Data Structures in Java

Session 24

Instructor: Bert Huang

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

Announcements

- Homework 6 due Dec. 10, last day of class
- Final exam Thursday, Dec. 17th, 4-7 PM, Hamilton 602 (this room)
  - same format as midterm (open book/notes)

Review

- Note about hw4: rehashing order
- Finish discussion of complexity
  - Polynomial Time Approximation Schemes
  - Graph Isomorphism
- k-d trees

Today's Plan

- A couple topics on data structures in Artificial Intelligence:
  - Game trees
  - Graphical Models
- Final Review (part 1)

Artificial Intelligence

- Sub-field of Computer Science concerned with algorithms that behave *intelligently*
  - or if we're truly ambitious, **optimally.**
- An AI program is commonly called an **agent**
  - which makes decisions based on its **percepts**

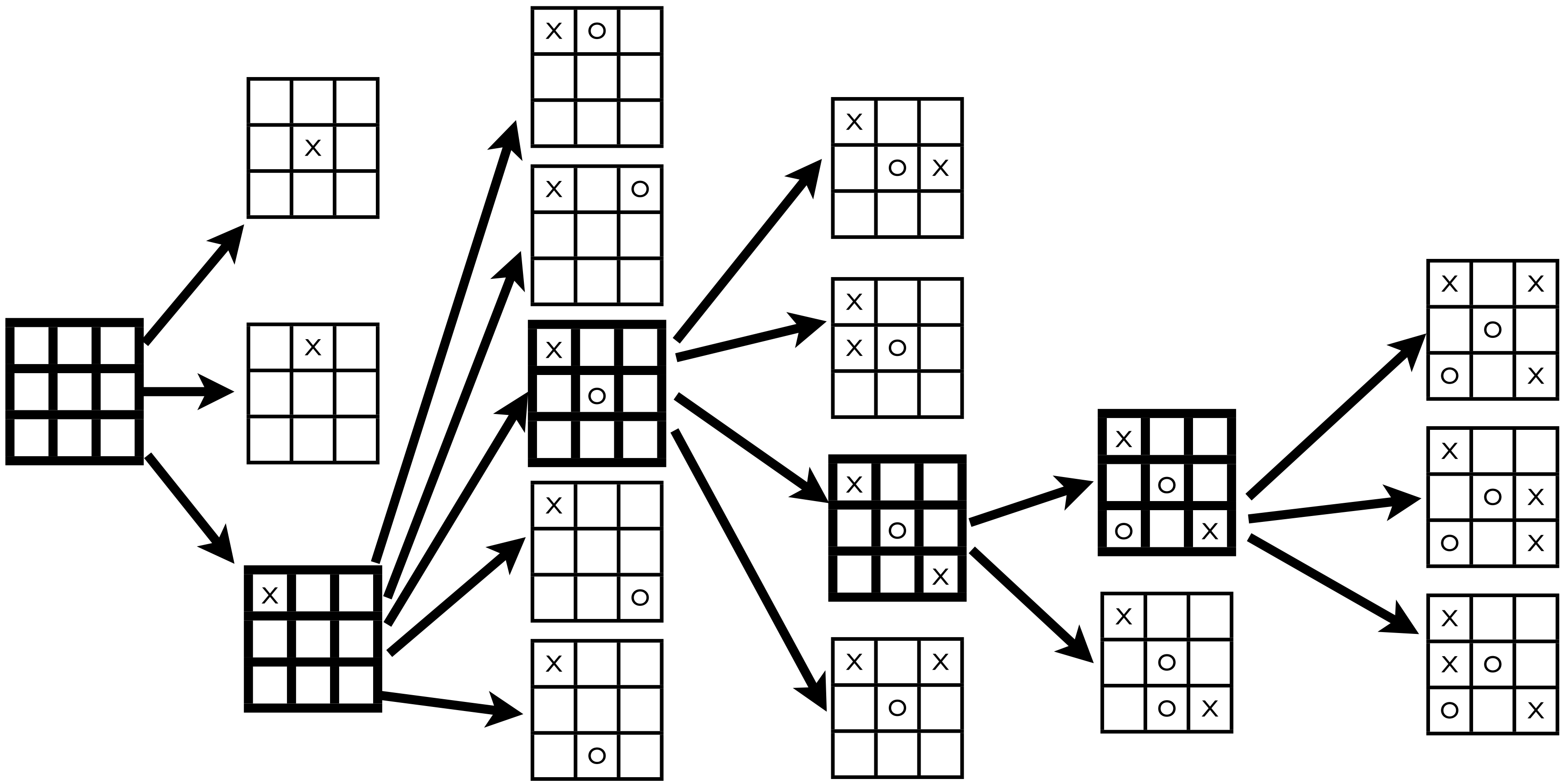
A.I. in Games

- AI still needs to simplify the environment for its agents, so games are a nice starting point
- Many board games are turn-based, so we can take some time to compute a good decision at each turn
- Deterministic turn-based games can be represented as **game trees**

Game Trees

- The root node is the starting state of the game
- Children correspond to possible moves
- If 2-player, every other level is the computer's turn
- The other levels are the adversary's turns
- In a simple game, we can consider/store the whole tree, make decisions based on the subtrees

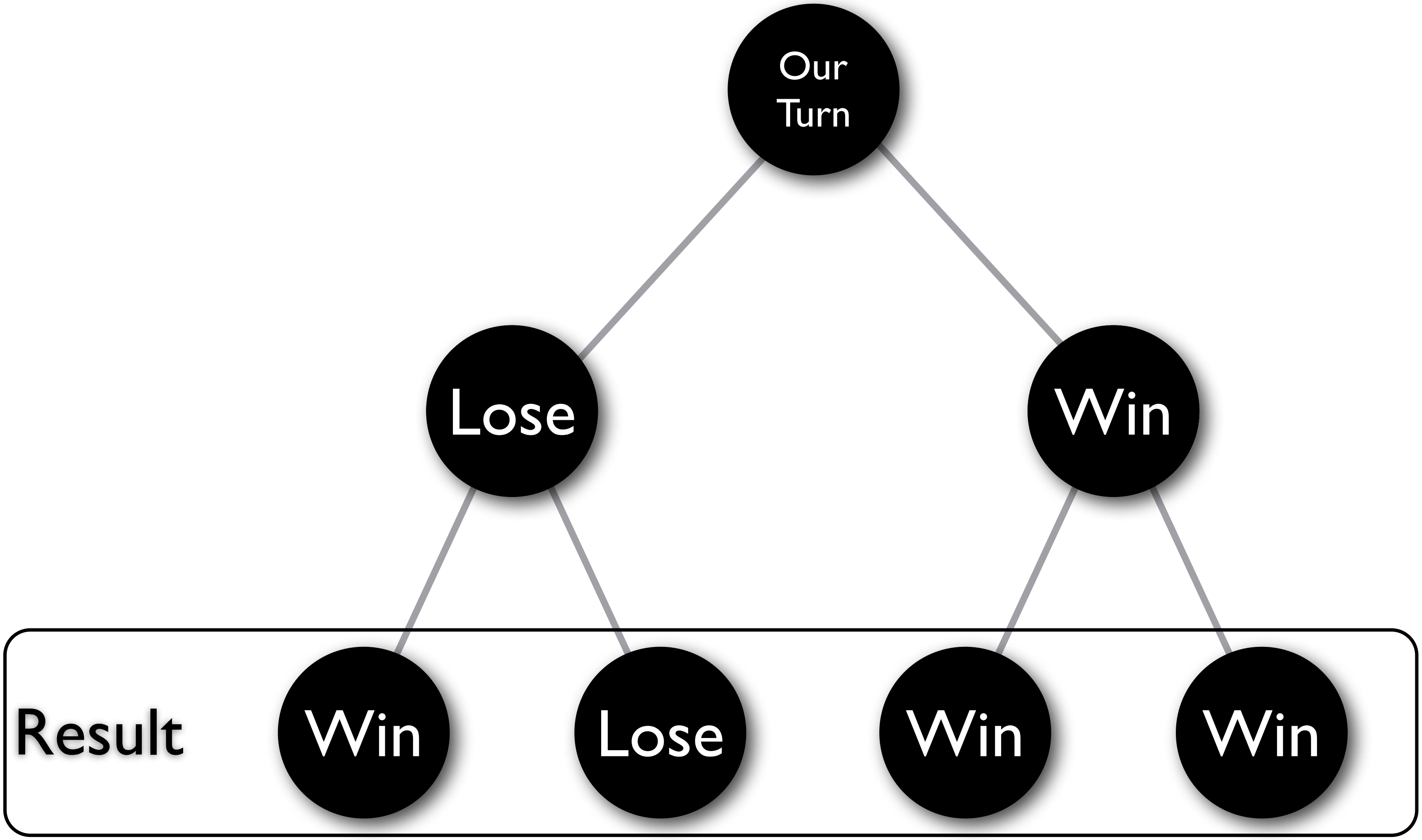
Partial Tic-Tac-Toe Game Tree



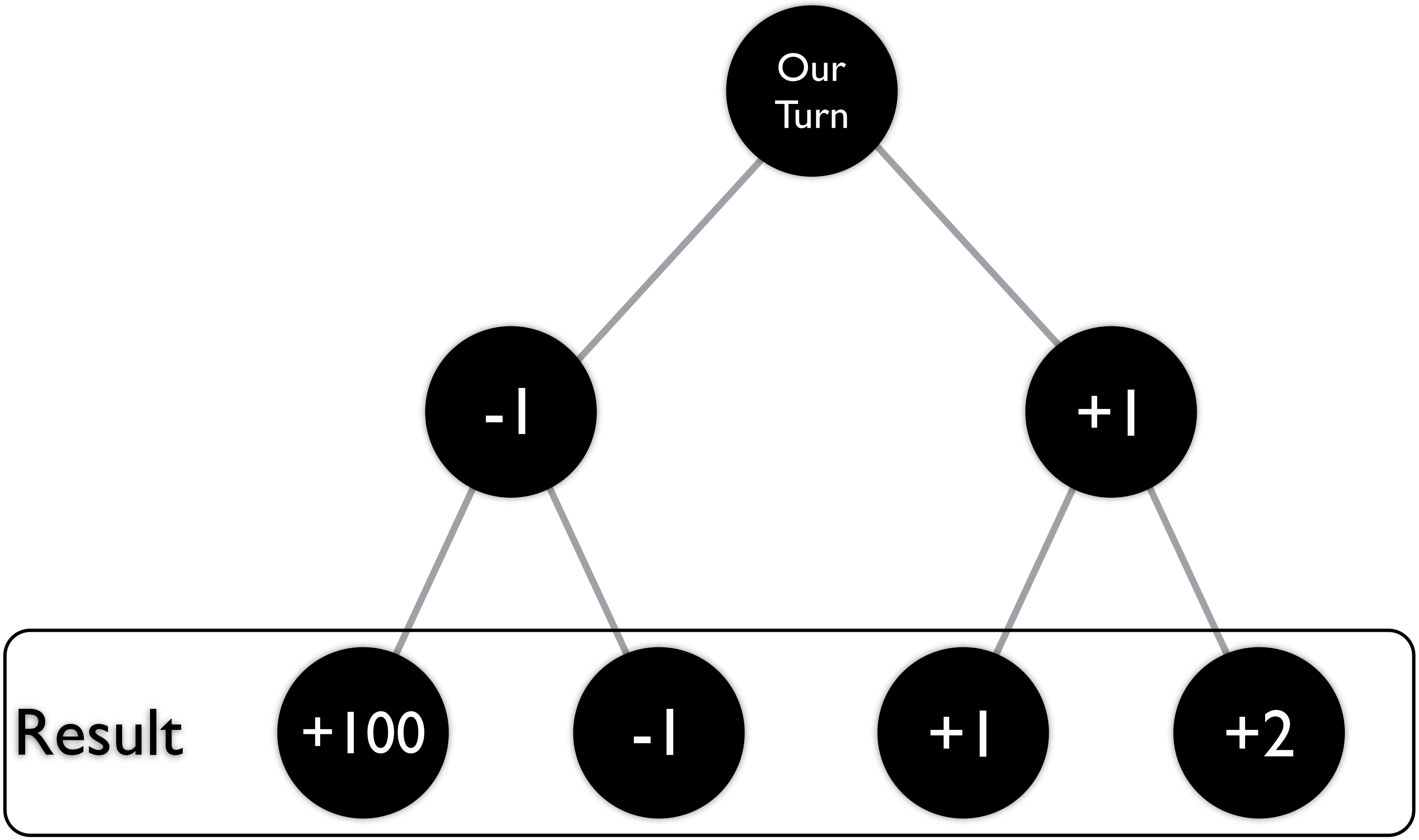
Tree Strategy

- Thinking about the game as a tree helps organize computational strategy
- If adversary plays optimally, we can define the optimal strategy via the **minimax** algorithm
- Assume the adversary will play the optimal move at the next level. Use that result to decide which move is optimal at current level.

Simple Tree



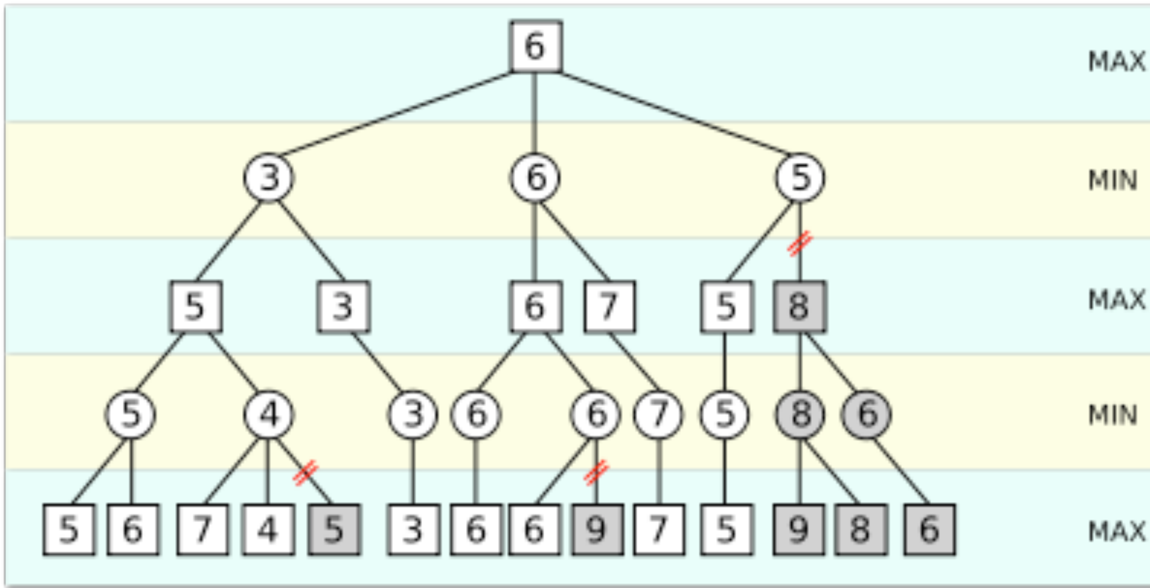
Numerical Rewards



Minimax Details

- Depth first search (postorder) to find leaves; propagate information up
- Adversary also assume you will play optimally
- Impossible to store full tree for most games, use heuristic measures
  - e.g., Chess piece values, # controlled squares
- Cut off after a certain level

Pruning

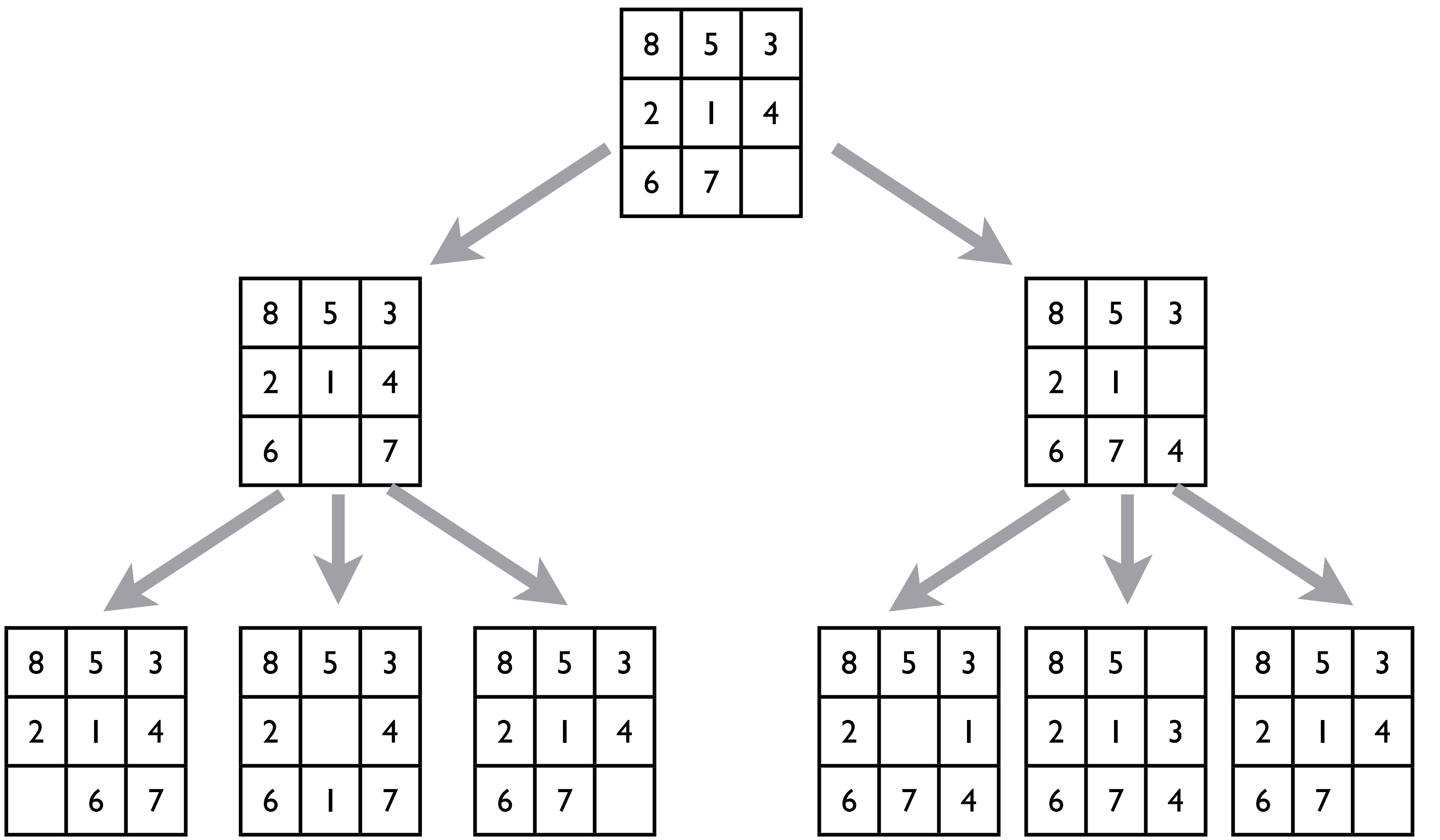


- We can also ignore parts of the tree if we see a subtree that can't possibly be better than one we saw earlier
- This is called **alpha-beta** pruning
- Figure from wikipedia article on alpha-beta pruning

Search

- Some puzzles can be thought of as trees too
- 15-puzzle, Rubik's Cube, Sudoku
- Discrete moves move from current state to children states
- A.I. wants to find the solution state efficiently

8-puzzle


8 5 3
2 1 4
6 7
8 5 3
2 1 4
6 7
8 5 3
2 1
6 7 4
8 5 3
2 1 4
6 7
8 5 3
2 4
6 1 7
8 5 3
2 1 4
6 7
8 5 3
2 1
6 7 4
8 5
2 1 3
6 7 4
8 5 3
2 1 4
6 7

Simple Idea

- Breadth first search; level-order
  - Try every move from current state
  - Try 2 moves from current state
  - Try 3 moves from current state
  - ...

Another Idea

- Depth first search
  - Try a move
  - Try another move...
  - If we get stuck, backtrack

Heuristic Search

- The main problem is without any knowledge, we are guessing arbitrarily
- Instead, design a heuristic and choose the next state to try according to heuristic
  - e.g., # of tiles in the correct location, distance from maze goal

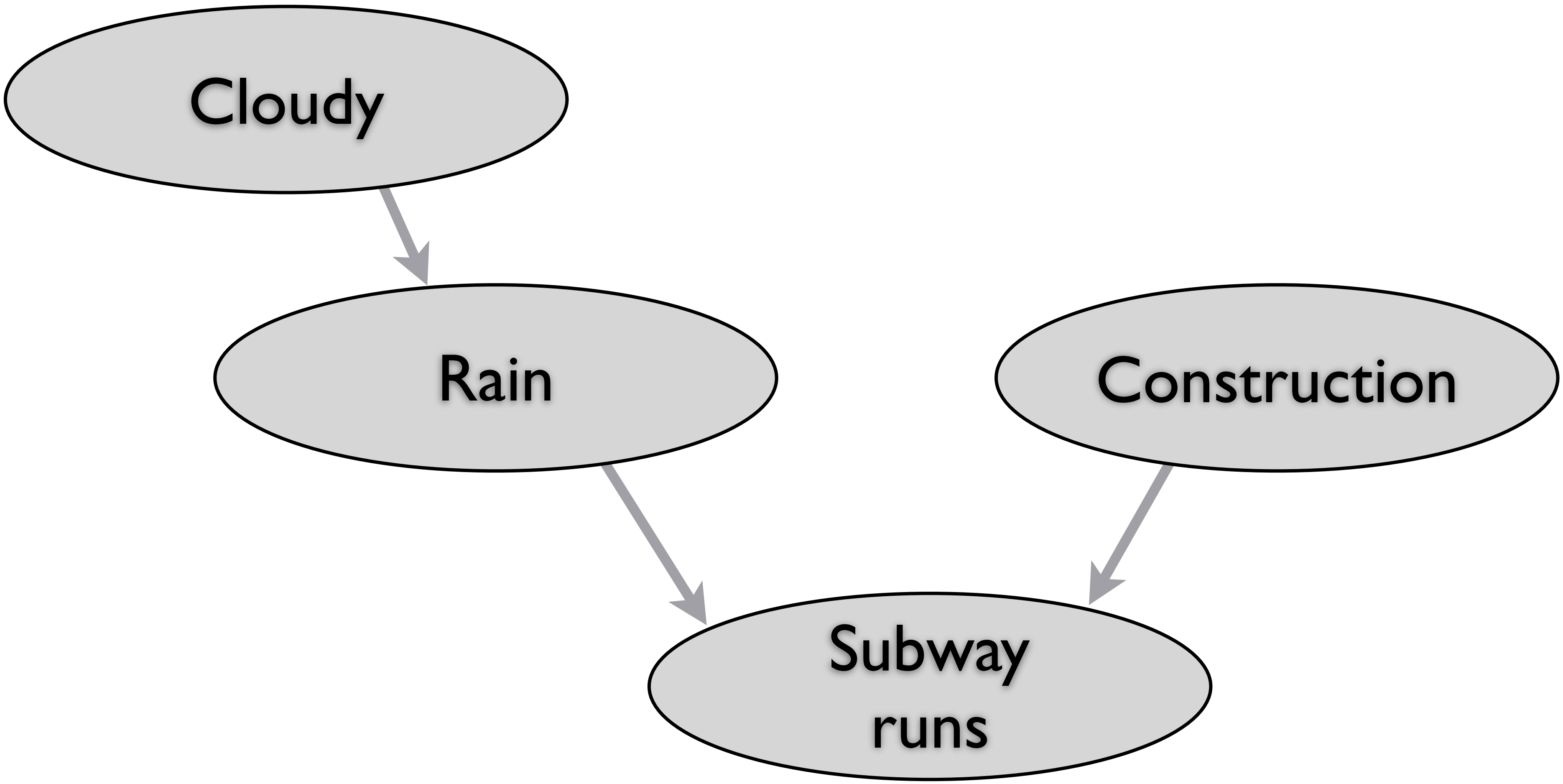
Probabilistic Inference

- Some of these decisions are too hard to compute exactly, and often there is insufficient information to make an exact decision
- Instead, model uncertainty via probability
- An important application for graph theory is using graphs to represent **probabilistic independence**

Independent Coins

- 1. Suppose I flip coin twice, what is the probability of both flips landing heads?
- 2. Compare to if we flip a coin, and if it lands heads, we flip a second coin. What is the probability of two heads?
- In Scenario 1, we can reason with less computation by taking advantage of independence

A Simple Bayesian Network



Inference Rules of Thumb

- Trees and DAGs are easier to reason
  - We can use similar strategy to Topological sort:
  - Only compute probability once all incoming neighbors have been computed
- Cyclic graphs are difficult; NP-hard in some settings

About the Final

- Theory only (no programming)
- Bring your book and notes
- No electronic devices
- Covers both halves of the semester, mostly 2nd half.

Course Topics

- Lists, Stacks, Queues
- General Trees
- Binary Search Trees
  - AVL Trees
  - Splay Trees
- Tries
- Priority Queues (heaps)
- Hash Tables
- Graphs
  - Topological Sort, Shortest Paths, Spanning Tree
- Disjoint Sets
- Sorting Algorithms
- Complexity Classes
- kd-Trees

Definitions

- For N greater than some constant, we have the following definitions:

$$T(N) = O(f(N)) \leftarrow T(N) \leq cf(N)$$

$$T(N) = \Omega(g(N)) \leftarrow T(N) \geq cg(N)$$

$$T(N) = \Theta(h(N)) \leftarrow \begin{array}{c} T(N) = O(h(N)) \\ T(N) = \Omega(h(N)) \end{array}$$

- There exists some constant c such that cf(N) bounds T(N)

Abstract Data Type: Lists

- An ordered series of objects
- Each object has a previous and next
  - Except **first** has no prev., **last** has no next
- We can insert an object (at location *k*)
- We can remove an object (at location *k*)
- We can read an object from (location *k*)

List Methods

- Insert object (at index)
- Delete by index
- Get by index

Stack Definition

- Essentially a restricted List
- Two (main) operations:
  - Push(AnyType x)
  - Pop()
- Analogy – Cafeteria Trays, PEZ

Stack Implementations

- Linked List:
  - Push(x) <-> add(x) <-> add(x,0)
  - Pop() <-> remove(0)
- Array:
  - Push(x) <-> Array[k] = x; k = k+1;
  - Pop() <-> k = k-1; return Array[k]

Queue ADT

- Stacks are **L**ast **I**n **F**irst **O**ut
- Queues are **F**irst **I**n **F**irst **O**ut, first-come first-served
- Operations: **enqueue** and **dequeue**
- Analogy: standing in line, garden hose, etc

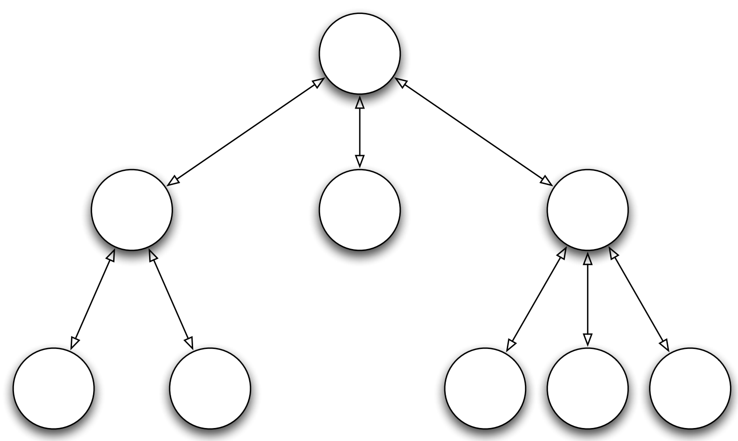
Queue Implementation

- Linked List
  - add(x,0) to enqueue, remove(N-1) to dequeue
- Array List won’t work well!
  - add(x,0) is expensive
  - Solution: use a circular array

Circular Array

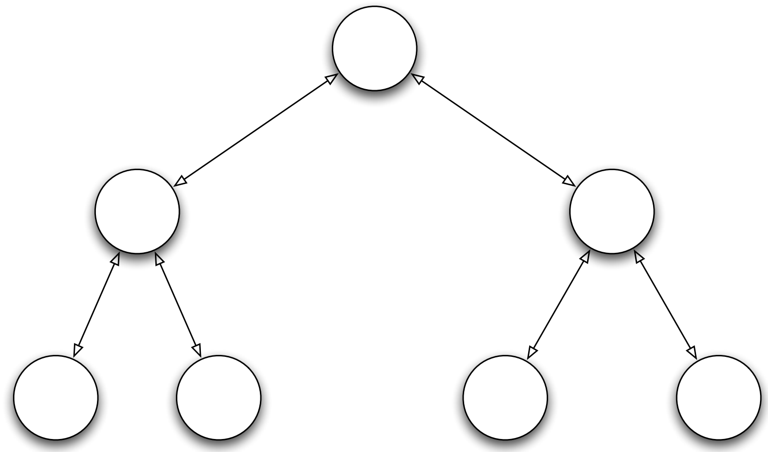
- Don’t shift after removing from array list
- Keep track of start and end of queue
- When run out of space, wrap around; modular arithmetic
- When array is full, increase size using list tactic

Tree Implementation

- Many possible implementations
- One approach: each node stores a list of children
- ```
  public class TreeNode<T> {
      T Data;
      Collection<TreeNode<T>> myChildren;
  }
  ```
  ```

Tree Traversals

- Suppose we want to print all nodes in a tree
- What order should we visit the nodes?
  - **Preorder** - read the parent before its children
  - **Postorder** - read the parent after its children

Preorder vs. Postorder

- ```
  // parent before children
  preorder(node x)
      print(x)
      for child : myChildren
          preorder(child)
  ```
- ```
  // parent after children
  postorder(node x)
      for child : myChildren
          postorder(child)
      print(x)
  ```



Binary Trees

- Nodes can only have two children:
  - left child and right child
- Simplifies implementation and logic
- ```
  public class BinaryNode<T> {
      T element;
      BinaryNode<T> left;
      BinaryNode<T> right;
  }
  ```
- Provides new **inorder** traversal
  ```

Inorder Traversal

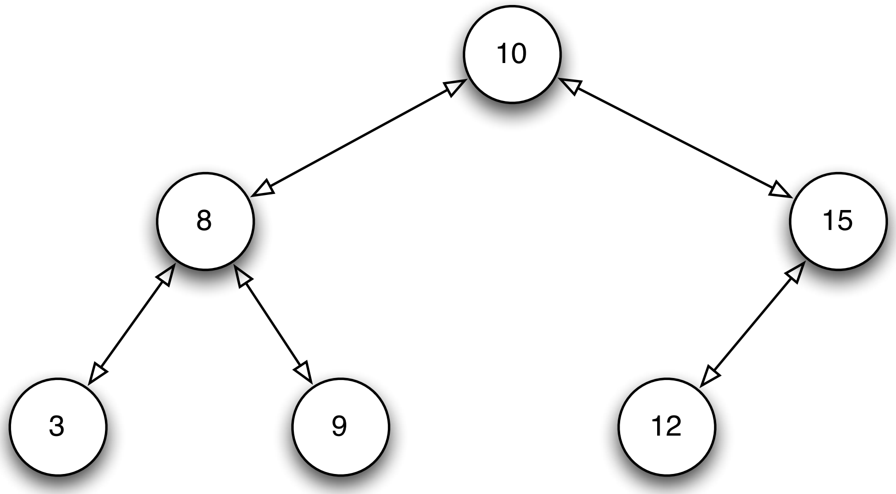
- Read left child, then parent, then right child
- Essentially scans *whole* tree from left to right
- inorder(node x)
  inorder(x.left)
  print(x)
  inorder(x.right)



Search (Tree) ADT

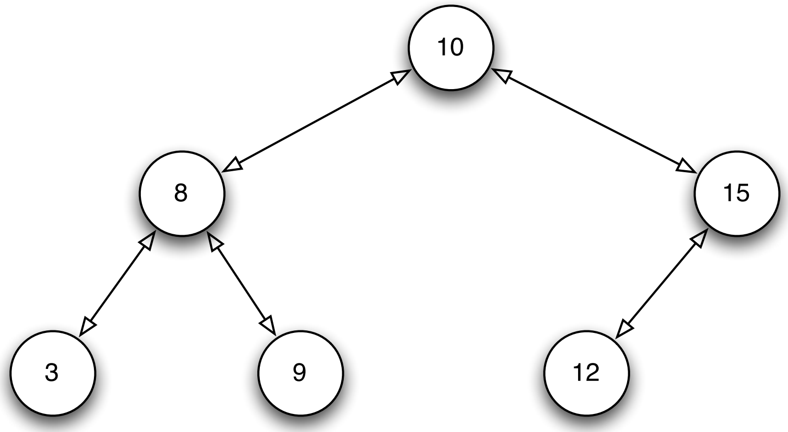
- ADT that allows insertion, removal, and searching by **key**
  - A **key** is a value that can be compared
  - In Java, we use the **Comparable** interface
  - Comparison must obey transitive property
- Search ADT doesn't use any index

Binary Search Tree



- Binary Search Tree Property:
  Keys in left subtree are less than root.
  Keys in right subtree are greater than root.
- BST property holds for all subtrees of a BST

Inserting into a BST



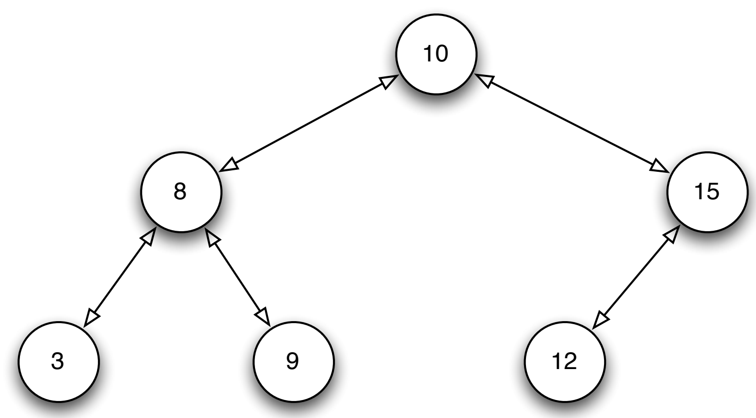
- Compare new value to current node, if greater, insert into right subtree, if lesser, insert into left subtree
- **insert(x, Node t)**
  if **(t == null)** return new Node(x)
  if (**x > t.key**), then **t.right = insert(x, t.right)**
  if (**x < t.key**), then **t.left = insert(x, t.left)**
  return **t**

Searching a BST

- **findMin(t)** // return left-most node
  if (**t.left == null**) return **t.key**
  else return **findMin(t.left)**
- **search(x,t)** // similar to insert
  if (**t == null**) return **false**
  if (**x == t.key**) return **true**
  if (**x > t.key**), then return **search(x, t.right)**
  if (**x < t.key**), then return **search(x, t.left)**

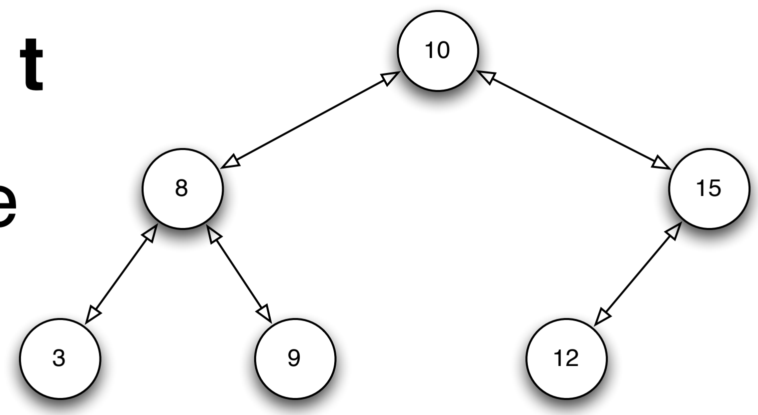
Deleting from a BST

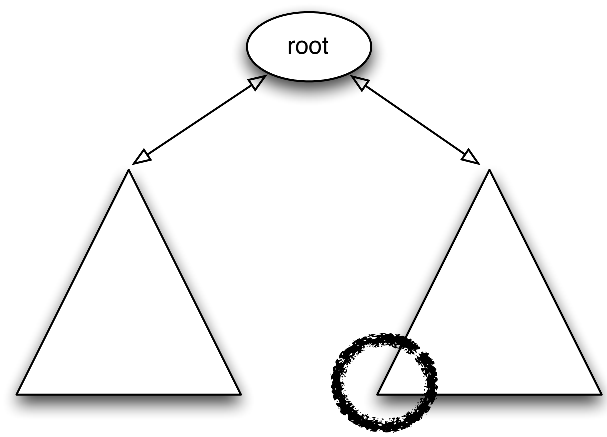
- Removing a leaf is easy, removing a node with one child is also easy
- Nodes with no grandchildren are easy
- What about nodes with grandchildren?



A Removal Strategy

- First, find node to be removed, **t**
- Replace with the smallest node from the right subtree
  - **a = findMin(t.right);**
    t.key = a.key;
- Then delete original smallest node in right subtree
  remove(a.key, t.right)

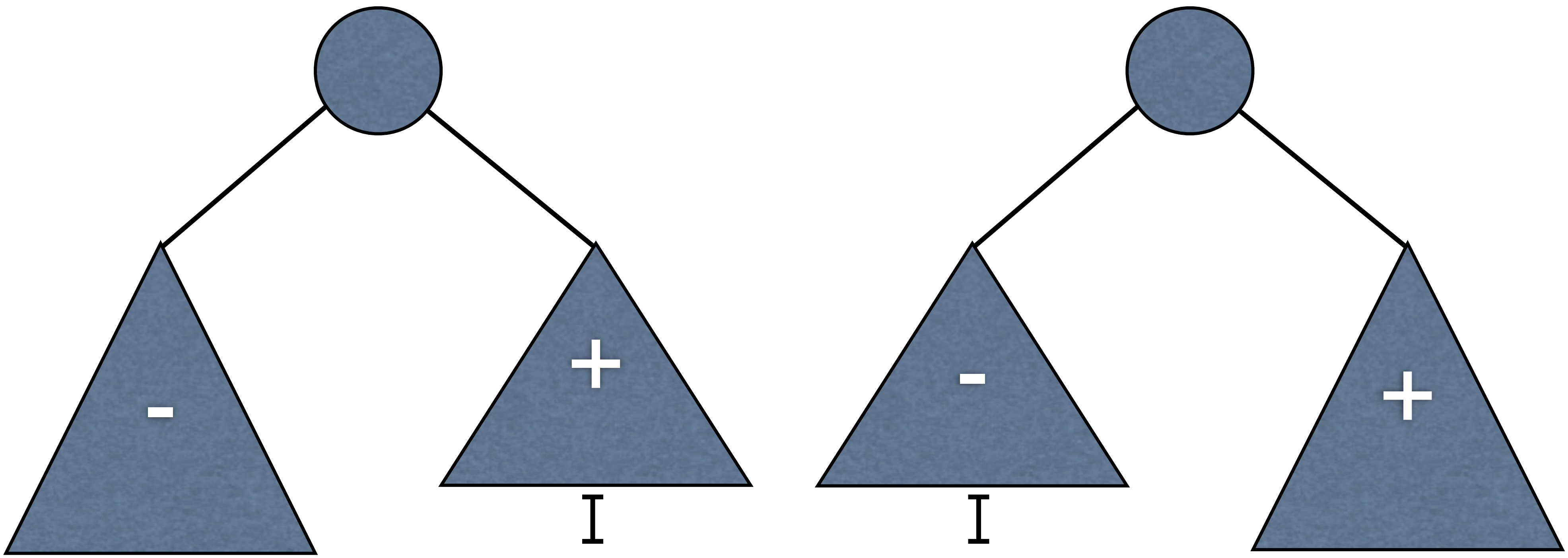




AVL Trees

- Motivation: want height of tree to be close to log N
- AVL Tree Property:
  For each node, all keys in its left subtree are less than the node's and all keys in its right subtree are greater. **Furthermore, the height of the left and right subtrees differ by at most 1**

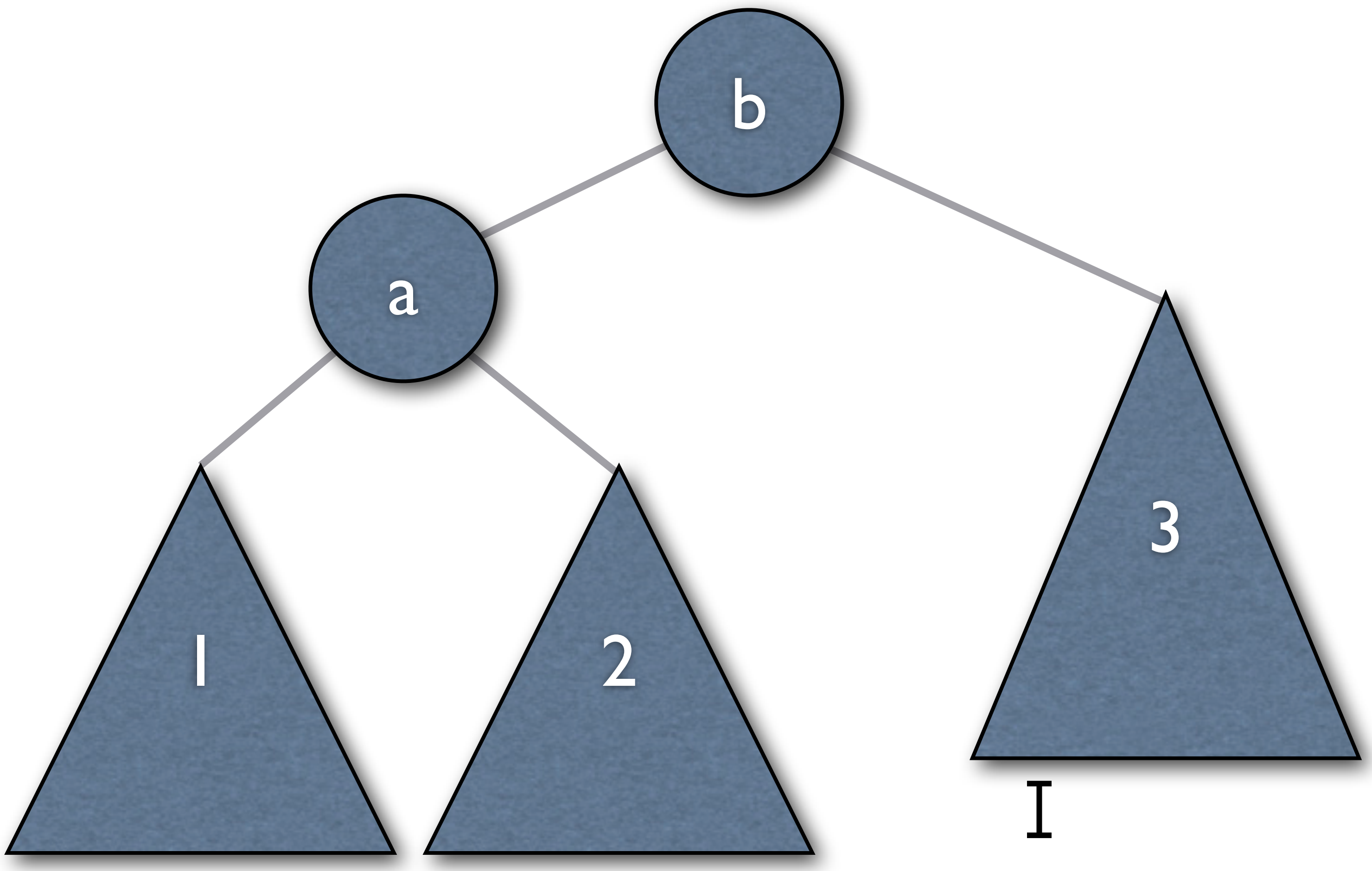
AVL Tree Visual



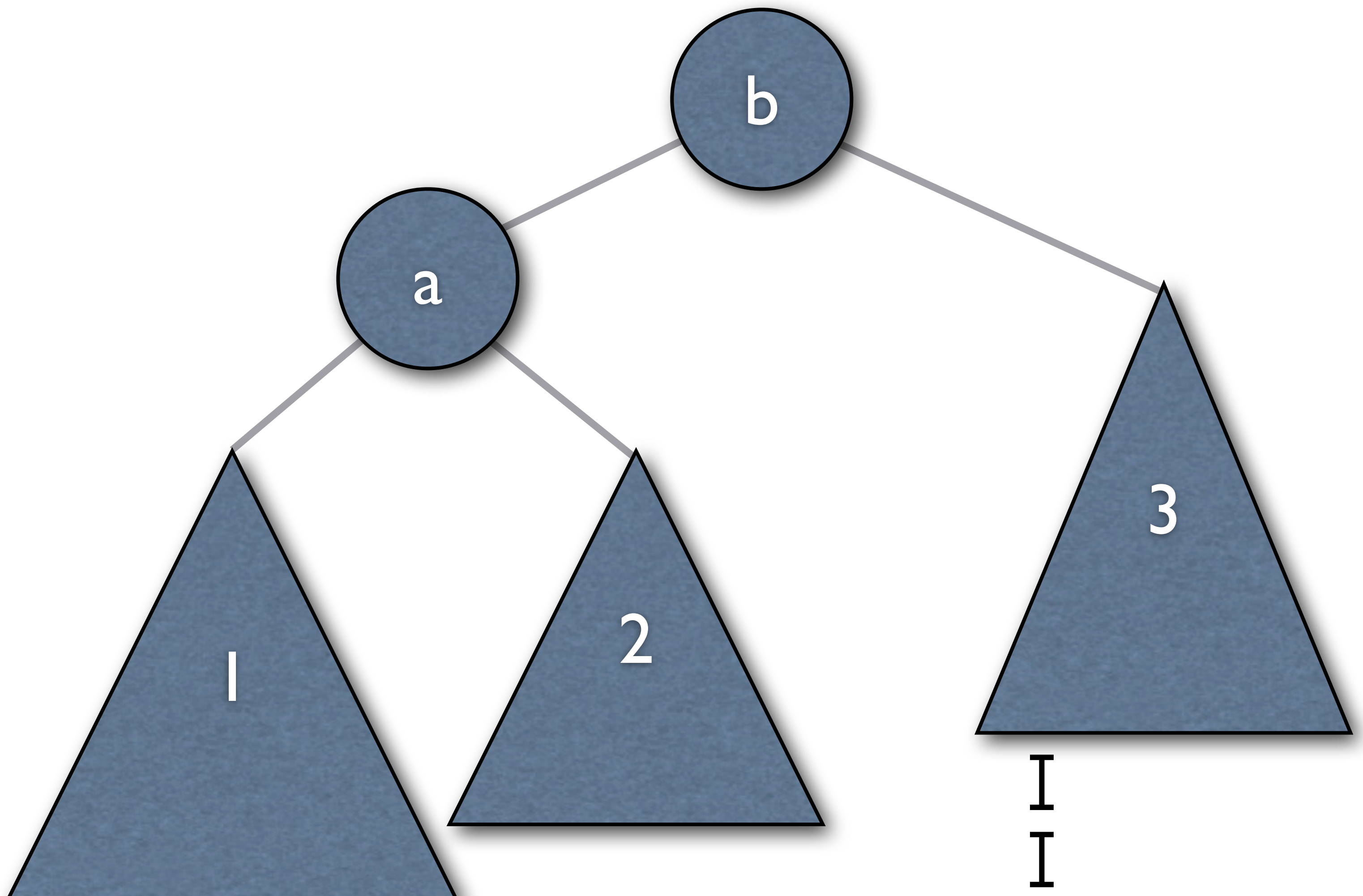
Tree Rotations

- To balance the tree after an insertion violates the AVL property,
  - rearrange the tree; make a new node the root.
  - This rearrangement is called a **rotation.**
  - There are 2 types of rotations.

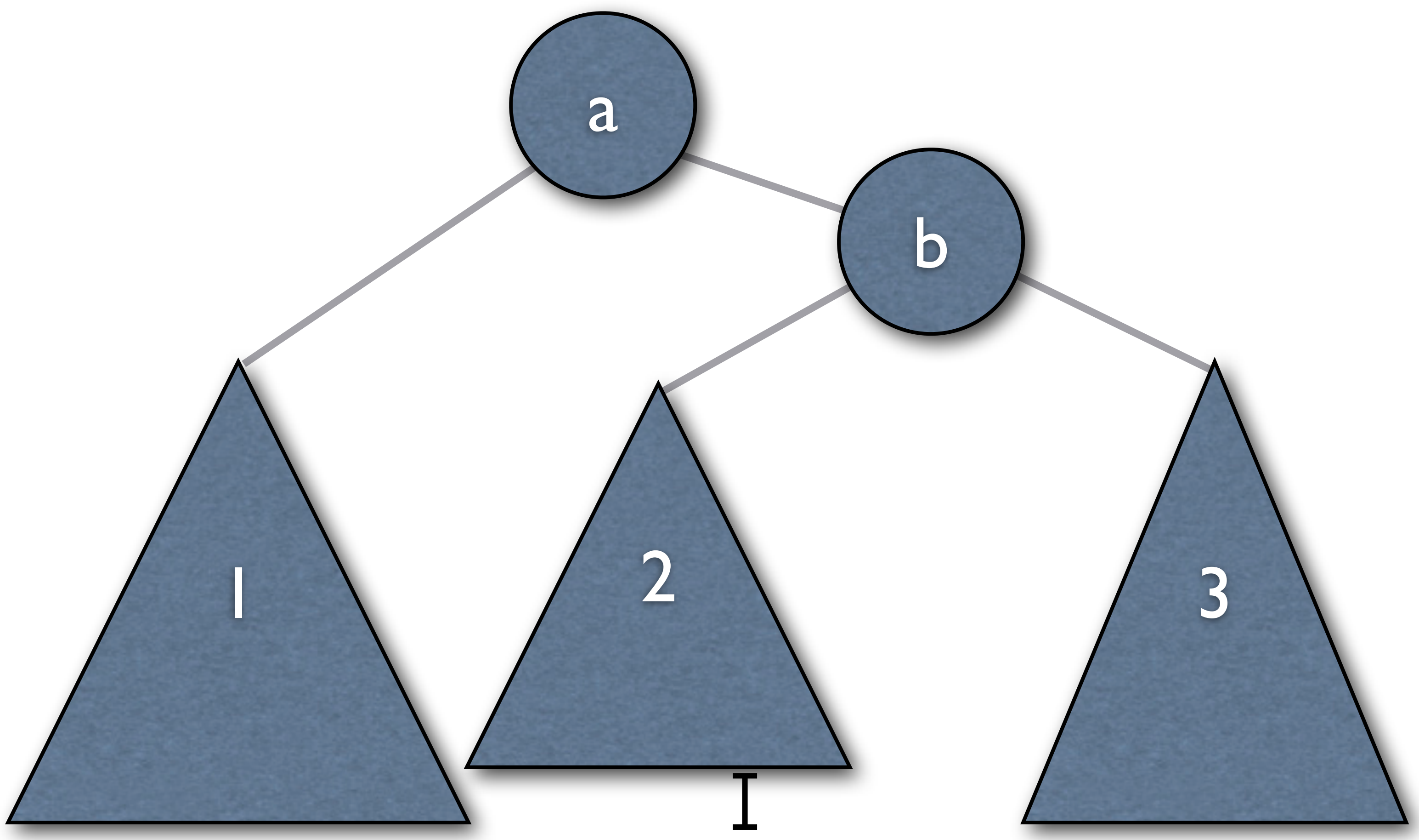
AVL Tree Visual: Before insert



AVL Tree Visual: After insert



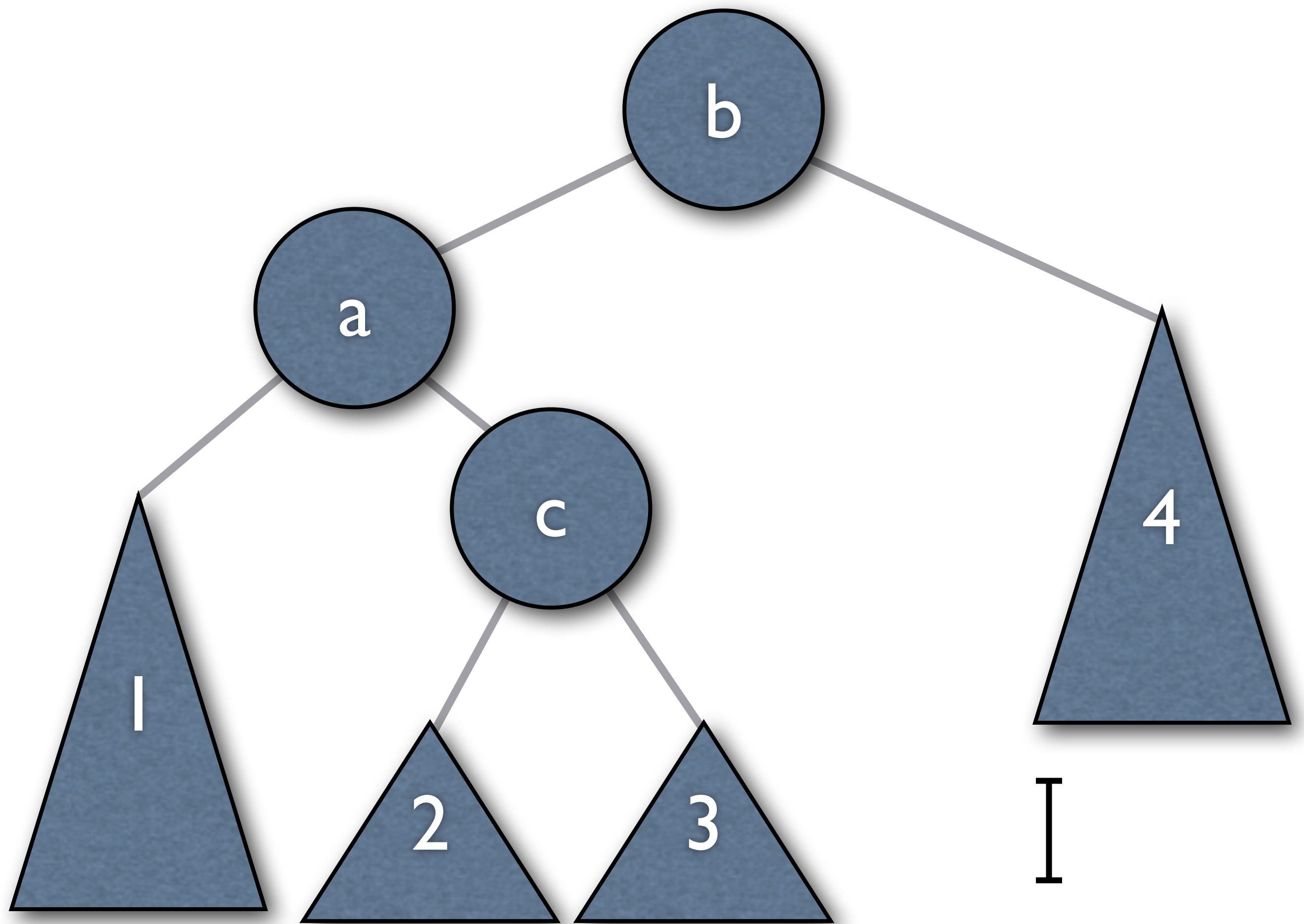
AVL Tree Visual: Single Rotation



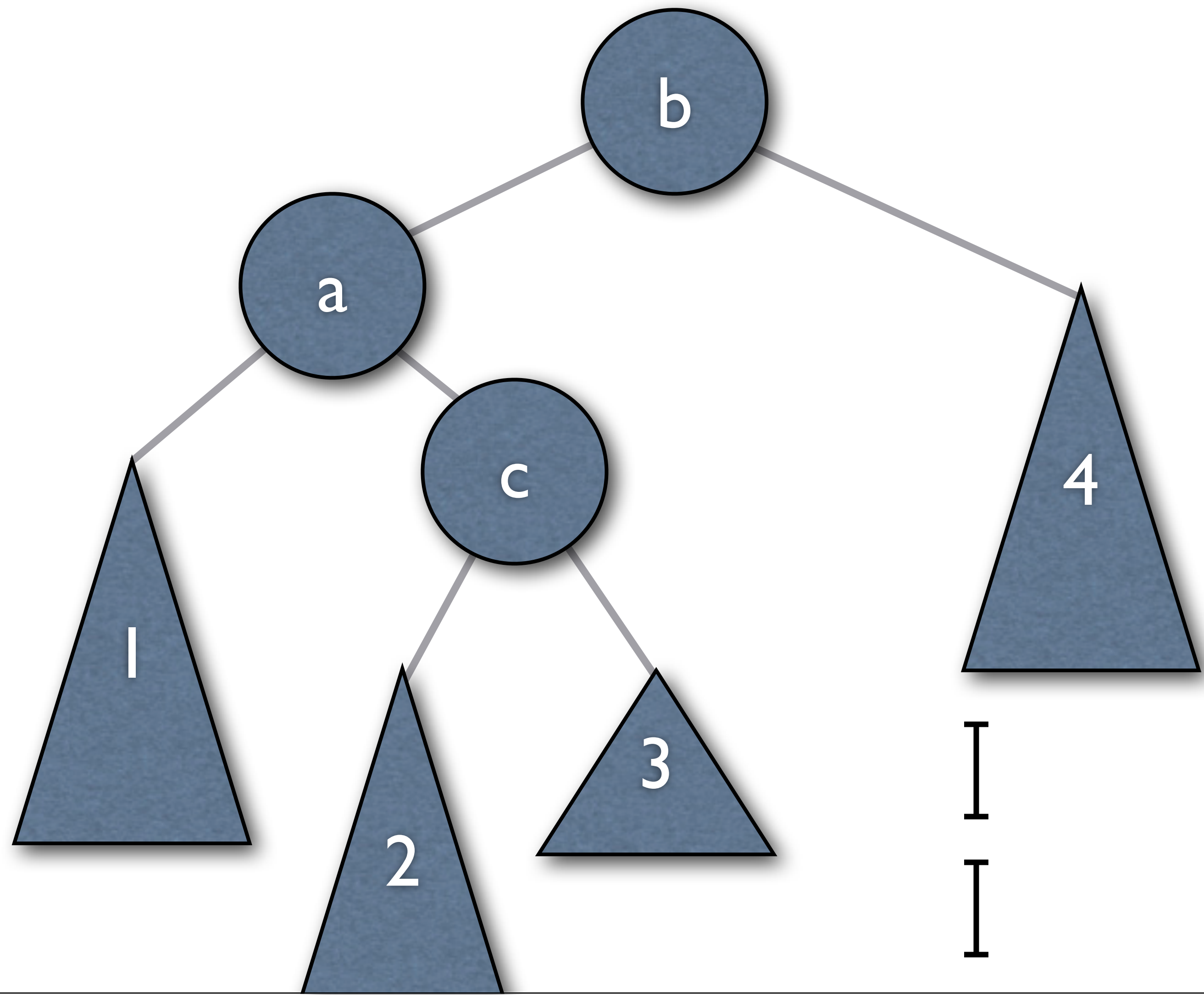
AVL Tree
Single Rotation

- Works when new node is added to outer subtree (left-left or right-right)
- What about inner subtrees? (left-right or right-left)

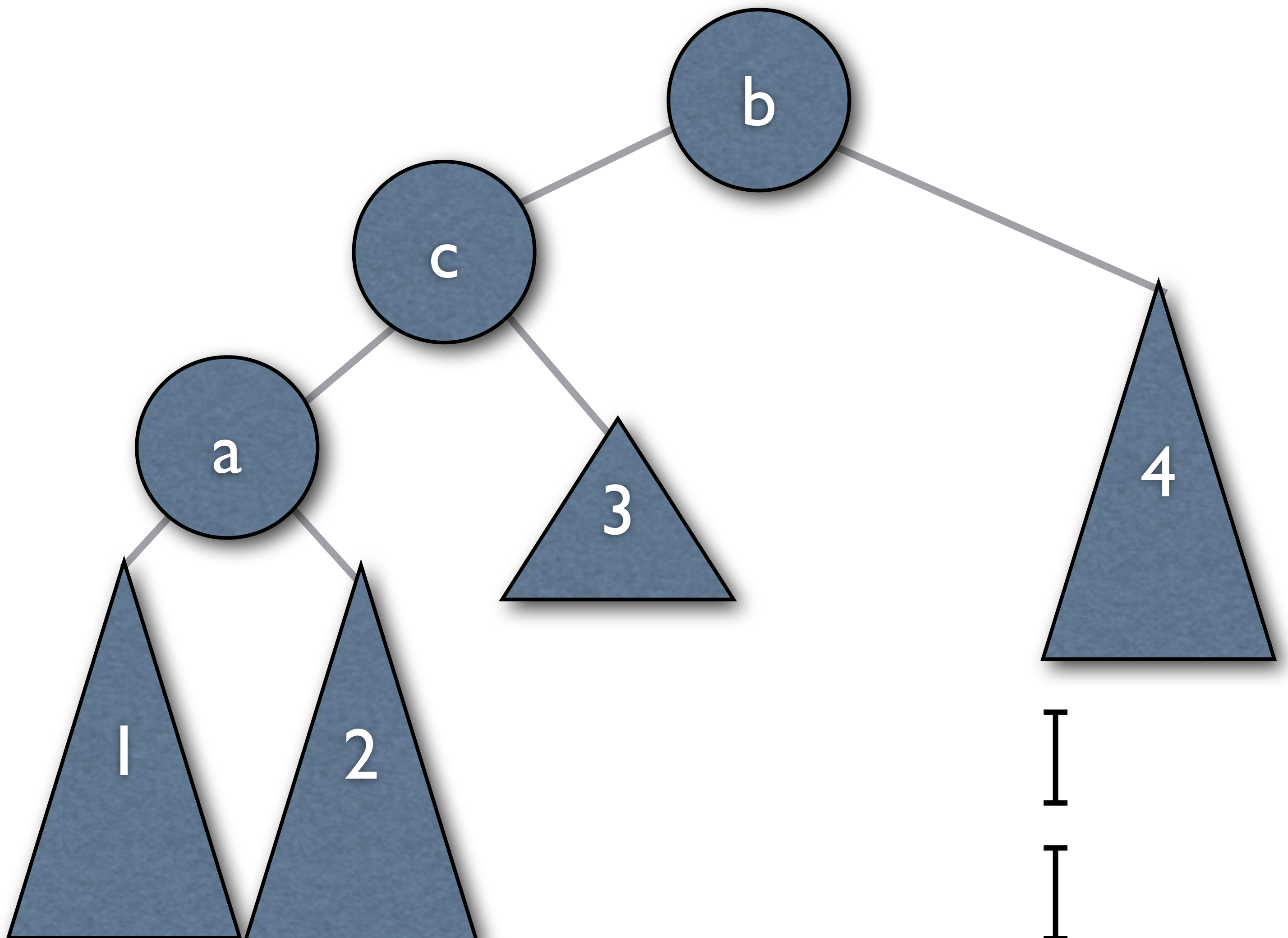
AVL Tree Visual: Before Insert 2



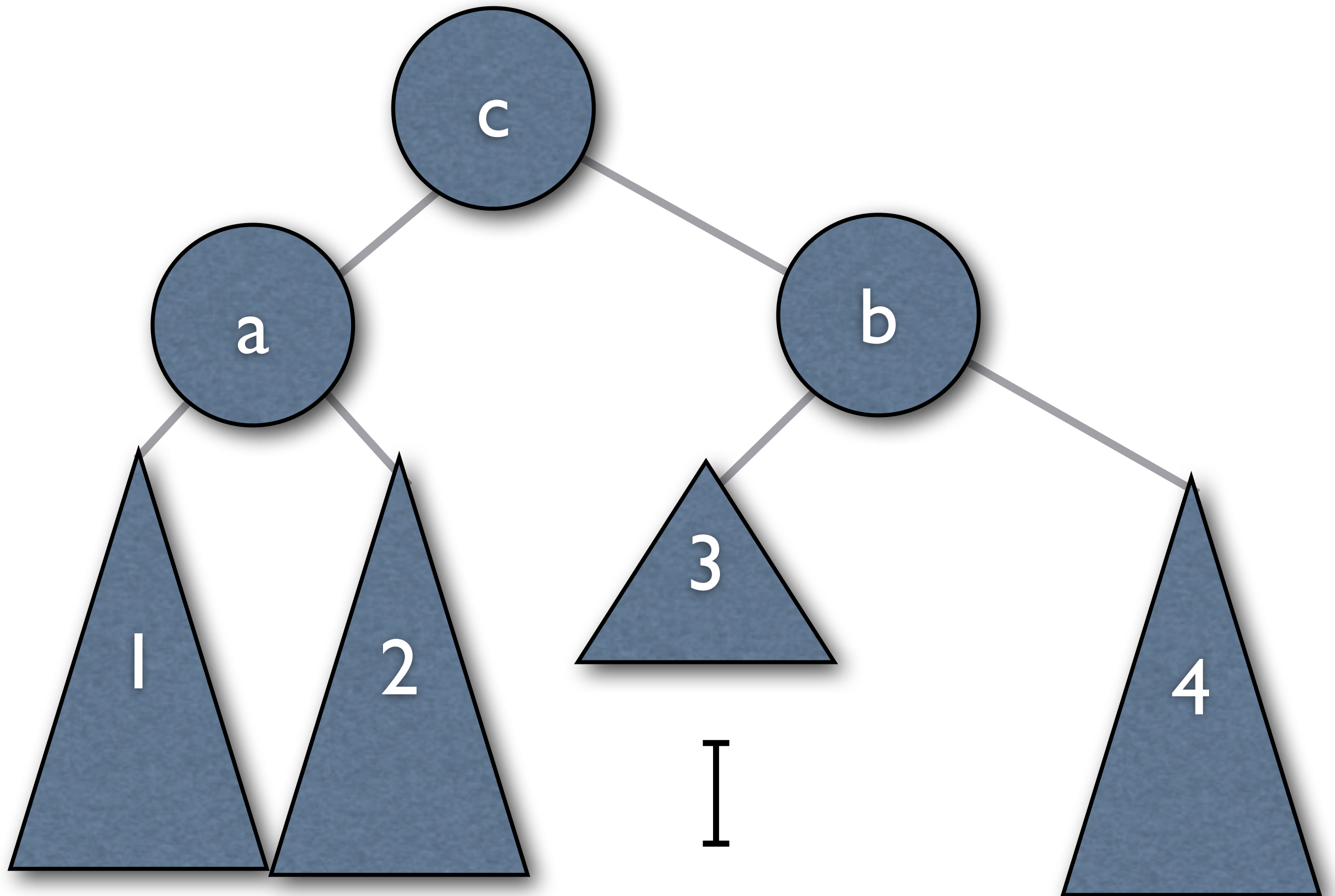
AVL Tree Visual: After Insert 2



AVL Tree Visual: Double Rotation



AVL Tree Visual: Double Rotation



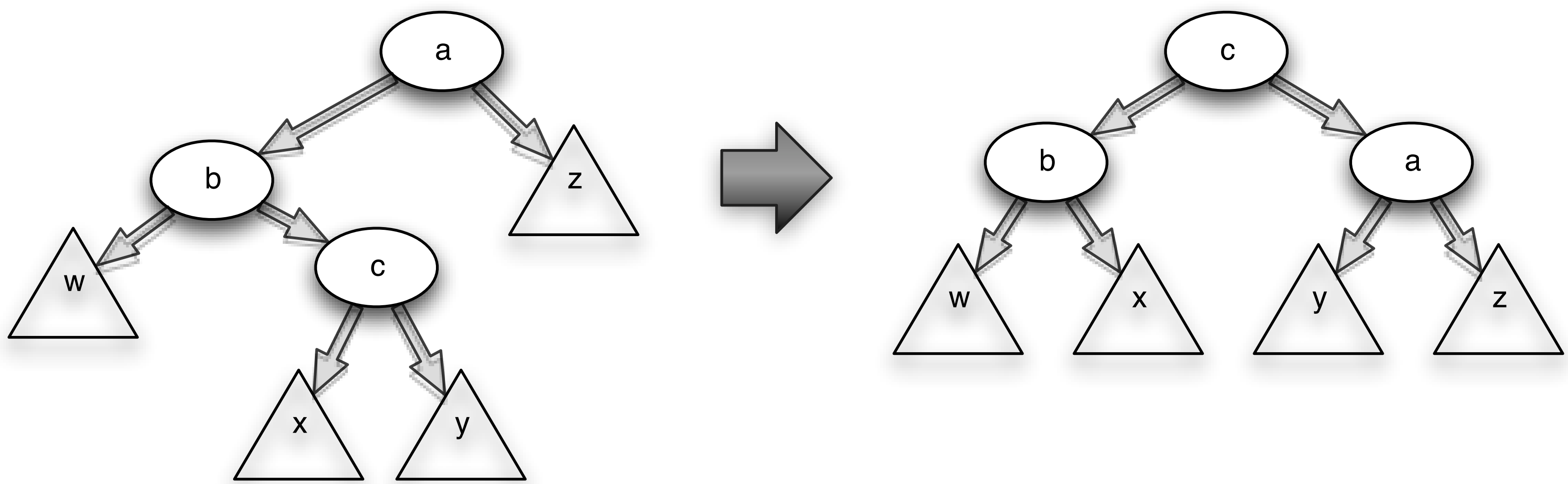
Splay Trees

- 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

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

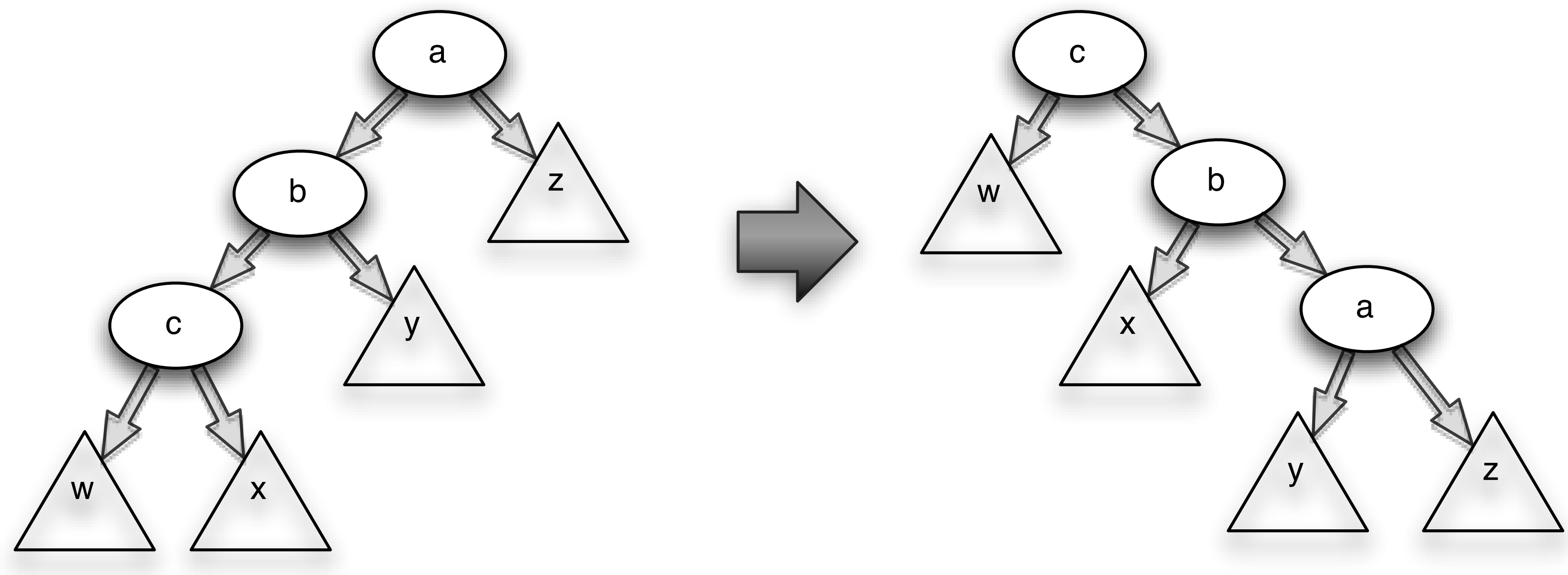


- Use when the node is the right child of a left child (or left-right)
- Double rotate, just like AVL tree

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



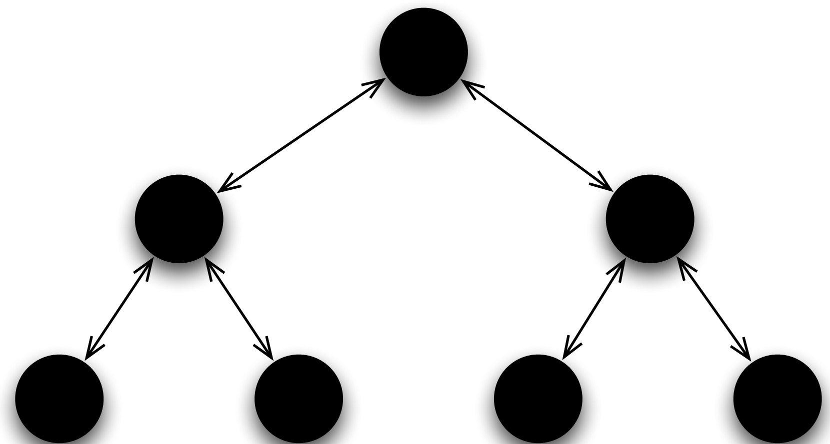
- Use when node is the right-right child (or left-left)
- Reverse the order of grandparent->parent->node
  - Make it node->parent->grandparent

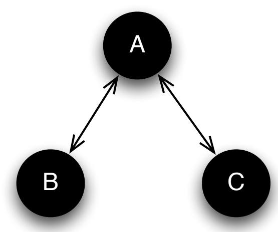
Priority Queues

- New abstract data type Priority Queue
  - Insert: add node with key
  - deleteMin: delete the node with smallest key
  - findMin: access the node with smallest key
  - (increase/decrease priority)

Heap Implementation

- Binary tree with special properties
- Heap Structure Property: all nodes are full*
- Heap Order Property: any node is smaller than its children





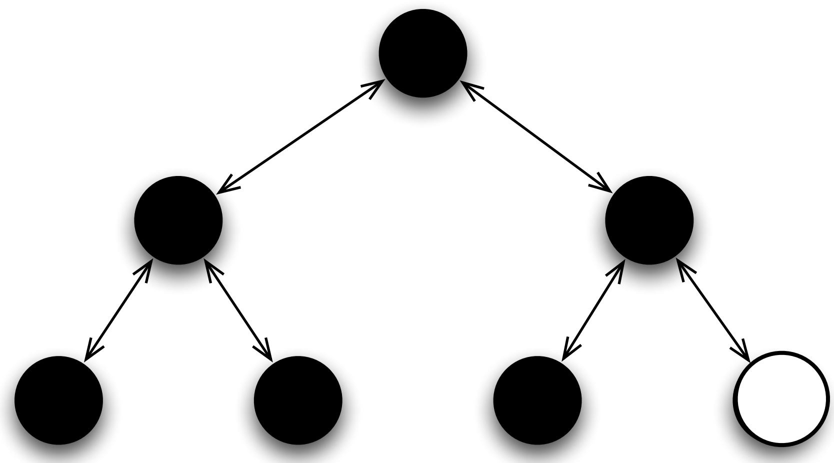
$$A < B$$
$$A < C$$
$$C \; ? \; B$$

Array Implementation

- A full tree is regular: we can store in an array
  - Root at **A[1]**
  - Root's children at **A[2]**, **A[3]**
  - Node **i** has children at **2i** and **(2i+1)**
  - Parent at **floor(i/2)**
- No links necessary, so much faster (but only constant speedup)

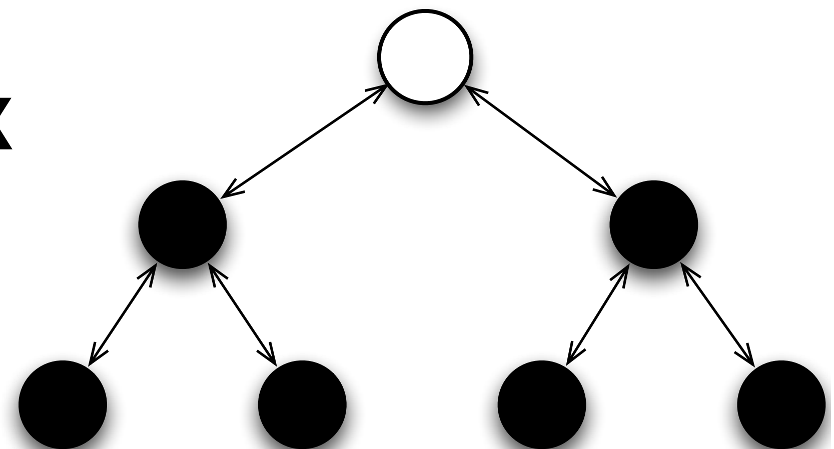
Insert

- To insert key **X**, create a hole in bottom level
- **Percolate up**
  - Is hole's parent is less than **X**
    - If so, put **X** in hole, heap order satisfied
    - If not, swap hole and parent and repeat



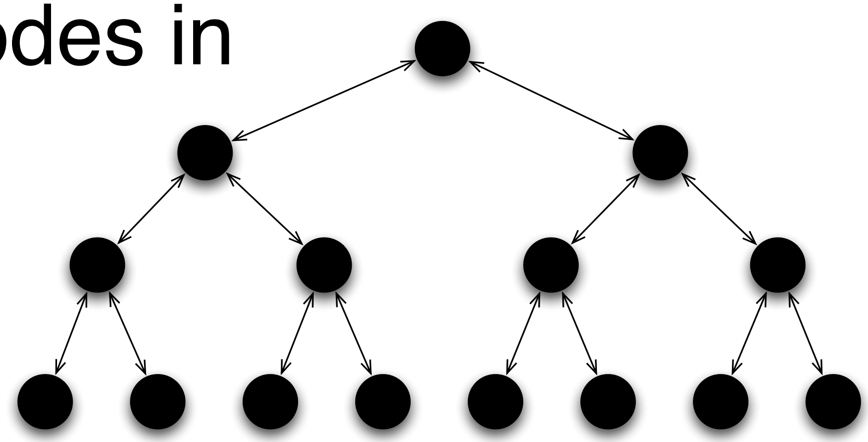
DeleteMin

- Save root node, and delete, creating a hole
- Take the last element in the heap **X**
- **Percolate down**:
  - is X is less than hole's children?
    - if so, we're done
    - if not, swap hole and smallest child and repeat



buildHeap

- Start at deepest non-leaf node
  - in array, this is node N/2
- **percolateDown** on all nodes in reverse level-order
  - for i = N/2 to 1
    percolateDown(i)



Heap Operations

- Insert – O(log N)
- deleteMin – O(log N)
- change key – O(log N)
- buildHeap – O(N)

Reading

- pre-midterm: Weiss Ch. 2, 3, 4, 6
- post-midterm: Weiss Ch. 5, 7, 8, 9, 12.6
- See schedule on class website for specific sections (i.e., which to skip)