Formal Verification of Skiplists with Arbitrarily Many Levels

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

[1]IMDEA Software Institute, Spain

[2]Institute for Information Security, CSIC, Spain

ATVA'14, Sydney, 5 November 2014

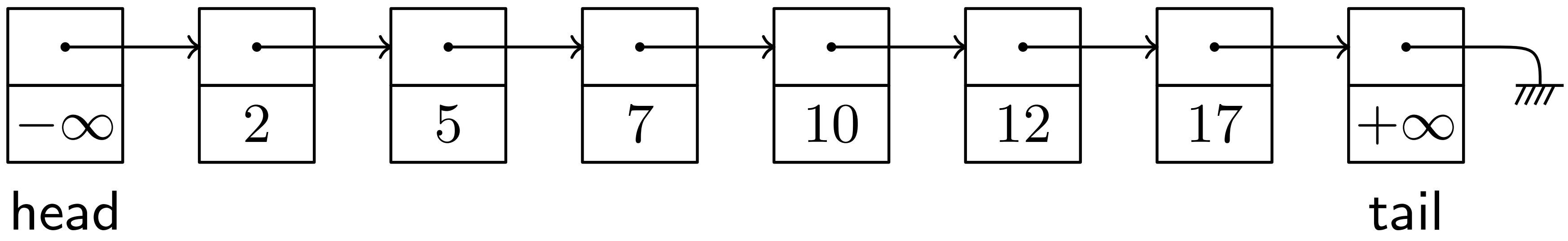
Skiplists

Skiplists

- Sorted list of elements

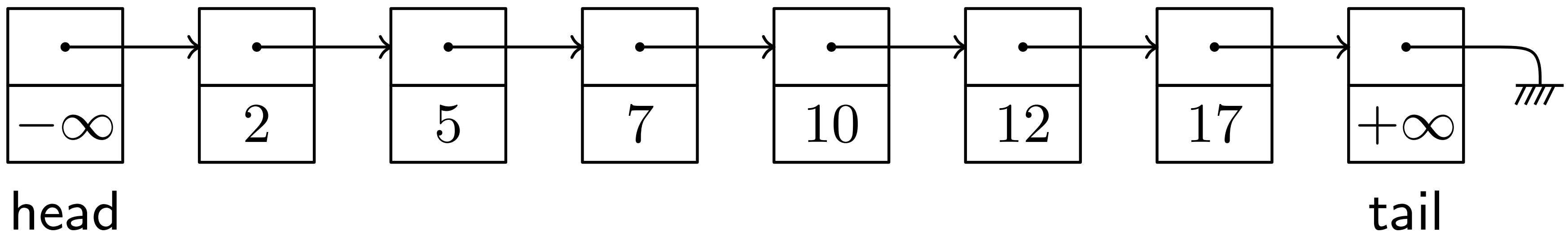
Skiplists

- Sorted list of elements



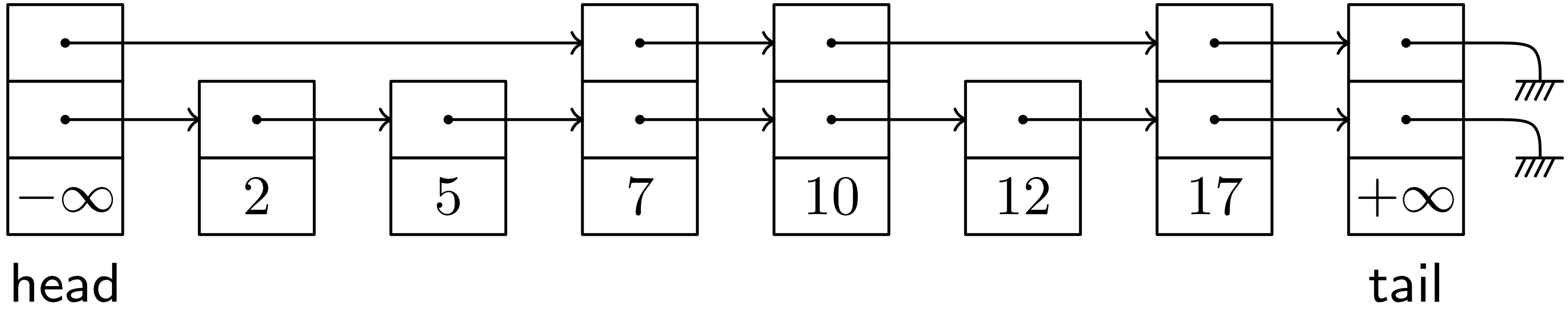
Skiplists

- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes



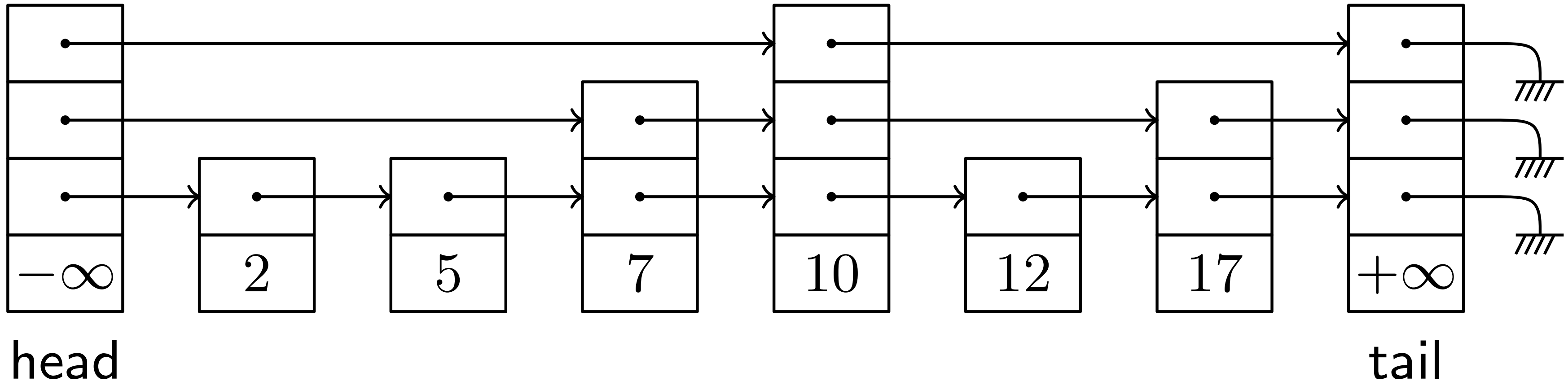
Skiplists

- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes



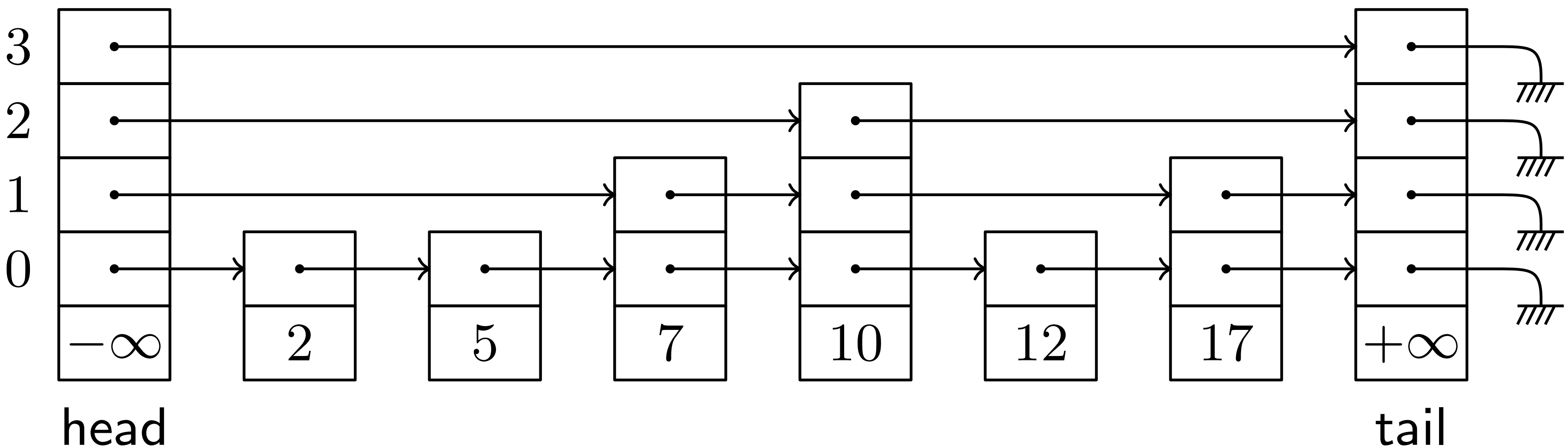
Skiplists

- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes



Skiplists

- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes

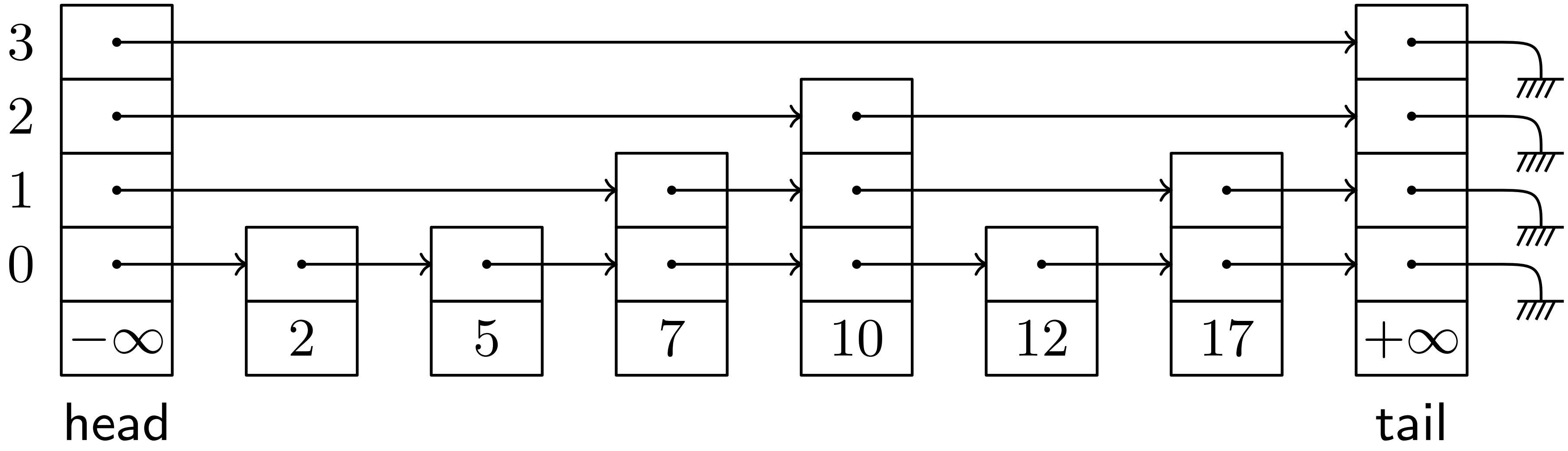


Skiplists

- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes

```
class Skiplist {            class Node {
        Node* head;                 Value v;
        Node* tail;                 Array<Node*>(4) next;
        int maxLevel;       }

}
```

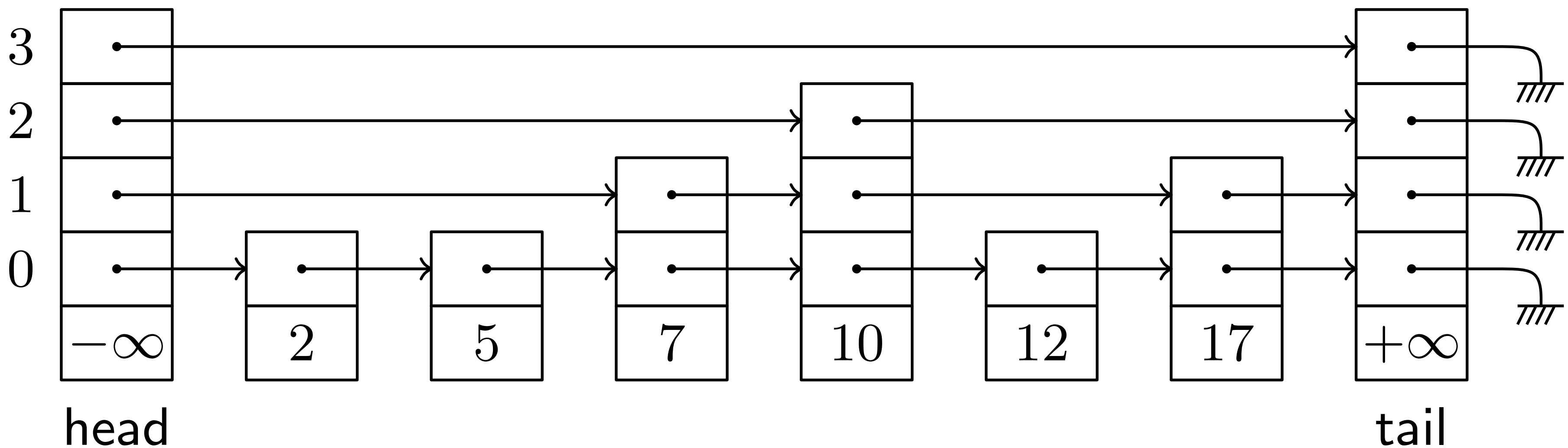


Skiplists

- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {             class Node {
        Node* head;                  Value v;
        Node* tail;                  Array<Node*>(4) next;
        int maxLevel;        }

}
```



Skiplists

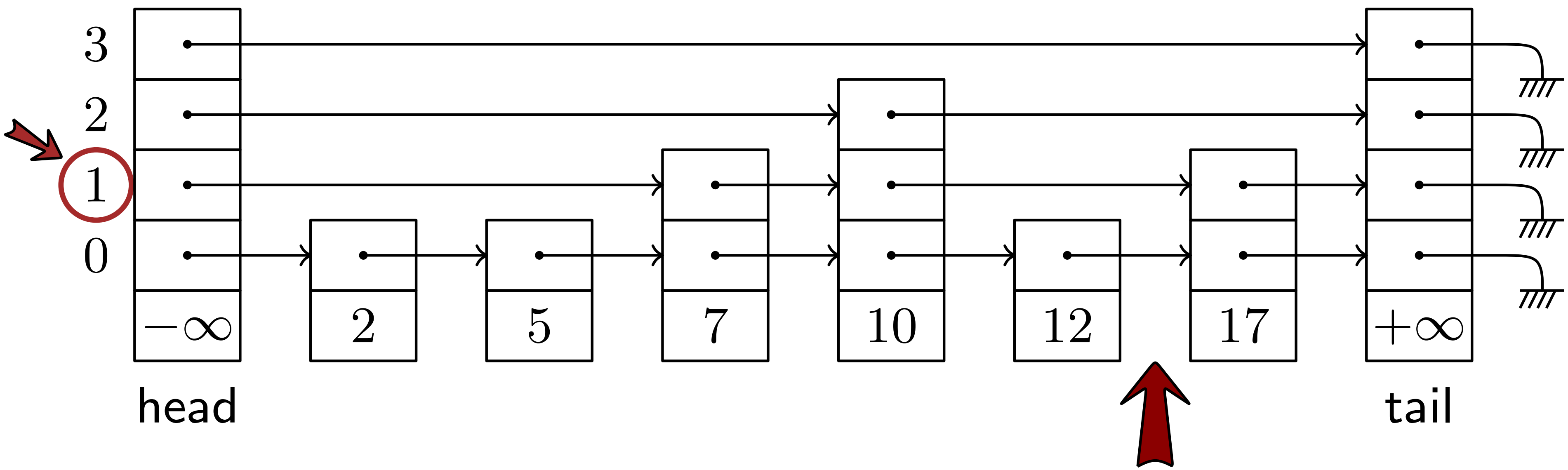
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

insert(14) with height 2



Skiplists

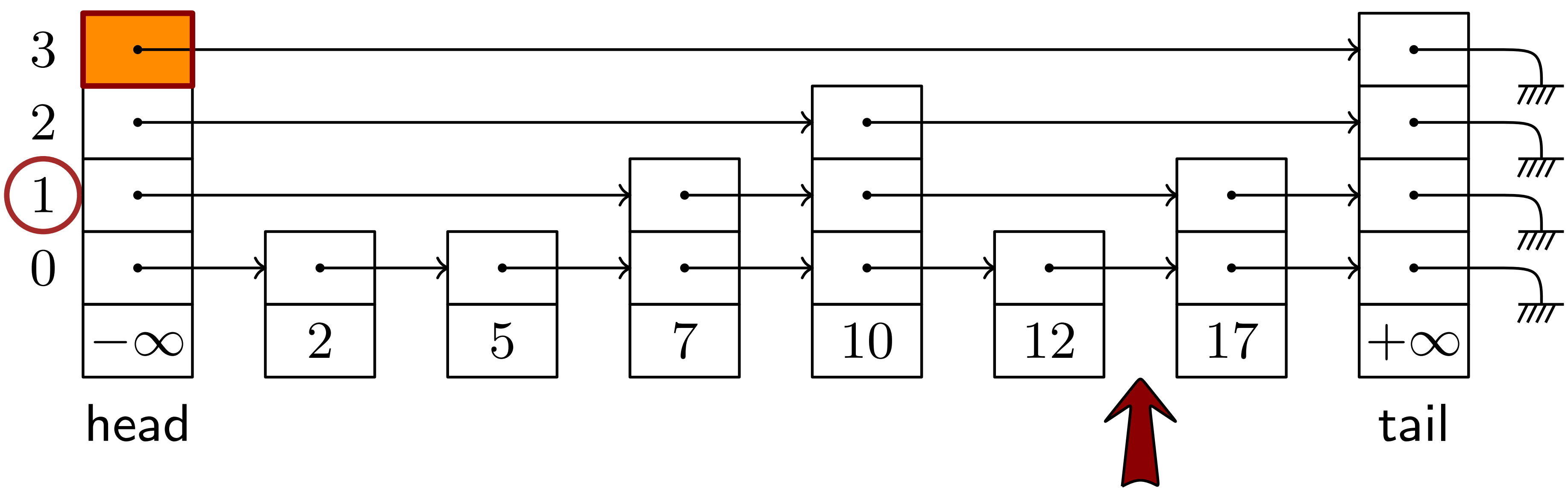
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

insert(14) with height 2



Skiplists

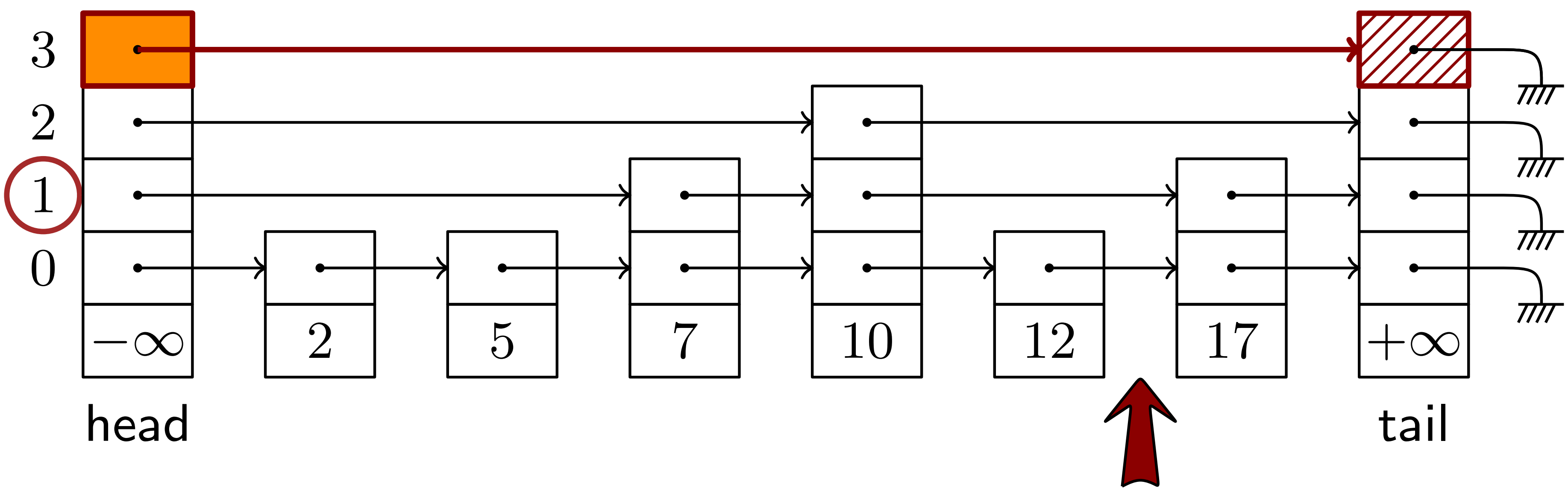
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

insert(14) with height 2



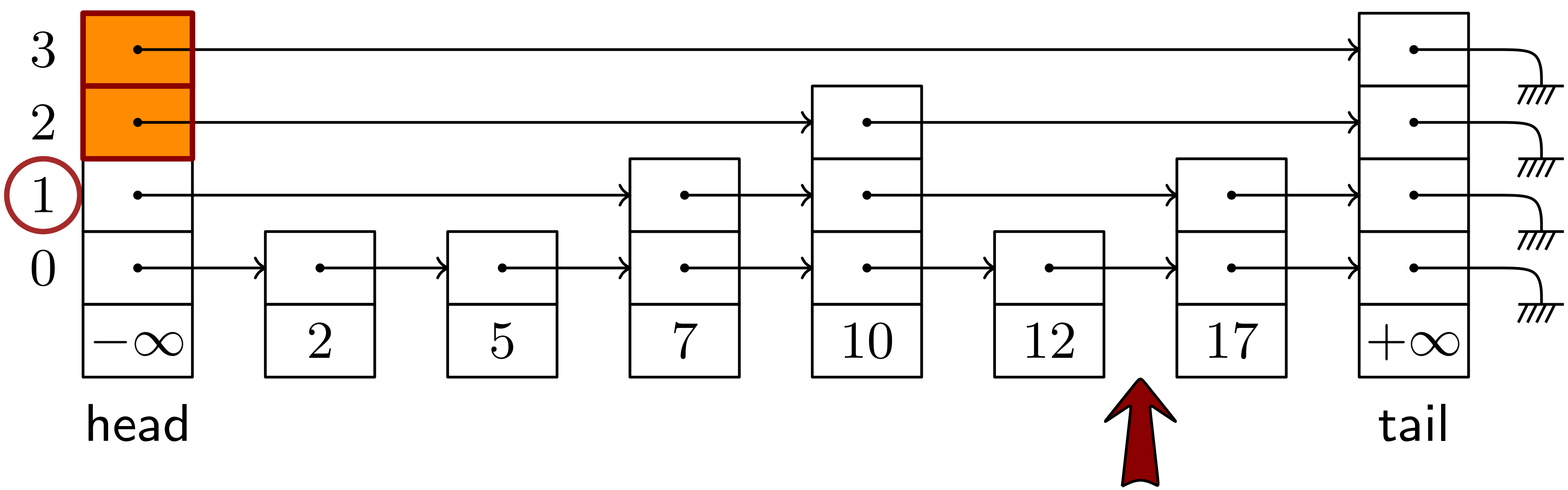
Skiplists

- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {                class Node {
        Node* head;                     Value v;
        Node* tail;                     Array<Node*>(4) next;
        int maxLevel;           }

}
```

insert(14) with height 2



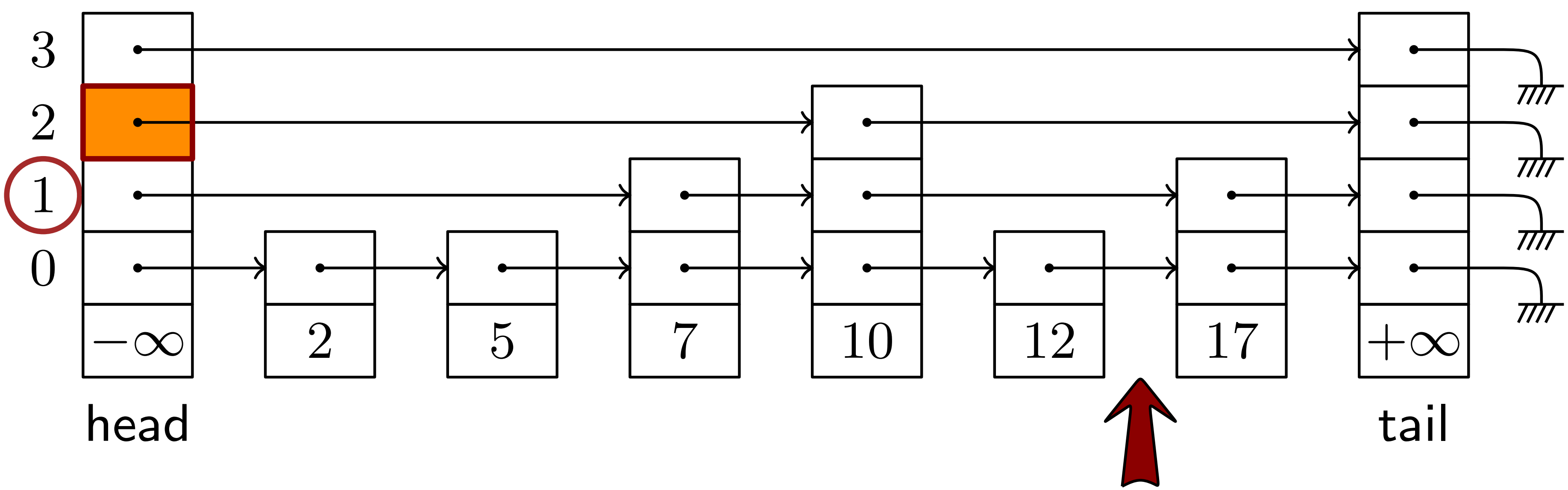
Skiplists

- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {          class Node {
        Node* head;               Value v;
        Node* tail;               Array<Node*>(4) next;
        int maxLevel;     }

}
```

insert(14) with height 2



Skiplists

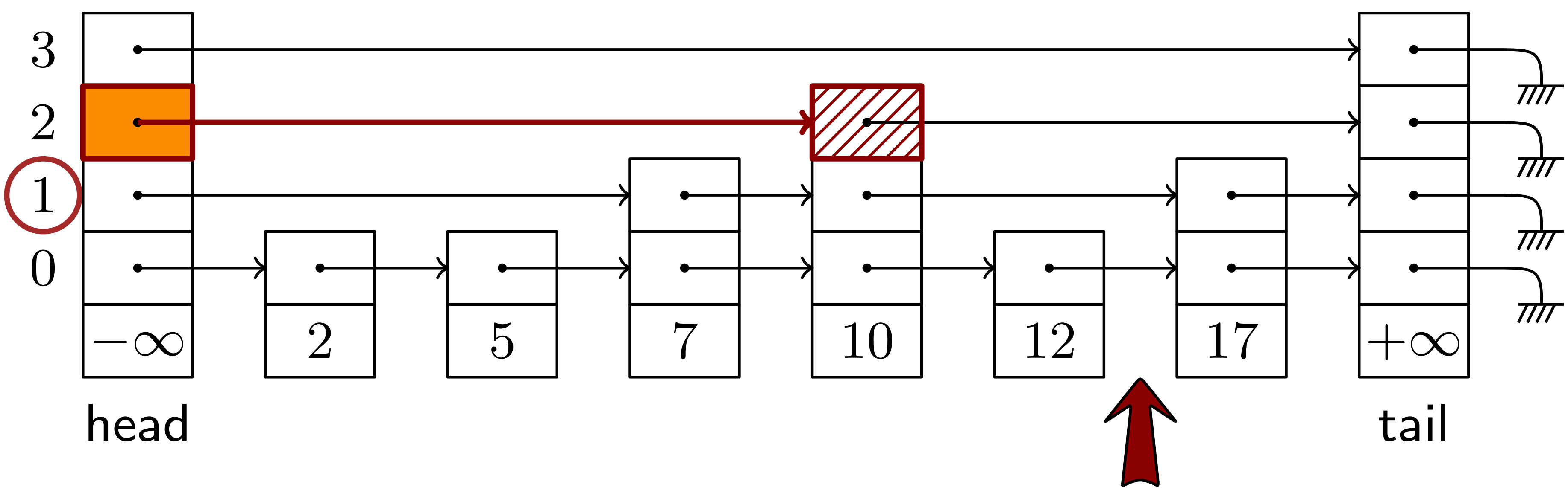
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

insert(14) with height 2



Skiplists

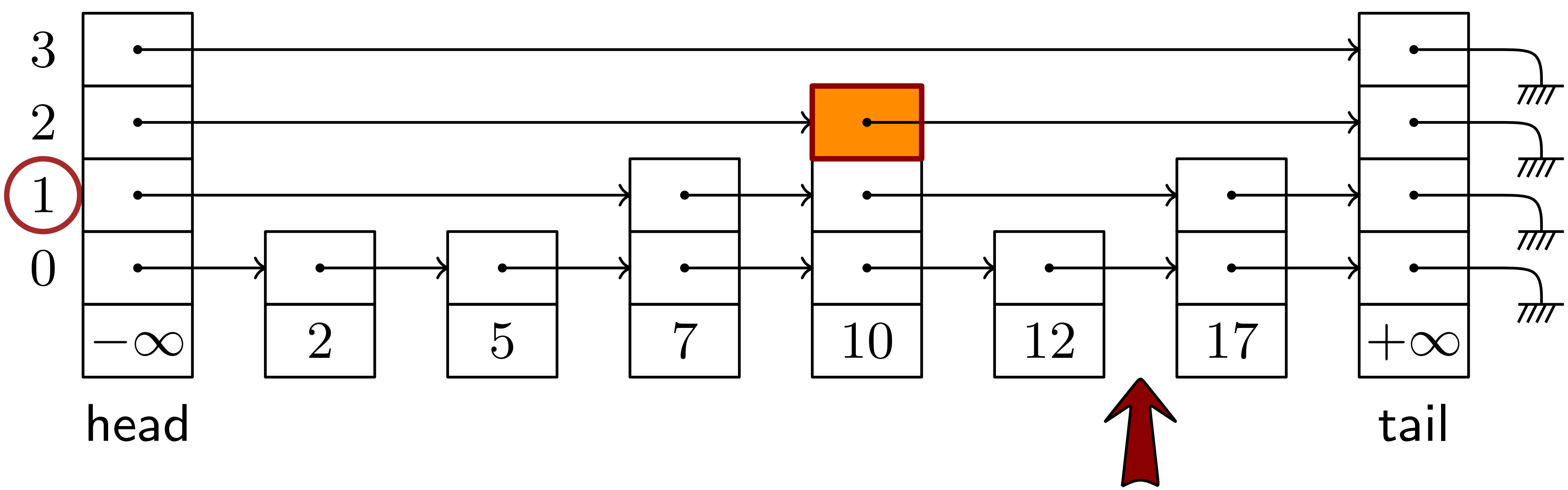
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

insert(14) with height 2



Skiplists

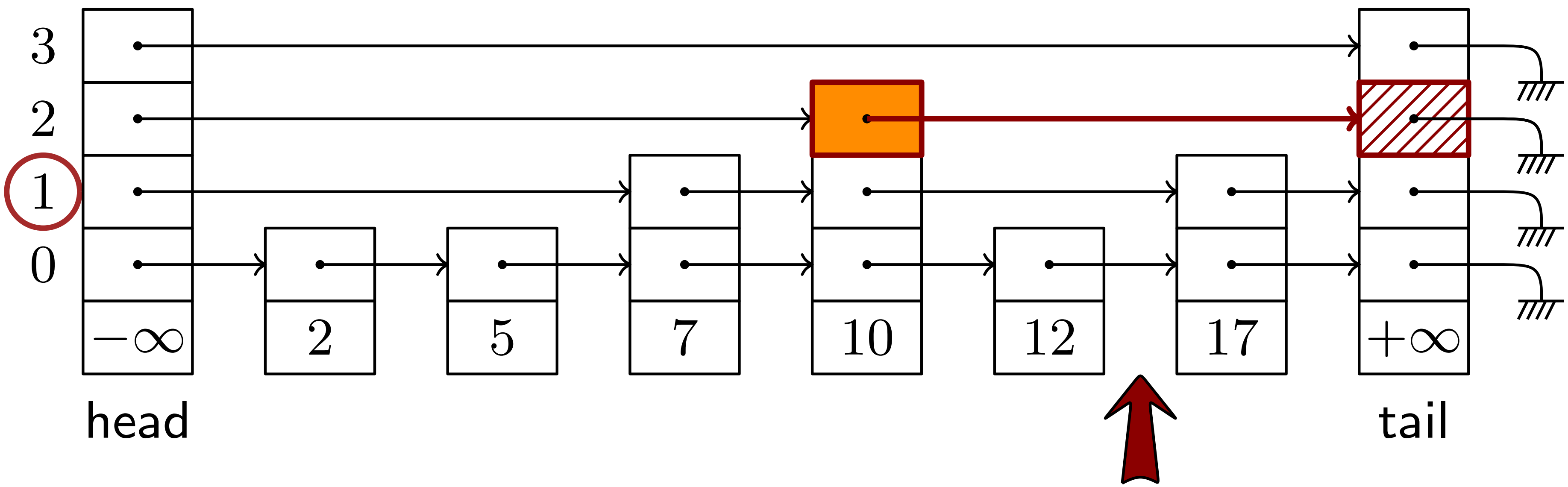
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

insert(14) with height 2



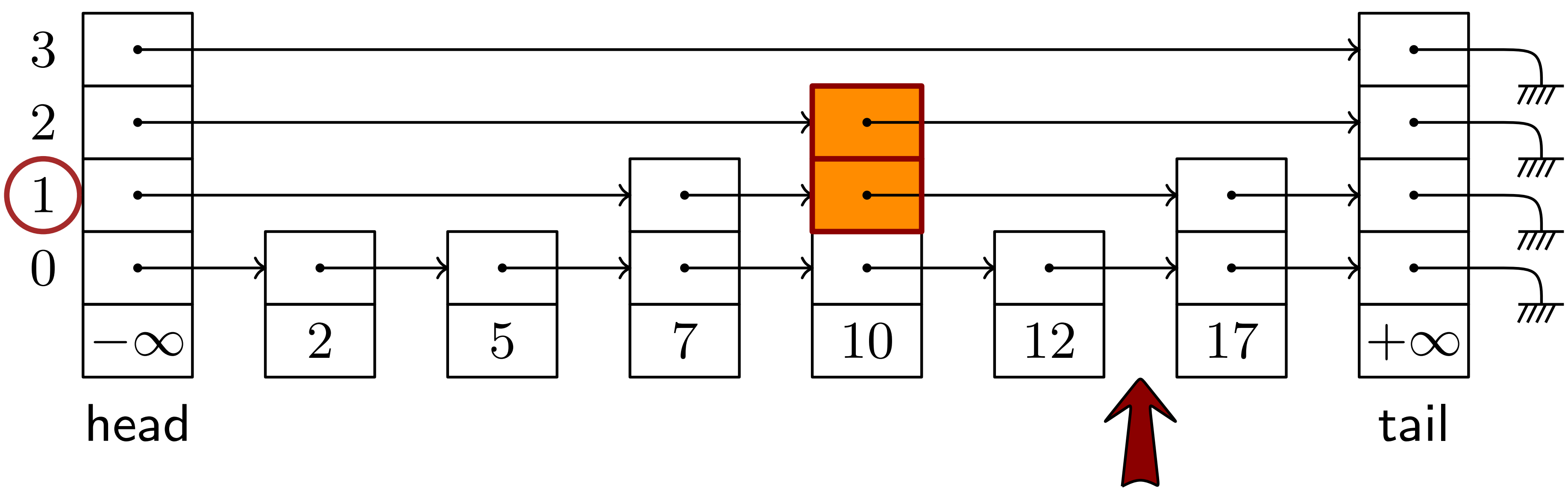
Skiplists

- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {            class Node {
        Node* head;                 Value v;
        Node* tail;                 Array<Node*>(4) next;
        int maxLevel;       }

}
```

insert(14) with height 2



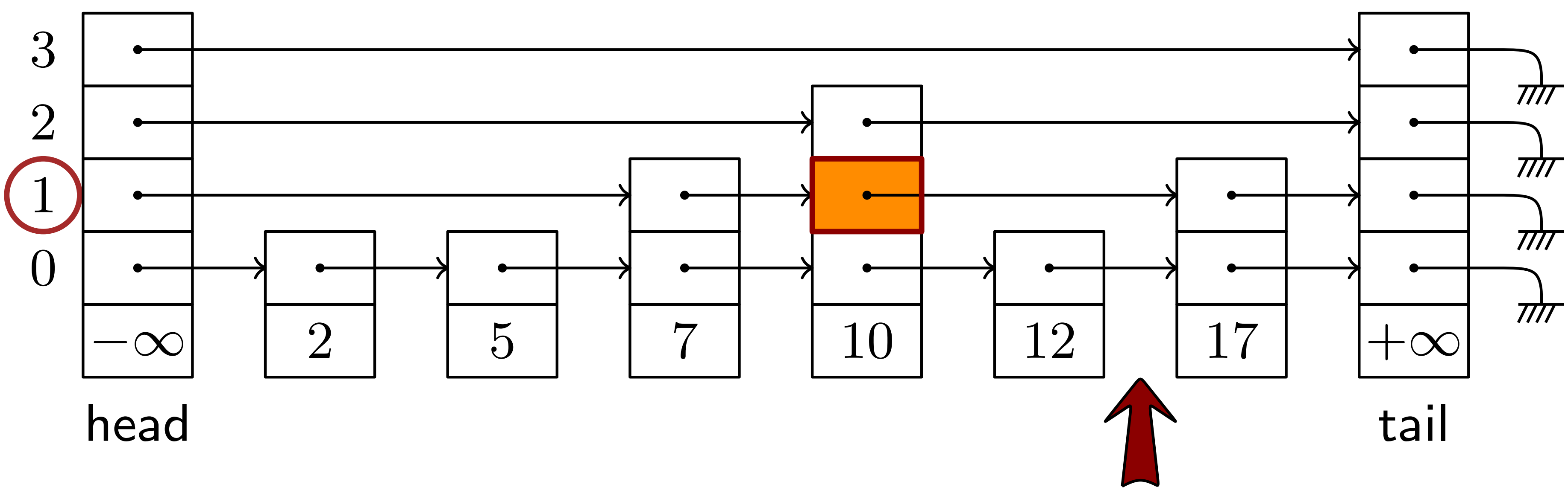
Skiplists

- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {              class Node {
        Node* head;                   Value v;
        Node* tail;                   Array<Node*>(4) next;
        int maxLevel;         }

}
```

insert(14) with height 2



Skiplists

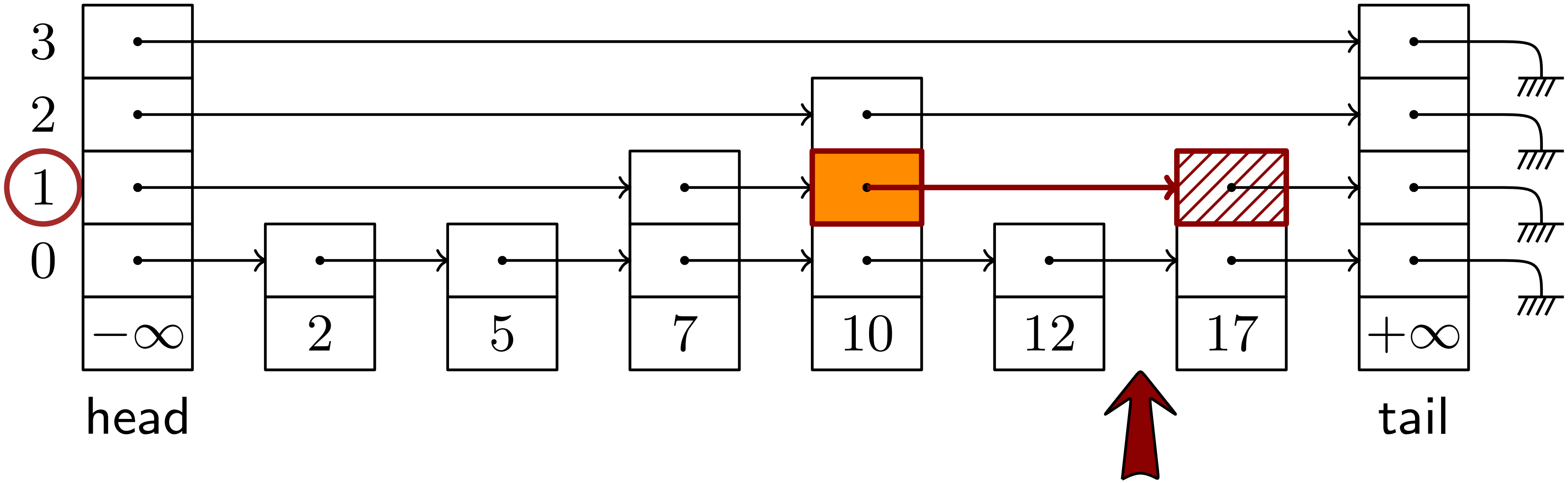
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

insert(14) with height 2



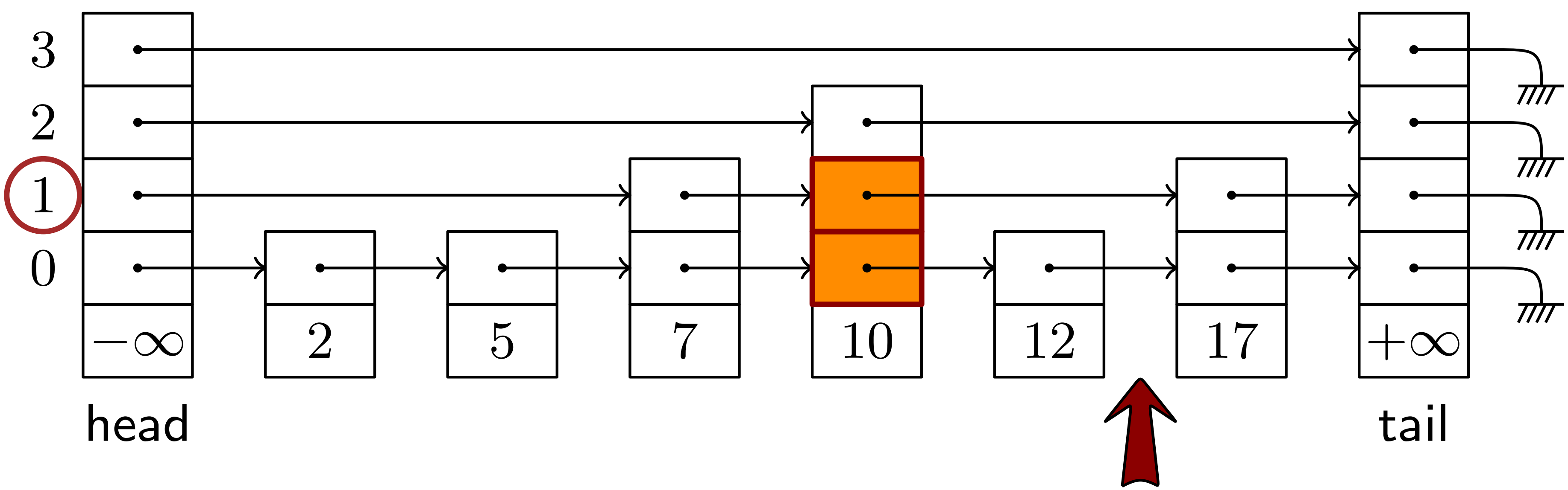
Skiplists

- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {                class Node {
        Node* head;                     Value v;
        Node* tail;                     Array<Node*>(4) next;
        int maxLevel;           }

}
```

insert(14) with height 2



Skiplists

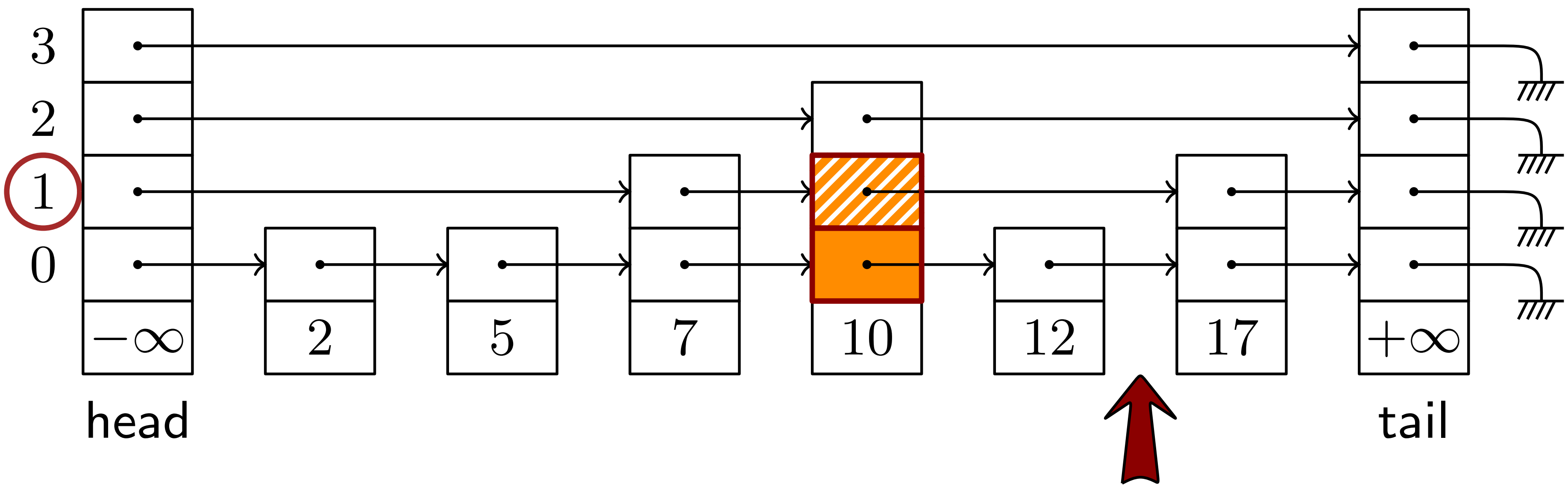
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

insert(14) with height 2



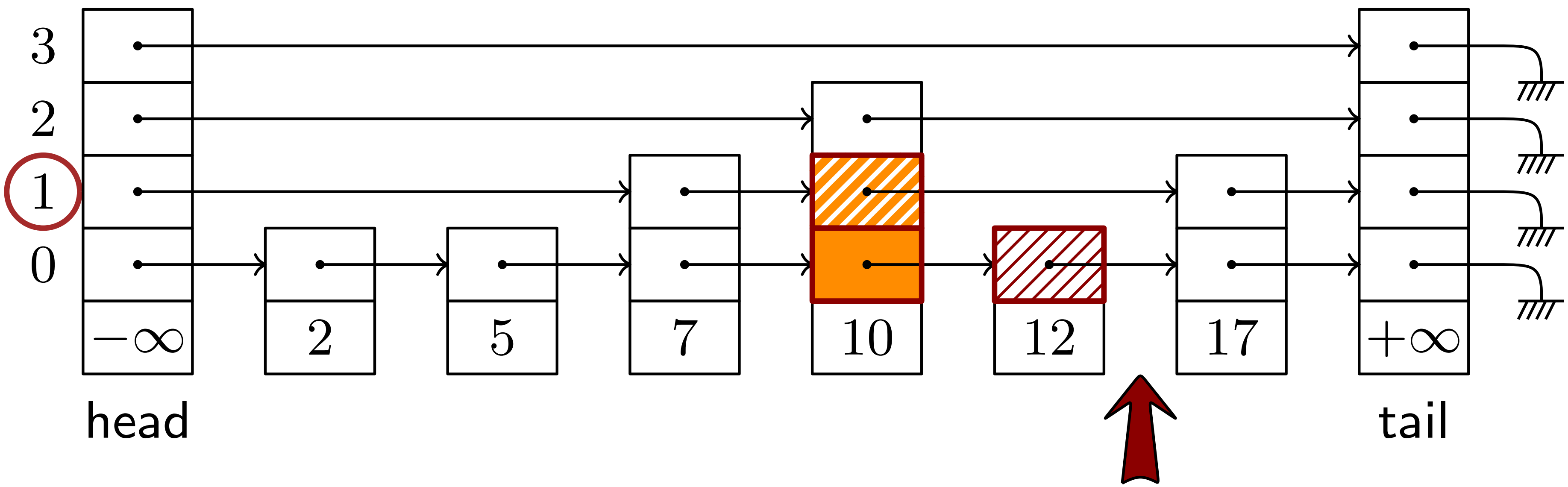
Skiplists

- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {            class Node {
        Node* head;                 Value v;
        Node* tail;                 Array<Node*>(4) next;
        int maxLevel;       }

}
```

insert(14) with height 2



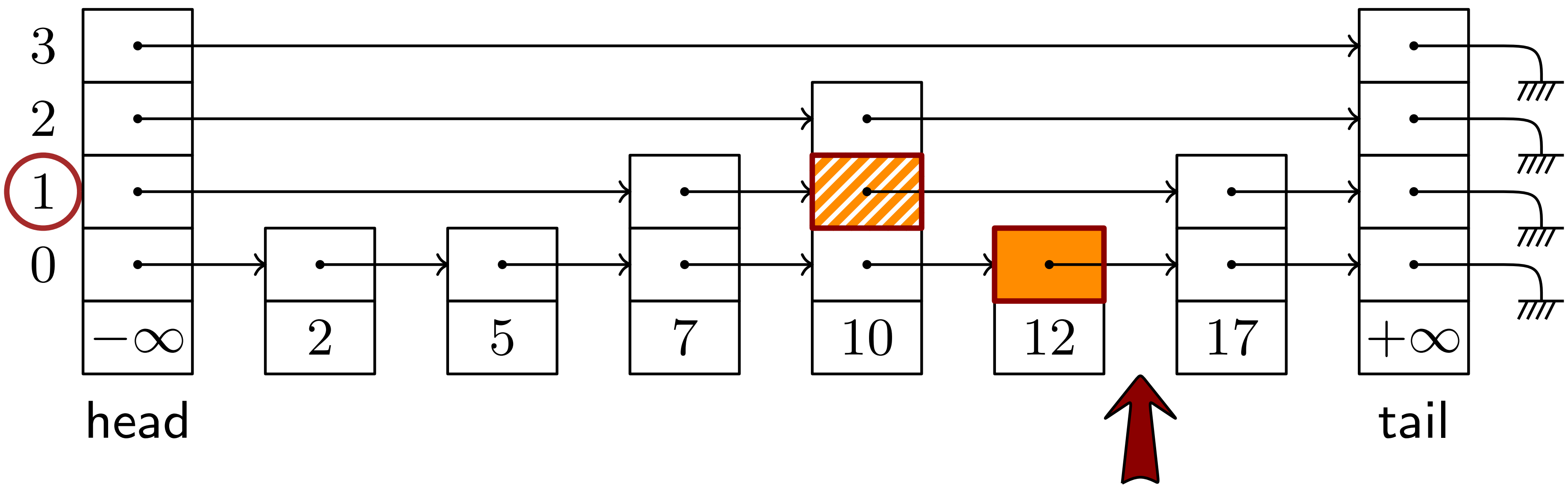
Skiplists

- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {              class Node {
        Node* head;                   Value v;
        Node* tail;                   Array<Node*>(4) next;
        int maxLevel;         }

}
```

insert(14) with height 2



Skiplists

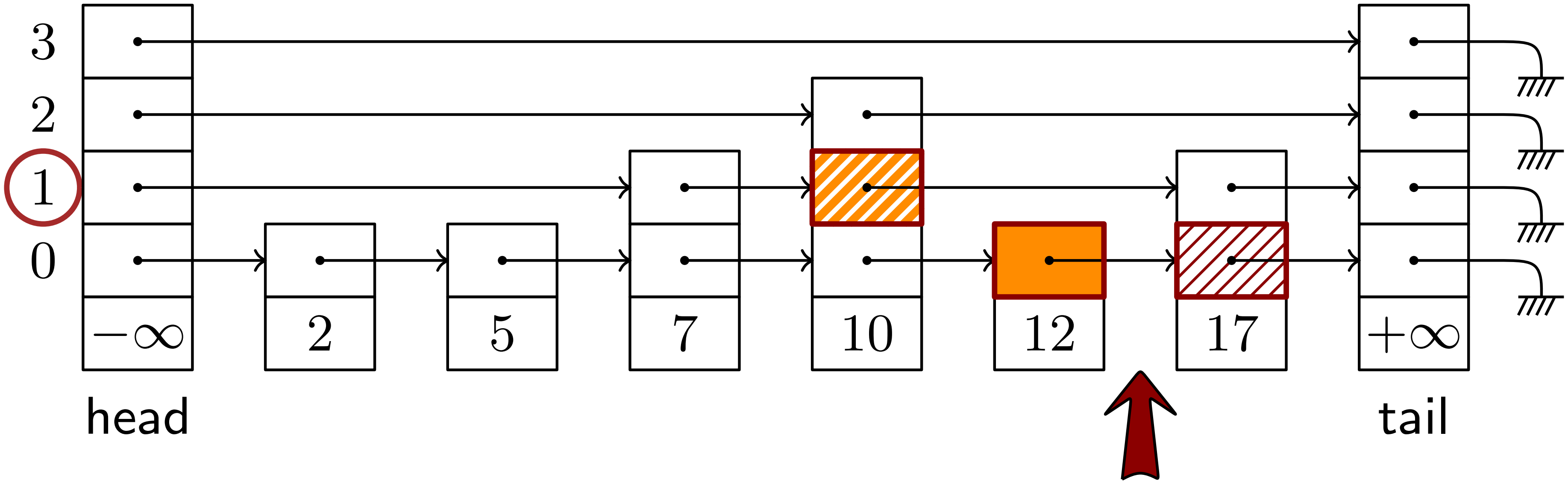
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

insert(14) with height 2



Skiplists

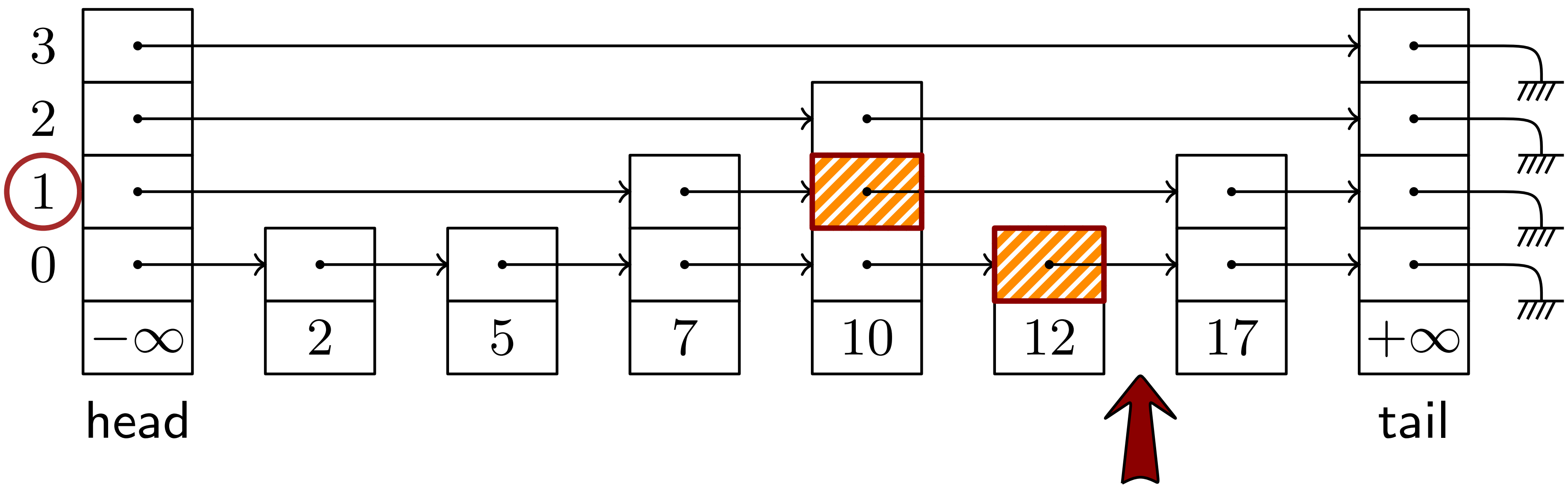
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

insert(14) with height 2



Skiplists

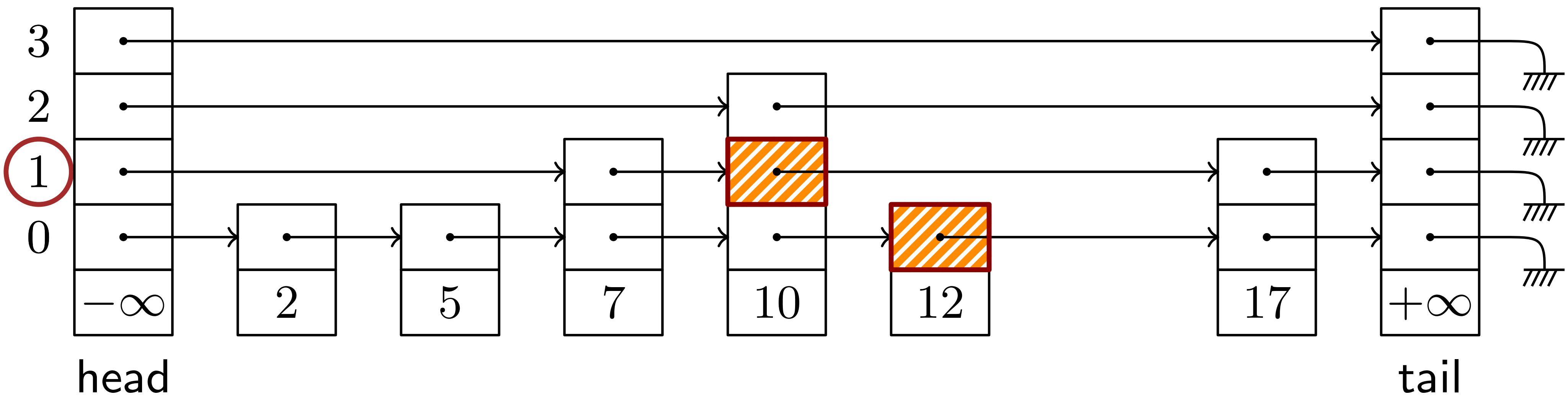
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

insert(14) with height 2



Skiplists

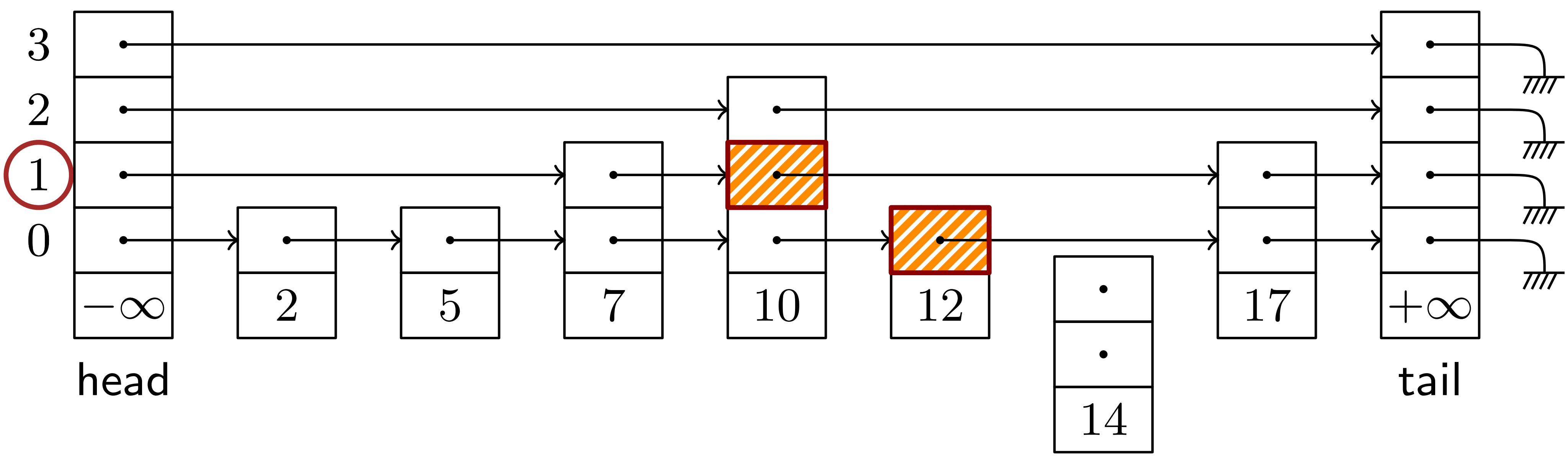
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

insert(14) with height 2



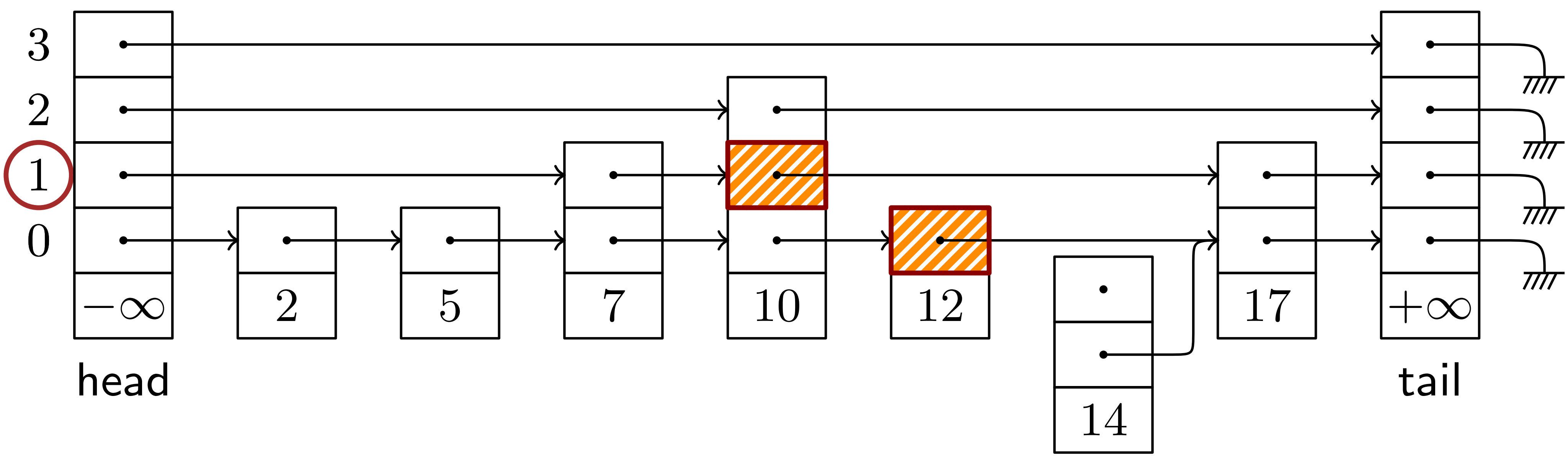
Skiplists

- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {            class Node {
        Node* head;                 Value v;
        Node* tail;                 Array<Node*>(4) next;
        int maxLevel;       }

}
```

insert(14) with height 2



Skiplists

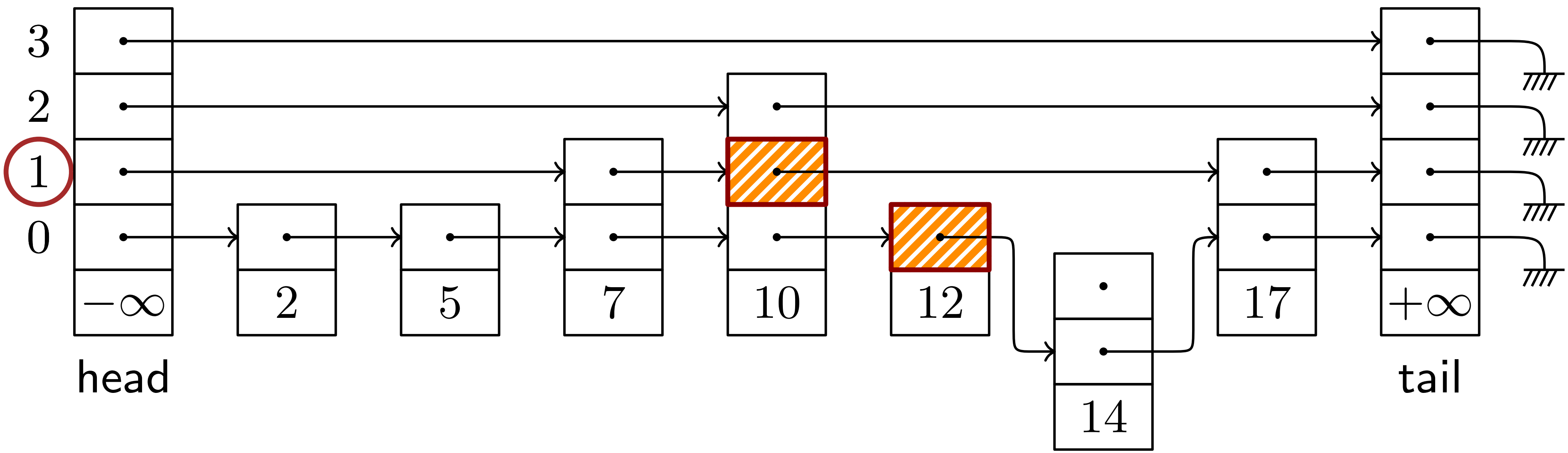
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

insert(14) with height 2



Skiplists

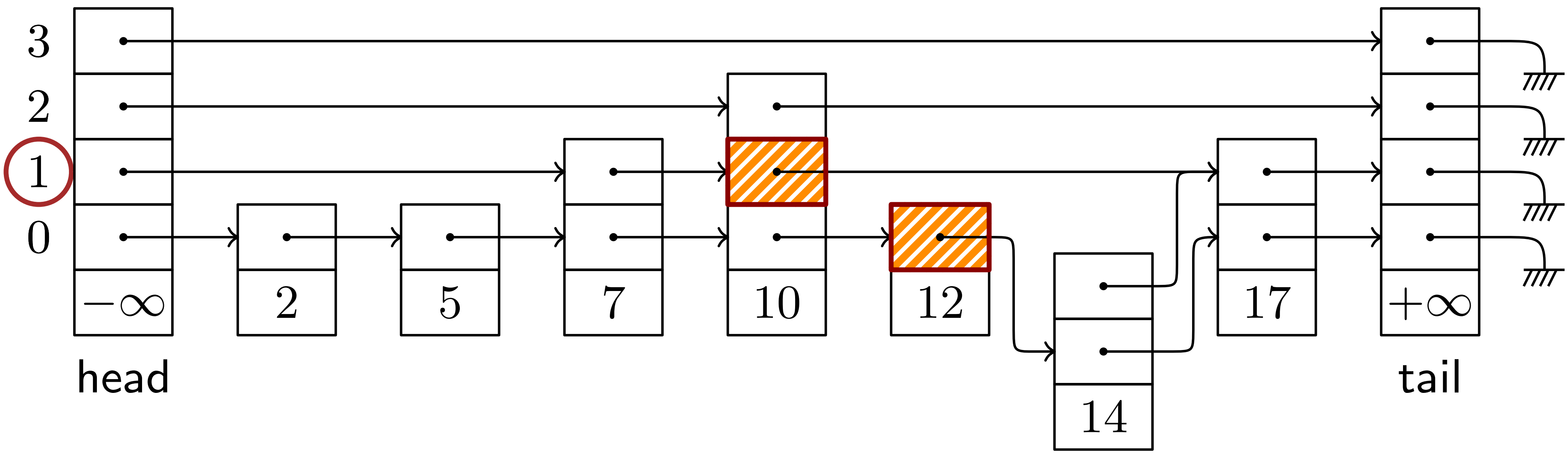
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

insert(14) with height 2



Skiplists

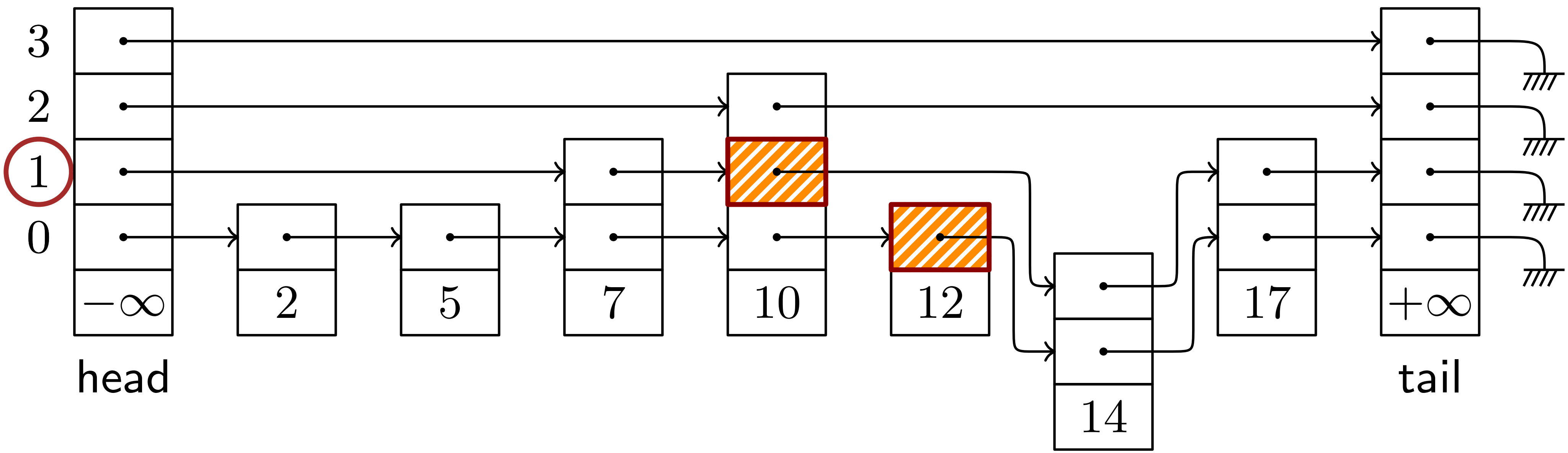
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

insert(14) with height 2



Skiplists

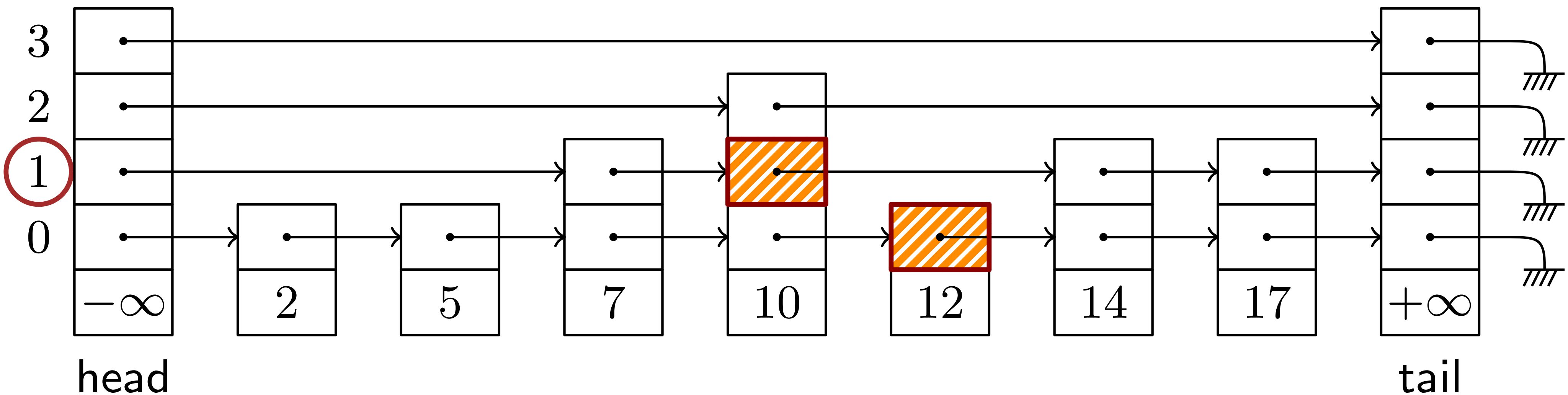
- Sorted list of elements
- Hierarchy of linked lists which **skip** nodes
- Efficiency comparable to balanced binary search trees

```
class Skiplist {
        Node* head;
        Node* tail;
        int maxLevel;

}
```

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

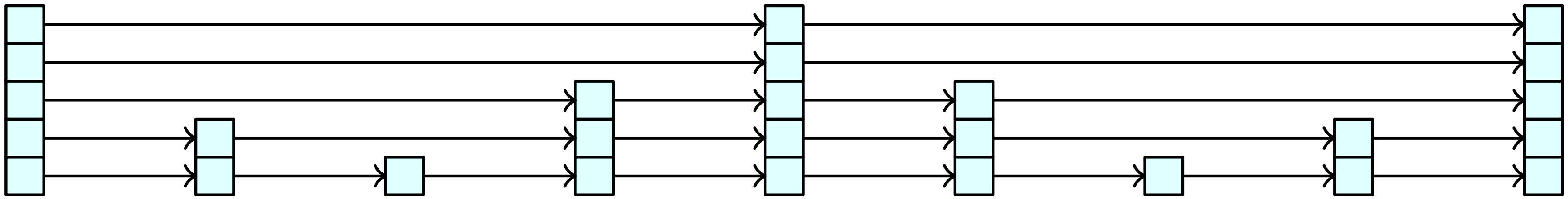
insert(14) with height 2



Why skiplists of unbounded height?

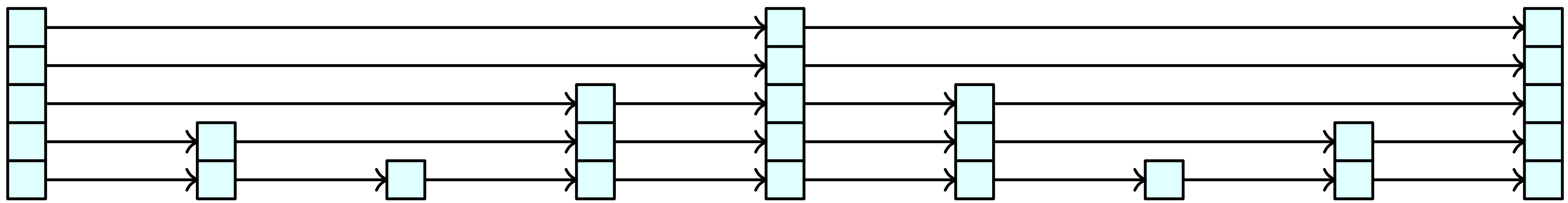
Why skiplists of unbounded height?

- We previously developed **TSL_K**



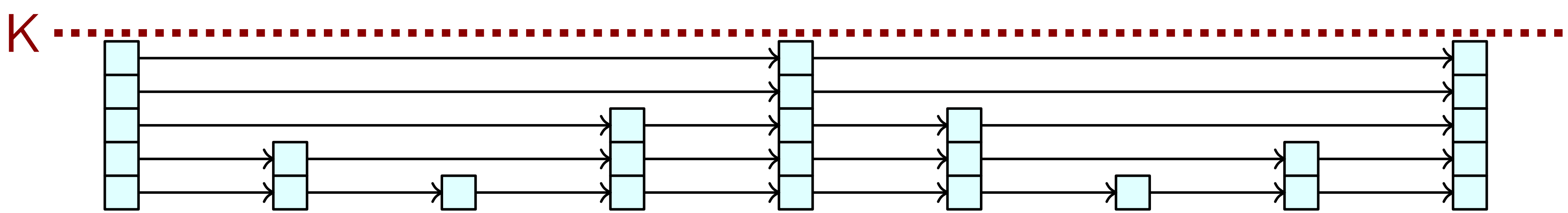
Why skiplists of unbounded height?

- We previously developed **TSL_K**
  - Show it decidable
  - Works for skiplists of arbitrary length...



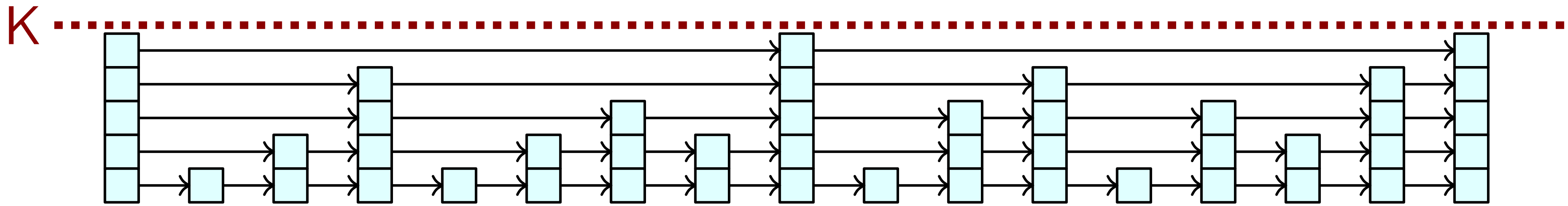
Why skiplists of unbounded height?

- We previously developed **TSL_K**
  - Show it decidable
  - Works for skiplists of arbitrary length...
  - ... but **bounded height**! (up to K levels)



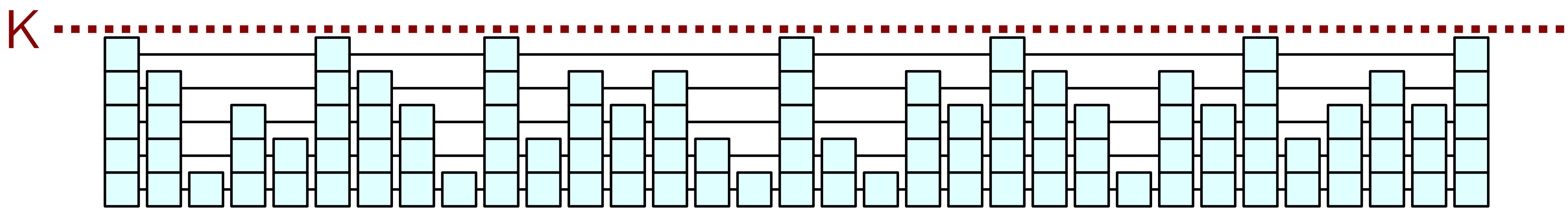
Why skiplists of unbounded height?

- We previously developed **TSL_K**
  - Show it decidable
  - Works for skiplists of arbitrary length...
  - ... but **bounded height**! (up to K levels)



Why skiplists of unbounded height?

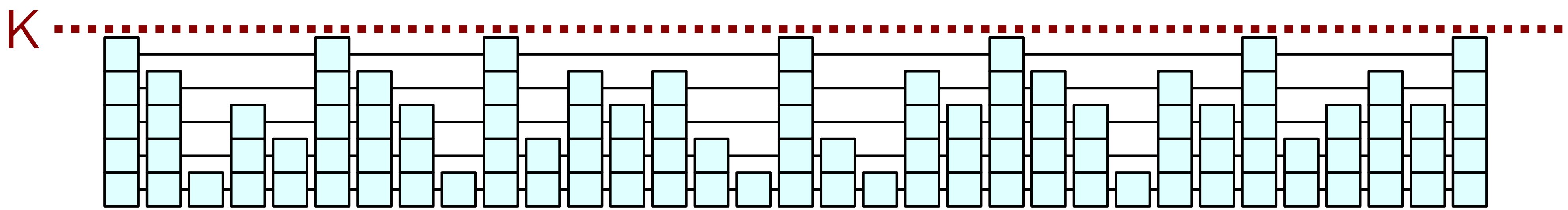
- We previously developed **TSL_K**
  - Show it decidable
  - Works for skiplists of arbitrary length...
  - ... but **bounded height**! (up to K levels)



Why skiplists of unbounded height?

- We previously developed **$\mathbf{TSL_K}$**
  - Show it decidable
  - Works for skiplists of arbitrary length...
  - ... but **bounded height**! (up to K levels)

In practice, **performance is lost!**



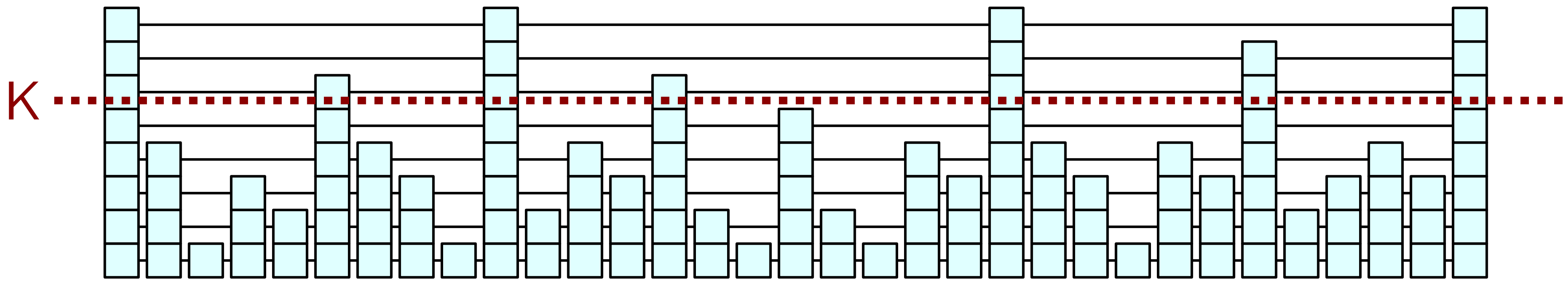
Why skiplists of unbounded height?

- We previously developed **TSL**~~$_K$~~
  - Show it decidable
  - Works for skiplists of arbitrary length...
  - ... but **bounded height**! (up to K levels)

TSL

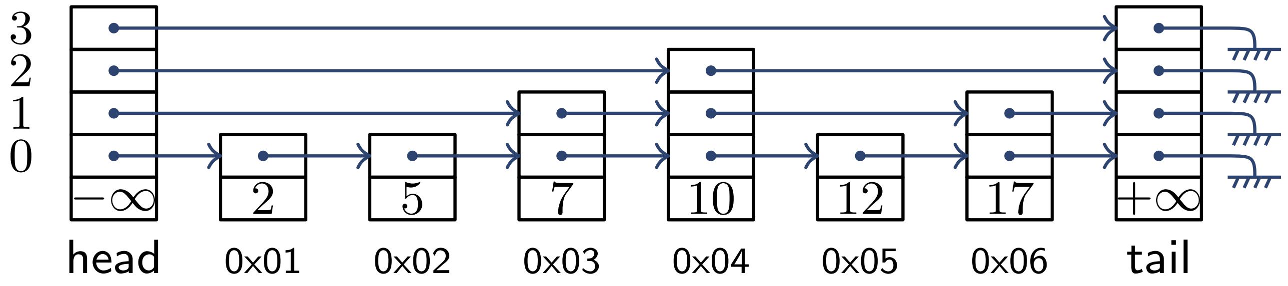
In practice, **performance is lost!**

Dynamic height is required



Verification of Skiplists

$M = 3$



Verification of Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList(heap, sl, r)$

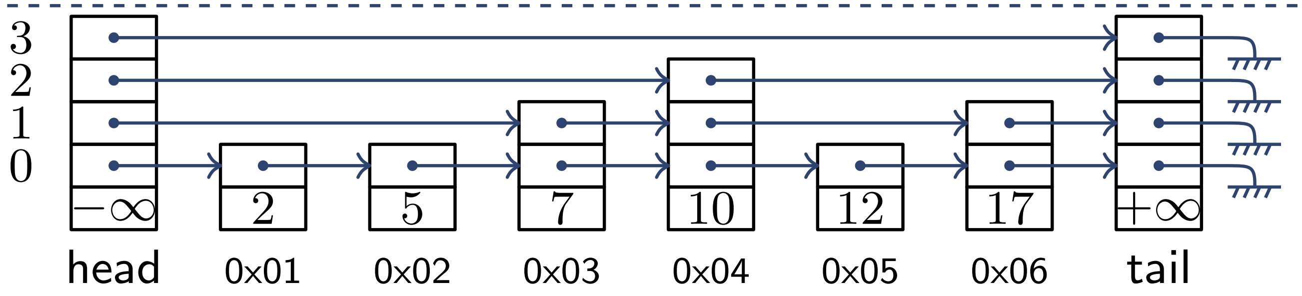
skiplist

$$SkipList(heap, sl, r) \hat{=}$$

heap

region

$M = 3$

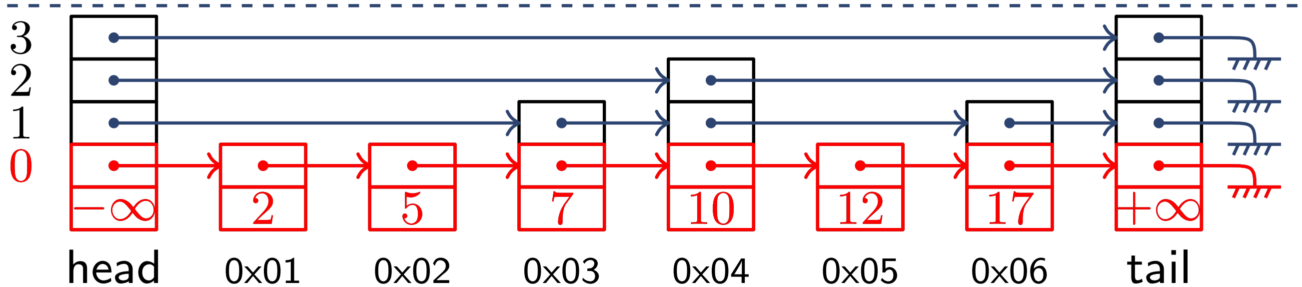


Verification of Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \hat{=} \left(ordList(heap, head, tail, 0) \right)$$

$M = 3$

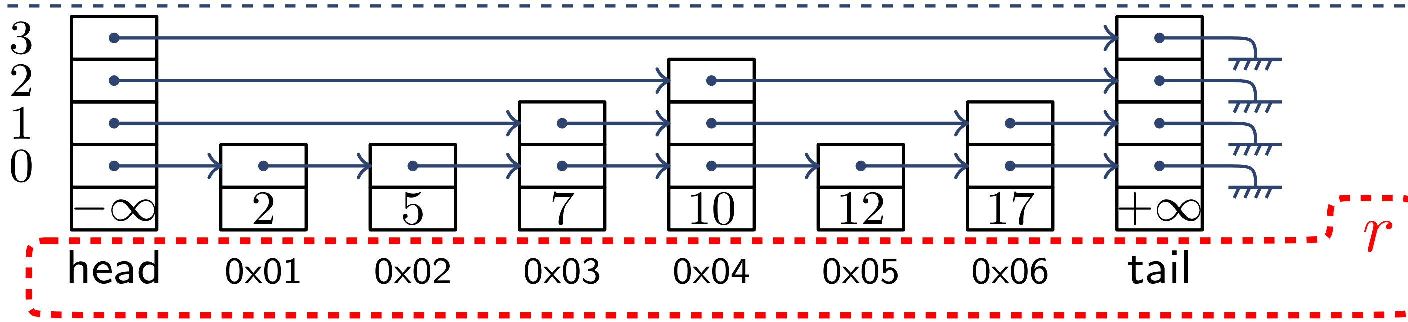


Verification of Skiplists

- **Skiplist** shape **preservation** : $\square SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \hat{=} \left(\begin{array}{c} ordList(heap, head, tail, 0) \quad \wedge \\ r = region(heap, head, tail) \end{array} \right)$$

$M = 3$

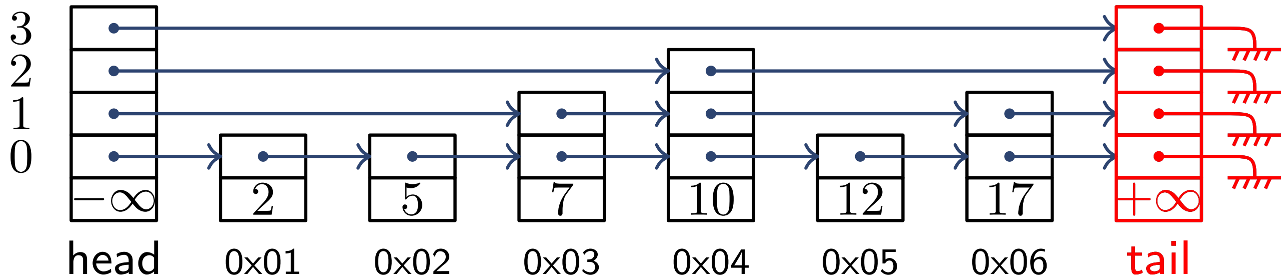


Verification of Skiplists

- **Skiplist** shape **preservation** : $\square SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \mathrel{\hat=} \left(\begin{array}{lr} ordList(heap, head, tail, 0) & \wedge \\ r = region(heap, head, tail) & \wedge \\ heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null & \end{array} \right)$$

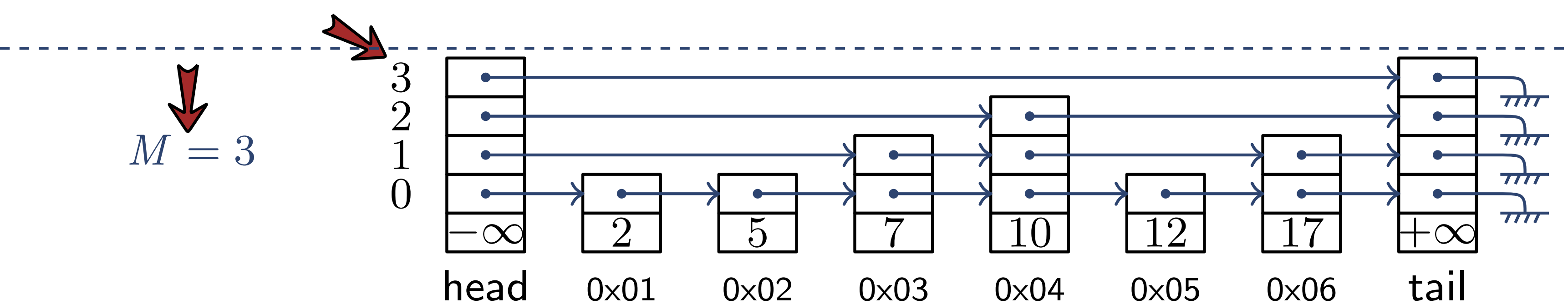
$M = 3$



Verification of Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \hat{=} \left(\begin{array}{ll} ordList(heap, head, tail, 0) & \wedge \\ r = region(heap, head, tail) & \wedge \\ heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null & \wedge \\ a \in r \rightarrow heap[a].level \leq M & \end{array} \right)$$

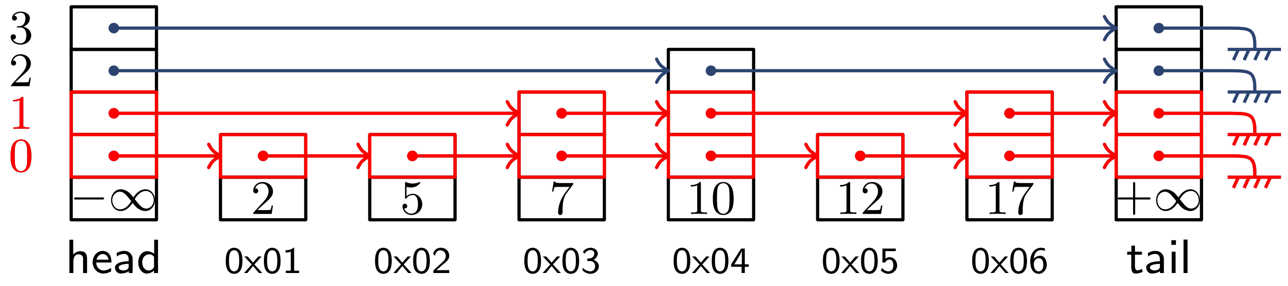


Verification of Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \triangleq \left(\begin{array}{rl} & ordList(heap, head, tail, 0) \wedge \\ & r = region(heap, head, tail) \wedge \\ & heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null \wedge \\ & a \in r \rightarrow heap[a].level \leq M \wedge \\ \bigwedge_{i \in 0 \ldots (M-1)} & addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) \end{array} \right)$$

$M = 3$

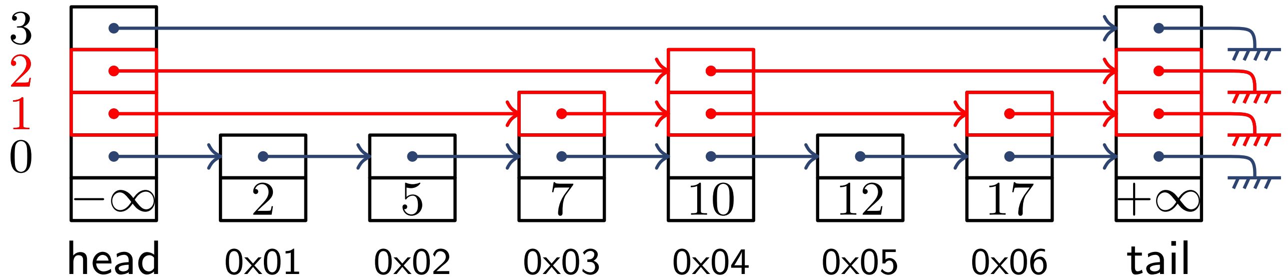


Verification of Skiplists

- **Skiplist** shape **preservation** : $\square SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \triangleq \left(\begin{array}{rl} & ordList(heap, head, tail, 0) \wedge \\ & r = region(heap, head, tail) \wedge \\ & heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null \wedge \\ & a \in r \rightarrow heap[a].level \leq M \wedge \\ \bigwedge_{i \in 0 \ldots (M-1)} & addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) \end{array} \right)$$

$M = 3$

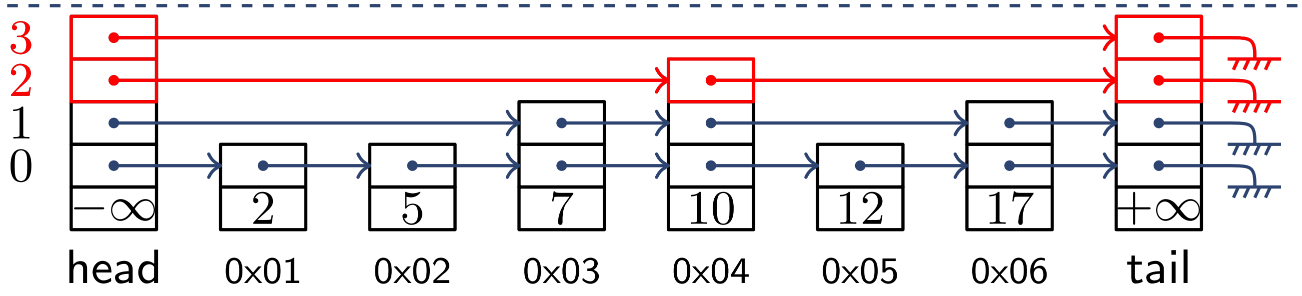


Verification of Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \mathrel{\hat{=}} \left(\begin{array}{ll} ordList(heap, head, tail, 0) & \wedge \\ r = region(heap, head, tail) & \wedge \\ heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null & \wedge \\ a \in r \rightarrow heap[a].level \leq M & \wedge \\ \bigwedge_{i \in 0 \ldots (M-1)} addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) & \end{array} \right)$$

$M = 3$



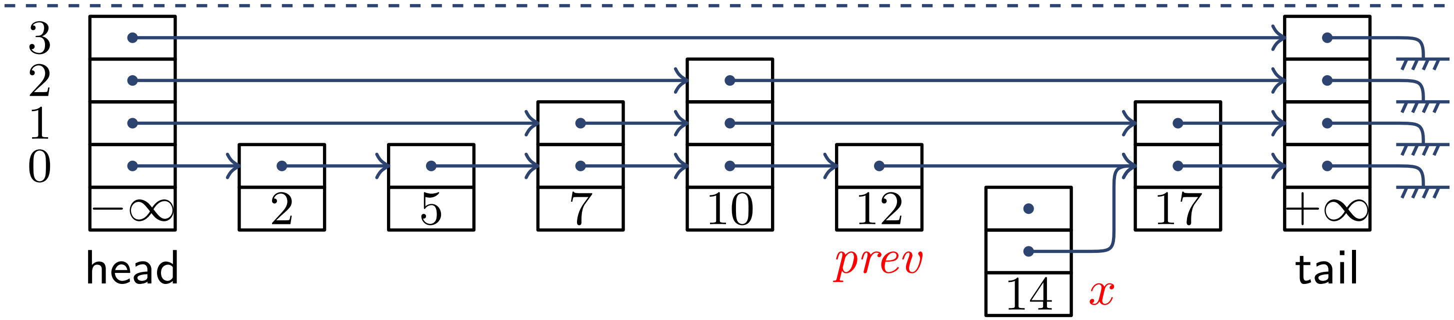
Verification of Skiplists

- **Skiplist** shape **preservation** : $\square SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \mathrel{\hat{=}} \left(\begin{array}{rll} & ordList(heap, head, tail, 0) & \wedge \\ & r = region(heap, head, tail) & \wedge \\ & heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null & \wedge \\ & a \in r \rightarrow heap[a].level \leq M & \wedge \\ \bigwedge\limits_{i \in 0 \ldots (M-1)} & addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) & \end{array} \right)$$

- Program **transitions** :

```
 9: . . .
10: prev.arr[0] := x
11: . . .
```



Verification of Skiplists

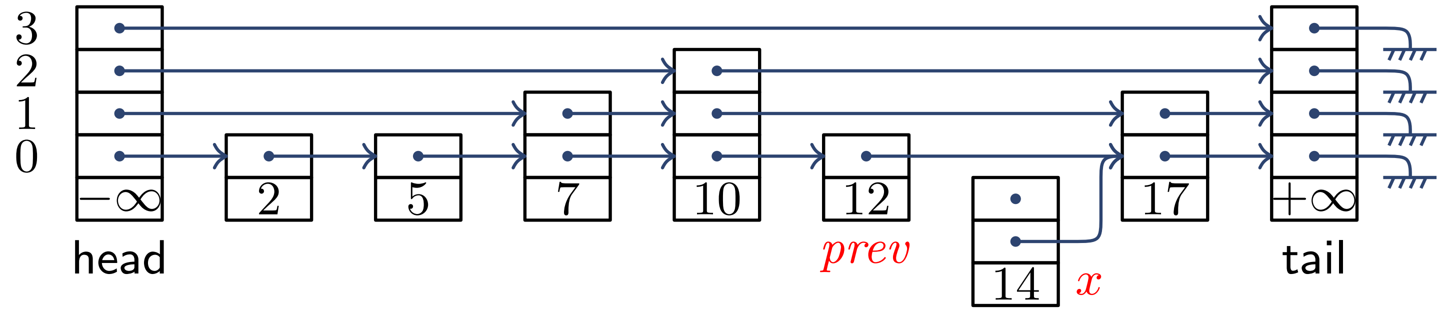
- **Skiplist** shape **preservation** : $\square SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \mathrel{\hat{=}} \left(\begin{array}{cc} & ordList(heap, head, tail, 0) \wedge \\ & r = region(heap, head, tail) \wedge \\ & heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null \wedge \\ & a \in r \rightarrow heap[a].level \leq M \wedge \\ \bigwedge_{i \in 0 \ldots (M-1)} & addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) \end{array} \right)$$

- Program **transitions** : $SL(h, sl, r)$

$SkipList(h, sl, r)$

```
 9: . . .
10: prev.arr[0] := x
11: . . .
```



Verification of Skiplists

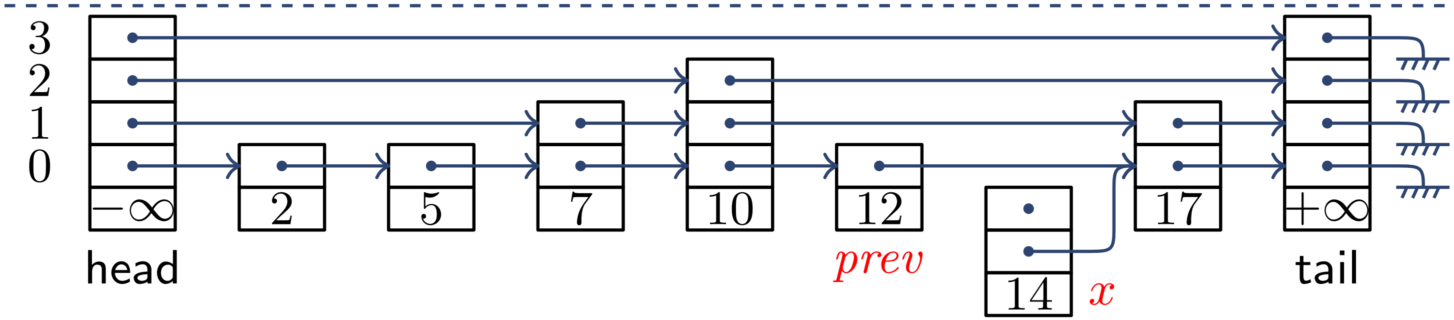
- **Skiplist** shape **preservation** : $\square SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \triangleq \left(\begin{array}{rll} & ordList(heap, head, tail, 0) & \wedge \\ & r = region(heap, head, tail) & \wedge \\ & heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null & \wedge \\ & a \in r \rightarrow heap[a].level \leq M & \wedge \\ \bigwedge_{i \in 0 \ldots (M-1)} & addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) & \end{array} \right)$$

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

$$SkipList(h, sl, r) \wedge \left(\begin{array}{rcll} x.val & = & 14 & \wedge \\ prev.val & < & 14 & \wedge \\ x.arr[0].val & > & 14 & \wedge \\ prev.arr[0] & = & x.arr[0] & \wedge \\ & x \notin r & & \end{array} \right)$$

```
 9: . . .
10: prev.arr[0] := x
11: . . .
```



Verification of Skiplists

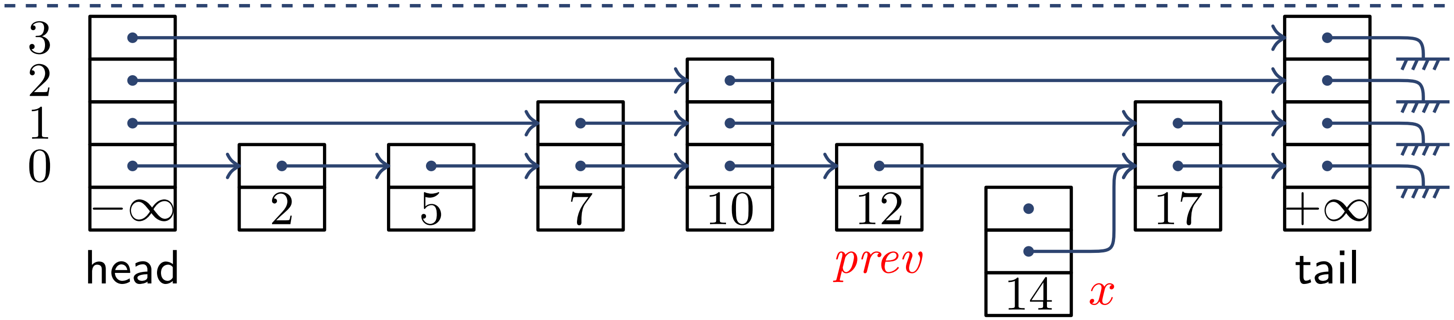
- **Skiplist** shape **preservation** : $\square SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \triangleq \left(\begin{array}{ll} ordList(heap, head, tail, 0) & \wedge \\ r = region(heap, head, tail) & \wedge \\ heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null & \wedge \\ a \in r \rightarrow heap[a].level \leq M & \wedge \\ \bigwedge_{i \in 0 \ldots (M-1)} addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) & \end{array} \right)$$

- Program **transitions** : $SL(h, sl, r) \wedge \varphi_{aux} \wedge \rho_{10}(V, V')$

$$SkipList(h, sl, r) \wedge \left(\begin{array}{rcll} x.val & = & 14 & \wedge \\ prev.val & < & 14 & \wedge \\ x.arr[0].val & > & 14 & \wedge \\ prev.arr[0] & = & x.arr[0] & \wedge \\ & x \notin r & & \end{array} \right) \wedge \left(\begin{array}{cl} at_{10} & \wedge \\ prev'.arr[0] = x & \wedge \\ at'_{11} & \wedge \\ h' = h \wedge sl = sl' & \wedge \\ r' = r \cup \{x\} \wedge x' = x & \ldots \end{array} \right)$$

```
 9: . . .
10: prev.arr[0] := x
11: . . .
```



Verification of Skiplists

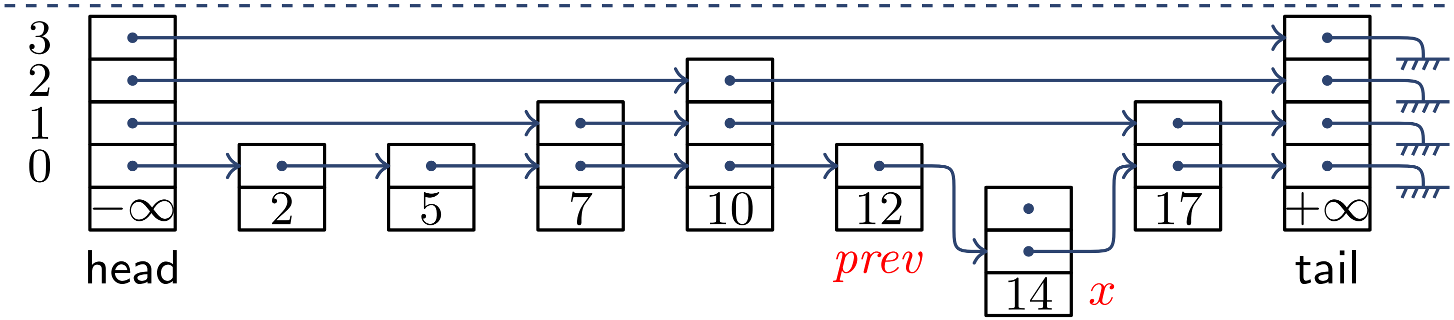
- **Skiplist** shape **preservation** : $\Box SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \triangleq \left(\begin{array}{rl} & ordList(heap, head, tail, 0) \quad \wedge \\ & r = region(heap, head, tail) \quad \wedge \\ & heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null \quad \wedge \\ & a \in r \to heap[a].level \leq M \quad \wedge \\ \bigwedge_{i \in 0 \ldots (M-1)} & addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) \end{array} \right)$$

- Program **transitions** : $SL(h, sl, r) \wedge \varphi_{aux} \wedge \rho_{10}(V, V')$

$$SkipList(h, sl, r) \wedge$$
$$\left(\begin{array}{rclc} x.val & = & 14 & \wedge \\ prev.val & < & 14 & \wedge \\ x.arr[0].val & > & 14 & \wedge \\ prev.arr[0] & = & x.arr[0] & \wedge \\ & x \notin r & & \end{array} \right) \wedge \left(\begin{array}{cc} at_{10} & \wedge \\ prev'.arr[0] = x & \wedge \\ at'_{11} & \wedge \\ h' = h \wedge sl = sl' & \wedge \\ r' = r \cup \{x\} \wedge x' = x & \ldots \end{array} \right)$$

```
9:  . . .
10: prev.arr[0] := x
11: . . .
```



Verification of Skiplists

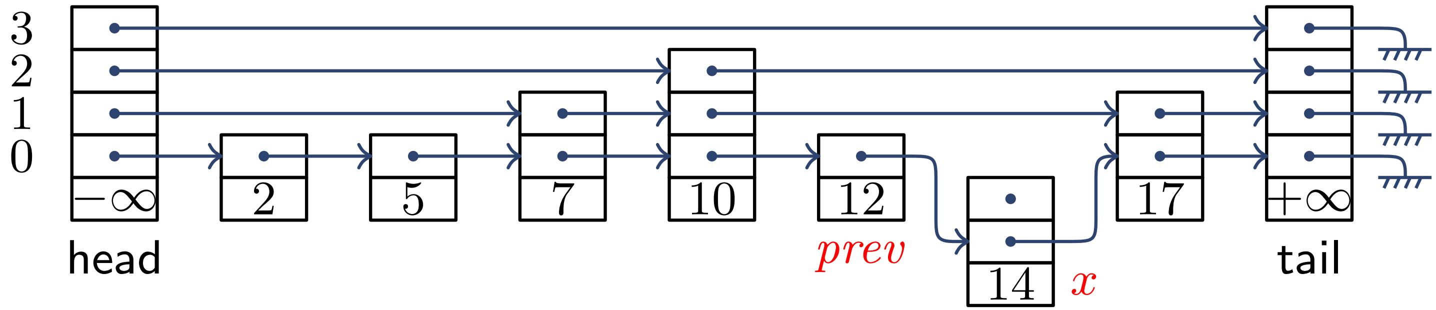
- **Skiplist** shape **preservation** : $\square SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \triangleq \left(\begin{array}{rl} & ordList(heap, head, tail, 0) \wedge \\ & r = region(heap, head, tail) \wedge \\ & heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null \wedge \\ & a \in r \rightarrow heap[a].level \leq M \wedge \\ \bigwedge_{i \in 0\ldots(M-1)} & addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) \end{array} \right)$$

- Program **transitions** : $SL(h, sl, r) \wedge \varphi_{aux} \wedge \rho_{10}(V, V')$

$$SkipList(h, sl, r) \wedge \left(\begin{array}{rcll} x.val & = & 14 & \wedge \\ prev.val & < & 14 & \wedge \\ x.arr[0].val & > & 14 & \wedge \\ prev.arr[0] & = & x.arr[0] & \wedge \\ & x \notin r & & \end{array} \right) \wedge \left(\begin{array}{cl} at_{10} & \wedge \\ prev'.arr[0] = x & \wedge \\ at'_{11} & \wedge \\ h' = h \wedge sl = sl' & \wedge \\ r' = r \cup \{x\} \wedge x' = x & \ldots \end{array} \right)$$

```
 9: . . .
10: prev.arr[0] := x
11: . . .
```



Verification of Skiplists

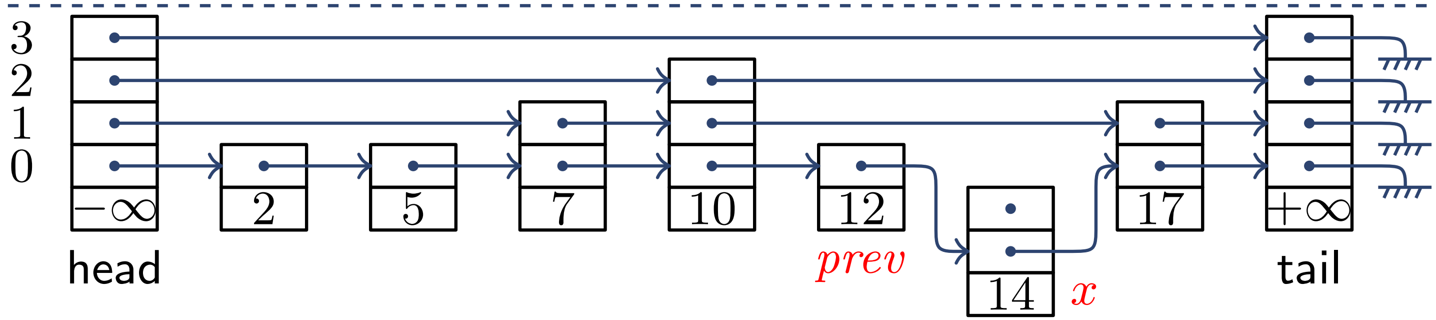
- **Skiplist** shape **preservation** : $\square SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \triangleq \left(\begin{array}{rlr} & ordList(heap, head, tail, 0) & \wedge \\ & r = region(heap, head, tail) & \wedge \\ & heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null & \wedge \\ & a \in r \rightarrow heap[a].level \leq M & \wedge \\ \bigwedge\limits_{i \in 0 \ldots (M-1)} & addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) & \end{array} \right)$$

- Program **transitions** : $SL(h, sl, r) \wedge \varphi_{aux} \wedge \rho_{10}(V, V') \rightarrow SL(h', sl', r')$

$$SkipList(h, sl, r) \wedge \left(\begin{array}{rcll} x.val & = & 14 & \wedge \\ prev.val & < & 14 & \wedge \\ x.arr[0].val & > & 14 & \wedge \\ prev.arr[0] & = & x.arr[0] & \wedge \\ & x \notin r & & \end{array} \right) \wedge \left(\begin{array}{cr} at_{10} & \wedge \\ prev'.arr[0] = x & \wedge \\ at'_{11} & \wedge \\ h' = h \wedge sl = sl' & \wedge \\ r' = r \cup \{x\} \wedge x' = x & \ldots \end{array} \right) \rightarrow SkipList(h', sl', r')$$

```
9:  . . .
10: prev.arr[0] := x
11: . . .
```



Verification of Skiplists

- **Skiplist** shape **preservation** : $\square SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \triangleq \left(\begin{array}{rl} & ordList(heap, head, tail, 0) \quad \wedge \\ & r = region(heap, head, tail) \quad \wedge \\ & heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null \quad \wedge \\ & a \in r \rightarrow heap[a].level \leq M \quad \wedge \\ \bigwedge_{i \in 0 \ldots (M-1)} & addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) \end{array} \right)$$

- Program **transitions** : $SL(h, sl, r) \wedge \varphi_{aux} \wedge \rho_{10}(V, V') \rightarrow SL(h', sl', r'$

$$SkipList(h, sl, r) \wedge$$

$$\left(\begin{array}{rcll} x.val & = & 14 & \wedge \\ prev.val & < & 14 & \wedge \\ x.arr[0].val & > & 14 & \wedge \\ prev.arr[0] & = & x.arr[0] & \wedge \\ & x \notin r & & \end{array} \right) \wedge \left(\begin{array}{cc} at_{10} & \wedge \\ prev'.arr[0] = x & \wedge \\ at'_{11} & \wedge \\ h' = h \wedge sl = sl' & \wedge \\ r' = r \cup \{x\} \wedge x' = x & \ldots \end{array} \right) \rightarrow SkipList(h', sl', r')$$

reason about

Verification of Skiplists

- **Skiplist** shape **preservation** : $\square SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \triangleq \left(\begin{array}{ll} ordList(heap, head, tail, 0) & \wedge \\ r = region(heap, head, tail) & \wedge \\ heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null & \wedge \\ a \in r \rightarrow heap[a].level \leq M & \wedge \\ \bigwedge_{i \in 0 \ldots (M-1)} addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) & \end{array} \right)$$

- Program **transitions** : $SL(h, sl, r) \wedge \varphi_{aux} \wedge \rho_{10}(V, V') \rightarrow SL(h', sl', r')$

$$SkipList(h, sl, r) \wedge \left(\begin{array}{lcll} x.val & = & 14 & \wedge \\ prev.val & < & 14 & \wedge \\ x.arr[0].val & > & 14 & \wedge \\ prev.arr[0] & = & x.arr[0] & \wedge \\ & x \notin r & & \end{array} \right) \wedge \left(\begin{array}{cl} at_{10} & \wedge \\ prev'.arr[0] = x & \wedge \\ at'_{11} & \wedge \\ h' = h \wedge sl = sl' & \wedge \\ r' = r \cup \{x\} \wedge x' = x & \ldots \end{array} \right) \rightarrow SkipList(h', sl', r')$$

reason about

ordered values + notion of ordered list

Verification of Skiplists

- **Skiplist** shape **preservation** : $\square SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \triangleq \left(\begin{array}{rl} & ordList(heap, head, tail, 0) \wedge \\ & r = region(heap, head, tail) \wedge \\ & heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null \wedge \\ & a \in r \rightarrow heap[a].level \leq M \wedge \\ \bigwedge_{i \in 0 \ldots (M-1)} & addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) \end{array} \right)$$

- Program **transitions** : $SL(h, sl, r) \wedge \varphi_{aux} \wedge \rho_{10}(V, V') \rightarrow SL(h', sl', r')$

$$SkipList(h, sl, r) \wedge \left(\begin{array}{rcll} x.val & = & 14 & \wedge \\ prev.val & < & 14 & \wedge \\ x.arr[0].val & > & 14 & \wedge \\ prev.arr[0] & = & x.arr[0] & \wedge \\ & x \notin r & & \end{array} \right) \wedge \left(\begin{array}{cc} at_{10} & \wedge \\ prev'.arr[0] = x & \wedge \\ at'_{11} & \wedge \\ h' = h \wedge sl = sl' & \wedge \\ r' = r \cup \{x\} \wedge x' = x & \ldots \end{array} \right) \rightarrow SkipList(h', sl', r')$$

reason about

levels

Verification of Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \triangleq \left(\begin{array}{ll} & ordList(heap, head, tail, 0) \wedge \\ & r = region(heap, head, tail) \wedge \\ & heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null \wedge \\ & a \in r \rightarrow heap[a].level \leq M \wedge \\ \bigwedge_{i \in 0 \ldots (M-1)} & addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) \end{array} \right)$$

- Program **transitions** : $SL(h, sl, r) \wedge \varphi_{aux} \wedge \rho_{10}(V, V') \rightarrow SL(h', sl', r')$

$$SkipList(h, sl, r) \wedge \left(\begin{array}{rcll} x.val & = & 14 & \wedge \\ prev.val & < & 14 & \wedge \\ x.arr[0].val & > & 14 & \wedge \\ prev.arr[0] & = & x.arr[0] & \wedge \\ & x \notin r & & \end{array} \right) \wedge \left(\begin{array}{cl} at_{10} & \wedge \\ prev'.arr[0] = x & \wedge \\ at'_{11} & \wedge \\ h' = h \wedge sl = sl' & \wedge \\ r' = r \cup \{x\} \wedge x' = x & \ldots \end{array} \right) \rightarrow SkipList(h', sl', r')$$

reason about

arrays

Verification of Skiplists

- **Skiplist** shape **preservation** : $\square SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \triangleq \left(\begin{array}{ll} ordList(heap, head, tail, 0) & \wedge \\ r = region(heap, head, tail) & \wedge \\ heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null & \wedge \\ a \in r \rightarrow heap[a].level \leq M & \wedge \\ \bigwedge_{i \in 0 \ldots (M-1)} addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) & \end{array} \right)$$

- Program **transitions** : $SL(h, sl, r) \wedge \varphi_{aux} \wedge \rho_{10}(V, V') \rightarrow SL(h', sl', r')$

$$SkipList(h, sl, r) \wedge$$
$$\left(\begin{array}{rcll} x.val & = & 14 & \wedge \\ prev.val & < & 14 & \wedge \\ x.arr[0].val & > & 14 & \wedge \\ prev.arr[0] & = & x.arr[0] & \wedge \\ & x \notin r & & \end{array} \right) \wedge \left(\begin{array}{cc} at_{10} & \wedge \\ prev'.arr[0] = x & \wedge \\ at'_{11} & \wedge \\ h' = h \wedge sl = sl' & \wedge \\ r' = r \cup \{x\} \wedge x' = x & \ldots \end{array} \right) \rightarrow SkipList(h', sl', r')$$

reason about

regions (sets)

Verification of Skiplists

- **Skiplist** shape **preservation** : $\Box SkipList(heap, sl, r)$

$$SkipList(heap, sl, r) \triangleq \left(\begin{array}{ll} ordList(heap, head, tail, 0) & \wedge \\ r = region(heap, head, tail) & \wedge \\ heap[tail].arr[0] = null \wedge \cdots \wedge heap[tail].arr[M]) = null & \wedge \\ a \in r \rightarrow heap[a].level \leq M & \wedge \\ \bigwedge_{i \in 0 \ldots (M-1)} addrs(heap, head, tail, i+1) \subseteq addrs(heap, head, tail, i) & \end{array} \right)$$

- Program **transitions** : $SL(h, sl, r) \wedge \varphi_{aux} \wedge \rho_{10}(V, V') \rightarrow SL(h', sl', r')$

$$SkipList(h, sl, r) \wedge$$
$$\left(\begin{array}{rclc} x.val & = & 14 & \wedge \\ prev.val & < & 14 & \wedge \\ x.arr[0].val & > & 14 & \wedge \\ prev.arr[0] & = & x.arr[0] & \wedge \\ & x \notin r & & \end{array} \right) \wedge \left(\begin{array}{cc} at_{10} & \wedge \\ prev'.arr[0] = x & \wedge \\ at'_{11} & \wedge \\ h' = h \wedge sl = sl' & \wedge \\ r' = r \cup \{x\} \wedge x' = x & \ldots \end{array} \right) \rightarrow SkipList(h', sl', r')$$

reason about

memory, cells

Our Contribution

- **TSL**, a theory for skiplists of **arbitrary length and height**
- We show TSL **decidable**...
- ...by reducing **TSL satisfiability** to **TSL_K satisfiability**.
- Show it suitable for verifying **real world implementations**

TSL:Theory of Skiplist of Arbitrary Height

TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

Σ_{addr}

TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

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

TSL:Theory of Skiplist of Arbitrary Height

- TSL, like $\mathsf{TSL_K}$, is a **union of other theories**

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

TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

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

TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

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

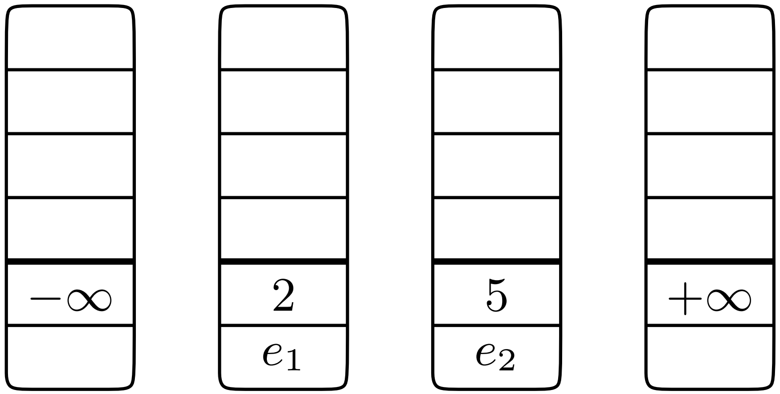
TSL_K	TSL

TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

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

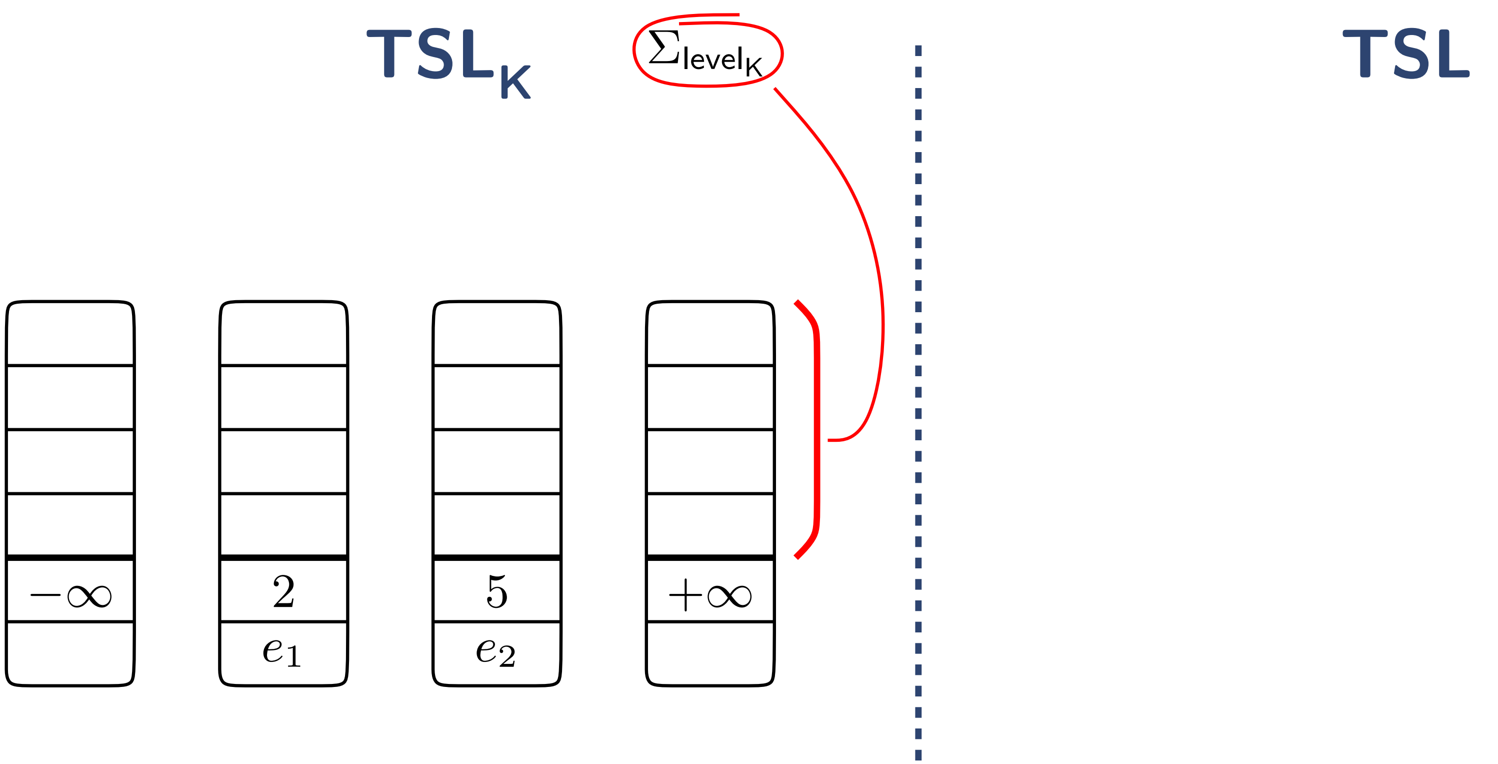
TSL_K | **TSL**



TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

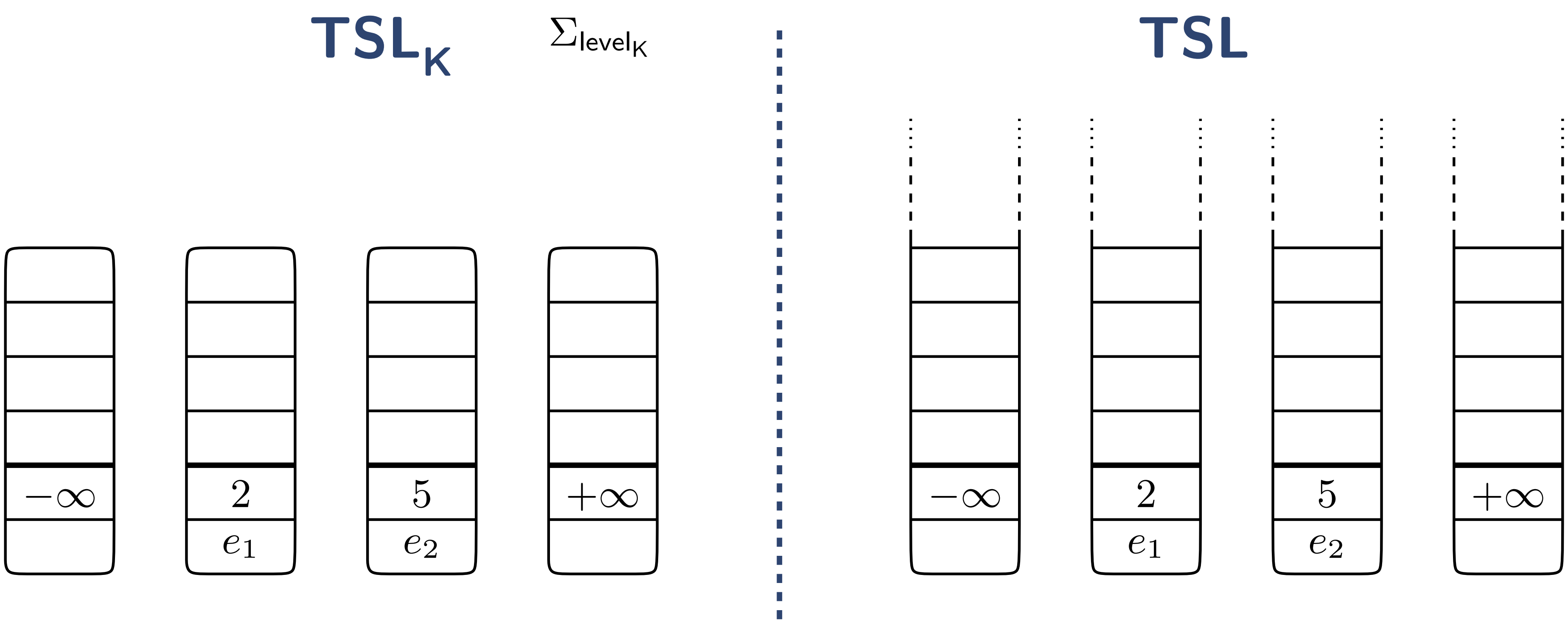
$\Sigma_{\text{addr}} \cup \Sigma_{\text{elem}} \cup \Sigma_{\text{ord}} \cup \Sigma_{\text{cell}}$



TSL:Theory of Skiplist of Arbitrary Height

- TSL, like $\mathsf{TSL_K}$, is a **union of other theories**

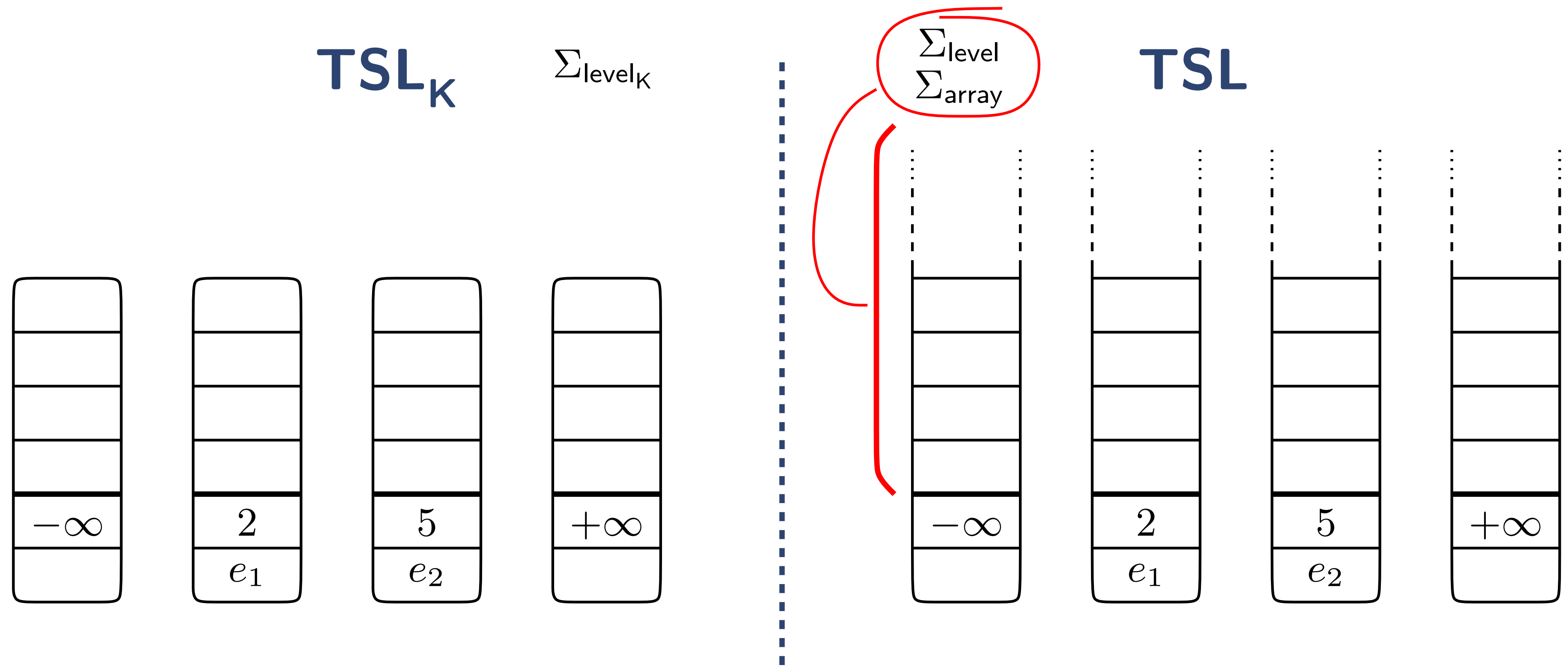
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}}$$



TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

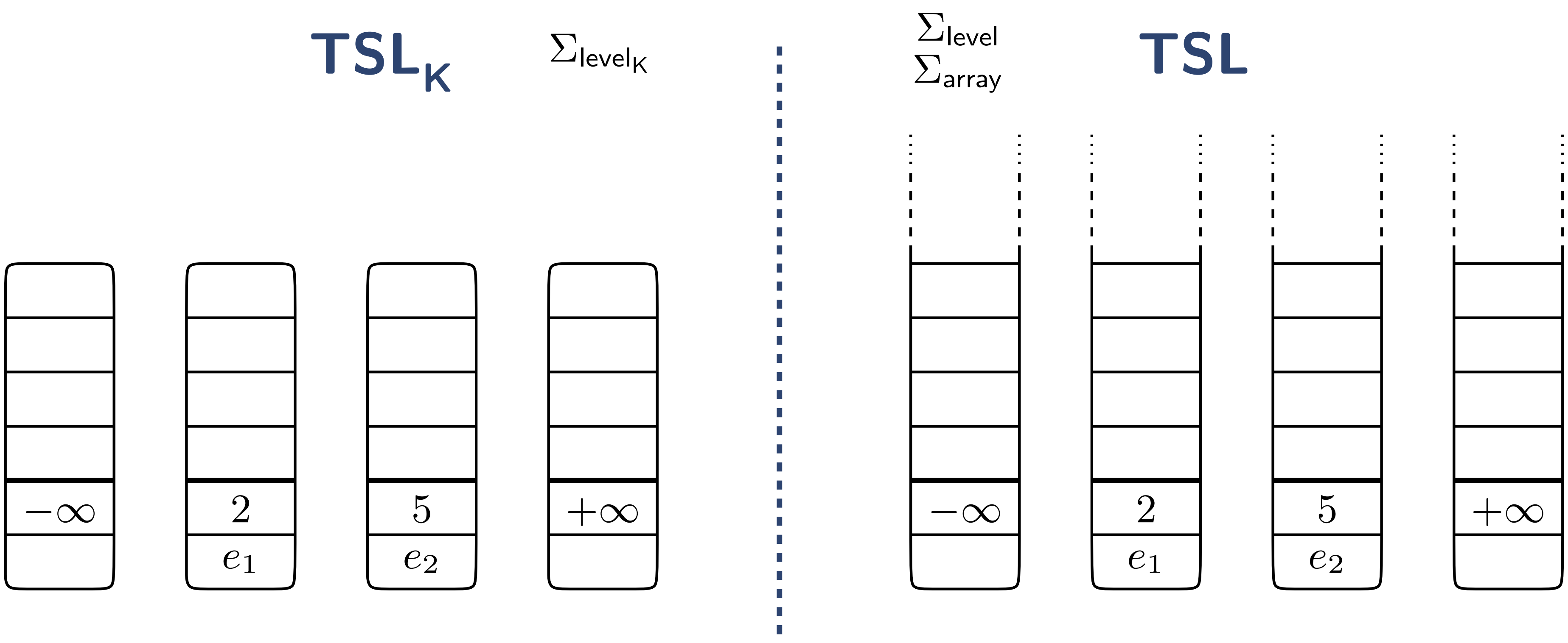
$$\Sigma_{\text{addr}} \cup \Sigma_{\text{elem}} \cup \Sigma_{\text{ord}} \cup \Sigma_{\text{cell}}$$



TSL: Theory of Skiplist of Arbitrary Height

- TSL, like $\mathsf{TSL_K}$, is a **union of other theories**

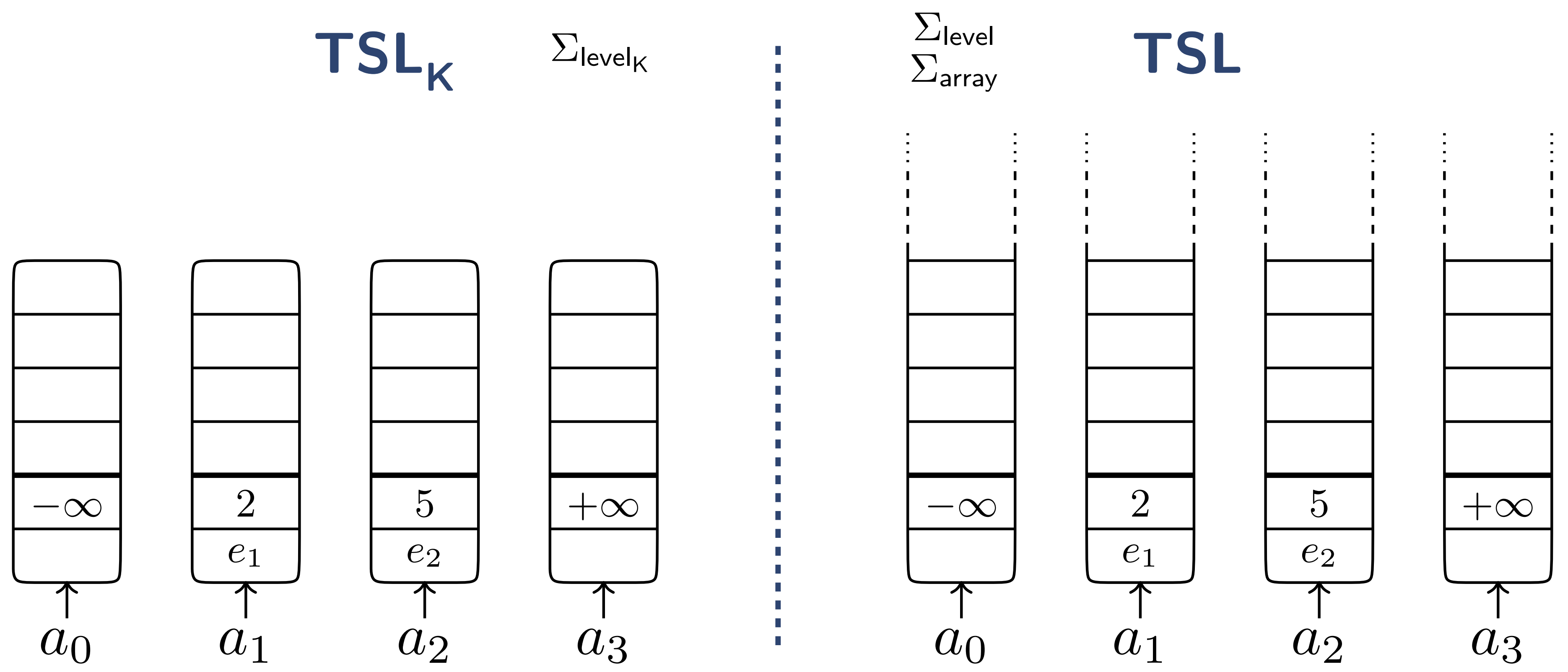
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{mem}}$$



TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

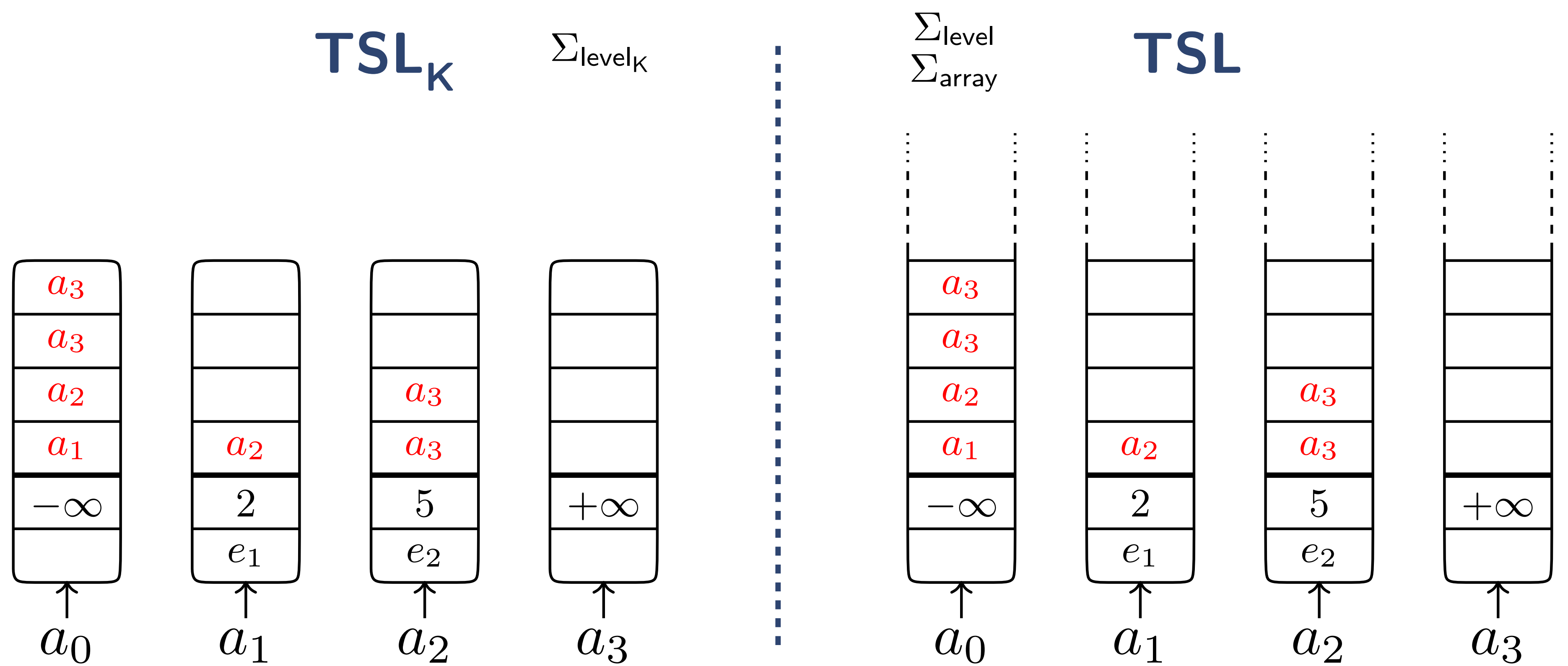
$$\Sigma_{\text{addr}} \cup \Sigma_{\text{elem}} \cup \Sigma_{\text{ord}} \cup \Sigma_{\text{cell}} \cup \Sigma_{\text{mem}}$$



TSL:Theory of Skiplist of Arbitrary Height

- TSL, like $\mathsf{TSL_K}$, is a **union of other theories**

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



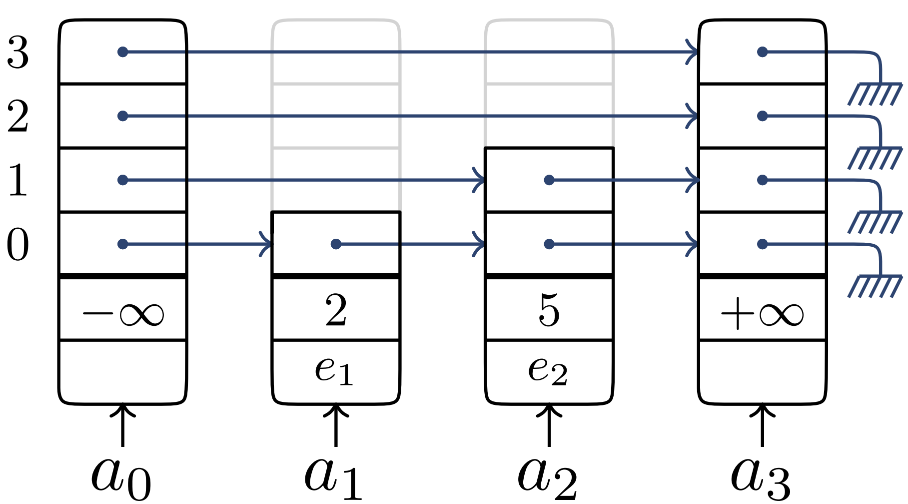
TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

$$\Sigma_{\text{addr}} \cup \Sigma_{\text{elem}} \cup \Sigma_{\text{ord}} \cup \Sigma_{\text{cell}} \cup \Sigma_{\text{mem}}$$

TSL_K

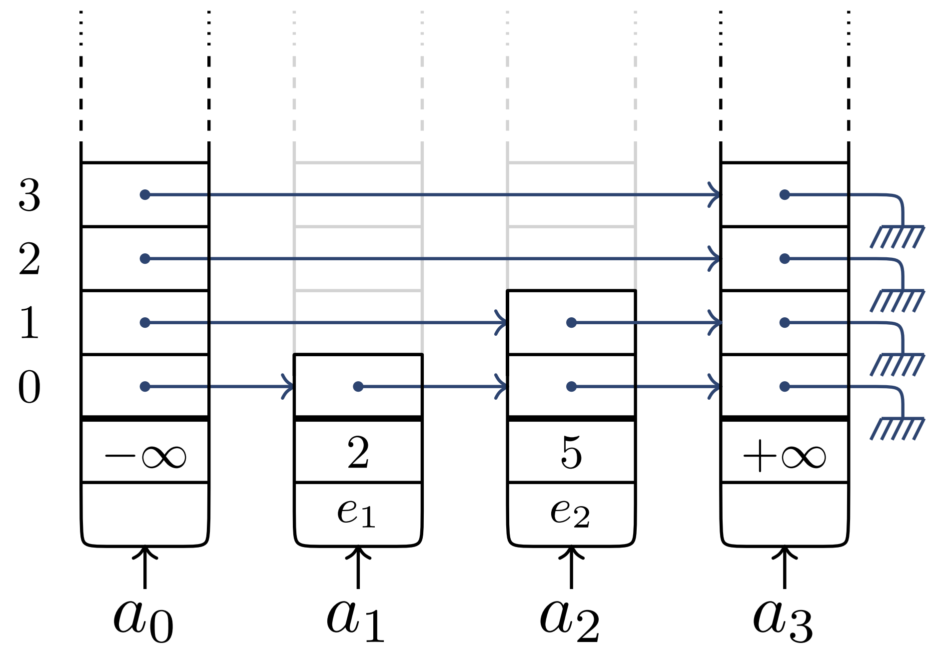
$\Sigma_{\text{level}_\text{K}}$



Σ_{level}

Σ_{array}

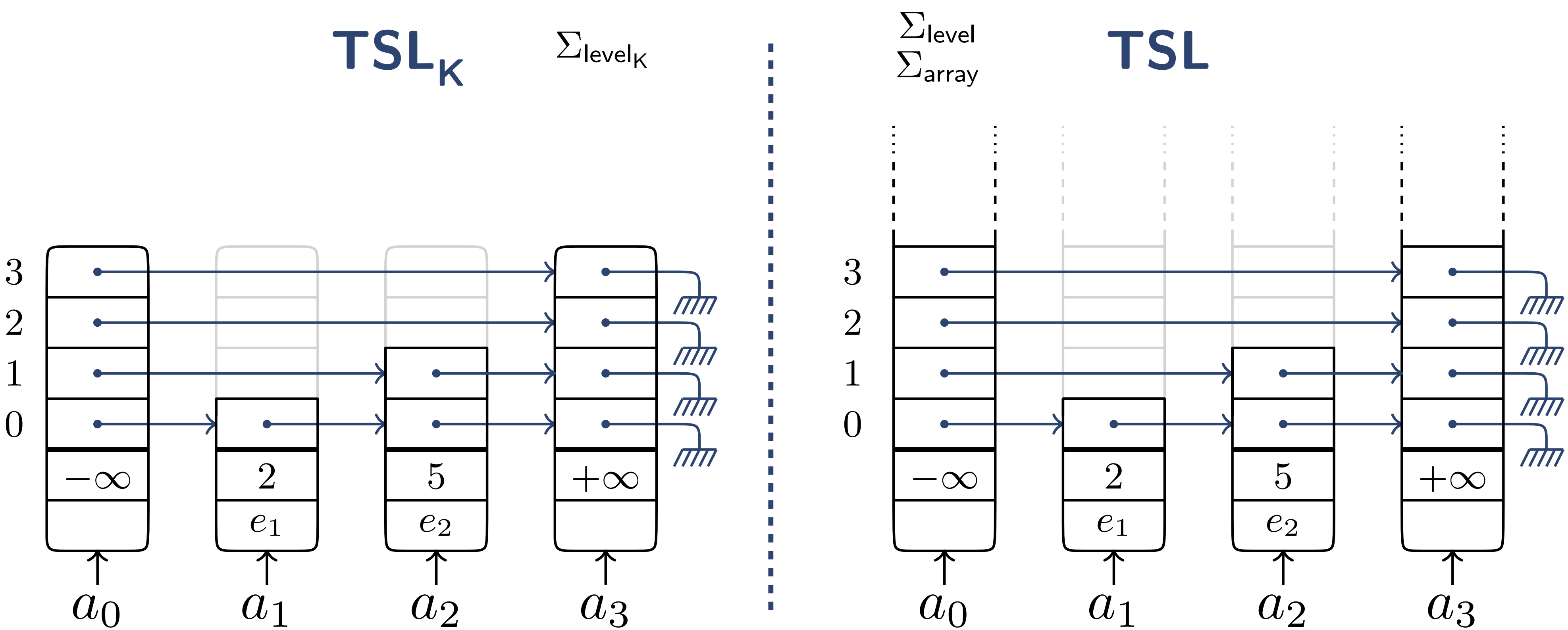
TSL



TSL: Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

$$\Sigma_{\text{addr}} \cup \Sigma_{\text{elem}} \cup \Sigma_{\text{ord}} \cup \Sigma_{\text{cell}} \cup \Sigma_{\text{mem}} \cup \Sigma_{\text{set}}$$



TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

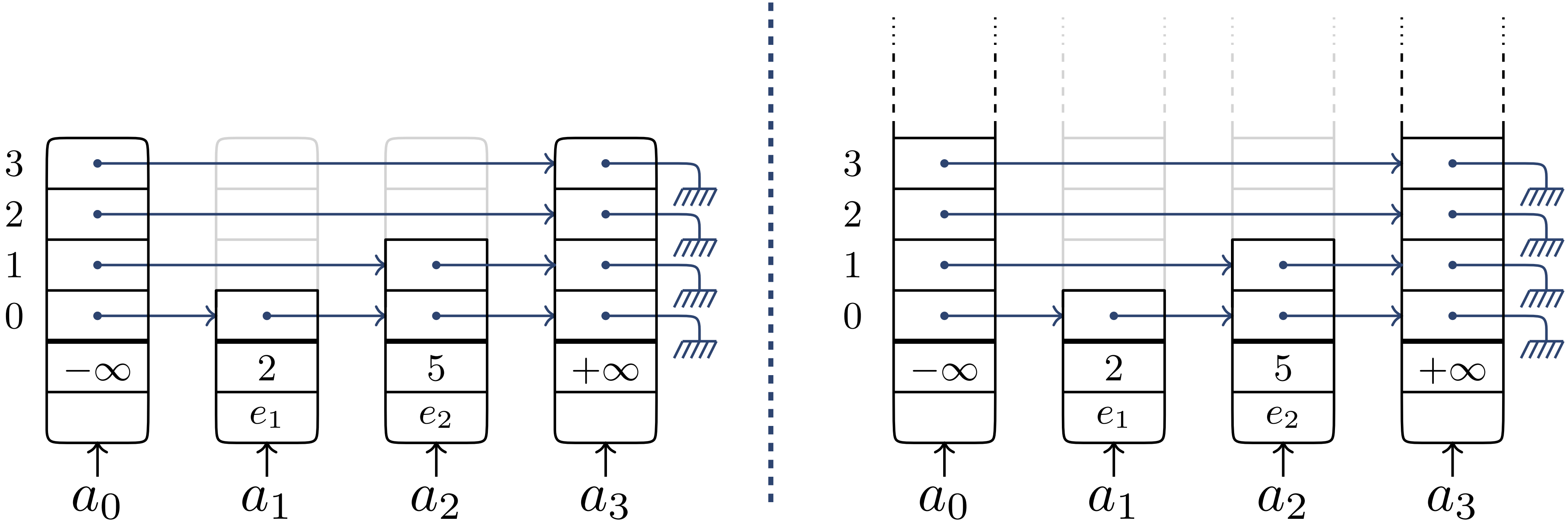
$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{reachability}}$$

path = a non-repeating sequence of addresses

$$[a_1, a_2, a_3]$$

TSL_K $\Sigma_{\mathsf{level}_\mathsf{K}}$

Σ_{level} Σ_{array} **TSL**



TSL:Theory of Skiplist of Arbitrary Height

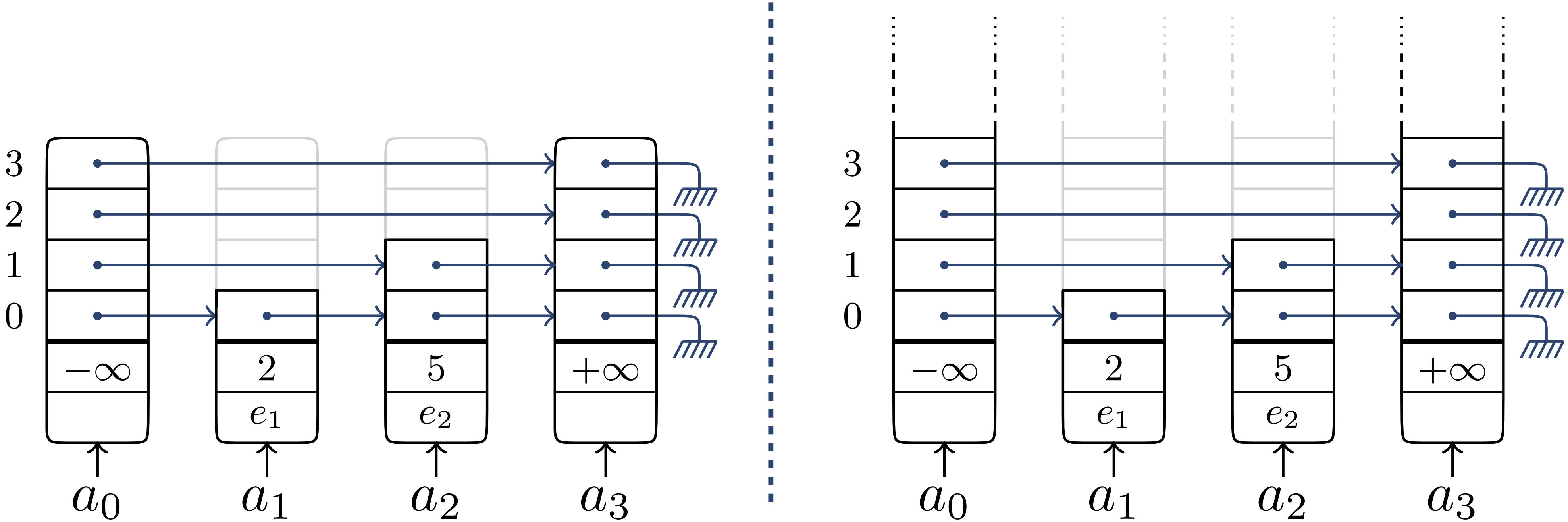
- TSL, like TSL_K, is a **union of other theories**

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

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

TSL_K $\Sigma_{\mathsf{level}_\mathsf{K}}$

Σ_{level} Σ_{array} **TSL**

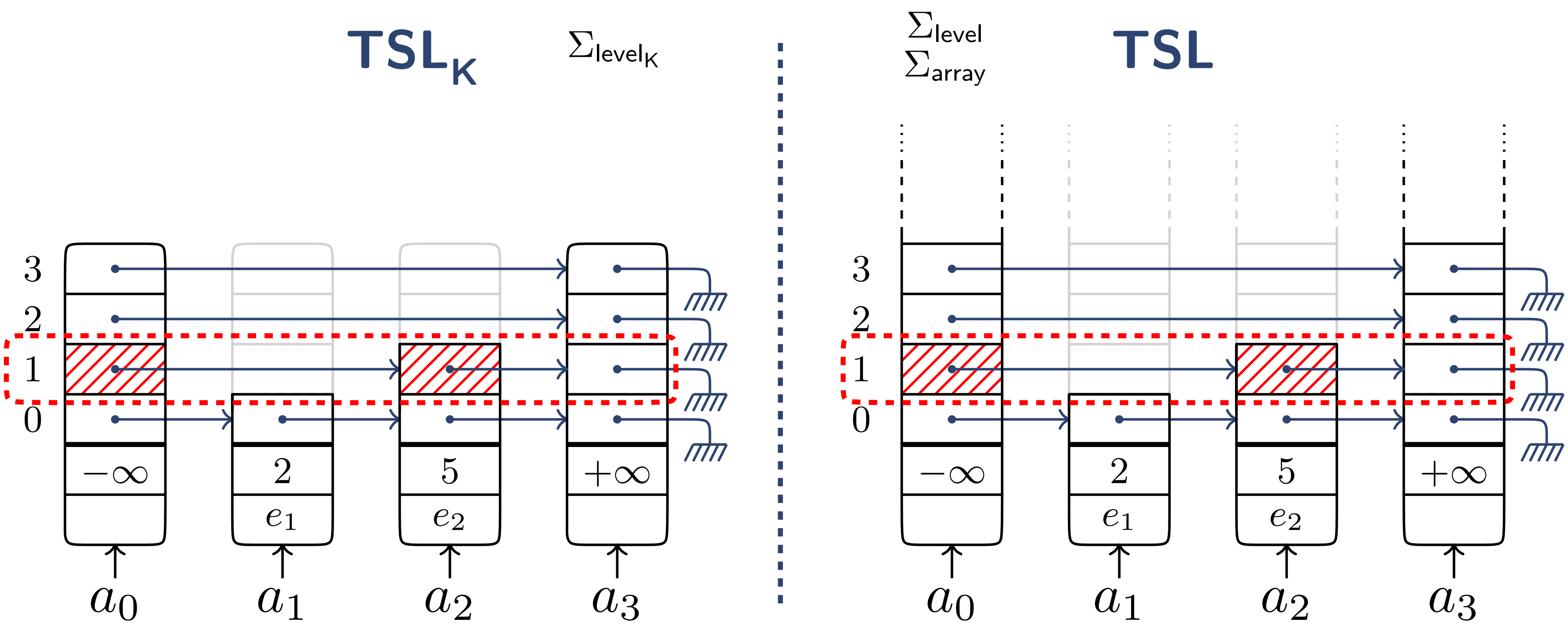


TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

$$\Sigma_{\text{addr}} \cup \Sigma_{\text{elem}} \cup \Sigma_{\text{ord}} \cup \Sigma_{\text{cell}} \cup \Sigma_{\text{mem}} \cup \Sigma_{\text{set}} \cup \Sigma_{\text{reachability}}$$

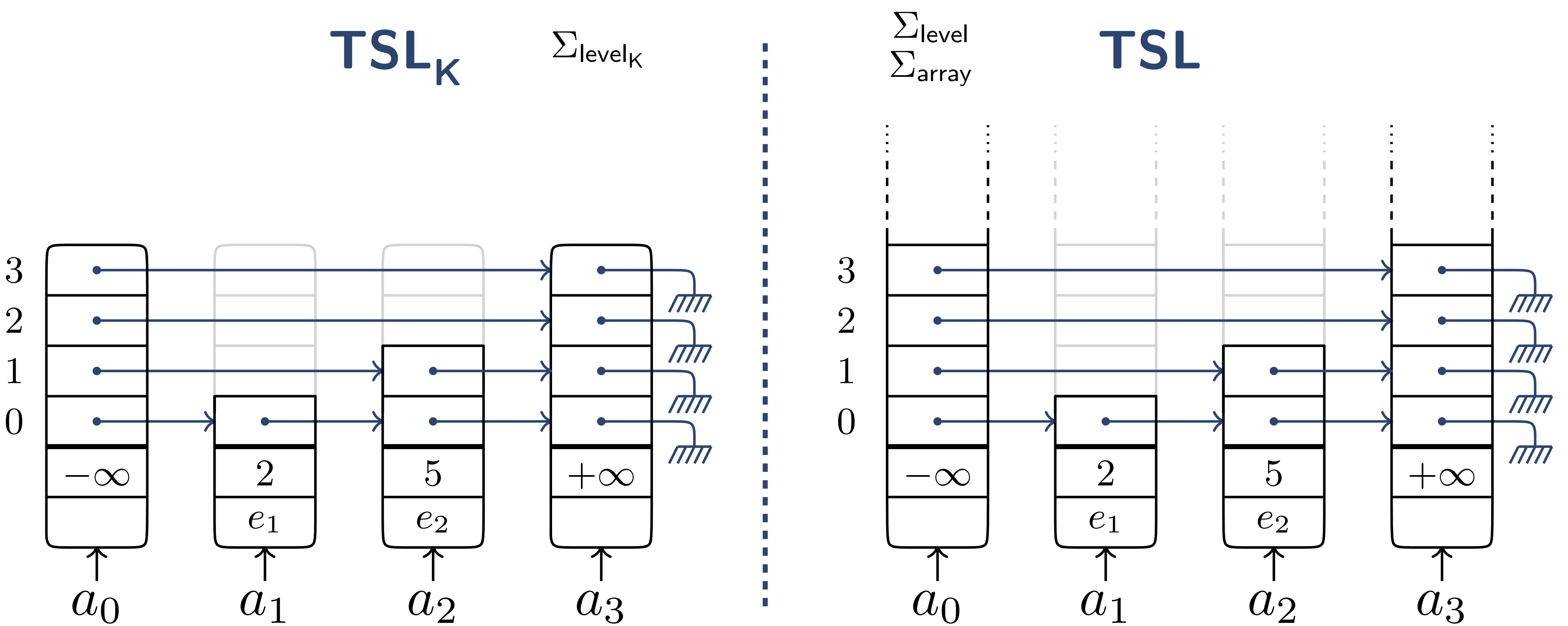
$$reach(a_0, a_3, 1, [a_0, a_2])$$



TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

$$\Sigma_{\text{addr}} \cup \Sigma_{\text{elem}} \cup \Sigma_{\text{ord}} \cup \Sigma_{\text{cell}} \cup \Sigma_{\text{mem}} \cup \Sigma_{\text{set}} \cup \Sigma_{\text{reachability}} \cup \Sigma_{\text{bridge}}$$

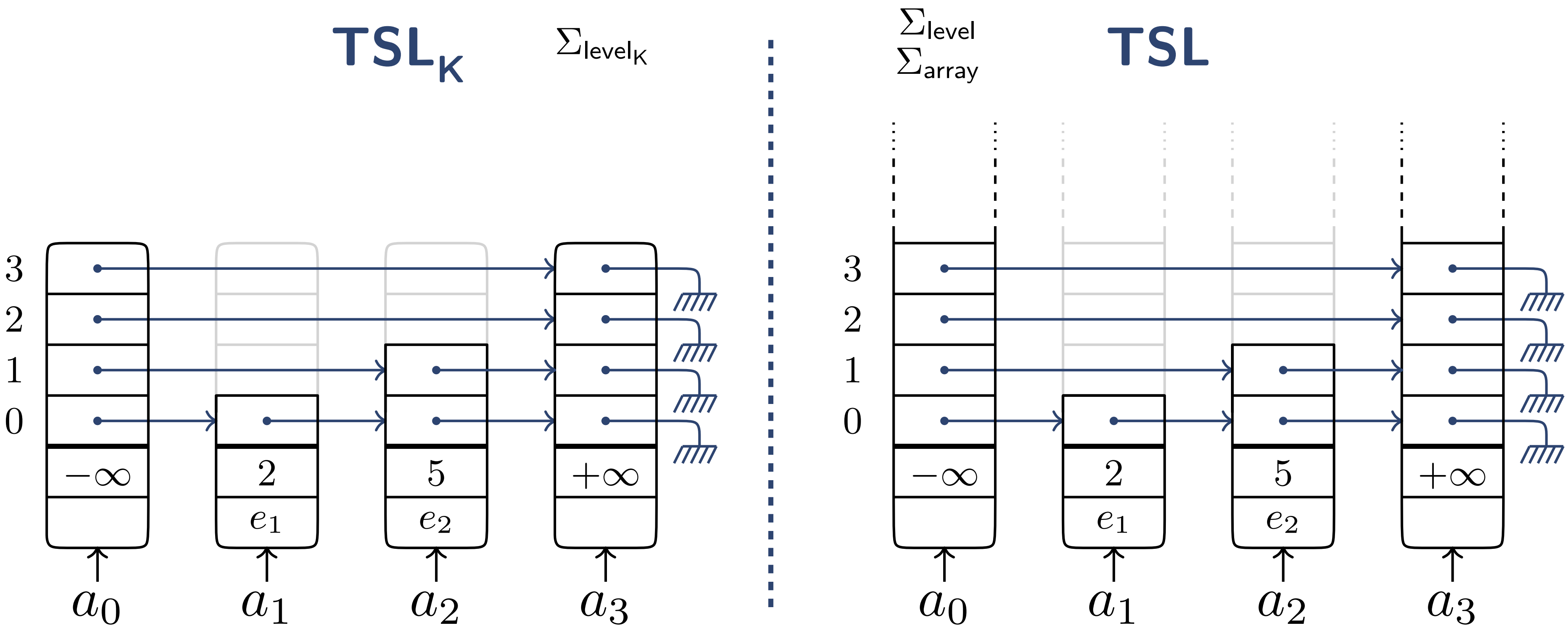


TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

$$\Sigma_{\text{addr}} \cup \Sigma_{\text{elem}} \cup \Sigma_{\text{ord}} \cup \Sigma_{\text{cell}} \cup \Sigma_{\text{mem}} \cup \Sigma_{\text{set}} \cup \Sigma_{\text{reachability}} \cup \Sigma_{\text{bridge}}$$

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

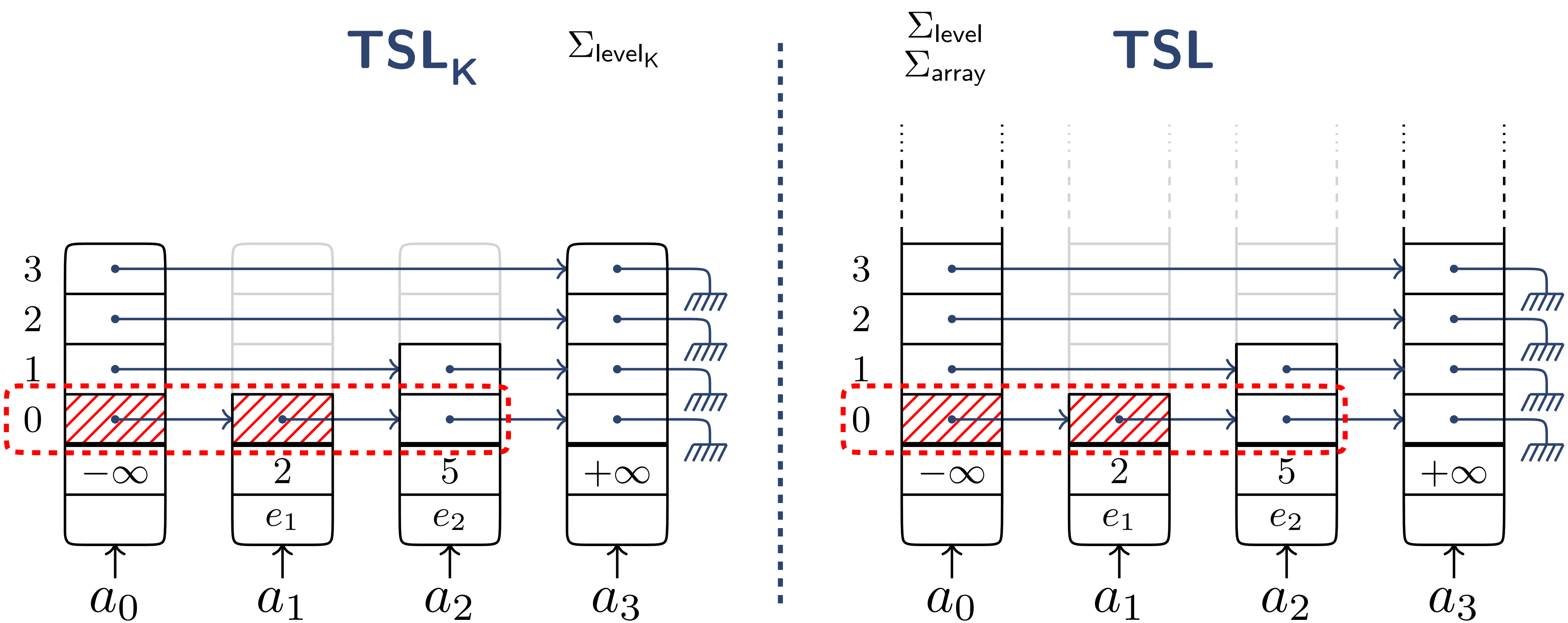


TSL: Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{reachability}} \cup \Sigma_{\mathsf{bridge}}$$

$$getp(a_0, a_2, 0) = [a_0, a_1]$$

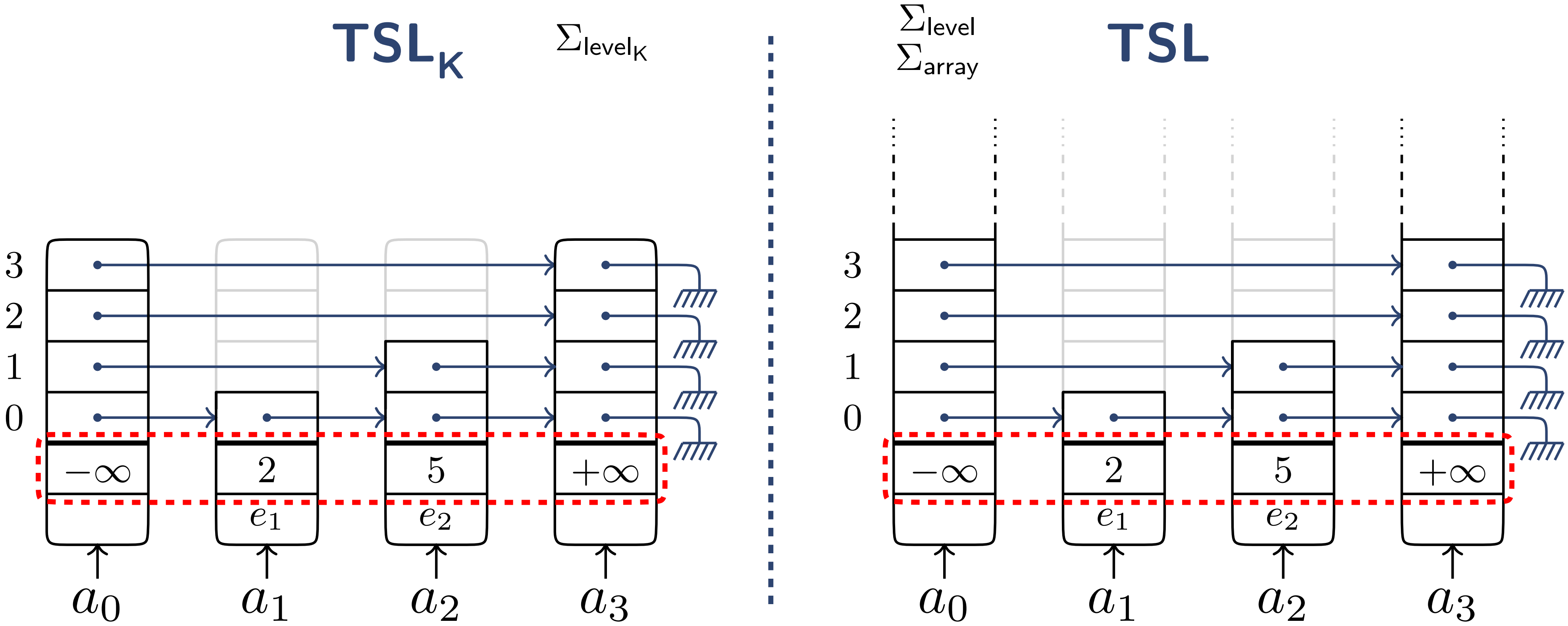


TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{reachability}} \cup \Sigma_{\mathsf{bridge}}$$

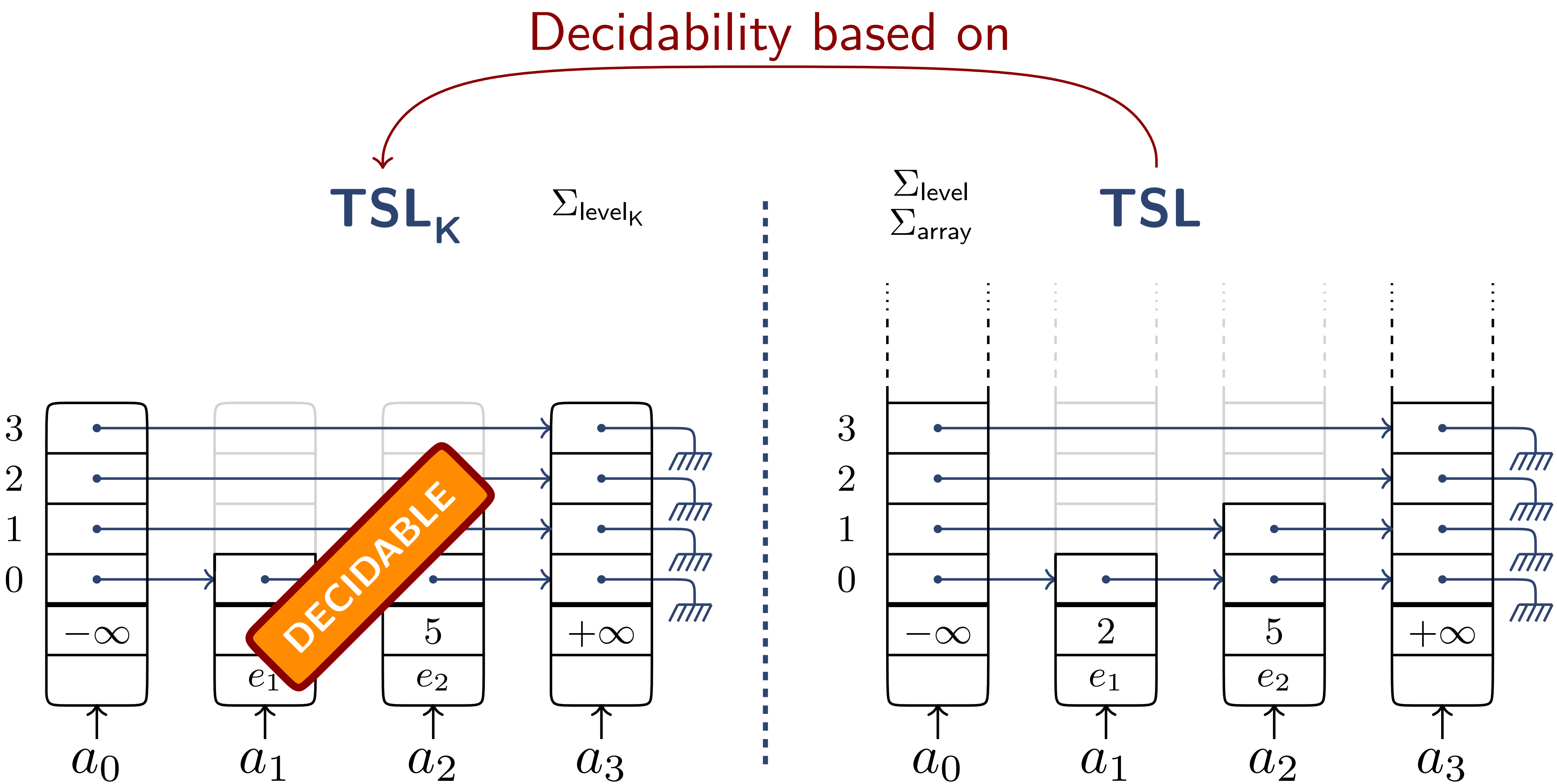
$$ordList([a_0, a_1, a_2, a_3])$$



TSL:Theory of Skiplist of Arbitrary Height

- TSL, like TSL_K, is a **union of other theories**

$$\Sigma_{\mathsf{addr}} \cup \Sigma_{\mathsf{elem}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{reachability}} \cup \Sigma_{\mathsf{bridge}}$$



Decision Procedure for TSL

Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula

Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula

$$\varphi \quad : \quad i = 0 \wedge \begin{pmatrix} A = heap[head].arr \quad \wedge \\ heap[head].max = 3 \end{pmatrix} \wedge B = A\{i \leftarrow tail\}$$

Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula

$$\varphi \quad : \quad i = 0 \wedge \underbrace{\begin{pmatrix} A = heap[head].arr & \wedge \\ heap[head].max = 3 & \end{pmatrix}} \wedge B = A\{i \leftarrow tail\}$$

Normalization

$$\varphi_{\mathsf{norm}} \quad : \quad i = 0 \wedge \begin{pmatrix} c = heap[head] & \wedge \\ c = mkcell(e, k, A, l) & \wedge \\ l = 3 & \end{pmatrix} \wedge B = A\{i \leftarrow tail\}$$

Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula
- **STEP 1.** Sanitize

$$\varphi_{\mathsf{sanit}} := \varphi_{\mathsf{norm}} \wedge \bigwedge_{B=A\{l\leftarrow a\} \in \varphi_{\mathsf{norm}}} (l_{new} = l + 1)$$

$$\varphi_{\mathsf{norm}} \quad : \quad i = 0 \wedge \left(\begin{array}{ll} c = heap[head] & \wedge \\ c = mkcell(e, k, A, l) & \wedge \\ l = 3 & \end{array} \right) \wedge B = A\{i \leftarrow tail\}$$

Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula
- **STEP 1.** Sanitize

$$\varphi_{\mathsf{sanit}} := \varphi_{\mathsf{norm}} \wedge \bigwedge_{B=A\{l \leftarrow a\} \,\in\, \varphi_{\mathsf{norm}}} (l_{new} = l + 1)$$

$$\varphi_{\mathsf{norm}} \quad : \quad i = 0 \wedge \begin{pmatrix} c = heap[head] & \wedge \\ c = mkcell(e, k, A, l) & \wedge \\ l = 3 & \end{pmatrix} \wedge \underbrace{B = A\{i \leftarrow tail\}}$$

$$\varphi_{\mathsf{sanit}} \quad : \quad \varphi_{\mathsf{norm}} \wedge l_{new} = i + 1$$

Why sanitization? Soon will be clear

Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula
- **STEP 1.** Sanitize

$$\varphi_{\text{sanit}} := \varphi_{\text{norm}} \wedge \bigwedge_{B=A\{l\leftarrow a\} \,\in\, \varphi_{\text{norm}}} (l_{new} = l + 1)$$

- **STEP 2.** Guess an arrangement α of level variables in φ_{sanit}

Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula
- **STEP 1.** Sanitize

$$\varphi_{\mathsf{sanit}} := \varphi_{\mathsf{norm}} \wedge \bigwedge_{B=A\{l\leftarrow a\} \,\in\, \varphi_{\mathsf{norm}}} (l_{new} = l + 1)$$

- **STEP 2.** Guess an arrangement α of level variables in φ_{sanit}

$$\varphi_{\mathsf{sanit}} \quad : \quad i = 0 \quad \wedge \quad \begin{pmatrix} c = heap[head] & \wedge \\ c = mkcell(e, k, A, l) & \wedge \\ l = 3 & \end{pmatrix} \quad \wedge \quad B = A\{i \leftarrow tail\} \quad \wedge \quad l_{new} = i + 1$$

Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula
- **STEP 1.** Sanitize

$$\varphi_{\text{sanit}} := \varphi_{\text{norm}} \wedge \bigwedge_{B=A\{l \leftarrow a\} \in \varphi_{\text{norm}}} (l_{new} = l + 1)$$

- **STEP 2.** Guess an arrangement α of level variables in φ_{sanit}

$$\varphi_{\text{sanit}} \quad : \quad i = 0 \quad \wedge \quad \left(\begin{array}{ll} c = heap[head] & \wedge \\ c = mkcell(e, k, A, l) & \wedge \\ l = 3 & \end{array} \right) \quad \wedge \quad B = A\{i \leftarrow tail\} \quad \wedge \quad l_{new} = i + 1$$

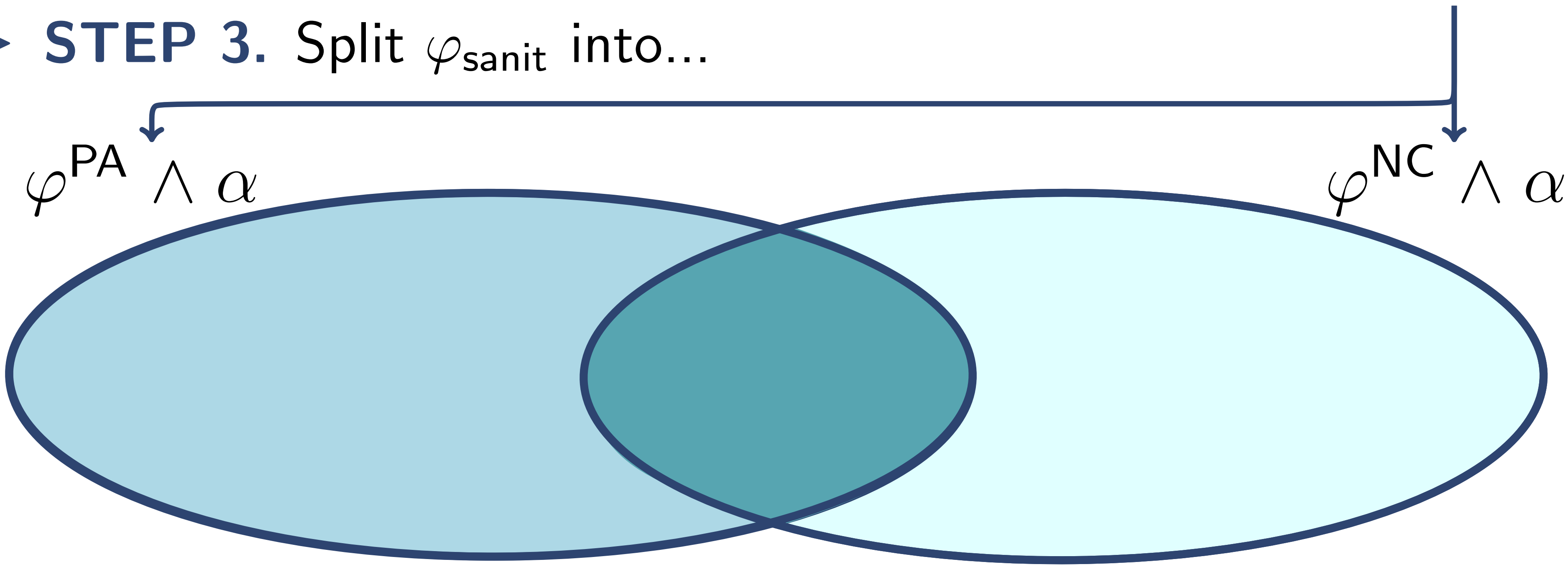
A possible arrangement: $\{i < l_{new} \ , \ i < l \ , \ l_{new} < l\}$

Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula
- **STEP 1.** Sanitize

$$\varphi_{\text{sanit}} := \varphi_{\text{norm}} \wedge \bigwedge_{B = A\{l \leftarrow a\} \,\in\, \varphi_{\text{norm}}} (l_{new} = l + 1)$$

- **STEP 2.** Guess an arrangement α of level variables in φ_{sanit}
- **STEP 3.** Split φ_{sanit} into...

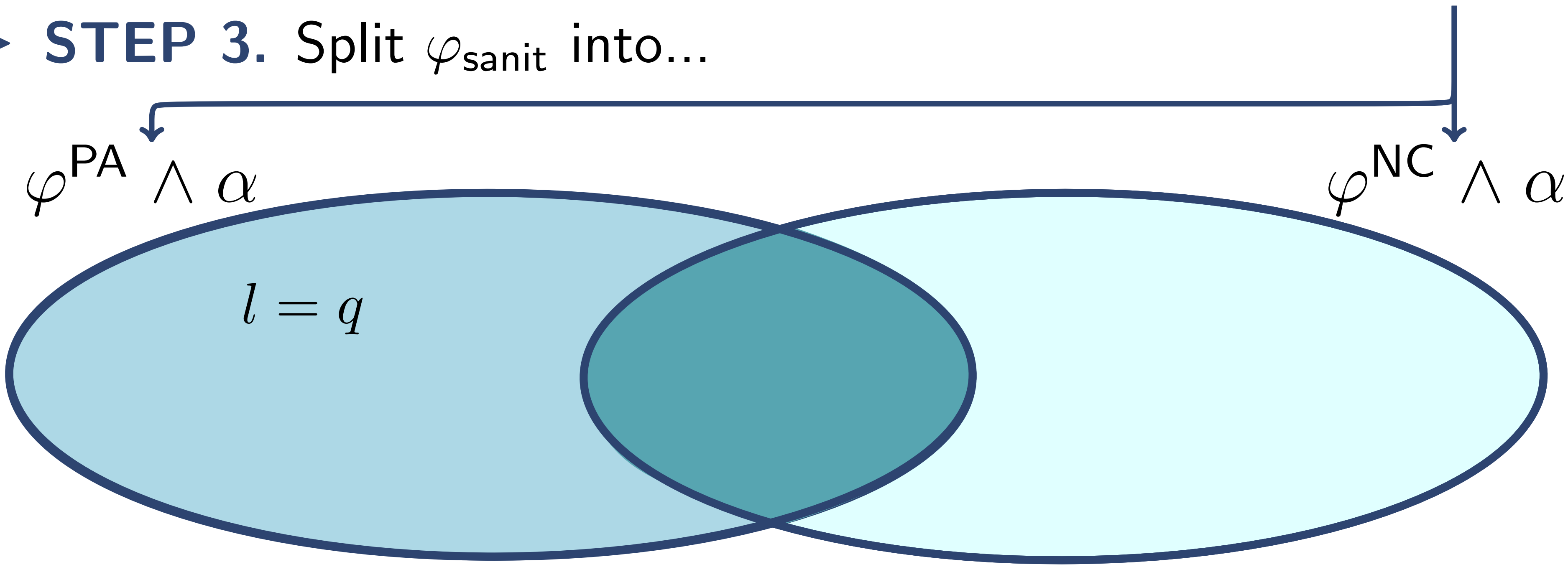


Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula
- **STEP 1.** Sanitize

$$\varphi_{\text{sanit}} := \varphi_{\text{norm}} \wedge \bigwedge_{B=A\{l\leftarrow a\} \in \varphi_{\text{norm}}} (l_{new} = l + 1)$$

- **STEP 2.** Guess an arrangement α of level variables in φ_{sanit}
- **STEP 3.** Split φ_{sanit} into...

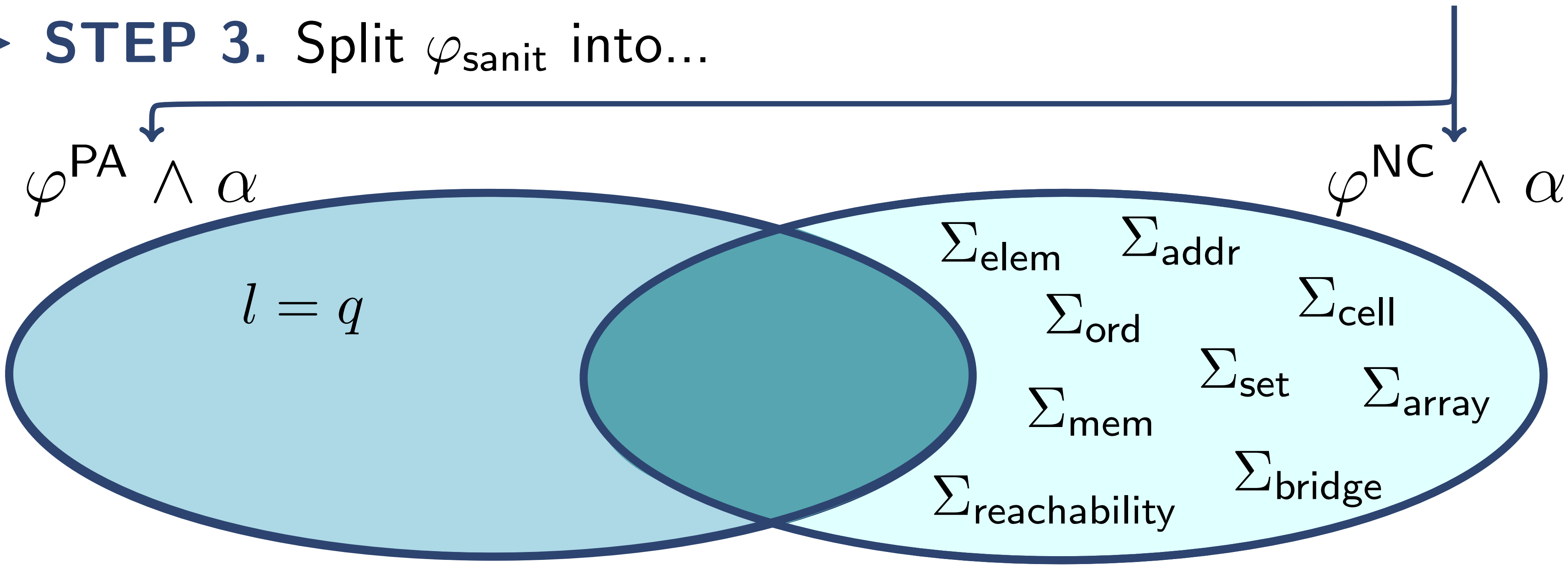


Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula
- **STEP 1.** Sanitize

$$\varphi_{\text{sanit}} := \varphi_{\text{norm}} \wedge \bigwedge_{B=A\{l \leftarrow a\} \in \varphi_{\text{norm}}} (l_{new} = l + 1)$$

- **STEP 2.** Guess an arrangement α of level variables in φ_{sanit}
- **STEP 3.** Split φ_{sanit} into...

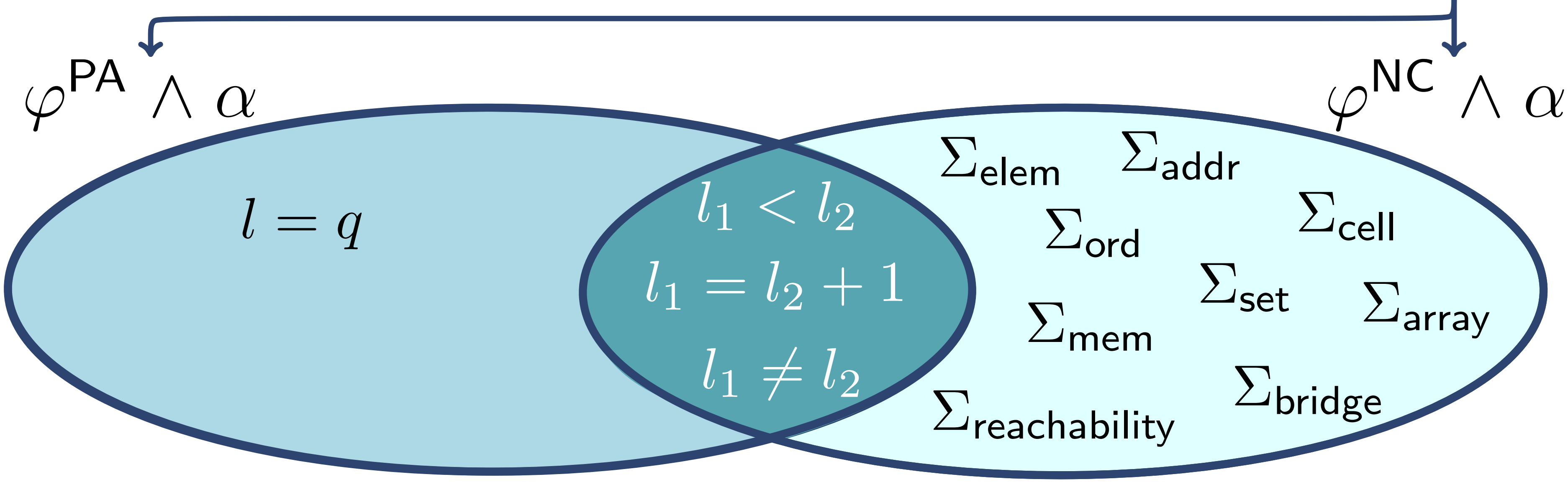


Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula
- **STEP 1.** Sanitize

$$\varphi_{\mathsf{sanit}} := \varphi_{\mathsf{norm}} \wedge \bigwedge_{B=A\{l \leftarrow a\} \,\in\, \varphi_{\mathsf{norm}}} (l_{new} = l + 1)$$

- **STEP 2.** Guess an arrangement α of level variables in φ_{sanit}
- **STEP 3.** Split φ_{sanit} into...

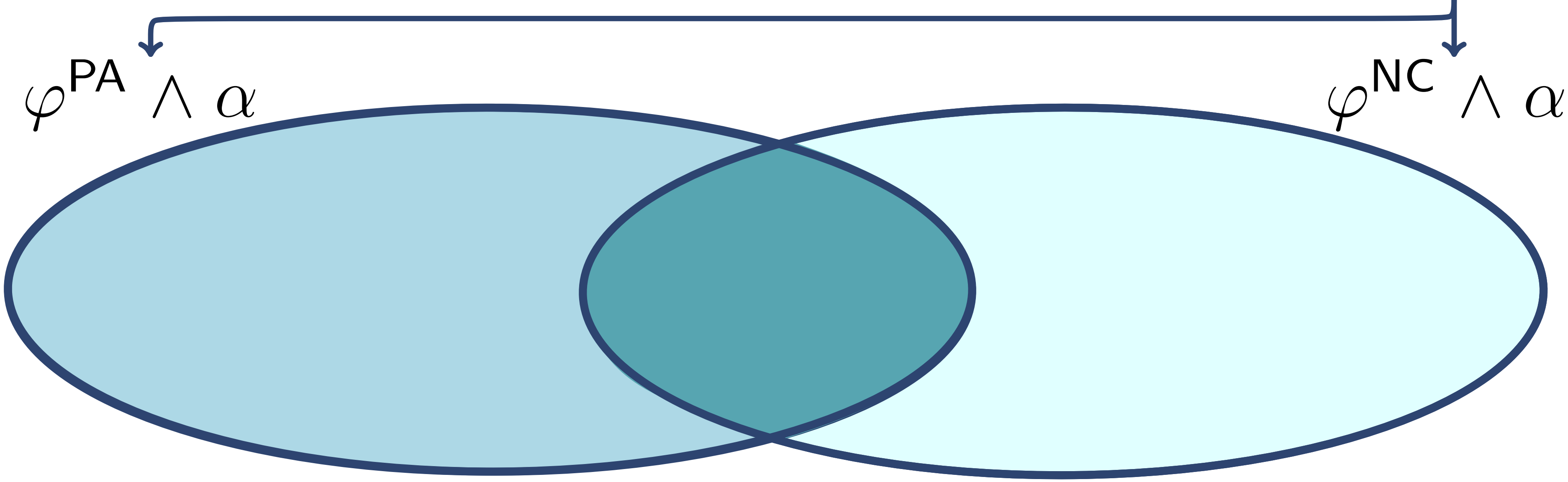


Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula
- **STEP 1.** Sanitize

$$\varphi_{\mathsf{sanit}} := \varphi_{\mathsf{norm}} \wedge \bigwedge_{B=A\{l\leftarrow a\} \in \varphi_{\mathsf{norm}}} (l_{new} = l + 1)$$

- **STEP 2.** Guess an arrangement α of level variables in φ_{sanit}
- **STEP 3.** Split φ_{sanit} into...



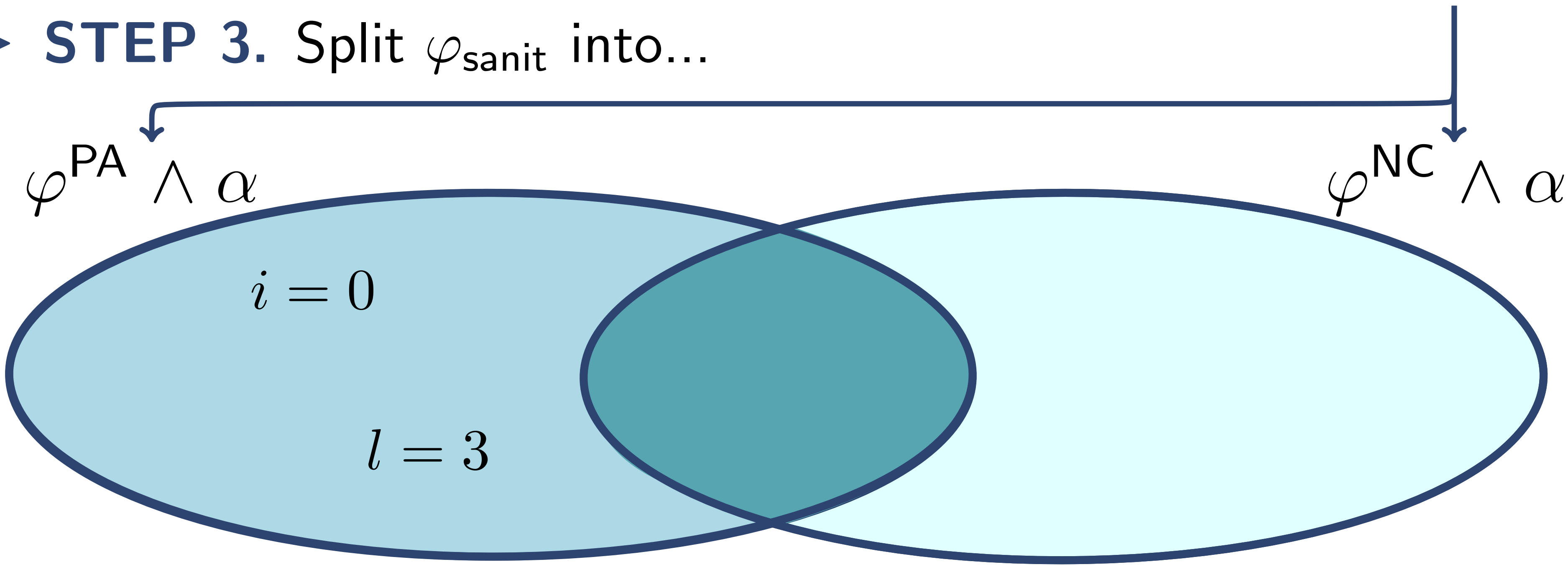
$$\varphi_{\mathsf{sanit}} \quad : \quad i = 0 \quad \wedge \quad \begin{pmatrix} c = heap[head] & \wedge \\ c = mkcell(e, k, A, l) & \wedge \\ l = 3 & \end{pmatrix} \quad \wedge \quad B = A\{i \leftarrow tail\} \quad \wedge \quad l_{new} = i + 1$$

Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula
- **STEP 1.** Sanitize

$$\varphi_{\mathsf{sanit}} := \varphi_{\mathsf{norm}} \wedge \bigwedge_{B=A\{l\leftarrow a\} \,\in\, \varphi_{\mathsf{norm}}} (l_{new} = l + 1)$$

- **STEP 2.** Guess an arrangement α of level variables in φ_{sanit}
- **STEP 3.** Split φ_{sanit} into...



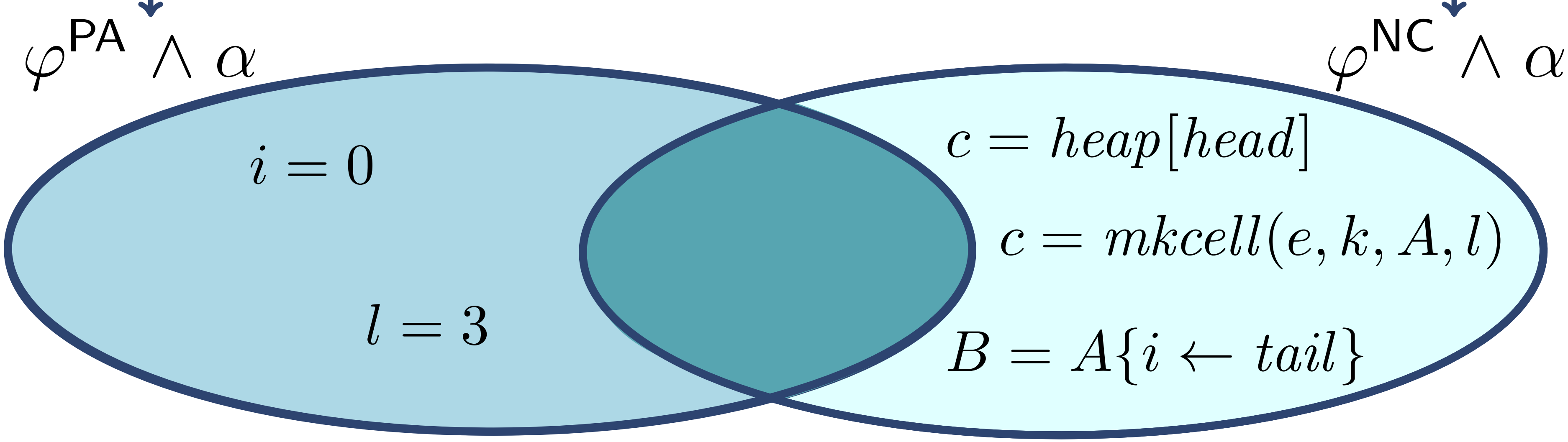
$$\varphi_{\mathsf{sanit}} \quad : \quad \boxed{i = 0} \quad \wedge \quad \begin{pmatrix} c = heap[head] & \wedge \\ c = mkcell(e, k, A, l) & \wedge \\ \boxed{l = 3} & \end{pmatrix} \quad \wedge \quad B = A\{i \leftarrow tail\} \quad \wedge \quad l_{new} = i + 1$$

Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula
- **STEP 1.** Sanitize

$$\varphi_{\text{sanit}} := \varphi_{\text{norm}} \wedge \bigwedge_{B=A\{l \leftarrow a\} \in \varphi_{\text{norm}}} (l_{new} = l + 1)$$

- **STEP 2.** Guess an arrangement α of level variables in φ_{sanit}
- **STEP 3.** Split φ_{sanit} into...



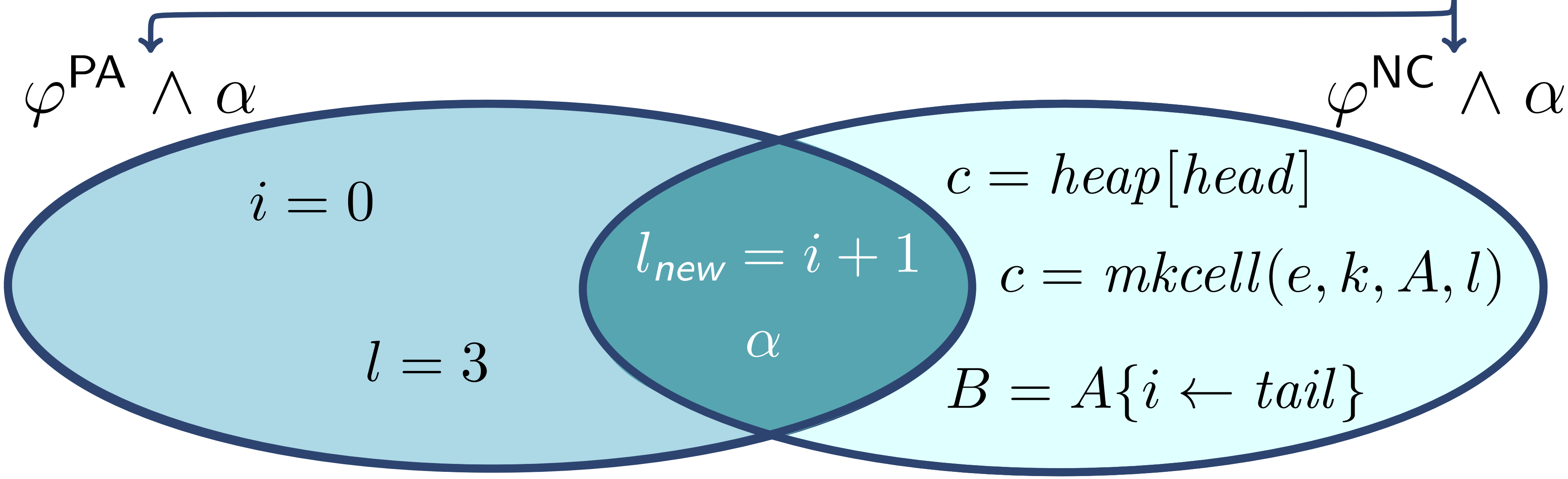
$$\varphi_{\text{sanit}} : \; i = 0 \;\wedge\; \begin{pmatrix} c = heap[head] \;\wedge \\ c = mkcell(e,k,A,l) \;\wedge \\ l = 3 \end{pmatrix} \;\wedge\; B = A\{i \leftarrow tail\} \;\wedge\; l_{new} = i + 1$$

Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula
- **STEP 1.** Sanitize

$$\varphi_{\text{sanit}} := \varphi_{\text{norm}} \wedge \bigwedge_{B=A\{l \leftarrow a\} \in \varphi_{\text{norm}}} (l_{new} = l + 1)$$

- **STEP 2.** Guess an arrangement α of level variables in φ_{sanit}
- **STEP 3.** Split φ_{sanit} into...



$$\varphi_{\text{sanit}} \; : \; i = 0 \;\wedge\; \left(\begin{array}{l} c = heap[head] \;\wedge \\ c = mkcell(e,k,A,l) \;\wedge \\ l = 3 \end{array} \right) \;\wedge\; B = A\{i \leftarrow tail\} \;\wedge\; l_{new} = i + 1$$

Decision Procedure for TSL

- Let φ_{norm} be a normalized TSL formula
- **STEP 1.** Sanitize

$$\varphi_{\text{sanit}} := \varphi_{\text{norm}} \wedge \bigwedge_{B=A\{l\leftarrow a\} \in \varphi_{\text{norm}}} (l_{new} = l+1)$$

- **STEP 2.** Guess an arrangement α of level variables in φ_{sanit}
- **STEP 3.** Split φ_{sanit} into...

$$\varphi^{\text{PA}} \wedge \alpha \qquad\qquad \varphi^{\text{NC}} \wedge \alpha$$

- **STEP 4.** Check satisfiability of $(\varphi^{\text{PA}} \wedge \alpha)$ and $(\varphi^{\text{NC}} \wedge \alpha)$

Decision Procedure for TSL: Correctness

- Let φ be a TSL formula

Decision Procedure for TSL: Correctness

- Let φ be a TSL formula

φ_{norm}

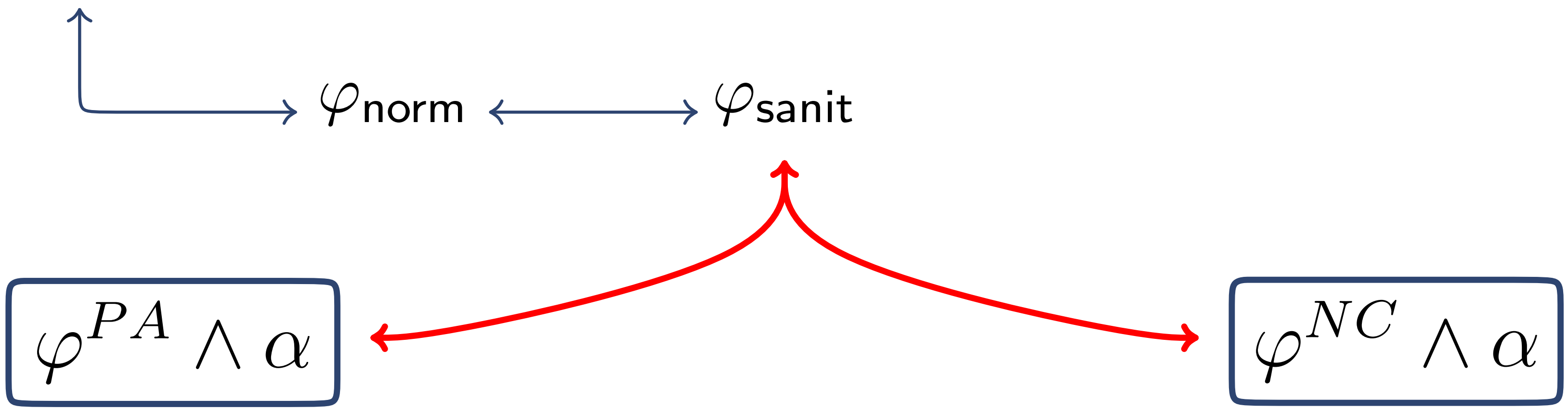
Decision Procedure for TSL: Correctness

- Let φ be a TSL formula

$$\varphi \to \varphi_{\text{norm}} \longleftrightarrow \varphi_{\text{sanit}}$$

Decision Procedure for TSL: Correctness

- Let φ be a TSL formula



Theorem:

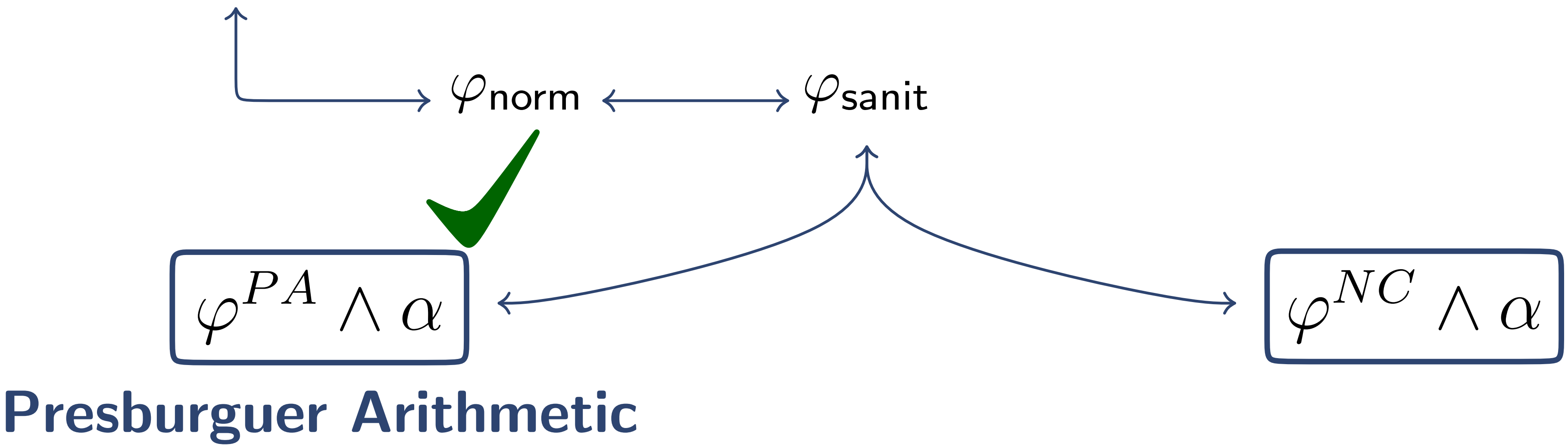
φ : TSL formula is satisfiable

iff

for some arrangement α, both
$(\varphi^{PA} \wedge \alpha)$ and $(\varphi^{NC} \wedge \alpha)$ are satisfiable

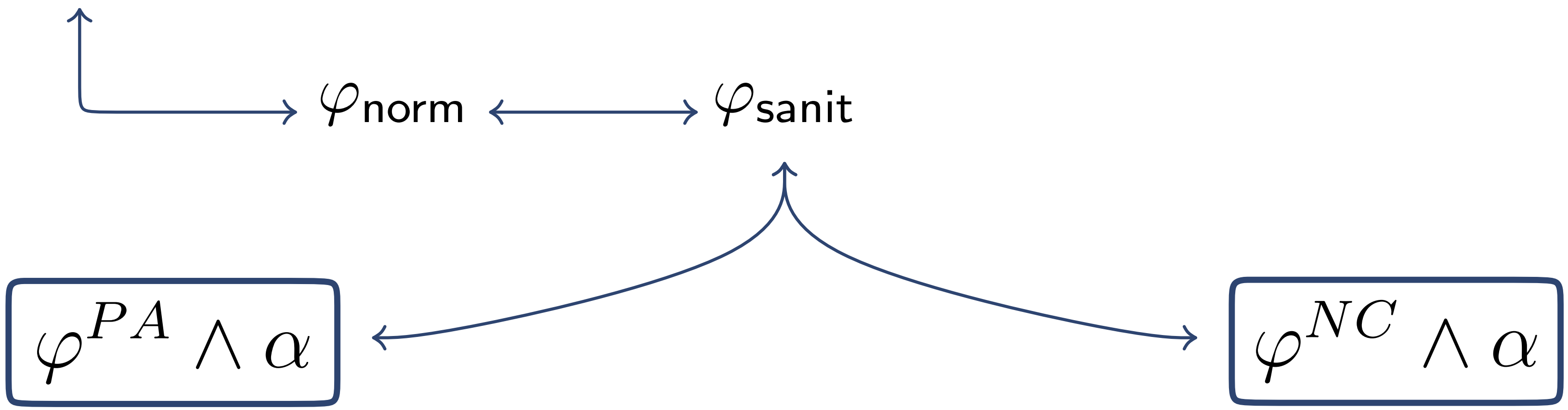
Decision Procedure for TSL: Correctness

- Let φ be a TSL formula



Decision Procedure for TSL: Correctness

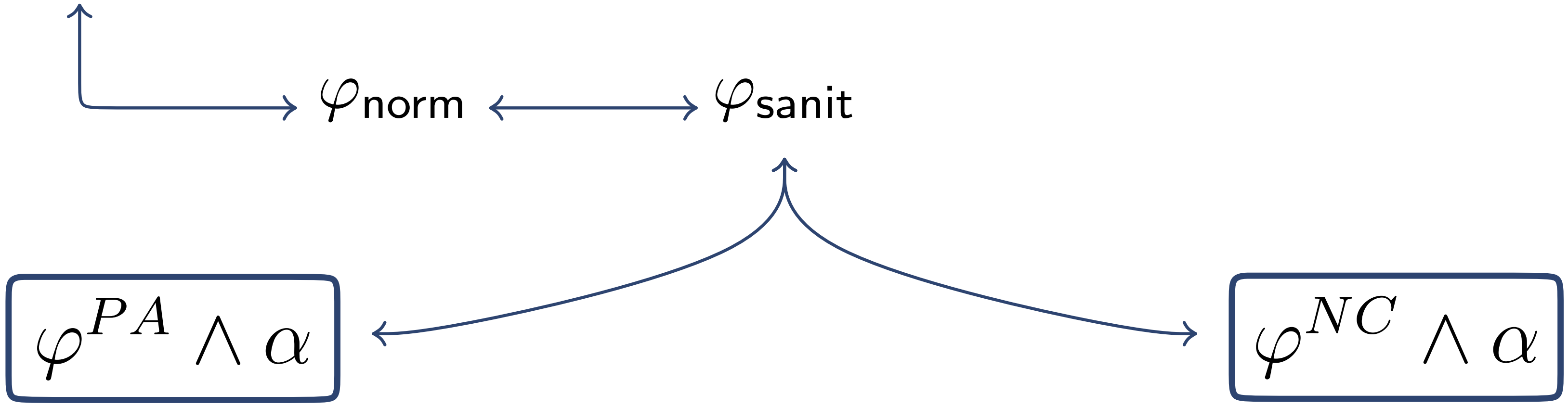
- Let φ be a TSL formula



- **Gapless model**: we stay only with **interesting levels**

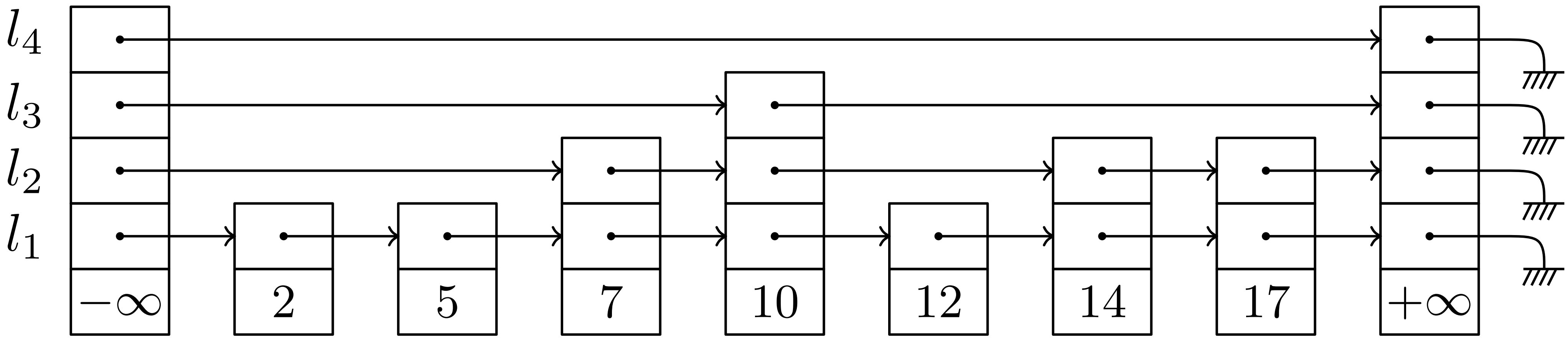
Decision Procedure for TSL: Correctness

- Let φ be a TSL formula



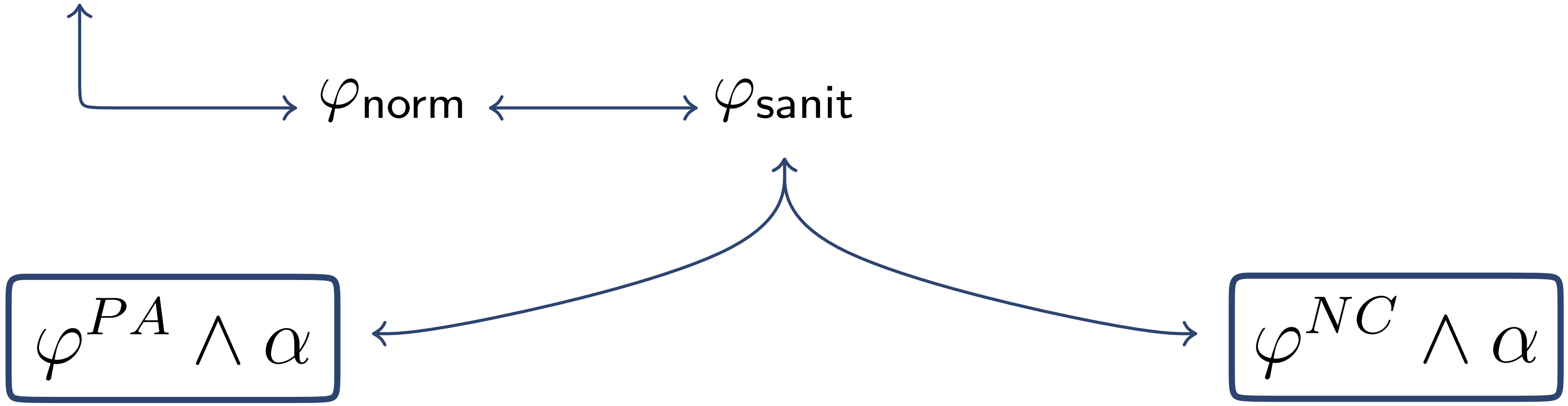
$$V_{\mathsf{level}}(\varphi^{NC} \wedge \alpha) = \{l_1, l_3\}$$

- **Gapless model**: we stay only with **interesting levels**



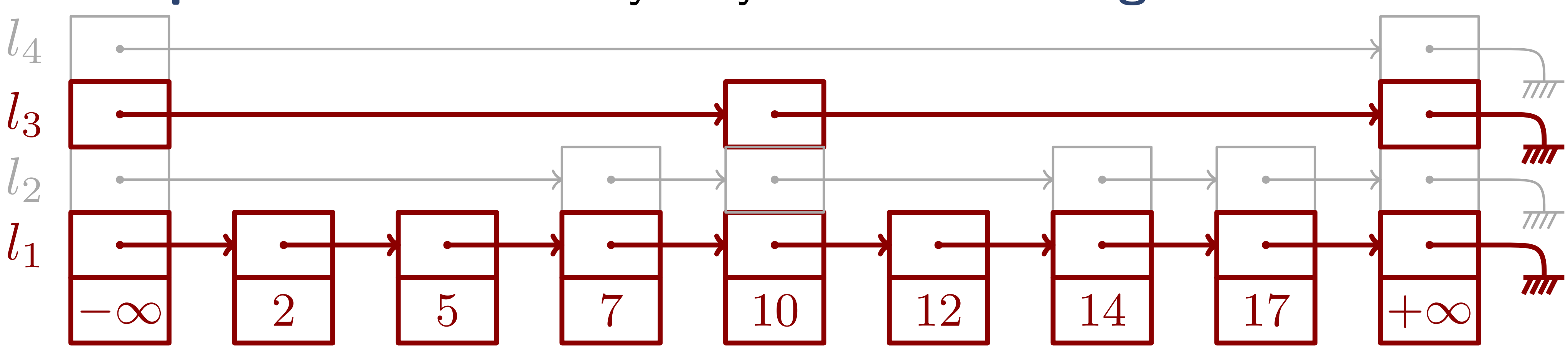
Decision Procedure for TSL: Correctness

- Let φ be a TSL formula



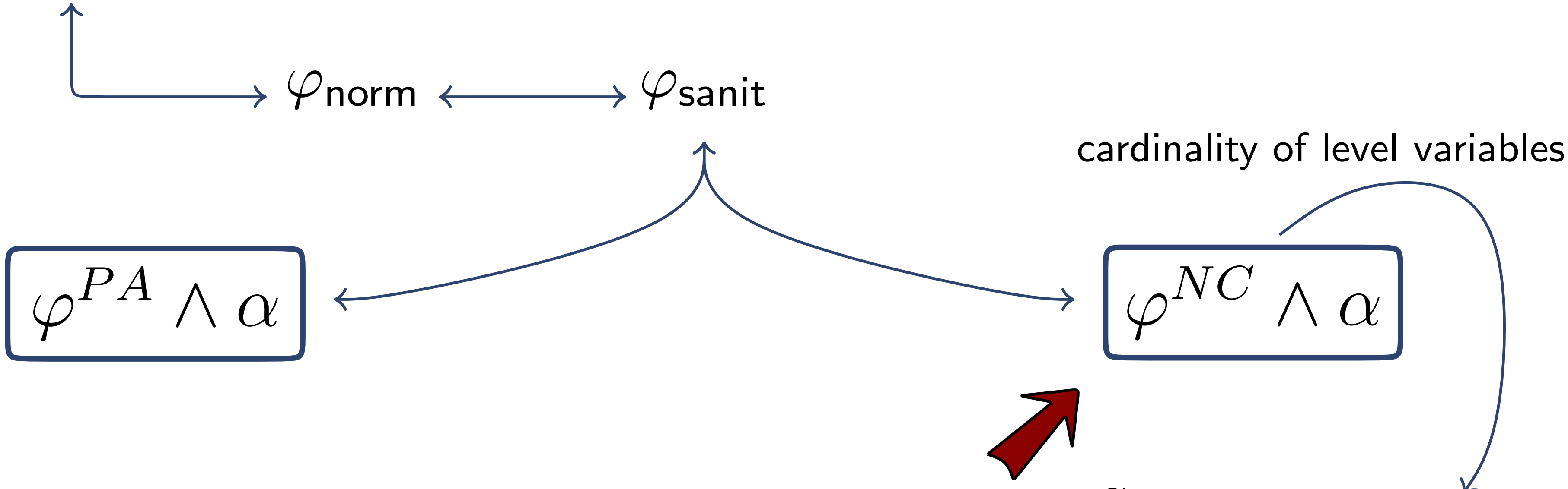
$$V_{\text{level}}(\varphi^{NC} \wedge \alpha) = \{l_1, l_3\}$$

- **Gapless model**: we stay only with **interesting levels**



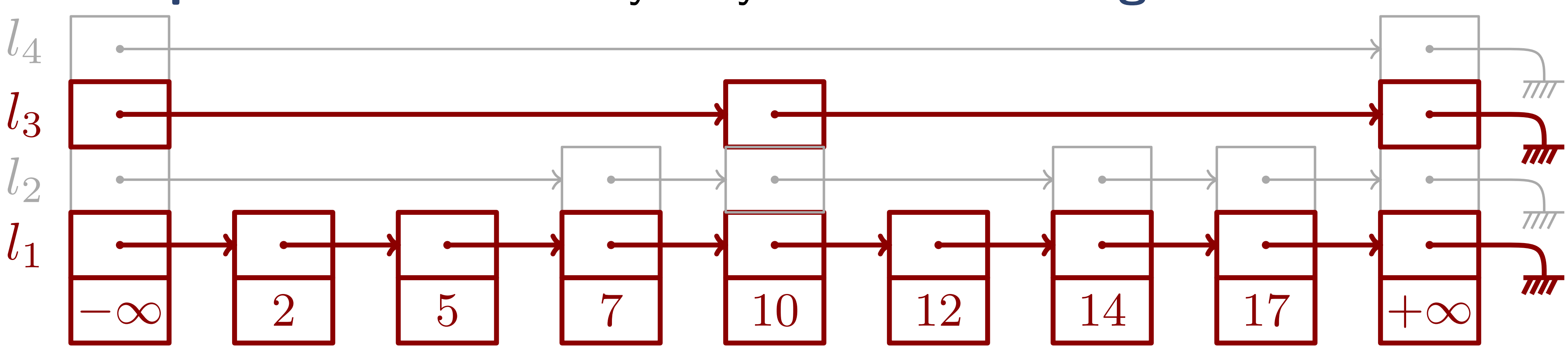
Decision Procedure for TSL: Correctness

- Let φ be a TSL formula



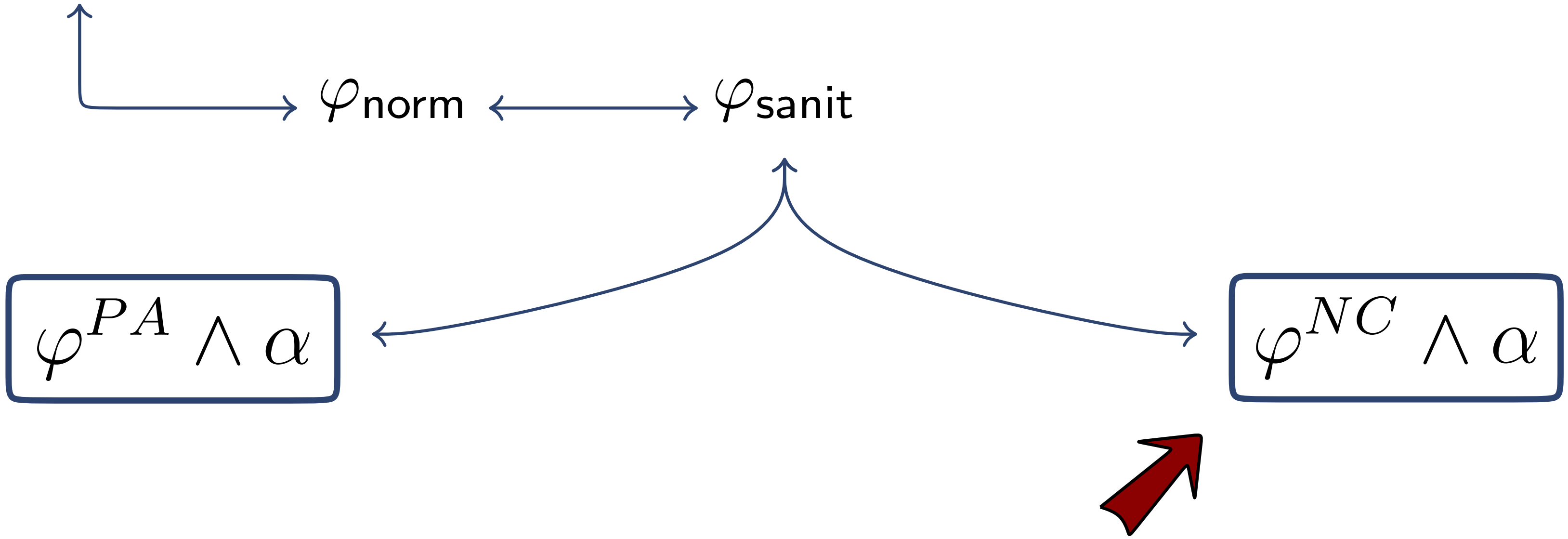
We reduce the formula to $\ulcorner \varphi^{NC} \wedge \alpha \urcorner : \text{TSL}_K$

- **Gapless model**: we stay only with **interesting levels**



Decision Procedure for TSL: Correctness

- Let φ be a TSL formula



We reduce the formula to $\ulcorner \varphi^{NC} \land \alpha \urcorner : \mathsf{TSL}_{\mathsf{K}}$

Theorem:

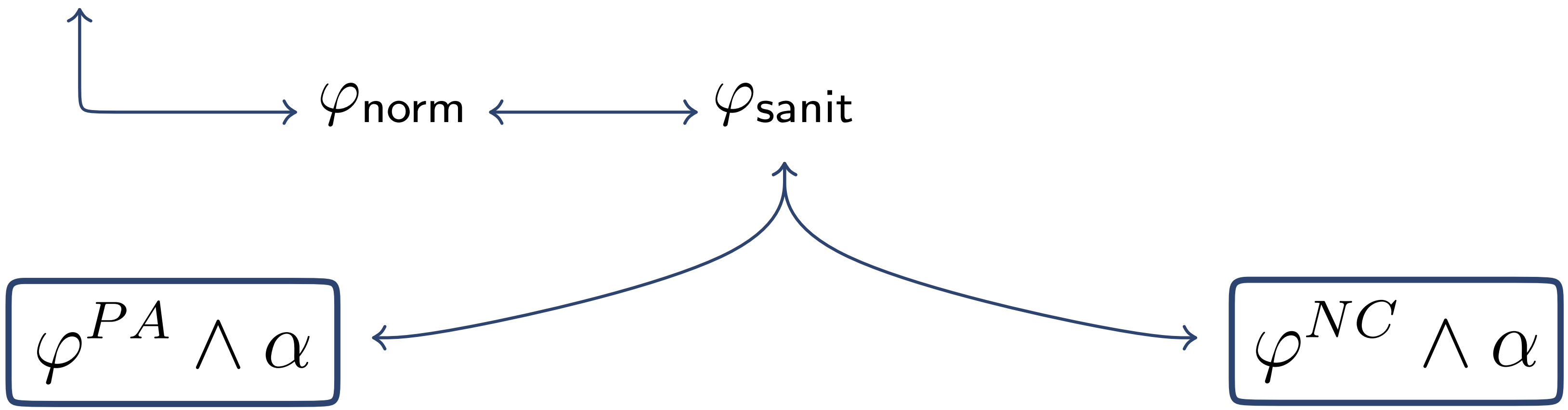
ψ a sanitized TSL formula without constant levels is satisfiable

iff

$\ulcorner \psi \urcorner : \mathsf{TSL}_{\mathsf{K}}$ is satisfiable

Decision Procedure for TSL: Correctness

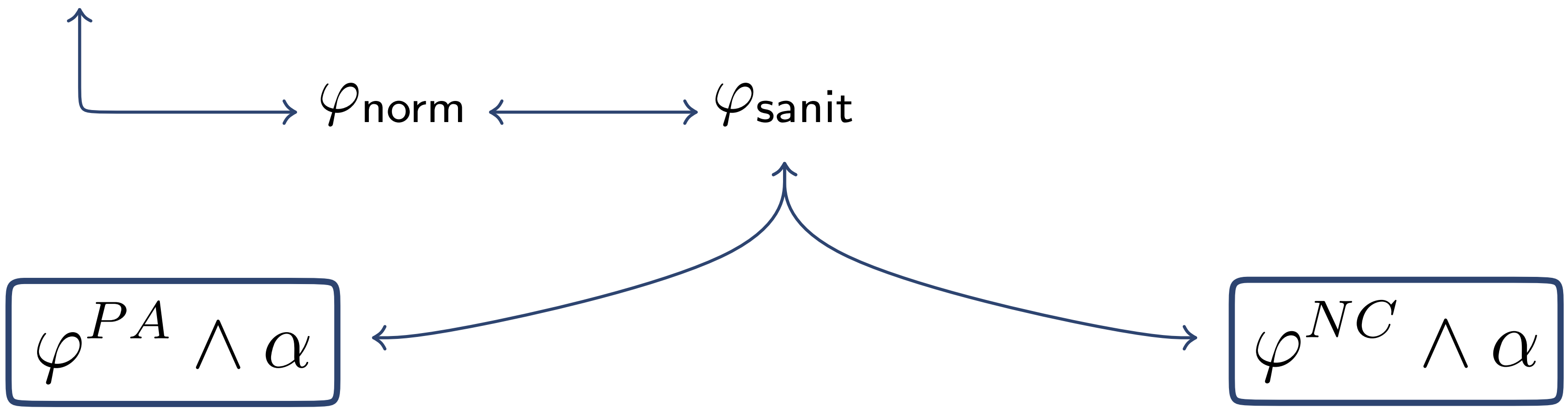
- Let φ be a TSL formula



- Reduction TSL $\longrightarrow$ $\mathsf{TSL}_{\mathsf{K}}$

Decision Procedure for TSL: Correctness

- Let φ be a TSL formula



- Reduction TSL $\longrightarrow$ TSL$_K$

$$\ulcorner c = mkcell(e, k, A, l) \urcorner \qquad c = (e, k, v_{A[0]}, \ldots, v_{A[K-1]})$$

$$v_{A[l]} = A(l)$$

Decision Procedure for TSL: Correctness

- Let φ be a TSL formula

$\varphi_{\text{norm}} \longleftrightarrow \varphi_{\text{sanit}}$

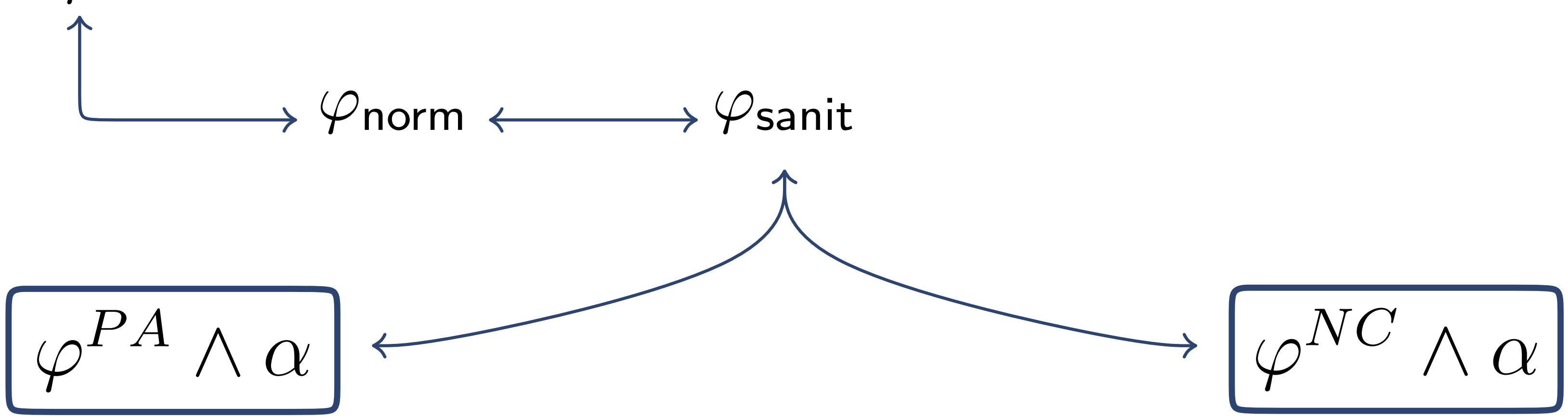
$\boxed{\varphi^{PA} \wedge \alpha}$ $\qquad$ $\boxed{\varphi^{NC} \wedge \alpha}$

- Reduction $\quad \mathsf{TSL} \longrightarrow \mathsf{TSL}_{\mathsf{K}}$

$\ulcorner c = mkcell(e, k, A, l) \urcorner$	$c = (e, k, v_{A[0]}, \ldots, v_{A[K-1]})$
$\ulcorner a = A[l] \urcorner$	$\bigwedge_{i=0 \ldots \mathsf{K}-1} l = i \rightarrow a = v_{A[i]}$

Decision Procedure for TSL: Correctness

- Let φ be a TSL formula

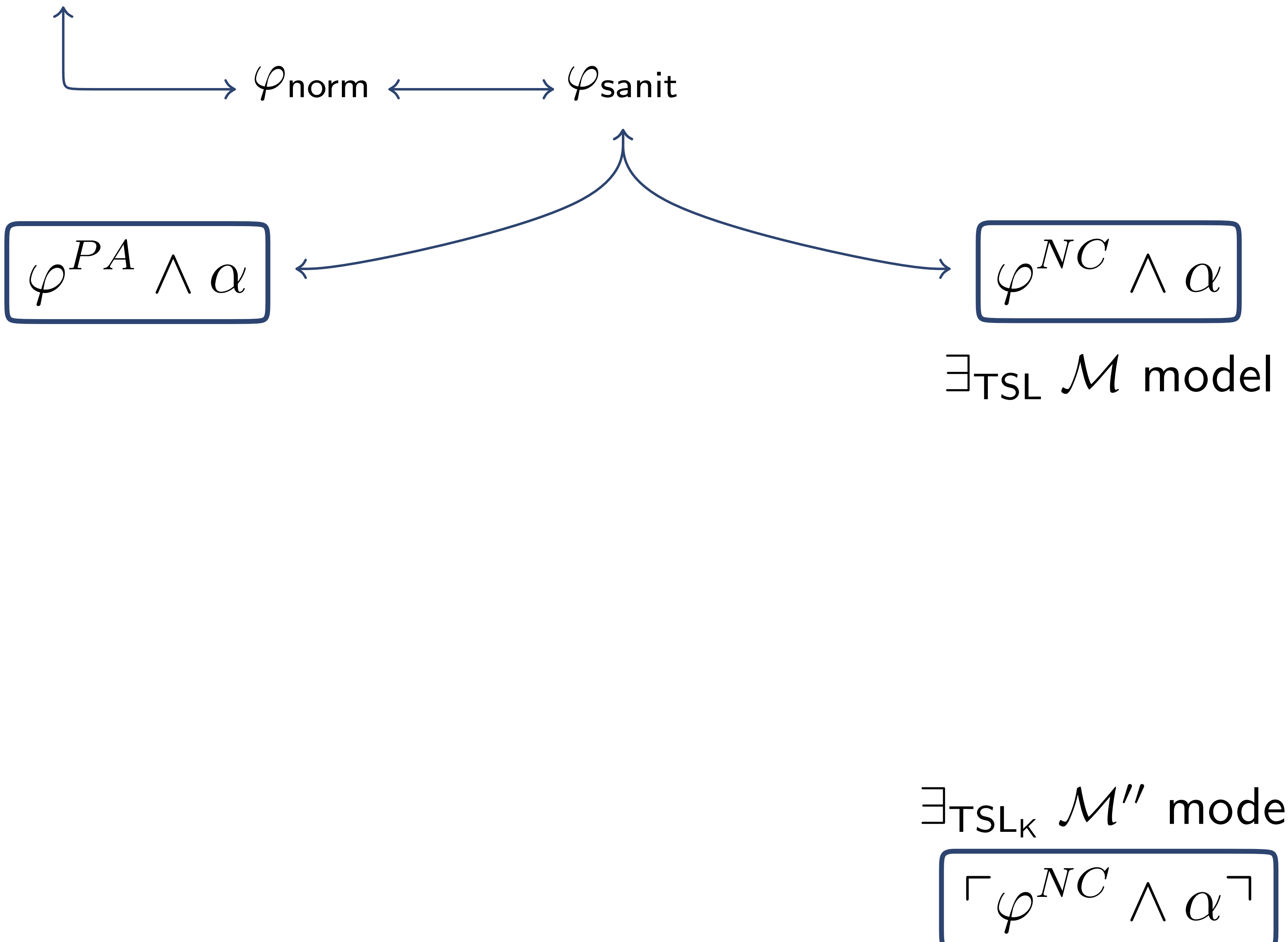


- Reduction

TSL	$\longrightarrow$ TSL$_K$
$\ulcorner c = mkcell(e, k, A, l) \urcorner$	$c = (e, k, v_{A[0]}, \ldots, v_{A[K-1]})$
$\ulcorner a = A[l] \urcorner$	$\bigwedge_{i=0 \ldots K-1} l = i \rightarrow a = v_{A[i]}$
$\ulcorner B = A\{l \leftarrow a\} \urcorner$	$\left(\bigwedge_{i=0 \ldots K-1} l = i \rightarrow a = v_{B[i]} \right) \wedge \left(\bigwedge_{j=0 \ldots K-1} l \neq j \rightarrow v_{B[j]} = v_{A[j]} \right)$

Decision Procedure for TSL: Correctness

- Let φ be a TSL formula



Decision Procedure for TSL: Correctness

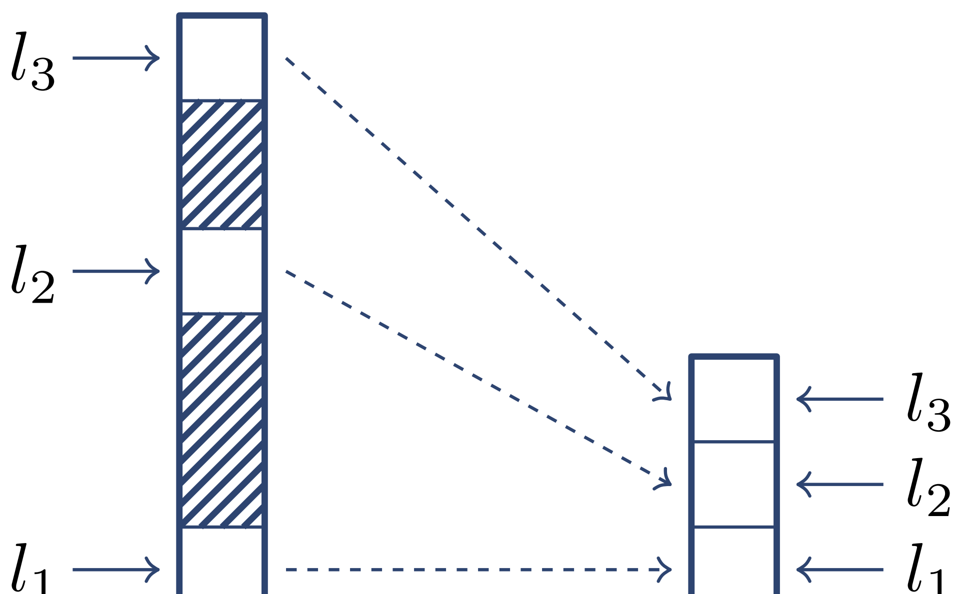
- Let φ be a TSL formula

φ_{norm} $\quad$ φ_{sanit}

$\varphi^{PA} \wedge \alpha$ $\quad$ $\varphi^{NC} \wedge \alpha$

$\exists_{\text{TSL}}\ \mathcal{M}$ model

$\exists_{\text{TSL}(gapless)}\ \mathcal{M}'$ model

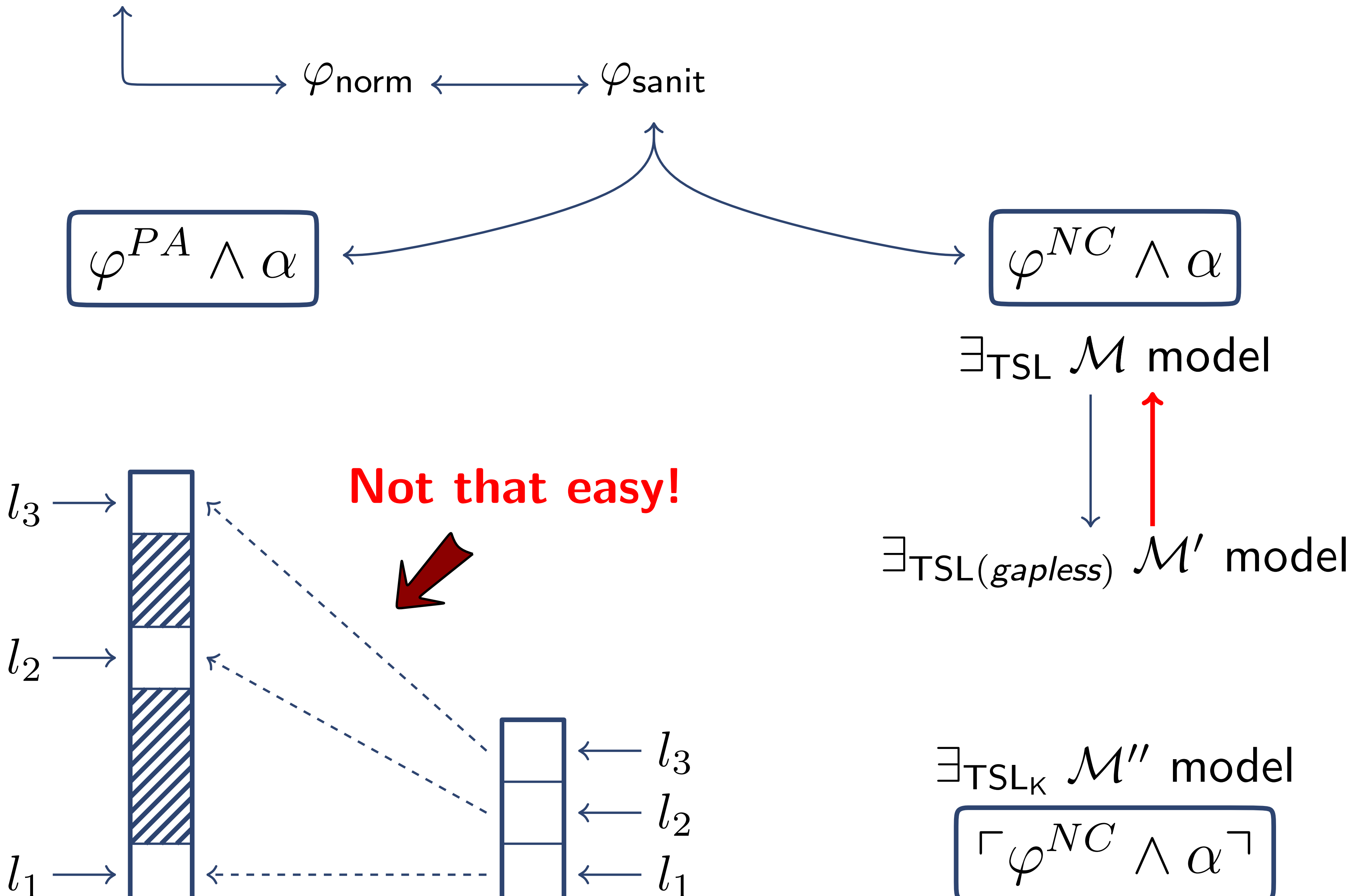


$\exists_{\text{TSL}_{\text{K}}}\ \mathcal{M}''$ model

$\ulcorner \varphi^{NC} \wedge \alpha \urcorner$

Decision Procedure for TSL: Correctness

- Let φ be a TSL formula



Decision Procedure for TSL: Correctness

- Let φ be a TSL formula

$\varphi_{\text{norm}} \longleftrightarrow \varphi_{\text{sanit}}$

$\varphi^{PA} \wedge \alpha$

$\varphi^{NC} \wedge \alpha$

$\exists_{\mathsf{TSL}}\ \mathcal{M}$ model

$B = A\{l_1 \leftarrow a\}$

$\exists_{\mathsf{TSL}(gapless)}\ \mathcal{M}'$ model

$\exists_{\mathsf{TSL_K}}\ \mathcal{M}''$ model

$\ulcorner \varphi^{NC} \wedge \alpha \urcorner$

- Let φ be a TSL formula

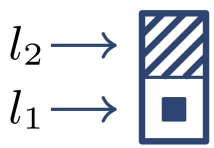
$\varphi_{\mathsf{norm}} \longleftrightarrow \varphi_{\mathsf{sanit}}$

$\boxed{\varphi^{PA} \wedge \alpha}$ $\qquad$ $\boxed{\varphi^{NC} \wedge \alpha}$

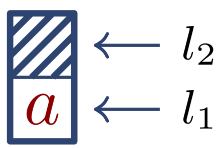
$\exists_{\mathsf{TSL}}\ \mathcal{M}$ model

$B = A\{l_1 \leftarrow a\}$

$\exists_{\mathsf{TSL}(gapless)}\ \mathcal{M}'$ model



A



B

$\exists_{\mathsf{TSL}_{\mathsf{K}}}\ \mathcal{M}''$ model

$\boxed{\ulcorner \varphi^{NC} \wedge \alpha \urcorner}$

Decision Procedure for TSL: Correctness

- Let φ be a TSL formula

$\varphi_{\text{norm}} \longleftrightarrow \varphi_{\text{sanit}}$

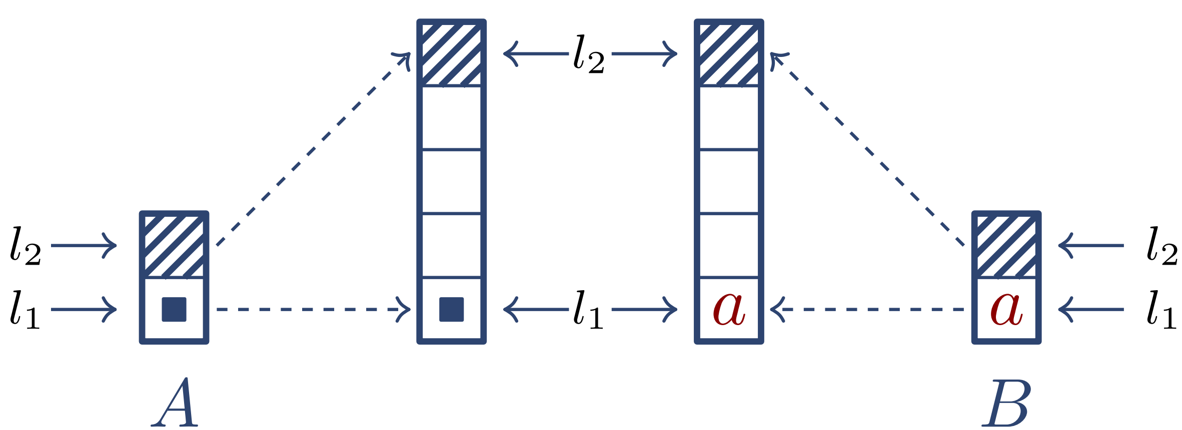
$\varphi^{PA} \wedge \alpha$

$\varphi^{NC} \wedge \alpha$

$\exists_{\text{TSL}}\ \mathcal{M}$ model

$B = A\{l_1 \leftarrow a\}$

$\exists_{\text{TSL}(gapless)}\ \mathcal{M}'$ model

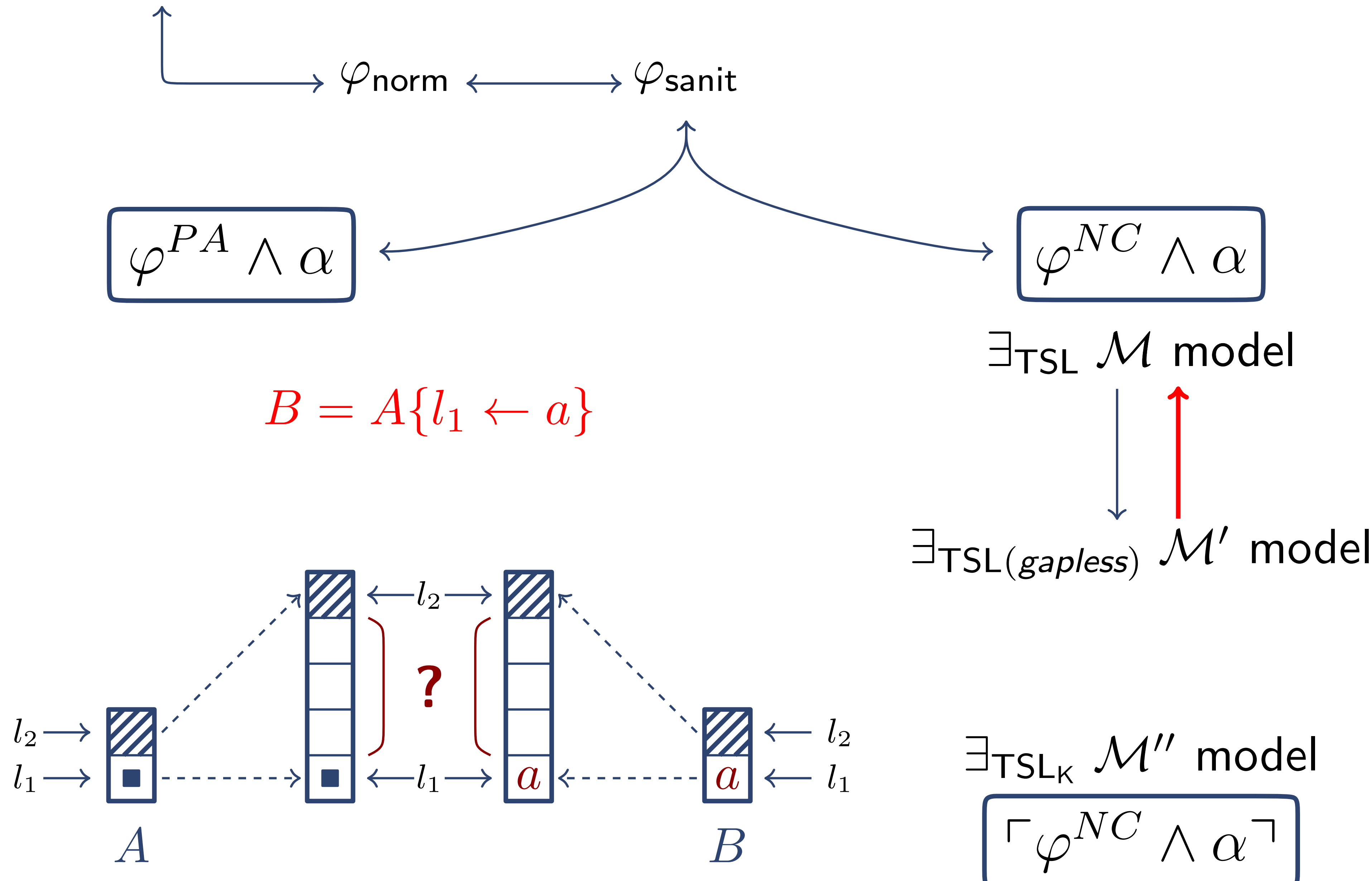


$\exists_{\text{TSL}_K}\ \mathcal{M}''$ model

$\ulcorner \varphi^{NC} \wedge \alpha \urcorner$

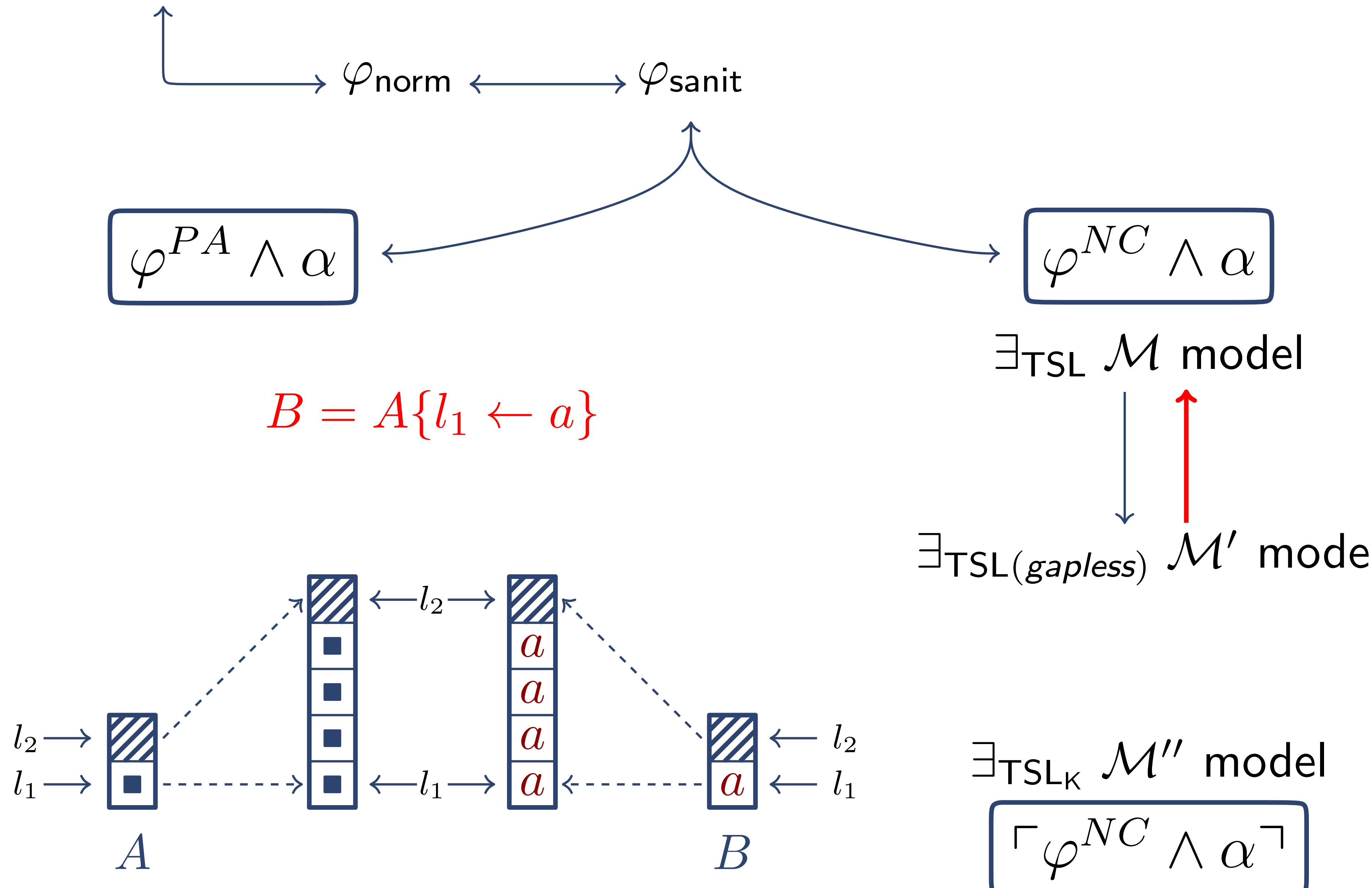
Decision Procedure for TSL: Correctness

- Let φ be a TSL formula



Decision Procedure for TSL: Correctness

- Let φ be a TSL formula



Decision Procedure for TSL: Correctness

- Let φ be a TSL formula

$\varphi_{\text{norm}} \longleftrightarrow \varphi_{\text{sanit}}$

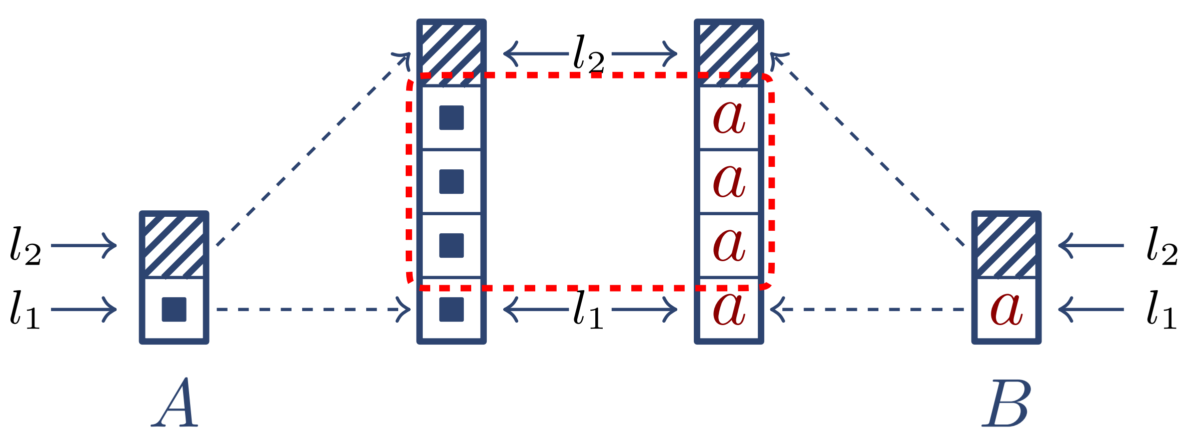
$\boxed{\varphi^{PA} \wedge \alpha}$

$\boxed{\varphi^{NC} \wedge \alpha}$

$\exists_{\text{TSL}}\ \mathcal{M}$ model

~~$B = A\{l_1 \leftarrow a\}$~~

$\exists_{\text{TSL}(\textit{gapless})}\ \mathcal{M}'$ model



$\exists_{\text{TSL}_{\text{K}}}\ \mathcal{M}''$ model

$\boxed{\ulcorner \varphi^{NC} \wedge \alpha \urcorner}$

Decision Procedure for TSL: Correctness

- Let φ be a TSL formula

$\varphi_{\text{norm}} \longleftrightarrow \varphi_{\text{sanit}}$

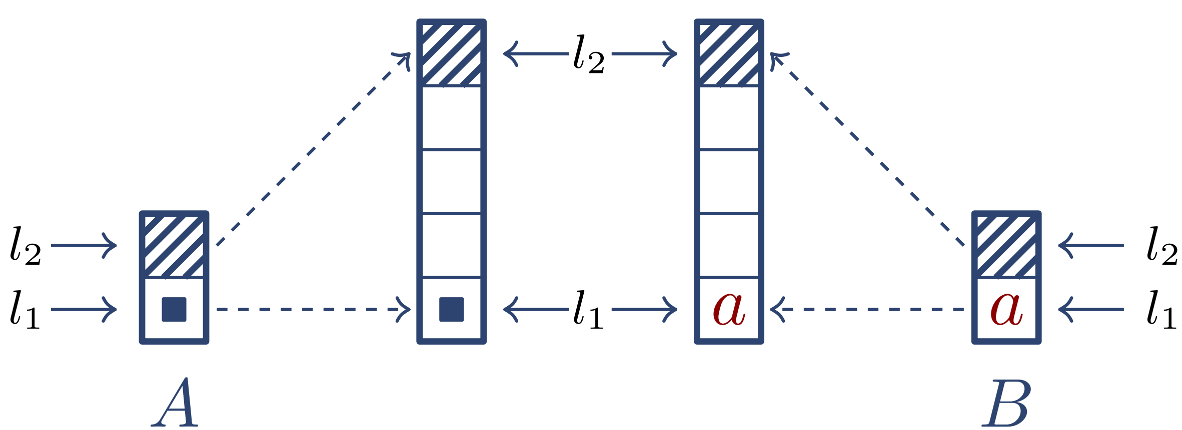
$\varphi^{PA} \wedge \alpha$

$\varphi^{NC} \wedge \alpha$

$\exists_{\text{TSL}}\ \mathcal{M}$ model

$B = A\{l_1 \leftarrow a\}$

$\exists_{\text{TSL}(gapless)}\ \mathcal{M}'$ model



$\exists_{\text{TSL}_K}\ \mathcal{M}''$ model

$\ulcorner \varphi^{NC} \wedge \alpha \urcorner$

Decision Procedure for TSL: Correctness

- Let φ be a TSL formula

$\varphi_{\mathsf{norm}} \longleftrightarrow \varphi_{\mathsf{sanit}}$

$\varphi^{PA} \wedge \alpha$

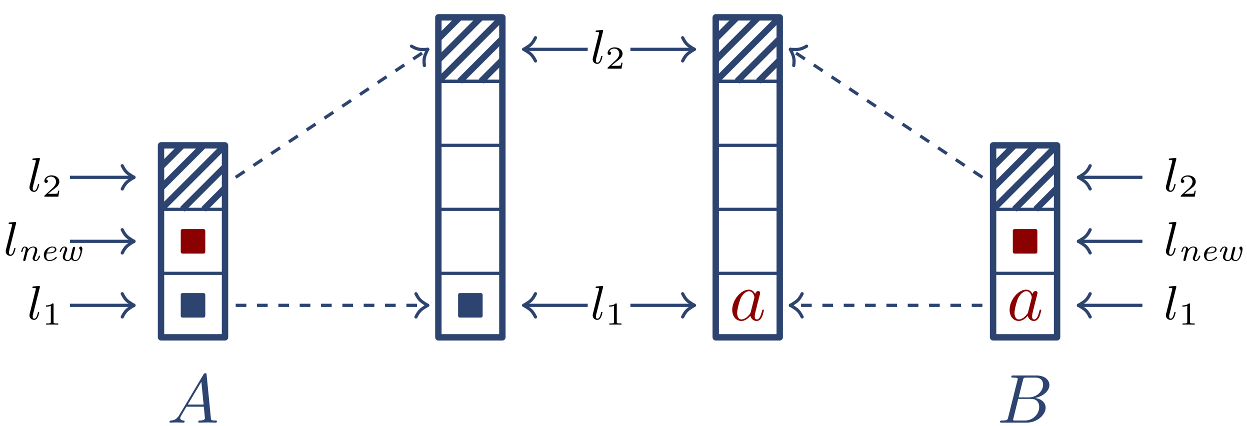
$\varphi^{NC} \wedge \alpha$

$\exists_{\mathsf{TSL}}\ \mathcal{M}$ model

$\exists_{\mathsf{TSL}(\mathit{gapless})}\ \mathcal{M}'$ model

$B = A\{l_1 \leftarrow a\}$

Add $l_{new} = l_1 + 1$



$\exists_{\mathsf{TSL_K}}\ \mathcal{M}''$ model

$\ulcorner \varphi^{NC} \wedge \alpha \urcorner$

Decision Procedure for TSL: Correctness

- Let φ be a TSL formula

$\varphi_{\textsf{norm}} \longleftrightarrow \varphi_{\textsf{sanit}}$

$\boxed{\varphi^{PA} \wedge \alpha}$

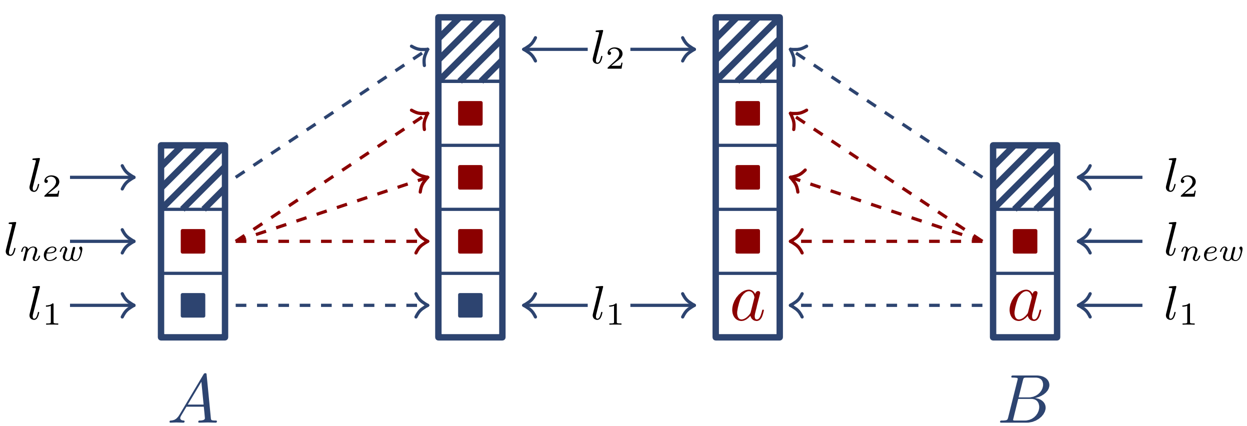
$\boxed{\varphi^{NC} \wedge \alpha}$

$\exists_{\textsf{TSL}}\ \mathcal{M}$ model

$\exists_{\textsf{TSL}(gapless)}\ \mathcal{M}'$ model

$B = A\{l_1 \leftarrow a\}$

Add $l_{new} = l_1 + 1$



$\exists_{\textsf{TSL}_\textsf{K}}\ \mathcal{M}''$ model

$\boxed{\ulcorner \varphi^{NC} \wedge \alpha \urcorner}$

Decision Procedure for TSL: Correctness

- Let φ be a TSL formula

φ_{norm} $\longleftrightarrow$ φ_{sanit}

$\boxed{\varphi^{PA} \wedge \alpha}$

$\boxed{\varphi^{NC} \wedge \alpha}$

$\exists_{\mathsf{TSL}}\ \mathcal{M}$ model

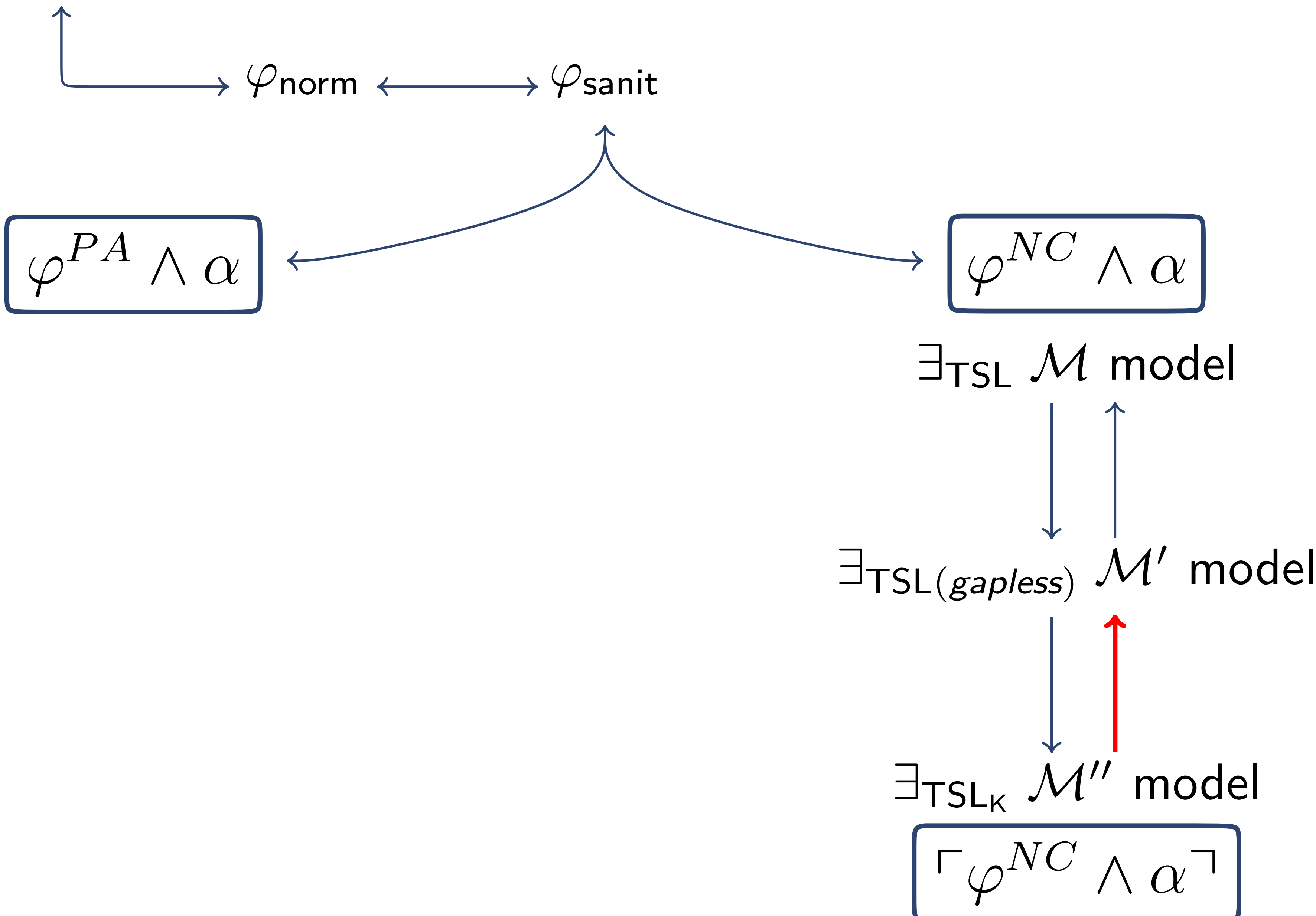
$\exists_{\mathsf{TSL}(gapless)}\ \mathcal{M}'$ model

$\exists_{\mathsf{TSL_K}}\ \mathcal{M}''$ model

$\boxed{\ulcorner \varphi^{NC} \wedge \alpha \urcorner}$

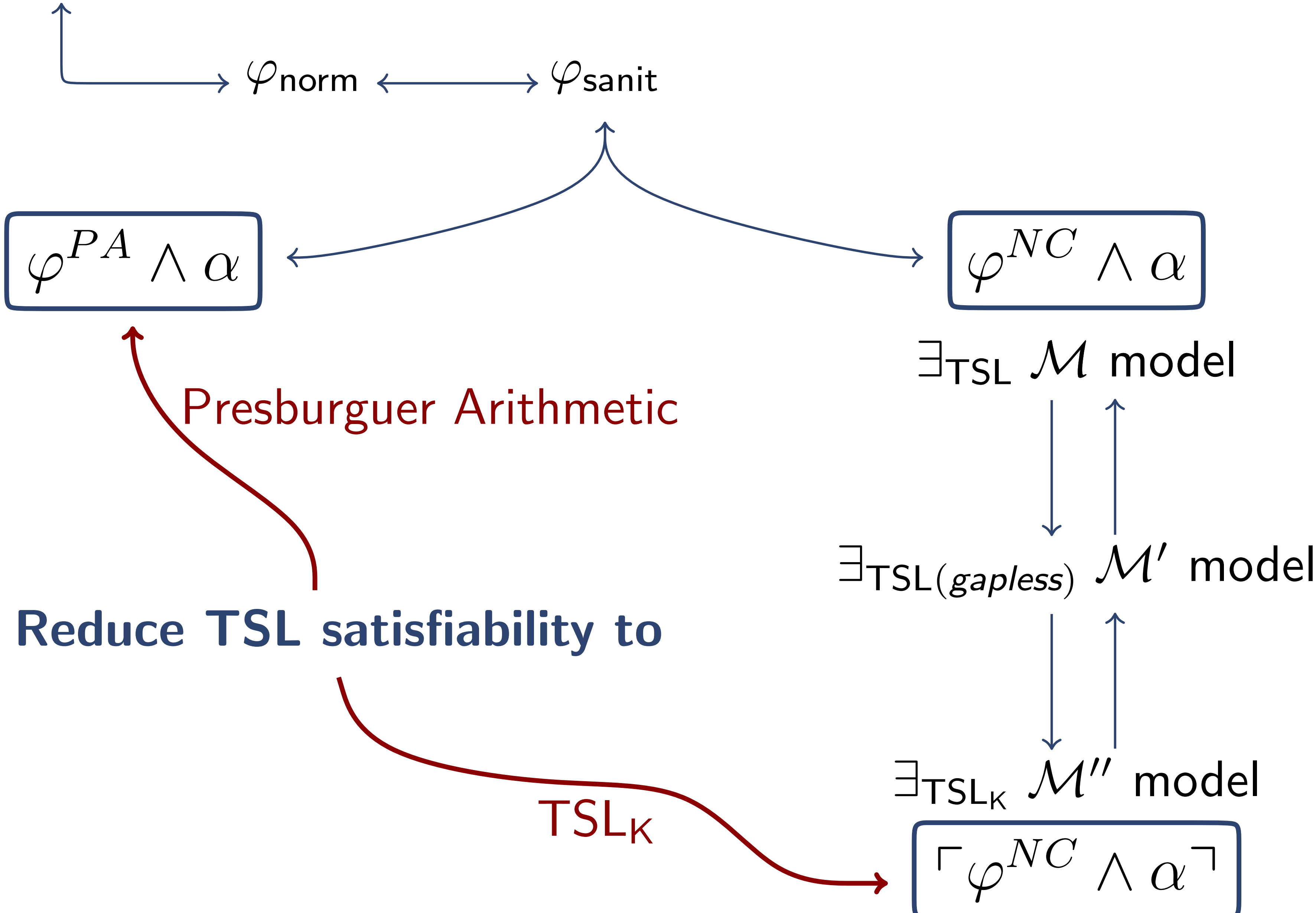
Decision Procedure for TSL: Correctness

- Let φ be a TSL formula



Decision Procedure for TSL: Correctness

- Let φ be a TSL formula



Empirical Evaluation

- We have **implemented** TSL decision procedure in **LEAP**
- We verify **shape preservation** and **functional properties**
- We **compare** TSL with previous TSL_K **performance**

Empirical Evaluation

	Form.	#Calls to DPs					VC time (s.)		Time (s.)
	$\#\varphi$	TSL	TSL_1	TSL_2	TSL_3	TSL_4	slowest	avg	DP
skiplist	560	28	45	92	38	14	5.40	0.24	19.64
region	1583	56	111	185	76	–	22.66	0.54	42.93
next	1899	30	39	55	22	–	0.32	0.02	1.60
order	2531	57	167	286	116	4	2.35	0.84	6.75
$skiplist_{KDE}$	214	14	37	61	32	12	5.93	0.24	13.14
$nodes_{KDE}$	585	32	99	174	76	–	3.10	0.17	9.36
$pointers_{KDE}$	1115	27	38	42	16	–	0.22	0.01	0.76
$values_{KDE}$	797	34	120	194	76	–	0.64	0.06	3.06
funcInsert	75	7	9	2	–	–	0.02	0.01	0.04
funcRemove	75	8	9	15	2	–	0.04	0.01	0.10
$skiplist_1$	119	–	32	–	–	–	0.10	0.01	0.32
$region_1$	119	–	27	–	–	–	0.14	0.01	0.28
$skiplist_2$	137	–	–	47	–	–	2.15	0.05	4.13
$region_2$	122	–	–	27	–	–	1.08	0.03	2.44
$skiplist_3$	154	–	–	–	62	–	776.45	15.27	1221.52
$region_3$	124	–	–	–	27	–	17.36	0.34	26.92
$skiplist_4$	171	–	–	–	–	77	**T.O.**	**T.O.**	**T.O.**
$region_4$	126	–	–	–	–	27	226.08	4.30	348.44

Empirical Evaluation

	Form.	#Calls to DPs					VC time (s.)		Time (s.)
	$\#\varphi$	TSL	TSL_1	TSL_2	TSL_3	TSL_4	slowest	avg	DP
skiplist	560	28	45	92	38	14	5.40	0.24	19.64
region	1583	56	111	185	76	–	22.66	0.54	42.93
next	1899	30	39	55	22	–	0.32	0.02	1.60
order	2531	57	167	286	116	4	2.35	0.84	6.75
$skiplist_{KDE}$	214	14	37	61	32	12	5.93	0.24	13.14
$nodes_{KDE}$	585	32	99	174	76	–	3.10	0.17	9.36
$pointers_{KDE}$	1115	27	38	42	16	–	0.22	0.01	0.76
$values_{KDE}$	797	34	120	194	76	–	0.64	0.06	3.06
funcInsert	75	7	9	2	–	–	0.02	0.01	0.04
funcRemove	75	8	9	15	2	–	0.04	0.01	0.10
$skiplist_1$	119	–	32	–	–	–	0.10	0.01	0.32
$region_1$	119	–	27	–	–	–	0.14	0.01	0.28
$skiplist_2$	137	–	–	47	–	–	2.15	0.05	4.13
$region_2$	122	–	–	27	–	–	1.08	0.03	2.44
$skiplist_3$	154	–	–	–	62	–	776.45	15.27	1221.52
$region_3$	124	–	–	–	27	–	17.36	0.34	26.92
$skiplist_4$	171	–	–	–	–	77	**T.O.**	**T.O.**	**T.O.**
$region_4$	126	–	–	–	–	27	226.08	4.30	348.44

TSL decision procedure vs TSL_K decision procedure

Empirical Evaluation

	Form.	#Calls to DPs					VC time (s.)		Time (s.)
	#φ	TSL	TSL_1	TSL_2	TSL_3	TSL_4	slowest	avg	DP
skiplist	560	28	45	92	38	14	5.40	0.24	19.64
region	1583	56	111	185	76	–	22.66	0.54	42.93
next	1899	30	39	55	22	–	0.32	0.02	1.60
order	2531	57	167	286	116	4	2.35	0.84	6.75
$skiplist_{KDE}$	214	14	37	61	32	12	5.93	0.24	13.14
$nodes_{KDE}$	585	32	99	174	76	–	3.10	0.17	9.36
$pointers_{KDE}$	1115	27	38	42	16	–	0.22	0.01	0.76
$values_{KDE}$	797	34	120	194	76	–	0.64	0.06	3.06
funcInsert	75	7	9	2	–	–	0.02	0.01	0.04
funcRemove	75	8	9	15	2	–	0.04	0.01	0.10
$skiplist_1$	119	–	32	–	–	–	0.10	0.01	0.32
$region_1$	119	–	27	–	–	–	0.14	0.01	0.28
$skiplist_2$	137	–	–	47	–	–	2.15	0.05	4.13
$region_2$	122	–	–	27	–	–	1.08	0.03	2.44
$skiplist_3$	154	–	–	–	62	–	776.45	15.27	1221.52
$region_3$	124	–	–	–	27	–	17.36	0.34	26.92
$skiplist_4$	171	–	–	–	–	77	**T.O.**	**T.O.**	**T.O.**
$region_4$	126	–	–	–	–	27	226.08	4.30	348.44

A TSL queries is **decomposed into multiple calls** to TSL_K

Empirical Evaluation

	Form. $\#\varphi$	#Calls to DPs TSL	TSL_1	TSL_2	TSL_3	TSL_4	VC time (s.) slowest	avg	Time (s.) DP
skiplist	560	28	45	92	38	14	5.40	0.24	19.64
region	1583	56	111	185	76	–	22.66	0.54	42.93
next	1899	30	39	55	22	–	0.32	0.02	1.60
order	2531	57	167	286	116	4	2.35	0.84	6.75
$skiplist_{KDE}$	214	14	37	61	32	12	5.93	0.24	13.14
$nodes_{KDE}$	585	32	99	174	76	–	3.10	0.17	9.36
$pointers_{KDE}$	1115	27	38	42	16	–	0.22	0.01	0.76
$values_{KDE}$	797	34	120	194	76	–	0.64	0.06	3.06
funcInsert	75	7	9	2	–	–	0.02	0.01	0.04
funcRemove	75	8	9	15	2	–	0.04	0.01	0.10
$skiplist_1$	119	–	32	–	–	–	0.10	0.01	0.32
$region_1$	119	–	27	–	–	–	0.14	0.01	0.28
$skiplist_2$	137	–	–	47	–	–	2.15	0.05	4.13
$region_2$	122	–	–	27	–	–	1.08	0.03	2.44
$skiplist_3$	154	–	–	–	62	–	776.45	15.27	1221.52
$region_3$	124	–	–	–	27	–	17.36	0.34	26.92
$skiplist_4$	171	–	–	–	–	77	**T.O.**	**T.O.**	**T.O.**
$region_4$	126	–	–	–	–	27	226.08	4.30	348.44

While **TSL_K did not scale beyond a skiplist with 4 levels**...

Empirical Evaluation

	Form.	#Calls to DPs					VC time (s.)		Time (s.)
	$\#\varphi$	TSL	TSL_1	TSL_2	TSL_3	TSL_4	slowest	avg	DP
skiplist	560	28	45	92	38	14	5.40	0.24	19.64
region	1583	56	111	185	76	–	22.66	0.54	42.93
next	1899	30	39	55	22	–	0.32	0.02	1.60
order	2531	57	167	286	116	4	2.35	0.84	6.75
$skiplist_{KDE}$	214	14	37	61	32	12	5.93	0.24	13.14
$nodes_{KDE}$	585	32	99	174	76	–	3.10	0.17	9.36
$pointers_{KDE}$	1115	27	38	42	16	–	0.22	0.01	0.76
$values_{KDE}$	797	34	120	194	76	–	0.64	0.06	3.06
funcInsert	75	7	9	2	–	–	0.02	0.01	0.04
funcRemove	75	8	9	15	2	–	0.04	0.01	0.10
$skiplist_1$	119	–	32	–	–	–	0.10	0.01	0.32
$region_1$	119	–	27	–	–	–	0.14	0.01	0.28
$skiplist_2$	137	–	–	47	–	–	2.15	0.05	4.13
$region_2$	122	–	–	27	–	–	1.08	0.03	2.44
$skiplist_3$	154	–	–	–	62	–	776.45	15.27	1221.52
$region_3$	124	–	–	–	27	–	17.36	0.34	26.92
$skiplist_4$	171	–	–	–	–	77	**T.O.**	**T.O.**	**T.O.**
$region_4$	126	–	–	–	–	27	226.08	4.30	348.44

...**TSL** verified the examples in **less than a minute for every height**

Conclusions

- We presented **TSL**, a theory for skiplists of arbitrary height
- **TSL** can reason about memory, cells, pointers, regions, reachability, ordered lists and sublists
- We proved **TSL decidable** and presented a **decision procedure**
- Decision procedure has been **implemented** as part of **LEAP**
- We used TSL to **verify real world implementations**
- **Future work**
  - Extend TSL for concurrent skiplists
  - Verify further implementations
  - Improve automation by generating and propagating invariants

LEAP and examples availblable at

`software.imdea.org/leap`