Software Qualities: Time Efficiency

Marco Faella

Software Qualit*ies*

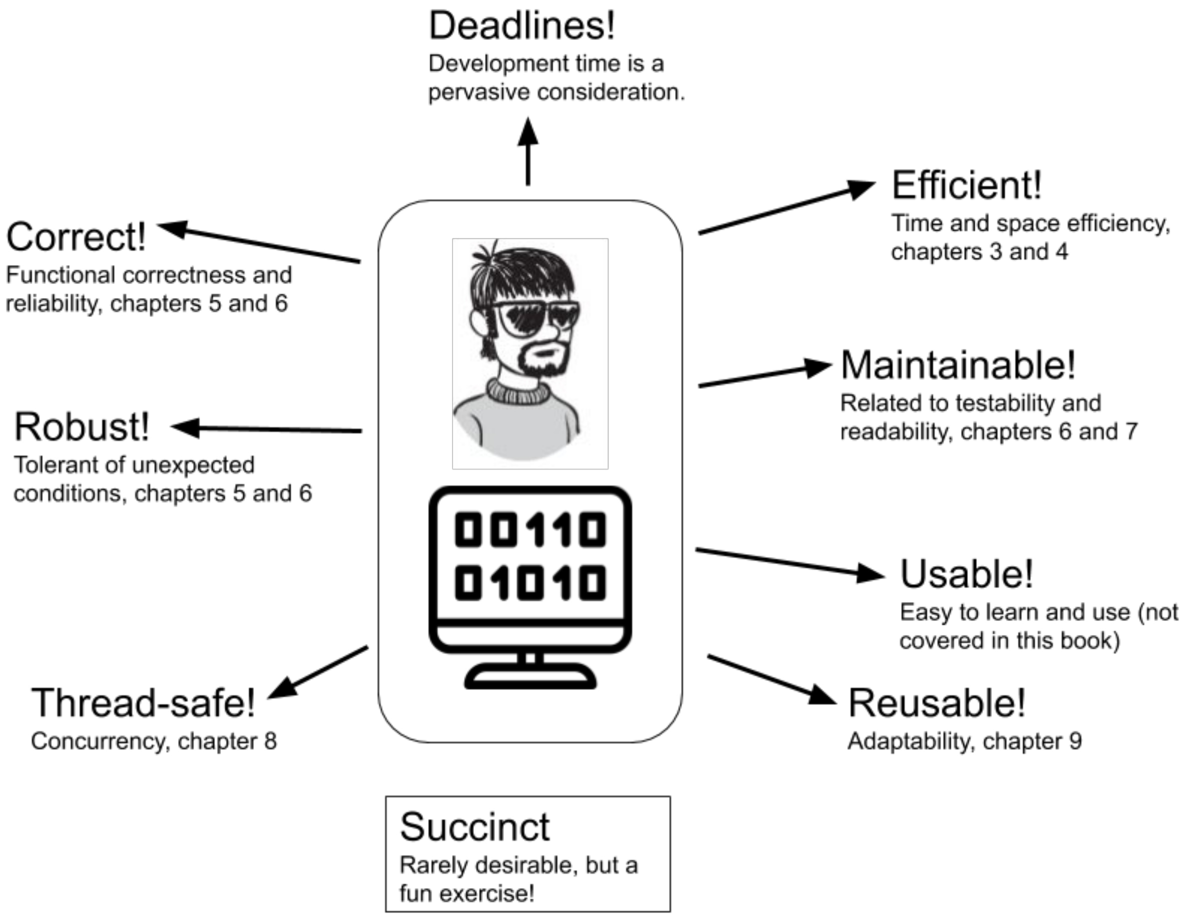


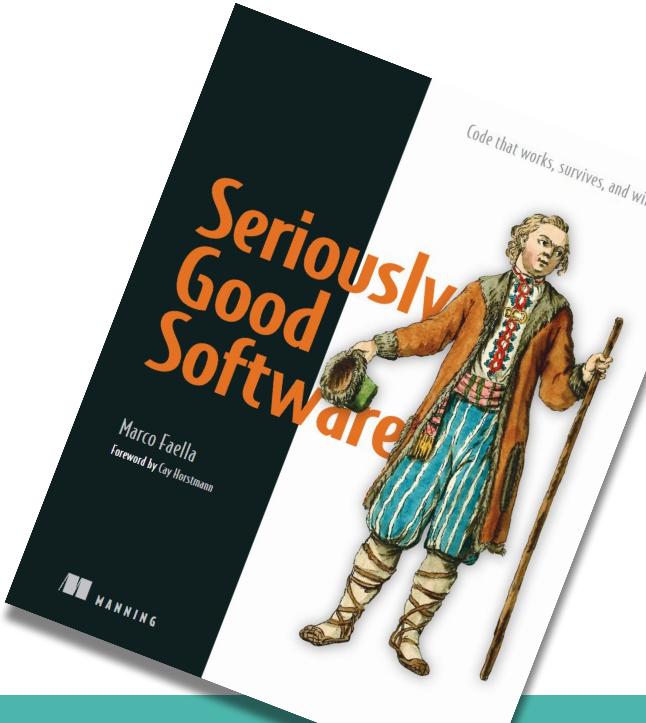
Figure from Seriously Good Software, by M. Faella © 2020 Manning Publications

Summary

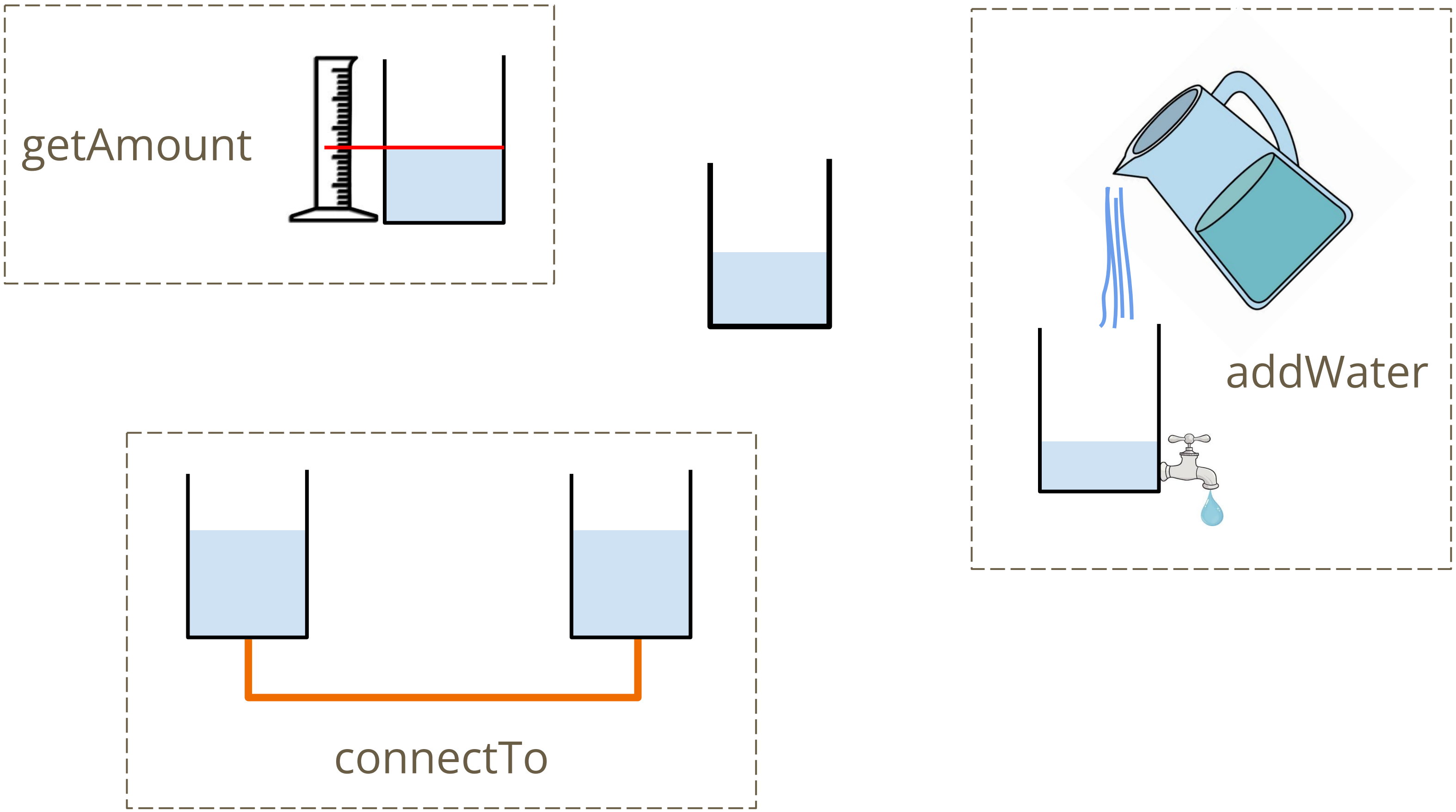
1. **Software qualities and a problem to solve**
2. **Reference implementation**
3. **Time efficiency**
4. **Space efficiency**
5. Reliability via monitoring
6. Reliability via testing
7. Readability
8. **Thread safety**
9. Generality

A. Succinctness
B. The ultimate water container

Code at: https://bitbucket.org/mfaella/exercisesinstyle



Water containers



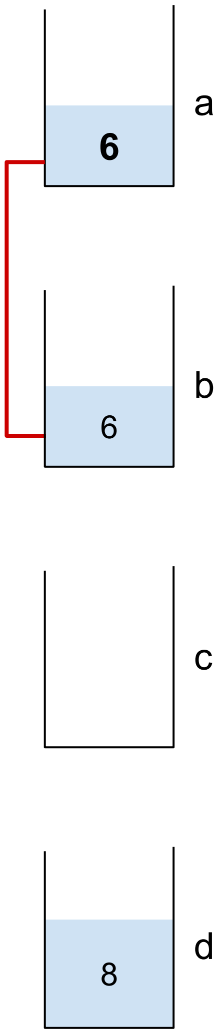
An API for water containers

- `double` **`getAmount`**`()`
- `void` **`addWater`**`(double amount)`
- `void` **`connectTo`**`(Container other)`

A use case

```
Container a = new Container(),
          b = new Container(),
          c = new Container(),
          d = new Container();

a.addWater(12);
d.addWater(8);
a.connectTo(b);
System.out.println(a.getAmount()); ➞ 6
```

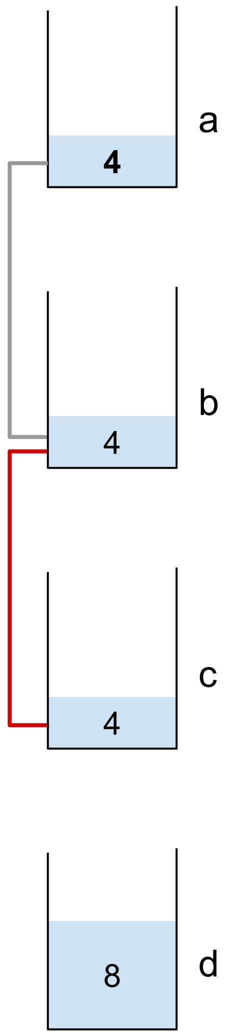


A use case

```
Container a = new Container(),
          b = new Container(),
          c = new Container(),
          d = new Container();

a.addWater(12);
d.addWater(8);
a.connectTo(b);
System.out.println(a.getAmount()); ➞ 6

b.connectTo(c);
System.out.println(a.getAmount()); ➞ 4
```



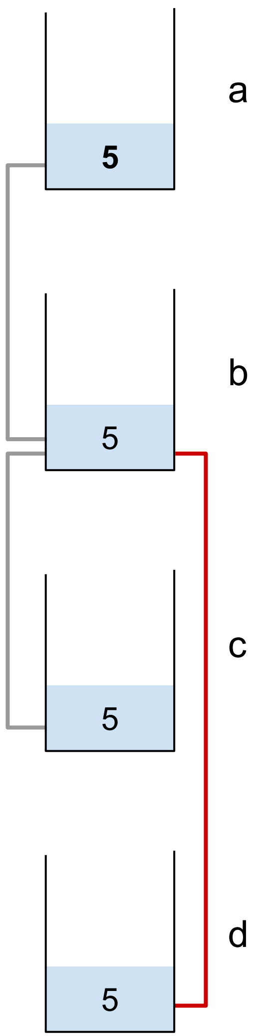
A use case

```
Container a = new Container(),
          b = new Container(),
          c = new Container(),
          d = new Container();

a.addWater(12);
d.addWater(8);
a.connectTo(b);
System.out.println(a.getAmount()); ➞ 6

b.connectTo(c);
System.out.println(a.getAmount()); ➞ 4

b.connectTo(d);
System.out.println(a.getAmount()); ➞ 5
```



60 seconds

to imagine your implementation

Reference implementation

The **fields**:

`double amount`	Amount of water in this container
`Set<Container> group`	Containers connected *directly or indirectly* to this one, including this one

The **constructor**:

```
public Container() {
   group = new HashSet<Container>();
   group.add(this);
}
```

Memory layout

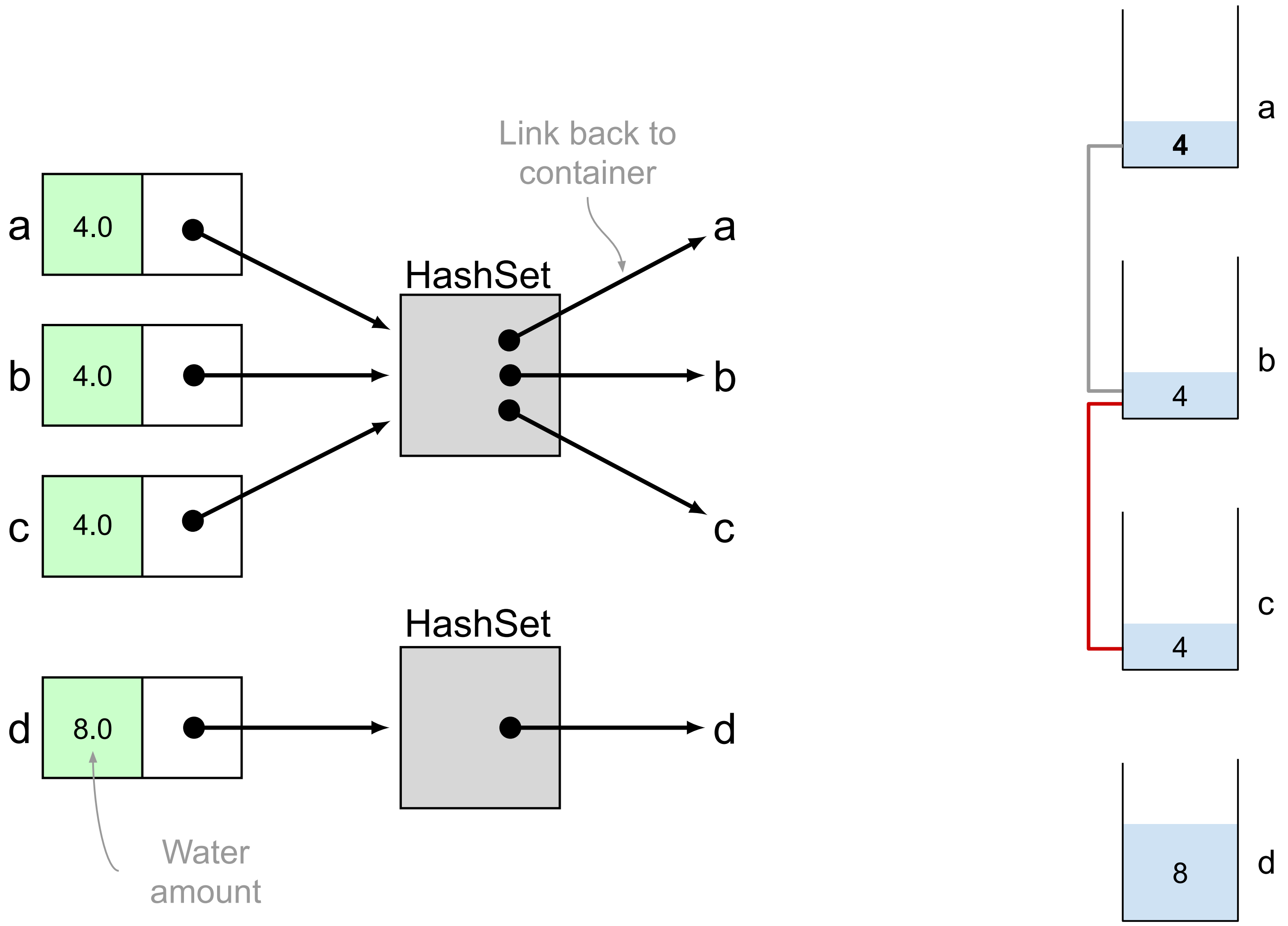


Figure from Seriously Good Software, by M. Faella © 2020 Manning Publications

Connecting two containers

```
public void connectTo(Container other) {
    if (group==other.group) return;
    int size1 = group.size(),
        size2 = other.group.size();
    double tot1 = amount * size1,
           tot2 = other.amount * size2,
           newAmount = (tot1 + tot2) / (size1 + size2);
    group.addAll(other.group);
    for (Container c: other.group)
        c.group = group;
    for (Container c: group)
        c.amount = newAmount;
}
```

If they are already connected, do nothing

Merge the two groups

Update group of containers connected with `other`

Update amount of all newly connected containers

Time Efficiency

Time efficiency in one slide

Step 0: Do you really need more speed?

Step 1: Asymptotic complexity

- Trend for increasing size of the inputs

Step 2: Profiling and optimizing

1. Profiling → guess, estimate, or measure the following:
   - **Usage** profile: How often do the clients call each method?
   - **Runtime** profile: Which methods actually take more time?
2. Optimize the most common/expensive method(s)

Complexity of reference implementation

Method	Time complexity
`getAmount`	$O(1)$
`connectTo`	$O(n)$
`addWater`	$O(n)$

Can we do better?

And what does “better” mean?

Complexity of multiple methods

Reference:

Method	Time complexity
`getAmount`	$O(1)$
`connectTo`	$O(n)$
`addWater`	$O(n)$

??

incomparable

Alternative 2:

Method	Time complexity
`getAmount`	$\boxed{O(n)}$
`connectTo`	$\boxed{O(1)}$
`addWater`	$\boxed{O(1)}$

Dominates! *(better)*

Alternative 1:

Method	Time complexity
`getAmount`	$O(1)$
`connectTo`	$O(n)$
`addWater`	$\boxed{O(1)}$

Dominance is a *partial order*

We want: not dominated by anything (*Pareto optimal*)

Can we add water in constant time?

Method	Time complexity
`getAmount`	$O(1)$
`connectTo`	$O(n)$
`addWater`	$\cancel{O(n)}$ ⇦ $\boxed{O(1)}$

Separate group objects

A single field:

```
Group group = new Group(this);
```

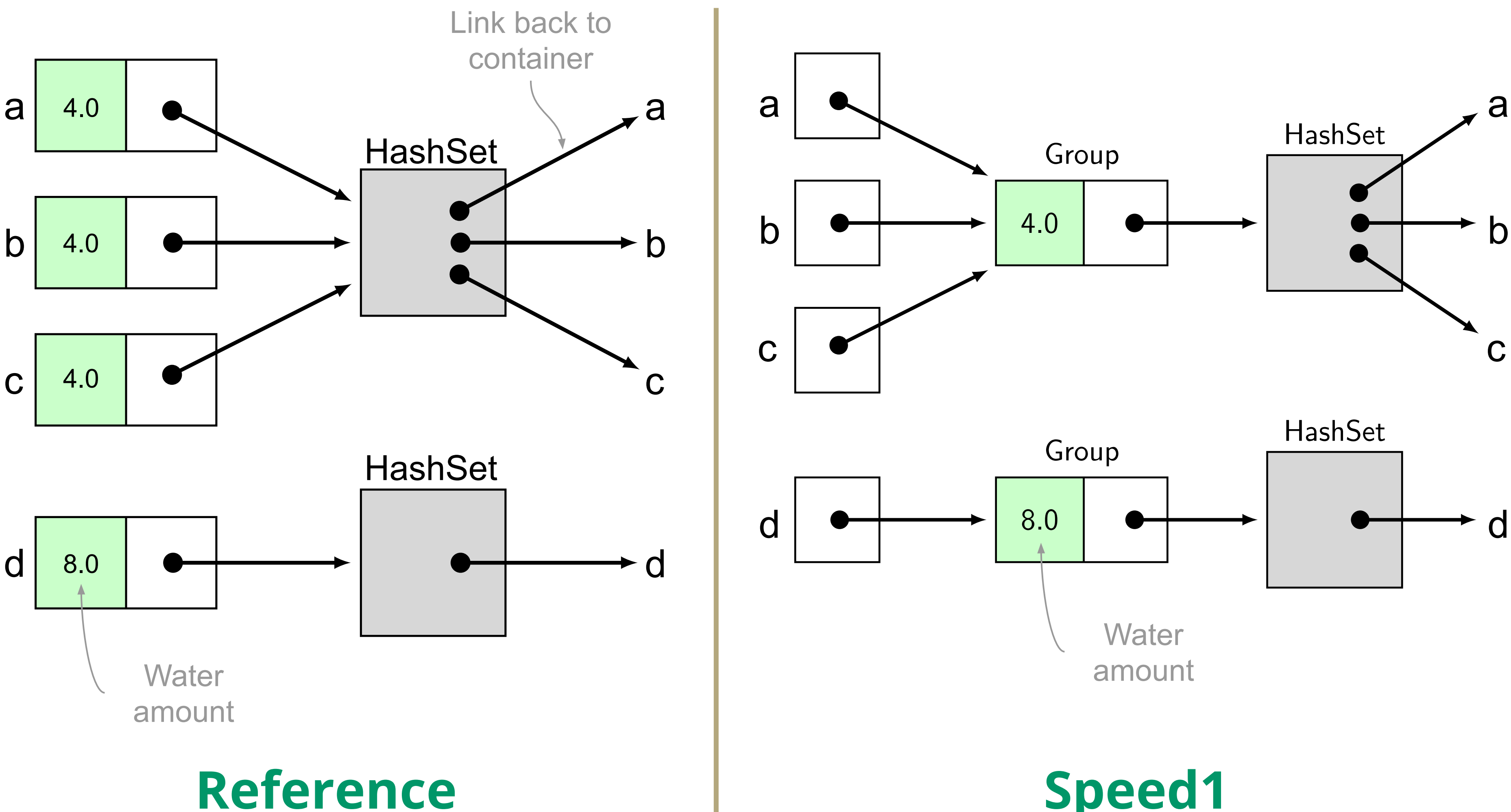
A nested class:

```
private static class Group {
    double amountPerContainer;
    Set<Container> members;

    Group(Container c) {
        members = new HashSet<>();
        members.add(c);
    }
}
```

[Speed1]

Memory layouts



Can we add water and connect in constant time?

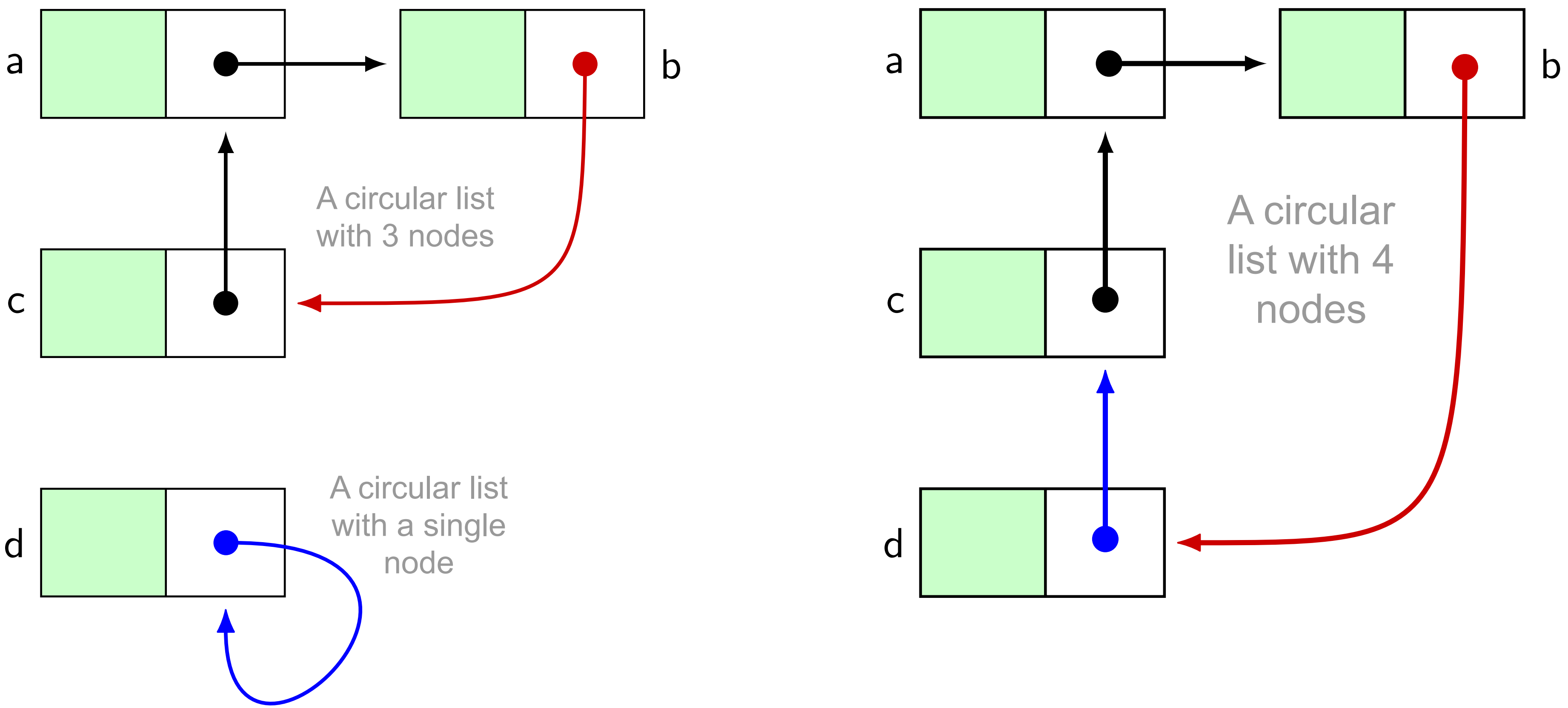
Method	Time complexity
getAmount	$O(n)$
connectTo	$\boxed{O(1)}$
addWater	$\boxed{O(1)}$

Reference:

Method	Time complexity
getAmount	$O(1)$
connectTo	$O(n)$
addWater	$O(n)$

Yes! Circular lists + laziness [Speed2]

Two circular lists can be merged in constant time



Complexity of Speed2

Method	Time complexity
`getAmount`	$O(n)$
`connectTo`	$\boxed{O(1)}$
`addWater`	$\boxed{O(1)}$

Distribute water

Merge 2 circular lists, don't touch water amounts

Add water *locally*

Implementation of Speed2

The fields:

```
double amount;
Container next;
```

Connecting two containers:

```
public void connectTo(Container other) {
    Container oldNext = next;
    next = other.next;
    other.next = oldNext;
}
```

Warning

We are not checking if *this* and *other* are already connected!

(And we cannot check in constant time)

Can we do *everything* in constant time?

Method	Time complexity
`getAmount`	$O(1)$
`connectTo`	$O(1)$
`addWater`	$O(1)$

No, but...

Union-find trees [Speed3]

- Classical data structure for maintaining *disjoint sets*

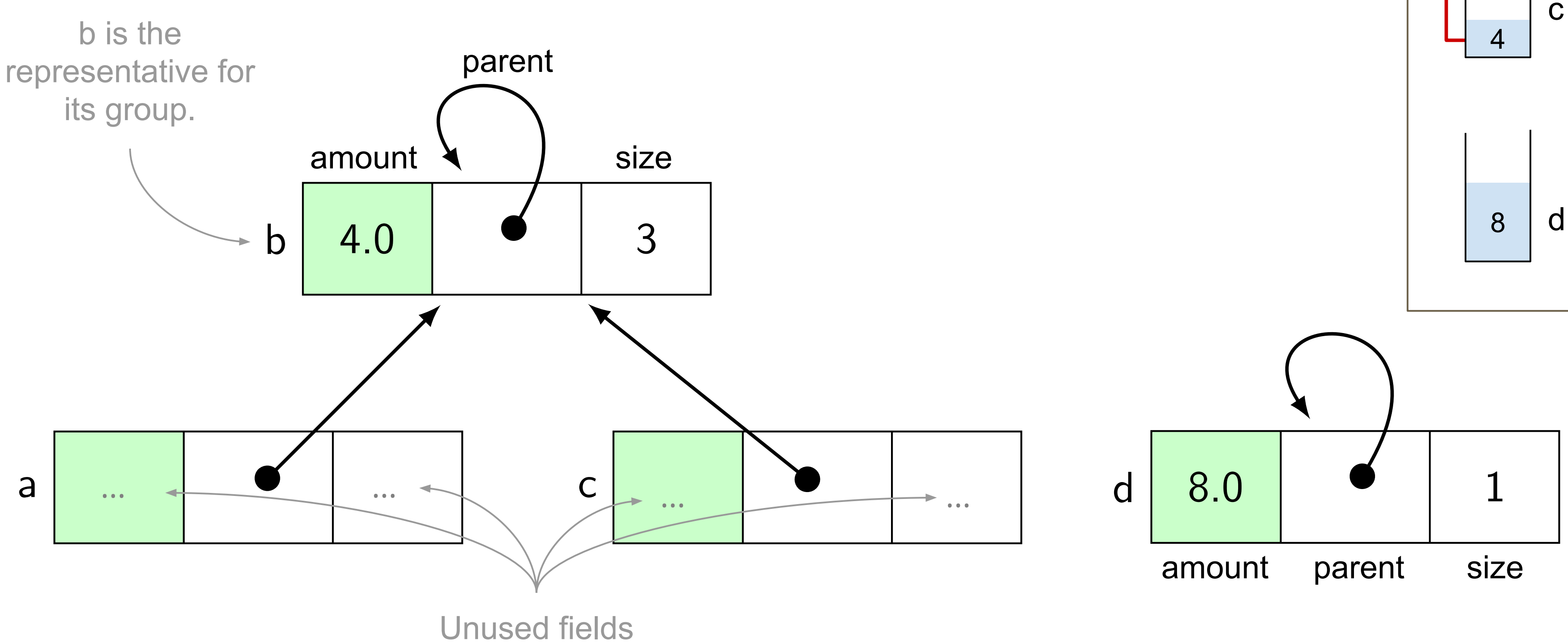
- Given a set of elements $e_1, \ldots, e_n$
- Initially, each element is isolated (a *singleton*)

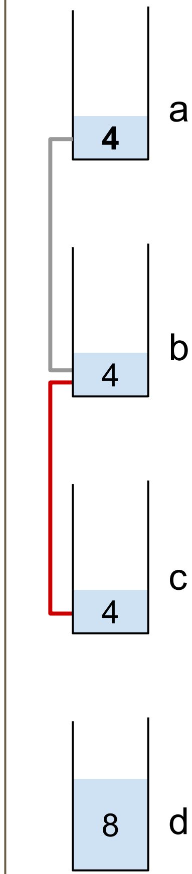
- **Union** operation: Given two elements, *merge their sets*
- **Find** operation: Given an element, obtain the *representative* of its set

- To check if two elements are in the same set, check if their representatives are the same

Union-find trees: implementation

- Each set is a *parent-pointer tree*
- Each node has three fields: amount, parent, size





Implementation of Speed3

The fields:

```
double amount;
Container parent = this;
int size = 1;
```

No constructor is needed

Groups of containers as union-find trees

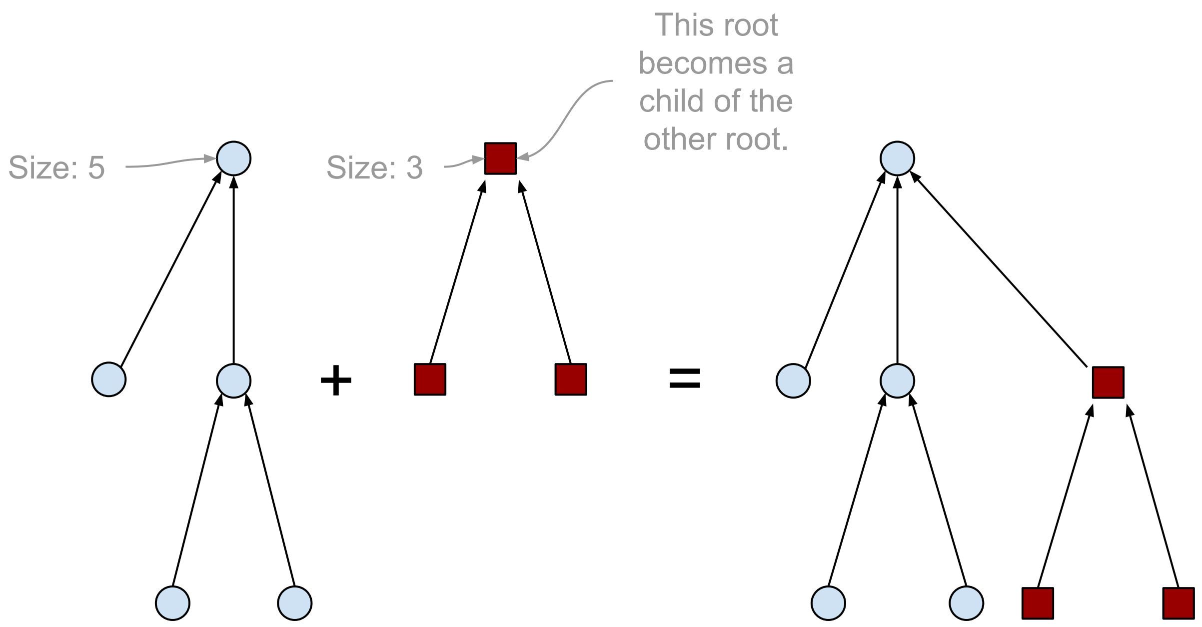
- **getAmount** (*find* operation):
  - find the root of that tree
  - return the amount field of the root
  - while applying *path compression*

- **connectTo** (*union* operation):
  - find the roots of both trees (and check that they are different)
  - merge the two trees by turning one root into a child of the other root
  - while applying the *link-by-size policy*

Link-by-size policy

When merging two trees,

link the smallest one to the root of the largest one



Link-by-size policy: worst case

Link-by-size ensures logarithmic worst-case height

Nodes	Height after merge
2	1
4	2
8	3

$3 = \log_2 8$

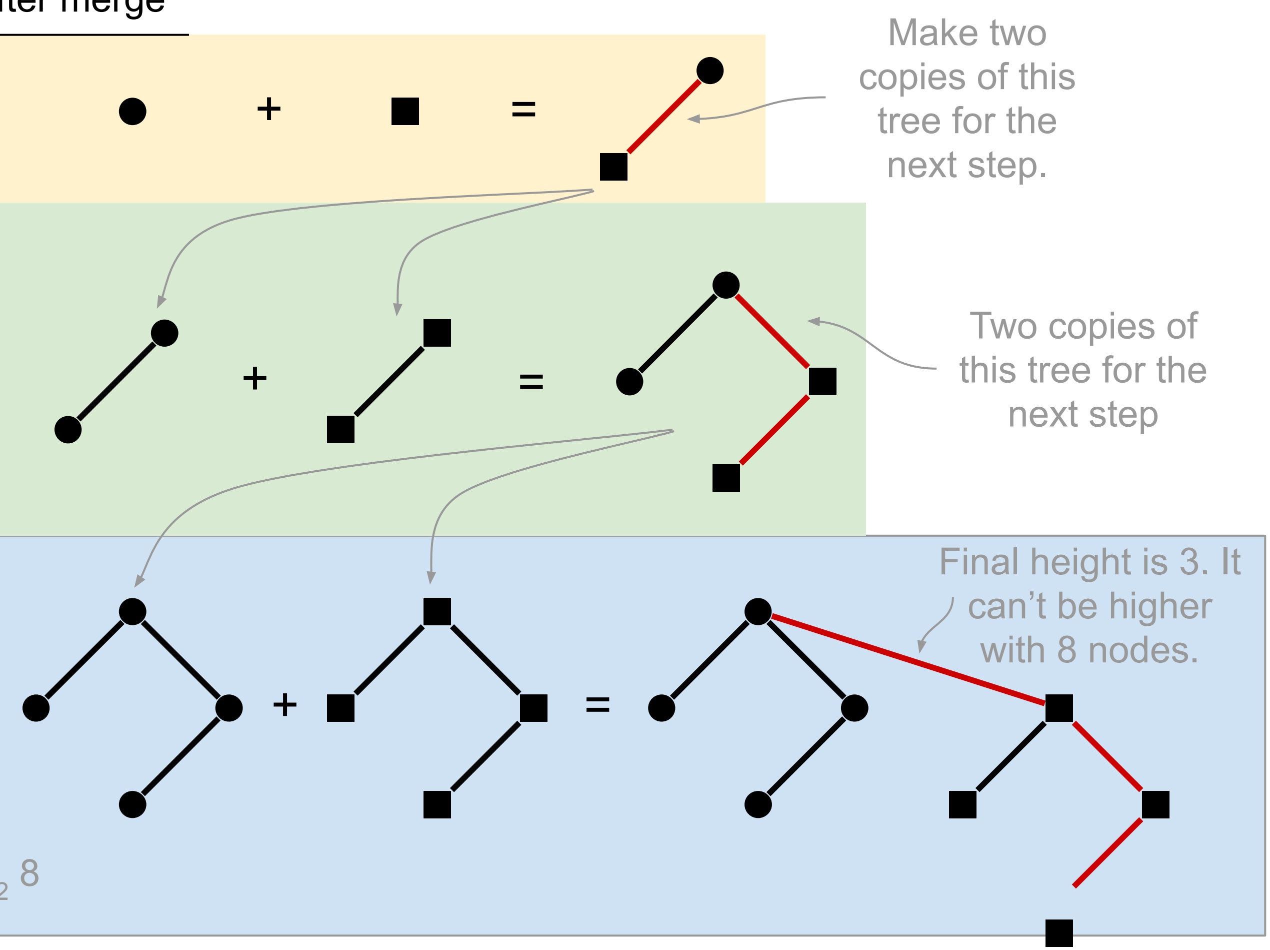
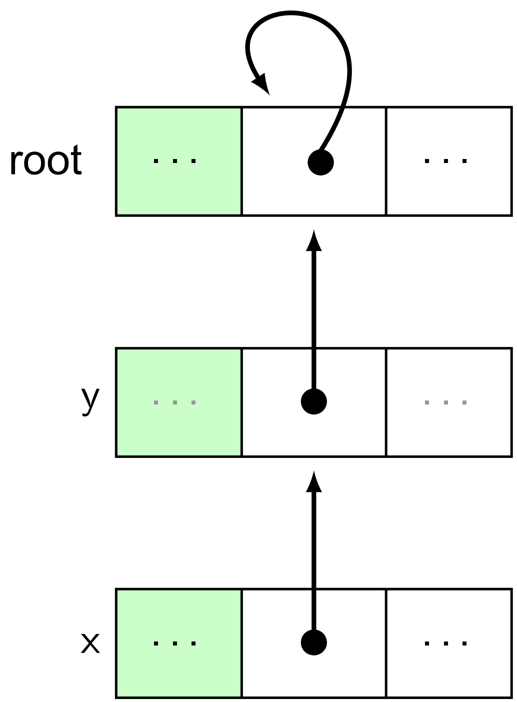


Figure from Seriously Good Software, by M. Faella © 2020 Manning Publications

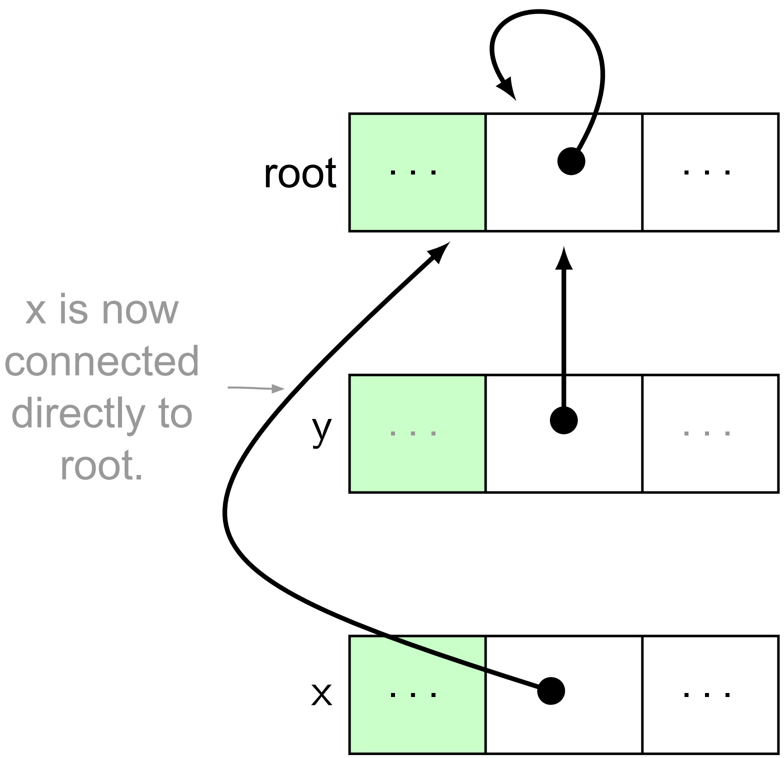
Path compression

When navigating from a node to the root, transform each node along the path into a **direct child of the root**

Before:



After:



Path compression: implementation

```java
public double getAmount() {
    Container root = findRootAndCompress();
    return root.amount;
}
```

```java
private Container findRootAndCompress() {
    if (parent != this)
        parent = parent.findRootAndCompress();
    return parent;
}
```

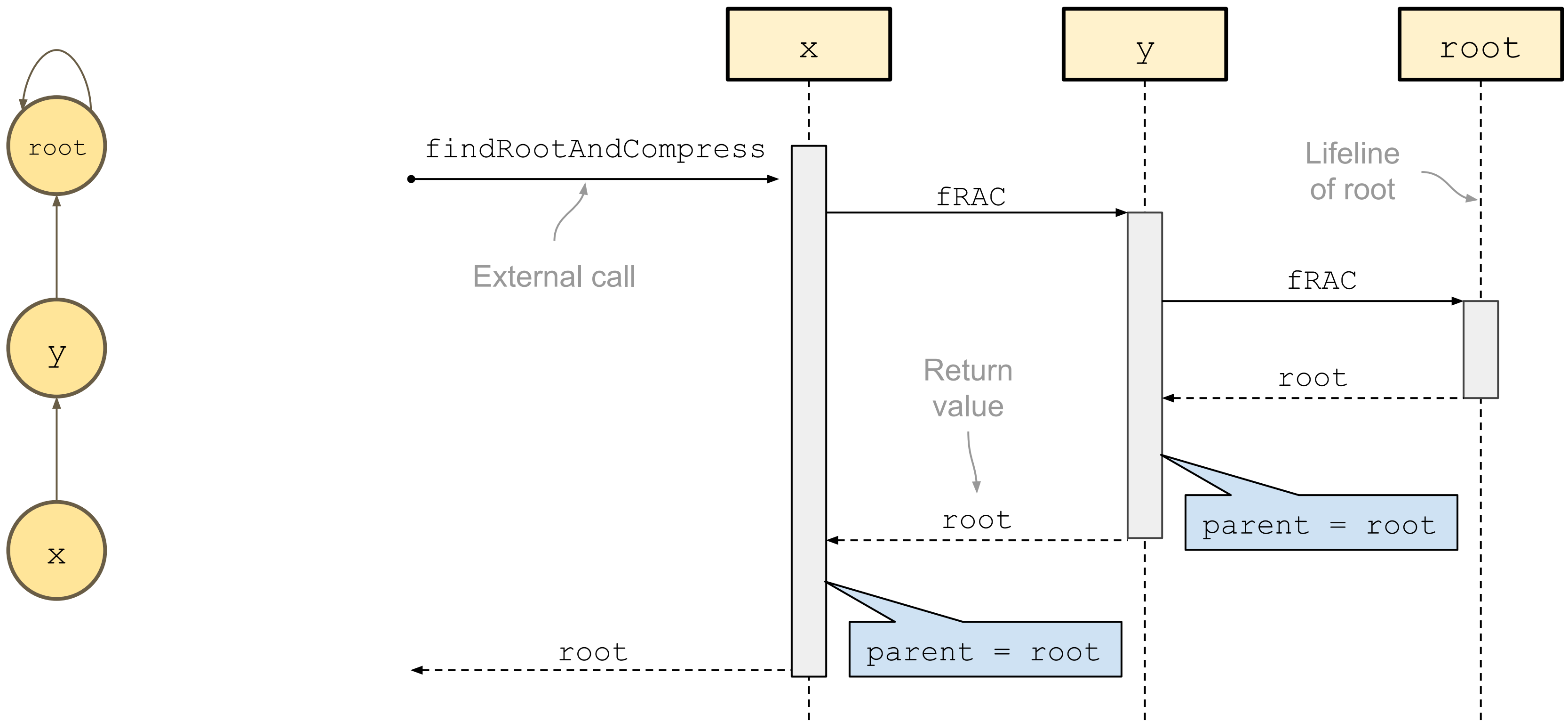


Figure from Seriously Good Software, by M. Faella © 2020 Manning Publications

Worst-case complexities

Method	Time complexity
`getAmount`	$O(\log n)$
`connectTo`	$O(\log n)$
`addWater`	$O(\log n)$

A reminder: types of complexity bounds

Worst-case complexity	Worst possible input/use case
Average-case complexity	Average over a given *distribution* of inputs
Amortized complexity	Average over a long sequence of operations

For algorithms that make *investments* for a future *benefit*

Amortized Complexity

Amortized complexity

- Fix a **sequence of n operations** on a data structure
- Compute its total cost T(n)
- The amortized complexity of the sequence is T(n)/n
  - Every step has average cost T(n)/n

Dynamically resizing arrays

ArrayList has initial capacity 10

It grows when full:

```
int newCapacity = oldCapacity + (oldCapacity >> 1);
…
elementData = Arrays.copyOf(elementData, newCapacity);
```

What’s the complexity of insertion (method add)?

- Worst-case: linear
- Amortized: ??

Dynamically resizing arrays

Consider a sequence of n insertions

Total cost: $\text{cost}(n) = \underbrace{1 + 1 + \ldots + 1}_{\text{10 adds}} + \underbrace{15}_{\text{grow}} + \underbrace{1 + 1 + \ldots + 1}_{\text{5 adds}} + \underbrace{22}_{\text{grow}} + 1 + 1 + \ldots$

Let k be the number of “grow” steps during n insertions:

$$10 * 1.5^k \geq n$$

$$k \geq \log_{1.5} \frac{n}{10}$$

So, k is the smallest integer that is at least $\log_{1.5} \frac{n}{10}$

Dynamically resizing arrays

Consider a sequence of n insertions

Total cost: $\text{cost}(n) = \underbrace{1 + 1 + \ldots + 1}_{\text{10 adds}} + \underbrace{15}_{\text{grow}} + \underbrace{1 + 1 + \ldots + 1}_{\text{5 adds}} + \underbrace{22}_{\text{grow}} + 1 + 1 + \ldots,$

$$\begin{aligned}
\text{cost}(n) &= 10 + (15 + 5) + (22 + 7) + (33 + 11) + \ldots \\
&= 10 + (15 * 1 + 5 * 1) + (15 * 1.5 + 5 * 1.5) + \left(15 * (1.5)^2 + 5 * (1.5)^2\right) + \ldots \\
&= 10 + \sum_{i=0}^{k} \left(15 * (1.5)^i + 5 * (1.5)^i\right) \\
&= 10 + 20 \sum_{i=0}^{k} (1.5)^i
\end{aligned}$$

Dynamically resizing arrays

$$\text{cost}(n) = 10 + 20 \sum_{i=0}^{k} (1.5)^i$$

where $k = \log_{1.5} \frac{n}{10}$

Apply this formula:

$$\boxed{\sum_{i=0}^{k} a^i = \frac{a^{k+1} - 1}{a - 1}}$$

$$\begin{aligned}
\text{cost}(n) &= 10 + 20 * \frac{1.5^{\left(\log_{1.5} \frac{n}{10} + 1\right)} - 1}{1.5 - 1} \\
&= 10 + 20 * \frac{1.5 * 1.5^{\left(\log_{1.5} \frac{n}{10}\right)} - 1}{0.5} \\
&= 10 + 20 * 2 * \left(1.5 * \frac{n}{10} - 1\right) \\
&= 10 + 60 * \frac{n}{10} - 40 \\
&= 6 * n - 30 \\
&= O(n) \,.
\end{aligned}$$

Amortized complexity of dynamic resizing

- When you grow by **any constant factor**, the complexity of *n* insertions is **linear** in *n*
- So, the amortized complexity of a single insertion is **constant** (O(1))

This technique is **pervasive** in programming languages:

- In Java, this applies to ArrayList, HashMap and HashSet
- In C++, this applies to vector, unordered_map, unordered_set
- In Python, this applies to lists and dictionaries
- etc.

Back to union-find trees

- Thanks to the link-by-size policy and path compression, the complexity of any sequence of m find and union operations is *almost* linear in m

Theorem [Tarjan, 1975]

Any sequence of m union or find operations on n elements takes at most O(m $\alpha(n)$) time, where α() is the *inverse Ackermann function*.

- $\alpha(n)$ is at most 4 for all n up to 10^{80}
- Cost of a single operation is essentially **constant**

Amortized complexity of Speed3

Scenario	Amortized time complexity
A sequence of m operations on n containers	$O(m * \alpha(n))$

Inverse Ackermann function

Comparing implementations

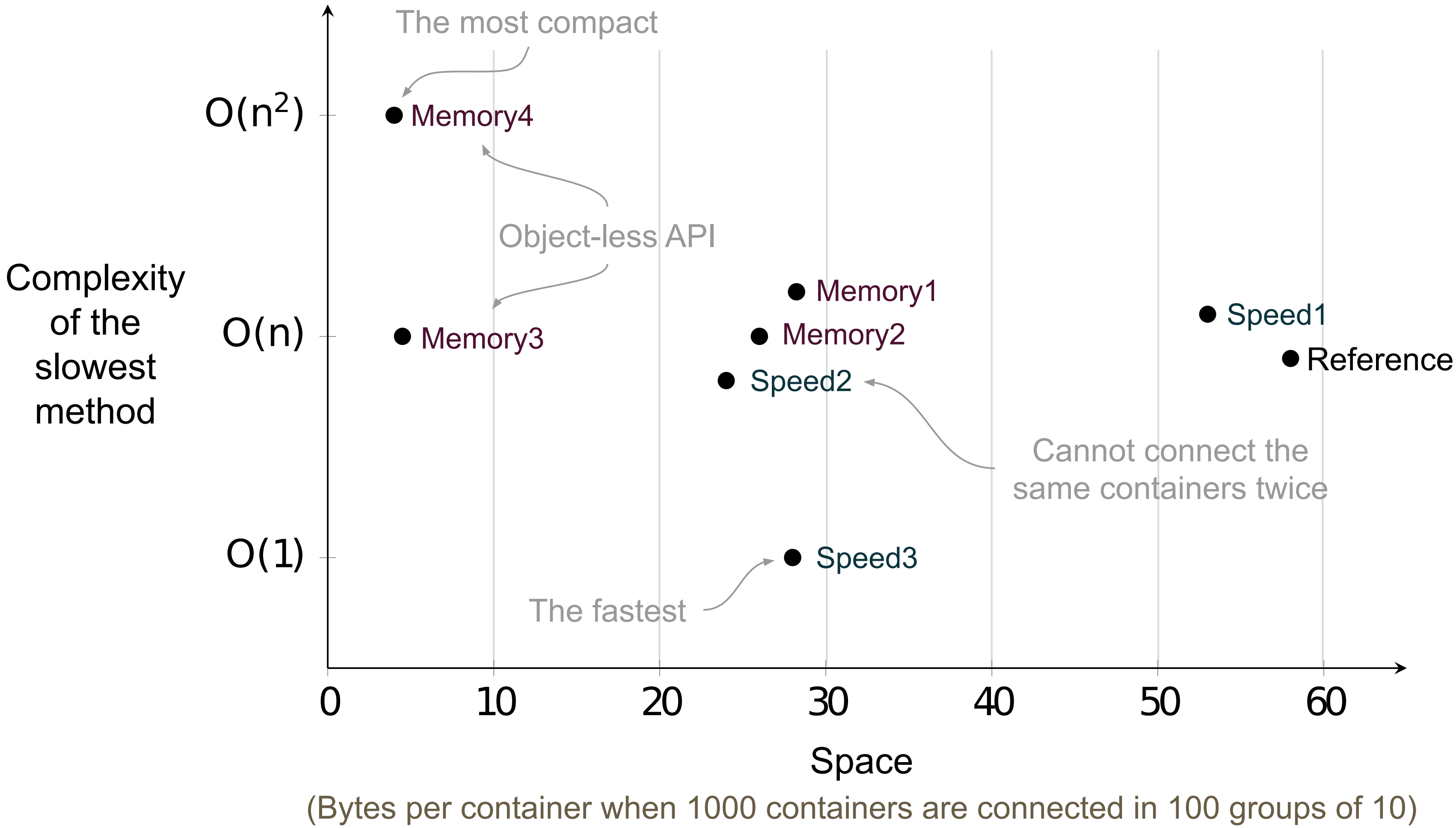


Figure from Seriously Good Software, by M. Faella © 2020 Manning Publications

Let's run it

Experiment 1: 20k constructor
40k addWater
20k connectTo
20k getAmount

Version	Time (msec)
Reference	2 300
Speed1	26
Speed2	505
Speed3	6

Experiment 2: 20k constructor
40k addWater
20k connectTo
1 getAmount

Version	Time (msec)
Reference	2 300
Speed1	25
Speed2	4
Speed3	5

Let's Run It

1. Create 20k containers and add some water
2. Connect containers in 10K pairs
3. Add some water to each pair
4. Query the amount in each pair
5. Connect pairs until they are all connected, while adding water and querying the amount

Total: 20k constructor
40k addWater
20k connectTo
20k getAmount

Version	Time (msec)
Reference	2 300
Speed1	26
Speed2	505
Speed3	6

Let's Run It Again

1. Create 20k containers and add some water
2. Connect containers in 10K pairs
3. Add some water to each pair
4. ~~Query the amount in each pair~~
5. Connect pairs until they are all connected, while adding water and ~~querying the amount~~
6. Query the final amount

Total: 20k constructor
40k addWater
20k connectTo
1 getAmount

Version	Time (msec)
Reference	2 300
Speed1	25
Speed2	$\boxed{4}$
Speed3	5

Time efficiency: conclusions

Conclusion 0: Do you really need more speed?

Conclusion 1: Trust asymptotic complexity

- In its various forms …

Conclusion 2: Profiling and optimizing

- Pay attention to your usage profile
- Consider shifting the effort from one place to another

Further reading

Kevin Wayne’s slides on union-find trees:

https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf