MULTICORE PROGRAMMING

Linked Data Structures

Lecture 9

Trevor Brown

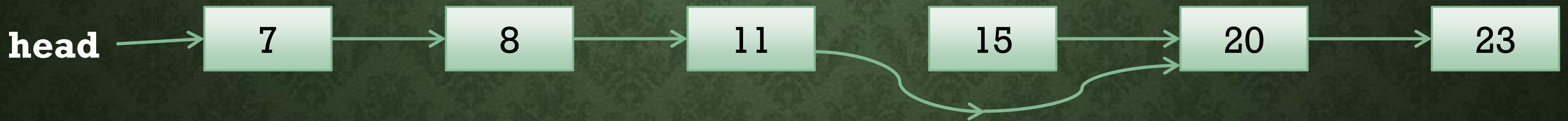
RECALL: LOCK-BASED SINGLY-LINKED LISTS

- Ordered set implemented with singly-linked list
- Hand-over-hand locking discipline:
  - must lock a node before accessing it
  - Can only acquire a lock on a node:
  - **if** it is the list **head**, or
    if you already hold a lock on the previous node

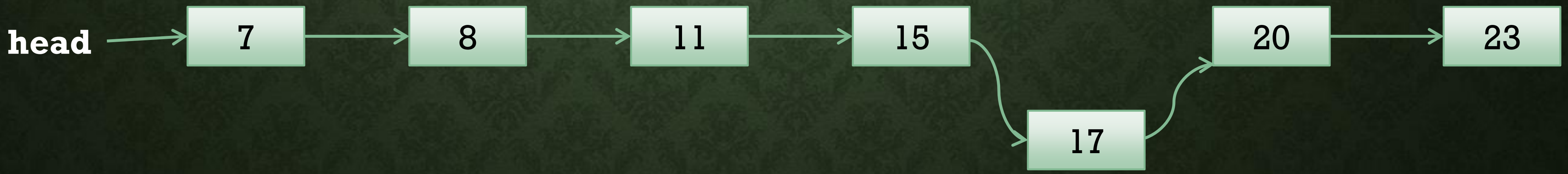
Locking causes **many** cache invalidations, even for searches!

Should **avoid locking** while searching/traversing the list!

- Delete(15)



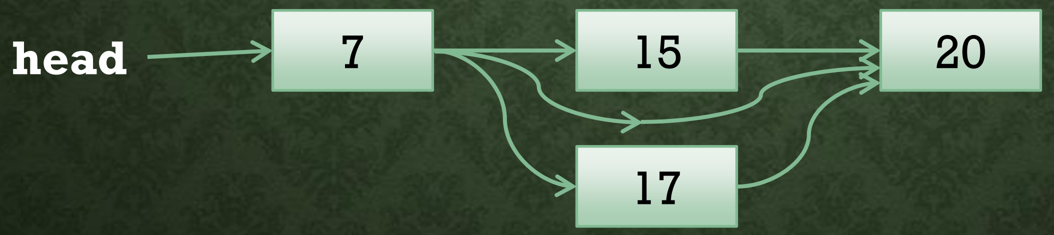
- Insert(17)



LOCK-FREE SINGLY-LINKED LISTS: ATTEMPTING TO USE CAS

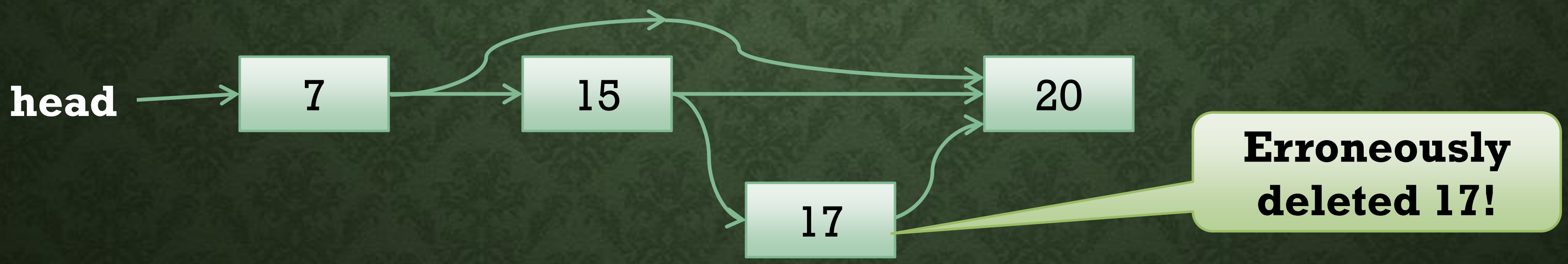
- Ordered set implemented with singly-linked list
- Delete(15)
  - Traverse list, then CAS 7 **.next** from 15 to 20
- Insert(17)
  - Traverse list, create node 17, then CAS 7 **.next** from 20 to 17

One approach is to design a completely lock-free list…



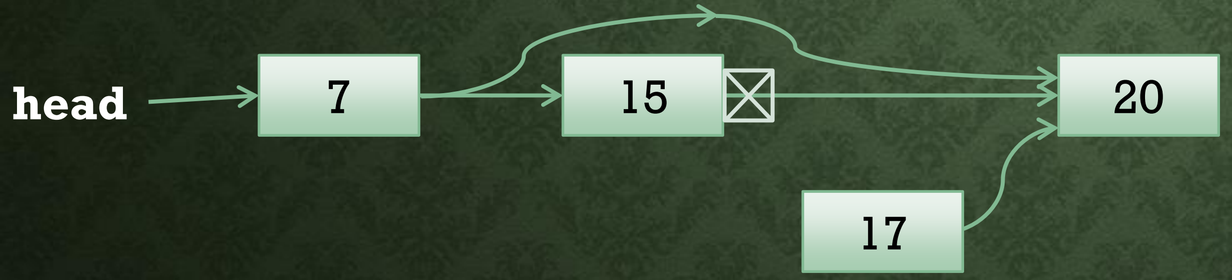
THE PROBLEM

- What if the operations are concurrent?
  - Delete(15): **pause** just before CAS 7 **.next** from 15 to 20
  - Insert(17): traverse list, create node 17, then CAS 15 **.next** from 20 to 17
  - Delete(15): **resume** and CAS 7 **.next** from 15 to 20



SOLUTION: MARKING [HARRIS2001]

- Idea: prevent changes to nodes that will be deleted
- Before deleting a node, *mark* its **next** pointer
  - How does this fix the Insert(17), Delete(15) example?
  - Delete(15) marks 15 *before* using **CAS** to delete it
  - Insert(17) cannot modify 15**.next** because it is marked



Whenever a thread encounters a **mark**, it tries to **help** the deletion

By doing a CAS to **unlink** the marked node…

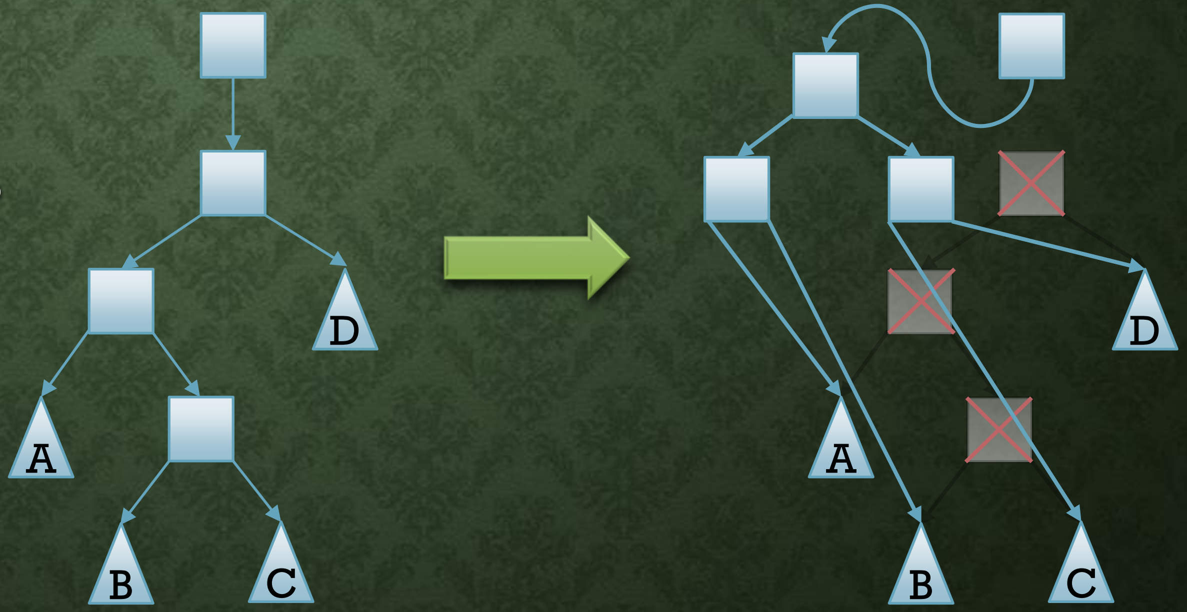
Even if the thread doing the deletion crashed, we still guarantee progress…

Okay. We can do lists!

Note: you can also do fast <u>**lock-based**</u> lists that avoid locking while searching…

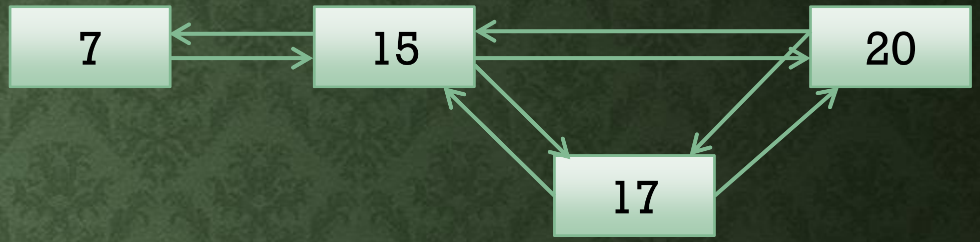
WHAT ABOUT REMOVING SEVERAL NODES?

- Ex1: atomically deleting consecutive nodes in a list…
  - Delete(15 AND 20)
    - Mark 15, **then** mark 20?
    - What can go wrong…
    - **Crash** between these steps?
    - Some change that makes it **incorrect** to mark 20?
- Ex2: performing tree **rotations** by replacing nodes…

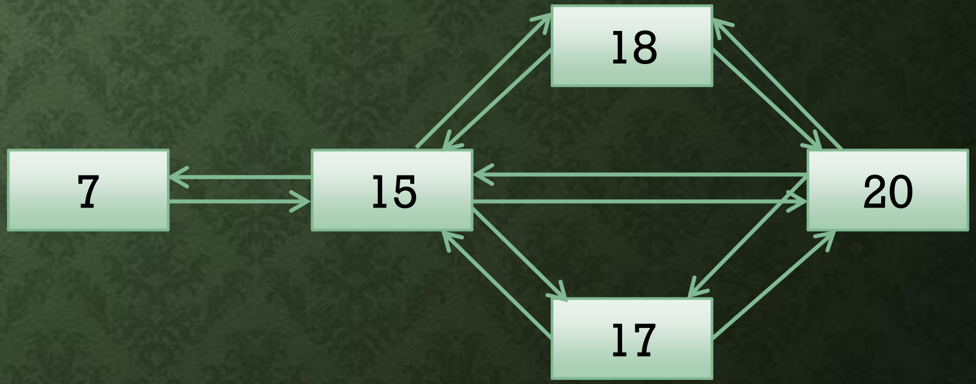


OR CHANGING TWO POINTERS AT ONCE?

- Doubly-linked list
- Insert(17)

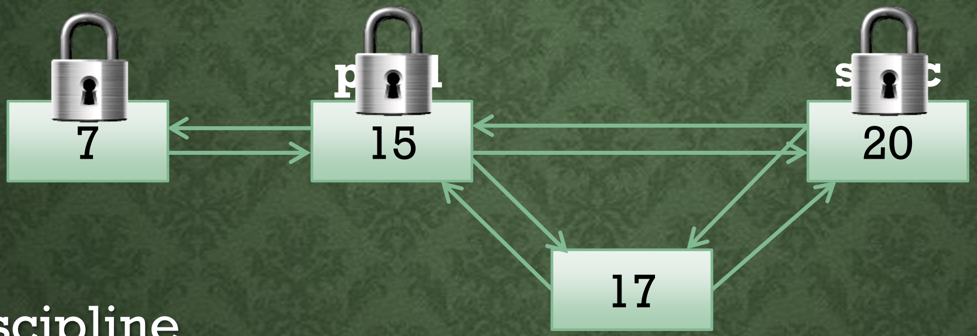


- If the two pointer changes are **not** atomic… (i.e., if they are done: left **then** right)
  - Insertions and deletions could happen between them!
  - Example:
    - Insert(17) does 15.next := 17
    - But before 20.pred := 17, a thread does Insert(18)
    - Then Insert(17) finishes
  - Result:
    - List structure is corrupted…
    - 18 is visible when searching left-to-right but not vice versa
    - 17 is visible when searching right-to-left but not vice versa



EASY LOCK-BASED DOUBLY-LINKED LIST

- Doubly-linked list
- Insert(17)



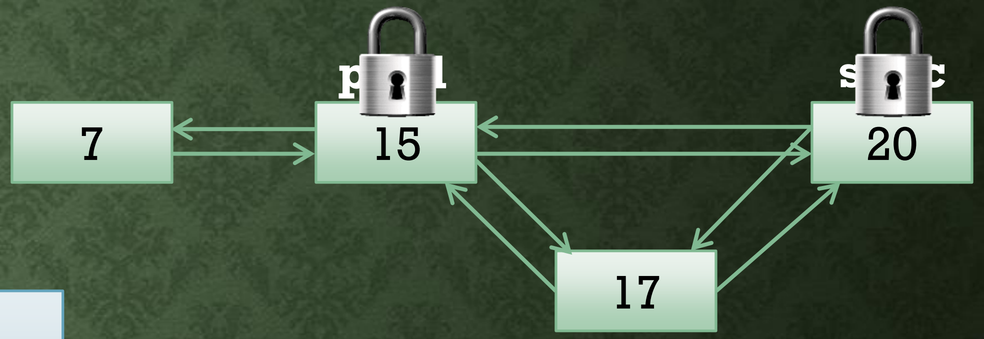
- Simple locking discipline
  - Hand-over-hand locking
  - Never access anything without locking it first
- Correct, but at what cost?
  - To respect the locking discipline, we have to **lock while searching**!
  - **Can we avoid locking during search?**

Deadlock possible if we search from both sides… for now, imagine we search only left-to-right…

CAN WE SEARCH A DOUBLY-LINKED LIST **WITHOUT** LOCKING NODES?

- Insert(k):
  - Search without locking until we reach nodes **pred & succ** where pred.key < k <= succ.key
  - If we found k, return false
  - Lock pred, lock succ
    - If pred.next != succ, unlock all & retry
    - Create new node n (containing k, pointing to pred & succ)
    - pred.next = n
    - succ.prev = n
  - Unlock all

Insert(17)



At what point does Insert affect the return value of Contains?

Where should we linearize insert?

Where should we linearize contains?

No single line of code works…

Let's see why this is true…

- Contains(k):
  - curr = head
  - Loop
    - If curr == NULL or curr.key > k then return false
    - If curr.key == k then return true
    - curr = curr.next

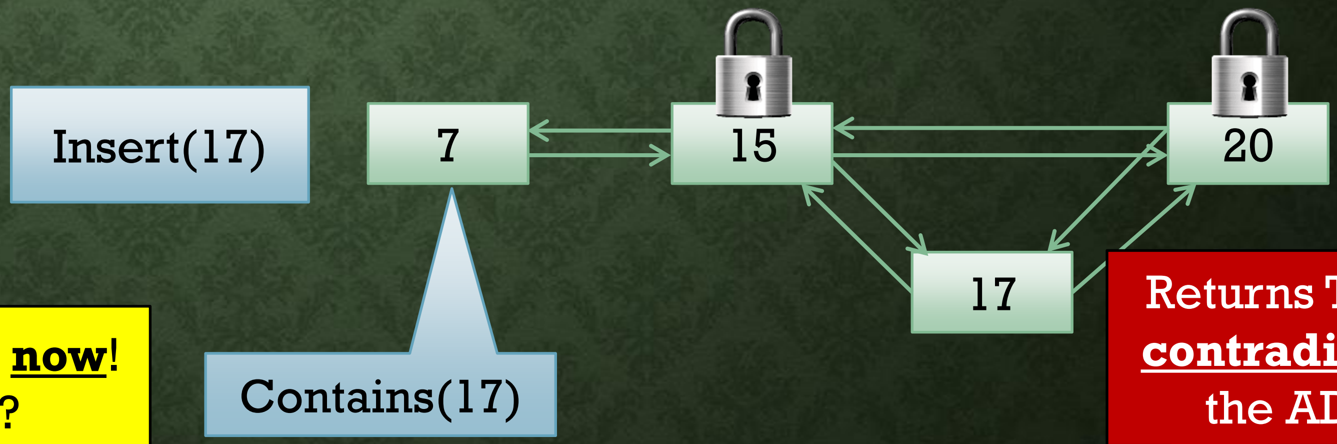
IT'S HARD TO LINEARIZE CONTAINS… **EXAMPLE 1**

- Insert(k):
  - Search without locking until we reach nodes **pred & succ** where pred.key < k <= succ.key
  - If we found k, return false
  - Lock pred, lock succ
    - If pred.next != succ, unlock all & retry
    - Create new node n
    - **pred.next = n**
    - succ.prev = n
  - Unlock all

- Contains(k):
  - curr = head
  - Loop
    - If curr == NULL or curr.key > k then return false
    - If curr.key == k then return true
    - curr = curr.next

suppose LP is here

Consider a **concurrent** Contains(17) and Insert(17) in this list



Returns TRUE, **contradicting** the ADT!

Contains(17) is linearized **now**! What should it return?

False!

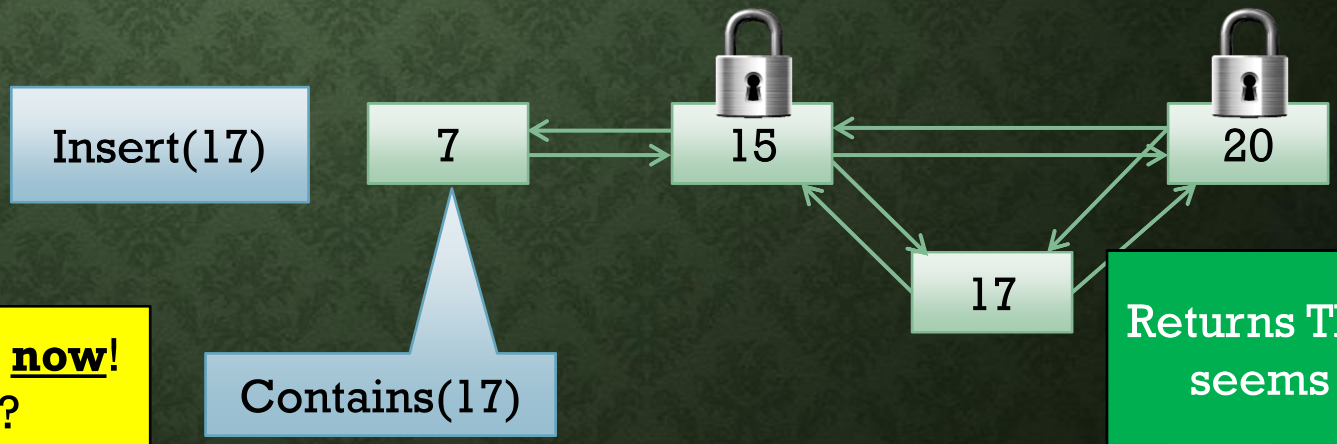
IT'S HARD TO LINEARIZE CONTAINS… **EXAMPLE 2**

- Insert(k):
  - Search without locking until we reach nodes **pred & succ** where pred.key < k <= succ.key
  - If we found k, return false
  - Lock pred, lock succ
    - If pred.next != succ, unlock all & retry
    - Create new node n
    - **pred.next = n**
    - succ.prev = n
  - Unlock all

- Contains(k):
  - curr = head
  - Loop
    - If curr == NULL or curr.key > k then return false
    - If curr.key == k then return true
    - curr = curr.next

Suppose LP is **last** execution of this line

Consider a **concurrent** Contains(17) and Insert(17) in this list



Returns TRUE… seems OK

Contains(17) is linearized **now**! What should it return?

True!

WHAT IF WE ALLOW KEY **DELETION** ALSO?

- Insert(k):
  - Search without locking until we reach nodes **pred & succ** where pred.key < k <= succ.key
  - If we found k, return false
  - Lock pred, lock succ
    - If pred.next != succ, unlock all & retry
    - Create new node n
    - **pred.next = n**
    - succ.prev = n
  - Unlock all

- Contains(k):
  - curr = head
  - Loop
    - If curr == NULL or curr.key > k then return false
    - If curr.key == k then return true
    - curr = curr.next

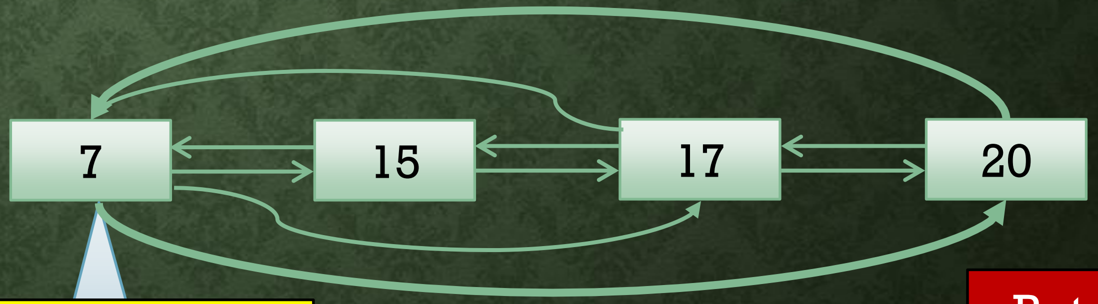
Suppose LP is **last** execution of this line

Consider a concurrent Contains(17), Delete(15) and Delete(17) in this list

Insert(17)

Delete(15)

Delete(17)



Contains(17) is linearized **now**! What should it return?

False!

Returns True! **Contradicts** the ADT!!

Idea: prove a suitable LP **exists** for every operation in every execution

Is there a time in this execution where we **could** linearize a return value of true?

INTUITION BEHIND LINEARIZATION ARGUMENT

- If Insert/Delete **changes** the data structure (returns true), LP is the write to pred.next
  - This is when Contains becomes aware of the change…
- Otherwise, Insert/Delete returns false (and we prove there exists some correct LP)
- Case 1: consider any Insert operation O that returns false
  - Must prove: ∃ a time during O when key was **in** the data structure (can linearize then)
  - Since we return false, we **do** find the key we are searching for in node **u** (but it might be deleted!)
  - Key idea: even if **u** is deleted, it must be in the list **at some time t** during O (or we couldn't reach it)
    - **Assume u** is **never** in the list at any time during O
    - Either u was inserted after O (can't find it), or deleted before O and never reinserted (can't find it)…
    - Contradiction in either case, so assumption must be wrong… So a valid LP exists.

This is only an intuitive argument!

Also need to argue in cases where we **do not** find the key! (Harder!) And for Delete…

To be theoretically rigorous here, you typically first prove basic list invariants, then prove inductively that that **each** node you find during a traversal was in the list at some time during the traversal…

WHAT IF WE HAVE DIFFERENT TYPES OF SEARCHES?

- Could imagine an application that wants a doubly linked list so:
  - Some threads can search left-to-right (containsLR)
  - Some threads can search right-to-left (containsRL)
- Can we linearize such an algorithm?