Data Structures in Java

Session 10

Instructor: Bert Huang

http://www1.cs.columbia.edu/~bert/courses/3134

Announcements

- Homework 3 due 10/20

Review

- AVL Trees
  - Single rotate for left-left imbalance
  - Double rotate for left-right imbalance
- Running time Analysis
  - Depth always O(log N)
  - Constant cost for rotations

Today's Plan

- Amortized Running time
  - Splay Trees
- Tries

Amortized Running Time

- So far, we measure the worst-case running time of each operation
- Usually we perform many operations
- Sometimes M $O(f(N))$ operations can run **provably** faster than $O(M\ f(N))$
- Then we can guarantee better average running time, aka **amortized**

Comparing Models

- Amortized and Average case average running time of many operations
- Amortized and Standard: adversary chooses input values and operations
- Average analysis, analyst chooses randomization scheme

Splay Trees

- Like AVL trees, use the standard binary search tree property
- After any operation on a node, make that node the new root of the tree
  - Make the node the root by repeating one of two moves that make the tree more spread out

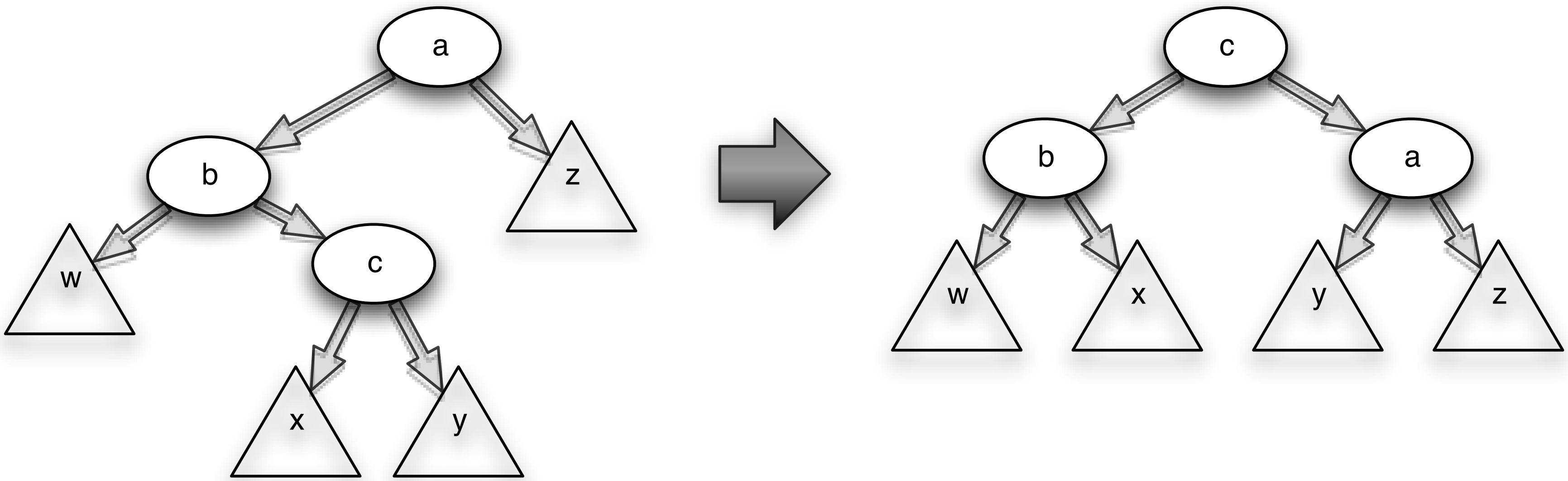
Informal Justification

- Similar to *caching*.
  - Heuristically, data that is accessed tends to be accessed often.
- Easier to implement than AVL trees
  - No height bookkeeping

Easy cases

- If node is root, do nothing
- If node is child of root, do single AVL rotation
- Otherwise, node has a grandparent, and there are two cases

Case 1: zig-zag

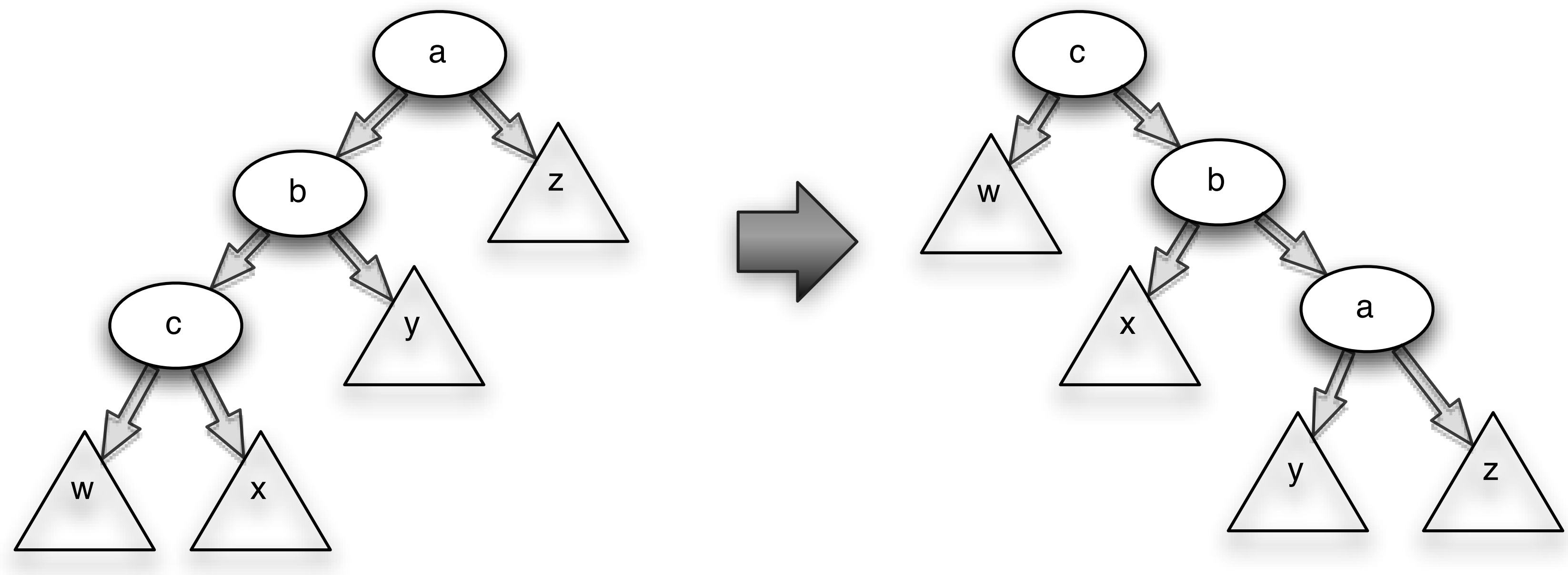


- Use when the node is the right child of a left child (or left-right)
- Double rotate, just like AVL tree

Case 2: zig-zig

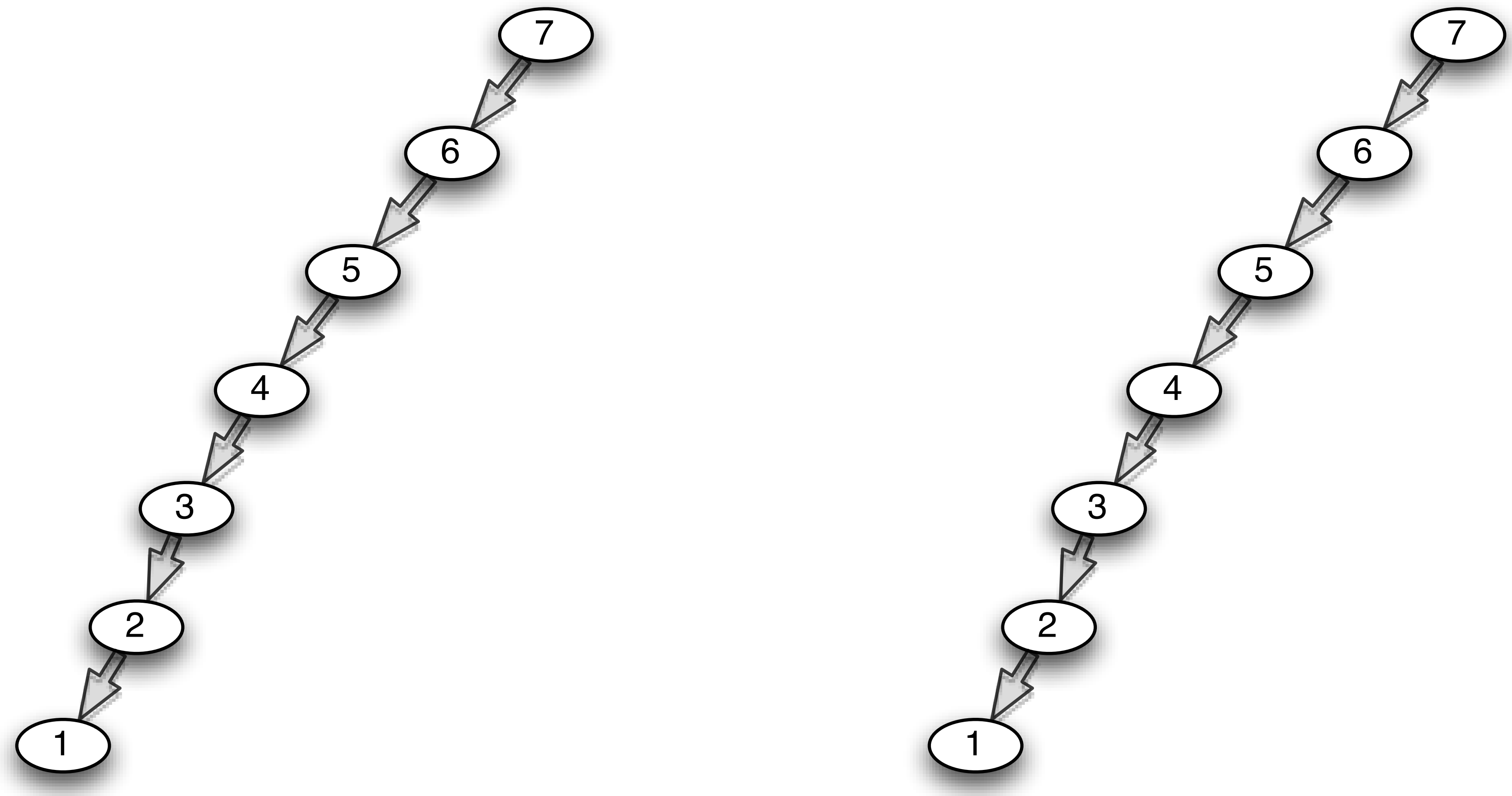
- We can’t use the single-rotation strategy like AVL trees
- Instead we use a different process, and we’ll compare this to single-rotation

Case 2: zig-zig



- Use when node is the right-right child (or left-left)
- Reverse the order of grandparent->parent->node
  - Make it node->parent->grandparent

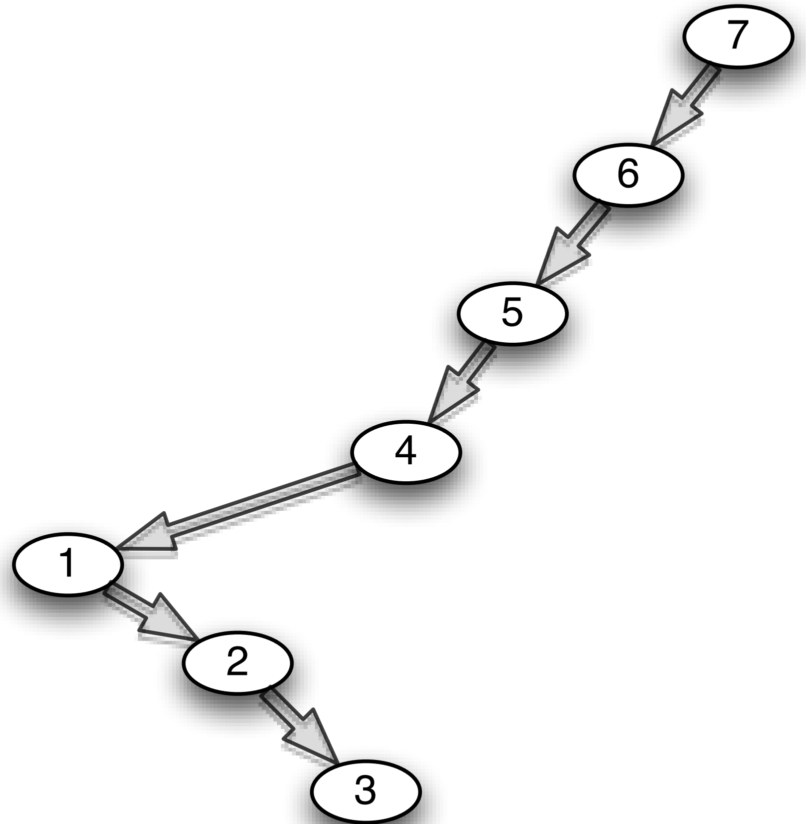
Case 2 versus Single Rotations 1



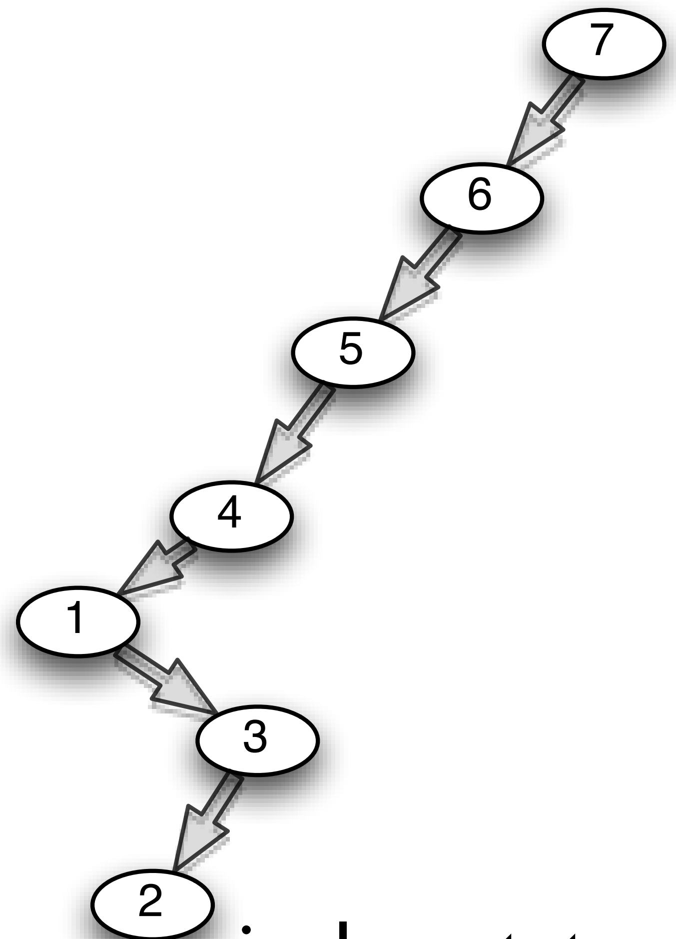
zig-zig

single-rotate

Case 2 versus Single Rotations 2

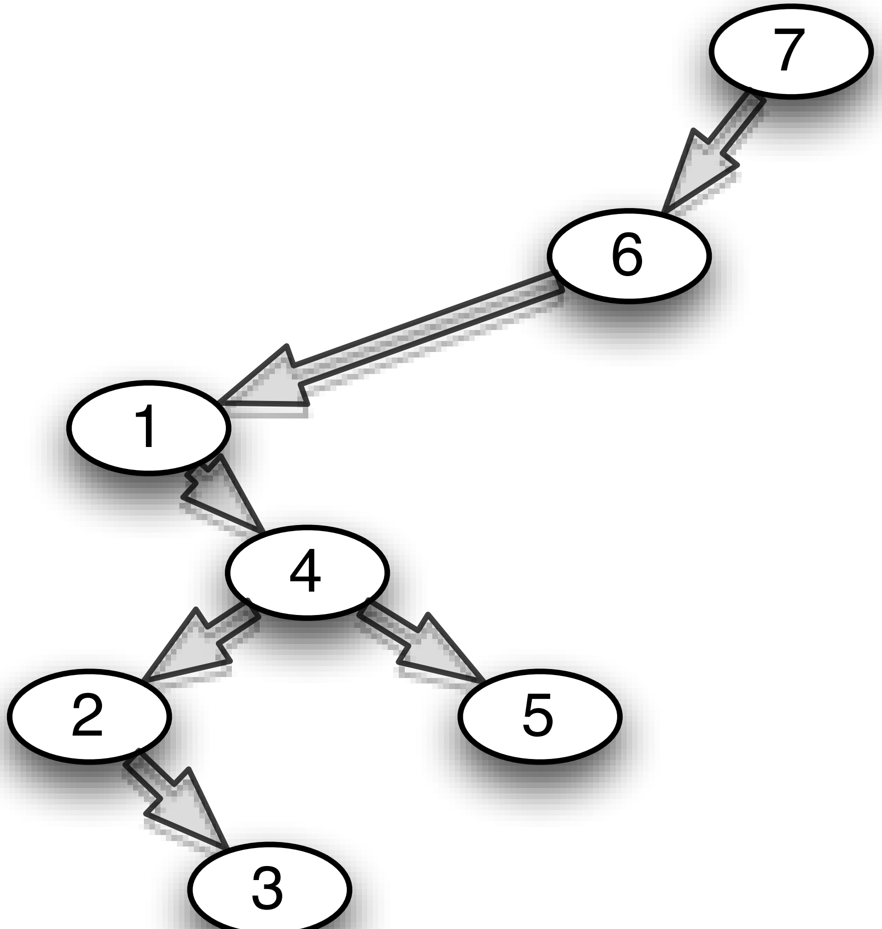


zig-zig

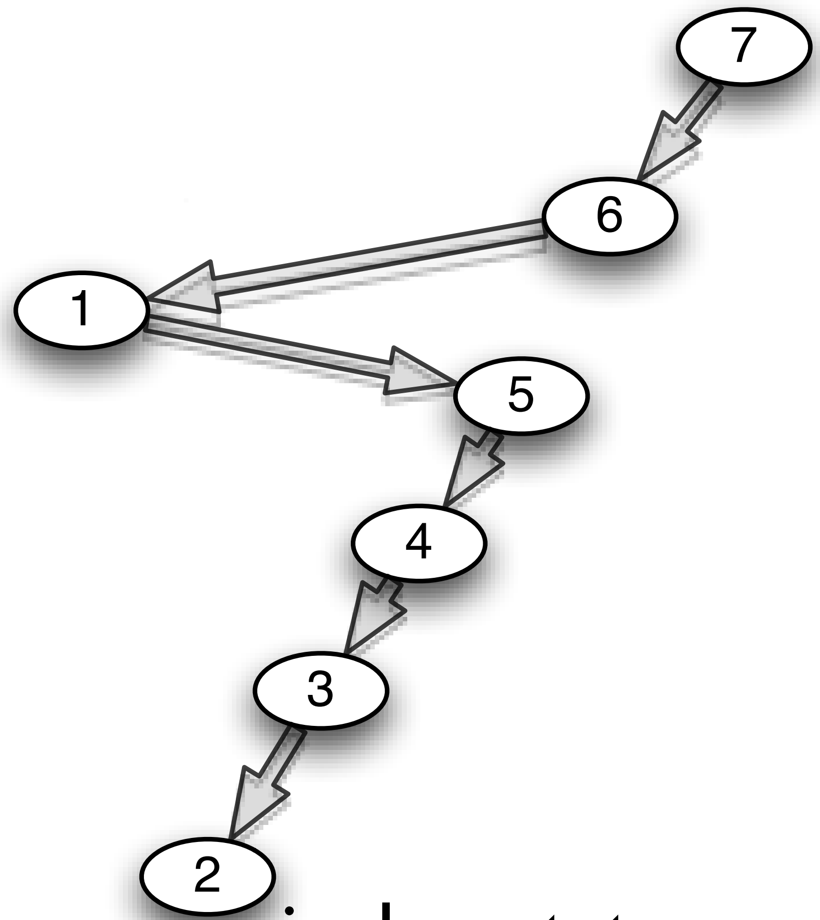


single-rotate

Case 2 versus Single Rotations 3

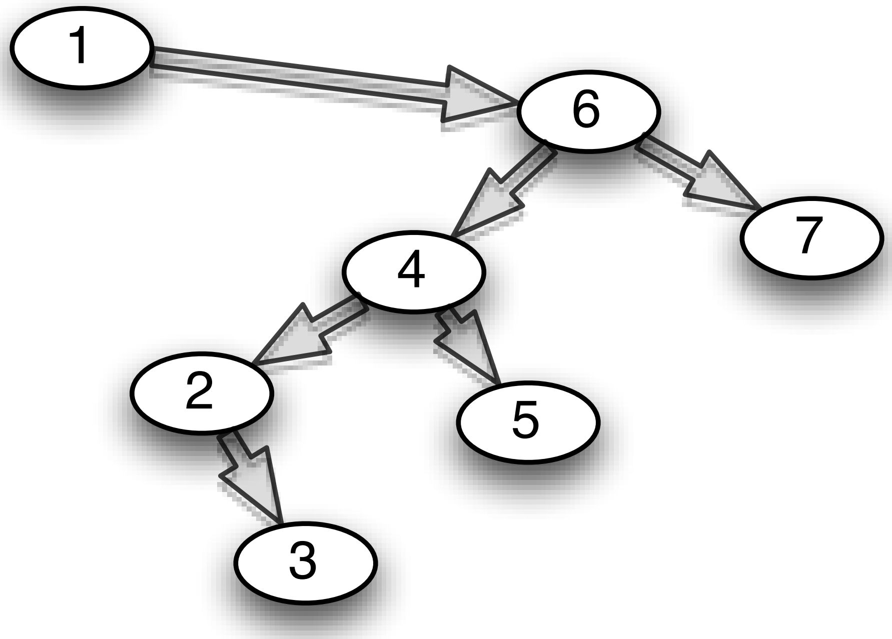


zig-zig

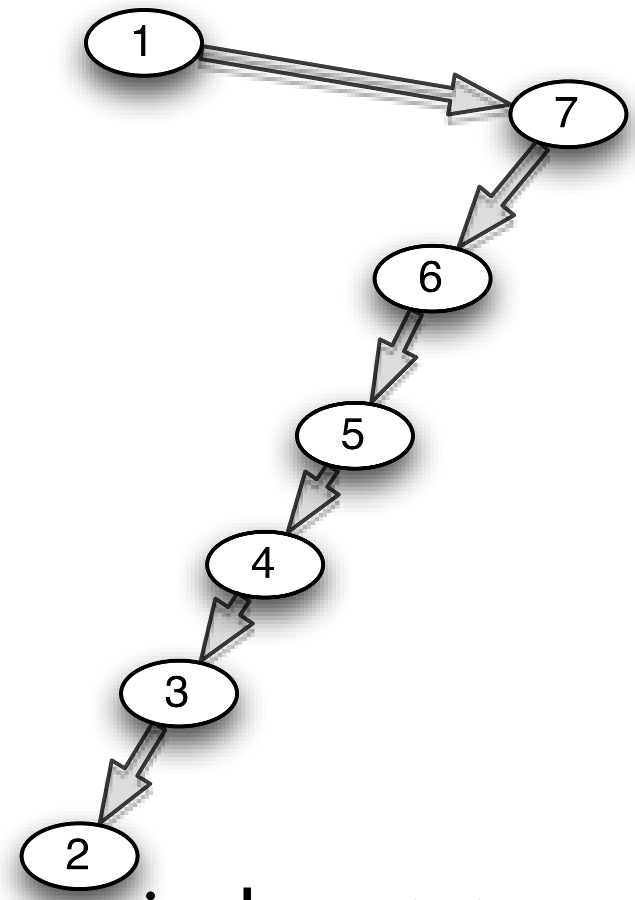


single-rotate

Case 2 versus Single Rotations 4



zig-zig



single-rotate

Splay Analysis (Informal)

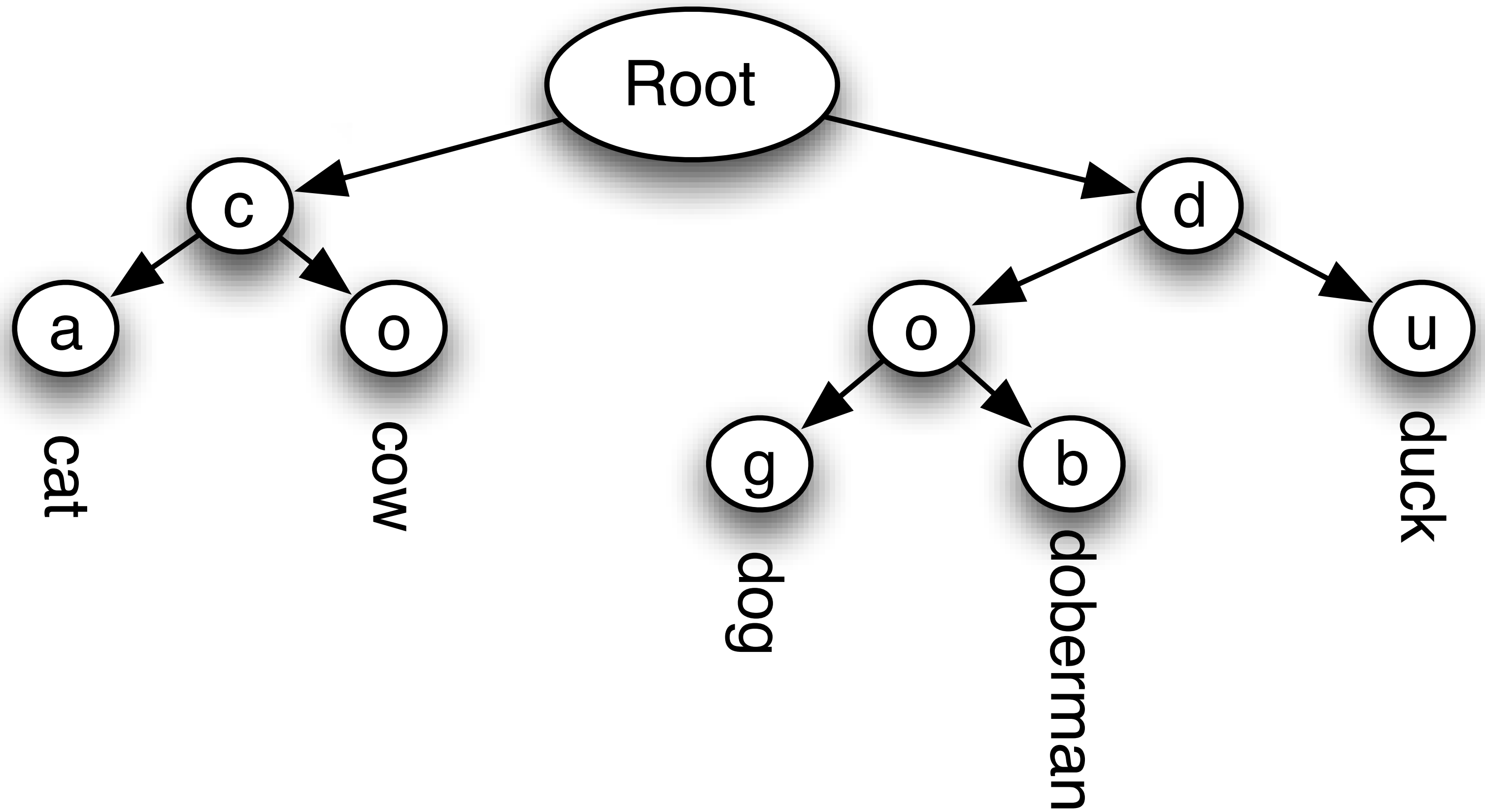
- We can make a chain by inserting nodes that make the tree its left child
  - Each of these operations is cheap
- Then we can search for deepest node
  - Splay operation squishes the tree; can only bad operations once before they become cheap
- M operations take $O(M \log N)$, so amortized $O(\log N)$ per operation (fyi, not proved)

Prefix Trees (Tries)

- Nicknamed “Trie”, short for re**trie**val
- Efficiently store objects for fast retrieval via keys
  - Usually key is a String
- Basic strategy:
  - split into sub-tries based on current letter

Trie Example

- "cat", "cow", "dog", "doberman", "duck"



Trie Analysis

- In the worst case, inserting a key of length **k** or (looking up) is **O(k)**
- This is not dependent on **N** (this is shocking!)
- Much better than **log(N)** for huge data like dictionaries

Reading

- Splay Trees: 4.5