Wait-Free Linked-Lists

Shahar Timnat [*] Anastasia Braginsky [†] Alex Kogan [‡] Erez Petrank [§]

September 23, 2012

Abstract

Wait-freedom is the strongest and most desirable progress guarantee, under which any thread must make progress when given enough CPU steps. Wait-freedom is required for hard real-time, and desirable in many other scenarios. However, because wait-freedom is hard to achieve, we usually settle for the weaker *lock-free* progress guarantee, under which one of the active threads is guaranteed to make progress. With lock-freedom (and unlike wait-freedom), starvation of all threads but one is possible.

The linked-list data structure is fundamental and ubiquitous. Lock-free versions of the linked-list are well known. However, whether it is possible to design a practical wait-free linked-list has remained an open question. In this work we present a practical wait-free linked-list based on the CAS primitive. To improve performance further, we also extend this design using the fast-path-slow-path methodology. The proposed design has been implemented and measurements demonstrate performance competitive with that of Harris's lock-free list, while still providing the desirable wait-free guarantee, required for real-time systems. A full proof of correctness and wait-freedom is also presented.

Keywords: Wait-free, Linked-list, Lock-free, Concurrent Data Structures

1 Introduction

A linked-list is one of the most commonly used data structures. The linked-list seems a good candidate for parallelization, as modifications to different parts of the list may be executed independently and concurrently. Indeed, parallel linked-lists with various progress properties are abundant in the literature. Among these are lock-free linked-lists. A lock-free data structure ensures that when several threads access the data structure concurrently, at least one makes progress within a bounded number of steps. While this property ensures general system progress, it does not prevent starvation of a particular thread, or of several threads. Wait-free data structures ensure that each thread makes progress within a bounded number of steps, regardless of other threads' concurrent execution. Wait-free data structures are crucial for real-time systems, where a deadline may not be missed even in a worst-case scenario. To allow real-time systems and other systems with critical worst-case demands make use of concurrent data structures, we must provide the strong wait-free guarantee. Furthermore, wait-freedom is a desirable progress property for many systems, and in particular operating systems, interactive systems, and systems with service-level guarantees. For all those, the elimination of starvation is highly desirable.

[*]Dept. of Computer Science, Technion, stimnat@cs.technion.ac.il. Fax: +97248293900
[†]Dept. of Computer Science, Technion, anastas@cs.technion.ac.il
[‡]Dept. of Computer Science, Technion, sakogan@cs.technion.ac.il
[§]Dept. of Computer Science, Technion, erez@cs.technion.ac.il

Despite the great practical need for data structures that ensure wait-freedom, almost no practical wait-free data structure is known, because data structures that ensure wait-freedom are notoriously hard to design. Recently, wait-free designs for the simple stack and queue data structures appeared in the literature [7, 2]. Wait-free stack and queue structures are not easy to design, but they are considered less challenging as they present limited parallelism, i.e., a limited number of contention points (the head of the stack, and the head and the tail of the queue). We are not aware of any practical wait-free design for any other data structure that allows multiple concurrent operations to occur simultaneously. In particular, to the best of our knowledge, there is no wait-free linked-list algorithm available in the literature except for algorithms of universal constructions, which do not provide practical efficiency.

The main contribution of this work is a practical, linearizable, fast and wait-free linked-list. Our construction builds on the lock-free linked-list of Harris [4], and extends it using a helping mechanism to become wait-free. The main technical difficulty is making sure that helping threads perform each operation correctly, apply each operation exactly once, and return a consistent result (of success or failure) according to whether each of the threads completed the operation successfully. This task is non-trivial and it is what makes wait-free algorithms notoriously hard to design. Our design deals with several races that come up, and a proof of correctness makes sure that no further races exist. Some of our techniques may be useful in future work, especially the *success bit* introduced to determine the owner of a successful operation. Next, we extend our design using the fast-path-slow-path methodology of Kogan and Petrank [8], in order to make it even more efficient, and achieve performance that is almost equivalent to that of the lock-free linked-list of Harris. Here, the idea is to combine both lock-free and wait-free algorithms so that the (lock-free) fast path runs with (almost) no overhead, but is able to switch to the (wait-free) slow path when contention interferes with its progress. It is also important that both paths are able to run concurrently and correctly. Combining the newly obtained wait-free linked-list with the existing lock-free linked-list of Harris is an additional design challenge that is, again, far from trivial.

We have implemented the new wait-free linked-list and compared its efficiency with that of Harris's lock-free linked-list. Our first design(slightly optimized) performs worse by a factor of 1.5 when compared to Harris's lock-free algorithm. This provides a practical, yet not optimal, solution. However, the fast-path-slow-path extension reduces the overhead significantly, bringing it to just 2-15 percents. This seems a reasonable price to pay for obtaining a data structure with the strongest wait-free guarantee, providing non-starvation even in worst-case scenarios, and making it available for use with real-time systems.

We begin in Section 2 with an overview of the algorithm and continue in Section 3 with a detailed description of it. Highlights of the correctness proof appear in Section 4. The linearization points of the algorithm are specified in Section 5. We give describe the fast-path-slow-path extension of the algorithm in Section 6, Section 7 presents the performance measurements, and we conclude in Section 8. A full correctness proof for the algorithm is given in Appendix A. Java implementations of the wait-free algorithm and of the fast-path-slow-path extension are given in Appendices B and D respectively. The basic algorithm uses versioned pointers (pointers with a counter associated with them). Appendix C gives a variation of the algorithm that eliminates their need, and uses only regular pointers.

1.1 Background and Related Work

The first lock-free linked-list was presented by Valois [12]. A simpler and more efficient lock-free algorithm was designed by Harris [4], and Michael [10] added a hazard-pointers mechanism to allow lock-free memory management for this algorithm. Fomitchev and Rupert achieved better theoretical complexity in [3]. Herlihy and Shavit implemented a variation of Harris's algorithm [6], and we used this implementation both for comparison and as the basis for the Java code we developed.

Wait-free queues were presented in [7, 2]. A different approach for building concurrent lock-free or wait-free data structures is the use of universal constructions [5, 6, 1]. However, universal constructions (at least for the linked-list) are not efficient enough to be applied in practice, and are often non-scalable.

Recently, Kogan and Petrank [8] presented the fast-path-slow-path technique mentioned above. We use the fast-path-slow-path methodology in this work to achieve an efficient and wait-free linked-list.

Our wait-free linked-list design follows the traditional practice, in which concurrent linked-list data structures realize a sorted list, where each key may only appear once in the list [3, 4, 6, 9, 12]. A brief announcement of this work appeared in [11].

2 An Overview of the Algorithm

Before getting into the technical details (in Section 3) we provide an overview of the design. The wait-free linked-list supports three operations: INSERT, DELETE, and CONTAINS. All of them run in a wait-free manner. The underlying structure of the linked-list is depicted in Figure 2. Similarly to Harris's linked-list, our list contains sentinel `head` and `tail` nodes, and the `next` pointer in each node can be marked using a special `mark bit` , to signify that the entry in the node is logically deleted.

To achieve wait-freedom, our list employs a helping mechanism. Before starting to execute an operation, a thread starts by publishing an *Operation Descriptor*, in a special `state` array, allowing all the threads to view the details of the operation it is executing. Once an operation is published, all threads may try to help execute it. When an operation is completed, the result is reported to the state array, using a CAS which replaces the existing operation descriptor with one that contains the result.

A top-level overview of the insert and delete operations is provided in Figure 1. When a thread wishes to

```
1: boolean insert(key)
2:     Allocate new node (without help)
3:     Publish the operation (without help)
4:     Search for a place to insert the node
5:         If key already exists, return with failure
6:     Direct the new node next pointer
7:     Insert the node by directing its predecessor next pointer
8:     Return with Success
```

```
1: boolean delete(key)
2:     Publish the operation (without help)
3:     Search for the node to delete
4:         If key doesn't exist, return with failure
5:     Announce in the state array the node to be deleted
6:     Mark the node next pointer to make it logically deleted
7:     Physically remove the node
8:     Report the node has been removed
9:     Compete for success (without help)
```

Figure 1: Insert and Delete Overview

INSERT a key k to the list, it first allocates a new node with key k, and then publishes an operation descriptor with a pointer to the new node. The rest of the operation can be executed by any of the threads in the system, and may also be run by many threads concurrently. Any thread that executes this operation starts by searching for a place to insert the new node. This is done using the search method, which, given a key k, returns a pair of pointers, *prev* and *curr*. The prev pointer points to the node with the highest key smaller than k, and the curr pointer points to the node with the smallest key larger than or equal to k. If the returned curr node holds a key equal to the key on the node to be inserted, then failure is reported. Otherwise the node should be inserted between prev and curr. This is done by first updating the new node's `next` pointer to point to curr, and then updating prev's `next` field to point to it. Both of these updates are done using a CAS to prevent race conditions, and the failure of any of these CASes will cause the operation to restart from the search method. Finally, after that node has been inserted, success is reported.

While the above description outlines the general process of inserting a node, the actual algorithm is a lot more complex, and requires care to avoid problematic races that can make things go wrong. For example, when two different threads help insert the same node, they might get different prev and curr pointers back from the search method, due to additional changes that are applied concurrently on the list. This could lead to various problems, such as one of the threads reporting failure (since it sees another node with the same key) while the other thread successfully inserts the node (since it doesn't see the same node, which has been removed). In addition to these possible inconsistencies, there is also a potential ABA problem that requires the use of a version mark on the `next` pointer field[1]. We discuss these and other potential races in Section 3.4.

When a thread wishes to DELETE a key k from the list, it starts by publishing the details of its operation in the `state` array. The next steps can be then executed by any of the threads in the system until the last step, which is executed only by the thread that initiated the operation, denoted the *owner thread*. The DELETE operation is executed (or helped) in two stages. First, the node to be deleted is chosen. To do this, the search method is invoked. If no node with the key k is found, failure is reported. Otherwise, the node to be deleted is *announced* in the `state` array. This is done by replacing the state descriptor that describes this operation to a state descriptor that has a pointer to the specific node to be deleted. This announcement helps to ascertain that concurrent helping threads will not delete two different nodes, as the victim node for this operation is determined to be the single node that is announced in the operation descriptor. In the second stage, deletion is executed similarly to Harris's linked-list: the removed node's `next` field is marked, and then this node is physically removed from the list. The node's removal is then reported back to the `state` array.

However, since multiple threads execute multiple operations, and as it is possible that several operations attempt to DELETE the same node, it is crucial that exactly one operation be declared as successfully deleting the node's key and that the others return failure. An additional (third) stage is required in order to consistently determine which operation can be considered successful. This step is executed only by the owner threads, and is given no help. The threads that initiated the concurrent delete operations compete among themselves for the ownership of the deletion. To this end, an extra `success-bit` designated for this purpose is added to each node in the list. The thread that successfully CASes this bit from false to true is the only one that reports success for this deletion. We believe that using an extra bit to determine an ownership of an operation is a useful mechanism for future wait-free constructions as well. The full details of the DELETE operation are given in Appendix 3.5.

The CONTAINS operation is much simpler than the other two. It starts by publishing the operation. Any helping thread will then search for it in the list, reporting success (on the operation record) if the key was found, or failure if it was not.

3 The Algorithm

In this section we present the details of the algorithm.

3.1 The Underlying Data Structures

The structure of the linked-list is depicted in Figure 2. A node of the linked list consists of three fields: a key, a `success bit` to be used when deleting this node, and a special pointer field. The special pointer field

[1]The versioning method provides a simple solution to the ABA problem. A more involved solution that does not require a versioned pointer appears in Appendix C.

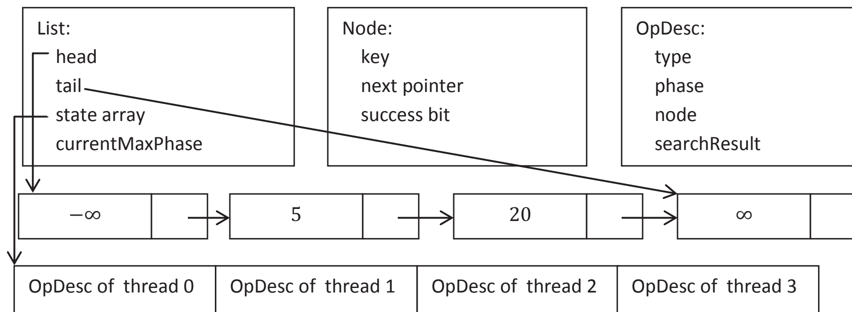

Figure 2: General structure

has its least significant bit used by the algorithm for signaling between threads. In addition, this pointer is versioned, in the sense that there is a counter associated with it (in an adjacent word) and each modification of it (or of its special bit) increments the counter. The modification and counter increment are assumed to be atomic. This can be implemented by squeezing all these fields into a single word, and limiting the size of the counter and pointer, or by using a double-word compare-and-swap when the platform allows. Alternatively, one can allocate a "pointer object" containing all these fields and bits, and then atomically replace the existing pointer object with a new one. The latter approach is commonly used with Java lock-free implementations, and we use it as well.

In addition to the nodes of the list, we also maintain an array with an operation-descriptor for each thread in the system. The OpDesc entry for each thread describes its current state. It consists of a phase field `phase` , the `OpType` field signifying which operation is currently being executed by this thread, a pointer to a node, denoted `node` , which serves the insert and delete operations, and a pair of pointers $(prev, curr)$, for recording the result of a search operation. Recall that the result of a SEARCH operation of a key, k, is a pair of pointers denoted *prev* and *curr*, as explained in Section 2 above.

The possible values for the operation type (OpType) in the operation descriptor state are:

insert	asking for help in inserting a node into the list.
search_delete	asking for help in finding a node with the key we wish to delete.
execute_delete	asking for help in marking a node as deleted (by tagging its next pointer) and unlinking it from the list.
contains	asking for help in finding out if a node with the given key exists.
success	operation was completed successfully.
failure	operation failed (deletion of a non-existing key or insertion of an existing key).
determine_delete	decide if a delete operation completed successfully.

The first four states in the above list are used to request help from other threads. The last three states indicate steps in the executions in which the thread does not require any help. The linked-list also contains an additional long field, `currentMaxPhase` , to support the helping mechanism, as described in Subsection 3.2.

3.2 The Helping Mechanism

Before a thread starts executing an operation, it first selects a phase number larger than all previously chosen phase numbers. The goal of assigning a phase number to each operation is to let new operations make

sure that old operations receive help and complete before new operations are executed. This ensures non-starvation. The phase selection mechanism ensures that if operation O_2 arrives strictly later than operation O_1, i.e., O_1 receives a phase number before O_2 starts selecting its own phase number, then O_2 will receive a higher phase number. The phase selection procedure is executed in the MAXPHASE method depicted in Figure 3. Note that although a CAS is used in this method, the success of this CAS is not checked, thus preserving wait-freedom. If the CAS fails, it means that another thread increased the counter concurrently, which is sufficient for the phase numbering. After selecting a phase number, the thread publishes the operation by updating its entry in the `state` array. It then goes through the array, helping all operations with a phase number lower than or equal to its own. This ensures wait-freedom: a delayed operation eventually receives help from all threads and soon completes. See Figure 3 for the pseudo-code.

```
 1: private long maxPhase() {
 2:    long result = currentMaxPhase.get();
 3:    currentMaxPhase.compareAndSet(result, result+1);
 4:    return result;
 5: }
 6:
 7: private void help(long phase) {
 8:    for (int i = 0; i < state.length(); i++) {
 9:       OpDesc desc = state.get(i);
10:       if (desc.phase <= phase) {        ▷ help older perations
11:          if (desc.type == OpType.insert) {
12:             helpInsert(i, desc.phase);
13:          } else if (desc.type == OpType.search_delete
14:             || desc.type == OpType.execute_delete) {
15:             helpDelete(i, desc.phase);
16:          } else if (desc.type == OpType.contains) {
17:             helpContains(i, desc.phase);
18:          } } } }
19:
20: private boolean isSearchStillPending(int tid, long ph) {
21:    OpDesc curr = state.get(tid);
22:    return (curr.type == OpType.insert ||
23:          curr.type == OpType.search_delete ||
24:          curr.type == OpType.execute_delete ||
25:          curr.type==OpType.contains) &&
26:       curr.phase == ph;
27: }
28: private Window search(int key, int tid, long phase) {
29:    Node pred = null, curr = null, succ = null;
30:    boolean[] marked = {false}; boolean snip;
31:    retry : while (true) {
32:       pred = head;
33:       curr = pred.next.getReference();        ▷ advancing curr
34:       while (true) {
35:                    ▷ Reading both the reference and the mark:
36:          succ = curr.next.get(marked);
37:          while (marked[0]) {        ▷ curr is logically deleted
38:                         ▷ Attempt to physically remove curr:
39:             snip = pred.next.compareAndSet
40:                   (curr, succ, false, false);
41:             if (!isSearchStillPending(tid,phase))
42:                return null;             ▷ to ensure wait-freedom.
43:             if (!snip) continue retry; ▷ list has changed, retry
44:             curr = succ;                       ▷ advancing curr
45:             succ = curr.next.get(marked);   ▷ advancing succ
46:          }
47:          if (curr.key >= key)           ▷ The window is found
48:             return new Window(pred, curr);
49:          pred = curr; curr = succ;   ▷ advancing pred & curr
50:       }
51:    }
52: }
53:
54:
55:
```

Figure 3: The Help and Search methods

3.3 The Search Methods

The CONTAINS method, which is part of the data structure interface, is used to check whether a certain key is a part of the list. The SEARCH method is used (internally) by the INSERT, DELETE, and CONTAINS methods to find the location of a key and perform some maintenance during the search. It is actually nearly identical to the original lock-free SEARCH method. The SEARCH method takes a key and returns a pair of pointers denoted *window*: `pred`, which points to the node containing the highest key less than the input key, and `curr`, which points to the node containing the lowest key higher than or equal to the requested key. When

traversing through the list, the SEARCH method attempts to physically remove any node that is logically deleted. If the remove attempt fails, the search is restarted from the head of the list. This endless attempt to fix the list seems to contradict wait-freedom, but the helping mechanism ensures that these attempts eventually succeed. When an operation delays long enough, all threads reach the point at which they are helping it. When that happens, the operation is guaranteed to succeed. The SEARCH operation will not re-iterate if the operation that executes it has completed, which is checked using the ISSEARCHSTILLPENDING method. If the associated operation is complete, then the SEARCH method returns a null. The pseudo-code for the search method is depicted in Figure 3.

3.4 The Insert Operation

Designing operations for a wait-free algorithm requires dealing with multiple threads executing each operation, which is substantially more difficult than designing a lock-free operation. In this section, we present the insert operation and discuss some of the races that occur and how we handle them. The basic idea is to coordinate the execution of all threads using the operation descriptor. But more actions are required, as explained below. Of-course, a proof is required to ensure that all races have been handled. The pseudo-code of the INSERT operation is provided in Figure 4. The thread that initiates the operation is denoted *the operation owner*. The *operation owner* starts the INSERT method by selecting a phase number, allocating a new node with the input key, and installing a link to it in the `state` array.

Next, the thread (or any helping thread) continues by searching the list for a location where the node with the new key can be inserted (Line 17 in the method HELPINSERT). In the original lock-free linked-list, finding a node with the same key is interpreted as failure. However, in the presence of the helping mechanism, it is possible that some other thread that is helping the same operation has already inserted the node but has not yet reported success. It is also possible that the node we are trying to insert was already inserted and then deleted, and then a different node, with the same key, was inserted into the list. To identify these cases, we check the node that was found in the search. If it is the same node that we are trying to insert, then we know that success should be reported. We also check if the (`next` field of the) node that we are trying to insert is *marked* for deletion. This happens if the node was already inserted into the list and then removed. In this case, we also report success. Otherwise, we attempt to report failure. If there is no node found with the same key, then we can try to insert the node between `pred` and `curr`. But first we check to see if the node was already inserted and deleted (line 35), in which case we can simply report success.

The existence of other threads that help execute the same operation creates various races that should be properly handled. One of them, described in the next paragraph, requires the INSERT method to proceed with executing something that may seem redundant at first glance. The INSERT method creates a state descriptor identical to the existing one and atomically replaces the old one with the new one (Lines 42–45). The replacement foils all pending CAS operations by other threads on this state descriptor, and avoids confusion as to whether the operation succeeds or fails. Next, the method executes the actual insertion of the node into the list (Lines 46–48) and it attempts to report success (Lines 49–52). If any of the atomic operations fail, the insertion starts from scratch. The actual insertion into the list (Lines 46–48) is different from the insertion in the original lock-free linked-list. First, the `next` pointer in the new node is not privately set, as it is now accessible by all threads that help the insert operation. It is set by a CAS which verifies that the pointer has not changed since before the search. Namely, the old value is read in Line 16 and used as the expected value in the CAS of Line 46. This verification avoids another race, which is presented below. Moreover, the atomic modification of the `next` pointer in the previous node to point to the inserted node (Lines 47–48) uses the version of that `next` pointer to avoid the ABA problem. This is also justified below.

Let us first present the race that justifies the (seemingly futile) replacement of the state descriptor in

```
1:  public boolean insert(int tid, int key) {
2:     long phase = maxPhase();                                  ▷ getting the phase for the op
3:     Node newNode = new Node(key);                             ▷ allocating the node
4:     OpDesc op = new OpDesc(phase, OpType.insert, newNode,null);
5:     state.set(tid, op);                                       ▷ publishing the operation
6:     help(phase);      ▷ when finished - no more pending operation with lower or equal phase
7:     return state.get(tid).type == OpType.success;
8:  }
9:
10: private void helpInsert(int tid, long phase) {
11:    while (true) {
12:       OpDesc op = state.get(tid);
13:       if (!(op.type == OpType.insert && op.phase == phase))
14:          return;                                             ▷ the op is no longer relevant, return
15:       Node node = op.node;                                   ▷ getting the node to be inserted
16:       Node node_next = node.next.getReference();
17:       Window window = search(node.key,tid,phase);
18:       if (window == null)                                    ▷ operation is no longer pending
19:          return;
20:       if (window.curr.key == node.key) {                     ▷ chance of a failure
21:          if ((window.curr==node)||(node.next.isMarked())){   ▷ success
22:             OpDesc success =
23:                new OpDesc(phase, OpType.success, node, null);
24:             if (state.compareAndSet(tid, op, success))
25:                return;
26:          }
27:          else {                                              ▷ the node was not yet inserted - failure
28:             OpDesc fail=new OpDesc(phase,OpType.failure,node,null);
29:                                   ▷ the following CAS may fail if search results are obsolete:
30:             if (state.compareAndSet(tid, op, fail))
31:                return;
32:          }
33:       }
34:       else {
35:          if (node.next.isMarked()){                          ▷ already inserted and deleted
36:             OpDesc success =
37:                new OpDesc(phase, OpType.success, node, null);
38:             if (state.compareAndSet(tid, op, success))
39:                return;
40:          }
41:          int version = window.pred.next.getVersion();        ▷ read version.
42:          OpDesc newOp=new OpDesc(phase,OpType.insert,node,null);
43:                                   ▷ preventing another thread from reporting a failure:
44:          if (!state.compareAndSet(tid, op, newOp))
45:             continue;                                        ▷ operation might have already reported as failure
46:          node.next.compareAndSet(node_next,window.curr,false,false);
47:          if (window.pred.next.compareAndSet
48:                (version, node.next.getReference(), node, false, false)) {
49:             OpDesc success =
50:                new OpDesc(phase, OpType.success, node, null);
51:             if (state.compareAndSet(tid, newOp, success))
52:                return;
53:          }
54:       }
55:    }
56: }
```

Figure 4: The insert operation

Lines 42–45. Suppose Thread T_1 is executing an INSERT operation of a key k. T_1 finds an existing node with the key k and is about to report failure. T_1 then gets stalled for a while, during which the other node with the key k is deleted and a different thread, T_2, helping the same INSERT operation that T_1 is executing, does find a proper place to insert the key k, and does insert it, but at that point T_1 regains control and changes the descriptor state to erroneously report failure. This sequence of events is bad, because a key has been inserted but failure has been reported. To avoid such a scenario, upon finding a location to insert k, T_2 modifies the operation descriptor to ensure that no stalled thread can wake up and succeed in writing a stale value into the operation descriptor.

Next, we present a race that justifies the setting of the `next` pointer in the new node (Line 46). The INSERT method verifies that this pointer has not been modified since it started the search. This is essential to avoid the following scenario. Suppose Thread T_1 is executing an INSERT of key k and finds a place to insert the new node N in between a node that contains $k-1$ and a node that contains $k+2$. Now T_1 gets stalled for a while and T_2, helping the same INSERT operation, inserts the node N with the key k , after which it also inserts another new node with key $k+1$, while T_1 is stalled. At this point, Thread T_1 resumes without knowing about the insertion of these two nodes. It modifies the next pointer of N to point to the node that contains $k+2$. This modification immediately foils the linked-list because it removes the node that contains $k+1$ from the list. By making T_1 replace the `next` field in N atomically only if this field has not changed since before the search, we know that there could be no node between N and the node that followed it at the time of the search.

Finally, we justify the use of a version for the `next` pointer in Line 47, by showing an ABA problem that could arise when several threads help executing the same insert operation. Suppose Thread T_1 is executing an INSERT of the key k into the list. It searches for a location for the insert, finds one, and gets stalled just before executing Line 47. While T_1 is stalled, T_2 inserts a different k into the list. After succeeding in that insert, T_2 tries to help the same insert of k that T_1 is attempting to perform. T_2 finds that k already exists and reports failure to the state descriptor. This should terminate the insertion that T_1 is executing with a failure report. But suppose further that the other k is then removed from the list, bringing the list back to exactly the same view as T_1 saw before it got stalled. Now T_1 resumes and the CAS of Line 47 actually succeeds. This course of events is bad, because a key is inserted into the list while a failure is reported about this insertion. This is a classical ABA problem, and we solve it using versioning of the next pointer. The version is incremented each time the `next` pointer is modified. Therefore, the insertion and deletion of a different k key while T_1 is stalled cannot go unnoticed. [2]

3.5 The Delete Operation

In this section we describe the DELETE operation. Again, a more complicated mechanism is required to safely execute the operation by multiple threads. Most of the problems are solved by a heavy use of the operation record to coordinate the concurrently executing threads. However, an interesting challenge here is the proper report of success or failure of the deletion in a consistent manner. We handle this problem using the `success bit` as described below.

The pseudo-code of the DELETE operation is provided in Figure 5. The DELETE operation starts when a thread changes its state descriptor to announce the key that needs to be deleted, and that the current state is *search_delete* (the first stage in the delete operation). The thread that performs this DELETE operation is called the *operation owner*. After setting its state descriptor, other threads may help the delete. The main

[2] We also implemented a more involved technique for handling this problem, using only a regular Markable Pointer. The full code for this alternative solution is given in Appendix C.

part of the DELETE operation, which is run in the HELPDELETE method, is partitioned into two. It starts with the initial state *search_delete* and searches for the requested key. If the requested key is found, then the state is updated to *execute_delete*, while leaving the PRED and CURR pair of pointers in the operation descriptor. From that point and on, there is a specific node whose deletion is attempted. In particular, when the state becomes *execute_delete*, it can never go back to *search_delete*. If the requested key is not found, HELPDELETE will attempt to report failure (Lines 27–28).

As the state becomes *execute_delete* and the node to be deleted is fixed, the second stage is executed in Lines 36–44. The attemptMark method used on the pointer in Line 38 tests that the pointer points to the expected reference, and if so, attempts by an atomic CAS to mark it for deletion. It returns true if the CAS succeeded, or if the node was marked already. In lines 37–39, the thread repeatedly attempts to mark the found node as deleted. After succeeding, it runs a search for the node. Our SEARCH method guarantees that the node of the corresponding DELETE operation is "physically" disconnected from the list. After deleting the node, the state is changed into *determine_delete* (Line 41–43), a special state meaning the operation is to be completed by the owner thread. The deleted node is linked to the operation descriptor, and the method returns.

Helping DELETE is different from helping INSERT in the sense that the help method in this case does not execute the entire DELETE operation to its completion. Instead, it stops before determining the success of the operation, and lets the operation owner decide whether its operation was successful. Note that this does not foil wait-freedom, as the operation owner will never get stuck on deciding whether the operation was successful. When the help method returns, there are two possibilities. The simpler possibility is that the requested key was not found in the list. Here it is clear that the operation failed and in that case the state is changed by the helper to a failure and the operation can terminate. The other possibility is that the requested key was found and deleted. In this case, it is possible that several DELETE operations for the same key were run concurrently by several operation owners and by several helping threads. As the delete succeeded, it has to be determined which operation owner succeeded. In such a case there are several operation owners for the deletion of the key *k* and only one operation owner can return success, because a single DELETE has been executed. The others operation owners must report failure. This decision is made by the operation owners (and not by the helping threads) in Line 9 of the DELETE method itself. It employs a designated `success bit` in each node. Whoever sets this bit becomes the owner of the deletion for that node in the list and can report success. We believe that this technique for determining the success of a thread in executing an operation in the presence of helping threads can be useful in future constructions of wait-free algorithms.

3.6 The Contains Operation

The CONTAINS method does not modify the list structure. Accordingly, some publications claim that it is wait-free even without the use of a help mechanism (see [6]). This is not entirely accurate. For example, consider a linked-list of sorted strings. A CONTAINS method traversing it without any help may never reach the letter B, because of infinite concurrent insertions of strings starting with an A. Thus, we provide here an implementation of the CONTAINS method that employs a help mechanism[3].

The CONTAINS operation starts when a thread changes its state descriptor to announce the key it wants to find. It then proceeds to the help method as usual. In the HELPCONTAINS method, a helping thread calls the SEARCH method, and uses a CAS to try to alter the state to a success or failure, depending on whether the wanted key was found. The help mechanism guarantees that the search will not suffer from infinite

[3]Technically, for a list of sorted integers, it is possible to easily implement a wait-free contains that does not use the help mechanism since the number of possible keys is bounded. However, this yields a poor bound on the time.

```
1:  public boolean delete(int tid, int key) {
2:     long phase = maxPhase();                                   ▷ getting the phase for the op
3:     state.set(tid, new OpDesc
4:     (phase, OpType.search_delete, new Node(key),null));        ▷ publishing
5:     help (phase);     ▷ when finished - no more pending operation with lower or equal phase
6:     OpDesc op = state.get(tid);
7:     if (op.type == OpType.determine_delete)
8:                        ▷ Need to compete on the ownership of deleting this node:
9:        return op.searchResult.curr.success.compareAndSet(false, true);
10:    return false;
11: }
12:
13: private void helpDelete(int tid, long phase) {
14:    while (true) {
15:       OpDesc op = state.get(tid);
16:       if (!((op.type == OpType.search_delete ||
17:          op.type == OpType.execute_delete) &&
18:          op.phase==phase))
19:          return;                                  ▷ the op is no longer relevant, return
20:       Node node = op.node;                          ▷ holds the key we want to delete
21:       if (op.type == OpType.search_delete) {
22:          Window window = search(node.key,tid,phase);
23:          if (window==null)
24:             continue;                     ▷ operation is no longer the same search_delete
25:          if (window.curr.key != node.key) {          ▷ key doesn't exist - failure
26:             OpDesc failure=new OpDesc(phase,OpType.failure,node,null);
27:             if (state.compareAndSet(tid, op, failure))
28:                return;
29:          }
30:          else {                             ▷ key exists - continue to executre_delete
31:             OpDesc found = new
32:                OpDesc(phase, OpType.execute_delete, node, window);
33:             state.compareAndSet(tid, op, found);
34:          }
35:       }
36:       else if (op.type == OpType.execute_delete) {
37:          Node next = op.searchResult.curr.next.getReference();
38:          if (!op.searchResult.curr.next.attemptMark(next, true))          ▷ mark
39:             continue;                  ▷ will continue to try to mark it, until it is marked
40:          search(op.node.key,tid,phase);                  ▷ to physically remove the node
41:          OpDesc determine = new OpDesc
42:             (op.phase,OpType.determine_delete,op.node,op.searchResult);
43:          state.compareAndSet(tid, op, determine);
44:          return;
45:       }
46:    }
47: }
```

Figure 5: The delete operation

```
1:  public boolean contains(int tid, int key) {
2:        long phase = maxPhase();
3:        Node n = new Node(key);
4:        OpDesc op = new OpDesc(phase, OpType.contains, n, null);
5:        state.set(tid, op);
6:        help(phase);
7:        return state.get(tid).type == OpType.success;
8:     }
9:
10:    private void helpContains(int tid, long phase) {
11:       OpDesc op = state.get(tid);
12:       if (!((op.type == OpType.contains) && op.phase==phase))
13:          return;                                   ▷ the op is no longer relevant, return
14:       Node node = op.node;                          ▷ the node holds the key we need to search
15:       Window window = search(node.key, tid, phase);
16:       if (window == null)
17:          return;                                   ▷ can only happen if operation is already complete.
18:       if (window.curr.key == node.key) {
19:          OpDesc success = new OpDesc(phase, OpType.success, node, null);
20:          state.compareAndSet(tid, op, success);
21:       }
22:       else {
23:          OpDesc failure = new OpDesc(phase, OpType.failure, node, null);
24:          state.compareAndSet(tid, op, failure);
25:       }
26:    }
```

Figure 6: The *contains* and *helpContains* methods

concurrent insertions of new keys, since other threads will help this operation before entering new keys (perhaps excluding a key they are already in the process of inserting). The pseudo-code for the CONTAINS and the HELPCONTAINS methods is depicted in Figure 6. The HELPCONTAINS method differs from the HELPINSERT and HELPDELETE methods in that it doesn't require a loop, as a failure of the CAS updating the state for this operation can only occur if the operation was already completed.

3.7 Memory management

The algorithm in this work relies on a garbage collector (GC) for memory management. A wait-free GC does not currently exist. This is a common difficulty for wait-free algorithms. A frequently used solution, which suits this algorithm as well, is Michael's Hazard Pointers technique [10]. Hazard pointers can be used for the reclamation of the operation descriptors as well, and not only for the reclamation of the list nodes themselves.

4 Highlights of the Correctness Proof

We now briefly explain how this algorithm is proven correct. A full proof appears in Appendix A. A full proof is crucial for a parallel algorithm as without it, one can never be sure that additional races are not lurking in the algorithm.

Basic Concepts and Definitions. The *mark bit*, is the bit on the next field of each node, and it is used to mark the node as logically deleted. A node can be marked or unmarked according to the value of this bit. We define the nodes that are *logically in the list* to be the unmarked nodes that are reachable from the list's head. Thus, a *logical change* to the list, is a change to the set of unmarked nodes reachable from the head. We say that a node is an *infant node* if it has never been reachable from the head. These are nodes that have been prepared for insertions but have not been inserted yet.

In the proof we show that at the linearization point of a successful insert, the inserted value becomes logically in the list and that at a linearization point of a successful delete, a node with the given value is logically deleted from the list. To show this, we look at the actual *physical* modifications that may occur to the list.

Proof Structure. One useful invariant is that a *physical change* to the list can only modify the node's next field, as a node's key is final and never changes after the initialization of a node. A second useful invariant is that a marked node is never unmarked, and that it's next field never changes (meaning, it will keep pointing to the same node). This is ascertained by examining all the code lines that change a node's next field, and noting that all of them do it using a CAS which prevents a change from taking effect if the node is marked. We next look at all possible physical changes to a node's next field, and show that each of them falls in one of the following four categories:

* Marking: changing the mark bit of a node that is *logically in the list* to true.
* Snipping: physically removing a *marked* node out of the list.
* Redirection: a modification of an infant node's next pointer (in preparation for its insertion).
* Insertion: a modification of a non-infant node to point to an infant node (making the latter non-infant after the modification).

Proving that every *physical change* to a node's next field falls into one of the four categories listed above, is the most complicated part of the formal proof, and is done by induction, with several intermediate invariants. Finally, it is shown that any operation in the *marking* category matches a successful delete operation and any operation in the *insertion* category matches a successful insert operation. Thus, at the proper linearization points the linked list changes according to its specification. Furthermore, it is shown that physical operations in the *Redirection* and *Snipping* categories cause no *logical* changes to the list, which completes the linearizability proof.

To show wait-freedom, we claim that the helping mechanism ensures that a limited number of concurrent operations can be executed while a given insert or delete execution is pending. At the point when this number is exhausted, all threads will help the pending operation, and then it will terminates within a limited number of steps.

5 Linearization Points

In this section we specify the linearization point for the different operations of the linked-list. The SEARCH method for a key k returns a pair of pointers, denoted *pred* and *curr*. The prev pointer points to the node with the highest key smaller than k, and the curr pointer points to the node with the smallest key larger than or equal to k. The linearization point of the SEARCH method is when the pointer that connects *pred* to *curr* is read. This can be either at Line 36 or 45 of the SEARCH method. Note that *curr*'s `next` field will be

subsequently read, to make sure it is not *marked*. Since it is an invariant of the algorithm that a marked node is never unmarked, it is guaranteed that at the linearization point both *pred* and *curr* nodes were unmarked.

The linearization point for a CONTAINS method is the linearization point of the appropriate SEARCH method. The appropriate SEARCH method is the one called from within the HELPCONTAINS method by the thread that subsequently successfully reports the result of the same CONTAINS operation. The linearization point of a *successful insert* is in Lines 47-48 (together they are a single instruction) of the *helpInsert* method. This is the CAS operation that physically links the node into the list. For a *failing insertion*, the linearization point is inside the linearization point of the SEARCH method executed by the thread that reported the failure.

The linearization point of a *successful delete* is at the point where the node is *logically* deleted, which means successfully marked (Line 38 in the helpDelete method). Note that it is possible that this is executed by a helping thread and not necessarily by the operation owner. Furthermore, the helping thread might be trying to help a different thread than the one that will eventually own the deletion. The linearization point of an *unsuccessful delete* is more complex. A delete operation may fail when the key is properly deleted, but a different thread is selected as the owner of the delete. In this case, the current thread returns failure, because of the failure of the CAS of the DELETE method (at Line 9). In this case, the linearization point is set to the point when the said node is logically deleted, in Line 38 of HELPDELETE. The linearization point of an *unsuccessful delete*, originating from simply not finding the key, is the linearization point of the SEARCH method executed by the thread that reported the failure.

6 A Fast-Path-Slow-Path Extension

6.1 overview

In this section, we describe the extension of the naive wait-free algorithm using the fast-path-slow-path methodology. The goal of this extension is is to improve performance and obtain a fast wait-free linked-list. We provide a short description of the method here. Full motivation and further details appear in [8]. A full Java code for the fast-path-slow-path list is presented in Appendix D.

The idea behind the fast-path-slow-path [8] approach is to combine a (fast) lock-free algorithm with a (slower) wait-free one. The lock free algorithm provides a basis for a fast path and we use Harris's lock-free linked-list for this purpose. The execution in the fast path begins by a check whether a help is required for any operation in the slow path. Next, the execution proceeds with running the fast lock-free version of the algorithm while counting the number of contentions that end with a failure (i.e., failed CASes)[4]. Typically, few failures occur and help is not required, and so the execution terminates after running the faster lock-free algorithm. If this fast path fails to make progress, the execution moves to the slow path, which runs the slower wait-free algorithm described in Section 3, requesting help (using an operation descriptor in its slot in the state array) and making sure the operation eventually terminates.

The number of CAS failures allowed in the fast path is limited by a parameter called MAX_FAILURES. The help is provided by threads running both the fast and slow path, which ensures wait-freedom: if a thread fails to complete its operation, its request for help is noticed both in the fast and in the slow path. Thus, eventually all other threads help it and its operation completes. However, help is not provided as intensively as described in Section 3. We use the *delayed help* mechanism, by which each thread only offers help to other threads once every several operations, determined by a parameter called HELPING_DELAY.

[4]Another point to consider is the possibility that a thread can't make progress since other threads keep inserting new nodes to the list, and it can't finish the search method. We address this potential problem in Appendix 6.7.

Combining the fast-path and the slow-path is not trivial, as care is needed to guarantee that both paths properly run concurrently. On top of other changes, it is useful to note that the DELETE operation must compete on the `success-bit` even in the fast-path, to avoid a situation where two threads running on the two different paths both think they were successful in deleting a node.

6.2 The Delayed Help Mechanism

In order to avoid slowing the fast path down, help is not provided to *all* threads in the beginning of *each* operation execution. Instead, help is provided to at most one thread, in a round-robin manner. Furthermore, help is not provided in each run of an operation, but only once every few operation executions. This scheme still guarantees wait-freedom for the threads that require help, but it does not overwhelm the system with contention of many helping threads attempting to run the same operation on the same part of the list.

The above mechanism is called *delayed-help*. In addition to an entry in the state array, each thread maintains a helping record. The first field in a helping record holds the *TID* of the *helped thread*. This thread is the next one in line to receive help, if needed. In addition to the TID of the helped thread, the helping record holds a *nextCheck* counter, initialized to the HELPING_DELAY parameter and decremented with each operation that does not provide help, and a phase number, recording the phase the helped thread had when the help of the previous thread terminated.

Before a thread T performs an operation (in the fast or slow path), T decrements the `nextCheck` counter in its helping record by one. If `nextCheck` reaches zero, than T checks whether the helped thread has a pending operation (i.e., it needs help) and whether this pending operation has the same phase that was previously recorded. This means that the helped thread made no progress for a while. If this is the case, then T helps it. After checking the helped thread's state and providing help if required, T updates its help record. The field holding the TID of the helped thread is incremented to hold the id of the next thread, the phase of this next thread is recorded, and NEXTCHECK is initialized to HELPING_DELAY. Pseudo-code for this is depicted in Figure 7

6.3 The Search Method

The FASTSEARCH method (Figure 8) is identical to the original lock-free search, except for counting the number of failed CAS operations. If this number reaches MAX_FAILURES, FASTSEARCH returns null. It is up to the caller (fastInsert or fastDelete) to move to the slow path, if null is returned. The slowSearch method (called SEARCH hereafter) operation is identical to the wait-free search introduced method in Section 3.

6.4 The Insert Operation

The INSERT operation (Figure 9) starts in the fast path and retreats to SLOWINSERT (Figure 10) when needed. It starts by checking if help is needed. After that, it operates as the original lock-free insert, except for counting CAS failures. It also checks whether FASTSEARCH has returned a null, in which case it reverts to the slow path.

The SLOWINSERT method (Figure 10) is similar to the wait-free insert, except that it performs its own operation only, and does not help other operations. The helpInsert method is identical to the wait-free method presented in Section 3.

```
1:  class HelpRecord {
2:        int curTid; long lastPhase; long nextCheck;
3:        HelpRecord() { curTid = -1; reset(); }
4:        public void reset() {
5:           curTid = (curTid + 1) % Test.numThreads;
6:           lastPhase = state.get(curTid).phase;
7:           nextCheck = HELPING_DELAY;
8:        }
9:     }
10:
11: private void helpIfNeeded(int tid) {
12:    HelpRecord rec = helpRecords[tid*width];
13:    if (rec.nextCheck-- == 0) {                          ▷ delay help HELPING_DELAY times
14:       OpDesc desc = state.get(rec.curTid);
15:       if (desc.phase == rec.lastPhase) {                ▷ help might be needed
16:          if (desc.type == OpType.insert)
17:             helpInsert(rec.curTid, rec.lastPhase);
18:          else if (desc.type == OpType.search_delete ||
19:             desc.type == OpType.execute_delete)
20:             helpDelete(rec.curTid, rec.lastPhase);
21:       }
22:       rec.reset();
23:    }
24: }
```

Figure 7: The delayed help mechanism

```
1:  public Window fastSearch(int key) {
2:     int tries = 0; Node pred = null, curr = null, succ = null;
3:     boolean[] marked = {false};
4:     boolean snip;
5:     retry : while (tries++ < MAX_FAILURES) {                       ▷ do I need help?
6:        pred = head;
7:        curr = pred.next.getReference();                            ▷ advancing curr
8:        while (true) {
9:           succ = curr.next.get(marked);                            ▷ advancing succ
10:          while (marked[0]) {                                      ▷ curr is logically deleted
11:                     ▷ The following line is an attempt to physically remove curr:
12:             snip = pred.next.compareAndSet(curr, succ, false, false);
13:             if (!snip) continue retry;                            ▷ list has changed, retry
14:             curr = succ;                                          ▷ advancing curr
15:             succ = curr.next.get(marked);                         ▷ advancing succ
16:          }
17:          if (curr.key >= key)                                     ▷ the window is found
18:             return new Window(pred, curr);
19:          pred = curr; curr = succ;                                ▷ advancing pred & curr
20:       }
21:    }
22:    return null;                                                   ▷ asking for help
23: }
```

Figure 8: The FPSP fastSearch method

```
 1: public boolean insert(int tid, int key) {
 2:    helpIfNeeded(tid);
 3:    int tries = 0;
 4:    while (tries++ < MAX_FAILURES) {                         ▷ do I need help?
 5:       Window window = fastSearch(key);
 6:       if (window == null)                    ▷ search failed MAX_FAILURES times
 7:          return slowInsert(tid, key);
 8:       Node pred = window.pred, curr = window.curr;
 9:          if (curr.key == key)
10:          return false;                          ▷ key exists - operation failed.
11:       else {
12:          Node node = new Node(key);                ▷ allocate the node to insert
13:          node.next = new
14:             VersionedAtomicMarkableReference<Node>(curr, false);
15:          if (pred.next.compareAndSet(curr, node, false, false))
16:             return true;                                  ▷ insertion succeeded
17:       }
18:    }
19:    return slowInsert(tid, key);
20: }
```

Figure 9: The FPSP insert method

```
 1: private boolean slowInsert(int tid, int key) {
 2:    long phase = maxPhase();                        ▷ getting the phase for the op
 3:    Node n = new Node(key);                                 ▷ allocating the node
 4:    n.next = new
 5:       VersionedAtomicMarkableReference<Node>(null, false);
 6:    OpDesc op = new OpDesc(phase, OpType.insert, n,null);
 7:    state.set(tid, op);              ▷ publishing the operation - asking for help
 8:    helpInsert(tid, phase);                           ▷ only helping itself here
 9:    return state.get(tid).type == OpType.success;
10: }
```

Figure 10: The FPSP slowInsert method

```
1:  public boolean delete(int tid, int key) {
2:     helpIfNeeded(tid);
3:     int tries = 0; boolean snip;
4:     while (tries++ < MAX_FAILURES) {                              ▷ do I need help?
5:        Window window = fastSearch(key);
6:        if (window == null)                          ▷ search failed MAX_FAILURES times
7:           return slowDelete(tid, key);
8:        Node pred = window.pred, curr = window.curr;
9:        if (curr.key != key)                       ▷ key doesn't exist - operation failed
10:          return false;
11:       else {
12:          Node succ = curr.next.getReference();
13:                       ▷ The following line is an attempt to logically delete curr:
14:          snip = curr.next.compareAndSet(succ, succ, false, true);
15:          if (!snip)
16:             continue;                                             ▷ try again
17:                     ▷ The following line is an attempt to physically remove curr:
18:          pred.next.compareAndSet(curr, succ, false, false);
19:                    ▷ the following is needed for cooperation with the slow path:
20:          return curr.d.compareAndSet(false, true);
21:       }
22:    }
23:    return slowDelete(tid, key);
24: }
```

Figure 11: The FPSP delete method

6.5 The Delete Operation

The DELETE method (Figure 11) is similar to the delete operation of the original lock-free list with some additions. In addition to checking the number of failures, further cooperation is required between threads. Determining which thread deleted a value is complicated in the wait-free algorithm and requires some cooperation from the fast path as well. In particular, after performing a delete that is considered successful in the fast path, the new DELETE method must also atomically compete (i.e., try to set) the extra `success bit` in the node. This bit is used by the wait-free algorithm to determine which thread owns the deletion of a node. Neglecting to take part in setting this bit may erroneously allow both a fast-path delete and a concurrent slow-path delete to conclude that they both are successful for the same delete. Upon failing to set the `success bit` in the node, DELETE returns failure.

The SLOWDELETE method (Figure 12) is similar to the wait-free version of the DELETE method, except that it does not need to help any other threads. The HELPDELETE method is identical to the one presented in Section 3.

6.6 Linearization Points

The linearization points are simply the linearization points of the lock-free and wait-free algorithms, according to the path in which the operation takes place. In the fast path, a successful insert operation is the CAS linking the node to the list (line 15 in the insert method), and an unsuccessful one is at the fastSearch method (line 9 or 15, whichever is read last). A successful delete is linearized in a successful CAS in line 14 of the delete method. Note that it is possible for an unsuccessful delete to be linearized at this point too, if a slow-path operation will own this deletion eventually. The usual unsuccessful delete (the key doesn't exist)

```
 1: private boolean slowDelete(int tid, int key) {
 2:    long phase = maxPhase();                                    ▷ getting the phase for the op
 3:    state.set(tid, new OpDesc
 4:       (phase, OpType.search_delete, new Node(key), null));
 5:    helpDelete(tid,phase);                                      ▷ only helping itself here
 6:    OpDesc op = state.get(tid);
 7:    if (op.type == OpType.determine_delete)
 8:                       ▷ the following competes on the ownership of deleting the node:
 9:       return op.searchResult.curr.d.compareAndSet(false, true);
10:    return false;
11: }
```

Figure 12: The FPSP slowDelete method

linearization point is similar to the one described in Section 5, at the begining of the fastSearch method if the key didn't exist then, or at the point when it was marked, if it did exist. The other linearization points, those of the slow-path, are unchanged from those elaborated on in Section 5. It is worth noting that the linearization point of a successful delete in the slow path, which is always upon marking the node, might actually happen during a run of the fast path of a delete method.

6.7 The Contains Operation and Handling Infinite Insertions

In Section 3.6, we noted that infinite concurrent insertions into the list create a challenge to the wait-freedom property, since the CONTAINS method may never be able to reach the desired key if more and more keys are inserted before it. This problem has a bound when dealing with integers, as there is a bound to the number of possible integer keys, but has no bound when dealing with other types of keys, such as strings. If every operation on the list is always done using the help mechanism, this problem cannot occur, since other threads will help the pending operations before entering new keys. This is how the problem was handled in the CONTAINS method in Section 3.6.

It is perhaps debatable whether a wait-free algorithm should offer a solution for this problem, as the failure does not happen due to contention, but due to the fact that the linear complexity of the problem (in the number of keys) increases while the thread is working on it. This debate as beyond the scope of our work, and our goal here is to offer solutions to the problem. For the basic wait-free algorithm, we could solve this problem by making sure that all operations (including CONTAINS) will use the helping mechanism. However, for the fast-path-slow-path extension, it is by definition impossible to force all threads to use the helping mechanism, as this would contradict the entire point of the fast-path-slow-path. Instead, a thread must be able to recognize when its operation is delayed due to many concurrent insertions, and ask for help (aka, switch to the slow path) if this problem occurs. The purpose of this Appendix is to suggest an efficient way to do that.

The idea is that each thread will read the number of total keys in the list prior to starting the search. During the search, it will count how many nodes it traversed, and if the number of traversed nodes is higher than the original total number of keys (plus some constant), it will abort the search and ask for help in its operation. The problem is that maintaining the size of the list in a wait-free manner can be very costly. Instead, we settle for maintaining a field that approximates the number of keys. The error of the approximation is also bounded by a constant (actually, a linear function in the number of threads operating on the list). Thus, before a thread starts traversing the list, it should read the approximation, denoted $Size_App$, and if it traverses a number of nodes that is greater than $Size_App + Max_Error + Const$, switch to the slow path and ask for help.

To maintain the approximation for the number of keys in the list, the list contains a global field with the approximation, and each thread holds a private counter. In its private counter, each thread holds the number of nodes it inserted to the list minus the number of nodes it deleted from the list since the last time it updated the global approximation field. To avoid too much contention in updating the global field, each thread only attempts to update it (by a CAS) once it reached a certain *soft_threshold* (in absolute value). If the CAS failed, the thread continues the operation as usual, and will attempt to update the global approximation field at its next insert or delete operation. If the private counter of a thread reached a certain *hard_threshold*, it asks for help in updating the global counter, similarly to asking help for other operations.

Some care is needed to implement the helping mechanism for updating the approximation field in a wait-free manner. This is not very complicated, but is also not completely trivial. The full Java code that also handles this difficulty is given if Appendix D.

7 Performance

Implementation and platform. We compared four Java implementations of the linked-list. The first is the lock-free linked-list of Harris, denoted *LF*, as implemented by Herlihy and Shavit in [6]. (This implementation was slightly modified to allow nodes with user-selected keys rather than the object's hash-code. We also did not use the *item* field.)

The basic algorithm described in Section is denoted *WF-Orig* in the graphs below. A slightly optimized version of it, denoted *WF-Opt*, was changed to employ a delayed help mechanism, similar to the one used in the fast-path- slow-path extension. This means that a thread helps another thread only once every k operations, where k is a parameter of the algorithm set to 3. The idea is to avoid contention by letting help arrive only after the original thread has a reasonable chance of finishing its operation on its own. This optimization is highly effective, as seen in the results. Note that delaying help is not equivalent to a fast-path-slow-path approach, because all threads always ask for help (there is no fast path). All the operations are still done in the *helpInsert* and *helpDelete* methods.

The fast-path-slow-path algorithm, denoted *FPSP*, was run with the HELPING_DELAY parameter set to 3, and MAX_FAILURES set to 5. This algorithm combines the new wait-free algorithm described in this paper with Harris's lock-free algorithm, to achieve both good performance and the stronger wait-freedom progress guarantee.

We ran the tests in two environments. The first was a SUN's Java SE Runtime, version 1.6.0 on an IBM x3400 system featuring 2 Intel(R) Xeon(R) E5310 1.60GHz quad core processors (overall 8 cores). The second was a SUN FIRE machine with an UltraSPARC T1 8 cores each running four hyper-threads.

Workload and methodology. In the micro-benchmarks tested, we ran each experiment for 2 seconds, and measured the overall number of operations performed by all the threads during that time. Each thread performed 60% CONTAINS, and 20% INSERT and DELETE operations, with keys chosen randomly and uniformly in the range $[1,1024]$. The number of threads ranges from 1-16 (in the Intel(R) Xeon(R)) or from 1-32 (In the UltraSPARC). We present the results in Figure 13. The graphs show the total number of operations done by all threads in thousands for all four implementations, as a function of the number of threads. In all the tests, we executed each evaluation 8 times, and the averages are reported in the figures.

Results. It can be seen that the fast-path-slow-path algorithm is almost as fast as the lock-free algorithm. On the Intel machine, the two algorithms are barely distinguishable; the difference in performance is 2-3%. On the UltraSPARC the fast-path-slow-path suffers a noticeable (yet, reasonable) overhead of 9-14%. The (slightly optimized) basic wait-free algorithm is slower by a factor of 1.3–1.6, depending on the number of threads. Also, these three algorithms provide an excellent speed up of about 7 when working with 8 threads

(on both machines), and about 24 when working with 32 multi-threads on the UltraSPARC. The basic non-optimized version of the wait-free algorithm doesn't scale as well. There, threads often work together on the same operation, causing a deterioration in performance and scalability. The simple delayed-help optimization enables concurrency without foiling the worst-case wait-freedom guarantee.

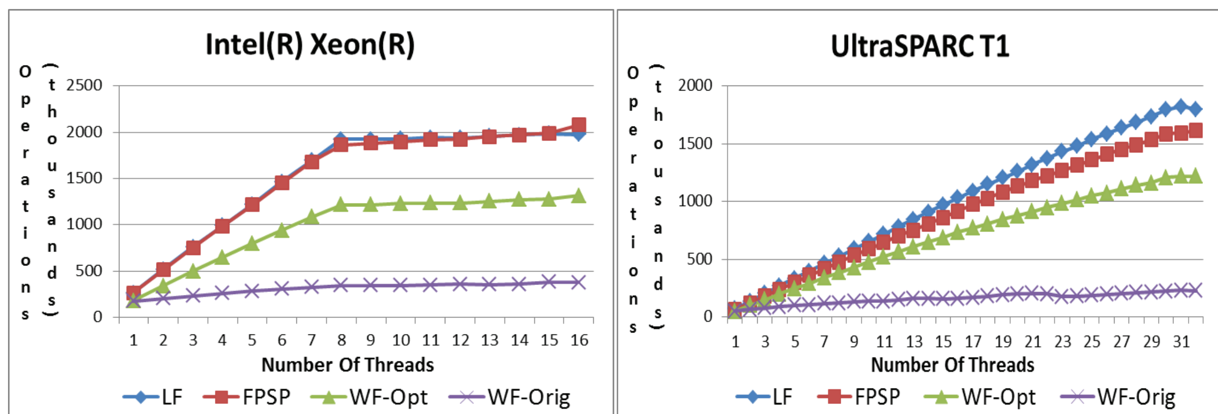

Figure 13: The number of operations done in two seconds as a function of the number of threads

8 Conclusion

We presented a wait-free linked-list. To the best of our knowledge, this is the first design of a wait-free linked-list in the literature, apart from impractical universal constructions. This design facilitates for the first time the use of linked-lists in environments that require timely responsiveness, such as real-time systems. We have implemented this linked-list in Java and compared it to Harris's lock-free linked-list. The naive wait-free implementation is slower than the original lock-free implementation by a factor of 1.3 to 1.6. We then combined our wait-free design with Harris's lock-free linked-list design using the fast-path-slow-path methodology, and implemented the extended version as well. The extended algorithm obtains performance which is very close to the original lock-free algorithm, while still guaranteeing non-starvation via wait-freedom.

References

[1] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *SPAA*, pages 335–344, 2010.

[2] Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In *SPAA*, pages 325–334, 2011.

[3] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *PODC*, pages 50–59, New York, NY, USA, 2004. ACM.

[4] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, London, UK, UK, 2001. Springer-Verlag.

[5] Maurice Herlihy. A methodology for implementing highly concurrent objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.

[6] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[7] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *PPOPP*, pages 223–234, 2011.

[8] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *PPOPP*, pages 141–150, 2012.

[9] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, New York, NY, USA, 2002. ACM.

[10] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[11] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *PPOPP*, pages 309–310, 2012.

[12] John D. Valois. Lock-free linked lists using compare-and-swap. In *PODC*, pages 214–222, New York, NY, USA, 1995. ACM.

A A Correctness Proof

In this appendix we elaborate the proof for correctness and wait-freedom of the algorithm described in Section 3, and in particular of its Java implementation in Appendix B. All references to lines of code refer to the implementation of Appendix B

A.1 General

The linked list interface corresponds to that of a set of keys. The keys considered to be in the set at any given point are the keys found on *unmarked* (see Definition 17) nodes reachable from the head. An insert(key) method should succeed (return true) and add the key to the set if and only if the key is not in the set; otherwise it should fail (return false). A delete(key) method should succeed (return true) and remove a key from the set if and only if the key is in the set; otherwise it should fail (return false). The contains method is not included in this proof, since it has not changed from previous implementations, and is independent of the rest of the proof.

A.2 Definitions

Definition 1 *Head key and Tail key. The Head key is defined to be smaller than all valid keys, and the tail key is greater than all valid keys.*

Definition 2 *A threadID (or tid). A threadID is a unique identifier for each thread.*

Definition 3 *Operation. An operation is an attempt to insert or delete a key from the list, and is initiated by calling either the insert or the delete method.*

Definition 4 *Legal operation. A legal operation is initiated by a thread calling either the insert(tid, key) or delete(tid, key) method with its own tid. (Calling it with a different tid is considered illegal.) Moreover, the key must be strictly greater than the head key, and strictly smaller than the tail key. We assume no illegal operations are attempted.*

Definition 5 *Operation phase number. Each operation receives a* phase *number, chosen at the insert or delete method that initiated it. This is the number returned from the maxPhase() method called from the appropriate (insert or delete) method.*

Definition 6 *Operation's methods. The insert or delete method that initiated an operation is part of the operation. In addition the search, helpInsert, and helpDelete methods all receive a tid and a phase number as parameters. They are thus considered as a part of the operation that corresponds to this tid & phase pair.*

Definition 7 *Operation owner. The operation owner, or the owner thread, is the thread that initiated the operation.*

Definition 8 *The operation's node, the operation's key. The (single) node allocated in each insert or delete operation will be called the operation's node. The node can also be said to belong to the operation. Its key will be called the operation's key. At an insert operation, we may also refer to the operation's node as the* inserted node.

Definition 9 *Successful operation. A successful operation is an operation for which the (insert or delete) method that initiated it returned true.*

Definition 10 *Thread's entry. A thread's entry in the state array is the entry in the state array corresponding to state[tid].*

Definition 11 *Thread's state. A thread's state is the OpType of its entry in the state array (one of: insert, search_delete, execute_delete, success, failure, determine_delete).*

Definition 12 *State's phase number. A state's* phase number *is the phase number present at the phase field of its entry in the state array.*

Definition 13 *Pending states. The insert, search_delete and execute_delete states are considered* pending *states. The other states are* non-pending.

Definition 14 *Pending operation. An operation is considered* pending *if its owner's state is pending with the phase number of the operation.*

Definition 15 *Publishing an operation. A thread* publishes *an operation, by (first) changing its state into pending, with the phase number of the operation. (This is done only at the insert and delete methods, and if all operations are legal, can only be done by the operation's owner.)*

Definition 16 *List initialization. The list initialization includes all the actions done in the constructor of the list. These operations must all be completed before the initialization of the first operation to the list.*

Definition 17 *Mark bit. A node's* mark bit *is the additional bit at the node's next field. A node is considered* marked *if this node is on (set to 1). Otherwise a node is said to be unmarked.*

Definition 18 *Reachable node. A* reachable *node is a node reachable from the head. Sometimes we shall specifically mention 'a node reachable from node x', but otherwise reachable means reachable from the head.*

Definition 19 *Nodes/Keys logically in the list. The set of nodes logically in the list is the set of unmarked reachable nodes. The set of keys logically in the list is the set of keys that are in the set of nodes logically in the list.*

Definition 20 *Logical change. A* logical change *to the list is a change to the set of* unmarked reachable *nodes.*

Definition 21 *Physical change. A* physical change *to the list is a change to one of the fields (key, next, or mark bit) of a node.*

Definition 22 *Infant node. At any given point, an infant node is a node that was not reachable until that point.*

Definition 23 *Node's logical set. A node's logical set is the set of unmarked nodes reachable from it, not including itself.*

Definition 24 *Node's inclusive logical set. A node's inclusive logical set is the set of unmarked nodes reachable from it, including itself. (Note that for a marked node, its logical set is identical to its inclusive logical set.)*

One final note regarding the definitions: in order to prove correctness, we must also assume the phase number will not overflow its range and thus become negative (or duplicate). When using the long field, this can be done by assuming no more than 2^{63} operations are executed on the list. Although this limit is surely enough for any practical use, we do not want to give any bound to the number of operations, because that will severely limit the value of the wait-freedom definition. Instead, we will assume that if the number of operations is bigger than 2^{63}, the phase field will be replaced with a field of sufficient size.

A.3 General List Invariants

Observation 25 *After the list initialization, the head and tail never change, meaning that the head and tail fields of the list always refer to the same nodes.*

Observation 26 *A node's key is never changed after its initialization.*

Observation 27 *New nodes are always allocated unmarked.*

Observation 28 *All nodes, excluding the head, are unreachable at the moment of allocation.*

Claim 29 *A marked node is never unmarked.*

Proof: Changes to a node's next field are made only in lines 18, 39, 75, 138, 140, 174. We shall go over them one by one.
Line 18, node.next is initialized as unmarked.
Line 39, head.next is set to unmarked.

Line 75, a CAS that cannot change the mark is performed.
Line 138, a CAS that cannot change the mark is performed.
Line 140, a CAS that cannot change the mark is performed.
Line 174, attemptMark is made to try and set the mark to true.

Claim 30 *A marked node's next field never changes.*

Proof: Changes to node.next field are made only in lines 18, 39, 75, 138, 140, 174. We shall go over them one by one.
Line 18, initialization. The node cannot be marked at this point.
Line 39, head.next is set. The head cannot be marked at this point, since this is executed only in the constructor of the list, and marking has not yet taken place.
Line 75, a CAS is performed that checks the node.next to be unmarked.
Line 138, a CAS is performed that checks the node.next to be unmarked.
Line 140, a CAS is performed that checks the node.next to be unmarked.
Line 174, attemptMark is performed. It is a CAS instruction that never changes the reference of node.next, and can only change the node from unmarked to marked.

Claim 31 *Once a node has become marked, its next field will never change.*

Proof: This follows directly from Claims 29,30

Observation 32 *The search method never touches infant nodes. In particular, a window (Pred, Curr) that is returned from the search method never contains infant nodes.*

This is correct since the search method only traces objects that are reachable (or were once reachable from the head).

Observation 33 *Throughout the code, window(Pred, Curr) instances are created only in the search method. This means that both Pred and Curr in any window instance are never infant nodes.*

We are about to introduce the most important and complicated lemma in the proof. Loosely speaking, this lemma characterizes all the possible physical changes that may be applied to a node. But before introducing the lemma, we need to define those changes.

Definition 34 *Marking, Snipping, Redirection, and Insertion*

Marking: the mark bit of the next field of a reachable node is set (from 0 to 1).
Snipping: the execution of an atomic change (CAS) from the state: A->R->B, when R is marked and A is unmarked and reachable, to A->B, when A is still unmarked.
Redirection: loosely speaking, redirection is the operation of preparing a new node A for insertion to the list. It consists of setting its next pointer to point to what should be its next node, B. Formally, a redirection is an atomic change of a node A's next field to point to a node B such that:
(a) B is not an infant (see Definition 22) at the time the CAS is executed.
(b) B.key $>$ A.key (recall that, by Observation 26, keys do not change during the execution).
(c) A's logical set (see Definition 23) at the time the CAS is executed (and before the CAS assignment takes effect) is a sub-set of B's inclusive logical set (see Definition 24) at the time the CAS is executed.

Insertion: loosely speaking, insertion is the atomic operation that adds a node B into the list by making a reachable node point to it. Formally, insertion is an atomic modification (CAS) of a node A's next field to point to B such that:
(a) A is reachable and unmarked at the time the CAS operation is executed, and also immediately after the CAS assignment takes effect.
(b) B.key > A.key
(c) B is an infant immediately before the CAS (as a result of this CAS, B ceases being an infant).
(d) Immediately before the CAS, A's logical set and B's logical set are identical. (Intuitively speaking, the insertion logically adds B to the list, without making any other logical changes).

Lemma 35 *After the list is initiated, there are only four possible modifications of a node's next field: Marking, Snipping, Redirection and Insertion, as defined in Definition 34. (These four possible changes do not include the allocation of nodes.) Furthermore:*
1) Marking can occur only in line 174, and line 174 may result in Marking or have no effect at all.
2) Insertion can occur only in line 140, and line 140 may result in Insertion, Redirection, or have no effect at all.

Proof: The proof is by induction. Before any modifications are made to any node's next field, it is trivially true that all the modifications were one of the allowed modifications. We shall prove that if all the modifications until a particular moment in time were one of the allowed four defined in Definition 34, then all the modifications made at that moment also fall into that category.
Let T_i be a moment in time, and assume that all modifications of a node's next field before T_i were Marking, Snipping, Redirection or Insertion. We shall prove that all the modifications made at T_i are also one of these four. But before proving it directly, we need several additional claims.

Claim 36 *Before T_i, an infant node cannot be marked.*

Proof: Before T_i, the only changes possible to a node's next field are the four mentioned above. Of these, only marking can result in a node being marked, and marking can only be done on a reachable (and thus non-infant) node.

Claim 37 *Before T_i, a reachable node cannot become unreachable while it is unmarked.*

Proof: Before T_i, the only possible changes to the next field of nodes are Marking, Snipping, Redirection and Insertion; none of them will cause an unmarked node to become unreachable:
Marking doesn't change reachability.
Snipping only snips out a single marked node.
Redirection may only add nodes to the set of unmarked reachable nodes of a given node.
Insertion may only add a node to the set of unmarked reachable nodes of a given node.

Claim 38 *Before T_i, if B is in A's logical set (see Definition 23) at any given moment, then it will remain in A's logical set as long as it is unmarked.*

The proof is by observing that none of the four possible changes to a next field can invalidate this invariant, and is similar to the proof of the previous claim.

Claim 39 *Before T_i, a node may only ever point to a node with a key higher than its own.*

Proof by induction: Before the first execution line after the initialization, the only node pointing to another node is head-> tail, and is thus sorted by the definition of the head key and tail key (Definition 1). By Observation 26, a node's key is never changed. Before T_i, the only possible changes to a node's next field are Marking, Snipping, Redirection and Insertion. Marking doesn't change the pointed node, and thus cannot invalidate the invariant. Snipping, by definition, only snips out a node, and thus, by transitivity, if the invariant holds before snipping, it will hold after it. Redirection and Insertion can, by definition, only change a node's next field to point to a node with a higher key.

Corollary 40 *Before T_i, the list is sorted in a strictly monotonously increasing order, and there are no two reachable nodes with the same key.*

Proof: This follows directly from Claim 39.

Claim 41 *Before T_i, the head is never marked.*

Proof: Changes to a node's next field are only made in lines 18, 39, 75, 138, 140, 174. Looking at these lines, we can see that the only place that a node can become marked is in line 174. In this line, an attempt is made to mark the node that appears as the Curr field in a window. By Observation 33, this window was originally returned from the search method. In the search method, Curr can only be read from the next field of a node. Before T_i, a node can only ever point to a node with a higher key than its own, by Claim 39. By Definition 1, no node can have a key smaller than the head key, so we conclude that before T_i, no node can point to the head, and thus the head cannot be returned as the Curr field in a window by the search method, and thus it cannot be marked.

The following Claim refers to the linearization point of the search method. Loosely speaking, it means that before T_i, the search method works correctly.

Claim 42 *Before T_i, when calling the search(key) method, if the method returns with a valid (not null) window (Pred,Curr), then during the method's execution there was a point (the search linearization point) in which all the following were true:*
(a) Pred.key < key, and Pred was the last node in the list satisfying this condition.
(b) Curr.key >= key, and Curr was the first node in the list satisfying this condition.
(c) Pred was unmarked.
(d) Curr was unmarked.
(e) Pred.next pointed to Curr.

Proof: We start by proving that Pred.key<key. Pred is initialized in line 69 as the head, and by Definition 1, head.key < all possible keys. Pred is later modified only in line 84, but the failure of the condition in line 82 guarantees that the new value of Pred.key will remain lower than the key (recall that, by Observation 26), a node's key never changes so pred.key < key throughout the run. Next, we show that curr.key >= key upon return from the search method. If the search method did not return null, then it must have returned via line 83. The condition in line 82 guarantees that Curr.key >= key. Given that Pred.key<key and Curr.key >= key and since the list is sorted in a strictly monotonously increasing order (by Corollary 40), showing (e), i.e., that Pred.next pointed to Curr, will guarantee the second part of (a), i.e., that Pred was the last node satisfying Pred.key < key, and similarly, the second part of (b), that Curr was the first node satisfying Curr.key >= key. So it remains to show that (e) holds, to conclude that (a) and (b) also hold. We next show that (e), (c), and (d) hold.

The last update of Curr before returning from the search method must happen after the last update of Pred, because whenever Pred is updated, there is always an update to Curr right after. (See Lines 69, 70, and 84 where Pred is modified.) There are three possible cases for when curr was last updated, and for each we will show that (c), (d), and (e) hold:

1. The last update to Curr was in line 70.

Then during the read of Pred's (head) next field, Pred pointed to Curr (e), and Pred was unmarked since, by Claim 41, the head is never marked (c). Now, if the condition in line 73 were true, then Curr would have been updated again (either in line 79 or again in 70) and thus after the last update to Curr it was false, meaning that Curr wasn't marked at line 72, which happened after line 70. Since a marked node is never unmarked (Claim 29), then it was also unmarked during the read of Pred's next field in line 70 (d).

2. The last update to Curr was in line 79.

The condition in line 78 guarantees that line 79 can only be reached if the CAS in line 75 (Snipping) succeeds. That CAS changes pred.next field to point to the value that Curr will receive in line 79, and only succeeds if Pred is unmarked. Thus, if we reached line 79, then at the point immediately after that CAS, Pred.next pointed to Curr eventual value (e), and Pred was unmarked (c). Similarly to the previous case, if this is the last update of Curr, then the loop condition in line 73 checked after this update must be false (otherwise there would be another update), and thus Curr was unmarked during the read of line 80, and since a marked node is never unmarked (Claim 29), then also during the CAS of line 73. (d)

3. The last update to Curr was in line 84.

In line 84 Pred gets the value of Curr, and right after that Curr gets the value of Succ. This Succ value was read either at line 72 or 80, in each case, from Curr.next. So in the execution of line 84, Pred gets the Curr that pointed to the Succ that is now being put into Curr. So during the setting of Succ (line 72 or 80) prior to the last update of Curr in line 84, the eventual Pred pointed to the eventual Curr (e). Also, after the read of Succ (the eventual Curr) either in line 72 or 80, the condition in line 73 is checked, and must be false (otherwise Curr would be updated again), which guarantees that at that point the (eventual) Pred wasn't marked (c). Finally, after the last update of Curr in line 84, curr.next is read again (line 72), and tested again (in line 73) to make sure Curr isn't marked either, and therefore was also not marked at any time before (d).

Claim 43 *Before T_i, a node's next field never points to an infant node.*

(Note that this isn't entirely trivial, since infant means 'has never been reachable from the head', and not 'has never been reachable from any node.) Proof: Let us see that the four possible changes to a node's next field cannot cause a non-reachable node to point to an infant node. (Making a reachable node point to an infant node will simply cause the node to cease being infant, not invalidate the claim). Marking, Snipping and Insertion specifically define that only the next field of a reachable node can be changed, and Redirection specifically defines that the newly pointed node must be a non-infant.

Corollary 44 *Before T_i, an unmarked node that is reachable from any node, is also reachable from the head. (Alternatively, the logical set (Definition 23) of any node is a subset of the set of nodes logically in the list (Definition 19).)*

Proof: This follows from Claims 29,37,43.

Definition 45 *A node's maximal set is the set of all the unmarked reachable nodes with a key greater than its own.*

Claim 46 *Before T_i, Redirection (Definition 34) cannot affect A's logical set (Definition 23) if A's logical set is already maximal prior to the Redirection.*

Proof: By definition, Redirection can only add nodes to a node's logical set. But it is impossible to add nodes to it, if it is already maximal, since by Corollary 44 and Definition 45, a node's logical set is always a subset of its maximal set.

Corollary 47 *An unmarked node that is not an infant is reachable and thus logically in the list.*

Proof: This follows from Claim 37.

Claim 48 *Before T_i, a Redirection of the next field of a node A cannot change the logical set of any non-infant node B.*

Proof: If A is an infant itself, then it cannot be reachable from any node (Claim 43), and thus redirection on its next field can only affect its own logical set, and since A is an infant, this is allowed. If A is marked, then by definition, Redirection cannot be applied on its next field anyway. If A is non-infant and unmarked, then it is reachable (Corollary 47). So, since the list is sorted (Corollary 40), all the unmarked reachable nodes with a key greater than A.key are reachable from A. By definition, Redirection can only increase A's logical set, but there are no keys larger than A's key in the list. Also, redirection cannot be made to point to an infant (i.e., unreachable) node. Thus, redirecting A's next pointer cannot change the logical set of A.

We are now ready to show that any modifications to a node's next field at time T_i are restricted to Marking, Snipping, Redirection, and Insertion, and thus conclude the proof of Lemma 35.
Changes to a node's next field are made only in lines 18, 39, 75, 138, 140, 174. We shall go over them one by one:
Line 18: The allocation of a new node is excluded from the statement of the Lemma.

Line 39: This line is only executed during the list initialization, and is thus also excluded from the Lemma assertion.

Line 75: This instruction line is Snipping, which is inside the search method. For the change to take place, the CAS must succeed. Let us verify that in this case all the Snipping requirements are met:
Pred, Curr, and Succ nodes of the search method are here A, R, and B of the snipping definition. We need to show that the CAS is from state Pred->Curr->Succ when Pred is unmarked and reachable and Curr is marked, to Pred->Succ, when Pred is still unmarked and reachable. The condition in line 73 guarantees that the Curr was marked. Claims 29 and 31 guarantee that once marked, it will remain marked, and that its next field will never change. Thus, if the CAS in line 75 succeeds, we know for certain that before its execution the state was Pred->Curr->Succ (that CAS checks Pred->Curr, and Curr->Succ is guaranteed by Claim 31), that Curr was marked (Claim 29), and that Pred wasn't marked (the CAS verifies this). Also, since the search method never reaches infant nodes (Observation 32) and Pred is unmarked, then Pred is reachable (By Corollary 47). Thus, after the execution, the state is Pred->Succ, Curr is still marked, and Pred is still not marked, and thus also surely reachable (Claim 37). Also note that A is surely reachable: this is true because line 75 is inside the search method, which never reaches infant nodes and A is also unmarked. We conclude that it is a legal Snipping, and thus line 75 can either do nothing or a legal Snipping.

Line 138: This instruction line is Redirection, which is done inside the helpInsert method. Let us see that if the CAS succeeds, then all the redirection requirements are met: this CAS attempts to set the inserted node's (see Definition 8) next field to point to window.curr. Window is the value returned from the search method called in line 110. (This search is for the operations's key.) We need to show:

(a) Window.curr is not an infant. (This is immediate from Observation 33.)
(b) Window.curr.key > operation's key.
(c) Immediately before the CAS, the inserted node's logical set is a subset of the Window.curr inclusive logical set (Definitions 23, 24).

(a) is immediate, as stated above.
(b) The search method linearization claim (Claim 42) guarantees that Window.curr.key >= the operation's key. The condition in line 113 guarantees that window.curr.key != operation's key, otherwise line 138 wouldn't have been reached, so window.curr.key must be larger than the operation's key.
(c) Now, line 138 is trying by a CAS to replace the inserted node's next with window.curr, and it compares the inserted node's next to the value node_next read in line 109, before the search method that returned the window is called. If node_next is null and the CAS succeeds, then the set of unmarked reachable nodes from the inserted node immediately before the CAS is the empty set, trivially fulfilling the condition. So we shall assume node_next is not null. By Claim 43, node_next is also not an infant node. By Claim 42 (the search method linearization claim), there was a point, the search linearization point, at which Window.curr was the first unmarked reachable node in the list, with a key >= the operation's key.
We will prove (c) by claiming following: Before T_i, suppose there is a point in time in which two nodes A and B satisfy that there exists a key K such that:
1. Neither A nor B are infants.
2. Both A.key and B.key >= K.
3. A is the first unmarked reachable node in the list satisfying A.key >= K.
Then B's logical set will always be a subset of A's logical set (as long as none of them is reclaimed). This is true by induction: at the search linearization point, the set of unmarked nodes reachable from A is the maximum possible set for nodes with a key greater than K, and thus, B's logical set is surely a subset of this set. The four possible changes to a node's next field before T_i:
Marking: Only affects the mark of the marked node, and not the reachability of any node from any node. This clearly can't produce an unmarked reachable node from B which is not reachable from A.
Snipping: Only snips out a marked node, and doesn't affect reachability of any unmarked nodes.
Redirection: Since neither A nor B are infant, by Claim 48 Redirection cannot change the set of unmarked reachable nodes from them.
Insertion: Two Cases:
1. At the search linearization point, B is marked; thus, its next field cannot be changed (by Claim 31), and in particular cannot be changed by insertion. So the only way to add a node to B's logical set is by insertion (changing the next field of one of the unmarked nodes reachable from B), but all these nodes are also reachable from A, and thus this will also add this node to the set of unmarked nodes reachable from A.
2. At the search linearization point, B is unmarked. Since B is not an infant, it is reachable from the head, and since A was the first unmarked node with a key greater than K in the list, then B was reachable from A. Since while B is unmarked it will remain reachable from A (by Claim 38), then also changing B's next field by insertion directly will add the new node to the set of nodes reachable from A. Once B is marked and its logical set is still a subset of A's logical set, we are back to case 1.

Line 140: This line is inside the helpInsert method as well. The instruction in it is normally an Insertion, and can sometimes be a (private case of) Redirection. Let us see that if the CAS succeeds, either all Insertion or all Redirection requirements are met.
This line contains a CAS that changes window.pred to point to the inserted node. Window was the search

result for the operation's key in line 110. Note that this CAS also checks for version. To prove a valid Insertion, we need to show that all the following are true:
(a) window.pred is reachable and unmarked immediately before the CAS, and also immediately after the CAS.
(b) The operation's key $>$ window.pred.key
(c) The inserted node is an infant immediately before the CAS.
(d) Immediately before the CAS, window.pred's logical set is identical to the inserted node's logical set.
Note that if (c) is not fulfilled, then this is a legal Redirection, so we will focus on proving (a), (b), (d).
(a) window.pred is clearly not an infant since it was returned by the search method. The CAS makes sure it is unmarked and, by Corollary 45, reachable.
(b) This is immediate from Claim 42 since window is the result of the search for the operation's key in line 110.
(d) In the CAS of line 140, we compare window.pred.next pointed node to the one previously read from the inserted node's next field. If the inserted node's next field hasn't changed between its reading and the CAS, then immediately before the CAS both the inserted node's next field and the window.pred.next point to the same node, so obviously the set of unmarked nodes reachable from both is identical. If the node pointed by the inserted node's next field has changed, then the set of unmarked nodes reachable from it could only have grown, since before T_i all changes to a node's next field can only add to the set of unmarked nodes reachable from it. However, window.pred is unmarked and reachable, its logical set is the maximum, and thus the two sets must still be equal.
Now, recall that we don't need (and can't) prove item (c), since line 140 can be either Insertion or Redirection. Note that if it is a Redirection, it is a futile one by Claim 48 (meaning that it doesn't change the logical set of any node). In general, all redirections changing the next field of a non-infant node are futile (unwanted, but harmless).

Line 174: This line contains the Marking instruction, and it is inside the helpDelete method. In this line we attempt to mark the next field of a node stored in the searchResult.curr field of a state entry. This field (the searchResult of the OpDesc class) can only be written to a value different than null in line 168. In line 168 this field(the searchResult of the OpDesc) receives the result of a search method. Thus, since the search method doesn't return in its window nodes that are infants (Observation 32), we know for sure that this is an attempt to mark a non-infant node. If it is already marked, then this line cannot possibly make any difference. If it is not, then by Corollary 47 this node is reachable, and thus this is a CAS to mark the next field of a reachable node, and thus a legal marking.

To conclude, we have seen that if all modifications to a node's next field before T_i are due to Marking, Snipping, Redirection or Insertion, then all modifications at T_i also belong to one of these categories, and we have finished proving Lemma 35.

Corollary 49 *All the claims used during the proof of Lemma 35 hold throughout the run (And not only 'before T_i').*

Proof: A direct result of proving Lemma 35. For the rest of the proof we shall treat those claims in their general form.

Corollary 50 *Insertion and Marking (as defined in Definition 34) are the only logical changes to the list, when Insertion adds exactly one node (the inserted node) into the list, and Marking removes exactly one node (the marked node) from the list.*

Proof: This is a direct result of Observation 26 (a node's key is never changed), and of Lemma 35.
Snipping: Only changes the reachability of a marked node, and thus makes no logical changes to the list.
Redirection: According to Claim 48, it is clear that Redirection cannot make logical changes to the list.
Marking: Since we mark a reachable node, it is clear this takes this node logically out of the list.
Insertion: By definition Insertion inserts a previously infant node, while making no other changes to the set of unmarked reachable nodes.

Notes about parallel CASes that happen at exactly the same time:
a. A node cannot be marked more than once even at the same moment, since this is done by a CAS on its next field. So it is safe to assume that each marking has the effect of logically removing a distinct node from the list.
b. The same node cannot be inserted more than once even at the same moment, since at any given moment the node has only one possible place in the list, and thus the Insertion CAS is on a specific field. So it is safe to assume that each Insertion has the effect of logically adding a distinct node to the list.
c. It can easily be shown that the same node cannot be marked and inserted at the same moment (it must be inserted before it is marked), but this is not necessary for the point of our discussion.

Claim 51 *An infant node can only cease being infant via its Insertion.*

Proof: By Claim 43, no node can ever point to an infant node, and thus, the first time a node is pointed to is when it stops being infant, meaning that the first time it is pointed to it is from a node reachable from the head. This first time it is pointed to must be when one of the four possible changes occurs:
Marking doesn't change the pointed node, and thus the change that causes the node to cease being infant cannot be Marking.
Snipping by definition changes a node's next field to point to a node that was already pointed to by another node before, and thus it cannot be the first time a node is pointed to.
Redirection by definition changes a node's next field to point to a node that is not an infant, and thus it cannot be the change that causes a node to cease being infant.
So the first time a node is pointed to by the next field of any node can only be when an Insertion occurs. This Insertion makes the node reachable from the head, and thus no longer infant.

Claim 52 *An infant node is never marked.*

Proof: Of the four possible instructions that modify a node's next field, the only one modifying the mark of a node is Marking, which is done on a reachable (and thus non-infant) node by definition.

Observation 53 *The fields of an Operation Descriptor (opDesc in the code) are final. That is, they are never changed after initialization.*

Claim 54 *After initialization, a non-pending state (success, failure or determine_delete) for a thread cannot be altered by any thread other than itself.*

Proof: The state array changes in lines 49, 56, 117, 123,130,136,142,163,169,179.
Lines 49 and 56: These lines are inside the insert method and the delete method. Both methods are only called by the operation owner thread to initiate the operation. By definition of legal operations (Definition 4), both are only called with the tid of the thread owner. Thus, both can only alter the state of the running thread.
Lines 117,123,130 and 136 are inside the helpInsert method. They contain a CAS that only succeeds if

the operation is the one read in line 105. The condition in line 106 guarantees that it is a pending (insert) operation, and Observation 53 guarantees that it will remain so.
Line 142: This line contains a CAS and only succeeds if the old operation is the one created in line 133. This operation is a pending (insert) state.
Lines 163,169,179: These lines are inside the helpDelete method, and they use a CAS that compares the given parameter to the value read at line 151. The condition in line 152 guarantees that the state in that case is pending (either search_delete or execute_delete).

Claim 55 *Each thread can execute at most one operation with a given phase number. This means that a pair consisting of a threadID (Definition 2) and a phase number (Definition 5) uniquely identifies an operation.*

Proof: By definition of legal operations (Definition 4), the insert and delete methods, which initiate operations, can only be executed with a threadID matching the thread that runs them. So for any given threadID, the operations are executed sequentially, one after the other. When a thread calls the maxPhase method twice in succession, then this method is guaranteed to return two different phaseIDs, since each call increases the maxPhase integer by at least one during the run of each maxPhase. Note that if the CAS that increases this integer fails, then it must hold that another thread has incremented this number, as all modifications to this number (except for its initialization) are increments.

Claim 56 *A non-pending operation (Definition 14) cannot revert to pending.*

Proof: We have already seen in the proof of the previous claim that a non-pending operation can only be changed inside the insert or delete methods (which are only executed by the owner thread and not by helper threads). But these methods never change an operation's state to non-pending; they only create a new state operation, which will have a different phaseID.

Claim 57 *A search method might only return a null window if its operation (Definition 6) is no longer pending.*

Proof: Immediate from the condition in line 76.

A.4 The Insert Operation

Recall first the definition of Insertion (Definition 34). Note the difference between an Insertion, which is a single CAS that inserts a node, and an insert operation (Definitions 3,6), which consists several methods that may be called by several different threads, and is initiated when the owner thread of the operation calls the insert method (Definition 7). Also recall that a successful operation is one for which the method that initiated it returned true (Definition 9). In this part of the proof, we want to establish a connection between Insertions and insert operations. In particular, we will show a one-to-one correspondence between Insertions and successful insert operations. We will use the (tid, phase) pair as a connector between them. First, we shall define four functions.

Definition 58 *The Insert Functions - A,B,C,D*

Function A: Insertion -> (tid, phase) pair.
Matches each Insertion to the (tid, phase) pair that were that parameters for the helpInsert method that the insertion was a part of. (Recall that, by Lemma 35, Insertions may only occur in line 140).

Function B: insert operation -> (tid, phase) pair.

Matches each insert operation to a (tid, phase) pair, such that the tid is that of the owner thread (which by definition of legal operations is also the tid parameter of the insert method), and the phase is the number returned from the maxPhase() method invoked inside the insert method that initiated this operation.

Function C: Insertion -> insert operation. C(x) = B^{-1}(A(x)).

Function D: insert operation -> Insertion or NULL.
D(y) = C^{-1}(y) if defined
or NULL otherwise

Claim 59 *A thread whose state is other than insert can only reach the insert state in an insert method called by the same thread.*

Proof:
Line 49: This line is indeed inside the insert method. It changes the state for the given tid and, by definition of legal operations (Definition 4) the given tid must be that of the running thread.
Line 136: This line contains a CAS that can only succeed if the current state of the operation owner is the one read in line 105, and the condition in line 106 guarantees that the state of the operation owner is already an insert.
The rest of the lines never attempt to write an operation with the insert state.

Claim 60 *Function A (Definition 58) is an injective function.*

Proof : Let x be an insertion. By Lemma 35, we know that Insertion can only take place in a successful CAS in line 140, so we know that x took place in line 140. In this line an attempt is made to insert the node read in line 108 from the variable op into the list. The condition in line 106 guarantees that this op has the same (tid, phase) pair as A(x). Claim 55 guarantees that there is no other operation with the same (tid, phase) pair. The node read in line 108 is the operation's node (Definition 8) allocated in the insert method, so if another Insertion x' exists such that A(x) = A(x'), then both Insertions are inserting the same node. But by definition of Insertion, it is inserting an infant node into the list, and immediately after that the node is no longer infant. So two Insertions of the same node cannot happen at two different times. Two Insertions of the same node also cannot happen at the same moment because the list is sorted (Corollary 40), and thus at a single moment a node can only be inserted into a specific place in the list. So two simultaneous insertions of the same node must execute a CAS on the same predecessor for this node, which cannot be done at the same time. We conclude that each node can only be inserted (via Insertion) once, and thus two distinct Insertions must insert two distinct nodes, and thus have two distinct (tid, phase) pairs.

Claim 61 *Function B is an injective function.*

This follows directly, and is a private case of Claim 55.

Claim 62 *Function C is defined for every insertion, and is injective.*

Proof: C(x) is defined as B^{-1}(A(x)). We should first note that B^{-1} is well defined. We know this from Claim 61 that B is injective. B^{-1} is thus also injective, and By Claim 60 A is also injective. So C is injective as a composition of injective functions. We still need to show that C is defined for every Insertion x. A is defined for every Insertion x, but B^{-1} is most certainly not defined for every (tid, phase) pair. However, B is defined for all insert operations, and thus B^{-1} is defined for all (tid, phase) pairs that match an insert operation. This is all we need, since for every insertion x, A(x) is indeed a (tid, phase) pair that matches an insert operation.

This is true since Insertion only happens at the helpInsert method, and helpInsert is only called in line 94, when the condition in line 93 guarantees that the (tid, phase) pair matches a state of a (pending) insert. Claim 59 guarantees that this can only be the case if the (tid, phase) pair matches an insert operation.

Claim 63 *Function D is well defined.*

Proof: This is true since C is an injective function (Claim 62).

Claim 64 *A helpInsert method will not be finished while its operation is pending.*

Proof : The HelpInsert method is comprised of an infinite loop and can only be ended in one of the following lines:
107: The condition in line 106 guarantees this can only happen if the insert operation is no longer pending.
112: The condition in line 111 guarantees this can only happen if the search method returned null, which can only happen if the operation (given to it) is no longer pending (Claim 57)
118: The condition in line 117 guarantees this can only happen if the state was successfully changed to non-pending in the same line (117).
124: The condition in line 123 guarantees this can only happen if the state was successfully changed to non-pending in the same line (123).
131: The condition in line 130 guarantees this can only happen if the state was successfully changed to non-pending in the same line (130).
143: The condition in line 142 guarantees this can only happen if the state was successfully changed to non-pending in the same line (142).

Corollary 65 *A help(phase) method will not finish while a pending insert operation with the same phase number exists.*

This is an immediate conclusion by the structure of the help method (it calls the helpInsert method) and Claim 64.

Corollary 66 *An insert method will not be finished while the operation initiated by it is still in a pending state.*

This is an immediate conclusion by the structure of the insert method (it calls the help method) method and Corollary 66.

Claim 67 *A thread in a pending inert state can only reach a different state in the helpInsert method.*

Proof: According to Corollary 66, the insert and delete methods cannot change this state since while the insert is pending the owner thread hasn't yet finished the insert operation. The helpDelete method cannot change this state since every change of a state in it is by a CAS that ensures it only changes a pending delete (search_delete or execute_delete) state. The rest of the lines that change a state are only inside the helpInsert method.

Claim 68 *A thread's state can only be changed into success in the helpInsert method, and only if the insert's operation's node is no longer an infant.*

Proof: A success state can only be written (throughout the code) in the helpInsert method. We will go over each of the lines that change a state into success and see that they can only be reached if the operation's node is no longer an infant. A success state can be written in the following lines:

Line 117: The condition in line 114 guarantees that this line can only be reached in one of two cases:
1. The operation's node is marked, and thus, by Claim 52 is not an infant.
2. The node was returned inside the window that the search method returned, and thus is not an infant (Observation 32).
Line 130: The condition in line 128 guarantees this line can only be reached if the inserted node is marked and thus non-infant (Claim 52).
line 142: The condition in line 140 guarantees this line can only be reached if the CAS in line 140 succeeded. By Lemma 35, if this CAS succeeds it is either an Insertion of the inserted node (which thus ceases being an infant node), or a Redirection, and thus the inserted node is already not an infant, by definition of Redirection (Definition 34).

Claim 69 *For each successful insert operation (recall successful means returned true) denoted y, D(y) is not NULL.*

Proof : Another way to formulate this claim is that for each insert method that returned true, an insertion took place during the insert operation. The structure of the insert method guarantees that it returns true if and only if the state of the owner will be changed into success. By Claim 68, it means that the operation's node is no longer an infant. By Claim 51, this implies that a corresponding Insertion took place.

Claim 70 *The key added to the list as a result of an Insertion (the Insertion is denoted x) is identical to the key given as a parameter to the insert method that initiated C(x).*

Proof: The key added to the list as a result of an Insertion x is the key that is on the inserted node, which is the operation's node read from the state in line 108. The operation's node for an insert operation is created in line 47, with the key given to the insert method.

Claim 71 *Immediately before insertion, the key to be inserted is not in the list.*

Proof: By Corollary 40 a valid insert retains the strict monotonicity of the list, and therefore the key cannot exist in the list during Insertions.

Claim 72 *For an unsuccessful insert operation (meaning that the insert method that initiated it returned false), denoted y, D(y) = NULL.*

Proof: An unsuccessful insert operation can only happen if the pending insert operation changed to something other than success. By Claim 67, this can only happen in the helpInsert method. By Corollary 66, the insert operation will not be finished while the state is still pending, and if the state changed to success, the operation will not fail. Since for an insert operation y, d(y) can only be an Insertion in which the inserted node is the operation's node, it is enough to show that the operation's node in a failing operation can never be inserted. A word of caution: it is not enough to show that when the insert operation ceases to be in a pending state, the node of that operation is still an infant. We also need to show that it cannot possibly be inserted later by other threads currently inside a helpInsert method for the same operation. Let us go over the changes of the state inside the helpInsert method. For each one, we shall see that one (and only one) of the following holds:
1. It changes the state to success, and thus the operation will result in being successful, not relevant here (lines 117, 130, 142).
2. It changes the state but to a state that is still a pending insert, and thus the insert method must still be in progress and cannot (yet) return false (line 136).
3. It changes the state to failure, but we can show that the node that belongs to the operation is certainly an

infant and also cannot be inserted later(line 123).
In line 110 a search is done for the operation's key. The condition in line 113 guarantees that line 123 can only be reached if the search method found a node with the same key. The condition in line 114 guarantees that line 123 can only be reached if that node is not the operation's node, and also that the operation's node is not marked (at least not at this time). So, during the search of line 110 we know that there was a point, the search linearization point, at which:
1. The operation's node was not in the list.
2. The operation's node was not marked. (Claim 29)
3. A different node with the same key was in the list (we will call it the hindering node).
1 and 2 together mean that the operation's node was infant at the time (using Claims 37, and also 29 again). Now, if an Insertion of the operation's node is to take place, then it must be in a concurrent thread running helpInsert of the same operation after reading op in line 105. (If it did not yet read op, it will find the operation no longer pending, and will return from the helpInsert method.) Now, before this concurrent thread gets to the Insertion (line 140), it must also change the state in a (successful) CAS in line 136. There are two possible cases:
1. The CAS of the failure in line 123 takes place before the concurrent CAS in line 136.
But then the CAS in line 136 cannot be successful, because it will compare the current state to the obsolete state read in line 105.
2. The CAS of the failure in line 123 takes place after the concurrent CAS in line 136.
Then, in order for the CAS of the failure in line 123 to succeed, it must have read the old state after the CAS of the concurrent thread in line 136. This also means that it reached line 110, and the search point, only after the concurrent thread read the version of its window.pred in line 133. Now, if at the time the concurrent thread read the version in line 133, the hindering node was already in the list, then the window.pred could not point past it (since the keys are sorted), and thus, the CAS of the Insertion in line 142 must have failed. It can only succeed if the pointer in window.pred hasn't changed, and also it points to a node equal to one read from the operation's node next field, which must have a key greater than the (identical) key of the hindering node and the operation's node. If at the time the concurrent thread read the version in line 133 the hindering node was not yet in the list, and window.pred.next pointed to a node with a key greater than the operation's key, then the hindering node must be later inserted into the list, and this must change the next field of the window.pred, advancing the version, and ensuring that the insertion CAS in line 140 cannot succeed (indeed, this is the reason why we needed the version in the first place).

Claim 73 *For every Insertion x, the CAS operation that caused it occured during the execution time of the insert method that initiated C(x).*

(This claim is necessary to show that this Insertion is a legal linearization point for the insert method.)
Proof: For the insert operation y = C(x), we know that D(y) = x, which means by Claim 72 that y was a successful insert, and returned true. That can only happen if the state of the owner thread was success, which by Claim 68, can only happen if the Insertion x has already taken place. The Insertion cannot take place before the insert operation C(x) starts, because the (tid, phase) pair of a state is created only at the insert method that initiates the operation.

Claim 74 *An insert operation can only fail if during a search method belonging to that operation, another node with the same key was logically in the list (at the linearization point of that search method).*

Proof: We have seen in the Proof of Claim 72 that an unsuccessful insert operation can only be the result of a CAS changing the state of the operation to failure in line 123. The combined conditions of lines 113 and

114 guarantee that this line can only be reached if an appropriate hindering node, with the same key, was returned from the search method of that operation that was called in line 110.

Lemma 75 *An insert method that finished 'works correctly', meaning that one of the following has happened:*

* *It returned true, inserted the key, and at the point of linearization no other node with the same key existed.*
* *It returned false, made no logical changes to the list, and at the point of linearization another node with the same key existed.*

Proof: If the insert method that initiated an operation denoted y returned true (i.e., the operation was successful), then by Claim 69 D(y) is a corresponding Insertion, which happened during the execution time of the insert (Claim 73), which inserted a key that was not in the list at that time (by Claim 71). If the insert method returned false, then by Claim 73 it corresponds to no insertion, and by Claim 74, another node with the same key existed at the linearization point of a search method that belonged to this operation, and this point is also defined as the linearization point of the insert operation.

A.5 The Delete Operation

Recall first the definition of Marking (Definition 34), and the definition of a delete operation (Definitions 3,6). Also recall that a successful operation is one for which the method that initiated it returned true (Definition 9).

Claim 76 *A thread's state can only be changed into search_delete in the delete method called by the same thread.*

Proof: Line 56 is inside the delete method, and indeed changes a thread's state into a search_delete. By the definition of legal operations (Definition 4) this can only be called by the same thread. The other lines that change a thread's state (49, 117, 123,130,136,142,163,169,179) never attempt to make the state a search_delete.

Claim 77 *A thread can only reach the execute_delete state directly from the search_delete state.*

Proof: An attempt to set a state to execute_delete is made only in line 169 using a CAS. The condition in line 156 guarantees that this CAS may only succeed if the value it is compared to (the previous state) is a search_delete.

Claim 78 *A delete method will not finish while the operation it belongs to is pending.*

Proof: The delete method is constructed so that it publishes a (pending) delete (using the search_delete state), and then calls the help method. The help method loops through the state array. If by the time it reads the state of the owner thread of the delete operation it is no longer pending, there is nothing left to prove. If it is still pending, it will call the helpDelete method. If so, we can now refer to the helpDelete method, which was called as part of this delete operation by the operation owner. This helpDelete method is constructed by an infinite loop that may only exit at one of the following lines:
line 154: In which case the condition in lines 152-153 guarantees that the operation is no longer pending. (By Claim 56 it cannot return to a pending state.)

line 164: In which case the condition in line 163 guarantees the operation is changed to no longer being pending.
line 180: The condition in line 172 guarantees that line 180 can only be reached if the operation was at state execute_delete. There are two possible cases for this.
1. When the helpDelete method called by the operation owner reached line 179, the CAS succeeded, and thus in line 180 the operation is no longer pending.
2. When the helpDelete method called by the operation owner reached line 179, the CAS didn't succeed. This can only happen if some other thread changed the state. But a different thread could not have done it in the insert or delete method, since those can only be called by the operation owner (by definition of legal operations). All other changes to a state are by means of a CAS. The only one that can possibly change an execute_delete state is the one in line 179, which would have made the state determine_delete and no longer pending. Other lines cannot be reached if the value that is compared to is an operation with a state of execute_delete. (In other words, there is no need to check that the CAS in line 179 succeeded, because it can only fail if another thread already executed the same CAS.)

Claim 79 *An execute_delete can only be changed into a determine_delete state, and only in a CAS in line 179.*

This is an immediate result of the proof of the previous claim. We shall briefly reiterate the relevant parts. An execute_delete state cannot be changed inside the delete or insert methods, since in these methods a thread only changes its own state (by definition of legal operations), but, according to the previous claim, the owner thread will not finish the delete method that initiated this operation while the operation is pending. The remaining changes to a state are executed by a CAS, and the only CAS that compares the previous value to a state with execute_delete is the one in line 179, which changes it into a determine_delete.

Definition 80 *Possible routes of a delete operation's state.*

Route 1: published search_delete in line 56 -> CAS into failure in line 163.
Route 2: published search_delete in line 56 -> CAS into execute_delete in line 169 -> CAS into determine_delete in line 179.

Claim 81 *The state of any delete operation from publishing until it is not pending can only follow one of the two routes in Definition 80.*

Proof:
Fact: A pending delete operation's state cannot be changed outside the helpDelete method.
This is because inside the helpInsert there is a CAS that checks that the operation is a pending insert operation, and the changes in the delete and insert methods cannot be made since the operation is still pending.
Using this fact, we can just focus on the changes inside the helpDelete method.
Line 163 is a CAS that leads to failure, and the condition in line 156 guarantees the previous state is search_delete.
Line 169 is a CAS that leads to execute_delete, and the condition in line 156 guarantees the previous state is search_delete.
Line 179 is a CAS that leads to determine_delete, and the condition in line 172 guarantees the previous state is execute_delete.

We shall now define two functions that will correlate between delete operations that followed route 2, and

Marking, as defined in Definition 34. Using a process similar to the one we used in our proof of the insert operation, we wish to prove a one-to-one correspondence between successful delete operations and Markings.

Definition 82 *The Delete Functions A,B*

Function A: Delete Operations that followed route 2 -> Marking.
For a delete operation that followed route 2 (as defined in Definition 80), denoted y, the operation was at some point in a state of execute_delete. At that point, there was a window stored in the searchResult field of that operation descriptor. (The condition in line 158 guarantees that a state of execute_delete always contains a valid (not null) window in the searchResult field.) We say that A(y) is the Marking of the node that was stored in searchResult.Curr. (We shall prove immediately that this defines a single Marking for every delete operation that followed route 2.)

Function B : Marking -> Delete Operation that followed route 2.
We say for a marking, denoted x, that : B(x) = y if and only if both of the following are true:
1. A(y) = x.
2. y was a successful delete operation (returned true).
We shall prove soon that function B is well defined and injective.

Claim 83 *A is defined for every delete operation that reached the execute_delete state, denoted y, and the Marking A(y) always takes place during the run of the operation y. (It always takes place between the invocation of the delete method that started it, and the end of the same delete method.) Furthermore, A(y) is a Marking of a node that has the same key as was given as a parameter to the delete method that initiated the (y) operation.*

Proof:
Part One: A(y) matches every delete operation that reached the execute_delete state to at least one Marking that executed during its run:
By Claim 81, execute_delete can only be changed in a CAS in line 179. The condition in line 174 guarantees that this CAS can only be reached if the node found at the op.searchResult.curr is marked. The searchResult window was returned from a search method called in line 157. By Claim 42, there was a point in time that this node was unmarked, and this certainly happened during this delete operation. The condition in line 160 guarantees that the execute_delete state would only have been reached if the searchResult.curr.key equalled the operation's key.
Part Two: A(y) matches every delete operation to no more than one Marking:
By Claim 81, execute_delete can only be reached once (with a specific operation descriptor) in a delete operation. The Marking can only be done on the op.searchResult.curr. This is only a single node, and each node cannot be marked more than once (Claim 29). Thus execute_delete correlates to no more than one Marking.

Claim 84 *Function B matches each Marking, denoted x, to a single and distinct delete operation. (By the definition of Function B (Definition 82), it also follows that this delete operation returned true.)*

Proof: Each delete operation that reached the execute_delete state matches the Marking of the node found in the op.searchResult.curr, and its state can only be changed into determine_delete (Claims 81,83). In the delete method belonging to this operation, after the help method is done the operation is no longer pending, and thus it must have reached the determine_delete at that point. Then all the delete operations that reached

execute_delete can only be exited in line 61. Line 61 contains a CAS on the op.searchResult.curr.d, trying to change it from false to true. Since this field is initiated as false, and is never modified apart from this line, then no more than one operation can succeed on this CAS, but if at least one of them tried, then at least (and exactly) one must succeed.

Claim 85 *For a successful delete operation denoted y, there exists a Marking x of a node with the same key as the operation's key (see Definition 8), satisfying B(x) = y.*

Proof: A successful delete operation can only go by route 2, since route 1 always ends with a failure by Definition 80. By Claim 83, we conclude that A(y) = x is a Marking of a node with the operation's key. Since the delete operation was successful, and A(y) = x, then by definition of function B, B(x) = y, and by Claim 84, B is well defined.

Claim 86 *A delete operation that followed route 1 made no logical changes to the list.*

Insertion can only take place inside the helpInsert method, which cannot be reached in a delete operation. Marking can take place only in line 174 (By Lemma 35), but the condition in line 172 guarantees that this line can only be reached if the state of the operation was at some point execute_delete, which means this delete operation followed route 2.

Claim 87 *A delete operation can only follow route 1 if at some point during its execution there is no node in the list with a key equal to the operation's key.*

Proof: Route 1 requires a CAS in line 163. The condition in line 160 guarantees that line 163 can only be reached if the search method called in line 157 returned a window with window.Curr.key != the operation's key. By Claim 42, there was a point, the search linearization point, when this node was the first node in the list satisfying that its key >= the operation's key, meaning that the operation's key was not in the list at that time. (This search linearization point is also the linearization point for a delete operation that followed route 1.)

Lemma 88 *A delete method that finished 'works correctly', meaning that one of the following has happened:*

- *It returned true, and during the operation a reachable node with a corresponding key was marked, and this Marking, denoted x, satisfies B(x) = y.*
- *It returned false. During the operation a node with a corresponding key was marked, but this Marking, denoted x, doesn't satisfy B(x) = y (and also no other Marking satisfies that condition).*
- *It returned false, without making any logical changes to the list, and during its run there was a moment in which the operation's key wasn't logically in the list.*

Proof: If the delete method returned from line 61, then by the condition in line 59 we know that it finished in the determine_delete state, meaning it followed route 2. If the CAS in line 61 succeeded, then the method returned true, and by Claim 85, there exists a Marking x satisfying B(x) = y as required. This is case 1. If the CAS in line 61 failed, then no Marking x can satisfy B(x) = y since by definition of function B (Definition 82) it can only match Markings to successful delete operations. This is case 2. If the method did not return from line 61, then it returned false, and by the condition in line 60, we know it followed route 1. Then by Claims 86 and 87, it made no logical changes to the list, and during its run there was the linearization point in which the operation's key wasn't in the list.

A.6 Wait-Freedom

Definition 89 *The pending time of an operation is the time interval in which the operation is pending.*

Claim 90 *The number of logical changes to the list at a given interval of time is bounded by the number of (delete and insert) operations that were pending during that interval.*

Proof: Recall by Corollary 50 that Insertion and Marking are the only logical changes to the list. Claim 62 matches every Insertion to a distinct insert operation, and Claim 73 guarantees that this Insertion happened during the execution of the insert operation. Claims 83 and 84 match every Marking to a distinct delete operation, and guarantee the marking happened during it. We conclude that, in a given time interval, every logical change to the list, be it Marking or Insertion, is matched in a one-to-one correspondence to a distinct operation that happened (at least partially) during this time interval, and thus the number of logical changes is bounded by the operations.

Claim 91 *The number of Redirections (as defined in Definition 34) at a given interval of time is bounded by [the number of insert operations that were pending (at least partially) at the time interval] * [Logical changes to the list linearized at that time interval + 1] * 2*

Proof: Redirections can result from a CAS either in line 138 or 140, both in the helpInsert method. For a given insert operation, and a given logical state of the list, the new value in both of these CASes is uniquely defined: In line 138, a pointer to the first node in the list with a key larger than its own, and in line 140, a pointer operation's node. The logical states that the list can be in an interval are: its initial state + another state for each logical change. Hence the total number of logical states the list can be in a given interval is the number of logical changes to the list linearized at that time interval + 1. We multiply by two because the Redirection can happen at either line 138 or 140. The last missing argument is that a different Redirections set exists for every operation's node; hence we also multiply by the number of insert operations.

Claim 92 *The number of Snippings (as defined in Definition 34) at a given interval of time is bounded by [Overall marked nodes that existed at some point during that time interval] * [Insertions + Redirections that happened during that interval of time + 1]*

Proof: By Definition 34, only reachable marked nodes can be snipped. Once a node is marked it is no longer reachable and thus cannot be snipped again, that is, unless it becomes reachable again. A node can become reachable again by Insertion or a Redirection. (Marking doesn't affect reachability and Snipping only makes a single reachable node unreachable). So any marked node can be snipped at most 1 + Number of Insertions + Number of Redirections.

Claim 93 *The number of successful CASes performed on nodes at any interval of time is bounded by the Insertions + Markings + Redirections + Snippings performed at that interval, and thus bounded.*

Proof: By Lemma 35, all the changes to a node's next field are either Markings, Snippings, Insertions, or Redirections. We have bounded all of those groups in the previous claims of this subsection, and thus the total number of successful CASes is bounded.

Claim 94 *Each CAS on the state array belongs to a specific operation.*

CASes on the state array are done only in the helpInsert and helpDelete methods. By Definition 6, each instance of these methods belongs to a specific operation.

Claim 95 *The number of successful CASes on the state array belonging to any delete operation is bounded by a constant of 2.*

By Claim 81 a delete operation may have either one successful CAS (from search_delete to failure) or two successful CASes (from search_delete to execute_delete to determine_delete).

Claim 96 *The number of operations that have a pending time with an overlap to the pending time of any given operation is bounded by twice the overall number of threads in the system.*

Intuitively, this is the outcome of the help mechanism, which basically guarantees that a thread will not move on to subsequent operations before helping a concurrent operation that began before its last operation. Proof: The structure of the maxPhase method guarantees that for two non-concurrent executions of it, the later one will receive a larger phase number. For a given operation, at the moment it becomes pending, any other thread is pending on no more than one operation. It can later begin a new operation, but this new operation will have a larger phase number. Each operation (be it an insert or a delete operation) calls the help method. The help method structure guarantees that it will not exit while there is a pending operation with a smaller phase number. So no thread can start a third concurrent operation while the given operation is still pending.

Claim 97 *The number of physical changes (i.e., successful CASes) on the list that can occur during the pending time of any given operation is bounded.*

Proof: By Claim 96, the number of other operations that can be pending while the given operation is pending is bounded. Thus, the number of physical changes that can happen during this time is bounded by Claims 90,91,92,93.

Claim 98 *The number of successful CASes on the state array belonging to an insert operation is bounded.*

CASes on the state array that are performed during an insert operation are only performed in the helpInsert method, and all of them check that the previous state is a pending insert state. Thus, once a CAS successfully changes the operation to something other than a pending insert, no more CASes are possible inside the helpInsert method. Thus, the only possible CAS that has the potential of unbounded repetition is the one in line 136. After a thread succeeds in that CAS, it will not attempt it again before it attempts the CAS in line 140. If it fails the CAS in line 140, it must be due to a physical change to a node's next field that was made since the (linearization point of the) search method called by in line 110, but that may only happen a limited number of times, by Claim 97. Thus, there is only a bounded number of times that the CAS in 136 can succeed in that insert operation, until the CAS in line 140 of that insert operation succeeds (at least once) as well. After the CAS in line 140 succeeds, the operation's node has already been inserted to the list. It cannot become unreachable while it is unmarked (Claim 37). Thus, after that point, each thread that restarts the loop of lines 104-146 will not reach line 136 again, because either the condition in line 113 or the one in line 128 will be true, and the method will exit.

Claim 99 *All the methods in the code will exit in a bounded number of steps.*

Proof: We shall go over all the methods one by one.
All constructors are just field initializations that contain no loops or conditions, and thus will be finished in a small number of steps.
The maxPhase method doesn't contain loops or conditions, and will thus finish after a small number of steps. (Note: it doesn't check the condition of the CAS, and will exit even if the CAS fails.)

The search method is bounded since it searches a specific key and only goes forward in the list, so it must reach it (or beyond it) after a bounded number of times and thus exit in line 83. (This is because the tail always holds a key larger than all other possible keys by definition 1, so there is at least one key that answers the condition in line 83.) The only possibility for a search method to go backwards in the list is if the condition in line 78 returns true. For this to happen, the CAS in line 75 must have failed, which may only happen a bounded number of times while the operation is pending. If the operation is no longer pending, the condition in line 76 guarantees that the search method will exit.
The helpDelete and helpInsert methods call the search method, which is bounded. Other than that, they might only enter another iteration of a loop because of changes that were made to the list or state, but these changes are bounded by Claims 97,94,95,98 while the operation is pending, and once it becomes non-pending, it will exit due to the condition in line 106 or 152.
The help method is a finite loop that calls helpInsert and helpDelete a finite number of times.
The insert and delete methods call the maxPhase and the help method, and have no loops. Note that even though the delete method attempts a CAS, it returns even if the CAS fails.

Corollary 100 *Wait-Freedom*

Proof: A result of the previous claim.

A.7 Final Conclusion

Corollary 101 *The described algorithm creates a wait-free linked-list.*

This follows from Lemmas 75, 88, and Corollary 100.

B A Full Java Implementation

In this appendix, we give a full Java implementation for the basic wait-free linked-list. This basic implementation also uses a VersionedAtomicMarkableReference, in which the reference is associated with a version number for avoiding an ABA problem. A solution and Java code with no versioning requirement is specified in Appendix C. The solution there only employs the standard AtomicMarkableReference. The source for the class VersionedAtomicMarkableReference which implements such versioned pointers is also given right after the WFList. It is obtained by slightly modifying the code of the AtomicMarkableReference of Doug Lea.

```
1  import java.util.concurrent.atomic.AtomicReferenceArray;
2  import java.util.concurrent.atomic.AtomicBoolean;
3  import java.util.concurrent.atomic.AtomicLong;
4
5  public class WFList {
6    enum OpType {insert, search_delete, execute_delete, success, failure,
         determine_delete, contains};
7
8    private class Window {
9      public final Node pred, curr;
10     public Window(Node p, Node c) { pred = p; curr = c; }
11   }
12
13   private class Node {
14     public final int key;
```

```
15      public final VersionedAtomicMarkableReference<Node> next;
16      public final AtomicBoolean d;
17      public Node (int key) {
18        next = new VersionedAtomicMarkableReference<Node>(null, false);
19        this.key = key; d = new AtomicBoolean(false);
20      }
21    }
22
23    private class OpDesc {
24      public final long phase; public final OpType type;
25      public final Node node; public final Window searchResult;
26      public OpDesc (long ph, OpType ty, Node n, Window sResult) {
27        phase = ph; type = ty; node = n; searchResult = sResult;
28      }
29    }
30    private final Node head, tail;
31    private final AtomicReferenceArray<OpDesc> state;
32    private final AtomicLong currentMaxPhase; // used in maxPhase method
33
34    public WFList () {
35      currentMaxPhase = new AtomicLong(); // used in maxPhase method
36      currentMaxPhase.set(0);
37      head = new Node(Integer.MIN_VALUE); // head's key is smaller than all the rests'
38      tail = new Node(Integer.MAX_VALUE); // tail's key is larger than all the rests'
39      head.next.set(tail, false);          // init list to be empty
40      state = new AtomicReferenceArray<OpDesc>(Test.numThreads);
41      for (int i = 0; i < state.length(); i++)  // state entry for each thread
42        state.set(i, new OpDesc(0, OpType.success, null,null));
43    }
44
45    public boolean insert(int tid, int key) {
46      long phase = maxPhase(); // getting the phase for the op
47      Node newNode = new Node(key); // allocating the node
48      OpDesc op = new OpDesc(phase, OpType.insert, newNode,null);
49      state.set(tid, op); // publishing the operation.
50      help(phase);  // when finished - no more pending operation with lower or equal
            phase
51      return state.get(tid).type == OpType.success;
52    }
53
54    public boolean delete(int tid, int key) {
55      long phase = maxPhase(); // getting the phase for the op.
56      state.set(tid, new OpDesc(phase, OpType.search_delete, new Node(key),null));//
            publishing.
57      help (phase); // when finished - no more pending operation with lower or equal
            phase
58      OpDesc op = state.get(tid);
59      if (op.type == OpType.determine_delete)
60        // compete on the ownership of deleting this node
61        return op.searchResult.curr.d.compareAndSet(false, true);
62      return false;
63    }
64
65    private Window search(int key, int tid, long phase) {
66      Node pred = null, curr = null, succ = null;
67      boolean[] marked = {false};  boolean snip;
```

```
68        retry : while (true) {
69          pred = head;
70          curr = pred.next.getReference(); // advancing curr
71          while (true) {
72            succ = curr.next.get(marked); // advancing succ and reading curr.next's mark
73            while (marked[0]) { // curr is logically deleted a should be removed
74              // remove a physically deleted node :
75              snip = pred.next.compareAndSet(curr, succ, false, false);
76              if (!isSearchStillPending(tid,phase))
77                return null; // to ensure wait-freedom.
78              if (!snip) continue retry; // list has changed, retry
79              curr = succ; // advancing curr
80              succ = curr.next.get(marked); // advancing succ and reading curr.next's mark
81            }
82            if (curr.key >= key)  // the curr.key is large enough - found the window
83              return new Window(pred, curr);
84            pred = curr; curr = succ; // advancing pred  & curr
85          }
86        }
87      }
88
89      private void help(long phase) {
90        for (int i = 0; i < state.length(); i++) {
91          OpDesc desc = state.get(i);
92          if (desc.phase <= phase) { // help all pending operations with a desc.phase <=
               phase
93            if (desc.type == OpType.insert) { // a pending insert operation.
94              helpInsert(i, desc.phase);
95            } else if (desc.type == OpType.search_delete
96                || desc.type == OpType.execute_delete) { // a pending delete operation
97              helpDelete(i, desc.phase);
98            } else if (desc.type == OpType.contains) { helpContains(i, desc.phase); }
99          }
100       }
101     }
102
103     private void helpInsert(int tid, long phase) {
104       while (true) {
105         OpDesc op = state.get(tid);
106         if (!(op.type == OpType.insert && op.phase == phase))
107           return; // the op is no longer relevant, return
108         Node node = op.node; // getting the node to be inserted
109         Node node_next = node.next.getReference(); //must read node_next before search
110         Window window = search(node.key,tid,phase); //search a window to insert the node
               into
111         if (window == null) // can only happen if operation is no longer pending
112           return;
113         if (window.curr.key == node.key) { // key exists - chance of a failure
114           if ((window.curr == node) || (node.next.isMarked())) {
115             // the node was already inserted - success
116             OpDesc success = new OpDesc(phase, OpType.success, node, null);
117             if (state.compareAndSet(tid, op, success))
118               return;
119           }
120           else { // the node was not yet inserted - failure
121             OpDesc fail = new OpDesc(phase, OpType.failure, node, null);
```

```
122             // CAS may fail if search results are obsolete
123             if (state.compareAndSet(tid, op, fail))
124               return;
125           }
126         }
127         else {
128           if (node.next.isMarked()) { // node was already inserted and marked (=deleted)
129             OpDesc success = new OpDesc(phase, OpType.success, node, null);
130             if (state.compareAndSet(tid, op, success))
131               return;
132           }
133           int version = window.pred.next.getVersion(); // read version for CAS later.
134           OpDesc newOp = new OpDesc(phase, OpType.insert, node, null);
135     // the following prevents another thread with obsolete search results to report
          failure:
136           if (!state.compareAndSet(tid, op, newOp))
137             continue; // operation might have already reported as failure
138           node.next.compareAndSet(node_next, window.curr, false, false);
139           // if successful - than the insert is linearized here :
140           if (window.pred.next.compareAndSet(version, node.next.getReference(), node,
                false, false)) {
141             OpDesc success = new OpDesc(phase, OpType.success, node, null);
142             if (state.compareAndSet(tid, newOp, success))
143               return;
144           }
145         }
146       }
147     }
148
149     private void helpDelete(int tid, long phase) {
150       while (true) {
151         OpDesc op = state.get(tid);
152         if (!((op.type == OpType.search_delete || op.type == OpType.execute_delete)
153             && op.phase==phase))
154           return; // the op is no longer relevant, return
155         Node node = op.node;  // the node holds the key we want to delete
156         if (op.type == OpType.search_delete) {  // need to search for the key
157           Window window = search(node.key,tid,phase);
158           if (window==null)
159             continue; // can only happen if operation is no longer the same
                  search_delete
160           if (window.curr.key != node.key) {
161             // key doesn't exist - failure
162             OpDesc failure = new OpDesc(phase, OpType.failure, node, null);
163             if (state.compareAndSet(tid, op, failure))
164               return;
165           }
166           else {
167             // key exists - continue to executre_delete
168             OpDesc found = new OpDesc(phase, OpType.execute_delete, node, window);
169             state.compareAndSet(tid, op, found);
170           }
171         }
172         else if (op.type == OpType.execute_delete) {
173           Node next = op.searchResult.curr.next.getReference();
174           if (!op.searchResult.curr.next.attemptMark(next, true)) // mark the node
```

```
175            continue; // will continue to try to mark it, until it is marked
176          search(op.node.key,tid,phase); // will physically remove the node
177          OpDesc determine =
178            new OpDesc(op.phase, OpType.determine_delete, op.node, op.searchResult);
179          state.compareAndSet(tid, op, determine);
180          return;
181        }
182      }
183    }
184
185    public boolean contains(int tid, int key) {
186      long phase = maxPhase();
187      Node n = new Node(key);
188      OpDesc op = new OpDesc(phase, OpType.contains, n, null);
189      state.set(tid, op);
190      help(phase);
191      return state.get(tid).type == OpType.success;
192    }
193
194    private void helpContains(int tid, long phase) {
195      OpDesc op = state.get(tid);
196      if (!((op.type == OpType.contains) && op.phase==phase))
197        return; // the op is no longer relevant, return
198      Node node = op.node;  // the node holds the key we want to find
199      Window window = search(node.key, tid, phase);
200      if (window == null)
201        return; // can only happen if operation is already complete.
202      if (window.curr.key == node.key) {
203        OpDesc success = new OpDesc(phase, OpType.success, node, null);
204        state.compareAndSet(tid, op, success);
205      }
206      else {
207        OpDesc failure = new OpDesc(phase, OpType.failure, node, null);
208        state.compareAndSet(tid, op, failure);
209      }
210    }
211
212    private long maxPhase() {
213      long result = currentMaxPhase.get();
214      // ensuring maxPhase will increment before this thread next operation :
215      currentMaxPhase.compareAndSet(result, result+1);
216      return result;
217    }
218
219    private boolean isSearchStillPending(int tid, long ph) {
220      OpDesc curr = state.get(tid);
221      return (curr.type == OpType.insert || curr.type == OpType.search_delete
222          || curr.type == OpType.execute_delete || curr.type == OpType.contains) &&
223        curr.phase == ph; // the operation is pending with a phase lower than ph.
224    }
225  }
226
227  public class VersionedAtomicMarkableReference<V>  {
228
229      private static class ReferenceBooleanTriplet<T> {
230          private final T reference;
```

```
231            private final boolean bit;
232            private final int version;
233            ReferenceBooleanTriplet(T r, boolean i, int v) {
234                reference = r; bit = i; version = v;
235            }
236        }
237
238        private final AtomicReference<ReferenceBooleanTriplet<V>>  atomicRef;
239
240        public VersionedAtomicMarkableReference(V initialRef, boolean initialMark) {
241            atomicRef = new AtomicReference<ReferenceBooleanTriplet<V>> (new
                   ReferenceBooleanTriplet<V>(initialRef, initialMark,0));
242        }
243
244        public V getReference() {
245            return atomicRef.get().reference;
246        }
247
248        public boolean isMarked() {
249            return atomicRef.get().bit;
250        }
251
252        public V get(boolean[] markHolder) {
253            ReferenceBooleanTriplet<V> p = atomicRef.get();
254            markHolder[0] = p.bit;
255            return p.reference;
256        }
257
258        public boolean weakCompareAndSet(V       expectedReference,
259                                         V       newReference,
260                                         boolean expectedMark,
261                                         boolean newMark) {
262            ReferenceBooleanTriplet<V> current = atomicRef.get();
263            return  expectedReference == current.reference &&
264                expectedMark == current.bit &&
265                ((newReference == current.reference && newMark == current.bit) ||
266                 atomicRef.weakCompareAndSet(current,
267                                         new ReferenceBooleanTriplet<V>(newReference,
268                                                                  newMark, current.version
                                                                     +1)));
269        }
270
271        public boolean compareAndSet(V       expectedReference,
272                                     V       newReference,
273                                     boolean expectedMark,
274                                     boolean newMark) {
275            ReferenceBooleanTriplet<V> current = atomicRef.get();
276            return  expectedReference == current.reference &&
277                expectedMark == current.bit &&
278                ((newReference == current.reference && newMark == current.bit) ||
279                 atomicRef.compareAndSet(current,
280                                         new ReferenceBooleanTriplet<V>(newReference,
281                                                                  newMark, current.version
                                                                     +1)));
282        }
283
```

```
284        public void set(V newReference, boolean newMark) {
285            ReferenceBooleanTriplet<V> current = atomicRef.get();
286            if (newReference != current.reference || newMark != current.bit)
287                atomicRef.set(new ReferenceBooleanTriplet<V>(newReference, newMark, current
                       .version+1));
288        }
289
290        public boolean attemptMark(V expectedReference, boolean newMark) {
291            ReferenceBooleanTriplet<V> current = atomicRef.get();
292            return  expectedReference == current.reference &&
293                (newMark == current.bit ||
294                 atomicRef.compareAndSet
295                 (current, new ReferenceBooleanTriplet<V>(expectedReference,
296                                                    newMark, current.version+1)));
297        }
298
299        public int getVersion()
300        {
301          return atomicRef.get().version;
302        }
303
304        public boolean compareAndSet(int version, V expectedReference, V newReference,
               boolean expectedMark, boolean newMark) {
305          ReferenceBooleanTriplet<V> current = atomicRef.get();
306            return  expectedReference == current.reference &&
307                expectedMark == current.bit && version == current.version &&
308                ((newReference == current.reference && newMark == current.bit) ||
309                 atomicRef.compareAndSet(current,
310                                    new ReferenceBooleanTriplet<V>(newReference,
311                                                         newMark, current.version
                                                            +1)));
312        }
313    }
```

C Avoiding Versioned Pointers

In the implementation of the basic wait-free linked-list, we used a versioned pointer at the next field of each node. While such a solution is the simplest, it requires the use of a wide CAS. In this appendix we provide a way to avoid the use of versioned pointers. This solution only uses regular pointers with a single mark bit, similarly to the original lock-free algorithm by Harris (in Java, this mark bit is implemented via the AtomicMarkeableReference class). In the basic implementation, we used the CAS of Line 140 (the line notations correspond to the code in Appendix B), when inserting a new node into the list. As described in Section 3, we need it to avoid the following ABA problem. Suppose Thread T_1 is executing an insert of the key k into the list. It searches for a location for the insert, it finds one, and gets stalled just before executing Line 140. While T_1 is stalled, T_2 inserts a different k into the list. After succeeding in that insert, T_2 tries to help the same insert of k that T_1 is attempting to perform. T_2 finds that k already exists and reports failure to the state descriptor. This should terminate the insertion that T_1 is executing with a failure report. But suppose further that the other k is then removed from the list, bringing the list back to exactly the same view as T_1 saw before it got stalled. Now T_1 resumes and the CAS of Line 140 actually succeeds. This course of events is bad, because a key is inserted into the list while a failure is reported about this insertion. Instead of using a versioned pointer to solve this problem, we can use a different path. We will mark the

node that is about to be inserted as logically deleted. This way, even if the ABA problem occurs, the node will never appear in the list. Namely, when failure is detected, we can mark the next pointer of the node we failed inserting. While this won't prevent the node from physically being inserted into the list because of the described ABA problem, it will only be inserted as a logically deleted node, and will be removed next time it is traversed, without ever influencing the logical state of the list. However, marking the next field of the node requires care. Most importantly, before we mark the node, we must be certain that it was not already inserted to the list (by another thread), and when we mark it, we ought to be sure that the operation will be correctly reported as failure (even if the marked node was later physically inserted). To ensure this, we use a gadget denoted *block*. The block is a node with two extra fields - the threadID and the phase of the operation it is meant to fail. The procedure for a failing insertion is thus as follows. Say an operation for inserting a node with key 4 is in progress. This node would be called the *failing node*. Upon searching the list, a node that contains key 4 was found. This node is the *hindering node*.

* Using a CAS, a block will be inserted right after the hindering node.
* The failing node's next field will be marked.
* The state of the operation will be changed to failure.
* The block will be removed.

By ensuring that a node right before a block (this is the hindering node) cannot be logically deleted, and that a new node cannot be inserted between the hindering node and the block, it is guaranteed that when marking the failing node as deleted, a failing node was not yet inserted into the list (since the block is still there, and thus also the hindering node). The block's next field will never be marked, and will enable traversing the list. The block key will be set to a value that is lower than all possible keys in the list (can be the same as the head key). This serves two purposes: first, it allows to differentiate between a regular node and a block (in a strongly typed language such as Java, this is done differently), and second, it allows the contains method to work unchanged, without being aware of the existence of blocks, since it will always traverse past a (node/block) with a smaller key than the one searched for. In Java, the block looks like this :

```
1  private class Block extends Node {
2      int tid; long phase;
3      public Block (int tid, long phase) {
4        super(Integer.MIN_VALUE); this.tid = tid; this.phase = phase;
5      }
6    }
```

Upon reaching a block, we need to make sure that the failing node's next field is marked, report the operation as failed, and then remove the block. This is done in the removeBlock method :

```
1  private void removeBlock(Node pred, Block curr) {
2      OpDesc op = state.get(curr.tid);
3      // both loops are certain to finish after test.numofThreads iterations (likely
           sooner)
4      while (op.type == OpType.insert && op.phase == curr.phase) {
5        // mark the node that its insertion is about to be set to failure
6        while (!op.node.next.attemptMark(op.node.next.getReference(), true));
7        OpDesc failure = new OpDesc(op.phase, OpType.failure, op.node, null);
8        state.compareAndSet(curr.tid, op, failure); // report failure
9        op = state.get(curr.tid);
10     }
11     // physically remove the block (if CAS fails, then the block was already removed)
```

```
12      pred.next.compareAndSet(curr, curr.next.getReference(), false, false);
13    }
```

Note that since the presence of a block doesn't allow certain modifications to the list until it is removed (such as deleting the hindering node), we must allow all threads to help remove a block in order to obtain wait-freedom (or even lock-freedom). Accordingly, the search method plays a role in removing blocks when it traverses them, similarly to the role it plays in physically removing marked nodes. Thus, the loop in the search method to remove marked nodes (lines 73-81) should be modified to :

```
1  while (marked[0] || curr instanceof Block) {
2            if (curr instanceof Block) {
3              removeBlock(pred, (Block)curr);
4            }
5            else {
6              // remove a physically deleted node :
7              snip = pred.next.compareAndSet(curr, succ, false, false);
8              if (!isSearchStillPending(tid,phase))
9                return null; // to ensure wait-freedom
10             if (!snip) continue retry; // list has changed, retry
11           }
12           curr = succ; // advancing curr
13           succ = curr.next.get(marked); // advancing succ and reading curr.next's mark
14         }
```

As mentioned above, we should also make sure that the hindering node will not be marked while the block is still after it. To ensure that, we modify the part in the helpDelete method, that handles the execute_delete OpType (lines 172-181) to be :

```
1  else if (op.type == OpType.execute_delete) {
2          Node next = op.searchResult.curr.next.getReference();
3          if (next instanceof Block) { // cannot delete a node while it is before a
               block
4            removeBlock(op.searchResult.curr, (Block)next);
5            continue;
6          }
7          if (!op.searchResult.curr.next.attemptMark(next, true)) // mark the node
8            continue; // will continue to try to mark it, until it is marked
9          search(op.node.key,tid,phase); // will physically remove the node
10         OpDesc determine =
11           new OpDesc(op.phase, OpType.determine_delete, op.node, op.searchResult);
12         state.compareAndSet(tid, op, determine);
13         return;
14       }
```

The only thing left is to modify the helpInsert method, so that it will insert a block upon failure. Some additional care is needed since, in the basic implementation, observing that the node to be inserted is marked was an indication that the operation succeeded. Now, it can only be used as such an indication if there is not a hindering node with block after it that is trying to fail that same operation. Once the block is removed, the fact that the node's next field is marked can indeed be used for an indication of success, since if it was marked because of a block, the fact that the block was already removed tells us that the operation was already reported as failing in the state array, and there is no danger it will be mistakenly considered a success. The modified helpInsert method is as follows :

```
1  private void helpInsert(int tid, long phase) {
2    while (true) {
```

```
3         OpDesc op = state.get(tid);
4         if (!(op.type == OpType.insert && op.phase == phase))
5           return; // the op is no longer relevant, return
6         Node node = op.node; // getting the node to be inserted
7         Node node_next = node.next.getReference(); //must read node_next before search
8         if (node_next instanceof Block)
9         {
10          removeBlock(node, (Block)node_next);
11          continue;
12        }
13        Window window = search(node.key,tid,phase);//search a window to insert the node
              into
14        if (window == null) // can only happen if operation is no longer pending
15          return;
16        if (window.curr.key == node.key) {// key exists - chance of a failure
17          if ((window.curr == node) || (node.next.isMarked())) {
18            Node window_succ = window.curr.next.getReference();
19            if (window_succ instanceof Block) {
20              removeBlock(window.curr, (Block)window_succ);
21              continue;
22            }
23            // the node was already inserted - success
24            OpDesc success = new OpDesc(phase, OpType.success, node, null);
25            if (state.compareAndSet(tid, op, success))
26              return;
27          }
28          else { // the node was not yet inserted - failure
29            Node window_succ = window.curr.next.getReference();
30            Block block = new Block(tid, op.phase);
31            block.next.set(window_succ, false);
32            // linearization point for failure :
33            if (window.curr.next.compareAndSet(window_succ, block, false, false))
34              removeBlock(window.curr, block); // will complete the operation
35          }
36        }
37        else {
38          if (node.next.isMarked()) { // node was already inserted and marked (=deleted)
39            OpDesc success = new OpDesc(phase, OpType.success, node, null);
40            if (state.compareAndSet(tid, op, success))
41              return;
42          }
43          OpDesc newOp = new OpDesc(phase, OpType.insert, node, null);
44      // the following prevents another thread with obsolete search results to report
            failure:
45          if (!state.compareAndSet(tid, op, newOp))
46            continue; // operation might have already reported as failure
47          node.next.compareAndSet(node_next, window.curr, false, false);
48          // if successful - than the insert is linearized here :
49          if (window.pred.next.compareAndSet(node.next.getReference(), node, false,
                false)) {
50            OpDesc success = new OpDesc(phase, OpType.success, node, null);
51            if (state.compareAndSet(tid, newOp, success))
52              return;
53          }
54        }
55      }
```

```
56    }
```

The linearization point of a failing insert operation is now moved to the CAS that inserts the block. The list is still wait-free, since each thread that comes upon a block can always remove it in a bounded number of steps.

D The Full Code of the Fast-Path-Slow-Path Extension

```
1  import java.util.concurrent.atomic.AtomicLong;
2  import java.util.concurrent.atomic.AtomicReference;
3  import java.util.concurrent.atomic.AtomicReferenceArray;
4  import java.util.concurrent.atomic.AtomicBoolean;
5
6  public class FPSPList implements {
7    enum OpType {insert, search_delete, execute_delete, success, failure,
         determine_delete, contains, update_approximation};
8
9    private class Window {
10     public final Node pred, curr;
11     public Window(Node p, Node c) { pred = p; curr = c; }
12   }
13
14   private class Node {
15     public final int key;
16     public VersionedAtomicMarkableReference<Node> next;
17     public final AtomicBoolean successBit;
18     public Node (int key) {
19       this.key = key;
20       successBit = new AtomicBoolean(false);
21       // the next field will be initialized later.
22     }
23   }
24
25   private class OpDesc {
26     public final long phase; public final OpType type;
27     public final Node node; public final Window searchResult;
28     public OpDesc (long ph, OpType ty, Node n, Window sResult) {
29       phase = ph; type = ty; node = n; searchResult = sResult;
30     }
31   }
32
33   class HelpRecord {
34     int curTid; long lastPhase; long nextCheck;
35     public HelpRecord() { curTid = -1; reset(); }
36     public void reset() {
37       curTid = (curTid + 1) % Test.numThreads;
38       lastPhase = state.get(curTid).phase;
39       nextCheck = HELPING_DELAY;
40     }
41   }
42
43   private class Approximation {
44
45     public Approximation (int size, int tid, long phase) {
```

```
46          this.app_size = size; this.tid = tid; this.phase = phase;
47        }
48        final int app_size;
49        final int tid;     // used to allow safe help
50        final long phase;  // used to allow safe help
51      }
52
53
54      private final Node head, tail;
55      private final AtomicReferenceArray<OpDesc> state;
56      private final AtomicLong currentMaxPhase;
57      private final HelpRecord helpRecords[];
58      private final long HELPING_DELAY = 20;
59      private final int MAX_FAILURES = 20;
60      private final int width = 128; // an optimization, to avoid false sharing.
61      private AtomicReference<Approximation> app; // holds the size approximation
62      private final int[] difCouners; // a private size counter for each thread
63      private final int soft_threshold = 35; // a thread will try to update size
          approximation
64      private final int hard_threshold = 50; // a thread will ask help to update size
          approximation
65
66      public FPSPList () {
67        currentMaxPhase = new AtomicLong(); // used in maxPhase method
68        currentMaxPhase.set(1);
69        head = new Node(Integer.MIN_VALUE); // head's key is smaller than all the rests'
70        tail = new Node(Integer.MAX_VALUE); // tail's key is larger than all the rests'
71        head.next = new VersionedAtomicMarkableReference<Node>(tail, false);// init an
            empty list
72        tail.next = new VersionedAtomicMarkableReference<Node>(tail, false);
73
74        state = new AtomicReferenceArray<OpDesc>(Test.numThreads);
75        helpRecords = new HelpRecord[Test.numThreads*width];
76
77        for (int i = 0; i < state.length(); i++) { // state & helpRecord entries for each
            thread
78          state.set(i, new OpDesc(0, OpType.success, null,null));
79          helpRecords[i*width] = new HelpRecord();
80        }
81
82        difCouners = new int[Test.numThreads*width];
83        app = new AtomicReference<Approximation>(new Approximation(0, -1, -1));
84      }
85
86      private void helpIfNeeded(int tid) {
87        HelpRecord rec = helpRecords[tid*width];
88        if (rec.nextCheck-- == 0) { // only check if help is needed after HELPING_DELAY
            times
89          OpDesc desc = state.get(rec.curTid);
90          if (desc.phase == rec.lastPhase) { // if the helped thread is on the same
              operation
91            if (desc.type == OpType.insert)
92              helpInsert(rec.curTid, rec.lastPhase);
93            else if (desc.type == OpType.search_delete || desc.type == OpType.
                execute_delete)
94              helpDelete(rec.curTid, rec.lastPhase);
```

```
95            else if (desc.type == OpType.contains)
96              helpContains(rec.curTid, rec.lastPhase);
97            else if (desc.type == OpType.update_approximation)
98              helpUpdateGlobalCounter(rec.curTid, rec.lastPhase);
99          }
100         rec.reset();
101       }
102     }
103
104     public boolean insert(int tid, int key) {
105       if (updateGlobalCounterIfNeeded(tid, difCouners[tid*width]))
106         difCouners[tid*width] = 0;
107       helpIfNeeded(tid);
108       int tries = 0;
109       while (tries++ < MAX_FAILURES) { // when tries reaches MAX_FAILURES - switch to
            slowPath
110         Window window = fastSearch(key, tid);
111         if (window == null) {// happens if search failed MAX_FAILURES times
112           boolean result = slowInsert(tid, key);
113           if (result)
114             difCouners[tid*width]++;
115           return result;
116         }
117         Node pred = window.pred, curr = window.curr;
118         if (curr.key == key)
119           return false; // key exists - operation failed.
120         else {
121           Node node = new Node(key); // allocate the node to insert
122           node.next = new VersionedAtomicMarkableReference<Node>(curr, false);
123           if (pred.next.compareAndSet(curr, node, false, false))
124             return true; // insertion succeeded
125         }
126       }
127       boolean result = slowInsert(tid, key);
128       if (result)
129         difCouners[tid*width]++;
130       return result;
131     }
132
133     public boolean delete(int tid, int key) {
134       if (updateGlobalCounterIfNeeded(tid, difCouners[tid*width]))
135         difCouners[tid*width] = 0;
136
137       helpIfNeeded(tid);
138       int tries = 0; boolean snip;
139       while (tries++ < MAX_FAILURES) { // when tries reaches MAX_FAILURES - switch to
            slowPath
140         Window window = fastSearch(key, tid);
141         if (window == null) {// happens if search failed MAX_FAILURES times
142           boolean result = slowDelete(tid, key);
143           if (result)
144             difCouners[tid*width]--;
145           return result;
146         }
147         Node pred = window.pred, curr = window.curr;
148         if (curr.key != key) // key doesn't exist - operation failed
```

```
149           return false;
150         else {
151           Node succ = curr.next.getReference();
152           snip = curr.next.compareAndSet(succ, succ, false, true); // logical delete
153           if (!snip)
154             continue; // try again
155           pred.next.compareAndSet(curr, succ, false, false); // physical delete (may
                fail)
156           boolean result = curr.successBit.compareAndSet(false, true); //needed for
                cooperation with slow path
157           if (result)
158             difCouners[tid*width]--;
159           return result;
160         }
161       }
162       boolean result = slowDelete(tid, key);
163       if (result)
164         difCouners[tid*width]--;
165       return result;
166     }
167
168
169     final long MaxError = Test.numThreads*hard_threshold;
170
171     public Window fastSearch(int key, int tid) {
172       long maxSteps = sizeApproximation()+MaxError;
173       int tries = 0;
174       Node pred = null, curr = null, succ = null;
175       boolean[] marked = {false};
176       boolean snip;
177       retry: while (tries++ < MAX_FAILURES) { // when tries reaches MAX_FAILURES -
            return null
178         long steps = 0;
179         pred = head;
180         curr = pred.next.getReference(); // advancing curr
181         while (true) {
182           steps++;
183           if (steps >= maxSteps)
184           {
185             return null;
186           }
187           succ = curr.next.get(marked); // advancing succ and reading curr.next's mark
188           while (marked[0]) { // curr is logically deleted a should be removed
189             if (steps >= maxSteps)
190             {
191               return null;
192             }
193             // remove a physically deleted node:
194             snip = pred.next.compareAndSet(curr, succ, false, false);
195             if (!snip) continue retry; // list has changed, retry
196             curr = succ; // advancing curr
197             succ = curr.next.get(marked); // advancing succ and reading curr.next's mark
198             steps++;
199           }
200           if (curr.key >= key) // the curr.key is large enough - found the window
201             return new Window(pred, curr);
```

```
202           pred = curr; curr = succ;  // advancing pred  & curr
203         }
204       }
205       return null;
206     }
207
208     private boolean slowInsert(int tid, int key) {
209       long phase = maxPhase(); // getting the phase for the op
210       Node n = new Node(key); // allocating the node
211       n.next = new VersionedAtomicMarkableReference<Node>(null, false); // allocate node
             .next
212       OpDesc op = new OpDesc(phase, OpType.insert, n,null);
213       state.set(tid, op); // publishing the operation - asking for help
214       helpInsert(tid, phase); // only helping itself here
215       return state.get(tid).type == OpType.success;
216     }
217
218     private boolean slowDelete(int tid, int key) {
219       long phase = maxPhase(); // getting the phase for the op
220       state.set(tid, new OpDesc(phase, OpType.search_delete, new Node(key), null));//
             publishing
221       helpDelete(tid,phase);    // only helping itself here
222       OpDesc op = state.get(tid);
223       if (op.type == OpType.determine_delete)
224         // compete on the ownership of deleting this node
225         return op.searchResult.curr.successBit.compareAndSet(false, true);
226       return false;
227     }
228
229     private Window search(int key, int tid, long phase) {
230       Node pred = null, curr = null, succ = null;
231       boolean[] marked = {false};  boolean snip;
232       retry : while (true) {
233         pred = head;
234         curr = pred.next.getReference(); // advancing curr
235         while (true) {
236           succ = curr.next.get(marked); // advancing succ and reading curr.next's mark
237           while (marked[0]) { // curr is logically deleted a should be removed
238             // remove a physically deleted node :
239             snip = pred.next.compareAndSet(curr, succ, false, false);
240             if (!isSearchStillPending(tid,phase))
241               return null; // to ensure wait-freedom.
242             if (!snip) continue retry; // list has changed, retry
243             curr = succ; // advancing curr
244             succ = curr.next.get(marked); // advancing succ and reading curr.next's mark
245           }
246           if (curr.key >= key)  // the curr.key is large enough - found the window
247             return new Window(pred, curr);
248           pred = curr; curr = succ; // advancing pred  & curr
249         }
250       }
251     }
252
253     private void helpInsert(int tid, long phase) {
254       while (true) {
255         OpDesc op = state.get(tid);
```

```
256         if (!(op.type == OpType.insert && op.phase == phase))
257           return; // the op is no longer relevant, return
258         Node node = op.node; // getting the node to be inserted
259         Node node_next = node.next.getReference(); //must read node_next before search
260         Window window = search(node.key,tid,phase); //search a window to insert the node
               into
261         if (window == null) // can only happen if operation is no longer pending
262           return;
263         if (window.curr.key == node.key) { // key exists - chance of a failure
264           if ((window.curr == node) || (node.next.isMarked())) {
265             // the node was already inserted - success
266             OpDesc success = new OpDesc(phase, OpType.success, node, null);
267             if (state.compareAndSet(tid, op, success))
268               return;
269           }
270           else { // the node was not yet inserted - failure
271             OpDesc fail = new OpDesc(phase, OpType.failure, node, null);
272             // CAS may fail if search results are obsolete
273             if (state.compareAndSet(tid, op, fail))
274               return;
275           }
276         }
277         else {
278           if (node.next.isMarked()) { // node was already inserted and marked (=deleted)
279             OpDesc success = new OpDesc(phase, OpType.success, node, null);
280             if (state.compareAndSet(tid, op, success))
281               return;
282           }
283           int version = window.pred.next.getVersion(); // read version for CAS later.
284           OpDesc newOp = new OpDesc(phase, OpType.insert, node, null);
285           // the following prevents another thread with obsolete search results to
                 report failure:
286           if (!state.compareAndSet(tid, op, newOp))
287             continue; // operation might have already reported as failure
288           node.next.compareAndSet(node_next, window.curr, false, false);
289           // if successful - than the insert is linearized here:
290           if (window.pred.next.compareAndSet(version, node_next, node, false, false)) {
291             OpDesc success = new OpDesc(phase, OpType.success, node, null);
292             if (state.compareAndSet(tid, newOp, success))
293               return;
294           }
295         }
296       }
297     }
298
299     private void helpDelete(int tid, long phase) {
300       while (true) {
301         OpDesc op = state.get(tid);
302         if (!((op.type == OpType.search_delete || op.type == OpType.execute_delete)
303             && op.phase==phase))
304           return; // the op is no longer relevant, return
305         Node node = op.node;  // the node holds the key we want to delete
306         if (op.type == OpType.search_delete) {  // need to search for the key
307           Window window = search(node.key,tid,phase);
308           if (window==null)
```

```
309             continue; // can only happen if operation is no longer the same
                    search_delete
310           if (window.curr.key != node.key) {
311             // key doesn't exist - failure
312             OpDesc failure = new OpDesc(phase, OpType.failure, node, null);
313             if (state.compareAndSet(tid, op, failure))
314               return;
315           }
316           else {
317             // key exists - continue to executre_delete
318             OpDesc found = new OpDesc(phase, OpType.execute_delete, node, window);
319             state.compareAndSet(tid, op, found);
320           }
321         }
322         else if (op.type == OpType.execute_delete) {
323           Node next = op.searchResult.curr.next.getReference();
324           if (!op.searchResult.curr.next.attemptMark(next, true)) // mark the node
325             continue; // will continue to try to mark it, until it is marked
326           search(op.node.key,tid,phase); // will physically remove the node
327           OpDesc determine =
328             new OpDesc(op.phase, OpType.determine_delete, op.node, op.searchResult);
329           state.compareAndSet(tid, op, determine);
330           return;
331         }
332       }
333     }
334
335     public boolean contains(int tid, int key) {
336       long maxSteps = sizeApproximation()+MaxError;
337       long steps = 0;
338       boolean[] marked = {false};
339       Node curr = head;
340       while (curr.key < key) { // search for the key
341         curr = curr.next.getReference();
342         curr.next.get(marked);
343         if (steps++ >= maxSteps)
344           return slowContains(tid, key);
345       }
346       return (curr.key == key && !marked[0]); // the key is found and is logically in
            the list
347     }
348
349     private boolean slowContains(int tid, int key) {;
350       long phase = maxPhase();
351       Node n = new Node(key);
352       OpDesc op = new OpDesc(phase, OpType.contains, n, null);
353       state.set(tid, op);
354       helpContains(tid, phase);
355       return state.get(tid).type == OpType.success;
356     }
357
358     private void helpContains(int tid, long phase) {
359       OpDesc op = state.get(tid);
360       if (!((op.type == OpType.contains) && op.phase==phase))
361         return; // the op is no longer relevant, return
362       Node node = op.node;  // the node holds the key we want to find
```

```
363       Window window = search(node.key, tid, phase);
364       if (window == null)
365         return; // can only happen if operation is already complete.
366       if (window.curr.key == node.key) {
367         OpDesc success = new OpDesc(phase, OpType.success, node, null);
368         state.compareAndSet(tid, op, success);
369       }
370       else {
371         OpDesc failure = new OpDesc(phase, OpType.failure, node, null);
372         state.compareAndSet(tid, op, failure);
373       }
374     }
375
376     private long maxPhase() {
377       long result = currentMaxPhase.get();
378       // ensuring maxPhase will increment before this thread next operation :
379       currentMaxPhase.compareAndSet(result, result+1);
380       return result;
381     }
382
383     private boolean isSearchStillPending(int tid, long ph) {
384       OpDesc curr = state.get(tid);
385       return (curr.type == OpType.insert || curr.type == OpType.search_delete
386           || curr.type == OpType.execute_delete || curr.type == OpType.contains) &&
387           curr.phase == ph; // the operation is pending with a phase lower than ph.
388     }
389
390
391
392
393     private boolean updateGlobalCounterIfNeeded(int tid, int updateSize) {
394       if (Math.abs(updateSize) < soft_threshold)
395         return false; // no update was done.
396       Approximation old = app.get();
397       // old.tid != -1 means you cannot update since a help for updating is currently in
            action
398       if (old.tid == -1)
399       {
400         Approximation newApp = new Approximation(old.app_size + updateSize, -1, -1);
401         if (app.compareAndSet(old, newApp))
402           return true; // update happened successfully.
403       }
404       if (Math.abs(updateSize) < hard_threshold)
405         return false; // update failed once, we will try again next operation.
406       // need to ask for help in updating the counter, since it reached hard_threshold
407       long phase = maxPhase();
408       Node n = new Node(updateSize); // we will use the node key field to hold the
            update size needed.
409       OpDesc desc = new OpDesc(phase, OpType.update_approximation, n, null);
410       state.set(tid, desc);
411       helpUpdateGlobalCounter(tid, phase);
412       // after the help returned, the counter is surely updated.
413       return true;
414     }
415
416     private void helpUpdateGlobalCounter(int tid, long phase) {
```

```
417       while (true) {
418         OpDesc op = state.get(tid);
419         if (!((op.type == OpType.update_approximation) && op.phase==phase))
420           return; // the op is no longer relevant, return
421         Approximation oldApp = app.get();
422         if (op != state.get(tid))
423           return; // validating op.
424         if (oldApp.tid != -1) { // some help (maybe this one) is in process
425           OpDesc helpedTid = state.get(oldApp.tid);
426           if (helpedTid.phase == oldApp.phase && helpedTid.type == OpType.
                  update_approximation) {
427             // need to report to the oldApp.tid that its update is completed.
428             OpDesc success = new OpDesc(helpedTid.phase, OpType.success, helpedTid.node,
                     null);
429             state.compareAndSet(oldApp.tid, helpedTid, success);
430           }
431           // now we are certain the success has been reported, clean the approximation
                  field.
432           Approximation clean = new Approximation(oldApp.app_size, -1, -1);
433           app.compareAndSet(oldApp, clean);
434           continue;
435         }
436         int updateSize = op.node.key; // here we hold the updateSize
437         Approximation newApp = new Approximation(oldApp.app_size+updateSize, tid, phase)
              ;
438         app.compareAndSet(oldApp, newApp);
439       }
440     }
441
442     private long sizeApproximation() {
443       return app.get().app_size;
444     }
445
446   }
```