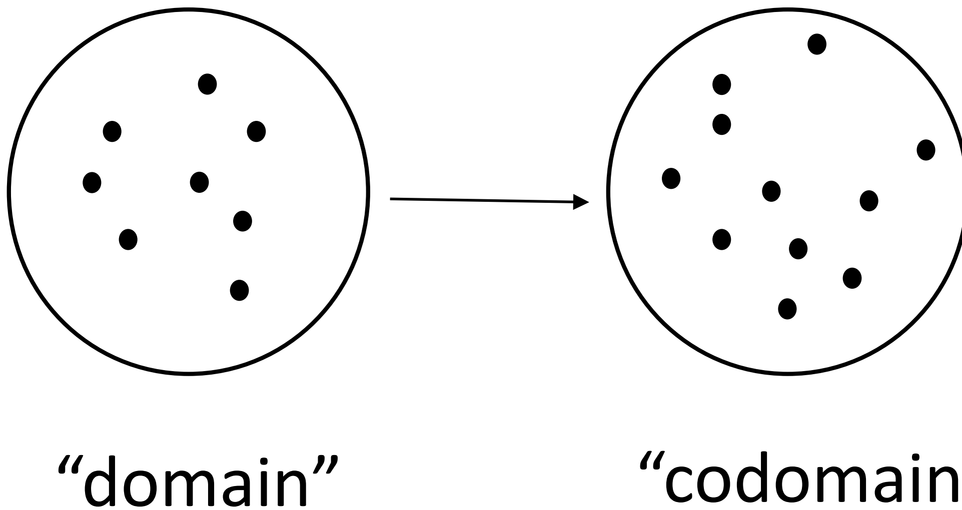
COMP 250

Lecture 26

maps

Nov. 8/9, 2017

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(x1) = f(x2) for different x1, x2, i.e. many-to-one.

Familiar examples

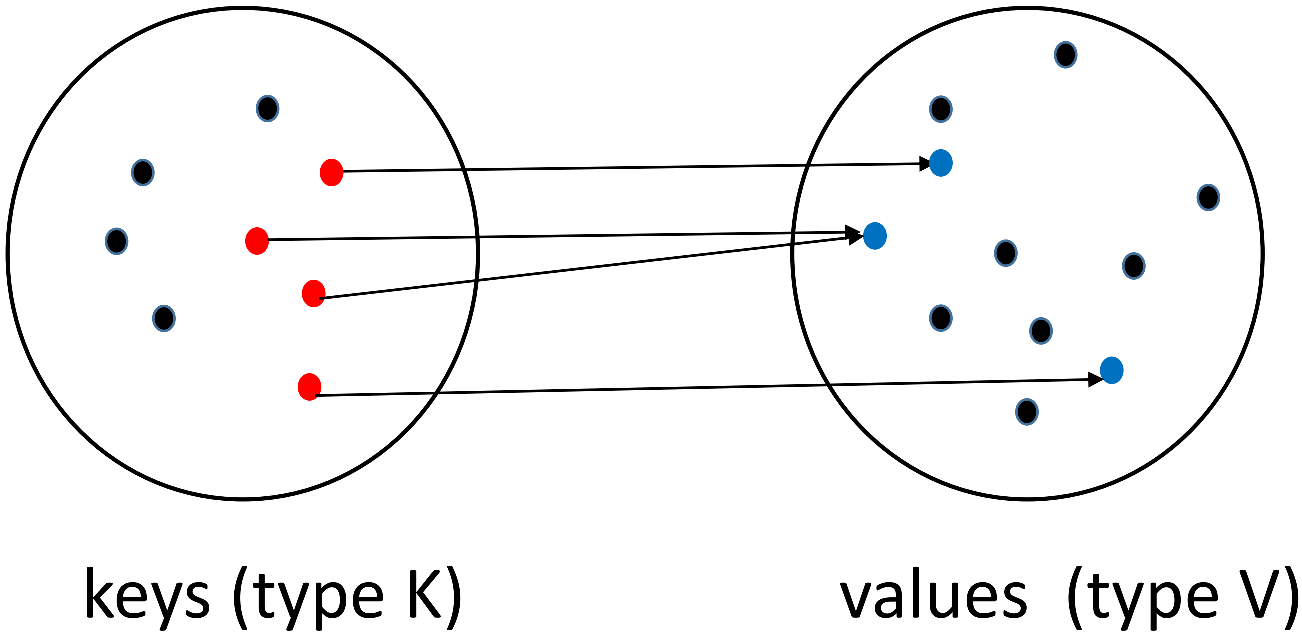
Calculus 1 and 2 (“functions”):

f : real numbers → real numbers

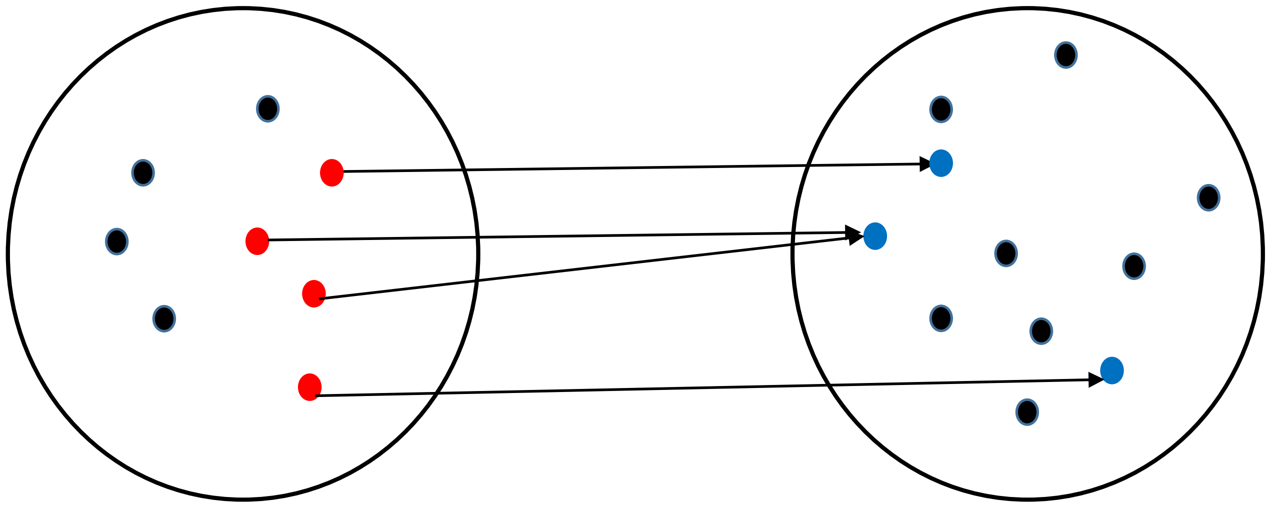
Asymptotic complexity in CS:

t : input size → number of steps in a algorithm.

Map (in COMP 250)



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



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

Map	Keys	Values
Address book		

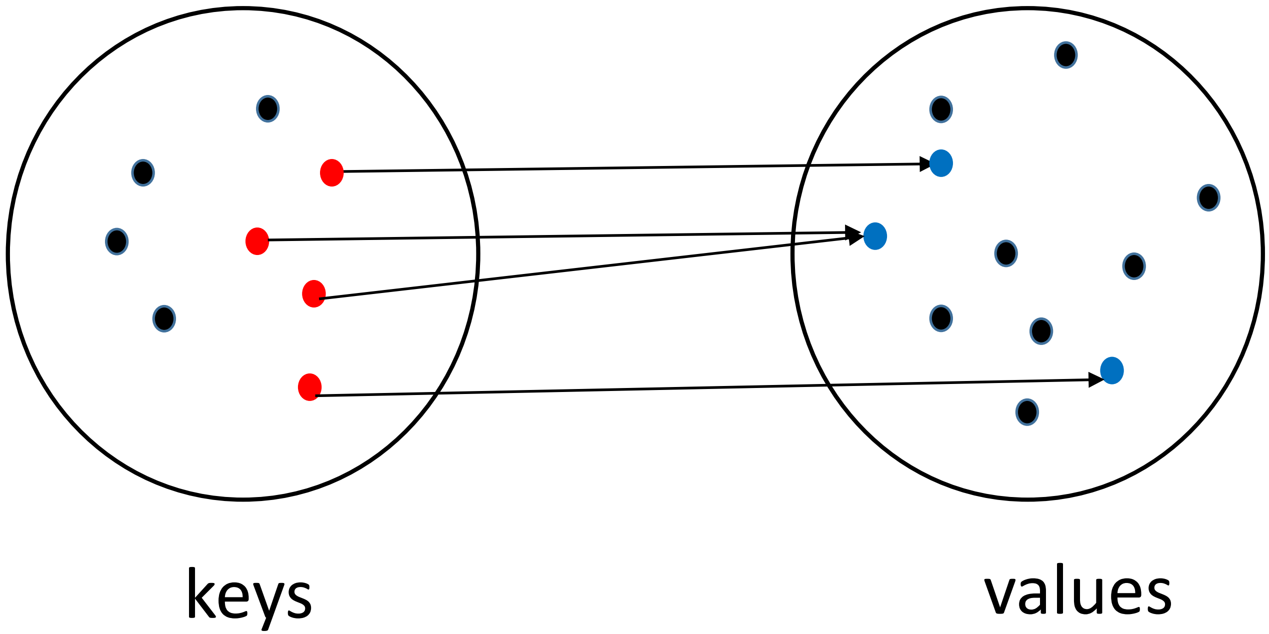
Map	Keys	Values
Address book	Name	Address, email..
Caller ID		

Map	Keys	Values
Address book	Name	Address, email..
Caller ID	Phone #	Name
Student file		

Map	Keys	Values
Address book	Name	Address, email..
Caller ID	Phone #	Name
Student file	ID or Name	Student record
Index at back of book		

Map	Keys	Values
Address book	Name	Address, email..
Caller ID	Phone #	Name
Student file	ID or Name	Student record
Index at back of book	keyword	List of book pages

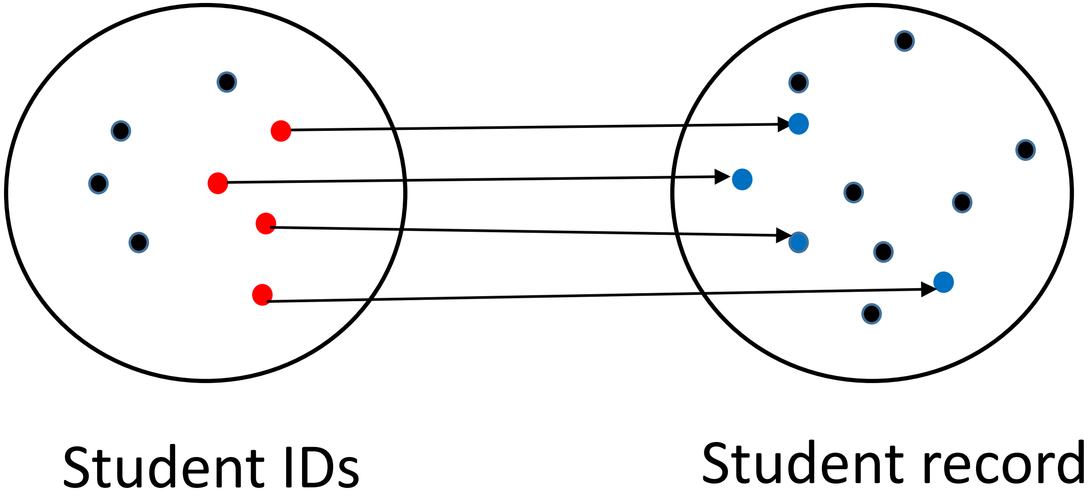
Map Entries



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 ~600 entries.

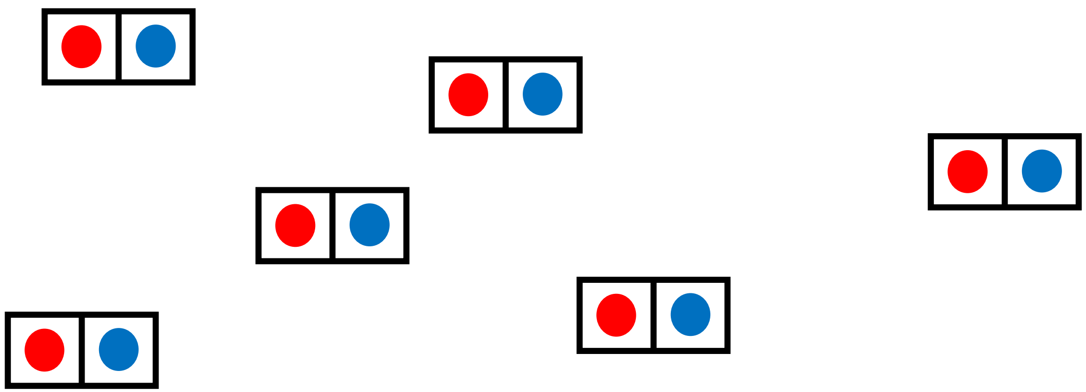
Most McGill students are not taking COMP 250 this semester.

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

Map ADT

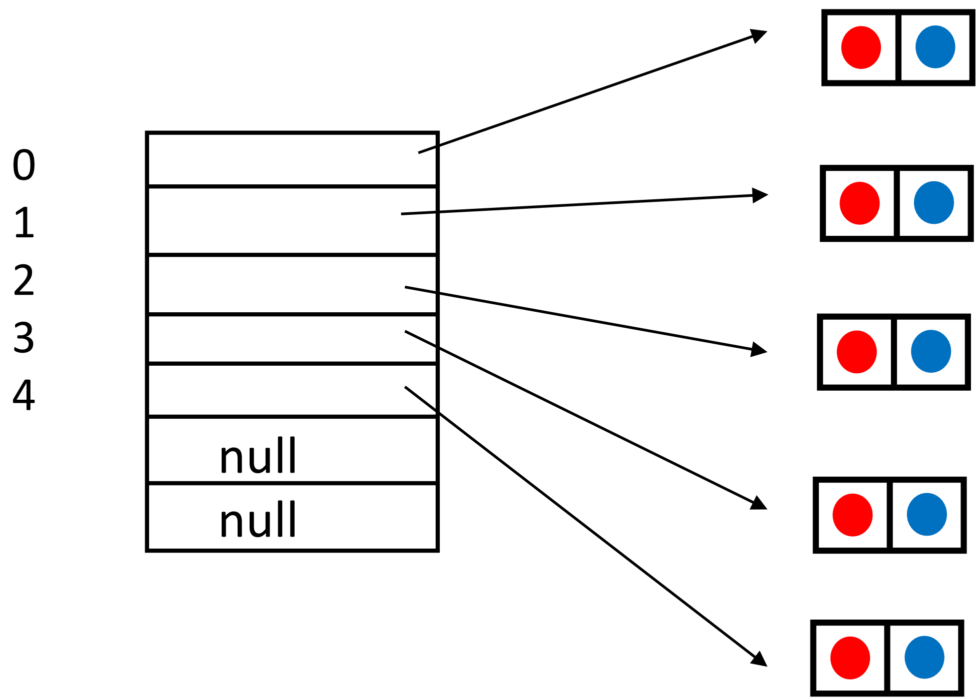
- put(key, value)   // add
- get(key)   //   why not get(key, value) ?
- remove(key)
- …

Data Structures for Maps



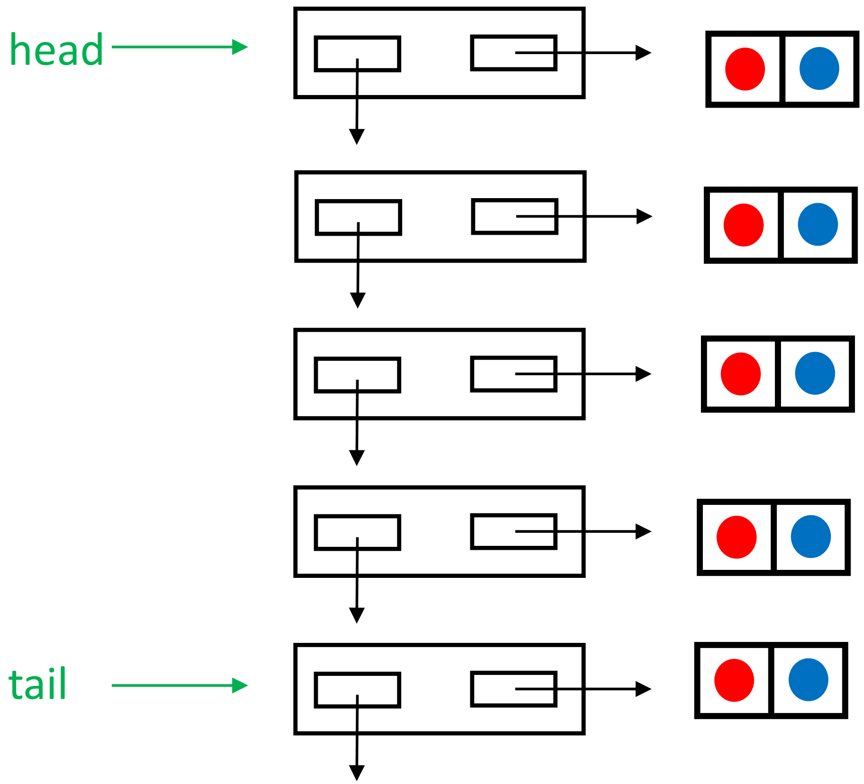
How to organize a set of (key, value) pairs, i.e. entries ?

Array list



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

Singly (or Doubly) linked list



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

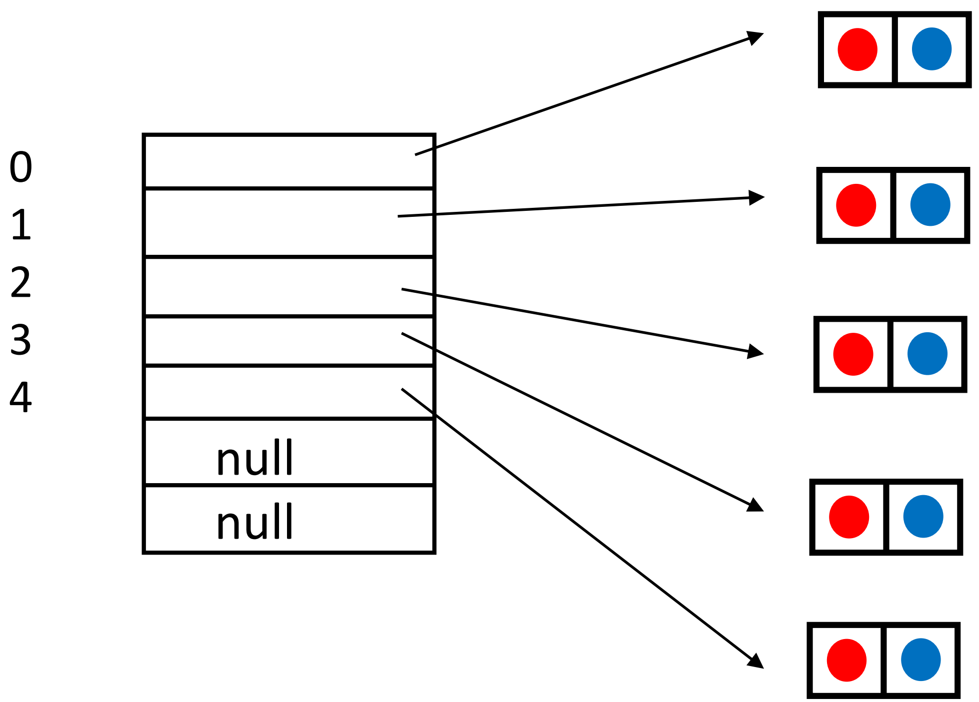
Important property of keys

Two different keys *can* have (map to) the same value.

One key *cannot* have (map to) two values.

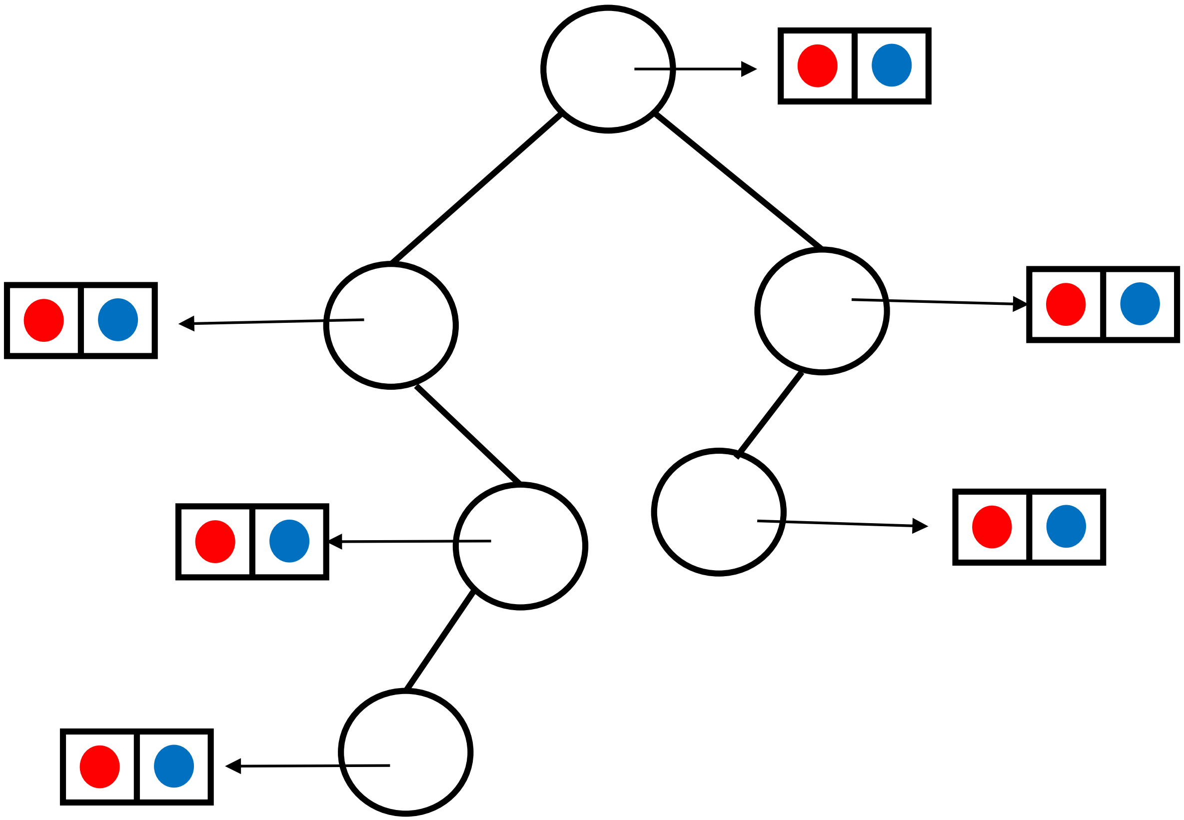
Special case #1: what if keys are *comparable* ?

Array list (sorted by key)



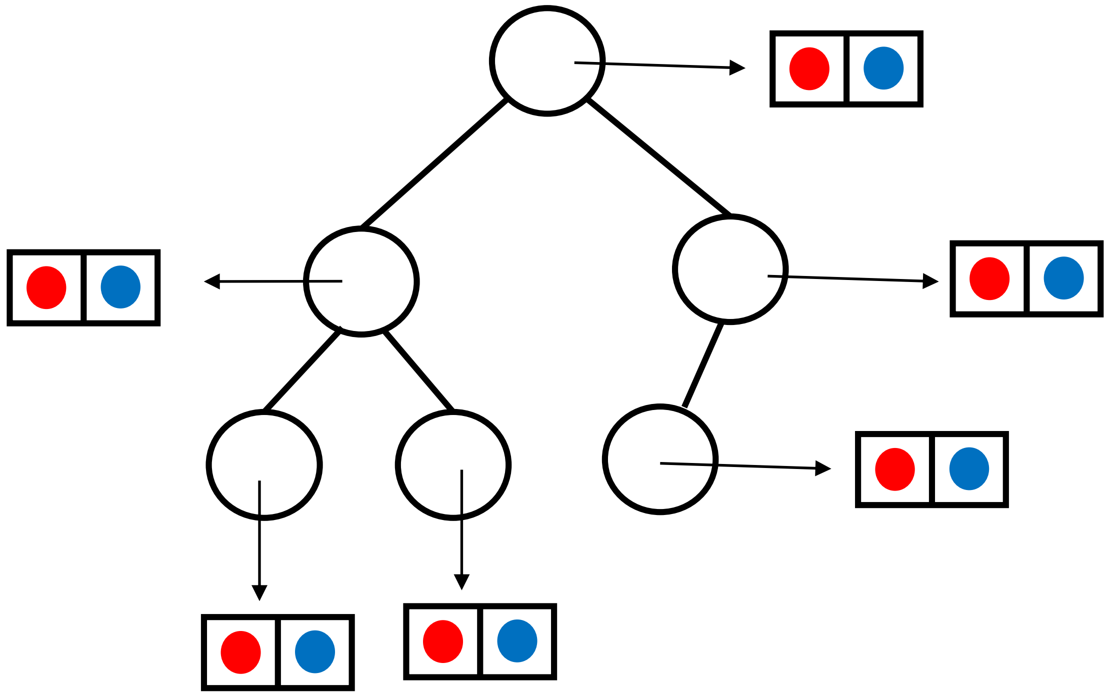
put(key, value)
get(key)
remove(key)

Binary Search Tree (sorted by key)



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

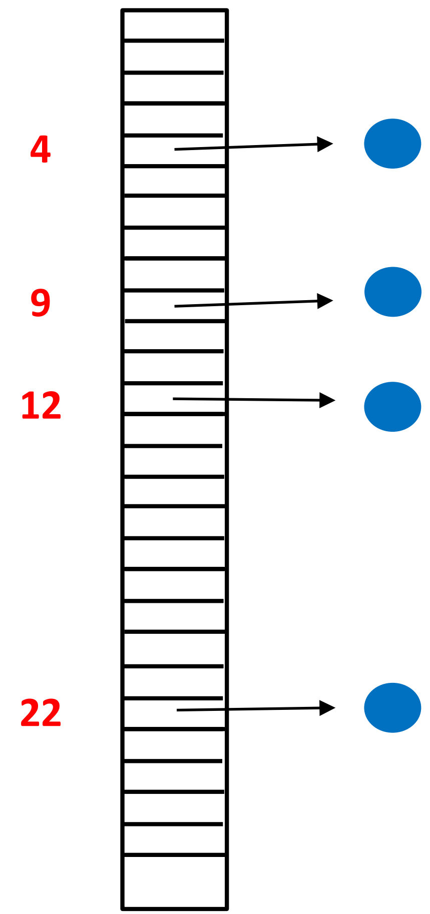
minHeap (priority defined by key)



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

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 ?

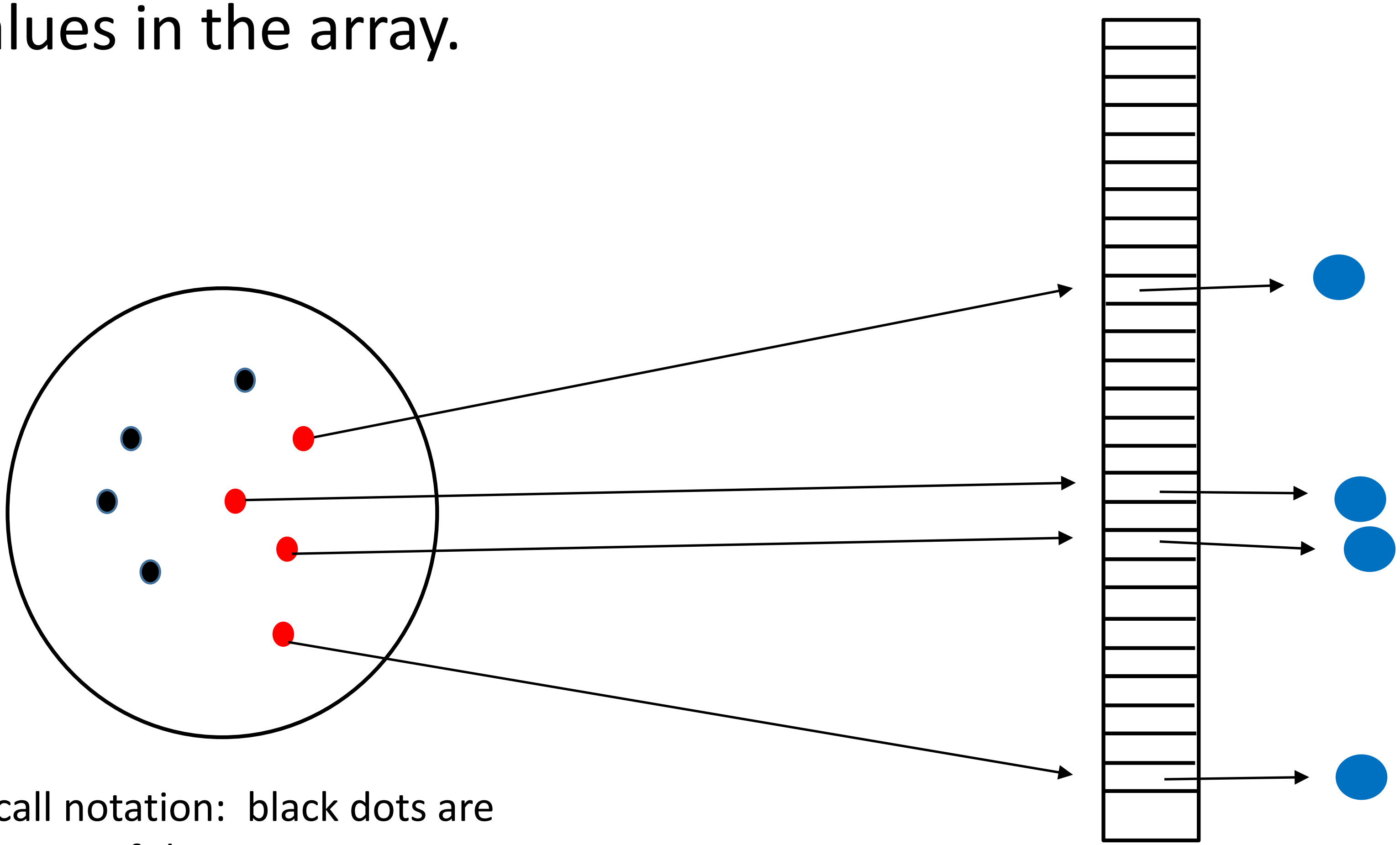
General Case:

Keys might not be comparable.

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

Strategy for the General Case (Hash Maps – next lecture):

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



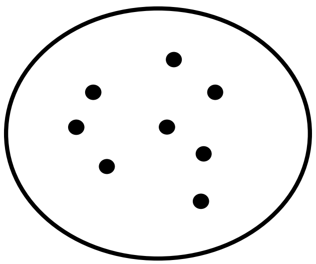
Recall notation: black dots are not part of the map.

Rest of today:

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

Such a map is called a *hash code*.

“default” hashcode() map in Java



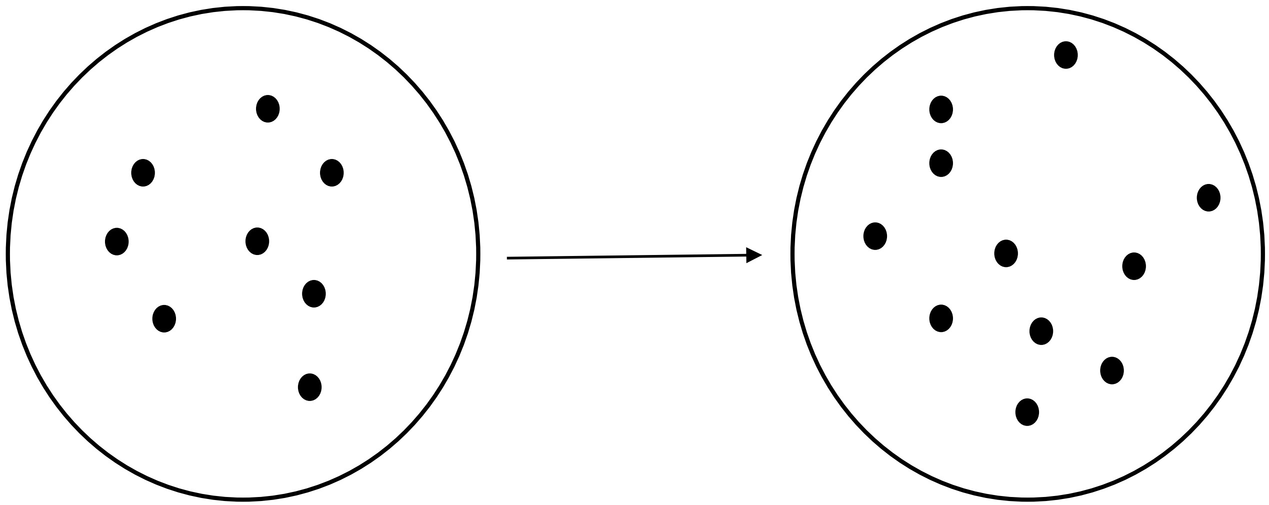
$$\xrightarrow[\text{(not many-to-1)}]{\text{1-to-1}} \{\,0, 1, 2, \ldots, 2^{24}\ - 1\}$$

objects in a Java program (runtime)

object’s starting (“base”) *address* in JVM memory (24 bits)

By default, “obj1 == obj2” means “obj1.hashcode() == obj2.hashcode()”

String.hashcode() in Java



Strings

`int` (32 bit)

For each String, define an integer.

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.

String.hashcode() in Java

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

where x = 31.

String.hashcode() in Java

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

where x = 31.

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

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

String.hashcode() in Java

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

where x = 31.

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

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

String.hashcode() in Java

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

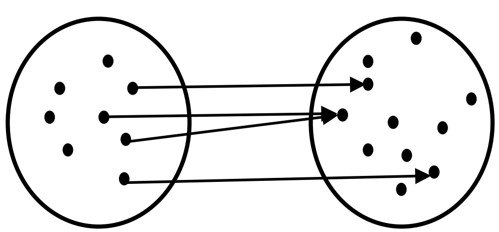
If s1.hashCode() == s2.hashCode() then … ?

String.hashcode() in Java

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

If s1.hashCode() == s2.hashCode() then … ?

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



String.hashcode() in Java

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

If s1.hashCode() == s2.hashCode() then … ?

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

If s1.hashCode() != s2.hashCode() then … ?

String.hashcode() in Java

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

If s1.hashCode() == s2.hashCode() then …

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

If s1.hashCode() != s2.hashCode() then …

s1 is a different string than s2.

ASIDE: Use Horner's rule for efficient polynomial evaluation

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

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

ASIDE: Use Horner's rule for efficient polynomial evaluation

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

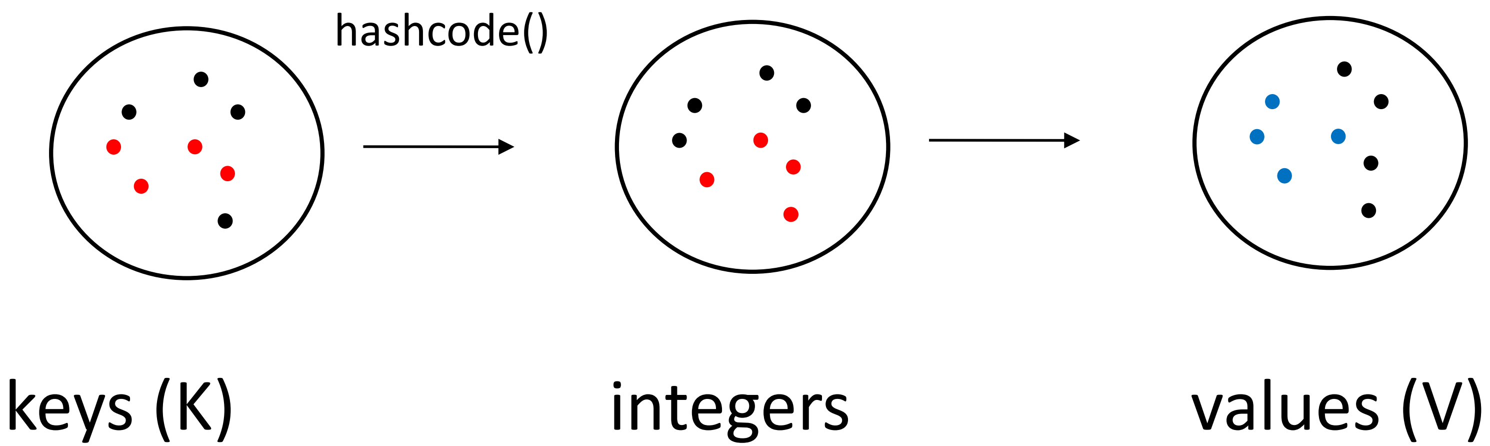
= (s[0] * 31^2 + s[1] * 31^1 + s[2]) * 31 + s[3]

= ((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



We want to map the keys to a *small* range of positive integers.

How ?