Concurrent Skip Lists

Christof Fetzer, TU Dresden

Based on slides by Maurice Herlihy and Nir Shavit

Set Object Interface

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

Expectation

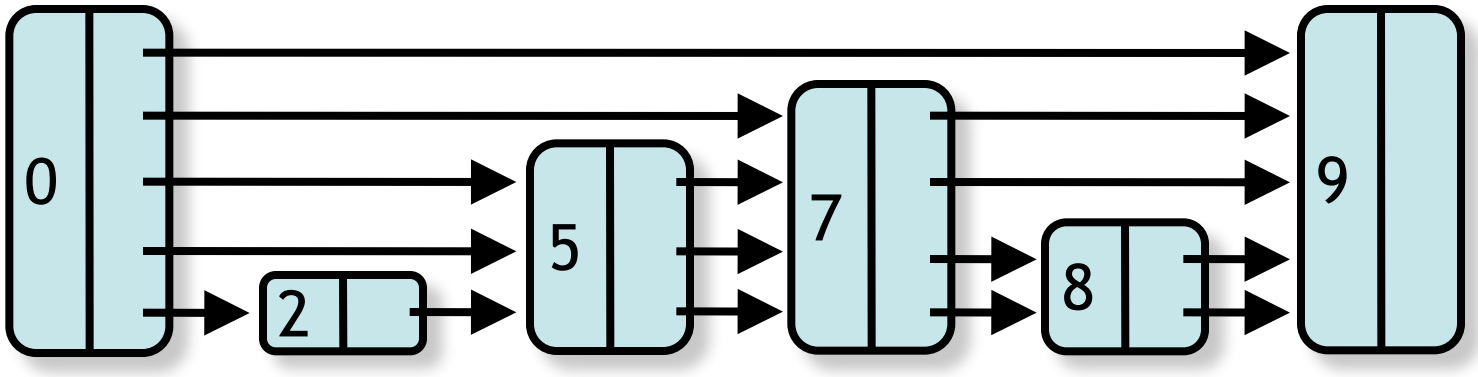
- Typically high % of contains() calls
- Many fewer add() calls
- And even fewer remove() calls
  - 90% contains()
  - 9% add()
  - 1% remove()
- Folklore?
  - Yes but probably mostly true

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

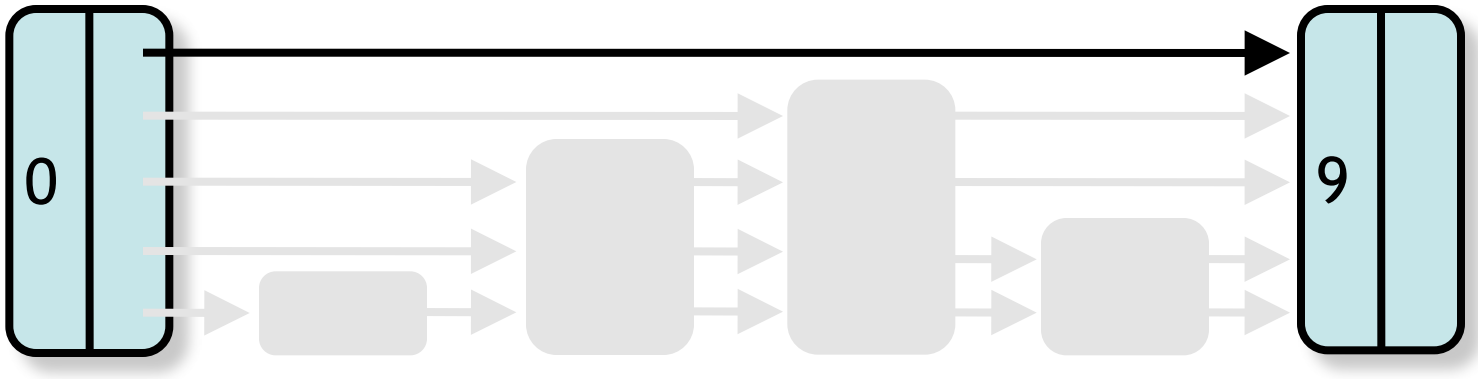
Skip Lists

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



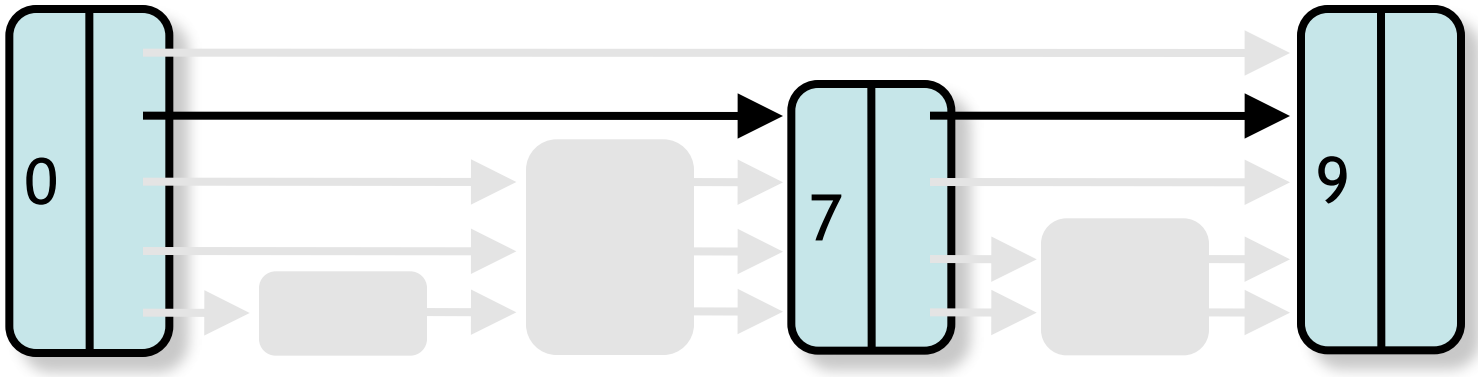
Skip List Property

- Each layer is sublist of lower-levels



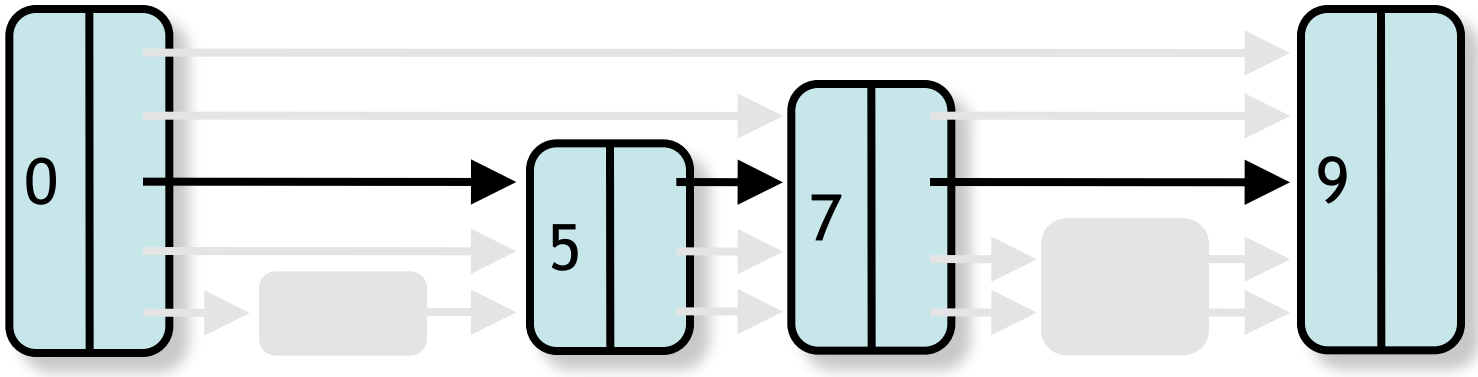
Skip List Property

- Each layer is sublist of lower-levels



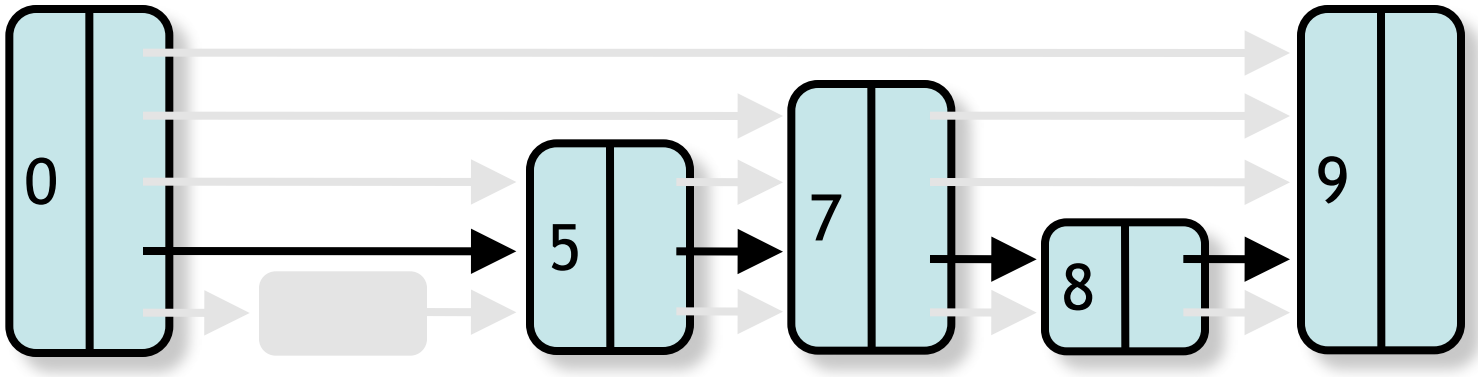
Skip List Property

- Each layer is sublist of lower-levels



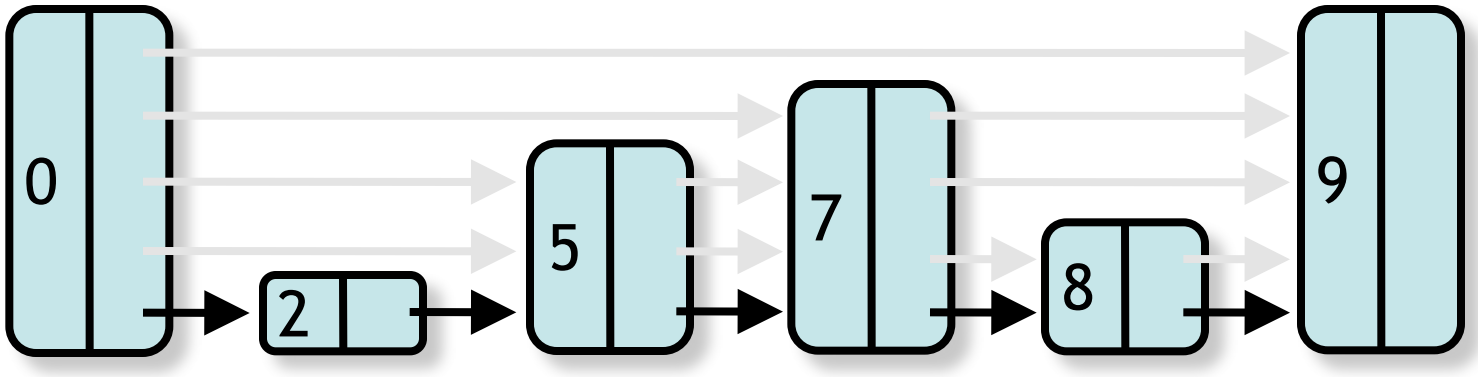
Skip List Property

- Each layer is sublist of lower-levels



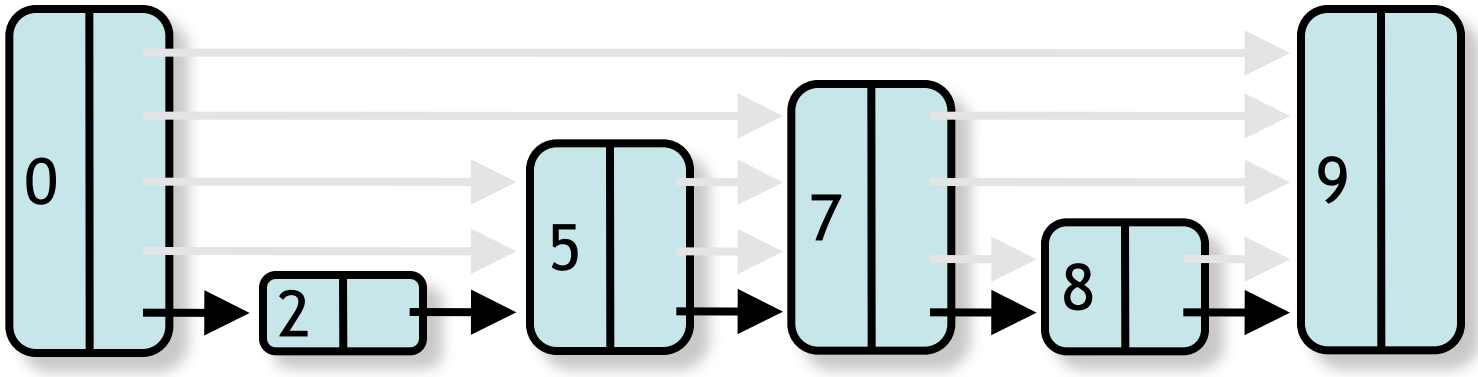
Skip List Property

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

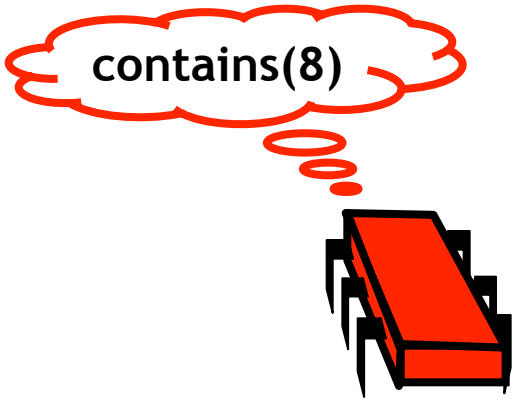


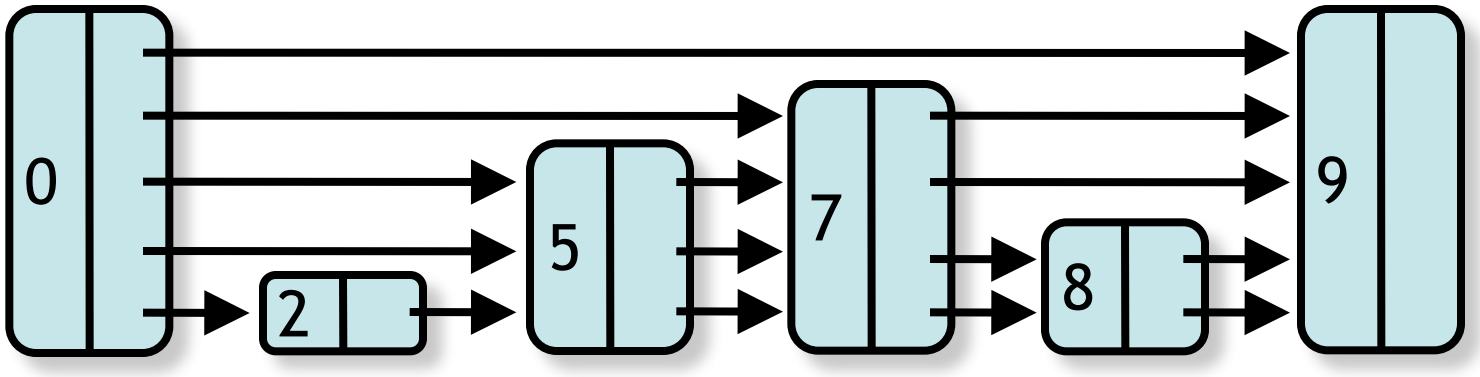
Skip List Property

- Each layer is sublist of lower-levels
- Not easy to preserve in concurrent implementations …

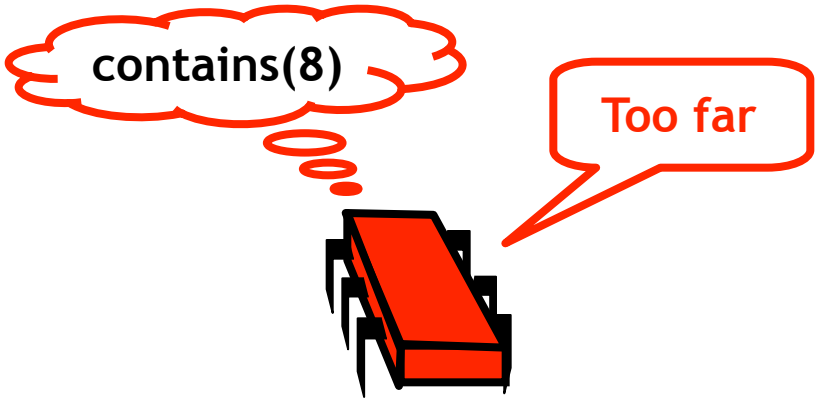


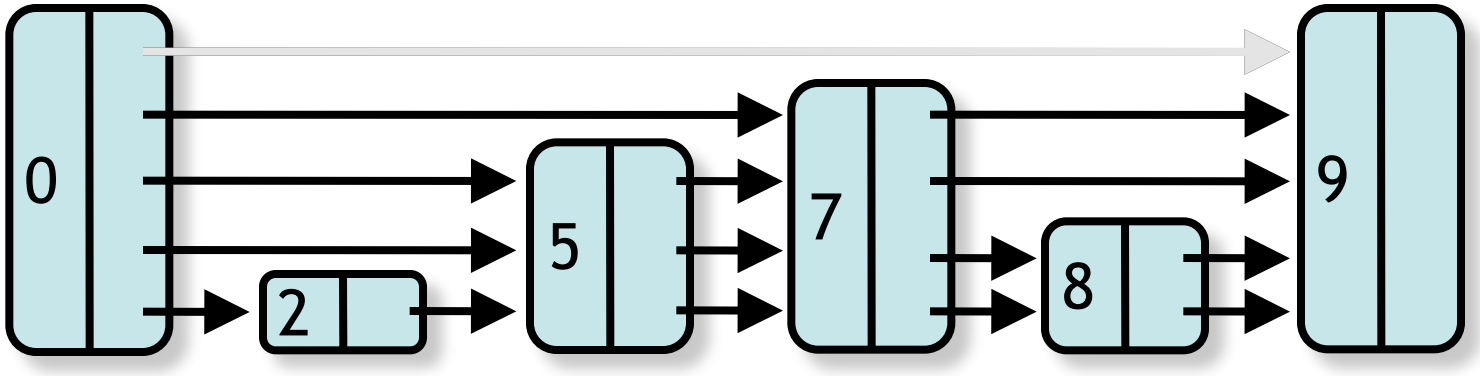
Search



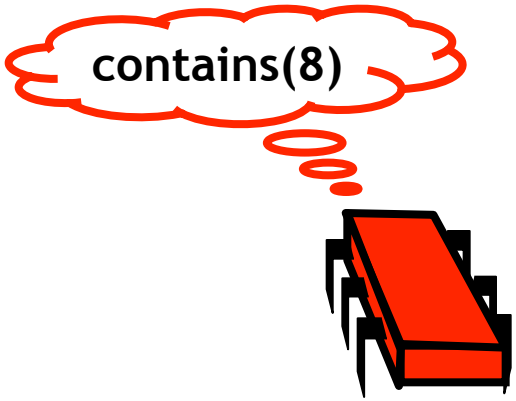


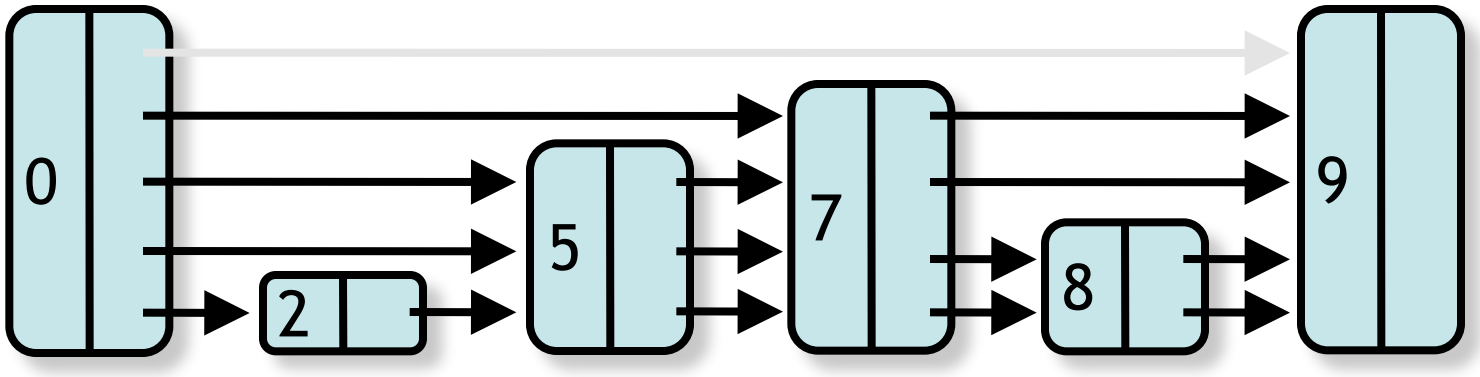
Search



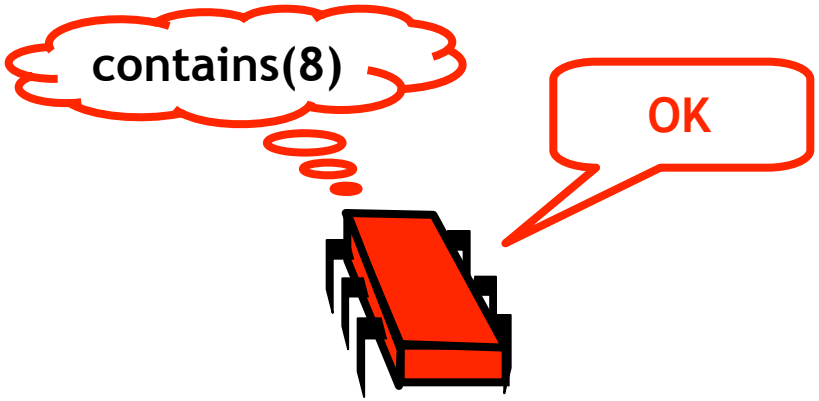


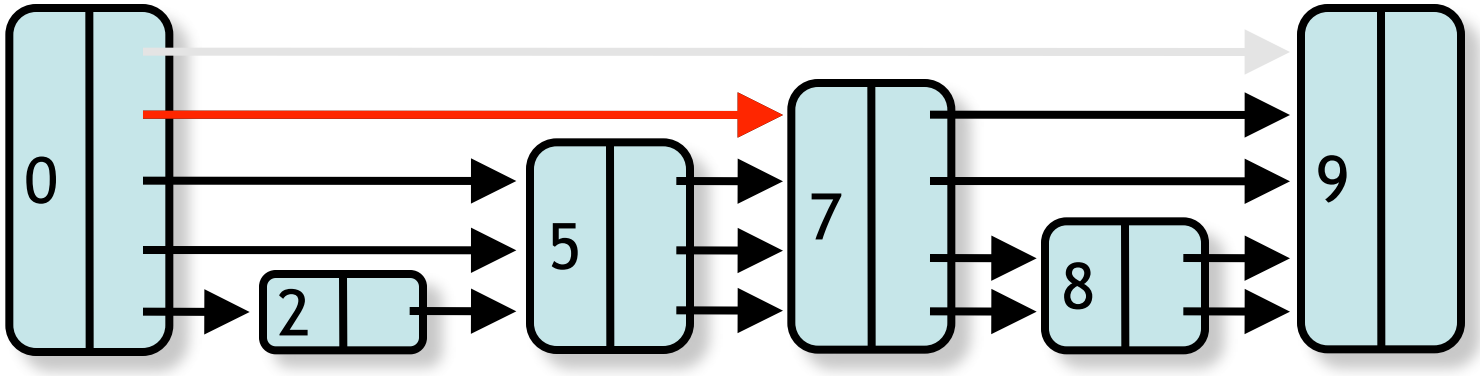
Search



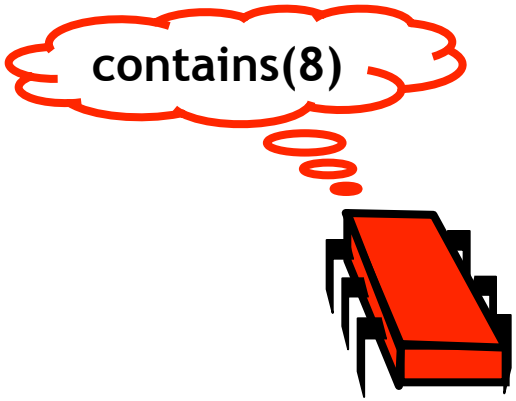


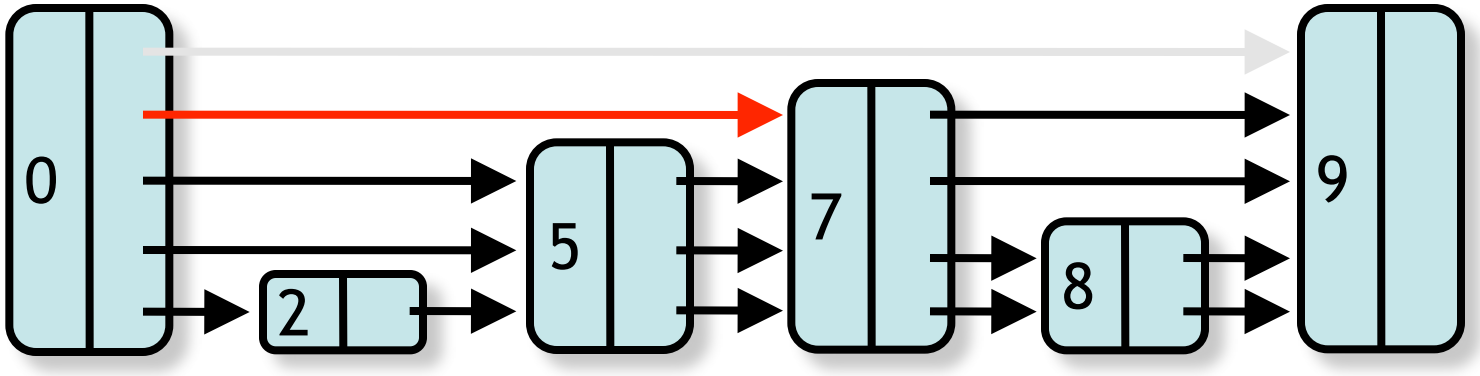
Search



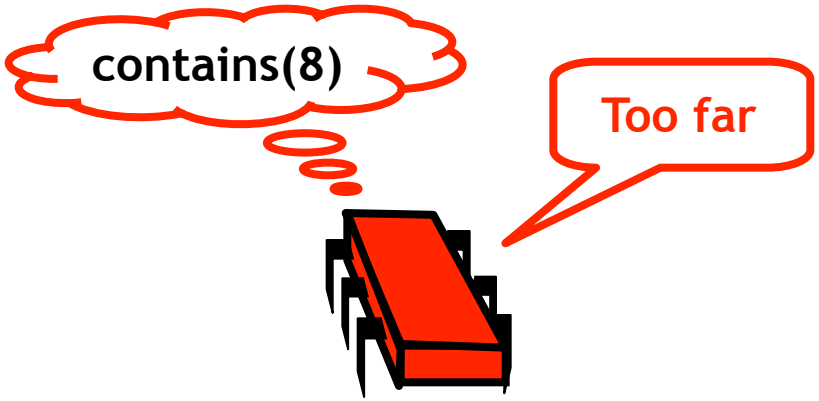


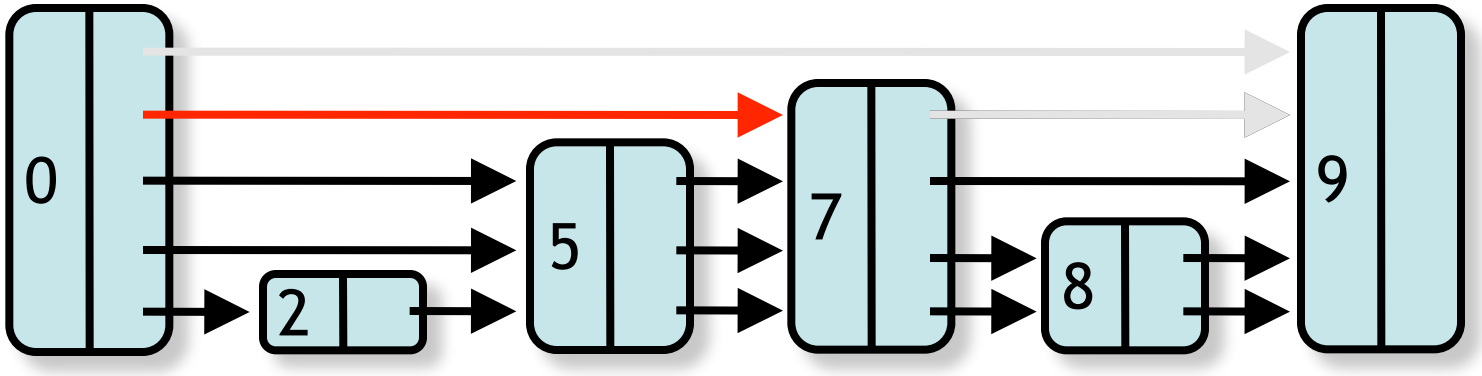
Search



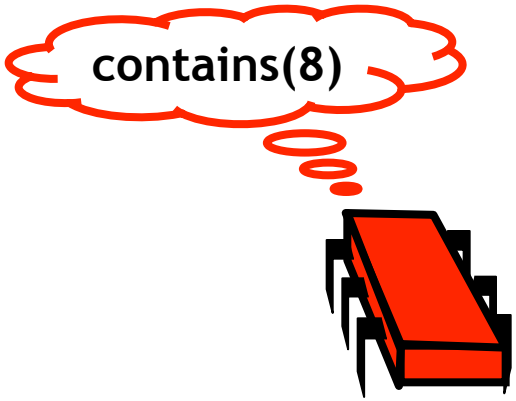


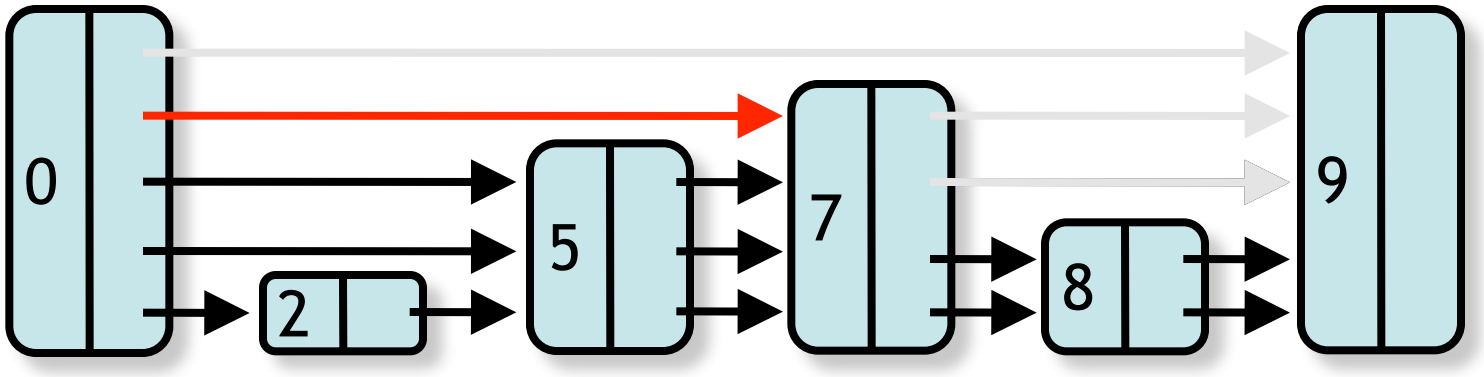
Search



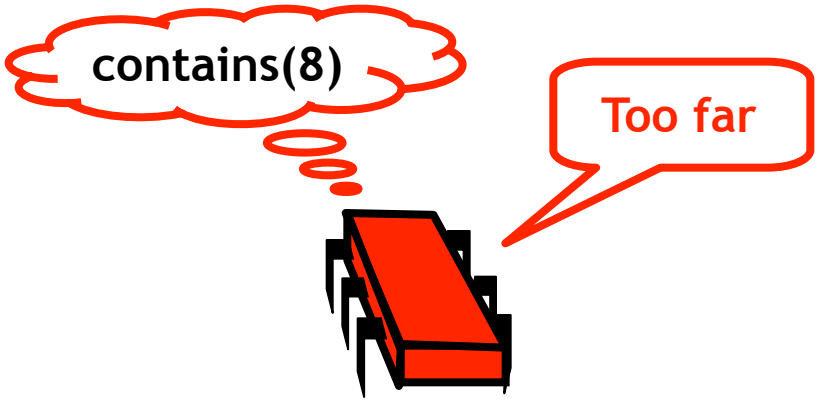


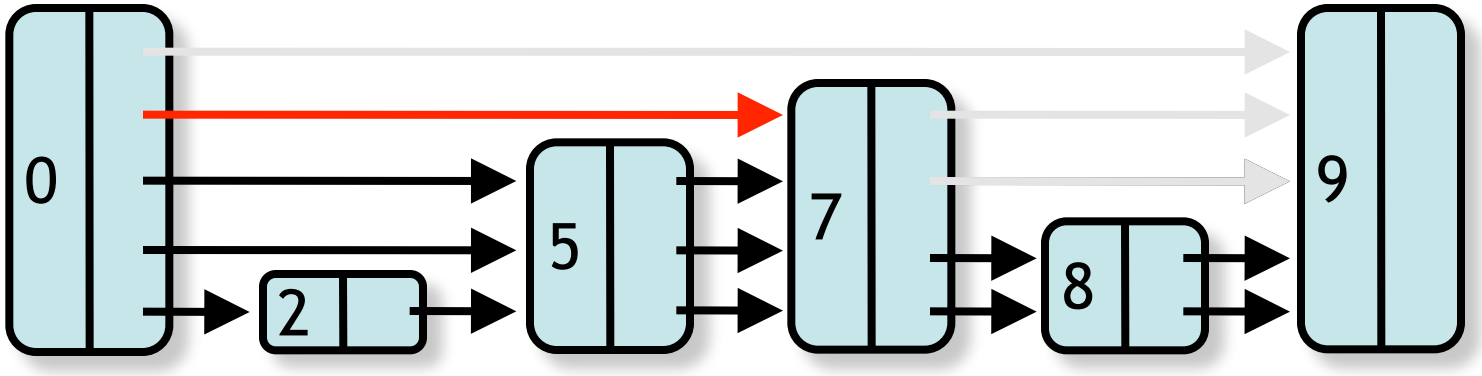
Search



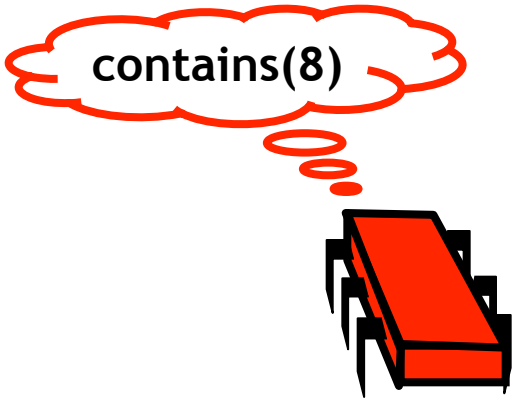


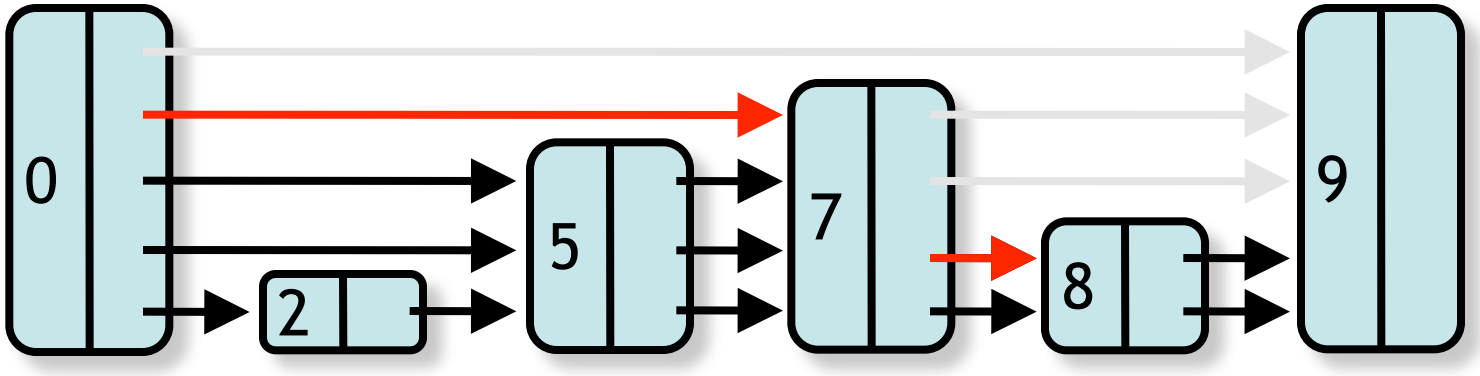
Search



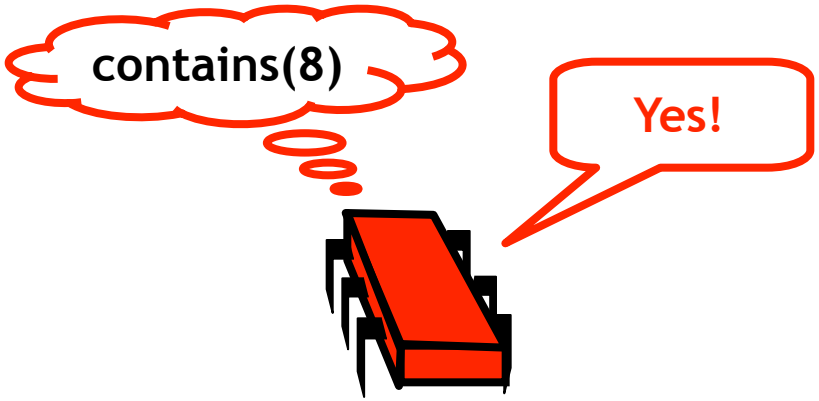


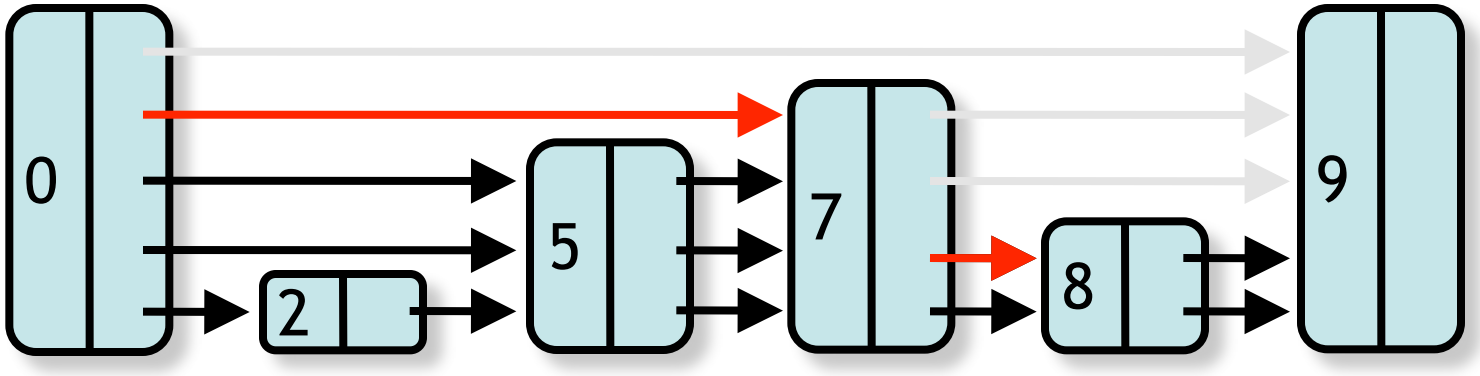
Search



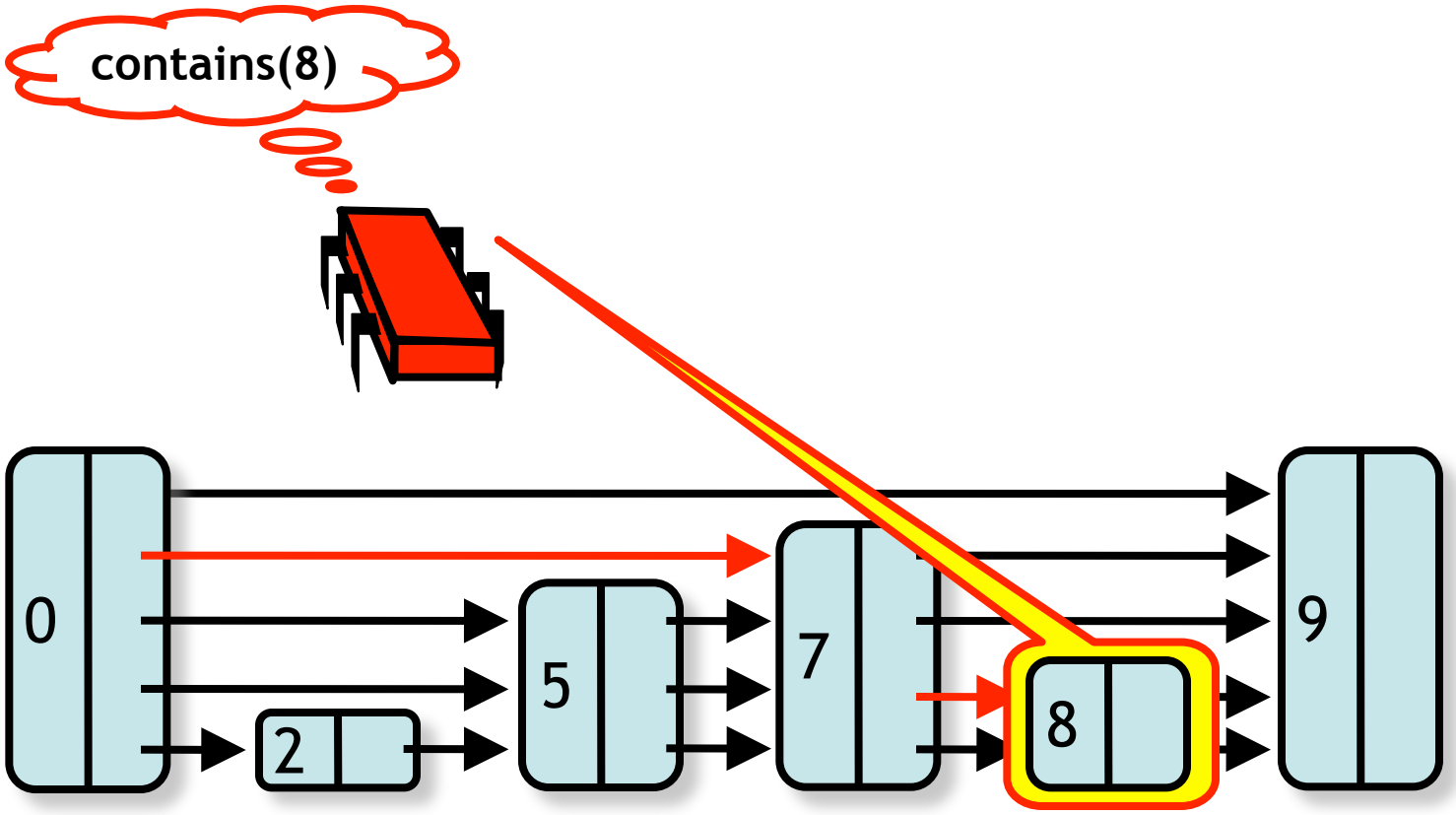


Search

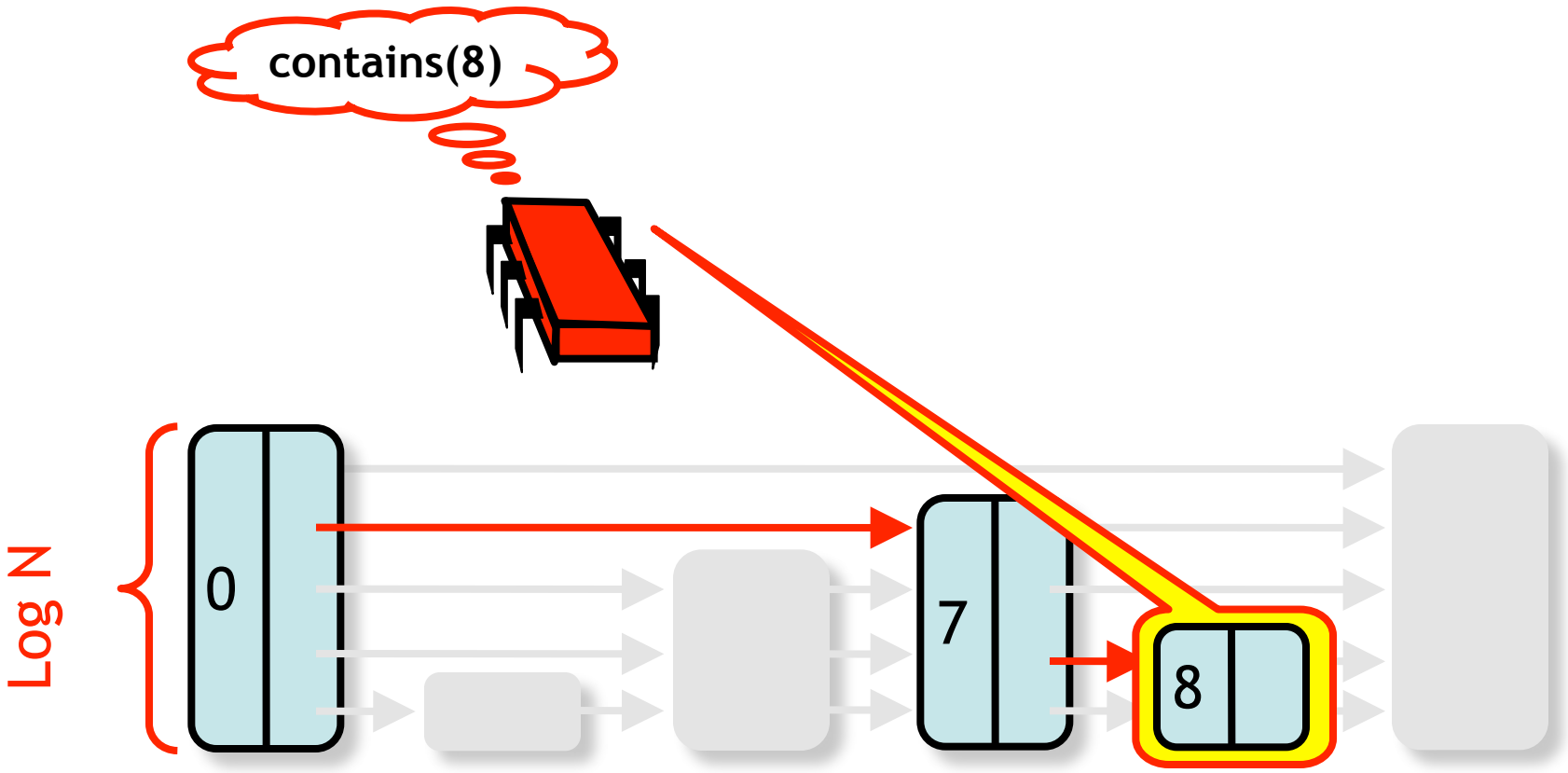




Search

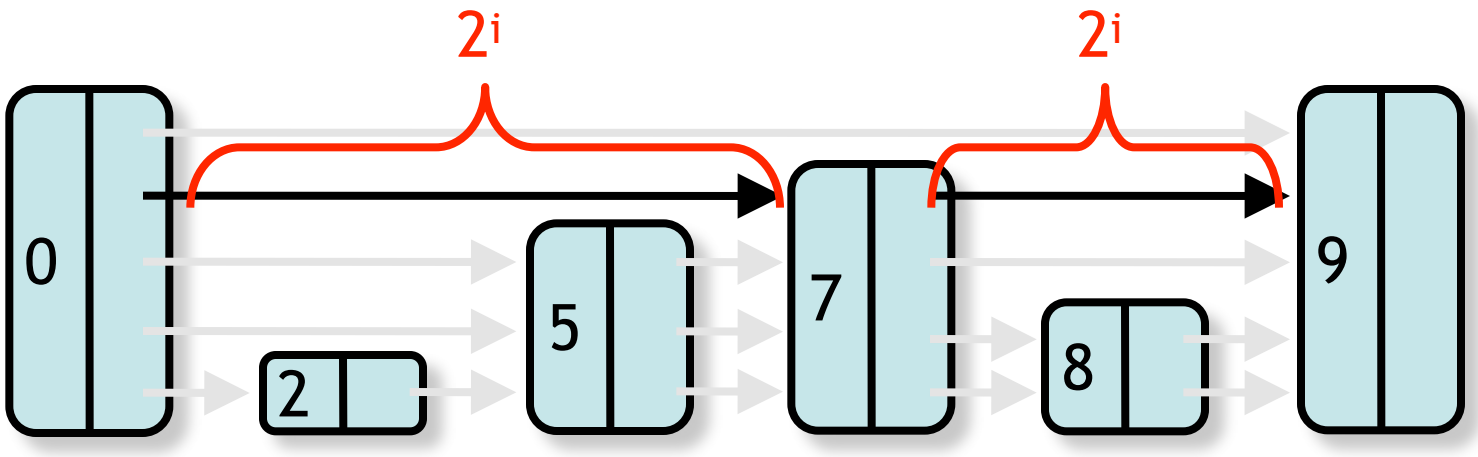


Logarithmic



Why Logarthimic

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



Find() -- Sequential

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

Find() -- Sequential

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

object height
(-1 if not there)

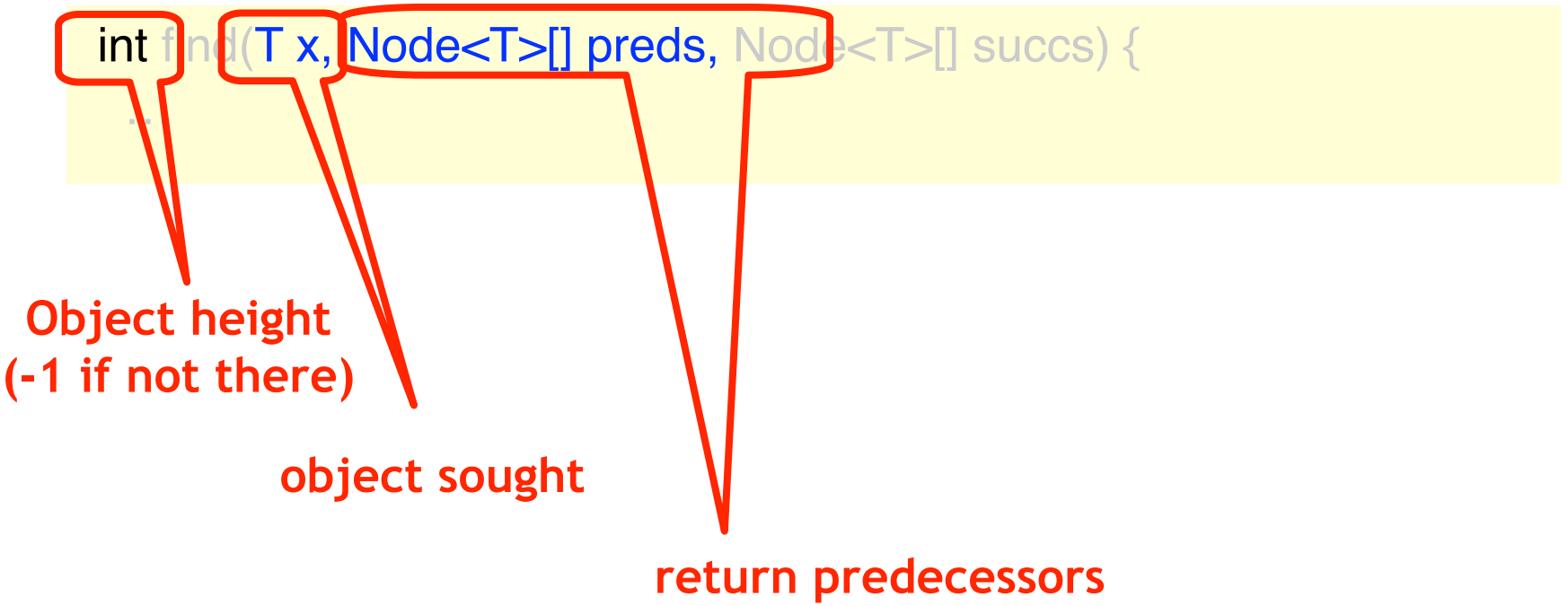
Find() -- Sequential

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

object height (-1 if not there)

Object sought

Find() -- Sequential



Find() -- Sequential

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

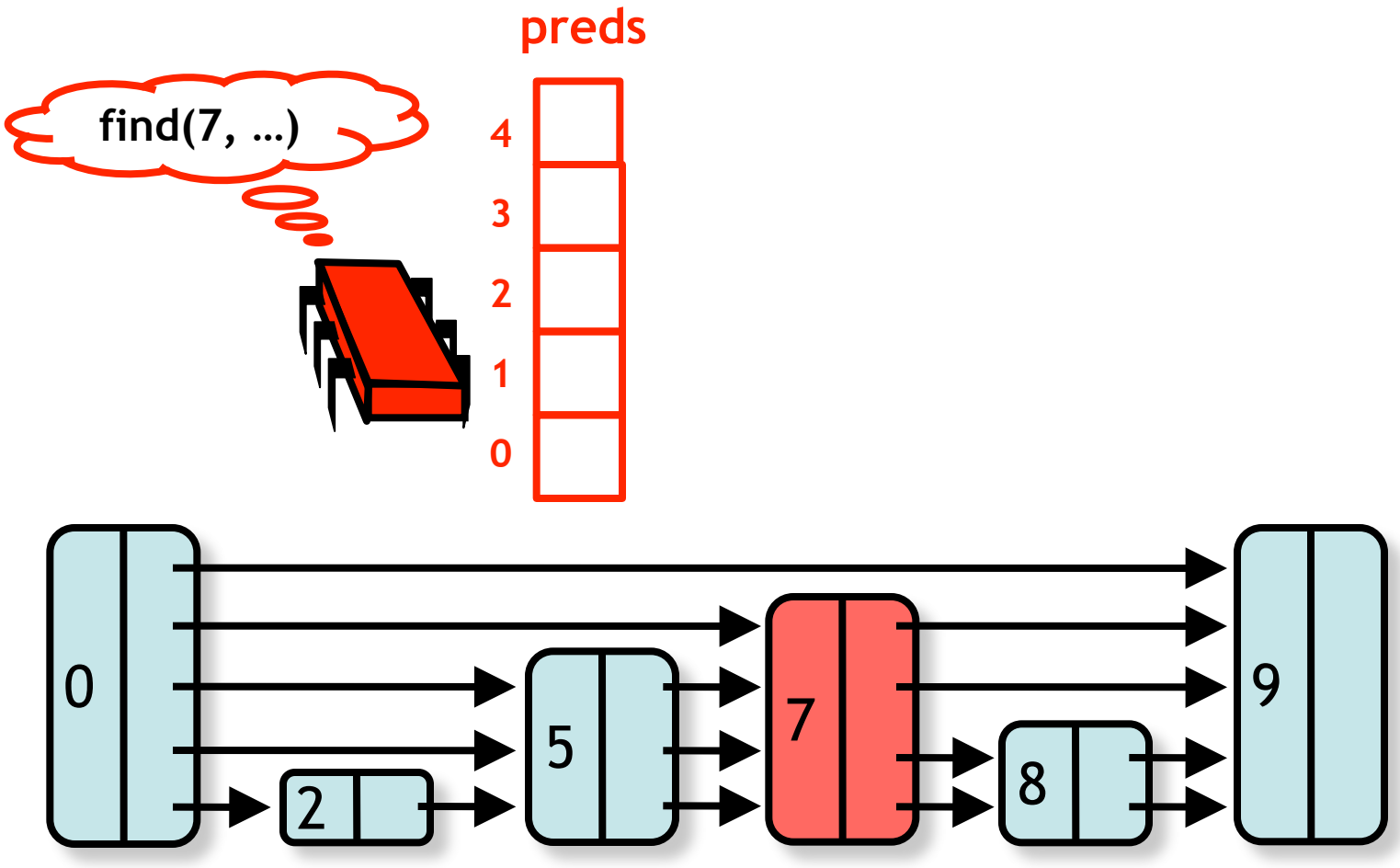
object height (-1 if not there)

object sought

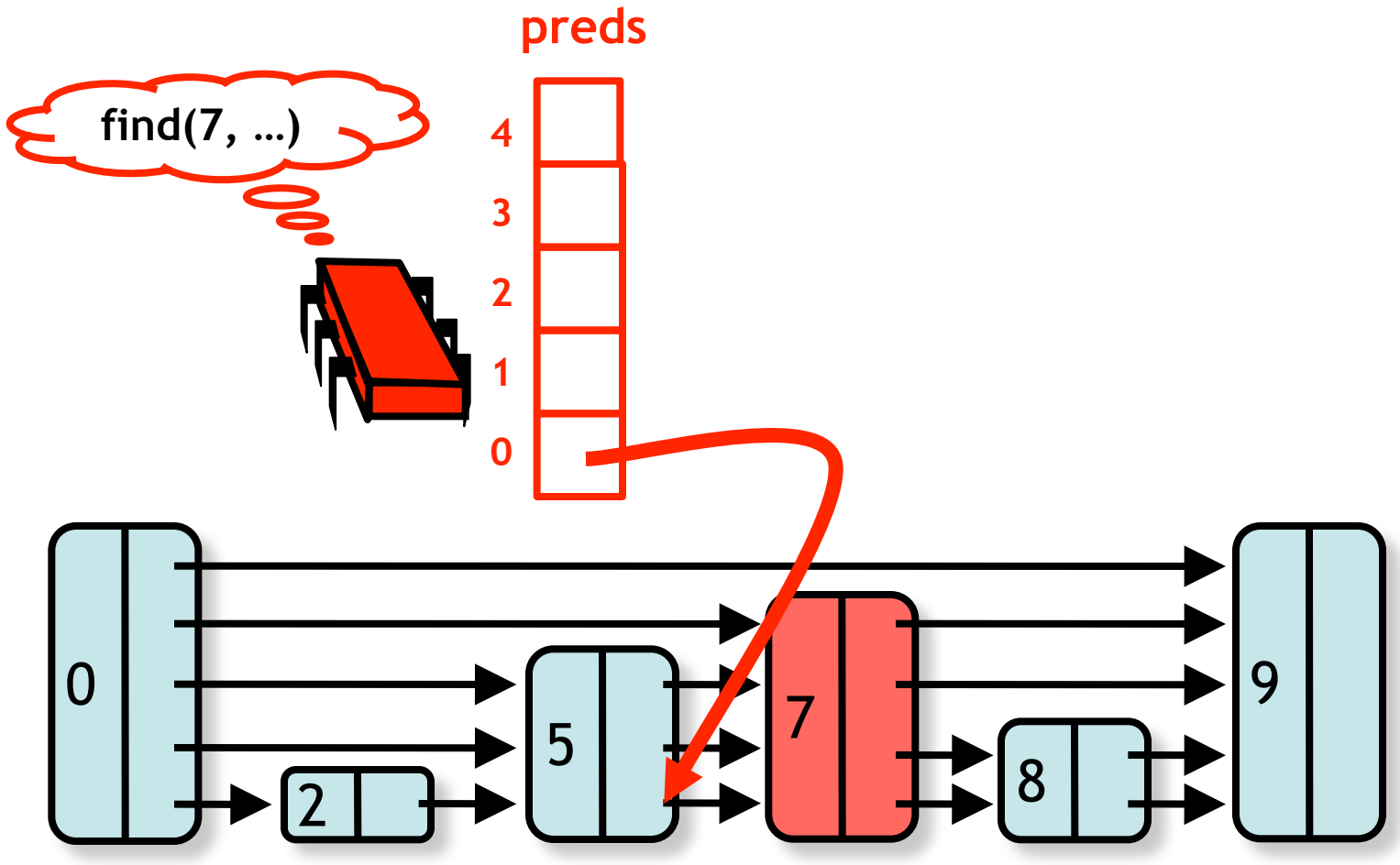
return predecessors

return successors

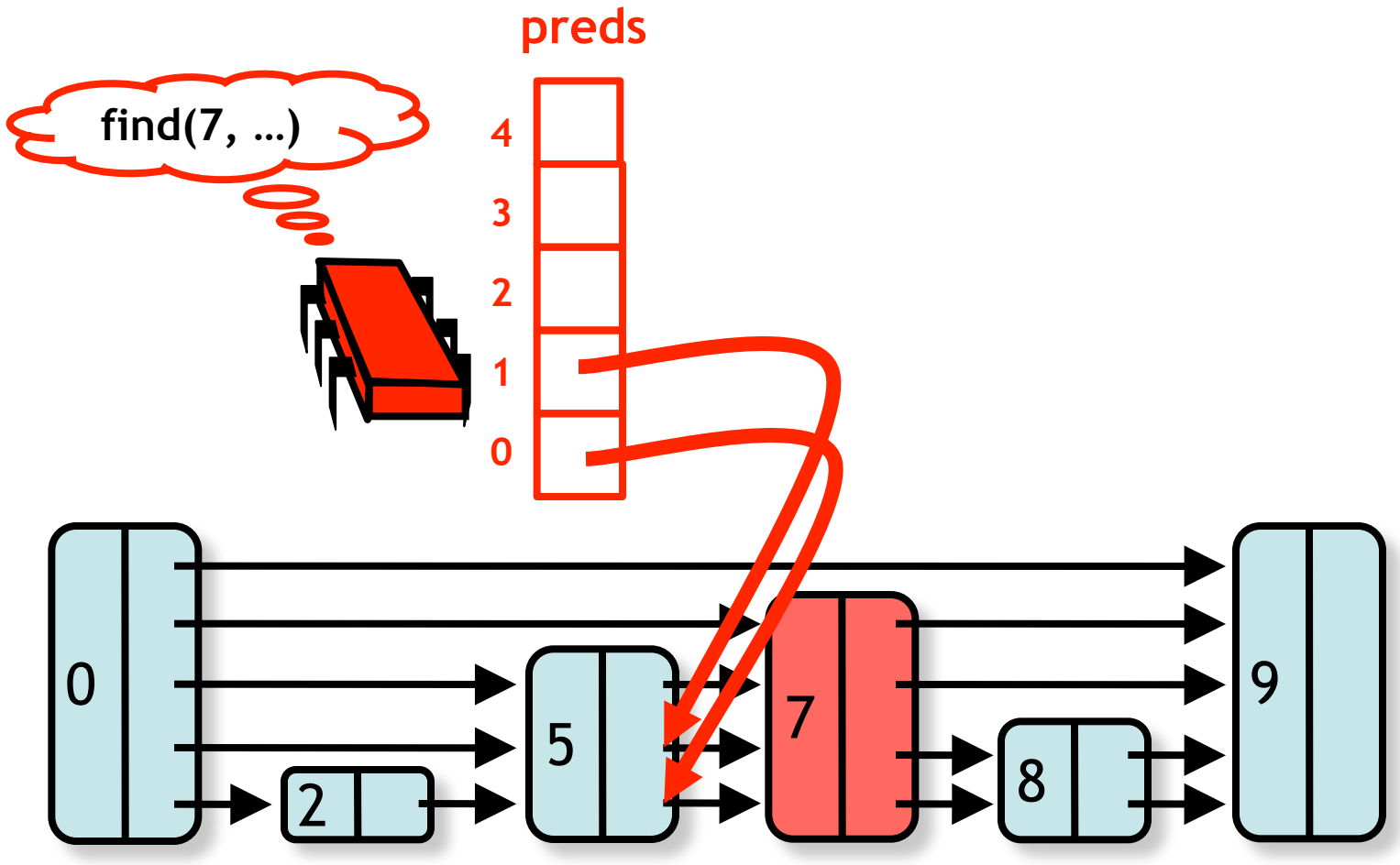
Successful Search



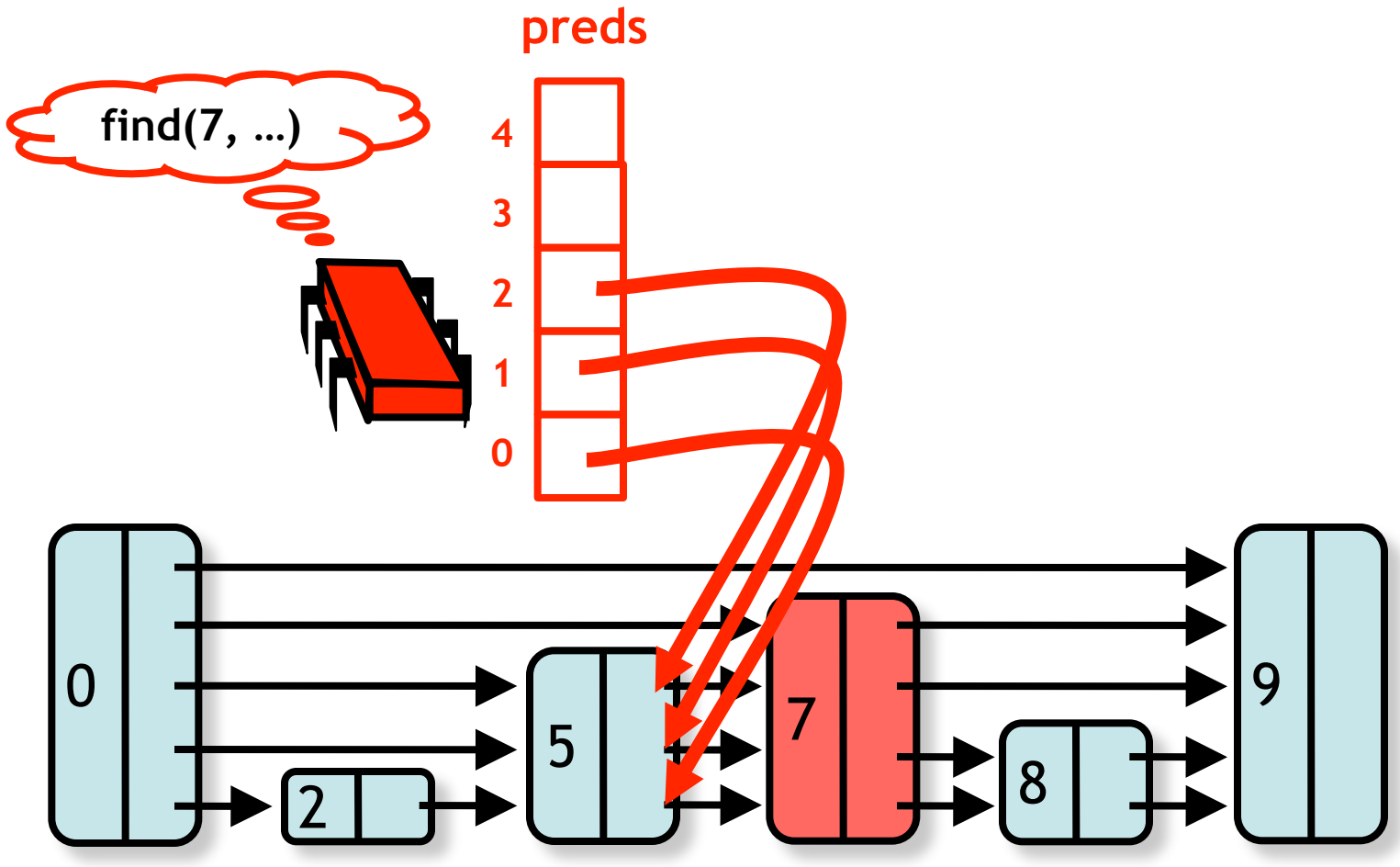
Successful Search



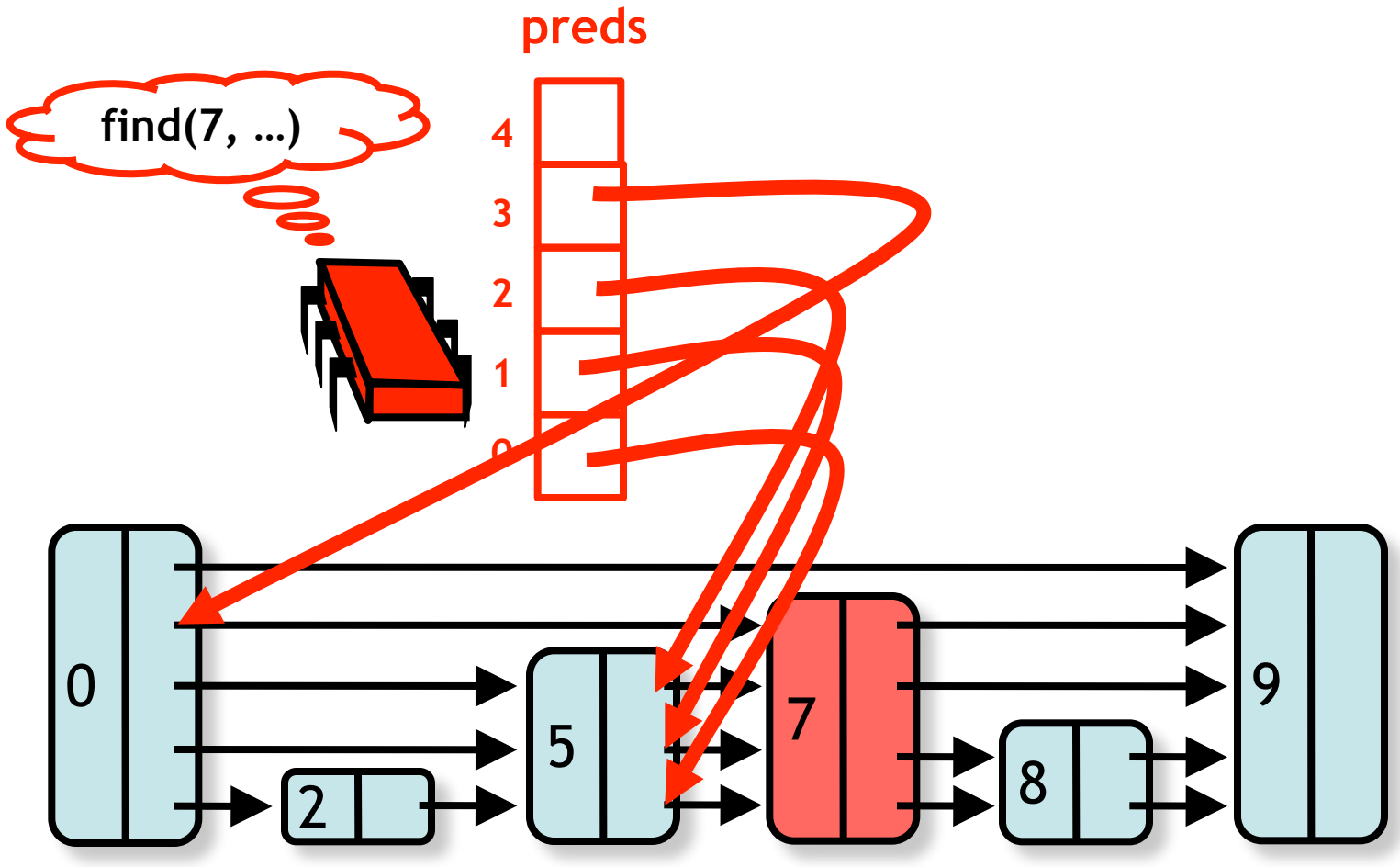
Successful Search



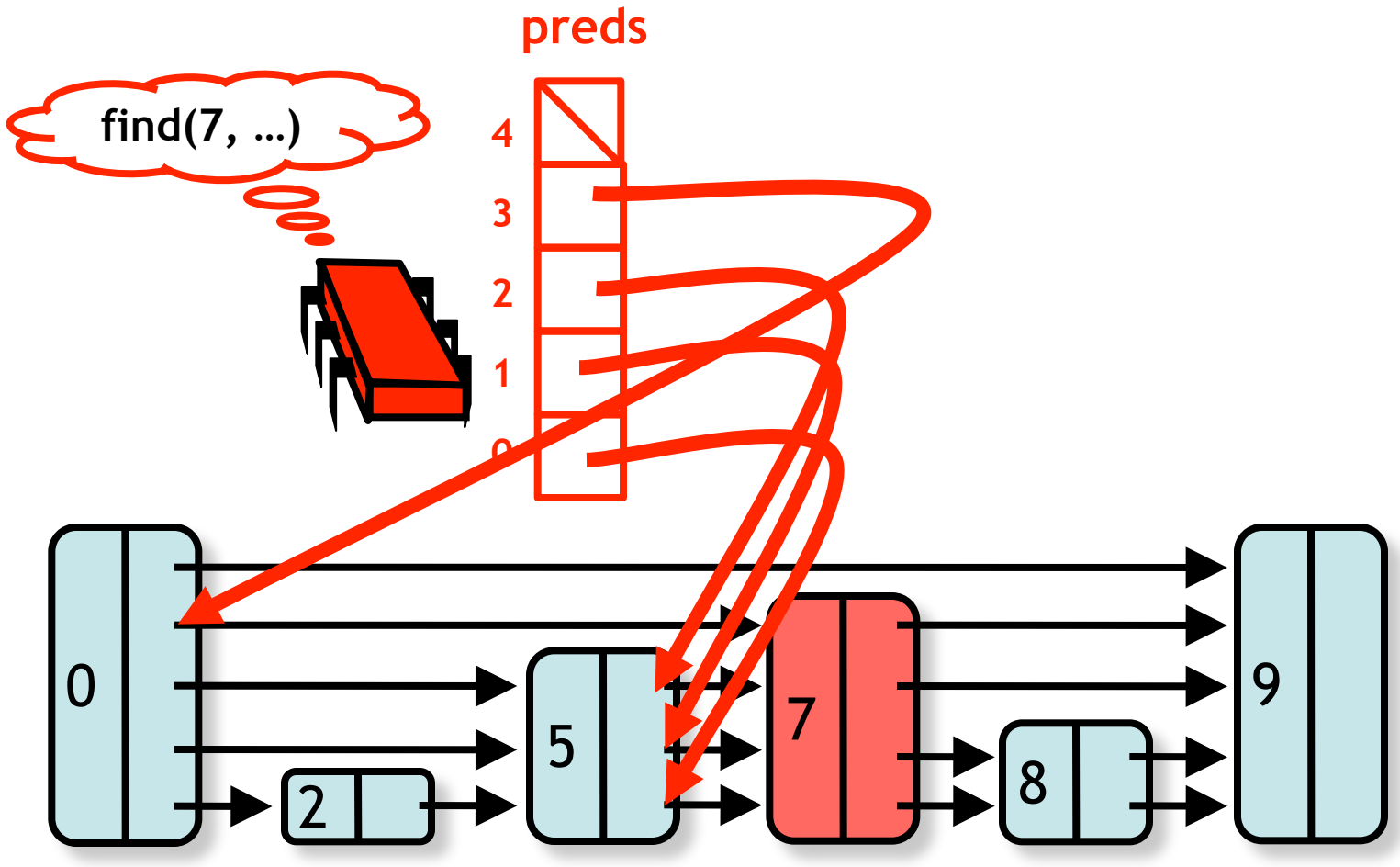
Successful Search



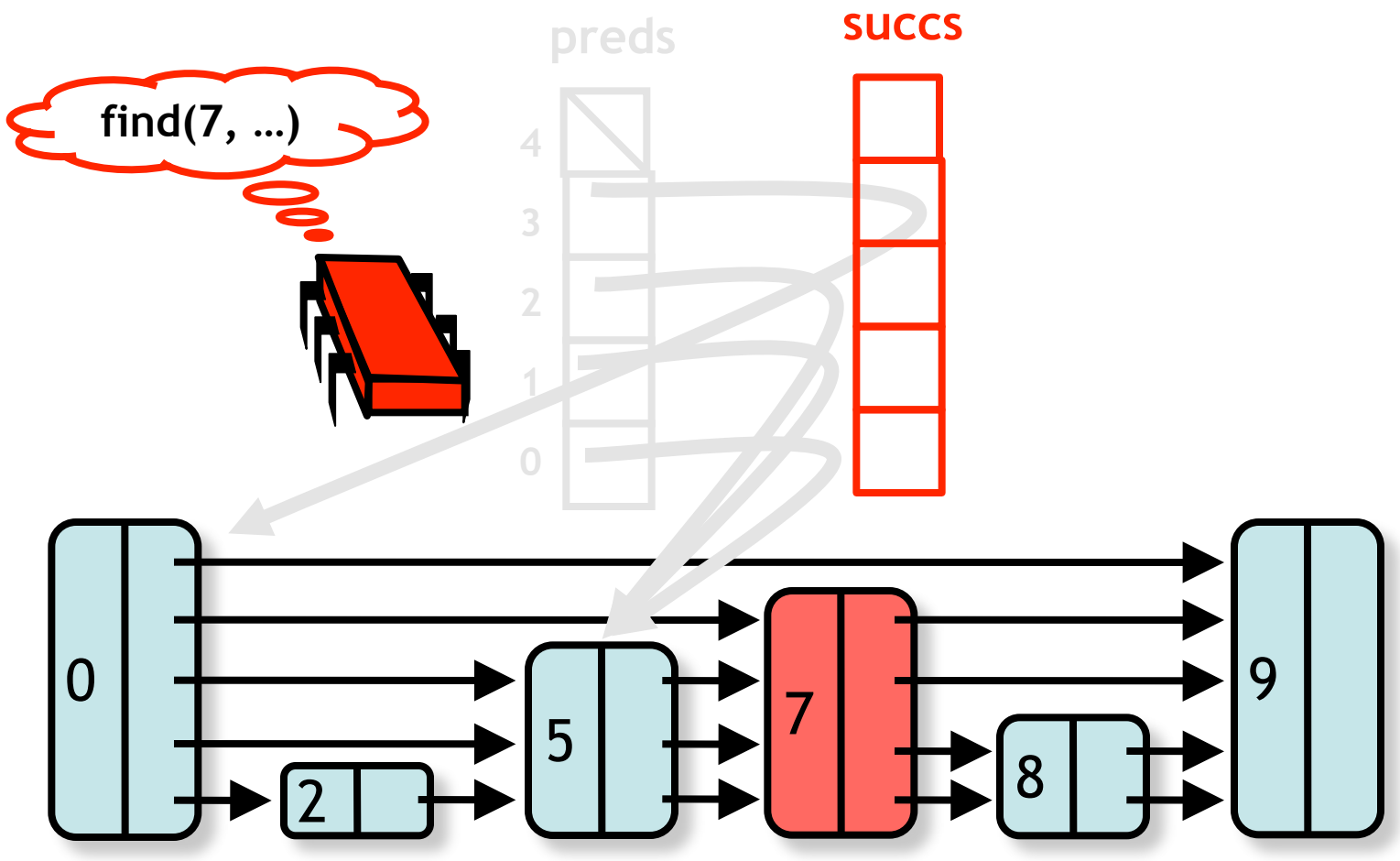
Successful Search



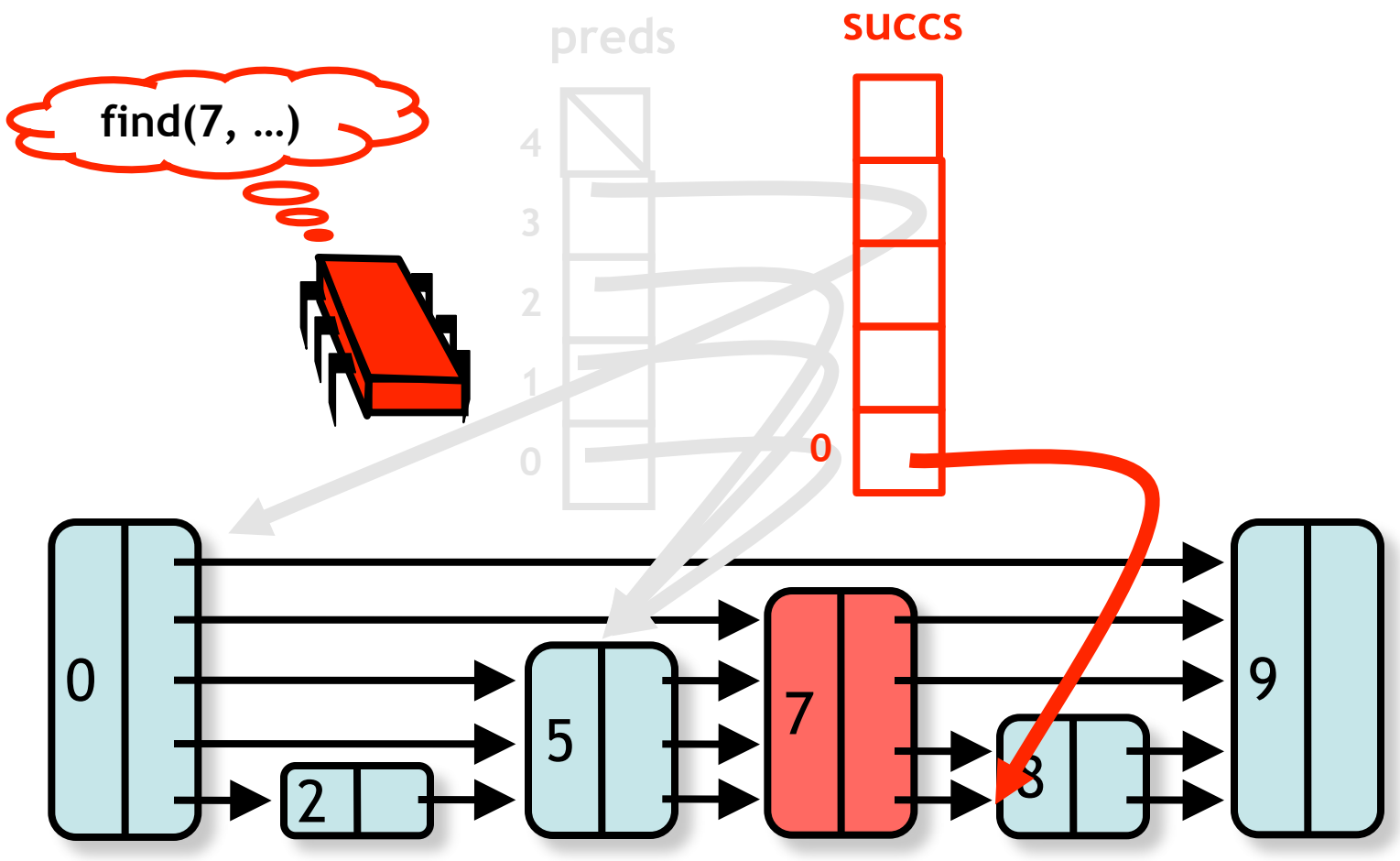
Successful Search



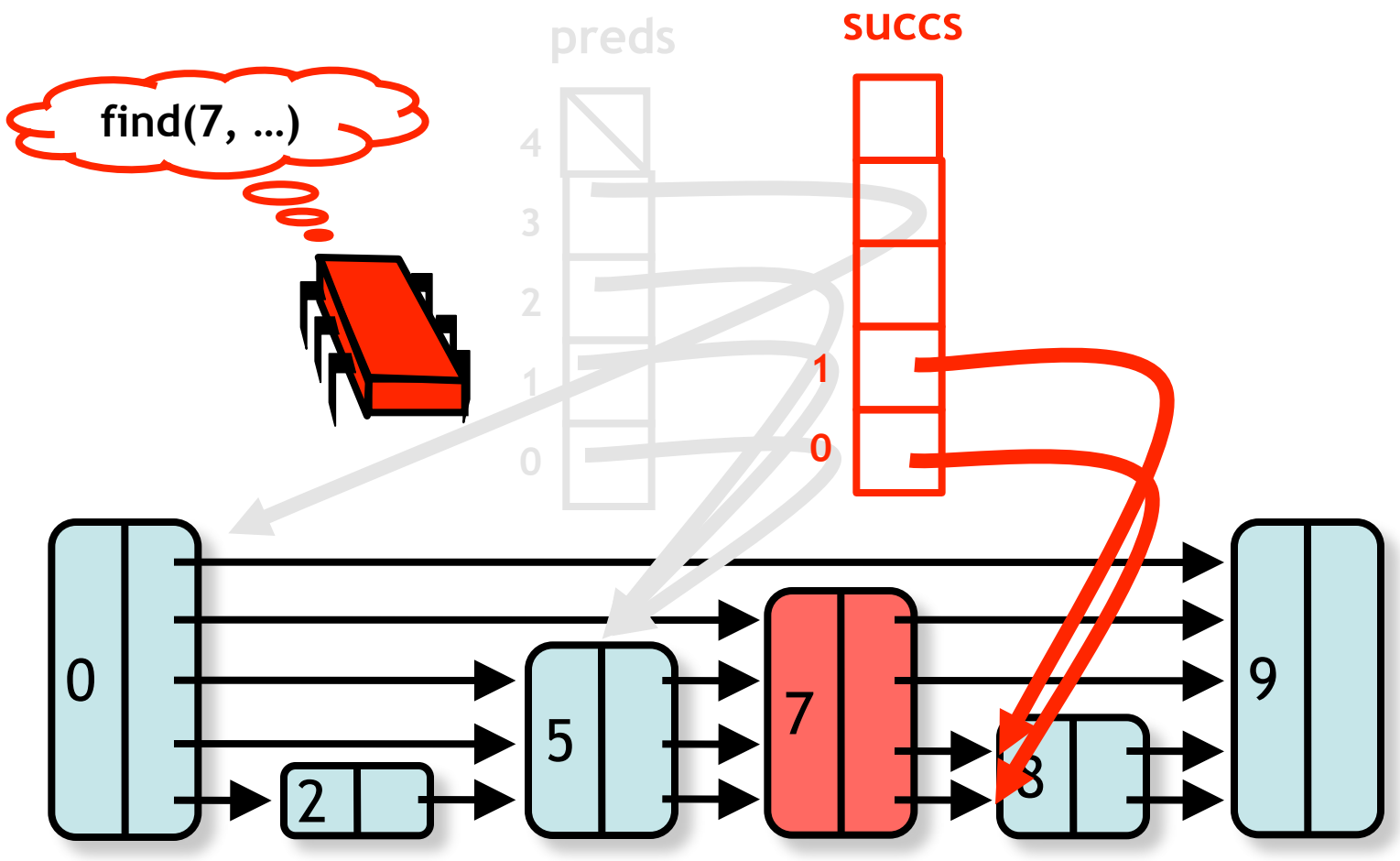
Successful Search



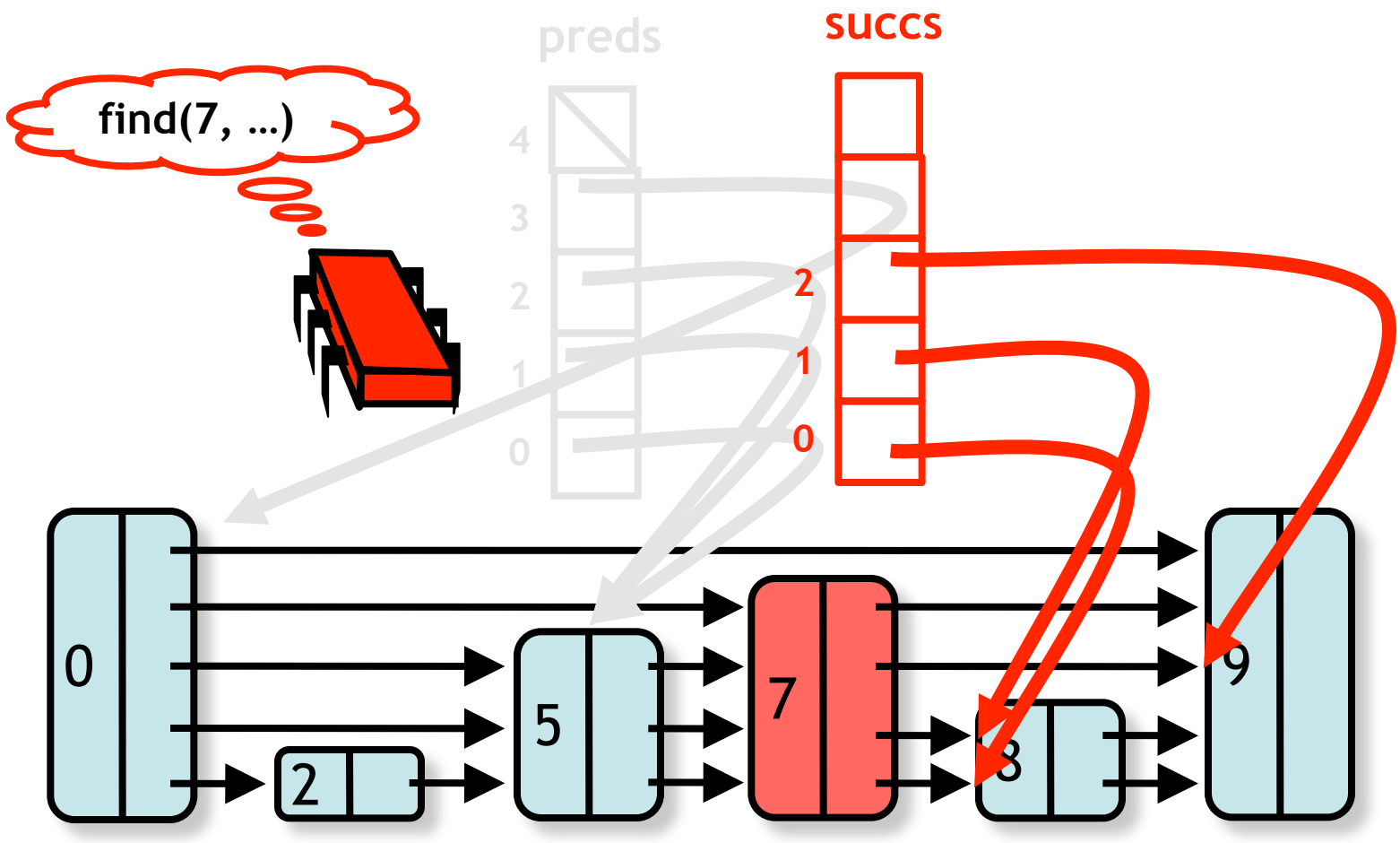
Successful Search



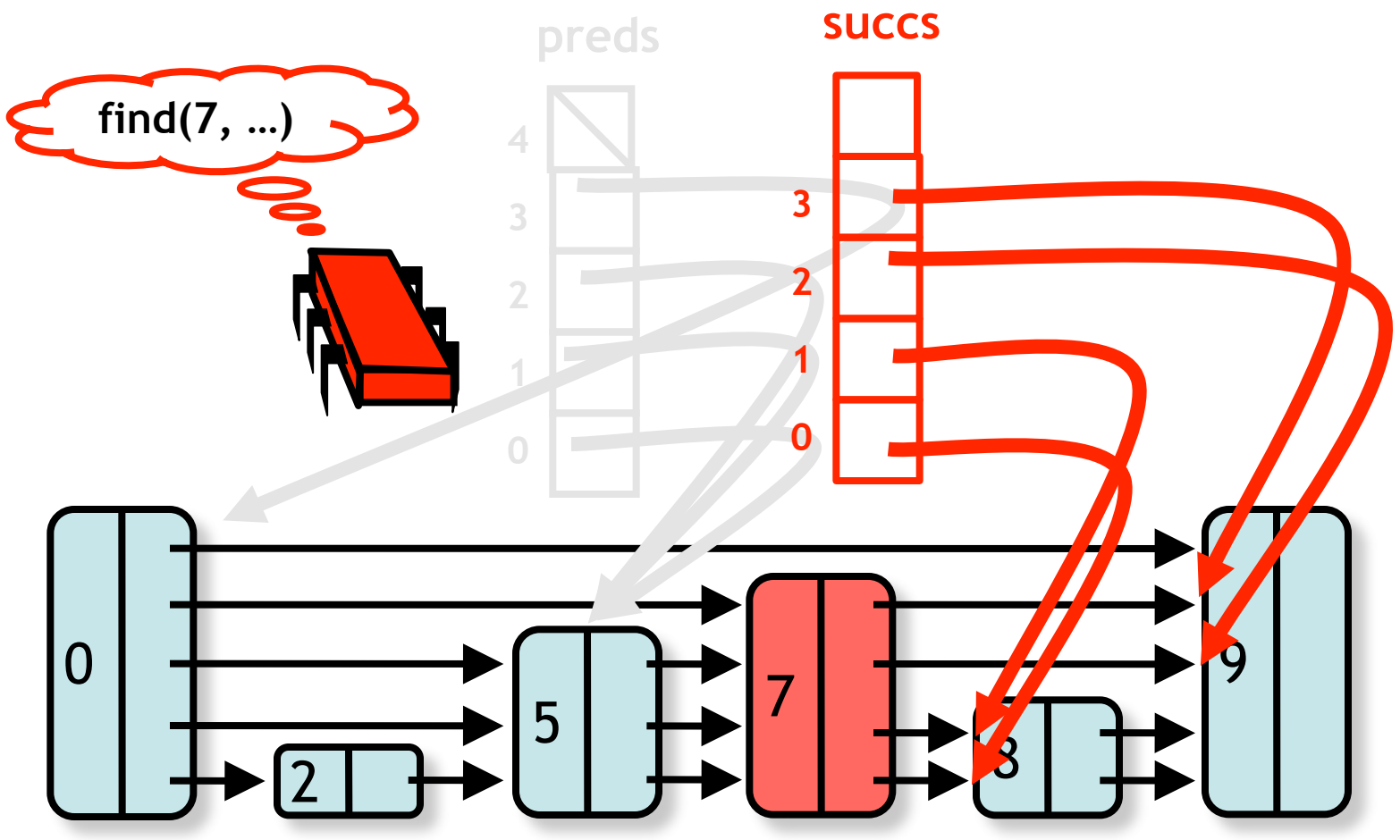
Successful Search



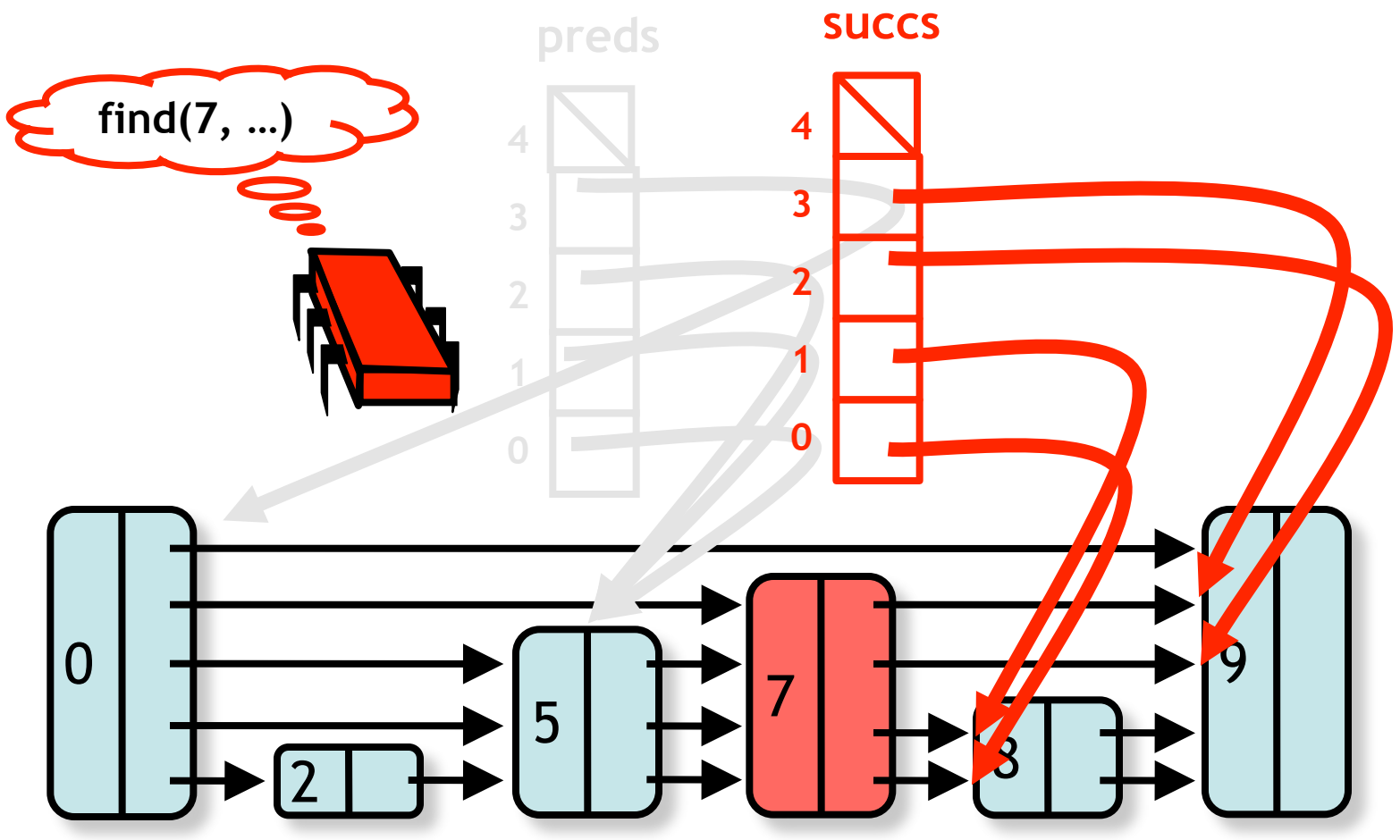
Successful Search



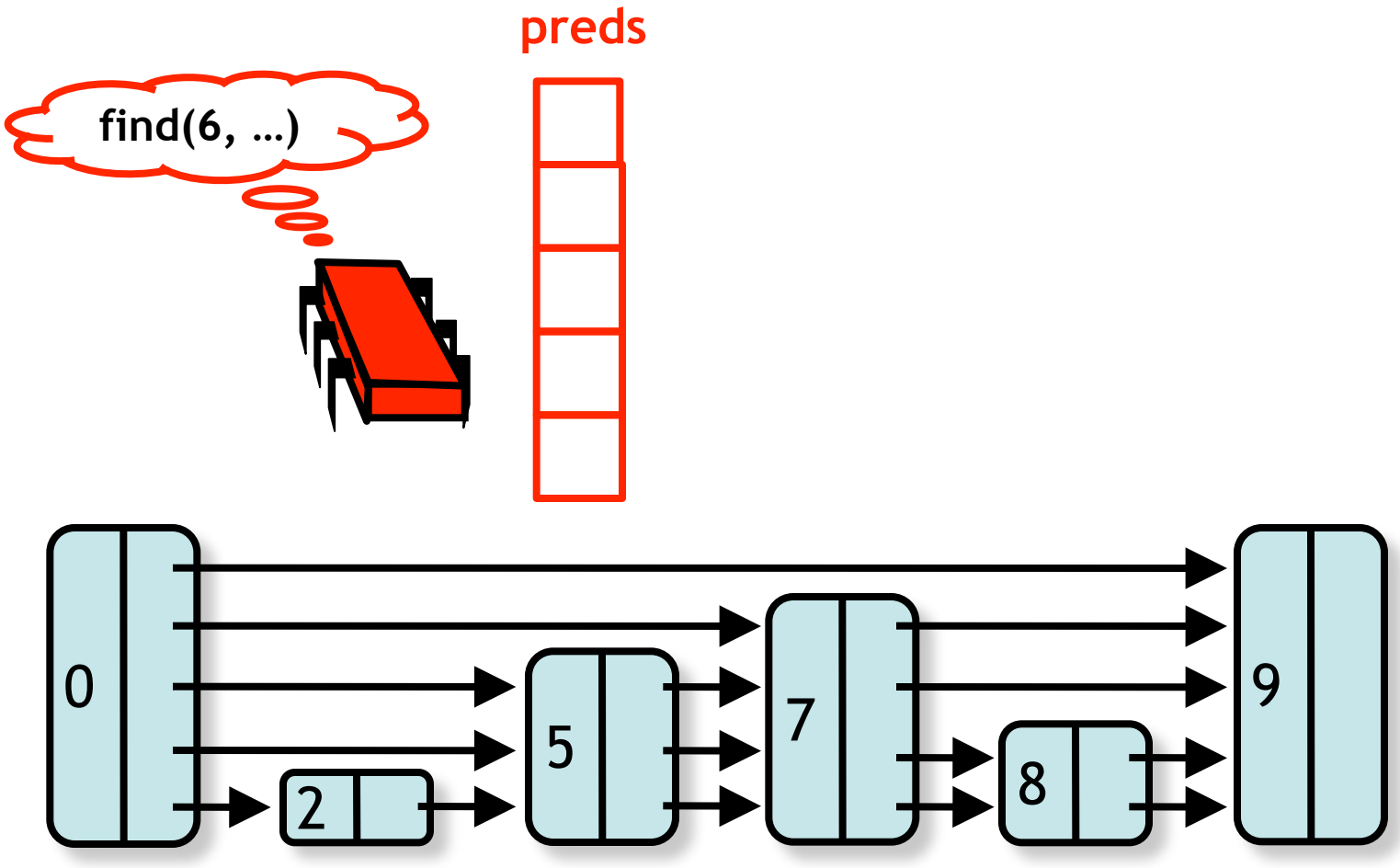
Successful Search



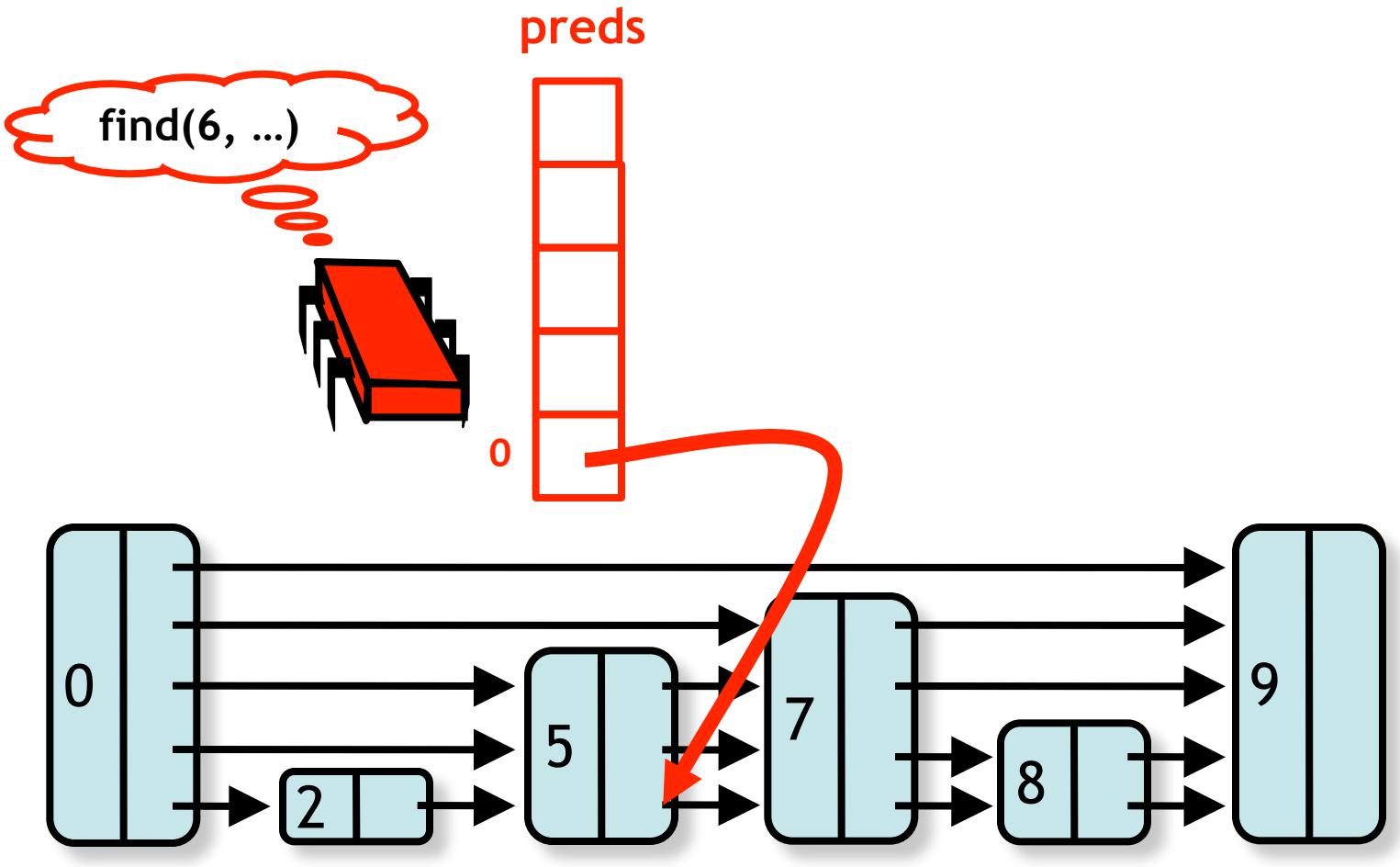
Successful Search



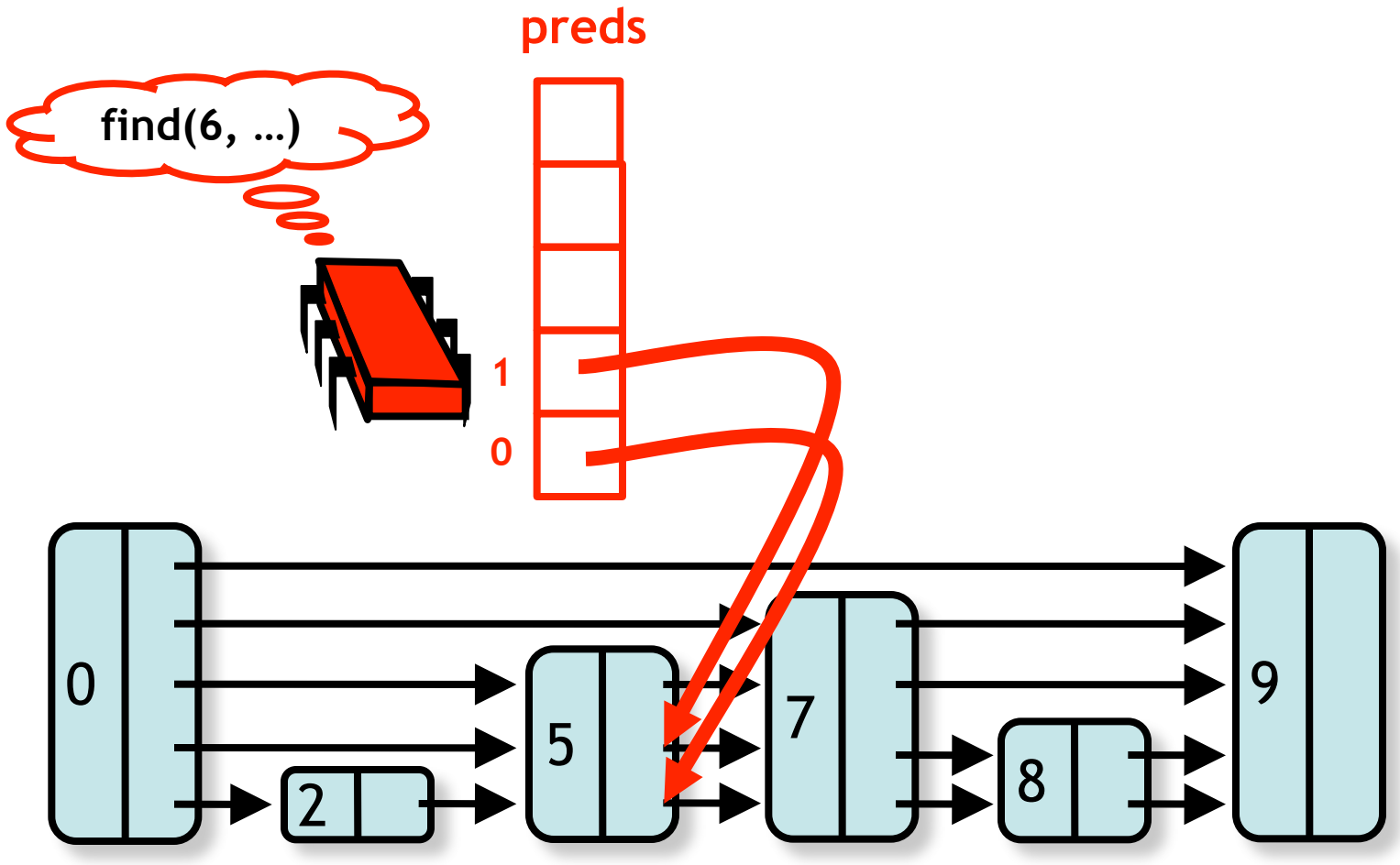
Unsuccessful Search



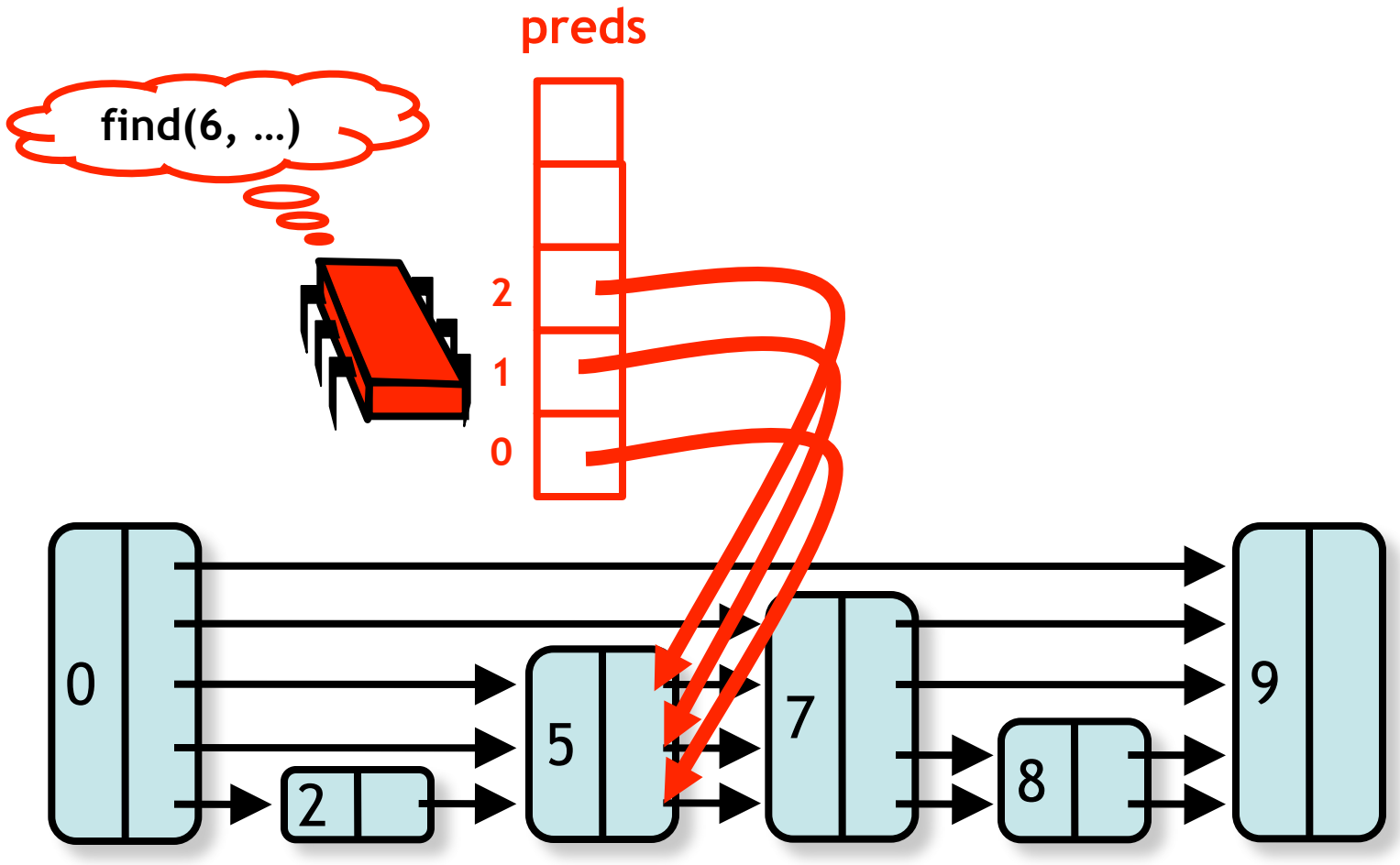
Unsuccessful Search



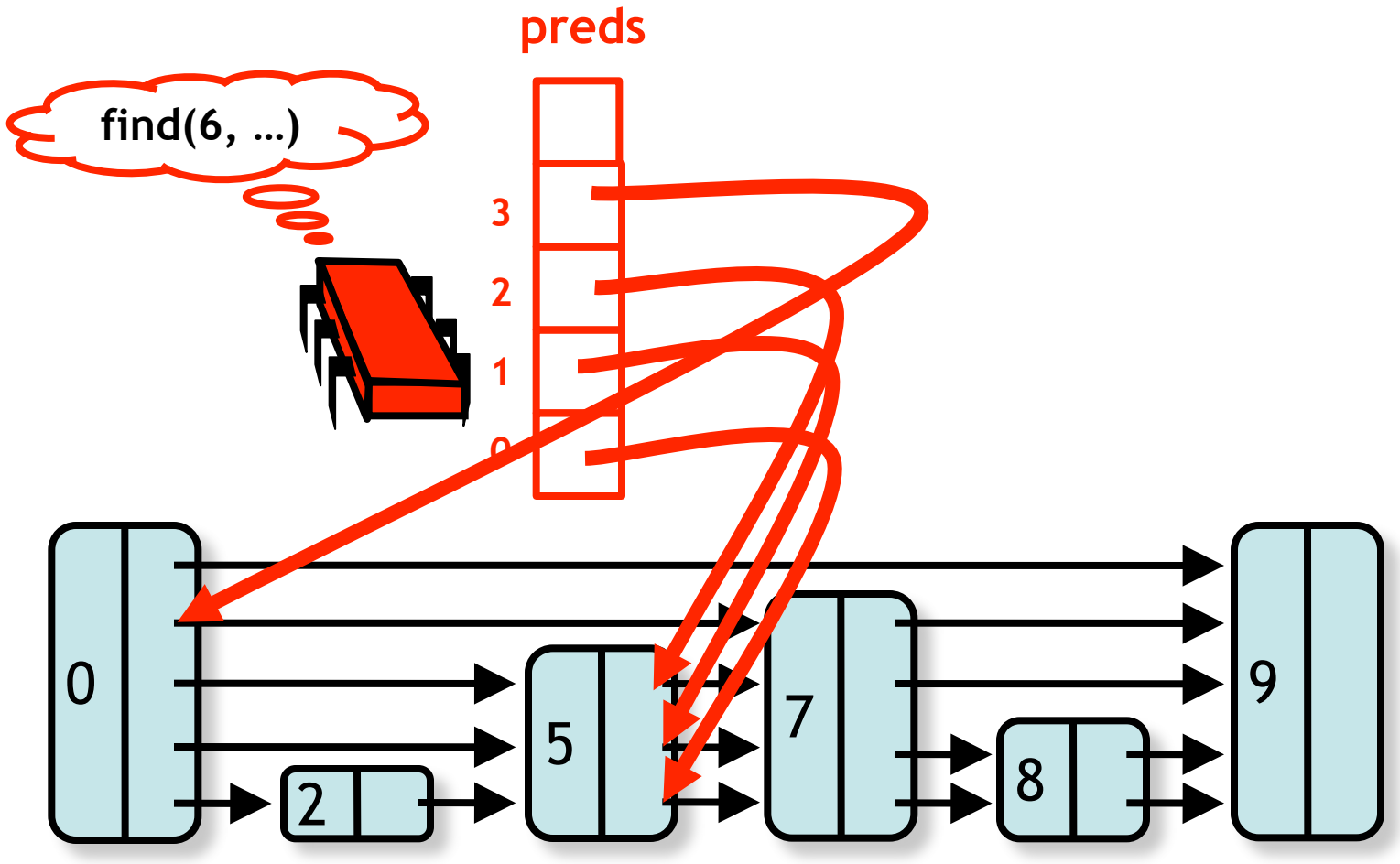
Unsuccessful Search



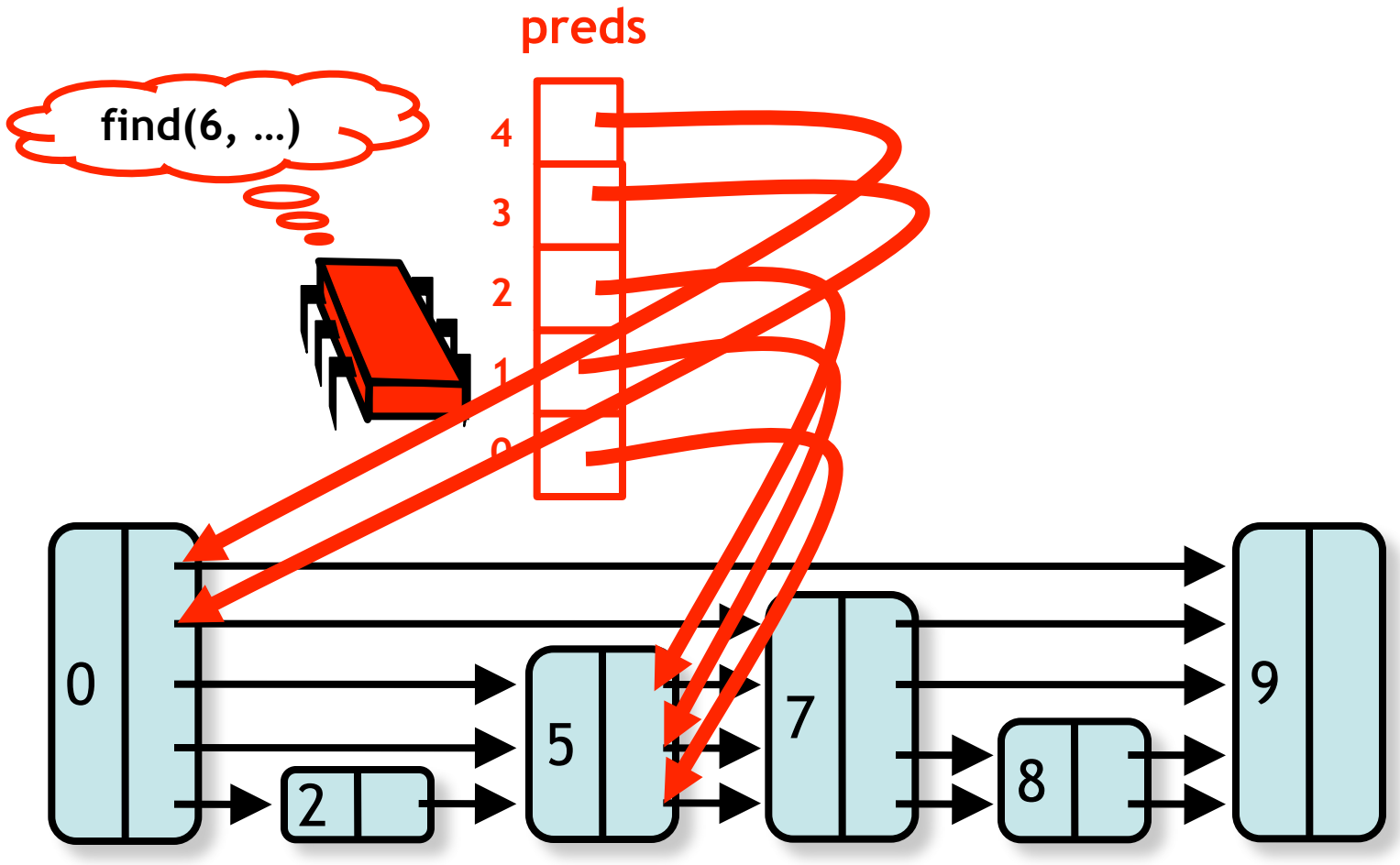
Unsuccessful Search



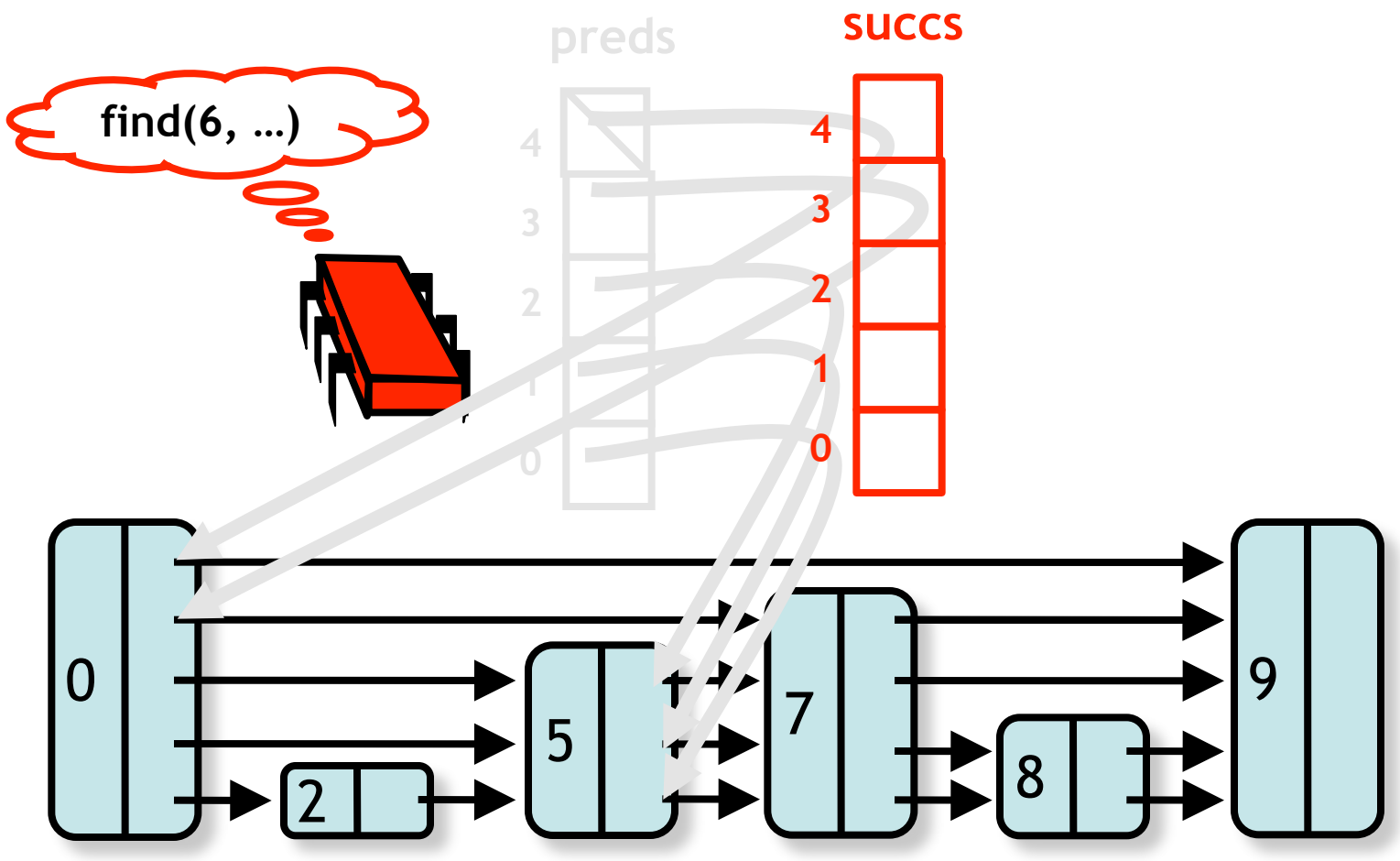
Unsuccessful Search



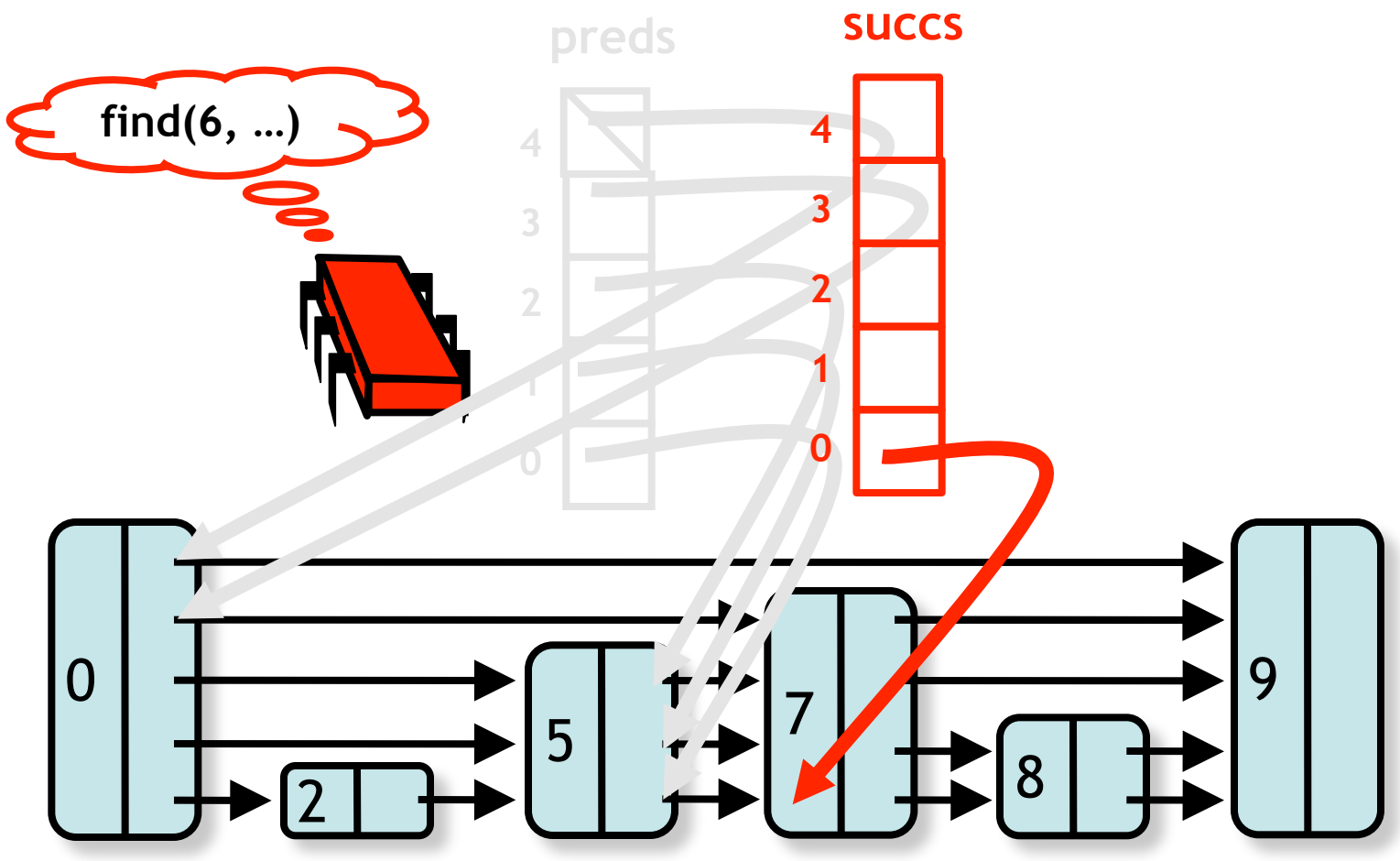
Unsuccessful Search



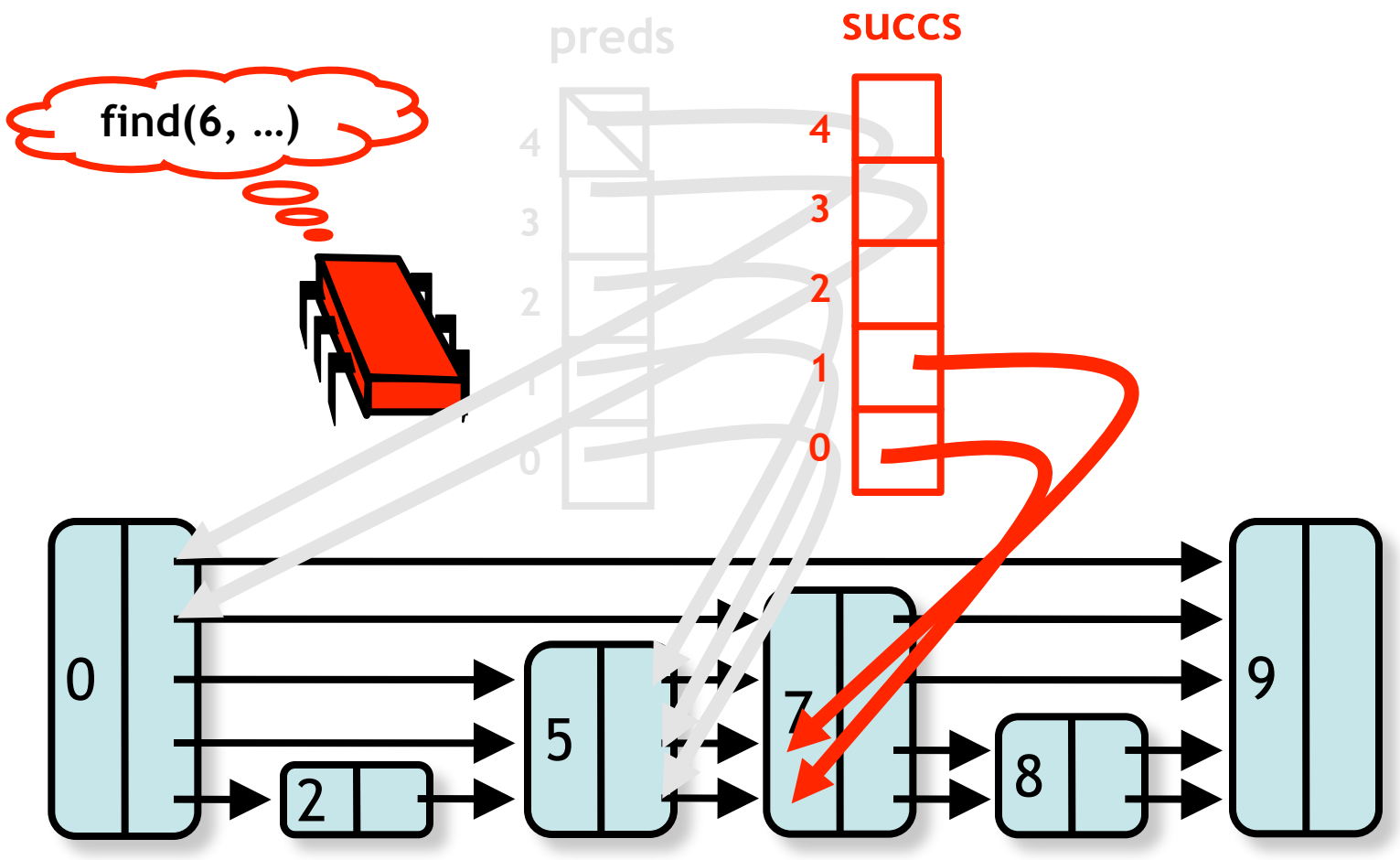
Unsuccessful Search



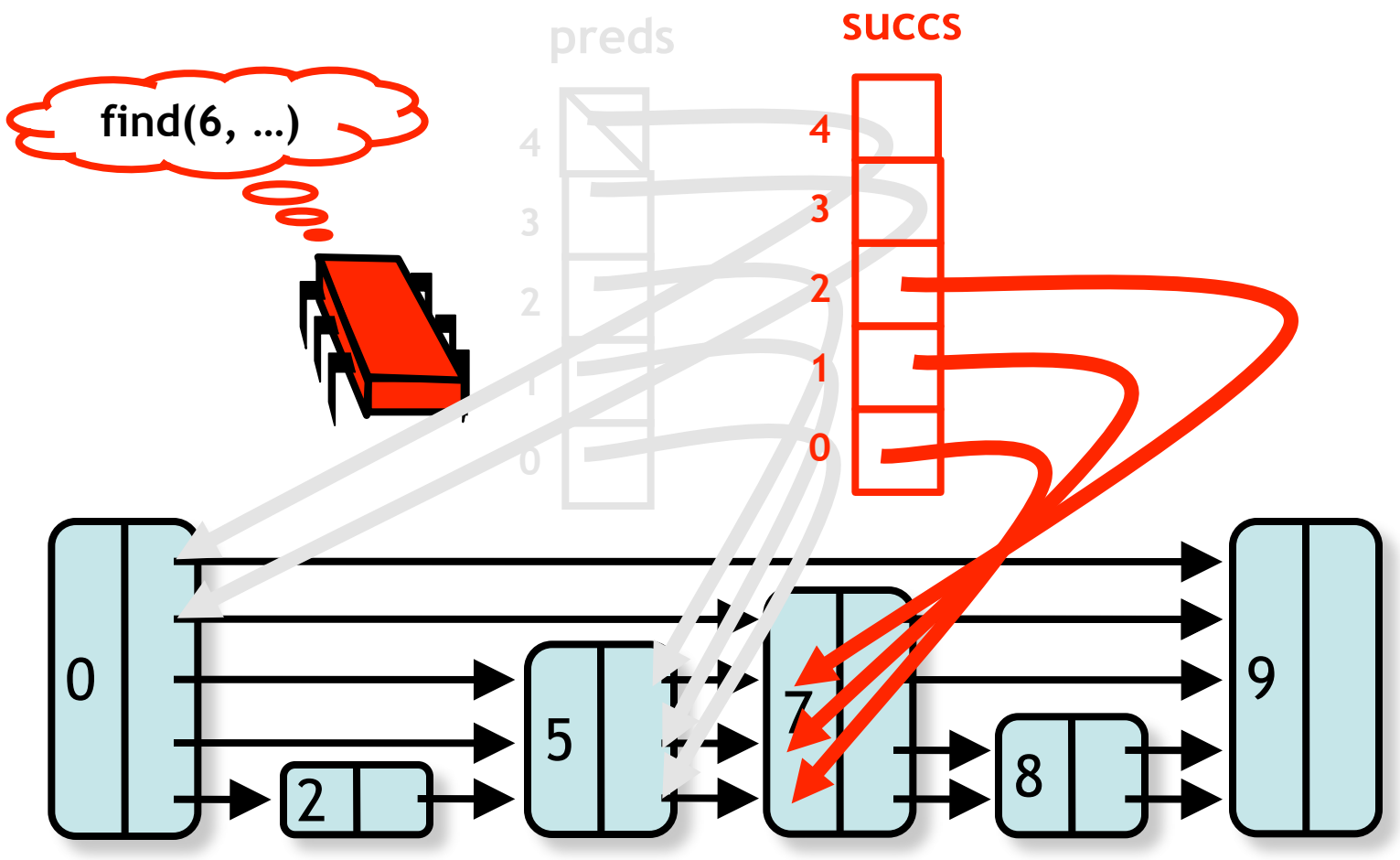
Unsuccessful Search



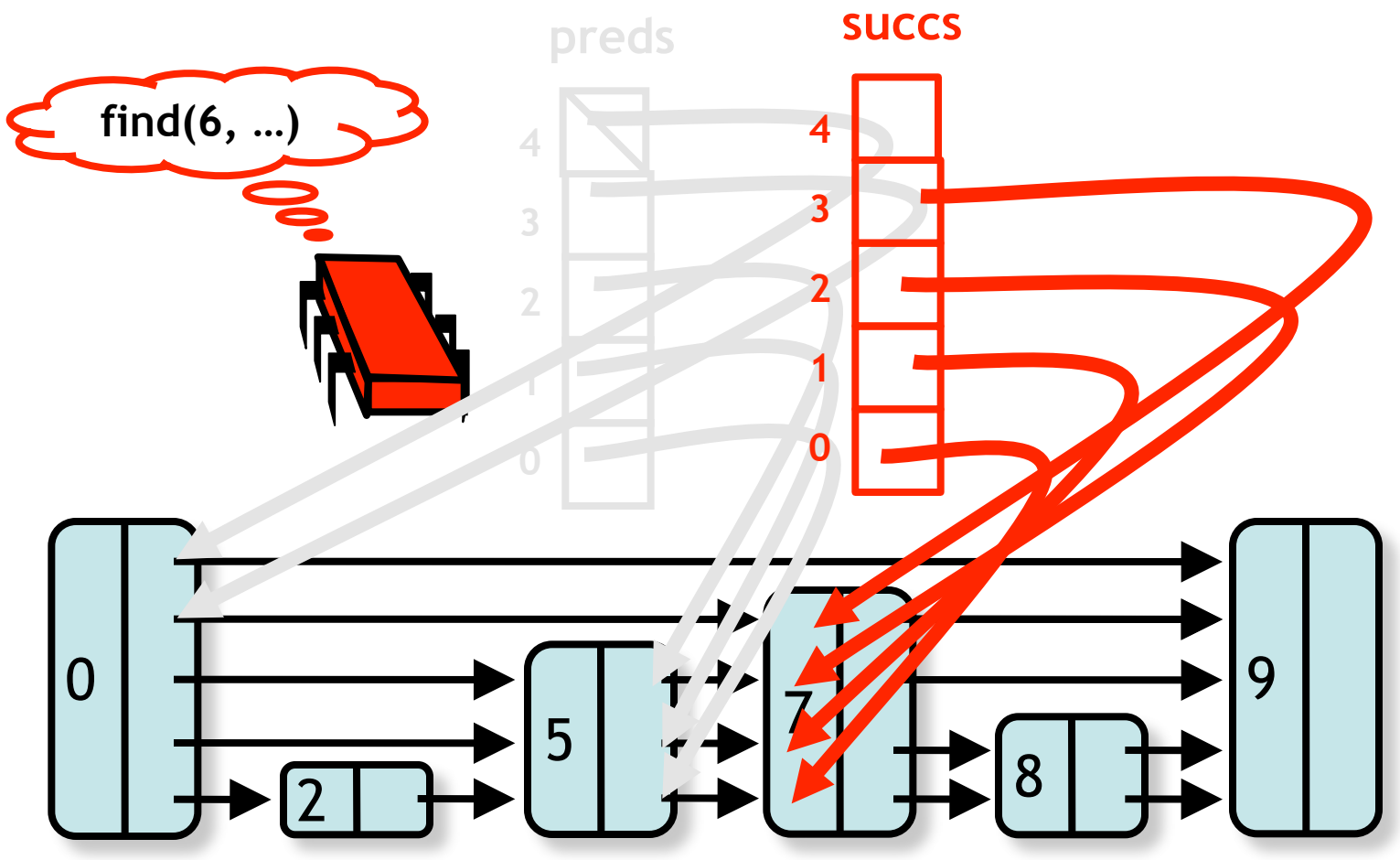
Unsuccessful Search



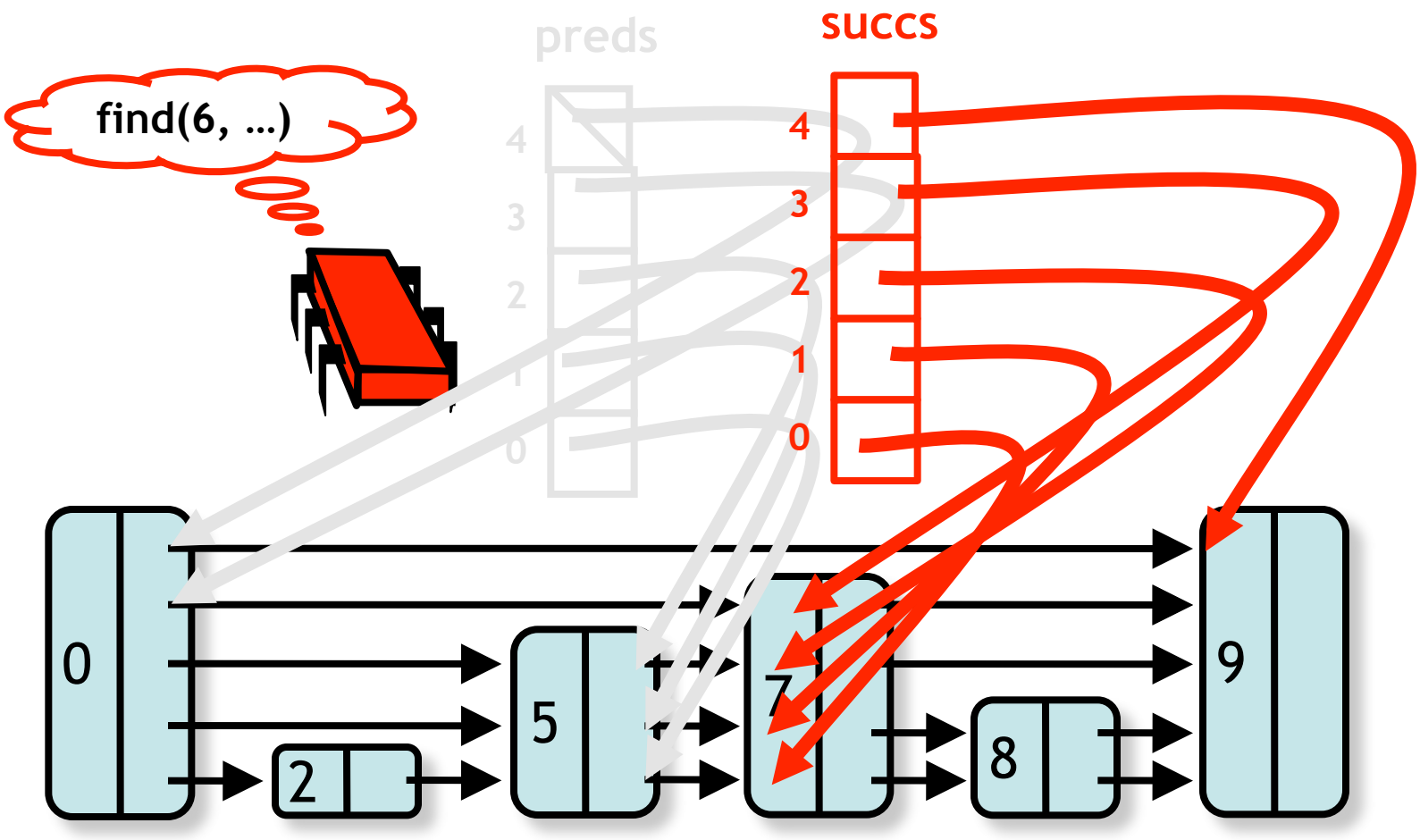
Unsuccessful Search



Unsuccessful Search



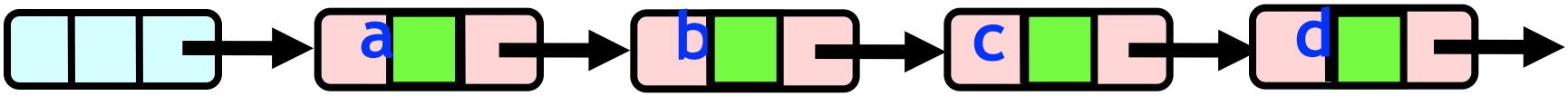
Unsuccessful Search



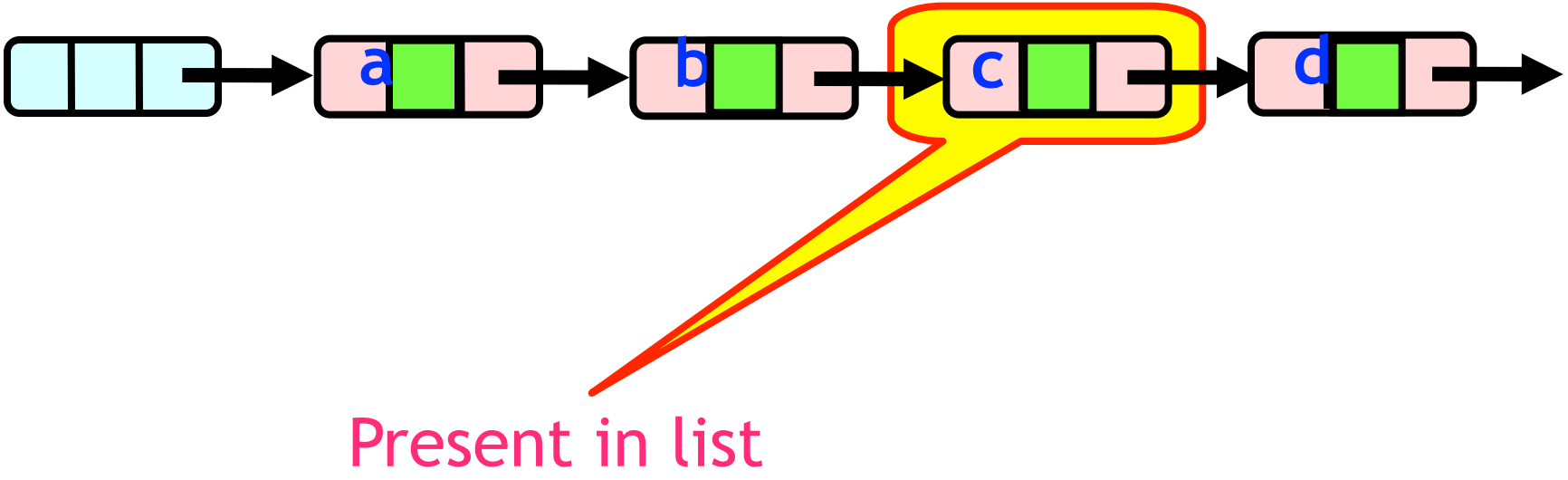
Lazy Skip List

- 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()

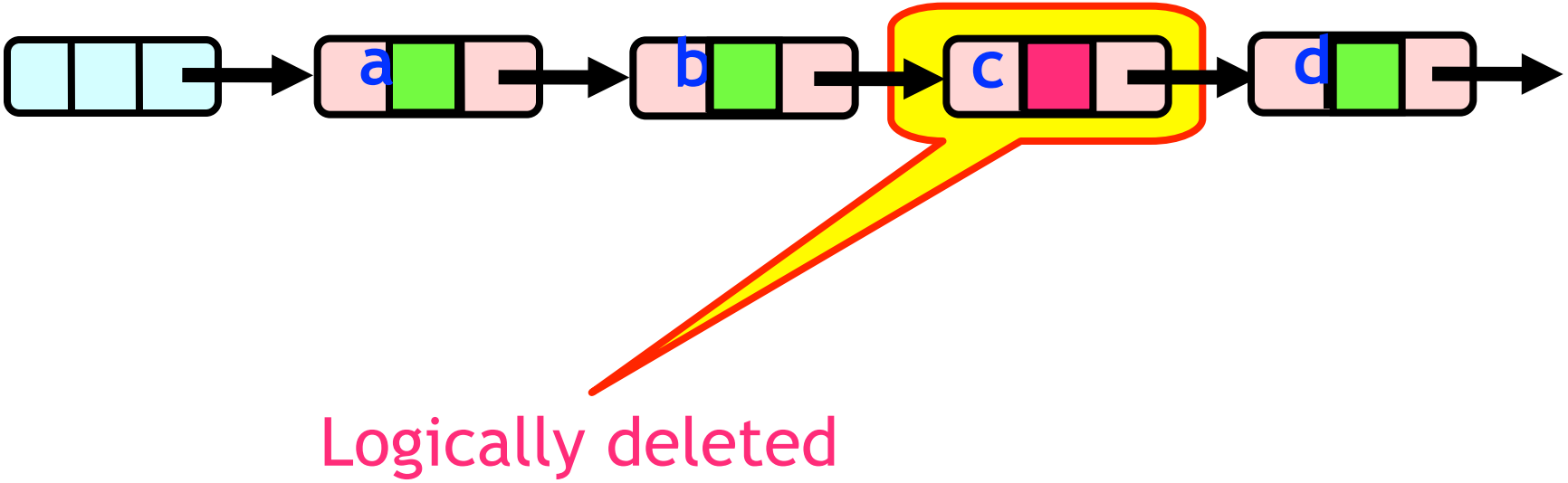
Review: Lazy List Remove



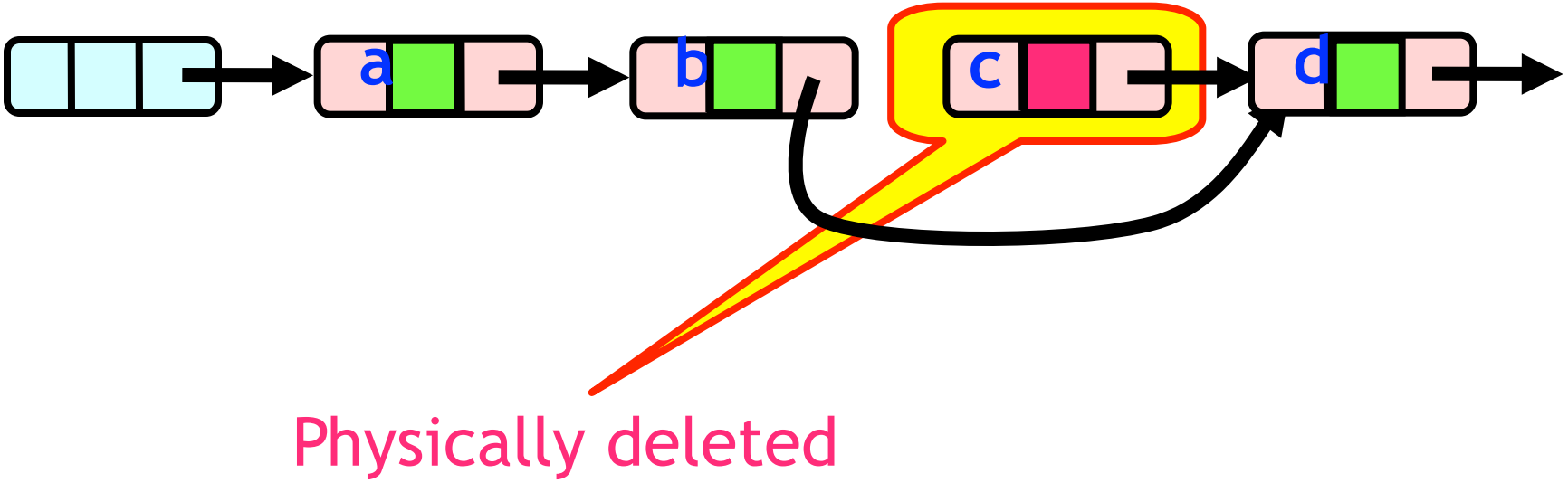
Review: Lazy List Remove



Review: Lazy List Remove

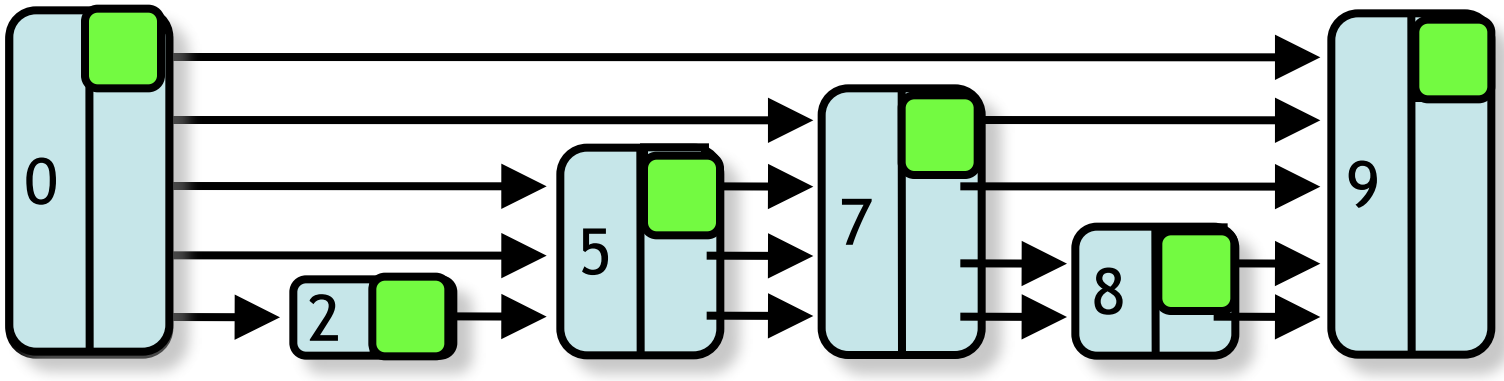


Review: Lazy List Remove



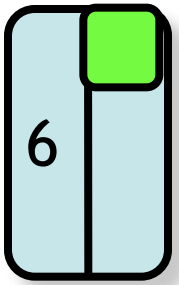
Lazy Skip Lists

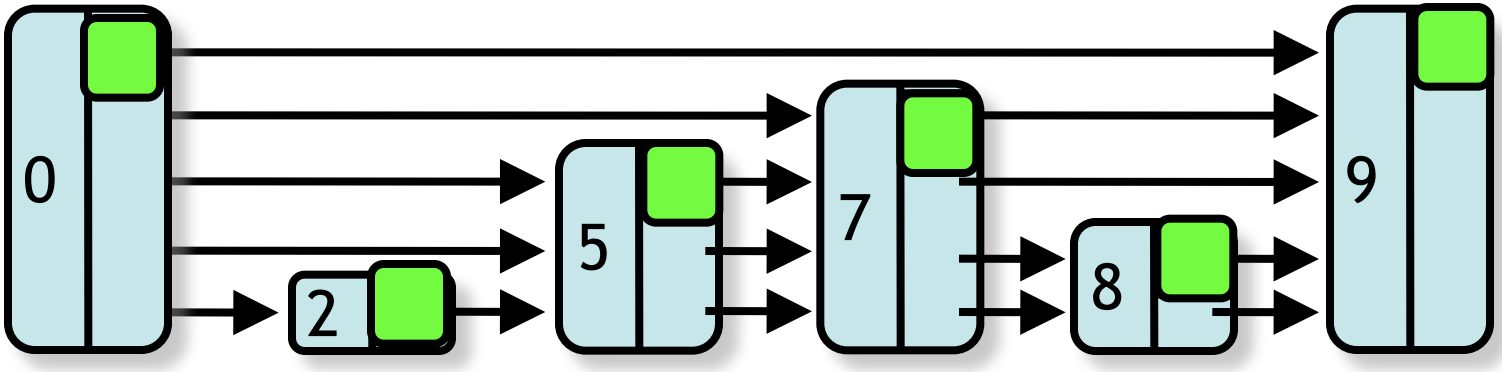
- Use a mark bit for logical deletion



add(6)

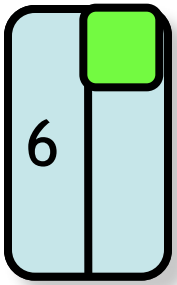
- Create node of (random) height 4

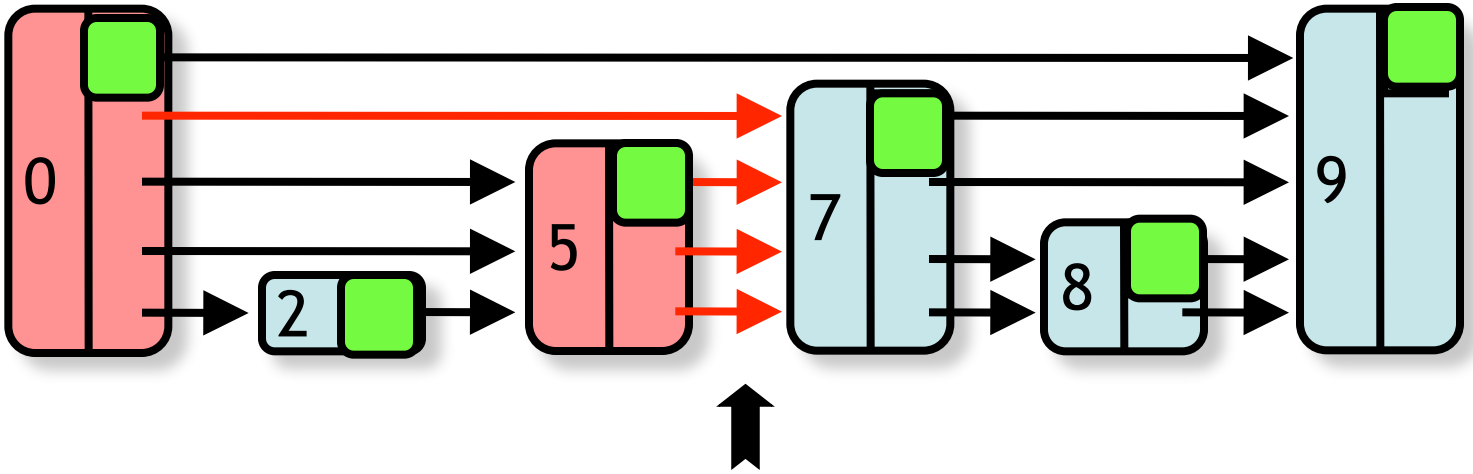




add(6)

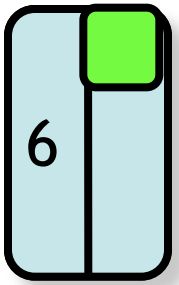
- **find()** predecessors

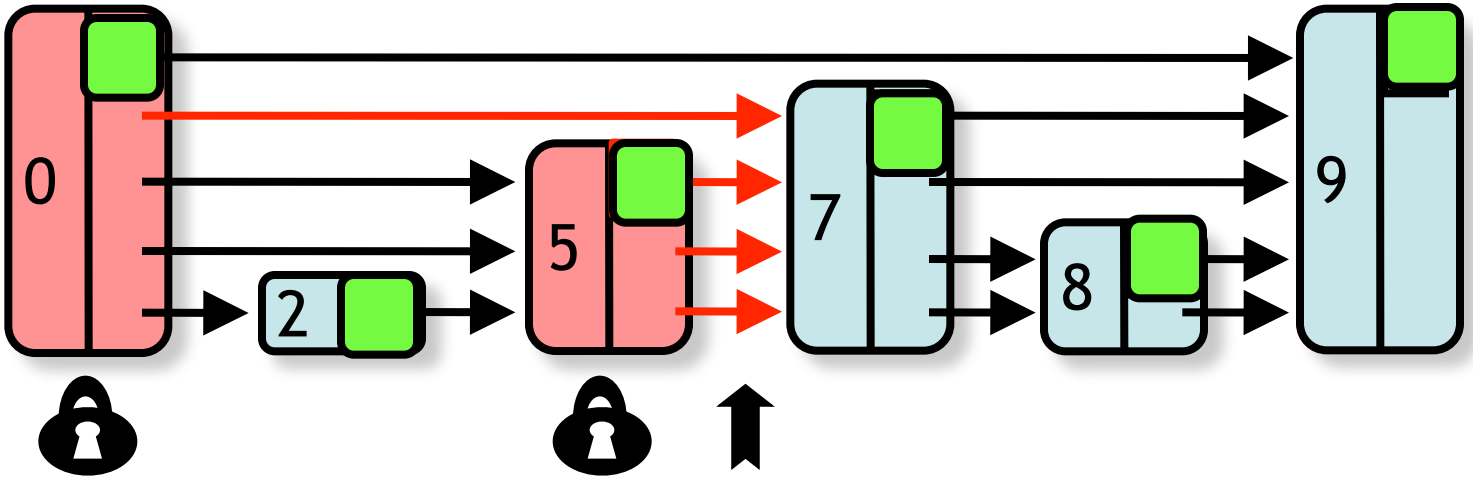




add(6)

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

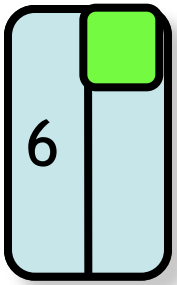


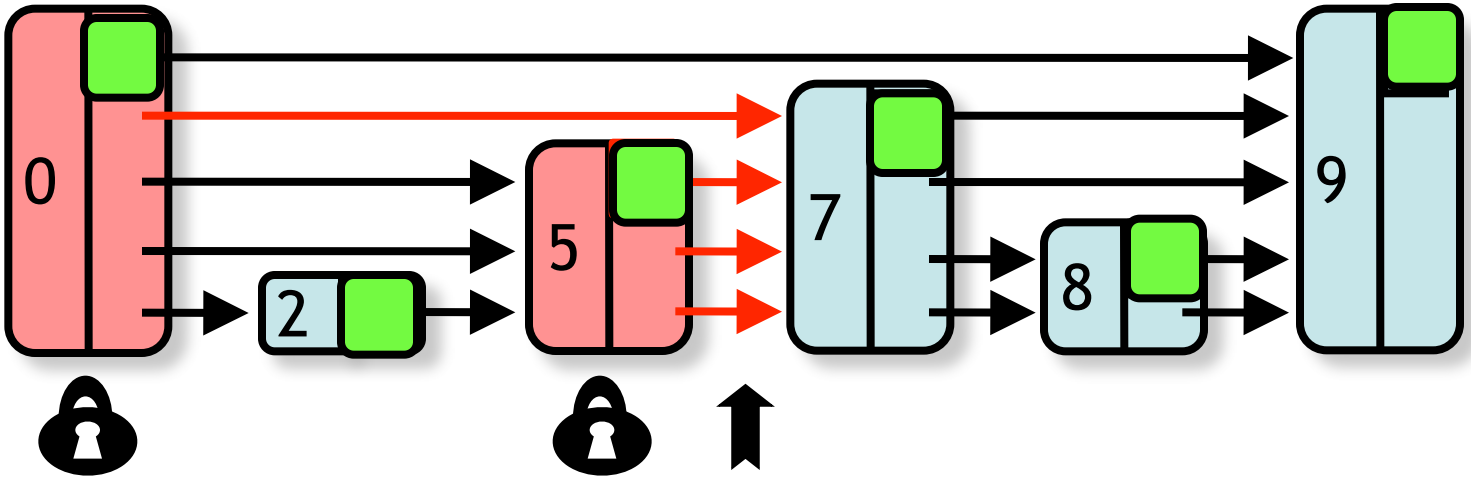


add(6)

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

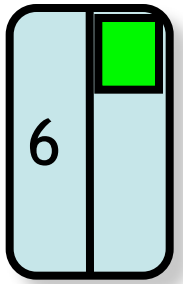
} Optimistic approach

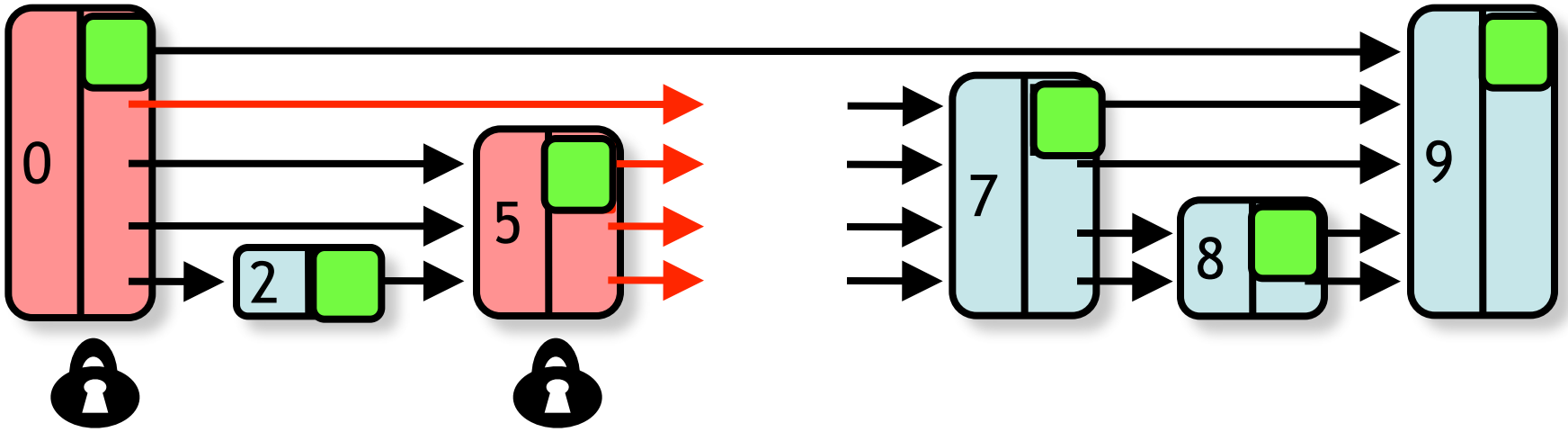




add(6)

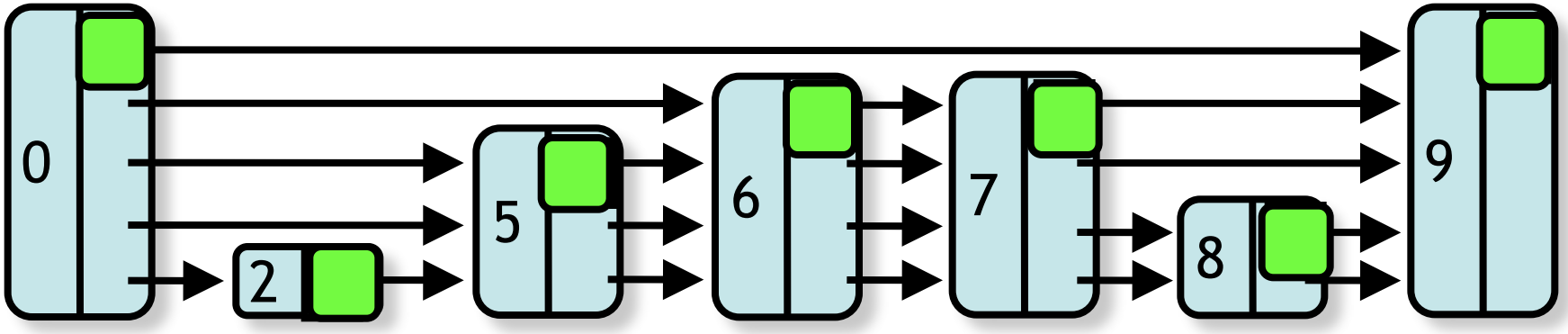
- **find()** predecessors
- Lock them
- Validate
- Splice



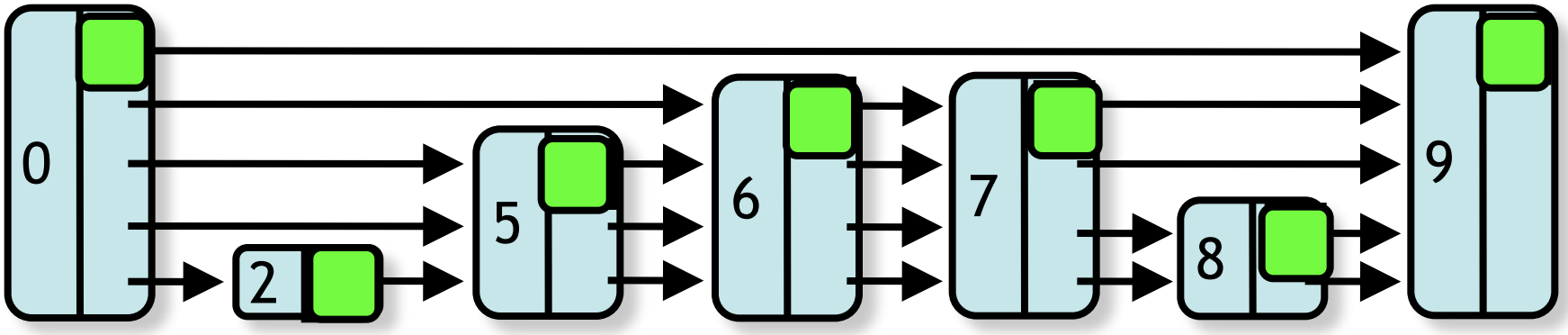


add(6)

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

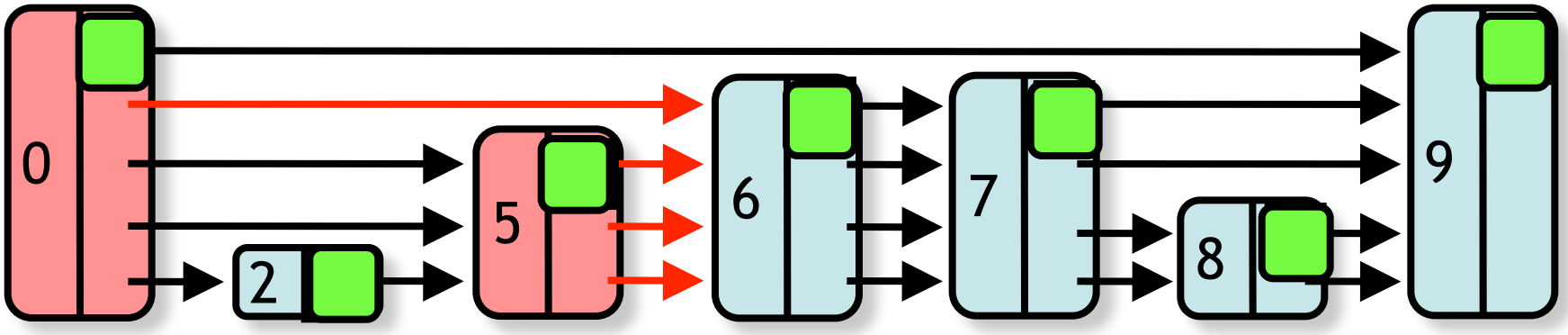


remove(6)



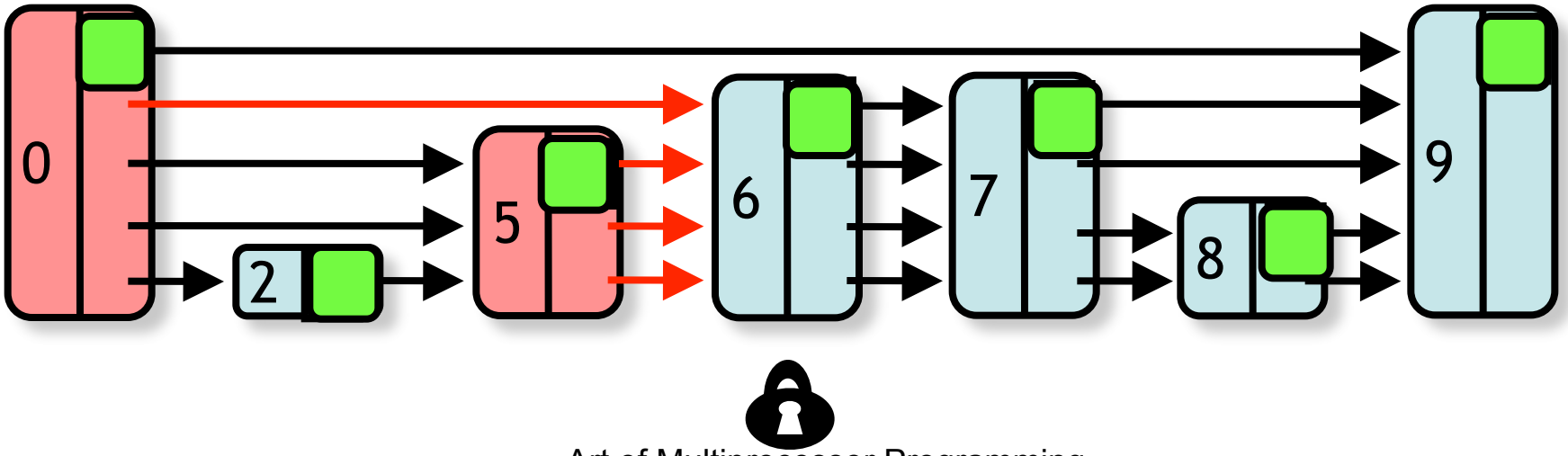
remove(6)

- **find()** predecessors



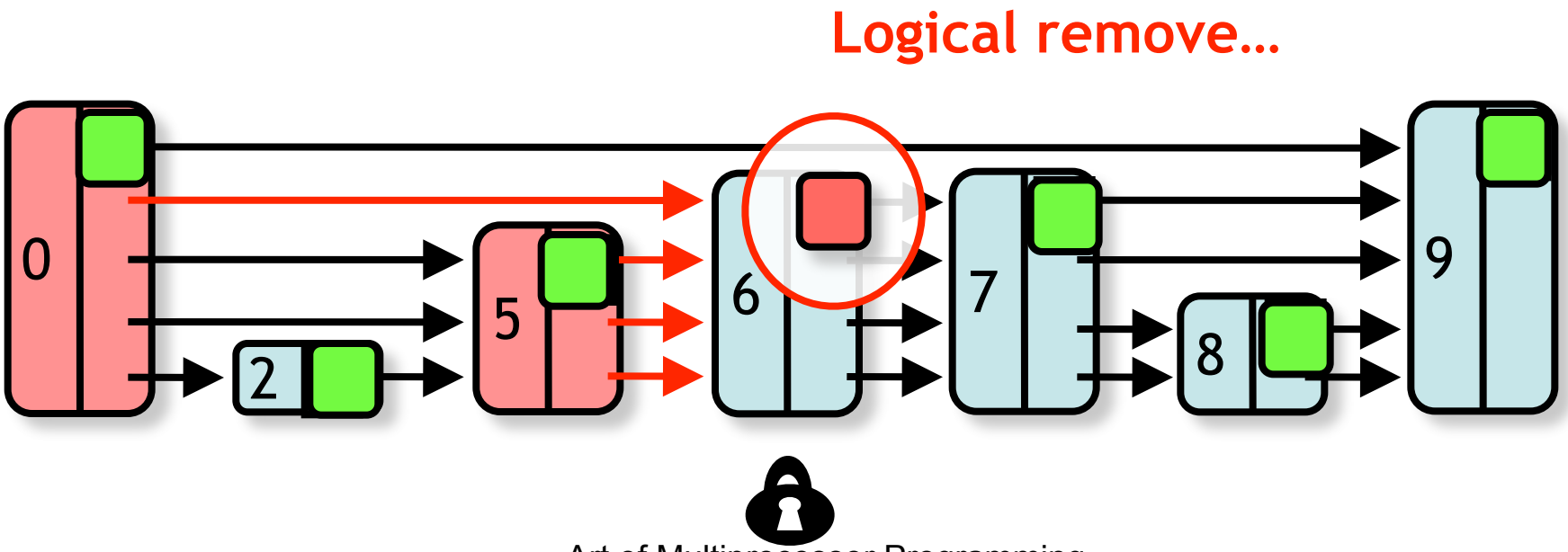
remove(6)

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



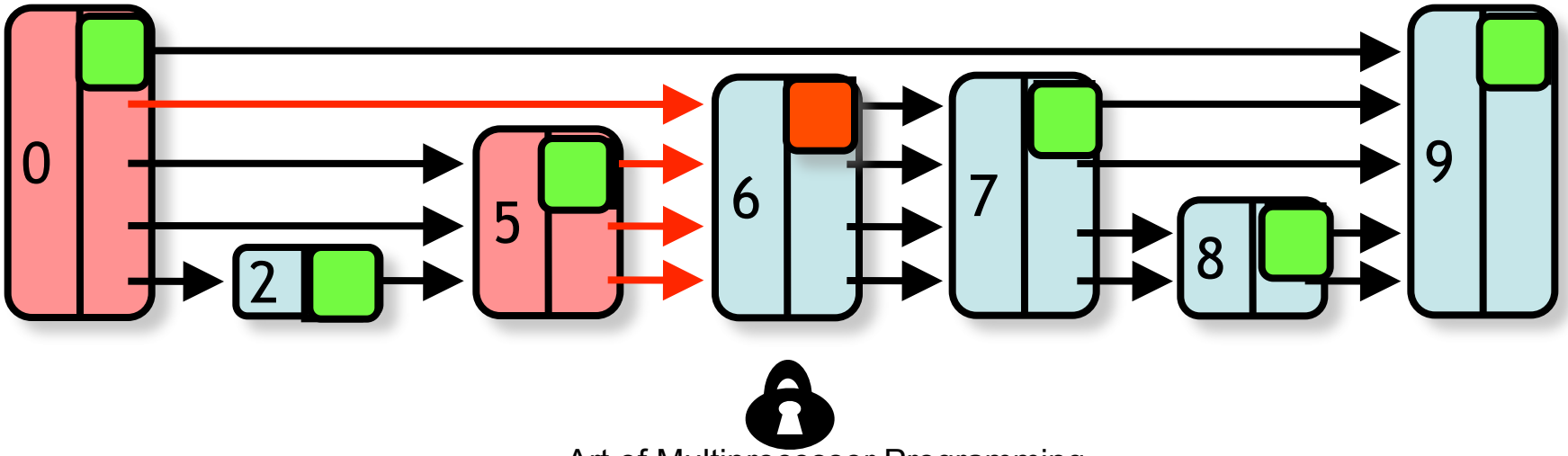
remove(6)

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



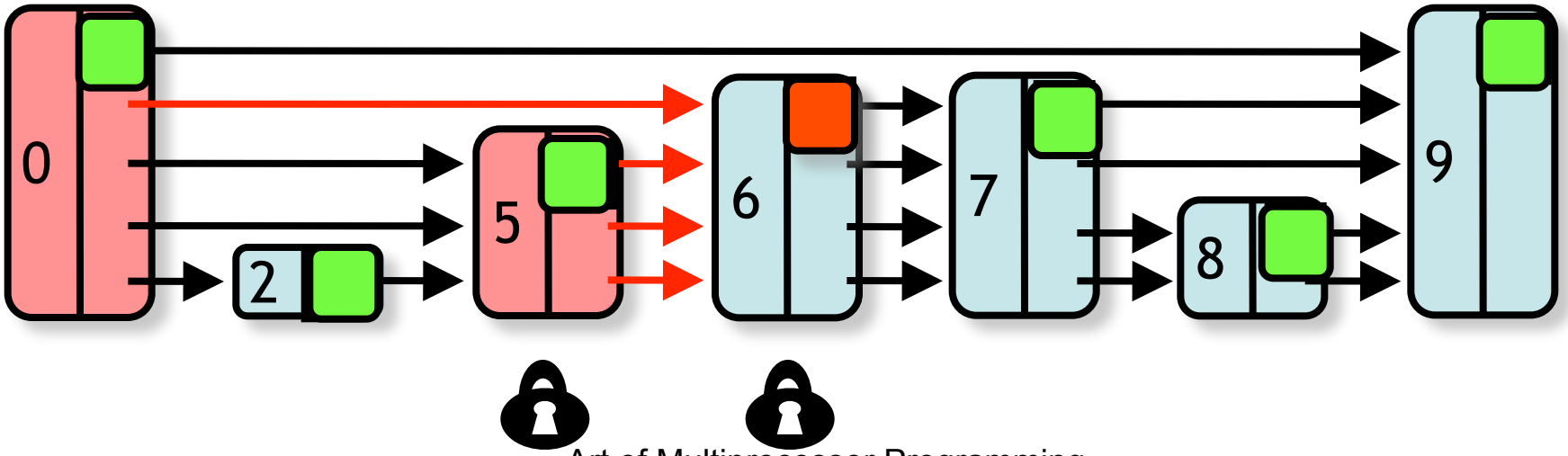
remove(6)

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



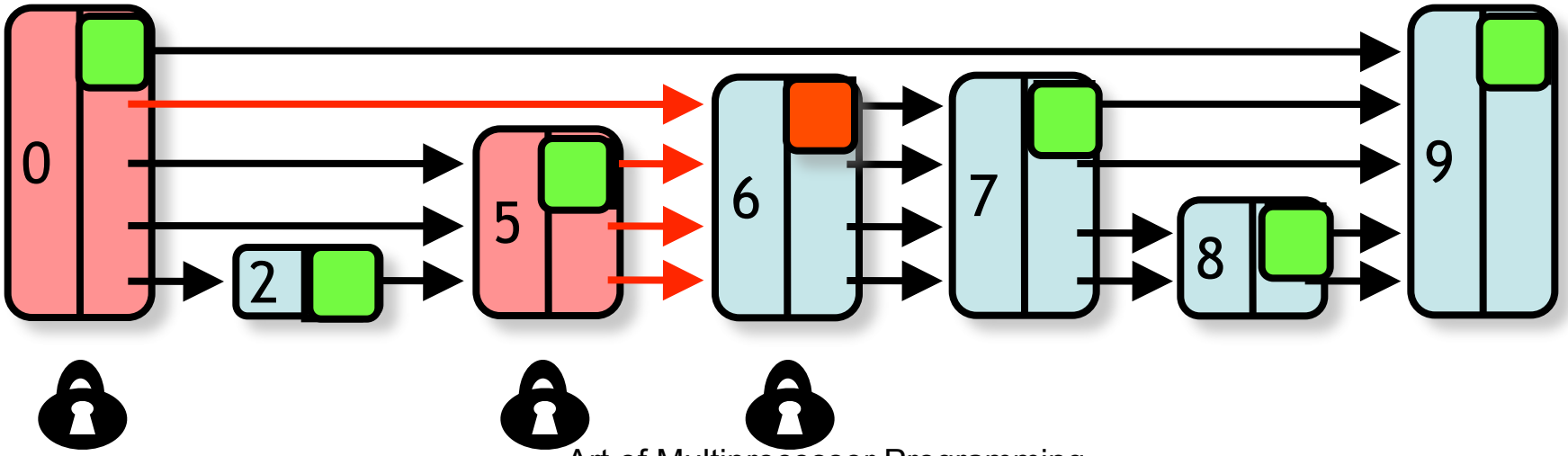
remove(6)

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



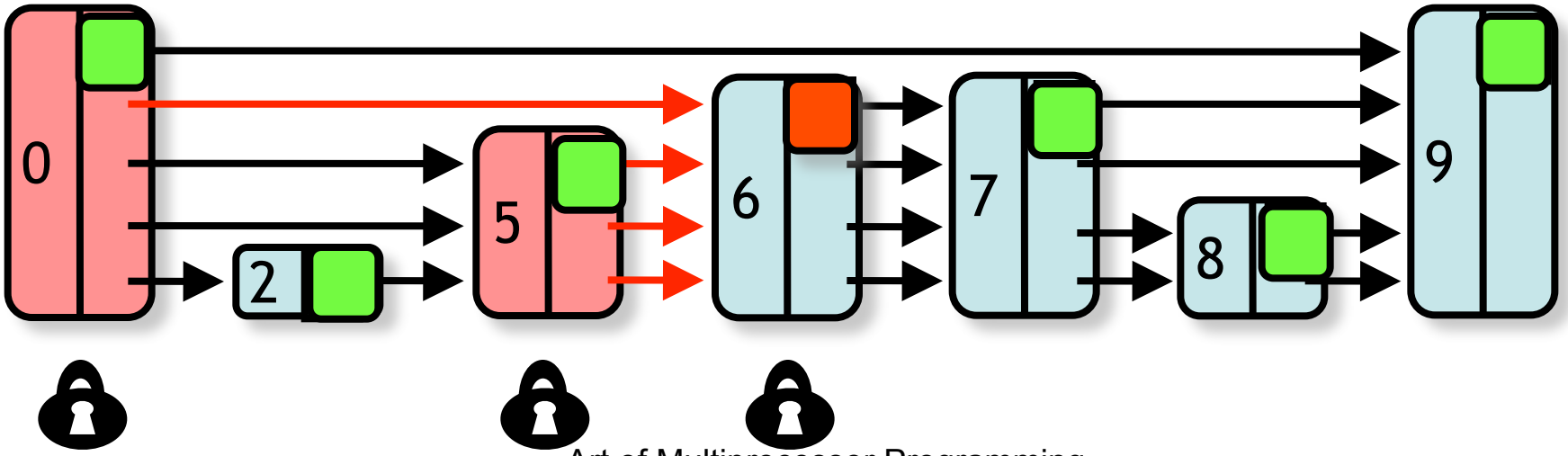
remove(6)

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



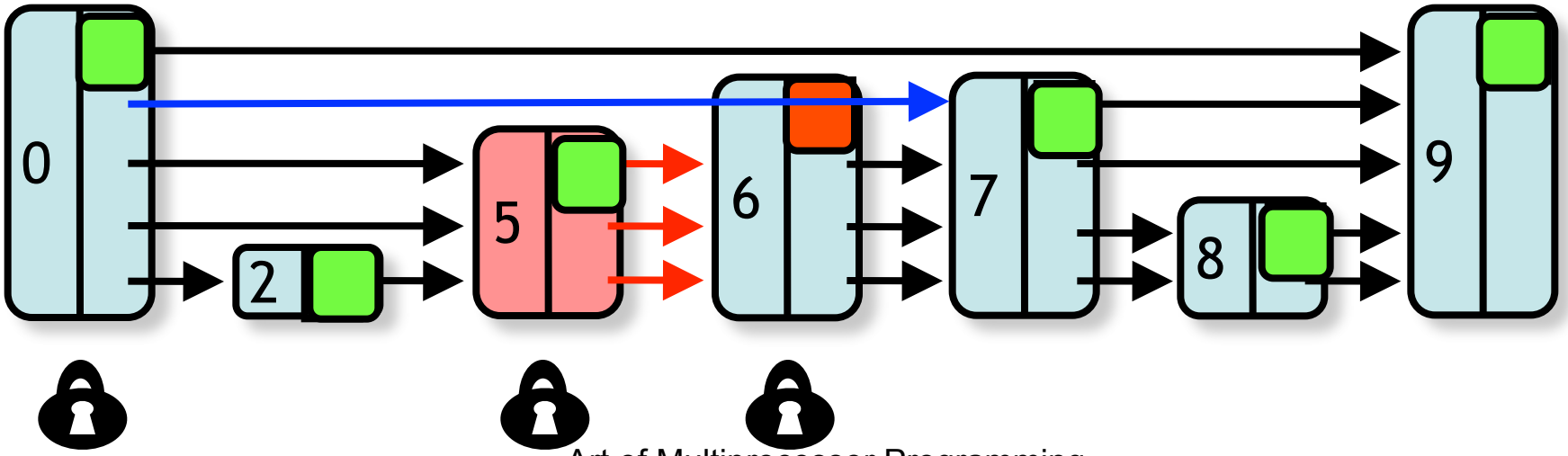
remove(6)

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



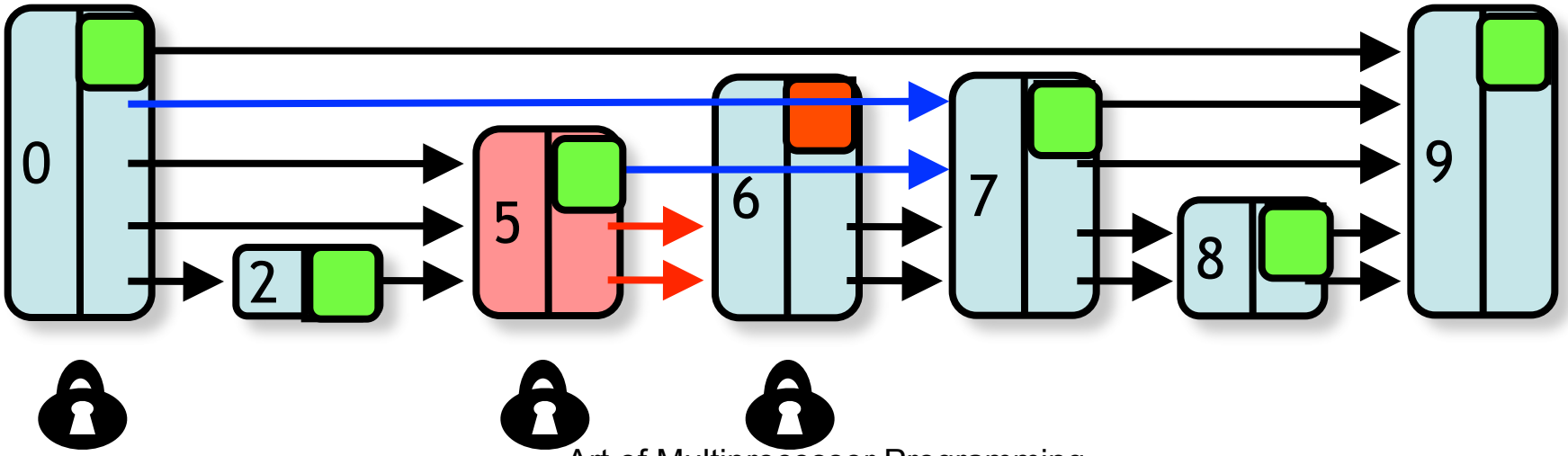
remove(6)

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



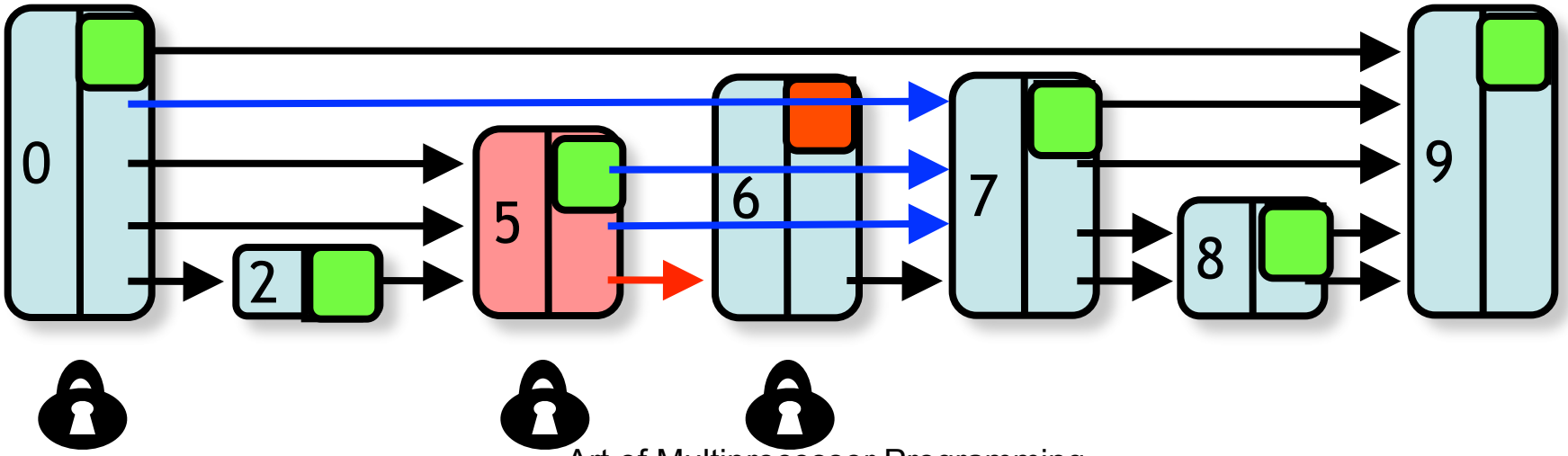
remove(6)

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



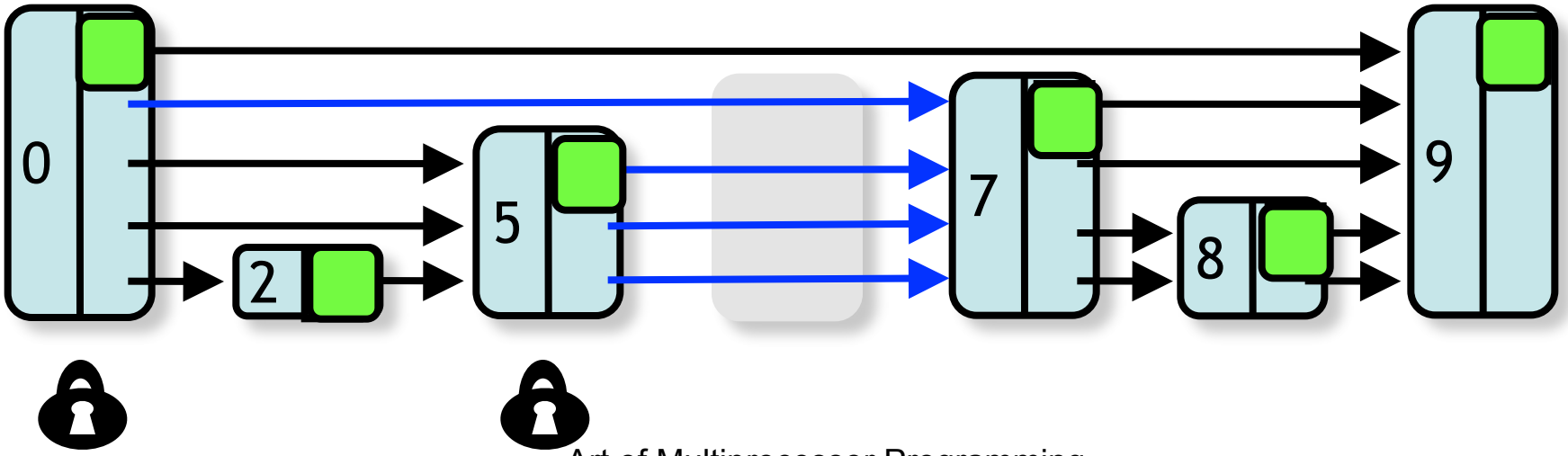
remove(6)

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



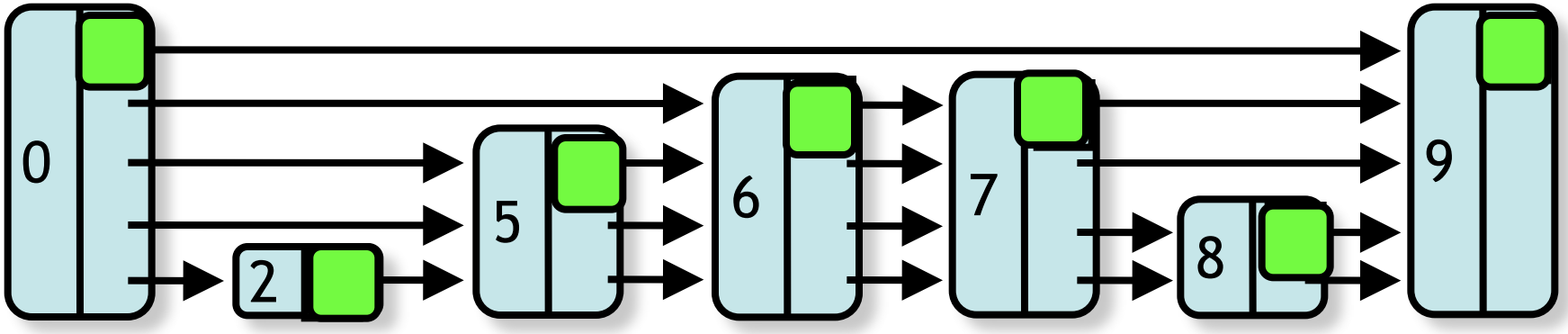
remove(6)

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

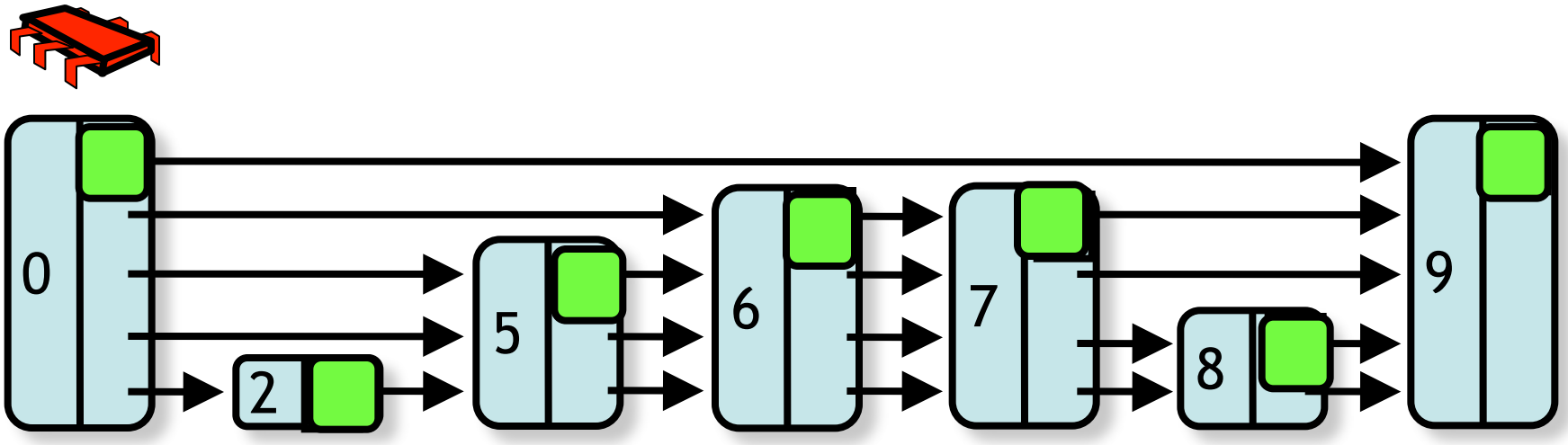


contains(8)

- Find() & not marked

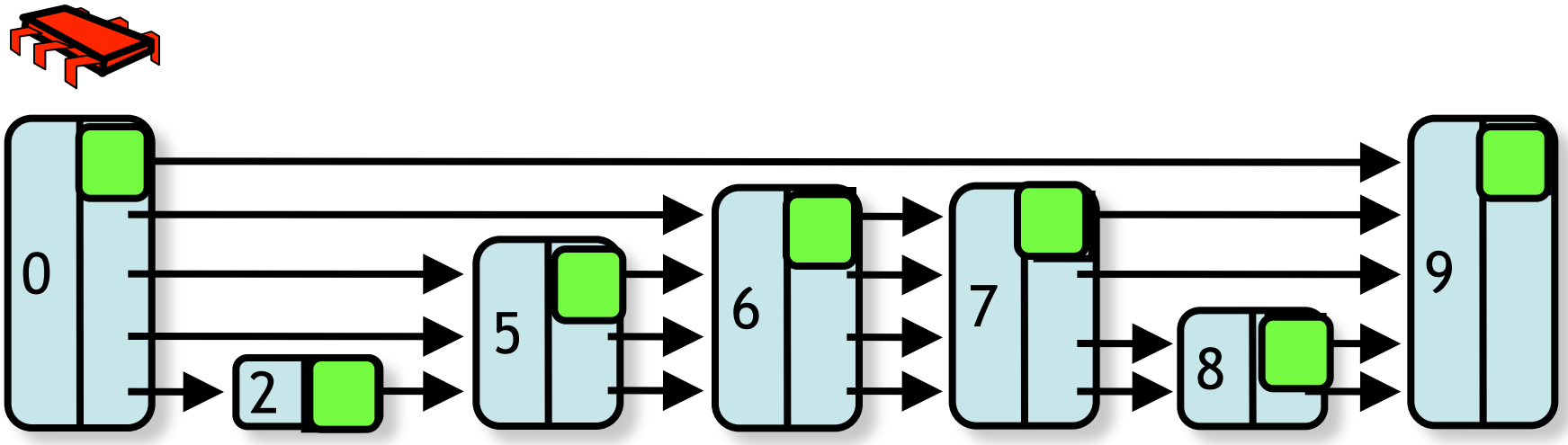


contains(8)



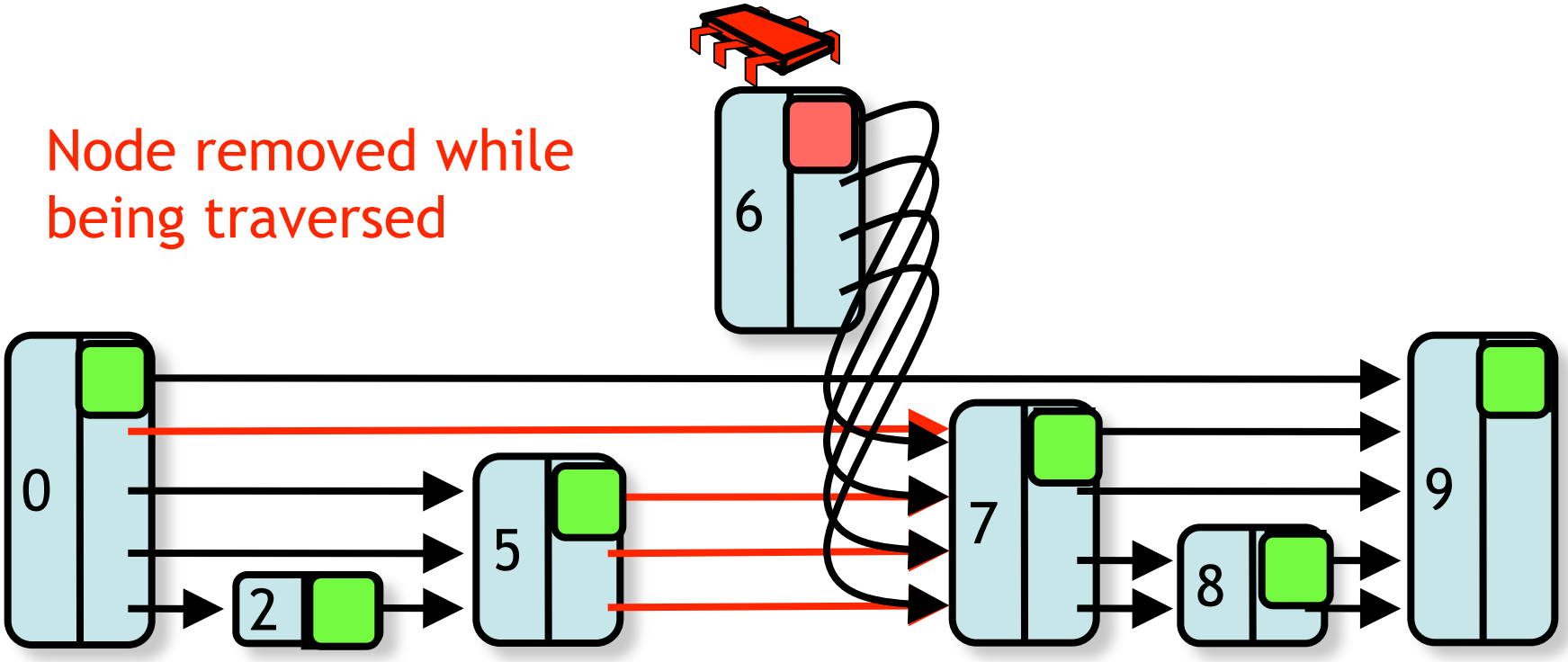
contains(8)

Node 6 removed while traversed



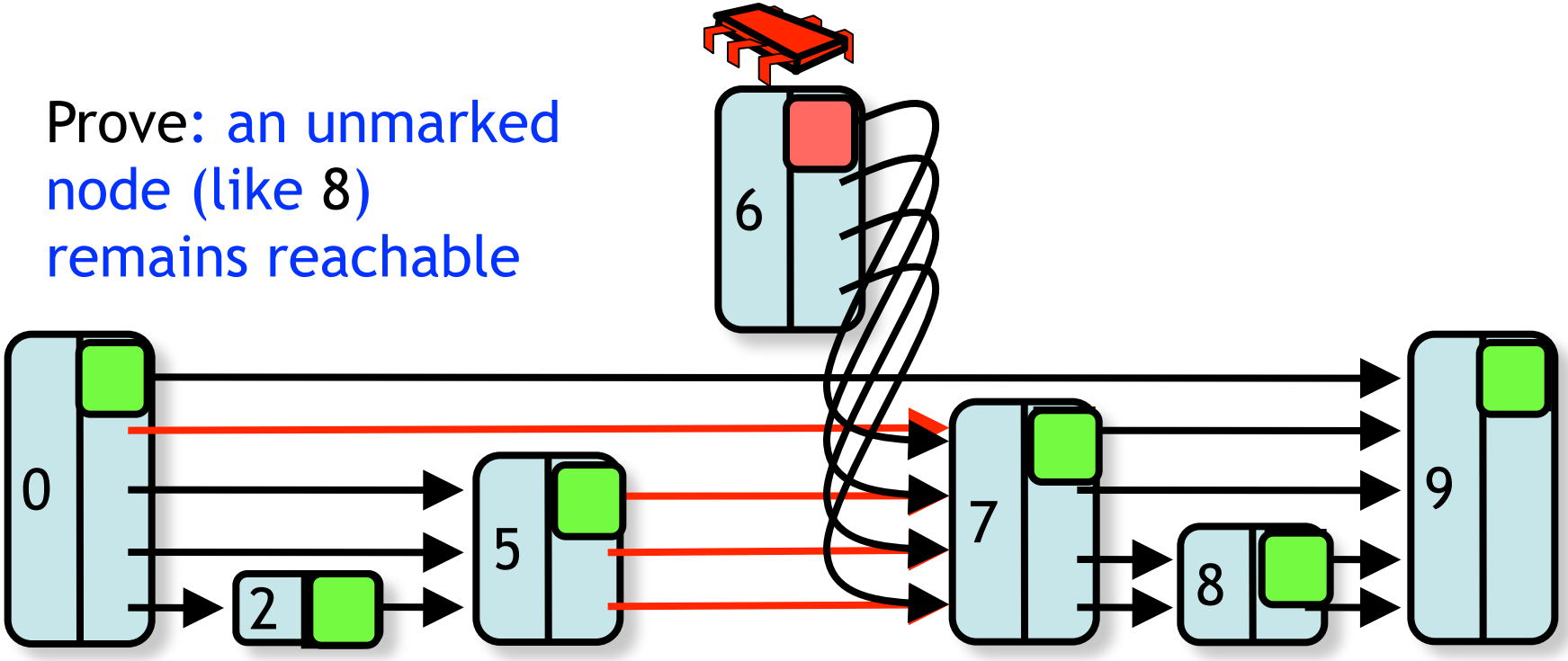
contains(8)

Node removed while being traversed



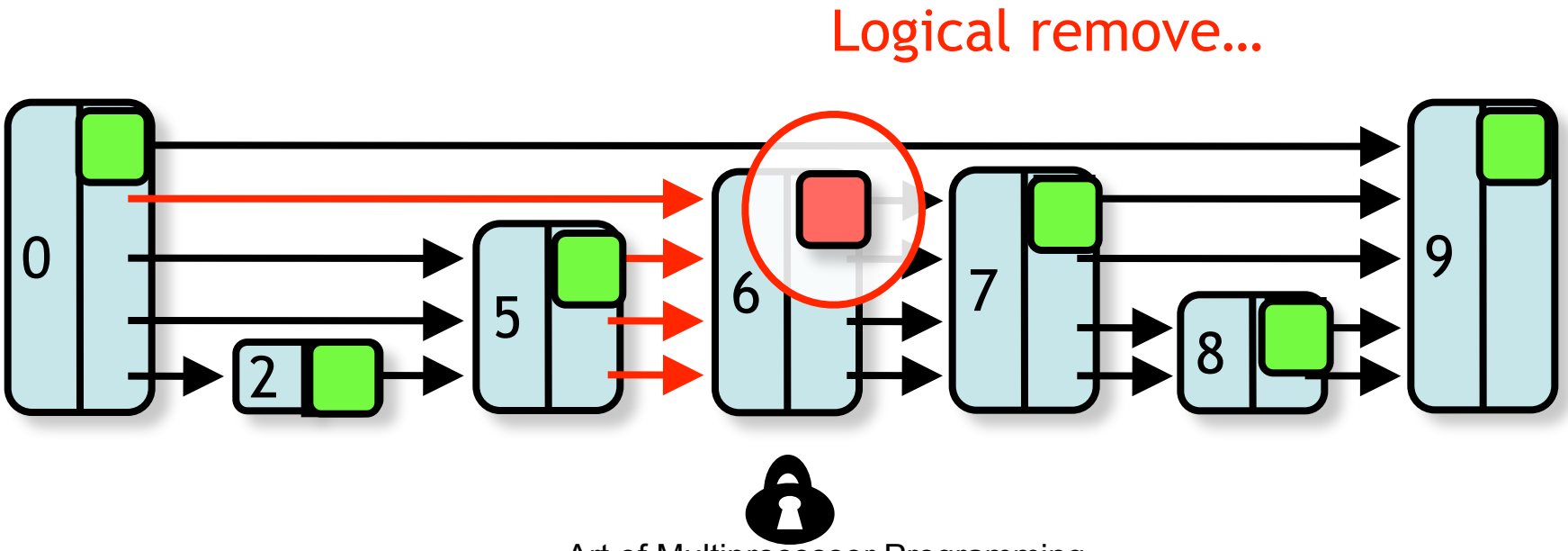
contains(8)

Prove: an unmarked node (like 8) remains reachable



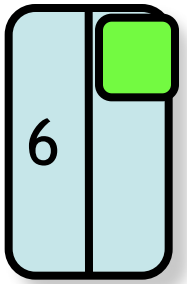
remove(6): Linearization

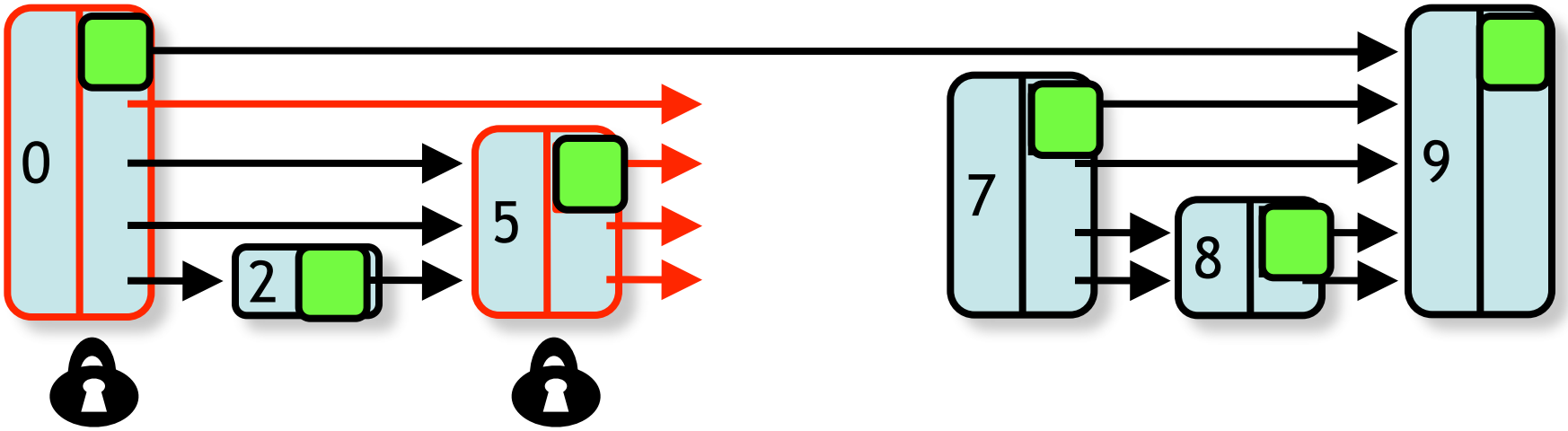
- Successful remove happens when bit is set



Add: Linearization

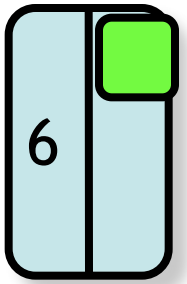
- Successful add() at point when **fully** linked
- Add fullyLinked bit to indicate this
- Bit tested by contains()

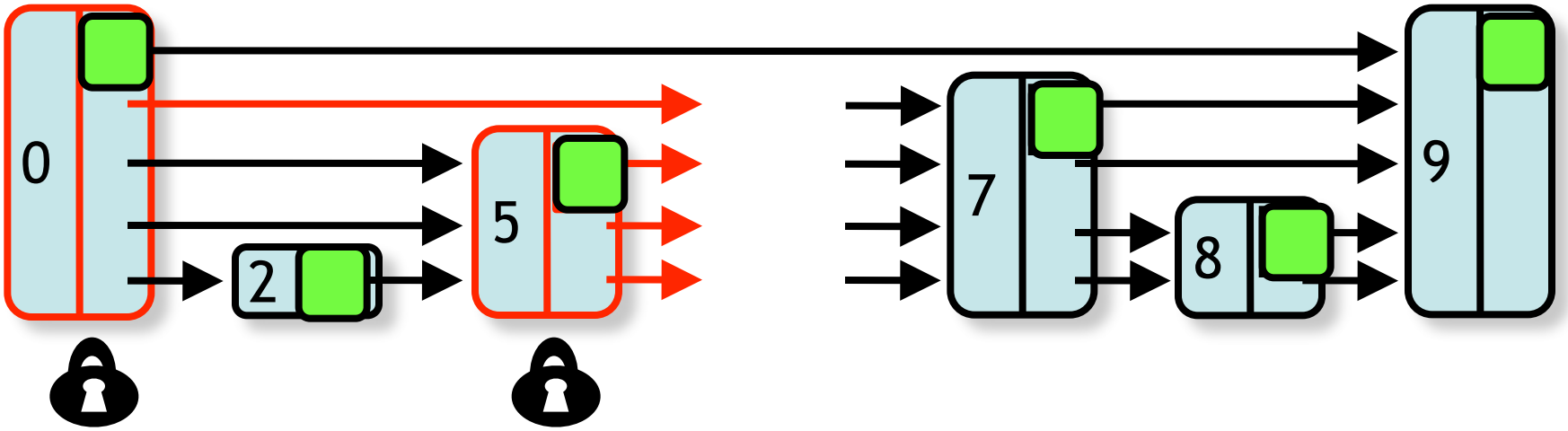




Add: Linearization

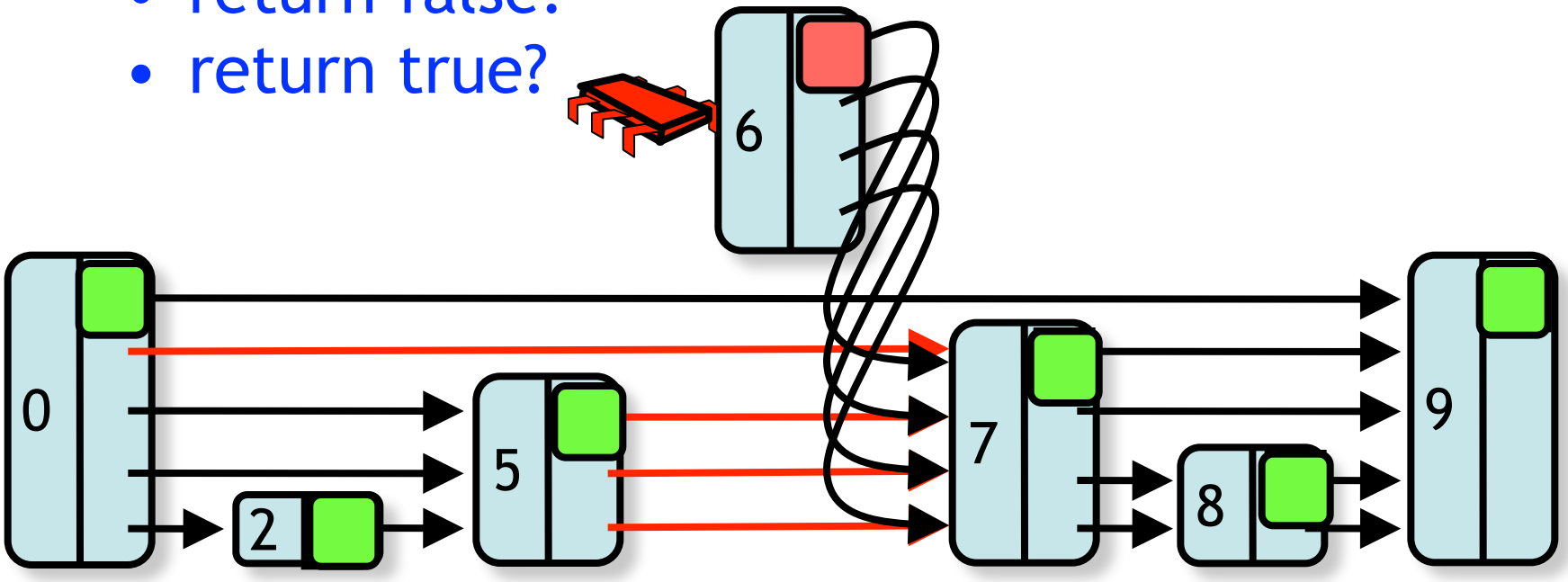
- Successful add() at point when **fully** linked
- Add fullyLinked bit to indicate this
- Bit tested by contains()





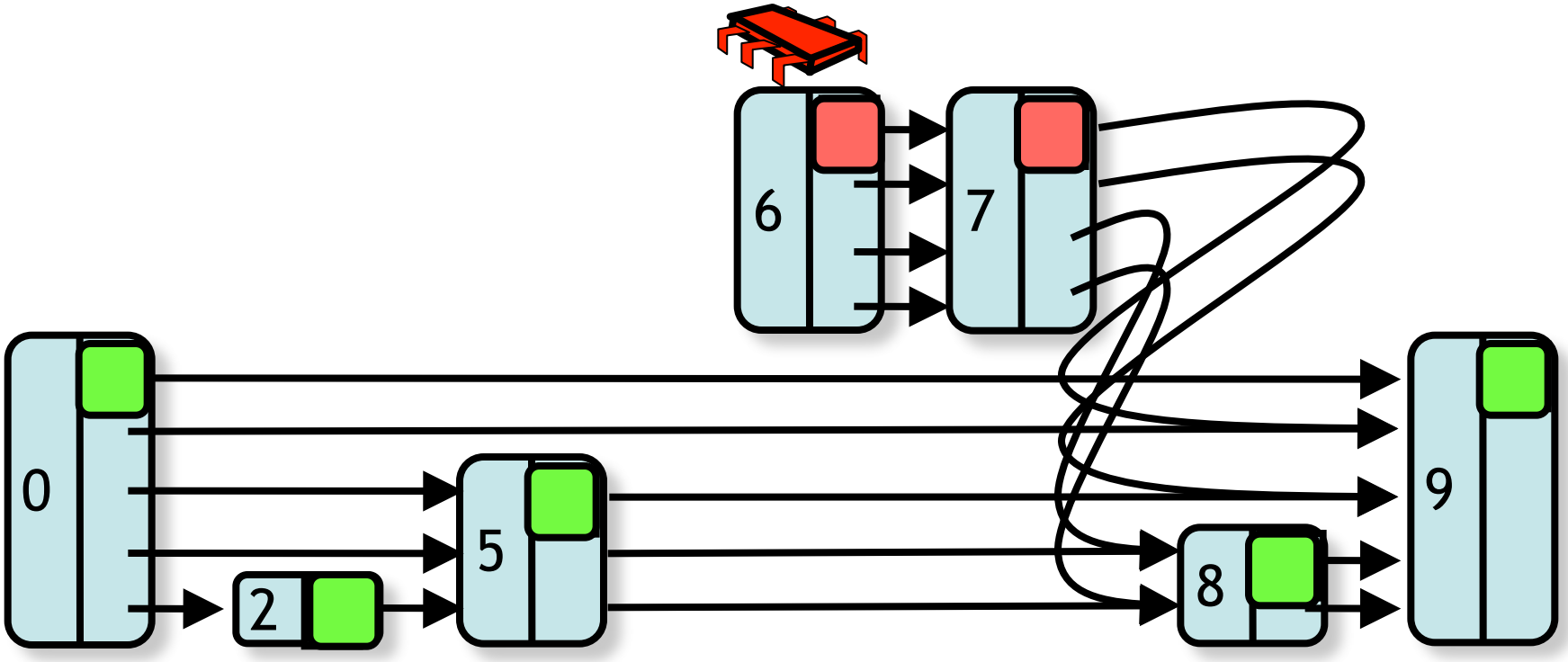
contains(7): Linearization

- When not fully-linked unmarked node found
  - pause while fullyLinked bit unset?
  - return false?
  - return true?



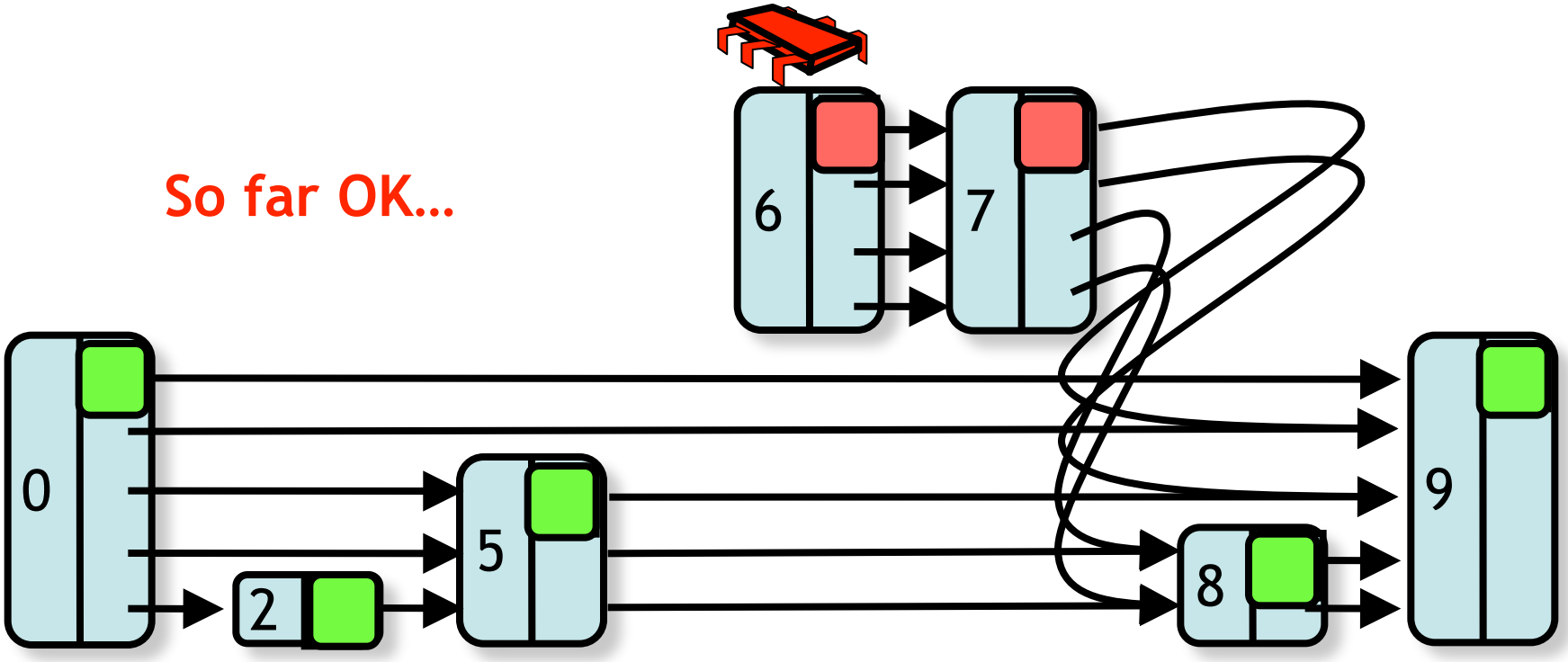
contains(7): Linearization

- When do we linearize unsuccessful Search?



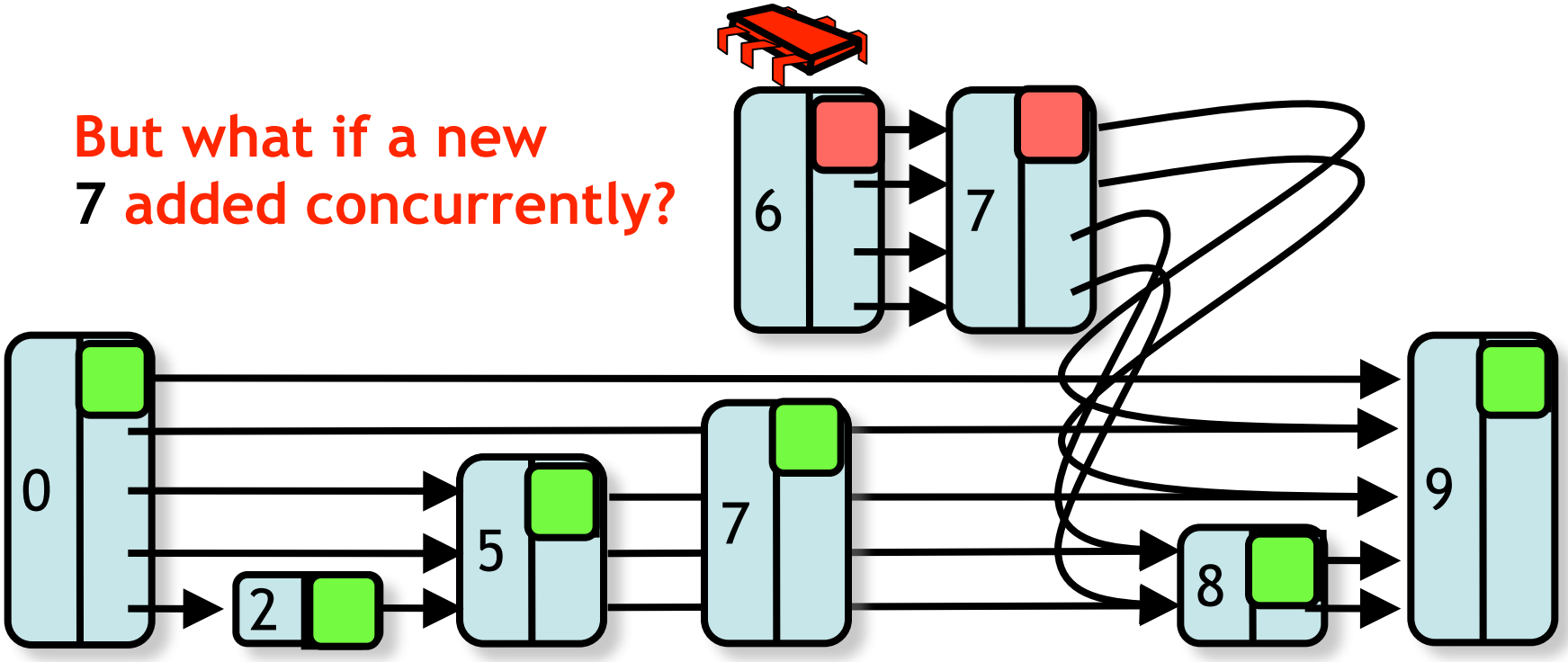
contains(7): Linearization

- When do we linearize unsuccessful Search?



contains(7): Linearization

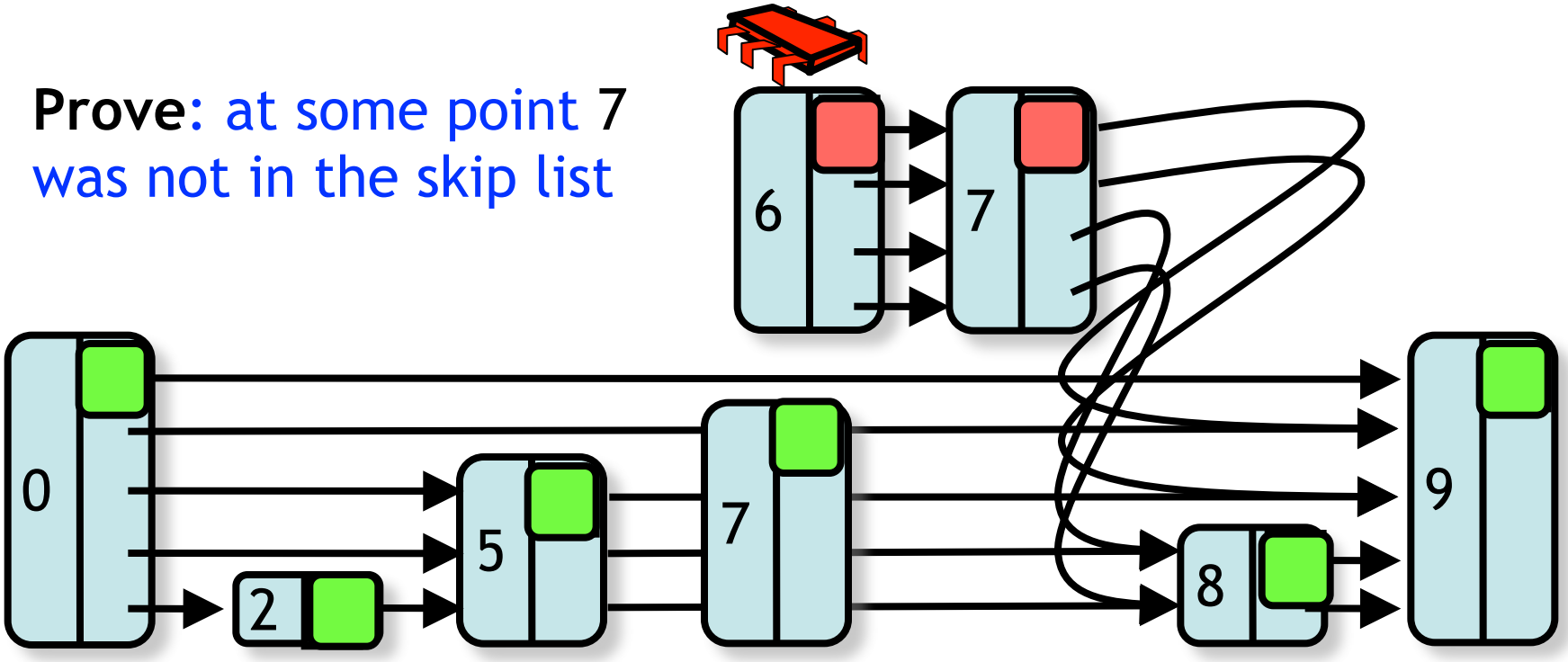
- When do we linearize unsuccessful Search?



contains(7): Linearization

- When do we linearize unsuccessful Search?

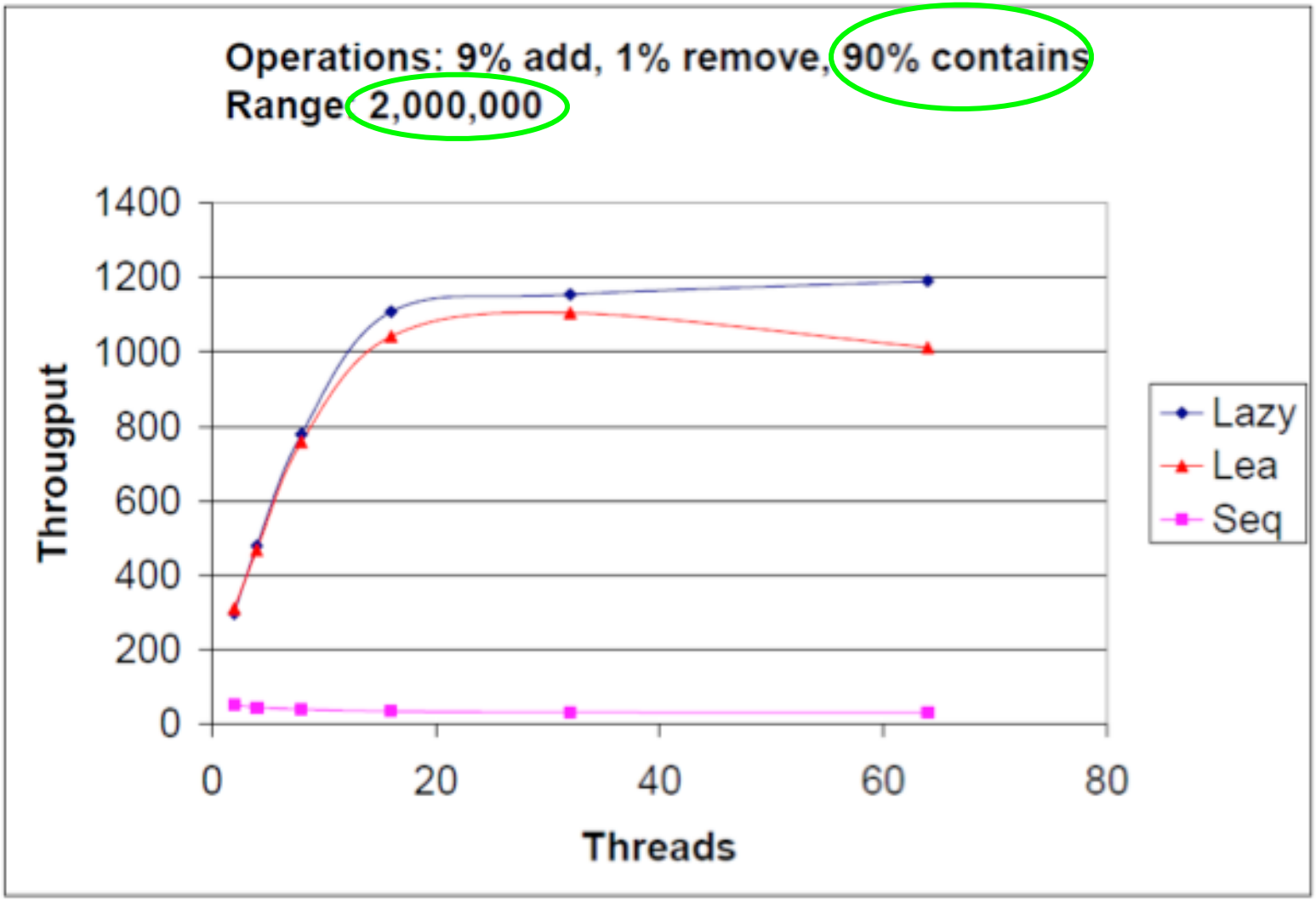
Prove: at some point 7 was not in the skip list



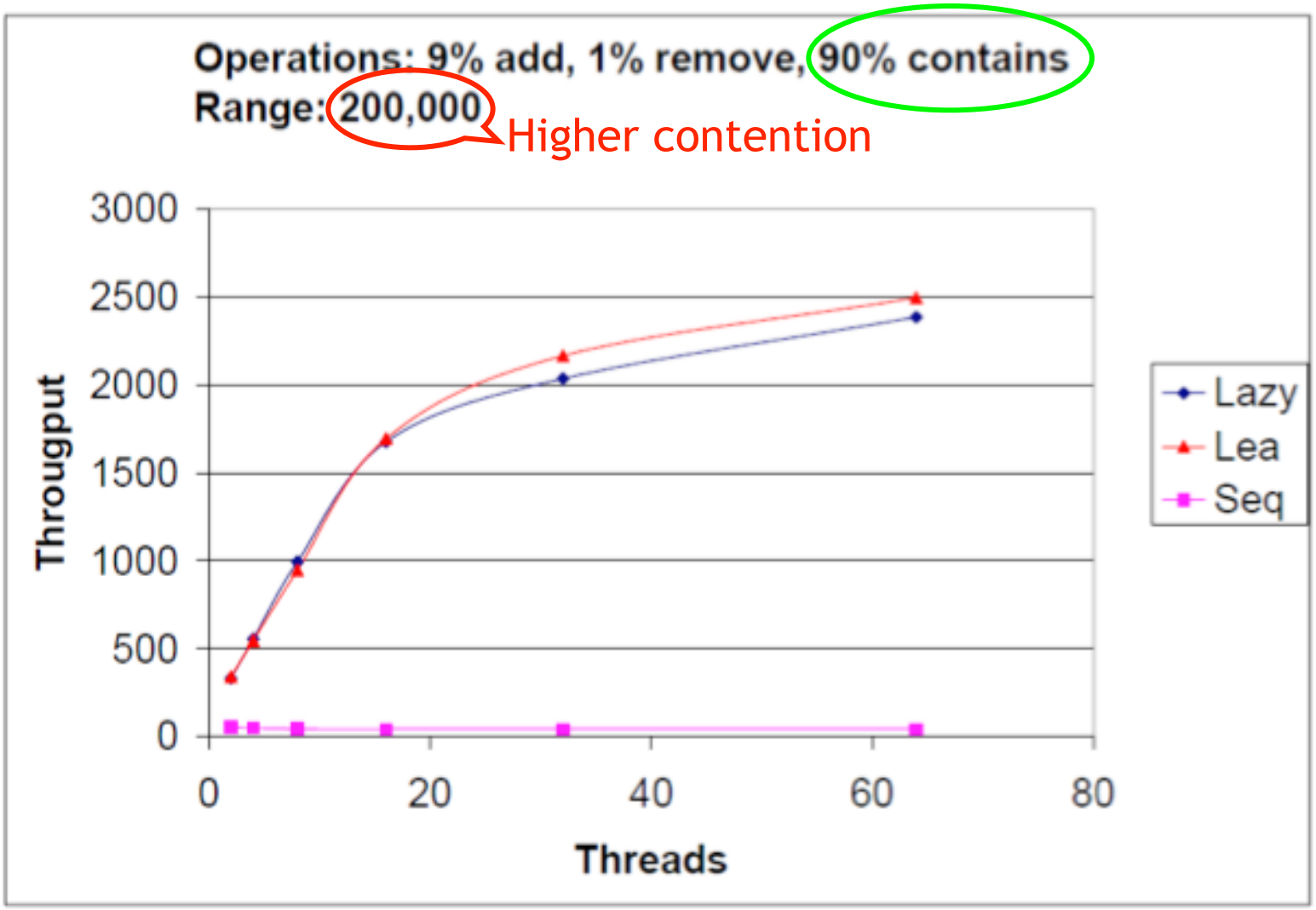
A Simple Experiment

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

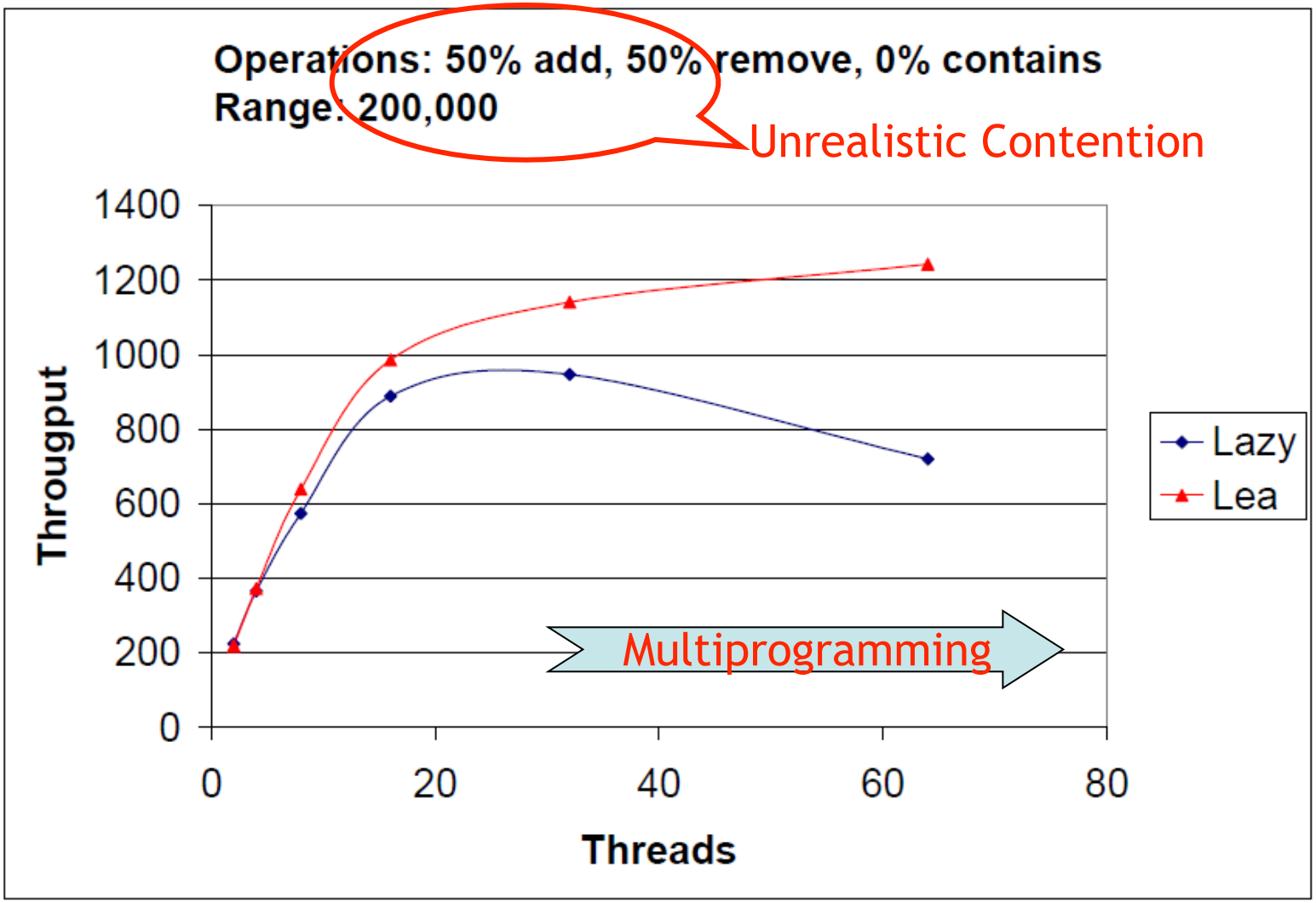
Lazy Skip List: Performance



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