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)

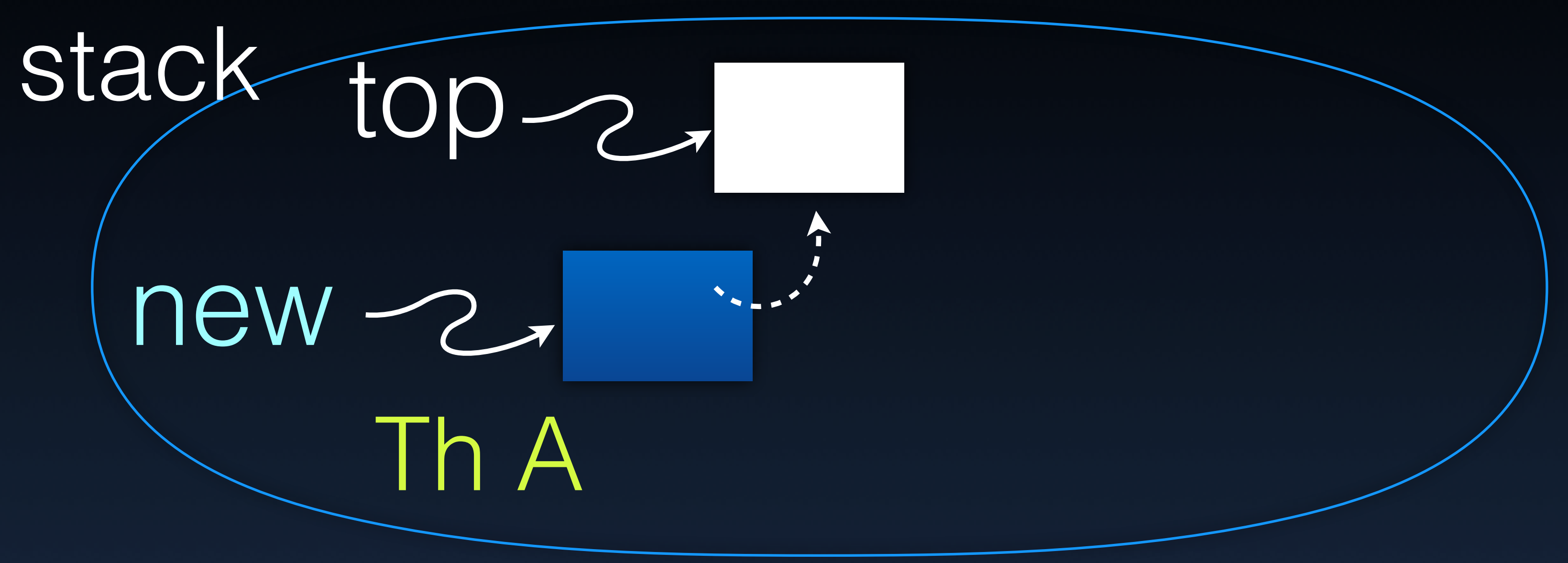
```
std::atomic<int> var(0);

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
  // }
```

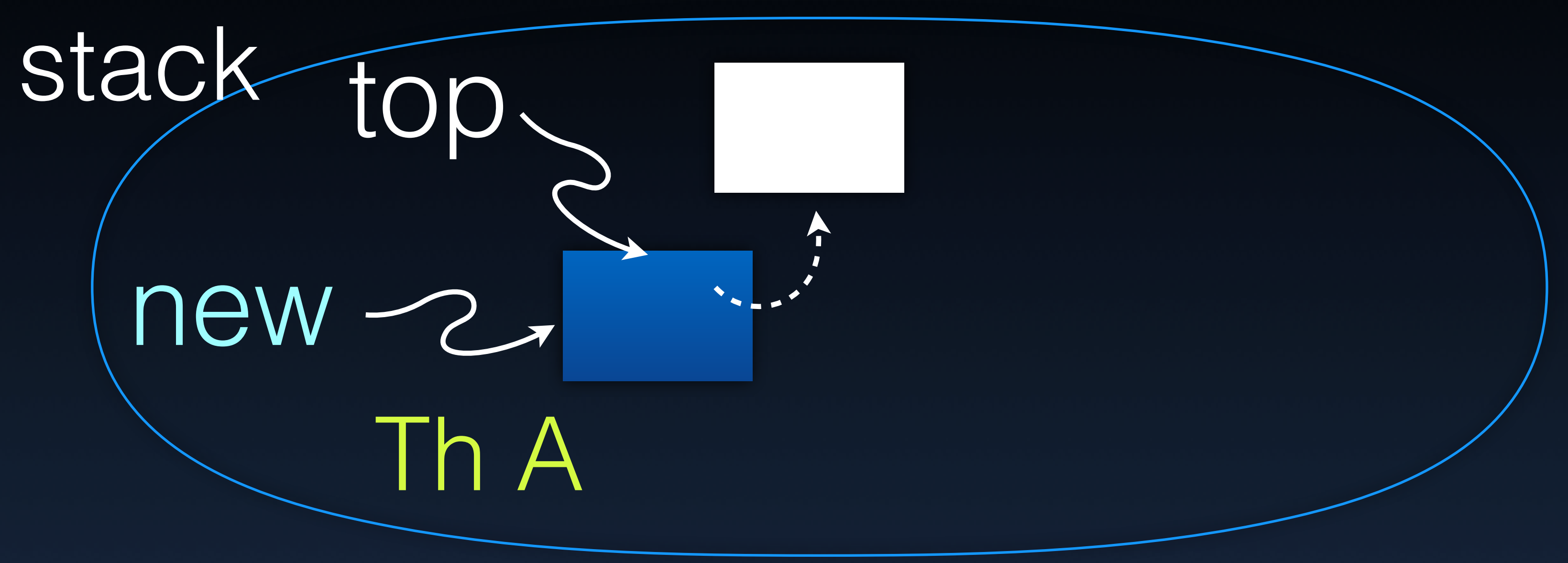
```
#pragma atomic
    var++;
```

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

Subodh Kumar

Compare & Exchange

(aka Compare & Swap)



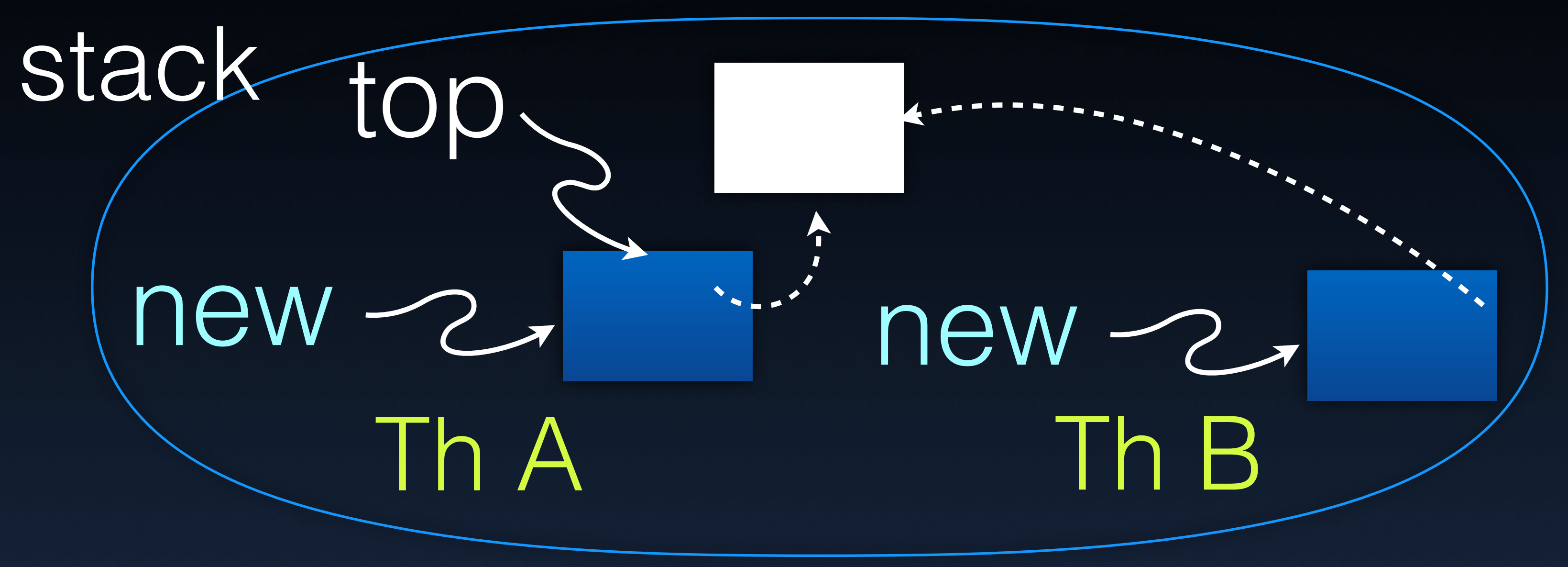
```
std::atomic<int> var(0);

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
 // }
```

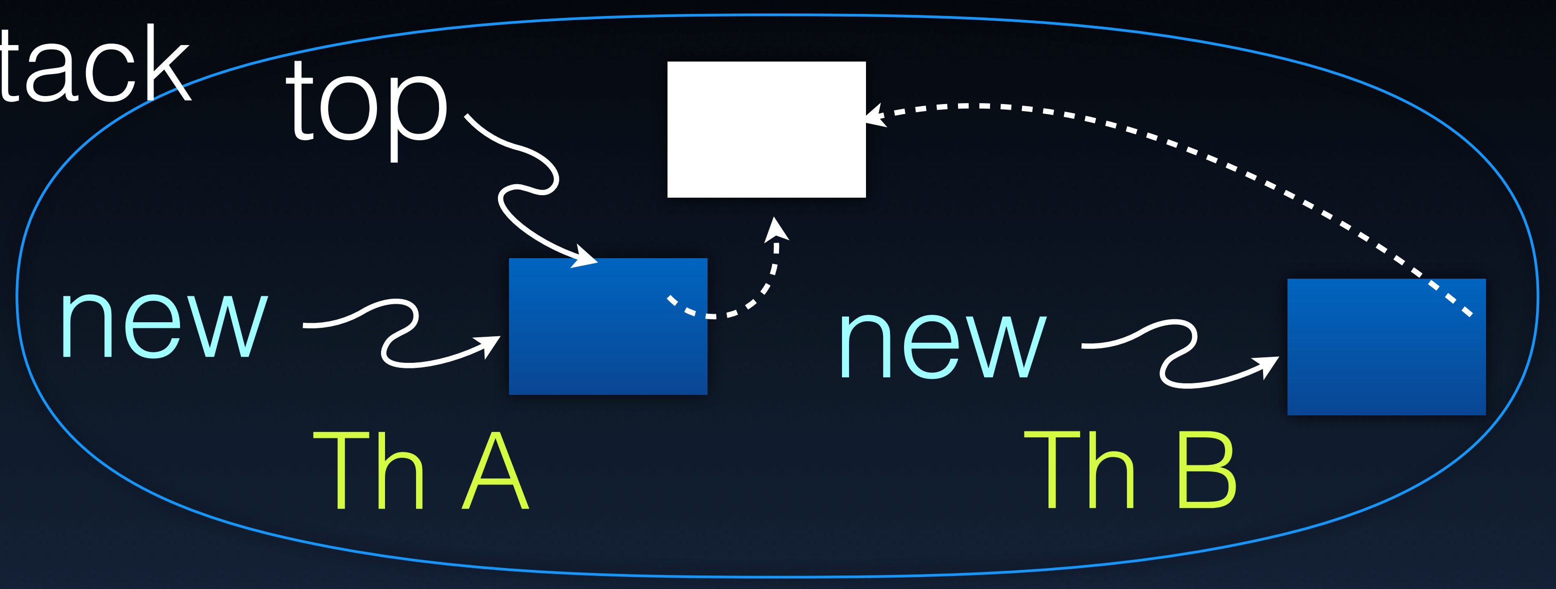
```
#pragma atomic
     var++;
```

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

Subodh Kumar

Compare & Exchange

(aka Compare & Swap)



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

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

```
#pragma atomic
```

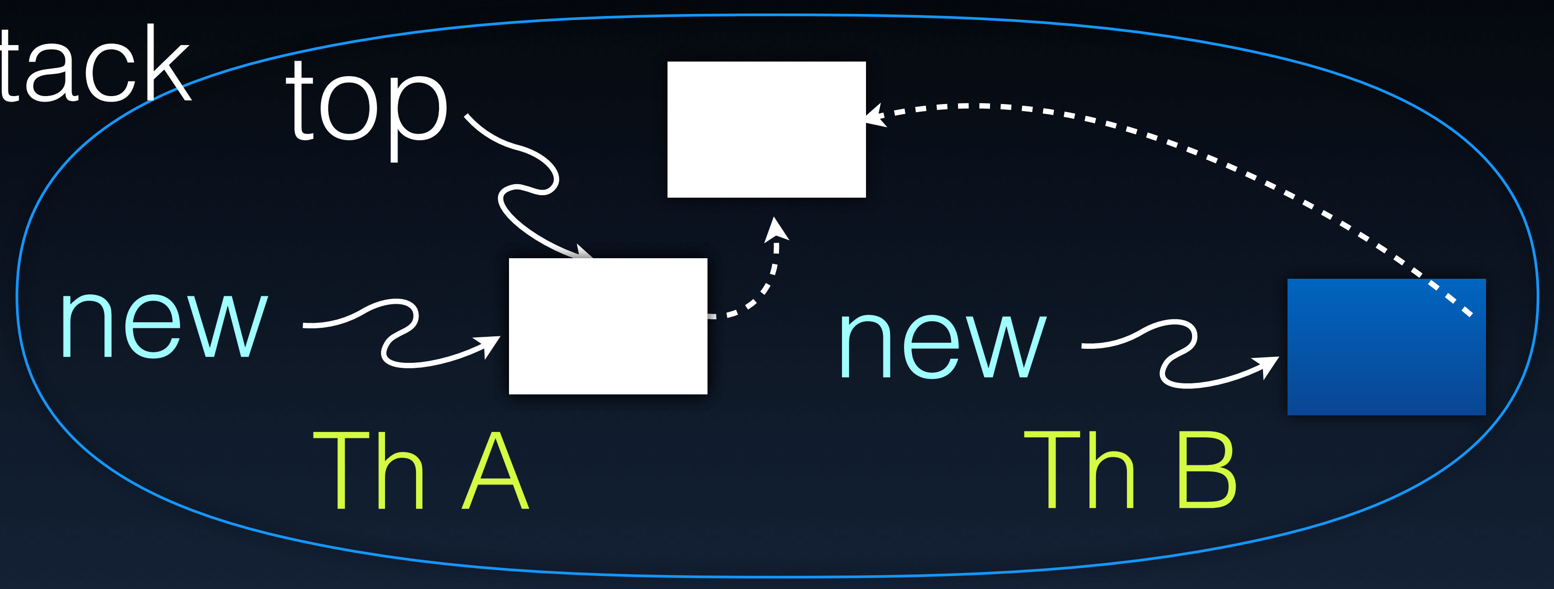
are

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

(aka Compare & Swap)



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

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

#pragma atomic

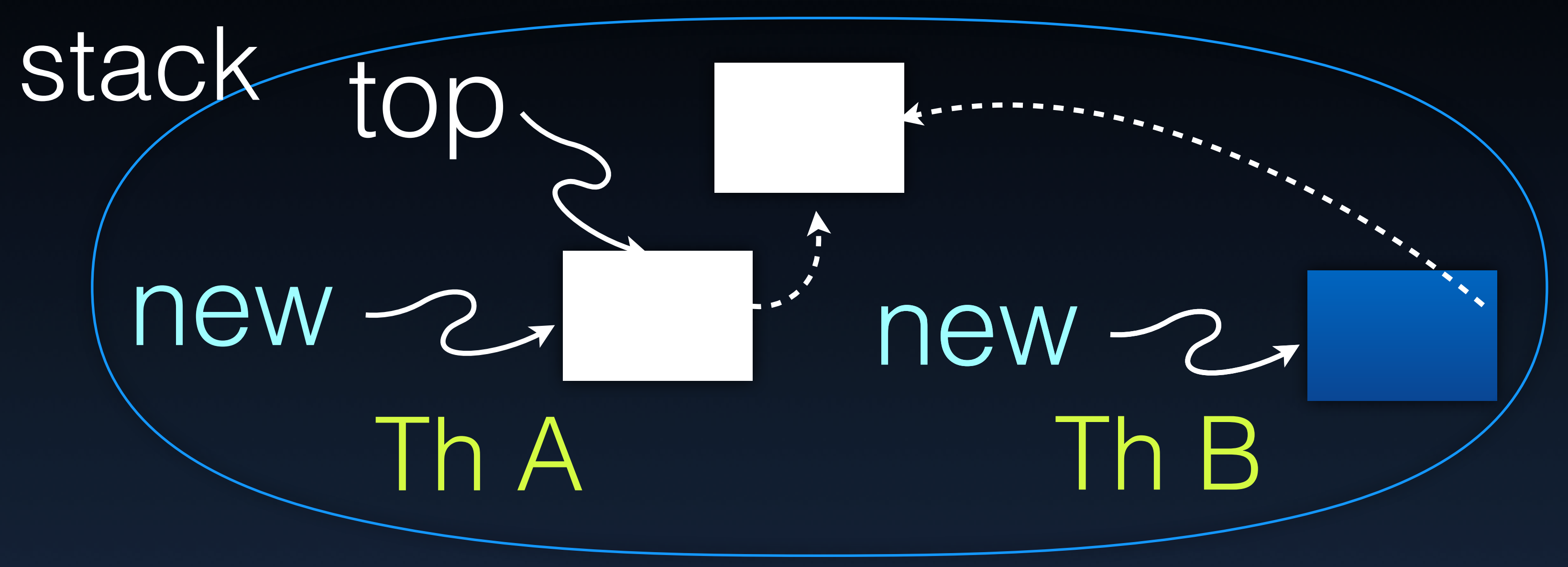
are

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



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

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

#pragma atomic

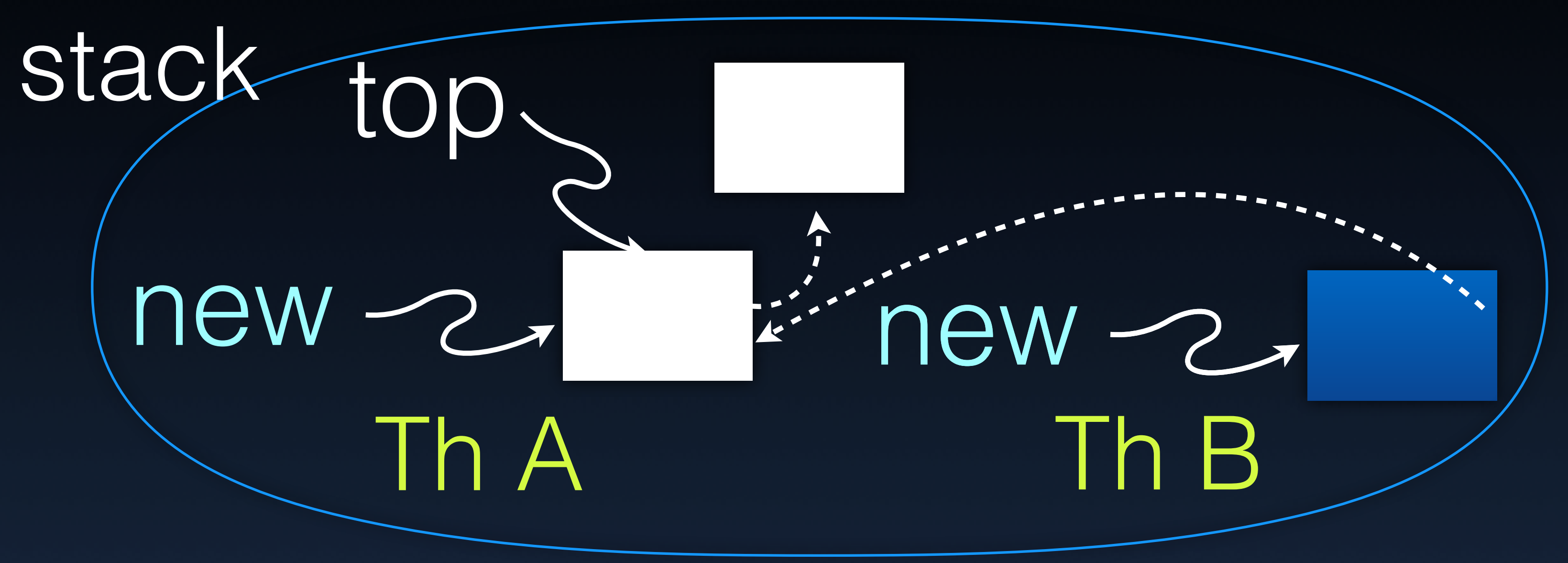
are

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));
}
```

Subodh Kumar

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--;
   }
}
```

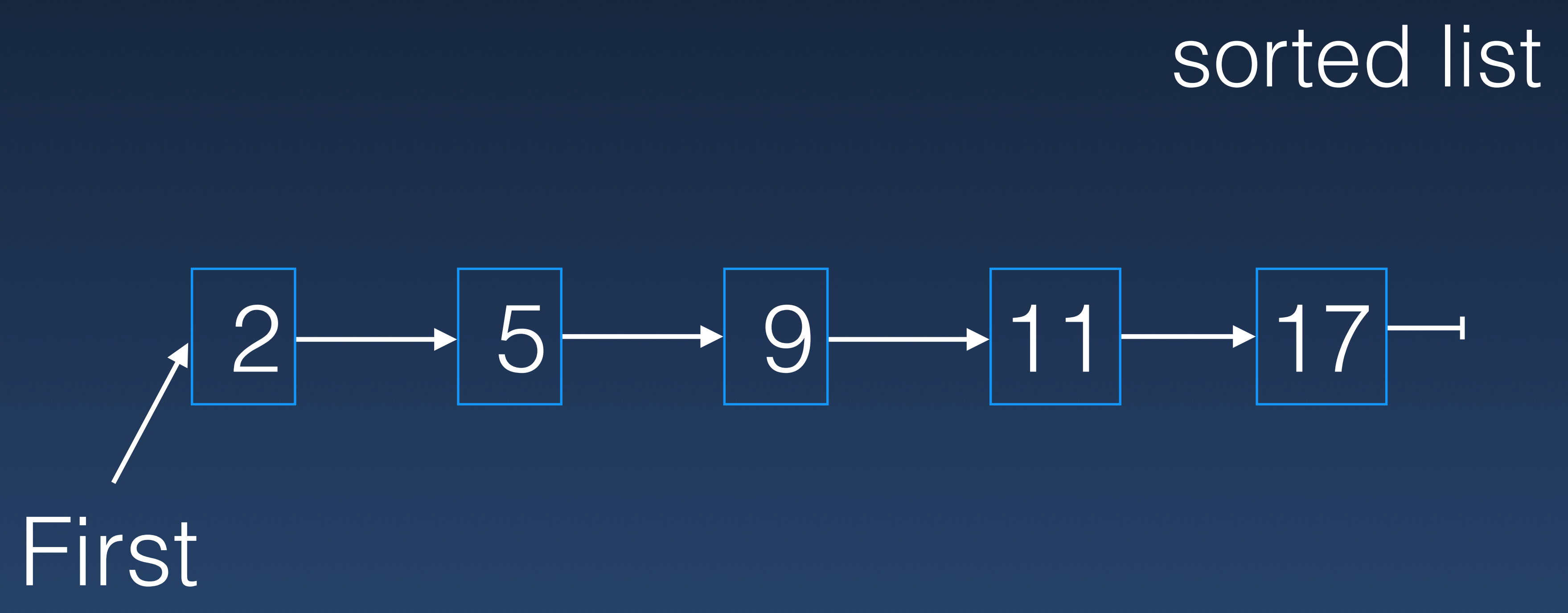
```
   std::atomic<int> counter{0};
```

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

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

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

Subodh Kumar

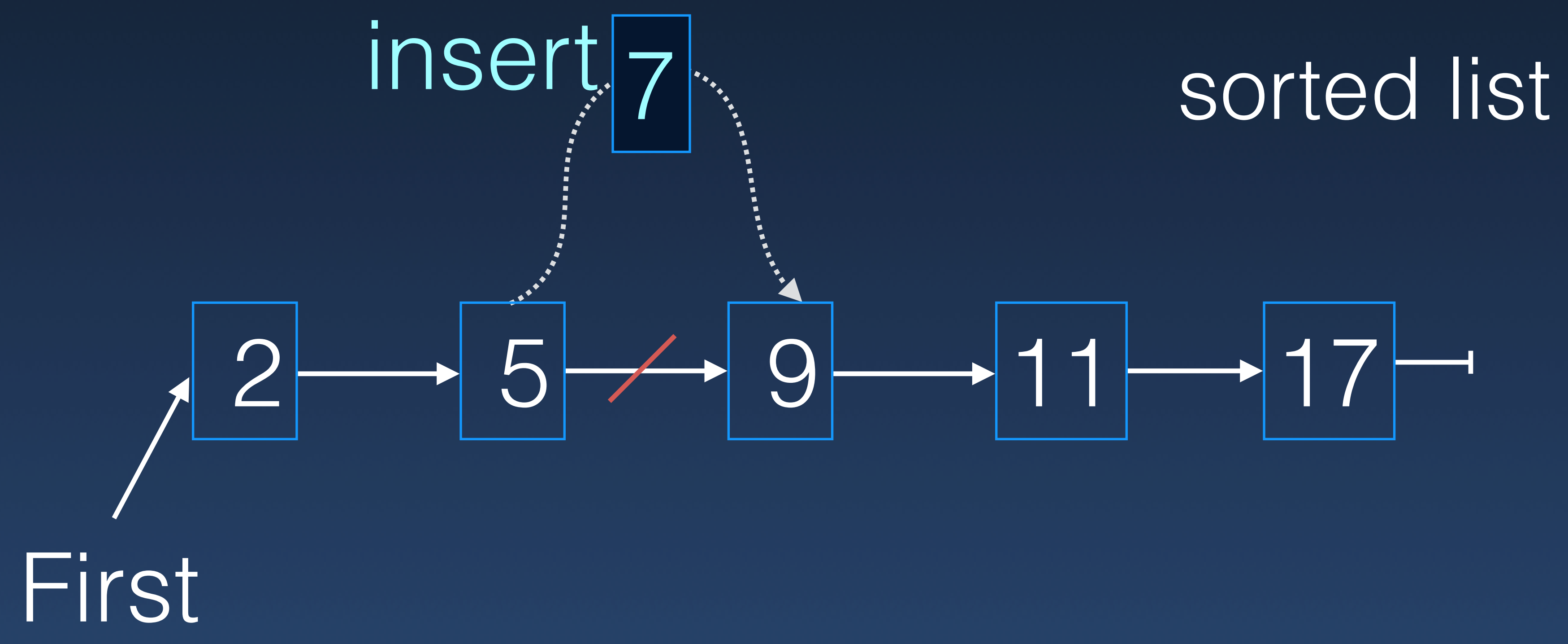
Lock

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



Lock

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

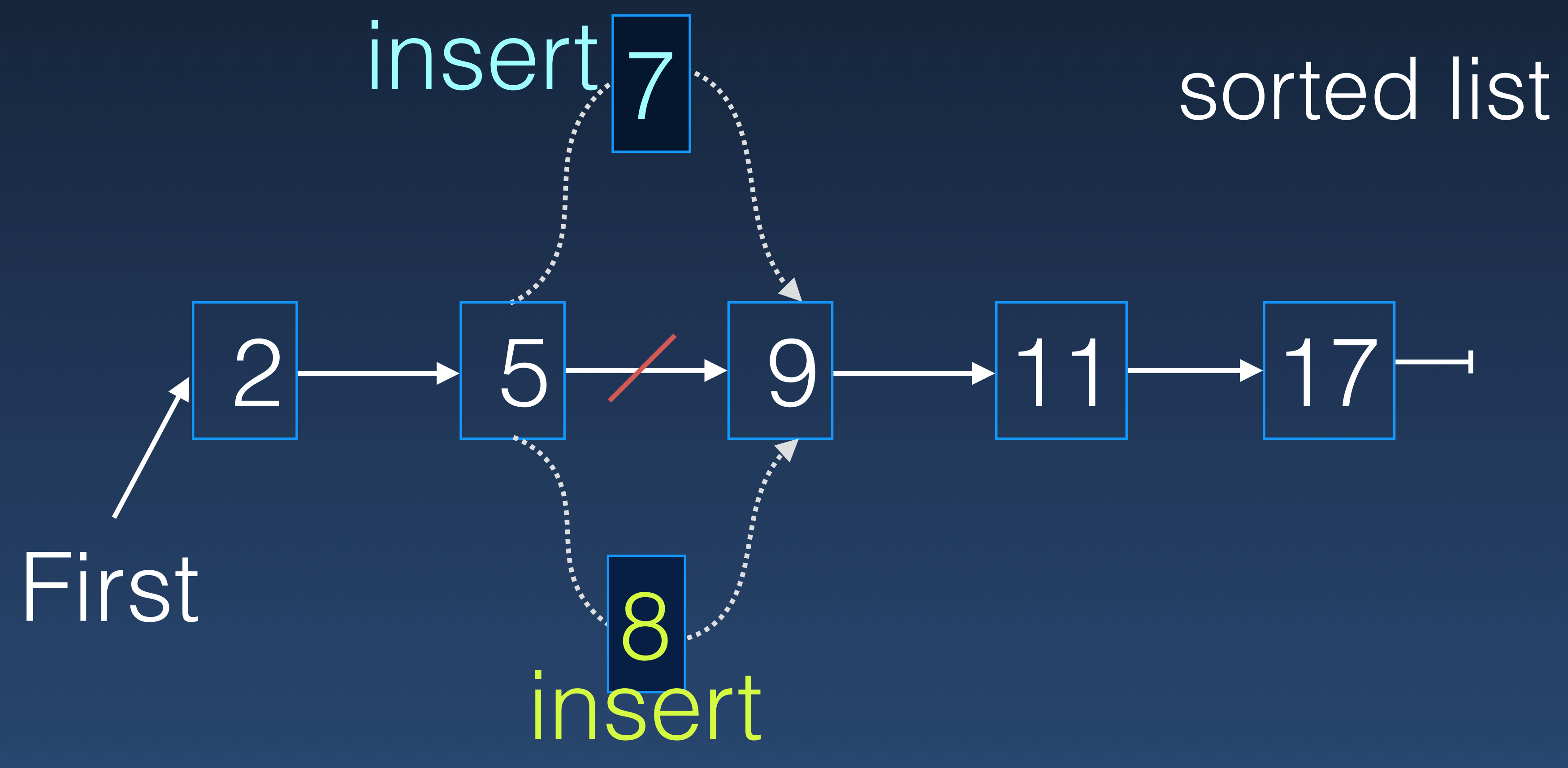


Lock

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

Correctness?

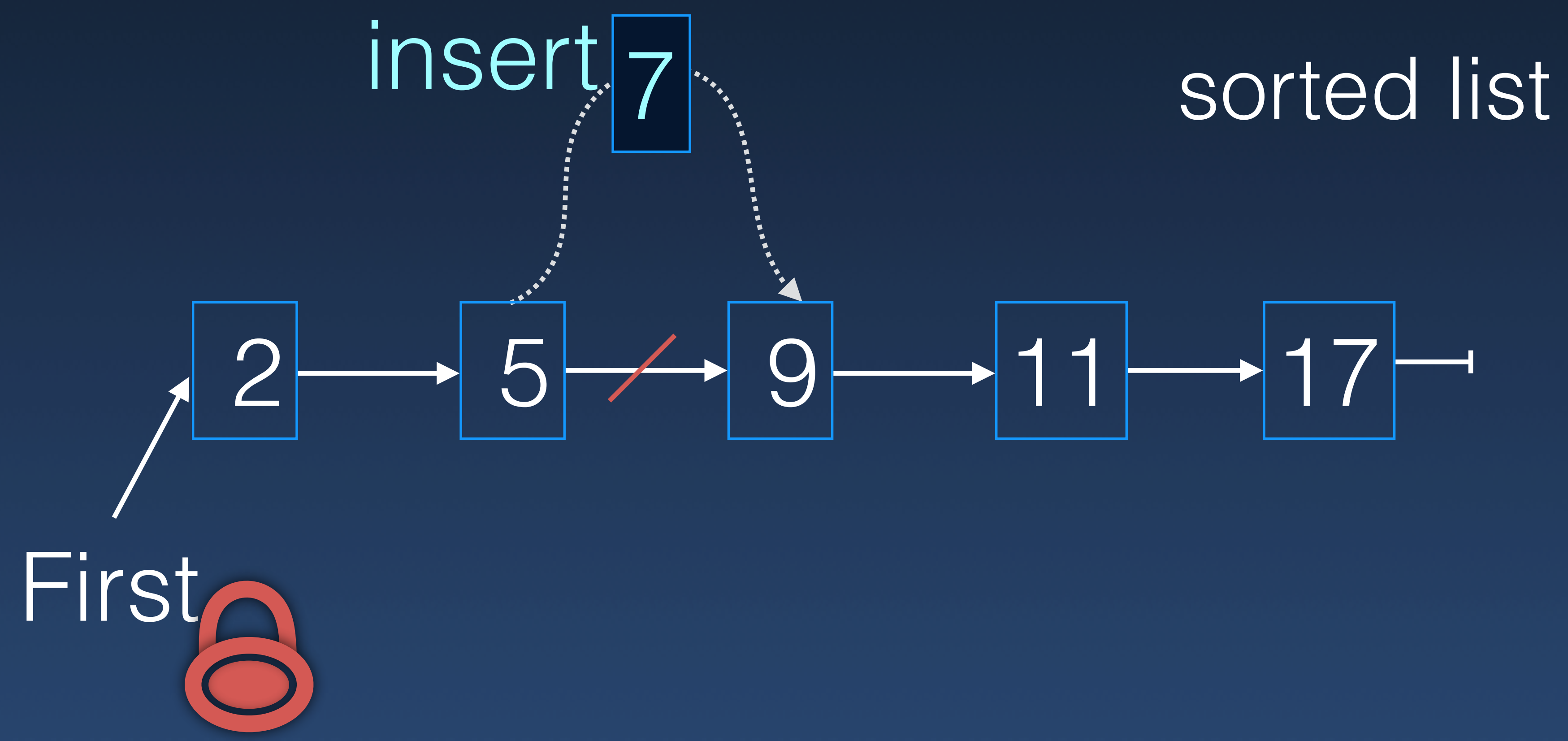
“Sequential Equivalence”



Subodh Kumar

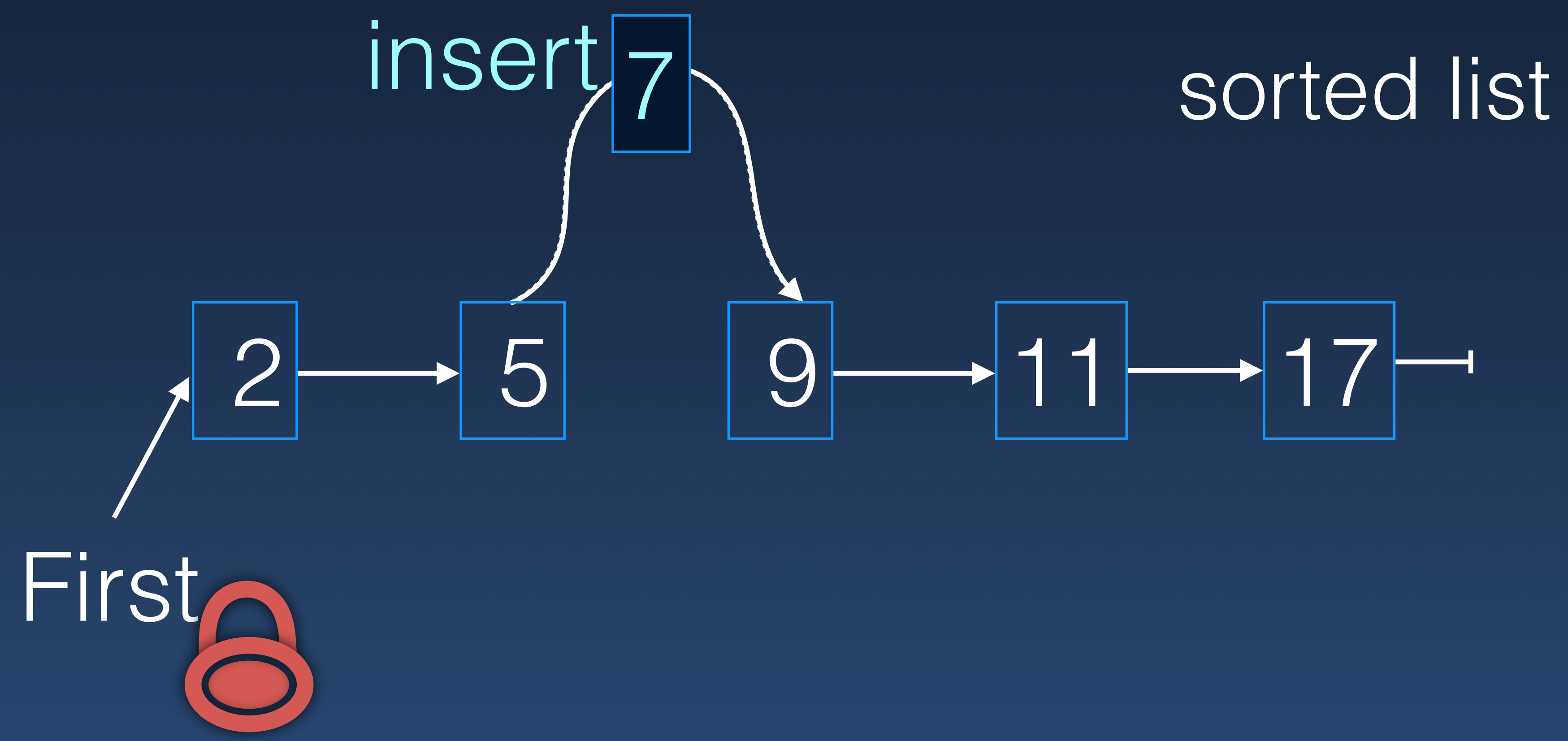
Lock

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



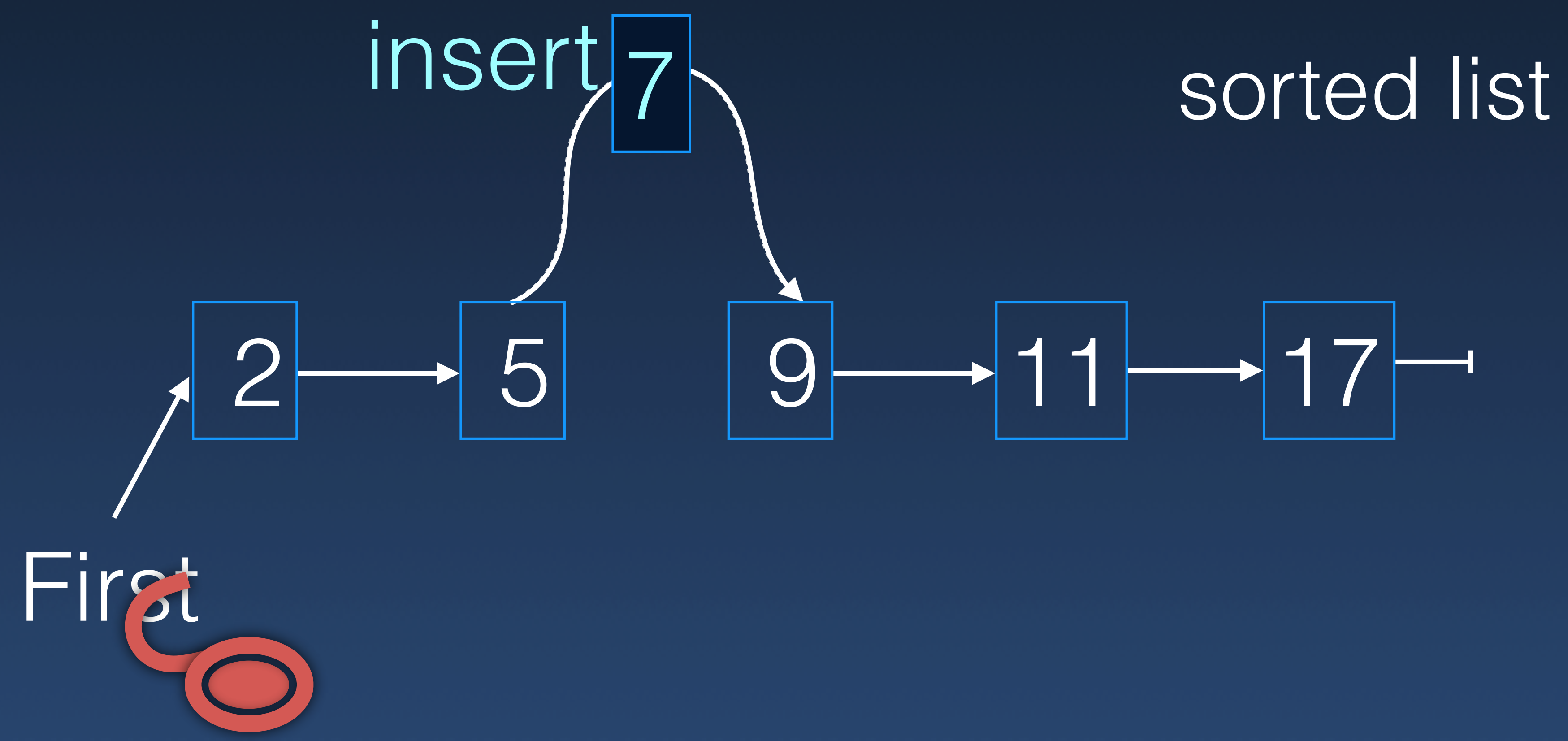
Lock

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



Lock

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



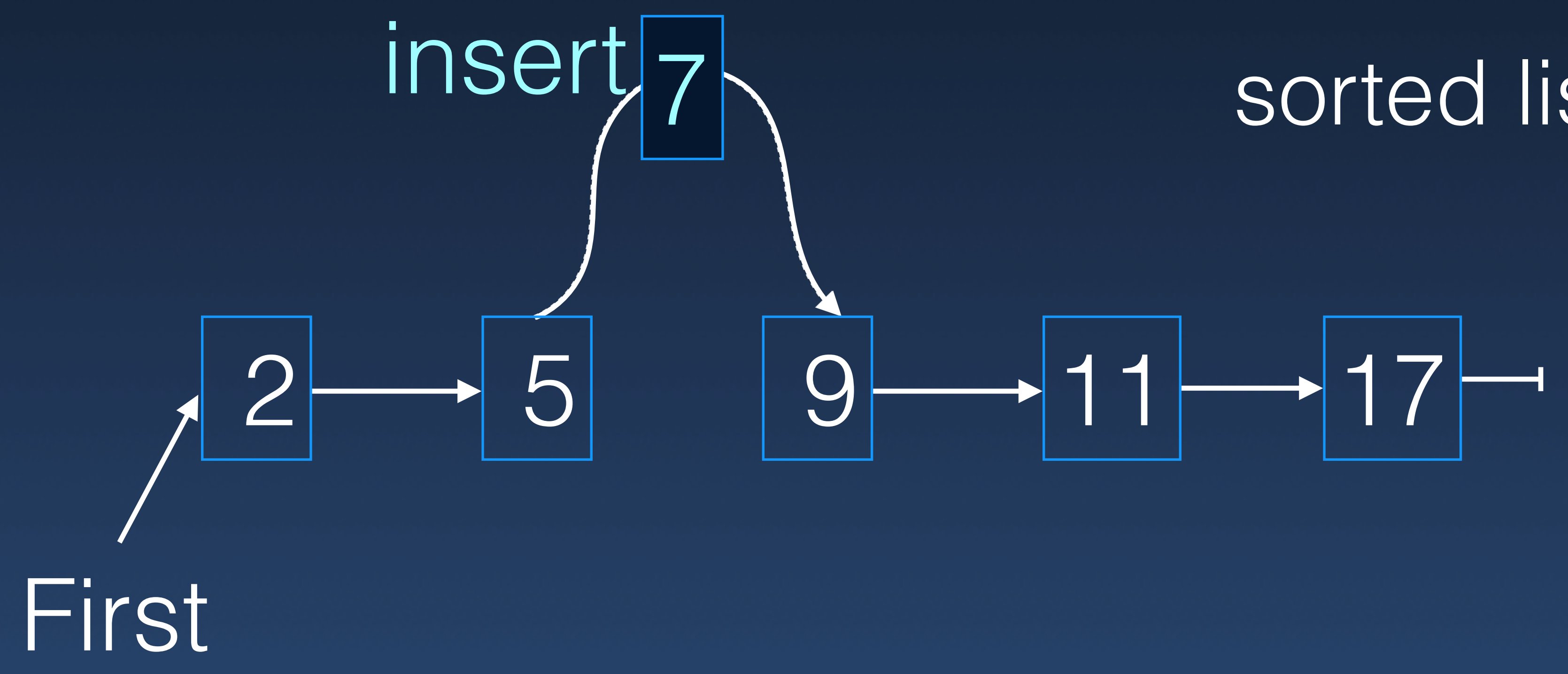
Subodh Kumar

Lock

- Lock “resources”

lock($lockA$)

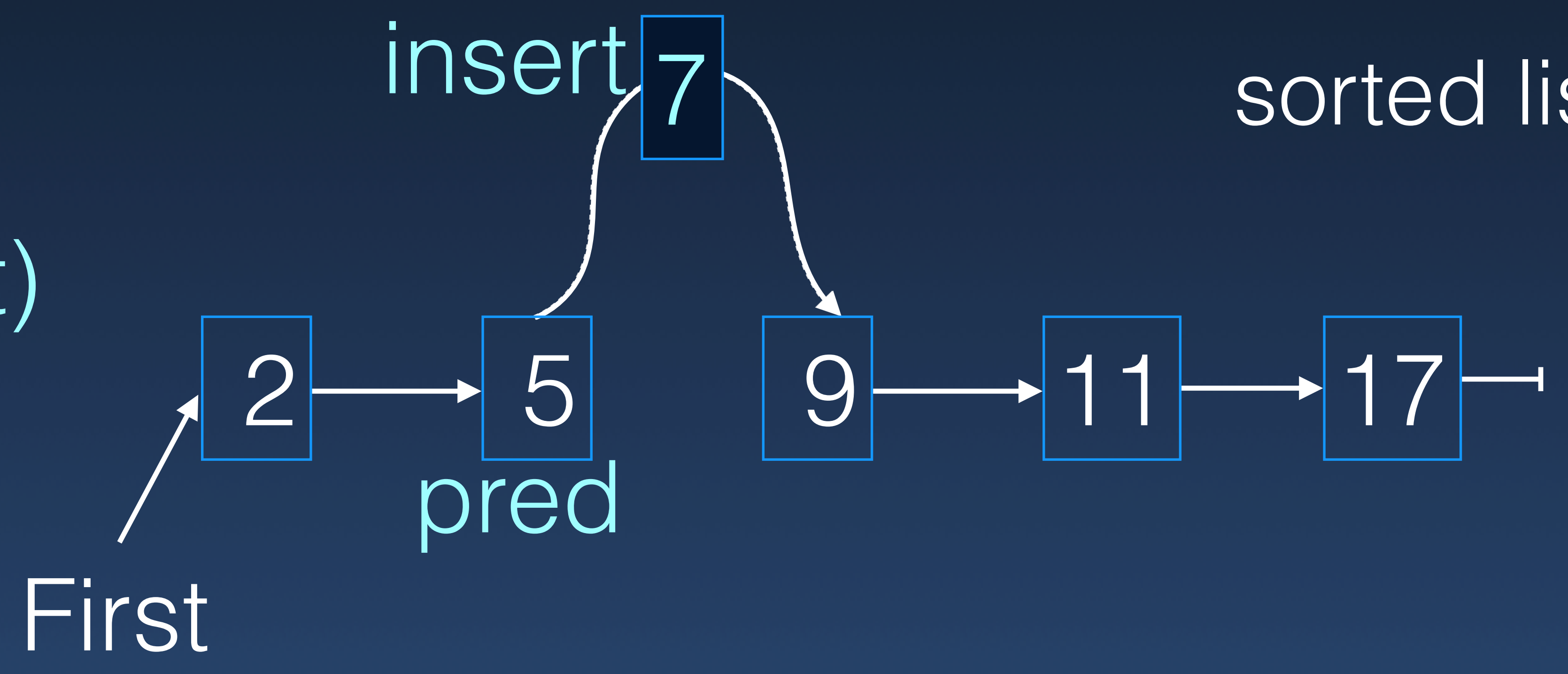
- Process
- Unlock “resources”



Lock

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

```
lock($lockA$)

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

unlock($lockA$)
```

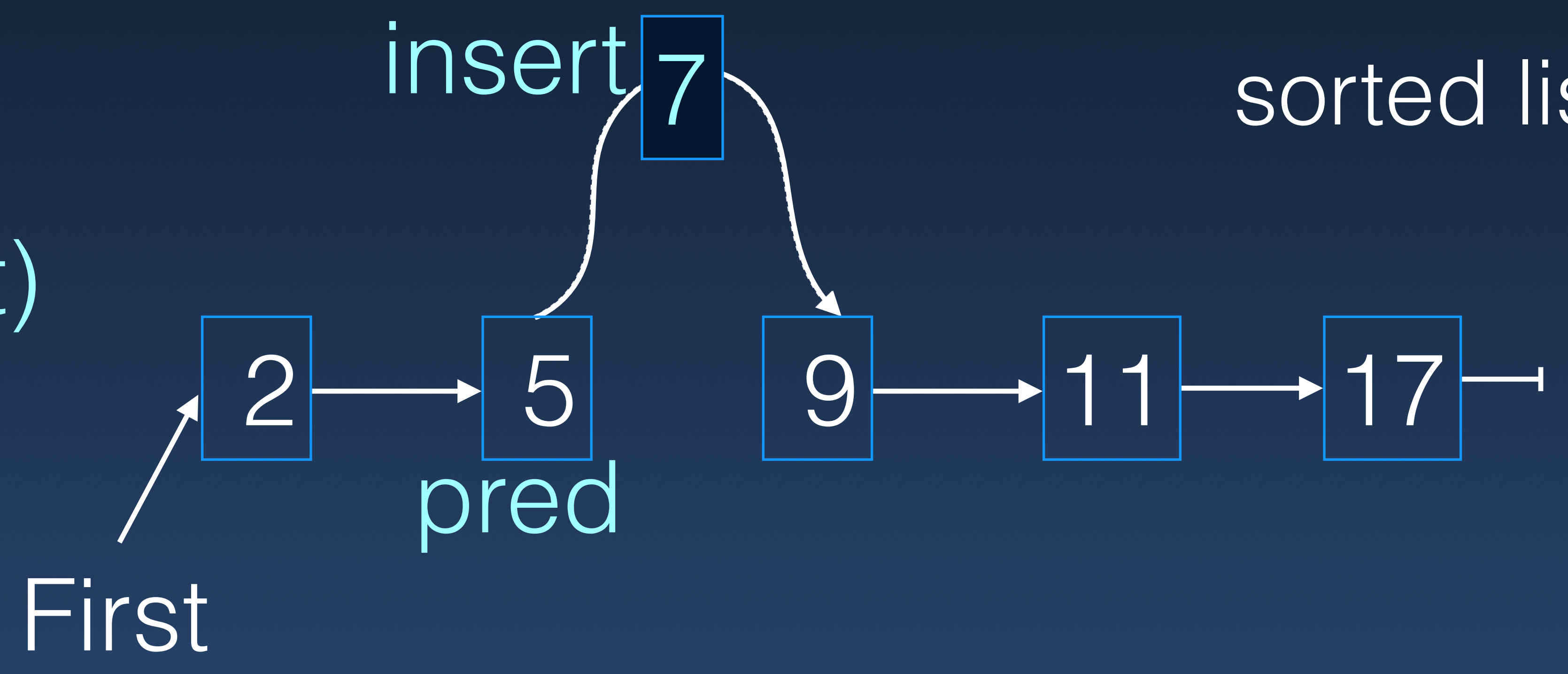


sorted list

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>

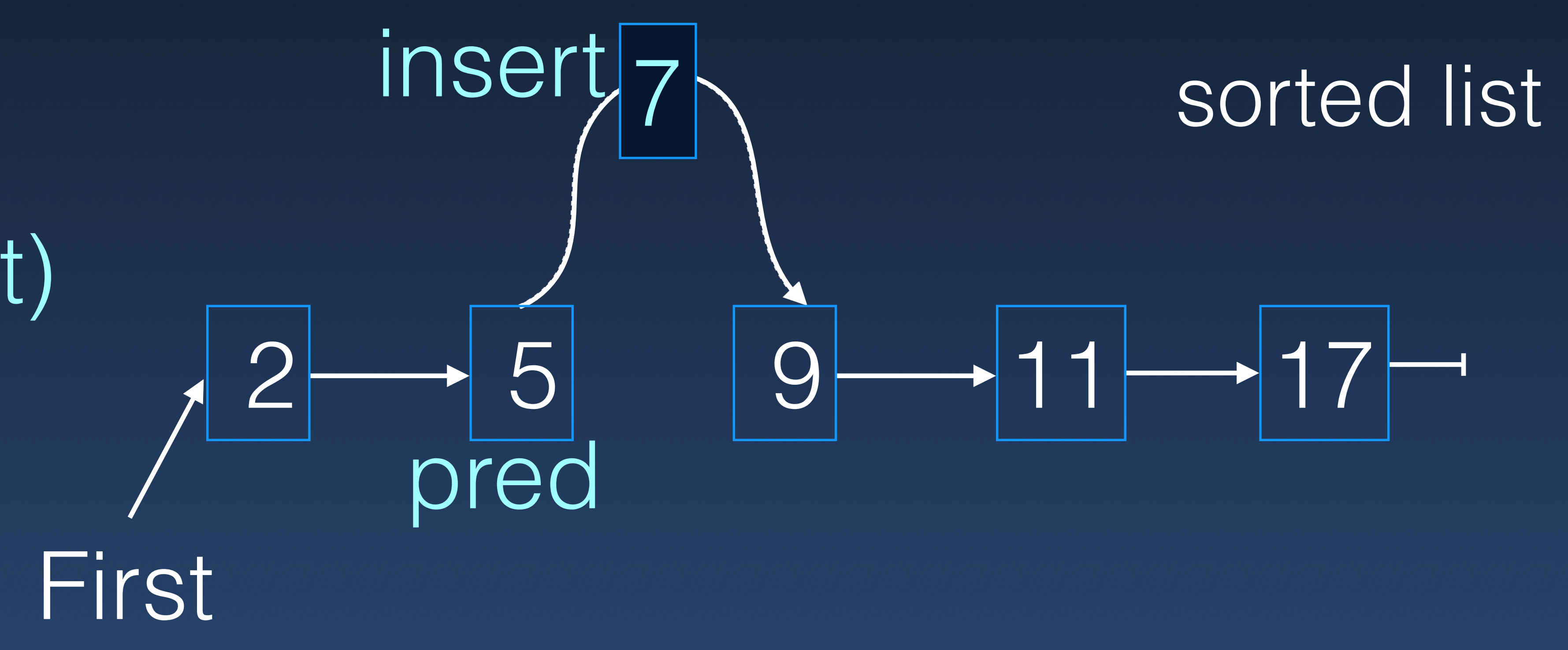
lock($lockA$)

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

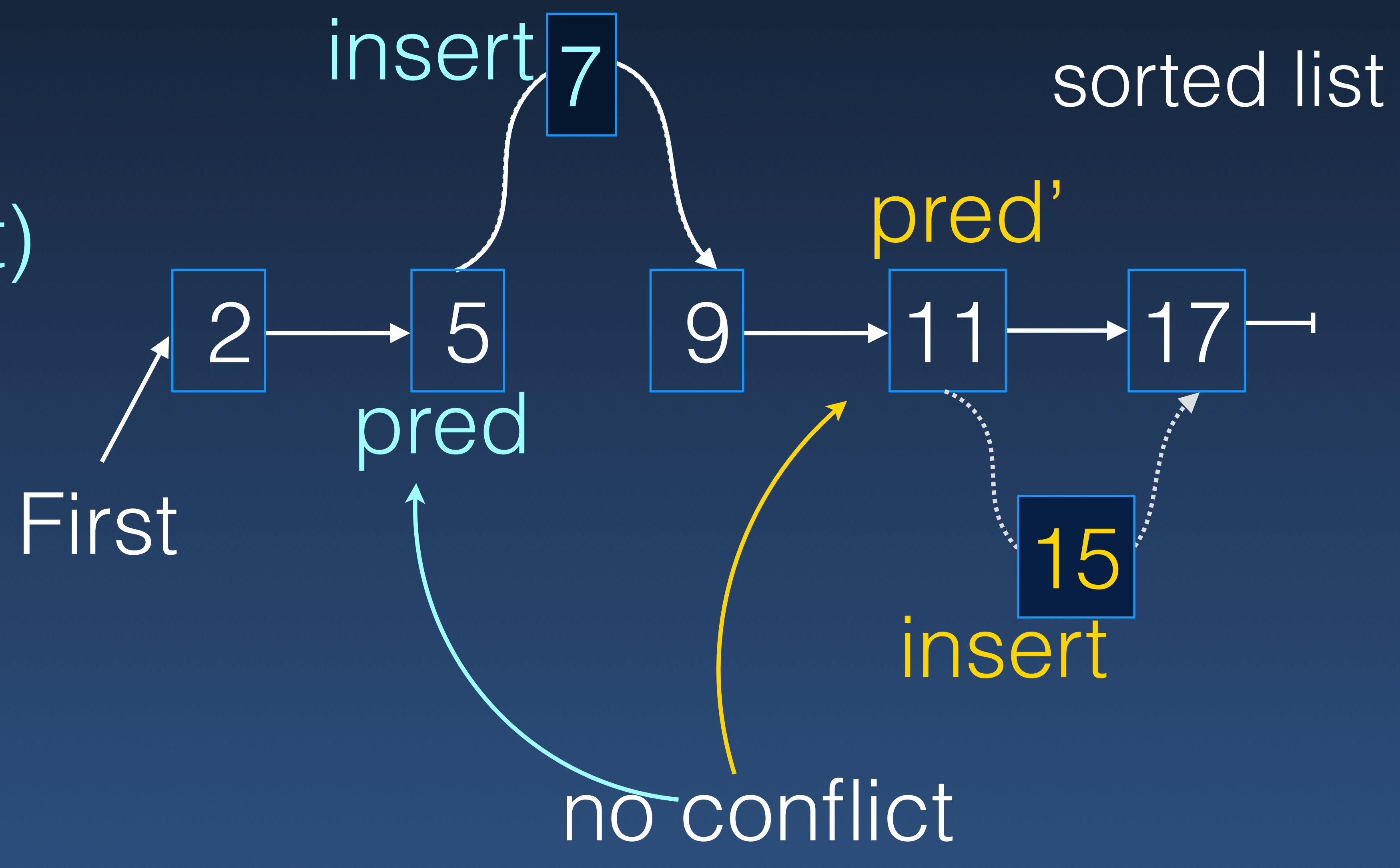
pred = Find(key)
pred.nxt = Node(key, pred.nxt)

- Unlock “resources”

unlock(*lockA*)

<Release> [schedule]

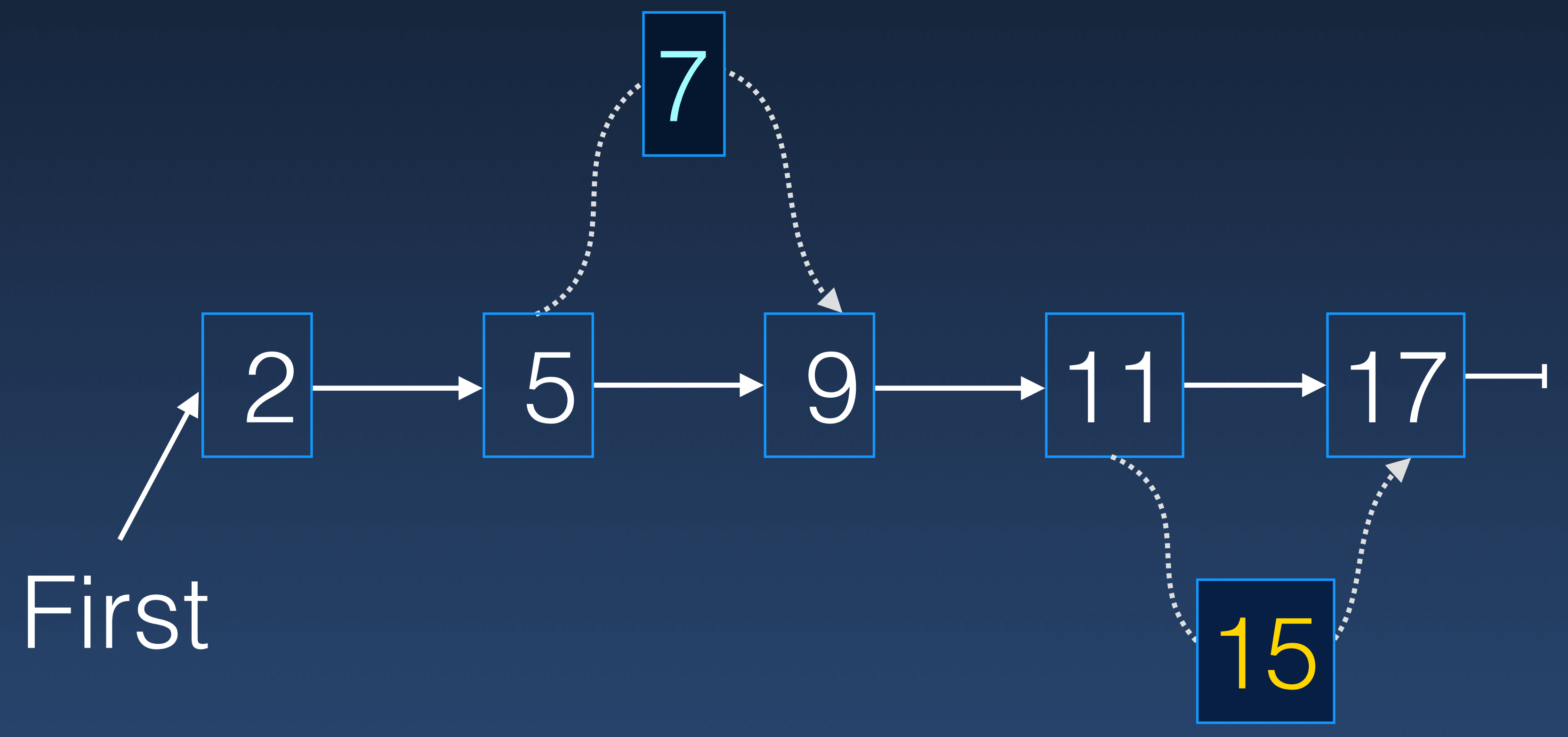
```
C++:
    std::mutex m;
    std::lock(m);
    doCriticalwork();
    std::unlock(m);
```



Lock the entire list?

List using Lock

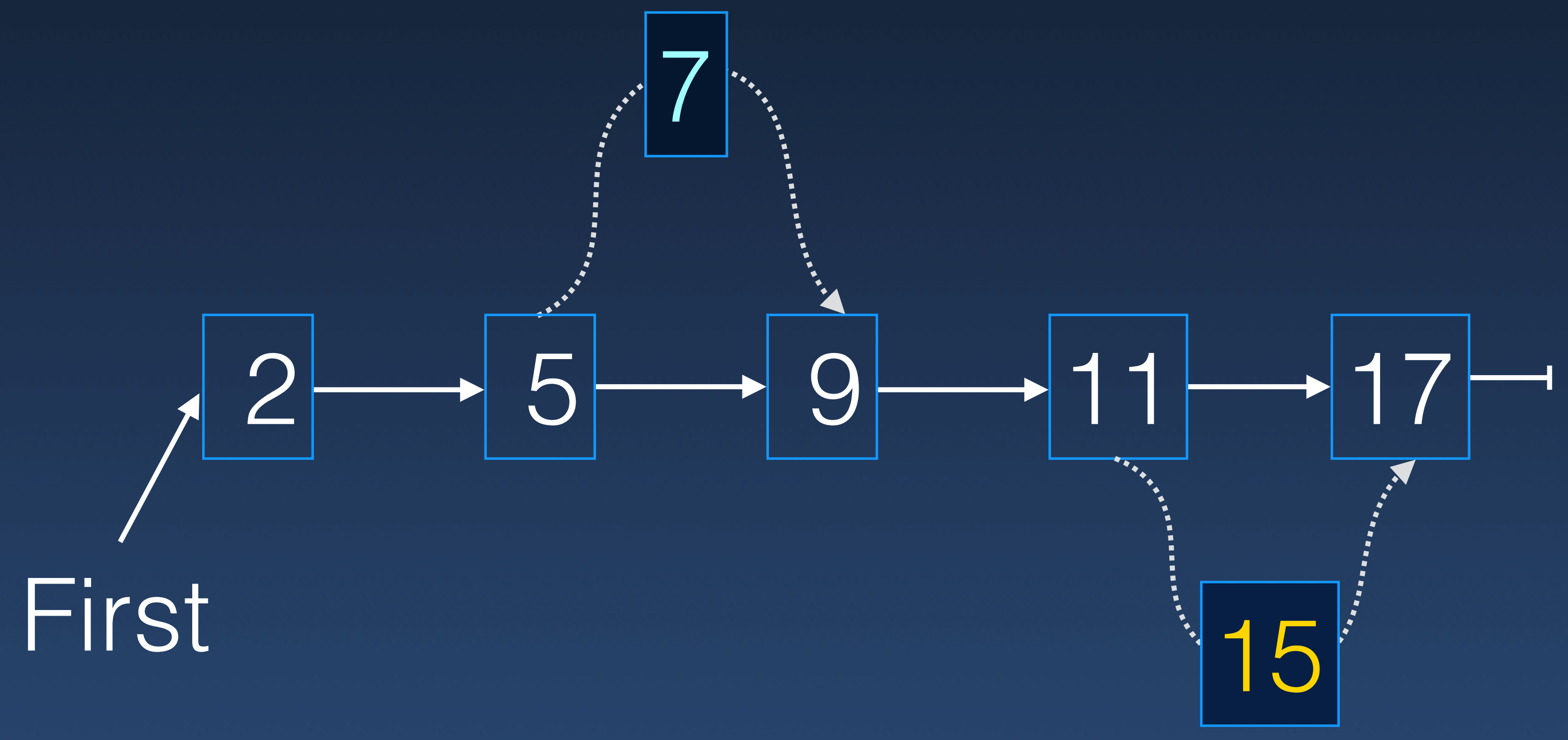
- Lock “resources”
- Process
- Unlock “resources”



Subodh Kumar

List using Lock

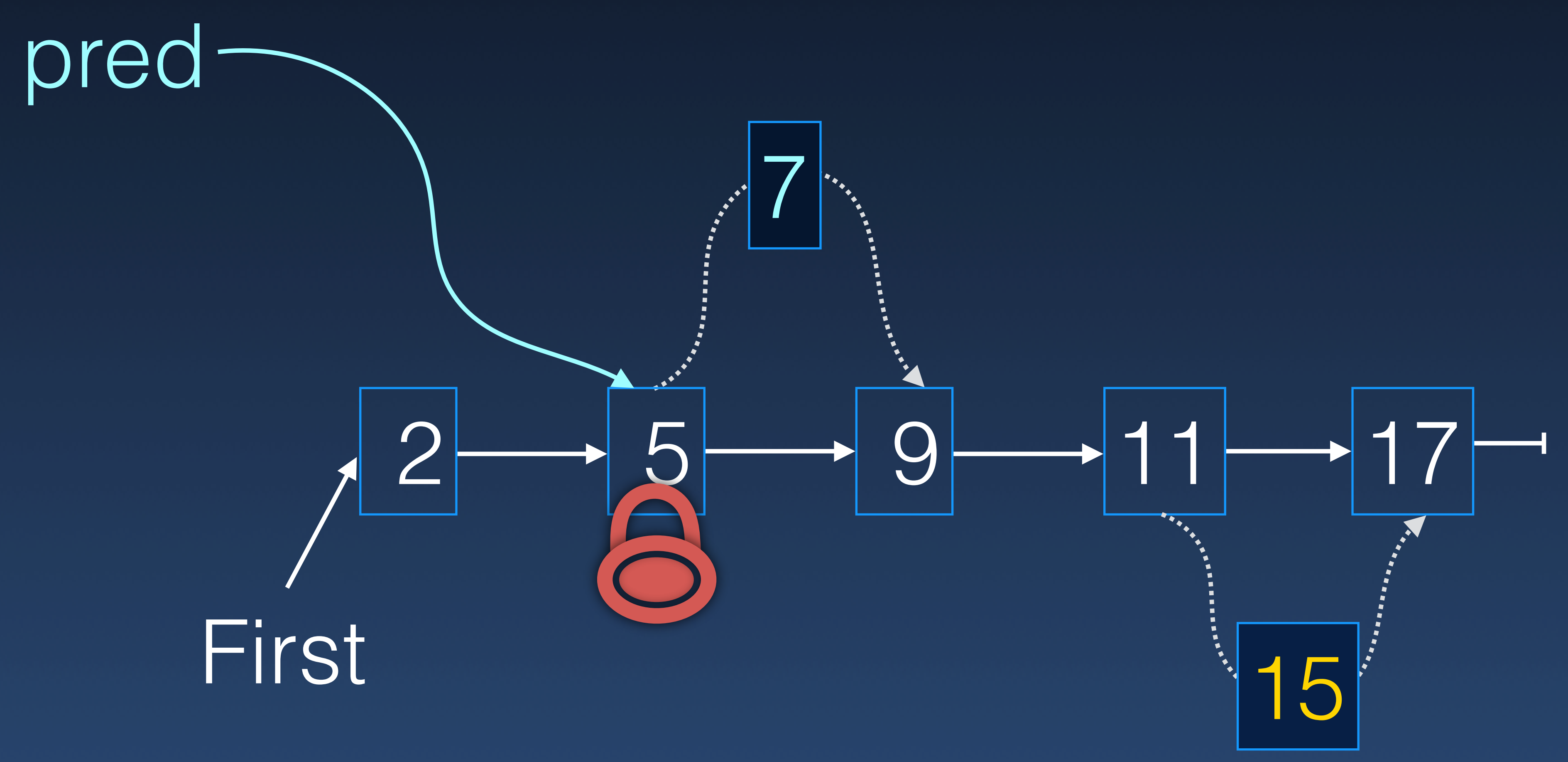
- Lock “resources”
- Process
- Unlock “resources”



```
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
```

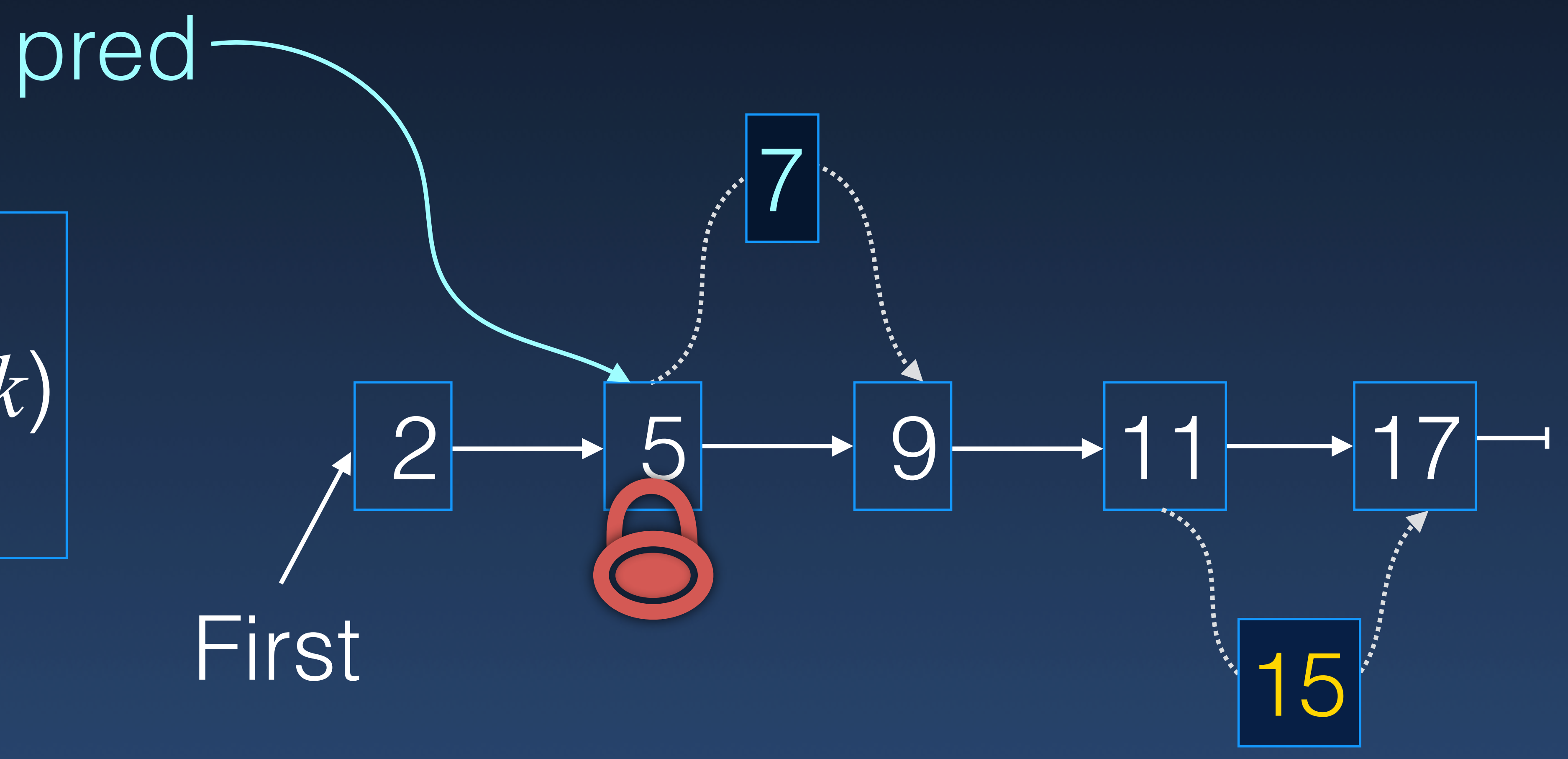
```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

Subodh Kumar

List using Lock

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

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



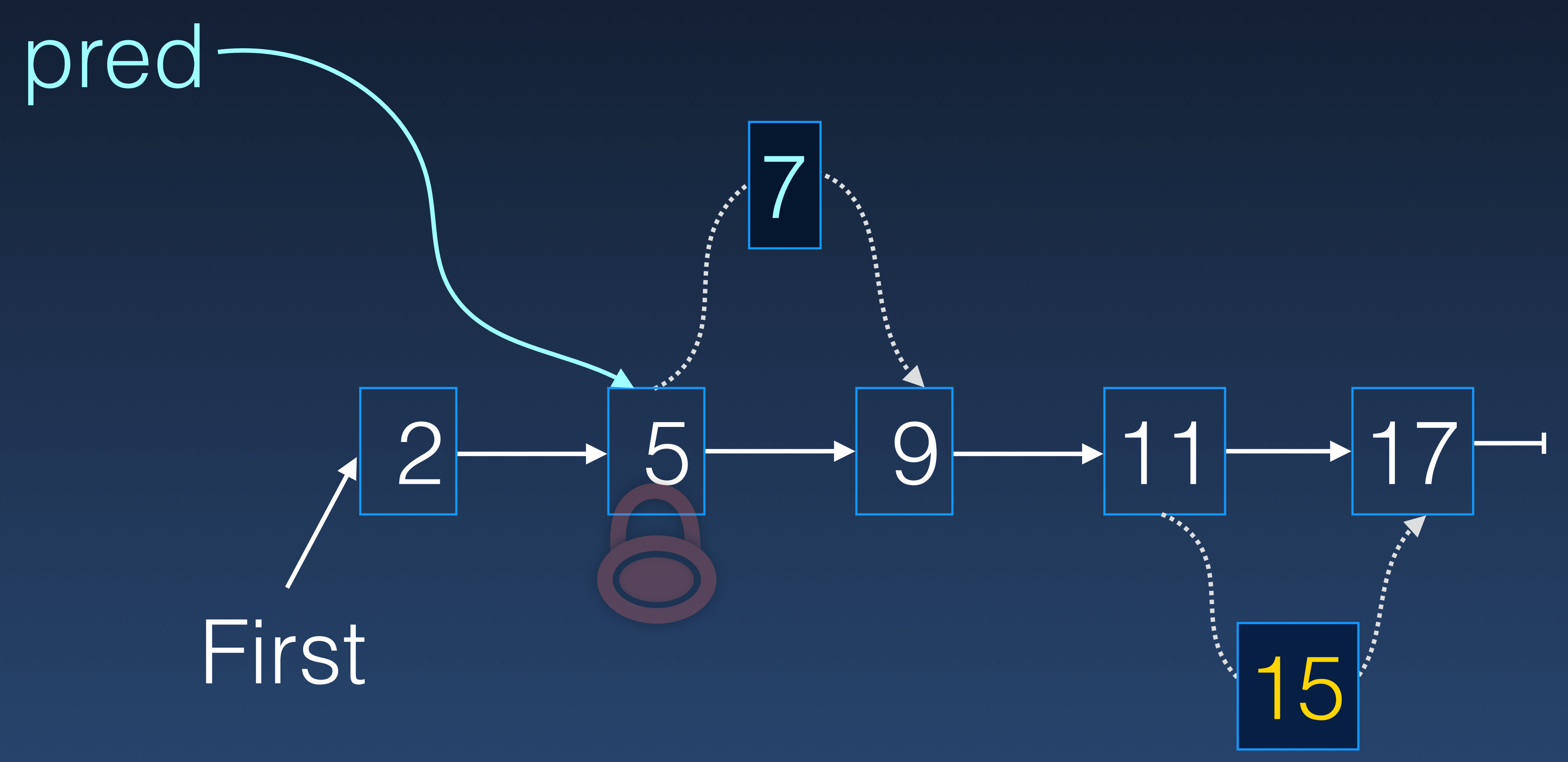
Insertion Loop

```
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
```

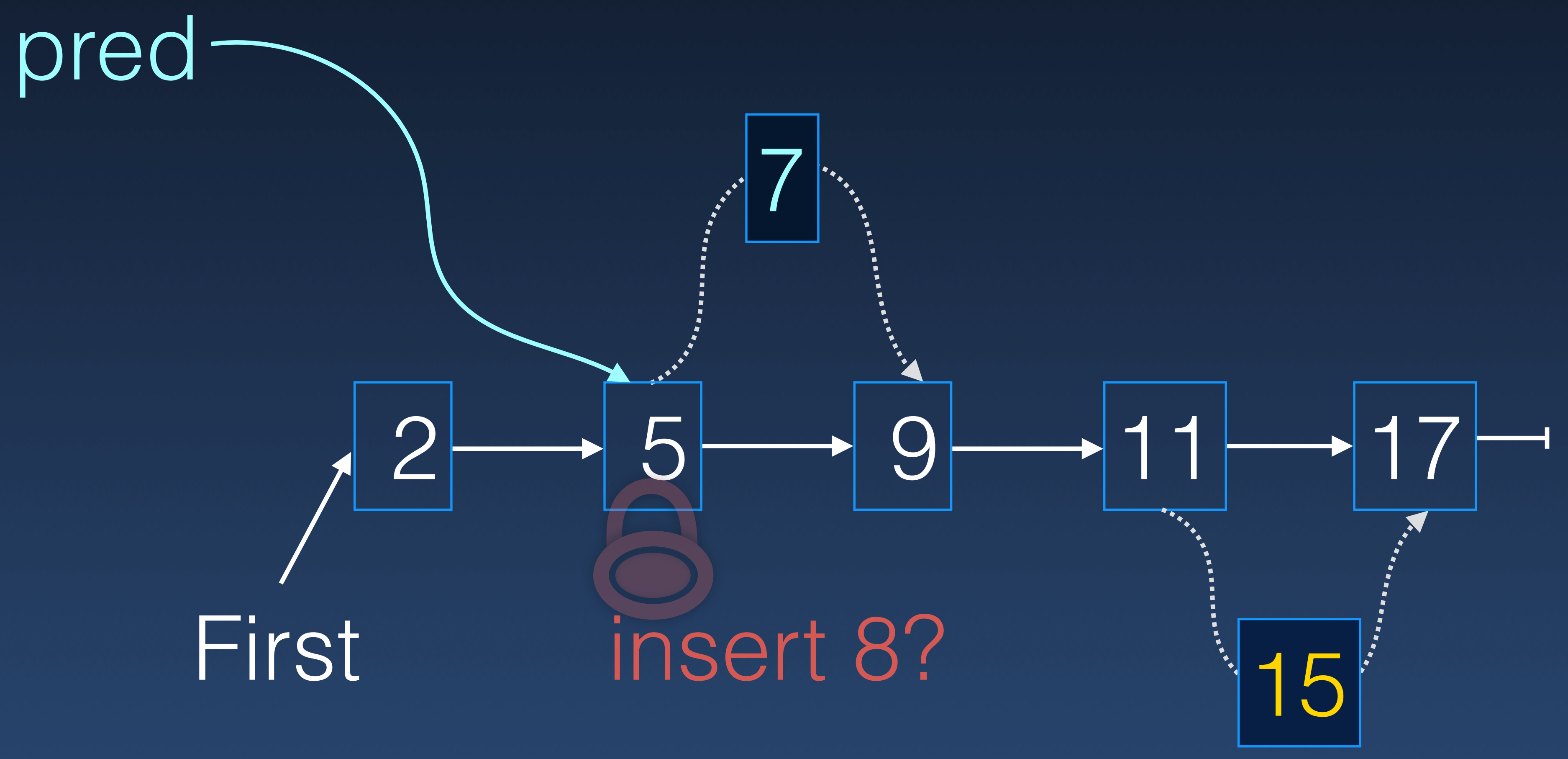
Is “9” still next to “5”?

```
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
```

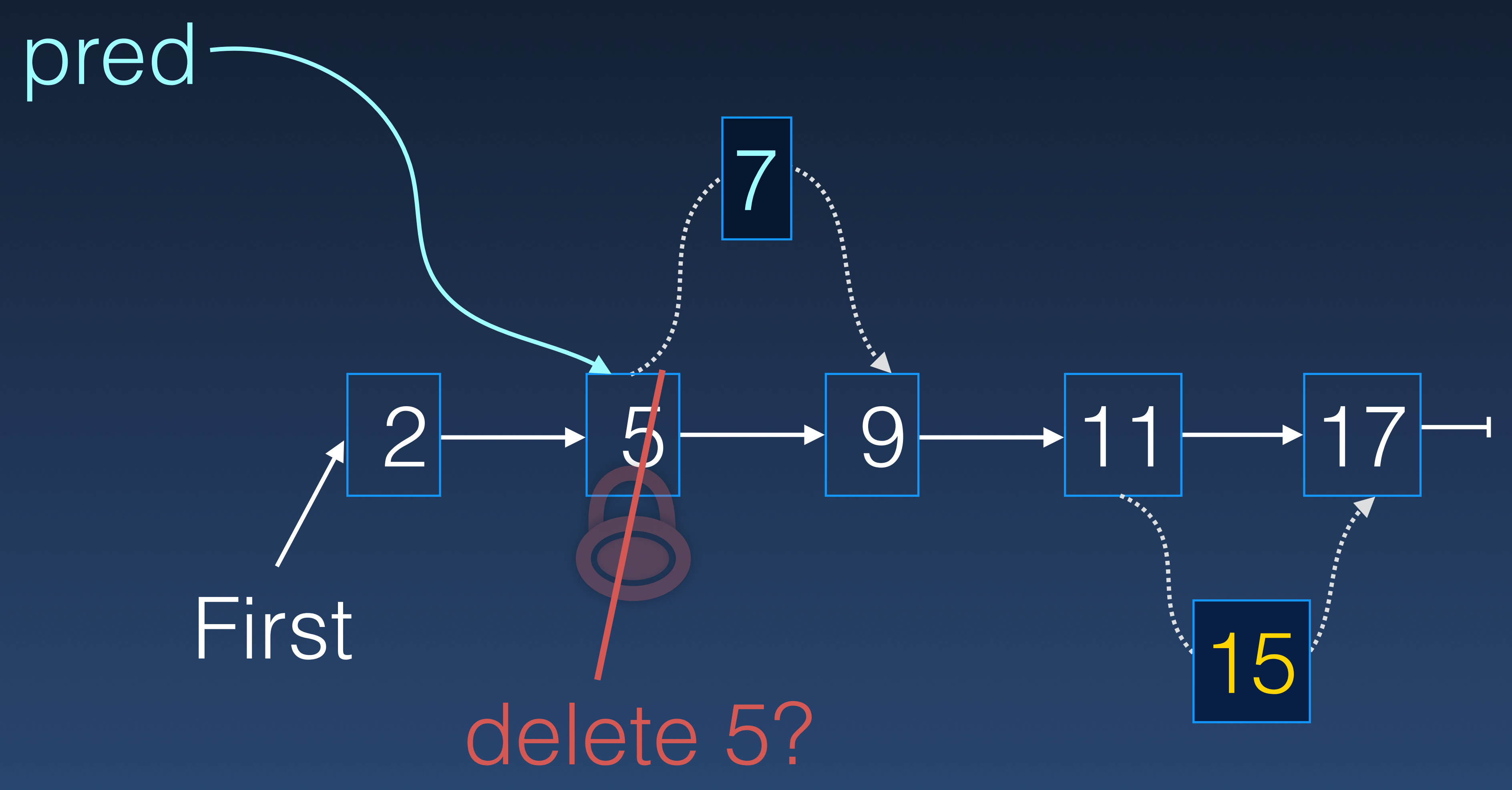
Is “9” still next to “5”?

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

Subodh Kumar

List using Lock

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



Insertion Loop

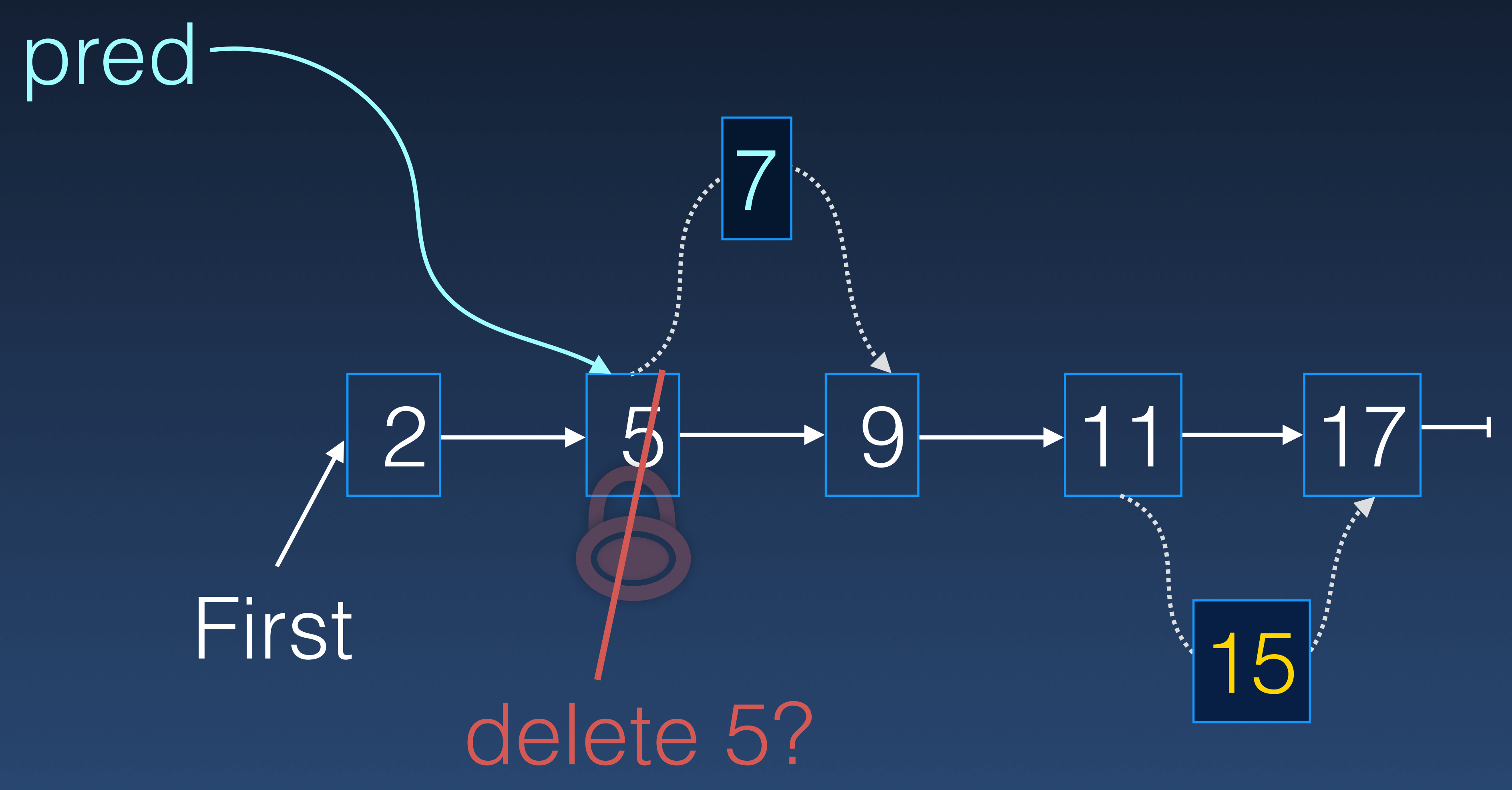
```
lock(pred)
if(key in [pred->key:pred->nxt->key)) {
      pred->nxt = Node(key, pred->nxt, new(Lock))
}
unlock(pred)
pred = pred->nxt Is “9” still next to “5”?
```

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

Subodh Kumar

List using Lock

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



Insertion Loop

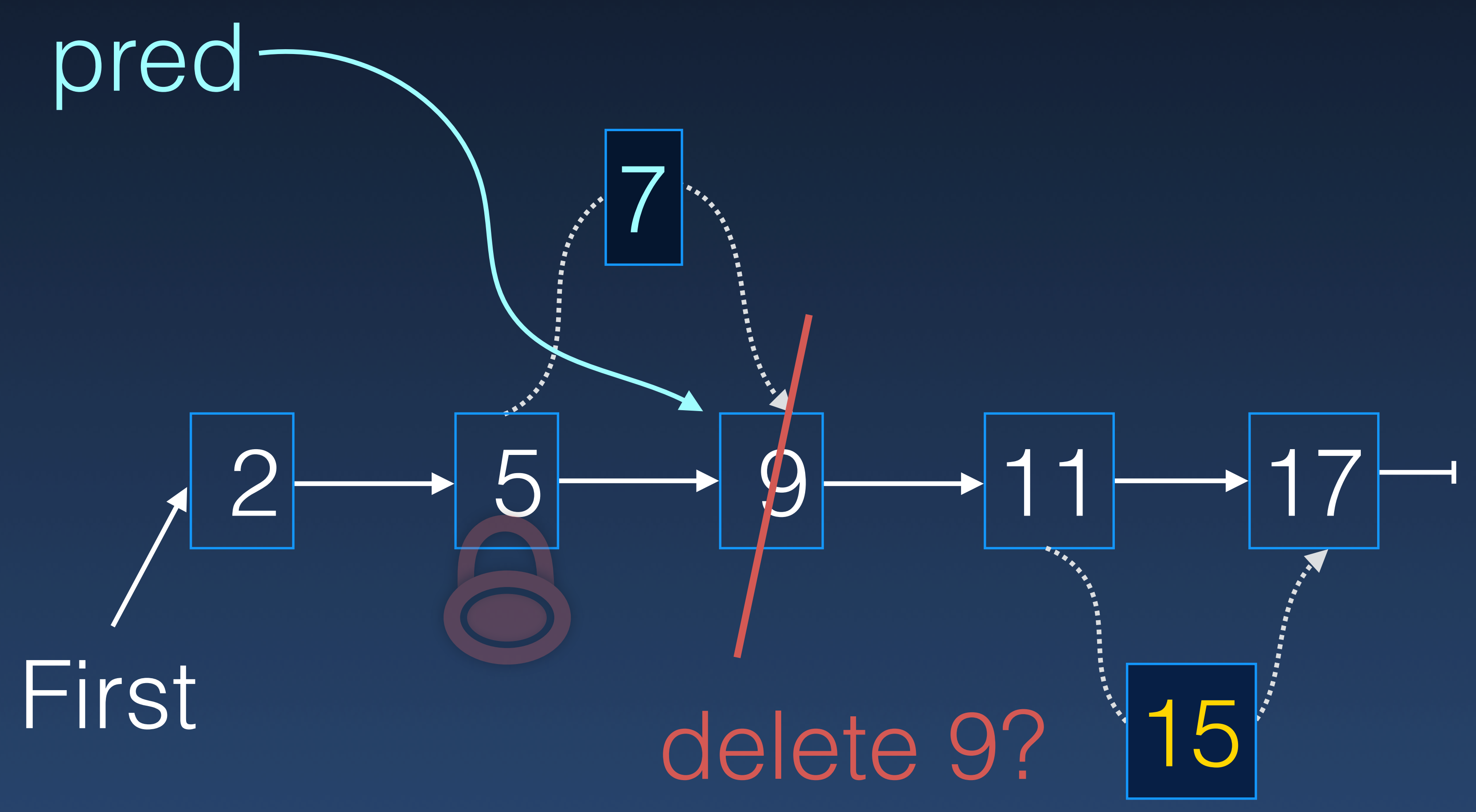
```
lock(pred)
if(key in [pred->key:pred->nxt->key)) {
      pred->nxt = Node(key, pred->nxt, new(Lock))
}
unlock(pred)
pred = pred->nxt
```

Before unlocking pred, capture ‘nxt’ locally?

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

List using Lock

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



Insertion Loop

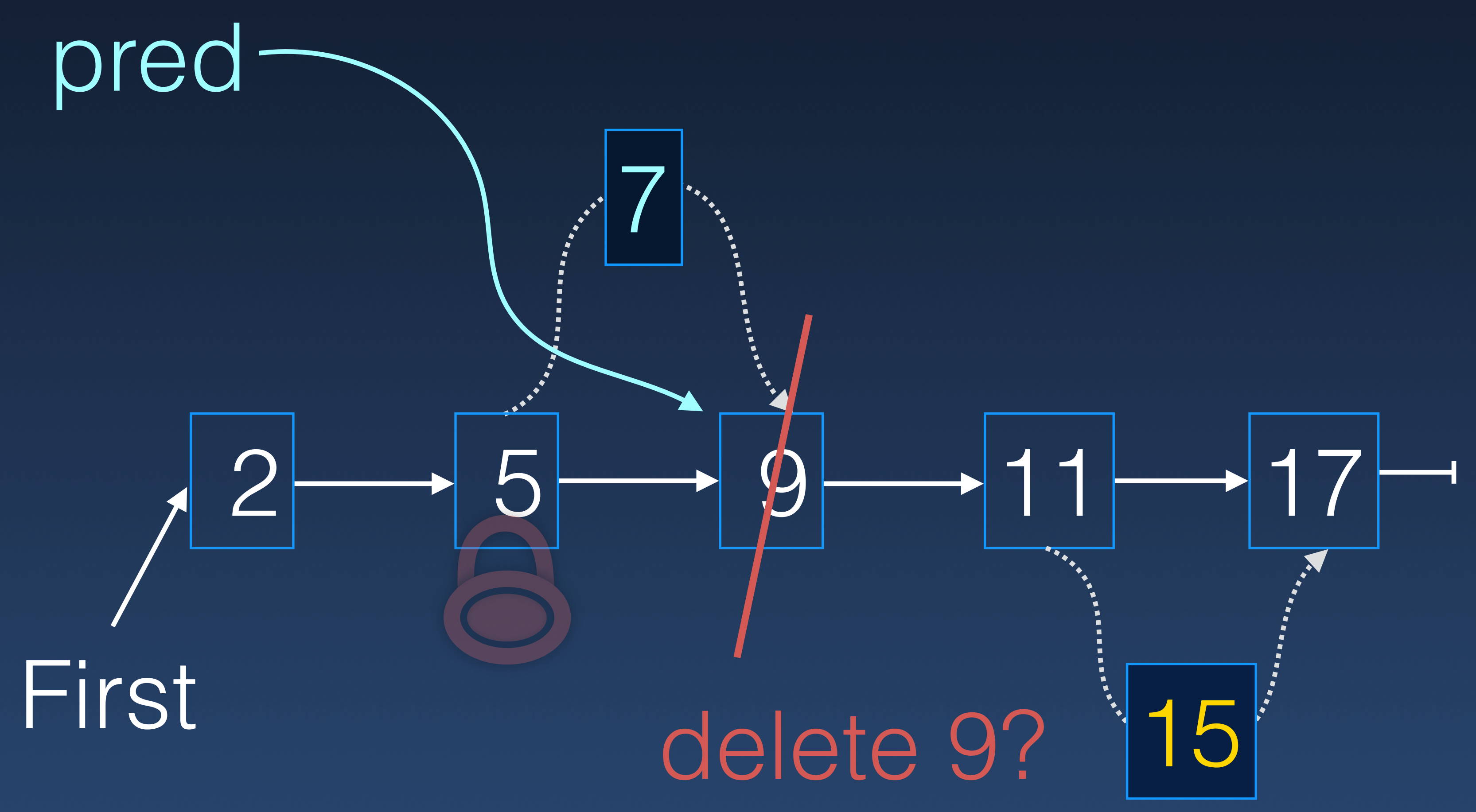
```
lock(pred)
if(key in [pred->key:pred->nxt->key)) {
      pred->nxt = Node(key, pred->nxt, new(Lock))
}
unlock(pred)
pred = pred->nxt
```

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
```

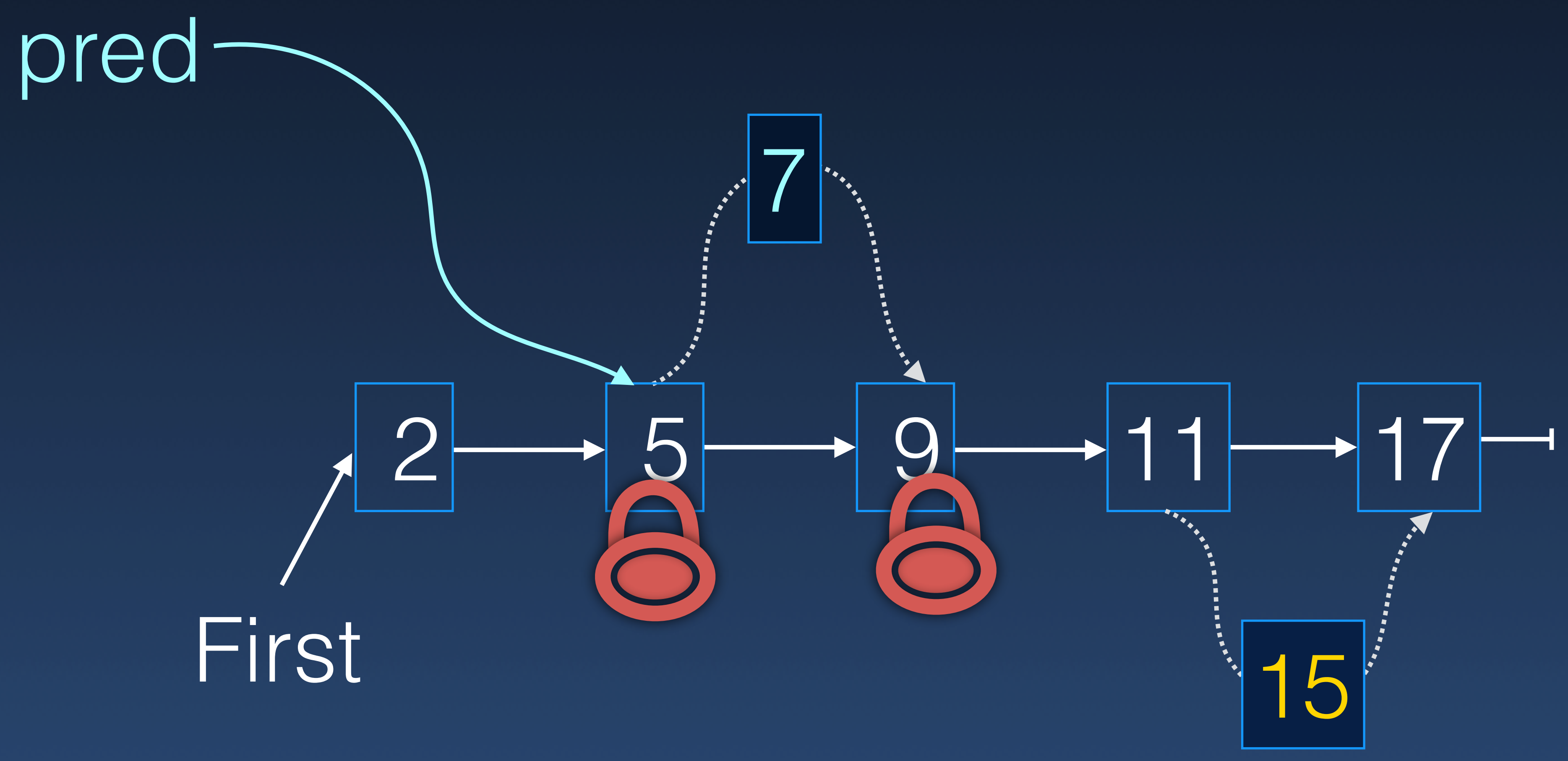
Local view?

Before unlocking pred, capture ‘nxt’ locally?

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

List using Lock

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



Insertion Loop

Local view?

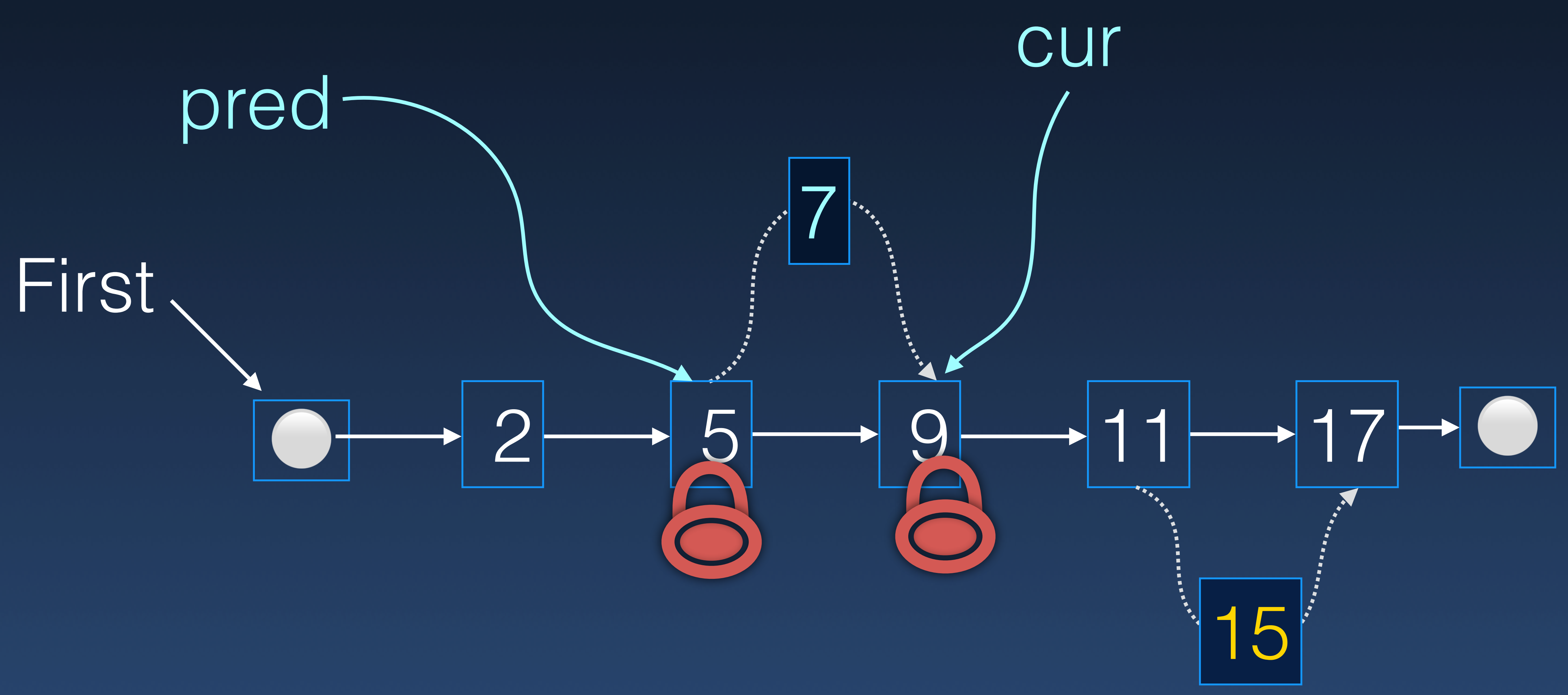
```
lock(pred)   And lock(pred->nxt)
if(key in [pred->key:pred->nxt->key)) {
      pred->nxt = Node(key, pred->nxt, new(Lock))
}
unlock(pred)
pred = pred->nxt
```

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

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)
```

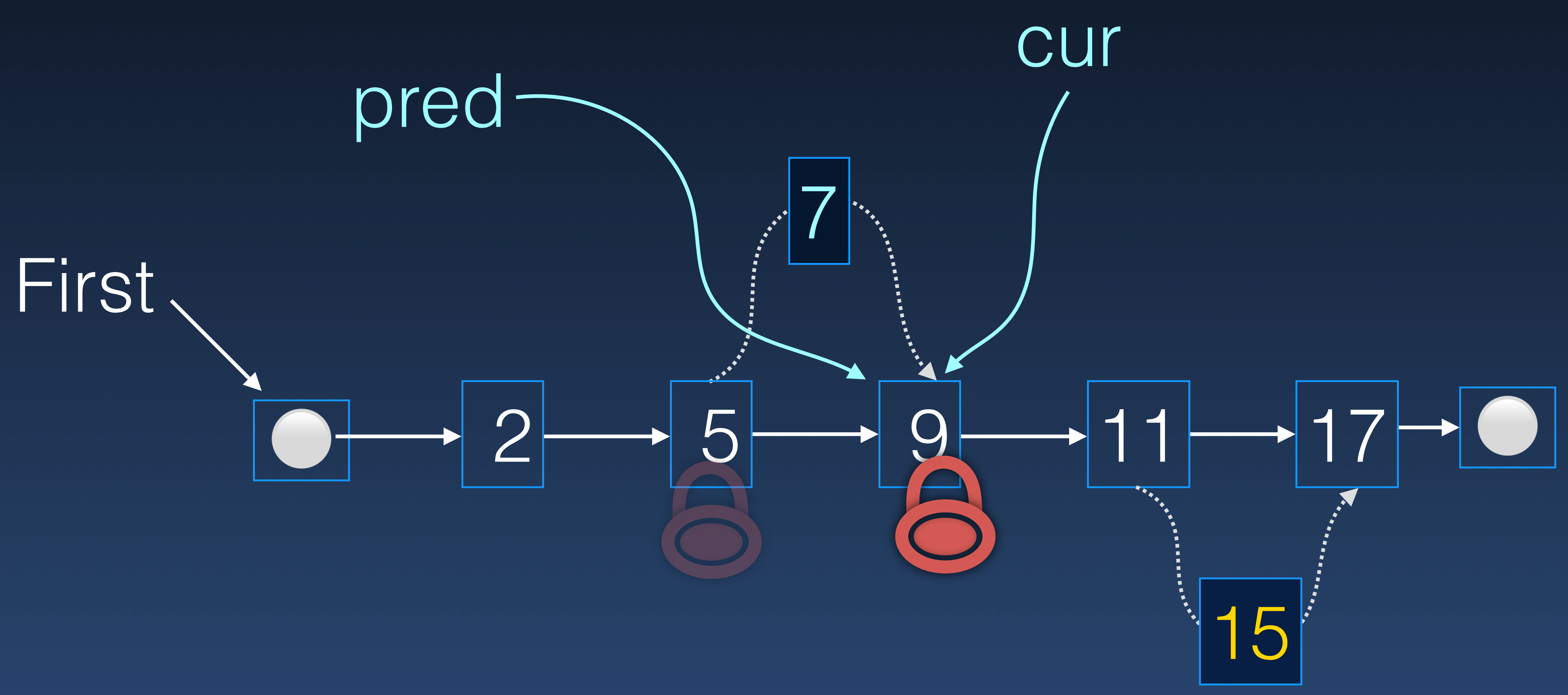


```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

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)
```

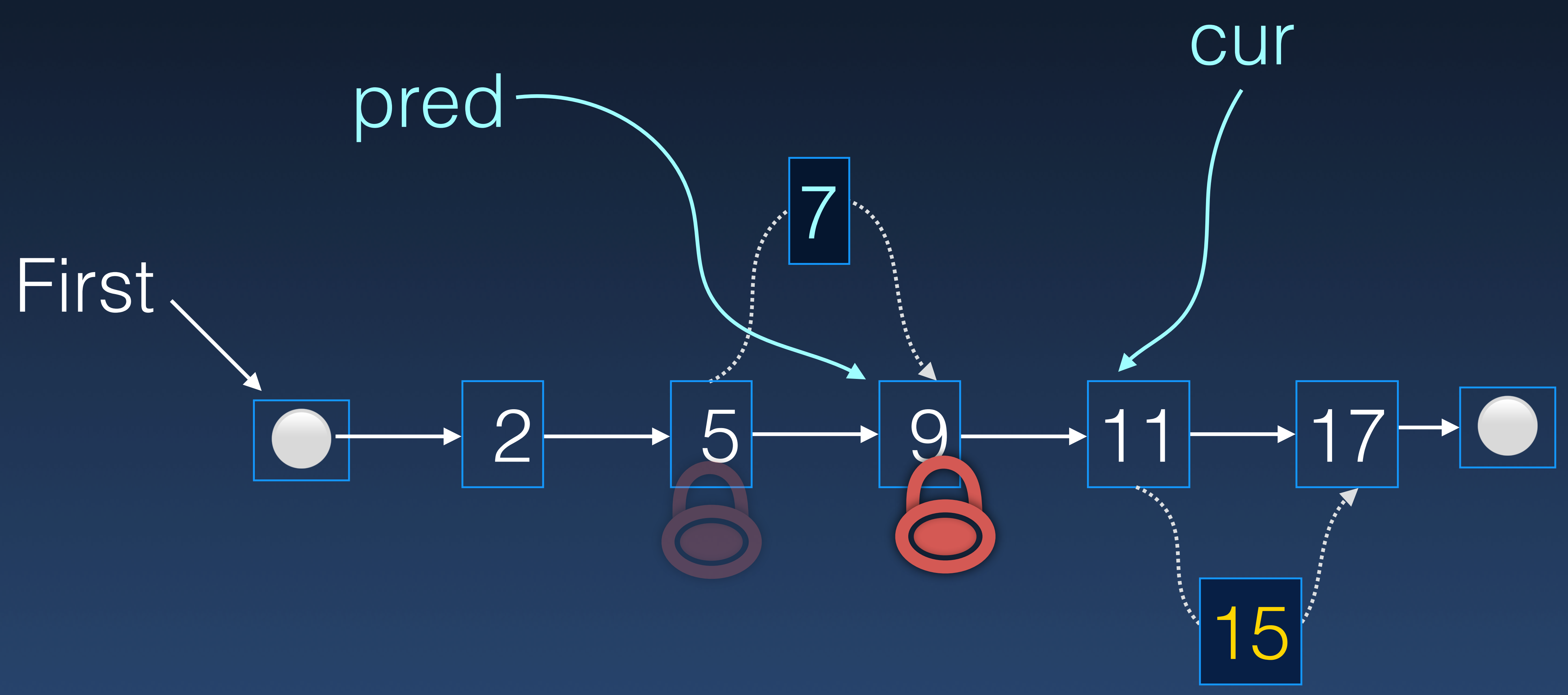


```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

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)
```

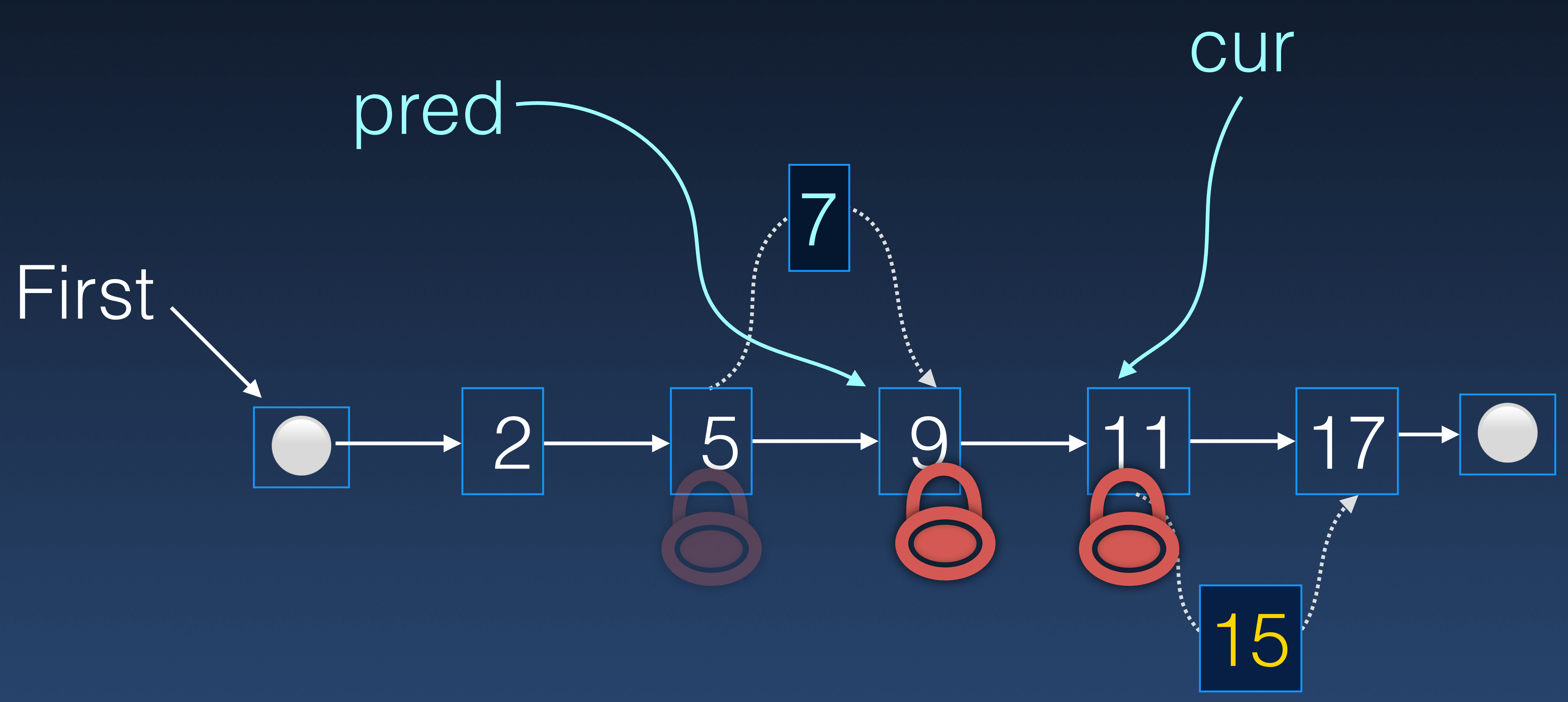


```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

Subodh Kumar

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)
```

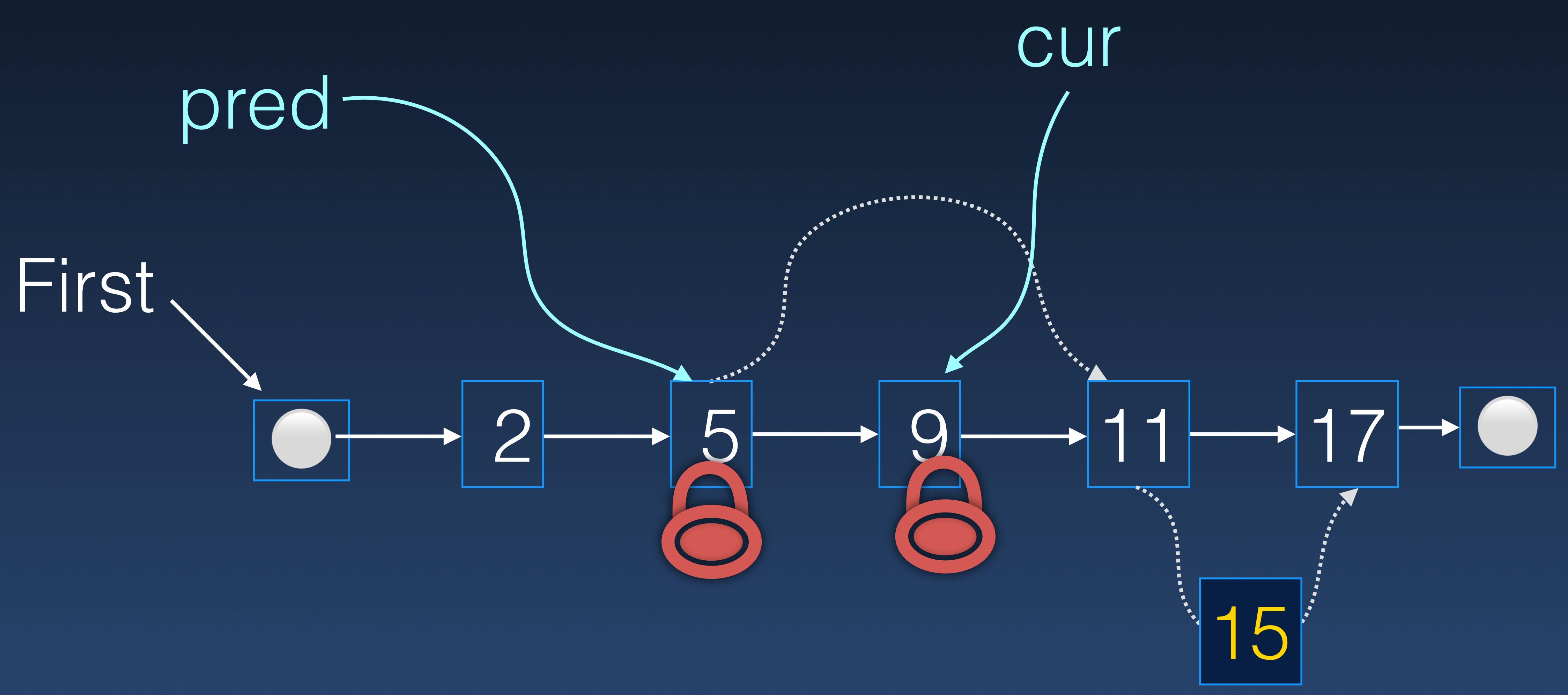


```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

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)
```



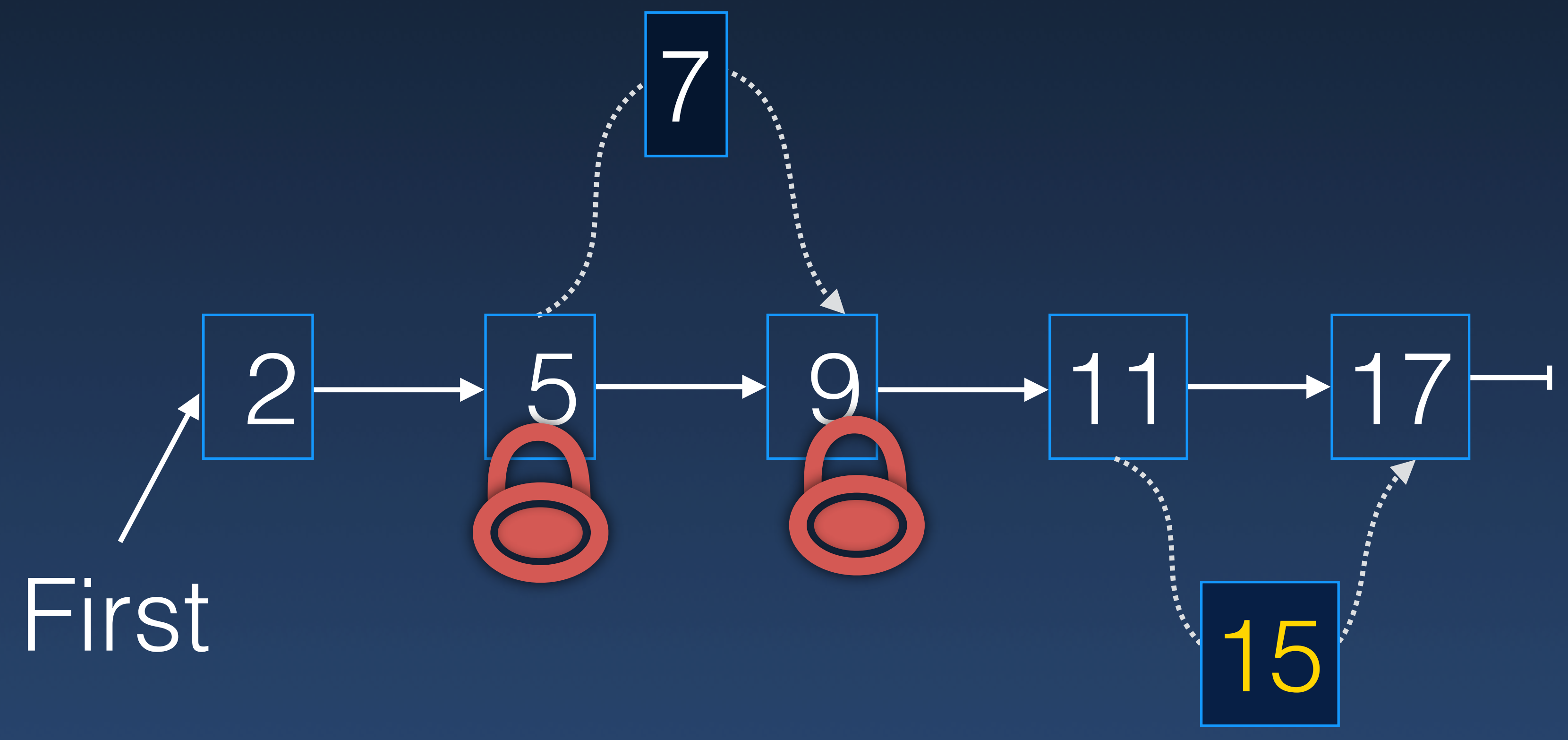
```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

Subodh Kumar

Lock

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

➡ Correctness depend on everyone following protocol



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

Subodh Kumar