# More on Hashing: Collisions

See Chapter 20 of the text.

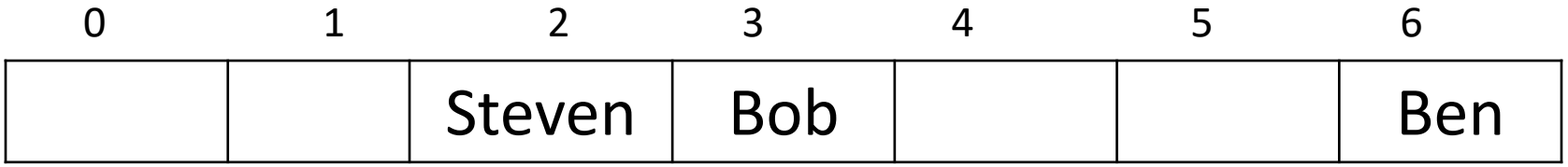
# Collisions

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

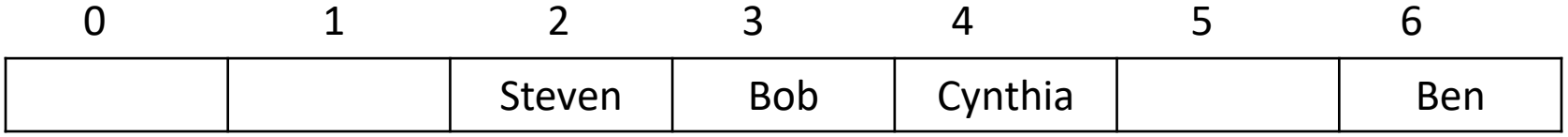
| Name | h = hash(name) | h%7 |
|---|---|---|
| Ben | 66667 | 6 |
| Bob | 66965 | 3 |
| Steven | -1808493797 | -5 -> 2 |
| Cynthia | -1392489180 | -5 -> 2 |
| Alexa | 63347171 | 6 |
| Jackie | -2083773093 | -3 -> 4 |

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.

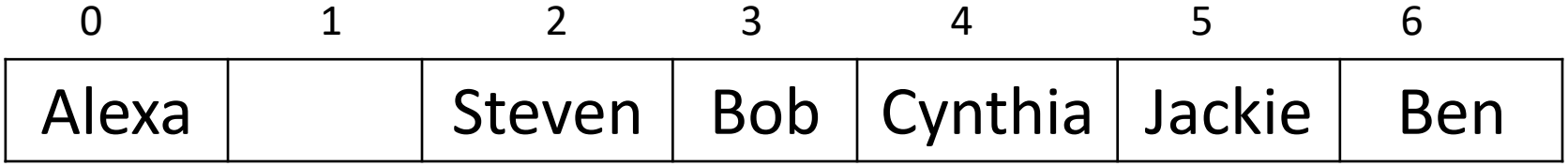
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | Steven | Bob | Cynthia | | Ben |

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.

Question 1: I want to put strings “A”, “B”, etc. into a HashTable of size 7 (so with indexes 0 through 6). Here are their hash values:

| Letter | A | B | C | D | E |
|---|---|---|---|---|---|
| Hash Value | 4 | 3 | 3 | 5 | 6 |

We add the letters to the table in alphabetical order. Find the index where each letter is stored after the collisions are resolved.

| Letter | A | B | C | D | E |
|---|---|---|---|---|---|
| Hash Value | 4 | 3 | 3 | 5 | 6 |

Answer 1:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| E | | | B | A | C | D |

Question 2, again with table of size 7 (indexed 0 through 6)

| Item | hashValue |
|---|---|
| one | 0 |
| two | 6 |
| three | 6 |
| four | 0 |

If we add the string in the order “one”, “two”, “three”, “four”, at what index does "four" end up?

Answer 2:  At index 2.

Question 3. We want to make a map where the keys are strings: letters “A”, “B”, and so forth, and the values are integers. Here are the key-value pairs:

| Letter | A | B | C | D | E |
|---|---|---|---|---|---|
| Value | 23 | 14 | 31 | 17 | 42 |

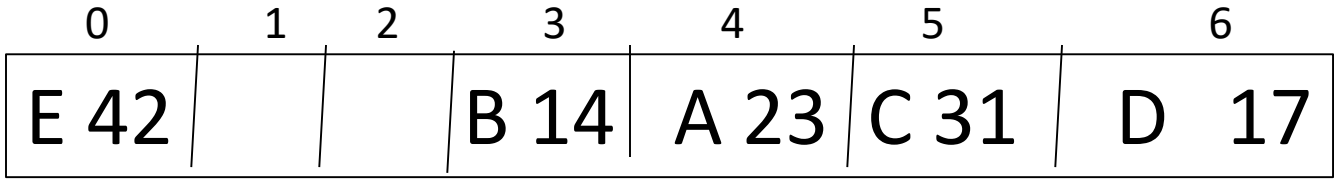
We will use the same hash values as before:

| Letter | A | B | C | D | E |
|---|---|---|---|---|---|
| Hash Value | 4 | 3 | 3 | 5 | 6 |

Again, we insert the data in alphabetical order of the keys. Draw a picture of the HashTable.

| Letter | A | B | C | D | E |
|---|---|---|---|---|---|
| Hash Value | 4 | 3 | 3 | 5 | 6 |

Answer 3:



Question 4: Class HashMap<K, V> has a method

V get(K k)

that returns the value associated with key k, or null if k is not a key. Given these hash values:

| Letter | A | B | C | D | E |
|---|---|---|---|---|---|
| Hash Value | 4 | 3 | 3 | 5 | 6 |

and this HashMap<String, Integer>:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| E 42 | | | B 14 | A 23 | C 31 | D 17 |

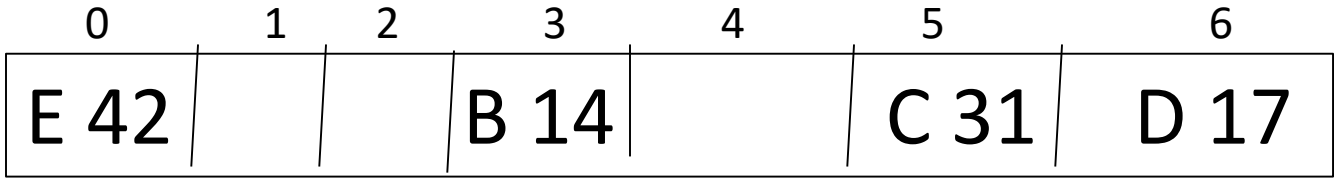
What will get("D") return?

Answer 4:  17

Question 5: Suppose we have these hash values

| Letter | A | B | C | D | E |
|---|---|---|---|---|---|
| Hash Value | 4 | 3 | 3 | 5 | 6 |

and this HashMap<String, Integer>:




what is the value of get( “C”) ?

Answer 5:  null

We start at index 3, the hash value of “C”.  That is occupied with a key that is not “C”, so we go to the next entry, at index 4.  That is not occupied, so we think that “C” is not a key in the map and return null.  We never get to the pairing of “C” with 31 at index 5.

Note that problems like this can result from deleting entries from a HashMap.