Advanced Multiprocessor Programming

Jesper Larsson Träff
traff@par.tuwien.ac.at
Research Group Parallel Computing
Faculty of Informatics, Institute of Computer Engineering
Vienna University of Technology (TU Wien)

Set/Dictinary based on hash tables (chap. 13)

Dictionary (set) data structure supports

- add(x)
- remove(x)
- contains(x)

for item (key) x. Again, no distinction between keys and items

Assume existence of hash function h(x) that maps item/key to integer, with good properties:

- $h(x) \neq h(y)$ for $x \neq y$ with high probability
- h(x) can be evaluated efficiently in O(1) steps, and fast

Currently much activity on fast, good hashing (Thorup ao.)

Idea:
Store items in hash table of some size (capacity), item x at position h(x) mod capacity. All operations in O(1) expected/high probability?

Much potential for concurrency: Disjoint-access parallelism

What to do on collisions, h(x)=h(y)?

Two possibilities:
- Closed addressing: Items with hash key h(x) stored in bucket at index h(x) mod capacity
- Open addressing: At most one item at index h(x) mod capacity

What to do on when table (buckets) become too full?

Solution: Resize, move item to new table

Closed address hash table

Each table entry maintains (small) bucket of elements (implemented as list-array)

To ensure constant O(1) access time, the buckets must have constant maximum size. When some bucket exceeds maximum (or average bucket size exceeds maximum), table is resized and items redistributed

Buckets as array lists (any suitable data structure can be used)

```
public abstract class BaseHashTable<T> {
  protected List<T> table;
  protected int size; // total number of items in set
  protected int capacity; // number of indices in table
  public BaseHashTable(int c) {
    size = 0;
    capacity = c;
    table = (List<T>[]) new List[capacity];
    for (int i=0; i<capacity; i++)
      table[i] = new ArrayList<T>(); // bucket
  }
  …
}
```

Implementation choice

Contains method from ArrayList

```
public boolean contains(T x) {
  acquire(x);
  try {
   return table[h(x)%capacity].contains(x);
  } finally {
    release(x);
  }
}
```

Acquire and release functions give exclusive access to the necessary parts of the hash table, ideally only the needed index (and the size)

```
public boolean add(T x) {
  int result = false;
  acquire(x);
  try {
    int ix = h(x)%capacity;
    if (!table[ix].contains(x)) {
      table[ix].add(x); // insert into bucket
      result = true;
      size++;
    }
  } finally {
    release(x);
  }
  if (policy()) resize();
  return result;
}
```

Policy function determines when to resize (to maintain O(1) access time)

```
public boolean remove(T x) {
  int result = false;
  acquire(x);
  try {
    int ix = h(x)%capacity;
    if (table[ix].contains(x)) {
      table[ix].remove(x); // delete bucket
      result = true;
      size--;
    }
  } finally {
    release(x);
  }
  // no shrinking here (exercise)
  return result;
}
```

Lock-based closed-address hash tables

Trivial solution: Coarse-grained locking, one lock for whole hash table

```
public class CoarseHashTable<T>
       extends BaseHashTable<T> {
  final Lock lock;
  public CoarseHashTable(int capacity) {
    super(capacity);
    lock = new ReentrantLock();
  }
  public final void acquire(T x) {
    lock.lock();
  }
  public final void release(T x) {
    lock.unlock();
  }
}
```

Resize:
To maintain constant size buckets (and constant time access), when average load exceeds some constant THRESHOLD, double the number of buckets

Implemented in policy and resize functions. Resizing acquires lock for whole table and "stops the world" (like most garbage collectors)

```
public boolean policy() {
  return size/capacity > THRESHOLD; // avg. load
}

public void resize() {
  lock.lock();
  try {
    if (!policy()) return;

    // double table capacity (next slide)
    …
  } finally lock.unlock();
}
```

Somebody else may have resized, recheck condition

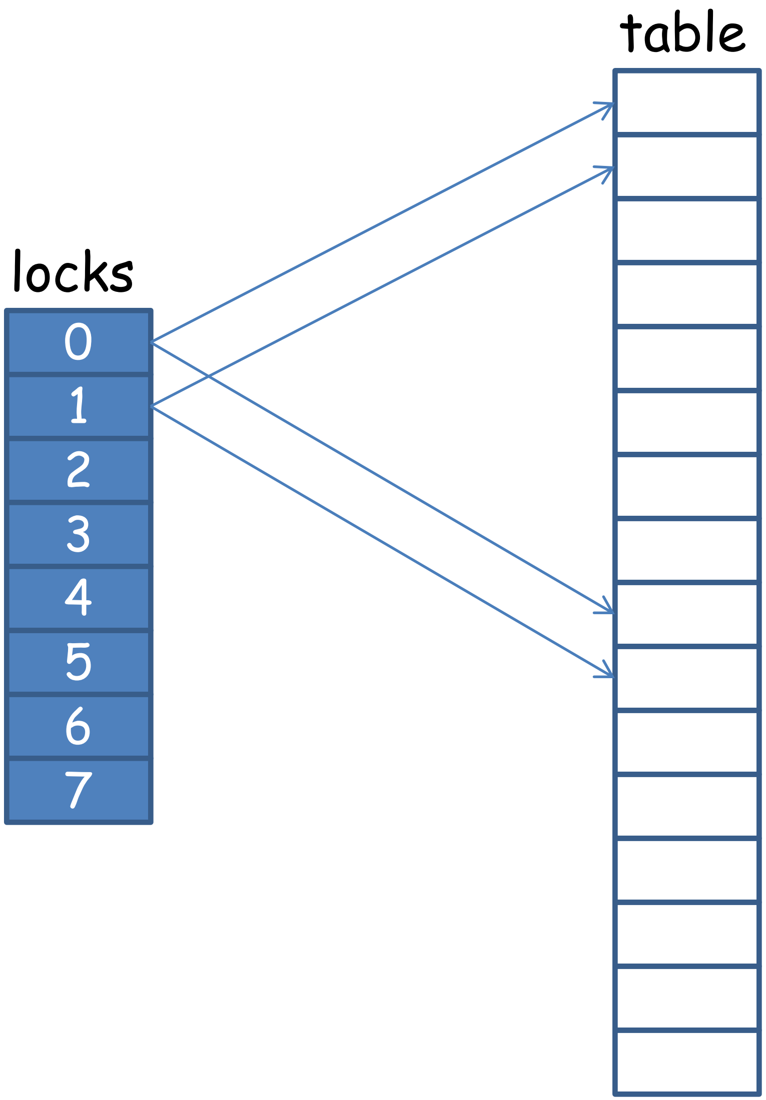
```
// double table capacity

capacity = 2*capacity;
List<T>[] oldtable = table;
table = (List<T>[]) new List[capacity];
for (int i=0; i<capacity; i++)
  table[i] = new ArrayList<T>();
for (int i=0; i<capacity; i++)
  for (int j=0; j<oldtable[i].length; j++)
    int x = oldtable[i][j];
    table[h(x)%capacity].add(x);
delete oldtable;
```

Better solution: Striped hash set, fixed array of locks

```
public class StripedHashTable<T>
        extends BaseHashTable<T> {
  final ReentrantLock[] locks;
  public StripedHashTable(int capacity) {
    super(capacity);
    locks = new Lock[capacity];
    for (int i=0; i<capacity; i++)
      locks[i] = new ReentrantLock();
  }
  public final void acquire(T x) {
    locks[h(x)%locks.length].lock();
  }
  public final void release(T x) {
    locks[h(x)%locks.length].unlock();
  }
}
```

Not same as table capacity



After 1^{st} resize, locks[i] protects j=h(x)%capacity with j%locks.length=i

Resize (table only!): Easy, needs to acquire all locks of the striped lock array. Done in ascending order, no deadlock

Drawback: Resize "stops the world", lock table is not changed, granularity gets larger and larger as table increases

```
public void resize() {
  int oldcapacity = table.length;
  for (Lock lock: locks) lock.lock();
  try {
    if (oldcapacity!=table.length) return;
    // already done by other thread
    int newcapacity = 2*oldcapacity;
    List<T>[] oldtable = table;
    table = (List<T>[])new List[newcapacity];
    for (int i=0; i<newcapacity; i++)
      table[i] = new ArrayList<T>();
    for (List<T>bucket: oldtable) // copy table elements
      for (T x: bucket)
        table[h(x)%table.length].add(x);
  } finally {
    for (Lock: lock) lock.unlock();
  }
}
```

Must be in fixed order

But:
Changing the lock array in the resize operation does not work:

Some thread may be trying to acquire lock assuming table capacity has not changed. This lock may either not exist, or not protect the right entry.

(Also note that a resize operation may recursively trigger nested resize calls (by the add); but this still works correctly (why?). For this reason, reentrant locks are needed)

Next refinement: Refinable hash table (resizable locks).

Introduce global marked reference (owner thread and flag) to indicate whether resize is in progress and by which thread. Acquire operation checks marked reference, and acquires lock only if resize is not in progress, spins otherwise. Release just releases the lock. Resize atomically sets flag and reference, and can now resize both table and lock array

Drawback:
Resize operation still "stops the world", no concurrency when resize in progress (for both striped and refinable hash table)

```java
public class RefinableHashTable
       extends BaseHashTable<T> {
  AtomicMarkableReference<Thread> owner;
  volatile ReentrantLock[] locks;
  public RefinableHashTable(int capacity) {
    super(capacity);
    locks = new ReentrantLock[capacity];
    for (int i=0; i<capacity; i++) {
      locks[i] = new ReentrantLock();
    }
    owner =
      new AtomicMarkableReference<Thread>(null,false);
    …
}
```

```java
public void acquire(T x) {
  boolean[] mark = {true};
  Thread t = Thread.currentThread();
  Thread w;
  while (true) {
    do { // spin while resizing in progress
      w = owner.get(mark);
    } while (mark[0]&&w!=t);
    ReentrantLock oldlocks = locks;
    ReentrantLock oldlock =
      oldlocks[h(x)%oldlocks.length];
    oldlock.lock();
    w = owner.get(mark);
    if ((!mark[0]||w==t)&&locks==oldlocks) return;
    else oldlock.unlock();
   // resizing has taken place, retry
  }
}
```

```java
public void resize() {
  int oldcapacity = table.length;
  boolean[] mark = {false};
  int newcapacity = 2*oldcapacity;
  Thread t = Thread.currentThread();
  if (owner.compareAndSet(null,t,false,true) {
    try {
      if (table.length!=oldcapacity) return;
      waitforquiescence();
      List<T>[] oldtable = table;
      table = (List<T>[])new List[capacity];
      for (int i=0; i<newcapacity; i++)
        table[i] = new ArrayList<T>();
      locks = new ReentrantLock[newcapacity];
      for (int j=0; j<locks.length; j++)
        locks[j] = new ReentrantLock();
      … // copy table elements to new table
    } finally owner.set(null,false);
  }
}
```

Resizing thread must wait for all operations on hash table to complete (quiescence)

```
protected void waitforquiescence () {
  for (ReentrantLock lock: locks) {
    // spin until lock is released
    while (lock.isLocked()) { }
  }
}
```

How good is this (in real applications)? Resizing may be a major bottleneck. Shrinking of tables and locks is not done

Lock-free closed address hash table

Fine-grained locks on buckets may not be too problematic(?); but "stop-the world" on resize could be a major bottleneck

Ideas:

- Use lock-free list to store all buckets, with references into bucket list from hash table
- Make resize purely local operation, breaking up too long bucket
- Be lazy

To make this work, the lock-free list must be organized such that a hashed item does not have to change position on resize.
How is this possible?

Maintain list in a certain order

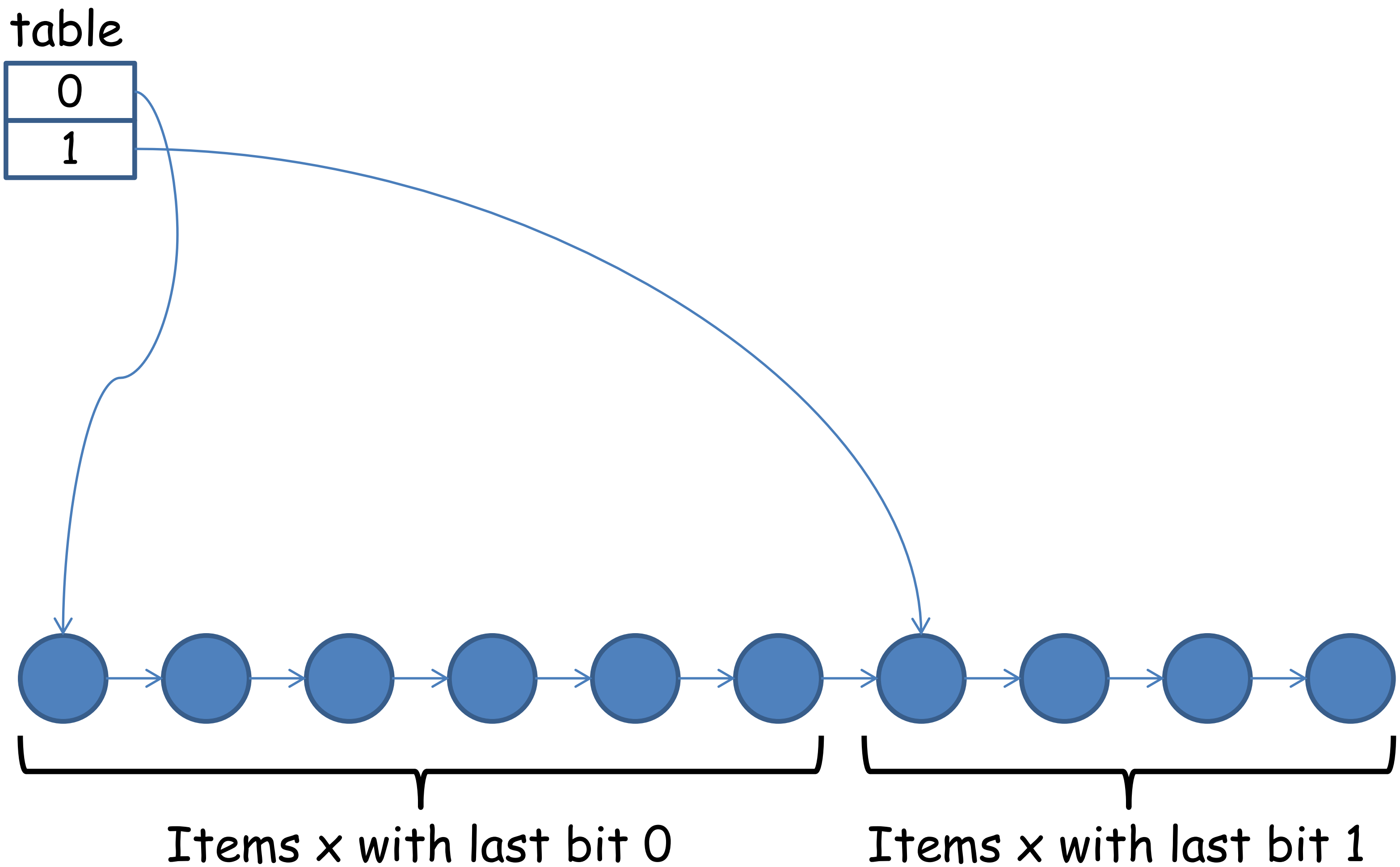
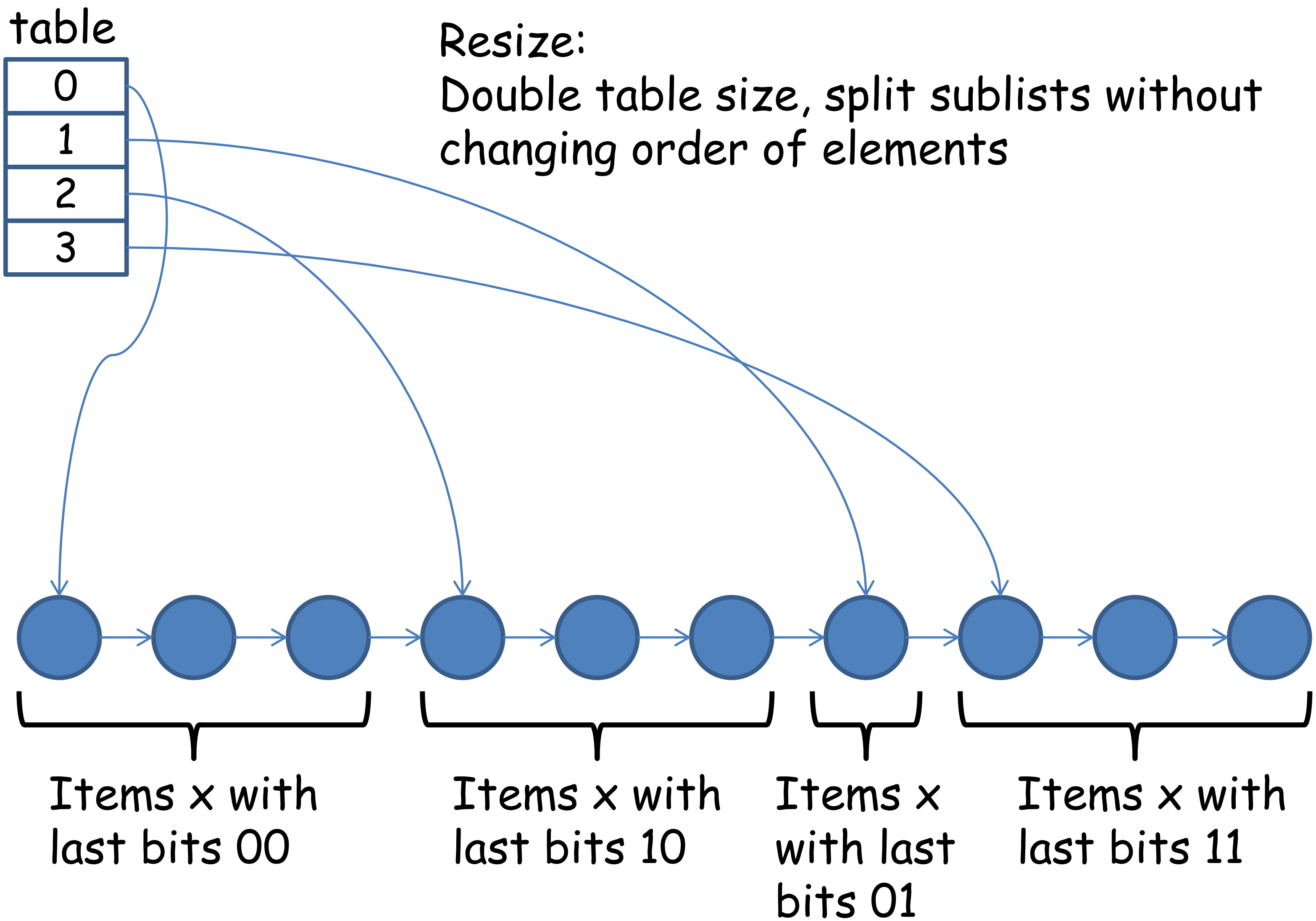
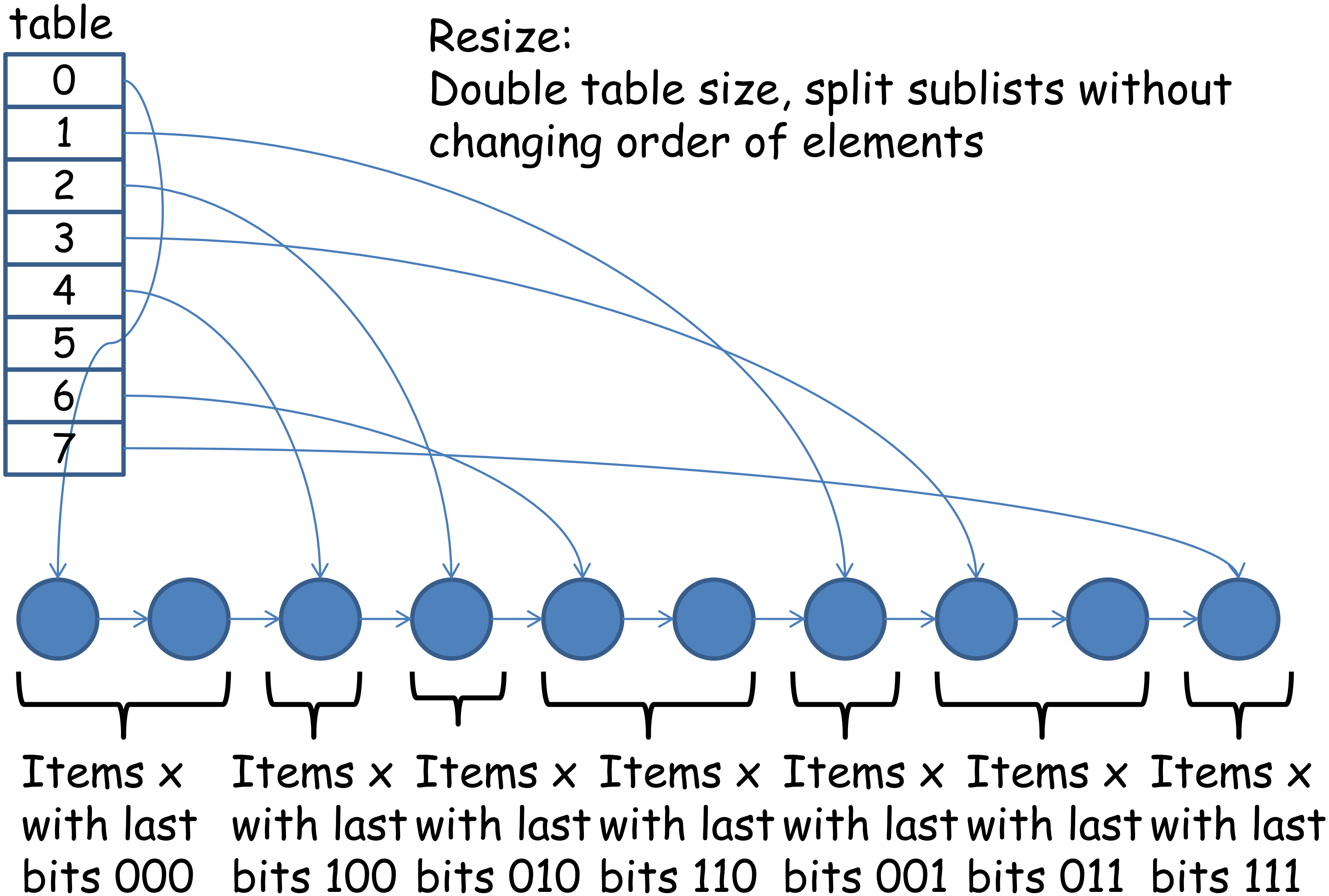


table
0
1
Items x with last bit 0
Items x with last bit 1

Resize:
Double table size, split sublists without changing order of elements



Resize:
Double table size, split sublists without changing order of elements

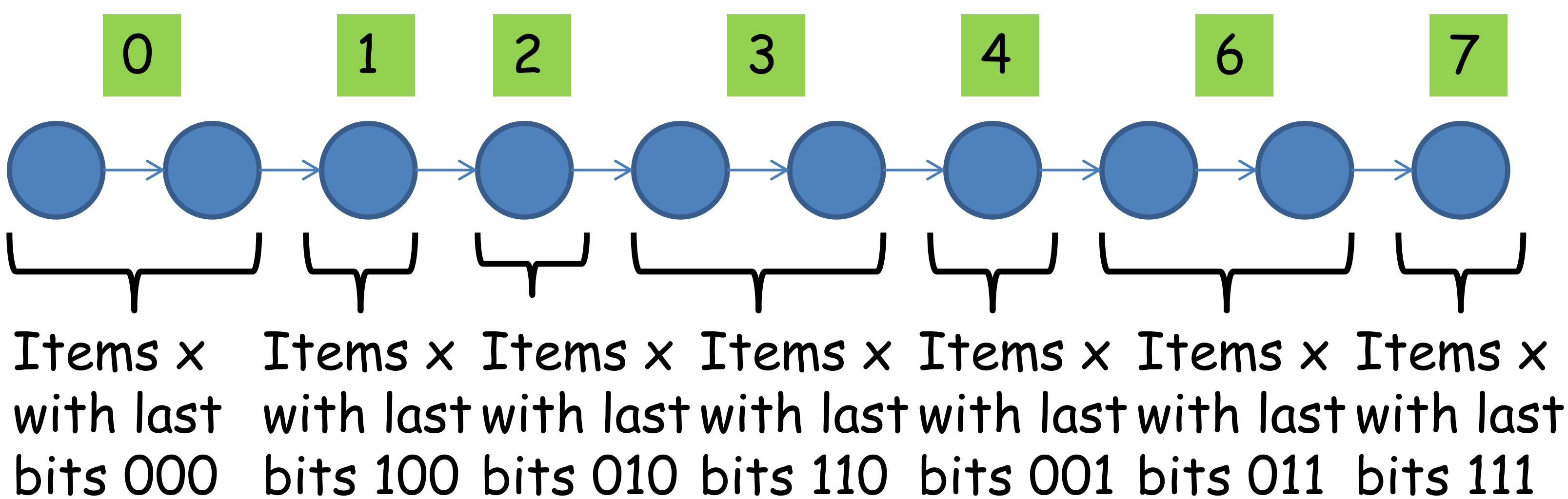


Splitting sublists without changing order possible if bucket list is maintained in sorted order on $\text{reverse}_w(h(x))$

See HPC lecture

$\text{reverse}_w(y)$:
Bit reversal of y, e.g., with w=8 reverse(01101101) = 10110110, reverse(11110000) = 00001111, reverse(00111100) = 00111100

3-bit reverse (example):



Technicalities:

- Lock-free list works best with sentinel head and tail elements (never changed)
- The recursive split-ordered list has several sentinel elements (corresponding to non-empty buckets), distinguish between sentinel and normal items by upper bit (set for normal, non-sentinel items)
- Implementations (next slides) for 32-bit hash keys
- Each sublist a lock-free BucketList
- If sentinel for some sublist is not present, it is added lazily

- Reuse find-method of lock-free list-based set. Find operations returns a window structure with pred and curr references, such that pred.key<x≤curr.key

```
public class BucketList<T> {
  Node head;
  static final int HIGH = 0x80000000;
  static final int MASK = 0x00FFFFFF;
  public BucketList() {
    head = new Node(0);
    head.next =
      new AtomicMarkedReference<Node>(
        new Node(Integer.MAX_VALUE),false);
  }
  public int makeNormalKey(T x) {
    int key = h(x) & MASK; // lowest bytes
    return reverse(key | HIGH);
  }
  public int makeSentinelKey(int key) {
    return reverse(key & MASK);
  }
  …
}
```

32-bit reverse function

Properties of item keys:

Sentinel smaller than all items belonging to bucket (e.g., sentinel for bucket 20, after reverse, has least significant bit 0, with same upper bits as all normal items of bucket 20, which have least significant bit 1)

All items in bucket smaller than sentinel of next bucket

Property of table size:

Starting from $1=2^0$, always a power of 2

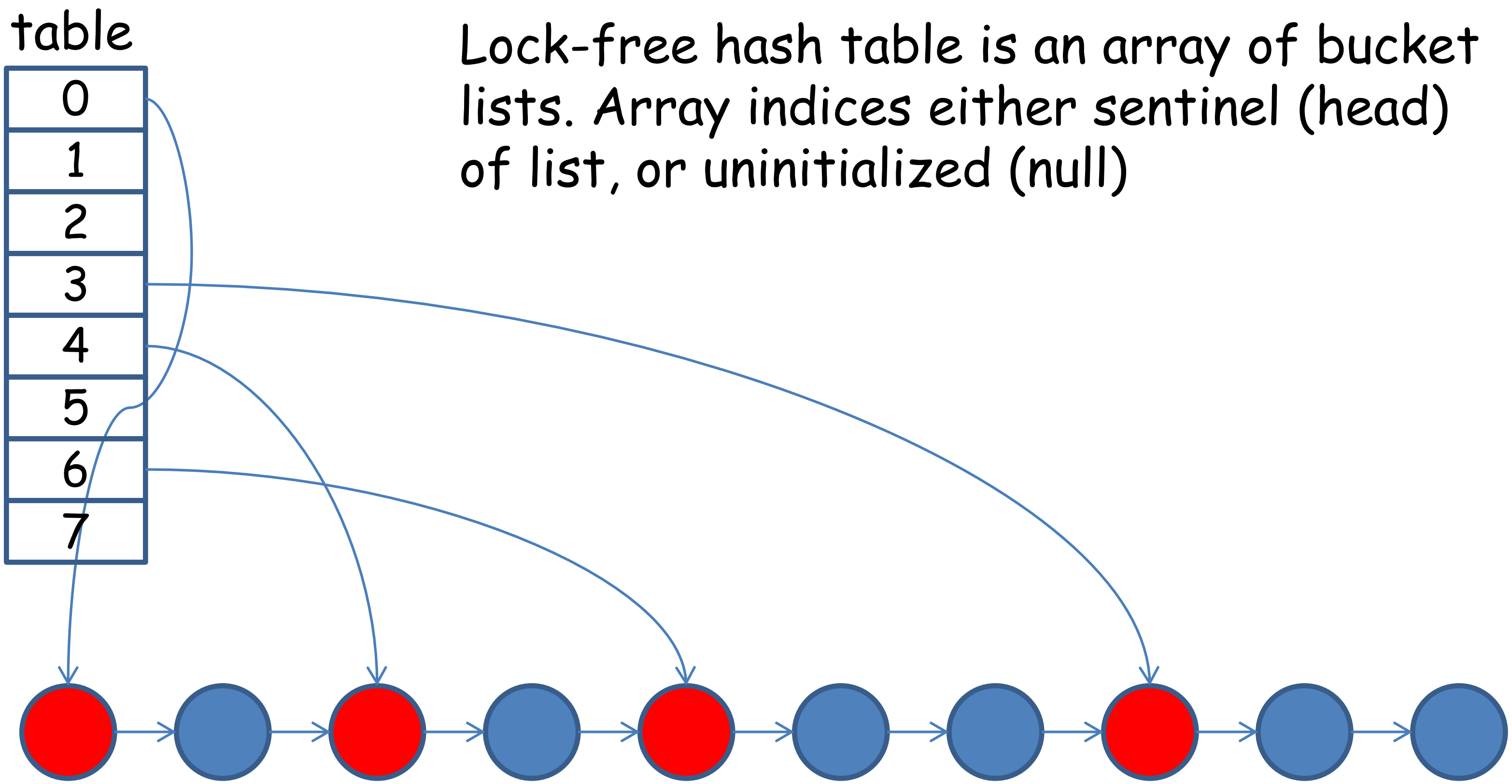
```java
public class BucketList<T> {
  private Window find(Node head, T x);

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

  public boolean add(T x); // also key reverse
  public boolean remove(T x);
}
```

All operations reverse key by makeNormalkey. Find operation from lock-free list-based set; returns window of pred and curr elements

```
public BucketList<T> getSentinel(int index) {
  int key = makeSentinelKey(index);
  while (true) {
    Window window = find(head,key);
    Node pred = window.pred;
    Node curr = window.curr;
    if (curr.key==key)
      return new BucketList<T>(curr);
    else {
      Node node = new Node(key);
      node.next.set(pred.next.GetReference(),false);
      if (pred.next.compareAndSet(curr,node,
                                  false,false)) {
        return new BucketList<T>(node);
      else continue;
    }
  }
}
```



Lock-free hash table is an array of bucket lists. Array indices either sentinel (head) of list, or uninitialized (null)

Note: This implementation preallocates whole table of maximum capacity. This can be improved

```
public class LockFreeHashTable<T> {
  protected BucketList<T>[] table;
  protected AtomicInteger tablesize;
  protected AtomicInteger size;
  public LockFreeHashTable(int c) {
    table = (BucketList<T>[]) new BucketList[c];
    table[0]  = new BucketList<T>();
    tablesize = new AtomicInteger(1);
    size      = new AtomicInteger(0);
  }
  … // private functions to get and init buckets
}
```

The lock-free hash table consist of (for now: fixed) table of bucket lists.
Maintain current table size (assumption: Always smaller than capacity c), and current number of items (size) in hash table

```
public boolean add(T x) {
  int bucket = h(x)%tablesize.get();
  BucketList<T> b = getBucket(bucket);
  if (!b.add(x)) return false;
  int s = size.getAndIncrement();
  int t = tablesize.get();
  if ((s+1)/t>THRESHOLD)
    tablesize.compareAndSet(t,2*t);
  return true;
}
```

Resize: Just double table size

Adding new element: Look for current table entry of bucket

Increase table size (implicit resize) when average bucket load beyond THRESHOLD

```
private BucketList<T> getBucket(int bucket) {
  if (table[bucket]==null) initializeBucket(bucket);
  return table[bucket];
}
```

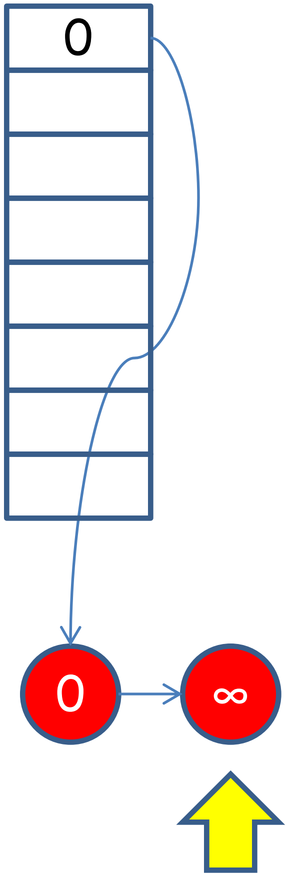
```
private void initializeBucket(int bucket) {
  int parent = getParent(bucket);
  if (table[parent]==null) initializeBucket(parent);
  BucketList<T> b =
    bucket[parent].getSentinel(bucket);
  if (b!=null) table[bucket] = b;
}
```

```
private void getParent(int bucket) {
  int parent = tablesize.get();
  do {
    parent = parent>>1;
  } while (parent>bucket);
  parent = bucket-parent;
  return parent;
}
```

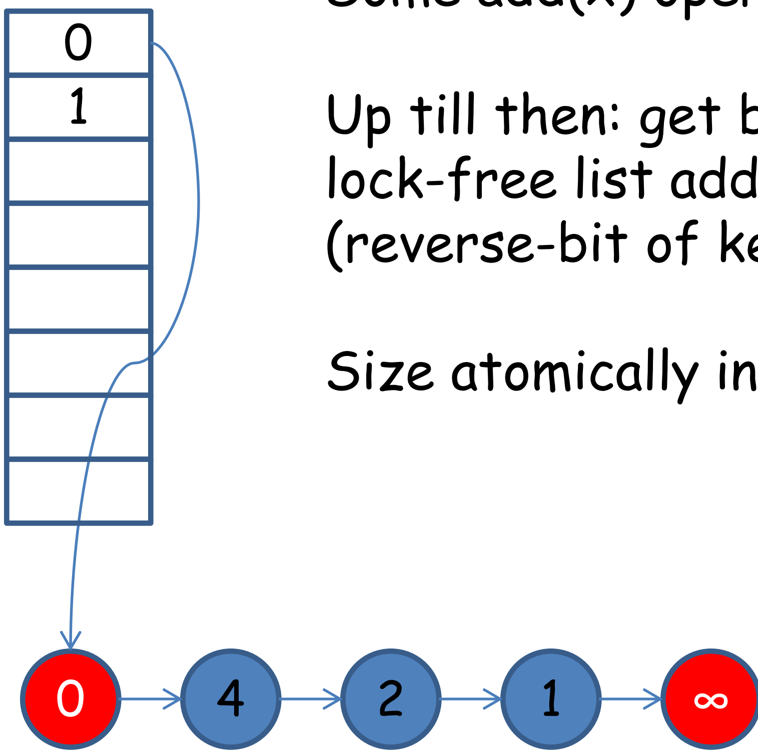
Parent is the index of a bucket to be split. Should be close to new bucket, but not lead to too many recursive initializations:

Find largest power of 2 not greater than bucket index. Use this to set most significant bit to zero in bucket index. E.g., 7 = 00000111_2 becomes 00000011_2=3

Initializing lock-free hash table with at most 8 entries



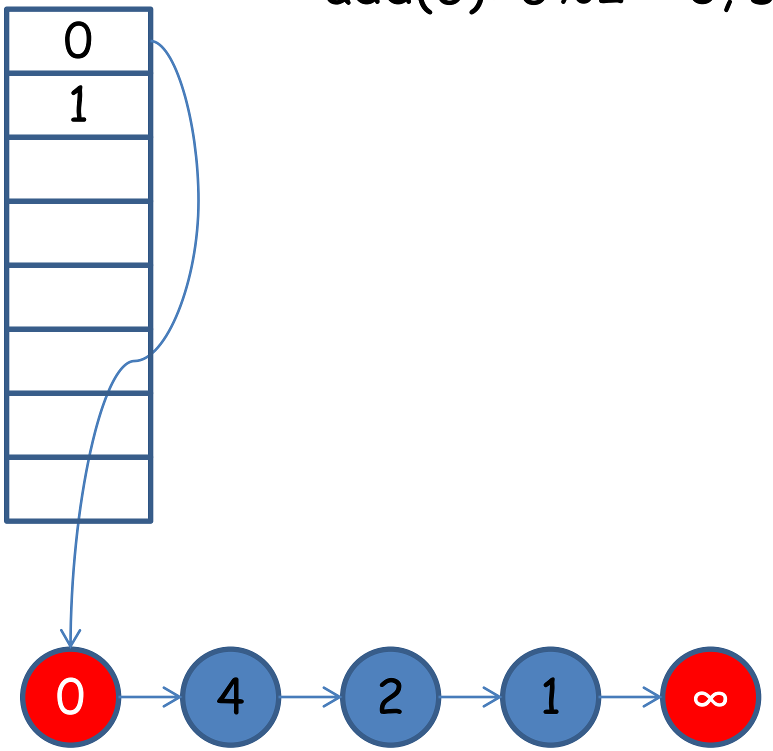
Sentinel elements



Some add(x) operations trigger resize.

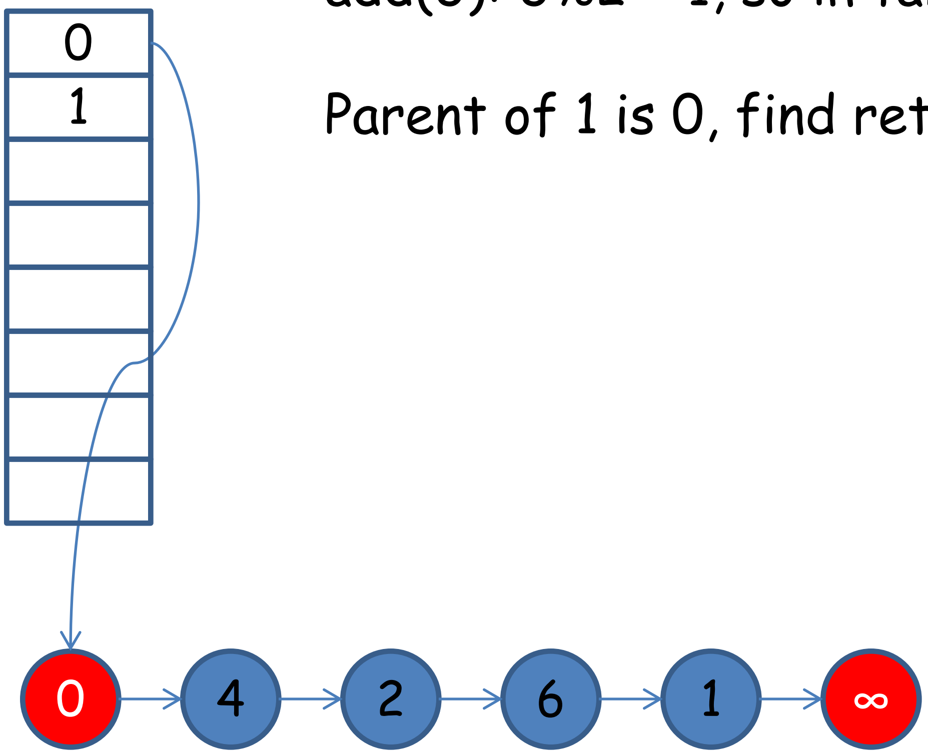
Up till then: get bucket returns table[0] bucket, lock-free list add maintains sorted order (reverse-bit of key)

Size atomically incremented to 3, bucketsize to 2 add(6): 6%2 = 0, so in table[0], does not split lists



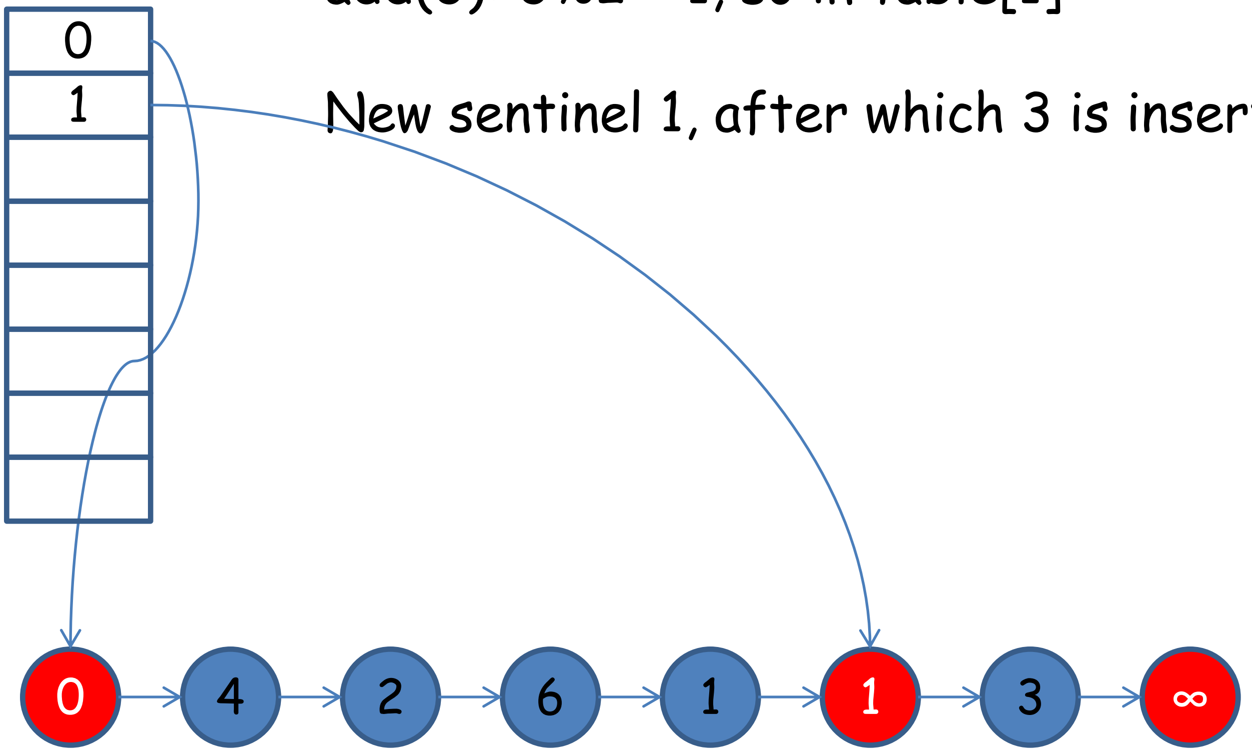
add(3): 3%2 = 1, so in table[1], uninitialized

Parent of 1 is 0, find returns node 1



add(3): 3%2 = 1, so in table[1]

New sentinel 1, after which 3 is inserted



Analysis: (non-trivial)

Initializing empty bucket (table[bucket]==**null**) can recursively trigger $\log_2$(tablesize) initializations
Can be shown that expected depth of recursion is constant
Overall (expected) complexity of all operations is O(1) steps

Note also:
Large buckets where all h(x)%newtablesize go into same bucket remain large

Contains and remove operations also initialize buckets (needed?)

```
public boolean remove(T x) {
  int bucket = h(x)%tablesize.get();
  BucketList<T> b = getBucket(bucket);
  if (b.remove(x)) return false;

  // no shrinking (should perhaps be done)
  return true;
}

public boolean contains(T x) {
  int bucket = h(x)%tablesize.get();
  BucketList<T> b = getBucket(bucket);
  return b.contains(x);
}
```

Properties:

- Lock-freedom inherited from lock-free list
- Linearizable; linearization points for concurrent updates to same bucket are the linearization points of corresponding list operations

Drawback: Fixed (large), preallocated table; not shrinkable (can be fixed)

Ori Shalev, Nir Shavit: Split-ordered lists: Lock-free extensible hash tables. J. ACM 53(3): 379-405 (2006)

Open addressing

All items stored in hash table, at most one item at h(x) mod capacity

Linear probing: To find item x, search linearly from index h(x) mod capacity until x found or empty slot

Some results, some open problems:

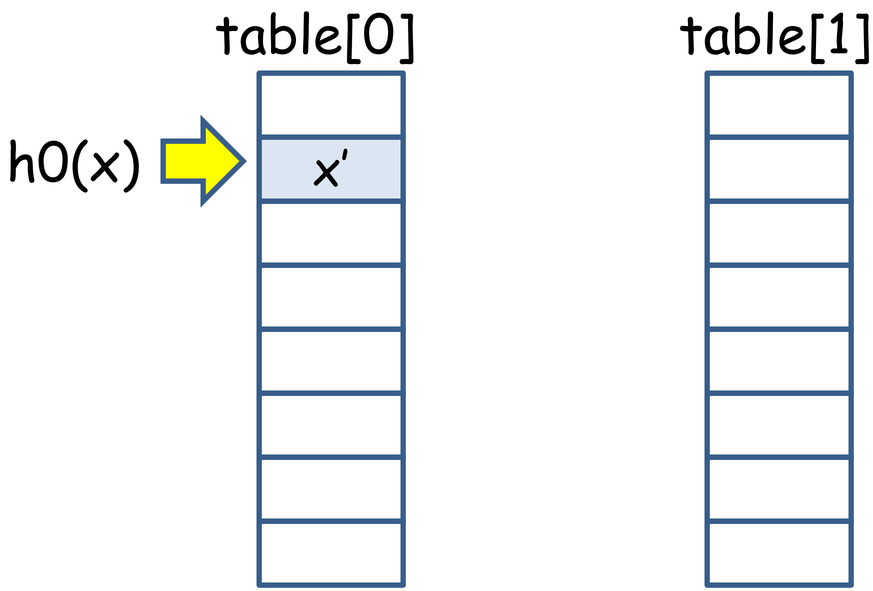
Chris Purcell, Tim Harris: Non-blocking Hashtables with Open Addressing. DISC 2005: 108-121
Hui Gao, Jan Friso Groote, Wim H. Hesselink: Lock-free dynamic hash tables with open addressing. Distributed Computing 18(1): 21-42 (2005)

Open address hashing: Cuckoo

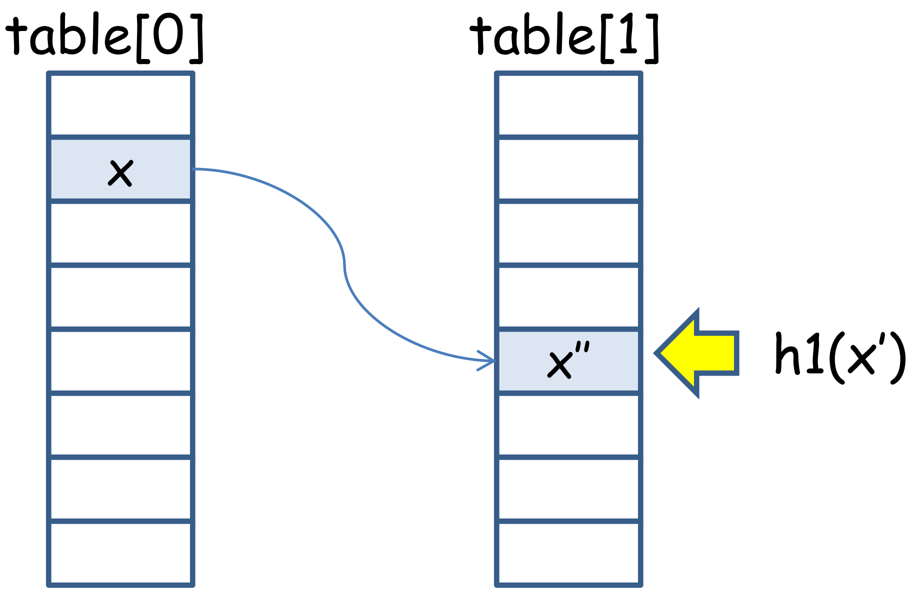
Idea:
A newly added item pushes out previous item in same slot (Cuckoo)

Use two hash tables (same size), and two hash functions h0(x) and h1(x)

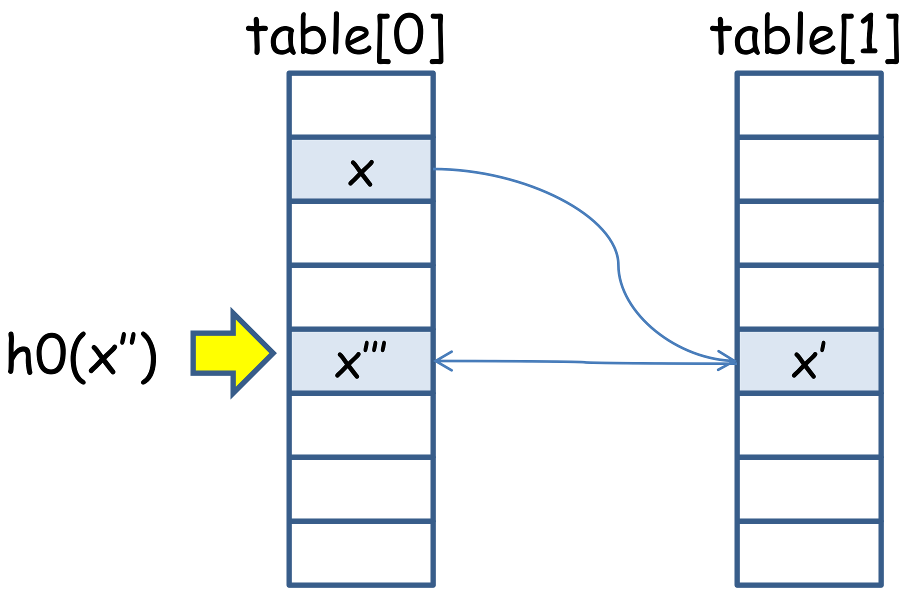
Example: add(x)



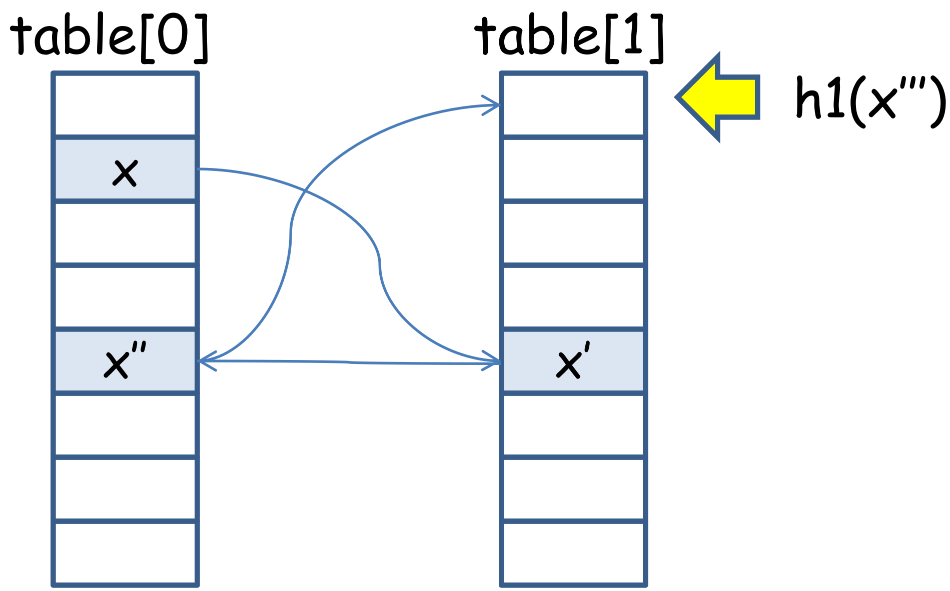
Example: add(x)



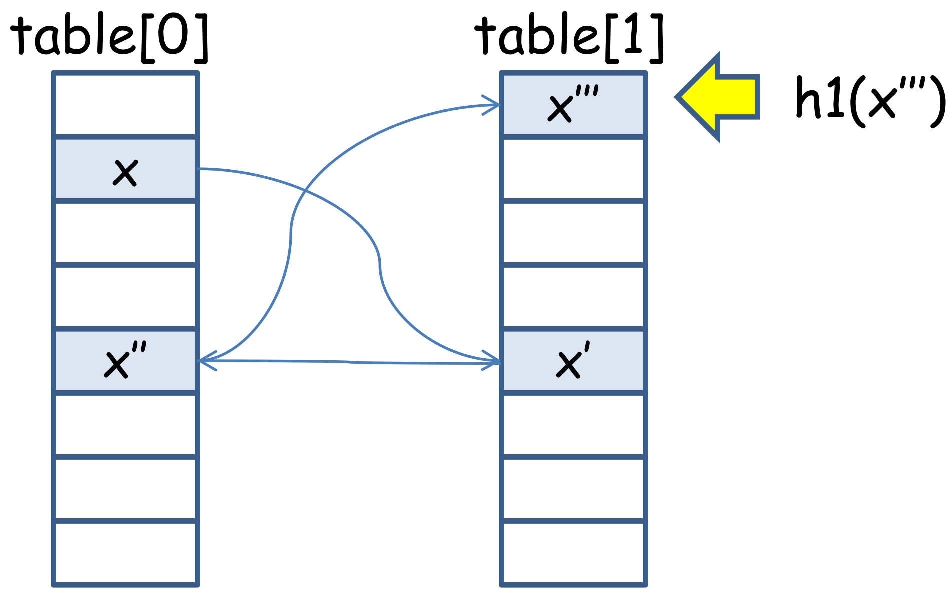
Example: add(x)



Example: add(x)

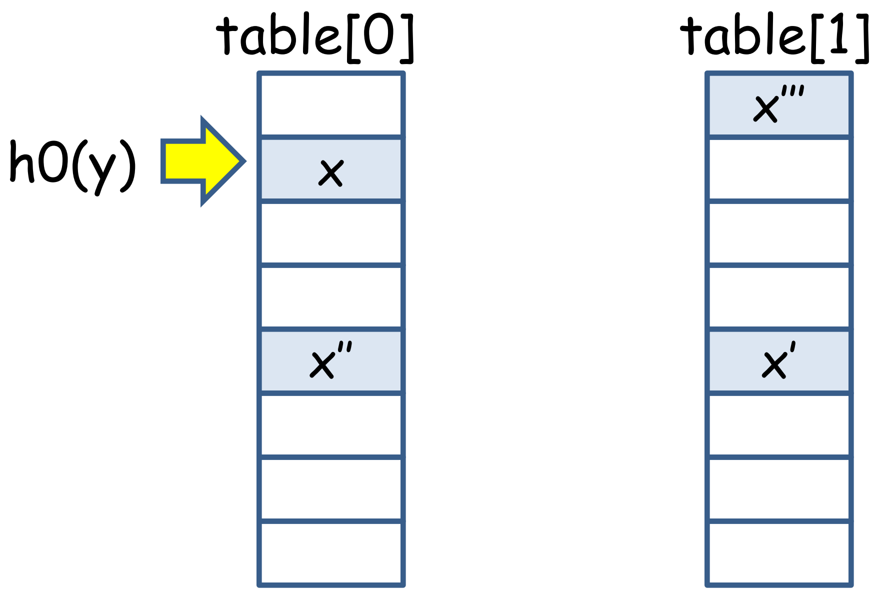


Example: add(x)

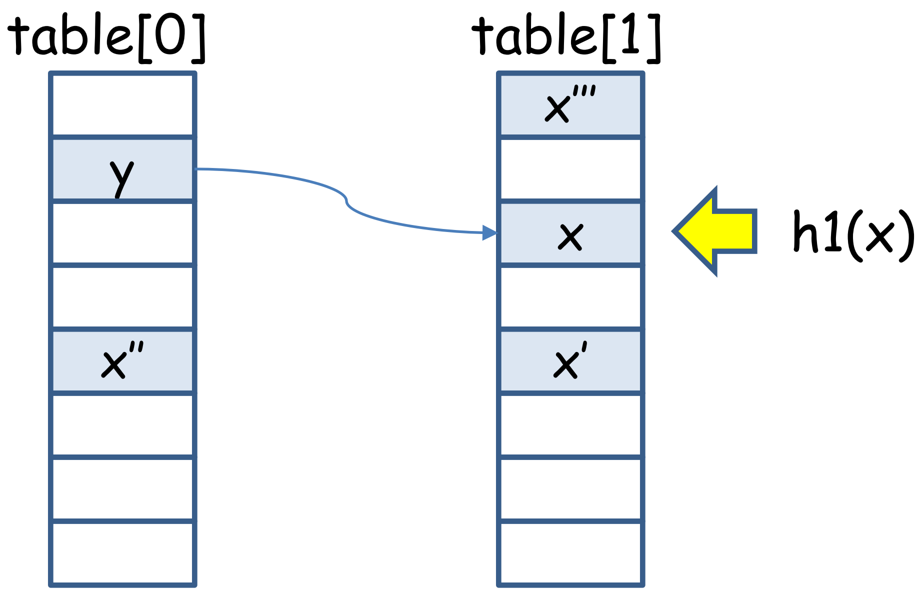


Done when free slot eventually found.

Example: add(x), add(y) with h0(x)=h0(y)

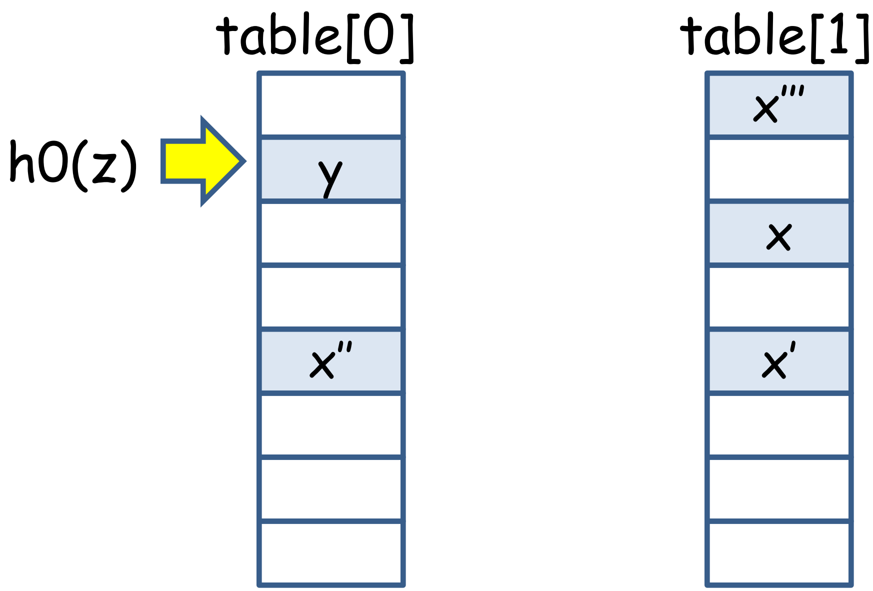


Example: add(x), add(y) with h0(x)=h0(y)

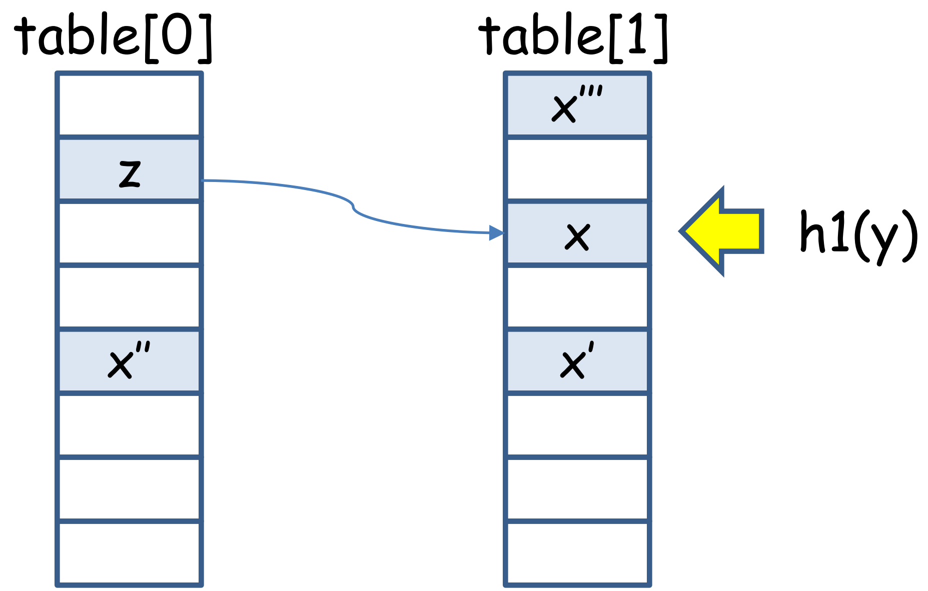


Observation: x either in table[0][h0(x)] or in table[1][h1(x)]

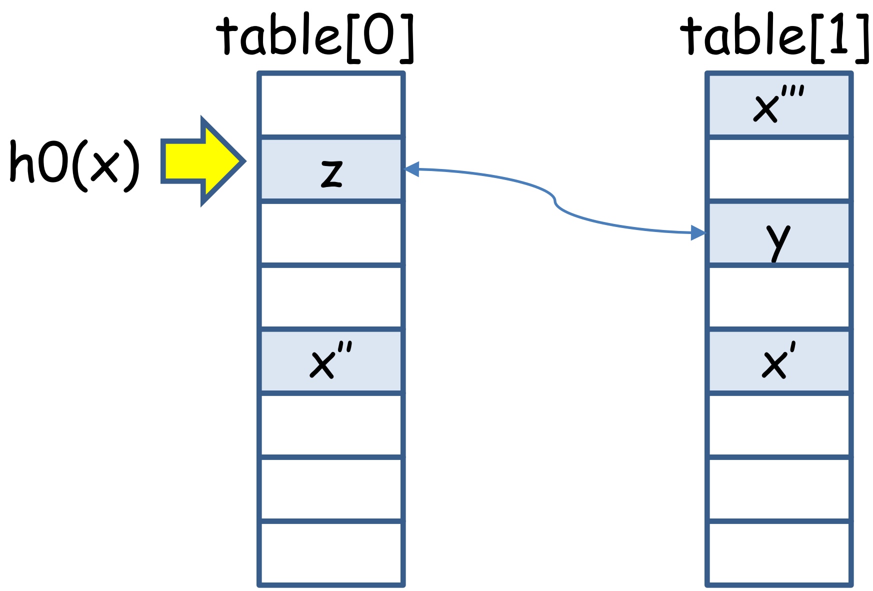
Example: add(x), add(y), add(z) with h0(z)=h0(y) and h1(y)=h1(x)



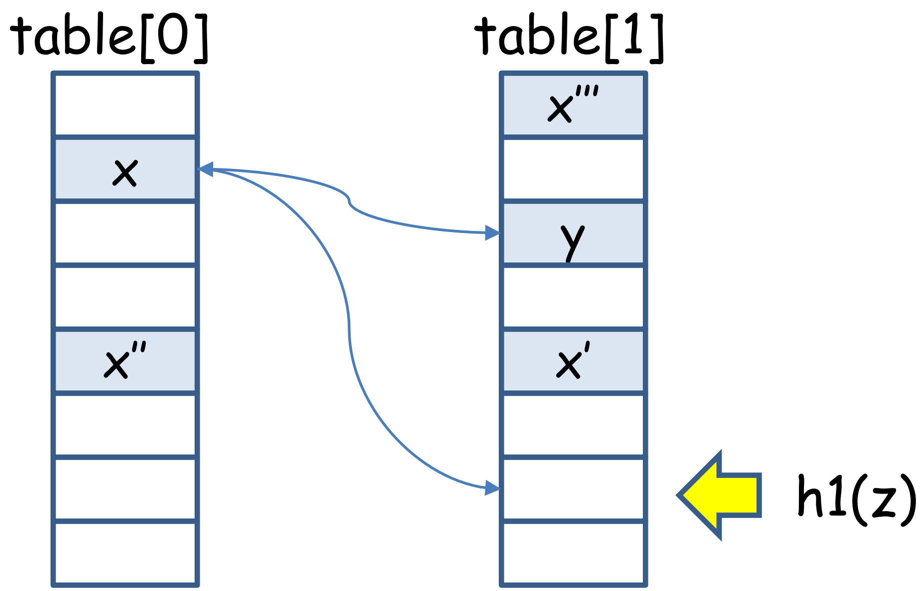
Example: add(x), add(y), add(z) with h0(z)=h0(y) and h1(y)=h1(x)



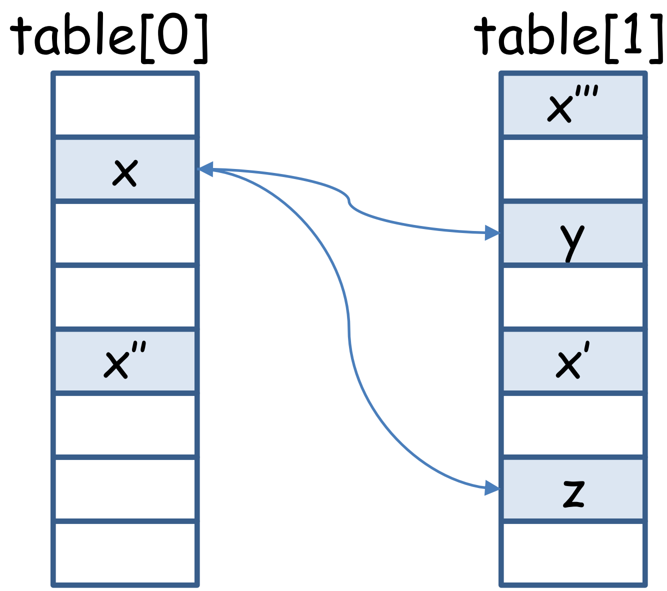
Example: add(x), add(y), add(z) with h0(z)=h0(y) and h1(y)=h1(x)



Example: add(x), add(y), add(z) with h0(z)=h0(y) and h1(y)=h1(x)



Example: add(x), add(y), add(z) with h0(z)=h0(y) and h1(y)=h1(x)



Done when free slot eventually found. Or cycle found. Or chain of kick-outs too long

```
public boolean add(T x) {
  if (contains(x)) return false;

  for (int i=0; i<LIMIT; i++) {
    if ((x=swap(0,h0(x),x))==null) return true;
    if ((x=swap(1,h1(x),x))==null) return true;
  }
  resize(); // failed: too long chain or cycle
  return add(x); // try again
}

public boolean contains(T x) {
  if (table[0][h0(x)]==x) return true;
  if (table[1][h1(x)]==x) return true;
  return false;
}
```

Resize operation increases table sizes, and chooses new hash functions

Sequential cuckoo hashing:
Constant time contains and remove operations.
Average number of displacements in add operations expected constant

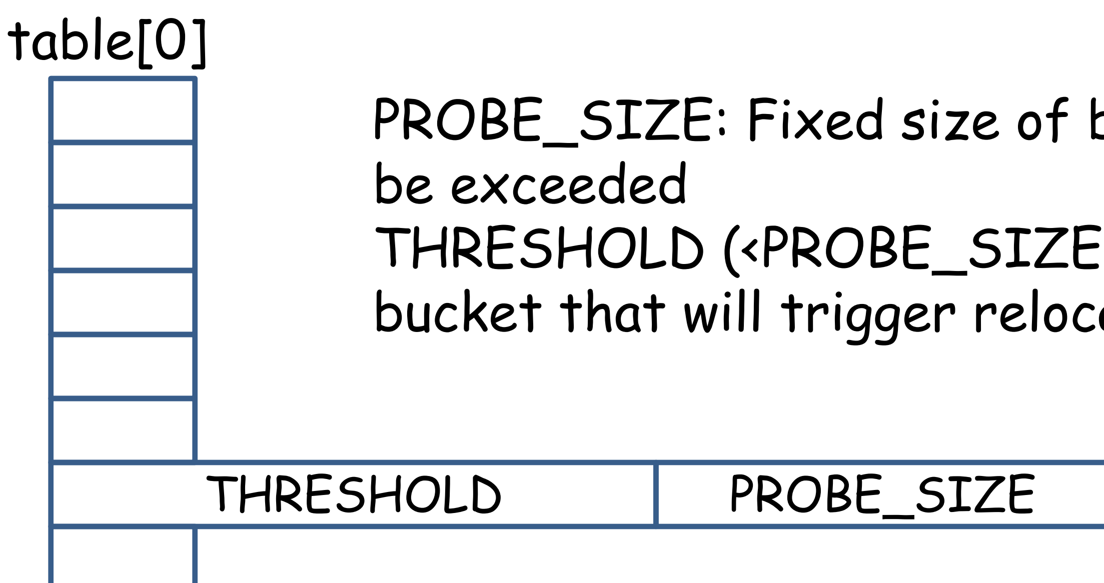
Rasmus Pagh: Cuckoo Hashing. Encyclopedia of Algorithms 2016: 478-481
Rasmus Pagh, Flemming Friche Rodler: Cuckoo hashing. J. Algorithms 51(2): 122-144 (2004)

Concurrent cuckoo hashing:
Phased, striped lock-based implementation, refine to use refinable locking

Phased, lock-based Cuckoo hashing:

Problem is to avoid having to lock long sequences of table indices due to long sequences of swaps.

Possible solution: Do the relocation in phases, maintain small, fixed buckets of items for each index in the two hash tables.



PROBE_SIZE: Fixed size of bucket, cannot be exceeded
THRESHOLD (<PROBE_SIZE): Size of bucket that will trigger relocation

```java
public abstract class PhasedCuckooHashTable<T> {
  volatile int capacity;
  volatile List<T>[][] table;
  public PhasedCuckooHashTable(int c) {
    capacity = c;
    table = (List<T>[][])new ArrayList[2][c]; // util
    for (int i=0; i<2; i++) {
      for (int j=0; j<c; j++) {
        table[i][j] = new ArrayList<T>(PROBE_SIZE);
      }
    }
  }
}
```

remove: Item is in either table[0] or table[1], remove from either if present
contains: Check table[0] and table[1]
add: If below THRESHOLD, add to fixed bucket, otherwise, try other table. If above THRESHOLD, trigger relocation of items. Resize hash table[0] and table[1] if both buckets full (PROBE_SIZE). Try relocation only up to LIMIT (cycle or chain too long)

```java
public boolean remove(T x) {
  acquire(x);
  try {
    List<T> set0 = table[0][h0(x)%capacity];
    List<T> set1 = table[1][h1(x)%capacity];
    if (set0.contains(x)) {
      set0.remove(x);
      return true;
    }
    if (set1.contains(x)) {
      set1.remove(x);
      return true;
    }
    return false;
  } finally release(x);
}
```

```
public boolean add(T x) {
  T y = null;
  acquire(x);
  int i = -1, h = -1; boolean mustresize = false;
  try {
    if (contains(x)) return false;
    List<T> set0 = table[0][h0(x)%capacity];
    List<T> set1 = table[1][h1(x)%capacity];
    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(x)%capacity;
    } else if (set1.size()<PROBE_SIZE) {
      set1.add(x); i = 1; h = h1(x)%capacity;
    } else mustresize = true;
  } finally release(x);
  // resize and/or relocate (next slide)
```

```
  // resize and/or relocate
  if (mustresize) {
    resize(); add(x);
  } else if (!relocate(i,h)) resize();
  return true; // x must have been present
}
```

Note: Locks have been released after (unsuccessful) insertion, other threads may use table concurrently

Which set? Hashed index in set

```
public boolean relocate(int i, int hi) {
  int hj;
  int j = 1-i;
  for (int r=0; r<LIMIT; i++) {
    List<T> seti = table[i][hi];
    T y = seti.get(0);
    if (i==0) hj = h1(y)%capacity;
    else hj = h0(y)%capacity;
    acquire(y);
    List<T> setj = table[j][hj];

    // relocate y from seti to setj
  }
}
```

Try to relocate oldest item from set

```
// relocate y from seti to setj
try {
  if (seti.remove(y)) {
    if (setj.size()<THRESHOLD) {
      setj.add(y); return true;
    } else if (setj.size()<PROBE_SIZE) {
      setj.add(y); i = 1-i; hi = hj; j = 1-j;
    } else {
      seti.add(y); return false;
    }
  } else if (seti.size()>=THRESHOLD) continue;
  else return true;
} finally release(y);
```

Could have been removed by other thread

Both sets full, add back and trigger resize

y gone, but still to do

… and back to the loop (LIMIT)

Locks for the items as for the simple, closed hash table

```
public class StripedCuckooHashTable<T>
       extends PhasedCuckooHashTable {
  final ReentrantLock[] lock;
  public StripedCuckooHashTable(int capacity) {
    super(capacity);
    lock = new ReentrantLock[2][capacity];
    for (int i=0; i<2; i++) {
      for (int j = 0; j<capacity; j++) {
        lock[i][j] = new ReentrantLock();
      }
    }
  }
  …
}
```

Locks for both table[0] and table[1] needed (not refinable here)

```
public final void acquire(T x) {
  lock[0][h0(x)%capacity].lock();
  lock[1][h1(x)%capacity].lock();
}

public final void release(T x) {
  lock[0][h0(x)%capacity].unlock();
  lock[1][h1(x)%capacity].unlock();
}
```

```
public void resize() {
  int oldcapacity = capacity;
  for (Lock lock0s: lock[0]) lock0s.lock();
  try {
    if (capacity!=oldcapacity) return; // other did it
    List<T>[][] oldtable = table;
    capacity = 2*capacity;
    table = (List<T>[][])new List[2][capacity];
    for (List<T>[] set: table) {
      for (int i=0; i<set.length(); i++) {
        set[i] = new ArrayList<T>(PROBE_SIZE);
      }
    }
    … // move items from old to new table
  } finally {
    for (Lock lock0s: lock[0]) lock0s.unlock();
  }
}
```

Concurrent cuckoo hashing:
Lock-free algorithms exist

Nhan Nguyen, Philippas Tsigas: Lock-Free Cuckoo Hashing. ICDCS 2014: 627-636

An adaptive open-address hash table, with elements of linear-probing and cuckoo-hashing (and attention to good cache behavior)

Maurice Herlihy, Nir Shavit, Moran Tzafrir: Hopscotch Hashing. DISC 2008: 350-364

Not lock-free (still open question?)

Recent, simple basic algorithms (open addressing, linear probing), with lots of practical considerations, comparisons to a number of other schemes

Tobias Maier, Peter Sanders, Roman Dementiev: Concurrent Hash Tables: Fast and General(?). TOPC 5(4): 16:1-16:32 (2019)