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

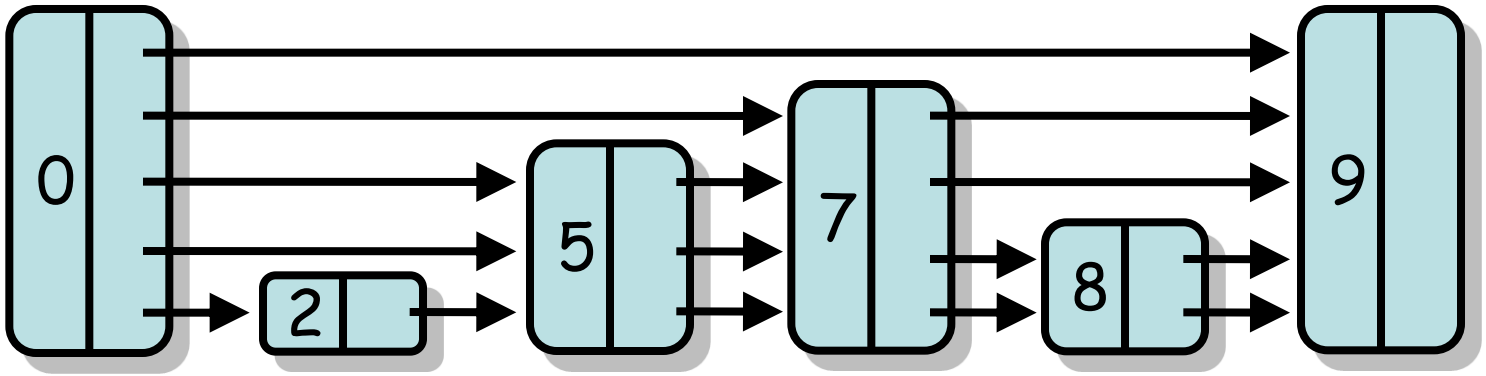
- Typically high % of contains() calls
- Many fewer add() calls
- And even fewer remove() calls
  - 90% contains()
  - 9% add()
  - 1% remove()
- Folklore?
  - Yes but probably mostly true

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

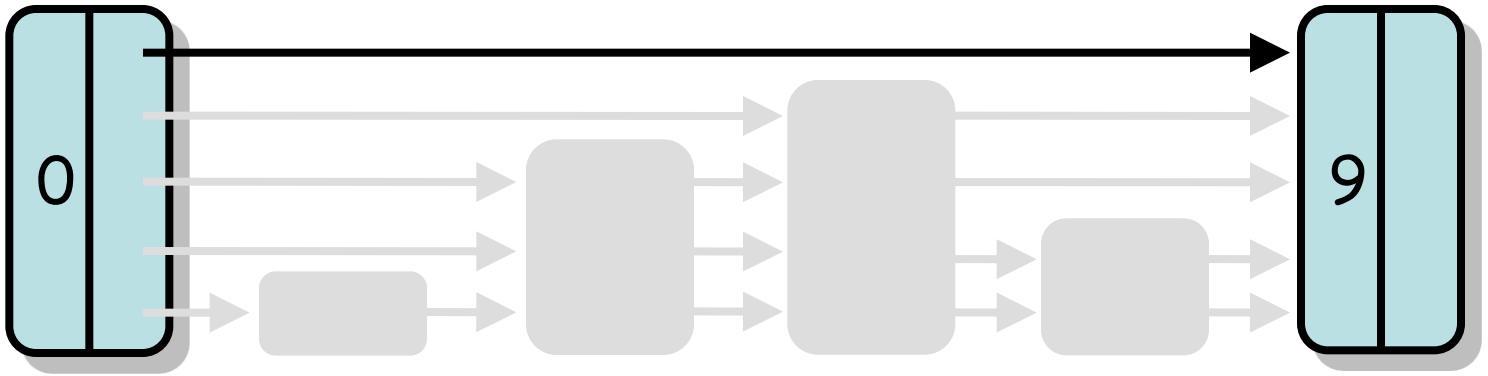
Skip Lists

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



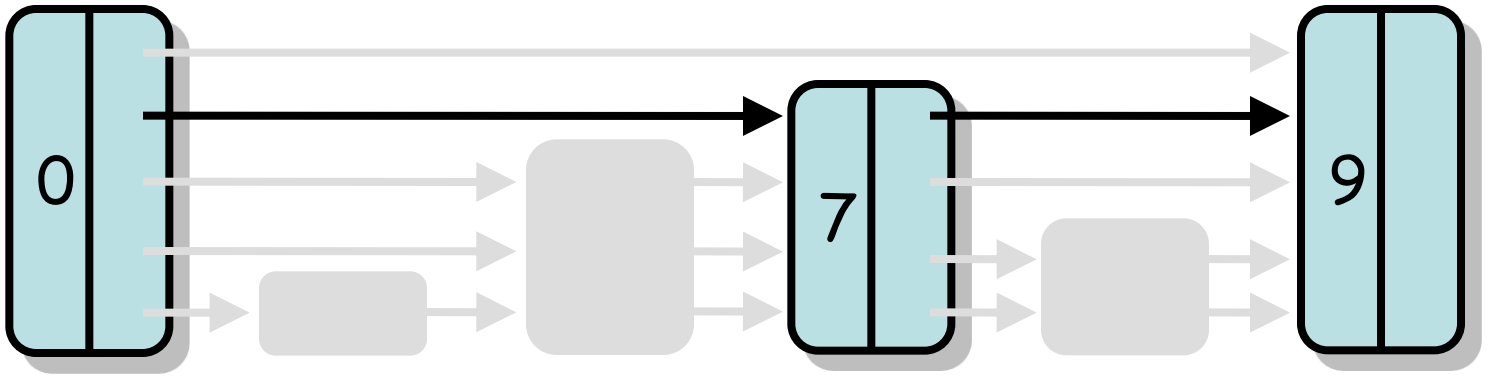
Skip List Property

- Each layer is sublist of lower-levels



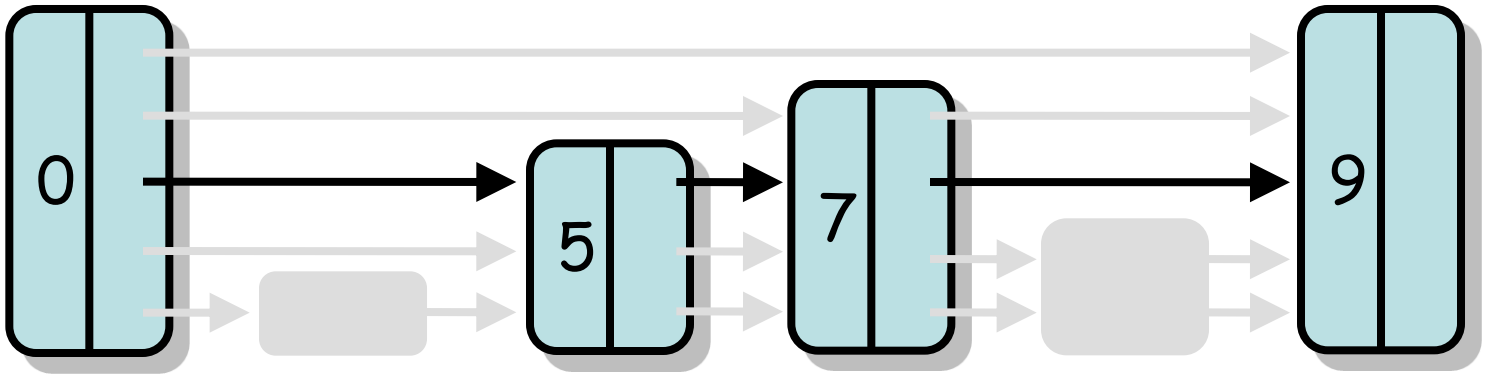
Skip List Property

- Each layer is sublist of lower-levels



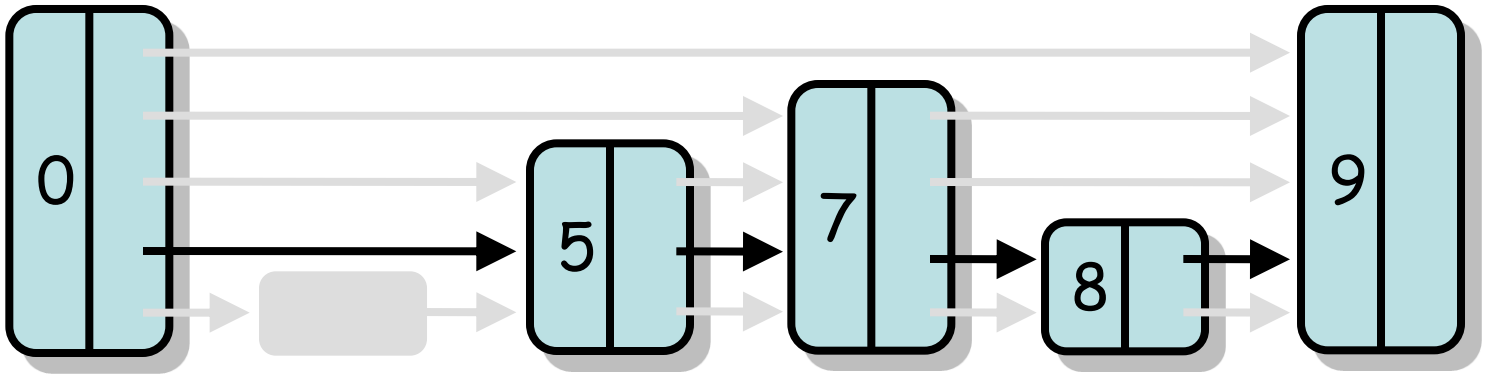
Skip List Property

- Each layer is sublist of lower-levels



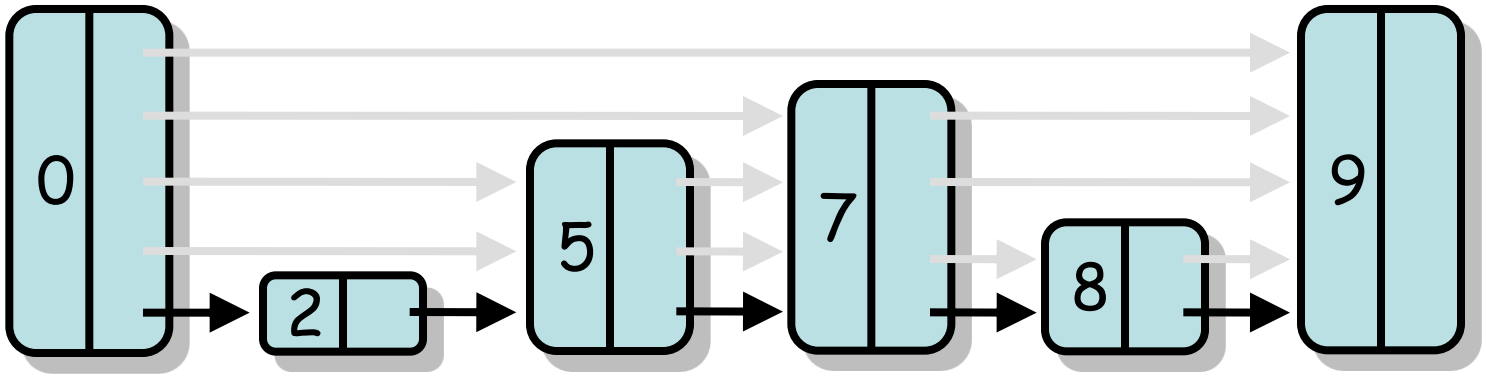
Skip List Property

- Each layer is sublist of lower-levels



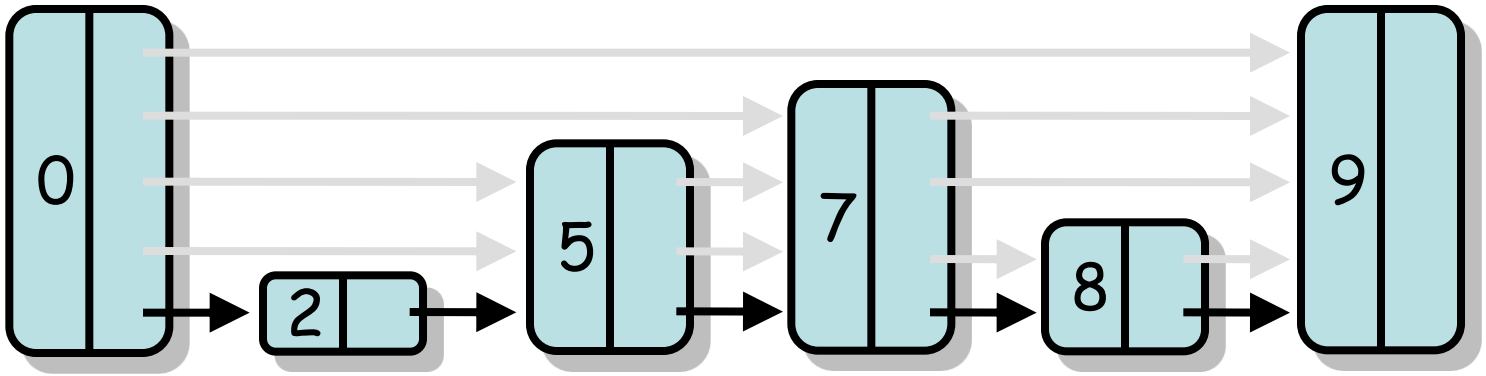
Skip List Property

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

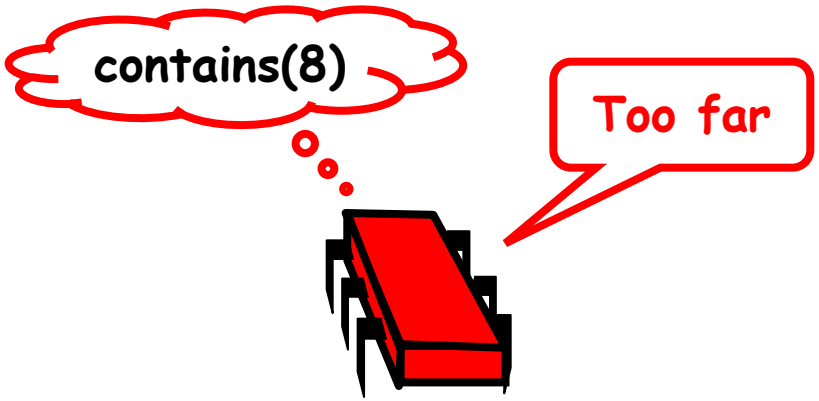


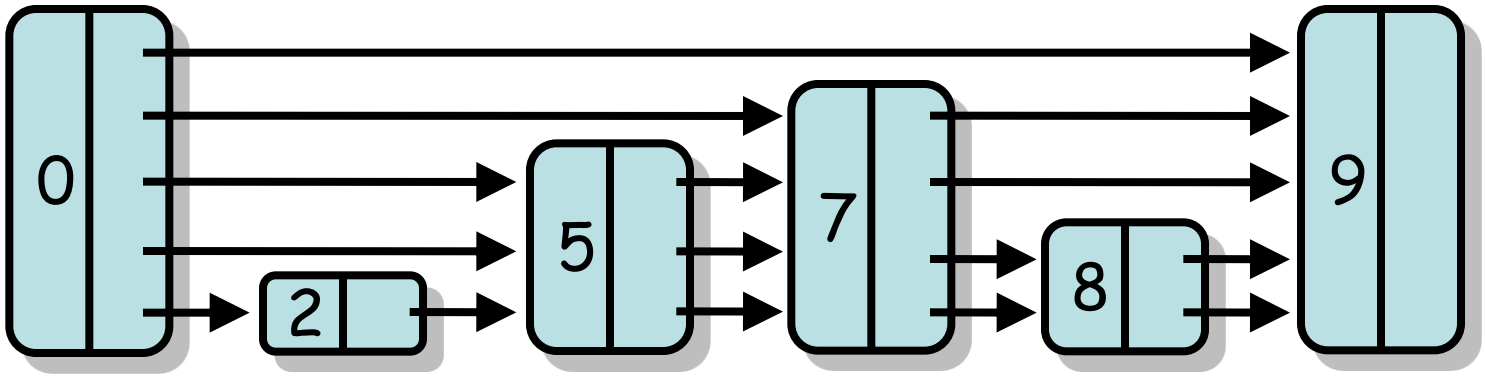
Skip List Property

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

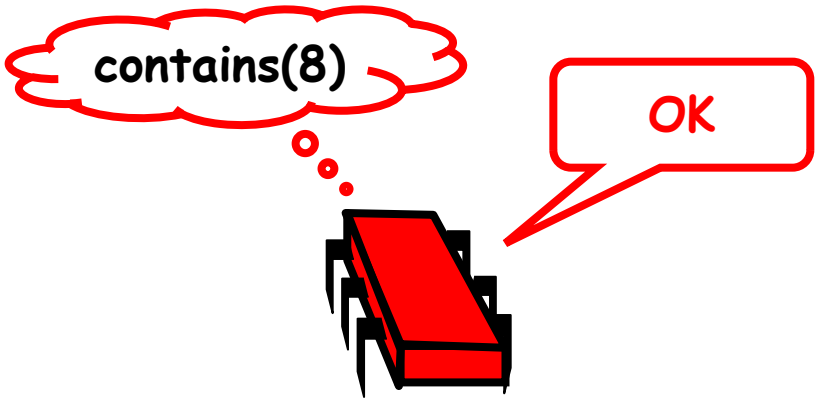


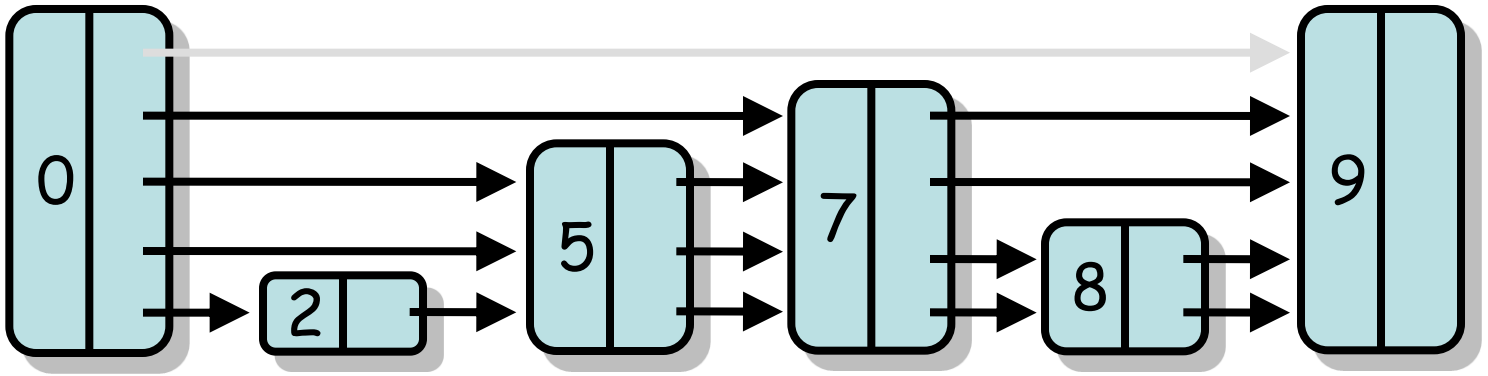
Search



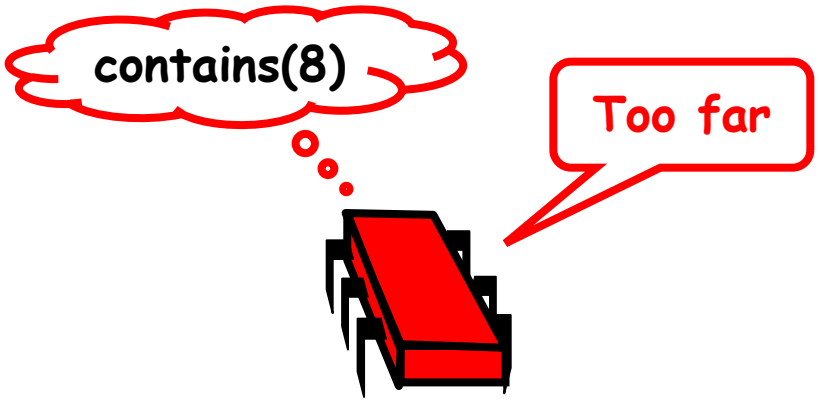


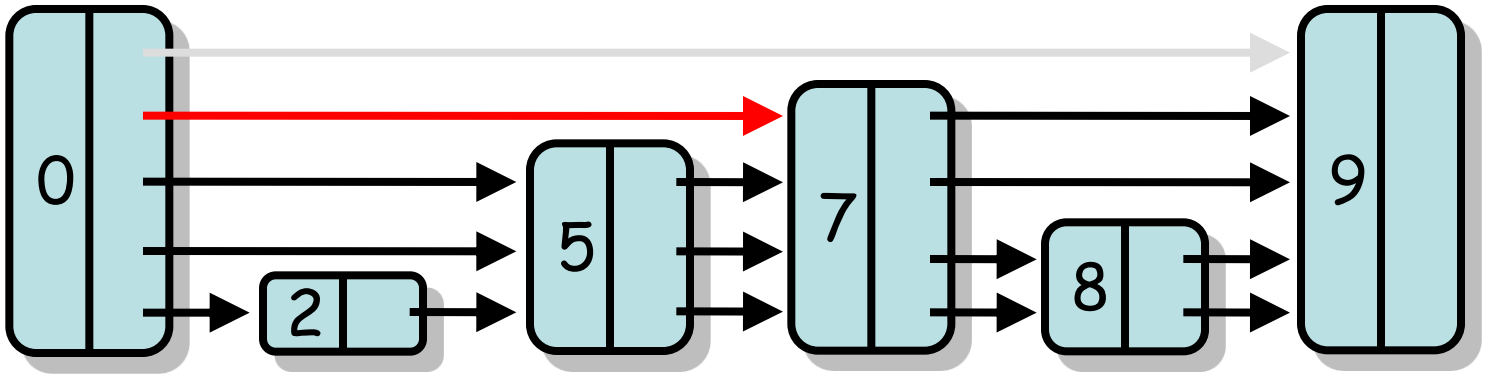
Search



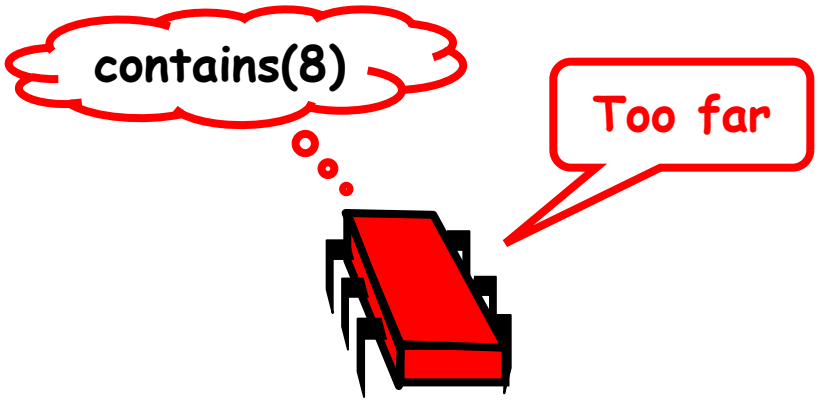


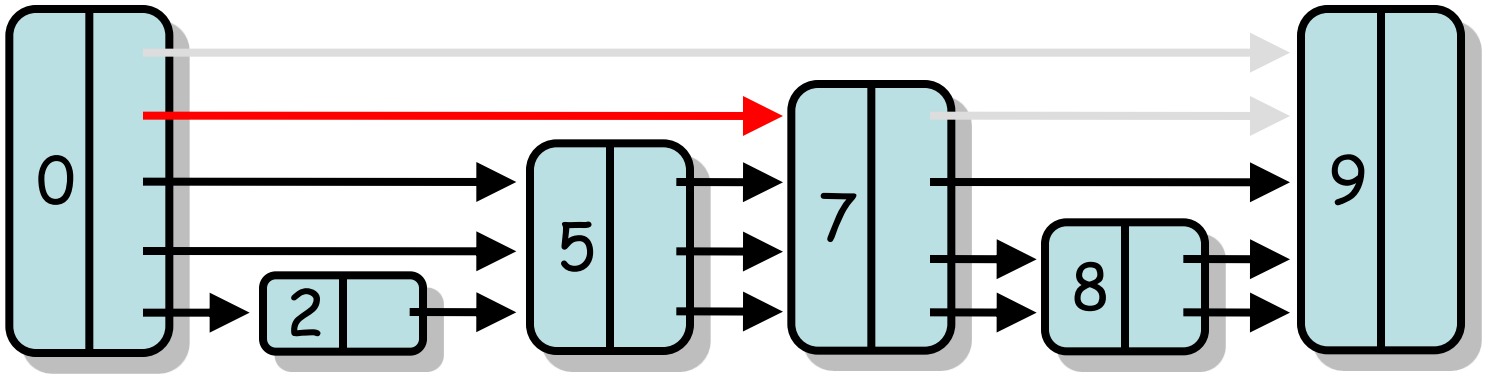
Search



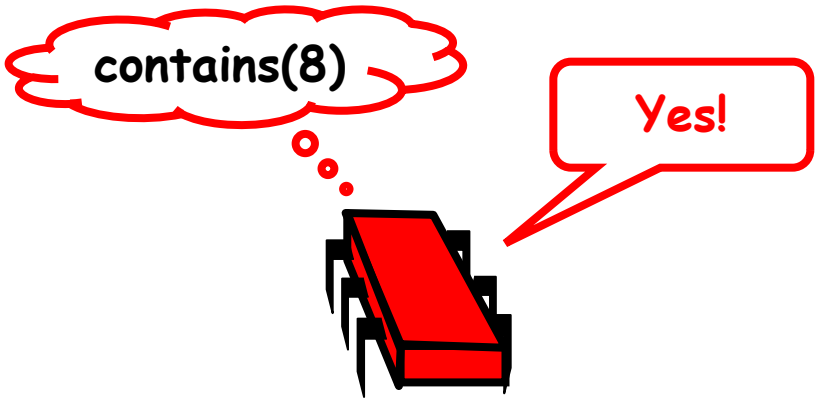


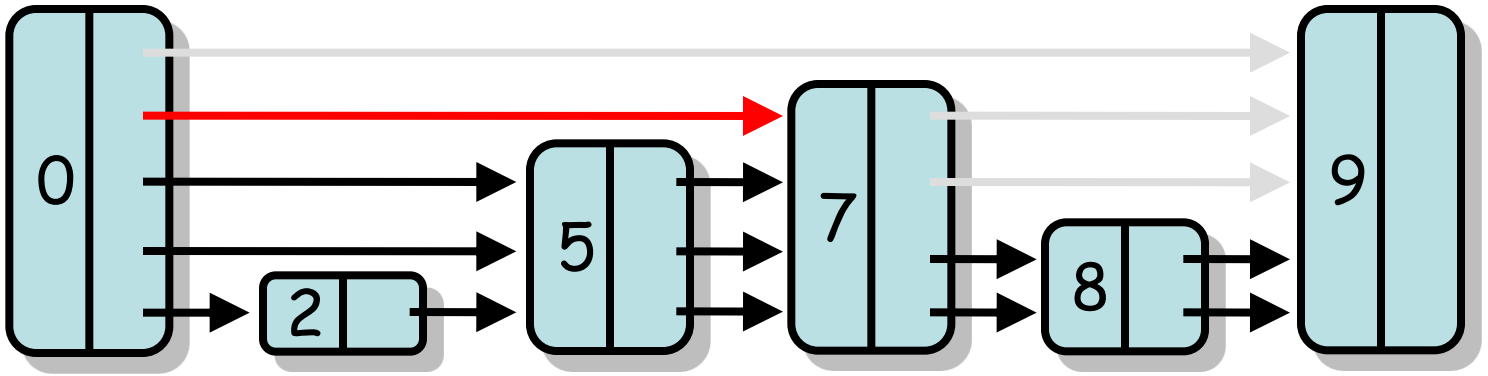
Search



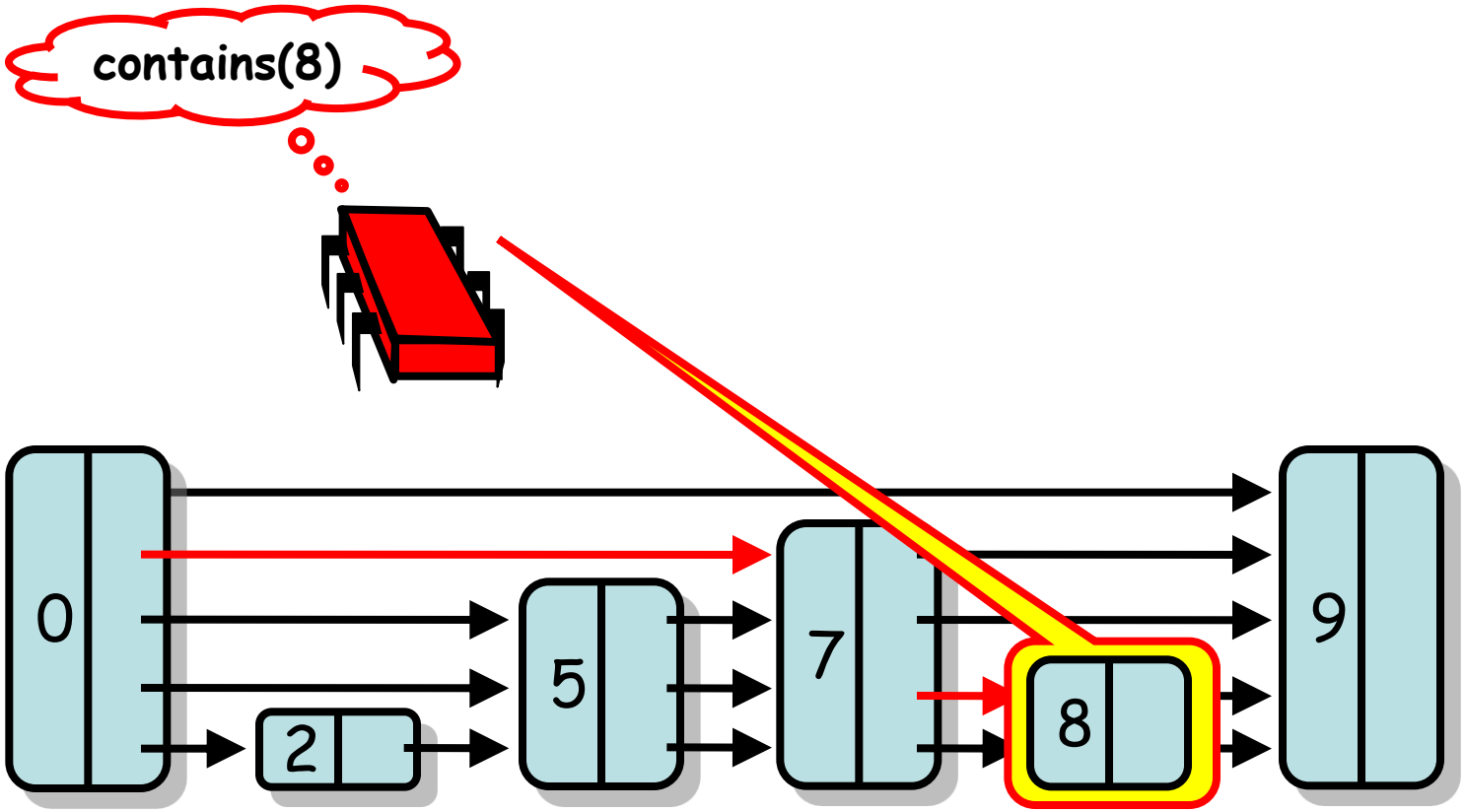


Search

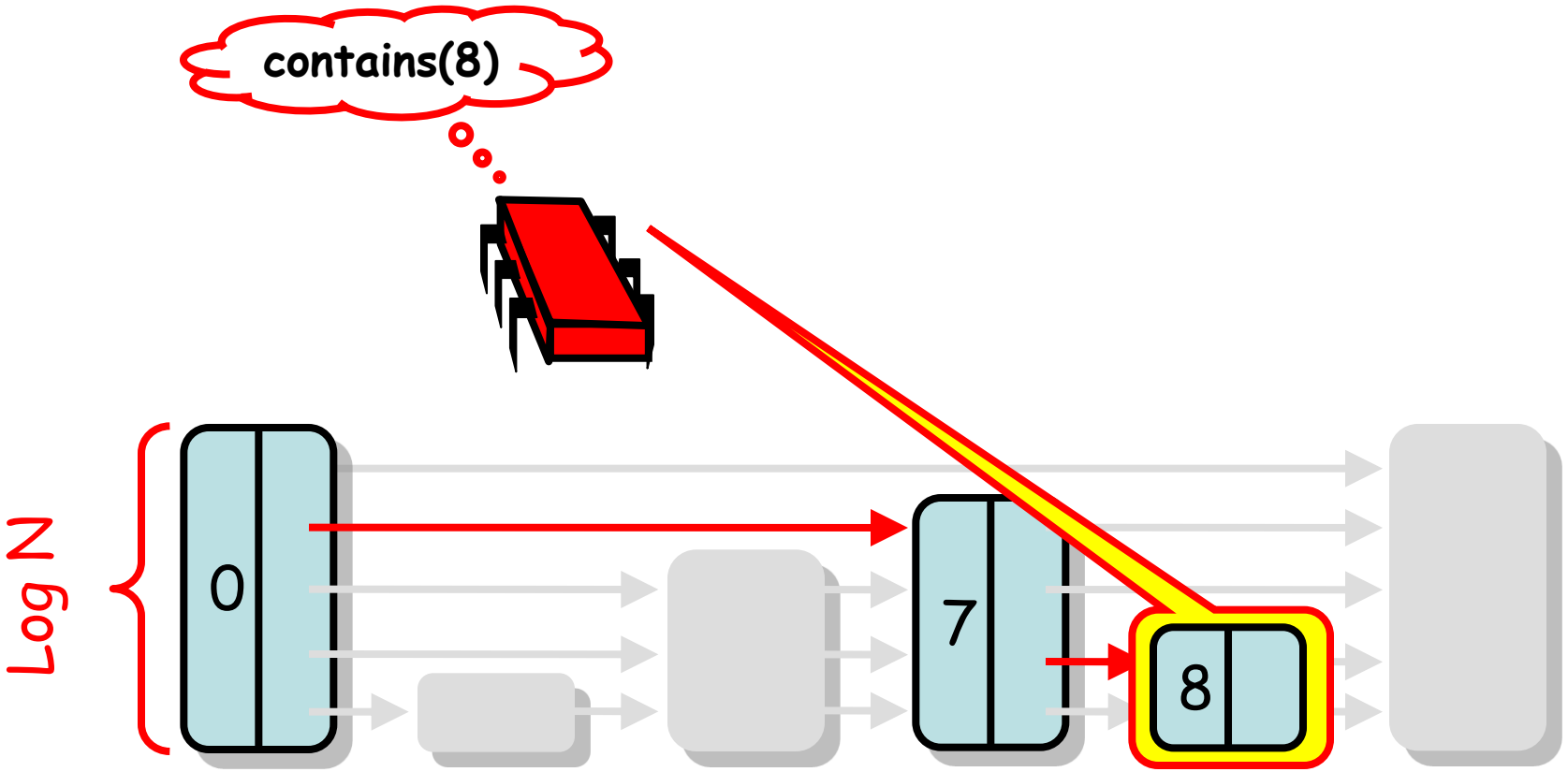




Search

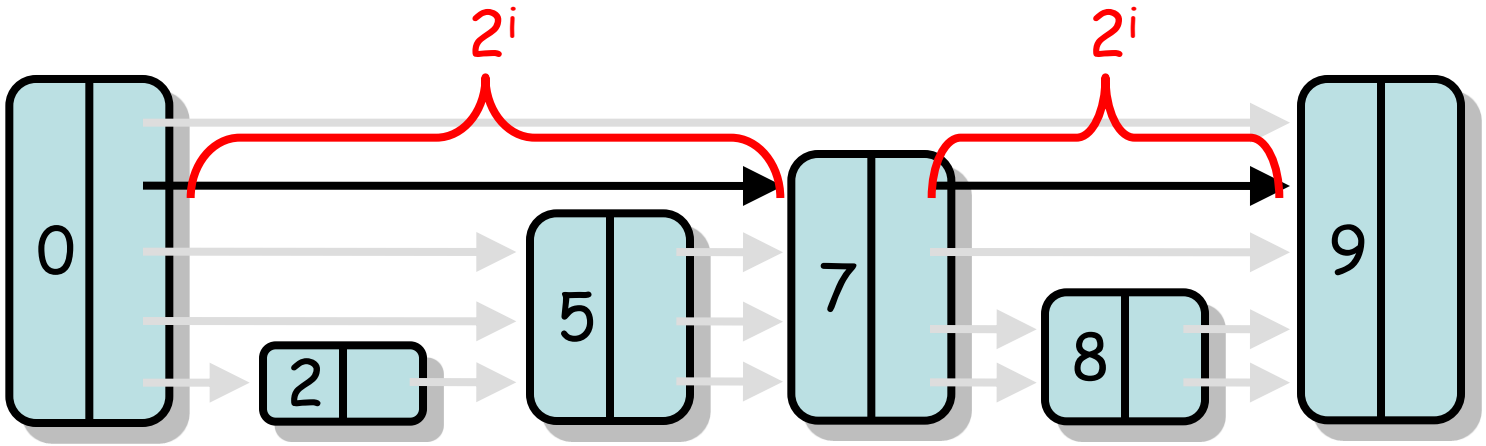


Logarithmic



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)

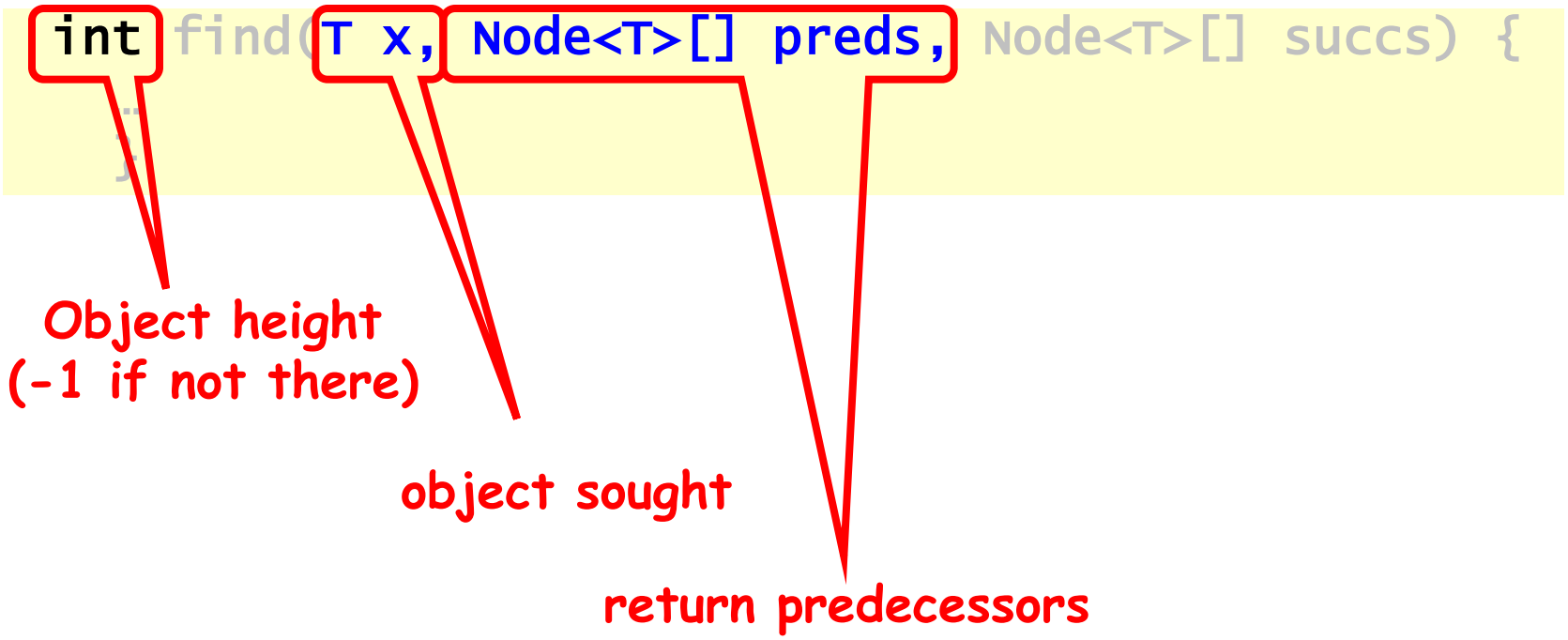
Find() -- Sequential

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

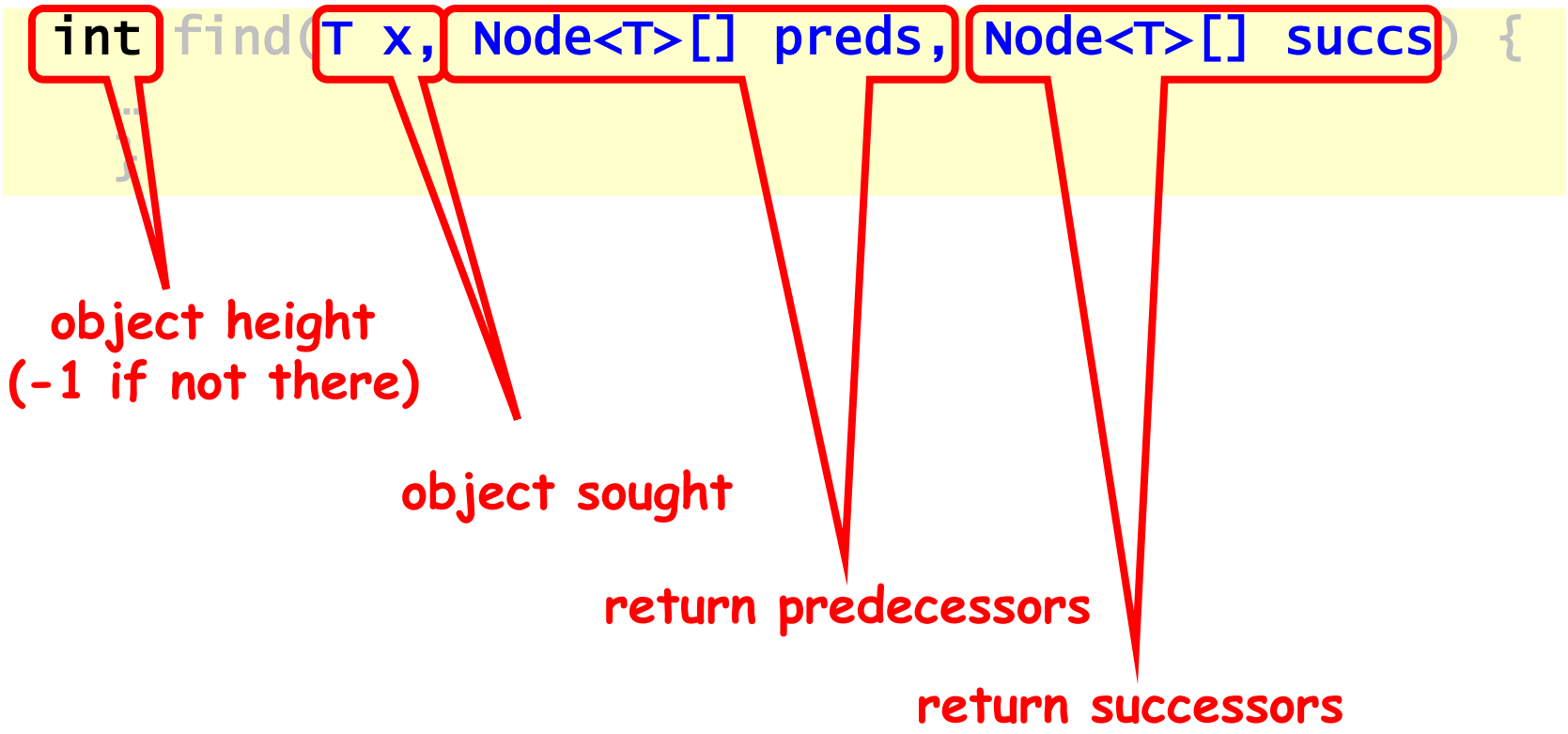
object height
(-1 if not there)

Object sought

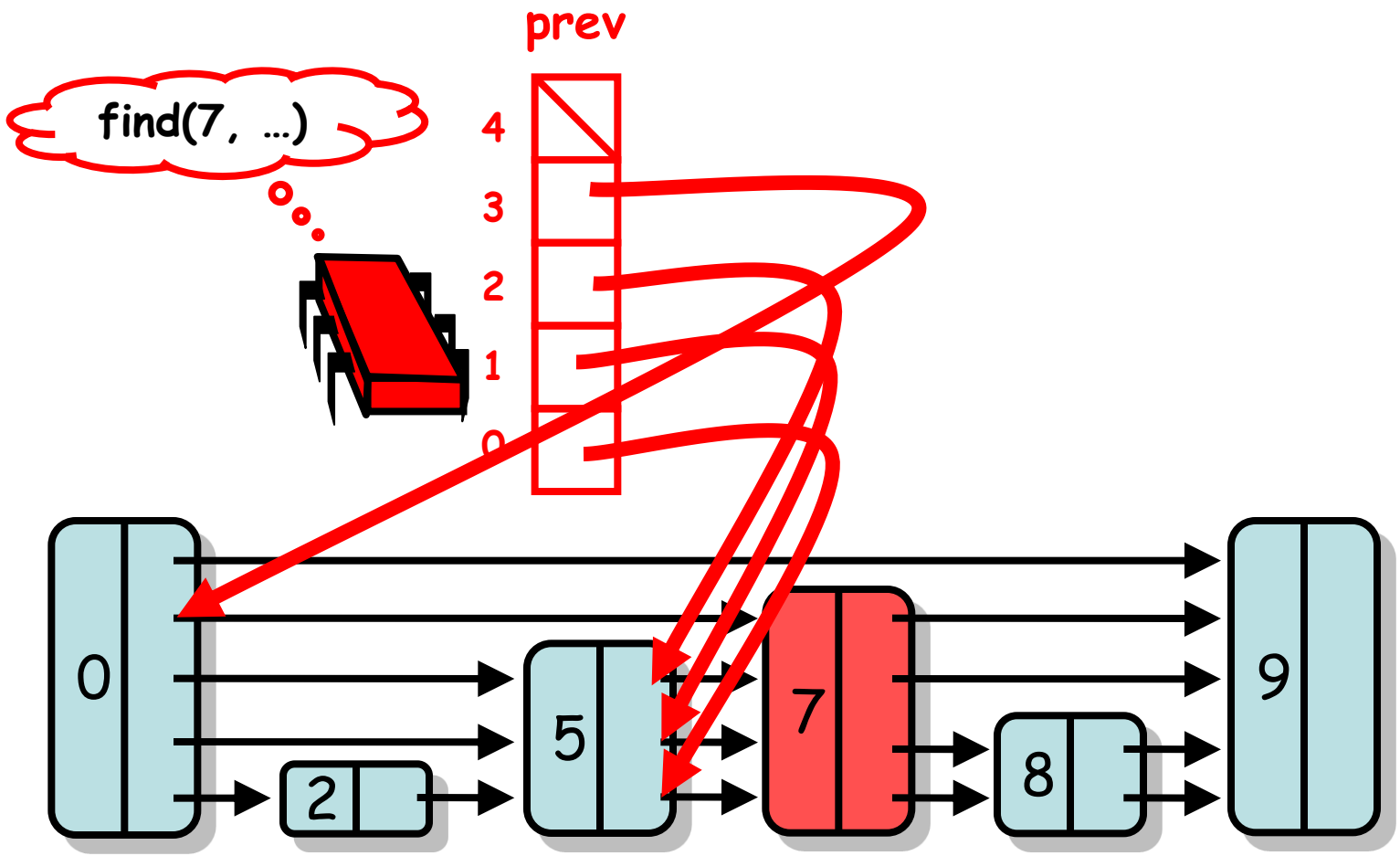
Find() -- Sequential



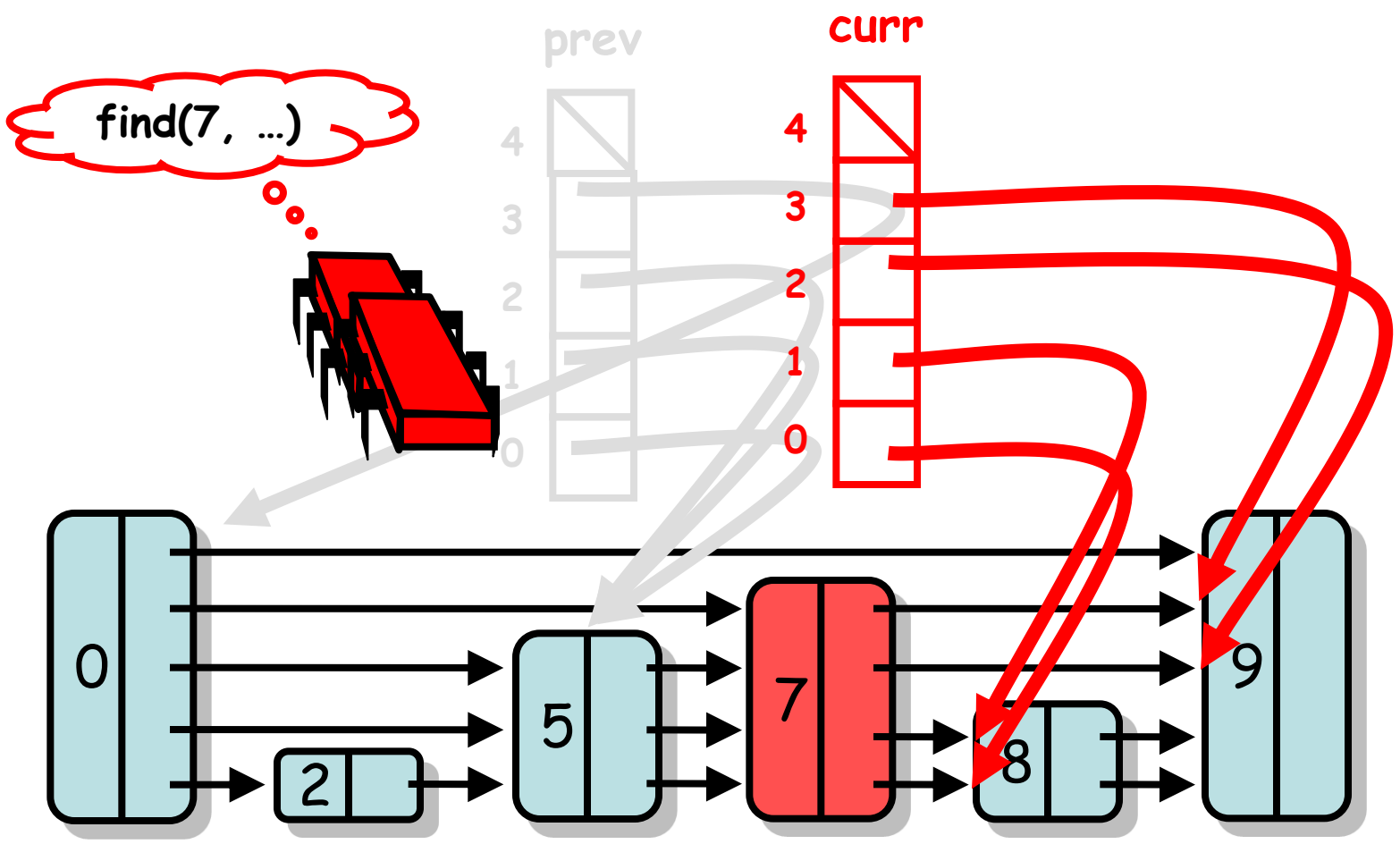
Find() -- Sequential



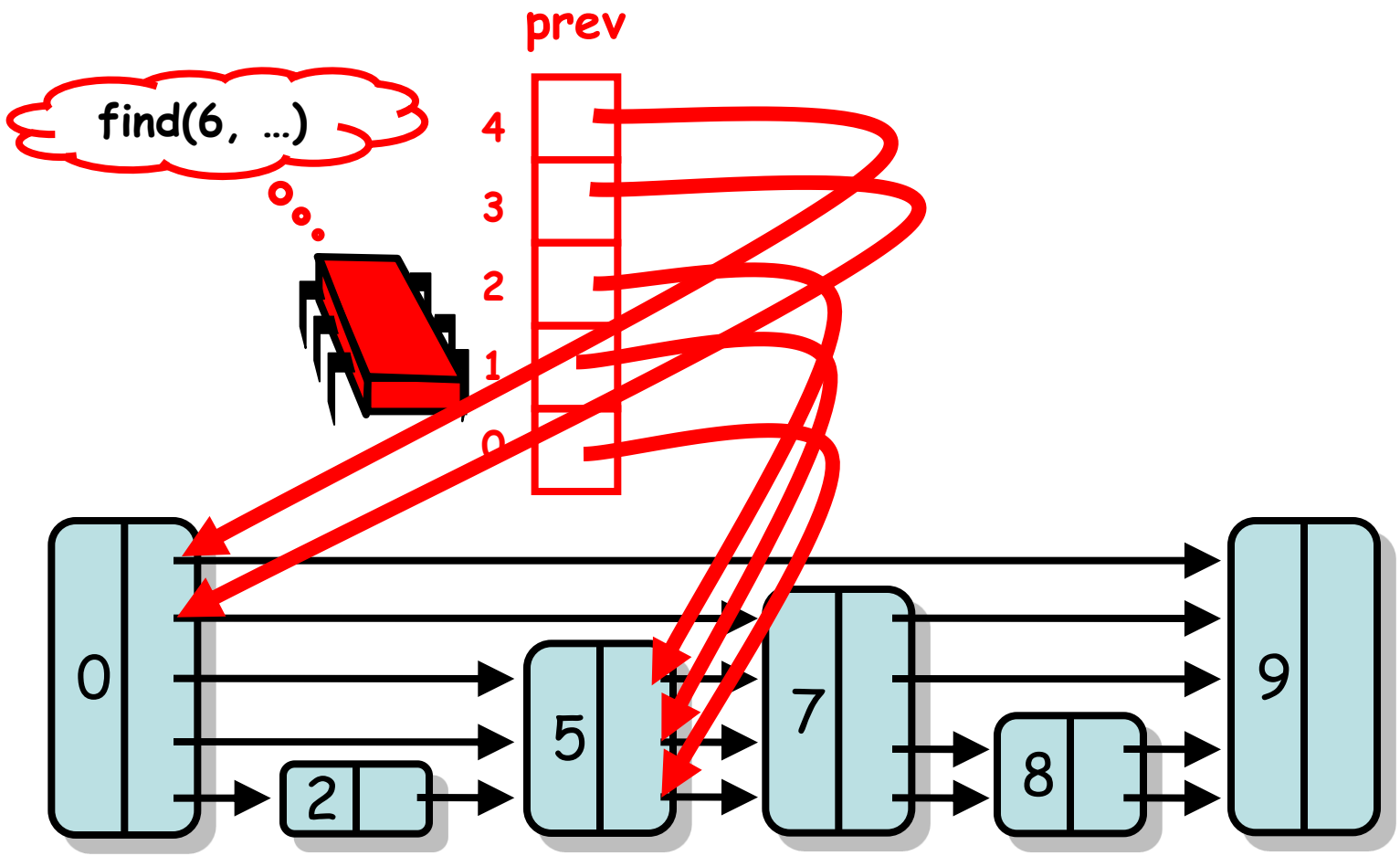
Successful Search



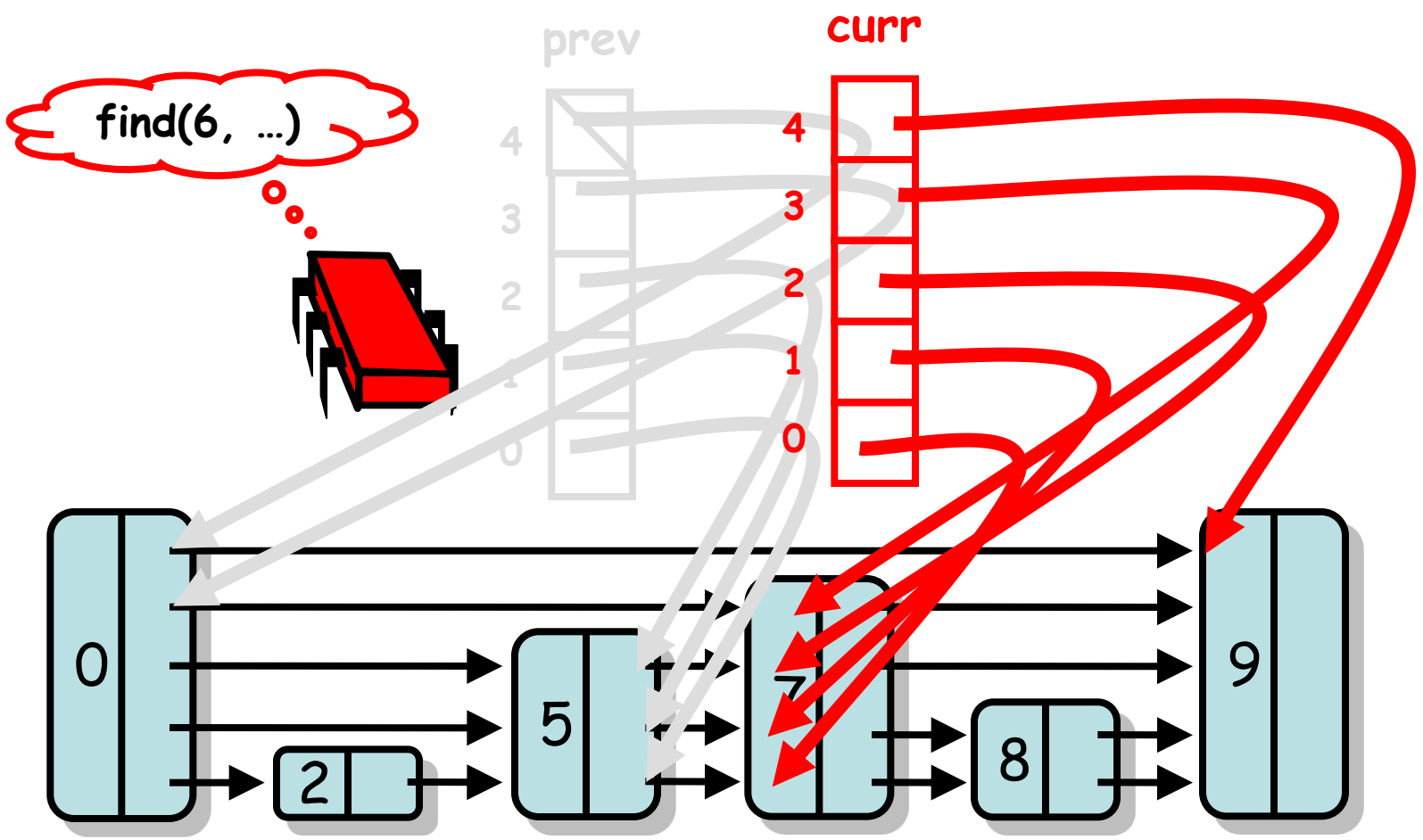
Successful Search



Unsuccessful Search



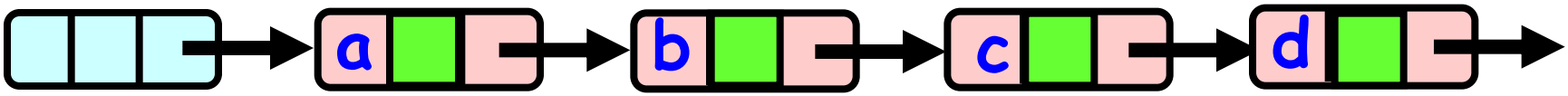
Unsuccessful Search



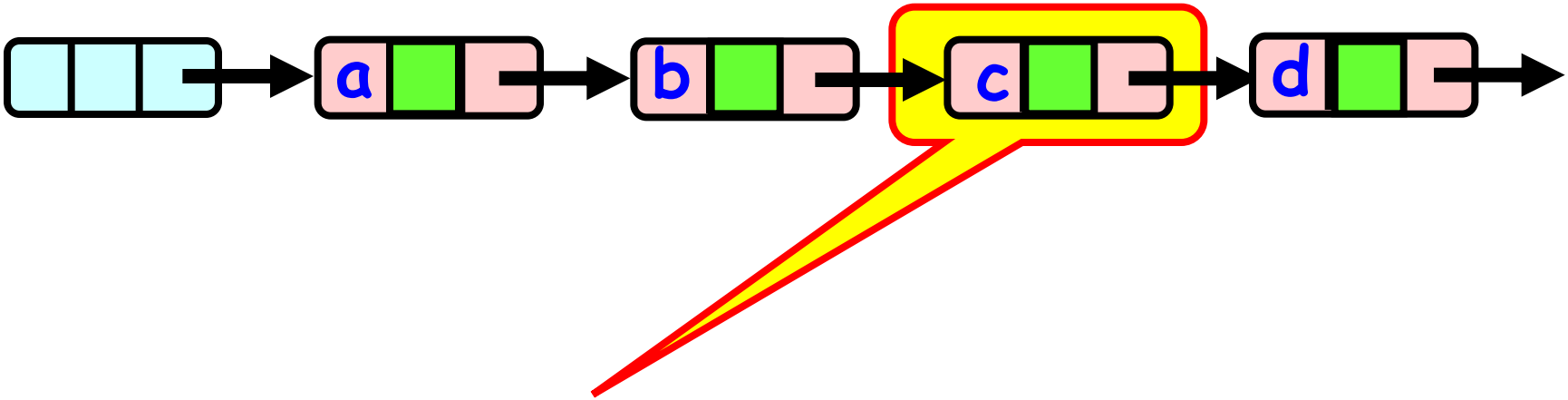
Lazy Skip List

- 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()

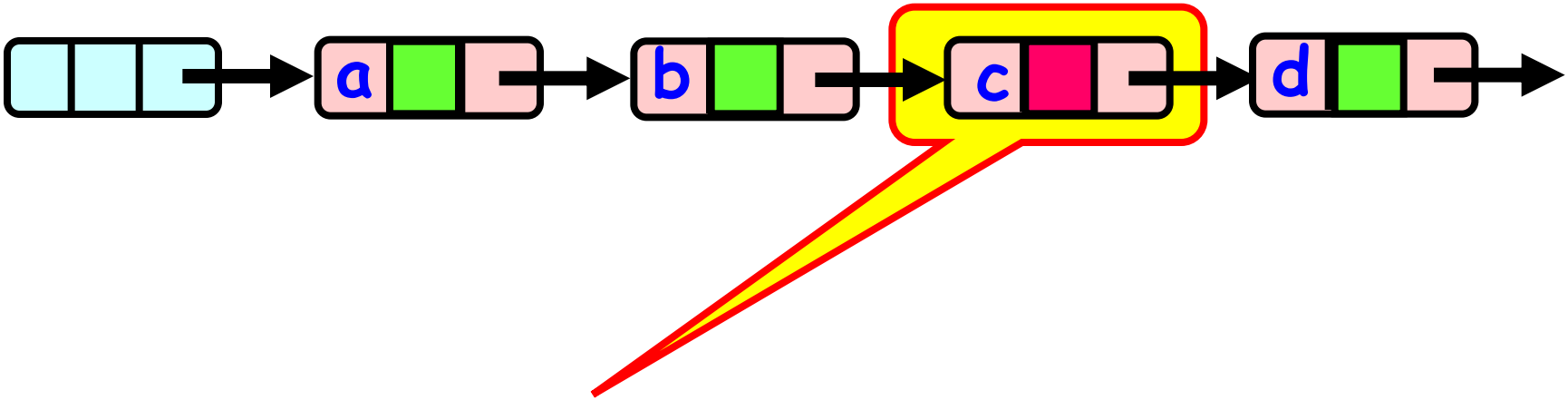
Review: Lazy List Remove



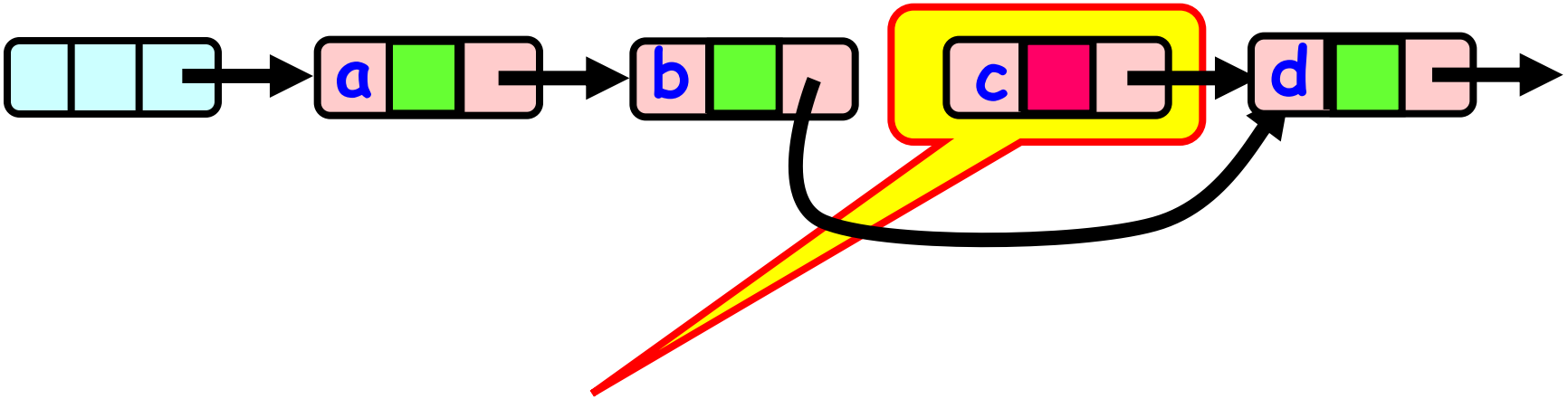
Review: Lazy List Remove



Review: Lazy List Remove

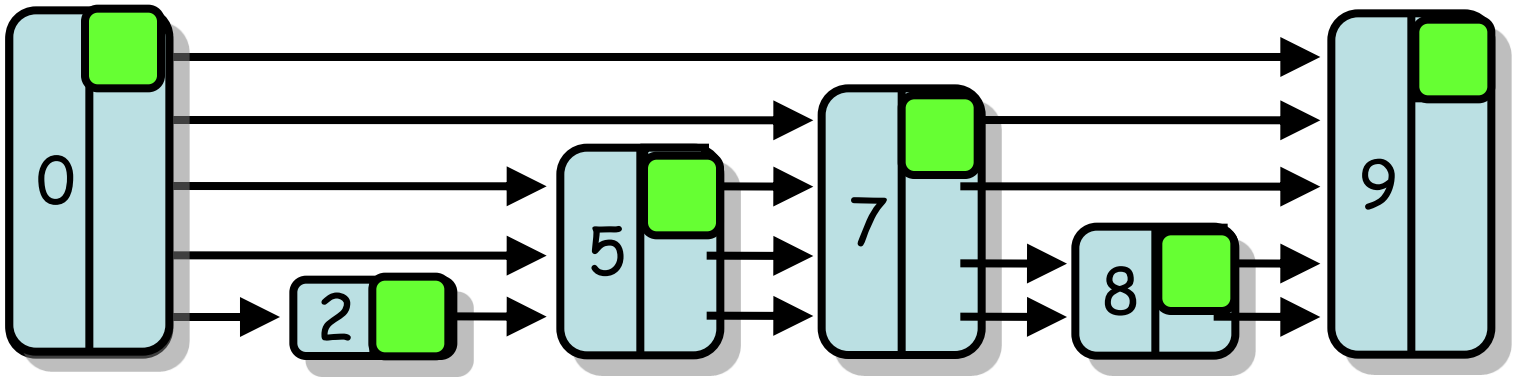


Review: Lazy List Remove



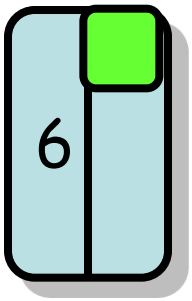
Lazy Skip Lists

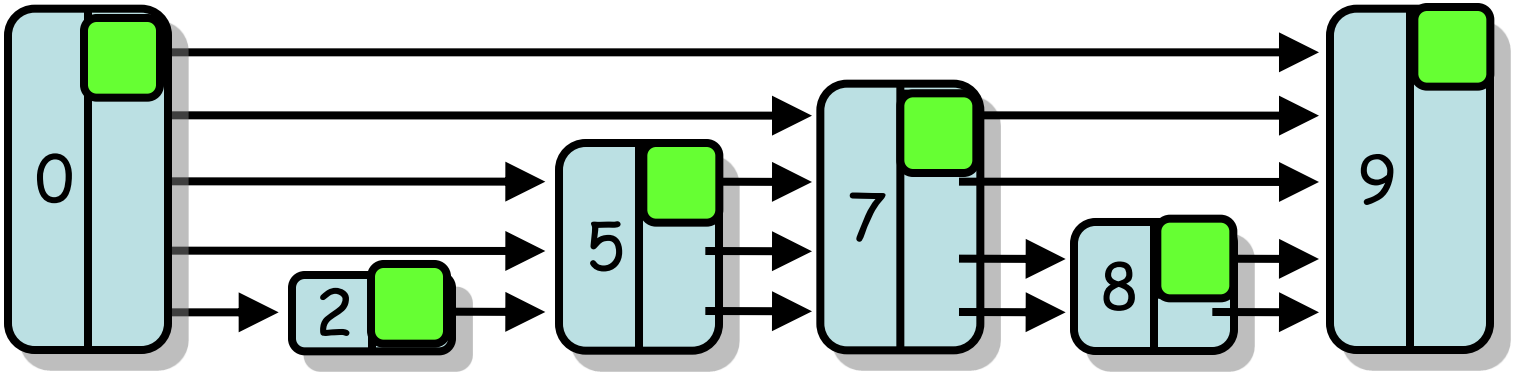
- Use a mark bit for logical deletion



add(6)

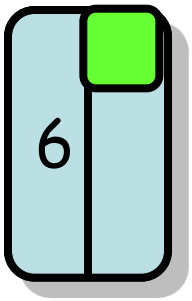
- Create node of (random) height 4

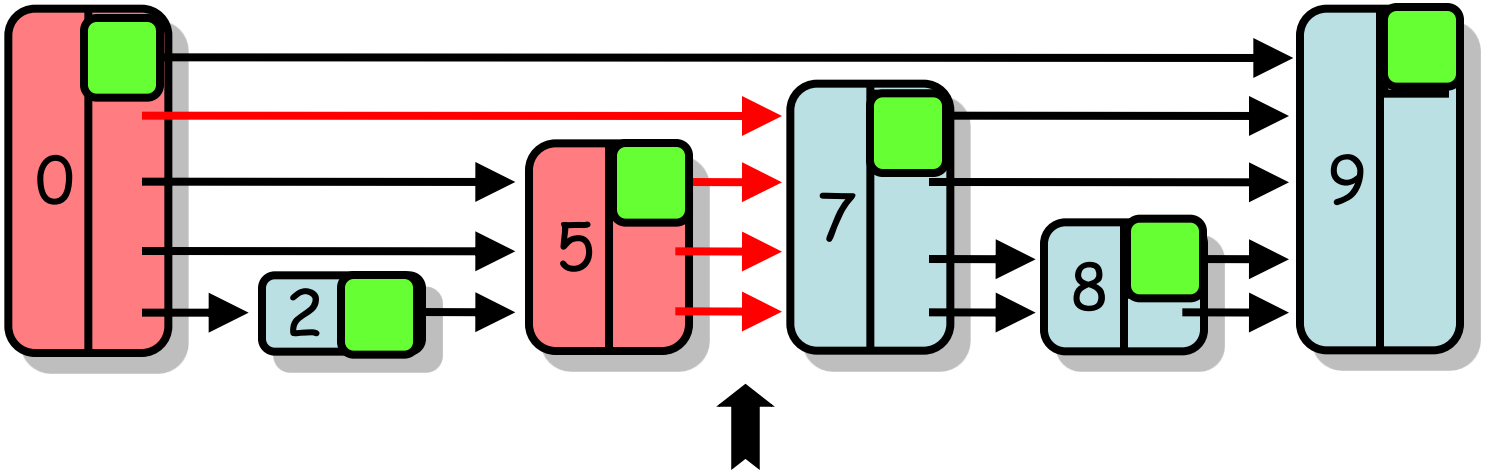




add(6)

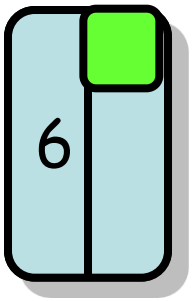
- find() predecessors

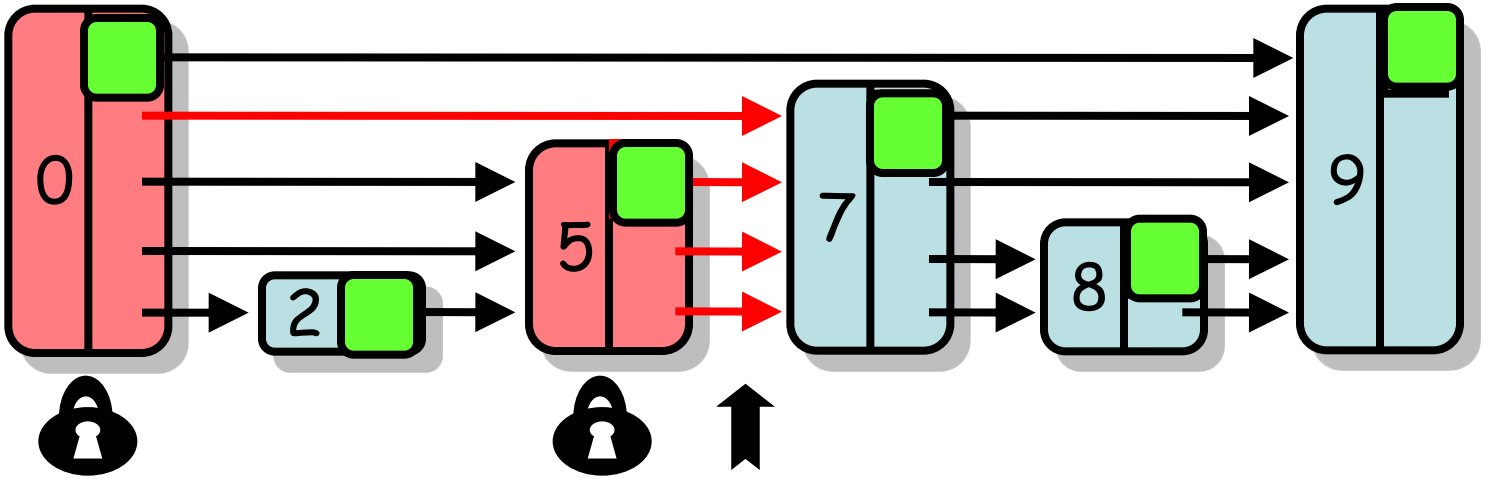




add(6)

- find() predecessors
- Lock them

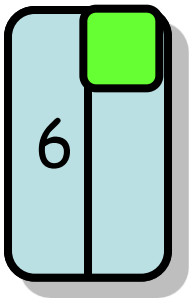


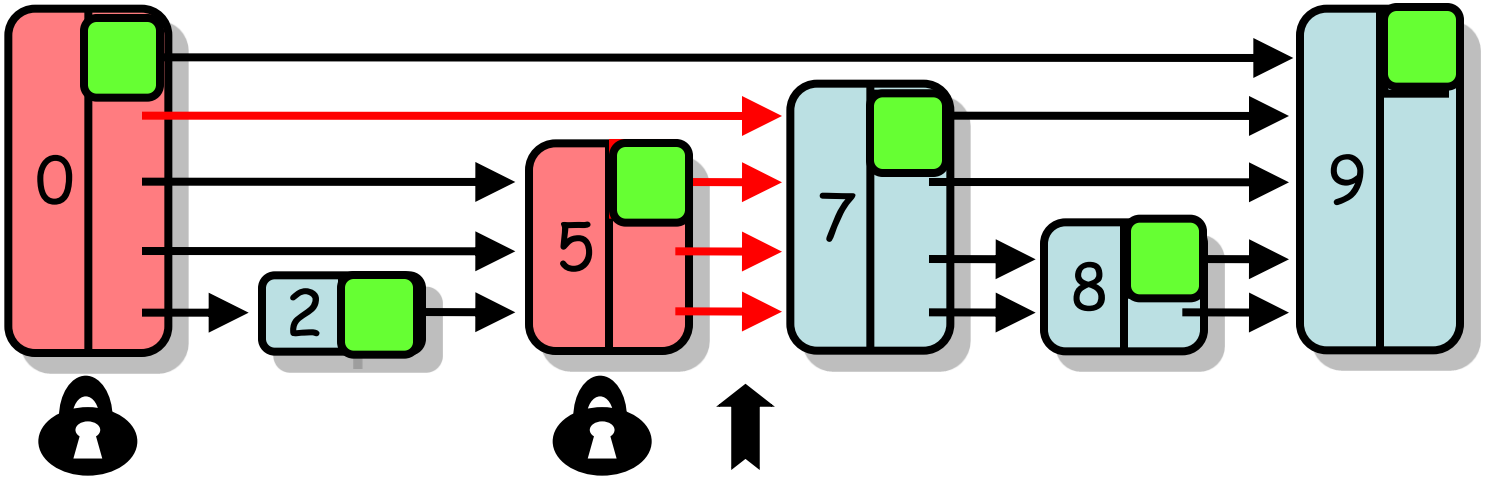


add(6)

- find() predecessors
- Lock them
- Validate

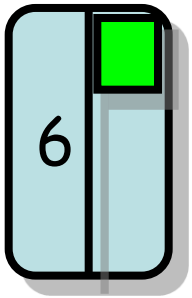
} Optimistic approach

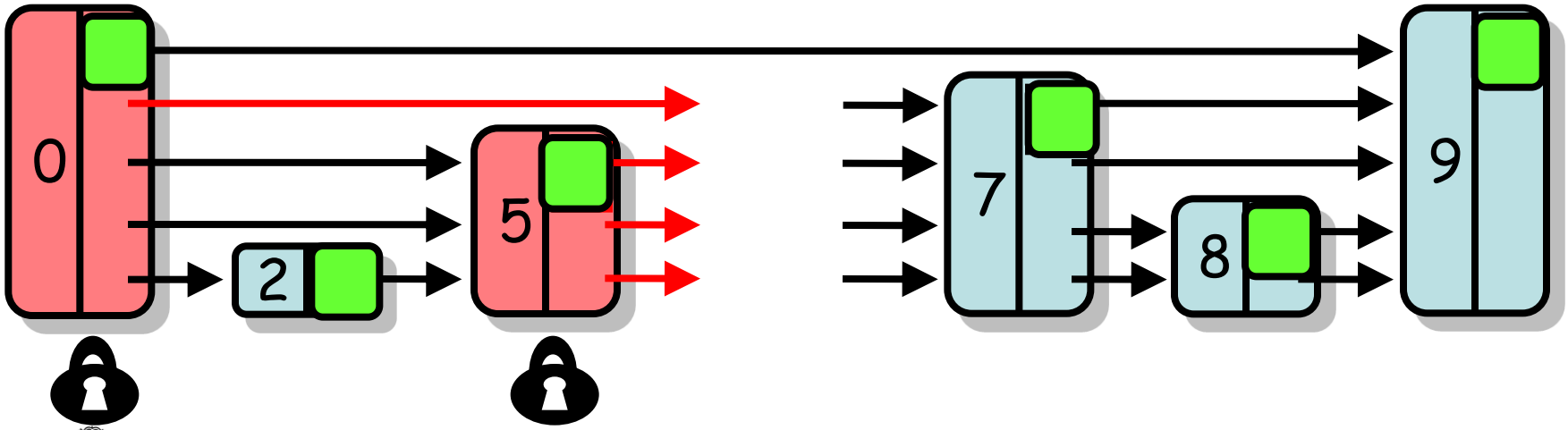




add(6)

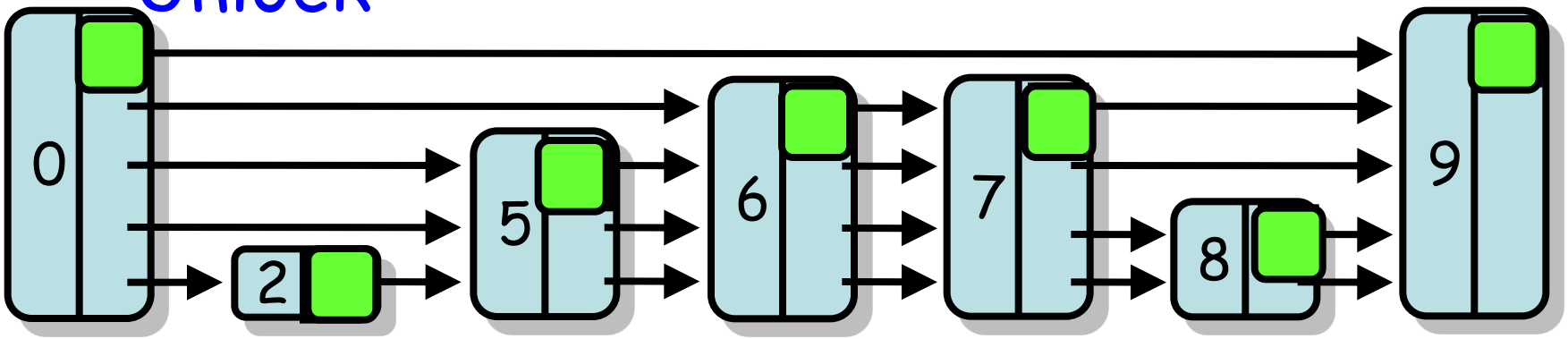
- **find()** predecessors
- Lock them
- Validate
- Splice



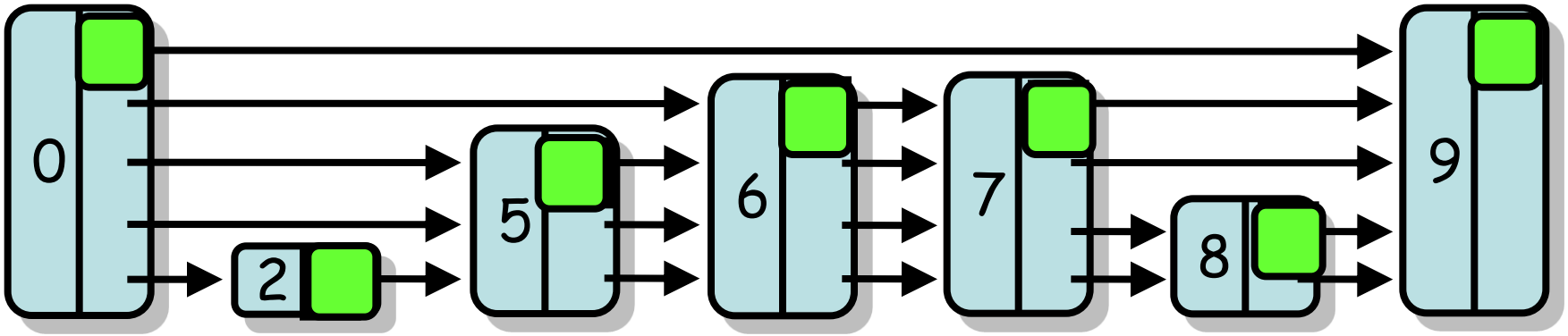


add(6)

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

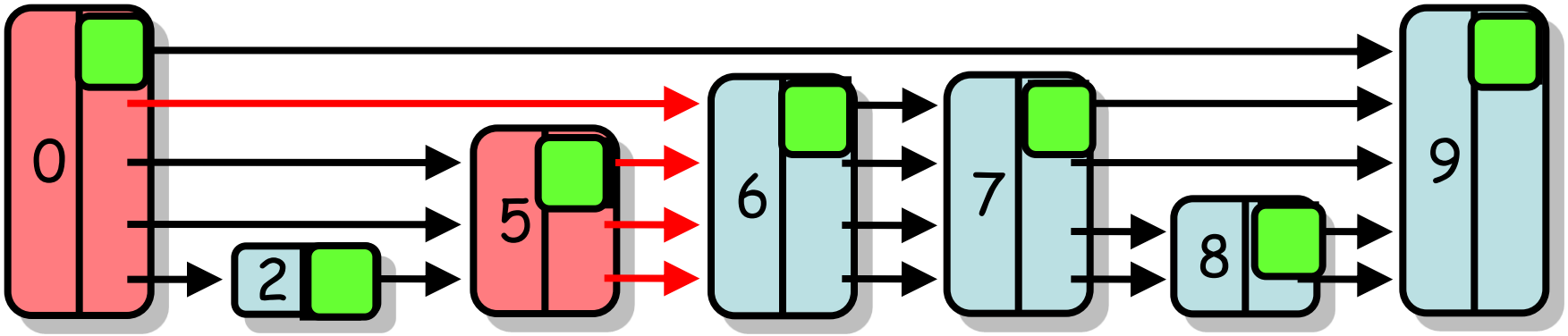


remove(6)



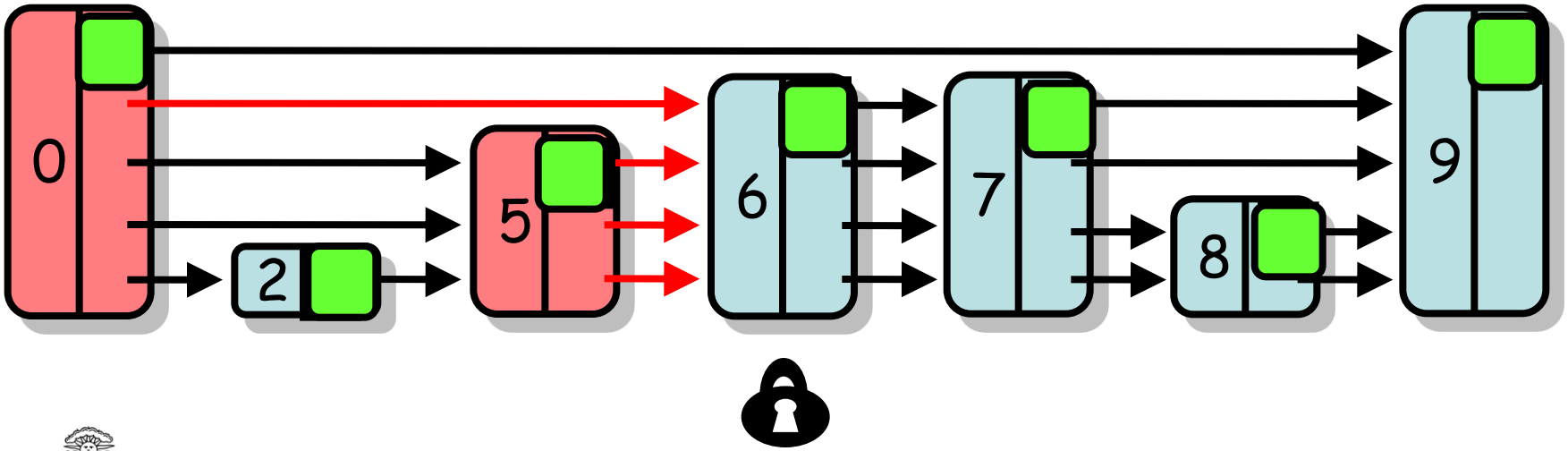
remove(6)

- find() predecessors



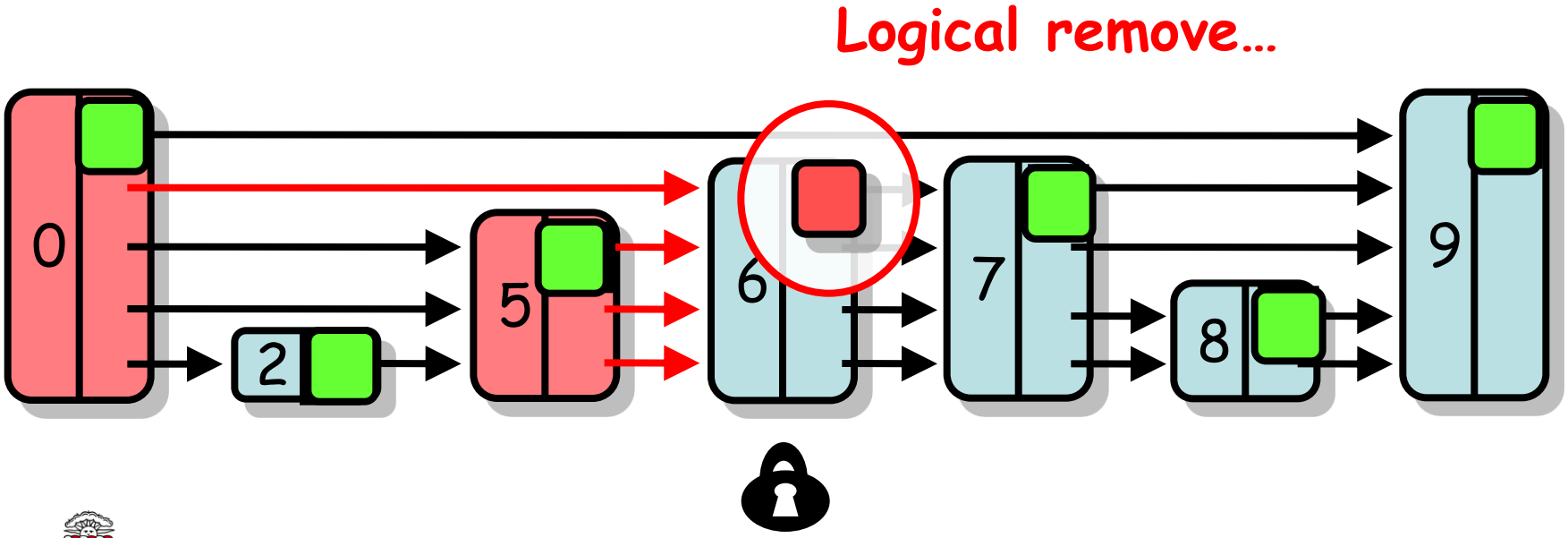
remove(6)

- find() predecessors
- Lock victim



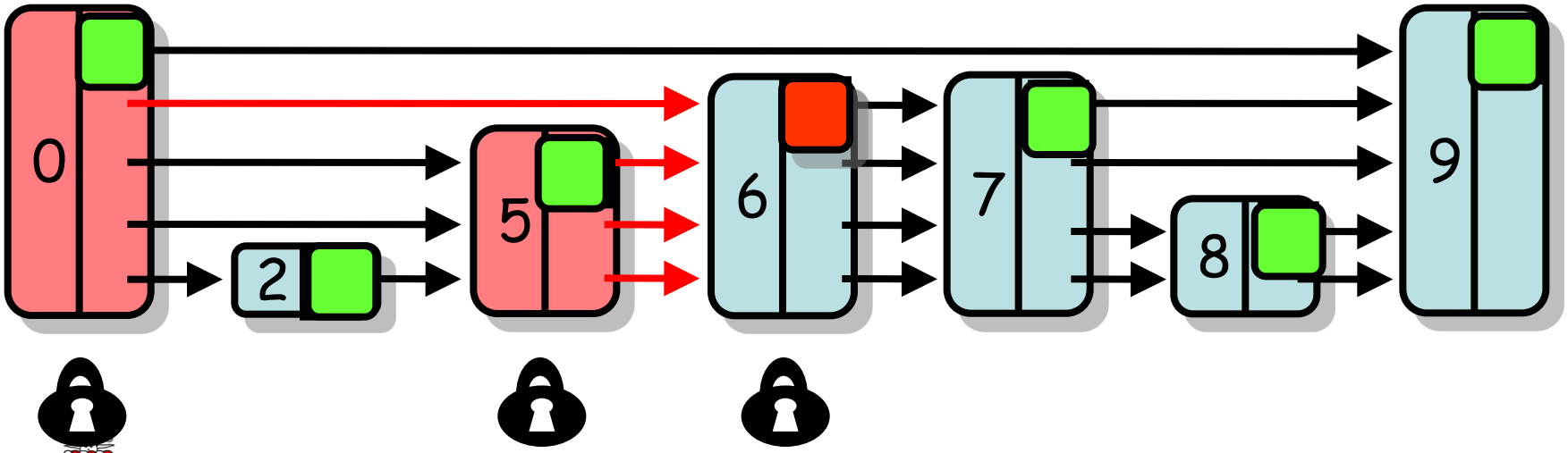
remove(6)

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



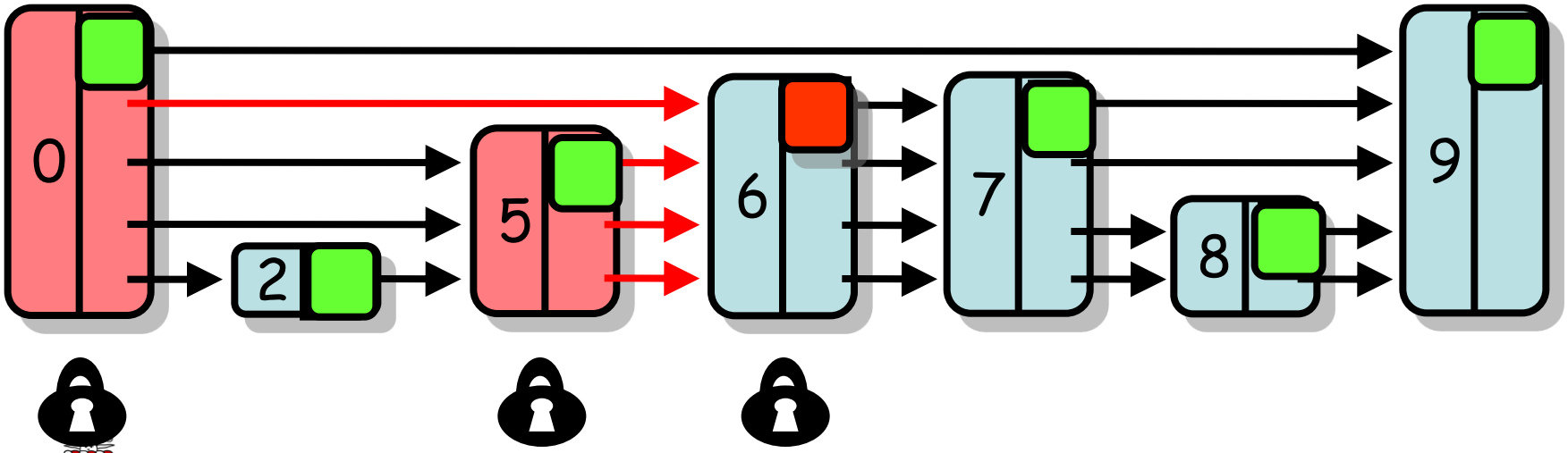
remove(6)

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



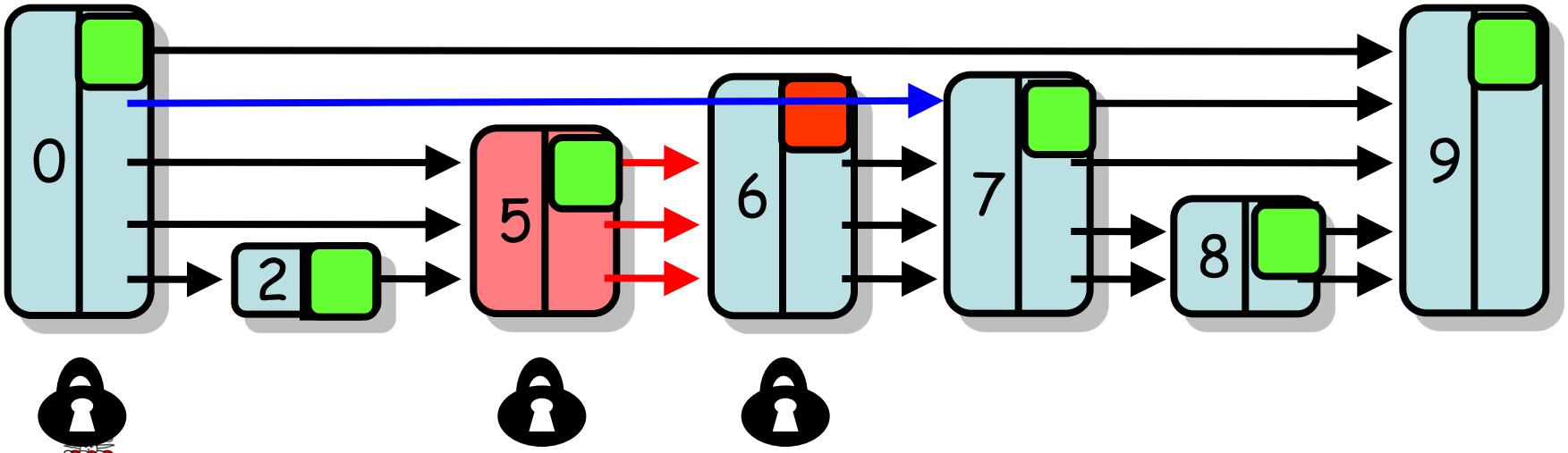
remove(6)

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



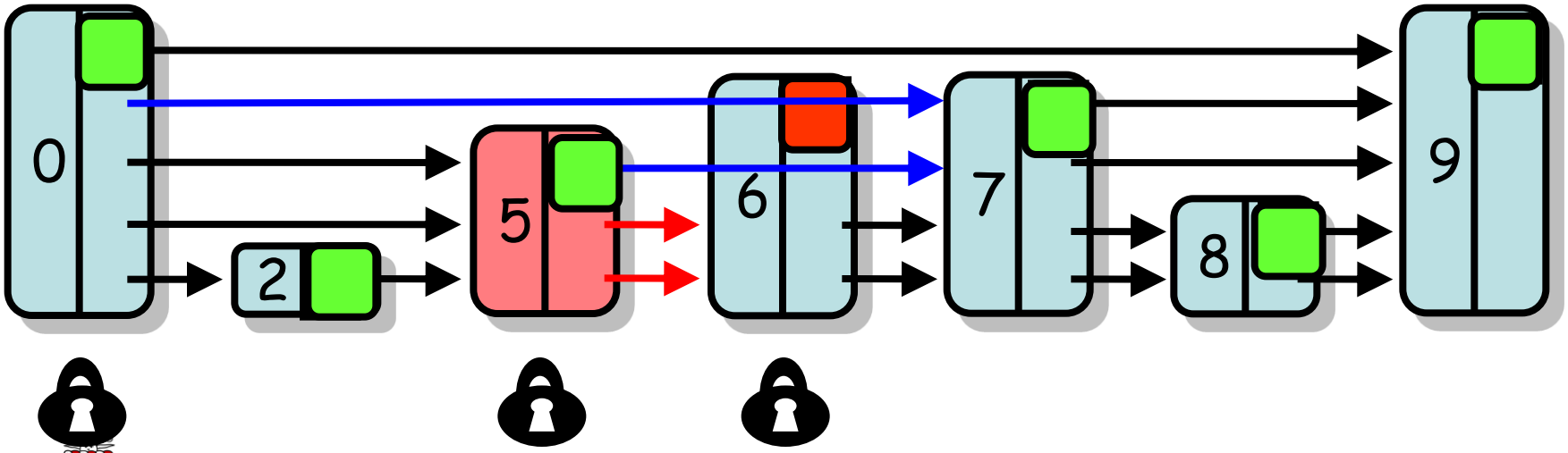
remove(6)

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



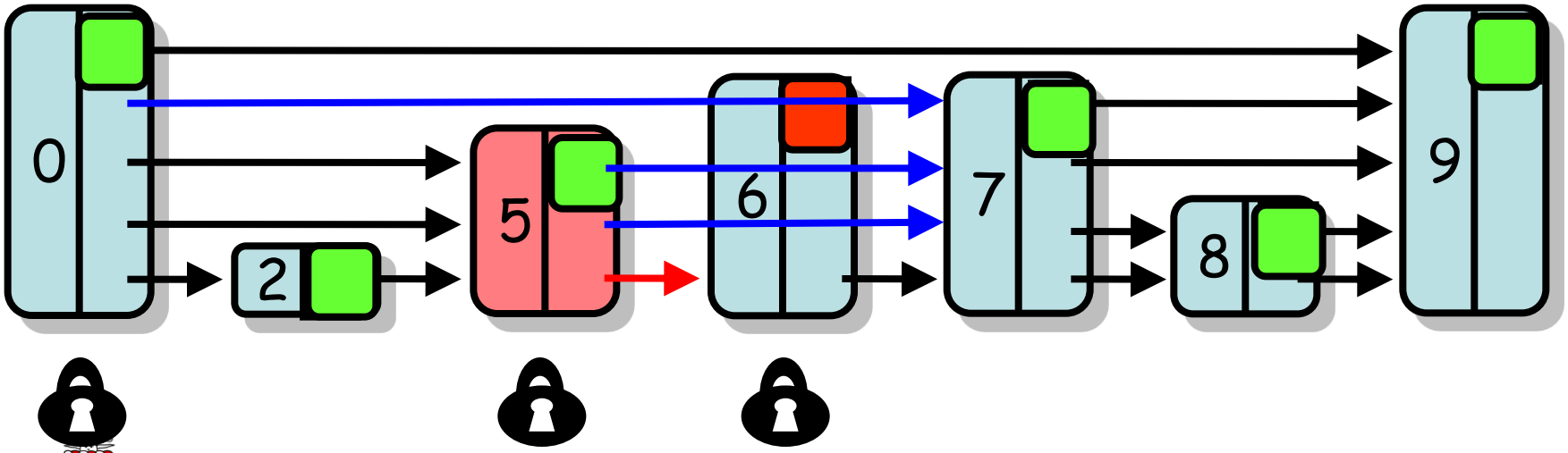
remove(6)

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



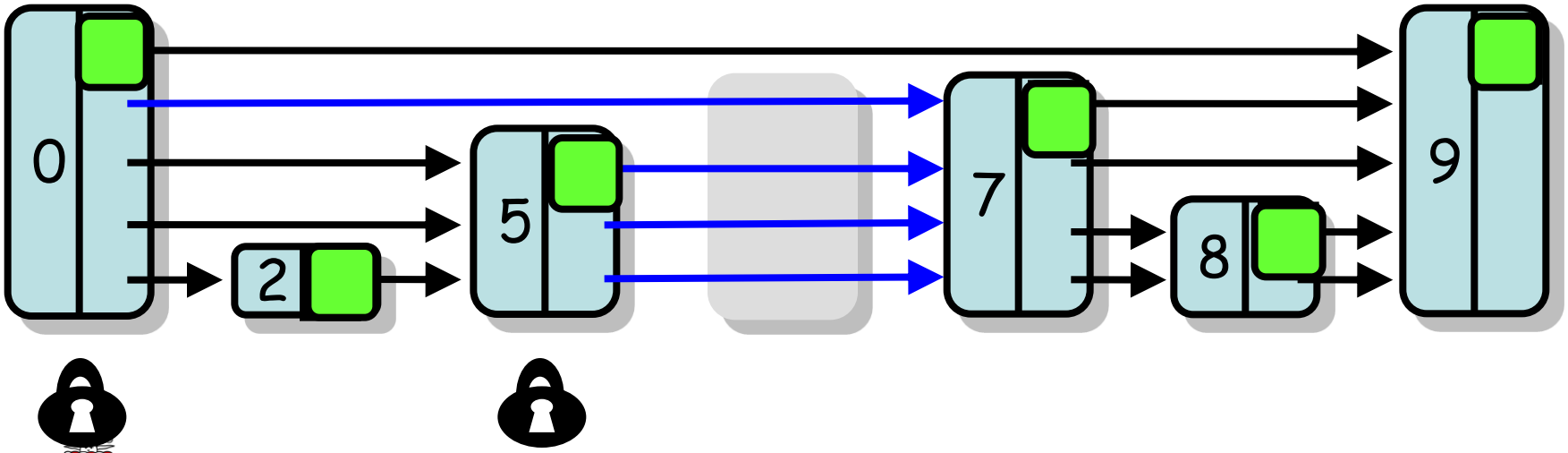
remove(6)

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



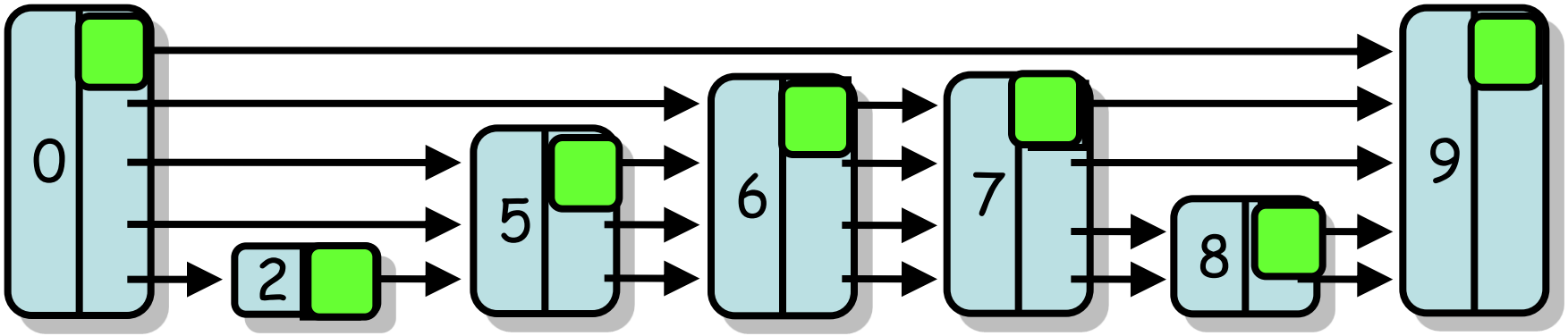
remove(6)

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



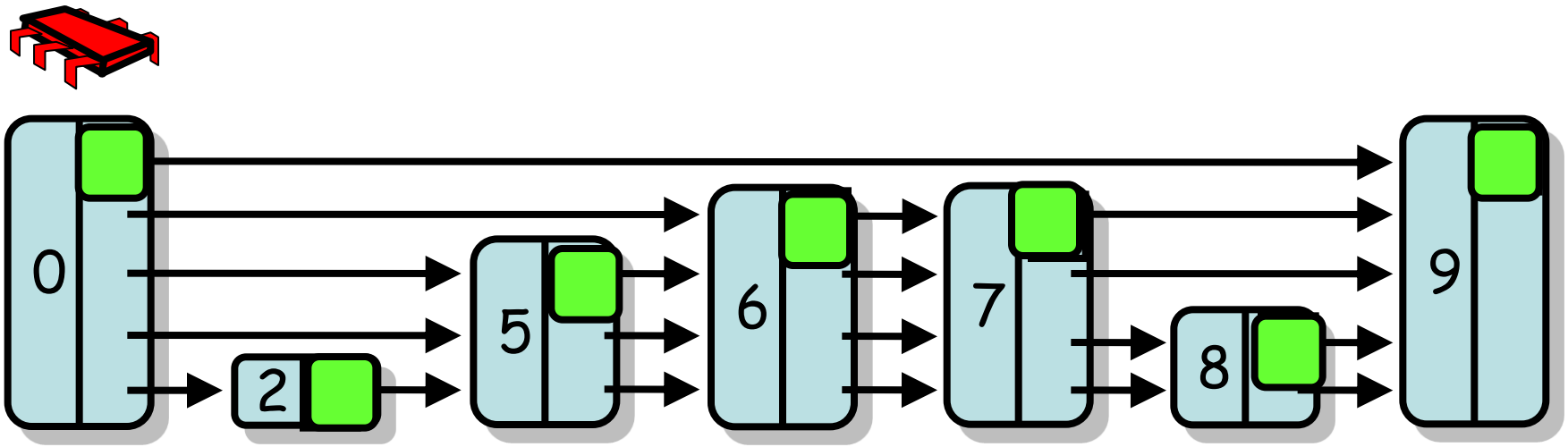
contains(8)

- Find() & not marked



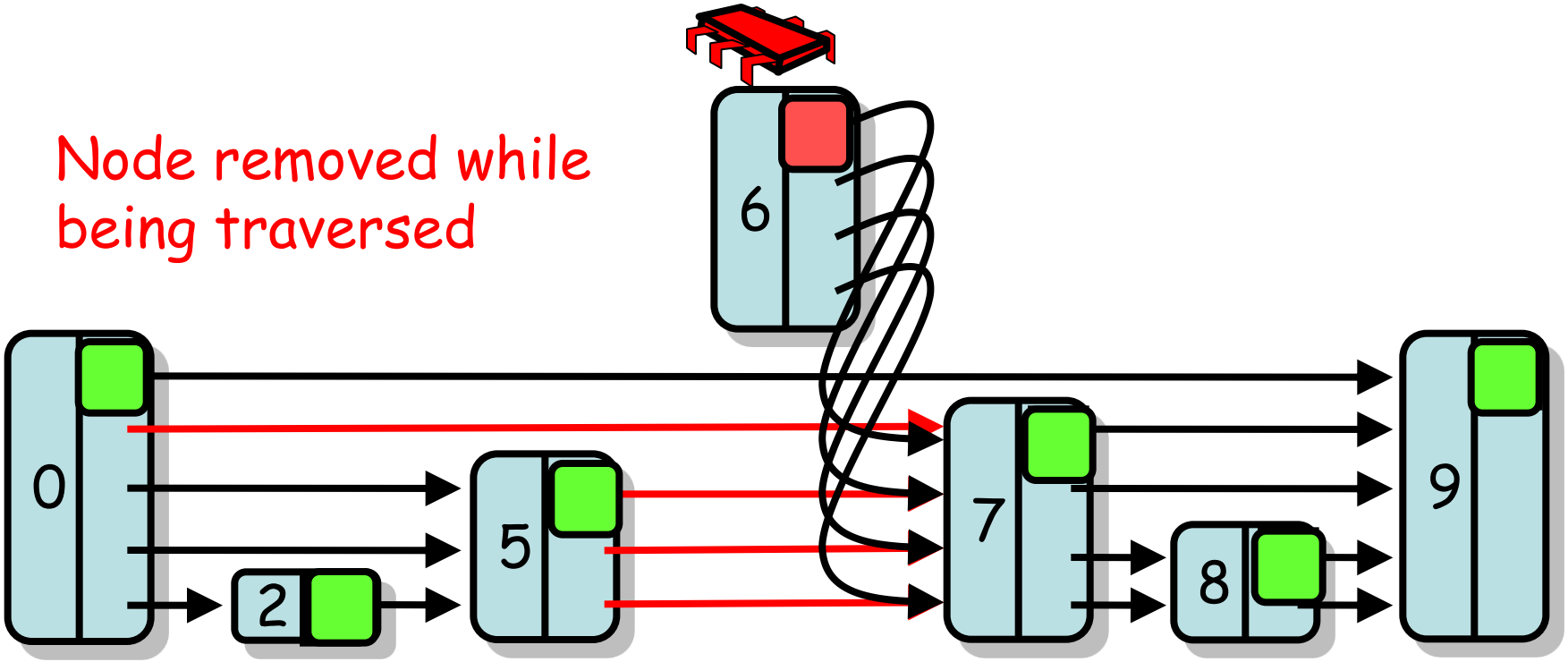
contains(8)

Node 6 removed while traversed



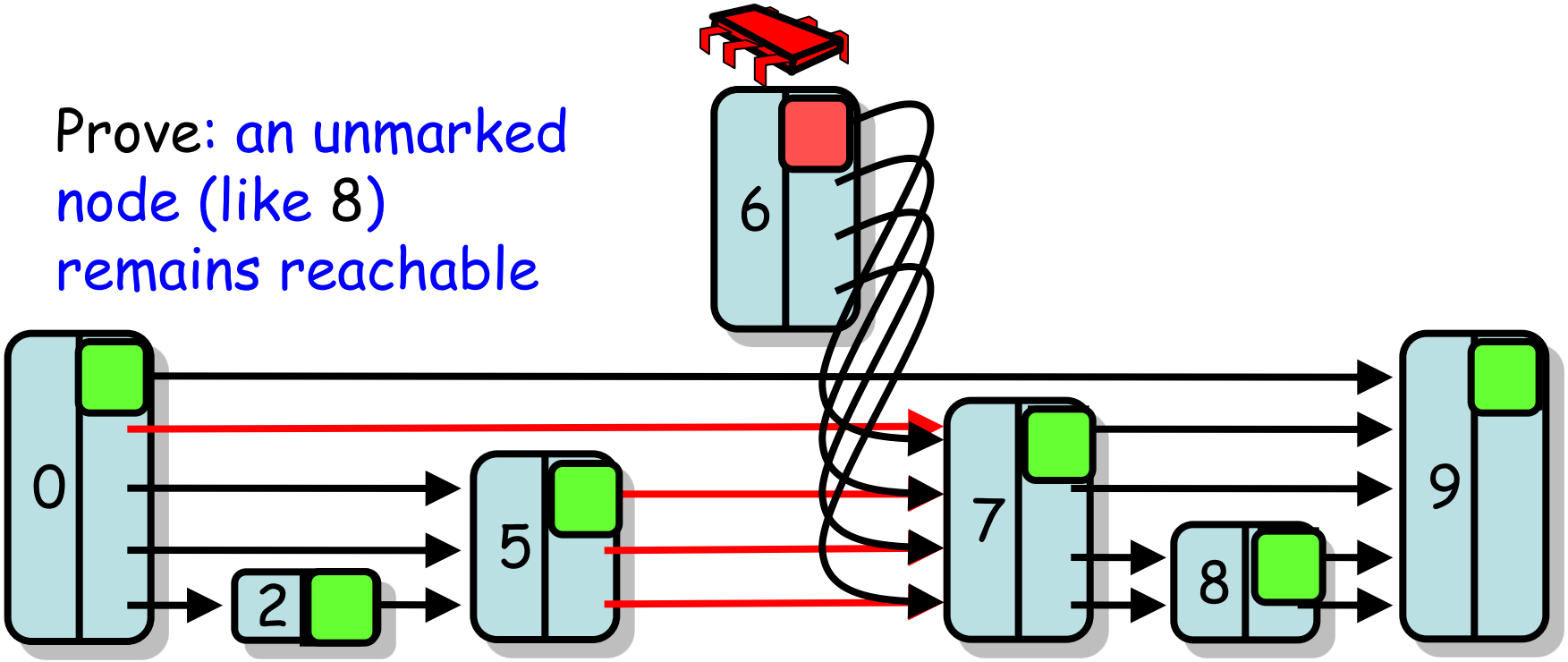
contains(8)

Node removed while being traversed



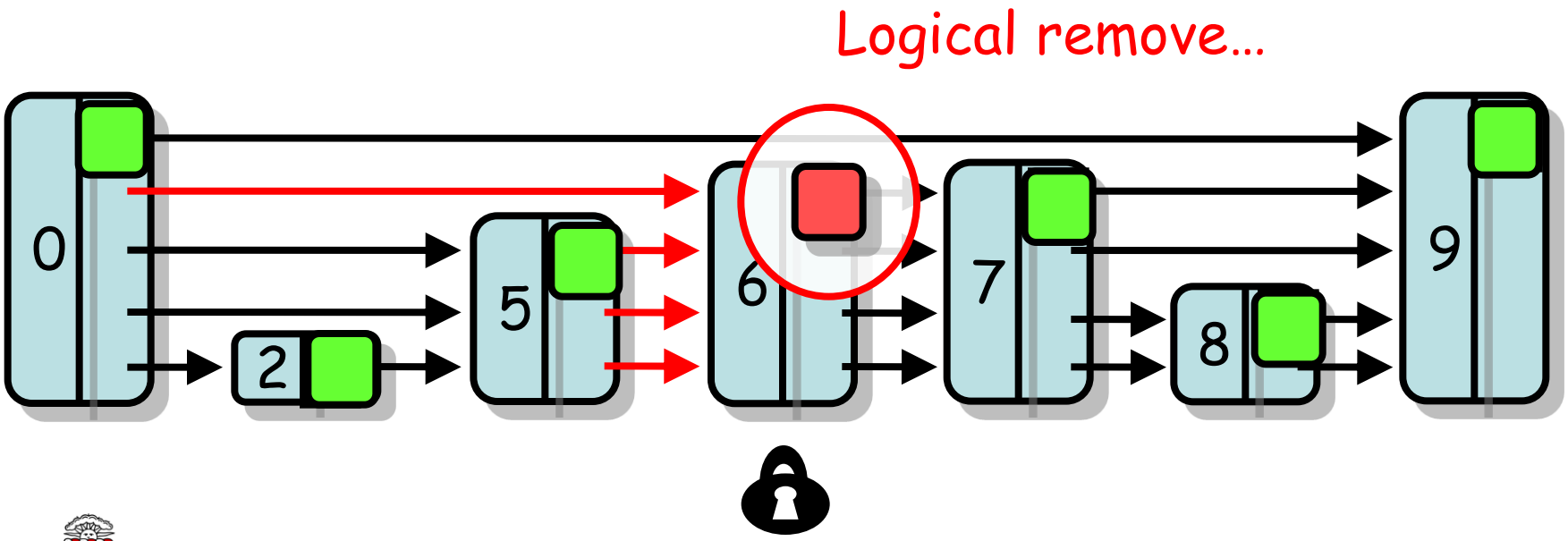
contains(8)

Prove: an unmarked node (like 8) remains reachable



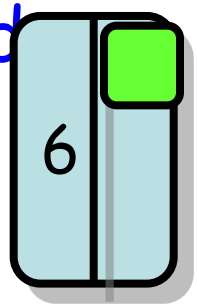
remove(6): Linearization

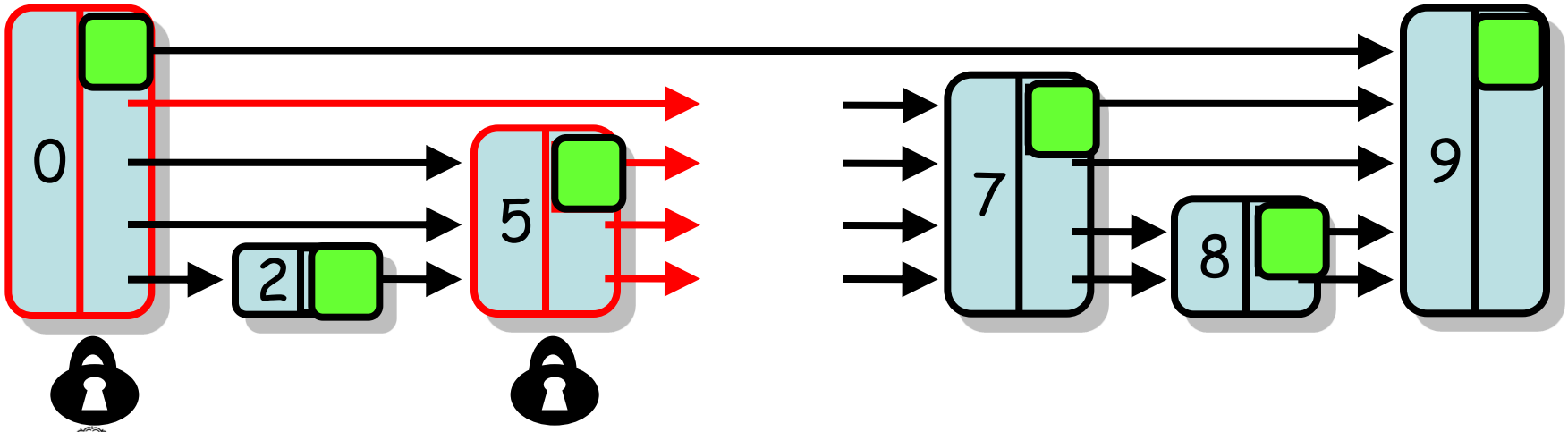
- Successful remove happens when bit is set



Add: Linearization

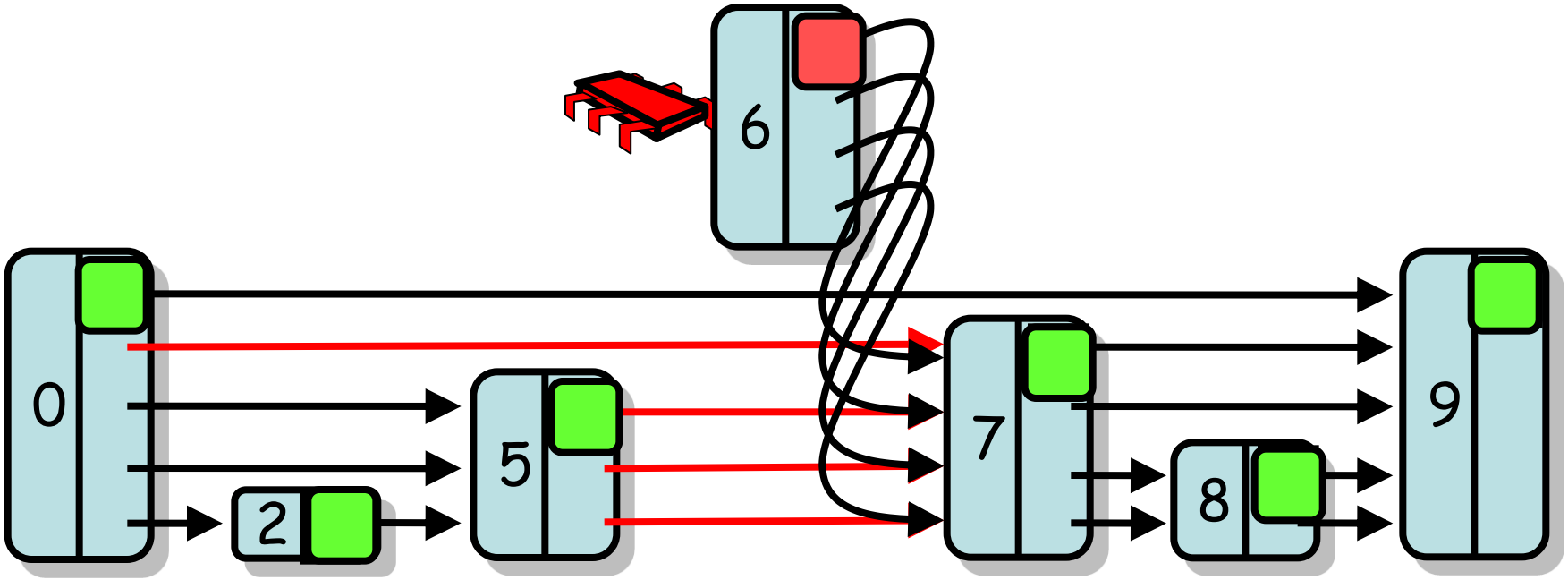
- Successful add() at point when fully linked
- Add fullyLinked bit to indicate this
- Bit tested by contains()





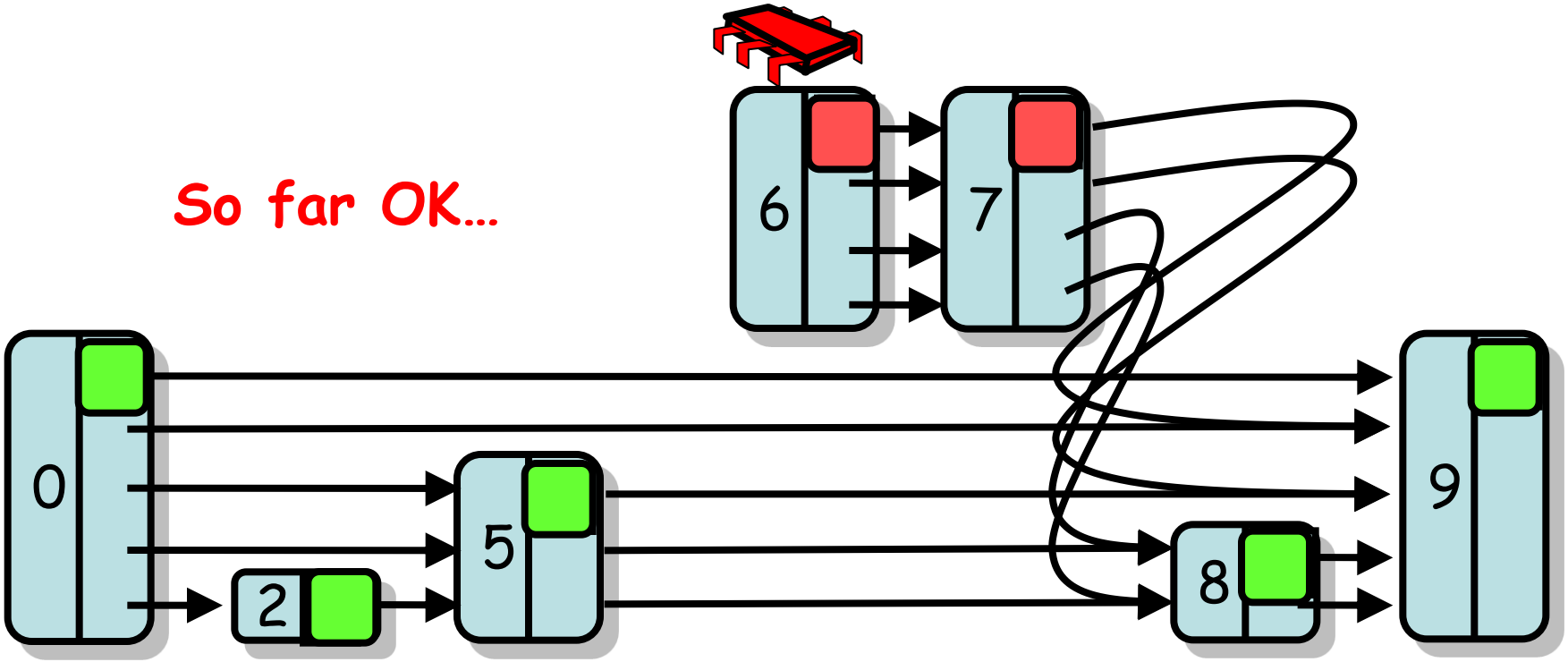
Unsuccessful add(6) Linearization

- When not fully-linked unmarked node found
- Pause while fullyLinked bit sets



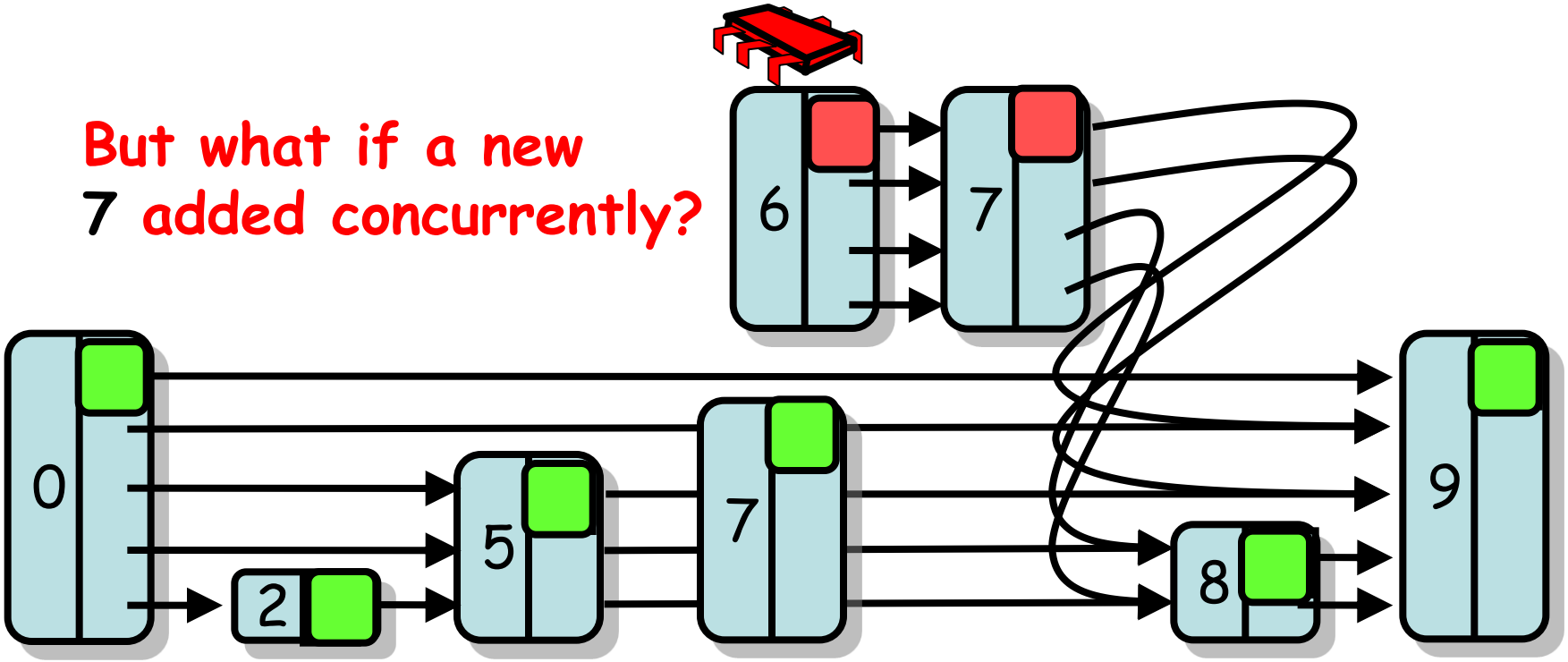
contains(7): Linearization

- When do we linearize unsuccessful Search?



contains(7): Linearization

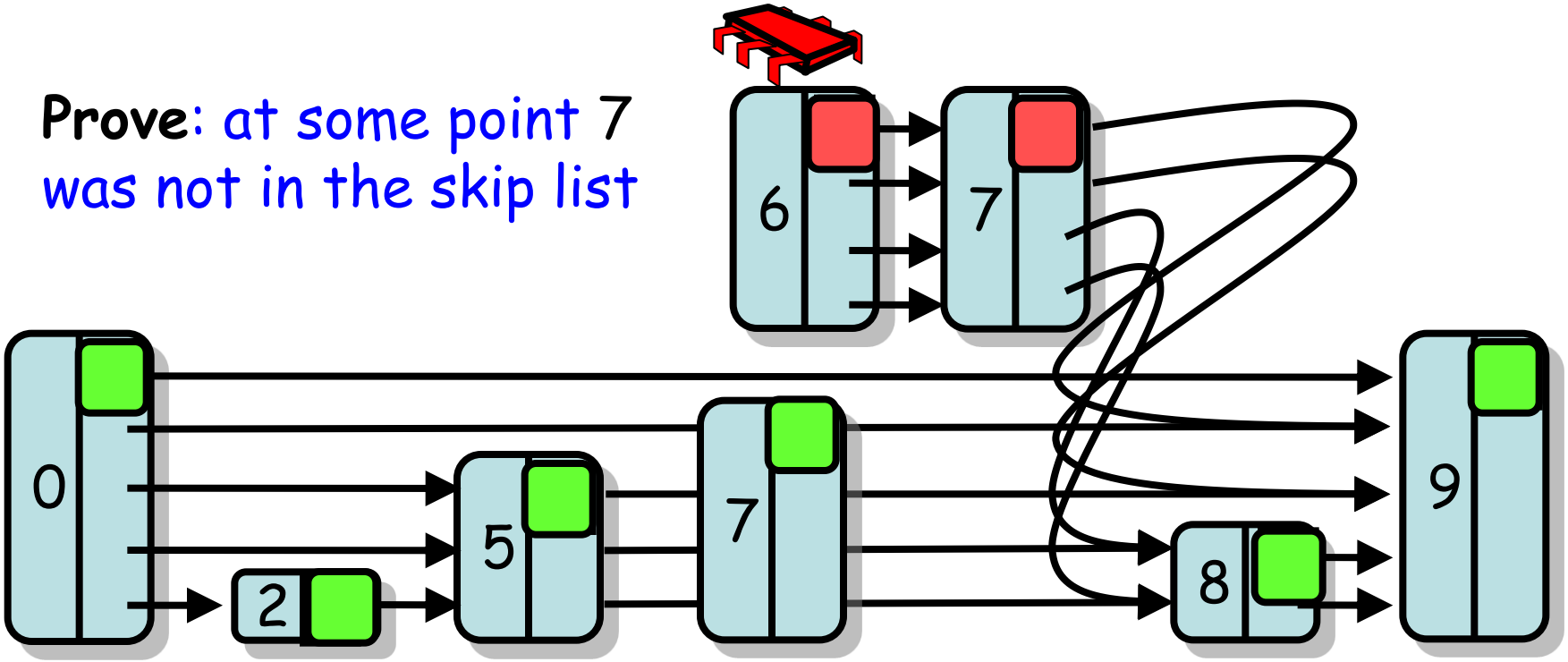
- When do we linearize unsuccessful Search?



contains(7): Linearization

- When do we linearize unsuccessful Search?

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

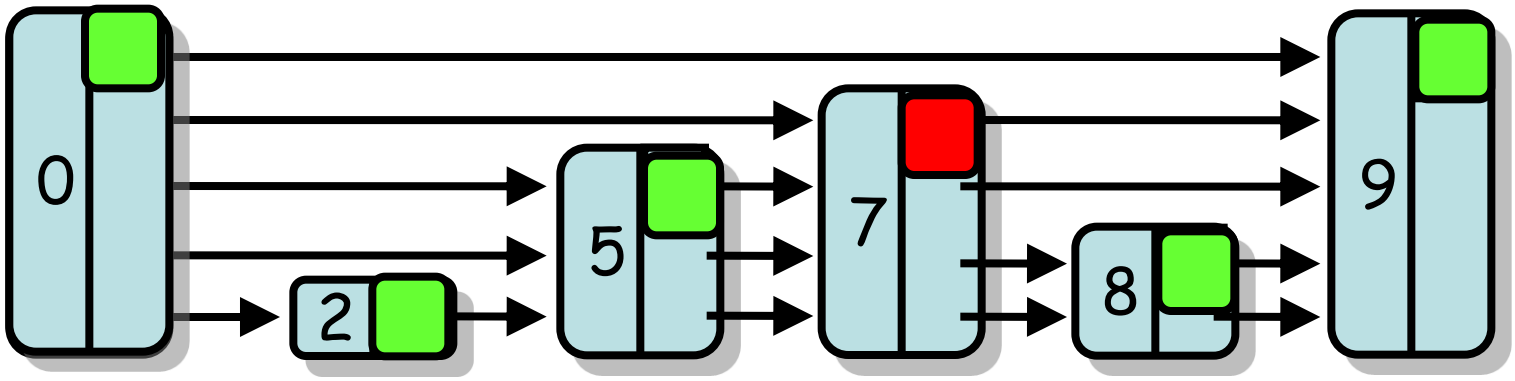


Look-Free Skip List

- We can't manipulate multiple reference at the same time
  - Can't maintain the SkipList property
  - Each list could be not a sublist of the list at levels below
- The Abstract set is defined by the bottom-level
  - A key is in the set if there is a node with that key whose next reference is unmarked in the bottom level list

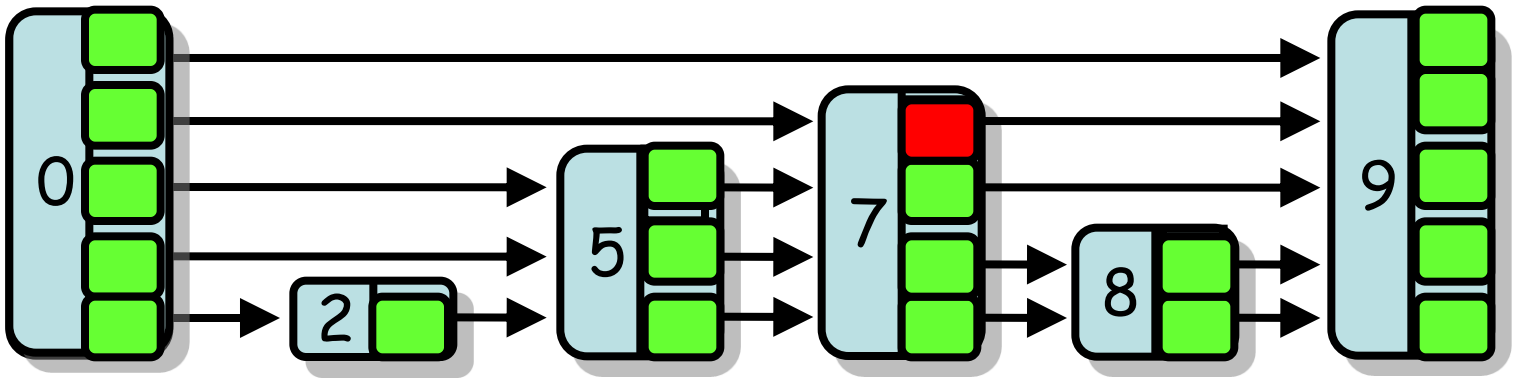
LockFree Skip List

- Use a mark bit for logical deletion
- Did I mark it ?



LockFree Skip List

- Use a mark bit for logical deletion
- Use a mark bit for each linked list



How to Add/Remove ?

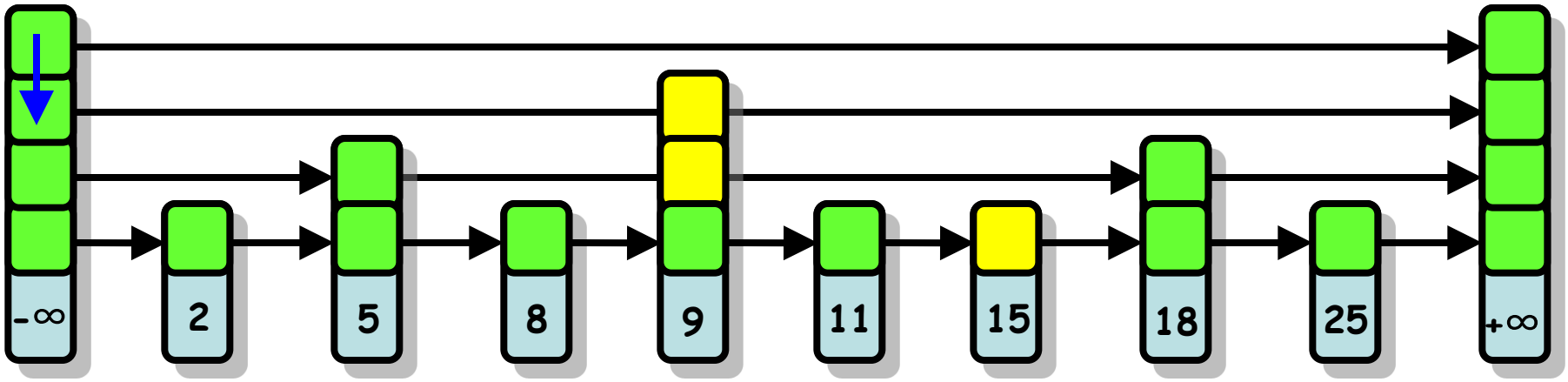
- Each level of the list is treated as a LockFree list
- Use CAS() to insert
- Mark the next reference to remove
- find() method cleans up marked nodes
  - It's possible for a node to be physically removed while it's being linked at lower level

Add()

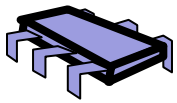
- Call find()
  - to determine whether a node is already in the list, find its set of predecessors and successors
- A new node is prepared
  - with randomly chosen topLevel, and its next references are directed to the potential successors
- Add to the bottom level LockFreeList
  - Successively links the node in higher levels

add(12)

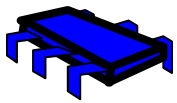
find(12)



Remove(2)

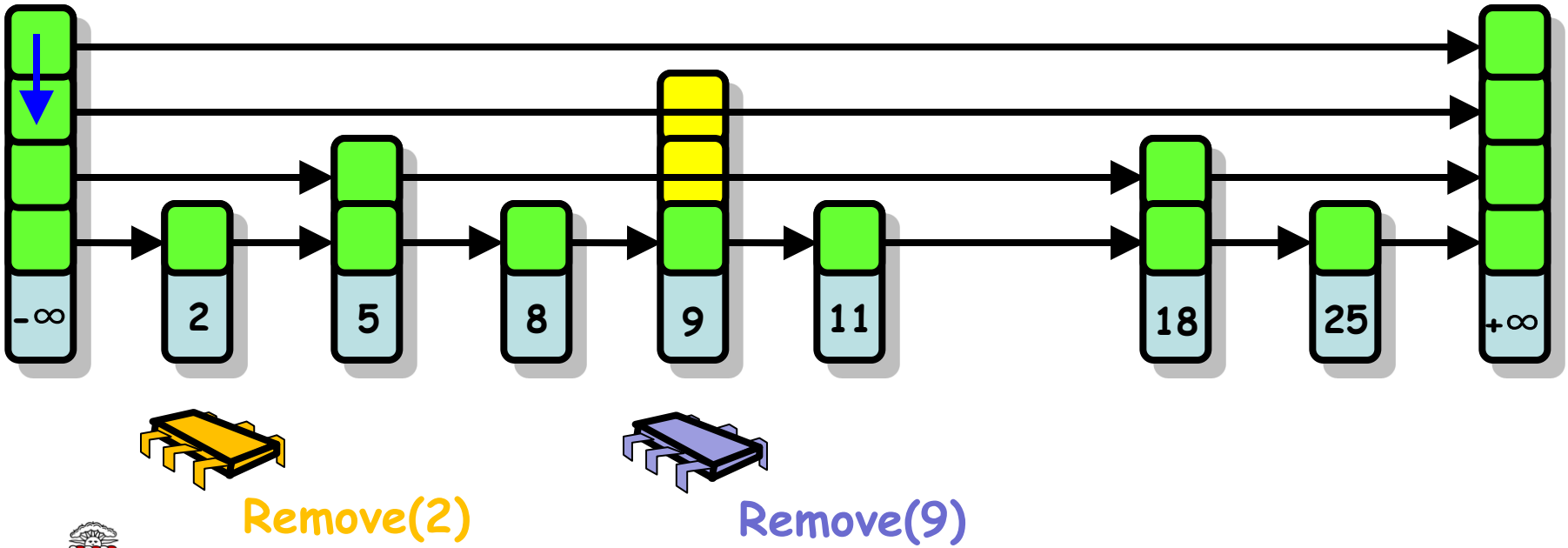


Remove(9)

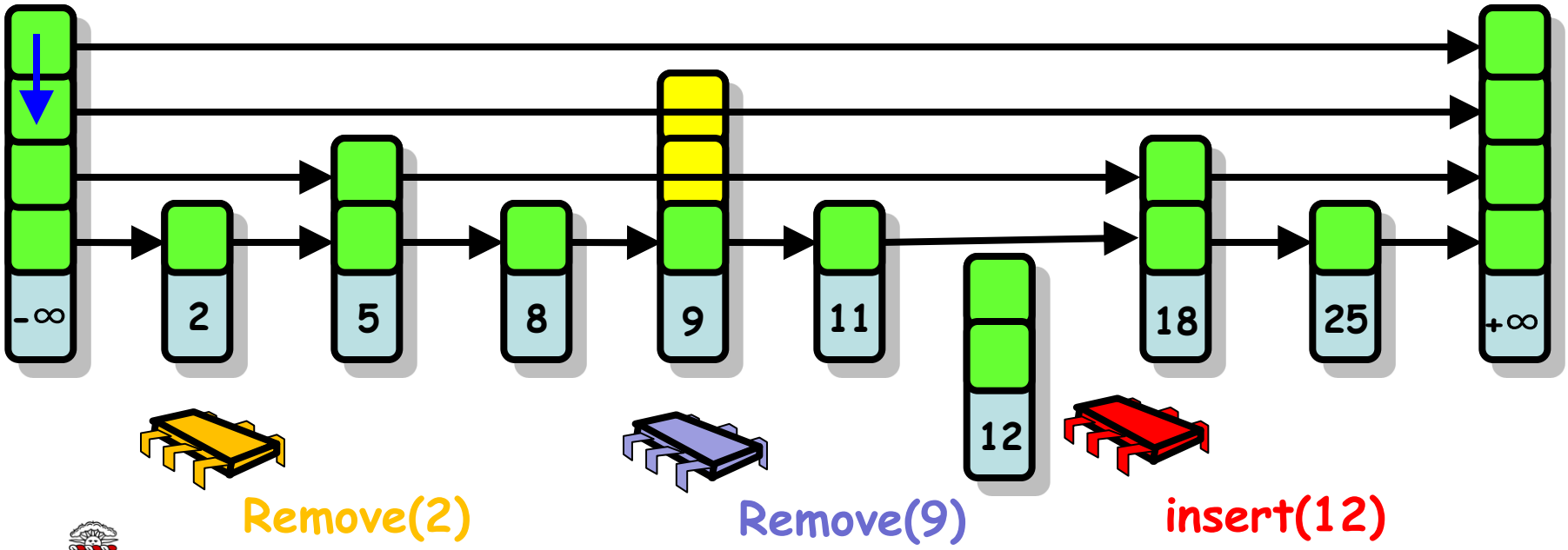


Remove(15)

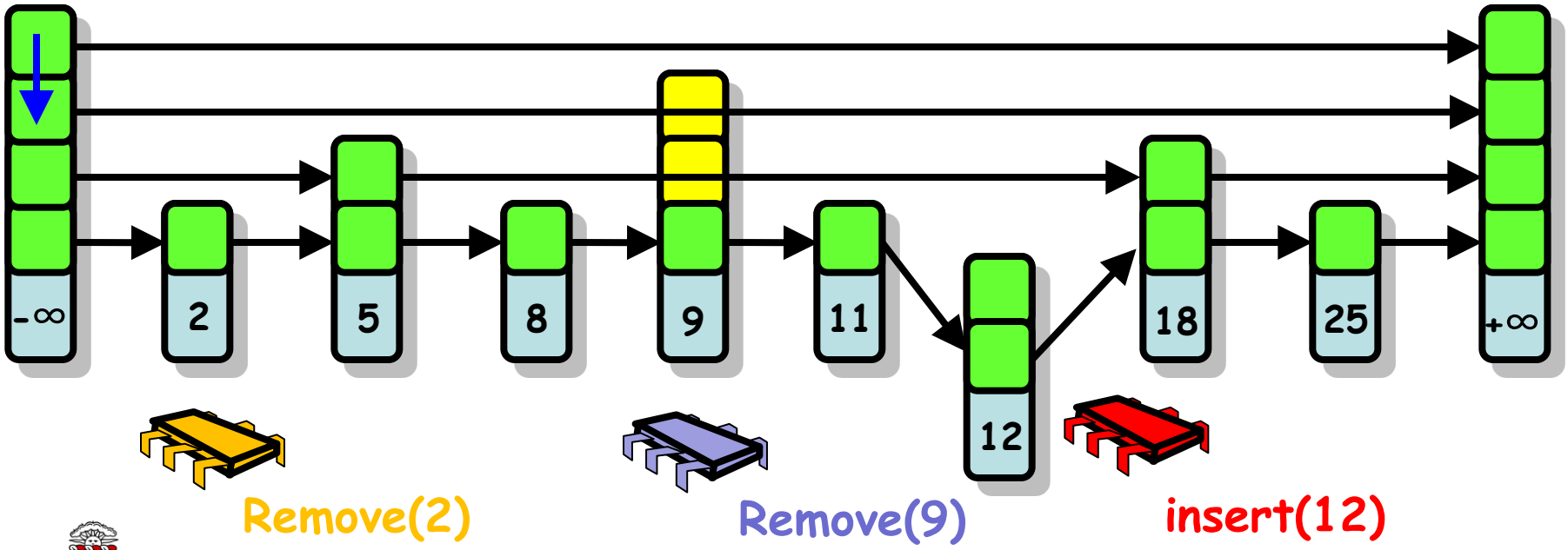
add(12)



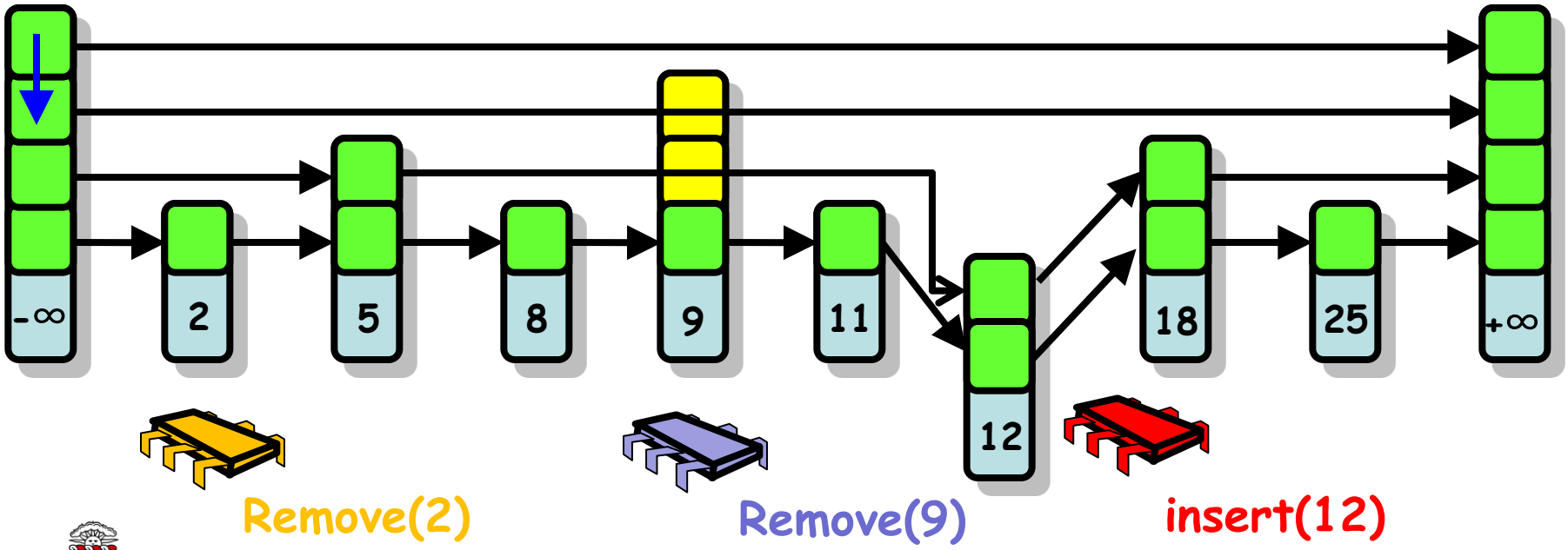
add(12)



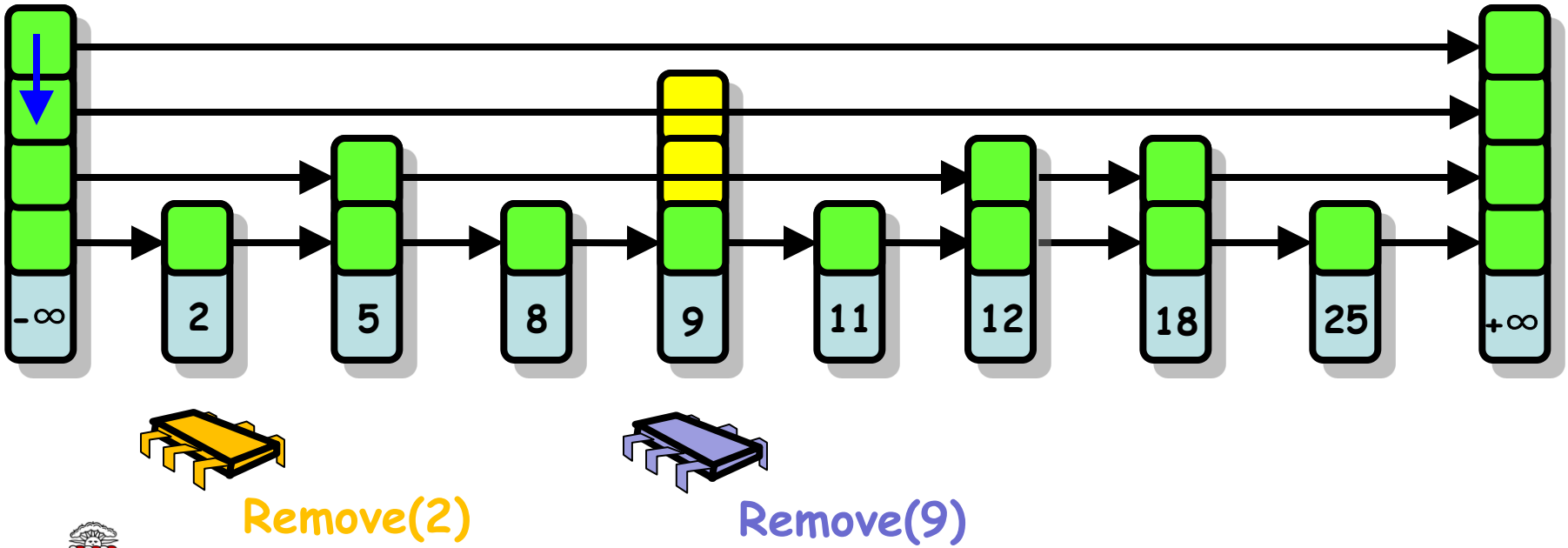
add(12)



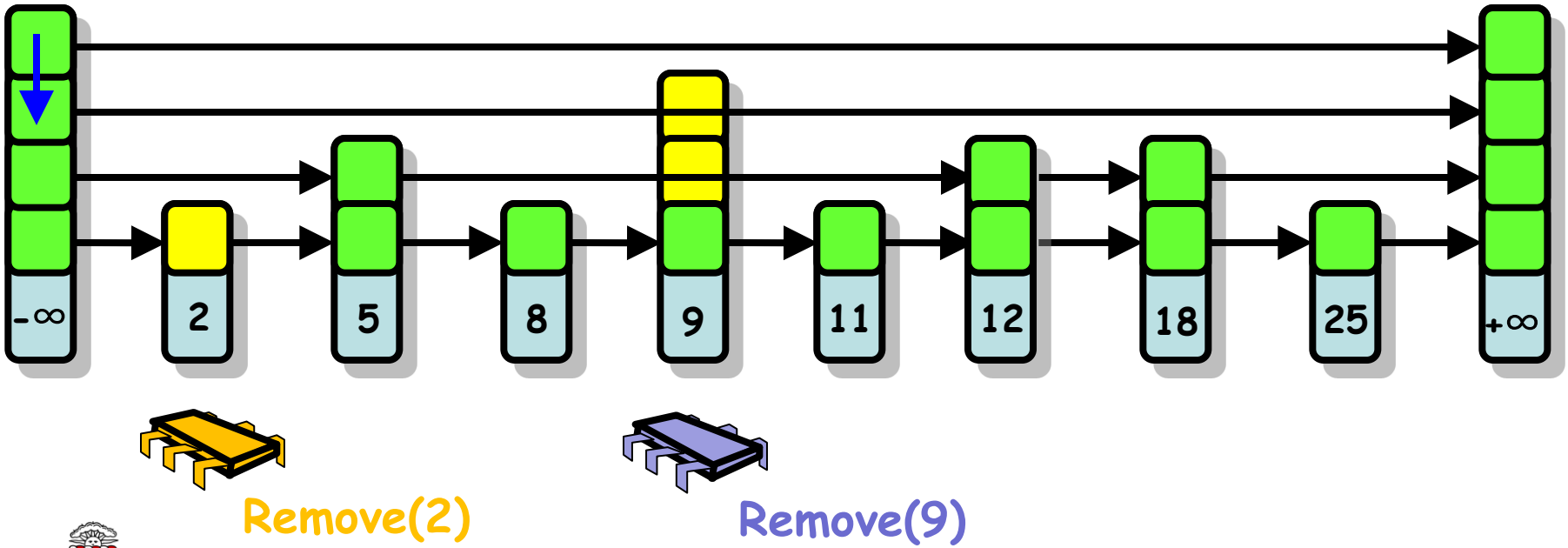
add(12)



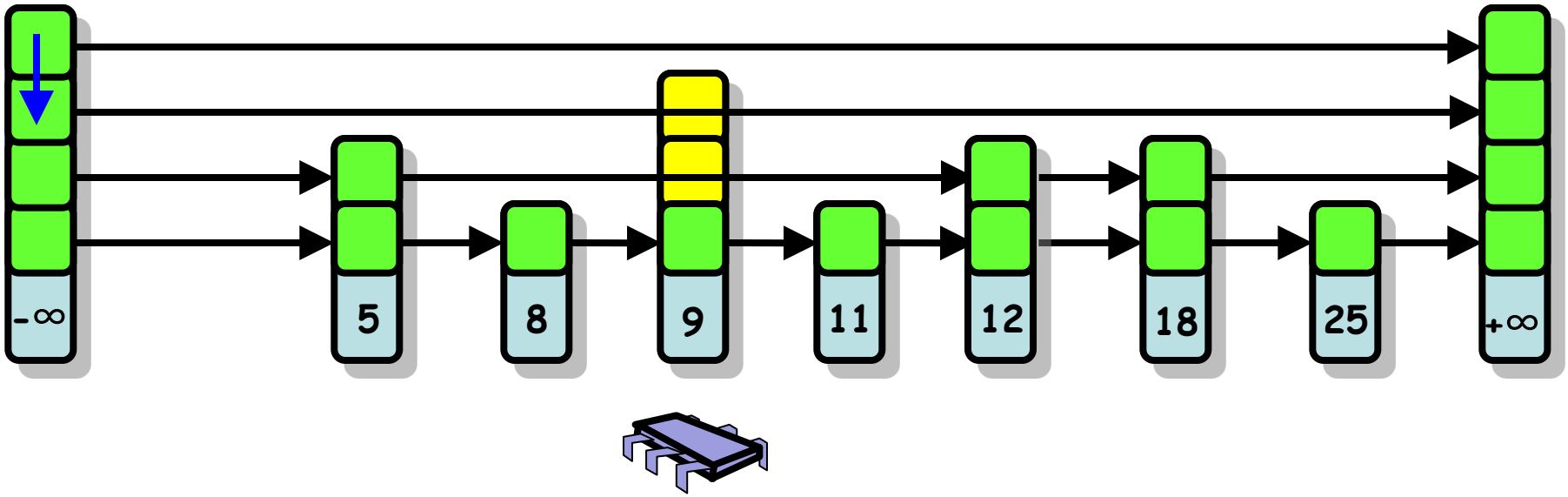
add(12)



add(12)



add(12)



Remove(9)

Remove()

- Call find()
  - to determine whether an unmarked node with the target key is in the bottom level list
- Logical remove
  - mark from the top level to the bottom level
- physical remove
  - is done either by itself or by the other find() traversing the link

contains()

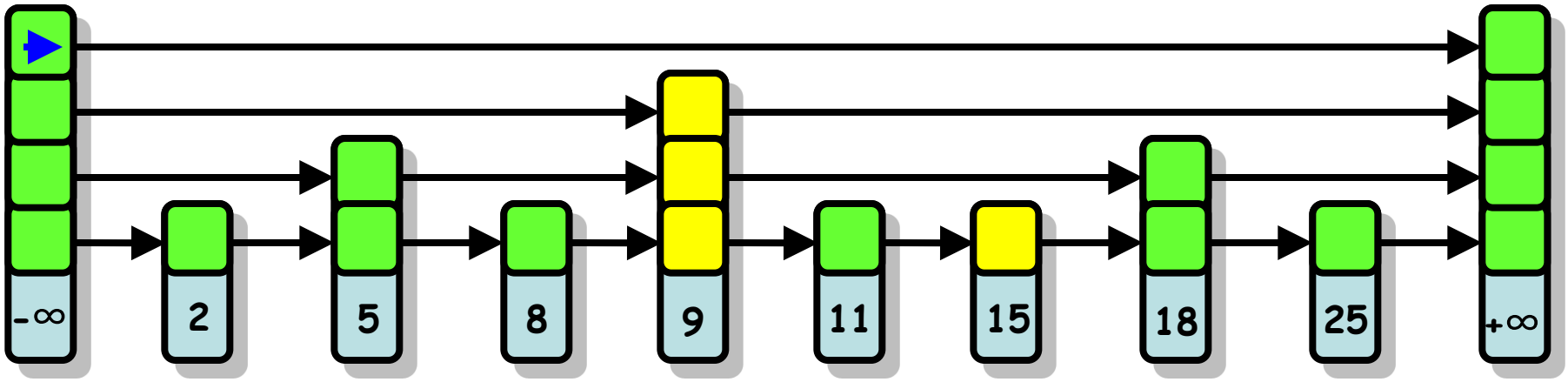
- Can't use find()
  - trying to remove marked nodes might generate too much contention
- Can't use LockFreeList's contains()
  - might skip nodes reachable from the bottom level
- Use find()
  - without actually doing physical remove

contains(18)

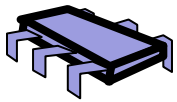
Contain(18)

current

prev



Remove(9)



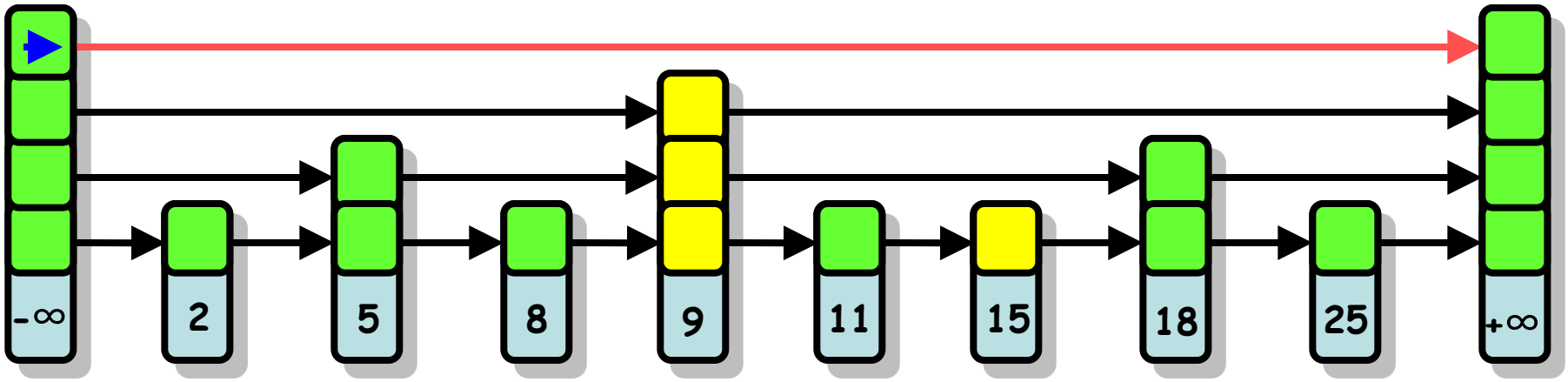
Remove(15)

contains(18)

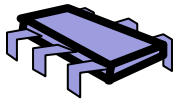
Contain(18)

current

prev



Remove(9)



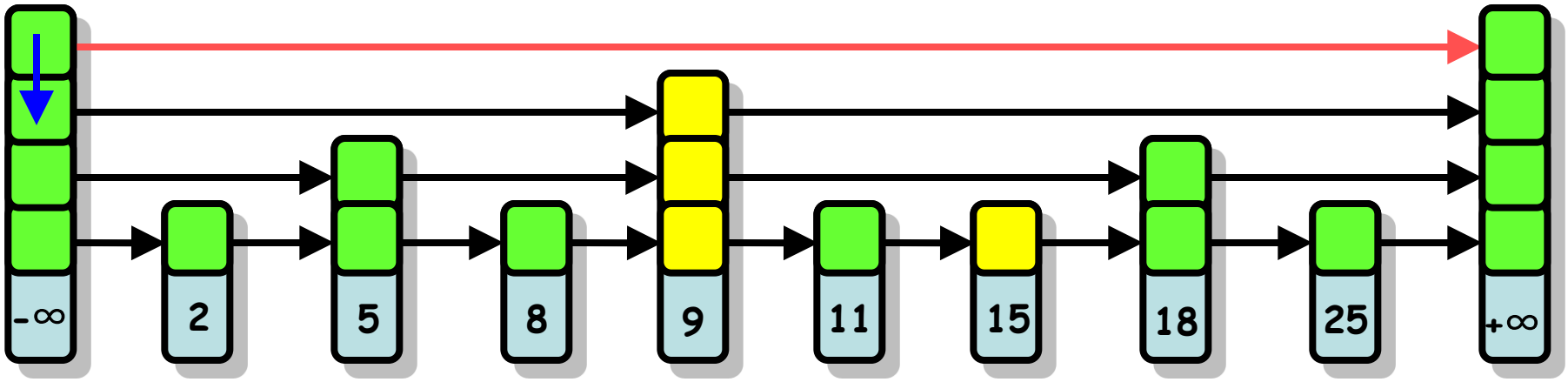
Remove(15)

contains(18)

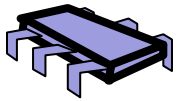
Contain(18)

current

prev

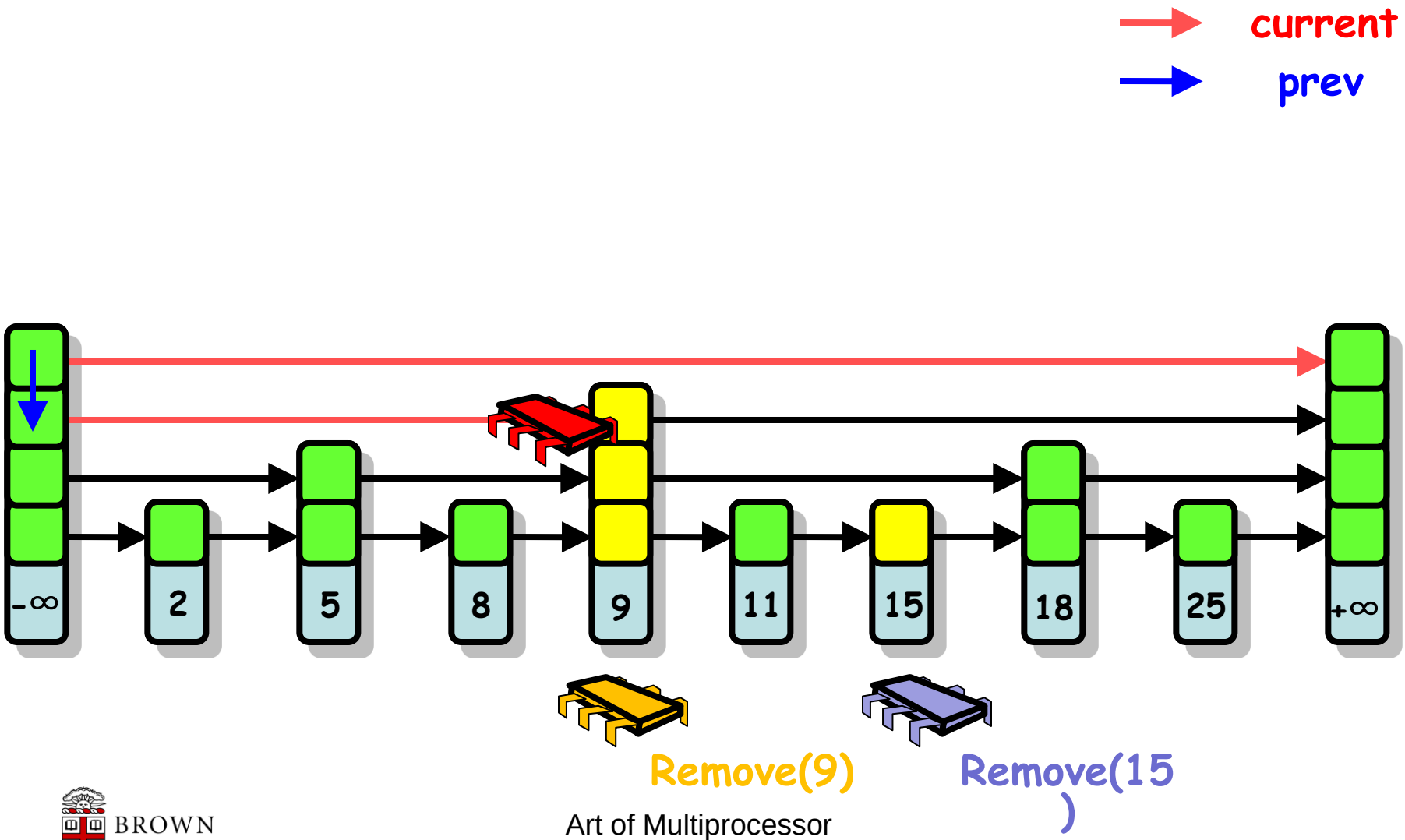


Remove(9)



Remove(15)

contains(18)

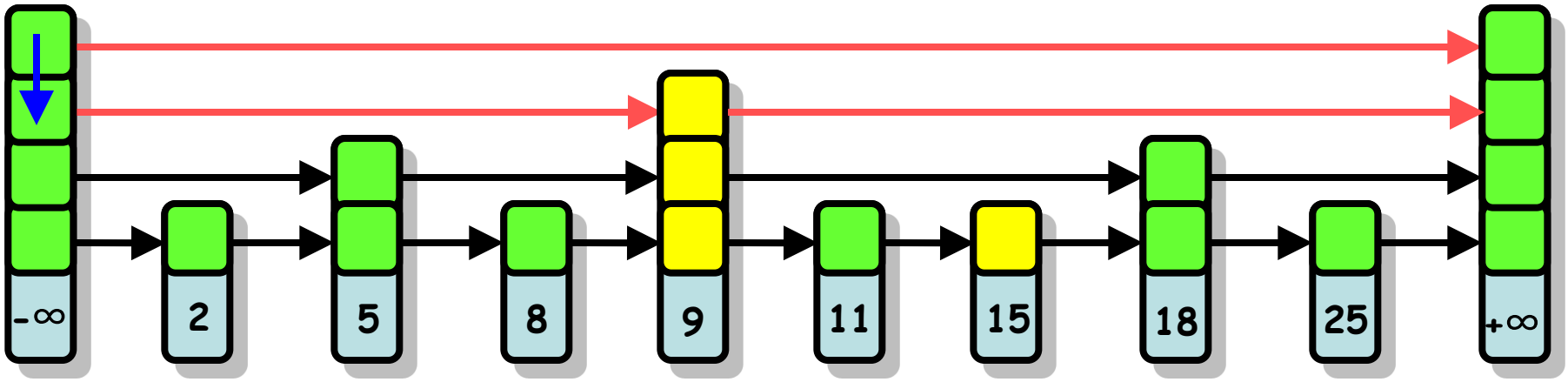


contains(18)

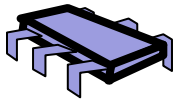
Contain(18)

current

prev



Remove(9)



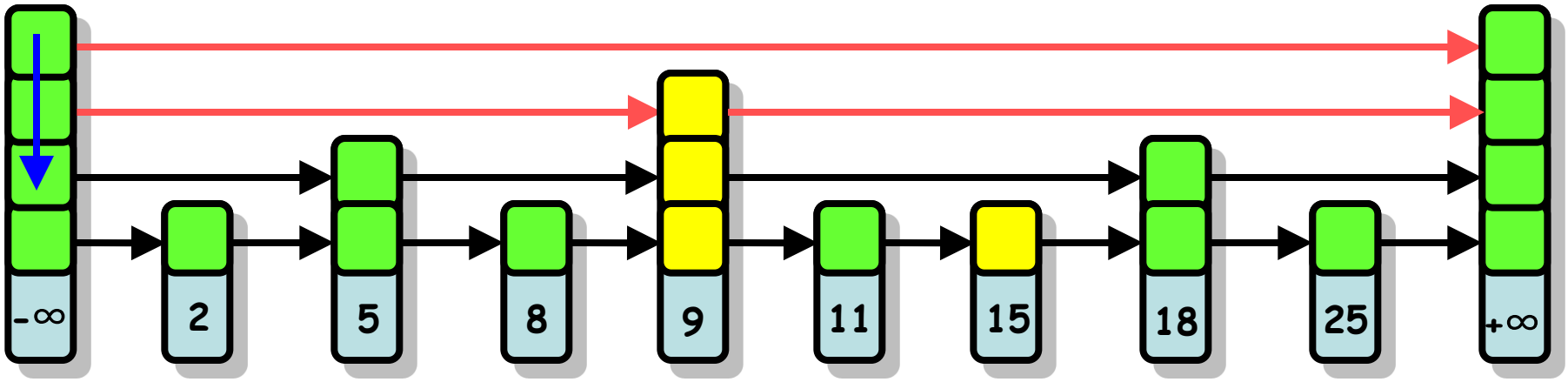
Remove(15)

contains(18)

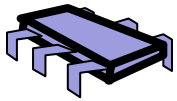
Contain(18)

current

prev

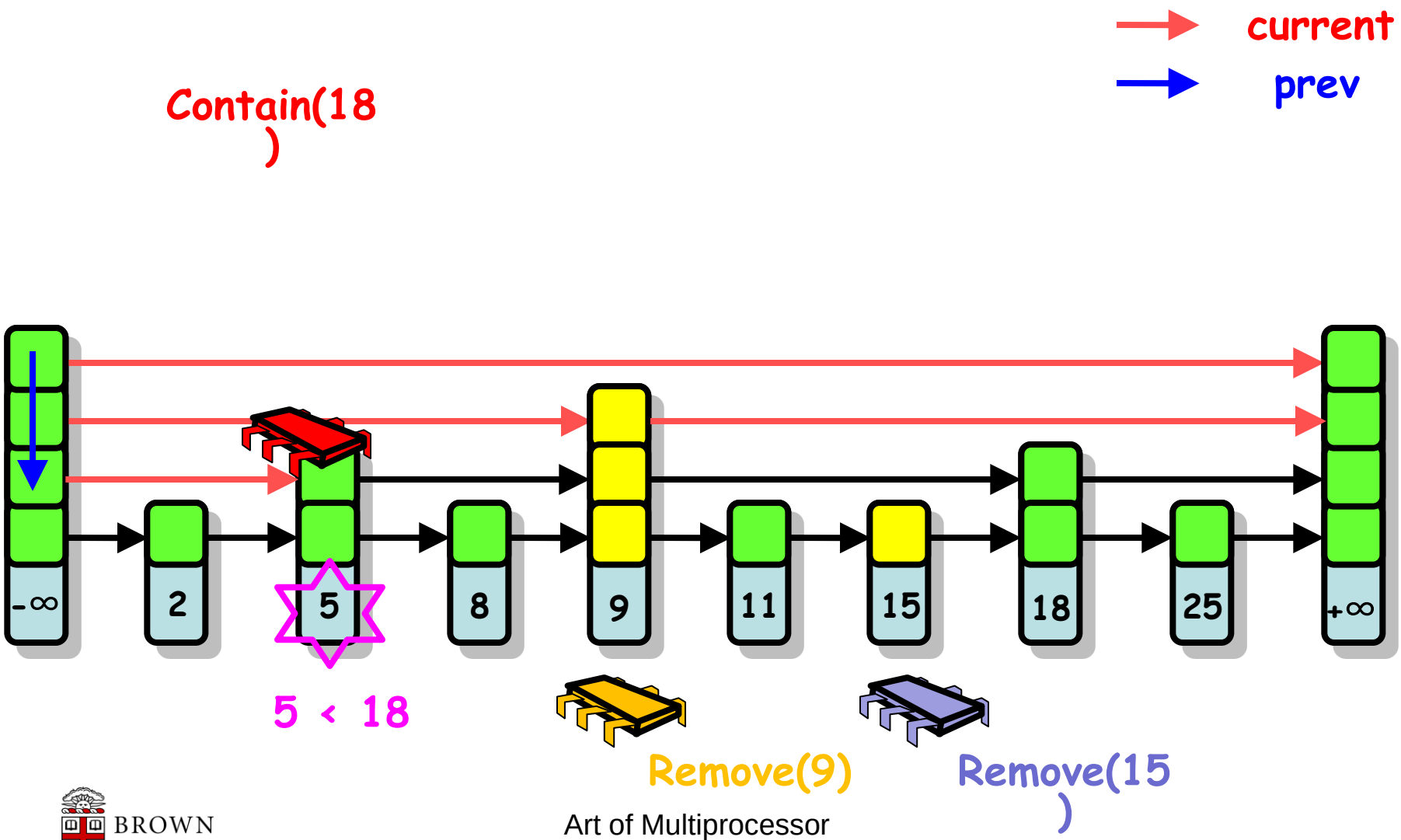


Remove(9)

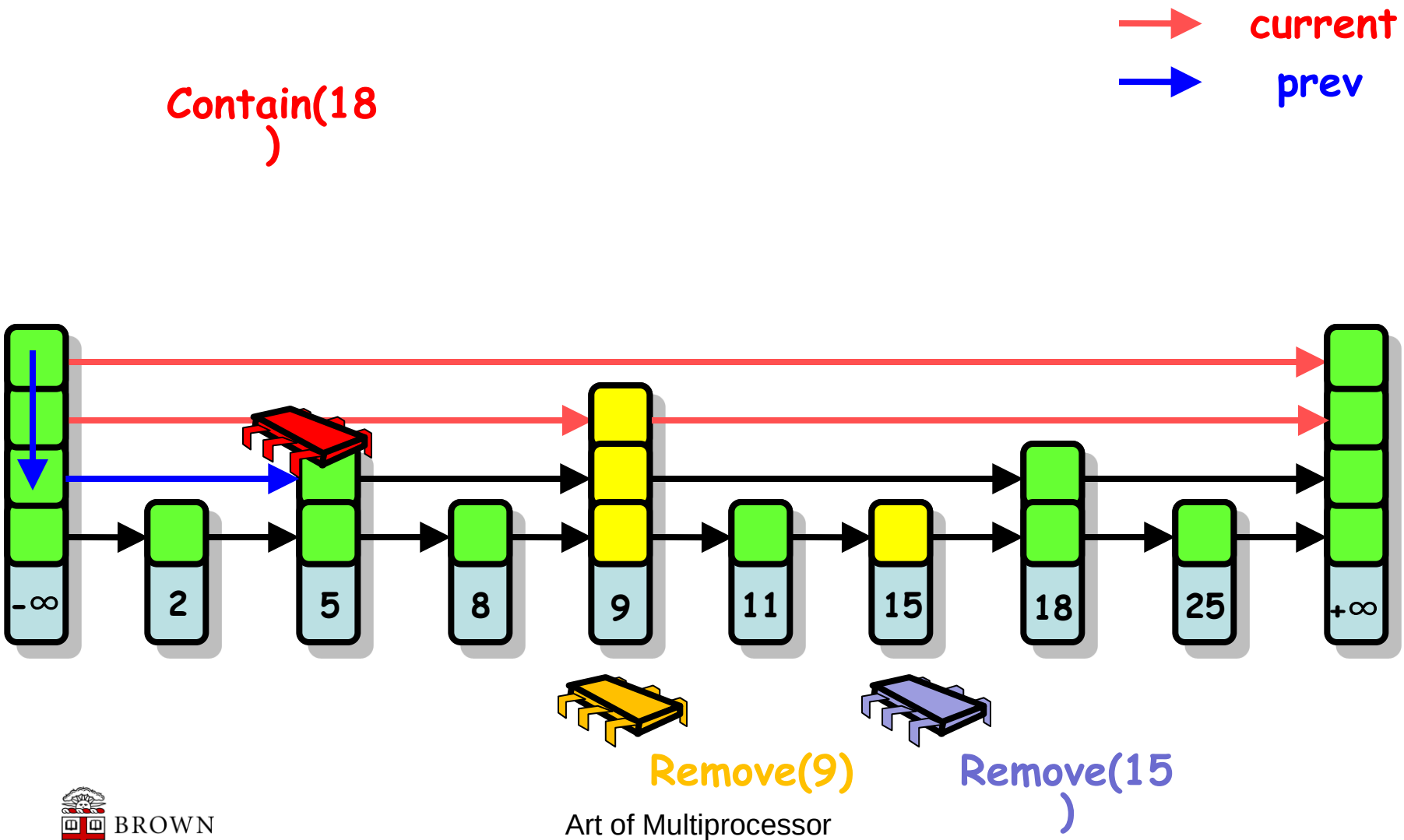


Remove(15)

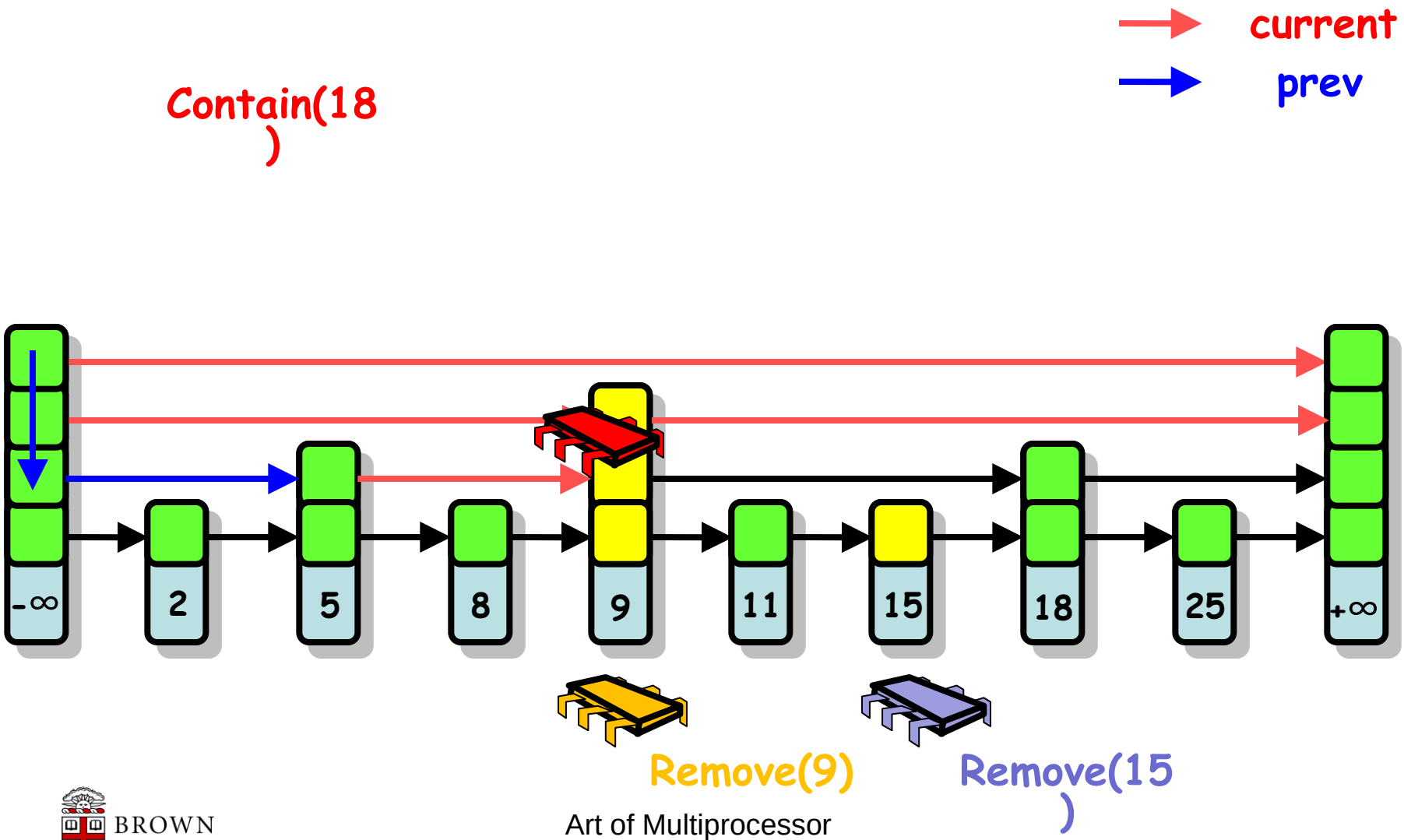
contains(18)


current
prev
Contain(18
)
-∞
2
5
8
9
11
15
18
25
+∞
5 < 18
Remove(9)
Remove(15
)

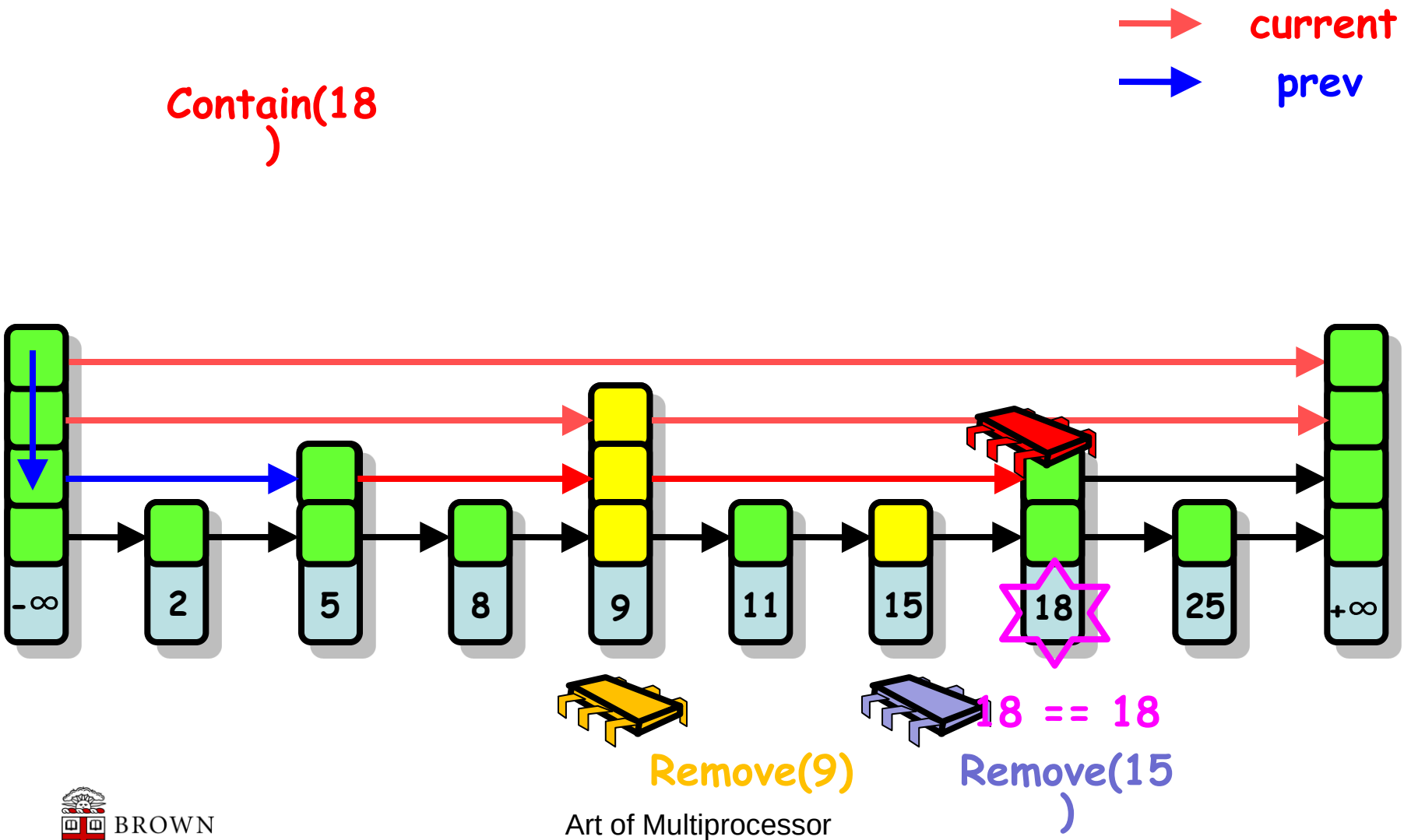
contains(18)



contains(18)



contains(18)

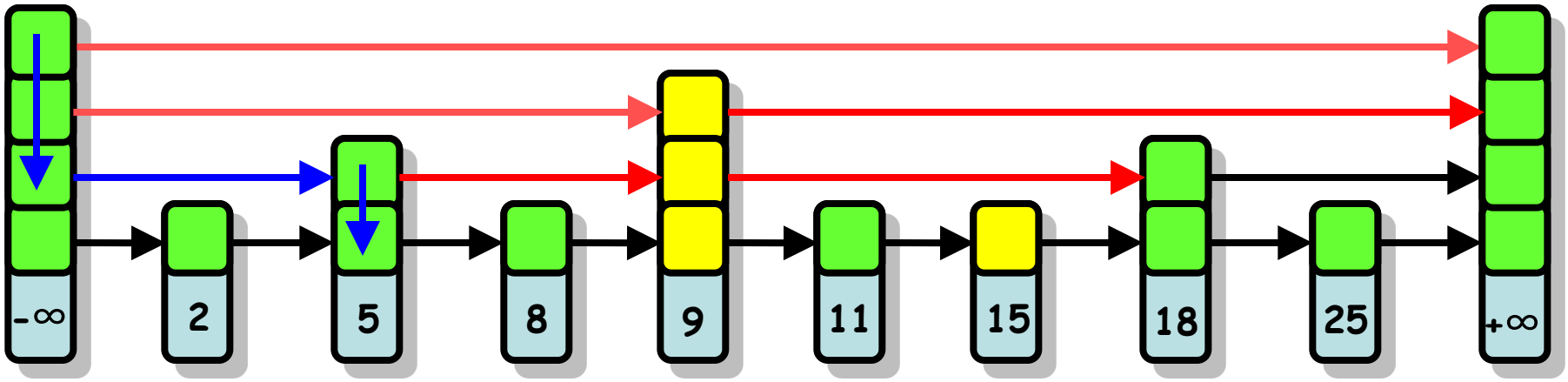

current
prev
Contain(18
)
-∞
2
5
8
9
11
15
18
25
+∞
18 == 18
Remove(9)
Remove(15
)

contains(18)

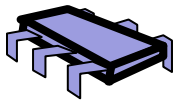
Contain(18)

current

prev

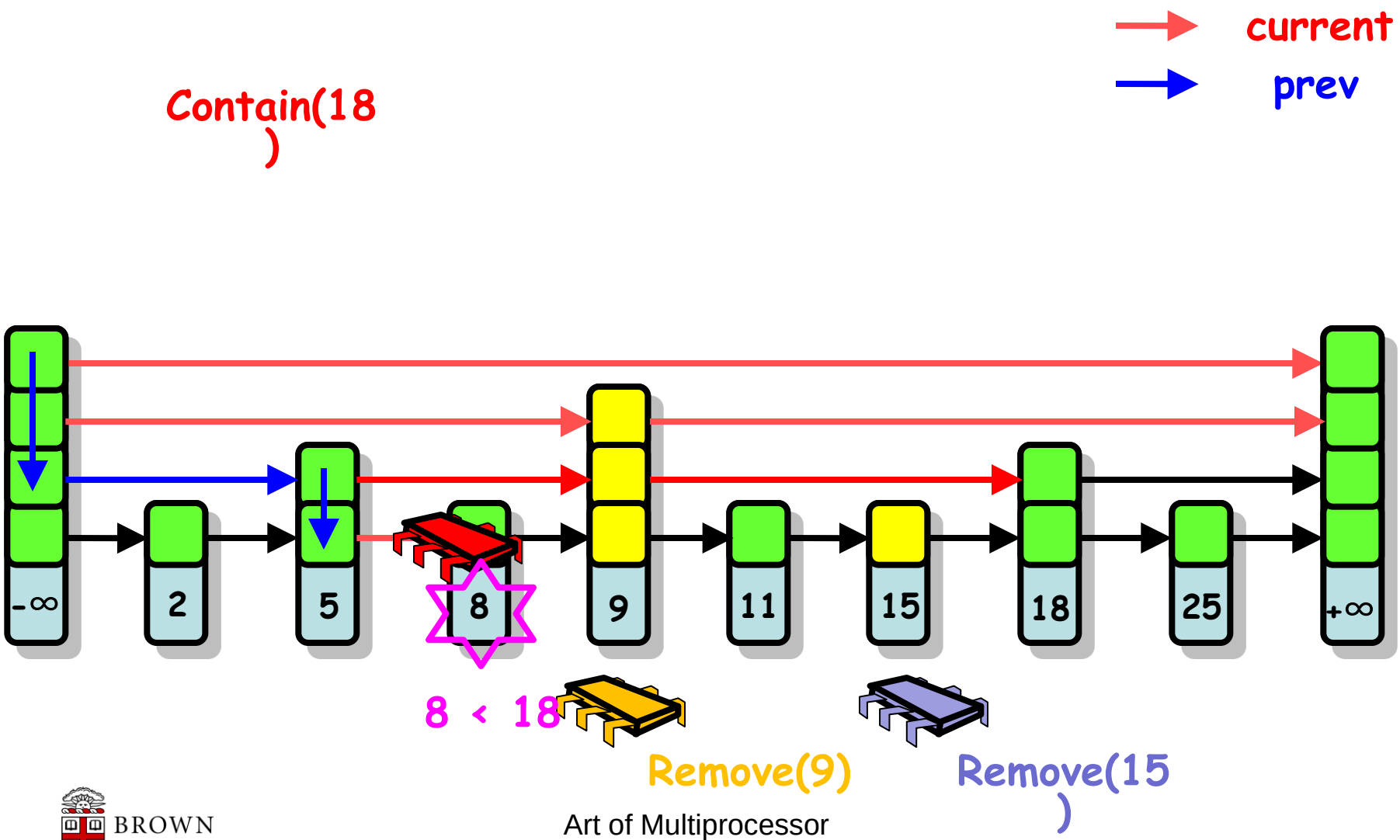


Remove(9)

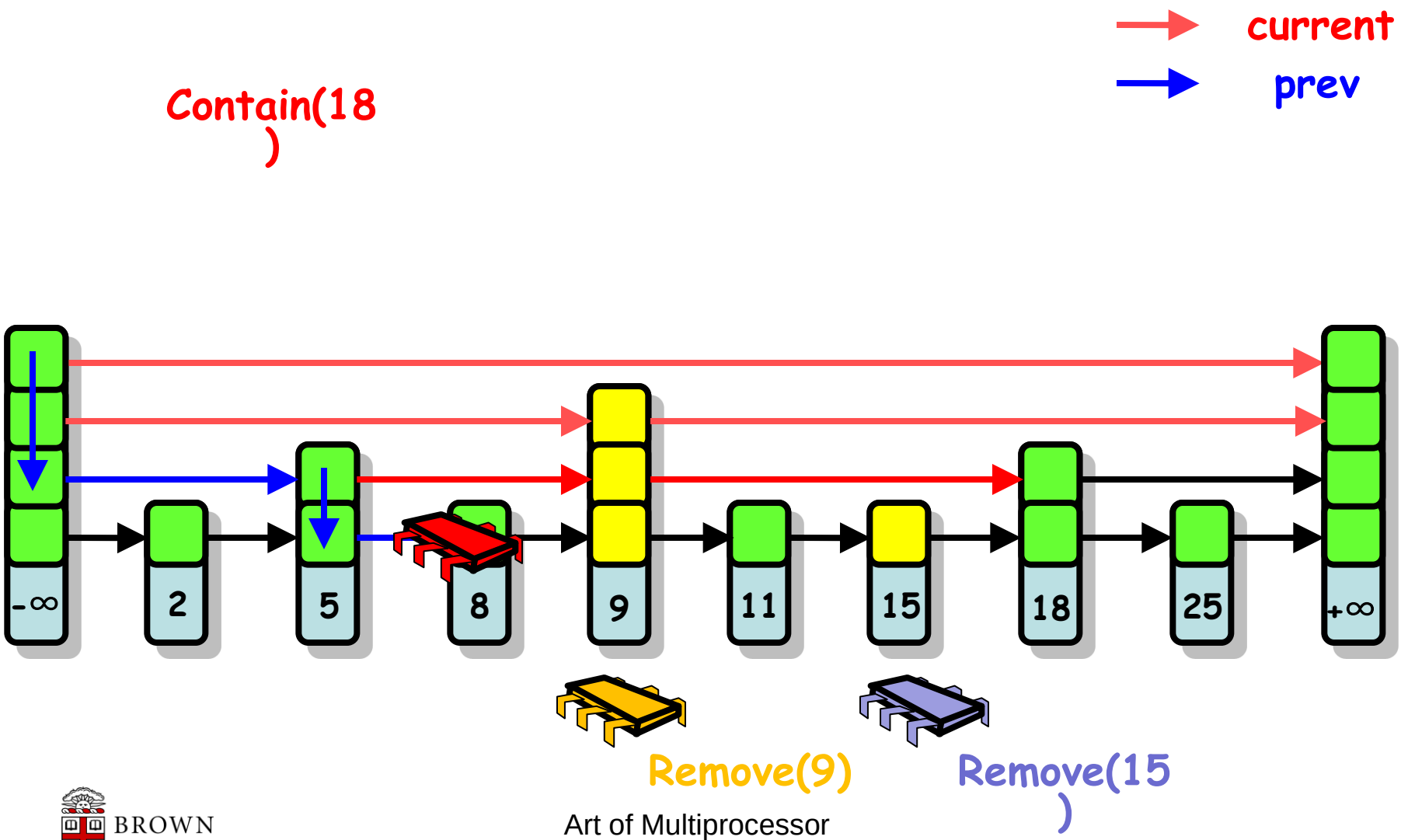


Remove(15)

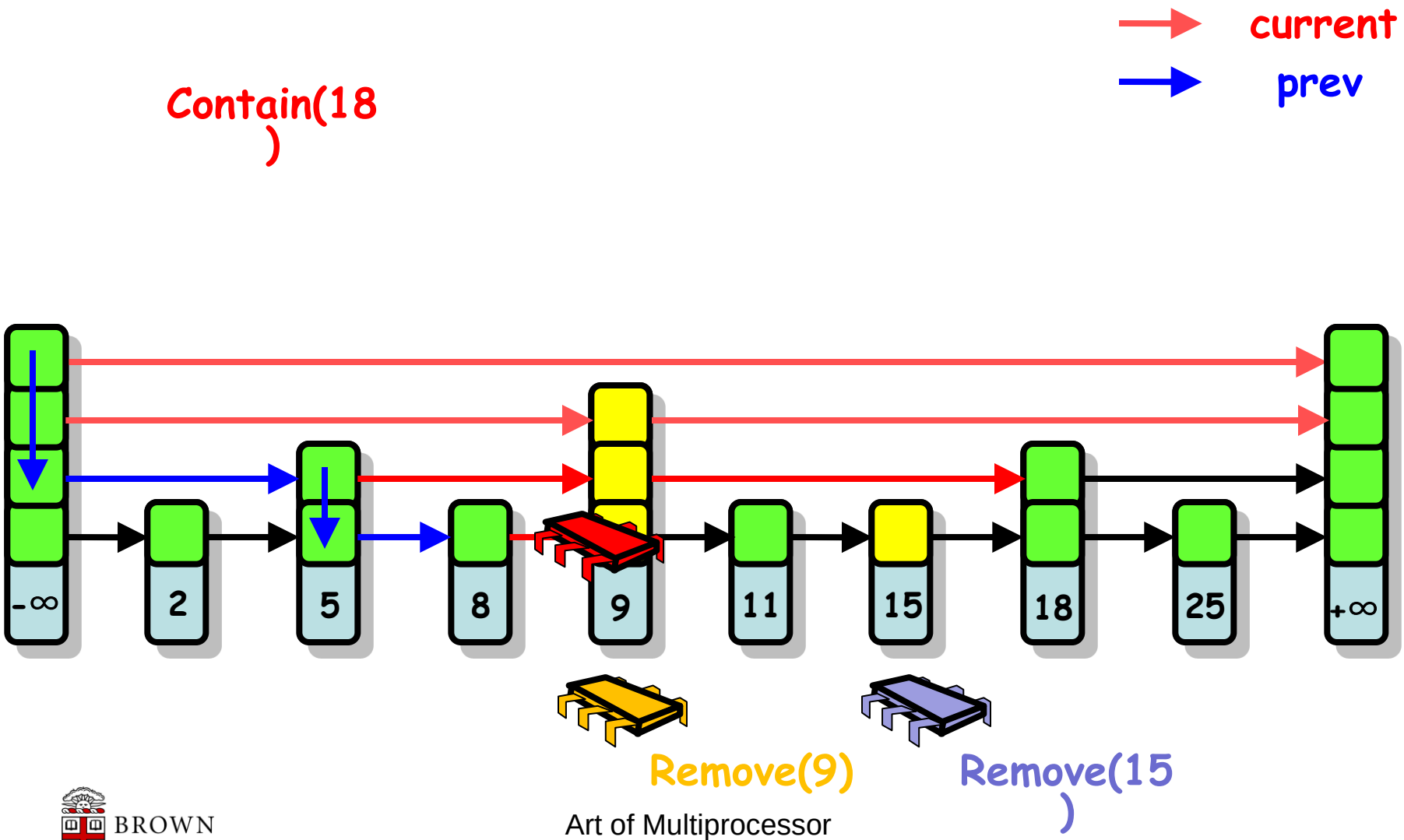
contains(18)


current
prev
Contain(18
)
-∞
2
5
8
9
11
15
18
25
+∞
8 < 18
Remove(9)
Remove(15
)

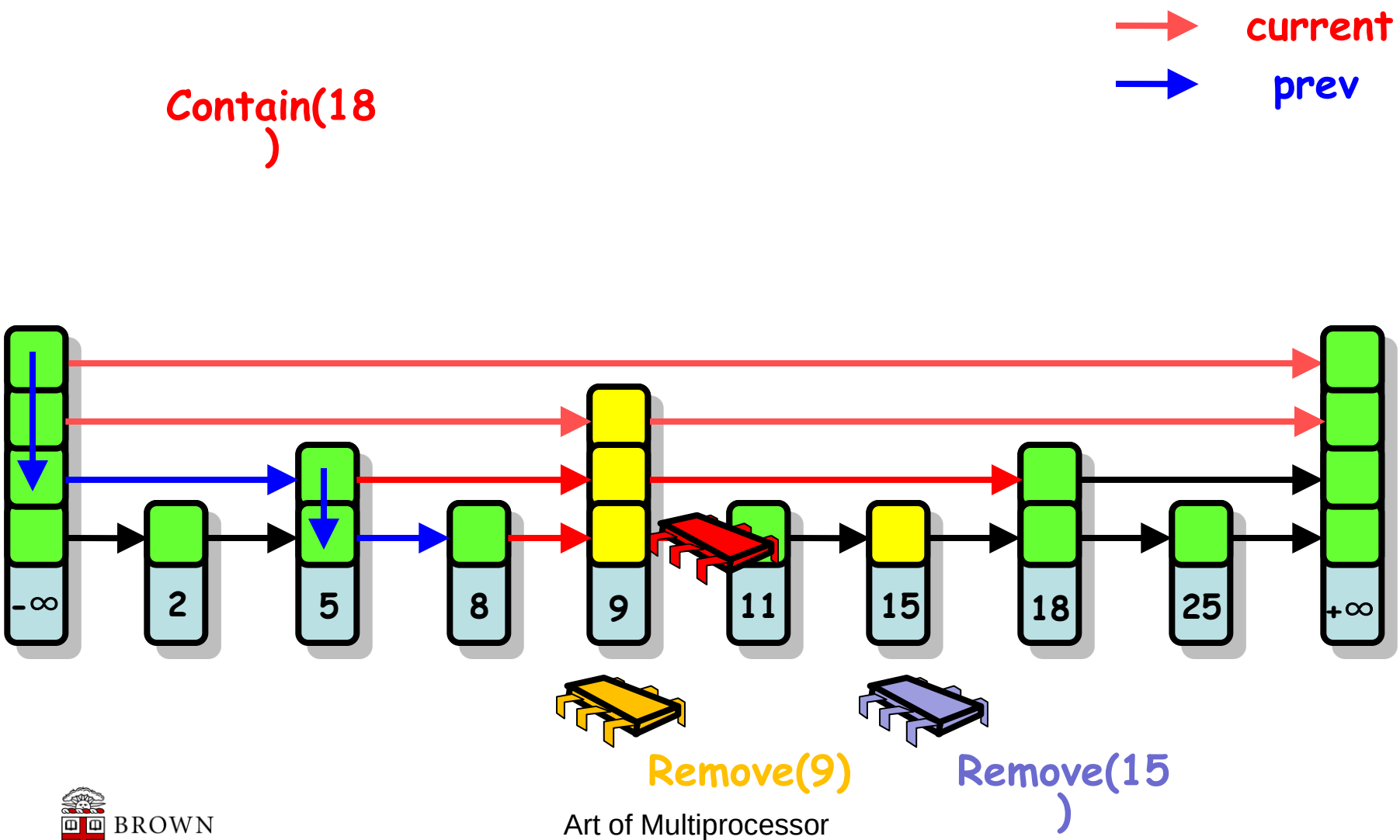
contains(18)



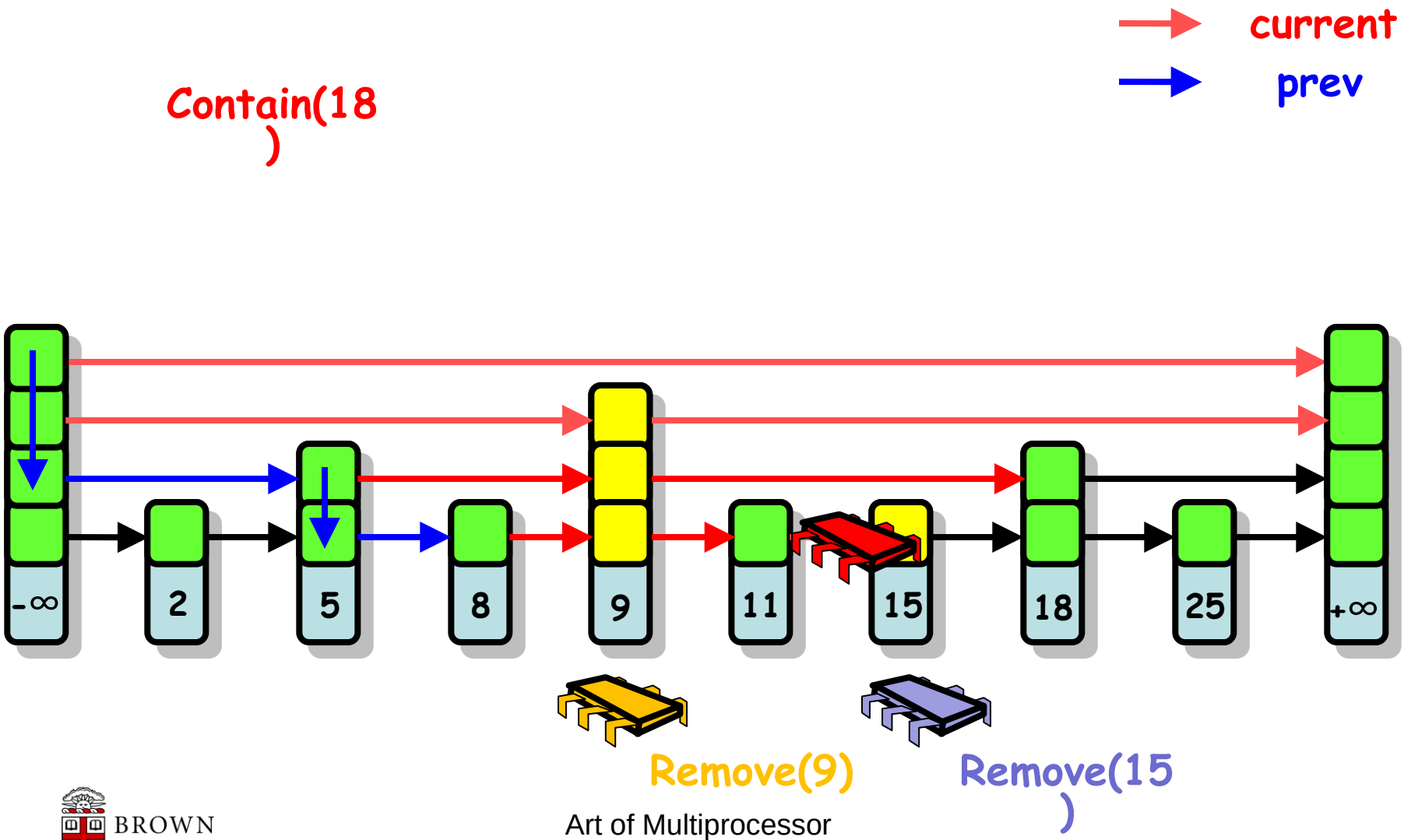
contains(18)



contains(18)



contains(18)

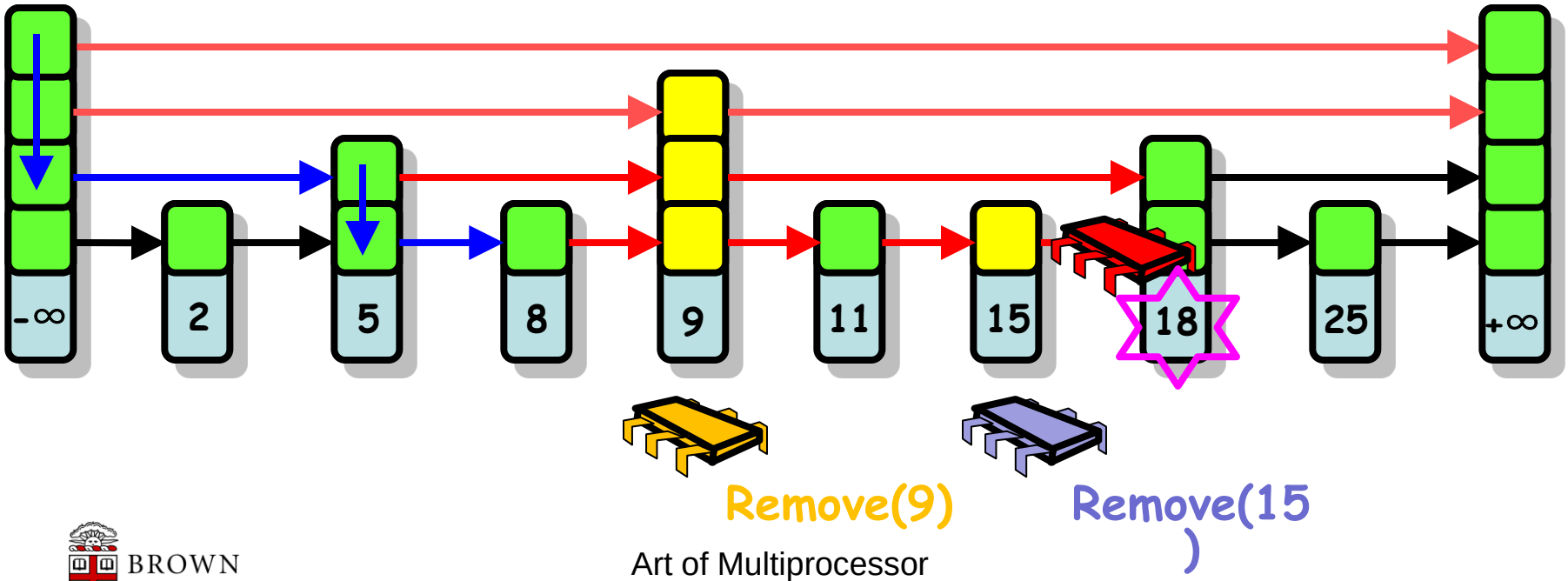


contains(18)

current

prev

Contain(18) returns TRUE



When is contains() linearized ?

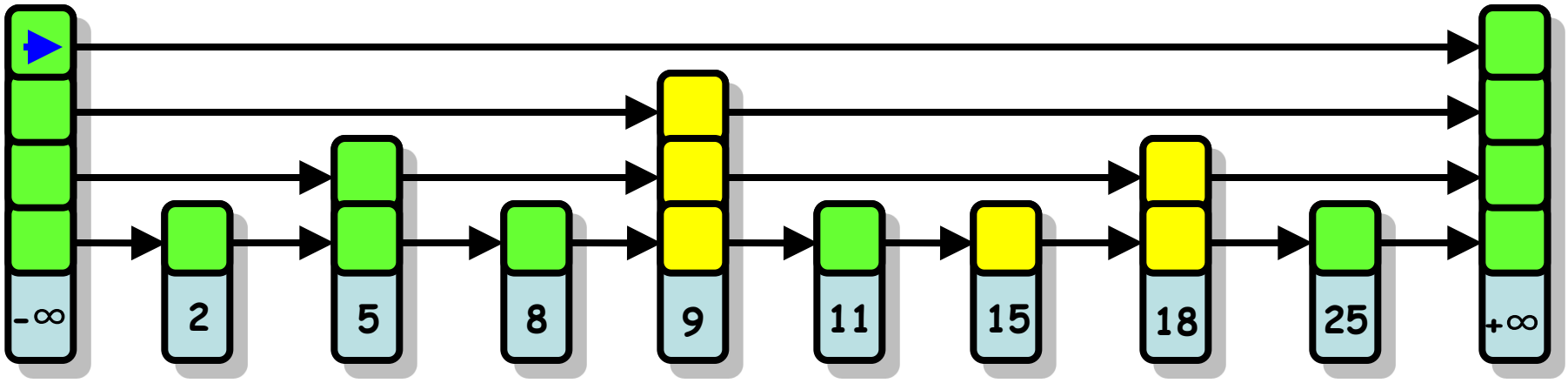
- It might be possible contains() returns false even though the item has been inserted !

contains(18)

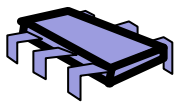
Contain(18)

current

prev



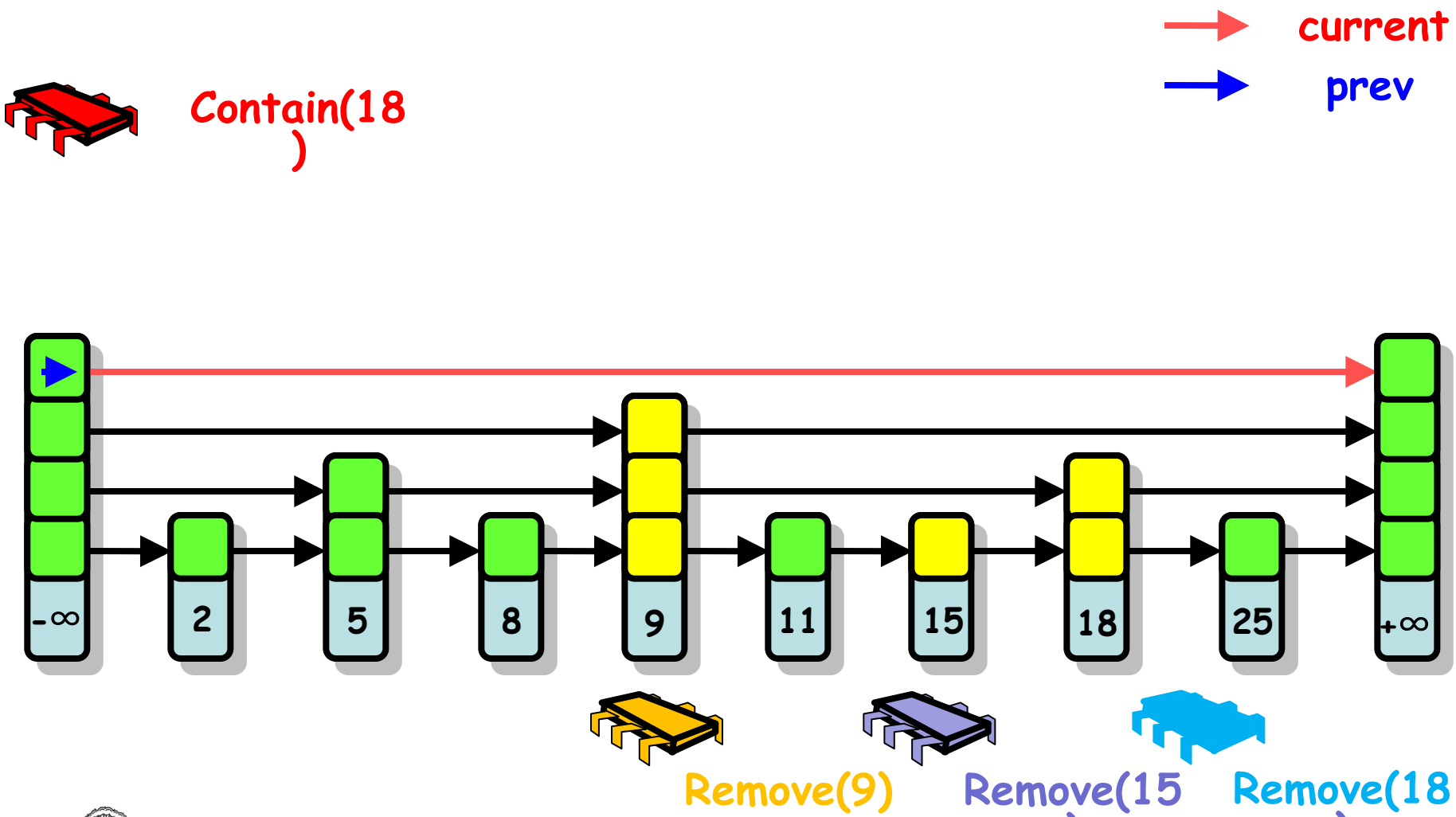
Remove(9)



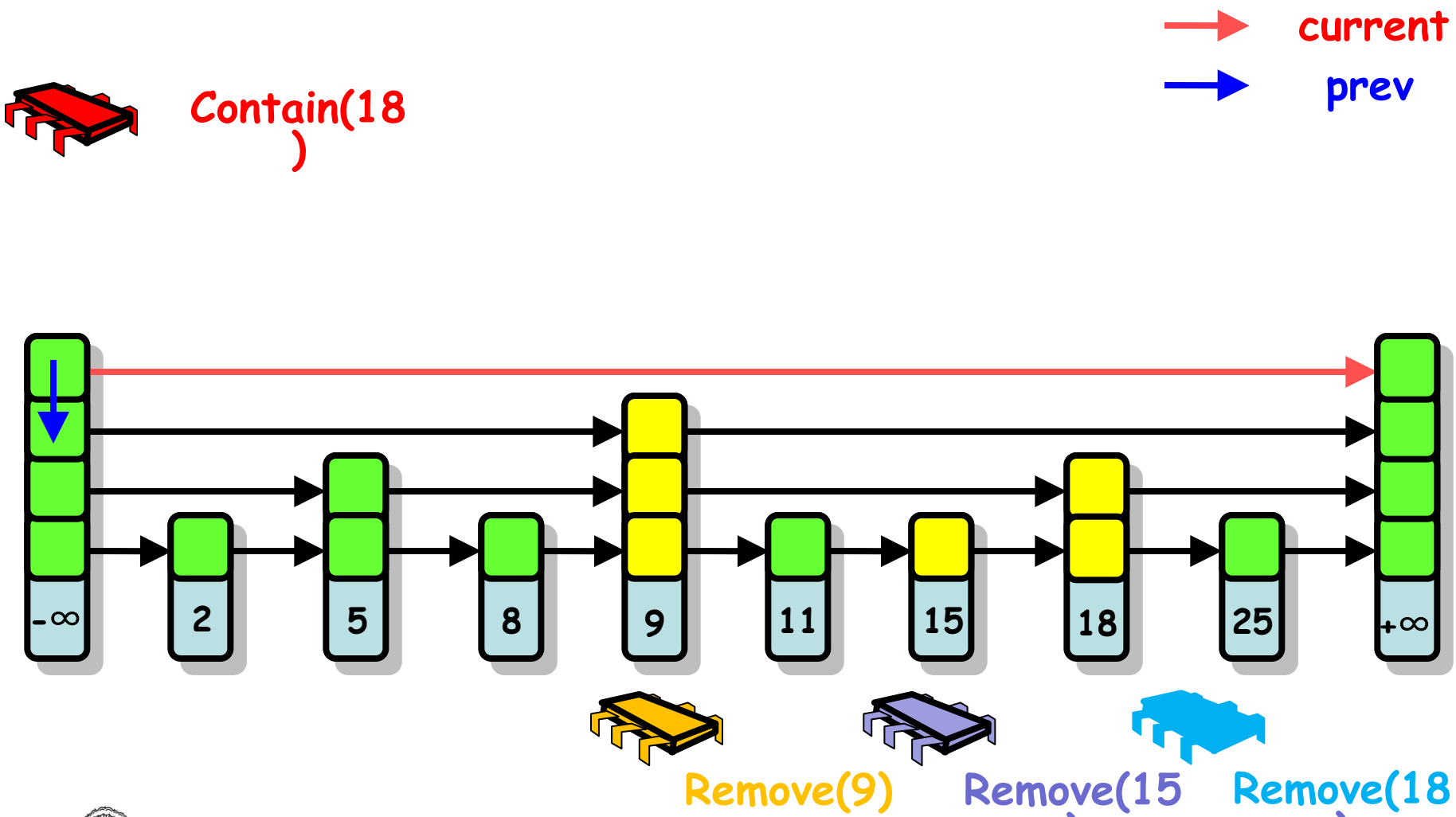
Remove(15)

Remove(18)

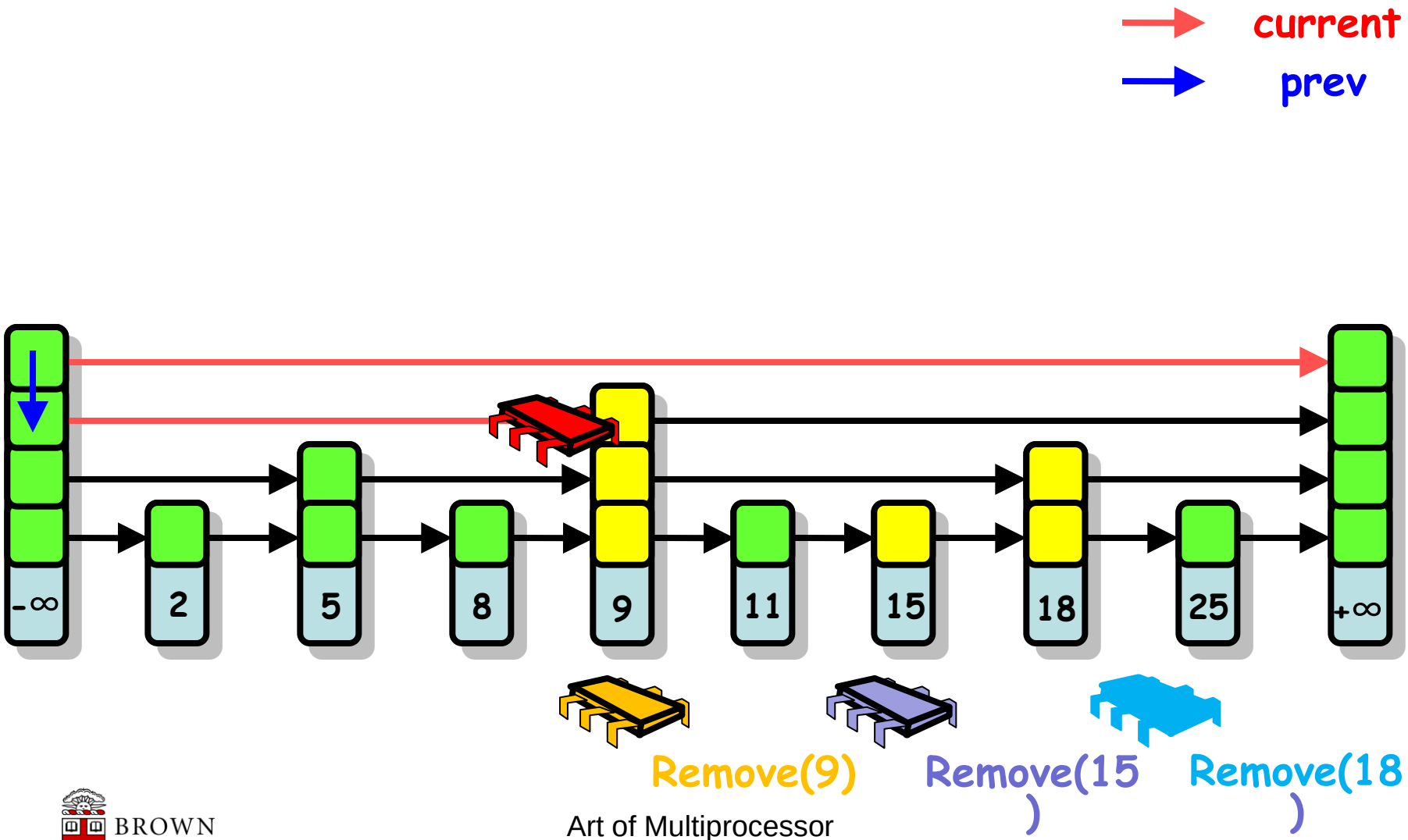
contains(18)


current
prev
Contain(18)
-∞
2
5
8
9
11
15
18
25
+∞
Remove(9)
Remove(15)
Remove(18)

contains(18)


current
prev
Contain(18)
-∞
2
5
8
9
11
15
18
25
+∞
Remove(9)
Remove(15)
Remove(18)

contains(18)

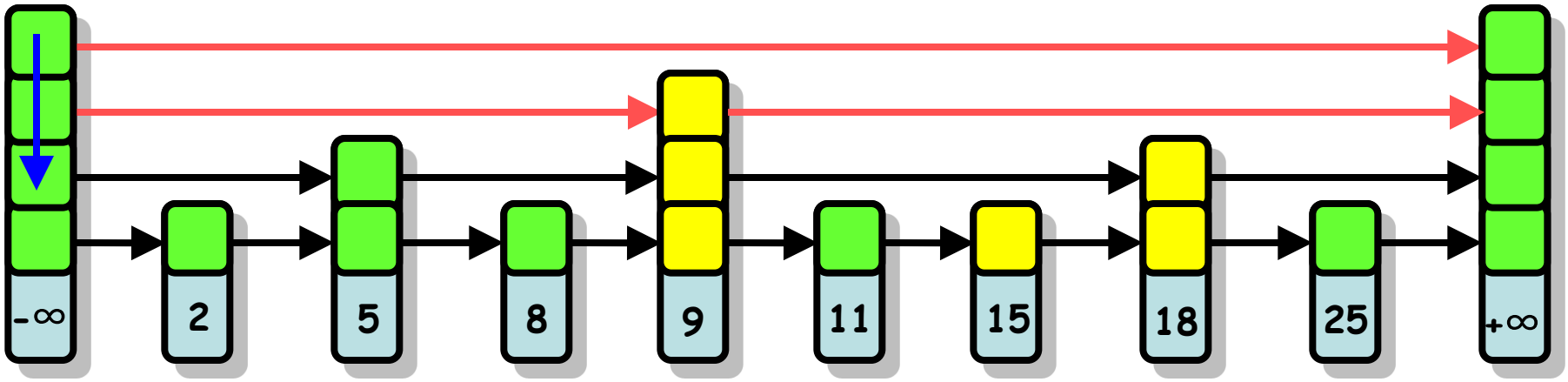


contains(18)

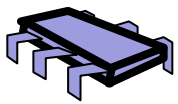
Contain(18)

current

prev



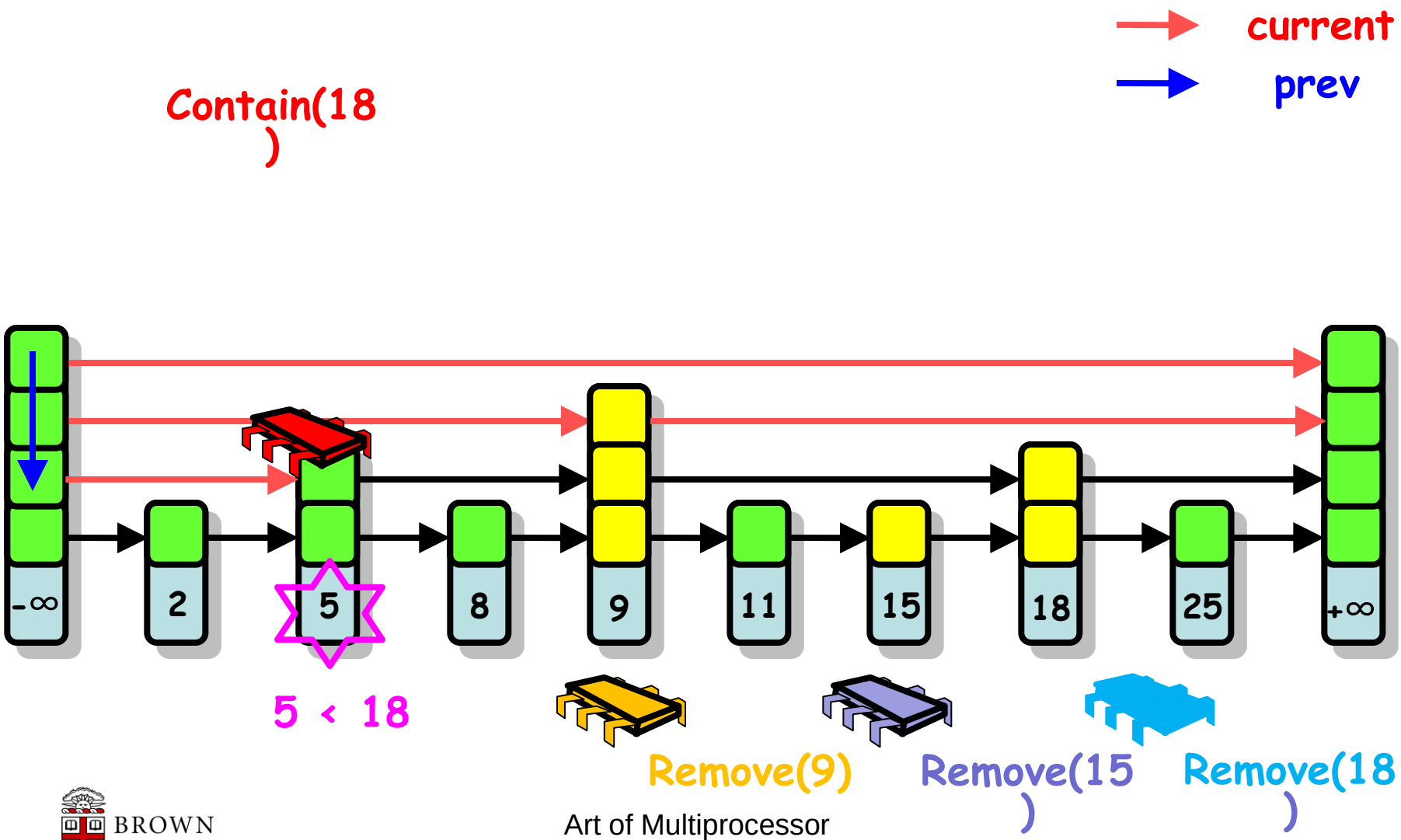
Remove(9)



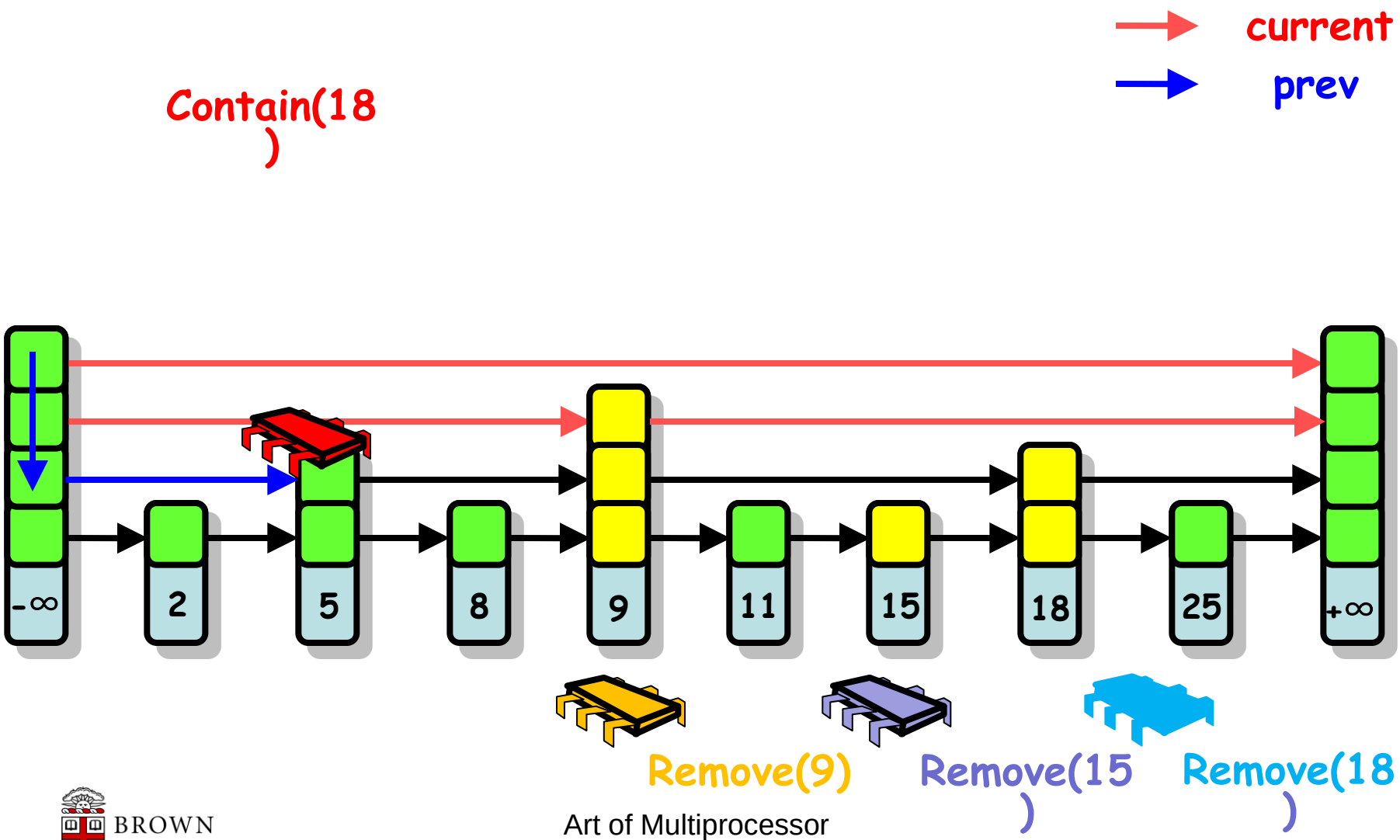
Remove(15)

Remove(18)

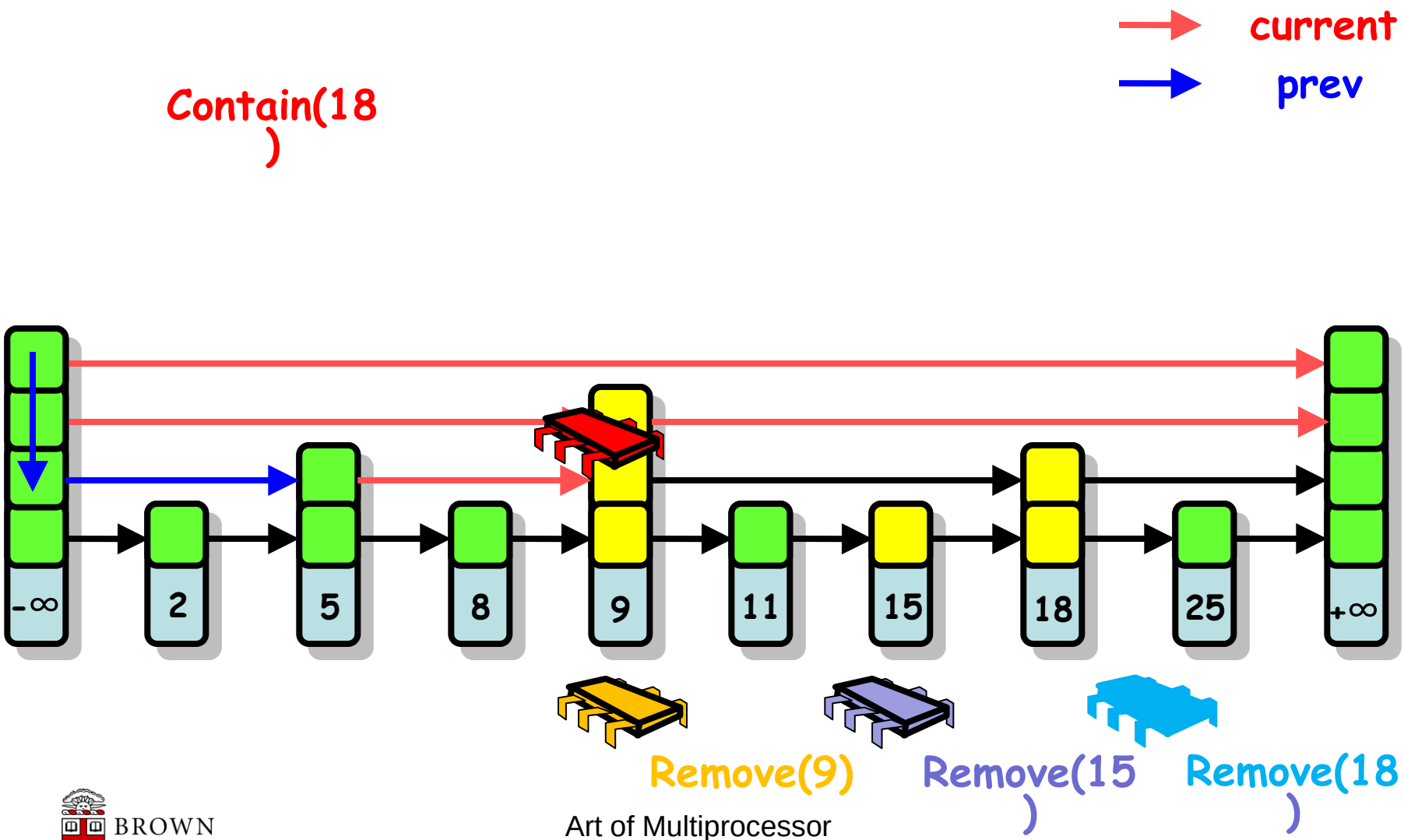
contains(18)


current
prev
Contain(18
)
-∞
2
5
8
9
11
15
18
25
+∞
5 < 18
Remove(9)
Remove(15
)
Remove(18
)

contains(18)


current
prev
Contain(18
)
-∞
2
5
8
9
11
15
18
25
+∞
Remove(9)
Remove(15
)
Remove(18
)

contains(18)

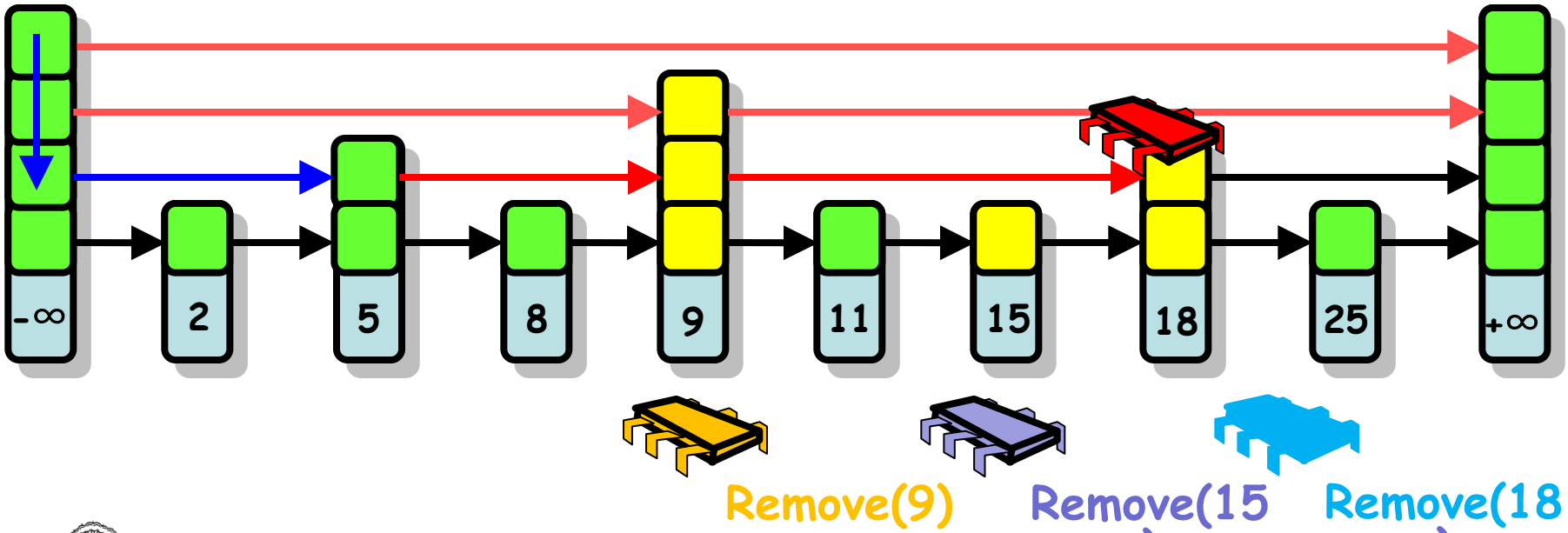

current
prev
Contain(18
)
-∞
2
5
8
9
11
15
18
25
+∞
Remove(9)
Remove(15
)
Remove(18
)

contains(18)

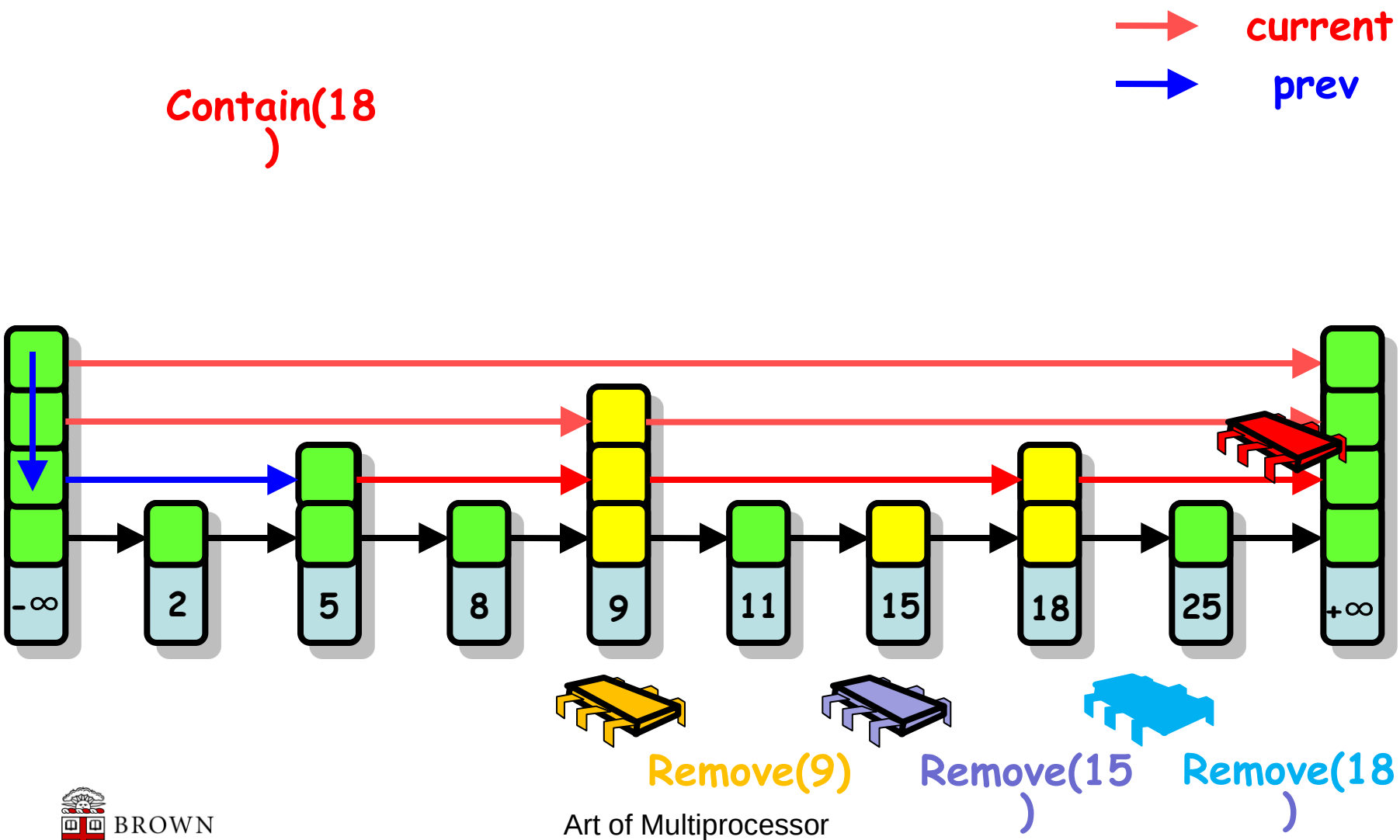
current

prev

Contain(18
)



contains(18)

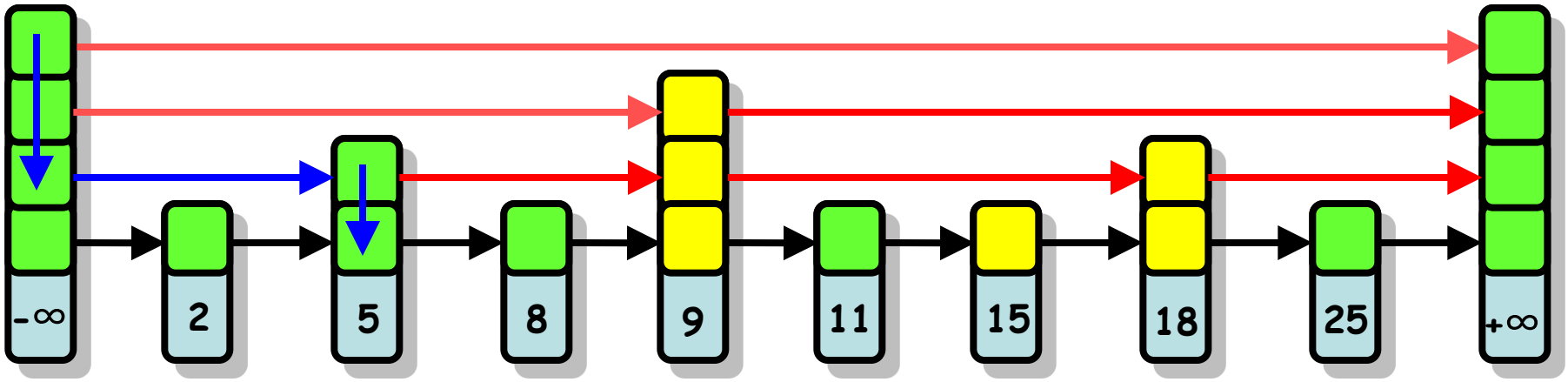

current
prev
Contain(18
)
-∞
2
5
8
9
11
15
18
25
+∞
Remove(9)
Remove(15
)
Remove(18
)

contains(18)

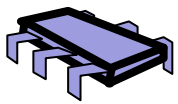
Contain(18)

current

prev



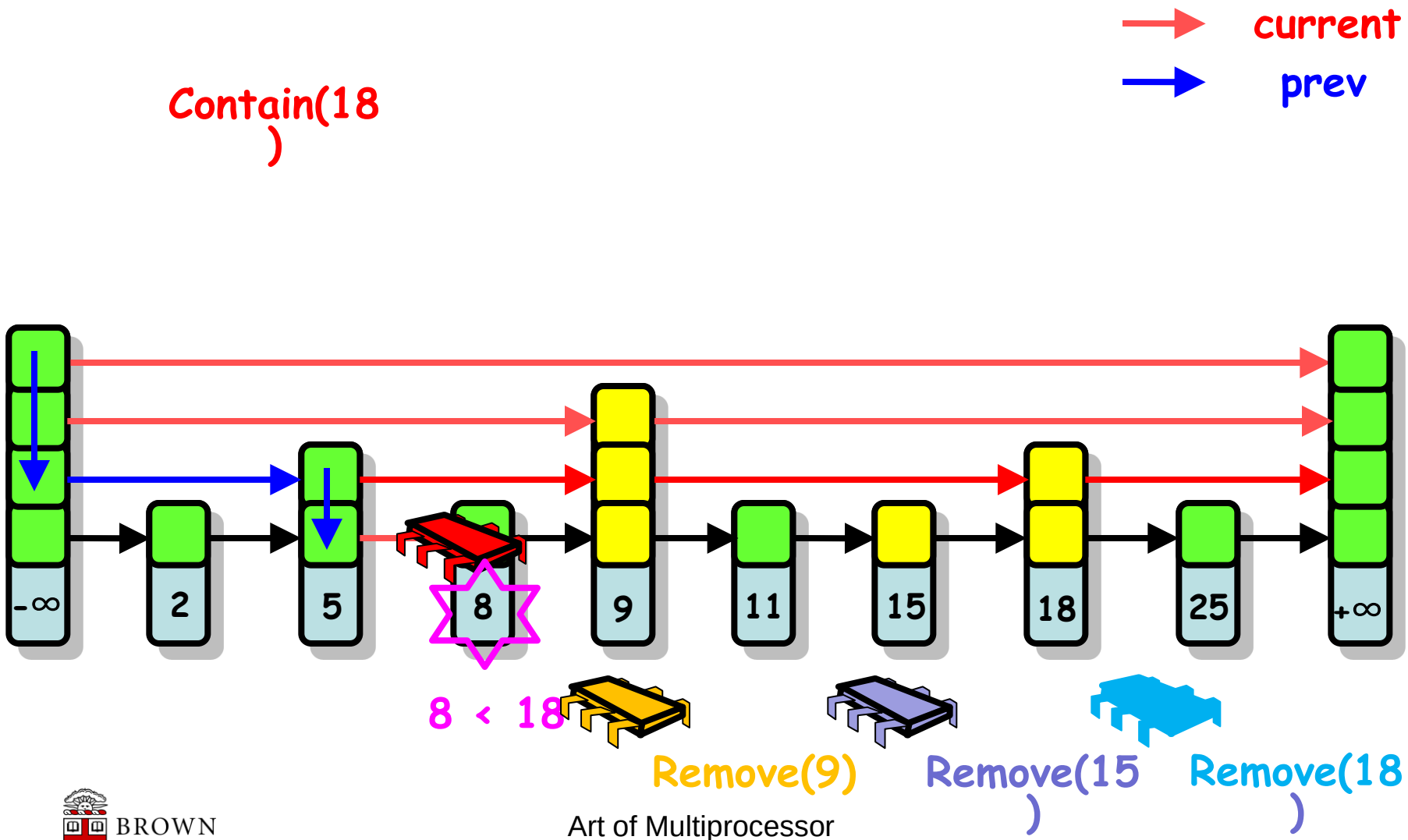
Remove(9)



Remove(15)

Remove(18)

contains(18)

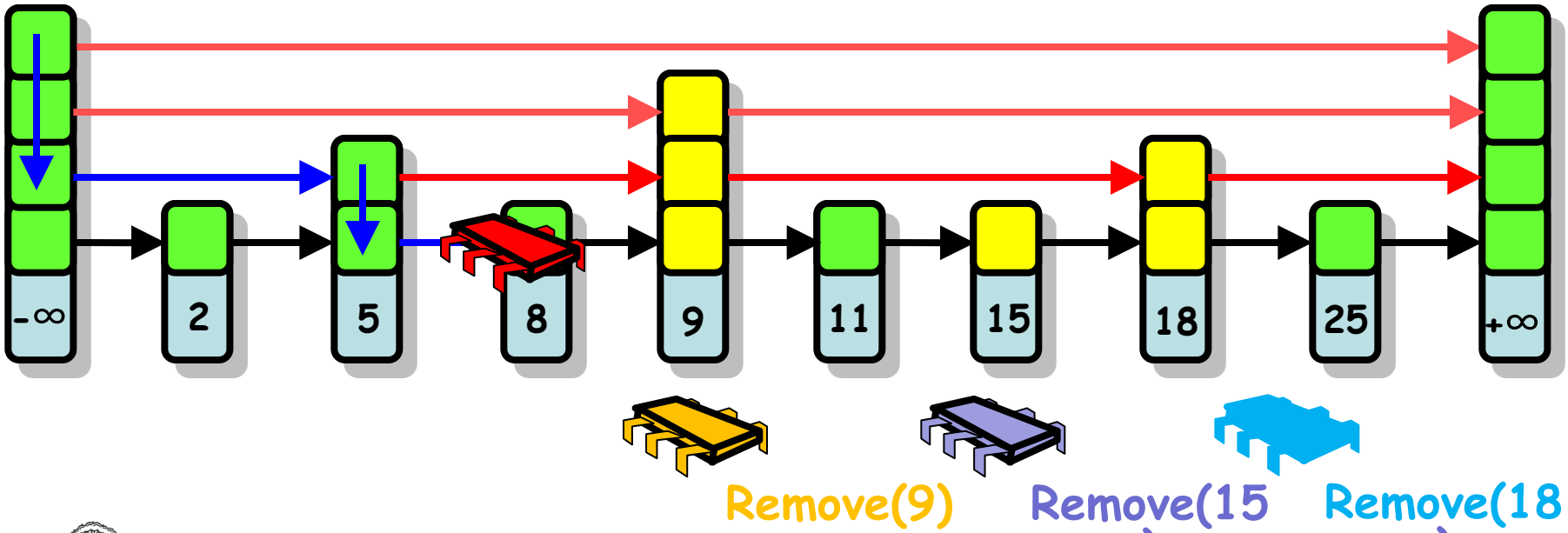

current
prev
Contain(18
)
-∞
2
5
8
9
11
15
18
25
+∞
8 < 18
Remove(9)
Remove(15
)
Remove(18
)

contains(18)

current

prev

Contain(18
)

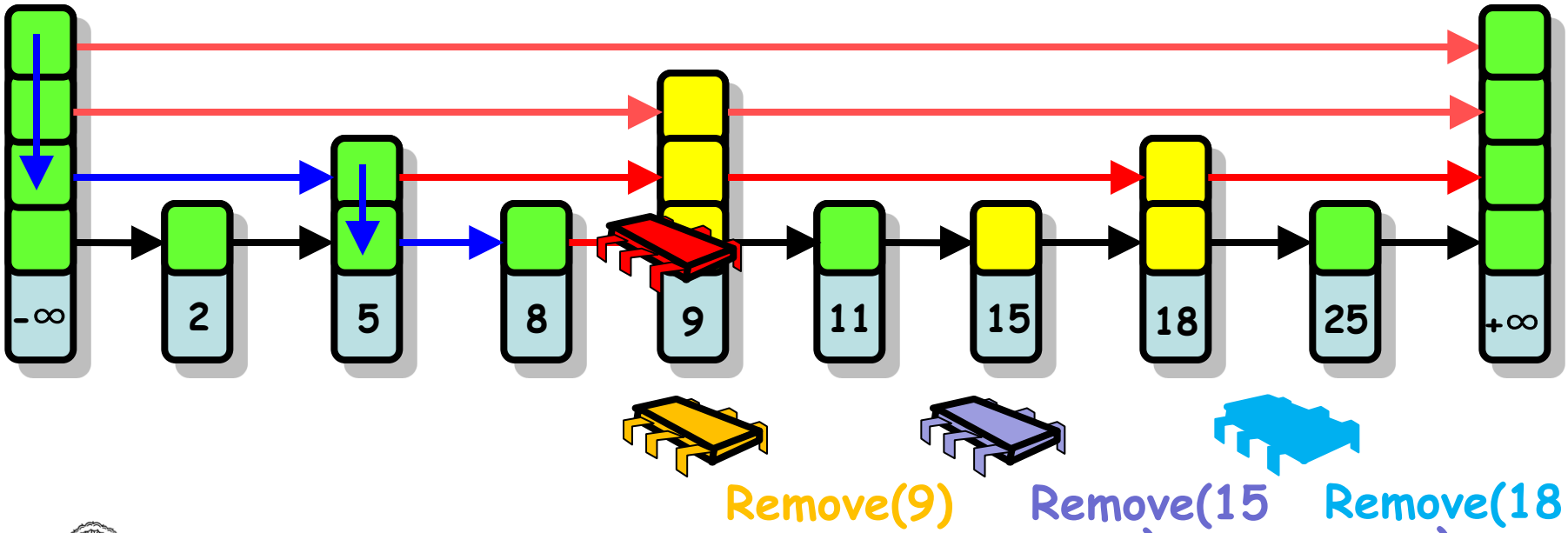


contains(18)

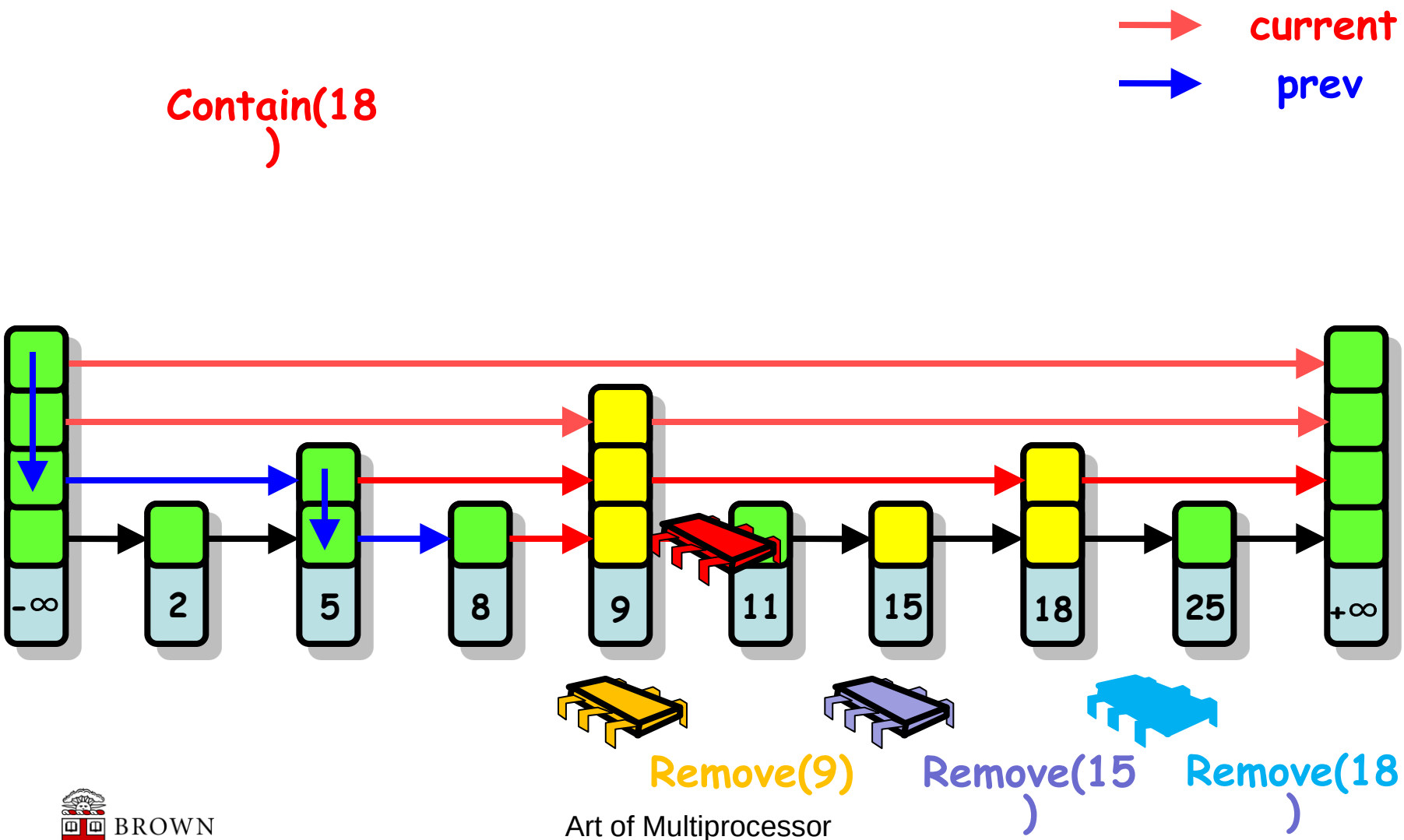
current

prev

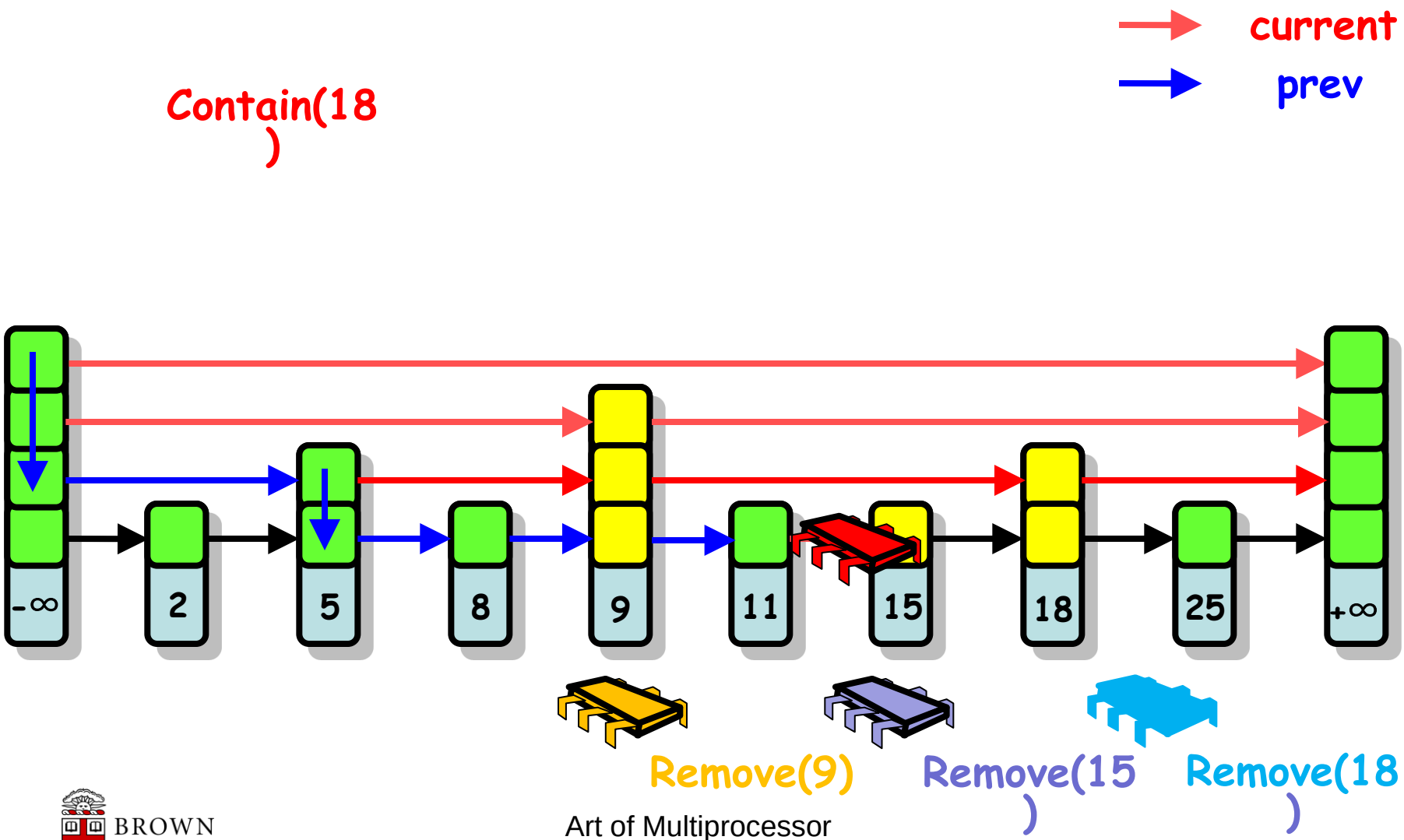
Contain(18
)



contains(18)


current
prev
Contain(18
)
-∞
2
5
8
9
11
15
18
25
+∞
Remove(9)
Remove(15
)
Remove(18
)

contains(18)

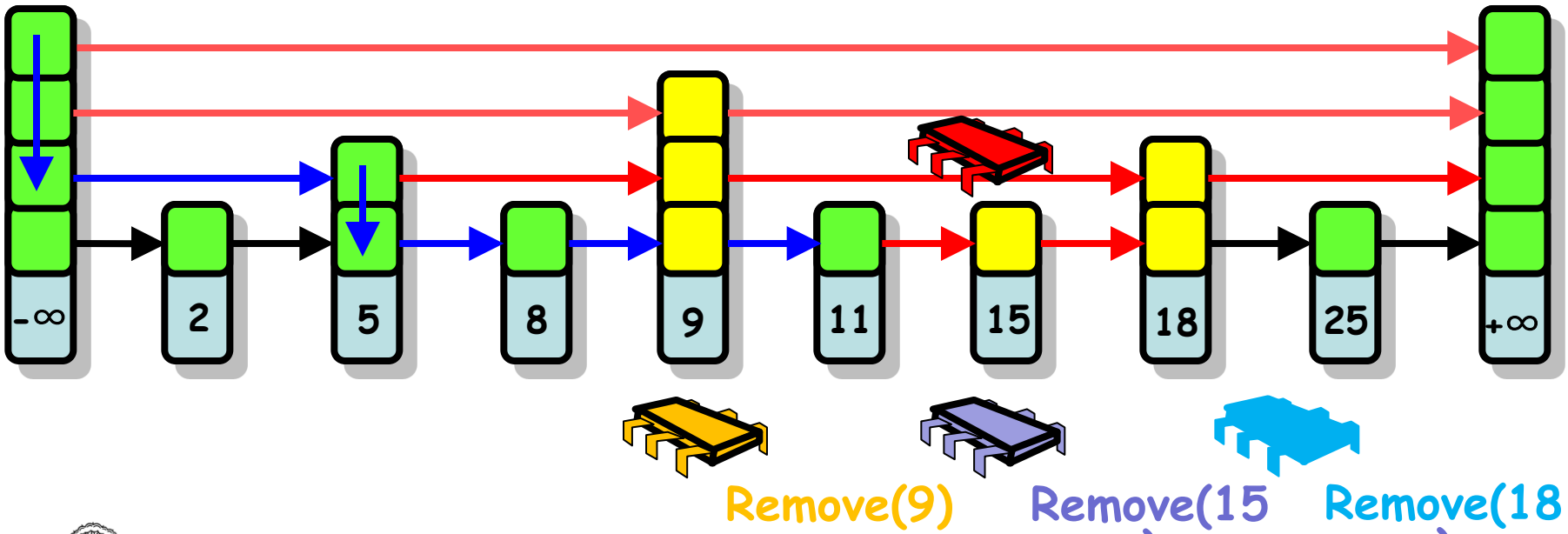

current
prev
Contain(18
)
-∞
2
5
8
9
11
15
18
25
+∞
Remove(9)
Remove(15
)
Remove(18
)

contains(18)

Contain(18)

current

prev



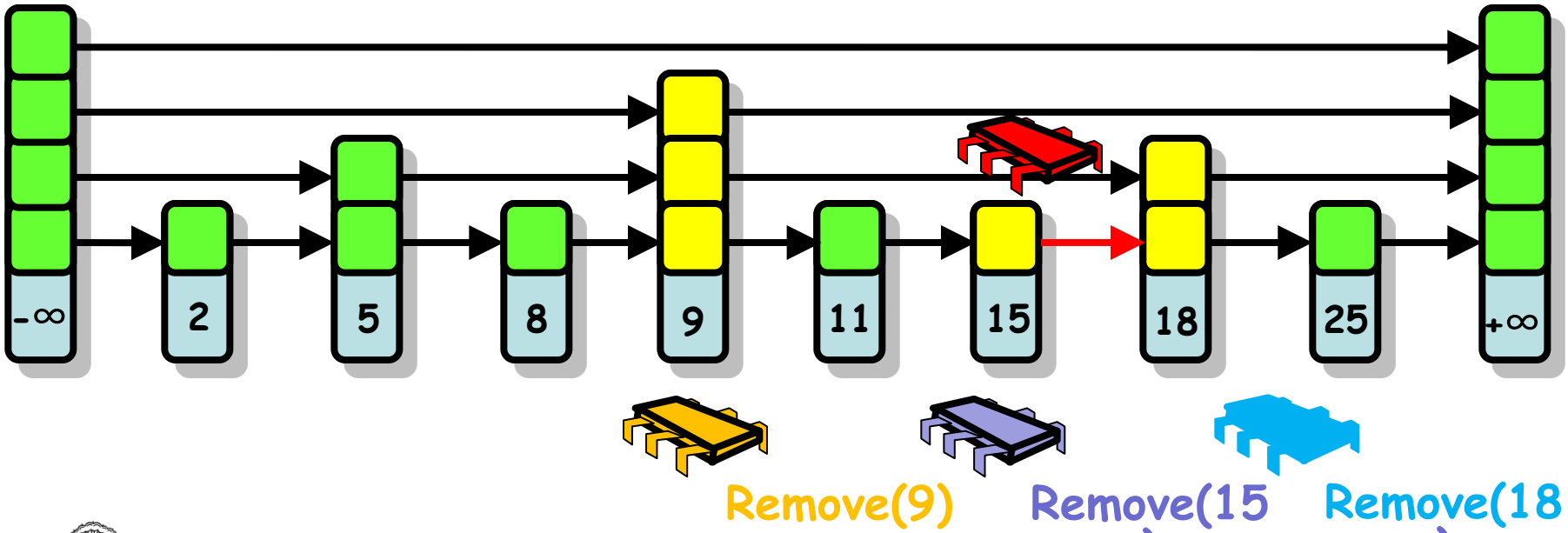
contains(18)

Contain(18)

Add(18)

current

prev



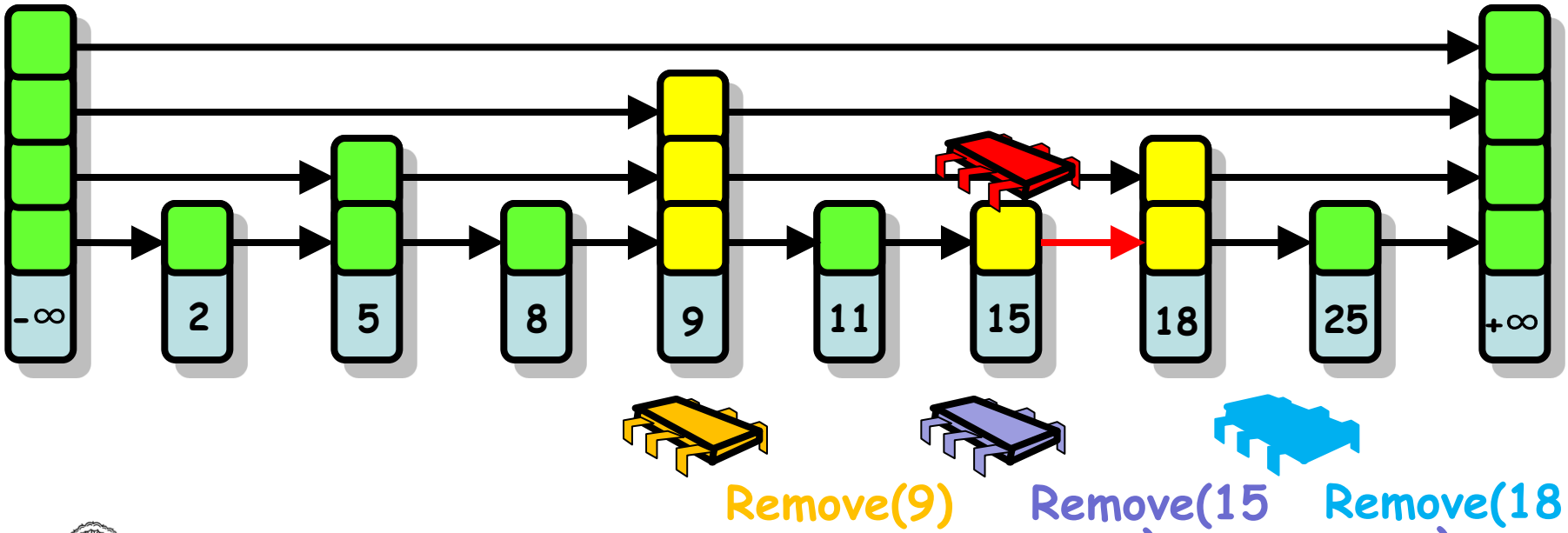
contains(18)

Contain(18
)

Add(18) -> find(18)

current

prev



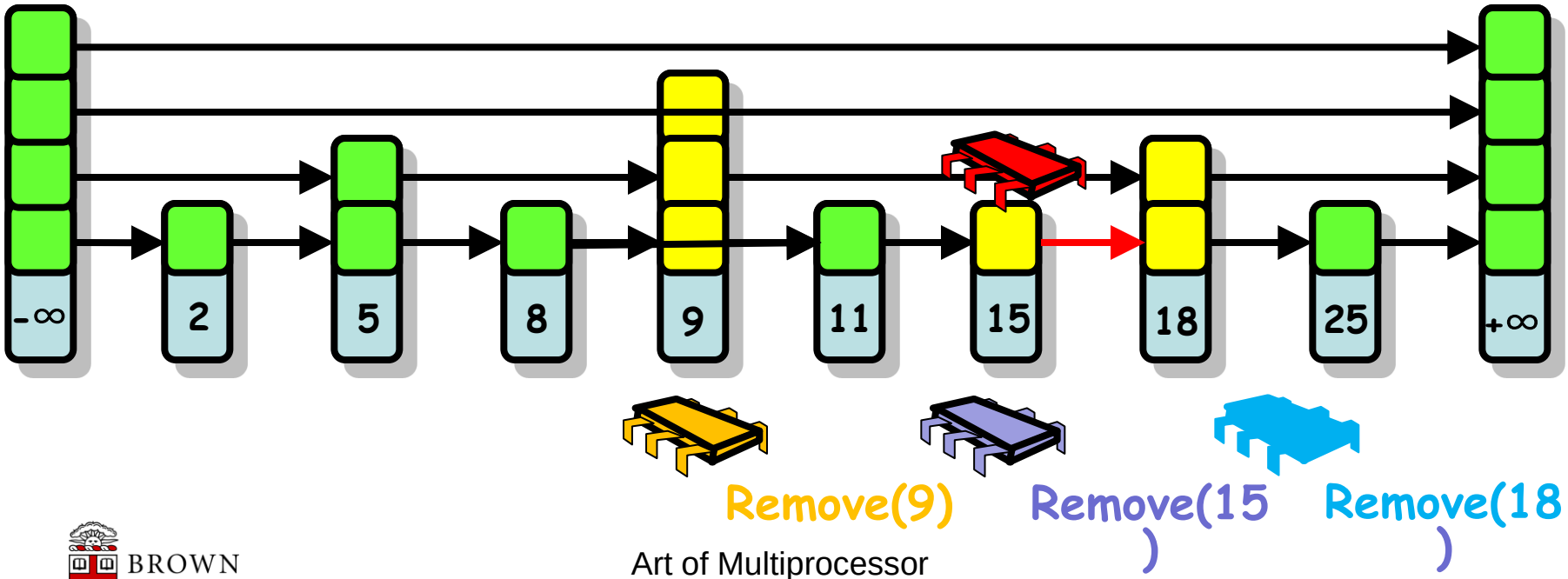
contains(18)

Contain(18
)

Add(18) -> find(18)

current

prev



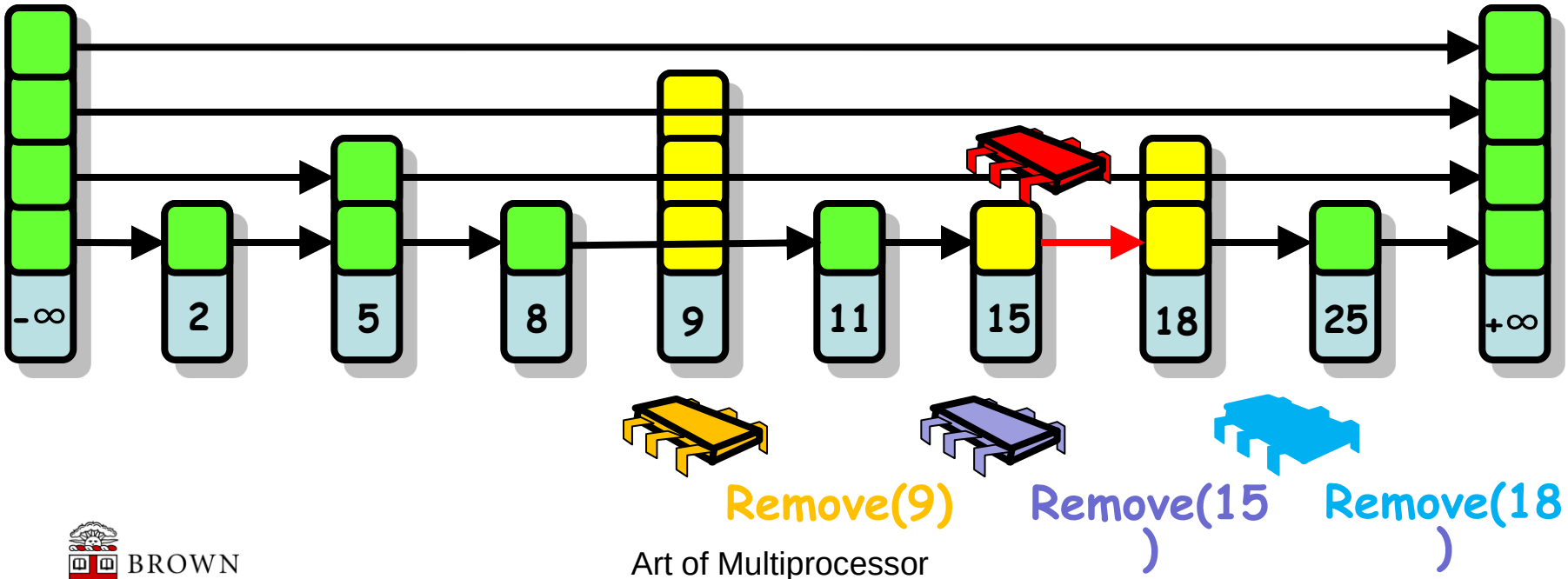
contains(18)

Contain(18
)

Add(18) -> find(18)

current

prev



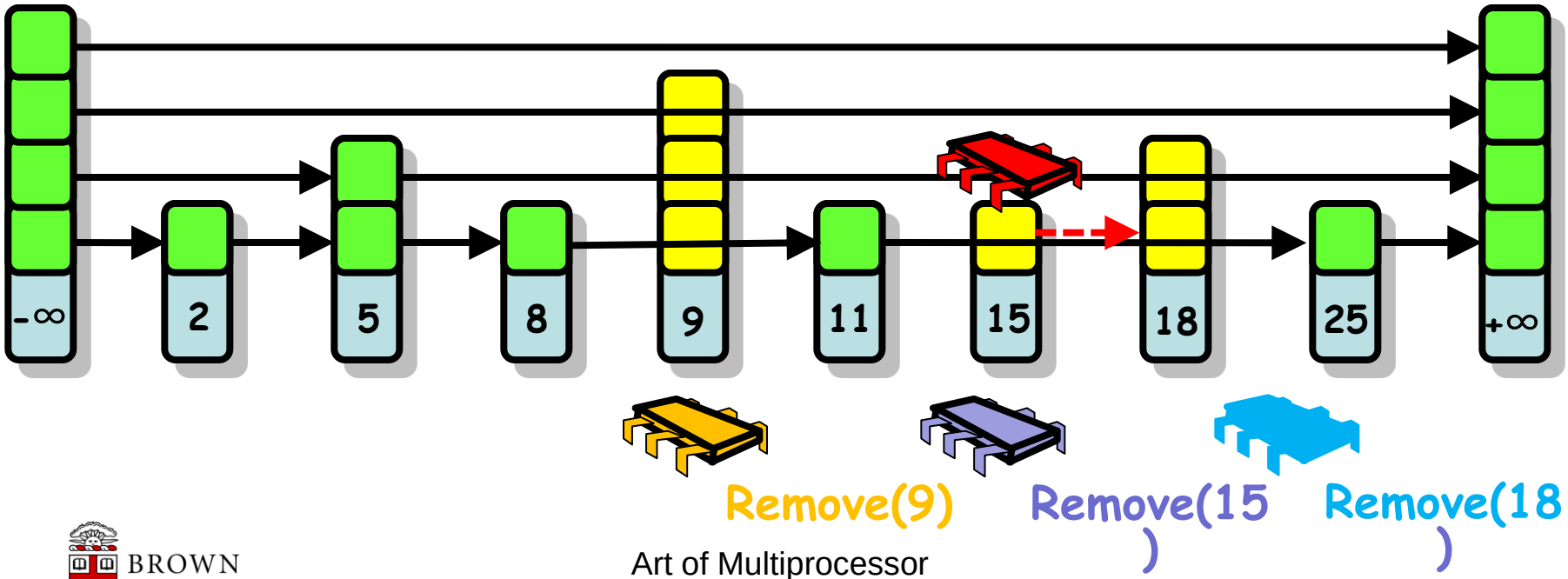
contains(18)

Contain(18
)

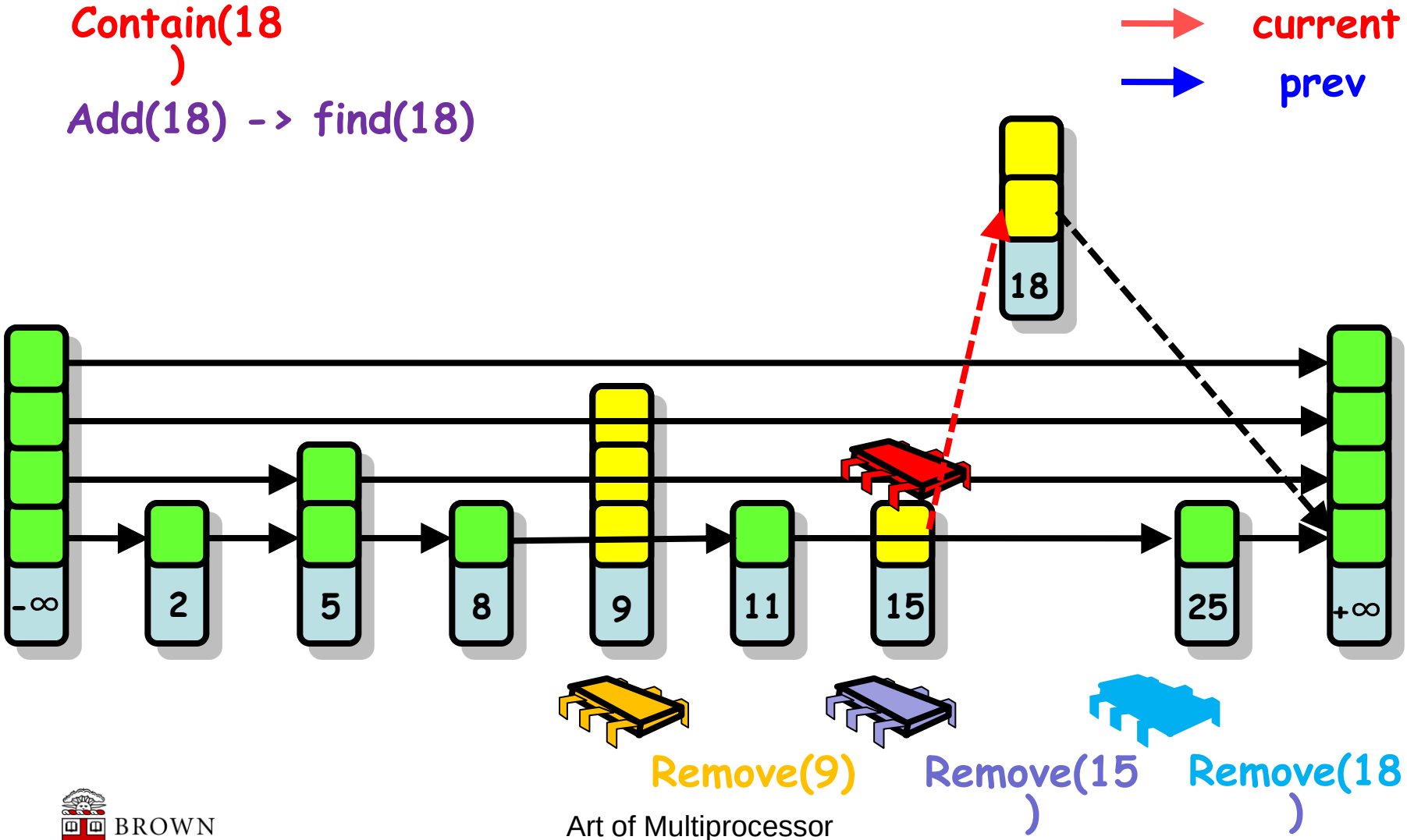
Add(18) -> find(18)

current

prev



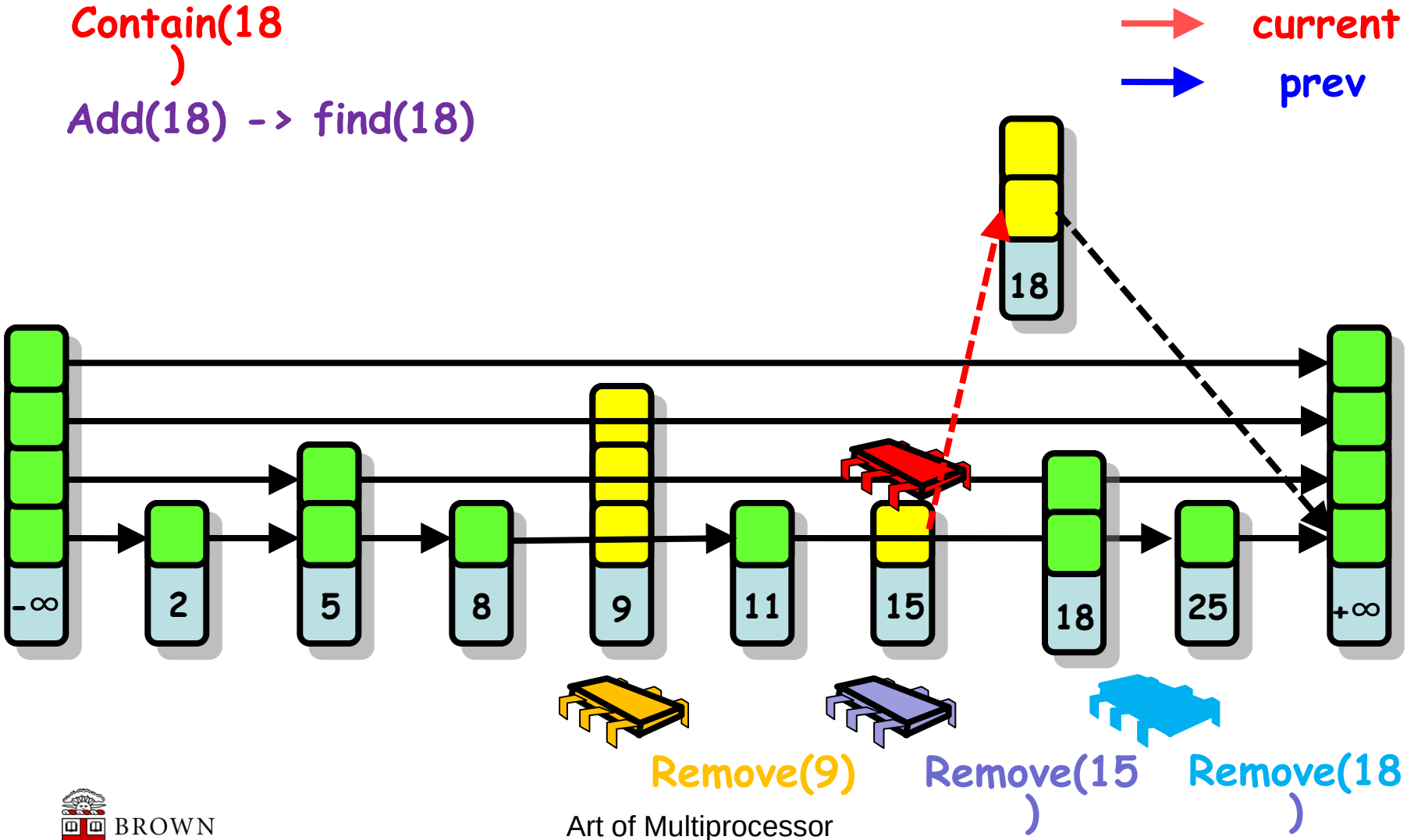
contains(18)


Contain(18
)
Add(18) -> find(18)
current
prev
18
-∞
2
5
8
9
11
15
25
+∞
Remove(9)
Remove(15
)
Remove(18
)

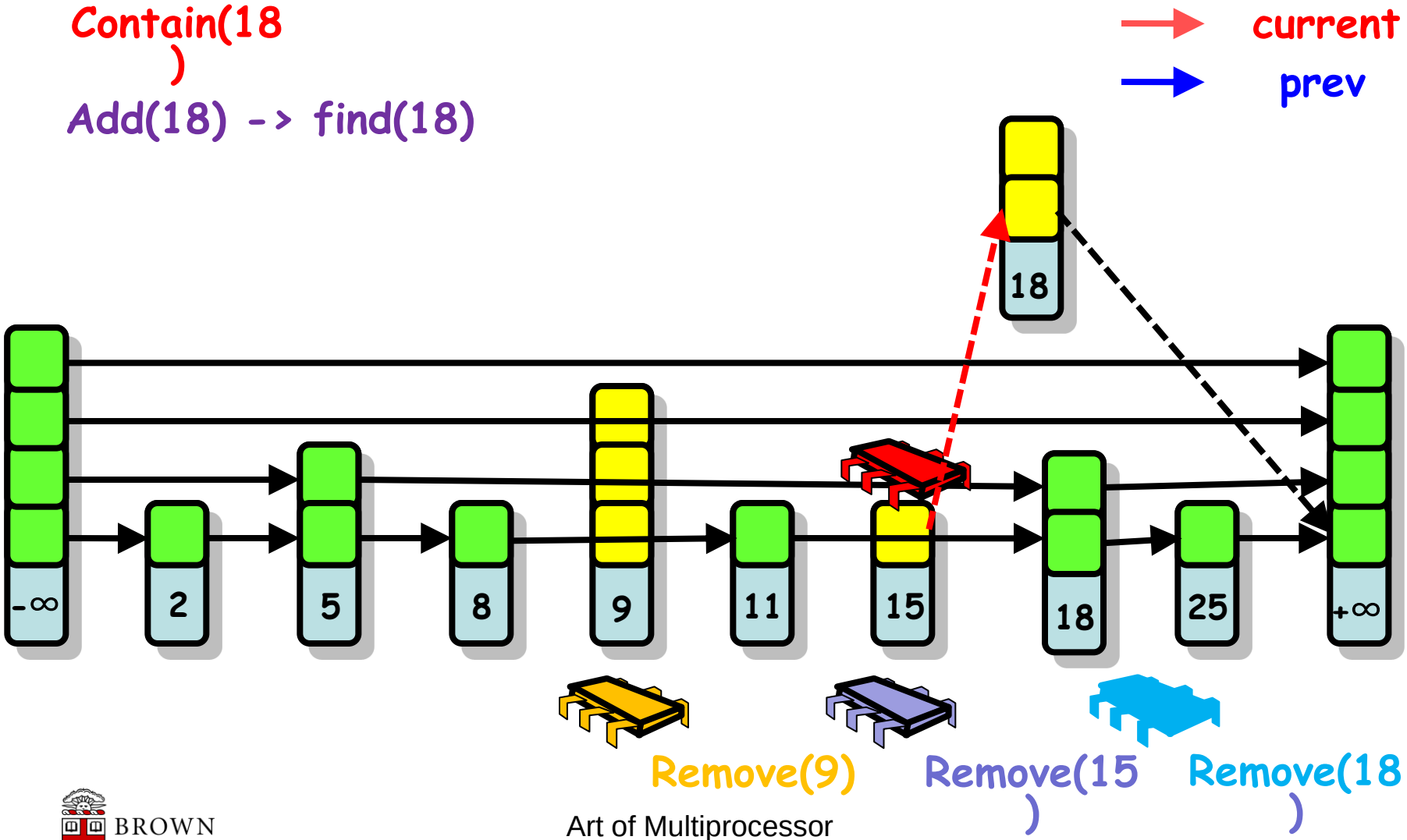
contains(18)

Contain(18
)

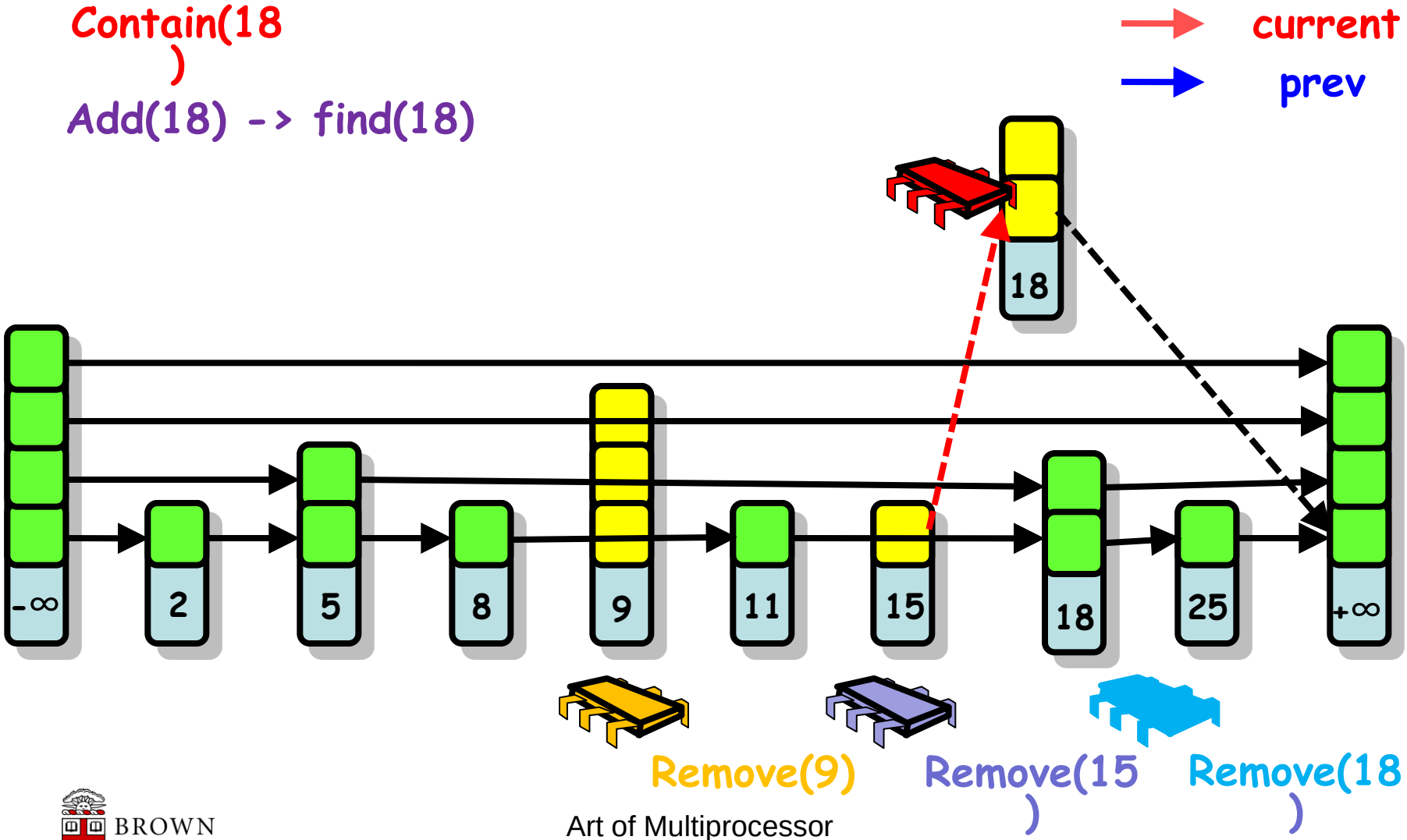
Add(18) -> find(18)



contains(18)


Contain(18
)
Add(18) -> find(18)
current
prev
-∞
2
5
8
9
11
15
18
18
25
+∞
Remove(9)
Remove(15
)
Remove(18
)

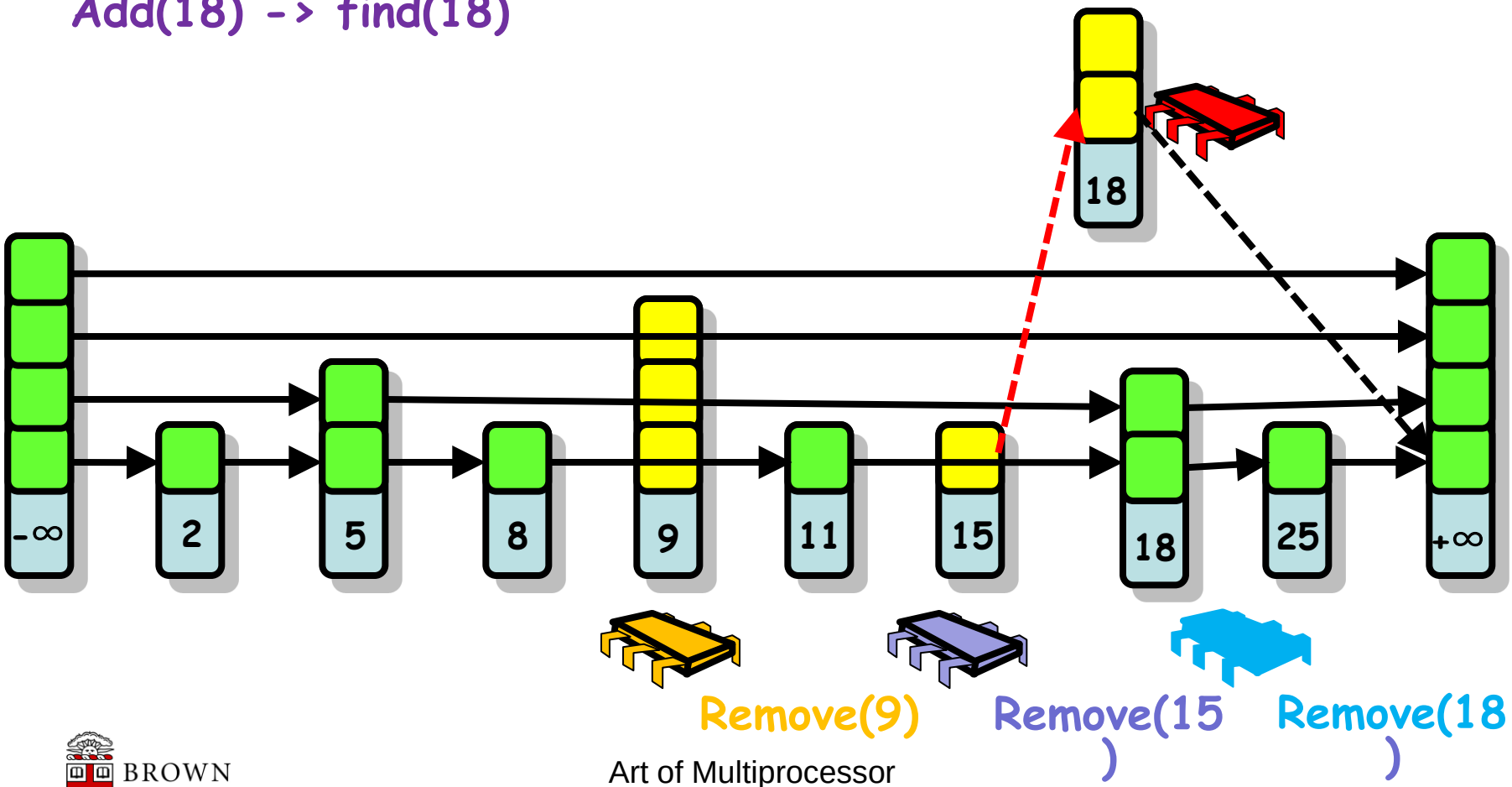
contains(18)


Contain(18
)
Add(18) -> find(18)
current
prev
18
-∞
2
5
8
9
11
15
18
25
+∞
Remove(9)
Remove(15
)
Remove(18
)

contains(18)

Contain(18) returns FALSE !

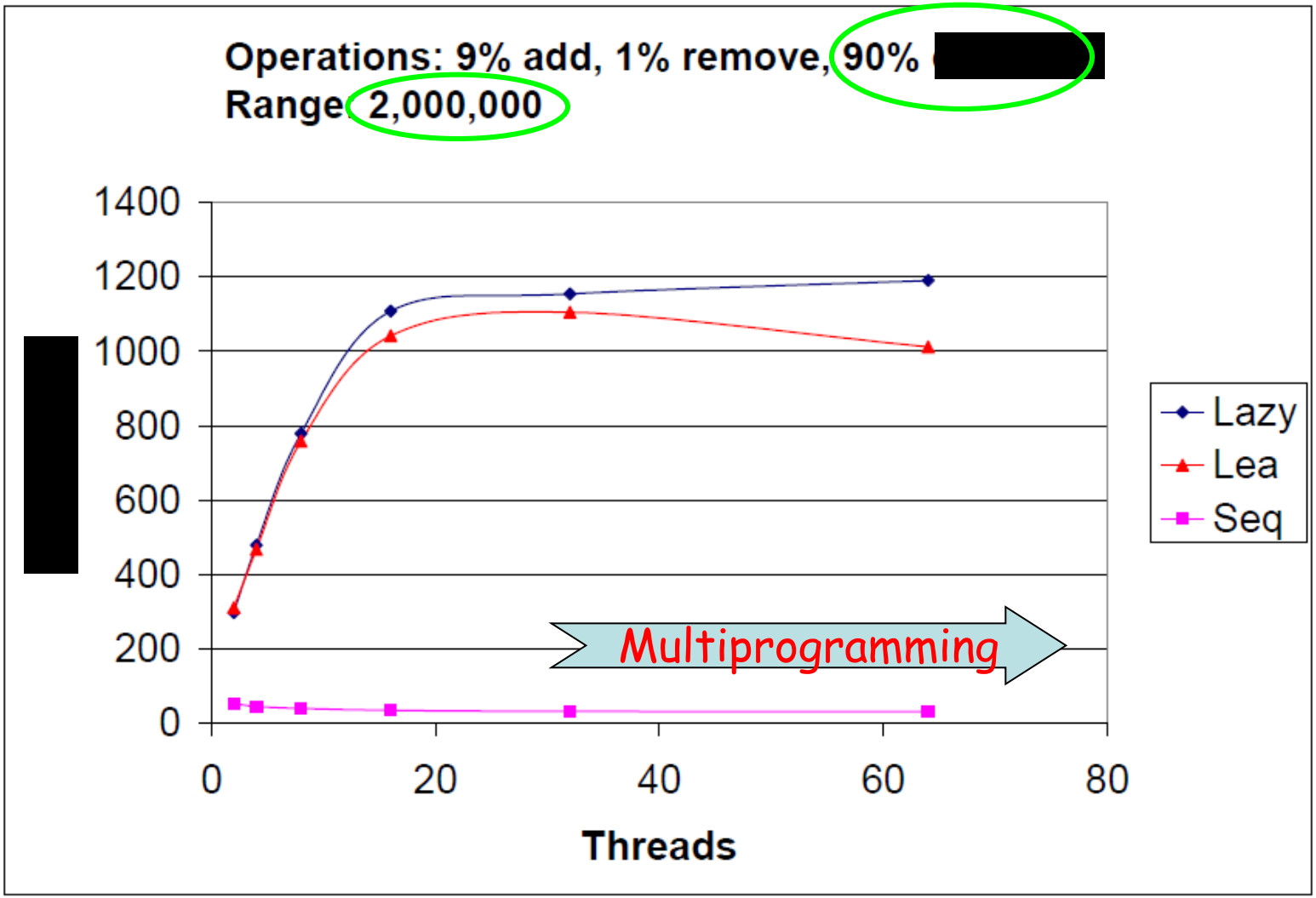
Add(18) -> find(18)



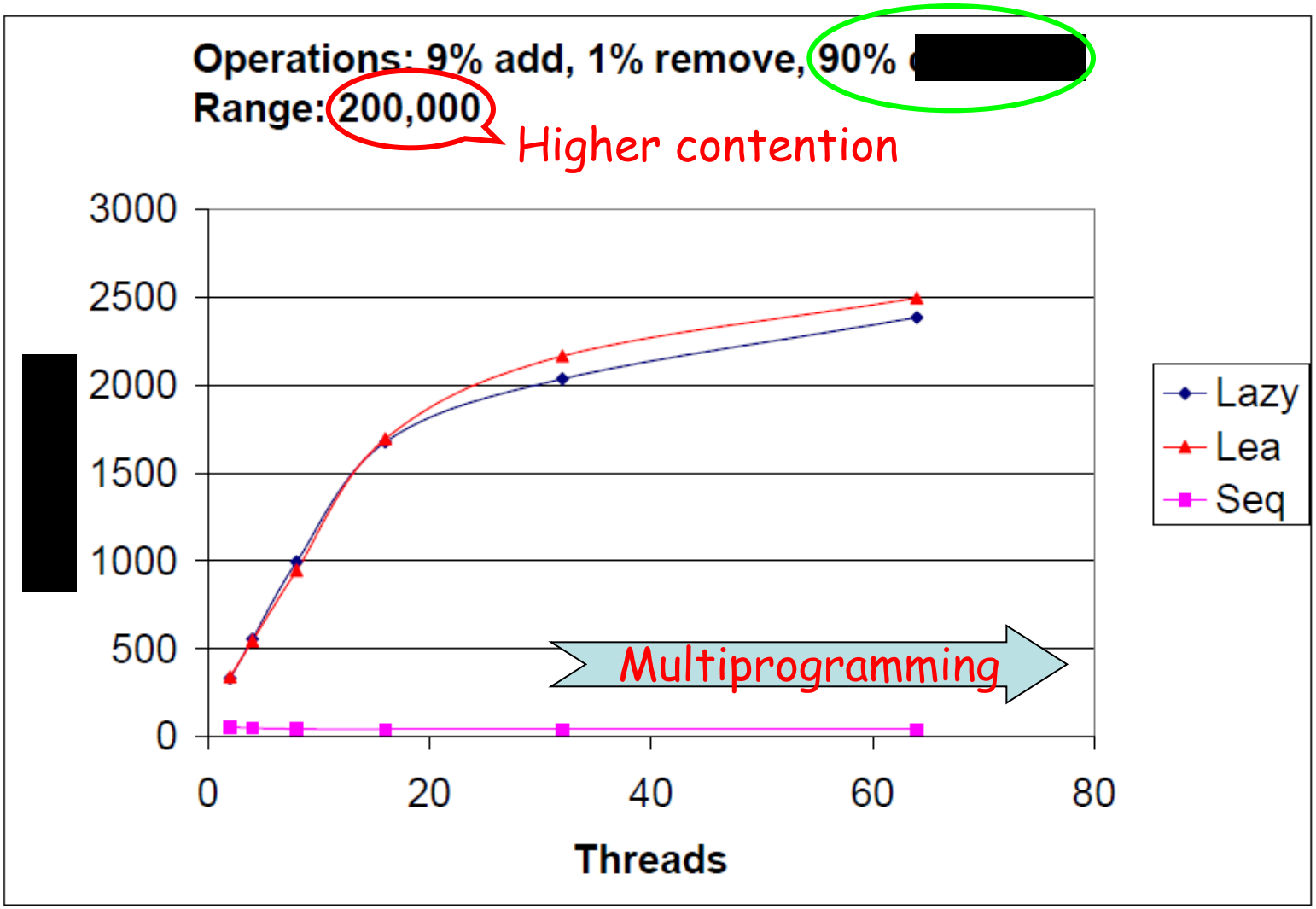
A Simple Experiment

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

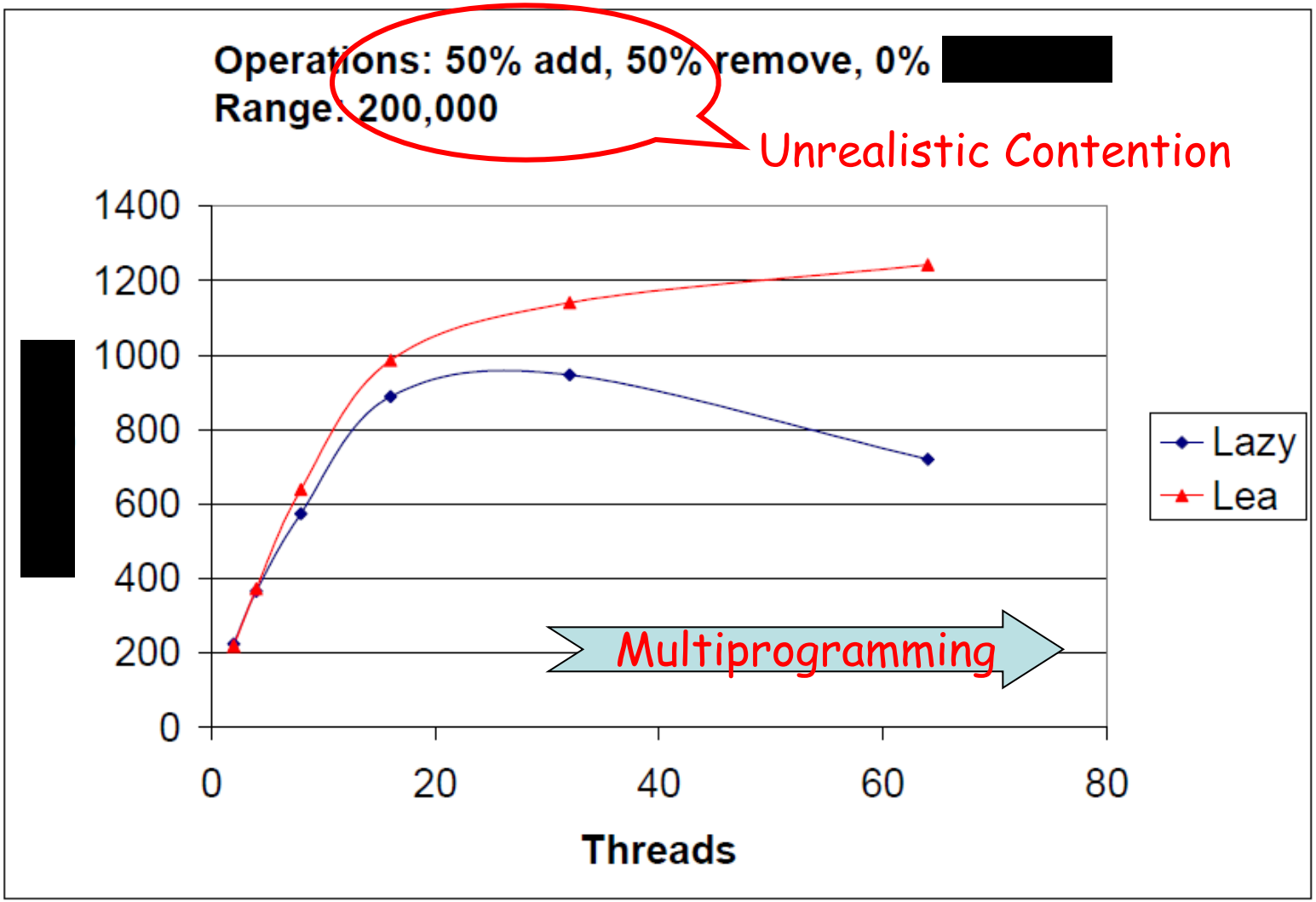
Lazy Skip List: Performance



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