Hashing

See Chapter 20 of Weiss.

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 over TreeMaps:

a) HashMaps to not require us to compare values of the keys, so we do not need comparators.
b) Under certain reasonable conditions HashMaps give **constant-time** searches.

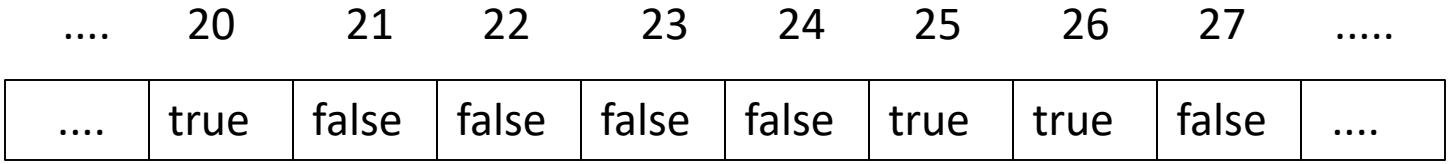
These properties don't come without any cost. You lose some things with HashMaps.

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 if number n is in the set, just return A[n]. That is certainly constant-time insertion and constant-time lookup.

Here is a picture of what part of that array might look like for a set that contains 20, 25, and 26 but not 21, 23, 24, 27, etc.

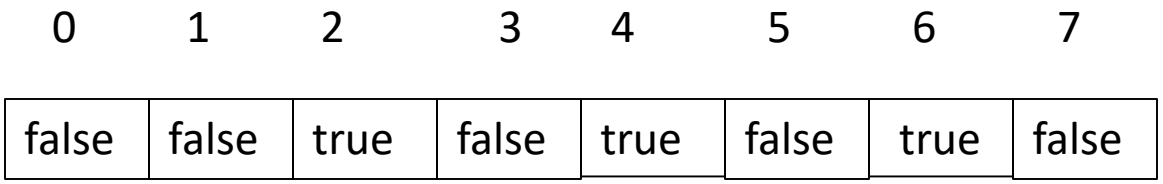


This is one instance of a common occurrence in algorithms, where time and space are interchangeable – it might seem inefficient in terms of space to use 1000 Booleans to represent a set that might have only a few values in it, but that gets us a very fast lookup. We could save on space at the cost of slower lookups. So in any specific situation decide whether space or time is more important to you.

Suppose instead of numbers we had a group of 8 people: "Joe", "Sal", "Steven","Tony", "Sagana", "Uma","Ezra",and "Stella" and we wanted to represent a set containing some of those people.

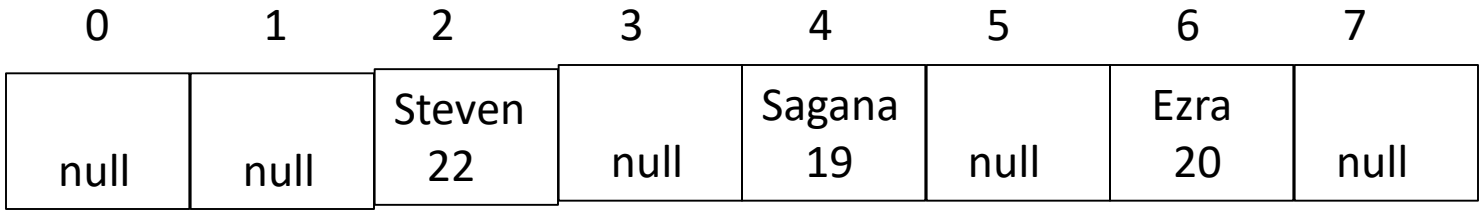
We could arbitrarily assign indices to the names, such as 0 for "Joe", 1 for "Sal", 2 for "Steven",3 for "Tony", 4 for "Sagana",5 for "Uma",6 for "Ezra",and 7 for "Stella" and then play the same game with a boolean array of size 8:

Here is the array that would describe the set {"Steven", "Sagana", "Ezra"}:



Key:
"Joe" 0
"Sal" 1
"Steven" 2
"Tony" 3
"Sagana" 4
"Uma" 5
"Ezra" 6
"Stella" 7

If we want to map people's names to their ages, where name is the key and age is the value, we could take this a step further, writing the (key, value) pairs into the array and leaving any unused slot null:



Key:
"Joe" 0
"Sal" 1
"Steven" 2
"Tony" 3
"Sagana" 4
"Uma" 5
"Ezra" 6
"Stella" 7

We need a way to assign array indices to objects. This 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 "Carmen Ambar" 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".

We need to talk about how to resolve collisions.

First, we will look at how Java makes hash functions.

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

Similarly, the Unicode values for 'O', 'z', 'u',and'm' are 79, 122, 117, and 109, so Java's hashCode for "Ozum" is

$$109*31^0+117*31^1+122*31^2+79*31^3 = 2474467$$

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

```
int hashFunction( Object x, int tableSize ) {
        int value = x.hashCode() % tableSize;
        if (value < 0)
                value += tableSize;
        return value;
}
```