CH9 SUMMARY

COS 226 Chapter 9: Linked Lists - The Role of Locking

Comprehensive Study Guide

---

1. Introduction and Motivation

The Challenge of Concurrent Data Structures

In previous chapters, we learned how to build **spin locks** - mechanisms that provide mutual exclusion efficiently. A spin lock works by having threads continuously check (or "spin") until they can acquire the lock, enter their critical section, execute their code, and then release the lock upon exit.

Now we face a bigger challenge: **How do we build scalable concurrent data structures?**

The Naive Approach: Coarse-Grained Synchronization

The most obvious solution is to:

1. Take a sequential (single-threaded) implementation of a data structure
2. Add a single scalable lock
3. Make every method acquire and release this lock

This is called **coarse-grained synchronization**.

The Problem: When you use a single lock for an entire data structure, it creates a **bottleneck**. While coarse-grained synchronization works well when there's low concurrency (few threads competing), it fails when many threads try to access the data structure simultaneously. They all queue up waiting for the same lock, defeating the purpose of having multiple threads.

---

2. Four Synchronization Patterns

This chapter introduces four patterns (or "bag of tricks") for building highly-concurrent objects where more threads lead to more throughput:

Pattern 1: Fine-Grained Synchronization

Instead of one lock for the entire object, split it into independently-synchronized components. Method calls only interfere when they access the same component at the same time.

Pattern 2: Optimistic Synchronization

Search through the data structure without locking. When you find what you need, lock it and validate that it's still correct. If validation fails, start over. This is usually cheaper than locking everything, but mistakes are expensive.

Pattern 3: Lazy Synchronization

Postpone the hard work. When removing items, split the process into:

- **Logical removal**: Mark the component as deleted
- **Physical removal**: Actually remove it from the structure later

Pattern 4: Lock-Free Synchronization

Don't use locks at all. Use atomic operations like `compareAndSet()` instead.

Advantages: No assumptions about thread scheduling, no risk of deadlock
Disadvantages: More complex, sometimes higher overhead

---

3. List-Based Sets: The Example Data Structure

We'll illustrate these patterns using a **list-based Set** - an unordered collection with no duplicates.

Set Interface Methods

```
public interface Set<T> {
    public boolean add(T x);      // Returns true if x was NOT already in set
    public boolean remove(T x);   // Returns true if x WAS in set
    public boolean contains(T x); // Returns true if x is in set
}
```

Implementation Details

Node Structure:

- **item**: The actual data being stored
- **key**: The item's hash code (nodes are sorted by key for efficient searching)

- **next**: Reference to the next node in the list

List Structure:

- Uses **sentinel nodes** at head and tail (special nodes that mark boundaries and cannot be removed)
- **Regular nodes** hold actual items
- Threads traversing the list maintain two pointers:
  - **pred** (predecessor): Points to the current node's predecessor
  - **curr** (current): Points to the current node

Critical Assumptions

For our reasoning to work, we assume:

1. Only `add()`, `remove()`, and `contains()` can modify nodes
2. Sentinel nodes cannot be added or removed
3. Nodes are sorted by keys
4. Keys are unique

Safety and Liveness Properties

Safety: The implementation must be **linearizable** (operations appear to happen instantaneously at some point between their invocation and response)

Liveness: The implementation should be:

- **Deadlock-free**: The system as a whole makes progress
- **Starvation-free**: Every thread eventually makes progress
- Potentially **non-blocking**: Some thread can always make progress regardless of scheduling

---

4. Coarse-Grained Locking (Pattern 1 - Baseline)

How It Works

The entire list is protected by a single lock. Every operation:

1. Acquires the lock
2. Performs its work
3. Releases the lock

Implementation Example: `add()` Method

```java
public boolean add(T item) {
    Node pred, curr;
    int key = item.hashCode();
    lock.lock();  // ACQUIRE LOCK
    try {
        pred = head;
        curr = pred.next;
        // Traverse to find correct position
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        // Check if element already exists
        if (key == curr.key)
            return false;
        else {
            // Create and insert new node
            Node node = new Node(item);
            node.next = curr;
            pred.next = node;
            return true;
        }
    } finally {
        lock.unlock();  // RELEASE LOCK
    }
}
```

Step-by-step:

1. Compute the hash code (key) for the item
2. Acquire the lock (blocking if necessary)
3. Start at the head of the list
4. Traverse until finding the correct position (where curr.key ≥ key)
5. If the key already exists, return false
6. Otherwise, create a new node and insert it between pred and curr
7. Always release the lock in the `finally` block

Evaluation

Advantages:

- Simple to implement
- Clear and obviously correct
- Deserves respect for its simplicity

Disadvantages:

- Creates a hotspot and bottleneck
- Poor performance with high contention
- Even with efficient queue locks, the fundamental bottleneck remains
- Only one thread can make progress at a time, even if threads are working on completely different parts of the list

---

5. Fine-Grained Synchronization (Pattern 2)

The Big Idea

Instead of locking the entire list, place a lock on **each individual node**. As threads traverse the list, they lock each node during their visit and unlock it when moving on. This allows multiple threads to traverse different parts of the list simultaneously.

Hand-Over-Hand Locking

The key technique is **hand-over-hand locking** (also called **lock coupling**):

1. Lock the current node
2. Lock the next node while still holding the current lock
3. Unlock the current node
4. Move forward (the next node becomes current)
5. Repeat

Critical Rule: You must acquire the next lock BEFORE releasing the current lock. Otherwise, concurrent modifications can cause problems.

Why This Ordering Matters: The Concurrent Remove Problem

Imagine two threads trying to remove adjacent nodes:

- Thread 1: remove(b)
- Thread 2: remove(c)

If Thread 1 unlocks `a` before locking `b`, and Thread 2 unlocks `b` before locking `c`:

1. Thread 1 locks `a`, prepares to remove `b`
2. Thread 2 locks `b`, prepares to remove `c`
3. Thread 1 removes `b` (makes `a` point to `c`)
4. Thread 2 removes `c` (makes `b` point to `d`)
5. **Problem:** Now `a` points to `c`, but `c` has been removed! The list is corrupted.

The Solution: Hand-Over-Hand

With hand-over-hand locking:

1. Thread 1 locks `a`, then locks `b` while holding `a`'s lock
2. Thread 2 tries to lock `b` for removing `c`, but must wait because Thread 1 holds `b`'s lock
3. Thread 1 completes removing `b`, releases both locks
4. Thread 2 can now proceed safely

Why It Works:

- To delete node `e`, you must lock both `e` and its predecessor
- Therefore, if you lock a node, it cannot be removed
- And neither can its successor (because you'd need to lock the current node first)

Implementation: `remove()` Method

```java
public boolean remove(Item item) {
    int key = item.hashCode();
    Node pred, curr;
    try {
        pred = head;
        pred.lock();        // Lock head
        curr = pred.next;
        curr.lock();        // Lock next

        // Search for the key
        while (curr.key <= key) {
            if (item == curr.item) {
                // Found it! Remove the node
                pred.next = curr.next;
                return true;
            }
            // Hand-over-hand: unlock pred, move forward, lock new curr
            pred.unlock();
            pred = curr;
            curr = curr.next;
            curr.lock();
        }
        return false;  // Not found
    } finally {
        curr.unlock();
        pred.unlock();
    }
}
```

At the start of each loop iteration:

- Both `curr` and `pred` are locked

- This ensures no other thread can modify them

Adding Nodes

Adding nodes follows the same pattern:

1. Traverse with hand-over-hand locking
2. Find the correct position (where new key should go)
3. Create the new node while holding locks on pred and curr
4. Insert it between them
5. Release locks

Lock Ordering and Deadlock

Question: Does it matter if threads acquire locks in the same order?

Answer: Yes! Consider:

- Thread 1: `add(a)` - acquires locks in order: head → b → c → d
- Thread 2: `remove(b)` - acquires locks in order: head → b → c

If these threads could acquire locks in arbitrary orders, deadlock could occur. However, because all threads traverse from head to tail and use hand-over-hand locking, they naturally acquire locks in the same order, preventing circular wait and thus preventing deadlock.

Evaluation

Advantages:

- Better than coarse-grained locking
- Multiple threads can work on different parts of the list simultaneously
- More scalable with moderate contention

Disadvantages:

- Still requires a potentially long sequence of lock acquisitions and releases
- Blocking: A thread removing the second item blocks all threads trying to access later nodes
- Every traversal requires acquiring and releasing many locks
- Performance overhead from all the locking operations

---

6. Optimistic Synchronization (Pattern 3)

The Core Insight

What if we take a chance? Instead of locking everything as we go, we:

1. **Search without locks** (optimistically assuming nothing will go wrong)
2. **Lock the nodes we found**
3. **Validate** that the locked nodes are still correct
4. If validation fails, start over

This is called "optimistic" because we optimistically hope that concurrent modifications won't invalidate our work.

What Could Go Wrong?

Problem 1: Node Removed During Traversal

Imagine we're adding `c` between `b` and `e`:

1. We traverse without locks and find that `c` should go between `b` and `e`
2. Before we lock anything, another thread removes `b`
3. We lock `b` and `e`, but `b` is no longer in the list!
4. If we insert `c` after `b`, it won't be reachable from head

Problem 2: Node Inserted During Traversal

Imagine we're adding `c`:

1. We traverse and decide `c` should go between `b` and `e`
2. Before we lock, another thread inserts `d` between `b` and `e`
3. We lock `b` and `e`
4. We insert `c`, but now the order is wrong (should be b → c → d → e, but we create b → c → e with d lost)

The Solution: Two-Part Validation

Validation Part 1: Verify that `pred` is still reachable from head

- Traverse from head without locks
- Check if we can reach `pred`
- If not, `pred` was removed, and we must start over

Validation Part 2: Verify that `pred` still points to `curr`

- While holding locks on both nodes
- Check if `pred.next == curr`
- If not, something was inserted or removed between them

Implementation: `validate()` Method

```java
private boolean validate(Node pred, Node curr) {
    Node node = head;
    // Traverse from head to see if pred is reachable
    while (node.key <= pred.key) {
        if (node == pred)
            // Found pred! Now check if it points to curr
            return pred.next == curr;
        node = node.next;
    }
    return false;  // pred not reachable
}
```

Implementation: `remove()` Method

```java
public boolean remove(Item item) {
    int key = item.hashCode();
    Node pred = head;
    Node curr = pred.next;

    // Search WITHOUT locking
    while (curr.key <= key) {
        if (item == curr.item)
            break;
        pred = curr;
        curr = curr.next;
    }

    // Now lock and validate
    try {
        pred.lock();
        curr.lock();
        if (validate(pred, curr)) {
            if (curr.item == item) {
                pred.next = curr.next;
                return true;
            } else {
                return false;
            }
        }
        // Validation failed - must retry (not shown)
    } finally {
        pred.unlock();
        curr.unlock();
    }
}
```

The `contains()` Method Challenge

For `contains()`, we have three options:

Option 1: Coarse-Grained

- Acquire the global lock
- Search the list
- Release the lock
- Simple but defeats the purpose of optimistic synchronization

Option 2: Fine-Grained

- Use hand-over-hand locking
- Lots of lock operations for a read-only operation

Option 3: Optimistic

```
public boolean contains(T item) {
    int key = item.hashCode();
    Node pred = head;
    Node curr = pred.next;

    // Traverse without locking
    while (curr.key < key) {
        pred = curr;
        curr = curr.next;
    }

    // Lock and validate
    try {
        pred.lock();
        curr.lock();
        if (validate(pred, curr))
            return (curr.key == key);
    } finally {
        pred.unlock();
        curr.unlock();
    }
}
```

Evaluation

Advantages:

- Limited hot-spots (only at modification points)
- No contention during the traversal phase
- Traversals are wait-free initially

- Usually cheaper than fine-grained locking

Disadvantages:

- Need to traverse the list twice (once to search, once to validate)
- `contains()` still requires locks
- Mistakes (validation failures) are expensive - must restart
- With high contention, many retries may be needed

---

7. Lazy Synchronization (Pattern 4)

The Key Insight

In real-world applications, `contains()` is typically called much more frequently than `add()` or `remove()`. Can we make `contains()` even faster?

The Big Idea: Logical vs. Physical Removal

Add a **marked** boolean field to each node:

- **marked = false**: Node is logically in the list
- **marked = true**: Node is logically removed (even if physically still present)

This separation allows:

1. **Logical removal**: Just set the marked flag (fast, simple)
2. **Physical removal**: Actually unlink the node (can be done lazily)

Why This Helps

Key Principle: Every unmarked node is reachable from head (by design and enforcement).

Therefore:

- No need to validate reachability during traversal
- If a node is unmarked, it's definitely in the list
- If a node is marked or not found, it's not in the list

Simplified Validation

```java
private boolean validate(Node pred, Node curr) {
    return !pred.marked &&      // Pred not logically removed
           !curr.marked &&      // Curr not logically removed
           pred.next == curr;   // Pred still points to curr
}
```

No traversal needed! Just check three simple conditions.

Wait-Free `contains()`

```java
public boolean contains(Item item) {
    int key = item.hashCode();
    Node curr = head;

    // Traverse without locking
    while (curr.key < key) {
        curr = curr.next;
    }

    // Simply check key and marked field - NO LOCKING!
    return curr.key == key && !curr.marked;
}
```

This is wait-free: No matter what other threads do, `contains()` completes in a bounded number of steps without any blocking.

The `remove()` Method: Two-Phase Removal

```java
public boolean remove(T item) {
    int key = item.hashCode();
    Node pred = head;
    Node curr = pred.next;

    // Traverse without locking
    while (curr.key < key) {
        pred = curr;
        curr = curr.next;
    }

    try {
        pred.lock();
        curr.lock();
        if (validate(pred, curr)) {
            if (curr.key == key) {
                // PHASE 1: Logical removal
                curr.marked = true;
                // PHASE 2: Physical removal
                pred.next = curr.next;
                return true;
            } else {
                return false;
            }
        }
    } finally {
        pred.unlock();
        curr.unlock();
    }
}
```

Phase 1 (Logical): Mark the node as removed. From this point, `contains()` will return false for this item.

Phase 2 (Physical): Unlink the node from the list. This is "lazy" because it happens after the logical removal.

The `add()` Method

The `add()` method is essentially the same as in optimistic synchronization:

1. Traverse without locks
2. Lock pred and curr
3. Validate (using the simpler validation)
4. Insert if not already present

Evaluation

Advantages:

- `contains()` is wait-free (never locks, never blocks)
- Perfect for workloads with many `contains()` calls
- Uncontended `add()` and `remove()` traverse only once (no validation traversal)
- Simpler validation than optimistic

Disadvantages:

- Contended `add()` and `remove()` may need to retry
- Still vulnerable to "traffic jam" problem: if one thread gets delayed while holding locks (cache miss, page fault, preemption), all other threads waiting for those locks are stuck

---

8. The Traffic Jam Problem

The Fundamental Weakness of Lock-Based Approaches

Any concurrent data structure based on mutual exclusion has this weakness:

If one thread:

1. Enters a critical section
2. Experiences a major delay ("eats the big muffin"):

    - Cache miss (data not in CPU cache, must fetch from RAM)
    - Page fault (data not in RAM, must fetch from disk)
    - Gets descheduled by the OS
    - Hardware interrupt

Then everyone else waiting for that lock is stuck!

This means we must trust the scheduler to be fair and not delay threads at inopportune times.

---

9. Lock-Free Synchronization (Pattern 5)

The Ultimate Goal

Create a data structure where:

- **No locks are used at all**
- **Guaranteed progress:** Some thread will always complete a method call, even if others halt at malicious times
- **No scheduler assumptions:** Works correctly regardless of how the OS schedules threads

The Tool: `compareAndSet()` (CAS)

The atomic operation `compareAndSet()` is the foundation:

```
boolean compareAndSet(expectedValue, newValue)
```

Atomically:

1. If current value equals `expectedValue`
2. Then set it to `newValue` and return true
3. Otherwise, do nothing and return false

This is a hardware-level atomic operation - no other thread can interfere.

First Attempt: Using CAS for Removal

Idea: To remove `c` from a → b → c → e, use CAS to change b's next field from `c` to `e`.

Why this doesn't work:

Consider this scenario:

1. Thread 1: `remove(a)` - prepares to change head.next from `a` to `b`

2. Thread 2: `add(b)` - prepares to insert `b` between `a` and `c`
3. Thread 1 executes CAS on head.next (a → b)
4. Thread 2 executes CAS on a.next (c → b)
5. Result: head → b, but `a` is orphaned! And b → c (wrong order)

The Core Problem: We need to ensure that a node's fields cannot be updated after that node has been removed from the list.

The Solution: AtomicMarkableReference

Java provides `AtomicMarkableReference<T>` which combines:

- A reference (pointer) to an object
- A boolean mark (flag)

Both can be updated **atomically together** using CAS!

```
[address | mark bit]
```

The `compareAndSet()` Method for AtomicMarkableReference

```
public boolean compareAndSet(
    Object expectedRef,     // Expected current reference
    Object updateRef,       // New reference value
    boolean expectedMark,   // Expected current mark
    boolean updateMark)     // New mark value
```

Atomically:

- If the current reference is `expectedRef` AND
- The current mark is `expectedMark`
- Then change to `updateRef` and `updateMark`
- Return true if successful, false otherwise

Two-Step Removal Process

Step 1: Mark the node

```
curr.next.attemptMark(succ, true)
```

This atomically sets the mark bit to `true`, indicating the node is logically removed. Any subsequent attempt to modify `curr.next` will fail because the mark bit has changed.

Step 2: Physical removal

```
pred.next.compareAndSet(curr, succ, false, false)
```

This physically unlinks the node from the list.

Concurrent Removal Example

Thread 1: `remove(c)`
Thread 2: `remove(b)`

1. Thread 1 marks `c` using CAS
2. Thread 2 tries to mark `b`, but Thread 2 also needs to update b's next pointer
3. Thread 1 tries to CAS pred.next (b → d), but Thread 2 has marked `b`
4. Thread 1's CAS fails because `b` is marked
5. Thread 2 successfully removes `b`
6. Thread 1 sees that `c` is no longer reachable in the way expected and handles it appropriately

Key insight: Once marked, a node is "protected" from having its next pointer modified.

Shared Physical Removal

In the lock-free approach, physical removal is **cooperative**:

- **All threads** calling `add()` or `remove()` help clean up the list
- As each thread traverses, it physically removes any marked nodes it encounters
- This is done using CAS operations

Why `contains()` doesn't remove nodes:
`contains()` traverses all nodes (marked or not) without removing them. It's still wait-free because it doesn't block, but it doesn't help with cleanup.

The `find()` Method

This is the workhorse method that both `add()` and `remove()` use:

```java
public Window find(Node head, int key) {
    boolean[] marked = {false};
    boolean snip;
    retry: while (true) {
        pred = head;
        curr = pred.next.getReference();
        while (true) {
            succ = curr.next.get(marked);  // Get next and mark status

            // Clean up marked nodes
            while (marked[0]) {
                snip = pred.next.CAS(curr, succ, false, false);
                if (!snip)
                    continue retry;  // CAS failed, start over
                curr = succ;
                succ = curr.next.get(marked);
            }

            if (curr.key >= key)
                return new Window(pred, curr);
            pred = curr;
            curr = succ;
        }
    }
}
```

What it does:

1. Traverses the list to find the position for `key`
2. Removes any marked nodes it encounters (using CAS)
3. If a CAS fails (someone else modified the list), retry from the beginning
4. Returns a "window" (pred, curr) where:
   - pred.key < key
   - curr.key ≥ key
   - Both nodes are unmarked and consecutive

The `add()` Method

```java
public boolean add(T item) {
    int key = item.hashCode();
    while (true) {
        Window window = find(head, key);  // Find position and clean up
        Node pred = window.pred;
        Node curr = window.curr;

        if (curr.key == key)
            return false;  // Already in list
        else {
            Node node = new Node(item);
            node.next = new AtomicMarkableReference(curr, false);

            // Try to CAS the new node into place
            if (pred.next.compareAndSet(curr, node, false, false))
                return true;
            // CAS failed, retry
        }
    }
}
```

Process:

1. Find the correct position (and clean up marked nodes)
2. If the item already exists, return false
3. Create a new node pointing to curr
4. Use CAS to insert it: if pred still points to curr and both are unmarked, change pred to point to node
5. If CAS fails (someone modified the list), retry

The `remove()` Method

```java
public boolean remove(T item) {
    int key = item.hashCode();
    boolean snip;
    while (true) {
        Window window = find(head, key);  // Find position and clean up
        Node pred = window.pred;
        Node curr = window.curr;

        if (curr.key != key)
            return false;  // Not in list
        else {
            Node succ = curr.next.getReference();

            // Try to mark the node as logically removed
            snip = curr.next.attemptMark(succ, true);
            if (!snip)
                continue;  // Failed, retry

            // Try to physically remove (but don't worry if it fails)
            pred.next.compareAndSet(curr, succ, false, false);
            return true;
        }
    }
}
```

Process:

1. Find the node to remove (and clean up)
2. If not found, return false
3. Mark the node atomically
4. If marking fails, retry (someone else modified it)
5. Attempt physical removal (if this fails, it's okay - another thread will do it during their traversal)

Why Physical Removal Failure is Okay

The key insight: Once a node is marked, it's logically removed. The physical removal is an optimization. If our CAS to physically remove fails, some other thread will remove it when they traverse that part of the list.

Answering Key Questions

Q: Why can't threads just traverse without removing marked nodes?
A: Because marked nodes accumulate and slow down traversals. Also, we need to ensure the "window" (pred, curr) returned by `find()` is accurate - both nodes must be unmarked and consecutive.

Q: Why aren't nodes removed immediately after marking?
A: The marking is the critical operation (it's the linearization point). Physical removal is cleanup that can happen asynchronously. This separation allows for better concurrency - a thread doesn't need to wait for physical removal to complete its operation.

Evaluation

Advantages:

- **Non-blocking:** Some thread always makes progress
- **No scheduler assumptions:** Works correctly even with adversarial scheduling
- **No deadlock possible:** No locks to create circular dependencies
- **Highly concurrent:** Multiple threads can modify different parts simultaneously

Disadvantages:

- **Complex:** Much harder to understand and implement correctly
- **Performance cost:** Atomic operations on references+marks have overhead
- **Extra work:** `add()` and `remove()` must clean up marked nodes during traversal
- **ABA problem potential:** (not discussed in detail, but a concern with CAS-based algorithms)

---

10. Comparison of All Approaches

Coarse-Grained Locking

- **Simplicity:** ⭐⭐⭐⭐⭐ (Easiest to implement and understand)
- **Performance:** ⭐ (Bottleneck with high contention)
- **Scalability:** ⭐ (One lock limits concurrency)
- **Blocking:** Yes (complete lock)

Fine-Grained Locking

- **Simplicity:** ⭐⭐⭐ (Hand-over-hand adds complexity)
- **Performance:** ⭐⭐ (Better than coarse, but many lock operations)
- **Scalability:** ⭐⭐⭐ (Multiple threads can work on different sections)
- **Blocking:** Yes (locks each node)

Optimistic Synchronization

- **Simplicity:** ⭐⭐ (Validation adds complexity)
- **Performance:** ⭐⭐⭐ (Good with low contention)
- **Scalability:** ⭐⭐⭐⭐ (Traverse without locks)
- **Blocking:** Yes (locks nodes for modification)
- **Special:** Traversals are wait-free until validation

Lazy Synchronization

- **Simplicity:** ⭐⭐ (Marked field and two-phase removal)
- **Performance:** ⭐⭐⭐⭐ (Excellent for read-heavy workloads)
- **Scalability:** ⭐⭐⭐⭐⭐ (Wait-free contains())
- **Blocking:** Yes (locks for add/remove)
- **Special:** Best choice if contains() dominates

Lock-Free Synchronization

- **Simplicity:** ⭐ (Most complex)
- **Performance:** ⭐⭐⭐⭐ (Good, but atomic operations have overhead)
- **Scalability:** ⭐⭐⭐⭐⭐ (Truly non-blocking)
- **Blocking:** No (lock-free)
- **Special:** Guaranteed progress, no scheduler assumptions

---

11. Summary Table: Optimistic vs. Lazy

Method	Optimistic Synchronization	Lazy Synchronization
contains()	Traverses without locks, then locks pred and curr, validates, returns result	Traverses without locks, checks marked field only - **wait-free**
validate()	Traverses from head to check if node is reachable and curr follows pred	Simply checks marked fields and pred.next == curr - **no traversal**
add()	Traverses without locks, locks pred and curr, validates, adds if valid	Same as optimistic
remove()	Traverses without locks, locks pred and curr, validates, removes if valid	Same but also sets marked field before physical removal

12. Key Takeaways

1. **Coarse-grained locking** is simple and correct but creates bottlenecks with high concurrency
2. **Fine-grained locking** improves scalability by allowing concurrent access to different parts of the data structure, but requires careful hand-over-hand locking protocol
3. **Optimistic synchronization** reduces lock contention by validating after the fact, trading occasional retries for better average-case performance
4. **Lazy synchronization** separates logical and physical removal, enabling wait-free reads - ideal for read-heavy workloads
5. **Lock-free synchronization** eliminates locks entirely using atomic operations, providing guaranteed progress but at the cost of implementation complexity
6. The choice of synchronization strategy depends on:
   - Expected workload (read-heavy vs. write-heavy)
   - Contention levels
   - Performance requirements
   - Complexity tolerance
7. All lock-based approaches suffer from the **traffic jam problem** - a delayed thread holding locks can block all other threads
8. **Lock-free approaches** solve the traffic jam problem but introduce their own complexity and overhead through cooperative cleanup and retry logic