# Hashing By David Gries

Hashing is a technique for maintaining a set of elements. Read Carrano (course text), chapter 21, but we give most important concepts and information in this short handout.

A set is just a collection of distinct (different) elements on which the following operations can be performed:

- Make the set empty
- Add an element to the set
- Remove an element from the set
- Get the size of the set (number of elements in it)
- Tell whether a value is in the set
- Tell whether the set is empty.

**Obvious first implementation**: Keep elements in an array b. The elements are in b[0..n-1], where variable n contains the size of the array. No duplicates allowed.

**Problems:** Adding an item takes time O(n) —it should not be inserted if it is already in the set, so b[0..n-1] has first to be searched for it. Removing an item also takes time O(n) in the worst case. We would like an implementation in which the expected time for these operations is constant: O(1).

**Solution:** Use *hashing*. We illustrate hashing assuming that the elements of the set are Strings.

**Basic idea**: Rather than keep the Strings in b[0..n-1], we allow them to be anywhere in the b. We use an array whose elements are of nested class HashEntry:

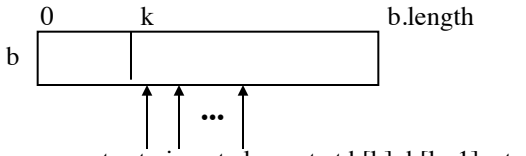
```
/** An instance is an element in hash array */
private static class HashEntry {
   public String  element;     // the element
   public boolean isInSet;     // = "element is in set"

   /** Constructor: an entry that is in the set  iff b */
   public HashEntry( String e, boolean b) {
         element=  e;
         isInSet= b;
   }
}
```

Each element of array b is either **null** or the name of a HashEntry, and that entry indicates whether it is in the set or not. So, to remove an element of the set, just set its isInSet field to **false**.

**Hashing with linear probing.** Here's the basic idea. Suppose we want to insert the String "bc" into the set. Compute an index k of the array, using what is called a hash function,

```
int k=  hashCode("bc");
```

and try to store the element at position b[k]. If that entry is already filled with some other element, try to store it in b[(k+1)%b.length] —use wraparound. If that position is filled, keep trying successive elements in the same way.




Each test of an array element to see whether it is null or the String is called a **probe.**

The hash function picks some index, depending on its argument. We show a hash function later.

Checking to see whether a String "xxx" is in the set is similar; compute k= hashCode("xxx") and look in successive elements of b[k..] (using wraparound) until a **null** element is reached or until "xxx" is found. If it is found, it is in the set iff the position in which it is found has its isInSet field **true**.

You might think that this is a weird way to implement the set, that it couldn't possibly work. But it does, provided the set doesn't fill up too much, and provided we later make some adjustments.

Basic fact:

> Suppose String s is in the set and hashCode(s) = k. Let b[j] be the first **null** element at or after after b[k] (with wraparound). Then s is one of the elements b[k], b[k+1], …, b[j-1] (with wraparound).

Because of the basic fact, we can write method add as follows, assuming that array b is never full:

# Hashing

```
/** Add s to this set */
public void add(String s) {
   int k= hashCode(s);
   while (b[k] != null  &&  !b[k].element.equals(s)) {
      k= (k+1) % b.length();
   }
   if (b[k] == null) {
      b[k]=  new HashEntry(s, true);
      size= size+1;
      return;
   }
   // s is in b[k] but may not be in set.
   if (!b[k].isInSet) {
         b[k].isInSet=  true;
         size=  size + 1;
    }
}
```

Removing an element is just as easy. Note that removing it leaves it in the array.

```
/** Remove s from this set (if it is in it) */
public void remove(String s) {
   int k= hashCode(s);
   while (b[k] != null  &&  !b[k].element.equals(s)) {
      k=  (k+1) % b.length();
   }
   if (b[k] == null  ||  !b[k].isInSet)
       return;
   // s is in the set; remove it.
   b[k].isInSet= false;
   size= size-1;
}
```

**Hash functions**

We need a function that turns a String s into an **int** that is in the range of array b. It doesn't matter what this function is as long as it distributes Strings to integers in a fairly even manner. Here is one function, assuming that s has 4 chars.

$$s[0]*37^3 + s[1]*37^2 + s[2]*37^1 + s[3]*37^0$$

i.e.

$$((s[0]*37 + s[1])*37 + s[2])*37 + s[3]$$

Reduce result modulo b.length to produce an **int** in the range of b. Some of the above calculations may overflow, but that's okay. The overflow produces an integer in the range of **int** that satisfies our needs.

We discuss other hash functions later.

**What about the load factor?**

The load factor lf is defined as follows:

lf = (size of elements of b in use) / (size of array b)

The load factor is an estimate of how full the array is. If it is close to 0, the array is relatively empty, and hashing will be quick. If lf is close to 1, then adding and removing elements will tend to take time linear in the size of b, which is bad. Here's what someone proved:

> Under certain independence assumptions, the average number of array elements examined in adding an element is 1/(1-lf).

So, if the array is half full, we can expect adding an element to look at 1/(1-1/2) = 2 array elements. That's pretty good! If the set contains 1,000 elements and the array size is 2,000, only 2 probes are expected!

So, we keep the array no more than half full. Whenever insertion of an element will increase the number of used elements to more than 1/2 the size of the array, we will "rehash". A new array will be created and the elements that are in the set will be hashed into it. Of course, this takes time, but it is worth it. Here's the method:

```
/** Rehash array b */
private void rehash( ) {
    HashEntry[] oldb= b;  // copy of array b

    // Create a new, empty array
      b= new HashEntry[nextPrime(4 * size)];
      size= 0;

    // Copy active elements from oldb to b
      for (int i= 0; i != oldb.length; i= i+1)
        if (oldb[i] !== null  &&  oldb[i].isInSet)
          b.add(oldb[i].element);
}
```

Size of new array: the smallest prime number that is at least 4*b.size(). The reason for choosing a prime number is explained on the next page.