

# COMP 250

## INTRODUCTION TO COMPUTER SCIENCE

32 - Maps

Giulia Alberini, Fall 2022

Slides adapted from Michael Langer's

# WHAT ARE WE GOING TO DO IN THIS VIDEO?



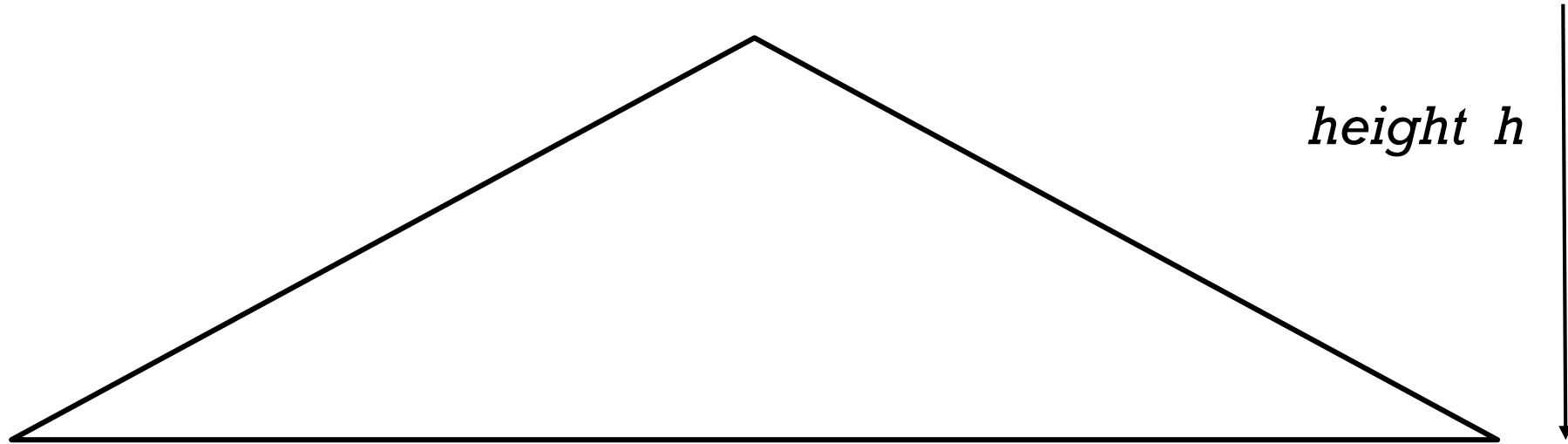
- Recap on heaps
- Maps

# HOW TO BUILD A HEAP

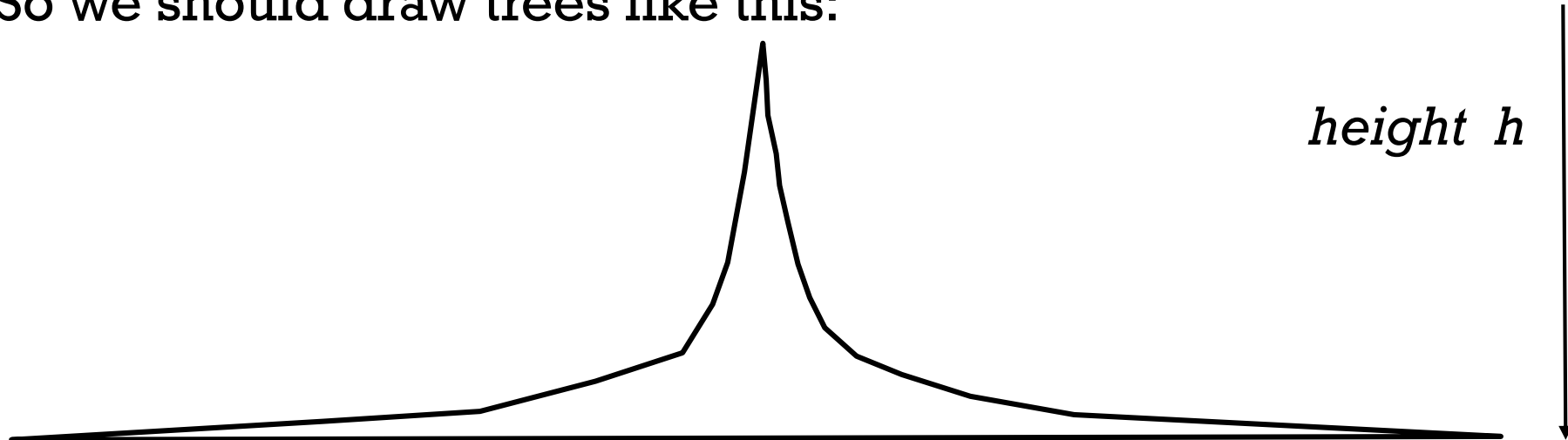
```
buildHeap(list){  
    create new heap array  
    for (k = 0; k < list.size(); k++)  
        add( list[k] )  
}
```

```
buildHeapFast(list){  
    // copy elements from list to heap array  
    for (k = size/2; k >= 1; k--)  
        downHeap( k, size )  
}
```

We tends to draw binary trees like this:

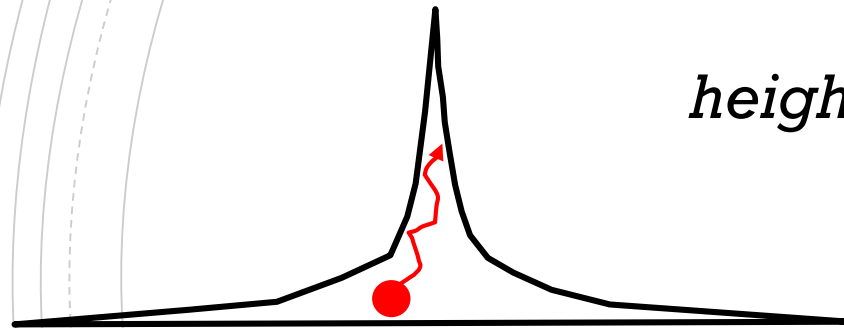


But the number of nodes doubles at each level.  
So we should draw trees like this:



# BUILDHEAP ALGORITHMS

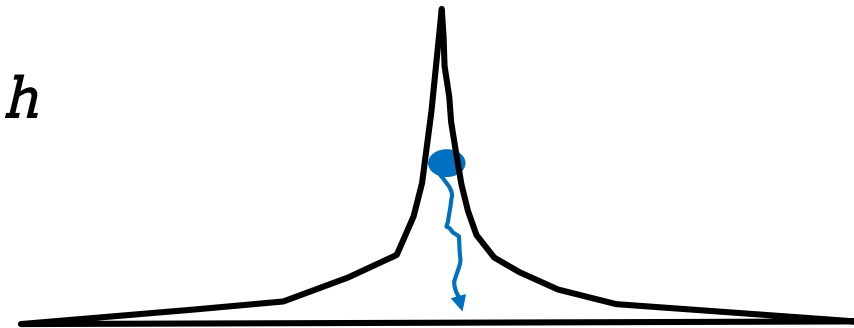
naive



Most nodes swap  $\sim h$   
times in worst case.

height  $h$

fast



Few nodes swap  $\sim h$   
times in worst case.

## HOW TO SHOW BUILDHEAPFAST IS $O(n)$ ?

The worst case number of swaps needed to downHeap node  $i$  is the height of that node.

$$t(n) = \sum_{i=1}^n \text{height of node } i$$

$\frac{1}{2}$  of the nodes do no swaps.

$\frac{1}{4}$  of the nodes do at most one swap.

$\frac{1}{8}$  of the nodes do at most two swaps....

ASSUME THE LAST LEVEL IS FULL

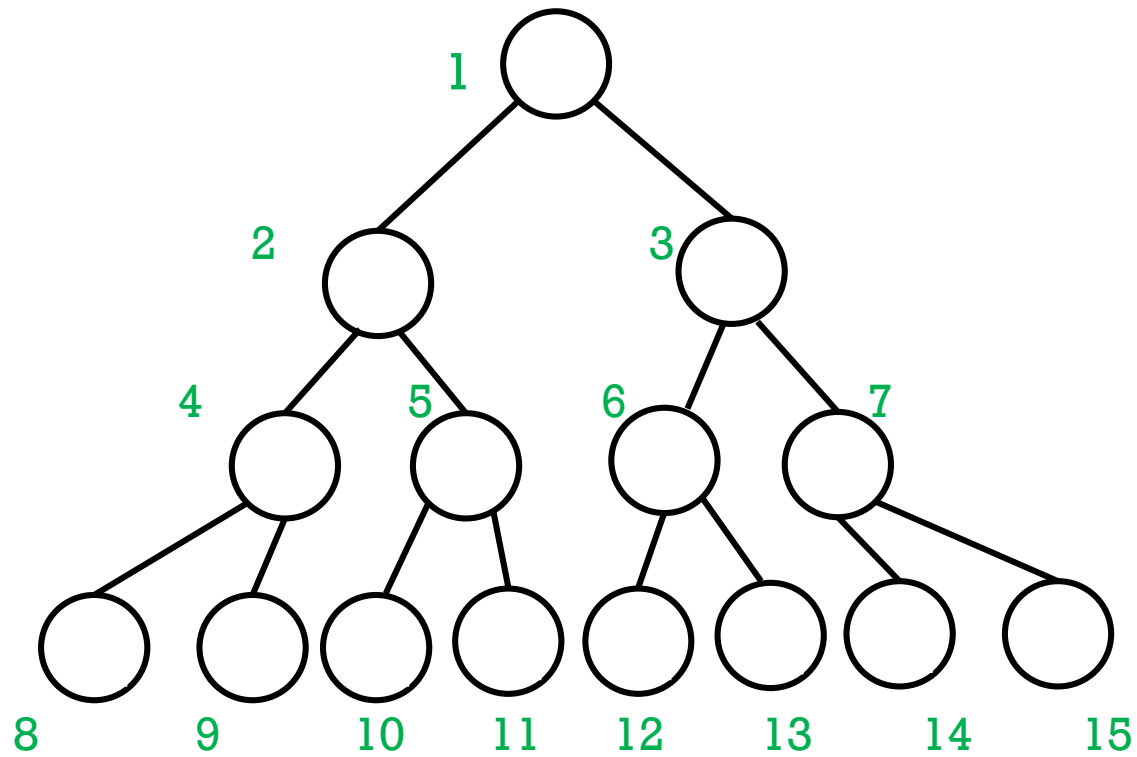
*height*

3

2

1

0



*level*

0

1

2

3

## WORSE CASE OF BUILDHEAPFAST ?

- How many elements at *level*  $l$  ? ( $l \in 0, \dots, h$ )
- What is the height of each *level*  $l$  node?



## WORSE CASE OF BUILDHEAPFAST ?

- How many elements at *level*  $l$  ? ( $l \in 0, \dots, h$ )
  - $2^l$
- What is the height of each *level*  $l$  node?
  - $h - l$

$$t(n) = \sum_{i=1}^n \text{height of node } i$$

$$= ?$$

## WORSE CASE OF BUILDHEAPFAST ?

- How many elements at *level*  $l$  ? ( $l \in 0, \dots, h$ )
  - $2^l$
- What is the height of each *level*  $l$  node?
  - $h - l$

$$\begin{aligned} t(n) &= \sum_{i=1}^n \text{height of node } i \\ &= \sum_{l=0}^h (h - l) 2^l \end{aligned}$$

$$\begin{aligned}
 t_{worstcase}(h) &= \sum_{l=0}^h (h-l) 2^l \\
 &= h \sum_{l=0}^h 2^l - \sum_{l=0}^h l 2^l
 \end{aligned}$$



Easy



Difficult

(number of nodes)

(sum of node levels)

$$t_{worstcase}(h) = \sum_{l=0}^h (h-l) 2^l$$

$$= h \sum_{l=0}^h 2^l - \sum_{l=0}^h l 2^l$$

(See next slide)

$$= h(2^{h+1} - 1) - (h-1)2^{h+1} - 2$$

$$\sum_{l=0}^h l 2^l = \sum_{l=0}^h l (2^{l+1} - 2^l) \quad (\text{trick})$$

$$= \sum_{l=0}^h l 2^{l+1} - \sum_{l=0}^h l 2^l$$

$$= \sum_{l=0}^h l 2^{l+1} - \sum_{l=0}^{h-1} (l+1) 2^{l+1}$$

Second term index  
goes to h-1 only

$$= h 2^{h+1} + 2 \sum_{l=0}^{h-1} (l - (l+1)) 2^l$$

$$= h 2^{h+1} - 2 \sum_{l=0}^{h-1} 2^l$$

$$= h 2^{h+1} - 2(2^h - 1)$$

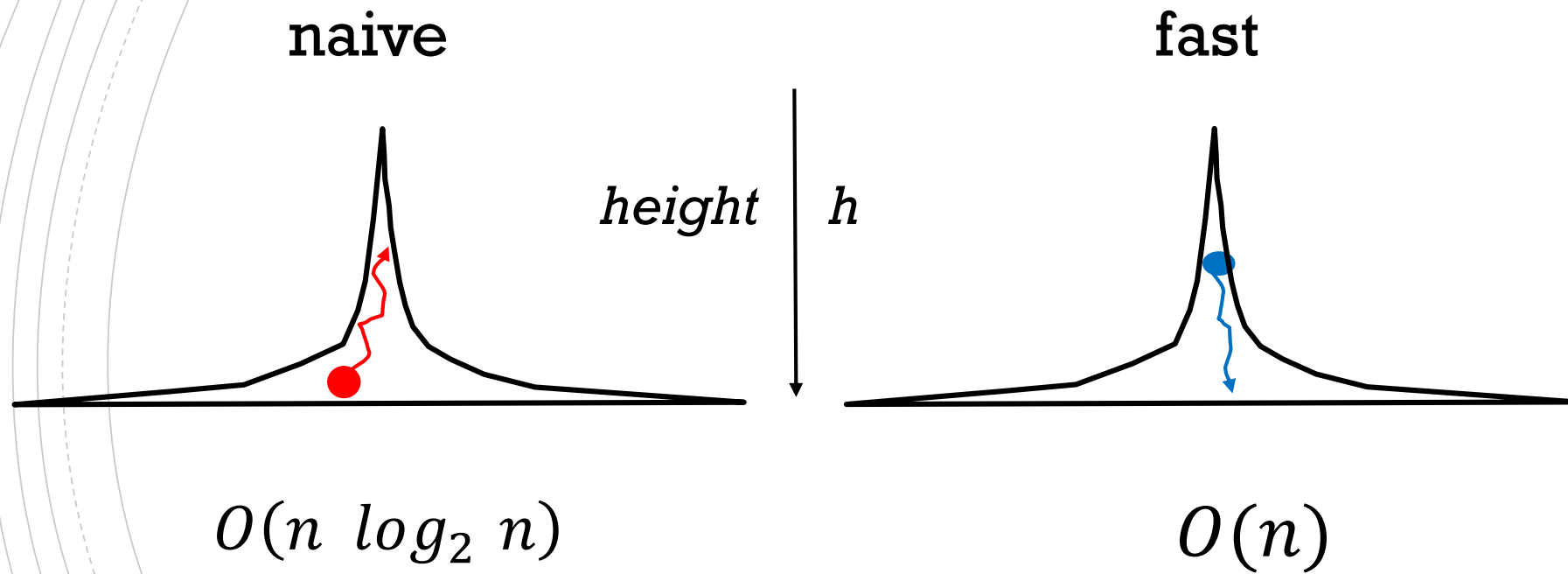
$$= (h-1)2^{h+1} + 2$$

$$\begin{aligned}
t_{worstcase}(h) &= \sum_{l=0}^h (h-l) 2^l \\
&= h \sum_{l=0}^h 2^l - \sum_{l=0}^h l 2^l \\
&= h(2^{h+1} - 1) - (h-1)2^{h+1} - 2 \quad \text{from above} \\
&= 2^{h+1} - h - 2
\end{aligned}$$

**Since**  $n = 2^{h+1} - 1$ , **we get :**

$$t_{worstcase}(n) = n - \log(n+1)$$

## SUMMARY: BUILDHEAP ALGORITHMS

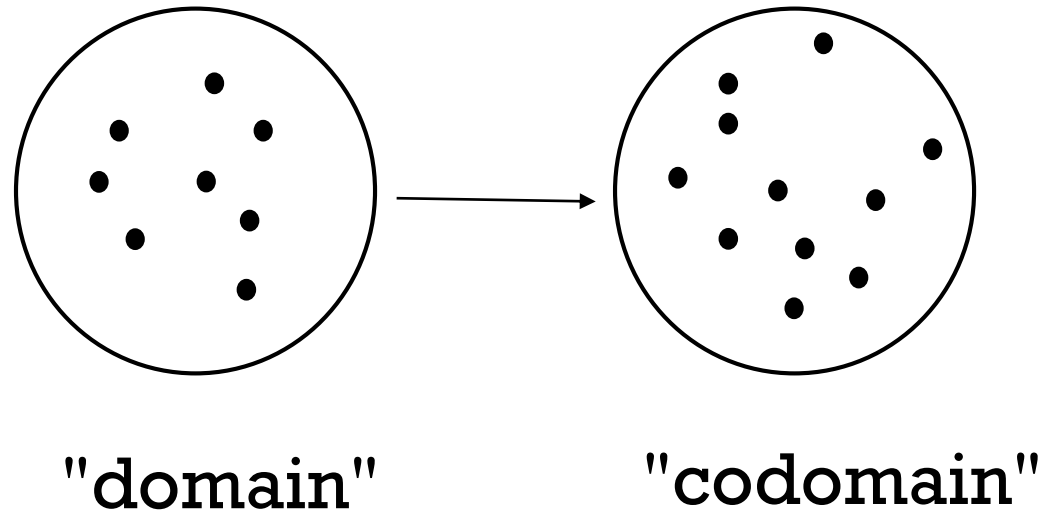


The background features a series of concentric circles in a light gray color, centered around the middle of the frame. A solid dark red rectangle is positioned in the center, containing the word 'MAPS' in white. Below this rectangle is a horizontal white line, followed by another solid dark red rectangle of the same width.

MAPS



## MAP (MATHEMATICS)



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

Each  $x$  in domain maps to exactly one  $f(x)$  in codomain, but it can happen that  $f(x_1) = f(x_2)$  for different  $x_1, x_2$ , i.e. many-to-one.

## FAMILIAR EXAMPLES

Calculus 1 and 2 ("functions"):

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^n$$

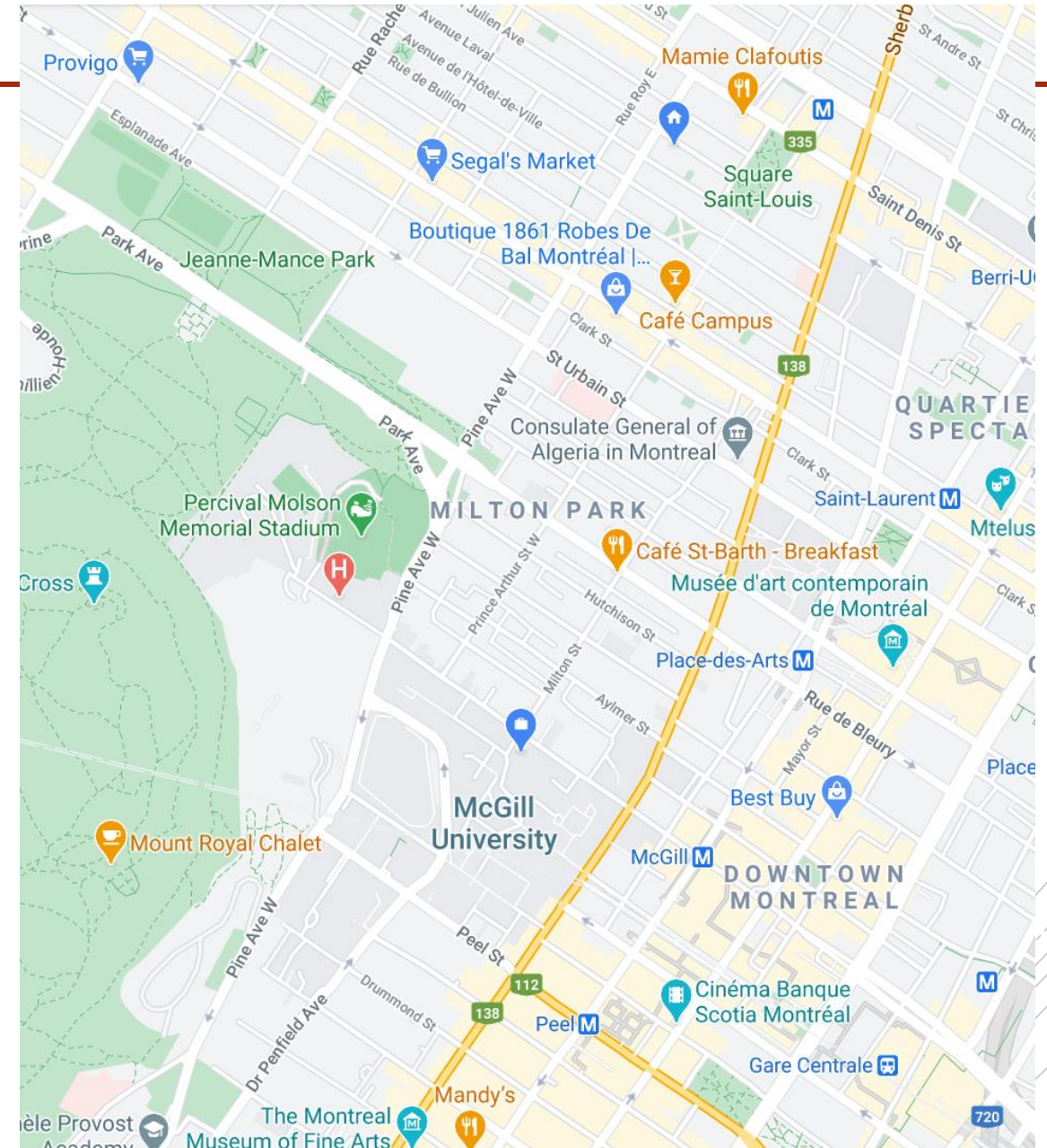
Asymptotic complexity in CS:

$t$  : input size  $\rightarrow$  number of steps in a algorithm.

# MAPS IN EVERYDAY LIFE

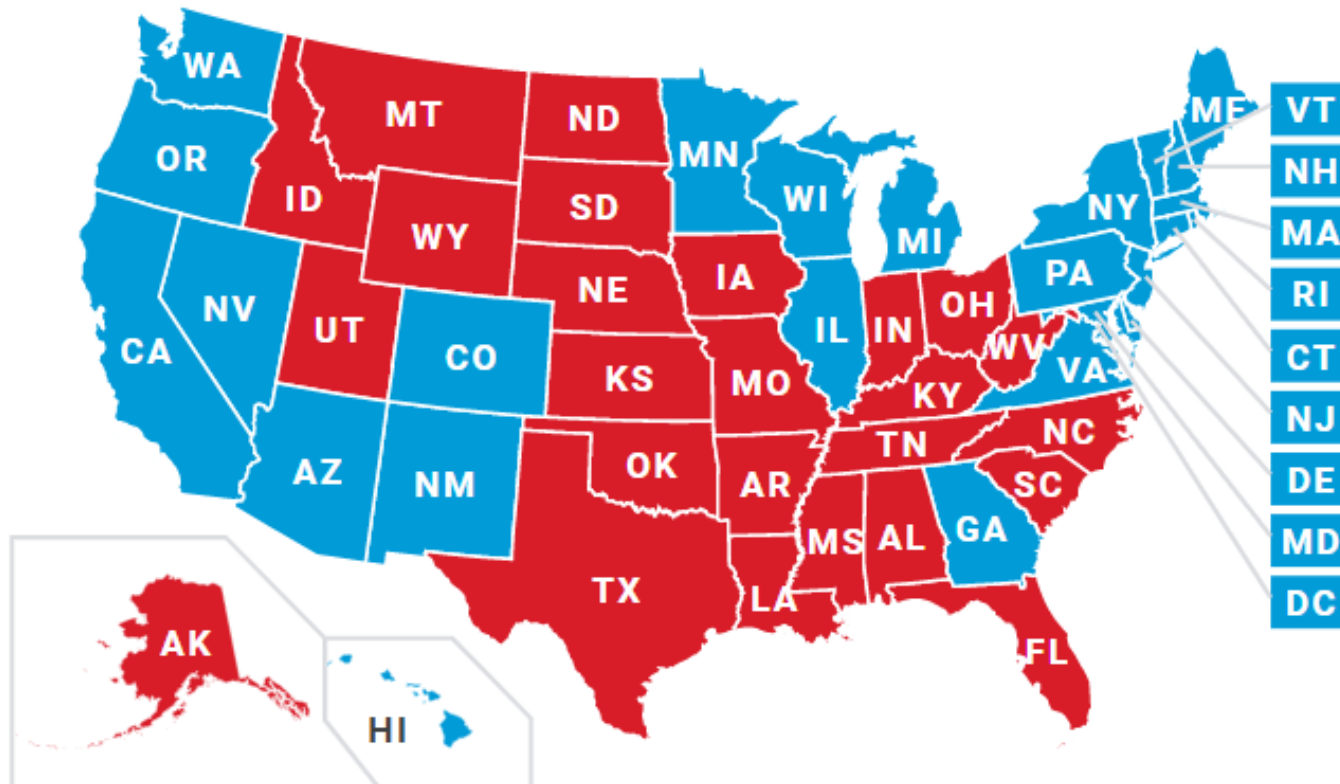
The term "map" commonly refers to a 2D spatial representation of a region of the earth's surface.

$\text{map}(x, y)$  : position in image  $\rightarrow$  position in 2D Montreal



## COLOR MAP

The color map representing the USA election results in 2020.



$\text{vote\_result} : \text{US\_state} \rightarrow \{\text{D}, \text{R}\}$

# RESTAURANT MENU

menu : dish\_name → price

## PLATS

**SAUMON POËLÉ**  
caponata, yogourt, rattes confites,  
épinards, citron  
**27**

**"THE" POUTINE**  
canard confit, champignons,  
oignons sautés au Jack Daniel's,  
Tomme du Haut Richelieu, fromage en grains  
**19.5**

**POULET AU BABEURRE**  
esquites de maïs,  
cotija, coriandre, lime, salade verte  
**27**

## STEAKS

servis avec deux accompagnements

**FILET MIGNON (7 OZ)**  
**BLACK ANGUS "1855"**  
beurre miso/truffe  
**38**

**BAVETTE(8 OZ)**  
**BLACK ANGUS "1855"**  
sauce au poivre  
**33**

**ONGLET(8 OZ)**  
**BLACK ANGUS "1855"**  
mariné au chimichurri  
**33**

## BURGERS

servis avec salade & choix de frites régulières ou de patates douces

**LE DOUBLE CHEESE**  
*Classique*  
2 boulettes de boeuf 4oz, fromage orange,  
sauce secrète du H, oignon rouge, pickle, bacon  
**16**

**GUÉDILLE DE HOMARD**  
1/2 HOMARD  
céleri, persil, oignons verts, mayo  
**22**

**LE BANH MI**  
haut de cuisse de poulet frits, légumes marinés,  
basilic thaï, mayo Sriracha,  
**18**  
*aussi offert en version  
végétarienne (tofu skin)*

**LE MONTIGNAC 2.0**  
boulette cerf 8oz, oignons caramélisés au Jack Daniel,  
bacon, Gruyère suisse, sauce BBQ,  
mayo moutarde à l'ancienne, rondelles d'oignons du H  
**19**

## ACCOMPAGNEMENTS

**POMME DE TERRE ALIGOT**  
purée, crème, cheddar vieilli  
**9**  
**FRITES PATATES DOUCE  
& MAYO**  
**7**

**FRITES & MAYO**  
**6**  
**POUTINE**  
sauce et fromage en grain  
**10**  
**CHAMPIGNONS**  
**9**

**SALADE VERTE**  
vinaigrette au gingembre  
**7**  
**RAPINIS AIL ET CITRON**  
**9**



# INDEX IN A BOOK

index : term → list of pages

Exponential inequality 185  
Exponential running time 186,  
661, 911  
Extended Church-Turing the-  
sis 910  
Extensible library 101  
External path length 418, 832

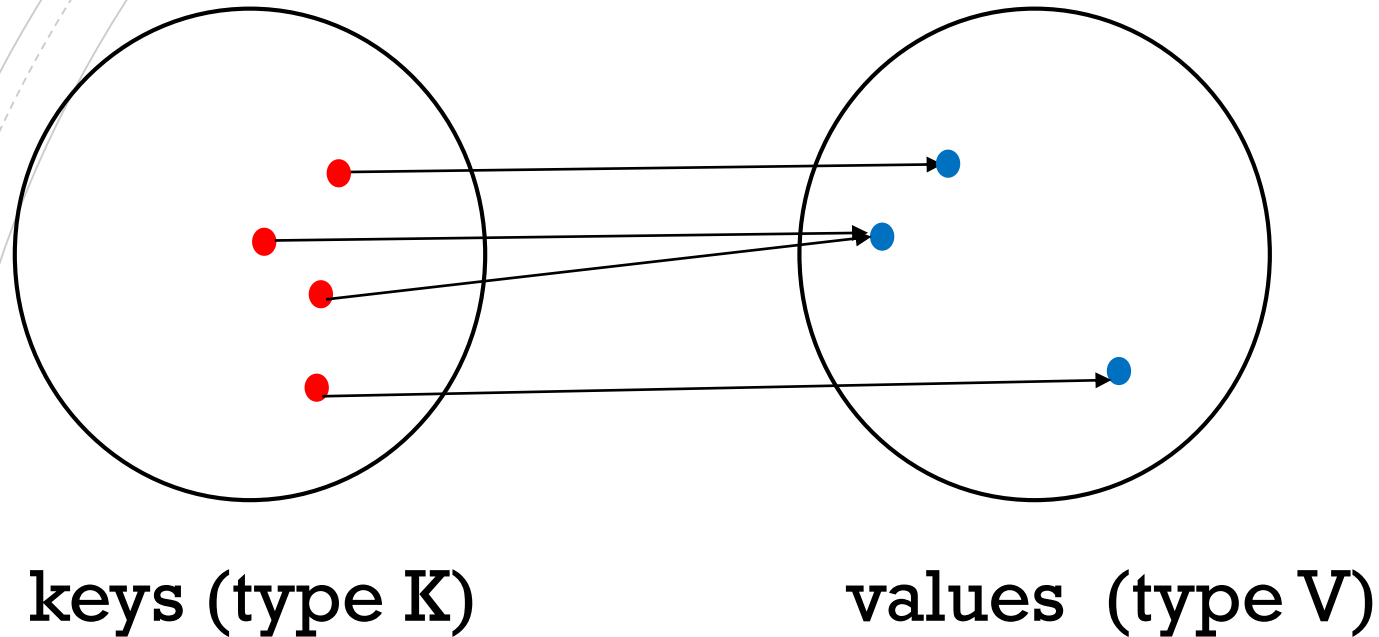
## F

Factor an integer 919  
Factorial function 185  
Fail-fast design 107  
Fail-fast iterator 160, 171  
Farthest pair 210  
Fibonacci heap 628, 682  
Fibonacci numbers 57  
FIFO. *See* First-in first-out policy  
FIFO queue.  
    *See* Queue data type  
File system 493  
Filter 60  
    blacklist 491  
    dedup 490  
    whitelist 8, 491  
Final access modifier 105–106  
Fingerprint search 774–778  
Finite state automaton.

*See also* Maxflow problem  
flow network 888  
inflow and outflow 888  
residual network 895  
st-flow 888  
st-flow network 888  
value 888  
Floyd, R. W. 326  
Floyd's method 327  
for loop 16  
Ford-Fulkerson 891–893  
    analysis of 900  
    maximum-capacity path 901  
    shortest augmenting path 897  
Ford, L. 683  
Foreach loop 138  
    arrays 160  
    strings 160  
Forest  
    graph 520  
    spanning 520  
Forest-of-trees 225  
Formatted output 37  
Fortran language 217  
Fragile base class problem 112  
Frazer, W. 306  
Fredman, M. L. 628  
Function call stack 246, 415

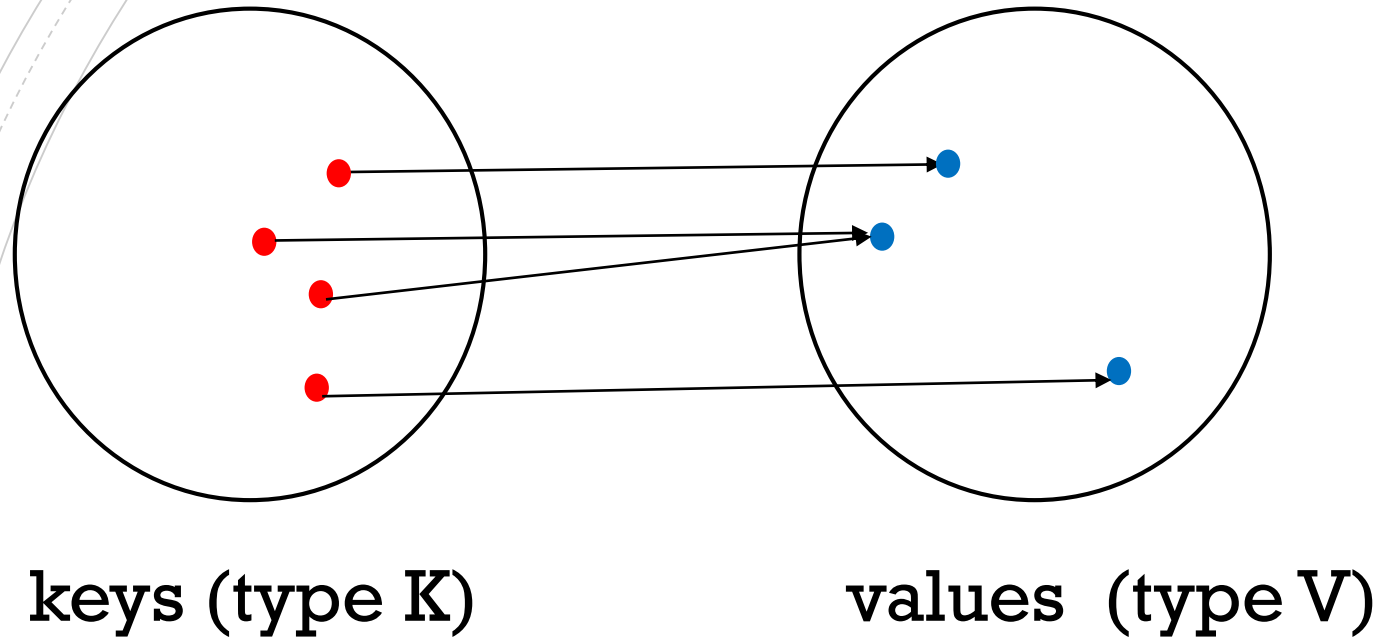
symbol tables 363  
type parameter 122, 134  
Genomics 492, 498  
Geometric data types 76–77  
Geometric sum 185  
getClass() method 101, 103  
Girth of a graph 559  
Global variable 113  
Gosper, R. W. 759  
Graph data type 522–527  
Graph isomorphism 561, 919  
Graph processing 514–693.  
    *See also* Directed graph;  
    *See also* Edge-weighted  
    digraph; *See also* Edge-  
    weighted graph; *See*  
    *also* Undirected graph;  
    *See also* Directed acyclic  
    graph  
    Bellman-Ford 668–681  
    breadth-first search 538–541  
    components 543–546  
    critical-path method 664–666  
    depth-first search 530–537  
    Dijkstra's algorithm 652  
    Kosaraju's algorithm 586–590  
    Kruskal's algorithm 624–627  
    longest paths 811, 812

## MAP (ADT)



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

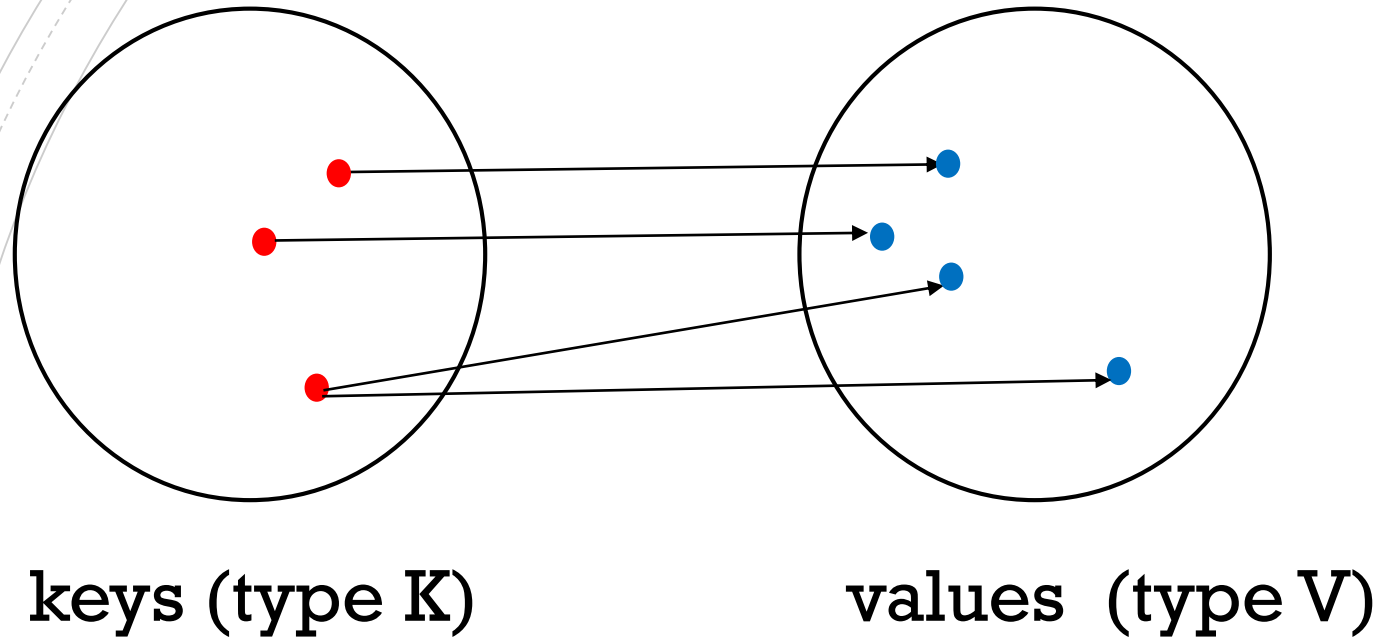
## MAP (ADT)



Note that it is possible for two keys to map to the same value.

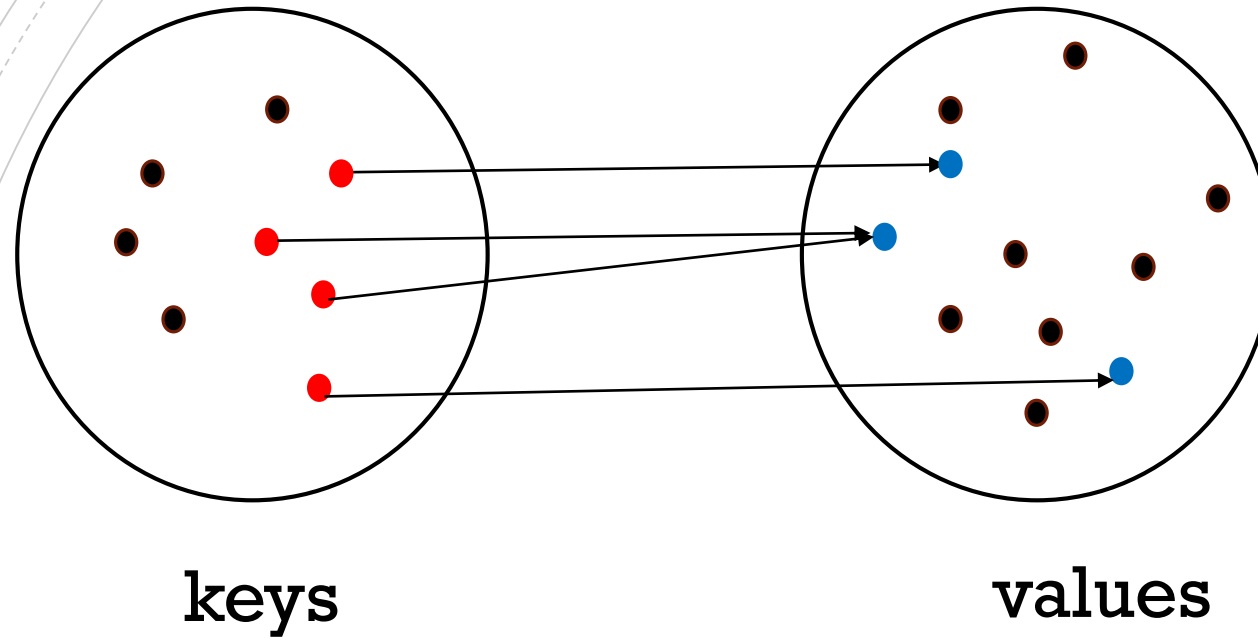


## MAP (ADT)



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

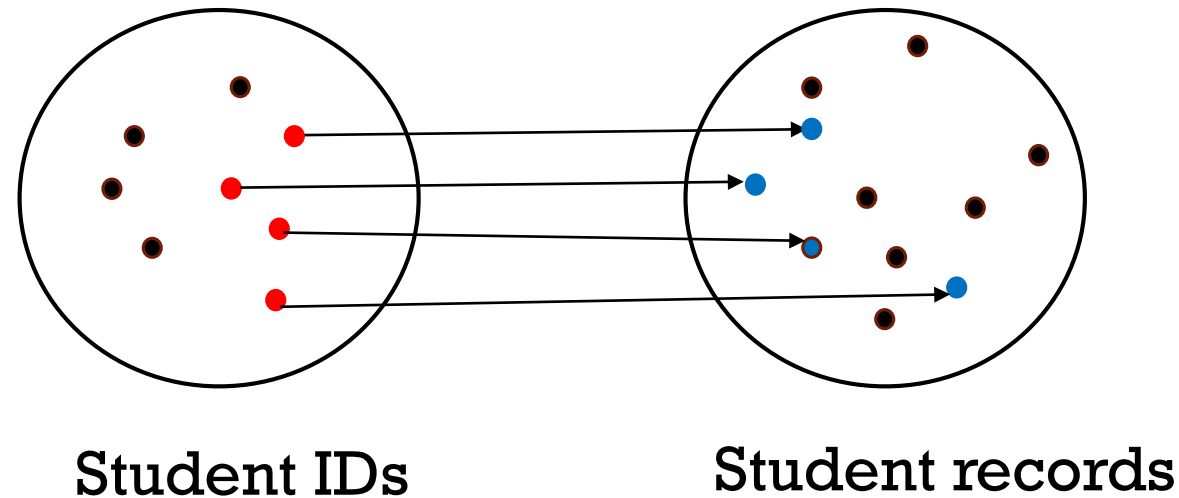
## MAP ENTRIES



The black dots here indicate objects (or potential objects) of type K or V that are *not* in the map.

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 ~650 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 the entry (key, value) to the map. If the map previously contained an entry with key, the old value is replaced by the specified value.

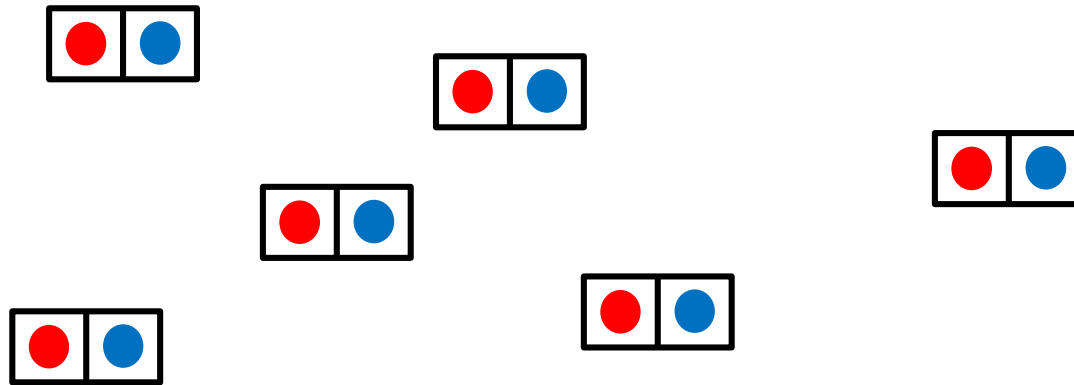
**get(key)**

// Returns the value to which the specified key is mapped. Why not get(key, value) ?

**remove(key)**

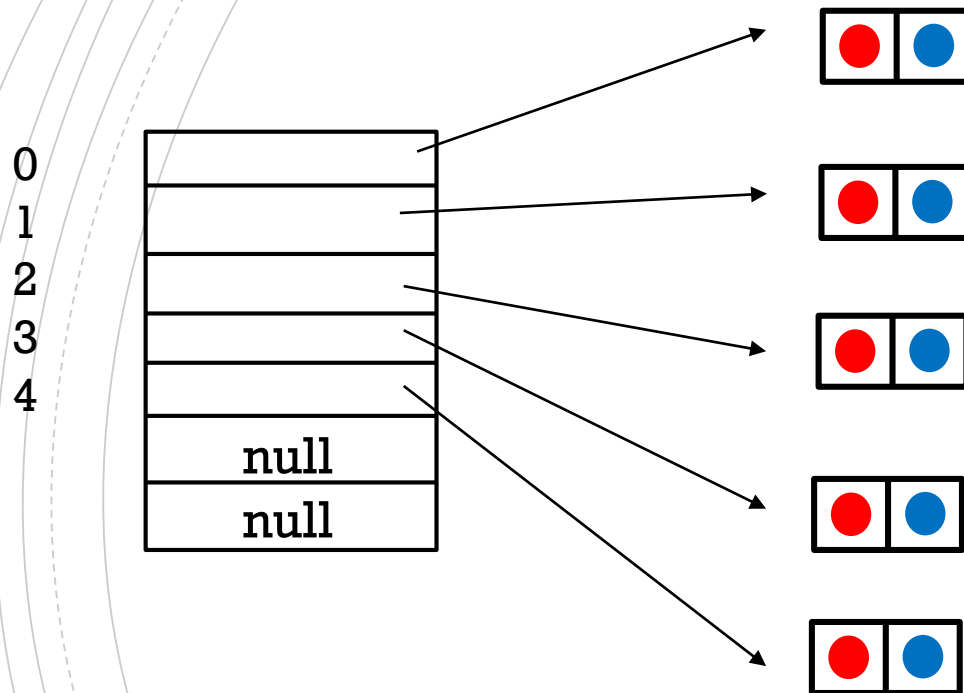
// Removes the entry with the specified key.  
Returns true if the entry was removed, false otherwise.

## DATA STRUCTURES FOR MAPS



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

# ARRAY LIST



How can we implement the following methods?

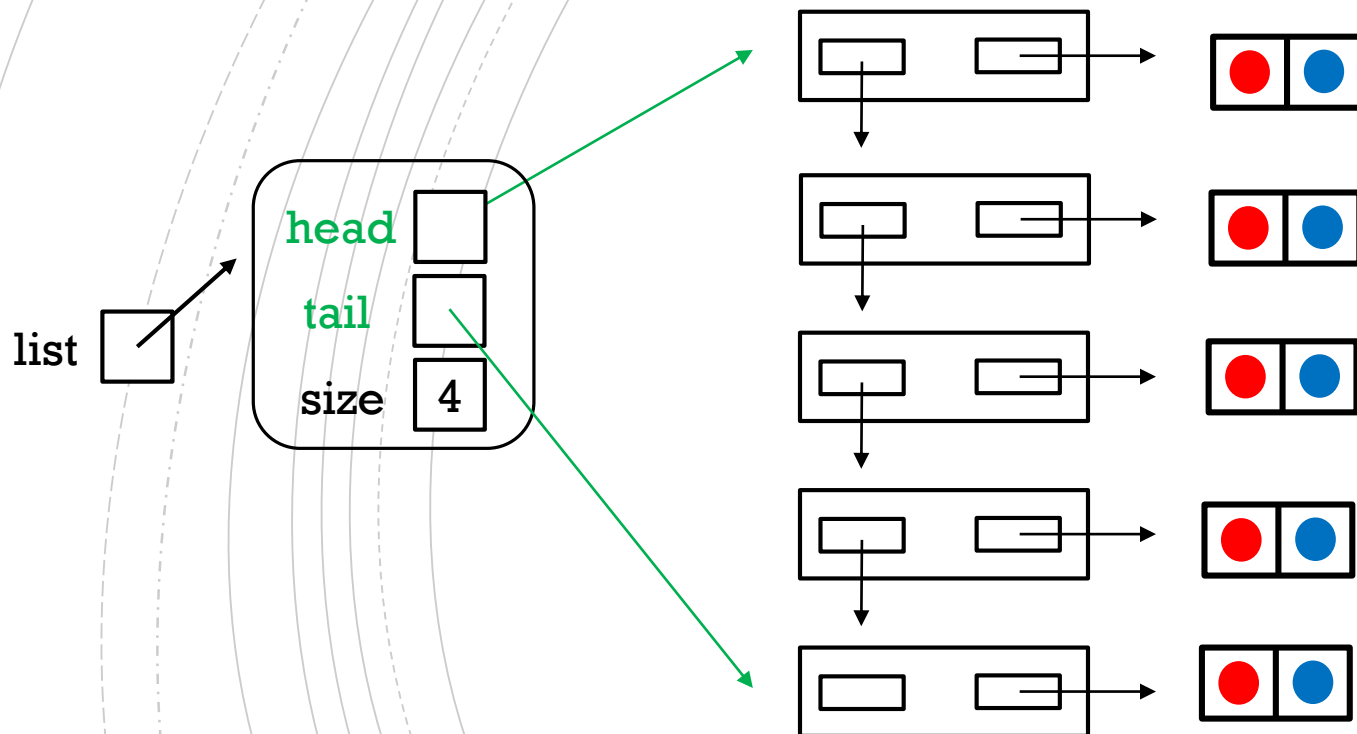
`put( key, value )`

`get(key)`

`remove(key)`

`put()`, `get()` and `remove()` would be  $O(n)$

# SINGLY (OR DOUBLY) LINKED LIST



How can we implement the following methods?

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

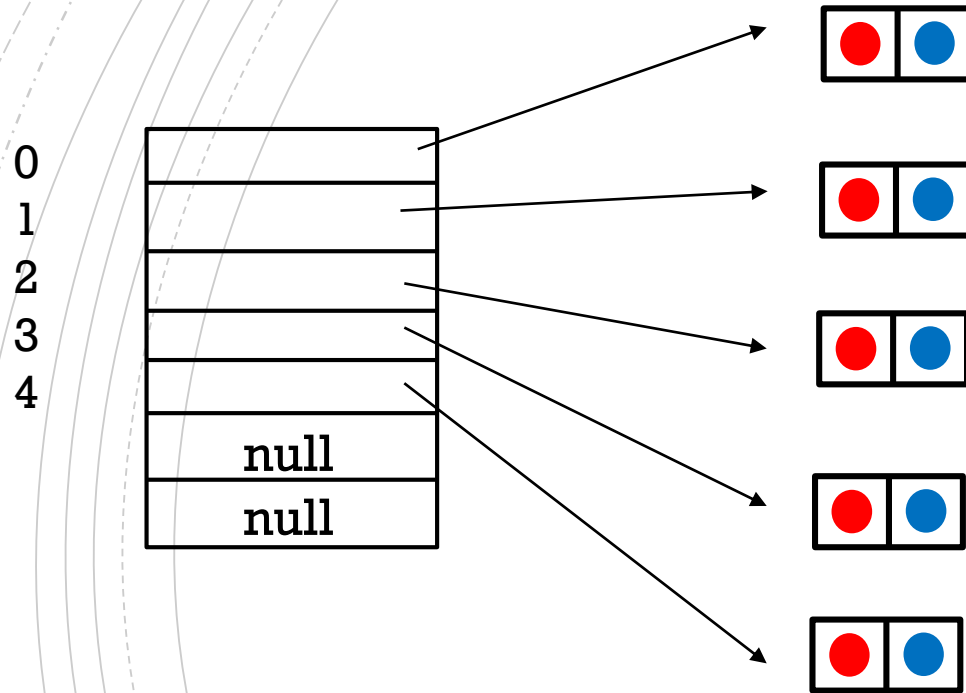
`put()`, `get()` and `remove()` would be  $O(n)$

## LET'S ADD ASSUMPTIONS

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



## ARRAY LIST (SORTED BY KEY)



How can we implement the following methods?

`put( key, value )`

`get(key)`

`remove(key)`

`put()` and `remove()` would be  $O(n)$ ,  
while `get()` could be performed in time  
 $O(\log n)$  using binary search

## BINARY SEARCH TREE (SORTED BY KEY)

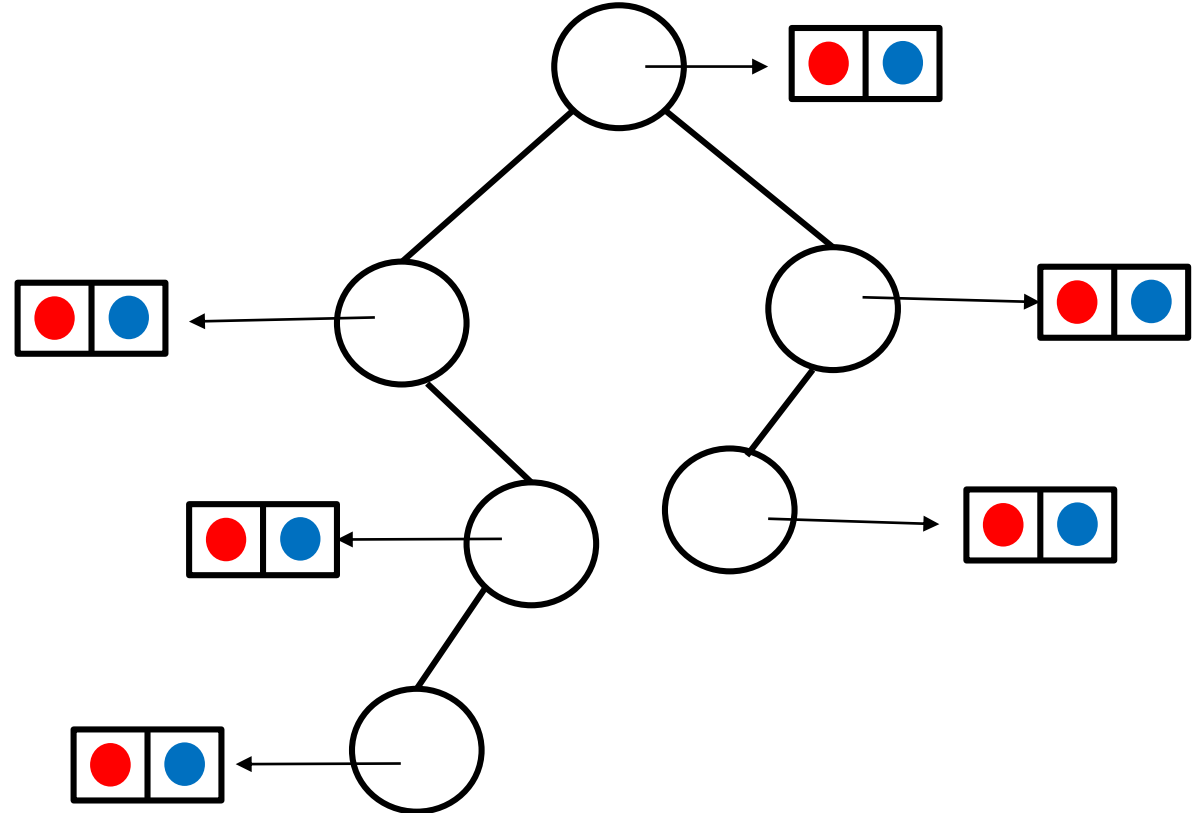
How can we implement the following methods?

`put( key, value )`

`get(key)`

`remove(key)`

The performance of `put()`, `get()` and `remove()` depends on the tree. If we have a balanced tree, then these operations would all take time  $O(\log n)$  in worst case. You will learn more about balanced tree in COMP 251.



## MINHEAP (PRIORITY DEFINED BY KEY)

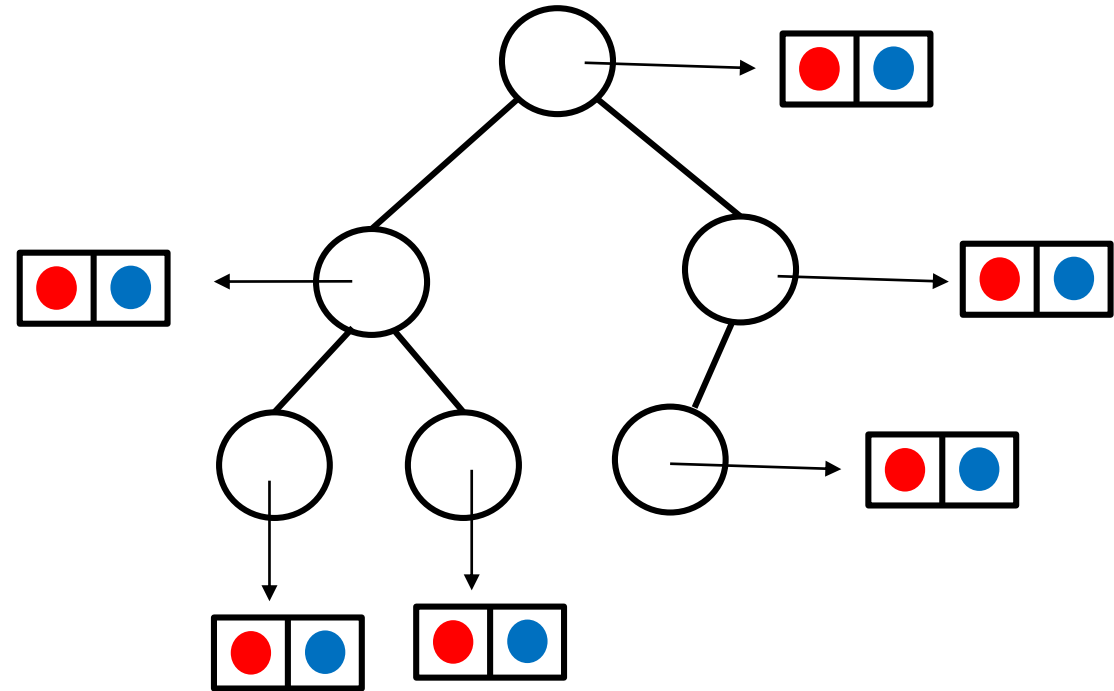
How can we implement the following methods?

`put( key, value )`

`get(key)`

`remove(key)`

The performance of `put()` would be  $O(\log n)$ . Implementing `get()` would require traversing the tree, so it would be  $O(n)$ . Implementing `remove()` would be a little weird for heaps...



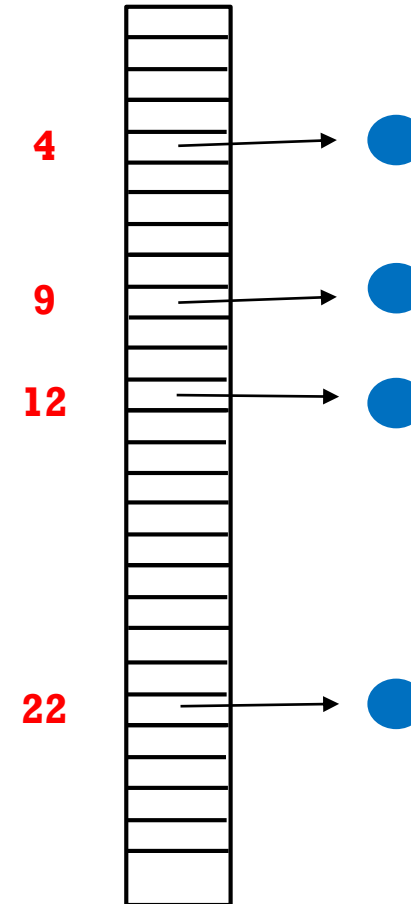
## LET'S ADD ASSUMPTIONS

- ~~Special case #1: what if keys are *comparable*?~~
- Special case #2: what if keys are unique positive integers in small range?

## ARRAYS OF VALUES

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.



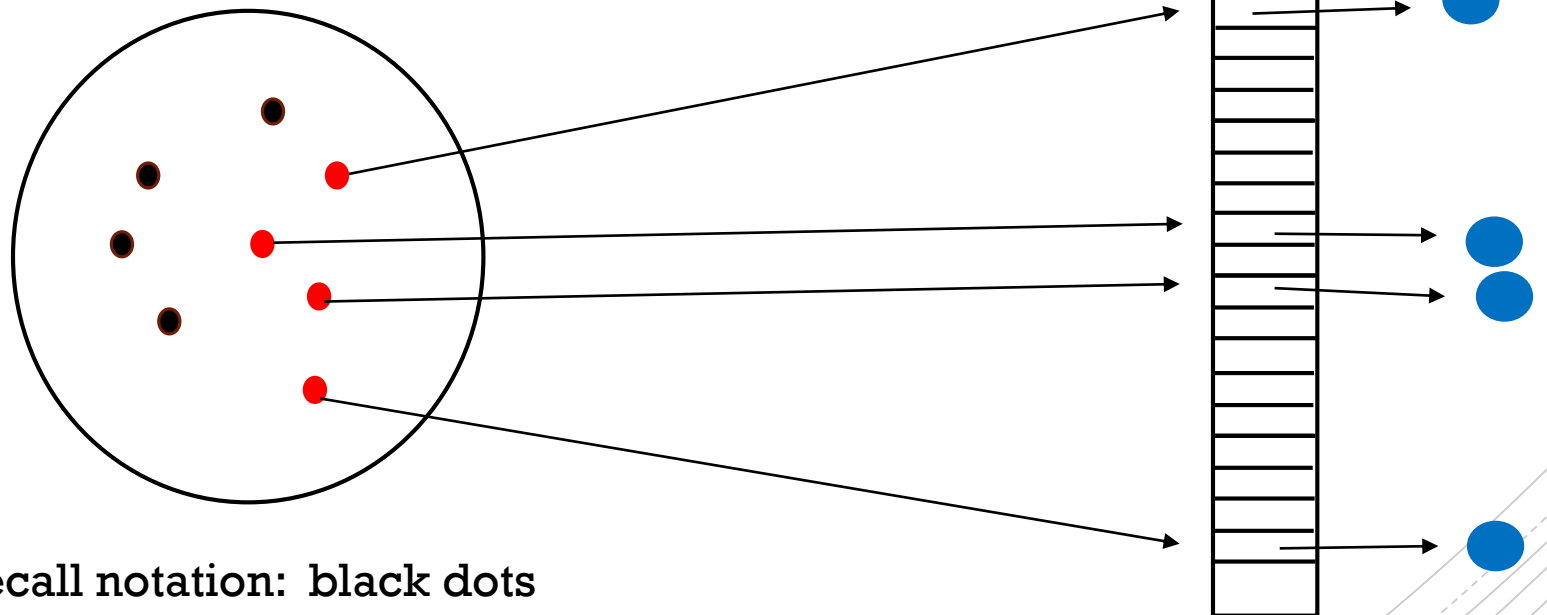
## IN GENERAL

---

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

## STRATEGY IN THE GENERAL CASE

Try to define a map from keys to *small* range of positive integers (array index), and then store the corresponding values in the array.



Recall notation: black dots  
are not part of the map.

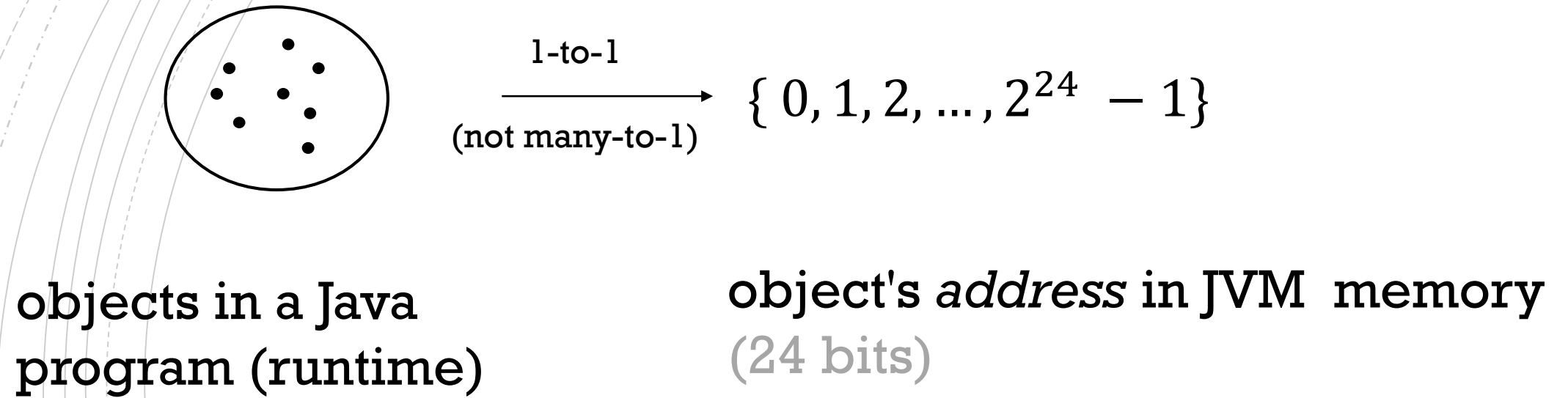
# TODAY

---

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



## JAVA'S `Object.hashCode()`



# JAVA'S String.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.

## EXAMPLE HASH CODE FOR STRINGS

**(not used in Java)**

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

**e.g.**

$$h("eat") = h("ate") = h("tea")$$

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

## JAVA'S `String.hashCode()`

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

where  $x = 31$ .

## JAVA'S `String.hashCode()`

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

where  $x = 31$ .

e.g. `s = "eat"` then `s.hashCode()` =  $101 * 31^2 + 97 * 31 + 116$

'e'

'a'

't'

`s[0]`

`s[1]`

`s[2]`

## JAVA'S `String.hashCode()`

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

where  $x = 31$ .

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

'a'

't'

'e'

`s[0]`

`s[1]`

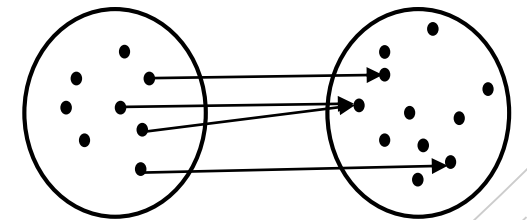
`s[2]`

## JAVA'S String.hashCode ()

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

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

*s1 may or may not be the same string as s2.*



## JAVA'S `String.hashCode()`

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

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

**`s1` is a different string than `s2`.**



The background features a series of concentric circles in the top-left and bottom-right corners. A horizontal line runs across the top, with a paint roller graphic on the right side. The roller has a red handle and a blue frame, with orange paint splatters and drips extending from the orange banner it is painting.

# Coming Soon

**Coming next:**

- Hash Maps