COSC160: Data Structures Balanced Trees

Jeremy Bolton, PhD

Assistant Teaching Professor

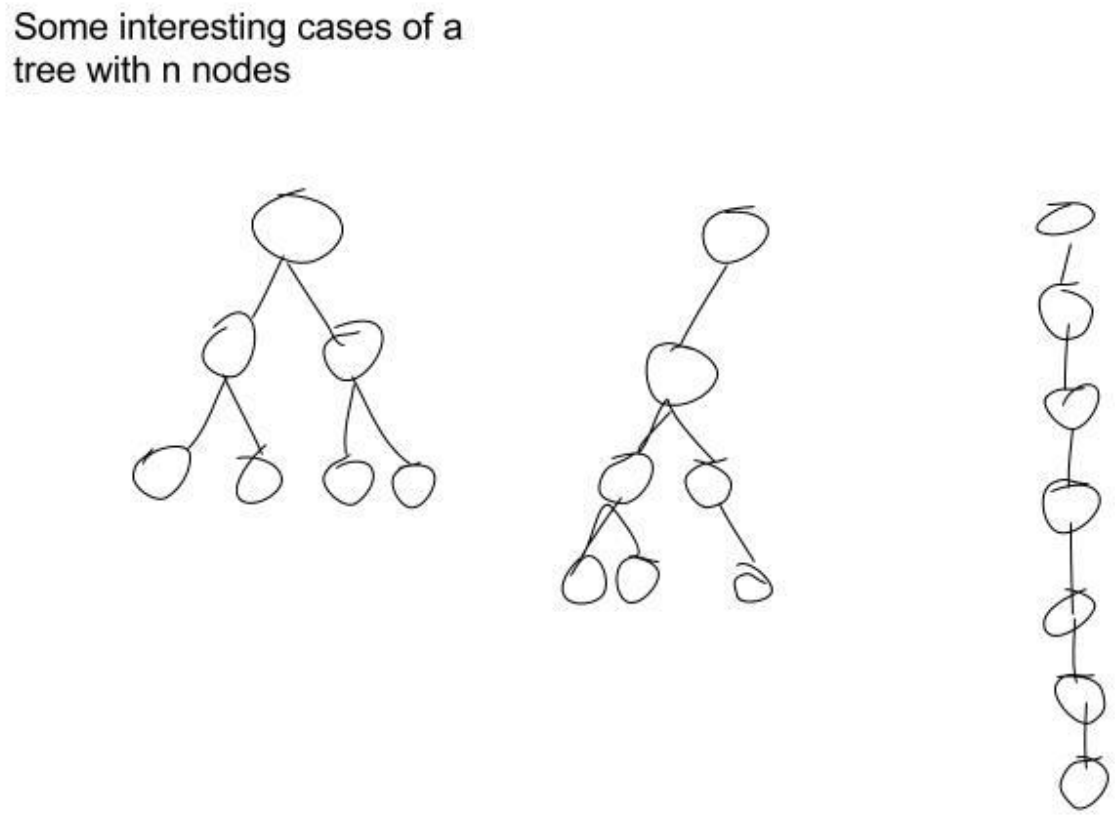
GEORGETOWN UNIVERSITY

Outline

I. Balanced Trees
   I. AVL Trees
      I. Balance Constraint
      II. Examples
      III. Searching
      IV. Insertions
      V. Removals

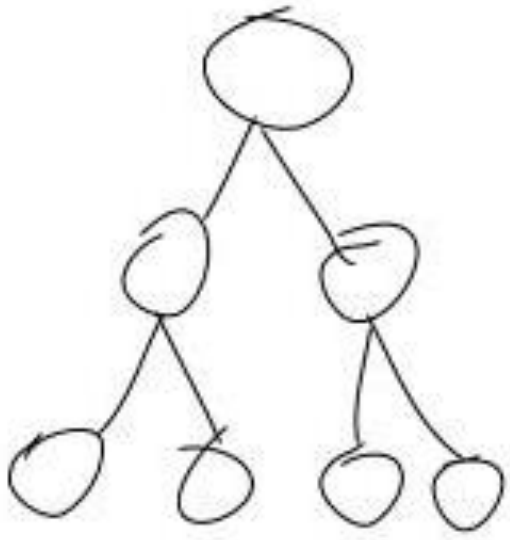
Balancing a Tree

- Binary trees lack a depth constraint. As a result the worst case insertion, removal and retrieval times are O(n).



Balanced Trees: Time Complexity Implications

- If a tree is “balanced” in some sense. The time complexity of insertions, removals and retrievals will have a logarithmic bound.

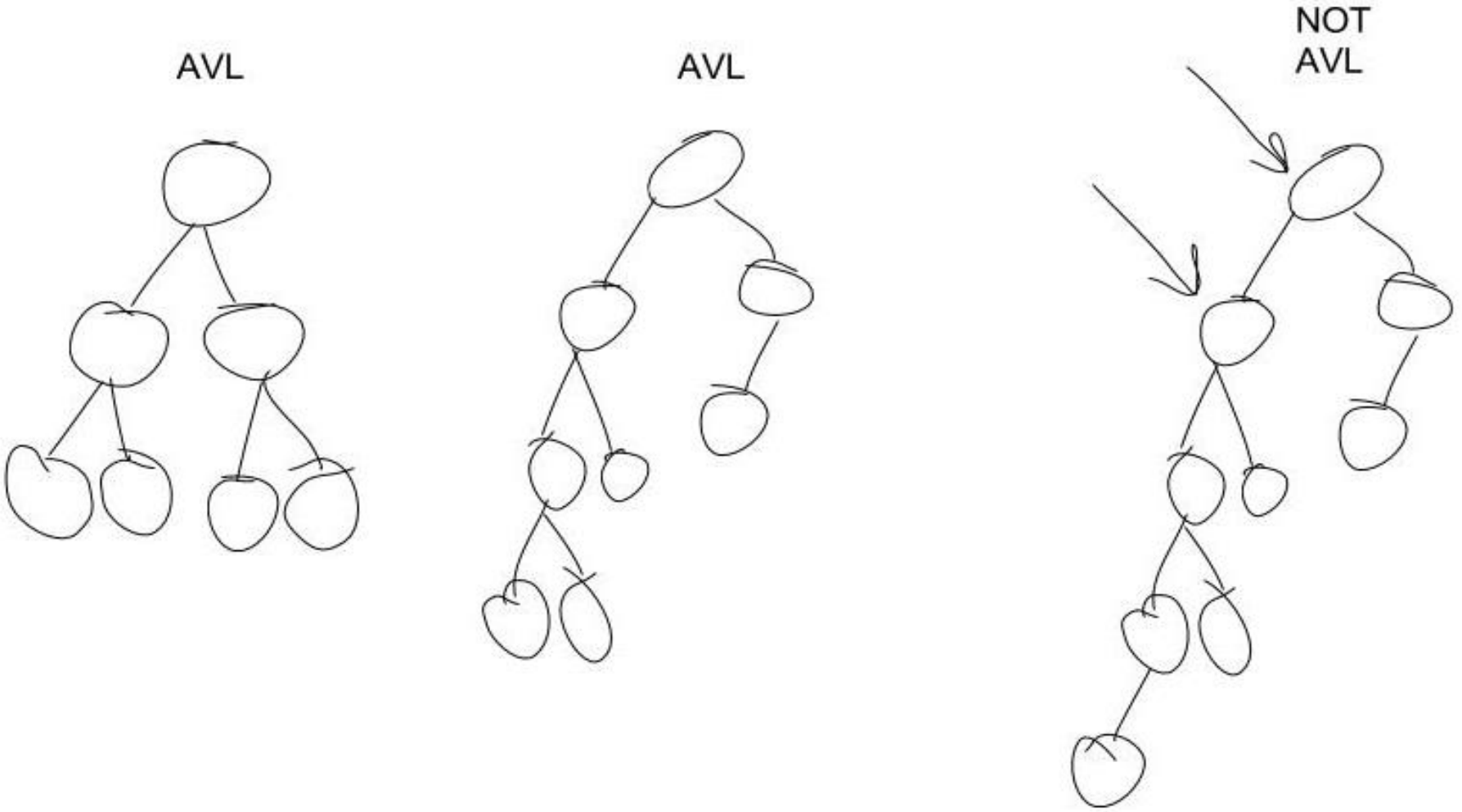


AVL Trees

- AVL trees
  - Binary Search Tree with added *balance* constraint
  - Adelson, Veliskii and Landis
  - Definition: AVL constraint
    - ***For any node in the tree, the height of the left and subtrees can differ in height only by one.***

- Uses weak notion of “balance”
  - Sufficient to ensure logarithmic depth in worst case!

Example AVL trees



Logarithmic Depth

- AVL trees are guaranteed to have logarithmic depth.
  - As a result, the **worst case time complexities** for insert, removal and retrieve is O($\log_2 n$).

- Proof idea: Prove the height can be represented as a logarithmic function of n the number of nodes.
  - By definition, the ***minimum number of nodes is in AVL tree of height h*** is defined by recurrence AVL_h:

$$AVL_h = AVL_{h-1} + AVL_{h-2} + 1$$
$$AVL_0 = 0$$
$$AVL_1 = 1$$

  - Solving and bounding this recurrence we get : $h < 1.4 \log_2(AVL_h + 2)$
    - See Appendix A.5
  - Thus worst case operations based on traversing the height h of the tree are O($\log_2 AVL_h$)

The Valid State of A Balanced Tree

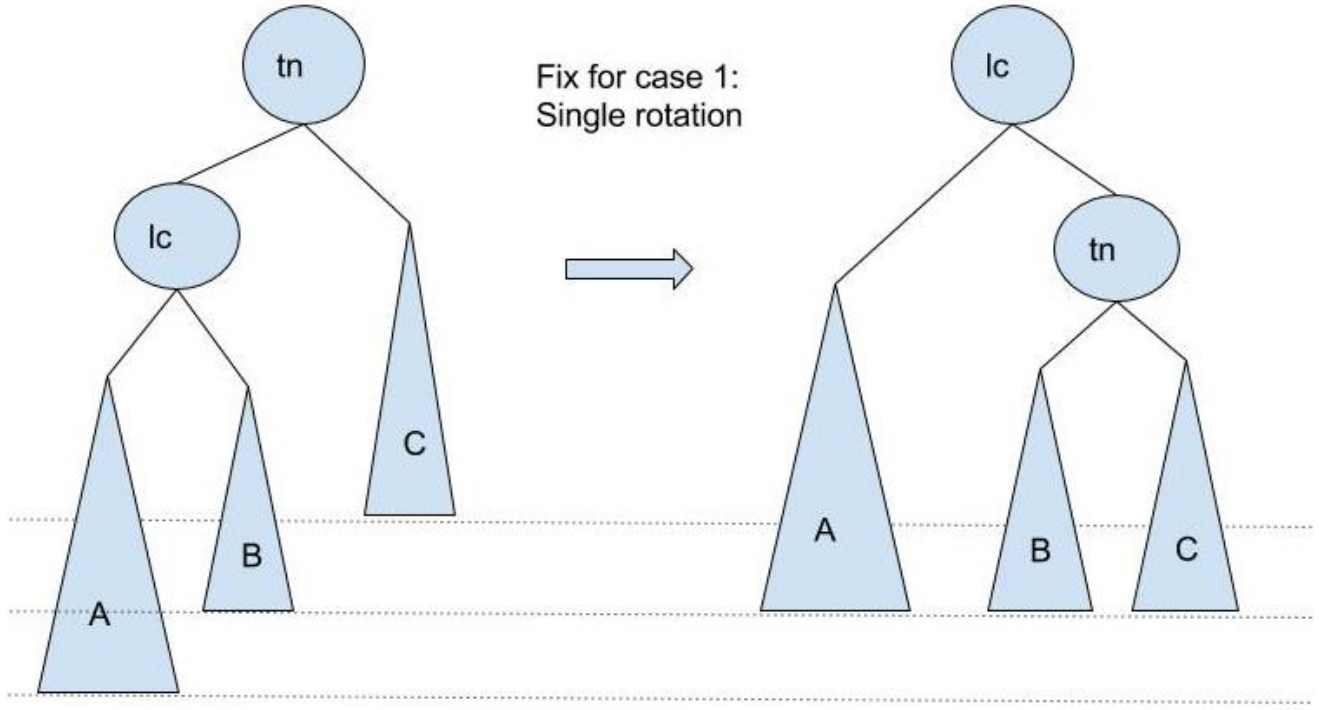
- Operations on an AVL tree may result in an “invalid” state.
  - Insert
    - Nodes along the path from the inserted node to the root might violate the balance condition.
  - Remove
    - Nodes along the path from the *replacement node* to the root might violate the balance condition.
- Re-balancing must be incorporated into these operations.

Insert into AVL Tree

- Assume node *thisNode* is the deepest node that violates the balance constraint and must be rebalanced.

- The violation may occur as a result of 1 of 4 Cases:
  - Case 1: Insertion into left subtree of the leftChild of thisNode
  - Case 2: Insertion into the right subtree of the leftChild of thisNode
  - Case 3: Insertion into the left subtree of the rightChild of thisNode
  - Case 4: Insertion into right subtree of the rightChild of thisNode

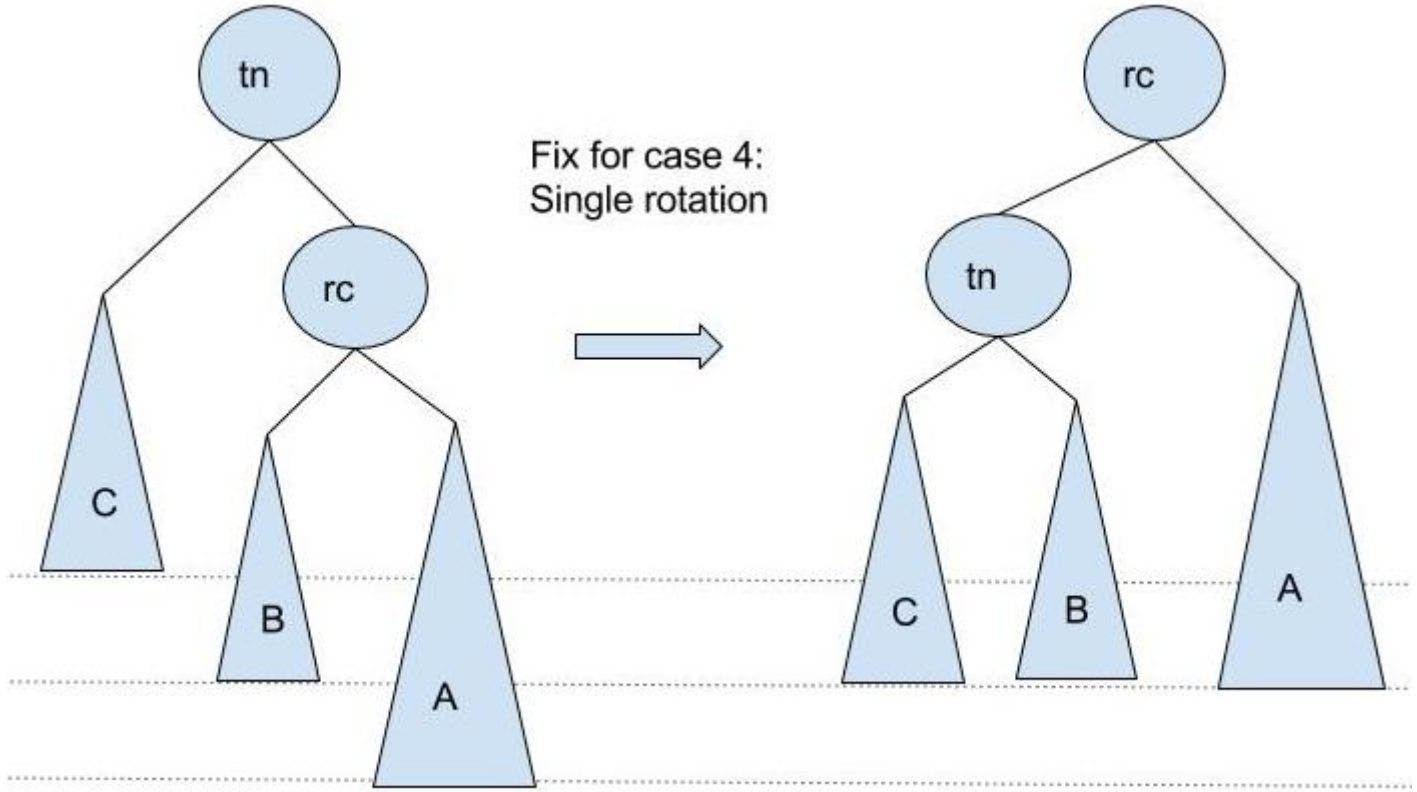
Balancing by Single Rotation Case 1




```
// rotate AVL tree single rotation
// solution for Case 1: Insertion into left subtree of the leftChild of thisNode
// performs rotation and returns new root of subtree, oldLeftChild

function rotateWithLeftChild(thisNode)
    oldLeftChild := thisNode.leftChild
    thisNode.leftChild := oldLeftChild.rightChild
    oldLeftChild.rightChild := thisNode
    return oldLeftChild
```

GEORGETOWN UNIVERSITY

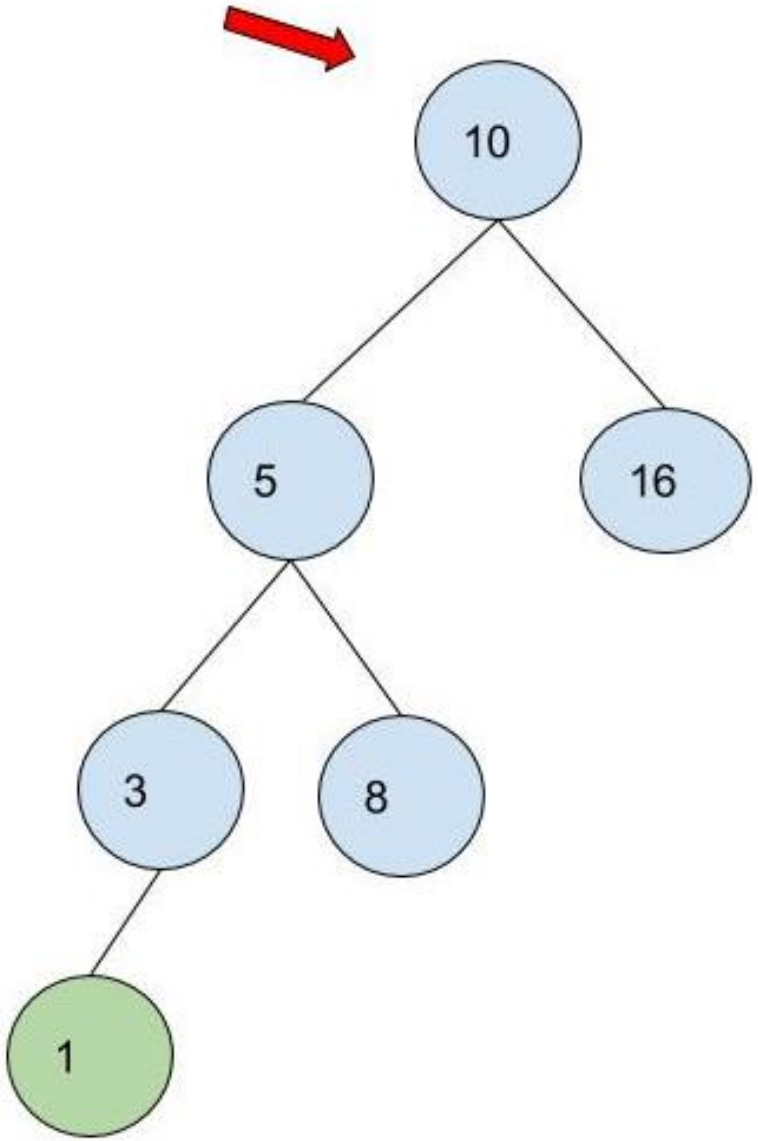
Balancing by Single Rotation Case 4



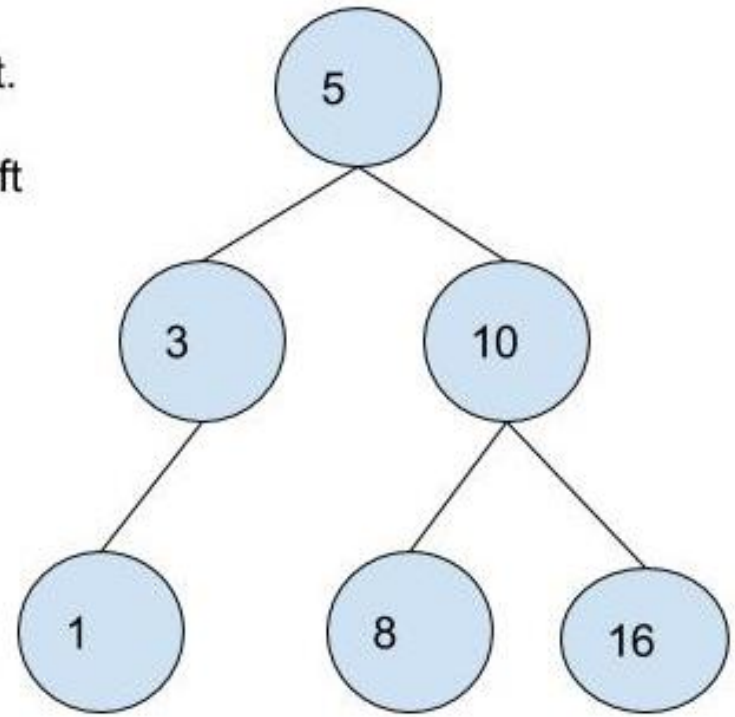

```
// rotate AVL tree single rotation
// solution for Case 4: Insertion into right subtree of the rightChild of thisNode
// performs rotation and returns new root of subtree, oldRightChild

function rotateWithRightChild(thisNode)
    oldRightChild := thisNode.rightChild
    thisNode.rightChild := oldRightChild.leftChild
    oldRightChild.leftChild := thisNode
    return oldRightChild
```

GEORGETOWN UNIVERSITY

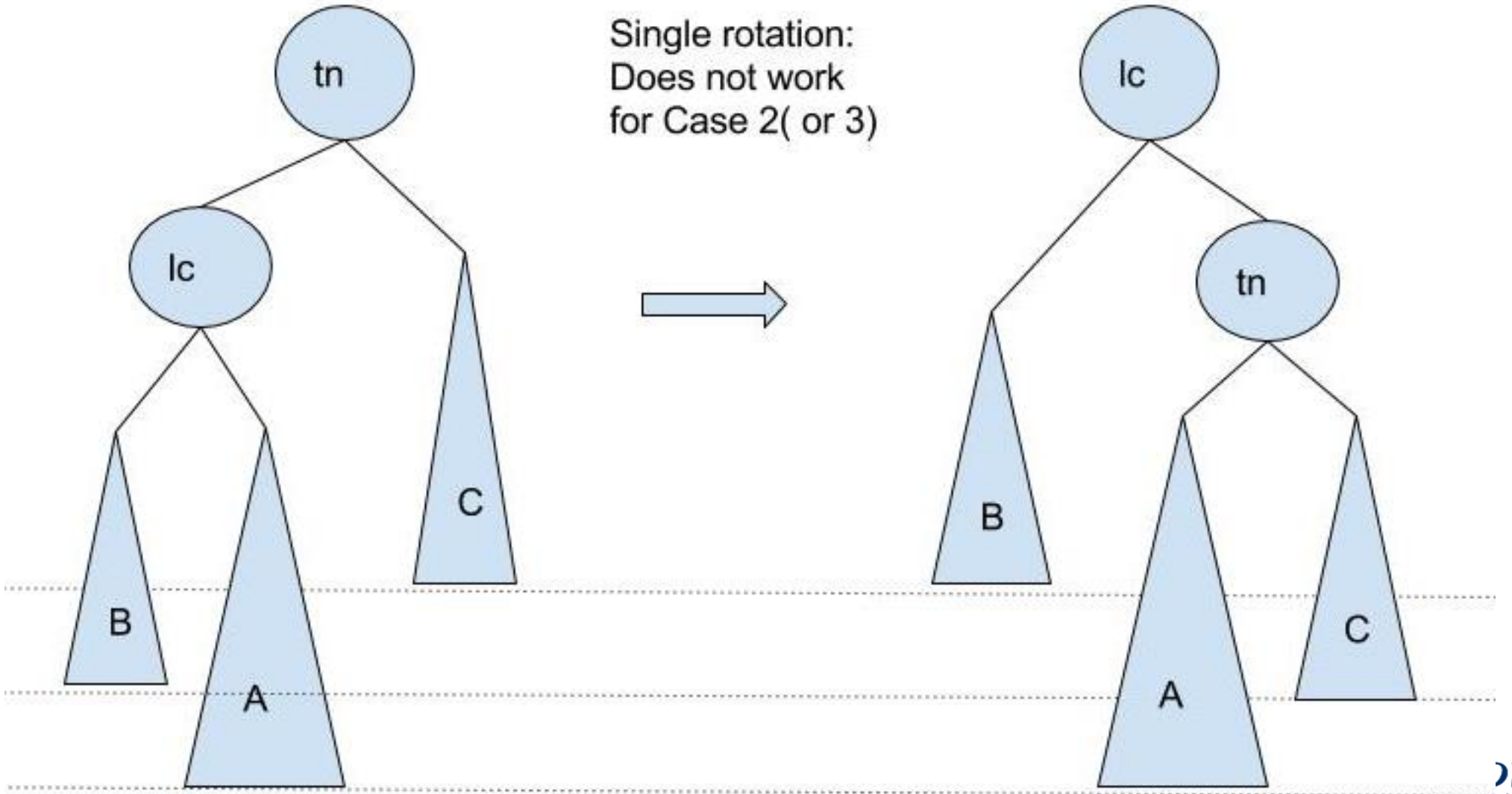
Balancing by Single Rotation



Fix for case 1:
Violation at root.
Single rotation
with root and left
child



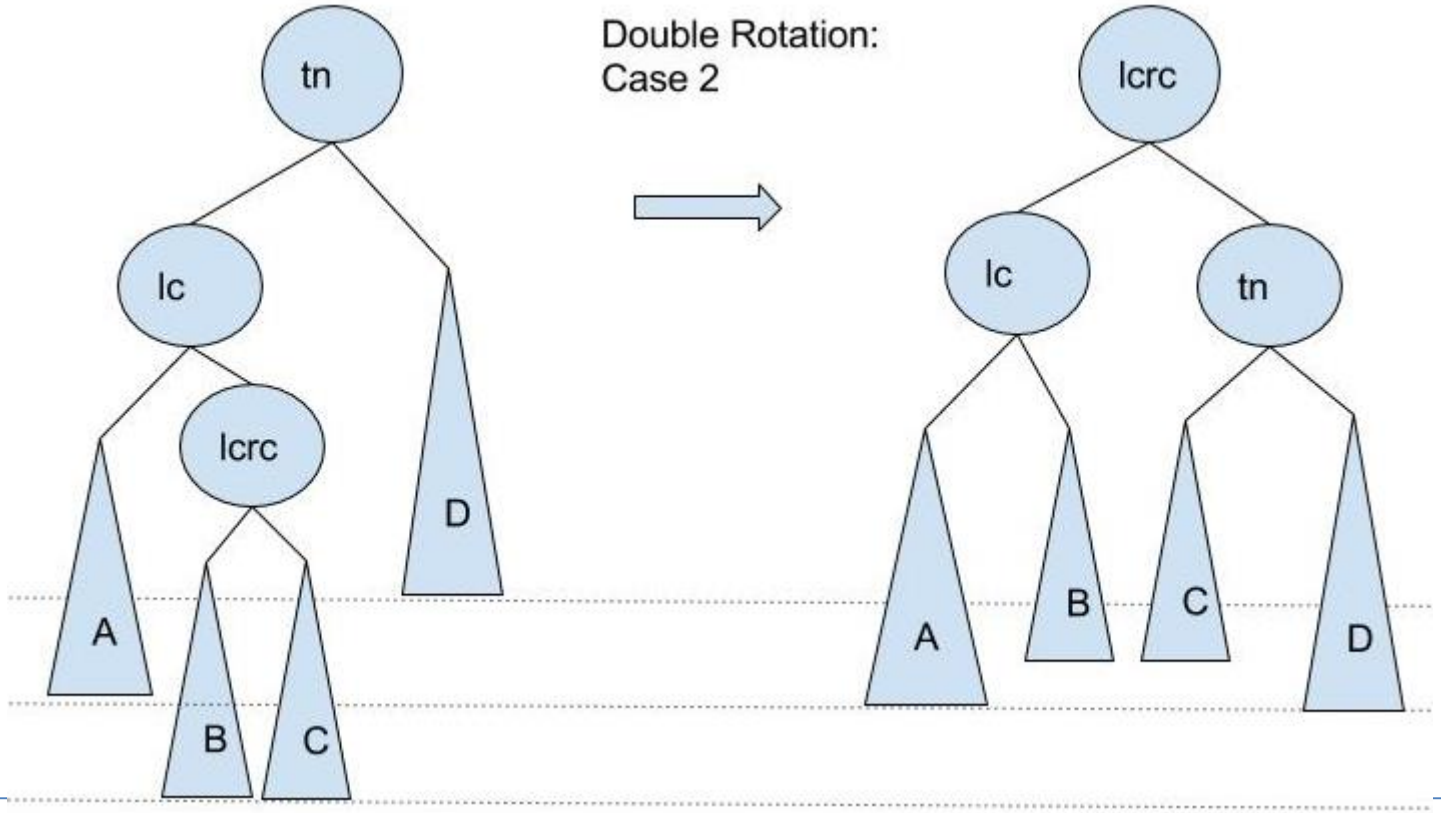
Single Rotation fails when "inner" subtree is largest



Double Rotations.

- Example: Double rotation balances case 2 (and 3)
  - Rotate between thisNode's child and grandchild
  - Rotate between thisNode and its new child

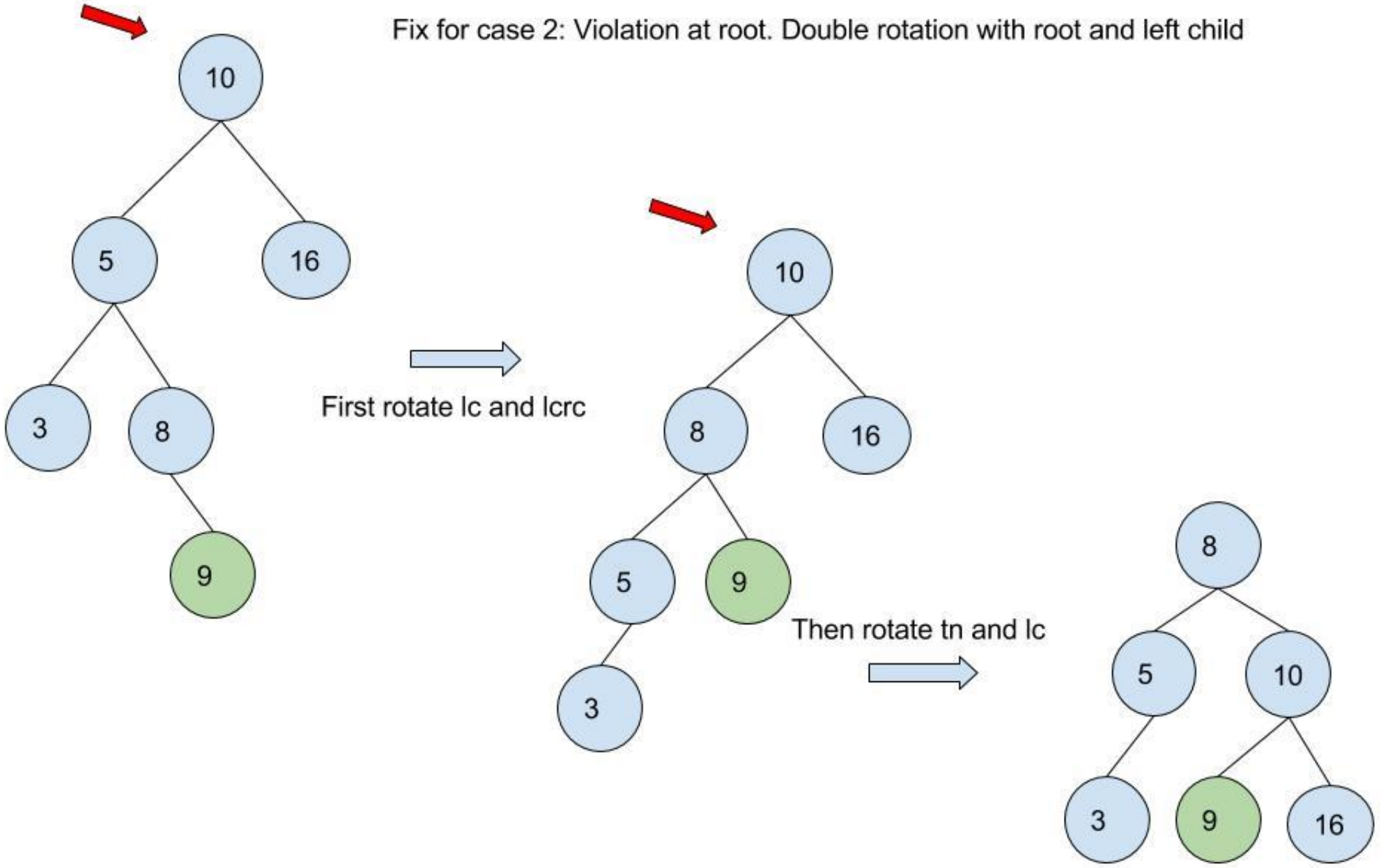
Double rotate with left child: Case 2



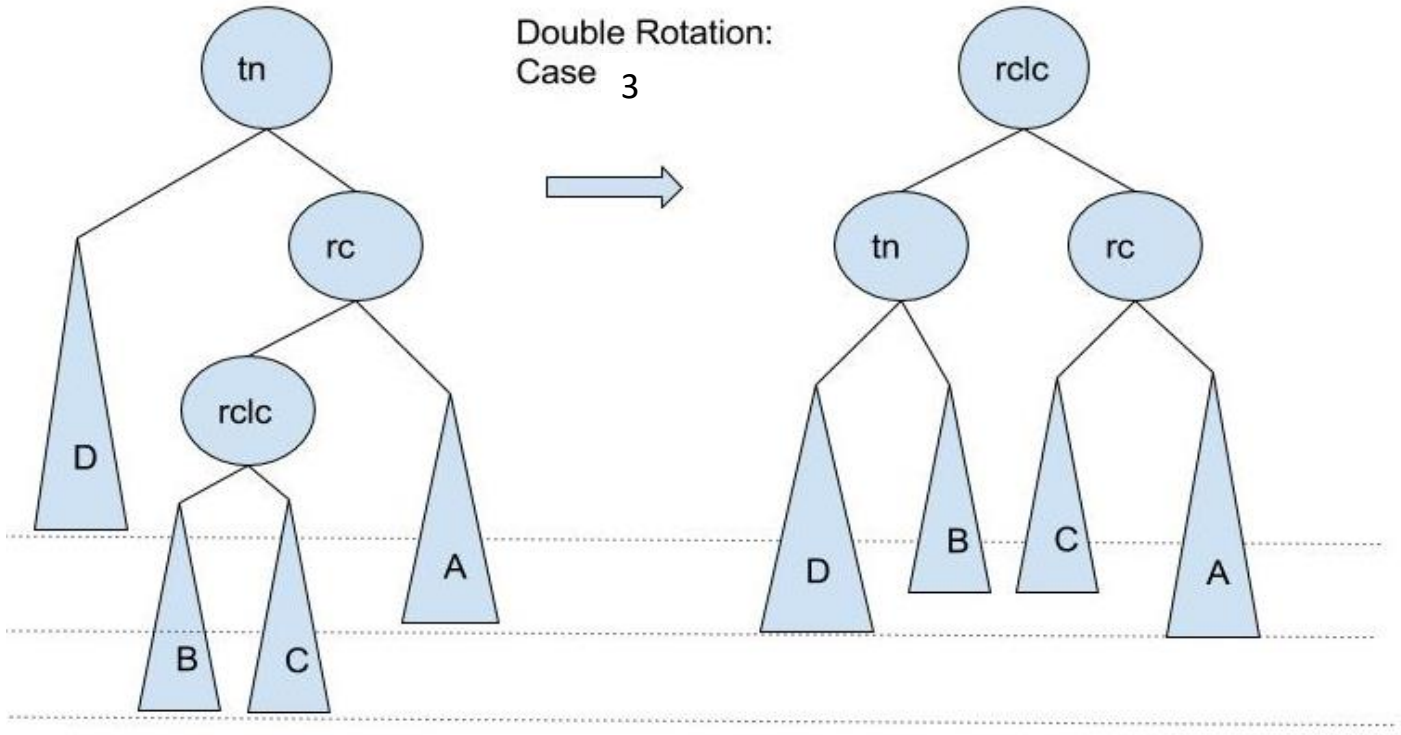

```
// rotate AVL tree single rotation
// solution for Case 2:
// performs double rotation and returns new root of subtree

function doubleRotateWithLeftChild(thisNode)
    thisNode.leftChild := rotateWithRightChild(thisNode.leftChild)
    return rotateWithLeftChild(thisNode)
```

GEORGETOWN UNIVERSITY

Example: Balancing by Double Rotation



Double rotate with right child: Case 3




```
// rotate AVL tree single rotation
// solution for Case 3
// performs double rotation and returns new root of subtree

function doubleRotateWithRightChild(thisNode)
    thisNode.rightChild := rotateWithLeftChild(thisNode.rightChild)
    return rotateWithRightChild(thisNode)
```

Insert into AVL Tree

```
// insert key in AVL tree rooted at root
function insertAVL(&root, key)
    if root is NULL   // base case: insert node here!
        root := new node
        root.key := key
    if root.key == key, then return // assume no duplicates
    if root.key > key // continue down left subtree
        insertAVL(root.leftChild, key)
        if |height(root.leftChild) − height(root.rightChild)| == 2
            if root.leftChild.key > key // case 1
                root := rotateWithLeftChild(root)
            else  // case 2
                root := doubleRotateWithLeftChild(root)
    if root.key < key // continue down right subtree
        insertAVL(root.rightChild, key)
        if |height(root.leftChild) − height(root.rightChild)| == 2
            if root.leftChild.key < key // case 4
                root := rotateWithRightChild(root)
            else // case 3
                root := doubleRotateWithRightChild(root)
```

Note: root must be passed by ***reference*** in this implementation! This allows for appropriate linking to parents for base case and rotations

Code facilitating insertion shown in blue

Code enforcing balance shown in orange

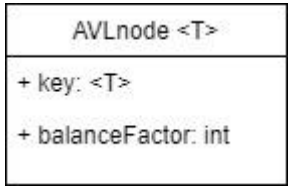
GEORGETOWN UNIVERSITY

Try this at home

- Note: we must check the height of each subtree to check for balance. Change the insert pseudocode so we do not need to recompute the height during each step of traversal for an insert!
  - In the pseudocode provided, this was done by calling a height function
    - This is slow! How slow?
    - Instead a height attribute -- AKA *balance factor* -- should be maintained at each node and updated during any insertion or removal.

Balance Factor

- Balance Factor.
  - Stored at each node in AVL tree
  - balanceFactor = height of right subtree less height of left subtree
  - Valid values are -1, 0, and +1.
  - During insertion and removal, all affected balance factors must be updated
  - Balance factors are then checked to determine if rotation is necessary



AVL Tree: Remove

- Removal is similar to insertion: perform alteration, then rebalance
  - Rebalance using single and double rotations
  - Difference: imbalance may propagate upwards, rotations at multiple nodes along the path to the root may be necessary

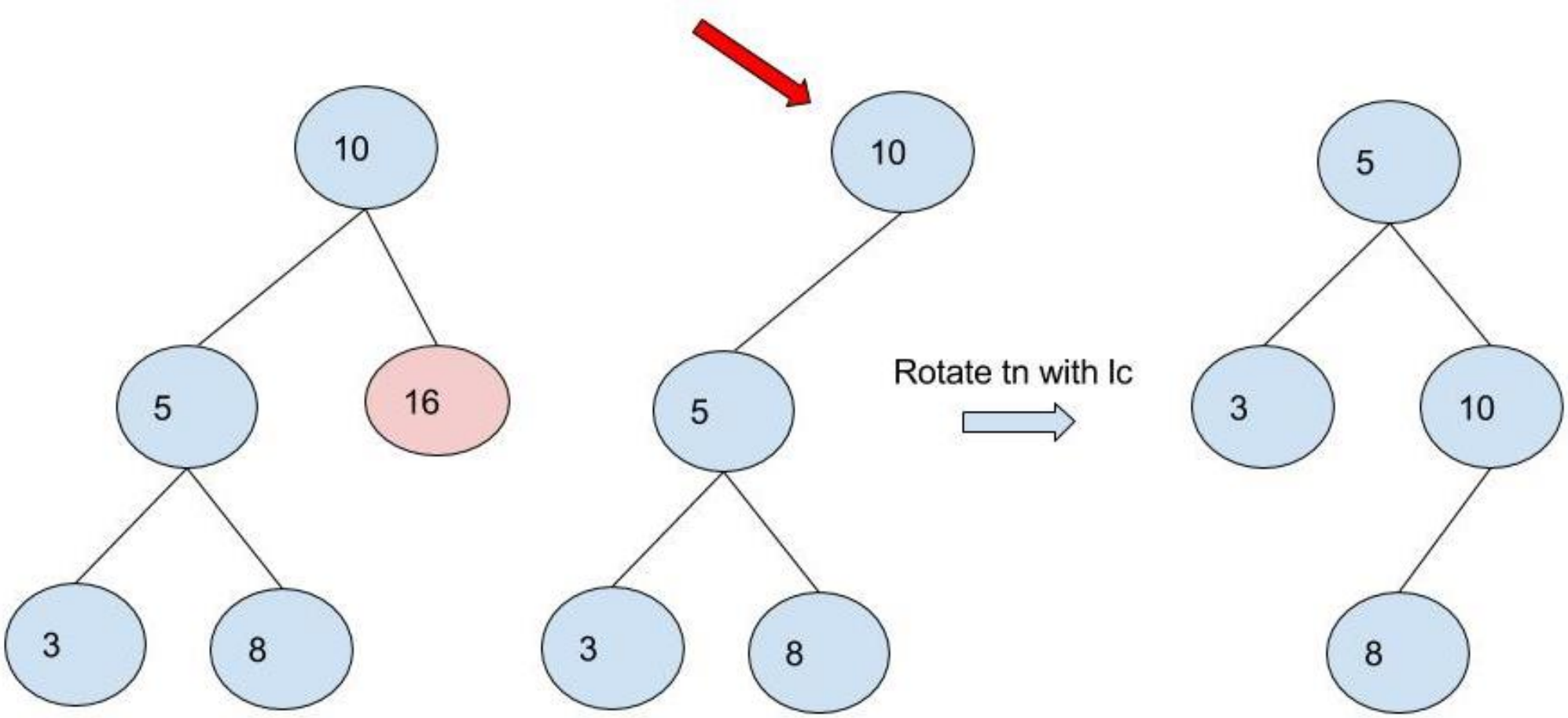
Cases for Removal from AVL tree

- Similar to cases for standard BST
- Delete node from AVL tree
  - Case 0: 0 children. delete node.
  - Case 1: 1 child. Connect child to parent.
  - Case 2: 2 children. Choose replacement node (e.g. min val in right subtree) and make the appropriate substitution.

- re-balance as needed
  - Case 0: all nodes from removed node to root must be checked
  - Case 1: all nodes from removed node to root must be checked
  - Case 2: all nodes from *deleted replacement node* to root must be checked

- Can rebalance while returning along the path.

Remove Example



Try at home

- Create pseudocode for AVL node removal
  - Cases and reconnection of separated tree similar to BST node removal
  - Height condition checks and rotations similar to AVL insert

Time Complexity including Balance

- Search: $\Theta(\log_2 n)$

- Note: *Height Balancing* only adds constant factor to complexity since
  - Balancing is performed while simultaneously traversing the tree for insert or remove
  - *ie* the traversal is “free”, and each balance case in done in a constant number of operations.
  - Should keep height attribute at each node; otherwise complexity may increase

- Insertion: $\Theta(\log_2 n)$

- Removal: $\Theta(\log_2 n)$

Summary: AVL Trees

- Efficient
  - Constraint ensures logarithmic depth
- Theoretically fast, but rebalancing will decrease execution by a constant factor.
- Other Notes:
  - For large trees, the log time will be a huge savings
  - But there are practical issues related to cache, memory, and other delays to consider
    - For example linear search time (without any memory delays) may be better than log search time (with memory delays).
    - B-Trees …