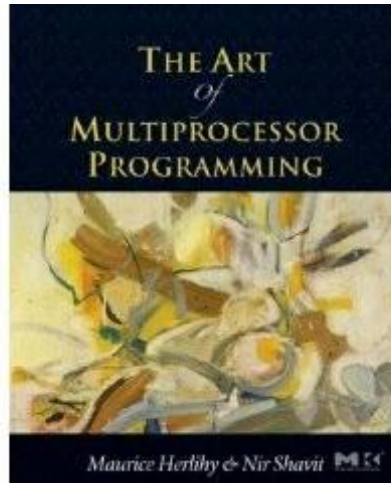


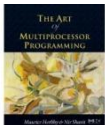
# Concurrent Skip Lists



Companion slides for  
The Art of Multiprocessor Programming  
by Maurice Herlihy & Nir Shavit

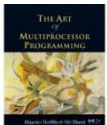
# Set Object Interface

- Collection of elements
- No duplicates
- Methods
  - add() a new element
  - remove() an element
  - contains() if element is present



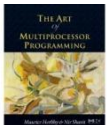
# Many are Cold but Few are Frozen

- Typically high % of contains() calls
- Many fewer add() calls
- And even fewer remove() calls
  - 90% contains()
  - 9% add()
  - 1% remove()
- Folklore?
  - Yes but probably mostly true



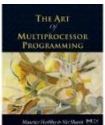
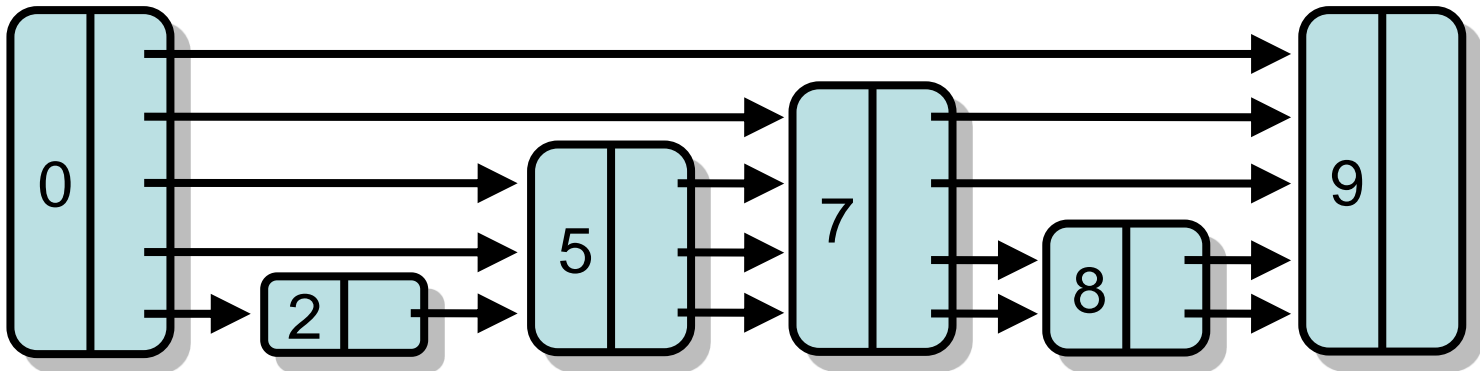
# Concurrent Sets

- Balanced Trees?
  - Red-Black trees, AVL trees, ...
- Problem: no one does this well ...
- ... because **rebalancing** after `add()` or `remove()` is a global operation



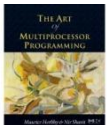
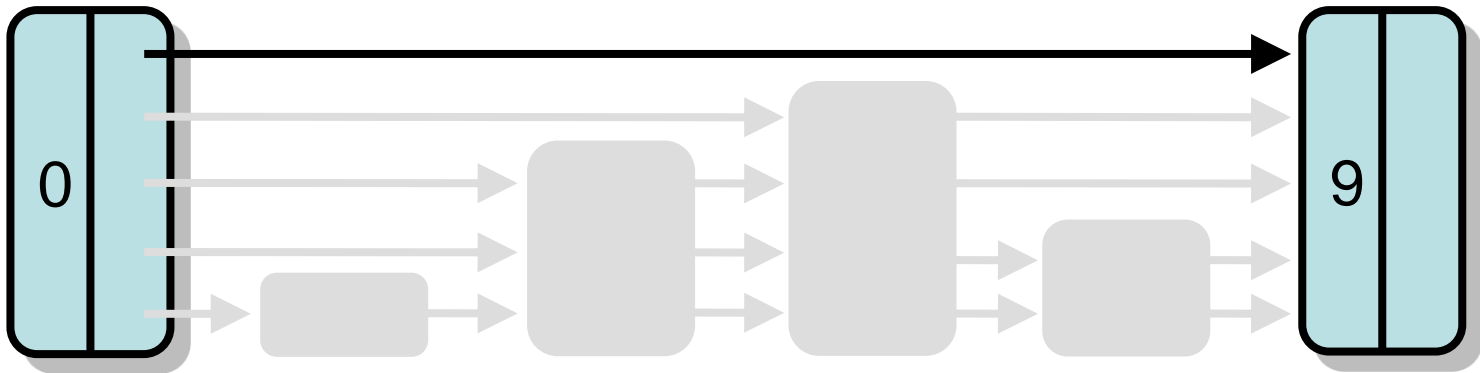
# Skip Lists

- Probabilistic Data Structure
- No global rebalancing
- Logarithmic-time search



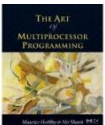
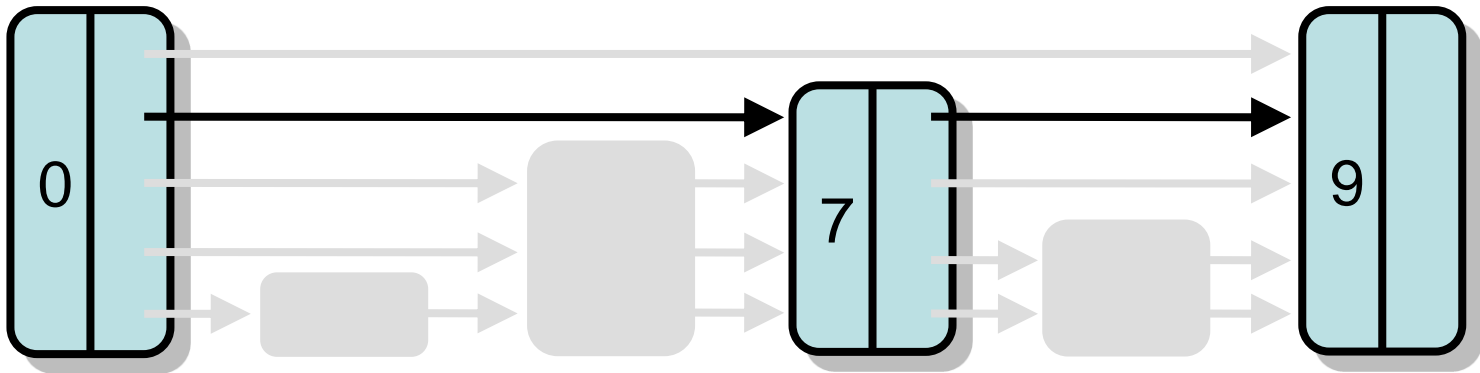
# Skip List Property

- Each layer is sub-list of lower levels



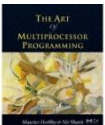
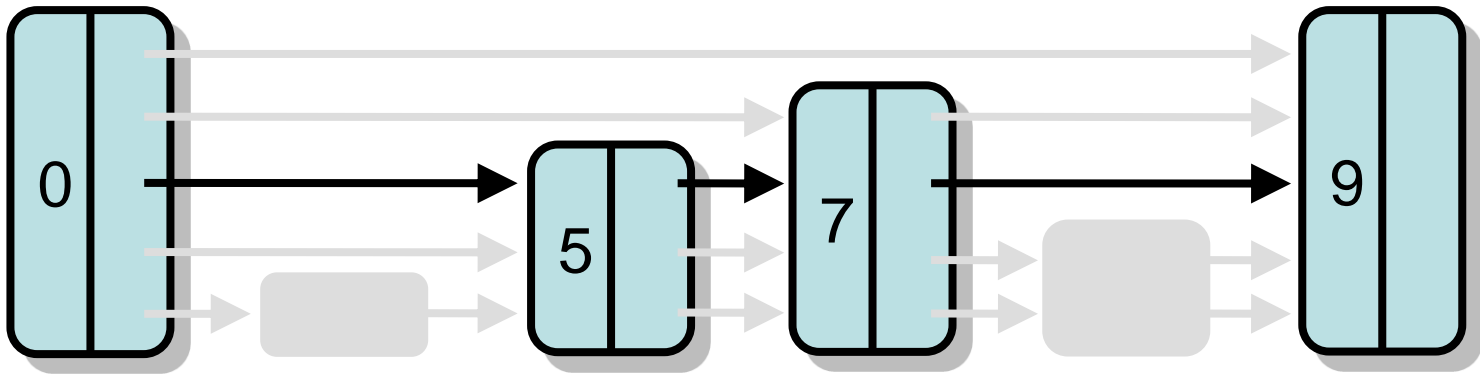
# Skip List Property

- Each layer is sub-list of lower-levels



# Skip List Property

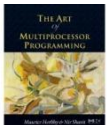
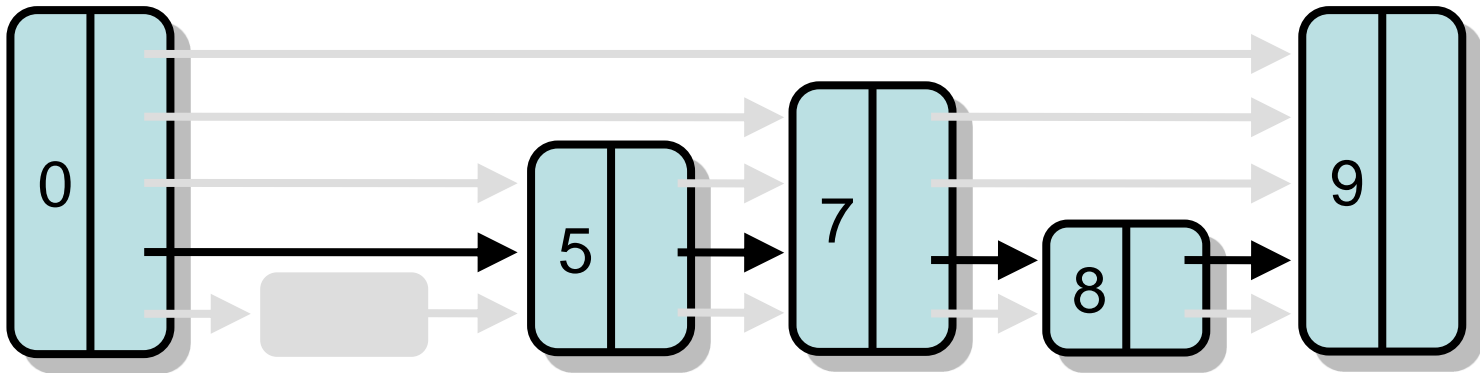
- Each layer is sub-list of lower levels





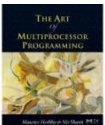
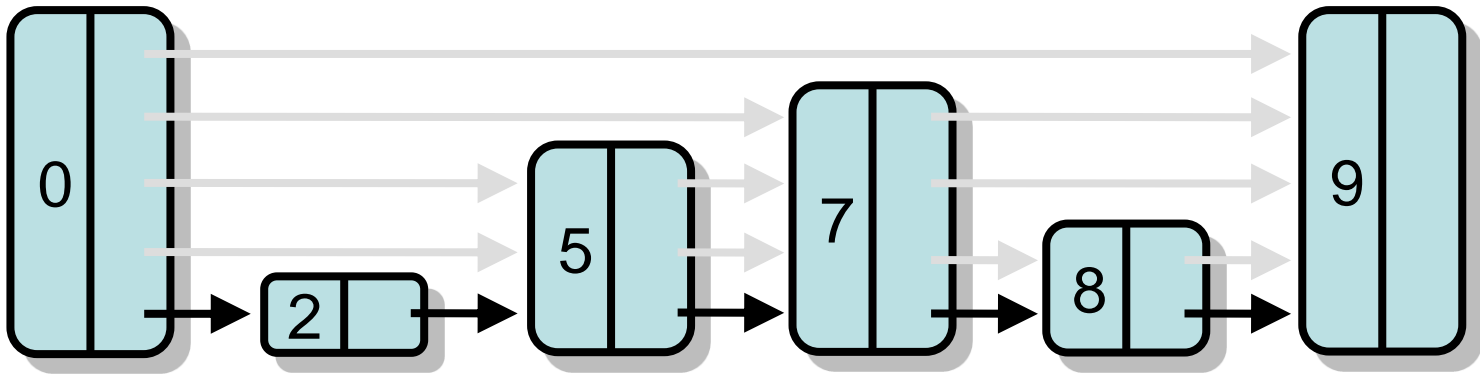
# Skip List Property

- Each layer is sub-list of lower levels



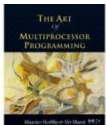
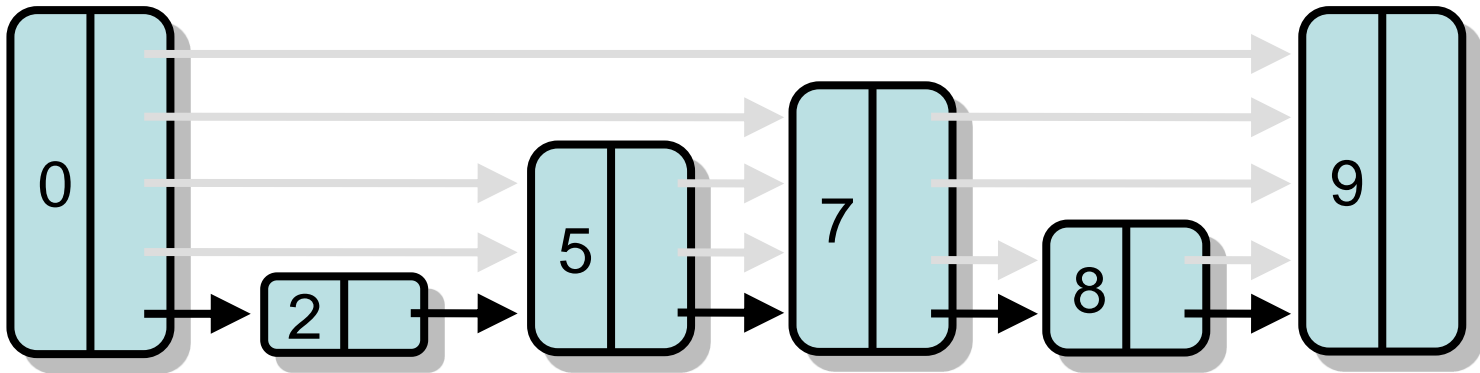
# Skip List Property

- Each layer is sub-list of lower levels
- Lowest level is entire list

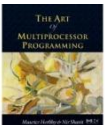
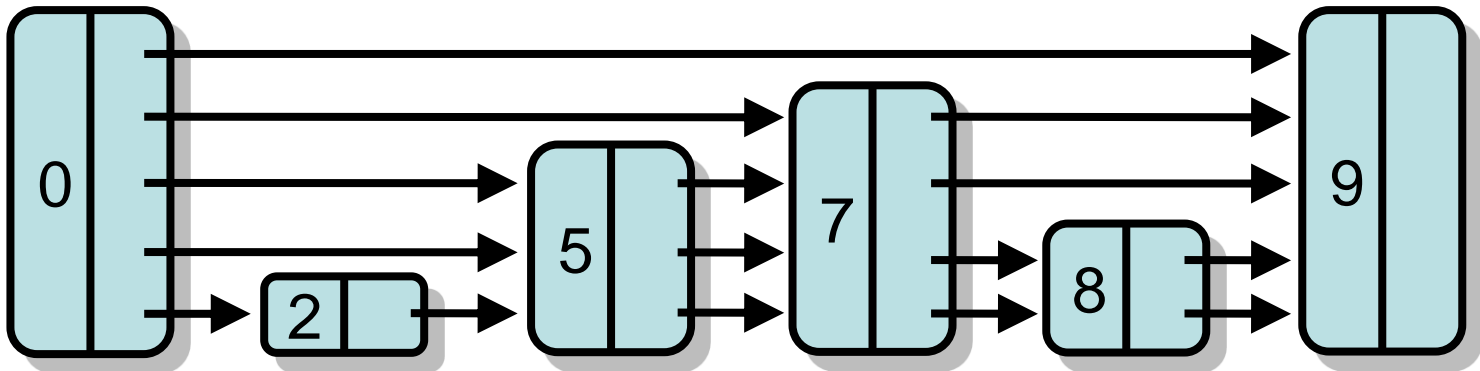
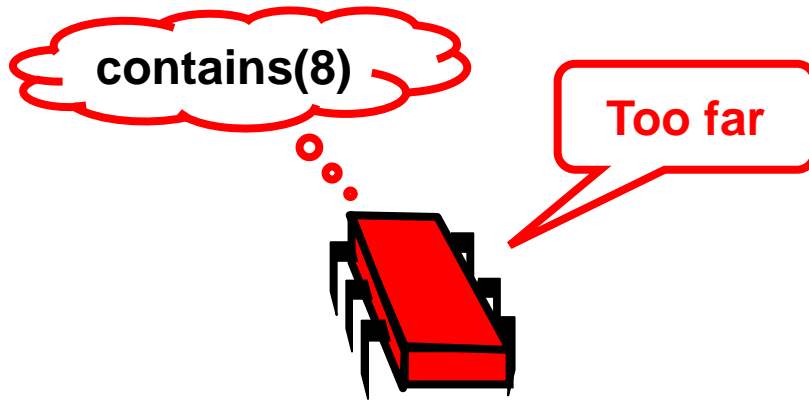


# Skip List Property

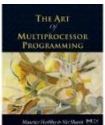
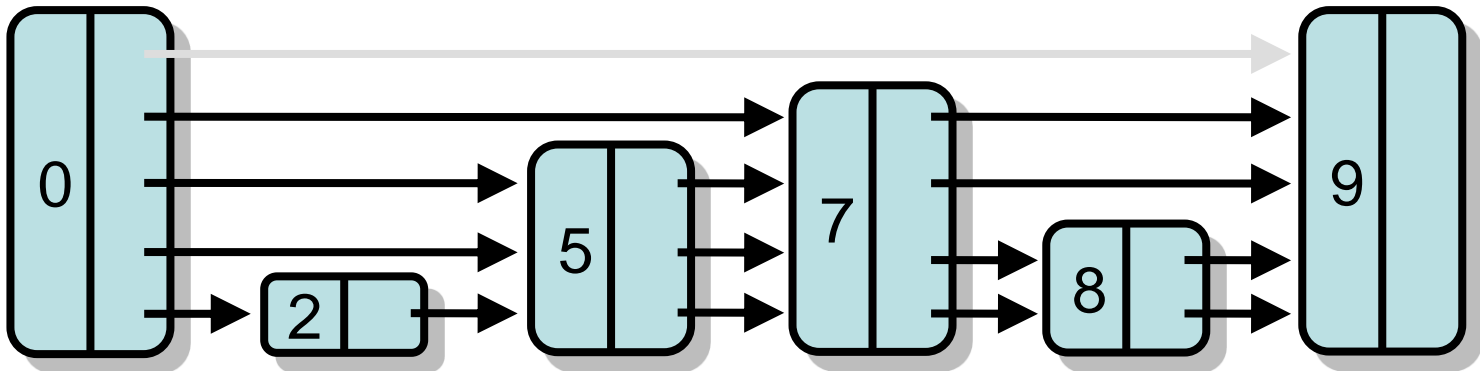
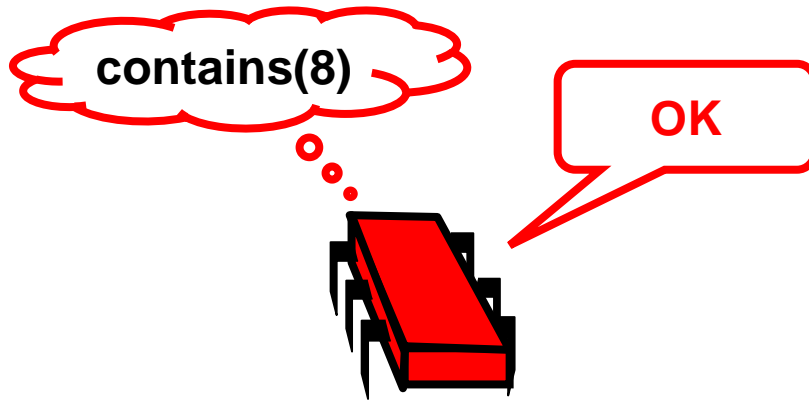
- Each layer is sub-list of lower levels
- Not easy to preserve in concurrent implementations ...



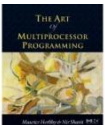
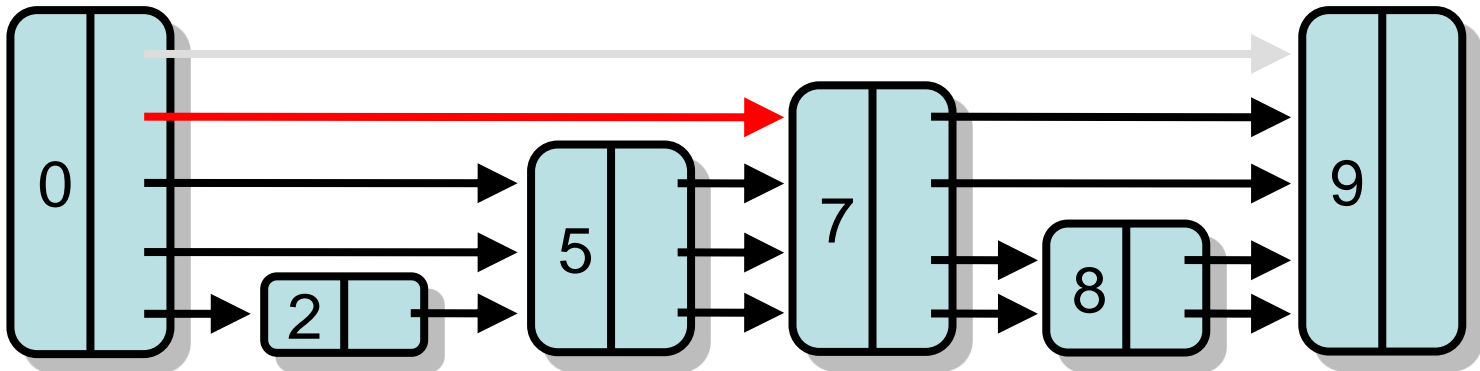
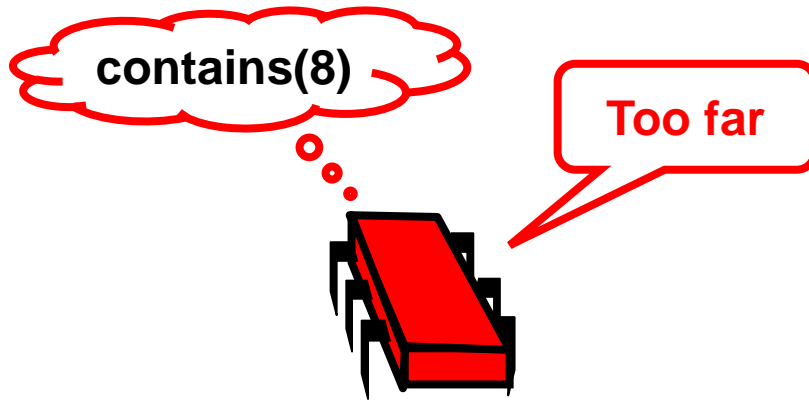
# Search



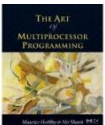
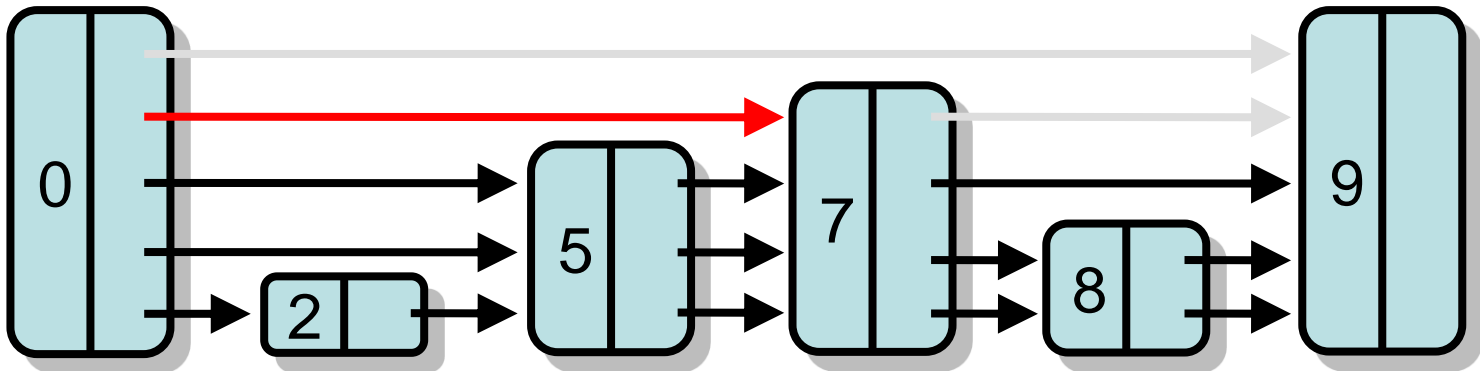
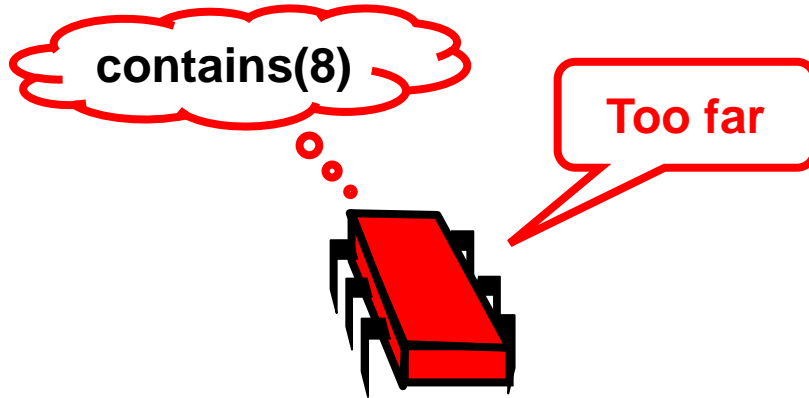
# Search



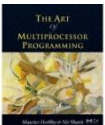
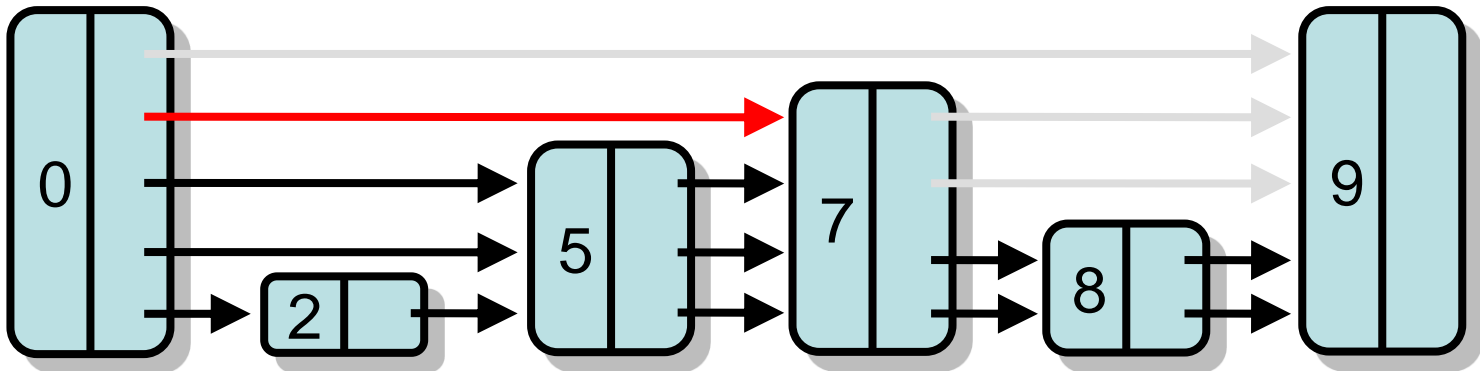
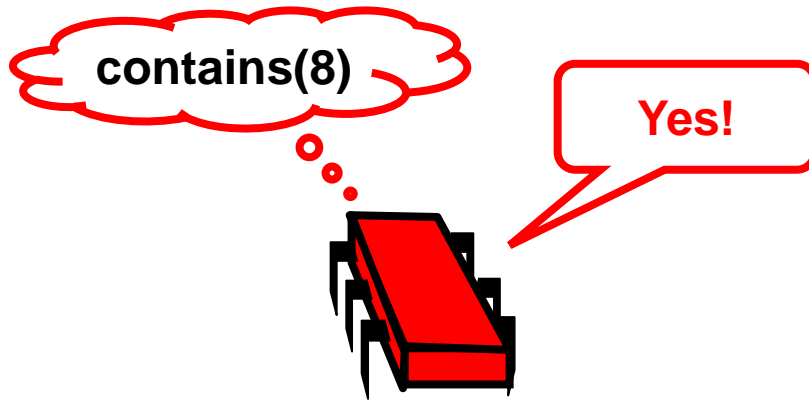
# Search



# Search

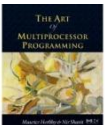
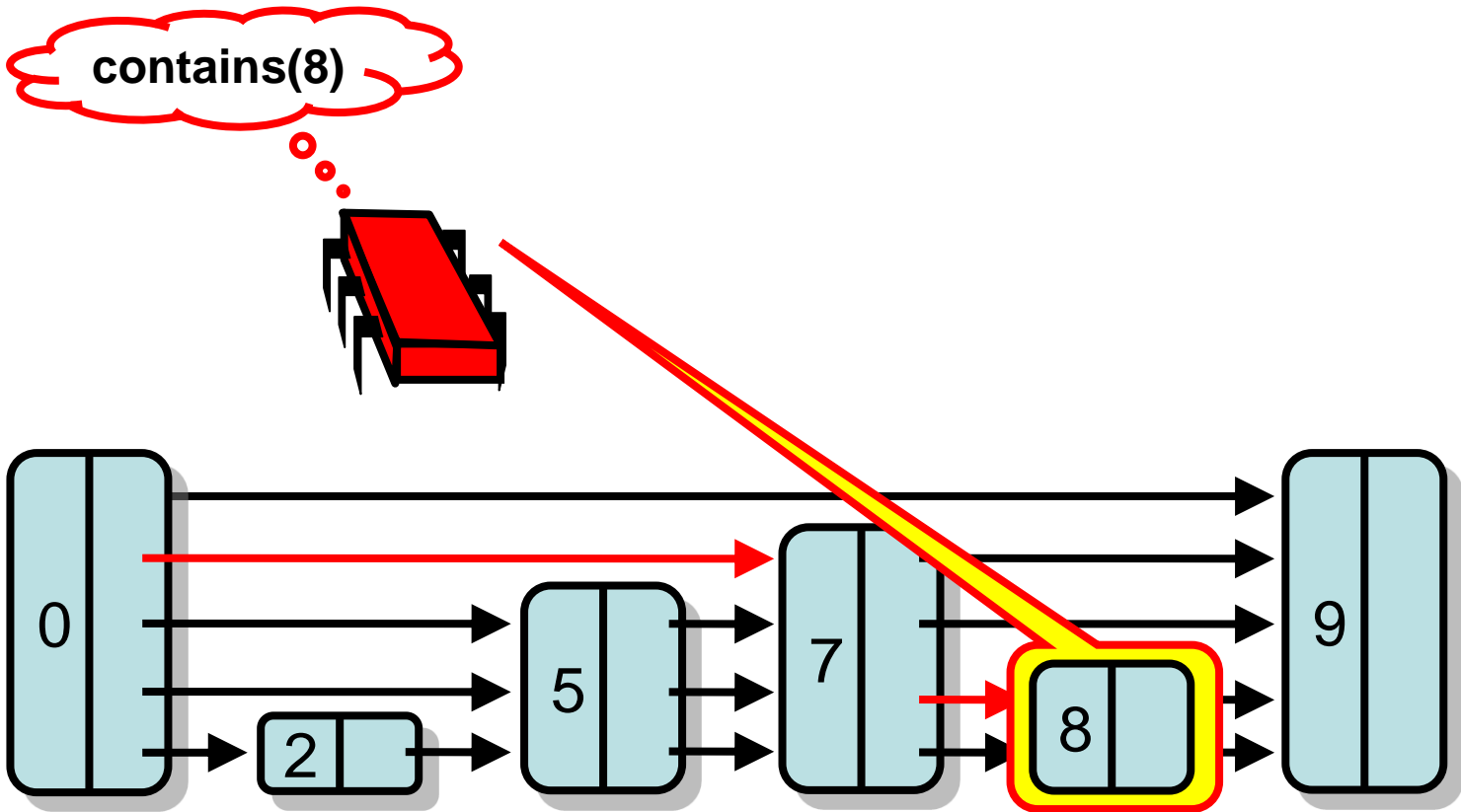


# Search

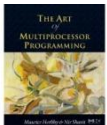
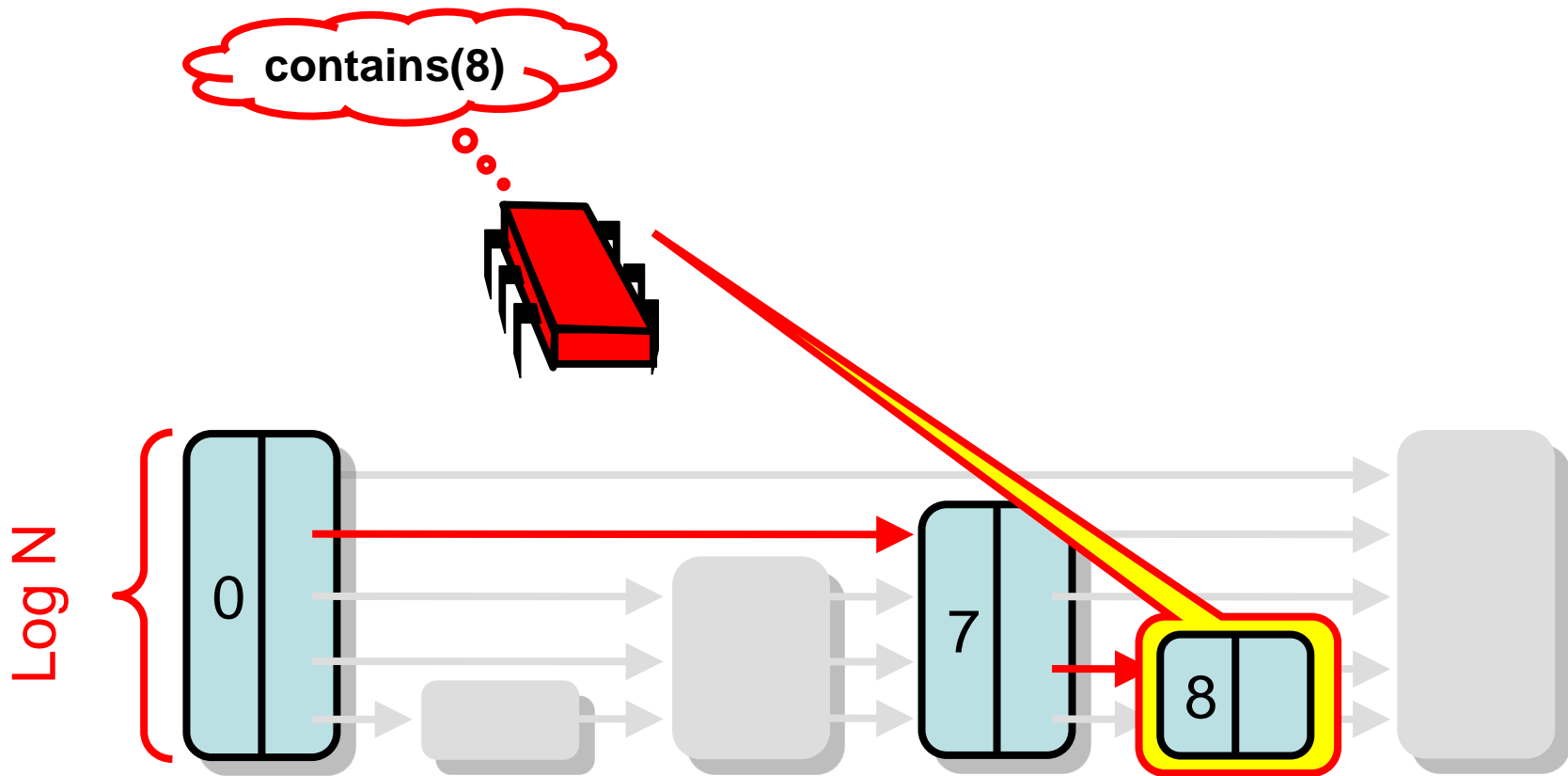




# Search

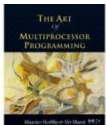
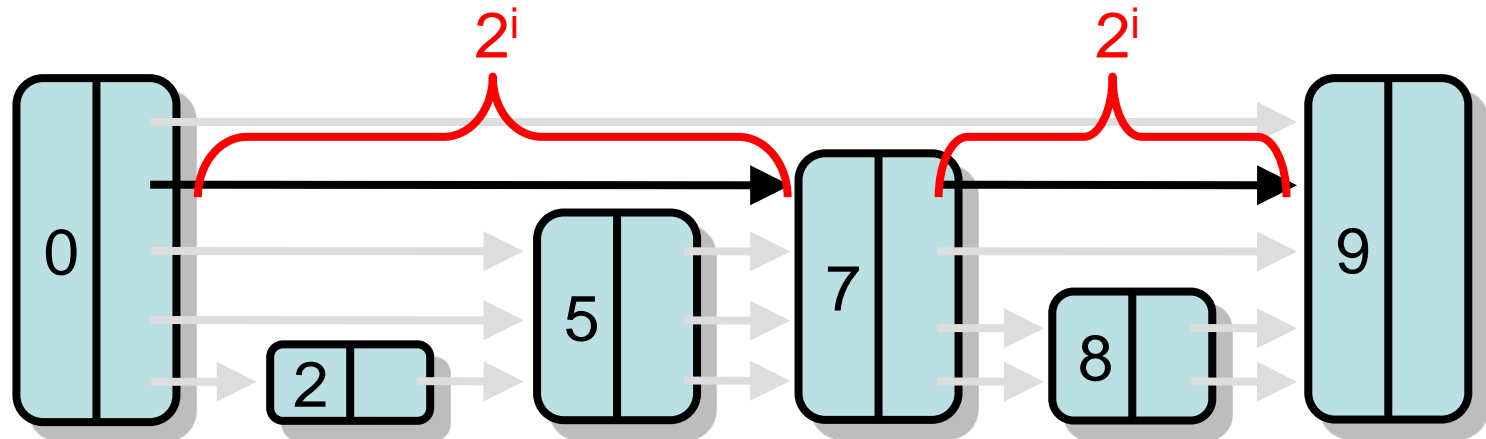


# Logarithmic



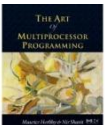
# Why Logarithmic

- Property: Each pointer at layer  $i$  jumps over roughly  $2^i$  nodes
- Pick node heights randomly so property guaranteed probabilistically



# Sequential Find

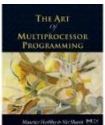
```
int find(T x, Node<T>[] preds, Node<T>[] succs) {  
    ...  
}
```



# Sequential Find

```
int find(T x, Node<T>[] preds, Node<T>[] succs) {
```

**object height  
(-1 if not there)**

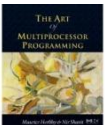


# Sequential Find

```
int find(T x, Node<T>[] preds, Node<T>[] succs) {
```

**object height  
(-1 if not there)**

**Object sought**



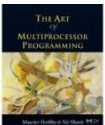
# Sequential Find

```
int find(T x, Node<T>[] preds, Node<T>[] succs) {
```

Object height  
(-1 if not there)

object sought

return predecessors



# Sequential Find

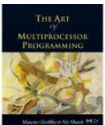
```
int find(T x, Node<T>[] preds, Node<T>[] succs) {
```

Object height  
(-1 if not there)

object sought

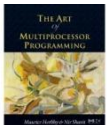
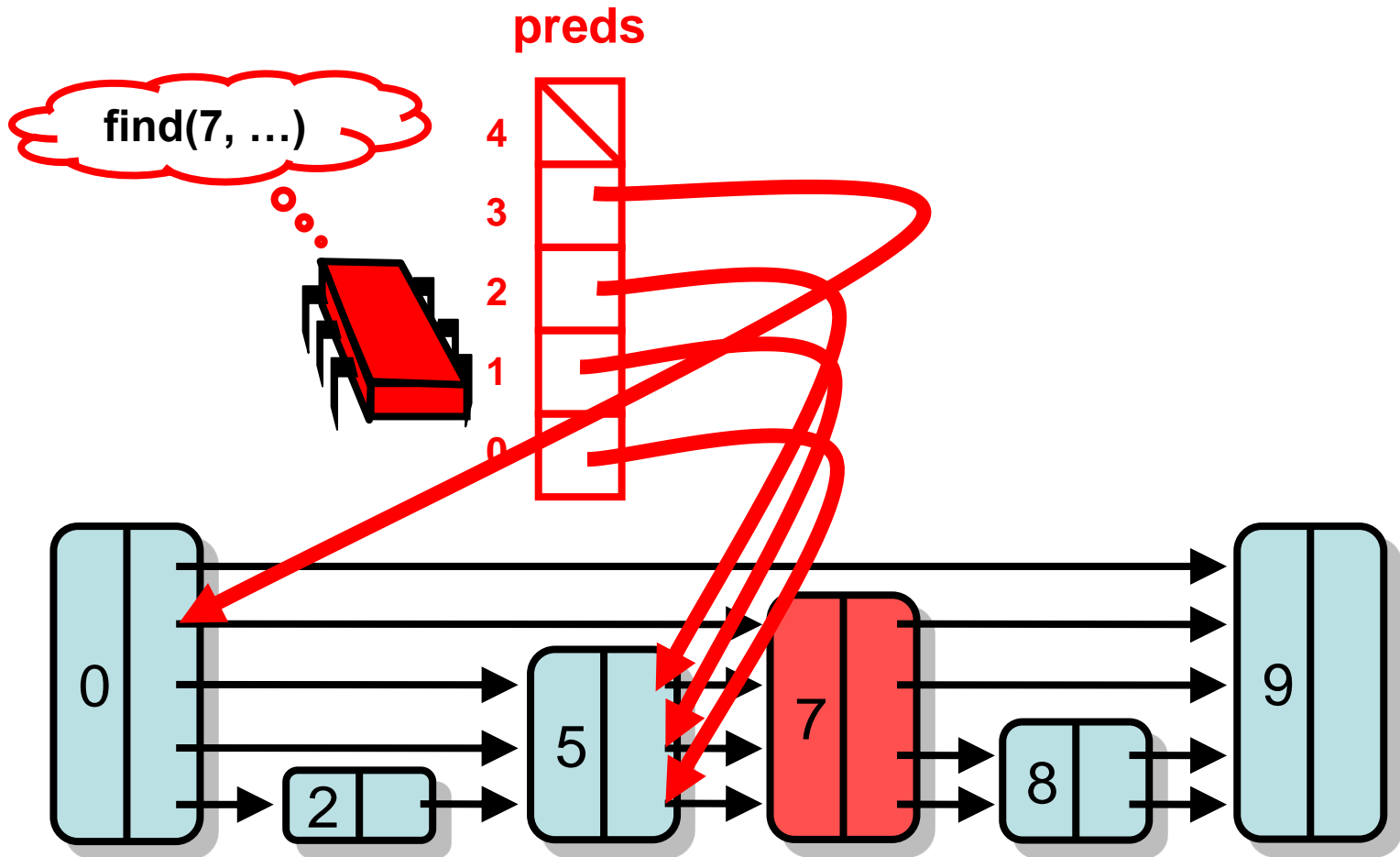
return predecessors

return successors

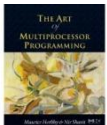
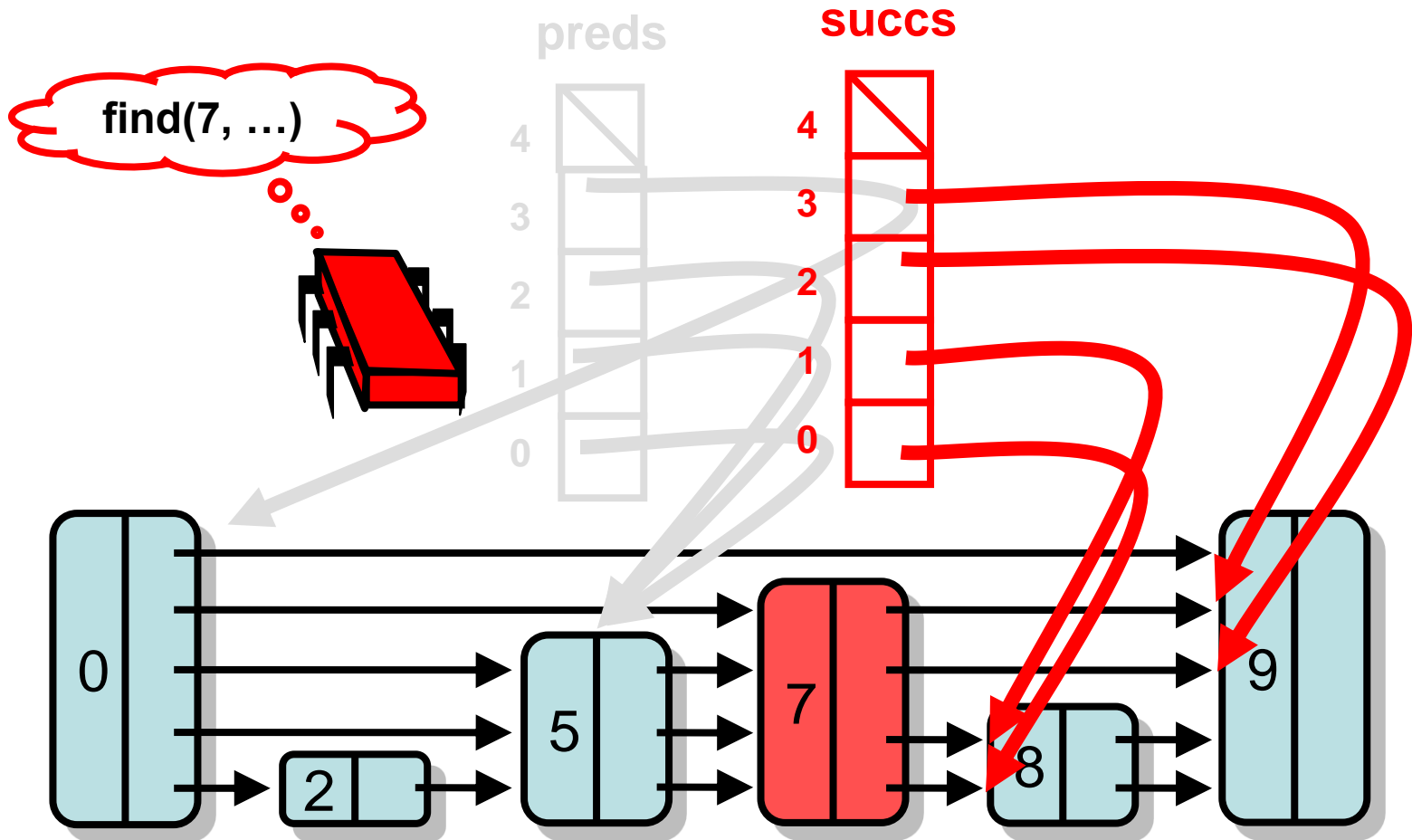




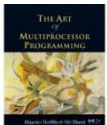
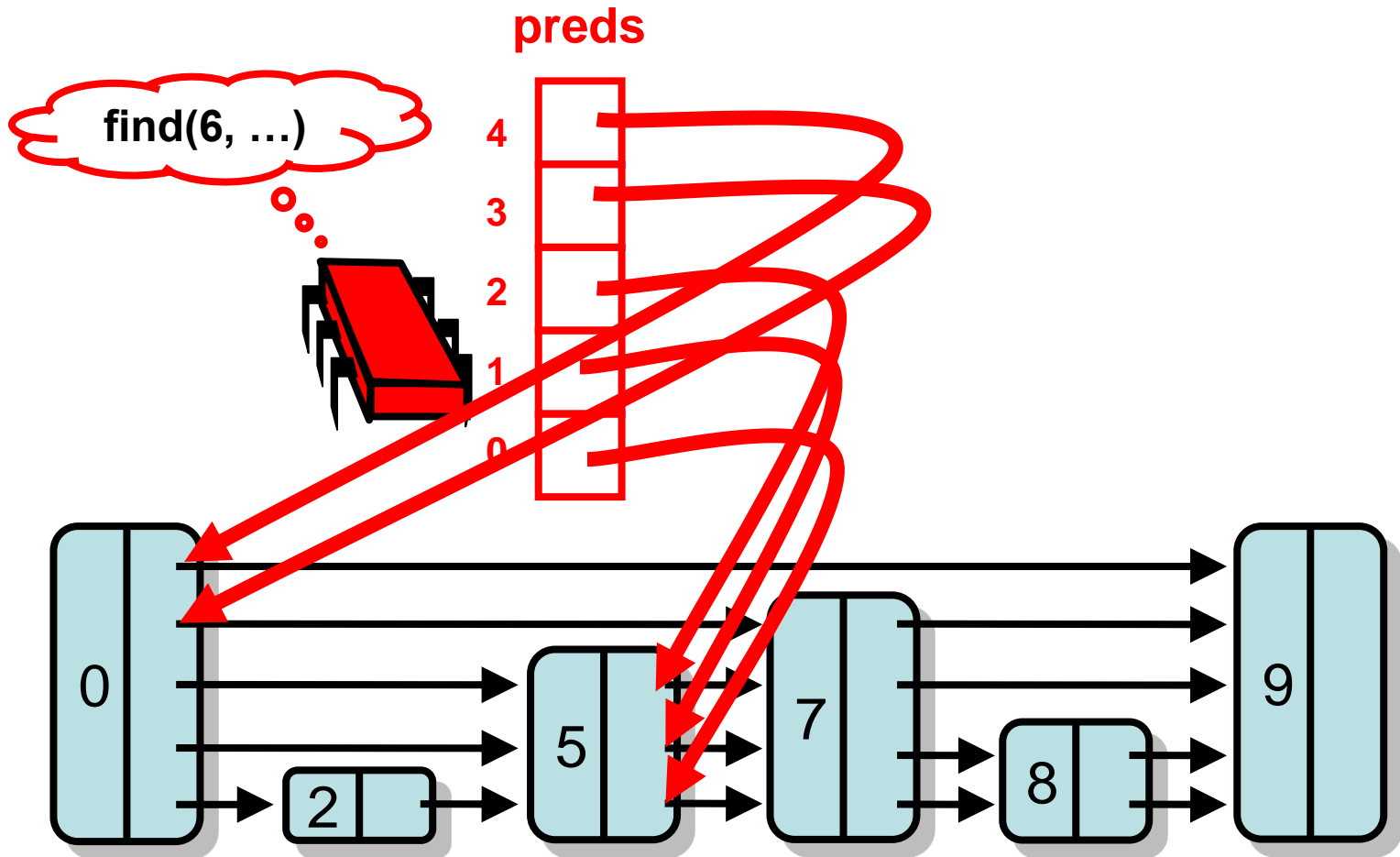
# Successful Search



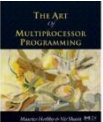
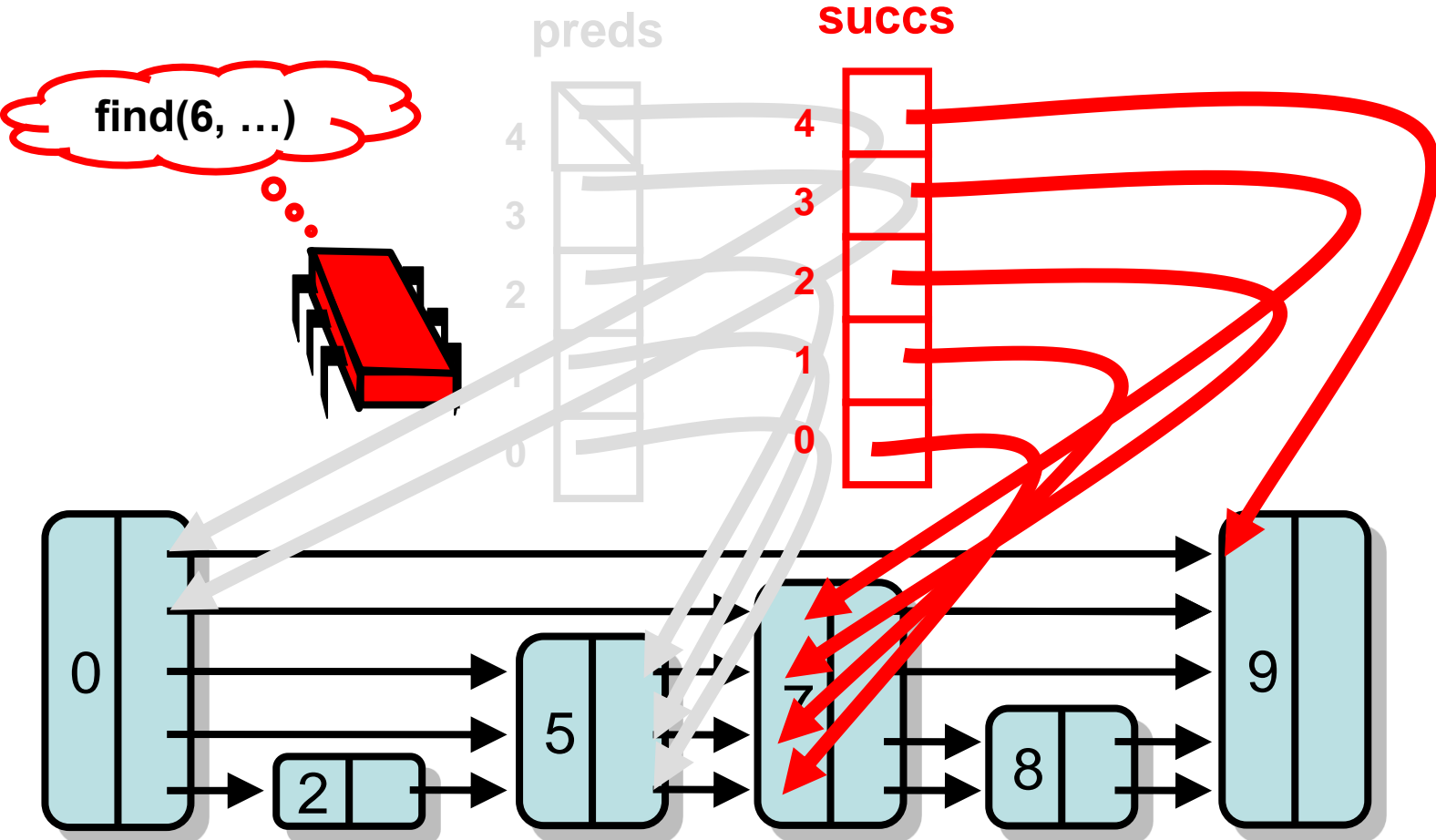
# Successful Search



# Unsuccessful Search

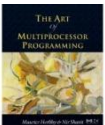


# Unsuccessful Search

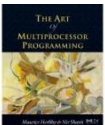


# Lazy Skip List

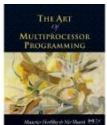
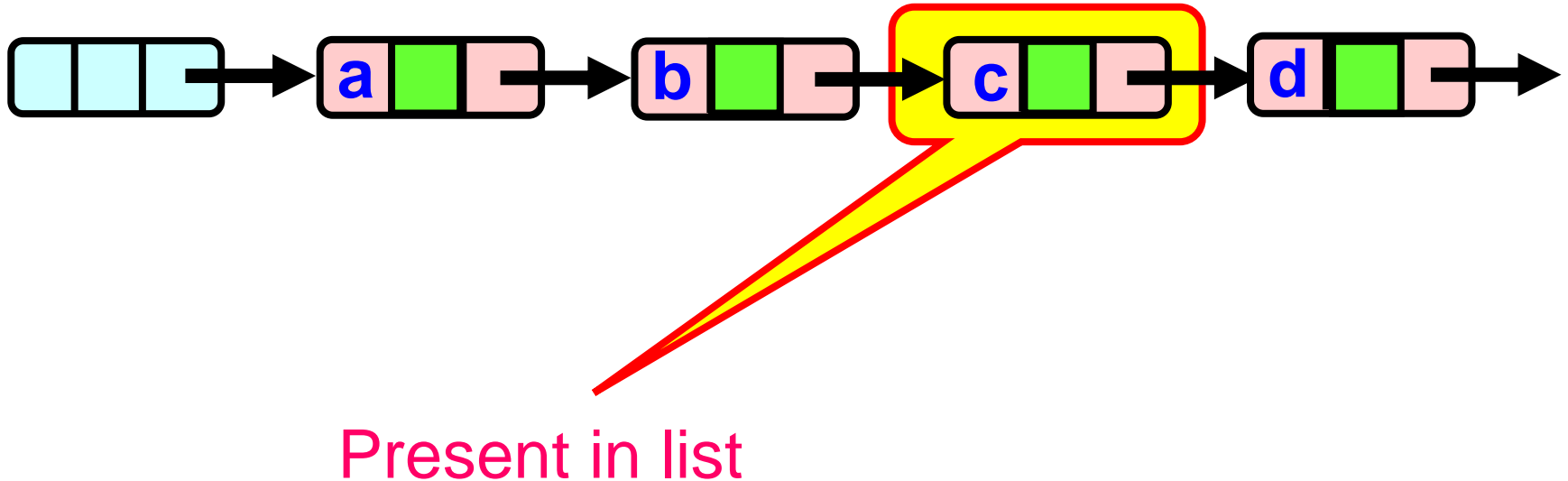
- Mix blocking and non-blocking techniques:
  - Use optimistic-lazy locking for `add()` and `remove()`
  - Wait-free `contains()`
- Remember: typically lots of `contains()` calls but few `add()` and `remove()`



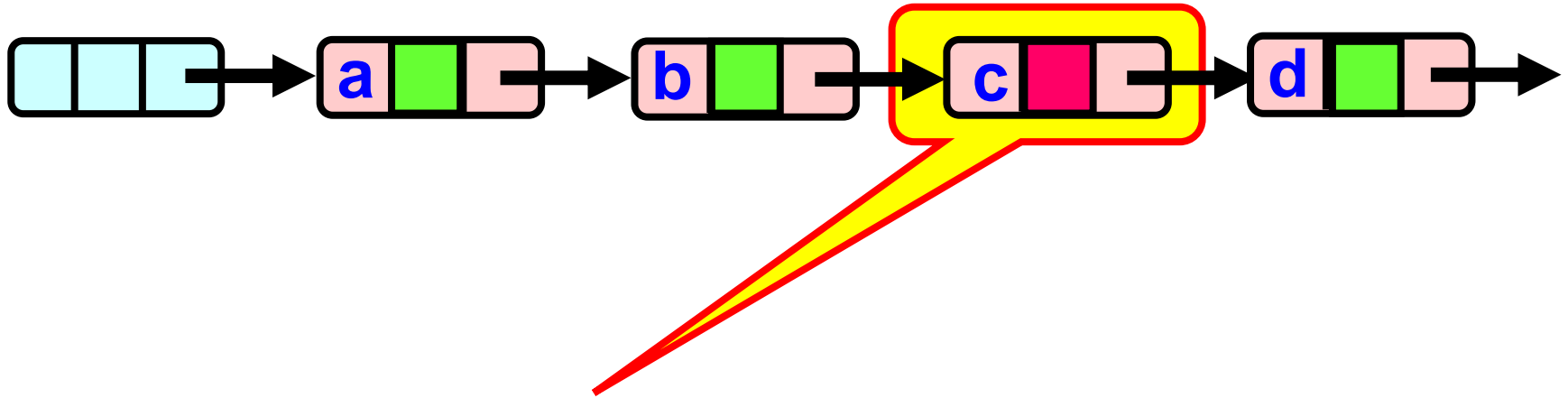
# Review: Lazy List Remove



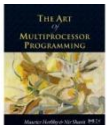
# Review: Lazy List Remove



# Review: Lazy List Remove

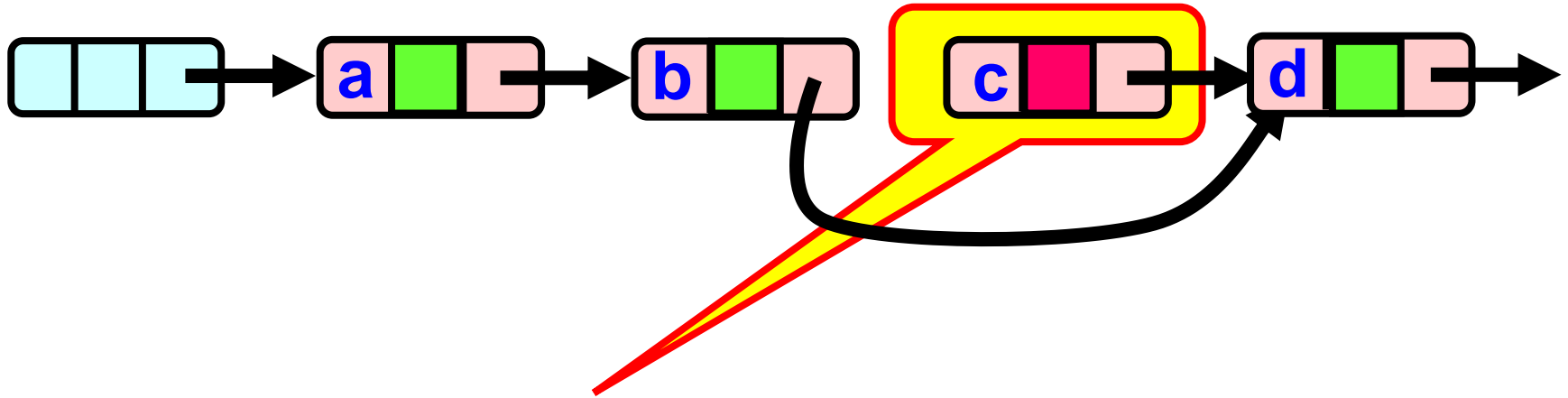


Logically deleted

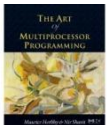




# Review: Lazy List Remove

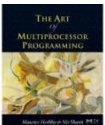
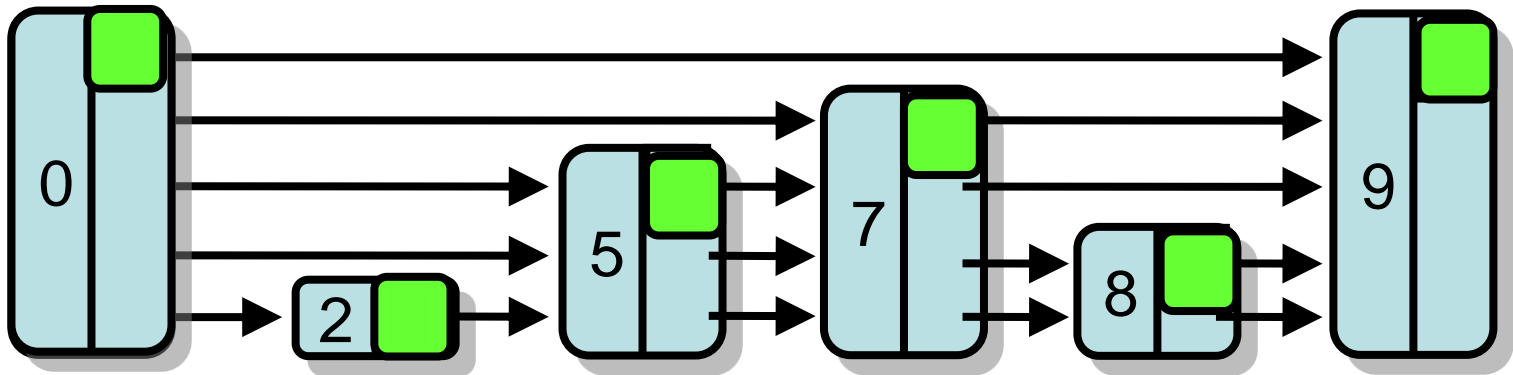


Physically deleted



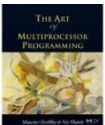
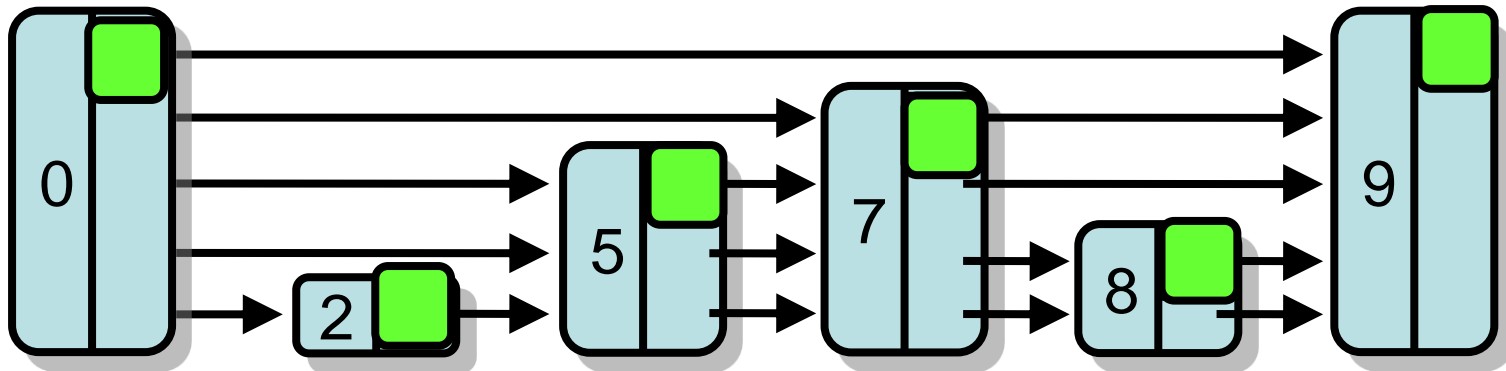
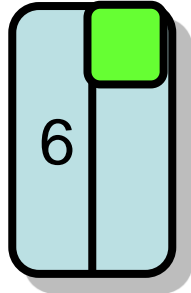
# Lazy Skip Lists

- Use a mark bit for logical deletion



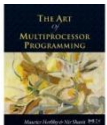
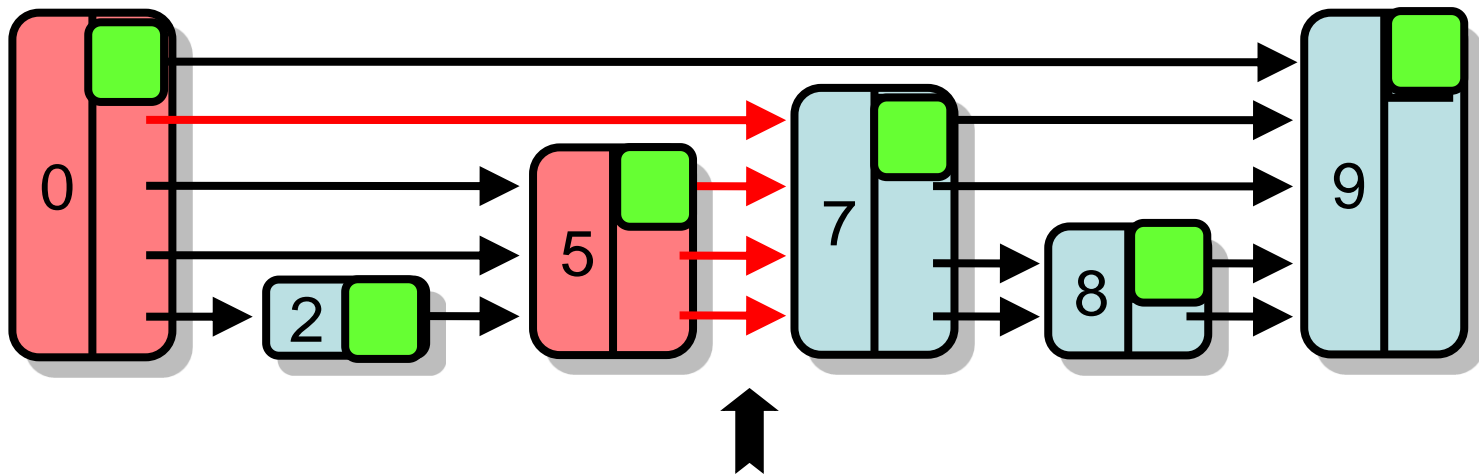
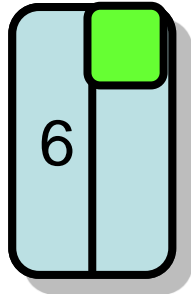
# add(6)

- Create node of (random) height 4



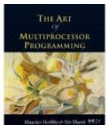
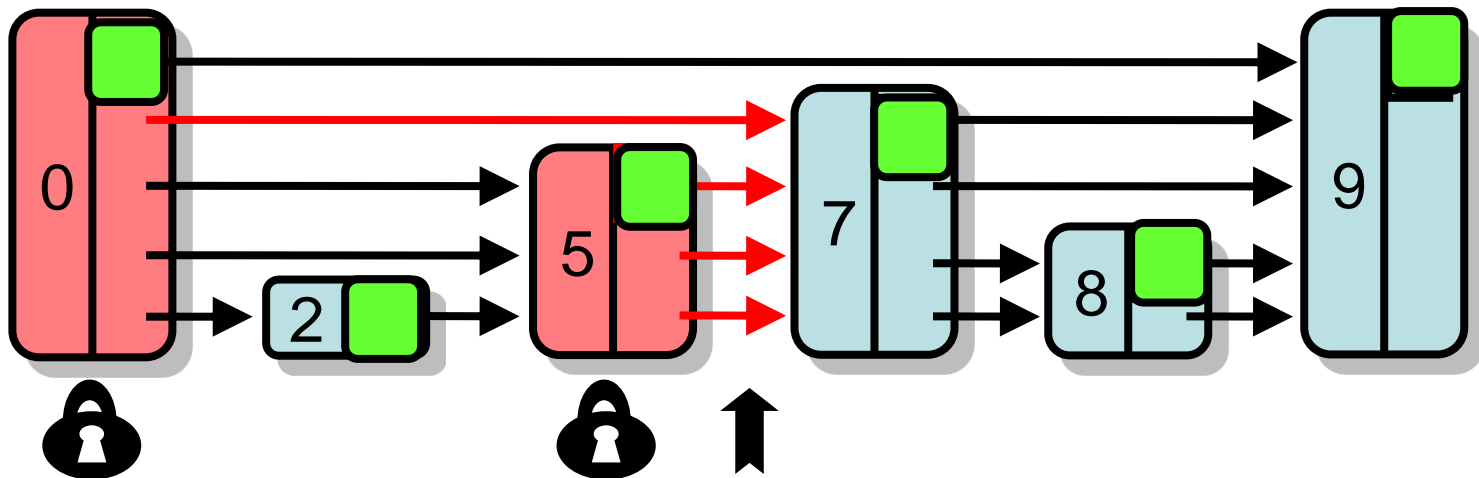
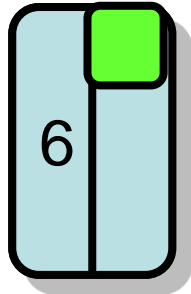
# add(6)

- **find()** predecessors



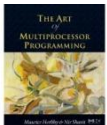
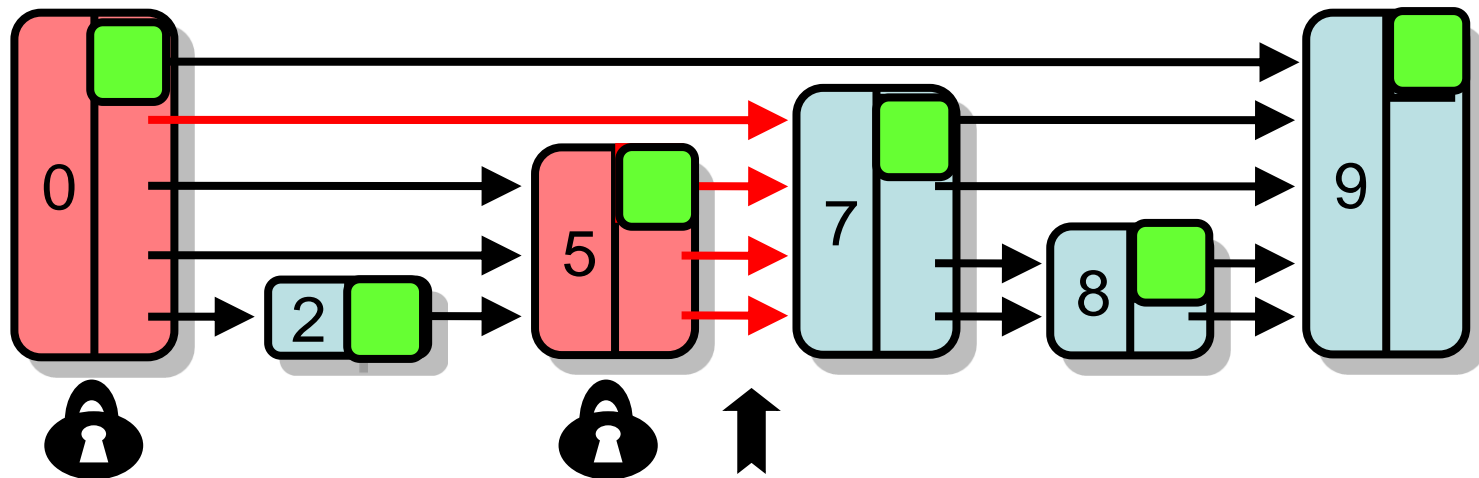
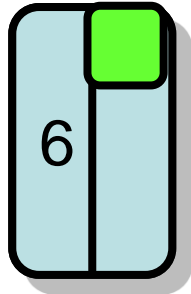
# add(6)

- **find()** predecessors
- Lock them



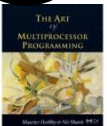
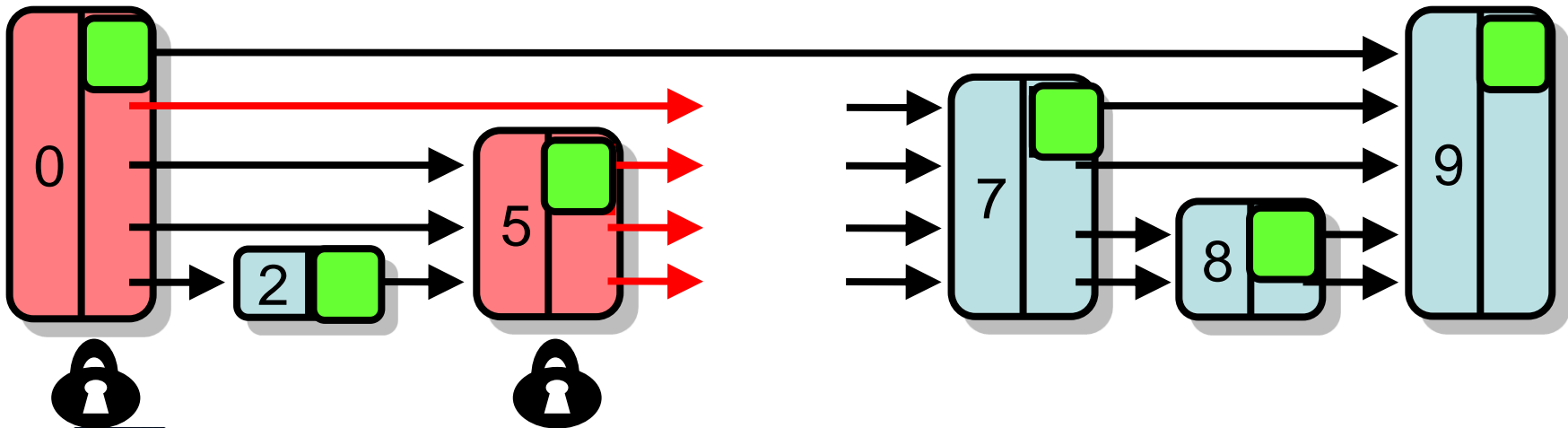
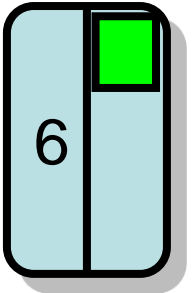
# add(6)

- **find()** predecessors
  - Lock them
  - Validate
- } **Optimistic approach**



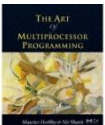
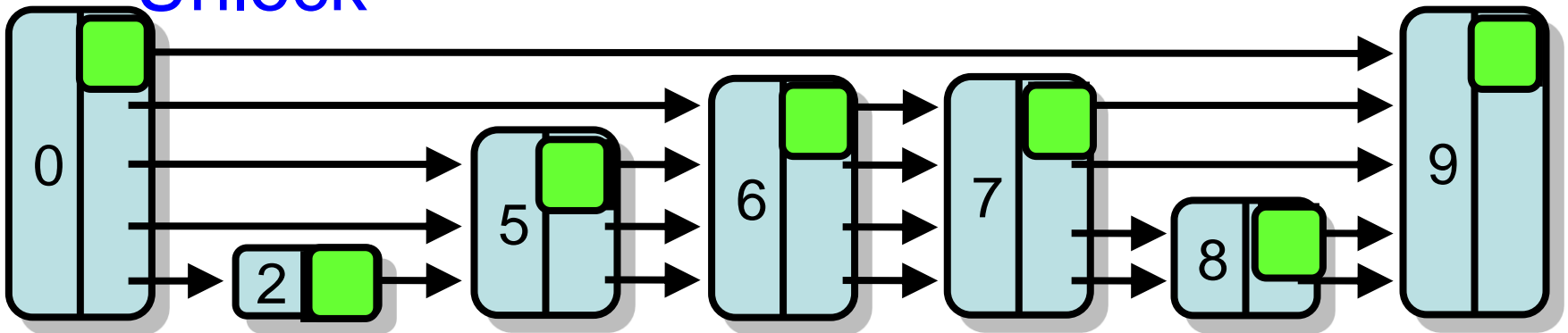
# add(6)

- **find()** predecessors
- Lock them
- Validate
- Splice



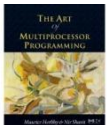
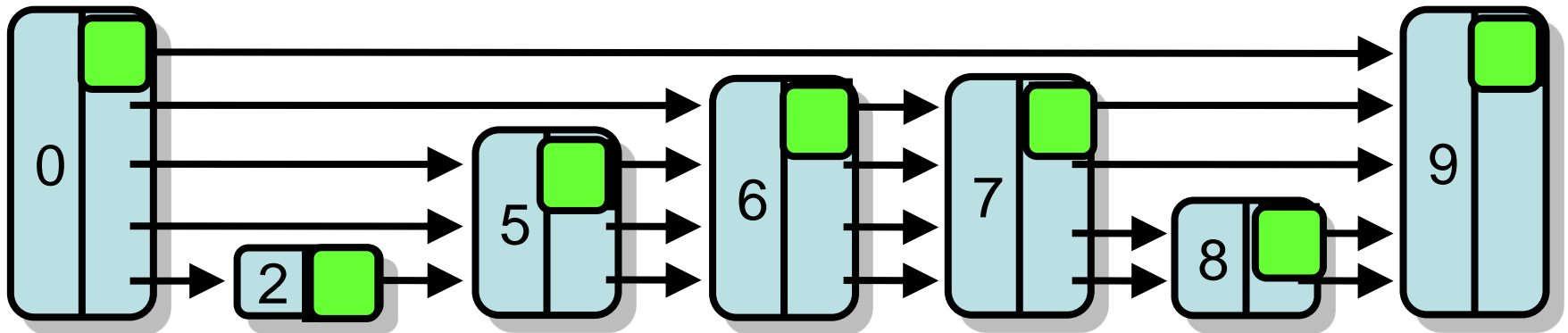
# add(6)

- **find()** predecessors
- Lock them
- Validate
- Splice
- Unlock



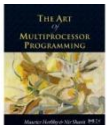
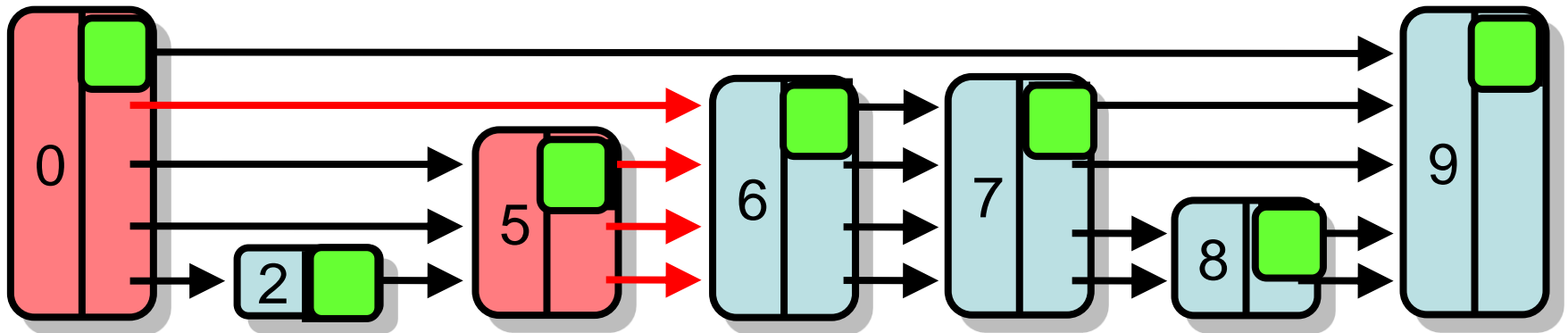


# remove(6)



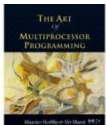
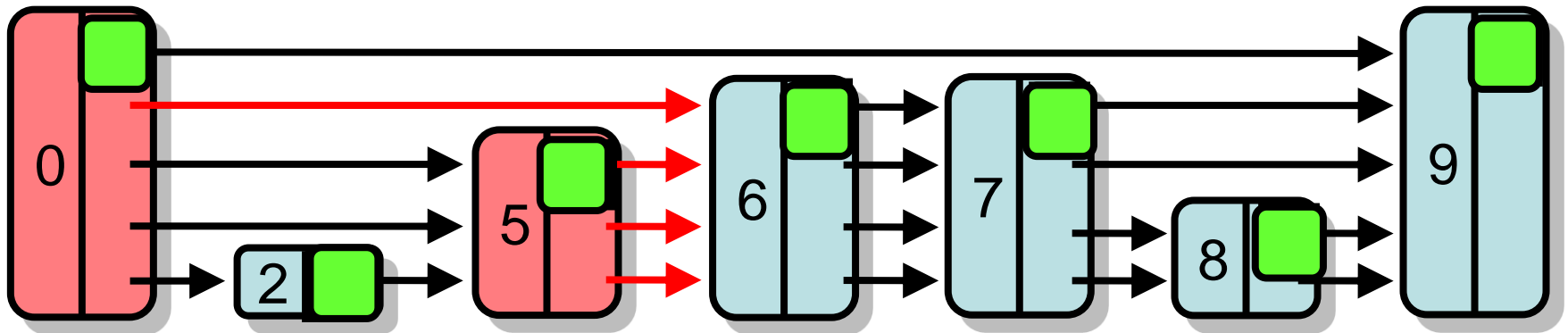
# remove(6)

- **find()** predecessors



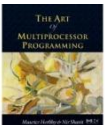
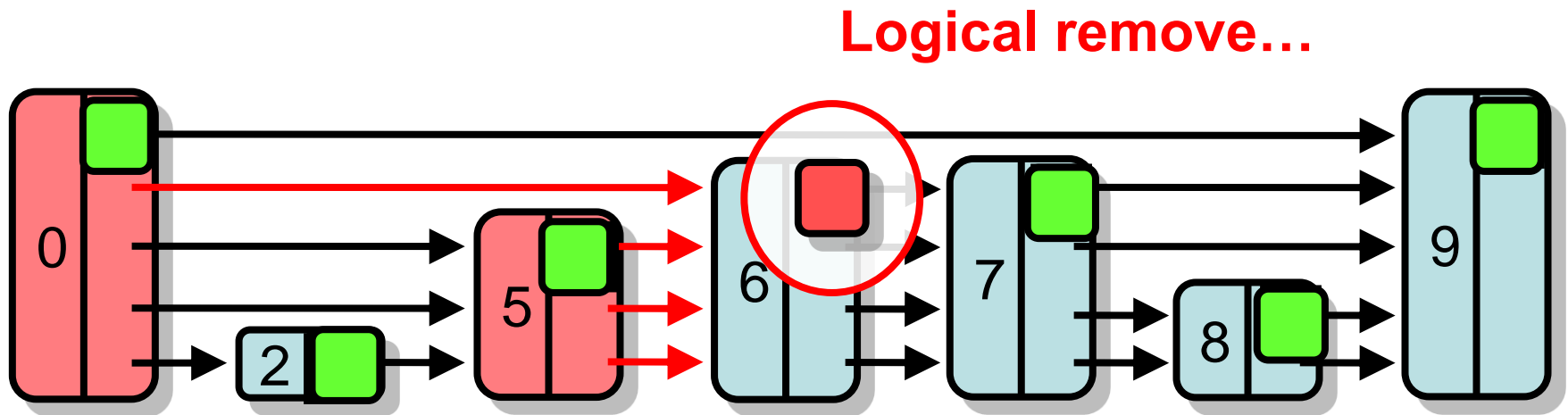
# remove(6)

- **find()** predecessors
- Lock victim



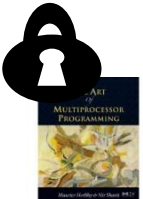
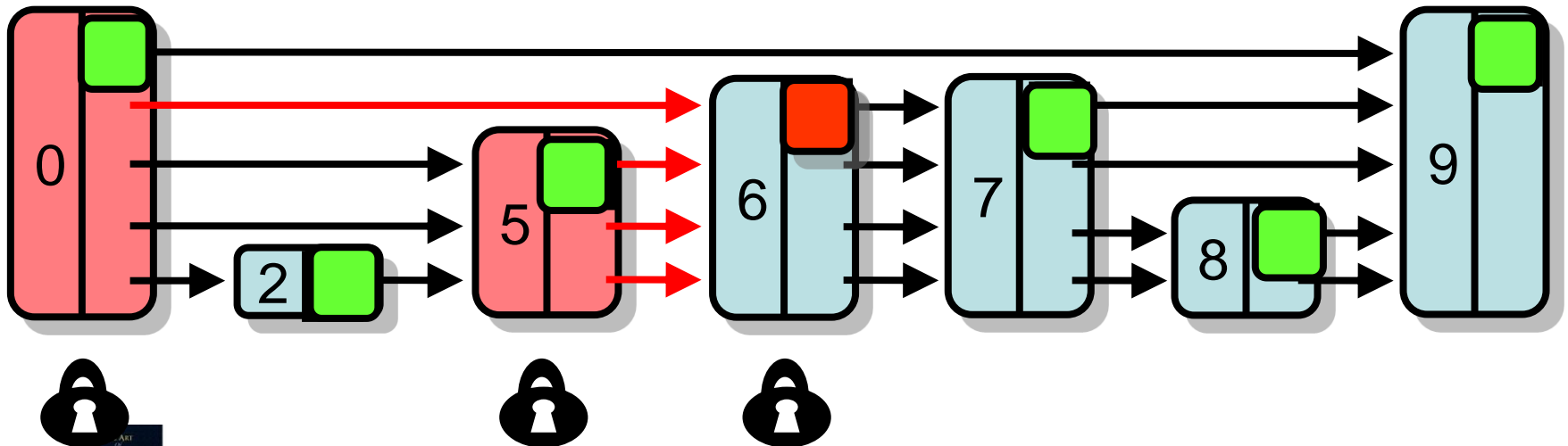
# remove(6)

- **find()** predecessors
- Lock victim
- Set mark (if not already set)



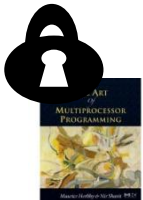
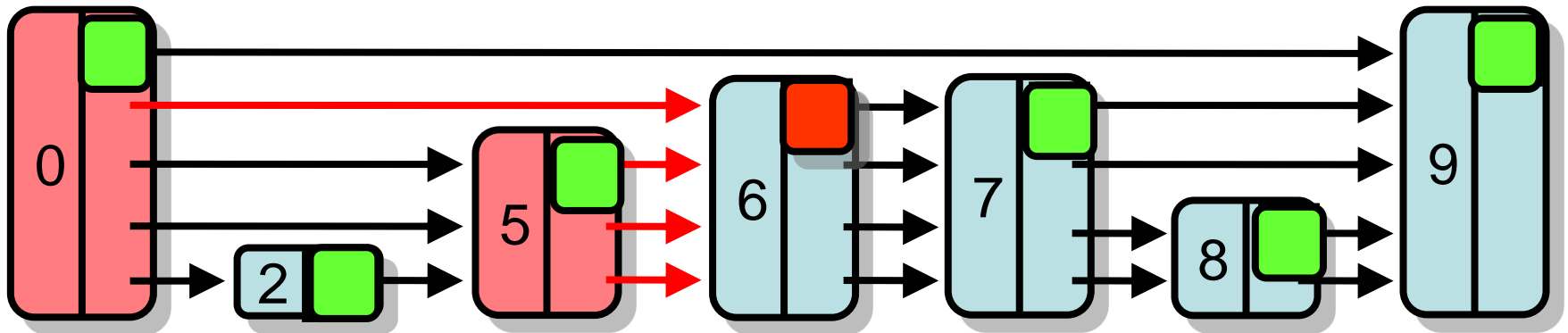
# remove(6)

- **find()** predecessors
- Lock victim
- Set mark (if not already set)
- Lock predecessors (ascending order) & validate



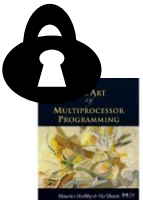
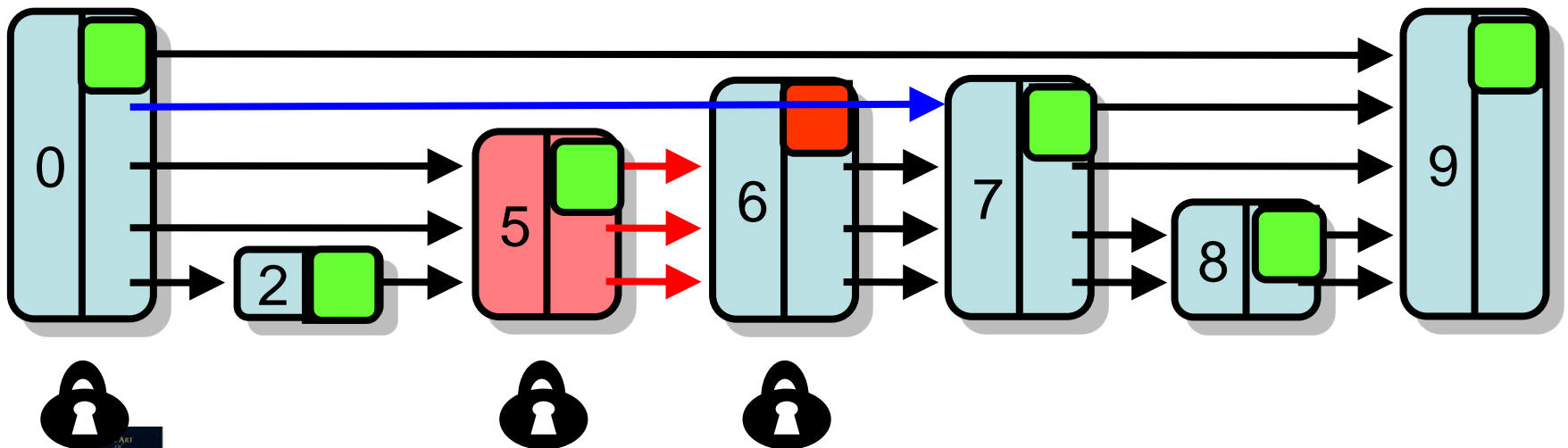
# remove(6)

- **find()** predecessors
- Lock victim
- Set mark (if not already set)
- Lock predecessors (ascending order) & validate
- Physically remove



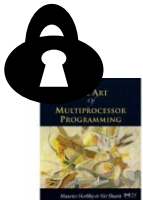
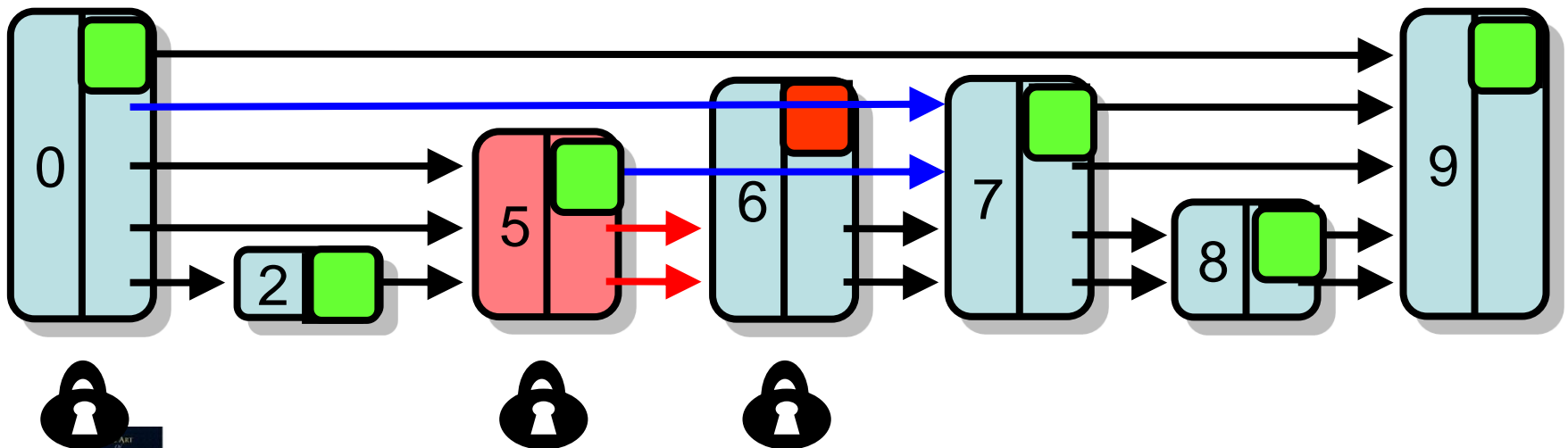
# remove(6)

- **find()** predecessors
- Lock victim
- Set mark (if not already set)
- Lock predecessors (ascending order) & validate
- Physically remove



# remove(6)

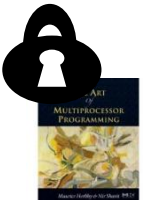
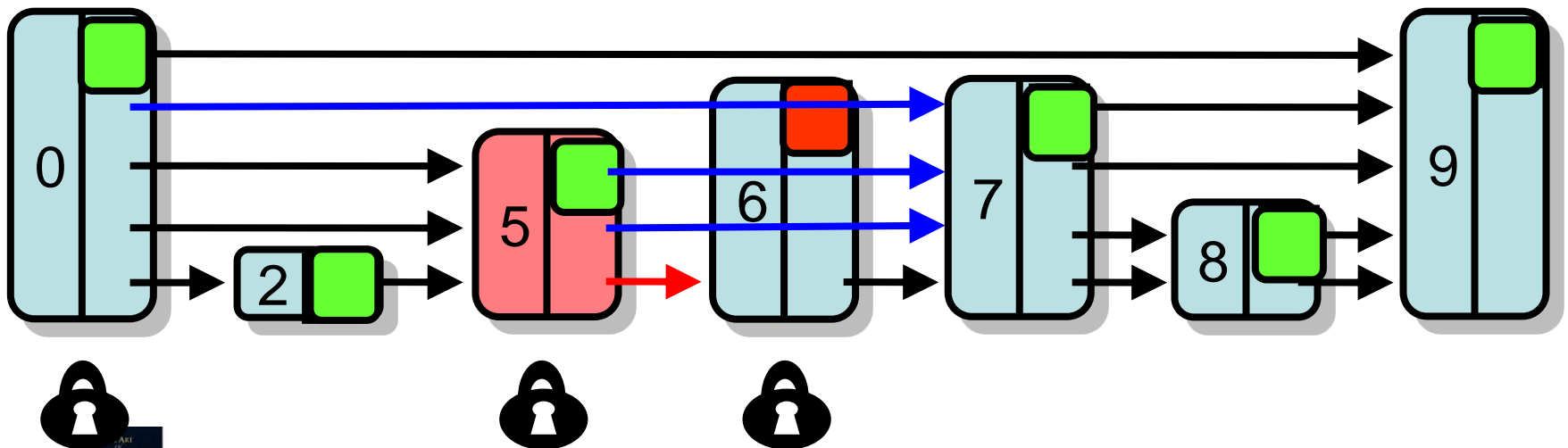
- **find()** predecessors
- Lock victim
- Set mark (if not already set)
- Lock predecessors (ascending order) & validate
- Physically remove





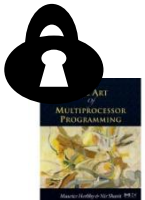
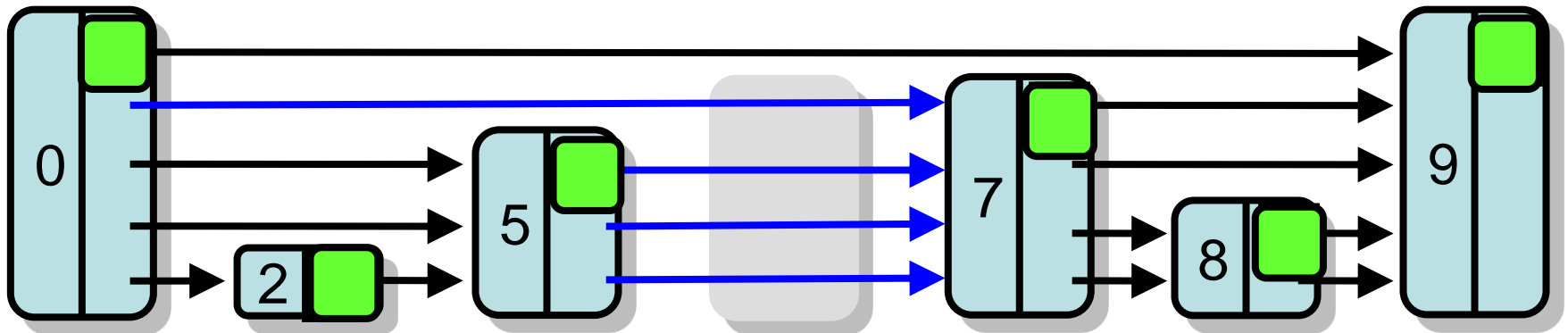
# remove(6)

- **find()** predecessors
- Lock victim
- Set mark (if not already set)
- Lock predecessors (ascending order) & validate
- Physically remove



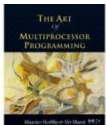
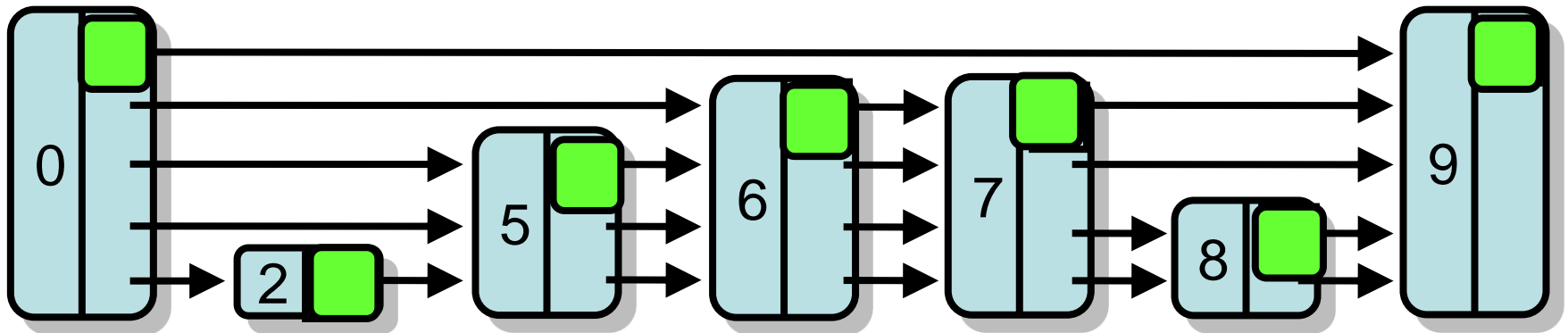
# remove(6)

- **find()** predecessors
- Lock victim
- Set mark (if not already set)
- Lock predecessors (ascending order) & validate
- Physically remove



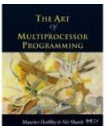
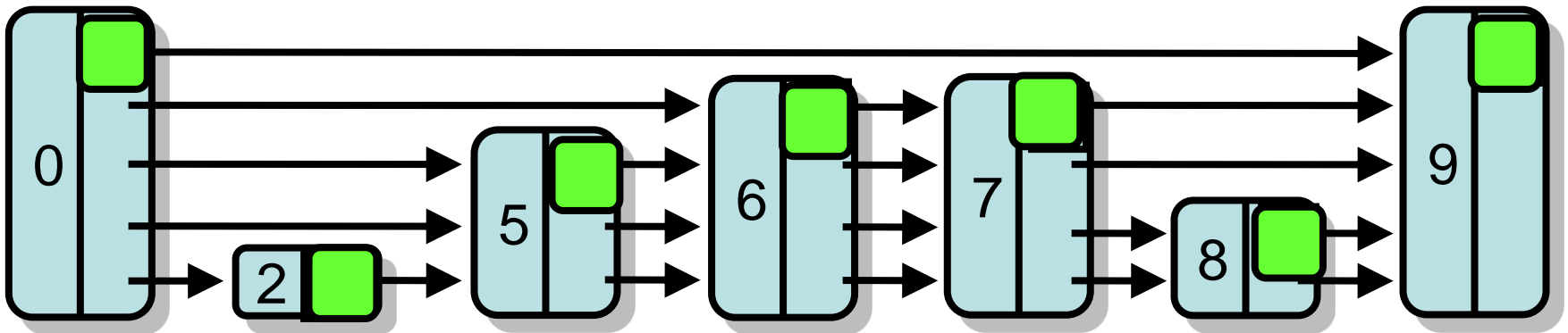
# contains(8)

- **find()** & not marked



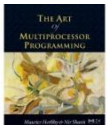
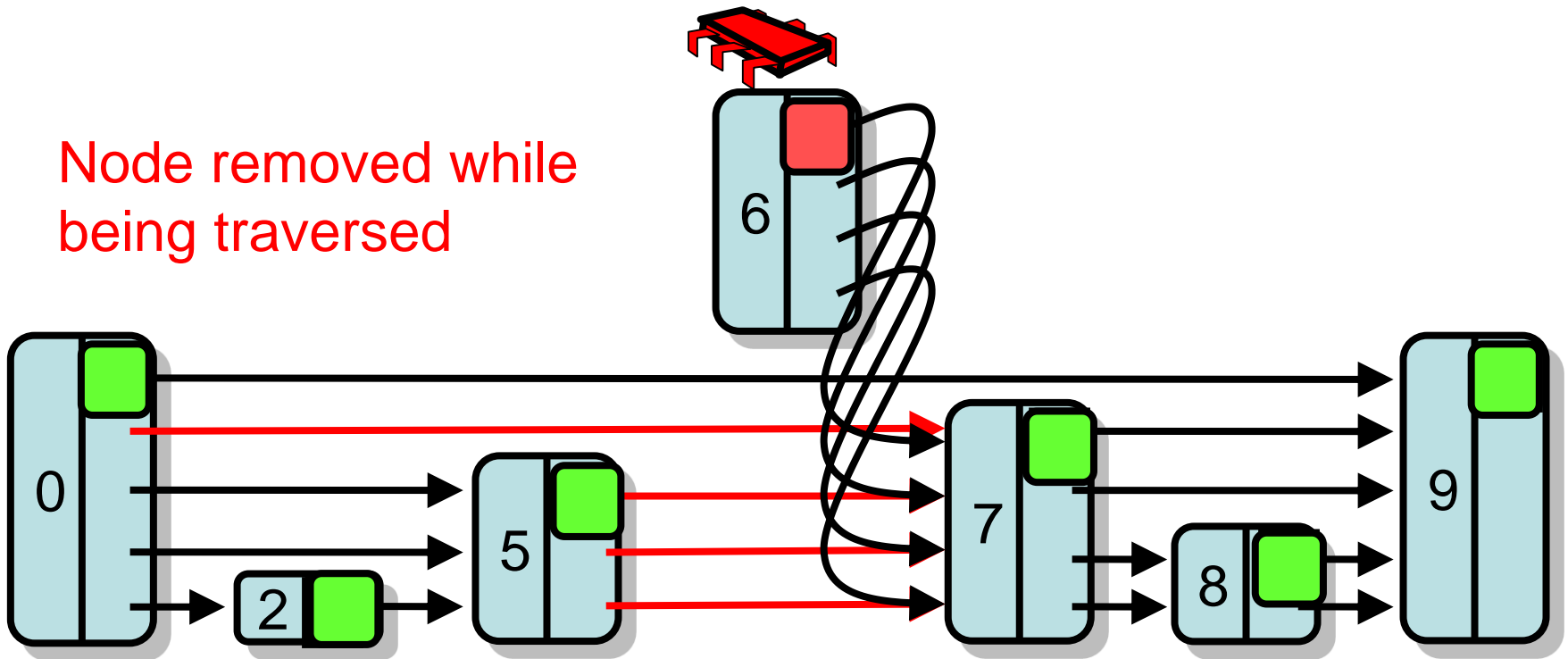
# contains(8)

Node 6 removed while traversed



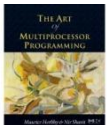
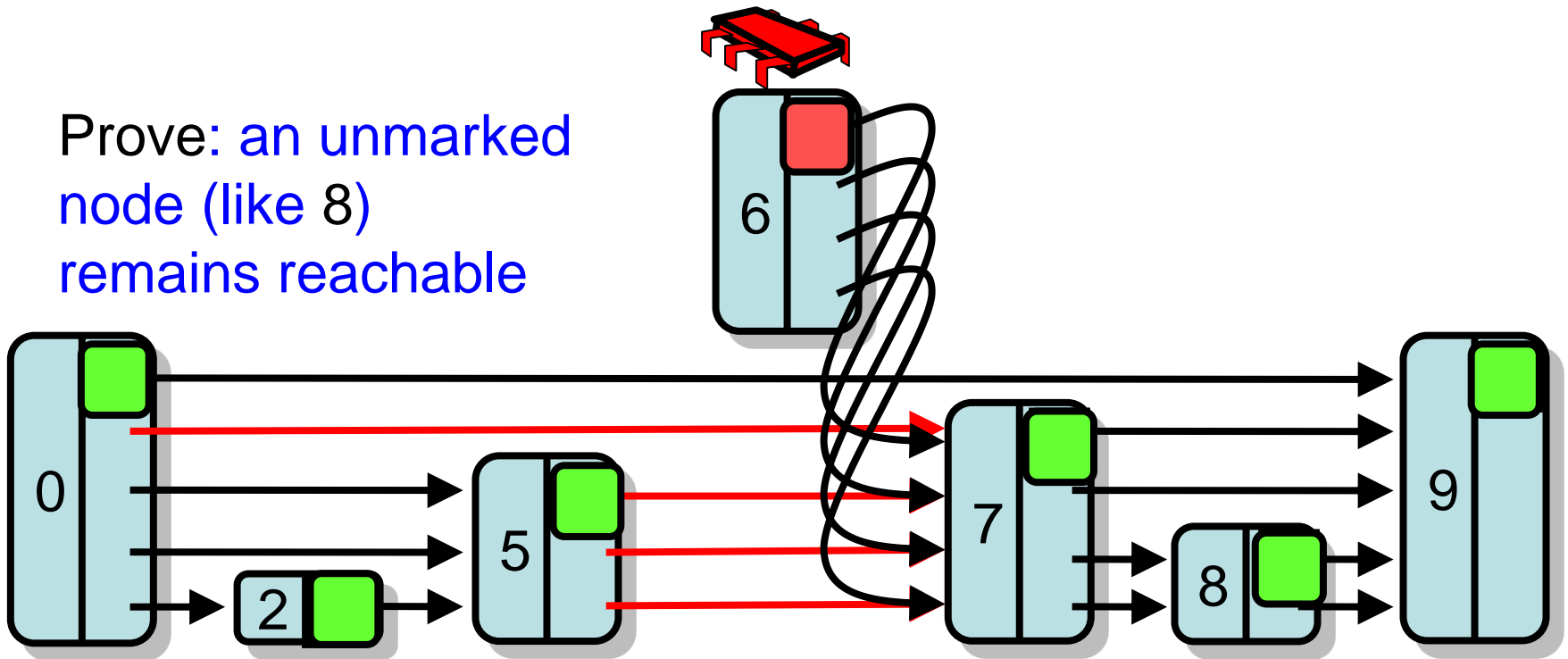
# contains(8)

Node removed while  
being traversed



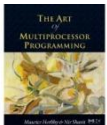
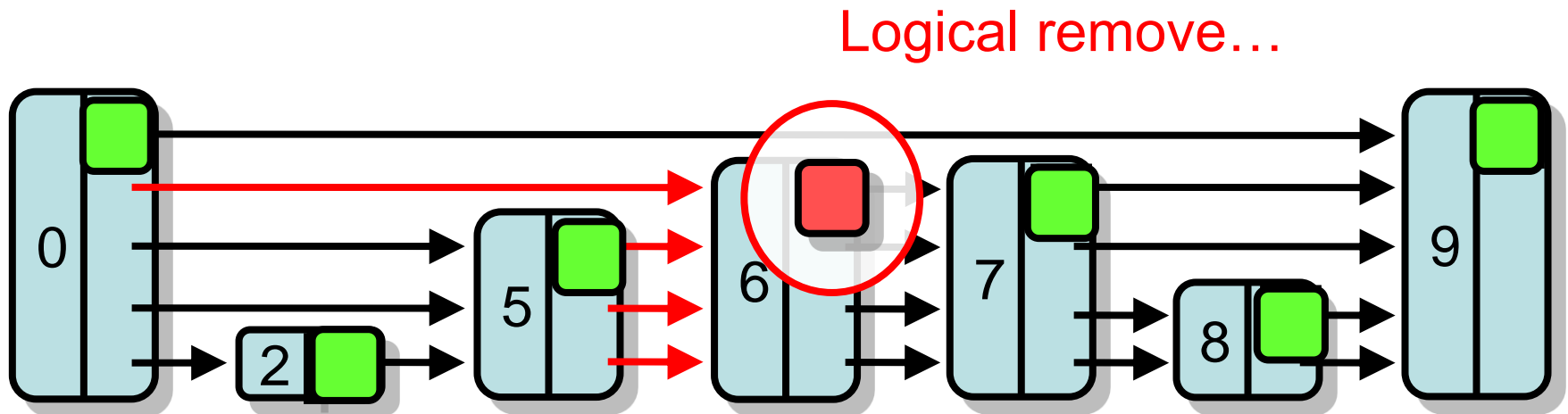
# contains(8)

Prove: an unmarked node (like 8) remains reachable



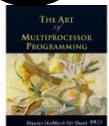
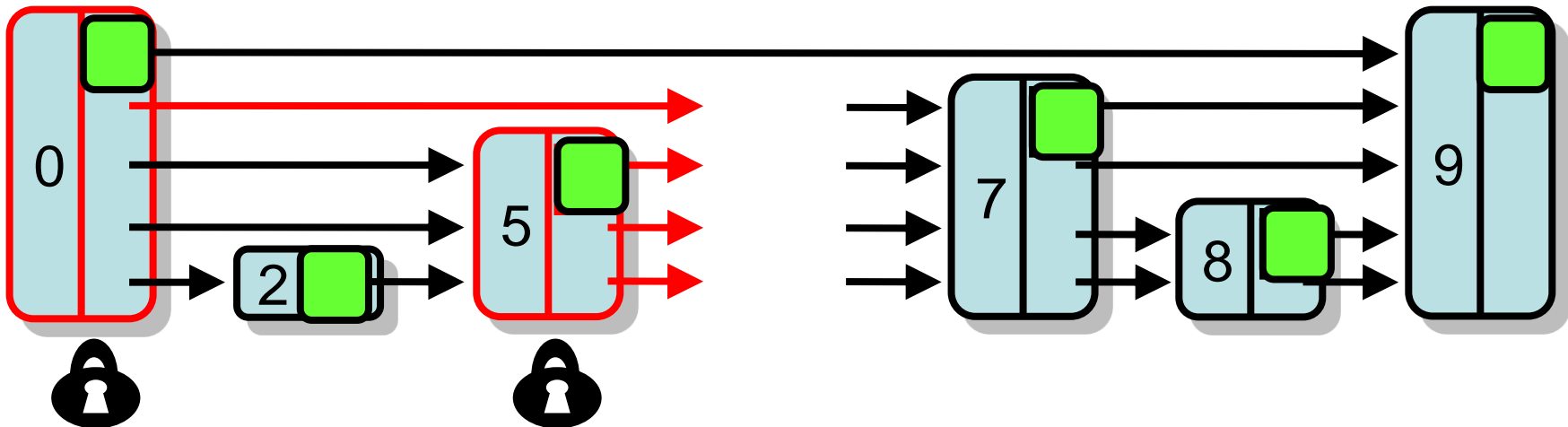
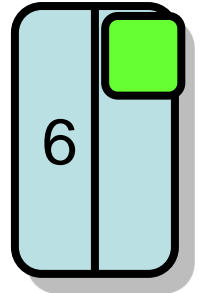
# remove(6): Linearization

- Successful remove happens when bit is set



# Add: Linearization

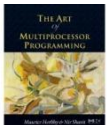
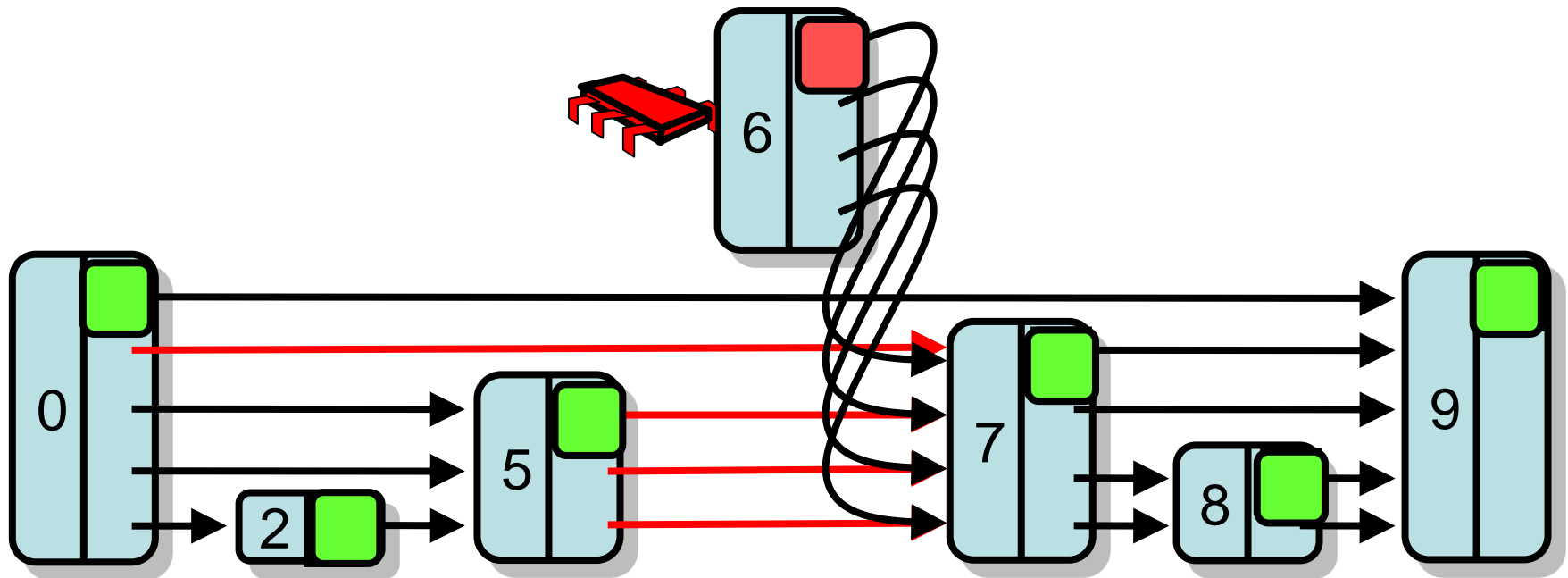
- Successful `add()` at point when fully linked
- Add `fullyLinked` bit to indicate this
- Bit tested by `contains()`





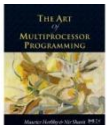
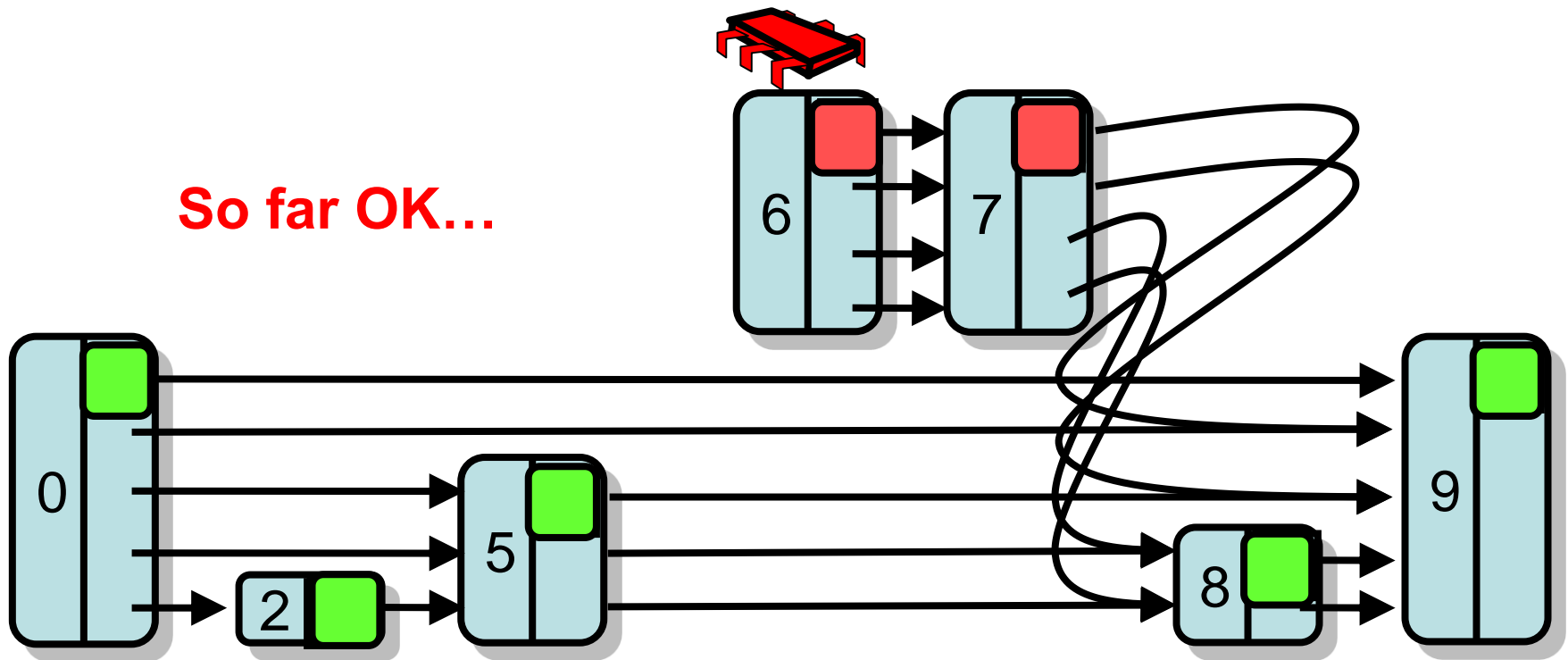
# contains(7): Linearization

- When fully-linked unmarked node found
- Pause while fullyLinked bit unset



# contains(7): Linearization

- When do we linearize unsuccessful Search?

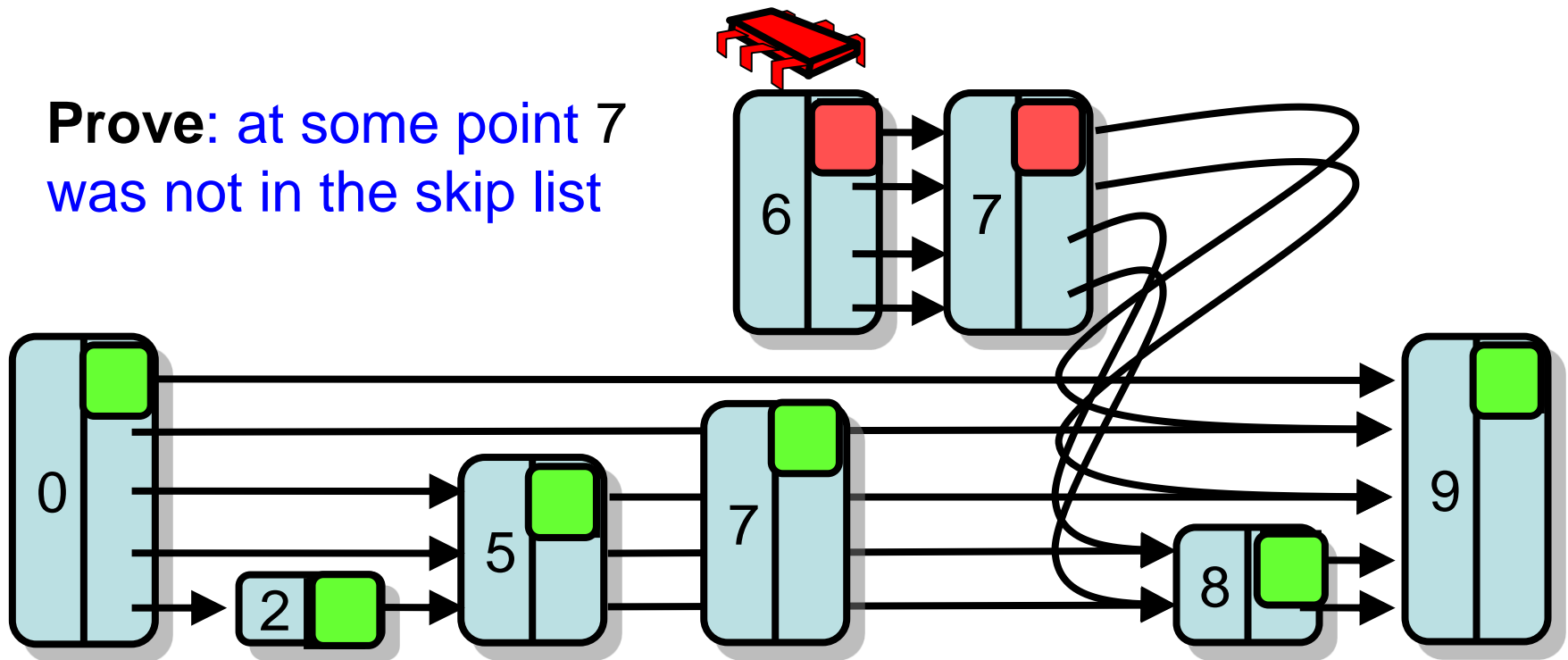




# contains(7): Linearization

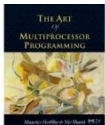
- When do we linearize unsuccessful Search?

**Prove:** at some point 7 was not in the skip list

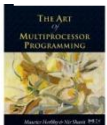
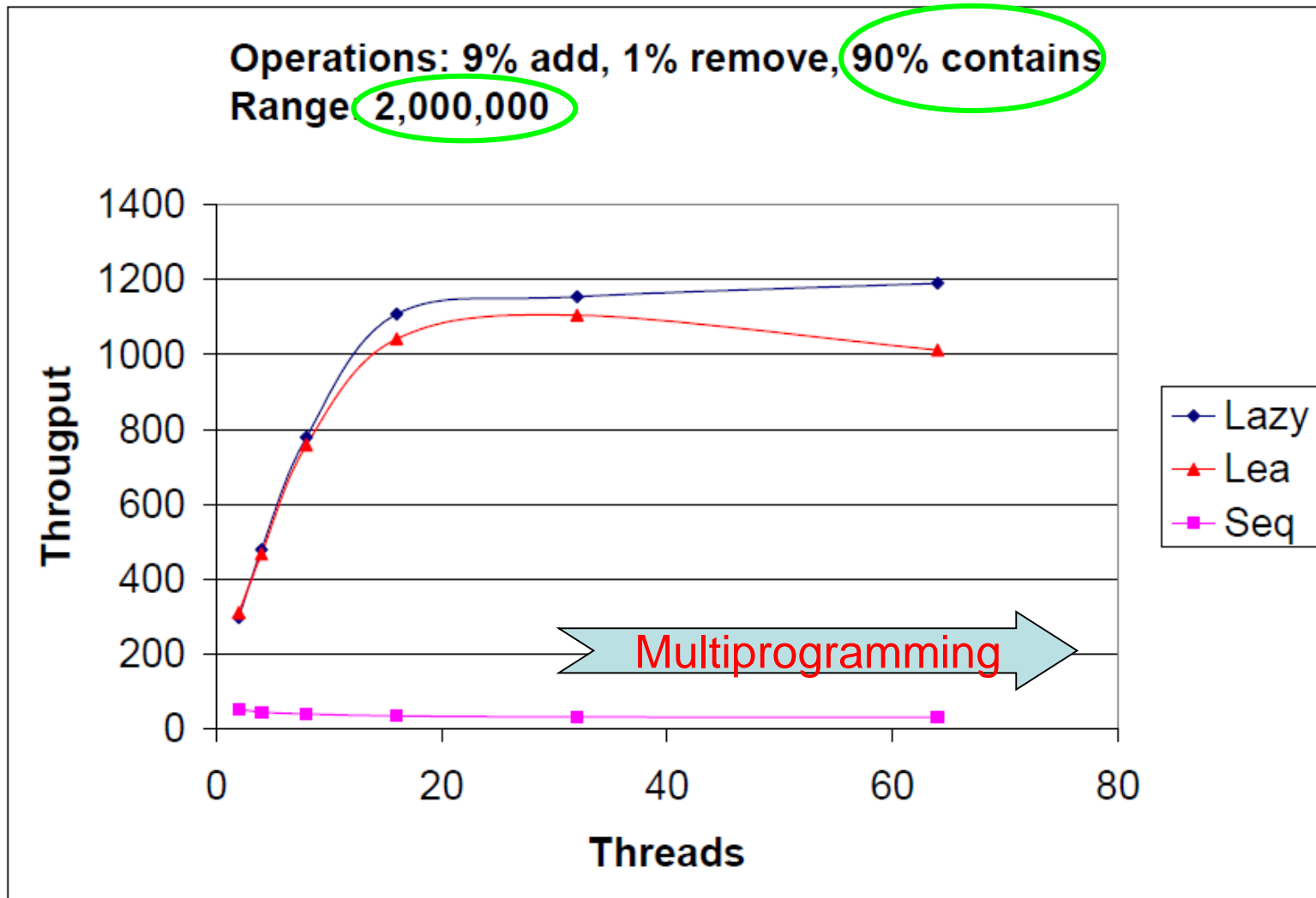


# A Simple Experiment

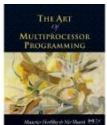
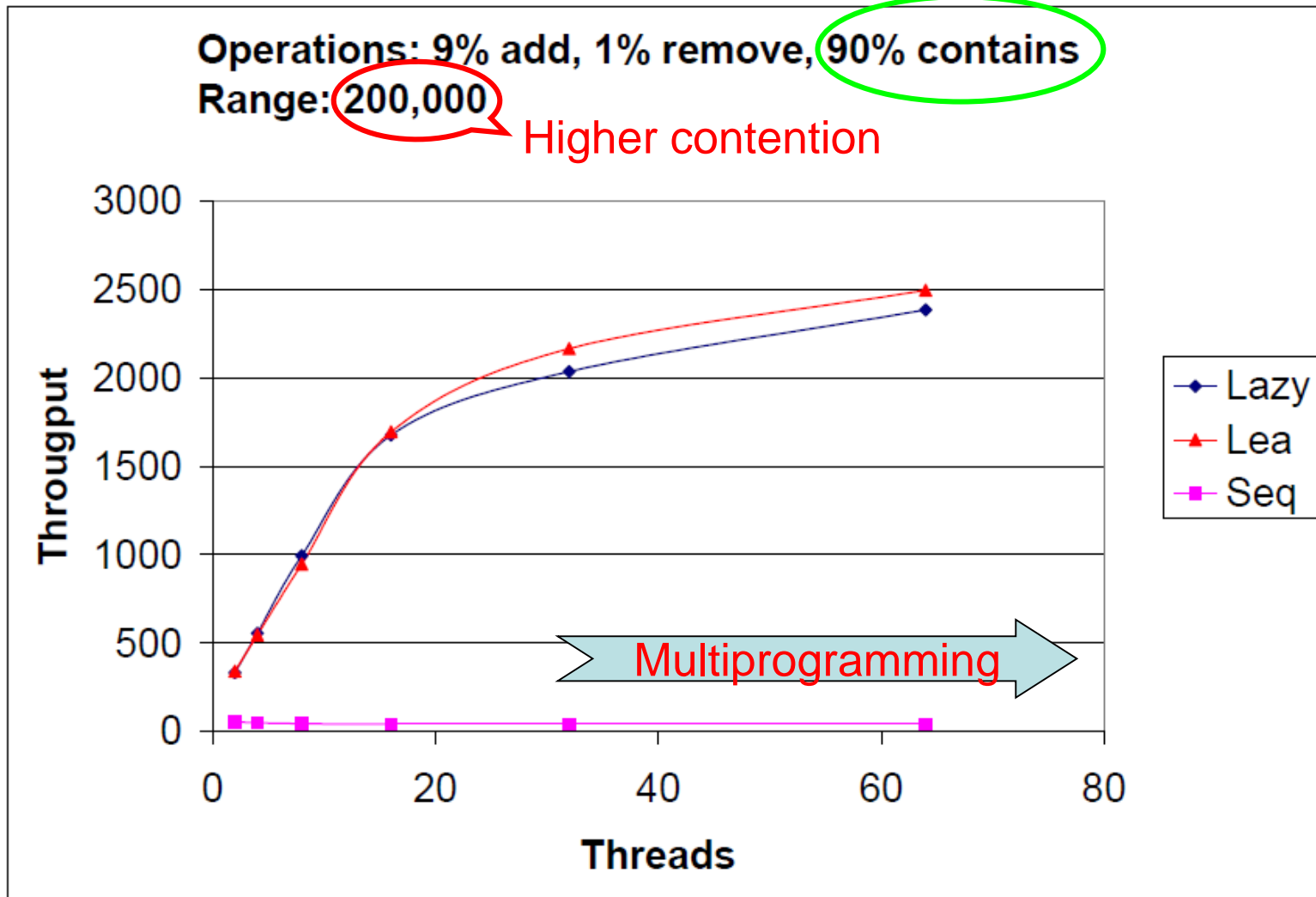
- Each thread runs 1 million iterations, each either:
  - **add()**
  - **remove()**
  - **contains()**
- Item and method chosen in random from some distribution



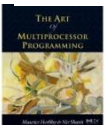
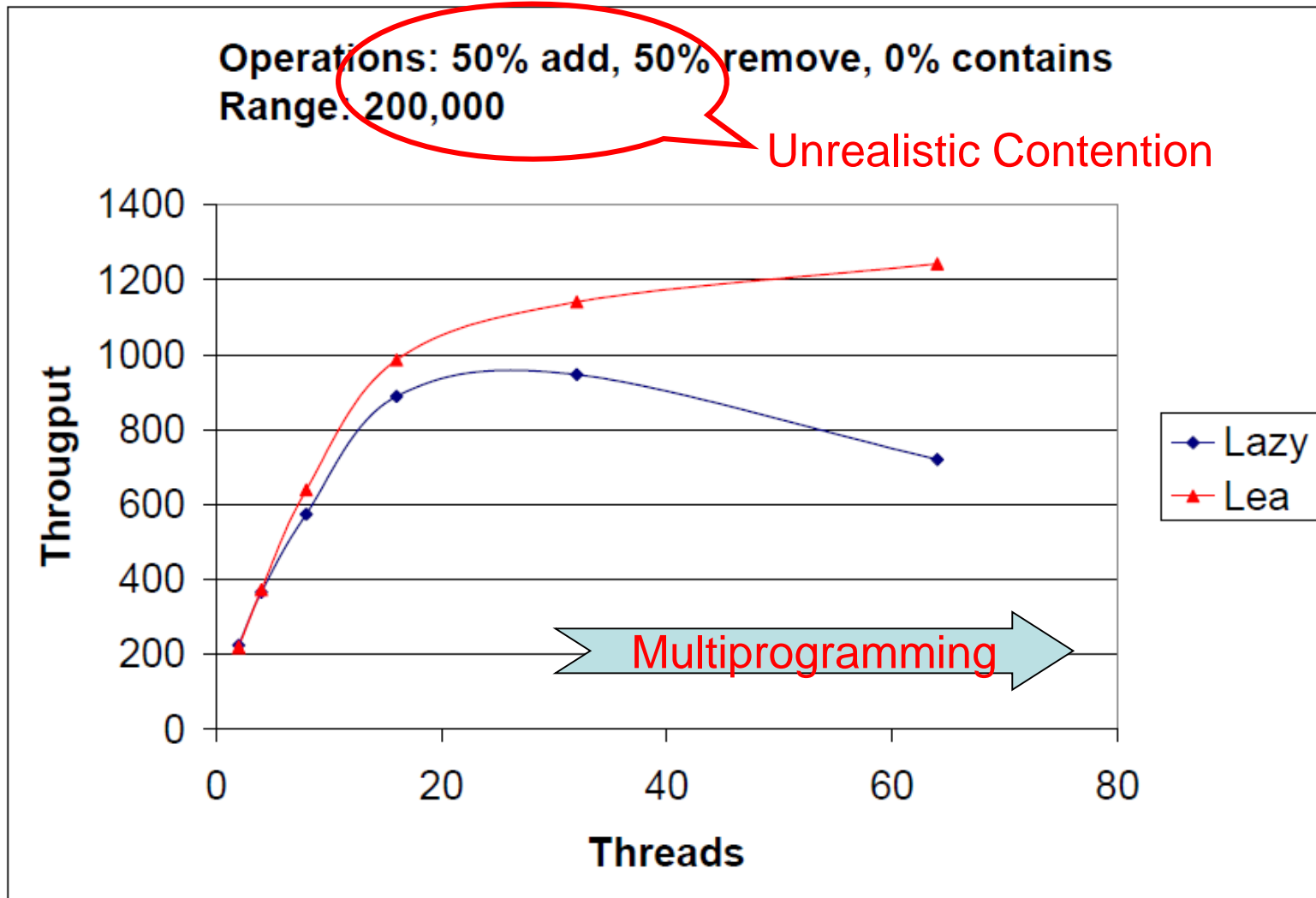
# Lazy Skip List: Performance



# Lazy Skip List: Performance



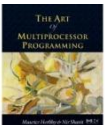
# Lazy Skip List: Performance





# Summary

- Lazy Skip List
  - Optimistic fine-grained Locking
- Performs as well as the lock-free solution in “common” cases
- Simple



This work is licensed under a [Creative Commons Attribution-ShareAlike 2.5 License](https://creativecommons.org/licenses/by-sa/3.0/).

- **You are free:**
  - **to Share** — to copy, distribute and transmit the work
  - **to Remix** — to adapt the work
- **Under the following conditions:**
  - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - <http://creativecommons.org/licenses/by-sa/3.0/>.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

