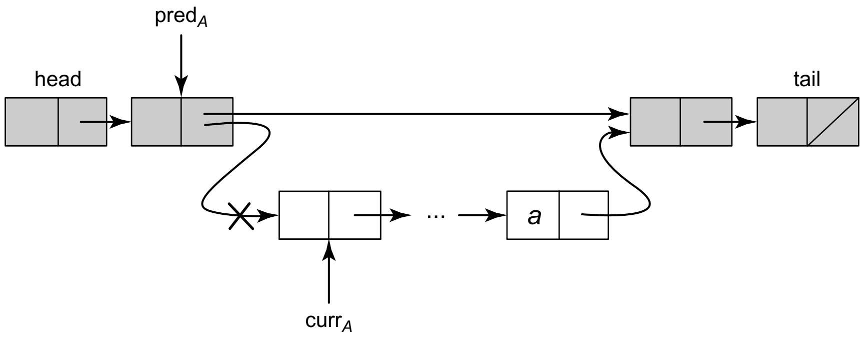

Figure 9.15 The `OptimisticList` class: why validation is needed. Thread *A* is attempting to remove a node *a*. While traversing the list, `curr`$_A$ and all nodes between `curr`$_A$ and *a* (including *a*) might be removed (denoted by a lighter node color). In such a case, thread *A* would proceed to the point where `curr`$_A$ points to *a*, and, without validation, would successfully remove *a*, even though it is no longer in the list. Validation is required to determine that *a* is no longer reachable from `head`.

The `OptimisticList` algorithm is not starvation-free, even if all node locks are individually starvation-free. A thread might be delayed forever if new nodes are repeatedly added and removed (see Exercise 107). Nevertheless, we would expect this algorithm to do well in practice, since starvation is rare.

9.7 Lazy Synchronization

The `OptimisticList` implementation works best if the cost of traversing the list twice without locking is significantly less than the cost of traversing the list once with locking. One drawback of this particular algorithm is that `contains()` acquires locks, which is unattractive since `contains()` calls are likely to be much more common than calls to other methods.

The next step is to refine this algorithm so that `contains()` calls are wait-free, and `add()` and `remove()` methods, while still blocking, traverse the list only once (in the absence of contention). We add to each node a Boolean `marked` field indicating whether that node is in the set. Now, traversals do not need to lock the target node, and there is no need to validate that the node is reachable by retraversing the whole list. Instead, the algorithm maintains the invariant that every unmarked node is reachable. If a traversing thread does not find a node, or finds it marked, then that item is not in the set. As a result, `contains()` needs only one wait-free traversal. To add an element to the list, `add()` traverses the list, locks the target's predecessor, and inserts the node. The `remove()` method is lazy, taking two steps: first, mark the target node, *logically* removing it, and second, redirect its predecessor's `next` field, *physically* removing it.

In more detail, all methods traverse the list (possibly traversing logically and physically removed nodes) ignoring the locks. The add() and remove() methods lock the $\mathtt{pred}_A$ and $\mathtt{curr}_A$ nodes as before (Figs. 9.16 and 9.17), but validation does not retraverse the entire list (Fig. 9.18) to determine whether a node is in the set. Instead, because a node must be marked before being physically removed, validation need only check that $\mathtt{curr}_A$ has not been marked. However, as Fig. 9.19 shows, for insertion and deletion, since $\mathtt{pred}_A$ is the one being modified, one must also check that $\mathtt{pred}_A$ itself is not marked, and that it points to $\mathtt{curr}_A$. Logical removals require a small change to the abstraction map: an item is in the set if, and only if it is referred to by an *unmarked* reachable node. Notice that the path along

```
private boolean validate(Node pred, Node curr) {
  return !pred.marked && !curr.marked && pred.next == curr;
}
```

Figure 9.16 The LazyList class: validation checks that neither the pred nor the curr nodes has been logically deleted, and that pred points to curr.

```
public boolean add(T item) {
 int key = item.hashCode();
 while (true) {
    Node pred = head;
    Node curr = head.next;
    while (curr.key < key) {
     pred = curr; curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key == key) {
            return false;
          } else {
            Node node = new Node(item);
            node.next = curr;
            pred.next = node;
            return true;
          }
        }
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
  }
}
```

Figure 9.17 The LazyList class: add() method.

```
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) {
   Node pred = head;
   Node curr = head.next;
   while (curr.key < key) {
    pred = curr; curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key != key) {
            return false;
          } else {
            curr.marked = true;
            pred.next = curr.next;
            return true;
          }
        }
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
  }
}
```

Figure 9.18 The `LazyList` class: the `remove()` method removes nodes in two steps, logical and physical.

```
public boolean contains(T item) {
  int key = item.hashCode();
  Node curr = head;
  while (curr.key < key)
    curr = curr.next;
  return curr.key == key && !curr.marked;
}
```

Figure 9.19 The `LazyList` class: the `contains()` method.

which the node is reachable may contain marked nodes. The reader should check that any unmarked reachable node remains reachable, even if its predecessor is logically or physically deleted. As in the `OptimisticList` algorithm, `add()` and `remove()` are not starvation-free, because list traversals may be arbitrarily delayed by ongoing modifications.

The `contains()` method (Fig. 9.20) traverses the list once ignoring locks and returns *true* if the node it was searching for is present and unmarked, and *false*

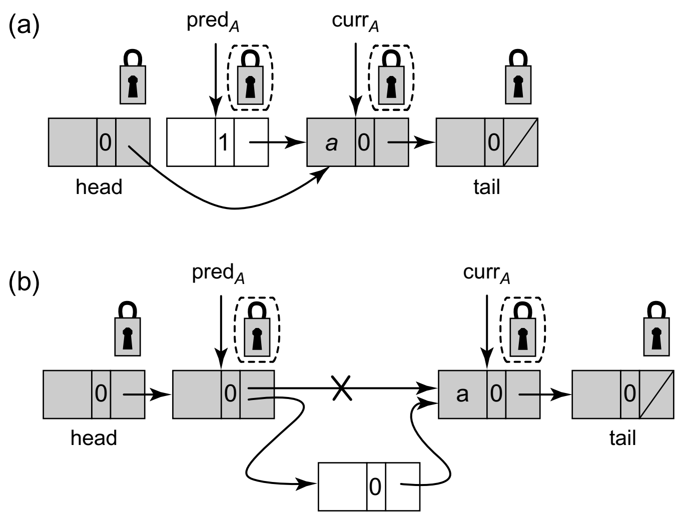

Figure 9.20 The `LazyList` class: why validation is needed. In Part (a) of the figure, thread *A* is attempting to remove node *a*. After it reaches the point where `pred`$_A$ refers to `curr`$_A$, and before it acquires locks on these nodes, the node `pred`$_A$ is logically and physically removed. After *A* acquires the locks, validation will detect the problem. In Part (b) of the figure, *A* is attempting to remove node *a*. After it reaches the point where `pred`$_A$ equals `curr`$_A$, and before it acquires locks on these nodes, a new node is added between `pred`$_A$ and `curr`$_A$. After *A* acquires the locks, even though neither `pred`$_A$ or `curr`$_A$ are marked, validation detects that `pred`$_A$ is not the same as `curr`$_A$, and *A*'s call to `remove()` will be restarted.

otherwise. It is thus wait-free.[4] A marked node's value is ignored. Each time the traversal moves to a new node, the new node has a larger key than the previous one, even if the node is logically deleted.

Logical removal requires a small change to the abstraction map: an item is in the set if, and only if it is referred to by an *unmarked* reachable node. Notice that the path along which the node is reachable may contain marked nodes. Physical list modifications and traversals occur exactly as in the `OptimisticList` class, and the reader should check that any unmarked reachable node remains reachable even if its predecessor is logically or physically deleted.

The linearization points for `LazyList` `add()` and unsuccessful `remove()` calls are the same as for the `OptimisticList`. A successful `remove()` call is linearized when the mark is set (Line 17), and a successful `contains()` call is linearized when an unmarked matching node is found.

To understand how to linearize an unsuccessful `contains()`, let us consider the scenario depicted in Fig. 9.21. In Part (a), node *a* is marked as removed (its `marked` field is set) and thread *A* is attempting to find the node matching *a*'s key.

4 Notice that the list ahead of a given traversing thread cannot grow forever due to newly inserted keys, since the key size is finite.

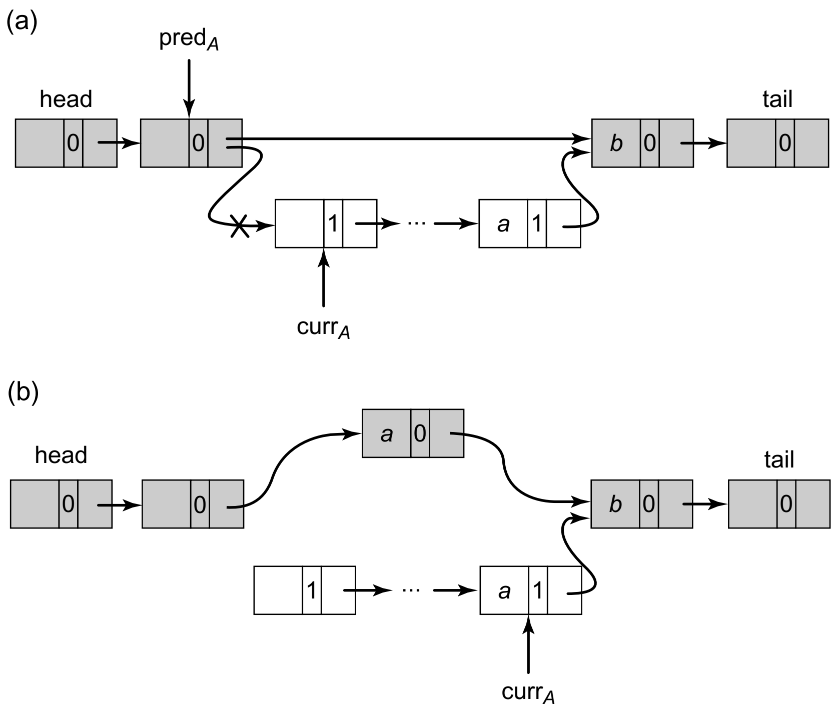

Figure 9.21 The `LazyList` class: linearizing an unsuccessful `contains()` call. Dark nodes are physically in the list and white nodes are physically removed. In Part (a), while thread *A* is traversing the list, a concurrent `remove()` call disconnects the sublist referred to by `curr`. Notice that nodes with items *a* and *b* are still reachable, so whether an item is actually in the list depends only on whether it is marked. Thread *A*'s call is linearized at the point when it sees that *a* is marked and is no longer in the abstract set. Alternatively, let us consider the scenario depicted in Part (b). While thread *A* is traversing the list leading to marked node *a*, another thread adds a new node with key *a*. It would be wrong to linearize thread *A*'s unsuccessful `contains()` call to when it found the marked node *a*, since this point occurs *after* the insertion of the new node with key *a* to the list.

While A is traversing the list, $\texttt{curr}_A$ and all nodes between $\texttt{curr}_A$ and a including a are removed, both logically and physically. Thread A would still proceed to the point where $\texttt{curr}_A$ points to a, and would detect that a is marked and no longer in the abstract set. The call could be linearized at this point.

Now let us consider the scenario depicted in Part (b). While A is traversing the removed section of the list leading to a, and before it reaches the removed node a, another thread adds a new node with a key a to the reachable part of the list. Linearizing thread A's unsuccessful `contains()` method at the point it finds the marked node a would be wrong, since this point occurs *after* the insertion of the new node with key a to the list. We therefore linearize an unsuccessful `contains()` method call within its execution interval at the earlier of the following points: (1) the point where a removed matching node, or a node with a key greater than the one being searched for is found, and (2) the point immediately before a new matching node is added to the list. Notice that the second is guaranteed to be within the execution interval because the insertion of the new node with the same key must have happened after the start of the `contains()` method, or the `contains()` method would have found that item. As can be seen, the linearization point of the unsuccessful `contains()` is determined by the ordering of events in the execution, and is not a predetermined point in the method's code.

One benefit of lazy synchronization is that we can separate unobtrusive logical steps such as setting a flag, from disruptive physical changes to the structure, such as disconnecting a node. The example presented here is simple because we disconnect one node at a time. In general, however, delayed operations can be batched and performed lazily at a convenient time, reducing the overall disruptiveness of physical modifications to the structure.

The principal disadvantage of the `LazyList` algorithm is that `add()` and `remove()` calls are blocking: if one thread is delayed, then others may also be delayed.

9.8 Non-Blocking Synchronization

We have seen that it is sometimes a good idea to mark nodes as logically removed before physically removing them from the list. We now show how to extend this idea to eliminate locks altogether, allowing all three methods, `add()`, `remove()`, and `contains()`, to be nonblocking. (The first two methods are lock-free and the last wait-free). A naïve approach would be to use `compareAndSet()` to change the `next` fields. Unfortunately, this idea does not work. The bottom part of Fig. 9.22 shows a thread *A* attempting to add node *a* between nodes $\texttt{pred}_A$ and $\texttt{curr}_A$. It sets *a*'s `next` field to $\texttt{curr}_A$, and then calls `compareAndSet()` to set $\texttt{pred}_A$'s `next` field to *a*. If *B* wants to remove $\texttt{curr}_B$ from the list, it might call `compareAndSet()` to set $\texttt{pred}_B$'s `next` field to $\texttt{curr}_B$'s successor. It is not hard to see that if these two threads try to remove these adjacent nodes concurrently, the list would end up with *b* not being removed. A similar situation for a pair of concurrent `add()` and `remove()` methods is depicted in the upper part of Fig. 9.22.

Clearly, we need a way to ensure that a node's fields cannot be updated, after that node has been logically or physically removed from the list. Our approach is to treat the node's `next` and `marked` fields as a single atomic unit: any attempt to update the `next` field when the `marked` field is *true* will fail.

Pragma 9.8.1. An `AtomicMarkableReference<T>` is an object from the **java.util.concurrent.atomic** package that encapsulates both a reference to an object of type `T` and a Boolean `mark`. These fields can be updated atomically,