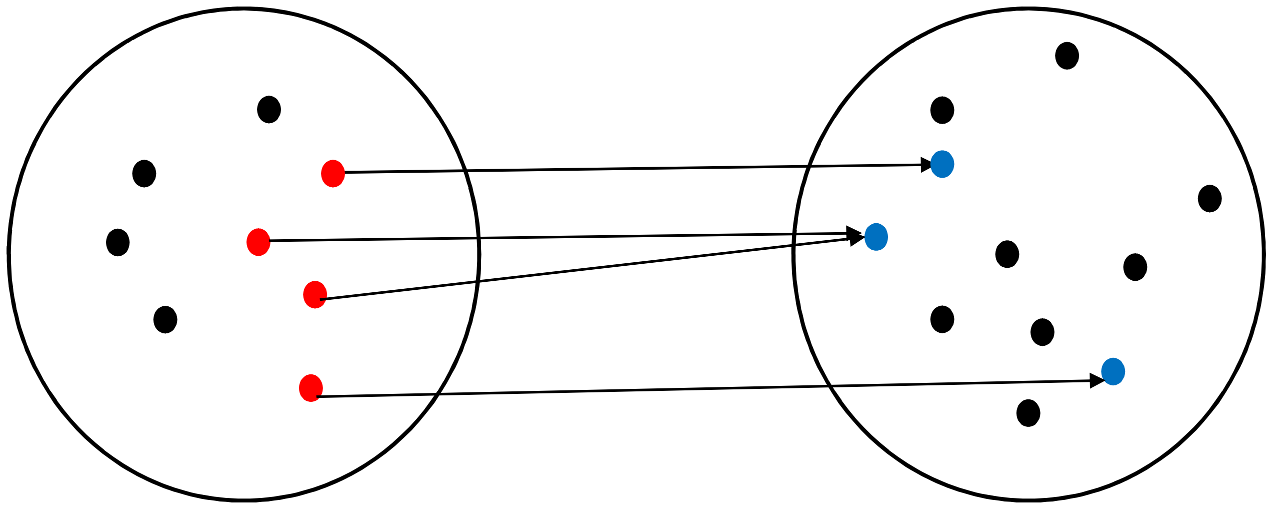
COMP 250

Lecture 30

hashing

March 22, 2022

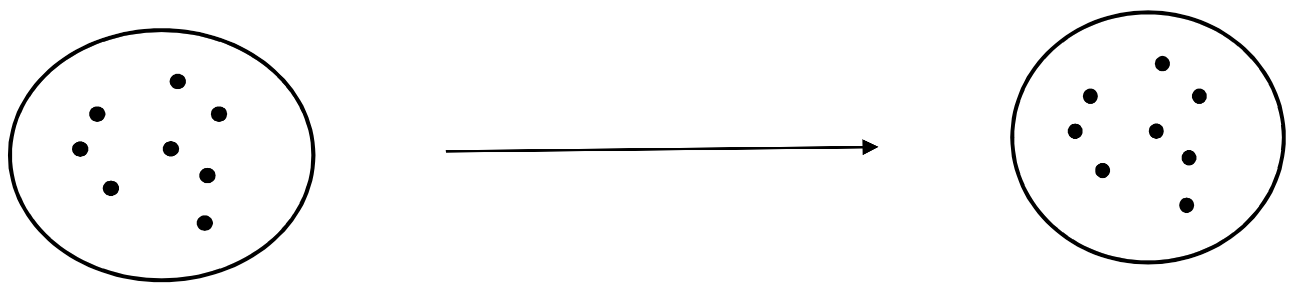
RECALL: Map



keys (type K) values (type V)

Each (key, value) pair is an "entry".
For each key, there is at most one value.

RECALL: Java `K.hashcode()`



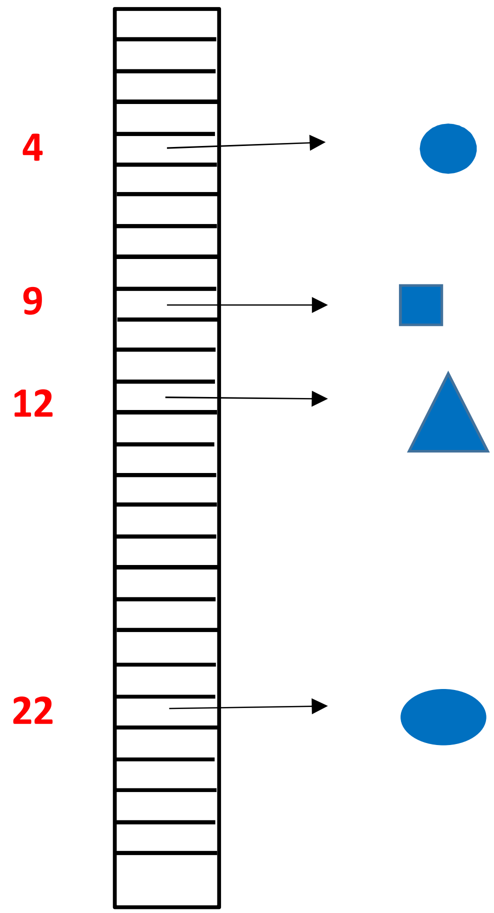
keys `K`

`int`
(32 bits)

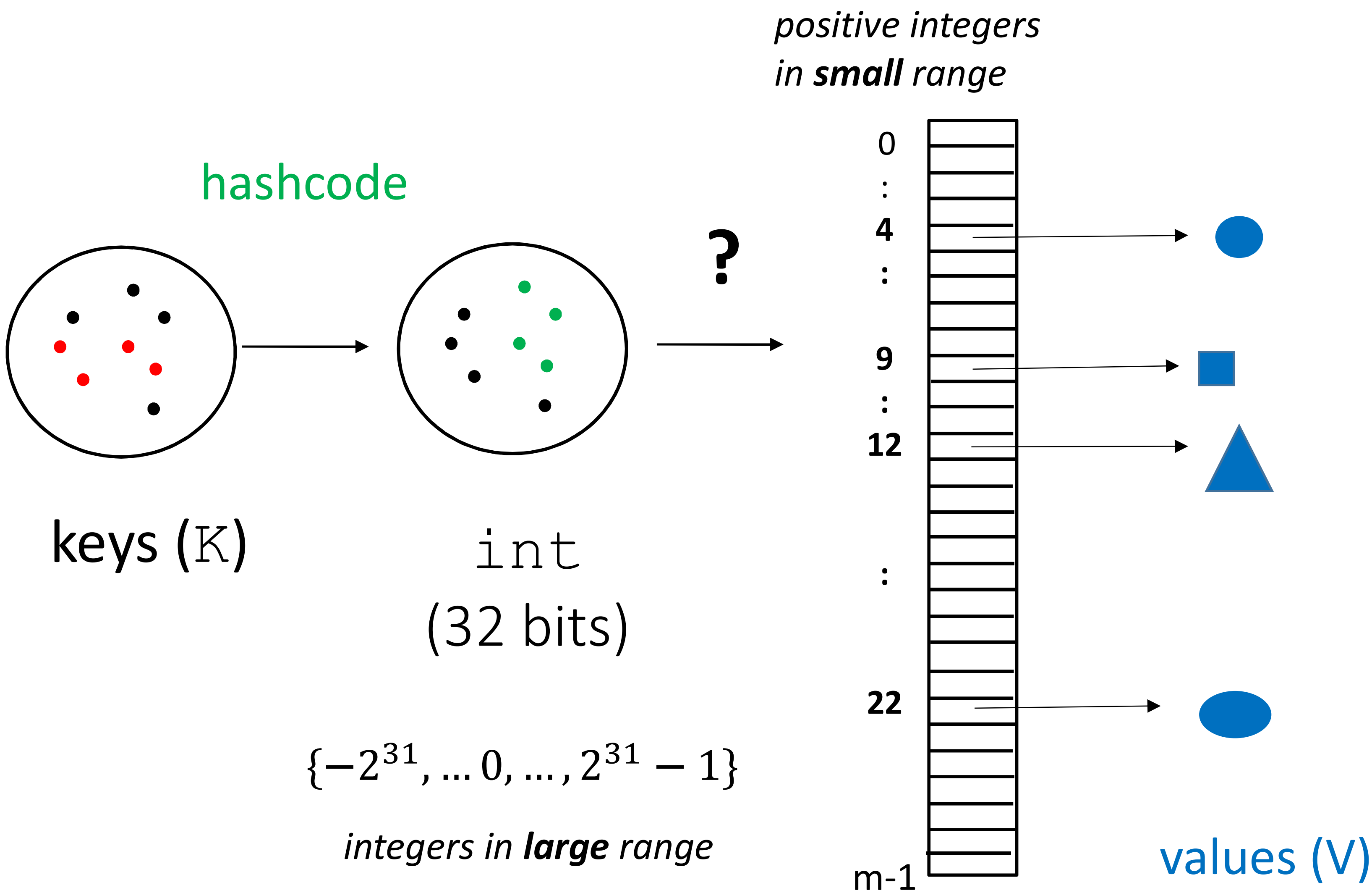
$\{-2^{31}, \ldots 0, \ldots, 2^{31} - 1\}$

integers in large range

RECALL: Special case that keys are positive integers in small range



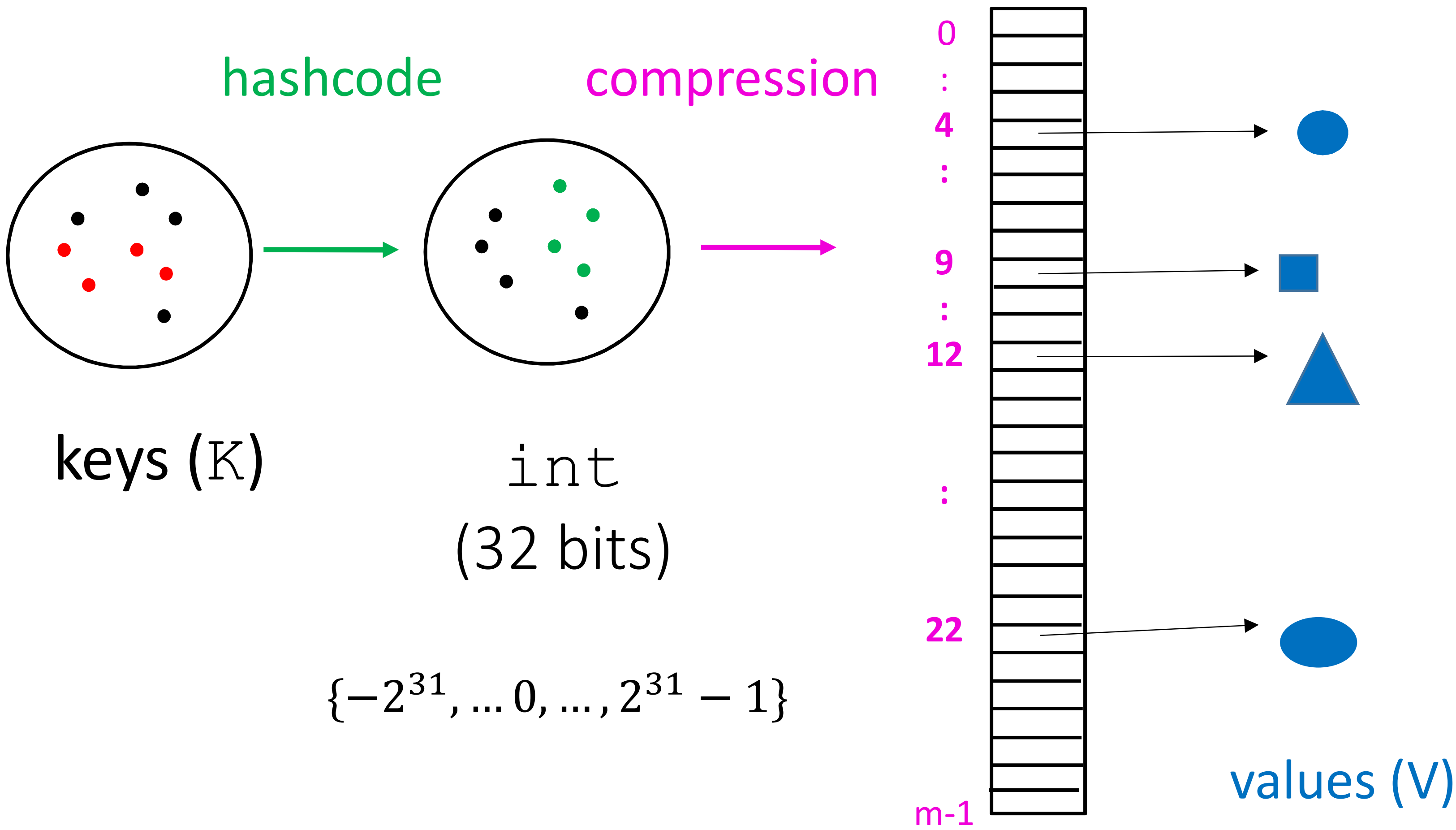
We will try to combine these two ideas as shown below....



... using a many-to-one "compression" map.

compression : $i \rightarrow |i| \; mod \;\; m,$

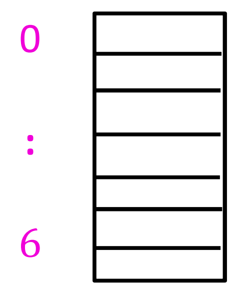
where m is the length of the array.



compression map: $i \rightarrow |i| \; mod \; m,$

Example: let $m = 7$.

<u>hash code</u>	"<u>hash value</u>"
- 41	6
16	2
25	4
21	0
- 36	1
35	0
53	4



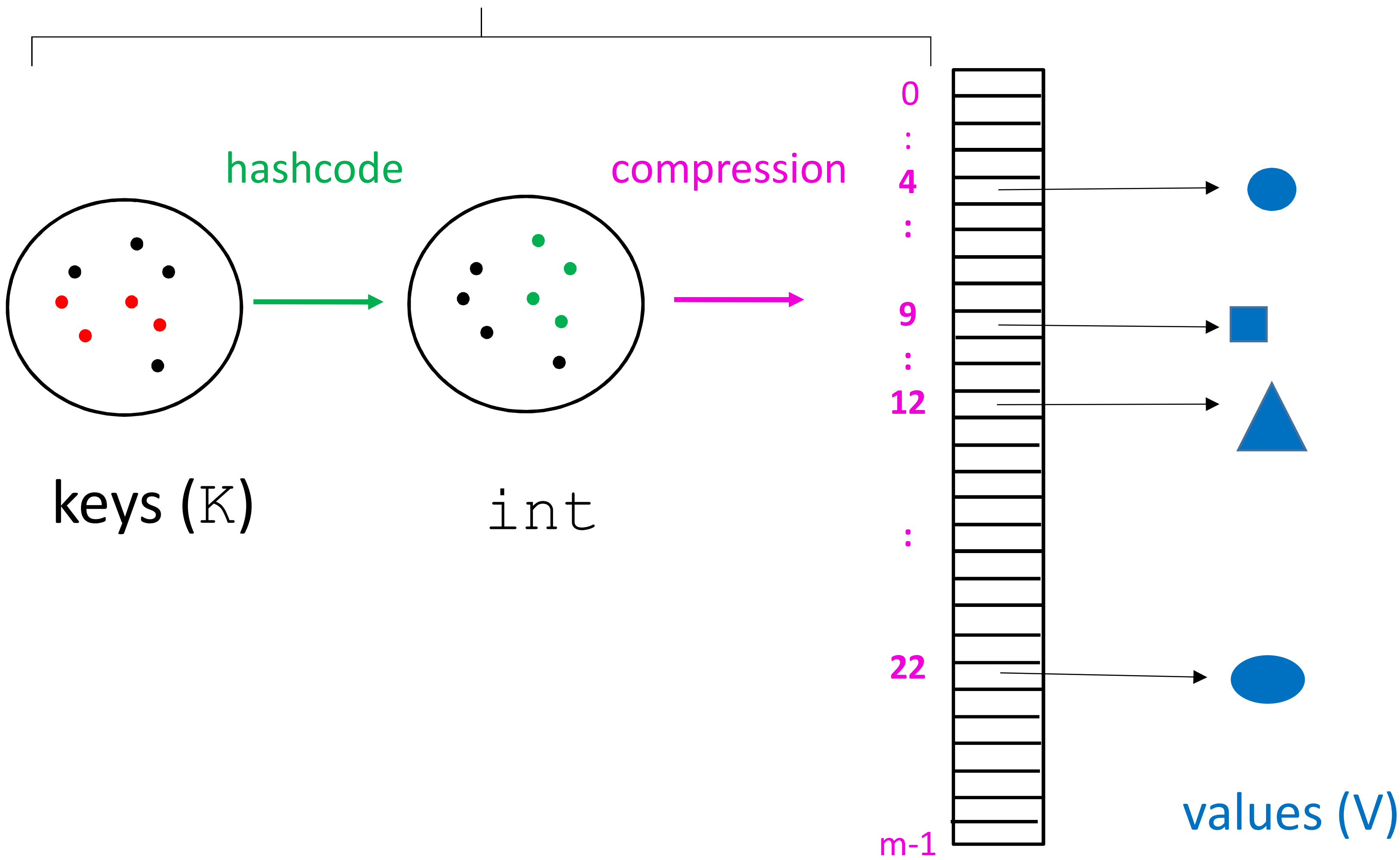
Hash function

"hash function" $\equiv$ compression ∘ hashCode

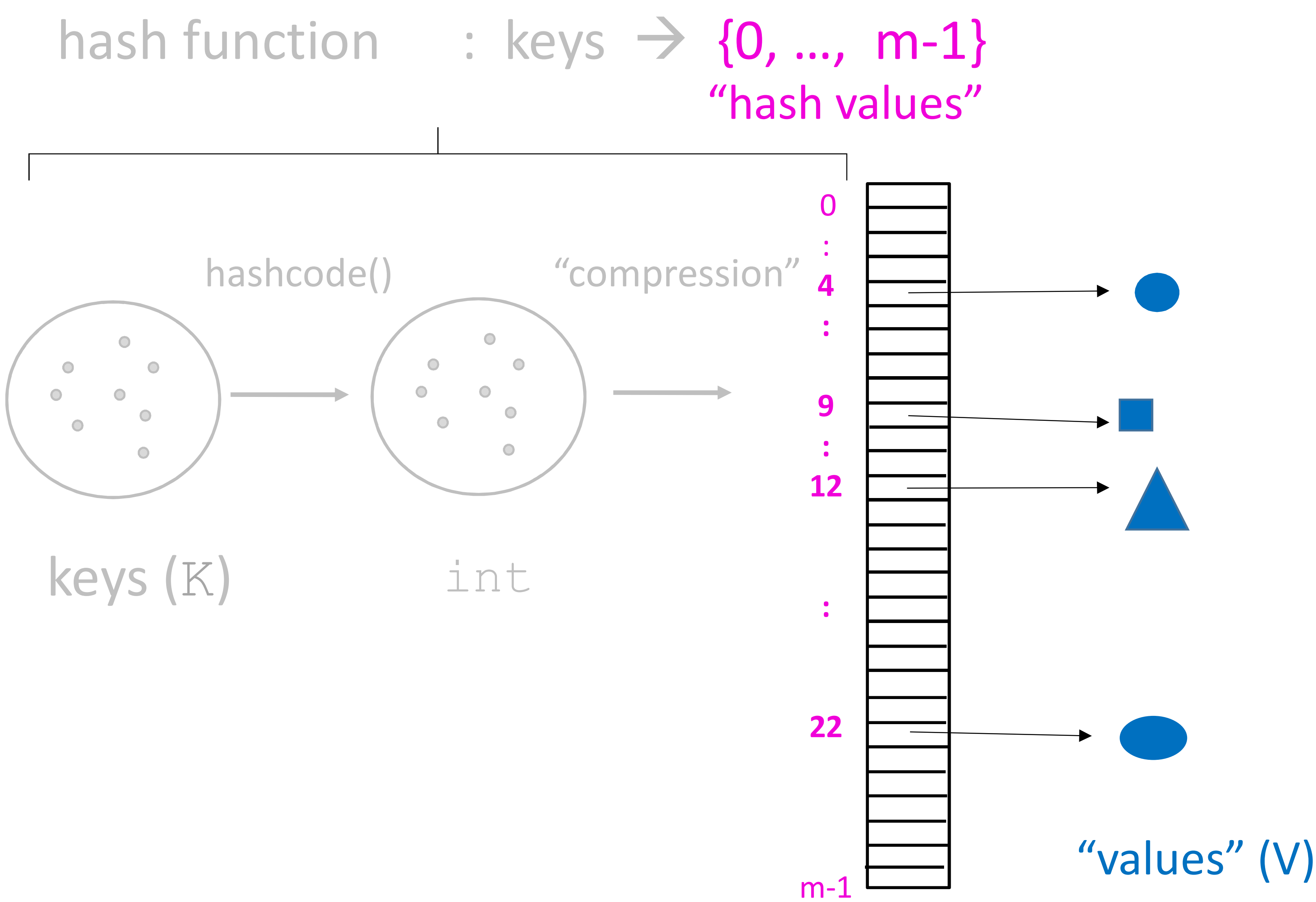
that is, hashCode followed by compression

"hash function" : keys → {0, ..., m-1}

"hash values"

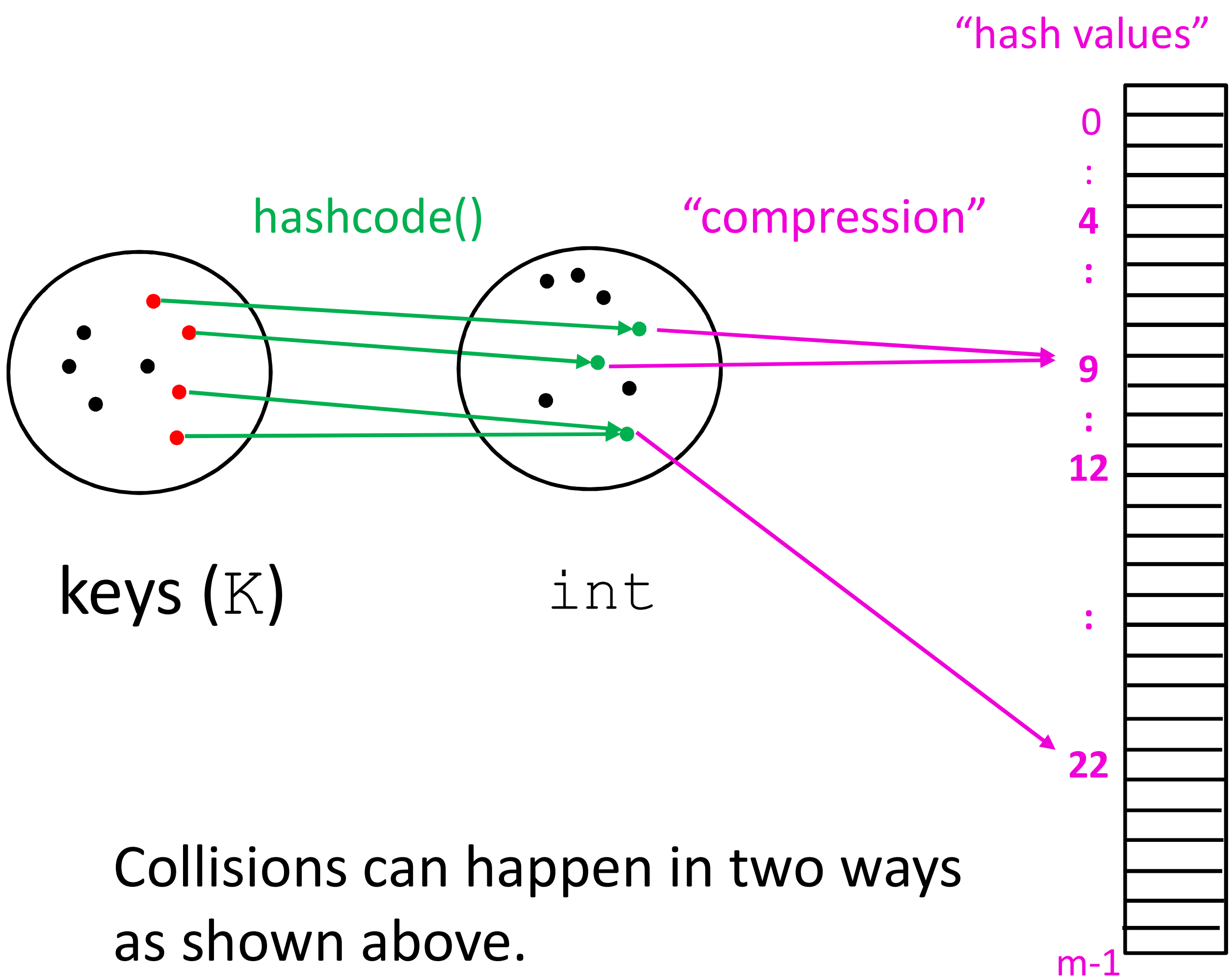


Heads up! The term “values” is used in two ways. (different maps)



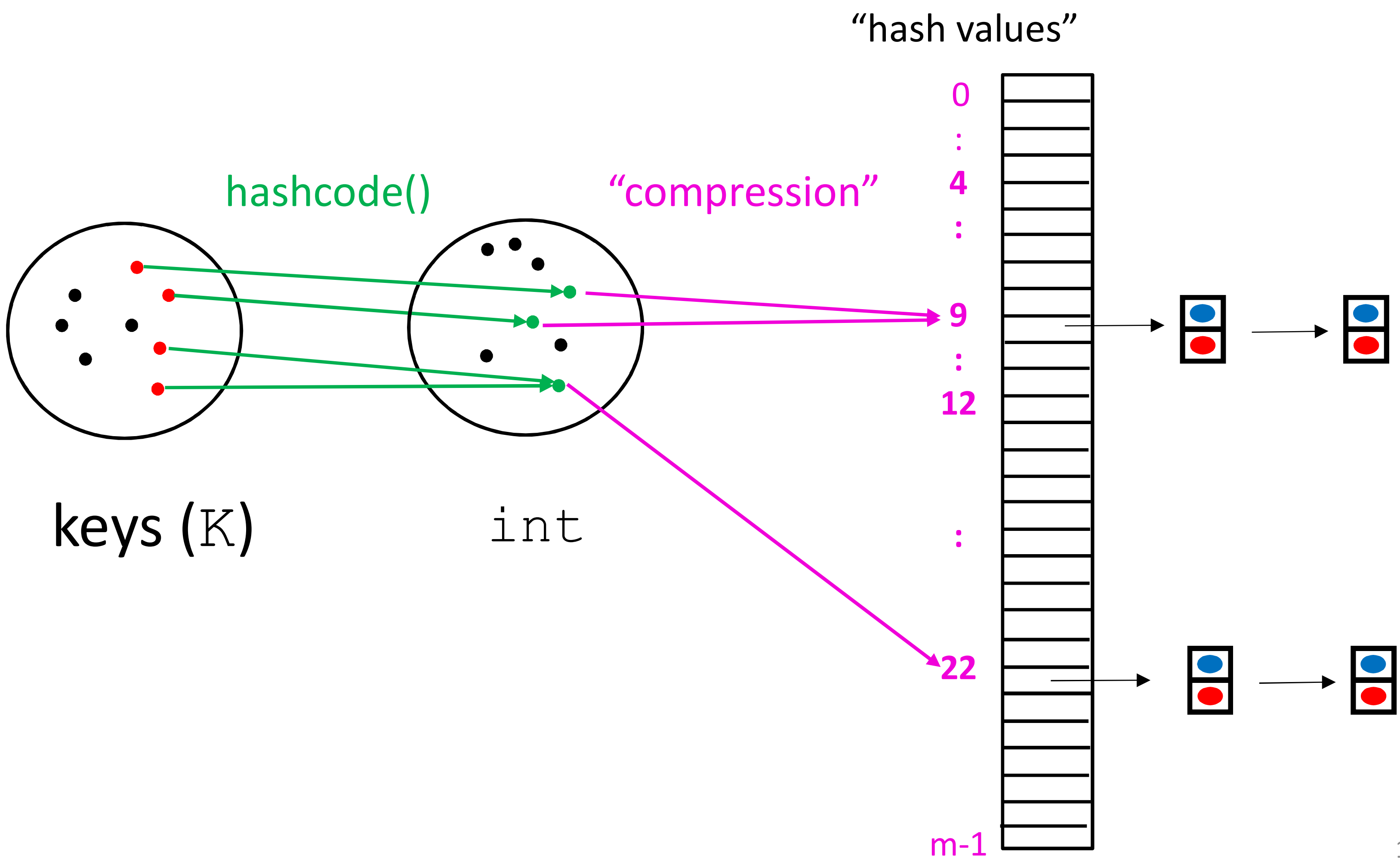
"Collisions"

(when two or more keys map to the same hash value)

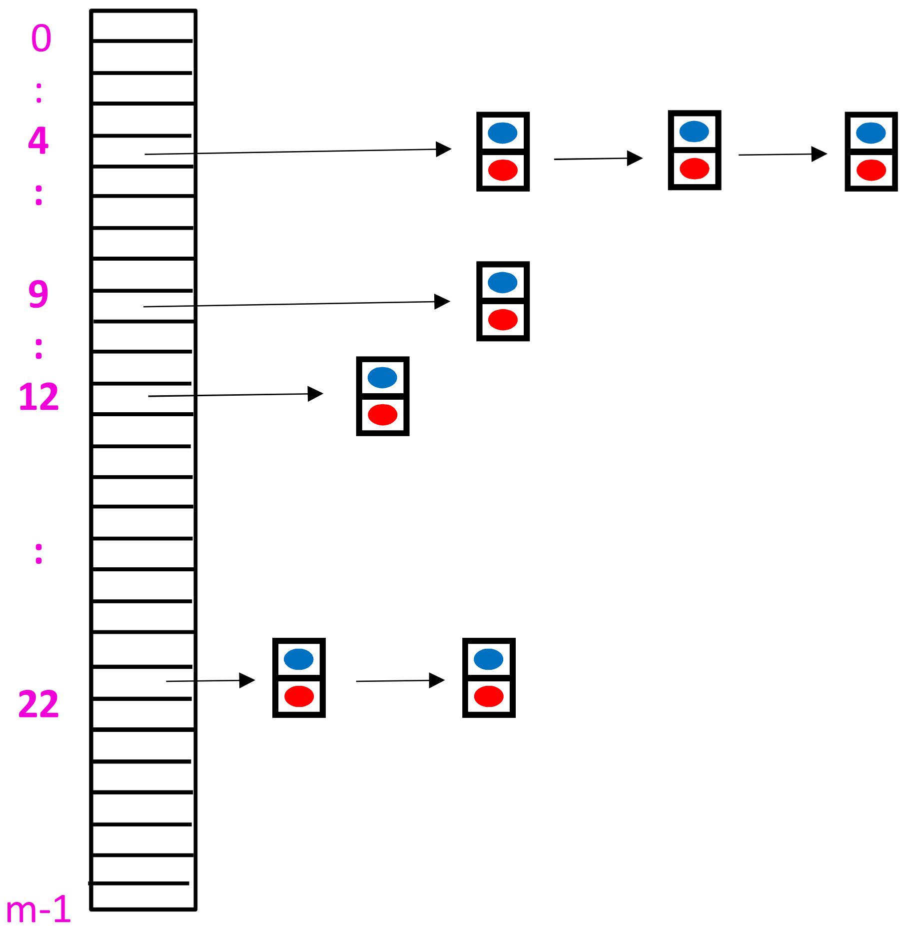


Collisions can happen in two ways as shown above.

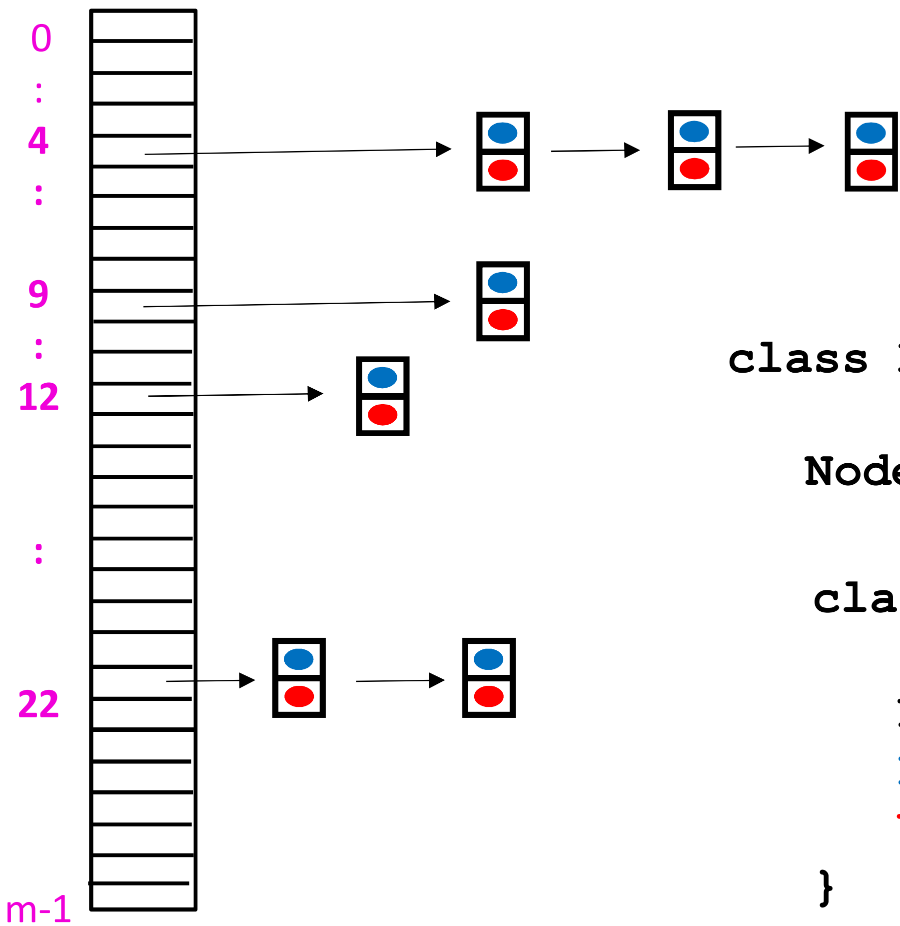
Solution to collision problem: "Hash Map" a.k.a. "Hash Table"
Each array slot holds a singly linked list of entries.
This is sometimes called "separate chaining"



Each array slot + linked list is called a **bucket**.
So there are m buckets.



How to implement this?




```
class MyHashTable<K, V > {

   Node<K, V >[]  buckets;

   class Node<K, V > {

      Node<E>     next;
      K           key;
      V           value;
   }
}
```

Why is it necessary to store (key, value) pairs in the linked list?

Why not just store the values?

Answer: Multiple keys can map to the same bucket (collisions). We need to keep track of which value corresponds to which key.

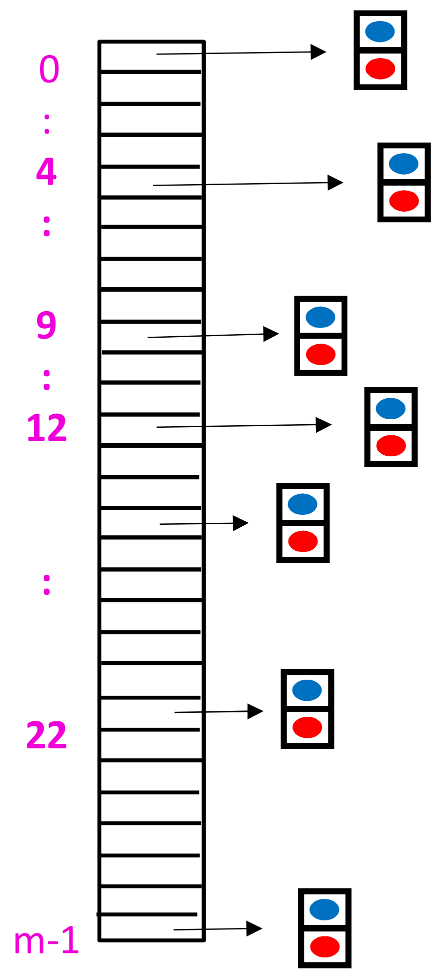
"Load factor" of a hash map

$$\equiv \frac{\text{number of entries (size)}}{\text{number of buckets (m)}}$$

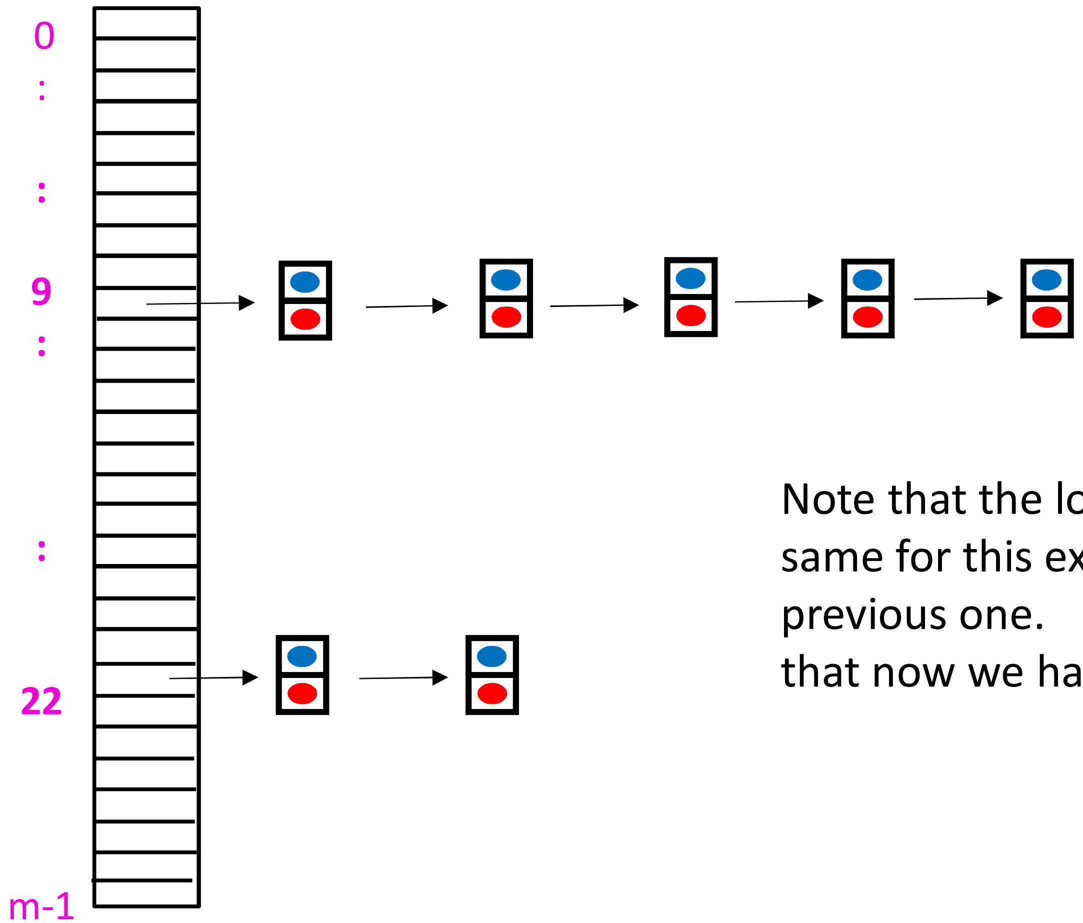
This is the average number of entries per bucket.

One typically wants the load factor to be below 1.

Example of a “good hash”



Example of a "bad hash"



Note that the load factor is the same for this example and the previous one. The issue here is that now we have a bad hash.

Example of a hash function

$$h : K \rightarrow \{0, 1, \ldots, m-1\}$$

Example: Suppose keys are McGill Student IDs,
e.g. 260745918.

How many buckets m to use ? (consider load factor)

Good hash function?

Bad hash function ?

Example of a hash function

$$h : K \rightarrow \{0, 1, \ldots, m-1\}$$

Example: Suppose keys are McGill Student IDs,

e.g. 260745918.

How many buckets m to use ? 100,000 (why?)

Good hash function?

Bad hash function ?

Example of a hash function

$$h : K \rightarrow \{0, 1, \ldots, m-1\}$$

Example: Suppose keys are McGill Student IDs, e.g. 260745918.

How many buckets m to use ? 100,000

Good hash function? rightmost 5 digits

Bad hash function ?

Example of a hash function

$h : K \rightarrow \{0, 1, \ldots, m-1\}$

Example: Suppose keys are McGill Student IDs,
e.g. 260745918.

How many buckets m to use ? 100,000

Good hash function? rightmost 5 digits

Bad hash function ? leftmost 5 digits

Performance of Hash Maps

- put(key, value)
- get(key)
- remove(key)

If load factor is less than 1 and if hash function is good, then these *operations are O(1) "in practice". This beats all potential map data structures that I considered last lecture !*

If we have a bad hash, we can choose a different hash function.

Performance of Hash Maps

- put(key, value)
- get(key)
- remove(key)
- contains(value)

In worst case, contains(value) will need to search through each of the m buckets i.e. search the linkedlists.

If there are n entries in total, then it will need to check each entry.

Performance of Hash Maps

- put(key, value)
- get(key)
- remove(key)
- contains(value)
- getKeys()
- getValues()

These last three methods all require traversing the hash table. This takes time $O(n+m)$ where n is number of entries and m is the number of buckets.

Java `HashMap<K,V>` class

- In the constructor, you can specify initial number m of buckets, and maximum load factor

(default initialization m = 16, and max load factor = .75)

- The hash function uses the key's hashCode() and compression

$$i \rightarrow |i| \ mod \ m,$$

Comparing keys with `K.equals()`

Recall that `put(K,V)`, `get(K)`, `remove(K)` all check if the key is already present in the map. This requires `K.equals()`.

Q: What should be the relationship between `K.equals()` and `K.hashCode()` ?

Hint: what should happen when you call `put(k1, v)` and later call `get(k2)` where `k1.equals(k2)` is true ?

A: If `k1.equals(k2)` is true, then we want `k1.hashCode() == k2.hashCode()` to be true.

Note that the converse doesn't hold: if two keys (e.g. strings) have equal hashCodes, then we cannot expect these keys to be equal.

Java `HashSet<E>` class

Similar to `HashMap<K,V>`, but there are no values.

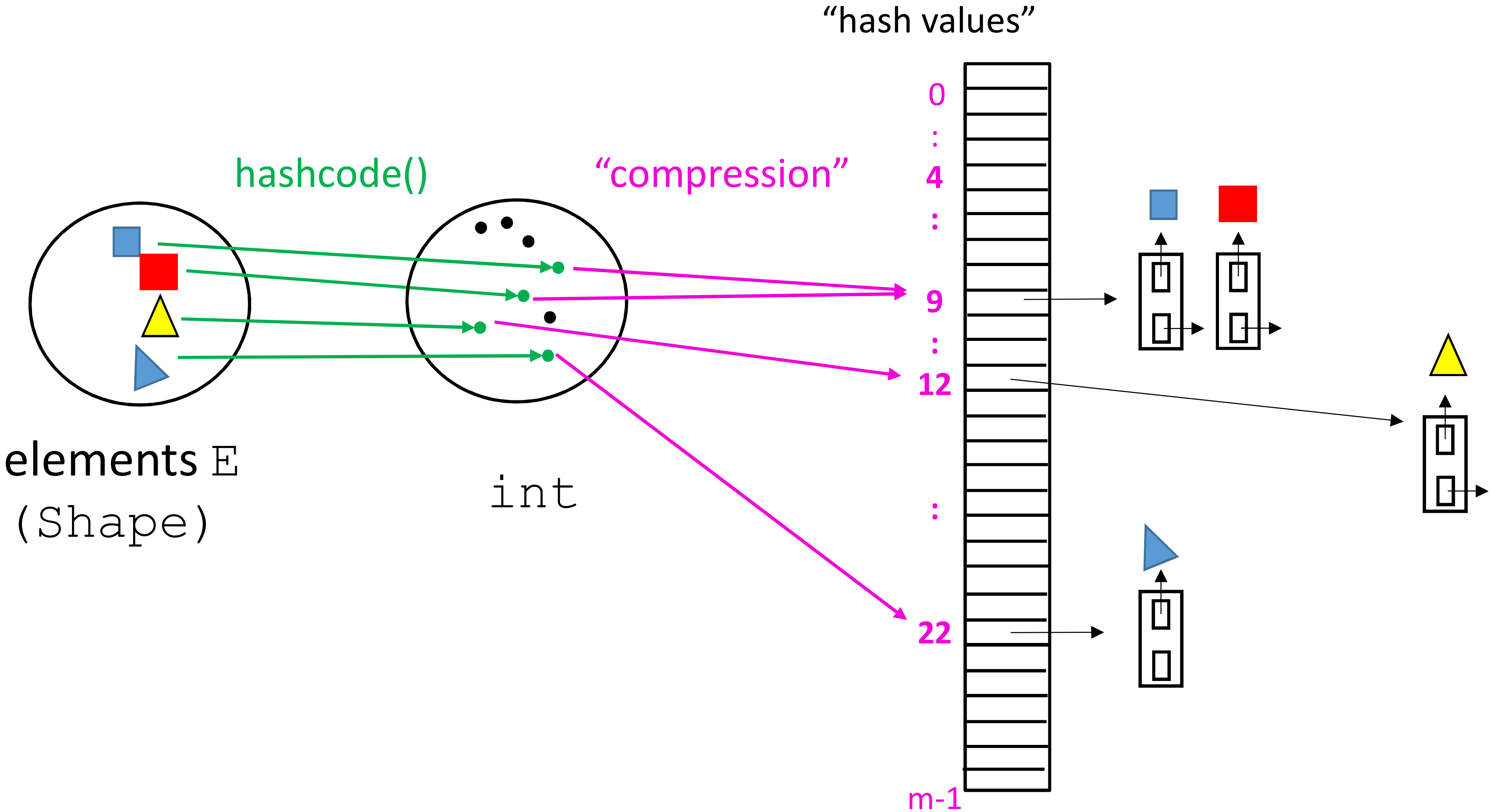
Use it to store a *set* of objects of some type.

- add(e)
- contains(e)
- remove(e)
- ……

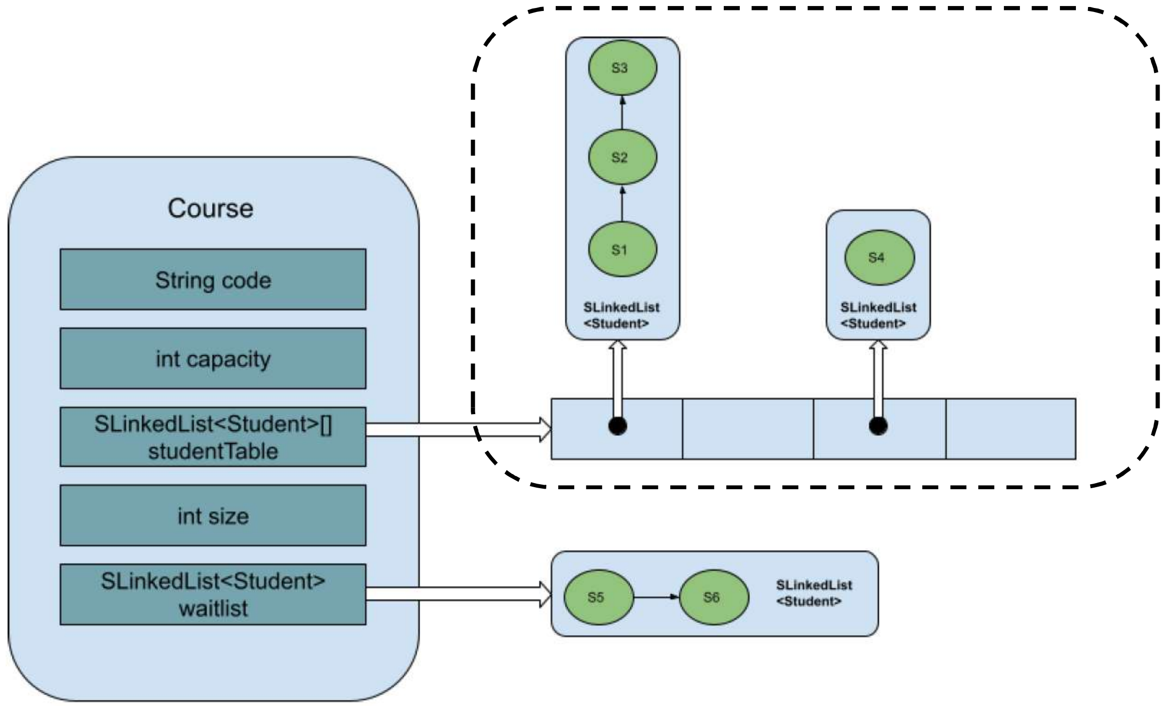
If hash function is good, then these operations are O(1).

Note that a `HashSet` is not a list. There is no 1^{st}, 2^{nd}, …. element.

Java `HashSet<E>` class



Recall Assignment 1



`SLinkedList<Student>[ ]  studentTable`

This was a HashSet. The hashCode was the student ID which was a field in `Student`.

ASIDE: Java classes

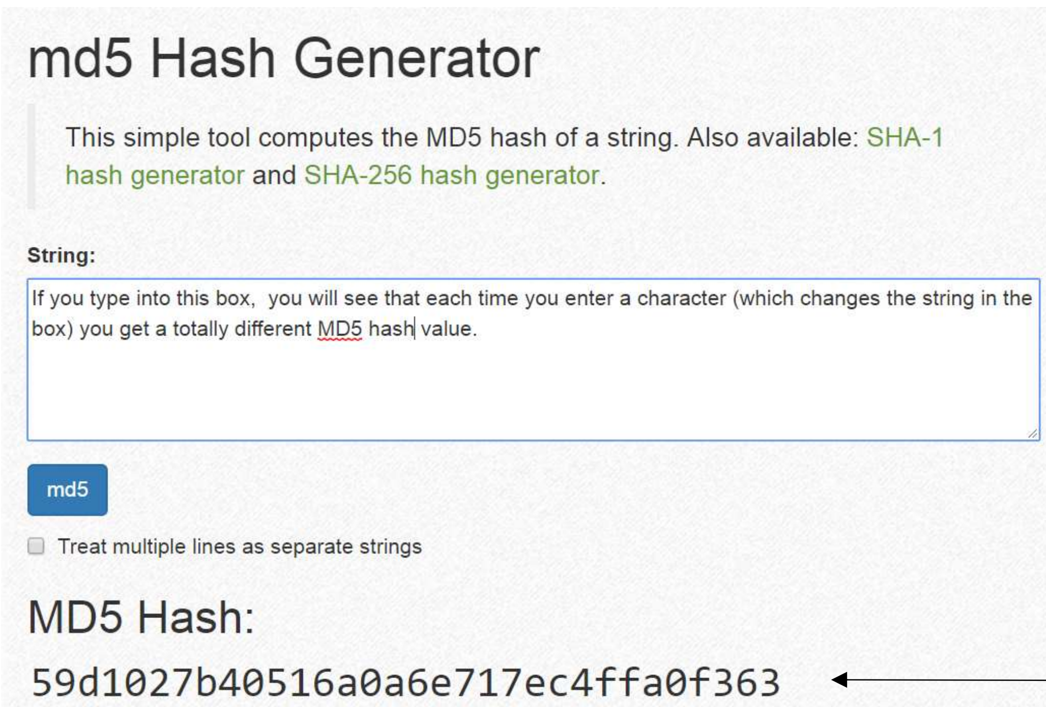
- `HashMap<K,V>` implements the interface `Map<K,V>`
- `HashTable<K,V>` implements the interface `Map<K,V>`
  (similar to `HashMap`)
- `TreeMap<K,V>` implements the interface `Map<K,V>`
  (uses a binary search tree for the keys → requires that keys are Comparable)
- `HashSet<E>` implements the interface `Set<E>`
- ...

Cryptographic Hashing (time permitting)

h: key (String) → hash value (e.g. 128 bits)

e.g. [online tool for computing md5 hash of a string](http://www.miraclesalad.com/webtools/md5.php)

http://www.miraclesalad.com/webtools/md5.php



32 hexadecimal digits (128 bits)

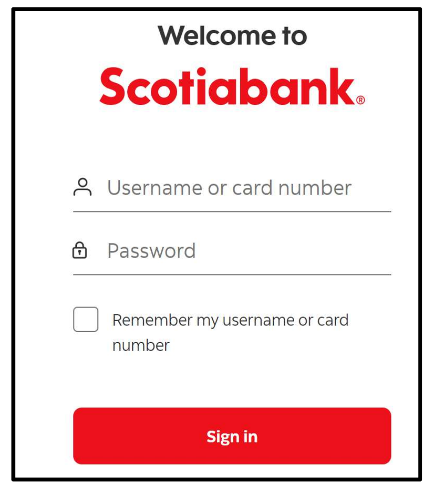
Application: Password Authentication

e.g.   Web page (server) needs to authenticate users.

{ (userID, password) }   defines a *map*.

Keys are Strings   (userID)

Values are Strings   (password)



Password Authentication (unsecure)

Suppose the {(userID, password)} map is stored in a *text* file on the web server where user logs in.

What would the user do to log in?

Enter username (key) and password (value).

What would the web server do?

Check if this entry matches what is stored in the map.

What could a mischievous hacker do?

Steal the text file, and then login to user accounts.

Password Authentication (secure)

The {(username, h(password)) } map is stored in a file on the web server.

What would the user do?

Enter a username and password.

What would the web server do ?

Hash the password, *throw away the password,* and compare the hashed password to that of the entry in map .

What could a mischievous hacker try to do?

Steal the text file. For some user name, guess the password:

“Brute force” or “dictionary” attack.

Cryptographic Hashing

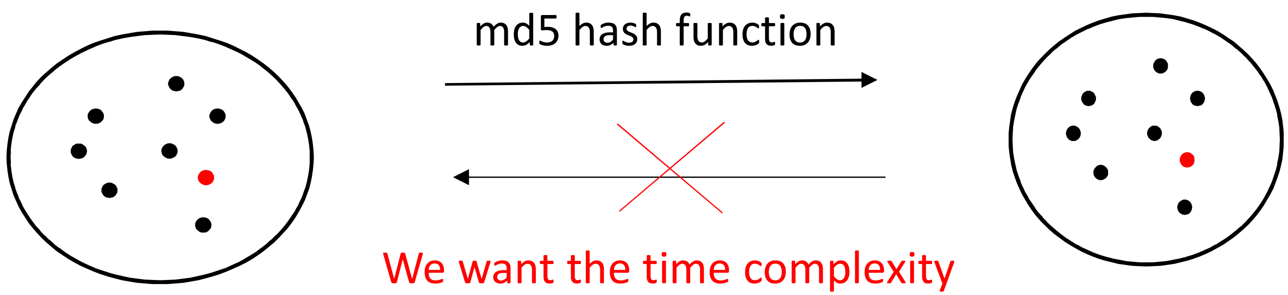
We want a hash function h(password), e.g. md5, such that one can infer almost nothing about the password from h(password).

Small changes in the key give very different hash values.

key (password)

hash value
(md5 hashed password)

md5 hash function

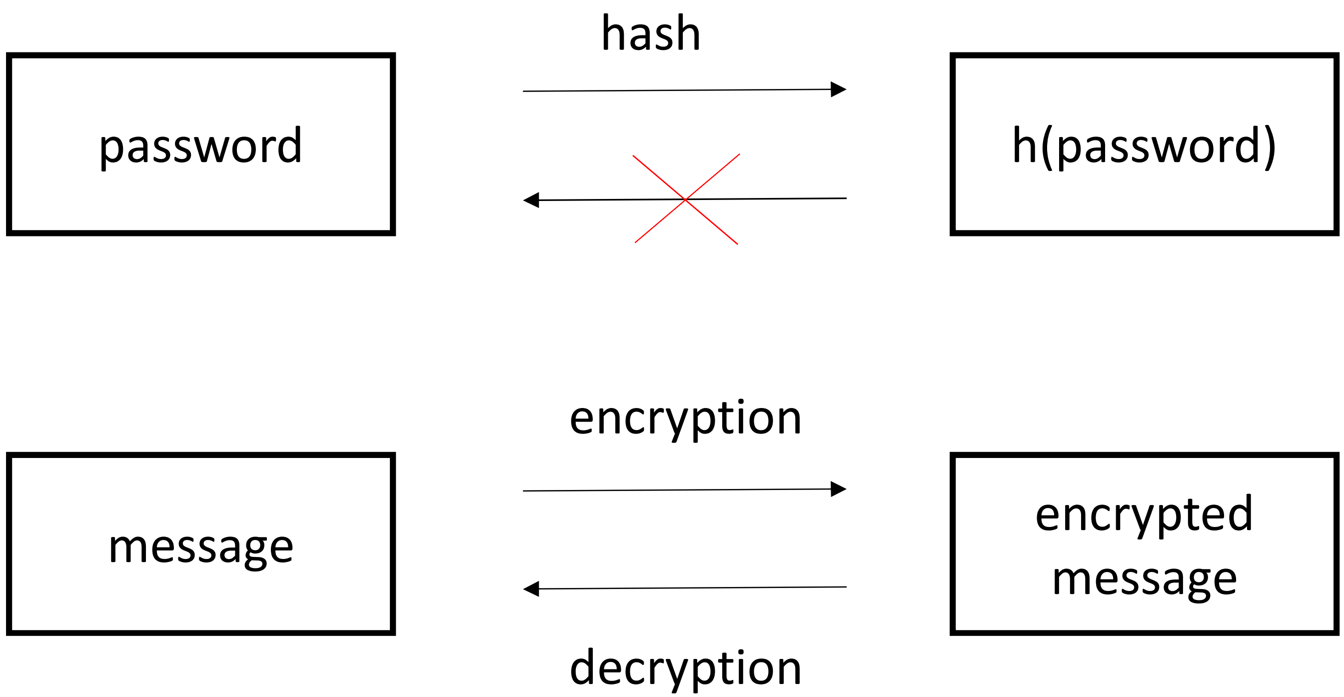


We want the time complexity of finding *some* key that maps to a given a hash value to be very high.

All strings

128 bit strings

[ASIDE: Do not confuse hashing with (RSA) encryption/decryption.]



You learn about RSA encryption in MATH 240 Discrete Structures and COMP 547 Cryptography and Data Security.

Coming up...

Lectures

Fri. March 25, Mon. March 28

Graphs 1

Wed March 30 ...

big O

Assessments

Quiz 5 is Mon. April 4.

Assignment 4 due Wed. April 6.