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 or get a list of the current keys. You don't easily get those 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 \texttt{<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 the string 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].

Then the hashCode for s is

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

For a long string this will overflow the size of an integer, which means that it might appear positive or negative.
The Java hashCode is computed independently of any particular table. Once you have a table you can compute the hash function as

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

Let's do an example -- add some people to a hash table of size 7.

<table>
<thead>
<tr>
<th>Name</th>
<th>h = hash(name)</th>
<th>h%7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ben</td>
<td>66667</td>
<td>6</td>
</tr>
<tr>
<td>Bob</td>
<td>66965</td>
<td>3</td>
</tr>
<tr>
<td>Steven</td>
<td>-1808493797</td>
<td>-5 -&gt; 2</td>
</tr>
<tr>
<td>Cynthia</td>
<td>-1392489180</td>
<td>-5 -&gt; 2</td>
</tr>
<tr>
<td>Alexa</td>
<td>63347171</td>
<td>6</td>
</tr>
<tr>
<td>Jackie</td>
<td>-2083773093</td>
<td>-3 -&gt; 4</td>
</tr>
</tbody>
</table>

The first three are simple:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Steven</td>
<td>Bob</td>
<td></td>
<td>Ben</td>
</tr>
</tbody>
</table>
Where do we now add Cynthia, who hashes to index 2? One answer is to move over until we find a free spot in the table. Indices 2 and 3 are occupied but 4 is not, so we insert Cynthia there:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Steven</td>
<td>Bob</td>
<td></td>
<td>Cynthia</td>
<td>Ben</td>
</tr>
</tbody>
</table>

We call such a situation, where we want to add an item to a hashtable at a location that is already occupied, a "collision". One sure way to get a collision is to have two object have the same hash values.
If we add Alexa, who also hashes to 6, to the table, we see she collides with Ben. There is no room to the right of Ben, so we wrap around and put Alexa at position 0:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alexa</td>
<td>Steven</td>
<td>Bob</td>
<td>Cynthia</td>
<td></td>
<td>Ben</td>
<td></td>
</tr>
</tbody>
</table>

Now suppose we want to add Jackie to the table. She hashes to 4, so she collides with Cynthia. Note that this is a new kind of collision. No one else in the table has the same hash value as Jackie, but she collides because Cynthia was moved away from the index she hashed to.
We resolve this collision in the same way as before and put Jackie at index 5.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alexa</td>
<td></td>
<td>Steven</td>
<td>Bob</td>
<td>Cynthia</td>
<td>Jackie</td>
<td>Ben</td>
</tr>
</tbody>
</table>

Now suppose we want to determine if Cynthia is in the table. She hashes to 2, which is occupied by someone else. But of course she could have collided with the person at index 2 (as she did) so we look to the right. She isn't at index 3, but she is at index 4. We can find items in the table even if they have moved because of a collision.
Now let's see if Chris is in the table. He hashes to index 3. Slots 3, 4, 5, and 6 are all occupied with someone other than Chris. We can't move to the right from index 6, so we wrap around to index 0. That slot is also occupied, but the next slot, at index 1, is not. If Chris was in the table he would have been at index 1, if not in one of the slots we examined earlier, so we can be certain that he is not in the table.
Consider what would happen if we removed Bob from the table. If we then searched for Cynthia, who hashes to 2, we would see that slot 2 is filled with someone else, but slot 3 was vacant. We would erroneously conclude that Cynthia was not in the table. Rather than actually removing Bob, we need to either replace it with a token "something used to be here" marker or else set a flag in the Bob entry that says it has been removed.
There are two standard approaches to resolving collisions -- *open addresses* and *chaining*. With open addresses we move through the table looking for open slots to insert items that collided with previous entries.
The simplest version of open addressing is "linear probing", which is exactly what we have just described. To insert an item, get its hash value and go to that entry of the table. If that entry is empty, that is where you insert the item. If it is not empty, go to the next entry of the table; if it is empty insert the item there. Continue this process, wrapping from the end of the table to the beginning, until you have found a place to insert the new object. If there is an empty entry in the table, this will find it.

To search for an item you must repeat this process. Start at the index given by the hash value for the object. If that is not the object you are seeking, go to the next object, the next, and so forth. If you get to a vacant spot without finding your object it is not in the table.
Most hash functions tend to have clusters of objects that have similar hash values. This can result in large sequential blocks of the hash table that are filled even when the table itself is nowhere near capacity. If you seek an object that hashes to the start of such a block you end up doing a linear search through the block looking for your object. This is bad, since the whole point of hashing is to get constant-time lookups.
Another collision resolution scheme, which many prefer, is called *quadratic probing*. Suppose an object hashes to index n. Linear probing considers entries n, (n+1), (n+2) etc. until either the object or an empty entry is found. Quadratic probing considers locations n, (n+1), (n+4), (n+9) etc. In general, linear probing looks at locations (n+i), while quadratic probing looks at (n+i^2).
It is obvious that if the table is not completely full then linear probing will eventually find a spot to insert any new item. This is not so obvious with quadratic probing. However, we can say this: if the table size is prime and if the table is no more than half full, then quadratic probing will find an empty location for any insertion. The proof of this is a little game with number theory:
Suppose the sequence \( n, n+1, n+4, n+9, \ldots \) repeats itself. This means we have values \( i \) and \( j \) with
\[
n + i^2 = n + j^2 \pmod{\text{Size}}
\]
Then
\[
i^2 = j^2 \pmod{\text{Size}}
\]
\[
i^2 - j^2 = 0 \pmod{\text{Size}}
\]
\[
(i-j)(i+j) = 0 \pmod{\text{Size}}
\]
This means that \((i-j)(i+j)\) is a multiple of \(\text{Size}\). If \(i\) and \(j\) are different and both no more than \(\text{Size}/2\), this can't happen because \(\text{Size}\) is prime. So the first \(\text{Size}/2\) entries of the sequence must all be different. If the table is no more than half full, one of these locations must be an open slot.