Data Structures in Java

Session 13

Instructor: Bert Huang

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

Announcements

- Homework 3 theory due now
- Midterm exam Thursday
- Homework 3 Programming due next Tuesday 10/27

Review

- buildHeap in linear time
  - jam array into heap structure
  - fix order by calling percolateDown on nodes in reverse order
  - Why in reverse order?

Today's Plan

- Review for the midterm

Math Background: Exponents

$$\begin{aligned} X^A X^B &= X^{A+B} \\ \frac{X^A}{X^B} &= X^{A-B} \\ \left(X^A\right)^B &= X^{AB} \\ X^N + X^N &= 2X^N \neq X^{2N} \\ 2^N + 2^N &= 2^{N+1} \end{aligned}$$

Math Background: Logarithms

$$X^A = B \text{ iff } \log_X B = A$$

$$\begin{aligned}
\log_A B &= \frac{\log_C B}{\log_C A}; \; \boxed{A, B, C > 0, A \neq 1} \\
\log AB &= \log A + \log B; \; \boxed{A, B > 0}
\end{aligned}$$

Math Background: Series

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$$

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$$

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$$

$$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$$

Definitions

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

$$
\begin{aligned}
T(N) &= O(f(N)) \leftarrow T(N) \leq cf(N) \\
T(N) &= \Omega(g(N)) \leftarrow T(N) \geq cg(N) \\
T(N) &= \Theta(h(N)) \leftarrow \begin{matrix} T(N) = O(h(N)) \\ T(N) = \Omega(h(N)) \end{matrix}
\end{aligned}
$$

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

Definitions

- Alternately, O(f(N)) can be thought of as meaning

$$T(N) = O(f(N)) \leftarrow \lim_{N\to\infty} f(N) \geq \lim_{N\to\infty} T(N)$$

- Big-Oh notation is also referred to as **asymptotic** analysis, for this reason.

Comparing Growth Rates

$$T_1(N) = O(f(N)) \text{ and } T_2(N) = O(g(N))$$

then

$$\begin{aligned} &(a) \qquad T_1(N) + T_2(N) = O(f(N) + g(N)) \\ &(b) \qquad T_1(N)T_2(N) = O(f(N)g(N)) \end{aligned}$$

- If you have to, use l'Hôpital's rule

$$\lim_{N\to\infty} f(N)/g(N) = \lim_{N\to\infty} f'(N)/g'(N)$$

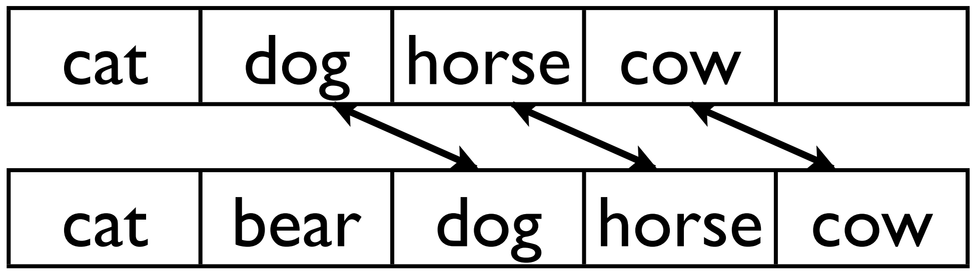
Abstract Data Type: Lists

- 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

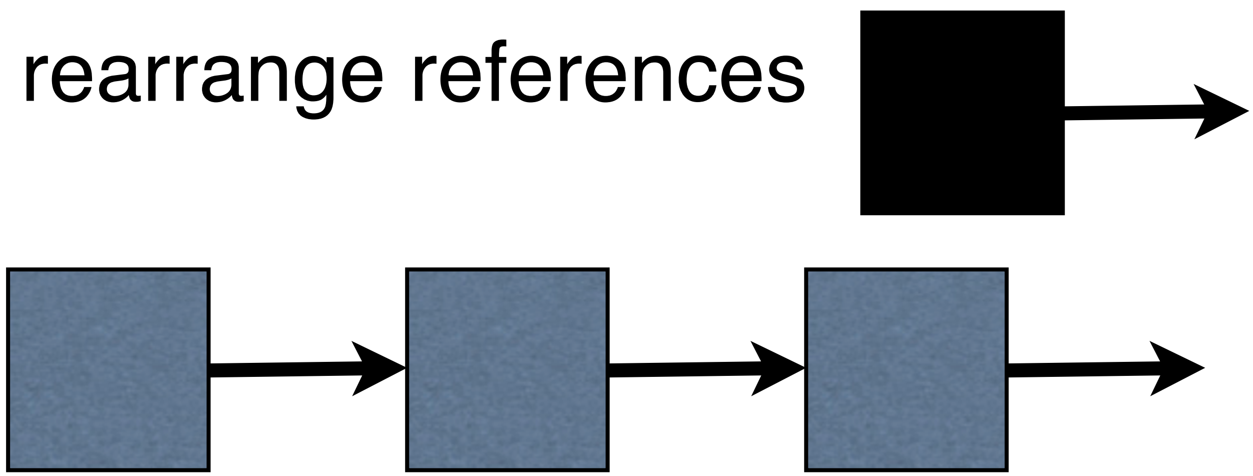
Array Implementation of Lists



- Insert - need to shift higher-indexed elements →
- Delete - need to shift higher-indexed elements ←
- Get - easy
- How to insert more than array size?
  - Create new, larger array. Copy to new array.

Linked List Implementation

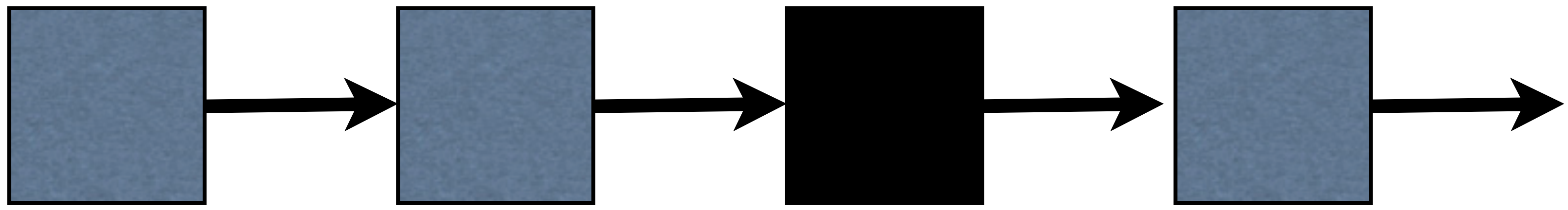
- Store elements in objects
- Each object has a reference to its next object
- Insert - rearrange references



- But we need to find the previous element

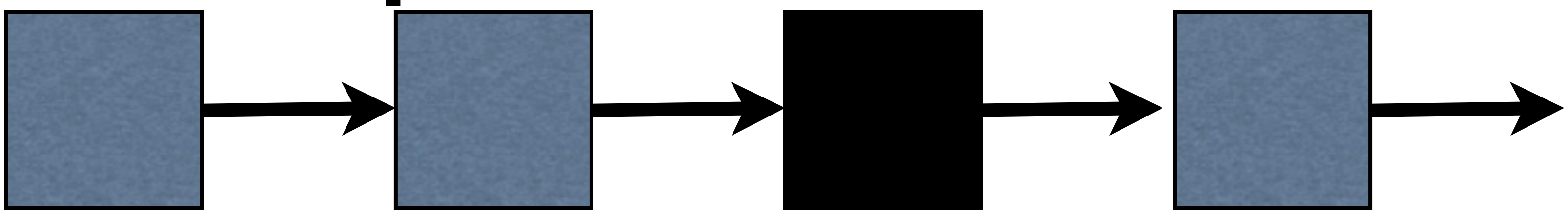
Linked List Implementation

- Store elements in objects
- Each object has a reference to its next object
- Insert - rearrange references



- But we need to find the previous element

Linked List Implementation



- Finding an element in a linked list is slower
- If we keep a **head** reference, finding the last element takes N steps
- If we keep a head and a **tail** reference*, finding the middle element takes N/2 steps
- Be careful iterating; navigate the list smartly

Linked Lists vs. Array Lists

- <u>Linked Lists</u>
- No additional penalty on size
- Insert/remove O(1)*
- get kth costs O(N)*
- Need some extra memory for links

- <u>Array Lists</u>
- Need to estimate size/grow array
- Insert/remove O(N)*
- get kth costs O(1)
- Arrays are compact in memory

Stacks

- A Stack is an ADT very similar to a list
- Can be implemented with a list, but limited to some O(1) operations
- Yet many important and powerful algorithms use stacks

Stack Definition

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

Evaluating Postfix

- Postfix notation places operator after operands
  - Ambiguous Infix: 3 + 2 * 10 ((3+2) * 10)
  - Postfix: 3 2 + 10 * ((3 2 +) 10 *)

(As opposed to) 3 2 10 * + (3 (2 10 *) +)

Evaluating Postfix

- Postfix notation places operator after operands
  - Ambiguous Infix: (3 + 2)* 10 ((3+2) * 10)
  - Postfix: 3 2 + 10 * ((3 2 +) 10 *)

(As opposed to) 3 2 10 * + (3 (2 10 *) +)

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

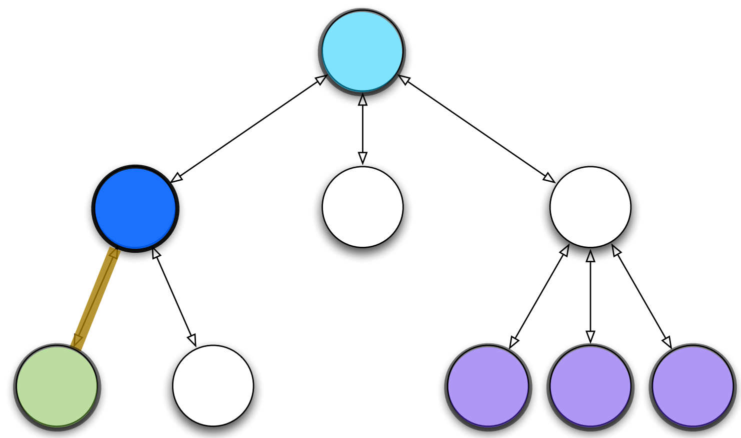
- Linked List
  - add(x,0) to enqueue, remove(N-1) to dequeue
- Array List won't work well!
  - add(x,0) is expensive
  - Solution: use a circular array

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

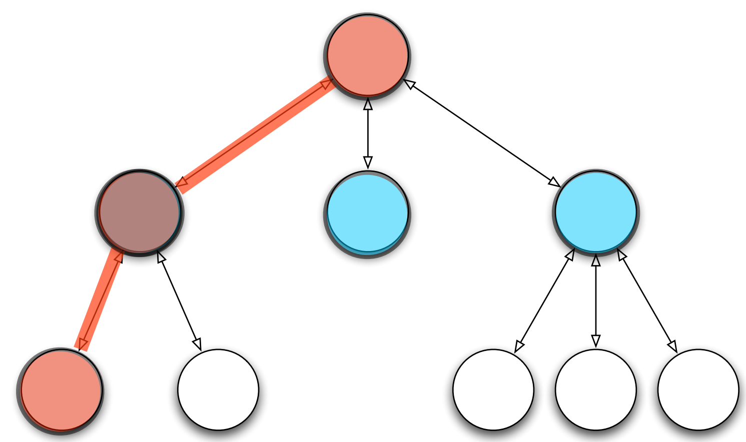
Tree Terminology

- Just like Linked Lists, **Trees** are collections of **nodes**
- Conceptualize trees upside down (like family trees)
  - the top node is the **root**
  - nodes are connected by **edges**
  - edges define **parent** and **child** nodes
  - nodes with no children are called **leaves**



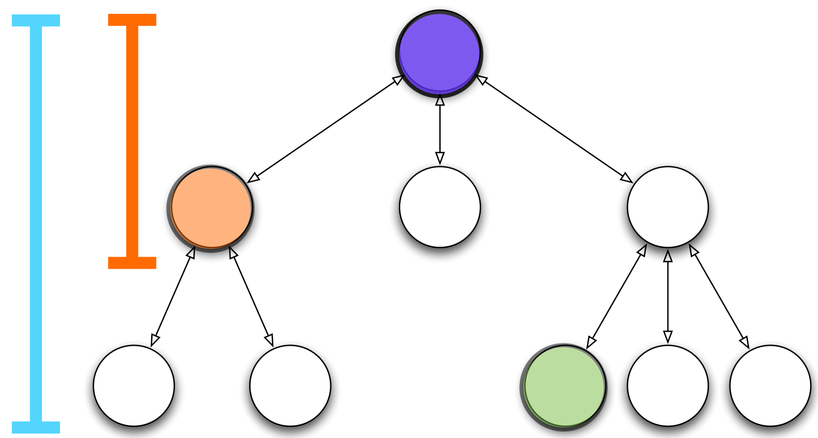
More Tree Terminology

- Nodes that share the same parent are **siblings**
- A **path** is a sequence of nodes such that the next node in the sequence is a child of the previous



More Tree Terminology

- a node's **depth** is the length of the path from root
- the **height** of a tree is the maximum depth
- if a path exists between two nodes, one is an **ancestor** and the other is a **descendant**



Tree Implementation

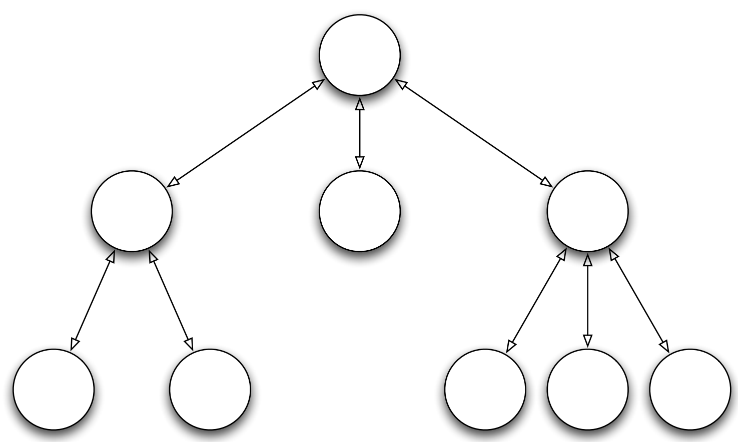
- Many possible implementations
- One approach: each node stores a list of children
-
```
public class TreeNode<T> {
    T Data;
    Collection<TreeNode<T>> myChildren;
}
```

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)
  ```



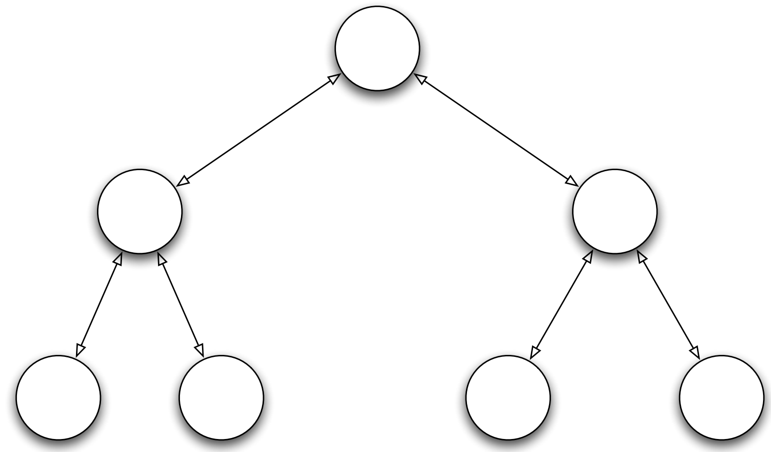
Binary Trees

- 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)

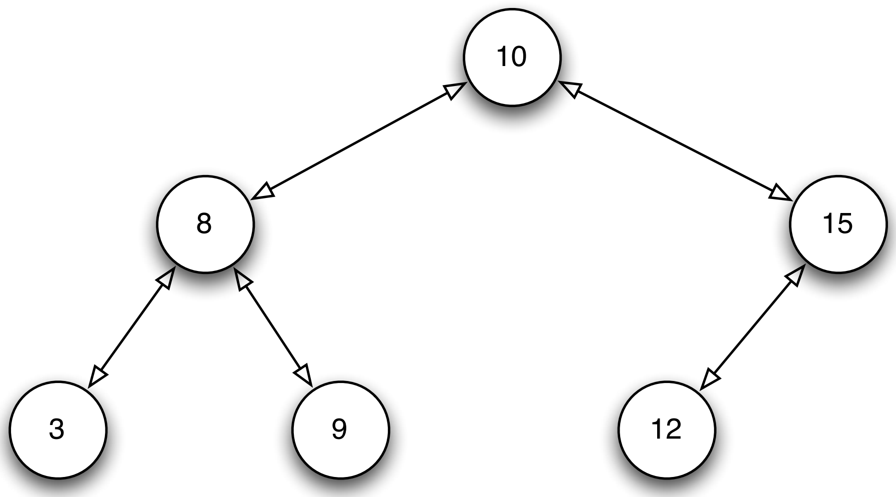
```
inorder(x.left)
print(x)
inorder(x.right)
```



Search (Tree) ADT

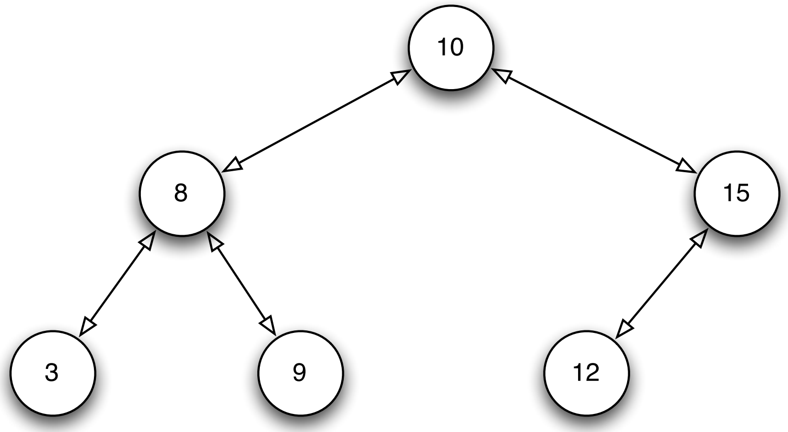
- ADT that allows insertion, removal, and searching by **key**
  - A **key** is a value that can be compared
  - In Java, we use the **Comparable** interface
  - Comparison must obey transitive property
- Search ADT doesn’t use any index

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



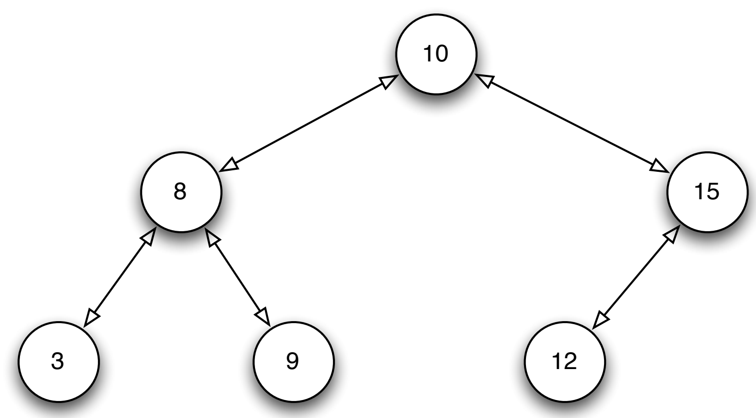
- Compare new value to current node, if greater, insert into right subtree, if lesser, insert into left subtree
- **insert(x, Node t)**
  if **(t == null)** return new Node(x)
  if (**x > t.key**), then **t.right = insert(x, t.right)**
  if (**x < t.key**), then **t.left = insert(x, t.left)**
  return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

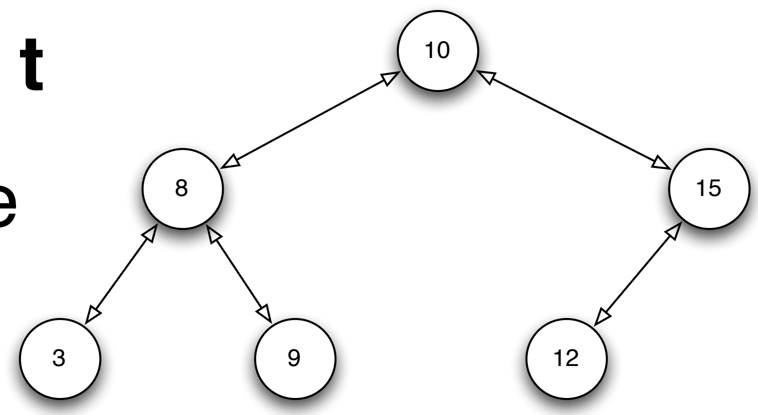
Deleting from a BST

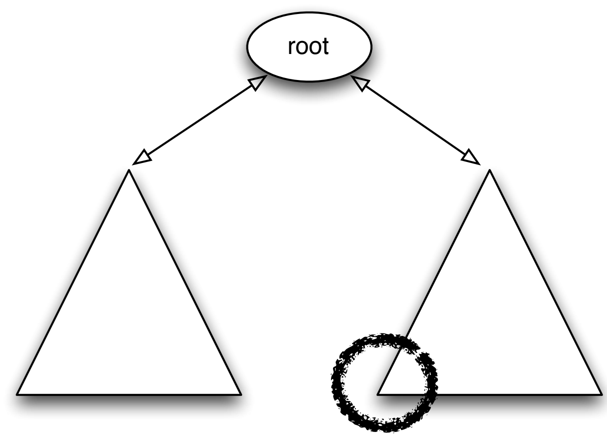
- Removing a leaf is easy, removing a node with one child is also easy
- Nodes with no grandchildren are easy
- What about nodes with grandchildren?



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)





Sorting with BST

- Suppose we have a built BST
- How to print out nodes in order?
  - inorder traversal
- Running time?
  - O(N)

Tradeoffs

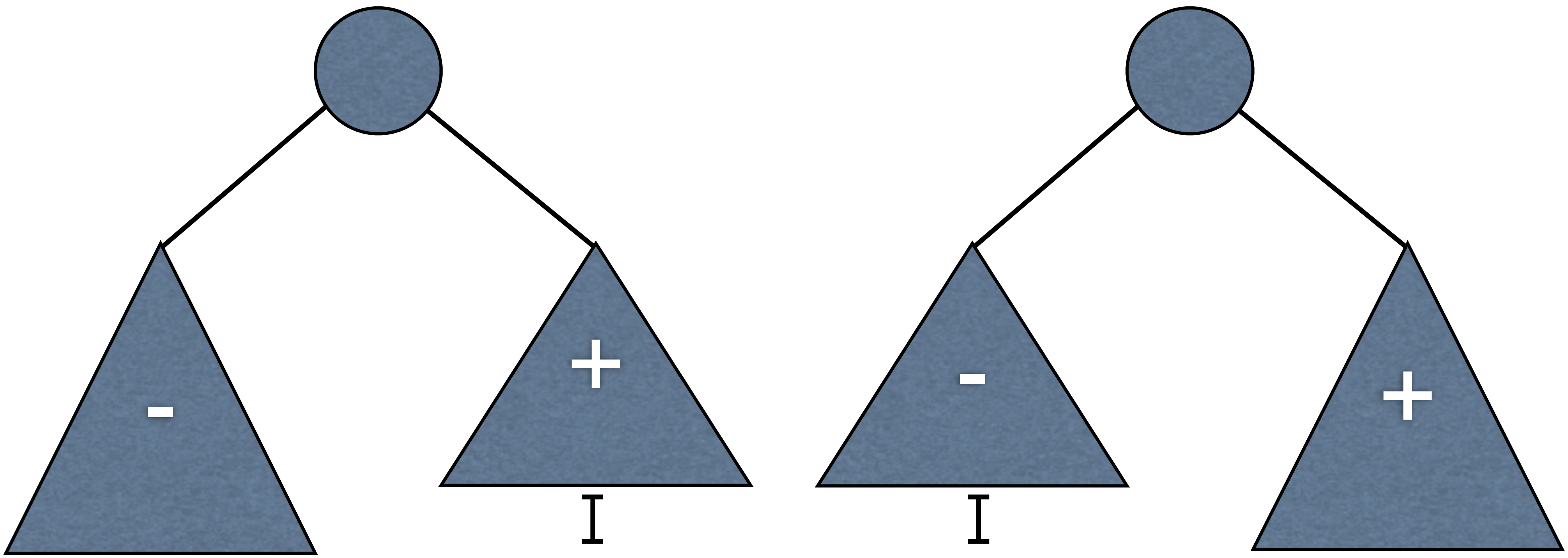
	insert	remove	search	index
ArrayList	O(N)	O(N)	O(N)	O(1)
LinkedList	O(1)	O(1)	O(N)	O(N)
Stack/Queue	O(1)	O(1)	N/A	N/A
BST	O(d)=O(N)	O(d)=O(N)	O(d)=O(N)	N/A
AVL	O(log N)	O(log N)	O(log N)	N/A

- There may not be free lunch, but sometimes there's a cheaper lunch

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**

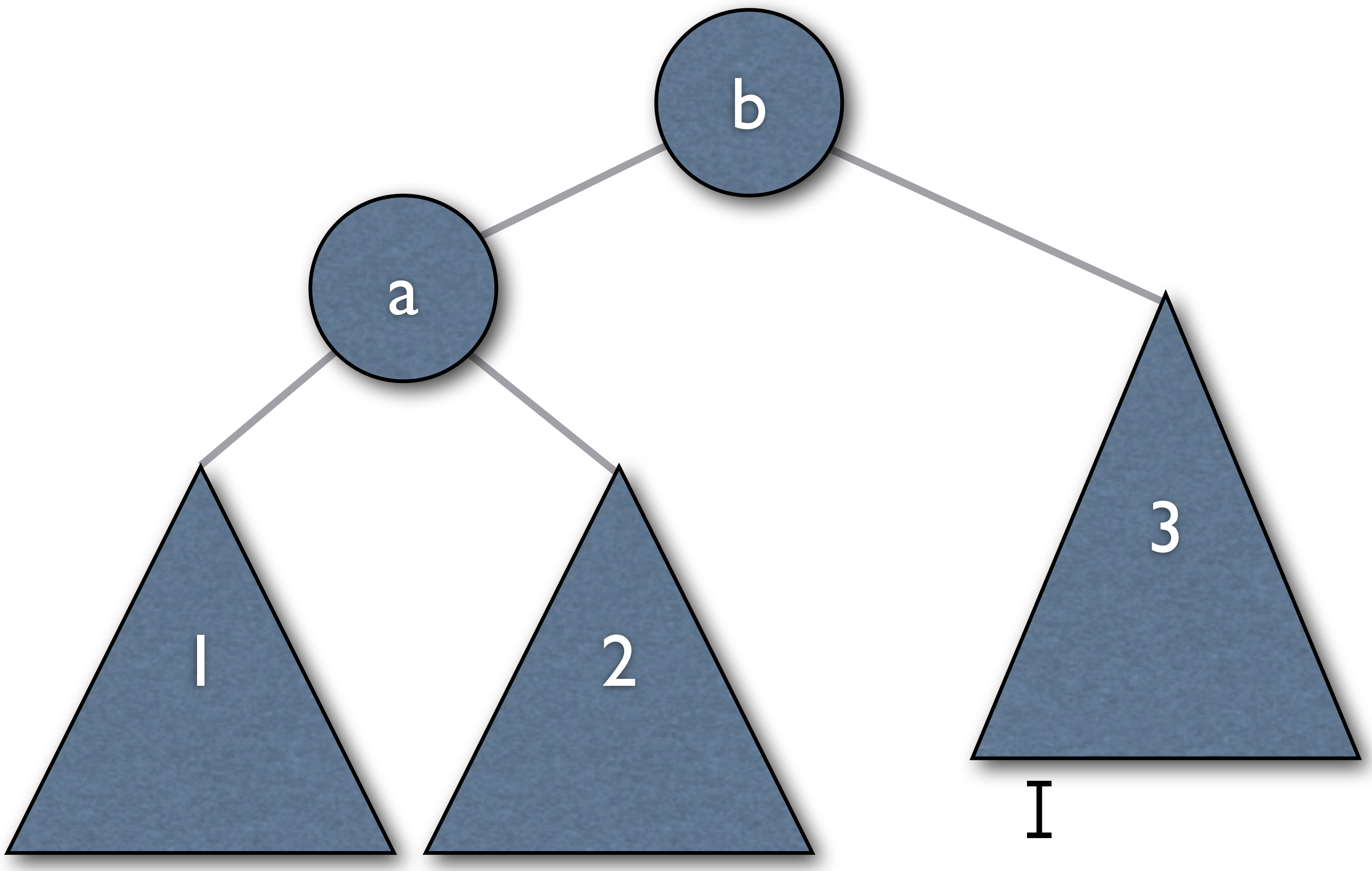
AVL Tree Visual



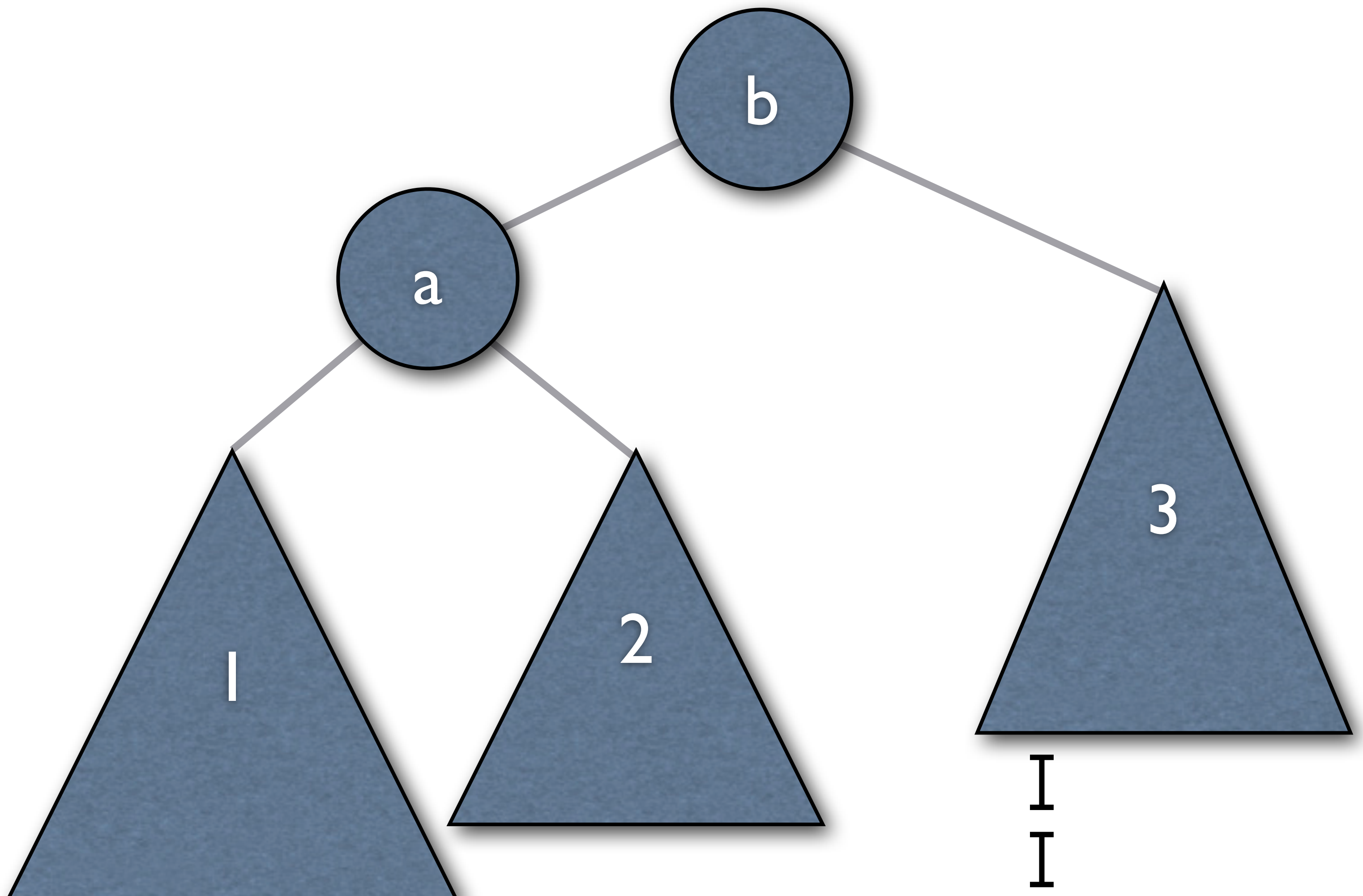
Tree Rotations

- 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.

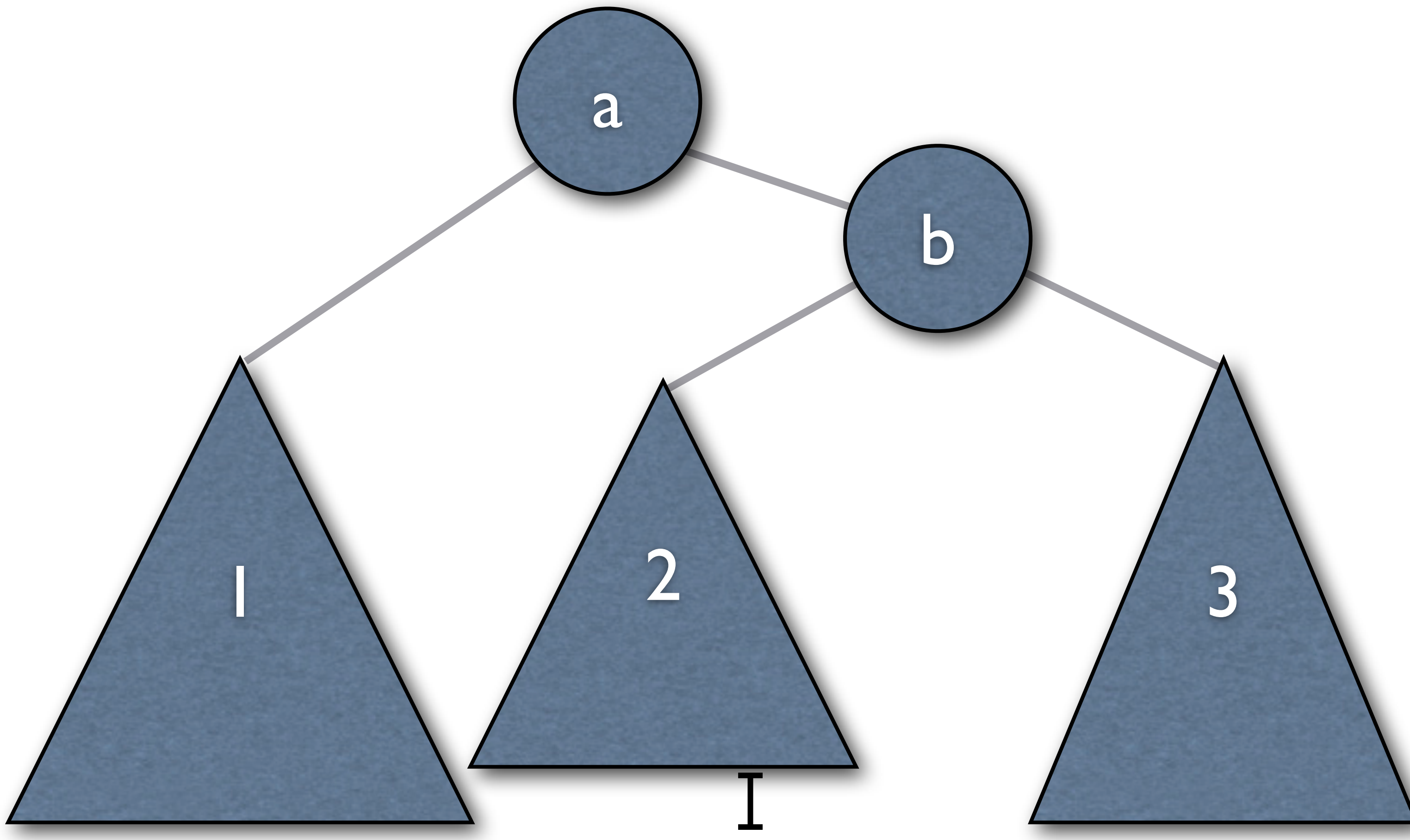
AVL Tree Visual: Before insert



AVL Tree Visual: After insert



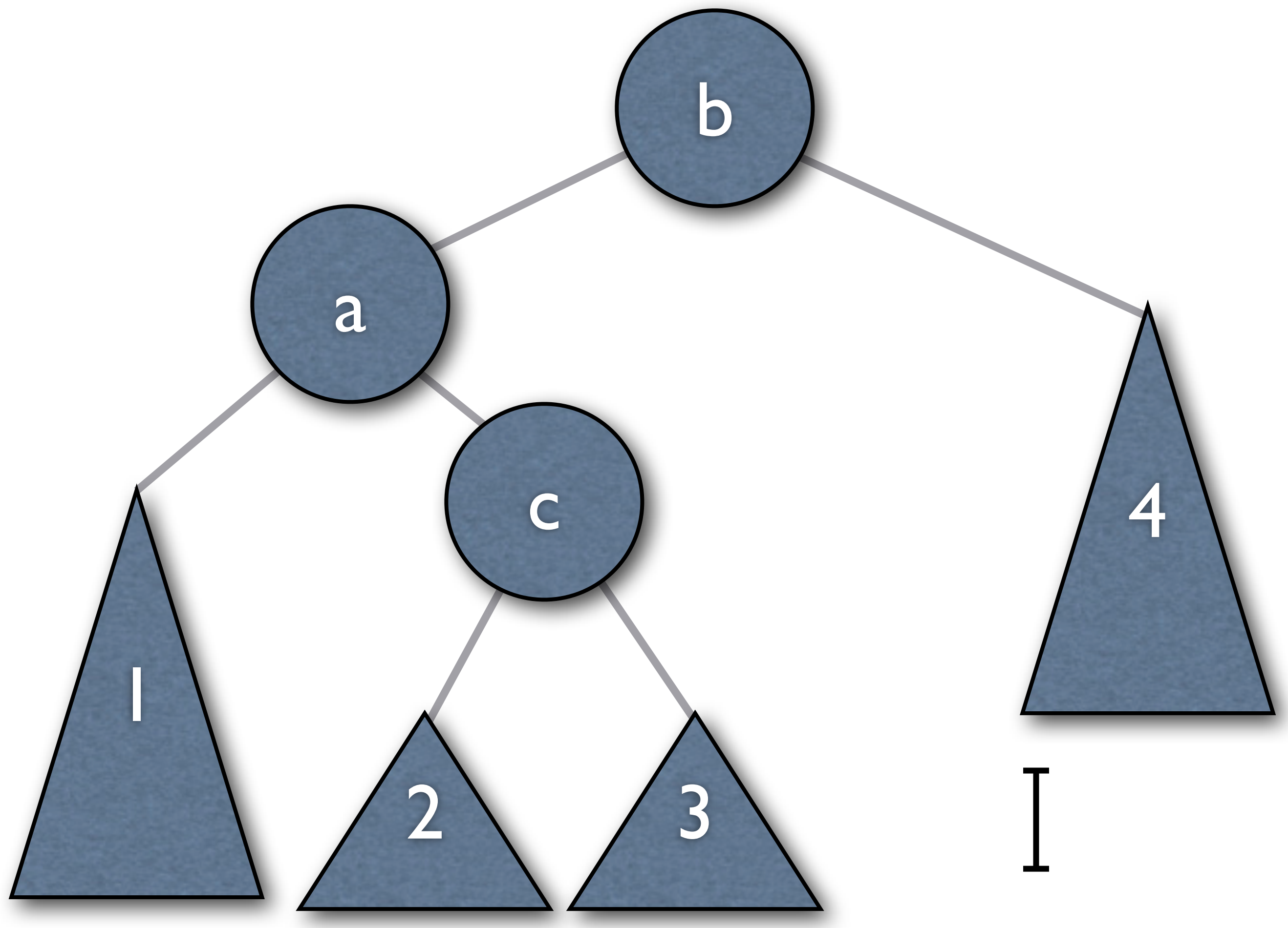
AVL Tree Visual: Single Rotation



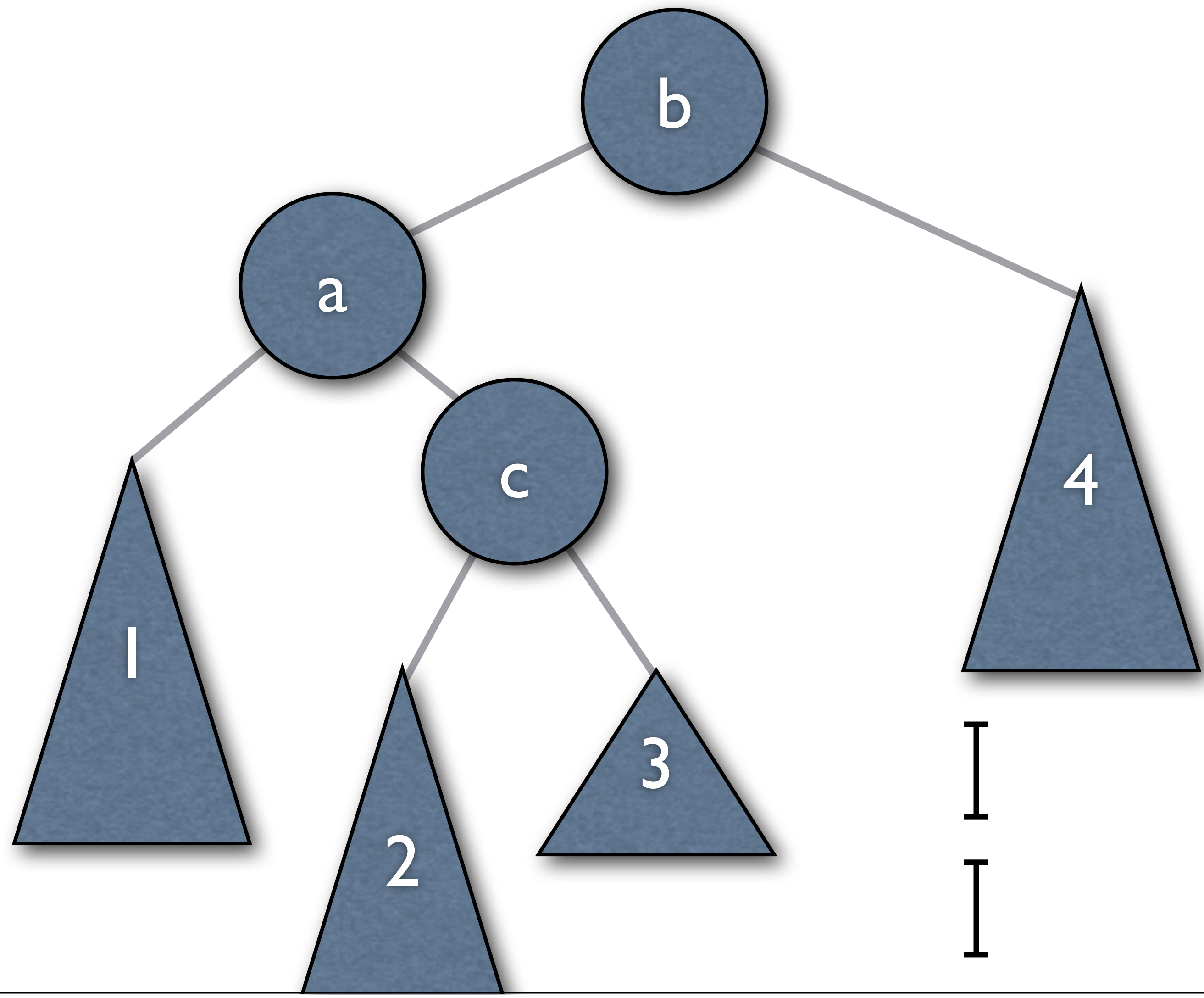
AVL Tree
Single Rotation

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

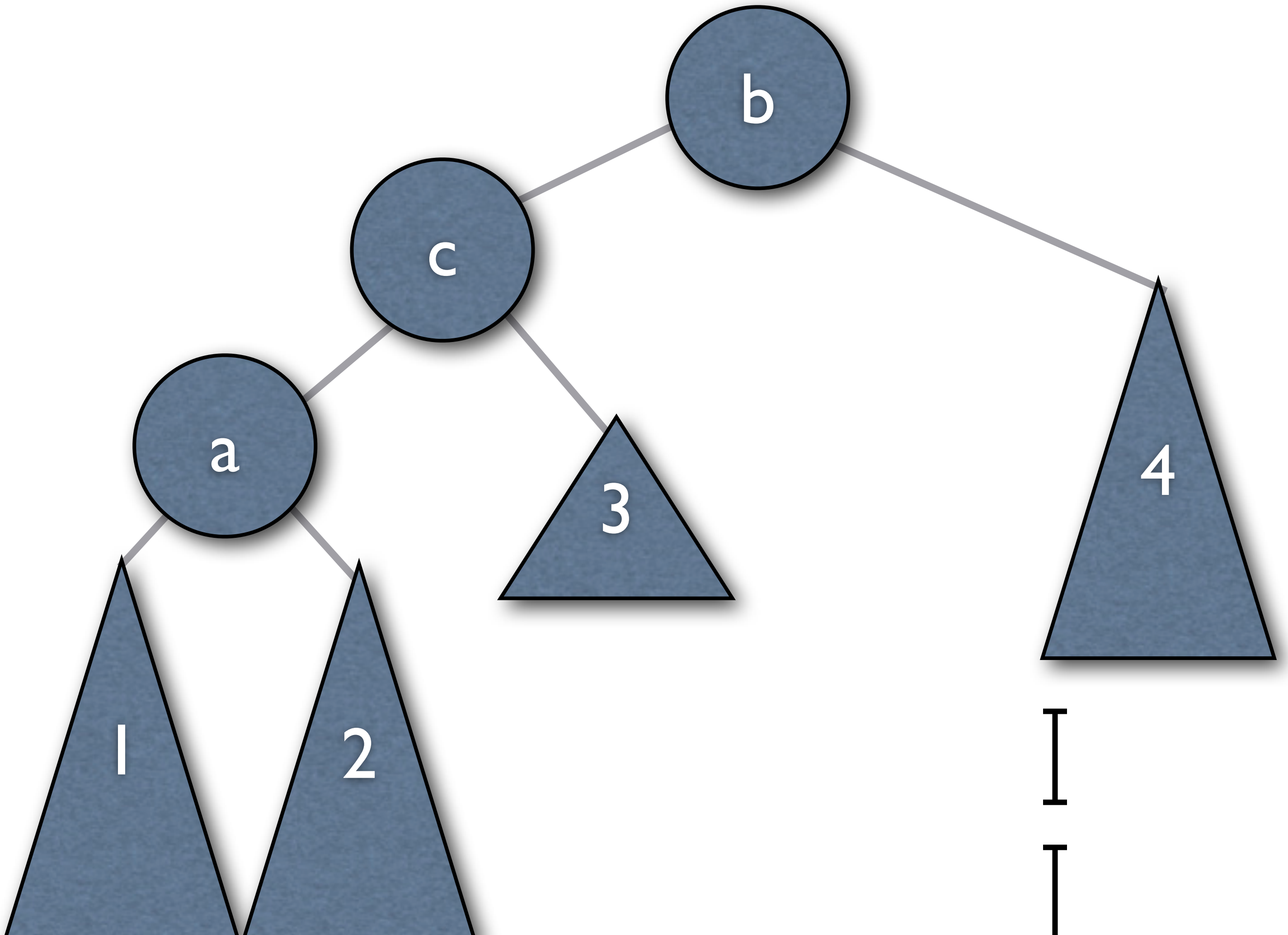
AVL Tree Visual: Before Insert 2



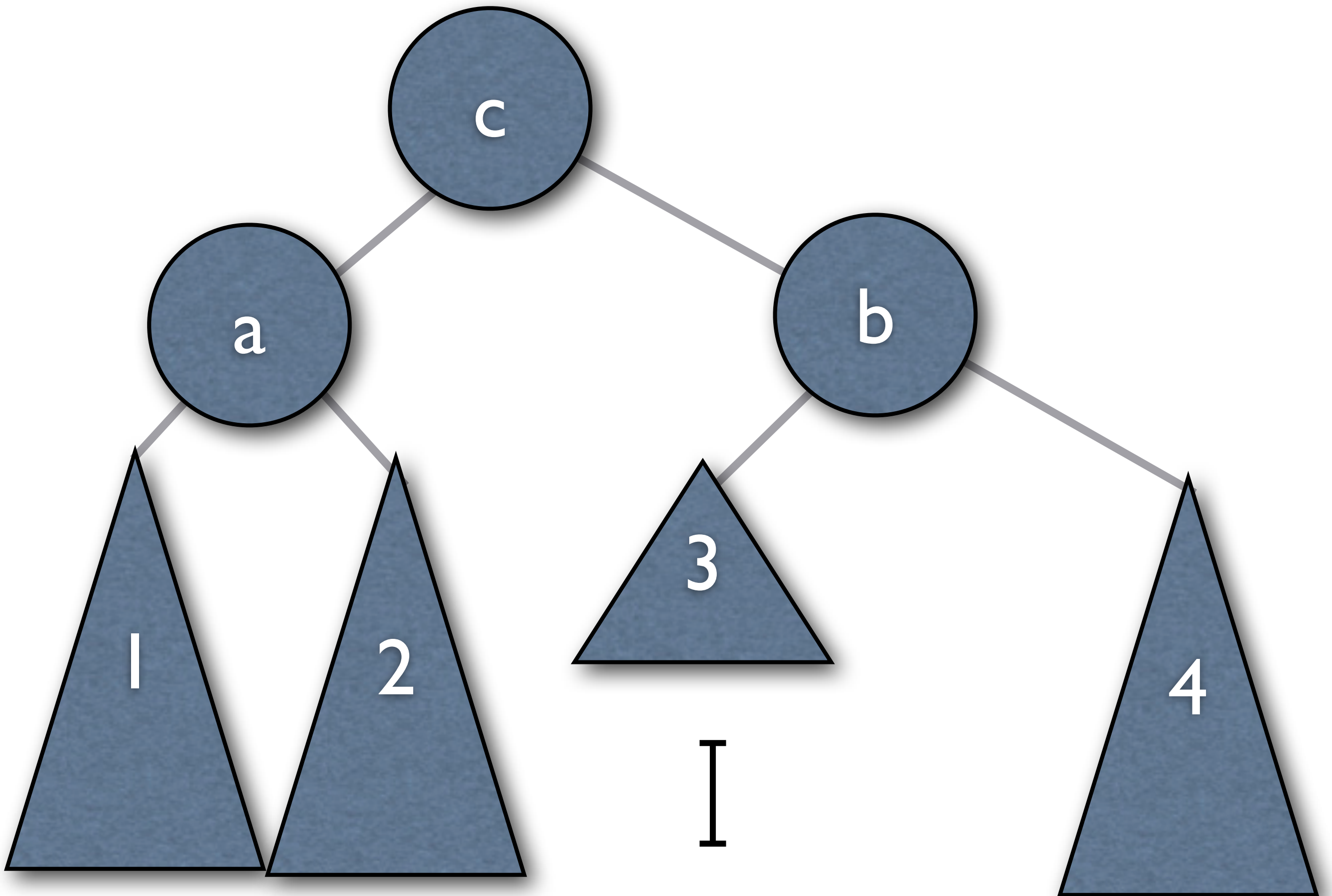
AVL Tree Visual: After Insert 2



AVL Tree Visual: Double Rotation



AVL Tree Visual: Double Rotation



Rotation running time

- Constant number of link rearrangements
- Double rotation needs twice as many, but still constant
  - So AVL rotations do not change O(d) running time of all BST operations*
- * remove() can require up to O(d) rotations; use lazy deletion

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

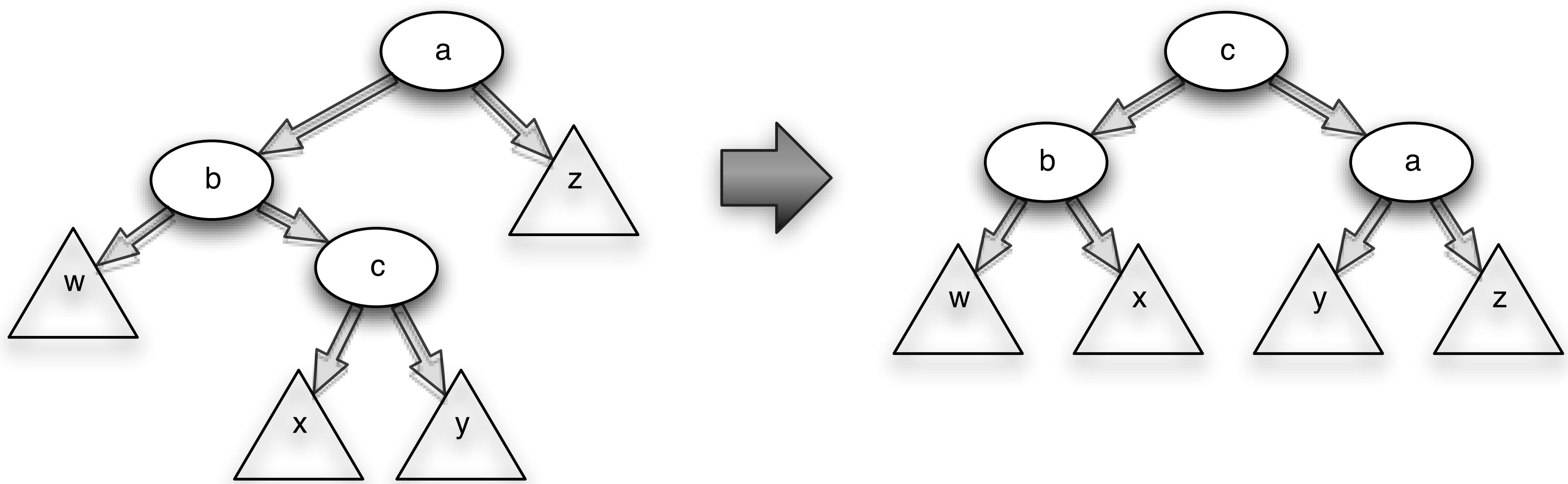
- Similar to *caching*.
  - Heuristically, data that is accessed tends to be accessed often.
- Easier to implement than AVL trees
  - No height bookkeeping

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

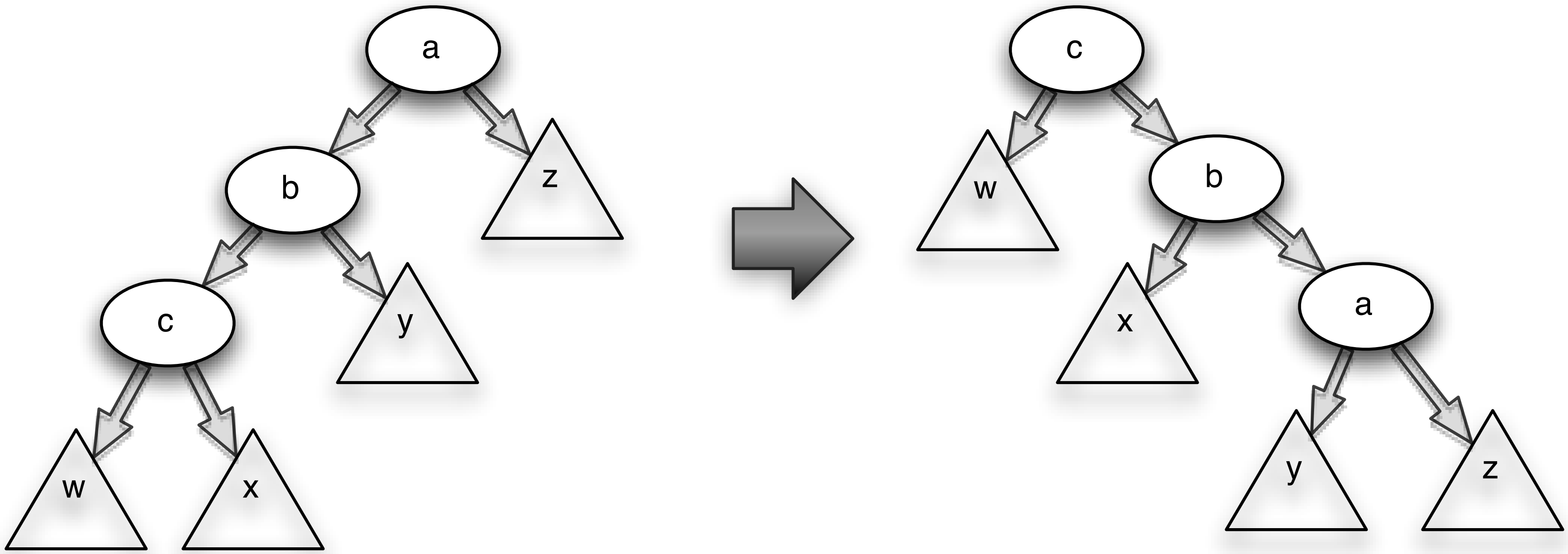


- Use when the node is the right child of a left child (or left-right)
- Double rotate, just like AVL tree

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

Splay Analysis (Informal)

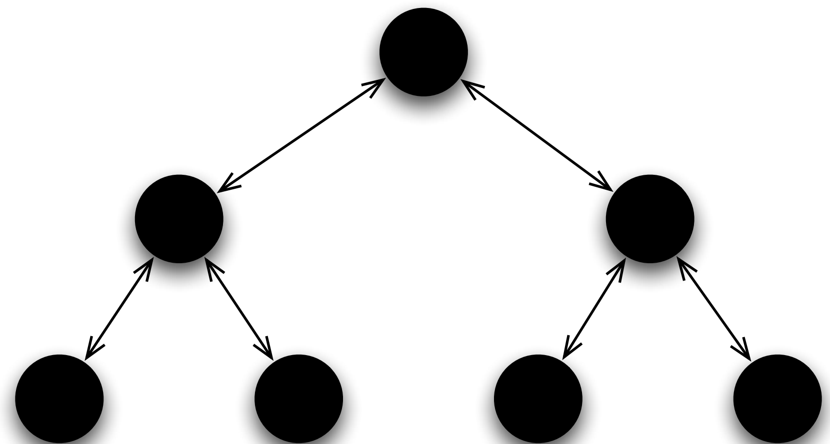
- We can make a chain by inserting nodes that make the tree its left child
  - Each of these operations is cheap
- Then we can search for deepest node
  - Splay operation squishes the tree;
    can only bad operations once before they become cheap
- *M* operations take $O(M \log N)$, so amortized $O(\log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

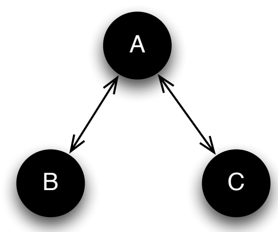
Priority Queues

- 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





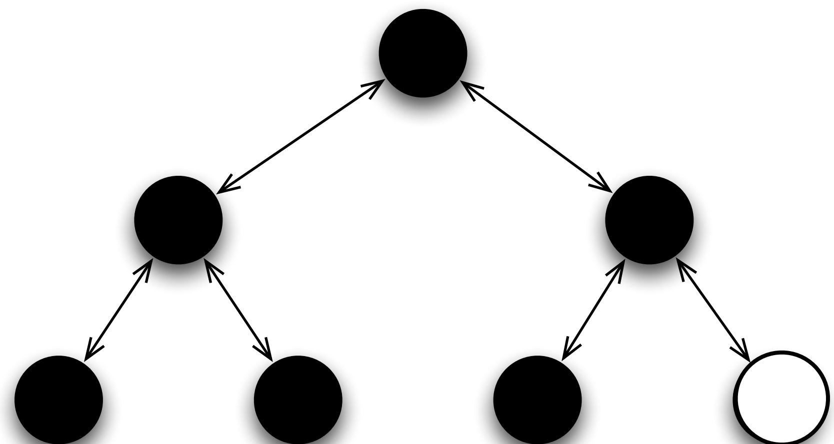
$$A < B$$
$$A < C$$
$$C \; ? \; B$$

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)

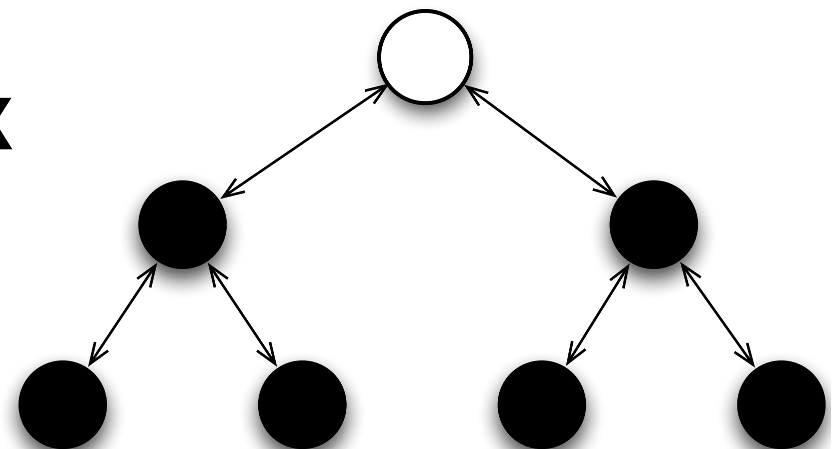
Insert

- 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



Changing a key

- Assuming you allow direct access to elements in heap
- decreaseKey: lower key, percolate up
- increaseKey: raise key, percolate down

Running times

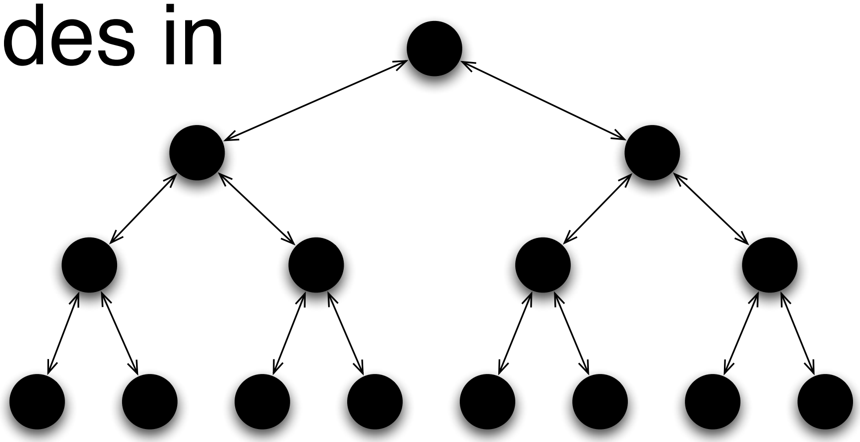
- Insert/deleteMin O(log N)
- findMin O(1)
- Where's the big gain?
  - buildHeap: given N items, creates a heap in linear time

Building a Heap from an Array

- How do we construct a binary heap from an array?
- Simple solution: insert each entry one at a time
- Each insert is worst case **O(log N)**, so creating a heap in this way is **O(N log N)**
- Instead, we can jam the entries into a full binary tree and run **percolateDown** intelligently

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)