Hashing

See Section 10.2 of the text.

Maps in general are associative structures -- they associate values with keys and allow for efficient searches based on the keys. TreeMaps use balanced binary search trees based on comparative properties of the keys. We know that we can search a balanced binary search tree with n items in time O(log(n)), so these perform well. However, there is a second Map implementation called a **HashMap** that is sometimes preferable. HashMaps have two advantages:

a) HashMaps to not require us to compare values of the keys, so the Key class does not need to implement the Comparable interface.
b) Under certain reasonable conditions HashMaps give constant-time searches.

These properties don't come without any cost. You lose some things with HashMaps.

TreeMaps make it easy to find the smallest key. In TreeMaps it isn't difficult to go from one key value to the next, or to get an ordered list of the current keys. You don't easily get those things with HashMaps.

Here is the idea of hashing. Suppose we want to represent a set of numbers in the range from 0 to 999. One way would be to make an AVL tree with base type Integer that held the numbers in the set. The lookup time to determine if something is in the set would be the logarithm of the size of the set.

Here is an alternative -- maintain an array A of 1000 booleans. Initialize the entries to false. Add a number n to the set by changing A[n] to true. Then to determine of number n is in the set, just return A[n]. That is certainly constant-time insertion and constant-time lookup.

Suppose instead we had sets of colors, with the only color options being red, green, blue, yellow, black, white, purple and chartreuse. We could arbitrarily assign numbers to the colors, such as red 0, green 1, blue 2, yellow 3, black 4, white 5, purple 6, and chartreuse 7 and play the same game with an array of 8 boolean entries -- element [3] of the array is true if the set includes the color yellow.

Such a function, which inputs an object and returns a number for it is called a "hash function". The array is called a HashTable and its use to provide dictionary-type structures (associating values with keys) is called a HashMap.

By the way, the "hash" part of the name comes not from hashish, which we know Alice B. Toklas put in brownies, but from hash as a mixture of foods (e.g. corned beef hash), since the data in a hash table is mixed up in what seems to be random order.

The hash function tells us where to look in the table or array for a value. There is one complication. In most situations the space of data values is vastly larger than the size of the table. For example, we might want to maintain a set of people, and use their names as the keys.

If you consider <first name, last name> pairs such as "Bob Geitz" or "Marvin Krislov" there is an enormous number of possible names. If the typical set size is 10 or so, it would be very wasteful to make a hash table with one entry for every possible name, even if we had a catalog of all possible names. If we use a small table and require the hash function to map keys into table indices, it is inevitable that some keys will hash to the same index. This is called a "collision".

Here is how Java computes the hash value of a string s: Suppose s has length n, so its entries are s[0], s[1], ... s[n-1].

Let u[i] be the numeric unicode value of s[i] (65 for 'A', 97 for 'a', etc.).

Then the hashCode for s is

$$u[n-1]31^0 + u[n-2]31^1 + \ldots + u[0]31^{n-1}$$

For a long string this will overflow the size of an integer, which means that it might appear positive or negative.

For example, the integer values of the characters 'b' and 'o' are 98 and 111 respectively. So the hashCode for "bob" is

$$98*31^0+111*31^1+98*31^2 = 97717.$$

Indeed, if you execute the line

```
System.out.println( "bob".hashCode() );
```

it prints 97717

The Java hashCode is computed independently of any particular hash table. Once you have a table you can compute the hash function as

```
hashValue = hashCode % tableSize;
if (hashValue < 0)
        hashValue += tableSize;
```