Linked Lists: Locking, Lock-Free, and Beyond …
Linked Lists

• We can make effective spin locks
  - Correct
  - Work well under contention

• Are we done with concurrent data structures?
Not Over Yet

• Contention
  - Solved by MCS or CLH locks

• Sequential Bottleneck
  - No “bag of tricks”

• Linked Lists
  - Simple data structure
  - Good testbed
Set Interface

• Unordered collection of objects
• No duplicates
• Methods
  - Add a new object
  - Remove an object
  - Test if object is present
List-Based Sets

public interface Set {
    public boolean add/Object x/);
    public boolean remove/Object x/);
    public boolean contains/Object x/);
}
List Entry

```java
public class Entry {
    public Object object;
    public int key;
    public Entry next;
}
```
List Entry

```java
public class Entry {
    public Object object;
    public int key;
    public Entry next;
}
```

Object of interest
List Entry

```java
public class Entry {
    public Object object;
    public int key;
    public Entry next;
}
```

Sort by key value (usually hash code)
List Entry

```java
public class Entry {
    public Object object;
    public int key;
    public Entry next;
}
```

Sorting makes it easy to detect absence
List Entry

public class Entry {
    public Object object;
    public int key;
    public Entry next;
}
List-Based Set

Sentinel node never deleted
(minimum possible key)
Adding an Entry
Removing an Entry

```
    a -> b -> c
```

Nu, What About Concurrency?

• Our bag of tricks
  - Coarse-grained locks
  - Fine-grained locks
  - Optimistic synchronization
  - Lock-free synchronization
Coarse-Grained Locking

• Easy, same as synchronized methods
  - “One lock to rule them all ...”

• Simple, clearly correct
  - Deserves respect!

• Works poorly with contention
  - Queue locks help
  - But bottleneck still an issue
Fine-grained Locking

• Requires careful thought
  – “Do not meddle in the affairs of wizards, for they are subtle and quick to anger”

• Split object into pieces
  – Each piece has own lock
  – Methods that work on disjoint pieces need not exclude each other
Optimistic Synchronization

• Requires very careful thought
  - “Do not meddle in the affairs of dragons, for you are crunchy and taste good with ketchup.”

• Try it without synchronization
  - If you win, you win
  - If not, try it again with synchronization
Lock-Free Synchronization

• Dump locking altogether ...
  - “You take the red pill and you stay in Wonderland and I show you how deep the rabbit-hole goes”

• No locks, just native atomic methods
  - Usually `compareAndSet`
Hand-over-Hand locking
Removing an Entry

remove b
Removing an Entry
Uh-oh
Problem

• To delete entry b
  - Swing entry a’s next field to c
• Problem is,
  - Someone could delete c concurrently
Insight

• If an entry is locked
  - No one can delete entry's successor
• If a thread locks
  - Entry to be deleted
  - And its predecessor
  - Then it works
Hand-Over-Hand Again

Found it!

remove b
Removing an Entry
Removing an Entry

© 2003 Herlihy & Shavit
Remove method

public boolean remove(Object object) {
    int key = object.hashCode();
    Entry predEntry, currEntry;
    try {
        ...
    } finally {
        currEntry.unlock();
        predEntry.unlock();
    }
}
public boolean remove(Object object) {
    int key = object.hashCode();
    Entry prevEntry, currEntry;
    try {
        ...
    } finally {
        currEntry.unlock();
        prevEntry.unlock();
    }
}
public boolean remove(Object object) {
    int key = object.hashCode();
    Entry predEntry, currEntry;
    try {
        ... 
    } finally {
        currEntry.unlock();
        prevEntry.unlock();
    }
}
Remove method

```java
public boolean remove(Object object) {
    int key = object.hashCode();
    Entry prevEntry, currEntry;

    try {
        ...
    } finally {
        currEntry.unlock();
        predEntry.unlock();
    }
}```
Remove method

```java
public boolean remove(Object object) {
    int key = object.hashCode();
    Entry prevEntry, currEntry;
    try {
        ...
    } finally {
        currEntry.unlock();
        prevEntry.unlock();
    }
}
```

Everything else
Remove method

```java
try {
    predEntry = this.head;
    predEntry.lock();
    currEntry = predEntry.next;
    currEntry.lock();
    ...
} finally { ... }
```
Remove method

```java
try {
    predEntry = this.head;
    predEntry.lock();
    currEntry = predEntry.next;
    currEntry.lock();
    ...
} finally { … }
```
Remove method

```java
try {
    prevEntry = this.head;
    prevEntry.lock();
    currEntry = predEntry.next;
    currEntry.lock();
    ...
} finally { ... }
```
Remove method

try {
    prevEntry = this.head;
    prevEntry.lock();
    currEntry = prevEntry.next;
    currEntry.lock();
    ...
} finally { ... }

Traversing list
while (currEntry.key <= key) {
    if (object == currEntry.object) {
        predEntry.next = currEntry.next;
        return true;
    }
    predEntry.unlock();
    predEntry = currEntry;
    currEntry = currEntry.next;
    currEntry.lock();
}
return false;
while (currEntry.key <= key) {
  if (object == currEntry.object) {
    prevEntry.next = currEntry.next;
    return true;
  }
  prevEntry.unlock();
  prevEntry = currEntry;
  currEntry = currEntry.next;
  currEntry.lock();
}
return false;
Remove: searching

```java
while (currEntry.key <= key) {
    if (object == currEntry.object) {
        prevEntry.next = currEntry.next;
        return true;
    }
    prevEntry.unlock();
    prevEntry = currEntry;
    currEntry = currEntry.next;
    currEntry.lock();
}
return false;
```

At start of each loop: `currEntry` and `predEntry` locked
Remove: searching

```c
while (currEntry.key <= key) {
    if (object == currEntry.object) {
        predEntry.next = currEntry.next;
        return true;
    }
    prevEntry.unlock();
    prevEntry = currEntry;
    currEntry = currEntry.next;
    currEntry.lock();
}
```

If entry found, remove it
Remove: searching

while (currEntry.key <= key) {
    if (object == currEntry.object) {
        predEntry.next = currEntry.next;
        return true;
    }
    prevEntry.unlock();
    prevEntry = currEntry;
    currEntry = currEntry.next;
    currEntry.lock();
}

If entry found, remove it
Remove: searching

while (currEntry.key <= key) {
    if (object == currEntry.object) {
        prevEntry.next = currEntry.next;
        return true;
    }
    predEntry.unlock();
    prevEntry = currEntry;
    currEntry = currEntry.next;
    currEntry.lock();
}
return false;
Remove: searching

```java
while (currEntry.key <= key) {
    if (object == currEntry.object) {
        prevEntry.next = currEntry.next;
        return true;
    }
    prevEntry.unlock();
    prevEntry = currEntry;
    currEntry = currEntry.next;
    currEntry.lock();
}
return false;
```

Only one entry locked!
Remove: searching

```java
while (currEntry.key <= key) {
    if (object == currEntry.object) {
        prevEntry.next = currEntry.next;
        return true;
    }
    prevEntry.unlock();
    predEntry = currEntry;
    currEntry = currEntry.next;
    currEntry.lock();
}
return false;
```
while (currEntry.key <= key) {
  if (object == currEntry.object) {
    prevEntry.next = currEntry.next; return true;
  }
  prevEntry.unlock();
  prevEntry = currEntry;
  currEntry = currEntry.next;
  currEntry.lock();
}
return false;

Find and lock new current
Remove: searching

```java
while (currEntry.key <= key) {
    if (object == currEntry.object) {
        prevEntry.next = currEntry.next;
        return true;
    }
    prevEntry.unlock();
    prevEntry = currEntry;
    currEntry = currEntry.next;
    currEntry.lock();
}
return false;
```

Lock invariant restored
Remove: searching

while (currEntry.key <= key) {
    if (object == currEntry.object) {
        prevEntry.next = currEntry.next;
        return true;
    }
}
prevEntry.unlock();
prevEntry = currEntry;
currEntry = currEntry.next;
currEntry.lock();

return false;

Otherwise, error 404, dude!
Why does this work?

• To remove entry \( e \)
  - Must lock \( e \)
  - Must lock \( e \)'s predecessor

• Therefore, if you lock an entry
  - It can't be removed
  - And neither can its successor
Lock and Load

• To move to successor entry for e
  - Lock e
  - Lock e.next
  - Unlock e

• While traversing
  - e cannot be removed
  - e.next cannot be removed

  Until next entry identified and locked

  Don’t release e
Adding Entries

• To add entry $e$
  - Must lock predecessor
  - Must lock successor

• Neither can be deleted
  - (Is successor lock actually required?)
Drawbacks

• Better than coarse-grained lock
  - Threads can traverse in parallel
• Still not ideal
  - Long chain of acquire/release
  - Inefficient
Optimistic Synchronization

- Find entries without locking
- Lock entries
- Check that everything is OK
Removing an Entry

```
remove c
return true
```
What Can Go Wrong?

![Diagram showing a sequence of states with arrows indicating transitions and notes to remove certain elements.]

© 2003 Herlihy & Shavit
Check that Entry is Still Accessible

remove c
What Can Go Wrong?

[Diagram showing a sequence of states a, b, c, d with arrows indicating transitions and red and green arrows indicating operations: remove c and add b'.]

© 2003 Herlihy & Shavit
What Can Go Wrong?

remove c
Check that Entries Still Adjacent

remove c
Correctness

• If
  - Entry b and Entry c both locked
  - Entry b still accessible
  - Entry c still successor to b

• Then
  - Neither will be deleted
  - OK to delete and return true
Removing an Absent Entry

```
remove
return false
```
Correctness

• If
  - Entry b and Entry d both locked
  - Entry b still accessible
  - Entry d still successor to b
• Then
  - Neither will be deleted
  - No thread can add c after b
  - OK to return false
private boolean validate(Entry predEntry, Entry currEntry) {
    Entry entry = head;
    while (entry.key <= predEntry.key) {
        if (entry == predEntry)
            return predEntry.next == currEntry;
        entry = entry.next;
    }
    return false;
}
Validation

private boolean validate(Entry predEntry, Entry currEntry) {
    Entry entry = head;
    while (entry.key <= predEntry.key) {
        if (entry == predEntry)
            return predEntry.next == currEntry;
        entry = entry.next;
    }
    return false;
}
private boolean validate(Entry predEntry, Entry currEntry) {
    Entry entry = head;
    while (entry.key <= predEntry.key) {
        if (entry == predEntry) {
            if (entry == predEntry)
                return predEntry.next == currEntry;
        }
        entry = entry.next;
    }
    return false;
}
Validation

```java
private boolean validate(Entry predEntry, Entry currEntry) {
    Entry entry = head;
    while (entry.key <= predEntry.key) {
        if (entry == predEntry)
            return predEntry.next == currEntry;
        entry = entry.next;
    }
    return false;
}
```

Search range of keys
private boolean validate(Entry predEntry, Entry currEntry) {
    Entry entry = head;
    while (entry.key <= predEntry.key) {
        if (entry == predEntry)
            return predEntry.next == currEntry;
        entry = entry.next;
    }
    return false;
}

Predecessor reachable
Validation

private boolean validate(Entry predEntry, Entry currEntry) {
    Entry entry = head;
    while (entry.key <= predEntry.key) {
        if (entry == predEntry)
            return predEntry.next == currEntry;
        entry = entry.next;
    }
    return false;
}
Validation

private boolean validate(Entry predEntry, Entry currEntry) {
    Entry entry = head;
    while (entry.key <= predEntry.key) {
        if (entry == predEntry)
            return predEntry.next == currEntry;
        entry = entry.next;
    }
    return false;
}
Validation

```java
private boolean validate(Entry predEntry, Entry currEntry) {
    Entry entry = head;
    while (entry.key <= predEntry.key) {
        if (entry == predEntry)
            return predEntry.next == currEntry;
        entry = entry.next;
    }
    return false;
}
```

Predecessor not reachable
Remove: searching

```java
public boolean remove(Object object) {
    int key = object.hashCode();
    retry: while (true) {
        Entry predEntry = this.head;
        Entry currEntry = predEntry.next;
        while (currEntry.key <= key) {
            if (object == currEntry.object)
                break;
            predEntry = currEntry;
            currEntry = currEntry.next;
        }
    }
}
```
public boolean remove(Object object) {
    int key = object.hashCode();
    retry: while (true) {
        Entry prevEntry = this.head;
        Entry currEntry = prevEntry.next;
        while (currEntry.key <= key) {
            if (object == currEntry.object) {
                break;
            } else {
                prevEntry = currEntry;
                currEntry = currEntry.next;
            }
        }
    }
}
public boolean remove(Object object) {
    int key = object.hashCode();
    retry: while (true) {
        Entry prevEntry = this.head;
        Entry currEntry = prevEntry.next;
        while (currEntry.key <= key) {
            if (object == currEntry.object)
                break;
            prevEntry = currEntry;
            currEntry = currEntry.next;
        }
    }
    ...  
    Retry on synchronization conflict
public boolean remove(Object object) {
    int key = object.hashCode();
    retry: while (true) {
        Entry predEntry = this.head;
        Entry currEntry = predEntry.next;
        while (currEntry.key <= key) {
            if (object == currEntry.object)
                break;
            prevEntry = currEntry;
            currEntry = currEntry.next;
        }
        // Examine predecessor and current entries
    }
}
public boolean remove(Object object) {
    int key = object.hashCode();
    retry: while (true) {
        Entry prevEntry = this.head;
        Entry currEntry = prevEntry.next;
        while (currEntry.key <= key) {
            if (object == currEntry.object)
                break;
            prevEntry = currEntry;
            currEntry = currEntry.next;
        }
    }
    return true;
}
Remove: searching

```java
public boolean remove(Object object) {
    int key = object.hashCode();
    retry: while (true) {
        Entry prevEntry = this.head;
        Entry currEntry = prevEntry.next;
        while (currEntry.key <= key) {
            if (object == currEntry.object)
                break;
            prevEntry = currEntry;
            currEntry = currEntry.next;
        }
        Stop if we find object
    }
}
```
Remove: searching

```java
public boolean remove(Object object) {
    int key = object.hashCode();
    retry: while (true) {
        Entry prevEntry = this.head;
        Entry currEntry = prevEntry.next;
        while (currEntry.key <= key) {
            if (object == currEntry.object)
                break;
            predEntry = currEntry;
            currEntry = currEntry.next;
        }
    }
}
```
On Exit from Loop

• If object is present
  - currEntry holds object
  - predEntry just before currEntry

• If object is absent
  - currEntry has first higher key
  - predEntry just before currEntry

• Assuming no synchronization problems
Remove Method

try {
    predEntry.lock(); currEntry.lock();
    if (validate(predEntry, currEntry) {
        if (currEntry.object == object) {
            predEntry.next = currEntry.next;
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
}} finally {
    predEntry.unlock();
    currEntry.unlock();
}}
Remove Method

```java
try {
predEntry.lock(); currEntry.lock();
if (validate(predEntry, currEntry)) {
    if (currEntry.object == object) {
        predEntry.next = currEntry.next;
        return true;
    } else {
        return false;
    }
} else {
    return false;
}
} finally {
predEntry.unlock();
currEntry.unlock();
}
```

Always unlock
Remove Method

```java
try {
    // Lock both entries
    predEntry.lock(); currEntry.lock();
    if (validate(predEntry, currEntry)) {
        if (currEntry.object == object) {
            predEntry.next = currEntry.next;
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
} finally {
    // Unlock both entries
    predEntry.unlock();
    currEntry.unlock();
}
```
Remove Method

```java
try {
    predEntry.lock(); currEntry.lock();
    if (validate(predEntry, currEntry)) {
        if (currEntry.object == object) {
            predEntry.next = currEntry.next;
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
} finally {
    predEntry.unlock();
    currEntry.unlock();
}
```

Check for synchronization conflicts
Remove Method

```java
try {
    predEntry.lock(); currEntry.lock();
    if (validate(predEntry, currEntry)) {
        if (currEntry.object == object) {
            predEntry.next = currEntry.next;
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
} finally {
    predEntry.unlock();
    currEntry.unlock();
}
```

Object found, remove entry
Remove Method

```java
try {
    predEntry.lock(); currEntry.lock();
    if (validate(predEntry, currEntry)) {
        if (currEntry.object == object) {
            predEntry.next = currEntry.next;
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
} finally {
    predEntry.unlock();
    currEntry.unlock();
}
```
So Far, So Good

• **Much less lock acquisition/release**
  - Performance
  - Concurrency

• **Problems**
  - Need to traverse list twice
  - `contains()` method acquires locks
    • Most common method call
Marked Lists

• Remove Method
  - Scans list (as before)
  - Locks predecessor & current (as before)
  - Marks current entry as removed (new!)
  - Redirects predecessor’s next (as before)
Marked Lists

• All Methods
  - Scan list
  - Do not scan past marked entry
  - Instead, start over from list head
Business as Usual
Interference
Interference
Interference
Marking a Node

• **AtomicMarkableReference** class
  - Java.util.concurrent.atomic package
Extracting Information

Public Object get(boolean[]);
Extracting Information

Public Object get(boolean[]);

- Returns reference
- Stores mark at index 0
Extracting Information

```java
public boolean isMarked();
```

Value of mark
public boolean compareAndSet(
    Object expectedRef,
    Object updateRef,
    boolean expectedMark,
    boolean updateMark);
Changing State

If this is the current reference ...

```
public boolean compareAndSet(
    Object expectedRef,
    Object updateRef,
    boolean expectedMark,
    boolean updateMark);
```

And this is the current mark ...
Changing State

```java
public boolean compareAndSet(
    Object expectedRef,
    Object updateRef,
    boolean expectedMark,
    boolean updateMark);
```

...then change to this new reference ...

... and this new mark
Changing State

```java
public boolean attemptMark(
    Object expectedRef,
    boolean updateMark);
```
Changing State

public boolean attemptMark(
    Object expectedRef,
    boolean updateMark);

If this is the current reference ...
Changing State

```java
public boolean attemptMark(
    Object expectedRef,
    boolean updateMark);
```

.. then change to this new mark.
Aside

- We will see that it is often useful to tag pointers with
  - Boolean values
  - Integer values
- Sometimes to mark, and sometimes to ensure pointers are unique
List Validate Method

```java
private boolean validate(Entry predEntry, Entry currEntry) {
    return (!predEntry.next.isMarked()) &&
           (!currEntry.next.isMarked()) &&
           (predEntry.next.getReference() == currEntry);
}
```
private boolean validate(Entry predEntry, Entry currEntry) {
    return (!predEntry.next.isMarked()) && (!currEntry.next.isMarked()) && (predEntry.next.getReference() == currEntry);
}
List Validate Method

```java
private boolean validate(Entry predEntry, Entry currEntry) {
    return (!predEntry.next.isMarked()) && (!currEntry.next.isMarked()) &&
            (predEntry.next.getReference() == currEntry);
}
```

Current not removed
List Validate Method

```java
private boolean validate(Entry predEntry, Entry currEntry) {
    return (!predEntry.next.isMarked()) && (!currEntry.next.isMarked()) &&
           (predEntry.next.getReference() == currEntry);
}
```
public boolean remove(Object object) {

    ... while (currEntry.key <= key) {
        Entry nextEntry =
            (Entry)currEntry.next.get(mark);
        if (mark[0])
            continue retry;
        if (object == currEntry.object)
            break;
        predEntry = currEntry;
        currEntry = currEntry.next;
    } ...
}
public boolean remove(Object object) {
    ... 
    while (currEntry.key <= key) {
        Entry nextEntry =
            (Entry)currEntry.next.get(mark);
        if (mark[0])
            continue retry;
        if (object == currEntry.object)
            break;
        predEntry = currEntry;
        currEntry = currEntry.next;
    } ...

    Atomically read reference & mark
public boolean remove(Object object) {
    …
    while (currEntry.key <= key) {
        Entry nextEntry = (Entry) currEntry.next.get(mark);
        if (mark[0])
            continue retry;
        if (object == currEntry.object)
            break;
        predEntry = currEntry;
        currEntry = currEntry.next;
    }
    …
    Panic if entry removed
Evaluation

• **Good:**
  - Contains method doesn’t need to lock
  - Uncontended calls don’t re-traverse

• **Bad**
  - Contended calls do re-traverse
  - Traffic jam if one thread delays
Traffic Jam

- Any concurrent data structure based on mutual exclusion has a weakness
- If one thread
  - Enters critical section
  - And “eats the big muffin” (stops running)
    - Cache miss, page fault, descheduled ...
    - Software error, ...
  - Everyone else using that lock is stuck!
Lock-Free Data Structures

• No matter what ...
  - Some thread will complete method call
  - Even if others halt at malicious times

• Implies that
  - You can’t use locks (why?)
  - Um, that’s why they call it lock-free
Lock-Free ≠ Wait-Free

- **Wait-free synchronization**
  - Every method call eventually finishes
  - What everyone really wants
- **Lock-free synchronization**
  - Some method call eventually finishes
  - What we are usually willing to pay for
  - Starvation rare in practice...
Lock-Free Lists

• Next logical step
• Eliminate locking entirely
• Use only `compareAndSet()`
• Invented by Maged Michael, 2003
Adding an Entry

![Diagram showing the process of adding an entry with CAS (Compare And Swap)]
Removing an Entry
Removing an Entry

```
| a | ←CAS→ | b |
| ←CAS→ | c | ←d→ |
```

- **CAS** (Comparing and Swapping)
- Remove **b**
- Remove **c**
Look Familiar?

remove b

remove c
Problem

- Method updates entry’s next field
- After entry has been removed
Solution

- **Use** `AtomicMarkableReference`
- **Remove in two steps**
  - Set **mark bit in next field**
  - Redirect predecessor’s pointer
- **CAS**
  - Fails if **mark bit set** *(entry removed)*
Removing an Entry

a → b → CAS → c → d

© 2003 Herlihy & Shavit
Removing an Entry

- CAS
- CAS
- failed
- remove b
- remove c
Removing an Entry

![Diagram of removing an entry with nodes labeled a, b, c, and d. Node b is the one being removed.]

© 2003 Herlihy & Shavit
Removing an Entry

remove b

remove c

remove d
Traversing the List

- **Q:** what do you do when you find a “logically” deleted entry in your path?
- **A:** finish the job.
  - CAS the predecessor’s next field
  - Proceed (repeat as needed)
Lock-Free Traversal
The Find Method

pred, curr, next = find(object);
The Find Method

\[ \text{pred, curr, succ} = \text{find(object)}; \]

At some instant,

pred, curr, succ

or ...
The Find Method

At some instant, \( \text{pred}, \text{curr}, \text{succ} = \text{find}(\text{object}); \)

object not in list

pred

curr = null

succ
public boolean remove(Object object) {
  while (true) {
    pred, curr, succ = find(object);
    if (curr == null)
      return false;
    if (!curr.next.attemptMark(succ, true))
      continue;
    if (!curr.next.attemptMark(succ, true))
      continue;
    pred.next.compareAndSet(curr, succ, false, false);
    return true;
  }
}
public boolean remove(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr == null)
            return false;
        if (!curr.next.attemptMark(succ, true))
            continue;
        pred.next.compareAndSet(curr, succ, false, false);
        return true;
    }
}
public boolean remove(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr == null)
            return false;
        if (!curr.next.attemptMark(succ, true))
            continue;
        pred.next.compareAndSet(curr, succ, false, false);
        return true;
    }
}
public boolean remove(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr == null)
            return false;
        if (!curr.next.attemptMark(succ, true))
            continue;
        pred.next.compareAndSet(curr, succ, false, false);
        return true;
    }
}

She's not there ...
Remove

Try to mark entry as deleted

```java
public boolean remove(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr == null)
            return false;
        if (!curr.next.attemptMark(succ, true))
            continue;
        pred.next.compareAndSet(curr, succ, false, false);
        return true;
    }
}
```
public boolean remove(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr == null)
            return false;
        if (!curr.next.attemptMark(succ, true))
            continue;
        pred.next.compareAndSet(curr, succ, false, false);
        return true;
    }
}

If it doesn't work, just retry
Remove

If it works, our job is (essentially) done

```java
while (true) {
    pred, curr, succ = find(object);
    if (curr == null)
        return false;
    if (!curr.next.attemptMark(succ, true))
        continue;
    pred.next.compareAndSet(curr, succ, false, false);
    return true;
}
```

© 2003 Herlihy & Shavit
Remove

public boolean remove(Object object) {
  while (true) {
    pred, curr, succ = find(object);
    if (curr == null)
      return false;
    if (!curr.next.attemptMark(succ, true))
      continue;
    pred.next.compareAndSet(curr, succ, false, false);
    return true;
  }
}

Try to advance reference (if we don't succeed, someone else did).
public boolean add(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr != null)
            return false;
        Entry entry = new Entry(object);
        entry.next = new AMR(succ, false);
        if (pred.next.CAS(succ, entry,
                         false, false))
            return true;
    }
}
public boolean add(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr != null) {
            return false;
        }
        Entry entry = new Entry(object);
        entry.next = new AMR(succ, false);
        if (pred.next.CAS(succ, entry, false, false)) {
            return true;
        }
    }
}

Object already there.
public boolean add(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr != null)
            return false;
        Entry entry = new Entry(object);
        entry.next = new AMR(succ, false);
        if (pred.next.CAS(succ, entry,
            false, false))
            return true;
    }
}
public boolean add(Object object) {
    Entry entry = new Entry(object);
    entry.next = new AMR(succ, false);
    if (pred.next.CAS(succ, entry, false, false))
        return true;
}

Install new entry
public boolean contains(Object obj) {
    while (true) {
        prev, curr, succ = find(object);
        return (curr != null);
    }
}
public boolean contains(Object obj) {
    while (true) {
        prev, curr, succ = find(object);
        return (curr != null);
    }
}
private Entry<Entry<Entry<Entry}
   find(Object object) {
      Entry pred, curr, succ;
      boolean[] pmark = new boolean[1];
      boolean[] cmark = new boolean[1];
      int key = object.hashCode();
      tryAgain: while (true) {
         ...
      }
   }}}}
private Entry, Entry, Entry
find(Object object) {
    Entry pred, curr, succ;
    boolean[] pmark = new boolean[1];
    boolean[] cmark = new boolean[1];
    int key = object.hashCode();
    tryAgain: while (true) {
        ...
    }
}
private Entry, Entry, Entry
find(Object object) {
    Entry pred, curr, succ;
    boolean[] pmark = new boolean[1];
    boolean[] cmark = new boolean[1];
    int key = object.hashCode(),
    tryAgain: while (true) {
        ...
    }}

Deleted bits for pred and curr
private Entry find(Object object) {
    Entry pred, curr, succ;
    boolean[] pmark = new boolean[1];
    boolean[] cmark = new boolean[1];
    int key = object.hashCode();
    tryAgain: while (true) {
        ...
    }
}

If list changes while traversed, start over
Find

private Entry, Entry, Entry
find(Object object) {
Entry pred, curr, succ;
boolean[] pmark = new boolean[1];
boolean[] cmark = new boolean[1];
int key = object.hashCode();

tryAgain: while (true) {
...
}}

Lock-Free because we start over only if someone else makes progress
Find

```java
tryAgain: while (true) {
    pred = this.head.getReference();
    curr = pred.next.get(pmark);
    while (true) {
        ...
    }
}
```

Start with first two entries
Find

tryAgain: while (true) {
    pred = this.head.getReference();
    curr = pred.next.get(pmark);
    while (true) {
        ...  
    }  
}}

Move down the list
while (true) {
    if (curr == null) {
        return pred, null, succ;
    }
    succ = curr.next.get(cmark);
    int ckey = curr.key;
    if (isChanged(pred.next))
        continue tryAgain;
}
... while (true) {
    if (curr == null)
        return pred, null, succ;
    succ = curr.next.get(cmark);
    int ckey = curr.key;
    if (isChanged(pred.next))
        continue tryAgain;
}}

Get ref to successor and current deleted bit
while (true) {
    if (curr == null)
        return pred, null, succ;
    succ = curr.next.get(cmark);
    int ckey = curr.key;
    if (isChanged(pred.next))
        continue tryAgain;
}
Find

while (true) {
  ...
  if (!cmark[0]) {
    if (curr.object == object)
      return pred, curr, succ;
    else if (ckey <= key) {
      pred = curr;
    } else
      return prev, null, curr;
  } else
    return prev, null, curr;
} else {
  ...
}}

If current node is not deleted
while (true) {
    ...
    if (!cmark[0]) {
        if (curr.object == object)
            return pred, curr, succ;
        else if (ckey <= key) {
            pred = curr;
        }
    } else {
        return prev, null, curr;
    }
    ...
}}

Object found
Find

while (true) {
    ...
    if (!cmark[0]) {
        if (curr.object == object)
            return pred, curr, succ;
        else if (ckey <= key) {
            pred = curr;
        } else
            return prev, null, curr;
    } else {
        ...
    }
}
Keep looking
Find

while (true) {
    ...
    if (!cmark[0]) {
        if (curr.object == object)
            return pred, curr, succ;
        else if (ckey <= key) {
            pred = curr;
        } else {
            return prev, null, curr;
        }
    } else {
        ...
    }
}
Find

... while (true) {
    ... if (!cmark[0]) {
        ...
    } else {
        ...}
        if (pred.next.compareAndSet(
            curr, succ, false, false))
            continue;
        else
            continue tryAgain;
    }

Current entry is logically deleted
while (true) {
    ...
    if (!cmark[0]) {
        ...
    } else {
        if (pred.next.compareAndSet(curr, succ, false, false))
            continue;
        else
            continue tryAgain;
    }
}

Try to redirect predecessor's next reference
Find

```java
while (true) {
    if (!cmark[0]) {
        continue;
    } else {
        if (pred.next.compareAndSet(curr, succ, false, false))
            continue;
        else
            continue tryAgain;
    }
}
```

On success, keep going, on failure, start over
Summary

- Coarse-grained locking
- Fine-grained locking
- Optimistic synchronization
- Lock-free synchronization
Scratch
Scratch
Scratch
Scratch
Scratch
Removing an Entry

- CAS