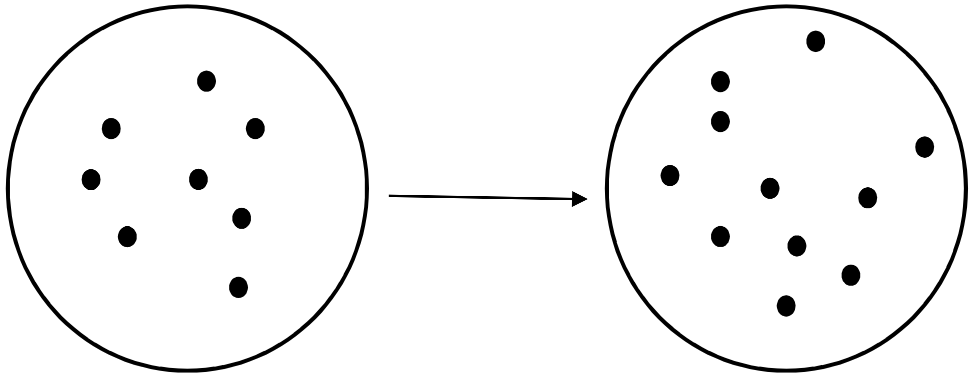
COMP 250

Lecture 29

maps

March 21, 2022

Map (Mathematics)



“domain” “codomain”

A map is a set of pairs { (x, f(x)) }.

Each x in domain maps to some f(x) in codomain.

Math examples

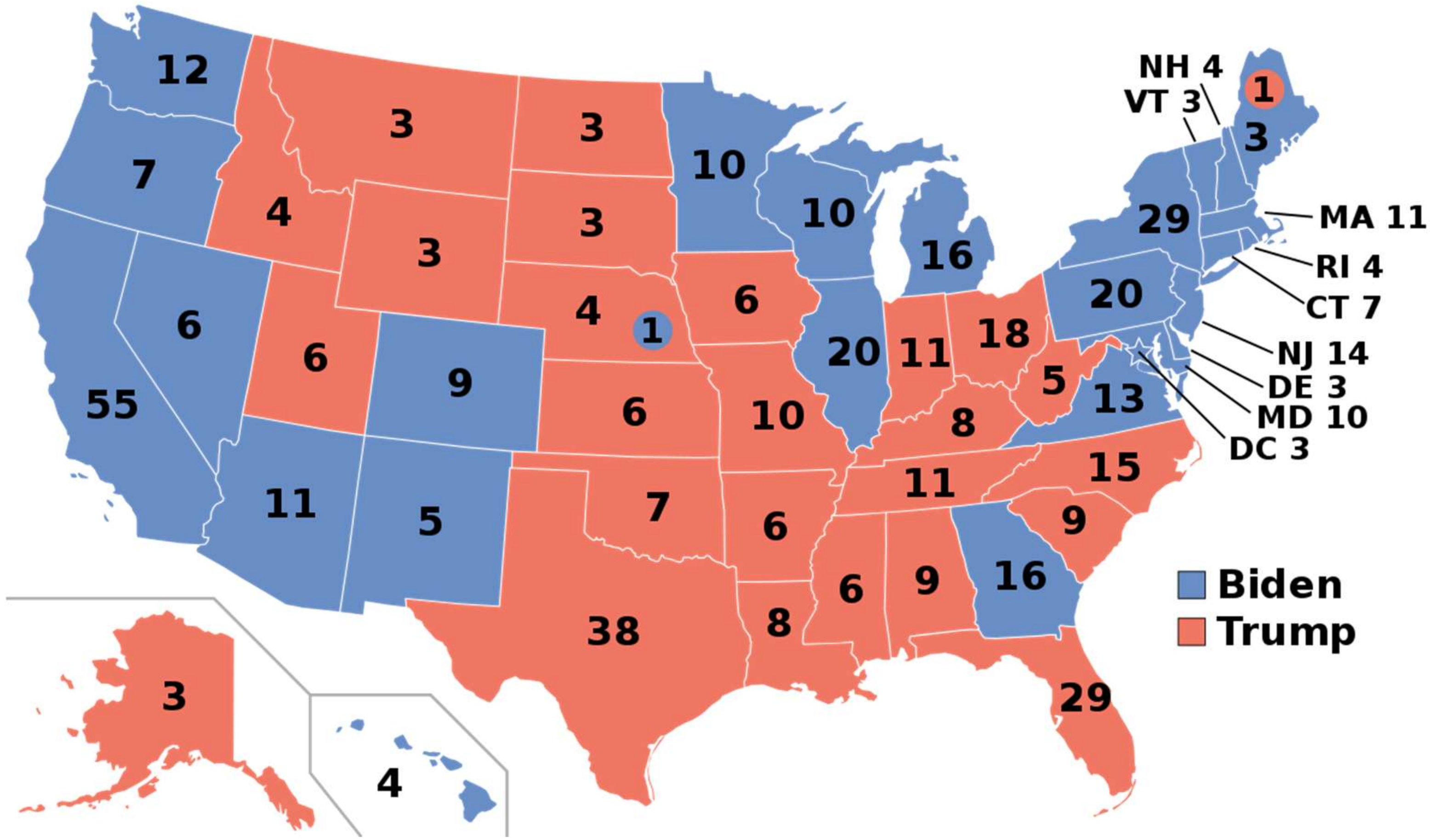
Calculus 1 and 2 (“functions”):

$$f(x): \quad \Re^n \rightarrow \Re^m$$

Maps in everyday life

map(x,y) : position in 2D image → 2D position in Montreal

Color map



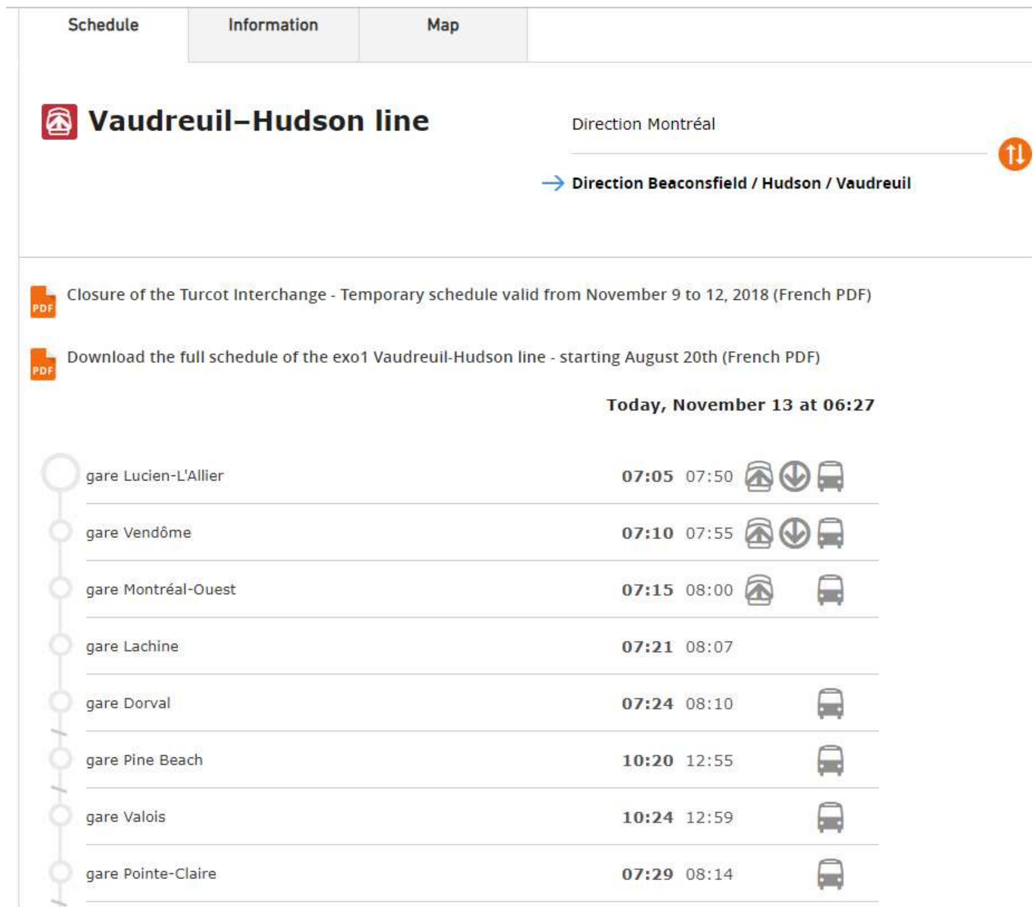
vote_result : US_state → { blue(Dem), red(Rep) }

Restaurant Menu

Poulet au cari vert & lait de coco Chicken in green curry & coconut milk	18.95
Poulet au cari jaune & lait de coco Chicken in yellow curry & coconut milk	18.95
Poulet au cari paneang & lait de coco Chicken in paneang curry & coconut milk	18.95
Poulet sauté aux oignons et piments forts Chicken sautéed with onions & chillies	18.95
Poulet sauté au basilic thaïlandais Chicken sautéed with thai basil	18.95
Poulet sauté aux noix de cajou Chicken sautéed with cashew nuts	18.95
Poulet sauté aux aubergines Chicken sautéed with eggplants	18.95
Poulet sauté aux haricots verts Chicken sautéed with green beans	18.95
Poulet sauté aux pousses de bambou Chicken sautéed with bamboo shoots	18.95
Poulet sauté au brocoli & sauce aux huîtres Chicken sautéed with broccoli & oyster sauce	18.95

menu : dish_name → price

Train Schedule

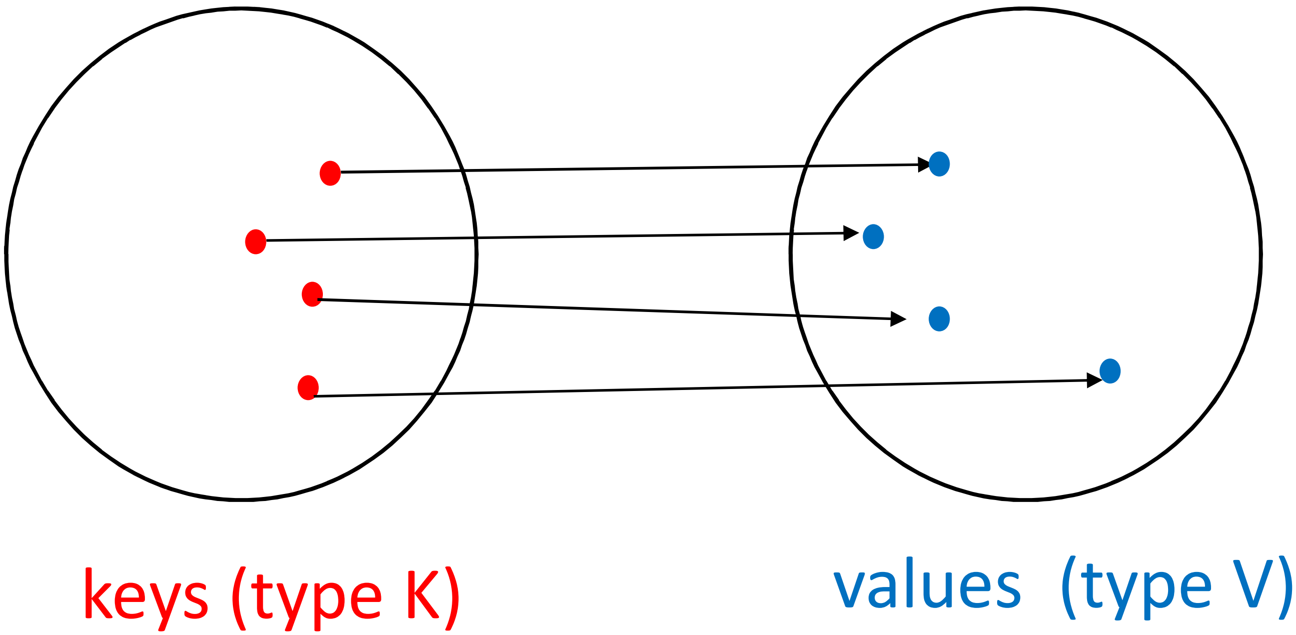


Schedule : station → time of next train (or list of times)

Index in a book

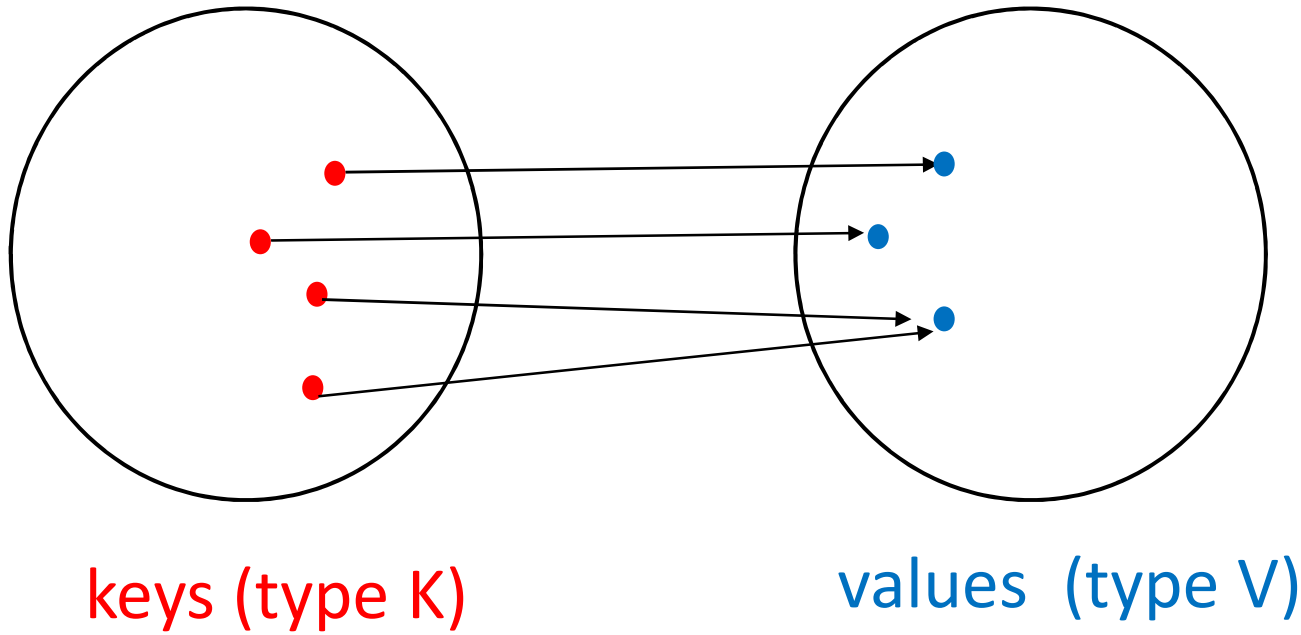
index : term → list of pages containing term

Map (ADT)



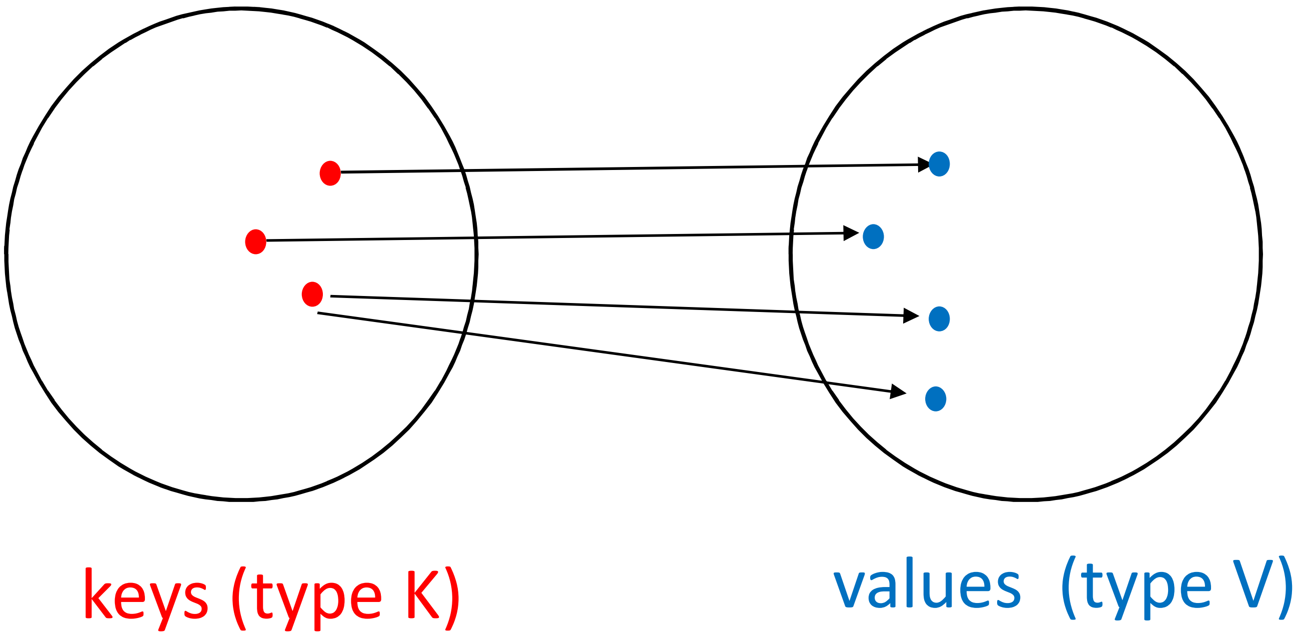
A map is a set of (key, value) pairs.
For each key, there is at *most one* value.

Map (ADT)



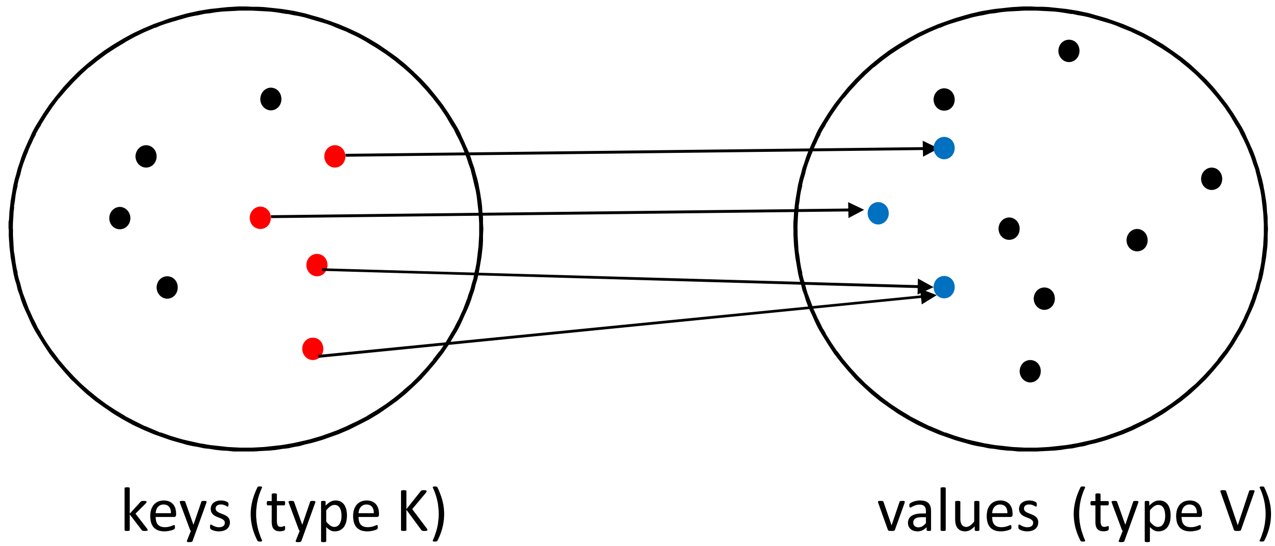
Two keys can map to the same value.

Map (ADT)



It is NOT allowed that one key maps to two different values.
The above example is NOT a map.

Map Entry

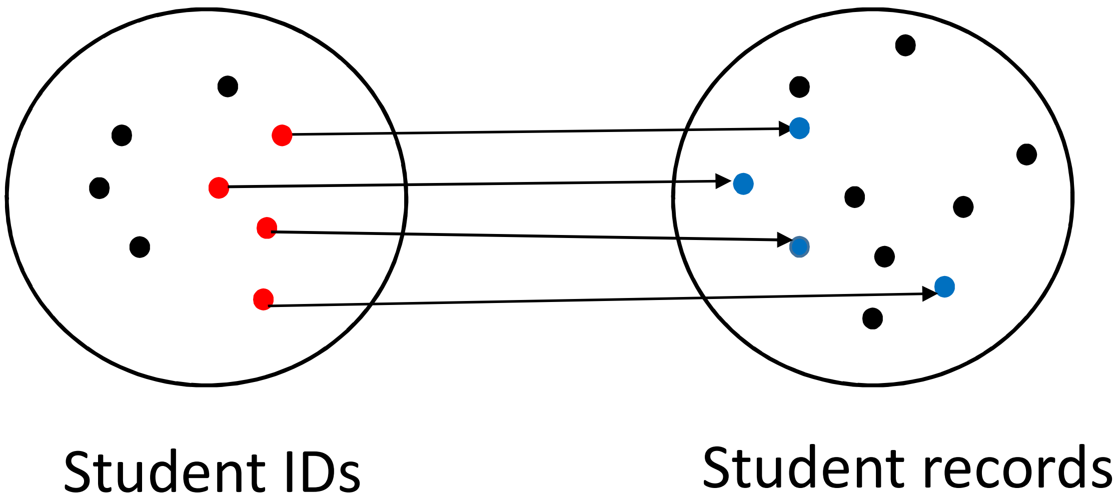


Each (key, value) pair is called an *entry*.

In this example, there are four entries.

The black dots here indicate keys or values that are *not* in the map.

Example



In COMP 250 this semester, the above mapping has ~600 entries.

Most McGill students are not taking COMP 250 this semester.

BTW, the student ID can also be part of the student record.

Map ADT

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

Map ADT

- put(key, value)

If the map previously contained a mapping for the key, then the old value is replaced by the specified value, and the previous value is returned. Otherwise, return null.

- get(key)
- remove(key)
- …

Map ADT

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

Returns the value to which the specified key is mapped, or return null if this map contains no entry for the key.

- remove(key)
- …

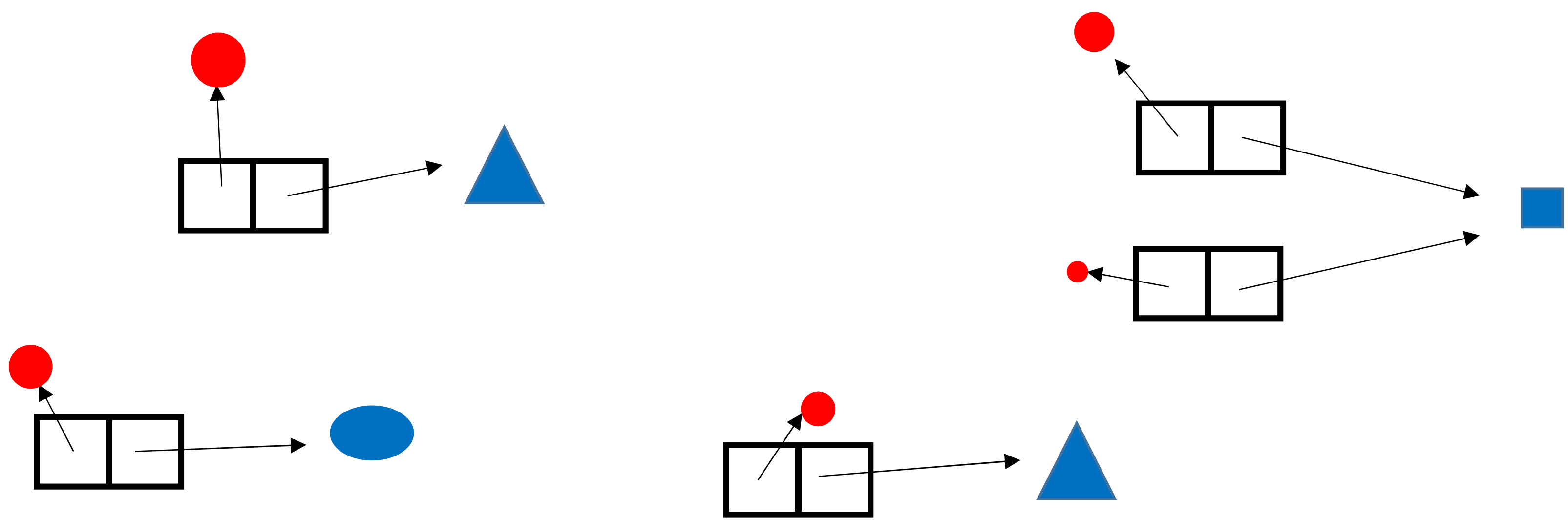
Map ADT

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

  Removes the entry for the key, if it is present, and returns the value.   Returns null if the map contains no mapping for the key.
- …

About the figures....

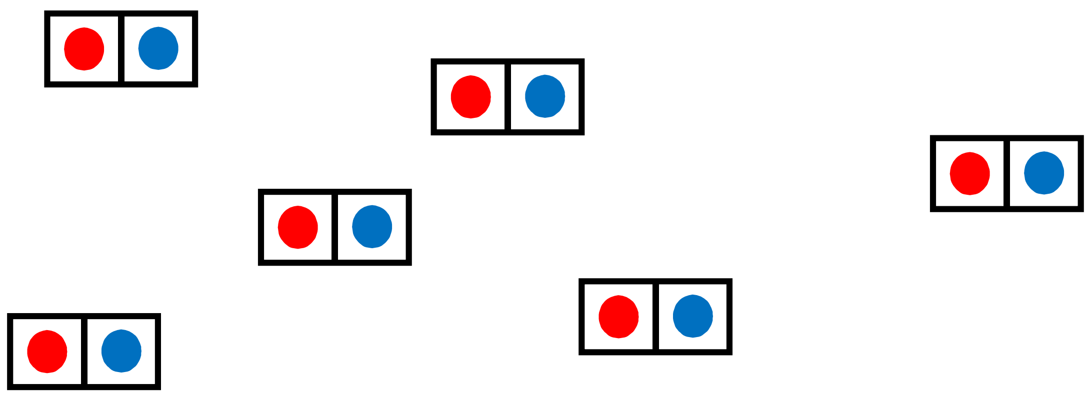
When programming with maps in Java, keys and value variables are *reference* types. On this slide, the keys as different sized red disks and the values are blue shapes.



In this example, two of the keys map to the same value.

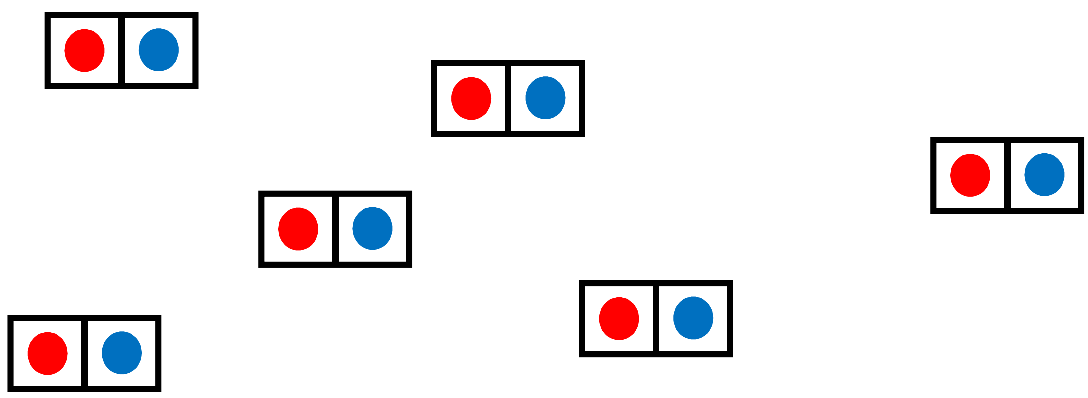
About the figures....

For the remaining slides today, I will draw a set of (key, value) pairs, i.e. entries, as shown below.
But try to keep the previous slide in mind...

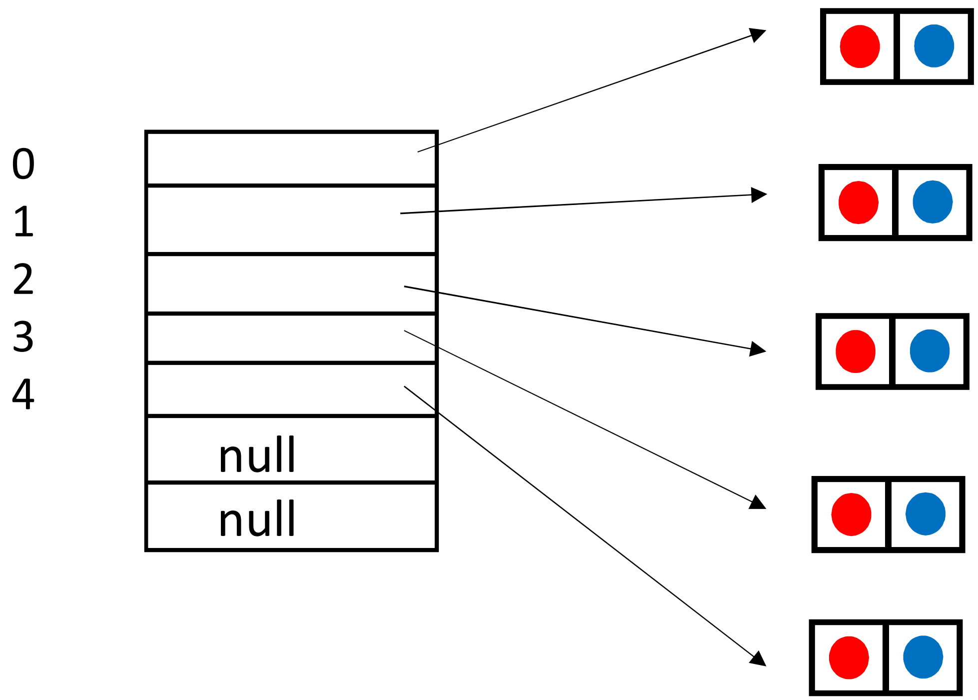


Data Structures for Maps ?

How to organize a set of (key, value) pairs, i.e. entries ?



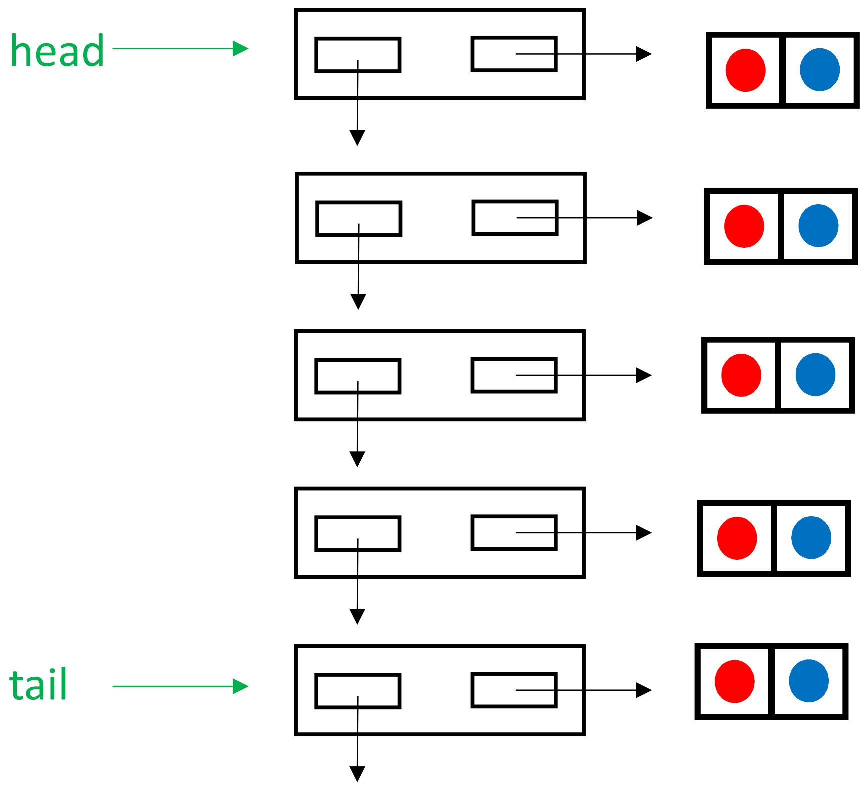
Array list



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

How would you implement these operations?
What are the best and worst case time complexities ?

Singly (or Doubly) linked list



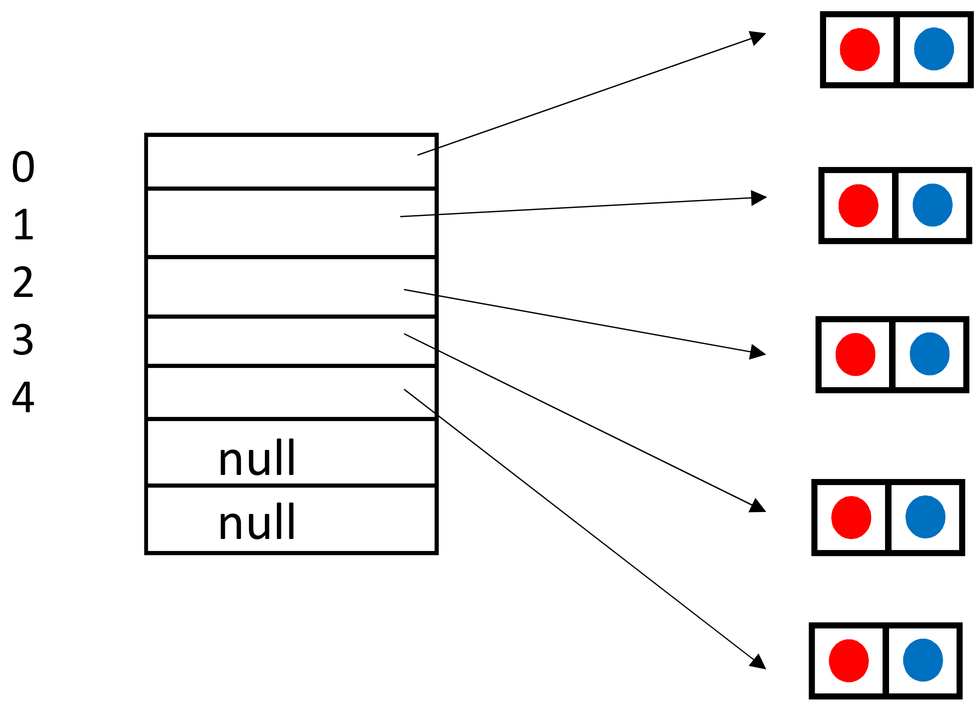
put(key, value)
get(key)
remove(key)

How would you implement these operations?
What are the best and worst case time complexities ?

Special case #1: what if keys are *comparable* ?

Can we take advantage of this?

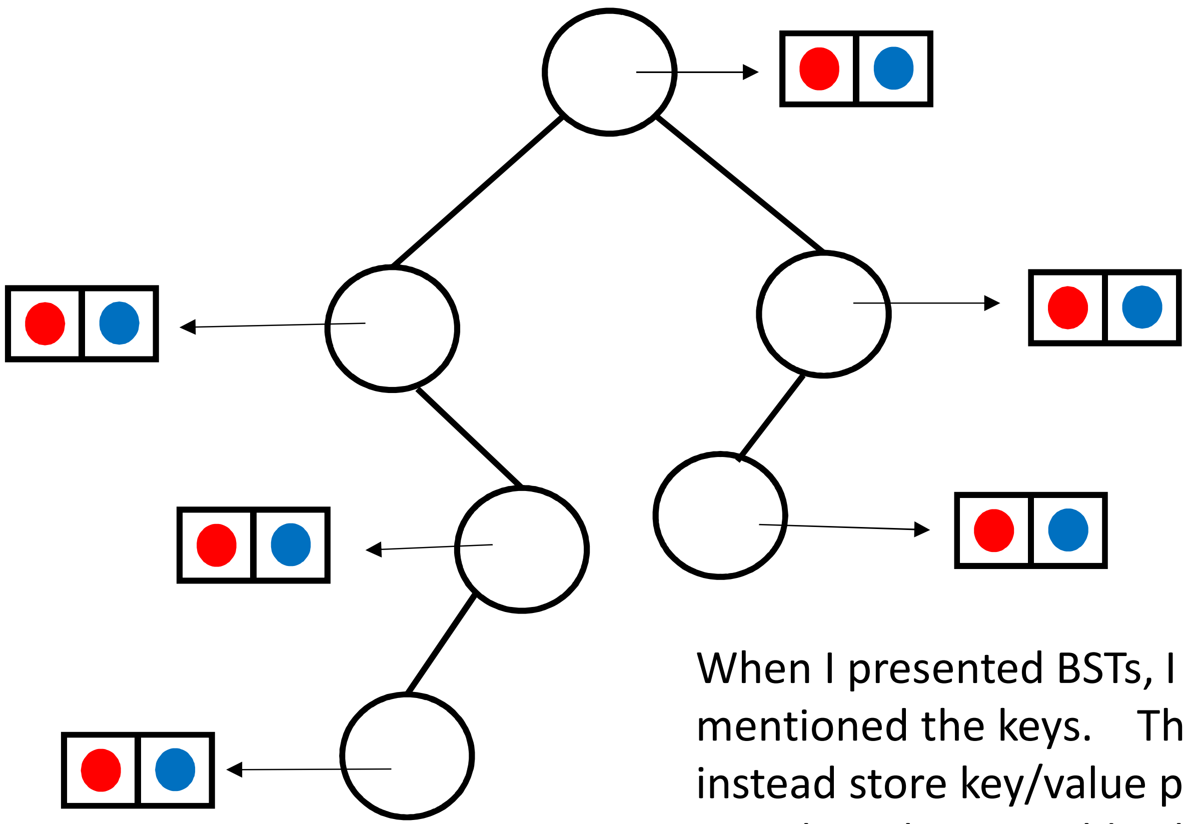
Array list (sorted by key)



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

How would you implement these operations?
What are the best and worst case time complexities ?

Binary Search Tree ("sorted" by key)

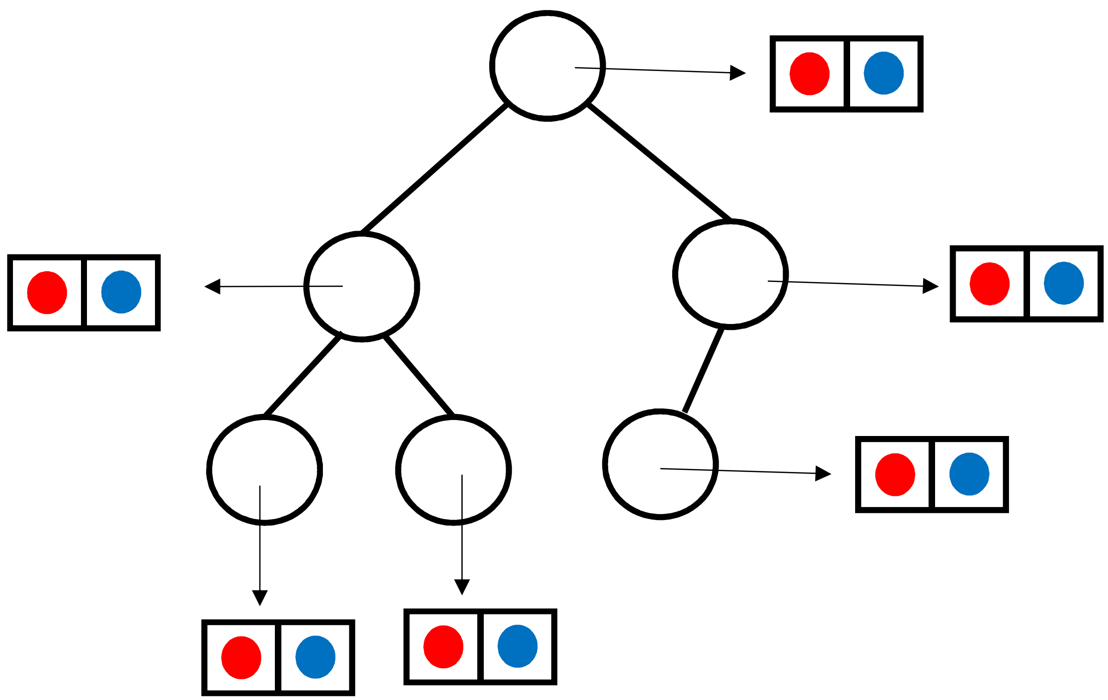


When I presented BSTs, I only mentioned the keys. The nodes could instead store key/value pairs and the BST algorithms would still work fine.

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

How would you implement these operations?
What are the best and worst case time complexities ?

minHeap (priority defined by key)



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

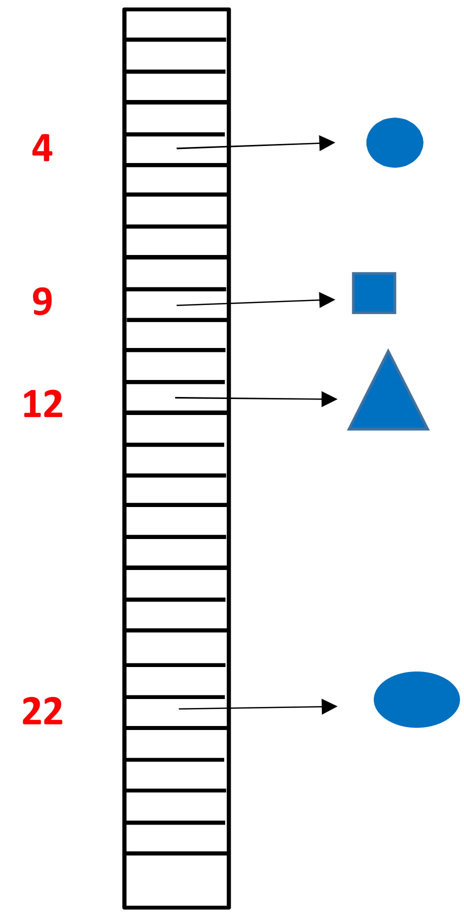
How would you implement these operations?
What are the best and worst case time complexities ?

Special case #1: what if keys are *comparable* ?

Special case #2: what if keys are positive integers in a *small* range ?

Then, we could use an array with elements of type V (value) and have O(1) access.

This would *not* work well if keys are 9 digit student IDs. Why not?



Special case #1: what if keys are *comparable* ?

Special case #2: what if keys are positive integers in small range ?

General case. What if keys are some other type ?

We will define a map from keys to a *large* range of positive integers. Such a map is called a *hash code*.

Next we will look at Java's `hashCode()` method.

Then, next lecture, I will tell you how to use this hash code.

Recall lecture 13: `Object.hashCode()`

<u>class Object</u>

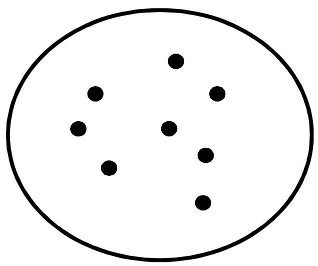
\+ Object()

\+ equals(Object) : boolean
\# clone( ) : Object
\+ hashCode( ) : int
\+ toString( ) : String
:

Returns a (positive) integer.

You can think of it as the address of the object, although this is not required in any technical sense.

`Object.hashcode()`



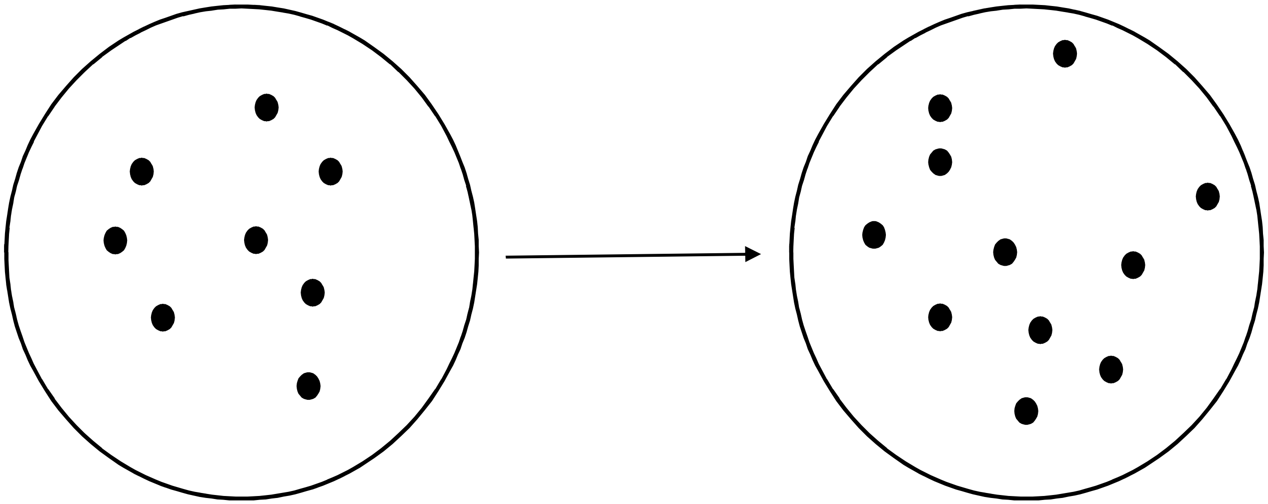
$$\xrightarrow[\text{(not many-to-1)}]{\text{1-to-1}} \{\, 0, 1, 2, \ldots, 2^{24} - 1\}$$

objects in a Java program (runtime)

object's *address* in JVM memory (24 bits)

If `obj1` *and* `obj2` *are reference variables, and if the objects that they reference inherit the* *`Object.hashCode()`* *method, then* `obj1.hashcode() == obj2.hashcode()`
is equivalent to `obj1 == obj2`.

`String.hashcode()`



Strings `int` (32 bit)

How is `String.hashcode()` defined?

Example of a *simpler* hash code for strings

(***<u>not</u>*** the definition of `String.hashCode()`)

Unicode (16 bit)

$$h(s) \equiv \sum_{i=0}^{s.length-1} s[i]$$

$s[0]$ is the first character in the sequence, $s[1]$ is second, etc.

e.g. $h("eat") = h("ate") = h("tea")$

ASCII values of 'a', 'e', 't' are 97, 101, 116.

String.hashcode()

$$\texttt{s.hashCode()} \equiv \sum_{i=0}^{s.length-1} s[i] * x^{s.length-1-i}$$

where $x = 31$.

e.g. s = “eat”, `s.hashcode()` = $101 * 31^2 + 97 * 31 + 116$

‘e’ ‘a’ ‘t’

s.length = 3 s[0] s[1] s[2]

String.hashcode()

$$\texttt{s.hashCode()} \equiv \sum_{i=0}^{s.length-1} s[i] * x^{s.length-1-i}$$

where $x = 31$.

e.g. s = "ate", `s.hashcode()` = $97 * 31^2 + 116 * 31 + 101$

	'a'	't'	'e'
s.length = 3	s[0]	s[1]	s[2]

docs.oracle.com/javase/8/docs/api/java/lang/String.html#hashCode--

hashCode

```
public int hashCode()
```

Returns a hash code for this string. The hash code for a `String` object is computed as

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

using `int` arithmetic, where `s[i]` is the *i*th character of the string, `n` is the length of the string, and ^ indicates exponentiation. (The hash value of the empty string is zero.)

Overrides:

`hashCode` in class `Object`

Returns:

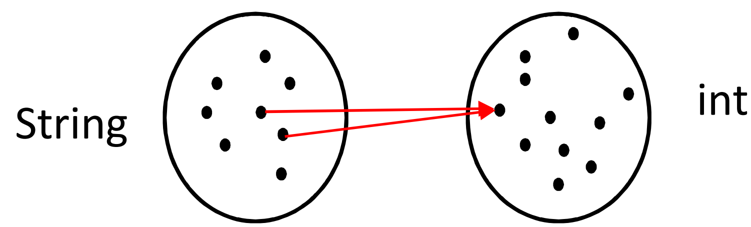
a hash code value for this object.

`String.hashcode()`

$$\texttt{s.hashCode()} \equiv \sum_{i=0}^{s.length-1} s[i] * (31)^{s.length-1-i}$$

Q: If `s1.hashCode() == s2.hashCode()`
then can we conclude `s1.equals(s2)` is true ?

A: No. `s1.equals(s2)` may be either true or false.



`s1.hashCode() == s2.hashCode()` is true, but `s1.equals(s2)` is false

`String.hashcode()`

$$\texttt{s.hashCode()} \equiv \sum_{i=0}^{s.length-1} s[i] * (31)^{s.length-1-i}$$

Q: If `s1.hashCode() != s2.hashCode()`
then what can we conclude about `s1.equals(s2)` ?

A: `s1.equals(s2)` is false.

ASIDE: Java uses "Horner's rule" for efficient polynomial evaluation

s[0] * 31^3 + s[1] * 31^2 + s[2]* 31 + s[3]

There is no need to compute each x^i separately.

ASIDE: Java uses "Horner's rule" for efficient polynomial evaluation

$$s[0] * 31^3 + s[1] * 31^2 + s[2] * 31 + s[3]$$

$$= (s[0] * 31^2 + s[1] * 31^1 + s[2]) * 31 + s[3]$$

$$= ((s[0] * 31^1 + s[1]) * 31 + s[2]) * 31 + s[3]$$

```
h = 0
for (i = 0; i < s.length; i++)
    h = h*31 + s[i]
```

For a degree n polynomial, Horner's rule uses O(n) multiplications, not $O(n^2)$.

Coming up…

Lectures

Wed 23

Hashing

Fri. March 25

Graphs 1

Assessments

Assignment 4 will be posted Wednesday, hopefully.