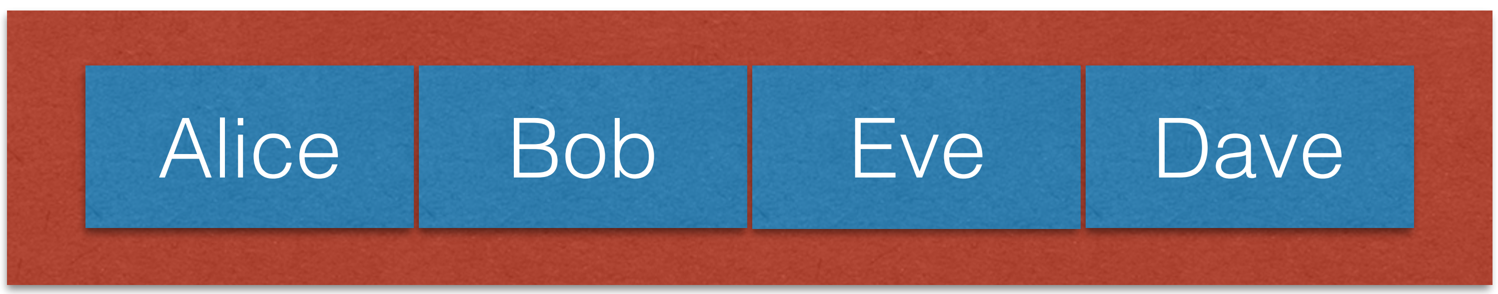
Functional Data Structures

Sept 9, 2016

(Multiple diagrams from ‘Purely Functional Datastructures’ by Chris Okasaki)

Mutable vs Immutable

`X = [ Alice, Bob, Carol, Dave ]`



`X[2]`

`X[2] := Eve`

Mutable vs Immutable

```
X = [ Alice, Bob, Carol, Dave ]
```

Alice | Bob | Carol | Dave

<u>Thread 1</u>

```
X[2] := Eve
```

<u>Thread 2</u>

```
X[2]
```

Carol ? Eve

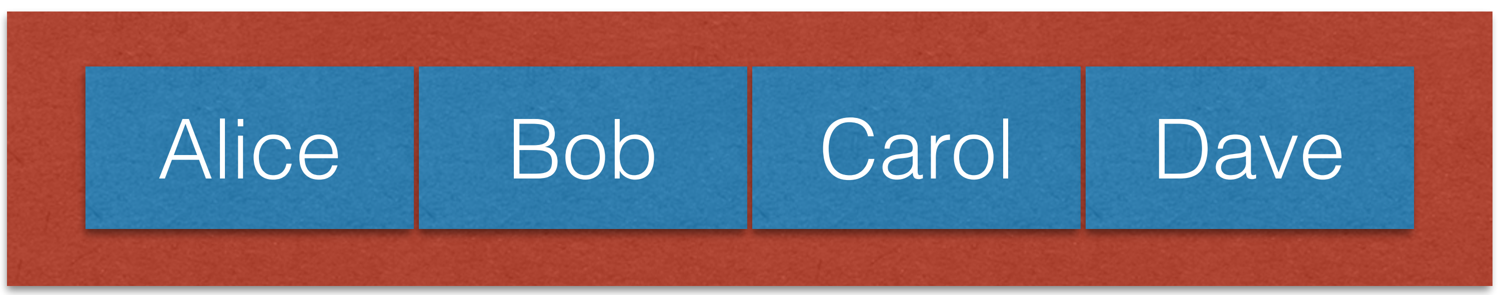
Mutable Datastructures

- The programmer's intended ordering is unclear
- Atomicity/Correctness requires locking
- Versioning requires copying the data structure
- Cache coherency is expensive!

Can these problems be avoided?

Immutable Data Structures

`X = [ Alice, Bob, Carol, Dave ]`

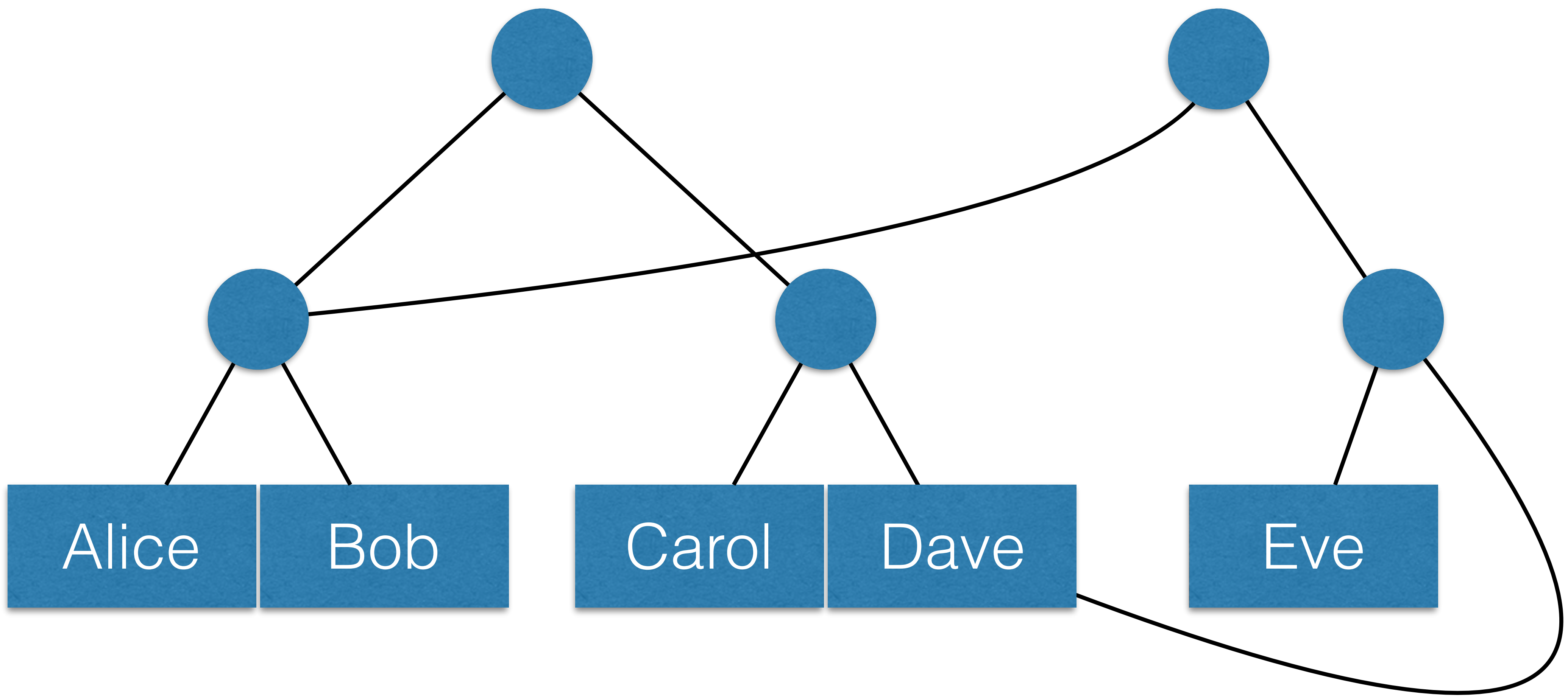


`X[2]` → Carol

~~`X[2] := Eve`~~ **Don’t allow writes!**

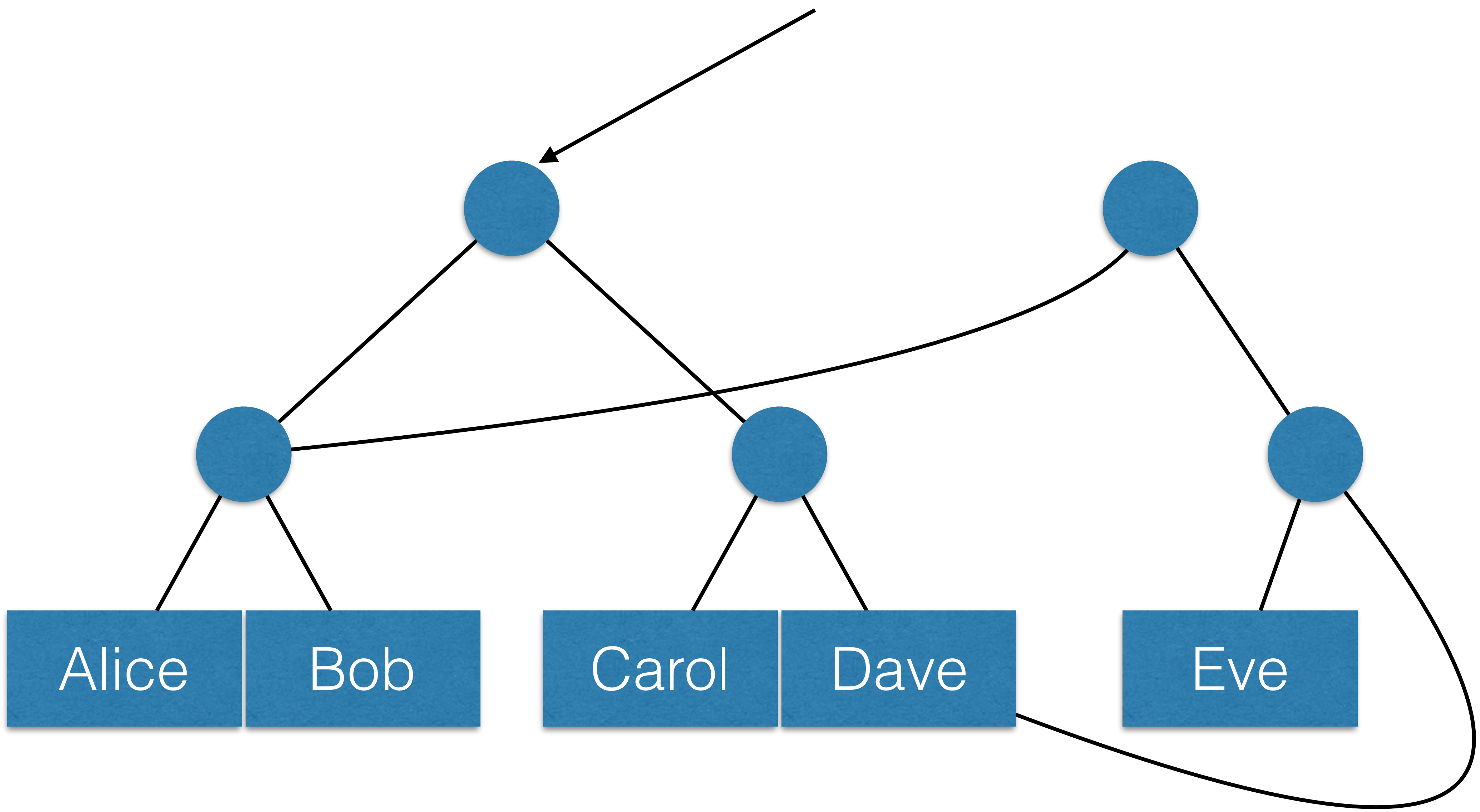
But what if we need to update the structure?

Immutable Data Structures



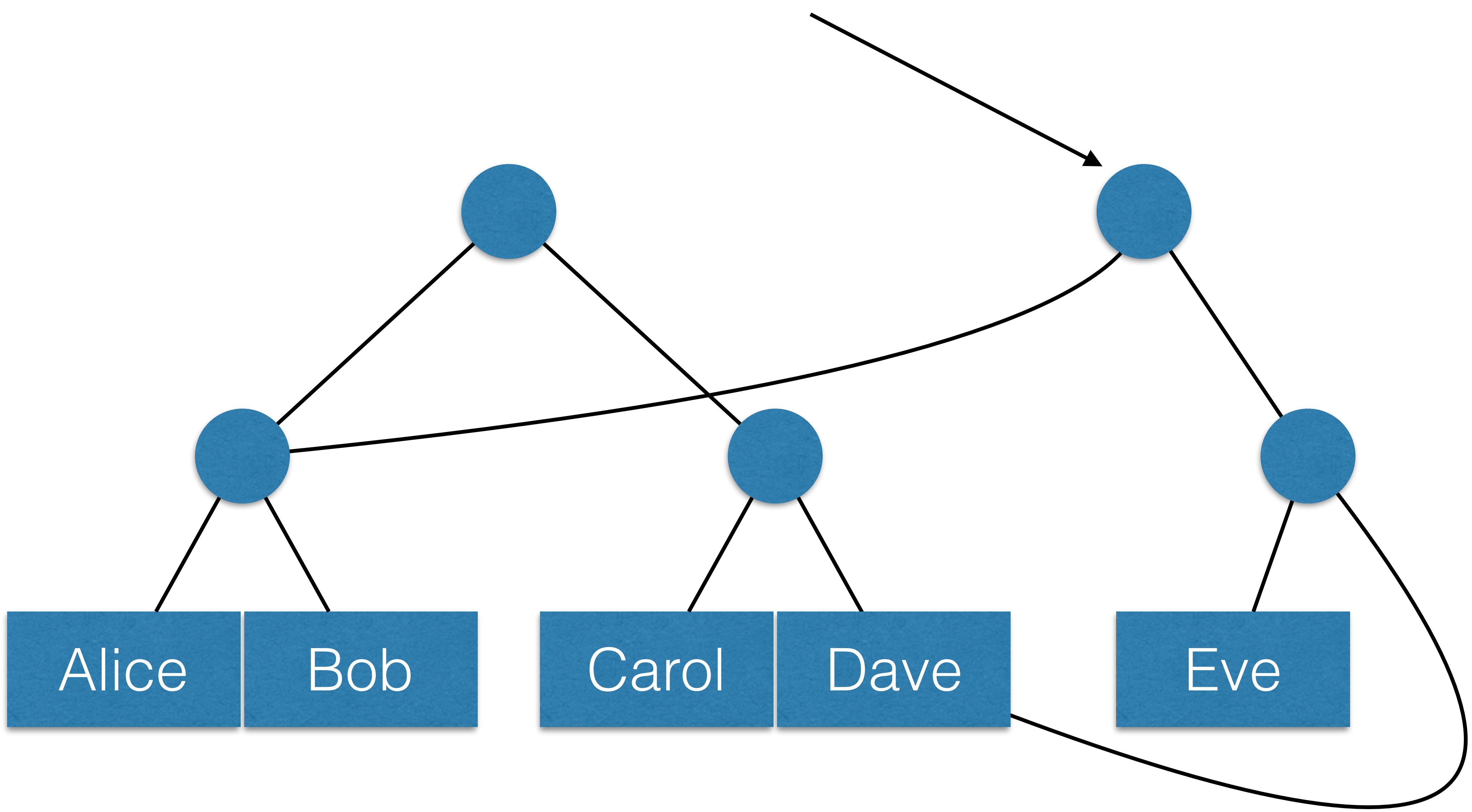
Key Insight: Immutable components can be re-used!

Immutable Data Structures



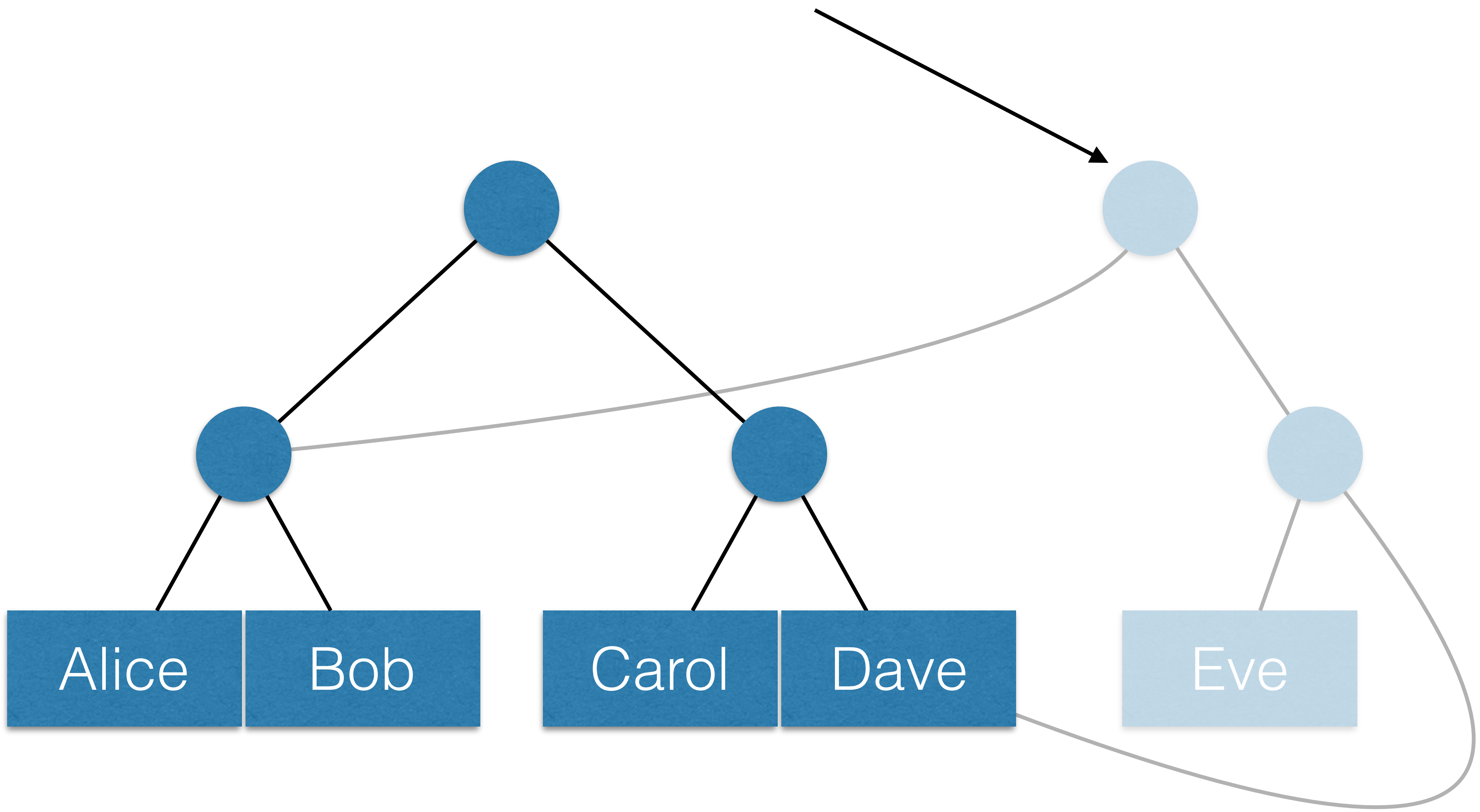
Key Insight: Immutable components can be re-used!

Immutable Data Structures



Semantics are clearer: Exactly one 'version' at any time

Immutable Data Structures



Data is added, not replaced: No cache coherency problems

Immutable Data Structures

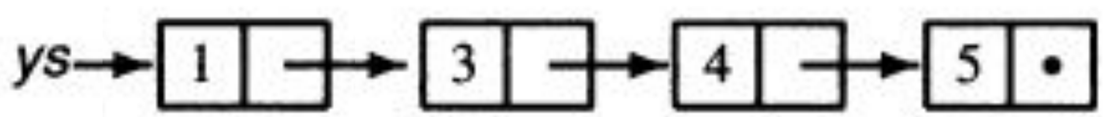
(a.k.a. ‘Functional’ or ‘Persistent’ Data Structures)

- Once an object is created, it never changes.
- When all pointers to an object go away, the object is garbage collected.
- Only the ‘root’ pointer can ever change (to point to a new version of the data structure)

Linked Lists


```
xs = pop(xs)
```

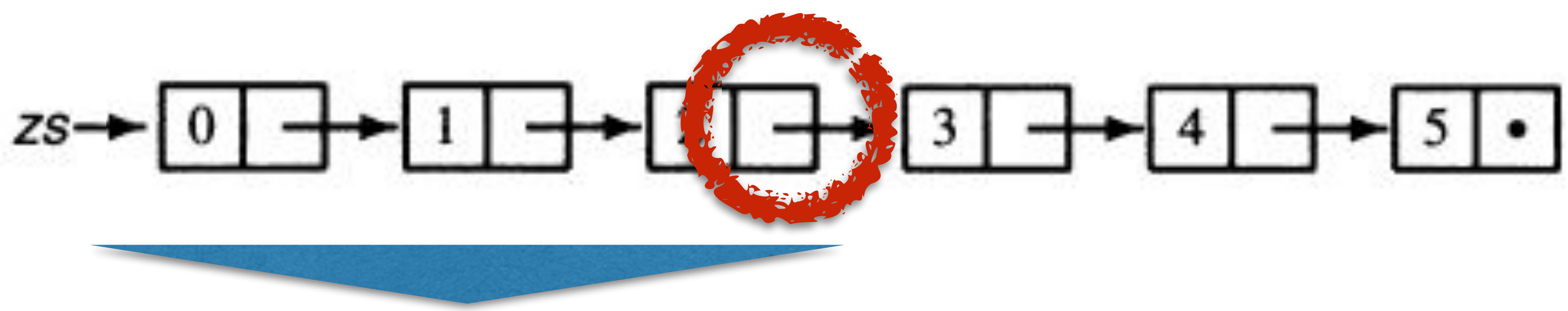

```
ys = push(ys,1)
```



Only xs and ys need to change

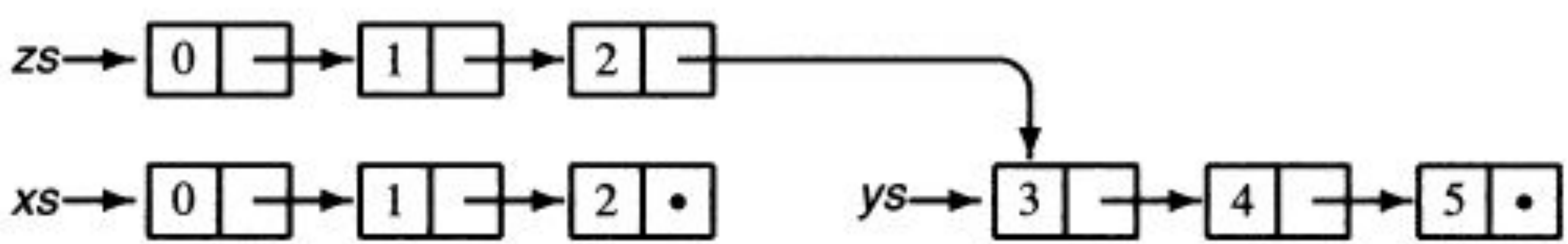
Linked Lists

`zs = append(xs,ys)`



This entire part needs to be rewritten

Linked Lists



Class Exercise 1

How would you implement

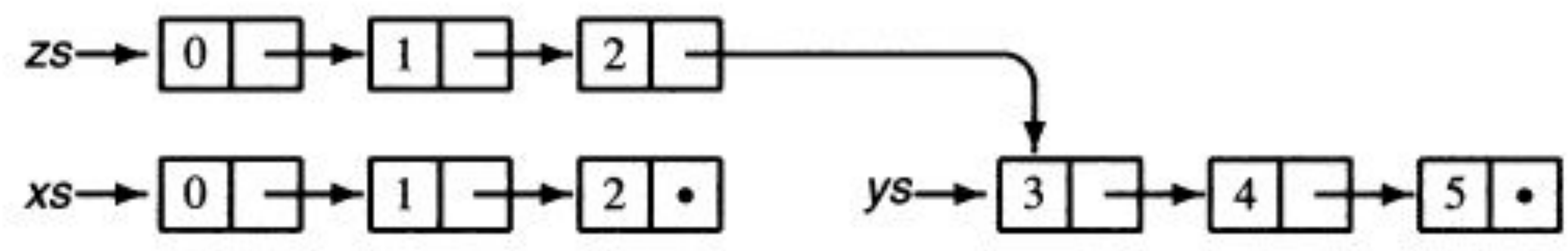
`update(list, index, value)`

Class Exercise 2

Implement a set with:

```
set init()
boolean member(set, elem)
set insert(set, elem)
```

Lazy Evaluation

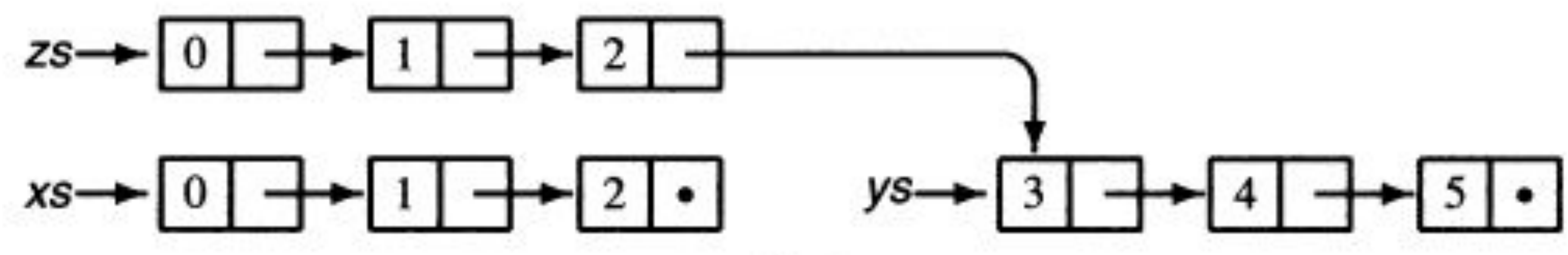


Can we do better?

Putting Off Work

`x = “expensive()”`	Fast (just saving a ‘todo’)
`print x`	Slow (performing the ‘todo’)
`print x`	Fast (‘todo’ already done)

Class Exercise 3



Make it better!

Putting Off Work

```
concatenate(a, b) {
  a’, front = pop(a)
  if a’ is empty
    return (front, b)
  else
    return (front, “concatenate(a’,b)”)
}
```

What is the time complexity of concatenate?
What happens to reads?

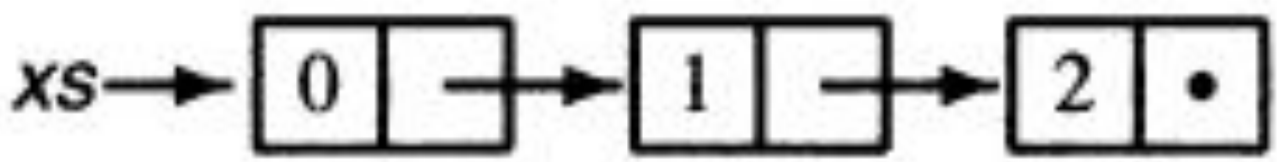
Lazy Evaluation

- Save work for later…
  - … and avoid work that is never required.
  - … to spread work out over multiple calls.
  - … for better ‘amortized’ costs.

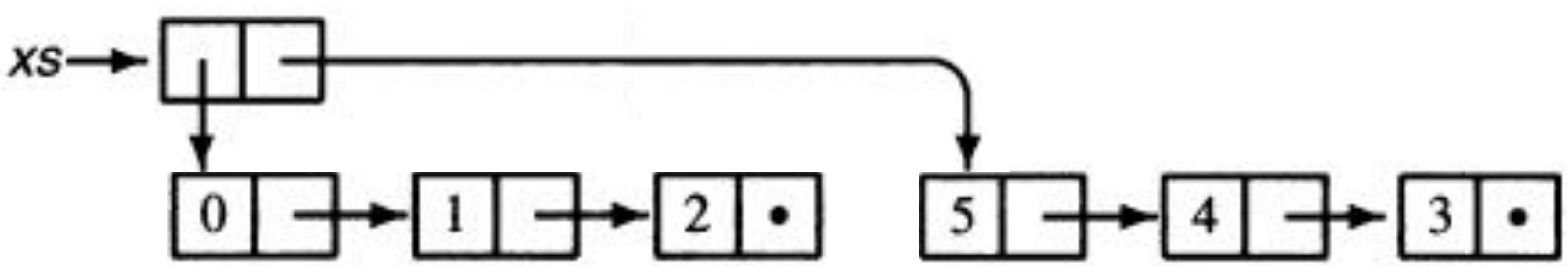
Amortized Analysis

- Allow operation A to ‘pay it forward’ for another operation B that hasn’t happened yet
  - A’s time complexity goes up by X.
  - B’s time complexity goes down by X.

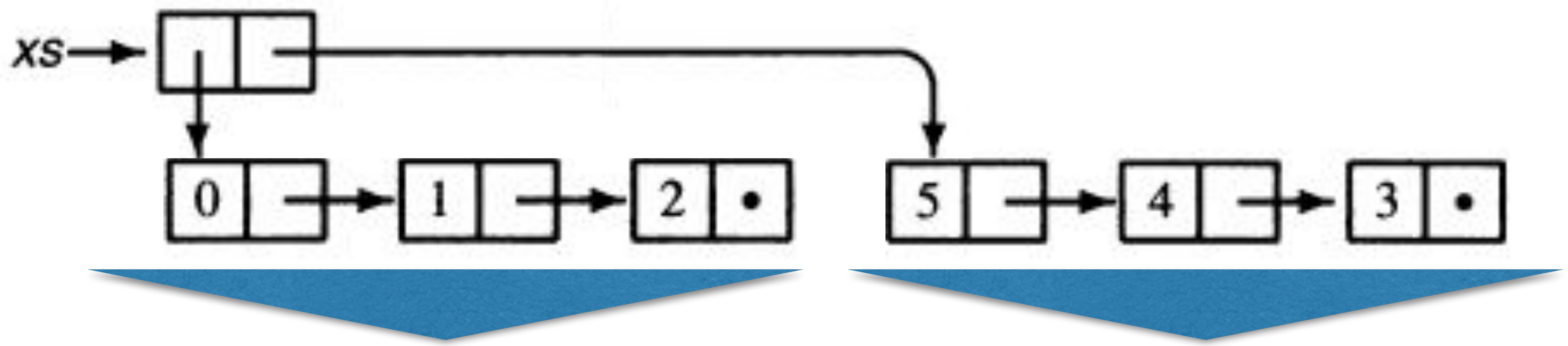
Example: Amortized Queues



Preliminaries: Implement an efficient `enqueue()`/`dequeue()`



Example: Amortized Queues



'current' queue

'todo' stack

`enqueue()`: Push onto 'todo' stack

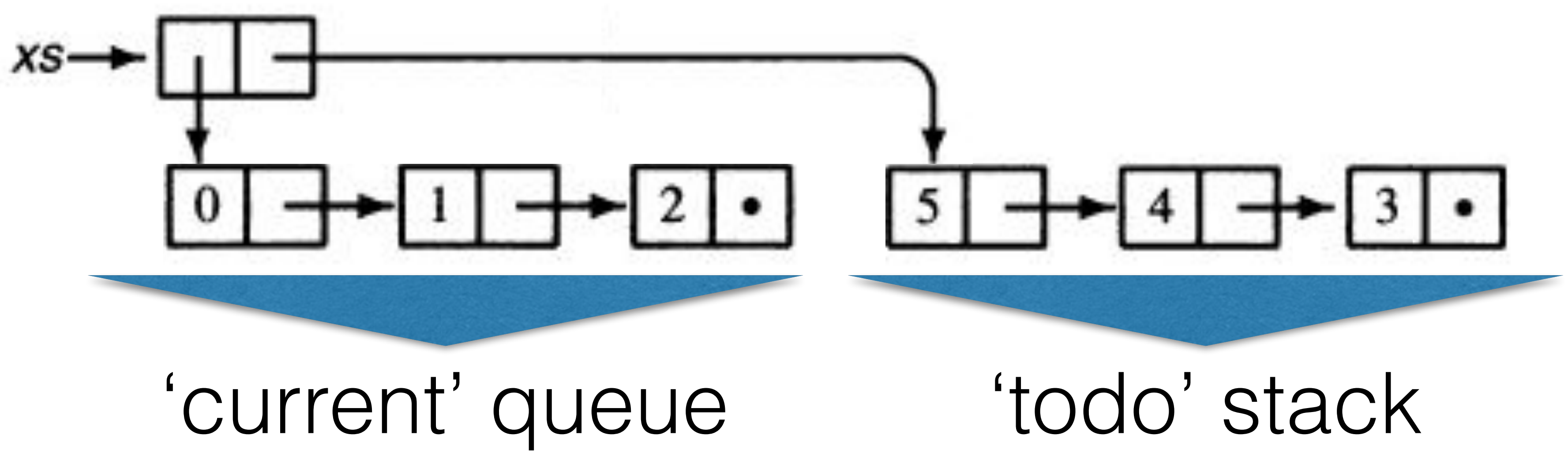
What is the cost?

`dequeue()`: Pop 'current' queue

if empty, reverse 'todo' stack to make new 'current' queue

What is the cost?

Example: Amortized Queues



`enqueue()`: Push onto 'todo' stack

push() is O(1) + 1 credit

`dequeue()`: Pop 'current' queue

if empty, reverse 'todo' stack to make new 'current' queue

Pop is O(1); Reverse uses N credits for O(1) amortized