More on Hashing: Collisions

See Chapter 20 of the text.

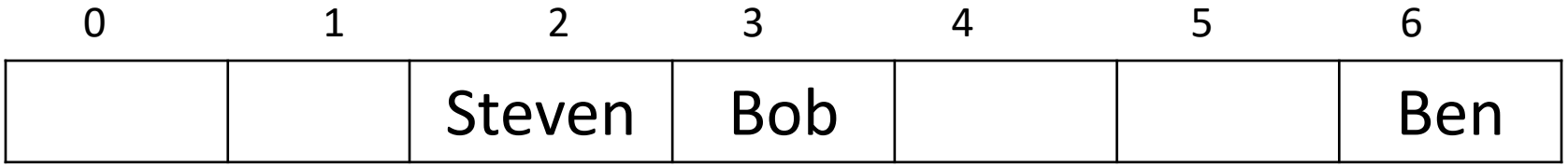
Collisions

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

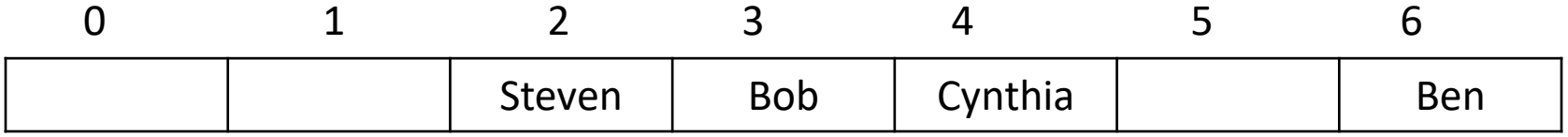
Name	h = hash(name)	h%7
Ben	66667	6
Bob	66965	3
Steven	-1808493797	-5 -> 2
Cynthia	-1392489180	-5 -> 2
Alexa	63347171	6
Jackie	-2083773093	-3 -> 4

The first three are simple:

0	1	2	3	4	5	6
		Steven	Bob			Ben



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:



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.

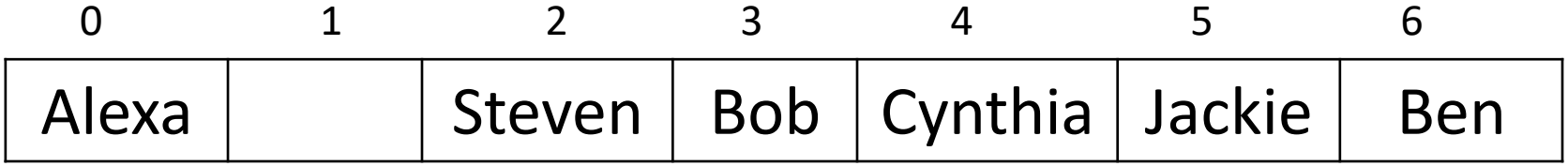
0	1	2	3	4	5	6
		Steven	Bob	Cynthia		Ben

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:

0	1	2	3	4	5	6
Alexa		Steven	Bob	Cynthia		Ben

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.



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.

0	1	2	3	4	5	6
Alexa		Steven	Bob	Cynthia	Jackie	Ben

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

0	1	2	3	4	5	6
Alexa		Steven	Bob	Cynthia	Jackie	Ben

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.

With chaining, which we will get to in a few minutes, each entry of the table is a list of all of the items that hash to that index.

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.

Clicker Question: I want to add, in order, the following words to a hash table of size 11:

Item	hashCode%size
one	7
two	5
three	4
four	5

At what index does "four" end up?

A. 3
B. 4
C. 5
D. 6

Another: I want to add, in order, the following words to a hash table of size 11:

Item	hashCode%size
one	7
two	5
three	6
four	5

At what index does "four" end up?

A. 5
B. 6
C. 7
D. 8

Once more, this time with table of size 7:

Item	hashCode%size
one	0
two	6
three	6
four	0

At what index does "four" end up?

A. 0
B. 1
C. 2
D. 3

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, ... 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 Size. If i and j are different and both no more than Size/2, this can't happen because Size is prime. So the first 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.

So far we have been discussing Hash Tables -- data stored by hashing into a table. A HashMap is a simple extension of this. As with all maps, we have a <Key, Value> pair. Both key and value are stored in the entries of the table, at a location determined by the hashCode of the key. To look up a value in the table we do a search on the key; when we find it we return the corresponding value.