Hashing

Hashing is a technique for maintaining a set of elements in an array. You should also read Weiss, chapter 20, which goes into more detail (but is harder to read).

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

Problems: Adding an item take time O(n) --it shouldn't 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 the following nested class type:

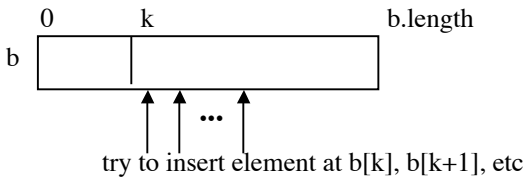
```
// An instance is an entry in array b
private static class HashEntry {
   public String  element;     // the element
   public boolean isInSet;     // = "element is in set"

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

Each element of our 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. We compute an index k of the array, using what's 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, we try to store it in b[(k+1)%b.length] --note that we use wraparound, just as in implementing a queue in an array. If that position is filled, we 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 just picks some index, depending on its argument. We'll 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..] 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.

Here's a basic fact:

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

Then, 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.isInSet)
      return;

   // s is not in the set; store it in b[k].
      b[k]= new HashEntry(s, true);
      size= size+1;
}
```

Removing an element is just as easy. Note that removing a value from the set 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;
}
```

Hashing 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 the function that Weiss uses, assuming that s has 4 characters.

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

i.e.

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

The result is then reduced modulo the size of array b 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.

See Sec. 20.2 of Weiss for an example of this hash function as a Java method.

What about the load factor?

The load factor, lf, is the value of

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 lf 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 an addition 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 over 2,000, only 2 probes are needed!

So, we will 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 copied over to 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)
              add(oldb[i].element);
}
```

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

Quadratic probing.

Linear probing looks for a String in the following entries, given that the String hashed to k (we implicitly assume that wraparound is being used):

b[k], b[k+1], b[k+2], b[k+3], …

This tends to produce clustering --long sequences of nonnull elements. This is because two Strings that hash to k and k+1 use almost the same probe sequence.

A better idea is to probe the following entries:

b[k], (for obvious reasons,
b[k + 1^2] this is called
b[k + 2^2] "quadratic probing")
b[k + 3^2]
...

This has been shown to remove the "primary clustering" that happens with linear probing. However, Strings that hash to the same value k still use the same sequence of probes. There are ways to eliminate this "secondary clustering", but we won't go into them here. We just want to present the basic ideas.

Quadratic probing has been shown to be feasible if the size of array b is a prime and if the table is always at least 1/2 empty. In this case, it has been proven that:

- A new element can always be added, and
- its probe sequence never probes the same array elements twice.

Calculating the next element to probe

The calculation of $k+i^2$ is expensive. We show how to make it more efficient.

Let $H_i = k+i^2$, for i = 0, 1, 2, 3

For i>0, we calculate:

$H_i - H_{i-1}$
= <definition of H_i and H_{i-1} >
$k+i*i - (k+(i-1)*(i-1))$
= <arithmetic>
$2*i - 1$

Therefore, we can calculate H_i from H_{i-1} using the formula $H_i = H_{i-1} + 2*i - 1$.

An implementation

The CS211 course website contains a file HashSet.java --look under "recitations". An instance of class HashSet implements a set as a hash table, using the material discussed in this handout. File Main.java contains a method main that is used to test HashSet (at least partially).

When you look at HashSet, think of the following:

• Class HashSet contains a nested class, HashEntry. This class can be static because it does not refer to any fields or methods of class HashSet. It is nested because there is no need for the user to know anything about it. One such good use of nested classes is information hiding, as we do here.

• Class HashSet contains an inner class, HashSet-Enumeration. It can't be a nested class because it DOES make use of fields of class HashSet. This is a good use of inner classes for information hiding.

•Enumerating the elements of the set does NOT produce them in ascending order.

•We do not use the String hash function described in this handout. Instead, we make use of a function hashCode that is supplied in many Java API classes. Method hashCode is first defined in class Object, the superest class of them all. Method hashCode in class String actually computes the hash code using the equivalent of:

((s[0]*31 + s[1])*31 + …)*31 + s[s.length()-1]