LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Clara Lüling, Stephan Bittner

# Linked Lists – The Role of Locking

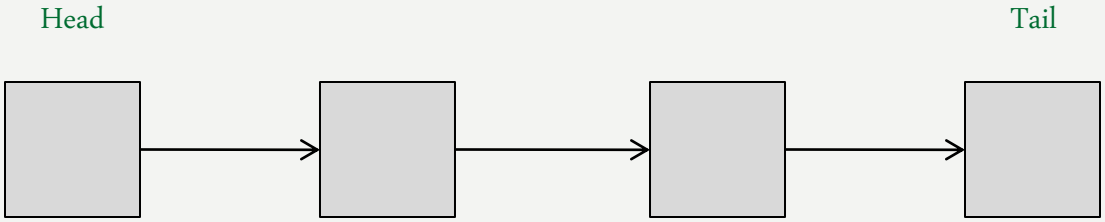
Verkettete Liste - Die Rolle des Sperrens

# Linked Lists – The Role of Locking

1. Verkettete Listen
2. Algorithmen
    1. Coarse-Grained Synchronization
    2. Fine-Grained Synchronization
    3. Optimistic Synchronization
    4. Lazy Synchronization
    5. Non-Blocking Synchronization
3. Abschließender Vergleich

## Funktionsschema

- Eine *verkettete Liste* besteht aus miteinander verlinkten Knoten (*Nodes):*



## Listen-Element *Node*

Ein Node enthält:

- das eigentliche Objekt (`item`)
- den Hashcode des Objekts als Key (`key`)
- eine Referenz auf den nachfolgenden Knoten (`next`)

```
1public class Node {
2
3    T item;
4    int key;
5    Node next;
6}
```

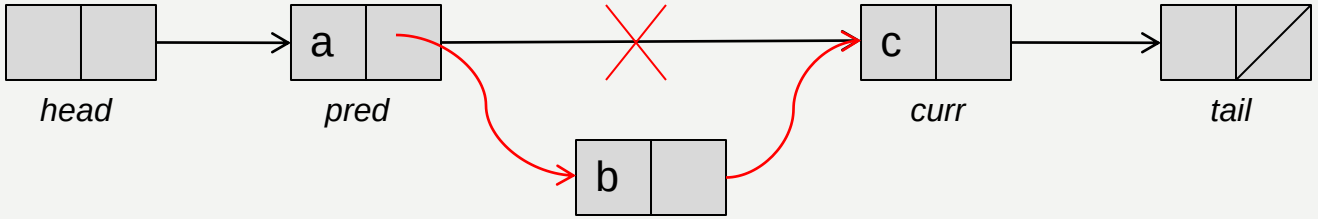
Ausnahme: head und tail sind sog. *sentinel nodes*: sie enthalten kein Objekt

# Manipulationsmethoden

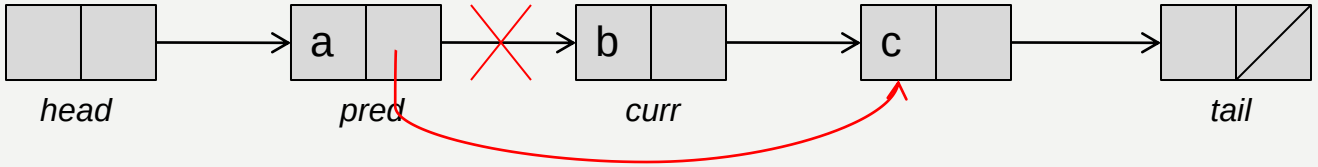
```
1 public interface Set<T> {
2     boolean add(T x);
3
4     boolean remove(T x);
5
6     boolean contains(T x);
7 }
```

# Manipulationsmethoden

Einfügen in List: add(b)




Entfernen von Knoten: remove(b)




Prüfung ob bereits vorhanden: contains(b)

# Algorithmen

→ **Coarse-Grained Synchronization**

- Fine-Grained Synchronization
- Optimistic Synchronization
- Lazy Synchronization
- Non-Blocking Synchronization

# Coarse-Grained Synchronization

Lock

kritischer Bereich

```
16    public boolean add(T item) {
17        Node pred, curr;
18        int key = item.hashCode();
19        lock.lock();
20        try {
21            pred = head;
22            curr = pred.next;
23            while (curr.key < key) {
24                pred = curr;
25                curr = curr.next;
26            }
27            if (key == curr.key) {
28                return false;
29            } else {
30                Node node = new Node(item);
31                node.next = curr;
32                pred.next = node;
33                return true;
34            }
35        } finally {
36            lock.unlock();
37        }
38    }
```

```
40    public boolean remove(T item) {
41        Node pred, curr;
42        int key = item.hashCode();
43        lock.lock();
44        try {
45            pred = head;
46            curr = pred.next;
47            while (curr.key < key) {
48                pred = curr;
49                curr = curr.next;
50            }
51            if (key == curr.key) {
52                pred.next = curr.next;
53                return true;
54            } else {
55                return false;
56            }
57        } finally {
58            lock.unlock();
59        }
60    }
```

## Coarse-Grained Synchronization

- Zugriff auf die Liste durch Lock der gesamten Liste
- Resultat: sequentielle Ausführung der Threads
- Vorteile: Algorithmus arbeitet korrekt
- Nachteil: nur ein Zugriff gleichzeitig auf die Liste

→sinnvoll nur bei wenigen nebenläufigen Threads, sonst hohe Wartezeiten

→ Wie können wir mehreren Threads gleichzeitig den Zugriff auf die Liste ermöglichen?

# Algorithmen

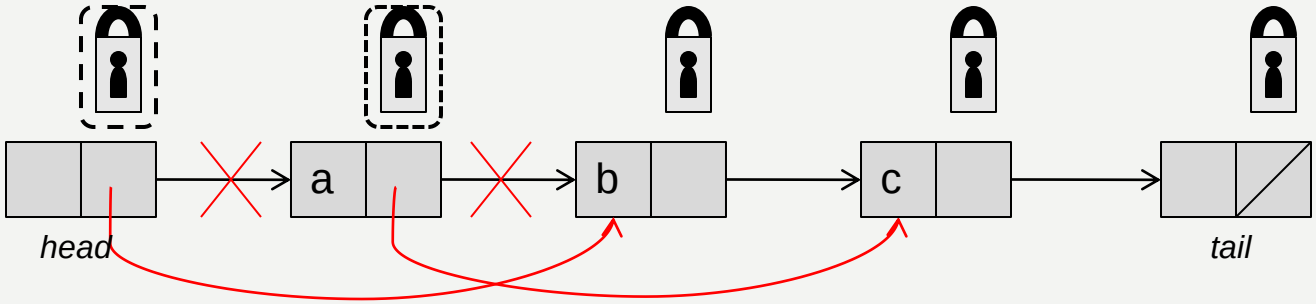
- Course-Grained Synchronization
- → **Fine-Grained Synchronization**
- Optimistic Synchronization
- Lazy Synchronization
- Non-Blocking Synchronization

# Fine-Grained Synchronization

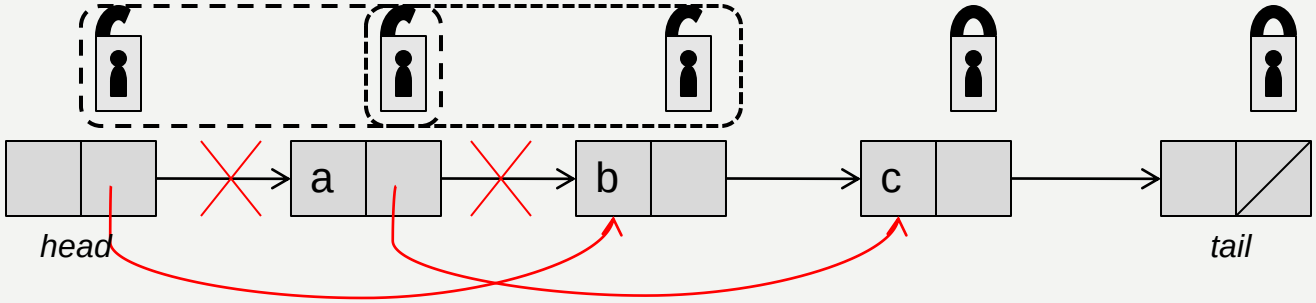
```
14  public boolean add(T item) {
15      int key = item.hashCode();
16      head.lock();
17      Node pred = head;
18      try {
19          Node curr = pred.next;
20          curr.lock();
21          try {
22              while (curr.key < key) {
23                  pred.unlock();
24                  pred = curr;
25                  curr = curr.next;
26                  curr.lock();
27              }
28              if (curr.key == key) {
29                  return false;
30              }
31              Node newNode = new Node(item);
32              newNode.next = curr;
33              pred.next = newNode;
34              return true;
35          } finally {
36              curr.unlock();
37          }
38      } finally {
39          pred.unlock();
40      }
41  }
```

```
43  public boolean remove(T item) {
44      Node pred = null, curr = null;
45      int key = item.hashCode();
46      head.lock();
47      try {
48          pred = head;
49          curr = pred.next;
50          curr.lock();
51          try {
52              while (curr.key < key) {
53                  pred.unlock();
54                  pred = curr;
55                  curr = curr.next;
56                  curr.lock();
57              }
58              if (curr.key == key) {
59                  pred.next = curr.next;
60                  return true;
61              }
62              return false;
63          } finally {
64              curr.unlock();
65          }
66      } finally {
67          pred.unlock();
68      }
69  }
```

## Fine-Grained Synchronization: Warum zwei Locks (*pred* und *curr*)?

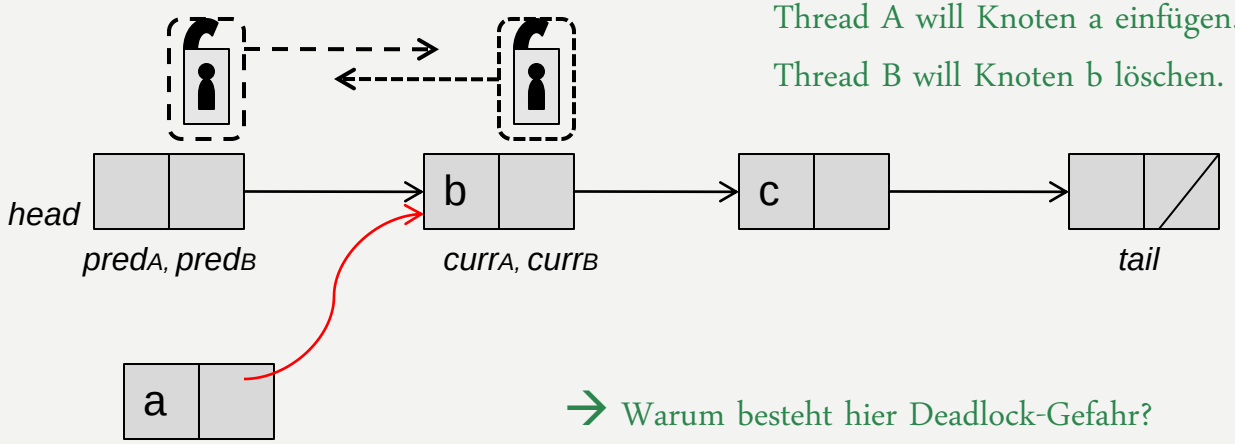



Thread A löscht a, B zur selben Zeit b. Was passiert?




Lösung: es müssen jeweils *pred* und *curr* gelockt werden.

# Fine-Grained Synchronization: Deadlock




Thread A will Knoten a einfügen.

Thread B will Knoten b löschen.

→ Warum besteht hier Deadlock-Gefahr?

Möglicher Ablauf:

A sichert b

B sichert head

A möchte head sichern: besetzt

B möchte b sichern: besetzt → Deadlock!

## Fine-Grained Synchronization

- Jeder Knoten ist durch einen eigenen Lock gesichert
- Nachteile:
  - Gefahr eines Deadlocks
  - Bei jedem Knoten muss die Freigabe angefordert werden

→ Wie können wir verhindern, dass auch nicht gesuchte Knoten gelockt werden müssen?

# Algorithmen

- Course-Grained Synchronization
- Fine-Grained Synchronization
- → **Optimistic Synchronization**
- Lazy Synchronization
- Non-Blocking Synchronization

} Lazy-Algorithmen

# Optimistic Synchronization

```
14    public boolean add(T item) {
41    }
```

```
43    public boolean remove(T item) {
69    }
```

```
71    public boolean contains(T item) {
91    }
```

```
93    private boolean validate(Entry pred, Entry curr) {
101   }
```

# Optimistic Synchronization

```
14    public boolean add(T item) {
15        int key = item.hashCode();
16        while (true) {
17            Entry pred = this.head;
18            Entry curr = pred.next;
19            while (curr.key <= key) {
20                pred = curr;
21                curr = curr.next;
22            }
23            pred.lock();
24            curr.lock();
25            try {
26                if (validate(pred, curr)) {
27                    if (curr.key == key) {
28                        return false;
29                    } else {
30                        Entry entry = new Entry(item);
31                        entry.next = curr;
32                        pred.next = entry;
33                        return true;
34                    }
35                }
36            } finally {
37                pred.unlock();
38                curr.unlock();
39            }
40        }
41    }
```

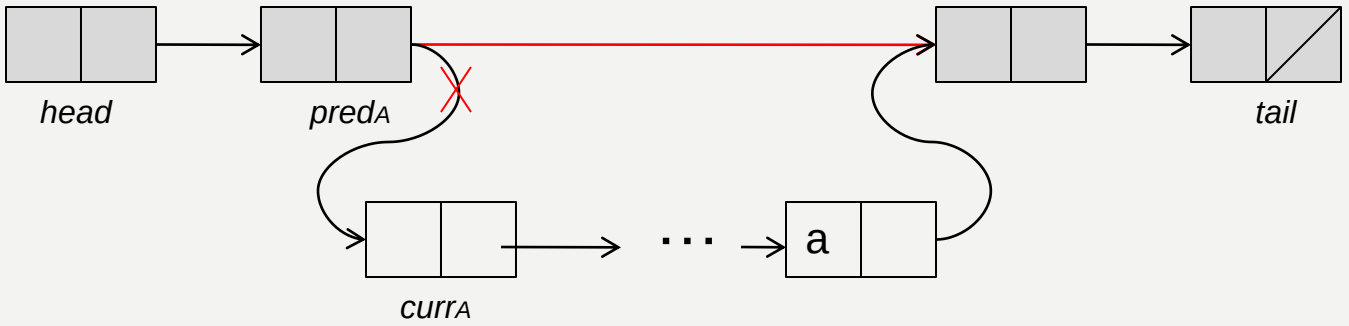
```
43    public boolean remove(T item) {
44        int key = item.hashCode();
45        while (true) {
46            Entry pred = this.head;
47            Entry curr = pred.next;
48            while (curr.key < key) {
49                pred = curr;
50                curr = curr.next;
51            }
52
53            pred.lock();
54            curr.lock();
55            try {
56                if (validate(pred, curr)) {
57                    if (curr.key == key) {
58                        pred.next = curr.next;
59                        return true;
60                    } else {
61                        return false;
62                    }
63                }
64            } finally {
65                pred.unlock();
66                curr.unlock();
67            }
68        }
69    }
```

# Optimistic Synchronization

```
71    public boolean contains(T item) {
72        int key = item.hashCode();
73        while (true) {
74            Entry pred = this.head;
75            Entry curr = pred.next;
76            while (curr.key < key) {
77                pred = curr;
78                curr = curr.next;
79            }
80            try {
81                pred.lock();
82                curr.lock();
83                if (validate(pred, curr)) {
84                    return (curr.key == key);
85                }
86            } finally {
87                pred.unlock();
88                curr.unlock();
89            }
90        }
91    }
```

```
93    private boolean validate(Entry pred, Entry curr) {
94        Entry entry = head;
95        while (entry.key <= pred.key) {
96            if (entry == pred)
97                return pred.next == curr;
98            entry = entry.next;
99        }
100       return false;
101   }
```

# Optimistic Synchronization: Warum *validate()* ?




Thread A will a löschen.

Während A die Liste durchquert, löscht Thread B alle Knoten von curr bis a.

A löscht trotzdem Knoten a.

Lösung: validate() checkt, ob a von head aus erreichbar ist.

## Optimistic Synchronization

- Lock() erst, wenn gesuchter Key gefunden wurde
- Abschließend validate()
- Nur optimal, wenn:

(Kosten für 2maliges Durchgehen der Liste) < < (Kosten für einmaliges Durchgehen der Liste mit Fine-grained Synchronisation)

- Nachteile:
  - Threads können verhungern
  - auch die contains()-Methode muss Knoten entsperren

→ Wie können wir verhindern, dass contains()-Calls warten müssen (wait-free)?

# Algorithmen

- Coarse-Grained Synchronization
- Fine-Grained Synchronization
- Optimistic Synchronization
- → **Lazy Synchronization**
- Non-Blocking Synchronization

} Lazy-Algorithmen

# Lazy Synchronization

```
14    public boolean add(T item) {
15        int key = item.hashCode();
16        while (true) {
43        }
44    }
```

```
46    public boolean remove(T item) {
47        int key = item.hashCode();
48        while (true) {
74        }
75    }
```

```
77    public boolean contains(T item) {
83    }
```

```
85    private boolean validate(Node pred, Node curr) {
87    }
```

# Lazy Synchronization

```
14  public boolean add(T item) {
15      int key = item.hashCode();
16      while (true) {
17          Node pred = head;
18          Node curr = head.next;
19          while (curr.key < key) {
20              pred = curr;
21              curr = curr.next;
22          }
23          pred.lock();
24          try {
25              curr.lock();
26              try {
27                  if (validate(pred, curr)) {
28                      if (curr.key == key) {
29                          return false;
30                      } else {
31                          Node node = new Node(item);
32                          node.next = curr;
33                          pred.next = node;
34                          return true;
35                      }
36                  }
37              } finally {
38                  curr.unlock();
39              }
40          } finally {
41              pred.unlock();
42          }
43      }
44  }
```

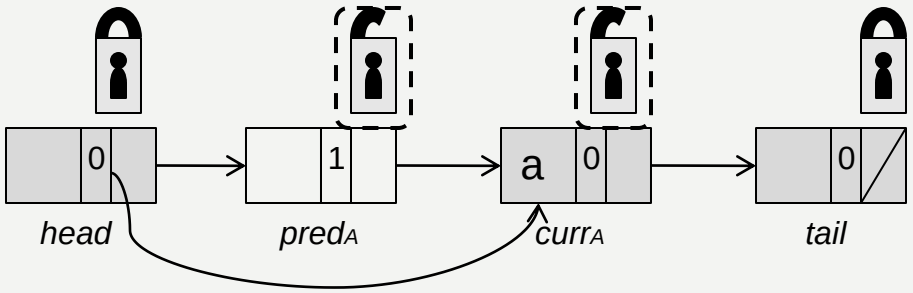
```
46  public boolean remove(T item) {
47      int key = item.hashCode();
48      while (true) {
49          Node pred = head;
50          Node curr = head.next;
51          while (curr.key < key) {
52              pred = curr;
53              curr = curr.next;
54          }
55          pred.lock();
56          try {
57              curr.lock();
58              try {
59                  if (validate(pred, curr)) {
60                      if (curr.key != key) {
61                          return false;
62                      } else {
63                          curr.marked = true;
64                          pred.next = curr.next;
65                          return true;
66                      }
67                  }
68              } finally {
69                  curr.unlock();
70              }
71          } finally {
72              pred.unlock();
73          }
74      }
75  }
```

# Lazy Synchronization

```
77    public boolean contains(T item) {
78        int key = item.hashCode();
79        Node curr = head;
80        while (curr.key < key)
81            curr = curr.next;
82        return curr.key == key && !curr.marked;
83    }
```

```
85    private boolean validate(Node pred, Node curr) {
86        return !pred.marked && !curr.marked && pred.next == curr;
87    }
```

## Lazy Synchronization: Warum *validate()* ?

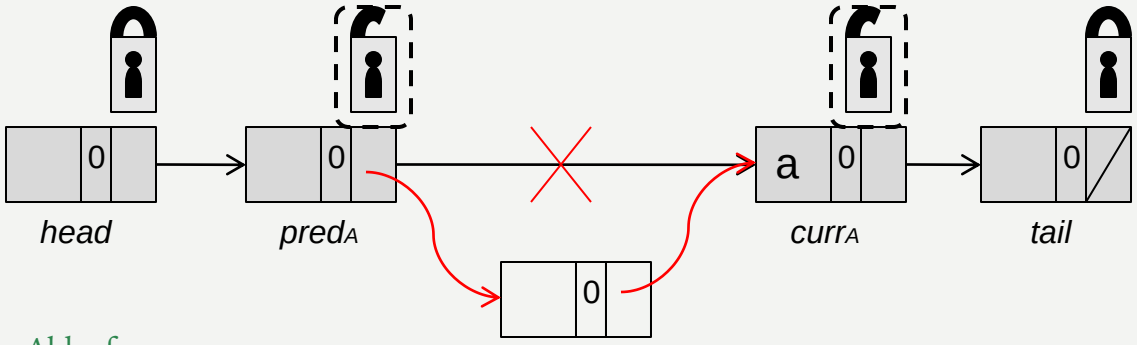



Ablauf:

Thread A erreicht gesuchten Knoten.

Knoten predA wird von anderem Thread gelöscht.

-> validate() weist A auf dieses Problem hin




Ablauf:

Thread A erreicht gesuchten Knoten. Neuer Knoten wird von anderem Thread eingefügt.

-> validate() weist A auf dieses Problem hin -> remove()-Call von A wird neu gestartet

# Lazy Synchronization

- Sorgt dafür, dass contains()-Anfragen nicht warten müssen:
- Remove()-Methode ist „lazy“ und löscht in 2 Schritten:
  - setzt marked-field des Knotens auf true
  - ersetzt anschließend den Zeiger auf den Knoten durch den auf den nachfolgenden Knoten.
- Contains()-Aufruf gibt nun false zurück, wenn Knoten nicht vorhanden oder „markiert“ ist → keine Locks mehr nötig
- Nachteil: add()- und remove()-Aufrufe können sich gegenseitig blockieren

# Algorithmen

- Coarse-Grained Synchronization
- Fine-Grained Synchronization
- Optimistic Synchronization
- Lazy Synchronization
- → **Non-Blocking Synchronization**

} Lazy-Algorithmen

# Non-Blocking Synchronization

```
82    public class Window {
92    }
```

```
94     private class Node {
111    }
```

```
14    public boolean add(T item) {
15        int key = item.hashCode();
16        while (true) {
43        }
44    }
```

```
46    public boolean remove(T item) {
47        int key = item.hashCode();
48        while (true) {
74        }
75    }
```

```
77     public boolean contains(T item) {
83     }
```

# Non-Blocking Synchronization

```
94    private class Node {
95
96        T item;
97        int key;
98        AtomicMarkableReference<Node> next;
99
100       Node(T item) {
101           this.item = item;
102           this.key = item.hashCode();
103           this.next = new AtomicMarkableReference<Node>(null, false);
104       }
105
106       Node(int key) {
107           this.item = null;
108           this.key = key;
109           this.next = new AtomicMarkableReference<Node>(null, false);
110       }
111   }
```

java.util.concurrent.atomic

## Class AtomicMarkableReference<V>

java.lang.Object
  └ java.util.concurrent.atomic.AtomicMarkableReference<V>

**Type Parameters:**

V - The type of object referred to by this reference

# Non-Blocking Synchronization

## compareAndSet

```
public boolean compareAndSet(V expectedReference,
                             V newReference,
                             boolean expectedMark,
                             boolean newMark)
```

Atomically sets the value of both the reference and mark to the given update values if the current reference is == to the expected reference and the current mark is equal to the expected mark.

**Parameters:**

`expectedReference` - the expected value of the reference
`newReference` - the new value for the reference
`expectedMark` - the expected value of the mark
`newMark` - the new value for the mark

**Returns:**

true if successful

# Non-Blocking Synchronization

```
82    public class Window {
83
84        public Node pred;
85
86        public Node curr;
87
88        Window(Node pred, Node curr) {
89            this.pred = pred;
90            this.curr = curr;
91        }
92    }
```

```
51     public boolean contains(T item) {
52         int key = item.hashCode();
53         Window window = find(head, key)
54         Node pred = window.pred, curr =
55         return (curr.key == key);
56     }
```

# Non-Blocking Synchronization

```
58    public Window find(Node head, int key) {
59        Node pred = null, curr = null, succ = null;
60        boolean[] marked = { false };
61        boolean snip;
62        retry: while (true) {
63            pred = head;
64            curr = pred.next.getReference();
65            while (true) {
66                succ = curr.next.get(marked);
67                while (marked[0]) {
68                    snip = pred.next.compareAndSet(curr, succ, false, false);
69                    if (!snip)
70                        continue retry;
71                    curr = succ;
72                    succ = curr.next.get(marked);
73                }
74                if (curr.key >= key)
75                    return new Window(pred, curr);
76                pred = curr;
77                curr = succ;
78            }
79        }
80    }
```

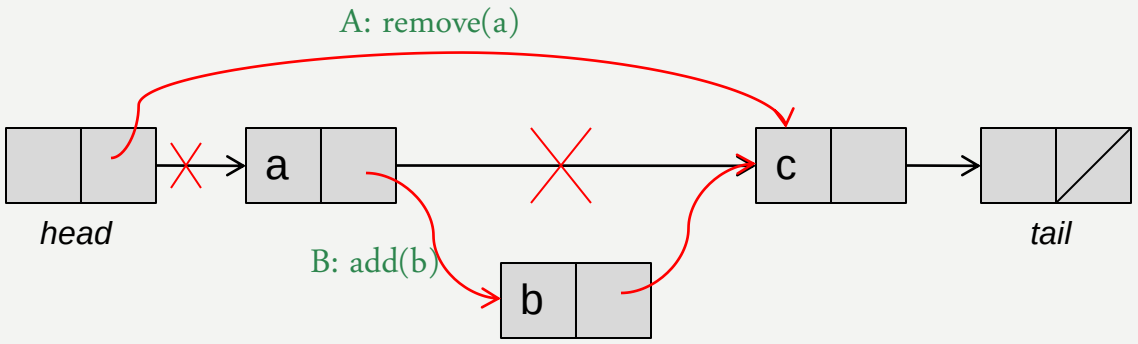
# Non-Blocking Synchronization

```
14    public boolean add(T item) {
15        int key = item.hashCode();
16        boolean splice;
17        while (true) {
18            Window window = find(head, key);
19            Node pred = window.pred, curr = window.curr;
20            if (curr.key == key) {
21                return false;
22            } else {
23                Node node = new Node(item);
24                node.next = new AtomicMarkableReference(curr, false);
25                if (pred.next.compareAndSet(curr, node, false, false)) {
26                    return true;
27                }
28            }
29        }
30    }
```

# Non-Blocking Synchronization

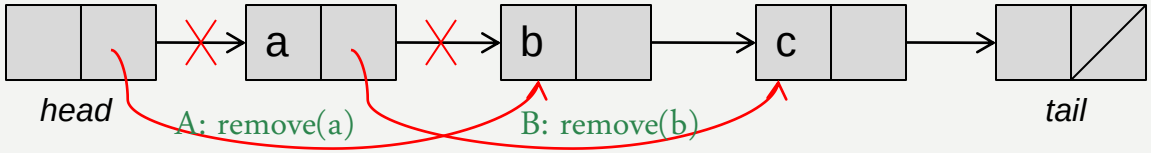
```
32    public boolean remove(T item) {
33        int key = item.hashCode();
34        boolean snip;
35        while (true) {
36            Window window = find(head, key);
37            Node pred = window.pred, curr = window.curr;
38            if (curr.key != key) {
39                return false;
40            } else {
41                Node succ = curr.next.getReference();
42                snip = curr.next.attemptMark(succ, true);
43                if (!snip)
44                    continue;
45                pred.next.compareAndSet(curr, succ, false, false);
46                return true;
47            }
48        }
49    }
```

# Non-Blocking Synchronization: Grund für atomare Änderung




Thread A ruft compareAndSet() für head.next auf

Thread B ruft gleichzeitig compareAndSet() für a.next auf → b wird nicht in die Liste eingefügt




Thread A ruft compareAndSet() für head.next auf

Thread B ruft gleichzeitig compareAndSet() für a.next auf → b wird nicht gelöscht

## Non-Blocking Synchronization

- CompareAndSet()-Calls allein nicht ausreichend für korrekte Ausführung
- Next- und marked-field müssen wie *ein* Feld behandelt werden (AtomicMarkableReference<T>)

  →stellt sicher, dass ein Knoten nicht geändert wird, nachdem er phys. oder log. gelöscht wurde

- Vorteil: weniger willkürliche Wartezeiten
- Nachteil: Aufwandkosten für Ermöglichung der atomaren Änderung und einen Boolean Wert

## Vergleich

| Coarse-Grained | Fine-Grained | Optimistic | Lazy | Non-Blocking |
|---|---|---|---|---|
| Single Lock | | | | |
| sequentielle → | parallel | | | |
| | Lock bei jedem → | Lock von relevanten | | |
| | Knoten | Knoten | | |
| | | contains() → | Marked – field | |
| | | benötigt Locks | | |
| | | | | ↔ Kein Locking: compareAndSet() + atomares Ändern |

# Fragen ?

# Vielen Dank für eure Aufmerksamkeit!