Lecture 17
AVL Trees

EECS 281: Data Structures & Algorithms

Search/Insert

- Retrieval of a particular piece of information from large volumes of previously stored data
- Purpose is typically to access information within the item (not just the key)
- Recall that arrays, linked lists are worst-case $O(n)$ for either searching or inserting

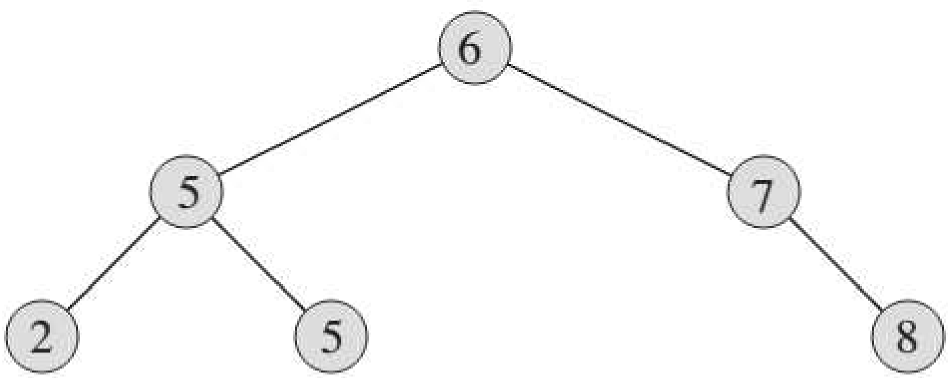
Need data structure with optimal efficiency for searching and inserting

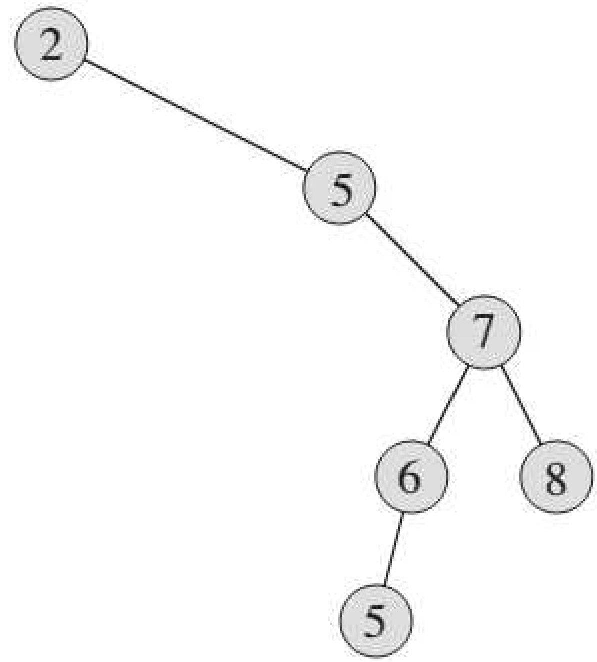
Review: Properties of BSTs

- The complexity of tree functions depends on the height of the tree
- Average case (balanced): about log *n* nodes between root and each external node/leaf
- Worst case (unbalanced): about *n* nodes between root and each external node/leaf

Review: BST Property

- The key of any node is:
  ≥ keys of all nodes in its left subtree and
  ≤ keys of all nodes in its right subtree



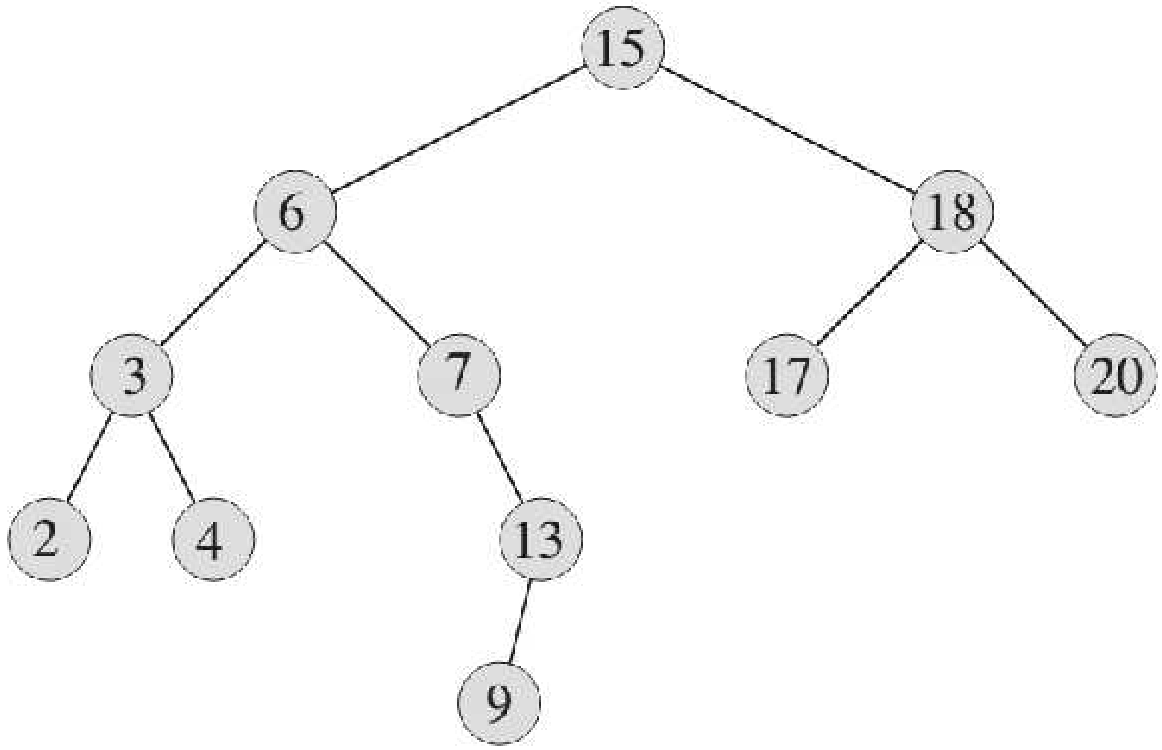


Review: Insert

- How do we insert a new key into the tree?
- Similar to search
- Start at the root, and trace a path downwards, looking for a leaf to append the node

Insert Example

```
tree_insert(10);
```



Tree Terminology Review

Depth:

depth(empty) = 0;

depth(node) = depth(parent) + 1;

Height:

height(empty) = 0;

height(node) = max(height(children)) + 1;

Max Height/Depth:

maximum height/depth of tree's nodes

AVL Tree

Named for <u>A</u>delson-<u>V</u>elskii, and <u>L</u>andis

- Change worst case search/insert to $O(\log n)$
- Height Balance Property
  - For every internal node *v* of *T*, the heights of the children of *v* differ by at most *1*
- Use rotations to correct imbalance ASAP

Proof of Height Balance Property

- Define the minimum number of nodes in tree of height h as $n(h)$
  - $n(0) = 0$, $n(1) = 1$
  - For $n > 1$, an AVL tree of height h, with a minimum number of nodes contains the root node, one AVL subtree of height $n - 1$ and another of height $n - 2$
  - Thus $n(h) = 1 + n(h - 1) + n(h - 2)$
  - Knowing $n(h - 1) > n(h - 2)$, $n(h) > 2n(h - 2)$, then by induction, $n(h) > 2^i n(h - 2^i)$
  - Closed form solution, $n(h) > 2^{h/2 - 1}$
  - Taking logarithms: $h < 2 \log n(h) + 2$
  - Thus the height of the tree is $O(\log n)$

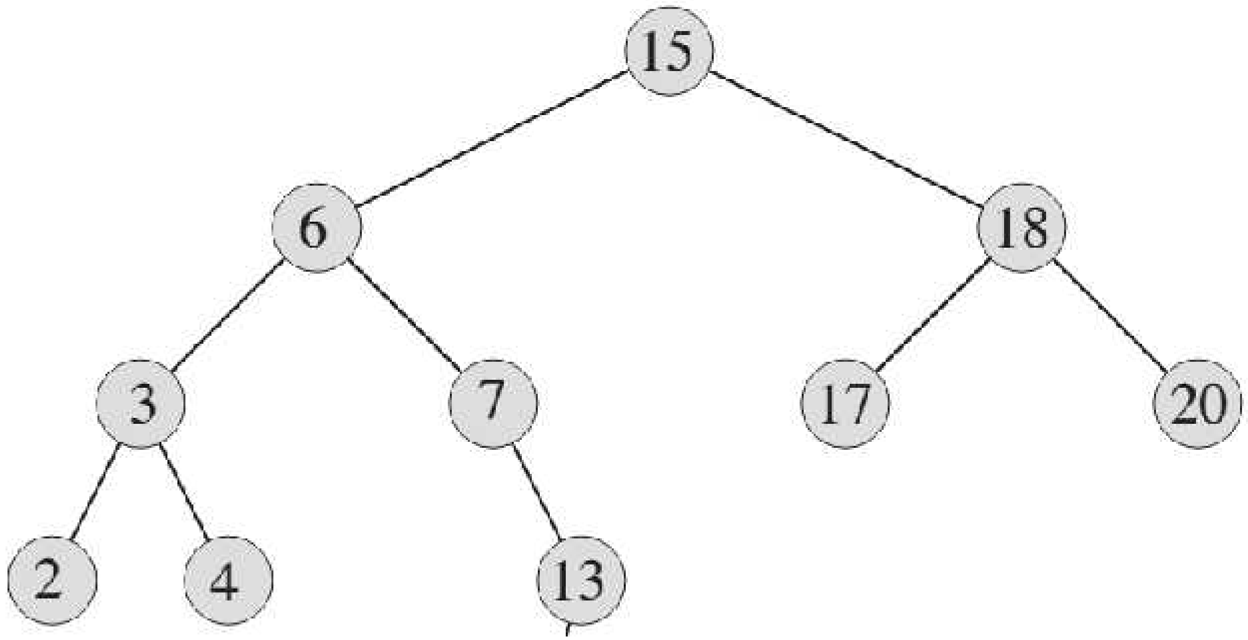
AVL Tree Search

```
//return a pointer to a node with key k if
//one exists; otherwise, return null
Node *tree_search(Node *x, Key k) {
  if (x == nullptr || x->key == k) return x;
  if (k < x->key)
    return tree_search(x->left, k);
  return tree_search(x->right, k);
}
```

- Same as BST
- Trace a downward path starting at the root
- Next node depends on the outcome of the comparison of k with the key of the current node

AVL Tree Search Example

```
tree_search(<ptr to 15>, 9);
```



AVL Tree Sort

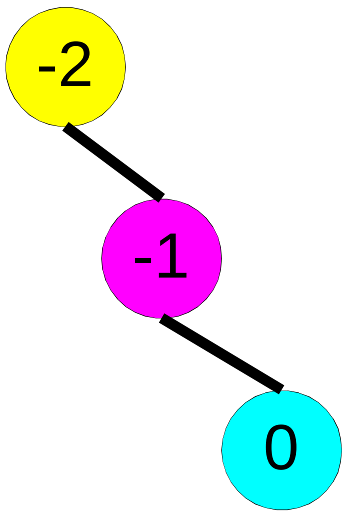
- Same as BST
- Sorting an AVL tree *is* an inorder traversal

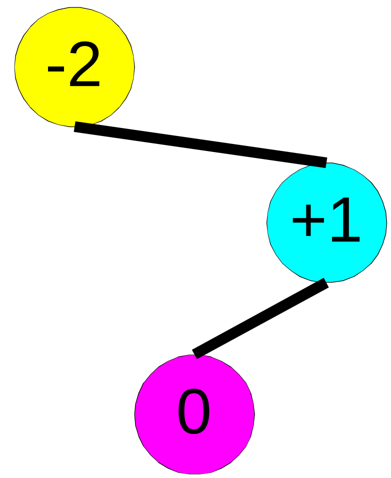
```
void inorder(Node *x) {
  if (!x) return;
  inorder(x->left);
  print(x->key);
  inorder(x->right);
}
```

AVL Tree Insert

- The basic idea:
  1. Insert like a BST
  2. Rearrange tree to balance height
- Each node records its height
- Can compute a node's *balance factor*
  - `bal(n) = height(n->left) – height(n->right)`
- A node that is AVL-balanced:
  - `bal(n) =  0`: both subtrees equal
  - `bal(n) = +1`: left taller by one
  - `bal(n) = –1`: right taller by one
- **`|bal(n)| > 1`**: node is out of balance

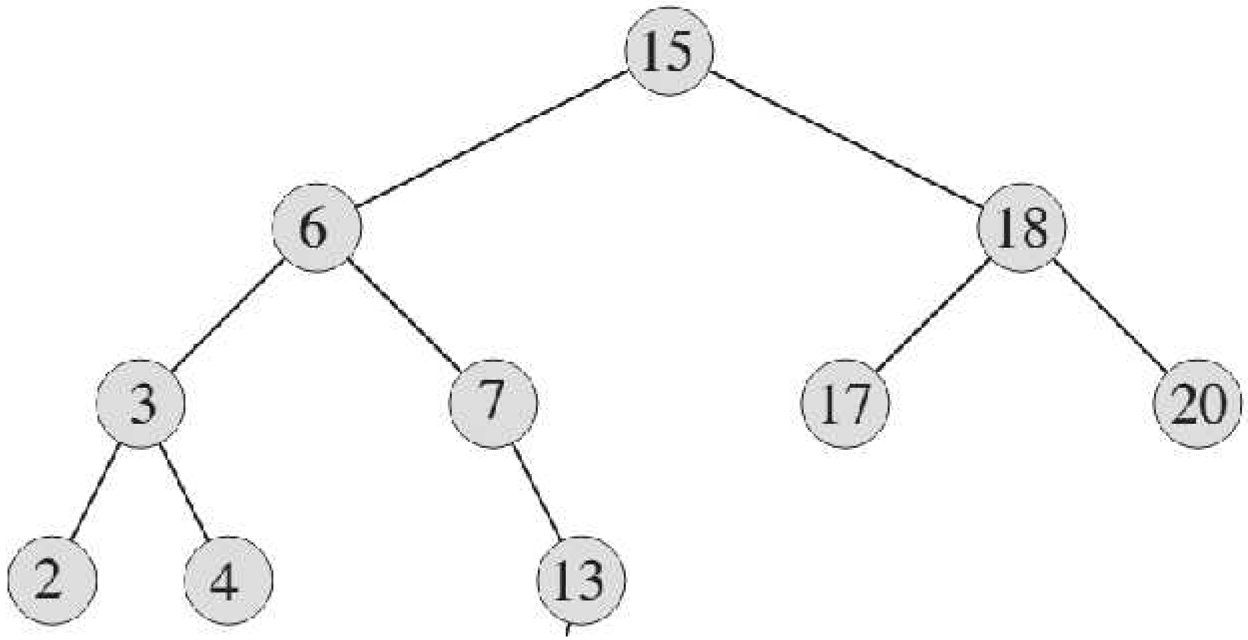
Balance Factor Example





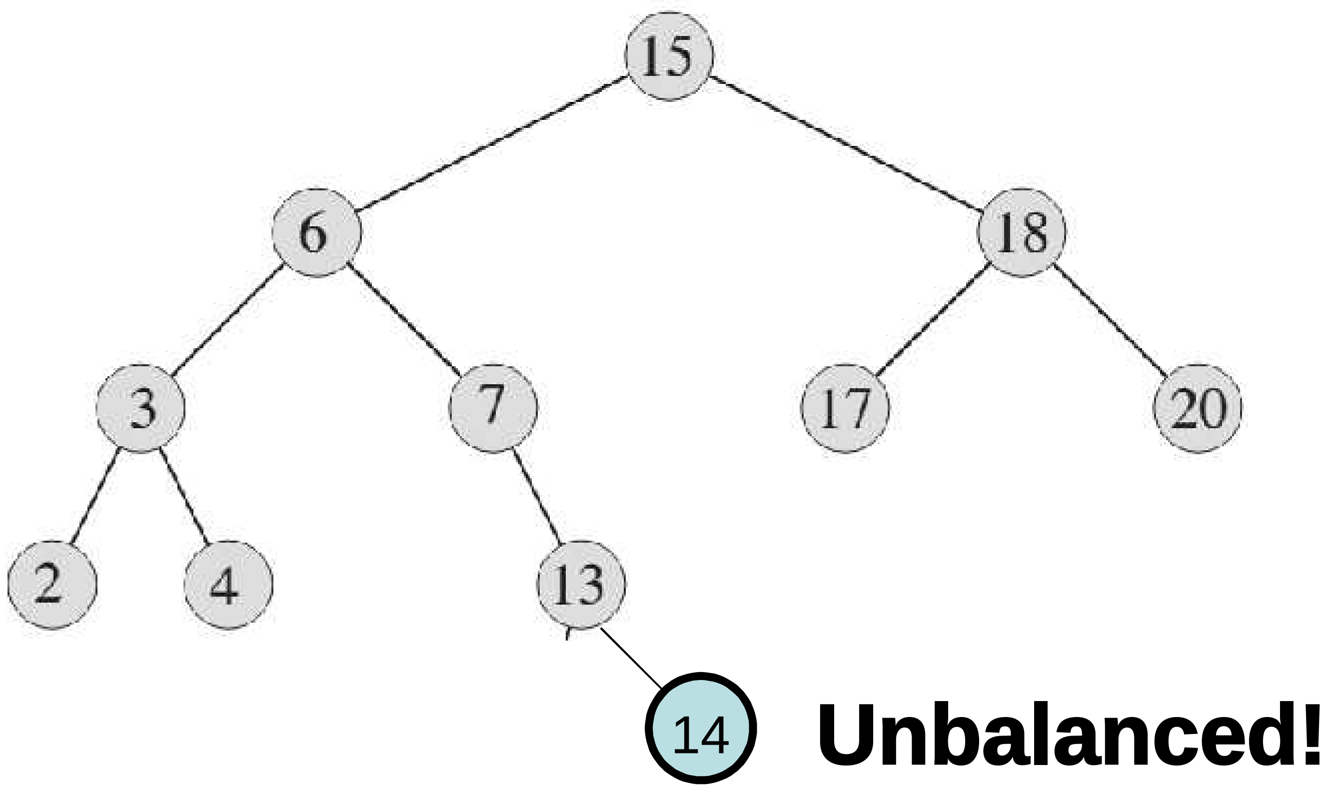
Balance Factor Exercise

Label the balance factor on each node



Insert (begins like BST)

```
tree_insert(14);
```



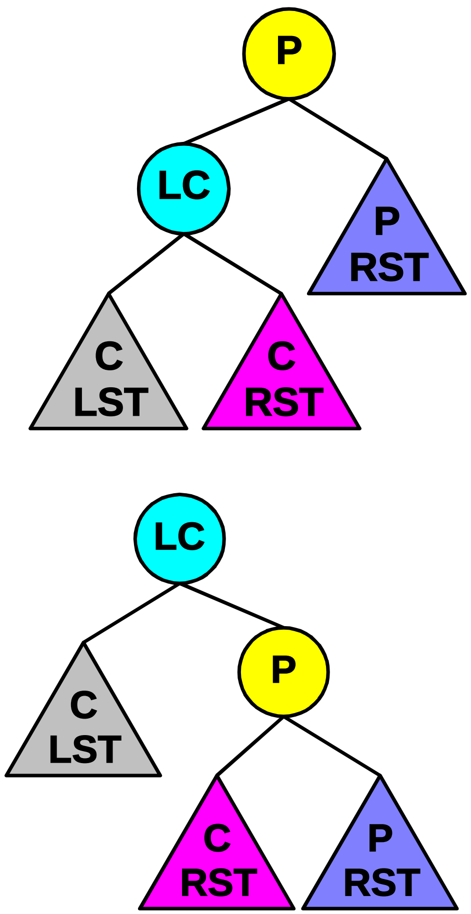
Rotations

- We use *rotations* to rebalance the binary tree
  - Interchange the role of a parent and one of its children in a tree…
  - While still preserving the BST ordering among the keys in the nodes
- The second part is tricky
  - Right rotation: copy the right pointer of the left child to be the left pointer of the old parent
  - Left rotation: copy the left pointer of the right child to be the right pointer of the old parent

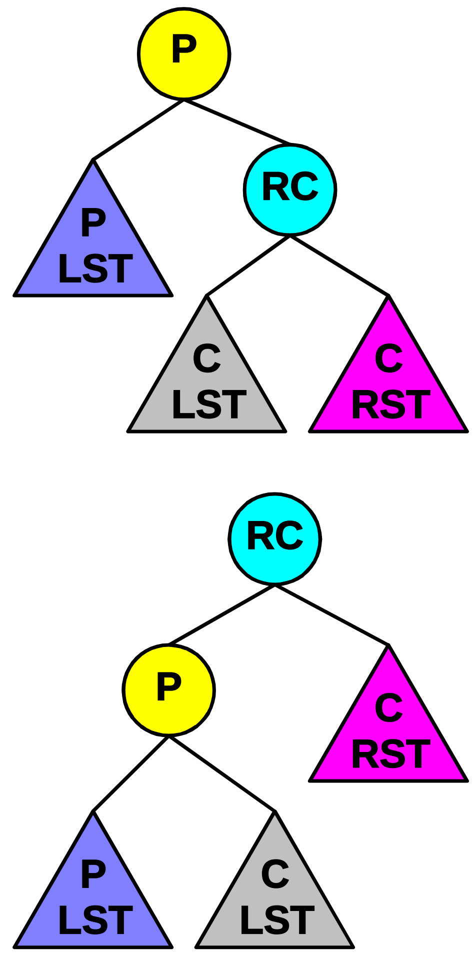
Rotation is a local change involving only three pointers and two nodes

Rotations

Rotate Right: RR(P)

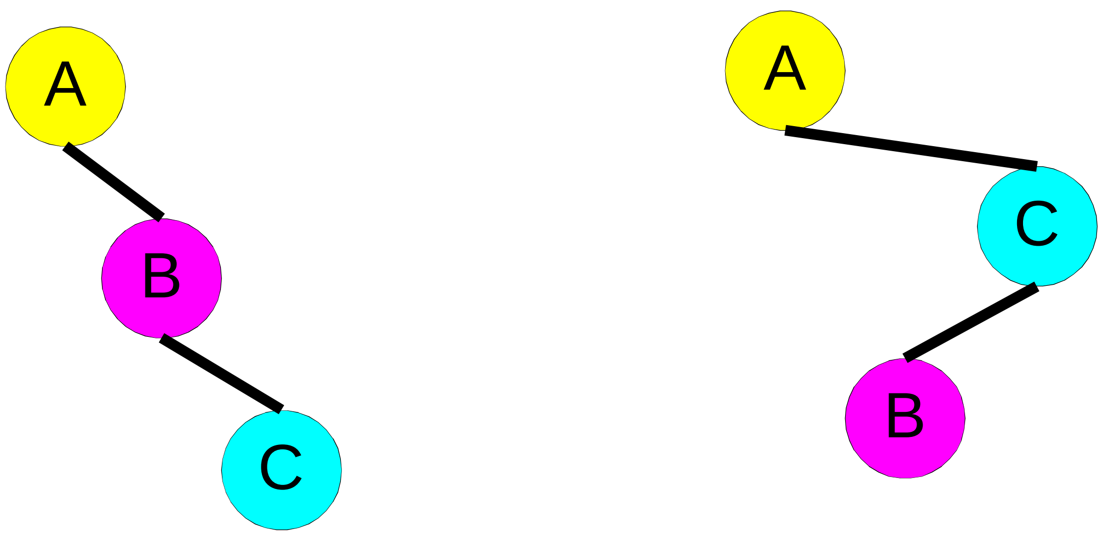


Rotate Left: RL(P)



Rotation Exercise

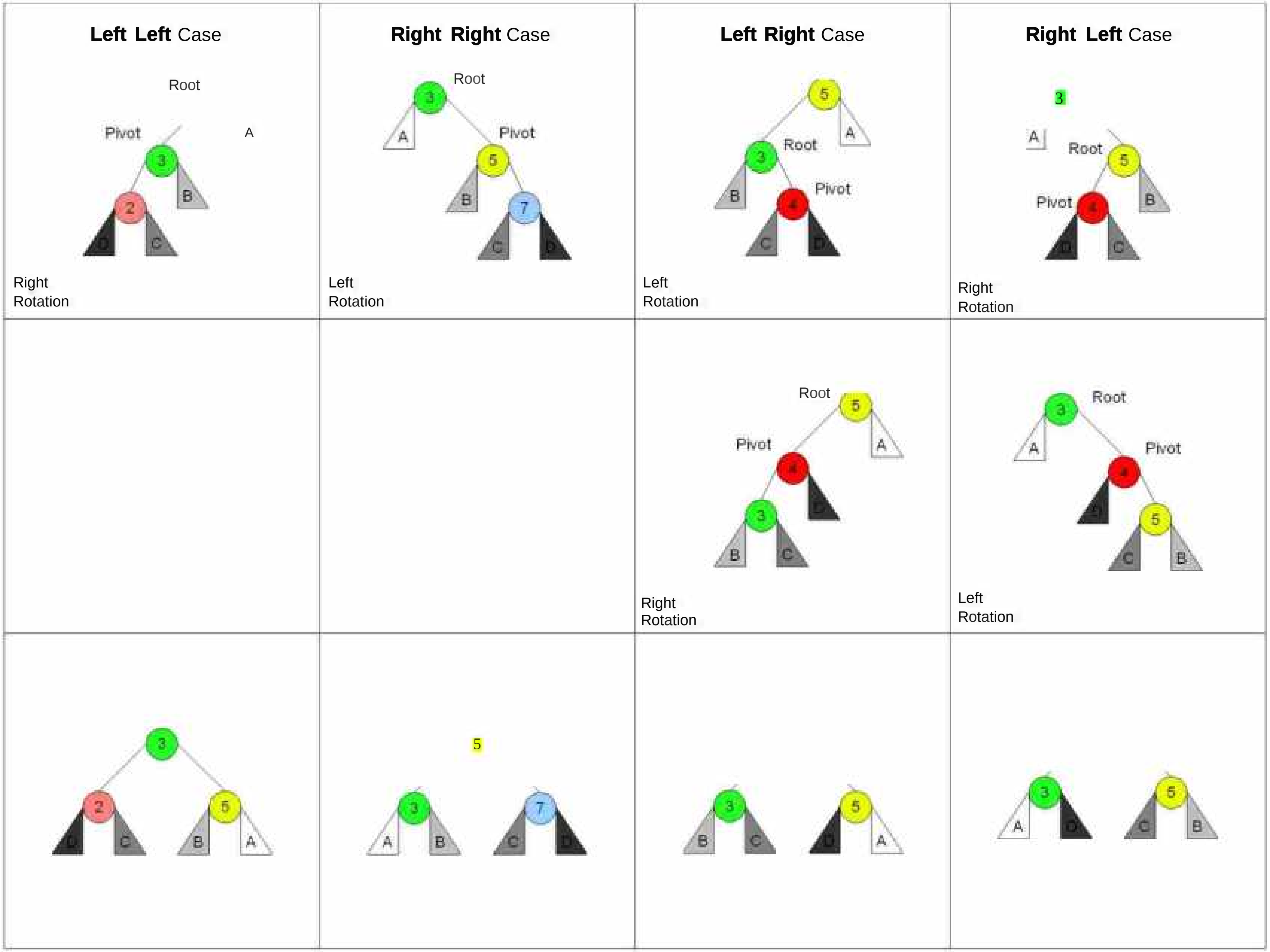
- Use rotations to balance these trees



Insert

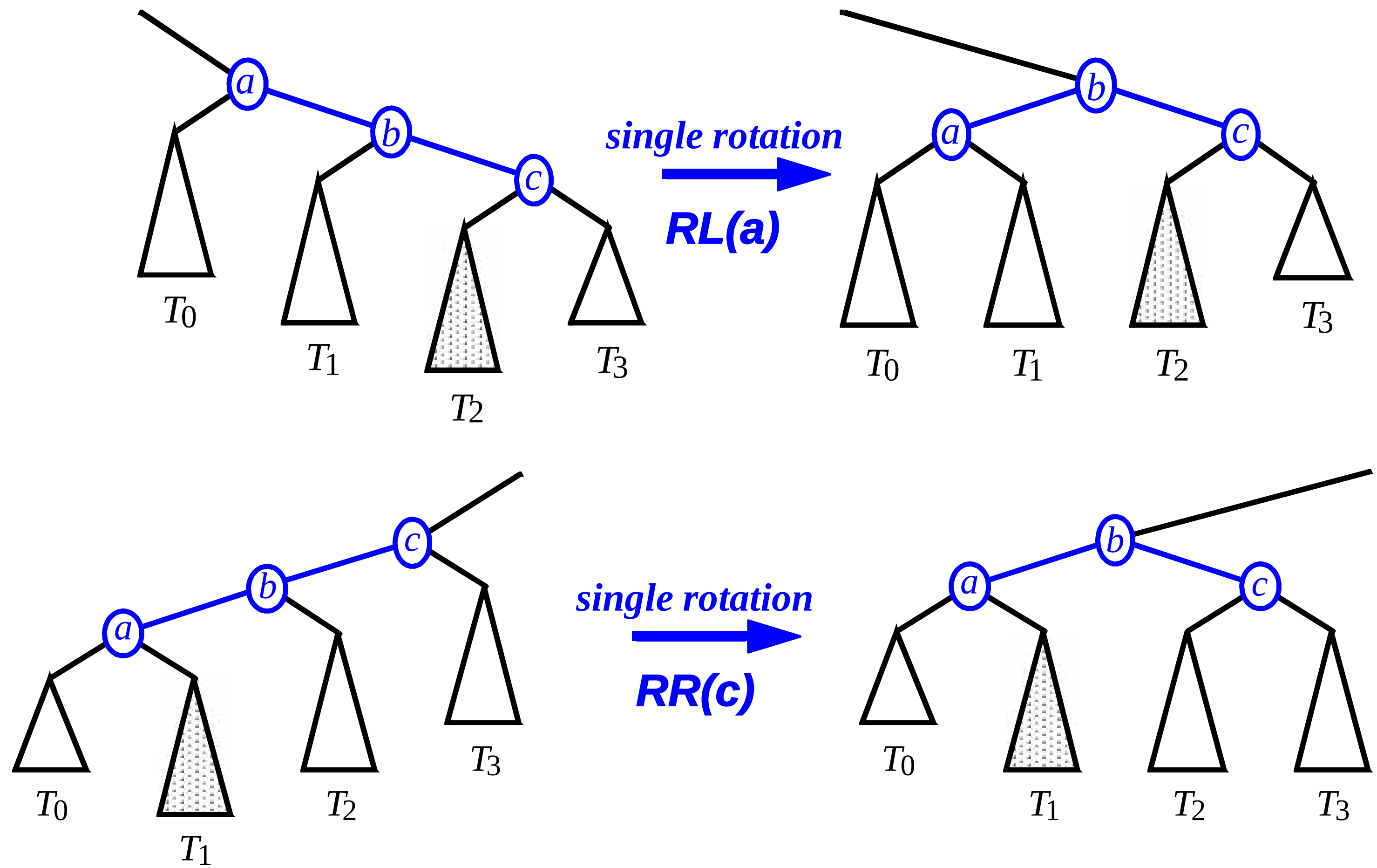
Four Cases

1. Single left rotation
   - RL(a)
2. Single right rotation
   - RR(a)
3. Double rotation
   a. RR(c)
   b. RL(a)
4. Double rotation
   a. RL(a)
   b. RR(c)

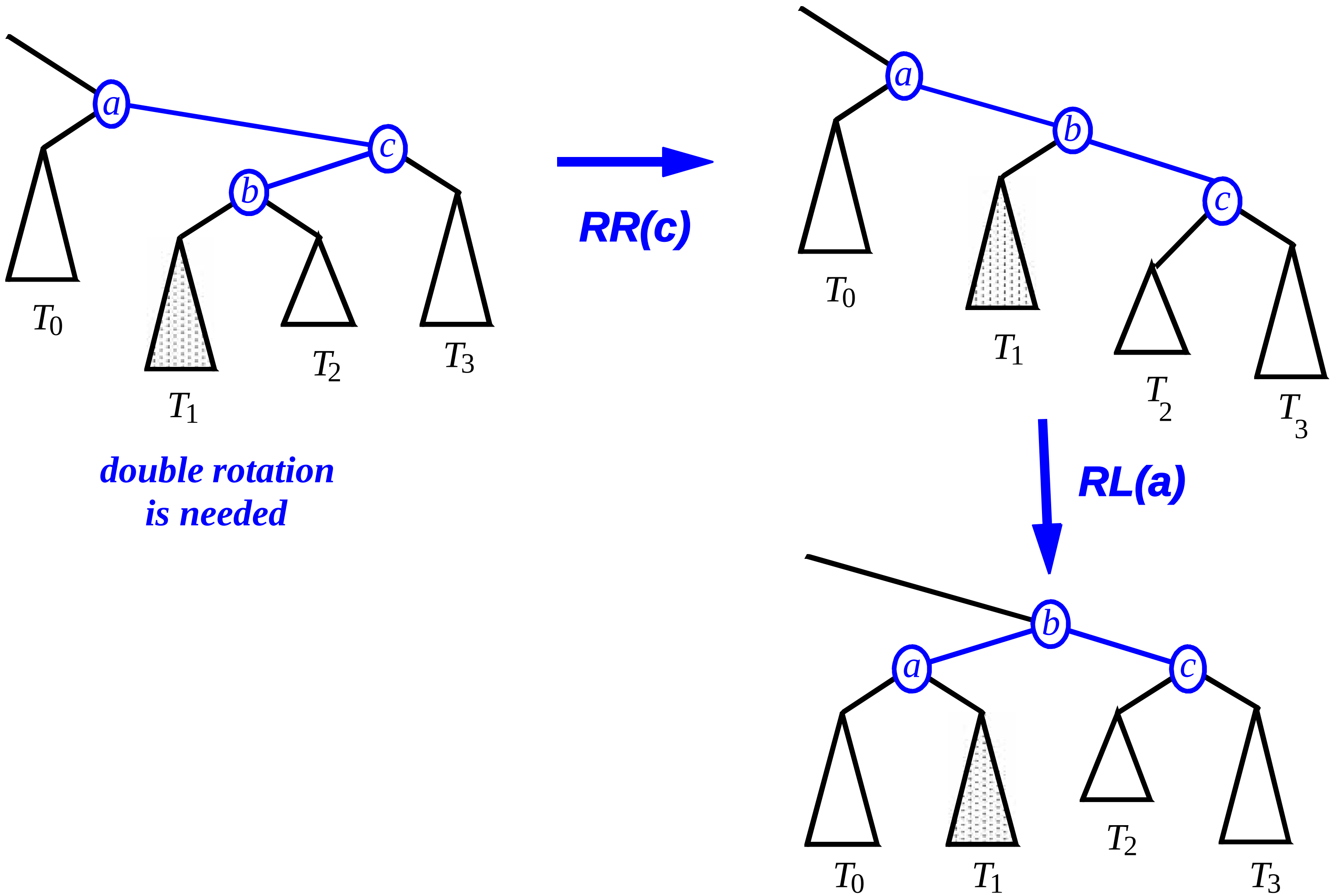


http://upload.wikimedia.org/wikipedia/commons/c/c4/Tree_Rebalancing.gif

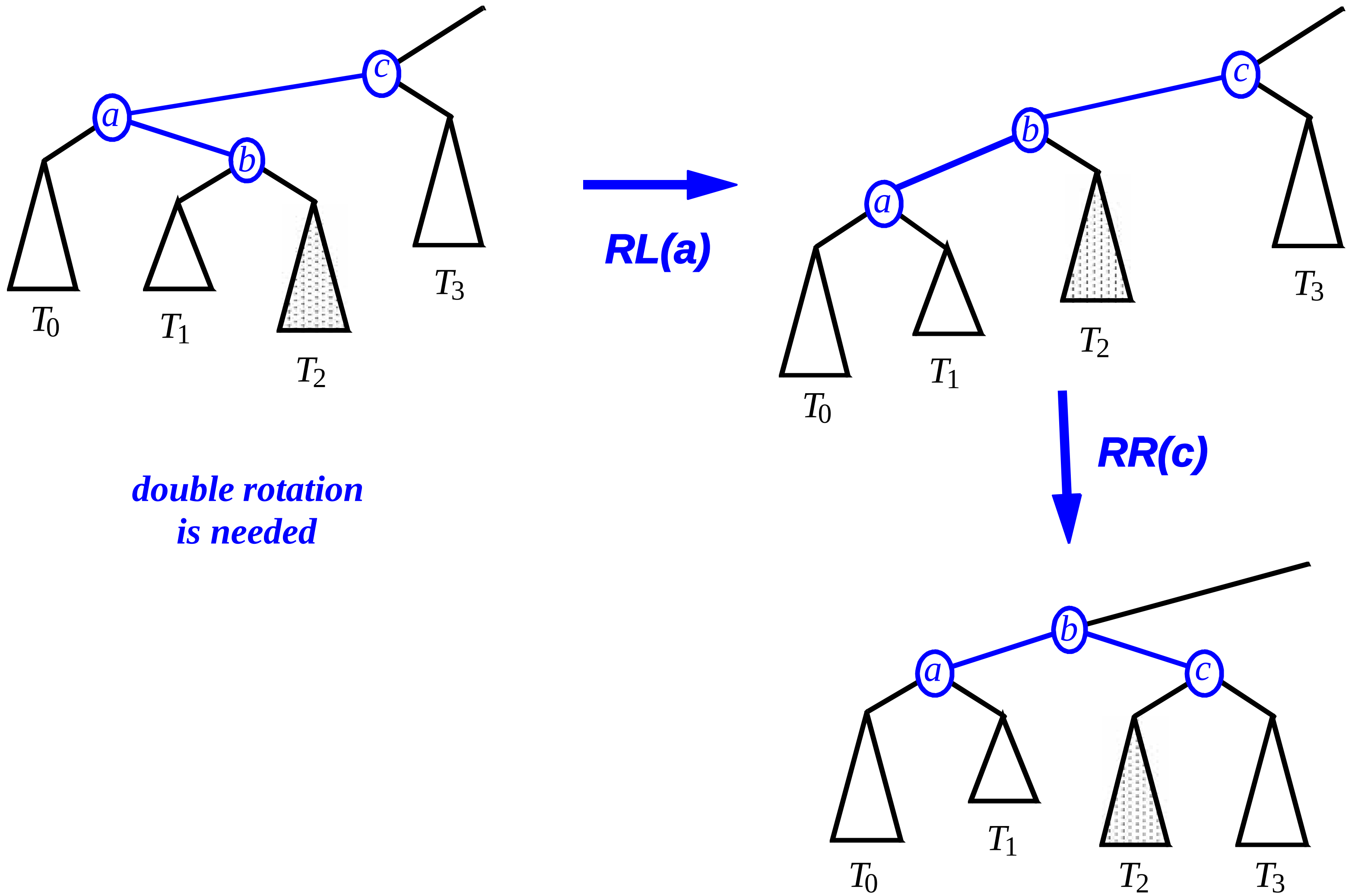
Single Rotations



Double Rotations



Double Rotations



double rotation is needed

Checking and Balancing

```
Algorithm checkAndBal(Node *n)
    if bal(n) > +1
        if bal(n->left) < 0
            rotateL(n->left)
        rotateR(n)
    else if bal(n) < -1
        if bal(n->right) > 0
            rotateR(n->right)
        rotateL(n)
```

- Outermost if:
  Q: Is node out of balance?

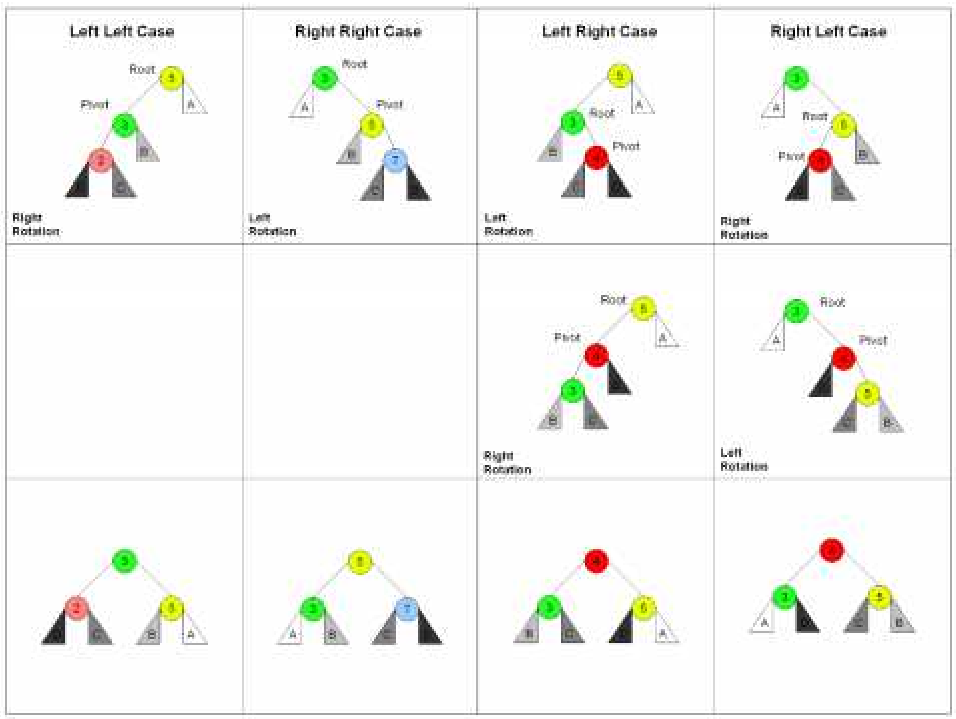
  A: > +1: left too big

  < -1: right too big
- Inner ifs:
  Q: Do we need a double rotation?

  A: only if signs disagree

Rotation Exercise

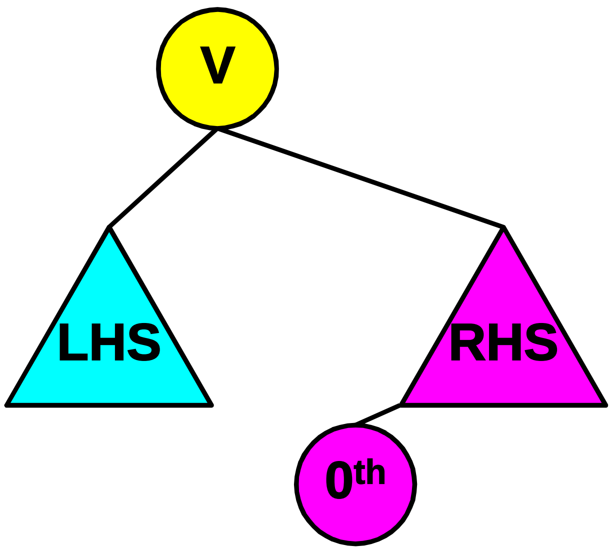
- Insert these keys into an AVL tree, rebalancing when necessary
- 3, 2, 1, 4, 5, 6, 7, 16, 15, 14



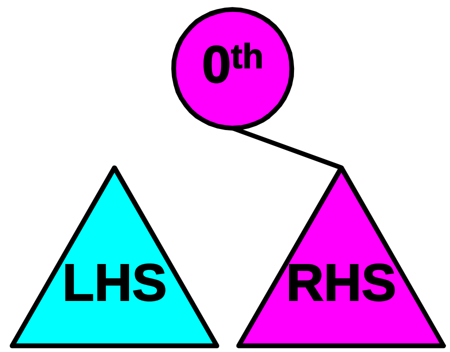
AVL Tree Delete

- The basic idea:
  1. Delete like a BST
  2. Rearrange tree to balance height
- Key observation
  - All keys in LHS <= all in keys RHS
  - Rearrange RHS so that its smallest node is its root
    - Must be some such node, since RHS is not empty
    - New RHS root has a right child, but no left child
  - Make the RHS root’s left child the LHS root

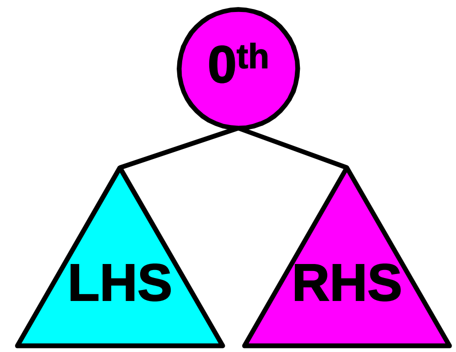
Joining Two Children, Illustrated



Need to
Remove V



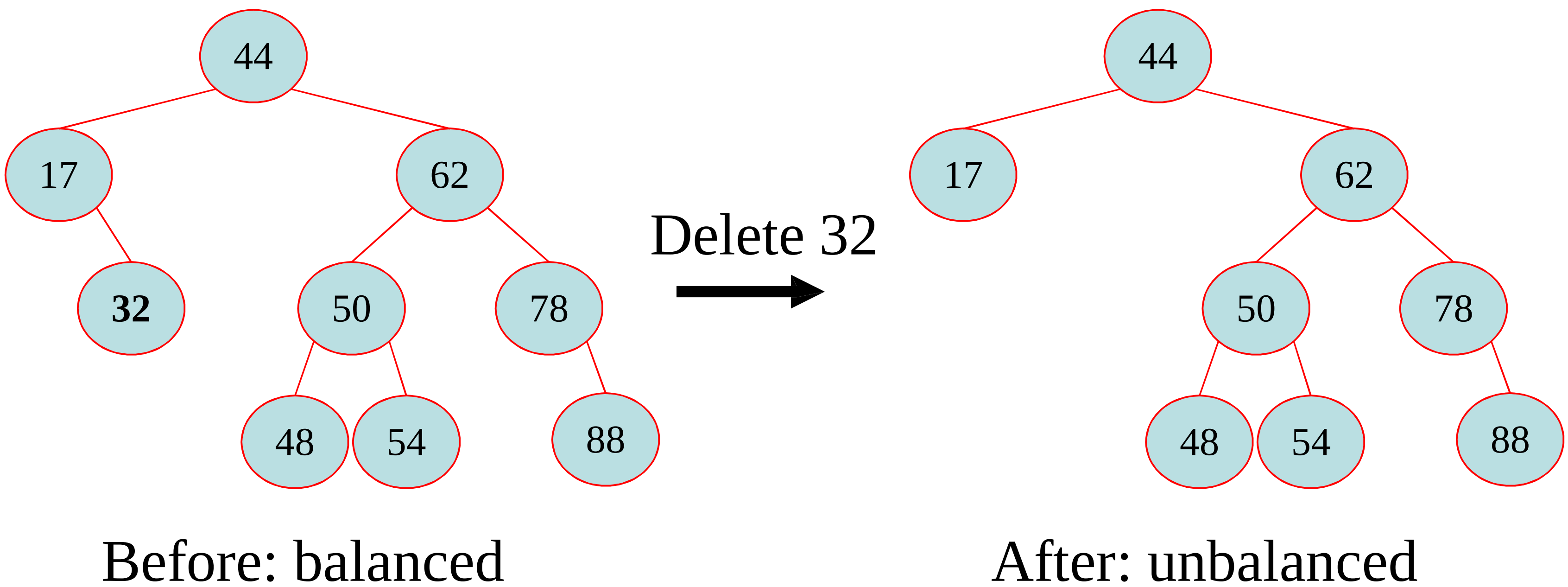
Partition
RHS



Merge Two
Sides

Delete in an AVL Tree

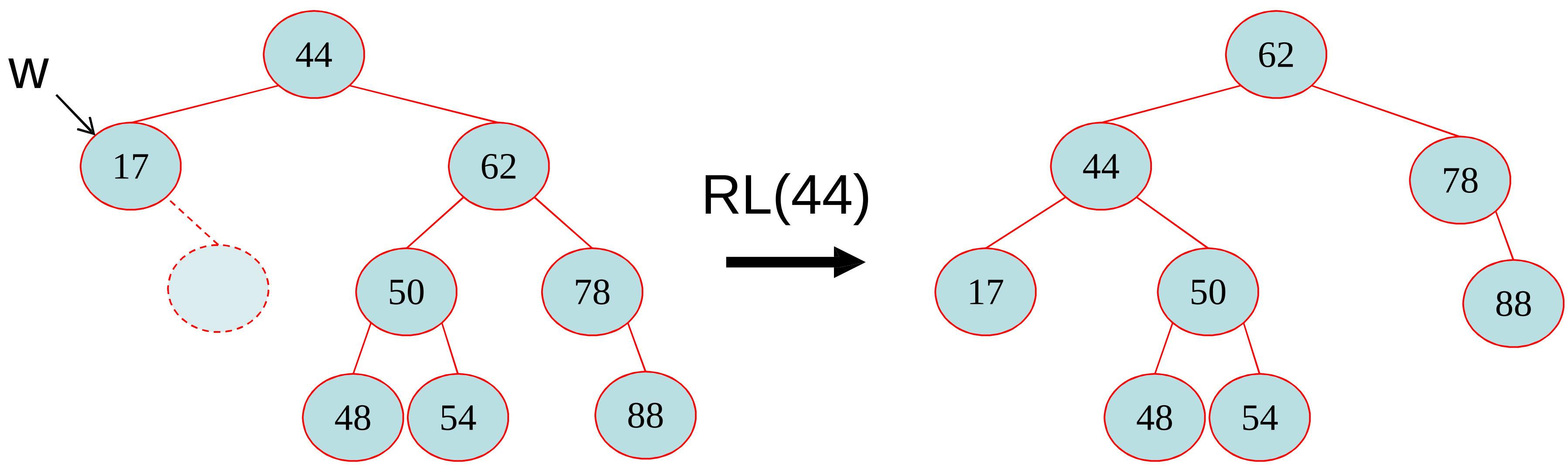
- Delete as in a binary search tree
- Rebalance if an imbalance has been created



Now we rebalance

Rebalancing after a Delete

- Travel up the tree from *w*, the parent of the deleted node
- At the first unbalanced node encountered, rotate as needed
- This restructuring may unbalance one of its ancestors, so we continue checking and rebalancing up to the root



Summary: AVL Trees

- Binary Search Tree
  - Worst case insert or search is $O(n)$
- AVL Tree
  - Worst case insert or search is $O(\log n)$
  - Must guarantee height balance property
- Operations
  - Search: $\boldsymbol{O(\log n)}$ (same algorithm as BST, but faster)
  - Sort: $\boldsymbol{O(n)}$ (same as BST)
  - Insert: $\boldsymbol{O(\log n)}$ (Starts like BST, then may rebalance)
  - Delete: $\boldsymbol{O(\log n)}$ (Starts like BST, then may rebalance)