departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Alternative Synchronization Strategies — Lazy Locking

lecture 18 (2020-05-05)

Master in Computer Science and Engineering

— Concurrency and Parallelism / 2019-20 —

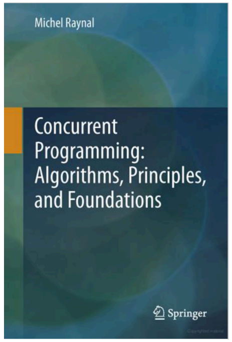
João Lourenço <joao.lourenco@fct.unl.pt>

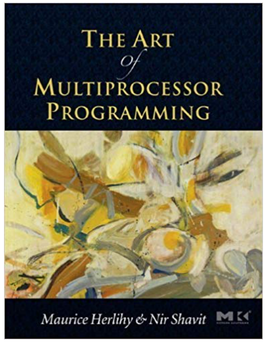
Alternative Synchronization Strategies

- Contents:
  - Liveness: Types of Progress
  - Coarse-Grained Synchronization
  - Fine-Grained Synchronization
  - Optimistic Synchronization
  - Lazy Synchronization
  - Lock-Free Synchronization

Past lectures

Today





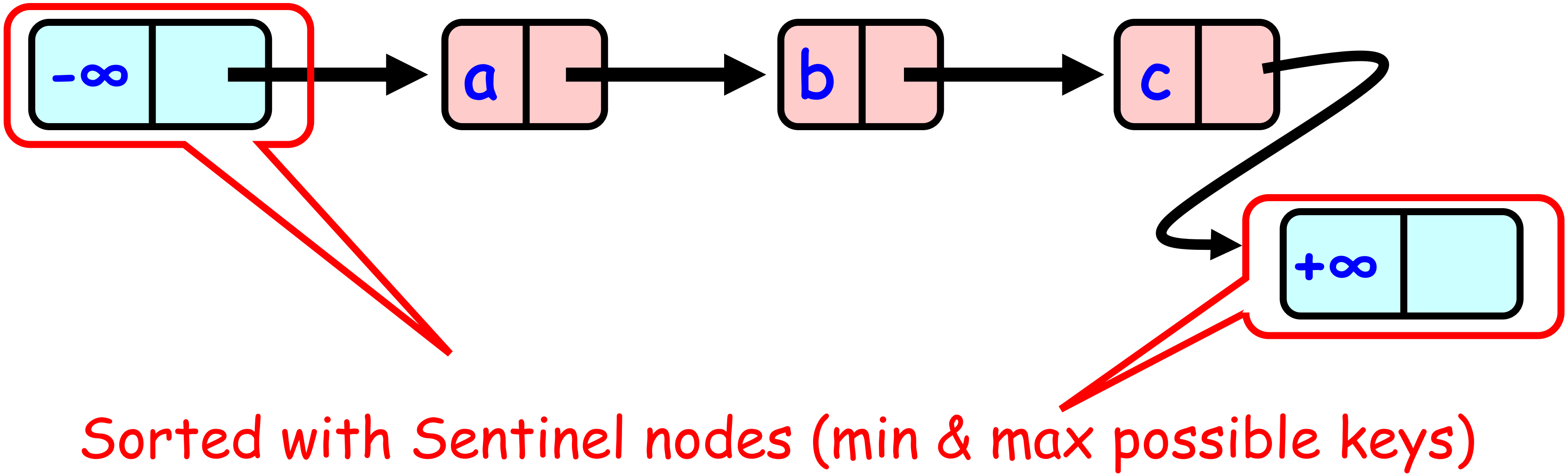
- Reading list:
  - chapter 5 of the Textbook
  - Chapter 9 of "The Art of Multiprocessor Programming" by Maurice Herlihy & Nir Shavit *(available at clip)*

Lazy Synchronization

- Procrastinate! Procrastinate! Procrastinate! ☺
- Make common operations fast
- Postpone hard work
  - E.g., removing components is tricky… use two phases:
    - Logical removal
      - Mark component to be deleted
    - Physical removal
      - Do what needs to be done to remove the component
- Evaluation
  - ✓ Recheck after locking is simpler (just that nodes are unmarked)
  - ✓ Also usually cheaper than hand-over-hand locking
  - ✗ Mistakes are expensive (safety easily compromised)
  - ✗ Is not starvation free on add and remove (liveness compromised)
  - ✓ Is starvation free on contains

Linked List

- Illustrate these patterns …
- Using a list-based Set
  – Common application
  – Building block for other apps



Set Interface

- Unordered collection of items
- No duplicates
- Methods
  - *add(x)* put x in set *true if x was not in the set*
  - *remove(x)* take x out of set *true if x was in the set*
  - *contains(x)* tests if x in set *true if x is in the set*

List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

Add item to set

List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(Tt x);
}
```

Remove item from set

List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

Is item in set?

List Node

```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

List Node

```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

item of interest

List Node

```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

Usually hash code

List Node

```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

Reference to next node

Optimistic Concurrency List

- Works best if the cost of traversing the list twice without locking is significantly less than the cost of traversing the list once with locking.
- One drawback of this Optimistic Concurrency List algorithm is that `contains()` needs to acquire locks, which is unattractive since `contains()` calls are likely to be much more common than calls to other methods.

Lazy Concurrency List

- Refine the Optimistic Concurrency List algorith so that…
- Calls to `contains()` are wait-free
- The `add()` and `remove()` methods, while still blocking, traverse the list only once (in the absence of contention)

Lazy Concurrency List HOWTO

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

Lazy Concurrency List HOWTO

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

Lazy Validate

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

Validate do not traverse the list anymore.
Just check if nodes are nor marked as deleted
and that 'pred.next' still points to 'curr'

Lazy Add

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

Calculate hash

Try until
success or failure

Lazy Add

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

Initialize pointers to traverse the list

Traverse the list looking for ‘item’

Lock the nodes

Try the operation and either succeed or fail

Always unlock (with both success and failure)

Lazy Add

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

If any of the nodes is marked as deleted then restart the operation

If item already in list, fail

If item not present, create new node insert into the list, and succeed

Lazy Remove

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

Calculate hash

Try until success or failure

Lazy Remove

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

Initialize pointers to traverse the list

Traverse the list looking for 'item'

Lock the nodes

Try the operation and either succeed or fail

Always unlock (with both success and failure)

Lazy Remove

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

If any of the nodes is marked as deleted then restart the operation

If item not in list, fail

If item is present,
first mark it as deleted (logical delete)
and then remove it (physical dele)

Optimistic Contains

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

No while (ture) loop anymore!
Contains always returns.

Optimistic Contains

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

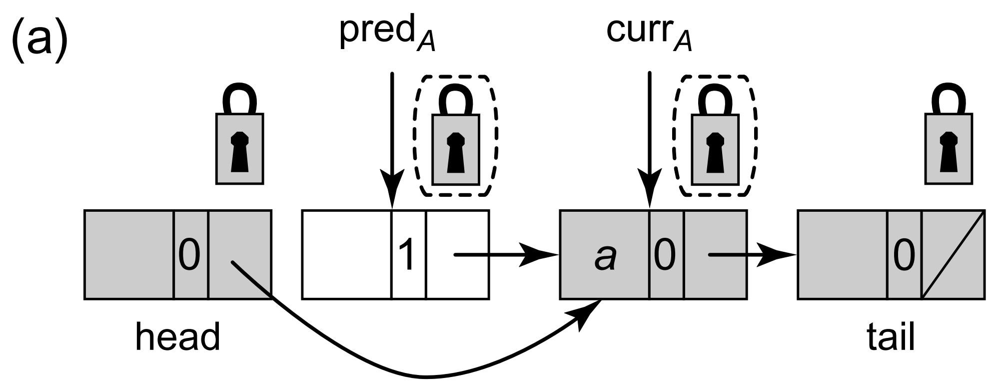
Calculate hash

Start traversing the list from the beinning

Traverse the list looking for 'item'

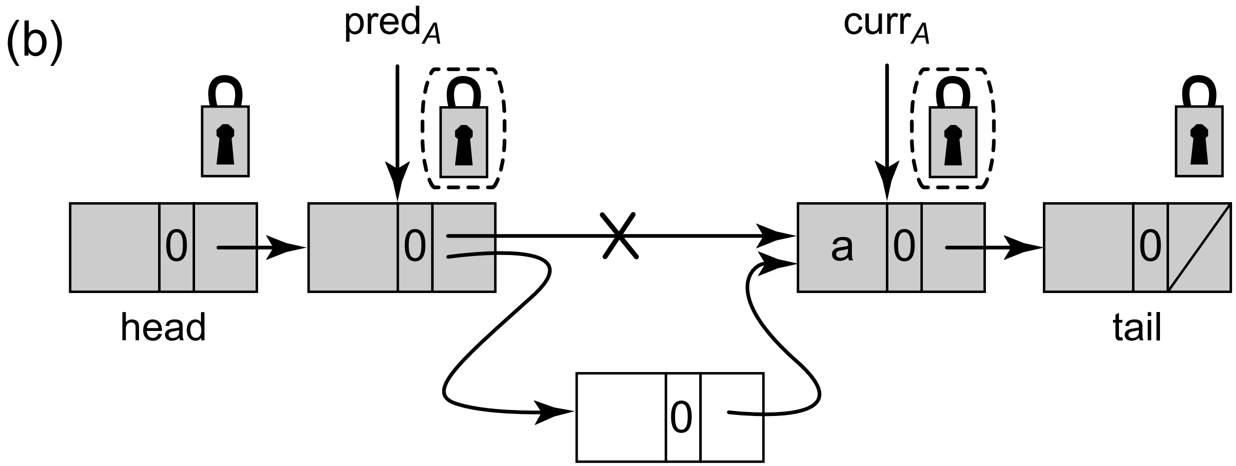
Return true is item was found and is nor marked as deleted

Why validation is still necessary?



- Thread `A` is attempting to remove node `a`. After it reaches the point where $\texttt{pred}_A$ refers to $\texttt{curr}_A$, and before it acquires locks on these nodes, the node $\texttt{pred}_A$ is logically and physically removed. After `A` acquires the locks, validation will detect the problem and `A`'s call to `remove()` will be restarted.

Why validation is still necessary?

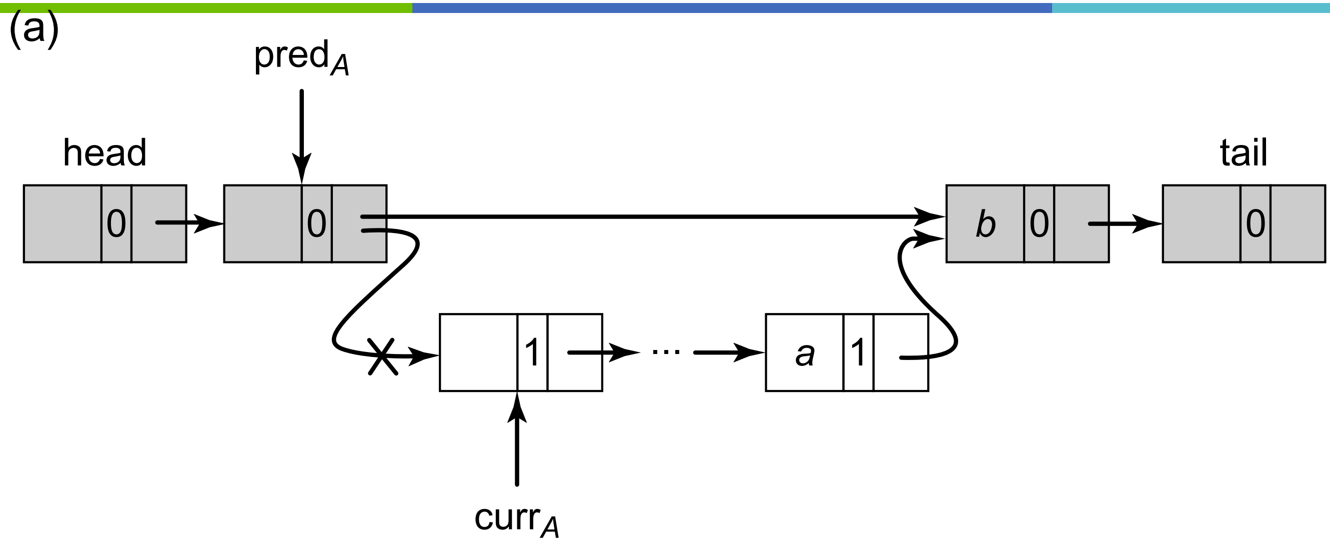


- Thread **A** is attempting to remove node **a**. After it reaches the point where $\texttt{pred}_A$ refers to $\texttt{curr}_A$, and before it acquires locks on these nodes, a new node is added between $\texttt{pred}_A$ and $\texttt{curr}_A$. After **A** acquires the locks, even though neither $\texttt{pred}_A$ or $\texttt{curr}_A$ are marked, validation detects that $\texttt{pred}_A\texttt{.NEXT}$ is not the same as $\texttt{curr}_A$, and **A**'s call to `remove()` will be restarted.

Lazy List linearization points

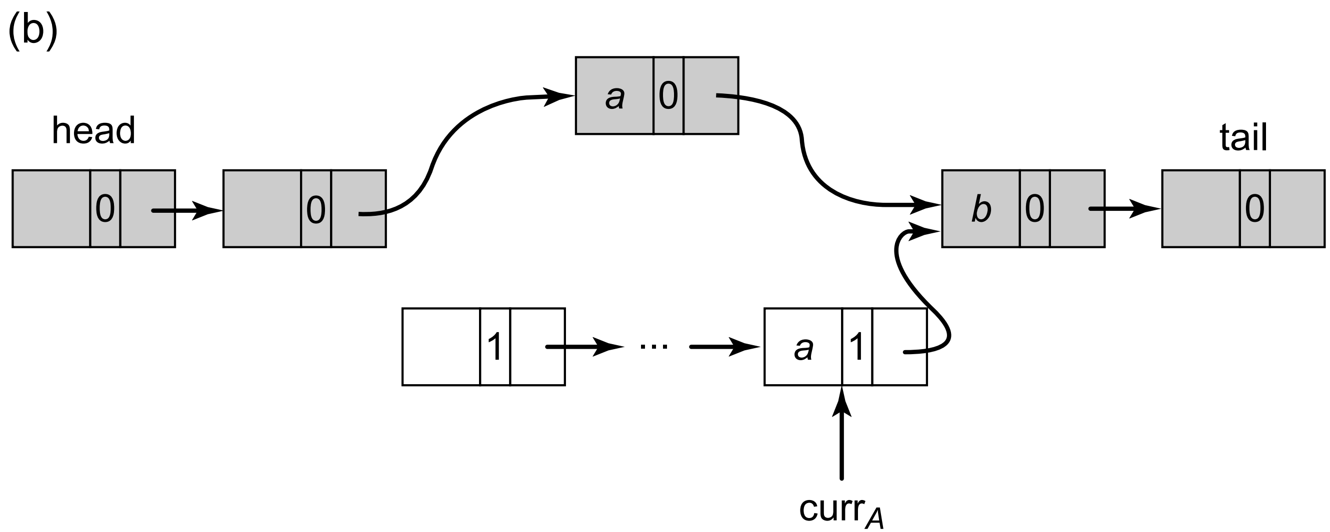
- `add()` — linearized when the first lock is removed (before returning)
- Failed `remove()` — linearized when the first lock is removed (before returning)
- Successful `remove()` — linearized when the mark is set
- Successful `contains()` — linearized when an unmarked matching node is found
- Failed `contains()` — ??

Lazy List linearization of a failed contains()



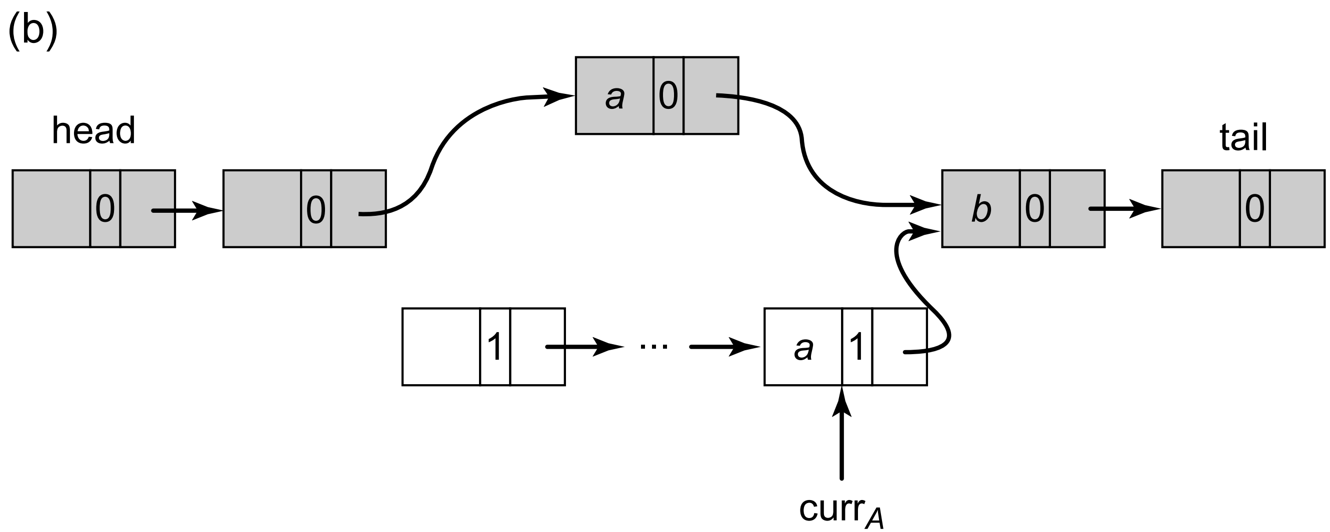
- While thread A is traversing the list, a concurrent `remove()` call disconnects the sublist referred to by `curr`. Notice that nodes with items a and b are still reachable, so whether an item is actually in the list depends only on whether it is not marked. Thread A's call is linearized at the point when it sees that node a is marked and is no longer in the abstract set.

Lazy List linearization of a failed contains()



- While thread **A** is traversing the list leading to marked node **a**, another thread adds a new node with key **a**. It would be wrong to linearize thread **A**'s unsuccessful `contains()` call to when it found the marked node **a**, since this point occurs after the insertion of the new node with key **a** to the list.

Lazy List linearization of a failed contains()



- An unsuccessful contains() method call is linearized within its execution interval at the earlier of the following points:
  - (1) the point where a removed matching node, or a node with a key greater than the one being searched for is found, and
  - (2) the point immediately before a new matching node is added to the list

The END