LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Michael Stockerl, Christian Simon

Concurrent Hashing

Proseminar
Nebenläufige Programmierung
SS 2010

Gliederung

Einleitung: Wozu Hash Sets?

Nachteile bei Linked Lists:

- Hoher Synchronisationsaufwand
- Häufigste Operation ist contains() (90%) die benötigt bei Linked Lists lineare Laufzeit

→alternative Datenstruktur nötig

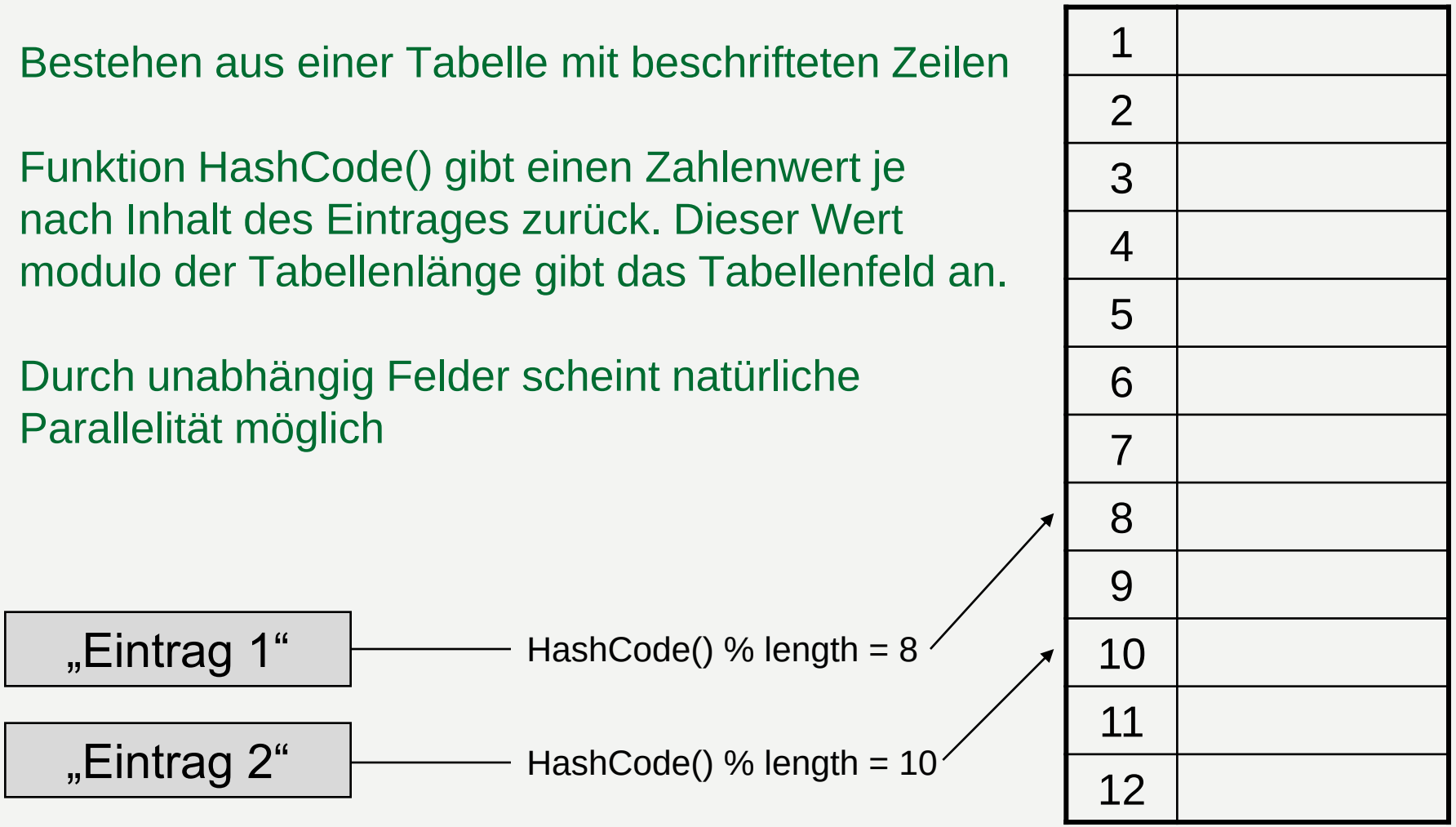
Es bieten sich Hash Sets an

Was sind Hash Sets?

Bestehen aus einer Tabelle mit beschrifteten Zeilen

Funktion HashCode() gibt einen Zahlenwert je nach Inhalt des Eintrages zurück. Dieser Wert modulo der Tabellenlänge gibt das Tabellenfeld an.

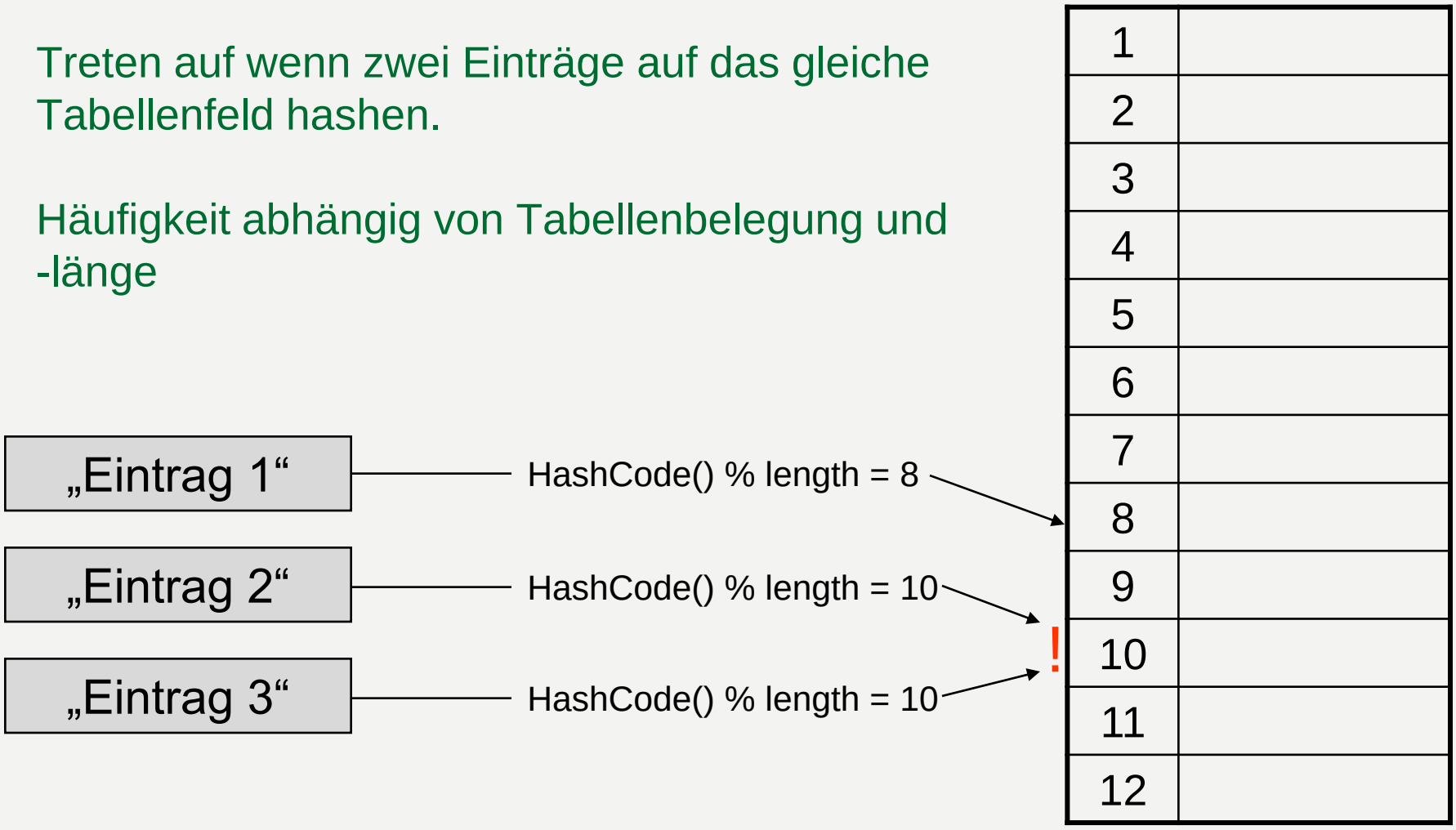
Durch unabhängig Felder scheint natürliche Parallelität möglich



Kollisionen

Treten auf wenn zwei Einträge auf das gleiche Tabellenfeld hashen.

Häufigkeit abhängig von Tabellenbelegung und -länge



Tabellengröße

Je höher die Belegung einer Tabelle ist, desto höher die Wahrscheinlichkeit einer Kollission.

Daher muss ab einer bestimmten Belegung (`policy()`) die Tabelle vergrößert werden. Dies geschieht mit der `resize()` Methode.

Da die Löschoperationen seltener Vorkommen als Hinzufügeoperationen. Wird davon ausgegangen das Tabellen nur wachsen.

Der Resize der Tabelle erfordert wesentlich mehr Rechenzeit als die übrigen Operationen

Kollisionsbehandlung

Es gibt unterschiedliche Strategien

Geschlossene Adressierung
- Ein Tabellenfeld kann mehrere Einträge in einer Liste enthalten

Offene Adressierung
- Jedes Tabellenfeld enthält max. einen Eintrag
- Bei Kollision wird eine andere Hashfunktion verwendet

Geschlossen adressierte Hash Sets

```
public abstract class BaseHashSet<T> {                          // Abstract HashSet
      protected List<T>[] table;
      protected int size;

      public BaseHashSet(int capacity) {                        // Größe zu Beginn
            size = 0;
            table = (List<T>[]) new List[capacity];
            for (int i = 0; i < capacity; i++) {
                  table[i] = new ArrayList<T>();                // Schubladen sind Listen
            }
      }

      public boolean contains(T x) {
            acquire(x);                                         // Hole Lock
            try {
                  int myBucket = Math.abs(x.hashCode() % table.length);   // Berechne Schublade
                  return table[myBucket].contains(x);           // ist x in Schublade ??
            } finally {
                  release(x);                                   // Entferne Lock immer
            }
      }

      public boolean add(T x) {
            boolean result = false;
            acquire(x);                                         // Hole Lock
            try {
                  int myBucket = Math.abs(x.hashCode() % table.length);   // Berechne Schublade
                  result = table[myBucket].add(x);              // Füge x in der Schublade hinzu
                  size = result ? size + 1 : size;
            } finally {
                  release(x);                                   // Entferne Lock immer
            }
            ...
```

Geschlossen adressierte Hash Sets

```
        if (policy())
        resize();                                          // Wenn nötig vergrößere Hash Set
        return result;
    }

    public boolean remove(T x) {
        acquire(x);                                        // Hole Lock
        try {
            int myBucket = Math.abs(x.hashCode() % table.length);   // Berechne Schublade
            boolean result = table[myBucket].remove(x);    // Entferne x aus der Schublade
            size = result ? size - 1 : size;
            return result;
        } finally {
            release(x);                                    // Entferne Lock immer
        }
    }

    public abstract void acquire(T x);                     // Lock für Element x
    public abstract void release(T x);                     // Unlock für Element x
    public abstract void resize();                         // Resize Hash Set
    public abstract boolean policy();                      // Resize notwendig?
}
```

Coarse Grained Hash Sets (Ein Lock)

Beim sog. Coarse-Grained Hash Set wird bei jeder Operation auf dem Hash Set das komplette HashSet **gesperrt**.

Sobald das Belegungsverhältnis einen **festen Schwellwert** erreicht, wird die Tabelle auf die **doppelte Größe** vergrößert.

Beim Vergrößern ist **erneutes Hashing** aller Einträge nötig. Die Alte Tabelle wird dabei einfach in die Neue kopiert

Coarse Grained Hash Sets (Ein Lock)

```
public class CoarseHashSet<T> extends BaseHashSet<T>{
      final Lock lock;

      CoarseHashSet(int capacity) {
            super(capacity);
            lock  = new ReentrantLock();  // Erstelle Lock
      }

      public final void acquire(T x) {
            lock.lock();                  // Locke komplettes Set
      }
      public void release(T x) {
            lock.unlock();                // Unlocke komplettes Set
      }
      public boolean policy() {
            return size / table.length > 4;
                                          // Bedingung für Resize
      }
      ...
}
```

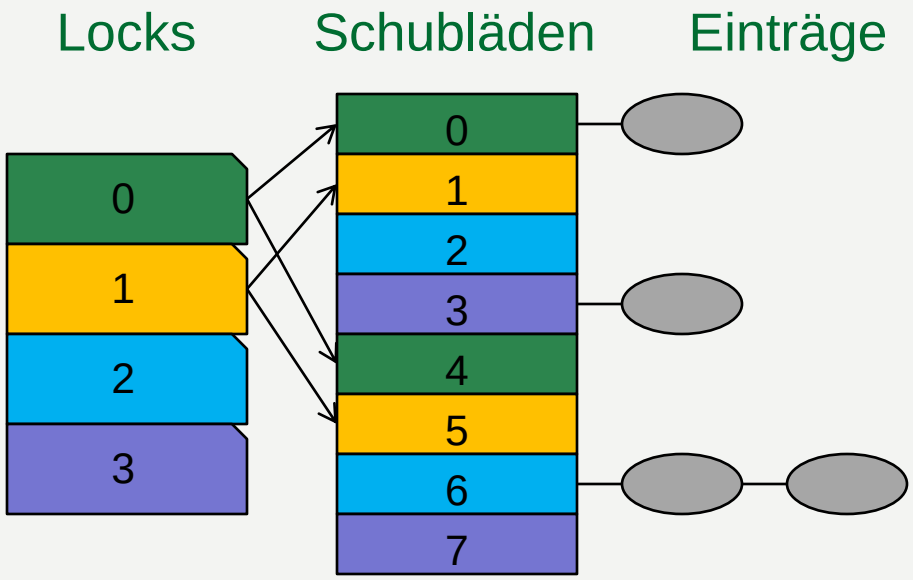
Ein einzelner Zugriff sperrt die komplette Hash Tabelle

→ **Sequentieller Ablauf der Zugriffe**

Striped Hash Sets (Statische Größe der Locktabelle)

Das Striped Hash Set hat neben der Tabelle noch **Locktabelle**, diese Locktabelle wird allerdings **nicht vergrößert** und besitzt zu Beginn die Größe des Hash Sets.

Das Vergrößern läuft ähnlich wie beim Coarse-Grained Hash Set, lediglich ist drauf zu achten, dass dabei alle Locks gesperrt werden müssen. Durch Speichern der Größe wird redundantes Vergrößern verhindert.



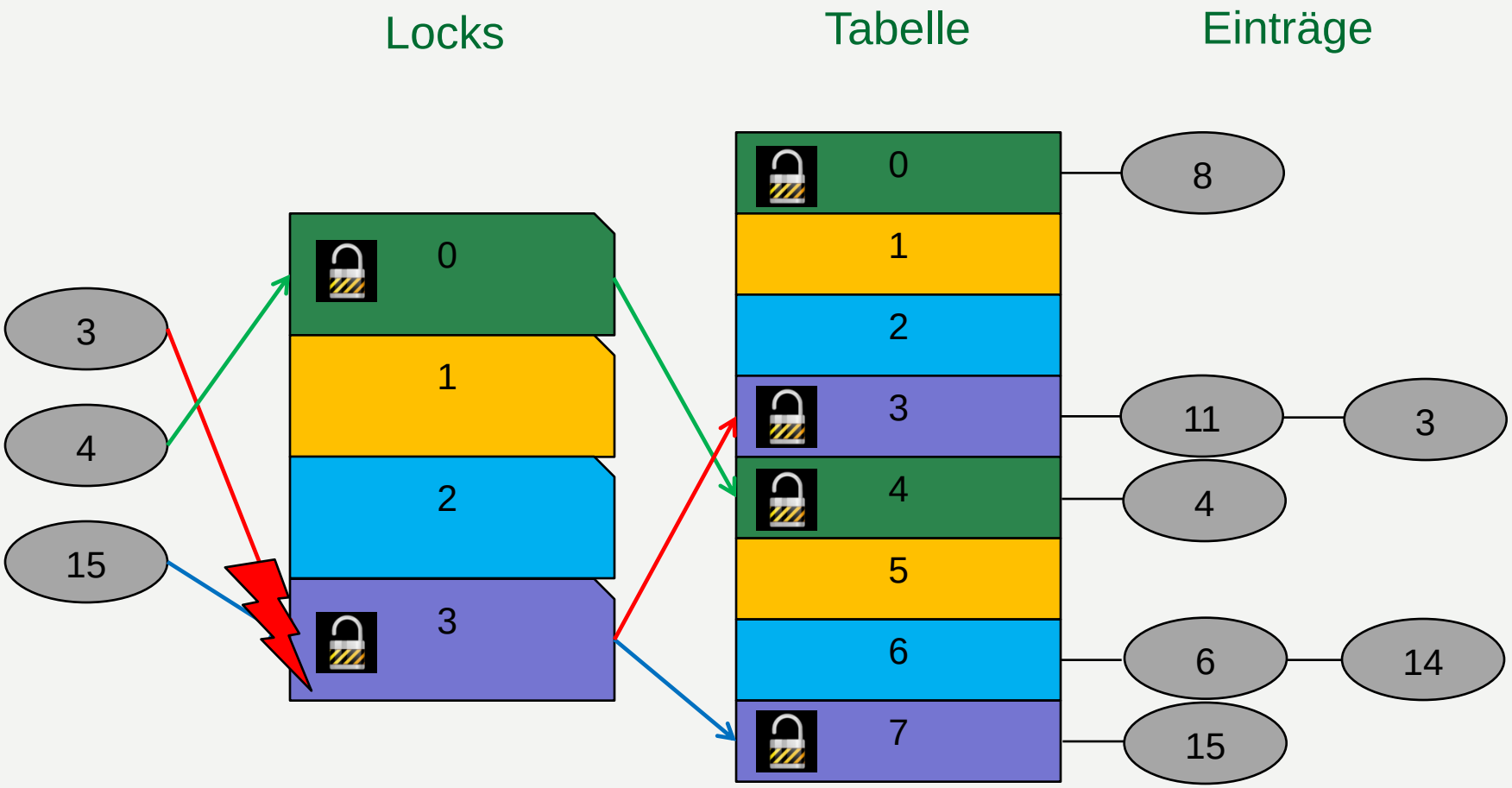
Striped Hash Sets (Statische Größe der Locktabelle)

```
public class StripedHashSet<T> extends BaseHashSet<T>{
      final Lock[] locks;
      public StripedHashSet(int capacity) {
            super(capacity);
            locks  = new Lock[capacity];                    //Locktabelle so groß wie Kapazität
            for (int j = 0; j < locks.length; j++) {
                  locks[j] = new ReentrantLock();           //Lege Locks an
            }
      }

      public final void acquire(T x) {
            int myBucket = Math.abs(x.hashCode() % locks.length);
            locks[myBucket].lock();                         //Lock nur was auf HashCode passt
      }

      public void release(T x) {
            int myBucket = Math.abs(x.hashCode() % locks.length);
            locks[myBucket].unlock();                       //Unlock worauf HashCode passt
      }
}
```

Dies sorgt für hohe Parallelität bei `add()`,`contains()` und `remove()` Aufrufen. Allerdings werden auch hier beim **Vergrößern** alle **anderen Threads ausgesperrt**.

Je öfter vergrößert wird, desto schlechter ist die Performance, da die Locktabelle nicht mit wächst.

Beispiel: Einfügen von Einträgen



Refineable Hash Sets (Variable Größe der Locktabelle)

Beim Refinable Hash Set **wächst** die **Locktabelle** beim Vergrößern des Hash Sets **mit**.

```
public class RefinableHashSet<T> extends BaseHashSet<T>{
      AtomicMarkableReference<Thread> owner;                    // Zeigt auf Resize Thread
      volatile ReentrantLock[] locks;                           // Lockarray jetzt volatile

      public RefinableHashSet(int capacity) {
            super(capacity);
            locks = new ReentrantLock[capacity];
            for (int j = 0; j < capacity; j++) {
            locks[j] = new ReentrantLock();
            }
            owner = new AtomicMarkableReference<Thread>(null, false);   // owner -> kein Resize findet
                                                                           statt
      }
      ...
}
```

In der AtomicMarkableReference `owner` wird der Thread gespeichert der gerade vergrößert und im Mark ob überhaupt vergrößert wird.

Refineable Hash Sets (Variable Größe der Locktabelle)

```
public void acquire(T x) {
      boolean[] mark = {true};
      Thread me = Thread.currentThread();
      Thread who;
      while (true) {
            do {                                    // Warte auf Ende des Resizen
                  who = owner.get(mark);
            } while (mark[0] && who != me);
            ReentrantLock[] oldLocks = this.locks;
            int myBucket = Math.abs(x.hashCode() % oldLocks.length);
            ReentrantLock oldLock = oldLocks[myBucket];
            oldLock.lock();  // acquire lock
            who = owner.get(mark);
            if ((!mark[0] || who == me) && this.locks == oldLocks) {
                                                    // Test auf Änderung am Lockarray
                  return;
            } else {                                // Abbruch und erneuter Versuch
                  oldLock.unlock();
            }
      }
}
```

Refineable Hash Sets (Variable Größe der Locktabelle)

```
protected void quiesce() {
      for (ReentrantLock lock : locks) {
            while (lock.isLocked()) {}                    // Warte bis alle Locks freigegeben sind
      }
}

public void resize() {
      int oldCapacity = table.length;
      int newCapacity = 2 * oldCapacity;
      Thread me = Thread.currentThread();
      if (owner.compareAndSet(null, me, false, true)) {   // Setze/checke owner
            try {
                  if (table.length != oldCapacity) {      // Ein Anderer war schneller
                        return;
                  }
                  ...

            }
            finally {
                  owner.set(null, false);                 // Setze alten Status bei owner
            }
      }
}
```

Bisheriges Problem:

Resize Operation stoppt alle Zugriffe auf das Hash Set für längere Zeit.

Idee:

- Resize inkrementell => keine explizite resize-Methode
- keine Locks
- Synchronisation mit compareAndSet()

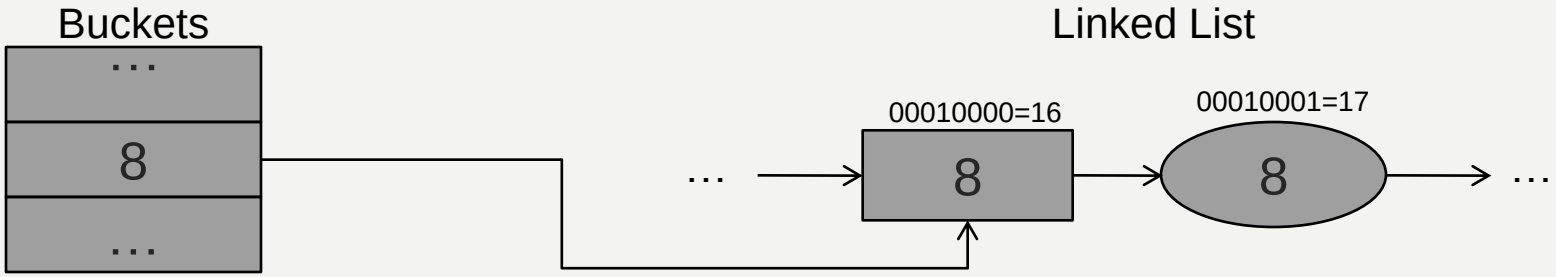
Recursive Split-Ordering

Anstatt die Elemente zwischen den einzelnen Buckets zu verschieben, werden die Buckets verschoben.

- Elemente in einer Liste (ordinary nodes)
- Bucket ist nur eine Referenz
  - Buckets sind Verweise auf bestimmte Knoten(sentinel nodes)
  - gewährleistet konstante Zeit
- Mehr Elemente → mehr Referenzen

Ordnung auf der Linked List

Symbol	Art	Wert	Bit	Rev. Bit	
8	Sentinel	8	00001000	00010000	00010000=16
8	Ordinary	8	00001000	00010000	00010001=17



Elemente sind in der umgekehrten Bitreihenfolge sortiert.
Trennung von Sentinel und Ordinary Nodes anhand des letzten Bits:

Sentinel Node: least significant bit = 0
Ordinary Node: least significant bit = 1

Inkrementelle Vergrößerung

Keine explizite Resize() Operation

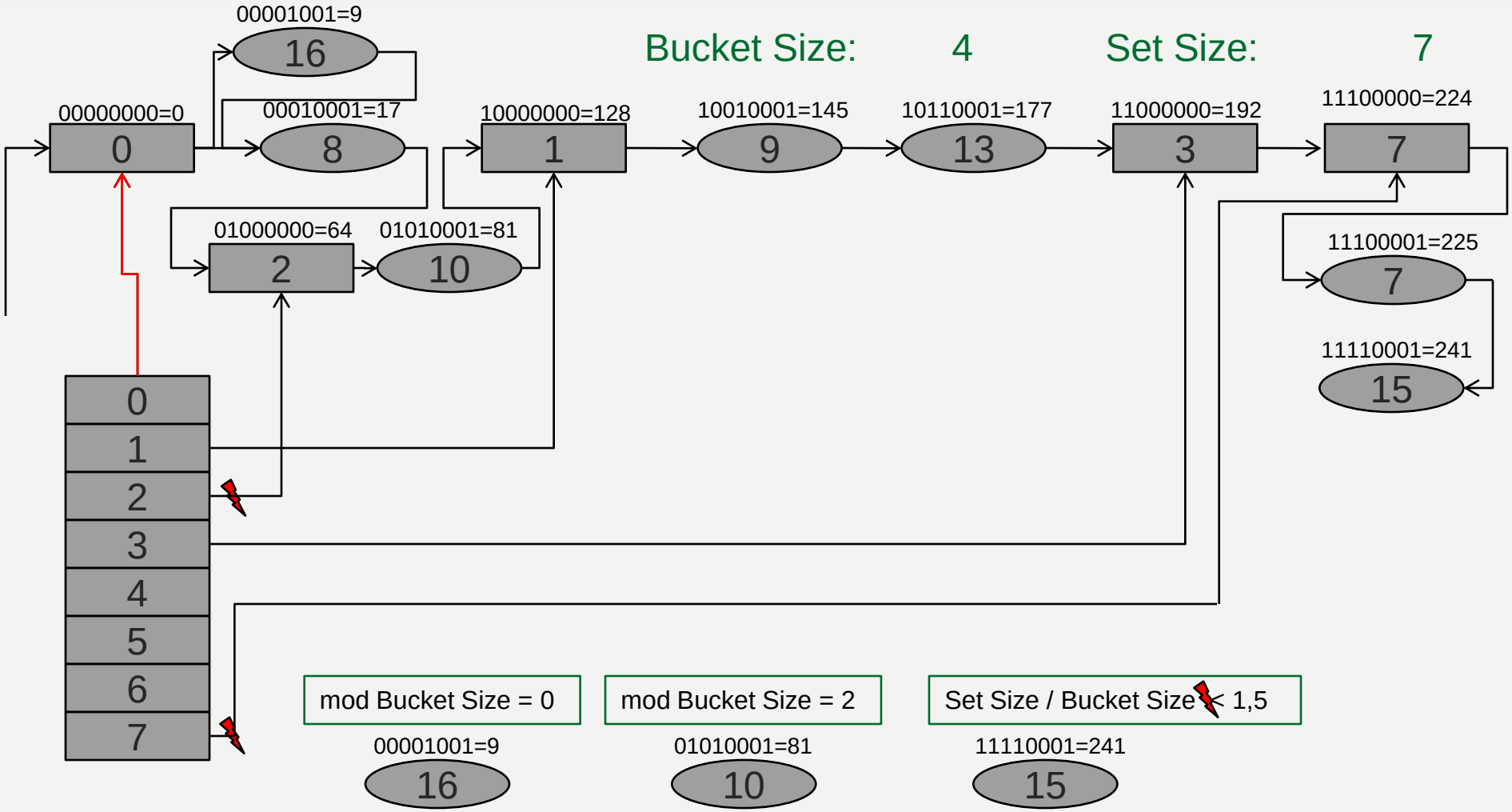
Vergrößerung findet beim Einfügen statt

Wenn Verhältnis von **Set Size** zu **Bucket Size** Schwellwert übersteigt
⇨ Bucket Size wird verdoppelt

Bucket Size: Anzahl aktiver Buckets (2er Potenzen)

Set Size: Anzahl der Ordinary Nodes

Beispiel einfügen in LockFreeList



BucketList Class

Implementiert die LockFreeList

2 Unterschiede:

- sortiert in der umgekehrten Bitreihenfolge
- Sentinel Node zu Beginn eines jeden Bucket

```
  static final int HI_MASK = 0x00800000;
  static final int MASK = 0x00FFFFFF;

public int makeRegularKey(T x) {
    int code = x.hashCode() & MASK;
    return reverse(code | HI_MASK);
  }
  private int makeSentinelKey(int key) {
    return reverse(key & MASK);
  }
```

```
public boolean contains(T x) {
   int key = makeRegularKey(x);
   Window window = find(head, key);
   Node pred = window.pred;
   Node curr = window.curr;
   return (curr.key == key);
 }
```

LockFreeHashSet<T> Class

```
 public LockFreeHashSet(int capacity) {
    bucket = (BucketList<T>[]) new BucketList[capacity];
    bucket[0] = new BucketList<T>();
    bucketSize = new AtomicInteger(2);
    setSize = new AtomicInteger(0);
  }

  public boolean add(T x) {
    int myBucket = Math.abs(BucketList.hashCode(x) %
bucketSize.get());
    BucketList<T> b = getBucketList(myBucket);
    if (!b.add(x))
      return false;
    int setSizeNow = setSize.getAndIncrement();
    int bucketSizeNow = bucketSize.get();
    if (setSizeNow / (double)bucketSizeNow > THRESHOLD)
      bucketSize.compareAndSet(bucketSizeNow, 2 *
bucketSizeNow);    //Vergrößert bei Bedarf
    return true;
  }
```

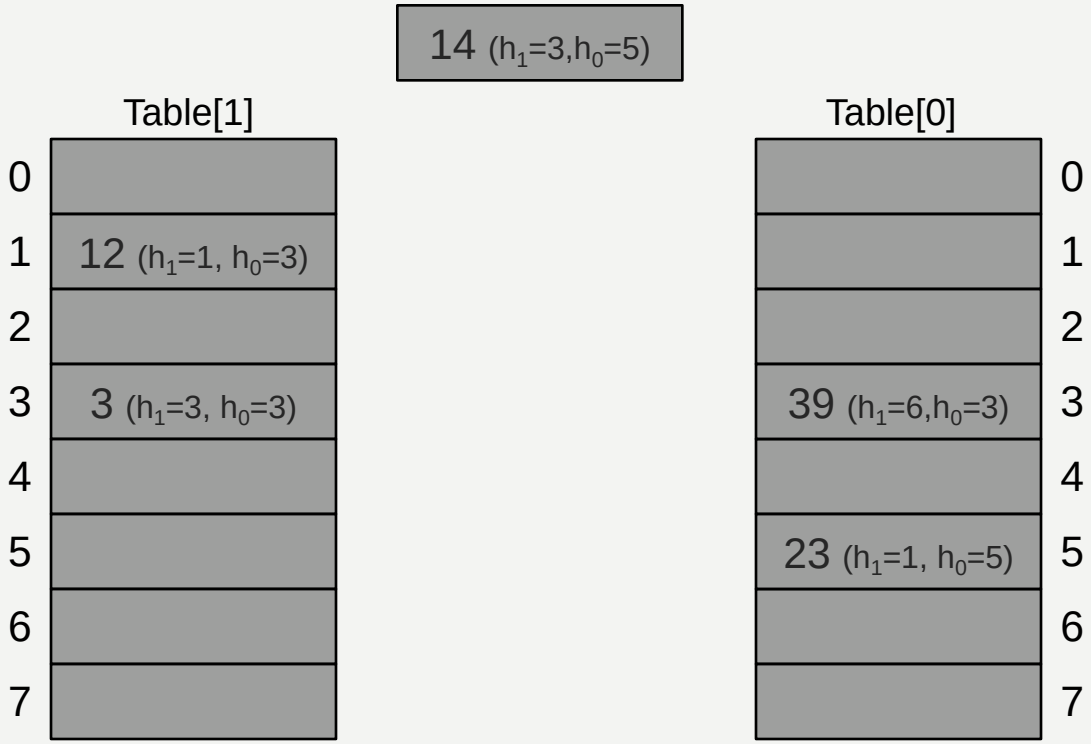
```
 private void initializeBucket(int myBucket) {
    int parent = getParent(myBucket);
    if (bucket[parent] == null)
      initializeBucket(parent);
    BucketList<T> b = bucket[parent].getSentinel(myBucket);
    if (b != null)
      bucket[myBucket] = b;
  }
```

An Open-Adressed Hash Set

-Jeder Eintrag enthält nur ein Element

-Schwieriger nebenläufig umzusetzen ?

Cuckoo Hashing

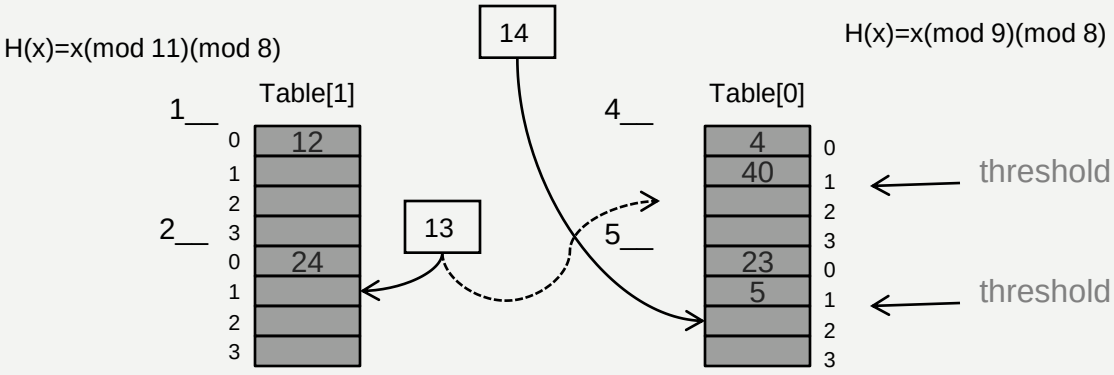


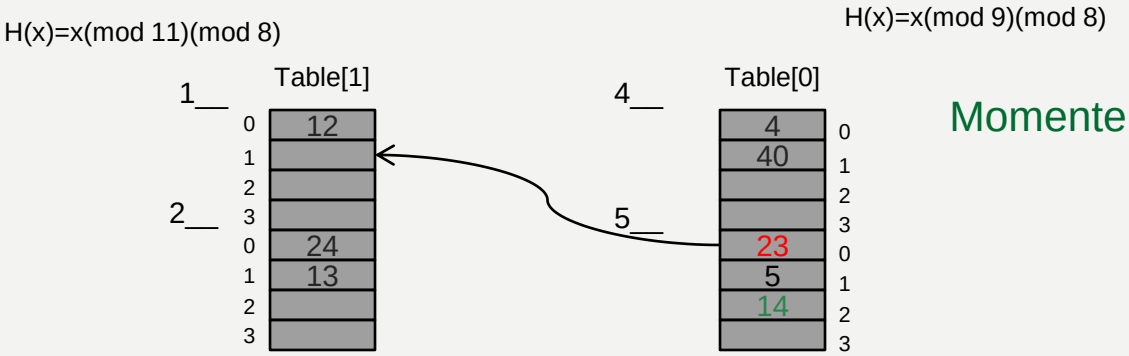
Hash Funktionen:

$h_0(x)=x(\text{mod } 9)(\text{mod } 8)$

$h_1(x)=x(\text{mod } 11)(\text{mod } 8)$

Concurrent Cuckoo Hashing





Momente der Stille:

- Ausgleichen, wenn Schwellenwert überschritten

Concurrent Cuckoo Hashing

```
 public boolean add(T x) {
…
      if (set0.size() < THRESHOLD) {
        set0.add(x);
        return true;
      } else if (set1.size() < THRESHOLD) {
        set1.add(x);
        return true;
      } else if (set0.size() < PROBE_SIZE)
{
        set0.add(x);
        i = 0; h = h0;
      } else if (set1.size() < PROBE_SIZE)
{
        set1.add(x);
        i = 1; h = h1;
      } else {
        mustResize = true;
      }
    } finally {
      release(x);
    }
    if (mustResize) {
      resize();
      add(x);
    } else if (!relocate(i, h)) {
      resize();
    }
…
```

```
 protected boolean relocate(int i, int hi) {
…
      acquire(y);
      List<T> jSet = table[j][hj];
      try {
        if (iSet.remove(y)) {
          if (jSet.size() < THRESHOLD) {
            jSet.add(y);
            return true;
          } else if (jSet.size() < PROBE_SIZE) {
            jSet.add(y);
            i = 1 - i;
            hi = hj;
            j = 1 - j;
          } else {
            iSet.add(y);
            return false;
          }
        } …
```

Striped Concurrent Cuckoo Hashing

-Erbt von Concurrent Cuckoo Hashing

-Verwendet Locks um die Hash Sets Vergrößern zu können

-Deadlocks werden so vermieden

Striped Concurrent Cuckoo Hashing

```
public void resize() {
   int oldCapacity = capacity;
   for (Lock _lock : lock[0]) {
     _lock.lock();
   }
   try {
     if (capacity != oldCapacity) {  // ein anderer hat bereits die Größe verändert
       return;
     }
     List<T>[][] oldTable = table;
     capacity = 2 * capacity;
     table = (List<T>[][]) new List[2][capacity];
     for (List<T>[] row : table) {
       for (int i = 0; i < row.length; i++) {
         row[i]  = new ArrayList<T>(PROBE_SIZE);
       }
     }
     for (List<T>[] row : oldTable) {
       for (List<T> set : row) {
         for (T z : set) {
           add(z);
         }
       }
     }
   } finally {
     for (Lock _lock : lock[0]) {
       _lock.unlock();
     }
   }
 }
```

Fazit

Hash Set ist sehr effiziente Datenstruktur
- Zugriff in konstanter Zeit
- Schreiboperationen im Mittel konstant

Für Concurrent Hash Sets gilt:
- Möglichst gleiche Menge an Locks wie Tabellenplätzen
- Explizites Resize Command unterbindet jeden Zugriff auf Hash Set

Sehr praktikable Lösung:
Lock-Free Hash Set
- Ohne Locks
- Inkrementelle Vergrößerung durch Recursive-Split Order

Ende

Vielen Dank für die Aufmerksamkeit!

Quellen:
M. Herlihy and N. Shavit. **The art of multiprocessor programming. 2008, Morgan Kaufmann Publishers.**
O. Shalev and N. Shavit. **Split-ordered lists: Lock-free extensible hash tables. 2006, J. ACM**