AVLs, 2-4 Trees

1332 Recitation: Week of 2020-06-09

Announcement

- Homework 4 (HashMap) is due tonight at 11:55 PM EDT
- Homework 5 (AVL) has been released and is due Monday, June 15th.
-

AVLs

- AVLs are a type of BST that do automatic self-balancing.
  - It prevents a BST from becoming a degenerate tree, which basically looks like a linked list.
- But how?
  - Problem of regular BST: recall the worst case for all BST operations, which is O(n)
  - Solution: restructure the tree after add() or remove() to ensure that it is balance (has O(log(n)) height)

Structure/ Properties

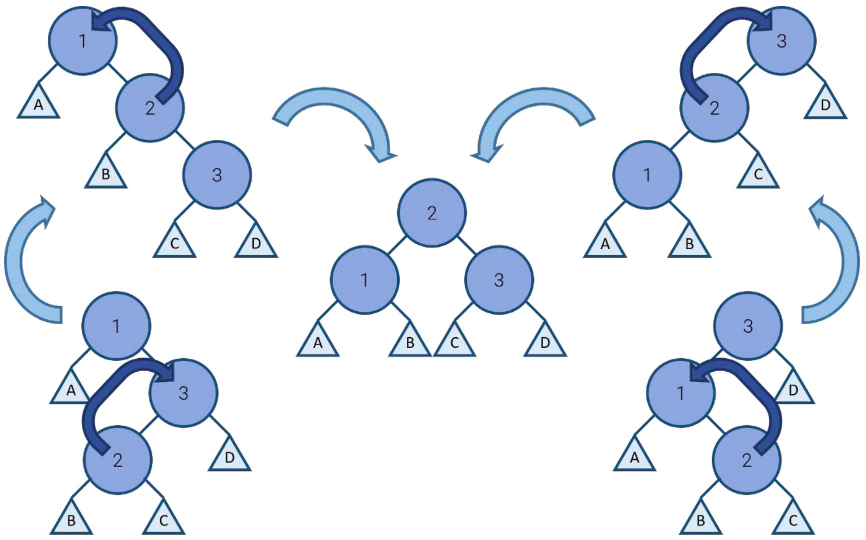
- Each node stores five pieces of information
    - Left and right child pointers
    - Data
    - Height
        - Same definition as BST: height = max(left child’s height, right child’s height) + 1
        - Height of null = -1
        - Height of leaves = 0
    - Balance Factor (BF) - this forms a more rigid definition for balance
        - The difference between the children’s heights
        - node.balanceFactor = node.left.height - node.right.height
        - In an AVL tree, the BF will always be between -1 and 1, this is why AVL trees are always balanced
        - During an add() or remove(), if some node has an out of bounds balance factor (<-1 or >1) then we rotate about this node using pointer reinforcement (hopefully) to fix its balance factor

Rotations

- Now what do we do if the balance factor is < -1 of > 1?
  - We rotate!
- Negative BF means that node is right-heavy and positive BF means it is left-heavy, this relates back to our formula
- Sometimes we also want to make decision about rotation's based on a node's child
  - E.g. if we know a node is right heavy, but we also know its right child is left heavy, we have a zig-zag structure and want to perform a double-rotation
- Why wouldn't a single rotation work?

Rotation

Parent BF	Left Child BF	Right Child BF	Rotation
-2	–	-1, 0	Left
-2	–	1	Right-left
2	-1	–	Left-right
2	0, 1	–	Right

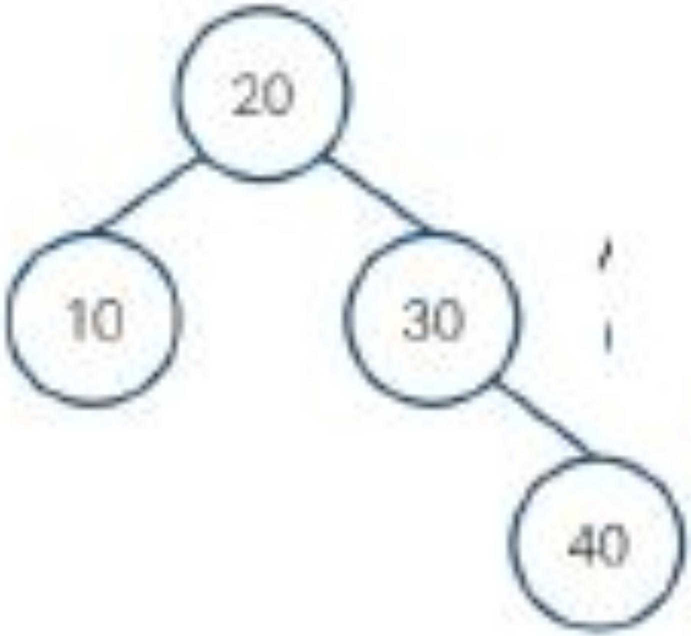


Adding

- Traverse tree and add node just like a BST (use pointer reinforcement)
- As you “unroll” the recursion and traverse back up the tree
  - Update the height and balance factor
  - If the node has balance factor outside bounds (< -1 or > 1), apply appropriate rotation

Example

- Add 50
- Add 1
- Add 4



Removing

- Traverse tree and remove node just like BST
- As you “unroll” the recursion and traverse back up the tree
  - Update the height and balance factor of each node
  - If node has balance factor outside bounds (< -1 or > 1), apply appropriate rotation
  - This includes going back up the tree after removing the predecessor/ successor node
- Example
  - Remove 4

Helpful Helper Methods

- Node updateAndBalance (Node node)
  - Should update the node's height and BF based on the info stored in the children
  - If unbalanced, perform rotations by calling methods below
- Node rightRotate (Node node)
  - In addition to performing the rotation, should also update heights and BFs by calling the above method
- Node leftRotate (Node node)
  - Same as rightRotate
- int getHeight (Node node)
  - Return the height stored in the node or -1 if node is null
  - May seem redundant, but prevents NullPointerException's and simplifies getting children's heights

Efficiencies

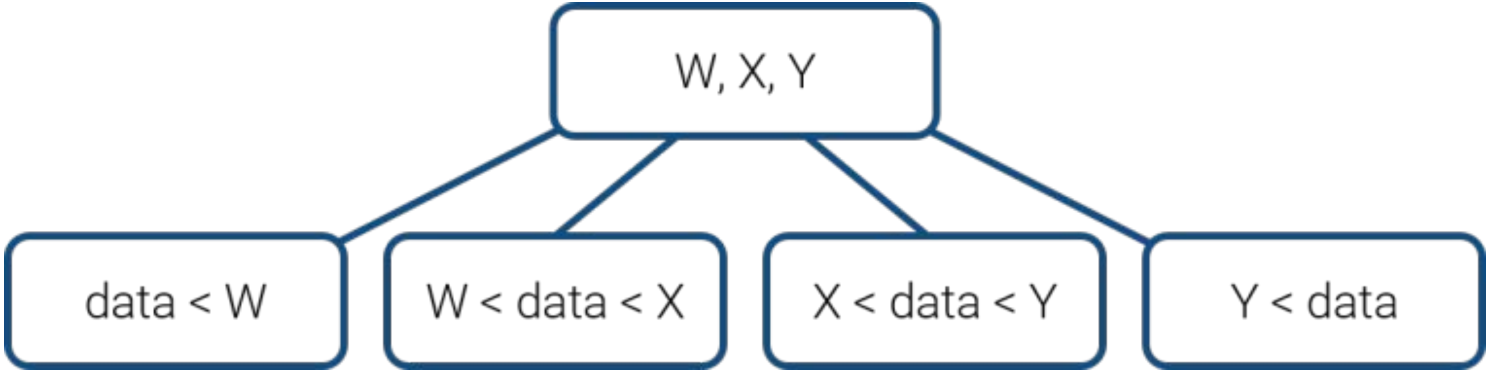
	Adding	Removing	Accessing
Average	O(log n)	O(log n)	O(log n)
Worst	O(log n)	O(log n)	O(log n)

Things to note for HW

1. Fix BST HW based on feedback once it’s released
2. Remember that each node contains its height! DO NOT calculate height recursively
3. Don’t use children's BF to calculate BF instead of using height
4. Update height first then BF

2-4 Trees

- A tree with multiple data in each node and multiple children
- Each node has:
  - Multiple data items between 1 and 3
  - (# of data + 1) children, which could be between 2 and 4
- **Order property**: data is stored in ascending order from left to right
- **Shape properties**: Full, Complete, *and* all leaves are always the same depth
- Height of the tree is O(logn)



Adding

- Search for correct spot for the data using the properties above to find a leaf node
- Add the data to the leaf node
- If there are 4 data elements in the leaf node, then
  - Promote the second or third item to the parent (create a new root if necessary)
    - What to promote is an implementation detail and will be specified on exams
  - Split the node into items less than promoted data and items greater than promoted data
  - If promotion causes parent to have 4 data elements, repeat promotion on parent
- Example

Removing

- Basic steps:
    - Remove data element from the node
    - If removing it creates an empty node, we *handle underflow*
        - Try transfer: move data element from a neighboring sibling node with > 1 data
        - Try fusion: (only if transfer is not possible) pulling down data from parent and fusing with sibling
- Simple remove
- We will cover more complicated cases in our next recitation!

Efficiencies

	Adding	Removing	Accessing
Average	O(log n)	O(log n)	O(log n)
Worst	O(log n)	O(log n)	O(log n)