

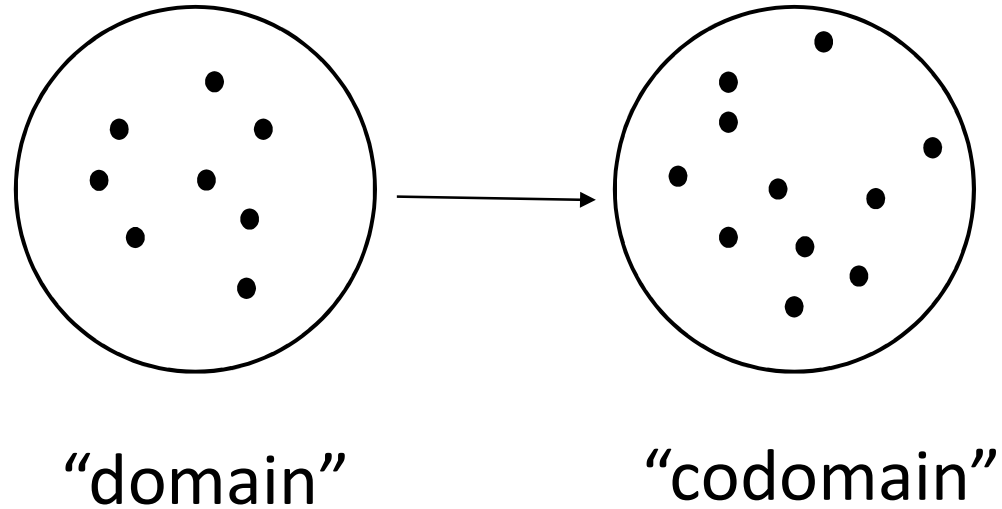
COMP 250

Lecture 29

maps

March 21, 2022

# Map (Mathematics)



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”):

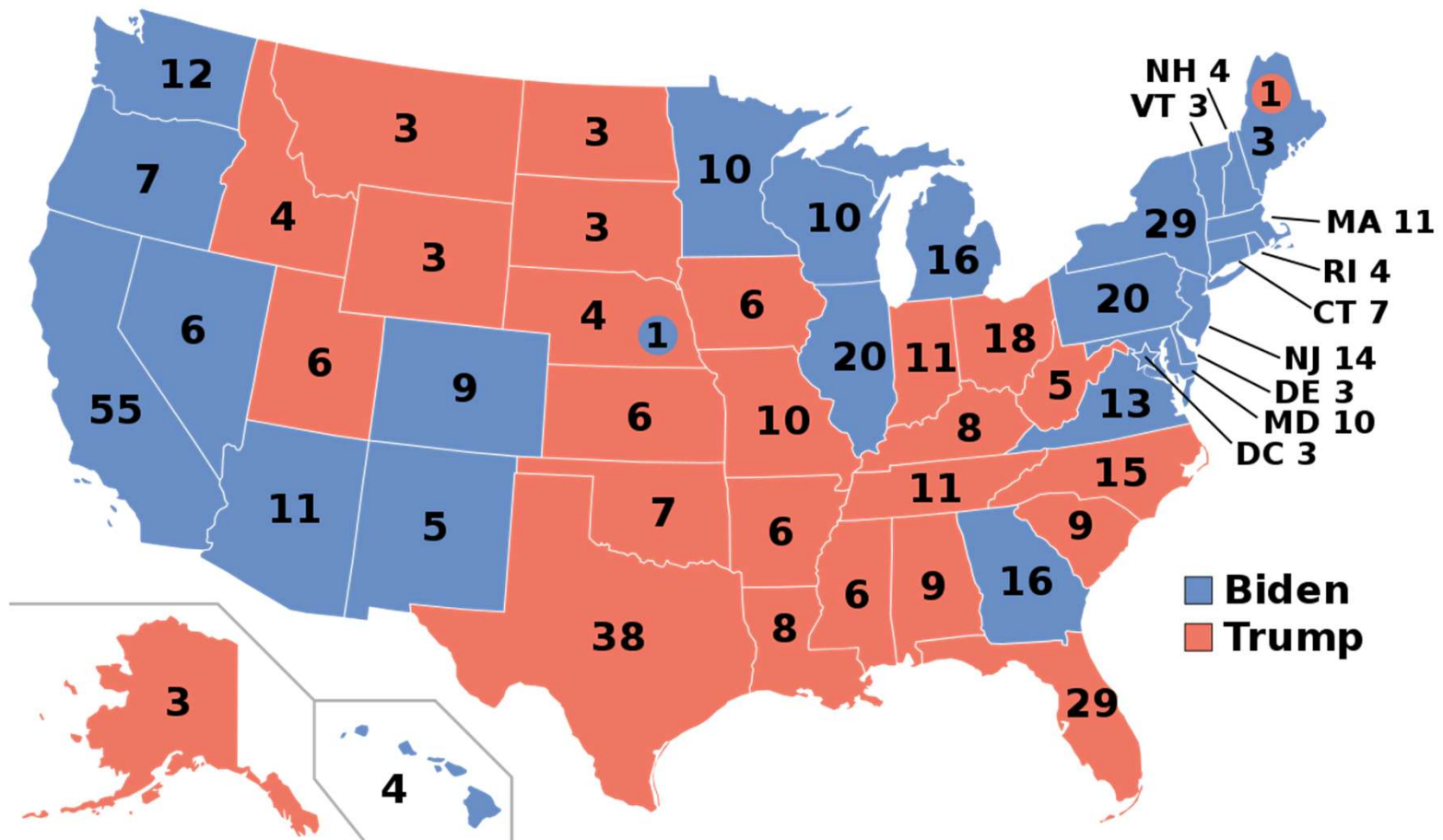
$$f(x): \mathfrak{R}^n \rightarrow \mathfrak{R}^m$$

# Maps in everyday life



$\text{map}(x,y)$  : position in 2D image  $\rightarrow$  2D position in Montreal <sup>4</sup>

# Color map



vote\_result : US\_state → { blue(Dem), red(Rep) }


# Restaurant Menu


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


menu : dish\_name → price


# Train Schedule

Schedule Information Map





















 **Vaudreuil-Hudson line** Direction Montréal

→ Direction Beaconsfield / Hudson / Vaudreuil 

 Closure of the Turcot Interchange - Temporary schedule valid from November 9 to 12, 2018 (French PDF)

 Download the full schedule of the exo1 Vaudreuil-Hudson line - starting August 20th (French PDF)

**Today, November 13 at 06:27**

 gare Lucien-L'Allier	<b>07:05</b> 07:50   
 gare Vendôme	<b>07:10</b> 07:55   
 gare Montréal-Ouest	<b>07:15</b> 08:00  
 gare Lachine	<b>07:21</b> 08:07
 gare Dorval	<b>07:24</b> 08:10 
 gare Pine Beach	<b>10:20</b> 12:55 
 gare Valois	<b>10:24</b> 12:59 
 gare Pointe-Claire	<b>07:29</b> 08:14 

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

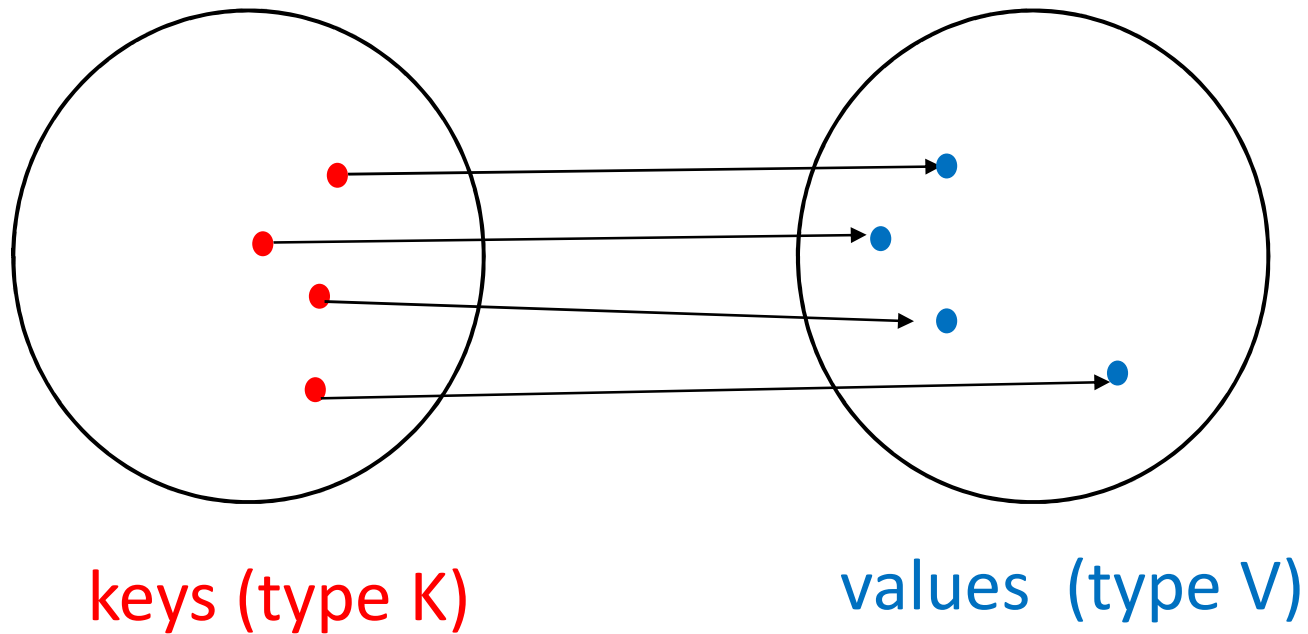
# Index in a book

edge, 310  
  destination, 613  
  endpoint, 613  
  incident, 613  
  multiple, 614  
  origin, 613  
  outgoing, 613  
  parallel, 614  
  self-loop, 614  
edge list, 619–621  
edge of a graph, 612  
edge relaxation, 653  
edit distance, 608  
element uniqueness problem,  
  174–175, 215  
encapsulation, 62  
encryption, 115  
endpoints, 613  
enum, 22  
equals method, 25, 138–140  
equivalence relation, 138  
equivalence testing, 138–140  
erasure, 140  
Error class, 86, 87  
Euclidean norm, 56  
Euler tour of a graph, 677, 681  
Euler tour tree traversal,  
  348–349, 358  
favorites list, 294–299  
FavoritesList class, 295–296  
FavoritesListMTF class, 298,  
  399  
Fibonacci heap, 659  
Fibonacci series, 73, 180, 186,  
  216–217, 480  
field, 5  
FIFO, *see* first-in, first-out  
File class, 200  
file system, 198–201, 310, 345  
**final** modifier, 11  
first-fit algorithm, 692  
first-in, first-out (FIFO)  
  protocol, **238**, 255, 336,  
  360, 699–700  
Flajolet, Philippe, 188  
Flanagan, David, 57  
floor function, **163**, 209  
flowchart, 31  
Floyd, Robert, 400, 686  
Floyd-Warshall algorithm,  
  644–646, 686  
for-each loop, 36, 283  
forest, 615  
fractal, 193  
fragmentation of memory, 692  
frame, 192, 688  
adjacency list, 619,  
  622–623  
  adjacency map, 619, 624,  
  626  
  adjacency matrix, 619, 625  
  edge list, 619–621  
  depth-first search, 631–639  
  directed, 612, 613, 647–649  
  mixed, 613  
  reachability, 643–646  
  shortest paths, 651–661  
  simple, 614  
  strongly connected, 615  
  traversal, 630–642  
  undirected, 612, 613  
  weighted, 651–686  
greedy method, 597, 652, 653  
Guava library, 448  
Guibas, Leonidas, 530  
Guttag, John, 101, 256, 305  
Harmonic number, 171, 221  
hash code, 411–415  
  cyclic-shift, 413–414  
  polynomial, 413, 609  
hash table, 410–427  
  clustering, 419  
  collision, 411

index : term → list of pages containing term

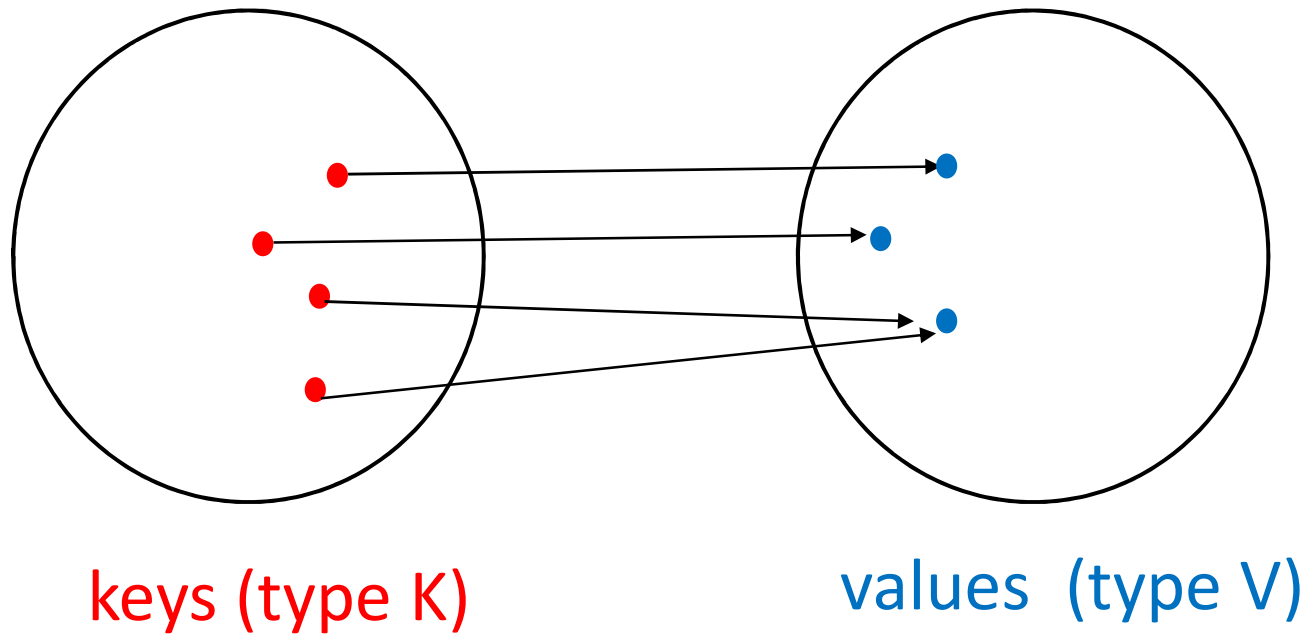


# Map (ADT)



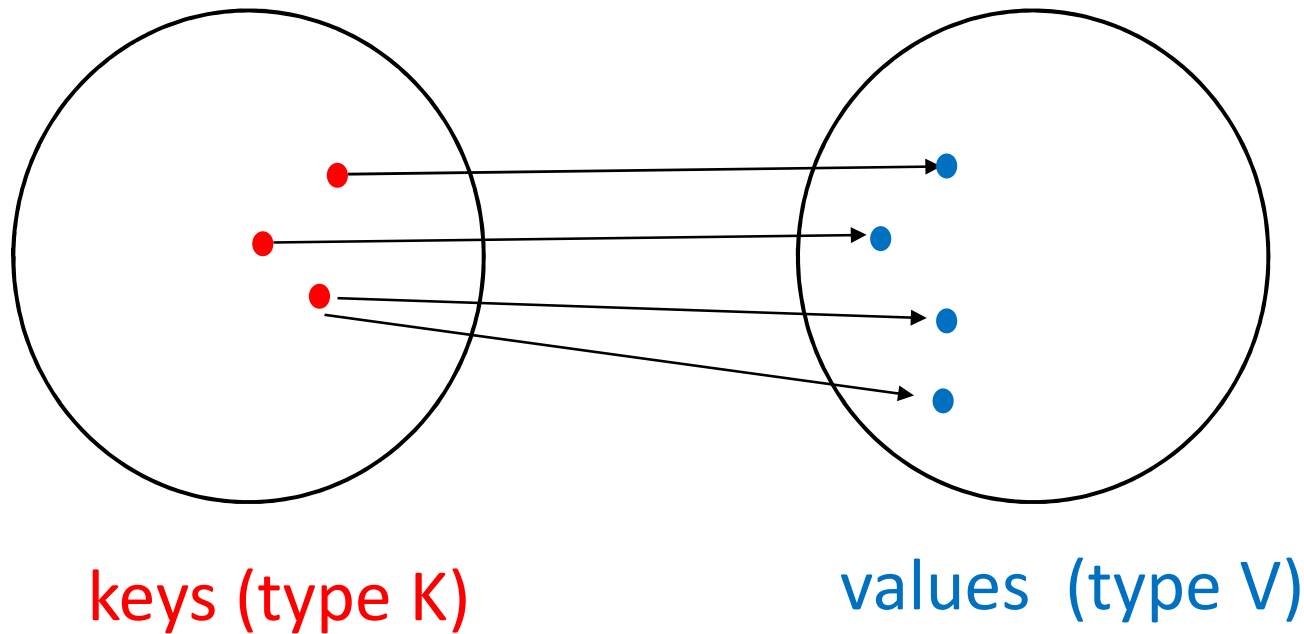
A map is a set of (key, value) pairs.  
For each key, there is at *most one* value.

# Map (ADT)



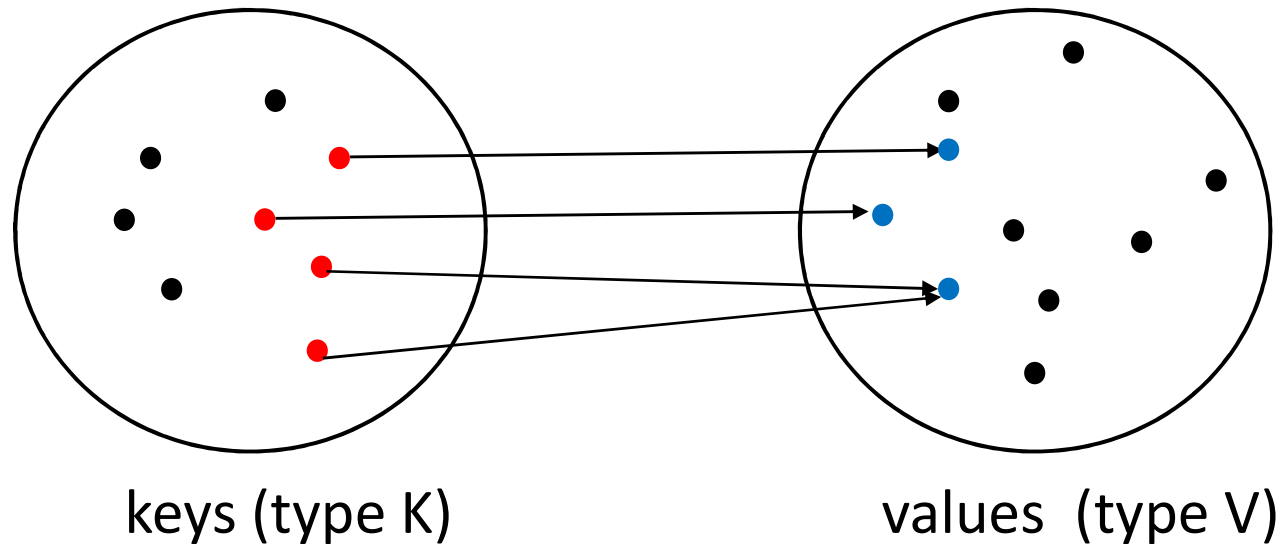
Two keys can map to the same value.

# Map (ADT)



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

# Map Entry

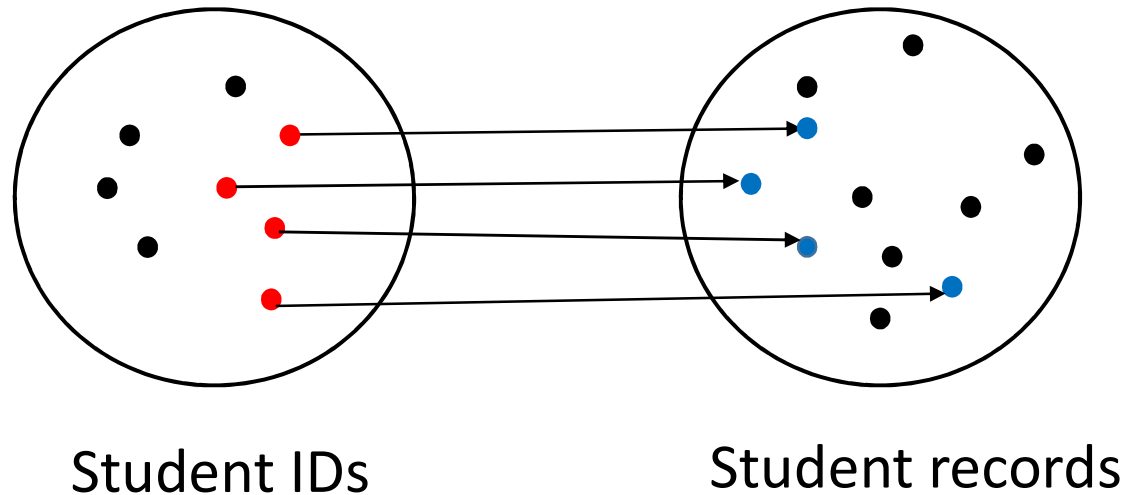


Each (key, value) pair is called an *entry*.

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)

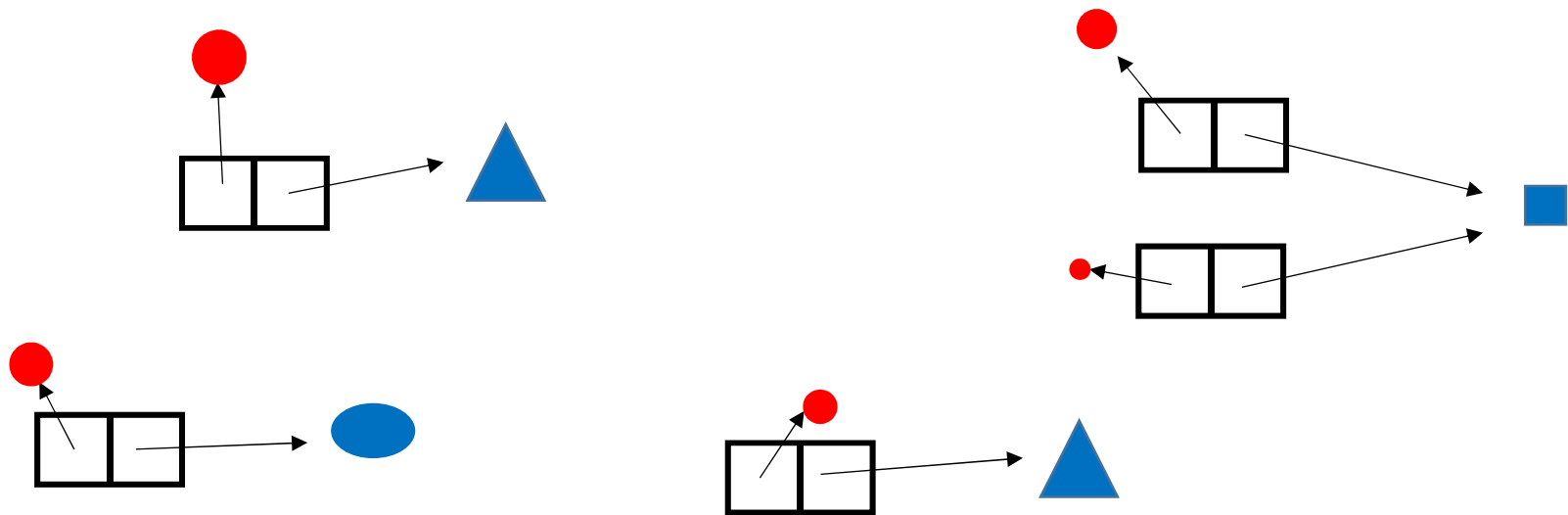
- remove(key)

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....

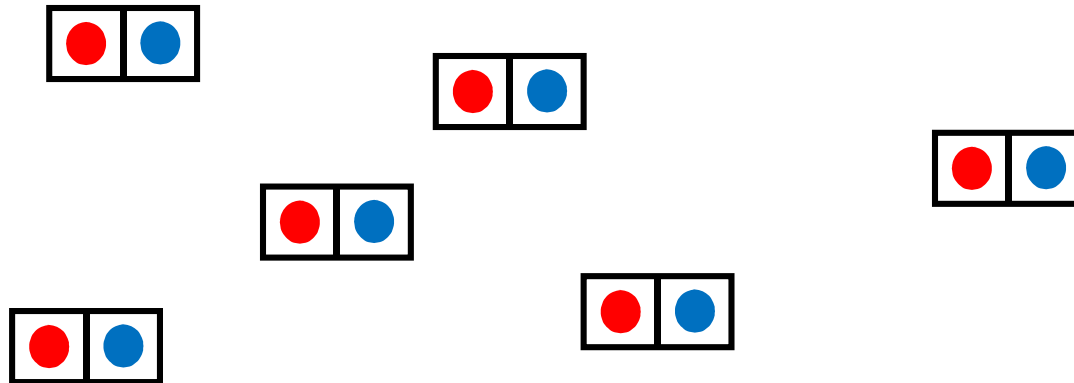
When programming with maps in Java, keys and value variables are *reference* types. On this slide, the keys are different sized red disks and the values are blue shapes.



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

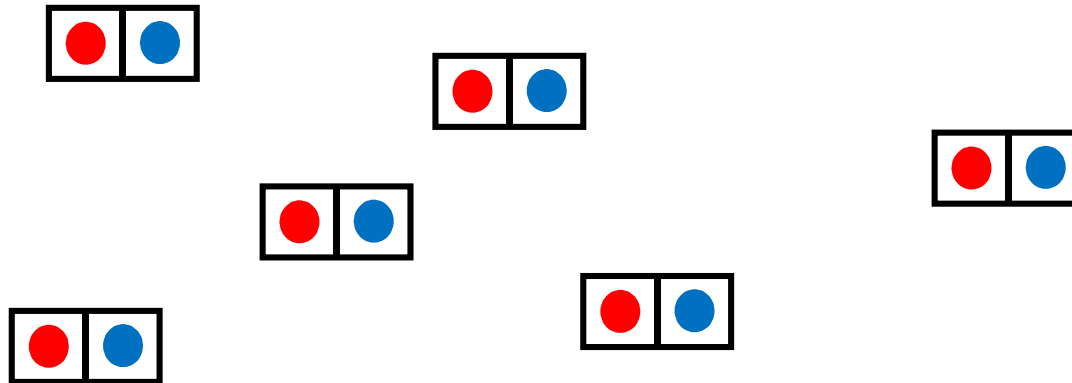
# About the figures....

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...

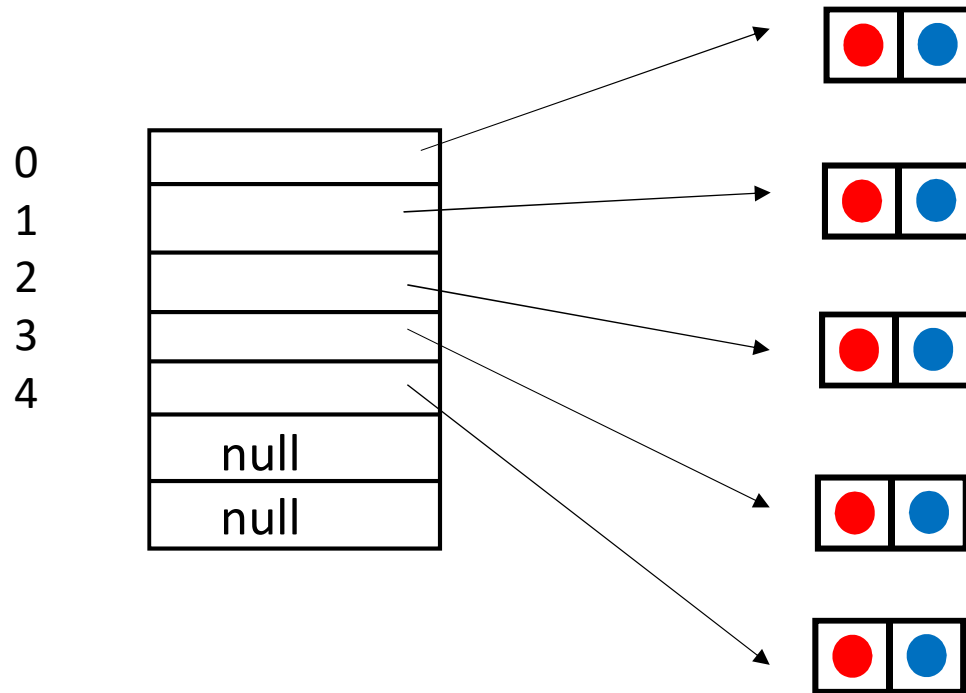


# Data Structures for Maps ?

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



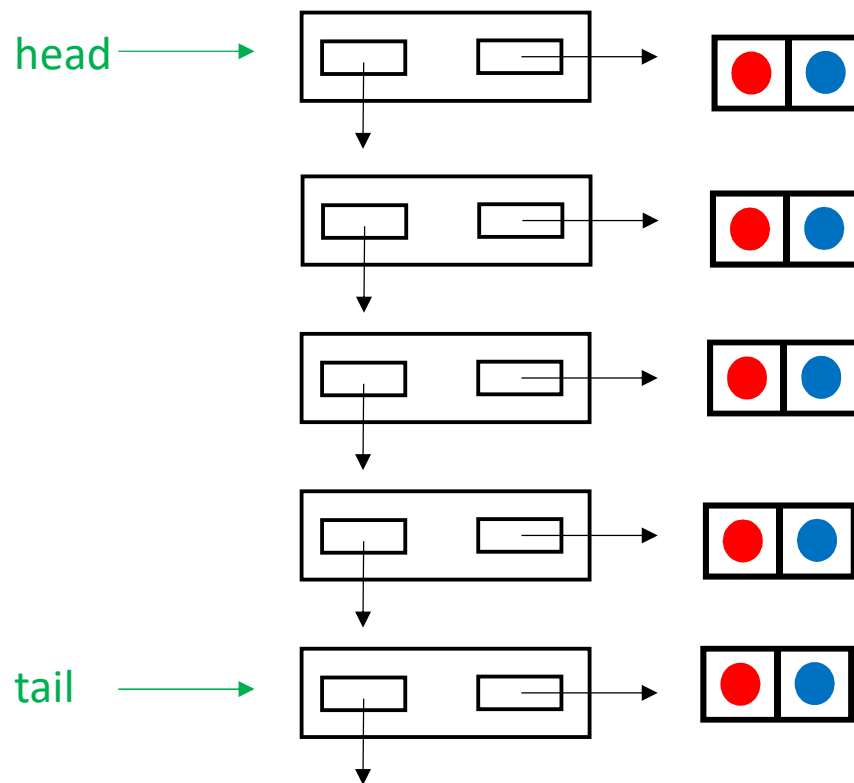
# Array list



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



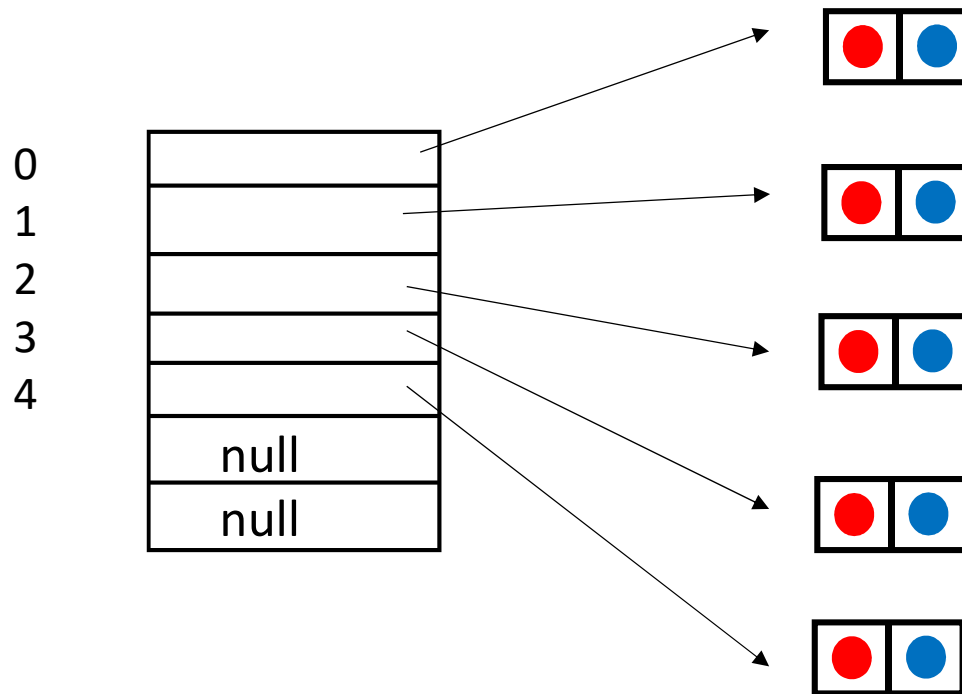
put( key, value )  
get(key)  
remove(key)

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)

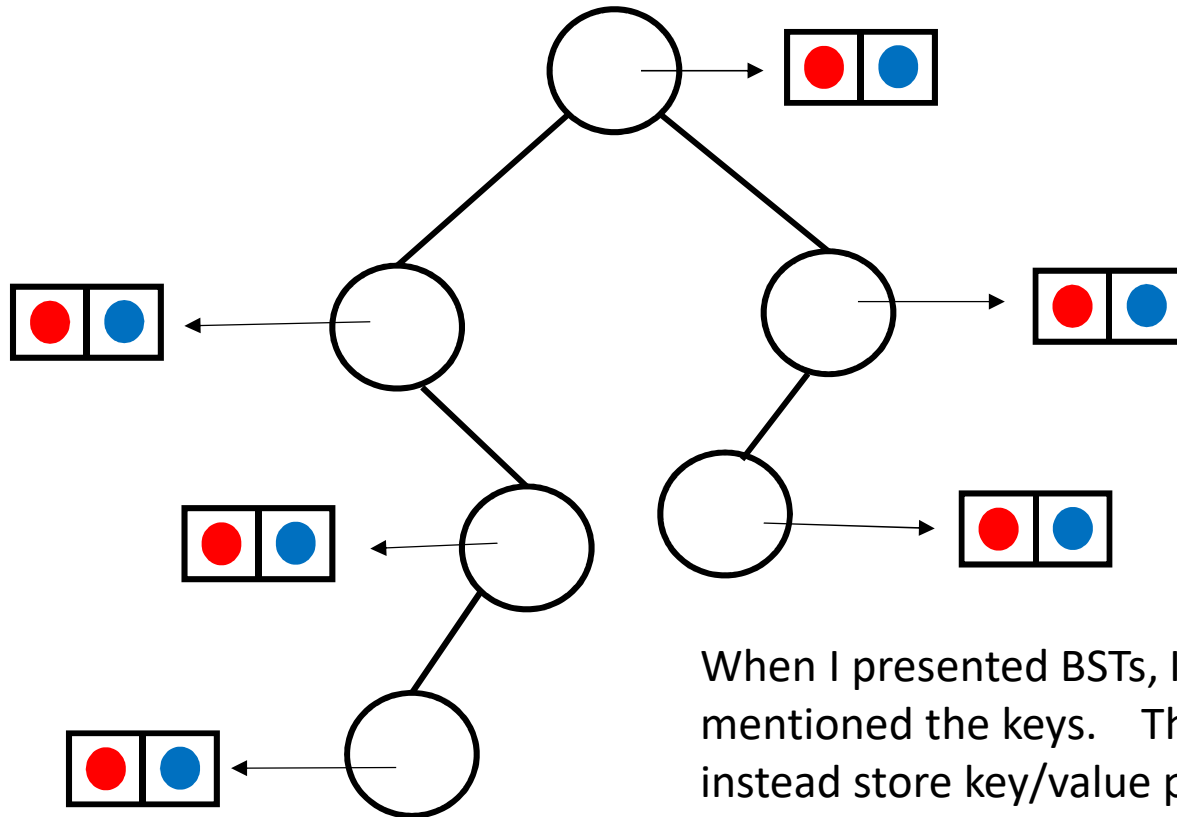


put(key,value)  
get(key)  
remove(key)

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



# Binary Search Tree (“sorted” by key)

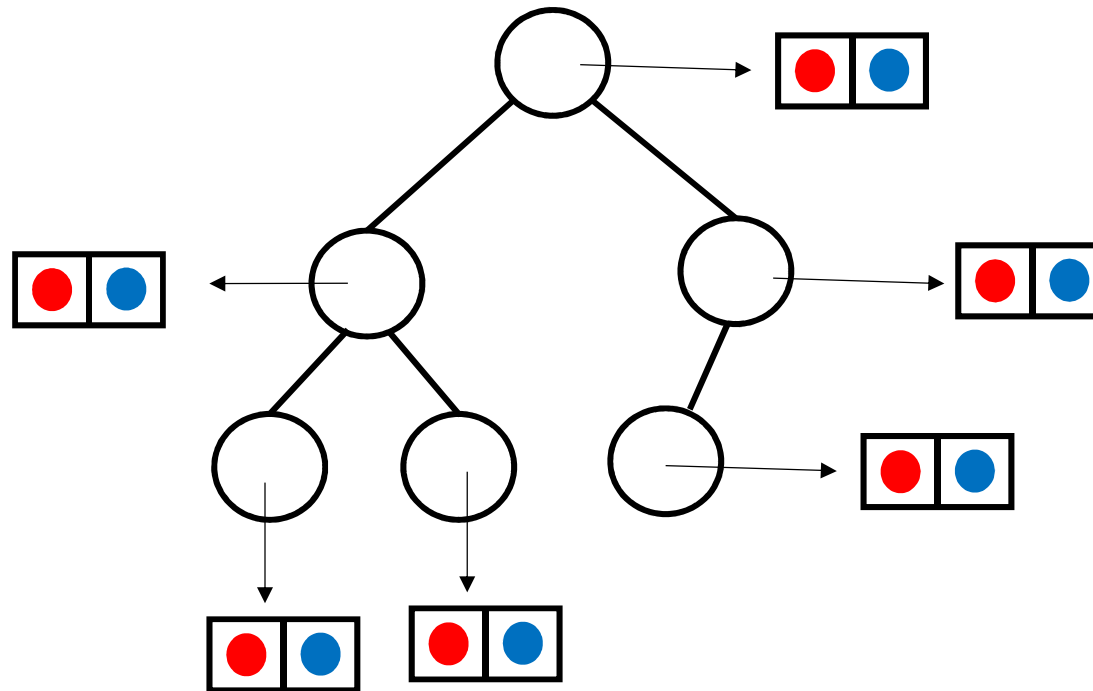


When I presented BSTs, I only mentioned the keys. The nodes could instead store key/value pairs and the BST algorithms would still work fine.

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)

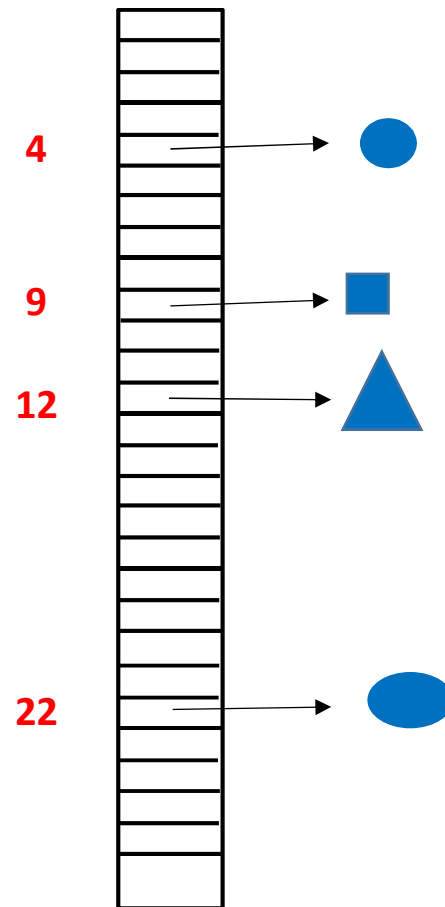
How would you implement these operations?  
What are the best and worst case time complexities ?

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.

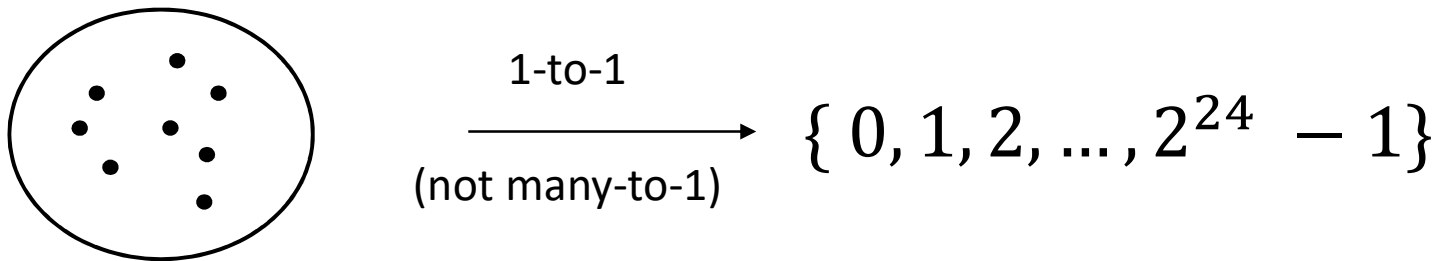
# Recall lecture 13: Object.hashCode()

```
class Object  
  
+ 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 ()

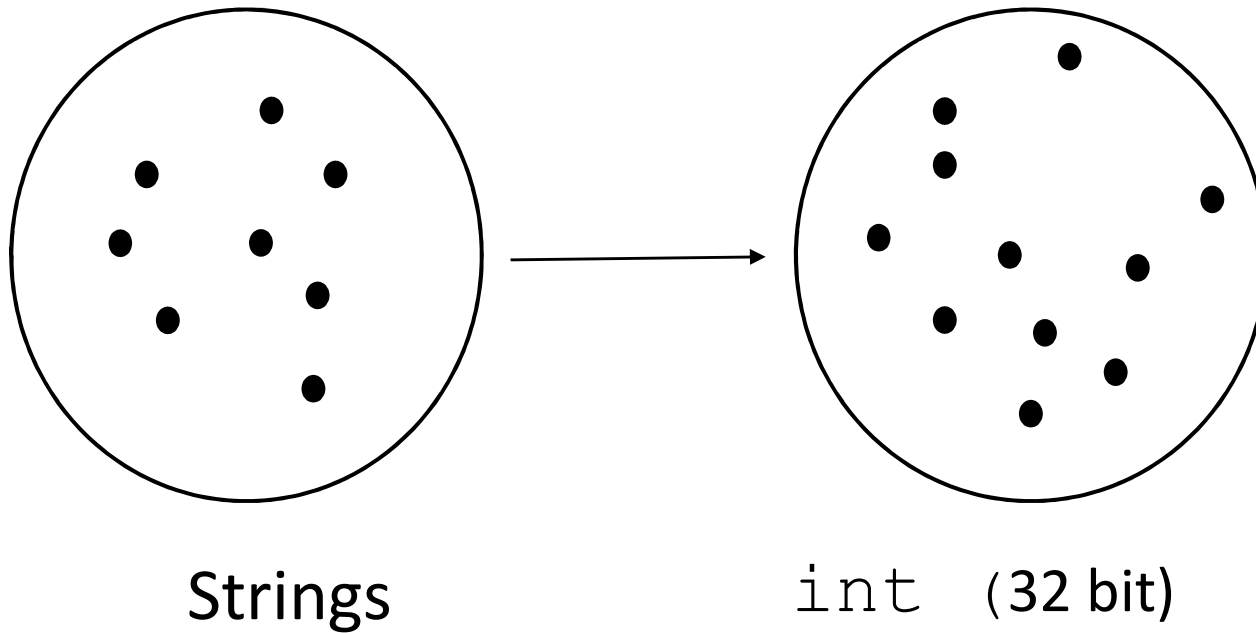


objects in a Java  
program (runtime)

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()`

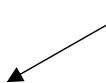


How is `String.hashCode()` defined?

# Example of a *simpler* hash code for strings

(**not** the definition of `String.hashCode()`)

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

Unicode (16 bit) 

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

e.g.  $h(\text{"eat"}) = h(\text{"ate"}) = h(\text{"tea"})$

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



# String.hashCode ()

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

where  $x = 31$ .

e.g.  $s = \text{"eat"}$ ,  $s.hashCode () = 101 * 31^2 + 97 * 31 + 116$

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

# String.hashCode ()

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

where  $x = 31$ .

e.g.  $s = \text{"ate"}$ ,  $s.hashCode () = 97 * 31^2 + 116 * 31 + 101$

	'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

$$s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \dots + s[n-1]$$

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:

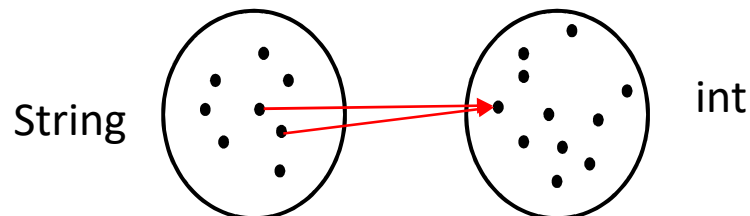
a hash code value for this object.

# String.hashCode ()

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

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()

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

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

$$\begin{aligned} & s[0] * 31^3 + s[1] * 31^2 + s[2] * 31 + s[3] \\ = & ( s[0] * 31^2 + s[1] * 31^1 + s[2] ) * 31 + s[3] \\ = & ( ( s[0] * 31^1 + s[1] ) * 31 + s[2] ) * 31 + s[3] \end{aligned}$$

```
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.

