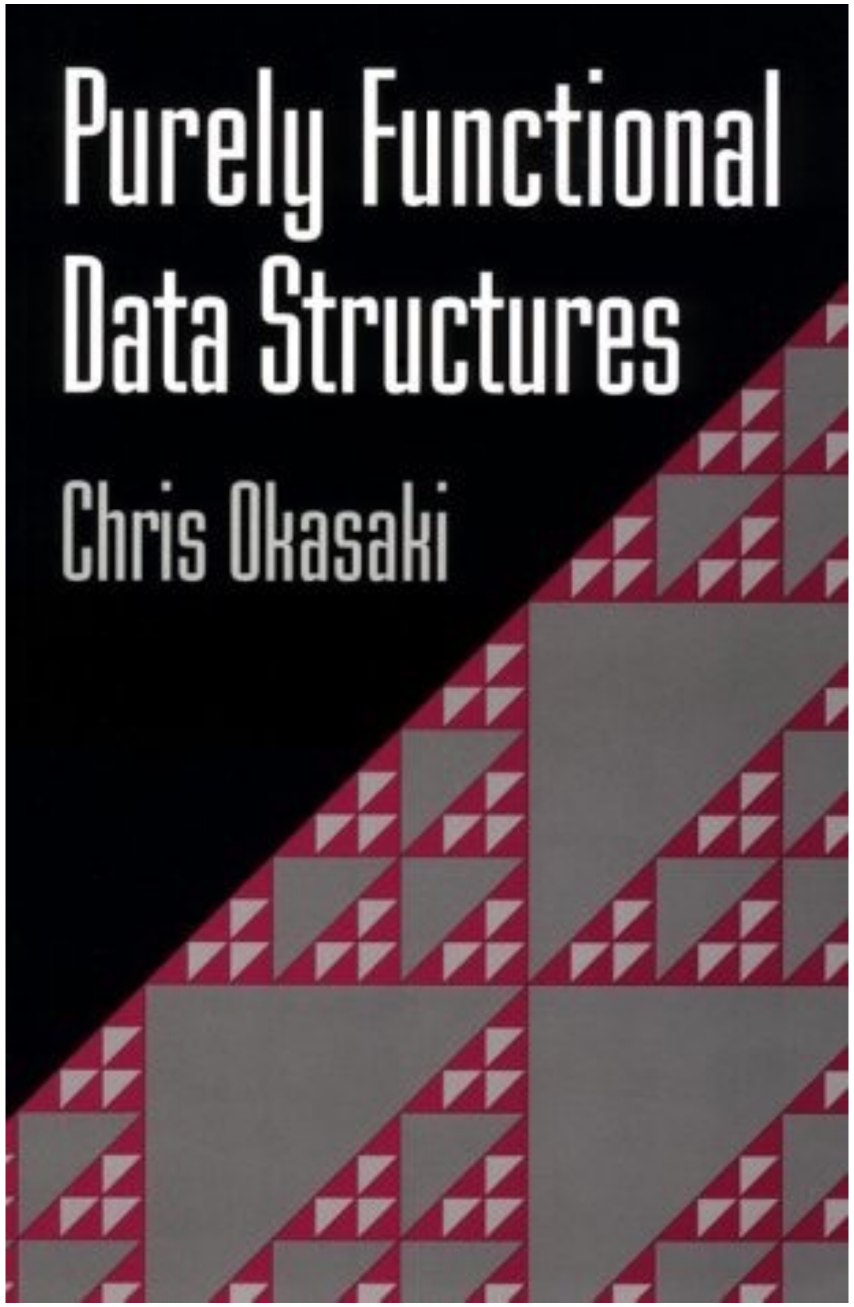
Purely Functional Data Structures



Outline

- **Persistence**
- Functional vs. imperative data structures
- Example: red-black trees
- Amortized complexity analysis
- Amortization for persistent data structures

Immutability/persistence of data in FP

Persistence: updates do not affect existing references

Haskell:

```
xs = [1,2,3]
ys = [7,8]
zs = xs ++ ys

> zs
[1,2,3,7,8]

> xs
[1,2,3]
```

*data is **persistent***

Ruby:

```
xs = [1,2,3]
ys = [7,8]
zs = xs.concat(ys)

> zs
[1,2,3,7,8]

> xs
[1,2,3,7,8]
```

*data is **ephemeral***

Persistent data structures

Ephemeral (i.e. traditional) data structures:

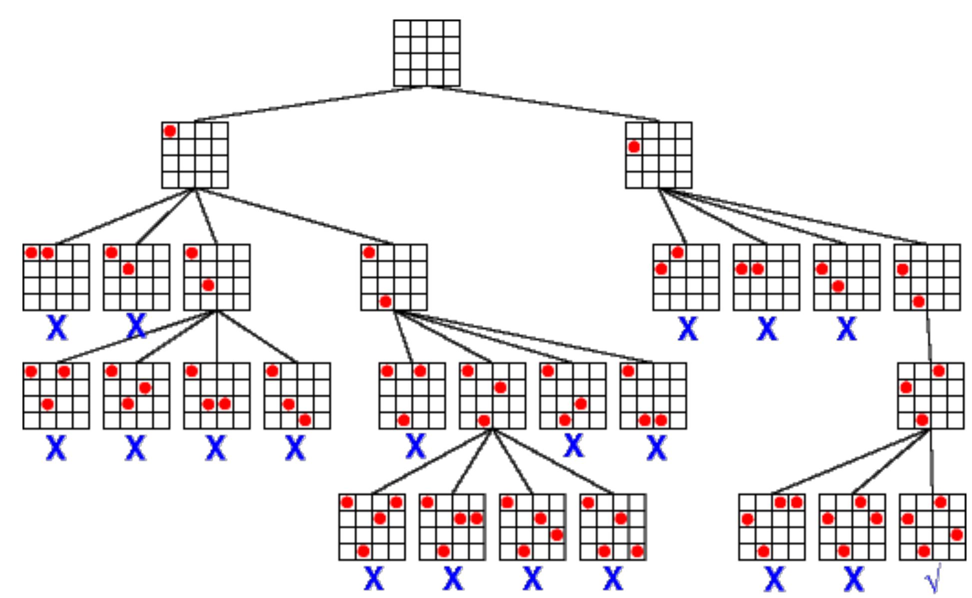
- updates destroy old versions

Persistent data structures:

- old versions are unchanged by updates

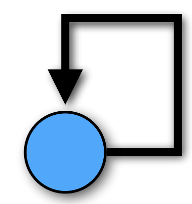
Applications independent of pure FP:

- editors (undo), version control, etc.
- backtracking search
- thread-safe data sharing
- computational geometry algorithms

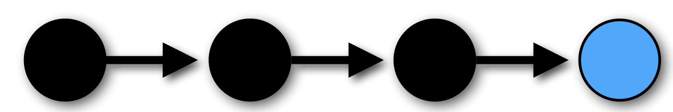


Degrees of persistence

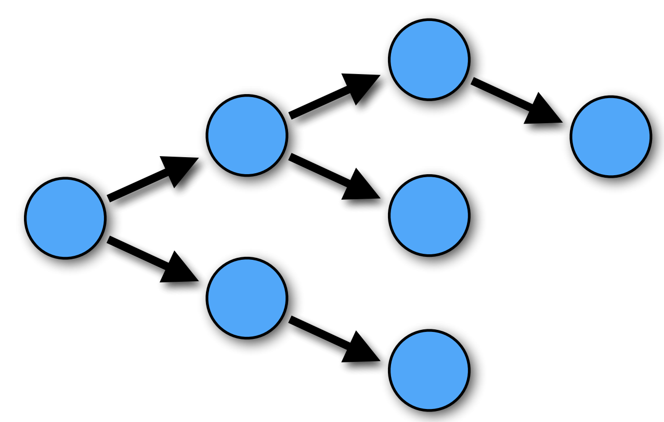
- **no persistence**
  one version



- **partial persistence**
  update only last version



- **full persistence**
  update all versions



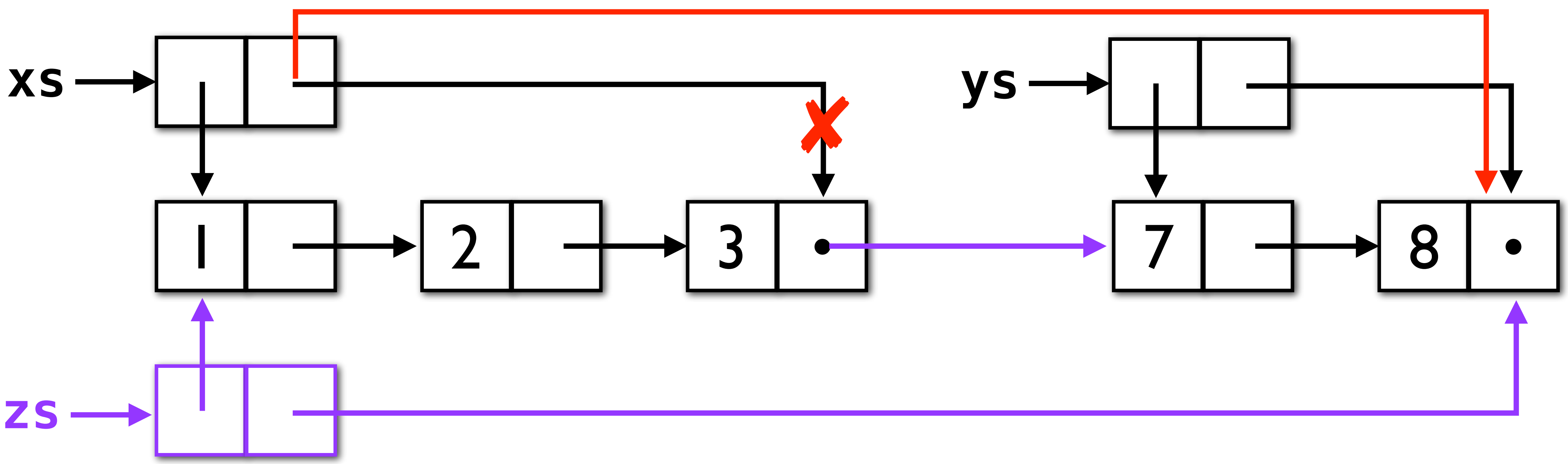
Purely functional** = all data structures are **fully persistent

Outline

- Persistence
- **Functional vs. imperative data structures**
- Example: red-black trees
- Amortized complexity analysis
- Amortization for persistent data structures

Example: imperative list concatenation

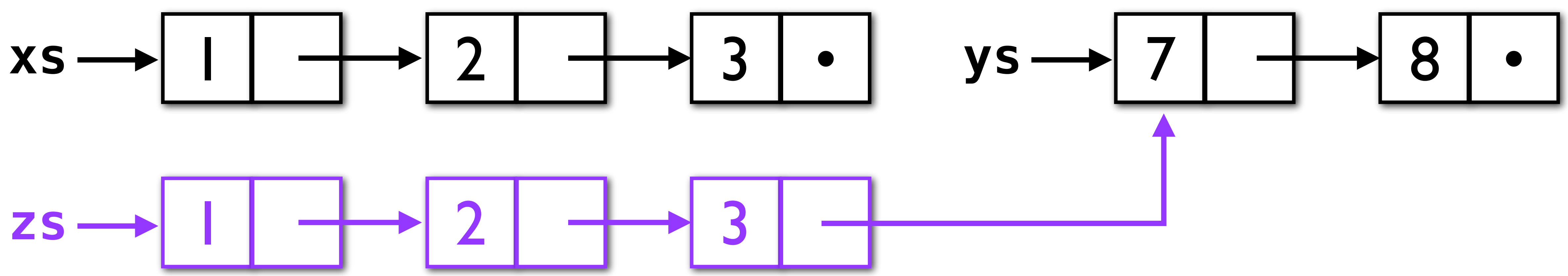
```
xs = [1,2,3]
ys = [7,8]
zs = xs.concat(ys)
```



- efficient: O(1) time and space
- side-effects (error prone!)
- not persistent

Example: functional list concatenation

```
xs = [1,2,3]
ys = [7,8]
zs = xs ++ ys
```



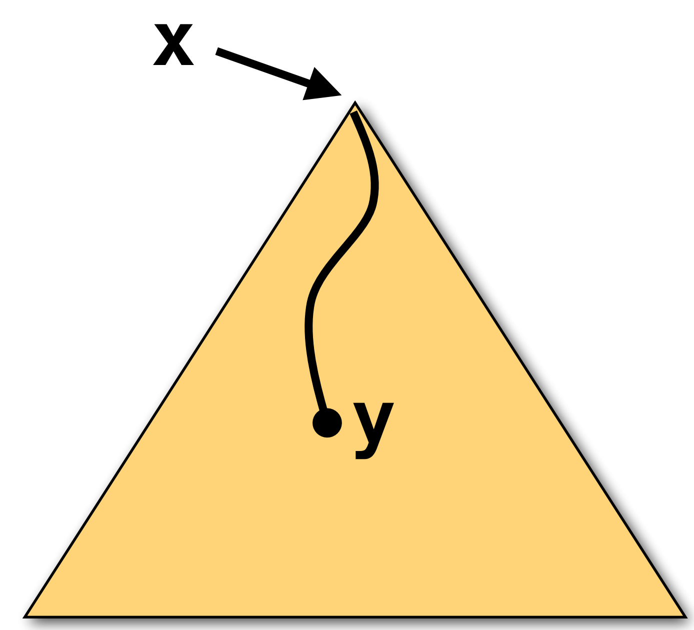
- O(|xs|) time and space requirement
- no side-effects (safe!)
- fully persistent

Model of functional data structures

```
data T = C ... | D ...

x :: T
x = D (C ...) ...
```

x is represented as a pointer data structure (tree/graph) in the heap

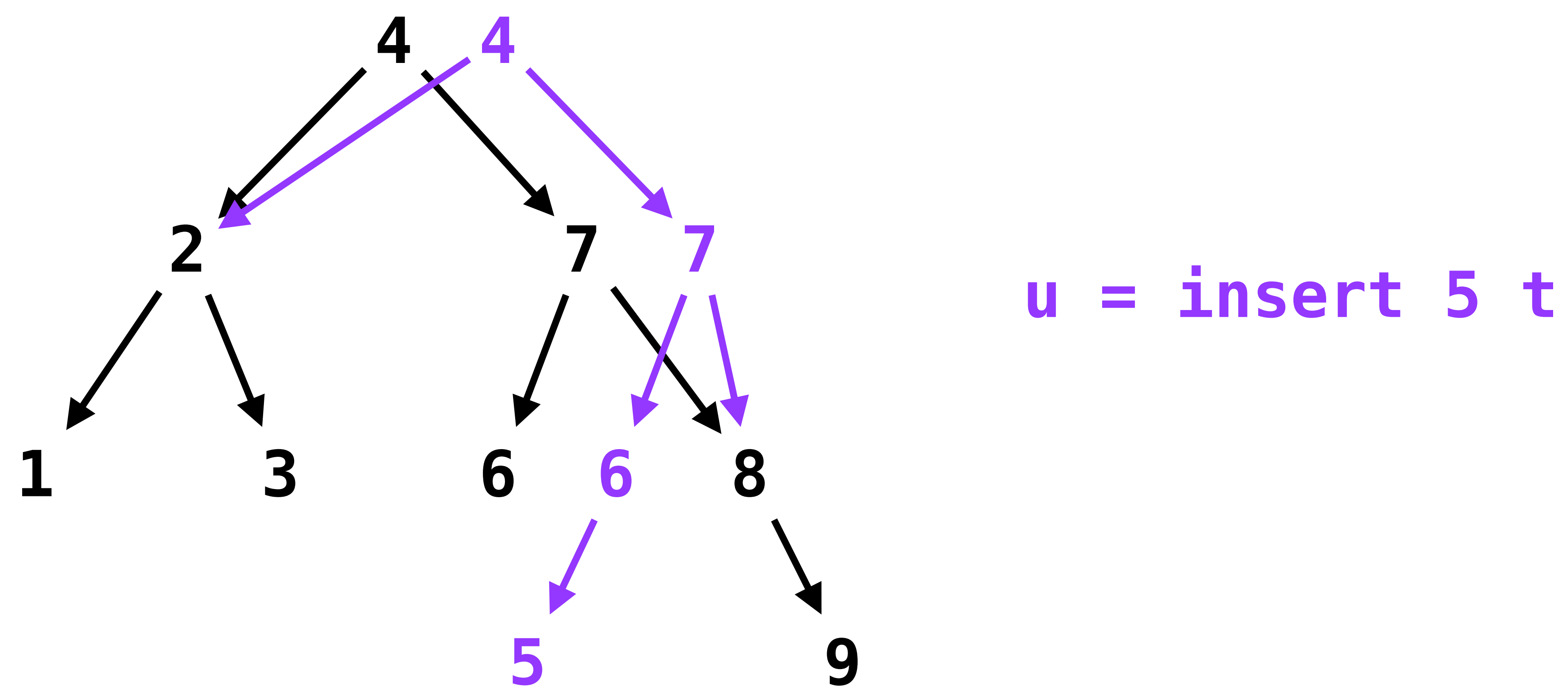


To update the subterm **y**:

- *update a copy* of the corresponding cell **y** in the heap
- *copy* all nodes on the *path* from the root to **y**
- (rest of the data structure is *shared* between **x** and **y**)

Example: insert in binary search tree

```
insert :: Ord a => a -> Tree a -> Tree a
insert x Leaf                             = Node x Leaf Leaf
insert x (Node y l r) | x < y     = Node y (insert x l) r
                      | otherwise = Node y l (insert x r)
```




```
t = Node 4 (Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf))
           (Node 7 (Node 6 Leaf Leaf) (Node 8 Leaf (Node 9 Leaf Leaf))
```

Challenges

How to *implement* functional data structures *efficiently*?

- Optimize data type representation for common operations
- Goals: *minimize traversal* and *copying*
    - *e.g. Haskell lists are optimized for* ***stack*** *operations but inefficient as* ***queues***
    - *these goals are the rationale for the* ***zipper*** *pattern*

How to *analyze* their time and space complexity?

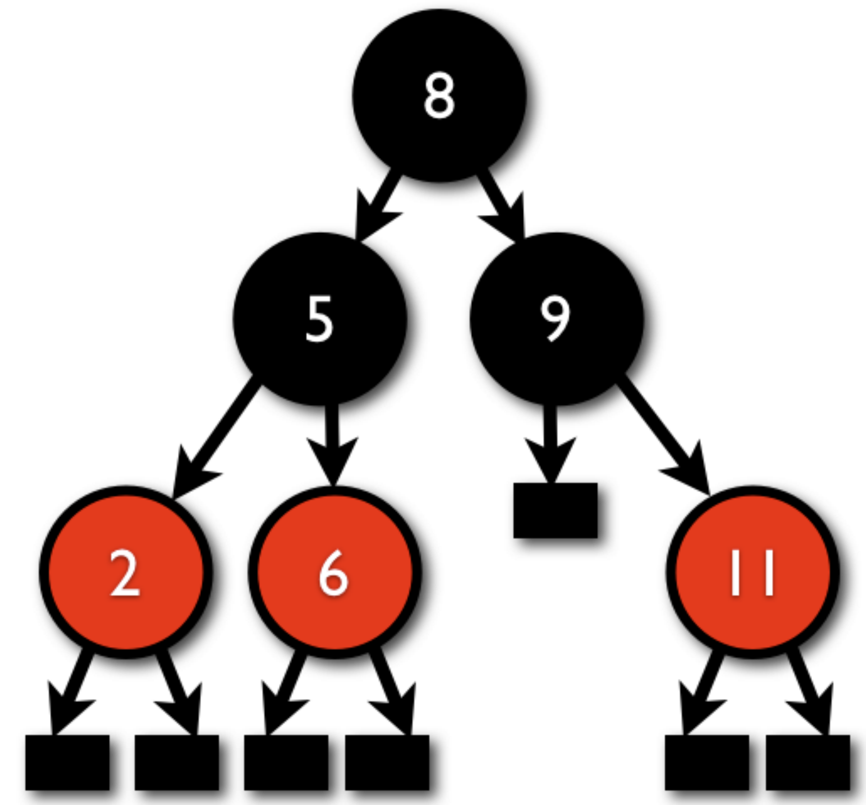
- *Worst-case analysis* is basically the same
- *Amortized analysis* is much harder!
    - Lazy evaluation is crucial for amortizing w/ persistence

(Queue.hs)

Outline

- Persistence
- Functional vs. imperative data structures
- **Example: red-black trees**
- Amortized complexity analysis
- Amortization for persistent data structures

Red-black trees



A **self-balancing** binary search tree:

- every node is **red** or **black**
- leaves are valueless and **black**

Invariants:

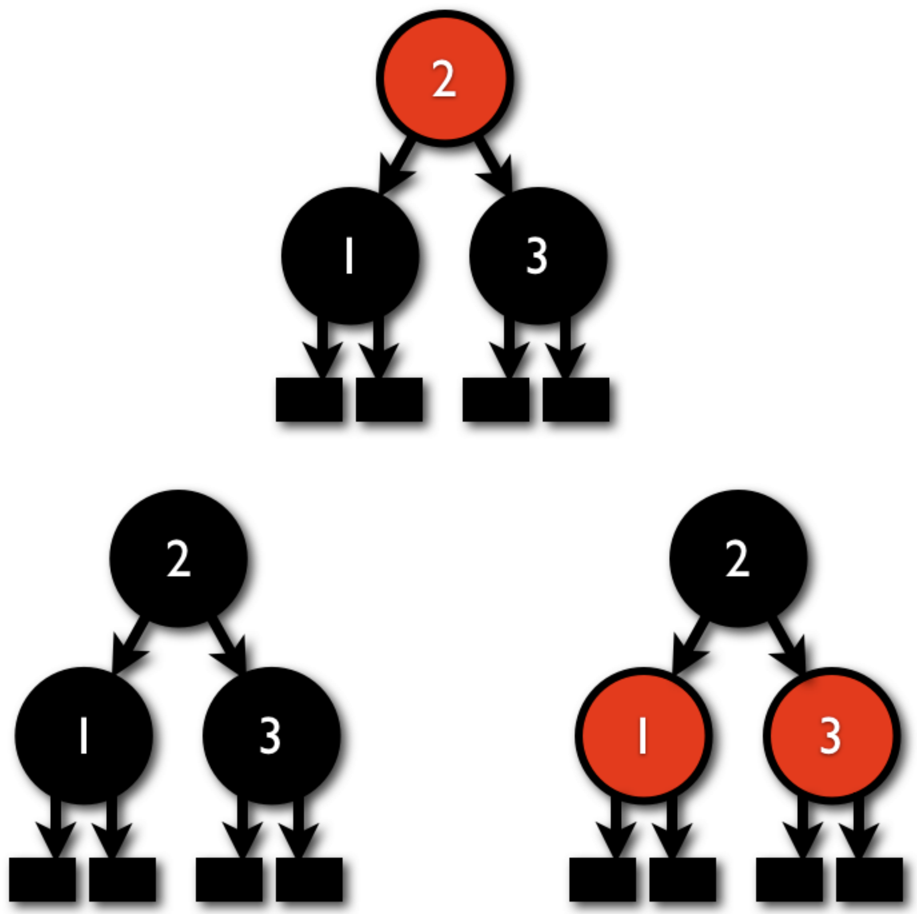
- usual binary search tree invariant
- same # of **black** nodes on every root-to-leaf path
- every **red** node has two **black** children

Guarantee: longest path $\leq 2 \times$ shortest path

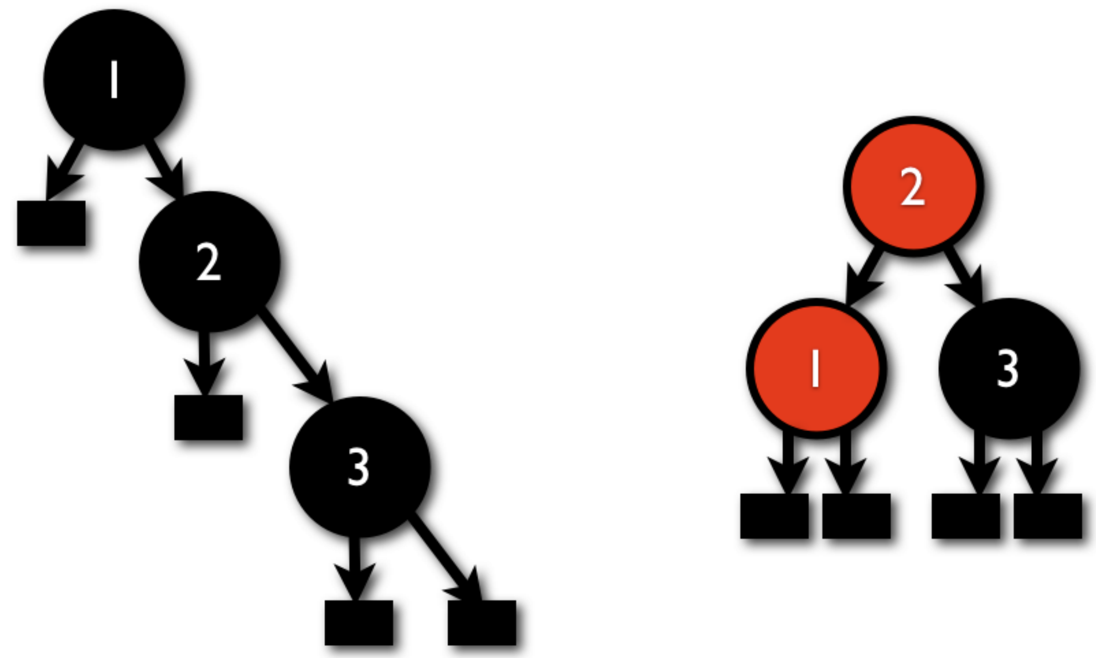
images from: matt.might.net/articles/red-black-delete/

Examples

Valid red-black trees:



Invalid red-black trees:



Insertion

Balance invariants

(1) same # of **black** nodes on every root-to-leaf path
(2) every **red** node has two **black** children

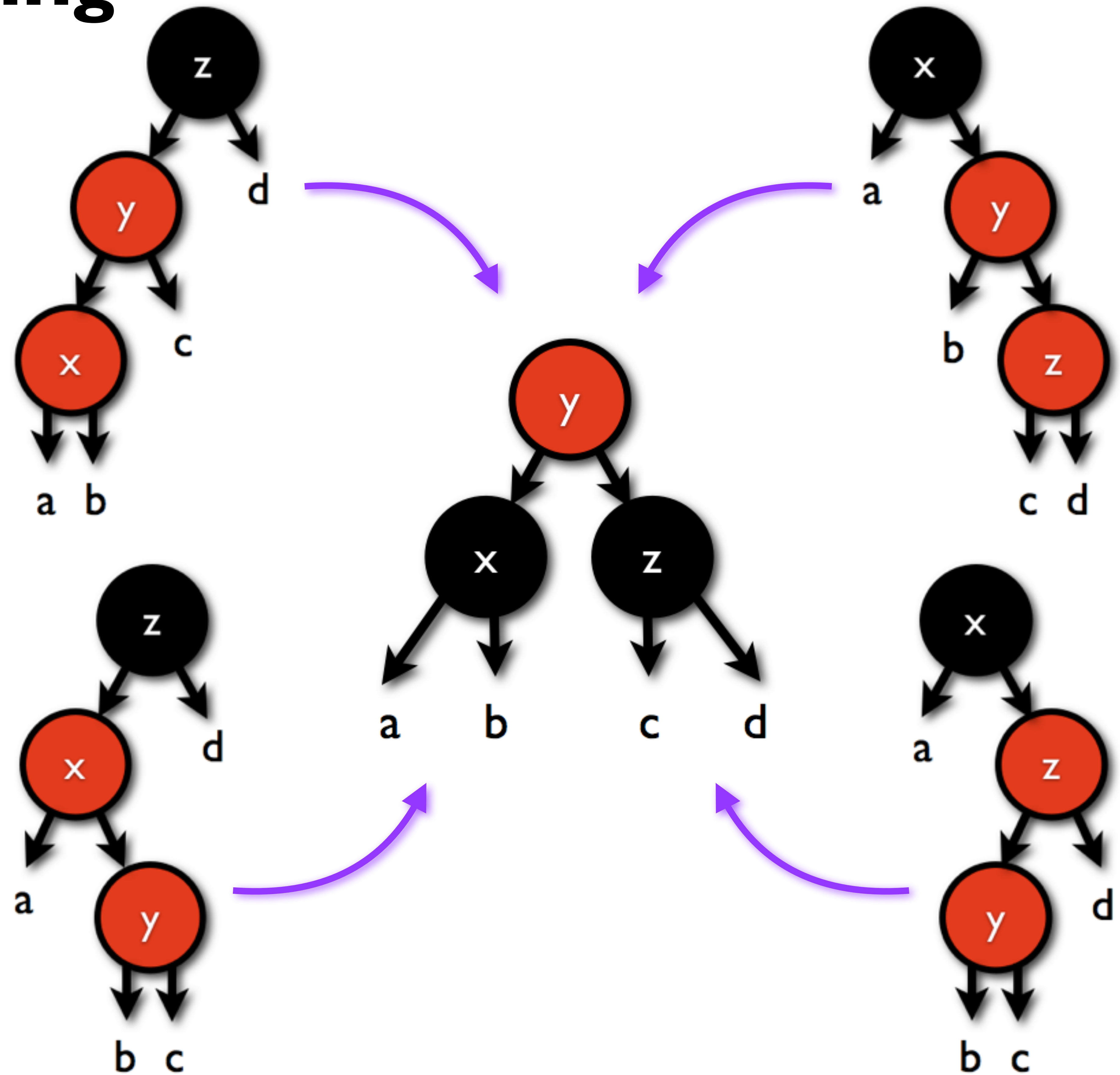
Strategy:

- always insert a **red** node
- if added after a **black** node, we're done!
- else, "rebalance" to eliminate the **red**-**red** violation
  (may cause a new **red**-**red** violation, so recurse up the tree)
- set root to **black**

Rebalancing

After insert, four possible invalid cases:

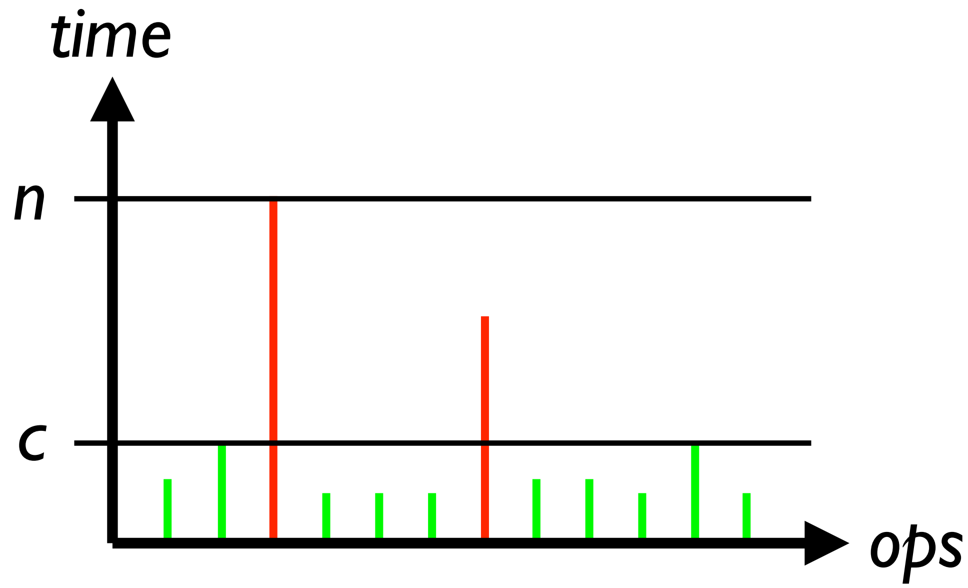
If y's parent is red, must rebalance again!

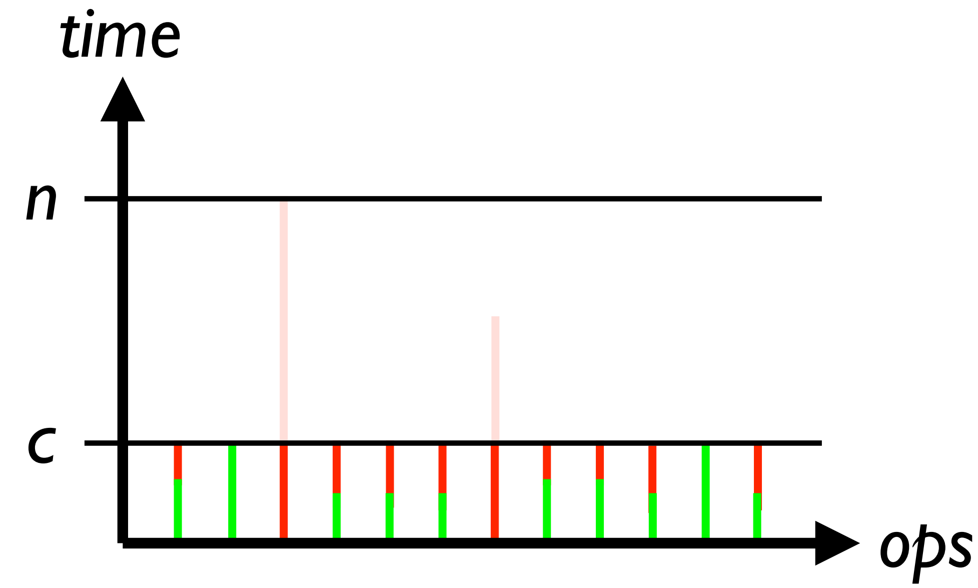


Outline

- Persistence
- Functional vs. imperative data structures
- Example: red-black trees
- **Amortized complexity analysis**
- Amortization for persistent data structures

Amortized vs. worst-case analysis





"worst" worst case:
always assume maximal cost

amortized worst case:
costs can be distributed over ops

n ops × $O(n)$ cost
$\in O(n^2)$ total cost

n ops × $O(1)$ amortized cost
$\in O(n)$ total cost

Tradeoffs of amortized analysis

- more accurate over lifetime of data structure
- opens up new design space *e.g. self-adjusting data structures*
  - can lead to overall faster data structures (in practice, or asymptotically over lifetime)
    e.g. splay trees, union-find
- weaker guarantees about individual operations
  - not suitable for real-time applications

Banker's method

For each operation i, define:

- a_i : amortized cost
- t_i : actual cost

Each operation gets a_i credits

credits are saved-to/spent-from locations in the data structure

Op is …	if …	then …
cheap	$t_i < a_i$	save $a_i - t_i$ credits
neutral	$t_i = a_i$	
expensive	$t_i > a_i$	spend $a_i - t_i$ previously saved credits

To show that a_i is the amortized cost:

Show that we never run out of credits

Banker's analysis of "two-stack" queue

(**L** *and* **R**)

Credits given (a_i):

- **enqueue**: 2 credits
- **dequeue**: 1 credit

Actual cost (t_i):

- **enqueue**: 1 credit *– save 1 credit to* ***R***
- **dequeue**:
  - **|L| > 0**: 1 credit
  - **|L| = 0**: 1 + |**R**| credits *– spend the credits saved on* ***R***

So, both operations have amortized O(1) cost

Outline

- Persistence
- Functional vs. imperative data structures
- Example: red-black trees
- Amortized complexity analysis
- **Amortization for persistent data structures**

Amortization and persistence

Bad news: if data structure is persistent, we can go into debt!

```
q  = foldr enqueue empty [1..5]   – save 5 credits on R
r1 = dequeue q       – spend all credits on R
r2 = dequeue q       – spend all credits on R again!
```

Problem: persistence is working *against* us

Solution: make lazy evaluation work *for* us :-)

Keys: structure data type and functions so that:

- expensive operations are *memoized*
- expensive operations can be "locally" paid for

buy them "on layaway"

(Queue.hs)