MULTICORE PROGRAMMING

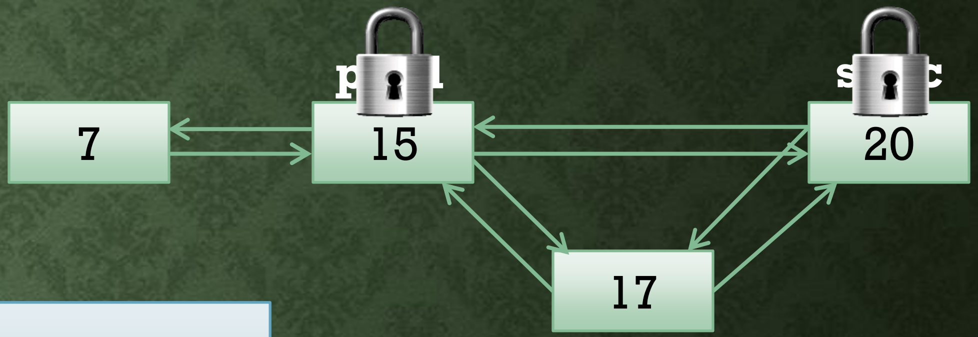
Higher level sync primitives: Doubly-linked list via **k-word CAS**

Lecture 10

Trevor Brown

RECALL: TRAVERSING A DOUBLY-LINKED LIST WITHOUT LOCKING NODES?

Insert(17)



Works if traversals are **only left-to-right**

- 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

- 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

WHAT IF WE HAVE BI-DIRECTIONAL TRAVERSALS?

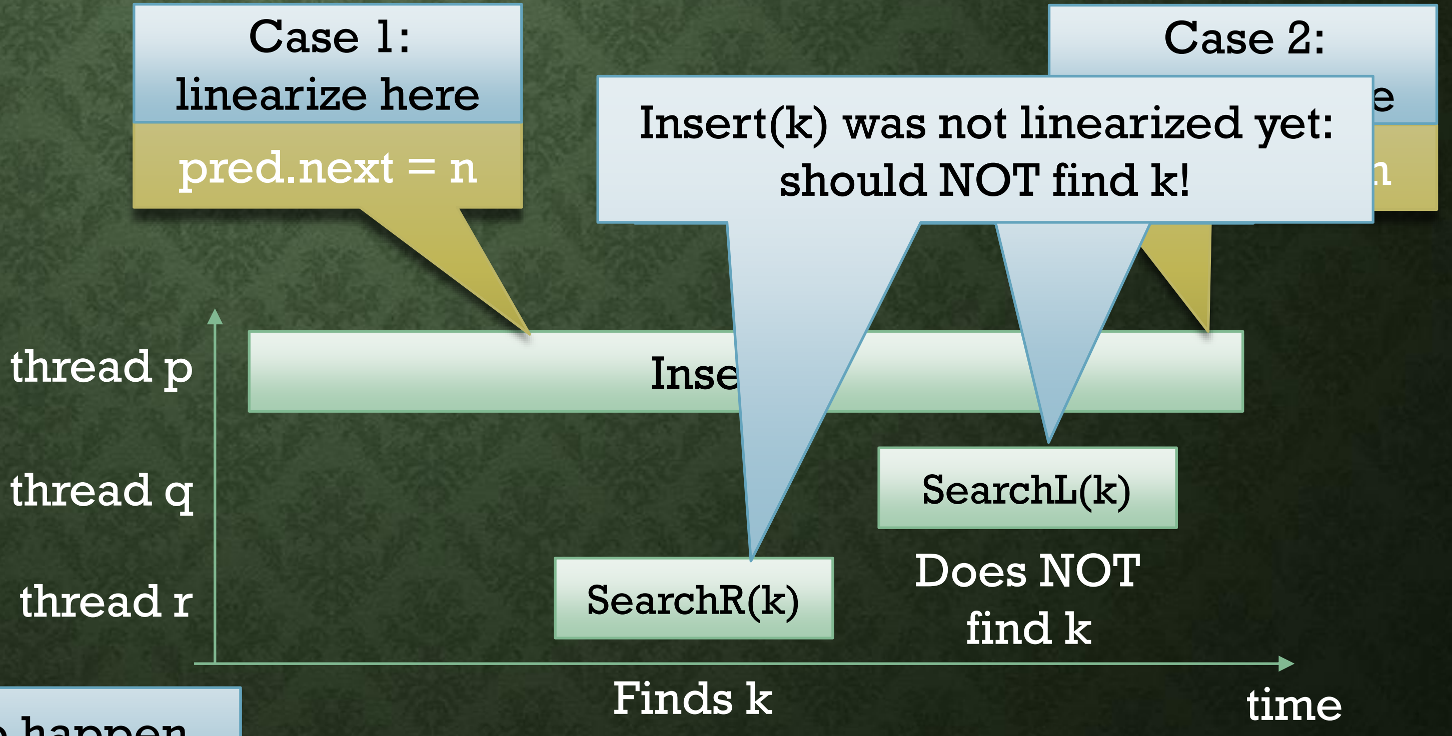
- Could imagine an application that wants a doubly linked list so:
  - Some threads can traverse left-to-right (containsLR)
  - Some threads can traverse right-to-left (containsRL)
- Can we linearize such an algorithm?

LOCK-FREE **BI-DIRECTIONAL** TRAVERSALS COMPLICATE LINEARIZATION

- 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

Need these to happen atomically together…

Where should we linearize a successful insert?



MAKING **TWO** CHANGES APPEAR **ATOMIC** TO A **LOCKLESS TRAVERSAL**

- Something stronger than CAS?
- Double compare-and-swap (DCAS)
  - Like CAS, but on any **two** memory locations
  - DCAS(addr1, addr2, exp1, exp2, new1, new2)
  - **Not** implemented in modern hardware
  - **But** we can implement it in software, using CAS!

DCAS OBJECT: SEQUENTIAL SEMANTICS

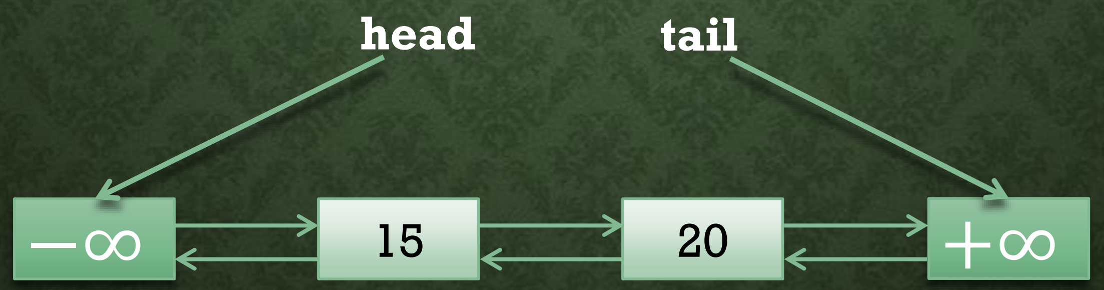
```
DCAS(addr1, addr2, exp1, exp2, new1, new2)
  atomic {
    if (*addr1 == exp1 && *addr2 == exp2) {
      *addr1 = new1;
      *addr2 = new2;
      return true;
    } else return false;
  }
```

```
DCASRead(addr)
  return the value last stored in *addr by a DCAS
```

- Usage - addresses that are modified by DCAS:
  - **must not** be modified with writes/CAS
  - **must** be read using DCASRead

DCAS-BASED DOUBLY-LINKED LIST

- Add **sentinel nodes** to avoid edge cases when list is empty
  - Consequence: **never** update head or tail pointers
- Use DCAS to change pointers (but not keys)
  - Consequence: must use DCASRead to read pointers (but not keys)
  - Note: no need to read head or tail with DCASRead!

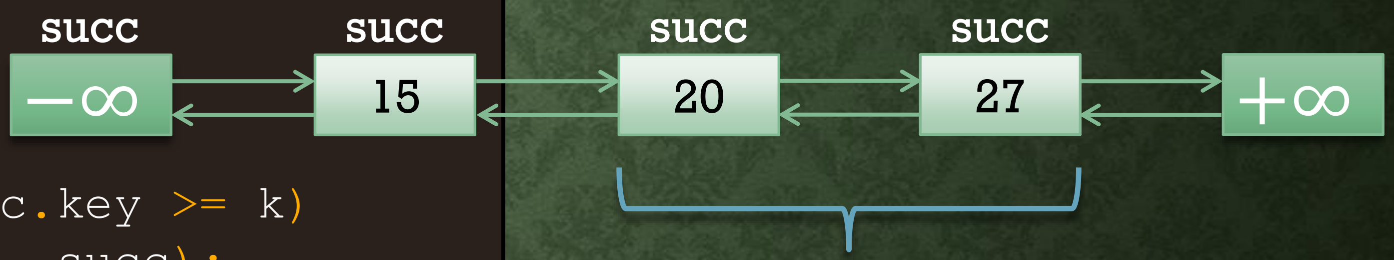


FIRST ATTEMPT AT AN IMPLEMENTATION

Contains(23)

```
pair<node, node> InternalSearch(key_t k)
  pred = head
  succ = head
  while (true)
    if (succ == NULL or succ.key >= k)
      return make_pair(pred, succ);
    pred = succ;
    succ = DCASRead(succ.next);
```

```
bool Contains(key_t k)
  pred, succ = InternalSearch(k);
  return (succ.key == k);
```



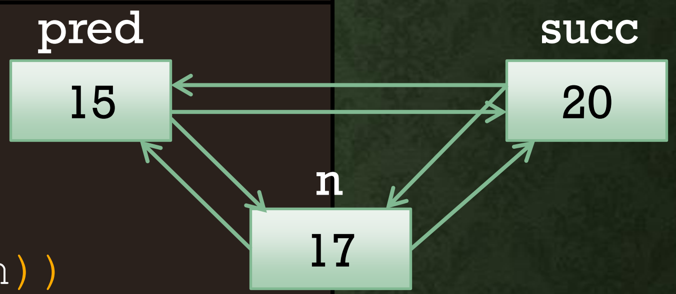
InternalSearch returns pointers to these

Contains(23) sees succ.key != k, and returns false

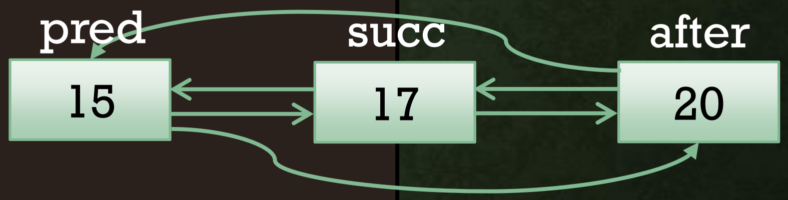
InternalSearch postcondition:
pred.key < k ≤ succ.key

FIRST ATTEMPT AT AN IMPLEMENTATION

```
bool Insert(key_t k)
  while (true)
    pred, succ = InternalSearch(k);
    if (succ.key == k) return false;
    n = new node(k);
    if (DCAS(&pred.next, &succ.prev, succ, pred, n, n))
      return true;
    else delete n;
```




```
bool Delete(key_t k)
  while (true)
    pred, succ = InternalSearch(k);
    if (succ.key != k) return false;
    after = DCASRead(succ.next);
    if (DCAS(&pred.next, &after.prev, succ, succ, after, pred))
      return true; // not covered: how to free succ
```



IS THIS ALGORITHM CORRECT?

- Recall: main difficulties in node-based data structures
  - *Atomically* modifying two or more variables
  - Preventing changes to deleted nodes

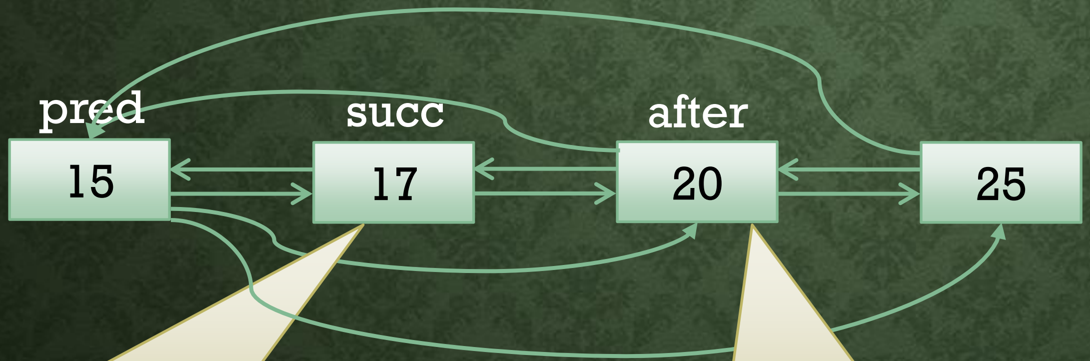
DCAS helps with this

Can we argue deleted nodes don't get changed?

Plausible idea: Once a node is deleted, no node points to it? And we only change nodes that are pointed to by other nodes?

Delete(17)

Delete(20)



Just after a node is deleted, no node points to it… Right?

Nope! This is deleted, but 15 **still** points to it!

So could one of these nodes actually be modified?

A COUNTEREXAMPLE

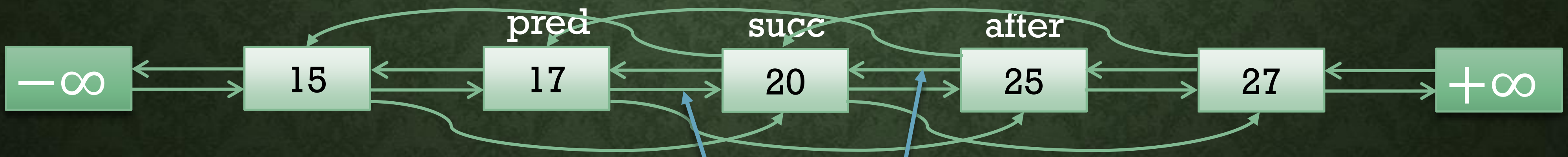
Thread p: start Delete(20), find **pred**, **succ**, **after**

Thread p: **sleep** just before executing DCAS(&pred.next, &after.prev, succ, succ, after, pred)

Thread q: Delete(17)

Thread q: Delete(25)

Thread p: DCAS succeeds, modifying deleted nodes!
Delete(20) returns true, but 20 is not deleted!



p wants to change these **from succ** to after/pred

Let's remove the garbage…

OVERCOMING THIS PROBLEM: MARKING

- Recall: **marking** is often used prevent changes to deleted nodes
- How to atomically change two pointers AND mark other pointers/nodes using DCAS?
- Use an even stronger primitive…
  - k-word compare-and-swap (KCAS)
  - Like a CAS that atomically operations on k memory addresses
  - Can be implemented in software from CAS

KCAS OBJECT: MAKING K CHANGES APPEAR ATOMIC

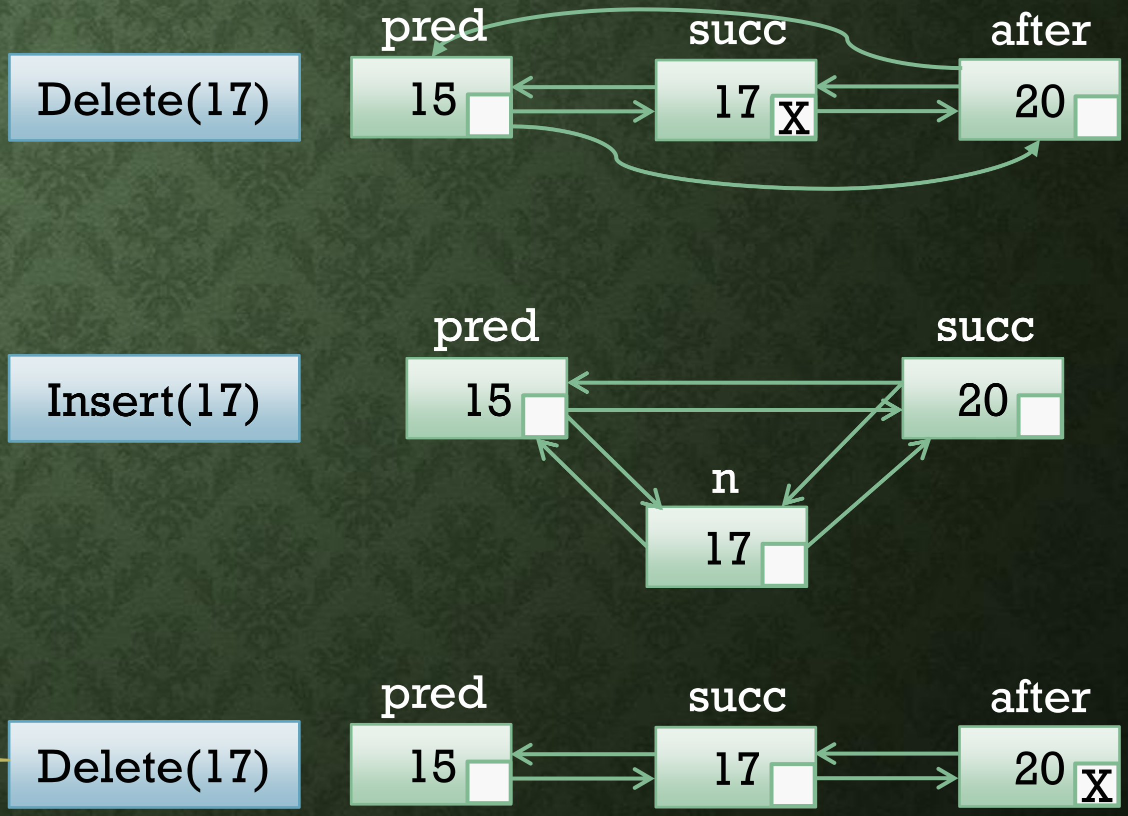
- Operations
  - KCAS($addr_1..addr_k$, $exp_1..exp_k$, $new_1..new_k$)
    - Atomically:
      - If all addresses contain their expected values, sets all addresses to their new values and return true
        else return false
  - KCASRead(addr):
    return the last value stored in addr by a KCAS

- Addresses that are modified by KCAS:
  - **must only** be modified with KCAS
  - **must only** be read with KCASRead

Suppose we are **given KCAS**.
Let's see how to use it.
(We'll see how to actually **implement** KCAS later.)

KCAS-BASED DOUBLY-LINKED LIST

- Based on our attempt using DCAS
- When **deleting** a node,
  use KCAS to also **mark** that node
- When **modifying** or **deleting** any node,
  use KCAS to verify the node is not marked
- Note: since we use KCAS to mark nodes,
  we must use KCASRead to read marks



KCAS fails!

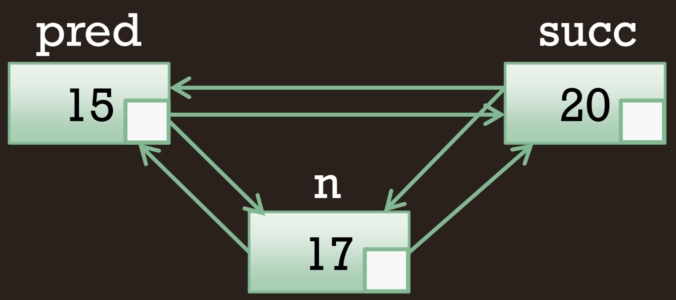
LOCK-FREE DOUBLY-LINKED LIST USING KCAS

```
pair<node, node> InternalSearch(key_t k)
  pred = head
  succ = head
  while (true)
    if (succ == NULL or succ.key >= k)
      return make_pair(pred, succ);
    pred = succ;
    succ = KCASRead(succ.next);
```

```
bool Contains(key_t k)
  pred, succ = InternalSearch(k);
  return (succ.key == k);
```

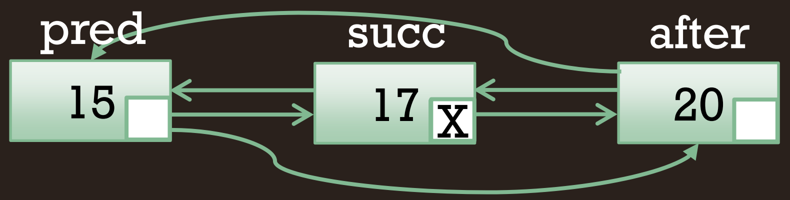
IMPLEMENTATION OF INSERT

```
bool Insert(key_t k)
while (true)
  pred, succ = InternalSearch(k);
  if (succ.key == k) return false;
  n = new node(k);
  if (KCAS(&pred.mark, false, false,
           &succ.mark, false, false,
           &pred.next, succ,  n,
           &succ.prev, pred,  n))
      return true;
  else delete n;
```



IMPLEMENTATION OF DELETE

```
bool Delete(key_t k)
  while (true)
    pred, succ = InternalSearch(k);
    if (succ.key != k) return false;
    after = KCASRead(succ.next);
    if (KCAS(&pred.mark,  false, false,
             &succ.mark,  false, true,
             &after.mark, false, false,
             &pred.next,  succ,  after,
             &after.prev, succ,  pred))
      return true; // not covered yet: freeing succ
```



IS THIS ALGORITHM CORRECT?

- Main challenges
  - *Atomically* modifying two or more variables
  - Preventing changes to deleted nodes
- Let's sketch the correctness argument…

KCAS makes this easy

Marking (with KCAS) makes this easy

SKETCHING THE DIFFICULT ARGUMENT: LINEARIZING CONTAINS

```
pair<node, node> InternalSearch(key_t k)
  pred = head
  succ = head
  while (true)
    if (succ == NULL or succ.key >= k)
      return make_pair(pred, succ);
    pred = succ;
    succ = KCASRead(succ.next);
```

```
bool Contains(key_t k)
  pred, succ = InternalSearch(k);
  return (succ.key == k);
```

Where to linearize Contains that returns true?

Prove there exists a time during Contains when succ is in the list, and linearize then

Where to linearize Contains that returns false?

Prove there exists a time during Contains when: pred and succ are **both** in the list **and** point to each other.

Implies 16 through 19 can't be in the list…

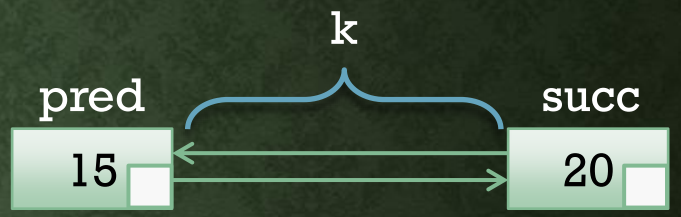
A CONTAINS THAT RETURNS TRUE

- Observation: we reached succ (which contains k) by following pointer pred.next
- Case 1: Suppose at the time we read pred.next, pred was in the list
  - Then, at that time, succ was also in the list.
  - So, at that time, k was in the list. Linearize then!
- Case 2: Suppose at the time we read pred.next, pred was already deleted
  - Lemma: pred was deleted **during** our Contains (or else we could not reach it)
  - Since nodes are not changed after they are deleted (---thanks, marking!), pred.next must have pointed to succ just before it was deleted, which was during our Contains – linearize at that time!

To be theoretically rigorous here, typically you'd prove several claims at once inductively:
each node you found was in the list at some time during your InternalSearch,
deleted nodes are never modified or reinserted into the data structure,
the data structure is always a list (no cycles) ordered by keys, etc…

A CONTAINS THAT RETURNS <u>FALSE</u>

- Observation: we reached succ by following pred.next
- Case 1: Suppose at the time we followed pred.next, pred was in the list
  - Then, at that time, pred and succ were both in the list, and we have pred.key < k <= succ.key from InternalSearch
  - Linearize at that time!
- Case 2: Suppose at the time we followed pred.next, pred was already deleted
  - Lemma: pred was deleted during our Contains
  - Since deleted nodes are not changed, pred.next must have pointed to succ **just before pred was deleted**
  - This was during our contains --- linearize at that time!



Prove there exists a time during InternalSearch when: pred and succ were **both** in the list and pred points to succ. Linearize then.

LINEARIZING INSERT

```
bool Insert(key_t k)
  while (true)
    pred, succ = InternalSearch(k);
    if (succ.key == k) return false;
    n = new node(k);
    if (KCAS(&pred.mark, false, false,
             &succ.mark, false, false,
             &pred.next, succ,  n,
             &succ.prev, pred,  n))
        return true;
    else delete n;
```

Where to linearize Insert that returns true?
At its successful KCAS

Where to linearize Insert that returns false?
Prove there exists a time during Insert when succ was in the list, and linearize then (same argument as Contains returning true)

LINEARIZING DELETE

```
bool Delete(key_t k)
  while (true)
    pred, succ = InternalSearch(k);
    if (succ.key != k) return false;
    after = KCASRead(succ.next);
    if (KCAS(&pred.mark,  false, false,
             &succ.mark,  false, true,
             &after.mark, false, false,
             &pred.next,  succ,  after,
             &after.prev, succ,  pred))
        return true;
```

Where to linearize Delete that returns true?

At its successful KCAS

Where to linearize Delete that returns false?

Prove exists a time during Delete when:
pred & succ are **both** in the list, point to each other
(same argument as Contains returning false)