Towards Temporal Verification of Concurrent Data-Structures

Alejandro Sánchez[1] **César Sánchez**[1,2]

[1]The IMDEA Software Institute, Spain

[2]Spanish Council for Scientific Research (CSIC), Spain

SVARM'10, Edinburgh, 21 July 2010

What are we interested in

- Imperative programs

$$P$$

What are we interested in

- Imperative programs
- Concurrent data-structures

$$P_1 || P_2 || \cdots || P_n$$

What are we interested in

- Imperative programs

- Concurrent data-structures

$$\frac{P_1 || P_2 || \cdots || P_n}{\downarrow}$$

data structures
(heap)

What are we interested in

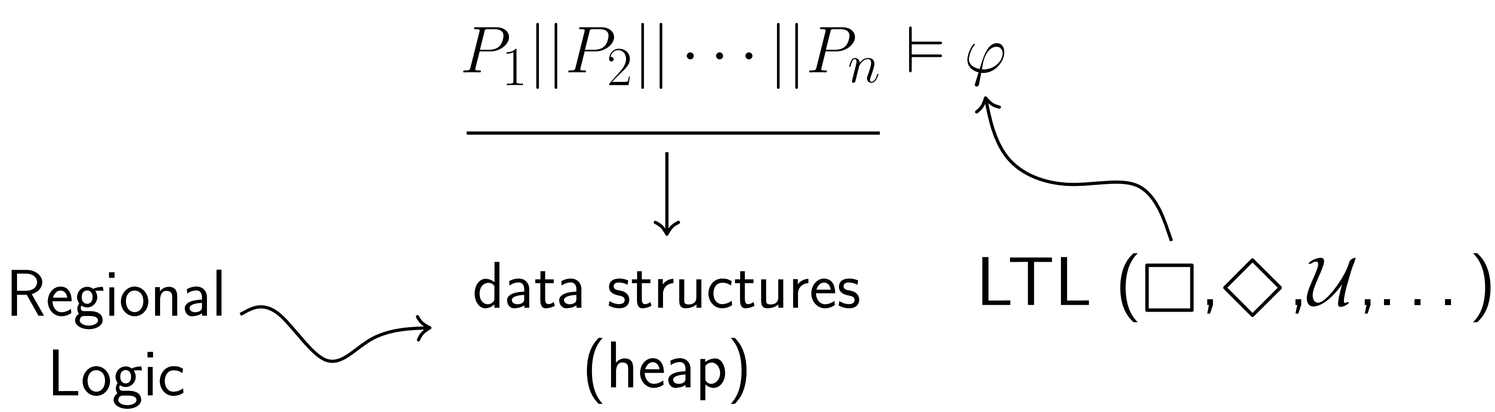
- Imperative programs
- Concurrent data-structures
- Temporal properties (safety, liveness)

$$\underbrace{P_1||P_2||\cdots||P_n}_{\downarrow} \models \varphi$$

data structures
(heap)

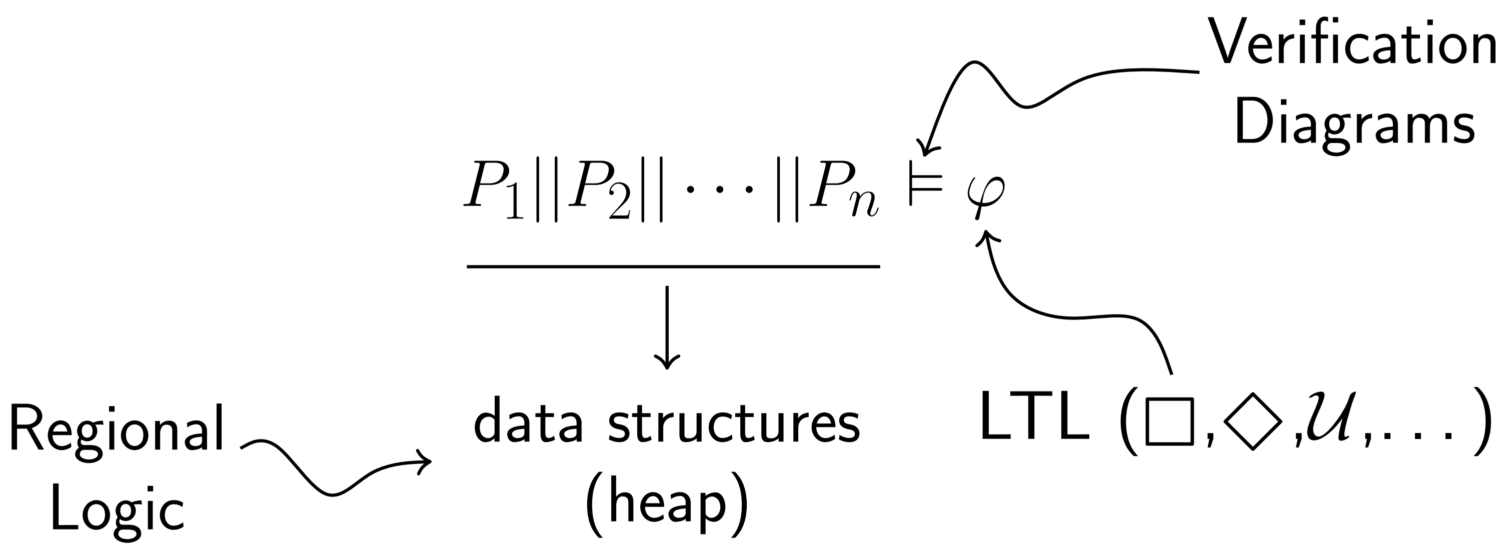
What are we interested in

- Imperative programs
- Concurrent data-structures
- Temporal properties (safety, liveness)
- Formal verification



What are we interested in

- Imperative programs
- Concurrent data-structures
- Temporal properties (safety, liveness)
- Formal verification



Reasoning About the Heap

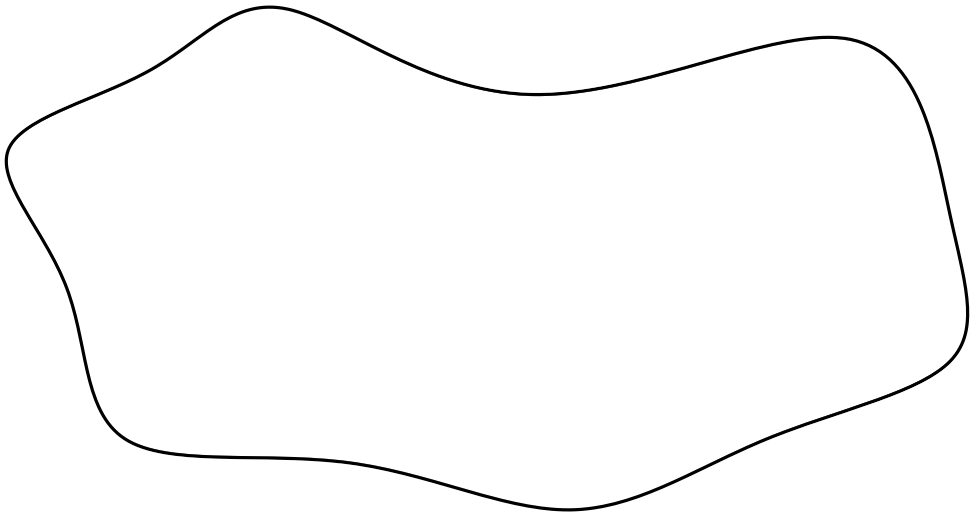
- Separation Logic

Reasoning About the Heap

- Separation Logic
  - Non-classical logic to reason about shared mutable data-structures

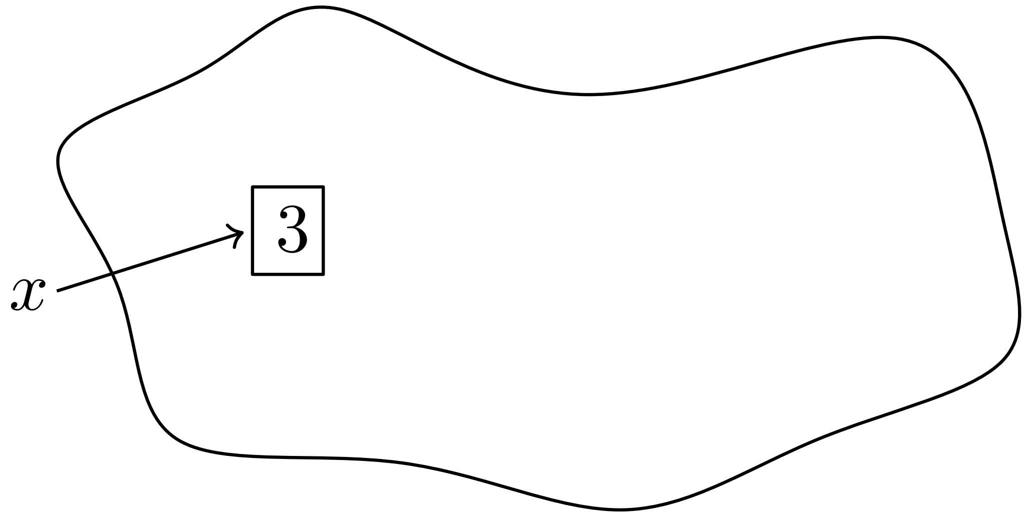
Reasoning About the Heap

- Separation Logic
  - Non-classical logic to reason about shared mutable data-structures
  - `emp`



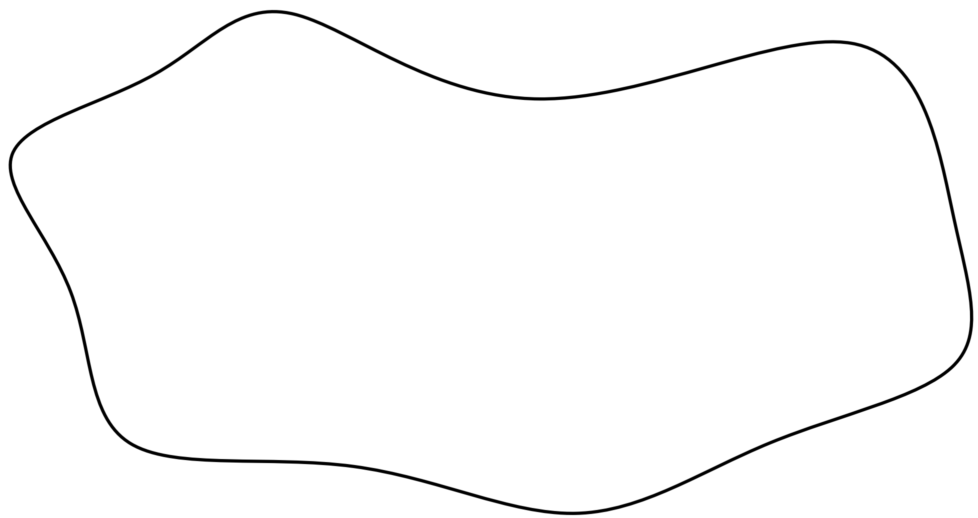
Reasoning About the Heap

- Separation Logic
  - Non-classical logic to reason about shared mutable data-structures
  - `emp`, $\mathtt{x} \longrightarrow 3$



Reasoning About the Heap

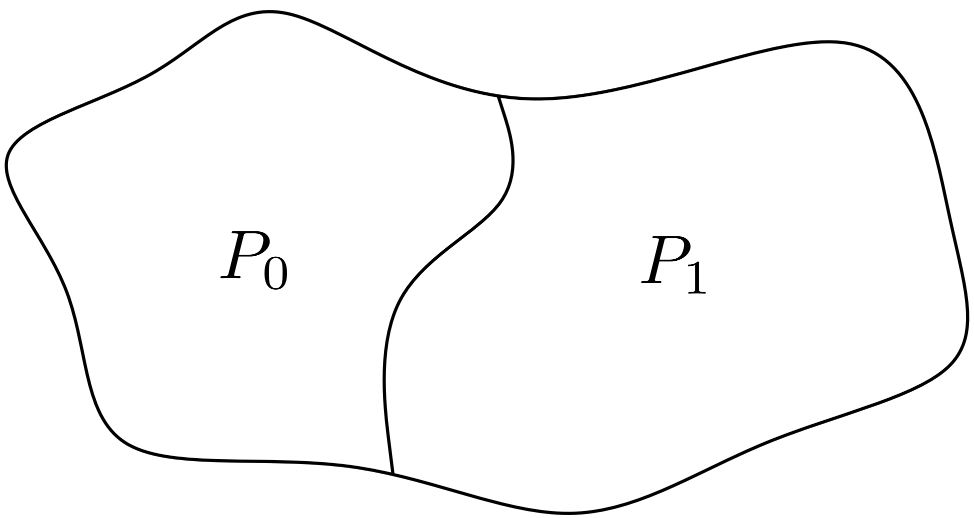
- Separation Logic
  - Non-classical logic to reason about shared mutable data-structures
  - `emp`, `x` $\longrightarrow$ 3, $*$



$$[P_0 * P_1]\ s\ h \stackrel{\text{def}}{=}$$
$$\exists h_0, h_1\ .\ h_1 \perp h_0 \wedge h_0 \cup h_1 = h \wedge [P_0]\ s\ h_0 \wedge [P_1]\ s\ h_1$$

Reasoning About the Heap

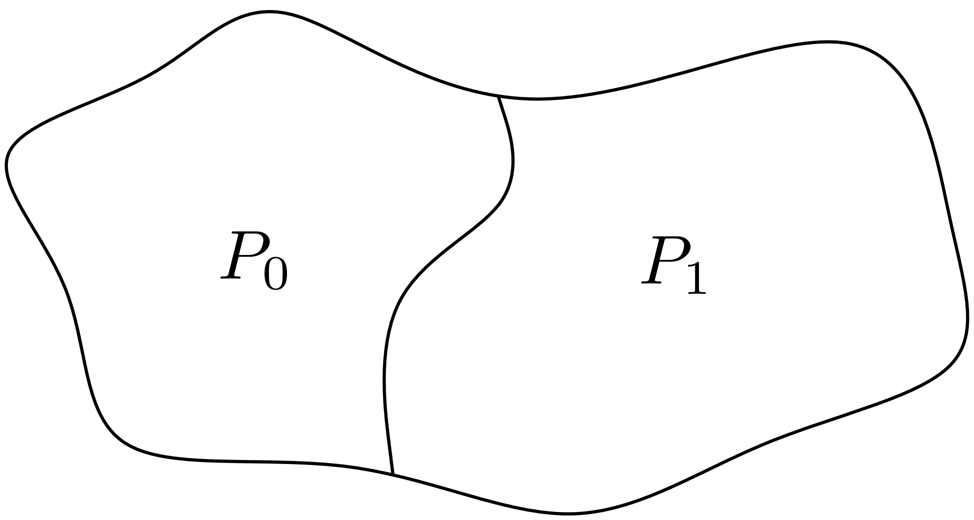
- Separation Logic
  - Non-classical logic to reason about shared mutable data-structures
  - `emp`, $\mathtt{x} \longrightarrow 3$, $*$



$$[P_0 * P_1]\ s\ h \stackrel{\text{def}}{=}$$
$$\exists h_0, h_1\ .\ h_1 \perp h_0 \wedge h_0 \cup h_1 = h \wedge [P_0]\ s\ h_0 \wedge [P_1]\ s\ h_1$$

Reasoning About the Heap

- Separation Logic
  - Non-classical logic to reason about shared mutable data-structures
  - $\texttt{emp}$, $\texttt{x} \longrightarrow 3$, $*$, $-\!\!*$



$$[P_0 * P_1]\ s\ h \stackrel{\text{def}}{=} \exists h_0, h_1\ .\ h_1 \perp h_0 \wedge h_0 \cup h_1 = h \wedge [P_0]\ s\ h_0 \wedge [P_1]\ s\ h_1$$

Reasoning About the Heap

- Regional Logic

Reasoning About the Heap

- Regional Logic
  - Classical Logic
  - Ghost fields and variables
  - Explicit region manipulation: `emp`, $\cup$, $\cap$, $\#$, $\setminus$
  - Region assertion language: $R_1 \subseteq R_2$, $R_1 \# R_2$, $R_1.f \subseteq R_2$

Verification Diagrams

$$P \models \varphi$$

Verification Diagrams

$$P \models \varphi$$

$$\mathcal{D}$$

Verification Diagrams

- A system P is a Fair Transition System

$$P \models \varphi$$

$$\mathcal{D}$$

Verification Diagrams

- A system P is a Fair Transition System
- Verification Diagrams are Sound and Complete

$$P \models \varphi$$

$$\mathcal{D}$$

Verification Diagrams

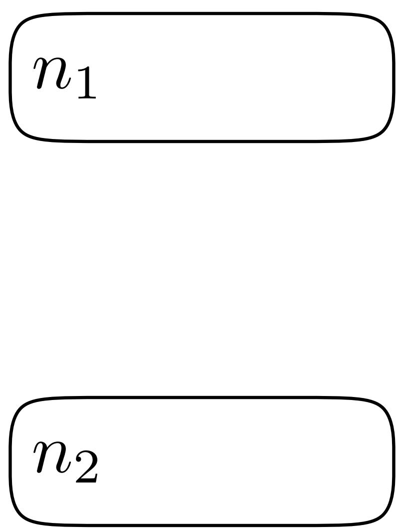
- A system P is a Fair Transition System
- Verification Diagrams are Sound and Complete

$$\mathcal{D} : \langle N, N_0, E, \mu, \mathcal{F}, \eta, \Delta, f \rangle$$

Verification Diagrams

- A system P is a Fair Transition System
- Verification Diagrams are Sound and Complete

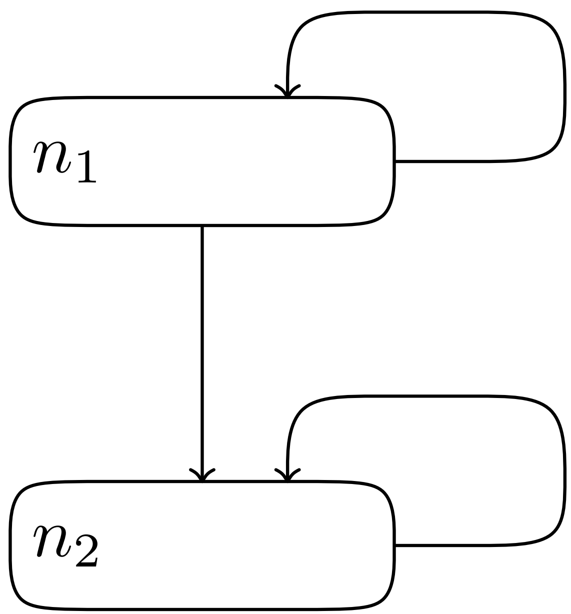
$$\mathcal{D} : \langle N, N_0, E, \mu, \mathcal{F}, \eta, \Delta, f \rangle$$



Verification Diagrams

- A system P is a Fair Transition System
- Verification Diagrams are Sound and Complete

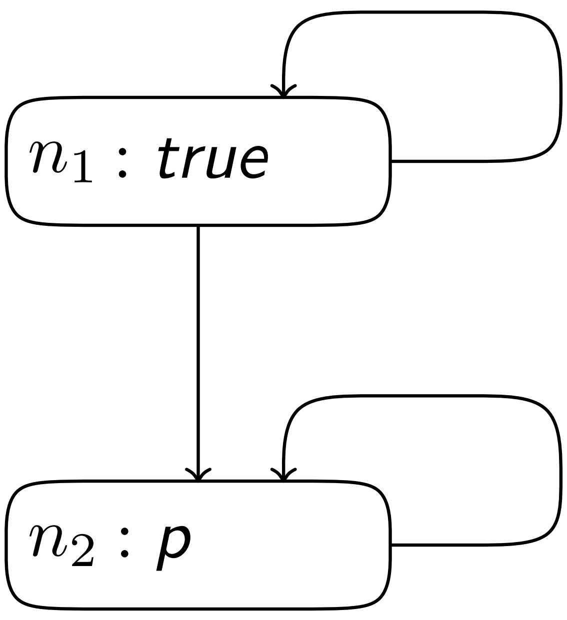
$$\mathcal{D} : \langle N, N_0, E, \mu, \mathcal{F}, \eta, \Delta, f \rangle$$



Verification Diagrams

- A system P is a Fair Transition System
- Verification Diagrams are Sound and Complete

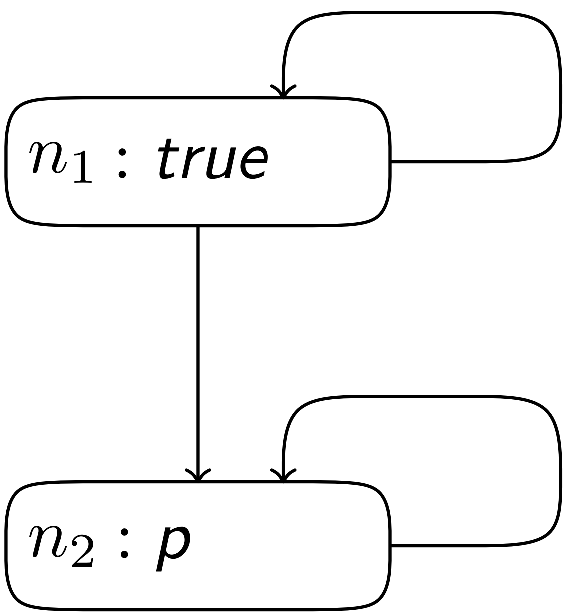
$$\mathcal{D} : \langle N, N_0, E, \mu, \mathcal{F}, \eta, \Delta, f \rangle$$



Verification Diagrams

- A system P is a Fair Transition System
- Verification Diagrams are Sound and Complete

$$\mathcal{D} : \langle N, N_0, E, \mu, \mathcal{F}, \eta, \Delta, f \rangle$$

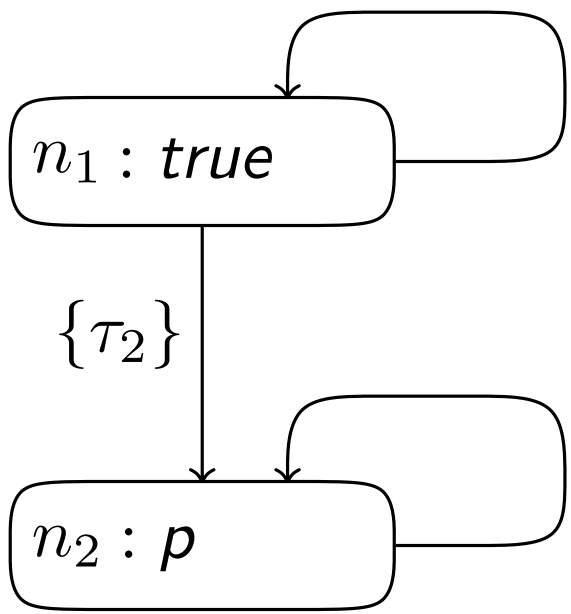


$$\Diamond \Box p$$

Verification Diagrams

- A system P is a Fair Transition System
- Verification Diagrams are Sound and Complete

$$\mathcal{D} : \langle N, N_0, E, \mu, \mathcal{F}, \eta, \Delta, f \rangle$$

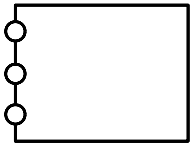


$$\Diamond \Box p$$

Verification of Concurrent Data-structures

Main Idea

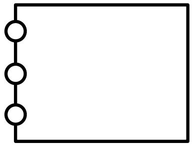
Concurrent DataStructure



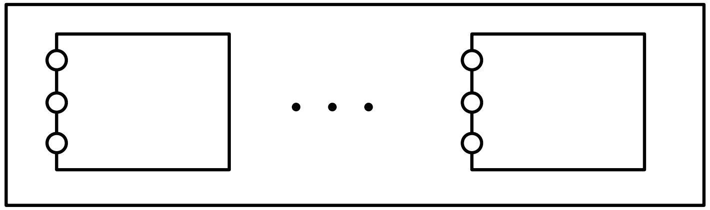
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



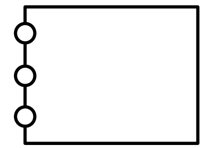
Most General Client



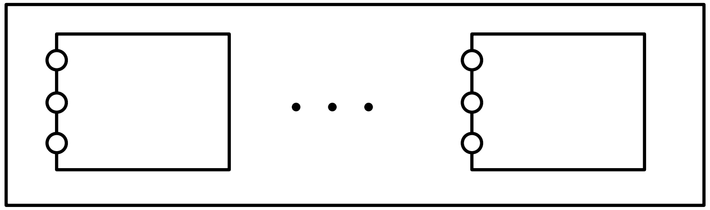
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client

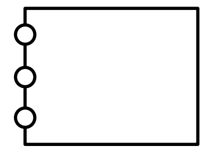


$$P[N] : P(1)||\cdots||P(N)$$

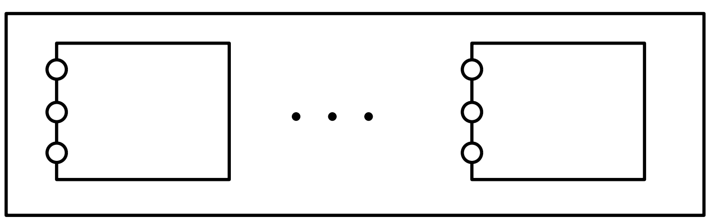
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client



$$P[N] : P(1)||\cdots||P(N)$$

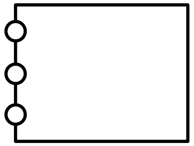
$$+$$

ghost variables

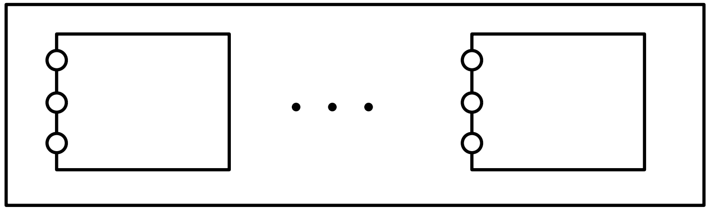
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client



$$P[N] : P(1) || \cdots || P(N)$$

$+$

ghost variables

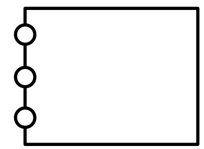
Property

$$\varphi^{(k)}$$

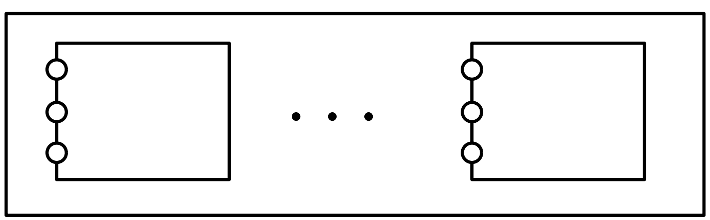
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client



$$P[N] : P(1) || \cdots || P(N)$$

$+$

ghost variables

Diagram

$\mathcal{D}$

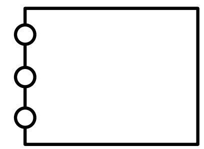
Property

$\varphi^{(k)}$

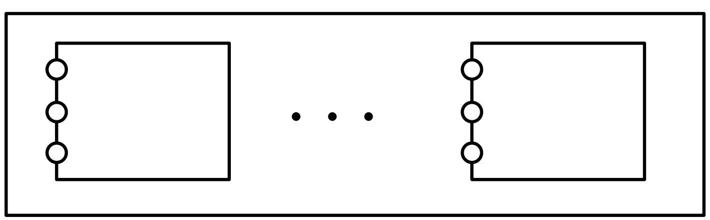
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client



$$P[N] : P(1)||\cdots||P(N)$$

$+$

ghost variables

Diagram

Property

$\models$ $\mathcal{D}$ $\models$ $\varphi^{(k)}$

Verification Conditions:

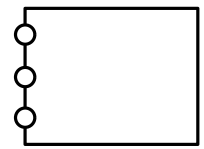
- Initiation
- Consecution
- Acceptance
- Fairness

Satisfaction
(Model Checking)

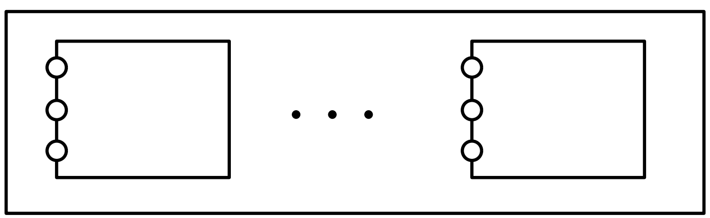
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client



$P[N] : P(1) || \cdots || P(N)$

$+$

ghost variables

Diagram

$\vDash \quad \mathcal{D} \quad \vDash \quad \varphi^{(k)}$

Property

Verification Conditions:

- Initiation
- Consecution
- Acceptance
- Fairness

Decision Procedures

Satisfaction (Model Checking)

Verification Conditions

Verification Conditions

- *Initiation*

$$\Theta \longrightarrow \mu(N_0)$$

Verification Conditions

- *Initiation*

$$\Theta \rightarrow \mu(N_0)$$

- *Consecution*: for all n and τ:

$$\mu(n)(s) \wedge \rho_\tau(s, s') \rightarrow \mu(\textit{next}(n))(s')$$

Verification Conditions

- *Initiation*

  $$\Theta \to \mu(N_0)$$

- *Consecution*: for all n and τ:

  $$\mu(n)(s) \wedge \rho_\tau(s, s') \to \mu(\mathit{next}(n))(s')$$

- *Acceptance*: if $(n_1, n_2) \in P \setminus R$ then

  $$\mu(n_1)(s) \wedge \mu(n_2)(s') \wedge \rho_\tau(s, s') \to \delta_{n_1}(s) \geq \delta_{n_2}(s')$$

  and if $(n_1, n_2) \notin P \cup R$:

  $$\mu(n_1)(s) \wedge \mu(n_2)(s') \wedge \rho_\tau(s, s') \to \delta_{n_1}(s) > \delta_{n_2}(s')$$

Verification Conditions

- *Initiation*

$$\Theta \rightarrow \mu(N_0)$$

- *Consecution*: for all n and τ:

$$\mu(n)(s) \wedge \rho_\tau(s, s') \rightarrow \mu(\textit{next}(n))(s')$$

- *Acceptance*: if $(n_1, n_2) \in P \setminus R$ then

$$\mu(n_1)(s) \wedge \mu(n_2)(s') \wedge \rho_\tau(s, s') \rightarrow \delta_{n_1}(s) \geq \delta_{n_2}(s')$$

and if $(n_1, n_2) \notin P \cup R$:

$$\mu(n_1)(s) \wedge \mu(n_2)(s') \wedge \rho_\tau(s, s') \rightarrow \delta_{n_1}(s) > \delta_{n_2}(s')$$

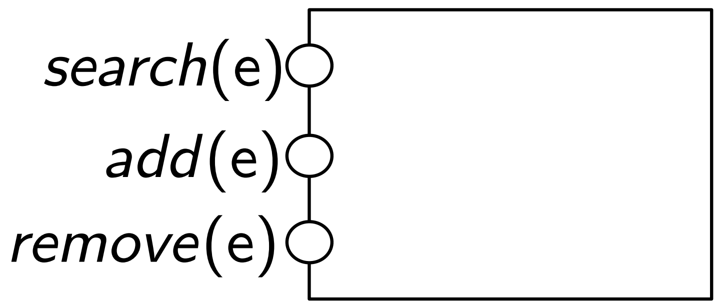
- *Fairness*: for all n and $\tau \in \eta(n, n')$:

$$\mu(n)(s) \rightarrow En_\tau(s)$$

$$\mu(n)(s) \wedge \rho_\tau(s, s') \rightarrow \mu(\tau(n))(s')$$

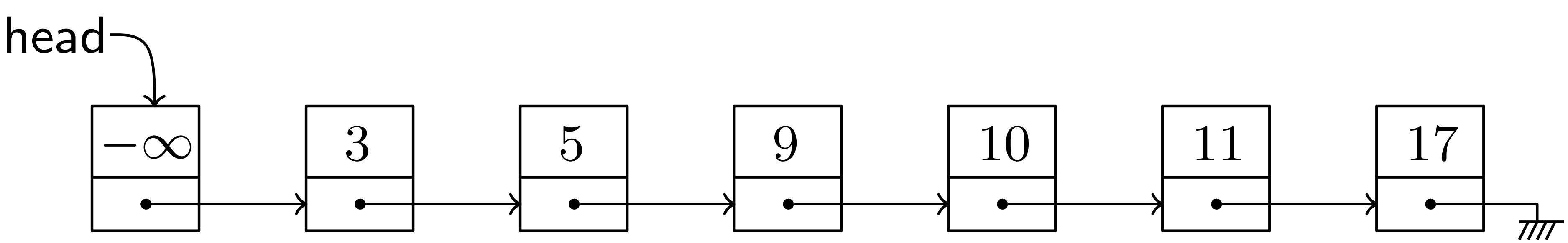
Lock-coupling Lists

- A concurrent implementation of sets



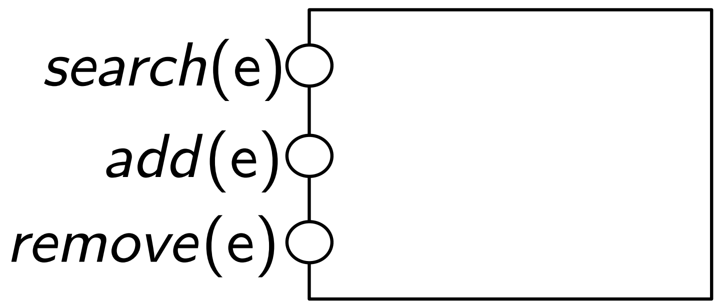

```
class List {
      Node list;
}
```

```
class Node {
      Value val;
      Node next;
      Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

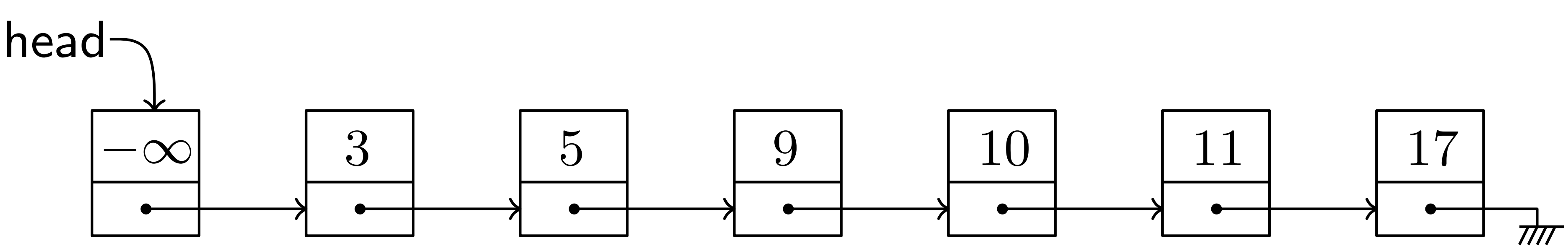


Most general client

```
while true
    e:= NonDet();
    select
        search(e);
    or
        add(e);
    or
        remove(e);
end;
```

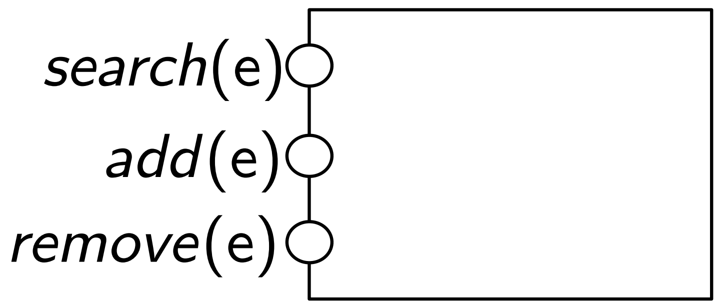
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

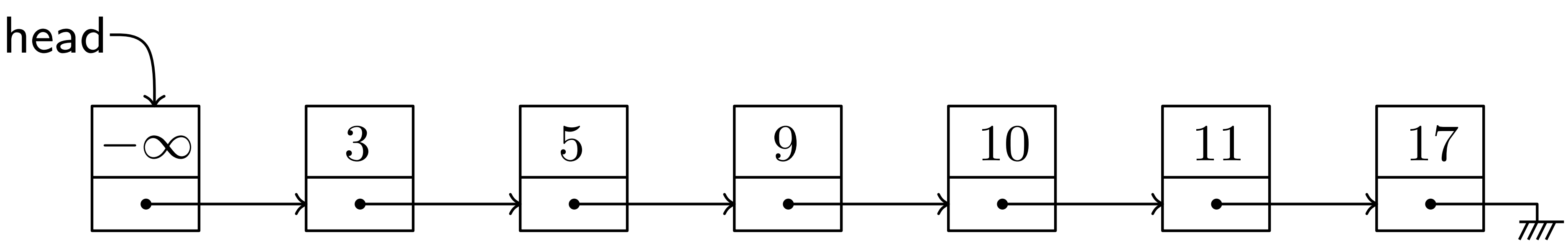


Locate

```
prev := Head;
prev.lock();
curr := prev.next;
curr.lock();
while curr.val < e do
    prev.unlock();
    prev := curr;
    curr := curr.next;
    curr.lock();
end;
return (prev,curr)
```

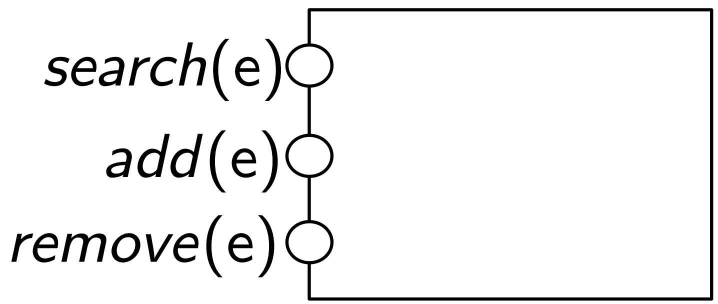
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

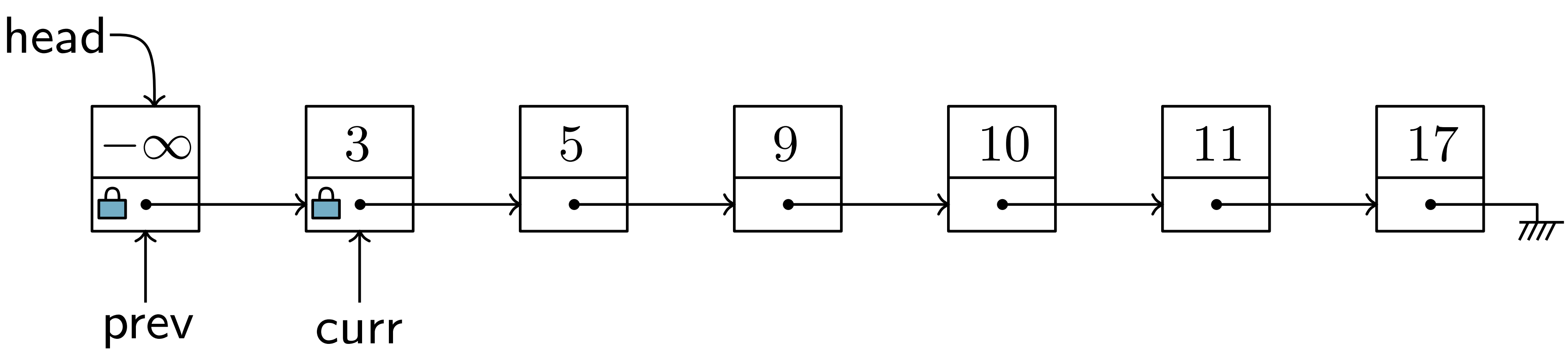


Locate 8

```
prev := Head;
prev.lock();
curr := prev.next;
curr.lock();
while curr.val < e do
    prev.unlock();
    prev := curr;
    curr := curr.next;
    curr.lock();
end;
return (prev,curr)
```

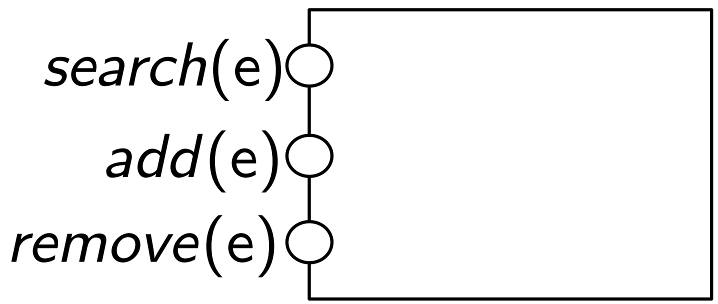
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

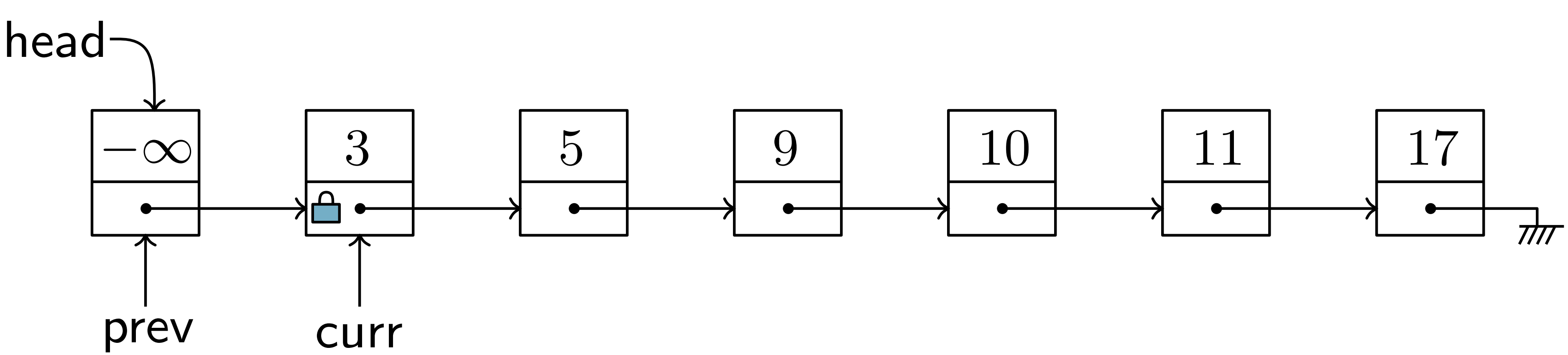


Locate 8

```
prev := Head;
prev.lock();
curr := prev.next;
curr.lock();
while curr.val < e do
    prev.unlock();
    prev := curr;
    curr := curr.next;
    curr.lock();
end;
return (prev,curr)
```

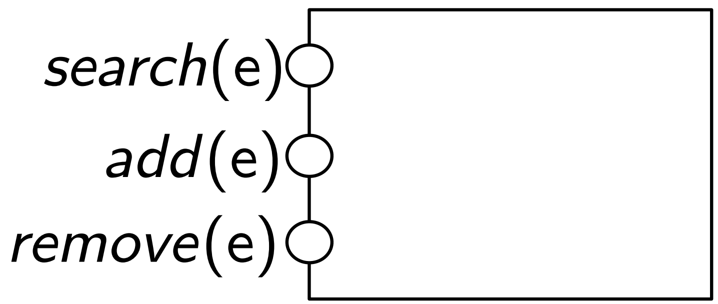
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

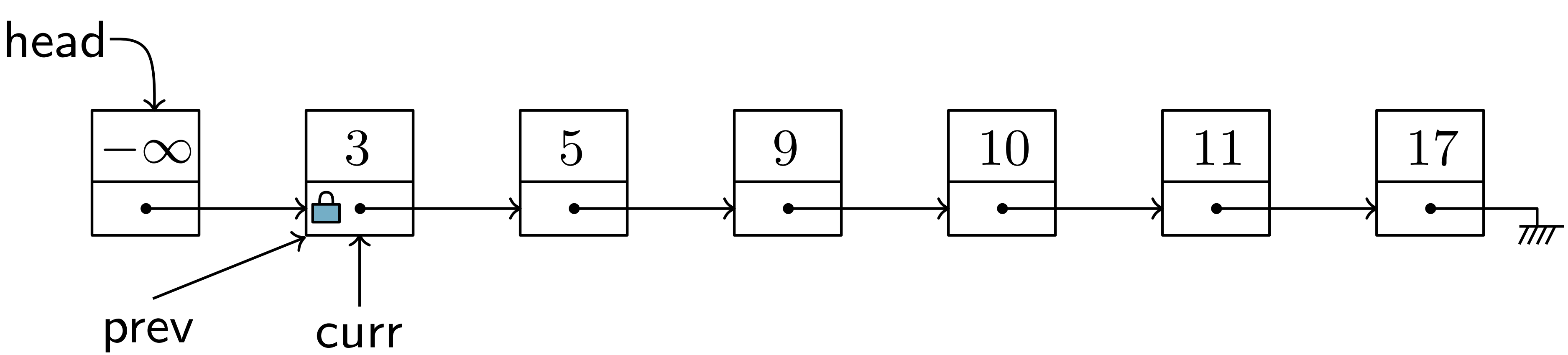


Locate 8

```
prev := Head;
prev.lock();
curr := prev.next;
curr.lock();
while curr.val < e do
    prev.unlock();
    prev := curr;
    curr := curr.next;
    curr.lock();
end;
return (prev,curr)
```

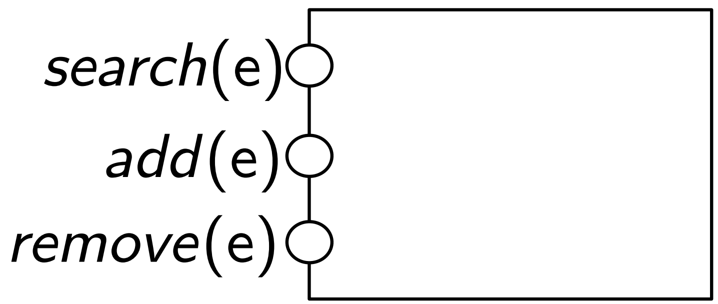
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

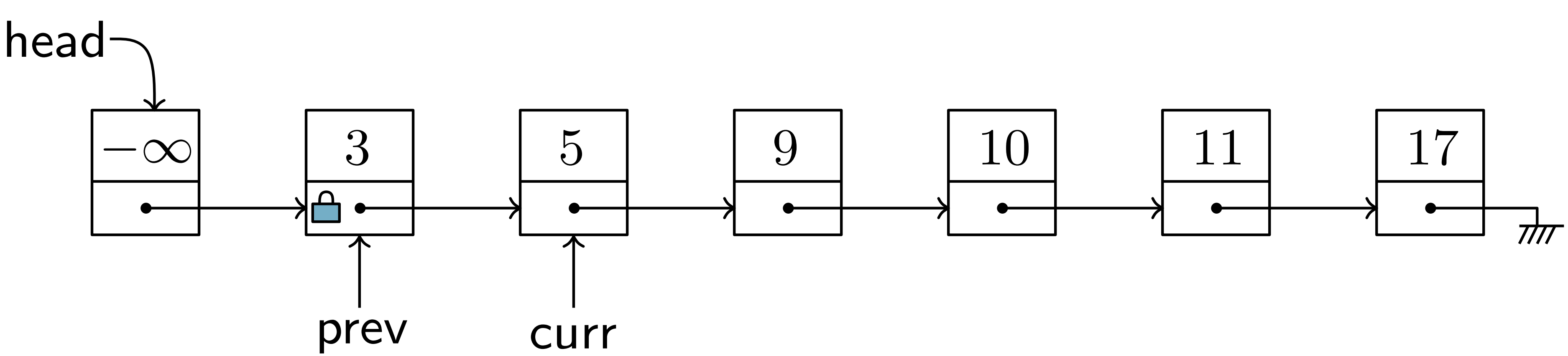


Locate 8

```
prev := Head;
prev.lock();
curr := prev.next;
curr.lock();
while curr.val < e do
    prev.unlock();
    prev := curr;
    curr := curr.next;
    curr.lock();
end;
return (prev,curr)
```

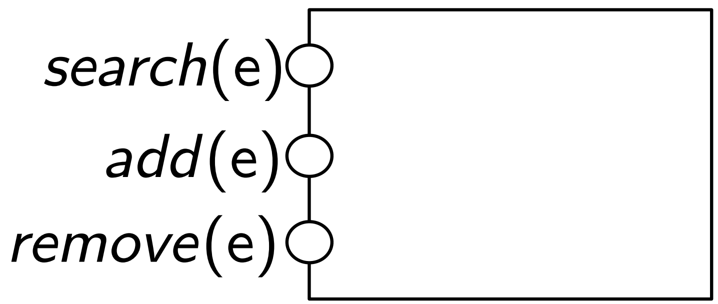
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

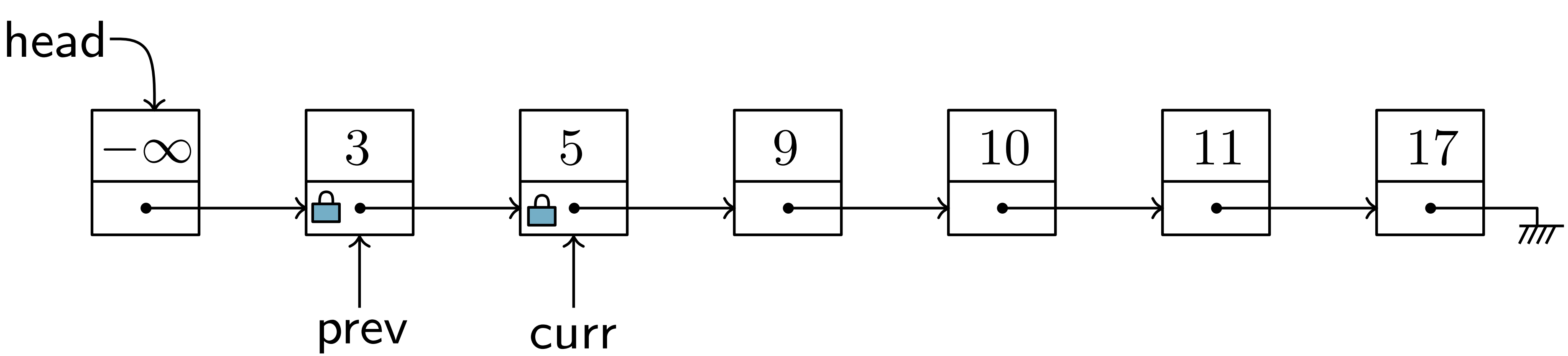


Locate 8

```
prev := Head;
prev.lock();
curr := prev.next;
curr.lock();
while curr.val < e do
    prev.unlock();
    prev := curr;
    curr := curr.next;
    curr.lock();
end;
return (prev,curr)
```

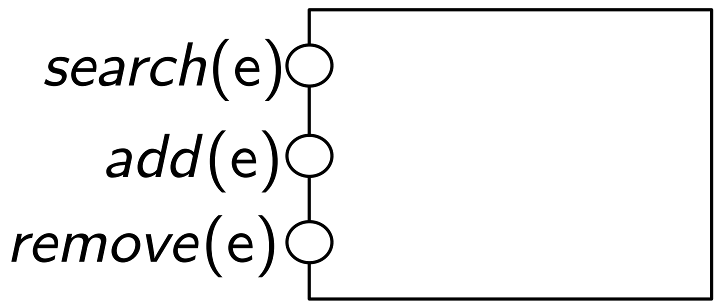
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

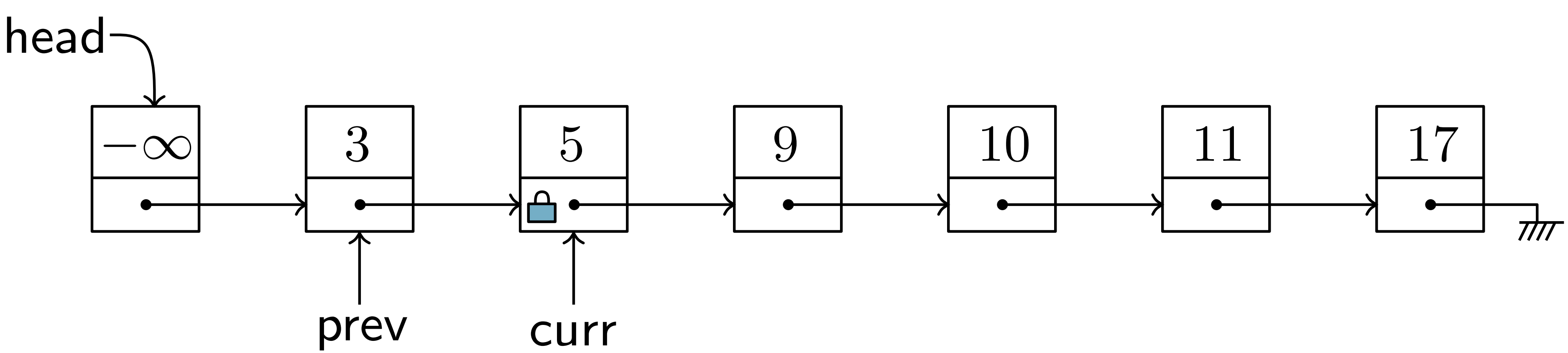


Locate 8

```
prev := Head;
prev.lock();
curr := prev.next;
curr.lock();
while curr.val < e do
    prev.unlock();
    prev := curr;
    curr := curr.next;
    curr.lock();
end;
return (prev,curr)
```

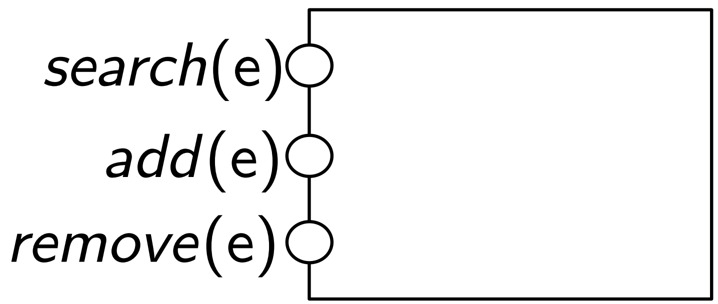
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

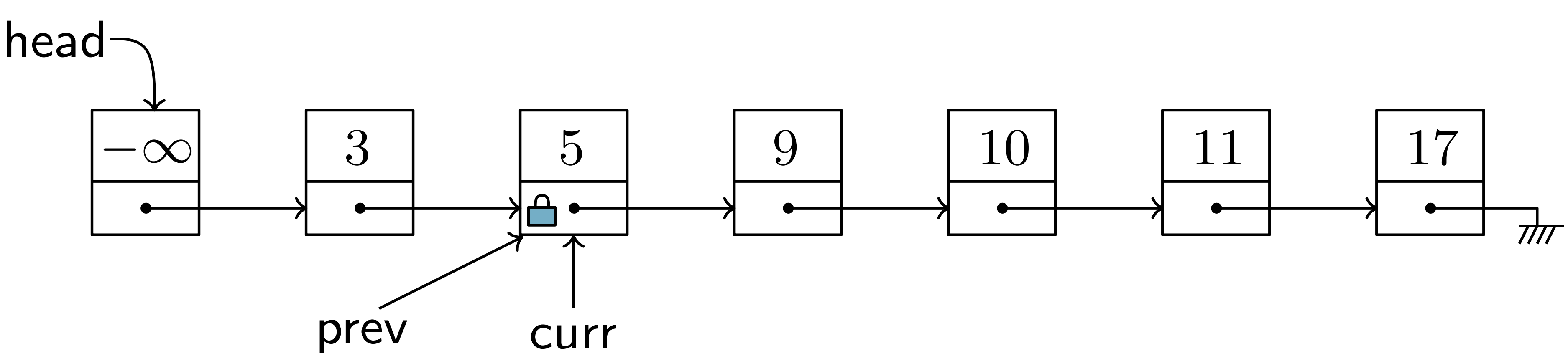


Locate 8

```
prev := Head;
prev.lock();
curr := prev.next;
curr.lock();
while curr.val < e do
    prev.unlock();
    prev := curr;
    curr := curr.next;
    curr.lock();
end;
return (prev,curr)
```

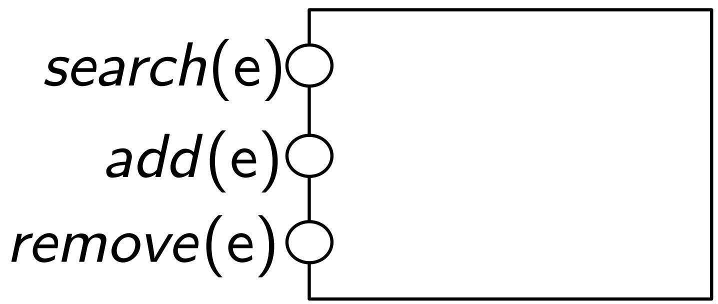
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

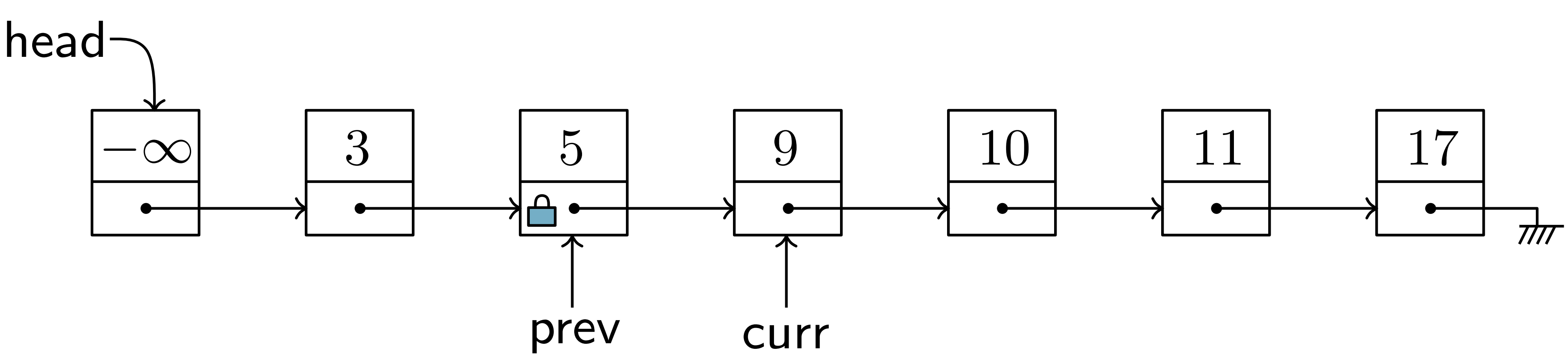


Locate 8

```
prev := Head;
prev.lock();
curr := prev.next;
curr.lock();
while curr.val < e do
    prev.unlock();
    prev := curr;
    curr := curr.next;
    curr.lock();
end;
return (prev,curr)
```

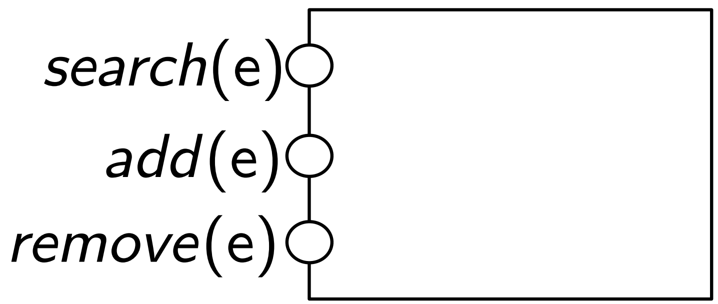
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

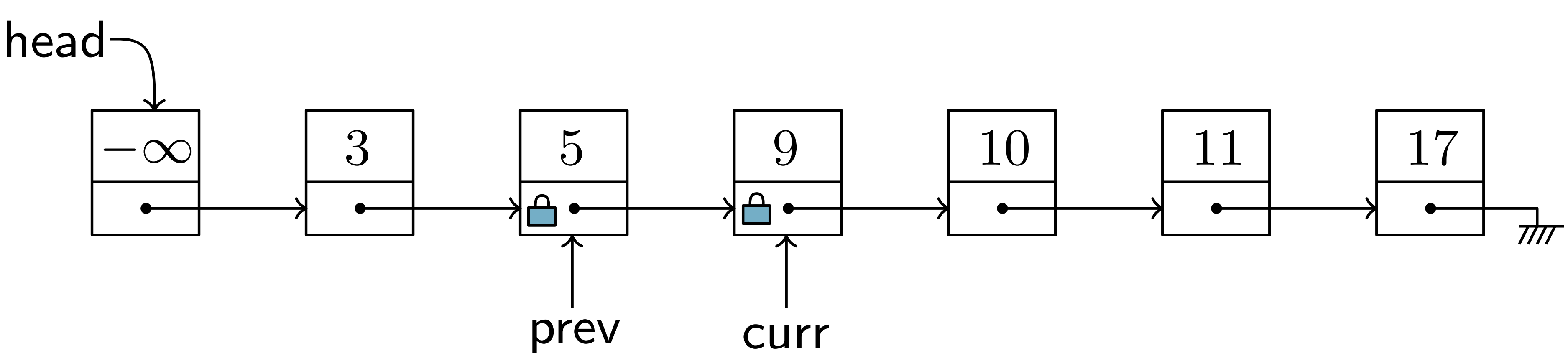


Locate 8

```
prev := Head;
prev.lock();
curr := prev.next;
curr.lock();
while curr.val < e do
    prev.unlock();
    prev := curr;
    curr := curr.next;
    curr.lock();
end;
return (prev,curr)
```

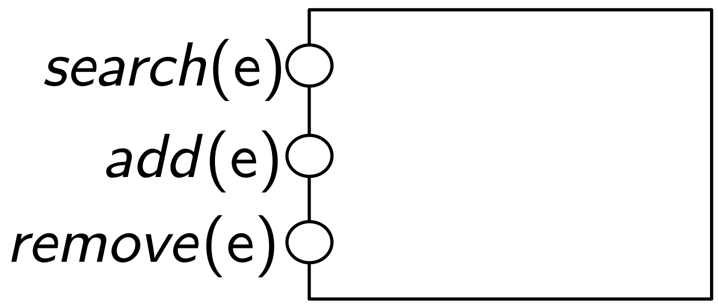
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

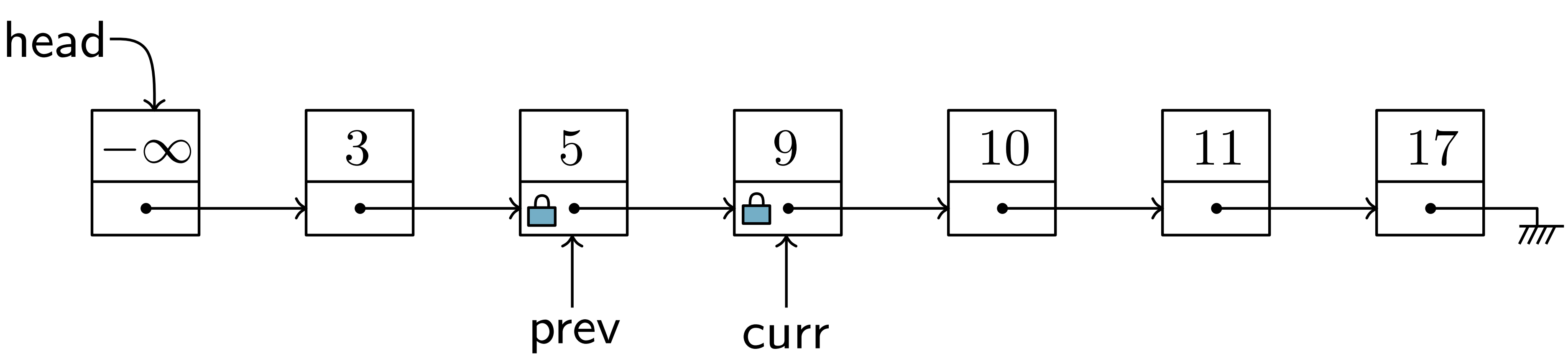


Search 8

```
prev, cur := locate (e);
if curr.val = e then
    result := true;
else
    result := false;
curr.unlock();
prev.unlock();
return result;
```

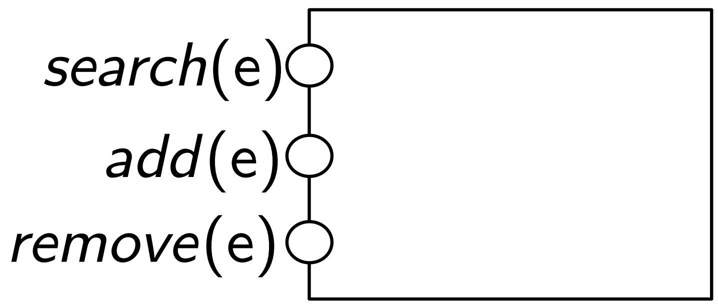
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

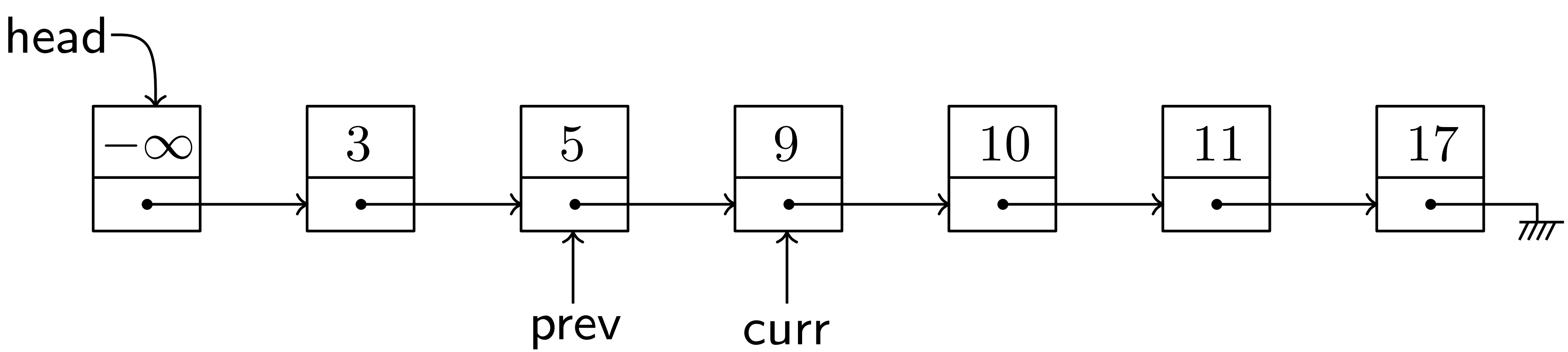


Search 8

```
prev, cur := locate (e);
if curr.val = e then
    result := true;
else
    result := false;
curr.unlock();
prev.unlock();
return result;
```

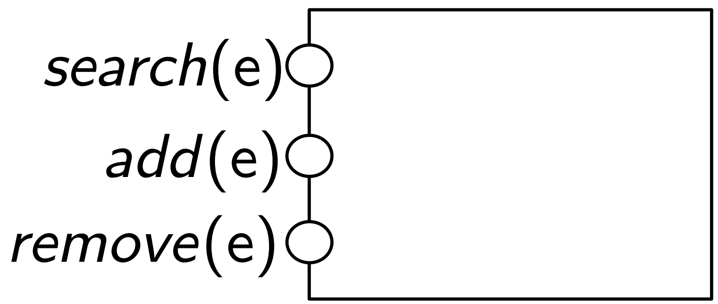
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

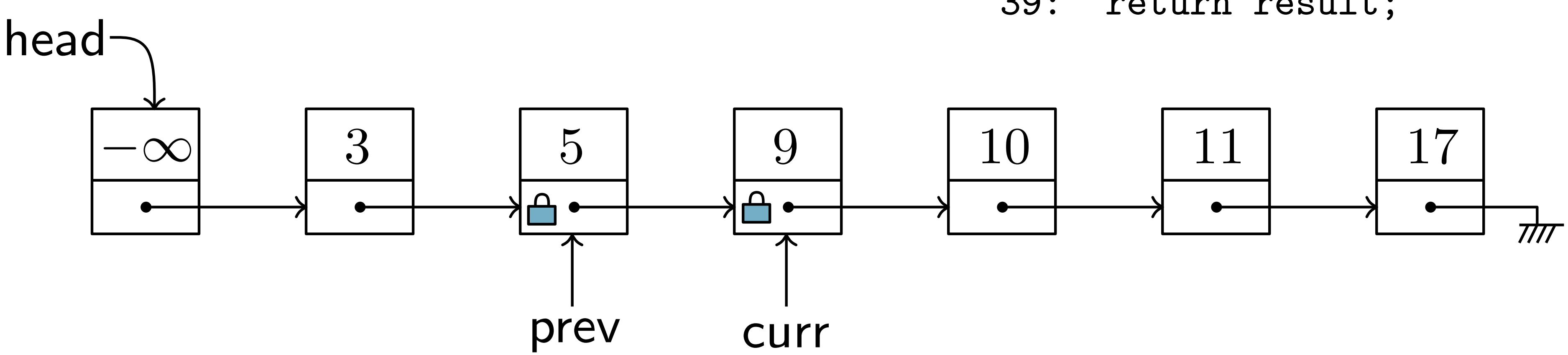


Add 8

```
prev, curr := locate(e);
if curr.val != e then
     aux := new Node(e);
     aux.next := curr;
     prev.next := aux;
     result := true;
else
     result := false;
end;
prev.unlock();
curr.unlock();
return result;
```

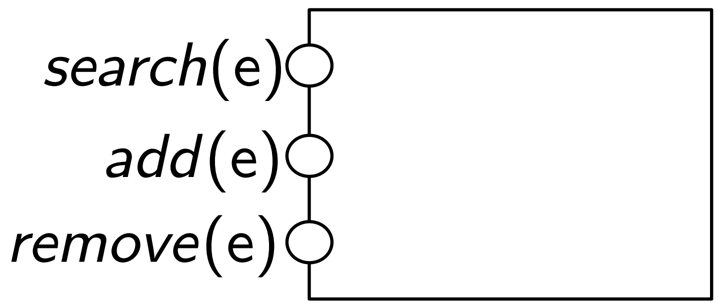
```
class List {
     Node list;
}
```

```
class Node {
     Value val;
     Node next;
     Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

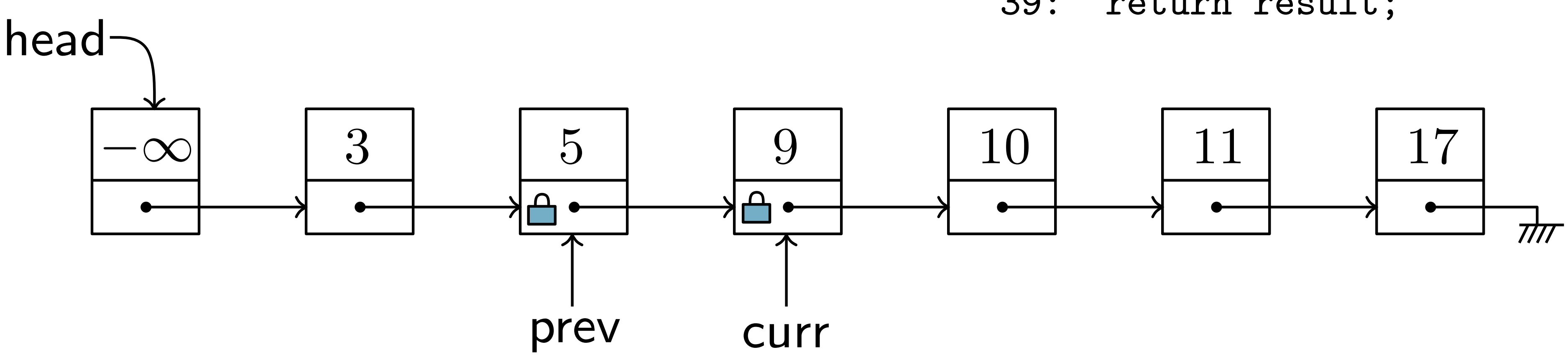


Add 8

```
prev, curr := locate(e);
if curr.val != e then
    aux := new Node(e);
    aux.next := curr;
    prev.next := aux;
    result := true;
else
    result := false;
end;
prev.unlock();
curr.unlock();
return result;
```

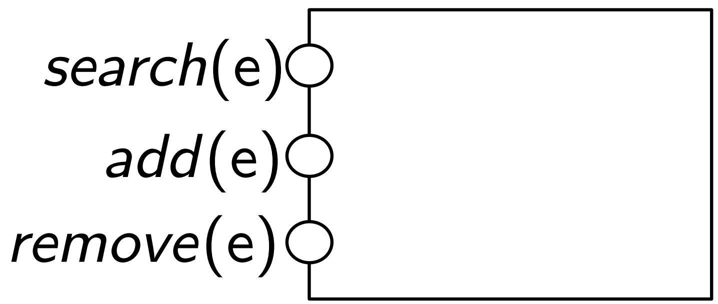
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

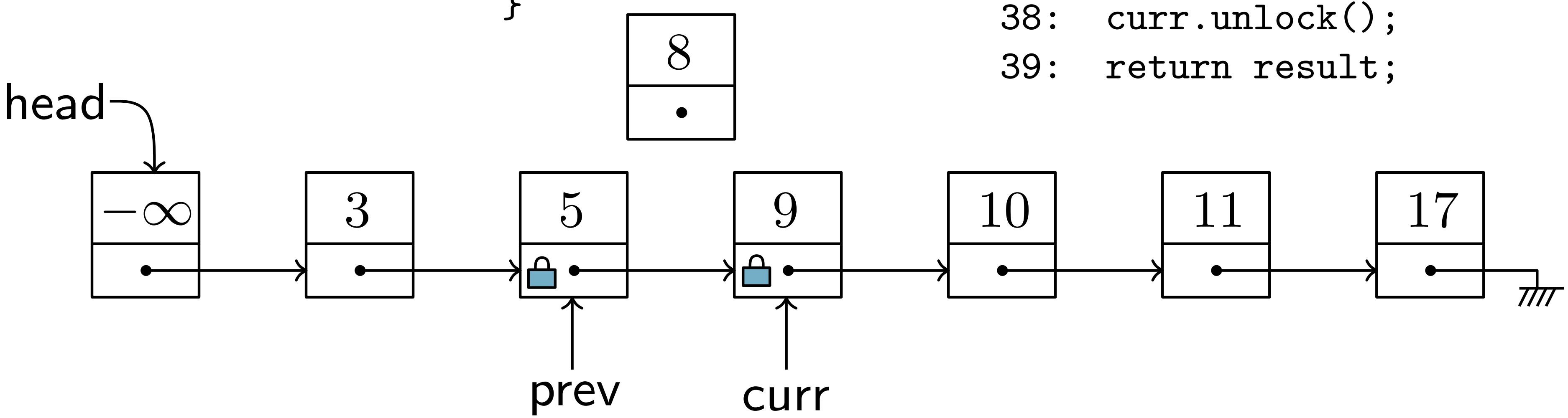


Add 8

```
prev, curr := locate(e);
if curr.val != e then
    aux := new Node(e);
    aux.next := curr;
    prev.next := aux;
    result := true;
else
    result := false;
end;
prev.unlock();
curr.unlock();
return result;
```

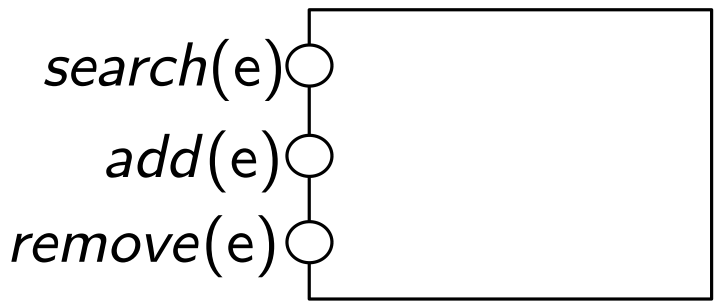
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

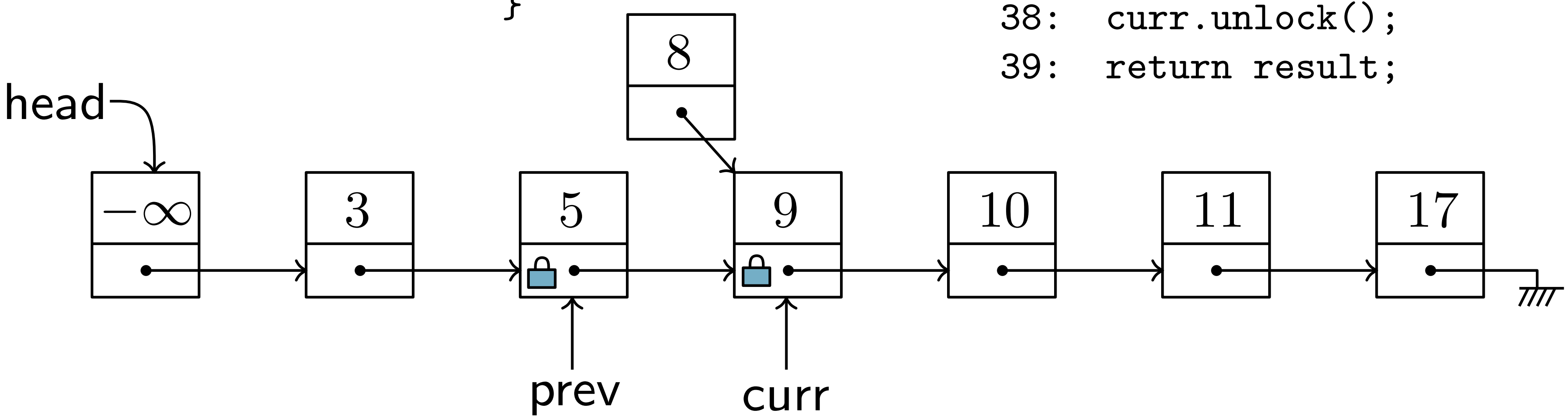


Add 8

```
prev, curr := locate(e);
if curr.val != e then
    aux := new Node(e);
    aux.next := curr;
    prev.next := aux;
    result := true;
else
    result := false;
end;
prev.unlock();
curr.unlock();
return result;
```

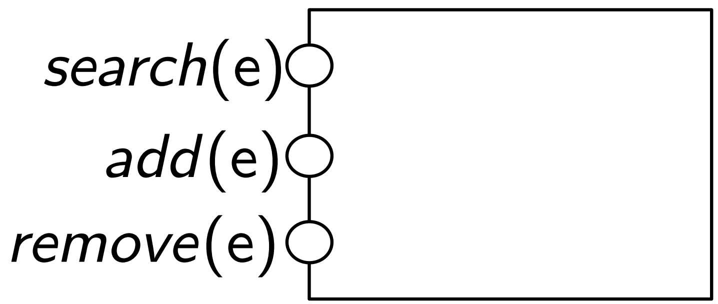
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

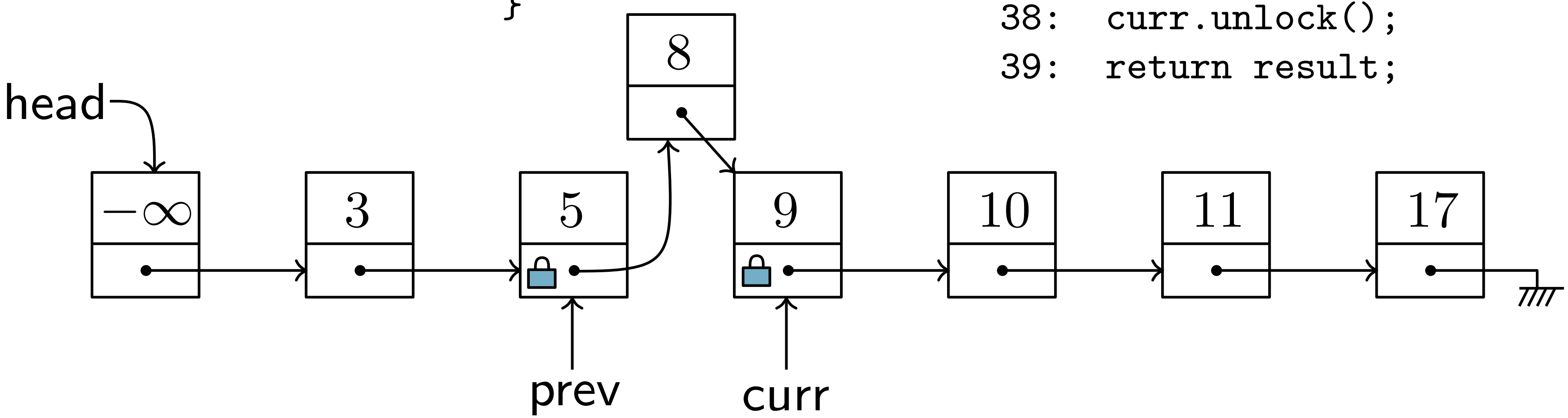


Add 8

```
prev, curr := locate(e);
if curr.val != e then
    aux := new Node(e);
    aux.next := curr;
    prev.next := aux;
    result := true;
else
    result := false;
end;
prev.unlock();
curr.unlock();
return result;
```

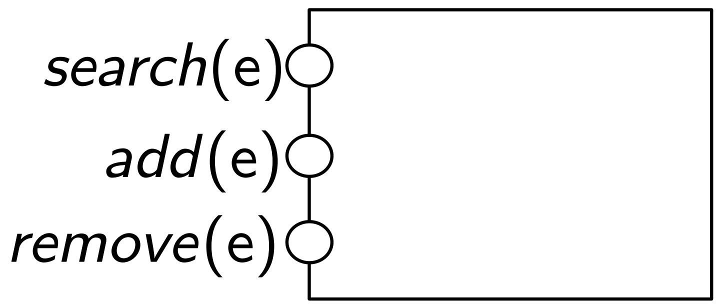
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

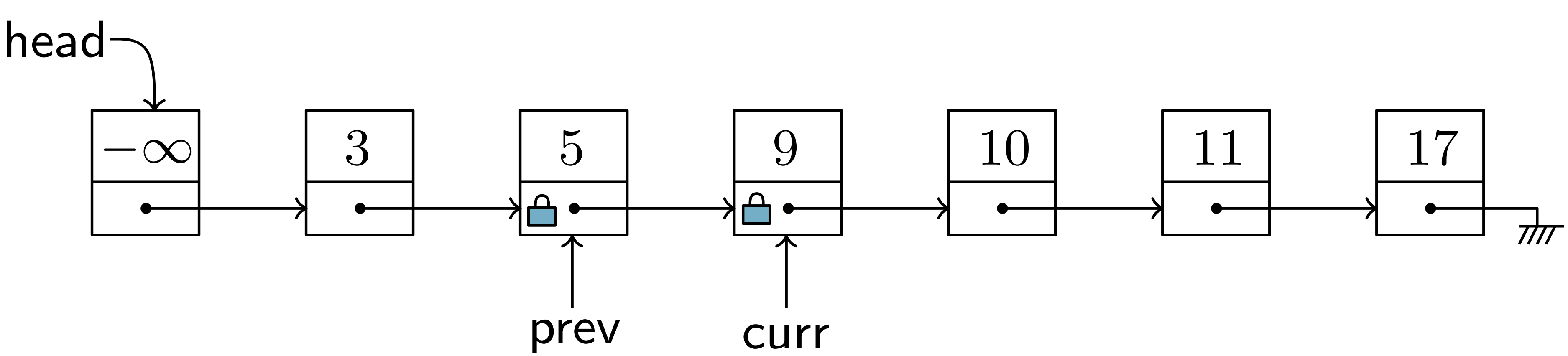


Remove 9

```
prev, curr := locate(e);
if curr.val == e then
    aux := curr.next;
    prev.next := aux;
    result := true;
else
    result := false;
end;
prev.unlock();
curr.unlock();
return result;
```

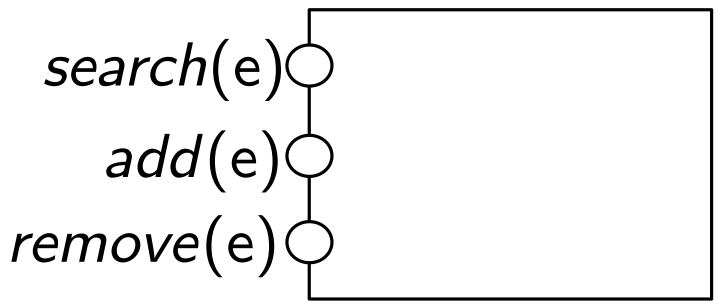
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

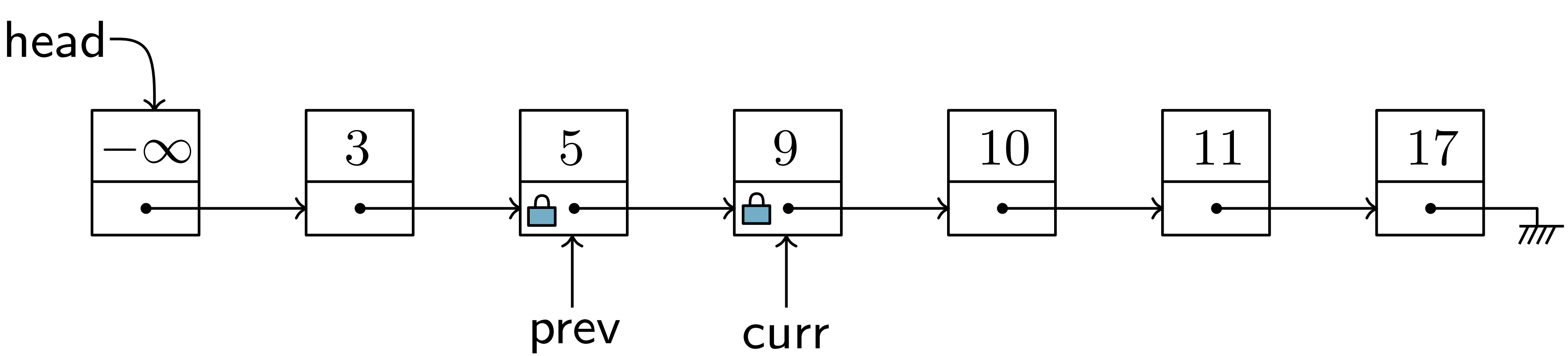


Remove 9

```
prev, curr := locate(e);
if curr.val == e then
    aux := curr.next;
    prev.next := aux;
    result := true;
else
    result := false;
end;
prev.unlock();
curr.unlock();
return result;
```

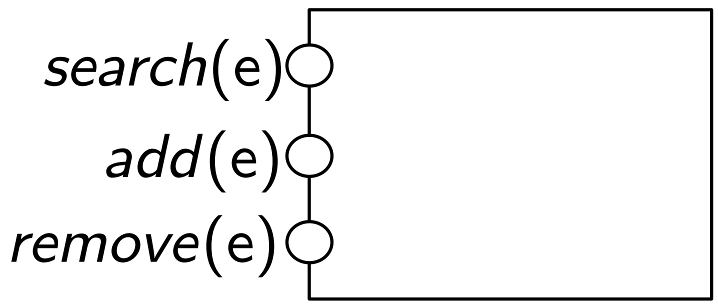
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

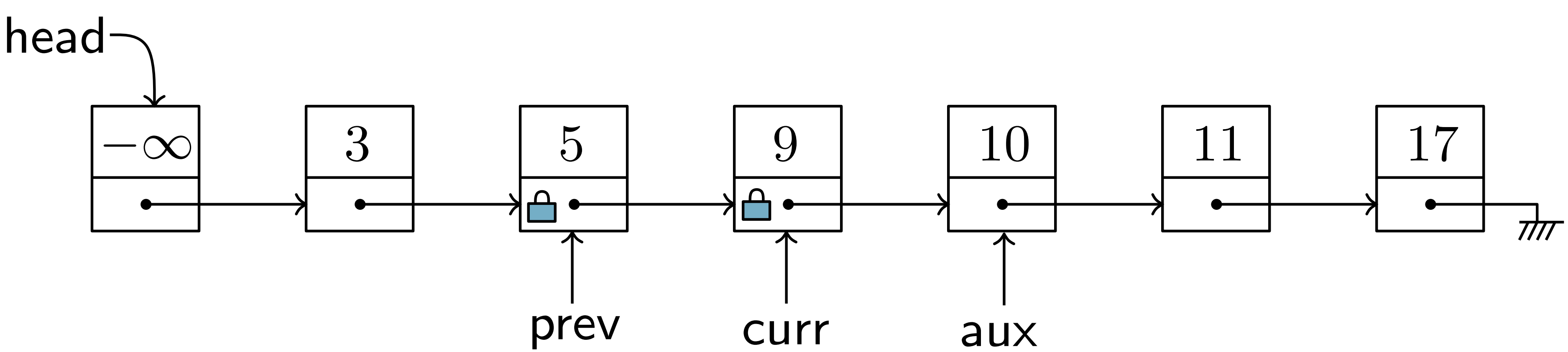


Remove 9

```
prev, curr := locate(e);
if curr.val == e then
    aux := curr.next;
    prev.next := aux;
    result := true;
else
    result := false;
end;
prev.unlock();
curr.unlock();
return result;
```

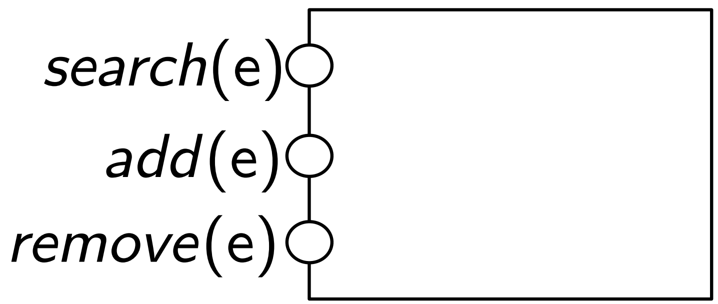
```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Lock-coupling Lists

- A concurrent implementation of sets

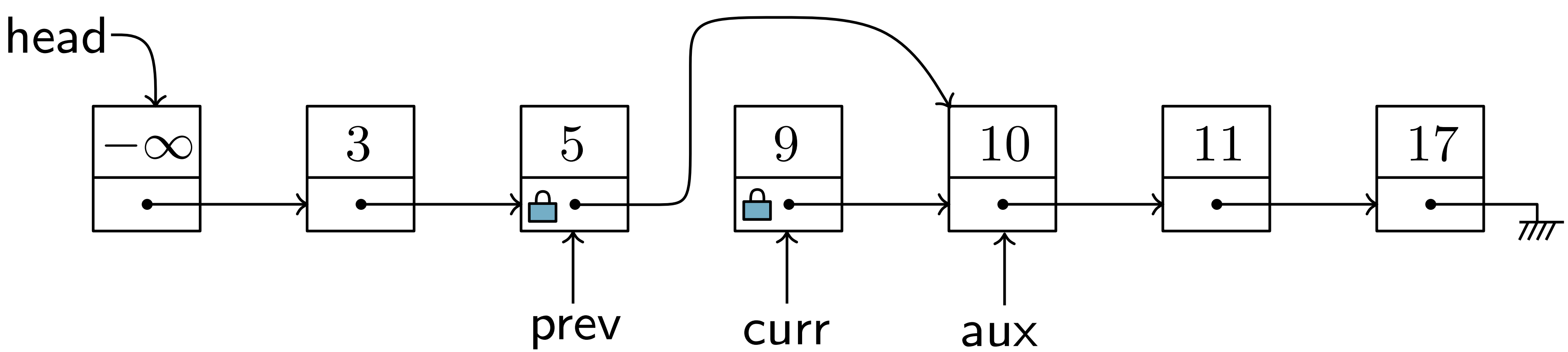


Remove 9

```
prev, curr := locate(e);
if curr.val == e then
    aux := curr.next;
    prev.next := aux;
    result := true;
else
    result := false;
end;
prev.unlock();
curr.unlock();
return result;
```

```
class List {
    Node list;
}
```

```
class Node {
    Value val;
    Node next;
    Lock lock;
}
```



Proof example

- Leading thread terminates:

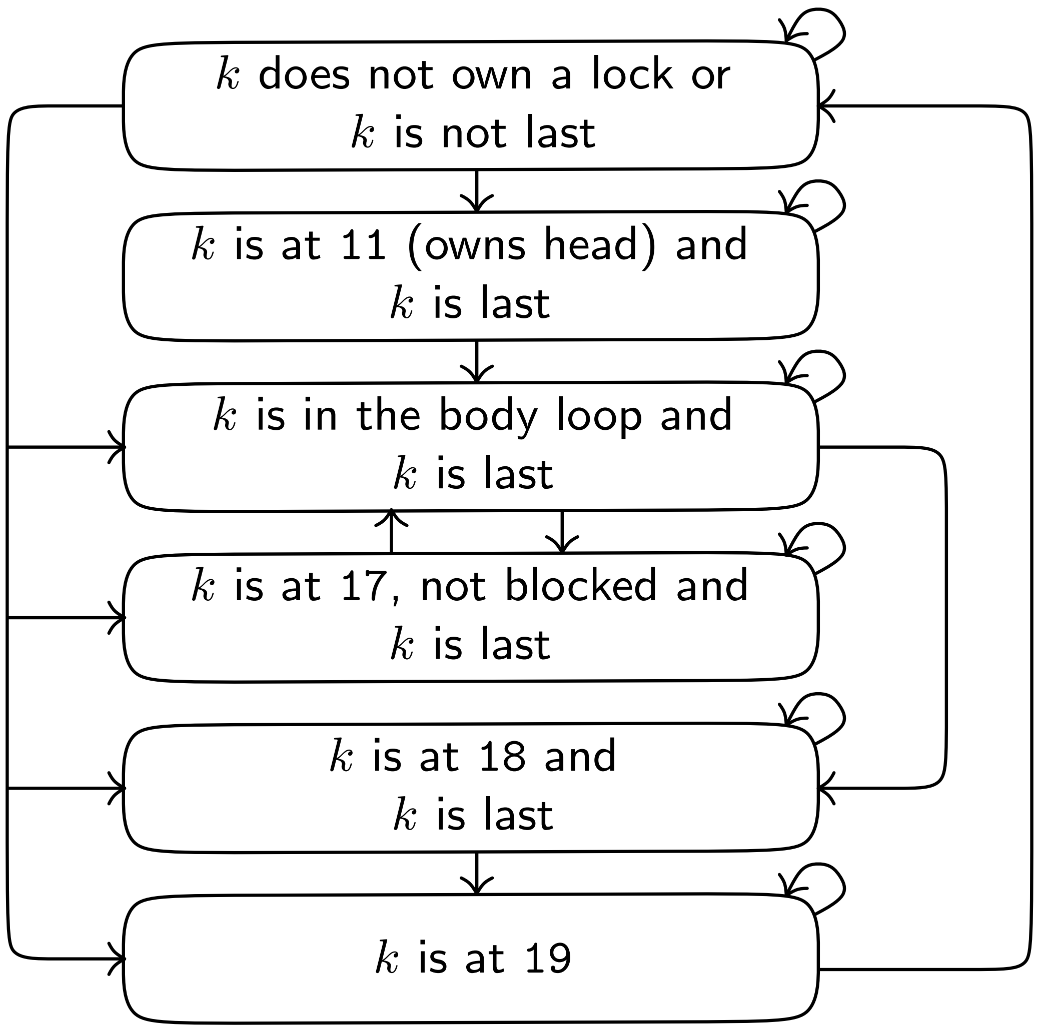
$$\varphi^{(k)} \stackrel{\text{def}}{=} \Box(at_{11}^{k} \land IsLast(k) \rightarrow \Diamond at_{19}^{k})$$

Proof example

- Leading thread terminates:

$$\varphi^{(k)} \stackrel{\text{def}}{=} \Box(at_{11}^{k} \wedge IsLast(k) \rightarrow \Diamond at_{19}^{k})$$

- Informal justification



Ghost variables

```
class List {
    Node list;
    rgn r;
}
```

```
prev, curr := locate(e);
if curr.val != e then
    aux := new Node(e);
    aux.next := curr;
    prev.next := aux; r:=r ∪ ⟨aux⟩
    result := true;
else
    result := false;
end;
prev.unlock();
curr.unlock();
return result;
```

```
prev, curr := locate(e);
if curr.val == e then
    aux := curr.next;
    prev.next := aux; r:=r ∪ −⟨curr⟩
    result := true;
else
    result := false;
end;
prev.unlock();
curr.unlock();
return result;
```

The Theory TLL3

Signature	Sorts	Functions
Σ_{cell}	cell elem addr thid	$error$: cell $mkcell$: elem × addr × thid → cell $\_.data$: cell → elem $\_.next$: cell → addr $\_.lockid$: cell → thid $\_.lock$: cell → thid → cell $\_.unlock$: cell → cell
Σ_{mem}	mem addr cell	$null$: addr $\_[\_]$: mem × addr → cell upd : mem × addr × cell → mem
$\Sigma_{\mathsf{Reachability}}$	mem addr path	ϵ : path $[\_]$: addr → path
Σ_{set}	addr set	$\emptyset$: set $\{\_\}$: addr → set $\cup, \cap, \setminus$: set × set → set
Σ_{setth}	thid setth	$\emptyset_T$: setth $\{\_\}_T$: thid → setth $\cup_T, \cap_T, \setminus_T$: setth × setth → setth
Σ_{Bridge}	mem addr set path	$path2set$: path → set $addr2set$: mem × addr → set $getp$: mem × addr × addr → path $firstlocked$: mem × path → addr

The Theory TLL3

Signature	Sorts	Predicates
Σ_{cell}	cell elem addr thid	
Σ_{mem}	mem addr cell	
$\Sigma_{\text{Reachability}}$	mem addr path	$append : \text{path} \times \text{path} \times \text{path}$ $reach : \text{mem} \times \text{addr} \times \text{addr} \times \text{path}$
Σ_{set}	addr set	$\in : \text{addr} \times \text{set}$ $\subseteq : \text{set} \times \text{set}$
Σ_{setth}	thid setth	$\in_T : \text{thid} \times \text{setth}$ $\subseteq_T : \text{setth} \times \text{setth}$
Σ_{Bridge}	mem addr set path	

Auxiliary functions

$List$: mem $\times$ addr $\times$ set
$List(h, a, r) \leftrightarrow null \in addr2set(h, a) \wedge r = path2set(getp(h, a, null))$

f_a : mem $\times$ addr $\to$ path
$f_a(h, n) = \begin{cases} \epsilon & \text{if } n = null \\ getp(h, h[n].next, null) & \text{if } n \neq null \end{cases}$

$LastMarked$: mem $\times$ path $\to$ addr
$LastMarked(m, p) = firstlocked(m, rev(p))$

$NoMarks$: mem $\times$ path
$NoMarks(m, p) \leftrightarrow firstlocked(m, p) = null$

$SomeMark$: mem $\times$ path
$SomeMark(m, p) \leftrightarrow firstlocked(m, p) \neq null$

Theories of Lists with Regions

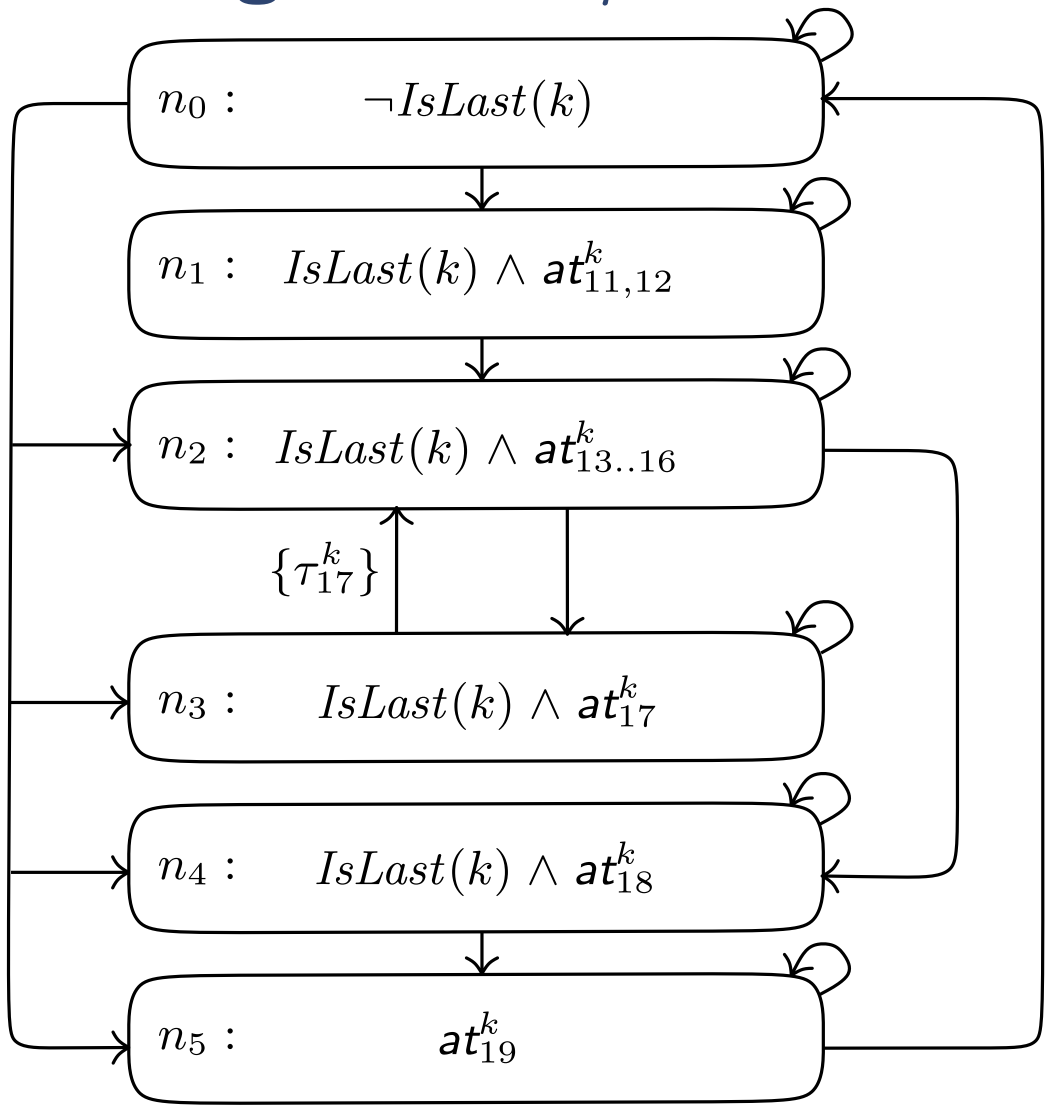
- The following predicate defines that k is last:

$$
\begin{aligned}
IsLast(k) \stackrel{\text{def}}{=} \quad & List(h, l.list, l.r) \wedge \\
& SomeMark\big(h, getp(h.l, list, null)\big) \wedge \\
& LastMarked\big(h, getp(h, l.list, null)\big) = a \wedge \\
& h[a].lockid = k
\end{aligned}
$$

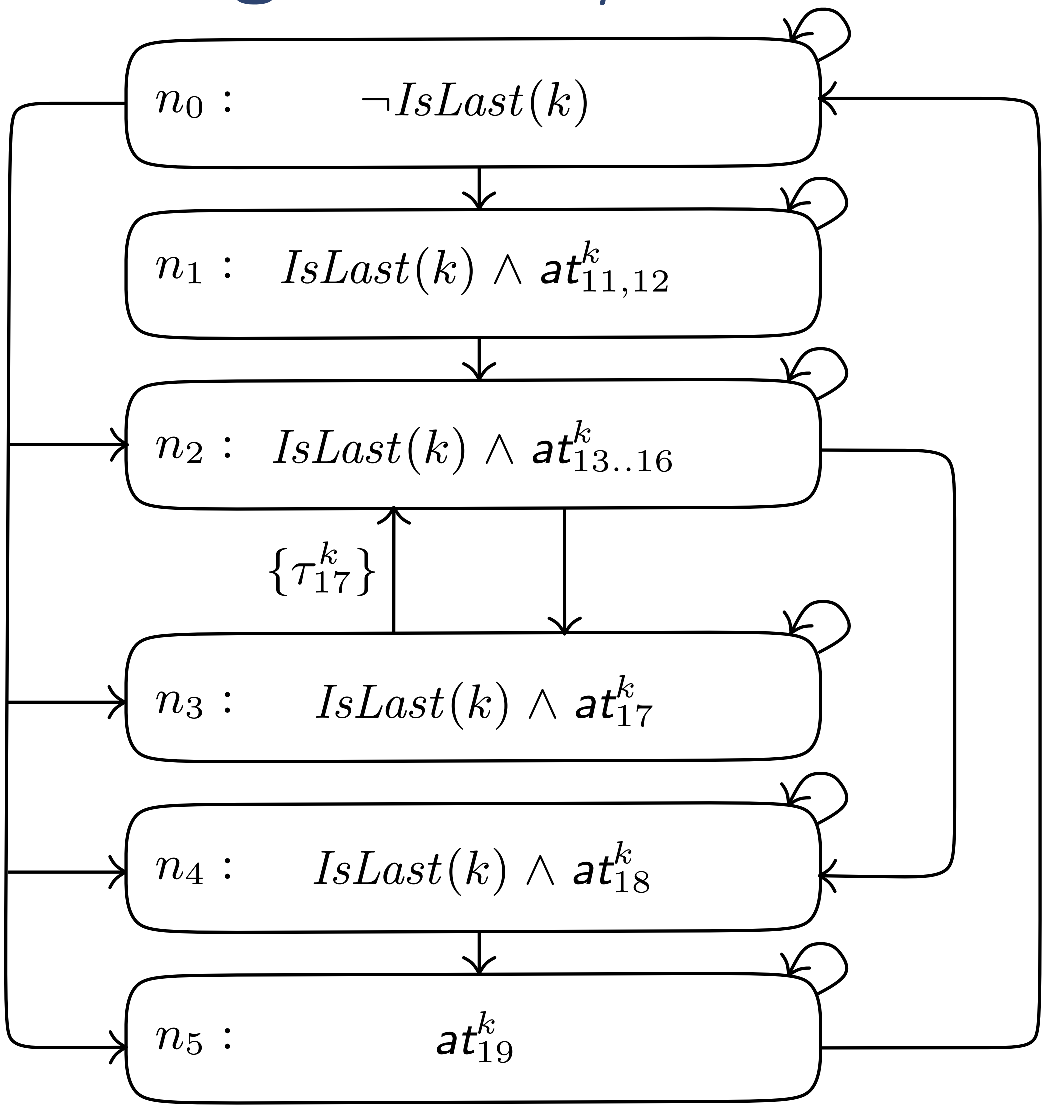
- The following function captures the *ahead* region:

$$\delta(s) = path2set\left(f_a(h, curr^{[k]})\right)$$

Verification Diagram for $\varphi^{(k)}$

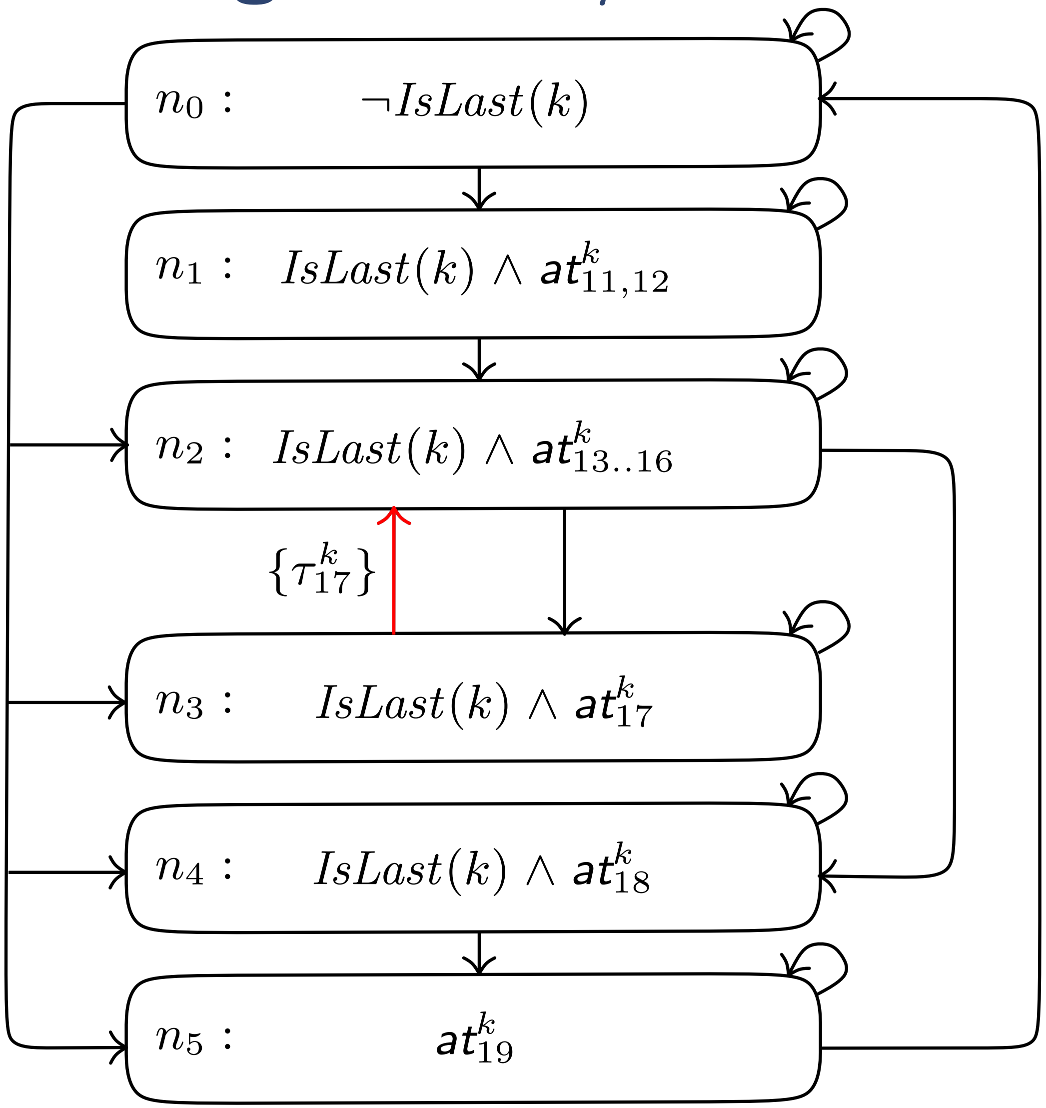


Verification Diagram for $\varphi^{(k)}$



This diagram clearly satisfies $\varphi^{(k)} : \Box(at^k_{11} \wedge IsLast(k) \rightarrow \Diamond at^k_{19})$

Verification Diagram for $\varphi^{(k)}$



$$\begin{pmatrix} IsLast(k) \wedge \\ at^k_{17} \end{pmatrix} \wedge \begin{pmatrix} at^k_{17} \wedge curr^k.lockid = NULL \wedge \\ at'^k_{13} \wedge curr'^k lockid = k \end{pmatrix} \rightarrow (IsLast(k) \wedge at^k_{13..16})$$

Conclusions

- Concurrent Datastructures are very hard to prove
- due to the interaction of
  - unstructured concurrency
  - unbounded concurrency
  - dynamic memory
- Liveness is even harder
- Verification Diagrams provide a separation between temporal reasoning and data reasoning...
- ... which requires sophisticated decision procedures
- Current work: parametrized verification diagrams, skip-lists, concurrent hash-maps, concurrent Schorr-Waite.
- Many possible collaborations: DPs as combinations, SMTs, implementation