Hashing with linear probing (part 2)

The fields for implementing the set

We use an array b of type E[] for the buckets. We want the space required for the booleans to be a minimum: one bit per boolean. However, a Java boolean array is implemented with each boolean take *at least* a full byte —8 times as many bits as is necessary! But Java class java.util.BitSet *does* implement a collection of booleans with 1 bit per boolean. The name is ill chosen; it should be called BitList, not BitSet, because it implements a list of bits that grows as needed, much like an ArrayList. It has the obvious method calls get(i), set(i) to set the bit to true, and flip(i) to flip the bit. So, we declare field in.

Let's write down the class invariant; we'll use array notation for in.

```
/*      (1) The set consists of the b[i] for which in[i] is true.
 *          No element is in b more than once.
 *      (2) b[i] = null implies in[i] is false
        (3) If an element hashed to h but was placed in b[h+k] (with wraparound),
            then all elements b[h..h+k] are not null.
        (4) To remove b[i] from the set, set in[i] to false */
private E[] b;
private BitSet in;
```

First,

Second, since null is not allowed in the set,…

Third, to enforce that remove cannot set an element to null,

Fourth, this is not part of the invariant but just a reminder about what

We need a field to contain the size of the set and a field load, which contains the number of b[i] that are not null —if elements are removed, load > size. We'll see how load is used later.

Method linearProbe

With this definition, we can write method linearProbe to search for element e, returning either the bucket where it resides or the null bucket that ended the search. First, hash e to get a bucket number h. Now perform a conventional linear search but with wraparound, starting at b[h]. The search should stop when either e or **null** is found, so we write the invariant. Then, the initialization and loop is written in standard fashion. Index i is returned.

```
/** = index in array b of e or where e will be put (using linear probing).
  *   Precondition: e may not be null. */
private int linearProbe(E e) {
   assert e != null;
   int h= Math.abs(e.hashCode() % b.length);
   int i= h;
   // inv: e is not one of b[h..k+i-1] (with wraparound)
   while (b[i] != null  &&  !e.equals(b[i])) {
      i= (i+1) % b.length;
   }
   return i;
   }
```

We write method add. First, call linearProbe and store the result in h. There are now three cases to consider:

1. in[h] is true. By the class invariant, e is already in the set, in b[h].
2. in[h] is false and b[h] is not null. Then b[h] = e, and e can be added to the set by setting in[h] to true.
3. in[h] is false and b[h] is null. Then e is to be placed in b[h] and in[h] has to be set to true.

In handling these cases, we also have to maintain fields size and load and also rehash if necessary. Here we go.

First, if in[h] is true, e is already in the set, in b[h], and false is returned.

Next, since in[h] is false, e has to added to the set, so the size of the set is increased and bit b[h] is set to true. If b[h] is not null, then b[h] equals e. This means that e was previously removed from the set. Nothing more needs to be done and true is returned.

Finally, since b[h] is null, e is stored in b[h] and the load is increased by 1, since 1 more element of b is non-null. The set is rehashed if necessary, and true is returned since e was added to the set.

Hashing with linear probing (part 2)

```
/** Ensure that e is in this set and return value of sentence
      "e was added because it was not in the set." */
public boolean add(E e) {
    int h= linearProbe(e);
    if (in.get(h)) return false; // e is already in the set

    size= size + 1;
    in.flip(h);

    if (b[h] != null) { // e was in the "removed" state. Put it back in.
        return true;
    }

    // e is not in set and is to be placed in b[h]
    b[h]= e;
    load= load + 1;
    if (load > .75 * b.length) rehash();  // b is too small
    return true;
}
```