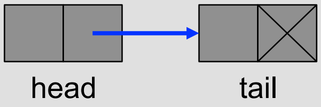
Locking and Data Structures

- How to use Locks
- Example: Concurrent Ordered List
  - Coarse-Grained Locking List
  - Fine-Grained Locking List
  - Optimistic List
  - Lazy List
  - Brief look at vanilla LockFreeList

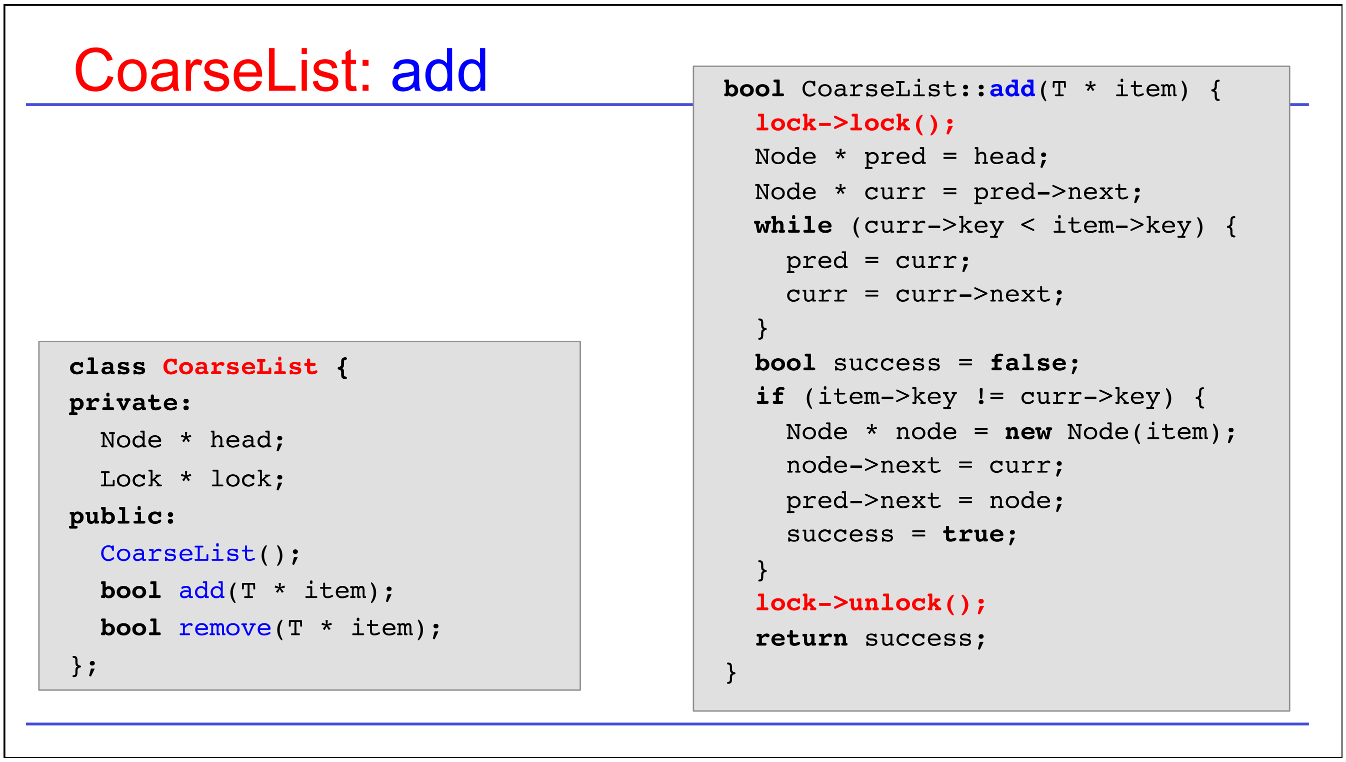
Example: Concurrent Lists

Constructor:

```
CoarseList::CoarseList {
  lock = new Lock();
  head = new Node(MIN_VALUE);
  head->next = new Node(MAX_VALUE);
}
```



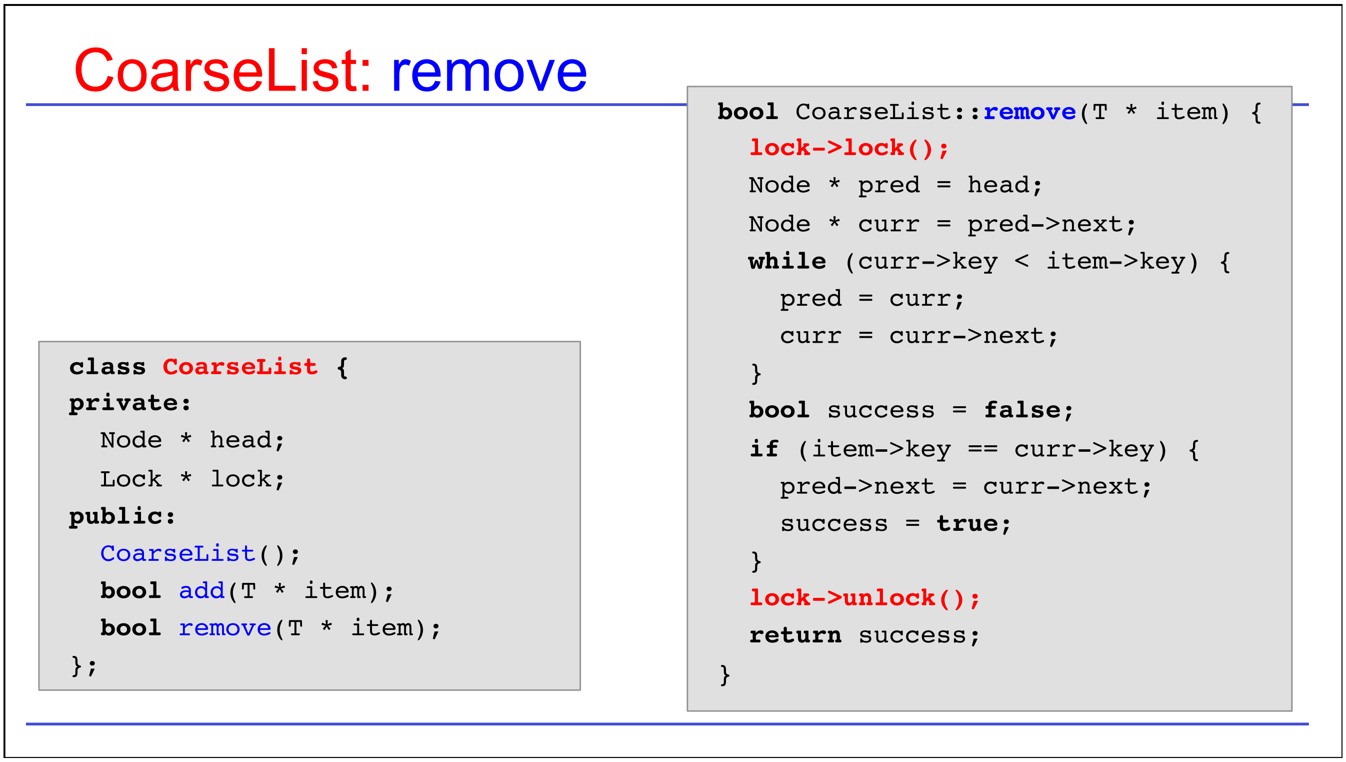
```
class CoarseList {
private:
  Node * head;
  Lock * lock;
public:
  CoarseList();
  bool add(T * item);
  bool remove(T * item);
};
```



CoarseList: add

```
class CoarseList {
private:
  Node * head;
  Lock * lock;
public:
  CoarseList();
  bool add(T * item);
  bool remove(T * item);
};
```

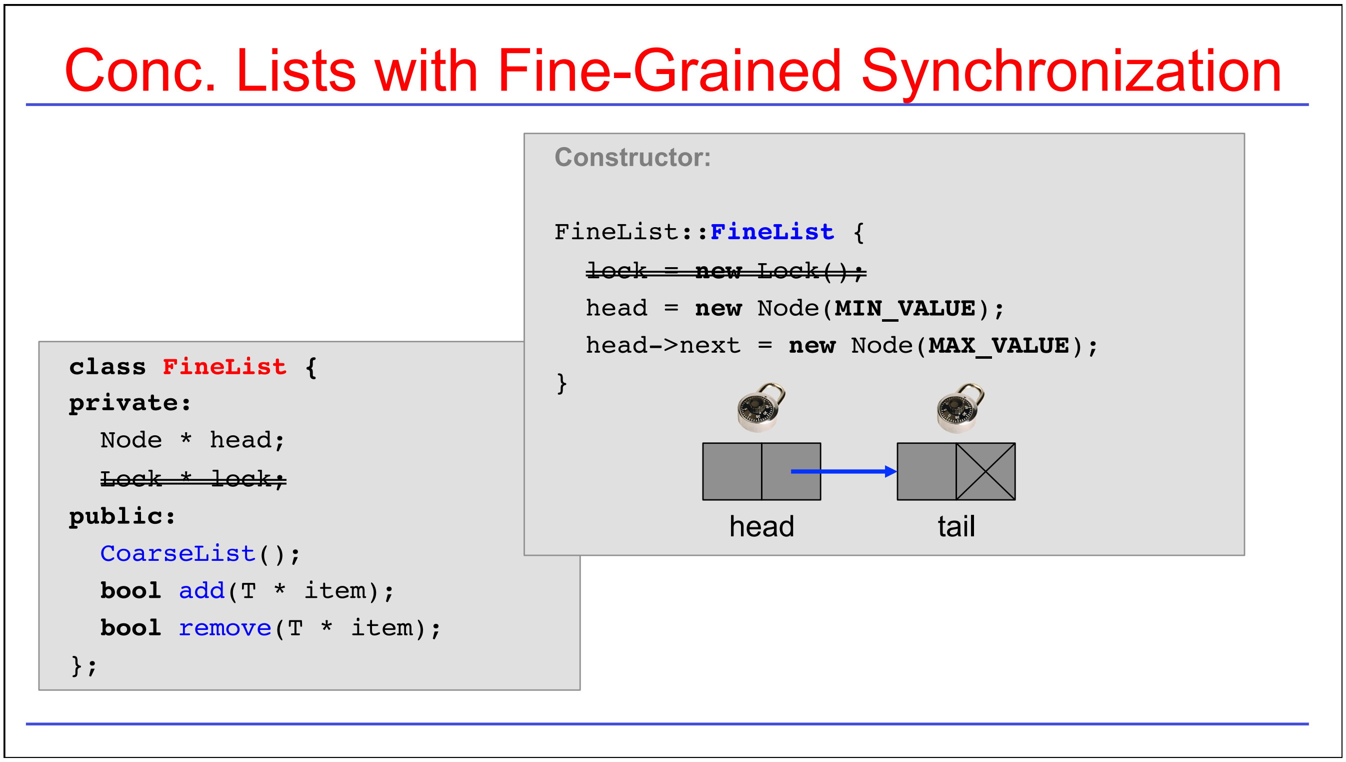
```
bool CoarseList::add(T * item) {
  lock->lock();
  Node * pred = head;
  Node * curr = pred->next;
  while (curr->key < item->key) {
    pred = curr;
    curr = curr->next;
  }
  bool success = false;
  if (item->key != curr->key) {
    Node * node = new Node(item);
    node->next = curr;
    pred->next = node;
    success = true;
  }
  lock->unlock();
  return success;
}
```



CoarseList: remove

```
class CoarseList {
private:
  Node * head;
  Lock * lock;
public:
  CoarseList();
  bool add(T * item);
  bool remove(T * item);
};
```

```
bool CoarseList::remove(T * item) {
  lock->lock();
  Node * pred = head;
  Node * curr = pred->next;
  while (curr->key < item->key) {
    pred = curr;
    curr = curr->next;
  }
  bool success = false;
  if (item->key == curr->key) {
    pred->next = curr->next;
    success = true;
  }
  lock->unlock();
  return success;
}
```

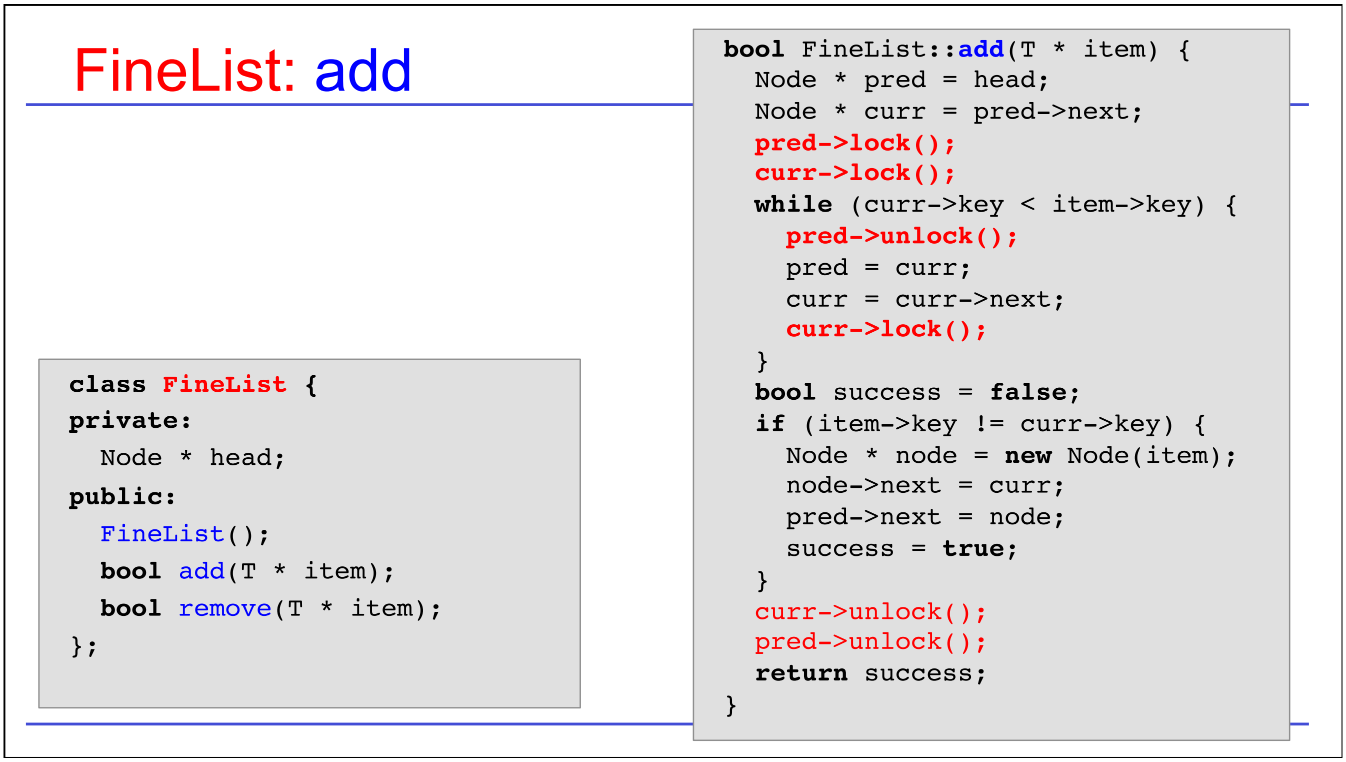


Conc. Lists with Fine-Grained Synchronization

```
class FineList {
private:
  Node * head;
  ~~Lock * lock;~~
public:
  CoarseList();
  bool add(T * item);
  bool remove(T * item);
};
```

Constructor:

```
FineList::FineList {
  ~~lock = new Lock();~~
  head = new Node(MIN_VALUE);
  head->next = new Node(MAX_VALUE);
}
```



FineList: add

```
class FineList {
private:
  Node * head;
public:
  FineList();
  bool add(T * item);
  bool remove(T * item);
};
```

```
bool FineList::add(T * item) {
  Node * pred = head;
  Node * curr = pred->next;
  pred->lock();
  curr->lock();
  while (curr->key < item->key) {
    pred->unlock();
    pred = curr;
    curr = curr->next;
    curr->lock();
  }
  bool success = false;
  if (item->key != curr->key) {
    Node * node = new Node(item);
    node->next = curr;
    pred->next = node;
    success = true;
  }
  curr->unlock();
  pred->unlock();
  return success;
}
```

FineList: remove

```
class FineList {
private:
  Node * head;
public:
  FineList();
  bool add(T * item);
  bool remove(T * item);
};
```

```
bool FineList::remove(T * item) {
  Node * pred = head;
  Node * curr = pred->next;
  pred->lock();
  curr->lock();
  while (curr->key < item->key) {
    pred->unlock();
    pred = curr;
    curr = curr->next;
    curr->lock();
  }
  bool success = false;
  if (item->key == curr->key) {
    pred->next = curr.next;
    success = true;
  }
  curr->unlock();
  pred->unlock();
  return success;
}
```

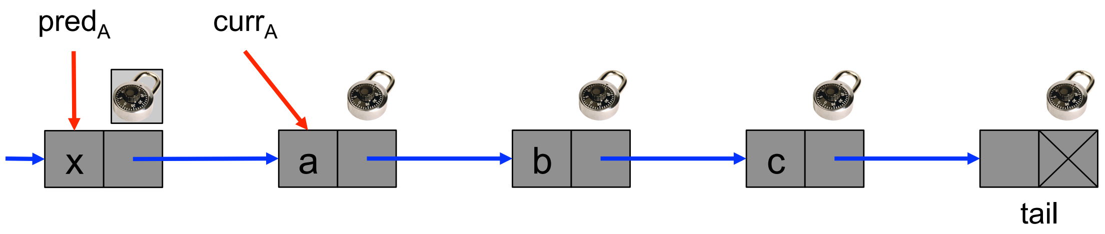
FineList: remove – why 2 Locks?

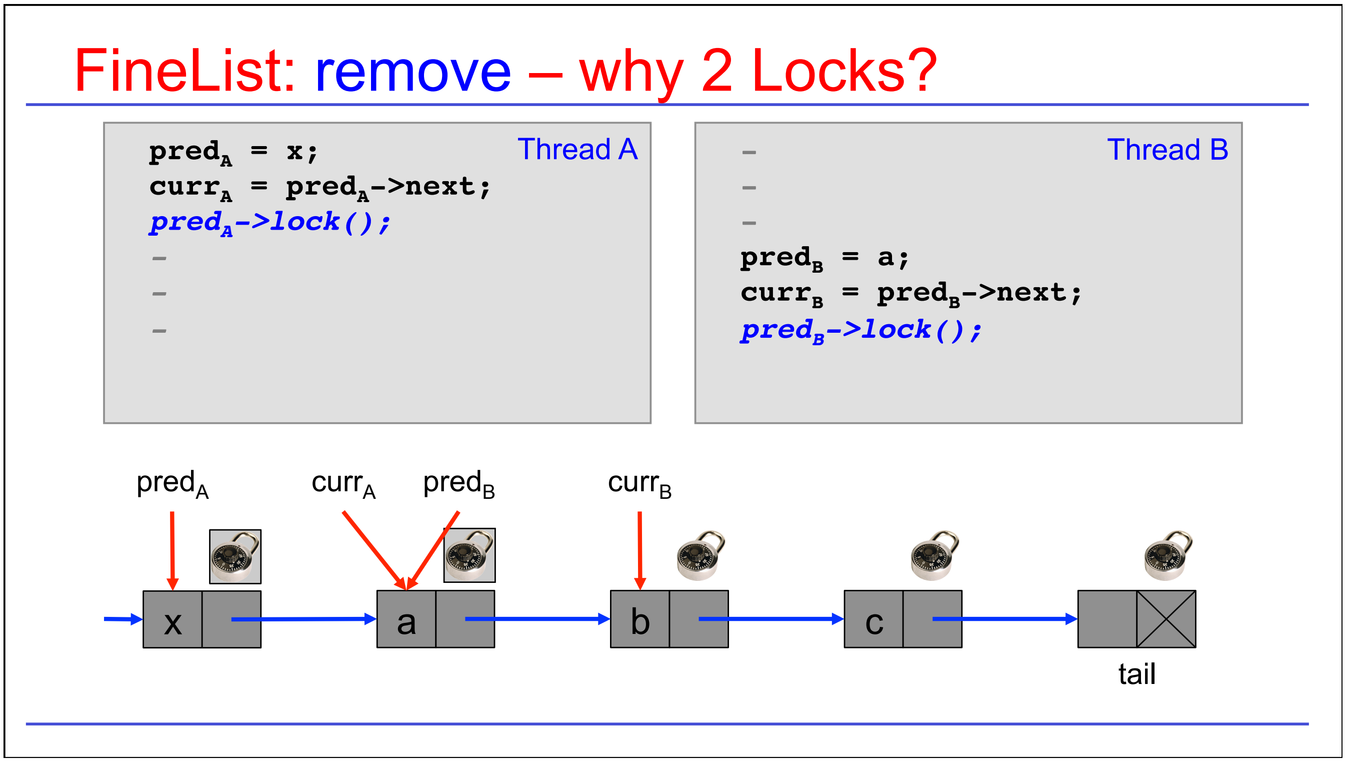
Thread A

```
pred_A = x;
curr_A = pred_A->next;
pred_A->lock();
```

Thread B

```
–
–
–
```





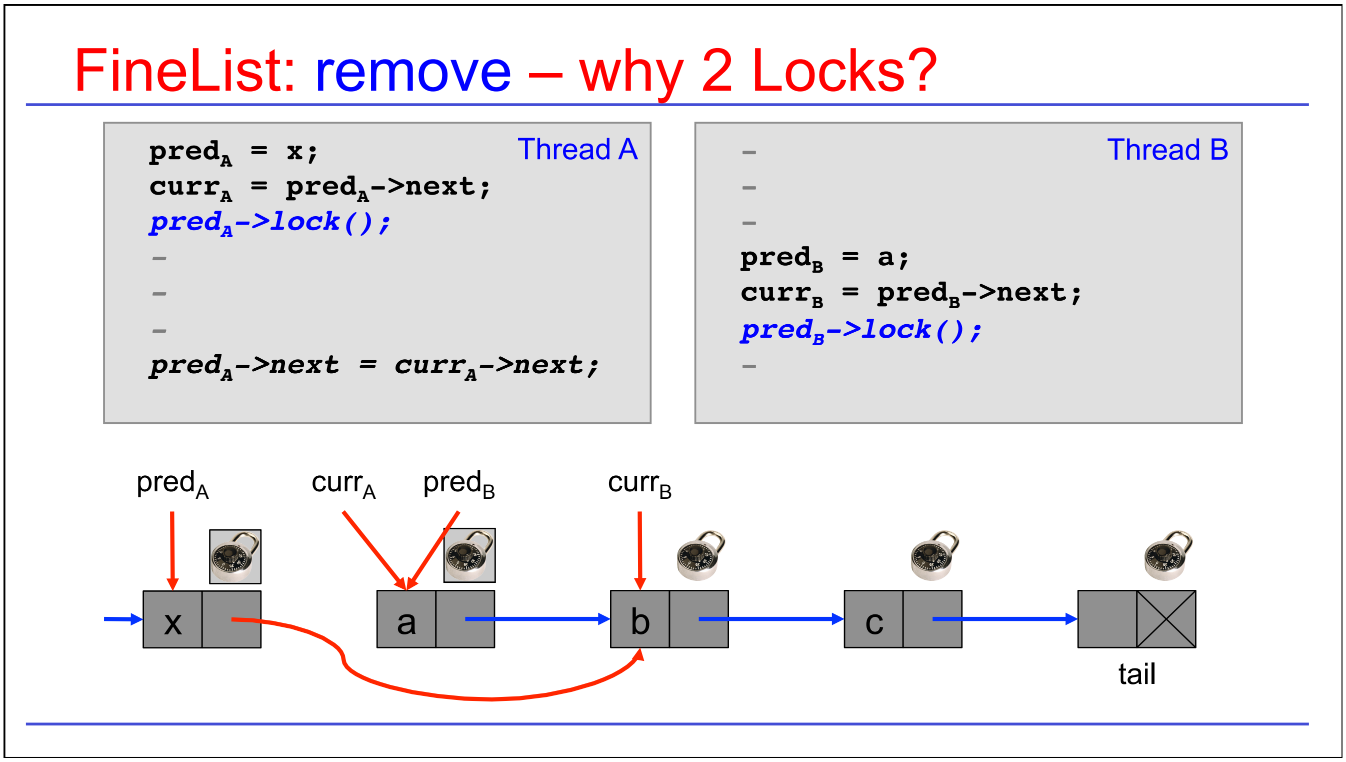
FineList: remove – why 2 Locks?

Thread A

```
pred_A = x;
curr_A = pred_A->next;
pred_A->lock();
–
–
–
```

Thread B

```
–
–
–
pred_B = a;
curr_B = pred_B->next;
pred_B->lock();
```



FineList: remove – why 2 Locks?

Thread A

```
pred_A = x;
curr_A = pred_A->next;
pred_A->lock();
–
–
–
pred_A->next = curr_A->next;
```

Thread B

```
–
–
–
pred_B = a;
curr_B = pred_B->next;
pred_B->lock();
–
```

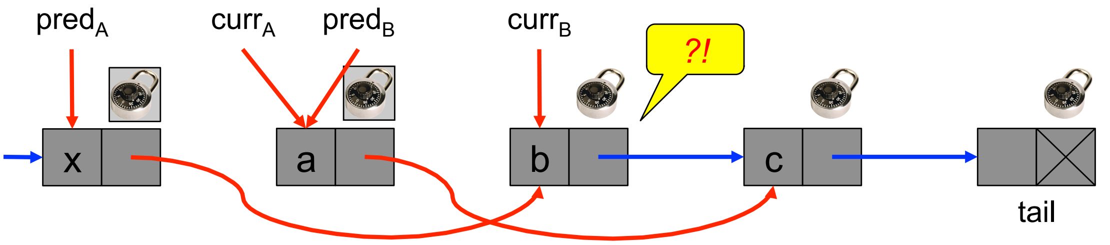
FineList: remove – why 2 Locks?

Thread A

```
predA = x;
currA = predA->next;
predA->lock();
–
–
–
predA->next = currA->next;
–
```

Thread B

```
–
–
–
predB = a;
currB = predB->next;
predB->lock();
–
predB->next = currB->next;
```



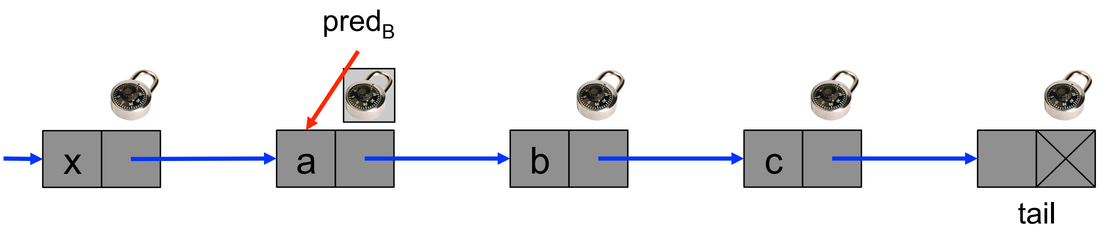
FineList: remove – why 2 Locks?

Thread A

```
–
–
```

Thread B

```
predB = a;
predB->lock();
```



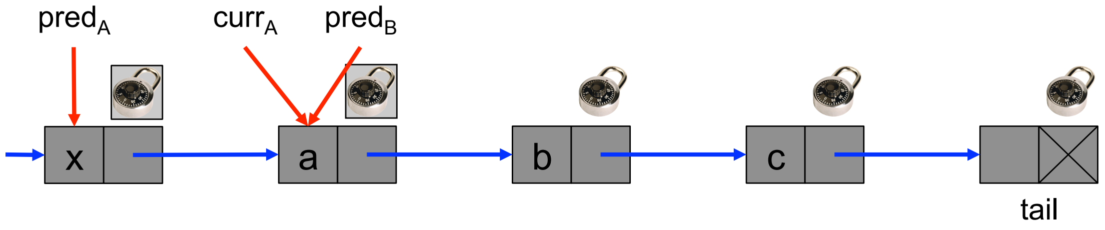
FineList: remove – why 2 Locks?

Thread A

```
–
–
x->lock();
pred_A = x;
curr_A = pred_A->next;
curr_A->lock();
```

Thread B

```
pred_B = a;
pred_B->lock();
–
–
–
–
```



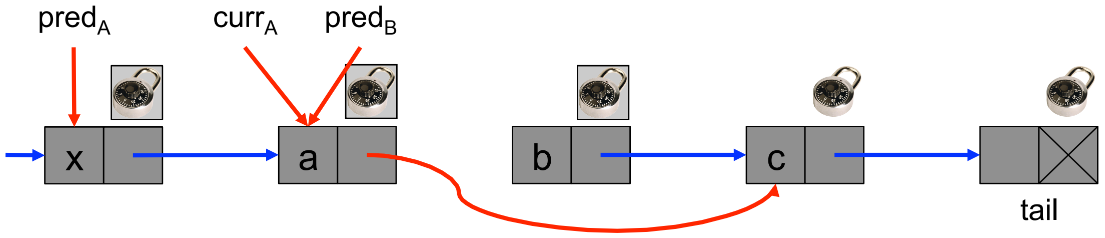
FineList: remove – why 2 Locks?

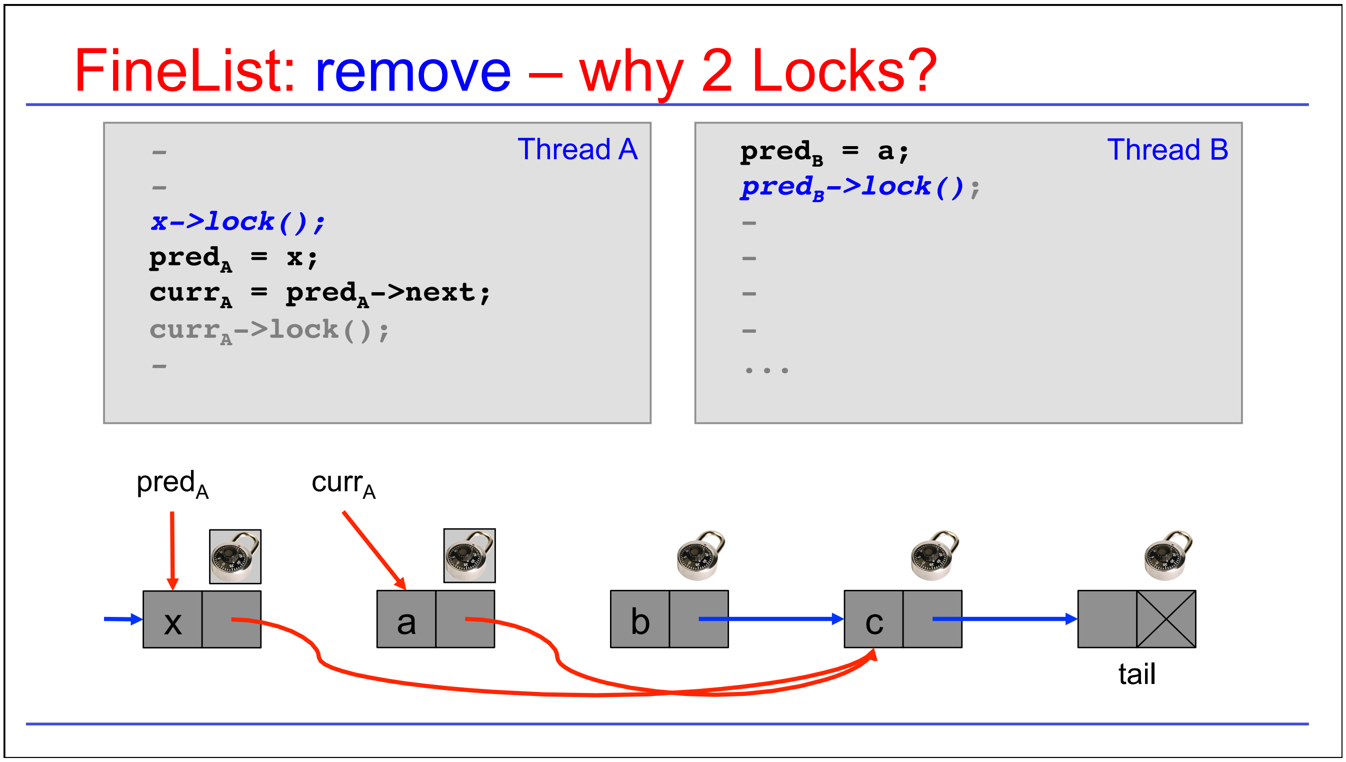
Thread A

```
–
–
x->lock();
pred_A = x;
curr_A = pred_A->next;
curr_A->lock();
–
```

Thread B

```
pred_B = a;
pred_B->lock();
–
–
–
–
...
```





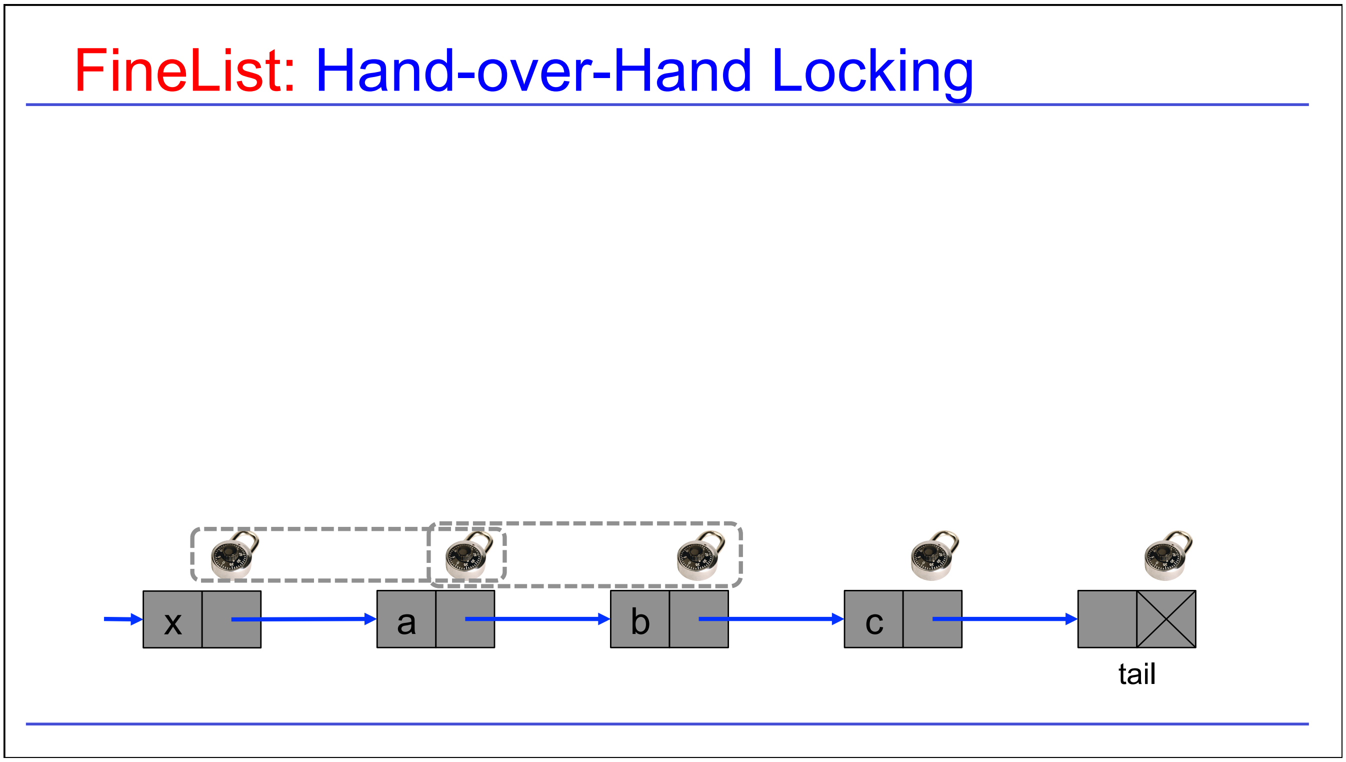
FineList: remove – why 2 Locks?

Thread A

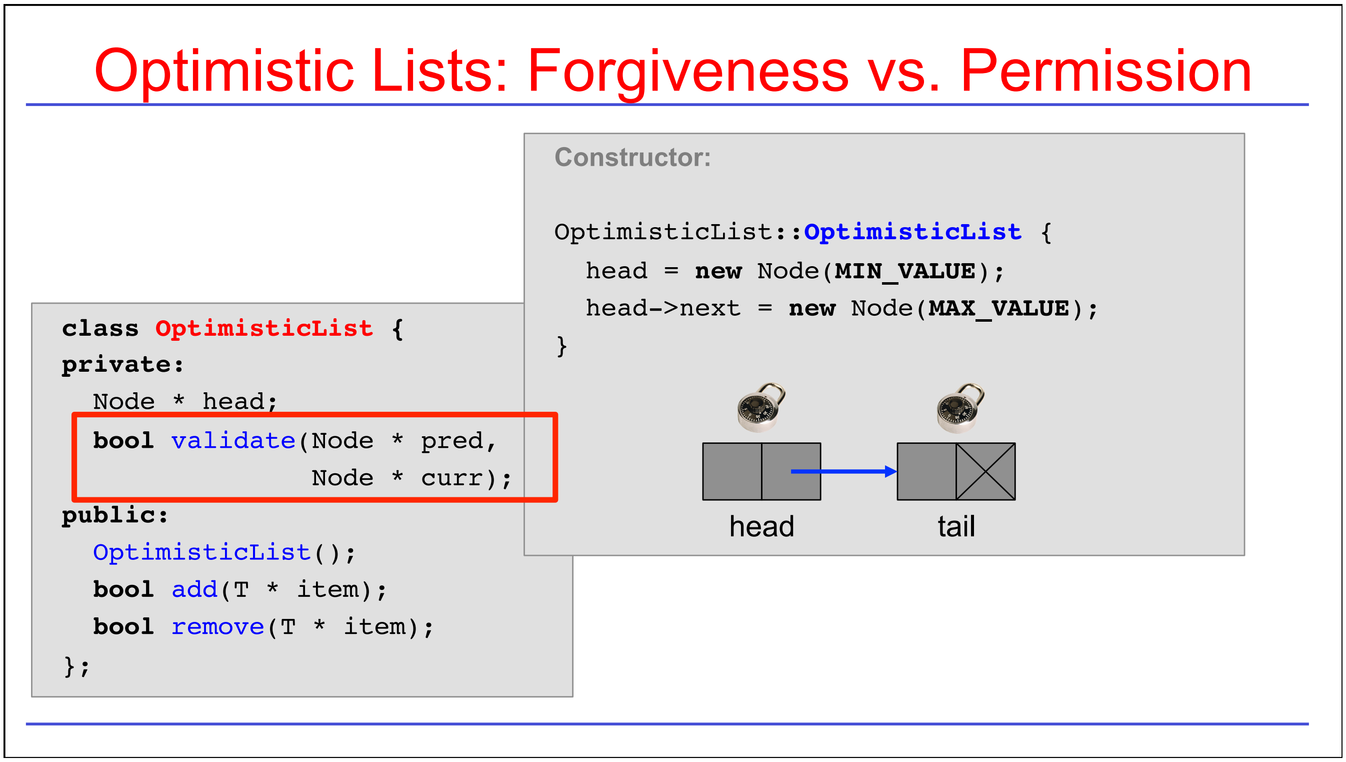
```
–
–
x->lock();
pred_A = x;
curr_A = pred_A->next;
curr_A->lock();
–
```

Thread B

```
pred_B = a;
pred_B->lock();
–
–
–
–
...
```



FineList: Hand-over-Hand Locking

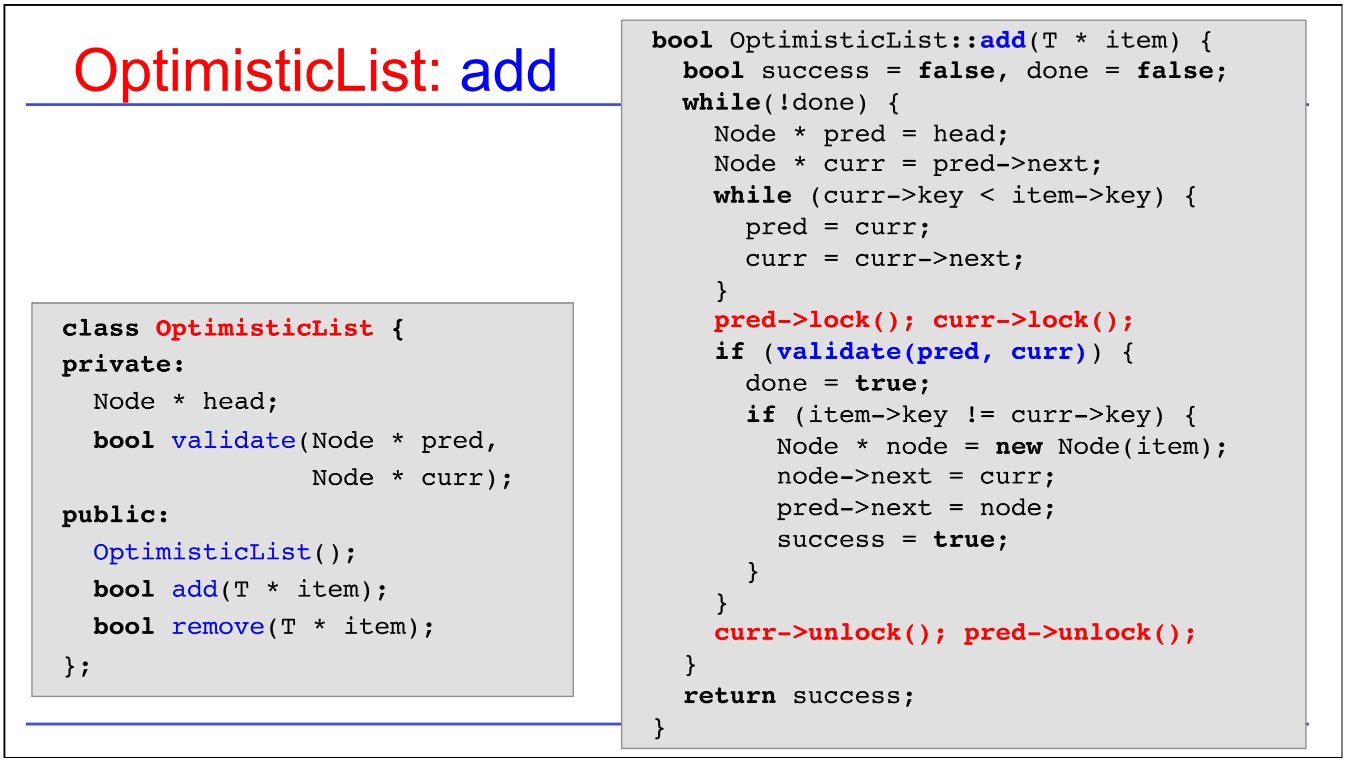


Optimistic Lists: Forgiveness vs. Permission

```
class OptimisticList {
private:
  Node * head;
  bool validate(Node * pred,
                Node * curr);
public:
  OptimisticList();
  bool add(T * item);
  bool remove(T * item);
};
```

Constructor:

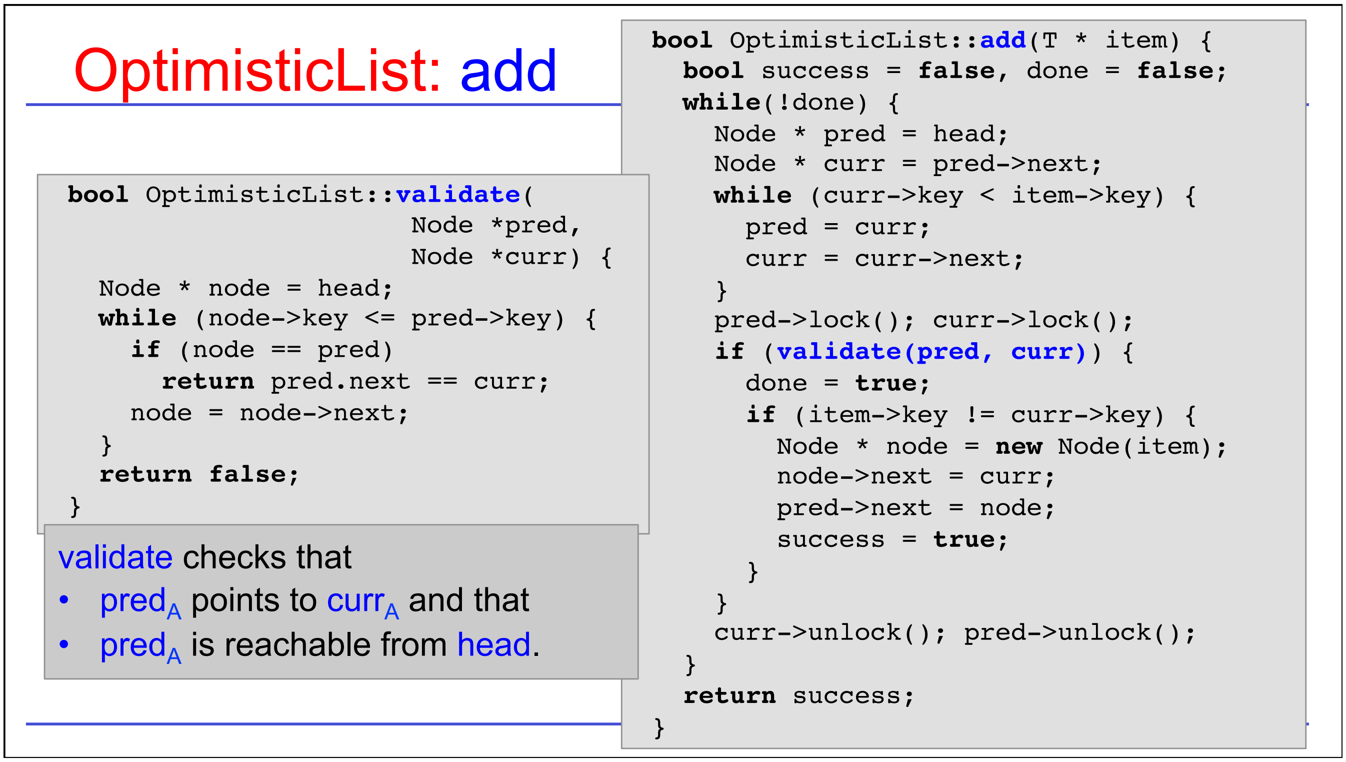
```
OptimisticList::OptimisticList {
  head = new Node(MIN_VALUE);
  head->next = new Node(MAX_VALUE);
}
```



OptimisticList: add

```
class OptimisticList {
private:
  Node * head;
  bool validate(Node * pred,
                Node * curr);
public:
  OptimisticList();
  bool add(T * item);
  bool remove(T * item);
};
```

```
bool OptimisticList::add(T * item) {
  bool success = false, done = false;
  while(!done) {
    Node * pred = head;
    Node * curr = pred->next;
    while (curr->key < item->key) {
      pred = curr;
      curr = curr->next;
    }
    pred->lock(); curr->lock();
    if (validate(pred, curr)) {
      done = true;
      if (item->key != curr->key) {
        Node * node = new Node(item);
        node->next = curr;
        pred->next = node;
        success = true;
      }
    }
    curr->unlock(); pred->unlock();
  }
  return success;
}
```



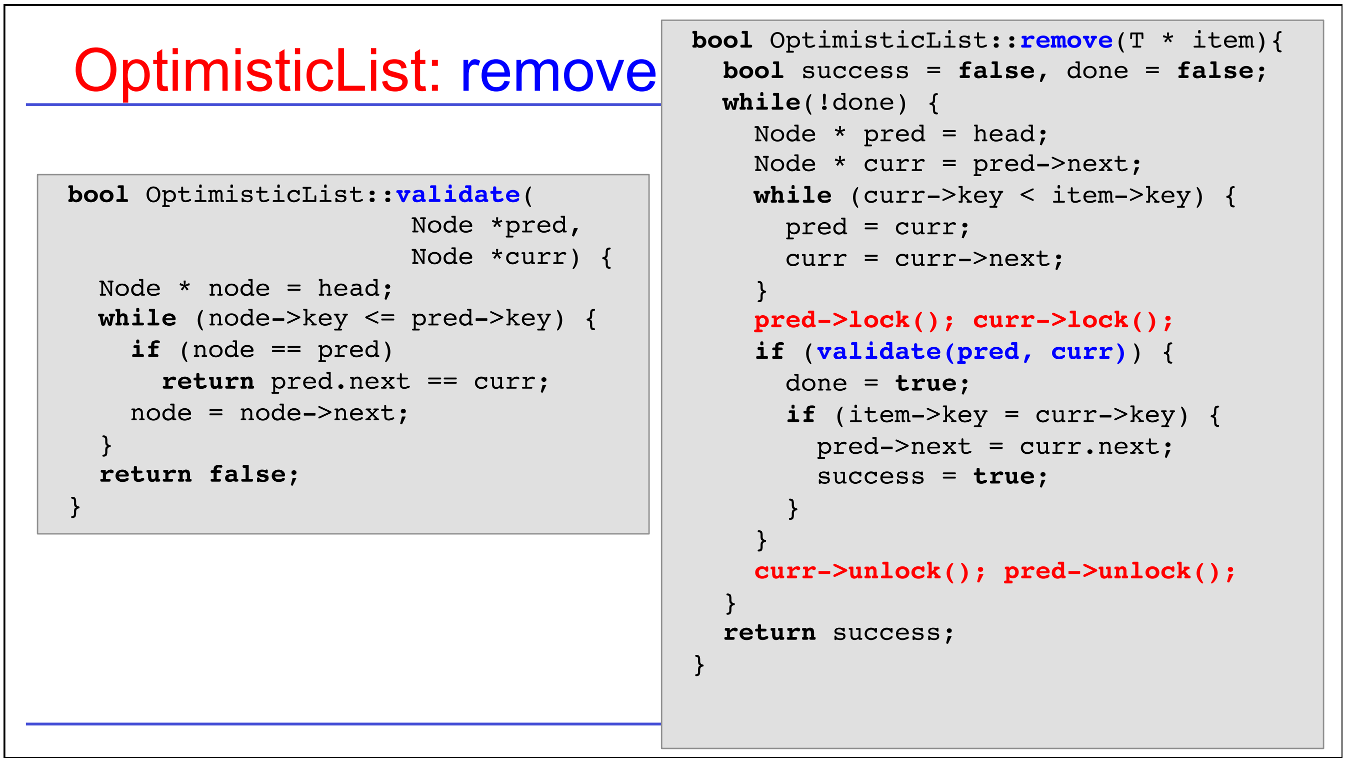
OptimisticList: add

```
bool OptimisticList::validate(
                           Node *pred,
                           Node *curr) {
  Node * node = head;
  while (node->key <= pred->key) {
    if (node == pred)
      return pred.next == curr;
    node = node->next;
  }
  return false;
}
```

validate checks that

- pred$_A$ points to curr$_A$ and that
- pred$_A$ is reachable from head.

```
bool OptimisticList::add(T * item) {
  bool success = false, done = false;
  while(!done) {
    Node * pred = head;
    Node * curr = pred->next;
    while (curr->key < item->key) {
      pred = curr;
      curr = curr->next;
    }
    pred->lock(); curr->lock();
    if (validate(pred, curr)) {
      done = true;
      if (item->key != curr->key) {
        Node * node = new Node(item);
        node->next = curr;
        pred->next = node;
        success = true;
      }
    }
    curr->unlock(); pred->unlock();
  }
  return success;
}
```



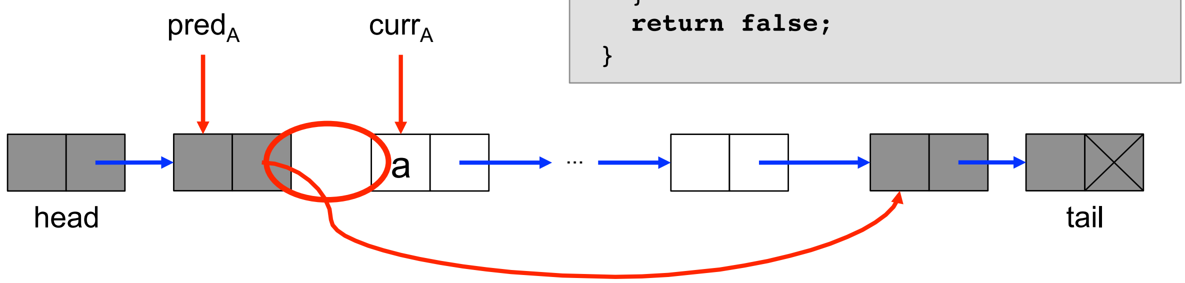
OptimisticList: remove

```
bool OptimisticList::validate(
                           Node *pred,
                           Node *curr) {
  Node * node = head;
  while (node->key <= pred->key) {
    if (node == pred)
      return pred.next == curr;
    node = node->next;
  }
  return false;
}
```

```
bool OptimisticList::remove(T * item){
  bool success = false, done = false;
  while(!done) {
    Node * pred = head;
    Node * curr = pred->next;
    while (curr->key < item->key) {
      pred = curr;
      curr = curr->next;
    }
    pred->lock(); curr->lock();
    if (validate(pred, curr)) {
      done = true;
      if (item->key = curr->key) {
        pred->next = curr.next;
        success = true;
      }
    }
    curr->unlock(); pred->unlock();
  }
  return success;
}
```

OptimisticList: Why Validation?

```
bool OptimisticList::validate(
                           Node *pred,
                           Node *curr) {
  Node * node = head;
  while (node->key <= pred->key) {
    if (node == pred)
      return pred.next == curr;
    node = node->next;
  }
  return false;
}
```

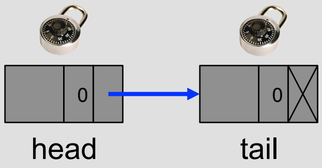


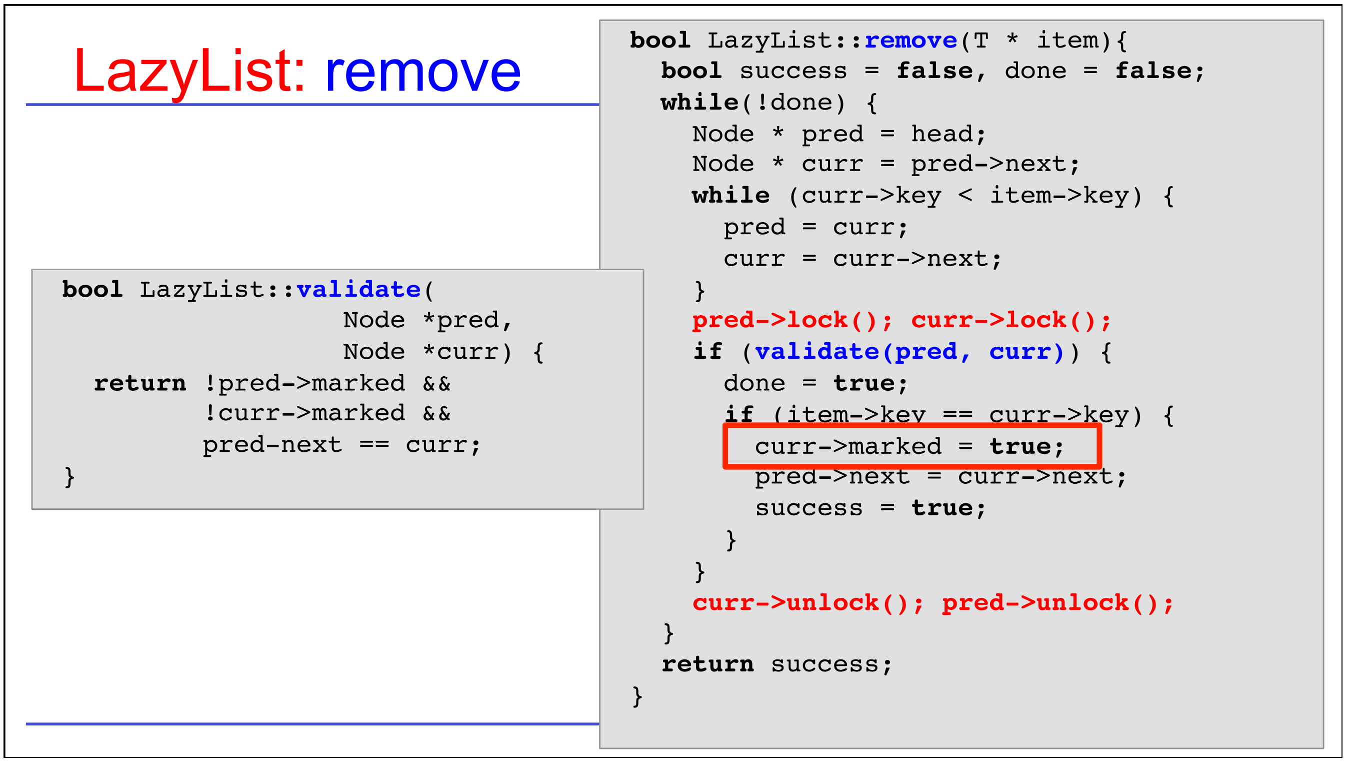
Lazy Lists: Mark Nodes

```
class LazyList {
private:
  Node * head;
  bool validate(Node * pred,
                Node * curr);
public:
  LazyList();
  bool add(T * item);
  bool remove(T * item);
};
```

Constructor:

```
LazyList::LazyList {
  head = new Node(MIN_VALUE);
  head->next = new Node(MAX_VALUE);
}
```

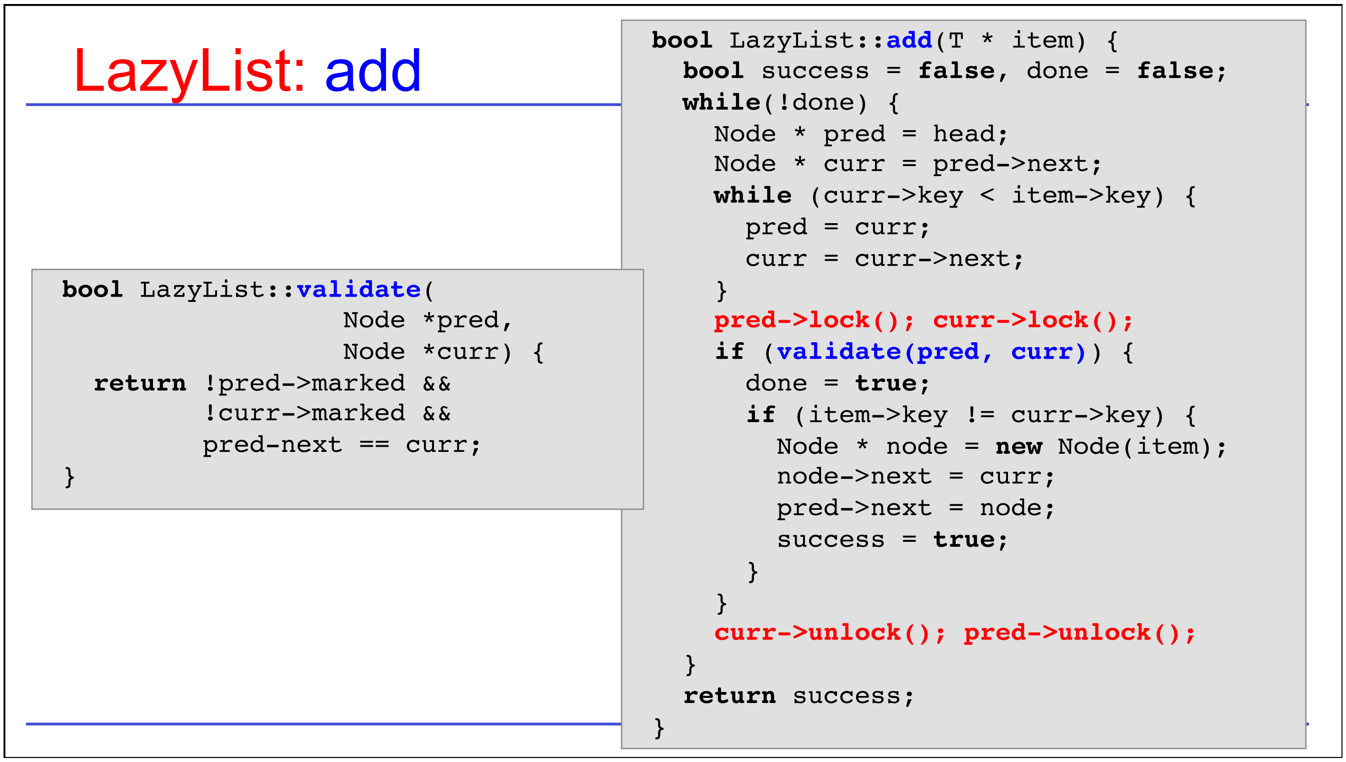




LazyList: remove

```
bool LazyList::validate(
                    Node *pred,
                    Node *curr) {
  return !pred->marked &&
         !curr->marked &&
         pred-next == curr;
}
```

```
bool LazyList::remove(T * item){
  bool success = false, done = false;
  while(!done) {
    Node * pred = head;
    Node * curr = pred->next;
    while (curr->key < item->key) {
      pred = curr;
      curr = curr->next;
    }
    pred->lock(); curr->lock();
    if (validate(pred, curr)) {
      done = true;
      if (item->key == curr->key) {
        curr->marked = true;
        pred->next = curr->next;
        success = true;
      }
    }
    curr->unlock(); pred->unlock();
  }
  return success;
}
```



LazyList: add

```
bool LazyList::validate(
                    Node *pred,
                    Node *curr) {
  return !pred->marked &&
         !curr->marked &&
         pred-next == curr;
}
```

```
bool LazyList::add(T * item) {
  bool success = false, done = false;
  while(!done) {
    Node * pred = head;
    Node * curr = pred->next;
    while (curr->key < item->key) {
      pred = curr;
      curr = curr->next;
    }
    pred->lock(); curr->lock();
    if (validate(pred, curr)) {
      done = true;
      if (item->key != curr->key) {
        Node * node = new Node(item);
        node->next = curr;
        pred->next = node;
        success = true;
      }
    }
    curr->unlock(); pred->unlock();
  }
  return success;
}
```

LazyList: contains

```
bool LazyList::contains(T * item){
  Node * curr = head;

  while (curr->key < item->key)
    curr = curr->next;

  return curr->key == item->key && !curr->marked;
}
```

LazyList: add

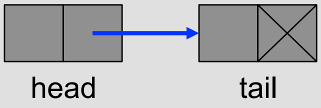
```
bool LazyList::add(T * item) {
  bool success = false, done = false;
  while(!done) {
    Node * pred = head;
    Node * curr = pred->next;
    while (curr->key < item->key) {
      pred = curr;
      curr = curr->next;
    }
    pred->lock(); curr->lock();
    if (validate(pred, curr)) {
      done = true;
      if (item->key != curr->key) {
        Node * node = new Node(item);
        node->next = curr;
        pred->next = node;
        success = true;
      }
    }
    curr->unlock(); pred->unlock();
  }
  return success;
}
```

Lock-Free Lists: Vanilla Attempt

```
class LockFreeList {
private:
  Node * head;
public:
  LockFreeList();
  bool add(T * item);
  ~~bool remove(T * item);~~
};
```

Constructor:

```
LockFreeList::LockFreeList {
  head = new Node(MIN_VALUE);
  head->next = new Node(MAX_VALUE);
}
```

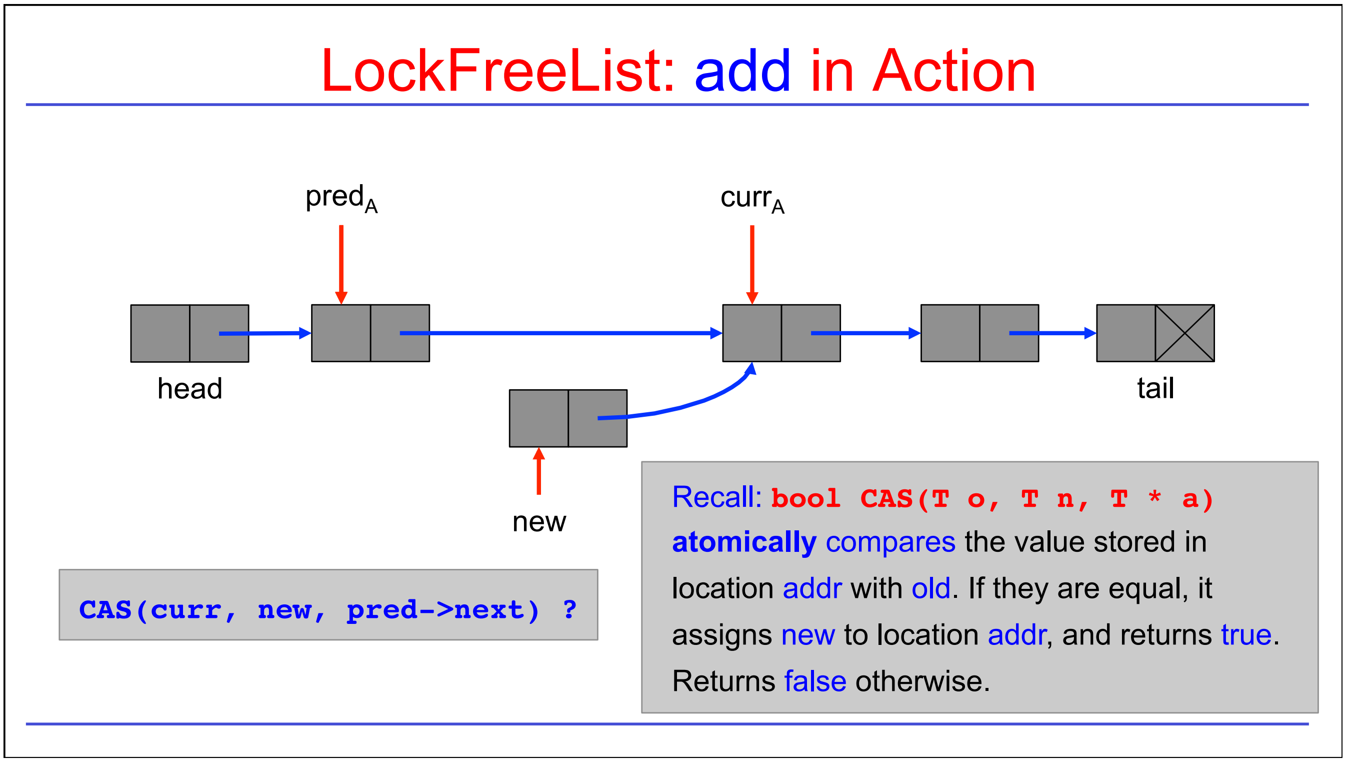


LockFreeList: add

Recall: **bool CAS(T o, T n, T * a)** **atomically** compares the value stored in location addr with old. If they are equal, it assigns new to location addr, and returns true. Returns false otherwise.

```
class LockFreeList {
private:
  Node * head;
public:
  LockFreeList();
  bool add(T * item);
  ~~bool remove(T * item);~~
};
```

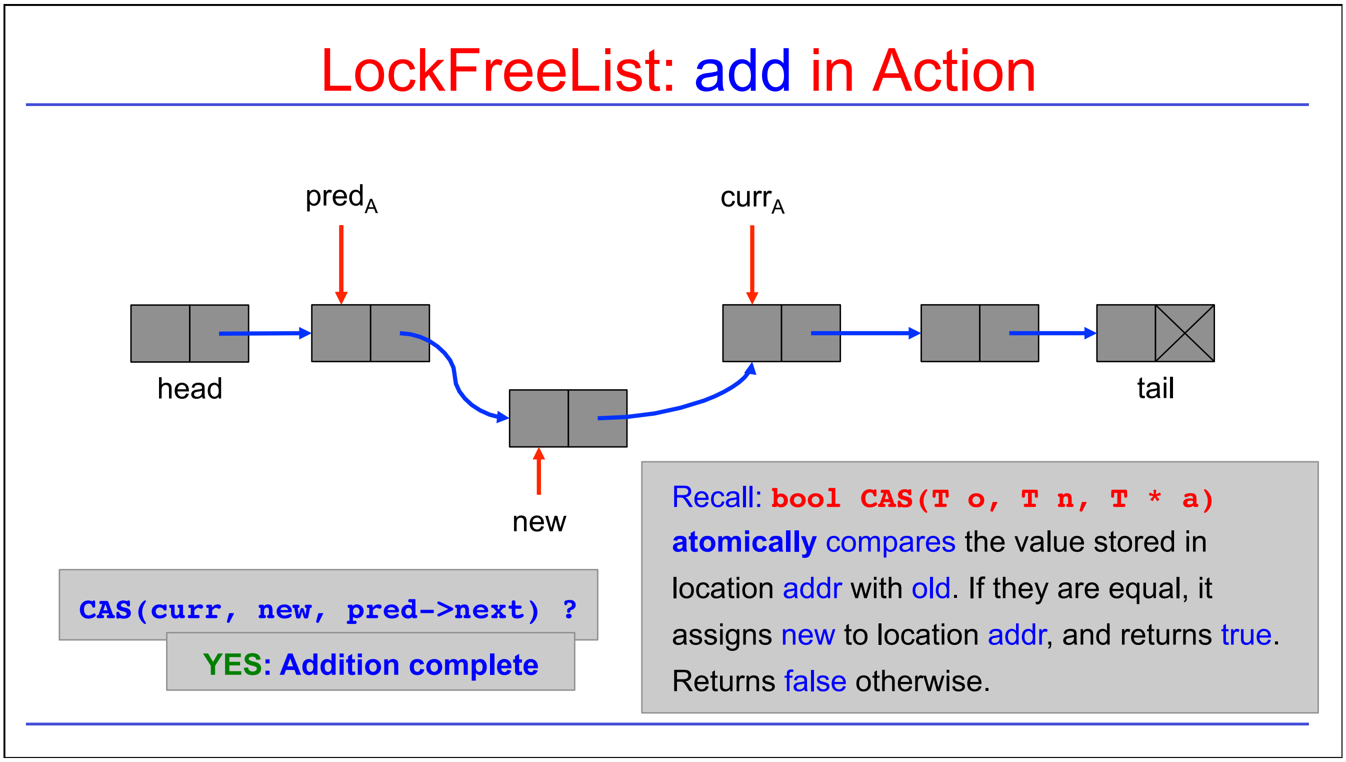
```
bool LockFreeList::add(T * item) {
  while(true) {
           e * pred = head;
           e * curr = pred->next;

           le (curr->key < item->key) {
           red = curr;
      curr = curr->next;
    }

    if (item->key == curr->key) {
      return false;
    }
    else {
      Node * new_node = new Node(item);
      new_node->next = curr;
      if (CAS(curr, new_node, pred->next))
        return true;
    }
  }
}
```



LockFreeList: add in Action

`CAS(curr, new, pred->next) ?`

Recall: `bool CAS(T o, T n, T * a)` **atomically** compares the value stored in location addr with old. If they are equal, it assigns new to location addr, and returns true. Returns false otherwise.

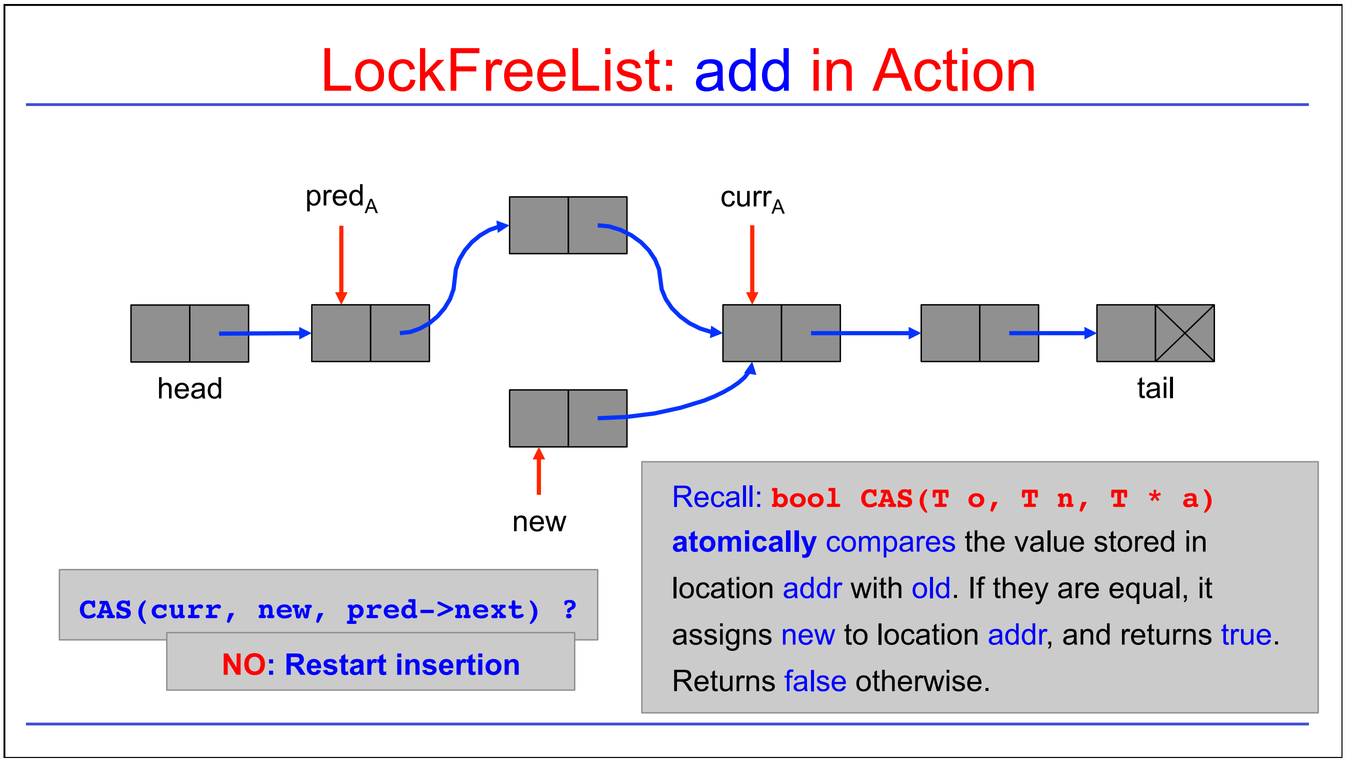


LockFreeList: add in Action

`CAS(curr, new, pred->next) ?`

YES: Addition complete

Recall: `bool CAS(T o, T n, T * a)` **atomically** compares the value stored in location addr with old. If they are equal, it assigns new to location addr, and returns true. Returns false otherwise.

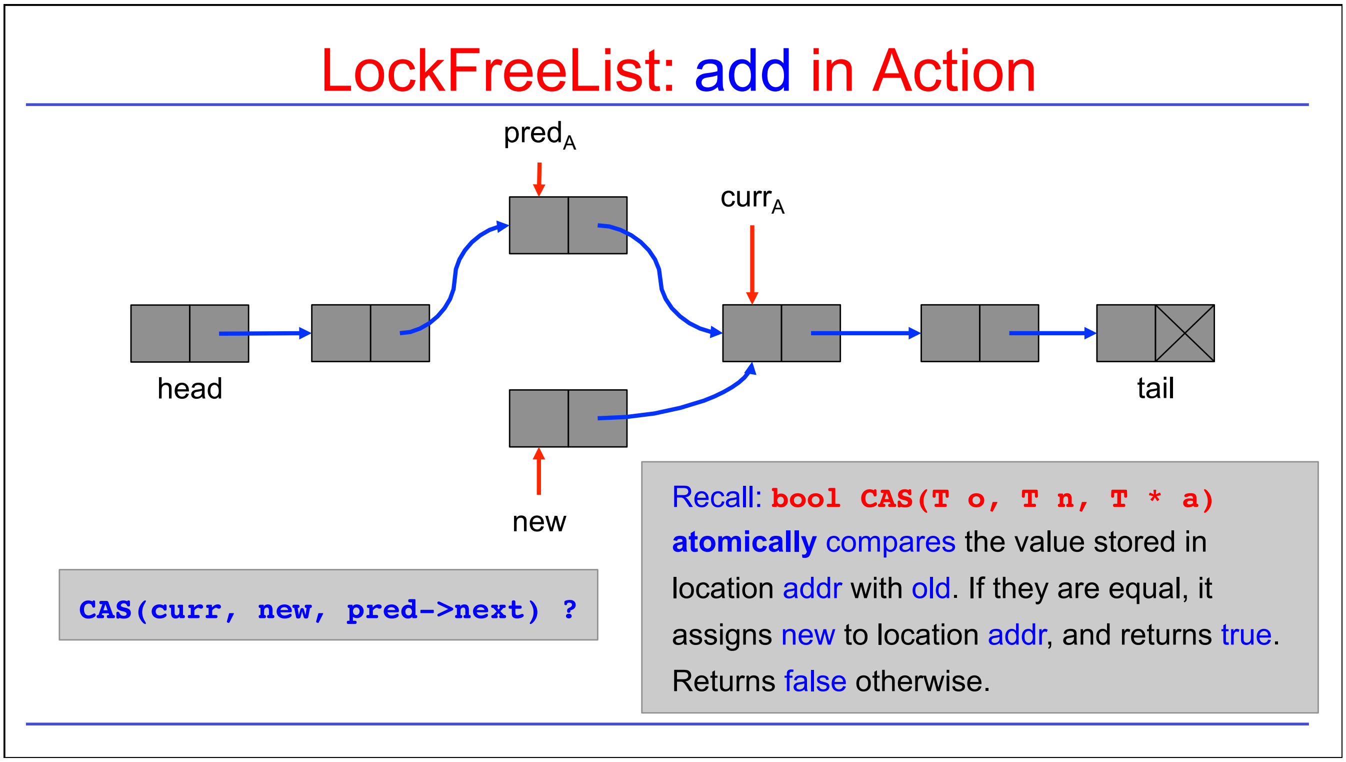


LockFreeList: add in Action

`CAS(curr, new, pred->next) ?`

NO: Restart insertion

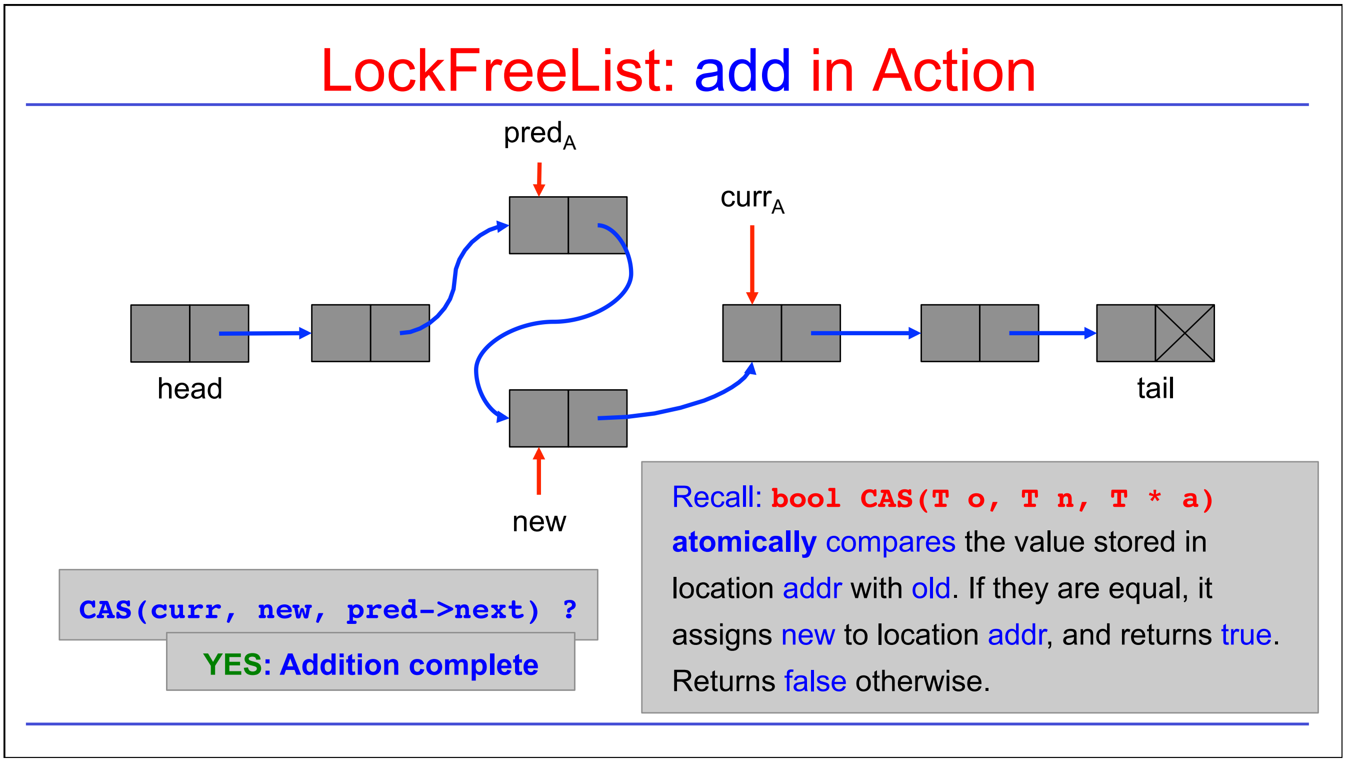
Recall: `bool CAS(T o, T n, T * a)` **atomically** compares the value stored in location addr with old. If they are equal, it assigns new to location addr, and returns true. Returns false otherwise.



LockFreeList: add in Action

`CAS(curr, new, pred->next) ?`

Recall: `bool CAS(T o, T n, T * a)` **atomically** compares the value stored in location addr with old. If they are equal, it assigns new to location addr, and returns true. Returns false otherwise.

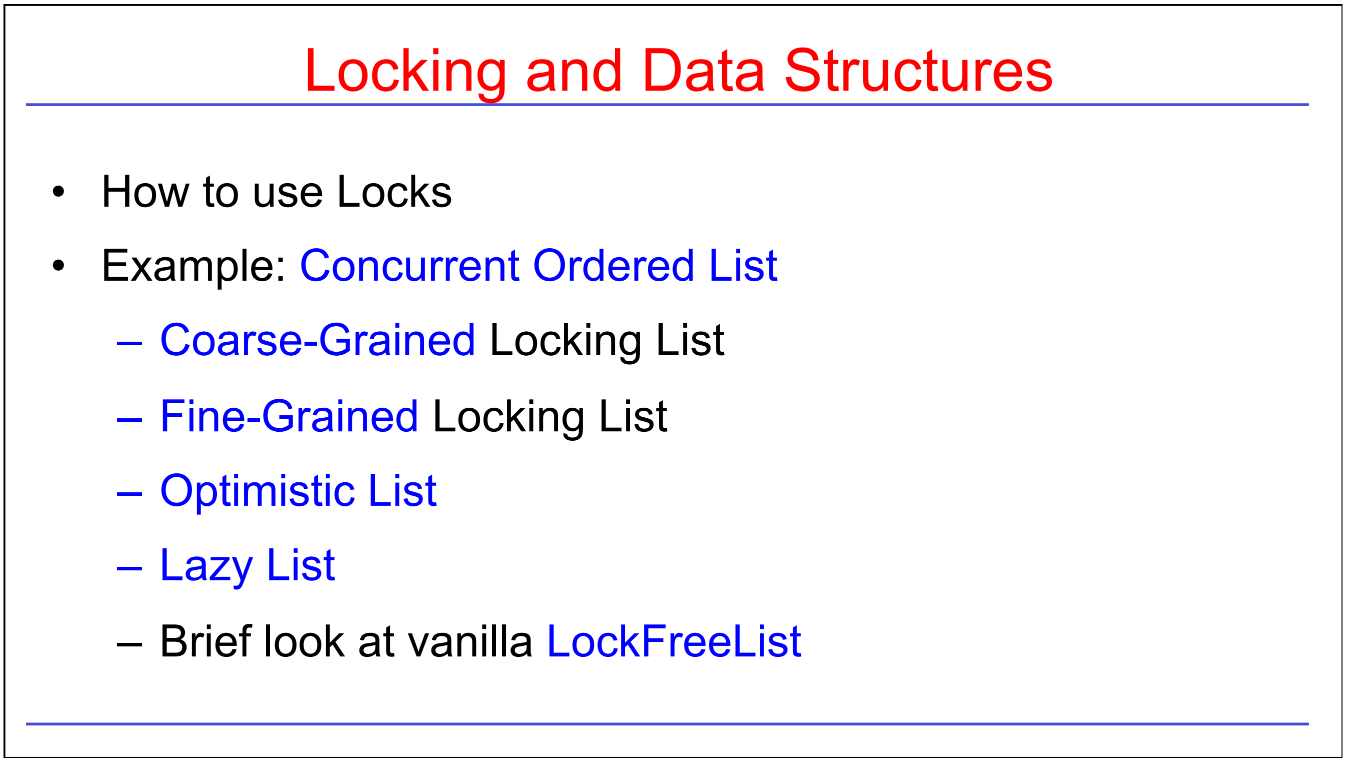


LockFreeList: add in Action

CAS(curr, new, pred->next) ?

YES: Addition complete

Recall: `bool CAS(T o, T n, T * a)` **atomically** compares the value stored in location addr with old. If they are equal, it assigns new to location addr, and returns true. Returns false otherwise.



Locking and Data Structures

- How to use Locks
- Example: Concurrent Ordered List
  - Coarse-Grained Locking List
  - Fine-Grained Locking List
  - Optimistic List
  - Lazy List
  - Brief look at vanilla LockFreeList