Solutions to Exercises in Chapter 12

12.1 Figure S12.1(a) shows the effect of successively adding student identifiers to a CBHT.

The number of comparisons when the CBHT is searched for each key is:

000014	990021	990019	970036	000015	970012	970023
2	2	1	3	1	2	1

The average number of comparisons is (2+2+1+3+1+2+1)/7 ≈ 1.7.

Figure S12.1(b) shows the effect of deleting 000014 from the CBHT.

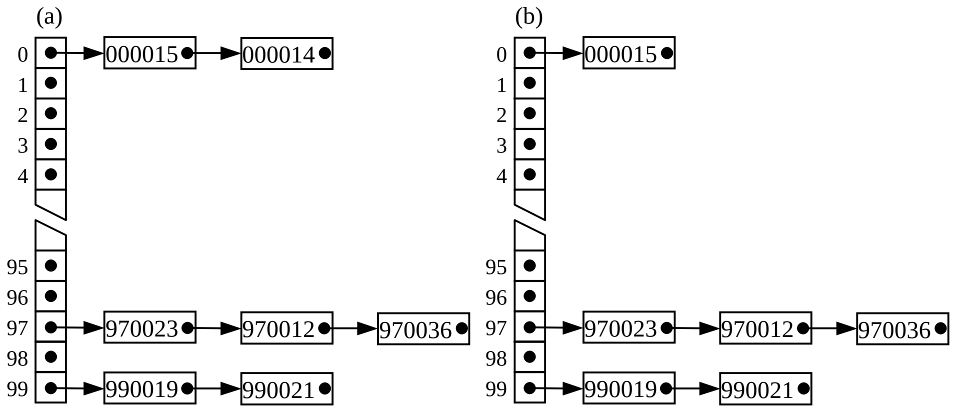

Figure S12.1 Effect of (a) successive insertions, and (b) a deletion, in a CBHT (m = 100, *hash*(*id*) = value of first two digits of *id*).

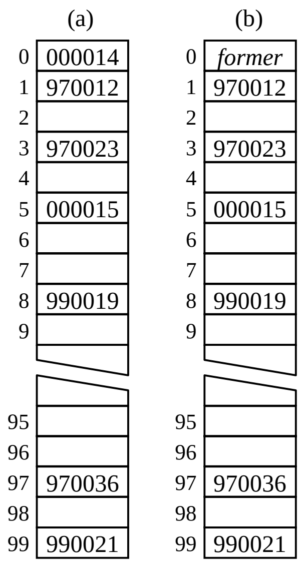

Figure S12.2 Effect of (a) successive insertions, and (b) a deletion, in a doubly-hashed OBHT (m = 100, *hash*(*id*) = value of first two digits of *id*, *step*(*id*) = value of last digit of *id*).

12.2 Figure S12.2(a) shows the effect of successively adding student identifiers to a doubly-hashed OBHT.

The number of comparisons when the OBHT is searched for each key is:

000014	990021	990019	970036	000015	970012	970023
1	1	2	1	2	3	3

The average number of comparisons is (1+1+2+1+2+3+3)/7 ≈ 1.9.

Figure S12.2(b) shows the effect of deleting 000014 from the OBHT.

12.3 If the keys are flight codes, and up to 200 entries are expected, we could choose a CBHT with:

m = 269
hash(*code*) = (weighted sum of characters of *code*) modulo 269

The chosen number of buckets is prime. The load factor is up to 200/269 ≈ 0.74. The chosen hash function will distribute the keys quite uniformly among the buckets.

(*Note:* A hash function that used only the serial number would be inferior, since there are likely to be patterns in the serial numbers used. A hash function that used only the airline code would be even worse, since the hash table might contain entries for few airlines, or even only one airline!)

A doubly-hashed OBHT would also be suitable. The second hash function could be *step*(*code*) = value of last digit of *code* + 2.

12.4 In an analysis of the CBHT search algorithm, if we assume that no bucket is occupied by more than *five* entries, the maximum number of comparisons is 5.

If instead we assume that all the entries are distributed (evenly) over *ten* buckets, the maximum number of comparisons is *n*/10.

The algorithm's best-case and worst-case time complexities are still *O*(1) and *O*(*n*), respectively.

12.5 If the keys are Web server names, a hash function that used only the first six characters would be bad because there is a strong pattern in server names: nearly all have the prefix "www.".

A more suitable hash function would be:

hash(*name*) = (weighted sum of characters of *name*) modulo *m*

This could be further improved by ignoring the prefix of *name*.

12.6 The auxiliary `expand` method in the `OBHT` class can be implemented as follows. First make `buckets` refer to a new and longer array, in which every bucket is never-occupied. Then take every occupied entry in the old array, and re-insert that entry using the `insert` method. (This works correctly because both the `insert` method and the auxiliary `hash` method access `buckets`, which now refers to the *new* array.)

```
private void expand () {
// Expand the number of buckets, rehashing all existing entries.
   BucketEntry[] oldBuckets = buckets;
   int newLength = oldBuckets.length*3/2;
   buckets = new BucketEntry[newLength];
   load = 0;
   for (int b = 0; b < oldBuckets.length; b++) {
      BucketEntry oldEntry = oldBuckets[b];
      if (oldEntry != null
            && oldEntry != BucketEntry.FORMER)
         insert(oldEntry.key, oldEntry.value);
   }
}
```

(*Note:* The above implementation simply multiplies the number of buckets by

3/2. The resulting number of buckets might be nonprime. A possible improvement would be to find the smallest prime number not less than 3/2 times the old number of buckets.)

The `insert` method in the `OBHT` class is easily modified to make it limit the load factor to 0.75, as follows (change from Program 12.18 italicized):

```
public void insert (Object key, Object val) {
// Insert the entry ‹key, val› into this OBHT.
  BucketEntry newEntry =
      new BucketEntry(key, val);
  int b = hash(key);
  for (;;) {
    BucketEntry oldEntry = buckets[b];
    if (oldEntry == null) {
      if (++load > buckets.length*3/4) {
        expand();
        b = hash(key);
        continue;
      }
      buckets[b] = newEntry;
      return;
    } else if (oldEntry == BucketEntry.FORMER
        || key.equals(oldEntry.key)) {
      buckets[b] = newEntry;
      return;
    } else
      b = (b + 1) % buckets.length;
  }
}
```

12.7 It makes sense to seek a perfect hash function only when the set of keys is fixed and known by the programmer. (The idea of a perfect function was invented by a compiler writer, who wanted a hash table to contain the set of keywords of the C programming language.)

A perfect hash function guarantees no collisions, so we can simplify the hash table algorithms (and code) by eliminating collision resolution.

A perfect hash function for {CA, MX, US} is:

m = 3
hash(*ctry*) = (first letter of *ctry* – ‘A’) / 9

A perfect hash function for {AT, BE, DE, DK, ES, FI, FR, GR, IE, IT, LU, NL, PT, SE, UK} is:

m = 19
hash(*ctry*) = (3 × (first letter of *ctry* – ‘A’)
– (second letter of *ctry* – ‘A’) / 10) modulo 19

12.8 The `CBHT` class augmented by an unordered iterator is outlined in Program S12.3.

12.9 The `OBHT` class augmented by an unordered iterator is outlined in Program S12.4.

```
public class CBHT {

   private BucketNode[] buckets;

   …

   public Iterator iterator () {
      return new CBHT.UnorderedIterator();
   }

   //////// Inner class for CBHT unordered iterators ////////

   private class UnorderedIterator
                  implements Iterator {

      // A CBHT.UnorderedIterator object is an iterator that
      // will visit all entries of this CBHT, in no particular order.

      // This iterator is represented by a link to the node containing the
      // next entry to be visited, place, together with the index of
      // the bucket containing that node, b.
      private BucketNode place;
      private int b;

      public UnorderedIterator () {
         findNextOccupiedBucket(0);
      }

      public boolean hasNext () {
         return (b < buckets.length);
      }

      public Object next () {
         if (b == buckets.length)
            throw new NoSuchElementException();
         Object nextEntry = place;
         place = place.succ;
         if (place == null)
            findNextOccupiedBucket(b+1);
         return nextEntry;
      }

      private void findNextOccupiedBucket (
                  int from) {
         for (b = from; b < buckets.length; b++) {
            place = buckets[b];
            if (place != null)  break;
         }
      }

   }
}
```

Program S12.3 A CBHT unordered iterator.

```
public class OBHT {

   private BucketEntry[] buckets;

   …

   public Iterator iterator () {
      return new OBHT.UnorderedIterator();
   }

   //////// Inner class for OBHT unordered iterators ////////

   private class UnorderedIterator
                  implements Iterator {

      // An OBHT.UnorderedIterator object is an iterator that
      // will visit all entries of this OBHT, in no particular order.

      // This iterator is represented by the bucket index of the next
      // entry to be visited, b.
      private int b;

      public UnorderedIterator () {
         findNextOccupiedBucket(0);
      }

      public boolean hasNext () {
         return (b < buckets.length);
      }

      public Object next () {
         if (b == buckets.length)
            throw new NoSuchElementException();
         Object nextEntry = buckets[b];
         findNextOccupiedBucket(b+1);
         return nextEntry;
      }

      private void findNextOccupiedBucket (
                  int from) {
         for (b = from; b < buckets.length; b++) {
            BucketEntry entry = buckets[b];
            if (entry != null
                  && entry != BucketEntry.FORMER)
               break;
         }
      }

   }
}
```

Program S12.4 An OBHT unordered iterator.

12.11 The `OBHT` class augmented by an ascending iterator is outlined in Program S12.5.

An important disadvantage of this implementation is that the iterator will visit the entries that were in the hash table *at the time the iterator was created*. The iterator will fail to 'notice' any subsequent insertions or deletions.

```
public class OBHT {

   private BucketEntry[] buckets;

   …

   public Iterator iterator () {
      return new OBHT.AscendingIterator();
   }

   //////// Inner class for OBHT ascending iterators ////////

   private class AscendingIterator
                  implements Iterator {

      // An OBHT.AscendingIterator object is an iterator that
      // will visit all entries of this OBHT, in ascending order.

      // This iterator is represented by a sorted array of entries,
      // entries[0…count–1], together with the index of the next
      // entry to be visited, place.
      private BucketEntry[] entries;
      private int count, place;

      public AscendingIterator () {
         entries = new BucketEntry[buckets.length];
         count = 0;
         for (int b = 0; b < buckets.length; b++) {
            BucketEntry entry = buckets[b];
            if (entry != null
                  && entry != BucketEntry.FORMER) {
               count++;
               … // Insert entry into entries[0…count–1],
                  // keeping the subarray sorted.
            }
         }
         place = 0;
      }

      public boolean hasNext () {
         return (place < count);
      }

      public Object next () {
         if (place == count)
            throw new NoSuchElementException();
         return entries[place++];
      }

   }
}
```

Program S12.5 An OBHT ascending iterator.