COL380

Introduction to
Parallel & Distributed Programming

Synchronization Tools

- Memory Fences, consistent memory
    - Registers
- Atomic operations
    - Test & Set, Fetch & Add, Compare & Swap
- Critical section, Mutex, Ordered
- Barrier
- Lock
- Wait, Condition variables

Compare & Exchange

(aka Compare & Swap)

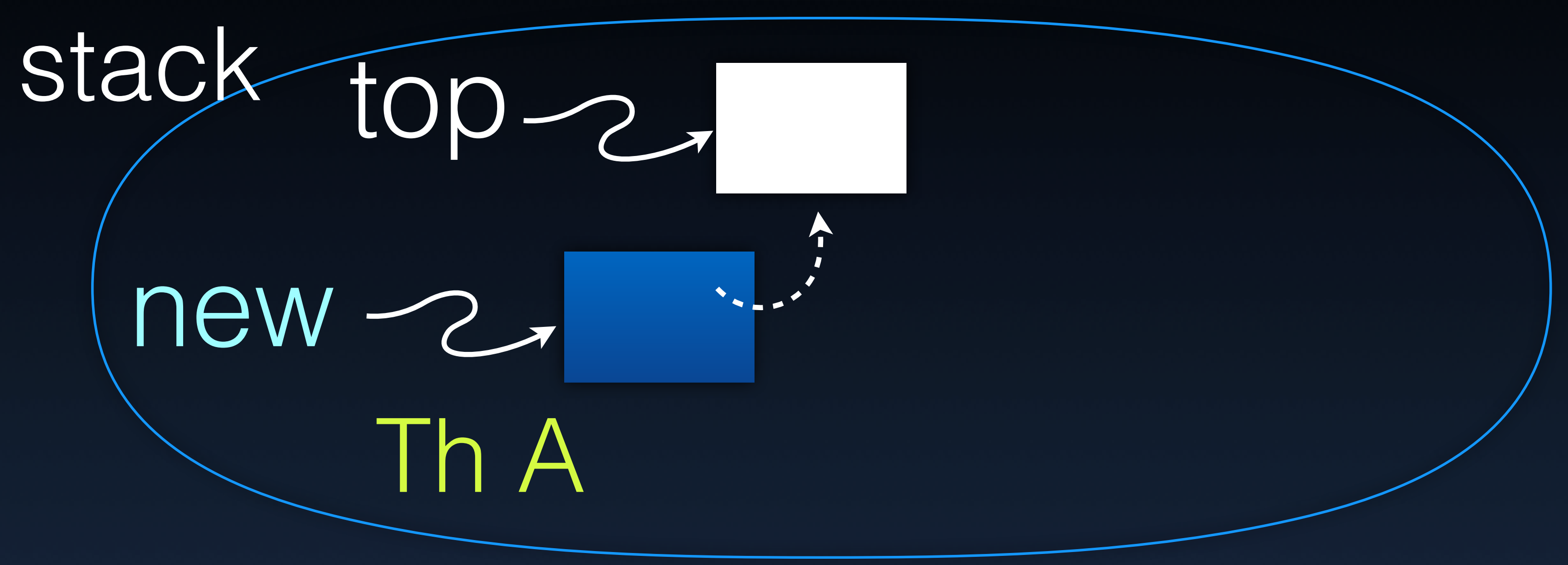
```
std::atomic<int> var(0);

var.compare_exchange_strong(expected, newval);
 // Atomically:
 // t = var.load();
 // if(t == expected) {
 //      var.store(newval);
 //      return true
 // } else {
 //      return false
 // }
```

```
#pragma atomic
    var++;
```

```
#pragma omp atomic capture compare
{
  old = svar;
  if (old == expected) svar = newval;
}
  // old == expected ⇒ success
```

Compare & Exchange

(aka Compare & Swap)



```
std::atomic<int> var(0);

var.compare_exchange_strong(expected, newval);
  // Atomically:
  // t = var.load();
  // if(t == expected) {
  //     var.store(newval);
  //     return true
  // } else {
  //     return false
  // }
```

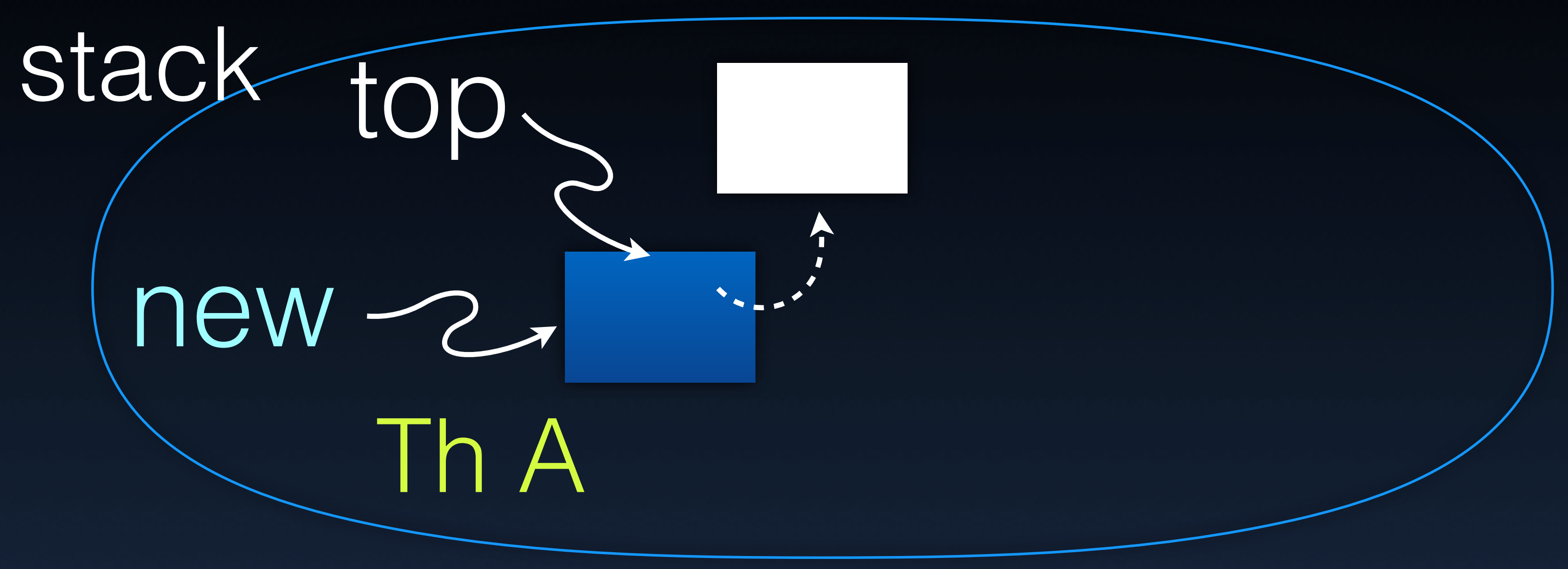
```
#pragma atomic
    var++;
```

```
#pragma omp atomic capture compare
{
  old = svar;
  if (old == expected) svar = newval;
}
  // old == expected ⇒ success
```

Subodh Kumar

Compare & Exchange

(aka Compare & Swap)



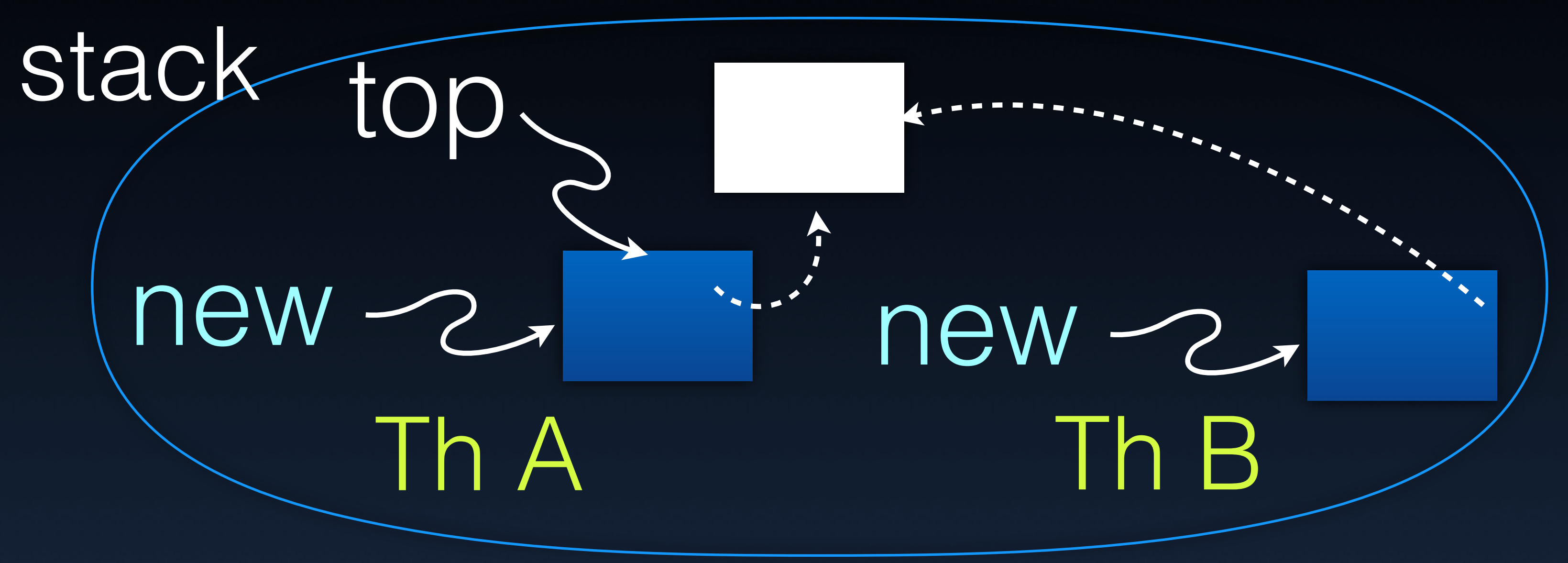
```
std::atomic<int> var(0);

var.compare_exchange_strong(expected, newval);
  // Atomically:
  // t = var.load();
  // if(t == expected) {
  //      var.store(newval);
  //      return true
  // } else {
  //      return false
  // }
```

```
#pragma atomic
     var++;
```

```
#pragma omp atomic capture compare
{
   old = svar;
   if (old == expected) svar = newval;
}
   // old == expected ⇒ success
```

Compare & Exchange

(aka Compare & Swap)



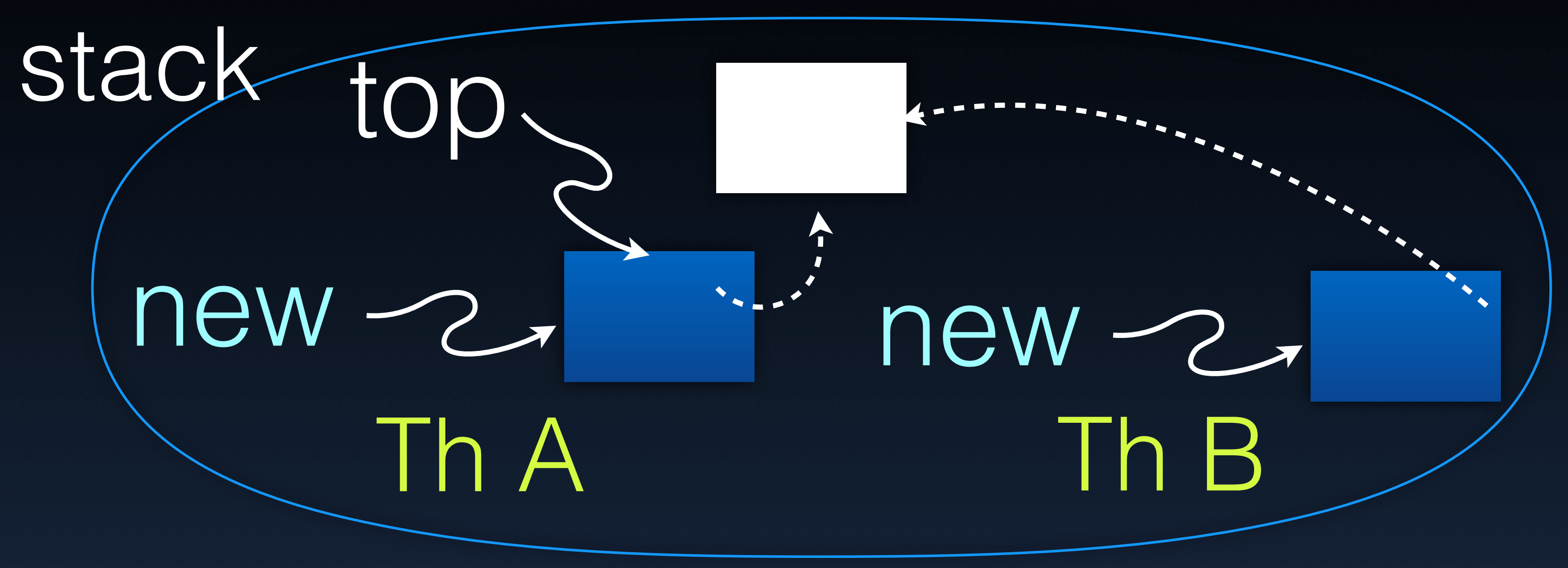
```
std::atomic<int> var(0);

var.compare_exchange_strong(expected, newval);
  // Atomically:
  // t = var.load();
  // if(t == expected) {
  //       var.store(newval);
  //       return true
  // } else {
  //       return false
  // }
```

```
#pragma atomic
     var++;
```

```
#pragma omp atomic capture compare
{
   old = svar;
   if (old == expected) svar = newval;
}
   // old == expected ⇒ success
```

Compare & Exchange

(aka Compare & Swap)



```
std::atomic<int> var(0);

var.compare_exchange_strong(expected, newval);
  // Atomically:
  //
  //
  //
  //
  //
  //
  //
```

```
#pragma atomic
```

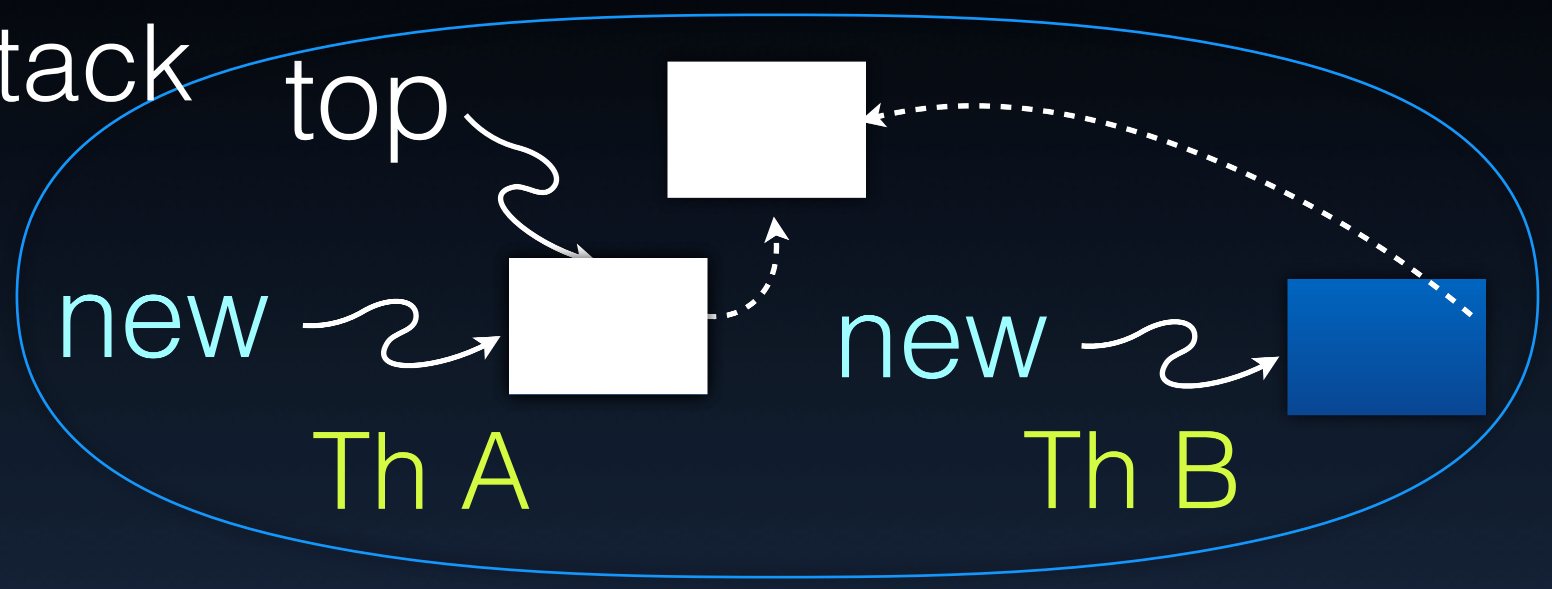
```
are
l;
```

```
std::atomic<node<T>*> top;
…
void push(const T& data) {
    node<T>* new_node = new node<T>(data);

    // put the current value of top into new_node->next
    new_node->next = top.load();

    // Update top to point to the new node
        top.store(new_node);
}
```

Subodh Kumar

Compare & Exchange

(aka Compare & Swap)



```
std::atomic<int> var(0);

var.compare_exchange_strong(expected, newval);
  // Atomically:
  //
  //
  //
  //
  //
  //
  //
```

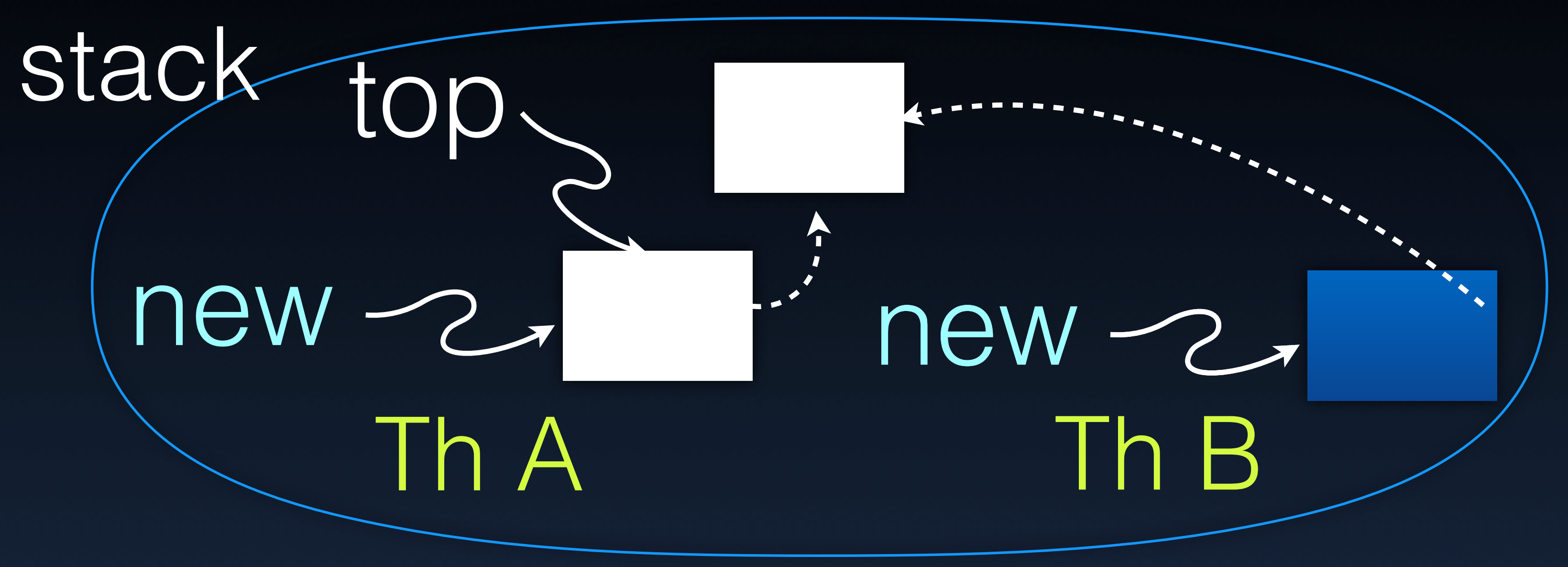
#pragma atomic

are

l;

```
std::atomic<node<T>*> top;
…
void push(const T& data) {
      node<T>* new_node = new node<T>(data);

      // put the current value of top into new_node->next
      new_node->next = top.load();

      // Update top to point to the new node
            top.store(new_node);
}
```

Compare & Exchange



```
std::atomic<int> var(0);

var.compare_exchange_strong(expected, newval);
  // Atomically:
  //
  //
  //
  //
  //
  //
  //
```

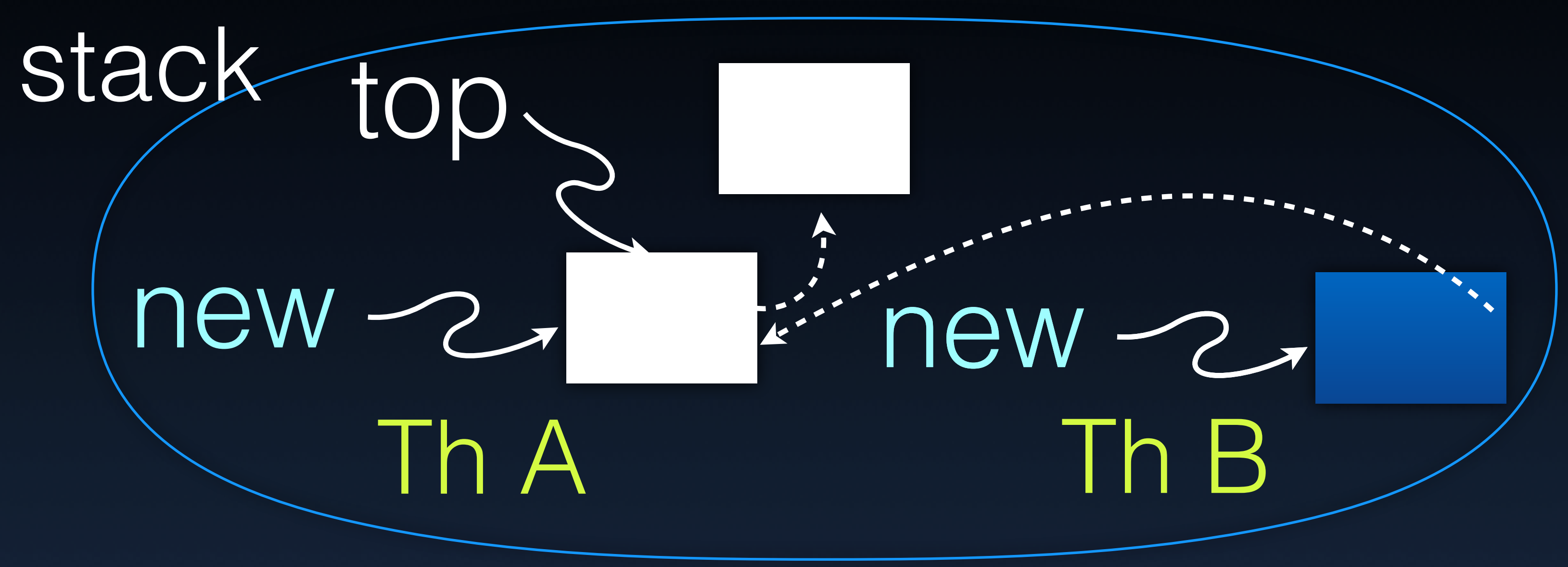
```
#pragma atomic
```

are

l;

```
std::atomic<node<T>*> top;
…
void push(const T& data) {
     node<T>* new_node = new node<T>(data);

     // put the current value of top into new_node->next
     do new_node->next = top.load();

     // make new_node the top, as long as top still equals new_node->next
      while(!top.compare_exchange_strong(new_node->next, new_node));
}
```

Compare & Exchange



```
std::atomic<int> var(0);

var.compare_exchange_strong(expected, newval);
  // Atomically:
  //
  //
  //
  //
  //
  //
  //
```

#pragma atomic

are

l;

```
std::atomic<node<T>*> top;
…
void push(const T& data) {
     node<T>* new_node = new node<T>(data);

     // put the current value of top into new_node->next
     do new_node->next = top.load();

     // make new_node the top, as long as top still equals new_node->next
     while(!top.compare_exchange_strong(new_node->next, new_node));
}
```

Fetch & Add

```
#pragma omp atomic capture
{
  old = svar;
  svar += tval;
}
```

```
#pragma omp atomic capture
{
  old = slock;
  slock += 1;
}
if(old == 0) {
   criticalSection();
   #pragma omp atomic
   slock--;
} else {
   havefuninthesun();
}
```

Test & Set

```
#pragma omp atomic capture
{
  old = svar;
  svar = tval;
}
```

```
#pragma omp atomic capture
{
  old = slock;
  slock = 1;
}
if(old == 0) {
   criticalSection();
   #pragma omp atomic write
   slock=0;
} else {
   havefuninthesun();
}
```

Condition Variable

- Raise the condition
- Wait for a condition to ‘hold’

```
Produce();
acv.notify_one();
```

```
std::condition_variable acv;
…

std::unique_lock<std::mutex> alock(amutex);
acv.wait(alock);
.. Condition Holds Now ..
Consume();
```

Condition Variable

```
void producer(std::condition_variable *cv) {
    while(1) {
        produce();
        cv->notify_one();
    }
}

void consumer(std::mutex *mtx,
                         std::condition_variable *cv) {
    while(1) {
       std::unique_lock<mutex> lock(*mtx);
       cv->wait(lock, 1);
       consume();

    }
}
```

```
{
    std::mutex mtx;
    std::condition_variable cv;

    thread p(producer, &cv);
    thread c(consumer, &mtx, &cv);

    c.join(); p.join();
}
```

Condition Variable

```
void producer(std::condition_variable *cv) {
   while(1) {
      produce(); counter++;
      cv->notify_one();
   }
}

void consumer(std::mutex *mtx,
                   std::condition_variable *cv) {
   while(1) {
     std::unique_lock<mutex> lock(*mtx);
     cv->wait(lock, [] { return counter > 0; }
     consume();
     counter--;
   }
}
```

```
   std::atomic<int> counter{0};
```

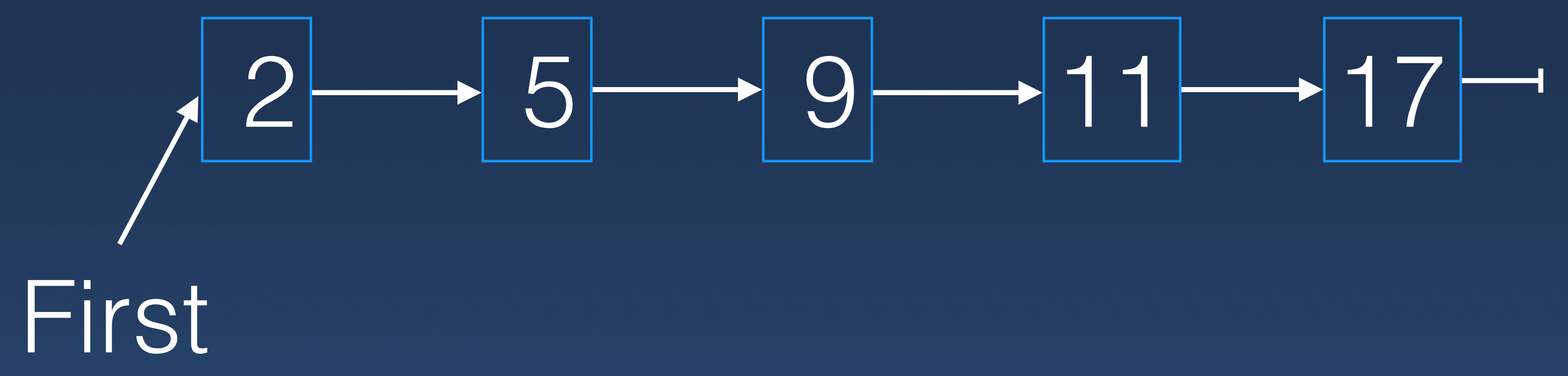
```
{
   std::mutex mtx;
   std::condition_variable cv;

   thread p(producer, &cv);
   thread c(consumer, &mtx, &cv);

   c.join(); p.join();
}
```

Subodh Kumar

Lock

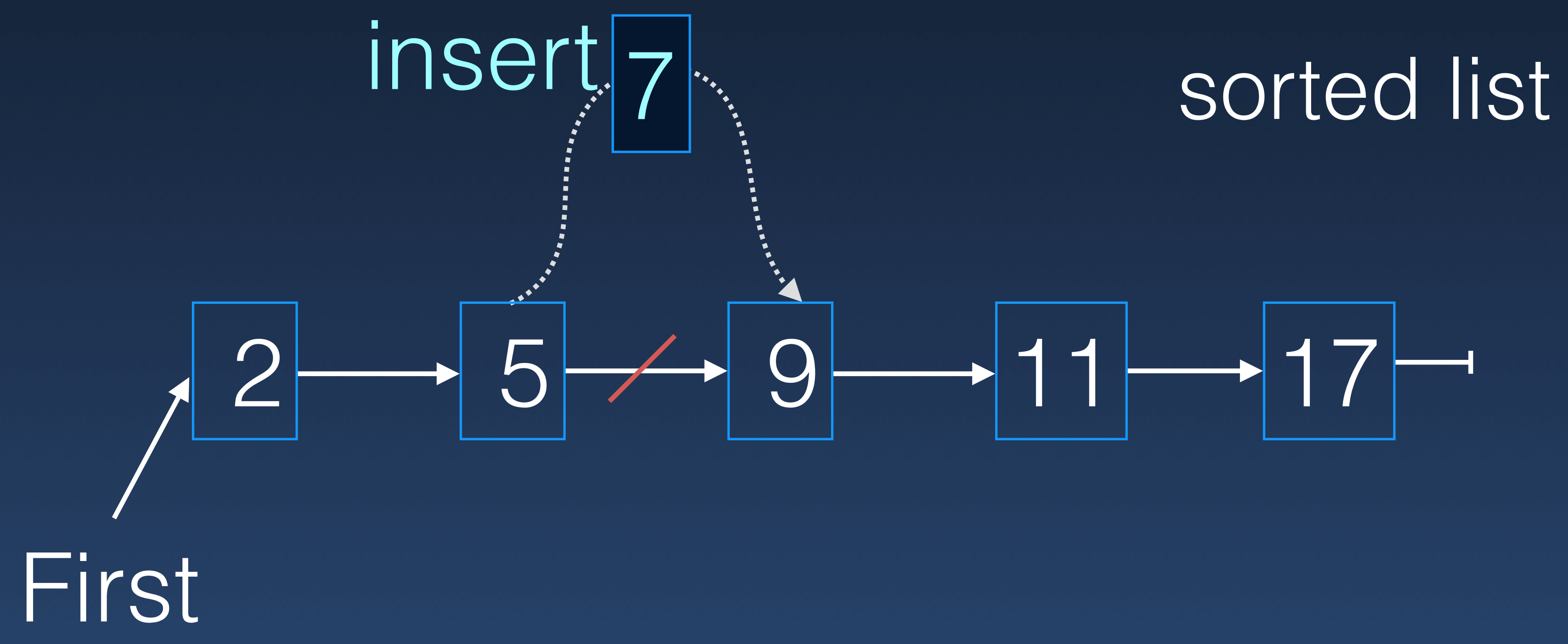
- Lock “resources”
- Process
- Unlock “resources”

sorted list



Lock

- Lock “resources”
- Process
- Unlock “resources”

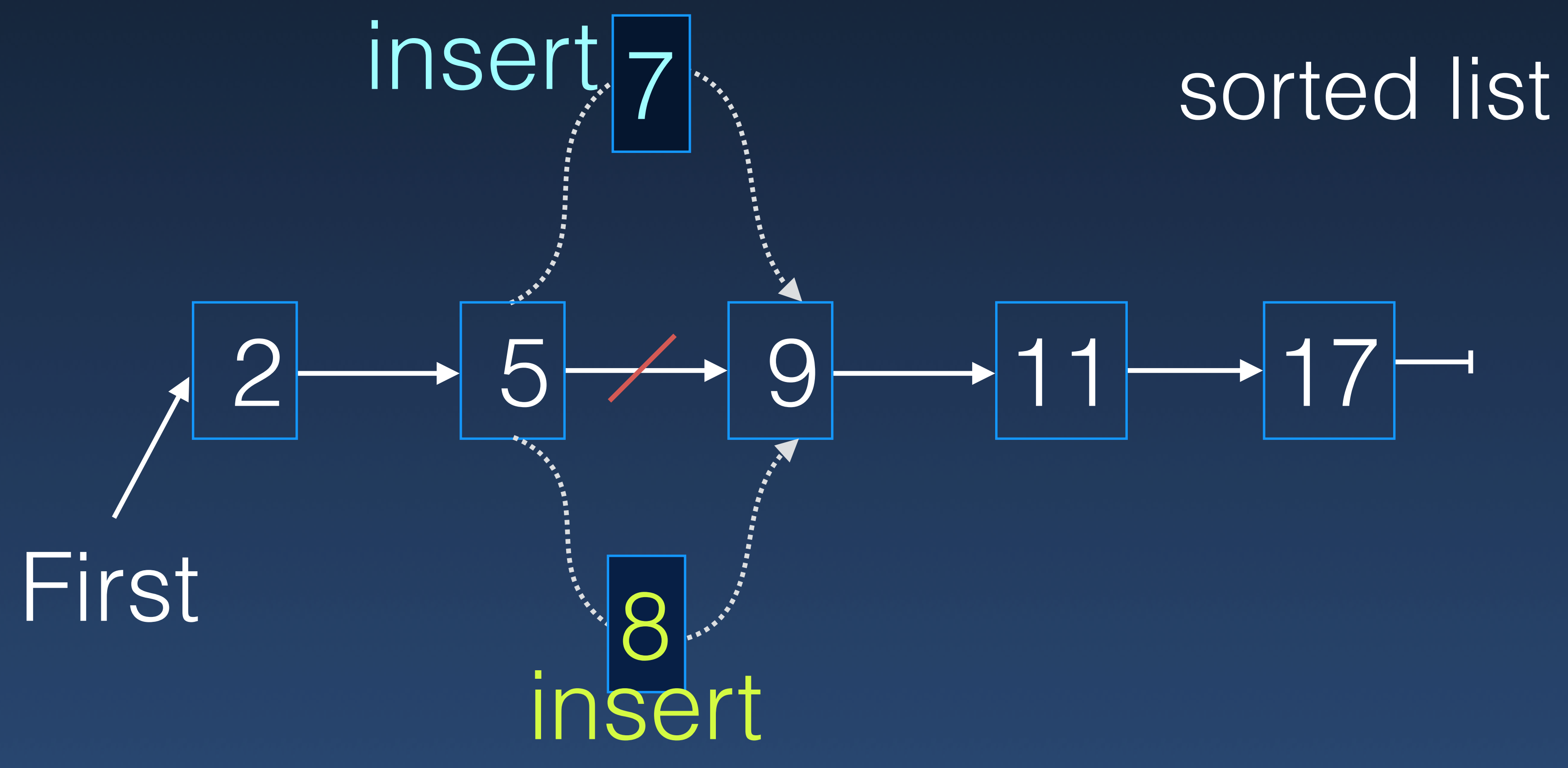


Lock

- Lock “resources”
- Process
- Unlock “resources”

Correctness?

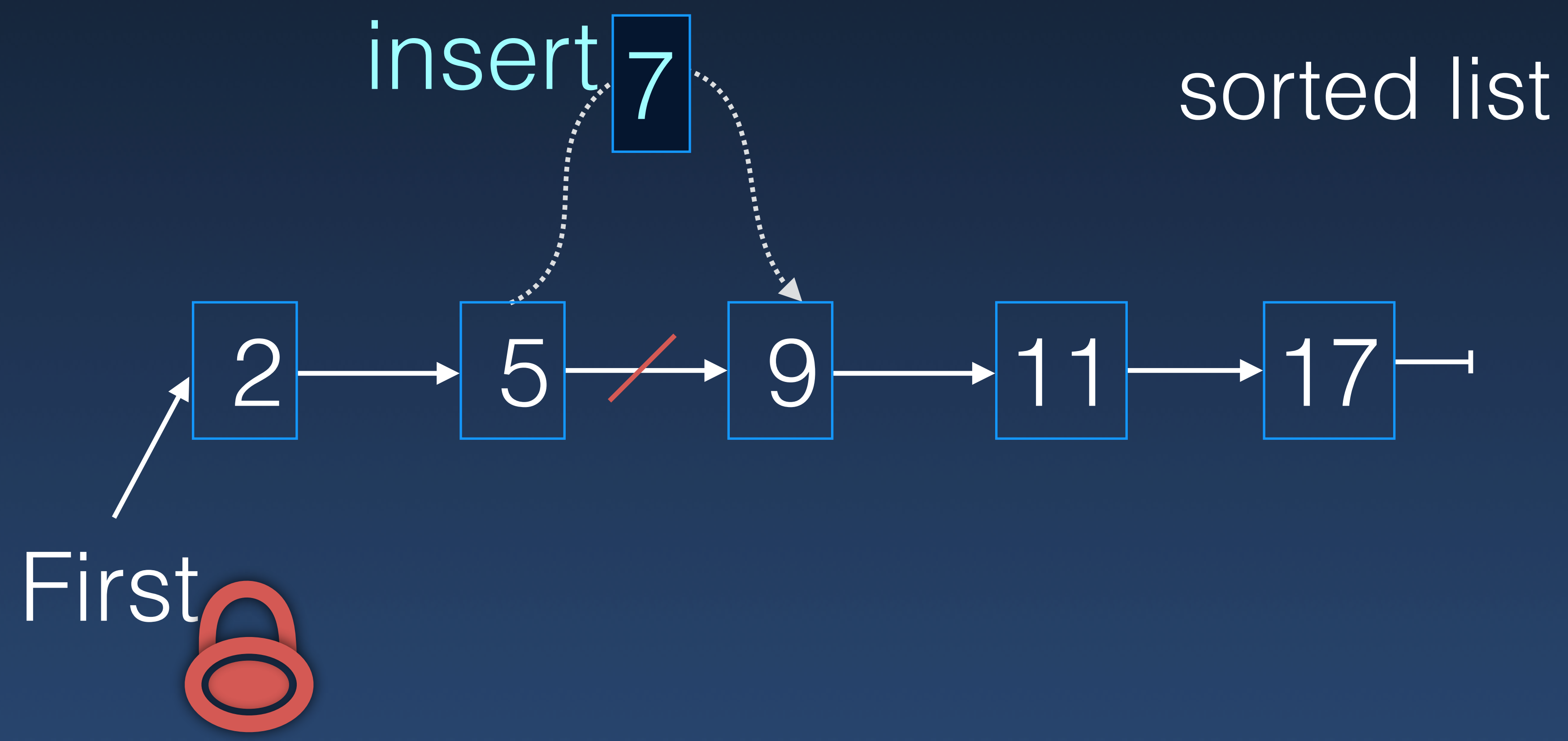
“Sequential Equivalence”



Subodh Kumar

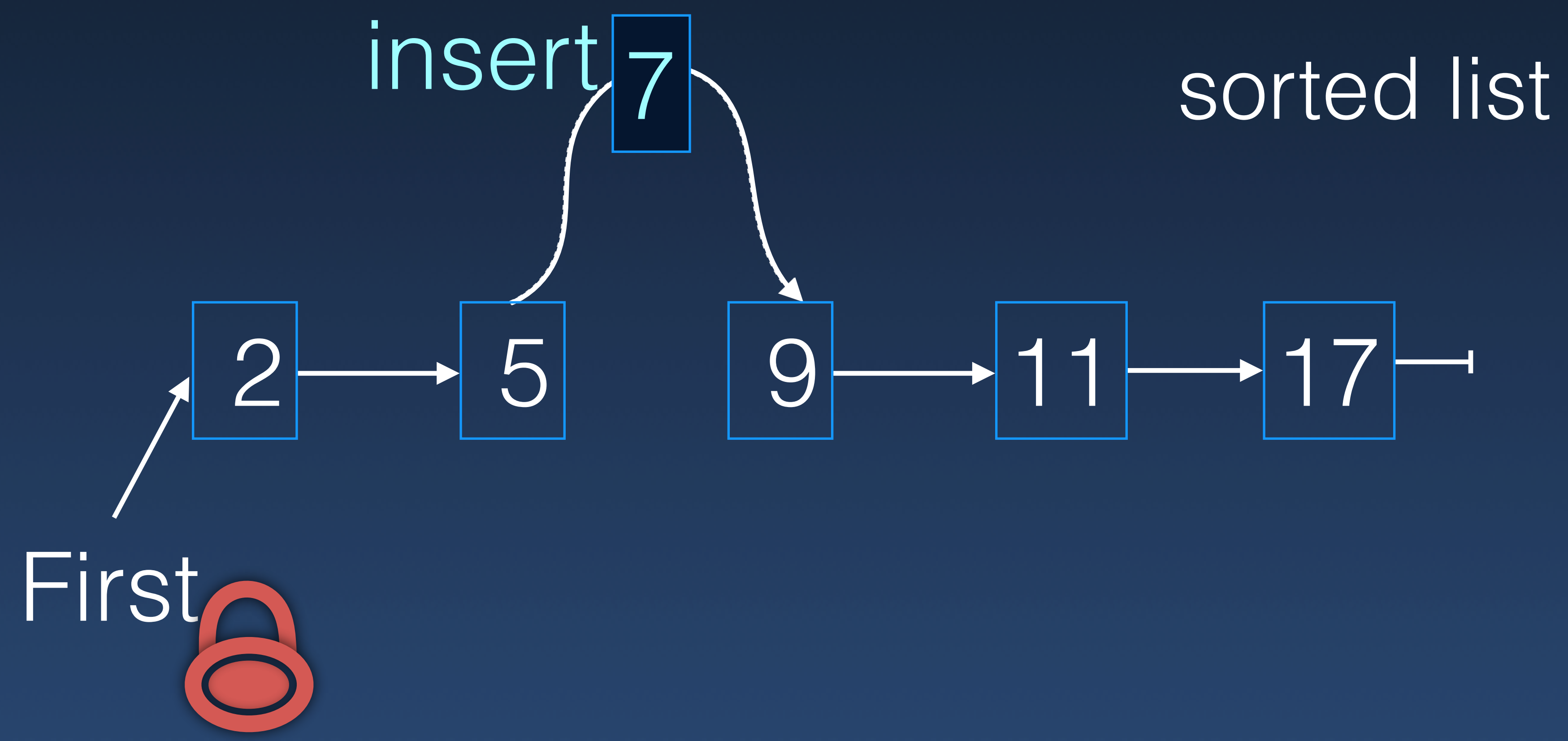
Lock

- Lock “resources”
- Process
- Unlock “resources”



Lock

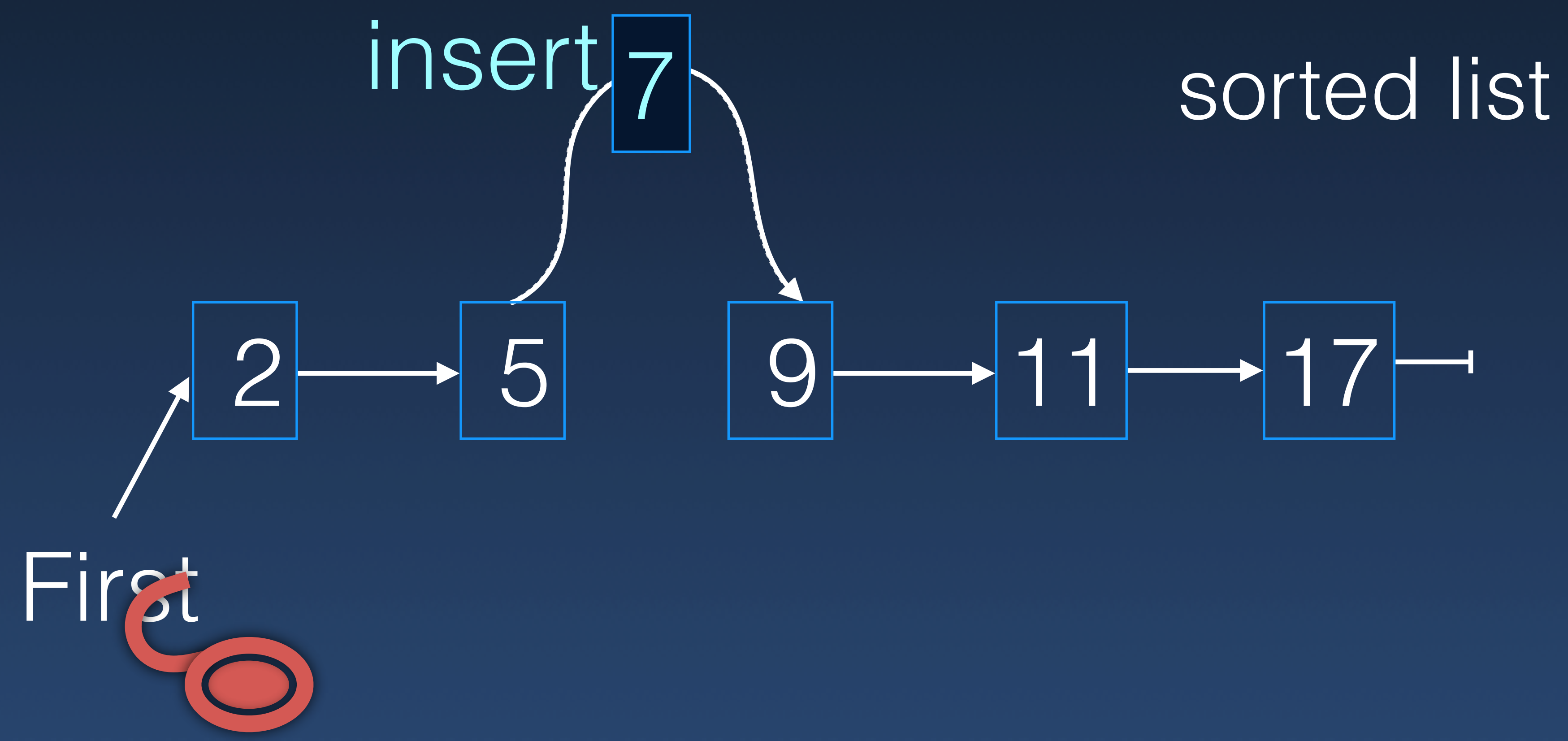
- Lock “resources”
- Process
- Unlock “resources”



Subodh Kumar

Lock

- Lock “resources”
- Process
- Unlock “resources”

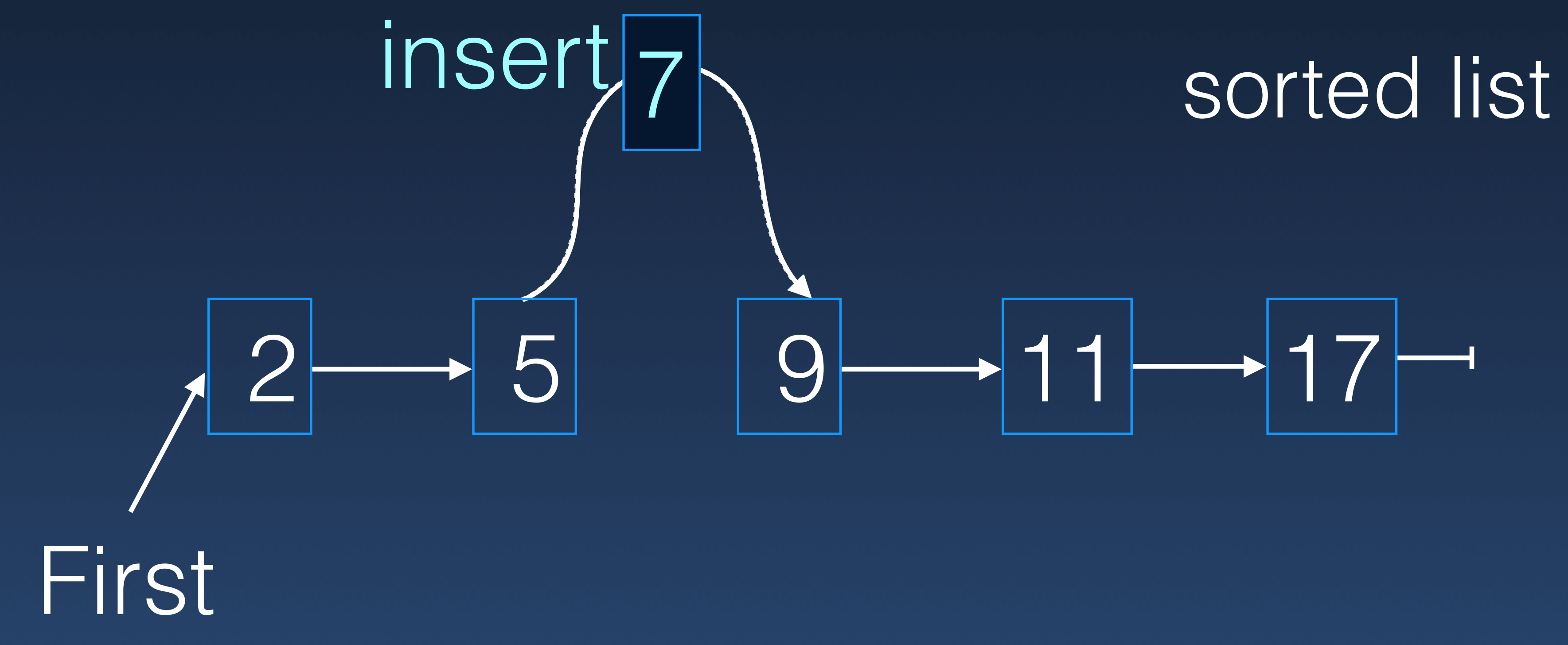


Subodh Kumar

Lock

- Lock “resources”
- Process
- Unlock “resources”

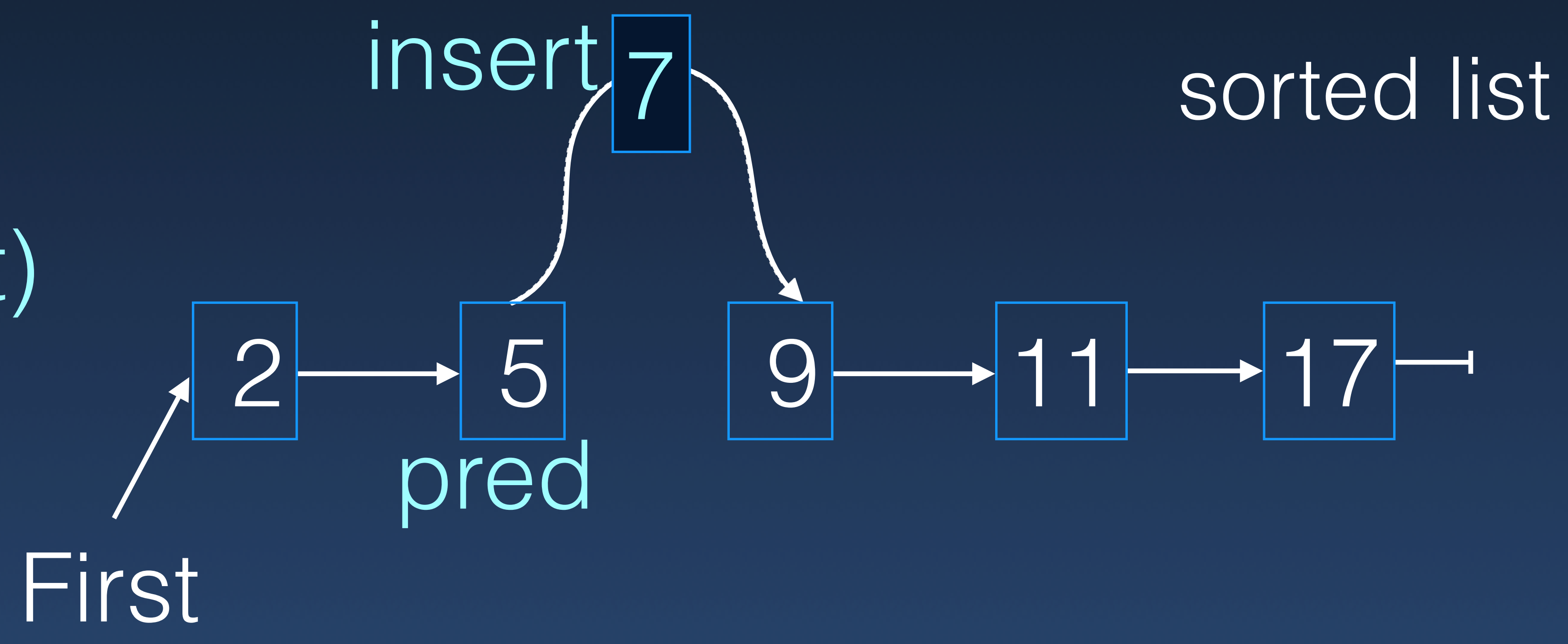
lock($lockA$)



Lock

- Lock “resources”
- Process
- Unlock “resources”

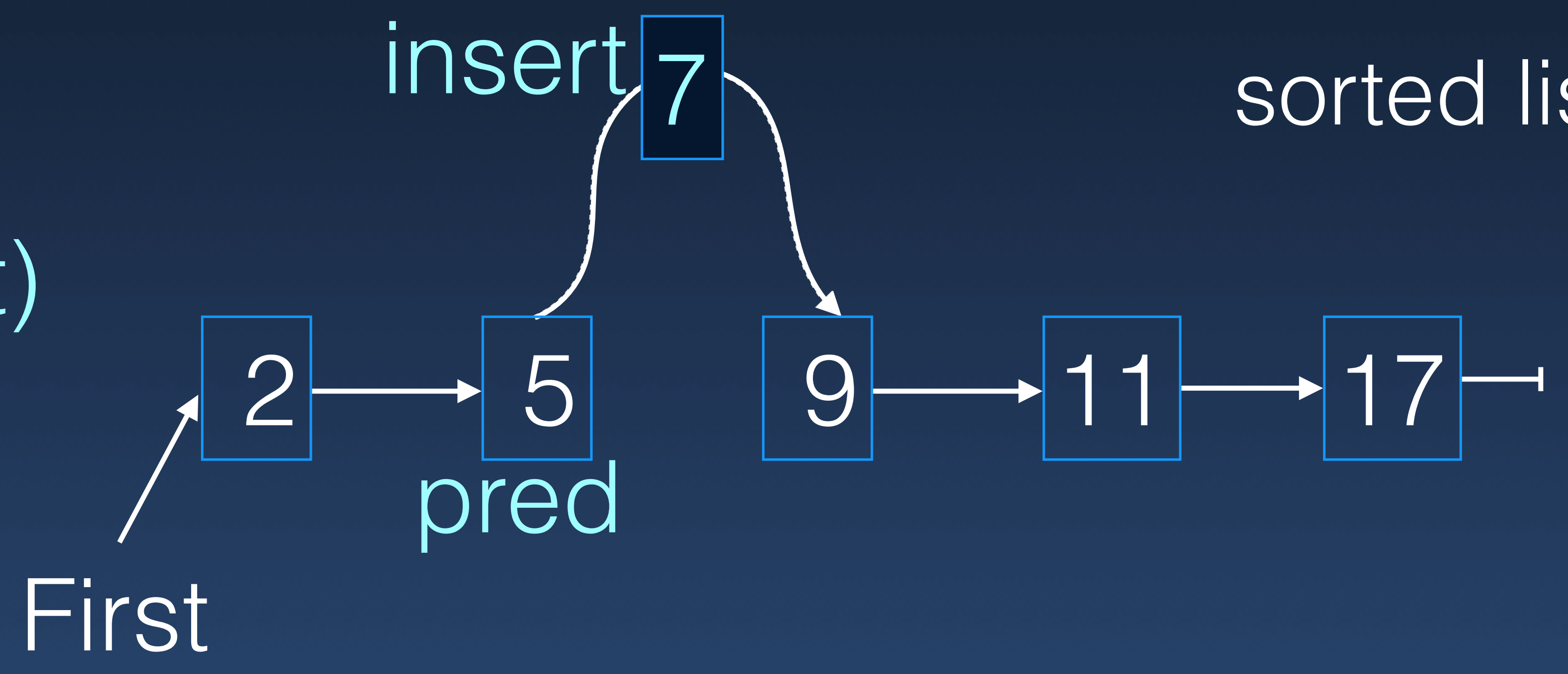
```
lock($lockA$)

pred = Find(key)
pred.nxt = Node(key, pred.nxt)

unlock($lockA$)
```



Lock

- Lock “resources”
- Process
- Unlock “resources”

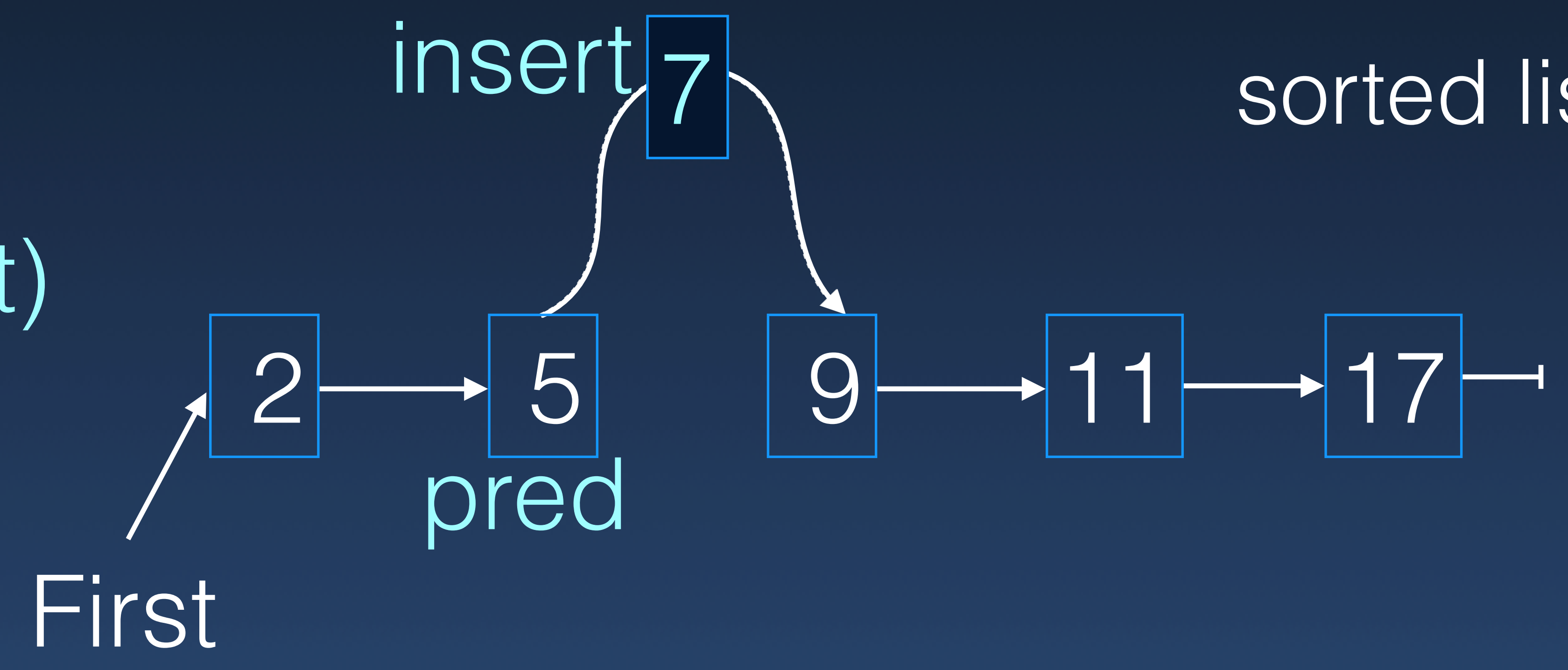
```
<Request> [?block] <Acquired>
  lock(lockA)

  pred = Find(key)
  pred.nxt = Node(key, pred.nxt)

  unlock(lockA)
 <Release> [schedule]
```



Lock

- Lock “resources”
- Process
- Unlock “resources”

```
<Request> [?block] <Acquired>
 lock(lockA)

 pred = Find(key)
 pred.nxt = Node(key, pred.nxt)

 unlock(lockA)
<Release> [schedule]
```



```
C++:
    std::mutex m;
    std::lock(m);
    doCriticalwork();
    std::unlock(m);
```

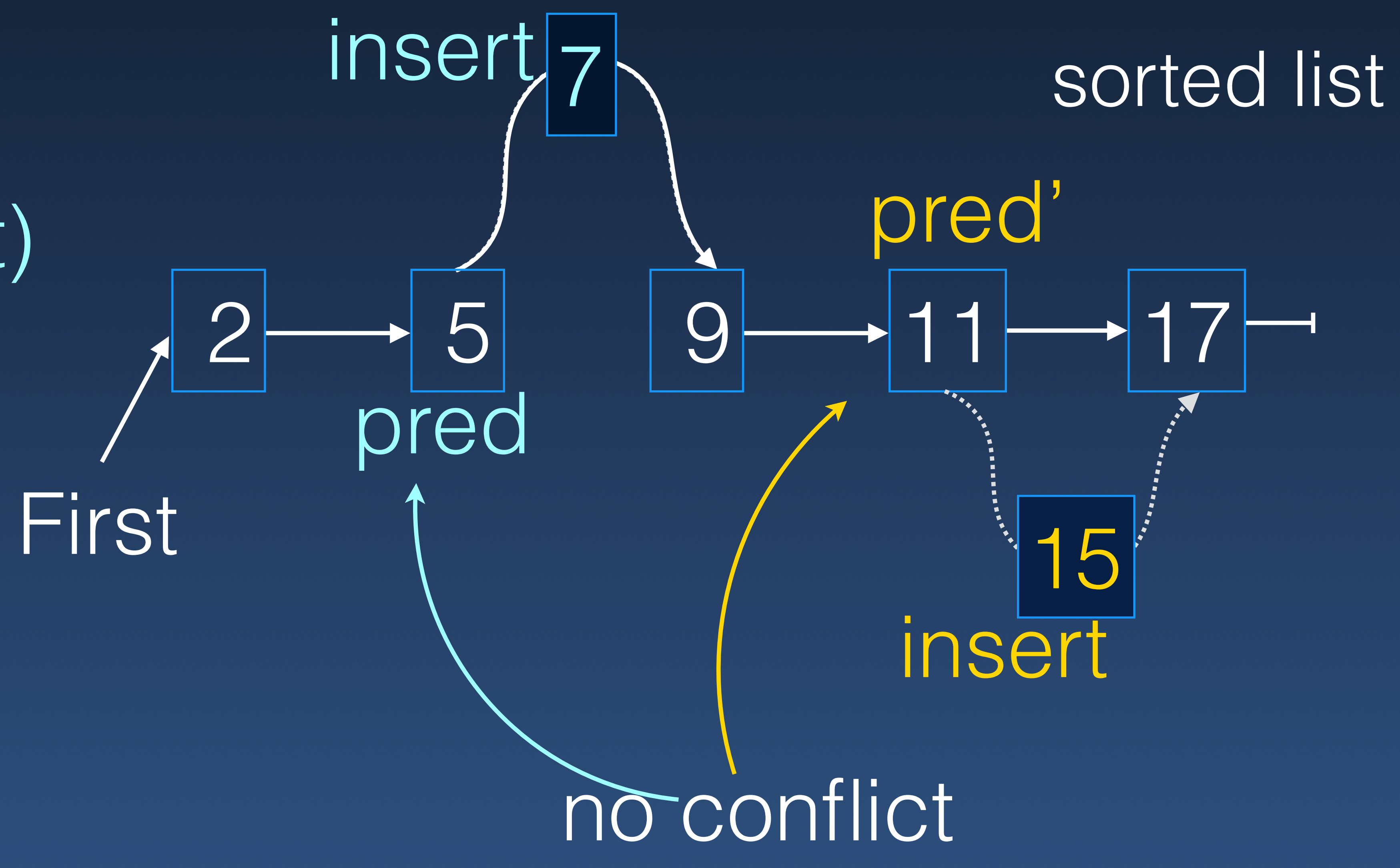
Lock

<Request> [?block] <Acquired>

- Lock “resources” lock($lockA$)
- Process pred = Find(key)
  pred.nxt = Node(key, pred.nxt)
- Unlock “resources” unlock($lockA$)

<Release> [schedule]

```
C++:
    std::mutex m;
    std::lock(m);
    doCriticalwork();
    std::unlock(m);
```

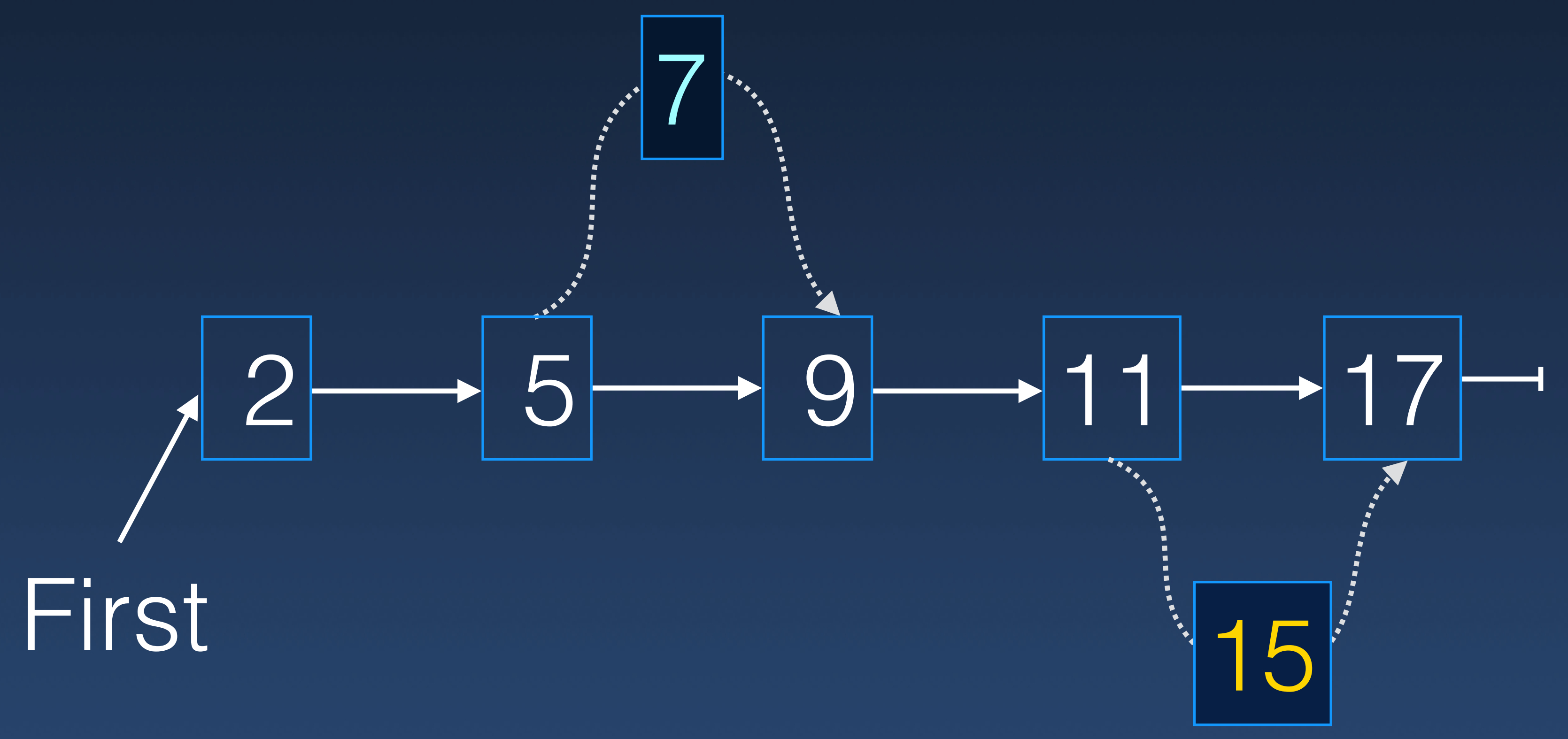


Lock the entire list?

Subodh Kumar

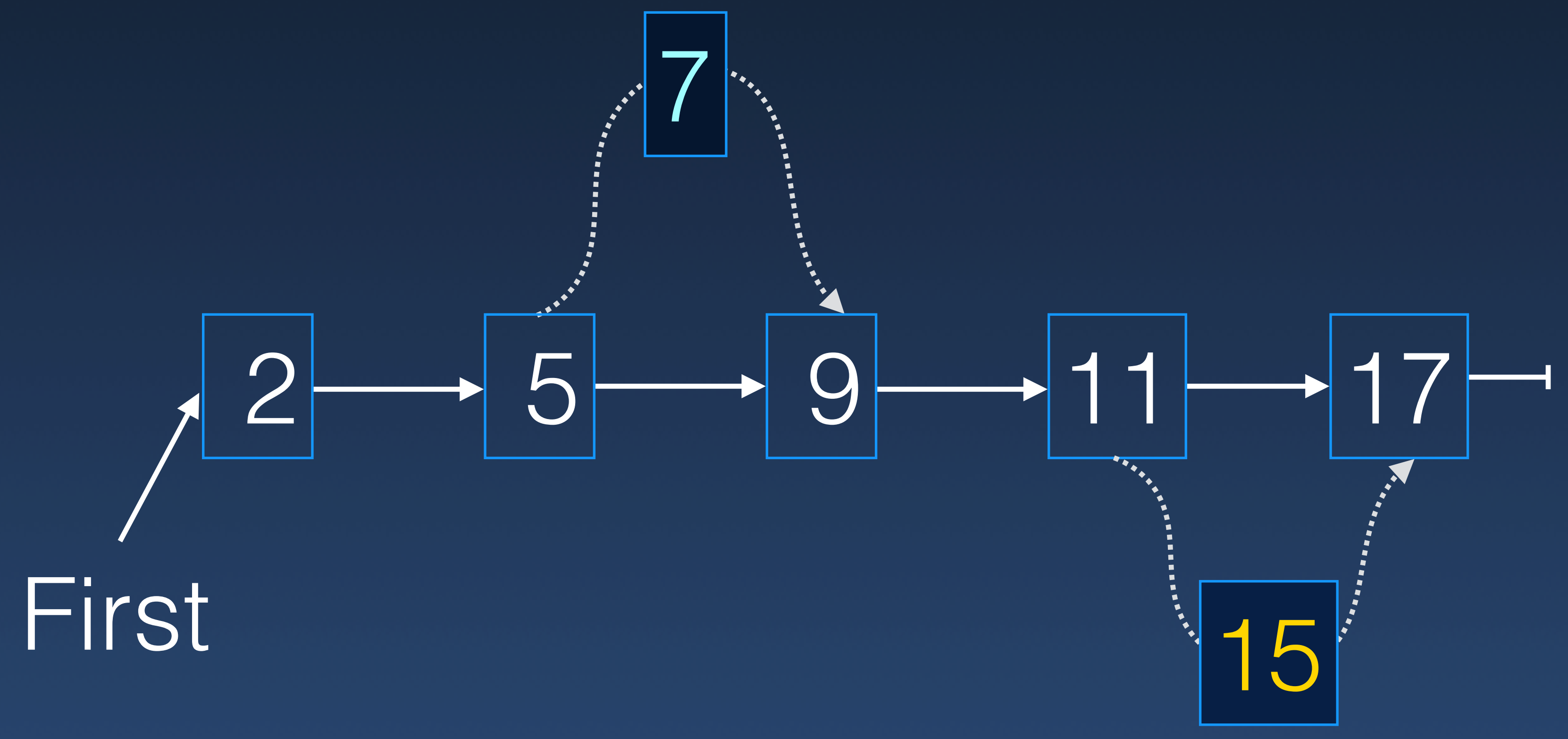
List using Lock

- Lock “resources”
- Process
- Unlock “resources”



List using Lock

- Lock “resources”
- Process
- Unlock “resources”

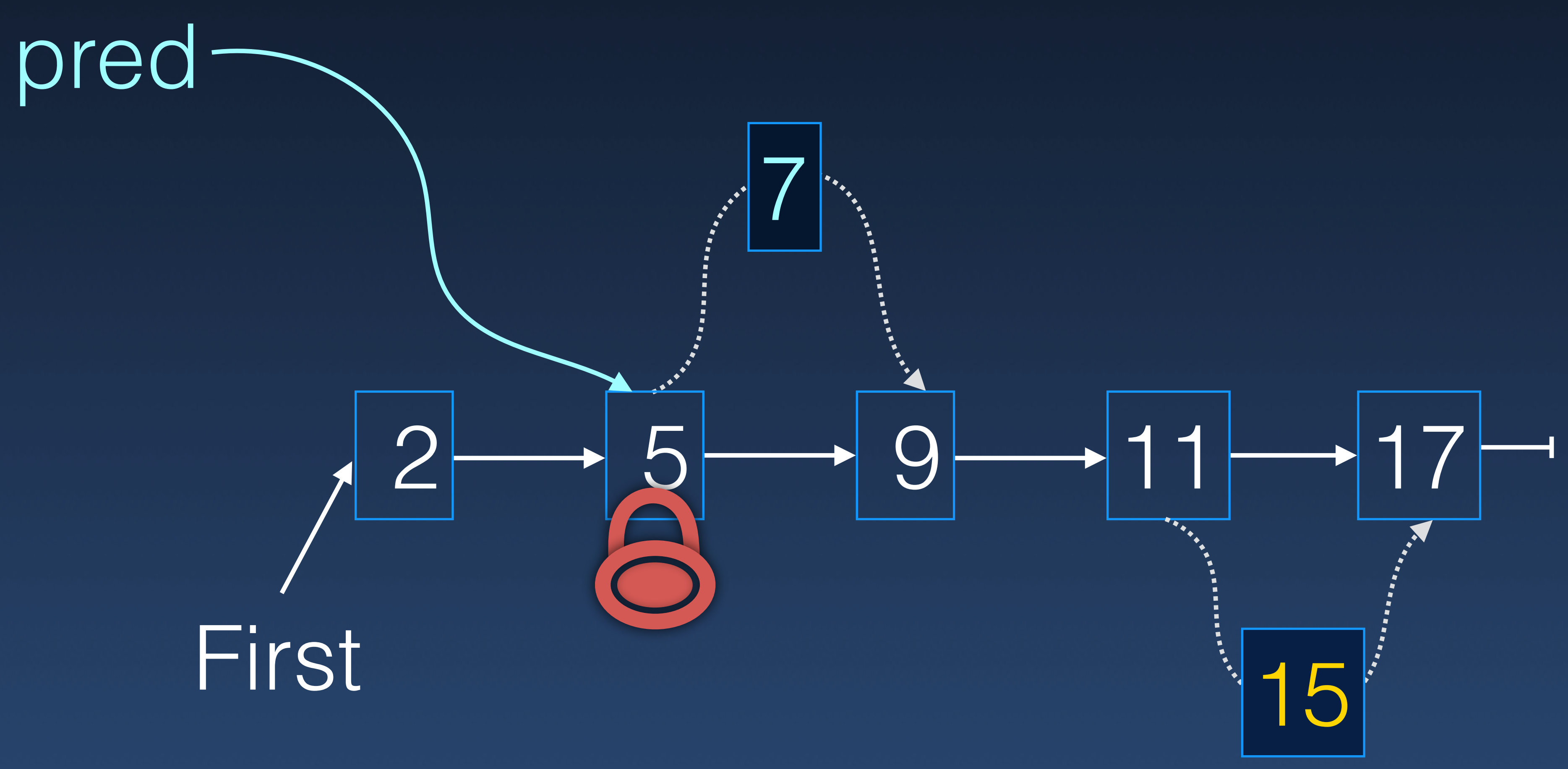


```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

Subodh Kumar

List using Lock

- Lock “resources”
- Process
- Unlock “resources”



Insertion Loop

```
lock(pred)
if(key in [pred->key:pred->nxt->key)) {
      pred->nxt = Node(key, pred->nxt, new(Lock))
}
unlock(pred)
pred = pred->nxt
```

```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

Subodh Kumar

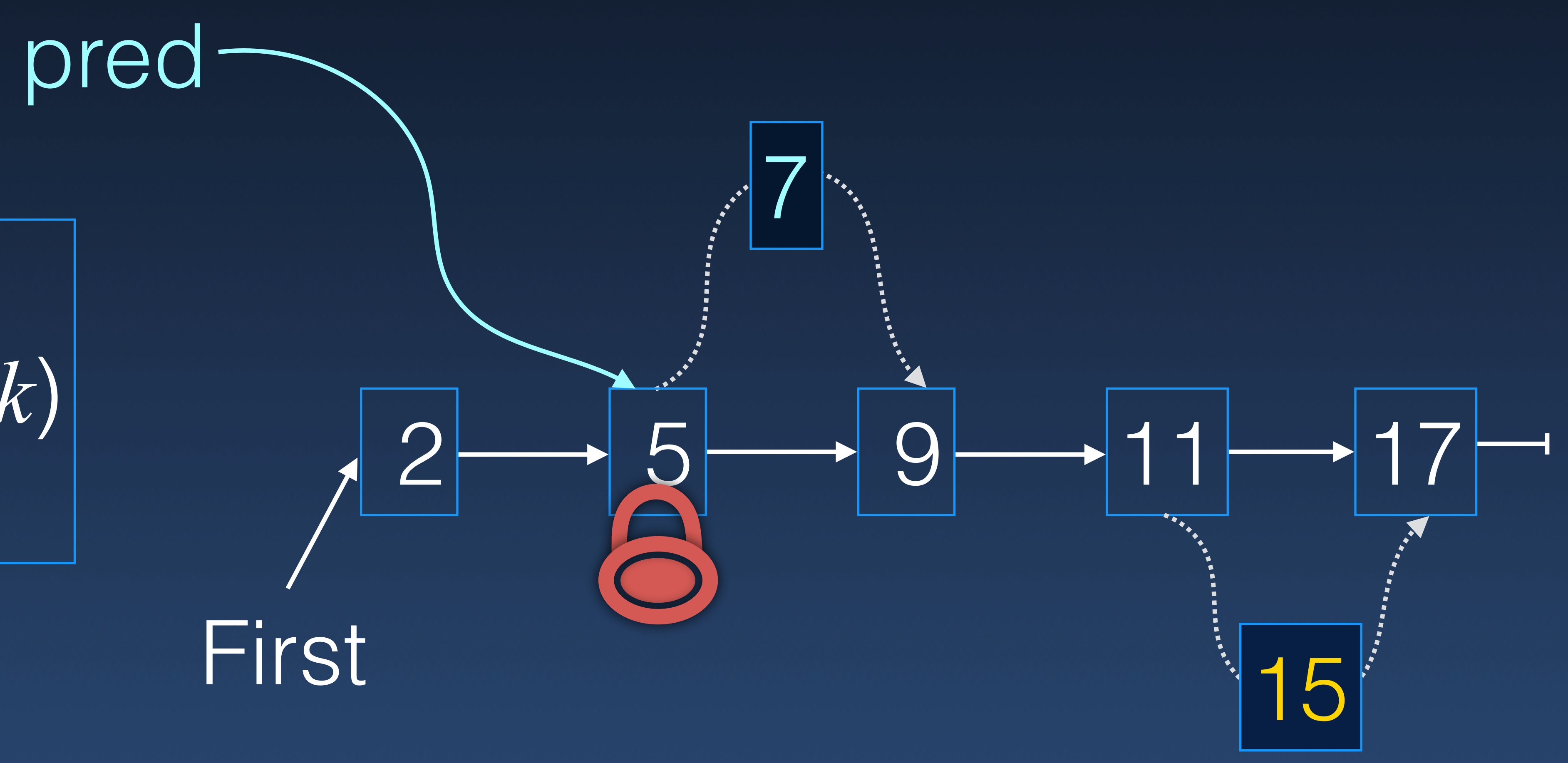
List using Lock

- Lock “resources”
- Process
- Unlock “resources”

e.g.,
omp_set_lock(&pred->*lock*)
or, pred->*lock*.lock()

Insertion Loop

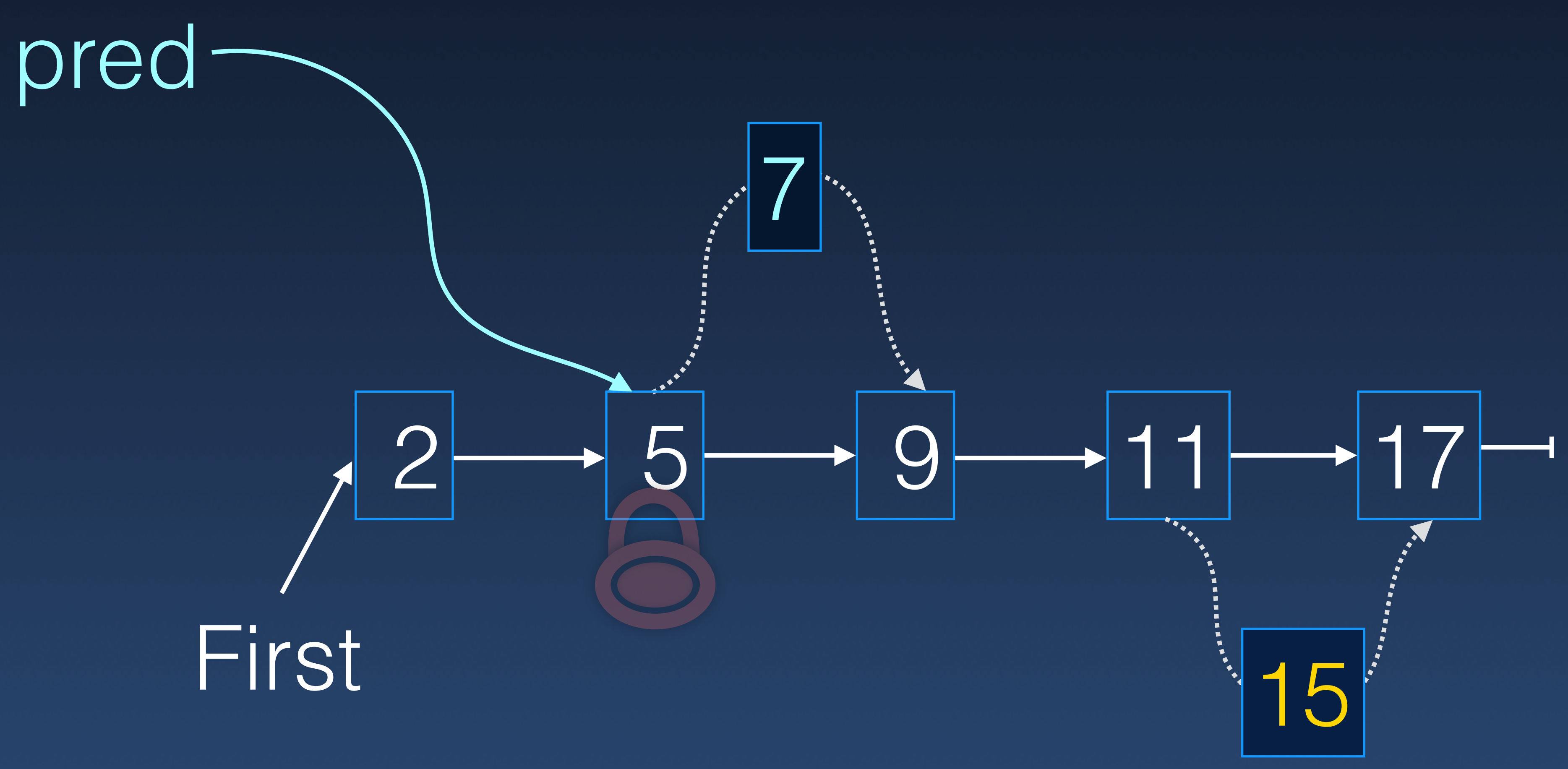
```
lock(pred)
if(key in [pred->key:pred->nxt->key)) {
     pred->nxt = Node(key, pred->nxt, new(Lock))
}
unlock(pred)
pred = pred->nxt
```



```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

List using Lock

- Lock “resources”
- Process
- Unlock “resources”



Insertion Loop

```
lock(pred)
if(key in [pred->key:pred->nxt->key)) {
      pred->nxt = Node(key, pred->nxt, new(Lock))
}
unlock(pred)
pred = pred->nxt
```

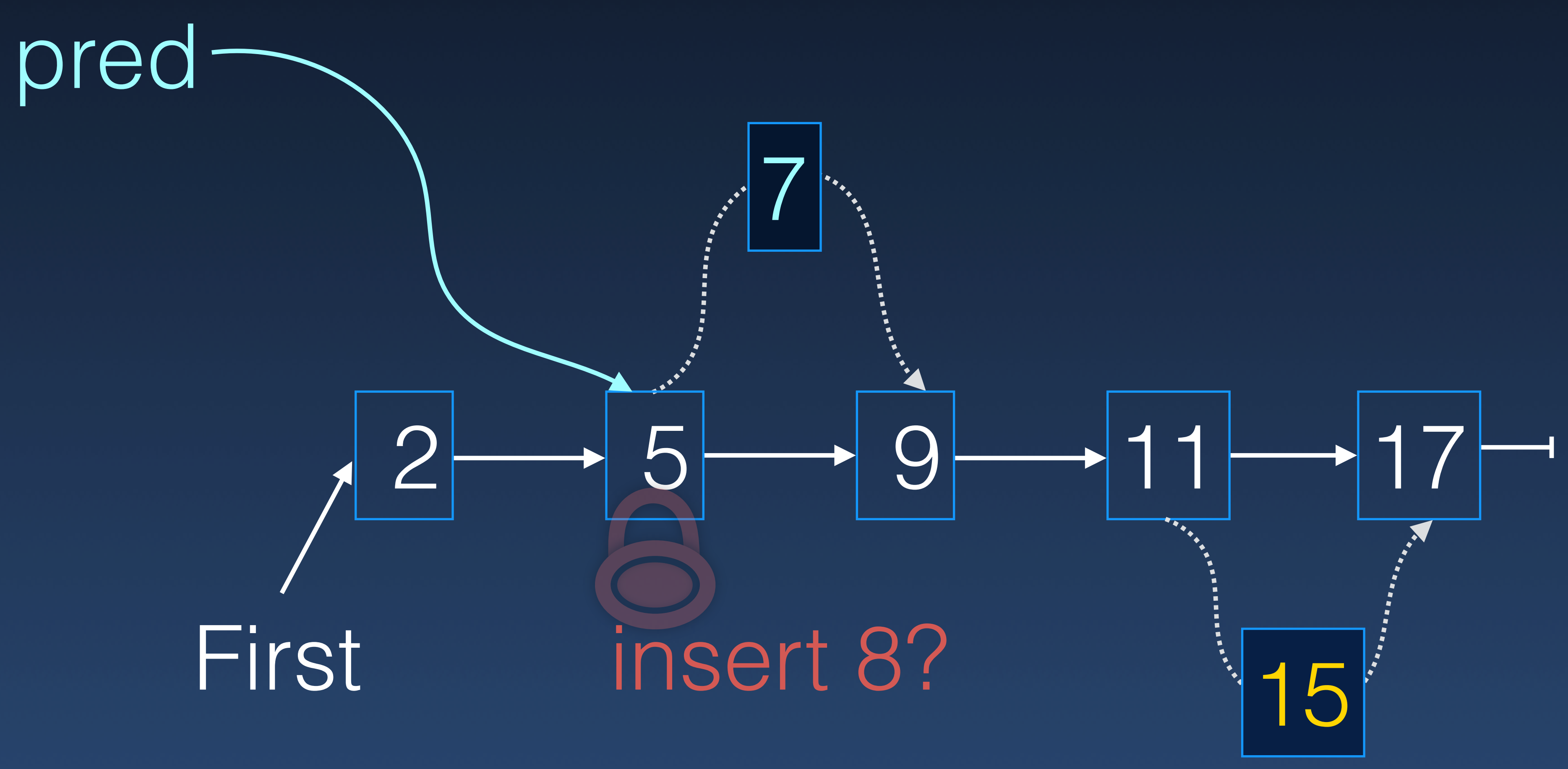
Is “9” still next to “5”?

```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

Subodh Kumar

List using Lock

- Lock “resources”
- Process
- Unlock “resources”



Insertion Loop

```
lock(pred)
if(key in [pred->key:pred->nxt->key)) {
    pred->nxt = Node(key, pred->nxt, new(Lock))
}
unlock(pred)
pred = pred->nxt
```

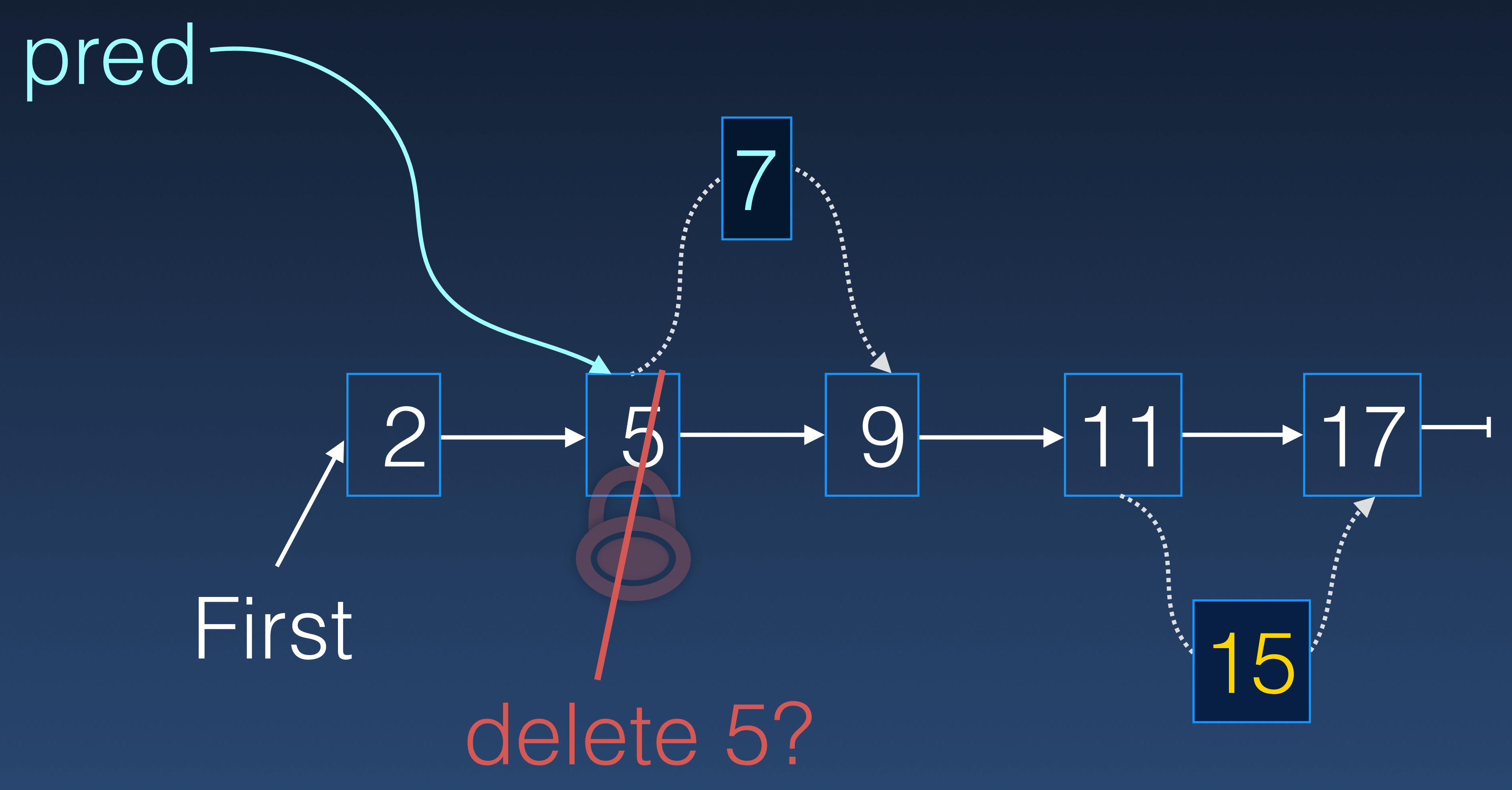
Is “9” still next to “5”?

```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

Subodh Kumar

List using Lock

- Lock “resources”
- Process
- Unlock “resources”



Insertion Loop

```
lock(pred)
if(key in [pred->key:pred->nxt->key)) {
      pred->nxt = Node(key, pred->nxt, new(Lock))
}
unlock(pred)
pred = pred->nxt
```

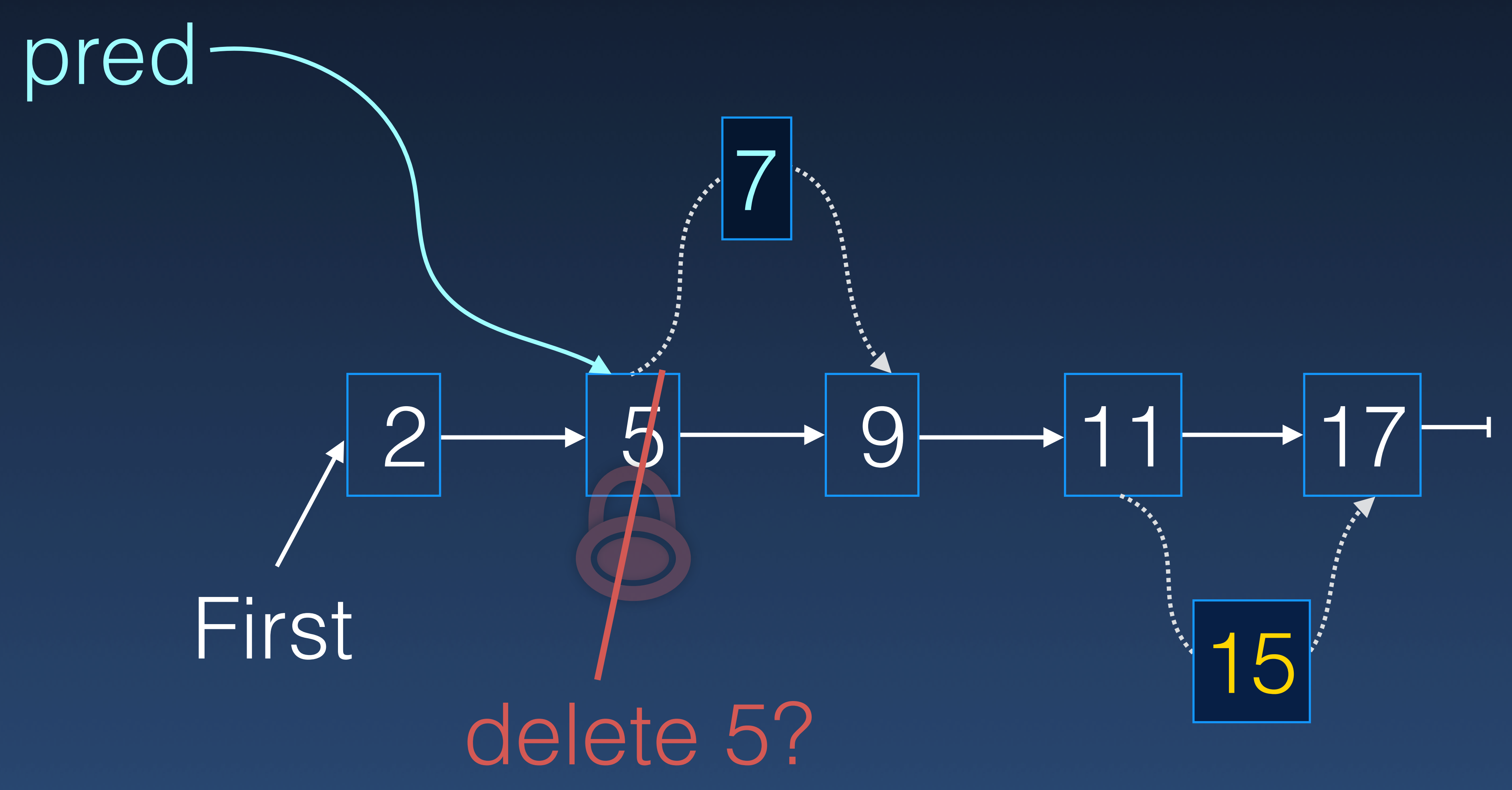
Is “9” still next to “5”?

```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

Subodh Kumar

List using Lock

- Lock “resources”
- Process
- Unlock “resources”



Insertion Loop

```
lock(pred)
if(key in [pred->key:pred->nxt->key)) {
      pred->nxt = Node(key, pred->nxt, new(Lock))
}
unlock(pred)
pred = pred->nxt
```

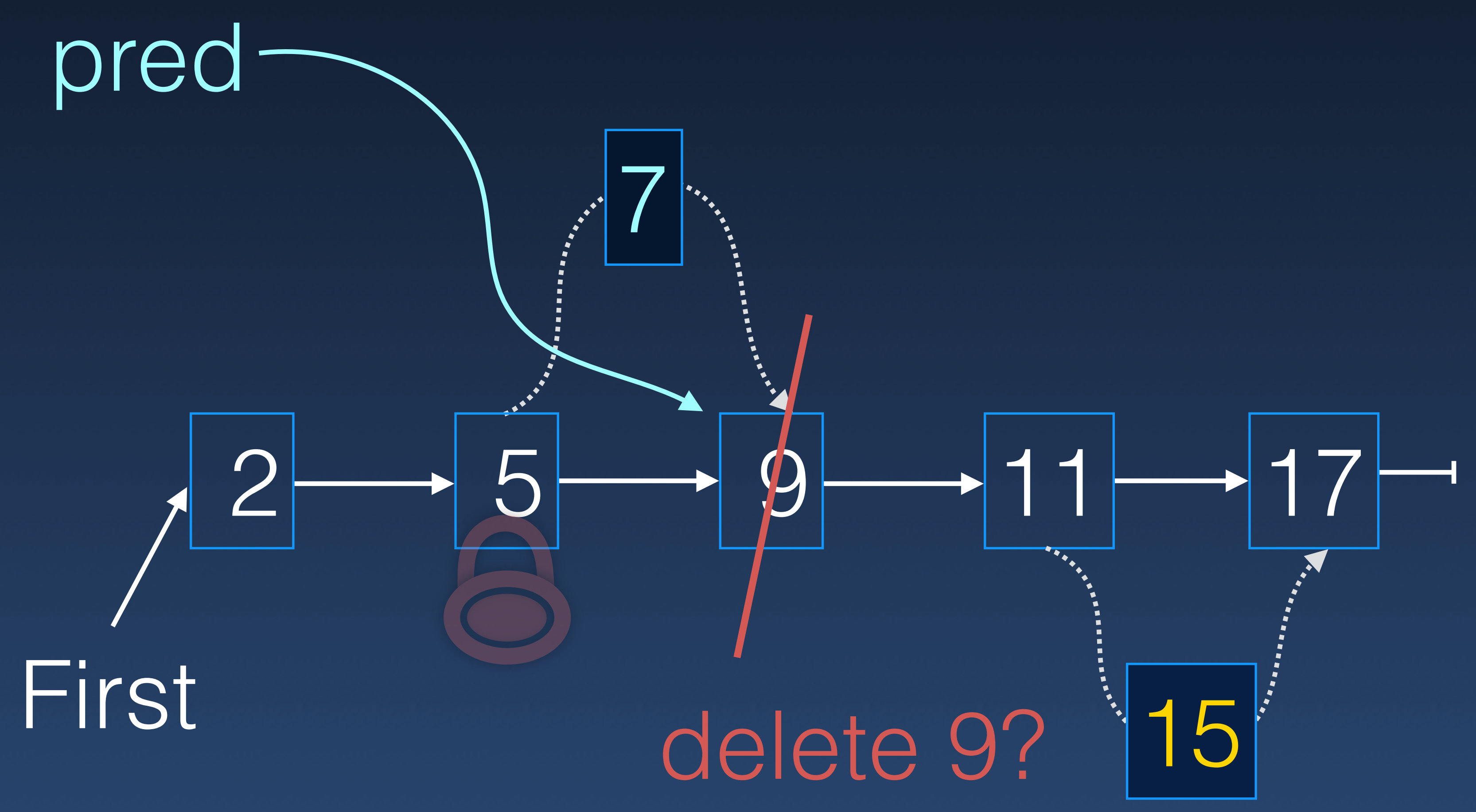
Before unlocking pred, capture ‘nxt’ locally?

```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

Subodh Kumar

List using Lock

- Lock “resources”
- Process
- Unlock “resources”



Insertion Loop

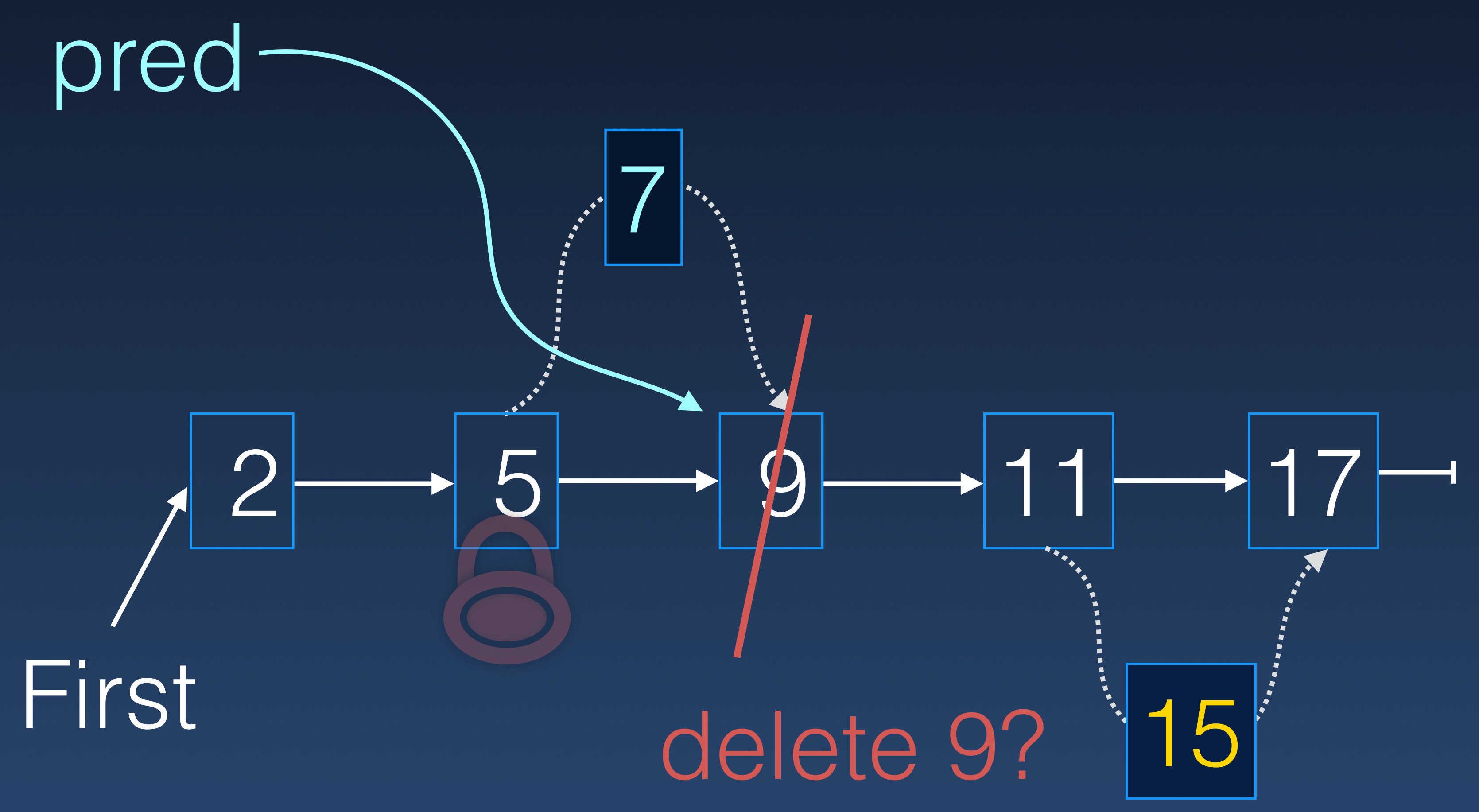
```
lock(pred)
if(key in [pred->key:pred->nxt->key)) {
      pred->nxt = Node(key, pred->nxt, new(Lock))
}
unlock(pred)
pred = pred->nxt
```

Before unlocking pred, capture ‘nxt’ locally?

```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

List using Lock

- Lock “resources”
- Process
- Unlock “resources”



Insertion Loop

```
lock(pred)
if(key in [pred->key:pred->nxt->key)) {
    pred->nxt = Node(key, pred->nxt, new(Lock))
}
unlock(pred)
pred = pred->nxt
```

Local view?

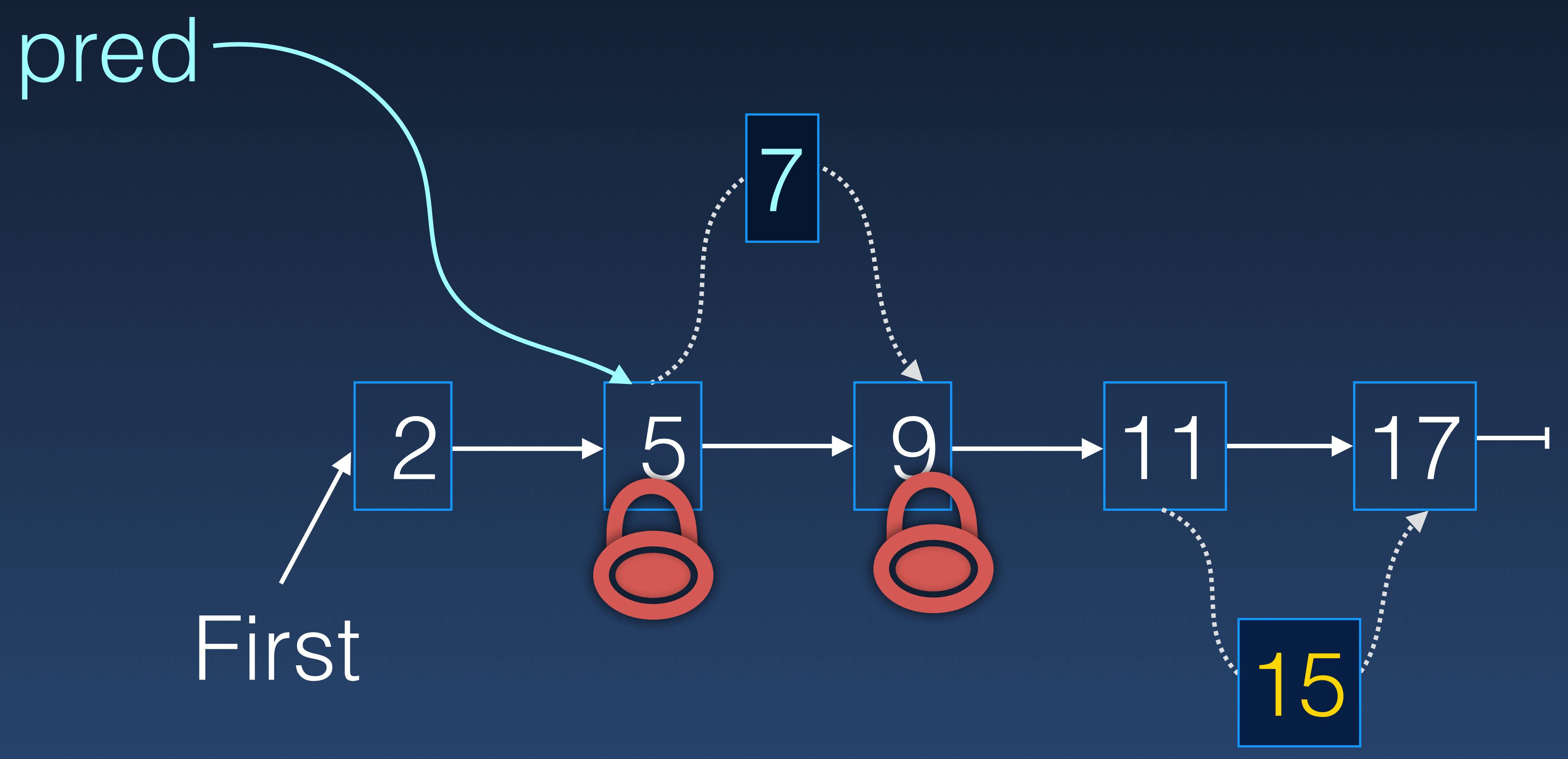
Before unlocking pred, capture ‘nxt’ locally?

```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

Subodh Kumar

List using Lock

- Lock “resources”
- Process
- Unlock “resources”



Insertion Loop

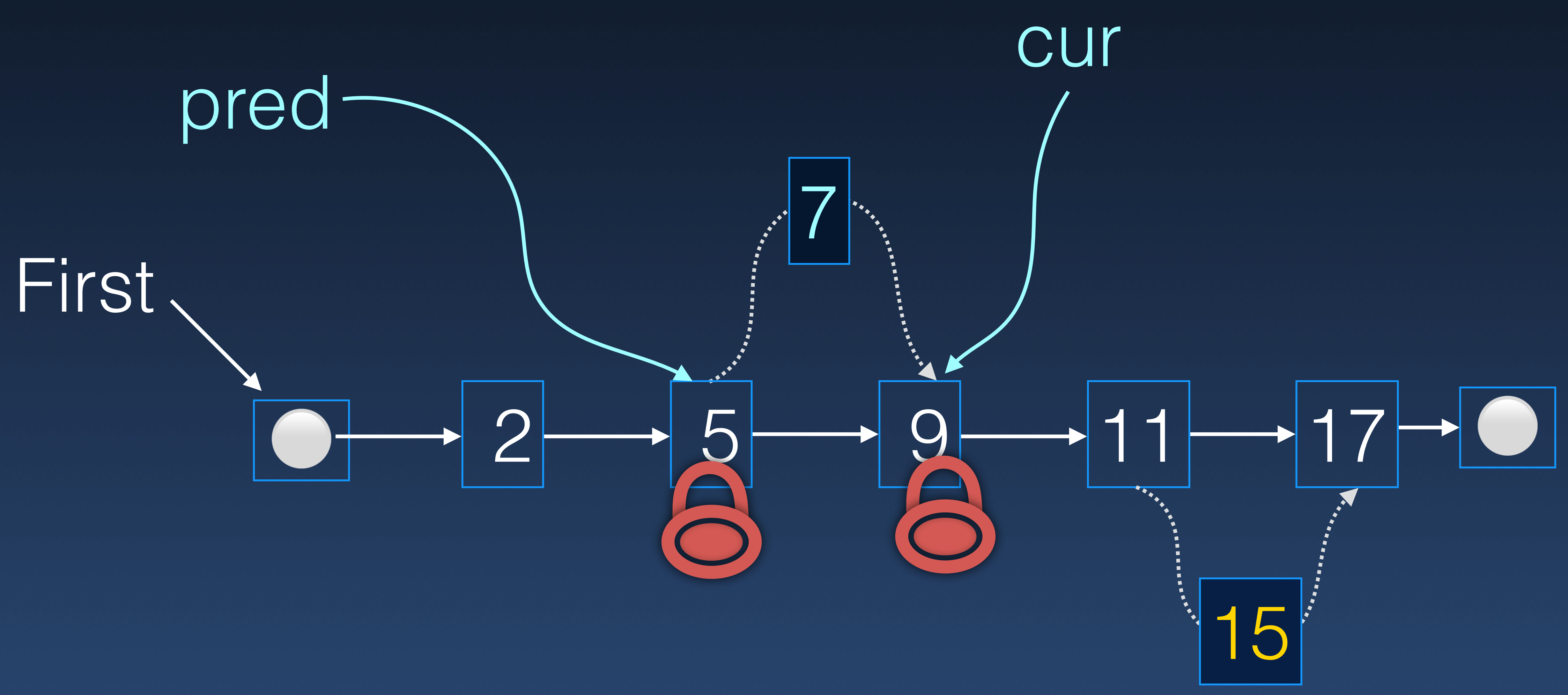
Local view?

```
lock(pred)   And lock(pred->nxt)
if(key in [pred->key:pred->nxt->key)) {
      pred->nxt = Node(key, pred->nxt, new(Lock))
}
unlock(pred)
pred = pred->nxt
```

```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

Insert

```
lock(First); lock(First->nxt)
pred = First; cur = pred->nxt
while(cur != Last && cur->key < key) {
   unlock(pred)
   pred = cur
   cur = cur->nxt
   lock(cur)
}
pred->nxt = new Node(key, cur, new lock())
unlock(pred); unlock(curr)
```

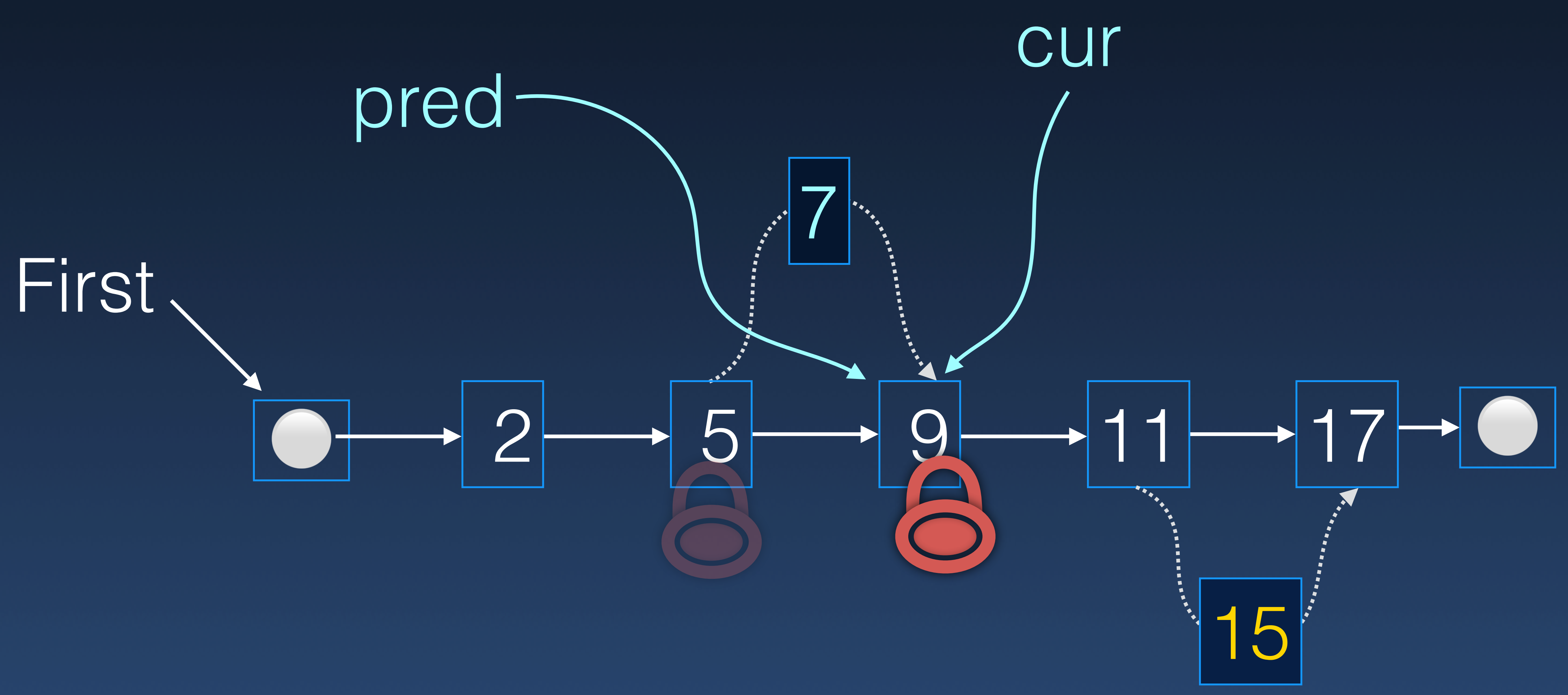


```
Node {
    Key    key
    Node nxt
    Lock  lock
}
```

Subodh Kumar

Insert

```
lock(First); lock(First->nxt)
pred = First; cur = pred->nxt
while(cur != Last && cur->key < key) {
   unlock(pred)
   pred = cur
   cur = cur->nxt
   lock(cur)
}
pred->nxt = new Node(key, cur, new lock())
unlock(pred); unlock(curr)
```

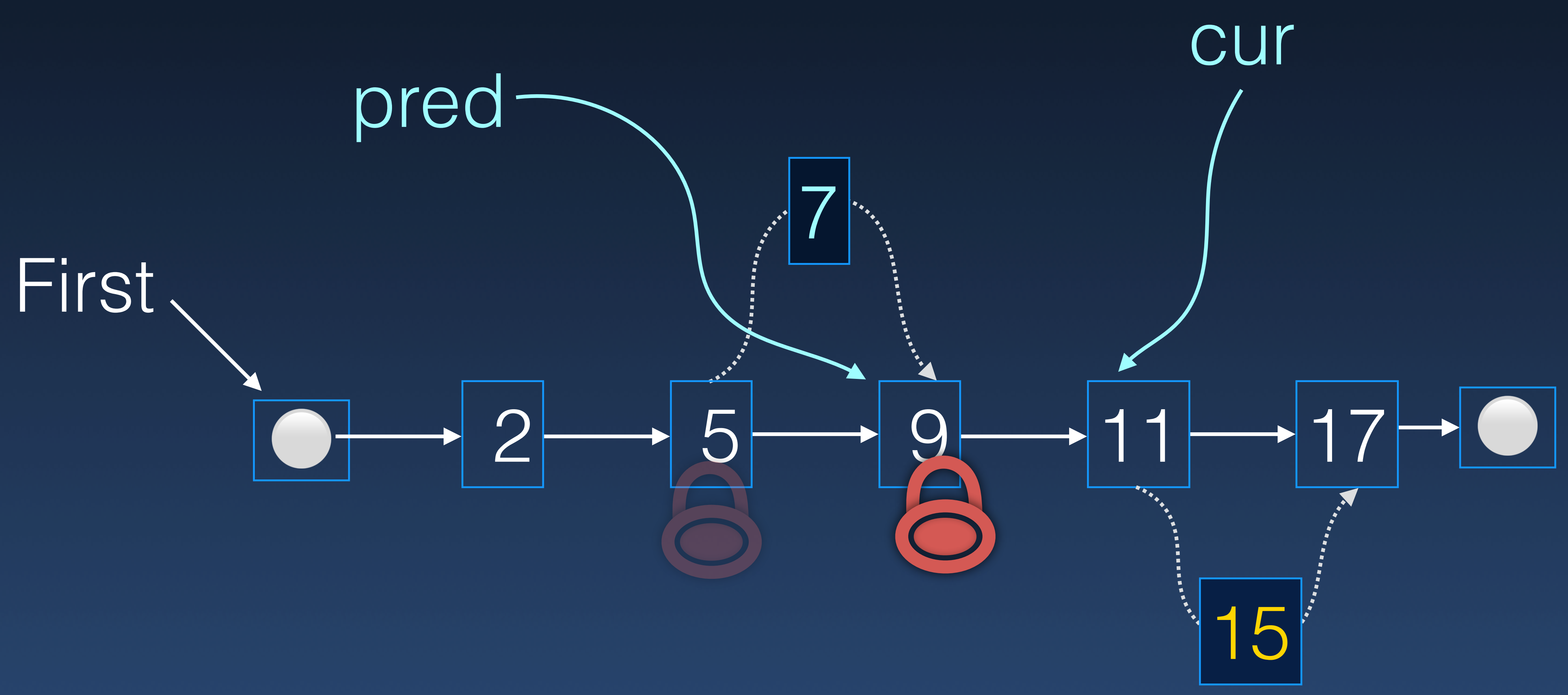


```
Node {
   Key   key
   Node nxt
   Lock  lock
}
```

Subodh Kumar

Insert

```
lock(First); lock(First->nxt)
pred = First; cur = pred->nxt
while(cur != Last && cur->key < key) {
   unlock(pred)
   pred = cur
   cur = cur->nxt
   lock(cur)
}
pred->nxt = new Node(key, cur, new lock())
unlock(pred); unlock(curr)
```

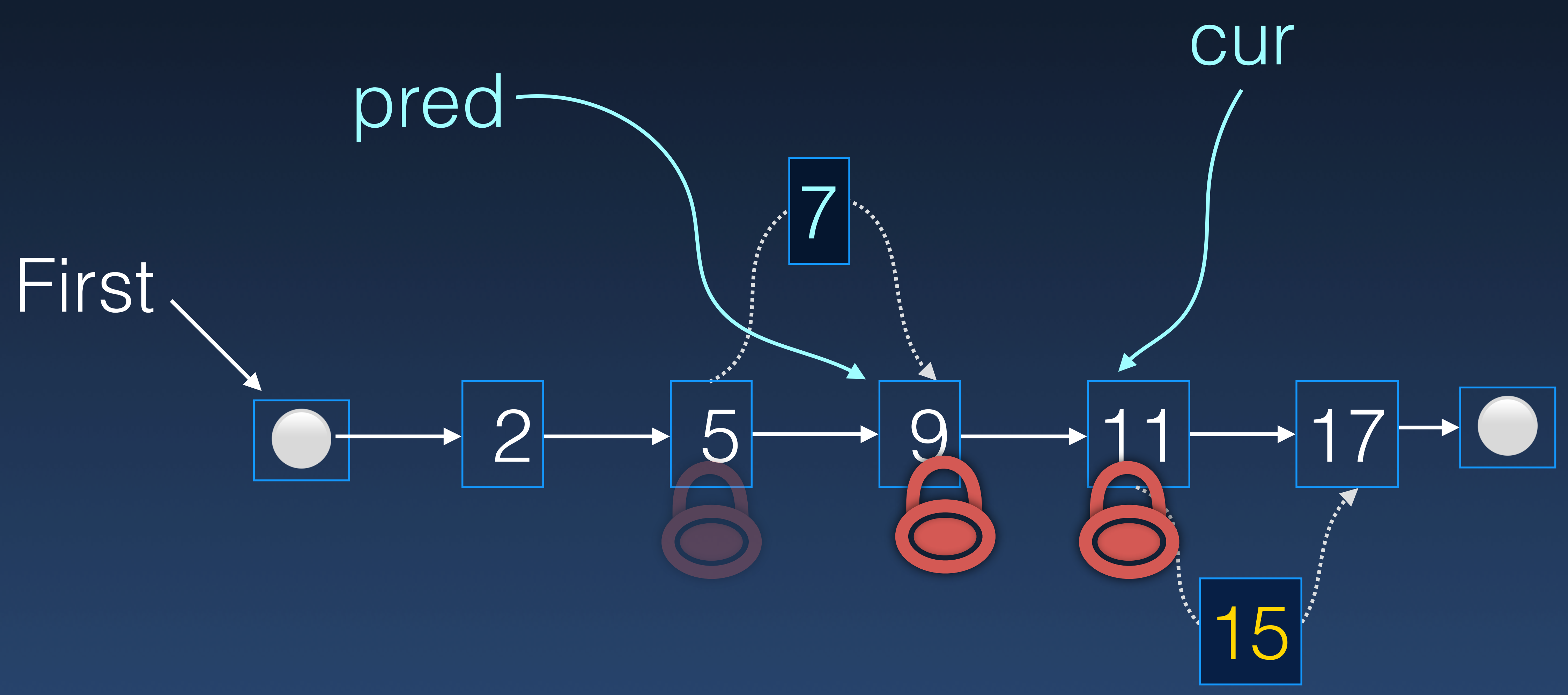


```
Node {
   Key   key
   Node nxt
   Lock  lock
}
```

Subodh Kumar

Insert

```
lock(First); lock(First->nxt)
pred = First; cur = pred->nxt
while(cur != Last && cur->key < key) {
   unlock(pred)
   pred = cur
   cur = cur->nxt
   lock(cur)
}
pred->nxt = new Node(key, cur, new lock())
unlock(pred); unlock(curr)
```

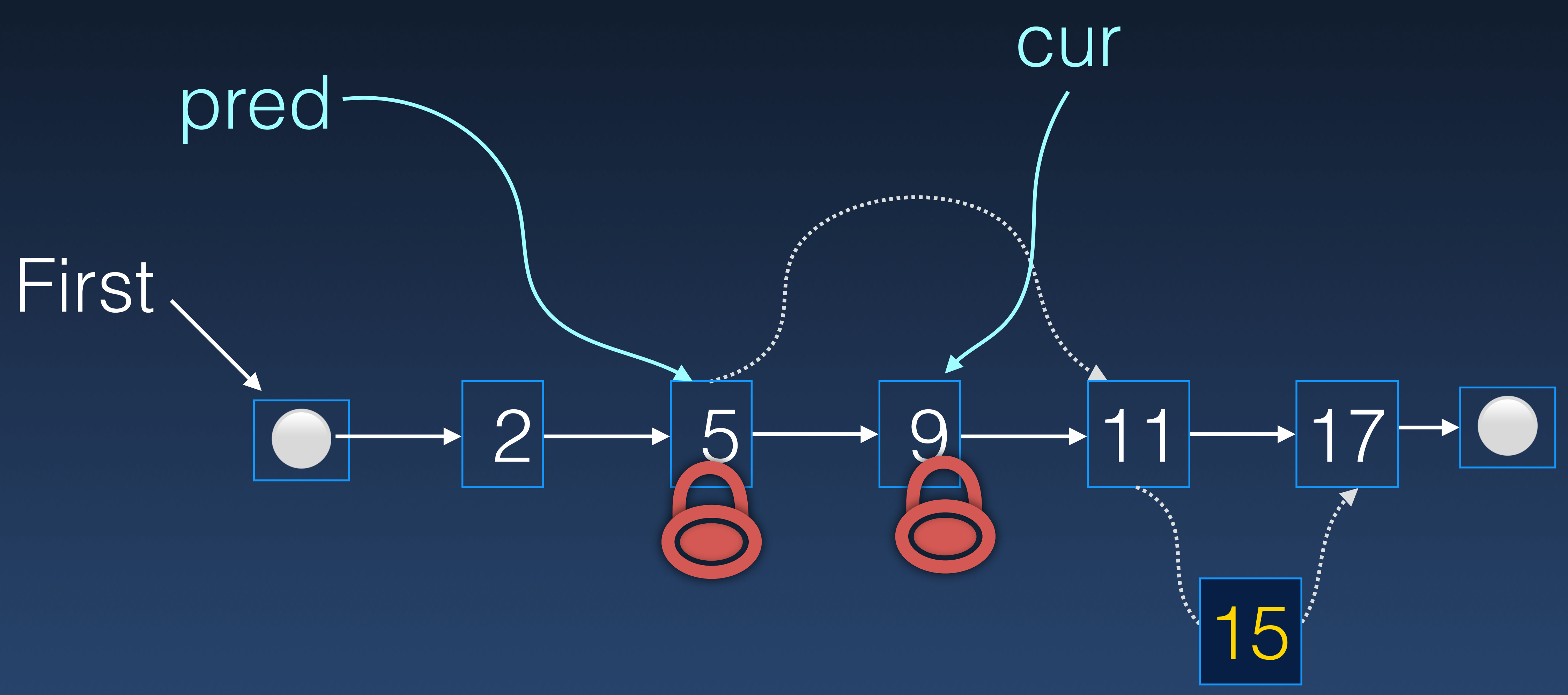


```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

Subodh Kumar

Delete

```
lock(First); lock(First->nxt)
pred = First; cur = pred->nxt
while(cur != Last && cur->key < key) {
   unlock(pred)
   pred = cur
   cur = cur->nxt
   lock(cur)
}
if(cur.key == key) {
   pred->nxt = cur->nxt
}
unlock(pred); unlock(curr)
```



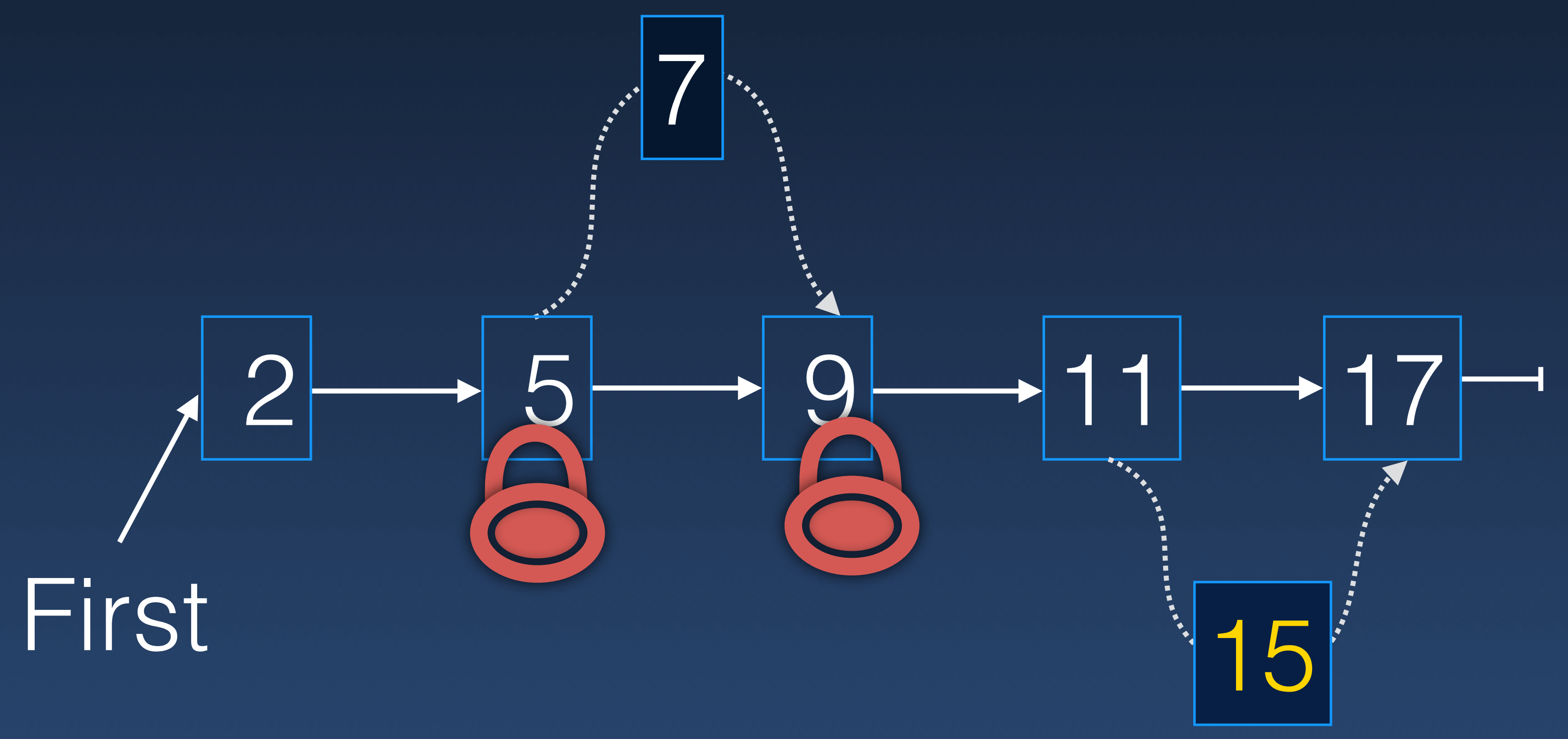
```
Node {
   Key   key
   Node nxt
   Lock  lock
}
```

Subodh Kumar

Lock

- Lock “resources”
- Process
- Unlock “resources”

➡ Correctness depends on everyone following protocol



```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

Subodh Kumar

Peterson's Mutex Algorithm

Initially: want = {false, false}

Thread 0

```
1 want[0] = true
2 turn = 1
             3             4
  while (want[1] && turn == 1);
  {// critical section
  }
  want[0] = false
```

Thread 1

```
a want[1] = true
b turn = 0
             c             d
  while (want[0] && turn == 0);
  { // critical section
  }
  want[1] = false
```

Safe?
(mutex guaranteed)

Peterson’s Mutex Algorithm

Initially: want = {false, false}

Thread 0

```
1 want[0] = true
2 turn = 1
          3          4
  while (want[1] && turn == 1);
  {// critical section
  }
  want[0] = false
```

want[1]:false
Or turn:0

Thread 1

```
a want[1] = true
b turn = 0
          c          d
  while (want[0] && turn == 0);
  { // critical section
  }
  want[1] = false
```

want[0]:false
Or turn:1

Safe?
(mutex guaranteed)

Peterson's Mutex Algorithm

Initially: want = {false, false}

Thread 0

```
1 want[0] = true
2 turn = 1
          3              4
while (want[1] && turn == 1);
{// critical section
}
want[0] = false
```

want[1]:false
Or turn:0

Thread 1

```
a want[1] = true
b turn = 0
          c              d
while (want[0] && turn == 0);
{ // critical section
}
want[1] = false
```

want[0]:false
Or turn:1

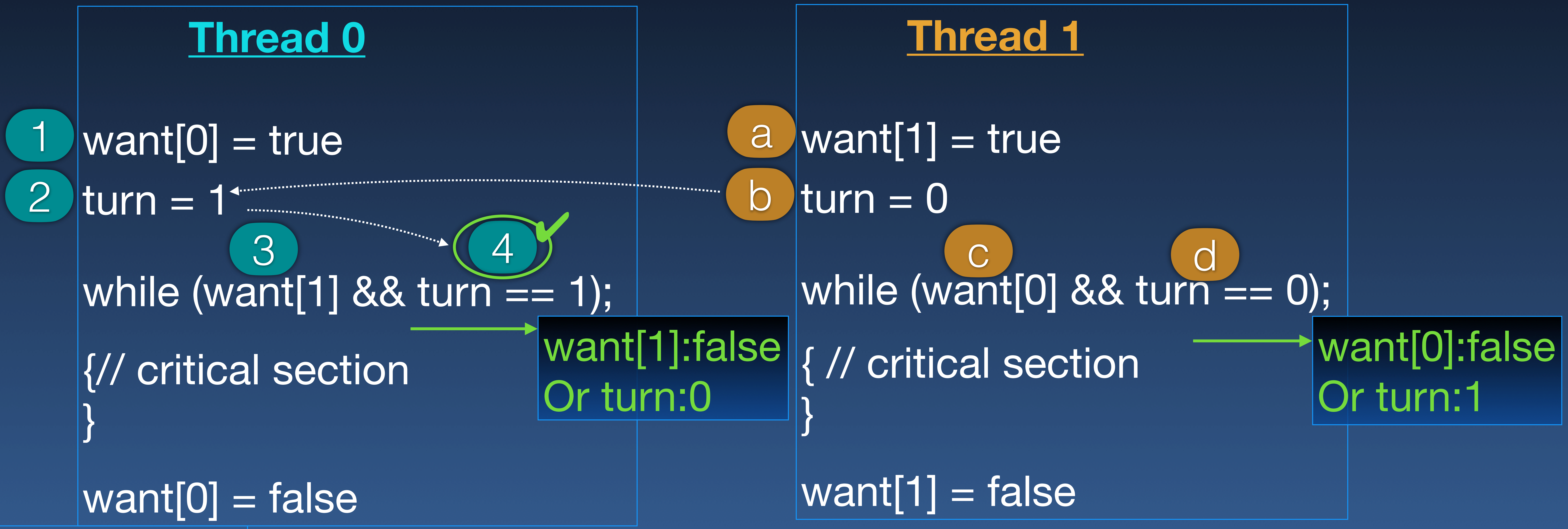
Safe?
(mutex guaranteed)

(and Th 0 exits loop)

Suppose: b → 2

Peterson's Mutex Algorithm

Initially: want = {false, false}



Thread 0

```
want[0] = true
turn = 1
while (want[1] && turn == 1);
{// critical section
}
want[0] = false
```

Thread 1

```
want[1] = true
turn = 0
while (want[0] && turn == 0);
{ // critical section
}
want[1] = false
```

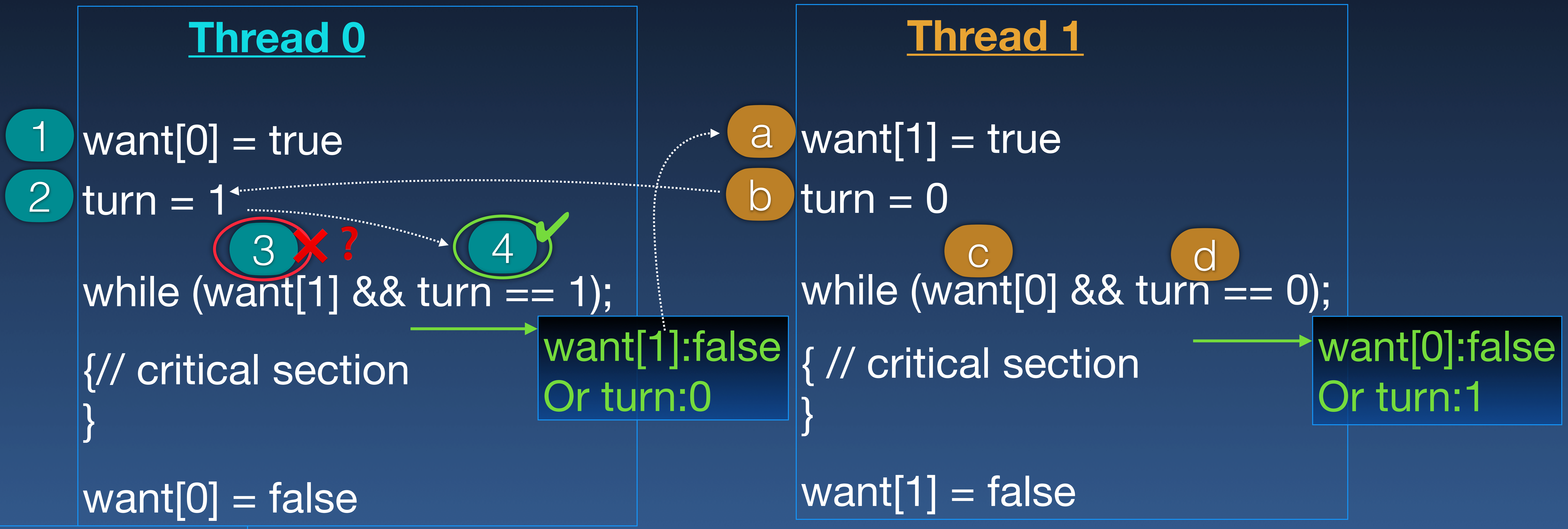
Safe?
(mutex guaranteed)

(and Th 0 exits loop)

Suppose: b → 2

Peterson’s Mutex Algorithm

Initially: want = {false, false}



Thread 0

```
1 want[0] = true
2 turn = 1
  3 ✗ ?   4 ✓
  while (want[1] && turn == 1);
  {// critical section
  }
  want[0] = false
```

want[1]:false
Or turn:0

Thread 1

```
a want[1] = true
b turn = 0
  c   d
  while (want[0] && turn == 0);
  { // critical section
  }
  want[1] = false
```

want[0]:false
Or turn:1

Safe?
(mutex guaranteed)

(and Th 0 exits loop)

Suppose: b → 2 → 3 → a

Subodh Kumar

Peterson’s Mutex Algorithm

Initially: want = {false, false}

Thread 0

```
1  want[0] = true
2  turn = 1
       3 ✗ ?          4 ✓
   while (want[1] && turn == 1);  →  want[1]:false
                                     Or turn:0
   {// critical section
   }
   want[0] = false
```

Thread 1

```
a  want[1] = true
b  turn = 0
         c              d
   while (want[0] && turn == 0);  →  want[0]:false
                                     Or turn:1
   { // critical section
   }
   want[1] = false
```

Safe?
(mutex guaranteed)

(and Th 0 exits loop)

Suppose: b → 2 → 3 → a

Subodh Kumar

Mutex w/Registers

- Mutual exclusion does not require hardware synchronization

- Peterson and Bakery use minimal number of registers

Lamport's Bakery Algorithm

```
— Not Critical Section —
want [ID] = 1;
token[ID] = 1 + max(token)
want[ID] = 0;
for other != ID {
        while(want([other] == 1);
        while(token[other] > 0 && (token[other]#other) < (token[ID]#ID);
}
— Critical Section —
token[ID] = 0
```

concurrent

concurrent

Mutex w/Registers

- Mutual exclusion does not require hardware synchronization
- Peterson and Bakery use minimal number of registers

Lamport's Bakery Algorithm

```
— Not Critical Section —
1: want [ID] = 1;
2: token[ID] = 1 + max(token)
3: want[ID] = 0;
4: for other != ID {
5:      while(want([other] == 1);
6:      while(token[other] > 0 && (token[other]#other) < (token[ID]#ID);
7: }
— Critical Section —
8: token[ID] = 0
```

concurrent

concurrent

@#6, if Th.i saw token[i] < token[j], can Th.j see token[j] < token[i] (with i still in critical section)?

If token[j] changes, it may only increase

Mutex w/Registers

- Mutual exclusion does not require hardware synchronization
- Peterson and Bakery use minimal number of registers
  - ➡ Too many variables? (and ever increasing counter values)

Lamport's Bakery Algorithm

```
— Not Critical Section —
1: want [ID] = 1;
2: token[ID] = 1 + max(token)
3: want[ID] = 0;
4: for other != ID {
5:       while(want([other] == 1);
6:       while(token[other] > 0 && (token[other]#other) < (token[ID]#ID);
7: }
— Critical Section —
8: token[ID] = 0
```

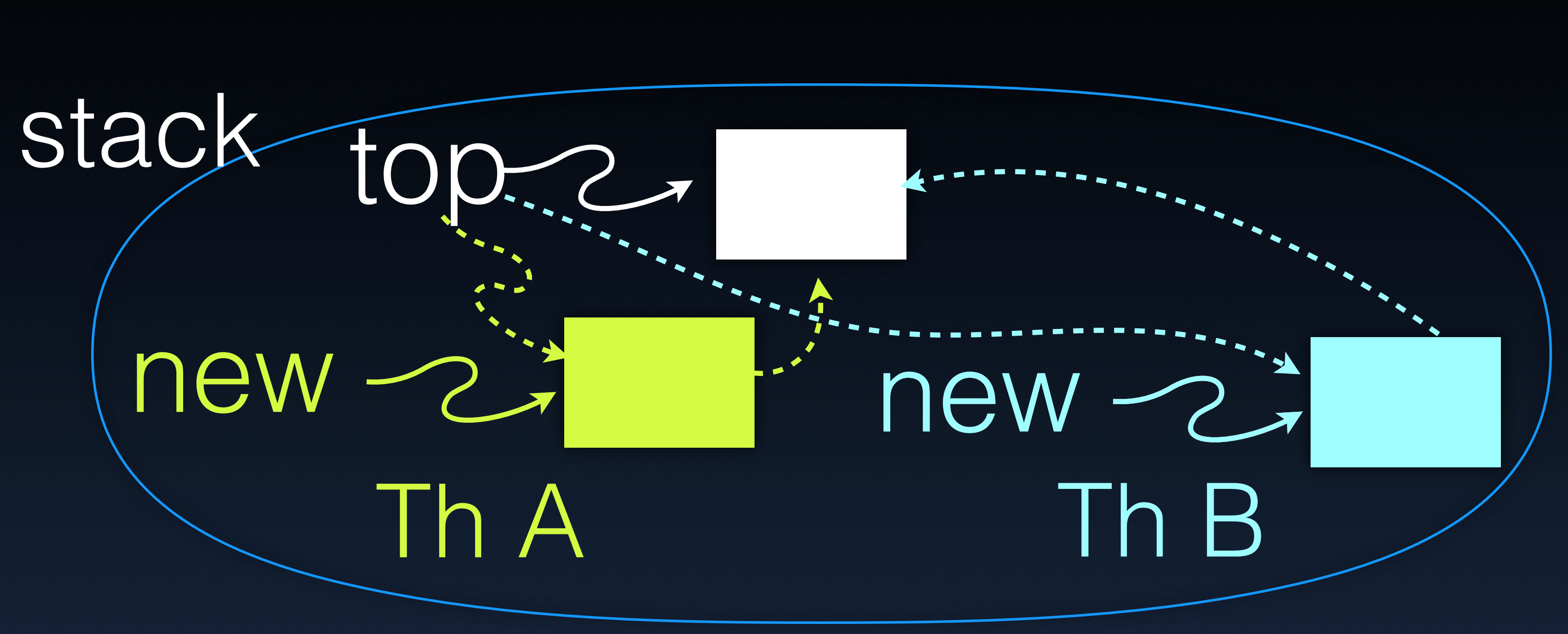
concurrent

wait if anyone in the midst of taking token

concurrent

Th.i waits @#5 for Th.j to complete writing its token @#2, so that in case j < i, and both i and j get the same value, Th.i does not proceed to #6, where it may incorrectly win if write to token[j] is delayed.

✔Lock-free? Wait-free?



```
std::atomic<node<T>*> top;
…
void push(const T& data) {
    node<T>* new_node = new node<T>(data);

    // put the current value of top into new_node->next
    do new_node->next = top.load();

    // make new_node the top, as long as top still equals new_node->next
     while(!top.compare_exchange_strong(new_node->next, new_node));
}
```

Non-shared Logical Clock

- Each entity maintains a <u>counter</u>
  - ➡ increments every *step*, at its own pace
- Interaction between entities is through messages
  - ➡ Data + <u>counter</u>
- On message receipt:
  - ➡ If recipient <u>counter</u> < received <u>count</u>
    - ▶ Increase local <u>counter</u> to received <u>count</u>
    - ▶ Receive is also a '*step*,' so increment by one

[Lamport's Timestamp algorithm]

Distributed Mutex

type time

Request Critical Section:

Broadcast **R** = <request, local-time>

Add **R** to local-queue

Enter Critical section (**R**)

if **R**.time has the lowest time value in local-queue .AND.

Have received some **m**essage from every other thread with **m**.time > **R**.time

---

Exit Critical section (**R**):

Remove **R** from local-queue

Broadcast <release> message to all

Distributed Mutex

type time

```
Request Critical Section:
    Broadcast R = <request, local-time>
    Add R to local-queue

Enter Critical section (R)
    if R.time has the lowest time value in local-queue .AND.
    Have received some message from every other thread with m.time > R.time

Exit Critical section (R):
    Remove R from local-queue
    Broadcast <release> message to all
```

```
Receive R
    update(local-time)
    if(R.type == request)
        Add R to local-queue
        Reply <ack, local-time>
    if(R.type == release)
        Remove R from local-queue
```

Subodh Kumar

Distributed Mutex

type time

```
Request Critical Section:
    Broadcast R = <request, local-time>
    Add R to local-queue
```

if a request was made by another thread with time < **m**.time, it must have been received before **m**

```
Enter Critical section (R)
    if R.time has the lowest time value in local-queue .AND.
    Have received some message from every other thread with m.time > R.time
```

```
Exit Critical section (R):
    Remove R from local-queue
    Broadcast <release> message to all
```

```
Receive R
    update(local-time)
    if(R.type == request)
        Add R to local-queue
        Reply <ack, local-time>
    if(R.type == release)
        Remove R from local-queue
```

Synchronization Review

- Synchronization primitives
  - ➡ Memory fences and consistency
  - ➡ Lock/Mutex, Condition variables, Atomics, Test & Set, Fetch & Add, Compare & Swap, Critical section, Barrier, Wait, Ordered
  - ➡ Registers, Logical clocks, Peterson’s algorithm, Bakery algorithm, Distributed mutex
- Properties of Synchronization
  - ➡ Safety, Progress, Liveness
  - ➡ Blocking/Non-blocking, Starvation-free, Deadlock-free, Lockfree, Waitfree