Lecture 32: Lazy Linked Lists

COSC 273: Parallel and Distributed Computing

Spring 2023

Annoucements

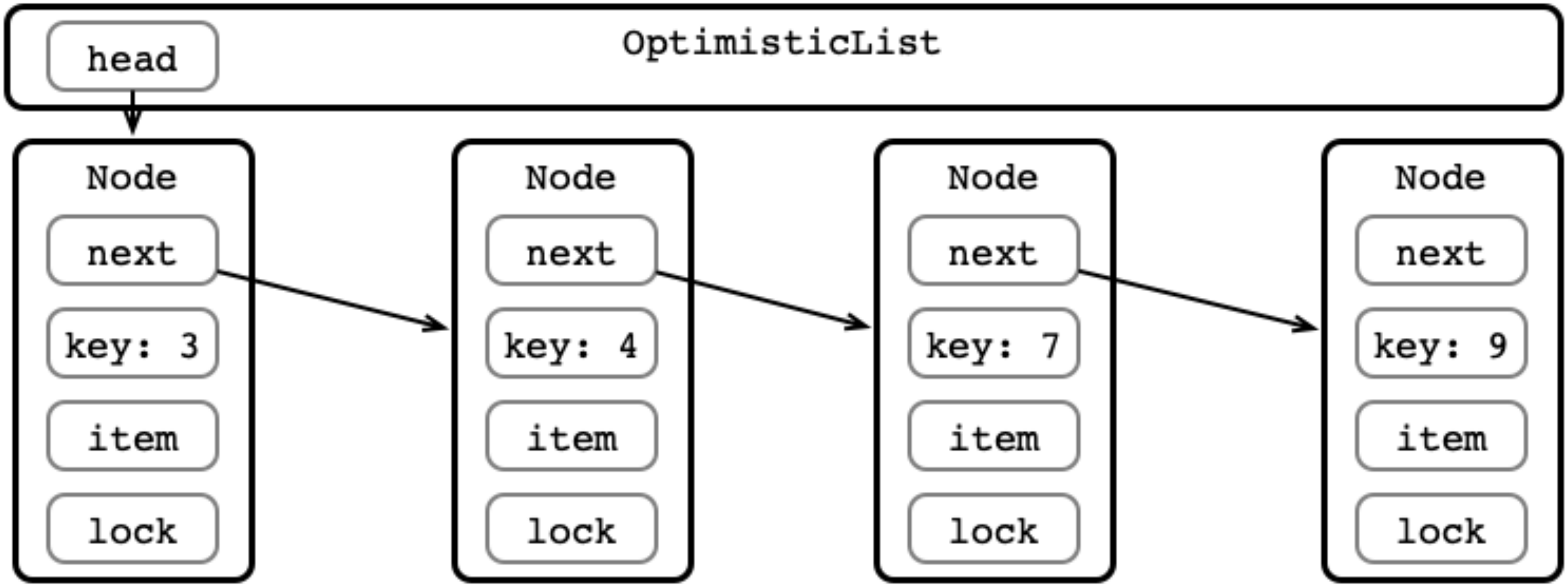
1. Quiz on concurrent linked lists released today, due Friday
2. Next leaderboard submission on Monday
3. First leaderboad results soon

Last Time

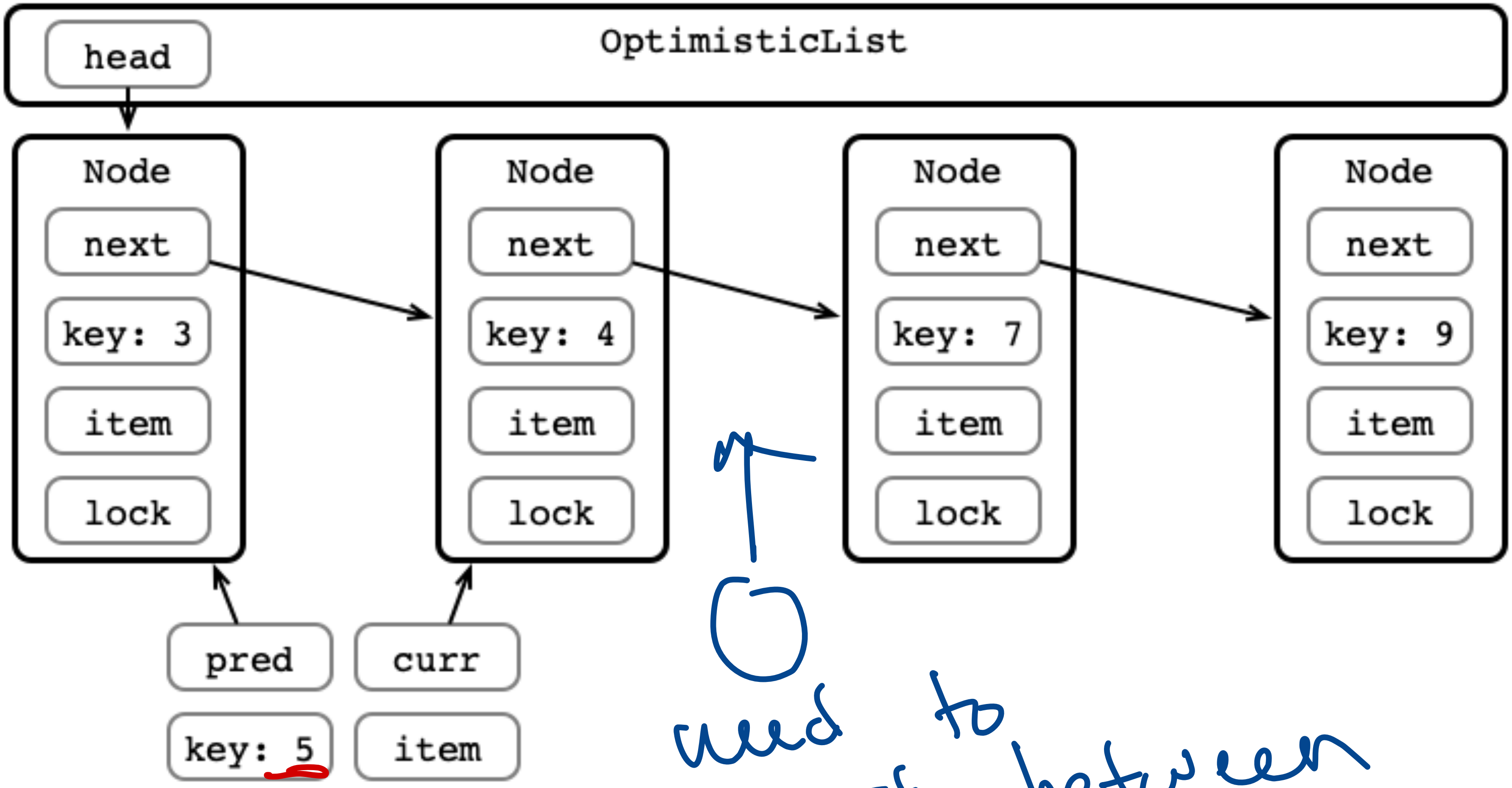
Concurrent Linked Lists, Three Ways:

1. Coarse locking
    - lock the whole data structure for every operation
2. Fine-grained locking
    - lock individual nodes to avoid conflicts
3. Optimistic locking
    - search without locks, lock on find, then validate

Optimistic Insertion

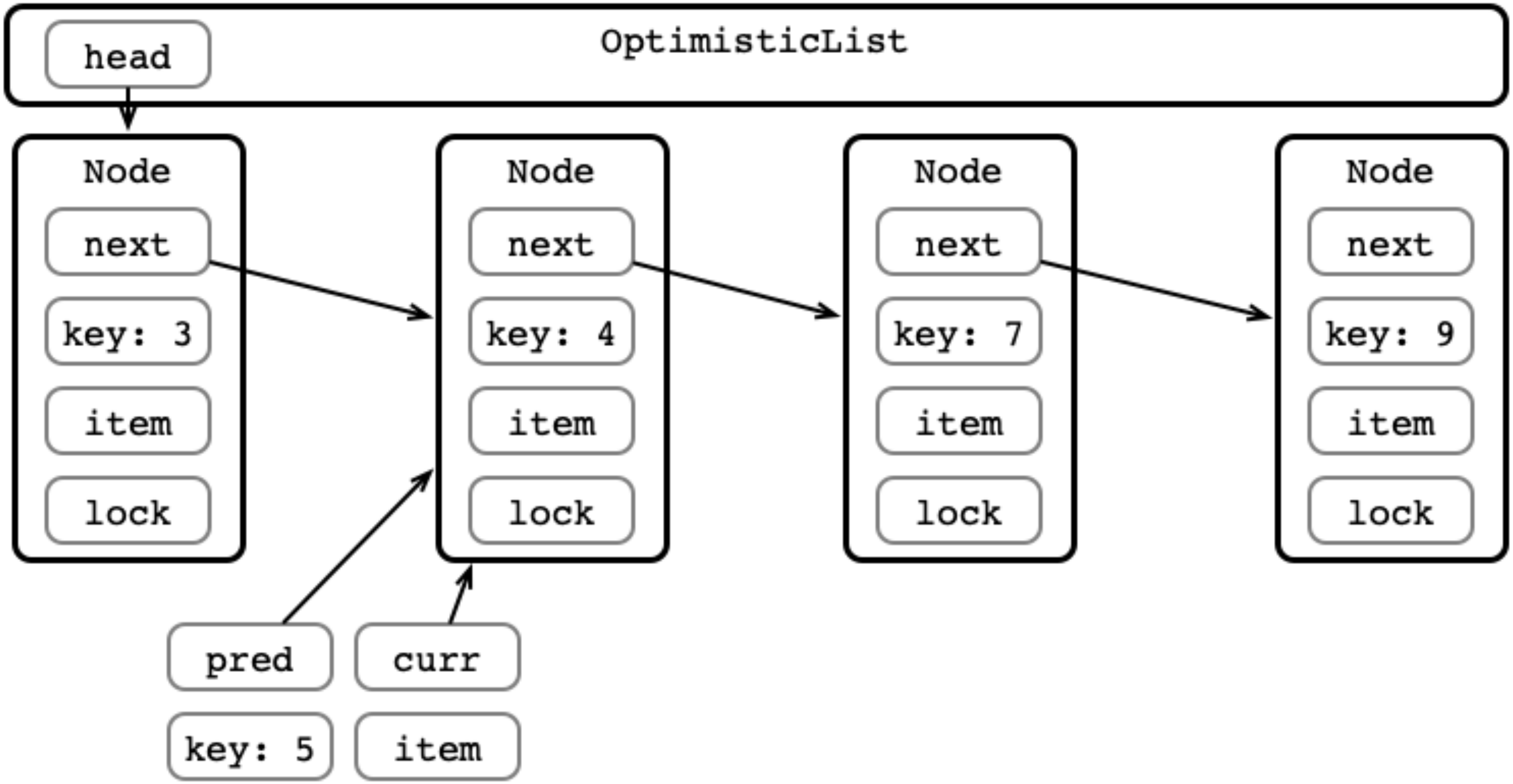


Step 1: Traverse the List

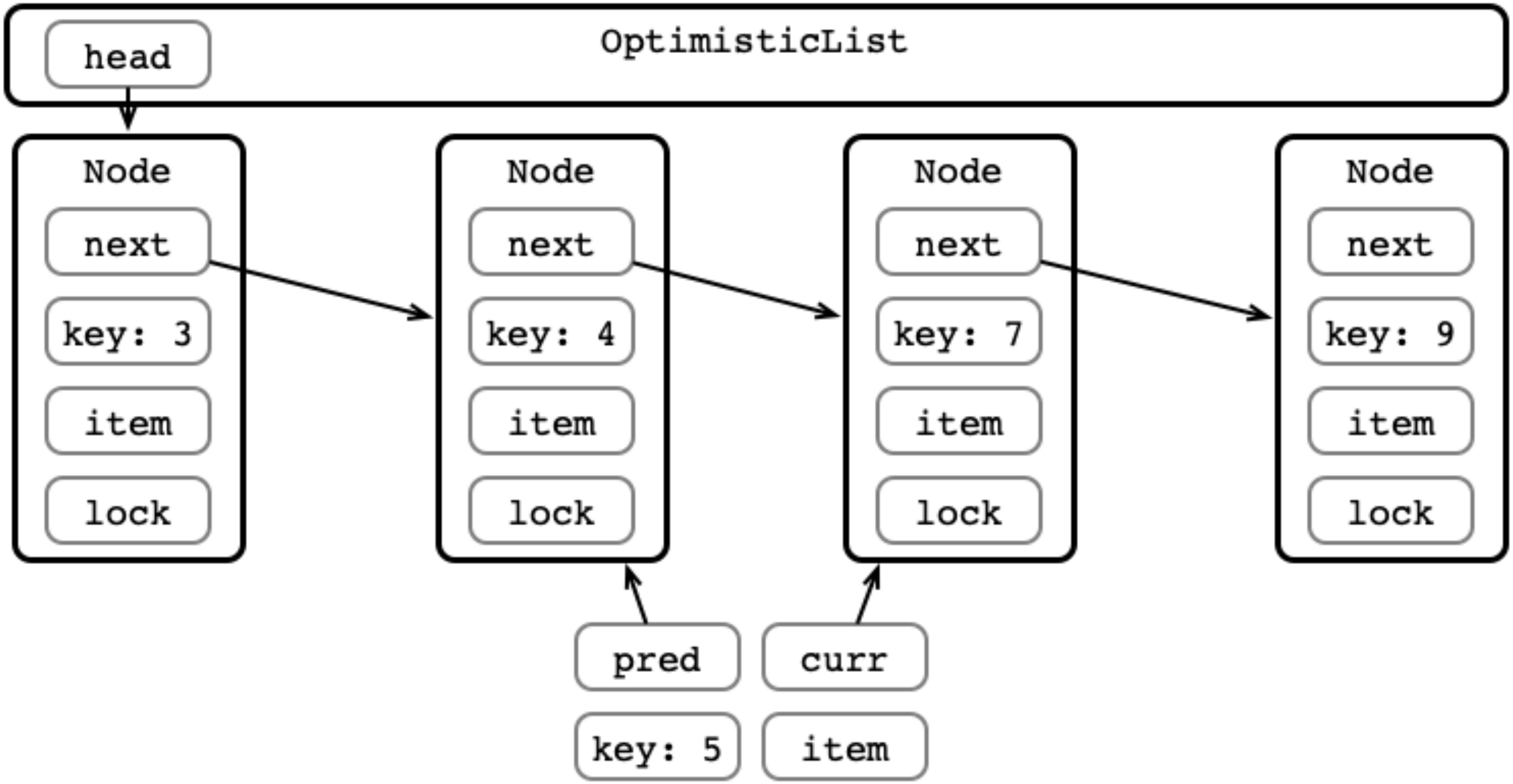


need to insert between these nodes

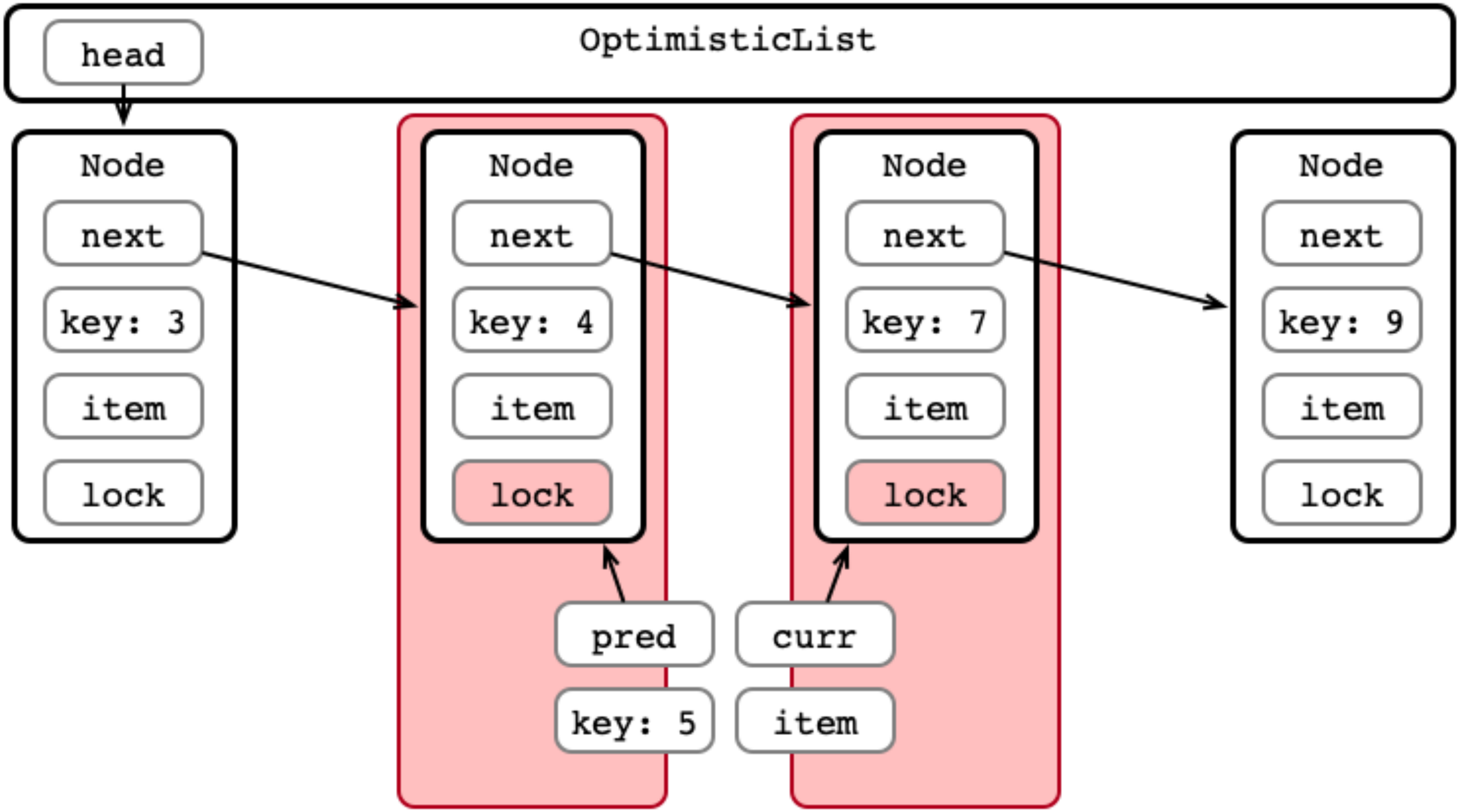
Step 1: Traverse the List



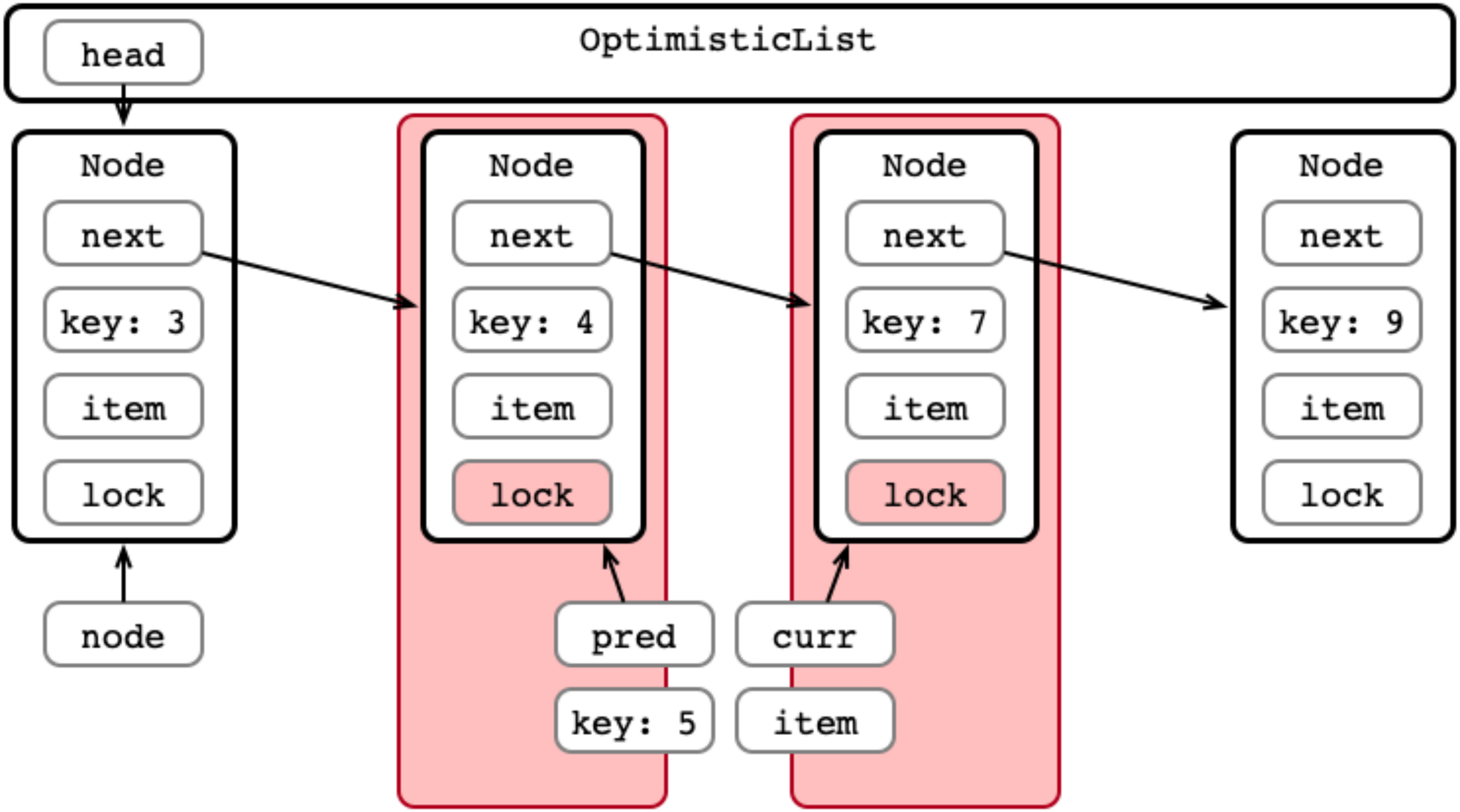
Step 1: Traverse the List



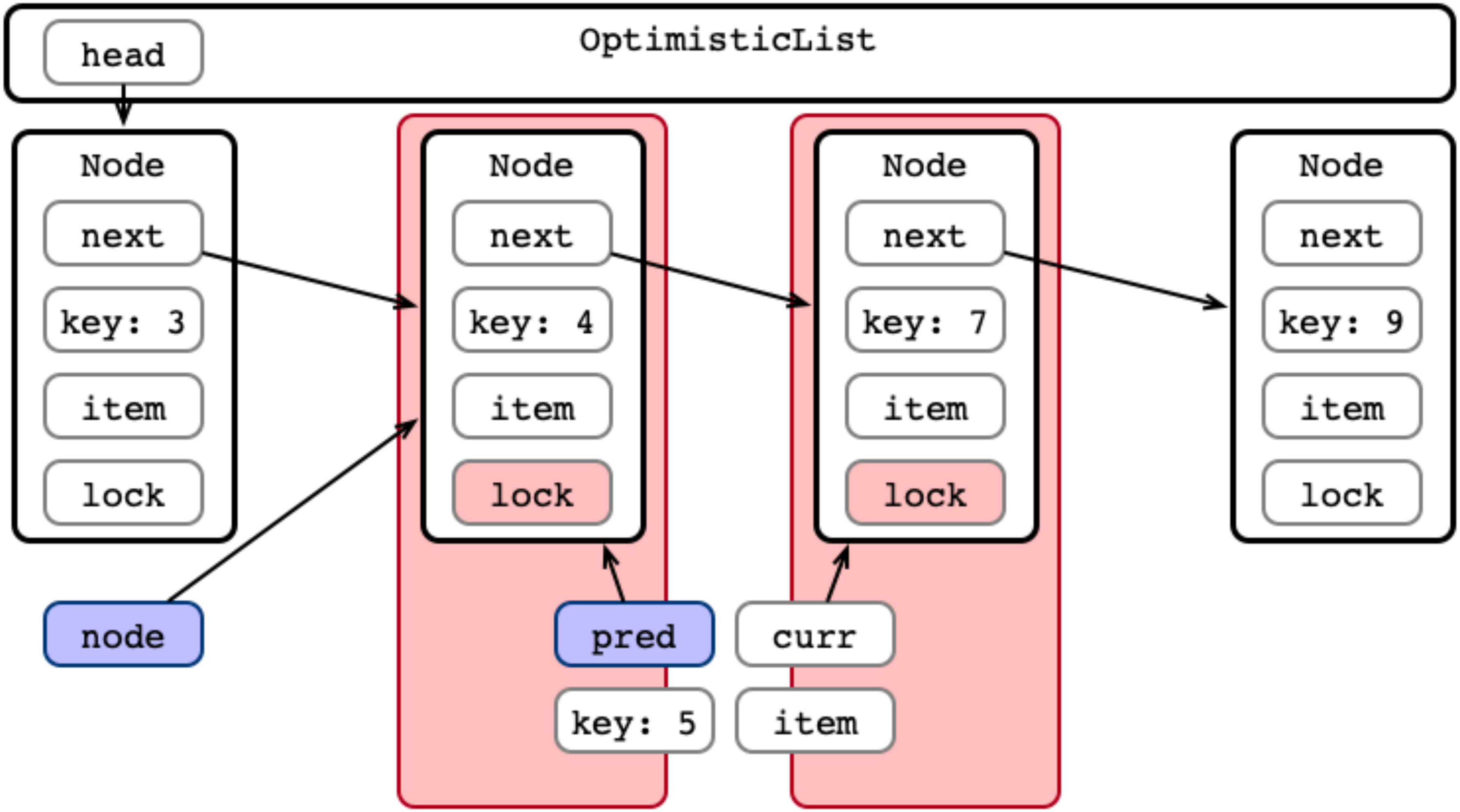
Step 2: Acquire Locks



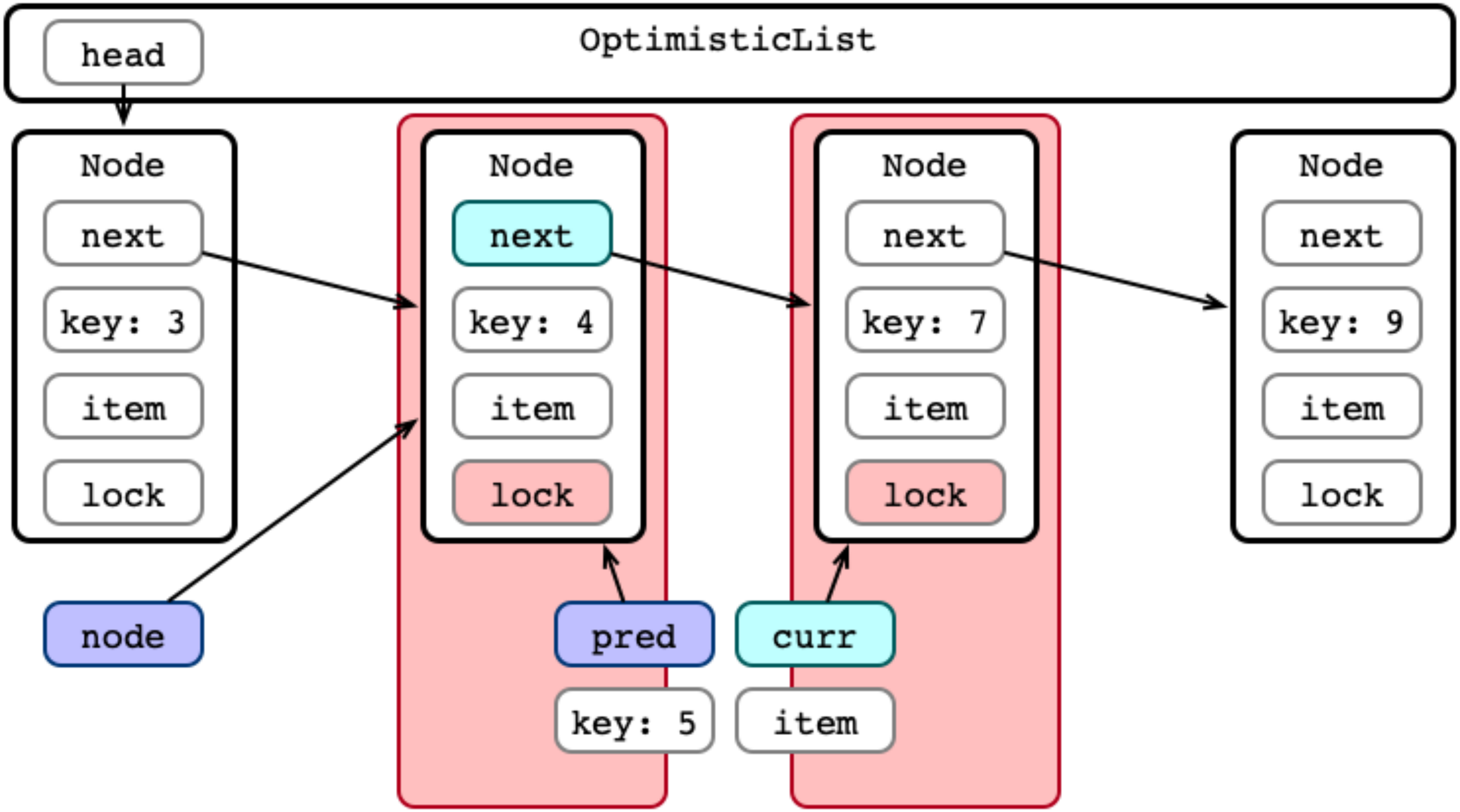
Step 3: Validate List - Traverse



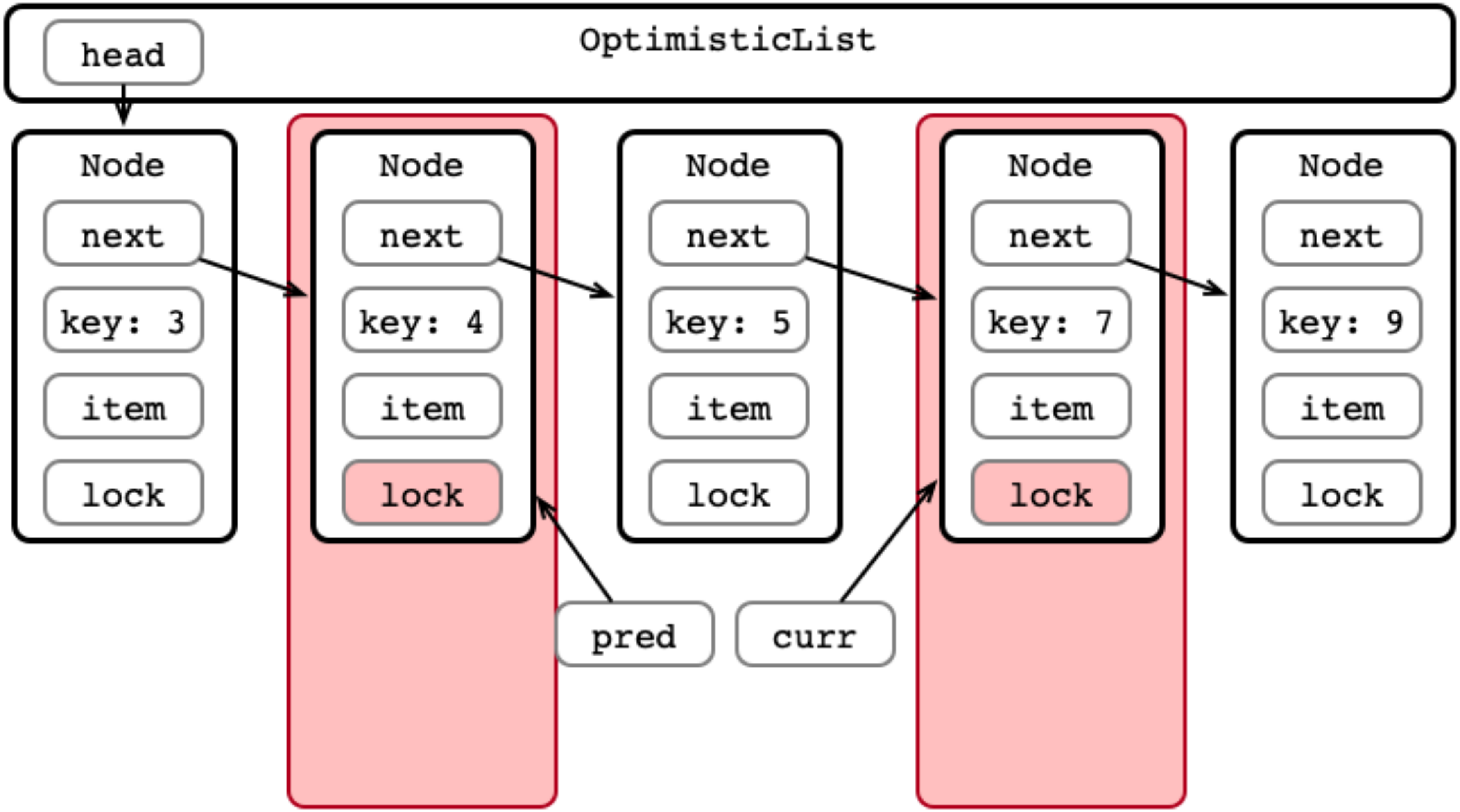
Step 3: Validate List - pred Reachable?



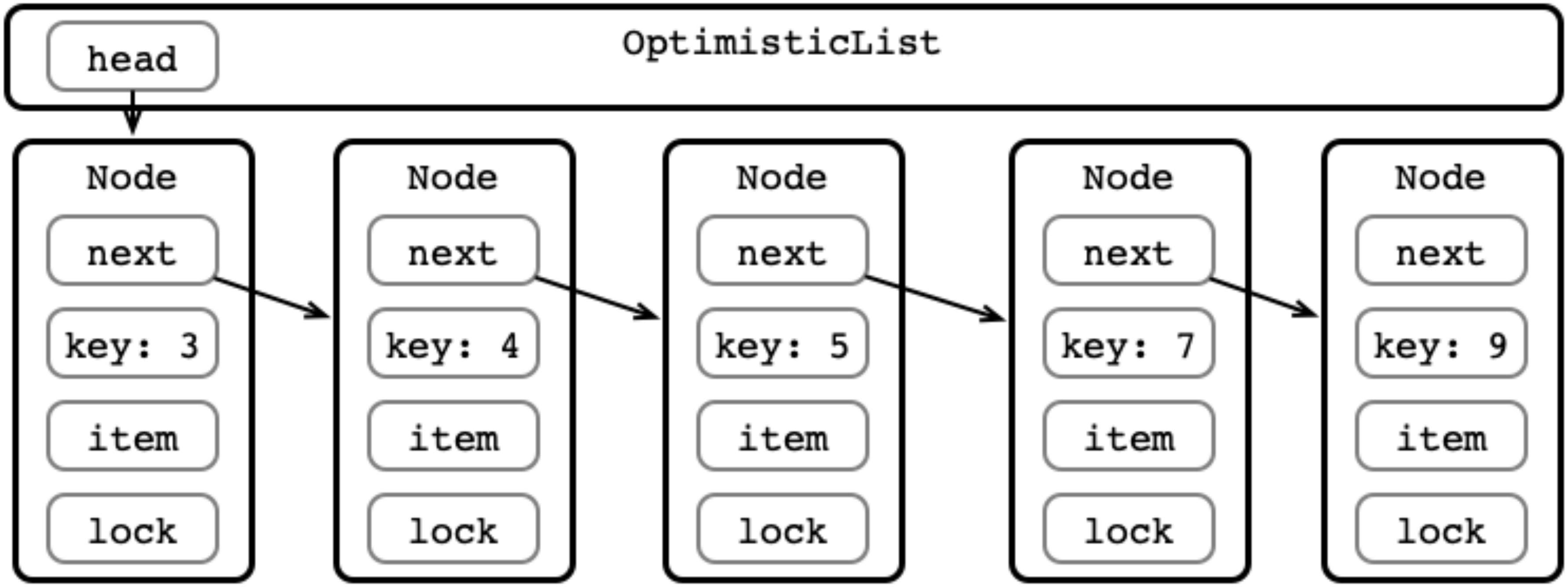
Step 3: Validate List - Is curr next?



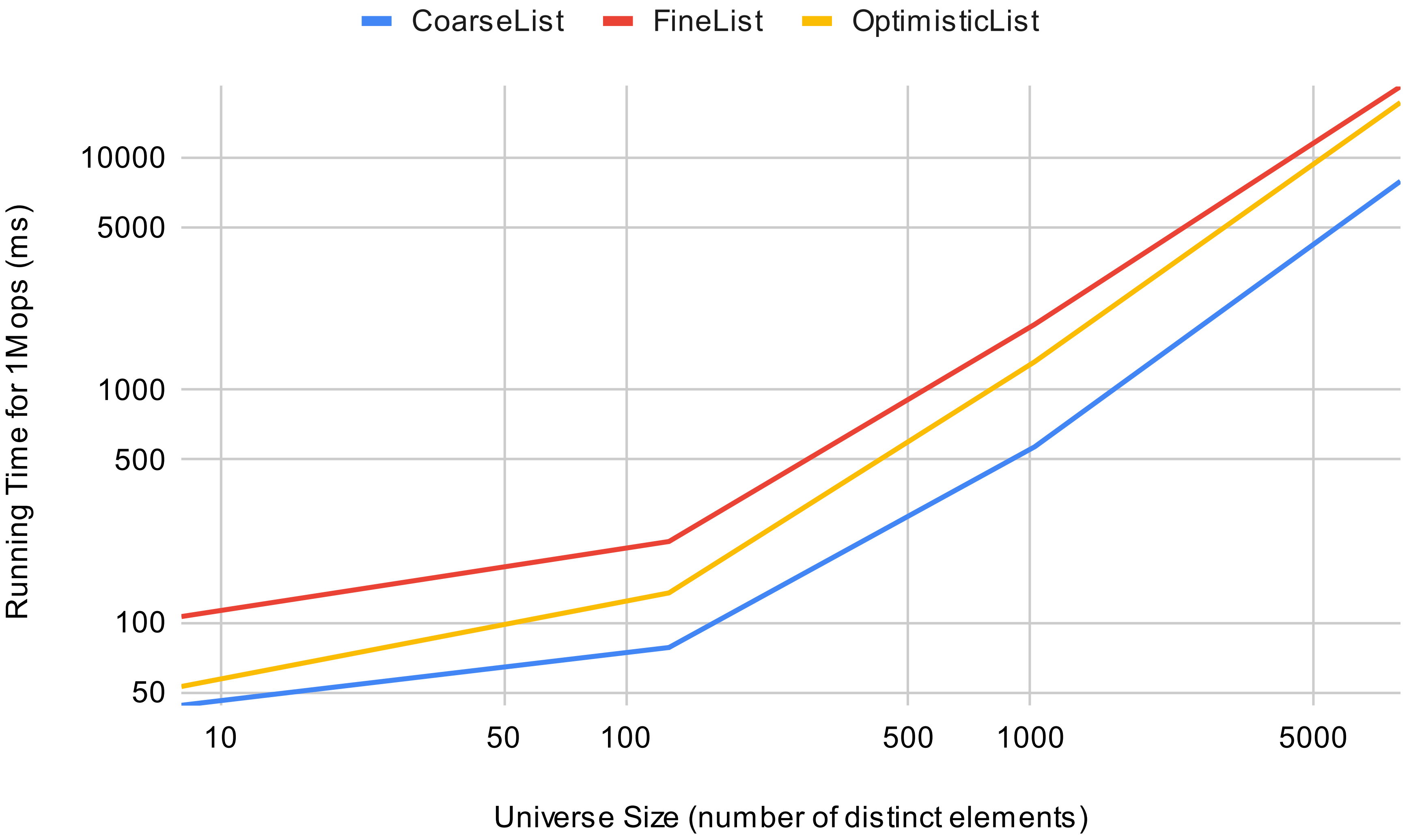
Step 4: Perform Insertion



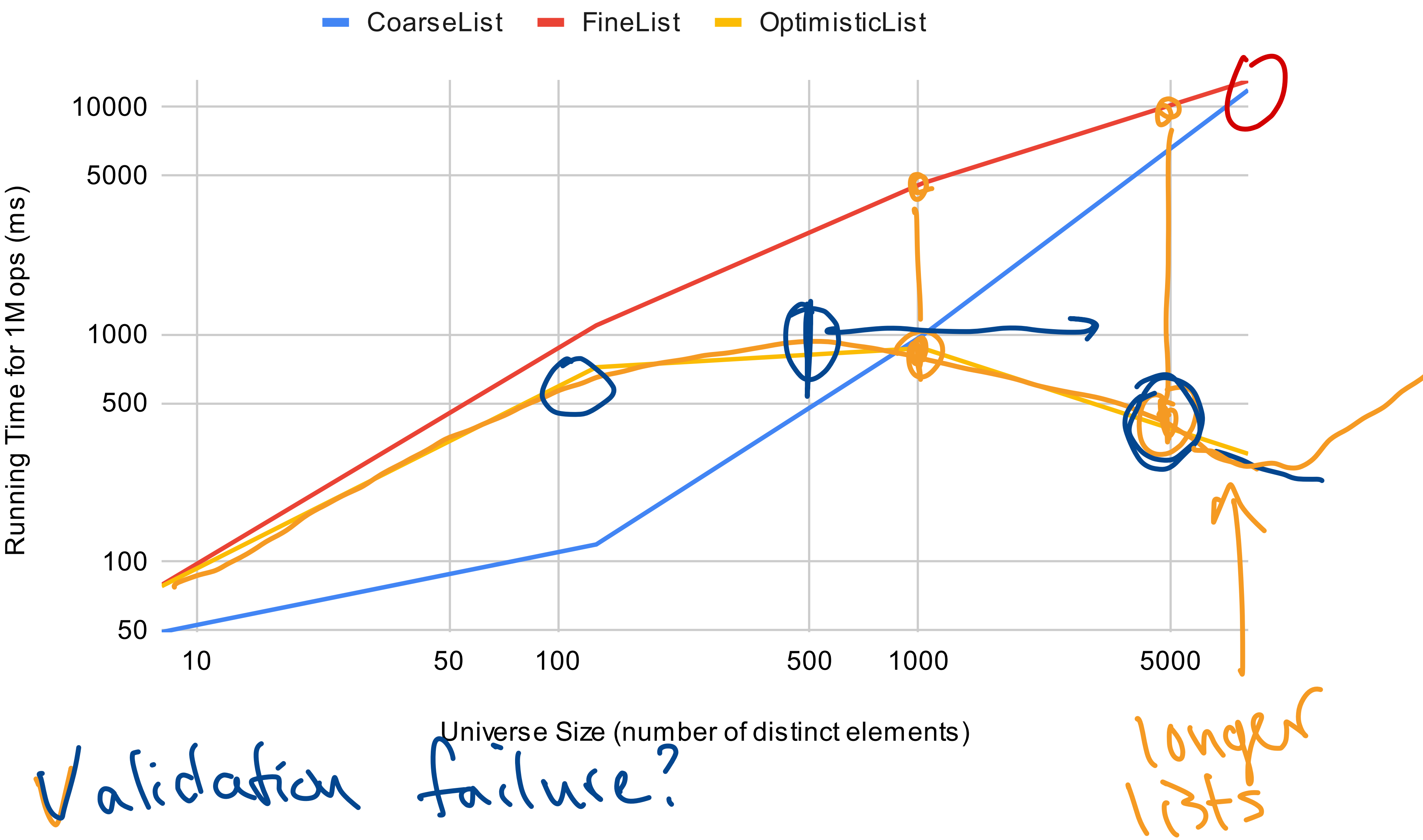
Step 5: Release Locks



Performance v. Size, 1 Thread

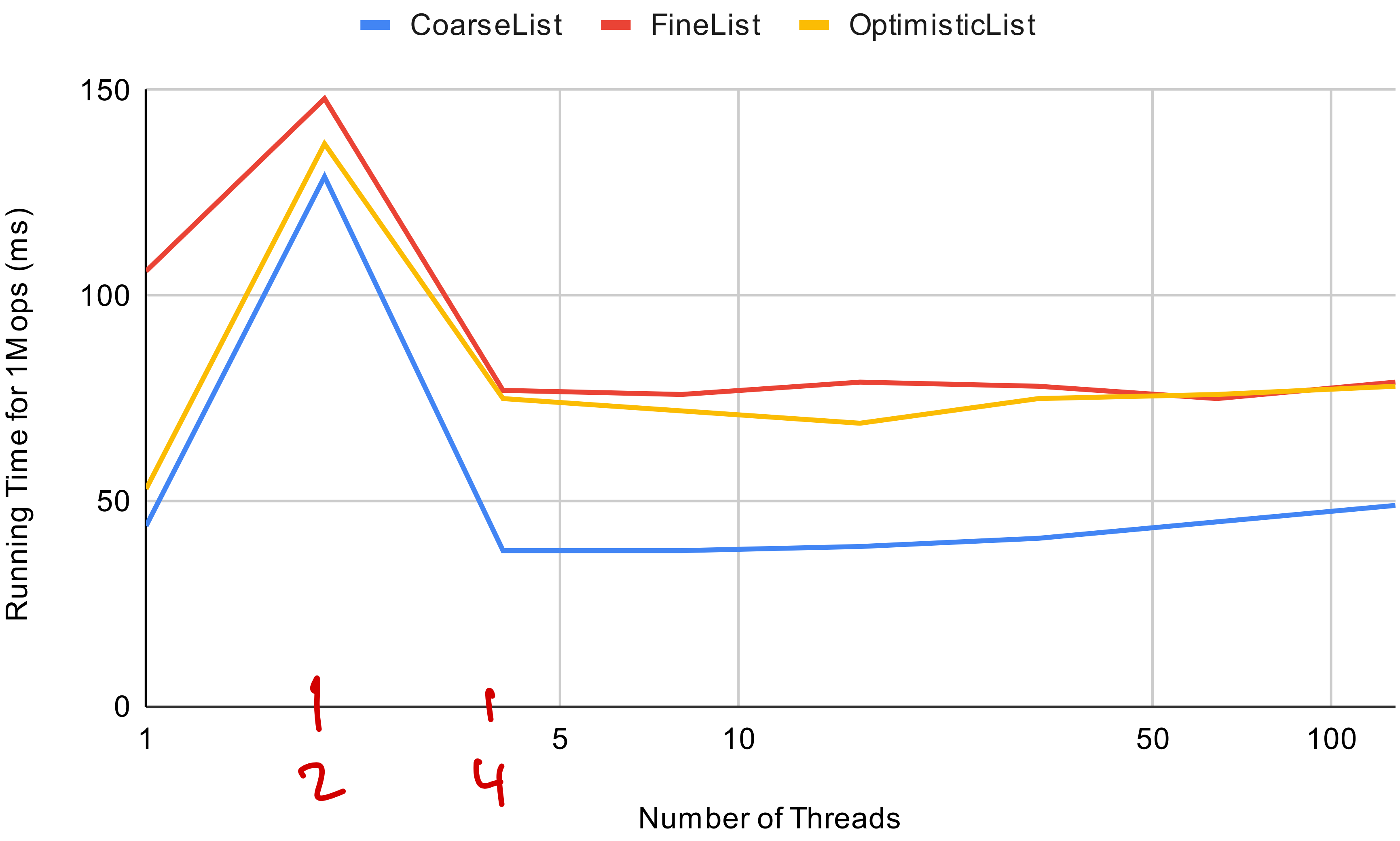


Performance v. Size, 128 Threads

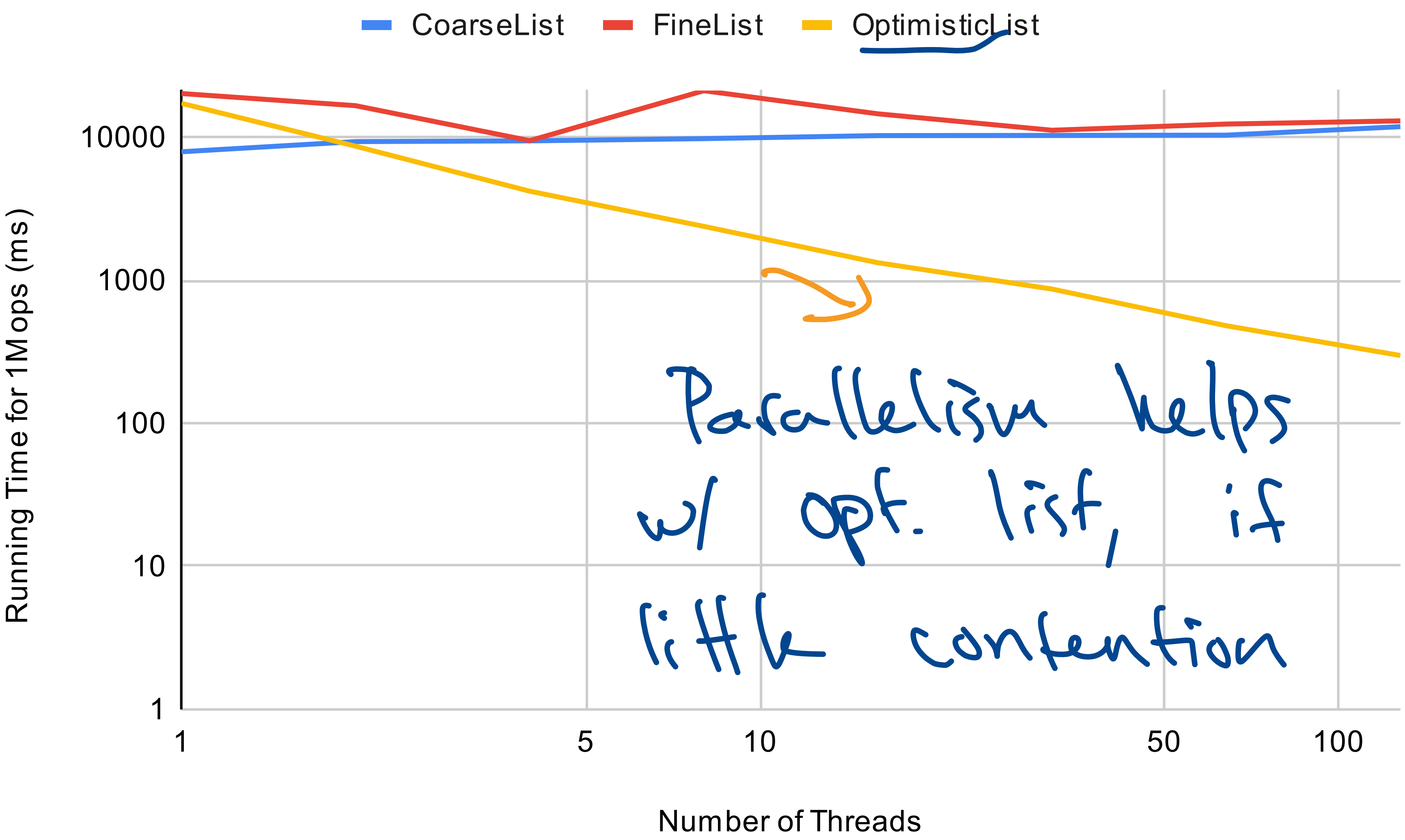


Validation failure?

Time v. Threads, 8 Elements

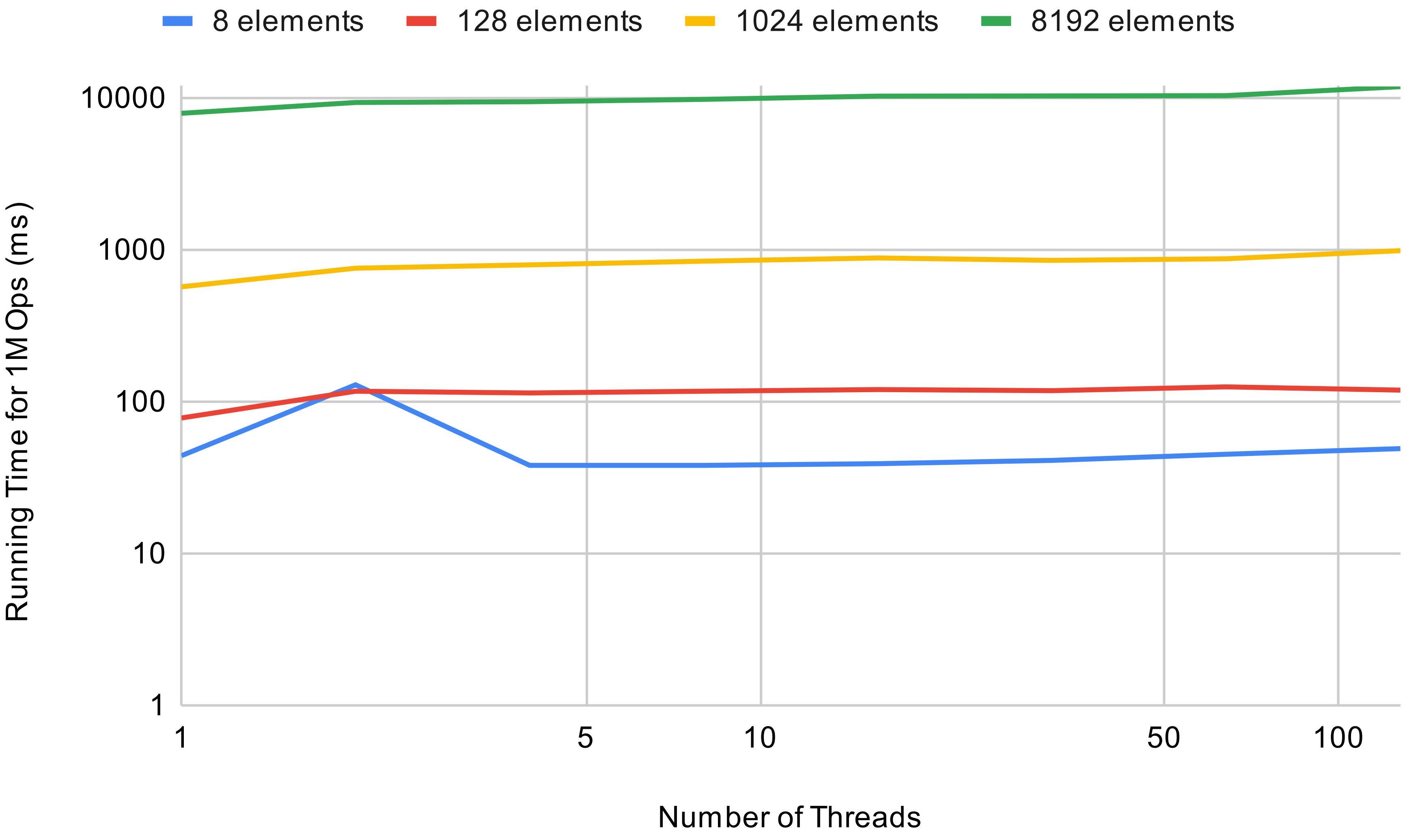


Time v. Threads, 8,192 Elements

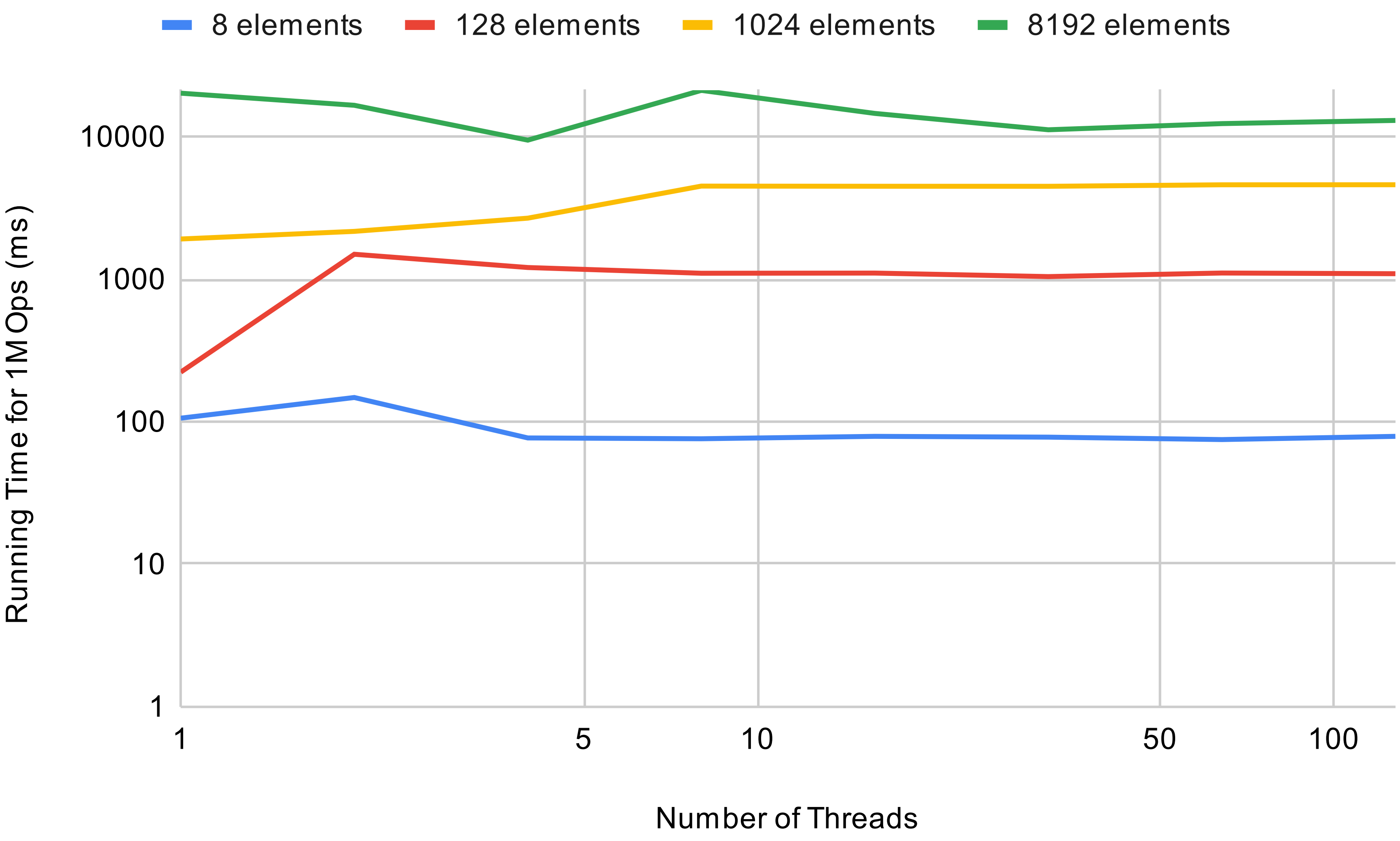


Parallelism helps w/ opt. list, if little contention

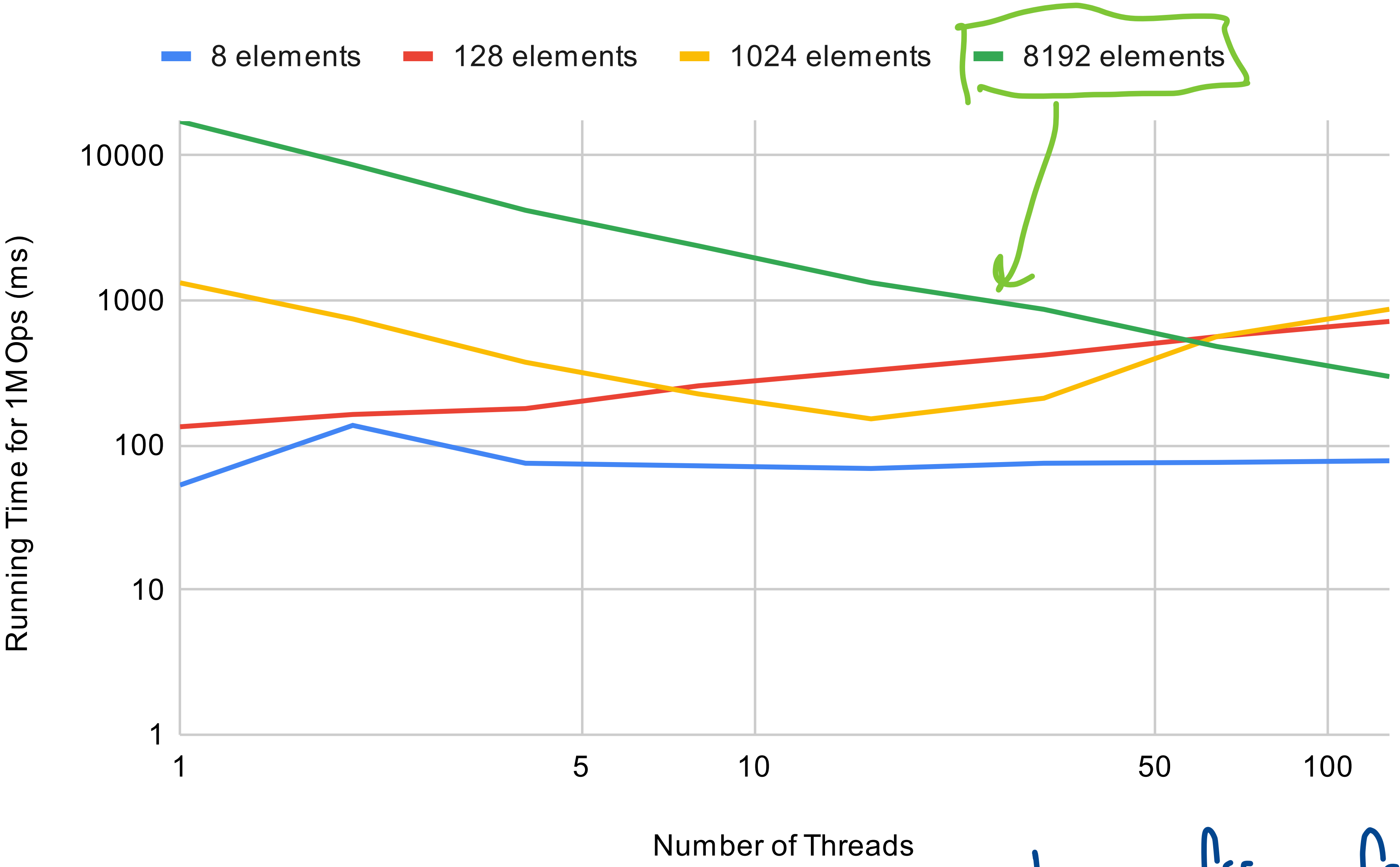
Coarse Time v. Threads



Fine Time v. Threads



Optimistic Time v. Threads



Larger set = more benefit from
more cores/threads

Further Improvement?

Question. What is undesireable about optimistic locking?

→ Validation requires second list traversal

→ Validation failure costs even more

Optimism and Validation

Under best circumstances:

- validation succeeds
    - likely if little contention
- still traverse the list twice

Under contention:

- all operations are *blocking*
    - not wait-free
- contention can lead to validation failures
    - not starvation-free

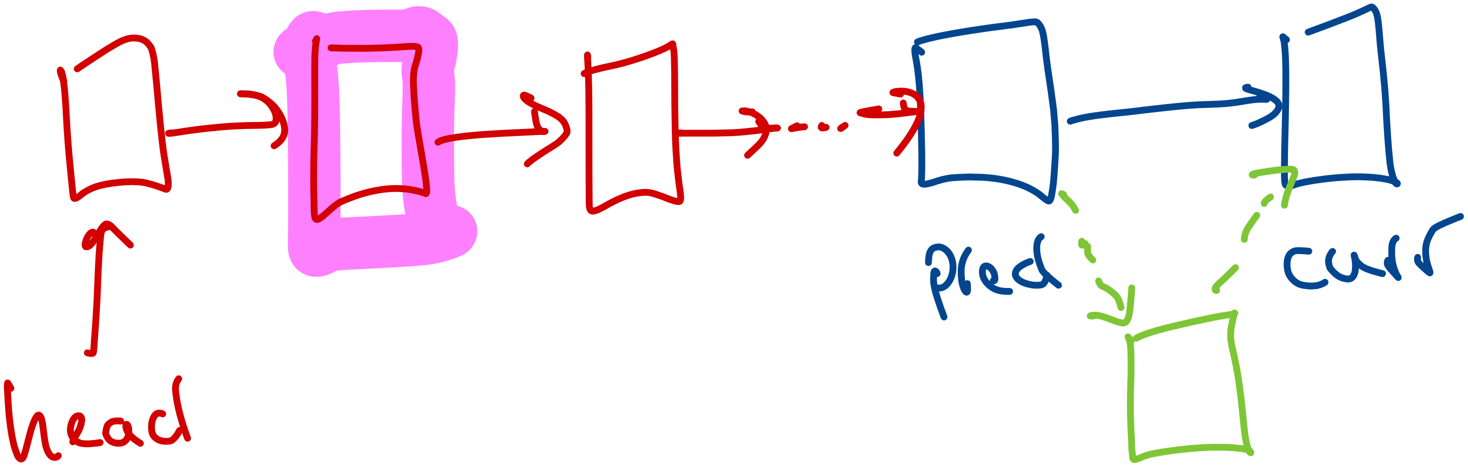
Observation

Operations are complicated because they consist of several steps

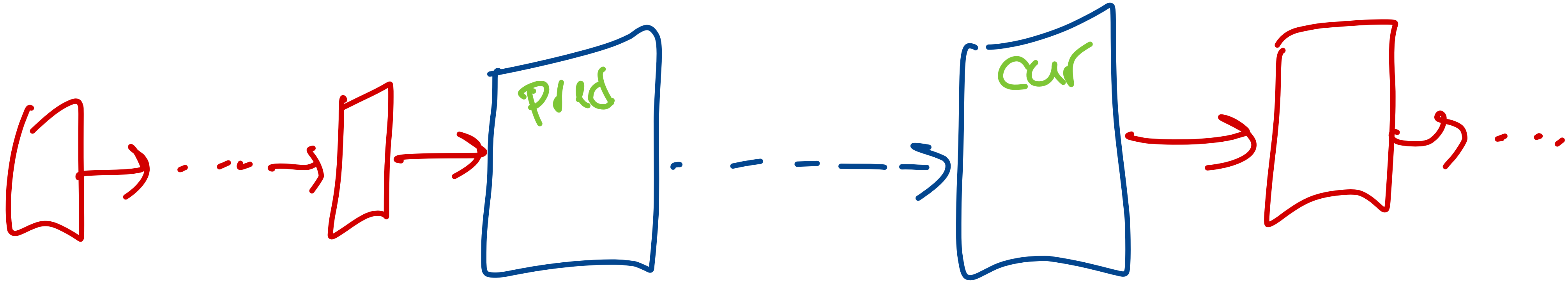
- hard to reason about *when* the operation appears to take place
- coarse/fine-grained synchronization stop other threads from seeing operations “in progress”
- optimistic synchronization may encounter “in progress” operations before locking
    - validation required

Overly Optimistic?

Question. What operation(s) interfere with `add`/`remove` and how? When do we *need* to validate starting at the head?



Conflicting add Operations

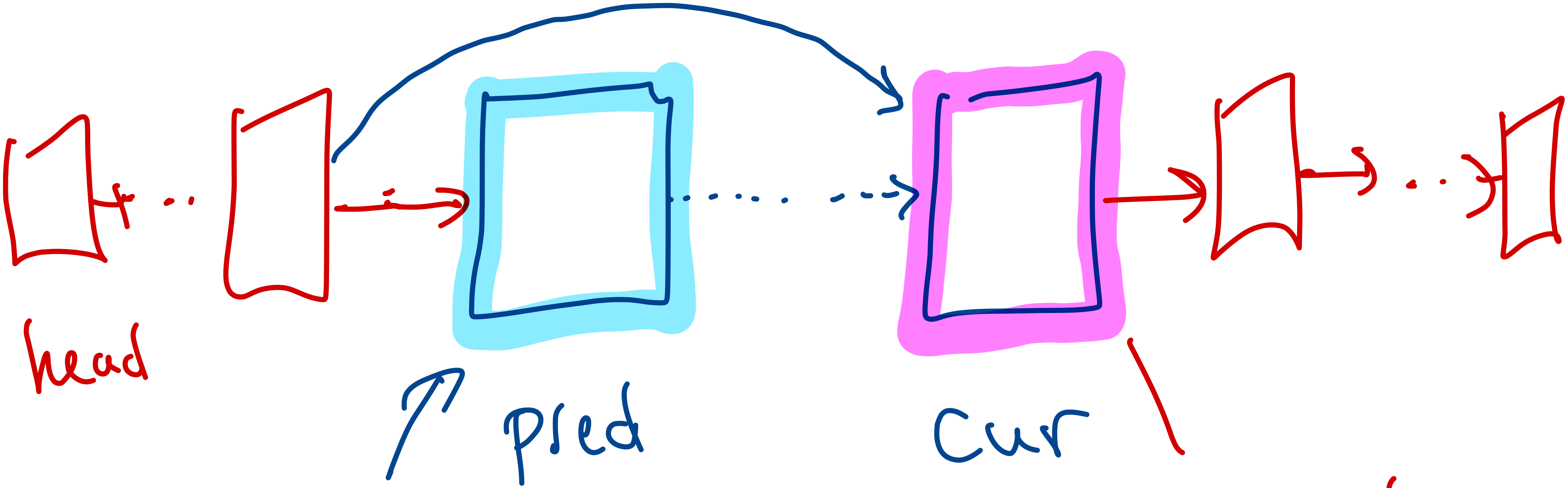


Possible contention only w/ add between pred/cur

→ locally checkable: examine pred.next

Suggestion: add boolean "is removd" value to node

Conflicting remove Operations



removed

now: need to validate from head

removed

=> pred.next was updated (local check)

Alt. Check pred's next?
when removing set pred.next = null?

Improved Validation?

Question. How could we modify `remove` method to make validation more efficient?

- Add boolean val to indicate logical removal to each node
- start removal by flipping bit.

Lazy Synchronization

A simple strategy

- **Mark** a node before physical removal
    - marked nodes are *logically removed*, still physically present
- Only marked nodes are ever removed

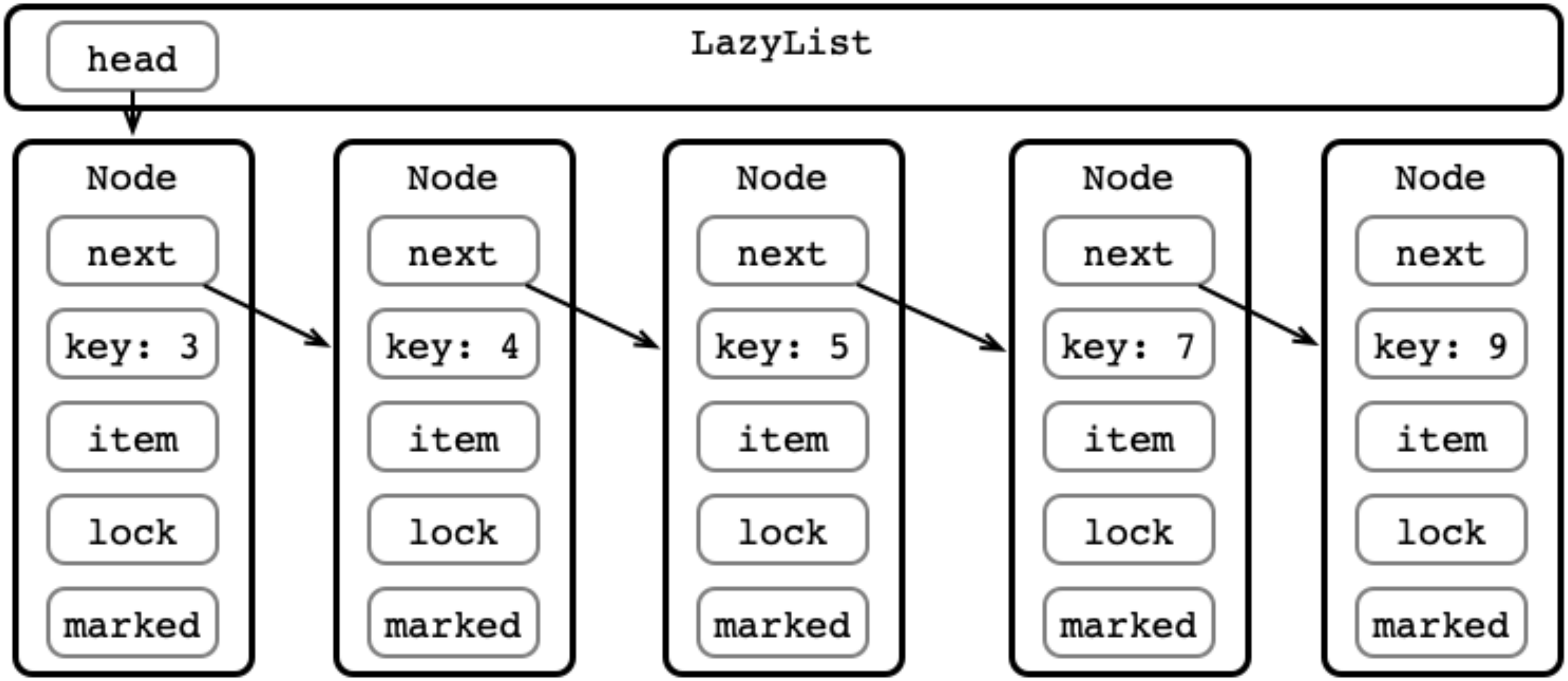
Validation simplified:

- Just check if nodes are marked
- No need to traverse whole list!

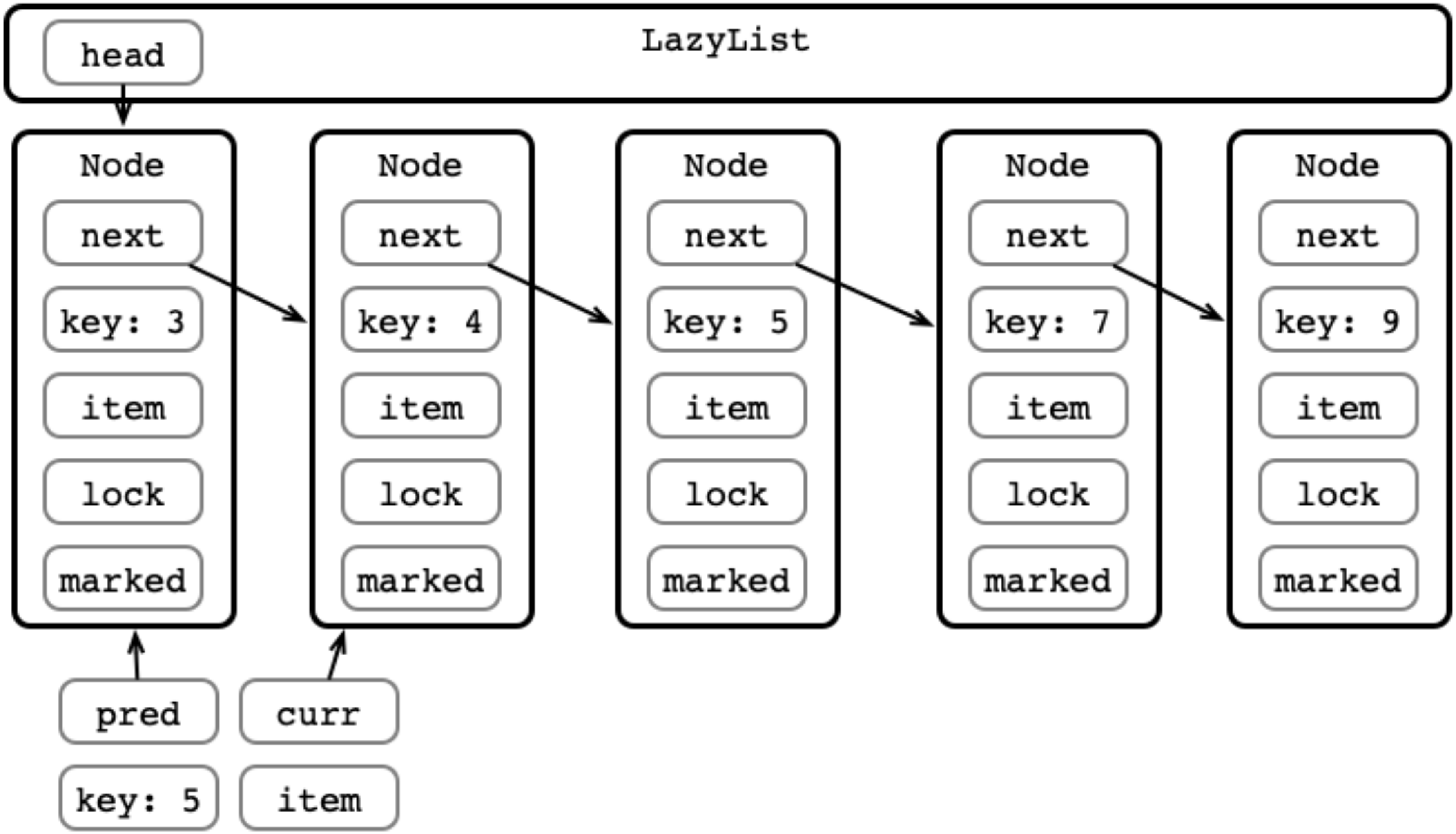
Lazy Operation

1. Traverse without locking
2. Lock relevant nodes
3. Validate list
    - check nodes are
        - not marked
        - correct relationship
    - if validation fails, go back to Step 1
4. Perform operation
    - for removal, mark node first
5. Unlock nodes

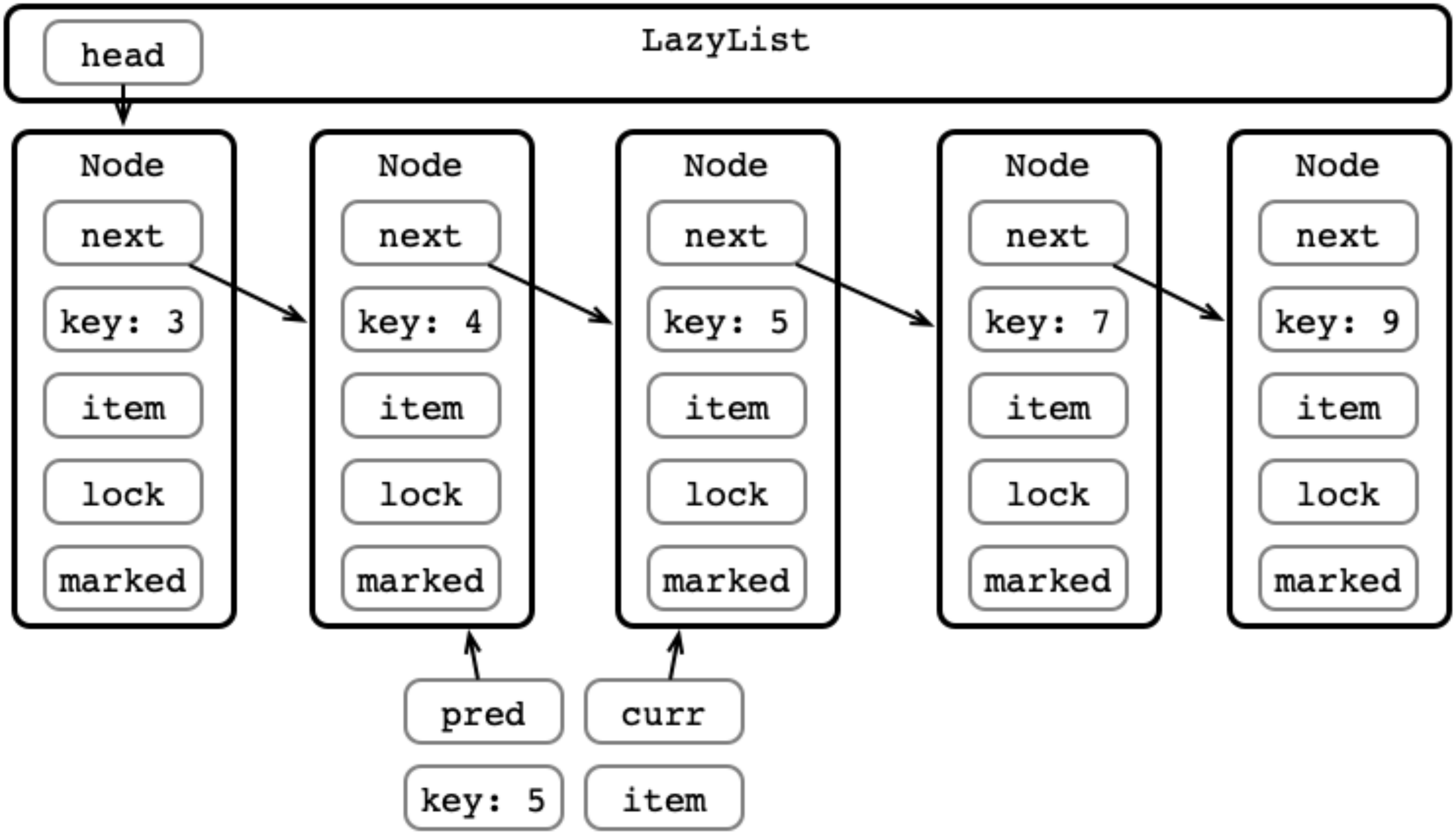
Lazy Removal Illustrated



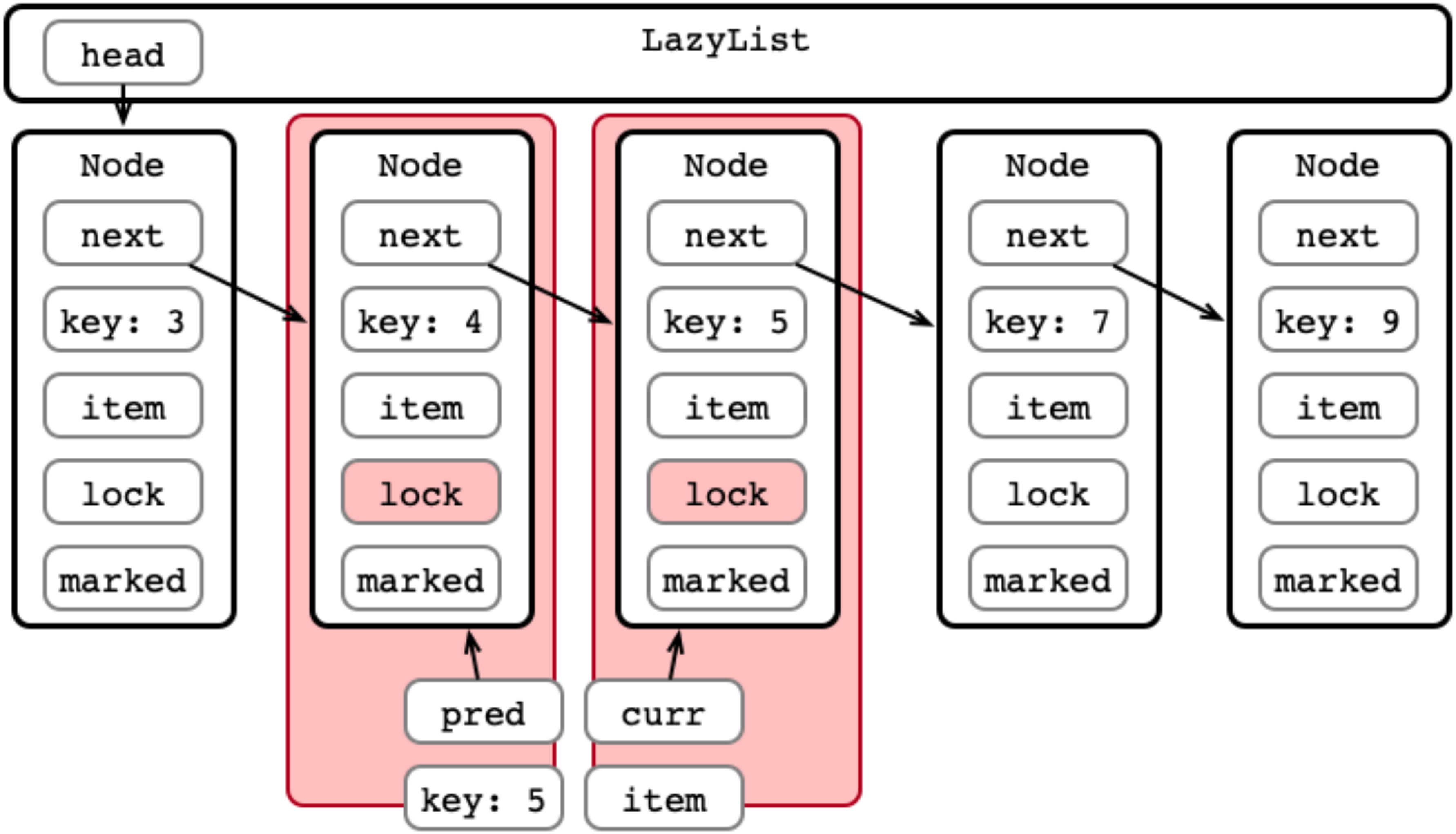
Step 1: Traverse List



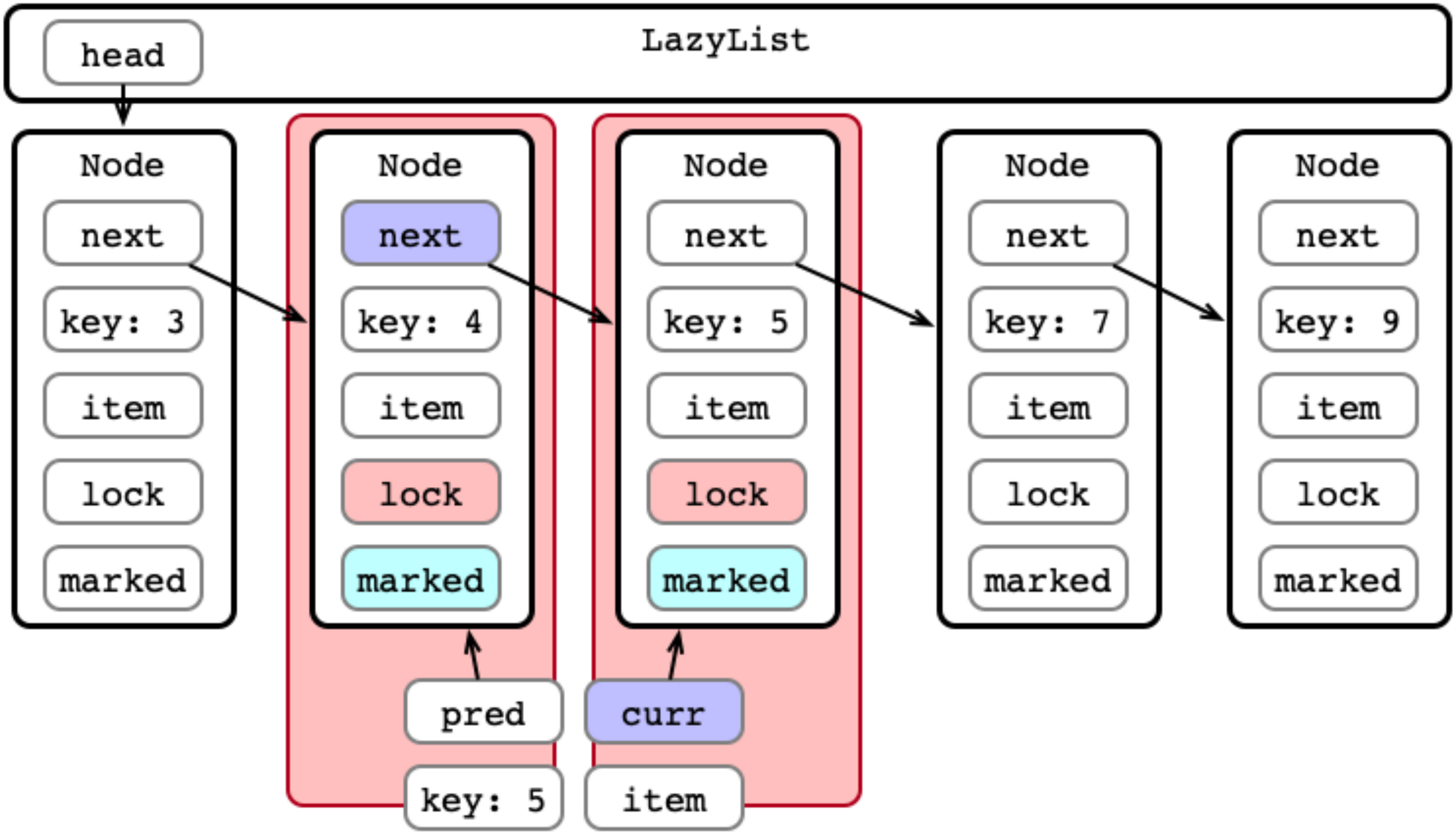
Step 1: Traverse List



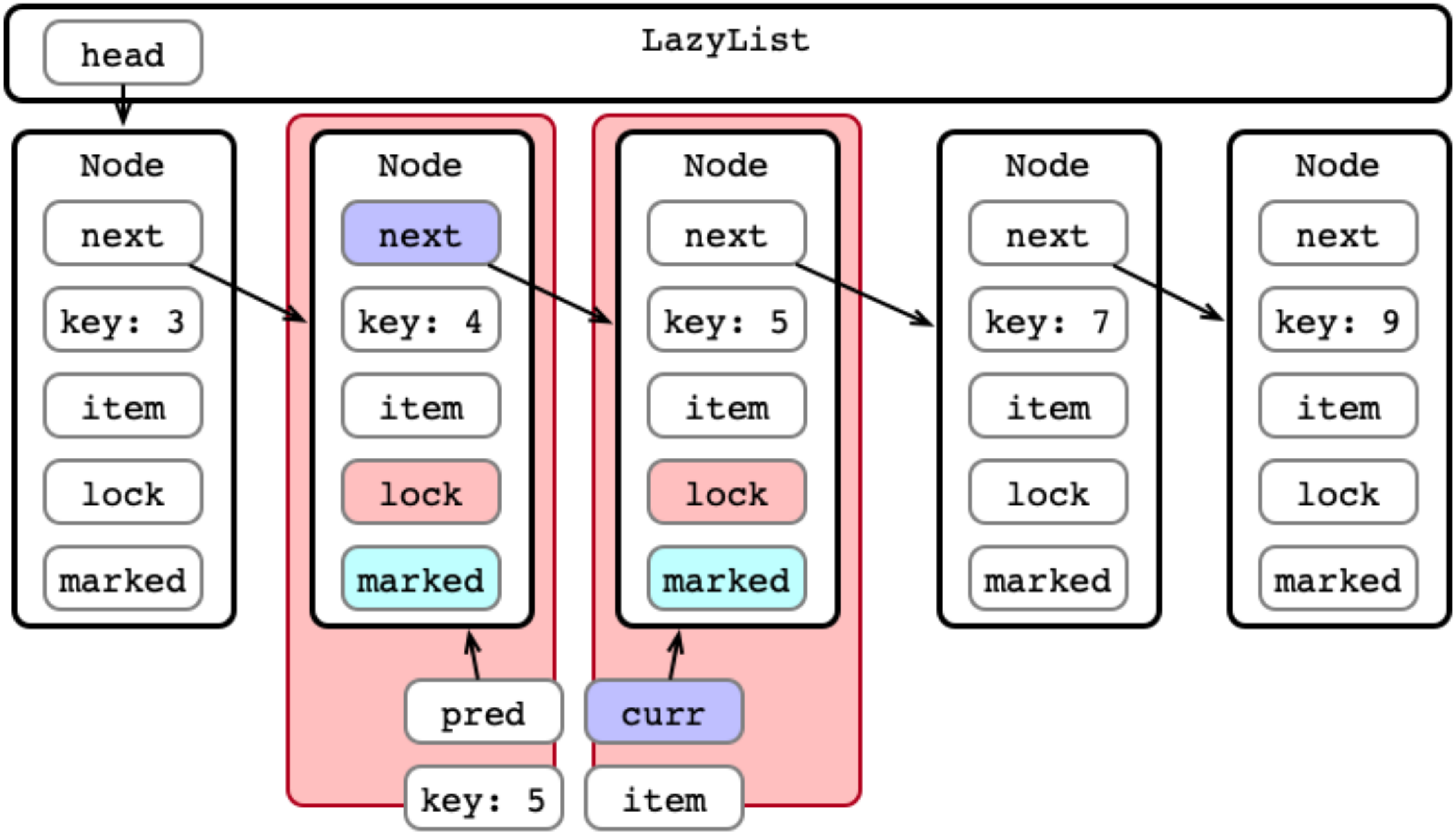
Step 2: Lock Nodes



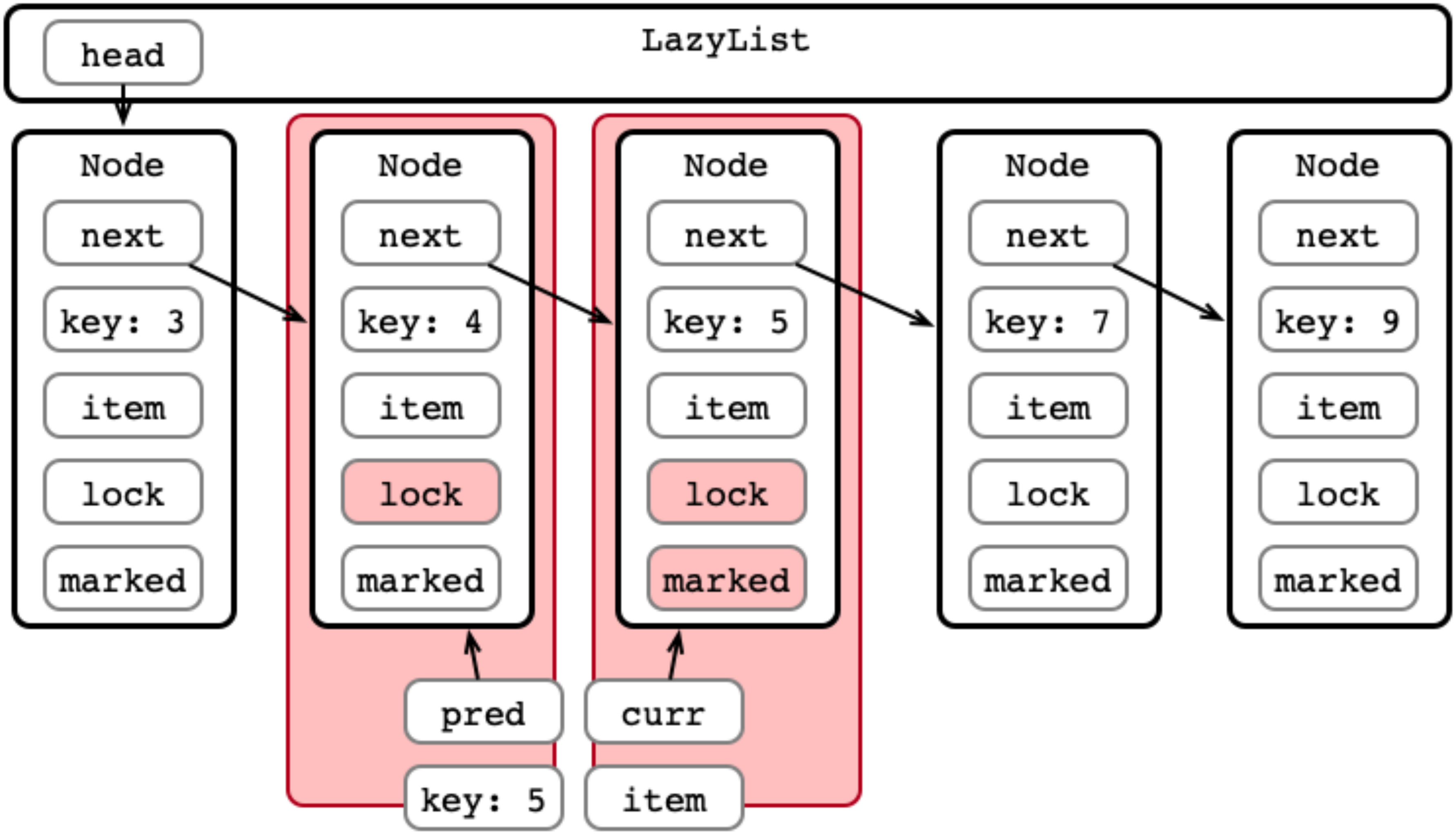
Step 3: Validate `pred.next == curr`?



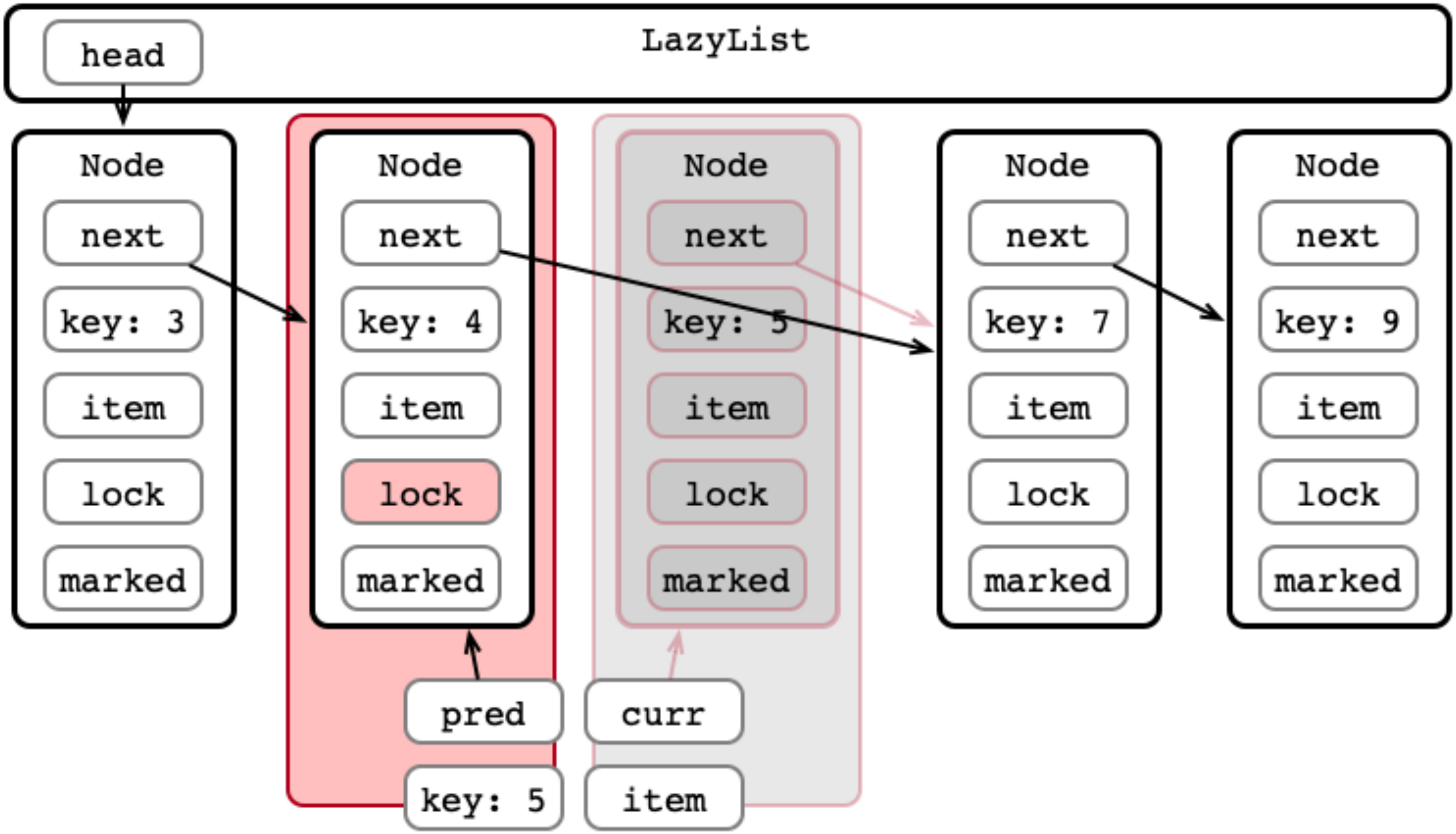
Step 3: Validate not marked?



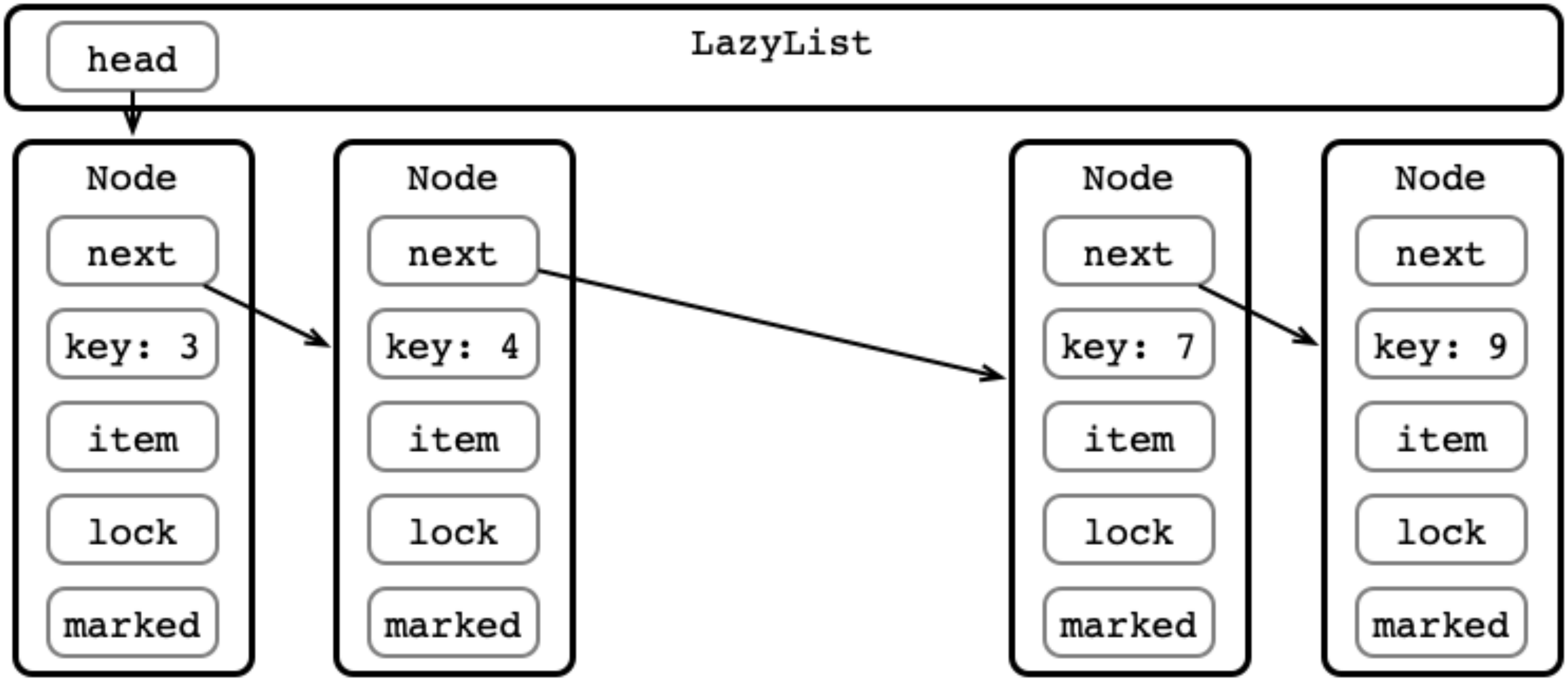
Step 4a: Perform Logical Removal



Step 4b: Perform Physical Removal



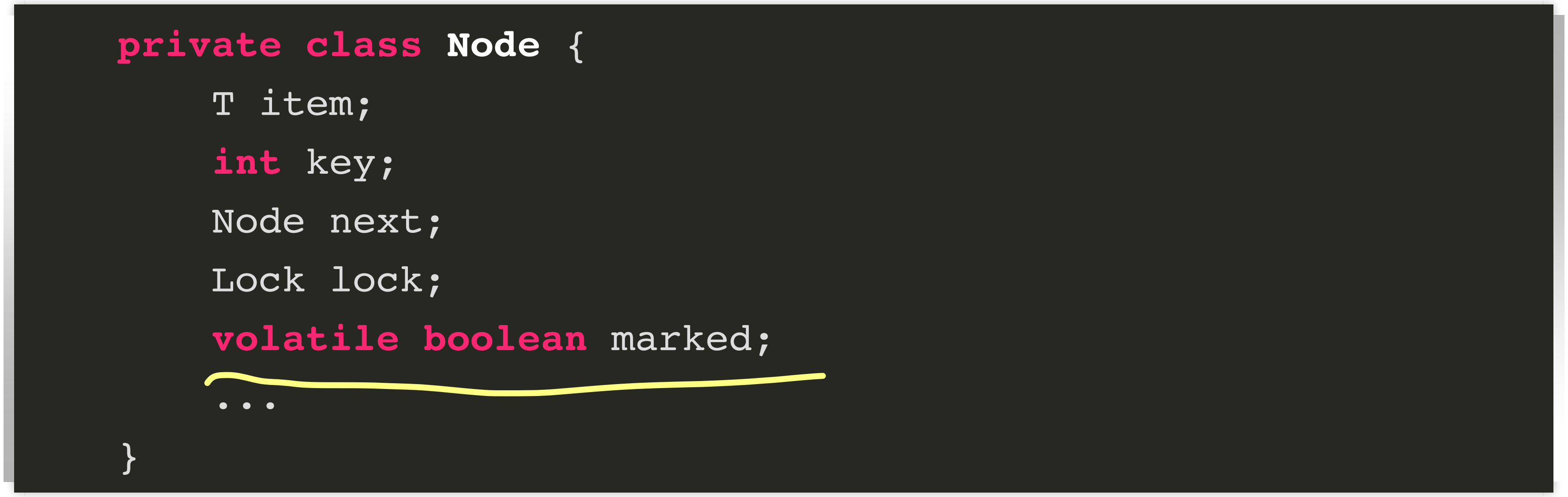
Step 5: Release Locks and Done!



In Code

- `LazyList.java` in `linked-lists.zip`

A Node in Code



```
private class Node {
    T item;
    int key;
    Node next;
    Lock lock;
    volatile boolean marked;
    ...
}
```

Validation, Simplified

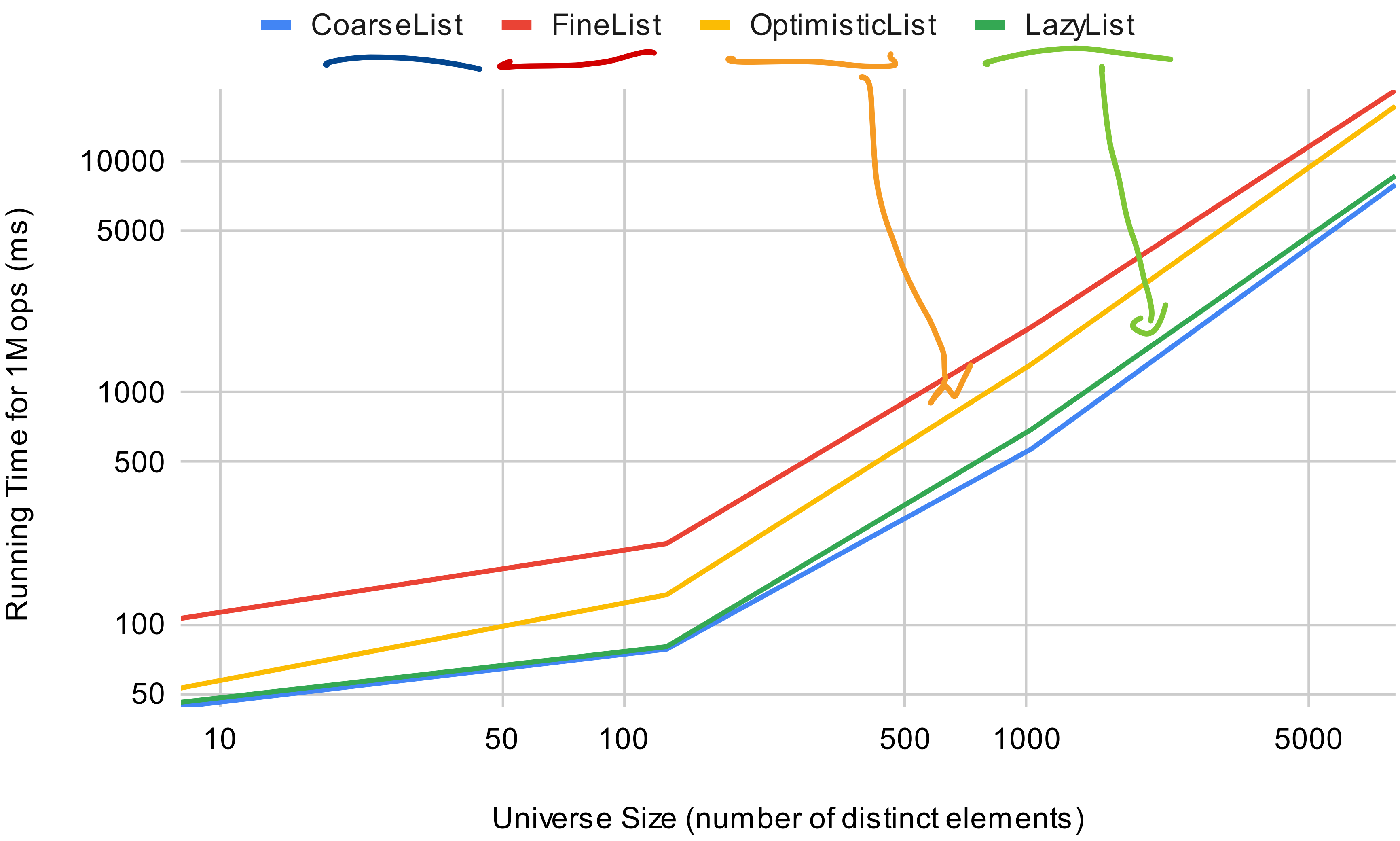
```
private boolean validate (Node pred, Node curr) {
  return !pred.marked && !curr.marked && pred.next == curr;
}
```

Improvements?

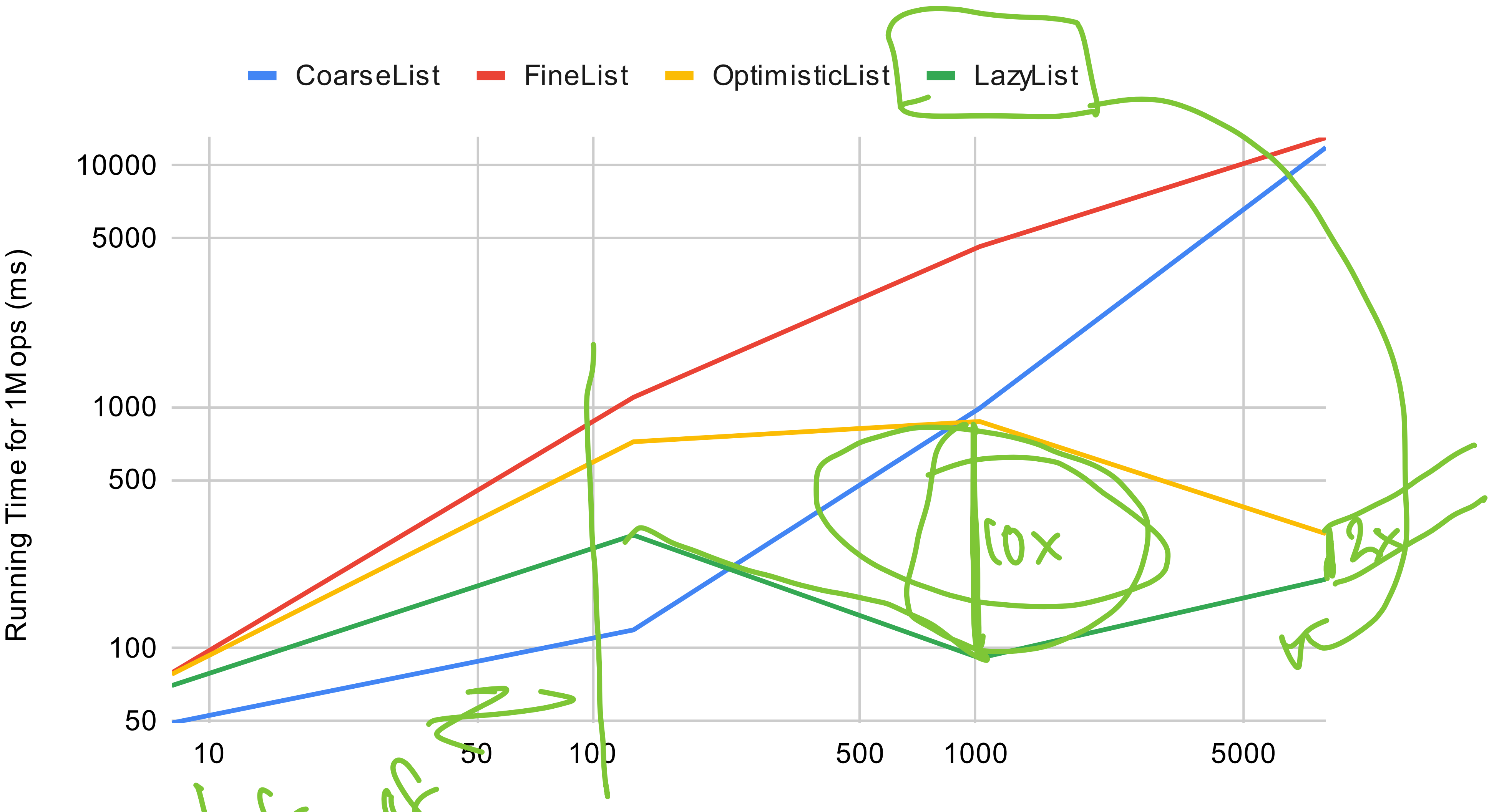
1. Limited locking as in optimistic synchronization
2. Simpler validation
   - faster—no second list traversal
   - more likely to succeed?
3. Logical removal easier to reason about
   - linearization point at logical removal line
4. `contains()` no longer acquires locks
   - often most frequent operation
   - now it is wait-free!

What About Performance?

Performance v. Size, 1 Thread

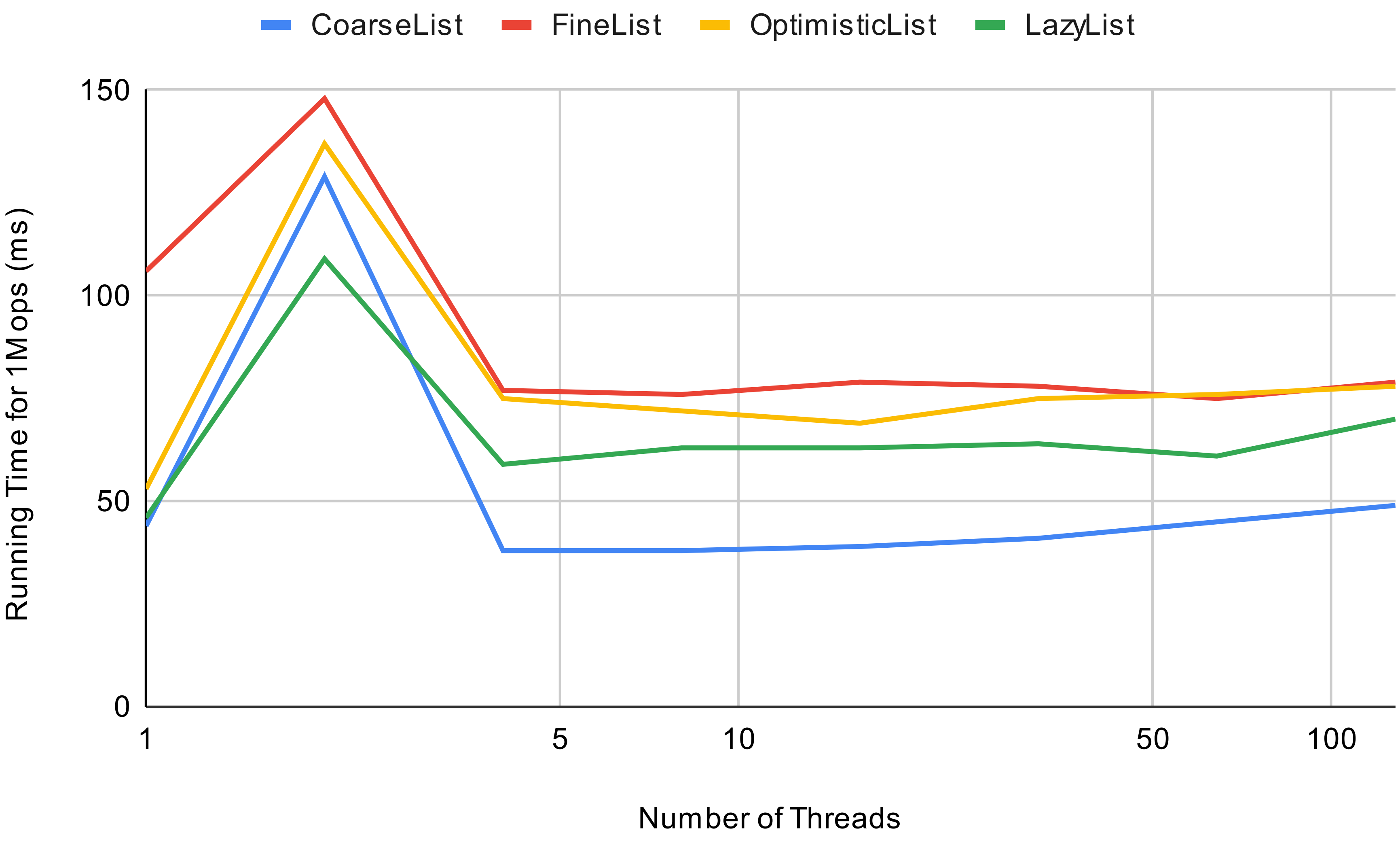


Performance v. Size, 128 Threads

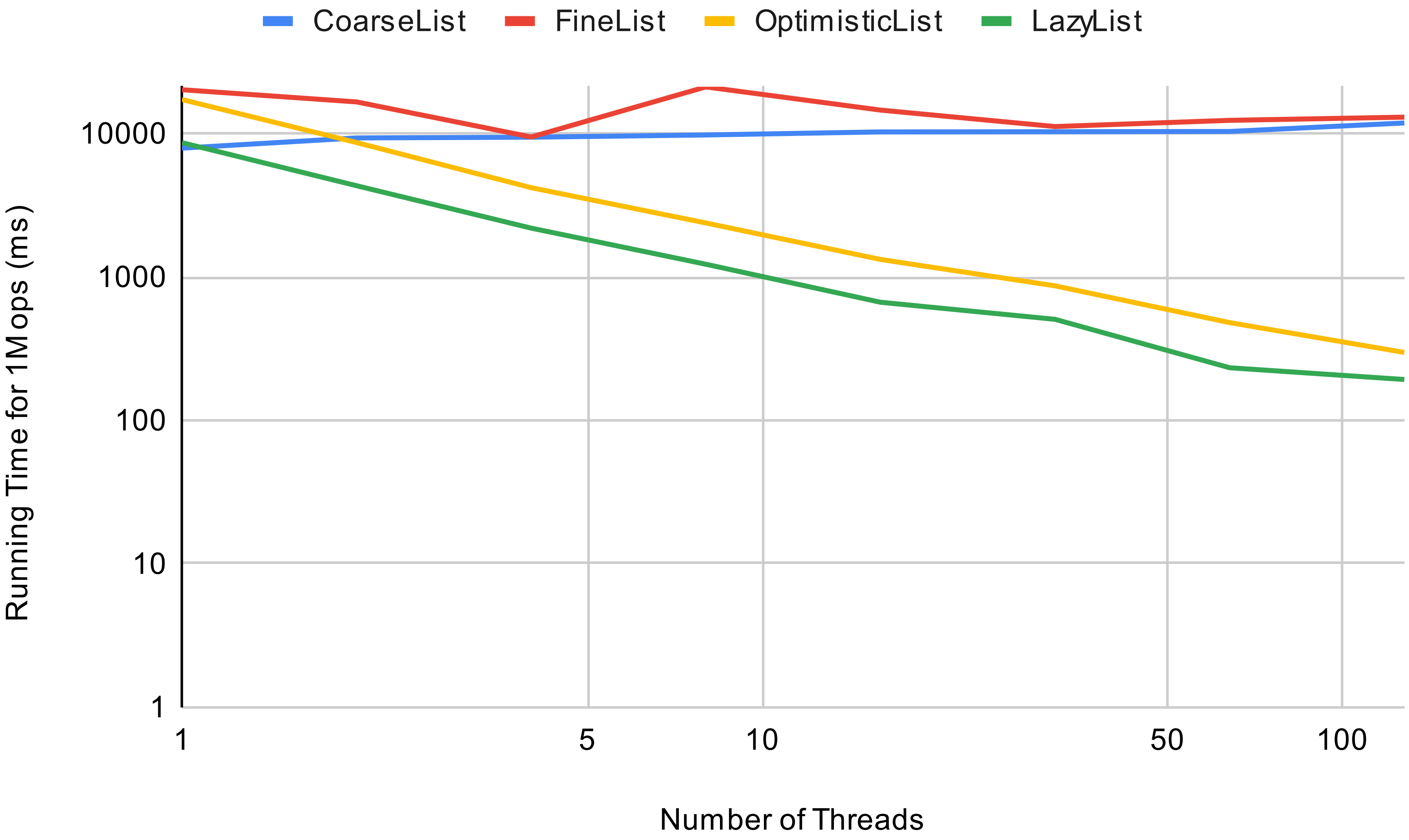


lots of contention (validation errors)

Time v. Threads, 8 Elements



Time v. Threads, 8,192 Elements



Further Improvements?

What could be done better?

1. concurrent `add`/`remove` operations can still block one another
2. operations are still *not* starvation free

Further Improvements?

What could be done better?

1. concurrent `add`/`remove` operations can still block one another
2. operations are still *not* starvation free

Question. Can we avoid locks entirely?