}'(S)$ to be the sum of the amortized costs of steps in Q that are mapped to the steps of the solo execution of S by γ, i.e. $\hat{t}'(S) = \sum_{\substack{s\in \mathrm{range}(\beta)\\ \gamma(s)\in \mathrm{solo}(S)}} \hat{t}'(s) = \sum_{\substack{s\in Q\\ \gamma(s)\in \mathrm{solo}(S)}} \hat{t}'(S)$. Since the function γ maps every step in range(β) to a step of the solo execution of some operation in E, $\sum_{s\in Q} \hat{t}'(s) = \sum_{S\in E} \sum_{\substack{s\in Q\\ \gamma(s)\in \mathrm{solo}(S)}} \hat{t}'(S) = \sum_{S\in E} \hat{t}'(S)$.

Before proving our bound on the amortized cost of the operations, we show that γ is injective, i.e. no two steps from range(β) are mapped to the same step in a solo execution. Function γ maps successful C&S's that are part of operation S to the C&S's of the same type in the solo execution of S. Since no more than one C&S of each type can be part of S, no two C&S's can be mapped to the same C&S step by γ. Let us show that γ cannot map two curr_node updates to the same curr_node update in the solo execution. This is true, because if two updates u_1 and u_2 are mapped to the same update u', then it means that both u_1 and u_2 were performed during the execution of S and they both set curr_node to the same node, but by Proposition 46 this is not possible. So, γ is injective.

Since γ is injective, it follows from Proposition 28 that for any operation S, the number of steps in range(β), which have their cost assigned by γ to S is O(n(S)), where n(S) is the number of elements in the dictionary when S is invoked. Also for any operation S, there are no more than three successful C&S's that are part of it, and therefore among the steps that have their cost assigned to S, no more than three can be successful C&S's steps (whose amortized cost $\hat{t}'$ is O(c(S))), and the rest are curr_node pointer updates in line 8 of SearchFrom (whose amortized $\hat{t}'$ cost is C_1). So, $\hat{t}'(S) \leq 3 \cdot O(c(S)) + C_1 \cdot O(n(S)) = O(n(S) + c(S))$. Since $t(E) < t'(E) = \sum_{S \in E} t'(S) = \sum_{S \in E} \hat{t}'(S)$, and $\hat{t}'(S) = O(n(S) + c(S))$, it follows that $t(E) \in O\left[\sum_{S \in E} (n(S) + c(S))\right]$.■

Since the cost of the entire execution $t(E) \in O\left[\sum_{S \in E} (n(S) + c(S))\right]$, the amortized cost of an operation S in E, denoted $\hat{t}(S)$, is O(n + c(S)). The average cost of an operation during E, denoted $\bar{t}_E(S)$, is $O\left[\frac{\sum_{S \in E} (n(S) + c(S))}{m}\right]$ where m is the total number of operations performed during E. As we showed at the beginning of the section, one can also use the following two bounds, that are less tight if convenient: $\bar{t}_E(S) = O(\max(n) + \max(c))$ and $\bar{t}_E(S) = O(m)$.

It is also apparent from our proofs that there are no large constant factors in the complexity of our algorithms.

4 Skip Lists

4.1 High-level description

A skip list is a dictionary data structure that stores the nodes on several levels. The nodes that are on the same level are connected to one another like the nodes of a linked list. Therefore, techniques similar to those described in the previous chapter can be applied to maintain the structure of the nodes. There are, however, several new issues that arise for a lock-free skip list implementation. We will describe these issues and outline the possible approaches to deal with them in the next few subsections. Then we will present lock-free algorithms for the skip list and prove their correctness.

4.1.1 The skip list data structure

A skip list [Pug90] is a dictionary data structure, supporting searches, insertions, and deletions. Skip lists are a probabilistic alternative to balanced and self-adjusting trees. Although the worst-case cost of operations on the skip list is high, the expected cost of any operation is O(log(n)), where n is the number of elements in the skip-list. The expectation is taken over the random numbers generated inside the algorithms; there is no assumption about the distribution of the inputs, except that it is assumed that the input does not depend on the generated random numbers. In other words, the adversary constructing a worst-case sequence of operation has no knowledge of or influence over the random numbers generated inside the algorithms. Skip lists are balanced probabilistically, and therefore their algorithms are simpler and, according to [Pug90], often faster (within a constant factor) than the algorithms for alternative data structures, such as balanced trees (e.g. AVL trees [AVL62]) or self-adjusting trees (e.g. splay trees [ST85]). Skip lists are also space-efficient. In this section we will present skip lists as they were originally introduced in [Pug90].

When we are searching through a sorted linked list, we may need to examine every node of the list before we find the one we are looking for. Now suppose, every second node has a pointer to the node following its successor. If we perform a search in such a list, we can use these additional pointers to skip ahead, and as a result, we will need to visit no more than $\lceil n/2 \rceil$ + 1 nodes, where n is the number of keys in the list. Extending this idea, if we give every 2^i-th node a pointer 2^i nodes ahead (Figure 22), we will need to examine at most $\lceil \log_2 n \rceil$ nodes during the search. Notice some similarities between this data structure and a perfectly balanced BST: the node that has the biggest number of pointers is like a root (node D on Figure 22), the nodes that have one pointer less are its children, and so on. The only difference is that if the number of keys in the list is not a power of 2, part of the right "subtree" of the "root" collapses into a tail node. This data structure could be used for searching, but to perform insertions and deletions, one would need to rebalance the whole list, which would take too much time.

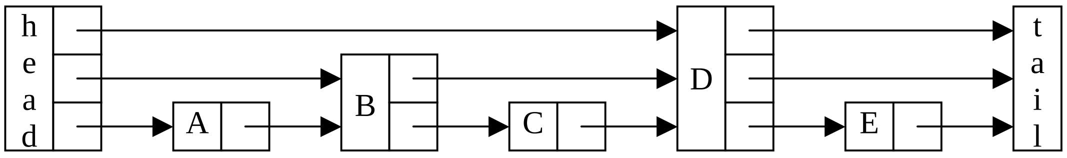


Figure 22: Linked list with additional pointers.

Skip lists exploit the idea of using the extra pointers to "skip" the nodes during the search, but the number of forward pointers of a node is chosen randomly, instead of being dependent on the node's position. The probability distribution of the number of forward pointers is designed to approximate the above data structure, i.e. with probability p a node has one forward pointer, with a probability p^2 it has two, with a probability p^3 it has three, and so on. Parameter p is usually chosen in the interval (0, ½]. Pugh suggests using p = ¼, unless the variance of the running times is a primary concern, in which case p = ½ should be chosen.

Figure 23 shows an example of a skip list. Obviously, some possible configurations of the skip list yield poor execution times (e.g. when all nodes have only one forward pointer), but Pugh shows that they are very rare: the probability of a search taking more than k times the expected execution time decreases exponentially with k.

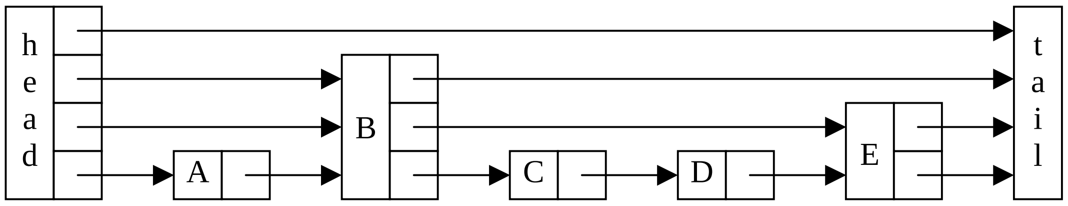


Figure 23: Example of a skip list.

4.1.2 Lock-free skip list design

The skip list design we use is slightly different from the one originally introduced by Pugh. The differences are mostly cosmetic – they do not change the main properties of the data structure, but they make it easier to reuse our linked-list algorithms from the previous chapter, changing them as little as possible. We will save the discussion on how the data structure can be tweaked to improve the constant factors in the performance until the end of this chapter.

Figure 24 shows the same skip list as Figure 23, but with our design employed. We replace each node that has k forward pointers with k separate nodes, organized in a *tower*. All the nodes of a tower store the same key and are connected by pointers from top to bottom. The bottom node of a tower is called a *root node*, and it acts as a representative of the whole tower. Only the root nodes store the elements associated with the keys. The first tower of the skip list is called the *head tower*, and the last one is called the *tail tower*. The nodes of these towers have the keys equal to -∞ and +∞ respectively, and they are not associated with any elements. A tower that has H nodes in it is said to have *height* H. Horizontally, the nodes of the skip list are arranged in *levels*: the root nodes are on *level* one, the nodes immediately above them are on level two, and so on. Nodes of the same level form a singly linked list, sorted accordingly to their keys. For each node Q the linked list formed by the nodes of the same level as Q is called a *level-list* of Q.

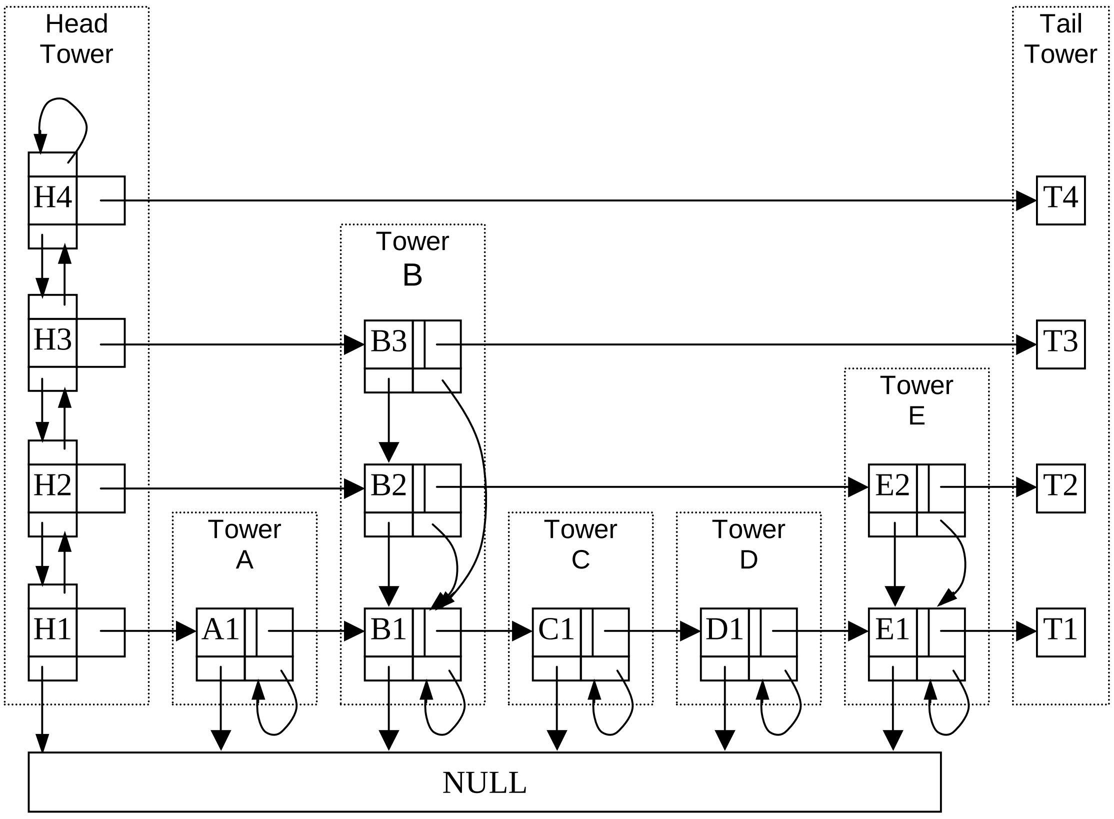


Figure 24: Lock-free skip list design.

Let us now examine the fields of the nodes in detail. We will first describe the fields of the usual nodes, and then the fields of the nodes of the head and the tail towers (which are different).

A node Q that is not a node of the head or tail tower has the following fields (see Figure 25):

- key, back_link, succ – these are same as in the lock-free linked lists. If Q is a root node, it also has an element field.
- down – a pointer to the node below, i.e. the node that belongs to the same tower and is one level lower than Q. If Q is a root node, this pointer is null.
- tower_root – a pointer to the root of the tower.

The upper-left box of a node in the diagrams contains a name of the node (e.g. "Q" on Figure 25), which is only used for convenience and does not represent an actual field. The keys of the nodes are not shown on the diagrams.

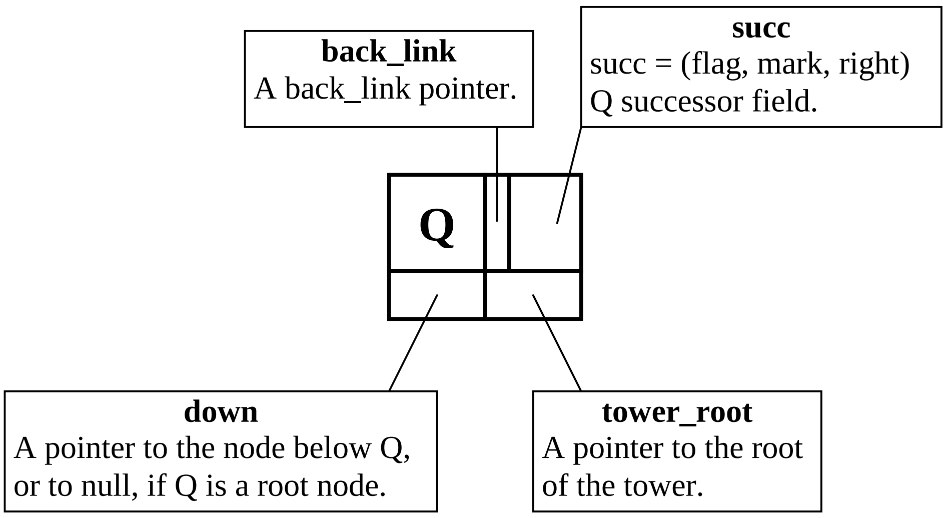

back_link
A back_link pointer.
succ
succ = (flag, mark, right)
Q successor field.
Q
down
A pointer to the node below Q, or to null, if Q is a root node.
tower_root
A pointer to the root of the tower.

Figure 25: Fields of a node.

Nodes of the head tower do not have element pointers, back_links or tower_root pointers, but they have *up* pointers, pointing to the nodes above. The top node of the head tower has its up pointer pointing to itself. Nodes of the tail tower contain only keys and nothing else.

The head and tail towers are created when the data structure is initialized and they are not modified after that. These towers have equal height, which should be greater than the heights of all the other towers of the list. For simplicity, we use the same approach as Pugh and limit the height of the towers in the skip list. We introduce a constant *maxLevel* and ensure that the height of all the normal towers is strictly less than maxLevel, and the height of the head and the tail towers is maxLevel. The value of maxLevel should be chosen so that it does not hamper the performance of the data structure. For example, if N is the upper bound on the number of elements in the skip list, and p is a parameter of the probability distribution of the heights of the towers (introduced in the previous section), then, as Pugh shows, choosing maxLevel = L(N) = $\log_{1/p}(N)$ will not hamper the performance of the data structure. It is also possible to set maxLevel dynamically, but that makes the algorithms more complicated.

4.1.3 Implementation issues

As we pointed out in a previous subsection, each level of a skip list can be viewed as a linked list. Therefore, we will use the algorithms from the previous chapter to implement insertions and deletions of individual nodes. The main challenge when doing so is ensuring that all the operations can still work efficiently.

In our implementation, insertions build the towers from bottom to the top, i.e. first the root node at level one is inserted, then if the tower must have height of two or greater, the node at level two is inserted, and so on. The insertions will be linearized at the moment when the root node is inserted, since after that moment, all the searches are able to find the key of that node.

Let us consider how to implement deletions. Suppose tower A is being deleted and there are several nodes in this tower. Since the root node is representative of the tower, and all the searches end at the bottom level, it makes sense to choose the moment when the root node of A gets marked as the linearization point for the deletion. At that moment the root node becomes a logically deleted node in its level-list. We also say that at that moment tower A and all its nodes become *superfluous*. Generally speaking, the fact that node Q is superfluous does not impose any restrictions on Q's status in its level-list. I.e., applying the definitions introduced in the previous chapter, there can be superfluous regular nodes, superfluous logically deleted nodes, and superfluous physically deleted nodes.

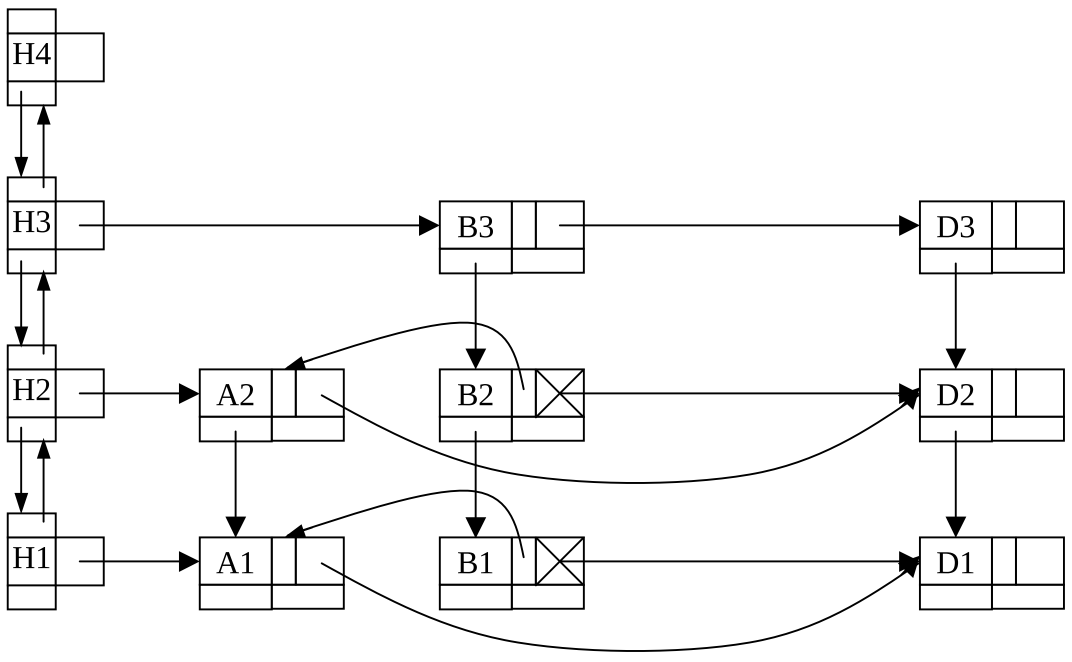
It is important to physically delete the nodes after they become superfluous, because if superfluous nodes are not physically deleted, the searches will need to traverse these nodes, which will result in decreased performance.

Consider two possible options for implementing deletions. One is to start the deletions from the bottom, i.e. first delete the root node, and then delete the other nodes of the tower. The other option is to start the deletions from the top. Obviously, if the first option is used, the data structure may contain superfluous regular nodes and superfluous logically deleted nodes. Let us illustrate that this can also happen if the second option is chosen. Consider the following scenario: process P1 starts an insertion of a tower of height H, inserts the nodes at levels 1…H – 1, then starts inserting the last node at level H, but gets delayed before performing the C&S. Before P1 takes any further steps, another process P2 starts deleting this tower, and deletes nodes at levels 1…H – 1. Finally, P1 resumes its operation and inserts the node at level H. This node is a superfluous regular node. So, for either implementation of deletions, there can be superfluous regular nodes and superfluous logically deleted nodes in the data structure. As we will now show, this leads to a problem with the implementation of searches.

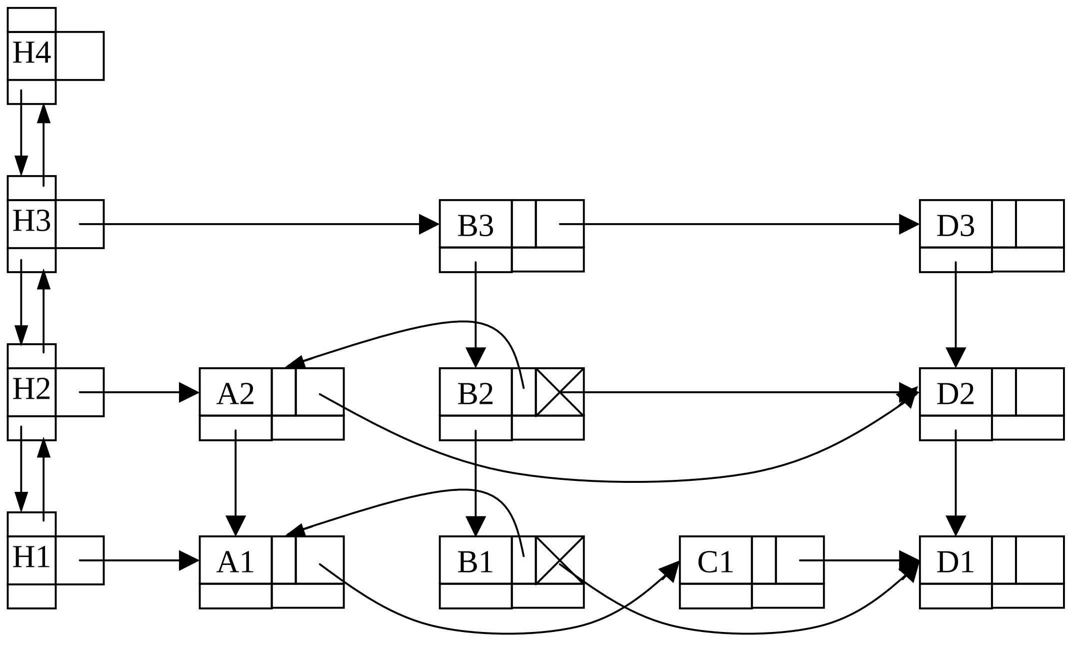
Suppose the searches are implemented in a straightforward way, i.e. using the SearchFrom routine from the previous chapter to move right through a level-list and going down one level when curr_node.key ≤ k < next_node.key. Consider the configuration shown in Figure 26 (a), where B3 is a superfluous regular node. Suppose some process P performs an insertion of a new tower C of height 1 between B and D from that configuration (see Figure 26 (b)). Let k be the key of that tower. Suppose another process starts a search for k. At some point the search will set next_node = B3, and since B3 is a regular node, the search will just move further. B3.key < k < D3.key, so the search will set curr_node = B3, next_node = B3.right = D3, and then go down, setting curr_node = B2. B2's right pointer is set to D2, which has a key greater than k, so the search will go down once again, setting curr_node = B1. The right pointer of B1 is set to D1, and B1.key < k < D1.key, therefore the search will return NO_SUCH_KEY, which would be incorrect, because the insertion of tower C with key k completed before the search began.

Consider two possible solutions to this problem. The first is to make the search follow the back_link pointers if it sees that curr_node is marked, the second is to check if the node is superfluous before entering it, and if it is, delete it. Our algorithms make use of the second idea, because helping the deletions is "useful" for the general progress of the system, and backtracking is not.

One last issue with the lock-free skip list implementation is maintaining a pointer to the node from which the searches start. Ideally, the searches should begin from the highest node of the head tower that has a non-null right pointer. Maintaining a shared pointer to this node as a part of a lock-free skip list data structure is possible, but it makes the algorithms more complicated. Instead, in our implementation, when the search is initialized, it starts the from the bottom node of the head tower, goes up until it finds a node with a null right pointer, and then begins a normal search from that node. Making searches do this additional work adds only a constant factor to their cost. A pointer to the bottom node of the head tower is called the *head pointer*.



(a) Initial configuration: tower B partially deleted.



(b) Tower C has been inserted.

Figure 26: A problem with searches.

4.2 Algorithms

In this section we present our algorithms. The data structure and notation were explained above, so we will not explain it again here.

4.2.1 Pseudo code

Figures 27-38 show various routines used by our data structure. Figures 27, 31, and 33 show the pseudocode for the three major routines: Search_SL, Insert_SL, and Delete_SL; the rest of the figures show auxiliary routines. The variables of the Node type can be either node pointers, or one of the special return values (DUPLICATE_KEY, NO_SUCH_KEY, NO_SUCH_NODE) used to indicate that the operation failed.

```
Search_SL (Key k): RNode
1   (curr_node, next_node) = SearchToLevel_SL(k, 1)
2   if (curr_node.key == k)
3       return curr_node
4   else
5       return NO_SUCH_KEY
```

Figure 27: The Search_SL routine searches for a root node with the supplied key.

```
SearchToLevel_SL (Key k, Level v): (Node, Node)
1   (curr_node, curr_v) = FindStart_SL(v)
2   while (curr_v > v)                          // search down to level v + 1
3       (curr_node, next_node) = SearchRight(k, curr_node)
4       curr_node = curr_node.down
5       curr_v--
6   (curr_node, next_node) = SearchRight(k, curr_node)        // search on level v
7   return (curr_node, next_node)
```

Figure 28: The SearchToLevel_SL routine starts from the head tower and searches for two consecutive nodes on level v, such that the first has a key less than or equal to k, and the second has a key strictly greater than k.

```
FindStart_SL(v): (Node, Level)
1   curr_node = head
2   curr_v = 1
3   while ((curr_node.up.right.key != +∞) || (curr_v < v))
4       curr_node = curr_node.up
5       curr_v++
6   return (curr_node, curr_v)
```

Figure 29: The FindStart routine searches the head tower for the lowest node that points to the tail tower.

SearchRight (Key k, Node *curr_node): (Node, Node)

```
next_node = curr_node.right
while (next_node.key <= k)
    while (next_node.tower_root.mark == 1)      // if the tower is superfluous, delete next_node
        (curr_node, status, result) = TryFlagNode(curr_node, next_node)
        if (status == IN)                       // next_node's predecessor curr_node was flagged
            HelpFlagged(curr_node, next_node)
        next_node = curr_node.right
    if (next_node.key <= k)
        curr_node = next_node
        next_node = curr_node.right
return (curr_node, next_node)
```

Figure 30: The SearchRight routine starts from node curr_node and searches the level for two consecutive nodes such that the first has a key less than or equal to k, and the second has a key strictly greater than k.

TryFlagNode (Node *prev_node, Node *target_node): (Node, status, result)

```
loop
    if (prev_node.succ == (target_node, 0, 1))    // predecessor is already flagged
        return (prev_node, IN, false)
    result = c&s(prev_node.succ, target_node, 0, 0), (target_node, 0, 1))
    if (result == (target_node, 0, 0))            // c&s was successful
        return (prev_node, IN, true)
     /* Failure */
    if (result == (target_node, 0, 1))            // failure due to flagging
        return (prev_node, IN, false)
    while (prev_node.mark == 1)                   // possibly failure due to marking
         prev_node = prev_node.back_link
    (prev_node, del_node) = SearchRight(target_node.key – $\varepsilon$, prev_node)
    if (del_node != target_node)                  // target_node was deleted from the list
        return (prev_node, DELETED, false)
end loop
```

Figure 31: The TryFlagNode routine attempts to flag the predecessor of target_node.

Insert_SL (Key k, Elem e): RNode

```
(prev_node, next_node) = SearchToLevel_SL(k, 1)
if (prev_node.key == k)
    return DUPLICATE_KEY
newRNode = new rnode(key = k, elem = e, down = null, tower_root = self)  // create root node
newNode = newRNode                       // pointer to the node currently being insertied into the tower
tH = 1
while ((FlipCoin() == head) && (tH <= maxLevel – 1))    // determine the desired height of the tower
    tH++
curr_v = 1                               // The level at which newNode is to be inserted
loop                                     // Each iteration increases the height of the new tower by 1
    (prev_node, result) = InsertNode(newNode, prev_node, next_node)
    if ((result == DUPLICATE_KEY) && (curr_v == 1))
        free newNode
        return DUPLICATE_KEY
    if (newRNode.mark == 1)                              // if the tower became superfluous
        if ((result == newNode) && (newNode != newRNode))        // if newNode was inserted, and
            DeleteNode(prev_node, newNode)                       // it is not a root node, delete it
        return newRNode
    curr_v ++
    if (curr_v = tH + 1)                                 // stop building the tower
        return newRNode
    lastNode = newNode
    newNode = new node(key = k, down = lastNode, tower_root = newRNode)
    (prev_node, next_node) = SearchToLevel_SL(k, curr_v)
end loop
```

Figure 32: The Insert_SL routine attempts to insert a new tower into the skip list.

InsertNode(Node *newNode, Node *prev_node, Node *next_node): (Node, Node)

```
if (prev_node.key == newNode.key)
    return (prev_node, DUPLICATE_KEY)
loop
    prev_succ = prev_node.succ
    if (prev_succ.flag == 1)                  // if prev_node is flagged
        HelpFlagged(prev_node, prev_succ.right)
    else
        newNode.succ = (next_node, 0, 0)
        result = c&s(prev_node.succ, (next_node, 0, 0),  (newNode, 0, 0))
        if (result == (newNode, 0, 0))       // SUCCESS
            return (prev_node, newNode)
        else                                 // FAILURE
            if (result == (*, 0, 1))             // failure due to flagging
                HelpFlagged(prev_node, result.right)
            while (prev_node.mark == 1)        // possibly a failure due to marking
                prev_node = prev_node.back_link
    (prev_node, next_node) = SearchRight(newNode.key, prev_node)
    if (prev_node.key == newNode.key)
        return (prev_node, DUPLICATE_KEY)
end loop
```

Figure 33: The InsertNode routine attempts to insert node newNode into the list. Nodes prev_node and next_node specify the position where InsertNode will attempt to insert newNode.

```
Delete_SL(Key k): RNode
(prev_node, del_node) = SearchToLevel_SL($k – \varepsilon$, 1)
if (del_node.key != k)                    // k is not found in the list
    return NO_SUCH_KEY
result = DeleteNode(prev_node, del_node)
if (result == NO_SUCH_NODE)
    return NO_SUCH_KEY
SearchToLevel_SL(k, 2)                    // Deletes the nodes at the higher levels of the tower
return del_node
```

Figure 34: The Delete_SL routine attempts to delete a tower with the supplied key.

```
DeleteNode(Node *prev_node, Node *del_node): Node
(prev_node, status, result) = TryFlagNode(prev_node, del_node)
if (status == IN)
    HelpFlagged(prev_node, del_node)
if (result == false)
    return NO_SUCH_NODE
return del_node
```

Figure 35: The DeleteNode routine attempts to delete node del_node.

```
HelpMarked(Node *prev_node,
           Node *del_node)
next_node = del_node.right
c&s(prev_node.succ, (del_node, 0, 1), (next_node, 0, 0))
```

Figure 36: The HelpMarked routine attempts to physically delete the marked node del_node.

```
HelpFlagged(Node *prev_node, Node *del_node)
del_node.back_link = prev_node
if (del_node.mark == 0)
    TryMark(del_node)
HelpMarked(prev_node, del_node)
```

Figure 37: The HelpFlagged routine attempts to mark and physically delete the successor of the flagged node prev_node.

```
TryMark(Node del_node)
repeat
    next_node = del_node.right
    result = c&s(del_node.succ, (next_node, 0, 0), (next_node, 1, 0))
    if (result == (*, 0, 1))          // failure due to flagging
        HelpFlagged(del_node, result.right)
until (del_node.mark == 1)
```

Figure 38: The TryMark routine attempts to mark the node del_node.

The Search_SL routine (Figure 27) calls the SearchToLevel_SL routine in its first line to find a root node with key k, and then uses the first of the two root nodes returned to determine if there is such a root node (and hence, a tower) in the list.

The SearchToLevel_SL routine (Figure 28) is used to perform the searches in the skip list. This routine takes a key and a level as its arguments. It starts by calling the

FindStart routine to locate the lowest node of the head tower, such that that node's right pointer is pointing to a tail tower (i.e. no other tower in the skip list, except head and tail, has nodes that high), and it is of level v or higher. SearchToLevel_SL uses that node as the starting point of its search. Then SearchToLevel_SL executes the loop in lines 2-6. In each iteration of the loop, it invokes SearchRight, which moves right through the level-list, until it finds a node with a key greater than k. SearchRight returns pointers to two consecutive nodes curr_node and next_node, that satisfy the following condition at some point of time during the execution of SearchRight: curr_node.right = next_node and curr_node.key ≤ k < next_node.key. Then SearchToLevel_SL moves into a node below curr_node (line 4), and enters the next iteration of the loop. It exits the loop when it reaches level v, and returns its current values of curr_node and next_node in line 7.

The FindStart_SL routine (Figure 29) accepts one parameter – level v. It searches the head tower, starting from the bottom and going up, until it finds a node that is of level v or higher and that has its right pointer pointing to a node of a head tower.

The SearchRight routine (Figure 30) is somewhat similar to the SearchFrom routine (see Subsection 3.2.2), which we implemented for our linked lists. It searches the level-list looking for two consecutive nodes curr_node and next_node such that curr_node.key ≤ k < next_node.key. The only difference from the SearchFrom routine is that SearchRight deletes the superfluous nodes along its way (lines 3-6), whereas SearchFrom deleted logically deleted nodes. To delete a superfluous node next_node, SearchRight first calls TryFlagNode in line 4 to flag next_node's predecessor and then, if next_node is still in the list (line 5), it calls HelpFlagged to physically delete it (line 6).

The TryFlagNode routine (Figure 31) is very similar to the TryFlag routine (see Subsection 3.2.2) implemented for the linked list, but, unlike TryFlag, TryFlagNode always returns a non-null pointer to a node it was trying to flag as its first parameter. SearchRight uses this node to continue searching from it after it is done with a deletion.

The Insert_SL routine (Figure 32) attempts to insert a new tower into the skip list. It accepts two arguments – the key and the element of the new tower, and it returns the root node of the new tower if the insertion is successful. It starts by calling SearchToLevel_SL in line 1 to determine if a tower with key k already exists. If not, it creates the new root node (line 4), randomly determines the height of the tower it is going to insert (lines 6-8), and enters the loop in lines 10-25. Each complete iteration of that loop increases the height of the new tower by one. Insert_SL exits from that loop either if it finishes the construction of the new tower (lines 20-21), or if the construction of a new tower gets interrupted in some way. If the construction gets interrupted before Insert_SL even inserts the root node, Insert_SL exits in line 14, reporting an unsuccessful insertion. If the construction gets interrupted after the root node is already inserted, Insert_SL exits in line 18. In the first case the construction gets interrupted by the insertion of a root node with key k, and in the second case – by the marking of a root node of the tower Insert_SL is building.

The InsertNode routine (Figure 33) is quite similar to the Insert routine (see Subsection 3.2.2), which we implemented for linked lists. Unlike the Insert routine, InsertNode has 3 parameters: a node newNode it is trying to insert, and a couple of nodes prev_node and next_node that were consecutive at some point of time, such that prev_node.key ≤ newNode.key < next_node.key. I.e. prev_node and next_node provide InsertNode with (possibly outdated) information on where to insert newNode. Also, InsertNode returns two Node type variables when it completes: the first variable is the last value of its prev_node pointer (in the case of a successful insertion this is a predecessor of newNode at the moment when the insertion was performed), the second variable is either newNode in the case of a successful insertion, or DUPLICATE_KEY in case of an unsuccessful one.

The Delete_SL routine (Figure 34) attempts to delete a tower with a supplied key k from the skip list, and returns the root node of the deleted tower in case of a successful deletion, or NO_SUCH_KEY in case of a failure. It starts by performing a search (line 1) to find a root node with key k. If such a node exists, it attempts to delete it by invoking the DeleteNode routine in line 4, and, if successful, it deletes the rest of the nodes of that tower by performing a search for key k. Since the searches delete superfluous nodes on their way, this search deletes all the nodes of del_node's tower.

The DeleteNode routine (Figure 35) is similar to the Delete routine (see Subsection 3.2.2), which we implemented for linked lists. It is supplied with the parameters that specify the node it has to delete (del_node) and the node that was its predecessor (prev_node) at some point of time. DeleteNode starts by trying to flag del_node's predecessor by calling the TryFlagNode routine. If TryFlagNode reports that del_node is still in the list, DeleteNode physically deletes it by calling HelpFlagged in line 3. Then, if the result returned by TryFlagNode was true (success), it returns del_node in line 6, otherwise it returns NO_SUCH_NODE in line 5.

The HelpMarked, HelpFlagged and TryMark routines are exactly the same as for linked lists.

4.3 Correctness

In this section we will give a proof of correctness of our algorithms implementing the skip list data structure.

4.3.1 Invariants for the levels of the skip list

In this subsection we will show that our skip list maintains the proper horizontal structure, i.e. that each level of the skip list is a linked list. As described in the previous section, our data structure uses algorithms very similar to the algorithms of the previous chapter to modify individual levels of the skip list. Therefore the invariants and the proofs for the horizontal structure will be very similar to those of the previous chapter.

The definitions of regular, logically deleted, and physically deleted nodes for the skip list are the same as for the linked list (see Def. 2-4). The definition of preinserted nodes is slightly different, and is stated below (compare with Def. 1 from the previous chapter).

Def 8: A node of the skip list is said to be a *preinserted* node, if it was created, but has not yet been inserted into the list. More precisely, node n is preinserted if it is referred to by a newNode or a newRNode pointer in the Insert routine and the C&S in line 9 of the InsertNode routine called with the first parameter equal to n has not yet been successfully executed.

The next definition formally defines the notion of levels in the skip list. Recall that when the skip list is initialized, the head and tail towers of height maxLevel are created, and the structure of these towers does not change throughout the execution.

Def 9: Let us number the nodes of the head tower 1…maxLevel from bottom to top. We say that a node of the head tower *belongs to level* v if it is numbered v. For other nodes of the skip list, we say that a node *belongs to level* v, if it is a regular node at some point during the execution (i.e. it cannot be a preinserted node), and when it first became a regular node, its predecessor belonged to level v. *Level* v of the skip list is the set of nodes that belong to level v.

Note that any node of the list that is not a preinserted node, was a regular node at some point of time during the execution, because when the node is inserted into the list, it is not marked. Therefore, all non-preinserted nodes of the skip list belong to some level. Also note that Def 9 implies that the level of a node never changes. To prove that the levels of the skip list have the proper structure, we prove several invariants. We prove that invariants 1 and 3-5 (see the previous chapter) hold for all the nodes of the skip list. We also prove that invariant 2 holds for each level of the skip list:

Inv 2′: Invariant 2 holds for the set of nodes at level v (for each $1 \le v \le$ maxLevel), with the node of the head tower playing the role of the head node, and the node of the tail tower playing the role of the tail node.

Note that if Inv 2′ holds, then all the regular and logically deleted nodes of level v (for each $1 \le v \le$ maxLevel) form a linked list. We also prove invariants 6 and 7, given below. Inv 6 actually follows from Inv 2, but by proving Inv 6 separately, we will be able to reuse some of the proofs from the previous chapter.

Inv 6: For any node n, if n belongs to level v and n is not a node of the tail tower, then node m = n.right also belongs to level v.

Inv 7: A non-null back_link pointer is always pointing to a node with a smaller key, i.e. if n.back_link ≠ null, then n.back_link.key < n.key.

It is easy to see that Inv 7 holds for the lock-free linked list (it follows from Inv 1, Inv 4, and Proposition 15). Here it is convenient to prove it as a separate statement, because the searches work differently than in the linked list.

We will now prove the invariants 1, 2′, 3-7 always hold. The proof will be very similar to the proof of the invariants in the previous chapter. We will start by proving the 5th invariant, as it is the easiest one. Then we will prove invariants 1, 2′, 3, 6, and 7 by induction on the number of modifications performed on our data structure by the algorithms. Finally, we will prove Inv 4. Along the way we will prove several useful propositions and lemmas.

Theorem 50: Invariant 5 holds for all the nodes of the skip list.
Proof:

The invariant obviously holds when the skip list is empty. When a new node is created, its successor pointer is set to be unflagged and unmarked (line 8 in InsertNode). The successor pointers of the nodes that are part of the list can only be modified by one of the four C&S operations: line 9 in InsertNode, line 4 in TryFlagNode, line 2 in HelpMarked, and line 3 in TryMark. None of them makes the successor field both marked and flagged, and therefore invariant 5 always holds. ■

Proposition 51: Once a node is marked, its successor field never changes.

Proof:

It is easy to see that none of the C&S's can change a marked successor pointer. ■

In the next three propositions we prove that SearchRight and SearchToLevel_SL return consecutive nodes with the correct keys. SearchRight and TryFlagNode call each other recursively, so we start by proving some weak postconditions for them together.

Proposition 52: If Inv 1 and Inv 7 hold up to time T, then the following postconditions hold for all SearchRight and TryFlagNode routines that finish before T:

- If SearchRight(k, n) returns (n1, n2), and n.key ≤ k, then n1.key ≤ k.
- If TryFlagNode(n1, n2) returns (n, status, result), and n1.key < n2.key, then n.key < n2.key.

Proof:

We prove the proposition by induction on the number of completed invocations of the SearchRight and TryFlagNode routines. The base case (after zero completions) is trivial. Let us prove the induction step. Suppose at some point of time the proposition claim holds for all completed invocations of SearchRight and TryFlagNode. Let us take the invocation I that completes next and prove that the claim holds for it as well.

Case 1: I is an invocation of SearchRight. Let (k, n) be its arguments, and (n1, n2) be the node pointers it returns. Assume n.key ≤ k. Node pointer n1 is the last value of curr_node before the routine returns. If we show that curr_node.key ≤ k at any time, then we prove the proposition.

When the SearchRight routine is invoked, curr_node = n, and n.key ≤ k. The value of curr_node can be modified in line 4 or in line 9. If it is modified in line 4, then it is set to the value returned by the TryFlagNode routine. Note that by the induction hypothesis the proposition holds for that invocation of TryFlagNode. Let n1′ and n2′ be the arguments of that TryFlagNode. Since line 4 was executed, the loop condition in line 2 was true, and therefore n2′.key = next_node.key ≤ k. Also, note that the value of next_node could be set only in line 1, 7, or 10, and therefore by Inv 1, n1′.key < n2′.key. Thus, by the induction hypothesis, immediately after curr_node is modified in line 4 curr_node.key < n2′.key ≤ k. If curr_node is modified in line 9, then curr_node ≤ k, because line 9 can only be executed if the condition in line 8 is true. So, curr_node.key ≤ k at any time during the execution of SearchRight.

Case 2: I is an invocation of TryFlagNode. Let (n1, n2) be its arguments, and (n, result) be its return values. Assume n1.key < n2.key. TryFlagNode returns n equal to the last value of its prev_node pointer. If we show that prev_node.key < n2.key at all times, then we prove the proposition.

When the TryFlagNode routine is invoked, prev_node = n1, and n1.key < n2.key. The value of prev_node can be modified in line 10 or in line 11. If it is modified in line 10, then by Inv 7, the key of prev_node decreases. If prev_node is modified in line 11, then it is returned by SearchRight. By the induction hypothesis, the proposition applies to this invocation of SearchRight. Before the execution of SearchRight, prev_node.key < n2.key, and therefore prev_node.key < n2.key – ε. Thus, by the induction hypothesis, after line 11 is executed, prev_node.key ≤ n2.key – ε < n2.key. So, prev_node.key < n2.key at any time during the execution of TryFlagNode. ■

Proposition 53 (Weak postconditions for SearchRight): If Inv 1 and 7 hold up to time T, then for all executions of SearchRight(k, n) that finish before T, the following is true: if n.key ≤ k, and (n1, n2) is the pair of nodes SearchRight returns, then n1.key ≤ k < n2.key, and there exists a point of time during the execution, such that at that time n1.right = n2.

Proof:

Since Inv 1 and 7 hold until SearchRight returns, Proposition 52 applies to this invocation of SearchRight, and n1.key ≤ k. Since the loop in lines 2-10 exits only when the condition in line 2 is false, we can conclude that n2.key > k. Let T1 be the moment of time when variable next_node is last assigned a value. That can be in line 1, 7, or 10. In any case, next_node is assigned a value of curr_node.right. Also, at T1 curr_node = n1, because the value of curr_node can only be changed in line 4 or 9, and if it is changed, next_node is changed as well. So at time T1, n2 = next_node = n1.right. ■

Proposition 54 (Weak postconditions for SearchToLevel_SL): If Inv 1 and 7 hold up to time T, then for all executions of SearchToLevel_SL(k, v) that finish before T, the following is true: if 1 ≤ v < maxLevel, and (n1, n2) is the pair of nodes SearchToLevel_SL returns, then n1.key ≤ k < n2.key, and there exists a point of time during the execution, such that at that time n1.right = n2.

Proof:

First notice that at any time during the execution of SearchToLevel_SL after curr_node is initialized, curr_node.key ≤ k. This is true because right after curr_node is initialized (line 1), it is one of the nodes of the head tower, and thus its key is -∞. Subsequently, curr_node can only be modified by executing SearchRight in line 3 or 6, which, by Proposition 52, will ensure that curr_node.key remains less than or equal to k.

Nodes n1 and n2 are equal to the nodes returned by SearchRight in line 6 of SearchToLevel_SL. We showed that the node that SearchRight starts from has a key less than or equal to k, and since Inv 1 and 7 hold until that SearchRight completes, the claim of the proposition follows from Proposition 52. ■

We are going to prove that invariants 1, 2′, 3, and 6 always hold by proving that they are preserved by all the C&S's performed by our algorithms. We start by proving this for the C&S in the Insert routine.

Proposition 55: The C&S in line 9 of the InsertNode routine preserves invariants 1, 2′, 3, 6, if Inv 7 holds until the execution of this C&S.

Proof:

The C&S is *result = c&s(prev_node.succ, (next_node, 0, 0), (newNode, 0, 0)).*

A successful execution of this C&S swings the right pointer of prev_node from next_node to newNode.

Let us show that before this C&S is executed, there are no other nodes linked to newNode. InsertNode can only be called from Insert_SL, so newNode was created in line 4 or in line 23 of Insert_SL. Notice that in each iteration of the loop in lines 10-25 of Insert_SL, newNode is re-initialized, so InsertNode cannot be called with the same newNode parameter twice. Also, only InsertNode can insert newNode into the data structure, and then it is poised to exit in line 11 without any further attempts to perform a C&S. Therefore, before the C&S is executed, newNode is a preinserted node, and since prev_node is a regular node, the successful execution of this C&S makes newNode regular as well.

The proof that the C&S preserves Inv 1 and 3 goes is exactly as in Proposition 4, except that when proving that Inv 1 is preserved, instead of using Proposition 3, we use Proposition 53 if nodes prev_node and next_node were returned by SearchRight in line 17 of InsertNode, or Proposition 54 if they were returned by SearchToLevel_SL in line 1 or 24 of Insert_SL. Note that we can use Propositions 53 and 54 because we assume that Inv 1 and 7 hold before the C&S.

Let us prove that Inv 6 is also preserved. Suppose prev_node belongs to level v. Then by Def 9 newNode also belongs to level v, and since Inv 6 held before the C&S, next_node also belongs to level v. So, prev_node, newNode, and next_node all belong to the same level, and since after the C&S prev_node.right = newNode and newNode.right = next_node, Inv 6 holds after the C&S.

Since prev_node, newNode, and next_node all belong to the same level, the proof that Inv 2′ holds is the same as the proof that the Inv 2 holds in Proposition 4. ■

As in the previous chapter, here we prove several auxiliary propositions concerning the HelpMarked routine, and then we show that the C&S's in HelpMarked and TryFlagNode preserve Inv 1, 2′, 3, and 6.

Proposition 56: If the HelpMarked routine is invoked with parameters prev_node and del_node, then del_node was marked at some point before this invocation.

Proof:

The HelpMarked routine can be called only from line 4 of the HelpFlagged routine. Before HelpFlagged calls HelpMarked, it ensures that del_node is marked: if the condition in line 2 in HelpFlagged is true, then TryMark(del_node) is called, and it exits only after del_node gets marked. ■

Lemma 57 (Physical deletion): Suppose HelpMarked(prev_node, del_node) successfully executes a C&S in line 2. Then if Inv 2′ held before this C&S, del_node is physically deleted immediately after the C&S.

Proof:

The proof is the same as the proof of Lemma 6, except that Propositions 51 and 56 should be used instead of Propositions 2 and 5. ■

Proposition 58: The C&S in line 2 of the HelpMarked routine preserves invariants 1, 2′, 3, and 6.

Proof:

The C&S is *c&s(prev_node.succ, (del_node, 0, 1), (next_node, 0, 0)).*

The proof that this C&S performs a physical deletion of del_node and preserves invariants 1 and 3 is the same as in Proposition 7, except that Proposition 56 and Lemma 57 should be used instead of Proposition 5 and Lemma 6.

By Proposition 56, del_node got marked before HelpMarked was called. Marked successor fields do not change, so since del_node.right = next_node when line 1 of HelpMarked is executed, del_node.right = next_node immediately before the C&S as well. Therefore, since Inv 6 held before the C&S, del_node and prev_node are on the same level. Since the C&S was successful, prev_node.right = del_node before the C&S, and again, since Inv 6 held before the C&S, prev_node and del_node are on the same level. Therefore all three nodes prev_node, del_node, and next_node are on the same level. Since after the C&S prev_node.right = next_node, Inv 6 holds.

Since prev_node, del_node, and next_node all belong to the same level, the proof that Inv 2′ holds is the same as the proof that the Inv 2 holds in Proposition 7. ■

Proposition 59: The C&S in line 4 of the TryFlagNode routine preserves invariants 1, 2′, 3, and 6.

Proof:

The C&S is *c&s(prev_node.succ, (target_node, 0, 0), (target_node, 0, 1)).*

A successful execution of this C&S flags the successor field of prev_node, which cannot violate invariants 1, 2′, 3, or 6. ■

As in the previous chapter, we now prove two lemmas, which will help us show that the C&S in the TryMark routine preserves Inv 1, 2′, 3, and 6.

Lemma 60 (HelpFlagged is invoked only if a flagged node is detected): Suppose process P invokes the HelpFlagged routine with parameters prev_node = n and del_node = m. Then there exists a time T before the invocation when n.succ = (m, 0, 1).

Proof:

HelpFlagged can be called in line 3 of the DeleteNode routine, in lines 6 or 14 of the InsertNode routine, in line 5 of the TryMark routine, and in line 6 of the SearchRight routine.

Suppose HelpFlagged was called from SearchRight. Then when TryFlagNode was called in line 4 for the last time, it must have returned status = IN, so that TryFlagNode invocation returned from line 3, 6, or 8. If it returned from line 3, then the moment when it executed the previous line (line 2) is a valid moment for T. If it returned from line 6 or 8, then the moment just after it last tried to perform a C&S in line 4 is a valid moment for T. Similarly, if HelpFlagged was called from DeleteNode, then the last time when TryFlagNode, which was called in line 1 of DeleteNode, executed line 2 or line 4, is a valid time for T.

The proofs for the cases when HelpFlagged is called by InsertNode or TryMark are the same as in Lemma 9: if HelpFlagged was called from InsertNode, then the moment when line 4 in InsertNode was executed for the last time is a valid moment for T, and if

HelpFlagged was called from TryMark, then the moment when TryMark executed line 3 for the last time is a valid moment for T. ■

Lemma 61 (Predecessor is still flagged when a node gets marked): Suppose the C&S in line 3 of the TryMark routine successfully marks del_node. Let v be the first parameter of the HelpFlagged routine that called this TryMark routine. Then starting from some time before HelpFlagged was invoked, and until the C&S in TryMark is performed, v.succ = (del_node, 0, 1).

Proof:

The proof is the same as the proof of Lemma 10, except that Propositions 51 and 56 should be used instead of Propositions 2 and 5, and Lemma 60 should be used instead of Lemma 9. ■

Proposition 62: The C&S in line 3 of the TryMark routine preserves Inv 1, 2′, 3, and 6.

Proof:

The C&S is *result = c&s(del_node.succ, (next_node, 0, 0), (next_node, 1, 0)).*

A successful execution of this C&S marks the successor field of del_node. No right pointers change, so Inv 1 and 6 hold. The proof for Inv 2′ and 3 is the same as the proof for Inv 2 and 3 in Proposition 11. ■

Theorem 63: Invariants 1, 2′, 3, 6, and 7 always hold.

Proof:

Initially the list contains no keys and all the invariants obviously hold. Our algorithms modify the data structure only by performing C&S operations or by setting back_links in line 1 of HelpFlagged routine. We shall prove that invariants 1, 2′, 3, 6, and 7 always hold by the induction on the number of such modifications. The base case (0 modifications) is trivial.

Since the keys of the nodes never change, and the back_links of the newly inserted nodes are always null, the C&S operations cannot violate Inv 7. Also, from Propositions 55, 58, 59, and 62, it follows that the C&S's cannot violate Inv 1, 2′, 3, and 6 either. Thus, to prove the theorem, it is sufficient to show that setting the back_link in line 1 of HelpFlagged preserves Inv 1, 2′, 3, 6, and 7. Obviously, only Inv 7 can be affected by such a modification. Suppose HelpFlagged called with parameters prev_node = n and del_node = m executes line 1. It sets m.back_link = n, and by Lemma 60, at some point before HelpFlagged was invoked, n.succ = (m, 0, 1), and since Inv 1 holds until the modification, n.key < m.key. ■

The only invariant we still have to prove is Inv 4. Before we prove it, we prove several auxiliary claims, as in the previous chapter.

Proposition 64: Once a node is physically deleted, it remains physically deleted forever.

Proof:

Same as the proof of Proposition 13, except that Proposition 56 should be used instead of Proposition 5. ■

Lemma 65: For any node m, after some flagged node n is linked to it, m will never have a regular predecessor other than n. When n’s successor field changes, m becomes physically deleted.

Proof:

Same as the proof of Lemma 14, except that Lemma 57 and Proposition 64 should be used instead of Lemma 6 and Proposition 13. ■

Proposition 66: Once back_link is set, it never changes.

Proof:

Same as the proof of Proposition 15, except that Lemmas 60 and 65 should be used instead of Lemmas 9 and 14. ■

Theorem 67: Inv 4 always holds.

Proof:

Same as the proof of Theorem 16, except that Propositions 56 and 66 should be used instead of Propositions 5 and 15, and Lemmas 57 and 61 should be used instead of Lemmas 6 and 10. ■

Proposition 68: For any node m of the skip list, if its back_link pointer is not null, then it is pointing to a node on the same level as m.

Proof:

Back_link pointers can be set only in line 1 of the HelpFlagged routine. Let n = prev_node, m = del_node when line 1 of HelpFlagged is executed. By Lemma 60, there was some time when n was m’s predecessor, and thus by Inv 6, n and m were on the same level. Since by Proposition 66 back_links never change, m’s back_link is always pointing to a node on the same level as m. ■

4.3.2 Critical steps of a node deletion

The definition of the three critical steps of a node deletion was given in the previous chapter (Def 5). We will prove that in the lock-free skip list these critical steps are performed in the same order as in the lock-free linked list.

Proposition 69 (Critical steps of a deletion): The three C&S steps described in Definition 5 can be successfully performed only once for each particular node del_node and only in the order they are listed.

Proof:

The proof goes exactly the same way as the proof of Proposition 17, except that the C&S that flags the node’s predecessor is performed by the TryFlagNode routine in line 4, not by the TryFlag routine. Also, Propositions 51, 56, and 64 should be used instead of Propositions 2, 5, 13, and Lemmas 57, 61, and 65 should be used instead of Lemmas 6, 10, 14. ■

4.3.3 Vertical structure of the skip list

The invariants in the previous subsection were concerned with the horizontal structure of the skip list. We proved that nodes of the skip list are organized horizontally into linked lists. In this subsection we will prove the invariants that will show that the vertical structure of the list is also proper. We will show that the nodes are organized vertically into towers so that the nodes of any tower are connected by down pointers into a linked list from top to bottom and all have the same key.

Def 10: A *root node* is a node that has an element field.

Def 11: Node n *belongs to the tower* of a root node rn if n is a regular, logically deleted, or physically deleted node, and one can get from n to rn by following down pointers, i.e. there exists a set of nodes m_1, m_2, …, m_k such that m1 = n, m_k = rn, and for $1 \leq i \leq k - 1$, m_i.down = m_{i+1}. The *tower* of a root node rn is the set of nodes that belong to the tower of rn.

Def 12: The *height* of a tower is the number of nodes in the tower.

Def 13: A tower is called *superfluous* if its root node is marked. A node is *superfluous* if it belongs to a superfluous tower.

Note that there can be superfluous regular nodes, superfluous logically deleted nodes and superfluous physically deleted nodes. Also, since the nodes never get unmarked, once a node becomes superfluous, it remains superfluous forever.

The nodes of the skip list are created by the Insert_SL routine. We will now show that each invocation of that routine constructs a separate tower.

Proposition 70: Suppose rn is a root node. Let ISL be the invocation of the Insert_SL routine that created rn. Then the following is always true:

- Each node that ISL created either belongs to rn's tower or is preinserted.
- All the nodes of rn's tower were created by ISL.

<u>Proof:</u>

Down pointers of nodes are set when the nodes are created (line 4 or line 23 of the Insert_SL routine) and never change after that. At the time when the nodes are created, their down pointers are always set to point either to null (for root nodes), or to the nodes that were created by the same invocation of the Insert_SL routine. Therefore, by induction on the number of nodes inserted by a particular Insert_SL, all the nodes that were created by the same invocation of Insert_SL belong to rn's tower or are preinserted. Since the nodes can be created only by the Insert_SL routine, and a node cannot belong to more than one tower, the second claim of the proposition follows from the first. ■

Proposition 71: All the nodes of one tower have the same key and their tower_root pointers are set to the root node of that tower. I.e. if node n belongs to the tower of root node rn, then n.key = rn.key, n.tower_root = rn.

<u>Proof:</u>

By Proposition 70, all the nodes of rn's tower are created by the same invocation of the Insert_SL routine. Thus, when they are created (in line 4 for root node, in line 23 for the rest of the nodes), their key is same as rn's key and their tower_root pointer is set to rn. Neither keys nor tower_root pointers ever change, so the proposition always holds. ■

To prove that every tower of the skip list has the proper structure, we prove the following two invariants. These invariants do not apply to preinserted nodes, nodes of the head tower, or nodes of the tail tower.

Inv 8: Node n is a root node if and only if it belongs to level 1. For any non-root node n, if n belongs to level j of the skip list, then node n.down belongs to level j – 1.

Inv 9: Each node of the skip list belongs to some tower, and nodes of each tower form a linked list. I.e. for any node n of the skip list the following holds:

1. Node n belongs to some tower.
2. n.down = null if and only if n is a root node.
3. If n.down ≠ null, then the node n.down belongs to the same tower as n.
4. If n.down ≠ null, then there is no other regular, logically deleted, or physically deleted node m in the skip list such that n.down = m.down.

To prove Inv 8 we need to show that the insertions build the towers properly from the bottom to the top, inserting exactly one node on each level until they complete or get interrupted. We start by proving several auxiliary propositions in this subsection. Then we prove some postconditions for the SearchRight and SearchToLevel_SL routines in Subsection 4.3.4. We use these postconditions to prove the required properties of the InsertNode and Insert_SL routines in Subsection 4.3.5, and finally we prove Inv 8 in Subsection 4.3.6. As we then show, Inv 9 follows easily from Inv 8.

The following lemma shows that the routines that are meant to operate on the single level of the skip list indeed do so.

Lemma 72: Let R be an invocation of SearchRight, InsertNode, DeleteNode, TryFlagNode, HelpMarked, HelpFlagged, or TryMark. If the node pointers given to routine R as parameters point to nodes on some level v, then all the nodes that R accesses during its execution belong to level v. Note that this implies that the nodes R returns also belong to level v.

Proof:

The only way routines SearchRight, InsertNode, DeleteNode, TryFlagNode, HelpMarked, HelpFlagged, and TryMark access the nodes of the skip list is by following either right pointers or back_link pointers of the nodes they already have pointers to. By Inv 6, following a right pointer leads to a node of the same level. By Proposition 68, following back_link also leads to a node of the same level. Therefore, all the nodes that R accesses during its execution belong to level v. ■

The following proposition shows that the SearchToLevel_SL routine returns pointers to nodes on the correct level.

Proposition 73: If Inv 8 holds up to time T then the following is true. Let STL be an invocation of the SearchToLevel_SL(k, v) routine that finishes before time T. If 1 ≤ v < maxLevel, then the nodes that STL returns belong to level v.

Proof:

Let (n, v1) be the values of curr_node and curr_v immediately after STL executes the FindStart_SL routine in line 1. Since the levels of the skip list are defined by the nodes of the head tower, it is easy to see that node n belongs to level curr_v. Also, curr_v ≥ v (see the loop condition in line 3 of FindStart_SL). By Inv 8, every time line 4 of SearchToLevel_SL is executed, the level of the node that curr_node points to decreases by one. By Lemma 72 the level of curr_node does not change in line 3, so each time the loop in lines 2-5 iterates, curr_v is decreased by one and the level of curr_node is decreased by one. Therefore, when the loop exits, curr_node belongs to level curr_v = v, and thus the nodes that SearchToLevel_SL returns also belong to level v. ■

The following two technical lemmas are concerned with the details of the execution of the InsertNode and Insert_SL routines.

Lemma 74: If an InsertNode called with parameter newNode = n fails to insert node n into the skip list, then it returns DUPLICATE_KEY.

Proof:

InsertNode can return in line 11 only if it executes the C&S in line 9 successfully. Successful execution of this C&S inserts n into the level-list, and thus, into the skip list. Thus, if InsertNode fails to insert node n into the list, it cannot return in line 11. Therefore, it will return in line 2 or 19, so it will return DUPLICATE_KEY. ■

Lemma 75: If Insert_SL fails to insert the root node newRNode into the list, it exits in line 14 during the first iteration of the loop in lines 10-25.

Proof:

Insert_SL attempts to insert the node newRNode into the list by calling InsertNode in line 11 during the first iteration of the loop in lines 10-25. If InsertNode routine fails to perform the insertion, then by Lemma 74, it returns DUPLICATE_KEY. Since during the first iteration curr_v = 1, Insert_SL will exit in line 11. ■

The following lemma shows that a tower always has a root node.

Lemma 76: If there are any nodes in the tower of a root node rn, then rn is a regular node, a logically deleted node, or a physically deleted node (i.e. not a preinserted node).

Proof:

Suppose rn’s tower is not empty. Let us examine the invocation ISL of the Insert_SL routine that created rn. By Lemma 75, if it failed to insert rn, it exited in line 14 without creating any more nodes. But that means that there are no nodes in rn’s tower, because, by Proposition 70, only invocation ISL can create such nodes, and the only node that it did create is rn, which is either preinserted or non-existent at any given point of time – a contradiction. ■

The following lemma shows that no critical steps of a non-root node deletion can be made until that node becomes superfluous. For non-root nodes, becoming superfluous can be considered the first deletion step, before flagging of the predecessor.

Lemma 77: If Inv 8 holds up to time T, then for any node n that is not a root node, if the first critical step of n's deletion was performed at some time T1 before T, then n became superfluous before T1.

Proof:

In order for the first critical step of the deletion of node n to be performed, n's predecessor must be flagged by the C&S in line 4 of the TryFlagNode routine. Therefore the TryFlagNode routine must be called with the second argument target_node = n.

TryFlagNode can be called by SearchRight in line 4 or by DeleteNode in line 1. If it was called by SearchRight, then the proposition holds, because next_node.tower_root is marked (line 3), and therefore, since tower_root points to the tower's root node by Proposition 71, next_node is superfluous.

Suppose it was called by DeleteNode in line 1. DeleteNode, in turn, can be called from line 17 of Insert_SL, or from line 4 of Delete_SL. Suppose it was called from line 17 of Insert_SL. Let n be the value of newNode and rn be the value of newRNode just before DeleteNode was called. Node rn is a root node of the tower which this invocation of Insert_SL builds. Since the last InsertNode returned result = n (line 16), n was inserted into the list (i.e. it is not a preinserted node), and thus, by Proposition 70, n belongs to rn's tower. Node rn is marked (line 15), so n is a superfluous node.

Finally, suppose DeleteNode was called from line 4 of Delete_SL. Since we assume that Inv 8 holds, the nodes prev_node and del_node that were returned by SearchToLevel_SL in line 1 belong to level 1. Therefore, by Inv 8, n = del_node is a root node, and we assumed that it is not. ■

Proposition 78: If Inv 8 holds up to time T, then for any node n, if n is not superfluous at T, then n is not marked.

Proof:

If n is a root node, then the proposition follows directly from the definition of a superfluous node. If n is not a root node, then the proposition follows from Lemma 77. ■

4.3.4 The SearchRight and SearchToHeight_SL routines

In this subsection we prove several properties and postconditions for the SearchRight and SearchToLevel_SL routines. We start by proving two simple propositions about marked nodes.

Proposition 79: If n1 and n2 are both marked nodes, and n1.right = n2, then n1 was marked before n2.

Proof:

(Analogue of Proposition 18.) Since marked pointers never change, n1.right = n2 when n1 got marked. By Inv 3, at that time n2 was not marked, so n1 was marked before n2. ■

Proposition 80: Back_links cannot be set to marked nodes. I.e. if n2.back_link = n1, then either n1 is not marked, or n1 got marked later than n2.

Proof:

(Analogue of Proposition 41). By Inv 3, when n2 got marked, n1 was flagged, and therefore not marked. ■

In the previous chapter, when we explored the behaviour of the searches performed by the lock-free list in Proposition 19, we proved that if a search enters a node n, which is unmarked at some time T, then all the nodes the search subsequently traverses are also unmarked at T. That important property helped us to prove the correctness of our linked list algorithms. We will prove a similar property for the skip list, but the searches here are more complicated: the SearchRight routine may update its local node-pointer curr_node by executing the TryFlagNode routine in line 4. TryFlagNode routine, in turn, may call another SearchRight in line 11 to update its pointer prev_node. Therefore, a complete sequence of the nodes traversed during a search must include all the nodes traversed by the SearchRight and TryFlagNode routines invoked during the execution of the search. The following definitions formally define such *node sequences* for the SearchRight, TryFlagNode, and SearchToLevel_SL routines.

Def 14: Suppose R is an execution of the SearchRight or the TryFlagNode routine by process P. Let us record the values of curr_node (for SearchRight) and prev_node (for TryFlagNode) pointers of all the SearchRight and TryFlagNode routines called by P during R. The resulting sequence of nodes is called a *node-sequence of R*, denoted N(R) = $\{n_1, \dots, n_s\}$. For $1 \le i \le s$, let T_i be the time when a curr_node or prev_node pointer gets set to n_i. Note that times T_i are increasing with i. Sequence $\{T_1, \dots, T_s\}$ is called a *time trace of R* and is denoted T(R).

Def 15: The *node-sequence of a SearchToLevel_SL routine* is the concatenation of the node sequences of the SearchRight routines it calls, concatenated in the order the SearchRight routines are invoked. The *time trace of a SearchToLevel_SL* routine is a similar concatenation of the time traces of the SearchRight routines it calls.

In the following two lemmas we prove a couple of weak claims about the nodes that are adjacent in the node sequence of a search.

Lemma 81: Suppose R is an execution of the SearchRight, TryFlagNode, or SearchToLevel_SL routine by process P. Let the node-sequence of R be N(R) = $\{n_1, n_2, \dots, n_s\}$ and the time trace of R be T(R) = $\{T_1, \dots, T_s\}$. Then for $1 \le i \le s$, one of the following transitions from n_{i-1} to n_i took place at time T_i:

1. SearchRight assigned a new value n_i to curr_node in line 4.
2. SearchRight assigned a new value n_i to curr_node in line 9.
3. A new SearchRight routine was invoked with curr_node = n_i.
4. TryFlagNode assigned a new value n_i to prev_node in line 10.
5. TryFlagNode assigned a new value n_i to prev_node in line 11.
6. A new TryFlagNode routine was invoked with prev_node = n_i.

Furthermore, the following is true for for $i > 1$:

- If a transition of type 1, 5, or 6 took place, then $n_i = n_{i-1}$.
- If a transition of type 3 took place, and R is an execution of SearchRight or TryFlagNode, then $n_i = n_{i-1}$. If R is an execution of SearchToLevel_SL, then either $n_i = n_{i-1}$, or $n_i = n_{i-1}$.down.
- If a type 2 transition took place, then n_i.key > n_{i-1}.key.
- If a type 4 transition took place, then n_i.key < n_{i-1}.key.

Proof:

A value can be assigned to a curr_node pointer in SearchRight only when SearchRight executes line 7 or 10, or when it is invoked. Similarly, a value can be assigned to a prev_node pointer in TryFlagNode only when TryFlagNode executes line 10 or 11, or when it is invoked.

If a transition of type 1 took place, then at T_i curr_node was assigned a value n_i = new_curr_node, which was returned by TryFlagNode in line 4. Immediately before TryFlagNode returned, its prev_node pointer was pointing to the node n_i it returned. Thus $n_i = n_{i-1}$. Similarly, if a transition of type 5 took place at T_i, the value assigned to prev_node was the same value curr_node had before SearchRight returned, and so $n_i = n_{i-1}$ in this case as well.

If a transition of type 6 took place at T_i, then, since $i > 1$, TryFlagNode was called from SearchRight. SearchRight can call TryFlagNode only from line 7, and it initializes prev_node to the value of its curr_node pointer, so $n_i = n_{i-1}$.

Suppose a transition of type 3 took place. If R is an execution of TryFlagNode or SearchRight, then SearchRight can be called only from line 11 of TryFlagNode, and it initializes curr_node to the value of its prev_node pointer, so $n_i = n_{i-1}$. If R is an execution of SearchToLevel_SL, SearchRight can also be called from line 3 or line 6 of SearchToLevel_SL, in which case n_{i-1} is the node returned by the previous SearchRight called by SearchToLevel_SL, and $n_i = n_{i-1}$.down.

If a transition of type 2 took place, then at T_i SearchRight assigned a value n_i = next_node to curr_node in line 9. Let T be the last time before T_i when next_node was assigned a value. At that time SearchFrom was executing line 1, or 7, or 10. In any case, $T > T_{i-1}$, because every time SearchRight updates its curr_node pointer (by initializing it after the invocation or explicitly in line 4, or 9), it then updates its next_node pointer before it makes the next update to curr_node. Since $T > T_{i-1}$, curr_node = n_{i-1} at T, and therefore next_node = n_{i-1}.right at T. Thus, by Inv 1, next_node.key > n_{i-1}.key at T, and since at T_i n_i = next_node, and next_node does not change between T and T_i, n_i.key > n_{i-1}.key.

If a transition of type 4 took place, then at T_i TryFlagNode assigned a value n_i = prev_node.back_link to prev_node in line 11. Immediately before this line was executed, prev_node = n_{i-1}. Thus by Inv 7, n_i.key < n_{i-1}.key. ■

Making a transition of type two is the only way a search can move from a node with a smaller key to a node with a bigger key. In the following lemma we prove a useful fact about the transitions of this type.

Lemma 82: Suppose R is an execution of the SearchRight or the TryFlagNode routine by process P. Let the node sequence N(R) be $\{n_1, n_2, \ldots, n_s\}$ and the time trace T(R) be $\{T_1, \ldots, T_s\}$. If for some i, such that $2 \le i \le s$, the transition from n_{i-1} to n_i was of type two (i.e. SearchRight assigned value n_i to curr_node in line 9 at T_i), then there is a time T, such that $T_{i-1} < T < T_i$ when node n_i has been inserted into the list, is not superfluous, and n_{i-1}.right = n_i.

Proof:

Since transition from n_{i-1} to n_i is of type 2, SearchRight assigned a new value n_i = next_node to curr_node in line 9 at T_i. Let T′ be the last time before T_i when this

SearchRight executed line 3, and let T be the last moment before T_i when next_node variable was assigned a value. As we showed in the previous proposition, every time SearchRight updates its curr_node pointer, it then updates its next_node pointer before it makes the next update to curr_node, so $T_{i-1} < T$. Also, SearchRight does not update its next_node pointer between T' and T_i, so $T < T'$. Therefore, $T_{i-1} < T < T' < T_i$.

At time T SearchRight was executing line 1, or 7, or 10, and thus next_node = curr_node.right at T. This value of next_node is assigned to curr_node at T_i, so next_node = n_i at T, and therefore n_i is inserted at T. Also, since $T_{i-1} < T$, curr_node = n_{i-1} at T. So, n_i = next_node = curr_node.right = n_{i-1}.right at T.

At time T' the loop condition in line 3 was false, so at time T', next_node was pointing at a node that was not superfluous. Since next_node does not change its value between T' and T_i, when next_node = n_i, node n_i is not superfluous at T. Since $T < T'$, node n_i is not superfluous at T either. ■

The following proposition is an analogue of Proposition 19 for skip lists.

Proposition 83 (SearchRight and TryFlagNode property): If Inv 8 holds up to time U, then the following is true. Suppose R is an execution of the SearchRight or the TryFlagNode routine by process P. Let the node sequence N(R) be $\{n_1, n_2, \ldots, n_s\}$ and the time trace T(R) be $\{T_1, \ldots, T_s\}$. Suppose some node n_j from the node sequence is not a marked node at some time $T_j' \le T_j$. Then for any node n_i, where $j \le i \le s$, if $T_i \le U$, then either n_i.key $\le n_j$.key and n_i was not marked at T_j', or n_i was not a superfluous node at T_j'.

Proof:

The claim clearly holds when $i = j$. Suppose the claim holds for nodes $n_j, \ldots, n_{k-1}$. We will prove that if $T_k \le U$, then the claim holds for node n_k as well.

Let us examine moment T_k, when the curr_node or prev_node pointer gets set to n_k. By Lemma 81, one of the following transitions took place:

1. At T_k SearchRight assigned a new value n_k to curr_node in line 4.
2. At T_k SearchRight assigned a new value n_k to curr_node in line 9.
3. At T_k a new SearchRight routine was invoked, with curr_node = n_k.
4. At T_k TryFlagNode assigned a new value n_k to prev_node in line 10.
5. At T_k TryFlagNode assigned a new value n_k to prev_node in line 11.
6. At T_k a new TryFlagNode routine was invoked, with prev_node = n_k.

If case 1, 3, 5, or 6 applies, then by Lemma 81, $n_k = n_{k-1}$, and the property holds for n_k. So, the only interesting cases are case 2 and case 4.

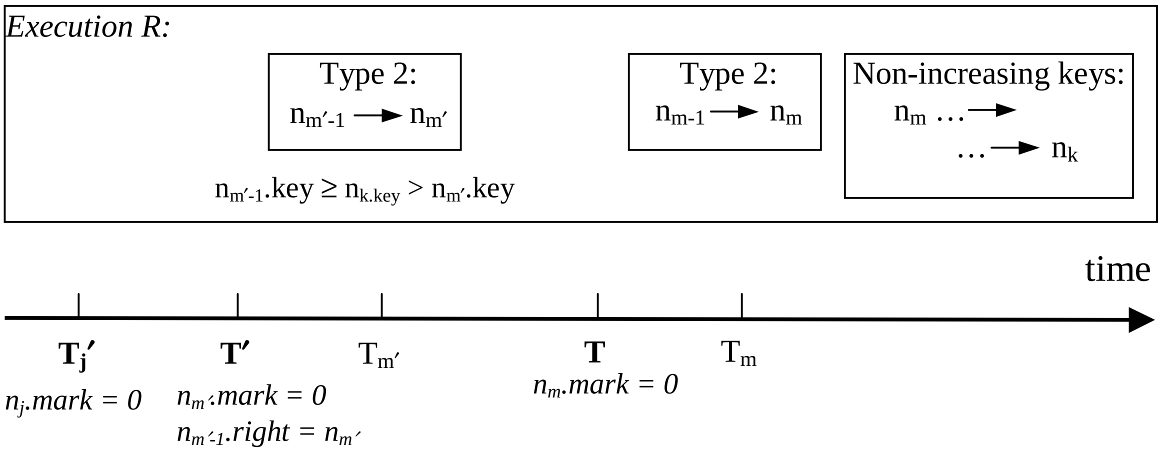
Suppose the 2nd case applies. Then by Lemma 82, there exists moment T, such that $T_{k-1} < T < T_k$, and n_k is not superfluous at T. Since $T > T_{k-1} \ge T_j \ge T_j'$, the claim holds for n_k in this case.

Suppose the 4th case applies. Then at T_k TryFlagNode assigned a value n_k = prev_node.back_link to prev_node in line 11. Immediately before T_k, prev_node = n_{k-1}, prev_node.back_link = n_k, and therefore n_{k-1}.back_link = n_k. We know that the claim holds for n_{k-1}, so at time T_j' n_{k-1} either was not a superfluous node, or was not a marked node. By Proposition 78, nodes that are not superfluous are not marked, so n_{k-1} was not a marked node at T_j'. Since at T_k n_{k-1}.back_link = n_k, it follows from Proposition 80 that n_k could not get marked before n_{k-1}, and therefore n_k was not marked at T_j'. Thus, if n_k.key $\le$ n_j.key, the desired property holds for n_k.

Suppose n_k.key > n_j.key. Let us prove that in this case either n_k is not yet inserted at T_j', or one of the nodes $n_j, \ldots, n_{k-1}$ is equal to n_k. By Lemma 82, n_{k-1}.key > n_k.key, so n_{k-1}.key > n_j.key. Therefore, there has to be an m such that $j < m \le k-1$ and n_m.key > n_{m-1}.key. Let us take the biggest such m. Then the keys of the nodes in the sequence $n_m, \ldots, n_k$ are non-increasing. So, by Lemma 81, for any h such that $m < h \le k$, either $n_h = n_{h-1}$, or a transition of type 4 from n_{h-1} to n_h took place at T_h, and prev_node was assigned value n_h in line 10 of TryFlagNode, in which case $n_h = n_{h-1}$.back_link. So, node n_k was reached from node n_m by following back_link pointers. It then follows from Proposition 80, that n_k was not marked when n_m got marked.

Since n_m.key > n_{m-1}.key, the transition from n_{m-1} to n_m was of type 2, and thus, by Lemma 82, there exists a time T such that $T_{m-1} < T < T_m$, and n_m is not superfluous at T. Then by Proposition 78 n_m is not marked at T (see Figure 39). Since n_k was not marked when n_m got marked, n_k is not marked at T either. Now notice that since n_m.key > n_k.key > n_j.key, there must exist some m', such that $j < m' \le m$ and $n_{m'}$.key ≥ n_k.key > $n_{m'-1}$.key.

Let us show that either $n_{m'} = n_k$, or n_k has not been inserted into the list at T_j'. Since $n_{m'}$.key > $n_{m'-1}$.key, the transition from $n_{m'-1}$ to $n_{m'}$ was of type 2, and thus, by Lemma 82, there exists time T′, when node $n_{m'}$ is inserted into the list, not superfluous, and $n_{m'-1}$.right = $n_{m'}$. Then, by Proposition 78, $n_{m'}$ is a regular node at T′ (see Figure 39). If $m' = m$, then $T = T'$. Otherwise $m' < m$ and then $T' < T_{m'} \le T_{m-1} < T$. So, $T' \le T$, and since n_k is not marked at T, n_k is not marked at T′ either. If n_k is not yet inserted into the list at time T′, then it was not inserted at $T_j' \le T_j \le T_{m'-1} < T'$, and then the proposition holds. Suppose n_k is inserted at T′. Then, since it is not marked, it is a regular node at T′.



Execution R:
Type 2:
$n_{m'-1} \longrightarrow n_{m'}$
Type 2:
$n_{m-1} \longrightarrow n_m$
Non-increasing keys:
$n_m \ldots \longrightarrow$
$\ldots \longrightarrow n_k$
$n_{m'-1}$.key ≥ $n_{k.key}$ > $n_{m'}$.key
time
T_j'
T'
$T_{m'}$
T
T_m
n_j.mark = 0
$n_{m'}$.mark = 0
$n_{m'-1}$.right = $n_{m'}$
n_m.mark = 0

Figure 39: Execution R.

Case 1: Suppose $n_{m'-1}$ is a regular or a logically deleted node at T′. Then by Inv 1 and 3, since $n_{m'}$.key ≥ n_k.key > $n_{m'-1}$.key, either $n_{m'} = n_k$, or n_k is between $n_{m'-1}$ and $n_{m'}$ in the linked list formed by regular and logically deleted nodes at T′. Since $n_{m'-1}$.right = $n_{m'}$ at T′, it must be the case that $n_{m'} = n_k$. The claim holds for $n_{m'}$, because $m' < k$, so it also holds for n_k.

Case 2: Suppose $n_{m'-1}$ is a physically deleted node at T′. Let us examine the moment of time $T'' < T'$, immediately before $n_{m'-1}$ got marked. Note that the claim holds for $n_{m'-1}$, so $n_{m'-1}$ is not a marked node at T_j', and thus $T'' > T_j'$. Since marked pointers do not change, $n_{m'-1}$.right = $n_{m'}$ at T″. If n_k is not yet inserted into the list at T″, then the proposition holds, because $T'' > T_j'$. If n_k was inserted before T″, then, since it is not marked at T′ > T″, it is not marked at T″ either, and thus n_k is a regular node at T″. Since $n_{m'}$ and $n_{m'-1}$ are also regular nodes at T″, by the same logic as in case 1, $n_{m'} = n_k$, and the claim holds for n_k. ■

The following proposition states the postconditions of the SearchRight routine. It is an extension of Proposition 53.

Proposition 84 (SearchRight postconditions): Suppose Inv 8 holds up to time U. Suppose an execution of SearchRight(k, n) completes before time U, and n.key ≤ k. Let (n1, n2) be the pair of nodes which this SearchRight returns. Then the following statements are true:

- n1.key ≤ k < n2.key.
- There exists a time T1 during the execution of SearchRight when n1.right = n2.
- For any time T before or when SearchRight is invoked, if at that time n is not a marked node, then
  1. There exists a point of time T2 between T and the moment SearchRight returns, when n1.right = n2 and n1.mark = 0.
  2. If in addition n1.key > n.key, then n1 is not a superfluous node at T2.

Proof:

In Proposition 53 we proved that n1.key ≤ k < n2.key, and that there exists a moment of time T1 during the execution of SearchRight, such that at T1 n1.right = n2.

Suppose n is not marked at time T before or when SearchRight is invoked. Let us show that there exists a point of time T2 between T and the moment when SearchRight returns, when n1.right = n2 and n1.mark = 0. If n1 is not marked at moment T1, then we are done with the proof. Suppose n1 got marked at some time T′ < T1. Since curr_node = n when SearchRight is invoked and curr_node = n1 just before SearchRight returns, it follows from Proposition 78 and 83 that n1 was not marked at T. Therefore, T < T′ < T1. Since successor fields of marked nodes do not change, and we know that at time T1 n1.right = n2, we can conclude that at time T′ n1.right = n2 as well, so just before T′ n1.mark = 0 and n1.right = n2. This is a valid moment for T2. If in addition n1.key > n.key, then n1 is not superfluous at T2 by Proposition 83. ■

In the following proposition we prove some weak postconditions for the SearchToLevel_SL routine. It is an extension of Proposition 54. After we prove Inv 8 and 9 we will prove stronger postconditions for the SearchToLevel_SL routine.

Proposition 85 (SearchToLevel_SL weak postconditions): If Inv 8 holds up to time U, then the following is true. Suppose an execution STL of the SearchToLevel_SL(k, v) completes before time U, and 1 ≤ v < maxLevel. Let (n1, n2) be the pair of nodes STL returns. Then the following statements are true:

- n1.key ≤ k < n2.key.
- Nodes n1 and n2 belong to level v.

- There exists a time T1 during the execution STL, such that at that time n1.right = n2.
- If n1.key = k, then there exists a point of time T2 during STL when n1 is not a superfluous node.

Proof:

In Proposition 54 we proved that n1.key $\leq$ k < n2.key, and that there exists a moment of time T1 during the execution STL when n1.right = n2. Also, by Proposition 73 nodes n1 and n2 belong to level v.

Suppose n1.key = k. Let W be the tower that n1 belongs to. Let SR be the first of the SearchRight routines called by STL that has a node that belongs to tower W in its node sequence. This node cannot be the first node in SR's sequence, because if it was, then the previously called SearchRight routine would have a node that belongs to W in its node sequence as well. Therefore there had to be a transition from $n_{i-1} \notin W$ to $n_i \in W$ made by SR. Since $n_{i-1} \neq n_i$, this must be a transition of type 2 or 4. Note that it cannot be a transition of type 4, because then by Lemma 81 n_{i-1}.key > n_i.key = k, but all the nodes in the node sequence have keys less than k, because searches do not enter the nodes with keys greater than the key they are looking for. Therefore, it had to be a transition of type 2, and thus by Lemma 82, there was a moment T2 between T_{i-1} and T_i (i.e. during the execution STL), when n_i was not a superfluous node. Since n_i and n1 both belong to the same tower W, n1 was not superfluous at that moment either. ■

4.3.5 The Insert_Node and Insert_SL routines

In this subsection we will prove two important properties of the Insert_Node and Insert_SL routines, which will help us prove Inv 8 and 9. We start by proving a lemma which shows that the nodes that are traversed during an insertion of key k have keys that are smaller or equal to k.

Lemma 86: Suppose ISL is an execution of the Insert_SL(k, e) routine. If prev_node = n at some point of time during the execution of Insert_SL(k, e) or during the execution of one of the InsertNode routines called during ISL, then n.key $\leq$ k. Furthermore, if n.key = k, then n was returned by a search for key k.

Proof:

Let $n_1, \ldots, n_s$ be the sequence values of the prev_node pointers during the execution of Insert_SL(k, e) and all the InsertNode routines called by Insert_SL(k, e). We prove by induction that n_i.key $\leq$ k.

Base case: The value n_1 was returned by the SearchToLevel_SL(k, 1) routine in line 1 of Insert_SL(k, e). It follows from Proposition 54 that n_1.key $\leq$ k, so the base case holds.

Induction step: Suppose n_i.key $\leq$ k. Let us prove that n_{i+1}.key $\leq$ k. There can be several cases:

1. The value n_{i+1} was returned by the SearchToLevel_SL(k, curr_v) routine in line 24 of Insert_SL(k, e).
2. The value n_{i+1} was returned by the InsertNode(newNode, prev_node, next_node) routine in line 11 of Insert_SL(k, e).
3. The value n_{i+1} is the initial value of one of the InsertNode routines called from Insert_SL(k, e).

4. The value n_{i+1} is the result of the back_link traversal in line 16 of an InsertNode called by Insert_SL(k, e).
5. The value n_{i+1} was returned by the SearchRight(newNode.key, prev_node) routine in line 17 of an InsertNode called by Insert_SL(k, e).

If case 1 applies, then n_{i+1}.key ≤ k by Proposition 54. If case 2 applies, then n_{i+1} was the predecessor of newNode just after it was inserted. Since newNode.key = k, n_{i+1}.key < k by Inv 1. If case 3 applies, then $n_{i+1} = n_i$, because when InsertNode is invoked, its prev_node pointer is initialized to the value of the prev_node pointer of the Insert_SL routine. By the induction hypothesis the lemma holds for n_i. If case 4 applies, then n_{i+1} = n_i.back_link, so n_{i+1}.key < n_i.key ≤ k by Inv 7. If case 5 applies, then n_{i+1} was returned by the SearchRight(n_i, k) routine, so by induction hypothesis and Proposition 53, n_{i+1}.key ≤ k. ■

The following proposition shows that each InsertNode routine called from Insert_SL, successfully inserts a node into the skip list, unless the root node of the tower that Insert_SL is constructing gets deleted.

Proposition 87 (InsertNode postconditions): If Inv 8 holds up to time U, then the following is true. Suppose the Insert_SL(k, e) routine is being executed, and it calls the InsertNode routine in line 11, when curr_v > 1. If this InsertNode completes before U, then either it successfully inserts newNode into the list and returns newNode, or newRNode gets physically deleted before InsertNode returns, and InsertNode returns DUPLICATE_KEY.

Proof:

Since curr_v > 1, Insert_SL performed at least one complete iteration of the loop in lines 10-25, and therefore, by Lemma 75, newRNode was inserted into the list, so at the moment when InsertNode returns, newRNode is a regular, a logically deleted, or a physically deleted node. If the InsertNode routine called in line 11 inserts newNode into the list, then it returns newNode in line 11. Suppose this InsertNode fails to insert newNode into the list. Then by Lemma 74, InsertNode returns DUPLICATE_KEY. Let us examine the flow of the execution of this InsertNode routine.

Let m be the value of prev_node immediately before InsertNode returned. From Lemma 72 and Proposition 85 it follows that m belongs to level curr_v. Again, from Lemma 72 and Proposition 85, it follows that this invocation of the Insert_SL routine did not insert any nodes on level curr_v or above, so m was created and inserted by a different invocation of Insert_SL.

We will show that at some time T after newRNode was inserted m is not a superfluous node. Since InsertNode returned result = DUPLICATE_KEY, it exited in line 2 or 19, so the condition in line 1 or 18 was true, and m.key = k. If InsertNode exited in line 2, then the value of prev_node = m was returned by SearchToLevel_SL in line 24 of the Insert_SL routine which called InsertNode. Then by Proposition 85 m is not superfluous at some moment during the execution of SearchToLevel_SL, and this is a valid moment for T.

Suppose InsertNode exited in line 19. Let m′ be the value of prev_node before it executed the last SearchRight in line 17. Let us show that m′.key < k and there exists a time T′ during the execution of InsertNode, when m′ was not marked.

By Lemma 86 m′.key ≤ k, and the equality is possible only if m′ was returned by one of the searches. If m′.key = k and m′ was returned by a search, then InsertNode would have exited earlier (in line 2 or 19), without performing another search. So, m′.key < k.

Let us examine the time when InsertNode executed line 5 for the last time. If at that time condition in line 5 was true, then this a valid moment for T′, because at that time prev_node = m′ is flagged, and thus not marked. If the condition in line 5 was false, InsertNode executed line 15 at least once before it performed a search. In this case the moment when it executed line 15 for the last time is a valid moment for T′.

So, m′ is not marked at some time T′ during the execution of InsertNode, and m′.key < k = m.key, and since the SearchRight that returned m started from node m′, by Proposition 84 there was a moment after T′, when m was not superfluous, and that is a valid moment for T.

So, there was a time T after newRNode was inserted, when m was not a superfluous node. Let rm be the root of m’s tower (m is not preinserted, so it belongs to some tower). Since m was created by a different invocation of Insert_SL, by Proposition 70 rm ≠ newRNode. Since m is not superfluous at T, rm is not marked at T. By Lemma 76 rm is not a preinserted, so it is a regular node at T. By Inv 8 all root nodes belong to level 1, so both newRNode and rm belong to level 1. Node rm is a regular node at T, and newRNode is regular, logically deleted, or physically deleted at T. Nodes newRNode and m have the same key k and belong to the same level, so it follows from Inv 1 and 2 that newRNode is physically deleted at T. ■

Proposition 88: Suppose Inv 8 holds up to time U. If the moment immediately after Insert_SL executes line 22 is before U, then at that moment lastNode belongs to level curr_v – 1.

Proof:

Let us prove this by induction on the number of times a given invocation of Insert_SL has executed line 22. When it executes this line for the first time, curr_v = 2 and newNode = newRNode. Since Insert_SL did not exit in line 14, by Lemma 75 newNode = newRNode was successfully inserted at level 1 by the InsertNode routine called in line 11.

Let us prove the induction step. Suppose after Insert_SL executed line 22 v times, the claim holds. Let us prove that it holds after it executes line 22 for the (v + 1)-th time. Let us examine the flow of execution of Insert_SL from the time it executed line 22 for the v-th time until it executed line 22 for the (v + 1)-th time.

After Insert_SL executed line 22 for the v-th time, curr_v = v + 1. Thus, by Proposition 73, the nodes returned by the search in line 24 belong to level v + 1. If the InsertNode routine called in line 11 did not insert newNode, then, by Proposition 87, newRNode is marked after InsertNode returns, and thus Insert_SL exits in line 18 before it reaches line 22. We assumed that it reaches line 22, so InsertNode successfully inserted newNode. By Lemma 72 newNode gets inserted at level v + 1. Therefore, after line 22 is executed for the (v + 1)-th time, lastNode belongs to level v + 1. ■

4.3.6 Proving invariants for the towers of the skip list

In this subsection we will prove Inv 8 and Inv 9, using the claims proved in the previous subsection. We start by proving Inv 8.

Theorem 89: Inv 8 always holds.

Proof:

We know that the nodes do not change levels, root nodes cannot become non-root nodes, and down pointers never change, thus, to prove that Inv 8 always holds, it is sufficient to prove that the C&S C in line 9 of InsertNode preserves Inv 8. (Obviously, the invariant holds when the list is empty.)

Suppose Inv 8 holds until the C&S C in line 9 of InsertNode is executed, let us prove that it holds after it is executed as well.

The C&S is *result = c&s(prev_node.succ, (next_node, 0, 0), (newNode, 0, 0)).*

The InsertNode routine is called only from Insert_SL. Suppose it was called from the v-th iteration of the loop 10-25. From Lemma 72 and Proposition 73 it follows that prev_node and next_node belong to level v, and thus newNode gets inserted at level v. If v = 1, then newNode = newRNode is a root node and the invariant is preserved. If v > 1, then newNode was created in line 23, so it is not a root node. The down pointer of newNode was set to lastNode, and by Proposition 88 lastNode belongs to level v – 1, so the invariant is preserved in this case as well. ■

Theorem 90: Inv 9 always holds.

Proof:

Invariant 9 consists of four claims. We are going to prove those claims one by one.

1. Node n is not preinserted, so it belongs to some level v of the skip list. If v = 1, then by Inv 8, v is a root node, and thus it belongs to its own tower. If v > 1, then by Inv 8, if we start from node n and follow the down pointers, we will get to a node of level 1 after v – 1 traversals. By Inv 8 this is a root node, and by Def 11, n belongs to its tower.
2. Root nodes are created only in line 4 of the Insert_SL routine and have their down pointer initialized to null. By the first claim we already proved, all other nodes belong to some tower, so it follows from Def 11 that their down pointers cannot be null.
3. Node n is not preinserted, so it belongs to some level v of the skip list. Then by Inv 8 node n.down belongs to level v – 1, and therefore it is not preinserted either. Then it follows from Def 11 that nodes n and n.down belong to the same tower.
4. Suppose such a node m exists. Since m.down = n.down, and m is not preinserted, m belongs to the same tower as n. From Inv 8 it also follows that m and n belong to the same level. But nodes of the same tower are created by the same invocation of Insert_SL, which inserts no more than one node on each level – a contradiction. ■

4.3.7 A stronger SearchToLevel_SL postcondition

In this subsection we will prove a stronger postcondition for the SearchToLevel_SL routine than the one given in Proposition 85. We will show that the first of the two nodes returned by an execution STL of the SearchToLevel_SL routine is unmarked at some time during STL. We will use this postcondition to prove the correctness of our algorithms. We start by proving a few technical lemmas.

The following lemma shows that when a process is building a tower, it deletes the preceding superfluous nodes under certain conditions. Specifically, if a process inserts a new node n on level v, then it deletes any superfluous node m such that at any time during the insertion of n, all the nodes on level v with keys between m.key and n.key are superfluous. The reason for this is that unless m gets physically deleted, the node sequence of one of the searches performed before n gets inserted must contain some node m′ such that m′ is superfluous throughout the execution of that search and m′'s tower is between m's tower and n's tower. However, that is not possible, because a search cannot enter a superfluous tower by following a right pointer (by Lemma 82), and since there are no non-superfluous nodes of the appropriate level between m′'s tower and n's tower, a search won't be able to enter m′'s tower by following a back_link either.

Lemma 91 (deleting superfluous nodes): Suppose nodes rn1 and rn2 are root nodes, such that k1 = rn1.key < rn2.key = k2. Suppose the heights of the towers of rn1 and rn2 are at least v, and v > 1. Let T1 be the time when the node on the (v – 1)-th level of rn2's tower got inserted, and T2 be the time when the node on the v-th level of rn2's tower got inserted. Suppose node n1 belongs to level v of rn1's tower at time T1. If rn1 is a marked node at T1, and in the interval between T1 and T2, all the nodes on level v with keys strictly between k1 and k2 are superfluous, then n1 will be a physically deleted node at time T2.

Proof:

Since rn1 is a marked node at T1, n1 is superfluous in the interval of time between T1 and T2.

Let n2 be the node on the v-th level of rn2's tower. Let us examine the predecessor m of node n2 immediately after n2 gets inserted at time T2. This must be a regular node with a key less than k2. Suppose node n1 is not physically deleted at time T2. We show that this leads us to a contradiction.

Since n1 is not physically deleted at T2, then either m = n1, or m.key > n1.key = k1. So, if m ≠ n1, then the key of m is strictly between k1 and k2, and therefore by the hypothesis of the proposition, m is superfluous between T1 and T2. If m = n1, then m is superfluous between T1 and T2 as well, because rn1 is a marked node at T1.

So in any case, k1 ≤ m.key < k2, and m is superfluous between T1 and T2. Since m is n2's predecessor at time T2 when n2 got inserted, m was returned by one of the search routines executed by the process doing the insertion of rn2's tower between T1 and T2 (SearchToLevel_SL in line 24 of Insert_SL or SearchRight in line 17 of InsertNode). Let us examine the first moment between T1 and T2 when that search made a transition into some node m′ of the tower m belongs to. By Lemma 81, this transition must be of type 2 or 4. From Lemma 82 it follows that it could not be a transition of type 2, because m (and thus, m′) was superfluous at T1, before the search started. Let us show that it could not be a transition of type 4 either.

Suppose it was a transition of type 4 (following a back_link). Let us examine the transitions before it, and let us take the last transition of type 2 (the search had to make at least one transition of type 2, because it starts from a node of the head tower with key –∞, and it can enter a node with a bigger key from a node with a smaller key only by making a transition of type 2). Suppose that was a transition into a node m″. Node m″ is on level v or higher (because that search did not examine the nodes lower than level v). By Inv 8 and 9 there exists a node m‴ of the same tower as m″, which belongs to level v. By Lemma 82 node m″ (and hence node m‴) was not superfluous at some moment between T1 and T2.

Node m′ was reached from node m″ by following back_links and down pointers (since there were no transitions of type 2), and thus m‴.key = m″.key > m′.key = m.key ≥ k1. Also, since the search was for key k2, and it was either a SearchToLevel_SL, or a SearchRight that, by lemma 86, started from a node with a key less or equal to k, it follows from Propositions 53 and 54, that m‴.key = m″.key ≤ k2. The key of m‴.key cannot be k2, because then the search, wouldn't have leaved the tower of m″ and m″ once it entered it. So, m‴.key is strictly less than k2. Therefore, node m‴ was not superfluous at some time between T1 and T2, belongs to level v, and has a key strictly between k1 and k2, but this contradicts the proposition hypothesis. ■

Lemma 92 (Inserting into a superfluous tower): After a tower becomes superfluous, at most one more node can be inserted into it.

Proof:

By Proposition 70, only one invocation of Insert_SL can insert nodes into a particular tower. After Insert_SL inserts a node into a superfluous tower, it will then exit in line 18 without inserting any more nodes. ■

In the previous lemma we showed that at most one node can get inserted into a superfluous tower. The following lemma shows that there are no back_links pointing to such nodes.

Lemma 93: If node n gets inserted after the tower it belongs to becomes superfluous, there can never be a node that has a back_link pointing to n.

Proof:

First notice that in order to set a back_link to n, a HelpFlagged routine with the first argument n must be called, which can happen only if there was a SearchRight or TryFlagNode routine execution which had n in its node sequence. So, if no SearchRight or TryFlagNode routine execution has n in its node sequence, no node will have a back_link set to n.

Suppose some processes perform SearchRight or TryFlagNode executions that have n in their node sequences. Let us take the SearchRight or TryFlagNode routine execution R that first made a transition into n. Let T be the time when that transition was made.

From Lemma 92 it follows that there is never a node with a down pointer pointing to n. So, R could not enter n by following a down pointer. Notice that R could not enter n by following a back_link either, because if there is a back_link set to n, then there was a SearchRight or TryFlagNode routine execution that made a transition into n earlier, and we assumed R is the first such execution. So, R entered n by following a right pointer. Then by Lemma 82, there was a moment when n was already inserted, and not superfluous, but n was inserted into a superfluous tower – a contradiction. ■

In Proposition 80 we showed that back_links cannot be set to marked nodes. If our algorithms did not allow back_links to be set to superfluous nodes either, it would be easy to show that all the nodes in the node sequence of the execution STL of the SearchToLevel_SL routine are unmarked at some point of time during STL. Then the desired SearchToLevel_SL postcondition (the first of the two nodes returned by STL is unmarked at some time during STL) would follow. However, in our data structure back_links can be set to superfluous nodes. Yet, we can still prove the desired postcondition. Note that when the SearchRight routine called by STL performed at least one transition of type 2 or 4 on its level, then the first of the two nodes it returns is unmarked at some time during its execution. (This follows from Lemma 83 and the fact that if TryFlagNode starts to traverse a back_link chain (lines 9-10), it traverses the chain until it reaches an unmarked node at the end.)

When we prove the SearchToLevel_SL postconditions, we will show that if STL enters a node n2 that was marked before STL started, then at some point of time before, STL traversed a back_link from node n to node m, which was set after m became superfluous. The following lemma will help us prove that the right pointer of n2 is pointing to a node with a key less or equal to n.key, and therefore STL makes a transition of type 2 or 4, leaving node n2 (thus the node STL eventually returns is not marked at some point of time during STL).

Lemma 94: Suppose nodes n and m belong to level v ≥ 2, n.back_link = m and m is superfluous. Let T1 be the time when m became superfluous and let T2 be the time when node n.down was inserted. Then either T1 ≥ T2, or for any time T such that T1 ≤ T ≤ T2, there exists a node q on level v – 1 such that m.key < q.key < n.key and q is inserted and not superfluous at T.

Proof:

Suppose T1 < T2 and there exists a time T such that T1 ≤ T ≤ T2, and at T there is no node q that satisfies the properties described in the lemma.

Let n″ be the first node that is inserted into the list after time T such that

- m.key ≤ n″.key ≤ n.key,
- n″ belongs to level v, and
- n″ is not superfluous when it gets inserted.

If there is no such node, then let n″ = null. Otherwise, let T″ be the time when n″ gets inserted. Let T′ be the time, when node n′ = n″.down got inserted. Note that T″ > T and T″ > T′. Let T3 be the time when node n gets inserted. We will prove that node m becomes physically deleted before time T3, which will lead us to a contradiction.

Case 1: Suppose T″ > T3 or n″ = null. Then at any time between T2 and T3 all the nodes on level v with keys strictly between m.key and n.key are superfluous, because there are no such nodes at time T ≤ T2, and no new ones get inserted at level v until T3. The root node of node m gets marked at time T1 ≤ T2, so by Lemma 91, m gets physically deleted before time T3.

Case 2: Suppose T″ ≤ T3. Node n″ is not superfluous at time T″ when it gets inserted, so node n′ is inserted and not superfluous from time T′ and until time T″. Node n′ belongs to level v – 1, so time T cannot be in the interval between T′ and T″. So, since T″ > T, T′ > T as well. Then at any time between T′ and T″ all the nodes on level v with keys strictly between m.key and n.key are superfluous, because there are no such nodes at time T < T′, and no new ones get inserted at level v until T″. The root node of node m gets marked at time T1 ≤ T < T′, so by Lemma 91, m gets physically deleted before time T″. Since T″ ≤ T3, m gets physically deleted before T3.

So, m gets physically deleted before T3. Note that node n is unmarked at time T3 when it gets inserted, so by Proposition 80, n's back_link cannot point to m – a contradiction. ■

Proposition 95 (the first node returned by SearchRight in SearchToLevel_SL is not marked): Let STL be an execution of the SearchToLevel_SL(k, v) routine, and suppose 1 ≤ v < maxLevel. Let (n1, n2) be the pair of nodes returned by an execution of one of the SearchRight routines called directly by STL. Then there exists a point of time T during the execution of STL, when n1 is not a marked node.

Proof:

Let k1 = n1.key, k2 = n2.key, and let v1 be the level of n1 and n2. Suppose the proposition does not hold. Then n1 was a marked node at time T′ when execution STL(k, v) was started. Let us examine the node sequence of STL. Let SR be the execution of the SearchRight routine we are interested in, and let n be the first node in the node sequence of SR. Note that if there are any other nodes in the node sequence of SR, then it means that SR performed a transition of type 2 or 4 from n. If it was a transition of type 2 (traversing a right pointer), then from Lemma 82 and Proposition 83 it follows that n1 was not marked, at some point during the execution of SR – a contradiction. If it was a transition of type 4 (a back_link traversal), then the process traversed a chain of back_links (lines 9-10 in TryFlagNode) until it ended up in an unmarked node, and again, by Proposition 83, n1 was not marked at some point during the execution of SR – a contradiction. So, SR does not make transitions of type 2 or 4 from n, and thus the first node it returns when it finishes is the same node it starts from, i.e. n = n1.

Node n = n1 is marked at T′ when STL is started. Let T1 < T′ be the time when n1 got marked. By Proposition 84, at some time during SR (after T′) n2 = n1.right, so at T1 n1.right = n2 as well. Therefore, it follows from Inv 1 and 2 that, at T1, there were no unmarked nodes at level v1 with keys strictly between k1 and k2 (see Figure 40).

Let m1 be the first node in the node sequence of STL that belongs to the same tower as n1. Since SR starts at n1, the SearchRight routine called by STL right before SR, had a node of the same tower as n1 as the last node in its node sequence. That SearchFrom was executed on level v1 + 1, and therefore node m1 belongs to level v2 ≥ v1 + 1.

Let us examine the last transition of type 2 in the node sequence of STL made before node m1 (there must be at least one transition of type 2 made by STL, because if there was none, n1 is a node of the head tower, and thus cannot be marked). That transition was made on level v2 or higher. Let m2 be the node the transition was made into. By Lemma 82, m2 is in the list and not superfluous (and thus, by Proposition 78, not marked) at some time T2 > T′. Also, note that m2.key ≥ m1.key = n1.key = k1, because no right pointer traversals are made after time T2. The equality is only possible if m2 is a node of the same tower as m1 and n1, in which case n1 is not superfluous (and thus not marked) at T2, because m2 is not superfluous at T2 – a contradiction. So, the inequality is strict: m2.key > k1.

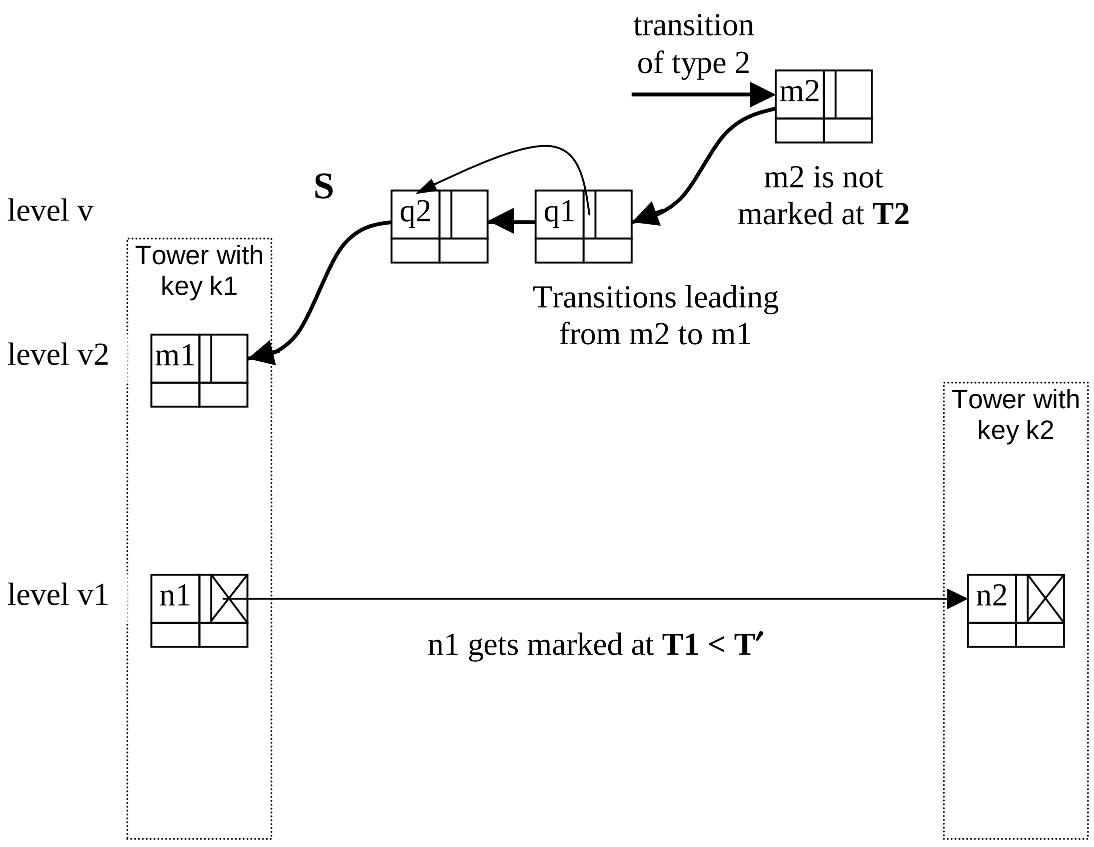


Figure 40: Execution STL.

Since node m2 is the last node before m1 in the node sequence of STL that was entered by following a right pointer, the keys of the nodes in the sequence between m2 and m1 are in non-increasing order. Let S be the part of the node sequence of STL starting from node m2 and ending in node m1.

Let S_{sp} = {node q | k1 ≤ q.key ≤ k2 and q is superfluous at T1}. Let S_{ni} = {node q | k1 ≤ q.key ≤ k2 and q.down was not yet inserted at T1}. We will show that there exist two nodes, q1 and q2 in S such that $q1 \in S_{ni}$, $q2 \in S_{sp}$, and q1.back_link = q2, and we will apply Lemma 94 to these nodes to prove the proposition.

Let us first show that all the nodes in S either belong to S_{ni} or to S_{sp}. Suppose some node q belongs to S. Since all the keys in S are in non-increasing order, q.key ≤ m2.key ≤ k < k2. Also, q.key ≥ k1, so k1 ≤ q.key ≤ k2. Node q is on level v2 ≥ v1 + 1 or higher, therefore, by Inv 9, there exists a node q′ of the same tower as q, that belongs to level v1. As we proved earlier, at time T1 there were no unmarked nodes at level v1 with keys between k1 and k2, therefore q′ is either marked, or not yet inserted at T1. If q′ is not yet inserted at T1, then q and q.down are not yet inserted at T1, i.e. $q \in S_{ni}$. If q′ is marked at T1, then q′ is superfluous at T1, and thus q is superfluous at T1, i.e. $q \in S_{sp}$. So, all the nodes in S either belong to S_{ni} or to S_{sp}.

Note that since k1 < m2.key < k2 and m2 is not superfluous at time T2 > T1, $m2 \in S_{ni}$. On the other hand, m1.key = k1, and m1 was marked at T1, so $m1 \in S_{sp}$. Node m2 is the first node in sequence S, and node m1 is the last in S, therefore there has to be two nodes q1 and q2 in S such that q1 belongs to S_{ni}, q2 belongs to S_{sp}, and STL makes a transition from q1 to q2. Note that STL enters all the nodes in S except m2 either by following back_links or by following down pointers. STL could not make a transition from q1 into q2 by following a down pointer, since $q1 \in S_{ni}$, $q2 \in S_{sp}$, and thus q1 and q2 must belong to different towers. Therefore, STL made a transition from q1 into q2 by following a back_link, so q1.back_link = q2.

Let v be the level of the nodes q1 and q2. Since $q1, q2 \in S$, $v \geq v2$. Since $q1 \in S_{ni}$ and $q2 \in S_{sp}$, node q1.down was not inserted at T1 and node q2 was superfluous at T1. It follows from Lemma 94 that at time T1 there was a node q on level $v - 1$, such that q2.key < q < q1.key and q was inserted and not superfluous at T1. Since $v \geq v2 \geq v1 + 1$, there exists a node q′ at level v1 that belongs to the same tower as q, and q′ is inserted and not superfluous at T1. So, by Proposition 78, q′ is a regular node at T1. Recall that the keys of the nodes that belong to S_{ni} and S_{sp} are between k1 and k2. Since q2.key < q′.key < q1.key, k1 < q′.key < k2. But we proved earlier that there can be no regular nodes with keys strictly between k1 and k2 at level v1 at time T1 – a contradiction. ■

We now use Proposition 95 to prove SearchToLevel_SL postconditions.

Proposition 96 (SearchToLevel_SL property and postconditions): Let STL be an execution of the SearchToLevel_SL(k, v) routine, and suppose $1 \leq v <$ maxLevel. Let (n1, n2) be the pair of nodes returned by STL or by an execution of one of the SearchRight routines called directly from STL. Then the following statements are true:

- n1.key ≤ k < n2.key
- If (n1, n2) are returned by STL, then n1 and n2 belong to level v.
- There exists a point of time T during the execution STL such that at that time n1.right = n2 and n1 is not marked.
- If n1.key = k, then there exists a time T′ during STL when n1 is not superfluous.

Proof:

Suppose (n1, n2) were returned by an execution of one of the SearchRight routines called directly from STL. Let SR be that execution. By Proposition 84 applied to SR, n1.key ≤ k < n2.key, and there exists some time T1 during SR, when n1.right = n2. On the other hand, by Proposition 95, there exists time T2 during the STL, when n1 is not marked. If n1 is not marked at T1, then T1 is a valid moment for T. Otherwise let T3 be the moment immediately before n1 got marked. Since n1 is marked at T1, but not at T2, T2 < T3 < T1, and thus T3 belongs to the interval of time when the execution STL is performed. Marked pointers do not change, and n1.right = n2 at T1. Therefore, n1.right = n2 at T3.

If (n1, n2) are returned by STL, then they were returned by the last SearchRight routine called directly from STL, so the first and the third claims of the proposition hold for (n1, n2). Additionally, by Proposition 85, n1 and n2 belong to level v.

The fourth claim was proved in Proposition 85. ■

4.3.8 Linearization points and correctness

Our skip list data structure allows three types of dictionary operations: searches, insertions, and deletions, which are executed by invoking the major routines Search_SL, Insert_SL, and Delete_SL respectively.

In this subsection we assign linearization points to each operation and prove that they are implemented correctly. We say that the set of the elements currently stored in the dictionary is the set of the elements of the regular nodes on level one. We show that this set is modified only at the linearization points of the insertions and deletions according to the specifications of these operations. We also prove that if an operation completes, it returns a correct result, according to its linearization point.

We start by assigning linearization points to the searches.

Theorem 97 (Search_SL correctness): If an execution of the Search_SL(k) routine returns NO_SUCH_KEY, then we can choose a linearization point, at which there was no regular root node with key k in the data structure. If an execution of the Search_SL(k) routine returns a pointer to a node, then this is a root node, the key of this node is k, and for this execution we can choose a linearization point, at which this node was a regular node.

Proof:

Let STL be the execution of the SearchToLevel_SL(k, 1) routine, performed in the first line of the Search_SL routine. Let (n1, n2) be the pair of nodes returned by STL. By Proposition 96, n1 and n2 belong to level 1, n1.key ≤ k < n2.key, and at some point of time T during the STL n1.right = n2 and n1 is unmarked. We linearize the Search_SL routine at T. At that moment n1 is a regular node, because it is not marked (and it is obviously not a preinserted node). Also, n1 and n2 are root nodes, because they belong to level 1.

Case 1: Suppose the Search_SL execution returns in line 3. Then it returns a pointer to n1 and n1.key = k (line 2). As we showed, n1 is a regular root node at the linearization point of STL.

Case 2: Suppose the Search_SL execution returns NO_SUCH_KEY in line 5. Then n1.key ≠ k. At the linearization point n1 and n2 are two consecutive nodes in the list of regular and logically deleted nodes of the first level, and n1.key < k < n2.key, which means that there is no regular root node with key k in the data structure, by Inv 1, 2. ■

Note that if an execution of the Search_SL(k) routine is non-terminated, we do not linearize it.

To prove the correctness of the Insert_SL routine we first prove the following lemma about the nodes that are traversed by the first InsertNode routine executed by Insert_SL.

Lemma 98: Suppose ISL is some execution of the Insert_SL routine. Let IN be the first execution of the InsertNode routine called by ISL (when curr_v = 1). Then at any time during IN, IN's prev_node pointer points to a node that was regular at some moment during ISL.

Proof:

Let us prove this by induction on the number of times prev_node changes its value. When IN is invoked, prev_node is pointing to a node returned by the SearchToLevel_SL routine in line 1 of ISL. So, by Proposition 96, there was a time during the execution of ISL, when this node was not marked.

The prev_node pointer can change its value in lines 16 and 17 of InsertNode. If it changes its value in line 16, then the claim of the lemma holds for the new value of prev_node by Proposition 80 and the induction hypothesis. If it changes its value in line 17, then the claim holds for the new value of prev_node by Proposition 84 and induction hypothesis. ■

In the following theorem we assign linearization points to executions of the Insert_SL routine. We will linearize successful insertions at the moment when they insert the root node. We will linearize unsuccessful insertions at the moment when there is a regular root node in the list that contains the key they are trying to insert. We also define a mapping that will help us to prove that an element can be inserted into the dictionary only by executing an appropriate insertion.

Theorem 99: (Insert_SL correctness): If an execution ISL of the Insert_SL(k, e) routine returns DUPLICATE_KEY (indicating an unsuccessful insertion), then for this execution we can choose a linearization point at which there was a regular root node with key k in the data structure. If ISL returns a pointer to a node (indicating a successful insertion), then this is a root node with key k, and for this execution we can choose a linearization point, at which this node gets inserted.

Furthermore, there exists a mapping ψ of all regular, logically deleted and physically deleted root nodes of the data structure to successful Insert_SL executions and non-terminated Insert_SL executions such that

1. ψ is injective.
2. For any successful execution ISL of the Insert_SL routine, ISL = ψ(rn) if and only if ISL returns rn.
3. At any time T, if a root node rn is regular, logically deleted, or physically deleted at T, then ψ(rn) is linearized when rn got inserted.

Proof:

We choose the linearization point depending on how the Insert_SL routine runs.

Case 1: Suppose ISL returns DUPLICATE_KEY in line 3. Let (n1, n2) be the pair of nodes SearchToLevel_SL returns in line 1. By Proposition 96 and Inv 8, n1 is a root node, and there exists a moment during the execution of SearchToLevel_SL in line 1, when n1 is unmarked. We linearize ISL at that point of time. At that moment, n1 is a regular root node, and n1.key = k (line 2).

Case 2: Suppose ISL returns DUPLICATE_KEY in line 14. This means that curr_v = 1, and the InsertNode routine called in line 11, returned DUPLICATE_KEY. InsertNode can return DUPLICATE_KEY only in line 2 or in line 19. Let n be the value of prev_node before line 2 or 19 was executed. Then n.key = newNode.key = k (line 1 or 18 respectively). By Lemma 98 there was a time T during the execution ISL, when n was in the list and not marked. We linearize ISL at that time. At that moment n is a regular root node (it belongs to level 1 by Lemma 72), and n.key = k.

Case 3: Suppose ISL returns newRNode in line 18 or 21. Since ISL did not exit in line 14, by Lemma 75 it inserted newRNode into the list. That happened when the first InsertNode routine called by ISL successfully executed the C&S in line 9. We linearize ISL at that moment of time.

Case 4: Suppose the execution ISL is non-terminated. If the first InsertNode routine called by ISL successfully performed a C&S in line 9, then we linearize ISL at the moment when that C&S was performed. This is non-ambiguous, because once InsertNode performs a successful C&S in line 9, it is poised to return in line 11, so for each execution of Insert_SL, there is at most one successful C&S performed by the first InsertNode routine Insert_SL calls. If ISL has not called any InsertNode routines, or if the first InsertNode routine called by ISL has not performed a successful C&S in line 9, we do not linearize ISL.

Finally, let us show that there exists a mapping ψ of the insertions of regular, logically deleted, and physically deleted root nodes to successful and non-terminated Insert_SL executions, which has the properties described in the proposition. In order for a new root node rn to be added to the data structure, a preinserted node must be created by the execution of the Insert_SL routine, and then the C&S in line 9 of the first InsertNode routine, called by that execution must be successfully performed. Let us define ψ so that it maps each root node rn to the Insert_SL execution that successfully performed that C&S. If Insert_SL execution ψ(rn) successfully performs the C&S in line 9 of the first InsertNode routine it calls, then it cannot return in lines 3 or 14, and thus it is poised to return rn in line 18 or 21, so at any point of time this Insert_SL execution is either successful or non-terminated.

For each execution of Insert_SL, there is at most one successful C&S performed by the first InsertNode routine Insert_SL calls, so ψ is injective (property 1 proved).

If ISL is a successful execution of the Insert_SL routine, and ISL returns rn, then, as we have shown above, the first InsertNode routine called by ISL successfully performed the C&S in line 9 that inserted rn, and thus ISL = ψ(rn). On the other hand, if ISL is successful and ISL = ψ(rn), then by the definition of ψ, the first InsertNode routine called by ISL performed a successful C&S that inserted rn, and therefore the node it returns is rn (property 2 proved).

Finally, if at time T, node n is regular, logically deleted, or physically deleted, then ψ(rn) is defined, and by property 2, execution ISL = ψ(rn) returns rn. From the way we assign linearization points to the executions of the Insert_SL routines, it follows that ISL is linearized at the moment when the first InsertNode routine it called successfully performed a C&S in line 9, inserting rn (property 3 proved). ■

Before we prove that the Delete_SL routine correctly implements deletion, we prove two lemmas about the TryFlagNode and HelpFlagged routines.

Lemma 100 (TryFlagNode invariant): Suppose TryFlagNode(n, m) is called and n.key < k. Then at any time during its execution, prev_node.key < m.key and target_node = m.

Proof:

The value of target_node never changes, so target_node = m throughout the execution. Notice that TryFlagNode can be called only from line 4 of the SearchRight routine, or line 1 of the DeleteNode routine. In the first case, it follows from Inv 1 that n.key < m.key, because there was a time during the execution of SearchRight when n = curr_node.right = next_node = m. Suppose TryFlagNode was called from the DeleteNode routine. DeleteNode can be called from line 4 of the Delete_SL routine, or from line 17 of the Insert_SL routine. In the first case, by Proposition 96, there was a time during the execution of SearchToHeight_SL in line 1 of the Delete_SL routine, when n.right = m, so n.key < m.key. In the second case, since the last InsertNode executed by Insert_SL, returned newNode (line 16), that InsertNode inserted newNode into the list, by Lemma 74. At the moment immediately after InsertNode performed a successful C&S in line 9, n.right = m, and therefore n.key < m.key.

So, at the beginning of the execution of TryFlagNode, prev_node.key < m.key. The value of prev_node can be changed only in line 10 or 11. In the first case it follows from Inv 7 that the key of prev_node decreases, and in the second case prev_node.key ≤ target.node.key – ε < m.key, by Proposition 84, so prev_node.key is always strictly less than m.key. ■

Lemma 101: If the HelpFlagged(n, m) routine is called, then by the time it completes, node m is physically deleted.

Proof:

HelpFlagged calls HelpMarked, which tries to execute a C&S C to physically delete m. By Lemma 60, at some point before HelpFlagged was called, n.succ = (m, 0, 1). If this is still true when C is executed, then C succeeds and m gets physically deleted. Otherwise n.succ was changed, and any change to n.succ physically deletes m, by Lemma 65. ■

In the following theorem we assign linearization points to executions of the Delete_SL routine. We will linearize each unsuccessful deletion at some moment when no regular root node has the key it searches for. We will linearize each successful deletion at the moment when a (regular) root node with the key it searches for gets marked. Our algorithms are designed in such a way that a deletion of some root node rn reports success only if it performs the first critical step of rn's deletion (flagging rn's predecessor). A successful deletion does not necessarily perform the second step (marking) itself, another process may do it. We also define a mapping that will help us prove that an element can be deleted from the dictionary only by executing an appropriate deletion. This is the last theorem we need to prove the correctness of our implementation.

Theorem 102 (Delete_SL correctness): If an execution DSL of the Delete_SL(k) routine returns NO_SUCH_KEY (indicating an unsuccessful deletion), then for this execution we can choose a linearization point, at which there was no regular root node with key k in the data structure. If DSL returns a pointer to a node (indicating a successful deletion), then this is a root node with key k, and for this execution we can choose a linearization point, at which this node became marked.

Furthermore, at any point of time, there exists a mapping σ of all marked root nodes to the successful and non-terminated Delete_SL executions such that

1. σ is injective
2. For any successful execution DSL of the Delete_SL routine, DSL = σ(rn) if and only if the node DSL returns is rn.

3. At any time T, if a root node rn is marked, then σ(rn) is linearized at the moment when rn got marked.

Proof:

We choose the linearization point for DSL depending on how it runs.

Case 1: Suppose DSL returns in line 3. This is a case when DSL could not find a node with key k in the list. Let n1 and n2 be the values of prev_node and del_node SearchToLevel_SL returns in line 1. By Proposition 96, n1 and n2 belong to level 1, n1.key ≤ k – ε < n2.key (i.e. n1.key < k ≤ n2.key), and at some time T during the execution of SearchToLevel_SL, n1.right = n2 and n1.mark = 0. We linearize DSL at T. Since n2.key ≠ k (line 2), it follows that n1.key < k < n2.key. At the linearization point node n1 is unmarked, thus n1 is a regular root node of the list, and since n1.right = n2, n2 is either a regular or a logically deleted root node. Since the union of regular and logically deleted root nodes is a sorted linked list, and since at the linearization point the value k is between the keys of the two consecutive nodes of this list, we can conclude that there is no regular root node with key k in our data structure at the linearization point of the deletion.

Case 2: Suppose DSL returns NO_SUCH_KEY in line 6. This is a case when DSL found a node with the required key, but still failed to delete it, and we must linearize DSL at a moment when there was no regular root node with key k in the data structure. Since DSL returned in line 6, the DeleteNode routine, which it called in line 4, returned NO_SUCH_NODE. Let n1 and n2 be the values of prev_node and del_node returned by the search in line 1 of DSL. By Proposition 96, n1 and n2 belong to level 1, and at some time T′ during the execution of SearchToLevel_SL, n1.right = n2 and n1 is unmarked. Since DSL did not exit in line 3, n2.key = k. At time T′ n1 is a regular root node, so n2 = n1.right is either a regular or a logically deleted root node. We will prove that there was a point of time T during execution DSL when n2 is a logically deleted node. We will linearize DSL at that moment, and then we will prove that this is a valid linearization, i.e. that at this moment there is no regular root node with key k in our data structure.

If n2 was a logically deleted node at T′, then T′ is a valid moment for T. Suppose n2 is a regular node at T′. Since DeleteNode returned NO_SUCH_NODE, it exited in line 5. This means that the TryFlagNode routine called from line 1 of DeleteNode returned result = false. Let us examine that invocation of TryFlagNode. Since it returned result = false, it could return only in line 3, 8, or 13. Also notice that it has second parameter target_node = n2.

Case 2(a): TryFlagNode returns in line 13. This is a case when n2 gets deleted before TryFlagNode can flag its predecessor. The proof is the same as the proof of Case 2(a) in Theorem 22, except that we use Proposition 84 and Lemma 100 instead of Proposition 20 and Lemma 21. Briefly, let (n3, n4) be the two nodes returned by the last SearchRight, which TryFlagNode called in line 11. At some point of time T″ during the execution of SearchRight, n3 was not marked and n3.right = n4. Since n3.key < n2.key ≤ n4.key and n4 ≠ n2, node n2 is physically deleted at T″. Since at T′ n2 was a regular node, there was some moment T during the execution DSL, when n2 was a logically deleted node.

Case 2(b): TryFlagNode returns in line 3 or 8. This is the case when some other process Q flags the predecessor of n2. Q is also executing a Delete_SL(k) routine and will report success (or die before finishing its deletion). DSL ensures that all critical steps of the deletion are performed and reports failure. The proof of this case goes the same way as the proof of Subcase 2 in Theorem 22, except that we use Proposition 69 instead of Proposition 17. Briefly, before DeleteNode returns, it calls HelpFlagged, which, by Lemma 101, does not exit until n2 is marked (and physically deleted). Since at T′ n2 was a regular node, there was a moment T during the execution DSL when n2 was a logically deleted node.

So in either of the above cases, there is a time T during DSL when n2 is a logically deleted node, and n2.key = k, so by Inv 1 and 2 there are no regular root nodes with key k in the list, and thus we can choose T as the linearization point for DSL.

Case 3: Suppose DSL returns del_node in line 8. This is the case of a successful deletion, and it must be linearized at the moment when del_node got marked. Since DSL returned in line 8, the DeleteNode routine called in line 4 did not return NO_SUCH_NODE, so the TryFlagNode routine called from line 1 of DeleteNode returned result = true. Let n1 be the node that TryFlagNode returned, and let n2 = del_node. Again, the proof of correctness in this case is the same as the proof in Case 3 of Theorem 22, except that we use Proposition 69 instead of Proposition 17: briefly, since TryFlagNode returned result = true, it successfully flagged the predecessor of n2. By Proposition 69, n2 was not marked at that moment, but before DeleteNode finishes the execution of HelpFlagged in line 3, by Lemma 101, n2 = del_node will get marked. We linearize DSL at the moment when it gets marked.

Case 4: Suppose the execution DSL is non-terminated. Suppose DSL has already called the DeleteNode routine in line 4, which in turn called TryFlagNode in line 1, and that TryFlagNode routine performed a successful C&S in line 4. Let n2 be the value of target_node immediately before that C&S. If n2 is currently marked, then we linearize DSL at time T when n2 got marked. By Proposition 69, marking of a node is performed after flagging of the node’s predecessor, so T is after DSL started. Therefore, T is a valid linearization point. In any other case we do not yet linearize DSL.

Finally, let us show that there exists a mapping σ of all marked root nodes to successful and non-terminated executions of the Delete_SL routines, which has the properties described in the proposition. By Proposition 69, before a node gets marked, its predecessor gets flagged. Nodes can only be flagged in the TryFlagNode routine, which can only be called from the DeleteNode routine and the SearchRight routine.

Note that if SearchRight calls TryFlagNode(n1, n2) and n2 is a root node, then n2.mark = n2.tower_root.mark = 1 (line 3), and thus by Proposition 69 the predecessor of n2 is already flagged, so TryFlagNode(n1, n2) will exit in line 8 without flagging any root nodes. The DeleteNode routine can be called only by Delete_SL in line 4 and by Insert_SL in line 17. Notice that Insert_SL cannot call DeleteNode on a root node, since line 16 requires that newNode $\neq$ newRNode. Thus, of all the routines, only Delete_SL can flag a predecessor of a root node. Let us define σ so that it maps a marked root node rn to the execution of the Delete_SL routine that flagged rn’s predecessor, i.e the Delete_SL execution that performed the first critical step of rn’s deletion. If a Delete_SL execution σ(rn) flags rn’s predecessor, then it is poised to return node rn in line 8, so at any point of time this execution is either successful or non-terminated.

Each Delete_SL invocation flag at most one node on the first level, so σ is injective (property 1 proved).

If DSL is a successful execution of the Delete_SL routine, and DSL returns a root node rn, then, as we have showed in Case 3 above, DSL has flagged rn's predecessor, and thus DSL = σ(rn). On the other hand, if DSL is successful and DSL = σ(n), then by the definition of σ, the TryFlagNode routine called by the DeleteNode that was called by DSL has flagged rn's predecessor by performing a successful C&S in line 4. Since that C&S has flagged rn's predecessor, TryFlagNode's local variable target_node is equal to rn immediately before the C&S. By Lemma 100, target_node never changes its value, so when TryFlagNode was called by DeleteNode, target_node = rn, and thus the DeleteNode's local variable del_node = rn. Then DSL's local variable del_node = rn as well. Therefore the node returned by DSL in line 8 is rn (property 2 proved).

Finally, if at time T node rn is marked, then σ(rn) is defined, and by property 2, execution DSL = σ(rn) returns rn. From the way we assign linearization points to the executions of the Delete_SL routines, it follows that DSL is linearized at the moment when rn got marked (property 3 proved). ■

It follows from Theorems 97, 99, and 102, that our data structure correctly implements the three dictionary operations. The set of the elements currently stored in the dictionary is the set of the elements of the regular root nodes of our data structure. An element of a root node rn is added to the dictionary at the moment when the insertion ψ(rn) is linearized, and is removed from the dictionary at the moment when the deletion σ(rn) is linearized.

4.3.9 The heights of the towers

In this subsection we discuss the heights of the towers in our skip list data structure. We show that unless an insertion of the tower is interrupted by a deletion, the chance that the tower reaches a given height is the same as in a sequential skip list.

Def 16: We say that the tower's *insertion is finished*, when the Insert_SL routine that inserted its root node returns.

Def 17: A tower is *under construction*, if its insertion has inserted a root node, but has not finished yet.

Def 18: Let W be a tower in the skip list. The *potential height* of W is the value of the tH variable in the Insert_SL routine invocation that inserted W's root node after it executed line 7 for the last time.

Def 19: Let W be a tower in the skip list, whose insertion is finished. We say that W is *full*, if its height is equal to its potential height. We say that the construction of W was *interrupted* if these heights are different.

In the following proposition we explain how a tower's construction can be interrupted.

Proposition 103: If the construction of tower W was interrupted, then its height H is less than its potential height H′, and W's root node got marked before W's insertion was finished.

Proof:

Let ISL be the invocation of the Insert_SL routine that created W. Let us first show that H cannot be greater than H′. Insert_SL adds nodes to the tower it is constructing by calling InsertNode in line 11. If H > H′, then ISL executed line 11 at least H′ + 1 times, so it executed line 19 at least H′ times. But after ISL executed line 19 for the H′-th time, curr_v = H′ + 1, so ISL was poised to exit in line 21 before it could execute InsertNode again – a contradiction. So, H ≤ H′. Since W's construction was interrupted, H ≠ H′, and thus H < H′.

Now let us show that if W's construction was interrupted, then W's root node got marked before ISL completed. We proved that H < H′, so ISL did not return in line 21. Also, ISL could not return in line 14, because it did insert W's root node, and thus, by Lemma 74, InsertNode did not return DUPLICATE_KEY in the first iteration of the loop in lines 10-25 (when curr_v = 1). Therefore, ISL returned in line 18, which means that W's root node got marked during ISL (line 15). ■

Note that it follows from the proposition above that the height of a tower is never greater than its potential height.

It is easy to see that the probability distribution of the potential heights of the full towers is the same in our data structure as in the sequential implementation. The probability is taken over all the possible executions, given the particular sequence of operations each process performs and the schedule (which specifies the order in which processes take steps). We allow an *adversary* to choose the schedule and sequence of the operations for each process. In the next few paragraphs we discuss the possible types of adversary.

The adversary is the agent that chooses which operations the processes of the system should perform, and in which order the processes of the system take steps. When the algorithms executed by the processes in the system are deterministic, the state of the system at any given time can be predicted, given the operations executed by the processes and the order in which the processes took steps, and therefore there is no reason for the adversary to make decisions based on the state of the system during run-time. Conversely, if the algorithms are non-deterministic, the state of the system cannot be predicted in such a way. For non-deterministic algorithms there are several types of adversaries: the *oblivious* adversary, who produces a sequence of operations and a schedule that is independent of the behaviour of the random variables in the system, and *adaptive* adversaries of various strengths, who have some kind of knowledge about the values of the random variables.

The algorithms for the lock-free linked list, presented in the previous chapter were deterministic. Therefore, the type of the adversary was irrelevant. The algorithms presented in this chapter, however, are non-deterministic, because the heights of the towers in the skip list depend upon a randomized routine FlipCoin().

Consider a strong type of adaptive adversary, who knows the heights of the towers. Such adversary will be able to delete all the tall towers in the list, hampering the performance of the data structure. For example, if the adversary deletes all the towers of height two or greater, the performance of the skip list will be no better than the performance of the linked list.

Here, we examine a weaker type of adaptive adversary. Our adversary specifies the *program* for each process, and the schedule for the processes. The program of a process can call the major routines of our data structure (Search_SL, Insert_SL, Delete_SL), and the Wait operation, and make decisions based on the outcome of these operations. (The Wait operation just makes a process pass its turn to take a step). The schedule is the ordering in which the processes take their steps. Thus, the adversary has no direct knowledge, nor direct control over the randomized parameters of the data structure (the heights of the towers). (However, as we will see later, the adversary can still influence the probability distribution of the heights of the towers indirectly by changing the process schedule.) This model of the adversary is quite realistic for many applications.

The following invariants set the conditions on the heights of the towers in our data structure.

Inv 10: Let p be the probability with which FlipCoin() routine returns "head". For each possible choice of programs and schedules by the adversary, each tower has potential height H with probability $(1 – p)p^{H – 1}$ for $1 \le H < \text{maxLevel} – 1$, and height maxLevel – 1 with probability $p^{\text{maxLevel} – 1}$.

Inv 11: At any time, the number of towers that are not full and also contain at least one node that is not physically deleted, is not greater than the current contention (i.e. the number of operations that are currently running).

These invariants give us a good insight into the structure of our lock-free skip list. If Inv 10 and 11 hold, then potential heights of the towers are distributed probabilistically as in the sequential skip list described by Pugh [Pug90], and the number of towers, which contain some nodes that are not physically deleted, and also have height different from their potential height is not greater than the current contention C. The proof of the invariants is given below.

Theorem 104: Inv 10 holds.

Proof:

From the definition of the potential height, it follows that a tower has a potential height H, if and only if the there were H independent invocations of the FlipCoin() routine, and either the first H – 1 invocations returned "head", and the last (H-th) invocation returned "tail", or all H invocations returned "head" and H = maxLevel. The probability that the routine returns "head" is p, and the probability that it returns "tail" is 1 – p, so the proposition holds. ■

Lemma 105: Suppose W is a tower with a root node rn, and rn.key = k. If the execution STL of the SearchToLevel(k, 2) routine is started at time T, and rn is physically deleted at T, then by the time STL completes, all the nodes of W that were in the list at T will be physically deleted.

Proof:

Suppose some node n belongs to level v of the tower W at time T. If v = 1, then n = rn, and the lemma holds. Suppose, v > 1. Let us take the execution SR of the SearchRight routine called by STL on level v. Suppose SR returns a pair of nodes (n1, n2). By Proposition 96, $n1.key \le k < n2.key$, and there exists some time T1 during SR, when n1.right = n2 and n1 is not marked. At time T1 n1 is a regular node, so n2 is either a regular or a logically deleted node. By Inv 1 and 2 there are no regular or logically deleted nodes with keys between n1.key and n2.key on level v at T1. Therefore, since n.key = k and n1.key ≤ k < n2.key, and n is inserted into the list at T < T1, either n is a physically deleted node at T1 (in which case the lemma holds), or n = n1.

Suppose n = n1. Let us examine the node sequence of STL, and let us take the first transition that entered the tower W. That transition could not be of type 4, because the keys of all the nodes in STL's node sequence are not greater than k, so it had to be of type 2. Then by Lemma 82, node n was not a superfluous node at some moment during STL, but n's root node rn was a physically deleted node, when STL was started – a contradiction. ■

Lemma 106: Suppose n1 and n2 are nodes on the same level of the skip list, and n1.key < n2.key. Then if the execution DN of the DeleteNode(n1, n2) routine is performed, n2 will be physically deleted by the time DN completes.

Proof:

The proof is somewhat similar to the proof of correctness of the Delete_SL routine.

DN calls TryFlagNode(n1, n2) in line 1. Suppose TryFlagNode returns status = IN. This means that TryFlagNode returned in line 3, 6, or 8, and the predecessor of n2 was flagged. Then DN will call HelpFlagged in line 3. By Lemma 101, that HelpFlagged routine will not exit until n2 will get physically deleted.

Suppose TryFlagNode returns status = DELETED. Then TryFlagNode returned in line 13. Let us examine the SearchRight routine, which TryFlagNode called in line 11 before it returned. Let n1′ be the value of prev_node before SearchRight was executed, and (n3, n4) be the pair of node returned by SearchRight. Since n1.key < n2.key, Lemma 100 applies, and n1′.key < n2.key. Also, n1′ was not marked at time T, when TryFlagNode executed line 9 for the last time. Then by Proposition 84, n3.mark = 0 and n3.right = n4 at some time T′ > T. Also, n3.key < n2.key ≤ n4.key, and since n4 ≠ n2 (line 12), node n2 is physically deleted at T′, and thus it is physically deleted when DN completes. ■

Proposition 107: If a tower W with a root node rn is not full, then either its insertion ψ(rn), or its deletion σ(rn) has not yet completed, or all of its nodes are physically deleted.

Proof:

Suppose a tower W is not full. Then either its insertion is not finished yet, or its construction was interrupted. If W's insertion is not finished, then ψ(rn) has not yet completed and the proposition holds.

Suppose W's construction was interrupted. Then by Proposition 103, W's insertion was finished after its root rn got marked. Let us examine the execution σ(rn) of the Delete_SL routine. If it is not completed yet, the proposition holds. Suppose σ(rn) has completed. Let T be the moment when σ(rn) finishes executing the DeleteNode routine in line 4. Root node rn is physically deleted at T. Since σ(rn) is a successful deletion, DeleteNode returns del_node = rn, and thus σ(rn) calls the SearchToLevel_SL routine in line 7. Let T′ be the time when this SearchToLevel_SL routine completes. By Lemma 105, all the nodes of the tower W that were in the list at time T, will be physically deleted at T′. So, when the deletion σ(rn) completes, all the nodes of W that were in the list at T, are physically deleted. If the insertion ψ(rn) does not insert any other nodes in W after T, then the proposition holds.

By Lemma 92, the execution ψ(rn) of the Insert_SL routine can insert at most one node in W after T. Suppose it inserts node n. Insert_SL inserts nodes by calling the InsertNode routine. After the InsertNode routine that inserted n completes, it returns result = n = newNode and prev_node pointing to the node, which was n's predecessor, when n was inserted. So, prev_node.key < newNode.key. Then, before ψ(rn) returns, it calls the DeleteNode routine in line 17. Since prev_node.key < newNode.key, by Lemma 106, this DeleteNode routine will physically delete node n = newNode. Thus, when the insertion ψ(rn) completes, all the nodes of tower W are physically deleted, and no nodes will be inserted in W after that. ■

Theorem 108: Invariant 11 always holds.
Proof:
Follows from Proposition 107. ■

So, Inv 10 and 11 hold, and thus the potential heights of the towers are distributed probabilistically as in a sequential skip list, and the number of towers, which contain some nodes that are not physically deleted, and also have heights different from their potential heights is not greater than the current contention C. The first impulse might be to conclude that the probabilistic distribution of the heights of all but C towers in our skip list is same as in the sequential skip list. In that case it would be easy to prove that the searches in our skip list that do not call TryFlagNode, take O(log(n) + C) time on average, where n is the number of the keys in the list. However, for some strategies of the adversary

1. The event that a tower is full at a given time T is not independent from the event that a tower has a certain potential height H.
2. The event that a tower is superfluous at a given time T is not independent from the event that a tower has a certain potential height H.

We show why these two statements are true in the next few paragraphs. Note that if they are true, it follows that the heights of the full towers that are not superfluous are not distributed the same as the heights of the towers in a sequential skip list, and therefore it is not clear whether the average running time achieved is O(log(n) + C).

Let us first show that the first statement is true. Suppose some tower W has a potential height H. Let us take moment T not long after the Insert_SL routine that created the tower was invoked. If H is small, then it is more likely that the insertion would finish by time T, than if H were large. Thus, the event that W is full at T is not independent from the event that W has a certain potential height H. For another example, suppose the deletion of W is started soon after the insertion is started. Then it is more likely for a tower of large height to become interrupted than for a tower of a small height, and thus again, the event that W is full is dependent upon the height of W. Note that for this example we can take any time T after the insertion exits.

In general, the event that W is superfluous at T is also dependent upon the potential height of W. Suppose a deletion of W is performed by the same process P that performs the insertion of W right after it finishes the insertion. Consider a time T shortly after the insertion has begun. It is more likely that the deletion of W has begun at time T if W has small potential height. So the event that W is superfluous at T is dependent on the potential height of W.

Thus, we cannot assume that the probability distribution of the potential heights of the full towers that contain non-physically deleted nodes is the same as the probability distribution of the potential heights of all the towers. The exact difference between these two distributions is something that can be explored in future work. However, the intuition is that the difference should not be large, and the performance of the searches in our lock-free skip list is not hampered considerably because of this difference.

4.4 Lock-Freedom and performance

In this section we will prove that our data structure is lock-free. Then we will describe several ways to improve the performance and space requirements of our data structure by constant factors.

4.4.1 Lock-freedom

We will prove the lock-freedom property of our data structure by proving that after some process takes a finite number of steps, a successful C&S will be performed on the data structure (by this process or by another one). Since each operation requires a constant number of successful C&S's to complete, this implies lock-freedom. We start by proving several technical lemmas.

Lemma 109: Suppose process P is executing a SearchRight routine SR. Then after P performs some finite number of steps, P will complete SR, or P will call a TryFlagNode routine in line 4, or there will be a successful C&S performed by some process.

<u>Proof:</u>

If SR enters the loop 3-7, it calls TryFlagNode. If it does not enter this loop, then each iteration of the loop in lines 2-10 costs O(1), and in each iteration lines 9 and 10 are executed. Note that each time SR executes line 9, it makes a transition of type 2. If no successful C&S's are performed, no new nodes get inserted into the list, and therefore SR can execute line 2 only a finite number of times, if it does not call TryFlagNode. Thus, after a finite number of steps, the number of steps P will complete SR, or call a TryFlagNode routine in line 4, or there will be a successful C&S performed on the data structure. ■

Lemma 110: Suppose process P is executing a TryFlagNode routine TFN. Then after P performs some finite number of steps, P will complete TFN, or P will call a SearchRight routine in line 11, or there will be a successful C&S performed by some process.

<u>Proof:</u>

If no C&S's are performed, no new nodes get inserted into the list, and therefore completing the loop in lines 9-10 takes finite time. All the other lines in TFN are executed at most once during the single iteration of the loop 1-14, and executing any of the lines except line 11, where SearchRight is called, takes O(1) time. Therefore, if no successful C&S's are performed, after a finite number of steps P will either complete TFN, or call a SearchRight routine in line 11. ■

Lemma 111: Suppose process P is executing the HelpFlagged, TryMark, or HelpMarked routine R. Then after P performs some finite number of steps, either P will complete R, or there will be a successful C&S performed by some process.

Proof:

HelpMarked takes O(1) steps to complete. After O(1) steps, HelpFlagged either completes or calls TryMark, and after O(1) steps TryMark either completes one iteration of the loop 1-6, or calls HelpFlagged.

Let us show that if an invocation of TryMark completes two iterations of the loop in lines 1-6 without calling HelpFlagged, then a successful C&S was performed on del_node.succ during the execution of those two iterations. If no C&S's were performed, then after the first iteration completed, del_node was not flagged (line 4) and was not marked (line 6). When the second iteration started, next_node was set to del_node.right in line 2. So unless another process performs a C&S on del_node.succ, the second iteration will perform a successful C&S on del_node.succ when it executes line 3.

So, if there are no successful C&S's, TryMark cannot complete more then one iteration of the loop in lines 1-6 without calling HelpFlagged. At any given time there can be no more flagged nodes than deletions in progress, and thus if no successful C&S's are performed, TryMark can call HelpFlagged, and then recursively itself, only a finite number of times.

Thus, after P performs a finite number of steps, either P will complete R, or there will be a successful C&S performed. ■

Lemma 112: Suppose at time T process P is executing a SearchRight or TryFlagNode routine R. Then after P performs some finite number of steps, either P will complete R, or there will be a successful C&S performed by some process.

Proof:

Case 1: Suppose R is the TryFlagNode routine. By Lemma 110, after R makes a finite number of steps, it will either complete, or start an execution SR of the SearchRight routine in line 11. Note that SR will start from a node that was unmarked (line 9) at some time T1 > T. By Lemma 109, it will take SR a finite number of steps to complete or call another TryFlagNode.

Case 1(a): Suppose SR completes. Let (n1, n2) be the nodes it returns. Since it started from a node that was unmarked at T1 > T, by Proposition 84, there exists some time T1′ after T, when n1.right = n2 and n1 is not marked. Then R enters the next iteration of the loop 1-14. If it then exits in line 2, or successfully executes the C&S in line 4, the proposition holds. If it fails the C&S, then there was a C&S performed on n1.succ after T1′ > T, and the proposition also holds.

Case 1(b): Suppose SR calls another TryFlagNode routine. Let n1 and n2 be the arguments with which it calls that TryFlagNode. By Proposition 83, n1 was not marked at T1 > T. Also, there was some time T1′ > T during the execution SR when n1.right = n2. Since marked successor pointers do not change, there was some time T2 after T and before SR TryFlagNode, when n1.right = n2 and n1 is not marked. Then by the same argument as in the previous case, either there will be a C&S performed on n1.succ (in which case the proposition holds), or TryFlagNode will exit in line 3. Suppose TryFlagNode exits in line 3. Then SR will proceed and call HelpFlagged(n1, n2) in line 6. Then by Lemma 111, after a finite number of steps, either a successful C&S will be performed (in which case the proposition holds), or HelpFlagged will complete. By Lemma 101, HelpFlagged does not return until it marks and physically deletes n2, and n2 was not physically deleted at T2, thus a successful C&S will be performed after T2 > T.

Case 2: Suppose R is the SearchRight routine. By Lemma 109, after R makes a finite number of steps, it will either complete, or call the TryFlagNode routine in line 4. As we just proved, after a finite number of steps, either this TryFlagNode will complete, or there will be a successful C&S on the data structure (in the latter case the proposition holds). Suppose TryFlagNode completes. We will again let SearchRight run, until it calls another TryFlagNode. Suppose this TryFlagNode routine has arguments n1 and n2. Node n2 is not physically deleted at T. If TryFlagNode returns status = IN, then HelpFlagged will be executed, and by Lemma 101 it will not return until n2 is physically deleted, so a successful C&S will be performed after T. If TryFlagNode returns status = DELETED, then n2 already got physically deleted (for a proof see Case 2(a) in Theorem 102). ■

Lemma 113: Suppose at time T process P is executing a SearchToLevel_SL routine STL. Then after a finite number of steps, either P will complete STL, or a successful C&S will be performed by some process.

Proof:

The execution of the first line in STL takes finite time, because the height of the head tower is finite. Since curr_v is finite after FindStart_SL returns, the number of iterations of the loop 2-5 that can be performed is also finite. Thus, it follows from Lemma 112, that after a finite number of steps, either P will complete STL, or a successful C&S will be performed. ■

Proposition 114: Suppose at time T process P is executing a Search_SL routine. Then after a finite number of steps, either P will complete Search_SL, or a successful C&S will be performed by some process.

Proof:

Follows from Lemma 113. ■

Lemma 115: Suppose at time T process P is executing an InsertNode routine IN. Then after P performs a finite number of steps, either it will complete IN, or a successful C&S will be performed by some process.

Proof:

First note that by Lemmas 111 and 112, after a finite number of steps, IN will either perform two full iterations of the loop in lines 3-20 (unless it returns earlier), or a successful C&S will be performed. Also note that if a condition in line 5 is true in one of the iterations, then IN will call HelpFlagged, which will ensure physical deletion of the prev_node's successor (by Lemma 101), so a successful C&S will be performed.

Suppose the condition in line 5 is always false. Let T′ be the moment when line 15 is executed for the last time during the first iteration. At that time n1′ = prev_node is not marked. Let (n1, n2) be the pair of nodes returned by SearchRight in line 17 in the first iteration. By Proposition 84, there is some time T′ > T, when n1.right = n2 and n1 is not marked. Let us now examine the second iteration. By our assumption, the condition in line 5 is always false, so n1 is not flagged when line 5 is executed. Therefore, either IN will successfully perform the C&S in line 9, or another process will perform a successful C&S on n1.succ after T′. ■

Lemma 116: Suppose at time T process P is executing an DeleteNode routine DN. Then after P performs a finite number of steps, either it will complete DN, or at least one successful C&S will be performed by some process.

Proof:

Follows from Lemmas 111 and 112. ■

Proposition 117: Suppose at time T process P is executing an Insert_SL routine ISL. Then after P performs a finite number of steps, either it will complete ISL, or at least one successful C&S will be performed by some process.

Proof:

First note that by Lemmas 111, 112, 113, 115, and 116, after a finite number of steps, ISL will either complete one full iteration of the loop in lines 10-25 (unless it returns earlier), or a successful C&S will be performed. Each full iteration of the loop 10-25, inserts a new node into the skip list by performing a C&S. Thus, the proposition holds. ■

Proposition 118: Suppose at time T process P is executing a Delete_SL routine DSL. Then after P performs a finite number of steps, either it will complete DSL, or at least one successful C&S will be performed by some process.

Proof:

Follows from Lemmas 111, 112, 113, and 116. ■

Theorem 119: Our skip-list data structure is lock-free.

Proof:

It follows from Propositions 114, 117, 118, that after a finite number of steps of some process, either that process completes its operation, or a successful C&S is performed on a node of the data structure. Any successful C&S is part of an insertion or a deletion operation, and for any operation there is a finite number of the C&S's that are part of it (for the search this number is 0, for the insertion of a tower of height H it is H, and for the deletion of a tower of height H it is 3H). The heights of the towers are bounded by the constant maxLevel, so after a finite number of steps of some process, some operation will complete, and thus our data structure is lock-free. ■

4.4.2 Fine tuning performance and space requirements

There are a few ways to improve the performance of operations in our skip list by a constant factor. In our implementation, when an insertion is constructing a tower, it performs a separate search (line 24 in Insert_SL) each time it needs to insert a new node into that tower. However, this is not necessary: if an insertion ISL remembers the node pairs returned by the SearchRight routines called from the first SearchToLevel_SL routine ISL executes (line 1 in Insert_SL), ISL can use these nodes to insert new nodes into the higher levels of the tower it constructs.

Similarly, a deletion does not need to perform a complete search (line 7 in Delete_SL) to delete the rest of the nodes of a tower after it deletes the tower's root node.

Instead, it can remember the node pair at the top of the tower it deletes and start a search from the first node of the pair after it deletes the root node. If this change to the deletions is implemented, the cleanups performed by the insertions when they find out that the root node of the tower they are constructing is marked (lines 16-17 in Insert_SL) should be modified as well. An insertion should delete all of the nodes in the tower, not only the one it just inserted. (This slightly increases the constant factors of the cost of the insertions that have to cleanup, but we likely win more by decreasing the cost of all of the deletions.)

For more space efficiency, we can use the same construction as Pugh [Pug90] (see Subsection 4.1.1) for our skip lists. Recall that Pugh used single node with multiple forward pointers to represent a tower. The reason that we used a different design was only to make the presentation of our algorithms easier. (In particular, to avoid modifying the syntax of our linked list algorithms much.) Employing Pugh’s design for our data structure would be a very easy change, and it would result in a better space efficiency, because there will be no down (or up) pointers and each tower would store only one copy of its key.

5 Memory Management

Developing a correct and efficient memory management scheme is important to make a data structure practical. Developing such a scheme for a lock-free data structure is often quite a challenging task. The difficulty lies in determining how and when memory that was once occupied by parts of the data structure (e.g. nodes of a linked list), can be freed and reused, so that the processes that might still be accessing those parts are able to complete their operations correctly.

With our data structure, we cannot simply free the nodes at the moment when they get physically deleted, because other processes might still be traversing them. In fact, a delayed process might have its local pointer set to a physically deleted node for an arbitrarily long time. We start by describing several existing approaches that could be used to perform memory management for our data structure. All of them have their flaws, which we also discuss. At the end we give an outline of a new memory management scheme we currently investigate.

5.1 Existing approaches

5.1.1 Reference counting

The first approach that can be used is the *reference counting* method. The method was used by Valois for his lock-fee linked list structure in [Val95]. It contained a few mistakes, which were later corrected by Michael and Scott in [MS95]. With this method, each node includes a reference counter that reflects the total number of references to this node from the data structure and from the processes operating on the data structure. A node is freed if and only if its reference counter reaches zero, i.e. there are no references to a node either from the data structure or the processes. When a node is freed, the reference counters of all the nodes it has pointers to are decreased (which can trigger the freeing of some of these nodes). The implementation uses four basic routines to operate on the nodes: SafeRead, Release, New, and Reclaim. SafeRead is used by the processes instead of a simple read. Given a pointer to node n, it increases n's reference counter and returns n. Release is called when a process finishes operating on a node. Called on node n, it decreases n's reference counter and determines if n can be freed and reused safely. If so, then Release calls Reclaim on n, which adds n to the list of nodes available for reuse. New() allocates a node from the list of nodes available for reuse, initializes its fields, and returns a pointer to it.

The method can be applied to any acyclic data structure. While our linked list and skip list data structures are not strictly acyclic because of the back_link pointers, it follows from Propositions 18 and 41 for linked lists and Propositions 79 and 80 for skip lists that there are no cycles among the physically deleted nodes, which is enough for the method to work.

The method has two flaws. First, SafeRead and Release have to be called each time a process updates its pointers, moving through the list. These routines use C&S's to update the reference counters of the nodes, and therefore, calling them often considerably slows down the execution. For example, Harris [Har01] reported a slowdown of his algorithms by a factor of 10 to 15 where this method was used, compared to the case where no memory management was enabled and deleted nodes were not reused. The second flaw of this method is that if a delayed process maintains a local pointer set to some deleted node n for a long time, while other processes continue to perform operations, the system might run out of memory even if the total number of elements in the dictionary is O(1). The reason is that the nodes that can be reached from n always have a reference counter greater than zero, so there can be an arbitrarily many deleted nodes that cannot be reused.

5.1.2 Garbage collectors

Another approach is to use a separate *garbage collector*, e.g. [HM91, Gre99]. A major problem with this approach is the existing garbage collectors that we are aware of, are either not lock-free (and thus their failure can prevent processes from making progress or can cause a system to run out of memory), or require special operating system support. The method also has the same vulnerability as the reference counting method: a long delay of a single process might cause the system to run out of memory.

5.1.3 Deferred freeing

The third approach is to use the *deferred freeing* method. This method was introduced by Harris in [Har01]. Each process maintains a local timestamp that is updated each time a process starts a new operation. Each node of the list is augmented with an additional field, through which it can be linked onto a to-be-freed list, when it is physically deleted. There are two to-be-freed lists per process: the old list and the new list. The old list also contains a timestamp, which is more recent than the time of physical deletion of any node in that old list. When the timestamp of an old list becomes smaller than the timestamps of all the processes in the system, the nodes in that old list are freed, and all the nodes of the respective new list are moved into that old list.

The main vulnerability of the method is that if some process P does not complete any operations, while other processes do, the system might run out of memory, because P’s timestamp will never get updated, and therefore the old lists of the other processes will not get freed, while the contents of the new lists will keep increasing in size.

5.2 The new approach

We also outline another approach for memory management, although we have not yet completed our research on it. The system maintains a shared queue consisting of the nodes that are deleted, but not yet reused. As in the deferred freeing method, each node is augmented with a field, through which it can be linked onto the queue list. Also, the successor fields of the nodes are augmented with tags that are incremented each time a node is reused. Tags allow the processes to determine if a node was reused after they entered it. Incorporating tags into the successor pointers helps to deal with the ABA problem.

After a process logically deletes a node, it enqueues it before the physical deletion. When a process needs to perform an insertion, it starts by checking if the node at the beginning of the queue has been physically deleted or not (for node n, this can be done by checking if n.back_link.right equals n). If it was not, the process completes the deletion. Then it increments the tag of that node (by performing a C&S on the successor pointer), and dequeues it. After a successful dequeue the process increments the tag of the node again (no need to perform a C&S this time) to prevent other processes who might think the node is still in the queue from modifying the node's successor field. The process then uses this node for the insertion.

While processes traverse the linked list data structure, they use tags to check if the nodes they are traversing were reused. This is done as follows. Suppose process P moves from node n1 into node n2 (either by following a right pointer or a back_link). From Propositions 18 and 41 it follows that if n2 is in the queue of nodes waiting to be reused, then n1 is also in that queue and ahead of n2, i.e. if n2 was reused, n1 was reused as well, and therefore, n1's tag changed. So, every time P moves into a new node, it records its tag value, and then verifies the tag value of the node it just left has not changed. If it did, P restarts its operation, otherwise it continues. Since P starts from a head node, which is never deleted or reused, it will always have valid tag values of the nodes it traverses.

We make tags big enough so that the full wraparound of a node's tag while some process maintains a pointer to that node is practically impossible. Back_link pointers are also augmented with tags, and processes use C&S's (not simple writes) to modify them.

If we ensure that the queue always has sufficiently many nodes in it (our estimate is 2p for the linked lists, where p is the number of processes), the number of restarts will be small compared to the number of successful operations, and the amortized cost of any operation on the linked list will still be O(n + c(S)). Below is a sketch of the proof.

Suppose P restarts operation S at time T2 because of the reuse of some node n it was traversing. Let T1 be the time when P last (re)started S before T2. It is easy to show that n was not marked at T1. Let N be the number of nodes in the queue when n was added to it. Since node n was reused, all N nodes ahead of it were reused as well. Then there were at least N – p successful insertions performed between T1 and T2. Let us distribute the cost of the restart of S between these insertions. As a result, each insertion will be billed for O(n + c(S)/(N – p). It is easy to show that one process does not bill the same insertion twice for the restart, and therefore there is no more than p operations billing any insertion for a restart. So, if we make N – p > p, the total cost billed to any insertion by the restarts is O(n + c(S)).

Applying this method to our skip lists is a bit more complicated, because of the down pointers: if n1.down = n2, we cannot guarantee that n1 will be ahead of n2 in the shared queue. One of the solutions to this problem is to use the queue to operate on whole towers instead of operating on the individual nodes. With this solution the tags of all the nodes in a tower are the same. Each tower contains the maximum possible number (maxLevel) of nodes, but the nodes that are too high (higher than the height of a tower during that particular reuse) are not linked into their respective level-lists. This way every tower can be reused as a tower of any legal height.

6 Conclusion

We presented new algorithms implementing a lock-free linked list and a lock-free skip list. We proved their correctness and lock-freedom.

For the linked list algorithms, we proved that the amortized execution time of an operation S is O(n(S) + c(S)), where n(S) is the number of elements in the list when S starts, and c(S) is the contention of S. This is better than the performance of any of the previously published algorithms for a lock-free linked list. Our proof showed that the contention overhead (i.e. the difference between the amortized costs of an operation executing solo, and an operation executing in the presence of a contention) is O(c(S)).

To perform our analysis, we used a complicated scheme of billing part of the cost of each operation to the successful C&S's that were performed by operations that are running concurrently. The technique of analyzing lock-free data structures in this way is unique in the area. We believe that this technique may not only help analyze other complicated lock-free data structure, but may also inspire other researches to create new efficient lock-free data structures, as our design was driven by this technique.

Our lock-free linked list algorithms will most likely perform best compared to Harris's [Har01] and Michael's [Mic02-1] algorithms when executed on fairly large lists in the presence of high contention, so that the C&S's performed by the processes fail often and the cost of recovery from such failures is high.

Our implementation of the lock-free skip lists uses our linked lists algorithms (with some slight modifications) to perform the operations on the individual levels of the skip list. It is not hard to see that the amortized cost of the operations in our lock-free skip list is Ω(log(n(S)) + c(S)) and O(m(S)), where m(S) is total number of operations invoked before S was called. Although the scenarios in which the amortized cost of the operations is significantly different from O(log(n) + c(S)) are fairly exotic, and probably are not likely to occur in a real system, performing a strict analysis is a much harder problem than with the linked list due to the following two reasons:

1. The searches performed by an operation S can traverse the back_links of nodes that were marked by deletions that completed before S was invoked. Therefore, the approach to the analysis which we used for the linked list does not give us good results for the skip list.
2. It is hard to capture the exact difference between the probability distribution of the heights of the towers in our skip list versus the sequential skip list. Our intuition is that this difference does not significantly hamper the performance of the searches, but proving this is a separate problem, which might be addressed in future work.

We believe that our work on lock-free skip-lists, apart from presenting the first lock-free implementation of this data structure that does not use universal constructions, provides good insight into the problems faced when designing such a data structure. One such problem is dealing with a tower whose deletion starts before its insertion is finished. Another problem is making sure that the probability distribution of the heights of the towers in a lock-free skip list is close to what it is in a sequential skip list. We have some ideas on how to improve on the data structure. Perhaps the most promising of them is to make the deletions help the insertions before they delete the tower.

It is worth noting that our design of linked lists and skip lists allows more than standard dictionary operations to be implemented. For example, operation DeleteGE, discussed by Harris [Har01] can be easily implemented for our data structures without any changes to the other operations. The operation DeleteGE(k) deletes and returns the smallest key that is greater than or equal to k. This operation cannot easily be implemented with Harris's design, because when a node gets marked, Harris's algorithm has no control over that node's predecessor. With our algorithms, however, this problem does not exist because of flags. For example, for skip lists this operation could be as shown in Figure 41.

DeleteGE_SL(Key k): RNode

```
(prev_node, del_node) = SearchToLevel_SL(k – $\varepsilon$, 1)
loop
    result = DeleteNode(prev_node, del_node)
    if (result == del_node)
        SearchToLevel_SL(del_node.key, 2)          // Deletes the nodes at higher levels of the tower
        return del_node
    (prev_node, del_node) = SearchRight(k – $\varepsilon$, prev_node)
end loop
```

Figure 41: Possible implementation of.DeleteGE_SL

Future work on our linked list and skip list data structures might include the improvement of the skip lists algorithms and the further development of the memory management scheme outlined in Section 5.2.

References

[AVL62] G. M. Adel'son-Vel'skiy, Ye. M. Landis, "An Algorithm for the Organization of Information," *Deklady Akademii Nauk USSR*, Moscow, Vol. 16, No. 2, 1962, pp. 263-266.

[Gre99] M Greenwald. Non-blocking synchronization and system design. *PhD thesis*, Stanford University, 1999. Technical Report STAN-CS-TR-99-1624.

[Har01] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. *Proceedings of the 15th International Symposium on Distributed Computing*, 2001, pp.300-314.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 1, 1991, pp.124-149.

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, Vol. 15, No.5, 1993, pp. 745-770.

[HM91] Maurice P. Herlihy and J. Eliot B. Moss. Lock-free garbage collection for multiprocessors. *Proceedings of the 3rd annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 229-236, 1991.

[HW90] Maurice Herlihy, Jeanette Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 3, 1990, pp. 463-492.

[IBM83] IBM System/370 Extended architecture, Principles of operation. IBM Publication No. SA22-7085, 1983.

[SMKKB00] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 149-160.

[Mic02-1] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. *Proceedings of the 14th annual ACM Symposium on Parallel Algorithms and Architectures*, 2002, pp.73-82.

[Mic02-2] Maged M. Michael. Safe memory reclamation for Dynamic lock-free objects using atomic reads and writes. *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, 2002.

[MS95] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical Report 599, Computer Science Department, University of Rochester, 1995.

[Pug90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of ACM*, Vol. 33, No. 6, June 1990, pp. 668-676.

[SS03] Ori Shalev and Nir Shavit. Split-ordered lists: lock-free extensible hash tables. *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, 2003, pp. 102-111.

[ST85] Daniel D. Sleator , Robert E. Tarjan, Self-adjusting binary search trees, *Journal of the ACM*, Vol. 32, No.3, July 1985, pp.652-686.

[Tre86] R. Kent Treiber. Systems programming: Coping with Parallelism. Research report RJ 5118, IBM Almaden Research Center, 1986.

[Val95] John D. Valois. Lock-free linked lists using compare-and-swap. *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, 1995, pp. 214-222.