A Theory of Skiplists with Applications to the Verification of Concurrent Datatypes

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

[1]IMDEA Software Institute, Spain

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

NFM'11, Pasadena, 18 April 2011

Motivation

Why do we want a decidable theory for skiplists?

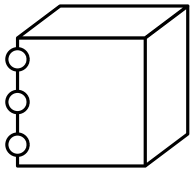
Verification of Concurrent Data-structures

Main Idea

Verification of Concurrent Data-structures

Main Idea

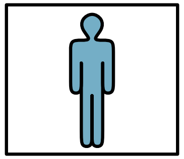
Concurrent DataStructure

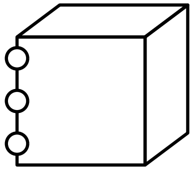


Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



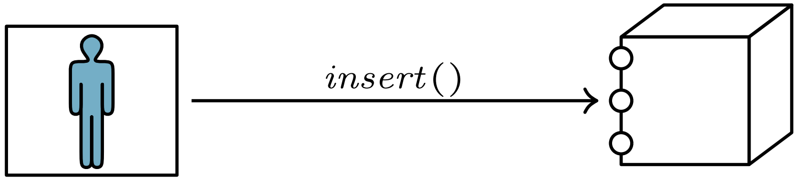


Most General Client

Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure

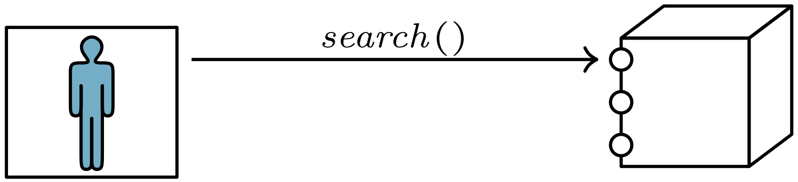


Most General Client

Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure

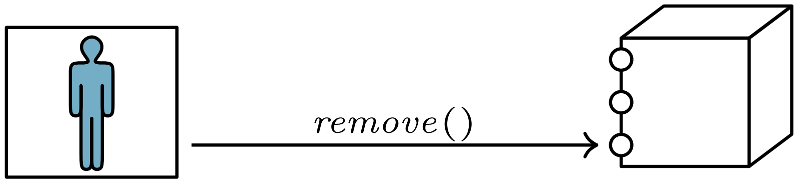


Most General Client

Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client

Verification of Concurrent Data-structures

Main Idea

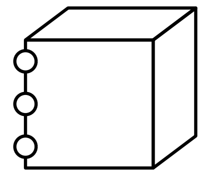
Concurrent DataStructure

Most General Client

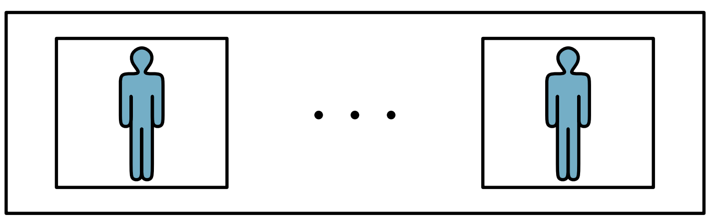
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



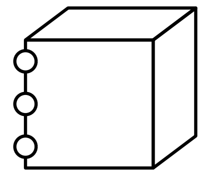
Most General Client



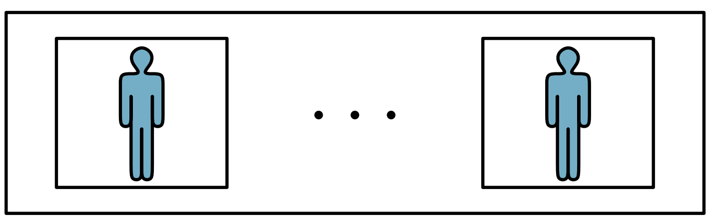
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client

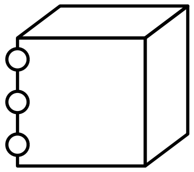


$$P[N] : P(1) || \cdots || P(N)$$

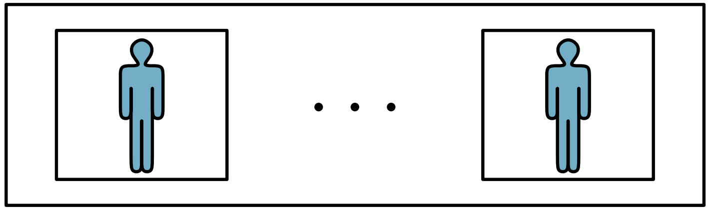
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client



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

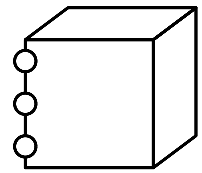
$+$

ghost variables

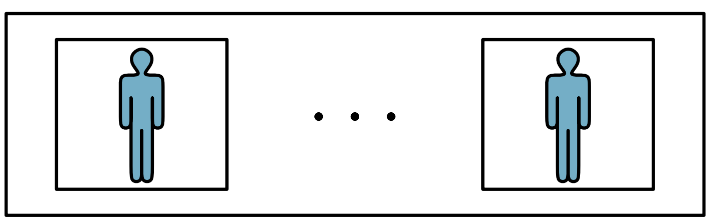
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client



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

$$+$$

ghost variables

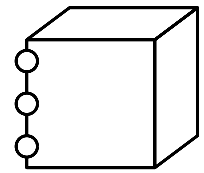
Property

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

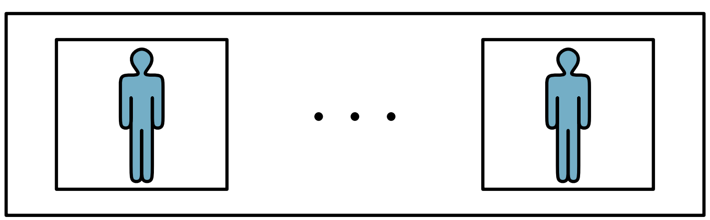
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client



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

$$+$$

ghost variables

Property

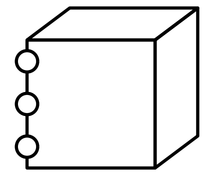
$$\varphi^{(k)}$$

LTL ($\Box$,$\Diamond$,$\mathcal{U}$,. . .)

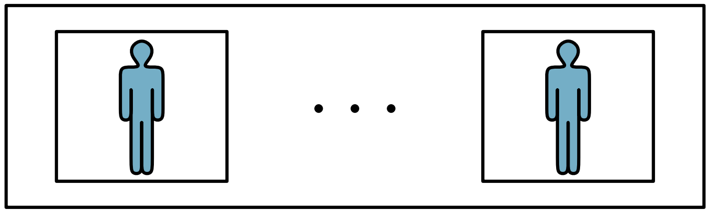
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client



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

$$+$$

ghost variables

Diagram

$$\mathcal{D}$$

Property

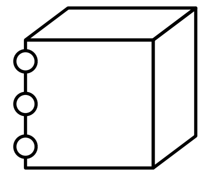
$$\varphi^{(k)}$$

LTL $(\Box, \Diamond, \mathcal{U}, \ldots)$

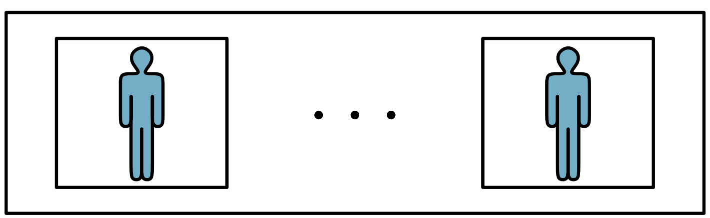
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client

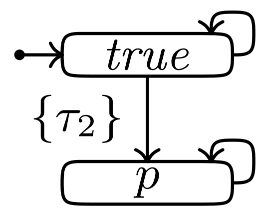


$$P[N] : P(1)||\cdots||P(N)$$

$$+$$

ghost variables

Diagram



Property

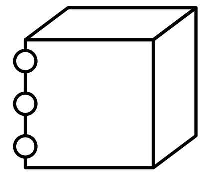
$$\varphi^{(k)}$$

LTL ($\Box$,$\Diamond$,$\mathcal{U}$,. . .)

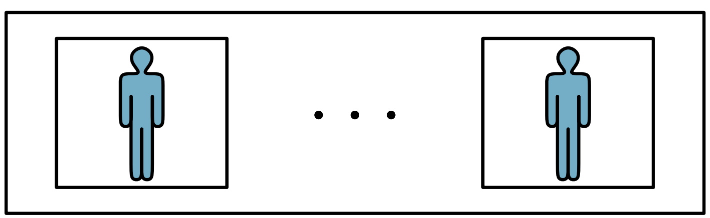
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client



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

$$+$$

ghost variables

Diagram

$$\mathcal{D}$$

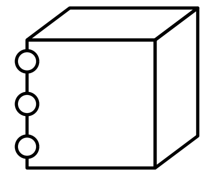
Property

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

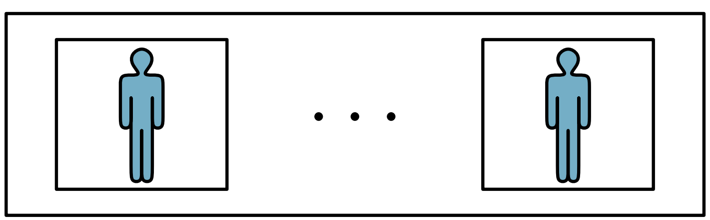
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client



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

$+$

ghost variables

Diagram

Property

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

Verification Conditions:

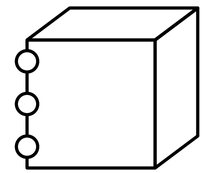
- Initiation
- Consecution
- Acceptance
- Fairness

Satisfaction (Model Checking)

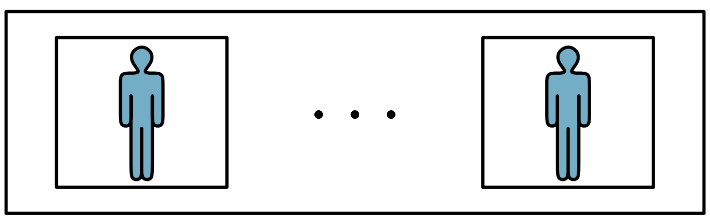
Verification of Concurrent Data-structures

Main Idea

Concurrent DataStructure



Most General Client



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

$+$

ghost variables

Diagram

Property

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

Verification Conditions:

- Initiation
- Consecution
- Acceptance
- Fairness

Satisfaction
(Model Checking)

Decision Procedures
(first order propositional logic)

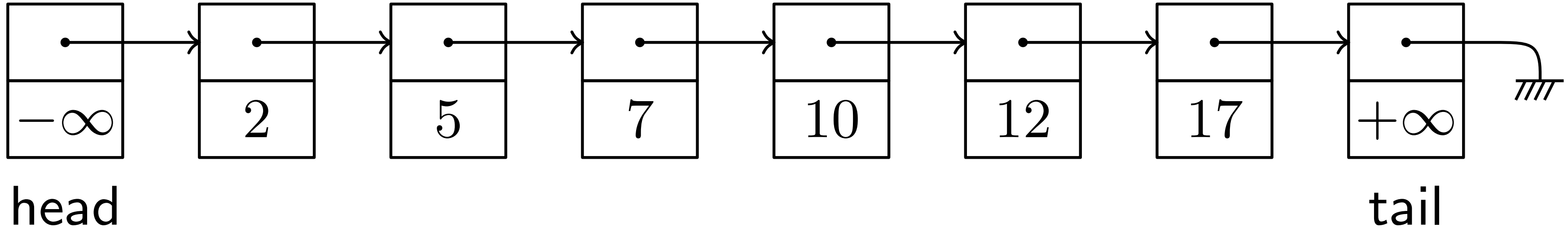
Concurrent Lock-Coupling Skiplists

Concurrent Lock-Coupling Skiplists

- Sorted list of elements

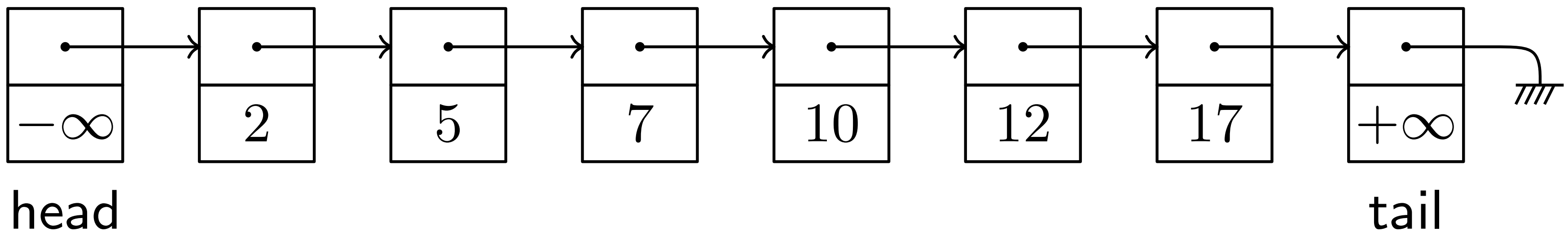
Concurrent Lock-Coupling Skiplists

- Sorted list of elements



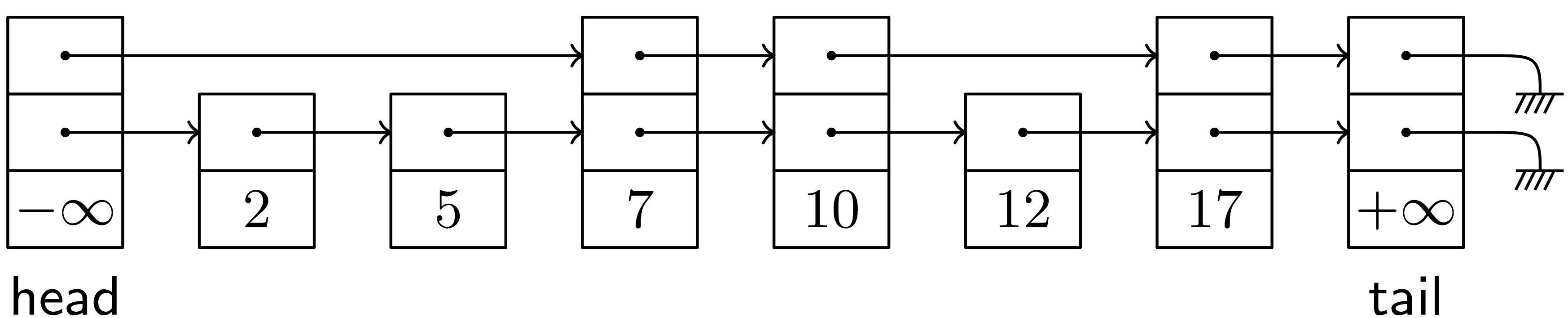
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists



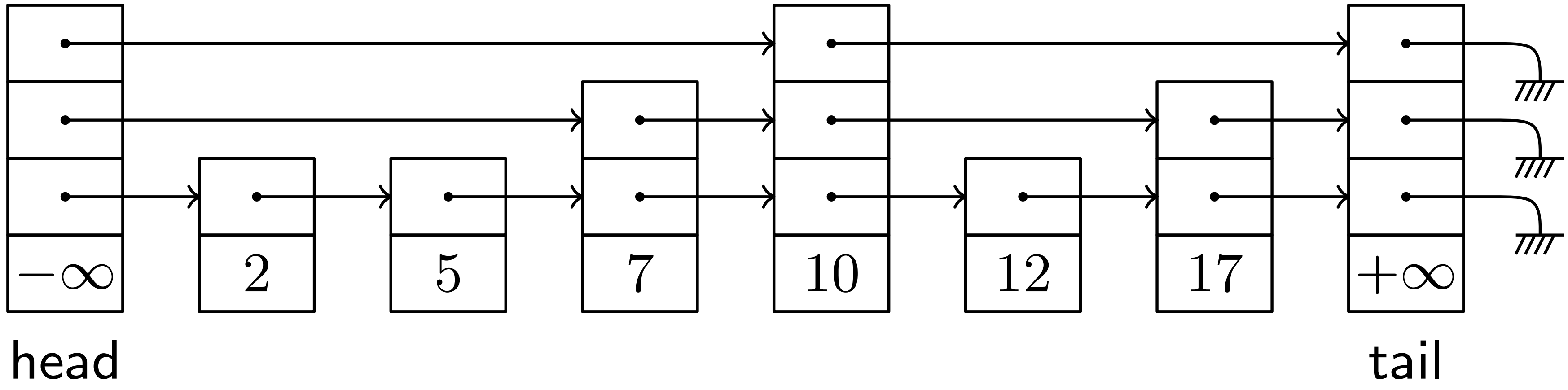
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists



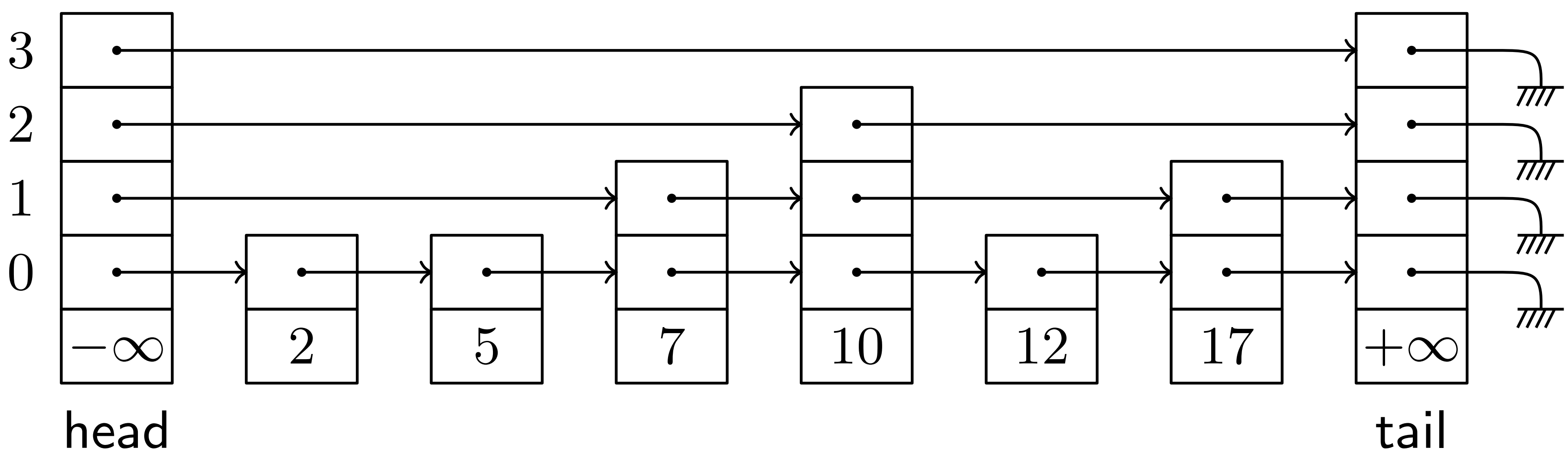
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists



Concurrent Lock-Coupling Skiplists

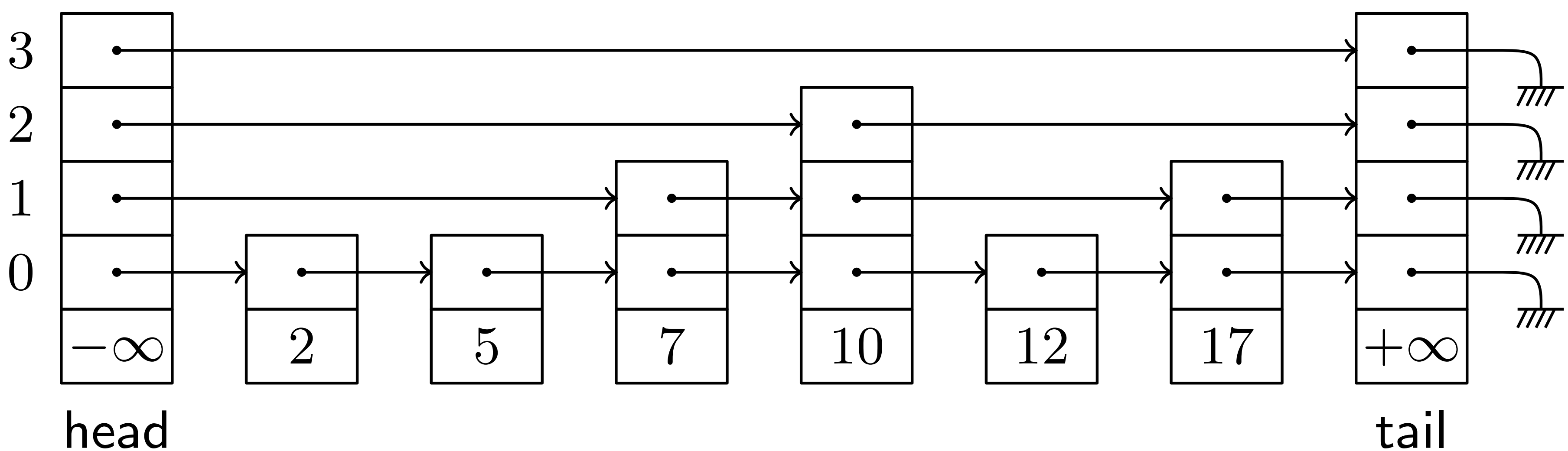
- Sorted list of elements
- Hierarchy of linked lists



Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists

```
class Skiplist {                class Node {
                                        Value v;
        Node* head;                     Key k;
        Node* tail;                     Array<Node*>(4) next;
                                        Array<Node*>(4) lock;
}                               }
```

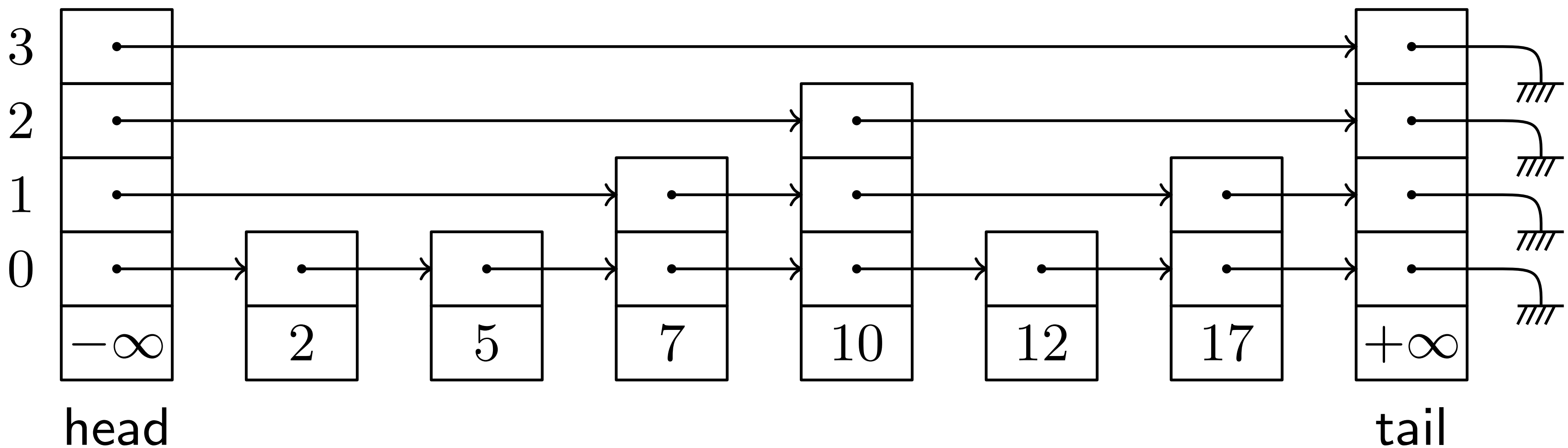


Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

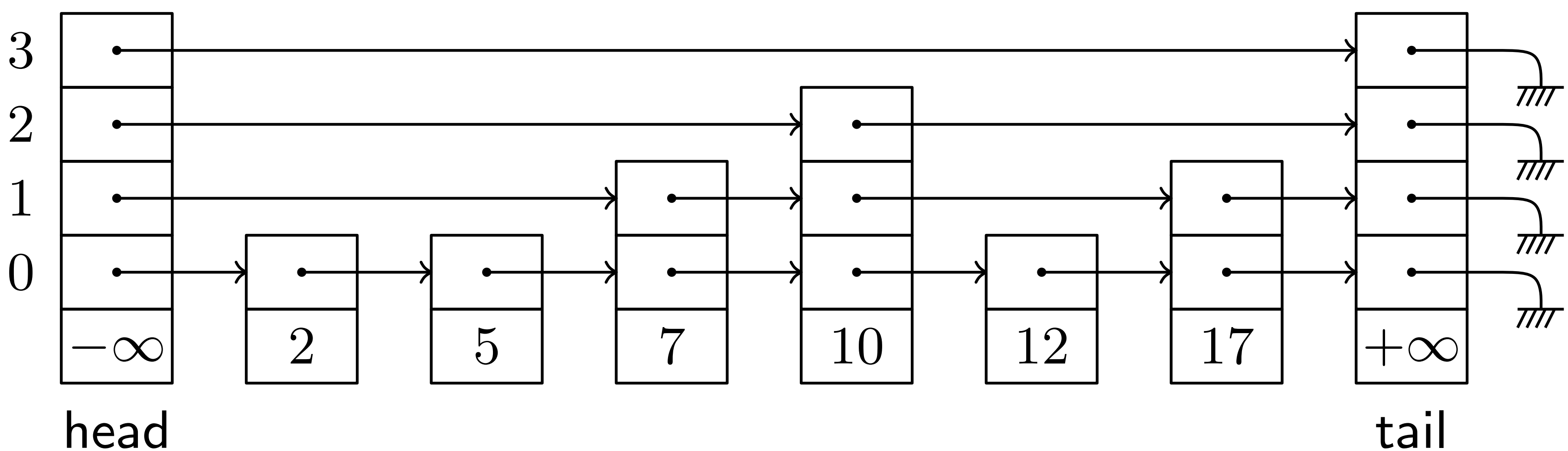


Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```



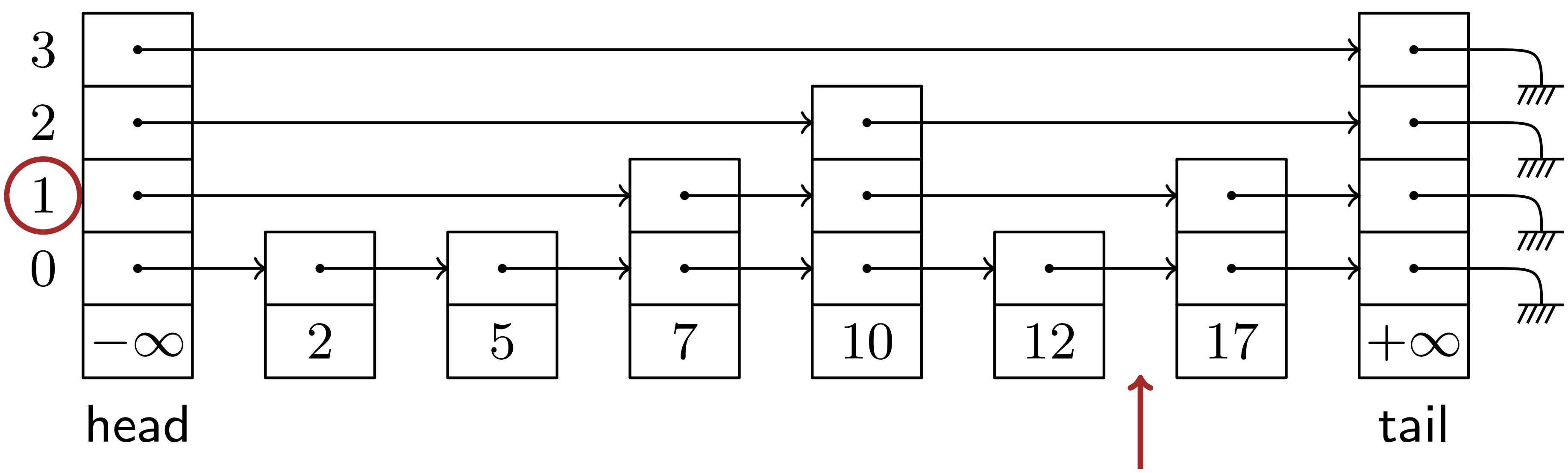
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



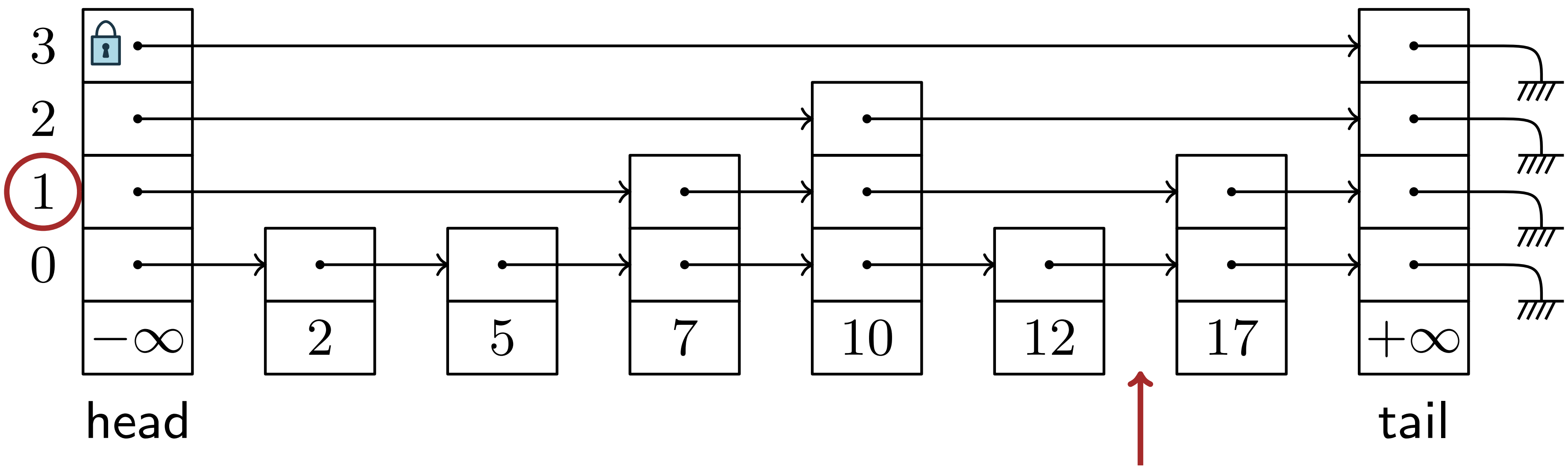
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



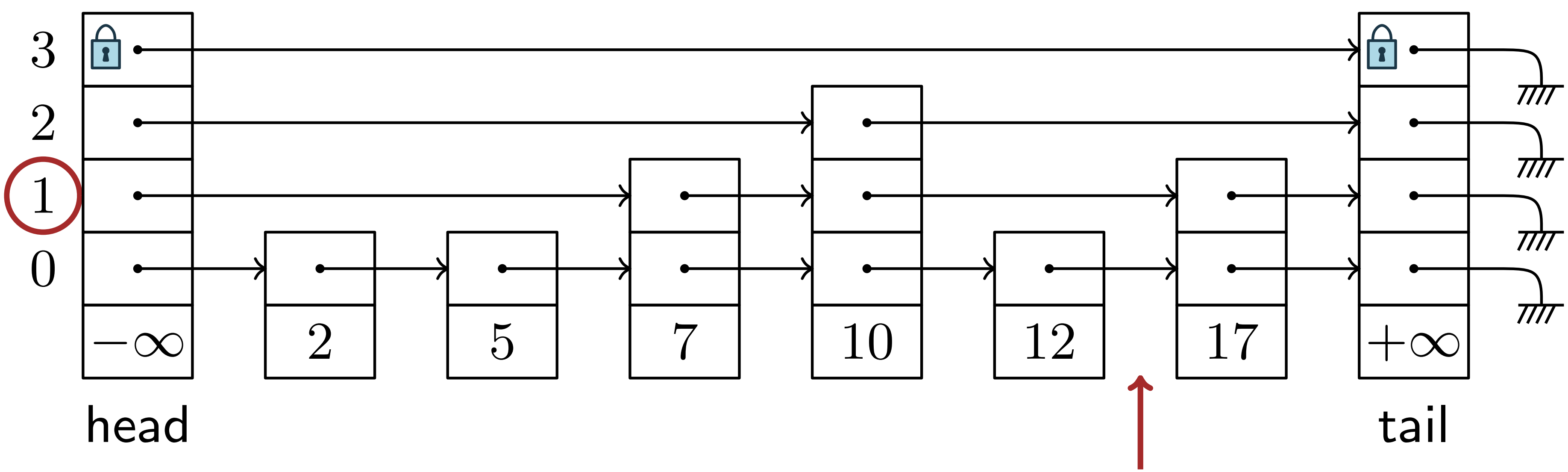
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



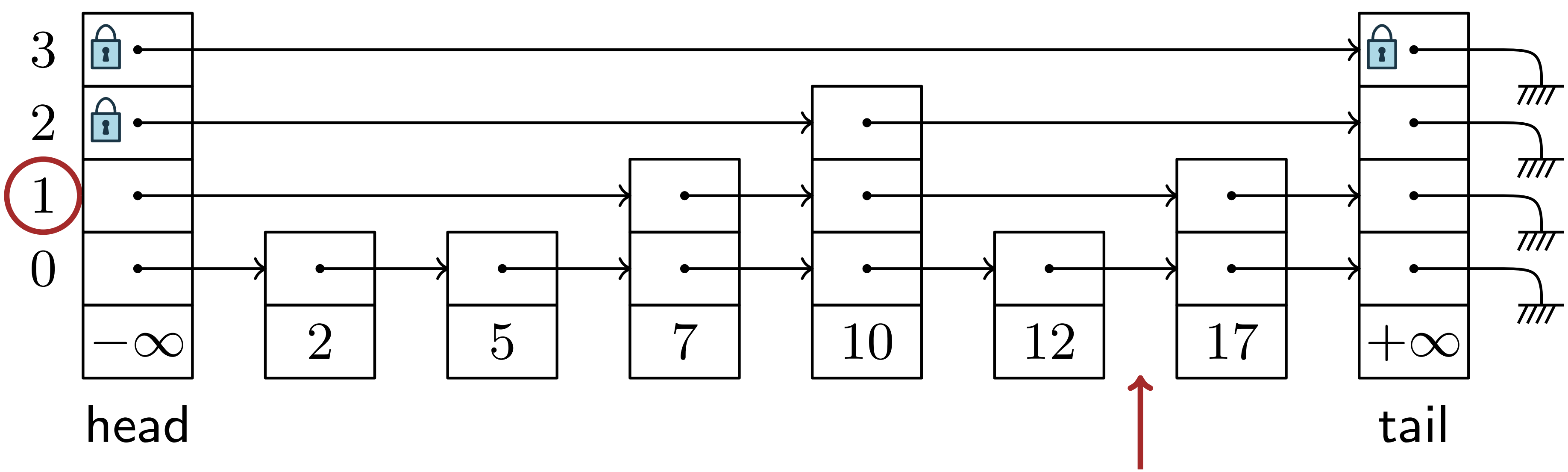
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {
        Node* head;
        Node* tail;
}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



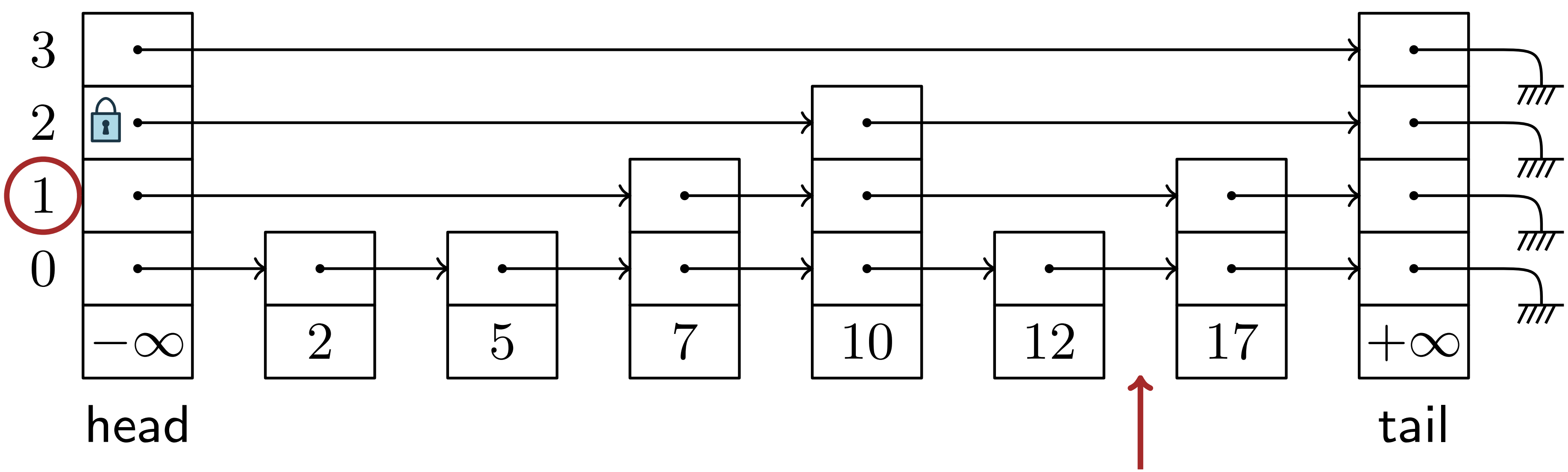
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1

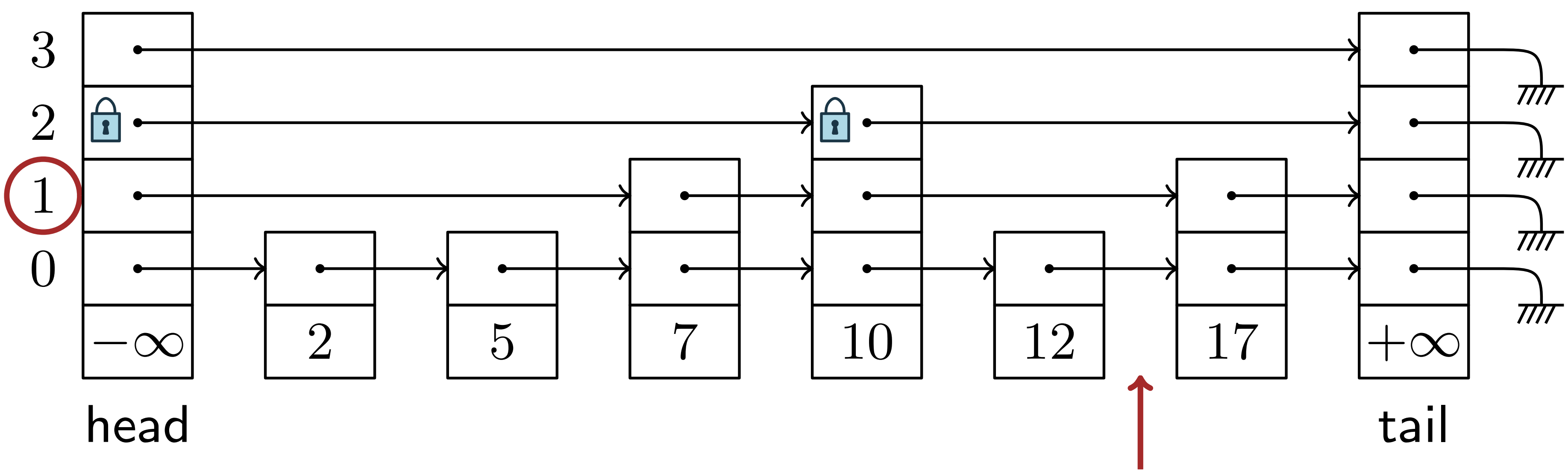


Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}

class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



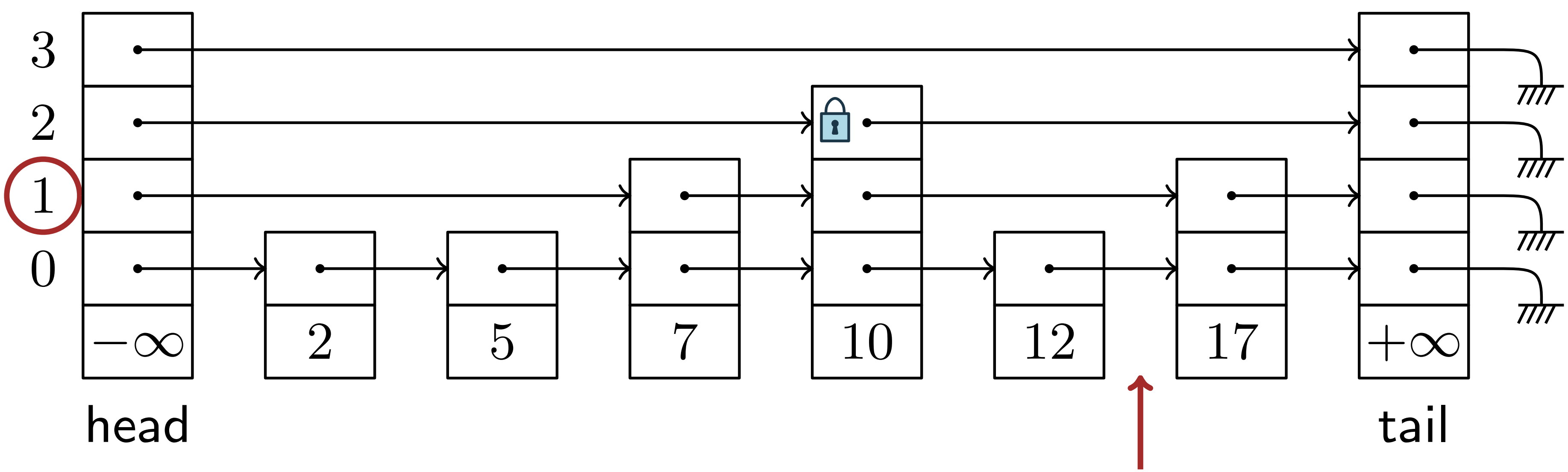
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



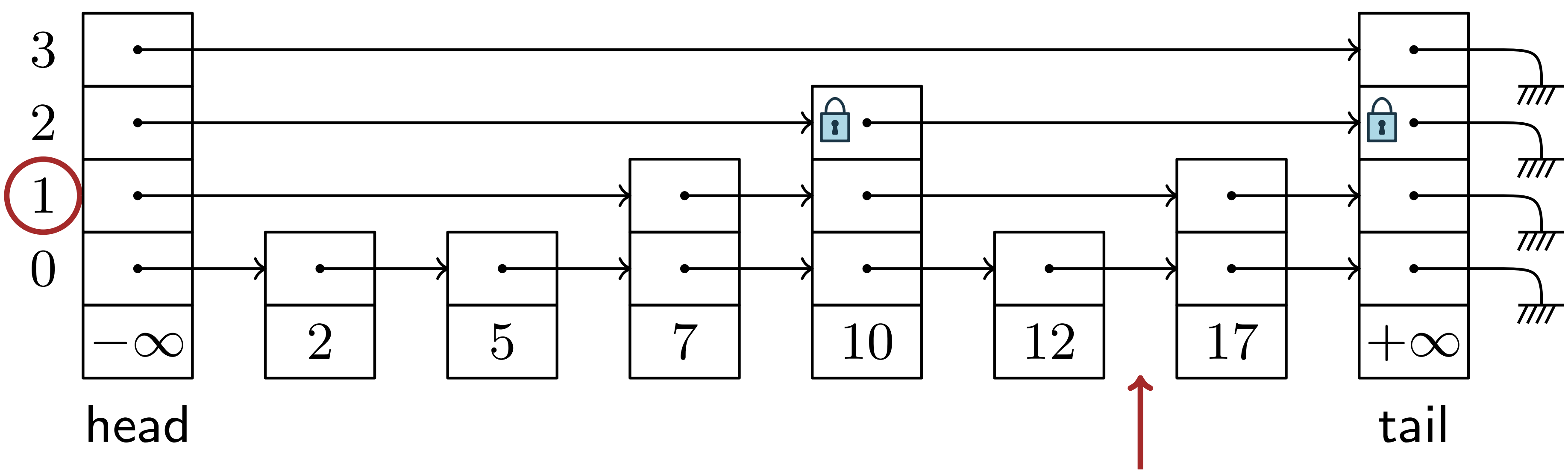
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



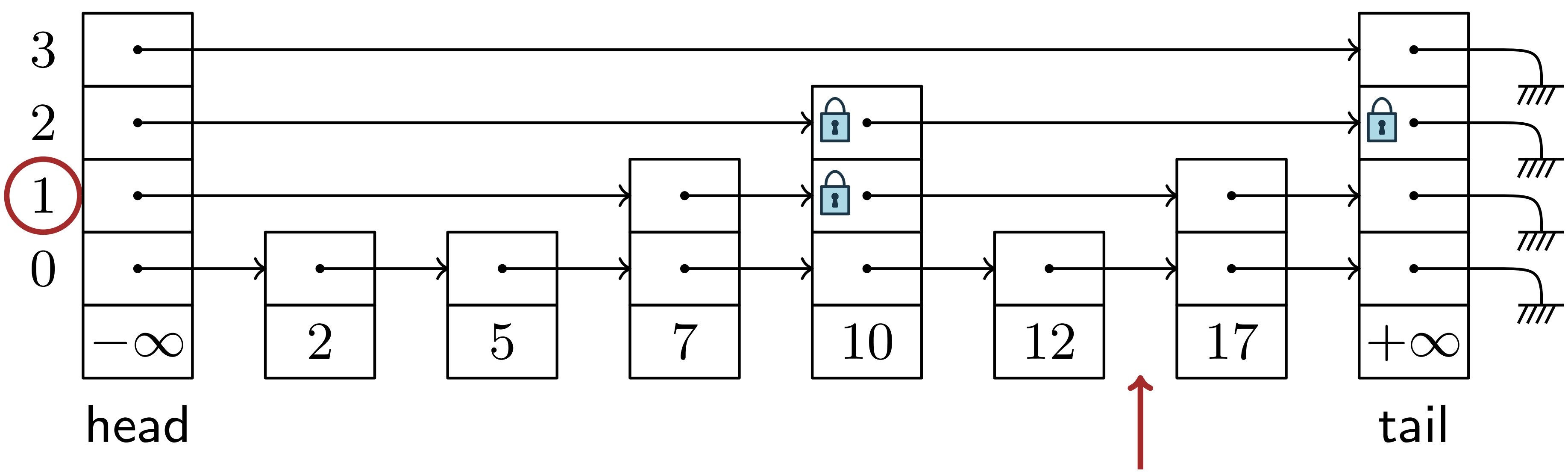
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



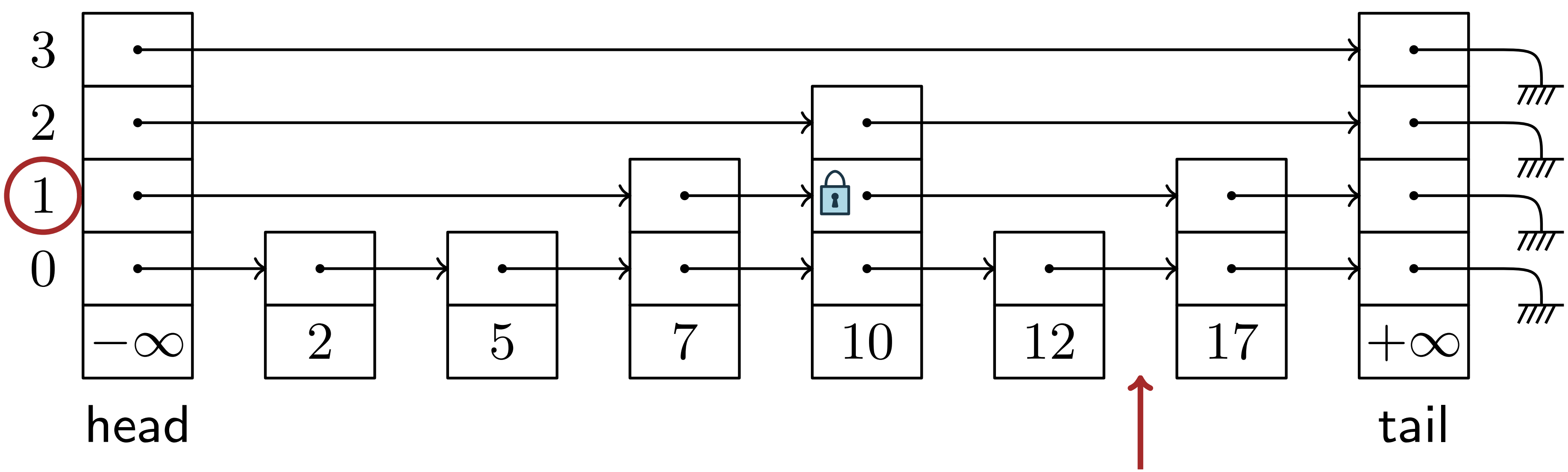
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



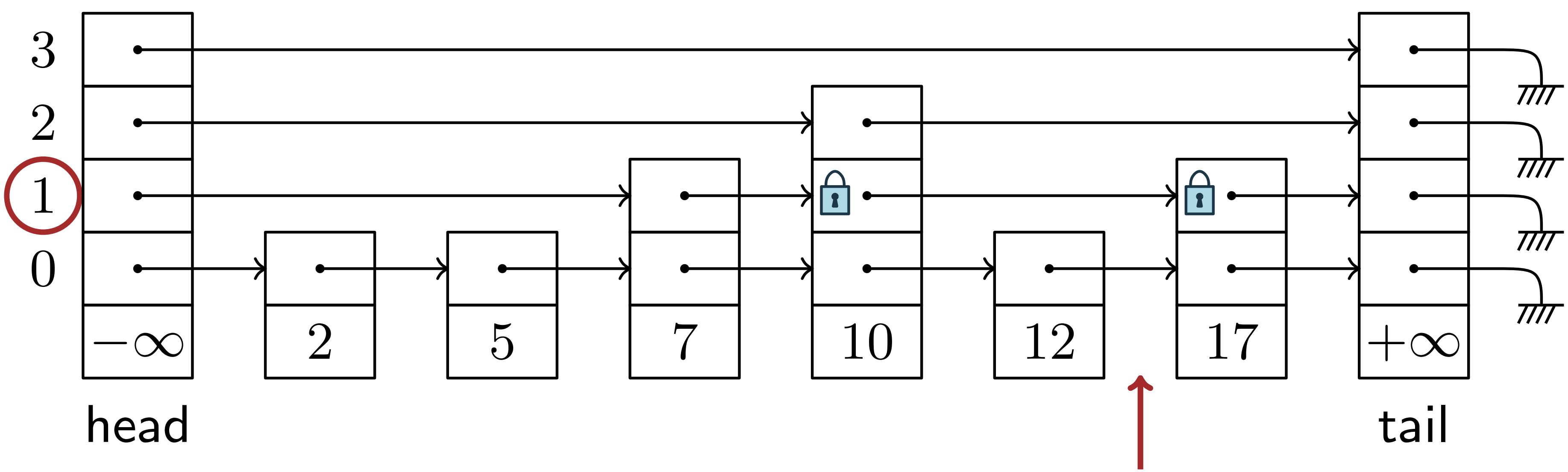
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



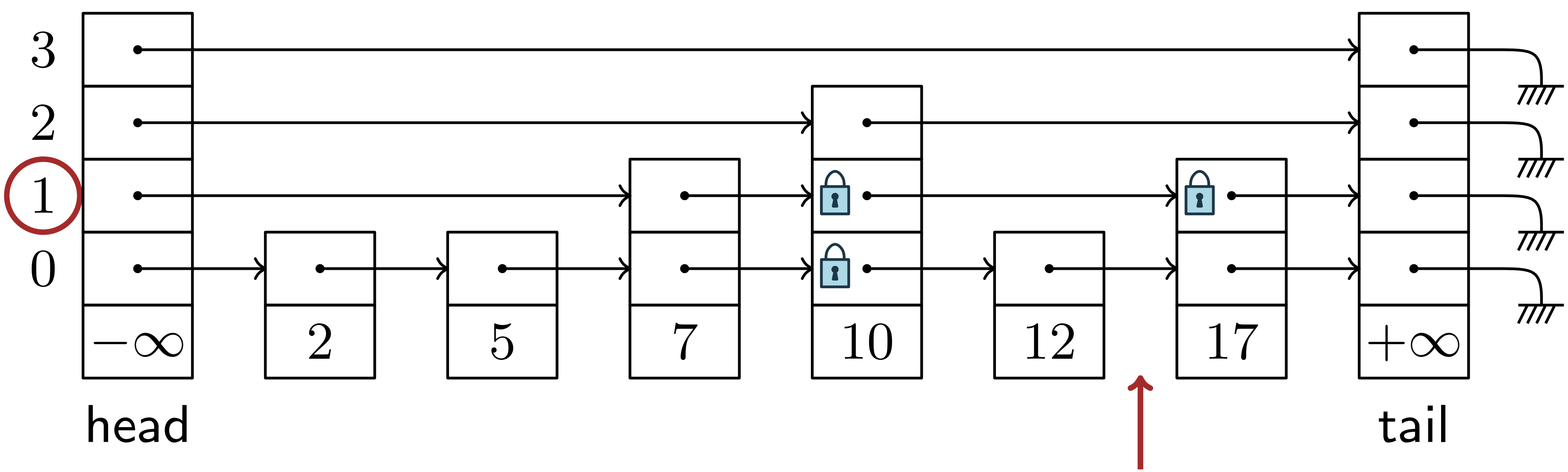
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



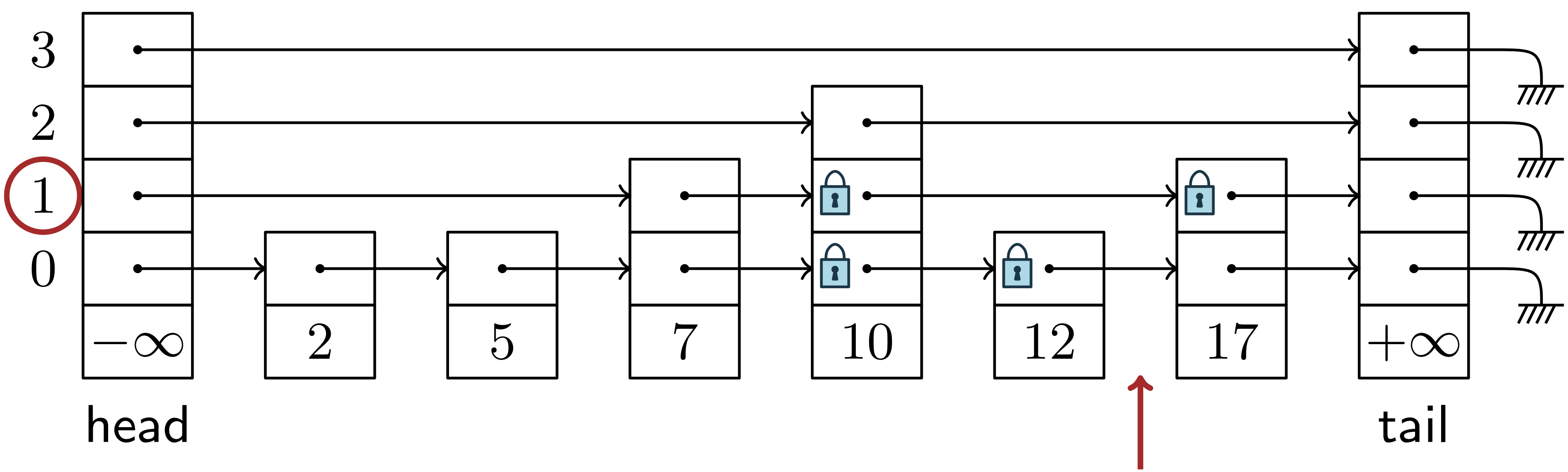
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



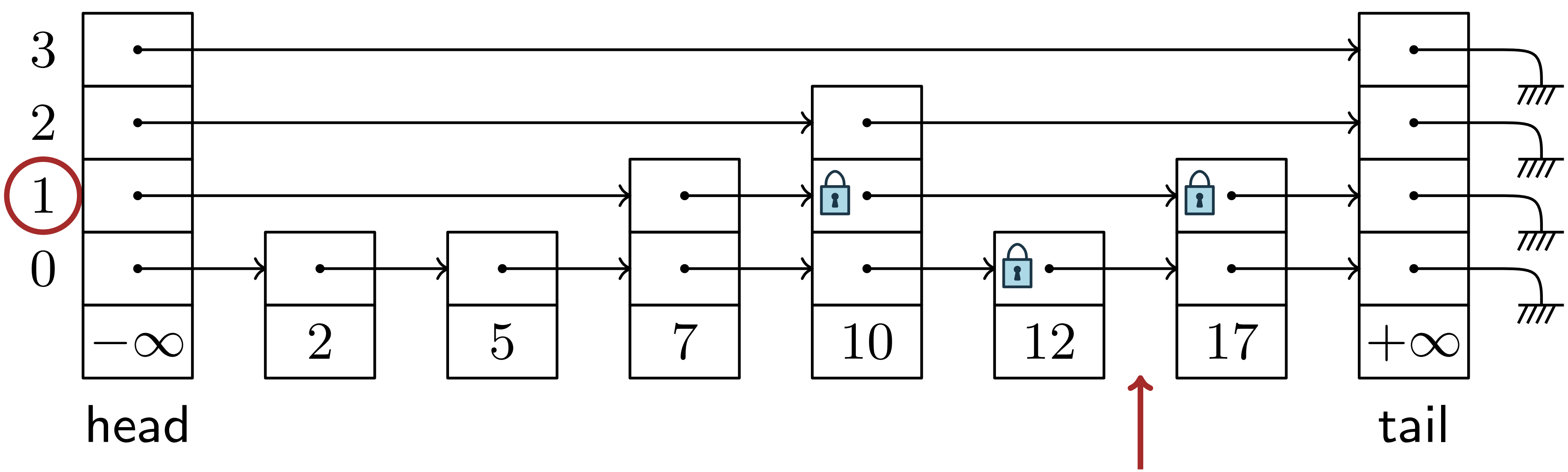
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



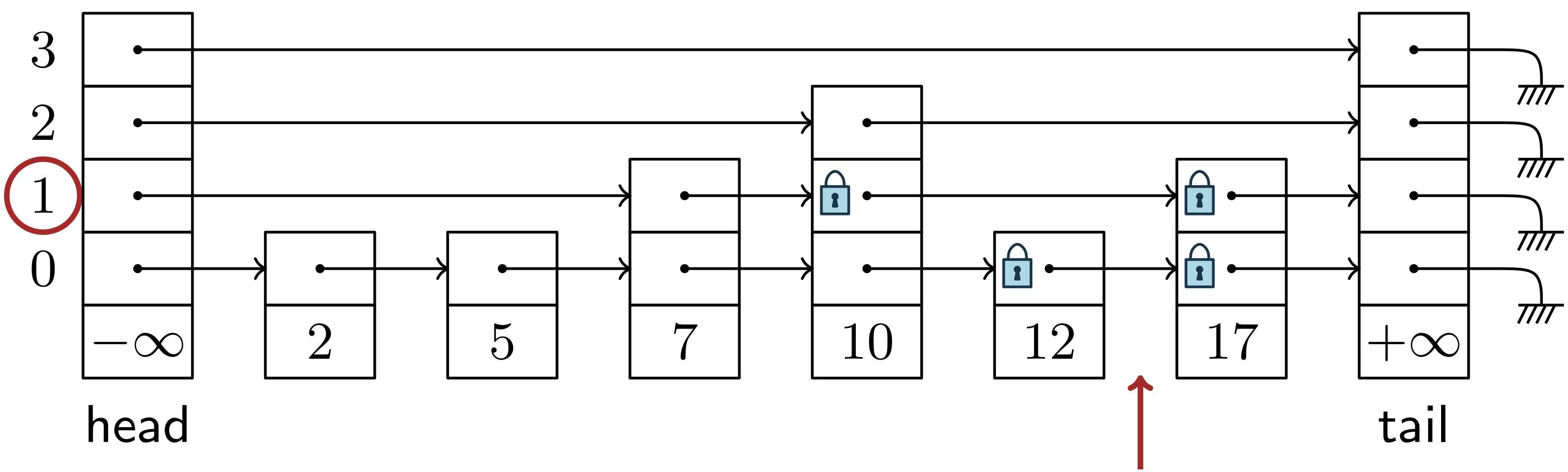
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



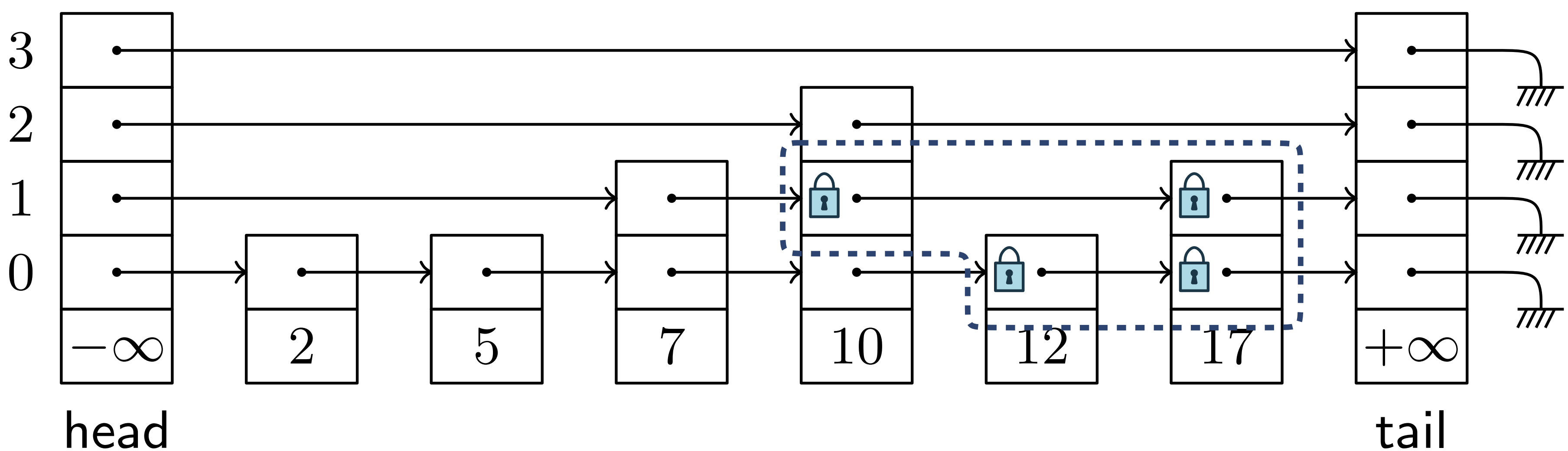
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



masked regions $= 2^{Node \times \mathbb{N}}$

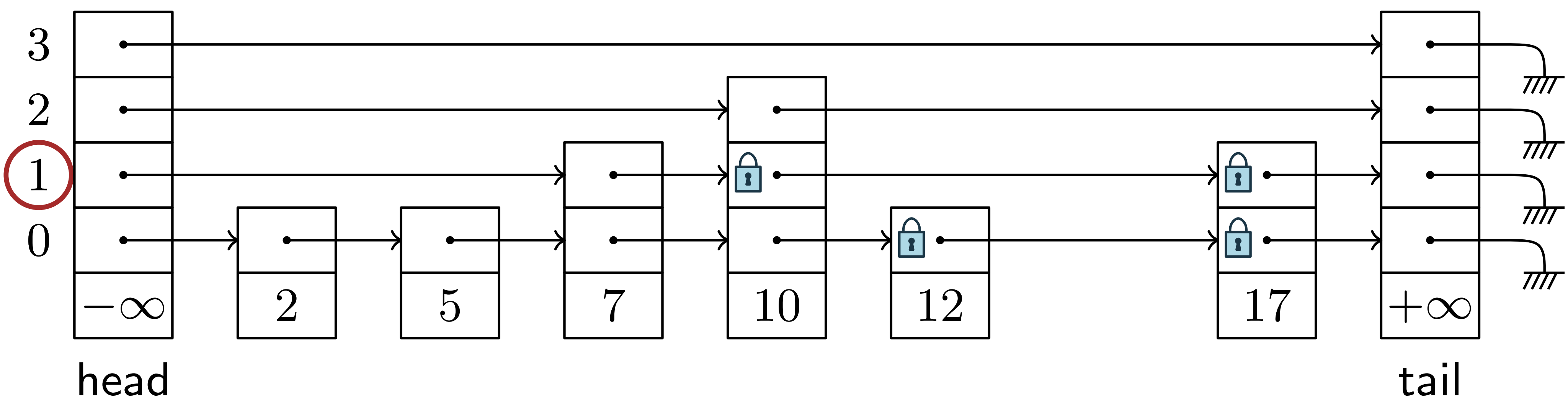
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



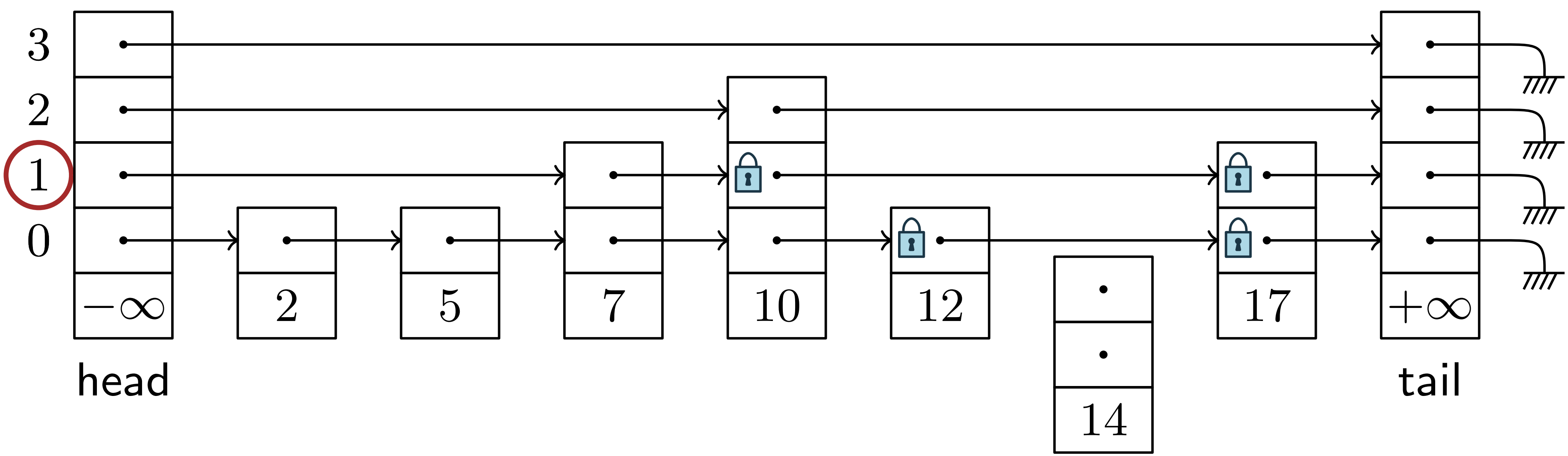
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



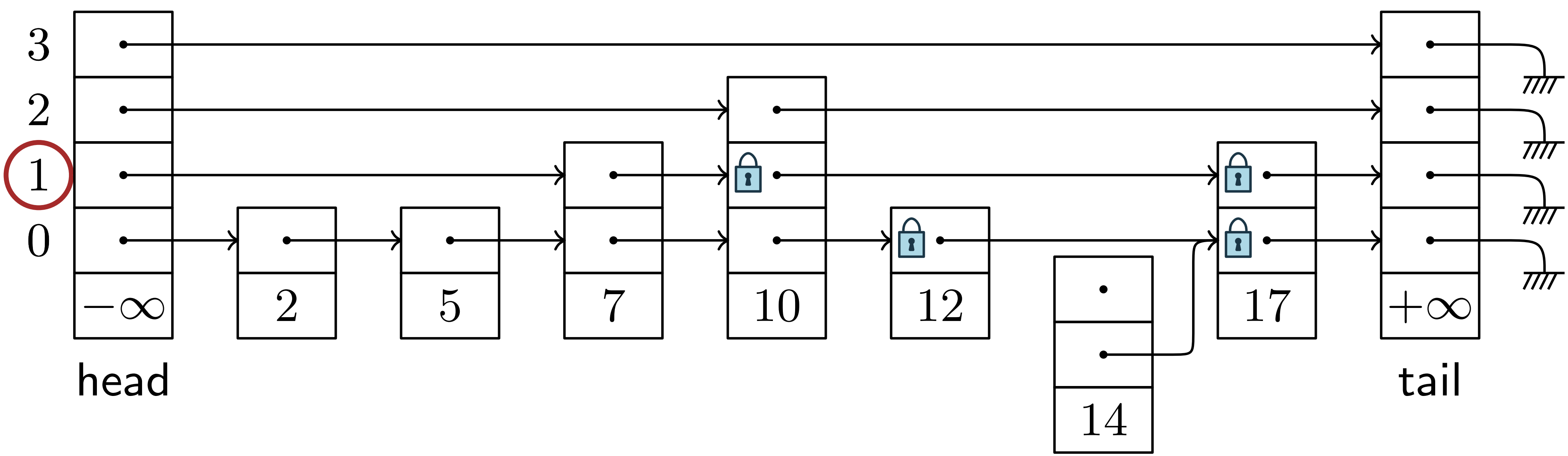
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



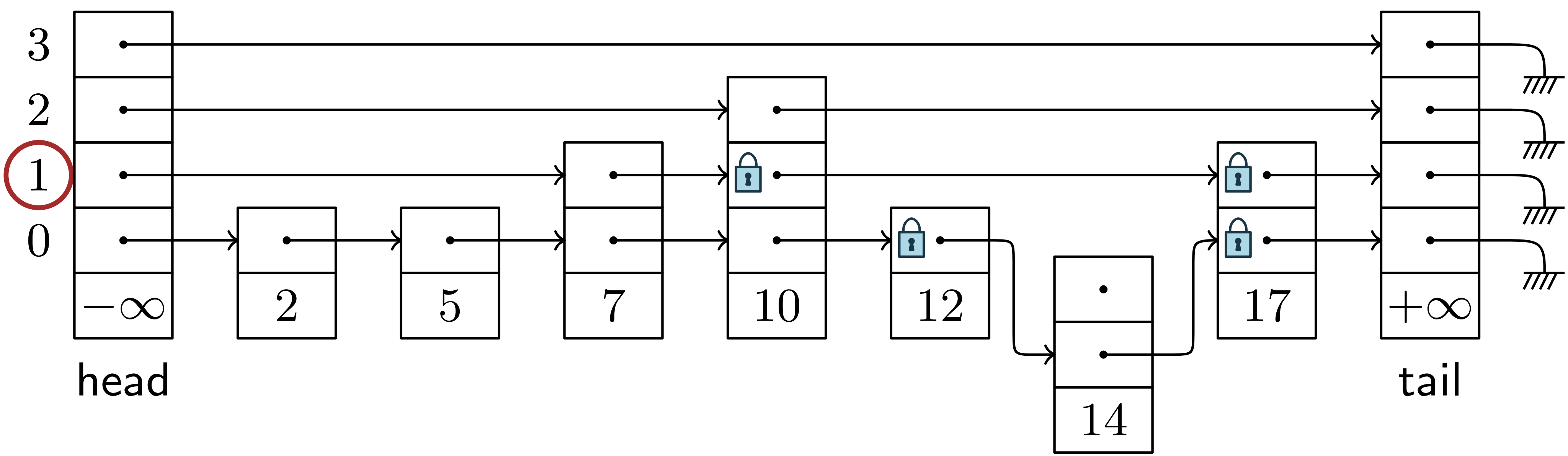
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



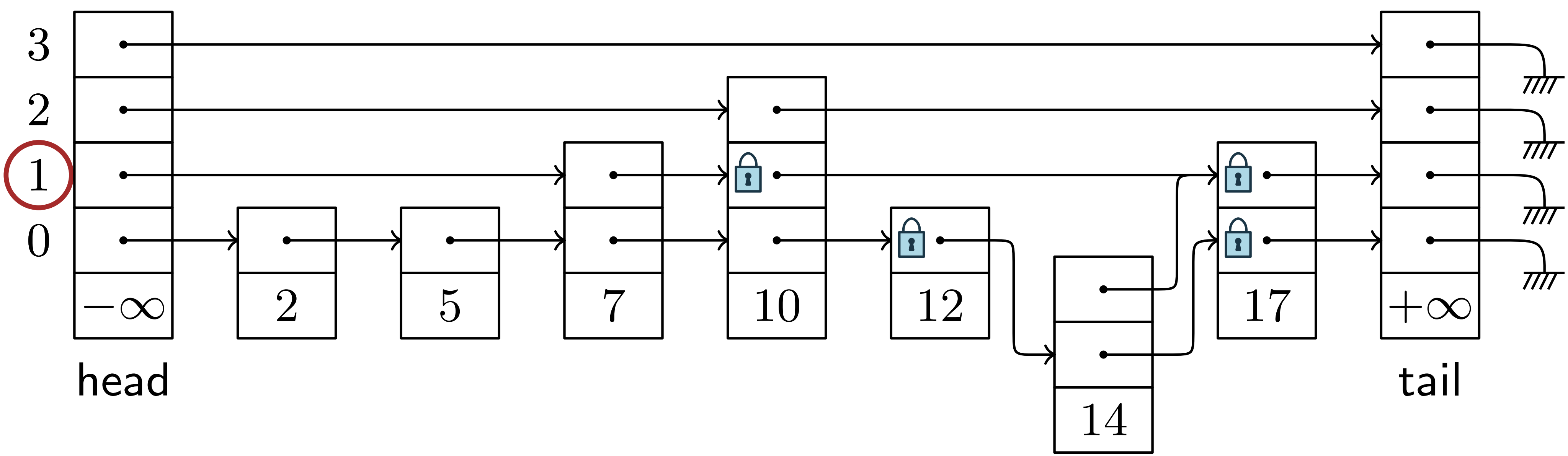
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



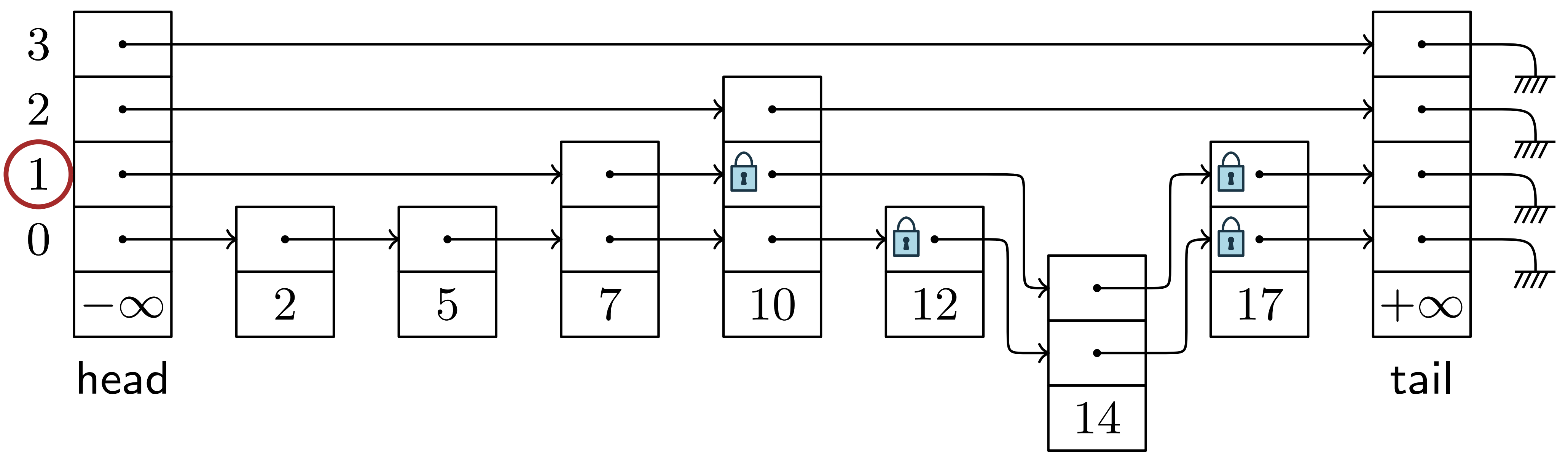
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



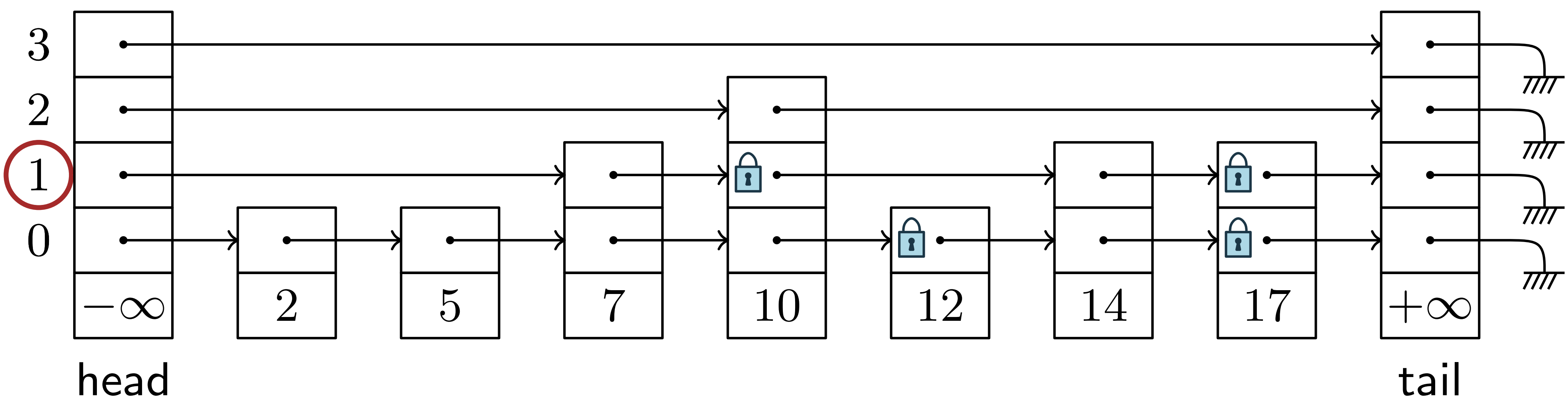
Concurrent Lock-Coupling Skiplists

- Sorted list of elements
- Hierarchy of linked lists
- Efficiency comparable to balanced binary search trees
- Reduce granularity of locks (in climbing fashion)

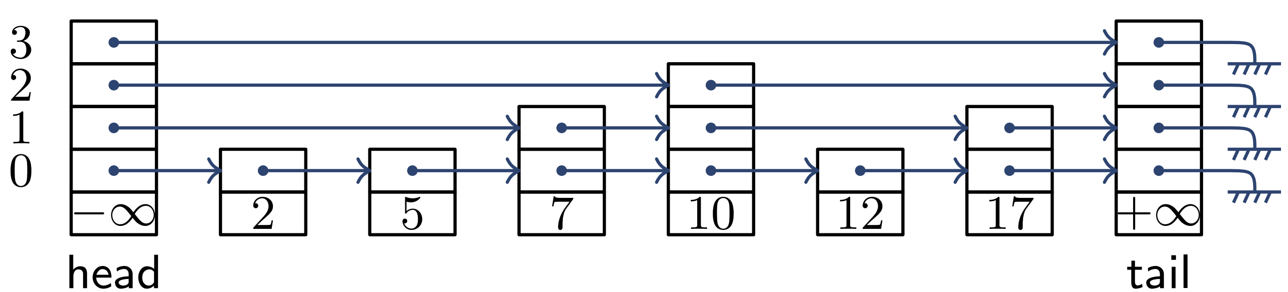
```
class Skiplist {

        Node* head;
        Node* tail;

}
```

```
class Node {
        Value v;
        Key k;
        Array<Node*>(4) next;
        Array<Node*>(4) lock;
}
```

insert(14) with height 1



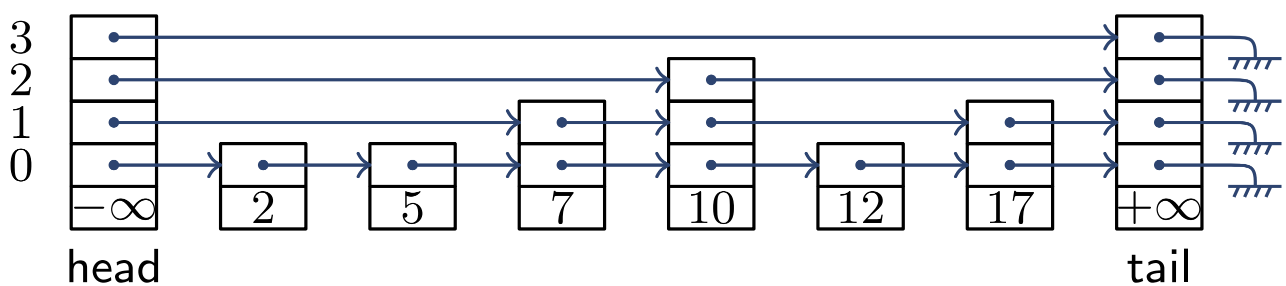
Verification of Concurrent Skiplists



Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList_4(h, sl)$

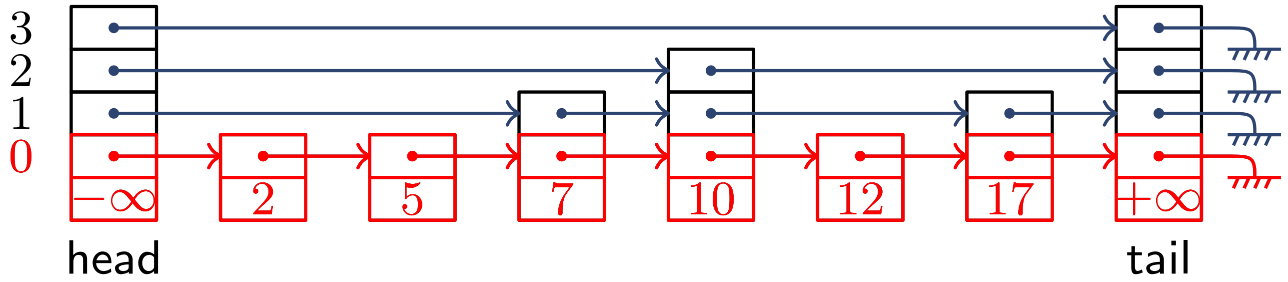
$$SkipList_4(h, sl : SkipList)$$



Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\square SkipList_4(h, sl)$

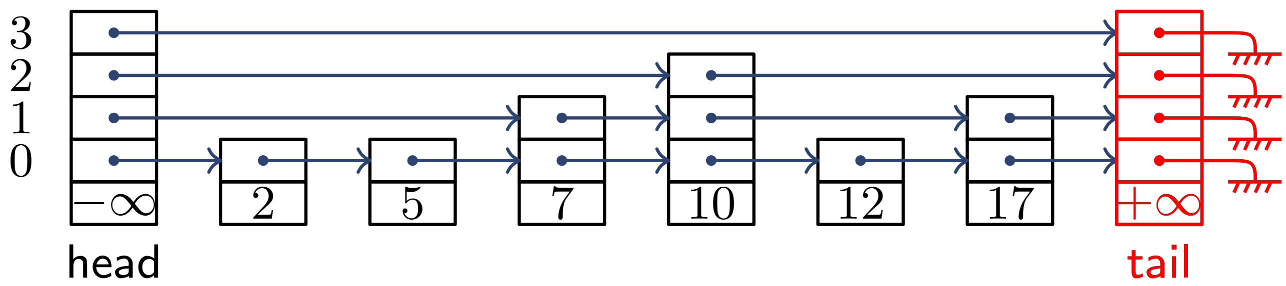
$$SkipList_4(h, sl : SkipList) \quad \hat{=} \quad {\color{red} OList(h, sl, 0)}$$



Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\square SkipList_4(h, sl)$

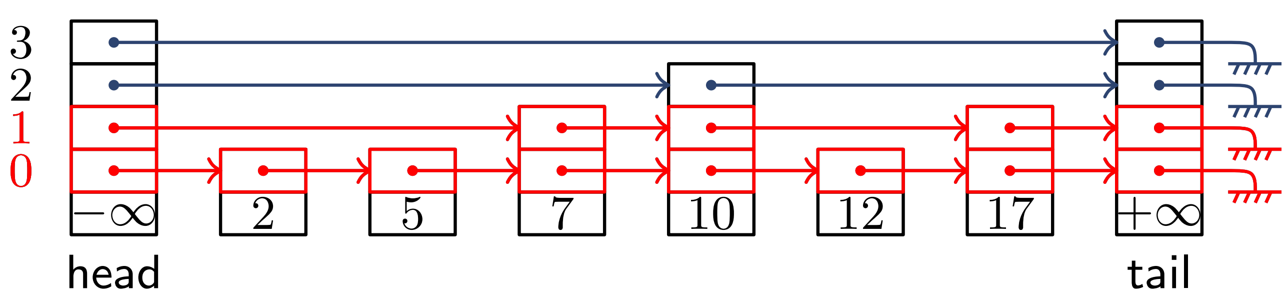
$$\begin{array}{rcl} SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) \hspace{12em} \wedge \\ & & \begin{pmatrix} h[sl].tail.next[0] = null \ \wedge \ h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \ \wedge \ h[sl].tail.next[3] = null \end{pmatrix} \end{array}$$



Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\square SkipList_4(h, sl)$

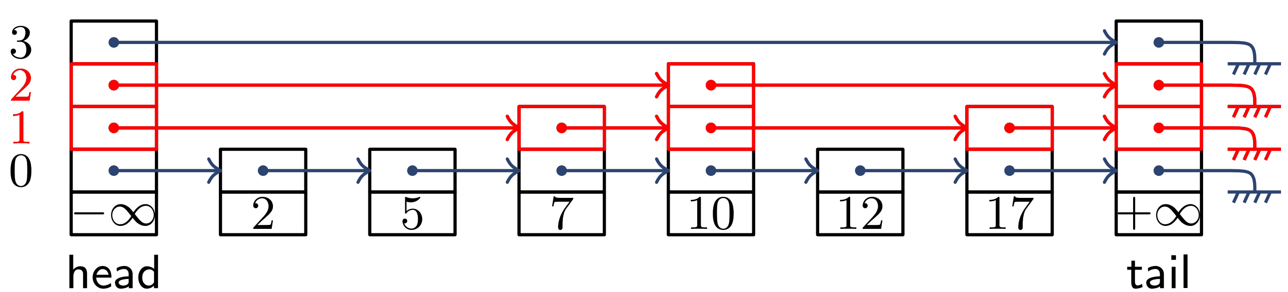
$$
\begin{array}{rcl}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) \hfill \wedge \\
& & \begin{pmatrix} h[sl].tail.next[0] = null \ \wedge \ h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \ \wedge \ h[sl].tail.next[3] = null \end{pmatrix} \quad \wedge \\
& & \begin{pmatrix} SubList(\boxed{h}, \boxed{sl.head, sl.tail, 1}, \boxed{sl.head, sl.tail, 0}) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix}
\end{array}
$$



Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList_4(h, sl)$

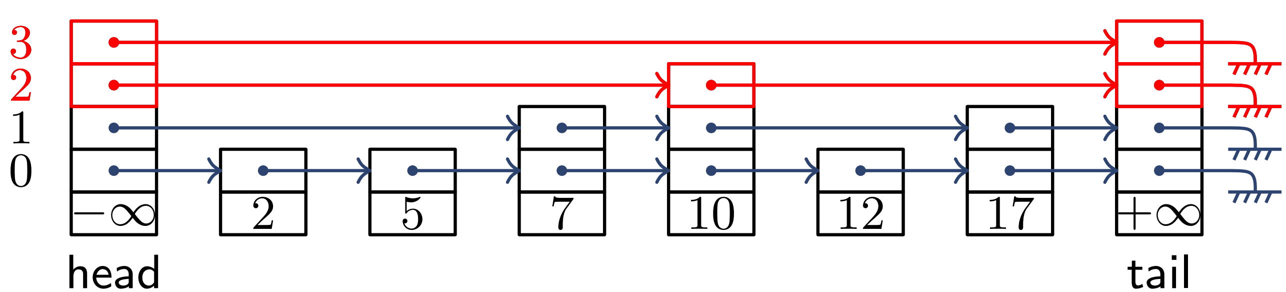
$$
\begin{array}{rcll}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) & \wedge \\
 & & \begin{pmatrix} h[sl].tail.next[0] = null \;\wedge\; h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \;\wedge\; h[sl].tail.next[3] = null \end{pmatrix} & \wedge \\
 & & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(\boxed{h}, \boxed{sl.head, sl.tail, 2}, \boxed{sl.head, sl.tail, 1}) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix} &
\end{array}
$$



Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\square SkipList_4(h, sl)$

$$
\begin{array}{lcll}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) & \wedge \\
& & \begin{pmatrix} h[sl].tail.next[0] = null \ \wedge \ h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \ \wedge \ h[sl].tail.next[3] = null \end{pmatrix} & \wedge \\
& & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ \color{red}{SubList(\boxed{h}, \boxed{sl.head, sl.tail, 3}, \boxed{sl.head, sl.tail, 2})} \end{pmatrix} &
\end{array}
$$

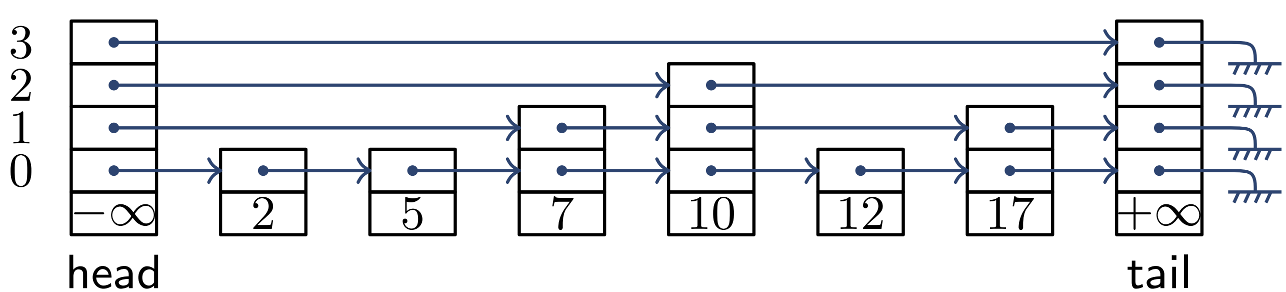


Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList_4(h, sl)$

$$\begin{array}{rcll} SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) & \wedge \\ & & \begin{pmatrix} h[sl].tail.next[0] = null \ \wedge \ h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \ \wedge \ h[sl].tail.next[3] = null \end{pmatrix} & \wedge \\ & & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \ \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \ \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix} & \end{array}$$

- Program **transitions**



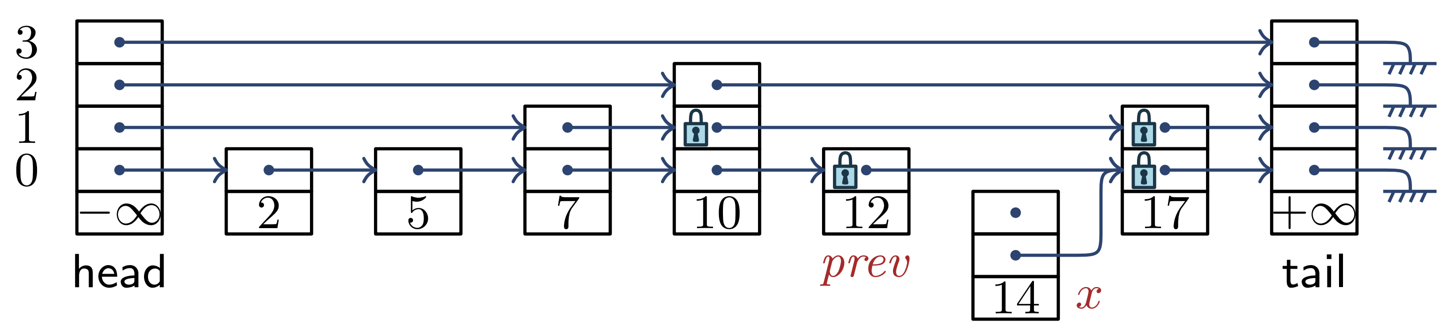
Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\square SkipList_4(h, sl)$

$$
\begin{array}{rcl}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) \qquad\qquad \wedge \\
 & & \begin{pmatrix} h[sl].tail.next[0] = null \ \wedge\ h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \ \wedge\ h[sl].tail.next[3] = null \end{pmatrix} \quad \wedge \\
 & & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix}
\end{array}
$$

- Program **transitions**

```
35: . . .
36: prev.next[0] := x
37: . . .
```



Verification of Concurrent Skiplists

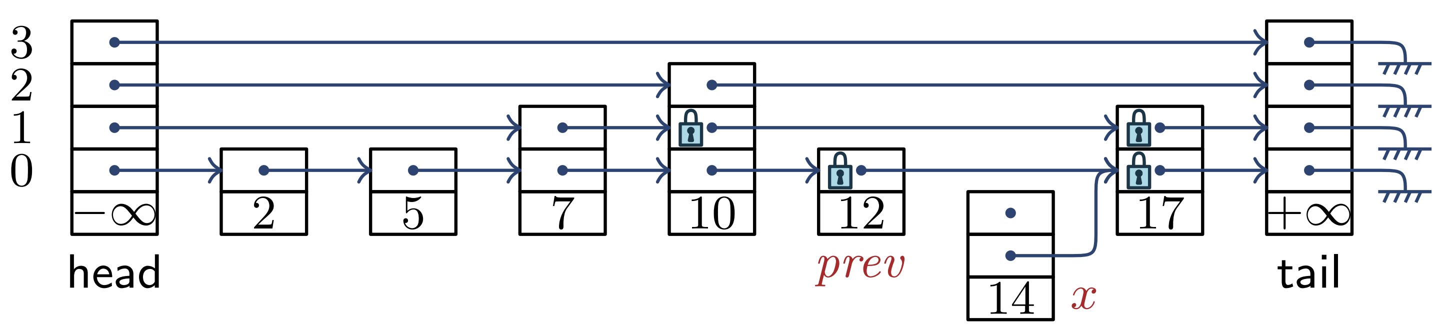
- **Skiplist** shape **preservation** : $\square SkipList_4(h, sl)$

$$
\begin{array}{rcl}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) \qquad \wedge \\
& & \begin{pmatrix} h[sl].tail.next[0] = null \ \wedge \ h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \ \wedge \ h[sl].tail.next[3] = null \end{pmatrix} \qquad \wedge \\
& & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix}
\end{array}
$$

- Program **transitions** : $SL_4(h, sl)$

$SkipList_4(h, sl)$

```
35: . . .
36: prev.next[0] := x
37: . . .
```



Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList_4(h, sl)$

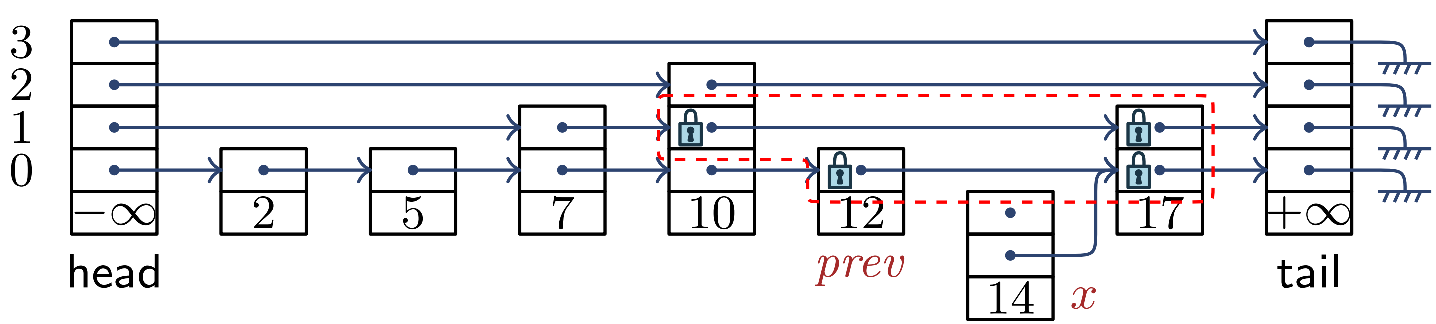
$$
\begin{array}{rcl}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) \qquad\qquad\qquad\qquad \wedge \\
& & \begin{pmatrix} h[sl].tail.next[0] = null \ \wedge \ h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \ \wedge \ h[sl].tail.next[3] = null \end{pmatrix} \quad \wedge \\
& & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix}
\end{array}
$$

- Program **transitions** : $SL_4(h, sl) \wedge \varphi_{aux}$

$$
SkipList_4(h, sl)
$$

$$
\begin{pmatrix}
x.key & = & 14 & \wedge \\
prev.key & < & 14 & \wedge \\
x.next[0].key & > & 14 & \wedge \\
prev.next[0] & = & x.next[0] & \wedge \\
\multicolumn{3}{c}{(x, 0) \notin sl.r \wedge 0 \leq 0 \leq 3} &
\end{pmatrix}
$$

```
35: . . .
36: prev.next[0] := x
37: . . .
```



Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\square SkipList_4(h, sl)$

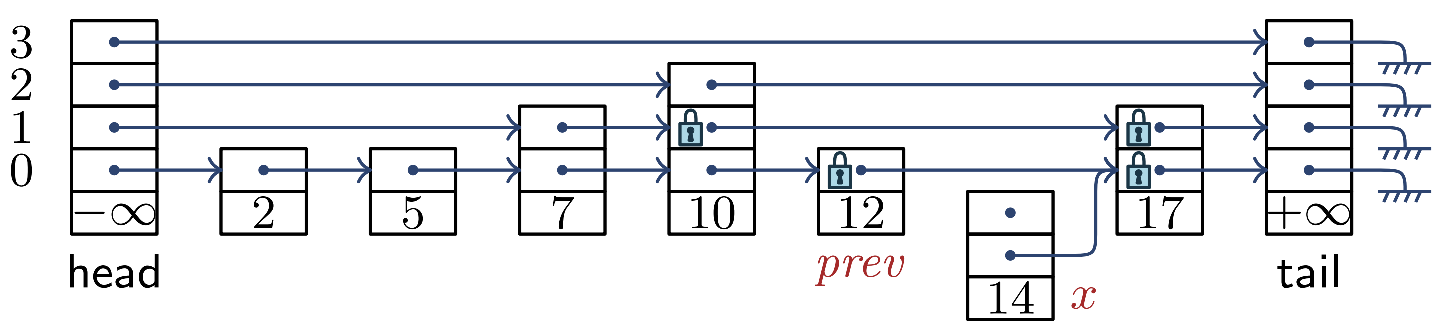
$$
\begin{array}{rcll}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) & \wedge \\
& & \begin{pmatrix} h[sl].tail.next[0] = null \;\wedge\; h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \;\wedge\; h[sl].tail.next[3] = null \end{pmatrix} & \wedge \\
& & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix} &
\end{array}
$$

- Program **transitions** : $SL_4(h, sl) \wedge \varphi_{aux} \wedge \rho_{36}^{[t]}(V, V')$

$$
SkipList_4(h, sl)
$$

$$
\begin{pmatrix}
x.key & = & 14 & \wedge \\
prev.key & < & 14 & \wedge \\
x.next[0].key & > & 14 & \wedge \\
prev.next[0] & = & x.next[0] & \wedge \\
\multicolumn{3}{r}{(x, 0) \notin sl.r \wedge 0 \leq 0 \leq 3} &
\end{pmatrix}
\wedge
\begin{pmatrix}
at_{36}[t] & \wedge \\
prev'.next[0] = x & \wedge \\
at'_{37}[t] & \wedge \\
h' = h \wedge sl = sl' & \wedge \\
x' = x & \ldots
\end{pmatrix}
$$

```
35: . . .
36: prev.next[0] := x
37: . . .
```



Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList_4(h, sl)$

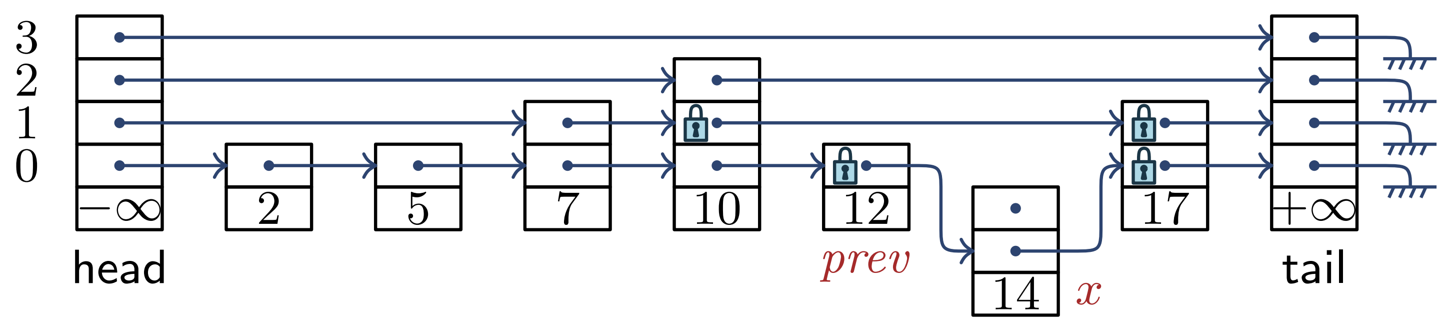
$$
\begin{array}{rcll}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) & \wedge \\
 & & \begin{pmatrix} h[sl].tail.next[0] = null \ \wedge \ h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \ \wedge \ h[sl].tail.next[3] = null \end{pmatrix} & \wedge \\
 & & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix} &
\end{array}
$$

- Program **transitions** : $SL_4(h, sl) \wedge \varphi_{aux} \wedge \rho_{36}^{[t]}(V, V')$

$$
SkipList_4(h, sl)
$$

$$
\begin{pmatrix}
x.key & = & 14 & \wedge \\
prev.key & < & 14 & \wedge \\
x.next[0].key & > & 14 & \wedge \\
prev.next[0] & = & x.next[0] & \wedge \\
\multicolumn{3}{r}{(x, 0) \notin sl.r \wedge 0 \leq 0 \leq 3} &
\end{pmatrix}
\wedge
\begin{pmatrix}
at_{36}[t] & \wedge \\
prev'.next[0] = x & \wedge \\
at'_{37}[t] & \wedge \\
h' = h \wedge sl = sl' & \wedge \\
x' = x & \ldots
\end{pmatrix}
$$

```
35: . . .
36: prev.next[0] := x
37: . . .
```



Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList_4(h, sl)$

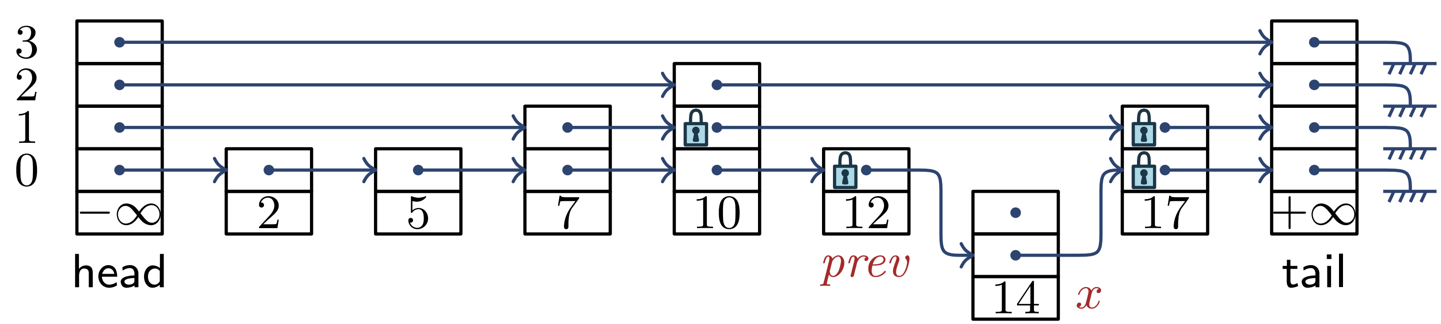
$$
\begin{array}{rcll}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) & \wedge \\
& & \begin{pmatrix} h[sl].tail.next[0] = null \;\wedge\; h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \;\wedge\; h[sl].tail.next[3] = null \end{pmatrix} & \wedge \\
& & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix} &
\end{array}
$$

- Program **transitions** : $SL_4(h, sl) \wedge \varphi_{aux} \wedge \rho^{[t]}_{36}(V, V')$

$$
SkipList_4(h, sl)
$$

$$
\begin{pmatrix}
x.key & = & 14 & \wedge \\
prev.key & < & 14 & \wedge \\
x.next[0].key & > & 14 & \wedge \\
prev.next[0] & = & x.next[0] & \wedge \\
\multicolumn{3}{c}{(x, 0) \notin sl.r \wedge 0 \leq 0 \leq 3} &
\end{pmatrix}
\wedge
\begin{pmatrix}
at_{36}[t] & \wedge \\
prev'.next[0] = x & \wedge \\
at'_{37}[t] & \wedge \\
h' = h \wedge sl = sl' & \wedge \\
x' = x & \ldots
\end{pmatrix}
$$

```
35: . . .
36: prev.next[0] := x
37: . . .
```



Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList_4(h, sl)$

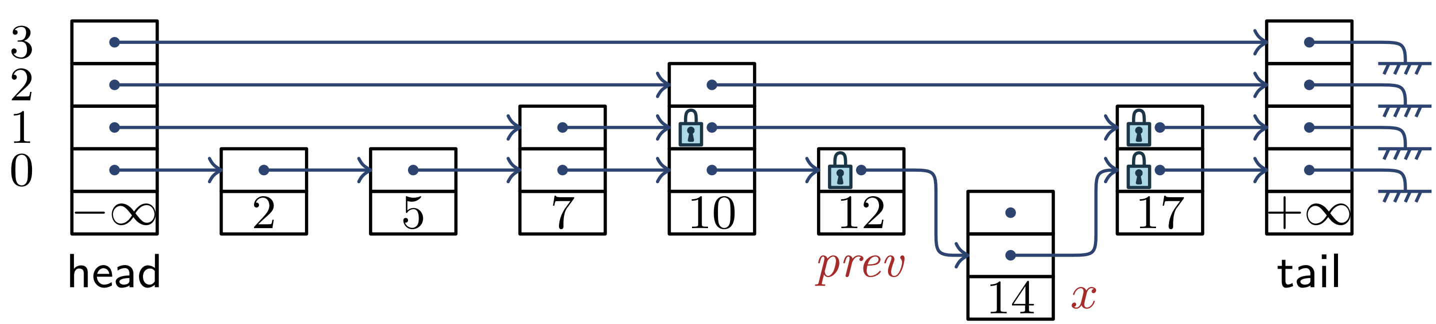
$$
\begin{array}{rcll}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) & \wedge \\
& & \begin{pmatrix} h[sl].tail.next[0] = null \;\wedge\; h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \;\wedge\; h[sl].tail.next[3] = null \end{pmatrix} & \wedge \\
& & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix} &
\end{array}
$$

- Program **transitions** : $SL_4(h, sl) \wedge \varphi_{aux} \wedge \rho_{36}^{[t]}(V, V') \rightarrow SL_4(h', sl')$

$$
SkipList_4(h, sl)
$$

$$
\begin{pmatrix}
x.key & = & 14 & \wedge \\
prev.key & < & 14 & \wedge \\
x.next[0].key & > & 14 & \wedge \\
prev.next[0] & = & x.next[0] & \wedge \\
\multicolumn{3}{c}{(x, 0) \notin sl.r \wedge 0 \leq 0 \leq 3} &
\end{pmatrix}
\wedge
\begin{pmatrix}
at_{36}[t] & \wedge \\
prev'.next[0] = x & \wedge \\
at'_{37}[t] & \wedge \\
h' = h \wedge sl = sl' & \wedge \\
x' = x & \ldots
\end{pmatrix}
\rightarrow SkipList_4(h', sl')
$$

```
35: . . .
36: prev.next[0] := x
37: . . .
```



Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList_4(h, sl)$

$$
\begin{array}{rcl}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) \qquad \wedge \\
& & \begin{pmatrix} h[sl].tail.next[0] = null \ \wedge \ h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \ \wedge \ h[sl].tail.next[3] = null \end{pmatrix} \quad \wedge \\
& & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \ \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \ \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix}
\end{array}
$$

- Program **transitions** : $SL_4(h, sl) \wedge \varphi_{aux} \wedge \rho_{36}^{[t]}(V, V') \rightarrow SL_4(h', sl')$

$$
SkipList_4(h, sl)
$$

$$
\begin{pmatrix}
x.key & = & 14 & \wedge \\
prev.key & < & 14 & \wedge \\
x.next[0].key & > & 14 & \wedge \\
prev.next[0] & = & x.next[0] & \wedge \\
\multicolumn{3}{c}{(x, 0) \notin sl.r \wedge 0 \leq 0 \leq 3} &
\end{pmatrix}
\wedge
\begin{pmatrix}
at_{36}[t] & \wedge \\
prev'.next[0] = x & \wedge \\
at'_{37}[t] & \wedge \\
h' = h \wedge sl = sl' & \wedge \\
x' = x & \ldots
\end{pmatrix}
\rightarrow SkipList_4(h', sl')
$$

reason about

locks

Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList_4(h, sl)$

$$
\begin{array}{rcll}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) & \wedge \\
& & \begin{pmatrix} h[sl].tail.next[0] = null \ \wedge \ h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \ \wedge \ h[sl].tail.next[3] = null \end{pmatrix} & \wedge \\
& & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix} &
\end{array}
$$

- Program **transitions** : $SL_4(h, sl) \wedge \varphi_{aux} \wedge \rho_{36}^{[t]}(V, V') \rightarrow SL_4(h', sl')$

$$
SkipList_4(h, sl)
\begin{pmatrix}
x.key & = & 14 & \wedge \\
prev.key & < & 14 & \wedge \\
x.next[0].key & > & 14 & \wedge \\
prev.next[0] & = & x.next[0] & \wedge \\
\multicolumn{3}{c}{(x, 0) \notin sl.r \wedge 0 \leq 0 \leq 3} &
\end{pmatrix}
\wedge
\begin{pmatrix}
at_{36}[t] & \wedge \\
prev'.next[0] = x & \wedge \\
at'_{37}[t] & \wedge \\
h' = h \wedge sl = sl' & \wedge \\
x' = x & \ldots
\end{pmatrix}
\rightarrow SkipList_4(h', sl')
$$

reason about

thread identifiers

Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList_4(h, sl)$

$$
\begin{array}{rcll}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) & \wedge \\
& & \begin{pmatrix} h[sl].tail.next[0] = null \ \wedge \ h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \ \wedge \ h[sl].tail.next[3] = null \end{pmatrix} & \wedge \\
& & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix} &
\end{array}
$$

- Program **transitions** : $SL_4(h, sl) \wedge \varphi_{aux} \wedge \rho_{36}^{[t]}(V, V') \rightarrow SL_4(h', sl')$

$$
SkipList_4(h, sl)
$$

$$
\begin{pmatrix}
x.key & = & 14 & \wedge \\
prev.key & < & 14 & \wedge \\
x.next[0].key & > & 14 & \wedge \\
prev.next[0] & = & x.next[0] & \wedge \\
\multicolumn{3}{c}{(x, 0) \notin sl.r \wedge 0 \leq 0 \leq 3} &
\end{pmatrix}
\wedge
\begin{pmatrix}
at_{36}[t] & \wedge \\
prev'.next[0] = x & \wedge \\
at'_{37}[t] & \wedge \\
h' = h \wedge sl = sl' & \wedge \\
x' = x & \ldots
\end{pmatrix}
\rightarrow SkipList_4(h', sl')
$$

reason about

program positions

Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList_4(h, sl)$

$$
\begin{array}{rcll}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) & \wedge \\
& & \begin{pmatrix} h[sl].tail.next[0] = null \ \wedge \ h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \ \wedge \ h[sl].tail.next[3] = null \end{pmatrix} & \wedge \\
& & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix} &
\end{array}
$$

- Program **transitions** : $SL_4(h, sl) \wedge \varphi_{aux} \wedge \rho_{36}^{[t]}(V, V') \rightarrow SL_4(h', sl')$

$$
SkipList_4(h, sl)
$$

$$
\begin{pmatrix}
x.key & = & 14 & \wedge \\
prev.key & < & 14 & \wedge \\
x.next[0].key & > & 14 & \wedge \\
prev.next[0] & = & x.next[0] & \wedge \\
\multicolumn{3}{c}{(x, 0) \notin sl.r \wedge 0 \leq 0 \leq 3} &
\end{pmatrix}
\wedge
\begin{pmatrix}
at_{36}[t] & \wedge \\
prev'.next[0] = x & \wedge \\
at'_{37}[t] & \wedge \\
h' = h \wedge sl = sl' & \wedge \\
x' = x & \ldots
\end{pmatrix}
\rightarrow SkipList_4(h', sl')
$$

reason about

ordered values + notion of ordered list

Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList_4(h, sl)$

$$
\begin{array}{rcll}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) & \wedge \\
& & \begin{pmatrix} h[sl].tail.next[0] = null \ \wedge \ h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \ \wedge \ h[sl].tail.next[3] = null \end{pmatrix} & \wedge \\
& & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix} &
\end{array}
$$

- Program **transitions** : $SL_4(h, sl) \wedge \varphi_{aux} \wedge \rho_{36}^{[t]}(V, V') \rightarrow SL_4(h', sl')$

$$
SkipList_4(h, sl)
$$

$$
\begin{pmatrix}
x.key & = & 14 & \wedge \\
prev.key & < & 14 & \wedge \\
x.next[0].key & > & 14 & \wedge \\
prev.next[0] & = & x.next[0] & \wedge \\
(x, 0) \notin sl.r \wedge 0 \leq 0 \leq 3 & & &
\end{pmatrix}
\wedge
\begin{pmatrix}
at_{36}[t] & \wedge \\
prev'.next[0] = x & \wedge \\
at'_{37}[t] & \wedge \\
h' = h \wedge sl = sl' & \wedge \\
x' = x & \ldots
\end{pmatrix}
\rightarrow SkipList_4(h', sl')
$$

reason about

levels

Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList_4(h, sl)$

$$
\begin{array}{rcl}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) \qquad \wedge \\
& & \begin{pmatrix} h[sl].tail.next[0] = null \;\wedge\; h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \;\wedge\; h[sl].tail.next[3] = null \end{pmatrix} \qquad \wedge \\
& & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix}
\end{array}
$$

- Program **transitions** : $SL_4(h, sl) \wedge \varphi_{aux} \wedge \rho_{36}^{[t]}(V, V') \to SL_4(h', sl')$

$$
SkipList_4(h, sl)
$$

$$
\begin{pmatrix}
x.key & = & 14 & \wedge \\
prev.key & < & 14 & \wedge \\
x.next[0].key & > & 14 & \wedge \\
prev.next[0] & = & x.next[0] & \wedge \\
\multicolumn{3}{r}{(x, 0) \notin sl.r \wedge 0 \leq 0 \leq 3} &
\end{pmatrix}
\wedge
\begin{pmatrix}
at_{36}[t] & \wedge \\
prev'.next[0] = x & \wedge \\
at'_{37}[t] & \wedge \\
h' = h \wedge sl = sl' & \wedge \\
x' = x & \ldots
\end{pmatrix}
\to SkipList_4(h', sl')
$$

reason about

masked regions

Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\square SkipList_4(h, sl)$

$$
\begin{array}{rcll}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) & \wedge \\
& & \left(\begin{array}{l} h[sl].tail.next[0] = null \;\wedge\; h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \;\wedge\; h[sl].tail.next[3] = null \end{array}\right) & \wedge \\
& & \left(\begin{array}{l} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{array}\right) &
\end{array}
$$

- Program **transitions** : $SL_4(h, sl) \wedge \varphi_{aux} \wedge \rho_{36}^{[t]}(V, V') \rightarrow SL_4(h', sl')$

$$
SkipList_4(h, sl)
$$

$$
\left(\begin{array}{rcll}
x.key & = & 14 & \wedge \\
prev.key & < & 14 & \wedge \\
x.next[0].key & > & 14 & \wedge \\
prev.next[0] & = & x.next[0] & \wedge \\
\multicolumn{3}{c}{(x, 0) \notin sl.r \wedge 0 \leq 0 \leq 3} &
\end{array}\right) \wedge
\left(\begin{array}{cl}
at_{36}[t] & \wedge \\
prev'.next[0] = x & \wedge \\
at'_{37}[t] & \wedge \\
h' = h \wedge sl = sl' & \wedge \\
x' = x & \ldots
\end{array}\right) \rightarrow SkipList_4(h', sl')
$$

reason about

memory, cells

Verification of Concurrent Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList_4(h, sl)$

$$
\begin{array}{rcll}
SkipList_4(h, sl : SkipList) & \hat{=} & OList(h, sl, 0) & \wedge \\
& & \begin{pmatrix} h[sl].tail.next[0] = null \ \wedge \ h[sl].tail.next[1] = null \\ h[sl].tail.next[2] = null \ \wedge \ h[sl].tail.next[3] = null \end{pmatrix} & \wedge \\
& & \begin{pmatrix} SubList(h, sl.head, sl.tail, 1, sl.head, sl.tail, 0) \wedge \\ SubList(h, sl.head, sl.tail, 2, sl.head, sl.tail, 1) \wedge \\ SubList(h, sl.head, sl.tail, 3, sl.head, sl.tail, 2) \end{pmatrix} &
\end{array}
$$

- Program **transitions** : $SL_4(h, sl) \wedge \varphi_{aux} \wedge \rho_{36}^{[t]}(V, V') \rightarrow SL_4(h', sl')$

$$
SkipList_4(h, sl)
$$

$$
\begin{pmatrix}
x.key & = & 14 & \wedge \\
prev.key & < & 14 & \wedge \\
x.next[0].key & > & 14 & \wedge \\
prev.next[0] & = & x.next[0] & \wedge \\
\multicolumn{3}{c}{(x, 0) \notin sl.r \wedge 0 \leq 0 \leq 3} &
\end{pmatrix}
\wedge
\begin{pmatrix}
at_{36}[t] & \wedge \\
prev'.next[0] = x & \wedge \\
at'_{37}[t] & \wedge \\
h' = h \wedge sl = sl' & \wedge \\
x' = x & \ldots
\end{pmatrix}
\rightarrow SkipList_4(h', sl')
$$

reason about

sublist

Our Contribution

- **TSL_K**, a theory for concurrent skiplist lists of height K
- We show TSL_K **decidable**
- We propose a combination based **decision procedure**

TSL_K: A Theory for Concurrent Skiplists of height K

- Based on Theory of Linked Lists (**TLL**)
- Each element is **ordered** by a **key**
- **Reasoning** on levels is extended to **K levels**
- Regions extended to **masked regions**
- List extended to **ordered lists** and **subpaths** of ordered lists

TSL_K: A Theory for Concurrent Skiplists

- TSL_K is a union of other theories

TSL_K: A Theory for Concurrent Skiplists

- TSL_K is a union of other theories

$$\Sigma_{\mathsf{addr}}$$

$\mathsf{TSL_K}$: A Theory for Concurrent Skiplists

- $\mathsf{TSL_K}$ is a union of other theories

$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}}$$

TSL_K: A Theory for Concurrent Skiplists

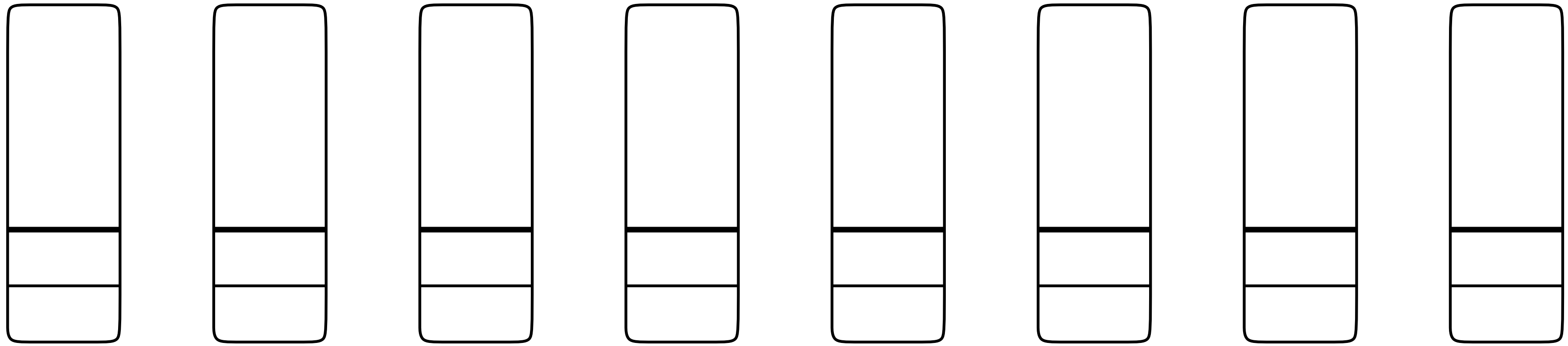
- TSL_K is a union of other theories

$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}}$$

TSL_K: A Theory for Concurrent Skiplists

- TSL_K is a union of other theories

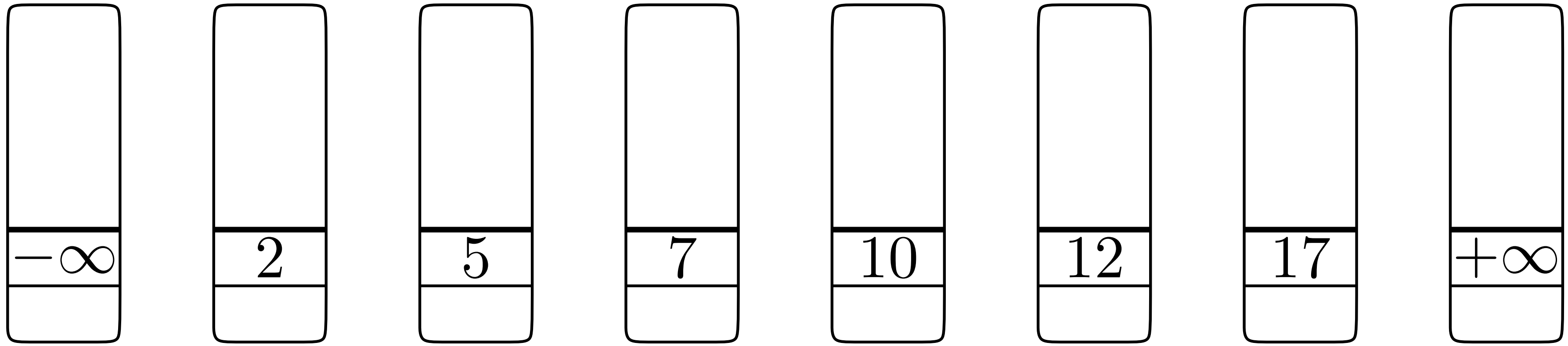
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}}$$



TSL_K: A Theory for Concurrent Skiplists

- TSL_K is a union of other theories

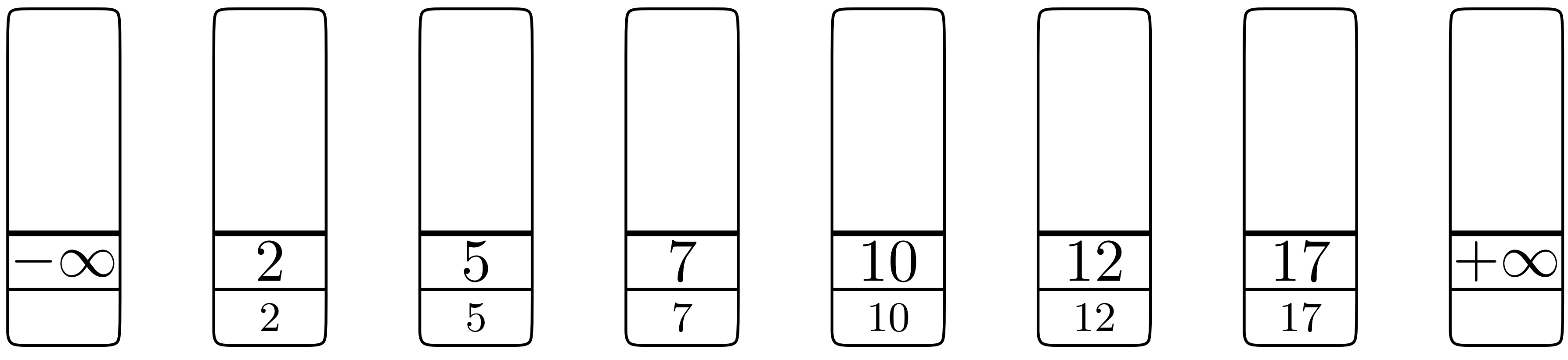
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}}$$



$\mathsf{TSL_K}$: A Theory for Concurrent Skiplists

- $\mathsf{TSL_K}$ is a union of other theories

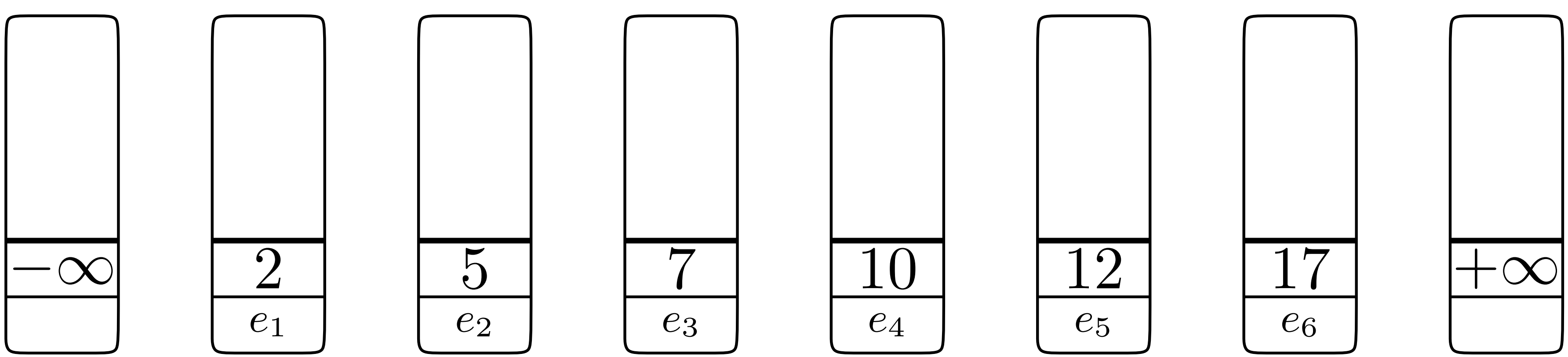
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}}$$



TSL_K: A Theory for Concurrent Skiplists

- TSL_K is a union of other theories

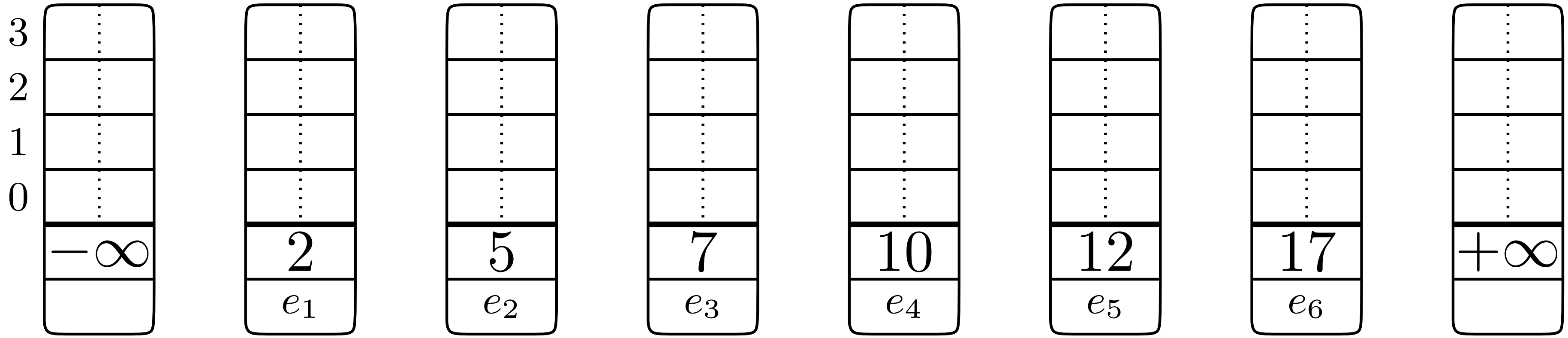
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}}$$



TSL_K: A Theory for Concurrent Skiplists

- TSL_K is a union of other theories

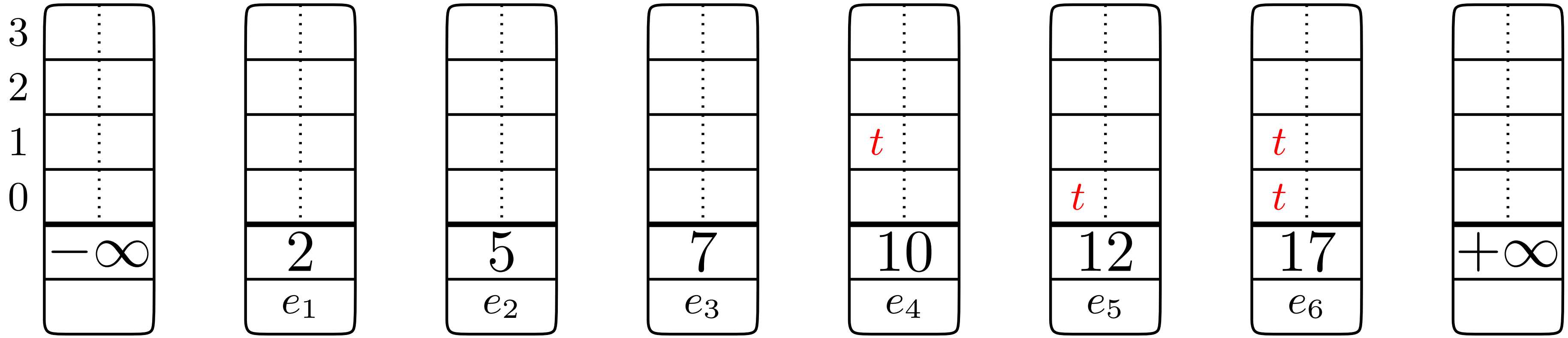
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level}_K}$$



$\mathsf{TSL_K}$: A Theory for Concurrent Skiplists

- $\mathsf{TSL_K}$ is a union of other theories

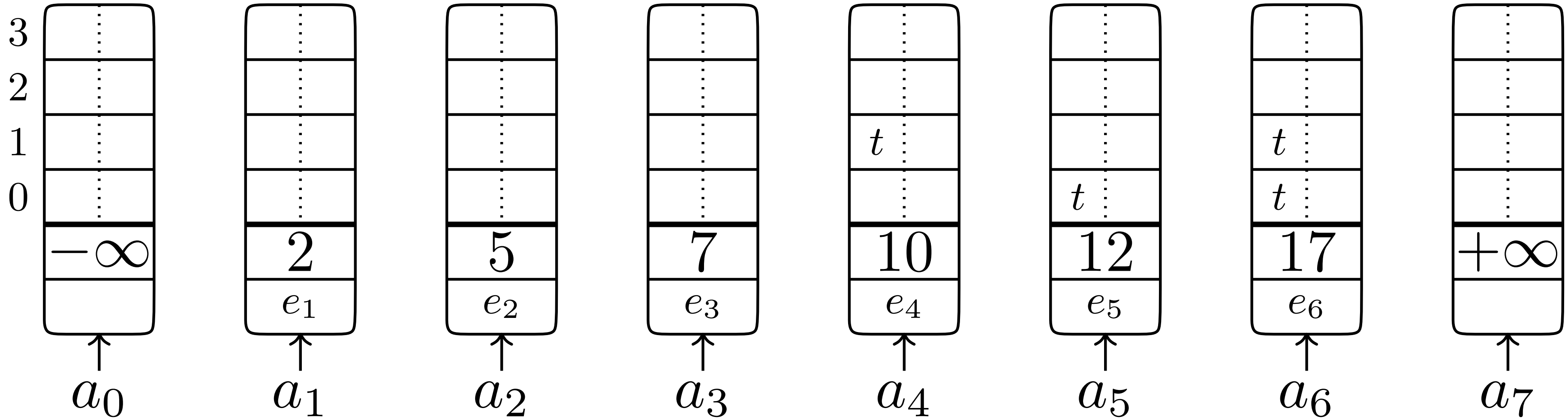
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{thid}}$$



TSL_K: A Theory for Concurrent Skiplists

- TSL_K is a union of other theories

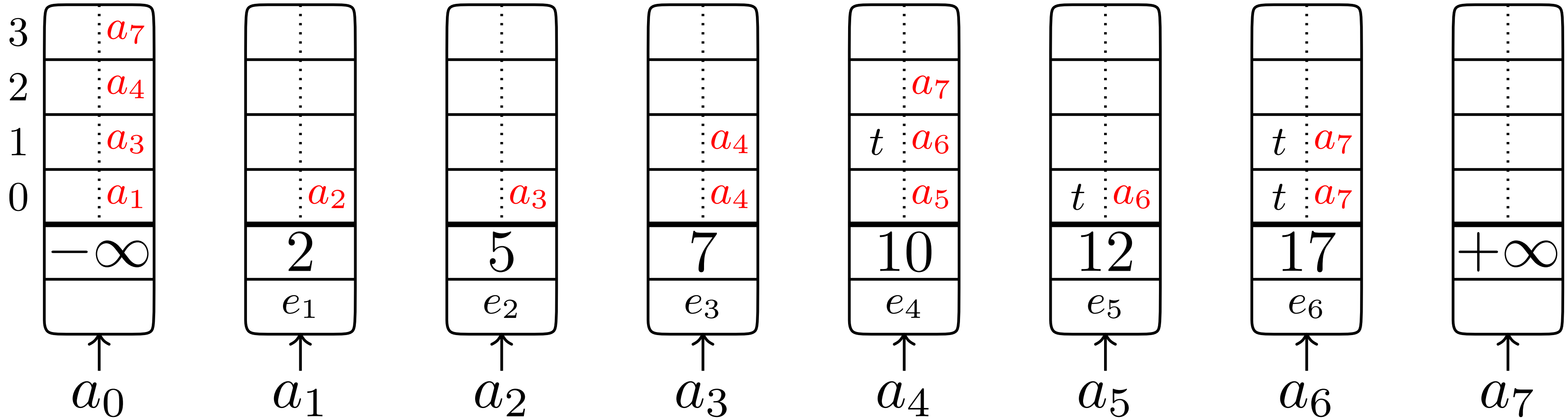
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level}_\mathsf{K}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}}$$



$\mathsf{TSL_K}$: A Theory for Concurrent Skiplists

- $\mathsf{TSL_K}$ is a union of other theories

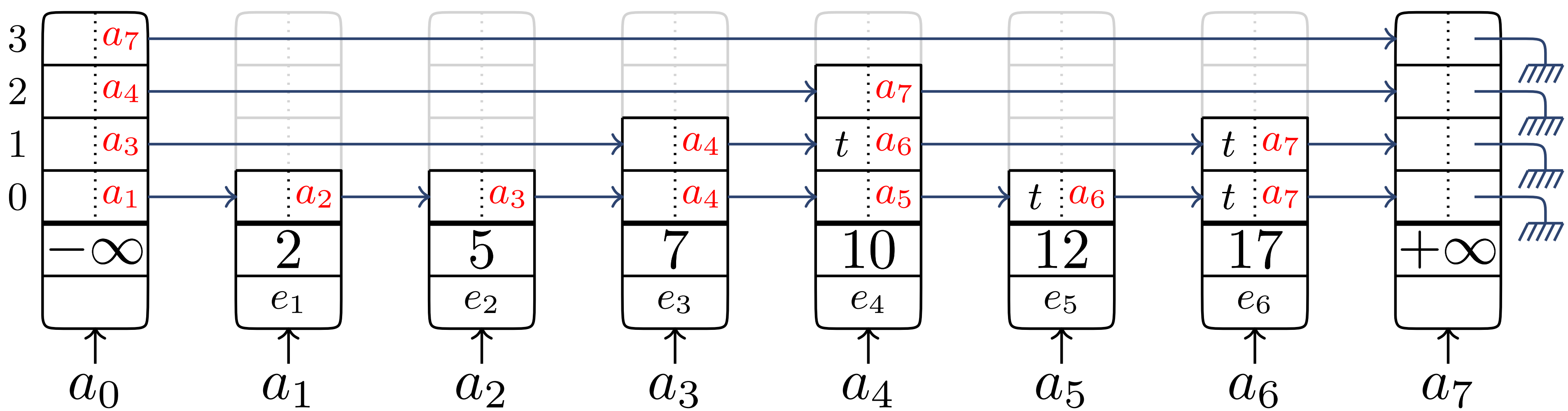
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}}$$



TSL$_K$: A Theory for Concurrent Skiplists

- TSL$_K$ is a union of other theories

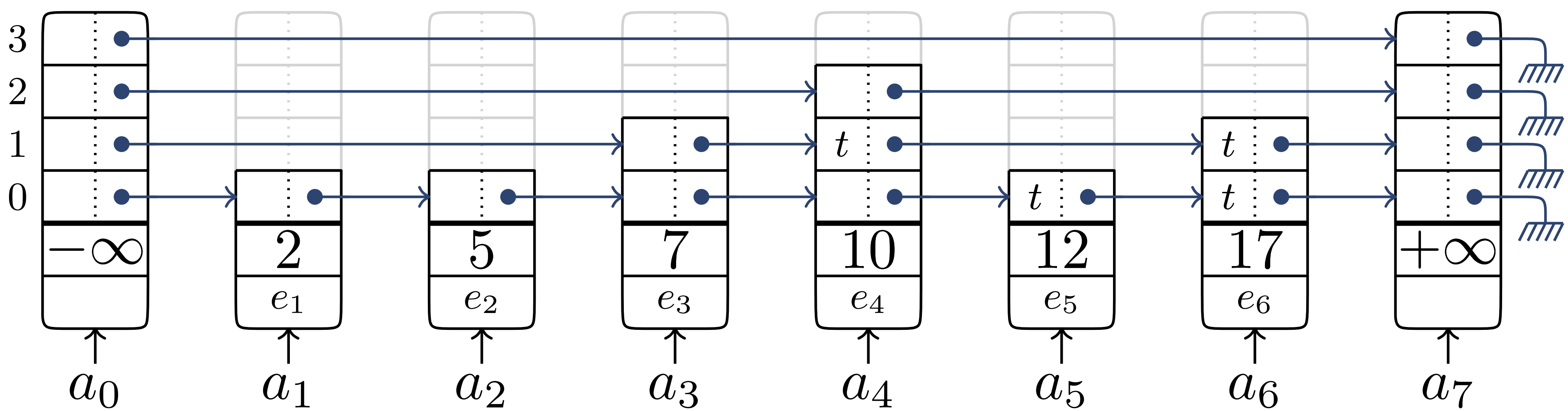
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level}_K} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}}$$



$\mathsf{TSL_K}$: A Theory for Concurrent Skiplists

- $\mathsf{TSL_K}$ is a union of other theories

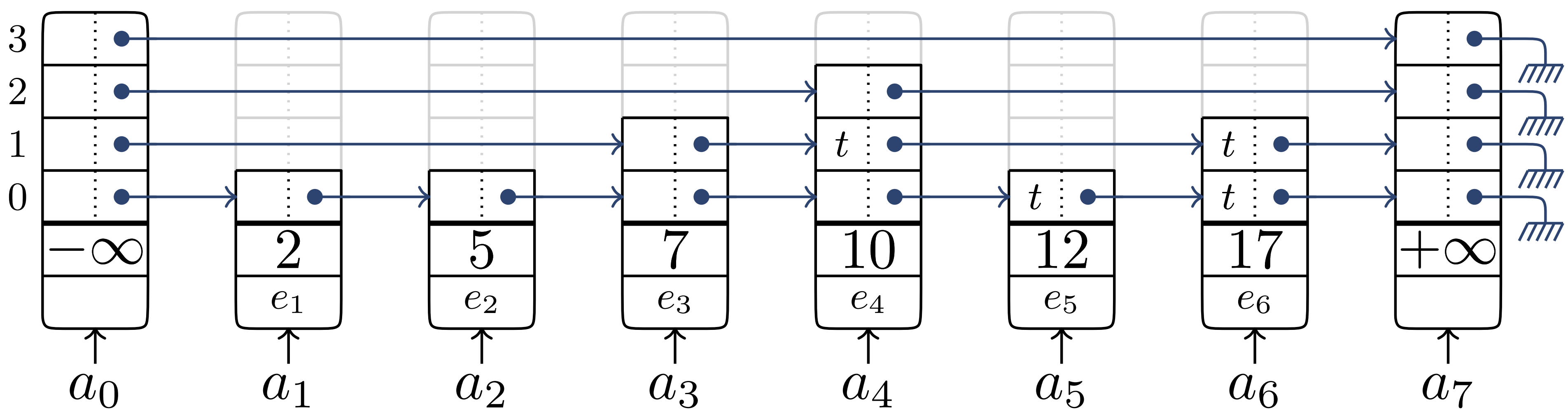
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}}$$



$\mathsf{TSL_K}$: A Theory for Concurrent Skiplists

- $\mathsf{TSL_K}$ is a union of other theories

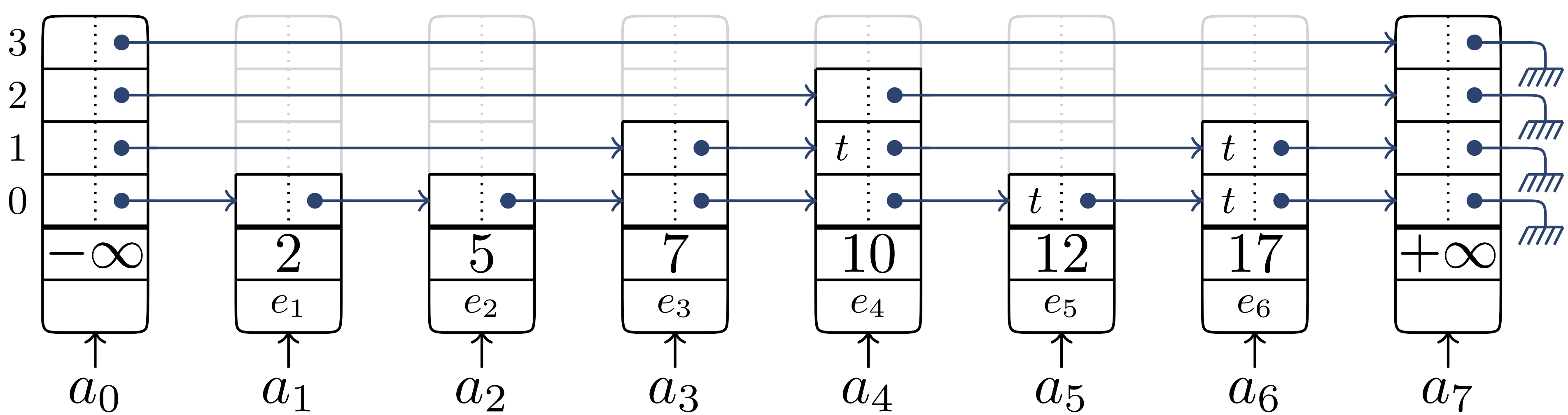
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{set}}$$



$\mathsf{TSL_K}$: A Theory for Concurrent Skiplists

- $\mathsf{TSL_K}$ is a union of other theories

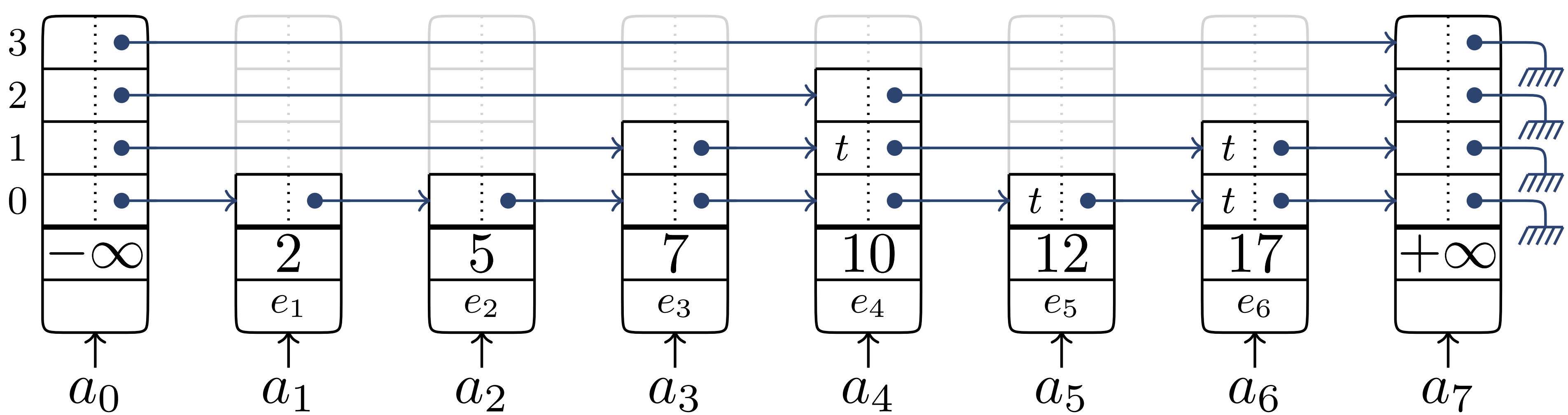
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{setth}}$$



$\mathsf{TSL_K}$: A Theory for Concurrent Skiplists

- $\mathsf{TSL_K}$ is a union of other theories

$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{setth}} \cup \Sigma_{\mathsf{mrgn}}$$



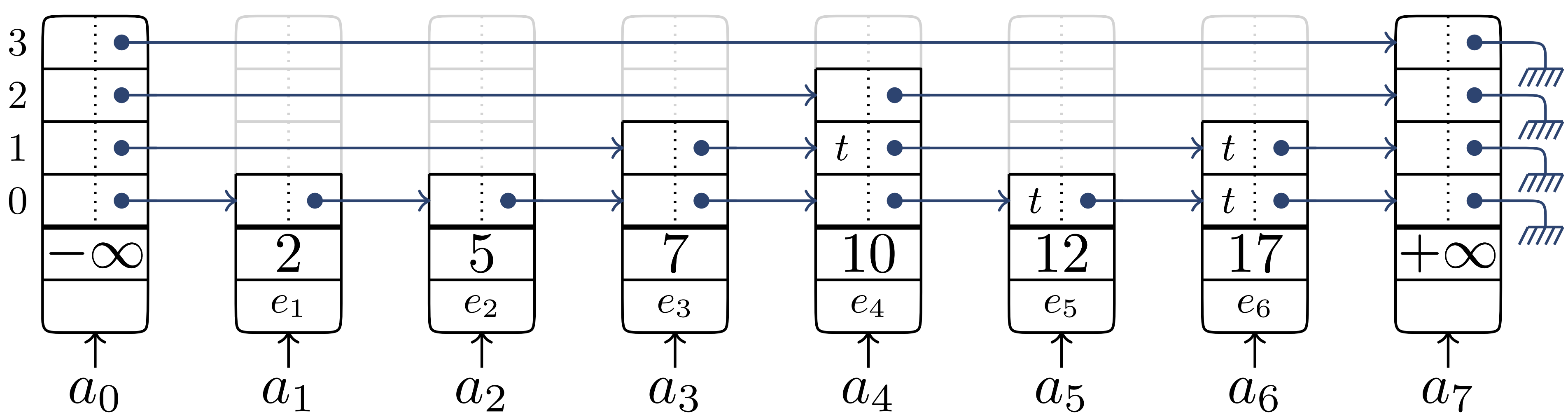
TSL$_\mathsf{K}$: A Theory for Concurrent Skiplists

- TSL$_\mathsf{K}$ is a union of other theories

$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{setth}} \cup \Sigma_{\mathsf{mrgn}} \cup \Sigma_{\mathsf{reachability}}$$

path = a non-repeating sequence of addresses

$$[a_1, a_2, a_3]$$

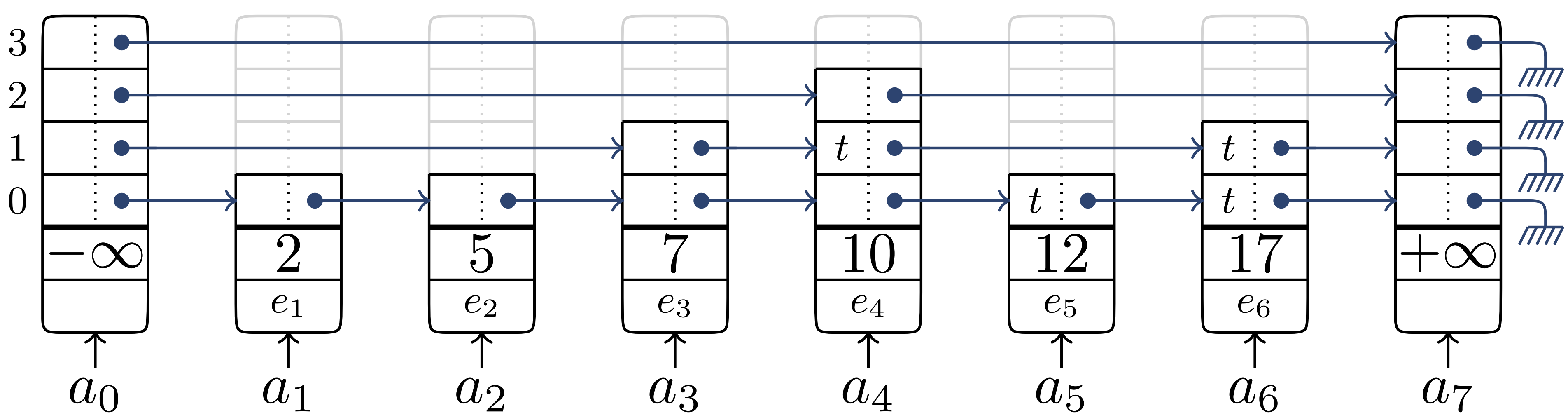


TSL_K: A Theory for Concurrent Skiplists

- TSL_K is a union of other theories

$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level}_\mathsf{K}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{setth}} \cup \Sigma_{\mathsf{mrgn}} \cup \Sigma_{\mathsf{reachability}}$$

$$append([a_1, a_2], [a_3], [a_1, a_2, a_3])$$

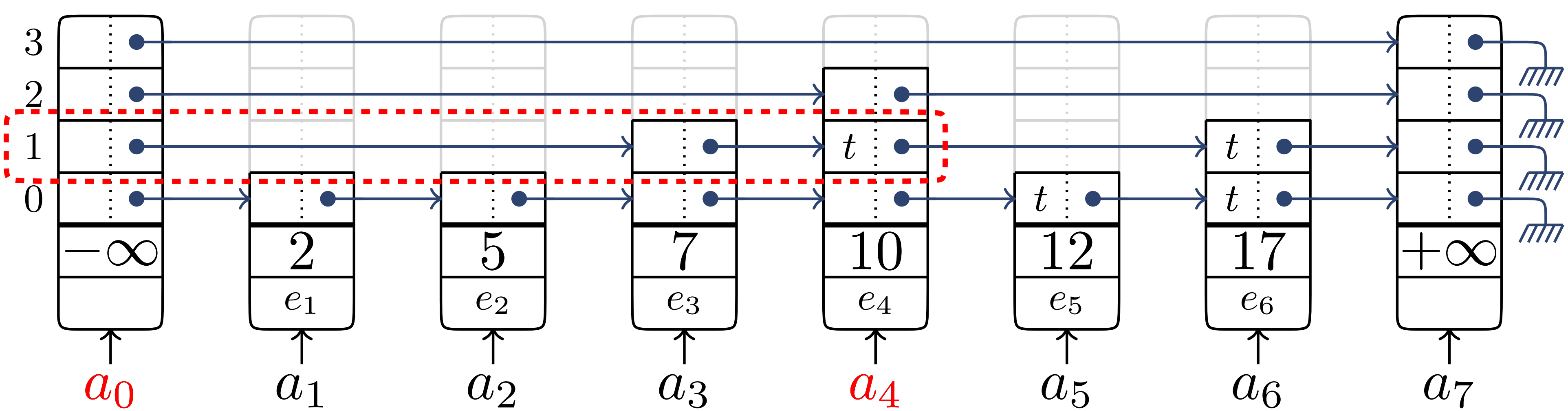


$\mathsf{TSL_K}$: A Theory for Concurrent Skiplists

- $\mathsf{TSL_K}$ is a union of other theories

$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{setth}} \cup \Sigma_{\mathsf{mrgn}} \cup \Sigma_{\mathsf{reachability}}$$

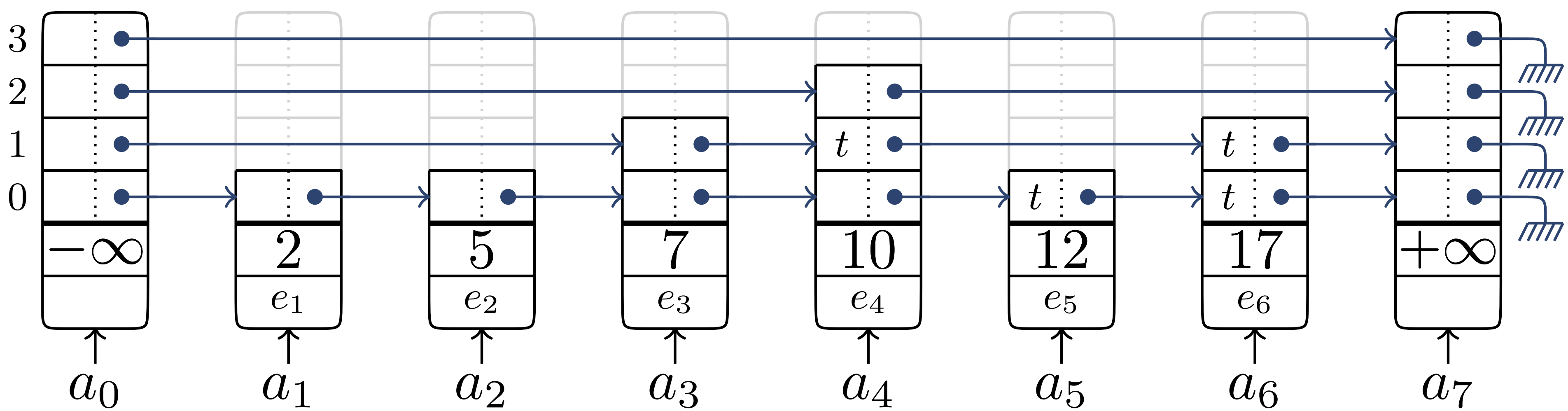
$$reach_{\mathsf{K}}(a_0, a_4, 1, [a_0, a_3])$$



$\mathsf{TSL_K}$: A Theory for Concurrent Skiplists

- $\mathsf{TSL_K}$ is a union of other theories

$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{setth}} \cup \Sigma_{\mathsf{mrgn}} \cup \Sigma_{\mathsf{reachability}} \cup \Sigma_{\mathsf{bridge}}$$

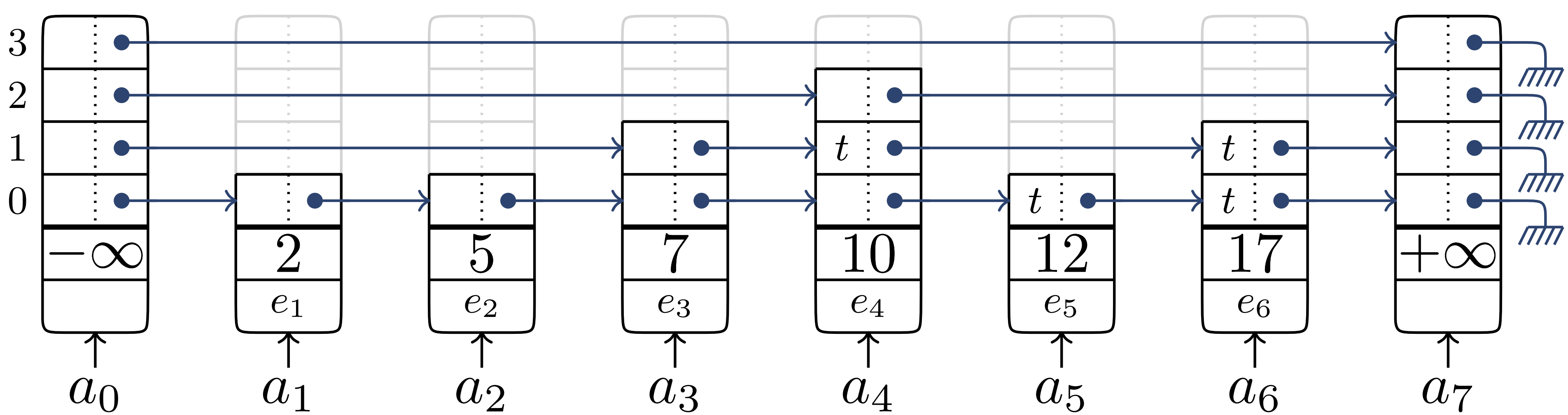


TSL$_\mathsf{K}$: A Theory for Concurrent Skiplists

- TSL$_\mathsf{K}$ is a union of other theories

$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}} \cup$$
$$\Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{setth}} \cup \Sigma_{\mathsf{mrgn}} \cup \Sigma_{\mathsf{reachability}} \cup \Sigma_{\mathsf{bridge}}$$

$$path2set([a_2, a_3]) = \{a_2, a_3\}$$

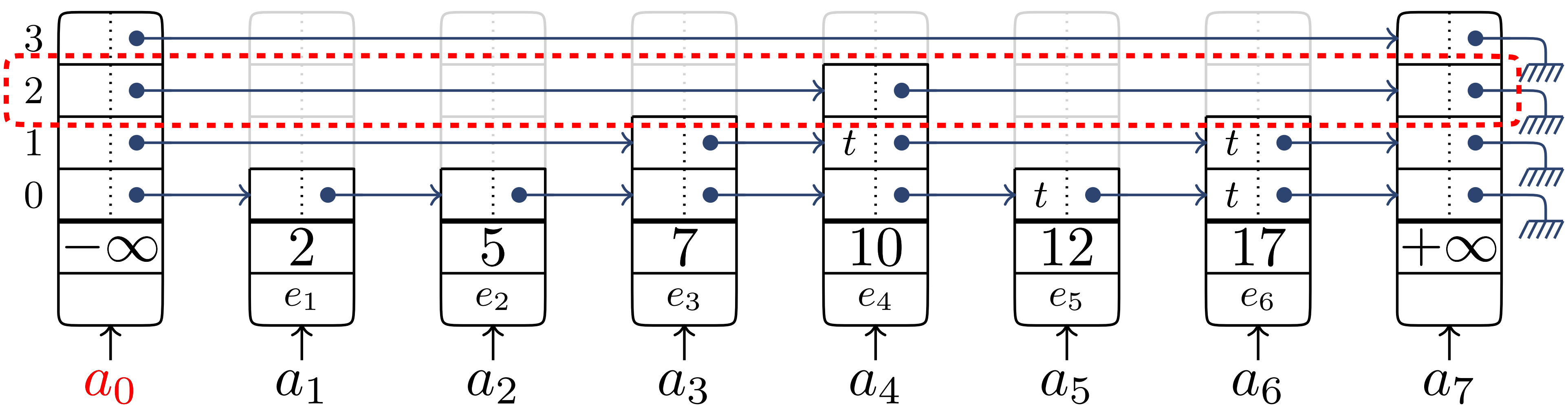


$\mathsf{TSL_K}$: A Theory for Concurrent Skiplists

- $\mathsf{TSL_K}$ is a union of other theories

$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{setth}} \cup \Sigma_{\mathsf{mrgn}} \cup \Sigma_{\mathsf{reachability}} \cup \Sigma_{\mathsf{bridge}}$$

$$addr2set_{\mathsf{K}}(a_0, 2) = \{a_0, a_4, a_7\}$$

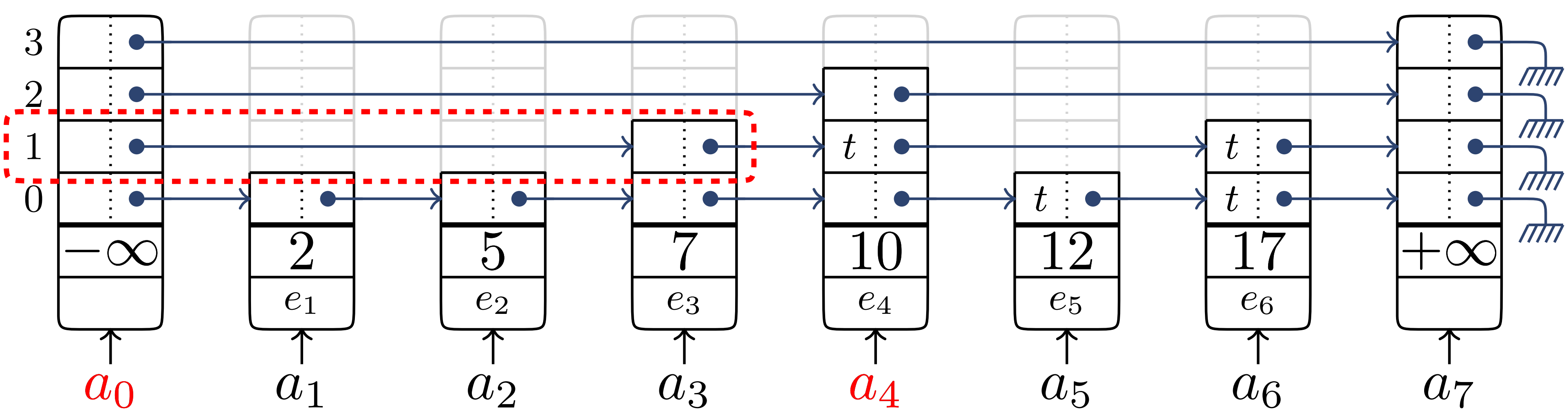


$\mathsf{TSL_K}$: A Theory for Concurrent Skiplists

- $\mathsf{TSL_K}$ is a union of other theories

$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}} \cup$$
$$\Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{setth}} \cup \Sigma_{\mathsf{mrgn}} \cup \Sigma_{\mathsf{reachability}} \cup \Sigma_{\mathsf{bridge}}$$

$$getp_{\mathsf{K}}(a_0, a_4, 1) = [a_0, a_3]$$

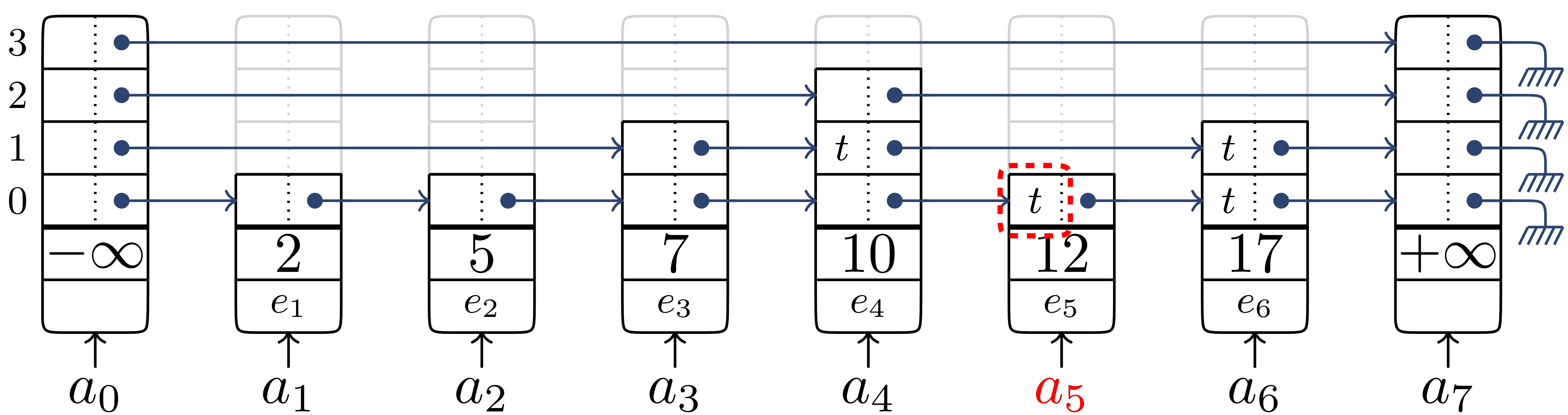


$\mathsf{TSL_K}$: A Theory for Concurrent Skiplists

- $\mathsf{TSL_K}$ is a union of other theories

$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{setth}} \cup \Sigma_{\mathsf{mrgn}} \cup \Sigma_{\mathsf{reachability}} \cup \Sigma_{\mathsf{bridge}}$$

$$\mathit{firstlocked}_{\mathsf{K}}([a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7], 0) = a_5$$

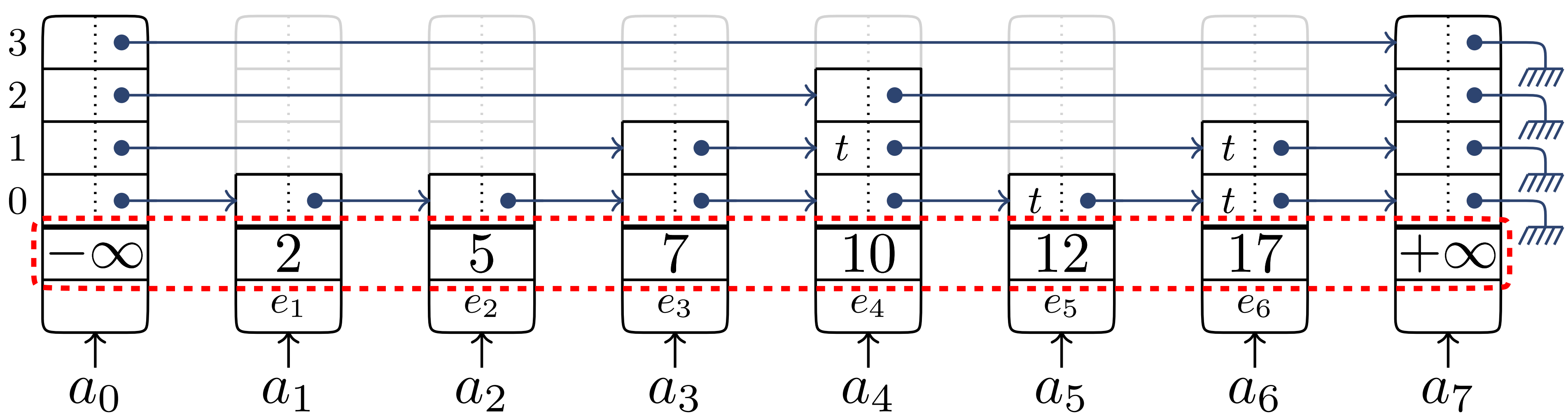


$\mathsf{TSL_K}$: A Theory for Concurrent Skiplists

- $\mathsf{TSL_K}$ is a union of other theories

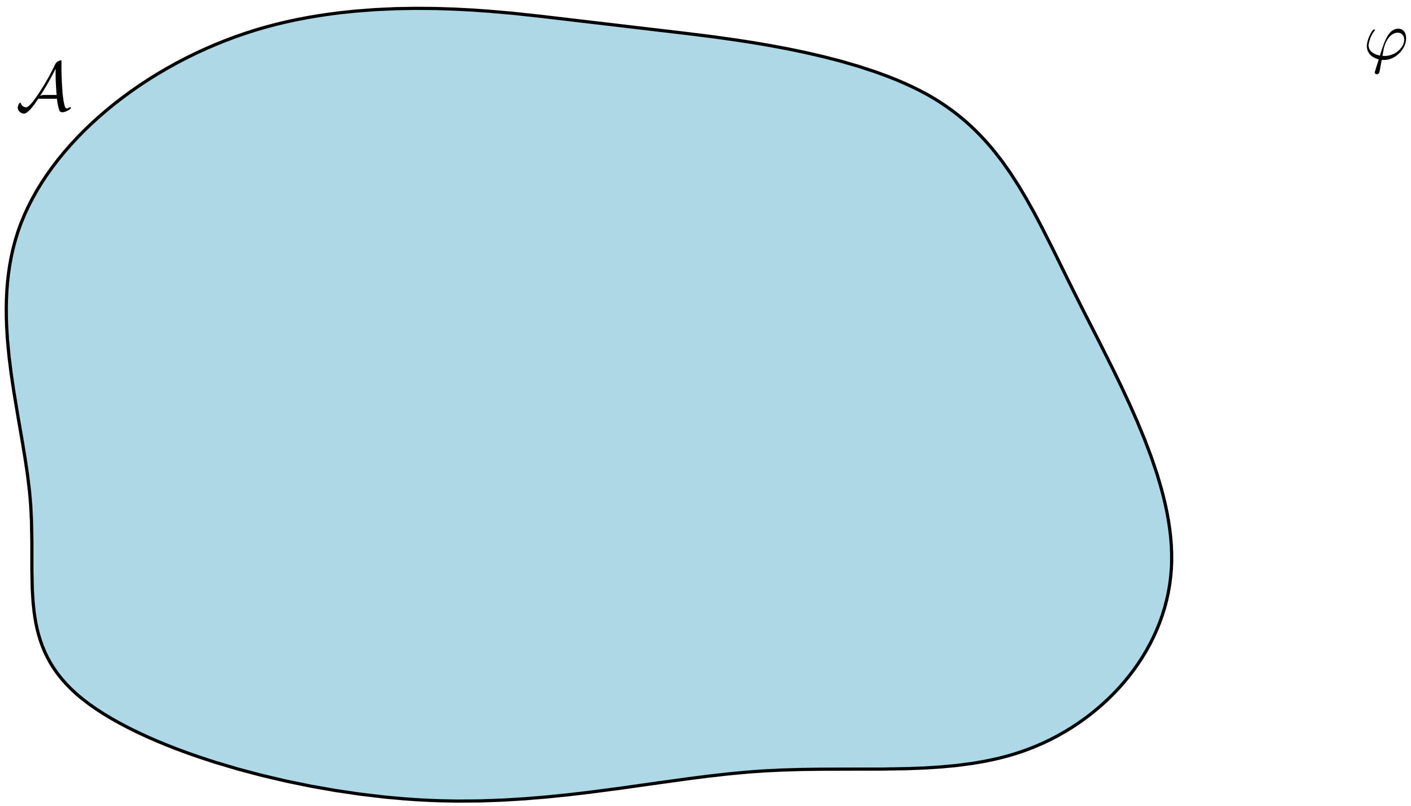
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{setth}} \cup \Sigma_{\mathsf{mrgn}} \cup \Sigma_{\mathsf{reachability}} \cup \Sigma_{\mathsf{bridge}}$$

$$ordList([a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7])$$

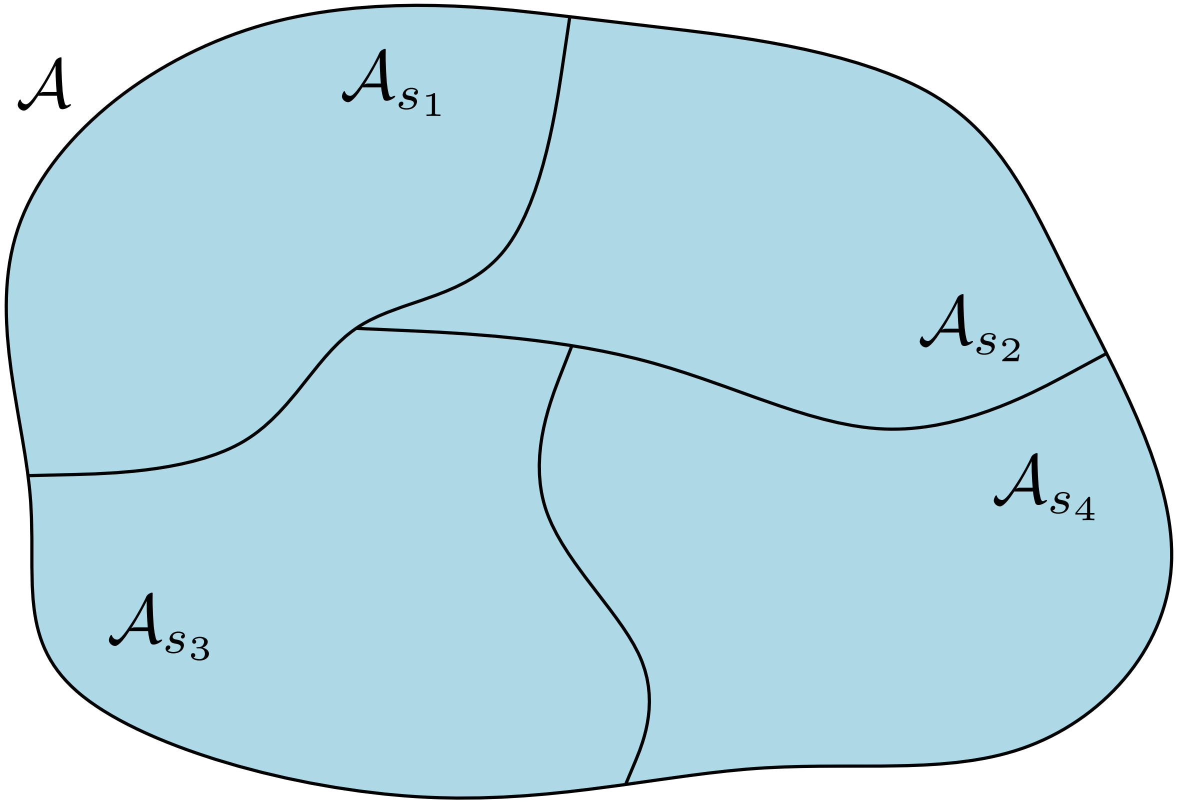


Decidability by Small Model Property (SMP)

Decidability by Small Model Property (SMP)

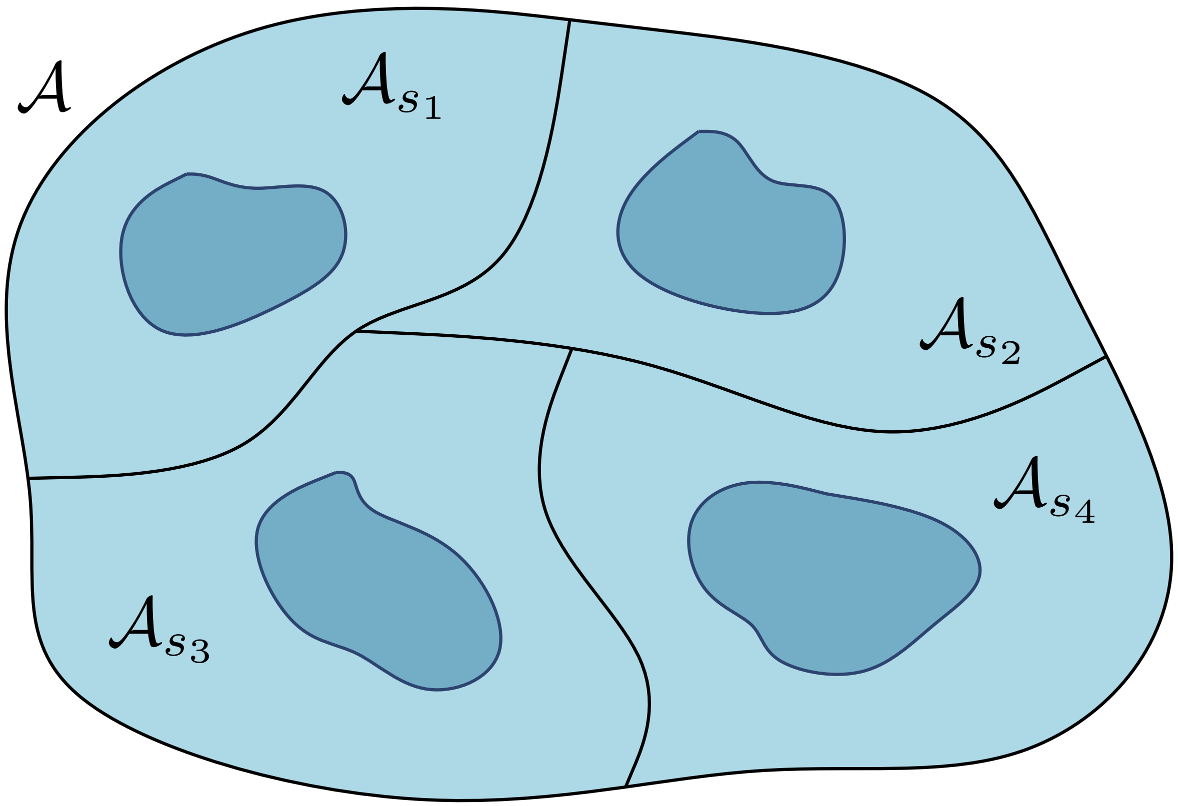


Decidability by Small Model Property (SMP)



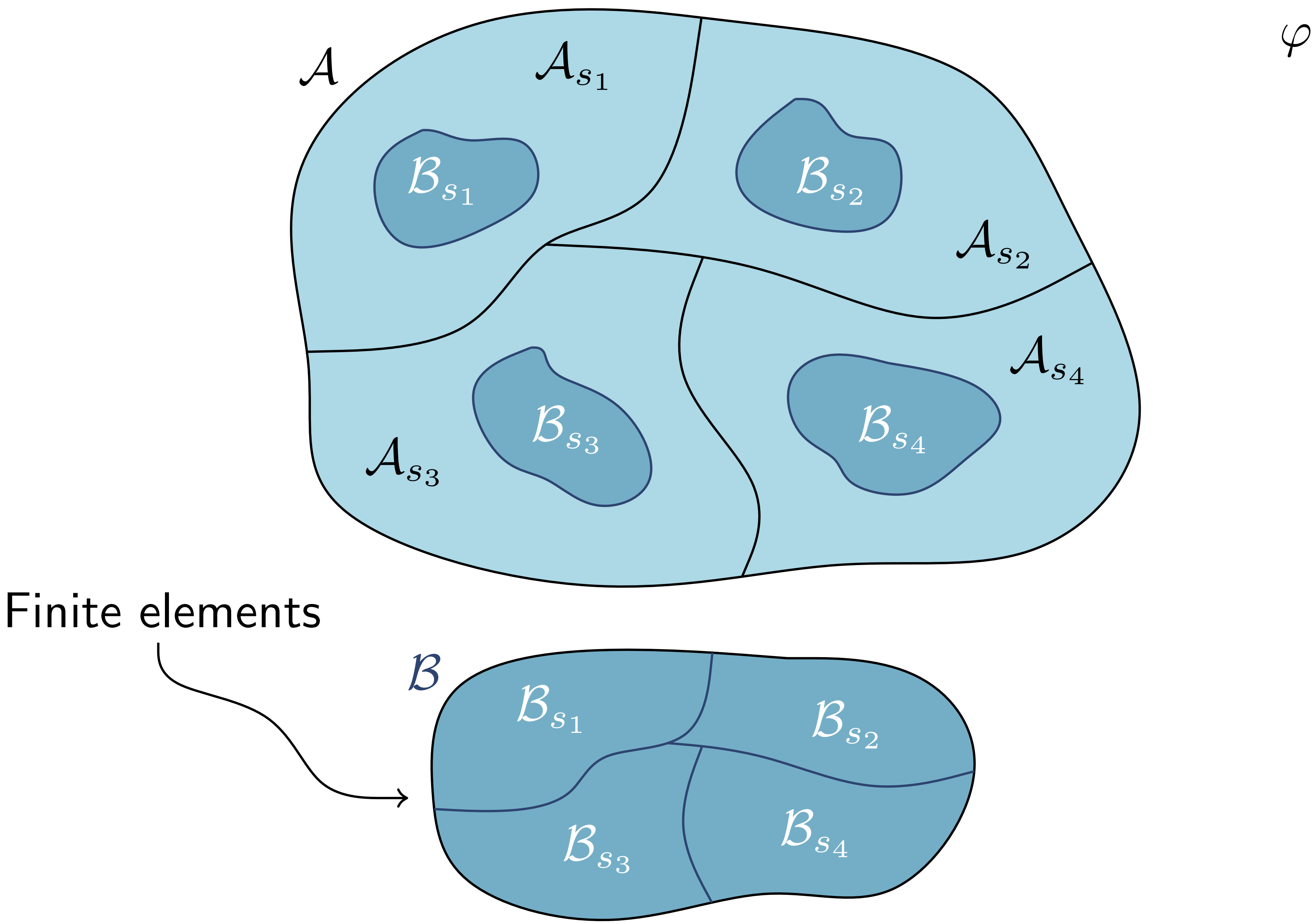
φ

Decidability by Small Model Property (SMP)



φ

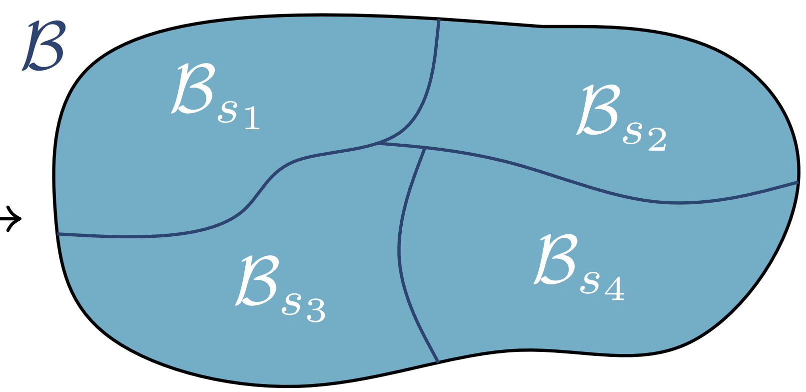
Decidability by Small Model Property (SMP)



Decidability by Small Model Property (SMP)

- Let Γ be a conjunction of $\mathsf{TSL_K}$-literals

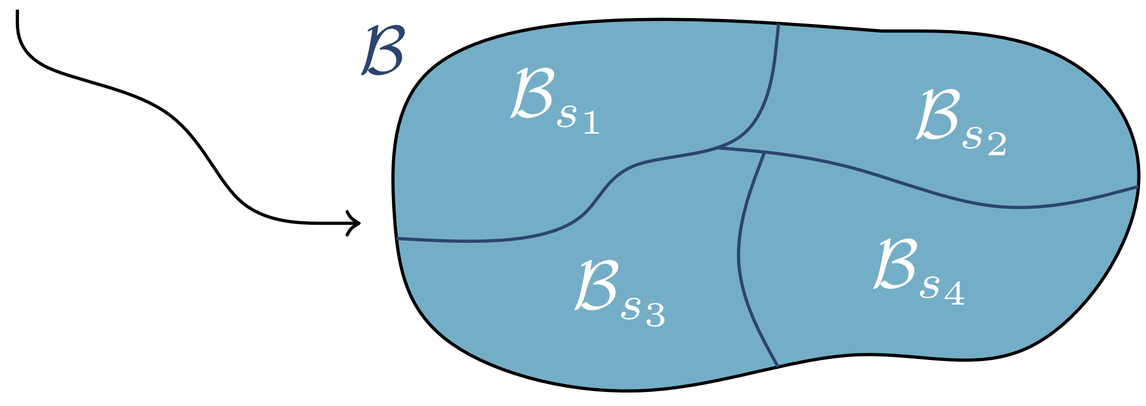
Finite elements



Decidability by Small Model Property (SMP)

- Let Γ be a conjunction of $\mathsf{TSL_K}$-literals
- If Γ is satisfied in an arbitrary $\mathsf{TSL_K}$ interpretation $\mathcal{A}$

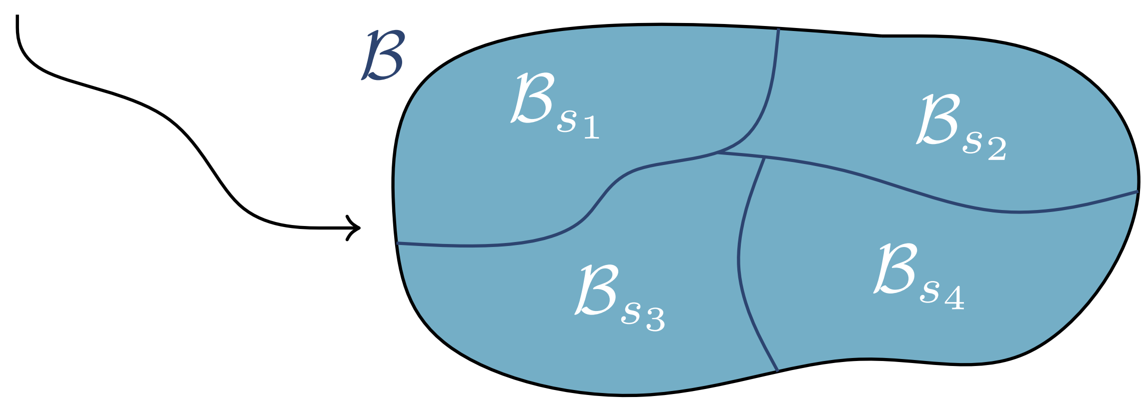
Finite elements



Decidability by Small Model Property (SMP)

- Let Γ be a conjunction of TSL_K-literals
- If Γ is satisfied in an arbitrary TSL_K interpretation $\mathcal{A}$
- We construct a new finite interpretation $\mathcal{B}$ **bounded by** Γ

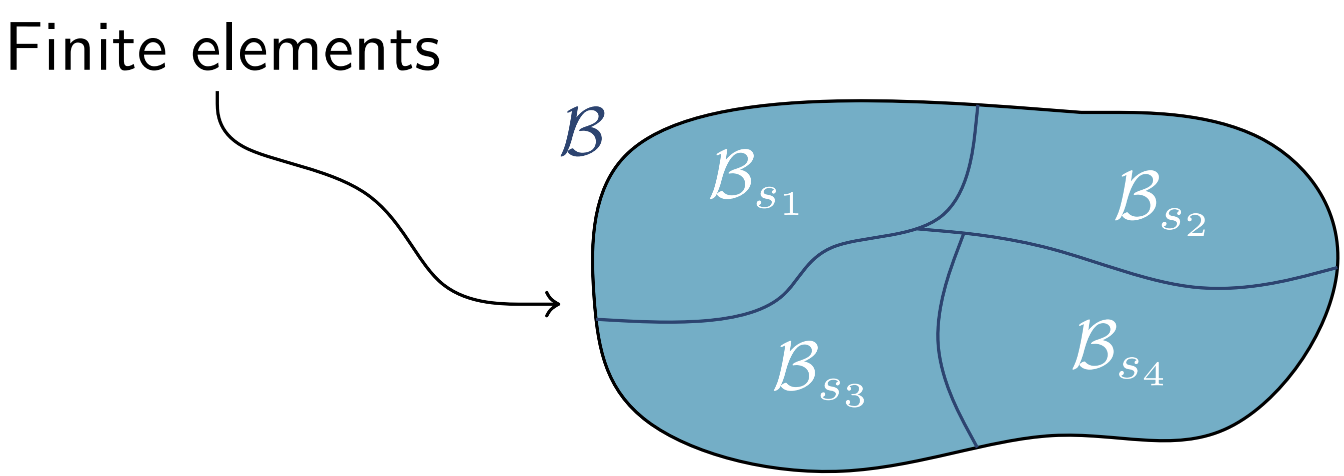
Finite elements



Decidability by Small Model Property (SMP)

- Let Γ be a conjunction of $\mathsf{TSL_K}$-literals
- If Γ is satisfied in an arbitrary $\mathsf{TSL_K}$ interpretation $\mathcal{A}$
- We construct a new finite interpretation $\mathcal{B}$ **bounded by Γ**

$\mathsf{TSL_K}$ is **decidable** by enumerating all possible elements



A Decision Procedure for $\mathsf{TSL}_{\mathsf{K}}$ (main idea)

A Decision Procedure for TSL_K (main idea)

- We use a **many-sorted variant of Nelson-Oppen**

A Decision Procedure for $\mathsf{TSL_K}$ (main idea)

- We use a **many-sorted variant of Nelson-Oppen**

To apply it, we require:

A Decision Procedure for TSL_K (main idea)

- We use a **many-sorted variant of Nelson-Oppen**

To apply it, we require:

- A **decision procedure** for each theory

A Decision Procedure for TSL_K (main idea)

- We use a **many-sorted variant of Nelson-Oppen**

To apply it, we require:

- A **decision procedure** for each theory

- Theories **share only sorts**

A Decision Procedure for TSL_K (main idea)

- We use a **many-sorted variant of Nelson-Oppen**

To apply it, we require:

- A **decision procedure** for each theory
- Theories **share only sorts**
- When two theories are combined:
  - Both are **stable infinite**
  - One is **polite** to the other with respect to shared sorts

Stable Infinite & Politeness

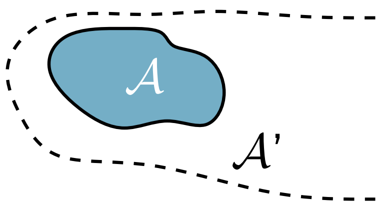
Stable Infinite & Politeness

- **Stable infinite**
- **Polite** with respect to sorts $s_1, \ldots s_n$

Stable Infinite & Politeness

- **Stable infinite**

  for all QF-formula φ, exists an infinite interpretation $\mathcal{A}$'

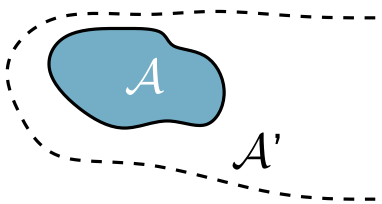


- **Polite** with respect to sorts $s_1, \ldots s_n$

Stable Infinite & Politeness

- **Stable infinite**

  for all QF-formula φ, exists an infinite interpretation $\mathcal{A}$'

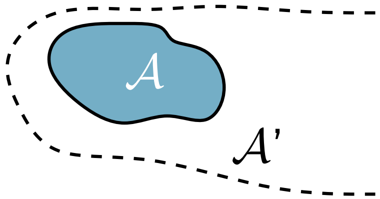


- **Polite** with respect to sorts $s_1, \ldots s_n$
  - **Smooth**
  - **Finite witnessable**

Stable Infinite & Politeness

- **Stable infinite**

  for all QF-formula φ, exists an infinite interpretation $\mathcal{A}'$



- **Polite** with respect to sorts $s_1, \ldots s_n$
  - **Smooth**

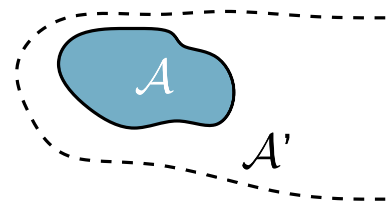
    if $\mathcal{A}$ is a model of φ, with domains $\mathcal{A}_{s_1} \cdots \mathcal{A}_{s_n}$

  - **Finite witnessable**

Stable Infinite & Politeness

- **Stable infinite**

  for all QF-formula φ, exists an infinite interpretation $\mathcal{A}'$



- **Polite** with respect to sorts $s_1, \ldots s_n$

  - **Smooth**

    if $\mathcal{A}$ is a model of φ, with domains $|\mathcal{A}_{s_1}| \cdots |\mathcal{A}_{s_n}|$

    then, for every $k_1 \cdots k_n$

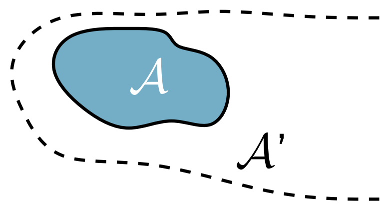
    $$\begin{array}{ccc} |\mathcal{A}_{s_1}| & \cdots & |\mathcal{A}_{s_n}| \\ \rotatebox{90}{$\geq$} & & \rotatebox{90}{$\geq$} \\ k_1 & \cdots & k_n \end{array}$$

  - **Finite witnessable**

Stable Infinite & Politeness

- **Stable infinite**

  for all QF-formula φ, exists an infinite interpretation $\mathcal{A}$'



- **Polite** with respect to sorts $s_1, \ldots s_n$

  - **Smooth**

    if $\mathcal{A}$ is a model of φ, with domains $\left|\mathcal{A}_{s_1}\right| \cdots \left|\mathcal{A}_{s_n}\right|$

    then, for every $k_1 \cdots k_n$

    there is a model $\mathcal{B}$ of φ, such that $\left|\mathcal{B}_{s_1}\right| \cdots \left|\mathcal{B}_{s_n}\right|$

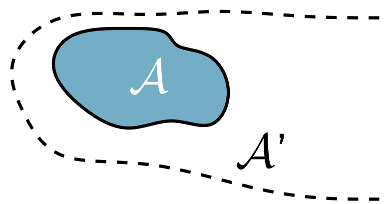
    $$\begin{array}{ccc} \left|\mathcal{A}_{s_1}\right| & \cdots & \left|\mathcal{A}_{s_n}\right| \\ \rotatebox{90}{$\geqslant$} & & \rotatebox{90}{$\geqslant$} \\ k_1 & \cdots & k_n \\ \| & & \| \\ \left|\mathcal{B}_{s_1}\right| & \cdots & \left|\mathcal{B}_{s_n}\right| \end{array}$$

  - **Finite witnessable**

Stable Infinite & Politeness

- **Stable infinite**

  for all QF-formula φ, exists an infinite interpretation $\mathcal{A}$'



- **Polite** with respect to sorts $s_1, \ldots s_n$
  - **Smooth**

    if $\mathcal{A}$ is a model of φ, with domains $\left|\mathcal{A}_{s_1}\right| \cdots \left|\mathcal{A}_{s_n}\right|$

    then, for every $k_1 \cdots k_n$

    there is a model $\mathcal{B}$ of φ, such that $\left|\mathcal{B}_{s_1}\right| \cdots \left|\mathcal{B}_{s_n}\right|$

$$\begin{array}{ccc} \left|\mathcal{A}_{s_1}\right| & \cdots & \left|\mathcal{A}_{s_n}\right| \\ \rotatebox{-90}{$\leq$} & & \rotatebox{-90}{$\leq$} \\ k_1 & \cdots & k_n \\ \| & & \| \\ \left|\mathcal{B}_{s_1}\right| & \cdots & \left|\mathcal{B}_{s_n}\right| \end{array}$$

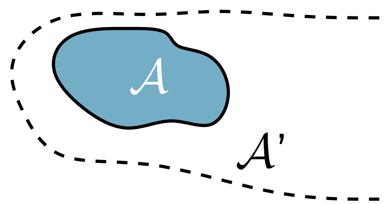
  - **Finite witnessable**

    f

Stable Infinite & Politeness

- **Stable infinite**

  for all QF-formula φ, exists an infinite interpretation $\mathcal{A}'$

- **Polite** with respect to sorts $s_1, \ldots s_n$

  - **Smooth**

    if $\mathcal{A}$ is a model of φ, with domains

    then, for every

    there is a model $\mathcal{B}$ of φ, such that

    $$\begin{array}{ccc} |\mathcal{A}_{s_1}| & \cdots & |\mathcal{A}_{s_n}| \\ \rotatebox{90}{$\not\geq$} & & \rotatebox{90}{$\not\geq$} \\ k_1 & \cdots & k_n \\ \| & & \| \\ |\mathcal{B}_{s_1}| & \cdots & |\mathcal{B}_{s_n}| \end{array}$$

  - **Finite witnessable**

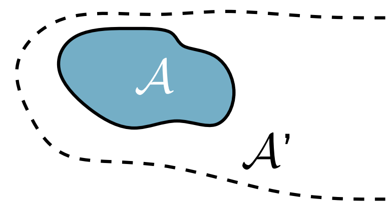
    f

    φ

Stable Infinite & Politeness

- **Stable infinite**

  for all QF-formula φ, exists an infinite interpretation $\mathcal{A}$'



- **Polite** with respect to sorts $s_1, \ldots s_n$
  - **Smooth**

    if $\mathcal{A}$ is a model of φ, with domains $\quad \left|\mathcal{A}_{s_1}\right| \cdots \left|\mathcal{A}_{s_n}\right|$

    $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rotatebox{90}{$\geq$} \qquad\quad \rotatebox{90}{$\geq$}$

    then, for every $\quad k_1 \cdots k_n$

    $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \| \qquad\quad \|$

    there is a model $\mathcal{B}$ of φ, such that $\quad \left|\mathcal{B}_{s_1}\right| \cdots \left|\mathcal{B}_{s_n}\right|$

  - **Finite witnessable**

$$\varphi \xrightarrow{\quad f \quad} (\exists \bar{v})\psi \quad \text{s.t. } [\bar{v} = V_\varphi \setminus V_\psi]$$

if ψ is **satisfiable**, then exists $\mathcal{B}$ with **one variable per value**

Checklist

- We use a **many-sorted variant of Nelson-Oppen**

To apply it, we require:

- A **decision procedure** for each theory
- Theories **share only sorts**
- When two theories are combined:
  - Both are **stable infinite**
  - One is **polite** to the other with respect to shared sorts

Checklist

- We use a **many-sorted variant of Nelson-Oppen**

To apply it, we require:

- A **decision procedure** for each theory ✓ (except $T_{\text{reachability}}$ and Σ_{bridge})
- Theories **share only sorts**
- When two theories are combined:
  - Both are **stable infinite**
  - One is **polite** to the other with respect to shared sorts

Checklist

- We use a **many-sorted variant of Nelson-Oppen**

To apply it, we require:

- A **decision procedure** for each theory ✓ (except $T_{\text{reachability}}$ and Σ_{bridge})
- Theories **share only sorts** ✓
- When two theories are combined:
  - Both are **stable infinite**
  - One is **polite** to the other with respect to shared sorts

A Combination-based Decision Procedure for $\mathsf{TSL_K}$

A Combination-based Decision Procedure for TSL_K

$$\mathsf{TSL}_\mathsf{K} = \cdots \cup T_{\mathsf{reachability}} \cup \cdots \cup \textit{bridge functions and predicates}$$

A Combination-based Decision Procedure for $\mathsf{TSL_K}$

$$\mathsf{TSL_K} = \cdots \cup T_{\mathsf{reachability}} \cup \cdots \cup \textit{bridge functions and predicates}$$

$$T_{\mathsf{SLKBase}} = T_{\mathsf{addr}} \cup T_{\mathsf{ord}} \cup T_{\mathsf{thid}} \cup T_{\mathsf{level_K}} \cup T_{\mathsf{cell}} \cup T_{\mathsf{mem}} \cup T_{\mathsf{fseq}} \cup T_{\mathsf{set}} \cup T_{\mathsf{setth}} \cup T_{\mathsf{mrgn}}$$

A Combination-based Decision Procedure for $\mathsf{TSL_K}$

$$\mathsf{TSL_K} = \cdots \cup T_{\mathsf{reachability}} \cup \cdots \cup \textit{bridge functions and predicates}$$

$$T_{\mathsf{SLKBase}} = T_{\mathsf{addr}} \cup T_{\mathsf{ord}} \cup T_{\mathsf{thid}} \cup T_{\mathsf{level_K}} \cup T_{\mathsf{cell}} \cup T_{\mathsf{mem}} \cup \overbrace{T_{\mathsf{fseq}}}^{T_{\mathsf{path}} \xleftarrow{PATH}} \cup T_{\mathsf{set}} \cup T_{\mathsf{setth}} \cup T_{\mathsf{mrgn}}$$

A Combination-based Decision Procedure for $\mathsf{TSL_K}$

$$\mathsf{TSL_K} = \cdots \cup T_{\mathsf{reachability}} \cup \cdots \cup \textit{bridge functions and predicates}$$

$$\widehat{\mathsf{TSL_K}}$$

GAP

T_{path}

$PATH$

$$T_{\mathsf{SLKBase}} = T_{\mathsf{addr}} \cup T_{\mathsf{ord}} \cup T_{\mathsf{thid}} \cup T_{\mathsf{level_K}} \cup T_{\mathsf{cell}} \cup T_{\mathsf{mem}} \cup T_{\mathsf{fseq}} \cup T_{\mathsf{set}} \cup T_{\mathsf{setth}} \cup T_{\mathsf{mrgn}}$$

A Combination-based Decision Procedure for $\mathsf{TSL_K}$

$$\mathsf{TSL_K} = \cdots \cup T_{\mathsf{reachability}} \cup \cdots \cup \textit{bridge functions and predicates}$$

normalized $\mathsf{TSL_K}$-literals

$$\widehat{\mathsf{TSL_K}}$$

$$GAP$$

$$T_{\mathsf{path}}$$

$$PATH$$

$$T_{\mathsf{SLKBase}} = T_{\mathsf{addr}} \cup T_{\mathsf{ord}} \cup T_{\mathsf{thid}} \cup T_{\mathsf{level_K}} \cup T_{\mathsf{cell}} \cup T_{\mathsf{mem}} \cup T_{\mathsf{fseq}} \cup T_{\mathsf{set}} \cup T_{\mathsf{setth}} \cup T_{\mathsf{mrgn}}$$

A Combination-based Decision Procedure for $\mathsf{TSL_K}$

$$\mathsf{TSL_K} = \cdots \cup T_{\mathsf{reachability}} \cup \cdots \cup \textit{bridge functions and predicates}$$

normalized $\mathsf{TSL_K}$-literals

$$\widehat{\mathsf{TSL_K}}$$

GAP

$$T_{\mathsf{path}}$$

$PATH$

$$T_{\mathsf{SLKBase}} = T_{\mathsf{addr}} \cup T_{\mathsf{ord}} \cup T_{\mathsf{thid}} \cup T_{\mathsf{level_K}} \cup T_{\mathsf{cell}} \cup T_{\mathsf{mem}} \cup T_{\mathsf{fseq}} \cup T_{\mathsf{set}} \cup T_{\mathsf{setth}} \cup T_{\mathsf{mrgn}}$$

- **Unfolding** of definitions in $PATH$ and GAP

A Combination-based Decision Procedure for $\mathsf{TSL_K}$

$$\mathsf{TSL_K} = \cdots \cup T_{\mathsf{reachability}} \cup \cdots \cup \textit{bridge functions and predicates}$$

normalized $\mathsf{TSL_K}$-literals

$$\widehat{\mathsf{TSL_K}}$$

GAP

T_{path}

$PATH$

$$T_{\mathsf{SLKBase}} = {\color{red}T_{\mathsf{addr}}} \cup {\color{red}T_{\mathsf{ord}}} \cup {\color{red}T_{\mathsf{thid}}} \cup {\color{red}T_{\mathsf{level_K}}} \cup T_{\mathsf{cell}} \cup T_{\mathsf{mem}} \cup \overbrace{T_{\mathsf{fseq}}} \cup T_{\mathsf{set}} \cup T_{\mathsf{setth}} \cup T_{\mathsf{mrgn}}$$

- **Unfolding** of definitions in $PATH$ and GAP
- Share sorts only, hence **trivial**

A Combination-based Decision Procedure for $\mathsf{TSL_K}$

$$\mathsf{TSL_K} = \cdots \cup T_{\mathsf{reachability}} \cup \cdots \cup \textit{bridge functions and predicates}$$

normalized $\mathsf{TSL_K}$-literals

$$\widehat{\mathsf{TSL_K}}$$

GAP

T_{path}

$PATH$

$$T_{\mathsf{SLKBase}} = T_{\mathsf{addr}} \cup T_{\mathsf{ord}} \cup T_{\mathsf{thid}} \cup T_{\mathsf{level_K}} \cup T_{\mathsf{cell}} \cup T_{\mathsf{mem}} \cup \overbrace{T_{\mathsf{fseq}}} \cup T_{\mathsf{set}} \cup T_{\mathsf{setth}} \cup T_{\mathsf{mrgn}}$$

- **Unfolding** of definitions in $PATH$ and GAP
- Share sorts only, hence **trivial**
- Some are **stable infinite**

A Combination-based Decision Procedure for $\mathsf{TSL_K}$

$$\mathsf{TSL_K} = \cdots \cup T_{\mathsf{reachability}} \cup \cdots \cup \textit{bridge functions and predicates}$$

normalized $\mathsf{TSL_K}$-literals

$$\widehat{\mathsf{TSL_K}}$$

GAP

T_{path}

$PATH$

$$T_{\mathsf{SLKBase}} = T_{\mathsf{addr}} \cup T_{\mathsf{ord}} \cup T_{\mathsf{thid}} \cup T_{\mathsf{level_K}} \cup T_{\mathsf{cell}} \cup T_{\mathsf{mem}} \cup T_{\mathsf{fseq}} \cup T_{\mathsf{set}} \cup T_{\mathsf{setth}} \cup T_{\mathsf{mrgn}}$$

- **Unfolding** of definitions in $PATH$ and GAP
- Share sorts only, hence **trivial**
- Some are **stable infinite**
- *SMP* guarantees that all theories are **finite witnessable**

A Combination-based Decision Procedure for $\mathsf{TSL_K}$

$$\mathsf{TSL_K} = \cdots \cup T_{\mathsf{reachability}} \cup \cdots \cup \textit{bridge functions and predicates}$$

normalized $\mathsf{TSL_K}$-literals

$$\widehat{\mathsf{TSL_K}}$$

GAP

T_{path}

$PATH$

$$T_{\mathsf{SLKBase}} = T_{\mathsf{addr}} \cup T_{\mathsf{ord}} \cup T_{\mathsf{thid}} \cup T_{\mathsf{level_K}} \cup T_{\mathsf{cell}} \cup T_{\mathsf{mem}} \cup T_{\mathsf{fseq}} \cup T_{\mathsf{set}} \cup T_{\mathsf{setth}} \cup T_{\mathsf{mrgn}}$$

- **Unfolding** of definitions in $PATH$ and GAP
- Share sorts only, hence **trivial**
- Some are **stable infinite**
- *SMP* guarantees that all theories are **finite witnessable**
- Smooth w.r.t. addr, $\mathsf{level_K}$, elem, ord and thid and hence **polite**

Conclusions

- We defined **TSL_K**, a theory for concurrent skiplists
- We proved TSL_K **decidable**, by *Small Model Property*
- We provide a **combination-based decision procedure** for TSL_K
- **TSL_K** can reason about memory, cells, pointers, masked regions, reachability, ordered lists and sublists
- A step towards the assisted verification of **temporal properties** over **concurrent data-types**: VD + DP
- **Current and future** work:
  parametrized verification diagrams, DP for concurrent structures, verification of current implementations, unbounded levels
- Many possible **collaborations**:
  DPs as combination, SMTs, implementation