Red/Black Trees

Frank Pfenning

15-150, March 26, 2020

Learning Objectives

- Data structure invariants
  - Defining
  - Maintaining
  - Restoring
- Persistent data structures
- Red/black trees

Outline

- Introduce red/black trees
- Walk through some examples
- Live code

Red/Black Trees

- Binary search tree refinement to maintain balance
- Goal: $O(\log n)$ insert and lookup
- Alternatives: AVL trees, splay trees, treaps, ...
- Quite clever, and easy to implement functionally
- Our version is persistent!
  - Insertion does not mutate the tree, but returns a new one
  - From Chris Okasaki, *Red-Black Trees in a Functional Setting*, Journal of Functional Programming 9(4): 471-477 (1999)

Red/Black Tree Invariants

1. Order Invariant: The tree is ordered
2. Height Invariant: The number of black nodes on every path from a leaf to the root is the same (black height)
3. Color Invariant: The parent of every red node is black

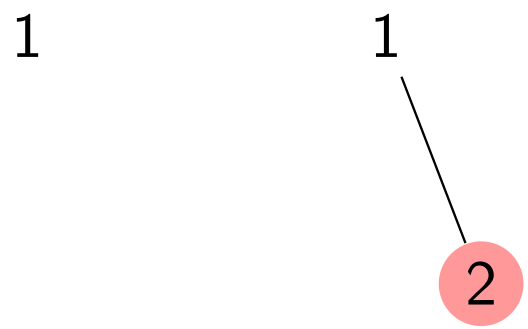
- These are enough to guarantee $O(\log n)$ insert and lookup
- These can be maintained by local operations
- Where not, they can be restored efficiently

Data Abstraction

- We can replace BST with RBT!
- Client will be unable to tell (except their code may be faster)
- While live coding we use integer keys and no data
- Implements sets of integers, with operations
  - Insert an integer into a set (insert)
  - Test membership in the set (lookup)

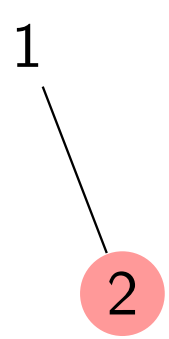
a trivial tree

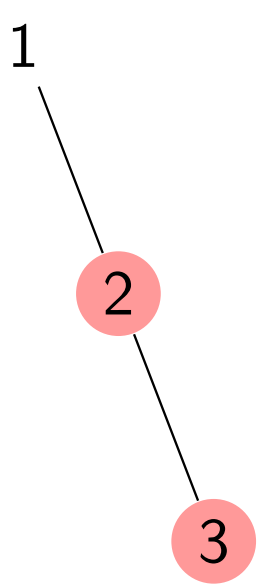
insert 2

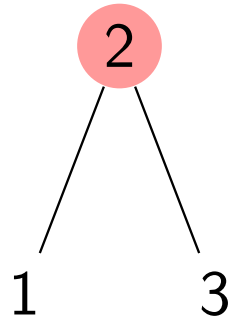


insert 3

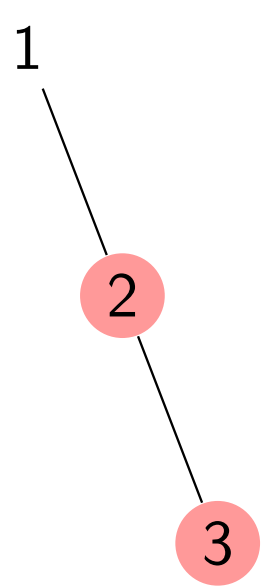
rebalance

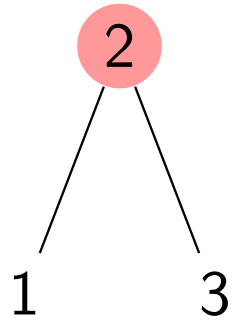


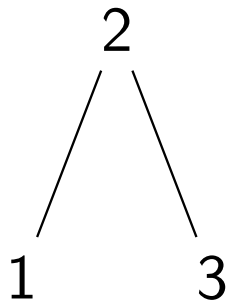




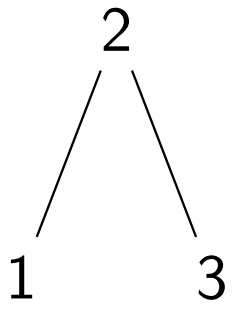
recolor root



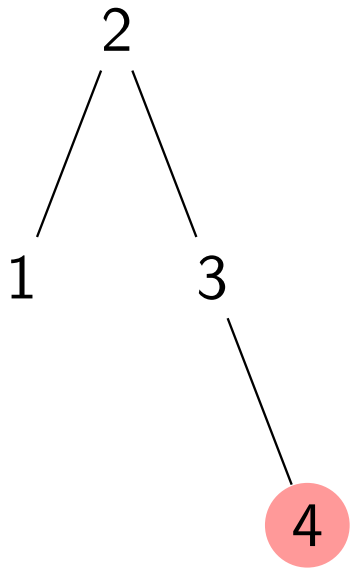


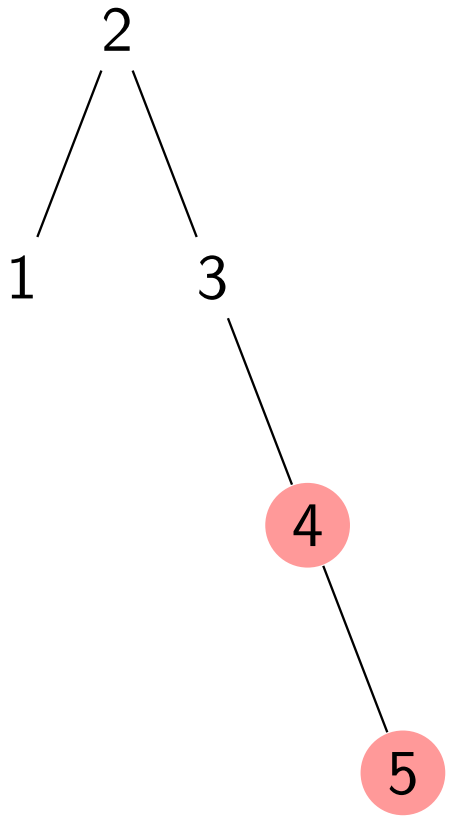


insert 4

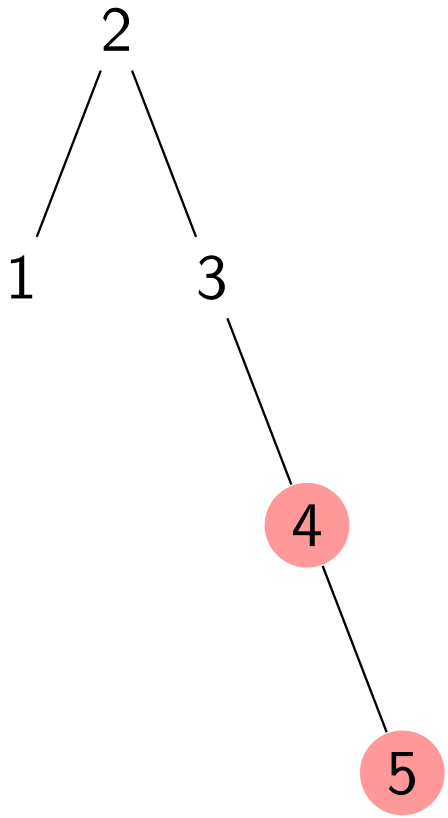


insert 5

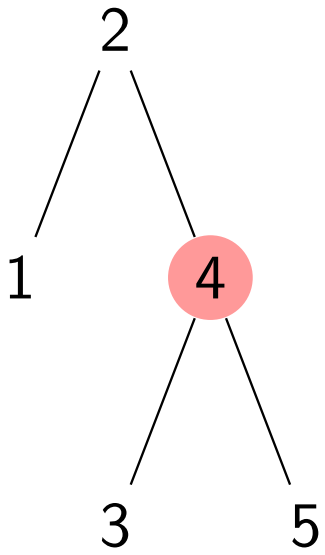


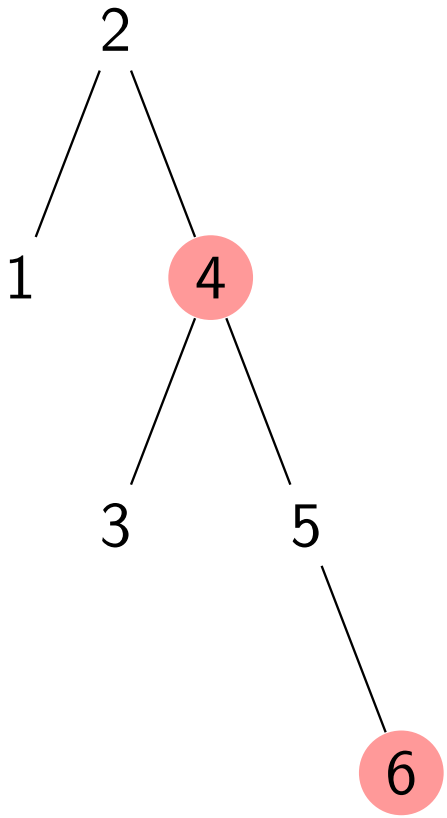


rebalance

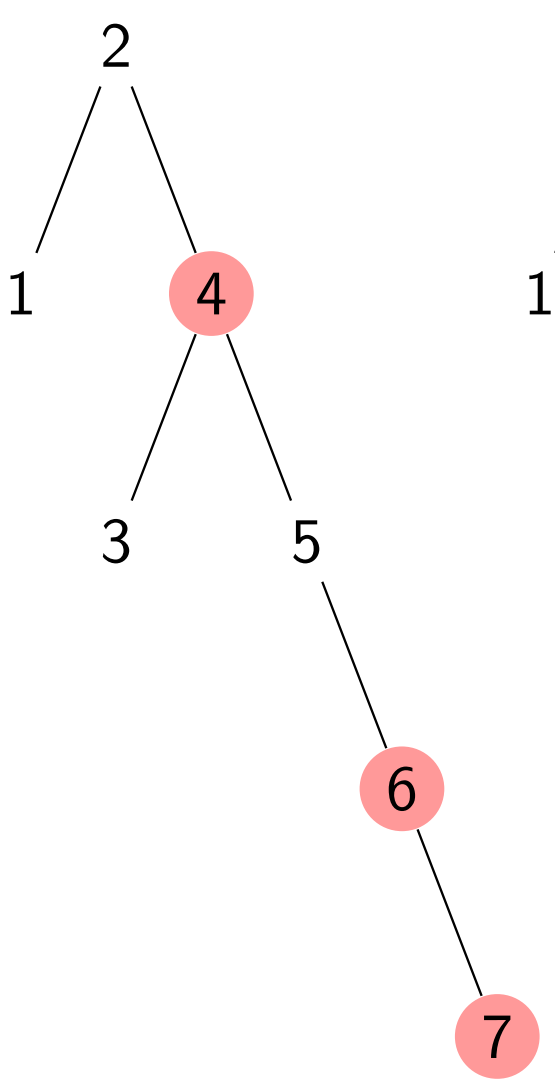


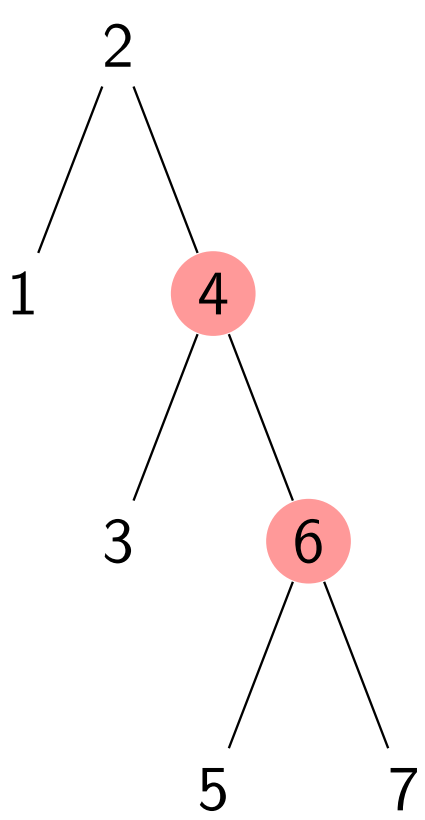
insert 6

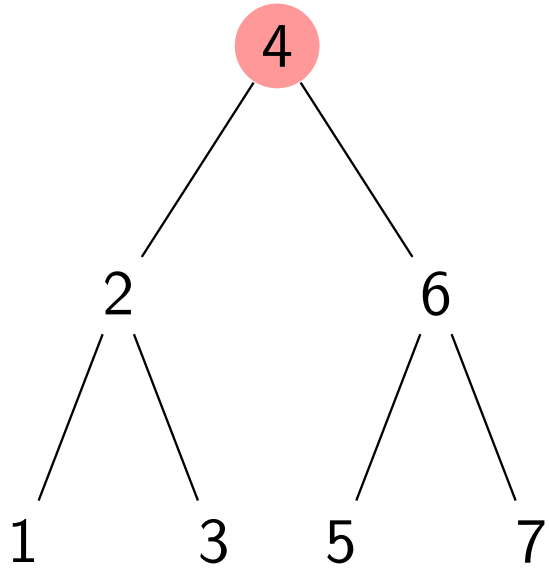




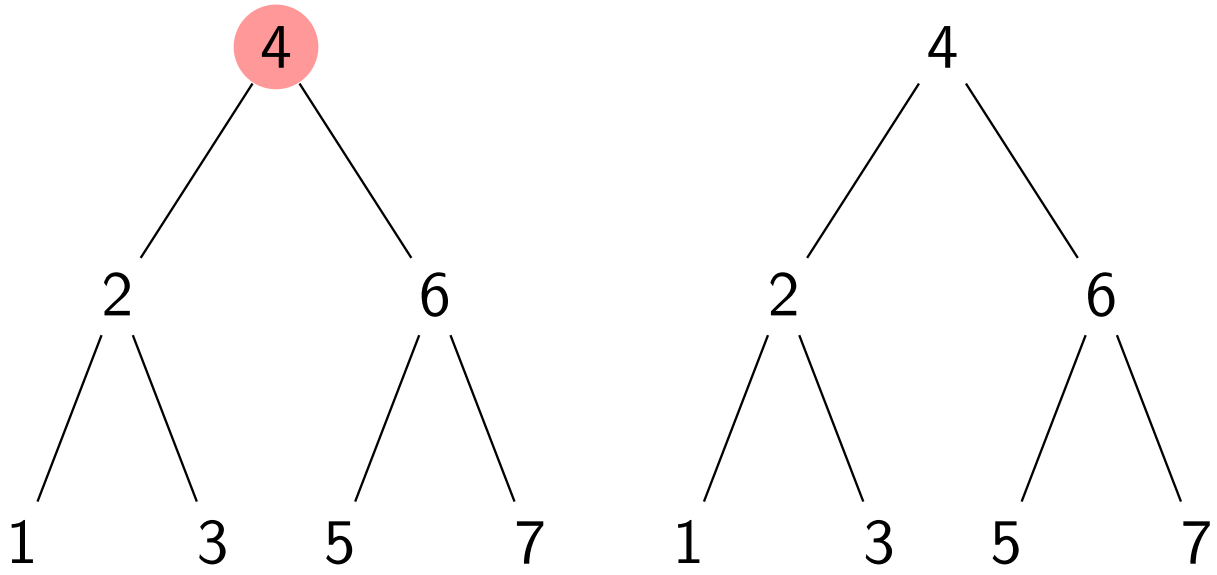
insert 7 rebalance rebalance



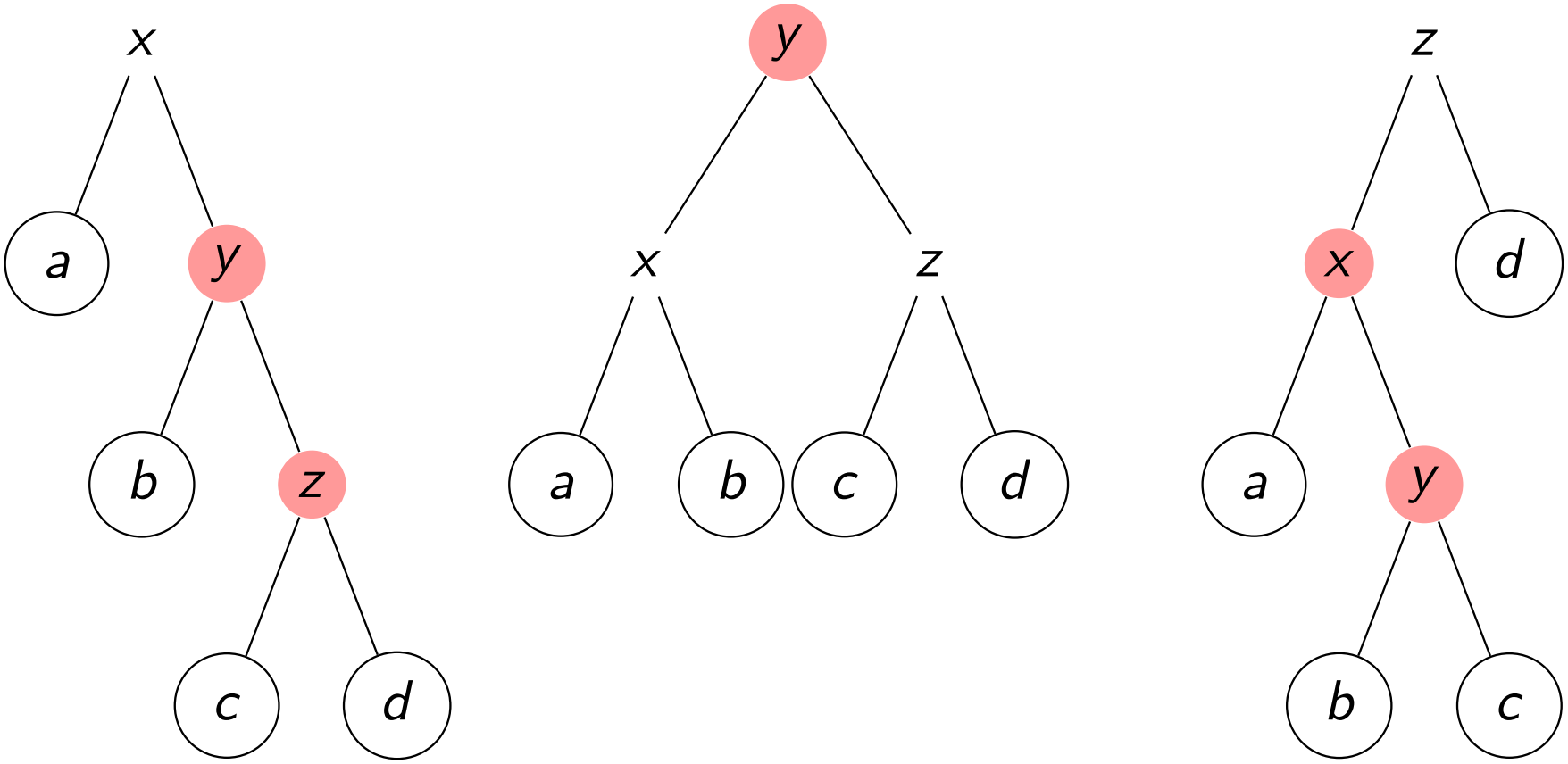




recolor root



Rebalancing (plus 2 more symmetric cases)



Summary

- Data structure invariants
  - Defining
  - Maintaining
  - Restoring
- Persistent data structures
- Red/black trees