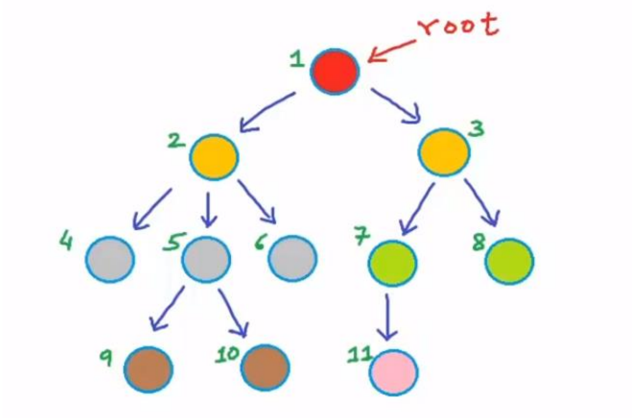
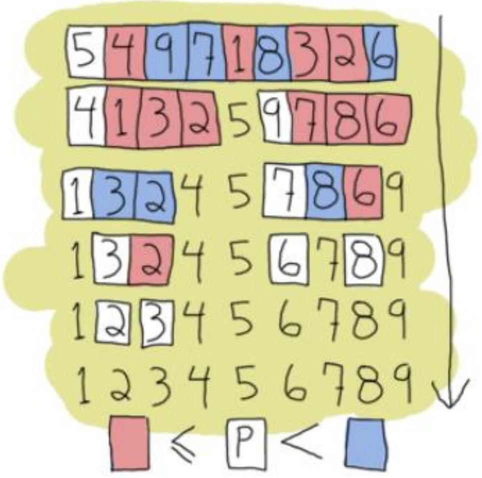
ECE 250 Data Structures & Algorithms



AVL Trees



Ziqiang Patrick Huang

Electrical and Computer Engineering

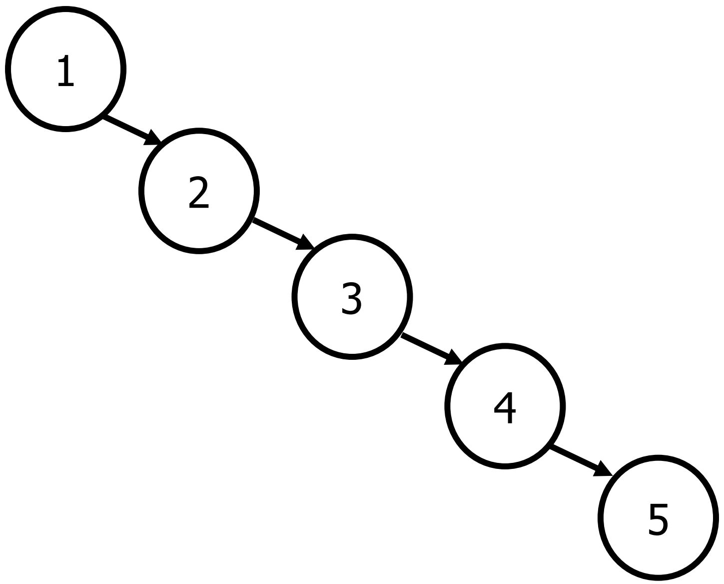
University of Waterloo

WATERLOO | ENGINEERING

Last Time: BSTs

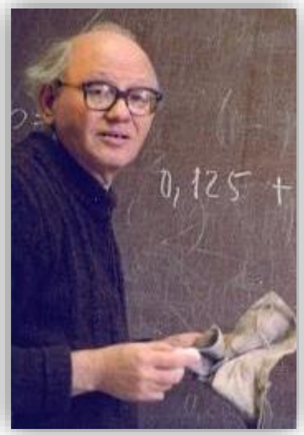
- Binary Search Trees
  - Idea: O(log(n)) access time (we hope)

- Can end up with degenerated BSTs
  - Example: add 1, 2, 3, 4, 5
  - O(N) access time
  - How likely are bad cases to come up? It depends …

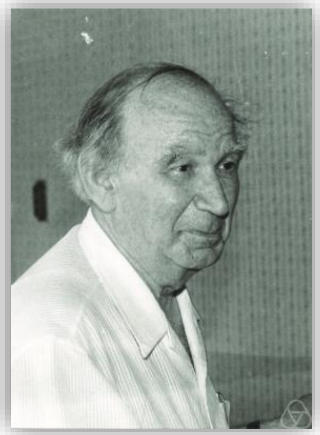


So what do we do about it?

- Two approaches (know both)
  - AVL (Adelson-Velskii and Landis) trees
    - Faster for lookup
  - Red-black trees
    - Faster for adding & removing

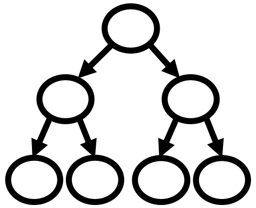


Georgy Adelson-Velsky



Evgenii Landis

- Will not guarantee “perfect tree” (very hard)
  - Perfect (binary) tree: all internal nodes have two children & all leaf nodes at same level
  - But will guarantee O(log(n))



AVL Trees

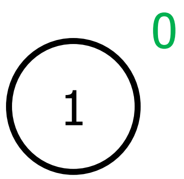
- AVL: A self-balancing BST used to eliminate O(n) worst case
  - Worst case for search/add/remove becomes O(log(n))

- AVL trees work on principle of balance
  - Height of two children(sub-trees) cannot differ by more than 1
    - Otherwise → "out-of-balance" or "imbalanced"
    - Difference of heights of two children → "balance factor"
  - Height(node) = max(height(node->left), height(node->right)) + 1
    - Base cases: height(leaf) = 0, Height(NULL) = -1
    - Some people define height(leaf) = 1, height(NULL) = 0
  - AVL = order property(from BST) + shape property

AVL Insertion

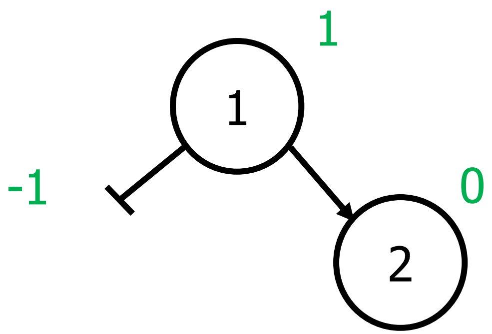
- Insertion starts as normal BST insertion
  - Using recursion
- After each recursive call returns:
  - Update the height information for each node
    - Stores heights to make calculations $O(1)$ as opposed to $O(n)$
  - Check for imbalance
    - If so, rotate the tree to fix

AVL Insertion Example



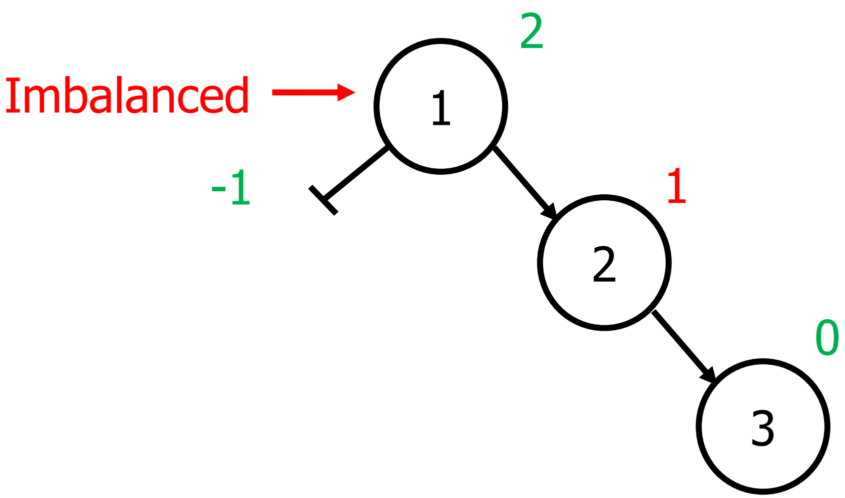
- Add 1 to empty tree
  - Heights are show next to nodes
    - Green = OK
    - Red = violating AVL rule
  - All good so far

AVL Insertion Example



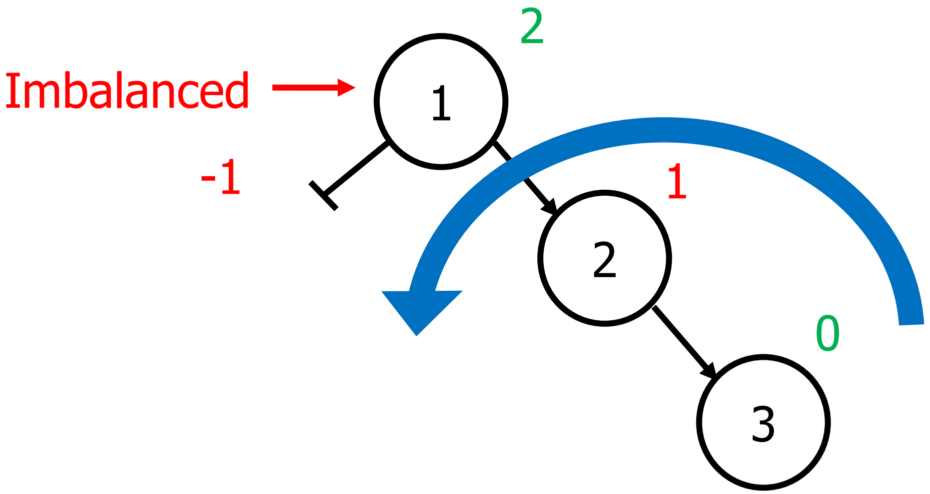
- Add 2 to tree
  - Children of 1 have height 0 and -1(NULL), differ by 1
  - Everything is still fine

AVL Insertion Example



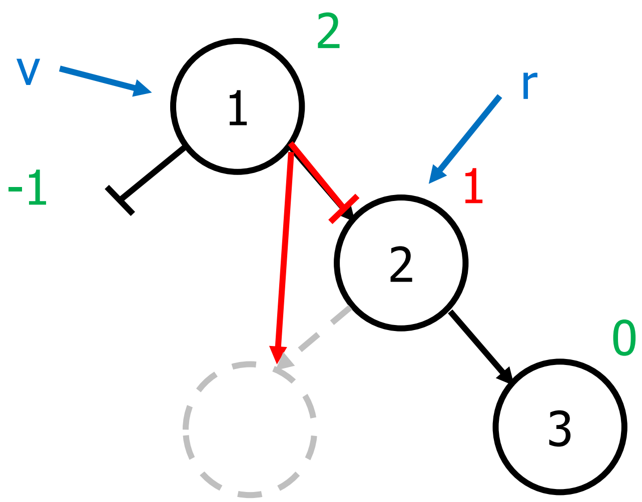
- Add 3 to tree
  - Now we have a problem at 1
    - Right child: height = 1
    - Left child: height = -1
    - Difference: 2

AVL Insertion Example



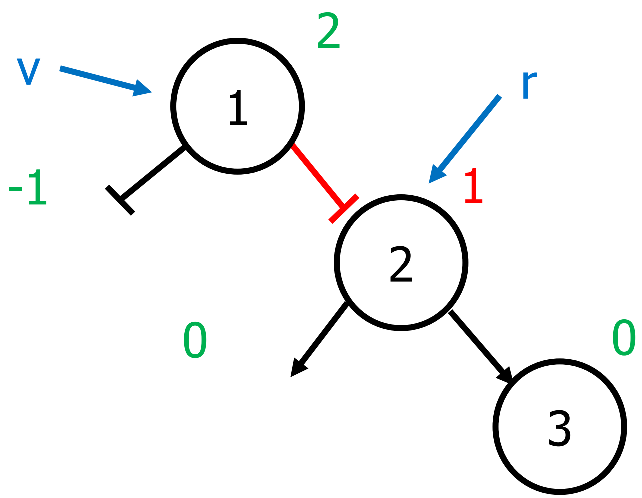
- Fix with single left rotation
  - Wait … what just happened?

AVL Insertion Example



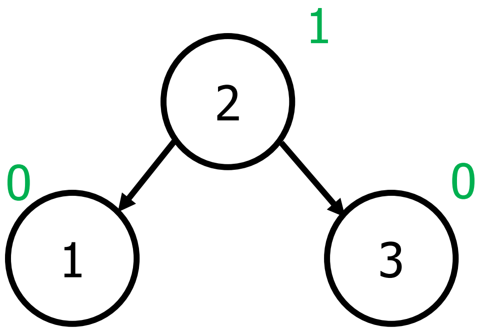
- Fix with single left rotation
  - v = violated node, r = v->right
  - Rotation step 1: v->right = NULL
    - More generally: v->right = r->left

AVL Insertion Example



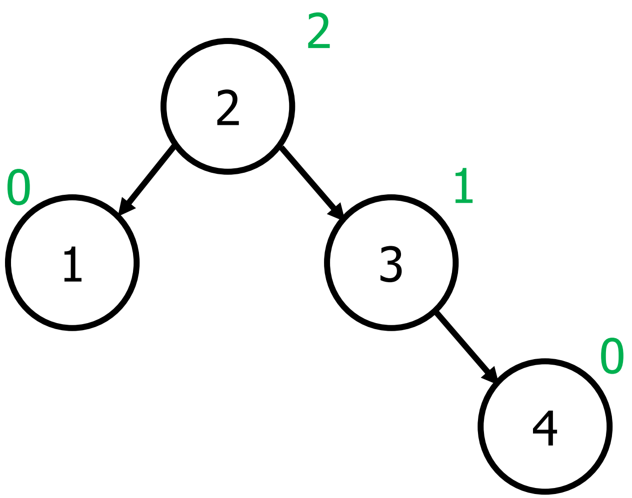
- Fix with single left rotation
  - v = violated node, r = v->right
  - Rotation step 1: v->right = r->left
  - Rotation step 2: r->left = v
  - How do we know this respects the BST rules?

AVL Insertion Example



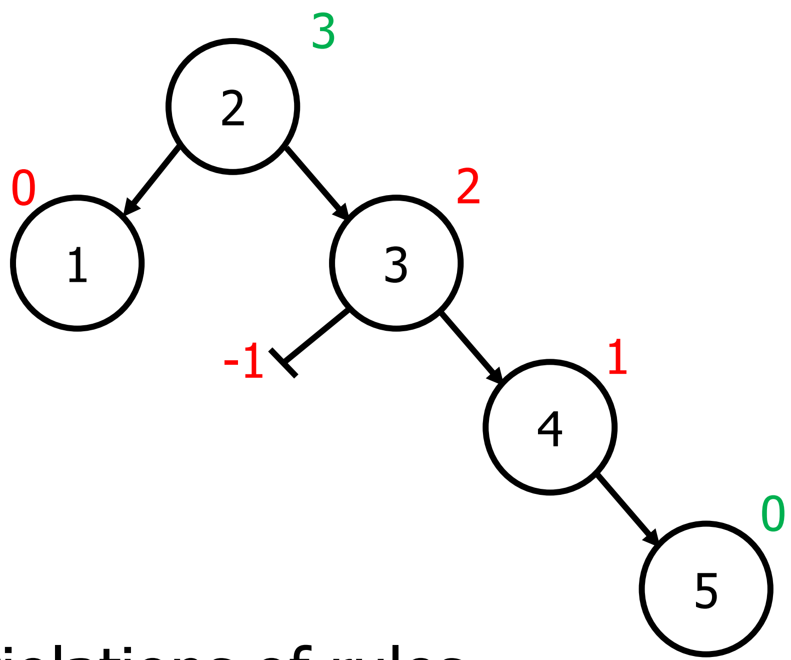
- Resulting tree respects AVL rules
  - Now let's add 4 and see what happens

AVL Insertion Example



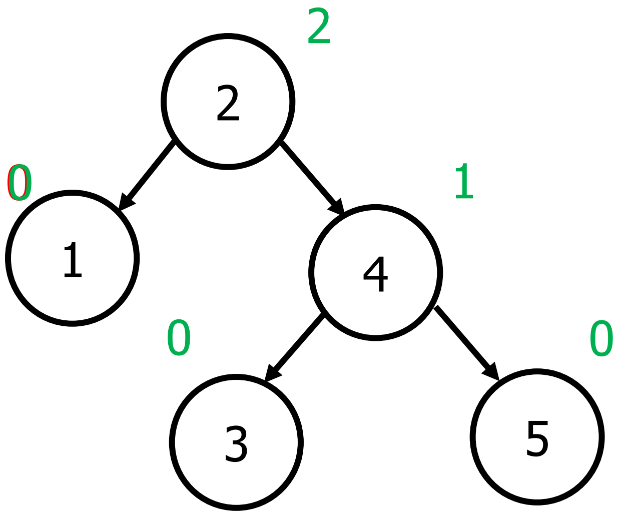
- Adding 4 works fine: no re-balance needed
  - Height difference at most 1 everywhere
  - Add 5?

AVL Insertion Example



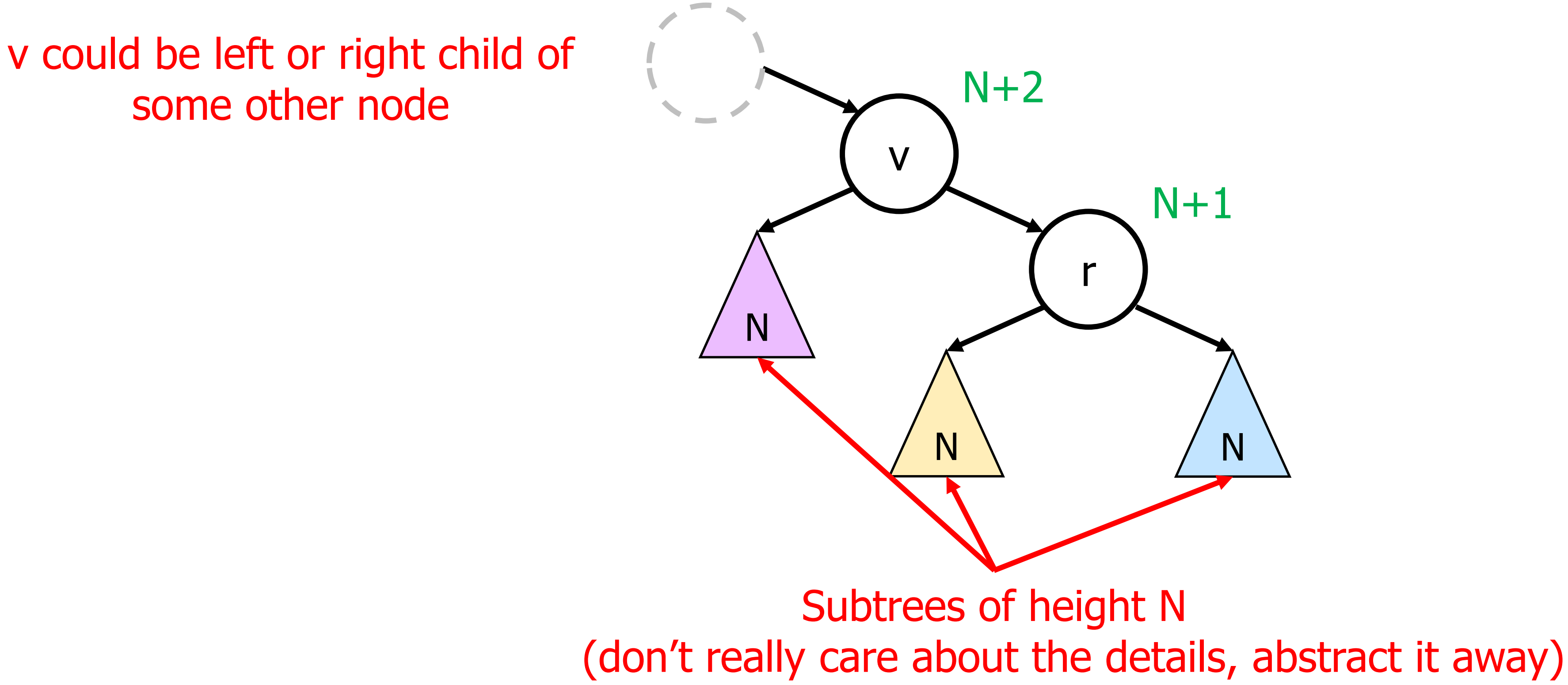
- Adding 5: looks like two violations of rules
  - First one at 3 : (-1 vs 1)
  - Second one at 2: (0 vs 2)
  - Reality: fix the one at 3, and everything is fine

AVL Insertion Example



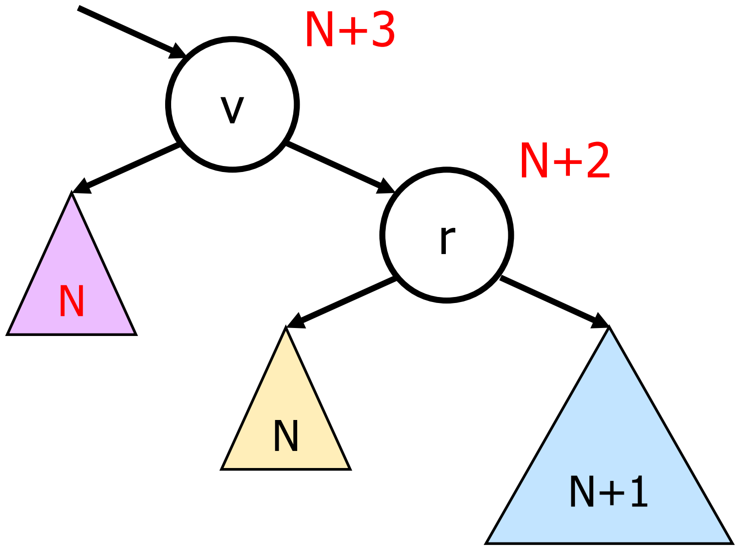
- Single left rotation at 3
  - 4 is the new root of that subtree

Generalizing AVL Insertion



- More generally
  - Start with something like this
    - Adding to the right side of r and increasing its height

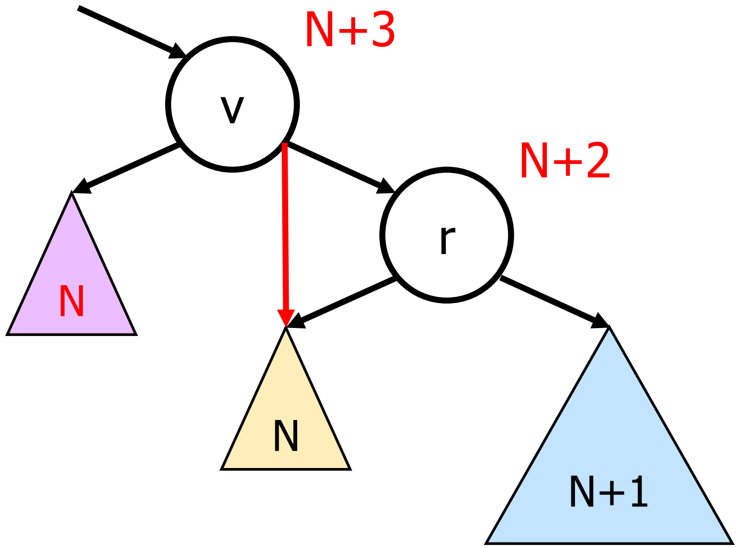
Generalizing AVL Insertion



- More generally
  - Start with something like this
    - Adding to the right side of r and increasing its height
    - This causes the violation

Generalizing AVL Insertion

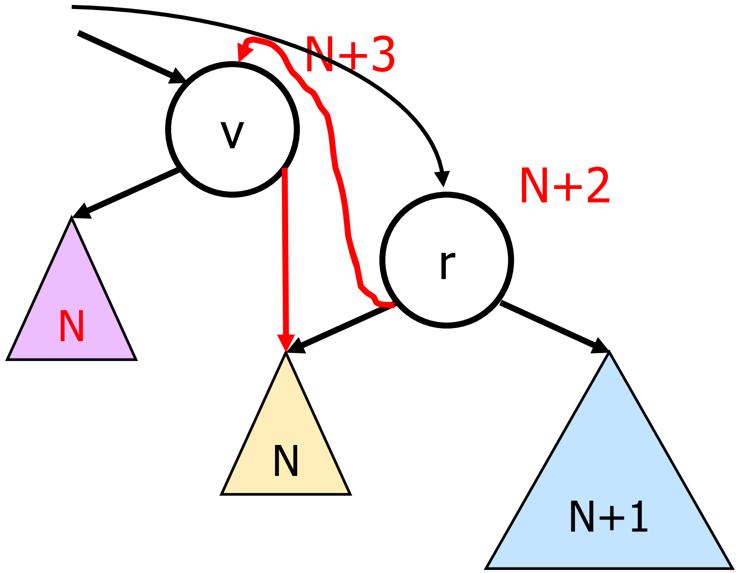
Rotate Left:
v->right = r->left



- More generally
  - Start with something like this
    - Adding to the right side of r and increasing its height
    - This causes the violation

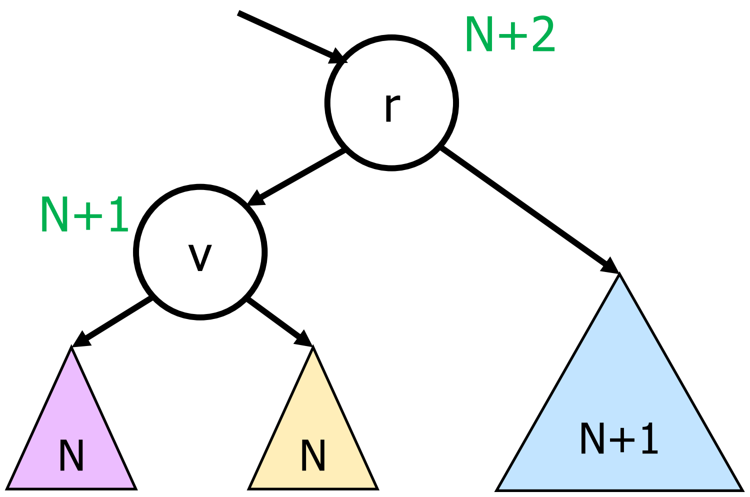
Generalizing AVL Insertion

Rotate Left:
v->right = r->left
r->left = v
r is root of subtree



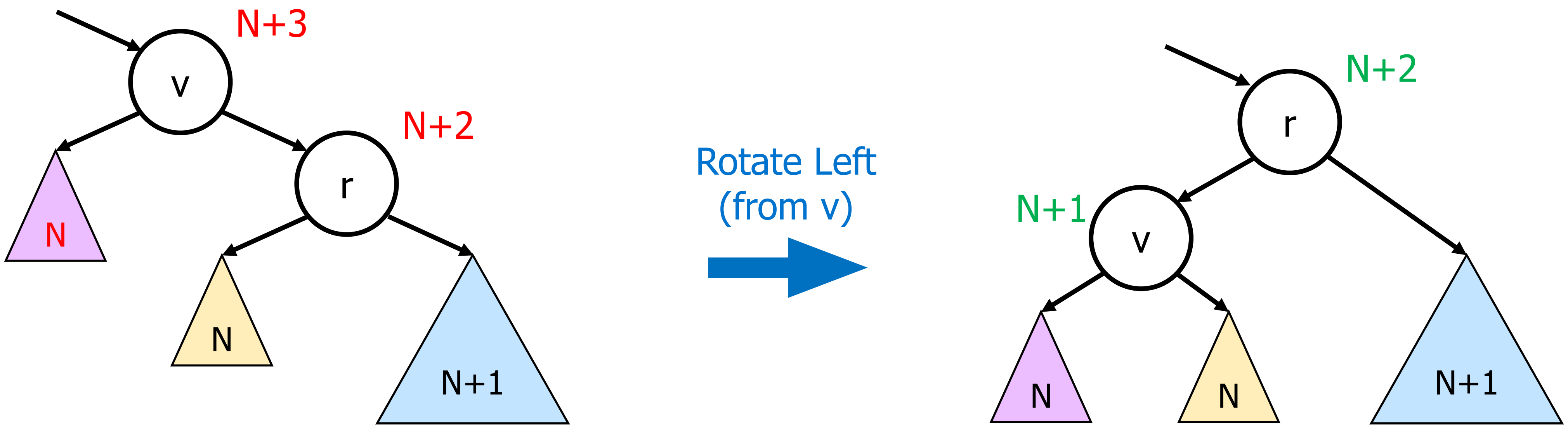
- More generally
  - Start with something like this
    - Adding to the right side of r and increasing its height
    - This causes the violation

Generalizing AVL Insertion



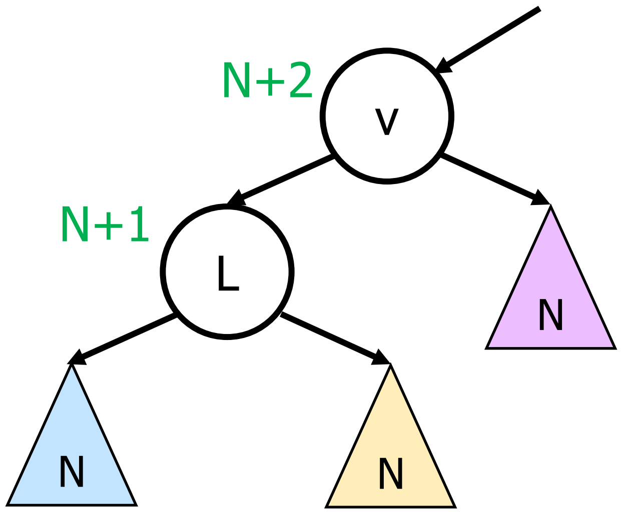
- More generally
  - Start with something like this
    - Adding to the right side of r and increasing its height
    - This causes the violation
  - Rotating fixes the violation

Generalizing AVL Insertion



- Summary
  - r moved up, v moved down
  - Restore the height of the subtree back to N+2

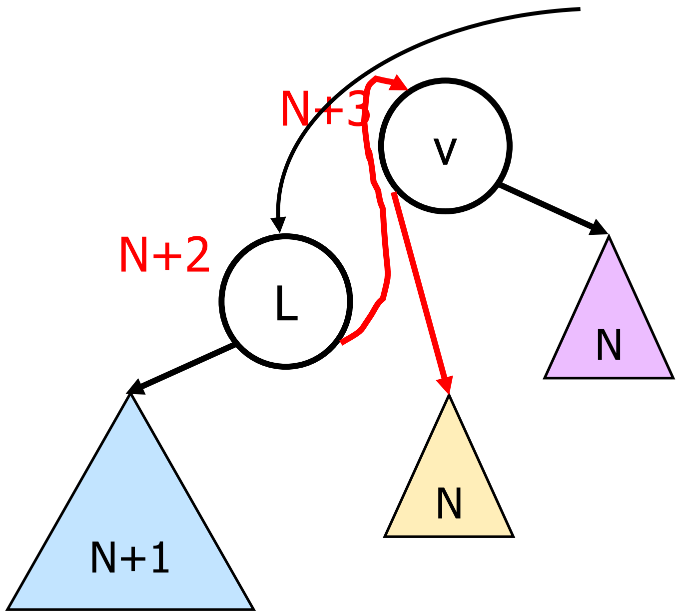
Generalizing AVL Insertion



- Mirror image case for left
  - E.g., if we added 5, 4, 3, 2, 1

Generalizing AVL Insertion

Rotate Right:
v->left = L->right
r->right = v
L is root of subtree

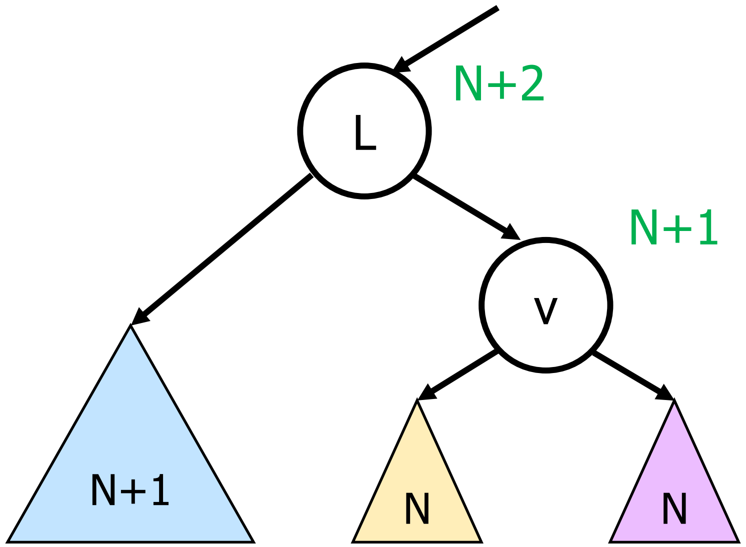


- Mirror image case for left
  - E.g., if we added 5, 4, 3, 2, 1

WATERLOO | ENGINEERING

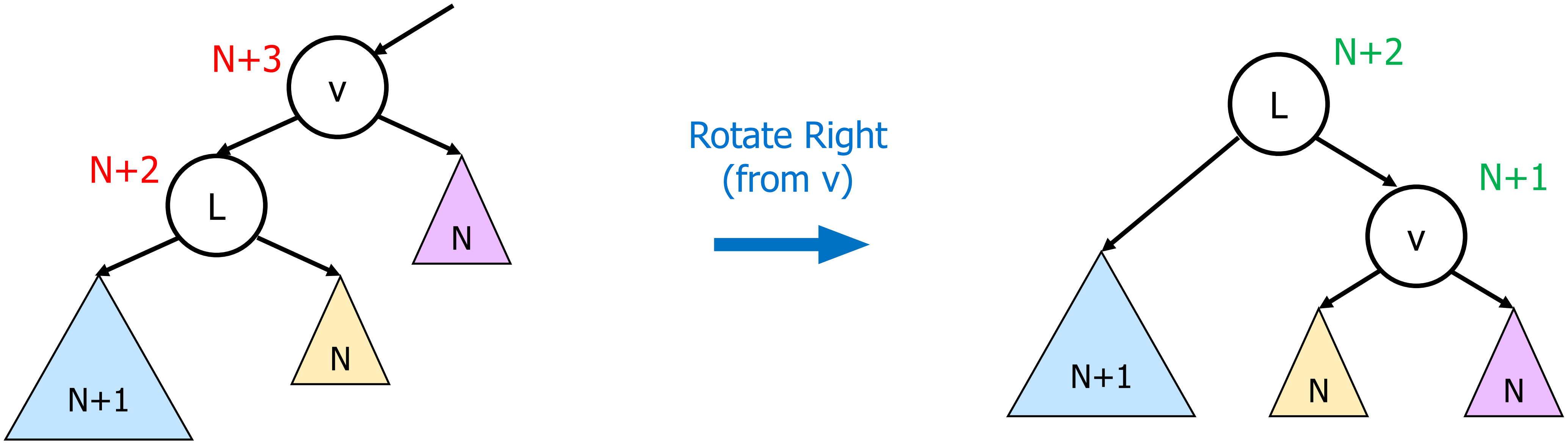
Generalizing AVL Insertion

Rotate Right:
v->left = L->right
r->right = v
L is root of subtree



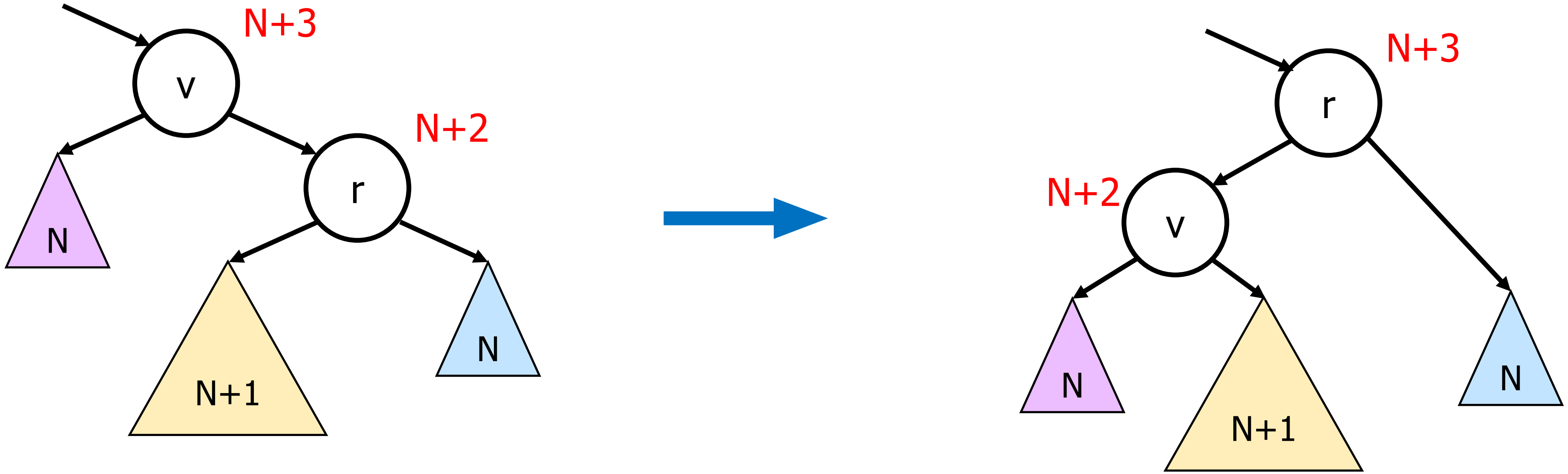
- Mirror image case for left
  - E.g., if we added 5, 4, 3, 2, 1

Generalizing AVL Insertion



- Summary
  - L moved up, v moved down
  - Restore the height of the subtree back to N+2

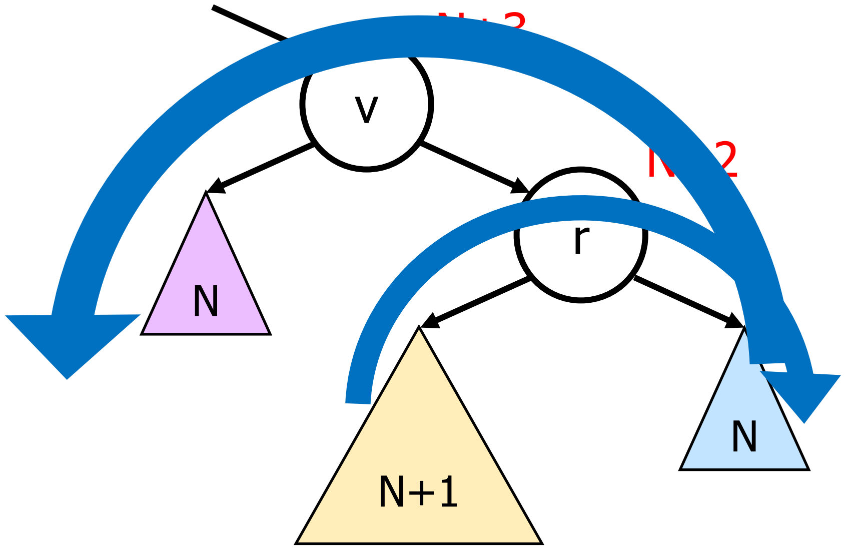
Generalizing AVL Insertion



- But what if we add to the left-side of the right
  - (or the right side of the left)
- Now doing a single rotation doesn't fix it
  - Just puts the problem on the other side

Double Rotation

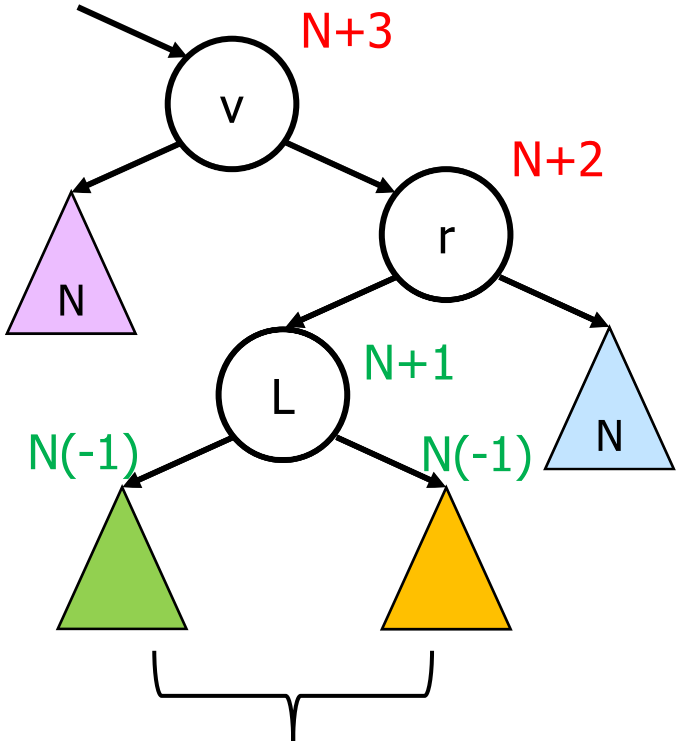
Return to this situation …



- For this case we need double rotation
  - First rotate right at r, put excess height on right side
  - Then rotate left at v, rebalance the tree

Double Rotation

To see how to do this, we need to "look inside" the yellow tree

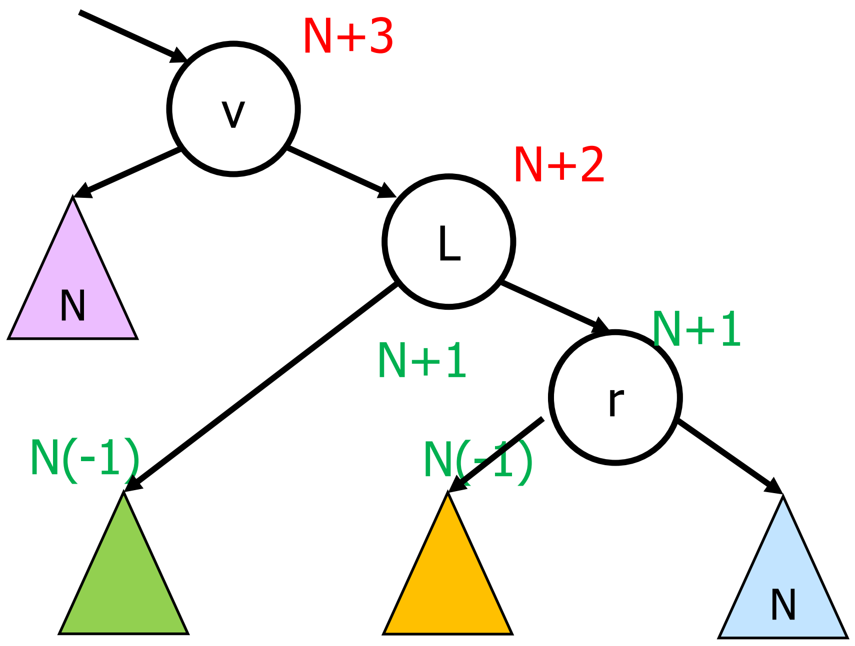


One has to be N, the other could be N or N-1

Remember v < L < r

Double Rotation

Now, rotate right at r

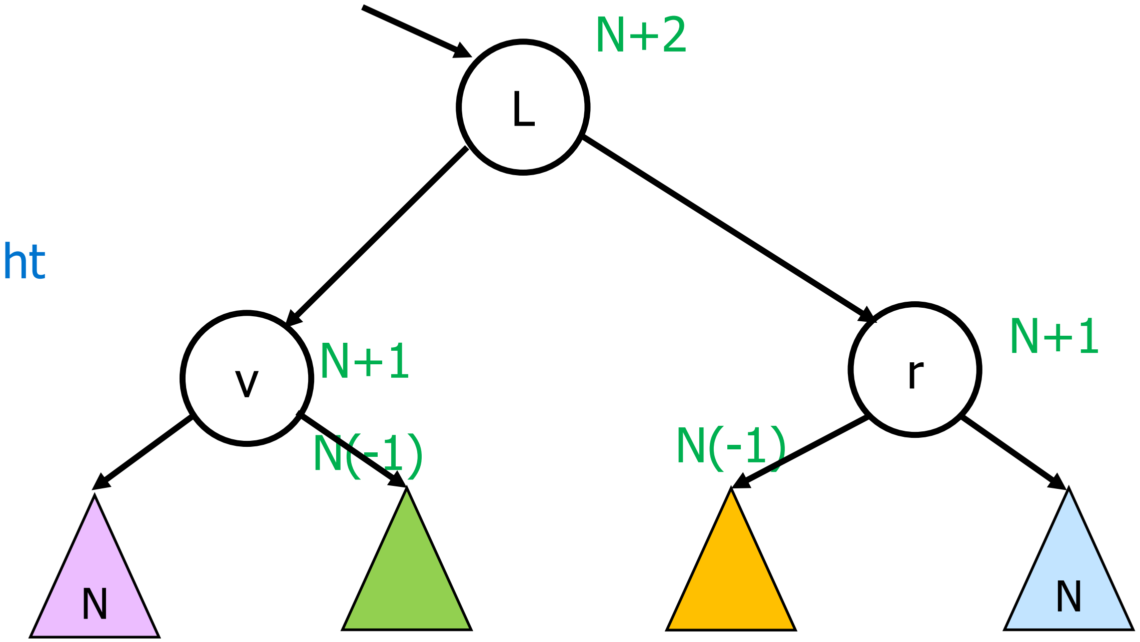


Remember v < L < r

Double Rotation

Observe that all nodes are balanced…

…and that the subtree's height is reduced back to N+2



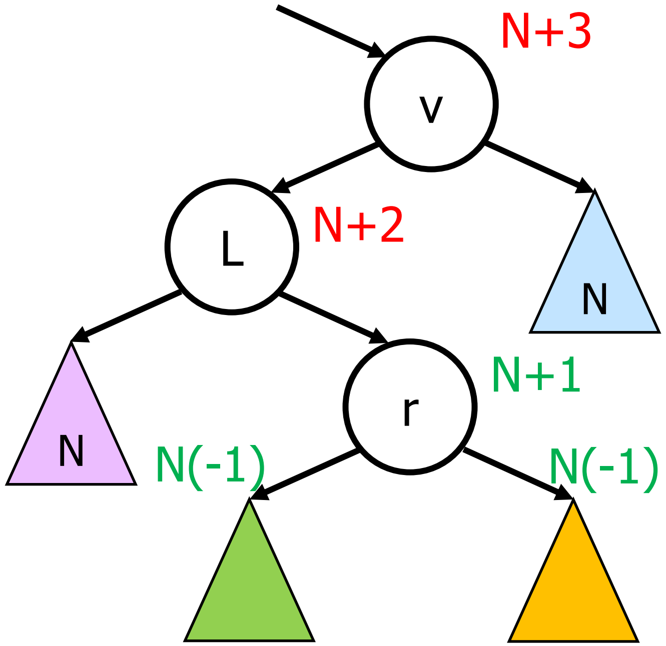
Remember v < L < r

WATERLOO | ENGINEERING

Double Rotation

Symmetric case for other side

First rotate left at L
Then rotate right at v



Remember L < r < v

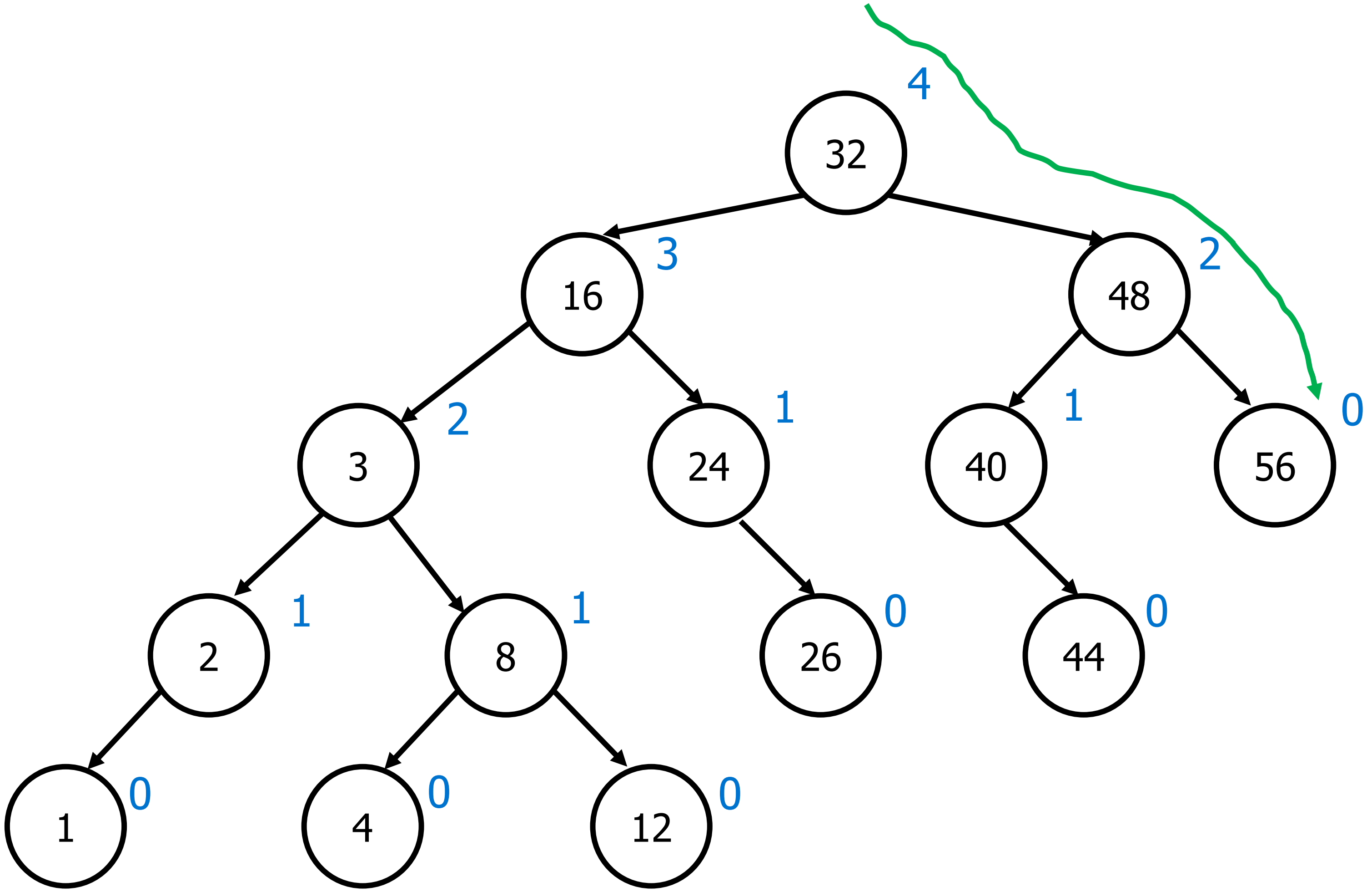
Exercise for you: 1. Draw the resulting tree after rotation;
2. Work out the pointer manipulations yourself from the drawing

AVL Deletion

- Deletion from AVL tree
  - Start with basic BST deletion algorithm (recursive)
  - On the way back up
    - Calculate balance
    - Rotate as needed
      - Same rotations
    - Update heights
  - Unlike add, multiple rotations may be required

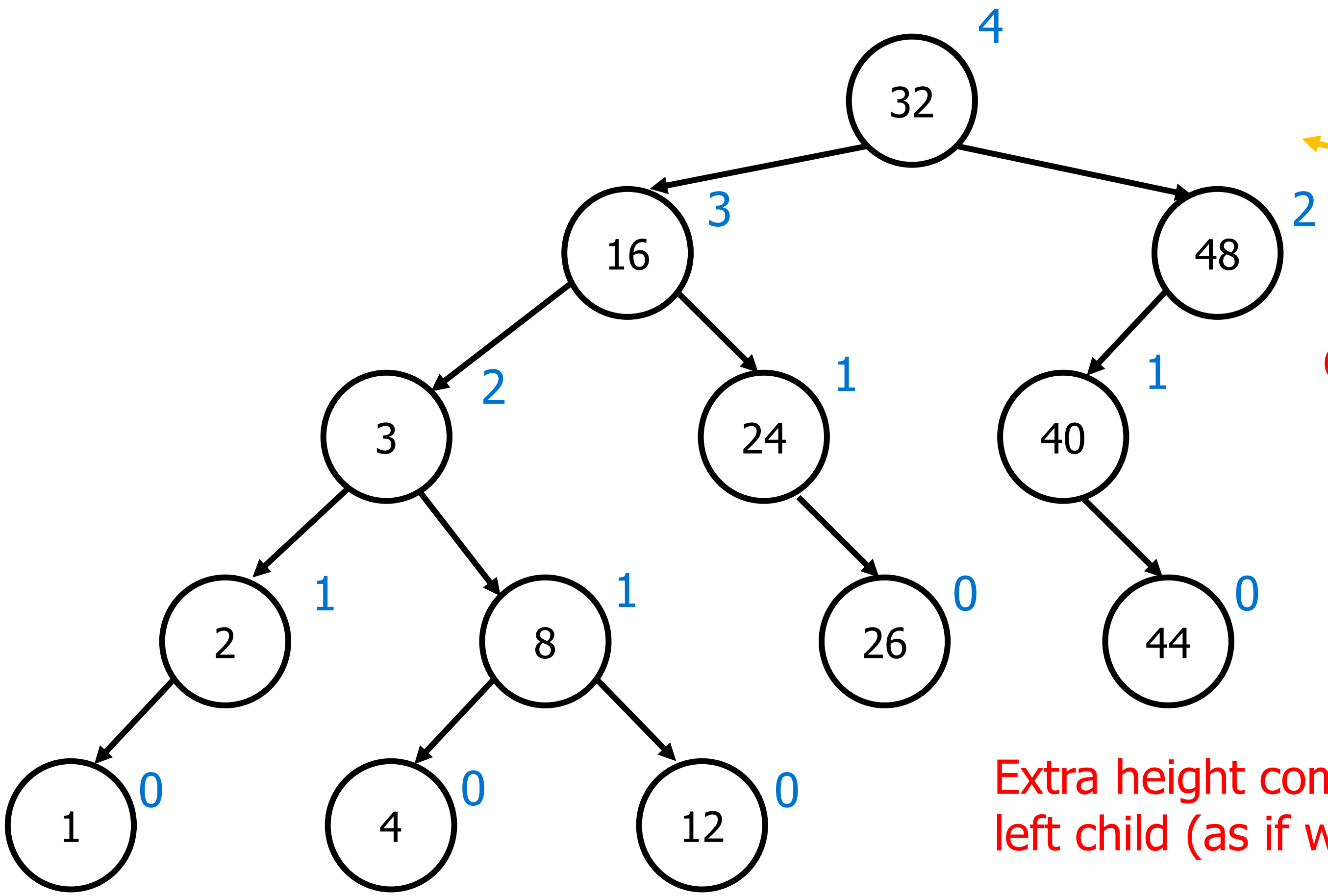
AVL Deletion Example

Delete 56



AVL Deletion Example

Delete 56



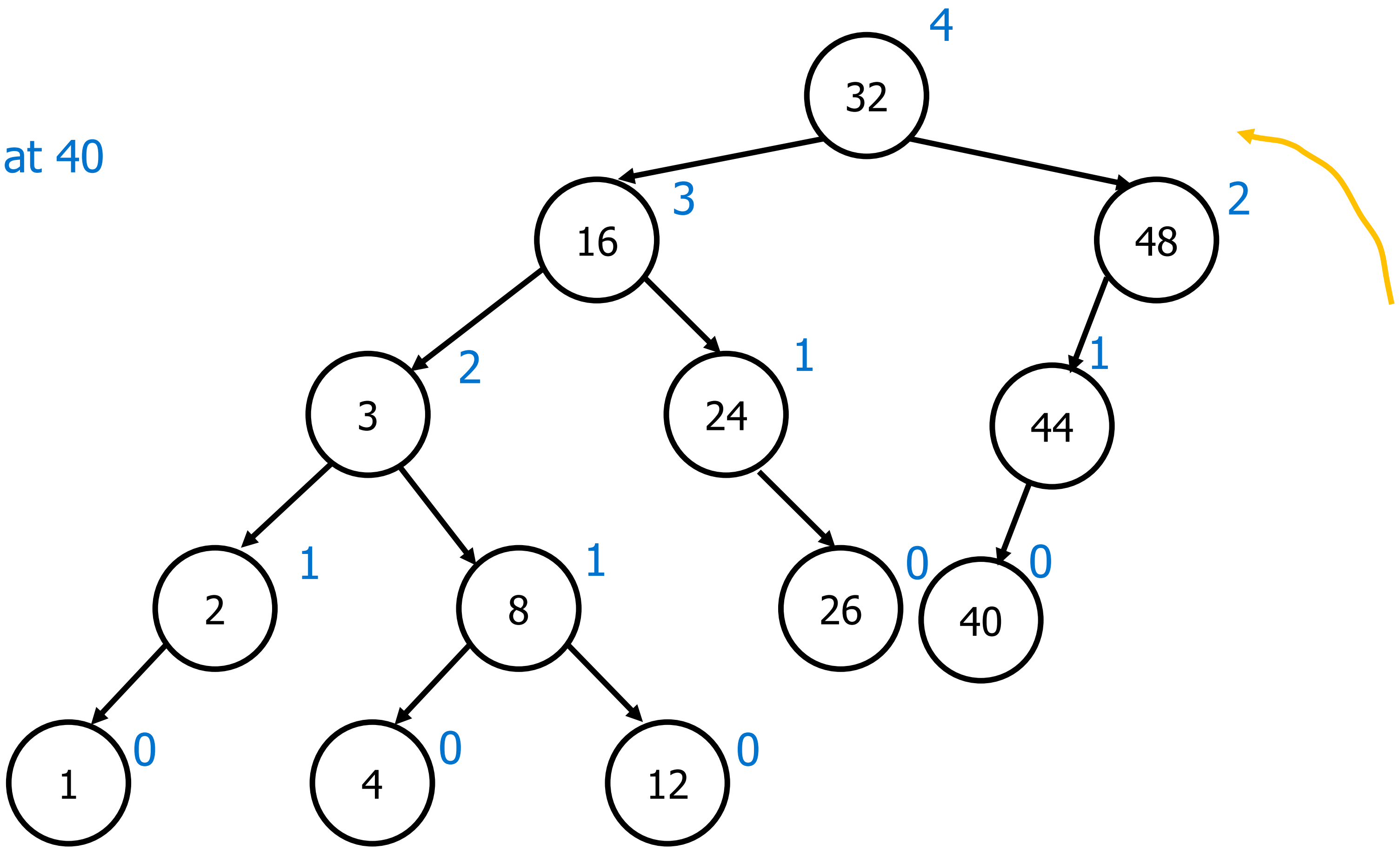
Double rotate:
- Left at 40
- Right at 48

Children heights 1 & -1 (unbalanced)

Extra height comes from right child of left child (as if we had added 44)

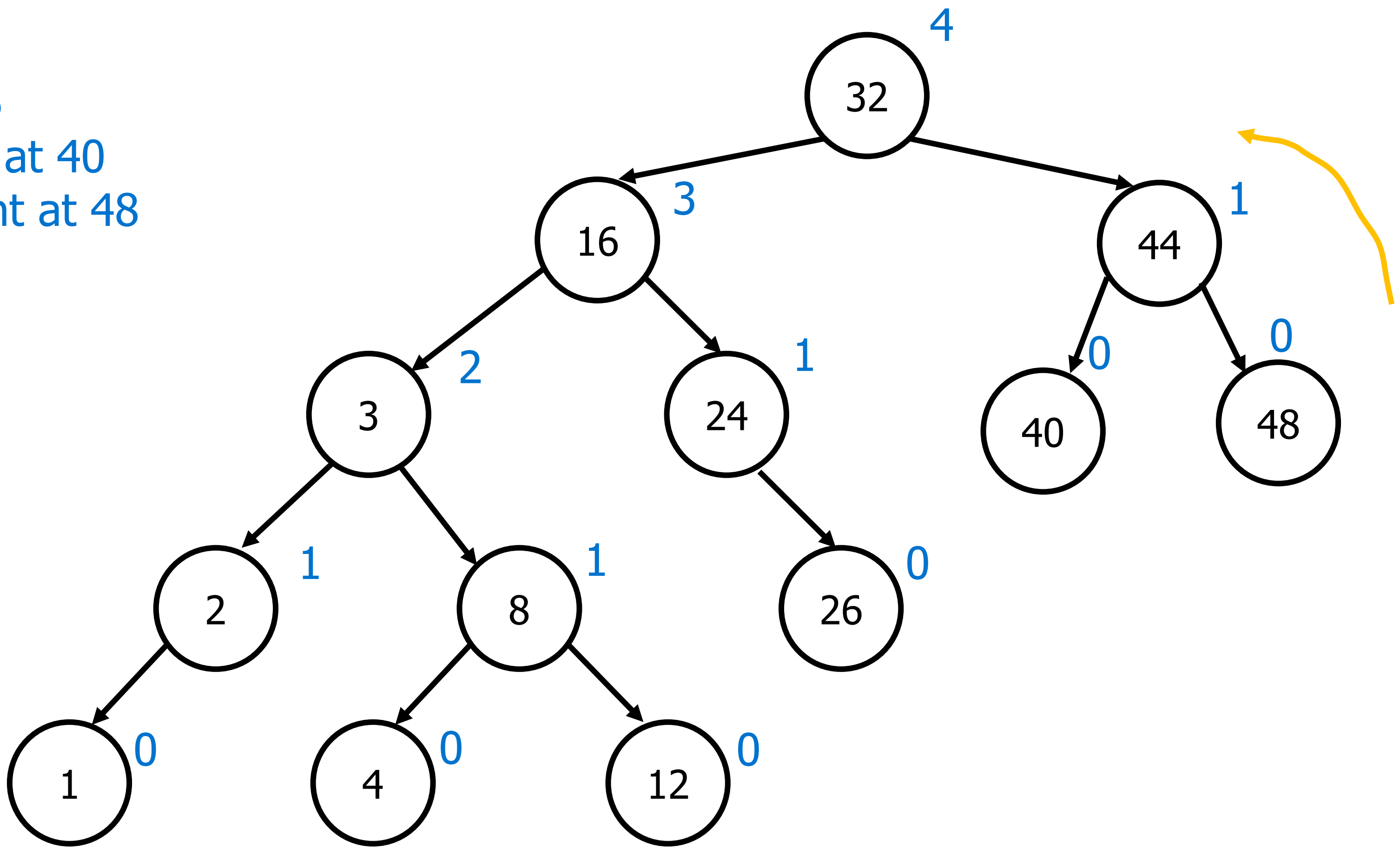
AVL Deletion Example

Delete 56
First: left at 40



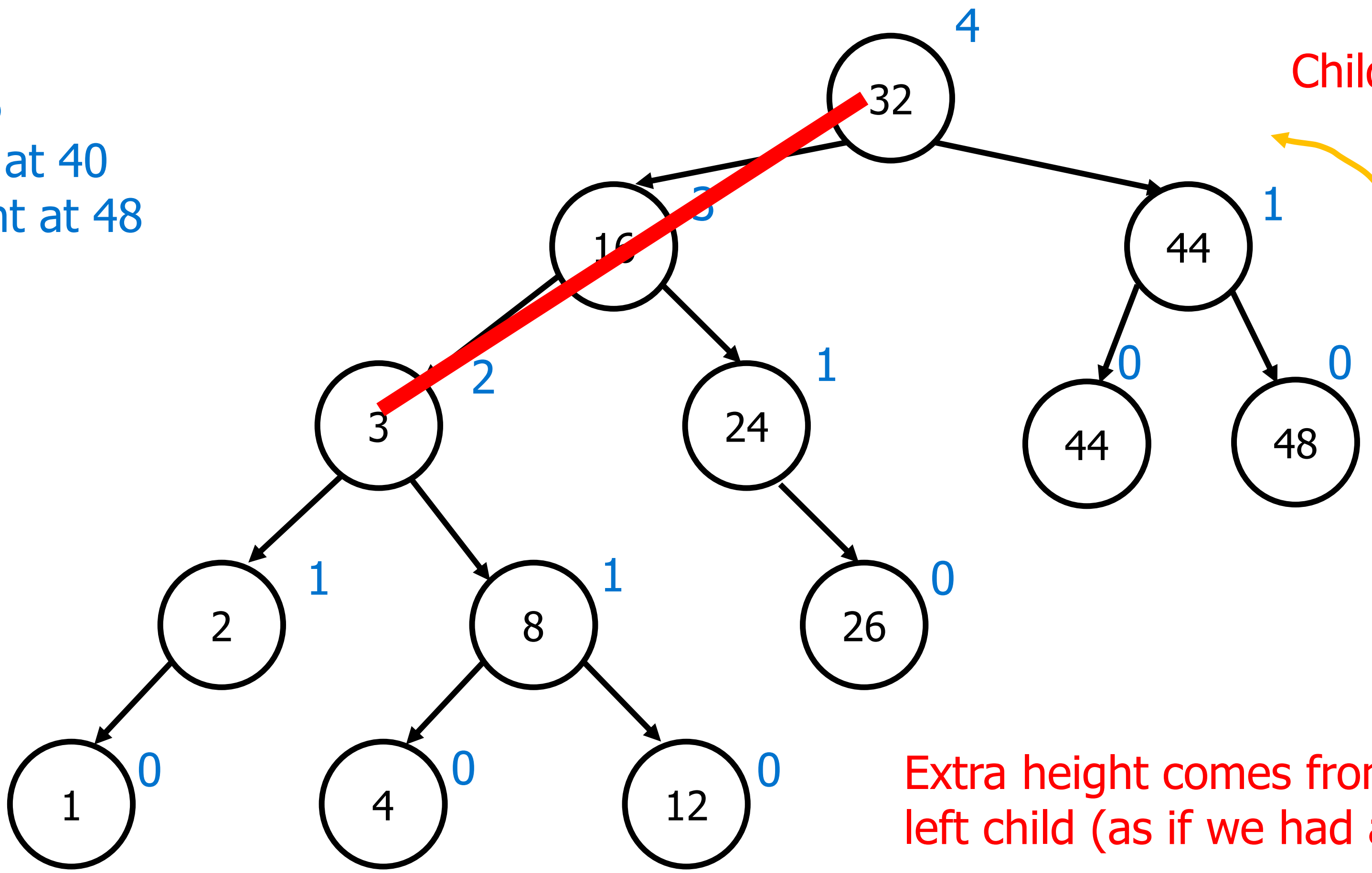
AVL Deletion Example

Delete 56
First: left at 40
Next: right at 48



AVL Deletion Example

Delete 56
First: left at 40
Next: right at 48

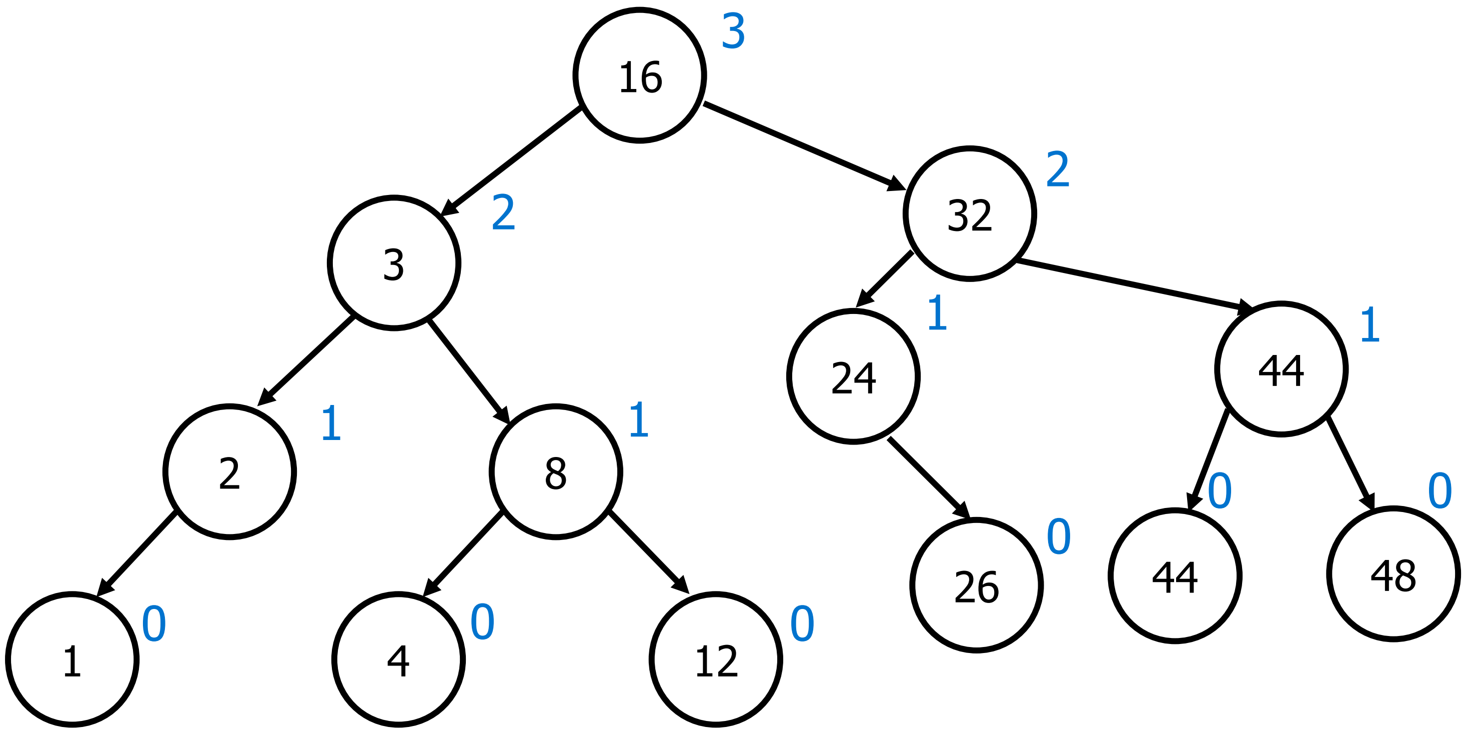


Children heights 3 & 1(unbalanced)

Extra height comes from left child of left child (as if we had added 1)

AVL Deletion Example

Delete 56
Right at 32

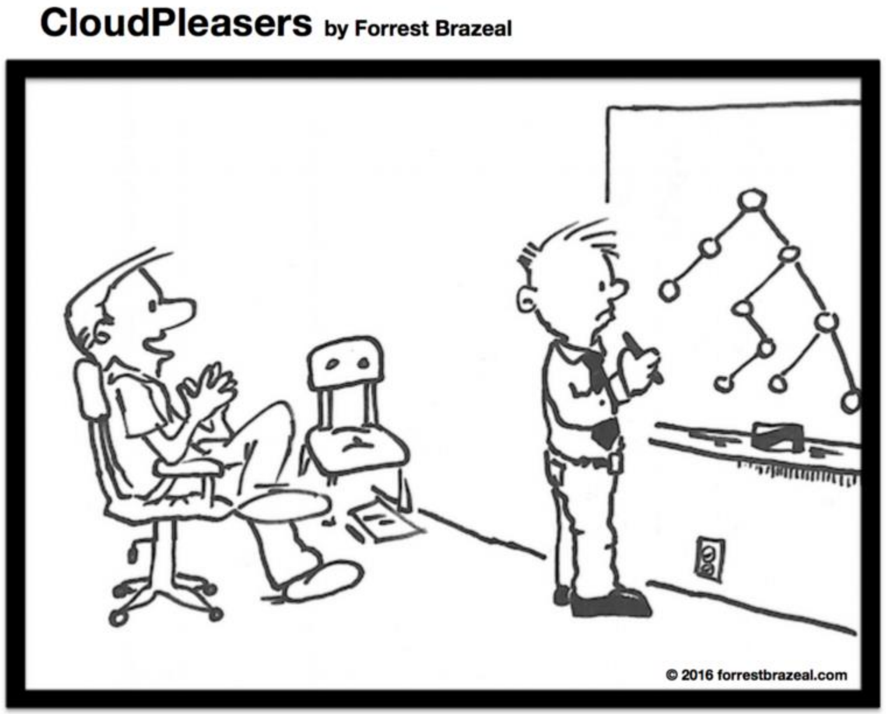


Wrap Up

- In this lecture we talked about
  - AVL trees
    - Self-balancing BSTs (via rotating nodes)
    - Ensure log(n) behavior
    - How insertion & deletion works
- Next up
  - Red-black trees

Suggested Complimentary Readings

- Data Structure and Algorithms in C++: Chapter 4.4



"We want our interviewees to solve real-world problems. So while you balance this binary search tree, I'll be changing the requirements, imposing arbitrary deadlines and auditing you for regulatory compliance."

source

Acknowledgement

- This slide builds on the hard work of the following amazing instructors:
  - Andrew Hilton (Duke)
  - Mary Hudachek-Buswell (Gatech)