Hash Tables
Sequential Closed Hash Map

\[ h(k) = k \mod 4 \]

2 Items

\{ \text{buckets} \}

\[ 0 \rightarrow 16 \]

\[ 1 \rightarrow 9 \]
Add an Item

3 Items

$h(k) = k \mod 4$
Add Another: Collision

4 Items

$h(k) = k \mod 4$
More Collisions

\[ h(k) = k \mod 4 \]

5 Items

- 0: 16 → 4
- 1: 9
- 2: 7 → 15

Diagram:
- 0: 16 → 4
- 1: 9
- 2: 7 → 15
- 3: (empty)

5 Items

h(k) = k mod 4
More Collisions

Problem: buckets getting too long

$h(k) = k \text{ mod } 4$
Resizing

Grow the array

\[ h(k) = k \text{ mod } 8 \]

5 Items
Resizing

Adjust hash function

h(k) = k mod 8

5 Items
Resizing

h(4) = 0 mod 8

h(k) = k mod 8
Resizing

$h(k) = k \mod 8$

$h(4) = 4 \mod 8$
Resizing

\[ h(15) = 7 \mod 8 \]

The diagram illustrates the function \( h(k) = k \mod 8 \). The node 15 is mapped to 7 in the modulus space.
Resizing

\[ h(k) = k \mod 8 \]

\[ h(15) = 7 \mod 8 \]
public class SimpleHashSet {
   protected LockFreeList[] table;
   public SimpleHashSet(int capacity) {
      table = new LockFreeList[capacity];
      for (int i = 0; i < capacity; i++)
         table[i] = new LockFreeList();
   }
   ...
}

Array of lock-free lists
public class SimpleHashSet {
    protected LockFreeList[] table;

    public SimpleHashSet(int capacity) {
        table = new LockFreeList[capacity];
        for (int i = 0; i < capacity; i++)
            table[i] = new LockFreeList();
    }

    ...
public class SimpleHashSet {
    protected LockFreeList[] table;

    public SimpleHashSet(int capacity) {
        table = new LockFreeList[capacity];
        for (int i = 0; i < capacity; i++)
            table[i] = new LockFreeList();
    }
    ...

    Allocate memory
Constructor

public class SimpleHashSet {
    protected LockFreeList[] table;

    public SimpleHashSet(int capacity) {
        table = new LockFreeList[capacity];
        for (int i = 0; i < capacity; i++)
            table[i] = new LockFreeList();
    }

    ...
public boolean add(Object key) {
    int hash =
        key.hashCode() % table.length;
    return table[hash].add(key);
}
public boolean add(Object key) {
    int hash =
        key.hashCode() % table.length;
    return table[hash].add(key);
}
Add Method

```java
public boolean add(Object key) {
    int hash =
        key.hashCode() % table.length;

    return table[hash].add(key);
}
```

Call bucket’s add() method
No Brainer?

- We just saw a
  - Simple
  - Lock-free
  - Concurrent hash-based set implementation

- What’s not to like?
No Brainer?

• We just saw a
  – Simple
  – Lock-free
  – Concurrent hash-based set implementation
• What’s not to like?
• We don’t know how to resize …
Is Resizing Necessary?

- Constant-time method calls require
  - Constant-length buckets
  - Table size proportional to set size
  - As set grows, must be able to resize
Set Method Mix

• Typical load
  – 90% contains()
  – 9% add ()
  – 1% remove()

• Growing is important

• Shrinking not so much
When to Resize?

- Many reasonable policies. Here’s one.
- Pick a threshold on num of items in a bucket
  - Global threshold
    - When $\geq \frac{1}{4}$ buckets exceed this value
  - Bucket threshold
    - When any bucket exceeds this value
Coarse-Grained Locking

- Good parts
  - Simple
  - Hard to mess up
- Bad parts
  - Sequential bottleneck
Each lock associated with one bucket
Make sure table reference didn’t change between resize decision and lock acquisition.

Acquire locks in ascending order.
Allocate new super-sized table
Resize This
Striped Locks: each lock now associated with two buckets
Observations

• We grow the table, but not locks
  – Resizing lock array is tricky …

• We use sequential lists
  – Not LockFreeList lists
  – If we’re locking anyway, why pay?
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}

Returns associated read lock
Read/Write Locks

```java
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

- Returns associated read lock
- Returns associated write lock
Lock Safety Properties

• Read lock:
  – Locks out writers
  – Allows concurrent readers

• Write lock
  – Locks out writers
  – Locks out readers
Read/Write Lock

• Safety
  – If readers > 0 then writer == false
  – If writer == true then readers == 0

• Liveness?
  – Will a continual stream of readers …
  – Lock out writers?
FIFO R/W Lock

- As soon as a writer requests a lock
- No more readers accepted
- Current readers “drain” from lock
- Writer gets in
The Story So Far

• Resizing is the hard part
• Fine-grained locks
  – Striped locks cover a range (not resized)
• Read/Write locks
  – FIFO property tricky
Optimistic Synchronization

• Let the `contains()` method
  – Scan without locking

• If it finds the key
  – OK to return true
  – Actually requires a proof ….

• What if it doesn’t find the key?
Optimistic Synchronization

- If it doesn’t find the key
  - May be victim of resizing
- Must try again
  - Getting a read lock this time
- Makes sense if
  - Keys are present
  - Resizes are rare
Stop The World Resizing

• Resizing stops all concurrent operations
• What about an incremental resize?
• Must avoid locking the table
• A lock-free table + incremental resizing?
Lock-Free Resizing Problem

```
0  4  8
1  9
2
3  7  15
```
Lock-Free Resizing Problem

Need to extend table
Lock-Free Resizing Problem
Lock-Free Resizing Problem

We need a new idea…
Don’t move the items

- Move the buckets instead
- Keep all items in a single lock-free list
- Buckets become “shortcut pointers” into the list
Recursive Split Ordering

0 → 4 → 2 → 6 → 1 → 5 → 3 → 7
Recursive Split Ordering
Recursive Split Ordering
Recursive Split Ordering

List entries sorted in order that allows recursive splitting. How?
Recursive Split Ordering
Recursive Split Ordering

LSB = Least significant Bit

LSB 0

0 → 4 → 2 → 6 → 1 → 5 → 3 → 7

LSB 1

0
1

LSB = Least significant Bit
Recursive Split Ordering

LSB 00  LSB 10  LSB 01  LSB 11

0 4 2 6 1 5 3 7

0 1 2 3
Split-Order

• If the table size is $2^i$,
  – Bucket $b$ contains keys $k$
    • $k = b \pmod{2^i}$
  – bucket index consists of key's $i$ LSBs
When Table Splits

• Some keys stay
  – $b = k \mod(2^{i+1})$

• Some move
  – $b + 2^i = k \mod(2^{i+1})$

• Determined by $(i+1)^{st} \text{ bit}$
  – Counting backwards

• Key must be accessible from both
  – Keys that will move must come later
A Bit of Magic

Real keys:

0  4  2  6  1  5  3  7
A Bit of Magic

Real keys:

0  4  2  6  1  5  3  7

Real key 1 is in the 4th location

Split-order:

0  1  2  3  4  5  6  7
A Bit of Magic

Real keys:

Split-order:

Real key 1 is in 4th location
A Bit of Magic

Real keys:

000  100  010  110  001  101  011  111

Split-order:

000  001  010  011  100  101  110  111
A Bit of Magic

Real keys:

000  100  010  110  001  101  011  111

Split-order:

000  001  010  011  100  101  110  111

Just reverse the order of the key bits
Split Ordered Hashing

Order according to reversed bits
Parent Always Provides a Short Cut
Problem: how to remove a node pointed by 2 sources using CAS
Sentinel Nodes

Solution: use a Sentinel node for each bucket
Sentinel vs Regular Keys

• Want sentinel key for $i$ ordered
  – before all keys that hash to bucket $i$
  – after all keys that hash to bucket $(i-1)$
Splitting a Bucket

• We can now split a bucket
• In a lock-free manner
• Using two CAS() calls ...
  – One to add the sentinel to the list
  – The other to point from the bucket to the sentinel
Initialization of Buckets
Initialization of Buckets

Need to initialize bucket 3 to split bucket 1
Adding 10

Must initialize bucket 2
Before adding 10

= 2 mod 4
Recursive Initialization

To add 7 to the list

\[ 7 \equiv 3 \mod 4 \]

Could be \( \log n \) depth

But expected depth is constant

Must initialize bucket 1

Must initialize bucket 3
Resize

• Divide set size by total number of buckets
• If quotient exceeds threshold
  – Double tableSize field
  – Up to fixed limit
Initialize Buckets

• Buckets originally null
• If you find one, initialize it
• Go to bucket’s parent
  – Earlier nearby bucket
  – Recursively initialize if necessary
• Constant expected work
Recall: Recursive Initialization

To add 7 to the list

\[ 7 \equiv 3 \mod 4 \]

\[ 7 \equiv 1 \mod 2 \]

expected depth is constant

Must initialize bucket 1

Must initialize bucket 3
Correctness

- Linearizable concurrent set
- Theorem: $O(1)$ expected time
  - No more than $O(1)$ items expected between two dummy nodes on average
  - Lazy initialization causes at most $O(1)$ expected recursion depth in initializeBucket()
Closed (Chained) Hashing

• Advantages:
  – with \( N \) buckets, \( M \) items, Uniform \( h \)
  – retains good performance as table density \((M/N)\) increases → less resizing

• Disadvantages:
  – dynamic memory allocation
  – bad cache behavior (no locality)

Oh, did we mention that cache behavior matters on a multicore?
contains(x) – search linearly from \( h(x) \) to \( h(x) + H \) recorded in bucket.

*Attributed to Amdahl…
Linear Probing

add(x) – put in first empty bucket, and update H.
Linear Probing

• Open address means $M \cdot N$
• Expected items in bucket same as Chaining
• Expected distance till open slot:
  \[
  \frac{1}{2} \left(1 + \frac{1}{(1 - M/N)}\right)^2
  \]
  
  $M/N = 0.5 \rightarrow$ search 2.5 buckets  
  $M/N = 0.9 \rightarrow$ search 50 buckets
Linear Probing

• Advantages:
  – Good locality \(\Rightarrow\) fewer cache misses

• Disadvantages:
  – As \(M/N\) increases more cache misses
    • searching 10s of unrelated buckets
    • “Clustering” of keys into neighboring buckets
  – As computation proceeds “Contamination” by deleted items \(\Rightarrow\) more cache misses
Cuckoo Hashing

Add(x) – if \( h_1(x) \) and \( h_2(x) \) full evict \( y \) and move it to \( h_2(y) \neq h_2(x) \). Then place \( x \) in its place.

But cycles can form
Cuckoo Hashing

• Advantages:
  – contains() : deterministic 2 buckets
  – No clustering or contamination

• Disadvantages:
  – 2 tables
  – $h_i(x)$ are complex
  – As $M/N$ increases $\Rightarrow$ relocation cycles
  – Above $M/N = 0.5$ Add() does not work!
Hopscotch Hashing

• Single Array, Simple hash function
• Idea: define neighborhood of original bucket
• In neighborhood items found quickly
• Use sequences of displacements to move items into their neighborhood
Hopscotch Hashing

\[ h(x) \]

contains \( x \) – search in at most \( H \) buckets (the hop-range) based on hop-info bitmap. In practice pick \( H \) to be 32.
Hopscotch Hashing

add(x) – probe linearly to find open slot. Move the empty slot via sequence of displacements into the hop-range of \( h(x) \).
Hopscotch Hashing

- contains
  - wait-free, just look in neighborhood
Hopscotch Hashing

- contains
  - wait-free, just look in neighborhood
- add
  - expected distance same as in linear probing
Hopscotch Hashing

• \textit{contains}
  – wait-free, just look in neighborhood

• \textit{add}
  – Expected distance same as in linear probing

• \textit{resize}
  – neighborhood full less likely as $H \rightarrow \log n$
  – one word hop-info bitmap, or use smaller $H$ and default to linear probing
Advantages

• Good locality and cache behavior
• As table density \((M/N)\) increases → less resizing
• Move cost to \(\text{add()}\) from \(\text{contains()}\)
• Easy to parallelize
Recall: Concurrent Chained Hashing

Striped Locks

Lock for `add()` and unsuccessful `contains()`
Concurrent Simple Hopscotch

\[ h(x) \]

contains() is wait-free
Concurrent Simple Hopscotch

Add(x) – lock bucket, mark empty slot using CAS, add x erasing mark
Concurrent Simple Hopscotch

`add(x)` – lock bucket, mark empty slot using CAS, lock bucket and update timestamp of bucket being displaced before erasing old value.
Concurrent Simple Hopscotch

A
Is performance dominated by cache behavior?

• Run algs on state of the art multicores and uniprocessors:
  – Sun 64 way Niagara II, and
  – Intel 3GHz Xeon

• Benchmarks pre-allocated memory to eliminate effects of memory management
Sequential SPARC Throughput
90% contain, 5% insert, 5% remove

with memory pre-allocated
Sequential SPARC High-Density: Throughput; 90% contain, 5% insert, 5% remove
Sequential CoreDuo; Throughput
90% contain, 5% insert, 5% remove

Cuckoo stops here
Concurrent SPARC Throughput

90% density; 70% contain, 15% insert, 15% remove

- Hopscotch_D
- Chained_PRE
- Chained_MTM

with memory pre-allocated
with allocation

ops /ms vs CPUs

1 8 16 24 32 40 48 56 64
Concurrent SPARC Throughput
90% density; Cache-Miss per UnSuccessful-Lookup

miss / ops

CPUs

Hopscotch_D
Chained_PRE
Chained_MTM
Summary

• Chained hash with striped locking is simple and effective in many cases
• Hopscotch with striped locking great cache behavior
• If incremental resizing needed go for split-ordered