Skiplists and Balanced Search

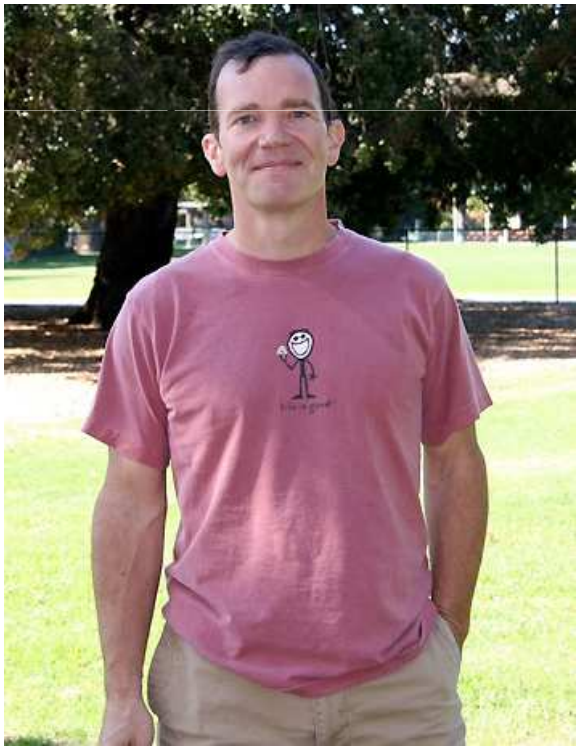
Adrian Fiedler Juliane Keller

Um was geht es?

- Nebenläufige Suchstrukturen mit logarithmischer Tiefe
- Alternativen: sequentielle Suchstrukturen wie Rot-Schwarzbäume oder AVL-Bäume
- Rebalancing bei nebenläufigen Suchstrukturen kann bottlenecks hervorrufen
- →nebenläufige Implementierung von Skiplists

Überblick

- 1990 erfunden von William Pugh (Miterfinder des Java Memory Models)



java.util.concurrent

Class ConcurrentSkipListSet<E>

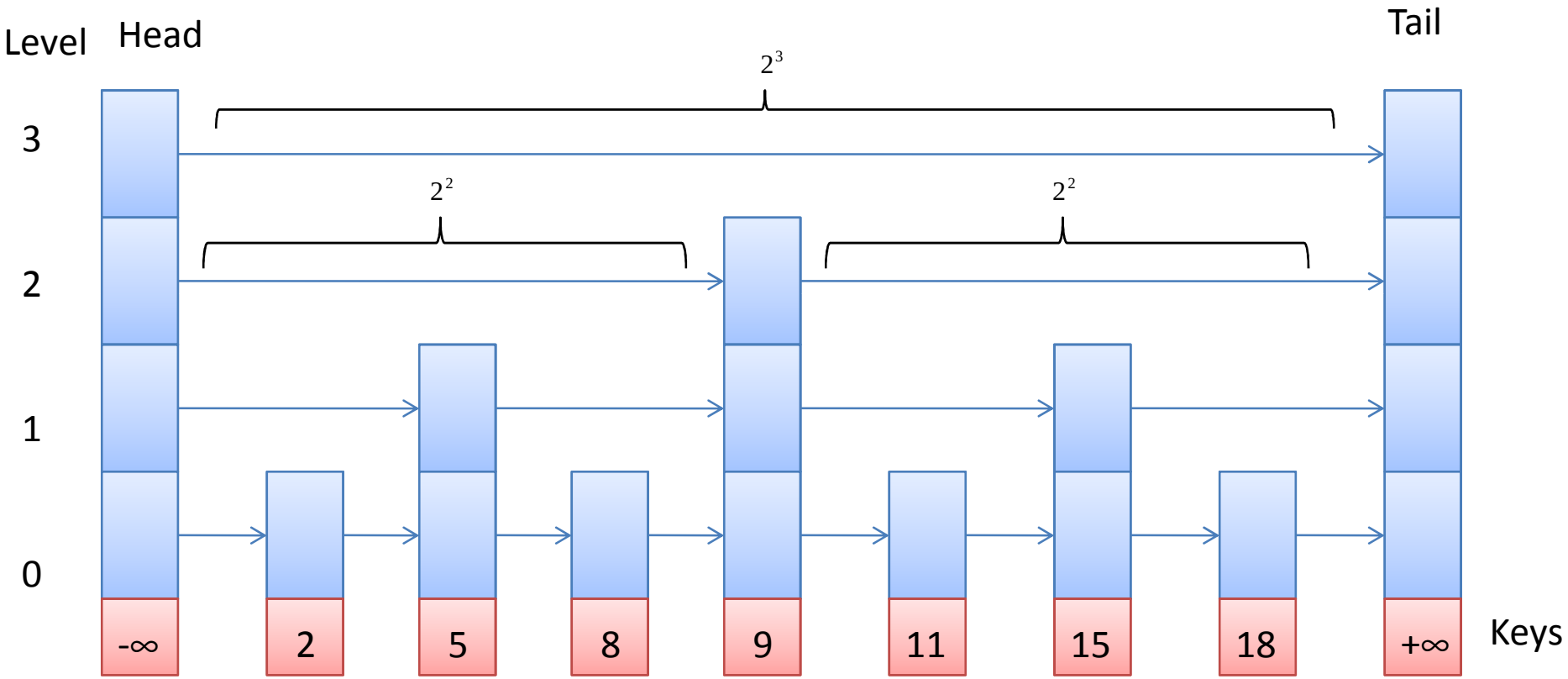
java.lang.Object
  └ java.util.AbstractCollection<E>
    └ java.util.AbstractSet<E>
      └ java.util.concurrent.ConcurrentSkipListSet<E>

Skiplists-Vorteile

- Erwartete logarithmische Suchzeit
- Kein Rebalancing
- contains() ist wait-free

- →LazySkipList
- →LockFreeSkipList

Aufbau



Sequentielle Skiplists

- Die Liste ist ein Set

  →Die Schlüssel sind einzigartig
- Ansammlung von Linked-Lists → imitiert Balanced-Search-Tree
- Die unterste Liste enthält alle Knoten

  → Höhere Listen sind eine Unterliste der unteren Listen

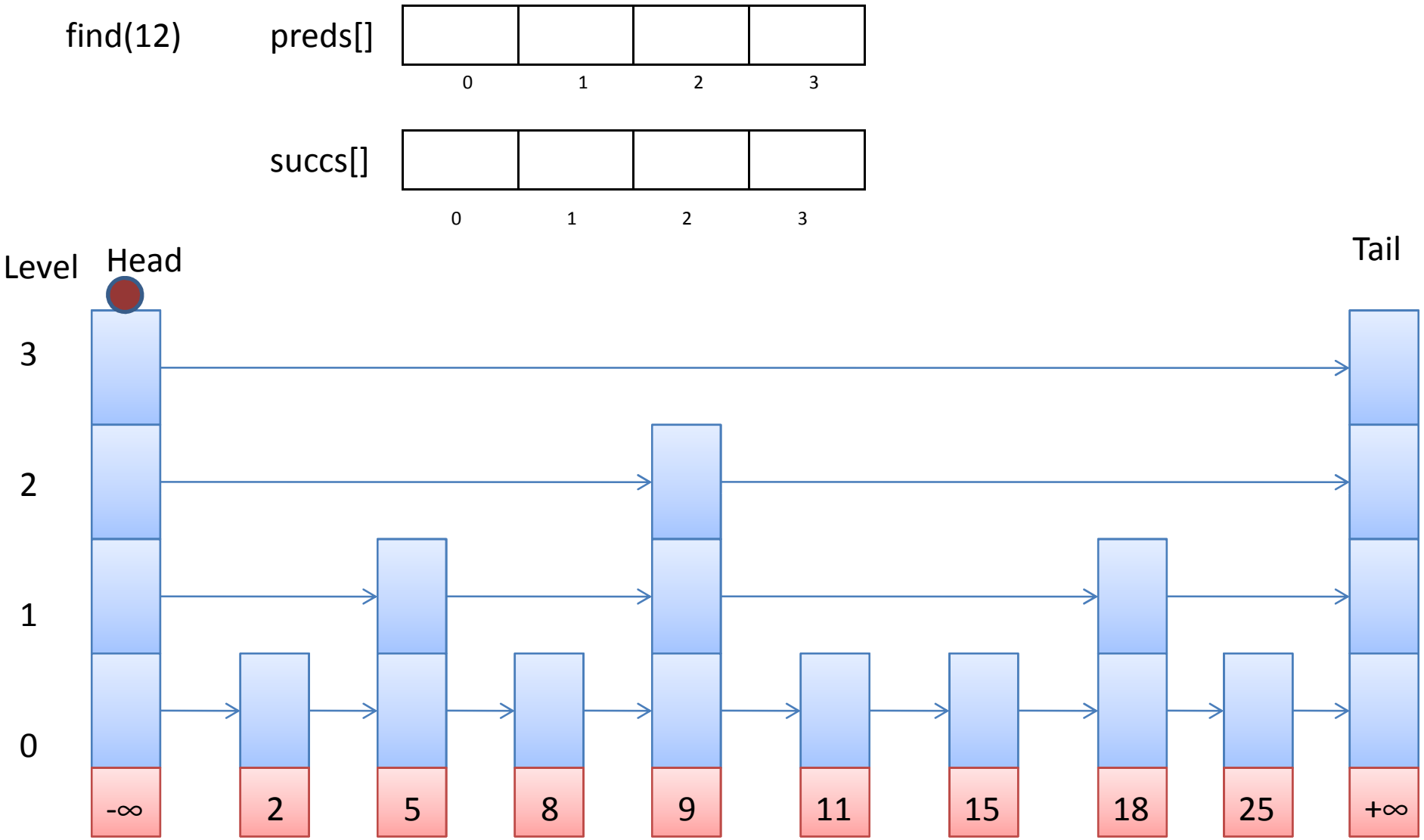
Idee: Höhere Listen sind „Abkürzungen“ in die unteren Listen

→ Jede Referenz auf Level x springt über 2^x Knoten in den darunter liegenden Listen

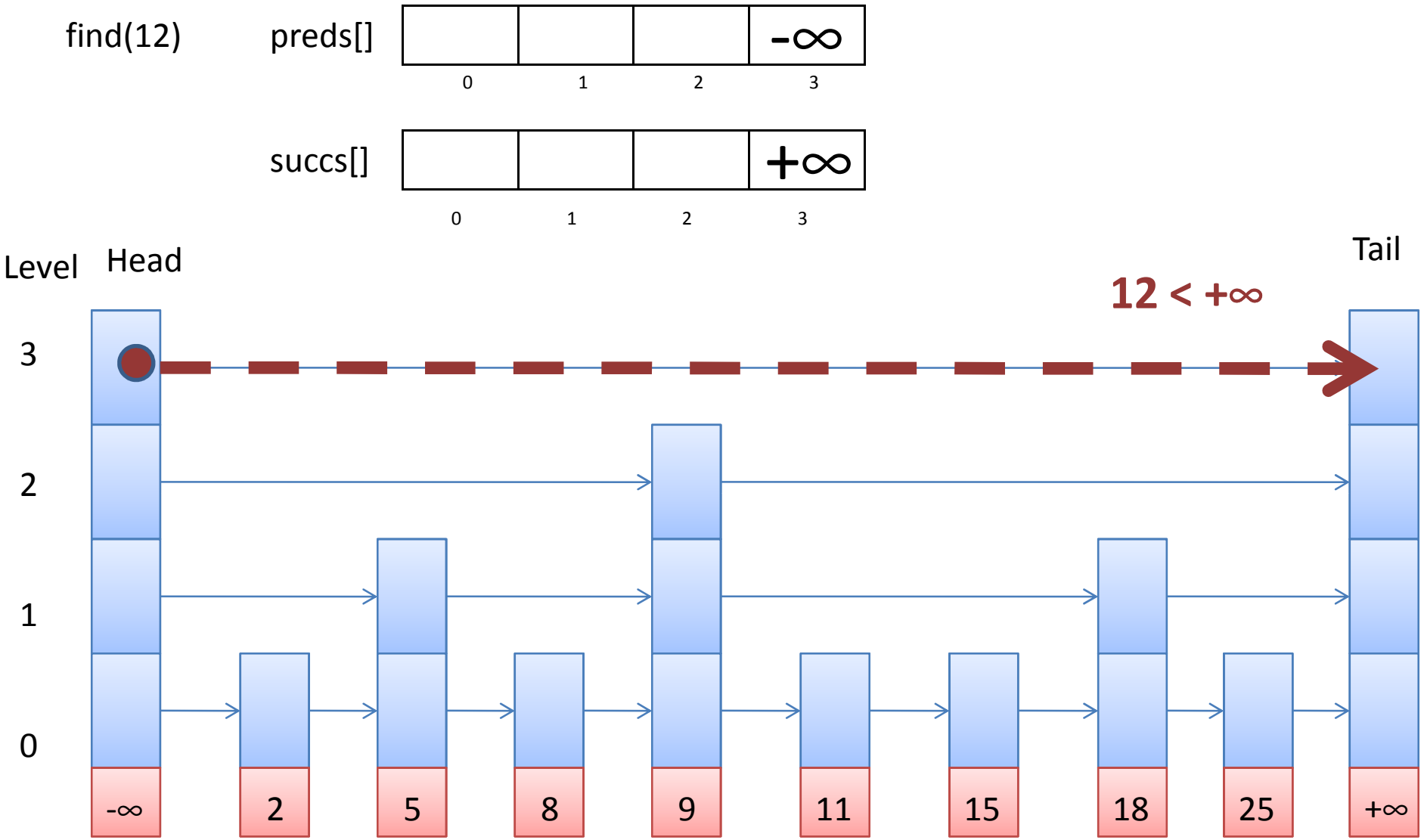
Balanciertheit

- Toplevel jeder Knoten wird zufällig gewählt
- Wahrscheinlichkeit z.B. 1/2, dass ein Knoten im Level i auch im Level i+1 erscheint
- => ½ der Knoten von Level 0 in Level 1
- => ½ der Knoten von Level 1 in Level 2
- => ¼ der Knoten von Level 0 in Level 2 usw.

→ balanciert aber ohne Rebalancing



Die Methode find() sucht den Knoten und füllt dabei die Arrays mit Vorgängern (preds[]) und Nachfolgern (succs[])


find(12)
preds[]
-∞
0
1
2
3
succs[]
+∞
0
1
2
3
Level
Head
Tail
12 < +∞
3
2
1
0
-∞
2
5
8
9
11
15
18
25
+∞

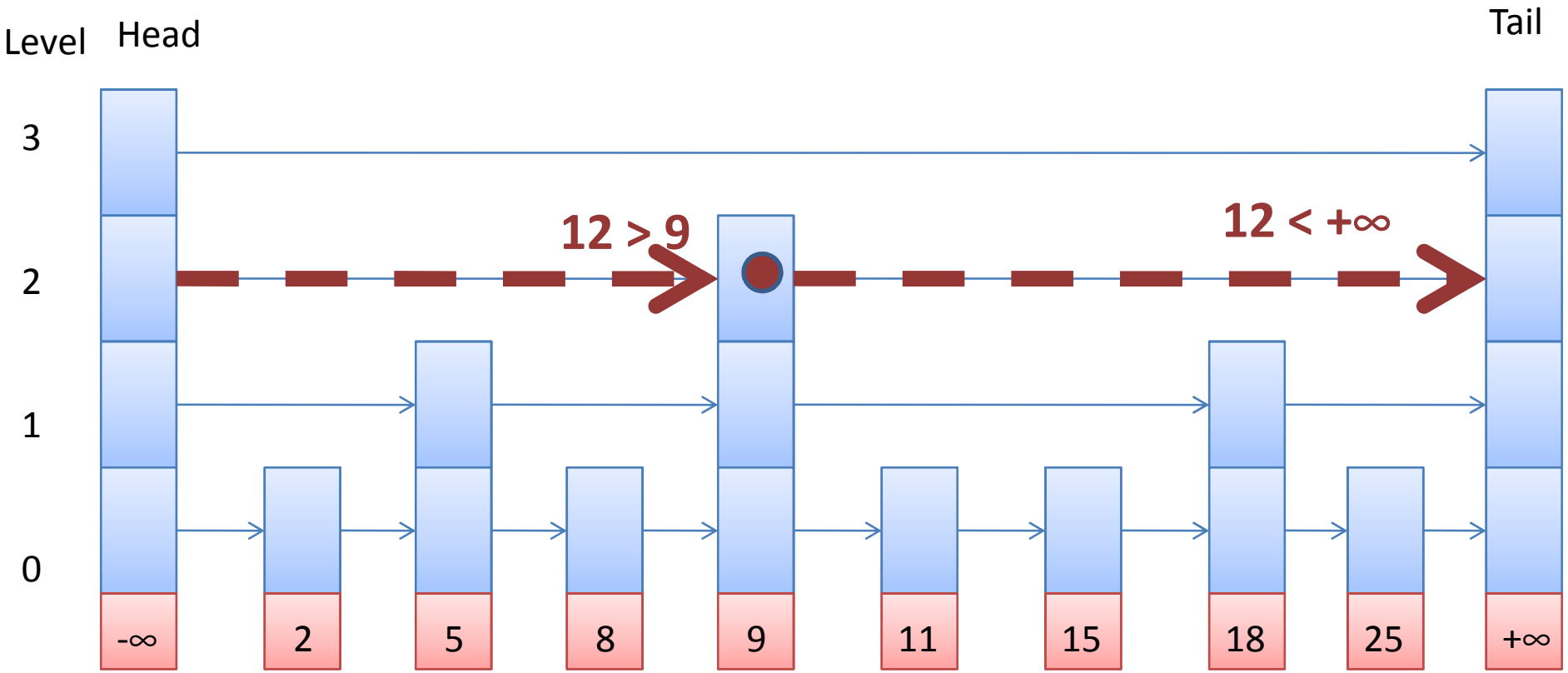
find(12)

preds[]

		9	-∞
0	1	2	3

succs[]

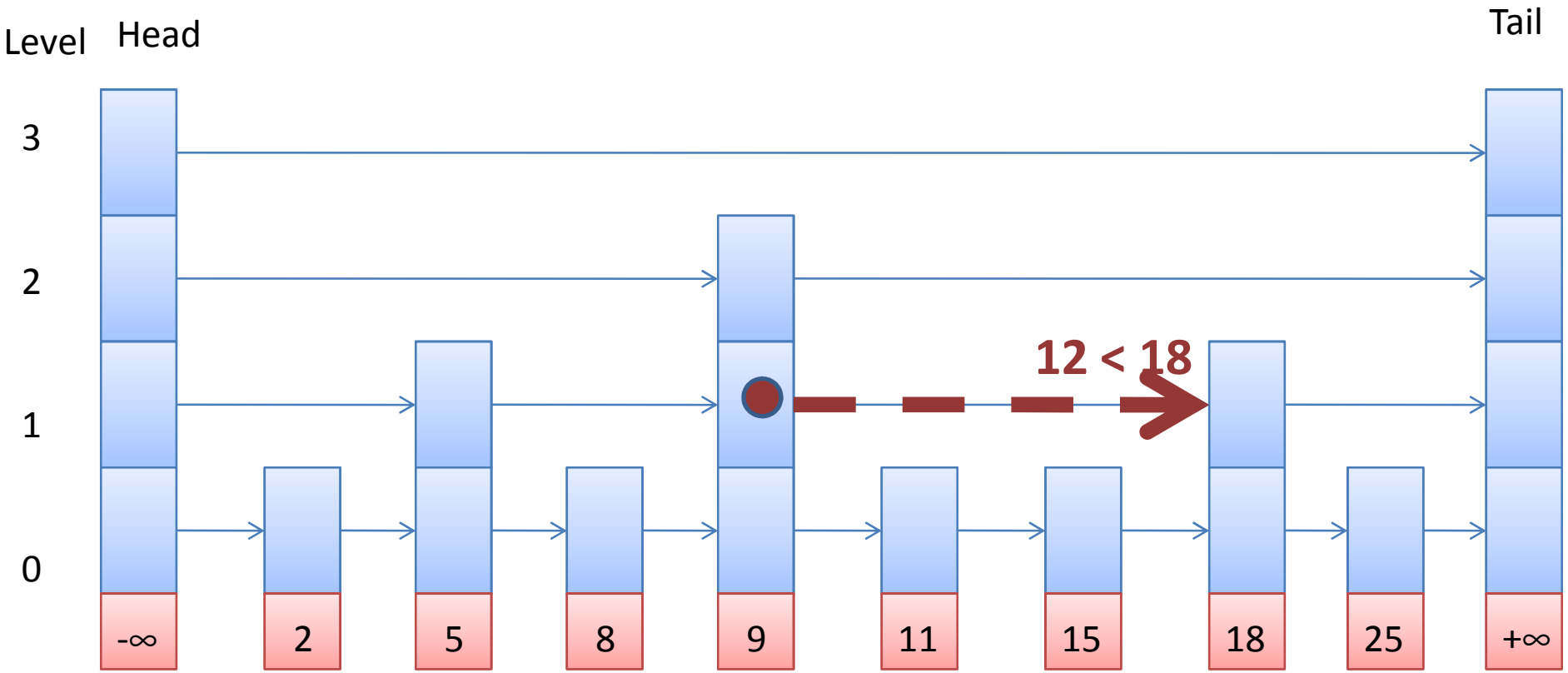
		+∞	+∞
0	1	2	3



find(12)

preds[]		9	9	-∞
	0	1	2	3

succs[]		18	+∞	+∞
	0	1	2	3



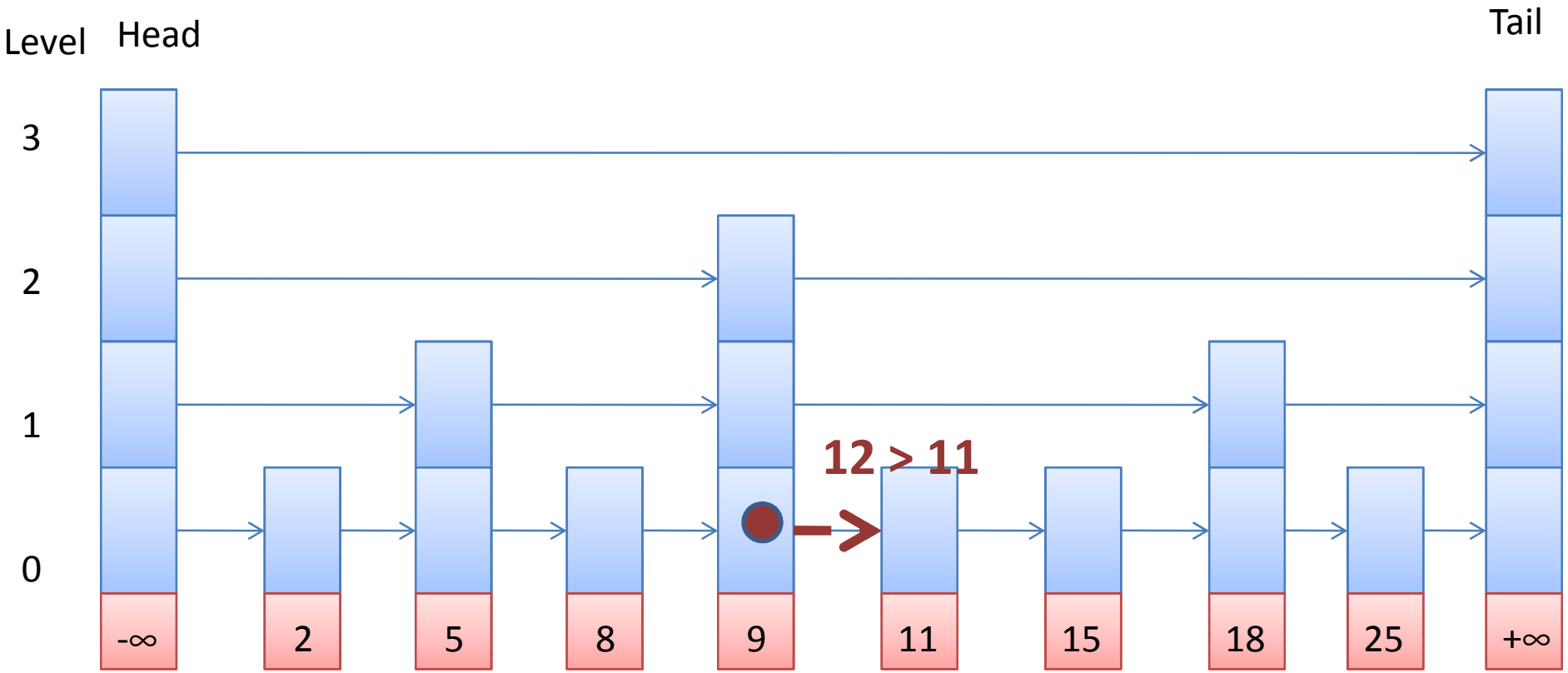
find(12)

preds[]

	9	9	-∞
0	1	2	3

succs[]

	18	+∞	+∞
0	1	2	3

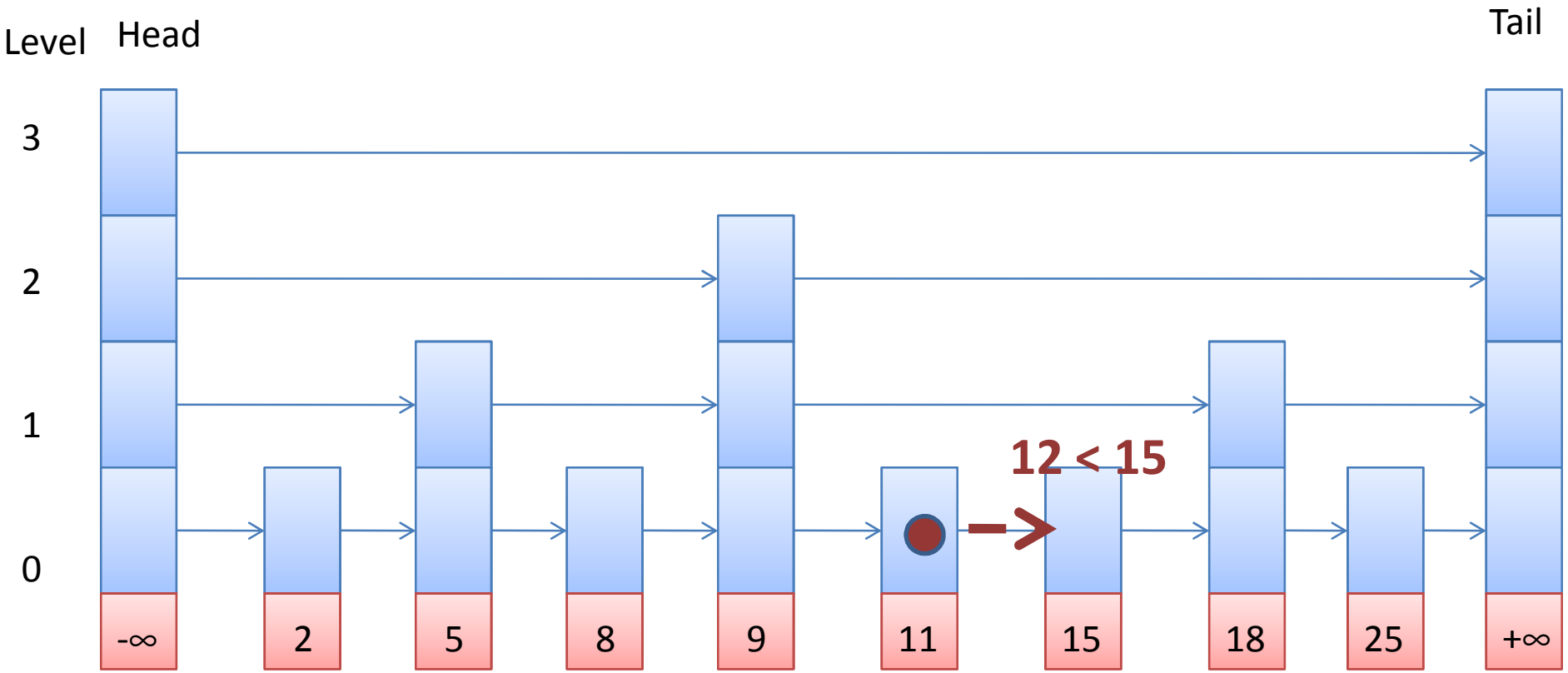


find(12)

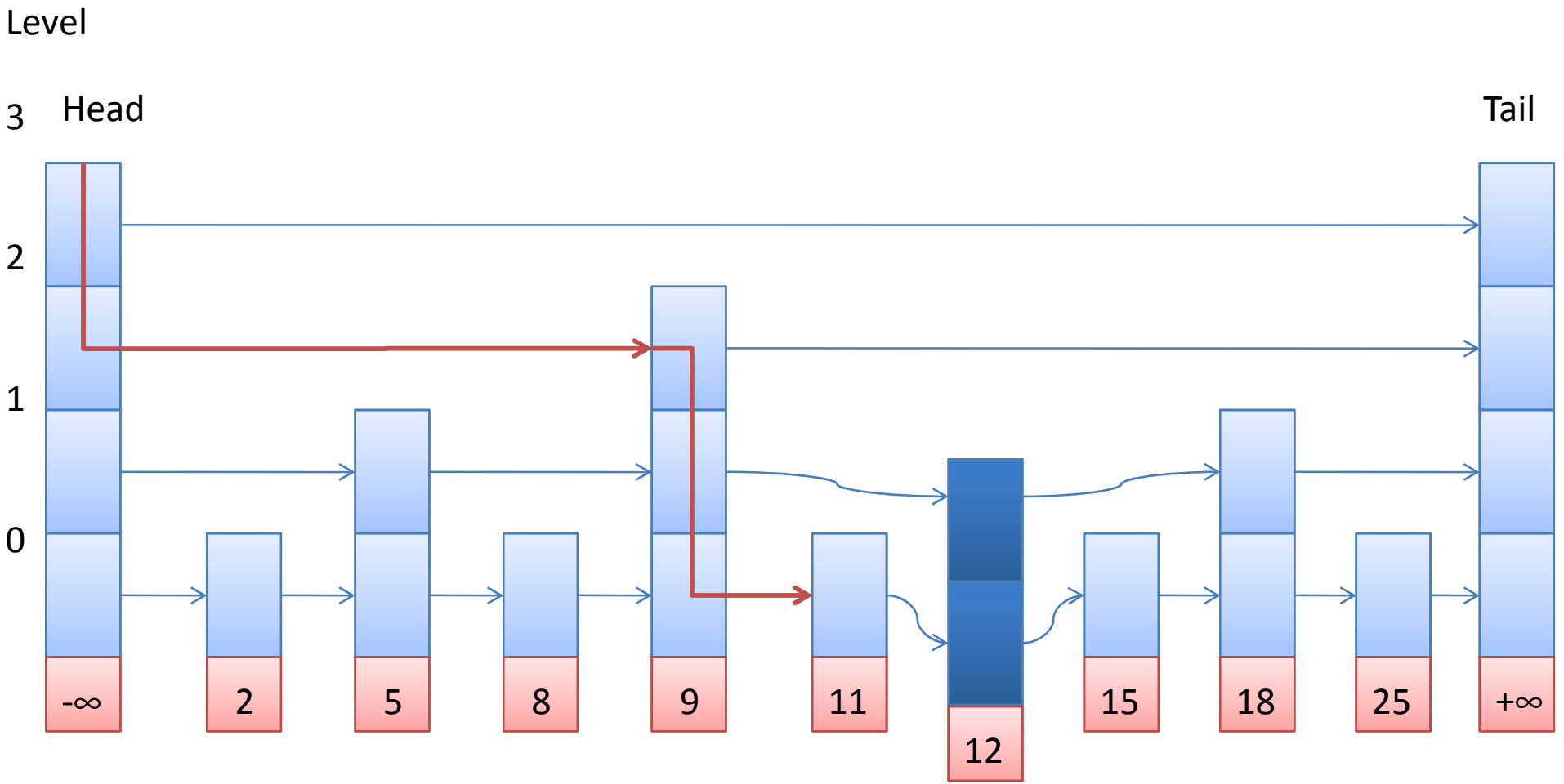
preds[]	11	9	9	-∞
	0	1	2	3

succs[]	15	18	+∞	+∞
	0	1	2	3

return false



add(12)



Die Links werden mit Hilfe der zuvor erstellten Arrays (preds[] und succs[]) gesetzt.

LazyList - Wiederholung

- Markierungen zeigen an, ob sich ein Knoten im Set befindet
- contains() ist wait-free

  → Markierungen und Locks werden ignoriert

LazySkiplist

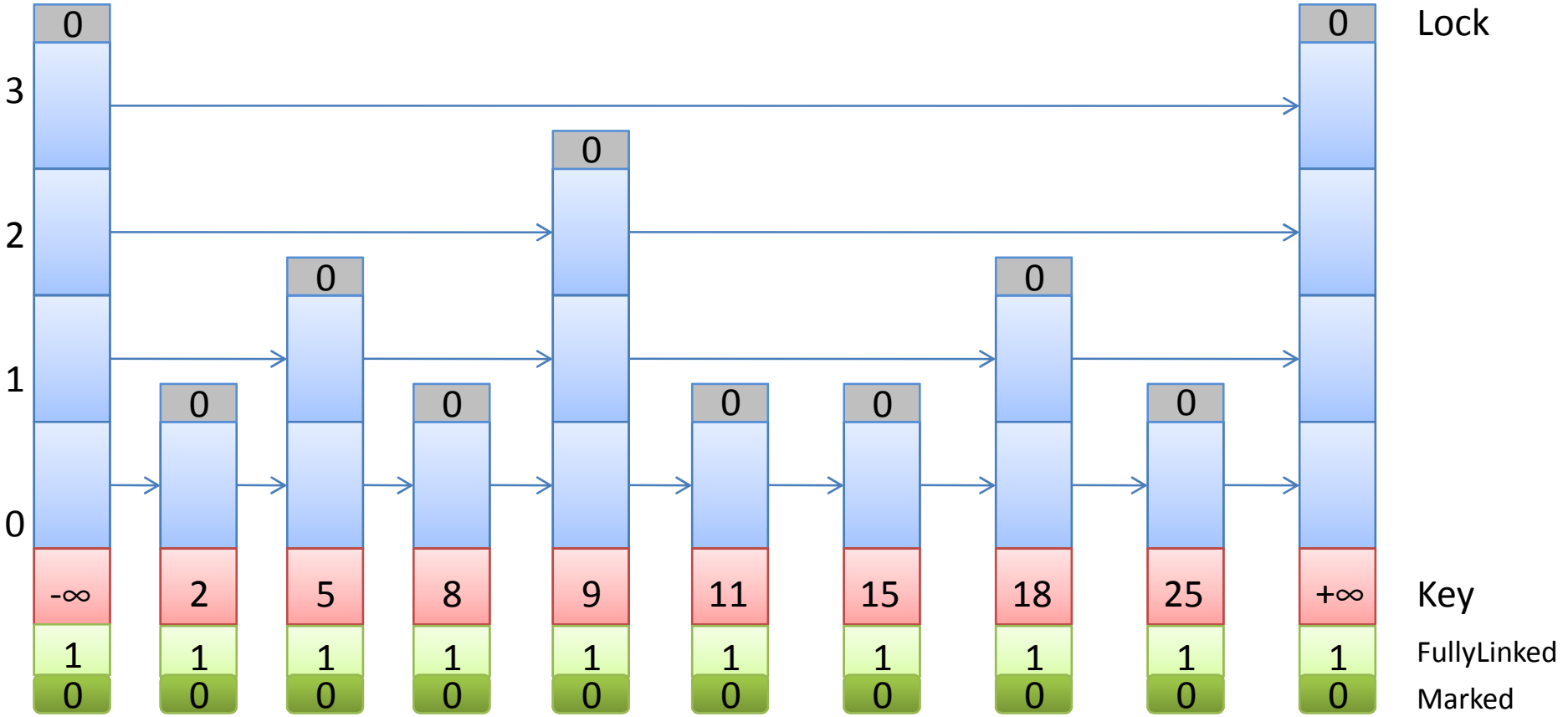
- Lock basiert nach *LazyList*

  →Jedes Level der Skiplist ist eine *LazyList*

- Mix aus blockierenden und nicht blockierenden Techniken
  - *add()* und *remove()* nutzen optimistic fine-grained Locking
  - *contains()* ist wait-free
- Zusätzliche „marked“ und „fullyLinked“-Felder

remove(18)

1. Schritt: find(18) für preds[] und succs[] des evtl. vorhandenen Knotens

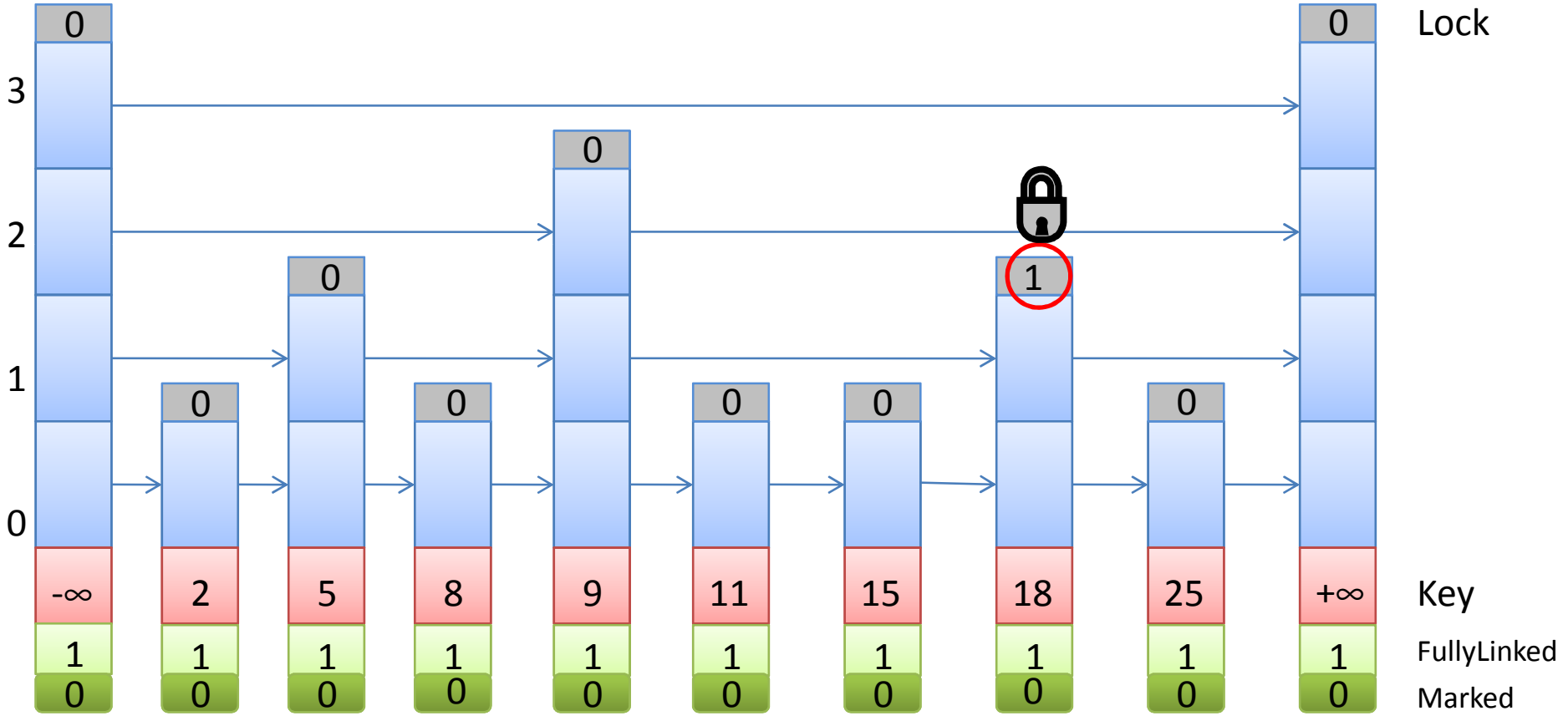


find()

```
1   int find(T x, Node<T>[] preds, Node<T>[] succs)
2   {
3       int key = x.hashCode();
4       int lFound = -1;          //Level in dem der Knoten
5                                 //gefunden wurde: wenn nicht
6                                 //bleibt es -1 bedeutet false
7       Node<T> pred = head;
8       for(int level = MAX_LEVEL; level>=0; level--)
9       {
10          Node<T> curr = pred.next[level];
11          while(key > curr.key)
12          {
13              pred = curr;
14              curr = pred.next[level];
15          }
16          if(lFound == -1 && key == curr.key)
17          {
18              lFound = level;
19          }
20          preds[level]=pred;
21          succs[level]=curr;
22      }
23      return lFound;
24  }
```

remove(18)

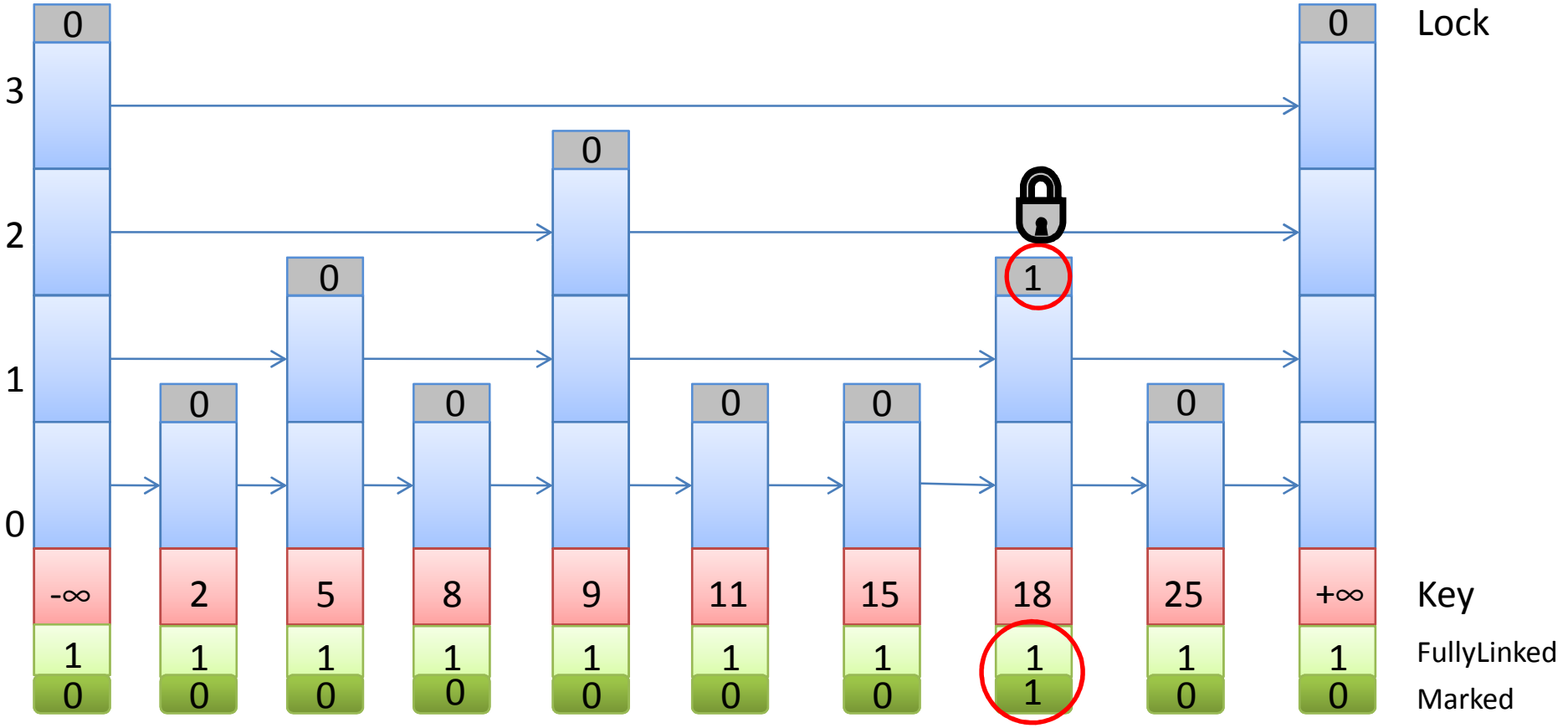
1. Schritt: find ✓ 2. Schritt: remove



Der Knoten wird erst gelockt, dann markiert und dann werden die Vorgänger gelockt, bevor die Links neu gesetzt werden

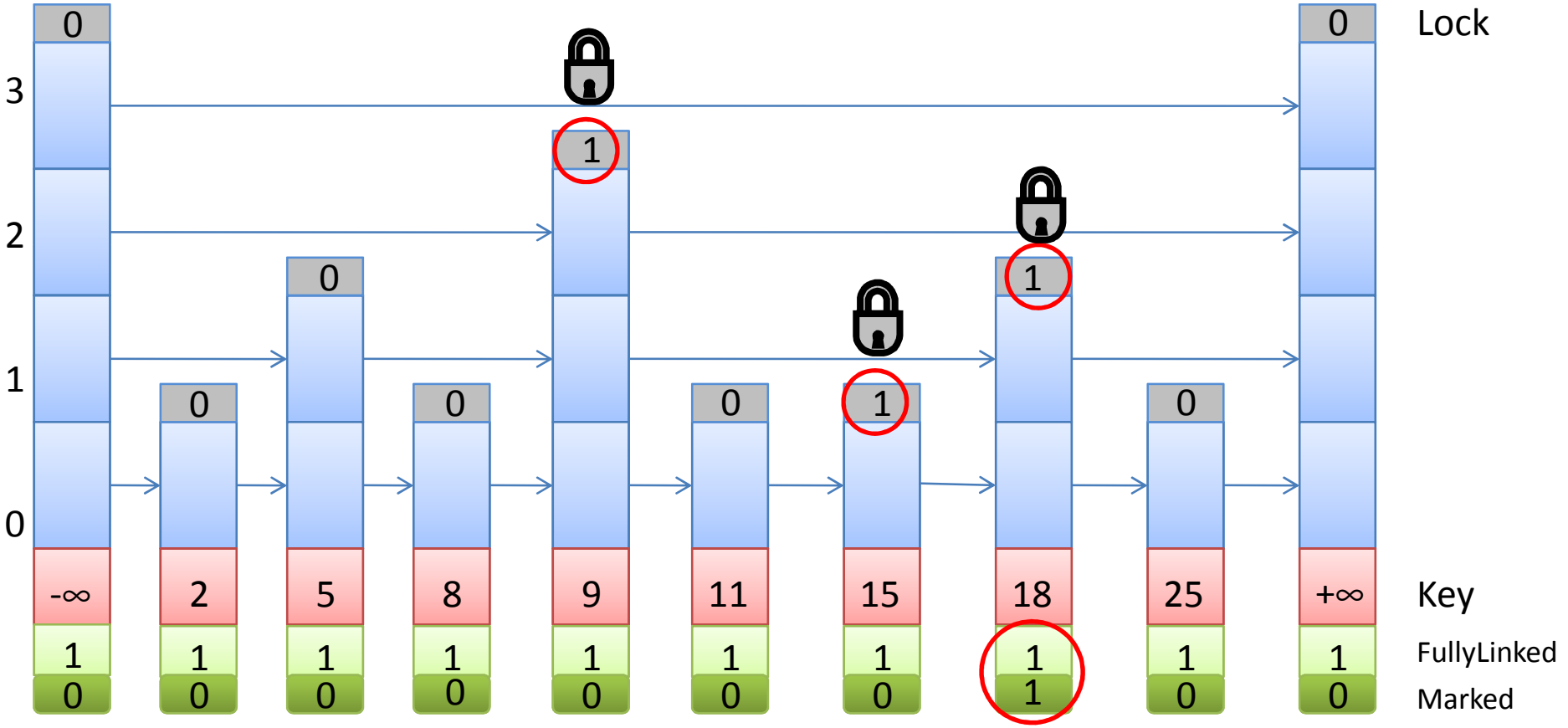
remove(18)

1. Schritt: find ✓ 2. Schritt: remove



remove(18)

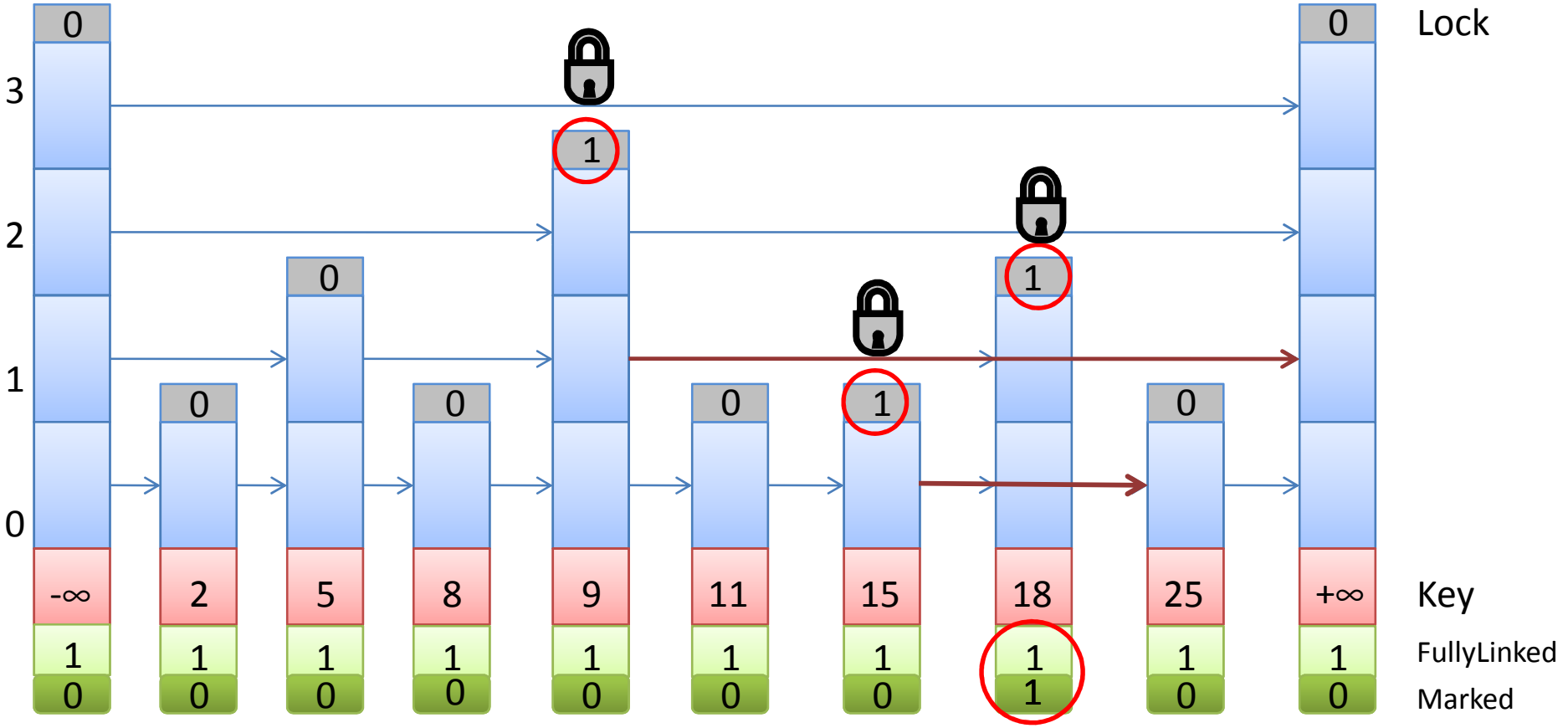
1. Schritt: find ✓ 2. Schritt: remove



remove(18)

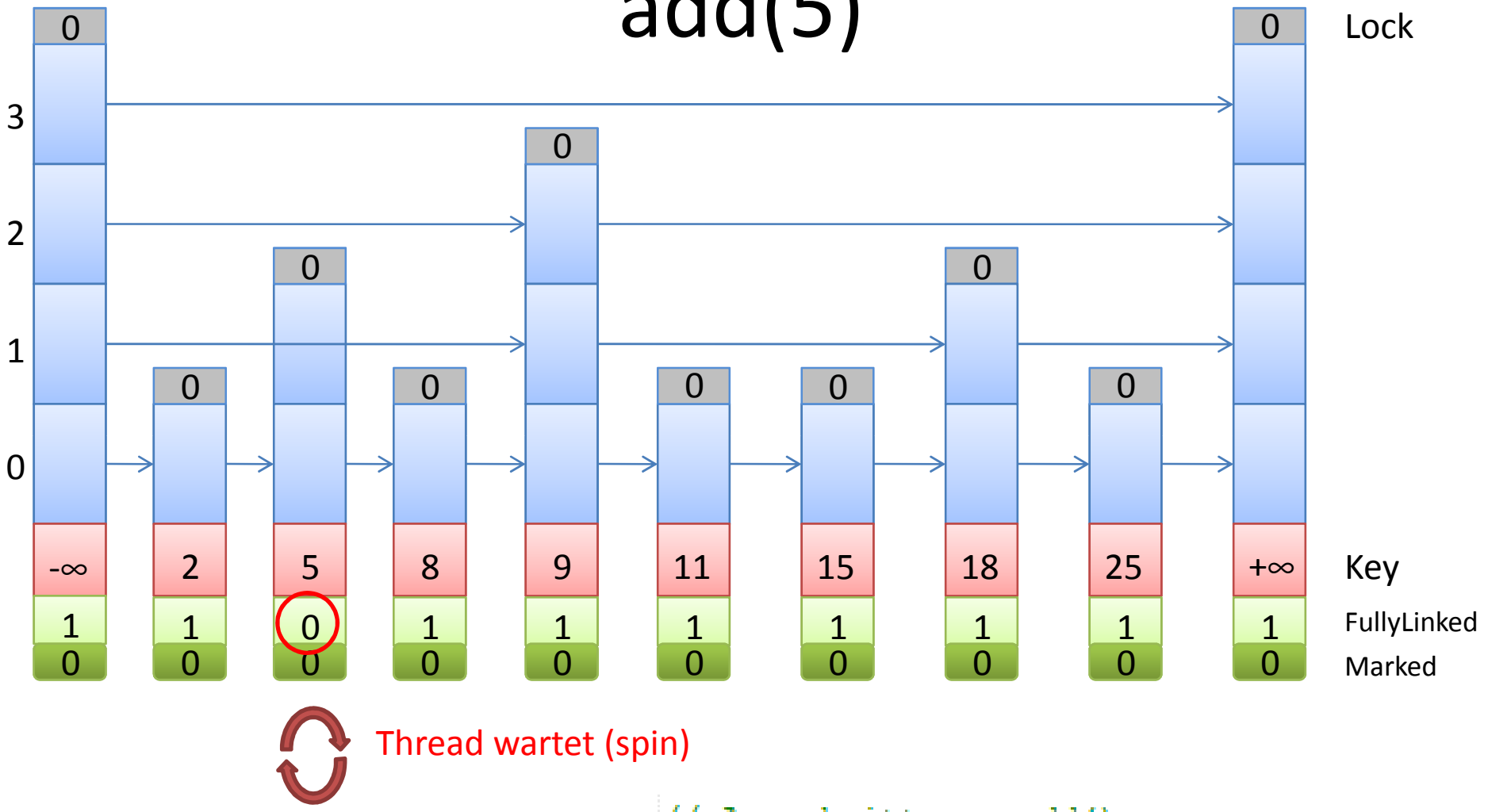
1. Schritt: find ✓

2. Schritt: remove



```
1  boolean remove(T x)
2  {
3      while(true)
4      {
5          <Suche nach x>
6          if(Suche erfolgreich)
7              {<zu löschende Knoten ist in succs[gefundenes Level]>}
8          if(zu löschende Knoten ist nicht markiert)
9              {<Knoten wird gelockt>}
10         else{return false}
11         <Knoten wird markiert>
12         for(level 0 bis Toplevel)
13             {<Vorgänger werden gelockt>}
14         <Überprüfen ob noch alles stimmt>
15         for(Toplevel bis Level 0)
16             {<Vorgänger werden jetzt auf Nachfolger des Knotens verlinkt>}
17         <gelöschterKnoten.unlock>
18         for(0 bis höchstes gelocktes Level)
19             {<preds[aktuelles Level].unlock>}
20         <return true>
21     }
22 }
23
```

add(5)




```
// Ausschnitt aus add()

while (!nodeFound.fullyLinked) {}
return false;
```

contains()

```
boolean contains(T x){
    Node<T>[]preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
    Node<T>[]succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
    int lFound = find(x, preds, succs);
    return (lFound != -1
        && succs[lFound].fullyLinked
        && !succs[lFound].marked);
}
```

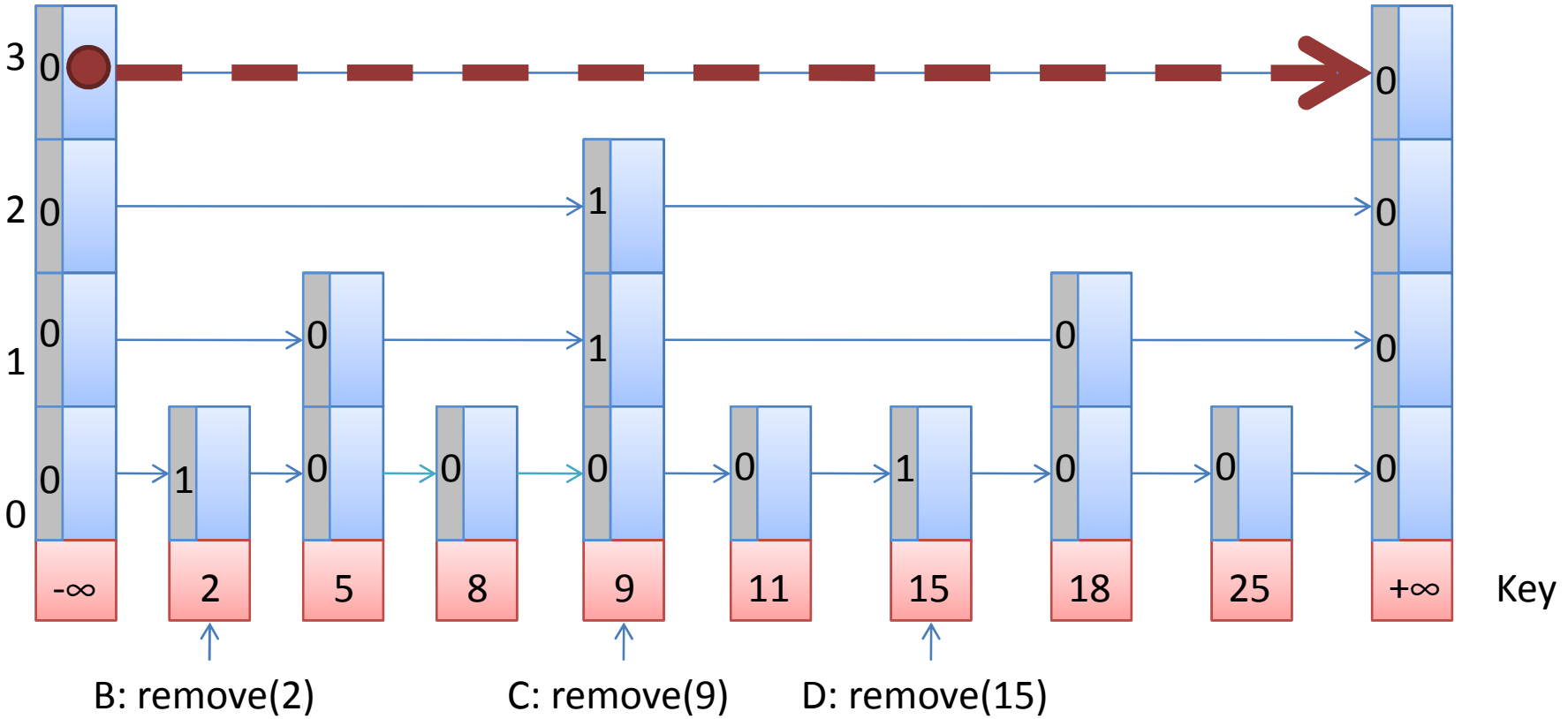
LockFreeSkiplist

- Basiert auf LockFreeList
- Keine Locks
  - →Skiplist-Eigenschaft kann nicht eingehalten werden
  - →Ein Schlüssel ist dann im Set, wenn seine Next- Referenz in der untersten Liste unmarkiert ist
    - →Fully-Linked-Feld nicht mehr nötig
    - →next ist AtomicMarkableReference<T>
  - → Veränderung durch *CompareAndSet()*
- Höhere Listen dienen nur noch als Abkürzungen
- Jedes Level der Skiplist ist eine LockFreeList
- *Find()* räumt markierte Knoten auf

A: add(12)

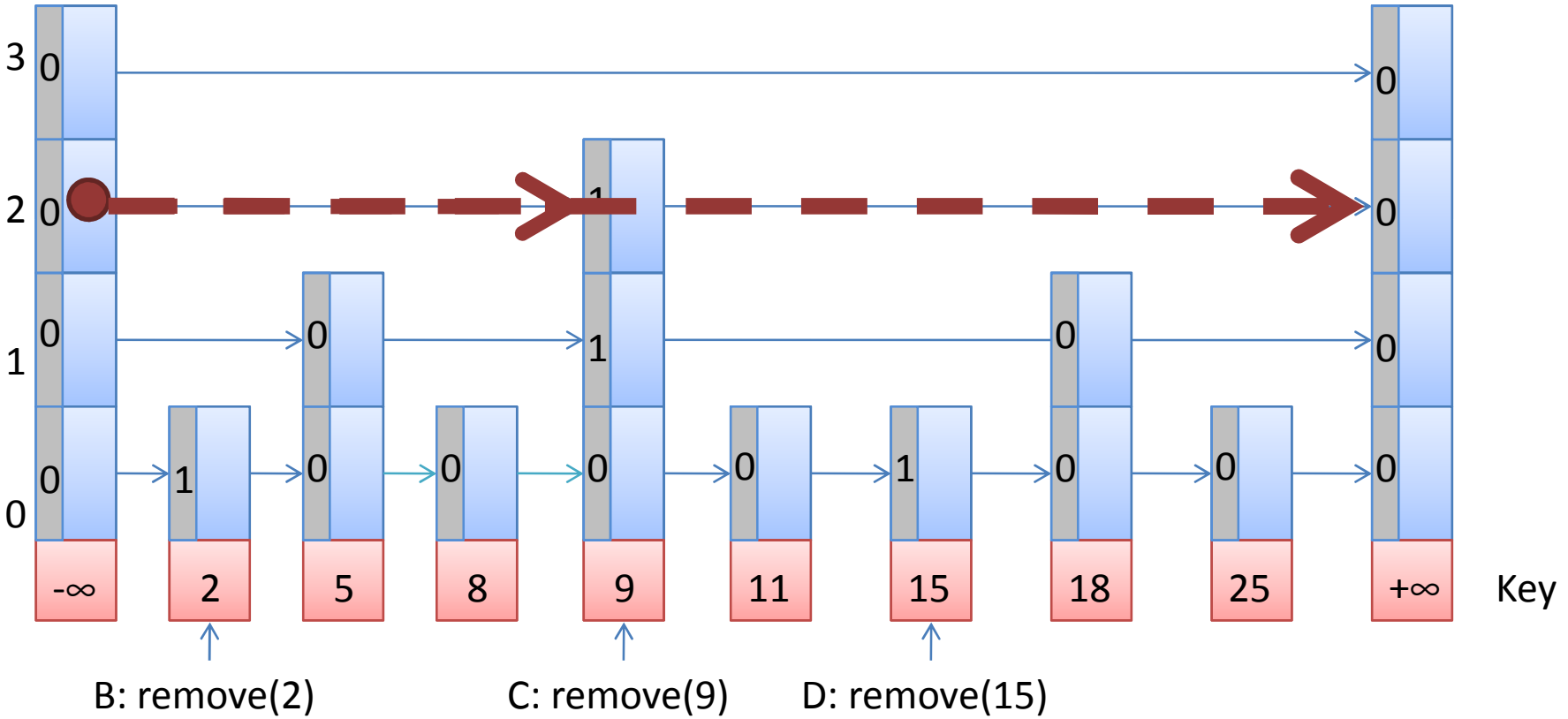
→ find(12)

Wenn find() ein markiertes next-Feld findet, löscht er die Referenz darauf und setzt sie neu.
Bei den LockfreeSkiplists wird ein Boolean zurückgegeben, da nur das unterste Level von Belang ist.



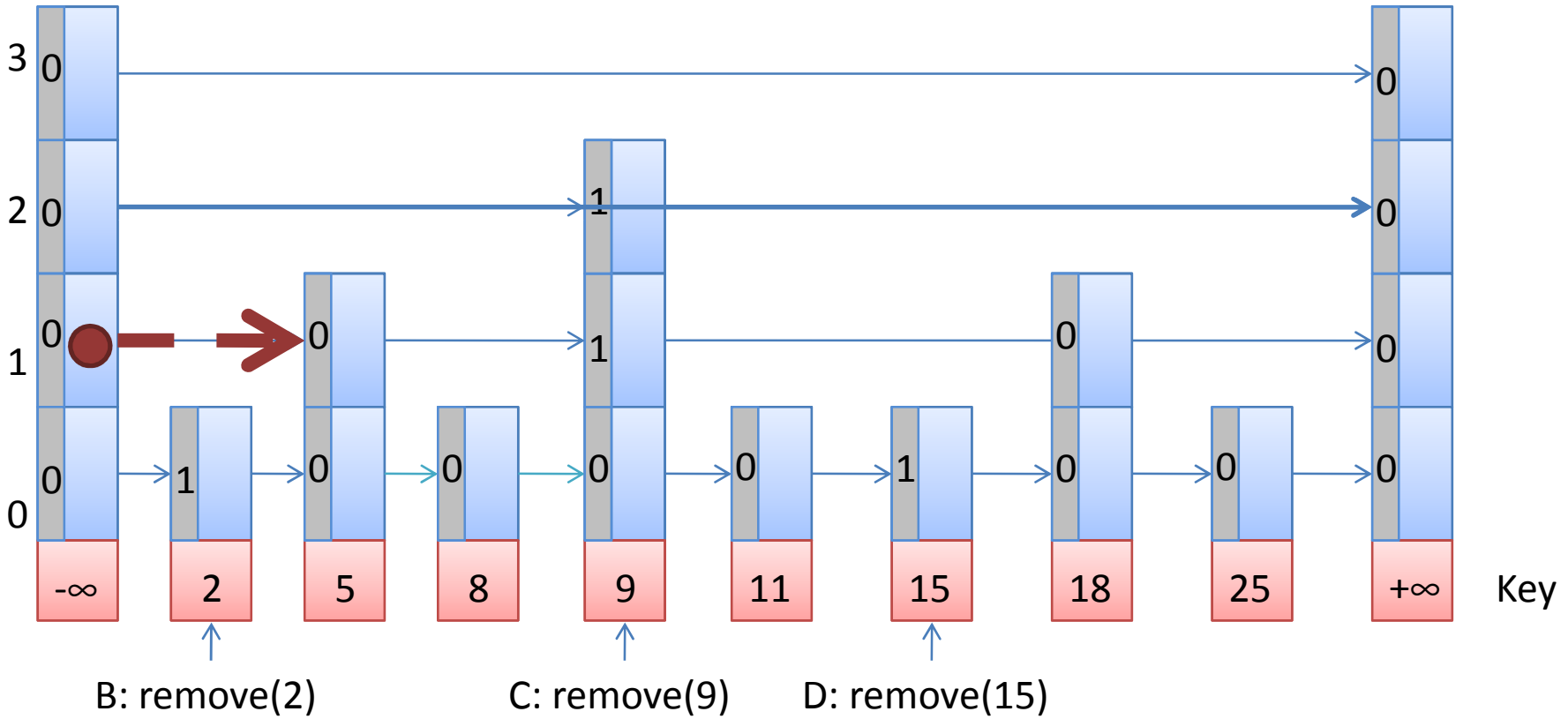
A: add(12)

→ find(12)



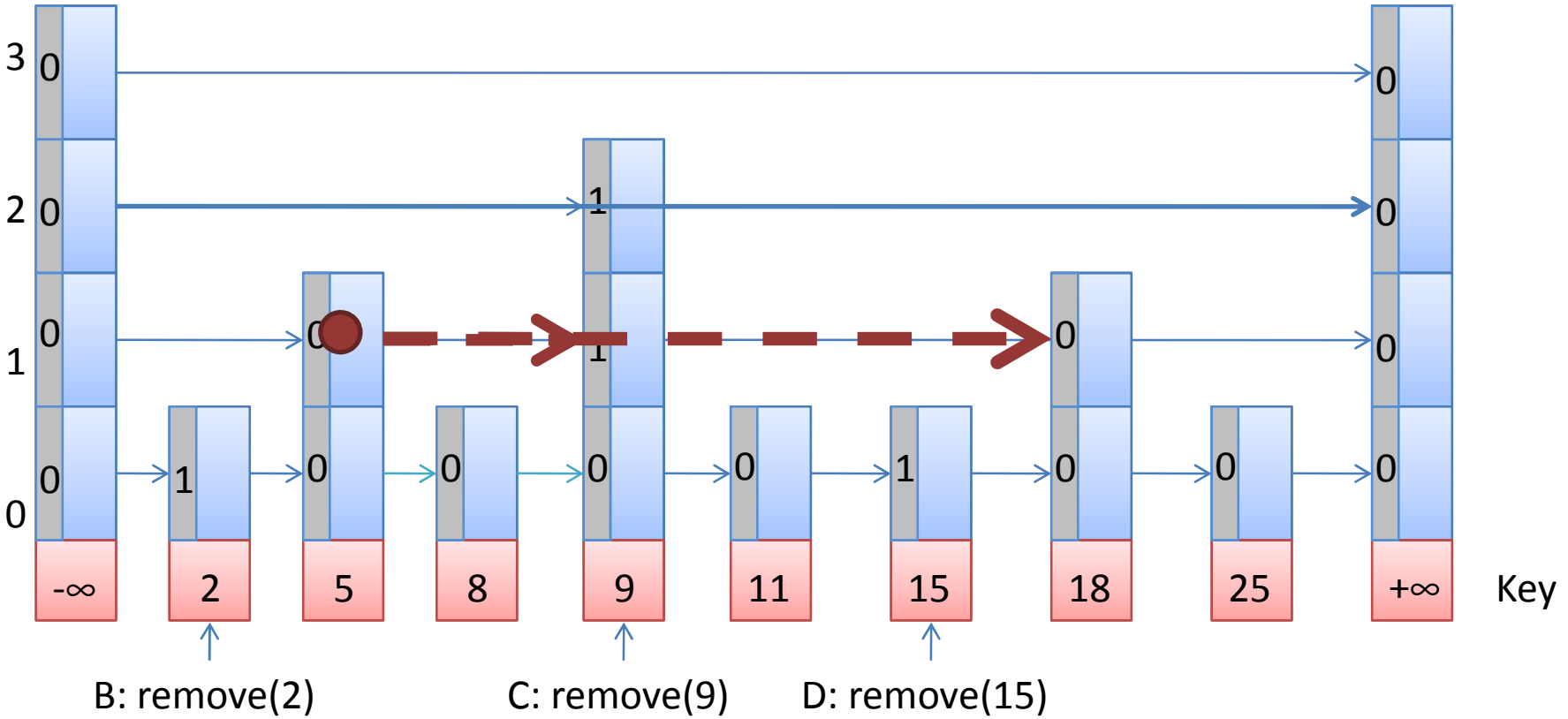
A: add(12)

→ find(12)



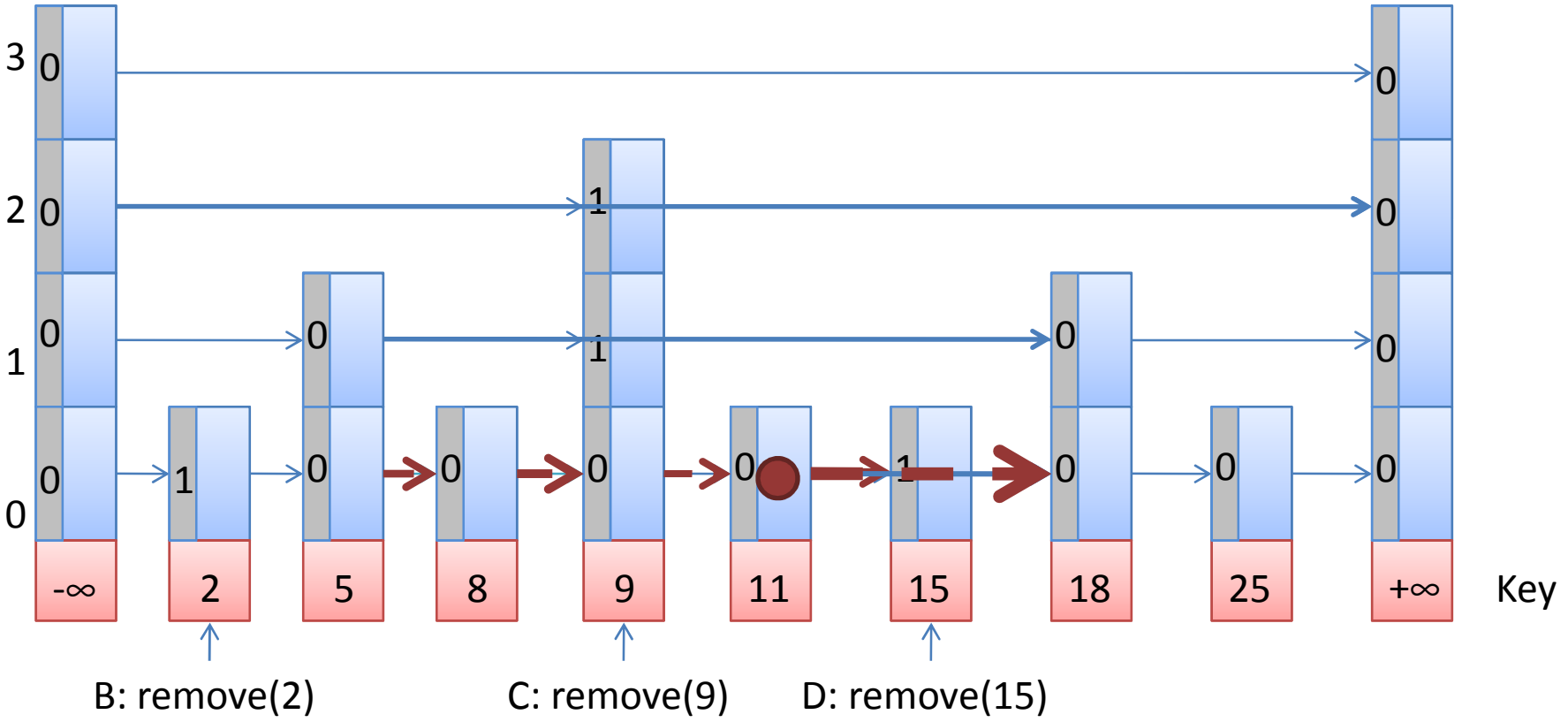
A: add(12)

→ find(12)

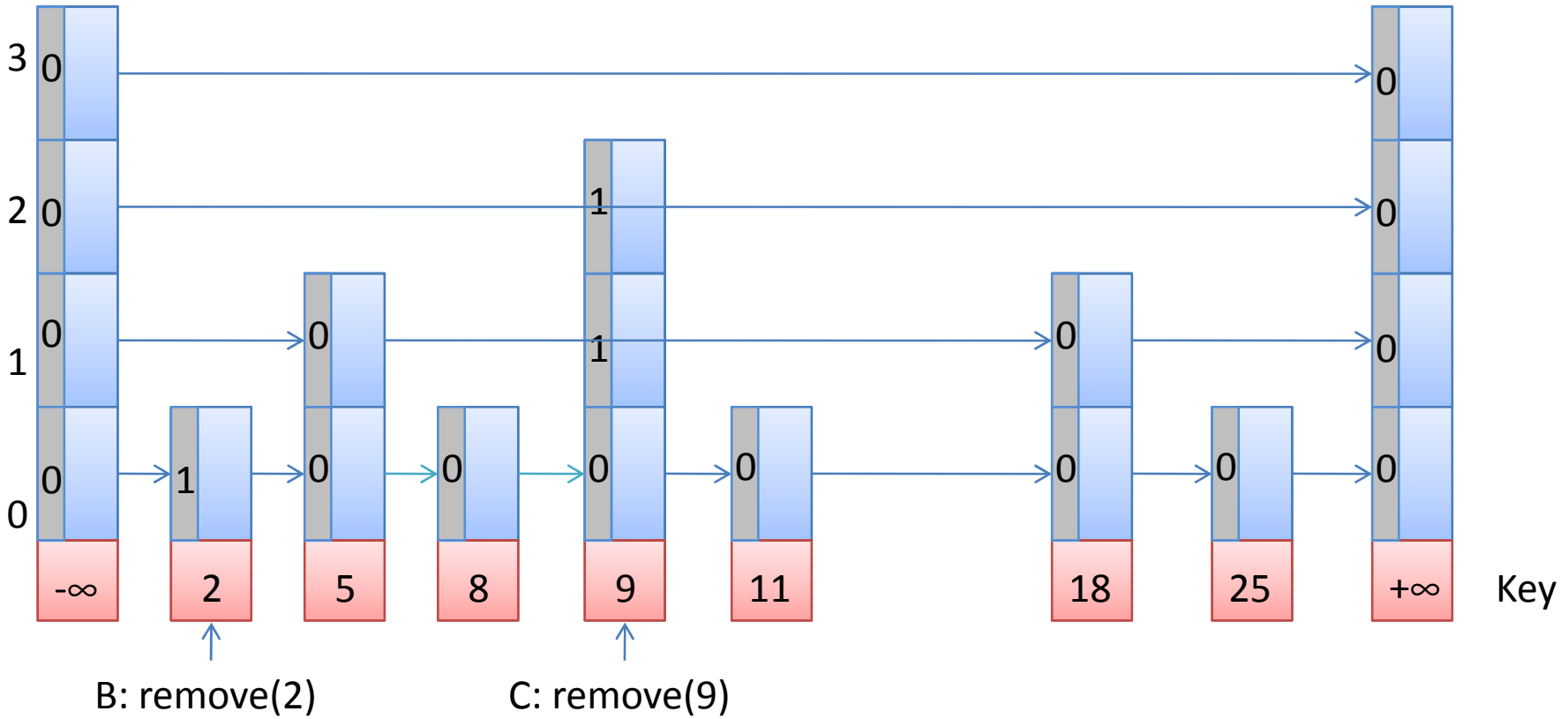


A: add(12)

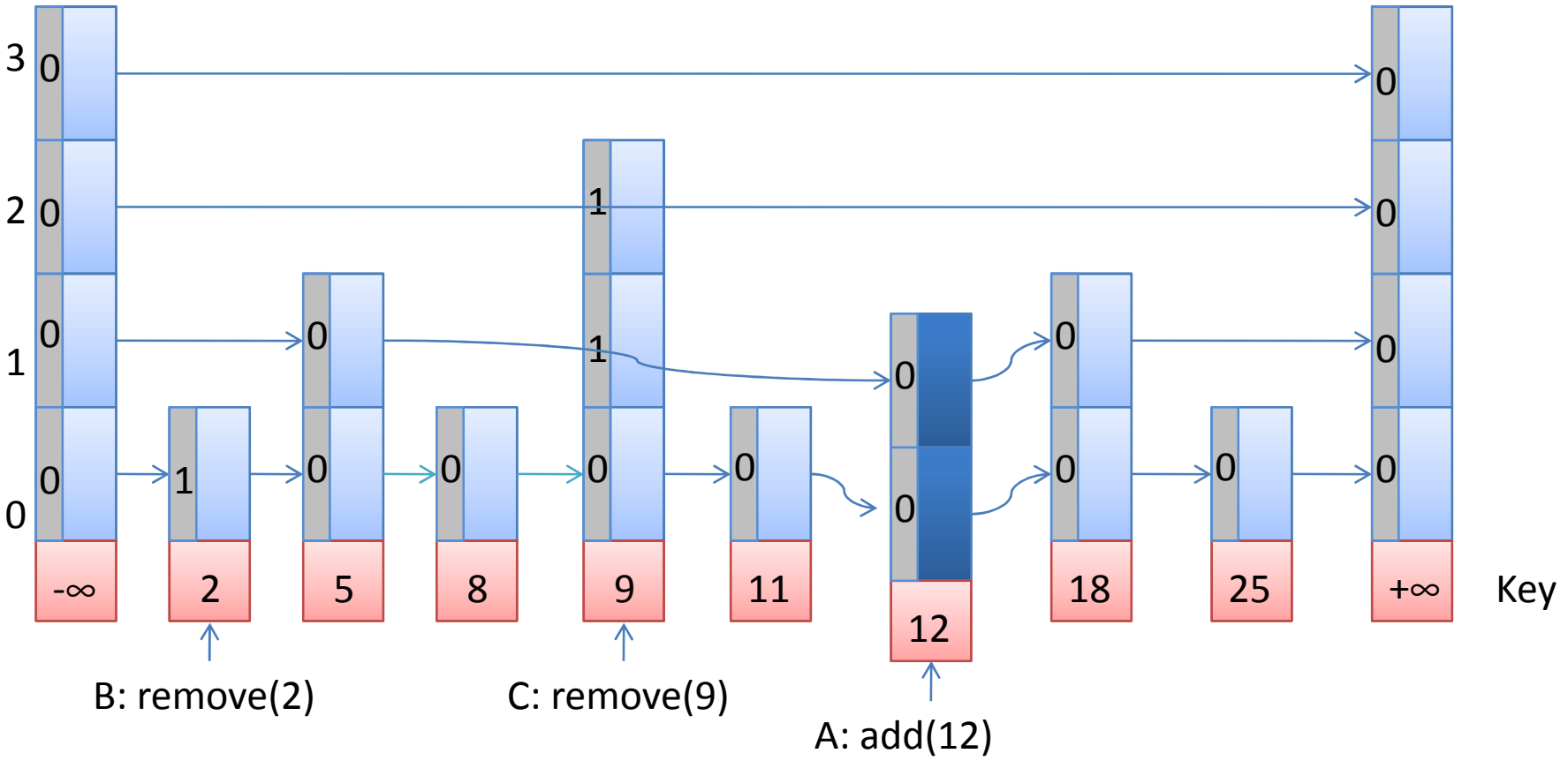
→ find(12)



A: add(12)


3
2
1
0
0
0
0
0
0
1
0
0
1
1
0
0
0
0
0
0
0
0
0
0
0
0
-∞
2
5
8
9
11
18
25
+∞
Key
B: remove(2)
C: remove(9)

A: add(12)



add() - Pseudocode

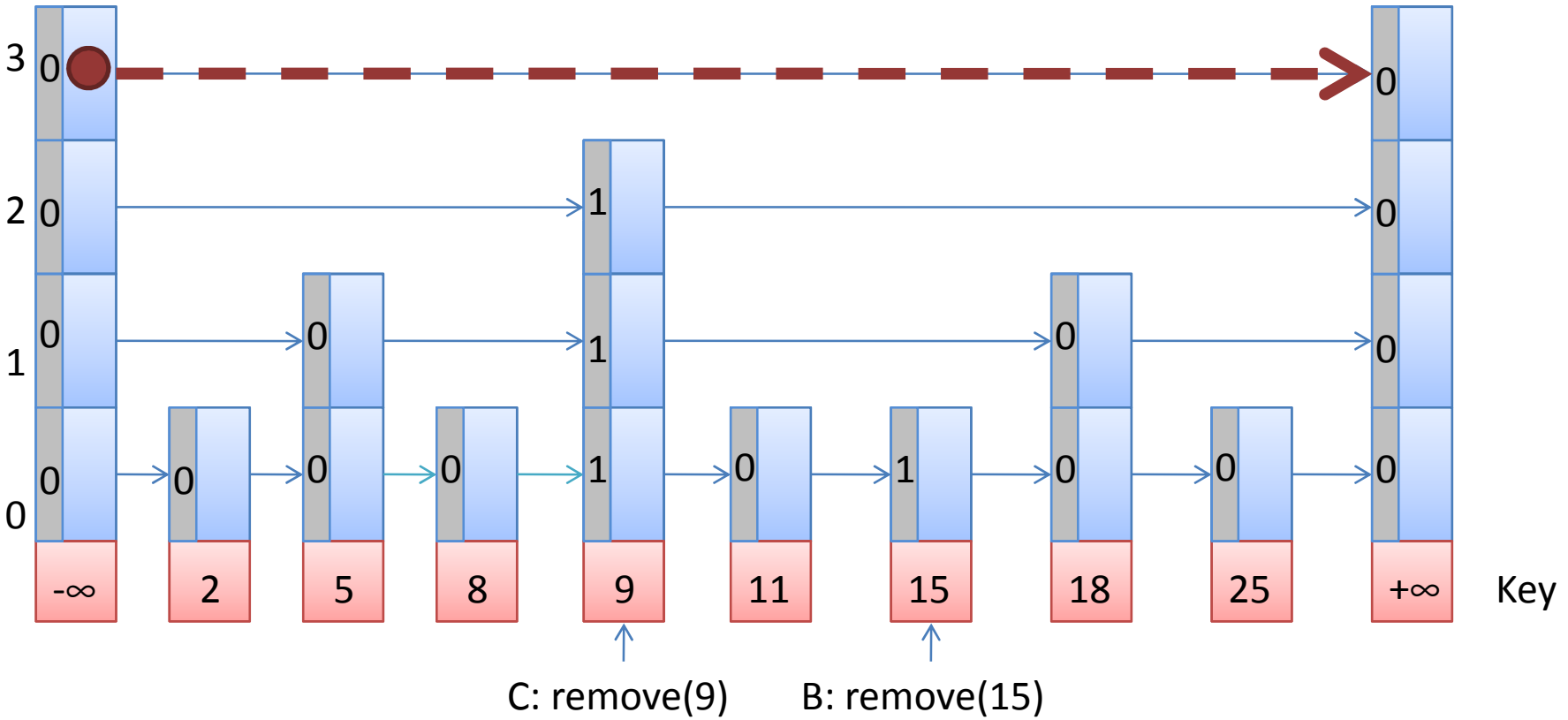
```
1  boolean add(T x){
2      if(Knoten bereits im Set)
3          return false;
4      else{
5          <erstellen neuen Knoten mit Key x und rand. Toplevel>
6          for(Bottom bis Top)
7              <Neuen Knoten mit allen Nachfolgern verlinken>
8
9          if(Vorgänger in Bottom-List zeigt noch auf Nachfolger in Bottom-List)
10             <Vorgänger in Bottom-List mit neuem Knoten verlinken>
11             <Knoten ist nun in Bottom-List komplett verlinkt>
12
13         for(Bottom+1 bis Top)
14             <Vorgänger mit Knoten verlinken>
15             <Knoten ist nun komplett verlinkt>
16             return true;
17     }
18 }
```

Remove()-Pseudocode

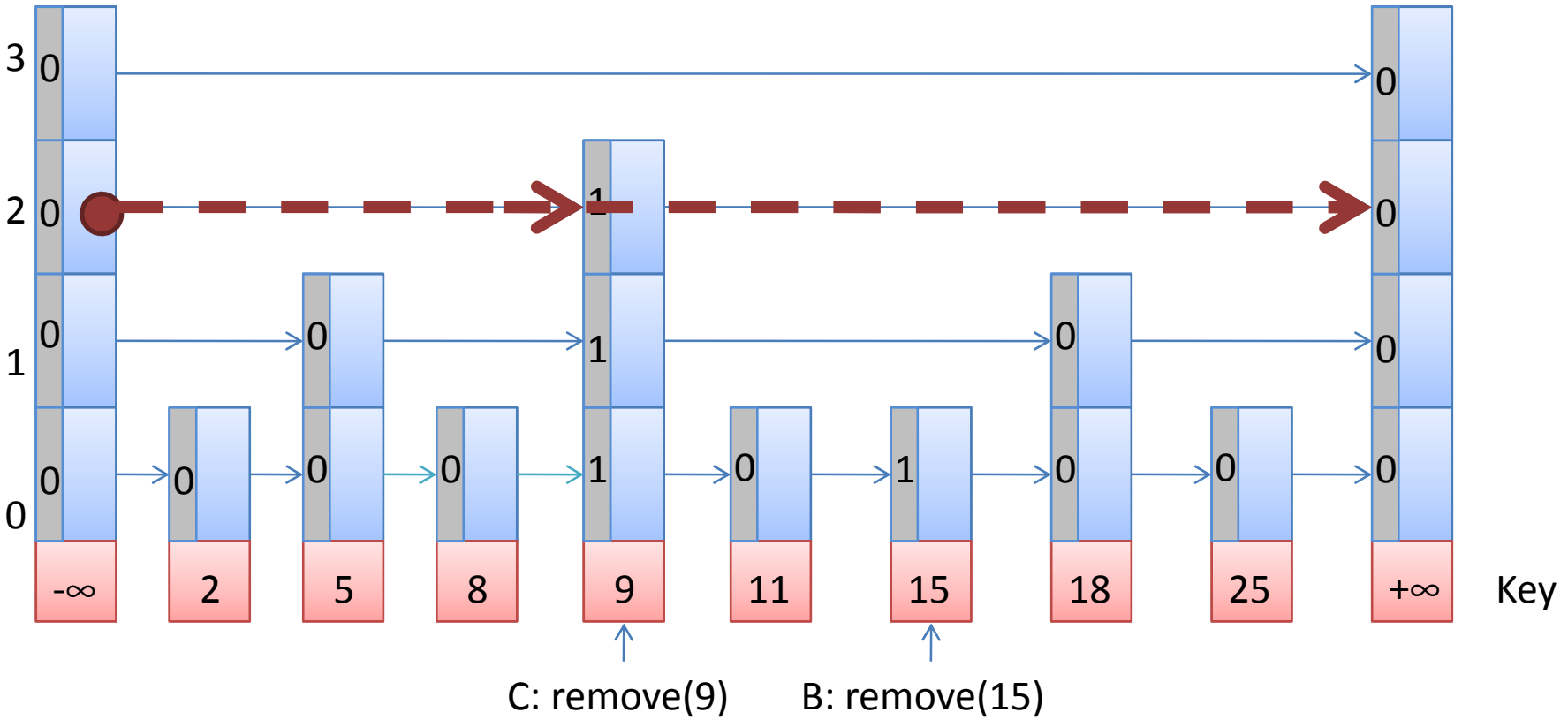
```
1  boolean remove(T x){
2      if(Knoten nicht im Set)
3          return false;
4      else{
5          for(Top bis Bottom+1)
6              <Markiere 'next' des zu löschenden Knotens>
7
8          <Markiere 'next' der Bottom-Ebene>
9          <Mit find() markierte Links löschen>
10         return true;
11     }
12 }
```

A: contains(18)

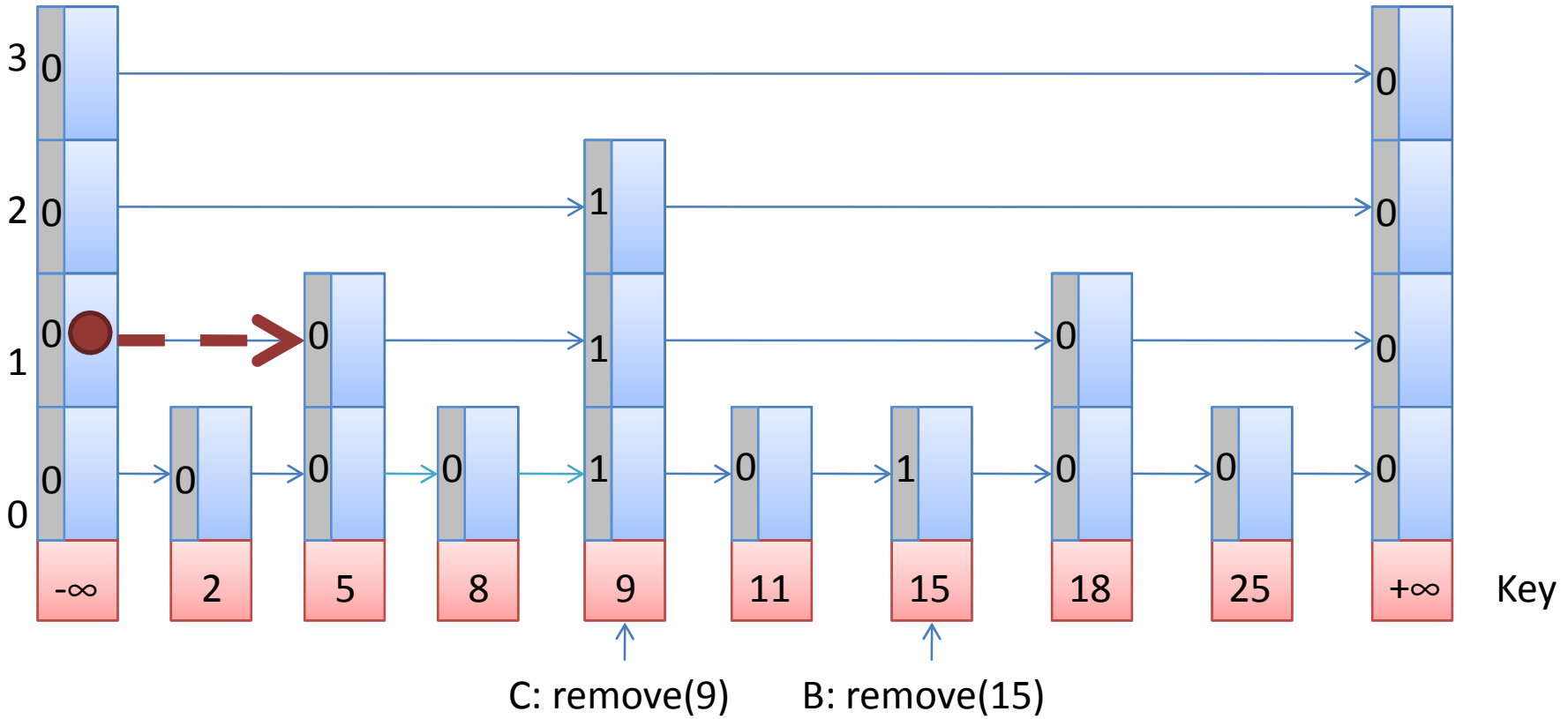
contains() ist wie find(), mit der Ausnahme, dass er die markierten next-Felder lediglich überspringt



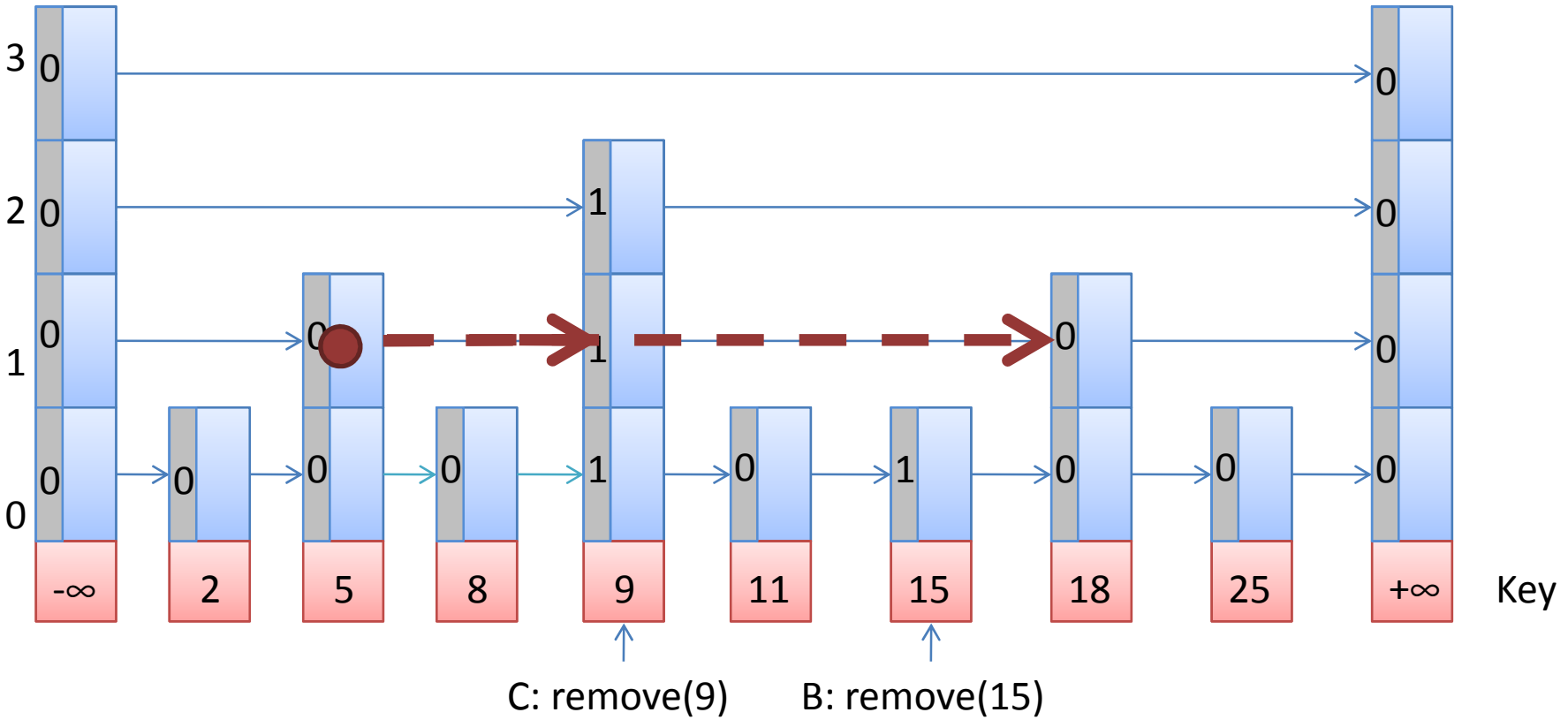
A: contains(18)



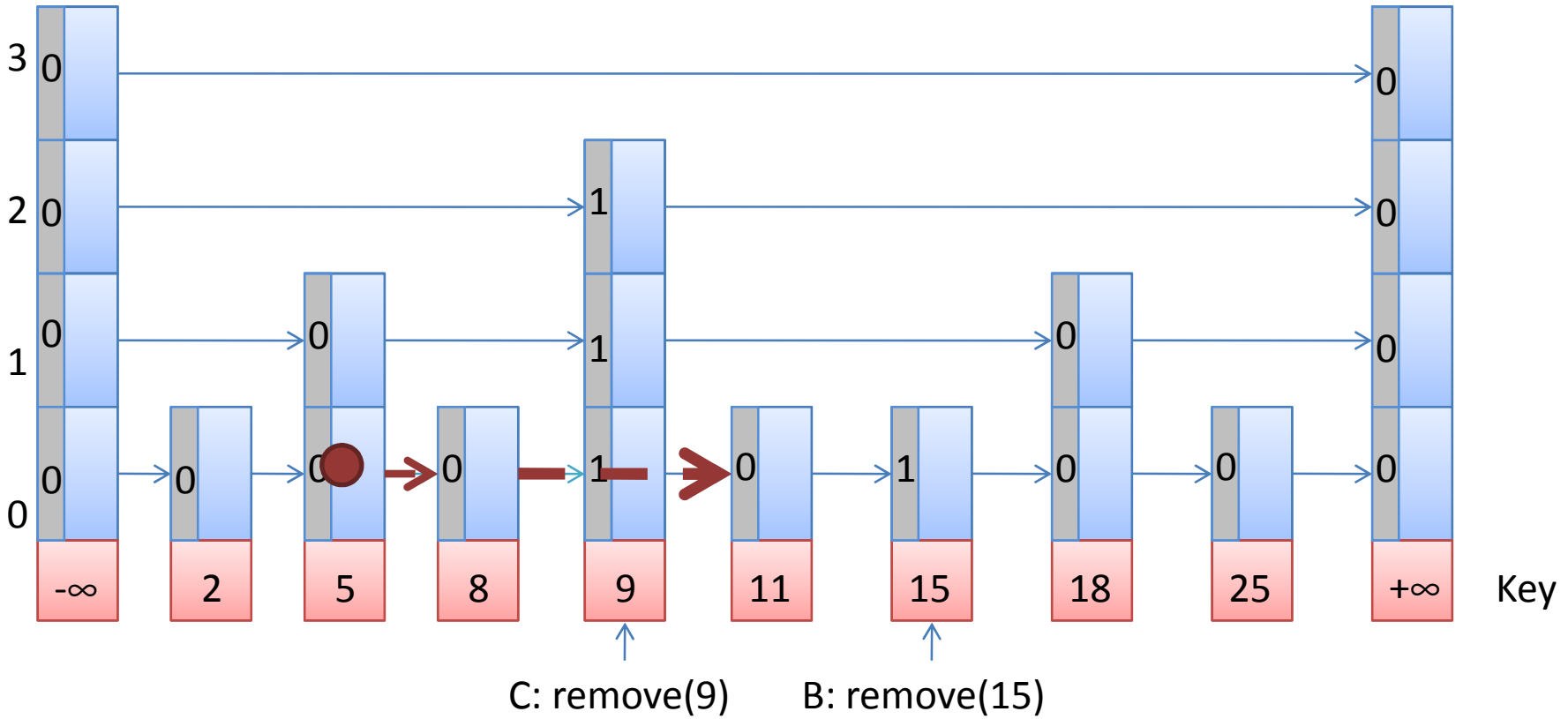
A: contains(18)



A: contains(18)

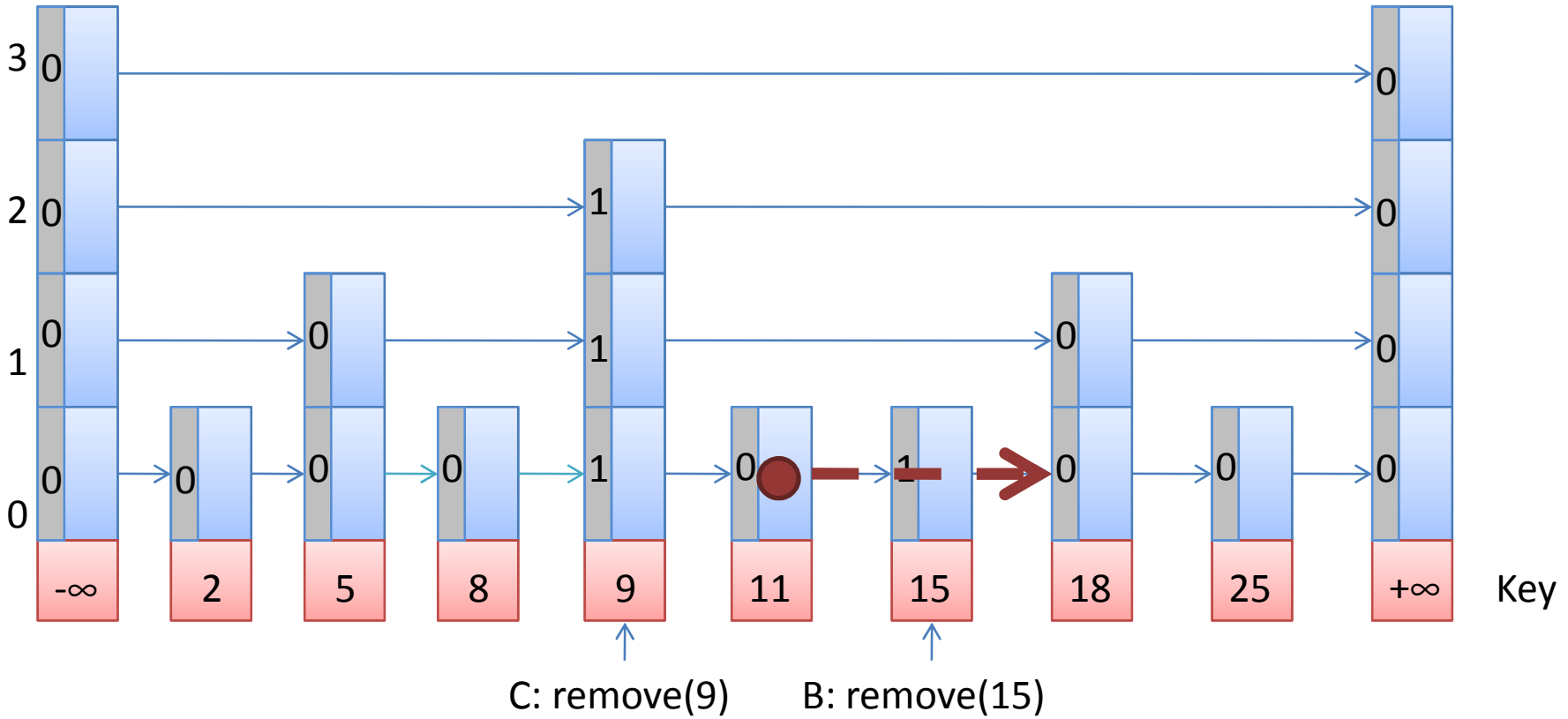


A: contains(18)



A: contains(18)

contains() vergleicht wie auch find() erst im BottomLevel den Nachfolger-Key mit dem Gesuchten – nicht schon vorher wie bei den LazySkipLists



Zusammenfassung

- Beide Varianten haben logarithmische Suchlaufzeit

  → Ohne Rebalancierung
- LazySkiplist
  - add() und remove() sind optimistic fine grained
- LockFreeSkiplist
  - Keine Locks, aber Skiplist-Eigenschaft kann nicht eingehalten werden
- contains() ist in beiden wait-free