# Determining execution time

## Terminology

Let's introduce some terminology. The *capacity* is the size of array b, in this case, 9. The *load factor* is the size of the set divided by the capacity, in this case 6/9, or two thirds. A *probe* is either a test of some b[i] to see whether it is null or a comparison of a linked-list element with some value. We investigate the number of probes to determine whether an element is in the set.

## Average time to search

An important point about the hash function is that it randomly directs elements to buckets. The diagram on the monitor shows *one* way in which the 6 elements could have been placed in the set, but just as likely, each of the 6 elements could have been placed in a single bucket, or all of them in one bucket.

Consider searching for an element e that is not in the set when all elements are in one bucket. In one possible case, 7 probes are made: the first probe finds that bucket b[2] is a linked list and then 6 probes determine that e is not in the list. In the other 8 cases, *one* probe is needed. Thus, on the *average*, (7 + 8) / 9 = 15/9 probes, less than 2 probes are needed.

This kind of analysis can be made with all other configurations. In fact, someone has proved that if the load factor is ½, the expected, or average, number of probes is at most 2, even if the set contains 1,000 elements! This is at first astonishing. That is why hashing is so effective.

## Making the array larger.

But as the load factor increases past ½, the average number of probes needed to determine if an element is in the set increases. At some point, it is best to create a new array, hash all the elements into the new array, and use the new array. The default for this in Java's class HashSet is when the load factor reaches.

Here's a method to do that. First, save a pointer to array b. Next, create a new array b and set size to 0. The length of the new array is 3*the size of the set. This method is used also to decrease the array size when the load factor gets too small; that's the reason it is not allowed to get smaller than its initial capacity.

Now, process each bucket in the original array. If it is a linked list (that is, not null), add each element of the linked list to the new array b. Note that the original method add is used! We will look at it in a moment.

```
/** Rehash array b */
private void rehash( ) {
    LinkedList<E>[] oldb= b;  // copy of array

    // Create a new, empty array and set size to 0.
    b= new LinkedList[Math.max(3*size, Initial_Capacity)];
    size= 0;

    // Copy elements from oldb to b
        for (LinkedList<E> list: oldb)
            if (list != null) {
                for (E e : list) add(e);
            }
}
```

## Method add

The last line of method add has to be changed. First, attempt to add the element to the linked list and store the result in a variable. Next, if the load factor is greater than .75, rehash. Note that the test is not made using an integer division because that would not work; the result of the division would be 0. Finally, the result of calling method add is returned.

## Amortizing rehashing into a new array

We *do* have to take into account the cost of the time of creating the new array and rehashing all the set elements into it. To do this, we introduce the notion of *amortization*, in the following way.

# Determining execution time

A year ago, we bought a SodaStream fizz maker, let's say for $100.00. We bought it to save money. We don't have to buy plastic bottles of fizzy water anymore, because this machine adds fizz to a glass of water.

Think about it this way. After making one glass of fizzy water with the machine, we said that the glass cost us $100.00. After making two glasses, we said each glass cost us $50.00. After making 100 glasses, each glass cost us $1.00; after 1,000 glasses, each glass cost us 10¢. We *amortized* the initial cost of the machine over the glasses of fizzy water that we made with it. That's amortization.

Now consider having to rehash each of the elements in the set. Each of those elements cost O(1) expected time to add them to the set, earlier. Now, we *amortize* the cost of rehashing the set over those elements, Each element now cost us expected time O(1) to initially add it plus expected time O(1) to rehash it; in total, that's still expected *amortized* time O(1) for each element. *Amortization* is a useful concept!