Engineering Robust Server Software

Scalability

Duke UNIVERSITY

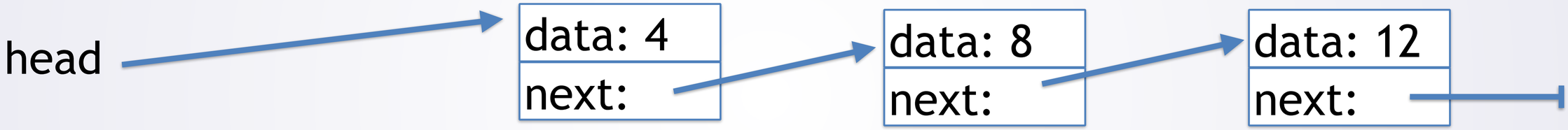
Lock Free Data Structures

- Atomics operations work great when they do what you need
  - E.g., increment an int
- What about more complicated things?
  - E.g., No hardware support for atomically adding to a BST
- Lock Free Data Structures
  - Good when few write conflicts
  - Generally based on atomic CAS
  - Freeing memory makes things hard
    - Much easier if we don't need to free things (GCed languages)

Lock Free LinkedList

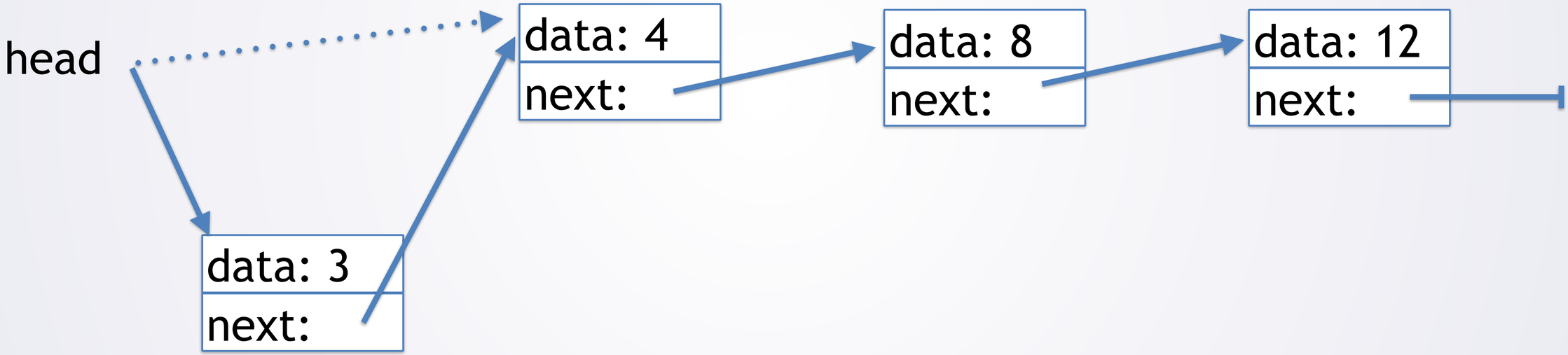
```
class LinkedList {
  class Node{
  public:
    const int data;
    std::atomic<Node *> next;
    Node(int d): data(d), next(nullptr) { }
    Node(int d, Node * n): data(d), next(n) { }
    ~Node() { }
  };
  std::atomic<Node*> head;

…
};
```

Lock Free LinkedList

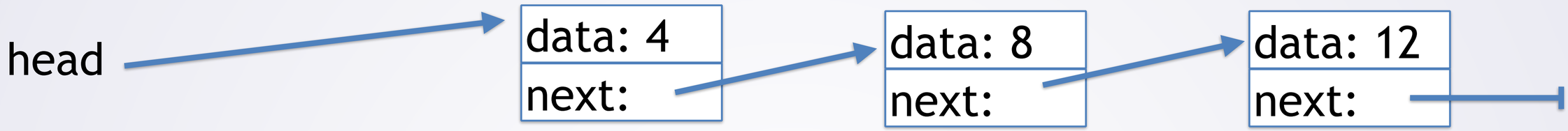


Lock Free LinkedList

addFront(3)

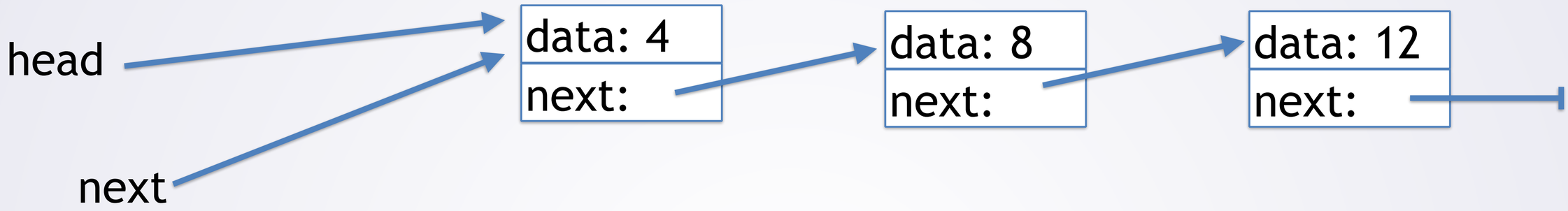


Lock Free LinkedList



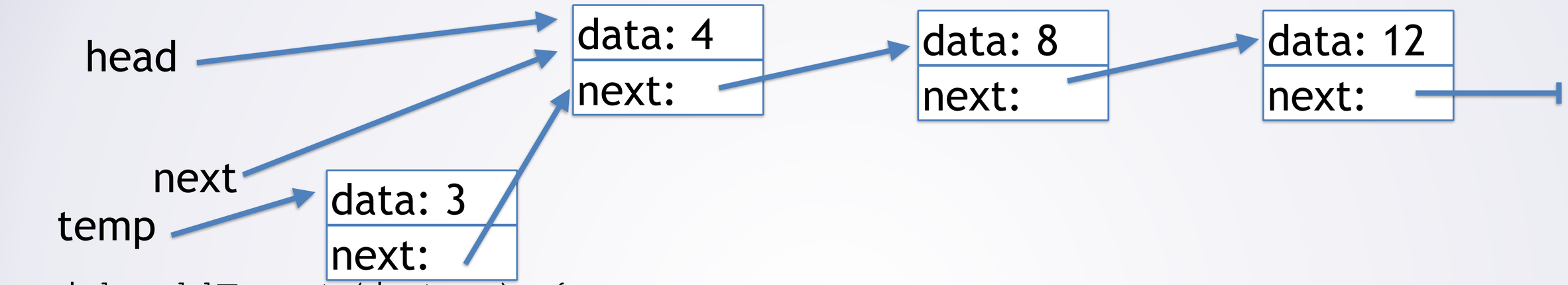

```
void addFront(int x) {
  Node * next = head.load(std::memory_order_???);
  Node * temp = new Node(x, next);
  while (!head.compare_exchange_weak(next,
                                     temp,
                                     std::memory_order_???)) {
    temp->next.store(next, std::memory_order_???);
  }
}
```

Lock Free LinkedList



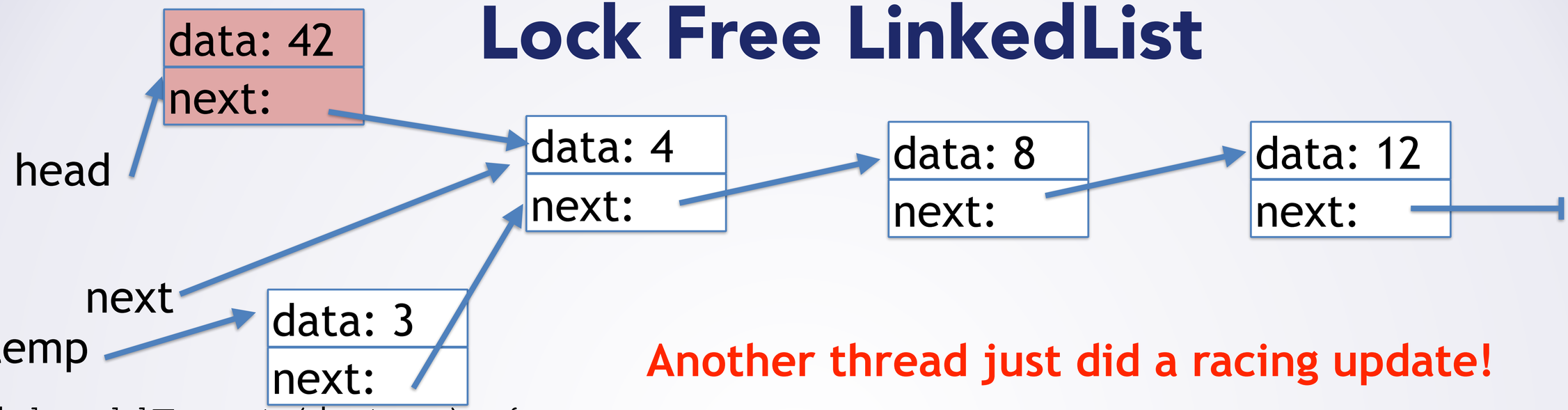

```
void addFront(int x) {
  Node * next = head.load(std::memory_order_???);
  Node * temp = new Node(x, next);
  while (!head.compare_exchange_weak(next,
                                     temp,
                                     std::memory_order_???)) {
    temp->next.store(next, std::memory_order_???);
  }
}
```

Lock Free LinkedList




```
void addFront(int x) {
   Node * next = head.load(std::memory_order_???);
   Node * temp = new Node(x, next);
   while (!head.compare_exchange_weak(next,
                                      temp,
                                      std::memory_order_???)) {
    temp->next.store(next, std::memory_order_???);
   }
}
```

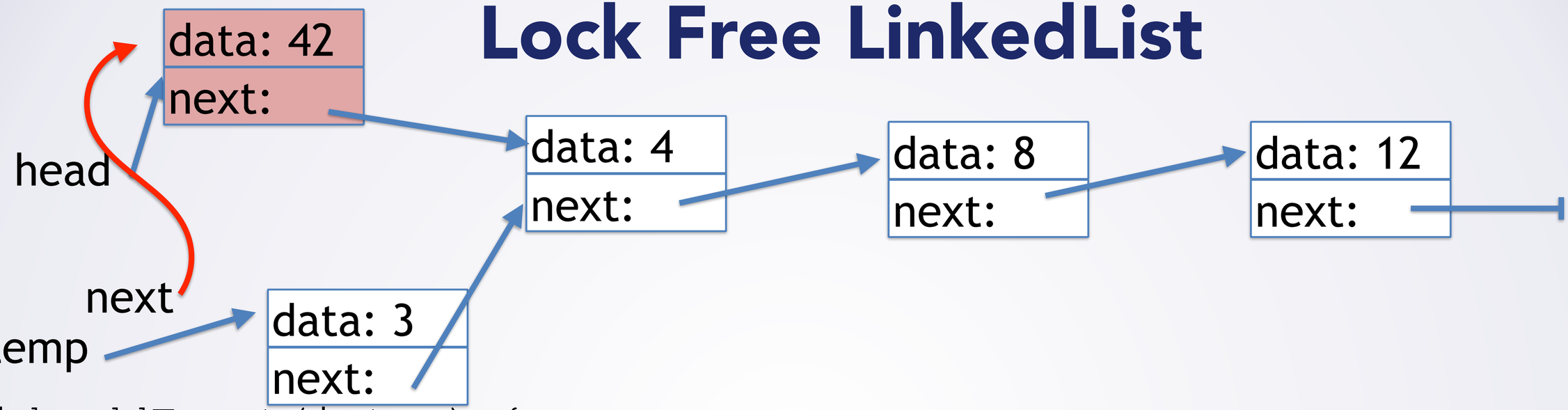
Lock Free LinkedList



Another thread just did a racing update!

```
void addFront(int x) {
  Node * next = head.load(std::memory_order_???);
  Node * temp = new Node(x, next);
  while (!head.compare_exchange_weak(next,
                                     temp,
                                     std::memory_order_???)) {
    temp->next.store(next, std::memory_order_???);
  }
}
```

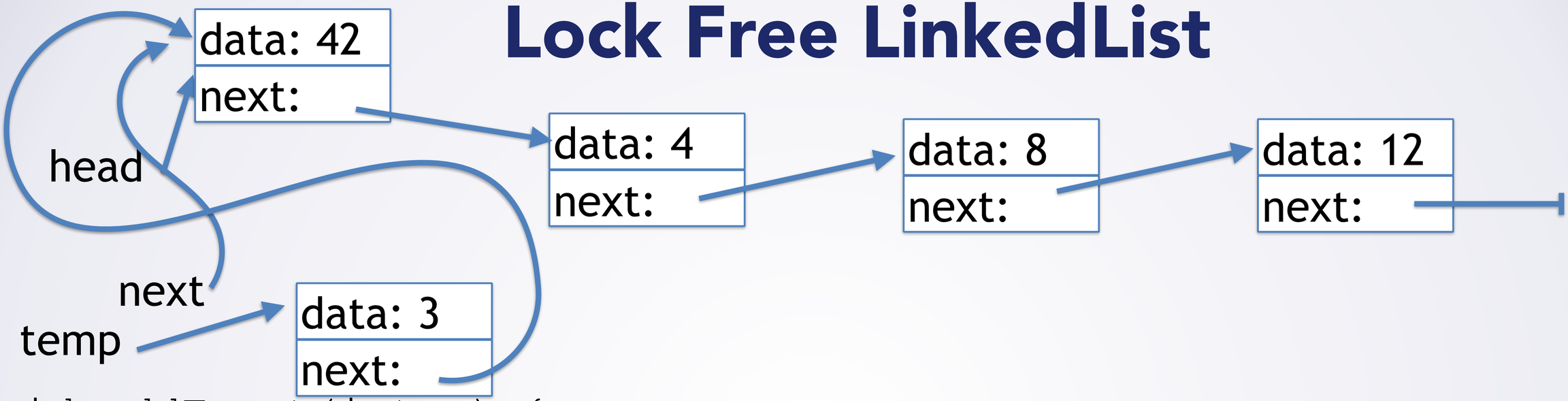
Lock Free LinkedList




```
void addFront(int x) {
   Node * next = head.load(std::memory_order_???);
   Node * temp = new Node(x, next);
   while (!head.compare_exchange_weak(next,
                                      temp,
                                      std::memory_order_???)) {
    temp->next.store(next, std::memory_order_???);
   }
}
```

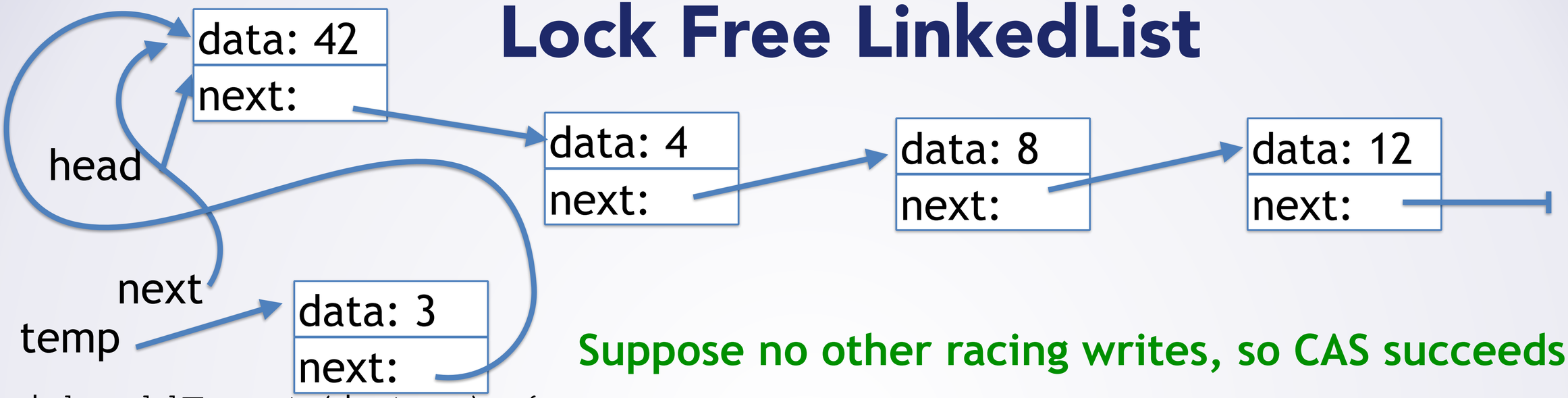
...so CAS fails (head != temp)

Lock Free LinkedList




```
void addFront(int x) {
   Node * next = head.load(std::memory_order_???);
   Node * temp = new Node(x, next);
   while (!head.compare_exchange_weak(next,
                                      temp,
                                      std::memory_order_???)) {
    temp->next.store(next, std::memory_order_???);
   }
}
```

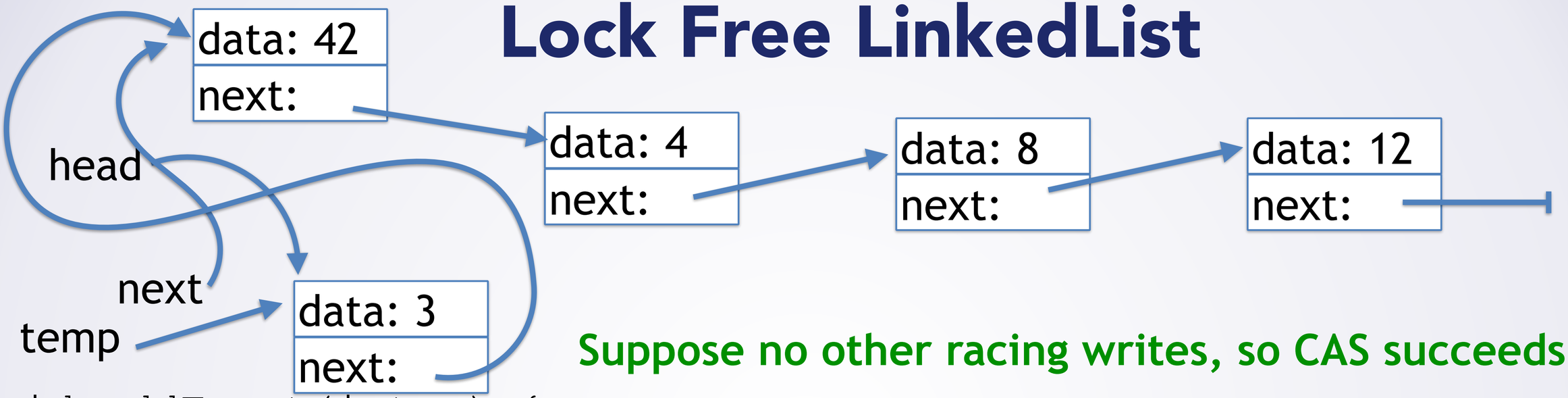
Lock Free LinkedList



Suppose no other racing writes, so CAS succeeds

```
void addFront(int x) {
  Node * next = head.load(std::memory_order_???);
  Node * temp = new Node(x, next);
  while (!head.compare_exchange_weak(next,
                                     temp,
                                     std::memory_order_???)) {
   temp->next.store(next, std::memory_order_???);
  }
}
```

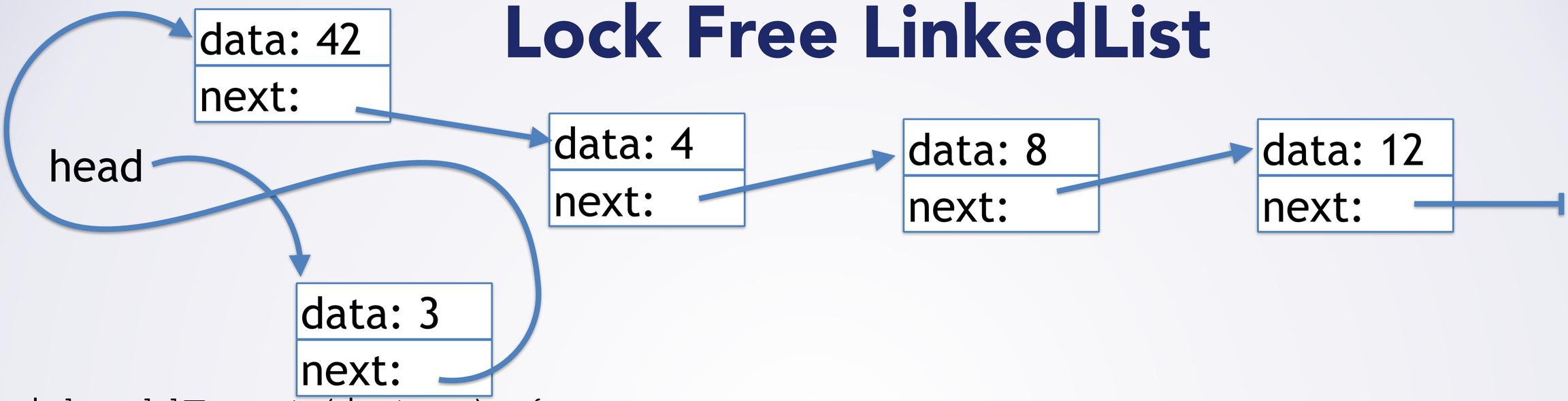
Lock Free LinkedList



Suppose no other racing writes, so CAS succeeds

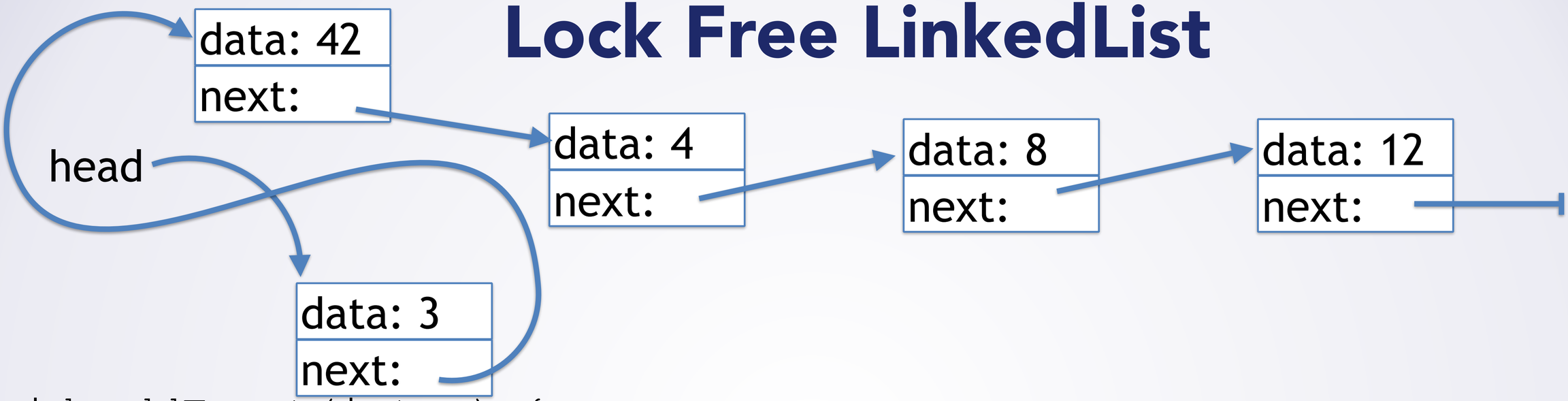
```
void addFront(int x) {
  Node * next = head.load(std::memory_order_???);
  Node * temp = new Node(x, next);
  while (!head.compare_exchange_weak(next,
                                     temp,
                                     std::memory_order_???)) {
   temp->next.store(next, std::memory_order_???);
  }
}
```

Lock Free LinkedList




```
void addFront(int x) {
  Node * next = head.load(std::memory_order_???);
  Node * temp = new Node(x, next);
  while (!head.compare_exchange_weak(next,
                                     temp,
                                     std::memory_order_???)) {
   temp->next.store(next, std::memory_order_???);
  }
}
```

Lock Free LinkedList




```
void addFront(int x) {
  Node * next = head.load(std::memory_order_???);
  Node * temp = new Node(x, next);
  while (!head.compare_exchange_weak(next,
                                     temp,
                                     std::memory_order_???)) {
   temp->next.store(next, std::memory_order_???);
  }
}
```

So what memory order do these need?

Lock Free LinkedList

Thread 0

For the the CASes, what can you say about happens-before?

....
CAS

Thread 1

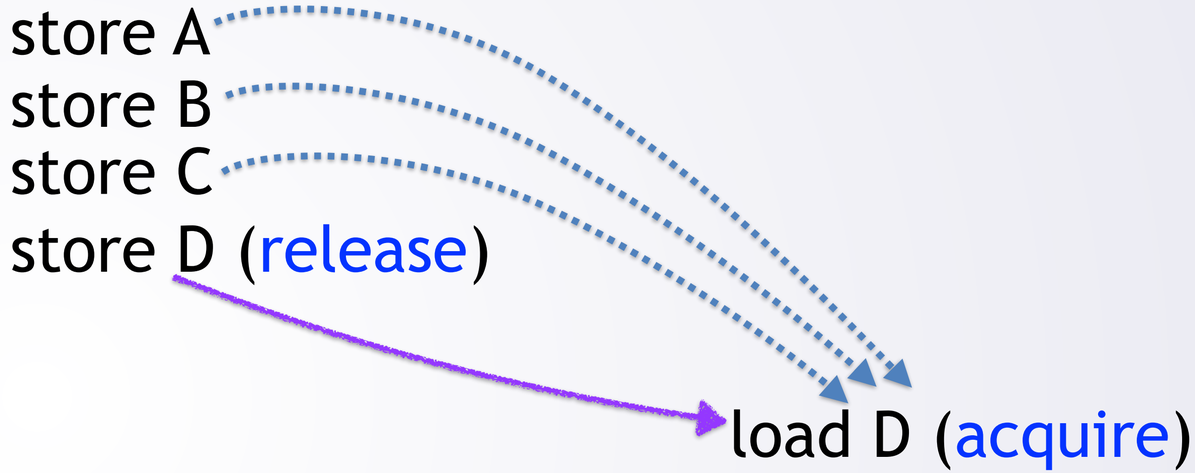
temp = allocate memory
store temp.data = x
store temp.next = next
CAS

What memory ordering does that sound like?

```
void addFront(int x) {
  Node * next = head.load(std::memory_order_???);
  Node * temp = new Node(x, next);
  while (!head.compare_exchange_weak(next,
                                     temp,
                                     std::memory_order_???)) {
   temp->next.store(next, std::memory_order_???);
  }
}
```

Acquire/Release Semantics

- What we want is acquire/release semantics
  - Load: acquire
  - Store: release



- When load (acquire) receives value from store (release)
  - All prior stores in the releasing thread become visible-side effects
  - In acquiring thread (only)
  - Effectively: establishes happens-before for all these stores

Lock Free LinkedList

```
void addFront(int x) {
  Node * next = head.load(std::memory_order_???);
  Node * temp = new Node(x, next);
  while (!head.compare_exchange_weak(next,
                                     temp,
                                     std::memory_order_acq_rel))
   temp->next.store(next, std::memory_order_???);
  }
}
```

Duke UNIVERSITY

Lock Free LinkedList

What about this store? What ordering do we need for it?

```
void addFront(int x) {
  Node * next = head.load(std::memory_order_???);
  Node * temp = new Node(x, next);
  while (!head.compare_exchange_weak(next,
                                     temp,
                                     std::memory_order_acq_rel))
   temp->next.store(next, std::memory_order_???);
  }
}
```

Duke UNIVERSITY

Lock Free LinkedList

What about this store? What ordering do we need for it?

- `temp` is still private to this thread
- the next thing we are going to do is CAS w/ acq_rel
  [will ensure this update is visible to any thread reading the new head value]

So we can use relaxed

```
void addFront(int x) {
  Node * next = head.load(std::memory_order_???);
  Node * temp = new Node(x, next);
  while (!head.compare_exchange_weak(next,
                                     temp,
                                     std::memory_order_acq_rel))
   temp->next.store(next, std::memory_order_relaxed);
  }
}
```

Lock Free LinkedList

What about this load? What ordering do we need for it?

```
void addFront(int x) {
  Node * next = head.load(std::memory_order_???);
  Node * temp = new Node(x, next);
  while (!head.compare_exchange_weak(next,
                                     temp,
                                     std::memory_order_acq_rel))
   temp->next.store(next, std::memory_order_relaxed);
  }
}
```

Duke UNIVERSITY

Lock Free LinkedList

What about this load? What ordering do we need for it?

- Don't care about relationship to other values until after CAS
- CAS will perform acquire [and fail if head has changed]

So we can use relaxed again

```
void addFront(int x) {
  Node * next = head.load(std::memory_order_relaxed);
  Node * temp = new Node(x, next);
  while (!head.compare_exchange_weak(next,
                                     temp,
                                     std::memory_order_acq_rel))
   temp->next.store(next, std::memory_order_relaxed);
  }
}
```

Lock Free LinkedList: addSorted

```
void addSorted(int x) {
  std::atomic<Node *> * ptr = &head;
  Node * newNode = new Node(x);
  Node * nextNode;
  do {
    while (true) {
      nextNode = ptr->load(std::memory_order_acquire);
      if (nextNode == nullptr || nextNode->data > x) {
        break;
      }
      ptr = &nextNode->next;
    }
    newNode->next.store(nextNode, std::memory_order_relaxed);
  }
  while (!ptr->compare_exchange_weak(nextNode,
                                     newNode,
                                     std::memory_order_acq_rel));
}
```

Lock Free LinkedList: addSorted

```
void addSorted(int x) {
  std::atomic<Node *> * ptr = &head;
  Node * newNode = new Node(x);
  Node * nextNode;
  do {
    while (true) {
      nextNode = ptr->load(std::memory_order_acquire);
      if (nextNode == nullptr || nextNode->data > x) {
        break;
      }
      ptr = &nextNode->next;
    }
    newNode->next.store(nextNode, std::memory_order_relaxed);
  }
  while (!ptr->compare_exchange_weak(nextNode,
                                     newNode,
                                     std::memory_order_acq_rel));
}
```

Why acquire?

Why Not Just SC?

```
void addSorted(int x) {
  std::atomic<Node *> * ptr = &head;
  Node * newNode = new Node(x);
  Node * nextNode;
  do {
    while (true) {
      nextNode = ptr->load(std::memory_order_seq_cst);
      if (nextNode == nullptr || nextNode->data > x) {
        break;
      }
      ptr = &nextNode->next;
    }
    newNode->next.store(nextNode, std::memory_order_seq_cst);
  }
  while (!ptr->compare_exchange_weak(nextNode,
                                     newNode,
                                     std::memory_order_seq_cst));
}
```

What if we make it all SC?

Why Not Just Do SC?

- If we use SC, it will be right
  - Too weak of a memory ordering -> bugs on some hardware
- Cost of SC?

Why Not Just Do SC?

- If we use SC, it will be right
  - Too weak of a memory ordering -> bugs on some hardware
- Cost of SC?
  - Slower
  - How much slower?

Why Not Just Do SC?

- If we use SC, it will be right
  - Too weak of a memory ordering -> bugs on some hardware
- Cost of SC?
  - Slower
  - How much slower?
- x86: not too much (already very strong memory consistency)
- Power8: 2x—4x (depending on data size)

Lock Free LinkedList: removeFront

```
bool removeFront(int & outData) {
  Node * temp = head.load(std::memory_order_acquire);
  if (temp == nullptr) {
    return false;
  }
  Node * next = temp->next.load(std::memory_order_relaxed);
  while (!head.compare_exchange_weak(temp,
                                     next,
                                     std::memory_order_acq_rel))
    if (temp == nullptr) { return false; }
    next = temp->next.load(std::memory_order_relaxed);
  }
  if (temp != nullptr) {
    outData = temp->data;
    return true;
  }
  return false;
}
```

What is wrong with this code?

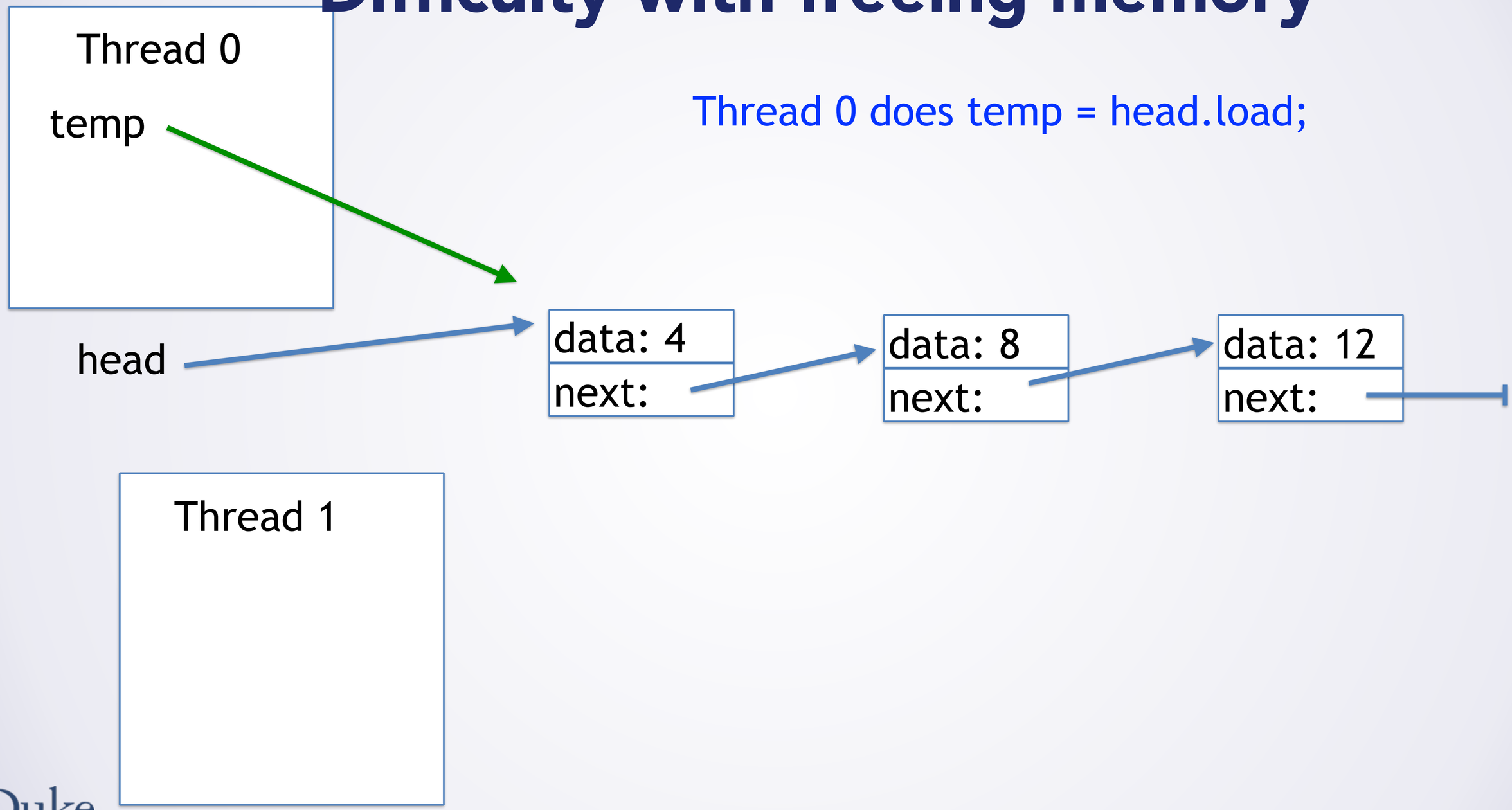
Lock Free LinkedList: removeFront

```
bool removeFront(int & outData) {
  Node * temp = head.load(std::memory_order_acquire);
  if (temp == nullptr) {
    return false;
  }
  Node * next = temp->next.load(std::memory_order_relaxed);
  while (!head.compare_exchange_weak(temp,
                                     next,
                                     std::memory_order_acq_rel))
    if (temp == nullptr) { return false; }
    next = temp->next.load(std::memory_order_relaxed);
  }
  if (temp != nullptr) {
    outData = temp->data;
    return true;
  }
  return false;
}
```

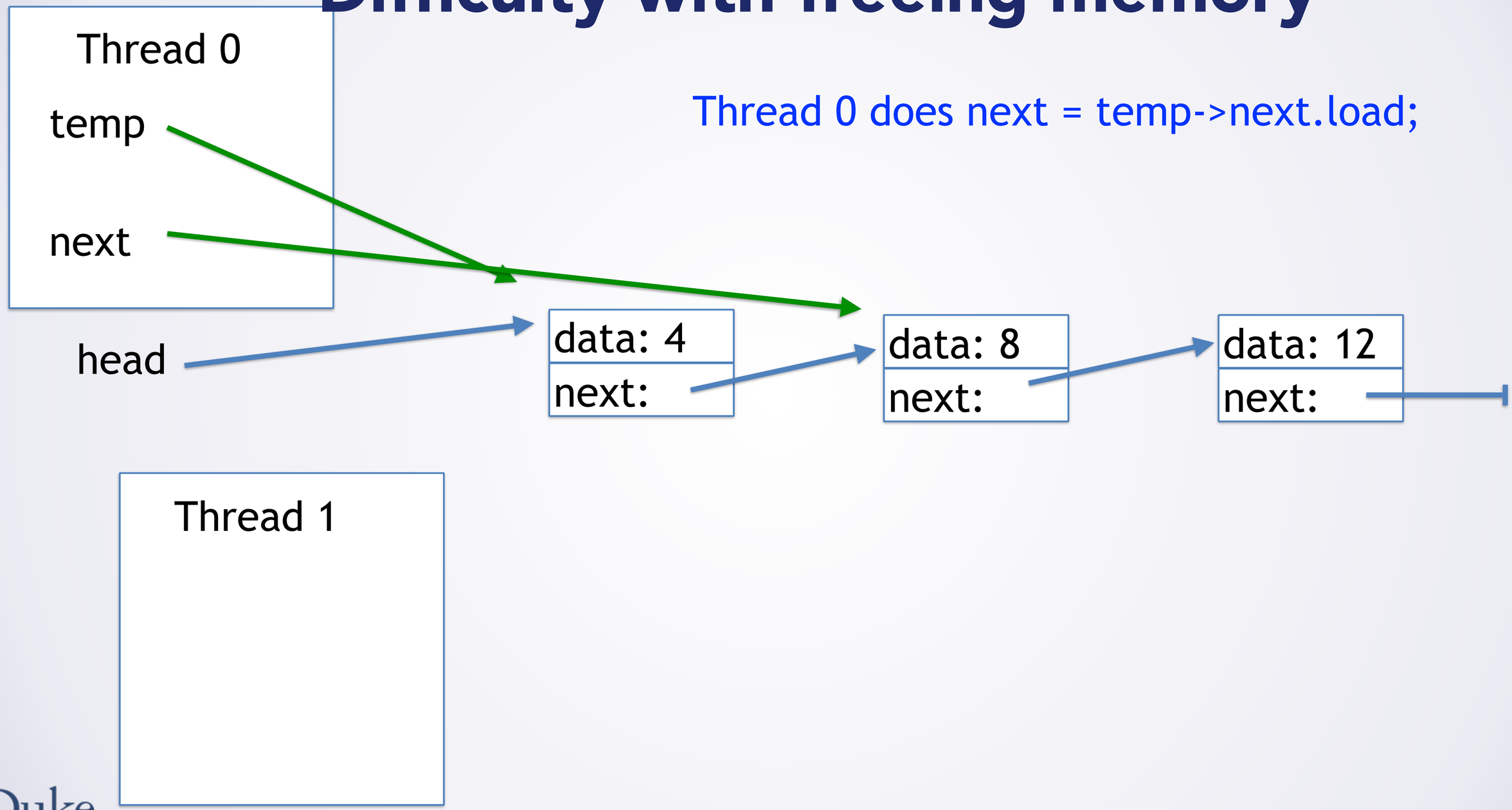
Leak memory here (temp is last reference to node)

… or is it?

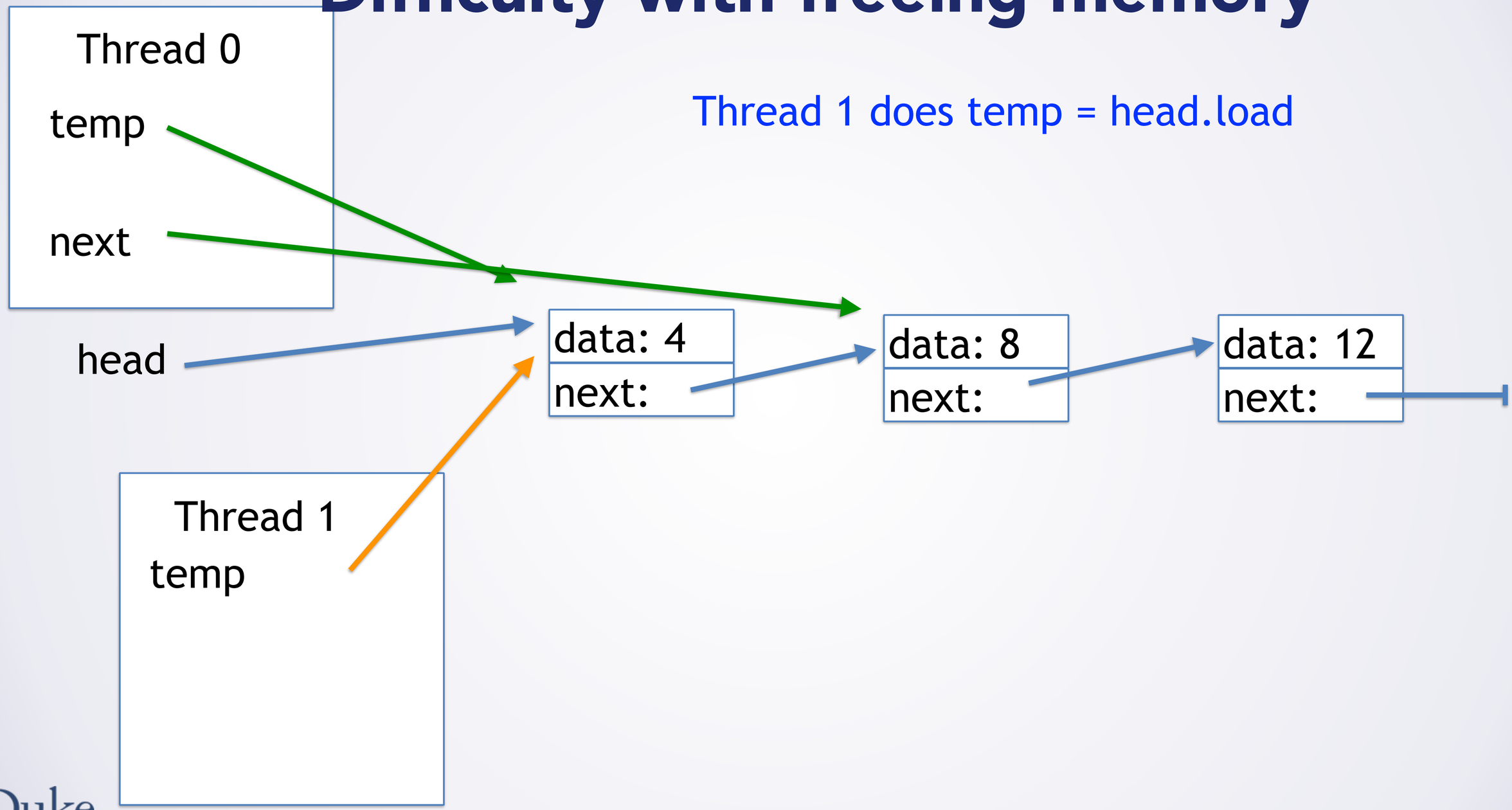
Difficulty with freeing memory



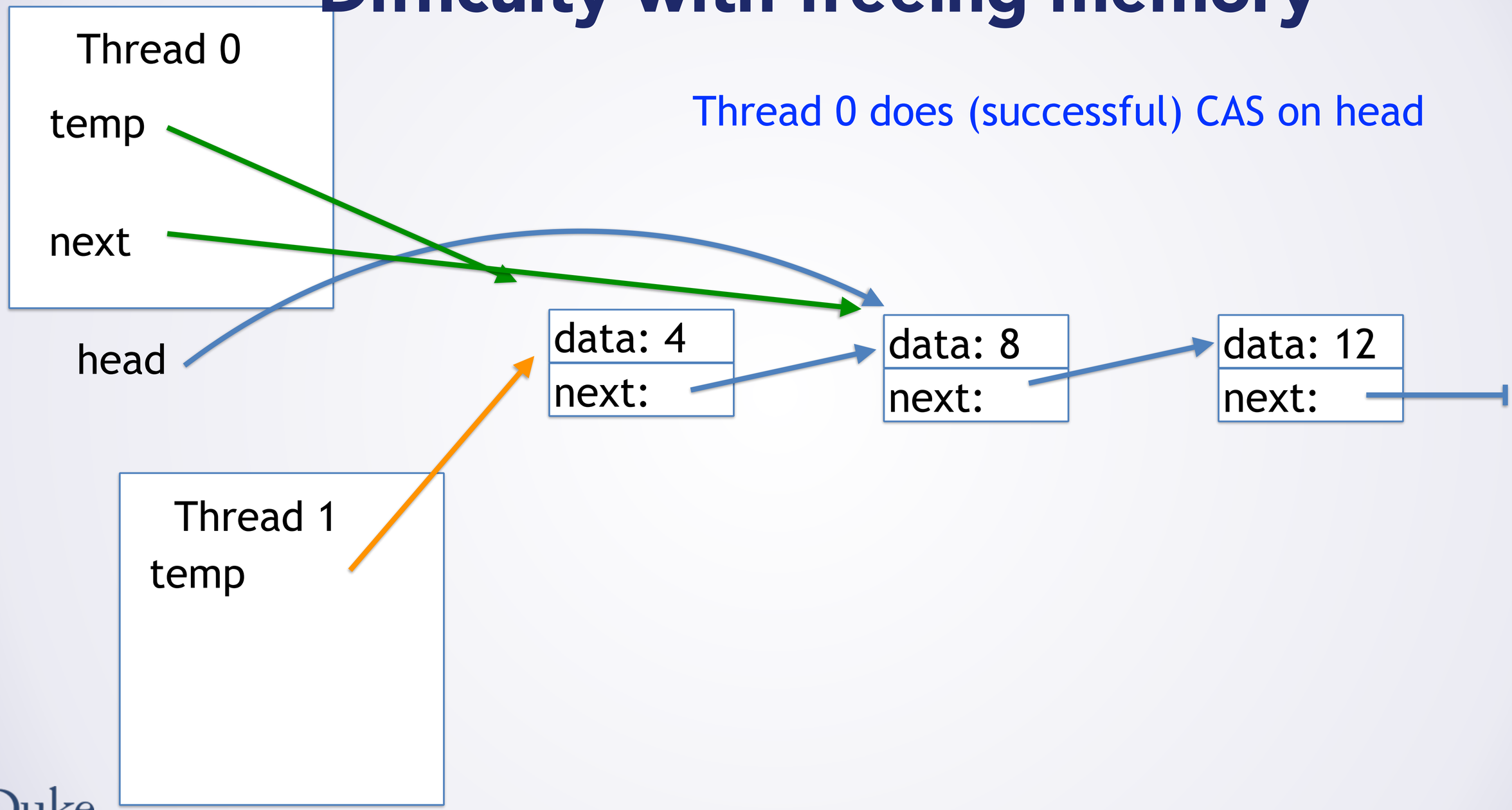
Difficulty with freeing memory



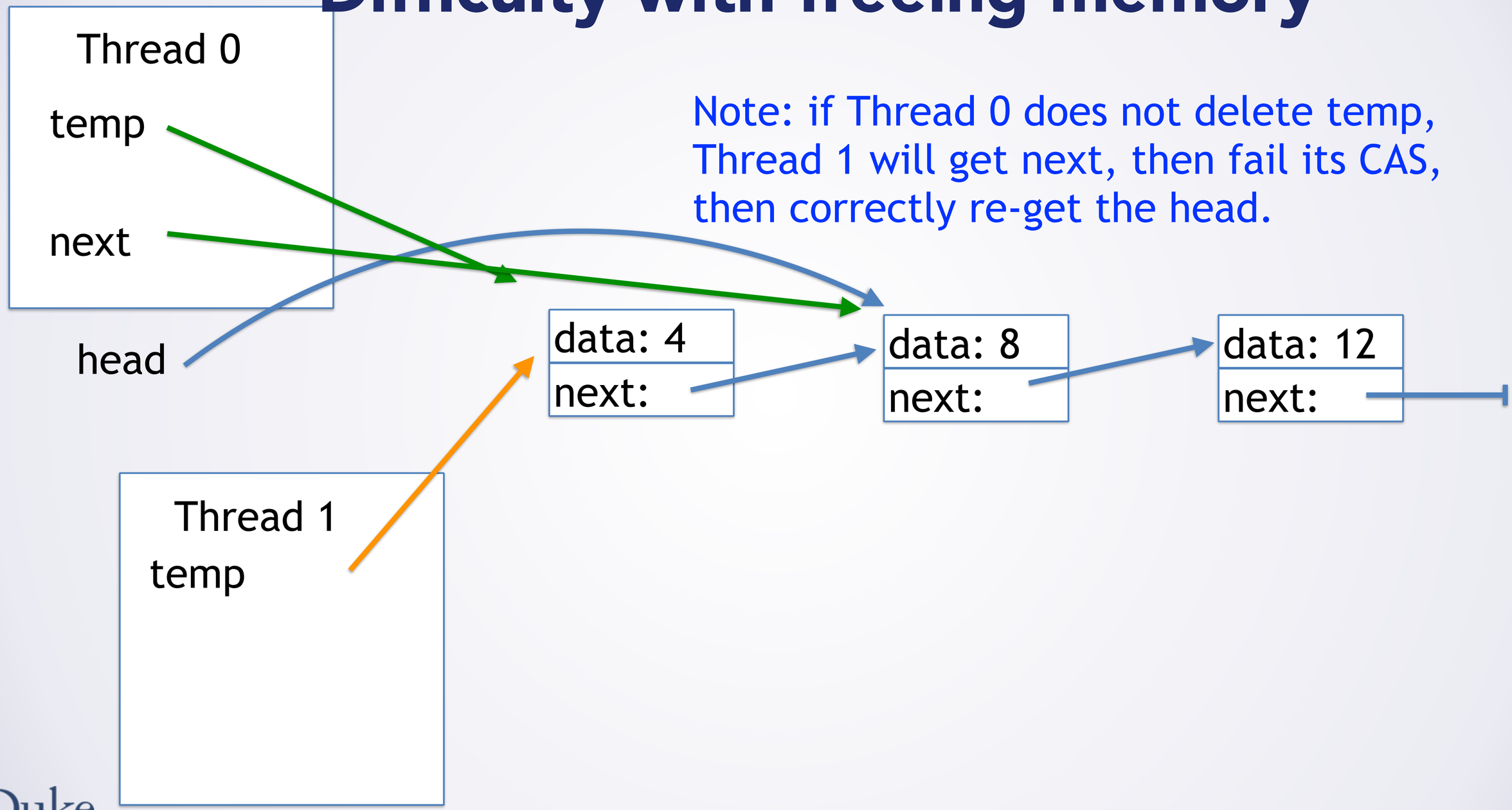
Difficulty with freeing memory



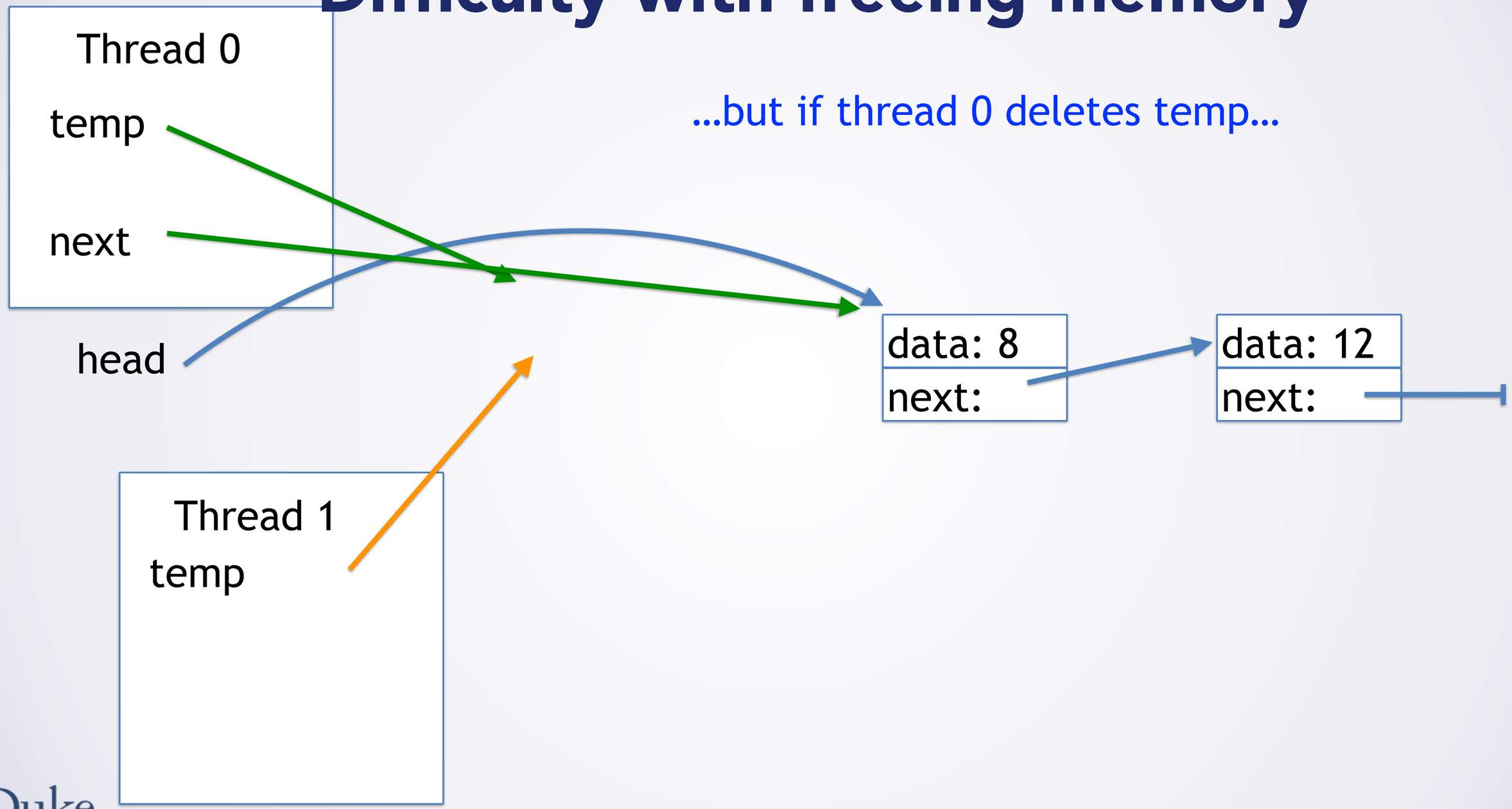
Difficulty with freeing memory



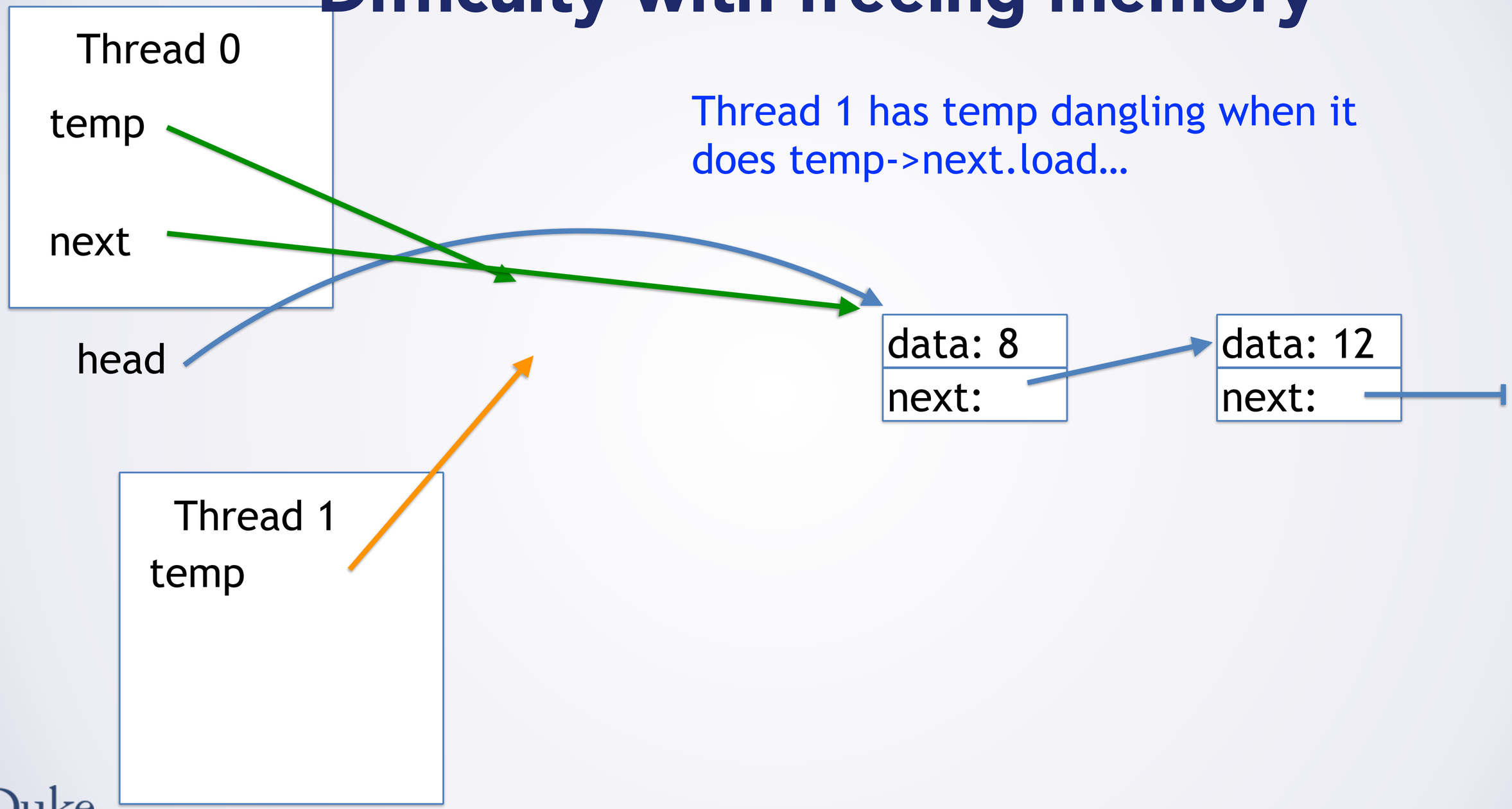
Difficulty with freeing memory



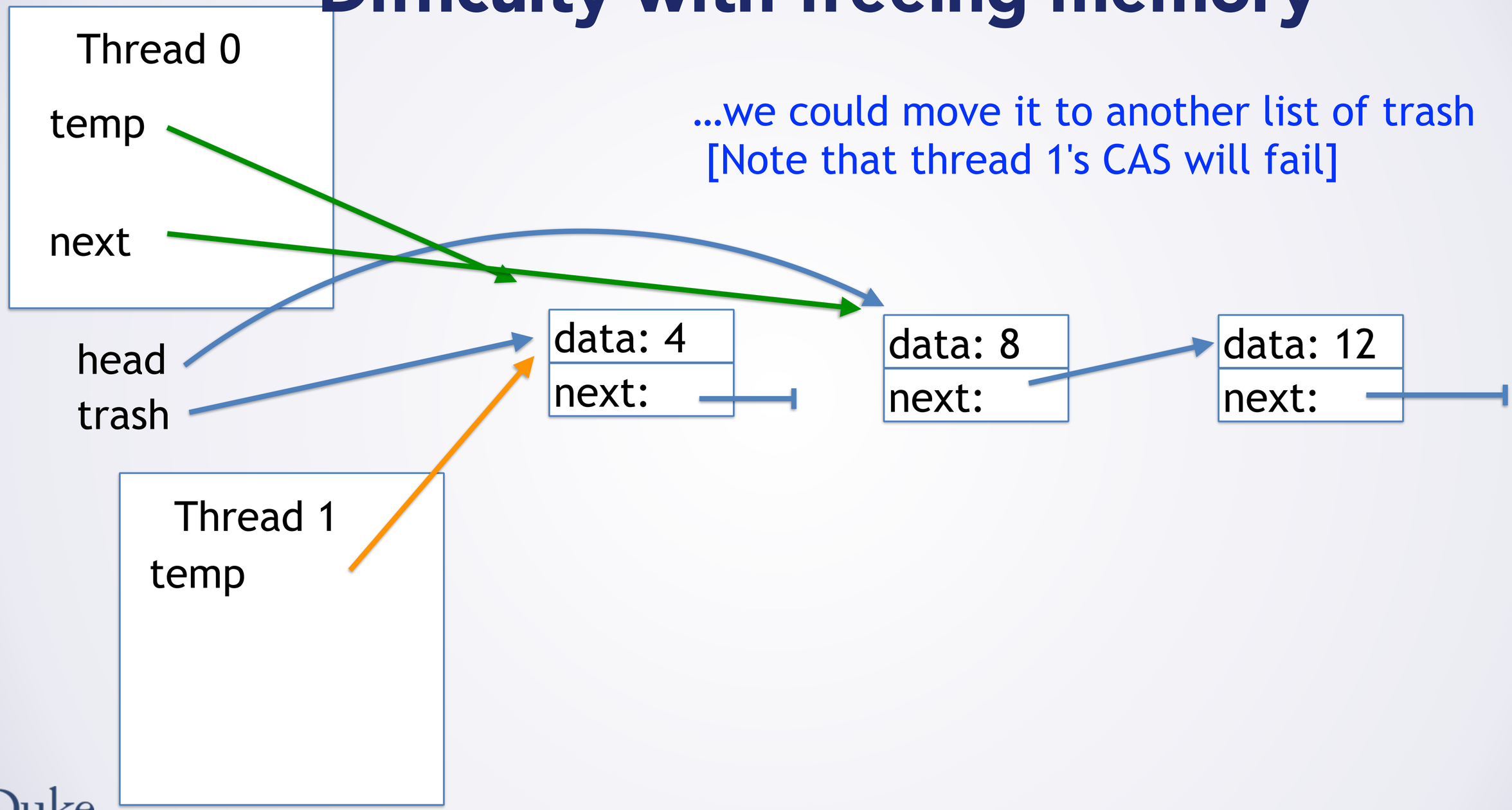
Difficulty with freeing memory



Difficulty with freeing memory



Difficulty with freeing memory



Difficulty Cleaning Up Memory

- How can we clean up our trash list? Delete "later"?
  - How long is later?
  - Must **ensure** no other thread still reference it
- Could we just recycle the nodes?
  - I.e., allocate nodes out of the trash for this same list?
  - ONLY if we can **ensure** no other threads still reference it..
  - Why? Otherwise might have old pointer to address X
    - Reallocate node at X
    - CAS succeeds (X=X) even though we've updated list

Freeing Data in LF DS

- Option 1: count threads operating in DS
  - If count == 1, only this thread -> free nodes
  - Difficulty: May never have only 1 thread in DS
- Option 2: require all threads to finish one operation after update
  - Only can see stale data if in same operation as update
  - Difficulty: Thread may not be doing any operations
- Option 3: track set of threads operating in DS
  - Difficulty: complicated
- Option 4: highly scalable R/W locks
  - R = normal operations, W = exclusive operations [freeing]
  - Difficulty: need high read scalability

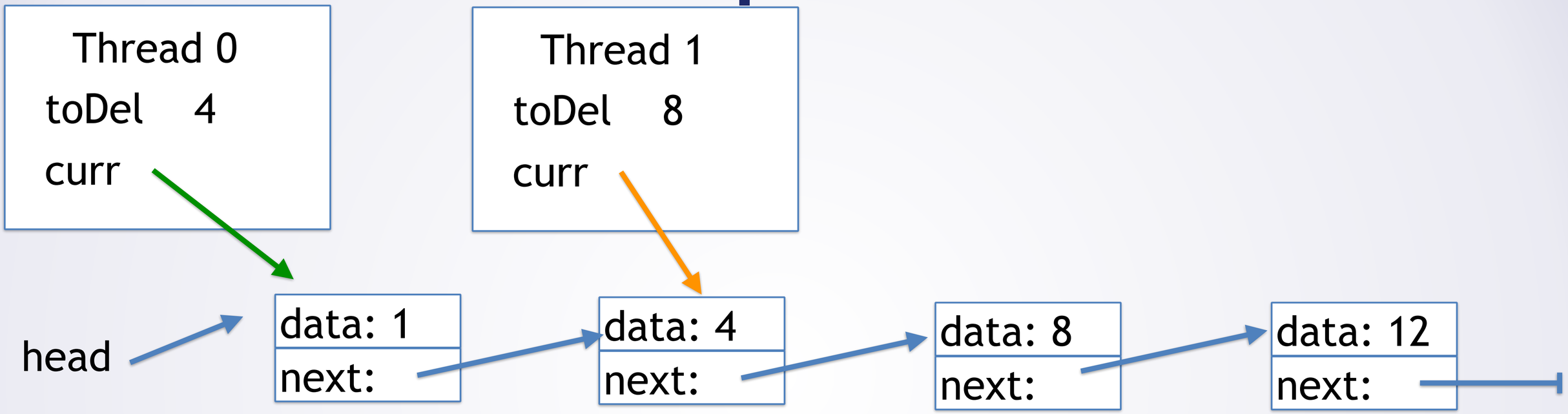
Freeing Data in Lock Free DSes

- GC makes it easy
  - Stop the world
  - Considers root sets from all threads
  - So much simpler in Java…

More Complex Deletions?

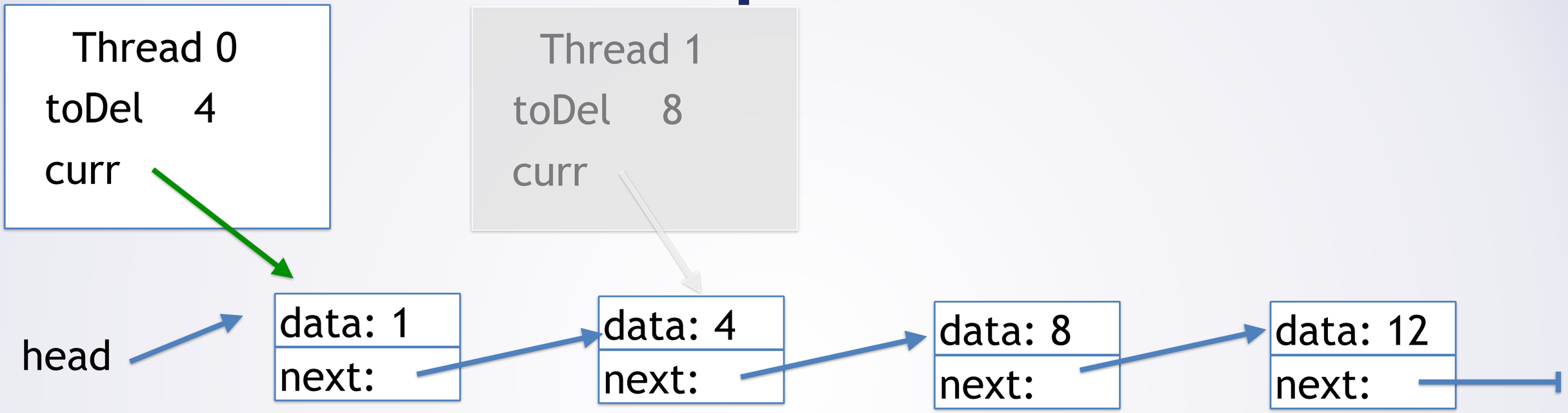
- What about deleting from an arbitrary position?
  - Delete specific value, index, …

More Complex Deletions?



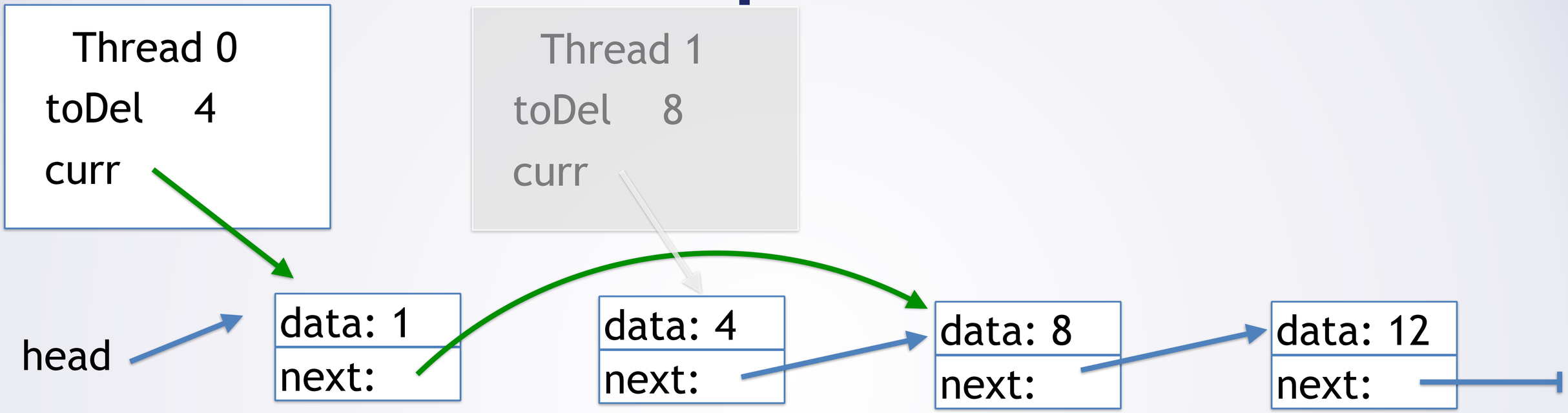
- What about deleting from an arbitrary position?
  - Delete specific value, index, …
- Turns out to be more complex..

More Complex Deletions?



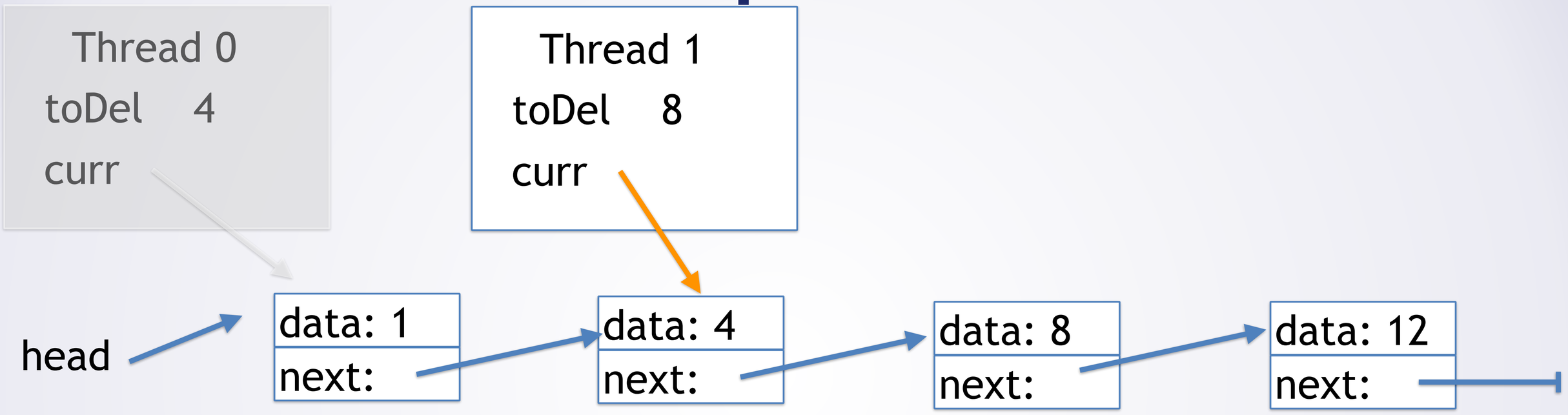
- What would thread 0 do by itself?

More Complex Deletions?



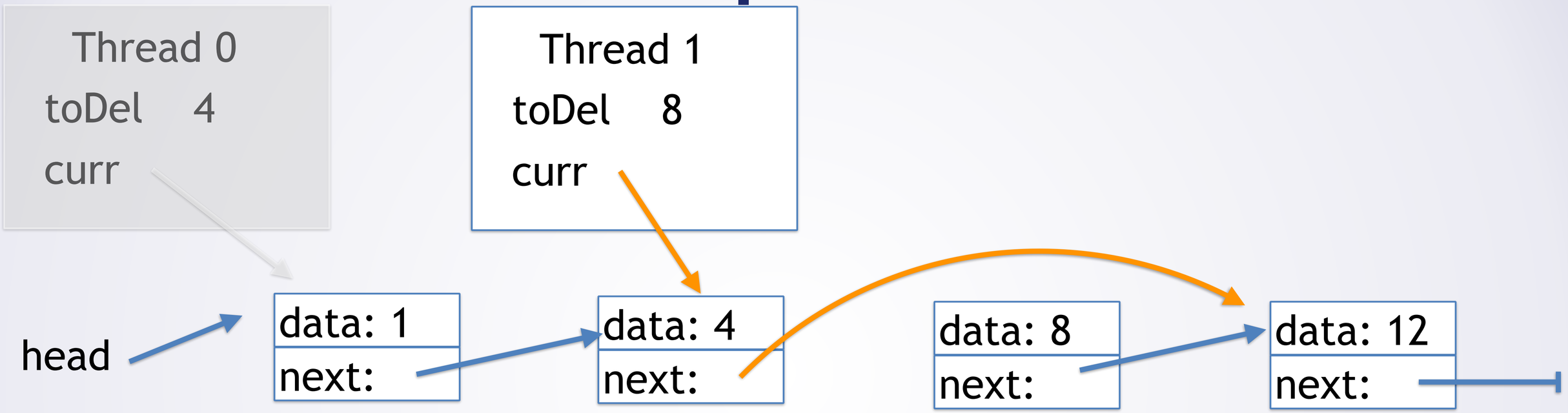
- What would thread 0 do by itself?
  - Atomic CAS is on 1's next field.

More Complex Deletions?



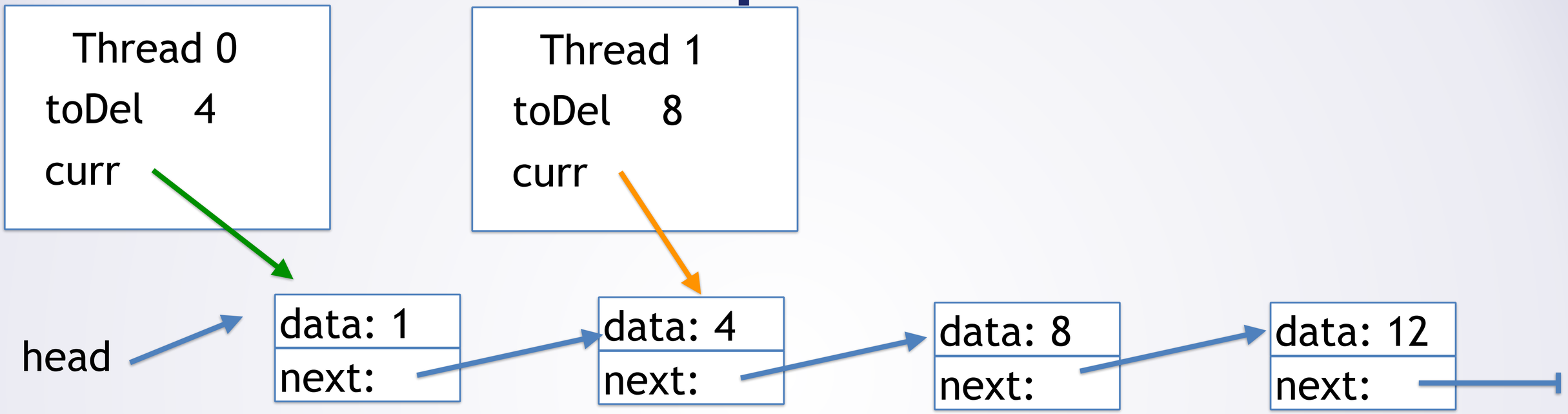
- What would thread 1 do by itself?

More Complex Deletions?



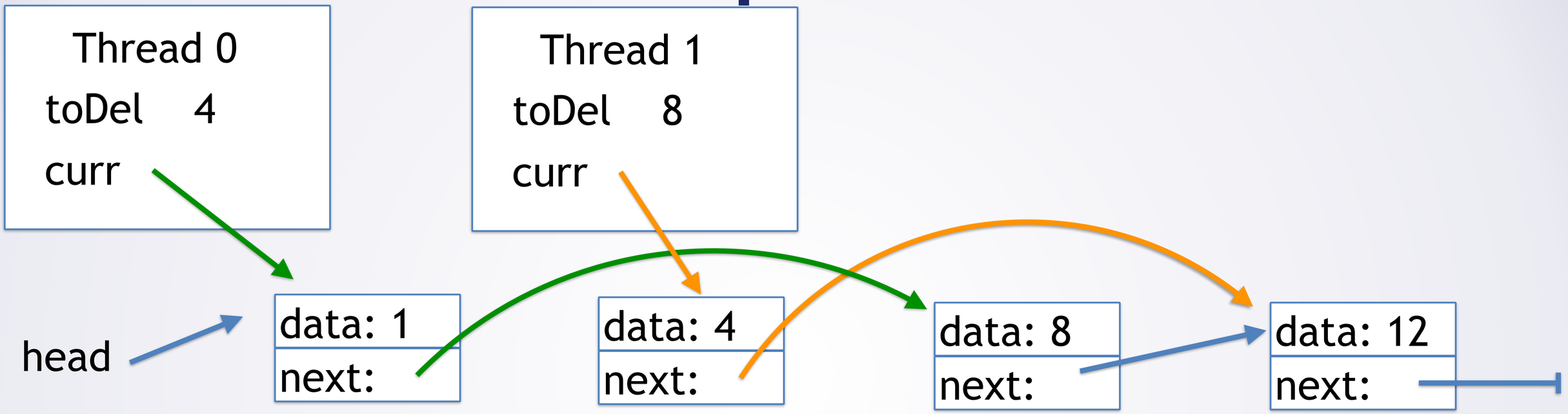
- What would thread 1 do by itself?
  - Atomic CAS is on 4's next

More Complex Deletions?



- What happens if we do both at the same time?
  - Do any CASes fail?

More Complex Deletions?

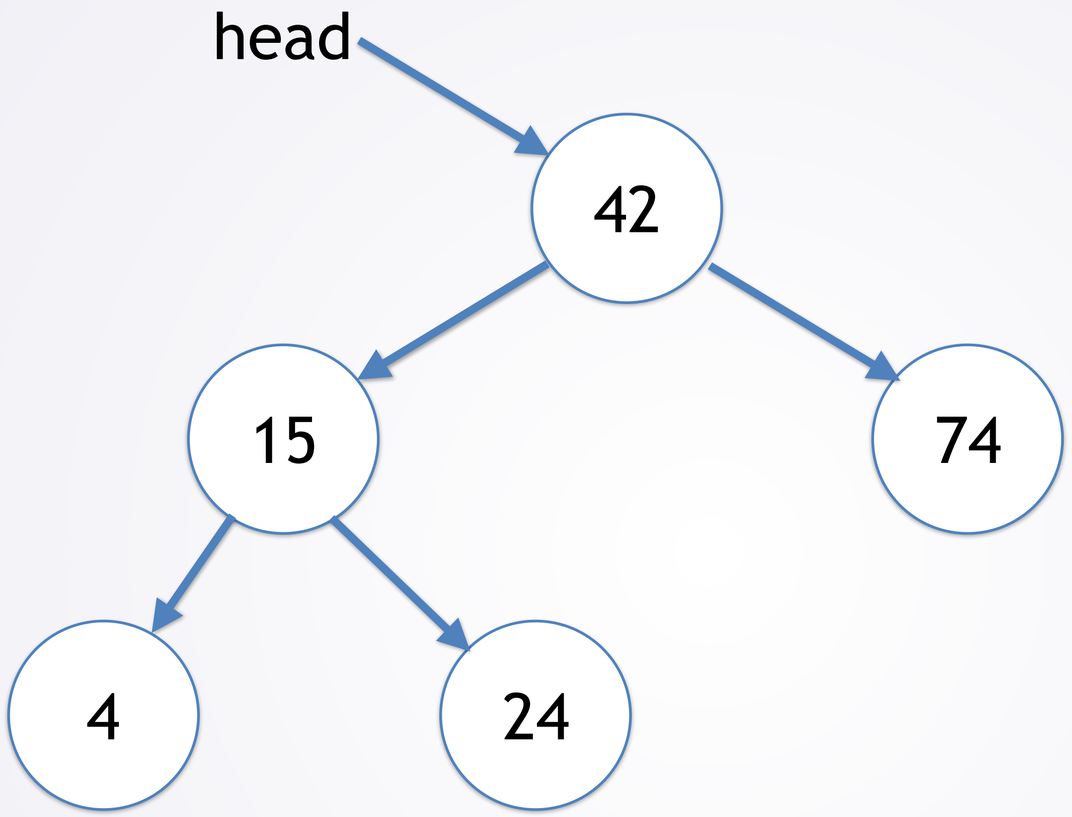


- What happens if we do both at the same time? **Undid 8's delete**
  - Do any CASes fail? **No: both succeed**
  - Similar problem with racing adds + deletes

Note: our **removeFront** only works if we only have **addFront**
(what race is there if we also have **addSorted**?)

Lock Free BST

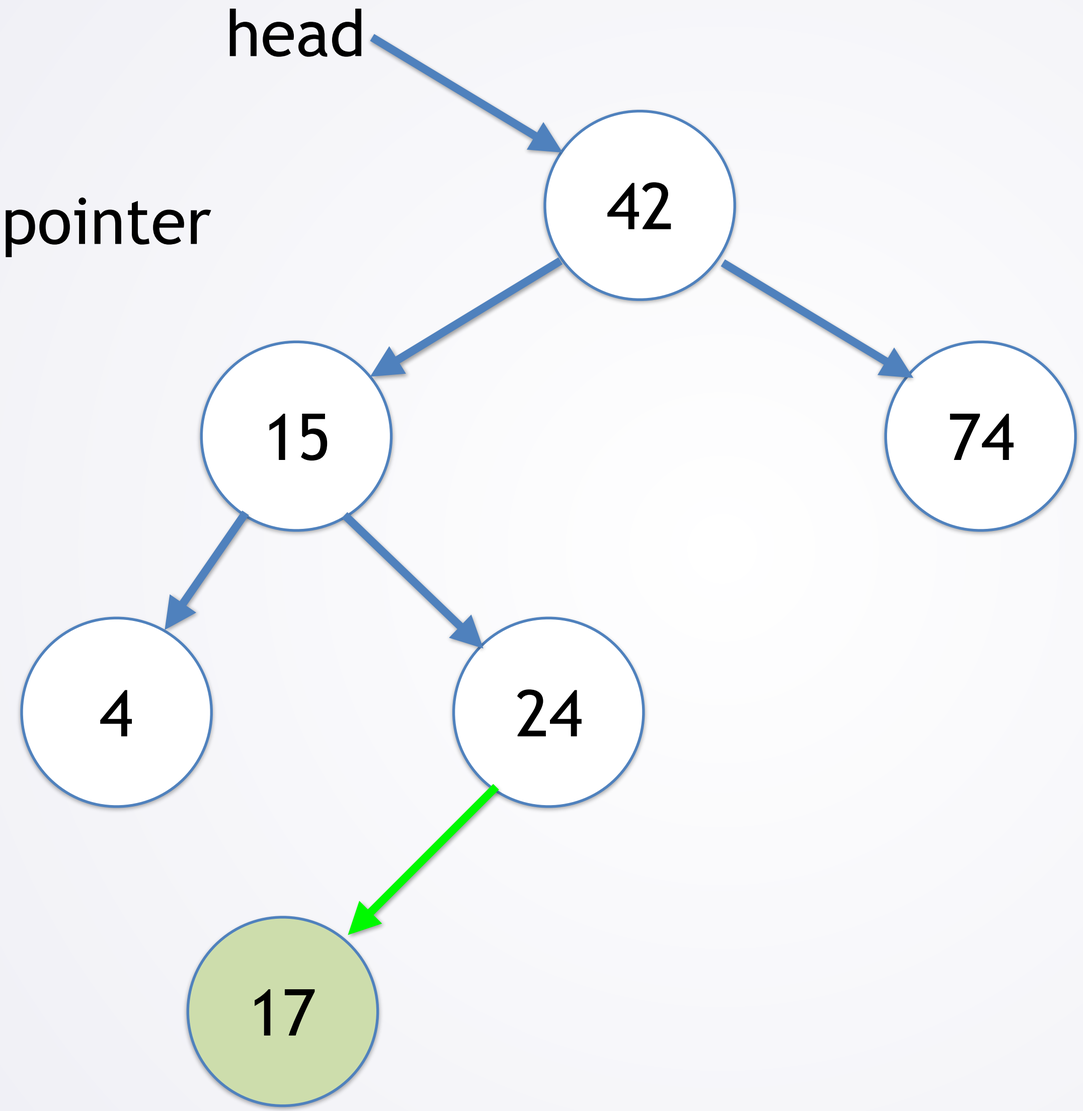
Add 17



Lock Free BST

Add 17

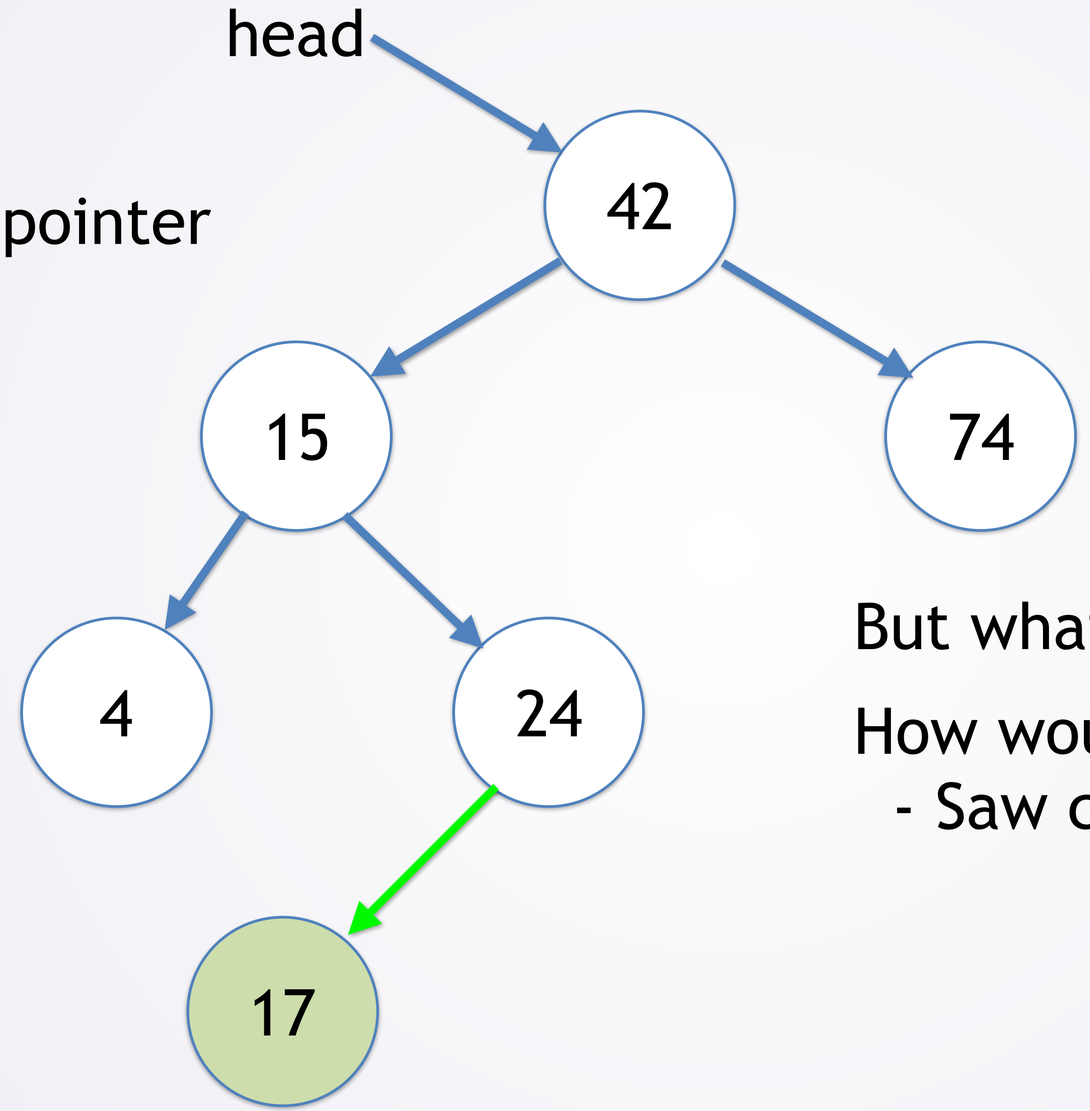
Option 1:
CAS 24's left pointer



Lock Free BST

Add 17

Option 1:
CAS 24's left pointer



But what if we want to rebalance?

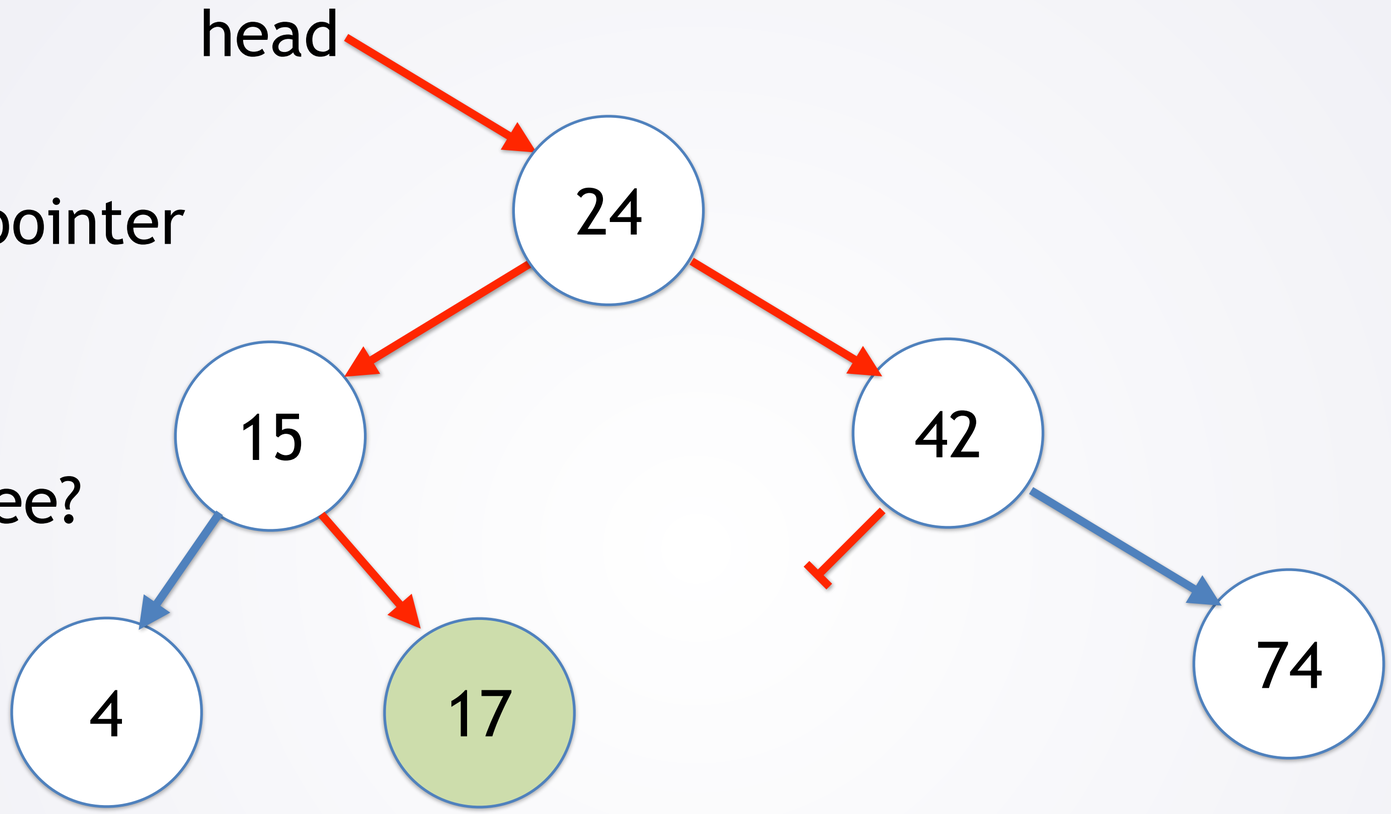
How would deletes work?
- Saw complex for LLs…

Lock Free BST

Add 17

Option 1:
CAS 24's left pointer

Rebalanced.
How would we
do that lock free?



Another Option

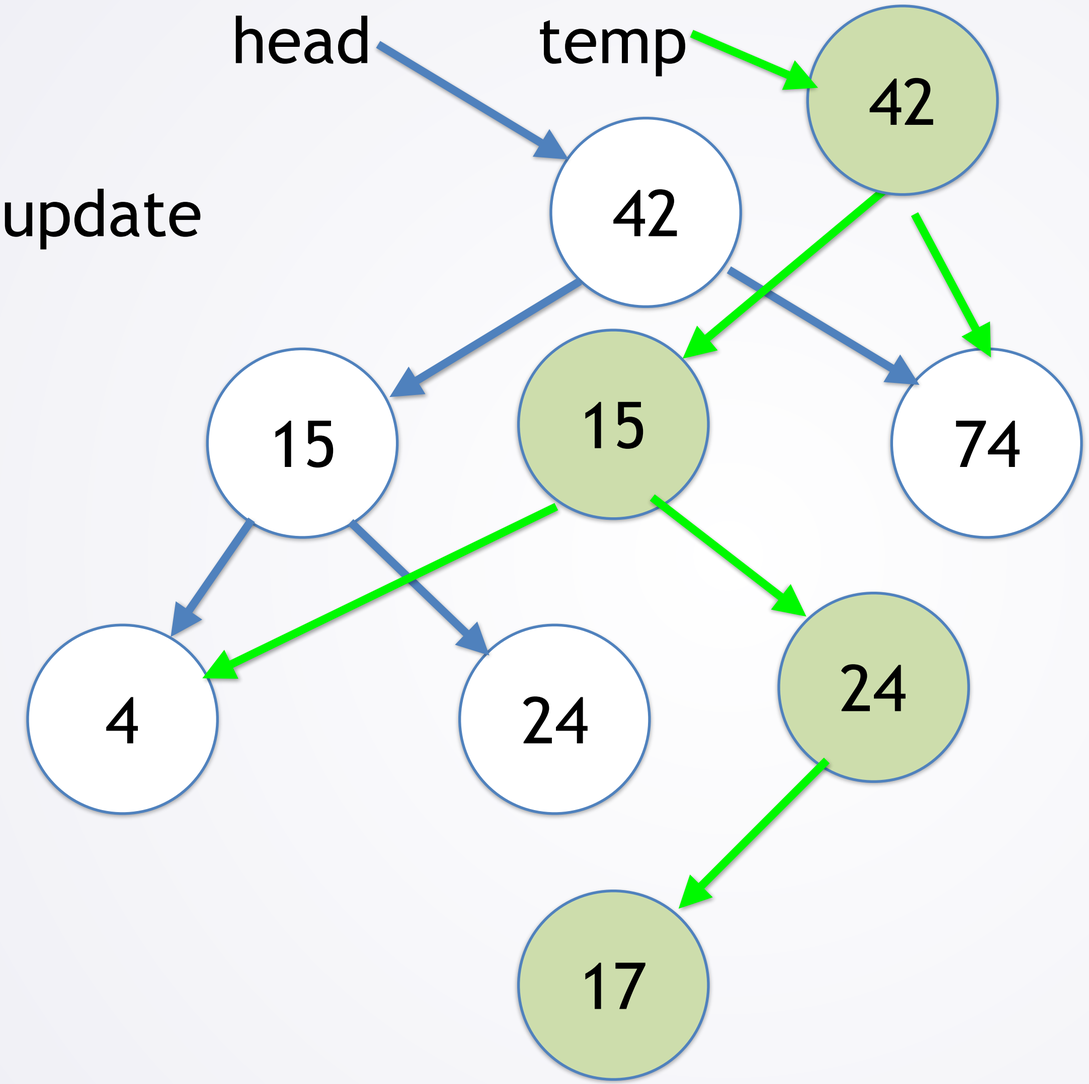
- Option 2:
  - Do functional update, and only CAS the root
  - For any operation (add, or delete)

Lock Free BST

Add 17

Option 2:

Do functional update



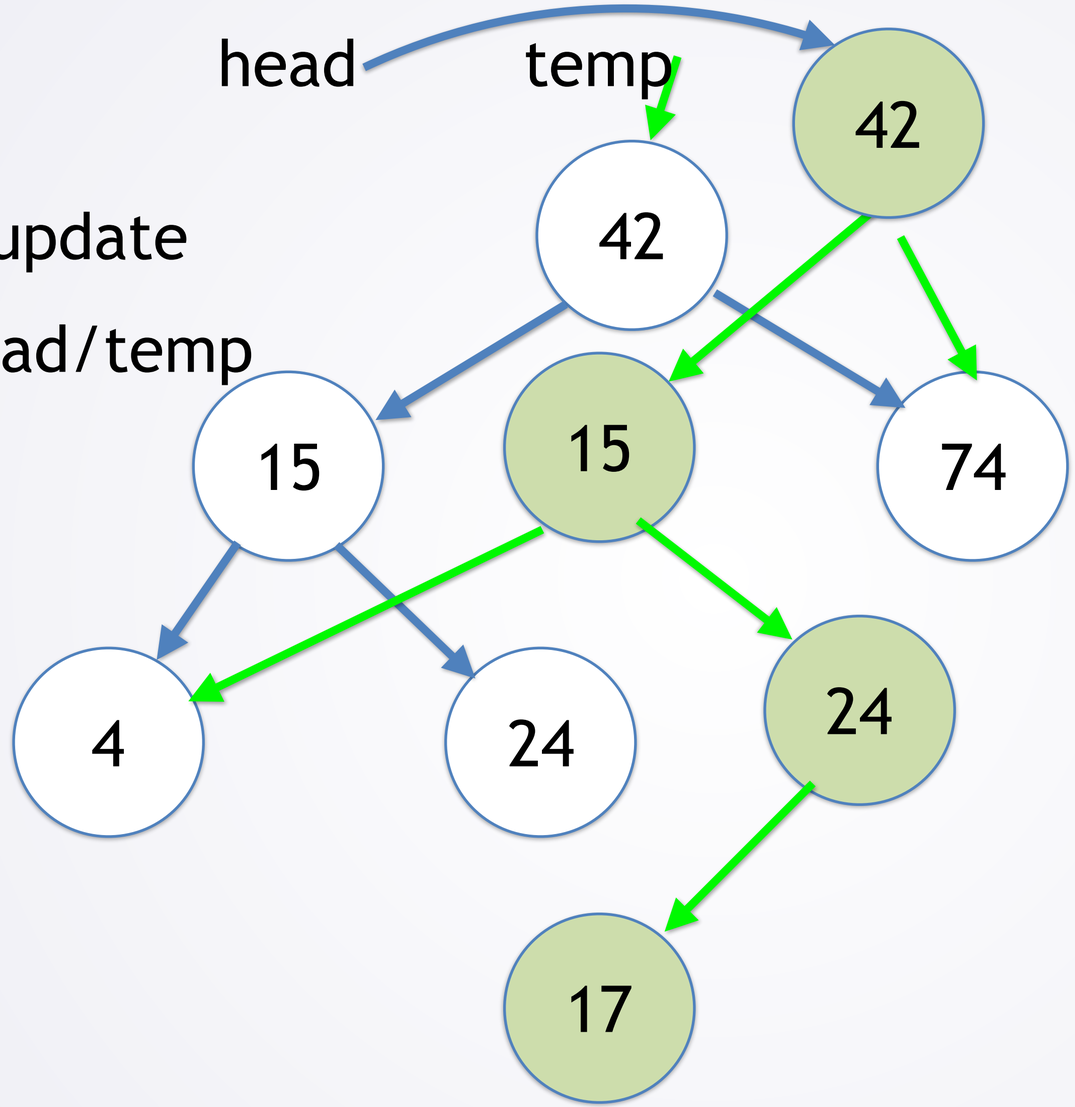
Lock Free BST

Add 17

Option 2:

Do functional update

Atomic CAS head/temp



Lock Free BST

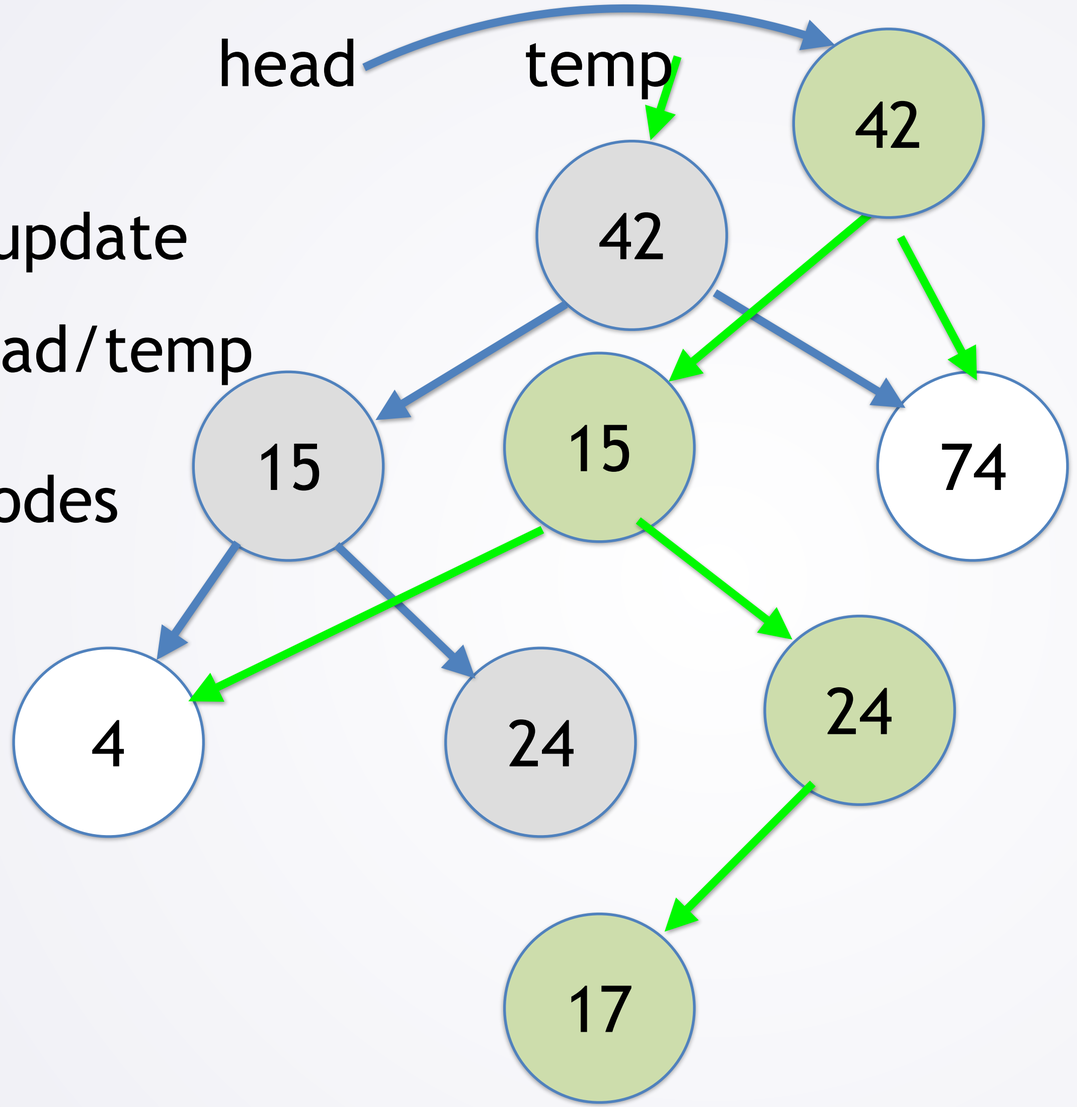
Add 17

Option 2:

Do functional update

Atomic CAS head/temp

Free unused nodes



Lock Free BST

Add 17

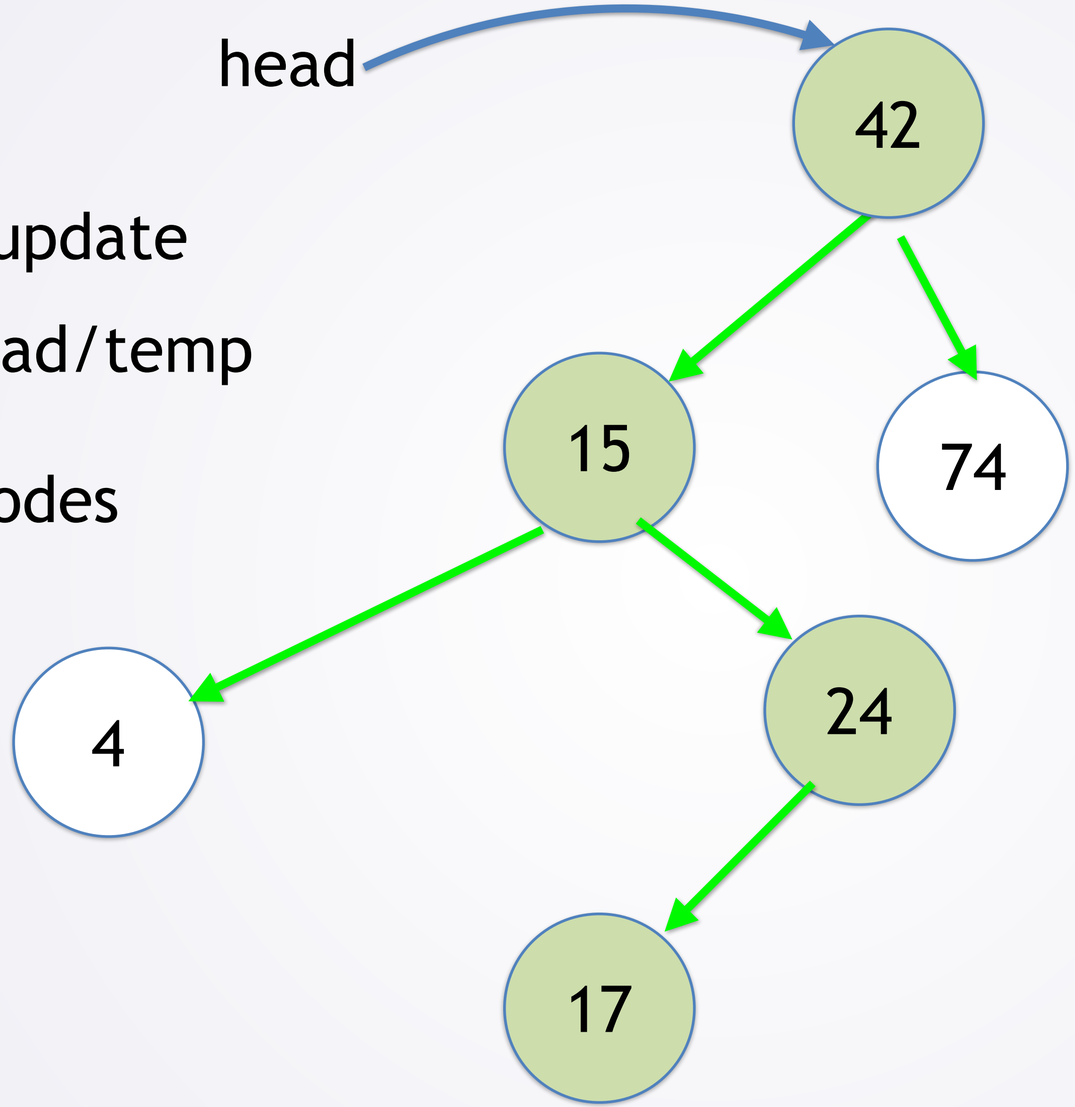
Option 2:

Do functional update

Atomic CAS head/temp

Free unused nodes

Done



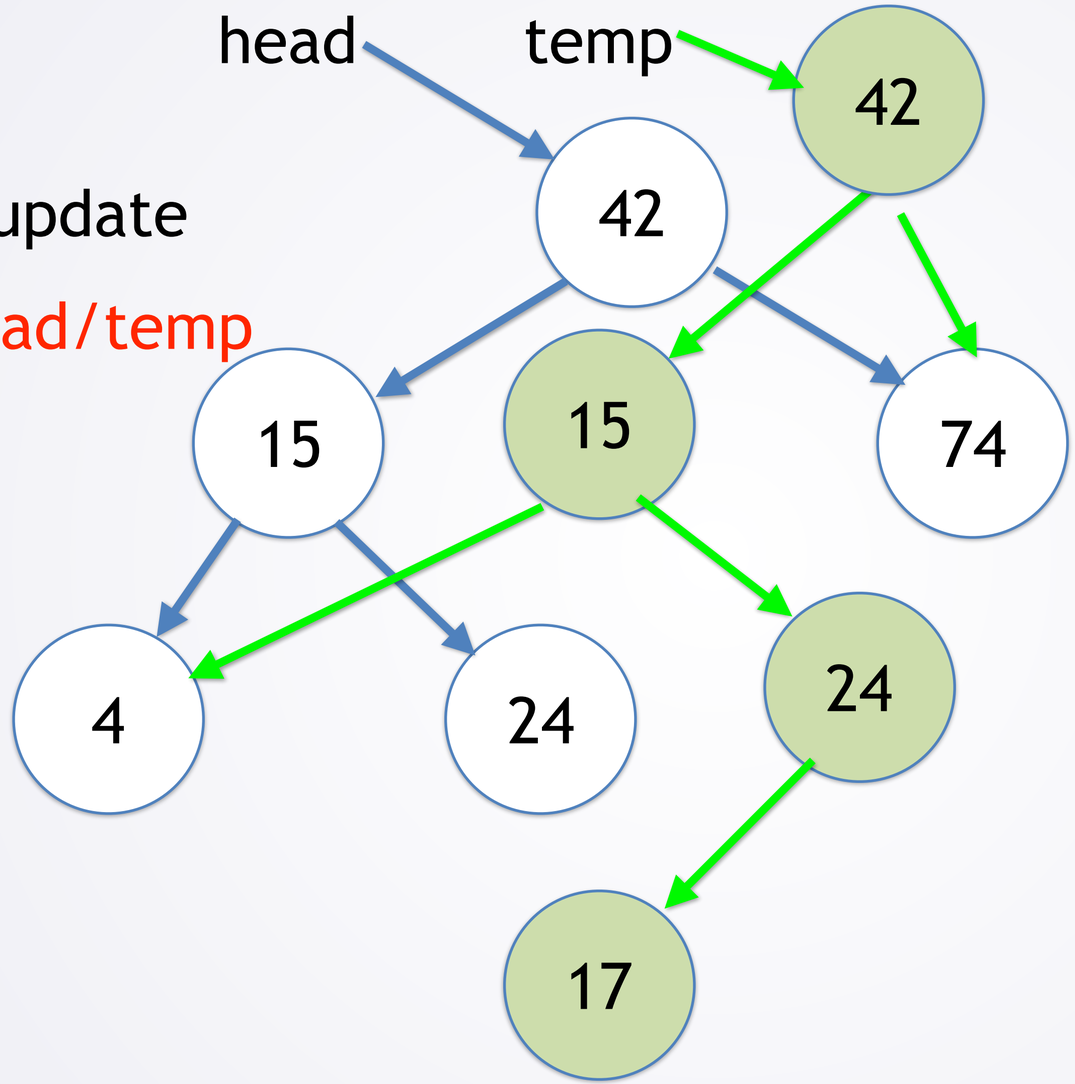
Lock Free BST

Add 17

Option 2:

Do functional update

Atomic CAS head/temp



What if CAS fails?

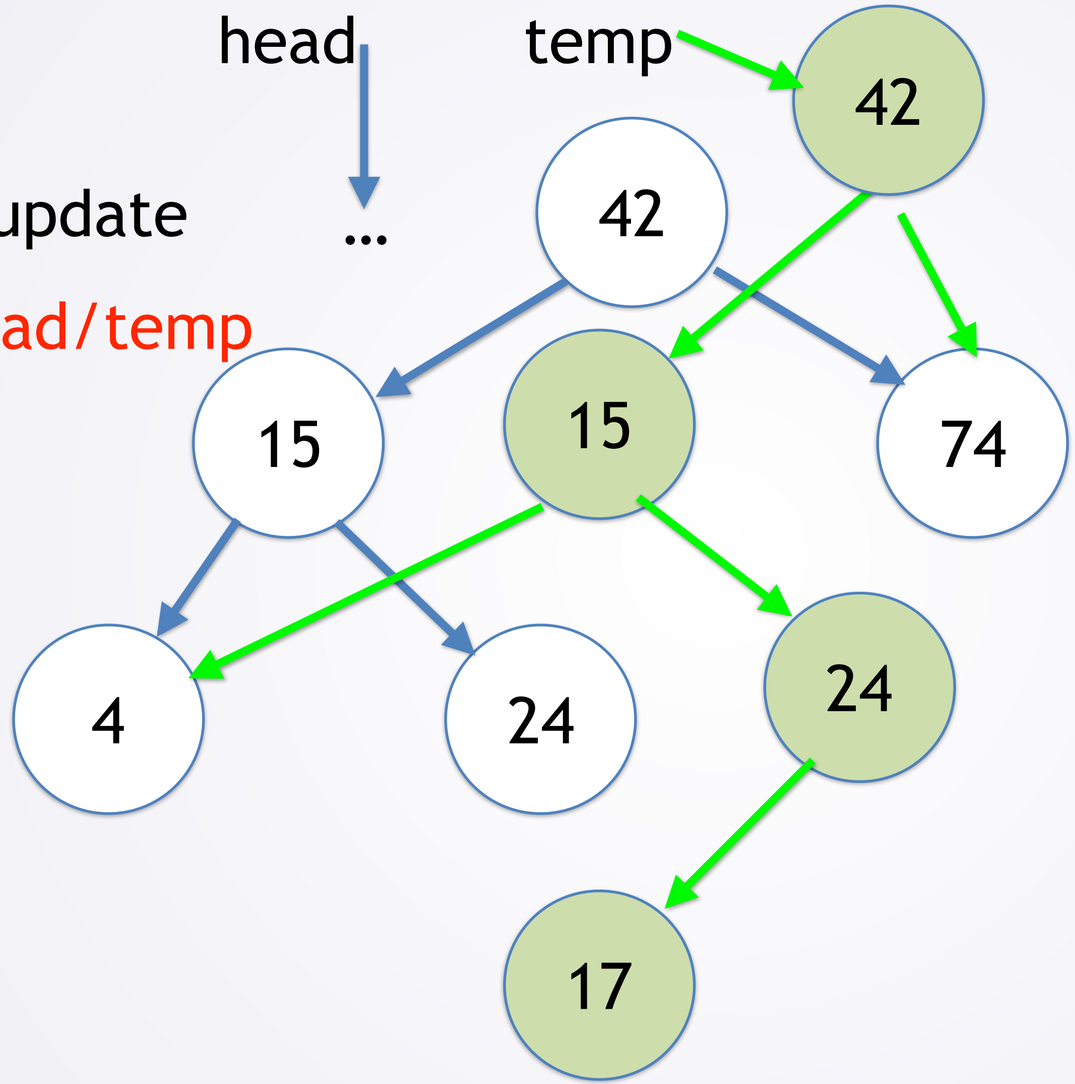
Lock Free BST

Add 17

Option 2:

Do functional update

Atomic CAS head/temp



What if CAS fails?

- Head has changed
- Conflicting writes
- Discard temp tree
- Do add again to head

Lock Free BST

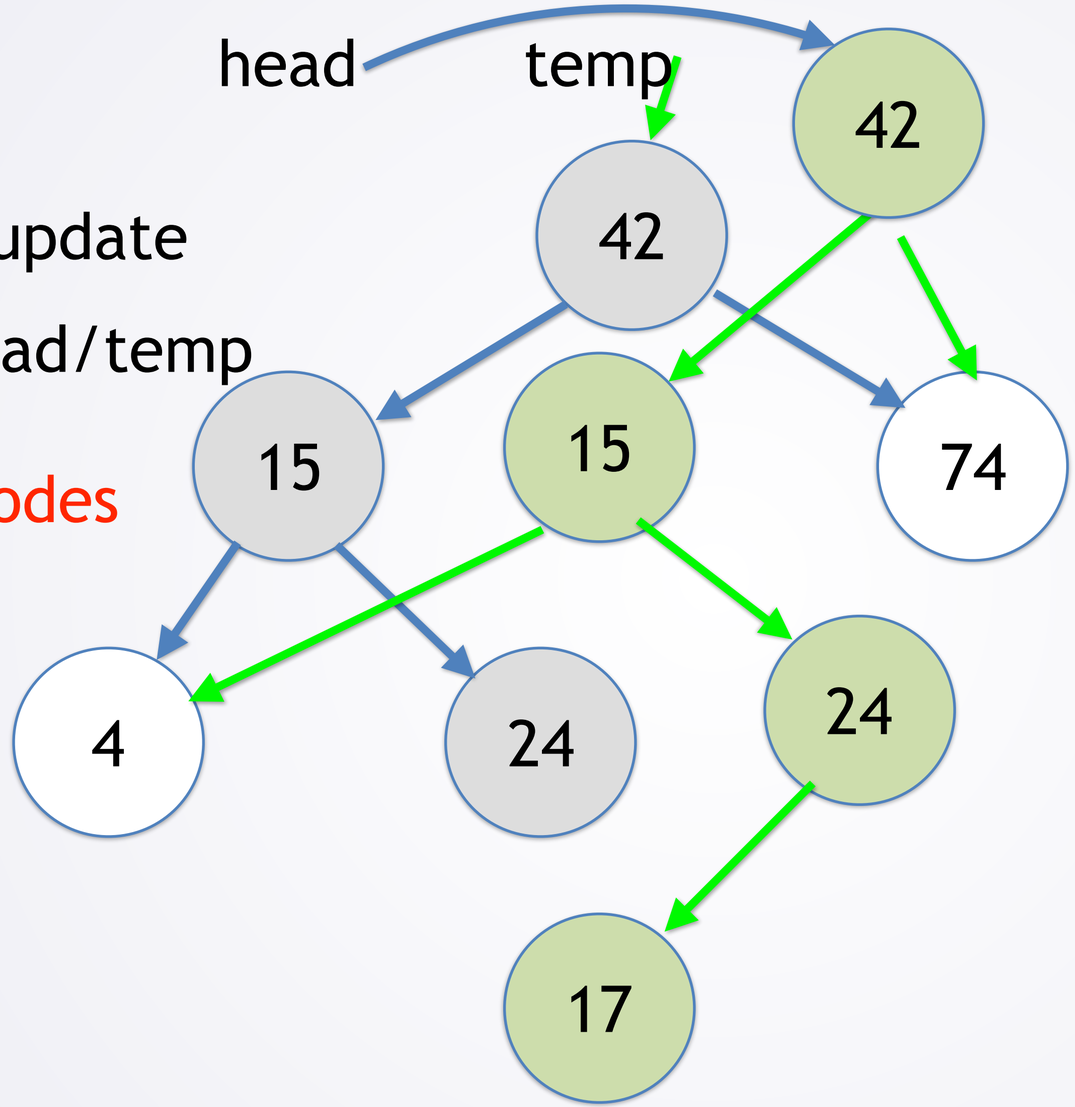
Add 17

Option 2:

Do functional update

Atomic CAS head/temp

Free unused nodes



For nodes seen by other threads just as hard as in LLs..

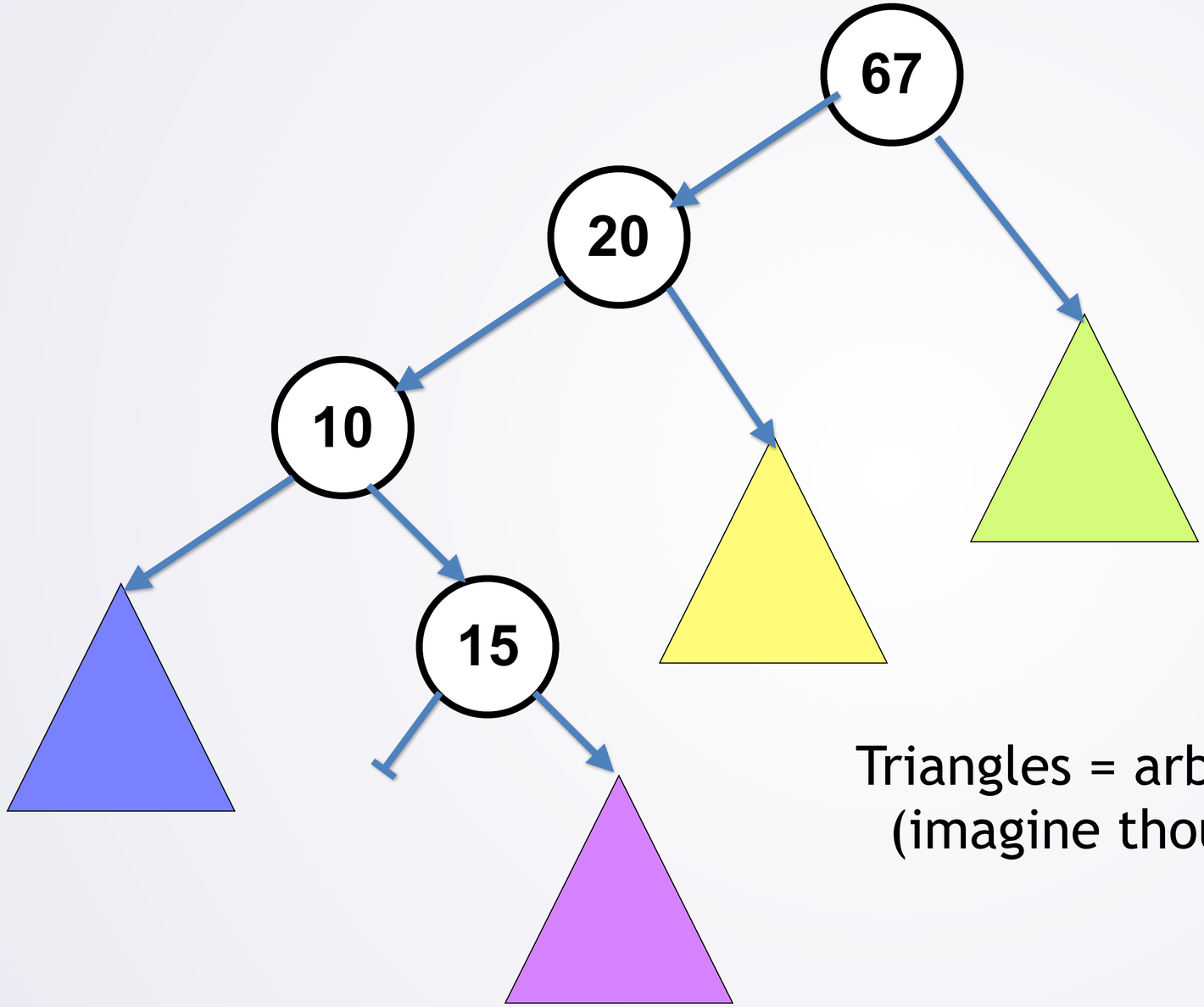
BST: State Re-creation

- What about all those extra nodes we make?
  - Isn't that inefficient?
  - How much state do we have to recreate for an N node tree?

BST: State Duplication

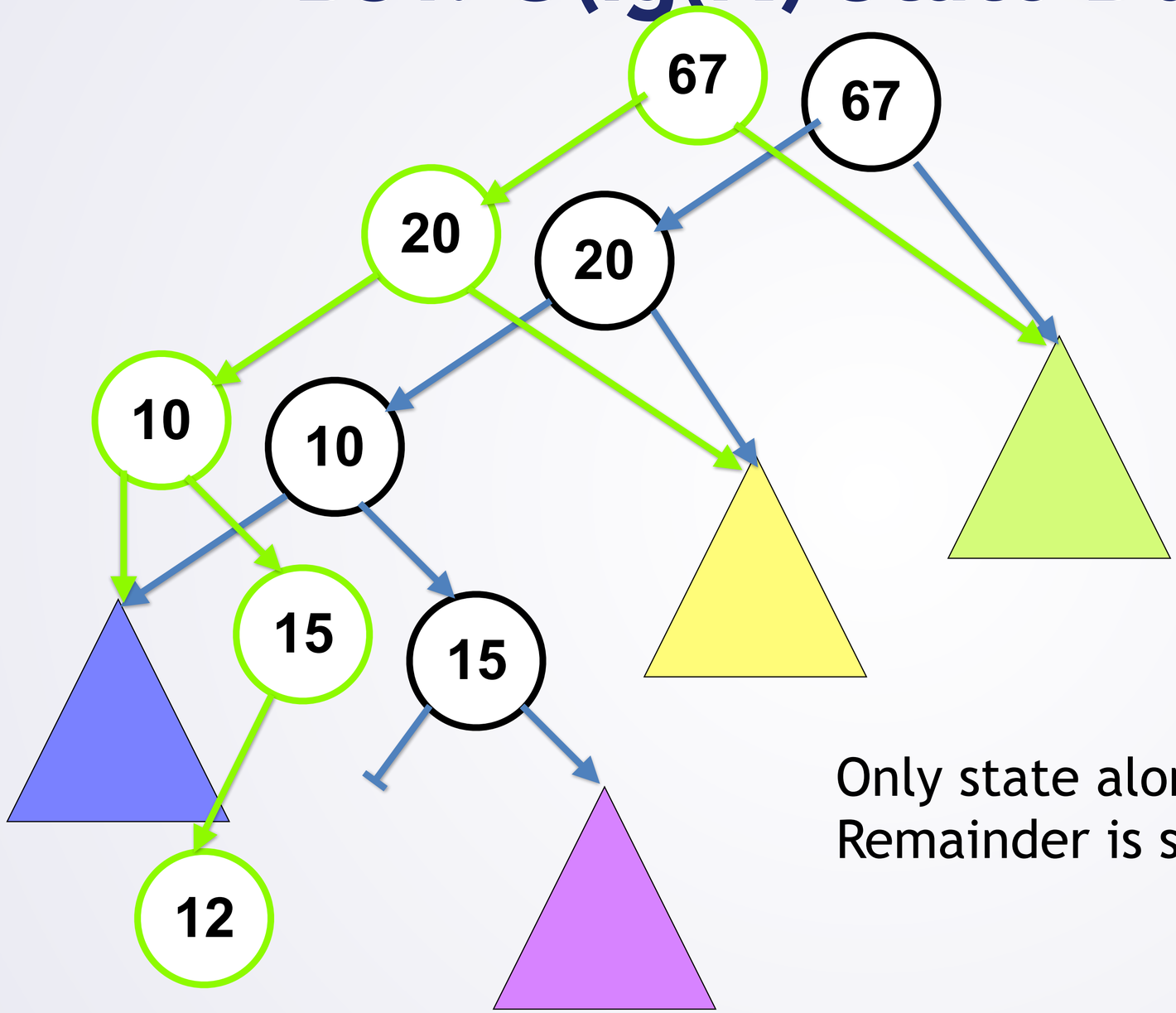
- What about all those extra nodes we make?
  - Isn't that inefficient?
  - How much state do we have to recreate for an N node tree?
    - O(lg(N))
  - Note that a LinkedList would have O(N) state duplication
    - Could use this idea there, just less efficient..

BST: O(lg(N) State Duplication



Triangles = arbitrarily large subtree
(imagine thousands or millions of nodes)

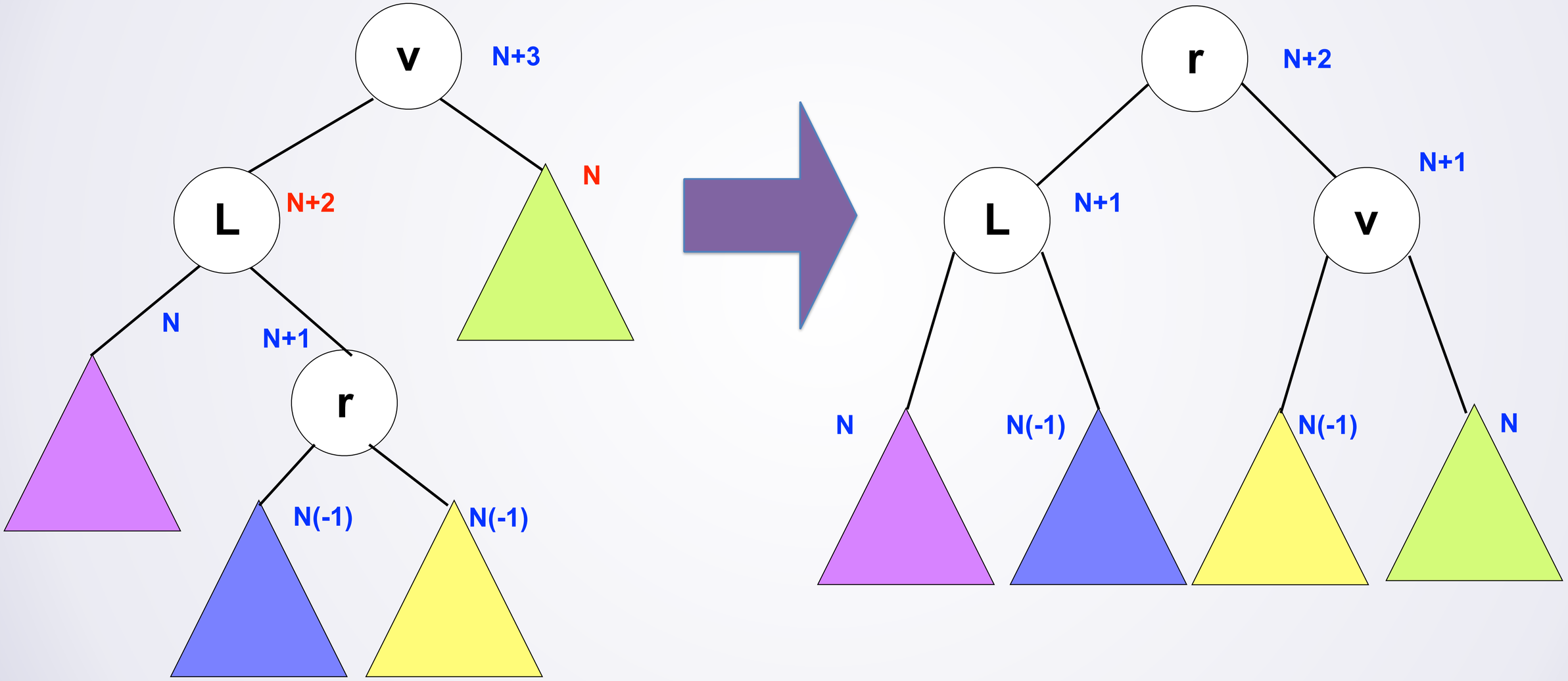
BST: O(lg(N) State Duplication



Only state along actual add path is duplicated
Remainder is shared

Rebalancing?

Rebalancing also only affects addition path.
Can use same idea: share unaffected state



Lock Free Data Structures: Wrap Up

- Upside: scalability
- Downside: complexity
  - Think carefully about races
  - Think carefully about memory ordering
  - Deletions are generally hardest
    - Freeing memory is tricky