# 13 Concurrent Hashing and Natural Parallelism

## 13.1 Introduction

In earlier chapters, we studied how to extract parallelism from data structures like queues, stacks, and counters, that 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.

*Hashing* is a technique commonly used in sequential `Set` implementations to ensure that `contains()`, `add()`, and `remove()` calls take constant average time. The concurrent `Set` implementations studied in Chapter 9 required time linear in the size of the set. In this chapter, we study ways to make hashing concurrent, sometimes using locks and sometimes not. Even though hashing seems naturally parallelizable, devising an effective concurrent hash algorithm is far from trivial.

As in earlier chapters, the `Set` interface provides the following methods, which return Boolean values:

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

When designing set implementations, we need to keep the following principle in mind: we can buy more memory, but we cannot buy more time. Given a choice between an algorithm that runs faster but 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 one 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.)

In some hash-based set algorithms, each table entry refers to a single item, an approach known as *open addressing*. In others, each table entry refers to a set of items, traditionally called a *bucket*, an approach known as *closed addressing*.

Any hash set algorithm must deal with *collisions*: what to do when two distinct items hash to the same table entry. Open-addressing algorithms typically resolve collisions by applying alternative hash functions to test alternative table entries. Closed-addressing algorithms place colliding items in the same bucket, until that bucket becomes too full. In both kinds of algorithms, it is sometimes necessary to *resize* the table. In open-addressing algorithms, the table may become too full to find alternative table entries, and in closed-addressing algorithms, buckets may become too large to search efficiently.

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).

It is easier to make closed-addressing hash set algorithms parallel, so we consider them first.

## 13.2 Closed-Address Hash Sets

*Pragma* 13.2.1. Here and elsewhere, we use the standard Java `List<T>` interface (in package **java.util.List** ). A `List<T>` is an ordered collection of `T` objects, where `T` is a type. Here, we make use of the following List methods: `add`($x$) appends $x$ to the end of the list, `get`($i$) returns (but does not remove) the item at position $i$, `contains`($x$) returns *true* if the list contains $x$. There are many more.

The `List` interface can be implemented by a number of classes. Here, it is convenient to use the `ArrayList` class.

We start by defining a *base* hash set implementation common to all the concurrent closed-addressing hash sets we consider here. The `BaseHashSet<T>` class is an *abstract* class, that is, it does not implement all its methods. Later, we look at three alternative synchronization techniques: one using a single coarse-grained lock, one using a fixed-size array of locks, and one using a resizable array of locks.

```
 1  public abstract class BaseHashSet<T> {
 2    protected List<T>[] table;
 3    protected int setSize;
 4    public BaseHashSet(int capacity) {
 5      setSize = 0;
 6      table = (List<T>[]) new List[capacity];
 7      for (int i = 0; i < capacity; i++) {
 8        table[i] = new ArrayList<T>();
 9      }
10    }
11    ...
12  }
```

**Figure 13.1** `BaseHashSet<T>` class: fields and constructor.

Fig. 13.1 shows the base hash set's fields and constructor. The `table[]` field is an array of buckets, each of which is a set implemented as a list (Line 2). We use `ArrayList<T>` lists for convenience, supporting the standard sequential `add()`, `remove()`, and `contains()` methods. The `setSize` field is the number of items in the table (Line 3). We sometimes refer to the length of the `table[]` array, that is, the number of buckets in it, as its *capacity*.

The `BaseHashSet<T>` class does not implement the following *abstract* methods: `acquire`($x$) acquires the locks necessary to manipulate item $x$, `release`($x$) releases them, `policy`() decides whether to resize the set, and `resize`() doubles the capacity of the `table[]` array. The `acquire`($x$) method must be *reentrant* (Chapter 8, Section 8.4), 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, 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 17), while `add`($x$) adds $x$ to the list if it is not already present (Line 27).

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 $x$, takes constant time. The second step, adding the item to the bucket, requires traversing a linked 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 table. This number may vary unpredictably over time, so to ensure that method call times remain (more-or-less) constant, we must *resize* the table every now and then to ensure that list lengths remain (more-or-less) constant.

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

```
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;
    result = table[myBucket].add(x);
    setSize = result ? setSize + 1 : setSize;
  } 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.

- If more than, say, 1/4 of the buckets exceed the bucket threshold, then double the table capacity, or
- If any single bucket exceeds the global threshold, then double the table capacity.

Both these strategies work well in practice, as do others. Open-addressing algorithms are slightly more complicated, and are discussed later.

### 13.2.1 A Coarse-Grained Hash Set

Fig. 13.3 shows the `CoarseHashSet<T>` class's fields, constructor, `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 20), and checks that no other thread has resized the table in the meantime (Line 23). It then allocates and initializes a new table with double the capacity (Lines 25–29) and transfers items from the old to the new buckets (Lines 30–34). Finally, it unlocks the set (Line 36).

```
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, `acquire()`, and `release()` methods.

```
  public boolean policy() {
    return setSize / table.length > 4;
  }
  public void resize() {
    int oldCapacity = table.length;
    lock.lock();
    try {
      if (oldCapacity != table.length) {
        return; // someone beat us to it
      }
      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) {
        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.

### 13.2.2 A Striped Hash Set

Like the coarse-grained list studied in Chapter 9, the coarse-grained hash set shown in the last section 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 there is no logical reason for them to do so.

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 capacity, `table[]` will grow when the set is resized, but `lock[]` will not. Every now and then, we double the table capacity $N$ without changing the lock array size $L$, so that lock $i$ eventually 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 every time 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 `resize`() acquires the locks in `lock[]` in ascending order (Lines 18–20). 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 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 the `CoarseHashSet<T>`, when a thread starts to resize the table, it records the current table capacity. If, after it has acquired all the locks, it discovers that some other thread has changed the table

```
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, `acquire()`, and `release()` methods.

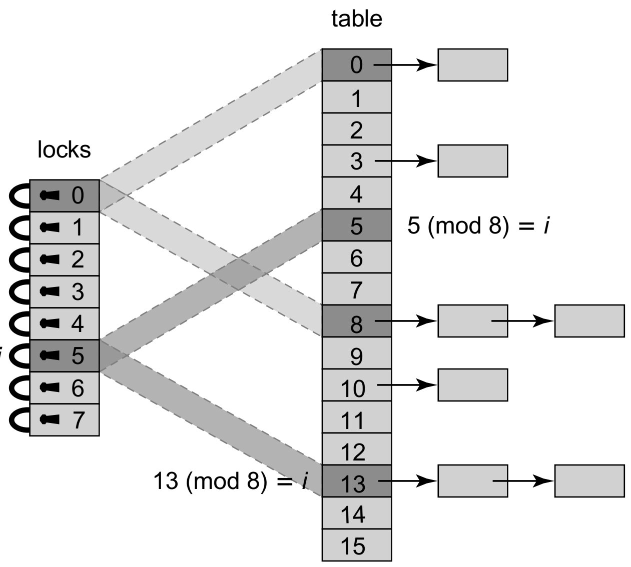

**Figure 13.6** Resizing a `StripedHashSet` lock-based hash table. As the table grows, the striping is adjusted to ensure that each lock covers $2^{N/L}$ entries. In the figure above, $N = 16$ and $L = 8$. When $N$ is doubled from 8 to 16, the memory is striped so that lock $i = 5$ for example covers both locations that are equal to 5 modulo $L$.

capacity (Line 23), 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 25), and transfer items from the old table to the new (Line 30). Finally, it releases the locks (Line 36). Because the `initializeFrom()` method calls `add()`, it may trigger nested calls to `resize()`. We leave it as an exercise to check that nested resizing works correctly in this and later hash set implementations.

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() {
    int oldCapacity = table.length;
    for (Lock lock : locks) {
      lock.lock();
    }
    try {
      if (oldCapacity != table.length) {
        return; // someone beat us to it
      }
      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) {
        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, then check that no other thread has resized the table in the meantime.

```
public class RefinableHashSet<T> extends BaseHashSet<T>{
  AtomicMarkableReference<Thread> owner;
  volatile ReentrantLock[] locks;
  public RefinableHashSet(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);
  }
  ...
}
```

Figure 13.8 `RefinableHashSet<T>` class: fields and constructor.

Fig. 13.8 shows the fields and constructor for the `RefinableHashSet<T>` class. To add a higher level of synchronization, we introduce a globally shared `owner` field that combines a Boolean value with a reference to a thread. 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. These two values are combined in an `AtomicMarkableReference<Thread>` to allow them to be modified atomically (see Pragma 9.8.1 in Chapter 9). We use the `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. 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 the locks (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 locks it has acquired could be out-of-date because of an ongoing update, so it releases them and starts over. 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 locks acquired by `acquire`(*x*).

The `resize()` method is almost identical to the `resize()` method for the `StripedHashSet` class. The one difference appears on Line 46: instead of acquiring all the locks in `lock[]`, the method calls `quiesce()` (Fig. 13.10) to ensure that no other thread is in the middle of an `add()`, `remove()`, or `contains()` call. The `quiesce()` method visits each lock and waits until it is unlocked.

```
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.

```
public void resize() {
  int oldCapacity = table.length;
  boolean[] mark = {false};
  int newCapacity = 2 * oldCapacity;
  Thread me = Thread.currentThread();
  if (owner.compareAndSet(null, me, false, true)) {
    try {
      if (table.length != oldCapacity) { // someone else resized first
        return;
      }
      quiesce();
      List<T>[] oldTable = table;
      table = (List<T>[]) new List[newCapacity];
      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();
      }
      initializeFrom(oldTable);
    } finally {
      owner.set(null, false);
    }
  }
}
```

Figure 13.10 `RefinableHashSet<T>` class: `resize()` method.

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

Figure 13.11 `RefinableHashSet<T>` class: `quiesce()` method.

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 the locks after `quiesce()` has completed will see that the set is in the processes of being resized, and will back off until the resizing is complete. Similarly, `resize()` will first set the `mark` field, then read the locks, and will not proceed while any `add()`, `remove()`, or `contains()` call's lock is set.

To summarize, we have seen that one can design a hash table in which both the number of buckets and the number of locks can be continuously 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, because 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.

Part (b) of 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 short-cuts 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.

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

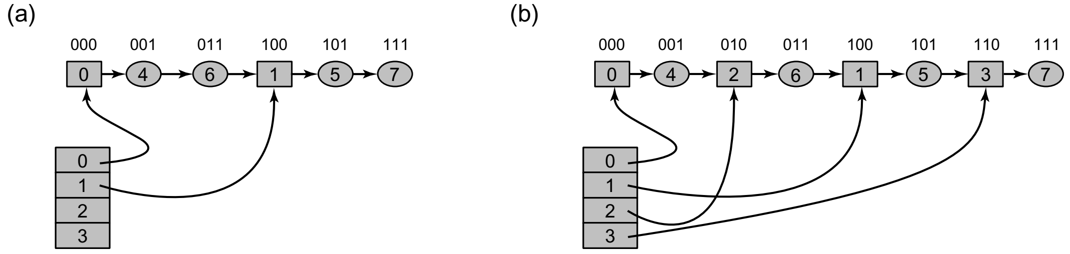

**Figure 13.12** This figure explains the recursive nature of the split ordering. Part (a) shows a split-ordered list consisting of two buckets. The array of buckets refer into a single linked list. The split-ordered keys (above each node) are the reverse of the bitwise representation of the items' keys. 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 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 2 buckets to four. The reverse bit values of the two added Buckets 2 and 3 happen to perfectly split the Buckets 0 and 1.

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 $i$ least significant bits (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 $i^{th}$ 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 least significant bit. 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 $i^{th}$ 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, that is, 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 set grows. When the `add()` method accesses an uninitialized bucket that should have been initialized given the current table capacity, it initializes it. While conceptually simple, this design is far from ideal,

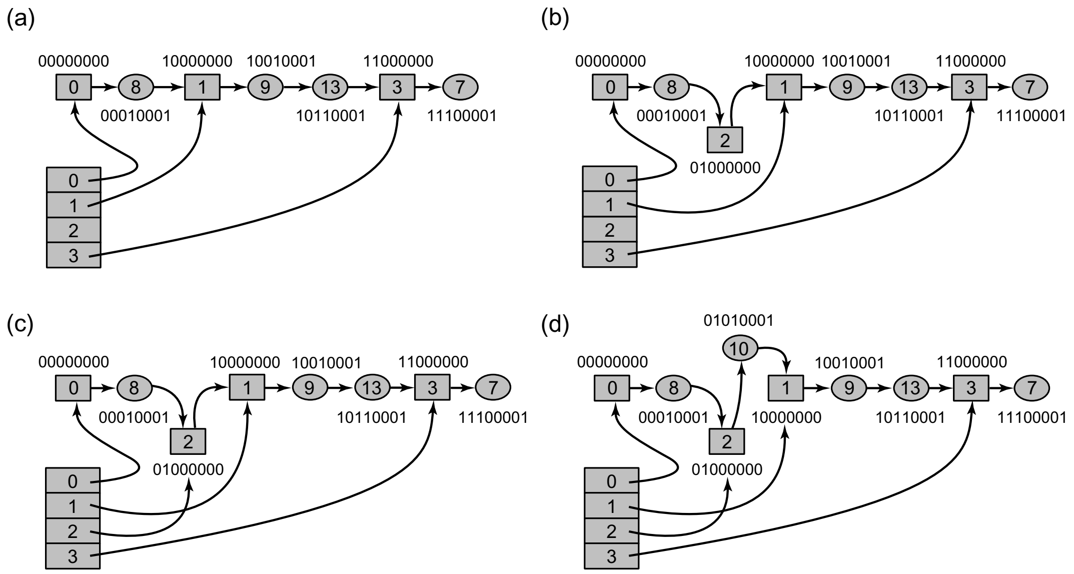

Figure 13.13 How the `add()` method places key 10 to the lock-free table. As in earlier figures, the split-order key values, expressed as 8-bit binary words, appear above the nodes. For example, the split-order value of 1 is the bit-wise reversal of its binary representation. In Step (a) Buckets 0, 1, 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.

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, 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, 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 the $x$'s node, if it 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

```
public class BucketList<T> implements Set<T> {
  static final int HI_MASK = 0x00800000;
  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 pred = window.pred;
    Node curr = window.curr;
    return (curr.key == key);
  }
```

**Figure 13.15** `BucketList<T>` class: the `contains()` method.

at the start of each 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 `LockFreeHashSet<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, used to decide when to resize.

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`($x$)

```
26    public BucketList<T> getSentinel(int index) {
27      int key = makeSentinelKey(index);
28      boolean splice;
29      while (true) {
30        Window window = find(head, key);
31        Node pred = window.pred;
32        Node curr = window.curr;
33        if (curr.key == key) {
34          return new BucketList<T>(curr);
35        } else {
36          Node node = new Node(key);
37          node.next.set(pred.next.getReference(), false);
38          splice = pred.next.compareAndSet(curr, node, false, false);
39          if (splice)
40            return new BucketList<T>(node);
41          else
42            continue;
43        }
44      }
45    }
```

Figure 13.16 BucketList<T> class: getSentinel() method.

```
 1  public class LockFreeHashSet<T> {
 2    protected BucketList<T>[] bucket;
 3    protected AtomicInteger bucketSize;
 4    protected AtomicInteger setSize;
 5    public LockFreeHashSet(int capacity) {
 6      bucket = (BucketList<T>[]) new BucketList[capacity];
 7      bucket[0] = new BucketList<T>();
 8      bucketSize = new AtomicInteger(2);
 9      setSize = new AtomicInteger(0);
10    }
11    ...
12  }
```

Figure 13.17 LockFreeHashSet<T> class: fields and constructor.

```
13    public boolean add(T x) {
14      int myBucket = BucketList.hashCode(x) % bucketSize.get();
15      BucketList<T> b = getBucketList(myBucket);
16      if (!b.add(x))
17        return false;
18      int setSizeNow = setSize.getAndIncrement();
19      int bucketSizeNow = bucketSize.get();
20      if (setSizeNow / bucketSizeNow > THRESHOLD)
21        bucketSize.compareAndSet(bucketSizeNow, 2 * bucketSizeNow);
22      return true;
23    }
```

Figure 13.18 LockFreeHashSet<T> class: add() method.

```
24    private BucketList<T> getBucketList(int myBucket) {
25      if (bucket[myBucket] == null)
26        initializeBucket(myBucket);
27      return bucket[myBucket];
28    }
29    private void initializeBucket(int myBucket) {
30      int parent = getParent(myBucket);
31      if (bucket[parent] == null)
32        initializeBucket(parent);
33      BucketList<T> b = bucket[parent].getSentinel(myBucket);
34      if (b != null)
35        bucket[myBucket] = b;
36    }
37    private int getParent(int myBucket){
38      int parent = bucketSize.get();
39      do {
40        parent = parent >> 1;
41      } while (parent > myBucket);
42      parent = myBucket - parent;
43      return parent;
44    }
```

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.

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.

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 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 total 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.

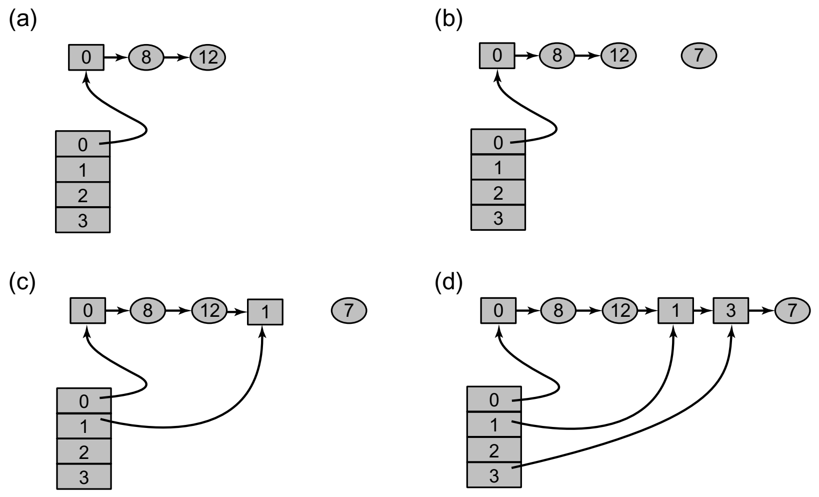

Figure 13.20 Recursive initialization of lock-free hash table buckets. (a) Table has four buckets; only bucket 0 is initialized. (b) We wish to insert the item with key 7. Bucket 3 now requires initialization, which in turn requires recursive initialization of Bucket 1. (c) Bucket 1 is initialized by 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.

## 13.4 An Open-Addressed Hash Set

We now turn our attention to a concurrent open hashing algorithm. Open hashing, in which each table entry holds a single item rather than a set, seems harder to make concurrent than closed 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.[1] 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,[2] and two independent hash functions,

$$h_0, h_1 : KeyRange \to 0, \ldots, k-1.$$

1 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.

2 This division of the table into two arrays will help 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 $2k$.