The Art of Multiprocessor Programming

Maurice Herlihy Nir Shavit

July 17, 2006

Contents

Chapter 11

Concurrent Hashing

11.1 Introduction

In Chapter 10, we studied concurrent implementations of the Set interface. These implementations share a substantial disadvantage: the time needed to find (or add, or remove) an item is linear in the size of the set. *Hashing* is a technique commonly used in sequential Set implementations to ensure that these method calls take constant time on the average. In this chapter, we study ways to adapt hashing to a concurrent environment, both using locks and lock-free.

When designing set implementations, keep the following principle in mind: within reason, space is cheaper than time. (Your money can buy more memory, but it can't buy more time.) More precisely, every year, one more Dollar, Pound, or Euro can buy either a much larger memory or a slightly faster processor.

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

In some hash set algorithms, each table element refers to a single item,

an approach known as *open addressing*. In others, each table element 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 elements. 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 growing hash sets (leaving the problem of shrinking them to the exercises).

Concurrent hash set algorithms that use closed addressing are easier to parallelize, so we will consider them first.

11.2 A Coarse-Grained Hash Set

Figure 11.1 shows a coarse-grained, closed-addressing hash set in which all methods synchronize via a single lock. The add(x) method computes item x's hash value, and uses that value (modulo table size) to choose a bucket. The bucket itself is implemented by a ListBucket class, which is simply an unsynchronized linked list. The hash set's add(x) method calls the bucket's add(x) method, and returns the result. The contains() and remove() methods work in a similar way.

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 size 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 sizes remain (more-or-less) constant. Figure 11.2 shows how to resize a coarse-grained hash set.

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*.

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

Both these strategies work well in practice, as do others. Open-addressing algorithms are slightly more complicated, and are discussed later. In our examples we will call a policy() method that indicates when to resize.

11.3 Fine-Grained Locking

Like the coarse-grained list studied in Chapter 10, the coarse-grained hash table shown in Figures 11.1 and 11.2 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.

First, we consider *fine-grained locks*. Instead of using a single lock to synchronize the entire set, we split the set into independently-synchronized pieces. We introduce a useful technique called *lock striping*, which will be useful for other data structures as well. We start with an array of L locks and a table of L entries, where lock i protects table entry i. Every now and then, we double the table size without changing the lock array size, so that lock i protects each table entry j, where $j \pmod L = i$. 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) can be tricky.

We leave the problem of growing the lock array as an exercise.

As shown in Figure 11.3, contains(x) hashes x to construct two indexes. Let L be the lock array size, and N (a multiple of L) be the table size.

The hash value modulo L determines which lock to use, and the hash value modulo N determines which bucket to use. The thread locks the bucket, and tests whether the item is present. The add(x) and remove(x) methods are similar.

After releasing its lock but before returning, each add() call checks the resizing policy method to determine whether to resize. Clearly, resizing the table affects all the table's buckets, so resize() must acquire all the locks before it can proceed. As shown in Figure 11.4, it acquires the locks in ascending order. A resize() call 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 happens if two or more concurrent add() calls both decide to resize at the same time? When a thread starts to resize the table, it records the current table size. If, after it has acquired all the locks, it discovers that some other thread has changed the table size, then it releases the locks and gives up. (It could just double the table size anyway, since it already holds all the locks.)

In summary, a hash set using fine-grained locking provides more concurrency than the same data structure using a single, coarse-grained lock, because method calls whose items hash to different locks can proceed in parallel.

11.4 Lock-Free Hashing

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 this move is not done atomically, entries might be temporarily lost or duplicated, compromising correctness.

In the absence of locks, our only means of synchronization are methods such as compareAndSet(), which operate on a single memory location. It is not easy to see how a single compareAndSet() call can atomically move an entry from one linked list to another. One might consider "marking" an entry as logically moved, in the same way we marked an entry as logically removed, but we do not know how to ensure that a thread searching the new bucket would find the marked entry.

To implement a concurrent hash set, we will have to figure out how to move items atomically from old buckets to new. To do so, we will, metaphorically speaking, flip the conventional hashing algorithm on its head:

Instead of moving the items among the buckets, we will move

the buckets among the items.

More specifically, we keep all entries in a single lock-free linked list similar to the LockFreeList class studied in Chapter 10. Buckets are references 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.

The lock-free hash set implementation has two components (see Figure 11.7): a linked list of entries and sentinels, and an expanding array of references into the list. These array entries are the logical "buckets" typical of most hashing structures. Any item in the hash set can be reached by traversing the list from its head, while the bucket pointers provide short-cuts into the list to minimize the number of entries traversed when searching.

The bucket, called LockFreeBucket, is a straightforward extension of the LockFreeList class studied in Chapter 10. There are two differences. First, while the LockFreeBucket class uses only two sentinels, LockFreeBucket requires the ability to insert sentinels at intermediate positions within the list, and to traverse the list starting from such sentinels. The LockFreeBucket class provides add(), remove(), and contains() methods just like LockFreeList, but also provides the following additional method:

public LockFreeBucket getSentinel(**int** index)

This method takes a bucket index, returns a LockFreeList object representing the part of the parent list following the new sentinel.

Second, the LockFreeBucket manipulates key values to distinguish between regular and sentinel entries, and orders regular entries differently. We will discuss these issues in detail later on.

The principal challenge is ensuring that the bucket references remain well-distributed as the number of list entries grows. Bucket references must be spaced evenly enough to allow constant-time access to any entry. It follows that new buckets must be created and assigned to sparsely-covered regions in the list.

The bucket array initially has size 2, and is doubled whenever the number of items in the table exceeds $s \cdot L$, where s is the current table size, and L is a small integer called the *load factor*, the maximum number of items one would expect to find in each logical bucket. Initially, all bucket references are **null**, except for the bucket at index 0, which points to an empty list, and is effectively the `head` reference for the LockFreeBucket list structure. Each entry in the bucket array is initialized when first accessed, and subsequently points to some entry in the list.

When an item with hash code k is inserted, removed, or searched for in the table, the set uses bucket index $k \pmod{s}$, where s, the current table size, is always a power of 2. If the table size is 2^i, then the bucket index is the integer represented by the key's i least significant bits (LSBs). Because the hash function depends on the table size, we must be careful when the table size changes. An item inserted before the table was resized must be accessible afterwards from both its previous and current buckets.

When the table size is 2^i, bucket b contains entries each of whose hash codes k satisfies $k \equiv b \pmod{2^i}$. When the size grows to 2^{i+1}, the entries in this bucket are split between two buckets: those for which $k \equiv b \pmod{2^{i+1}}$ remain in bucket b, while those for which $k \equiv b + 2^i \pmod{2^{i+1}}$ migrate to bucket $b+2^i$. Here is the key idea behind the algorithm: we will ensure that these two groups of entries are positioned one after the other in the list, so that splitting the bucket b is achieved by simply pointing 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.

Entries 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 set doubling will cause each group to split again into two groups differentiated by the $(i+1)^{st}$ bit, and so on. This process induces a total order on items, which we call *recursive split-ordering*. Given a key's hash code, its order is completely defined by its *bit-reversed* value.

To recapitulate: the *split-ordered hash set* is an array of (some possibly uninitialized) bucket references into a linked list of entries, where the entries are sorted by their bit-reversed hash codes. Buckets are initialized when they are accessed for the first time. List operations such as inserting, removing, or searching for an entry are implemented by the LockFreeBucket class based on the LockFreeList class studied in Chapter 10.

To avoid an awkward "corner case" that arises when deleting an entry pointed to by a bucket reference, we add a *sentinel* entry, which is never deleted, to the start of each bucket. Specifically, suppose the table size is 2^{i+1}. The first time bucket $b + 2^i$ is accessed, a sentinel entry is created with key $b + 2^i$. This entry is inserted to 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 i LSBs, but their i^{th} bit is 0. Therefore, the new sentinel entry is positioned in the exact location in the list that separates the entries of the new bucket from the remaining items of bucket b. To

distinguish sentinel items from regular items we set the most significant bit (MSB) of regular items to 1, and leave the sentinel items with 0 at the MSB. Figure 11.5 illustrates two methods: `makeRegularItem()`, which generates a split-ordered key for an object, and `makeSentinelKey()`, which generates a split-ordered key for a bucket index.

Figures 11.7 through 11.10 illustrate how inserting a new key into the set can cause a bucket to be initialized. Here, an object with hash code 10 is inserted when the table size is 4, and buckets 0, 1, and 3 are already initialized.

11.4.1 Growing the Table

Recall that each bucket is a linked list, with entries ordered based on the split-ordered hash values. As mentioned above, the table resizing mechanism is independent of the policy used to decide when to resize. To keep our example concrete, we will implement the following policy. We use a shared counter to track the average bucket load. When the average load crosses a threshold, we double the table size. To avoid some technical distractions, we keep the array of buckets in a large, fixed-size array. We start out using only the first array element, and use progressively more of the array as the set grows. More flexible alternative designs are discussed later.

When an uninitialized bucket is accessed in a table of size s, it might be necessary to recursively initialize (that is, split) as many as $O(\log s)$ of its parent buckets to allow the insertion of a new bucket. Although the total complexity in such a case is logarithmic, not constant, the expected length of any such chain is constant.

11.4.2 Lock-Free Hash Set Implementation

We now examine the details of the lock-free split-ordered set implementation illustrated in Figures 11.11 and 11.12. The set has three mutable fields:

- The bucket is an array of LockFreeHashSet objects, all references into the same list.
- the bucketSize field is an atomic integer that tracks how much of the bucket array is currently in use.
- the setSize field is an atomic integer that tracks how many objects are in the set. It is used to decide when to resize.

Here is how to add an object x with hash code k. The add(x) call retrieves bucket $k \pmod s$, where s is the current table size, initializing it if necessary. It then calls the LockFreeBucket's add(x) method. If x was not already present, it increments the set size, and checks whether to increase the number of active buckets. The LockFreeBucket's add(x) call transforms the object's hash code into a split-ordered regular key by setting the high-order bit and reversing the word. Using that transformed key, add(x) proceeds exactly as in the LockFreeList class's add(x). The contains(x) method and remove(x) method work in much the same way.

The initialBucket () method's role is to initialize the bucket field array entry at a particular index. The new value will be a reference to a new sentinel entry. The sentinel entry 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, the initialBucket () method 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 non-zero bit. The LockFreeBucket's getSentinel () method creates a new sentinel key for the index to be initialized, and tries to add it to the LockFreeBucket's list. It may discover that sentinel is present, if that bucket is being initialized concurrently by another thread. Either way, it returns the remainder of the list starting at the indicated sentinel and stores it in the bucket array.

As noted, we have "cheated" a little by keeping the array of buckets in a fixed-size array. 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, a task we leave as an exercise.

11.5 Cuckoo Hashing

We now turn our attention to a concurrent closed hashing algorithm. Closed hashing, in which each table entry holds a single item rather than a set, seems harder to make concurrent than open hashing. The straightforward approach, in which we lock each table entry, leads to deadlock. Instead, we need a way to ensure that non-interfering entries can be accessed in parallel, without danger of deadlock when entries interfere.

11.5.1 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-element array of items. We use a two-element array table[] of tables, and two independent hash functions,

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

mapping the set of keys, K, to slots 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 table[0][$h_0(x)$] or table[1][$h_1(x)$], and removes it if found.

The add(x) method is the most interesting. It successively "kicks out" conflicting items until every key has a slot. To add x, we swap x with the current occupant of table[0][$h_0(x)$] (line 6). If the prior value was **null**, we are done (line 7). Otherwise, we swap the newly nestless value for the current occupant of table[1][$h_1(x)$] in the same way (line 8), continuing until we find an empty slot.

We might not find an empty slot, either because the table is full, or because the sequence of displacements forms a cycle. We 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, in fact, sequential Cuckoo hashing works well in practice.

11.5.2 Concurrent Cuckoo Hashing

We now consider how to make Cuckoo hashing concurrent. One approach is simply to place a lock on each table entry, and to have methods acquire locks as they go. Because associating a lock with each table entry may take too much space, especially for tables that grow very large, so we use lock striping (as in Section 11.3. We use a 2-by-ℓ array of locks, where ℓ is fixed when the table is created. Initially, the table[][] array has size 2-by-ℓ,

[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.

but as it grows, we double the second dimension, replacing ℓ with 2ℓ, and so on. lock[i][j] protects the entry at table[i][k], where $k = j \bmod \ell$. For brevity, we introduce methods lock0(k) to lock table[0][k], and lock1(k) to lock table[1][k].

To avoid deadlock, we need to establish conventions on the order in which threads acquire locks. Here, we structure each method as a sequence of *phases*, where each phase first locks an element of table[0], and then locks an element of table[1]. If a method call requires multiple phases, then it releases the locks acquired by one phase before starting the next.

Here, the principal challenge is how to implement the add() method, which may repeatedly displace entries. To accommodate these cascading displacements, we add a new field: an array of fixed-size sets, called overflow[], which holds displaced entries in the interval after they have been displaced from the table[] array, but before they have been put back in the table[] array. It is convenient to implement each buffer as an ArrayList<T>.

There are two circumstances under which we resize the table. As before, we resize if there are too many cascaded entries, since (in the best case) too much cascading implies that the table is too full, and (in the worst case) there may be cycle of displacements. This implementation also resizes the table if any overflow buffer becomes full, because filling that buffer means that slot in the array is a "hot-spot", the focus of excessive contention.

Figure 11.14 shows the concurrent Cuckoo hash set's add(x) method. It locks table[0][$h_0(x)$] (line 5) and tests whether that entry is equal to x. If so, it returns false (line 8). Otherwise it does the same for table[1][$h_1(x)$] (lines 11 through 14). It then tests whether x is in the overflow buffer at position $h_1(x)$ (line 15). If x is not present in any of these places, the method checks table[0][$h_0(x)$] and table[1][$h_1(x)$], storing x in the first **null** slot it finds (lines 17 through 22). If no such slot is found, it sets y to the current contents y of table[1][$h_1(x)$]. If the matching overflow buffer has room (line 27), y is moved from table[1][$h_1(x)$] to overflow[$h_1(y)$]. After releasing the locks, the method then calls reAdd(y, 0) to put y back into the table (line 45). If, instead, the overflow buffer is full (line 31), then the method leaves the table unchanged, releases the locks, resizes the table, and tries again (line 41).

The reAdd(y, d) method (Figure 11.15) reinserts an overflowed entry into the table[] array. This method is a "benevolent side-effect" in the sense that it is not needed to ensure the correctness of the table, only its performance. It takes two arguments: y, the entry to be inserted, and d, which counts the recursion depth of the current call. This method acquires both locks for y (lines 2 and 3), and tests whether y is still in overflow[$h_1(y)$] (line 7). If not,

it has been removed by a concurrent remove(x) or resize(), and the method returns. Otherwise, it checks table[0][$h_0(y)$] and table[1][$h_1(y)$], storing y in the first **null** slot it finds (lines 12 and **??**). If both slots are already occupied (line 15), it swaps y with the contents z of table[1][$h_1(y)$], adds z to overflow[$h_1(x)$], releases the locks, and recursively calls reAdd($z, d + 1$). If this depth exceeds a fixed limit, it resizes the table. (Note that overflow[$h_1(x)$] is locked between the time x is removed and y is added, so there must be room in the overflow buffer for y.)

Resizing

There are several ways to resize a concurrent Cuckoo hash set. Here is the simplest way. The resize() method acquires the locks in table[0] in ascending order (). This way, we avoid deadlock with concurrent resize() calls, and ensure that no other thread is in the middle of an add(), remove() or contains() call. Once we acquire the locks, we check to see whether some other thread has already resized the table (line 6). If not, we double the table size (line 11), and install new, empty table[] and overflow[] arrays (lines 12-16). Finally, we transfer the entries from the old table[] and overflow[] arrays to the new ones (lines 17-30). Note that these nested calls to add() may cause nested calls to resize(). We leave it as an exercise to check that nested resizing still works.

11.6 Exercises

Exercise 11.6.1 Modify the FineHashSet to allow resizing of the range lock array. (Hint: use readers/writer locks.)

Solution: This can be done by adding a single Readers/Writers Lock to the implementation. Have all methods acquire a read lock before while running, except for the resize method which must acquire a write lock. All methods can run concurrently except for the resizing, and resizing has exclusive access. After resizing the reference to the locks array must be checked to see if someone else has already resized it, and if not, change the reference.

Exercise 11.6.2 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 11.6.3 Outline correctness arguments for LockFreeHashSet's add(), remove(), and contains() methods. (Hint: you may assume the correctness of the LockFreeList algorithms.)

Exercise 11.6.4 Verify that the Cuckoo Hash Set's resize() implementation works even if a call to add() triggers one or more nested resize() calls

Solution: An ongoing resize() call has exclusive access to the tables, so there is no concurrency. All locks are reentrant, so the thread does not conflict with itself.

We argue by induction on the maximum depth of resize() nesting. The claim is obvious when the depth is 1, so assume the claim for $d-1$, $d > 1$. Before the thread transfers the items, it creates a reference *oldTable* to an array containing the two tables, and does not reference any of the object's fields until the transfer is complete. If a call to add() provokes a nested resize(), then when the nested resize() returns, all of the items already added by the outer resize() to the tables are present in object's current fields

11.7 Chapter Notes

Michael [**?**] has shown that simple algorithms using a reader-writer lock [**?**] per bucket have reasonable performance without resizing. The split-ordered hash set described in Section 11.4 is due to Shalev and Shavit [**?**]. The optimistic and fine-grained hash sets are adapted from a hash set implementation by Lea [**?**], used in `java.util.concurrent`.

Other concurrent closed-addressing schemes include Gao, Groote, and Hesselink [**?**], Hsu and Yang [**?**], Kumar [**?**], Ellis [**?**], and Greenwald [**?**].

Cuckoo hashing is due to Pagh and Rodler [**?**]. Purcell and Harris [**?**] propose a concurrent non-blocking hash table with open addressing.

```
public class CoarseHashSet<Item> {
  protected List<Item>[] table;
  protected Lock lock;
  protected int size;
  public CoarseHashSet(int capacity) {
    lock = new ReentrantLock();
    table = (List<Item>[]) new List[capacity];
    for (int i = 0; i < table.length; i++) {
      table[i] = new ArrayList<Item>();
    }
  }
  public boolean contains(Item x) {
    lock.lock();
    try {
      return table[x.hashCode() % table.length].contains(x);
    } finally {
      lock.unlock();
    }
  }
  public boolean add(Item x) {
    lock.lock();
    boolean result;
    try {
      if (policy())
        resize();
      result = table[x.hashCode() % table.length].add(x);
      if (result) {
        size++;
      }
      return result;
    } finally {
      lock.unlock();
    }
  }
  public boolean remove(Item x) {
    lock.lock();
    boolean result;
    try {
      result = table[x.hashCode() % table.length].remove(x);
      if (result) {
        size--;
      }
      return result;
    } finally {
      lock.unlock();
    }
  }
  // policy and resize methods appear elsewhere
}
```

Figure 11.1: *Coarse-grained hash set: each method hashes the item to choose which bucket to use.*

```
protected void resize() {
  int oldCapacity = table.length;
  List<Item>[] oldTable = table;
  int newCapacity = 2 * table.length;
  table = (List<Item>[]) new List[newCapacity];
  for (int i = 0; i < newCapacity; i++) {
    table[i] = new ArrayList();
  }
  for (int i = 0; i < oldCapacity; i++) {
    for (Item x : oldTable[i]) {
      table[x.hashCode() % table.length].add(x);
    }
  }
}
protected boolean policy() {
  return size / table.length > 4;
}
```

Figure 11.2: *Coarse-grained hash set: Resize the table if the number of items in the hash set becomes too large.*

```
public class FineHashSet<Item> {
  protected Lock[] locks;
  protected LockFreeList<Item>[] table;
  protected int size;
  public FineHashSet(int capacity) {
    table = (LockFreeList<Item>[]) new LockFreeList[capacity];
    locks = new Lock[capacity];
    for (int i = 0; i < capacity; i++) {
      locks[i] = new ReentrantLock();
      table[i] = new LockFreeList<Item>();
    }
  }
  public boolean contains(Item x) {
    int myBucket = x.hashCode() % table.length;
    int myLock  = x.hashCode() % locks.length;
    locks[myLock].lock();
    try {
      return table[myBucket].contains(x);
    } finally {
      locks[myLock].unlock();
    }
  }
  public boolean add(Item x) {
    int myBucket = x.hashCode() % table.length;
    int myLock  = x.hashCode() % locks.length;
    boolean result = false;
    locks[myLock].lock();
    try {
      result  = table[myBucket].add(x);
      size  = result ? size + 1 : size;
    } finally {
      locks[myLock].unlock(); // always unlock
    }
    if (policy())
      resize(table.length);
    return result;
  }
  public boolean remove(Item x) {
    int myBucket = x.hashCode() % table.length;
    int myLock  = x.hashCode() % locks.length;
    locks[myLock].lock();
    try {
      boolean result = table[myBucket].remove(x);
      size  = result ? size - 1 : size;
      return result;
    } finally {
      locks[myLock].unlock(); // always unlock
    }
  }
  // resizing and policy methods appear elsewhere ...
}
```

```
  public void resize(int expectedSize) {
    for (Lock lock : locks) {
      lock.lock();
    }
    try {
      if (expectedSize != table.length) {
        return; // someone beat us to it
      }
      int oldCapacity = table.length;
      int newCapacity = 2 * oldCapacity;
      LockFreeList<Item>[] oldTable = table;
      table = (LockFreeList<Item>[]) new LockFreeList[newCapacity];
      for (int i = 0; i < newCapacity; i++)
        table[i] = new LockFreeList<Item>();
      for (LockFreeList<Item> bucket : oldTable) {
        for (Item x : bucket) {
          table[x.hashCode() % table.length].add(x);
        }
      }
    } finally {
      for (Lock lock : locks) {
        lock.unlock();
      }
    }
  }
}
```

Figure 11.4: *Fine-grained hash set: resizing the hash set requires locking each of the locks in order, and checking that no other thread has resized the table in the meantime.*

```
public class LockFreeBucket implements Bucket{
  static final int WORD_SIZE = 32;
  static final int LO_MASK = 0x00000001;
  static final int HI_MASK = 0x80000000;
  private static int makeRegularItem(Object x) {
    return reverse(x.hashCode() | HI _MASK);
   }
  private static int makeSentinelKey(int key) {
    return reverse(key);
   }
  private static int reverse(int key) {
    int loMask = LO_MASK;
    int hiMask = HI_MASK;
    int result = 0;
    for (int i = 0; i < WORD_SIZE; i++) {
      if ((key & loMask) != 0) { // bit set
        result |= hiMask;
        loMask <<= 1;
        hiMask >>= 1;
      }
    }
    return result;
  }
 }
```

Figure 11.5: *Distinguishing regular items from sentinel items*

```
public boolean add(T x) {
  int key = makeRegularKey(x);
  boolean splice;
  while (true) {
    // find predecessor and current entries
    Window window = find(head, key);
    Entry pred = window.pred;
    Entry curr = window.curr;
    // is the key present?
    if (curr.key == key) {
      return false;
    } else {
      // splice in new entry
      Entry entry = new Entry(key, x);
      entry.next.set(curr, false);
      splice = pred.next.compareAndSet(curr, entry, false, false);
      if (splice)
        return true;
      else
        continue;
    }
  }
}
public boolean contains(T x) {
  int key = makeRegularKey(x);
  Window window = find(head, key);
  Entry pred = window.pred;
  Entry curr = window.curr;
  return (curr.key == key);
}
public boolean remove(T x) {
  int key = makeRegularKey(x);
  boolean snip;
  while (true) {
    // find predecessor and current entries
    Window window = find(head, key);
    Entry pred = window.pred;
    Entry curr = window.curr;
    // is the key present?
    if (curr.key != key) {
      return false;
    } else {
      // snip out matching entry
      snip = pred.next.attemptMark(curr, true);
      if (snip)
        return true;
      else
        continue;
    }
  }
}
```

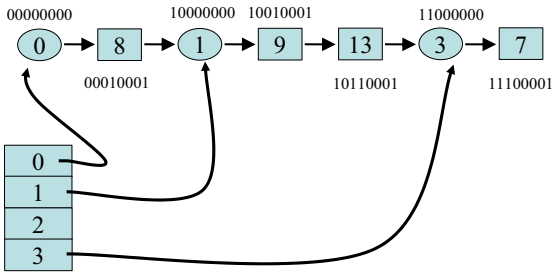

Figure 11.7: *Buckets 0, 1, and 3 are initialized. Bucket 2 is uninitialized.*

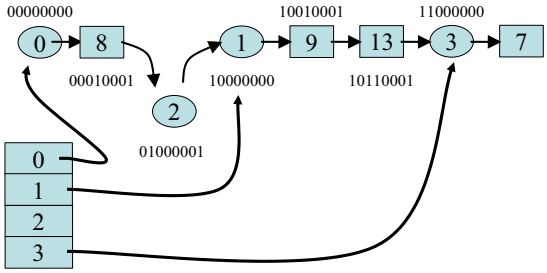

Figure 11.8: *An object with hash value 10 is inserted, requiring bucket 2 to be initialized. A new sentinel is inserted with split-order key 2.*

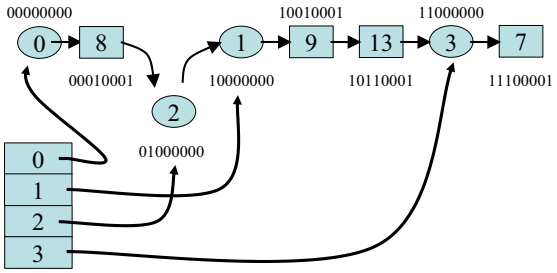

Figure 11.9: *Bucket 2 is assigned a new sentinel.*

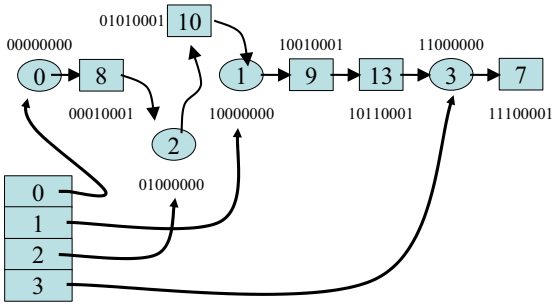

Figure 11.10: *The split-order regular key 10 is inserted in bucket 2.*

Figure 11.11: *LockFreeHashSet class: each method hashes its argument to choose a bucket, ensures that bucket is initialized, and searches the list starting from the bucket reference.*

Figure 11.12: *LockFreeHashSet class: if a bucket is uninitialized, initialize it by adding a new sentinel. Initializing a bucket may require initializing its parent.*

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

Figure 11.13: *Sequential Cuckoo Hashing: the add() method*

```
public boolean add(T x) {
  boolean mustResize = false;
  int h0 = hash0(x);
  T y = null;
  lock0(h0);
  try {
    if (x.equals(table[0][h0])) {
      return false;
    } else {
      int h1 = hash1(x);
      lock1(h1);
      try{
        if (x.equals(table[1][h1])) {
          return false;
        } else if (overflow[h1].contains(x)) {
          return false;
        } else if (table[0][h0] == null) {
          table[0][h0] = x;
          return true;
        } else if (table[1][h1] == null) {
          table[1][h1] = x;
          return true;
        } else {

          y = table[1][h1];
          List<T> over = overflow[h1];
          if (over.size() < OVERFLOW_LIMIT) {
            over.add(x);
            table[1][h1] = x;
          } else {
            mustResize = true;
          }
        }
      } finally {
        unlock1(h1);
      }
    }
  } finally {
    unlock0(h0);
  }
  if (mustResize) {
    resize(size);
    add(x);
  } else if (y != null) {
    reAdd(y, 0);
  }
  return true;
}
```

Figure 11.14: *Concurrent Cuckoo HashSet: add() method*

```
private void reAdd(T y, int depth) {
  int h0 = hash0(y); lock0(h0);
  int h1 = hash1(y); lock1(h1);
  T z = null;
  try {
    if (!overflow[h1].remove(y)) {
      return; // table was resized or entry removed
    }
    if (table[0][h0] == null) {
      table[0][h0] = y;
      return;
    } else if (table[1][h1] == null) {
      table[1][h1] = y;
      return;
    } else {
      z = table[1][h1];
      overflow[h1].add(z);  // there must be room
      table[1][h1] = y;
    }
  } finally {
    unlock1(h1);
    unlock0(h0);
  }
  if (z != null) {
    if (depth > LIMIT) {
      resize(size);
    } else {
      reAdd(z, depth + 1);
    }
  }
}
```

Figure 11.15: *Concurrent Cuckoo Hashing: the reAdd$(\cdot,\cdot)$ method*

```
void resize(int oldSize) {
  for (int i = 0; i < lock[0].length; i++) {
    lock[0][i].lock();
  }
  try {
    if (size > oldSize) { // someone beat us to it
      return;
    }
    T[][] oldtable = table;
    size = 2 * size;
    table = (T[][]) new Object[2][size];
    List<T>[] oldOverflow = overflow;
    overflow = (List<T>[]) new List[size];
    for (int i = 0; i < overflow.length; i++) {
      overflow[i] = new ArrayList<T>();
    }
    for (int i = 0; i < 2; i++) {
      for (T x : oldtable[i]) {
        if (x != null) {
          add(x);
        }
      }
    }
    for (List<T> list : oldOverflow) {
      for (T x: list) {
        if (x != null) {
          add(x);
        }
      }
    }
  } finally {
    for (int i = 0; i < lock[0].length; i++) {
      lock[0][i].unlock();
    }
  }
}
```

Figure 11.16: *Concurrent Cuckoo Hashing: the resize() method*