Advanced Multiprocessor Programming: Data-Structures Part 1

Martin Wimmer, Jesper Larsson Träff

TU Wien

April 24th, 2023

A first data structure: List-based set

General issues and concerns:

- Coarse-grained vs. fine-grained synchronization
- Safety and liveness properties
- Progress guarantees
- Data structure: Algorithms and complexity

Coarse-grained synchronization

The trivial solution: Use lock to protect all data structure operations

- Acquire lock when data structure is accessed (especially updated): Each method call acquires and releases
- Likely major scalability bottleneck!
  - Only one thread can access data-structure at any time
    → Even for read accesses!
  - No progress guarantees
    → Thread holding a lock may block other threads indefinitely
  - But linearizable (when lock is acquired)

Can we do better?

Fine-grained synchronization

- Decouple parts of the data-structure
  $\rightarrow$ multiple locks
- Get rid of locks altogether
  $\rightarrow$ Use higher-consensus operations instead (atomics)
- Provide progress guarantees

Liveness properties

- Deadlock-free
- Starvation-free
- Non-blocking
  - No thread can block other threads

    (Threads not actively participating at the moment shall not hinder other threads from progressing)
  - All threads may require an unbounded number of steps
- Lock-free
  - At least one thread progresses in a bounded number of steps
- Wait-free
  - All threads progress in a bounded number of steps

Correctness condition: Linearizability

- Operation appears to take effect as a single, atomic step somewhere during the execution
- Operations on data-structure appear as atomic operations to the outside
  - (other, concurrent method calls)

- Tricky to achieve

  → The combination of two linearizable operations is not necessarily linearizable (an insert combined with a delete will not be atomic to the outside observer)

Proving linearizability

For each implemented operation:

- Look at the outcome (return value)
- Show that there is some instant (a linearization point) during the execution of the operation where this outcome is correct in the linearized history (sequence of linearization points)
- The linerarization point is sometimes a specific operation, but sometimes depends on the history (concurrent execution of other operations)

The data structure: List-based set

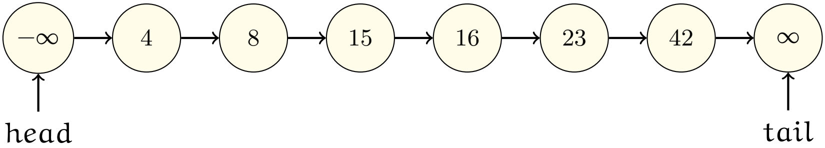
Set data-structure operations:

- `bool add(item)`
  - Add item to the set (if not already in set)
  - Return `true` if item was added, `false` if already existing
- `bool remove(item)`
  - Remove item from set
  - Return `true` on success, `false` if item was not found
- `bool contains(item)`
  - Check whether item exists in set

Implementation as ordered, linked list, items identified by key drawn from an ordered universe. Note: In code snippets, we do not distinguish between `item` and `key`

In terms of sequential complexity, not a good data structure, all operations are $O(n)$ and some require $\Omega(n)$

- Sorted, linked list of node elements
- Elements identified by unique key from ordered universe
- Items stored in nodes are comparable by key
- Sentinel nodes for head and tail
- Head and tail contain maximum and minimum key values ($-\infty$ and ∞)



List-based set: Assumptions and Invariants

- Assumptions
  - Freedom from interference

    (only operations `add`, `remove` and `contains` may modify nodes)
  - Garbage collection available

- Invariants: Properties that hold initially and are maintained by any (concurrent) operation; no thread can take a step that makes an invariant false.
  - Sentinel nodes are neither added nor removed
  - Nodes are sorted by item (key)
  - Items are unique
  - Item is in the set iff it is reachable from the head

List-based set with coarse-grained locking

- Single, global lock
- Acquired at beginning of each method call
- Released at the end
- Similar to a sequential implementation

```
Node* head;
mutex m;

bool add(T item) {
  // C++11 style locking
  lock_guard<mutex> g(m);

  // Search for item or successor
  Node* pred = head;
  Node* curr = pred->next;
  while (curr->item < item) {
    pred = curr;
    curr = curr->next;
  }

  // Item alread in set
  if (item == curr->item) return false;

  // Add item to set
  Node* n = new Node(item, curr);
  pred->next = n;

  return true; // Done
} // end of lock scope
```

List-based set with coarse-grained locking: Properties

- Obviously correct
- Execution is essentially sequential

- Linearization point
  - The instant the lock is acquired

- Liveness
  - Same as the lock implementation

- No progress guarantees!

  (Thread holding the lock may stall, progress of all other threads blocked)

Relaxation: Reader-Writer Lock

- Use Reader-Writer lock instead of a normal lock
- Allows concurrent read accesses
- Better performance if `contains` is the most common operation
  → Typically the case

```
Node* head;
rw_mutex m;

bool contains(T item) {
  // Pseudocode
  lock_guard<mutex> g(m.read_lock());

  // Search for item
  Node* curr = head;
  while (curr->item < item) {
    curr = curr->next;
  }

  if (item == curr->item)
    return true; // element in set
  else
    return false; // not in set
} // end of lock scope
```

Reader-Writer Lock: Properties

- Multiple `contains` checks can be performed at the same time
- Linearization point
  - The instant the lock is acquired
- Liveness
  - Same as the lock implementation
  - Still no progress guarantees!

    (A single writer may stall all other threads)
  - Special case: No writers → wait-free

    (if reader-lock implementation is wait-free)

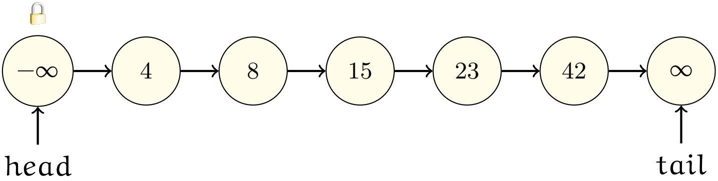
List-based set with fine-grained locking

- Split object into independently synchronized components
- Method calls only interfere when trying to access the same component at the same time

→ Instead of a global lock, use a lock per node

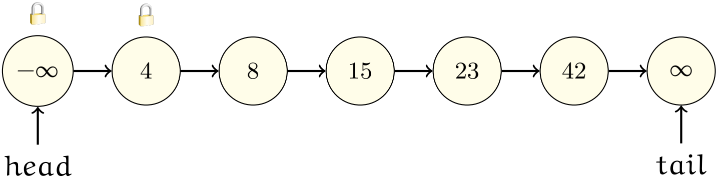
List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks



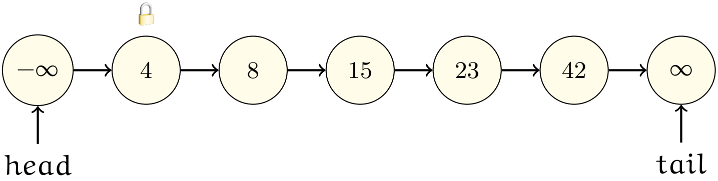
List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks



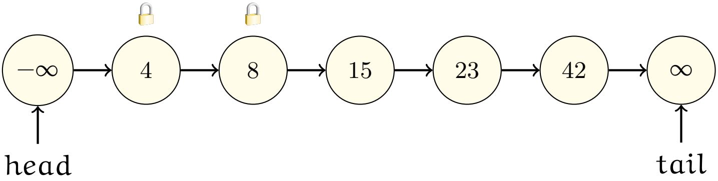
List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks



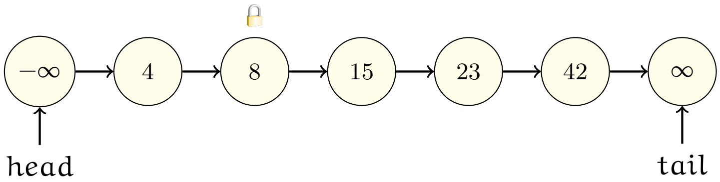
List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks



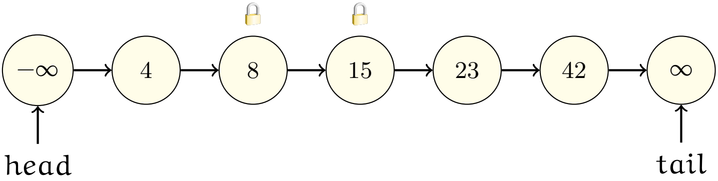
List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks



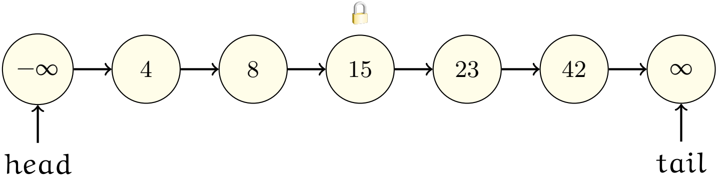
List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks



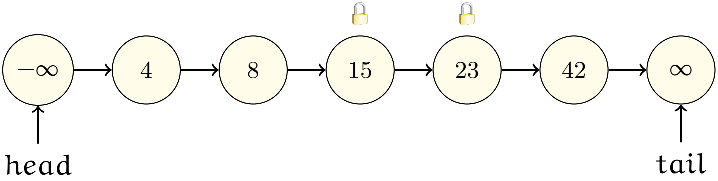
List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks



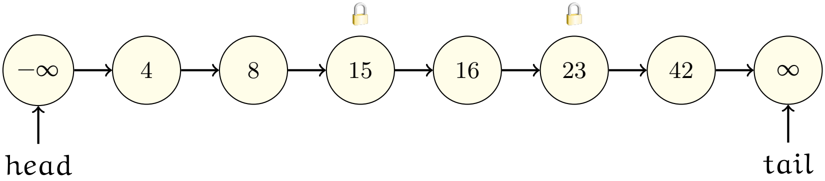
List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks



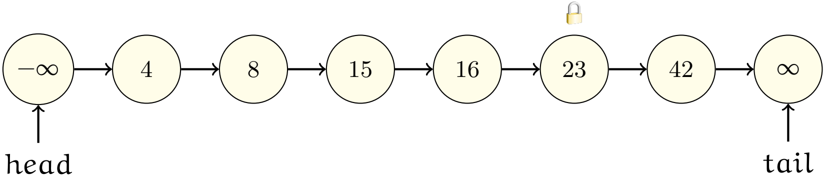
List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks



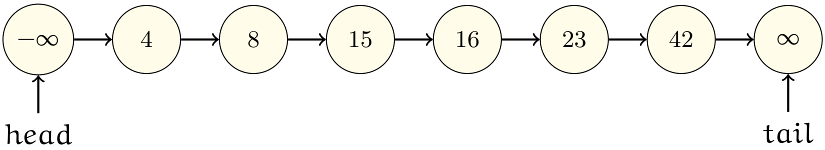
List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks

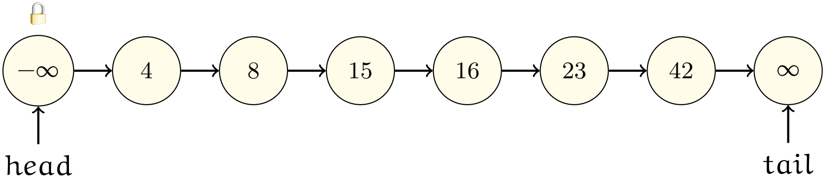


List-based set with fine-grained locks: `add(16)`

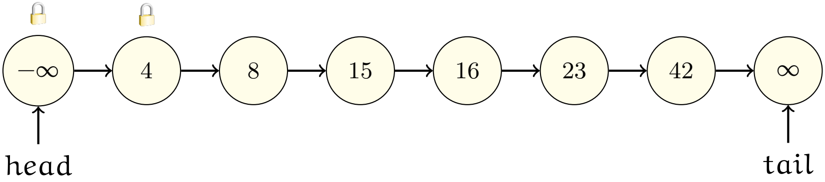
- Locks acquired in hand-over-hand manner
- New item inserted between locks



- Locks acquired in hand-over-hand manner
- Item removed after locking

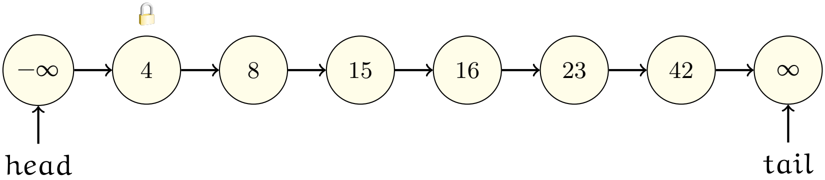


- Locks acquired in hand-over-hand manner
- Item removed after locking



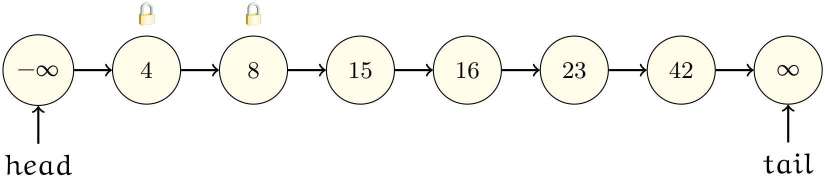
List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking



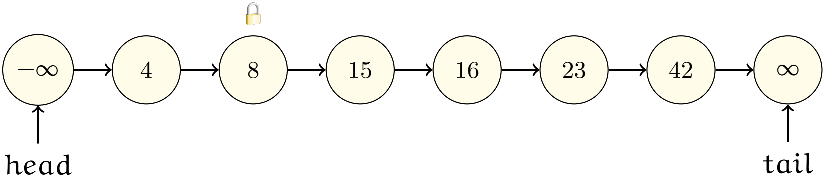
List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking



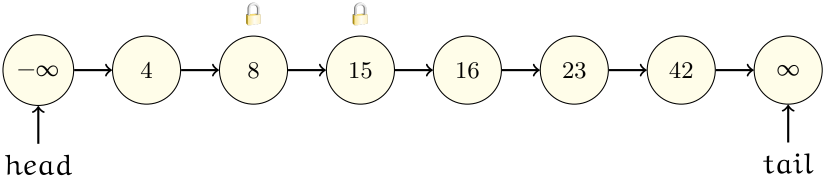
List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking



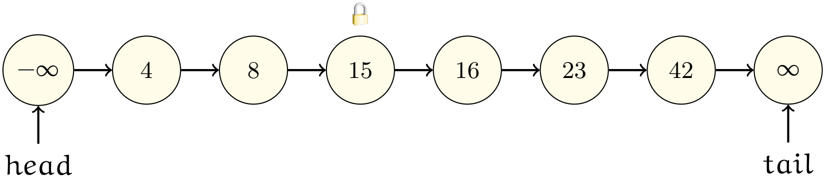
List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking



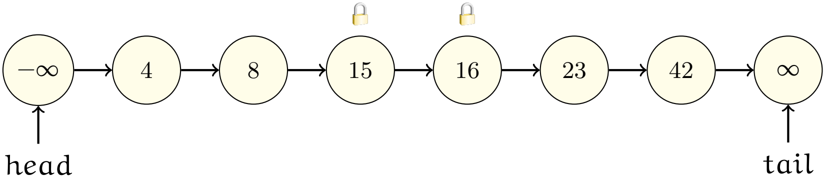
List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking



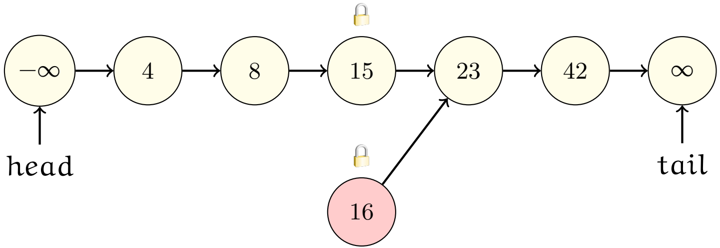
List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking



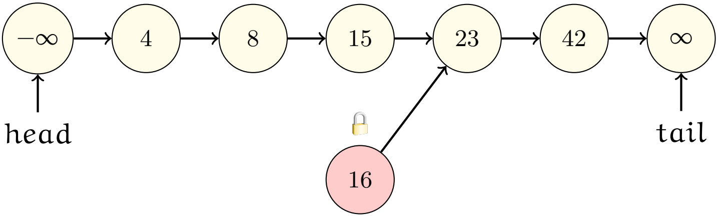
List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking



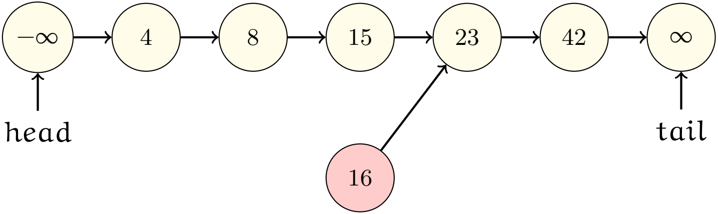
List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking



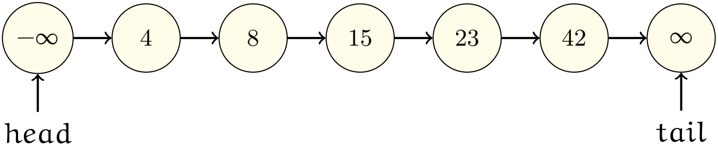
List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking



List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking



List-based set with fine-grained locks

```
Node* head;
struct Window {
  Node* pred;
  Node* curr;
}

Window find(T item) {
  // Search for item or successor
  Node* pred = head;
  pred->lock();

  // Exception safety for locks ignored
       for simplicity
  Node* curr = pred->next;
  curr->lock();

  while (curr->item < item) {
    pred->unlock();
    pred = curr;
    curr = curr->next;
    curr->lock();
  }

  return Window(pred, curr);
}

void unlock(Window w) {
  pred->unlock();
  curr->unlock();
}
```

```
bool contains(T item) {
  Window w = find(item);
  bool found = item == w.curr->item;
  unlock(w);
  return found;
}

bool add(T item) {
  Window w = find(item);
  if (item == w.curr->item) {
    unlock(w);
    return false;
  }
  Node* n = new Node(item, w.curr);
  w.pred->next = n;
  unlock(w);
  return true;
}

bool remove(T item) {
  Window w = find(item);
  if (item != w.curr->item) {
    unlock(w);
    return false;
  }
  w.pred->next = w.curr->next;
  unlock(w);
  delete(w.curr); // safe to delete
  return true;
}
```

List-based set with fine-grained locking: Invariants

- Sentinels are never added or removed
- Nodes are sorted by item
- Each item occurs exactly once
- An item is in the set iff it is reachable from head

From now on: The `find` operations locate position in list such that `pred->item` $<$ `item` $\leqslant$ `curr->item`

List-based set with fine-grained locking: Liveness

- Locks are always acquired in the same order
  $\rightarrow$ deadlock-free
- If the locks are starvation-free, so is the set
- No progress guarantees
- Limited concurrency

  (A thread having lock to a small item early in the list blocks other threads looking for larger items)

List-based set with fine-grained locking: Linearization

Linearization points:

Outcome: Item in set

- Linearize when item containing element is locked
  (Cannot be removed while locked)

Outcome: Item not in set

- Linearize when node containing next-higher item is locked
  (Predecessor node cannot be inserted while an item is locked)

(Earliest possible linearization point: When lock on head is acquired? As soon as head lock has been acquired, thread cannot be overtaken)

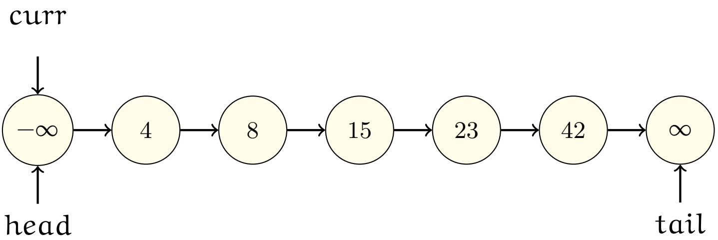
List-based set with optimistic synchronization

Idea:

- Search for a component without acquiring locks
- Lock the found nodes
- Confirm whether the nodes are correct: Validate that they are still reachable, and that `pred` points to `curr`
- On failure, release locks and start over
- Requires good validation!

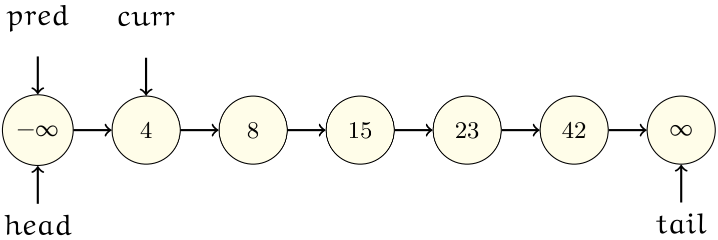
List-based set with optimistic synchronization: `add(16)`

- Search for item without locking



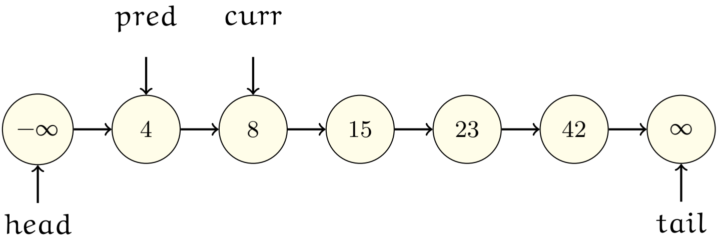
List-based set with optimistic synchronization: `add(16)`

- Search for item without locking



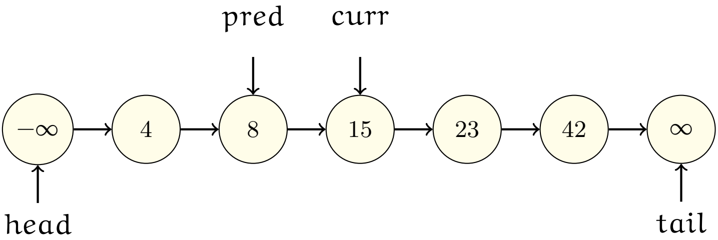
List-based set with optimistic synchronization: `add(16)`

- Search for item without locking



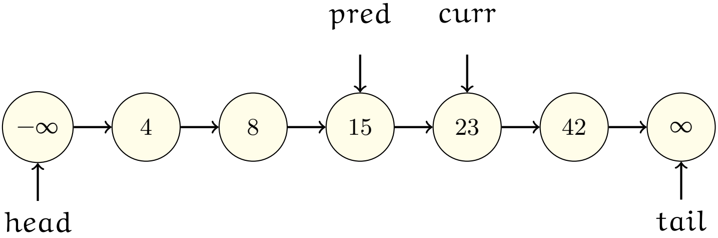
List-based set with optimistic synchronization: `add(16)`

- Search for item without locking



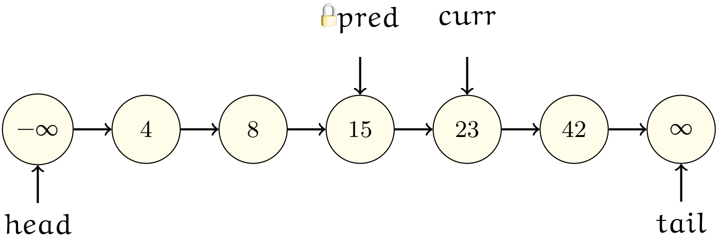
List-based set with optimistic synchronization: `add(16)`

- Search for item without locking



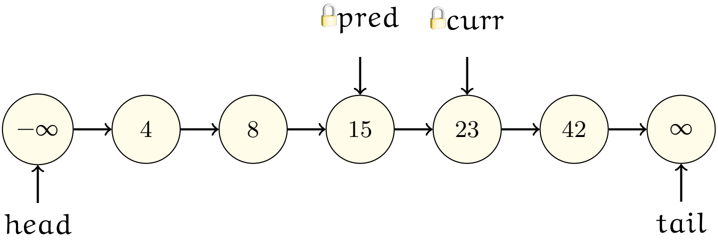
List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item



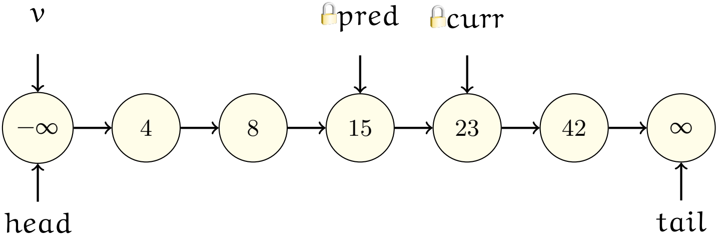
List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item



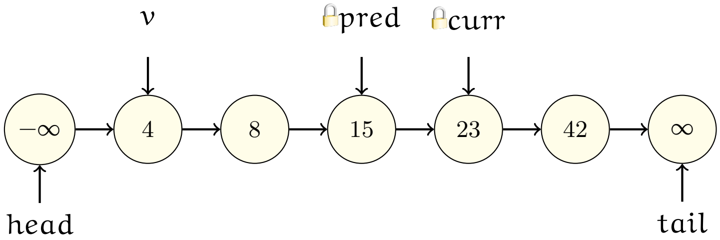
List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item
- Validate by searching for item again



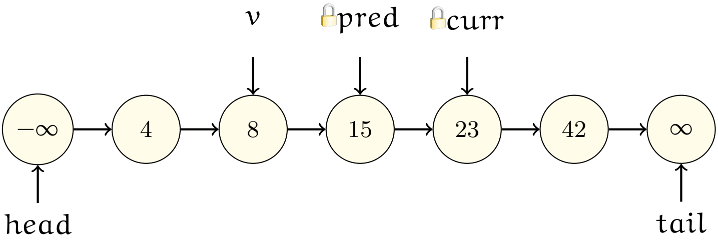
List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item
- Validate by searching for item again



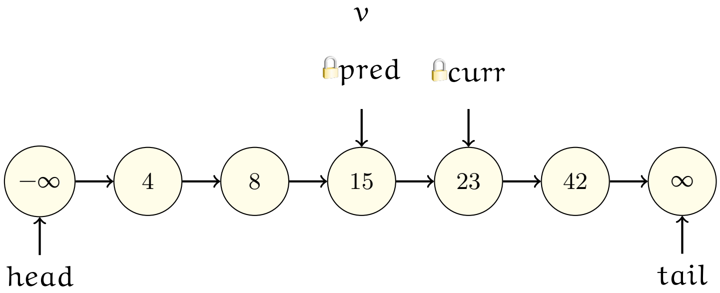
List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item
- Validate by searching for item again



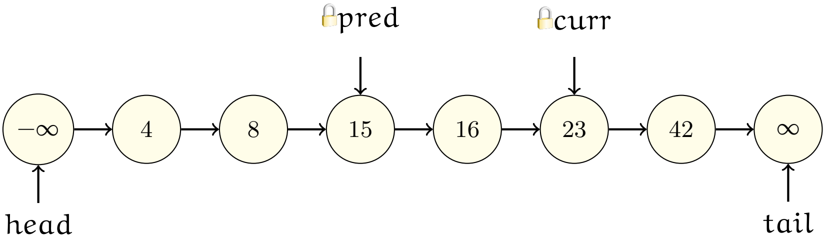
List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item
- Validate by searching for item again



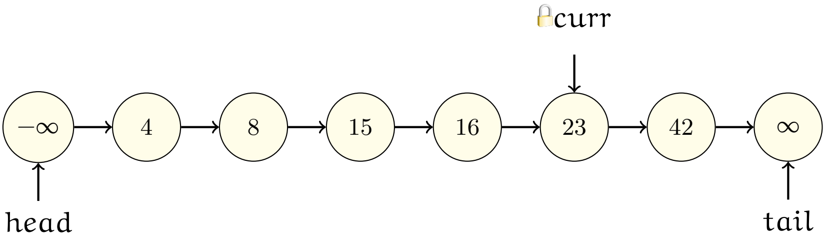
List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item
- Validate by searching for item again
- Add item



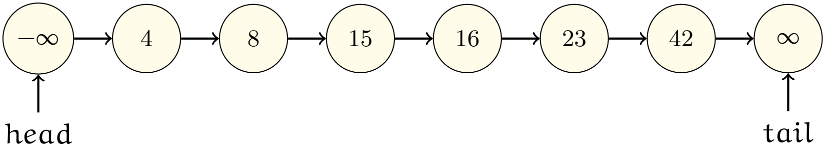
List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item
- Validate by searching for item again
- Add item
- Unlock



List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item
- Validate by searching for item again
- Add item
- Unlock



List-based set with optimistic synchronization

```
Node* head;
struct Window {
  Node* pred;
  Node* curr;
}

void lock(Window w) {
  pred->lock();
  curr->lock();
}

void unlock(Window w) {
  pred->unlock();
  curr->unlock();
}

bool validate(Window w) {
  Node* n = head;
  while (n->item <= w.pred->item) {
    if (n == w.pred)
      return n->next == w.curr;

    n = n->next;
  }
  return false;
}
```

```
Window find(T item) {
  while (true) {
    // Search for item or successor
    Node* pred = head;
    Node* curr = pred->next;

    while (curr->item < item) {
      pred = curr;
      curr = curr->next;
    }

    Window w(pred, curr);
    lock(w);
    if (validate(w)) return w;
    unlock(w);
  }
}
```

List-based set with optimistic synchronization

```
bool contains(T item) {
  // Identical to fine-grained locking:

  Window w = find(item);
  bool found = item == w.curr->item;
  unlock(w);
  return found;
}

bool add(T item) {
  // identical to fine-grained locking

  Window w = find(item);
  if (item == w.curr->item) {
    unlock(w);
    return false;
  }
  Node* n = new Node(item, w.curr);
  w.pred->next = n;
  unlock(w);
  return true;
}
```

```
bool remove(T item) {
  Window w = find(item);
  if (item != w.curr->item) {
    unlock(w);
    return false;
  }
  w.pred->next = w.curr->next;
  unlock(w);

  // Deletion of nodes would be unsafe!
  // delete(w.curr);

  return true;
}
```

List-based set with optimistic synchronization: Properties

- Nodes may be traversed even though they have been removed
- Rely on garbage collection for removed nodes
  - Manual memory reclamation is difficult

    (How can we know that no thread currently reads a node?)

- Previous invariants still apply
  - Sentinels are never added or removed
  - Nodes are sorted by the item
  - Each item occurs exactly once
  - An item is in the set iff it is reachable from head

    → But only if validated!
- Freedom of interference essential: `next` field only changed by operations on the list

List-based set with optimistic synchronization: Liveness

- Deadlock-free
  - Locks are always acquired in the same order
  - On failed validation both locks are freed

- Not starvation-free!
  - Validation might fail all the time as other threads add and remove nodes

- No progress guarantees!

  (But many threads can work concurrently on different items in list)

- Each validation traverses list from head!

List-based set with optimistic synchronization

Linearization points:

Outcome: Item in set

- Linearize when item containing element is successfully validated

Outcome: Item not in set

- Linearize when node containing next-higher item is validated

List-based set with lazy synchronization

Disadvantages of optimistic synchronization (aka: Speculation):

- Need to acquire locks for `contains` call

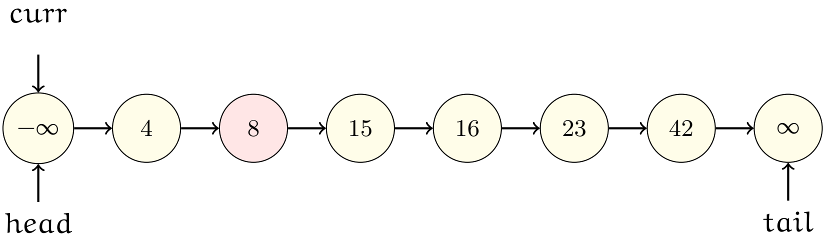
  (`contains` calls are typically more common than `add` or `remove` calls)
- Even though `contains` does not modify the list
- List is traversed twice, so computational overhead is doubled

List-based set with lazy synchronization

Idea: Be lazy and postpone some hard operations, or get help from other threads

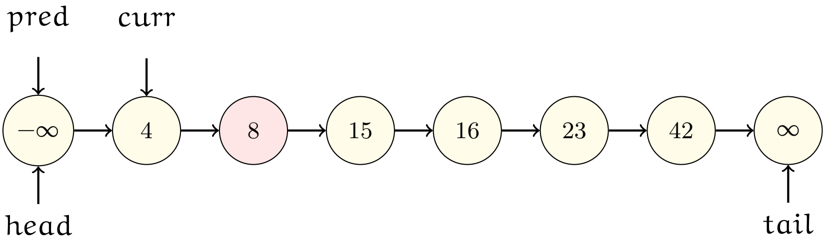
- Split `remove` operation into two parts
  - Logical removal
    - Introduce additional marked flag for each node
    - A node is in the set iff it is reachable and not marked
  - Physical removal
    - Redirect next pointer of predecessor node

- Search for item without locking



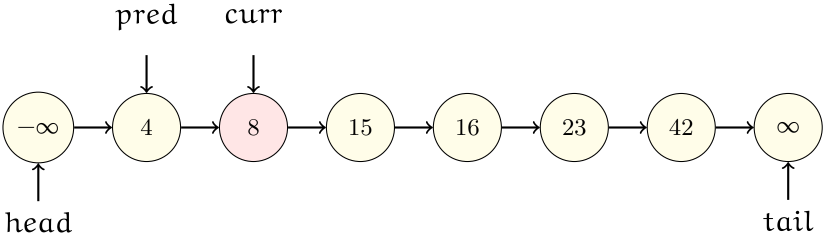
List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking



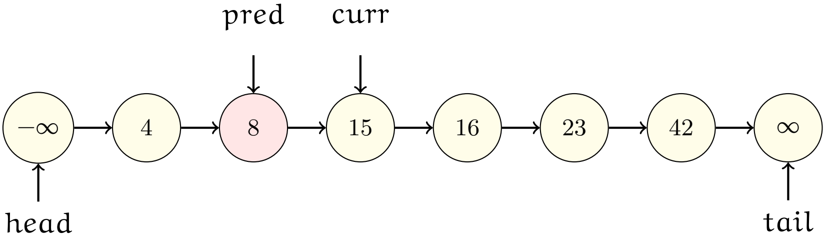
List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking



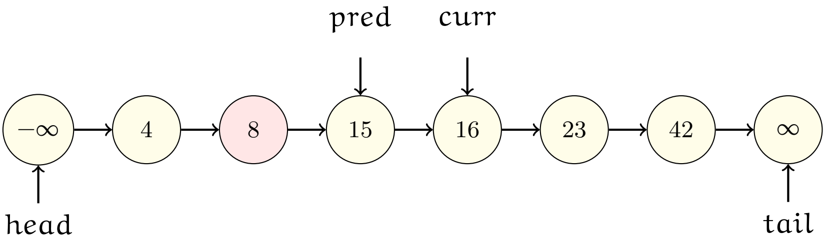
List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking



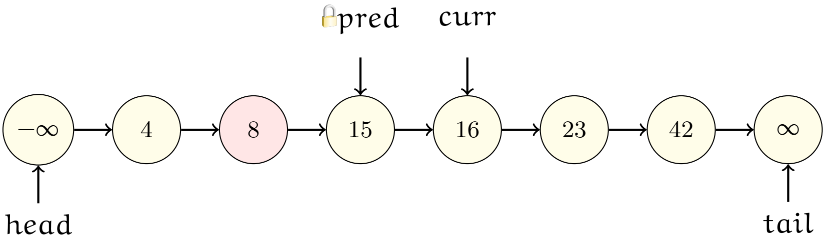
List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking



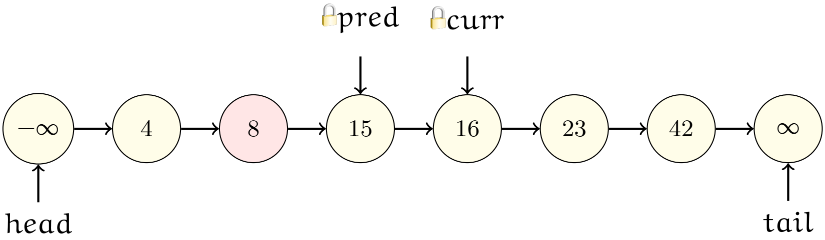
List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking
- Lock item



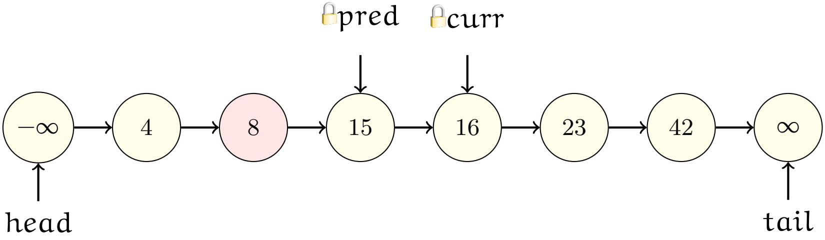
List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking
- Lock item



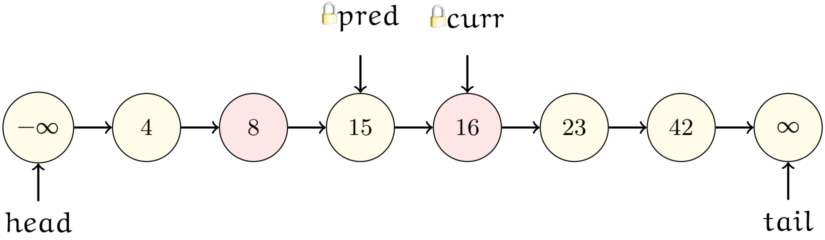
List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking
- Lock item
- Verify by checking marked flags of items

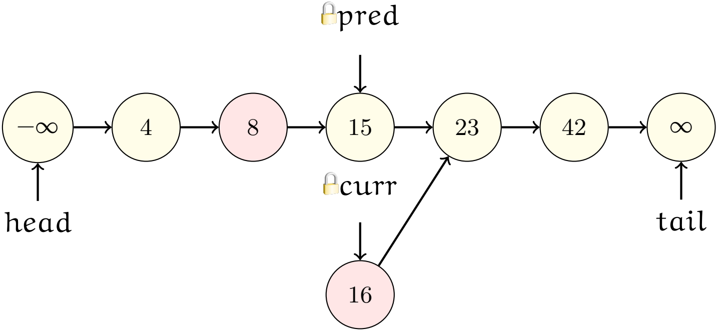


List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking
- Lock item
- Verify by checking marked flags of items
- Mark item as logically removed

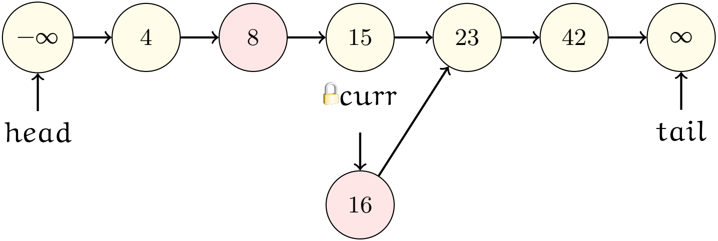


- Search for item without locking
- Lock item
- Verify by checking marked flags of items
- Mark item as logically removed
- Unlink item



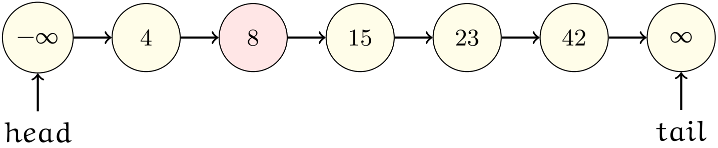
List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking
- Lock item
- Verify by checking marked flags of items
- Mark item as logically removed
- Unlink item
- Unlock



List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking
- Lock item
- Verify by checking marked flags of items
- Mark item as logically removed
- Unlink item
- Unlock



List-based set with lazy synchronization

```
Node* head;
struct Window {
  Node* pred;
  Node* curr;
}

Window find(T item) {
  while (true) {
    // Search for item or successor
    Node* pred = head;
    Node* curr = pred->next;

    while (curr->item < item) {
      pred = curr;
      curr = curr->next;
    }

    Window w(pred, curr);
    lock(w);
    if (validate(w)) return w;
    unlock(w);
  }
}

bool validate(Window w) {
  return !w.pred->marked &&
         !w.curr->marked &&
          w.pred->next == curr;
}
```

```
bool contains(T item) {
  Node* n = head;

  while (n->item < item) n = n->next;

  return n->item == item &&
        !n->marked;
}

bool add(T item) {
  // identical to fine-grained locking
}

bool remove(T item) {
  Window w = find(item);
  if (item != w.curr->item) {
    unlock(w);
    return false;
  }
  // logical remove (mark)
  w.curr->marked = true;

  // physical unlink
  w.pred->next = w.curr->next;
  // Careful with memory order!

  unlock(w);
  return true;
}
```

List-based set with lazy synchronization: Invariants

- Nodes may be traversed even though they have been removed
- We rely on garbage collection for removed nodes
- Sentinels are never added or removed
- Nodes are sorted by the item
- Each item occurs exactly once

- An item is in the set iff
  - it is reachable from head
  - it is not marked

List-based set with lazy synchronization: Liveness

- Deadlock-free
  - Locks are always acquired in the same order
  - On failed validation both locks are freed
  - No locks at all for `contains`
- `add` and `remove` not starvation-free!
  - Validation might fail all the time as other threads add and remove nodes
- Wait-free `contains` call
- Validation does not longer re-traverse list.

List-based set with lazy synchronization: Linearization

Linearization points:

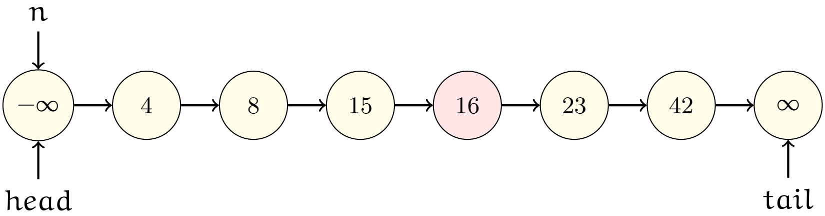
Outcome: Item in set

- `add` Successful validation
- `remove` Flag `marked` set to `true`
- `contains` Read of `marked` flag

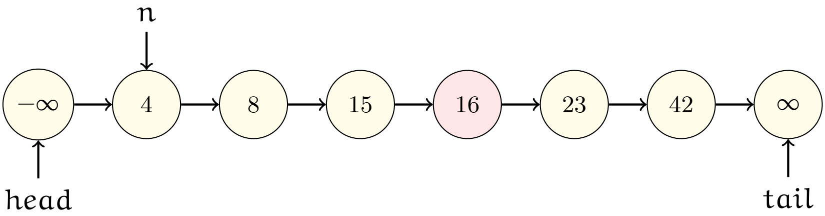
Outcome: Item not in set

- `add` Successful validation of next node
- `remove` Successful validation of next node
- `contains` Earlier of
  - Point where a removed matching node, or a node with `node->item` > `item` is found
  - Point immediatly before a new matching node is added to list

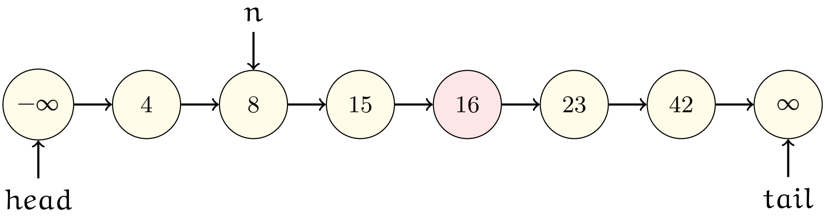
- Search for item without locking



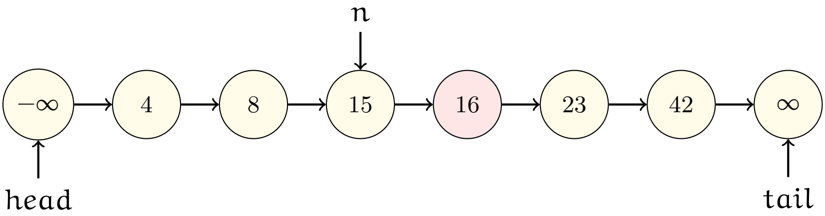
- Search for item without locking



- Search for item without locking



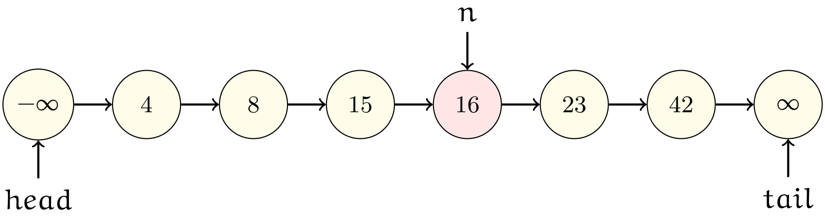
- Search for item without locking



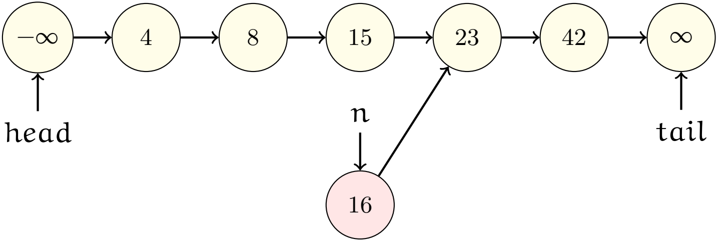
List-based set with lazy synchronization

Linearization of unsuccessful `contains(16)` call

- Search for item without locking



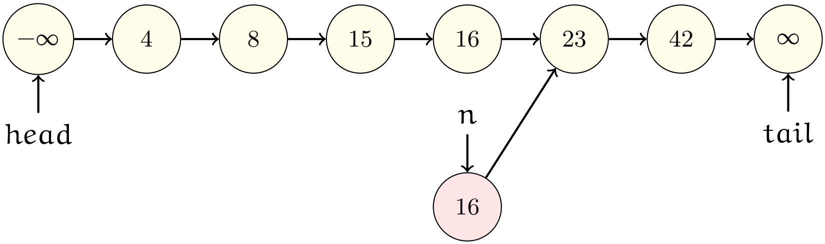
- Search for item without locking
- Node is unlinked by another thread



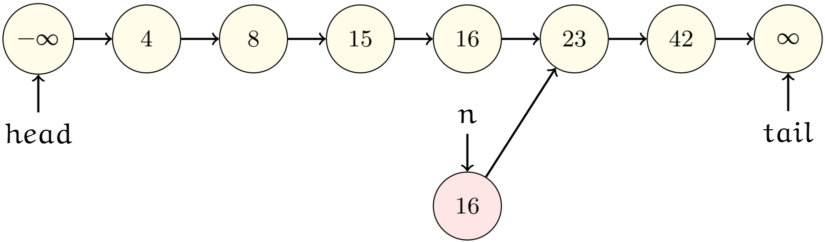
List-based set with lazy synchronization

Linearization of unsuccessful `contains(16)` call

- Search for item without locking
- Node is unlinked by another thread
- Same value node is linked by another thread



- Search for item without locking
- Node is unlinked by another thread
- Same value node is linked by another thread
- `contains` call must be linearized before the check of the marked flag!



List-based set with lazy synchronization: Linearization

Item not in list

add Successful validation of next node

remove Successful validation of next node

contains Earlier of

- Point where a removed matching node, or a node with `node->item` > `item` is found
- Point immediatly before a new matching node is added to list

Linearization at the point where the marked flag is checked would be wrong, since this can be after a new node with same key is added. The point immediately before the new node is added is correct, since this `add` operation must be concurrent with the `contains` call (otherwise, `contains` would have found the item).

Lesson: Linearization point is not always some predetermined instruction!

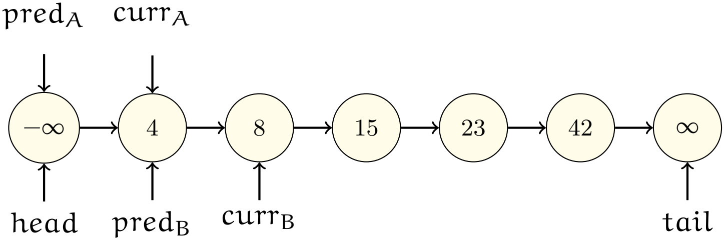
Making the list-based set lock-free

Idea: Use `compare_exchange` to update `next`-pointer when adding and removing items.

Does not work!

- Case 1: Thread A removes first item, thread B adds new second item. Both successfully perform `compare_exchange`, but B's item is not added
- Case 2: Thread A removes first item, thread B removes second item. Both successfully perform `compare_exchange`, but only the first item is removed

Concurrent `remove(4)`, `remove(8)`



Idea: Use `compare_exchange` to update `next`-pointer when adding and removing items.

Repair:

- Treat `next`-pointer and `marked`-flag as an atomic unit.
- `compare_exchange` operation succeeds only if both pointer and flag have the expected value
- Marked nodes cannot be traversed, but must be linked out during traversal by `add` and `remove` operations

In absence of double-word compare-and-swap, steal a bit from the pointer and use that as flag. Addresses are normally less than full 64 bits (48 bits, 32 bits, . . .)

Marked pointer implementation

Take some free (upper) bits from pointer (`next` field in `Node`); use bit operations to read out and modify flag.

```
void *getpointer(void *); // mask out pointer part
bool getflag(void *);     // return stolen bit from next pointer
void setflag(void **);    // set bit in next pointer
void resetflag(void **);  // reset bit in next pointer
```

Note: In code snippets masking out the pointer is mostly left out

Java: Use atomic marked reference (see next lectures)

Lock-free list-based set

```
Node* head;
struct Window {
  Node* pred;
  Node* curr;
}

bool contains(T item) {
  // same as lazy implementation
  // except marked flag is part of next
       pointer
  Node* n = head;

  while (n->item < item)
    n = getpointer(n->next);

  return n->item == item &&
         !getflag(n->next);
}
```

```
Window find(T item) {
  // Search for item or successor
retry:
  Node* pred = head;
  Node* curr = getpointer(pred->next);

  while (true) {
    Node *succ = getpointer(curr->next);
    while (getflag(curr->next)) {
      // link out marked item (curr)

      resetflag(&curr);
      resetflag(&succ);
      if (!compare_exchange(&pred->next,
                                    &curr, succ))
        goto retry;

      curr = succ;
      succ = getpointer(succ->next);
    }

    if (curr->item >= item) {
      Window w(pred, curr);
      return w;
    }
    pred = curr;
    curr = getpointer(curr->next);
  }
}
```

Lock-free list-based set

```
bool add(T item) {
  Window w;

  Node *n = new Node(item);

  while (true) {
    w = find(item);
    Node *pred = w.pred;
    Node *curr = w.curr;

    if (curr->item == item) {
      delete(n); // was never in list
      return false;
    }

    n->next = curr;

    // unmark new node
    resetflag(&n->next);
    resetflag(&curr);

    if (compare_exchange(&pred->next,
                         &curr, n))
      return true;
  }
}
```

```
bool remove(T item) {
  Window w;
  while (true) {
    w = find(item);
    if (item != w.curr->item)
      return false;

    Node *succ = w.curr->next;
    Node *markedsucc = succ;
    // mark as deleted
    setflag(&markedsucc);
    resetflag(&succ);
    if (!compare_exchange(&w.curr->next,
                          &succ,
                          markedsucc))
      continue;

    // attempt to unlink curr
    compare_exchange(&w.pred->next,
                     &w.curr, succ);
    return true;
}
```

Lock-free list-based set: Invariants

- Nodes may be traversed even though they have been removed
- We rely on garbage collection for removed nodes
- Sentinels are never added or removed
- Nodes are sorted by the item
- Each item occurs exactly once
- An item is in the set iff it is reachable from head and not marked

- `add` is lock-free
- `remove` is lock-free
- `contains` is wait-free

Argument: If a `compare_exchange` in `add` or `remove` fails, then some other thread has made progress.

- Each failure to mark/unlink or add item causes restart from head!

Linearization points:

Outcome: Item in set

`add` Earlier of

- Point where found node is checked against `item`
- Point immediatly before node is marked (concurrent `remove`)

`remove` Compare-exchange on marked flag

`contains` Read of `marked` flag

Lock-free list-based set: Linearization

Linearization points:

Outcome: Item not in set

`add` Compare-exchange

`remove` Earlier of

- Point where found node is checked against `item`
- Point immediatly before new node is inserted (concurrent `add`)

`contains` Earlier of

- Point where a removed matching node, or a node with `node->item` > `item` is found
- Point immediatly before a new matching node is added to list

Lock-free list-based set: Improvements

Instead of restarting from head on compare-exchange (CAS) failure, maintain approximate backward links, traverse backwards and restart from close predecessor node instead. Can save (significantly?) on the cost of restart when list is long.

- Mikhail Fomitchev, Eric Ruppert: Lock-free linked lists and skip lists. PODC 2004: 50-59

(with no benchmark results)

The `find` operation needs to restart only if predecessor node has been deleted (marked).

```
Window find(T item) {
  // Search for item or successor
retry:
  Node* pred = head;
  Node* curr = getpointer(pred->next);

  while (true) {
    Node *succ = getpointer(curr->next);
    while (getflag(curr->next)) {
      resetflag(&curr);
      resetflag(&succ);
      if (!compare_exchange(&pred->next,
                            &curr, succ))
      {
        if (getflag(pred->next))
          goto retry;
        else succ = pred->next;
      }

      curr = succ;
      succ = getpointer(succ->next);
    }

    ... as before
```

Lock-free list-based set: Simple improvement (delete)

The `delete` operation only needs to ensure that the found item has become marked. Only one traversal of list.

Linearizes earlier (no `find` to clean up and discover that other thread successfully deleted).

```
bool remove(T item) {
  Window w;

  w = find(item);
  if (item != w.curr->item) return false;

  Node *succ = w.curr->next;
  Node *markedsucc = succ;
  // mark as deleted
  setflag(&markedsucc);
  resetflag(&succ);
  while (!compare_exchange(&w.curr->next,
                           &succ,
                           markedsucc)) {
    if (getflag(w.curr->next))
      return false;
  }

  // attempt to unlink curr
  compare_exchange(&w.pred->next,
                   &w.curr, succ);
  return true;
}
```

The simple improvements were suggested and have been implemented in:

> Jesper Larsson Träff, Manuel Pöter: A more pragmatic implementation of the lock-free, ordered, linked list. PPoPP 2021: 457-459

with concrete code available via `https://github.com/parlab-tuwien/lockfree-linked-list`.

Another observation: For finding the point from which to start `find`, only aproximate predecessor pointers are needed.

Can these improvements and observations be used for similar data structures (skip lists, hash tables)?

Which implementation works best depends on the use-case, all ideas have merits and drawbacks.

- Coarse-grained synchronization (lock, synchronized regions, ...) can be used with any (sequentially efficient!) data structure (black box). Can easily become bottleneck when data structure operations are frequent
- All other implementations require deep knowledge of the data structure implementation, need care to achieve correctness, liveness (deadlock-freedom), and performance
- Lock- and wait-free data structures provide theoretically best progress guarantees, but come at a cost (expensive atomic operations, frequent need to restart validation)

References

- Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. SPAA 2002: 73–82.
- Timothy L. Harris: A Pragmatic Implementation of Non-blocking Linked-Lists. DISC 2001: 300–314
- John D. Valois: Lock-Free Linked Lists Using Compare-and-Swap. PODC 1995: 214–222
- Jesper Larsson Träff, Manuel Pöter: A more pragmatic implementation of the lock-free, ordered, linked list. PPoPP 2021: 457–459.