CHAPTER

Concurrent hashing and natural parallelism 13

13.1 Introduction

In earlier chapters, we studied how to extract parallelism from data structures like queues, stacks, and counters, which seemed to provide few opportunities for parallelism. In this chapter we take the opposite approach. We study *concurrent hashing*, a problem that seems to be "naturally parallelizable" or, using a more technical term, *disjoint-access-parallel*, meaning that concurrent method calls are likely to access disjoint locations, implying that there is little need for synchronization.

We study hashing in the context of `Set` implementations. Recall that the `Set` interface provides the following methods:

- `add`(x) adds x to the set, and returns *true* if x was absent, and *false* otherwise;
- `remove`(x) removes x from the set, and returns *true* if x was present, and *false* otherwise; and
- `contains`(x) returns *true* if x is present, and *false* otherwise.

In sequential programming, hashing is often used to implement these methods with constant average time complexity. In this chapter, we aim to do the same for concurrent `Set` implementations. (By contrast, the `Set` implementations of Chapter 9 require time linear in the size of the set.) Although hashing seems naturally parallelizable, devising an effective concurrent hash-based `Set` implementations is far from trivial.

When designing `Set` implementations, we should keep the following principle in mind: *We can buy more memory, but we cannot buy more time.* Between a fast algorithm that consumes more memory and a slower algorithm that consumes less memory, we tend to prefer the faster algorithm (within reason).

A *hash set* (sometimes called a *hash table*) is an efficient way to implement a set. A hash set is typically implemented as an array, called the *table*. Each table entry is a reference to zero or more *items*. A *hash function* maps items to integers so that distinct items usually map to distinct values. (Java provides each object with a `hashCode()` method that serves this purpose.) To add, remove, or test an item for membership, apply the hash function to the item (modulo the table size) to identify the table entry associated with that item. (We call this step *hashing* the item.)

Any hash set algorithm must deal with *collisions*: what to do when distinct items hash to the same table entry. *Closed addressing* simply stores a set of items, traditionally called a *bucket*, at each entry. *Open addressing* attempts to find an alternative table entry for the item, for example by applying alternative hash functions.

It is sometimes necessary to *resize* the table. In closed-address hash sets, buckets may become too large to search efficiently. In open-address hash sets, the table may become too full to find alternative table entries.

Anecdotal evidence suggests that in most applications, sets are subject to the following distribution of method calls: 90% `contains()`, 9% `add()`, and 1% `remove()` calls. As a practical matter, sets are more likely to grow than to shrink, so we focus here on *extensible hashing*, in which hash sets only grow (shrinking them is a problem for the exercises).

13.2 Closed-address hash sets

We start by defining a *base* hash set implementation common to all the concurrent closed-address hash sets we consider here. Later, we extend the base hash set with different synchronization mechanisms.

The `BaseHashSet<T>` class is an *abstract class*, that is, it does not implement all its methods. Fig. 13.1 shows its fields, constructor, and abstract methods. The `table[]` field is an array of buckets, each of which is a set implemented as a list. For convenience, we use `ArrayList<T>`, which supports the standard sequential `add()`, `remove()`, and `contains()` methods. We sometimes refer to the length of the `table[]` array, that is, the number of buckets in it, as its *capacity*. The `setSize` field stores the number of items in the set. The constructor takes the initial capacity of the table as an argument.

PRAGMA 13.2.1

Here and elsewhere, we use the standard Java `List<T>` interface (from package `java.util`). A `List<T>` is an ordered collection of `T` objects, where `T` is a type. It specifies many methods, of which we use the following: `add`(x), which appends x to the end of the list; `get`(i), which returns (but does not remove) the item at position i; and `contains`(x), which returns *true* if the list contains x.

The `List` interface is implemented by many classes. Here, we use the `ArrayList` class for convenience.

The *abstract methods* of `BaseHashSet<T>` class, which it does not implement, are: `acquire`(x), which acquires the locks necessary to manipulate item x; `release`(x), which releases them; `resize()`, which doubles the capacity of the `table[]` array; and `policy()`, which decides whether to resize. The `acquire`(x) method must be *reentrant*, meaning that if a thread that has already called `acquire`(x) makes the same call, then it will proceed without deadlocking with itself.

Fig. 13.2 shows the `contains`(x) and `add`(x) methods of the `BaseHashSet<T>` class. Each method first calls `acquire`(x) to perform the necessary synchronization and then enters a **`try`** block whose **`finally`** block calls `release`(x). The `contains`(x) method simply tests whether x is present in the associated bucket (line 21), while `add`(x) adds x to the list if it is not already present (line 30).

```
public abstract class BaseHashSet<T> {
  protected volatile List<T>[] table;
  protected AtomicInteger setSize;
  public BaseHashSet(int capacity) {
    setSize = new AtomicInteger(0);
    table = (List<T>[]) new List[capacity];
    for (int i = 0; i < capacity; i++) {
      table[i] = new ArrayList<T>();
    }
  }
  ...
  public abstract void acquire(T x);
  public abstract void release(T x);
  public abstract void resize();
  public abstract boolean policy();
}
```

FIGURE 13.1

`BaseHashSet<T>` class: fields, constructor, and abstract methods.

How big should the bucket array be to ensure that method calls take constant expected time? Consider an `add`(x) call. The first step, hashing x to determine the bucket, takes constant time. The second step, checking whether x is in the bucket, requires traversing the list. This traversal takes constant expected time only if the lists have constant expected length, so the table capacity should be proportional to the number of items in the set, that is, the size of the set. Because the set may vary in size over time, to ensure that method call times remain (more or less) constant, we must occasionally *resize* the table to ensure that list lengths remain (more or less) constant.

We still need to decide *when* to resize the table, and how `resize`() synchronizes with other methods. There are many reasonable alternatives. For closed-addressing algorithms, one simple strategy is to resize the table when the average bucket size exceeds a fixed threshold. An alternative policy employs two fixed quantities, the *bucket threshold* and the *global threshold*; we resize the table

- if more than, say, a quarter of the buckets exceed the bucket threshold, or
- if any single bucket exceeds the global threshold.

Either of these strategies work can well in practice. For simplicity, we adopt the first policy in this chapter.[1]

[1] This choice introduces a scalability bottleneck, threads adding or removing items all contend on the counter that tracks the size of the set. We use an `AtomicInteger`, which limits scalability. It can be replaced by other, more scalable, counter implementations, if necessary.

```
public boolean contains(T x) {
  acquire(x);
  try {
    int myBucket = x.hashCode() % table.length;
    return table[myBucket].contains(x);
  } finally {
    release(x);
  }
}
public boolean add(T x) {
  boolean result = false;
  acquire(x);
  try {
    int myBucket = x.hashCode() % table.length;
    if (! table[myBucket].contains(x)) {
      table[myBucket].add(x);
      result = true;
      setSize.getAndIncrement();
    }
  } finally {
    release(x);
  }
  if (policy())
    resize();
  return result;
}
```

FIGURE 13.2

BaseHashSet<T> class: the contains() and add() methods hash the item to choose a bucket.

13.2.1 A coarse-grained hash set

Fig. 13.3 shows the CoarseHashSet<T> class's fields, constructor, and acquire(x) and release(x) methods. The constructor first initializes its superclass (line 4). Synchronization is provided by a single reentrant lock (line 2), acquired by acquire(x) (line 8) and released by release(x) (line 11).

Fig. 13.4 shows the CoarseHashSet<T> class's policy() and resize() methods. We use a simple policy: We resize when the average bucket length exceeds 4 (line 16). The resize() method locks the set (line 19), and checks that no other thread has resized the table in the meantime (line 22). It then allocates and initializes a new table with double the capacity (lines 24–28) and transfers items from the old to the new buckets (lines 29–33). Finally, it unlocks the set (line 35).

Like the coarse-grained list studied in Chapter 9, the coarse-grained hash set is easy to understand and easy to implement. Unfortunately, it is also a sequential bottleneck. Method calls take effect in a one-at-a-time order, even when they access separate buckets (and do not resize).

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

FIGURE 13.3

`CoarseHashSet<T>` class: fields, constructor, and `acquire()` and `release()` methods.

```
  public boolean policy() {
    return setSize.get() / table.length > 4;
  }
  public void resize() {
    lock.lock();
    try {
      if (!policy()) {
        return; // someone beat us to it
      }
      int newCapacity = 2 * table.length;
      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) {
        for (T x : bucket) {
          table[x.hashCode() % table.length].add(x);
        }
      }
    } finally {
      lock.unlock();
    }
  }
```

FIGURE 13.4

`CoarseHashSet<T>` class: the `policy()` and `resize()` methods.

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

FIGURE 13.5

`StripedHashSet<T>` class: fields, constructor, and `acquire()` and `release()` methods.

13.2.2 A striped hash set

We now present a closed-address hash table with greater parallelism and less lock contention. Instead of using a single lock to synchronize the entire set, we split the set into independently synchronized pieces. We introduce a technique called *lock striping*, which will be useful for other data structures as well. Fig. 13.5 shows the fields and constructor for the `StripedHashSet<T>` class. The set is initialized with an array `locks`[] of L locks, and an array `table`[] of $N = L$ buckets, where each bucket is an unsynchronized `List<T>`. Although these arrays are initially of the same length, `table`[] will grow when the hash table is resized, but `lock`[] will not. When the hash table is resized, we double the table capacity N without changing the lock array size L; lock i protects each table entry j, where $j = i \pmod{L}$. The `acquire`(x) and `release`(x) methods use x's hash code to pick which lock to acquire or release. An example illustrating how a `StripedHashSet<T>` is resized appears in Fig. 13.6.

There are two reasons not to grow the lock array when we grow the table:

- Associating a lock with every table entry could consume too much space, especially when tables are large and contention is low.
- While resizing the table is straightforward, resizing the lock array (while in use) is more complex, as discussed in Section 13.2.3.

Resizing a `StripedHashSet` (Fig. 13.7) is almost identical to resizing a `CoarseHashSet<>`. One difference is that in `StripedHashSet`, the `resize`() method acquires all the locks in `lock`[] in ascending order (lines 19–21). It cannot deadlock with a `contains`(), `add`(), or `remove`() call because these methods acquire only a single lock. A `resize`() call cannot deadlock with another `resize`() call because both

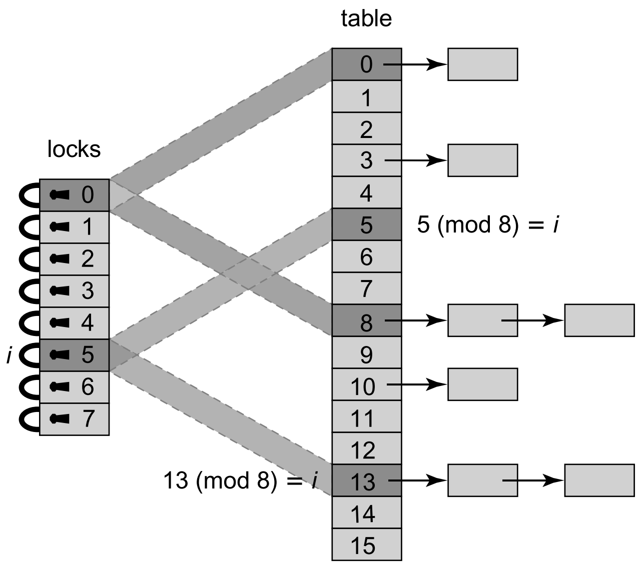

FIGURE 13.6

Resizing a `StripedHashSet` lock-based hash table. As the table grows, the striping is adjusted to ensure that each lock covers $2^{N/L}$ entries. In the figure above, N=16 and L=8. When N is doubled from 8 to 16, the memory is striped so that lock i=5, for example, covers both locations that are equal to 5 modulo L.

calls start without holding any locks, and acquire the locks in the same order. What if two or more threads try to resize at the same time? As in `CoarseHashSet<T>`, after a thread has acquired all the locks, if it discovers that some other thread has changed the table capacity (line 24), then it releases the locks and gives up. (It could just double the table size anyway, since it already holds all the locks.) Otherwise, it creates a new `table[]` array with twice the capacity (line 26), and transfers items from the old table to the new (line 31). Finally, it releases the locks (line 37).

To summarize, striped locking permits more concurrency than a single coarse-grained lock because method calls whose items hash to different locks can proceed in parallel. The `add()`, `contains()`, and `remove()` methods take constant expected time, but `resize()` takes linear time and is a "stop-the-world" operation: It halts all concurrent method calls while it increases the table's capacity.

13.2.3 A refinable hash set

What if we want to refine the granularity of locking as the table size grows, so that the number of locations in a stripe does not continuously grow? Clearly, if we want to resize the lock array, then we need to rely on another form of synchronization. Resizing is rare, so our principal goal is to devise a way to permit the lock array to be resized without substantially increasing the cost of normal method calls.

```
public void resize() {
  for (Lock lock : locks) {
    lock.lock();
  }
  try {
    if (!policy()) {
      return; // someone beat us to it
    }
    int newCapacity = 2 * table.length;
    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) {
      for (T x : bucket) {
        table[x.hashCode() % table.length].add(x);
      }
    }
  } finally {
    for (Lock lock : locks) {
      lock.unlock();
    }
  }
}
```

FIGURE 13.7

`StripedHashSet<T>` class: To resize the set, lock each lock in order, and then check that no other thread has resized the table in the meantime.

Fig. 13.8 shows the fields and constructor for the `RefinableHashSet<T>` class. To add a higher level of synchronization, we introduce a global `owner` field that combines a Boolean value with a reference to a thread in an `AtomicMarkableReference<Thread>` so they can be modified atomically (see Pragma 9.8.1). We use `owner` as a mutual exclusion flag between the `resize()` method and any of the `add()` methods, so that while resizing, there will be no successful updates, and while updating, there will be no successful resizes. Normally, the Boolean value is *false*, meaning that the set is not in the middle of resizing. While a resizing is in progress, however, the Boolean value is *true*, and the associated reference indicates the thread that is in charge of resizing. Every `add()` call must read the `owner` field. Because resizing is rare, the value of `owner` should usually be cached.

Each method locks the bucket for x by calling `acquire`(x), shown in Fig. 13.9. It spins until no other thread is resizing the set (lines 19–21), and then reads the lock array (line 22). It then acquires the item's lock (line 24), and checks again, this time while holding a lock (line 26), to make sure no other thread is resizing, and that no resizing took place between lines 21 and 26.

If it passes this test, the thread can proceed. Otherwise, the lock it has acquired could be out-of-date because of an ongoing update, so it releases it and starts over.

```
1  public class RefinableHashSet<T> extends BaseHashSet<T>{
2    AtomicMarkableReference<Thread> owner;
3    volatile ReentrantLock[] locks;
4    public RefinableHashSet(int capacity) {
5      super(capacity);
6      locks = new ReentrantLock[capacity];
7      for (int i = 0; i < capacity; i++) {
8        locks[i] = new ReentrantLock();
9      }
10     owner = new AtomicMarkableReference<Thread>(null, false);
11   }
12   ...
13 }
```

FIGURE 13.8

`RefinableHashSet<T>` class: fields and constructor.

```
14   public void acquire(T x) {
15     boolean[] mark = {true};
16     Thread me = Thread.currentThread();
17     Thread who;
18     while (true) {
19       do {
20         who = owner.get(mark);
21       } while (mark[0] && who != me);
22       ReentrantLock[] oldLocks = locks;
23       ReentrantLock oldLock = oldLocks[x.hashCode() % oldLocks.length];
24       oldLock.lock();
25       who = owner.get(mark);
26       if ((!mark[0] || who == me) && locks == oldLocks) {
27         return;
28       } else {
29         oldLock.unlock();
30       }
31     }
32   }
33   public void release(T x) {
34     locks[x.hashCode() % locks.length].unlock();
35   }
```

FIGURE 13.9

`RefinableHashSet<T>` class: `acquire()` and `release()` methods.

When starting over, it will first spin until the current resize completes (lines 19–21) before attempting to acquire the locks again. The `release`(x) method releases the lock acquired by `acquire`(x).

The `resize()` method (Fig. 13.10) is similar to the `resize()` method for the `StripedHashSet` class. However, instead of acquiring all the locks in `lock[]`, the

```
36   public void resize() {
37     boolean[] mark = {false};
38     Thread me = Thread.currentThread();
39     if (owner.compareAndSet(null, me, false, true)) {
40       try {
41         if (!policy()) { // someone else resized first
42           return;
43         }
44         quiesce();
45         int newCapacity = 2 * table.length;
46         List<T>[] oldTable = table;
47         table = (List<T>[]) new List[newCapacity];
48         for (int i = 0; i < newCapacity; i++)
49           table[i] = new ArrayList<T>();
50         locks = new ReentrantLock[newCapacity];
51         for (int j = 0; j < locks.length; j++) {
52           locks[j] = new ReentrantLock();
53         }
54         initializeFrom(oldTable);
55       } finally {
56         owner.set(null, false);
57       }
58     }
59   }
```

FIGURE 13.10

RefinableHashSet<T> class: resize() method.

```
60   protected void quiesce() {
61     for (ReentrantLock lock : locks) {
62       while (lock.isLocked()) {}
63     }
64   }
```

FIGURE 13.11

RefinableHashSet<T> class: quiesce() method.

method attempts to set itself as the owner (line 39) and then calls quiesce() (line 44) to ensure that no other thread is in the middle of an add(), remove(), or contains() call. The quiesce() method (Fig. 13.11) visits each lock and waits until it is unlocked.

The acquire() and the resize() methods guarantee mutually exclusive access via the flag principle using the mark field of the owner flag and the table's locks array: acquire() first acquires its locks and then reads the mark field, while resize() first sets mark and then reads the locks during the quiesce() call. This ordering ensures that any thread that acquires a lock after quiesce() has completed will see that the set is in the process of being resized, and will back off until the resizing is complete.

Similarly, `resize()` will first set the `mark` field and then read the locks, and will not proceed while any `add()`, `remove()`, or `contains()` call holds its lock.

To summarize, we have designed a hash table in which both the number of buckets and the number of locks can be continually resized. One limitation of this algorithm is that threads cannot access the items in the table during a resize.

13.3 A lock-free hash set

The next step is to make the hash set implementation lock-free, and to make resizing *incremental*, meaning that each `add()` method call performs a small fraction of the work associated with resizing. This way, we do not need to "stop the world" to resize the table. Each of the `contains()`, `add()`, and `remove()` methods takes constant expected time.

To make resizable hashing lock-free, it is not enough to make the individual buckets lock-free: Resizing the table requires atomically moving entries from old buckets to new buckets. If the table doubles in capacity, then we must split the items in the old bucket between two new buckets. If this move is not done atomically, entries might be temporarily lost or duplicated. Without locks, we must synchronize using atomic methods such as `compareAndSet()`. Unfortunately, these methods operate only on a single memory location, which makes it difficult to move a node atomically from one linked list to another.

13.3.1 Recursive split-ordering

We now describe a hash set implementation that works by flipping the conventional hashing structure on its head:

Instead of moving the items among the buckets, move the buckets among the items.

More specifically, keep all items in a single lock-free linked list, similar to the `LockFreeList` class studied in Chapter 9. A bucket is just a reference into the list. As the list grows, we introduce additional bucket references so that no object is ever too far from the start of a bucket. This algorithm ensures that once an item is placed in the list, it is never moved, but it does require that items be inserted according to a *recursive split-order* algorithm that we describe shortly.

Fig. 13.12 illustrates a lock-free hash set implementation. It shows two components: a lock-free linked list and an expanding array of references into the list. These references are *logical* buckets. Any item in the hash set can be reached by traversing the list from its head, while the bucket references provide shortcuts into the list to minimize the number of list nodes traversed when searching. The principal challenge is ensuring that the bucket references into the list remain well distributed as the number of items in the set grows. Bucket references should be spaced evenly enough to allow constant-time access to any node. It follows that new buckets must be created and assigned to sparsely covered regions in the list.

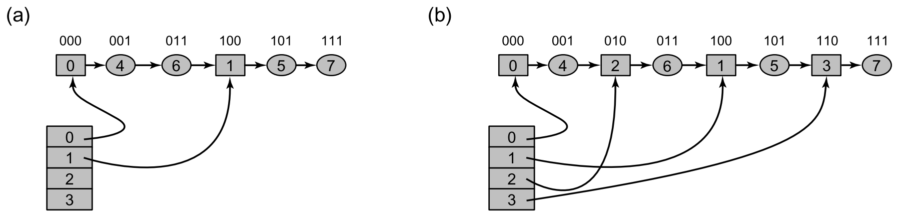

FIGURE 13.12

This figure explains the recursive nature of the split ordering. Part (a) shows a split-ordered list consisting of two buckets. The array of buckets refer into a single linked list. The split-ordered keys (above each node) are the reverse of the bit-wise representation of the items' keys. The active bucket array entries 0 and 1 have special sentinel nodes within the list (square nodes), while other (ordinary) nodes are round. Items 4 (whose reverse bit order is "001") and 6 (whose reverse bit order is "011") are in bucket 0, since the least significant bit (LSB) of the original key is "0." Items 5 and 7 (whose reverse bit orders are "101" and "111," respectively) are in bucket 1, since the LSB of their original key is 1. Part (b) shows how each of the two buckets is split in half once the table capacity grows from two buckets to four. The reverse-bit values of the two added buckets 2 and 3 happen to perfectly split buckets 0 and 1.

As before, the capacity N of the hash set is always a power of two. The bucket array initially has capacity 2 and all bucket references are *null*, except for the bucket at index 0, which refers to an empty list. We use the variable `bucketSize` to denote this changing capacity of the bucket structure. Each entry in the bucket array is initialized when first accessed, and subsequently refers to a node in the list.

When an item with hash code k is inserted, removed, or searched for, the hash set uses bucket index $k \pmod{N}$. As with earlier hash set implementations, we decide when to double the table capacity by consulting a `policy()` method. Here, however, the table is resized incrementally by the methods that modify it, so there is no explicit `resize()` method. If the table capacity is 2^i, then the bucket index is the integer represented by the key's i LSBs; in other words, each bucket b contains items each of whose hash code k satisfies $k = b \pmod{2^i}$.

Because the hash function depends on the table capacity, we must be careful when the table capacity changes. An item inserted before the table was resized must be accessible afterwards from both its previous and current buckets. When the capacity grows to 2^{i+1}, the items in bucket b are split between two buckets: Those for which $k = b \pmod{2^{i+1}}$ remain in bucket b, while those for which $k = b + 2^i \pmod{2^{i+1}}$ migrate to bucket $b + 2^i$. Here is the key idea behind the algorithm: We ensure that these two groups of items are positioned one after the other in the list, so that splitting bucket b is achieved by simply setting bucket $b + 2^i$ after the first group of items and before the second. This organization keeps each item in the second group accessible from bucket b.

As depicted in Fig. 13.12, items in the two groups are distinguished by their ith binary digits (counting backwards, from least significant to most significant). Those with digit 0 belong to the first group, and those with 1 to the second. The next hash table doubling will cause each group to split again into two groups differentiated by the $(i+1)$st bit, and so on. For example, the items 4 (“100” binary) and 6 (“110”) share the same LSB. When the table capacity is 2^1, they are in the same bucket, but when it grows to 2^2, they will be in distinct buckets because their second bits differ.

This process induces a total order on items, which we call *recursive split-ordering*, as can be seen in Fig. 13.12. Given a key’s hash code, its order is defined by its bit-reversed value.

To recapitulate: a *split-ordered hash set* is an array of buckets, where each bucket is a reference into a lock-free list where nodes are sorted by their bit-reversed hash codes. The number of buckets grows dynamically, and each new bucket is initialized when accessed for the first time.

To avoid an awkward “corner case” that arises when deleting a node referenced by a bucket reference, we add a *sentinel* node, which is never deleted, to the start of each bucket. Specifically, suppose the table capacity is 2^{i+1}. The first time that bucket $b+2^i$ is accessed, a sentinel node is created with key $b+2^i$. This node is inserted in the list via bucket b, the *parent* bucket of $b+2^i$. Under split-ordering, $b+2^i$ precedes all items of bucket $b+2^i$, since those items must end with $(i+1)$ bits forming the value $b+2^i$. This value also comes after all the items of bucket b that do not belong to $b+2^i$: They have identical LSBs, but their ith bit is 0. Therefore, the new sentinel node is positioned in the exact list location that separates the items of the new bucket from the remaining items of bucket b. To distinguish sentinel items from ordinary items, we set the most significant bit (MSB) of ordinary items to 1, and leave the sentinel items with 0 at the MSB. Fig. 13.17 illustrates two methods: `makeOrdinaryKey()`, which generates a split-ordered key for an object, and `makeSentinelKey()`, which generates a split-ordered key for a bucket index.

Fig. 13.13 illustrates how inserting a new key into the set can cause a bucket to be initialized. The split-order key values are written above the nodes using 8-bit words. For instance, the split-order value of 3 is the bit-reverse of its binary representation, which is 11000000. The square nodes are the sentinel nodes corresponding to buckets with original keys that are 0, 1, and 3 modulo 4 with their MSB being 0. The split-order keys of ordinary (round) nodes are exactly the bit-reversed images of the original keys after turning on their MSB. For example, items 9 and 13 are in the “1 (mod 4)” bucket, which can be recursively split in two by inserting a new node between them. The sequence of figures describes an object with hash code 10 being added when the table capacity is 4 and buckets 0, 1, and 3 are already initialized.

The table is grown incrementally; there is no explicit resize operation. Recall that each bucket is a linked list, with nodes ordered based on the split-ordered hash values. As mentioned earlier, the table resizing mechanism is independent of the policy used to decide when to resize. To keep the example concrete, we implement the following policy: We use a shared counter to allow `add()` calls to track the average bucket load. When the average load crosses a threshold, we double the table capacity.

To avoid technical distractions, we keep the array of buckets in a large, fixed-size array. We start out using only the first array entry, and use progressively more of the array as the capacity grows. When the `add()` method accesses an uninitialized

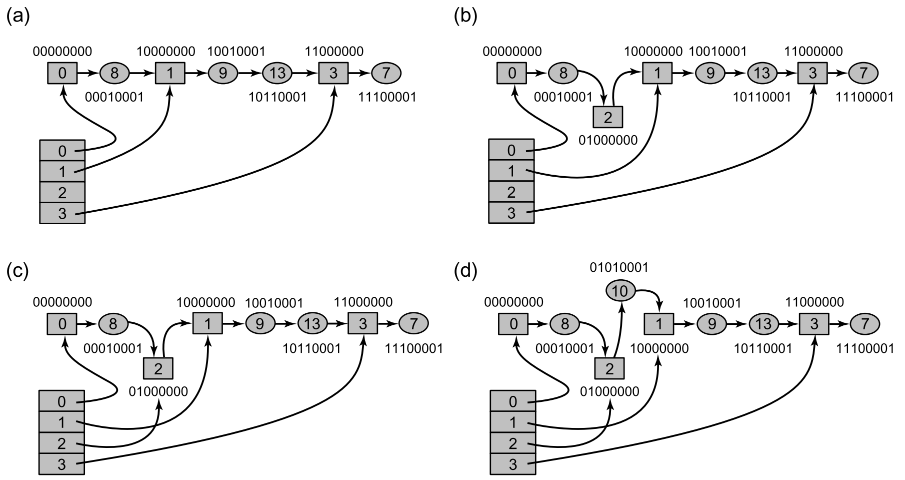

FIGURE 13.13

How the add() method places key 10 to the lock-free table. As in earlier figures, the split-order key values, expressed as 8-bit binary words, appear above the nodes. For example, the split-order value of 1 is the bit-wise reversal of its binary representation. In step (a), buckets 0, 1, and 3 are initialized, but bucket 2 is uninitialized. In step (b), an item with hash value 10 is inserted, causing bucket 2 to be initialized. A new sentinel is inserted with split-order key 2. In step (c), bucket 2 is assigned a new sentinel. Finally, in step (d), the split-order ordinary key 10 is added to bucket 2.

bucket that should have been initialized given the current table capacity, it initializes it. While conceptually simple, this design is far from ideal, since the fixed array size limits the ultimate number of buckets. In practice, it would be better to represent the buckets as a multilevel tree structure, which would cover the machine's full memory size, a task we leave as an exercise.

13.3.2 The BucketList class

Fig. 13.14 shows the fields, the constructor, and some utility methods of the BucketList class that implements the lock-free list used by the split-ordered hash set. Although this class is essentially the same as the LockFreeList class from Chapter 9, there are two important differences. The first is that items are sorted in recursive-split order, not simply by hash code. The makeOrdinaryKey() and makeSentinelKey() methods (lines 10 and 14) show how we compute these split-ordered keys. (To ensure that reversed keys are positive, we use only the lower three bytes of the hash code.) Fig. 13.15 shows how the contains() method is modified to use the split-ordered key. (As in the LockFreeList class, the find(x) method returns a record containing x's node, if it exists, along with the immediately preceding and subsequent nodes.)

The second difference is that while the LockFreeList class uses only two sentinels, one at each end of the list, the BucketList<T> class places a sentinel at the start of each

```
public class BucketList<T> implements Set<T> {
  static final int HI_MASK = 0x80000000;
  static final int MASK = 0x00FFFFFF;
  Node head;
  public BucketList() {
    head = new Node(0);
    head.next =
     new AtomicMarkableReference<Node>(new Node(Integer.MAX_VALUE), false);
  }
  public int makeOrdinaryKey(T x) {
    int code = x.hashCode() & MASK; // take 3 lowest bytes
    return reverse(code | HI_MASK);
  }
  private static int makeSentinelKey(int key) {
    return reverse(key & MASK);
  }
  ...
}
```

FIGURE 13.14

BucketList<T> class: fields, constructor, and utilities.

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

FIGURE 13.15

BucketList<T> class: the contains() method.

new bucket whenever the table is resized. It requires the ability to insert sentinels at intermediate positions within the list, and to traverse the list starting from such sentinels. The BucketList<T> class provides a getSentinel(x) method (Fig. 13.16) that takes a bucket index, finds the associated sentinel (inserting it if absent), and returns the tail of the BucketList<T> starting from that sentinel.

13.3.3 The LockFreeHashSet<T> class

Fig. 13.17 shows the fields and constructor for the LockFreeHashSet<T> class. The set has the following mutable fields: bucket is an array of BucketList<T> references into the list of items, bucketSize is an atomic integer that tracks how much of the bucket array is currently in use, and setSize is an atomic integer that tracks how many objects are in the set. These fields are used to decide when to resize.

```
  public BucketList<T> getSentinel(int index) {
    int key = makeSentinelKey(index);
    boolean splice;
    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);
        splice = pred.next.compareAndSet(curr, node, false, false);
        if (splice)
          return new BucketList<T>(node);
        else
          continue;
      }
    }
  }
```

FIGURE 13.16

`BucketList<T>` class: `getSentinel()` method.

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

FIGURE 13.17

`LockFreeHashSet<T>` class: fields and constructor.

Fig. 13.18 shows the `LockFreeHashSet<T>` class's `add()` method. If x has hash code k, `add`(x) retrieves bucket k (mod N), where N is the current table size, initializing it if necessary (line 15). It then calls the `BucketList<T>`'s `add`(x) method. If x was not already present (line 18), it increments `setSize` and checks whether to increase `bucketSize`, the number of active buckets. The `contains`(x) and `remove`(x) methods work in much the same way.

```
  public boolean add(T x) {
    int myBucket = 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 / bucketSizeNow > THRESHOLD)
      bucketSize.compareAndSet(bucketSizeNow, 2 * bucketSizeNow);
    return true;
  }
```

FIGURE 13.18

LockFreeHashSet<T> class: add() method.

```
  private BucketList<T> getBucketList(int myBucket) {
    if (bucket[myBucket] == null)
      initializeBucket(myBucket);
    return bucket[myBucket];
  }
  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;
  }
  private int getParent(int myBucket){
    int parent = bucketSize.get();
    do {
      parent = parent >> 1;
    } while (parent > myBucket);
    parent = myBucket - parent;
    return parent;
  }
```

FIGURE 13.19

LockFreeHashSet<T> class: If a bucket is uninitialized, initialize it by adding a new sentinel. Initializing a bucket may require initializing its parent.

Fig. 13.19 shows the initialBucket() method, whose role is to initialize the bucket array entry at a particular index, setting that entry to refer to a new sentinel node. The sentinel node is first created and added to an existing *parent* bucket, and then the array entry is assigned a reference to the sentinel. If the parent bucket is not

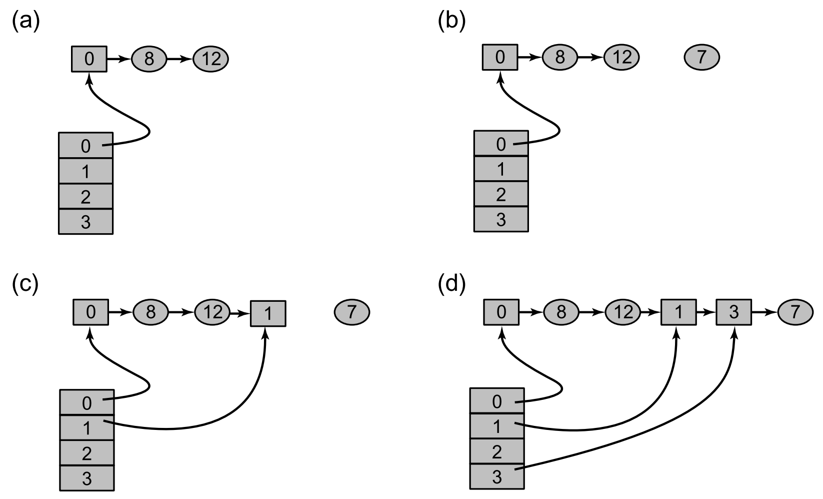

FIGURE 13.20

Recursive initialization of lock-free hash table buckets. (a) The table has four buckets; only bucket 0 is initialized. (b) We wish to insert the item with key 7. Bucket 3 now requires initialization, which in turn requires recursive initialization of bucket 1. (c) Bucket 1 is initialized by first adding the 1 sentinel to the list, then setting the bucket to this sentinel. (d) Then bucket 3 is initialized in a similar fashion, and finally 7 is added to the list. In the worst case, insertion of an item may require recursively initializing a number of buckets logarithmic in the table size, but it can be shown that the expected length of such a recursive sequence is constant.

initialized (line 31), `initialBucket()` is applied recursively to the parent. To control the recursion, we maintain the invariant that the parent index is less than the new bucket index. It is also prudent to choose the parent index as close as possible to the new bucket index, but still preceding it. We compute this index by unsetting the bucket index's most significant nonzero bit (line 39).

The `add()`, `remove()`, and `contains()` methods require a constant expected number of steps to find a key (or determine that the key is absent). To initialize a bucket in a table of `bucketSize` N, the `initialBucket()` method may need to recursively initialize (i.e., split) as many as $O(\log N)$ of its parent buckets to allow the insertion of a new bucket. An example of this recursive initialization is shown in Fig. 13.20. In part (a), the table has four buckets; only bucket 0 is initialized. In part (b), the item with key 7 is inserted. Bucket 3 now requires initialization, further requiring recursive initialization of bucket 1. In part (c), bucket 1 is initialized. Finally, in part (d), bucket 3 is initialized. Although the worst-case complexity in such a case is logarithmic, not constant, it can be shown that the *expected length* of any such recursive sequence of splits is constant, making the overall expected complexity of all the hash set operations constant.

13.4 An open-address hash set

We now turn our attention to a concurrent open-address hashing algorithm. Open-address hashing, in which each table entry holds a single item rather than a set, seems harder to make concurrent than closed-address hashing. We base our concurrent algorithm on a sequential algorithm known as cuckoo hashing.

13.4.1 Cuckoo hashing

Cuckoo hashing is a (sequential) hashing algorithm in which a newly added item displaces any earlier item occupying the same slot.[2] For brevity, a *table* is a k-entry array of items. For a hash set of size $N = 2k$, we use a two-entry array `table`[] of tables,[3] and two independent hash functions,

$$h_0, h_1 : \mathit{KeyRange} \to 0, \ldots, k-1$$

(denoted as `hash0()` and `hash1()` in the code), mapping the set of possible keys to entries in the array. To test whether a value x is in the set, `contains`(x) tests whether either `table`$[0][h_0(x)]$ or `table`$[1][h_1(x)]$ is equal to x. Similarly, `remove`(x) checks whether x is in either `table`$[0][h_0(x)]$ or `table`$[1][h_1(x)]$, and removes it if found.

The `add`(x) method (Fig. 13.21) is the most interesting. It successively "kicks out" conflicting items until every key has a slot. To add x, the method swaps x with y, the current occupant of `table`$[0][h_0(x)]$ (line 6). If the prior value y was *null*, it is done (line 7). Otherwise, it swaps the newly nestless value y for the current occupant of `table`$[1][h_1(y)]$ in the same way (line 8). As before, if the prior value was *null*, it is done. Otherwise, the method continues swapping entries (alternating tables) until it finds an empty slot. An example of such a sequence of displacements appears in Fig. 13.22.

We might not find an empty slot, either because the table is full, or because the sequence of displacements forms a cycle. We therefore need an upper limit on the number of successive displacements we are willing to undertake (line 5). When this limit is exceeded, we resize the hash table, choose new hash functions (line 12), and start over (line 13).

Sequential cuckoo hashing is attractive for its simplicity. It provides constant-time `contains`() and `remove`() methods, and it can be shown that over time, the average number of displacements caused by each `add`() call will be constant. Experimental evidence shows that sequential cuckoo hashing works well in practice.

[2] Cuckoos are a family of birds (not clocks) found in North America and Europe. Most species are nest parasites: they lay their eggs in other birds' nests. Cuckoo chicks hatch early, and quickly push the other eggs out of the nest.

[3] This division of the table into two arrays helps in presenting the concurrent algorithm. There are sequential cuckoo hashing algorithms that use, for the same number of hashed items, only a single array of size $2k$.

```java
public boolean add(T x) {
  if (contains(x)) {
    return false;
  }
  for (int i = 0; i < LIMIT; i++) {
    if ((x = swap(0, hash0(x), x)) == null) {
      return true;
    } else if ((x = swap(1, hash1(x), x)) == null) {
      return true;
    }
  }
  resize();
  add(x);
  }
```

FIGURE 13.21

Sequential cuckoo hashing: the `add()` method.

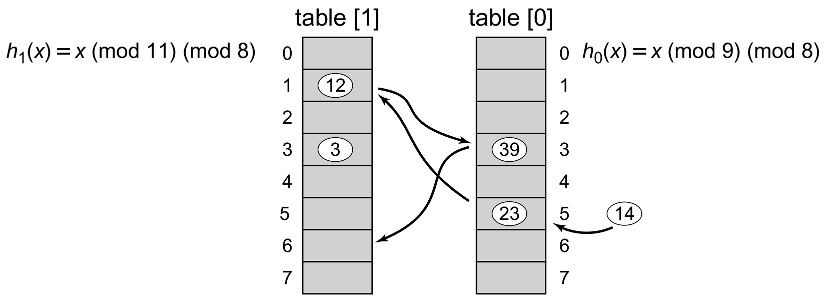
table [1]
table [0]
$h_1(x) = x \pmod{11} \pmod{8}$
$h_0(x) = x \pmod{9} \pmod{8}$
0
1
2
3
4
5
6
7
12
3
39
23
14

FIGURE 13.22

A sequence of displacements starts when an item with key 14 finds both locations Table[0][h_0(14)] and Table[1][h_1(14)] taken by the values 3 and 23, and ends when the item with key 39 is successfully placed in Table[1][h_1(39)].

13.4.2 Concurrent cuckoo hashing

The principal obstacle to making the sequential cuckoo hashing algorithm concurrent is the `add()` method's need to perform a long sequence of swaps. To address this problem, we now define an alternative cuckoo hashing algorithm, the `PhasedCuckooHashSet<T>` class. We break up each method call into a sequence of *phases*, where each phase adds, removes, or displaces a single item x.

Rather than organizing the set as a two-dimensional table of items, we use a two-dimensional table of *probe sets*, where a probe set is a constant-sized set of items with the same hash code. Each probe set holds at most `PROBE_SIZE` items, but the algorithm tries to ensure that when the set is quiescent (i.e., no method calls are in progress), each probe set holds no more than `THRESHOLD` < `PROBE_SIZE` items. An example of

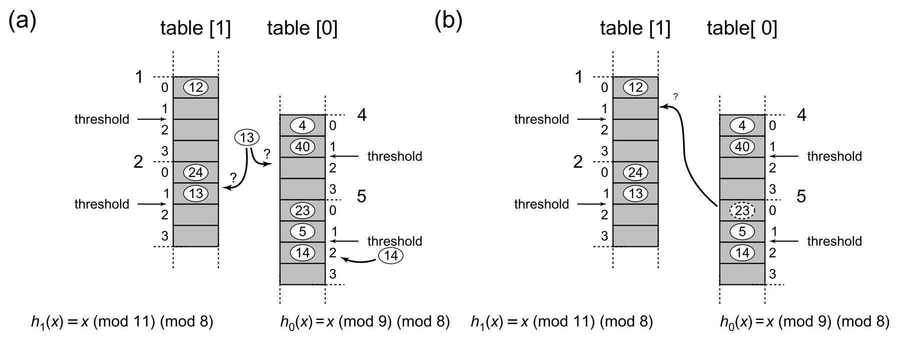
(a)
table [1]
table [0]
threshold
threshold
threshold
threshold
$h_1(x) = x \pmod{11} \pmod{8}$
$h_0(x) = x \pmod{9} \pmod{8}$
(b)
table [1]
table[0]
threshold
threshold
threshold
threshold
$h_1(x) = x \pmod{11} \pmod{8}$
$h_0(x) = x \pmod{9} \pmod{8}$

FIGURE 13.23

The `PhasedCuckooHashSet<T>` class: `add()` and `relocate()` methods. The figure shows the array segments consisting of eight probe sets of size 4 each, with a threshold of 2. Shown are probe sets 4 and 5 of Table[0][] and 1 and 2 of Table[1][]. In part (a), an item with key 13 finds Table[0][4] above threshold and Table[1][2] below threshold, so it adds the item to the probe set Table[1][2]. The item with key 14, on the other hand, finds that both of its probe sets are above threshold, so it adds its item to Table[0][5] and signals that the item should be relocated. In part (b), the method tries to relocate the item with key 23, the oldest item in Table[0][5]. Since Table[1][1] is below threshold, the item is successfully relocated. If Table[1][1] were above threshold, the algorithm would attempt to relocate item 12 from Table[1][1], and if Table[1][1] were at the probe set's size limit of four items, it would attempt to relocate the item with key 5, the next oldest item, from Table[0][5].

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

FIGURE 13.24

`PhasedCuckooHashSet<T>` class: fields and constructor.

the `PhasedCuckooHashSet` structure appears in Fig. 13.23, where the `PROBE_SIZE` is 4 and the `THRESHOLD` is 2. While method calls are in-flight, a probe set may temporarily hold more than `THRESHOLD` but never more than `PROBE_SIZE` items. (In our examples, it is convenient to implement each probe set as a fixed-size `List<T>`.) Fig. 13.24 shows the `PhasedCuckooHashSet<T>`'s fields and constructor.

```
public boolean remove(T x) {
  acquire(x);
  try {
    List<T> set0 = table[0][hash0(x) % capacity];
    if (set0.contains(x)) {
      set0.remove(x);
      return true;
    } else {
      List<T> set1 = table[1][hash1(x) % capacity];
      if (set1.contains(x)) {
        set1.remove(x);
        return true;
      }
    }
    return false;
  } finally {
    release(x);
  }
}
```

FIGURE 13.25

`PhasedCuckooHashSet<T>` class: the `remove()` method.

To postpone our discussion of synchronization, the `PhasedCuckooHashSet<T>` class is defined to be *abstract*: it does not implement all its methods. It has the same abstract methods as the `BaseHashSet<T>` class: The `acquire`(x) method acquires all the locks necessary to manipulate item x, `release`(x) releases them, and `resize`() resizes the set. (As before, we require `acquire`(x) to be reentrant.)

From a bird's-eye view, the `PhasedCuckooHashSet<T>` works as follows: It adds and removes items by first locking the associated probe sets in both tables. To remove an item, it proceeds as in the sequential algorithm, checking if it is in one of the probe sets and, if so, removing it. To add an item, it attempts to add it to one of the probe sets. An item's probe sets serve as temporary overflow buffers for long sequences of consecutive displacements that might occur when adding an item to the table. The `THRESHOLD` value is essentially the size of the probe sets in a sequential algorithm. If a probe set already has this many items, the item is added anyway to one of the `PROBE_SIZE–THRESHOLD` overflow slots. The algorithm then tries to relocate another item from the probe set. There are various policies one can use to choose which item to relocate. Here, we move the oldest items out first, until the probe set is below threshold. As in the sequential cuckoo hashing algorithm, one relocation may trigger another, and so on.

Fig. 13.25 shows the `PhasedCuckooHashSet<T>` class's `remove`(x) method. It calls the abstract `acquire`(x) method to acquire the necessary locks and then enters a **`try`** block whose **`finally`** block calls `release`(x). In the **`try`** block, the method simply checks whether x is present in Table[0][$h_0(x)$] or Table[1][$h_1(x)$]. If so, it removes x and returns *true*; otherwise, it returns *false*. The `contains`(x) method works similarly.

```
34  public boolean add(T x) {
35    T y = null;
36    acquire(x);
37    int h0 = hash0(x) % capacity, h1 = hash1(x) % capacity;
38    int i = -1, h = -1;
39    boolean mustResize = false;
40    try {
41      if (present(x)) return false;
42      List<T> set0 = table[0][h0];
43      List<T> set1 = table[1][h1];
44      if (set0.size() < THRESHOLD) {
45        set0.add(x); return true;
46      } else if (set1.size() < THRESHOLD) {
47        set1.add(x); return true;
48      } else if (set0.size() < PROBE_SIZE) {
49        set0.add(x); i = 0; h = h0;
50      } else if (set1.size() < PROBE_SIZE) {
51        set1.add(x); i = 1; h = h1;
52      } else {
53        mustResize = true;
54      }
55    } finally {
56      release(x);
57    }
58    if (mustResize) {
59      resize(); add(x);
60    } else if (!relocate(i, h)) {
61      resize();
62    }
63    return true; // x must have been present
64  }
```

FIGURE 13.26

PhasedCuckooHashSet<T> class: the add() method.

Fig. 13.26 illustrates the add(x) method. Like remove(), it calls acquire(x) to acquire the necessary locks and then enters a **try** block whose **finally** block calls release(x). It returns *false* if the item is already present (line 41). If either of the item's probe sets is below threshold (lines 44 and 46), it adds the item and returns. Otherwise, if either of the item's probe sets is above threshold but not full (lines 48 and 50), it adds the item and makes a note to rebalance the probe set later. Finally, if both sets are full, it makes a note to resize the entire set (line 53). It then releases the lock on x (line 56).

If the method was unable to add x because both its probe sets were full, it resizes the hash set and tries again (line 58). If the probe set at row r and column c was above threshold, it calls relocate(r, c) (described later) to rebalance probe set sizes.

```
  protected boolean relocate(int i, int hi) {
    int hj = 0;
    int j = 1 - i;
    for (int round = 0; round < LIMIT; round++) {
      List<T> iSet = table[i][hi];
      T y = iSet.get(0);
      switch (i) {
      case 0: hj = hash1(y) % capacity; break;
      case 1: hj = hash0(y) % capacity; break;
      }
      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;
          }
        } else if (iSet.size() >= THRESHOLD) {
          continue;
        } else {
          return true;
        }
      } finally {
        release(y);
      }
    }
    return false;
  }
```

FIGURE 13.27

`PhasedCuckooHashSet<T>` class: the `relocate()` method.

If the call returns *false*, indicating that it failed to rebalance the probe sets, then `add()` resizes the table.

The `relocate()` method appears in Fig. 13.27. It takes the row and column coordinates of a probe set observed to have more than `THRESHOLD` items, and tries to reduce its size below threshold by moving items from this probe set to alternative probe sets.

This method makes a fixed number (`LIMIT`) of attempts before giving up. Each time around the loop, the following invariants hold: `iSet` is the probe set we are trying to shrink, y is the oldest item in `iSet`, and `jSet` is the other probe set where y could be. The loop identifies y (line 70), locks both probe sets to which y could belong (line 75), and tries to remove y from the probe set (line 78). If it succeeds (another thread could have removed y between lines 70 and 78), then it prepares to add y to `jSet`. If `jSet` is below threshold (line 79), then the method adds y to `jSet` and returns *true* (no need to resize). If `jSet` is above threshold but not full (line 82), then it tries to shrink `jSet` by swapping `iSet` and `jSet` (lines 82–86) and resuming the loop. If `jSet` is full (line 87), the method puts y back in `iSet` and returns *false* (triggering a resize). Otherwise it tries to shrink `jSet` by swapping `iSet` and `jSet` (lines 82–86). If the method does not succeed in removing y at line 78, then it rechecks the size of `iSet`. If it is still above threshold (line 91), then the method resumes the loop and tries again to remove an item. Otherwise, `iSet` is below threshold, and the method returns *true* (no resize needed). Fig. 13.23 shows an example execution of the `PhasedCuckooHashSet<T>`, where the item with key 14 causes a relocation of the oldest item 23 from the probe set `table`[0][5].

13.4.3 Striped concurrent cuckoo hashing

We first consider a concurrent cuckoo hash set implementation using lock striping (Section 13.2.2). The `StripedCuckooHashSet` class extends `PhasedCuckooHashSet`, providing a fixed 2-by-L array of reentrant locks. As usual, `lock`$[i][j]$ protects `table`$[i][k]$, where $k \pmod{L} = j$. Fig. 13.28 shows the `StripedCuckooHashSet` class's fields and constructor. The constructor calls the `PhasedCuckooHashSet<T>` constructor (line 4) and then initializes the lock array.

The `StripedCuckooHashSet` class's `acquire`(x) and `release`(x) methods (Fig. 13.29) lock and unlock `lock`$[0][h_0(x)]$ and `lock`$[1][h_1(x)]$ (in that order, to avoid deadlock).

```
 1  public class StripedCuckooHashSet<T> extends PhasedCuckooHashSet<T>{
 2    final ReentrantLock[][] lock;
 3    public StripedCuckooHashSet(int capacity) {
 4      super(capacity);
 5      lock = new ReentrantLock[2][capacity];
 6      for (int i = 0; i < 2; i++) {
 7        for (int j = 0; j < capacity; j++) {
 8          lock[i][j] = new ReentrantLock();
 9        }
10      }
11    }
12    ...
13  }
```

FIGURE 13.28

`StripedCuckooHashSet` class: fields and constructor.

```
  public final void acquire(T x) {
    lock[0][hash0(x) % lock[0].length].lock();
    lock[1][hash1(x) % lock[1].length].lock();
  }
  public final void release(T x) {
    lock[0][hash0(x) % lock[0].length].unlock();
    lock[1][hash1(x) % lock[1].length].unlock();
  }
```

FIGURE 13.29

`StripedCuckooHashSet` class: `acquire()` and `release()`.

```
  public void resize() {
    int oldCapacity = capacity;
    for (Lock aLock : lock[0]) {
      aLock.lock();
    }
    try {
      if (capacity != oldCapacity) {
        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 aLock : lock[0]) {
        aLock.unlock();
      }
    }
  }
```

FIGURE 13.30

`StripedCuckooHashSet` class: the `resize()` method.

The only difference between the `resize()` methods of `StripedCuckooHashSet` (Fig. 13.30) and `StripedHashSet` is that the latter acquires the locks in `lock[0]` in ascending order (line 24). Acquiring these locks in this order ensures that no other thread is in the middle of an `add()`, `remove()`, or `contains()` call, and avoids deadlocks with other concurrent `resize()` calls.

13.4.4 A refinable concurrent cuckoo hash set

This section introduces the `RefinableCuckooHashSet` class (Fig. 13.31), using the methods of Section 13.2.3 to resize the lock arrays. Just as for the `RefinableHashSet` class, we introduce an `owner` field of type `AtomicMarkableReference<Thread>` that combines a Boolean value with a reference to a thread. If the Boolean value is *true*, the set is resizing, and the reference indicates which thread is in charge of resizing.

Each phase locks the buckets for x by calling `acquire`(x), shown in Fig. 13.32. It reads the lock array (line 24), and then spins until no other thread is resizing the set (lines 21–23). It then acquires the item's two locks (lines 27 and 28), and checks if the lock array is unchanged (line 30). If the lock array has not changed between lines 24 and 30, then the thread has acquired the locks it needs to proceed. Otherwise, the locks it has acquired are out of date, so it releases them and starts over. The `release`(x) method, also shown in Fig. 13.32, releases the locks acquired by `acquire`(x).

The `resize()` method (Fig. 13.33) is almost identical to the `resize()` method for `StripedCuckooHashSet`. One difference is that the `locks[]` array has two dimensions.

The `quiesce()` method (Fig. 13.34), like its counterpart in the `RefinableHashSet` class, visits each lock and waits until it is unlocked. The only difference is that it visits only the locks in `locks[0]`.

```java
public class RefinableCuckooHashSet<T> extends PhasedCuckooHashSet<T>{
  AtomicMarkableReference<Thread> owner;
  volatile ReentrantLock[][] locks;
  public RefinableCuckooHashSet(int capacity) {
    super(capacity);
    locks = new ReentrantLock[2][capacity];
    for (int i = 0; i < 2; i++) {
      for (int j = 0; j < capacity; j++) {
        locks[i][j] = new ReentrantLock();
      }
    }
    owner = new AtomicMarkableReference<Thread>(null, false);
  }
  ...
}
```

FIGURE 13.31

`RefinableCuckooHashSet<T>`: fields and constructor.

```
  public void acquire(T x) {
    boolean[] mark = {true};
    Thread me = Thread.currentThread();
    Thread who;
    while (true) {
      do { // wait until not resizing
        who = owner.get(mark);
      } while (mark[0] && who != me);
      ReentrantLock[][] oldLocks = locks;
      ReentrantLock oldLock0 = oldLocks[0][hash0(x) % oldLocks[0].length];
      ReentrantLock oldLock1 = oldLocks[1][hash1(x) % oldLocks[1].length];
      oldLock0.lock();
      oldLock1.lock();
      who = owner.get(mark);
      if ((!mark[0] || who == me) && locks == oldLocks) {
        return;
      } else {
        oldLock0.unlock();
        oldLock1.unlock();
      }
    }
  }
  public void release(T x) {
    locks[0][hash0(x)].unlock();
    locks[1][hash1(x)].unlock();
  }
```

FIGURE 13.32

`RefinableCuckooHashSet<T>`: `acquire()` and `release()` methods.

13.5 Chapter notes

The term *disjoint-access-parallelism* was coined by Amos Israeli and Lihu Rappoport [84]. Maged Michael [126] has shown that simple algorithms using a reader–writer lock [124] per bucket have reasonable performance without resizing. The lock-free hash set based on split-ordering described in Section 13.3.1 is by Ori Shalev and Nir Shavit [156]. The optimistic and fine-grained hash sets are adapted from a hash set implementation by Doug Lea [108], used in java.util.concurrent.

Other concurrent closed-addressing schemes include ones by Meichun Hsu and Wei-Pang Yang [79], Vijay Kumar [97], Carla Schlatter Ellis [43], and Michael Greenwald [54]. Hui Gao, Jan Friso Groote, and Wim Hesselink [50] proposed an almost wait-free extensible open-addressing hashing algorithm, and Chris Purcell and Tim Harris [143] proposed a concurrent nonblocking hash table with open addressing. Cuckoo hashing is credited to Rasmus Pagh and Flemming Rodler [136], and the concurrent version is by Maurice Herlihy, Nir Shavit, and Moran Tzafrir [73].

```
public void resize() {
  int oldCapacity = capacity;
  Thread me = Thread.currentThread();
  if (owner.compareAndSet(null, me, false, true)) {
    try {
      if (capacity != oldCapacity) { // someone else resized first
        return;
      }
      quiesce();
      capacity = 2 * capacity;
      List<T>[][] oldTable = table;
      table = (List<T>[][]) new List[2][capacity];
      locks = new ReentrantLock[2][capacity];
      for (int i = 0; i < 2; i++) {
        for (int j = 0; j < capacity; j++) {
          locks[i][j] = new ReentrantLock();
        }
      }
      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 {
      owner.set(null, false);
    }
  }
}
```

FIGURE 13.33

RefinableCuckooHashSet<T>: the resize() method.

```
protected void quiesce() {
  for (ReentrantLock lock : locks[0]) {
    while (lock.isLocked()) {}
  }
}
```

FIGURE 13.34

RefinableCuckooHashSet<T>: the quiesce() method.

```
public class UnboundedResizeLockFreeHashSet<T> {
   public UnboundedResizeLockFreeHashSet(int initialMinimumNumBuckets) { ... }
   private BucketList<T> getBucketList(int hashCode) { ... }
   private void resize() { ... }
   public boolean add(T x) { ... }
   public boolean remove(T x) { ... }
   public boolean contains(T x) { ... }
}
```

FIGURE 13.35

The `UnboundedResizeLockFreeHashSet` class.

13.6 Exercises

Exercise 13.1. Modify the `StripedHashSet` to allow resizing of the range lock array using read–write locks.

Exercise 13.2. For the `LockFreeHashSet`, show an example of the problem that arises when deleting an entry pointed to by a bucket reference, if we do not add a *sentinel* entry, which is never deleted, to the start of each bucket.

Exercise 13.3. For the `LockFreeHashSet`, when an uninitialized bucket is accessed in a table of size N, it might be necessary to recursively initialize (i.e., split) as many as $O(\log N)$ of its parent buckets to allow the insertion of a new bucket. Show an example of such a scenario. Explain why the expected length of any such recursive sequence of splits is constant.

Exercise 13.4. For the `LockFreeHashSet`, design a lock-free data structure to replace the fixed-size bucket array. Your data structure should allow an arbitrary number of buckets.

Exercise 13.5. For the `LockFreeHashSet`, design a lock-free data structure to replace the fixed-size bucket array. Your data structure should allow for unbounded doubling of the number of buckets in order to keep the average bucket length below THRESHOLD. Describe how you would implement the methods in Fig. 13.35 and how your implementation preserves lock-freedom, correctness, and expected or amortized $O(1)$ work.

Exercise 13.6. Outline correctness arguments for `LockFreeHashSet`'s `add()`, `remove()`, and `contains()` methods.

Hint: You may assume the `LockFreeList` algorithm's methods are correct.