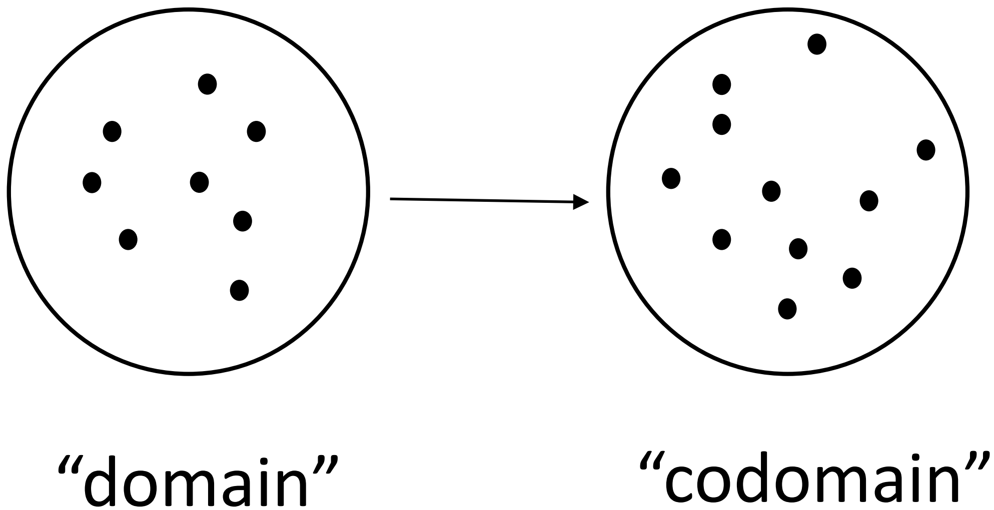
COMP 250

Lecture 28

maps

Nov. 14, 2018

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”):

$f(x)$: real numbers $\rightarrow$ real numbers

Asymptotic complexity in computer science:

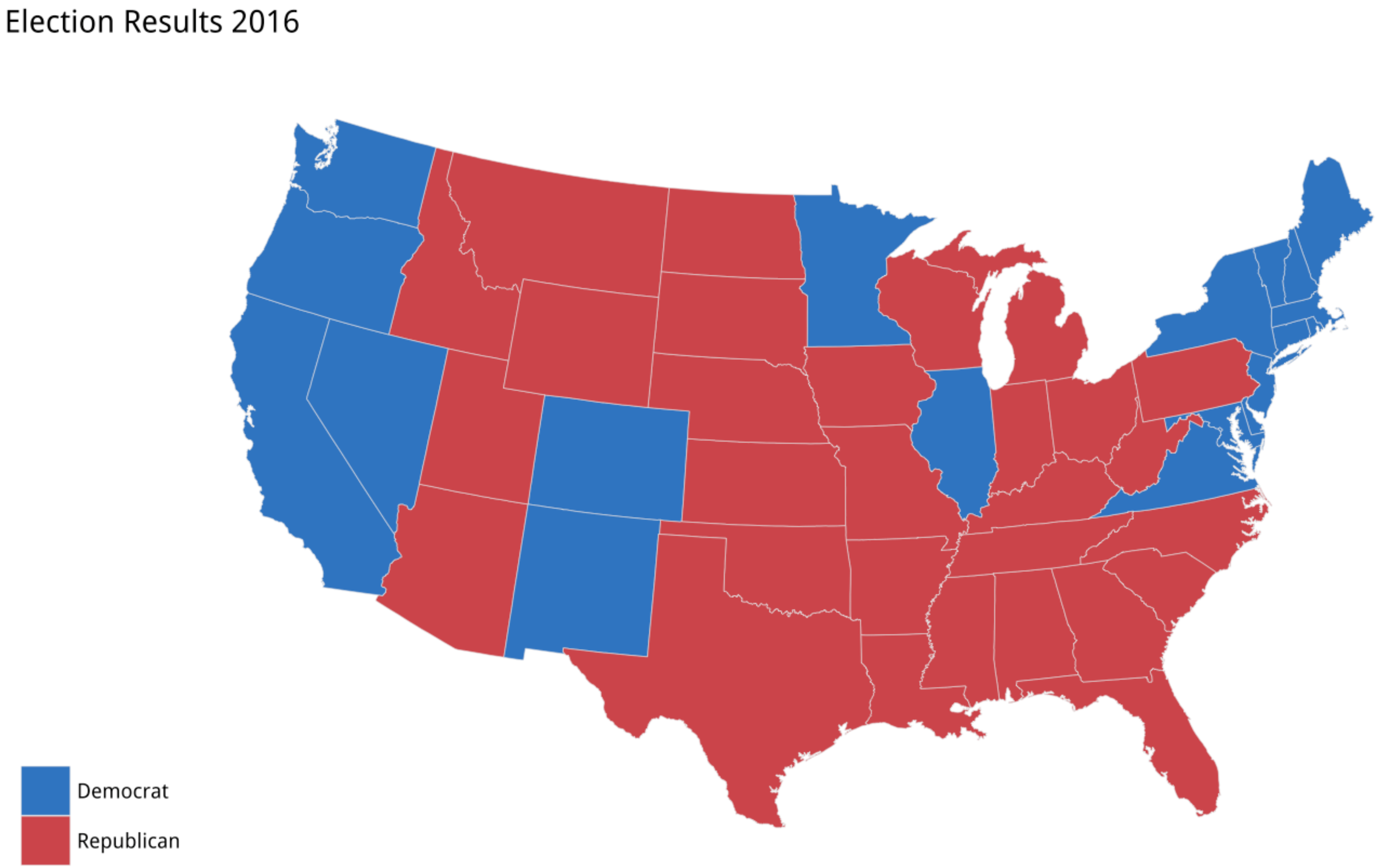
$t(n)$: input size $\rightarrow$ number of steps in an algorithm.

Maps in everyday life

The term “map” commonly refers to a 2D spatial representation of a region of the earth’s surface.

map(x,y) : position in image → position in 3D Montreal

Color map



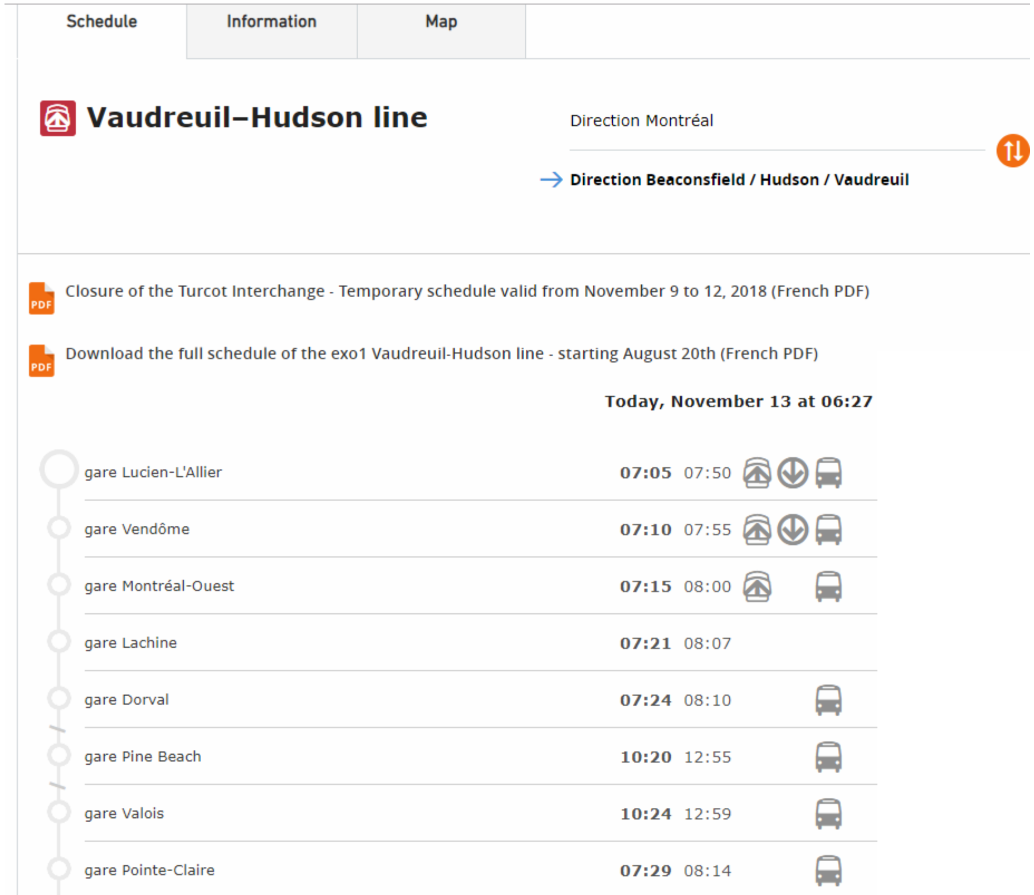
vote_result : US_state → {D, R}

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 → next train (or list of trains)

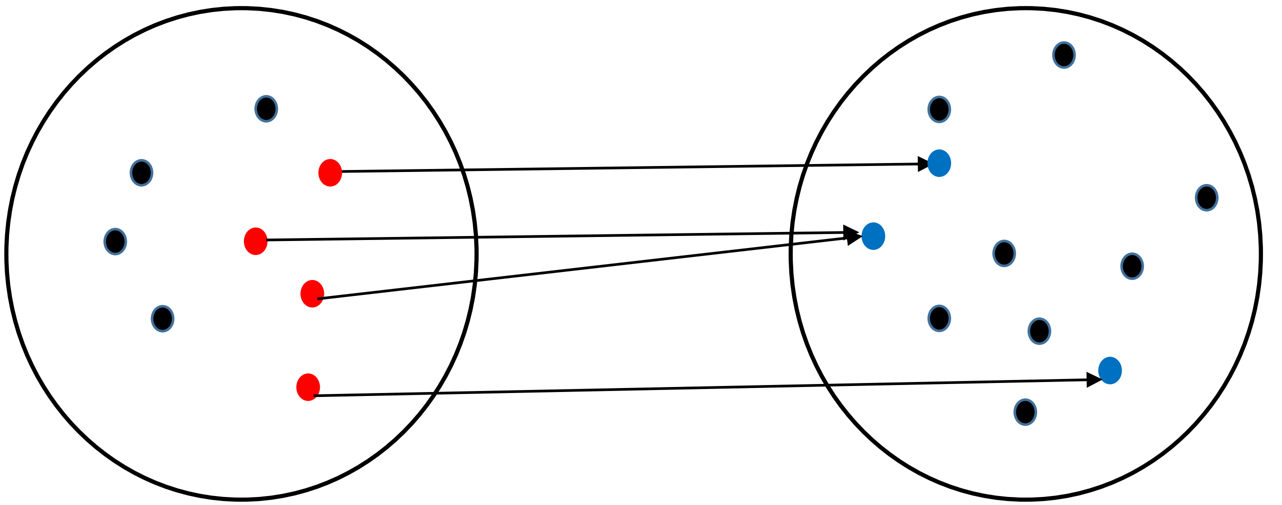
Index in a book

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)



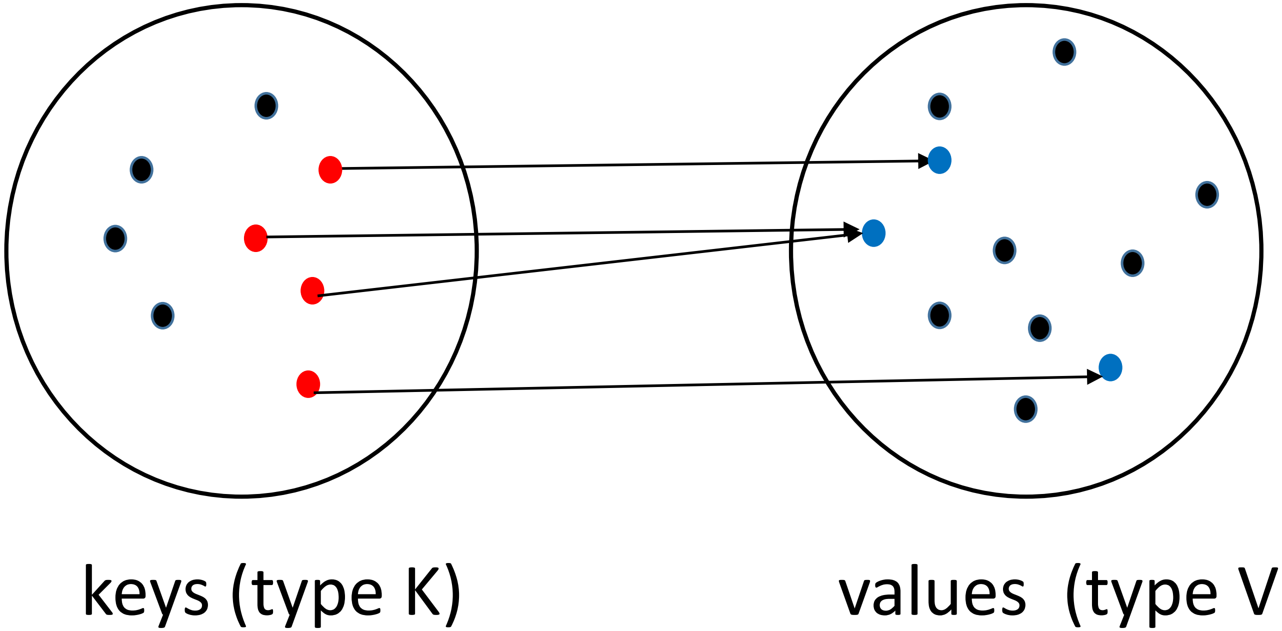
keys (type K) values (type V)

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

Map	Keys	Values
Vote result	State	Party (D, R)
Menu	Dish	Price
Train schedule	Station	Time of next train
Address book	Name	
Caller ID	Phone #	
Student file	ID (or Name)	

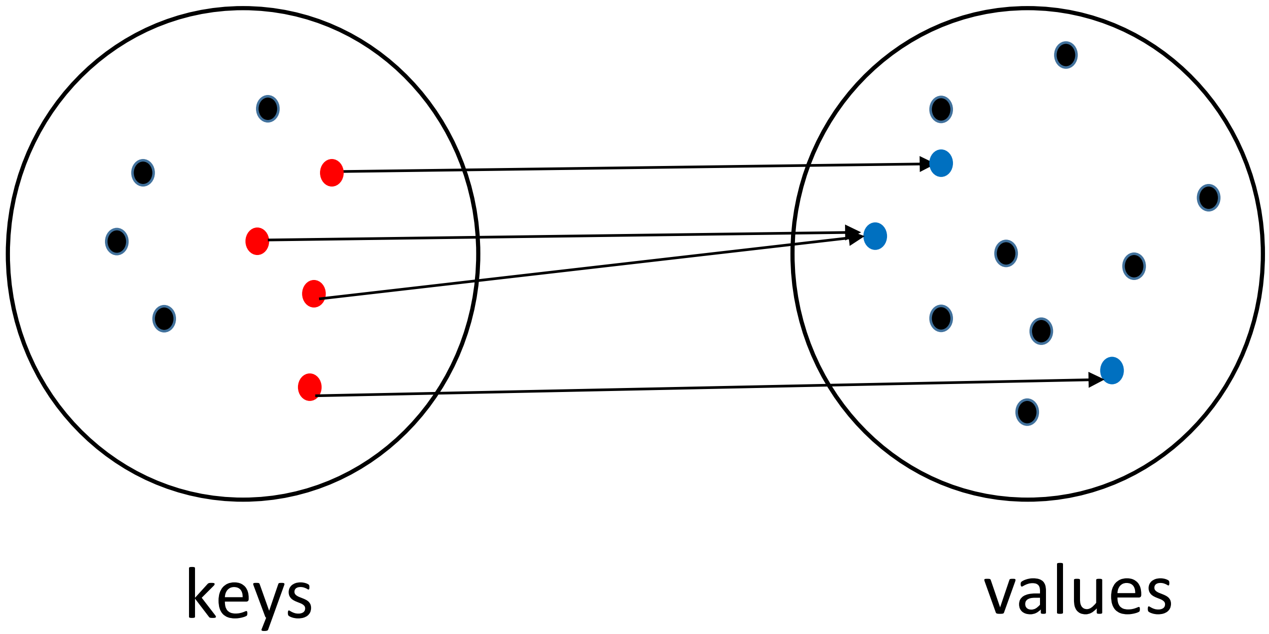
Map	Keys	Values
Vote result	State	Party (D, R)
Menu	Dish	Price
Train schedule	Station	Time of next train
Address book	Name	Address
Caller ID	Phone #	Name
Student file	ID (or Name)	Student record

Map (ADT)



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

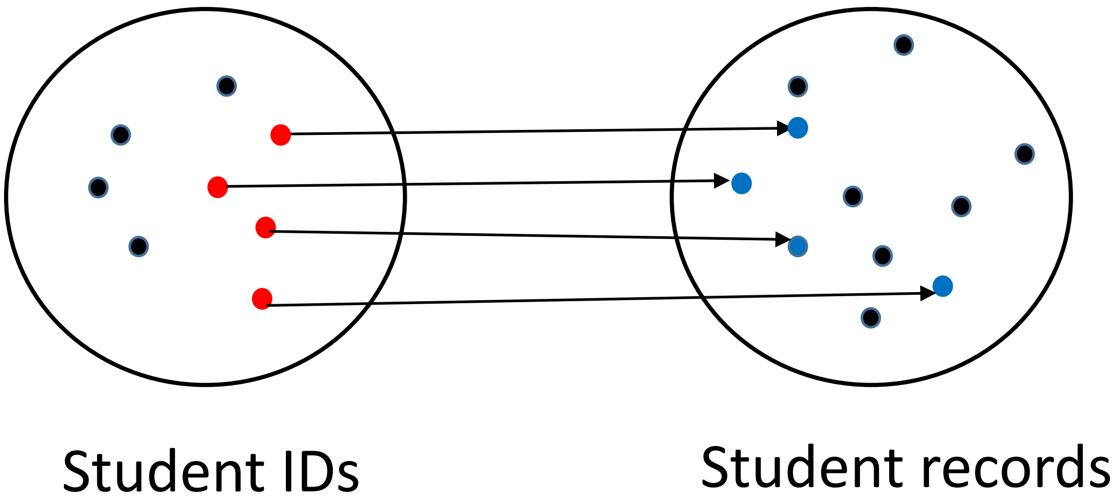
Map Entries



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

In this example, there are four entries.

Example



In COMP 250 this semester, the above mapping has ~670 entries.
Most McGill students are not taking COMP 250 this semester.

Student ID also happens to be part of the student record.

Map ADT

- put(key, value)   // add
- get(key)   //   returns a value
- remove(key)   //   returns a value
- …

Map ADT

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

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

Map ADT

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

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

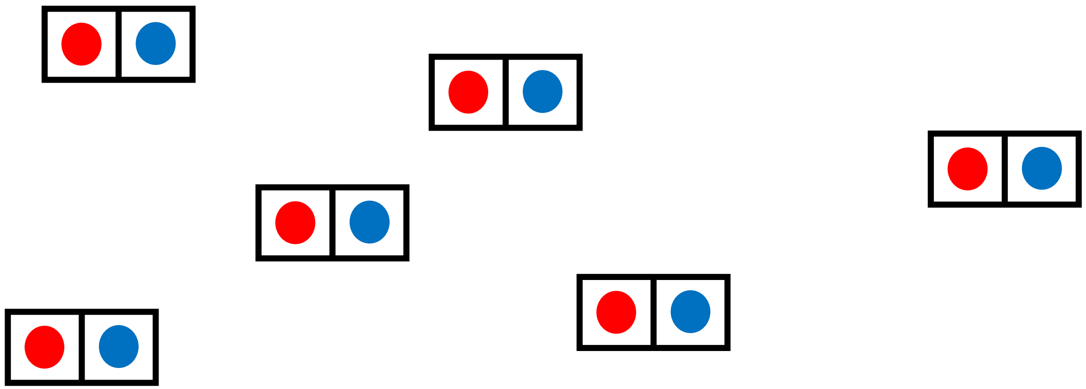
- remove(key)
- …

Map ADT

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

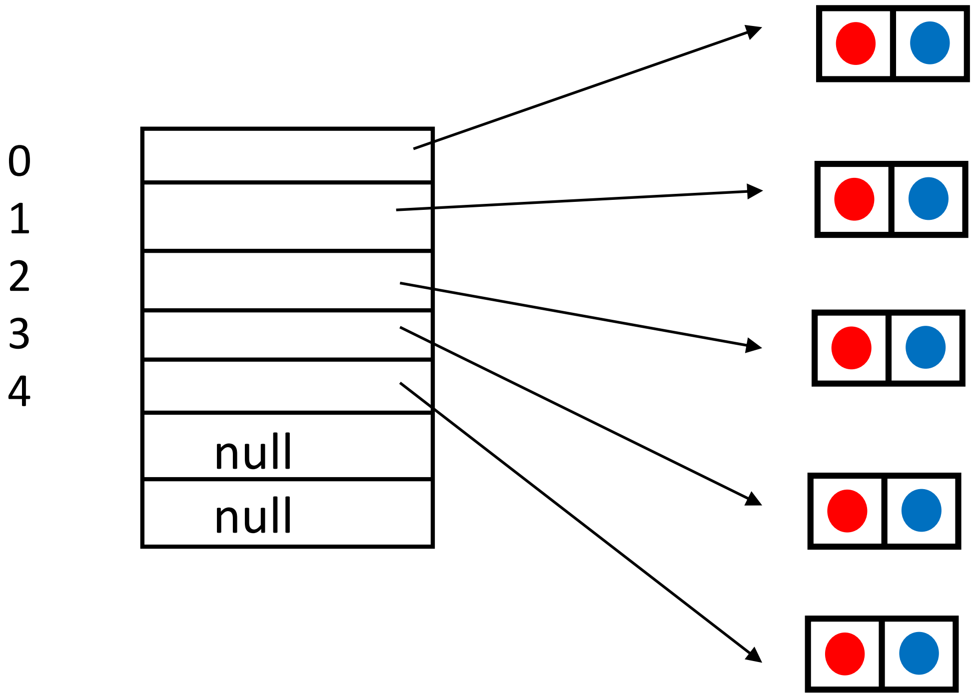
  Removes the entry for the key, if it is present, and returns the value to which this map previously associated the key, or null if the map contained no mapping for the key.
- …

Data Structures for Maps



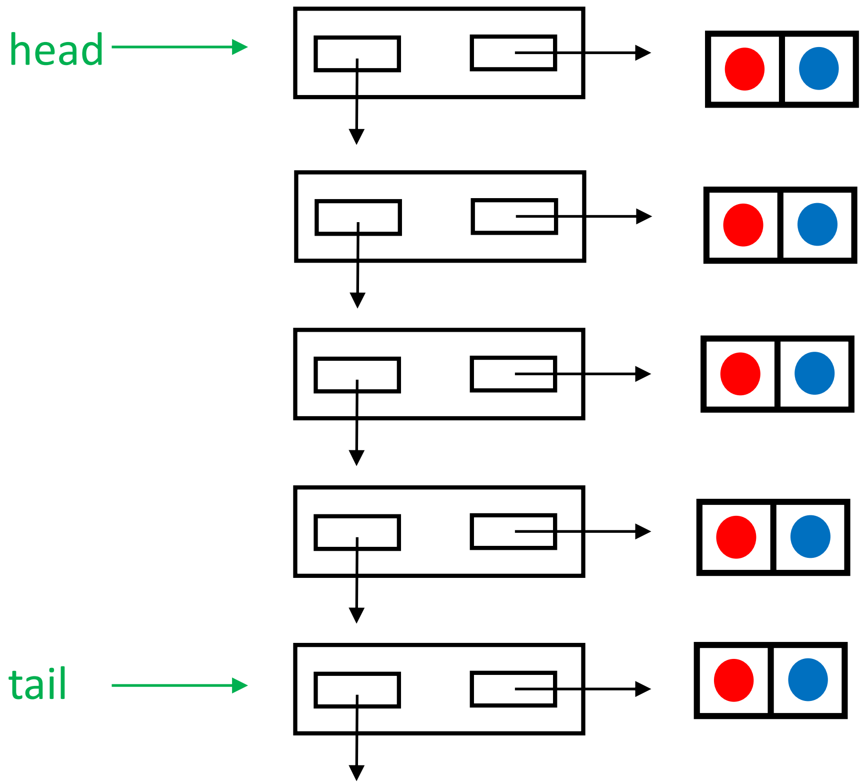
How to organize a set of (key, value) pairs, i.e. entries ?

Array list



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

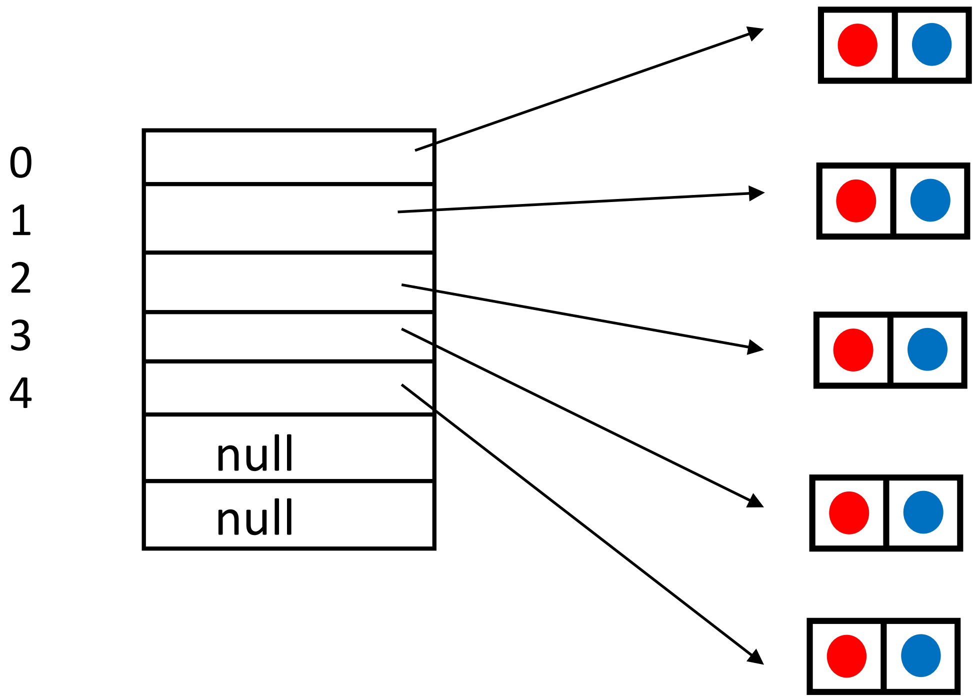
Singly (or Doubly) linked list



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

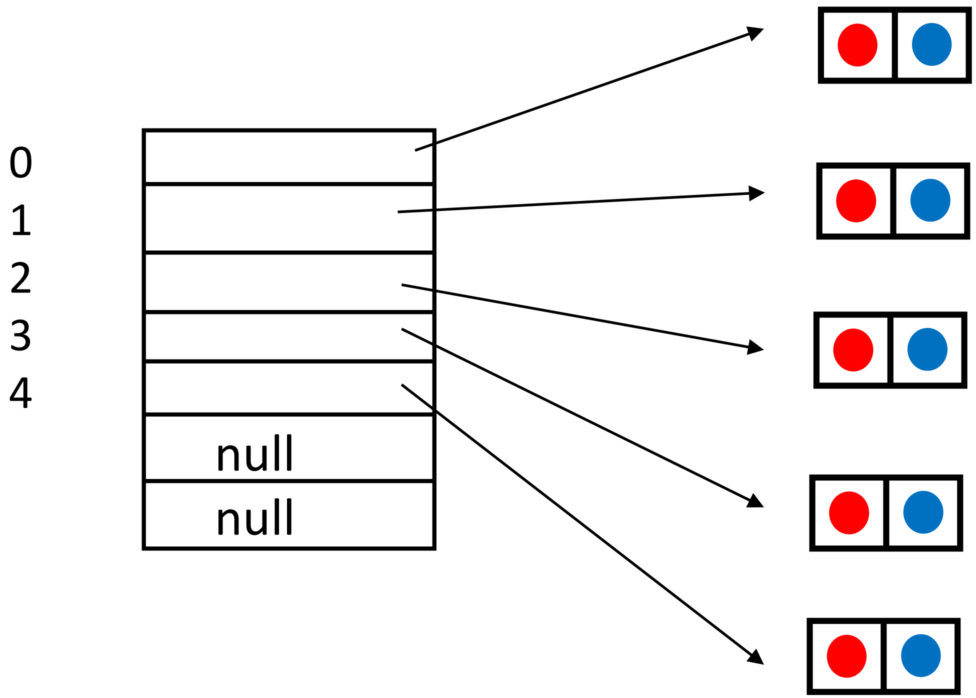
Special case #1: what if keys are *comparable* ?

Array list (sorted by key)



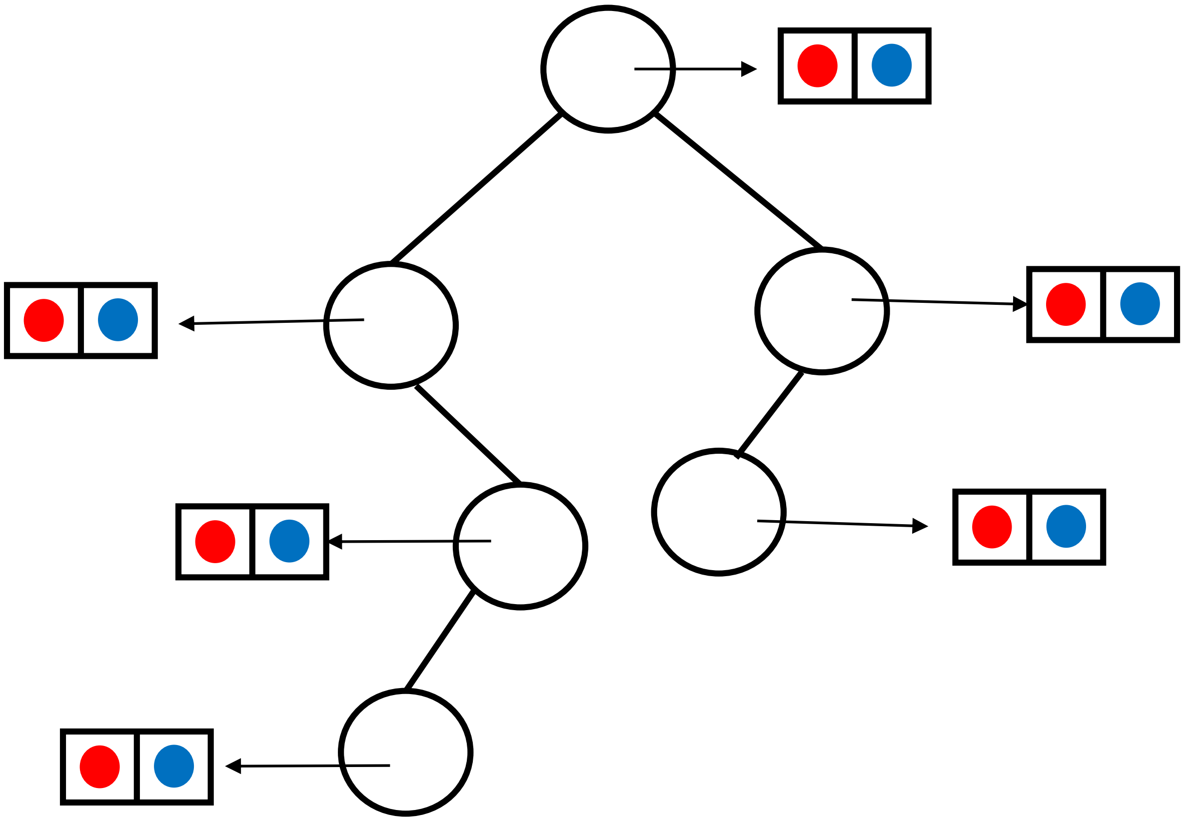
put(key,values), get(key), remove(key) ?

Array list (sorted by key)



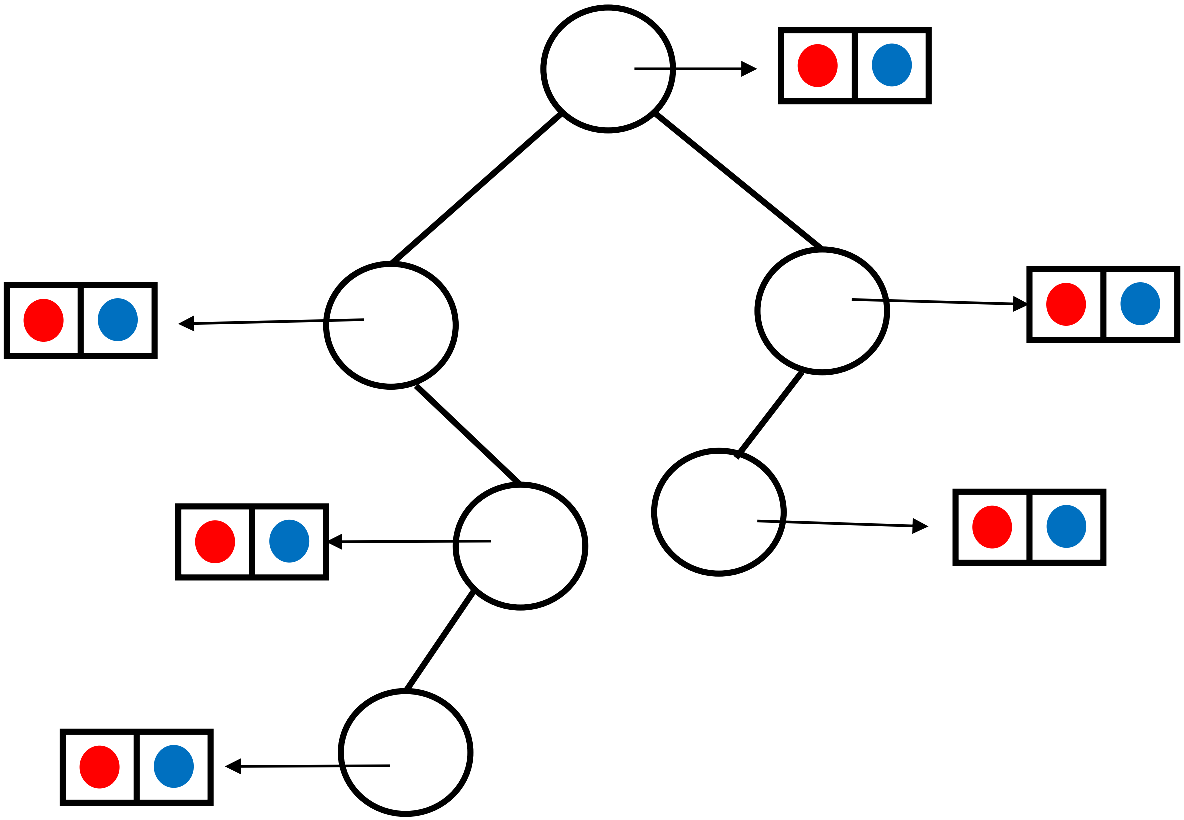
get could be performed in time O(log n), by binary search.
However, put and remove would be O(n).

Binary Search Tree (sorted by key)



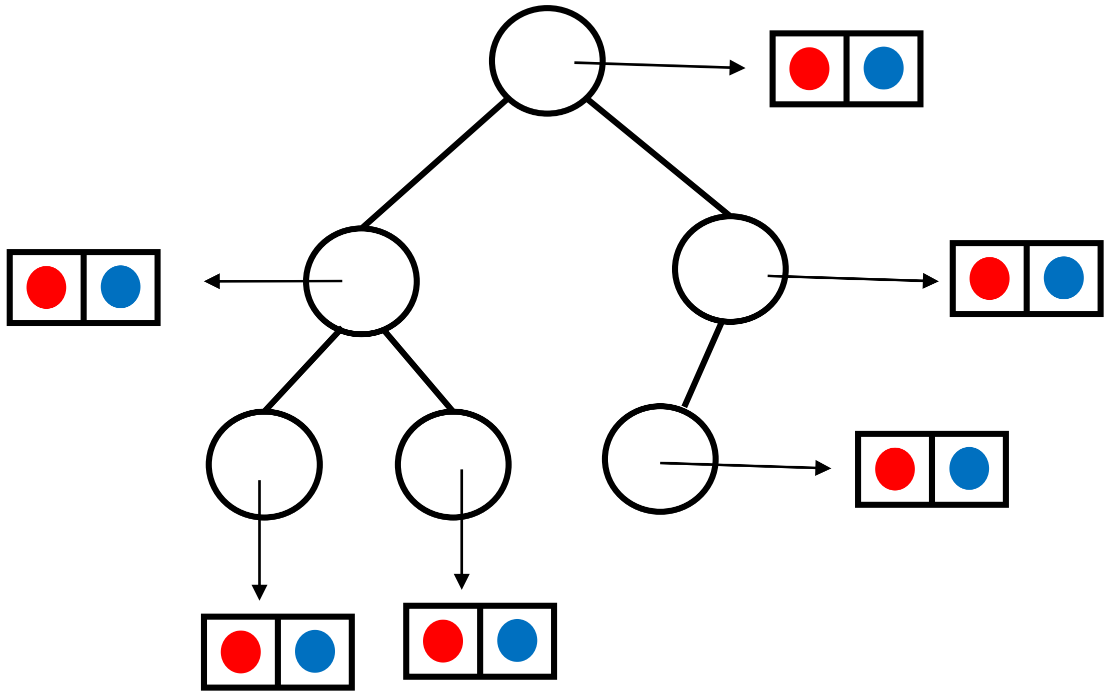
put(key,value), get(key), remove(key) ?

Binary Search Tree (sorted by key)



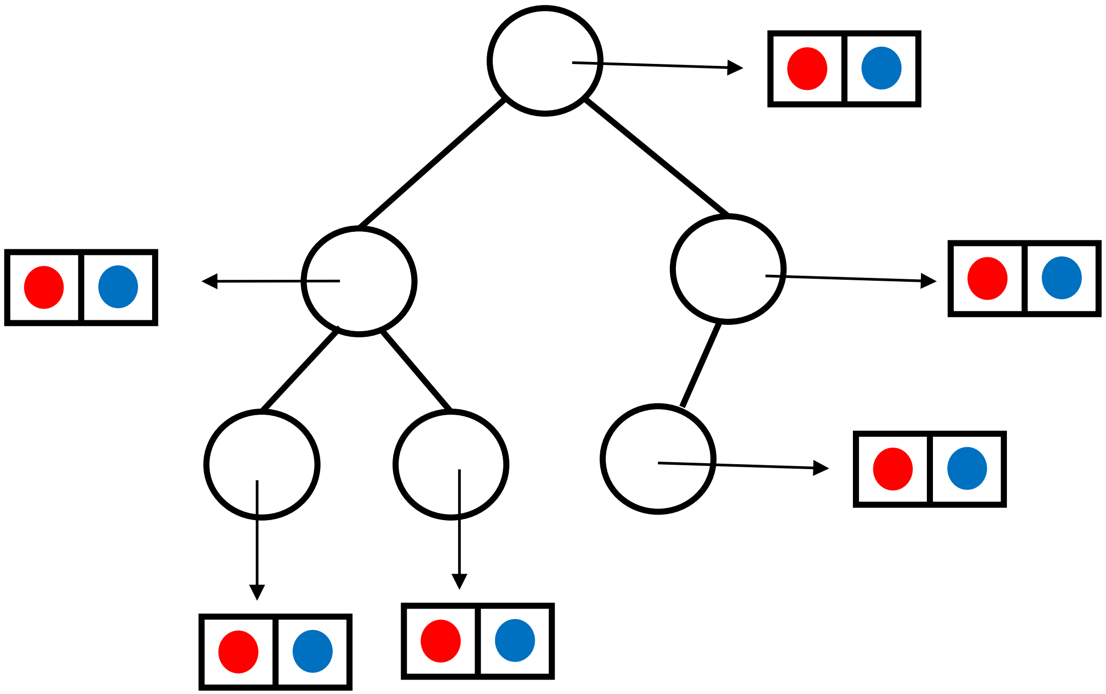
performance for put(key,values), get(key), remove(key) would depend on the tree. If it is is balanced, then these operations would all take time O(log n) in worst case.

minHeap (priority defined by key)



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

minHeap (priority defined by key)



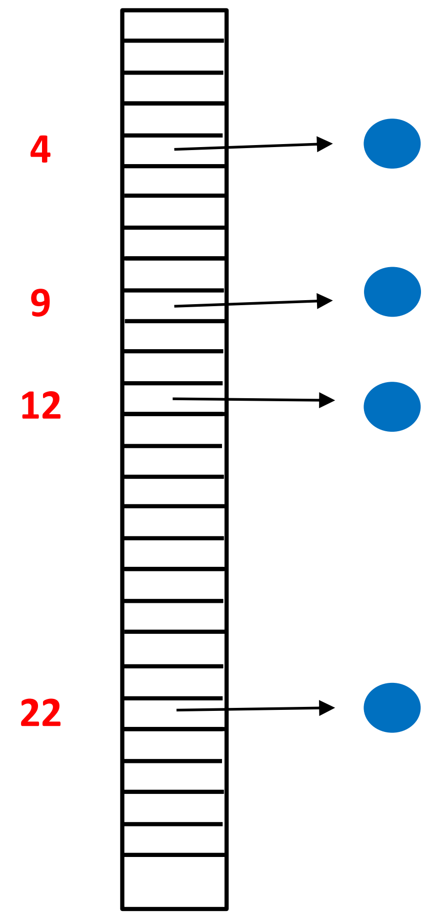
put(key, value) would take time O(log n).
get(key) would require traversing the tree -- O(n), not good.
remove(key) would be awkward. Why ?

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

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

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

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



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

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

General Case:

Keys might be not be positive integers.
e.g. Keys might be strings or some other type.

Keys might not even be comparable!

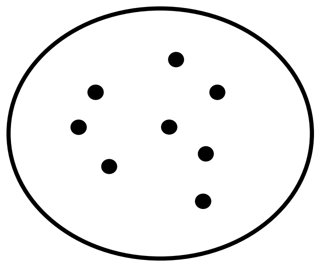
What to do?

Define a map from keys to *large* range of positive integers.

Such a map is called a *hash code*.

Recall (Giulia) briefly introduced the Java Object.hashCode() method in lecture 7.

Java's Object.hashcode() method



1-to-1

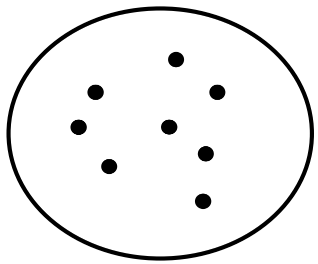
$$\xrightarrow[\text{(not many-to-1)}]{} \quad \{\, 0, 1, 2, \ldots, 2^{24} - 1\}$$

(not many-to-1)

objects in a Java program (runtime)

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

Java's Object.hashcode() method



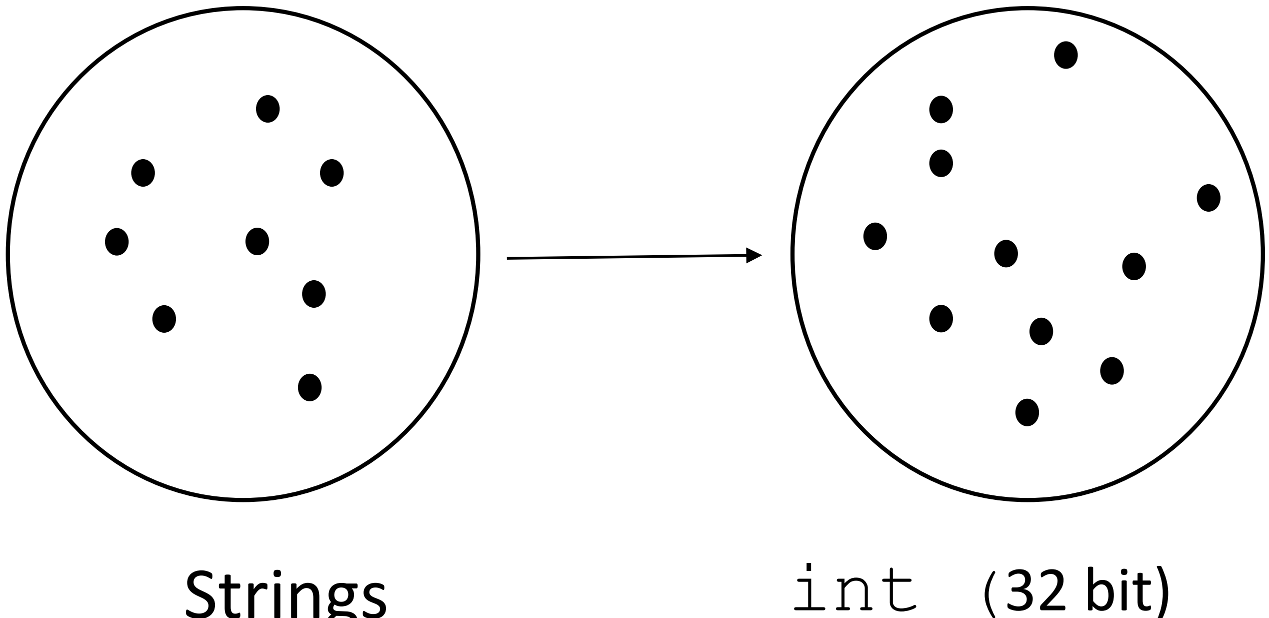
$\xrightarrow[\text{(not many-to-1)}]{\text{1-to-1}}$ $\{\,0, 1, 2, \ldots, 2^{24} - 1\}$

objects in a Java program (runtime)

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

If a class doesn't override Object.hashCode() then "obj1.hashcode() == obj2.hashcode()" is equivalent to "obj1 == obj2".

Java's String.hashcode()



For each String, it defines an integer.

From Giulia's lecture 7 slides:

EXAMPLE

The method `hashCode()` from the class `String`

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.

https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

Example of simple hash code for strings

(***not*** used by Java String class)

Unicode (16 bit)

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

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

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

Java’s String.hashcode()

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

where $n = s.length$ and x = 31.

Usually written instead like this:

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

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

‘e’	‘a’	‘t’
s[0]	s[1]	s[2]

s.length = 3

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

e.g. s = “ate” then s.hashcode() = 97 * 31^2 + 116 * 31 + 101

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

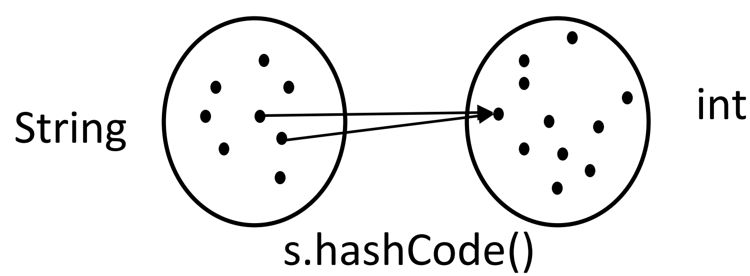
$$\text{s.hashCode()} \equiv \sum_{i=0}^{s.length-1} s[i] * (31)^{s.length - 1 - i}$$

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

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

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

Answer: Not much! It may be either true or false.



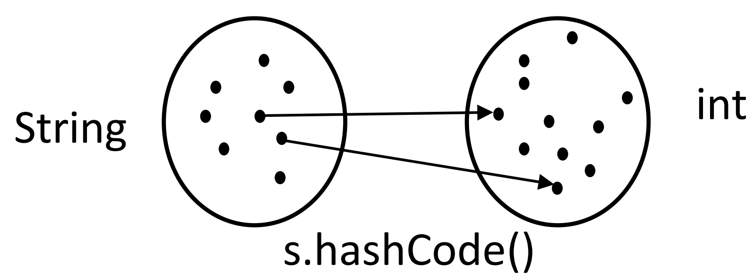
$$\text{s.hashCode()} \quad \equiv \sum_{i=0}^{s.length-1} s[i] * (31)^{s.length - 1 - i}$$

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

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

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

Answer: s1.equals(s2) must be 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).

Next lecture: hash maps (& hash tables)