CS 261
Data Structures

Lecture 15

AVL Trees

7/20/22, Wednesday

Oregon State University

Lecture Topics:

- AVL Trees

The Balance of BSTs

- Balance of BSTs:
  - All nodes have depths approximately log(n) or less

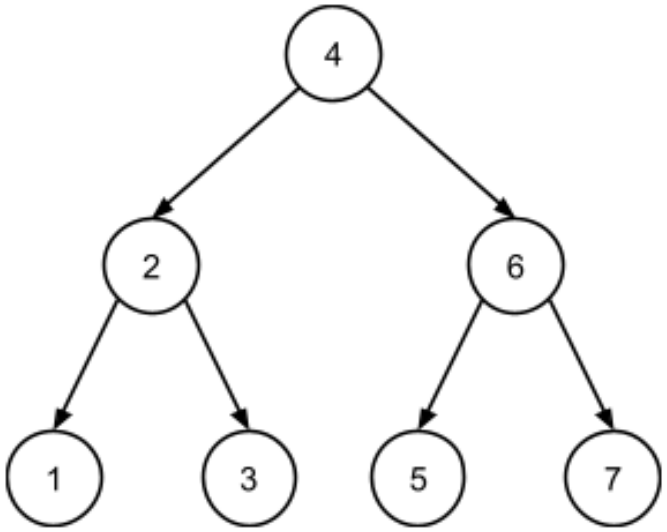
- Balance is important – primary operations on BSTs all have O(h) runtime complexity, where h is the height of the tree.

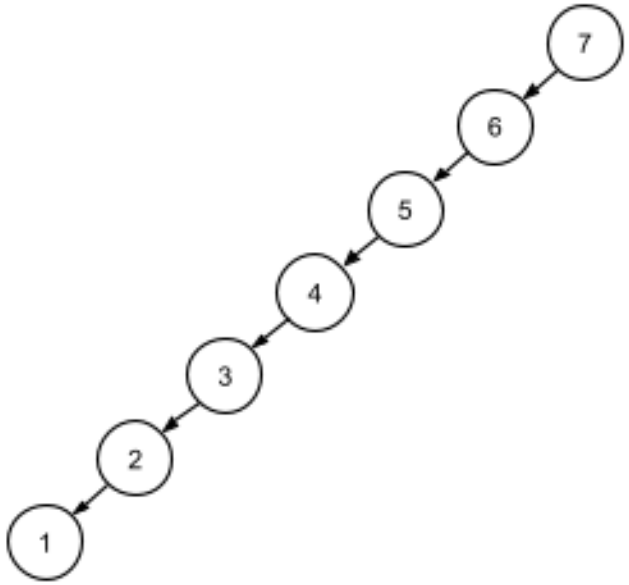
- With balanced BST, h → log(n), then O(h) will be fast
- With unbalanced BST, h → n, then O(h) will be slow

- Problem: plain BSTs cannot ensure itself is balanced

The Balance of BSTs

- Exercise:
  - Create a BST by inserting the elements with the following keys (order as they are):
    - 4 2 6 1 3 5 7
    - 7 6 5 4 3 2 1

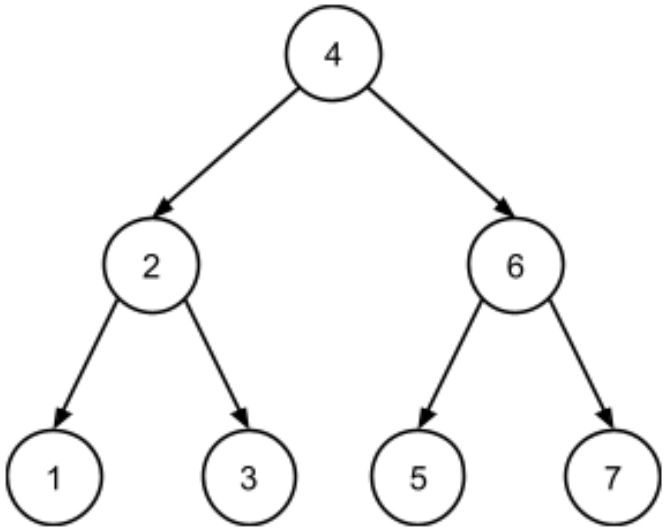


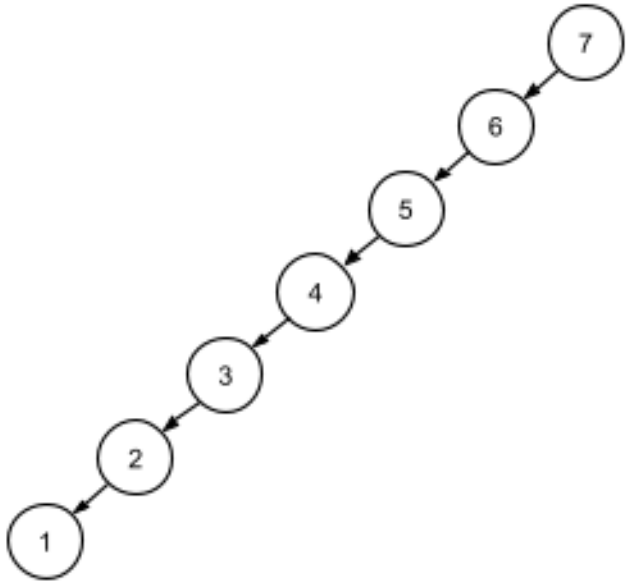


  - What do you notice?

The Balance of BSTs

- For a given set of keys, the shape of a BST depends on the order in which those keys are inserted into the tree.
  - Left: perfectly balanced, operations runtime close to O (log n)
  - Right: very unbalanced, operations runtime close to O (n)





The Balance of BSTs

- *Self-balancing BST*: does “extra work” to ensure that the tree is more-or-less balanced as elements are inserted and removed.
  - *Extra work – beyond that done by a plain BST

- A typical type of self-balancing BST known as an ***AVL tree***

Height Balance

- *Height Balance*: a measurable form of BST balance

- A BST is height balanced if, at every node in the tree, the subtree heights of the node's left and right subtrees differ by at most 1

- A height-balanced BST is guaranteed to have an overall height that's within a constant factor of log(n)
  - operations in a height-balanced BST are guaranteed to have *O(log n)* runtime complexity.

Balance Factor

- A BST node's *balance factor* – a metric to figure out whether the subtree rooted at that node is height balanced.

- the balance factor of the node N:
  - **`balanceFactor(N) = height(N.right) - height(N.left)`**

  - the height of a NULL node (i.e. an empty subtree) is -1

Balance Factor

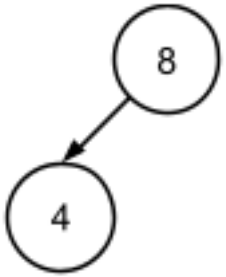
- An entire BST is *height balanced* if every node in the tree has a balance factor of -1, 0, or 1

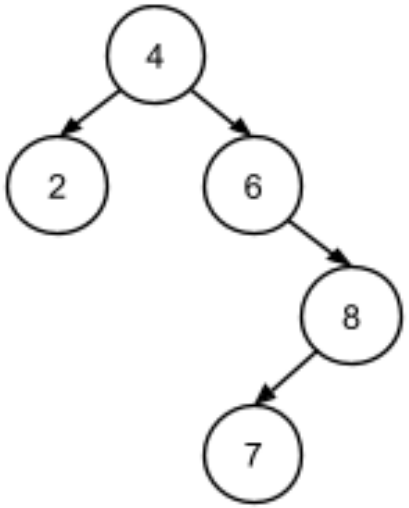
- If a node has a negative balance factor (i.e. balanceFactor(N) < 0), we call it *left-heavy*

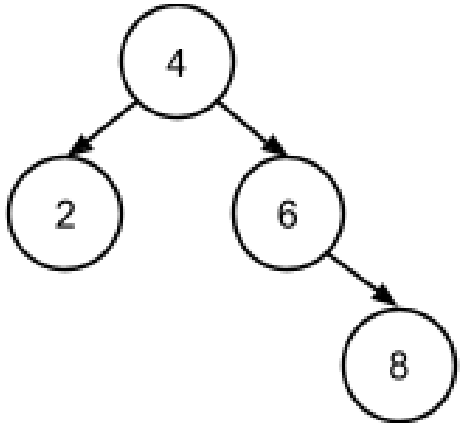
- If a node has a positive balance factor (i.e. balanceFactor(N) > 0), we call it *right-heavy*

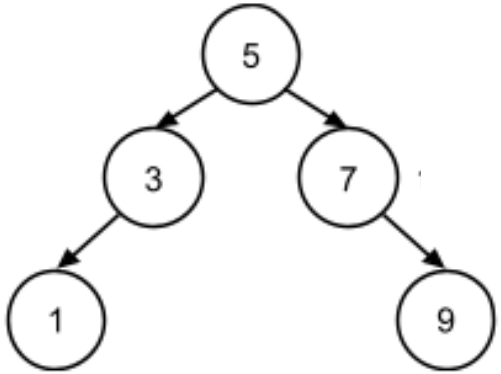
Height Balance and Balance Factor

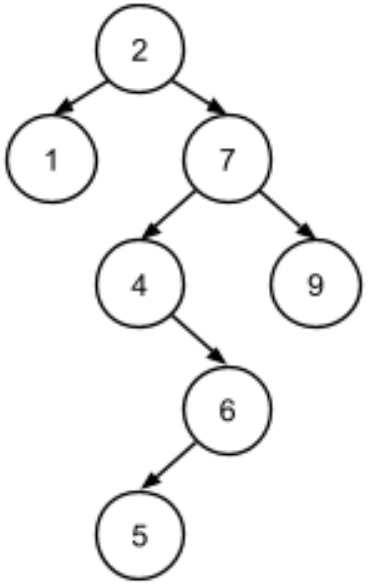
- Height-balanced, or un-balanced? Write down balance factor for each node.











Restructuring AVL Trees via Rotations

- The AVL tree is one of several existing types of self-balancing BST.
  - AVL is derived from the initials of the names of the tree's inventors: Adelson-Velsky and Landis.
  - Another popular one is the red-black tree.

- An AVL tree's operations include mechanisms to ensure that the tree always exhibits height balance
  - check the height balance of the tree after each insertion and removal
  - perform rebalancing operations known as *rotations* whenever height balance is lost

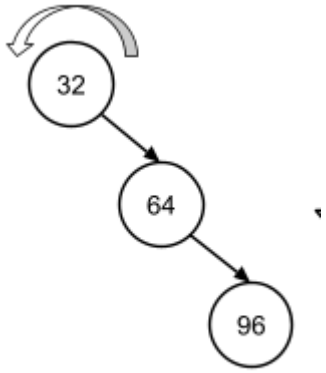
Restructuring AVL Trees via Rotations

- A rotation: an operation that restructures an isolated region of the tree by performing a limited number of *pointer updates* that result in one node moving "upwards" in the tree and another node moving "downwards."
  - preserve the BST property among all nodes in the tree

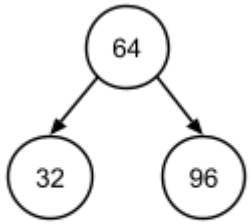
- Sometimes, a single rotation will be enough to restore height balance.
- Sometimes, a double rotation will be needed.

Restructuring AVL Trees via Rotations

- Each rotation has a center and a direction
- The center is the node at which the rotation is performed
- Direction: perform either a left rotation or a right rotation around this center node
  - A left rotation moves nodes in a “counterclockwise” direction, with the center moving downwards and nodes to its right moving upwards.
  - A right rotation moves nodes in a “clockwise” direction, with the center moving downwards and nodes to its left moving upwards



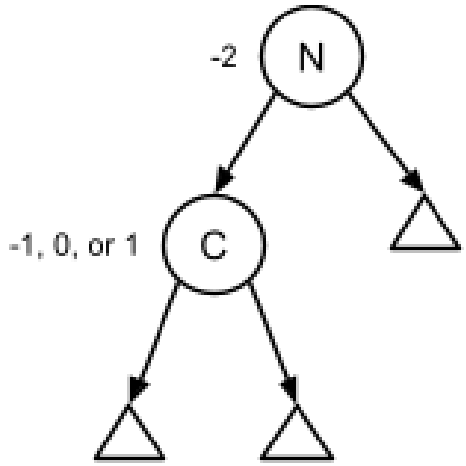
Before rotation



After rotation

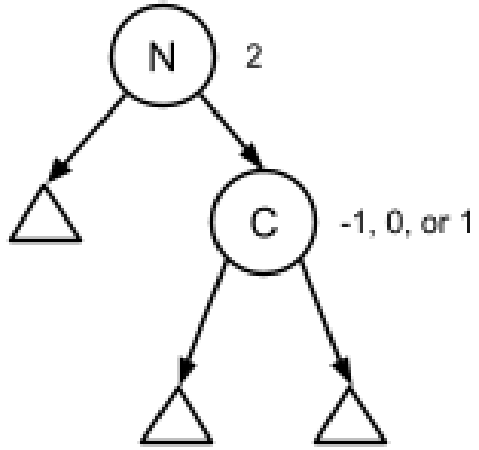
How to rotate?

- A rotation (i.e. single or double) will be needed any time an insertion into or removal from an AVL tree that leaves the tree (temporarily) with a node whose balance factor is either -2 or 2
- In other words, a rotation is needed when height balance is lost at a specific node in the tree. Let's call this node N.



N left-heavy

- If N has a balance factor of -2, this means N is left-heavy.
- If N has a balance factor of 2, this means N is right-heavy.
- Regardless of the direction of N's heaviness, let's refer to the heavier of N's children as C
- The node C itself will have a balance factor of -1, 0, or 1



N right-heavy

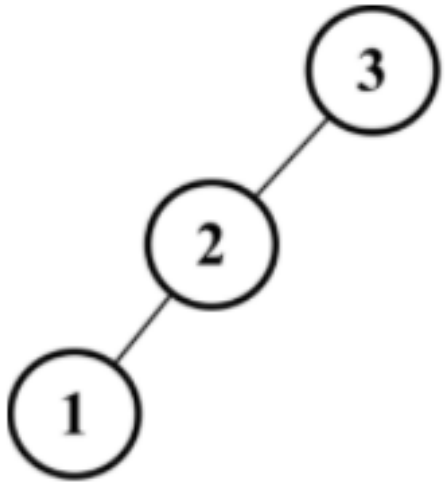
In class activity: How to rotate?

- Get into small groups, on the worksheet, for each unbalanced tree,
  - Determine whether a single rotation / a double rotation is needed
  - draw the height-balanced BSTs after rotating

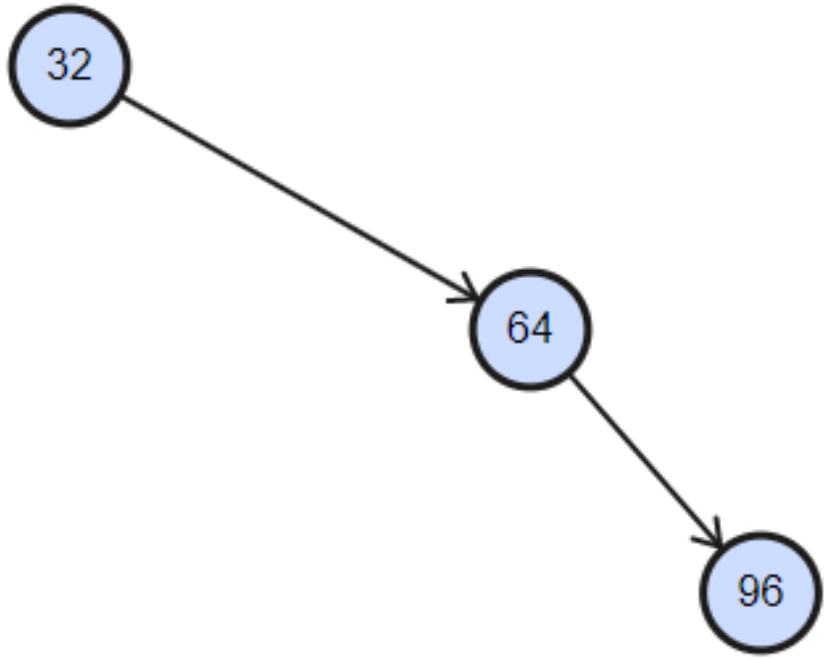
- Can you generalize the situations when a single rotation is needed?

- Can you generalize the situations when a double rotation is needed?

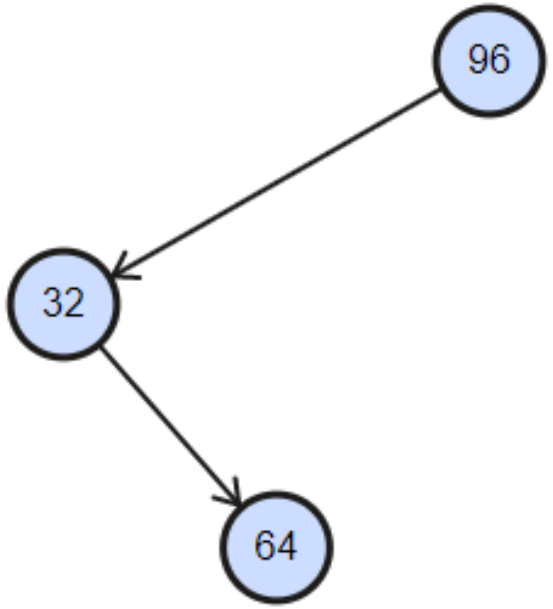
In class activity: How to rotate?



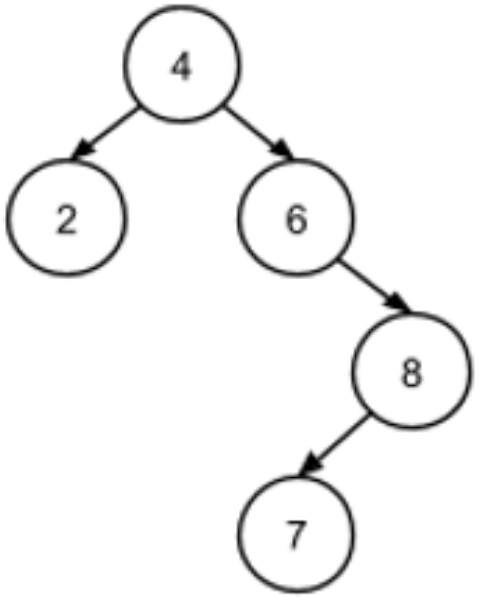
In class activity: How to rotate?



In class activity: How to rotate?

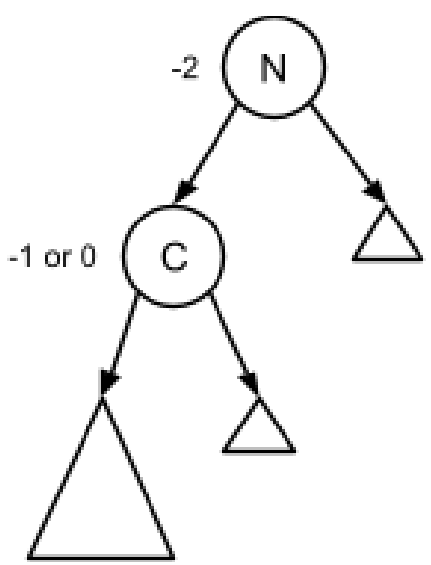


In class activity: How to rotate?

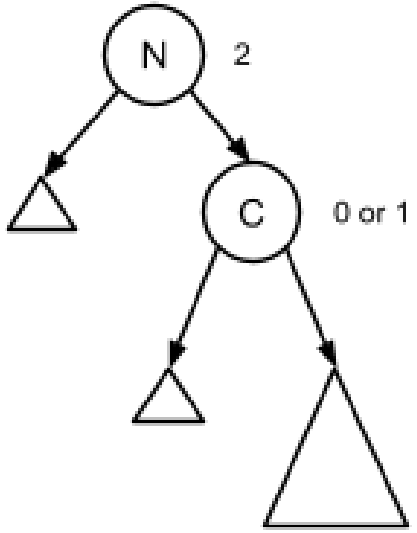


Single vs. Double Rotation

- If ***N* and *C* are heavy in the same direction**, then a single rotation is needed around N in the opposite direction as N's heaviness



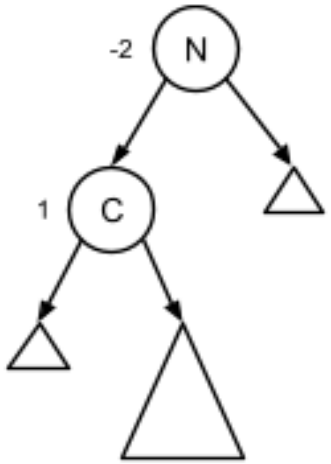
Single right rotation needed

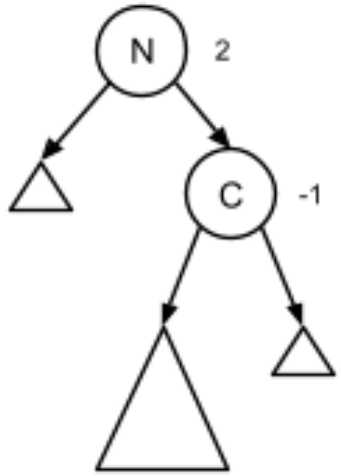


Single left rotation needed

Single vs. Double Rotation

- If ***N* and *C* are heavy in opposite directions**, then a double rotation is needed
  - If N is left-heavy and C is right-heavy, then we first rotate left around C then right around N.
  - If N is right-heavy and C is left-heavy, then we first rotate right around C then left around N





Single vs. Double Rotation

		`balanceFactor(N)`	
		-2 (left-heavy)	2 (right-heavy)
`balanceFactor(C)`	-1 (left-heavy)	**Left-left imbalance** Single rotation: right around *N*	**Right-left imbalance** Double rotation: 1. right around *C* 2. left around *N*
	0		
	1 (right-heavy)	**Left-right imbalance** Double rotation: 1. left around *C* 2. right around *N*	**Right-right imbalance** Single rotation: left around *N*