CSE 332: Data Structures and Parallelism

Fall 2022
Richard Anderson
Lecture 8: Адельсо́н-Ве́льский Ла́ндис деревья, Часть вторая,

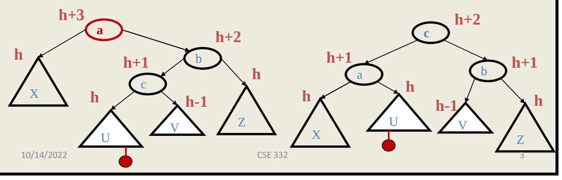
Lecture 8: AVL Trees, part two

Announcements

- Project 2, available now (?)
  - Checkpoint 1, Oct 23
    - MinFourHeapComparable, MoveToFrontList
  - Checkpoint 2, Nov 3
  - Deadline, Nov 10

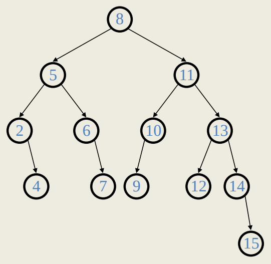
AVL Tree overview

- Balance condition
- Depth bound
- Rotations to rebalance the tree



The AVL Tree Data Structure

Structural properties
1. Binary tree property
2. Balance:
   left.height – right.height
3. Balance property:
   balance of every node is between -1 and 1
4. Tree of height h has at least ϕ^h nodes
5. Worst-case depth is O(log n)



AVL insert:

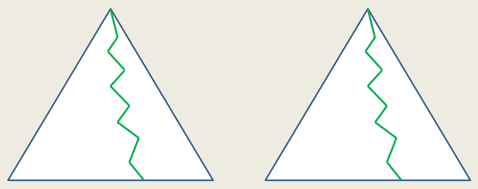
First BST **insert**, *then* check balance and potentially "fix" the AVL tree

Four different imbalance cases

AVL tree operations

- **AVL `find`**:
  - Same as BST **`find`**
- **AVL `insert`**:
  - First BST **`insert`**, *then* check balance and potentially "fix" the AVL tree
  - Four different imbalance cases
- **AVL `delete`**:
  - The "easy way" is lazy deletion
  - Otherwise, do the deletion and then have several imbalance cases (

AVL Tree Insert: High level idea



Insert new leaf, follow path back to root computing heights and balance factors

Find first unbalanced node

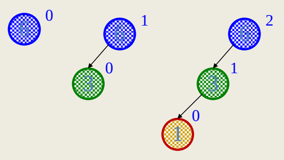
If there is an unbalanced node, apply a double rotation to fix it up

Case #1: Example

Insert(6)
Insert(3)
Insert(1)

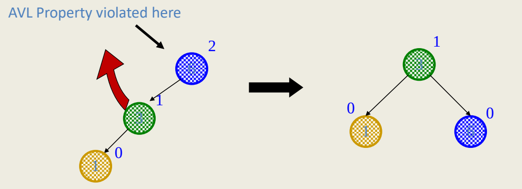
Third insertion violates balance property
- happens to be at the root

What is the only way to fix this?



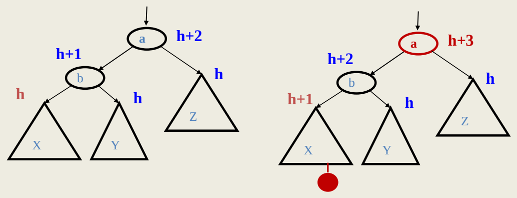
Fix: Apply "Single Rotation"

- *Single rotation:* The basic operation we'll use to rebalance
  - Move child of unbalanced node into parent position
  - Parent becomes the "other" child (always okay in a BST!)
  - Other subtrees move in only way BST allows (next slide)



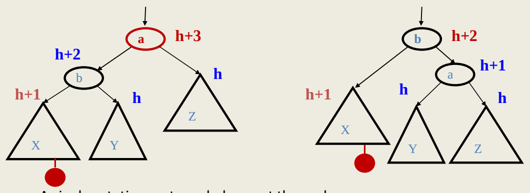
Left-left rebalancing

- Node imbalanced due to insertion *somewhere* in **left-left grandchild** increasing height
  - 1 of 4 possible imbalance causes (other three coming)
- First we did the insertion, which would make ***a*** imbalanced



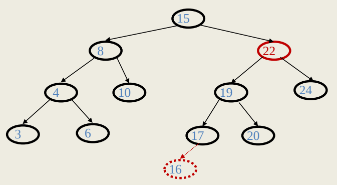
Left-left case

- Node imbalanced due to insertion *somewhere* in **left-left grandchild**
  - 1 of 4 possible imbalance causes (other three coming)
- So we rotate at ***a*,** using BST facts: X < b < Y < a < Z

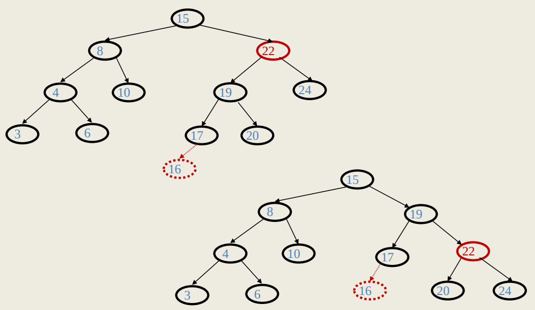


- A single rotation restores balance at the node
  - To same height as before insertion, so ancestors now balanced

Another example: **insert(16)**

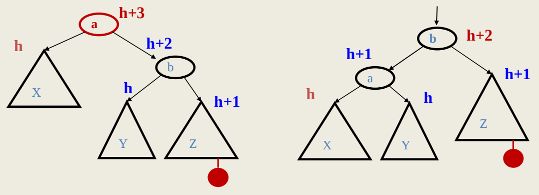


Another example: **insert(16)**



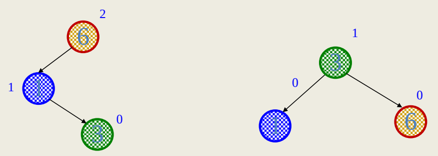
The right-right case

- Mirror image to left-left case, so you rotate the other way
  - Exact same concept, but need different code



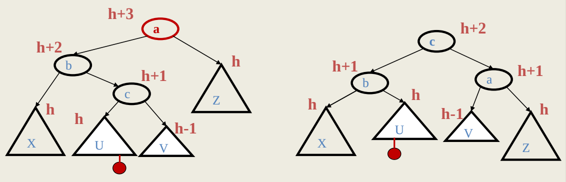
Two cases to go

Simple example: **`insert`**(6), **`insert`**(1), **`insert`**(3)



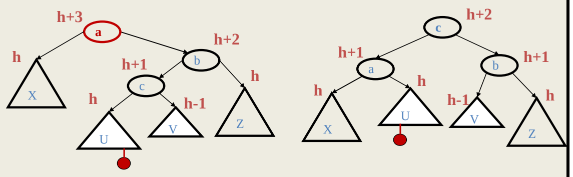
The last case: left-right

- Left-right grandchild promoted



Right-left case

- Mirror image to left-right case, so you rotate the other way
  - Exact same concept, but need different code



Insert, summarized

- Insert as in a BST
- Check back up path for imbalance, which will be 1 of 4 cases:
  - Node's left-left grandchild is too tall
  - Node's left-right grandchild is too tall
  - Node's right-left grandchild is too tall
  - Node's right-right grandchild is too tall
- Only one case occurs because tree was balanced before insert
- After the appropriate single or double rotation, the smallest-unbalanced subtree has the same height as before the insertion
  - So all ancestors are now balanced

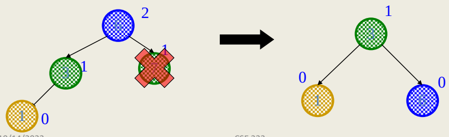
Efficiency

- Worst-case complexity of **`find`**: $O($**`log`** $n)$
  - Tree is balanced
- Worst-case complexity of **`insert`**: $O($**`log`** $n)$
  - Tree starts balanced
  - A rotation is $O(1)$ and there's an $O($`log` $n)$ path to root
  - (Same complexity even without one-rotation-is-enough fact)
  - Tree ends balanced
- Worst-case complexity of **`buildTree`**: $O(n$ **`log`** $n)$

Will take some more rotation action to handle **`delete`**…

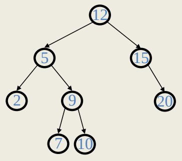
AVL Tree Deletion

- Similar to insertion: do the delete and then rebalance
  - Rotations and double rotations
  - Imbalance may propagate upward so rotations at multiple nodes along path to root may be needed (unlike with insert)
- Simple example: a deletion on the right causes the left-left grandchild to be too tall
  - Call this the *left-left case,* despite deletion on the *right*
  - `insert(6) insert(3) insert(7) insert(1) delete(7)`



Properties of BST delete

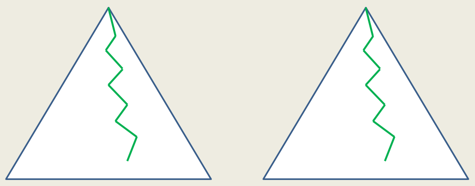
We first do the normal BST deletion:
- 0 children: just delete it
- 1 child: delete it, connect child to parent
- 2 children: put successor in your place, delete successor node



Which nodes' heights may have changed:
- 0 children: path from deleted node to root
- 1 child: path from deleted node to root
- 2 children: path from *deleted successor node* to root

Will rebalance as we return along the "path in question" to the root

AVL Tree Delete: High level idea



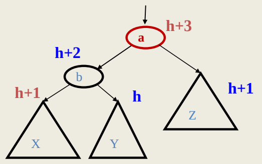
Delete the node and possibly replace it with its successor. Trace a path back from the node that was removed

Find first unbalanced node

If there is an unbalanced node, apply a double rotation to fix it up. Possibly continue up the tree and repeat

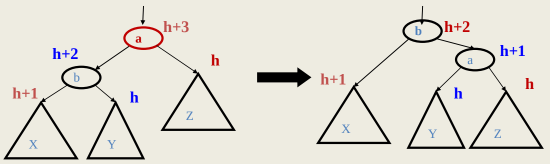
Case #1 Left-left due to right deletion

- Start with some subtree where if right child becomes shorter we are unbalanced due to height of left-left grandchild



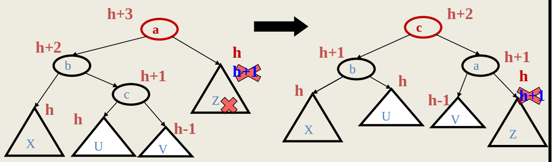
- A delete in the right child could cause this right-side shortening

Case #1: Left-left due to right deletion



- Same single rotation as when an insert in the left-left grandchild caused imbalance due to X becoming taller
- But here the "height" at the top decreases, so more rebalancing farther up the tree might still be necessary
- This case also applies when subtree y has height h+1, yielding a tree of height h+3, and no further rebalancing

Case #2: Left-right due to right deletion

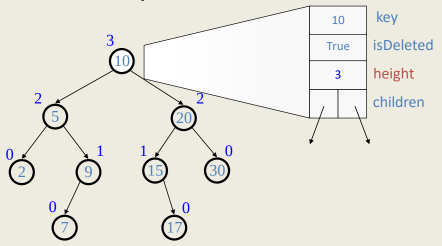


- Same double rotation when an insert in the left-right grandchild caused imbalance due to c becoming taller
- But here the "height" at the top decreases, so more rebalancing farther up the tree might still be necessary

And the other half

- Naturally two more mirror-image cases (not shown here)
  - Deletion in left causes right-right grandchild to be too tall
  - Deletion in left causes right-left grandchild to be too tall
  - (Deletion in left causes both right grandchildren to be too tall, in which case the right-right solution still works)
- And, remember, “lazy deletion” is a lot simpler and might suffice for your needs

Lazy Deletion

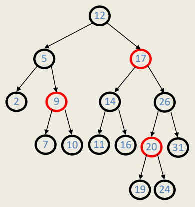


Lazy deletion

- General technique – just add a deleted flag
- Requires some additional logic in find/insert
- Increases amount of storage used
  - But usually this is not a problem
  - Possible to use with garbage collection
- Bad case for lazy deletion – if the number of “live” items is small because number of deletes is similar to the number of inserts

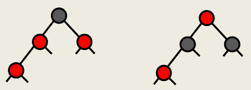
Red Black trees (optional)

- Binary search tree with rebalancing
  - Reasonable alternative to AVL trees
- O(log n) Find, Insert, Delete
- Nodes colored red or black
- Every root leaf path has the same number of black nodes
- Root is black
- No adjacent red nodes

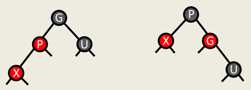


Inserting a node into a red black tree

- Insert at leaf and color red*
- If the parent is red, then fix up the tree with recoloring or rotation
- Repeat until coloring satisfies R-B rules
  - O(log n) steps



Recolor



Rotate

Other cases for RR, LR, RL

* Exception – insert at root is black