Structured Programming

COMP1110/6710

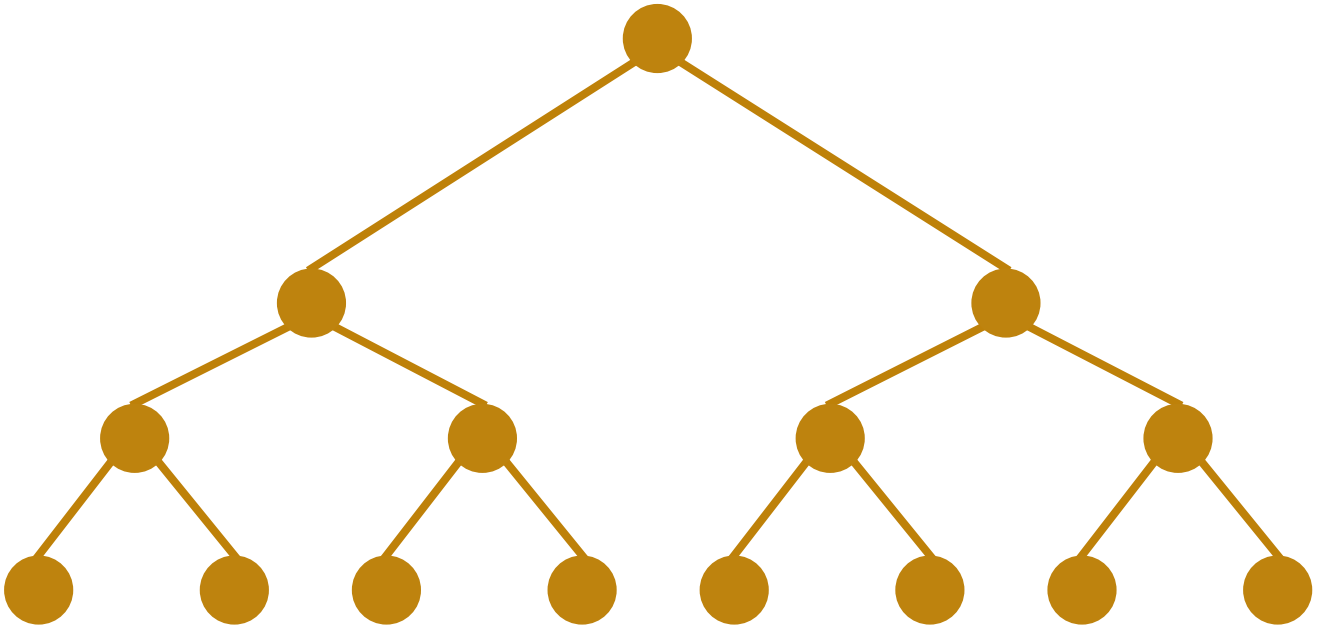
Australian National University

Image Courtesy NASA/JPL-Caltech.

BFS or DFS?

Australian National University

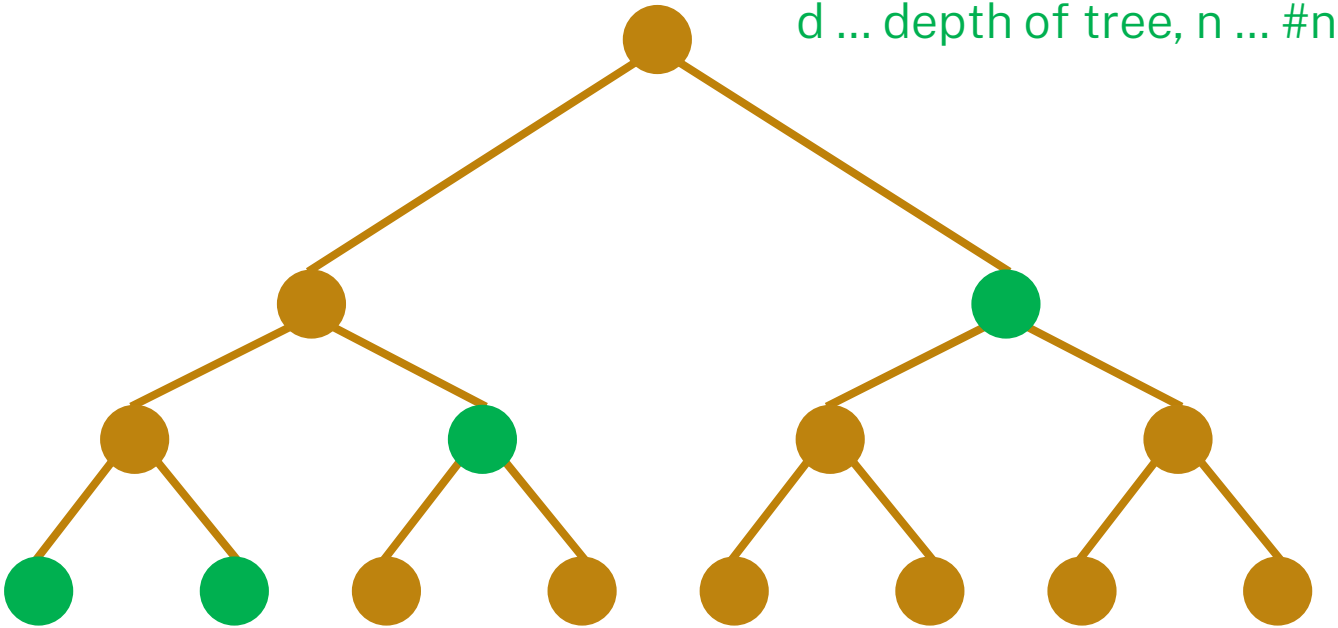
BFS or DFS?



BFS or DFS?

Nodes on DFS stack, worst case

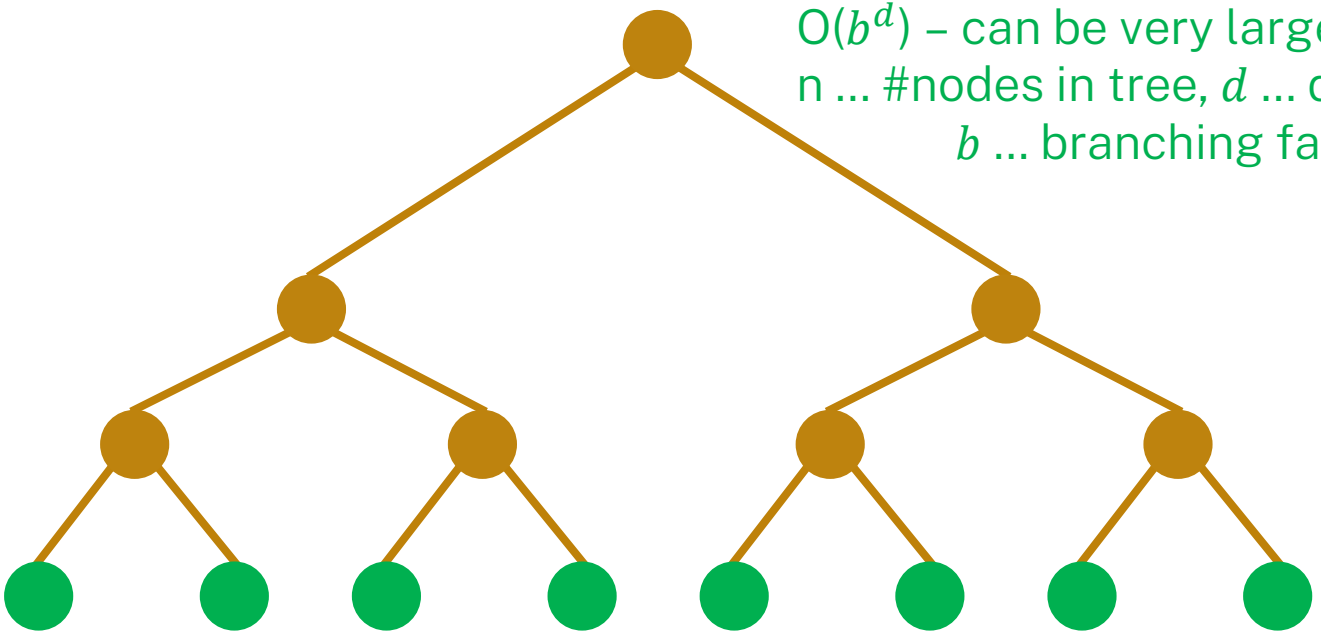
- Up to branching factor * depth
- For binary tree, O(d) = O(log n)
  d … depth of tree, n … #nodes in tree



BFS or DFS?

Nodes in BFS queue, worst case
- Up to width of tree
- For binary tree: O(n), generally $O(b^d)$ – can be very large

n … #nodes in tree, d … depth of tree
b … branching factor



This assumes that you do not need to keep track of where you came from, though with a branching factor ≥ 2, #past nodes < #next nodes

BFS or DFS?

Maximum Stack/Queue Size for binary trees

Depth	DFS	BFS
1	1	1
2	2	2
3	3	4
4	4	8
5	5	16
6	6	32
7	7	64
8	8	128
9	9	256

BFS or DFS?

Default recommendation: recursive DFS

Where trees get deep: switch to iteration

Reasons to go to BFS:

- Distance from start matters
- Tree is deep, but goal is likely close
- Special case: tree may be infinitely deep, so goal, if it exists, is always relatively close, but may be on a different branch

➔ BFS ensures an “even” search

Sets, Maps, and Hashing

One Weird Trick to get to O(1)...ish

Australian National University

Recall: BFS/DFS

```
explore(Node n) {
    remaining = new Queue/Stack<>(); remaining.add/push(n);
    seen = new List?<>(); seen.add(n);
    while(!remaining.isEmpty()) {          O(n)
       current = remaining.dequeue/pop();
       for(n : current.neighbours) {       O(n)
           if(!seen.contains(n)) {         ???
              remaining.add/push(n); seen.add(n);
           }
       }
    }}
```

Implementing “Contains”

Ideas?

- List: linear scan – O(n)
- Sorted Array: binary search: O(log n)

Can we do better?

Why Linear Scan or Binary Search?

Because we don’t know where in the list/array an element should be.

In a sorted list/array, the position in the list/array depends on what other unknown things are in the list/array.

In an unsorted list/array, the position is completely arbitrary.

BOOK

How to design programs : an introduction to programming and computing

Felleisen, Matthias.

c2001

Available at Hancock General (QA76.6 .H697 2001) >

TOP

SEND ITEM DE...

FIND IN LIBRA...

DETAILS

LINKS

REQUEST IT F...

VIRTUAL BRO...

Send item details

EXPORT TO EXCEL | CITATION | PERMALINK | EMAIL | QR | PRINT | EXPORT RIS

Find in Library

Please sign in to check if there are any request options. Sign in

BACK TO LOCATIONS

LOCATION ITEMS

Hancock

Available , General ; QA76.6 .H697 2001

(1 copy, 1 available, 0 requests)

Available
Loanable

Material Type: Book
Location: Hancock General QA76.6 .H697 2001
Barcode: 2493693

Details

Title	How to design programs : an introduction to programming and computing
Creator	Felleisen, Matthias. >
Subject	Computer programming >
	Electronic data processing >
Publisher	Cambridge, Mass. : MIT Press
Creation Date	c2001
Format	xxx, 693 p. : ill. ; 24 cm.

BOOK

How to design programs : an introduction to programming and com

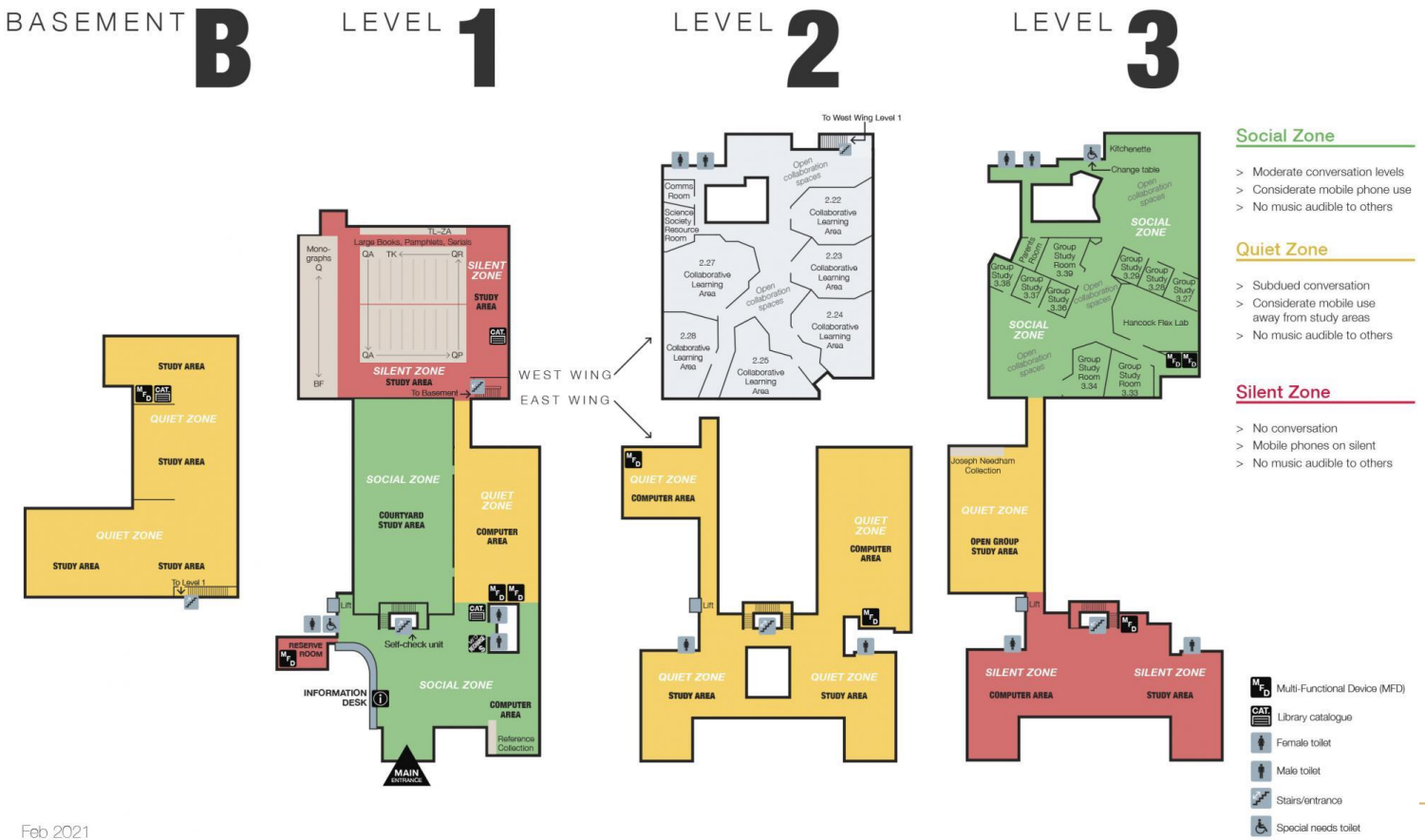
Felleisen, Matthias.

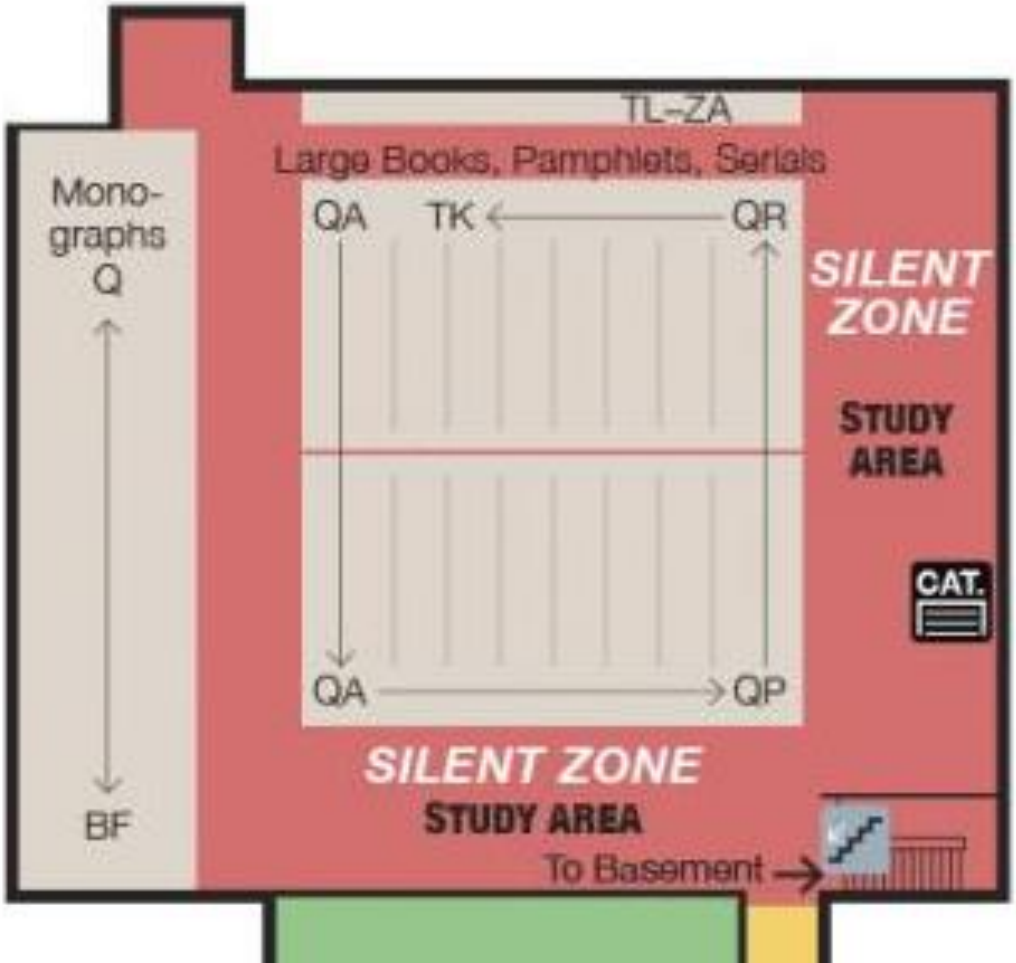
c2001

Available at Hancock General (QA76.6 .H697 2001) >

Send item details

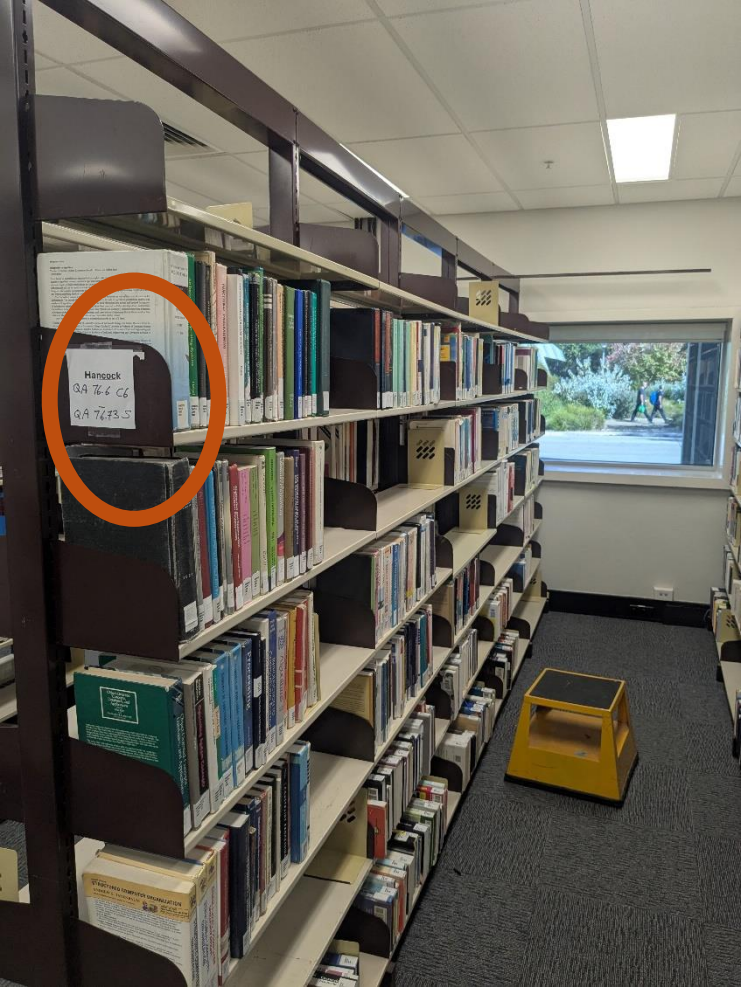
Hancock Library floor plan

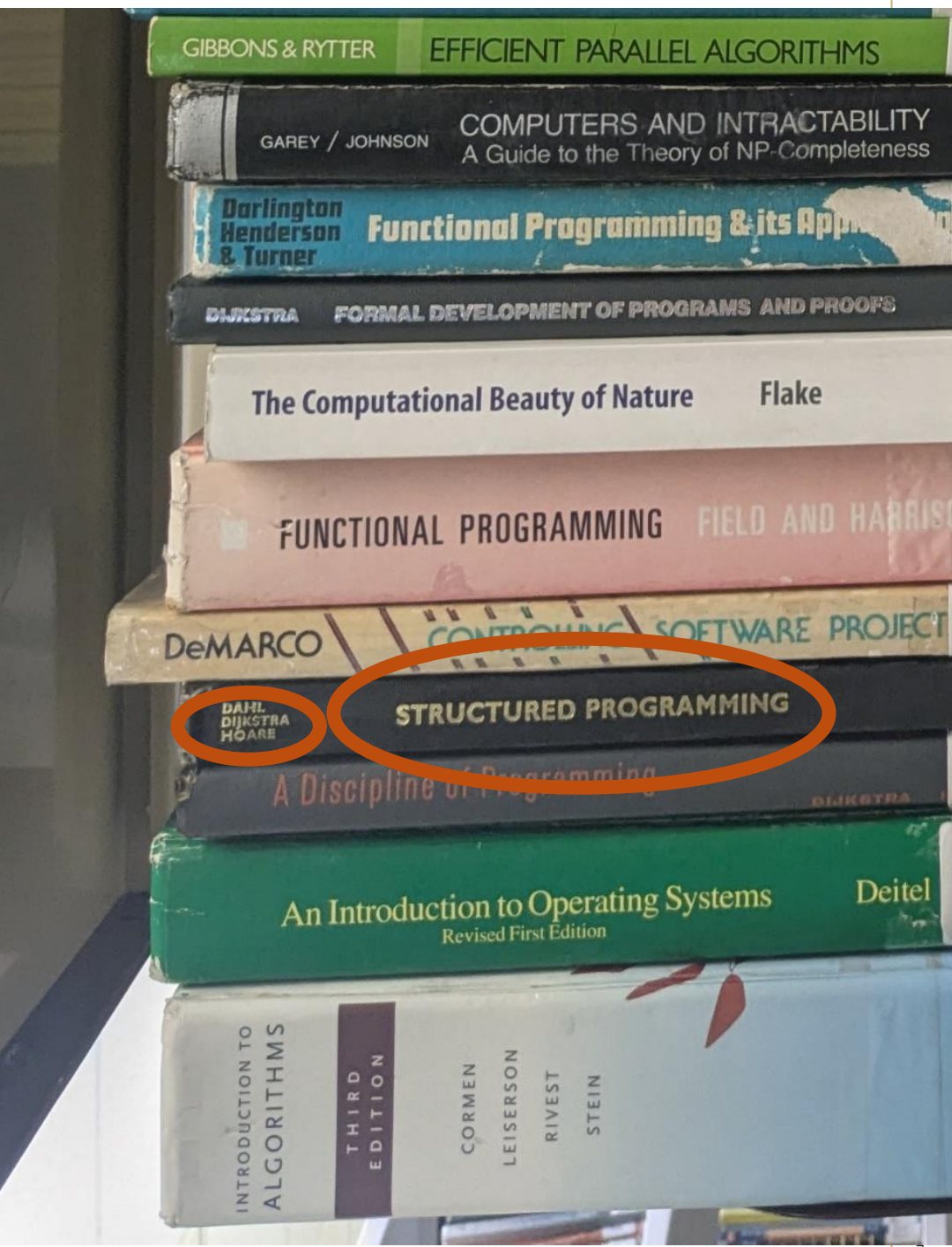


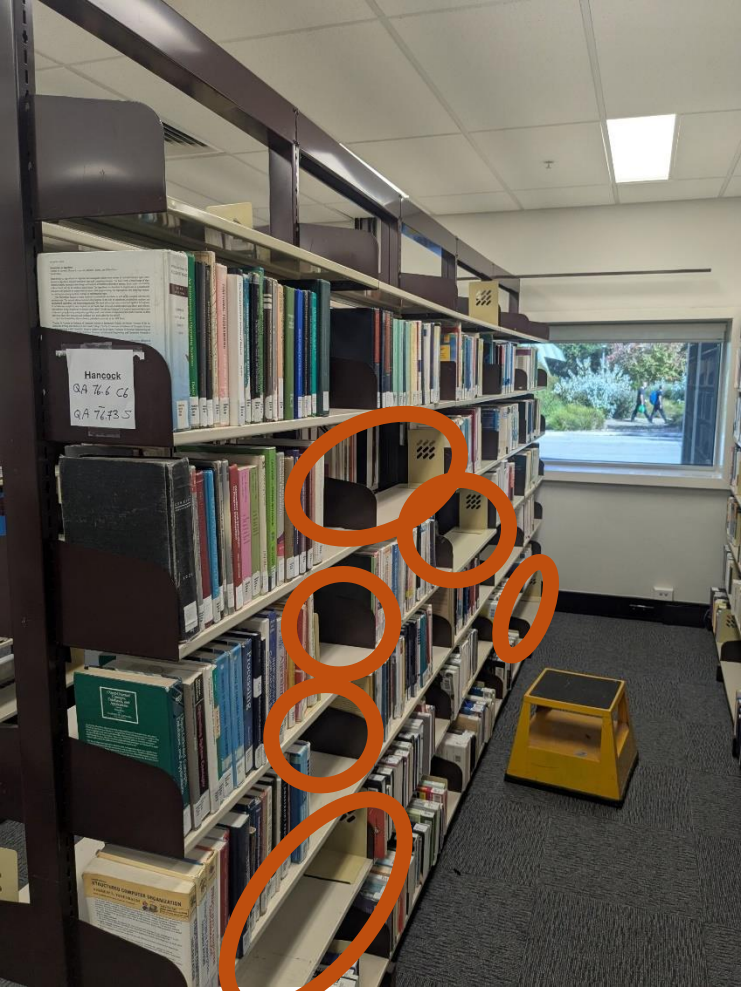


WEST WING

EAST WING




GIBBONS & RYTTER
EFFICIENT PARALLEL ALGORITHMS
GAREY / JOHNSON
COMPUTERS AND INTRACTABILITY
A Guide to the Theory of NP-Completeness
Darlington
Henderson
& Turner
Functional Programming & its App
DIJKSTRA
FORMAL DEVELOPMENT OF PROGRAMS AND PROOFS
The Computational Beauty of Nature
Flake
FUNCTIONAL PROGRAMMING
FIELD AND HARRIS
DeMARCO
SOFTWARE PROJECT
DAHL
DIJKSTRA
HOARE
STRUCTURED PROGRAMMING
A Discipline of Programming
DIJKSTRA
An Introduction to Operating Systems
Revised First Edition
Deitel
INTRODUCTION TO
ALGORITHMS
THIRD
EDITION
CORMEN
LEISERSON
RIVEST
STEIN



A Library (in principle)

- Each book has a specific location
- If you borrow a book, all the other books don’t move
- If you return a book, all the other books don’t move
- There is some additional empty space for when the library acquires new books
- Once in a while (very rarely), things get reorganized

➔ Adding, Finding, and Removing a book does not need a great deal of searching

In a Computer

If you have an array, and know an index into it, access is O(1)

➔ If we can assign every object a ~~unique~~ identifier, and have an array big enough for all them, we can access their slot instantly

Objects are much more short-lived than books – lots of empty space

Needs as much space as the rest of the program for every array

Lookup should be able to find objects that are equal, but not necessarily the same object (==)

hashCode

An int value for every object.

2^{32} (~4 billion) possible values.

Remember: longs have 2^{64} possible values, so there are 4 billion longs for every possible hashCode. That's fine, it does not have to be unique.

hashCode

Object implements hashCode based on an object's address in memory:

```
jshell> Object o=new Object()
o ==> java.lang.Object@34033bd0

jshell> o.hashCode()
$2 ==> 872627152
```

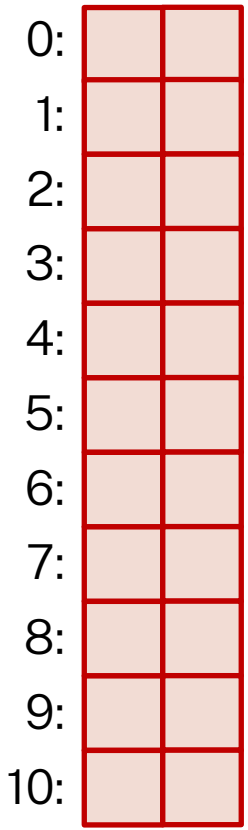
Hex 34033bd0 = Dec 872627152

Still a Large Array?

4 billion entries would be a lot. But now that we accept collisions, what's a few more?

➔ Pick a more reasonable size for your array (depending on how much data you have). Then take the (positive) remainder of dividing the hash code by the array size.

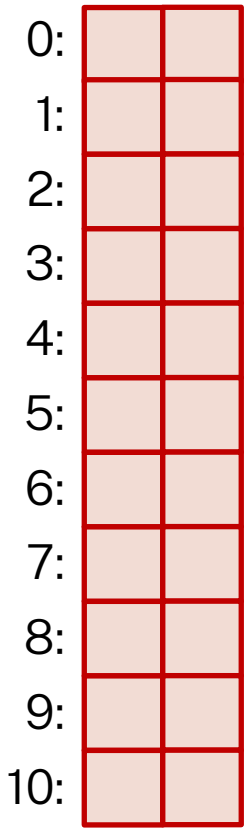
HashMap: Key Idea



e.g. Array Size: 11

```
HashMap<String, Integer> map = new HashMap<>();
String s = “Hello”;
Integer i = 5;
map.put(s, i);
map.containsKey(s);
```

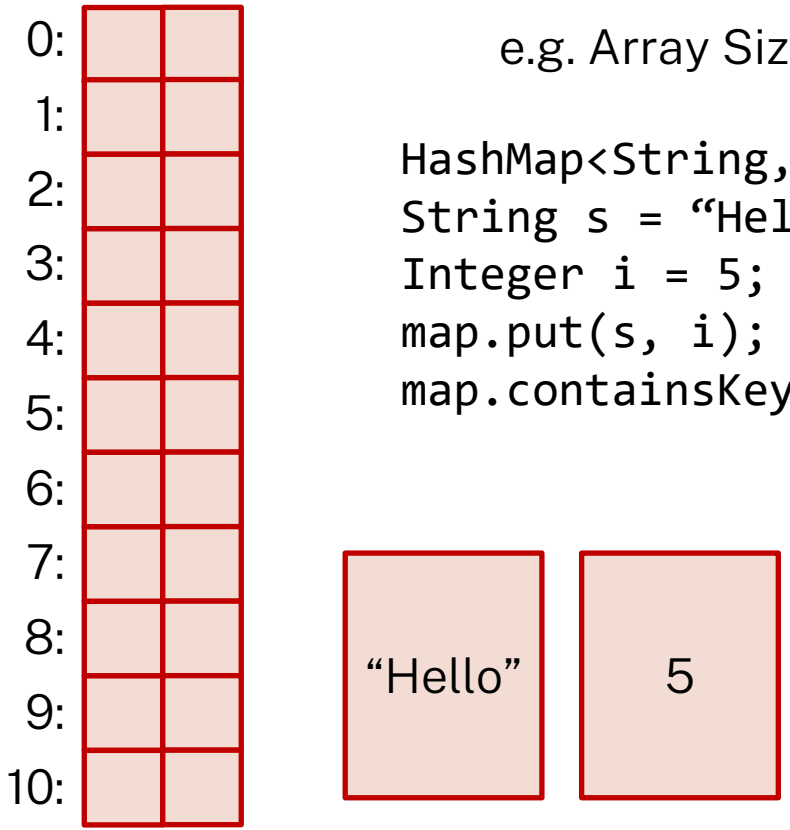
HashMap: Key Idea



e.g. Array Size: 11

```
HashMap<String, Integer> map = new HashMap<>();
String s = “Hello”;
Integer i = 5;
map.put(s, i);
map.containsKey(s);
```

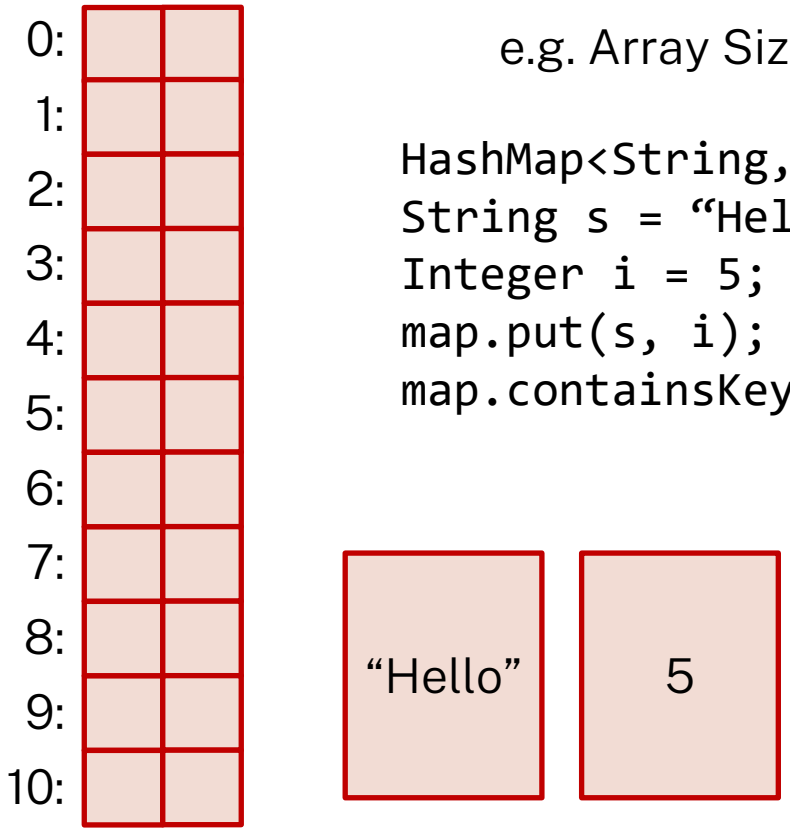
HashMap: Key Idea



e.g. Array Size: 11

```
HashMap<String, Integer> map = new HashMap<>();
String s = “Hello”;
Integer i = 5;
map.put(s, i);
map.containsKey(s);
```

HashMap: Key Idea

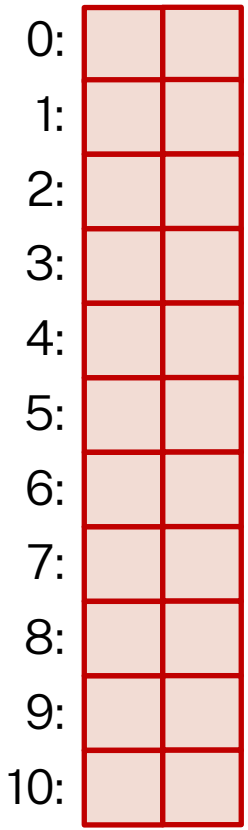


e.g. Array Size: 11

```
HashMap<String, Integer> map = new HashMap<>();
String s = “Hello”;
Integer i = 5;
map.put(s, i);
map.containsKey(s);
```

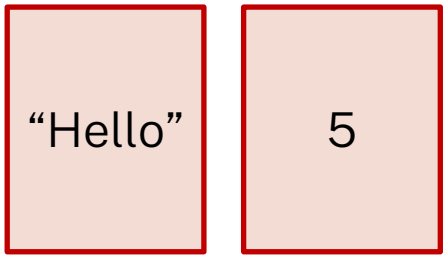
HashMap: Key Idea

"Hello".hashCode() = 69609650
% 11 = 0 → idx



e.g. Array Size: 11

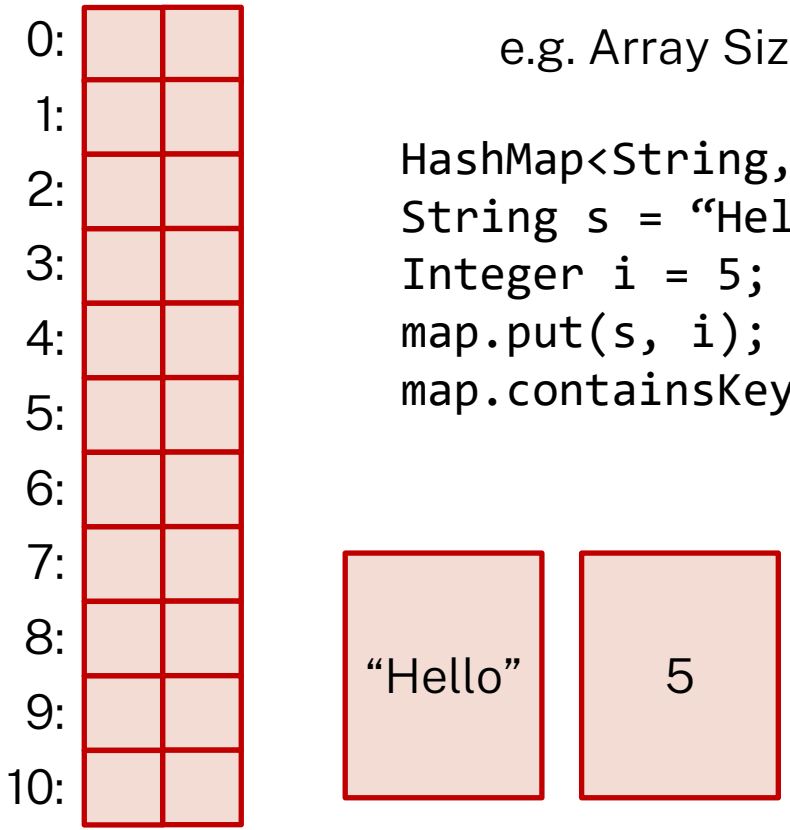
```
HashMap<String, Integer> map = new HashMap<>();
String s = "Hello";
Integer i = 5;
map.put(s, i);
map.containsKey(s);
```




```
Object put(String key, Integer val) {
  int idx = key.hashCode()%size;
  var old = arr[idx].val;
  arr[idx].key = key;
  arr[idx].val = val;
  return old;
}
```

HashMap: Key Idea

"Hello".hashCode() = 69609650
% 11 = 0 → idx



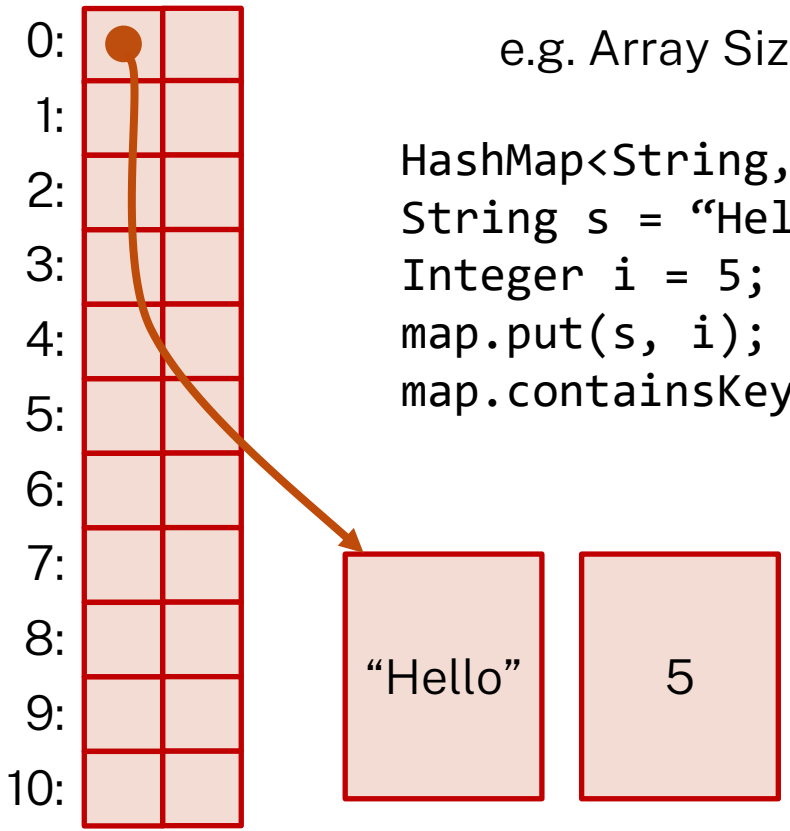
e.g. Array Size: 11

```
HashMap<String, Integer> map = new HashMap<>();
String s = "Hello";
Integer i = 5;
map.put(s, i);
map.containsKey(s);
```

```
Object put(String key, Integer val) {
  int idx = key.hashCode()%size;
  var old = arr[idx].val;  null
  arr[idx].key = key;
  arr[idx].val = val;
  return old;
}
```

HashMap: Key Idea

"Hello".hashCode() = 69609650
% 11 = 0 → idx



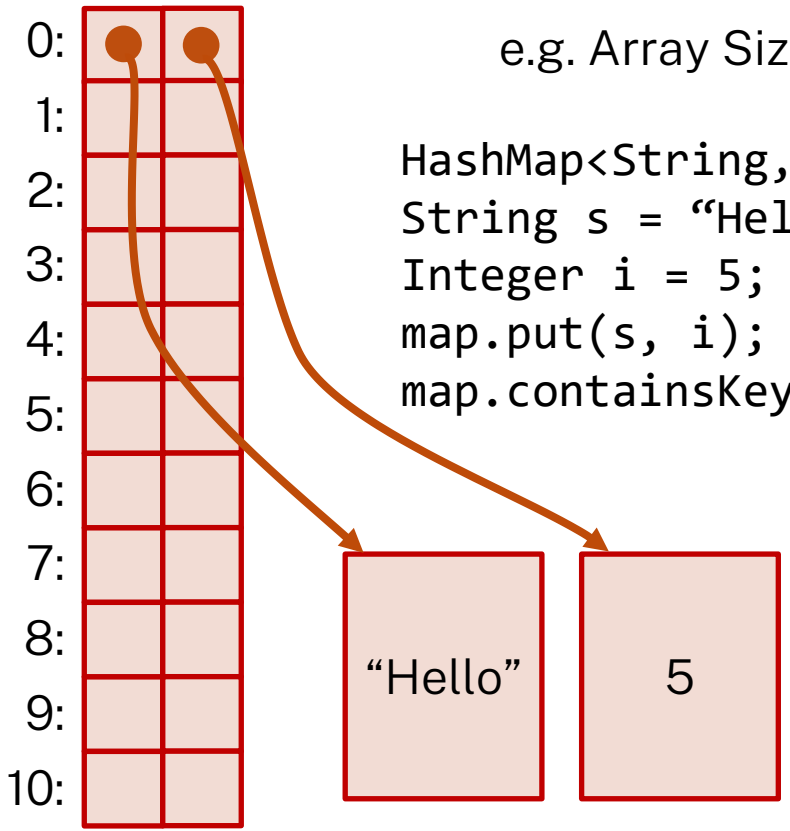
e.g. Array Size: 11

```
HashMap<String, Integer> map = new HashMap<>();
String s = "Hello";
Integer i = 5;
map.put(s, i);
map.containsKey(s);
```

```
Object put(String key, Integer val) {
  int idx = key.hashCode()%size;
  var old = arr[idx].val;  null
  arr[idx].key = key;
  arr[idx].val = val;
  return old;
}
```

HashMap: Key Idea

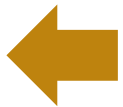
"Hello".hashCode() = 69609650
% 11 = 0 → idx



e.g. Array Size: 11

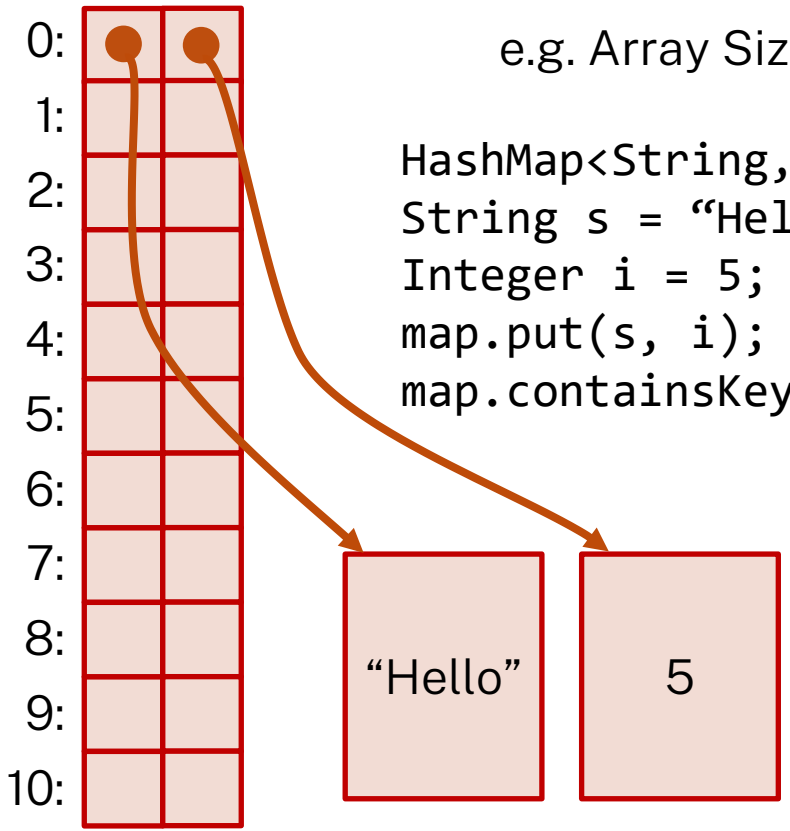
```
HashMap<String, Integer> map = new HashMap<>();
String s = "Hello";
Integer i = 5;
map.put(s, i);
map.containsKey(s);
```

```
Object put(String key, Integer val) {
  int idx = key.hashCode()%size;
  var old = arr[idx].val;  null
  arr[idx].key = key;
  arr[idx].val = val;
  return old;
}
```



HashMap: Key Idea

"Hello".hashCode() = 69609650
% 11 = 0 → idx



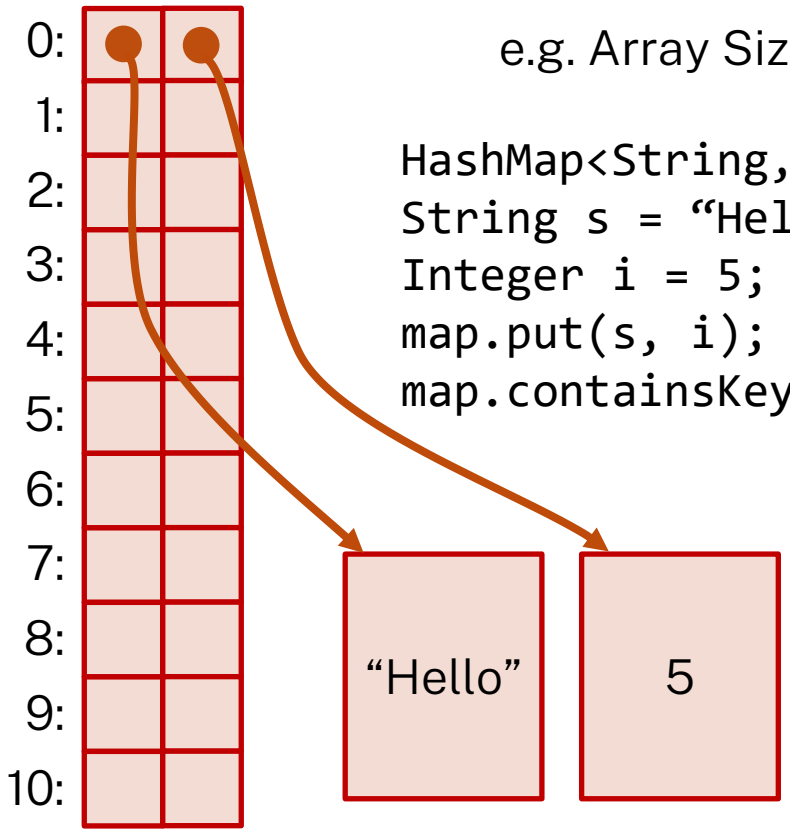
e.g. Array Size: 11

```
HashMap<String, Integer> map = new HashMap<>();
String s = "Hello";
Integer i = 5;
map.put(s, i);
map.containsKey(s);
```

```
Object put(String key, Integer val) {
  int idx = key.hashCode()%size;
  var old = arr[idx].val;  null
  arr[idx].key = key;
  arr[idx].val = val;
  return old;
}
```

HashMap: Key Idea

"Hello".hashCode() = 69609650
% 11 = 0 → idx



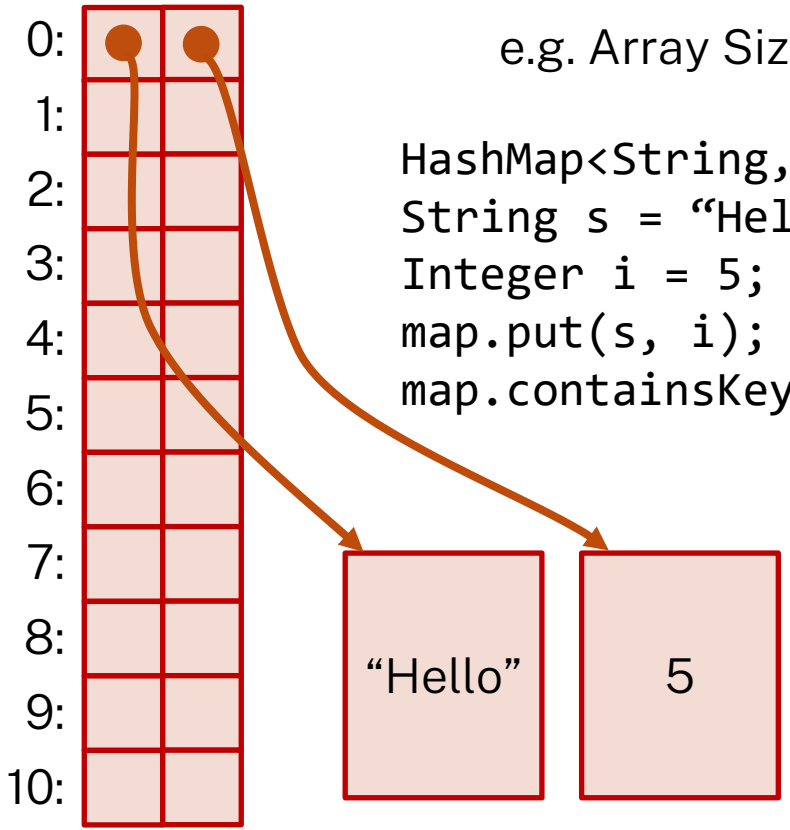
e.g. Array Size: 11

```
HashMap<String, Integer> map = new HashMap<>();
String s = "Hello";
Integer i = 5;
map.put(s, i);
map.containsKey(s);
```

```
Object put(String key, Integer val) {
  int idx = key.hashCode()%size;
  var old = arr[idx].val;
  arr[idx].key = key;
  arr[idx].val = val;
  return old;
}
```

null

HashMap: Key Idea

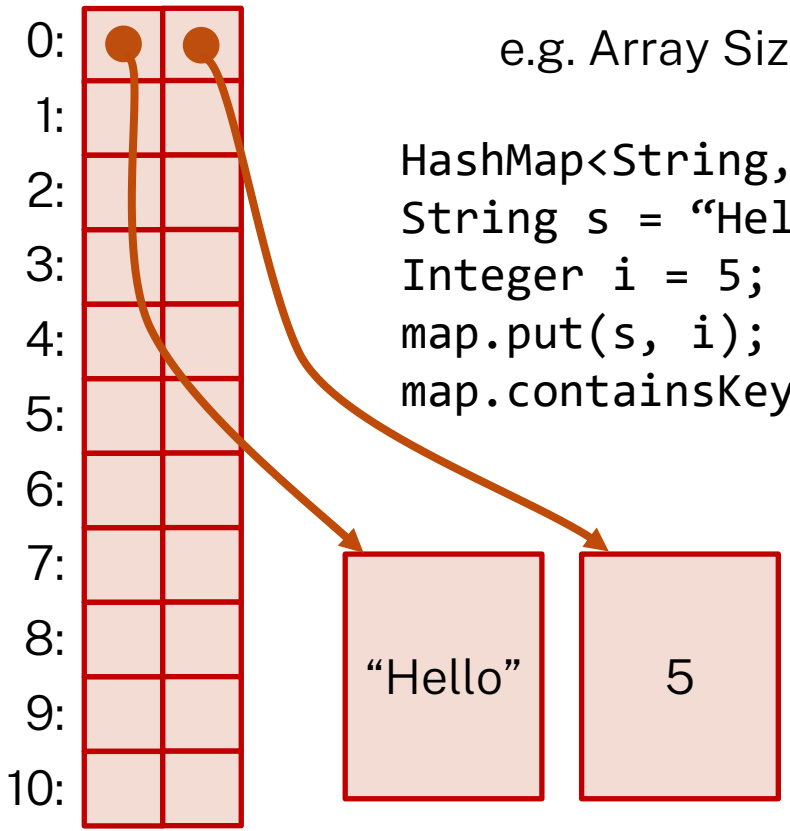


e.g. Array Size: 11

```
HashMap<String, Integer> map = new HashMap<>();
String s = “Hello”;
Integer i = 5;
map.put(s, i);
map.containsKey(s);
```

HashMap: Key Idea

"Hello".hashCode() = 69609650
% 11 = 0 → idx



e.g. Array Size: 11

```
HashMap<String, Integer> map = new HashMap<>();
String s = "Hello";
Integer i = 5;
map.put(s, i);
map.containsKey(s);
```

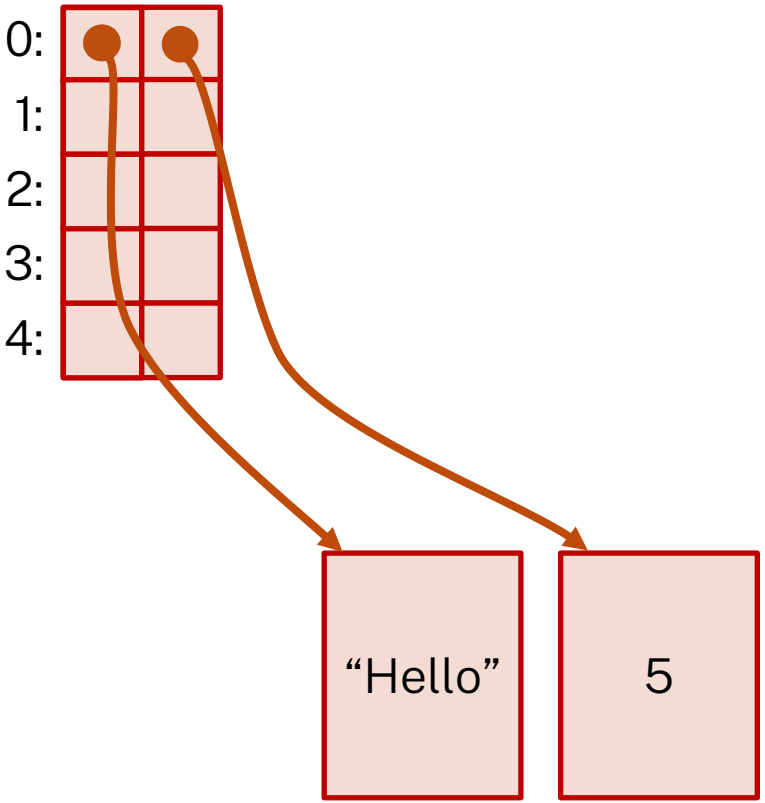
```
boolean containsKey(String key) {
  int idx = key.hashCode()%size;
  if(arr[idx] == null) return false;
  return arr[idx].key.equals(key);
}
```

Hash Collisions

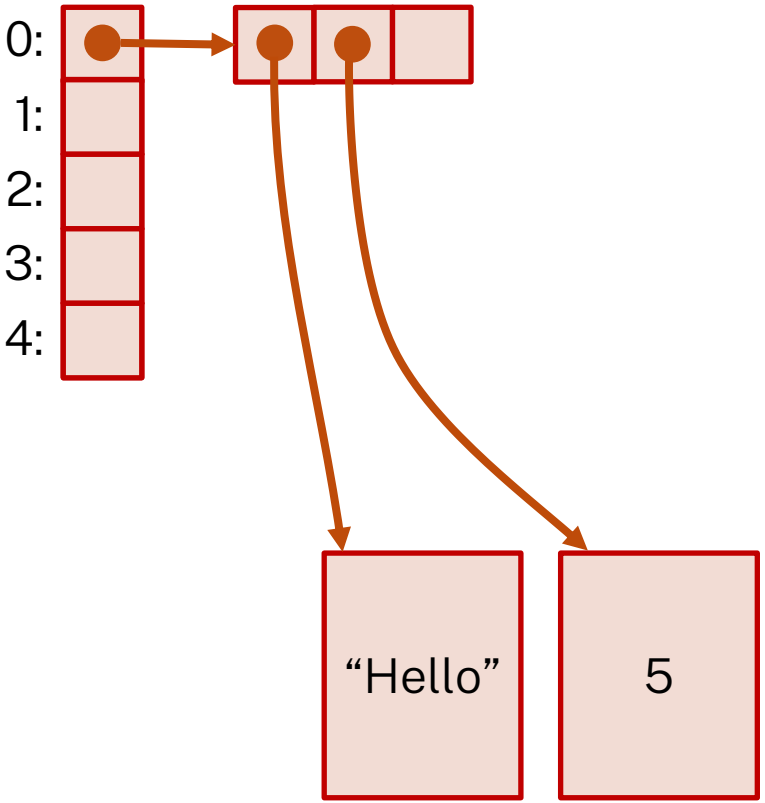
Many Objects will have the same hash code. Taking the remainder to get a smaller number increases the number of collisions further.

Many solutions, but simplest one is to have each entry point to a linked list – a “bucket”, which can contain multiple entries mapped to the same index. The equals method distinguishes between the keys.

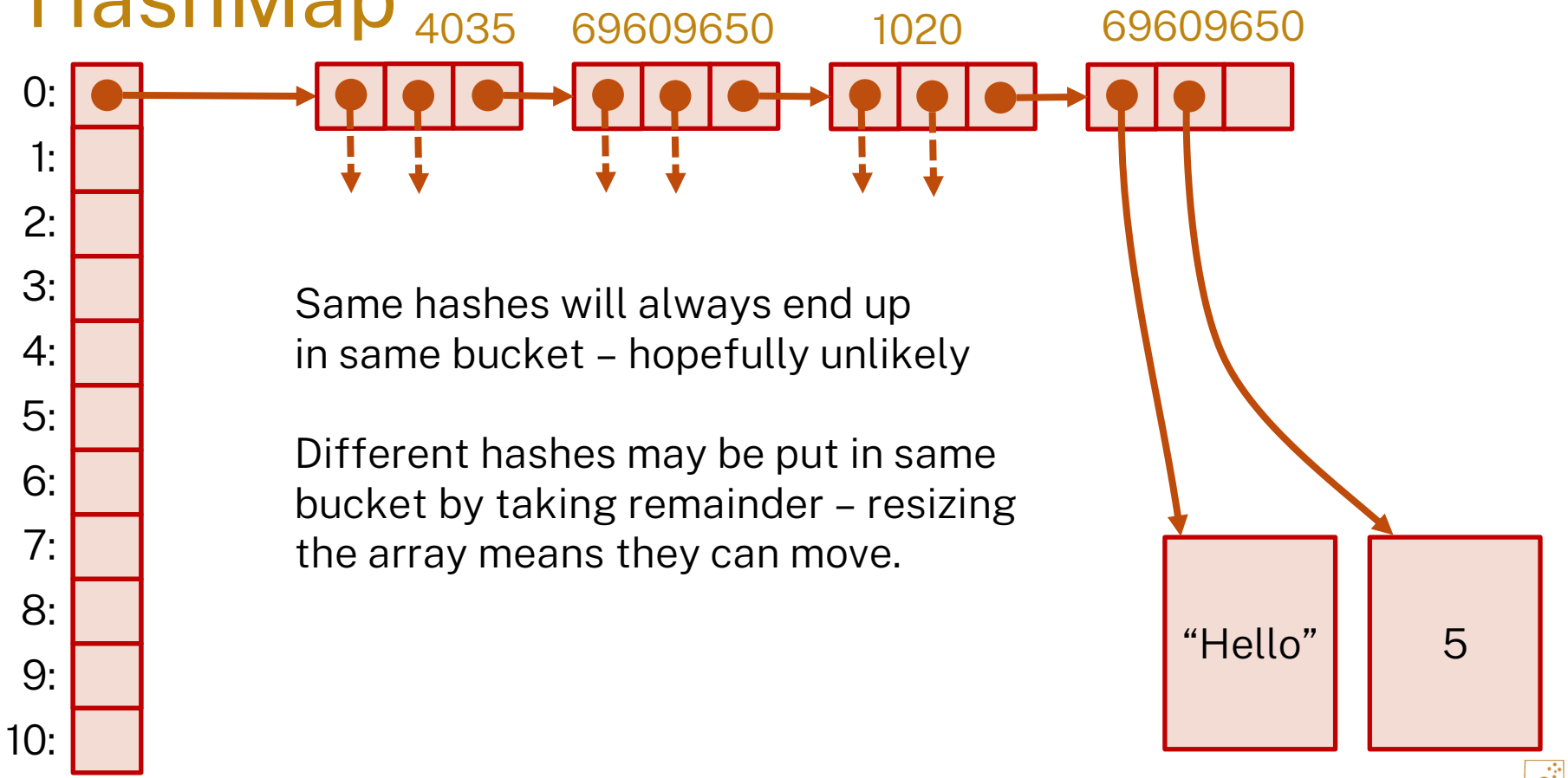
HashMap



HashMap



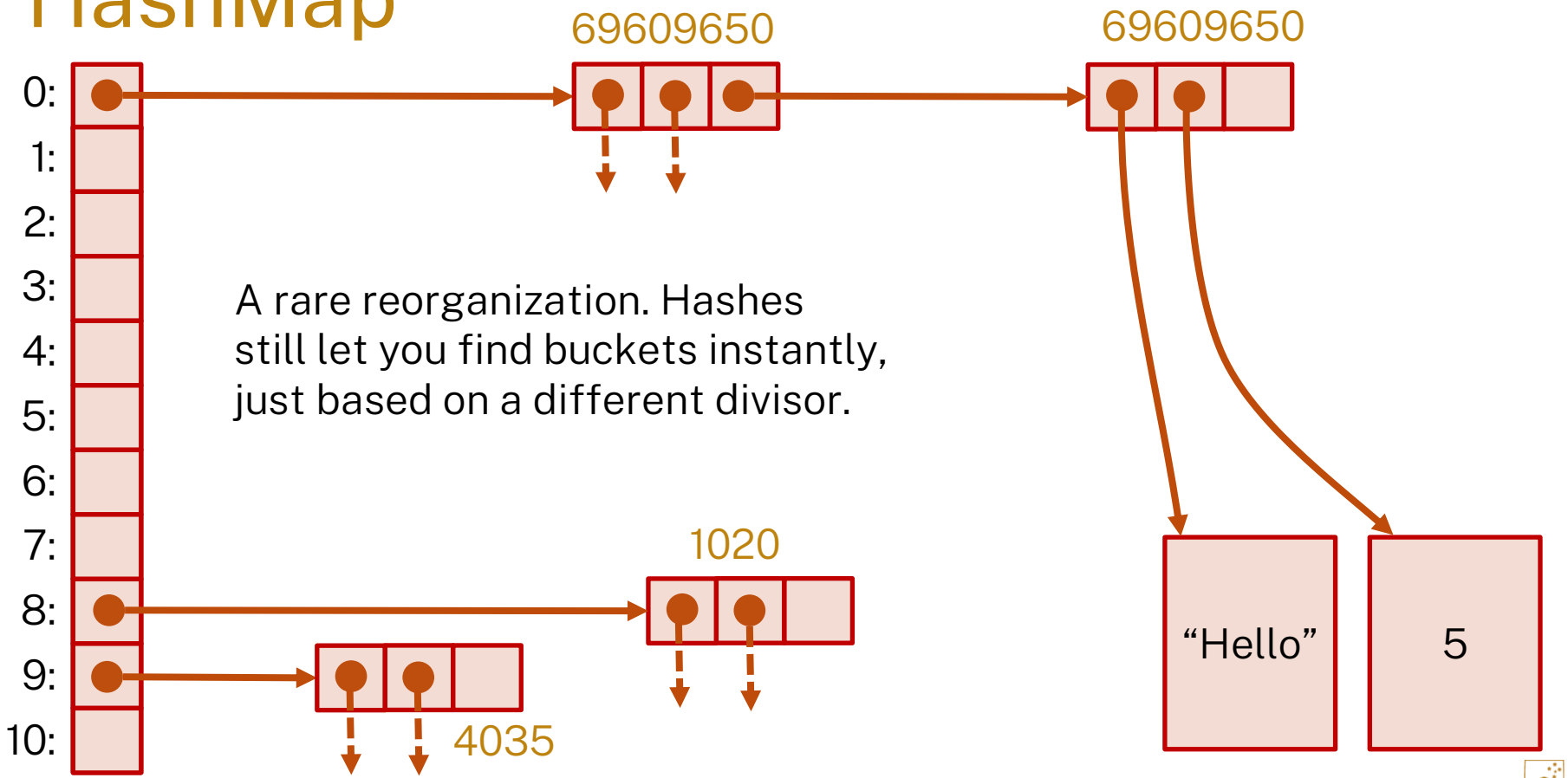
HashMap



Same hashes will always end up in same bucket – hopefully unlikely

Different hashes may be put in same bucket by taking remainder – resizing the array means they can move.

HashMap



This is a bucket

These are things in the bucket

HashSet

A concrete implementation of Abstract Data Type “Set”, which is a list without ordering or multiple entries of the same element.

- Add/Remove
- Check Membership
- Iterate through elements, in some arbitrary order

In Java, implementation based on HashMap – simply ignore the associated values.

Time Complexity

Depends on implementation.

For local linked lists, key operations (put, get, remove, containsKey) on average constant, but not quite O(1).

Better implementations can do O(1); in any case, we typically treat those operations as constant.

Overall analysis is complicated ➔ more advanced courses.

Key requirement: a good hash function

Hash Functions and Equality

Australian National University

Recap: equals vs. ==

== compares object identity, i.e. are left and right the same thing on the heap? Cannot be changed.

equals has a default implementation as ==, but can be overridden.

Many standard library classes do this, and you can, too.

Overriding Equals

```
class Person {
    String firstName;
    String lastName;
    …
    @Override
    boolean equals(Object other) {
        if(other instanceof Person p) {
            return firstName.equals(p.firstName) &&
                   lastName.equals(p.lastName);
        }
        return false;
    }
}
```

Generally, you want to compare the values of relevant fields.

records do this automatically.

Expectations on Equals

- Reflexive (every object is equal to itself)
- Transitive (A equals B and B equals C ➔ A equals C)
- Symmetric (A equals B ⇔ B equals A)
- Stable results (may change if relevant fields have changed)
- Consistent with results of Comparable<T>.compareTo (if applicable) and hashCode
  - A.compareTo(B) == 0 ⇔ A.equals(B)
  - A.equals(B) ➔ A.hashCode() == B.hashCode()

Key Concept in Computing/Logic: Implication (➔) ≠ Equivalence (⇔)

What's a Good Hash Function?

```
class Person {
    String firstName;
    String lastName;
    …
    @Override
    int hashCode() {
        return 0;
    }
}
```

Valid:
Since all objects have the same hash code, it is guaranteed to be equal for equal objects.

Not Good:
Destroys good HashMap properties – everything will always go into the same bucket!

What's a Good Hash Function?

```
class Person {
    String firstName;
    String lastName;
    …
    @Override
    int hashCode() {
        return firstName.length();
    }
}
```

Valid:

Since all objects with the same firstName length have the same hash code, it is guaranteed to be equal for equal objects.

Not Good:

Maps most plausible values to very small subset of possible integers. Still very large buckets.

What's a Good Hash Function?

General Criteria:

- Evenly distributes results across range (in Java: ~4 billion int values)
- Sensitive to small changes
  e.g. "squeak" and "quakes" should have different hashes
- Cheap to compute

A *perfect* hash function maps every input to a unique value.
Sometimes possible, but rare.

Domain knowledge and application-specific tradeoffs make a huge Difference!

What's a Good Hash Function?

General Recipe (from "Effective Java" by Josh Bloch):

```
@Override
int hashCode() {
  int result = 0;
  for (var field : fields) {
    var x = field.hashCode();
    result = 31 * result + x;
  }
  return result;
}
```

Why 31?
Generally, with modulo, odd primes are the best choice to ensure coverage. Very different from numbers people would choose (powers of 10 or 2).

Makes room in accumulator so x can influence result and field values don't just cancel each other out

Another Way to Look at Hashes

Comparing objects can be expensive. E.g., for two graphs to be equal, they need the same vertices and edges. Two graphs might only differ in one edge, and thus are different, but you need to find that edge first.

Often, you can store a hash code in an object after computing it for the first time. It's then a cheap way to pre-check equality. If the hashes of two objects are different, then the objects must not be equal. Integer comparisons are cheap. Since most objects aren't equal, that's a lot of work you can save this way.

Uses of Hash Codes

- Hash Tables
- Cheaper Equality Comparisons / Checksums
  - In Programs
  - Of Files
  - Of Messages / Network Packets
  - Blockchains / Cryptography / Digital Signatures: note – harder requirements on “good” hash functions