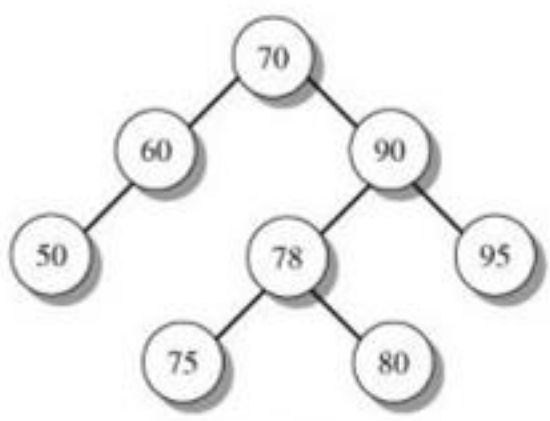
Balanced Search Trees

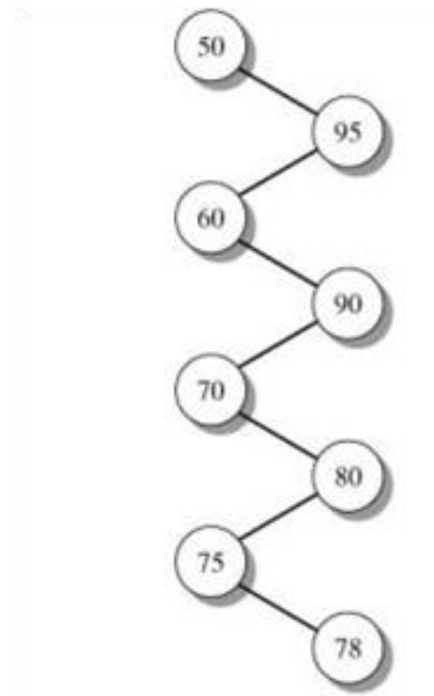
Contents

1.What is a Balanced Binary Search Tree?

- A balanced search tree is one where all the branches from the root have almost the same height.



balanced



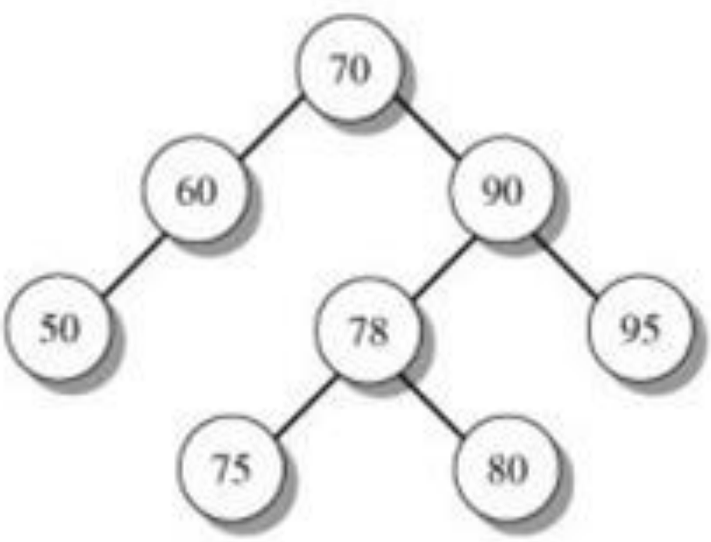
unbalanced

continued

- As a tree becomes more unbalanced, search running time decreases from O(log n) to O(n)
  - because the tree shape turns into a list
- We want to keep the binary search tree balanced as nodes are added/removed, so searching/insertion remain fast.

1.1. Balanced BSTs: AVL Trees

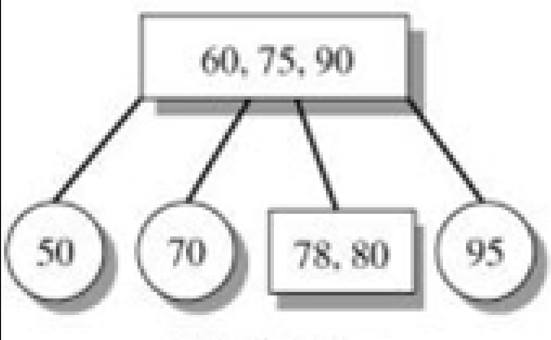
- An AVL tree maintains *height balance*
  - for each node, the difference in height of its two subtrees is in the range -1 to 1



AVL Tree

1.2. 2-3-4 Trees

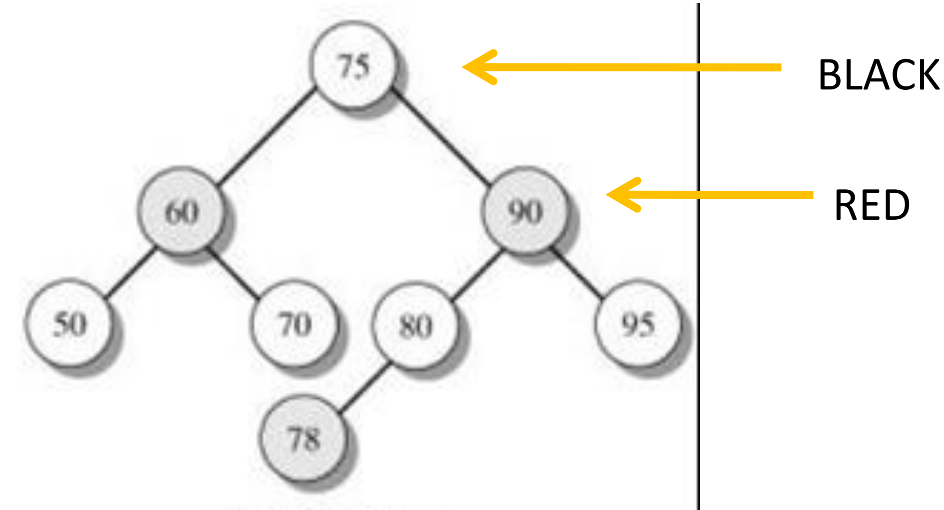
- A *multiway tree* where each node has at most 4 children, and a node can hold up to 3 values.
- A 2-3-4- tree can be perfectly balanced
  - no difference in height between branches
  - requires complex nodes and links



2-3-4 Tree

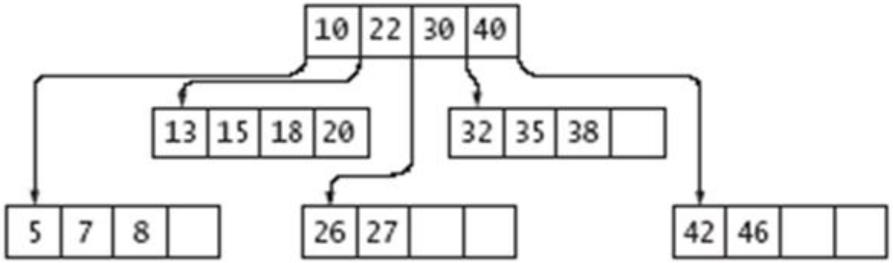
1.3. Red-Black Trees

- A red-black tree is a binary version of a 2-3-4 tree
  - the nodes have a 'color' attribute: BLACK or RED
  - the tree maintains a balance measure called the *BLACK height*



Red-Black Tree

1.4. B-Trees



- A multiway tree where each node has at most m children, and a node can hold up to m-1 values
  - a more general version of a 2-3-4 tree
- B-Trees are most commonly used in databases and filesystems
  - most nodes are stored in secondary storage such as hard drives

AVL Trees

- For each AVL tree node, the difference between the heights of its left and right subtrees is either -1, 0 or +1
  - this is called the *balance factor* of a node
    - balanceFactor =
      height(left subtree) - height(right subtree)

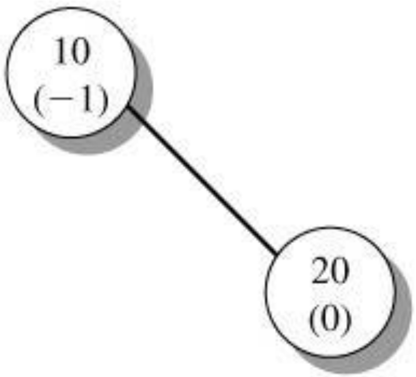
L - R

  - if balanceFactor > 1 or < -1 then the tree is too unbalanced, and needs 'rearranging' to make it more balanced

- *Heaviness*
  - if the balanceFactor is positive, then the node is "heavy on the left"
    - the height of the left subtree is greater than the height of the right subtree
  - a negative balanceFactor, means the node is "heavy on the right"

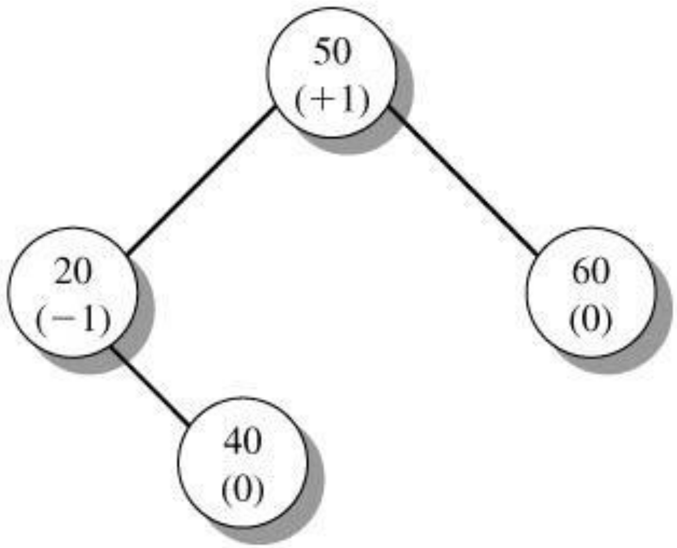
continued

L – R
= 0-1



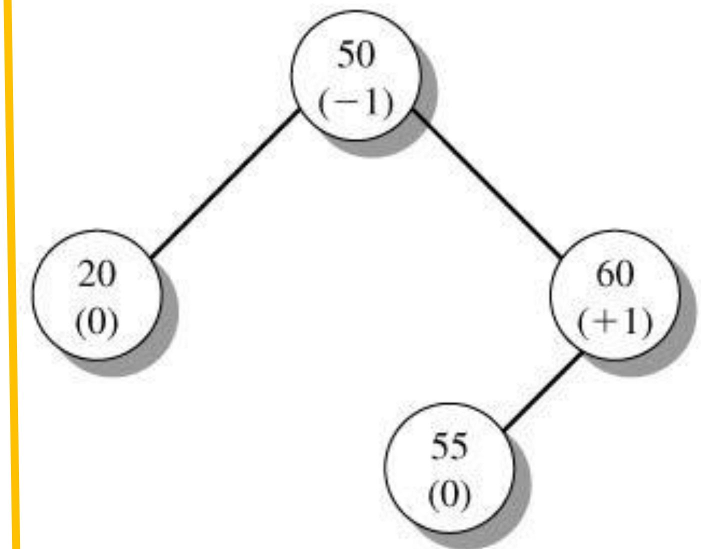
root is
heavy on
the right,
but still
balanced

L – R
= 2-1



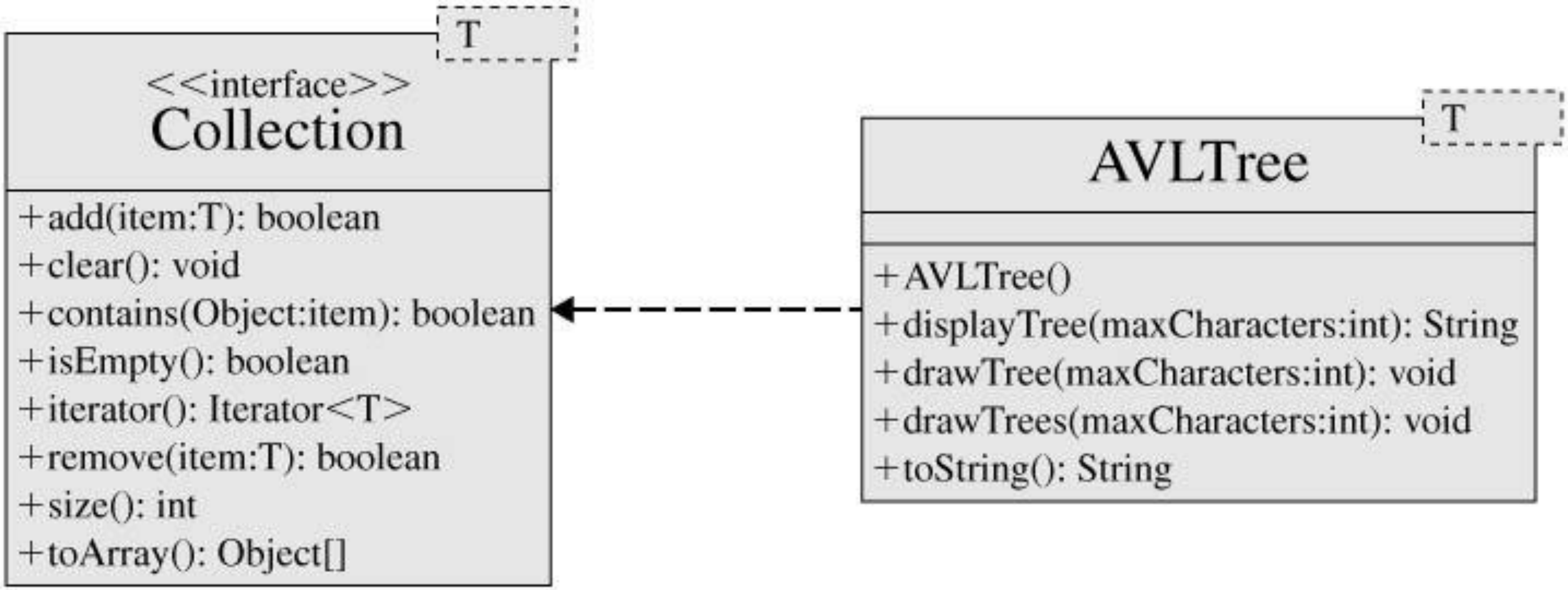
root is
heavy on
the left,
but still
balanced

L – R
= 1-2



root is
heavy on
the right,
but still
balanced

The AVLTree Class

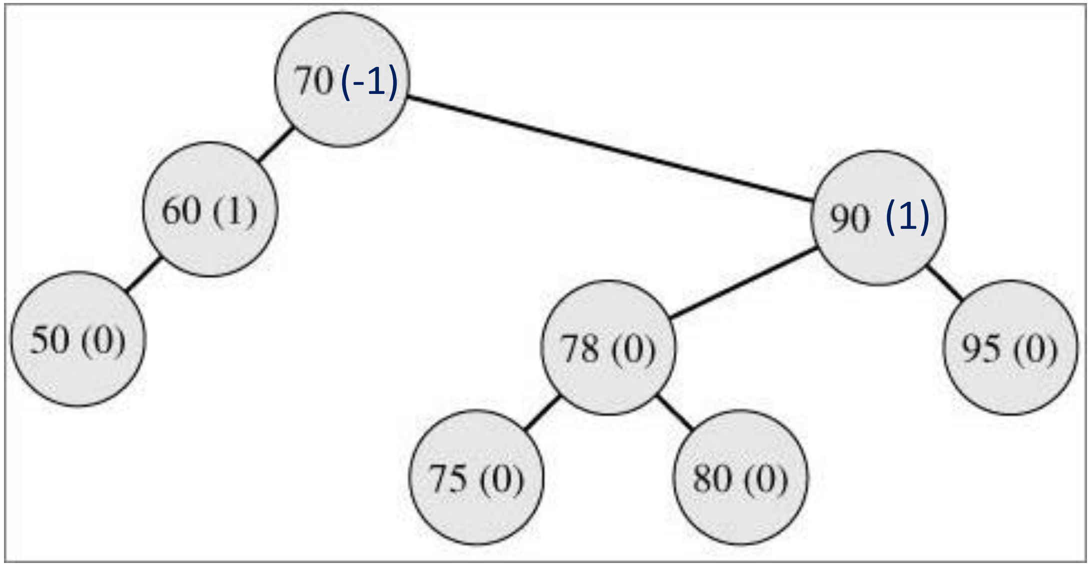


Using AVLTree

```
String[] stateList = {"NV", "NY", "MA", "CA", "GA"};
AVLTree<String> avltreeA = new AVLTree<String>();
for (int i = 0; i < stateList.length; i++)
  avltreeA.add(stateList[i]);

System.out.println("States: " + avltreeA);


int[] arr = {50, 95, 60, 90, 70, 80, 75, 78};
AVLTree<Integer> avltreeB = new AVLTree<Integer>();
for (int i = 0; i < arr.length; i++)
  avltreeB.add(arr[i]);

// display the tree
System.out.println(avltreeB.displayTree());
avltreeB.drawTree();
```

Execution

```
States: [CA, GA, MA, NV, NY]

                    70(-1)
       60(1)                          90(1)
  50(0)                  78(0)                  95(0)
                   75(0)      80(0)
```

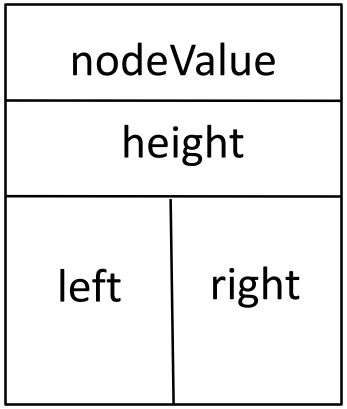


root is heavy on the right, but still balanced

The AVLTree Node

- An AVLNode contains the node's value, references to the node's two subtrees, and the node height.

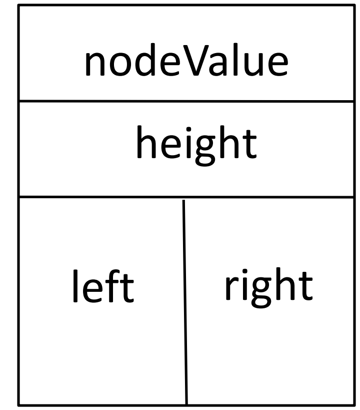
height(node) = max (height(node.left), height(node.right)) + 1;



AVLTreeNode object

continued

```
private static class AVLNode<T>
{
   public T nodeValue;    // node data

   public int height;

   // child links
   public AVLNode<T> left, right;

   public AVLNode (T item)
   {  nodeValue = item;
      height = 0;
      left = null;  right = null;
   }
}
```



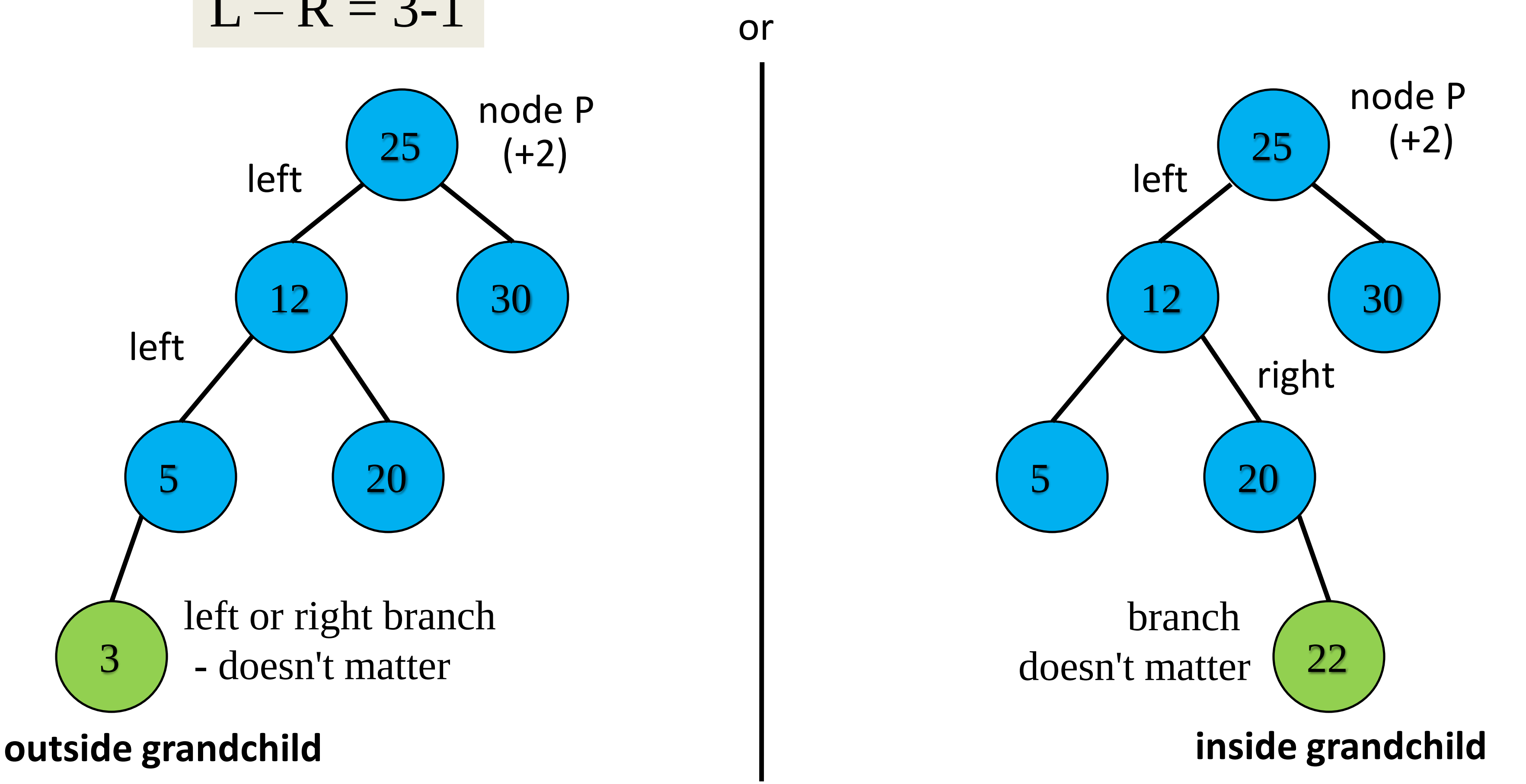
Adding a Node to the Tree

- The addition of a node may cause the tree to become unbalance.
  - addNode() adds a node *and* may reorder nodes
    - reordering is the new idea in AVL trees
  - the reordering is done using *single and double rotations*

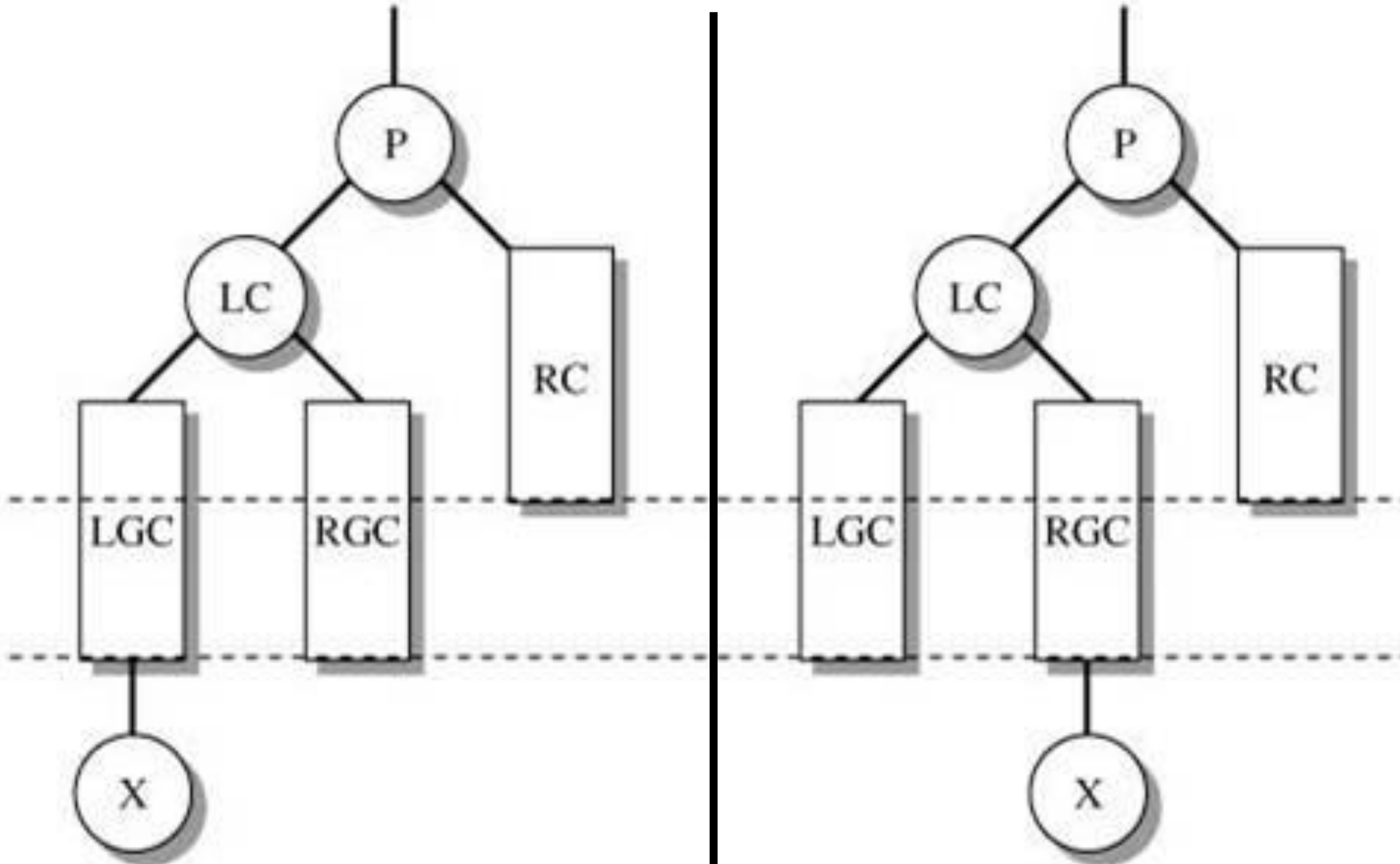
1. Too Heavy on the Left

Two cases

L – R = 3-1



continued



(a) Insert in left (outside) grandchild
Left subtree of LC

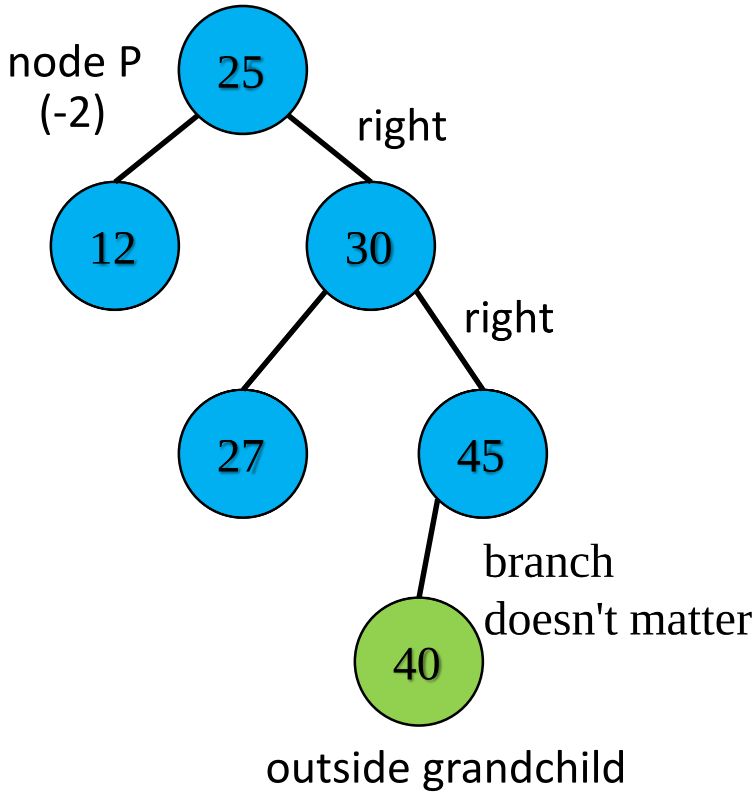
(b) Insert in right (inside) grandchild
Right subtree of LC

Inserting X imbalances the parent node P with balance factor 2.

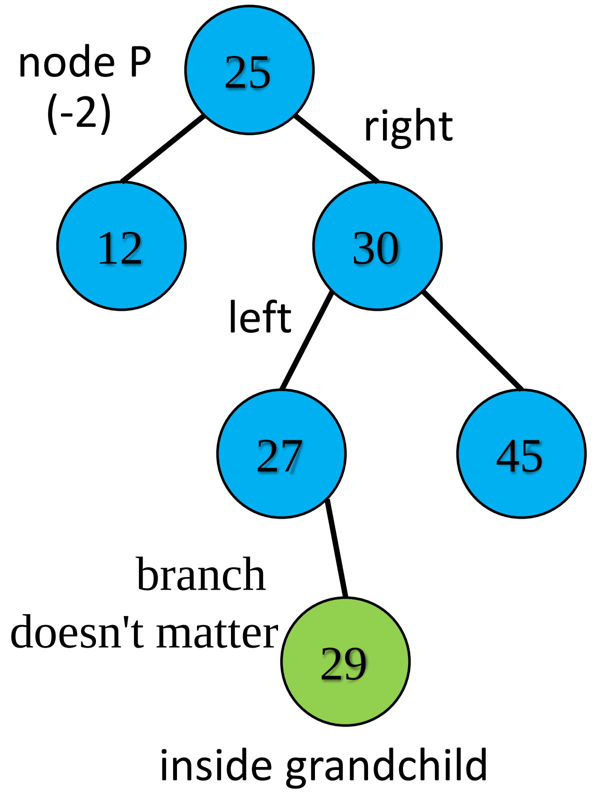
2. Too Heavy on the Right

Two cases

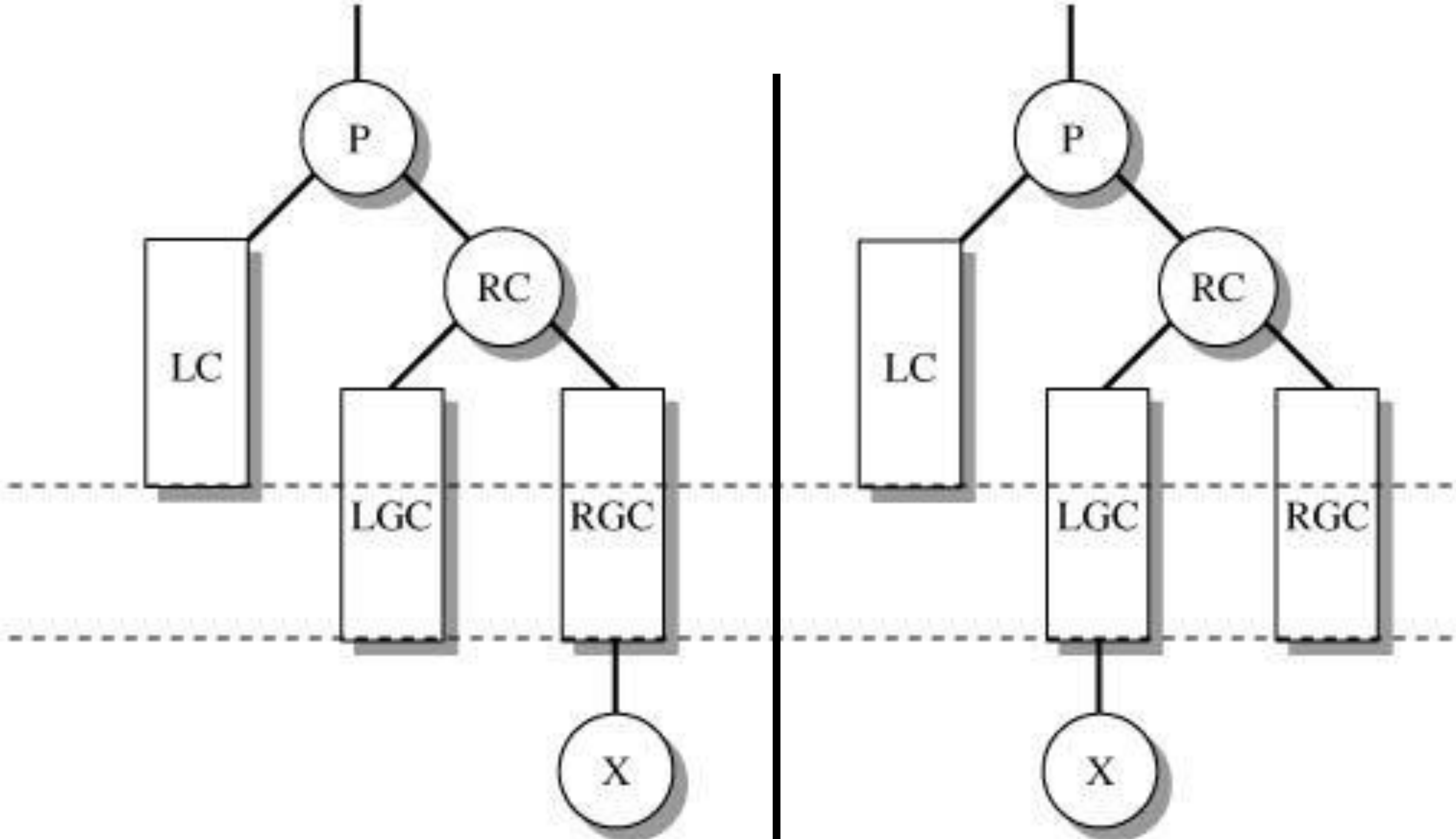
$L - R = 1\text{-}3$



or



continued



(a) Insert in right (outside) grandchild
Right subtree of RC

(b) Insert in left (inside) grandchild
Left subtree of RC

Inserting X imbalances the parent node P with balance factor – 2.

Single Rotations

- When a new item is added to the subtree for an *outside grandchild*, the imbalance is fixed with a single right or left rotation

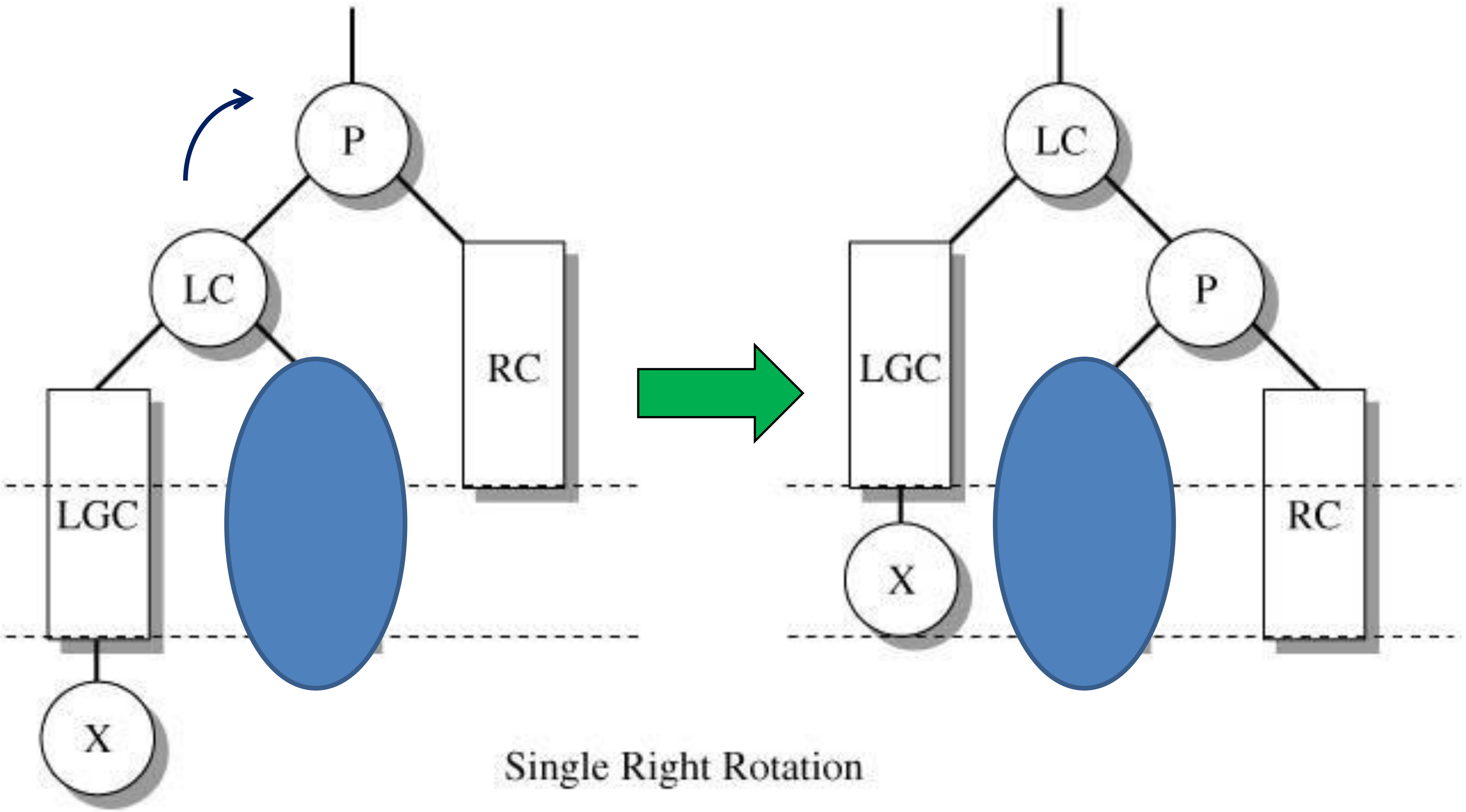
- Two cases:
  - left outside grandchild (left-left) -->
    single right rotation
  - right outside grandchild (right-right) -->
    single left rotation

1. Single Right Rotation

- A single right rotation occurs when a new element is added to the subtree of the *left outside grandchild* (left-left)

continued

Single Right Rotation

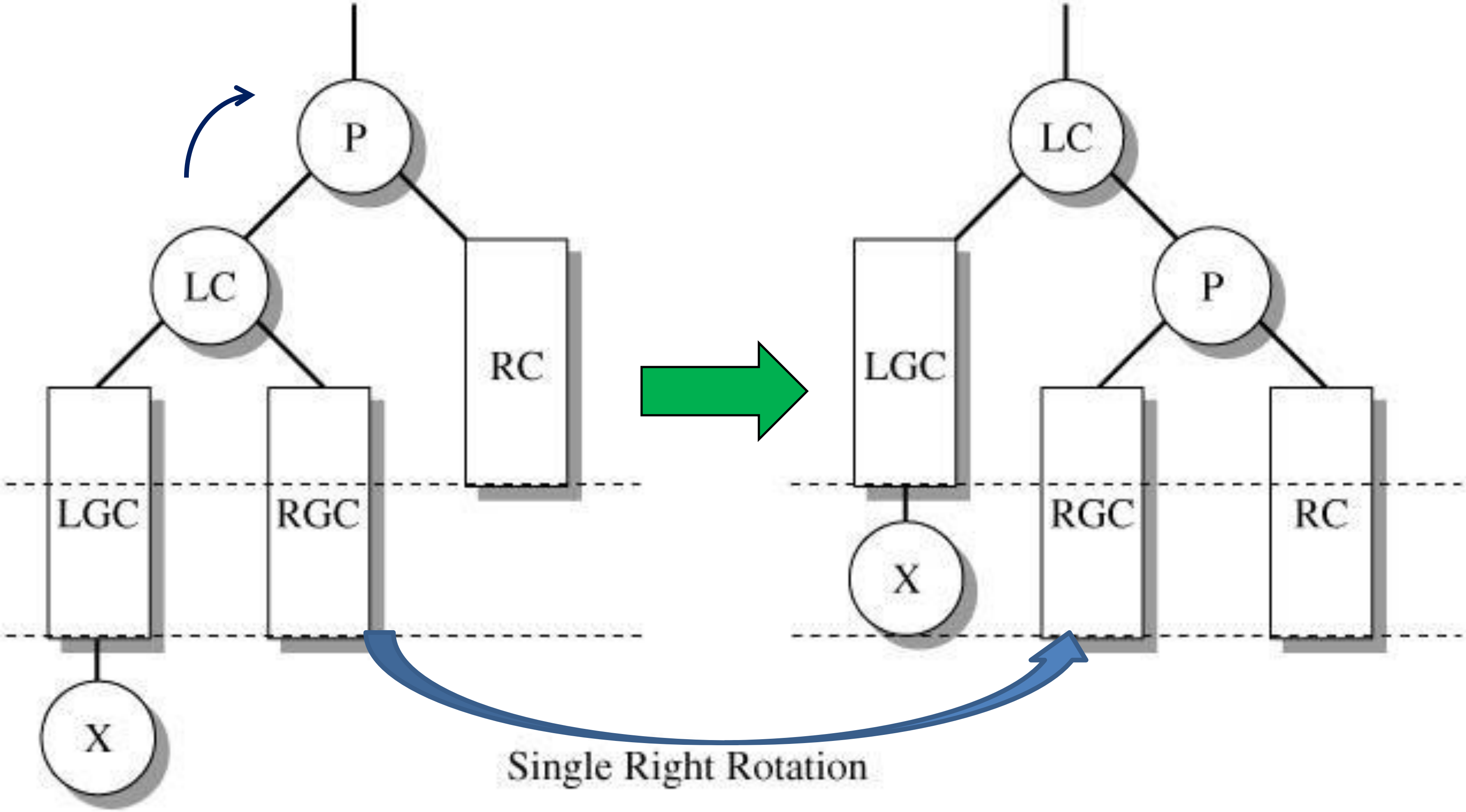


Single Right Rotation

left outside grandchild (left-left)

continued

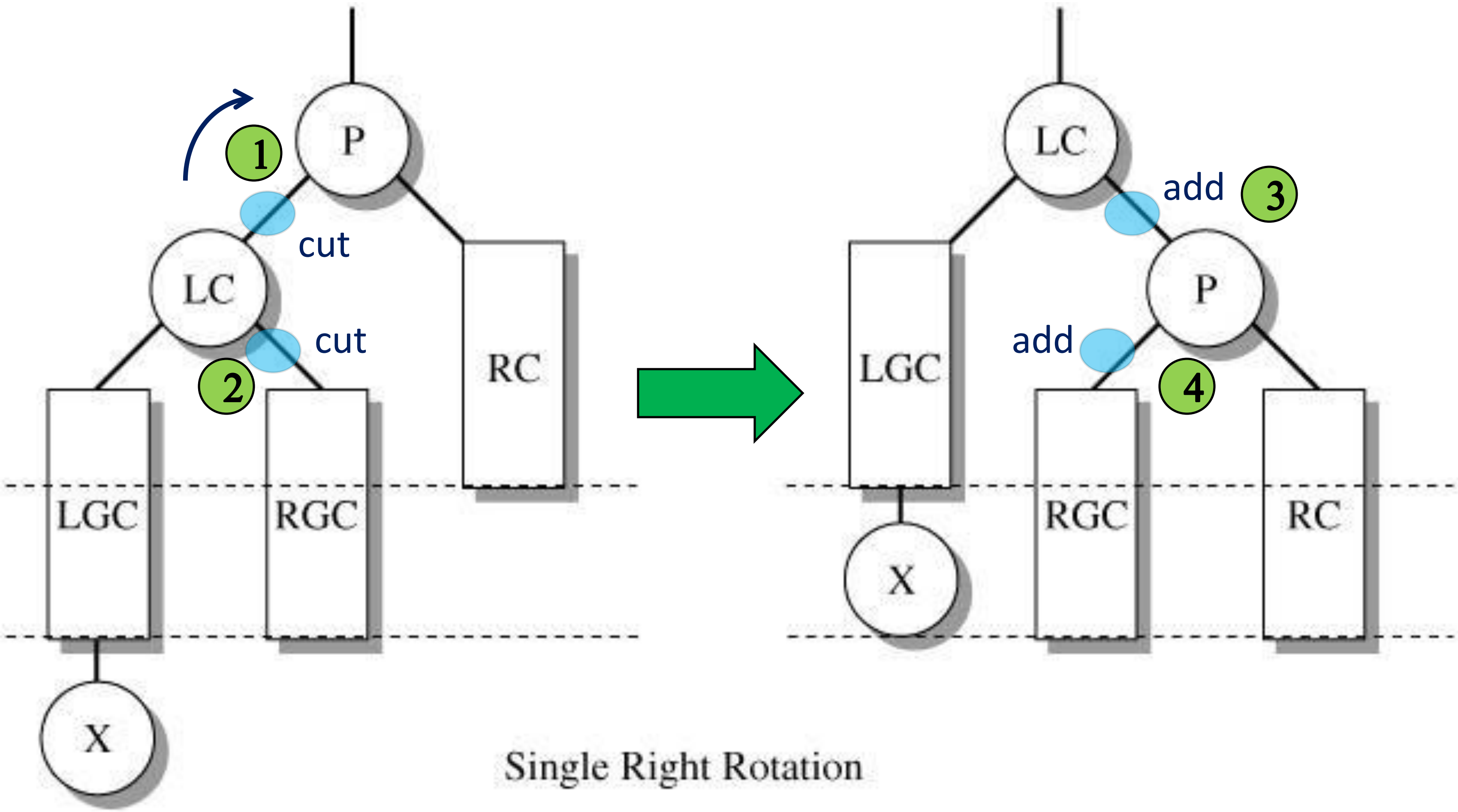
Single Right Rotation



left outside grandchild (left-left)

continued

Single Right Rotation



Single Right Rotation

left outside grandchild (left-left)

continued

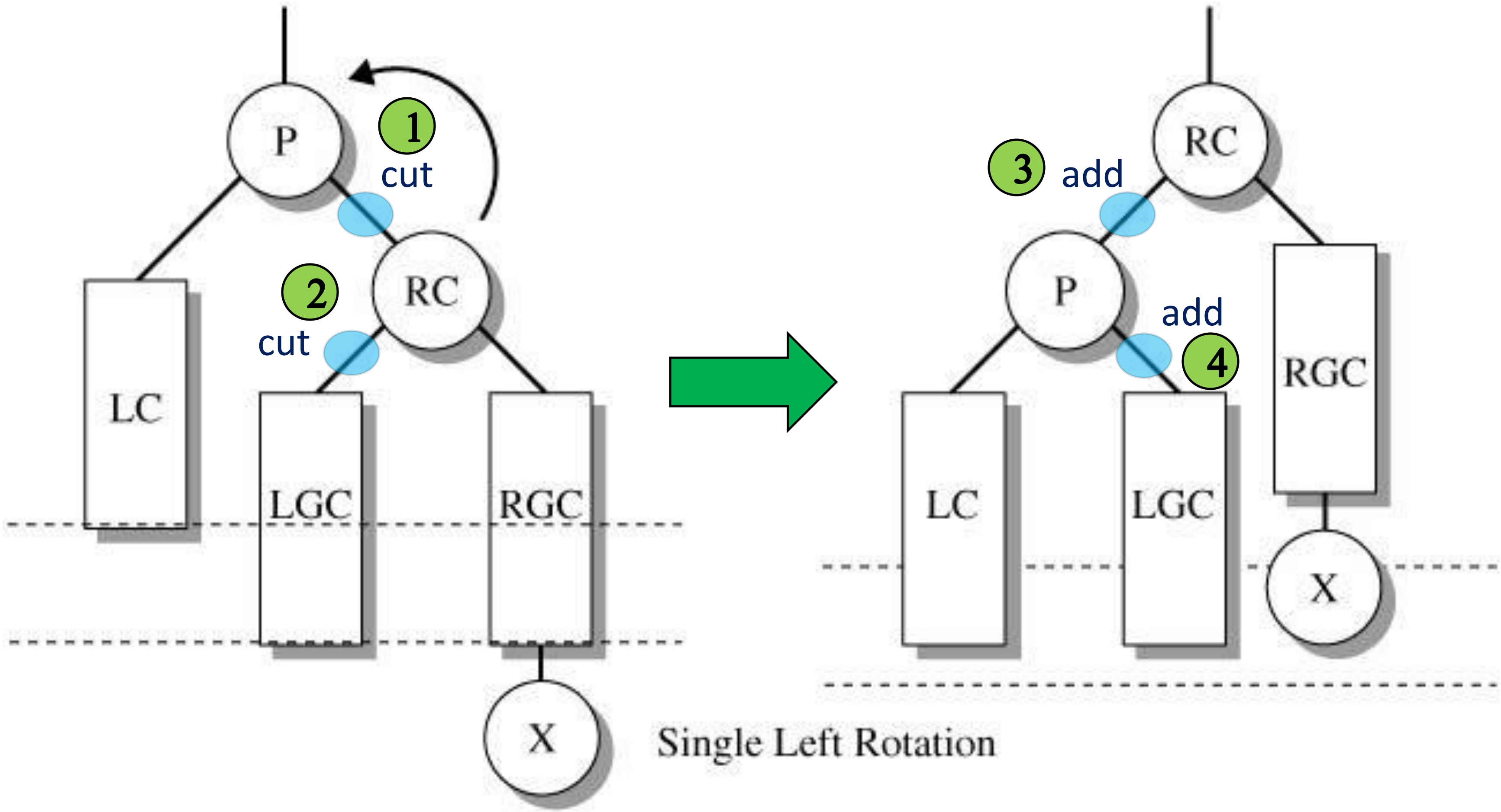
singleRotateRight()

```
// single right rotation on p

private AVLNode<T> singleRotateRight( AVLNode<T> p)
{
   AVLNode<T> lc = p.left;

   p.left = lc.right;   // 1 & 4
   lc.right = p;        // 2 & 3

   p.height = max(height(p.left), height(p.right)) + 1;
   lc.height = max(height(lc.left),
                   height(rc.right)) + 1;

   return lc;
}
```

2. Single Left Rotation

- A single left rotation occurs when a new element is added to the subtree of the *right outside grandchild* (right-right).

Single left Rotation



right outside grandchild (right-right)

singleRotateLeft()

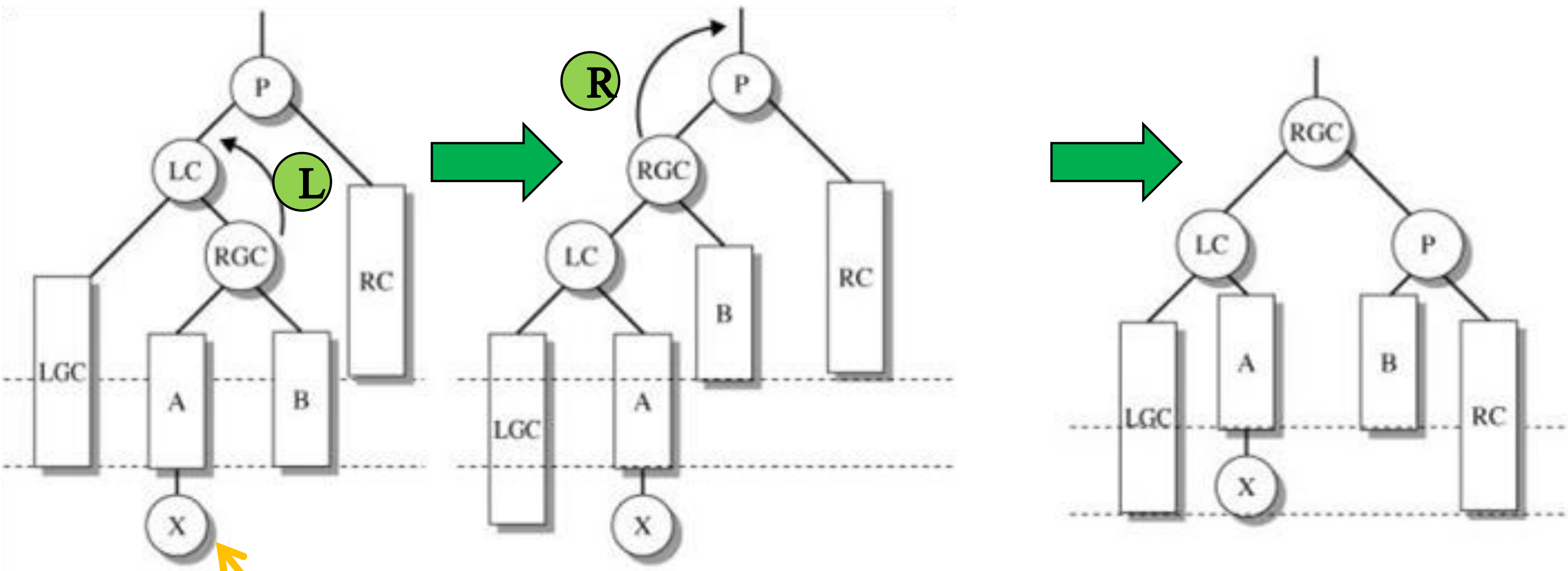
```
// single left rotation on p

private AVLNode<T> singleRotateLeft(AVLNode<T> p)
{
   AVLNode<T> rc = p.right;

   p.right = rc.left;     // 1 & 4
   rc.left = p;           // 2 & 3

   p.height = max(height(p.left),height(p.right)) + 1;
   rc.height = max(height(rc.left),
                   height(rc.right)) + 1;

   return rc;
}
```

Double Rotations

- When a new item is added to the subtree for an *inside grandchild*, the imbalance is fixed with a double right or left rotation
  - a double rotation is two single rotations

- Two cases:
  - Left-inside grandchild (left-right) -->
    double left-right rotation
  - Right-inside grandchild (right-left) -->
    double right-left rotation

1. A Double Left-Right Rotation



Single left rotation about LC

Single right rotation about P

balanced

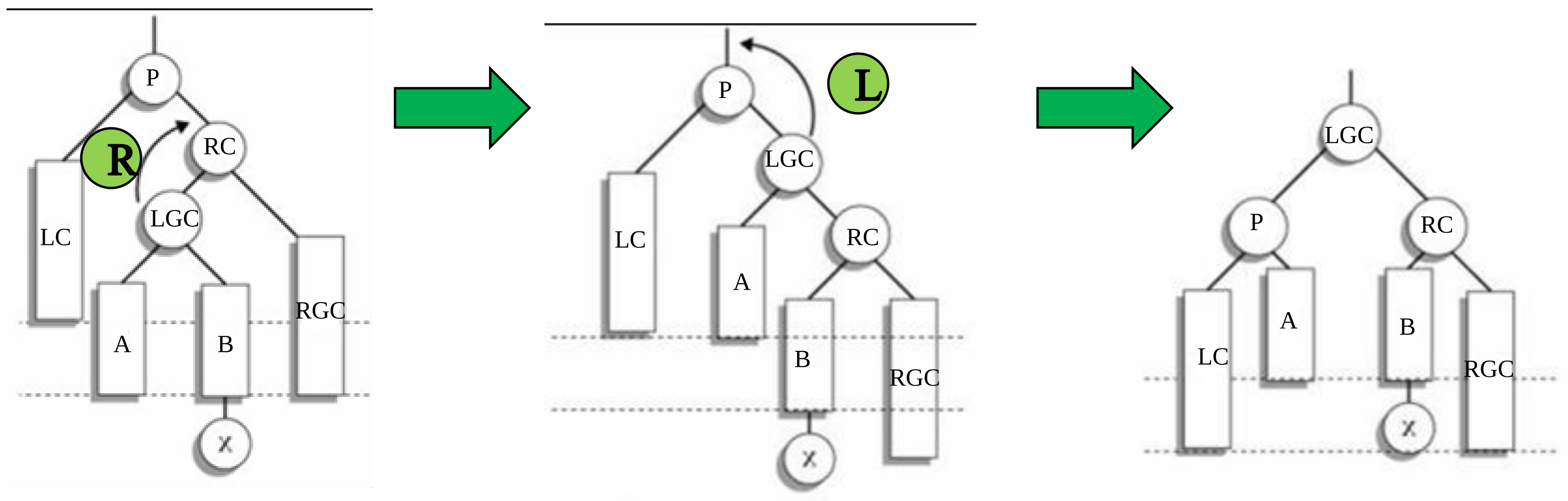
left inside grandchild (left-right)

Watch RGC rise to the top

doubleRotateLeftRight()

```
private static <T> AVLNode<T> doubleRotateLeftRight(
                                                AVLNode<T> p)
/* double right rotation on p is
        left rotation, then right rotation */
{
  p.left = singleRotateLeft(p.left);
  return singleRotateRight(p);
}
```

2. A Double Right-Left Rotation



Single right rotation about RC

Single left rotation about P

balanced

right inside grandchild (right-left)

Watch LGC rise to the top

doubleRotateRightLeft()

```
private static <T> AVLNode<T> doubleRotateRightLeft(
                                            AVLNode<T> p)
/* double left rotation on p is
        right rotation, then left rotation */
{
  p.right = singleRotateRight(p.right);
  return singleRotateLeft(p);
}
```

addNode()

- addNode() recurses down to the insertion point and inserts the node. (as in BST)
- As it returns, it visits the nodes in reverse order, fixing any imbalances using rotations.

- It must handle four cases:
  – balance height == 2: left-left, left-right
  – balance height == -2: right-left, right-right

Basic addNode()

```
private Node<T> addNode(Node<T> t, T item)
{
   if (t == null)   // found insertion point
     t = new Node<T>(item);

   else if (item.nodeValue < t.nodeValue)
     t.left = addNode( t.left, item);      // visit left subtree

   else if (item.nodeValue >= t.nodeValue )
      t.right = addNode(t.right, item);    // visit right

}  // end of addNode()
```

No AVL rotation code added yet

AVL rotation code added

```
private AVLNode<T> addNode(AVLNode<T> t, T item)
{
   if(t == null)   // found insertion point
     t = new AVLNode<T>(item);

   else if (item.nodeValue < t.nodeValue)
   {
     // visit left subtree: add node then maybe rotate
     t.left = addNode( t.left, item);  // add node, then...

     if (height(t.left) - height(t.right) == 2 ) //too heavy on left
     {
       if (item.nodeValue < t.left.nodeValue)
                                   // problem on left-left
         t = singleRotateRight(t);
       else                       // problem on left-right
         t = doubleRotateLeftRight(t);    // left then right rotation
     }
   }
```

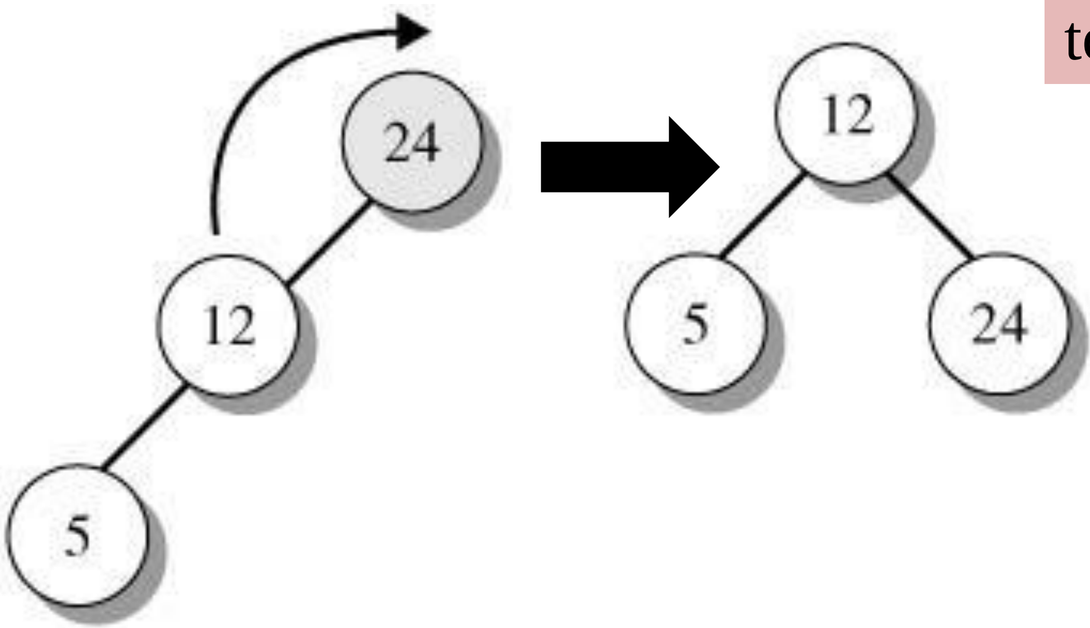
continued

```
   else if (item.nodeValue >= t.nodeValue ) {
      // visit right subtree: add node then maybe rotate
      t.right = addNode(t.right, item );  // add node, then...

      if (height(t.left)-height(t.right) == -2){ //too heavy on right
        if (item.nodeValue <= t.right.nodeValue)
                                    // problem on right-right
          t = singleRotateLeft(t);
        else     // problem on right-left
          t = doubleRotateRightLeft(t); // right then left rotation
      }
   }

   // calculate new height of t
   t.height = max(height(t.left), height(t.right)) + 1;


}  // end of addNode()
```

Building an AVL Tree

gray node is too heavy

left outside grandchild (left-left)

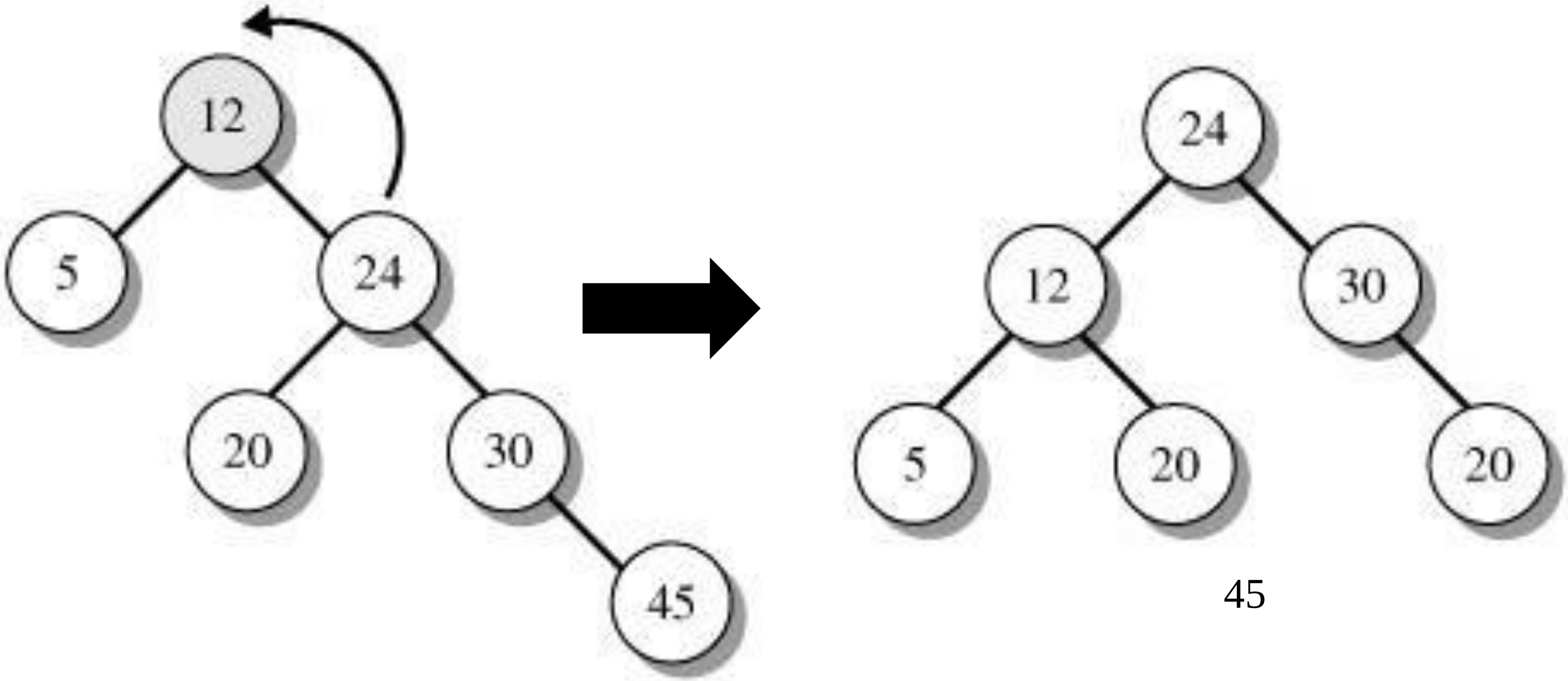


Insert 24 12 5　　Single Rotate Right (P = 24)

Insert the first three elements 24, 12, and 5.
At 5, node 24 has balance factor 2.

continued

right outside grandchild (right-right)

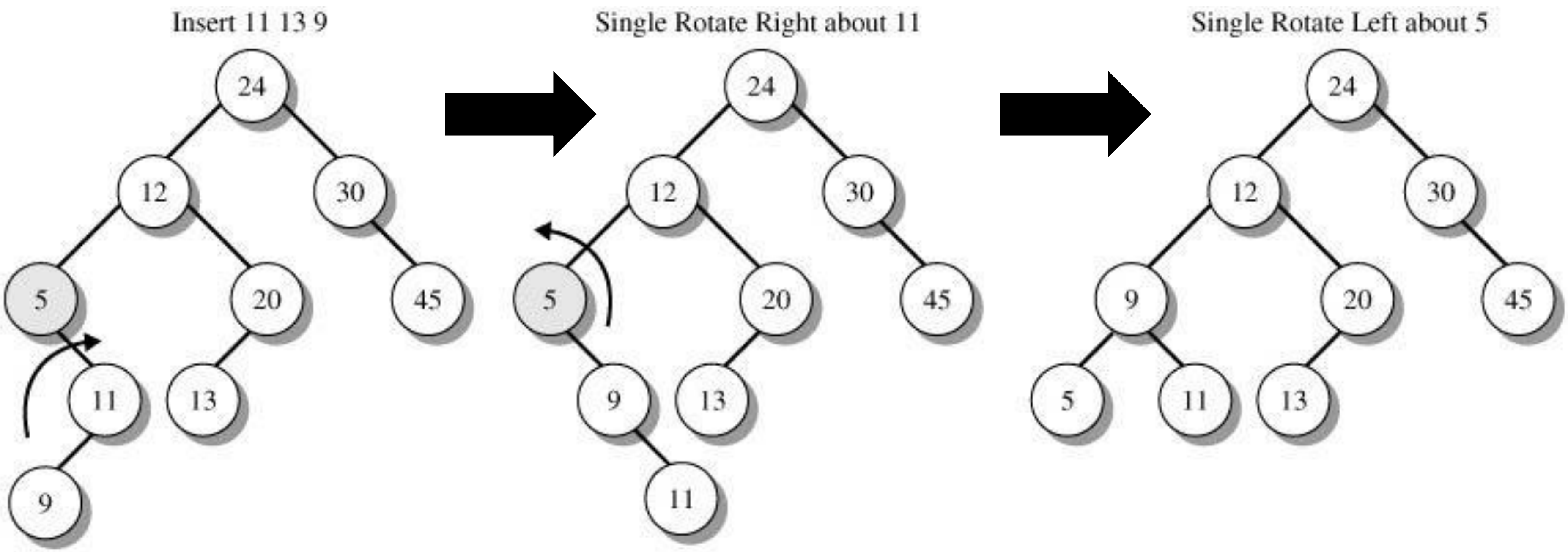


Insert 30 20 45

Single Rotate Left (P = 12)
attach 20 as right child of 12

Insert the next three elements 30, 20, and 45.
At 45, node 12 has balance factor -2.

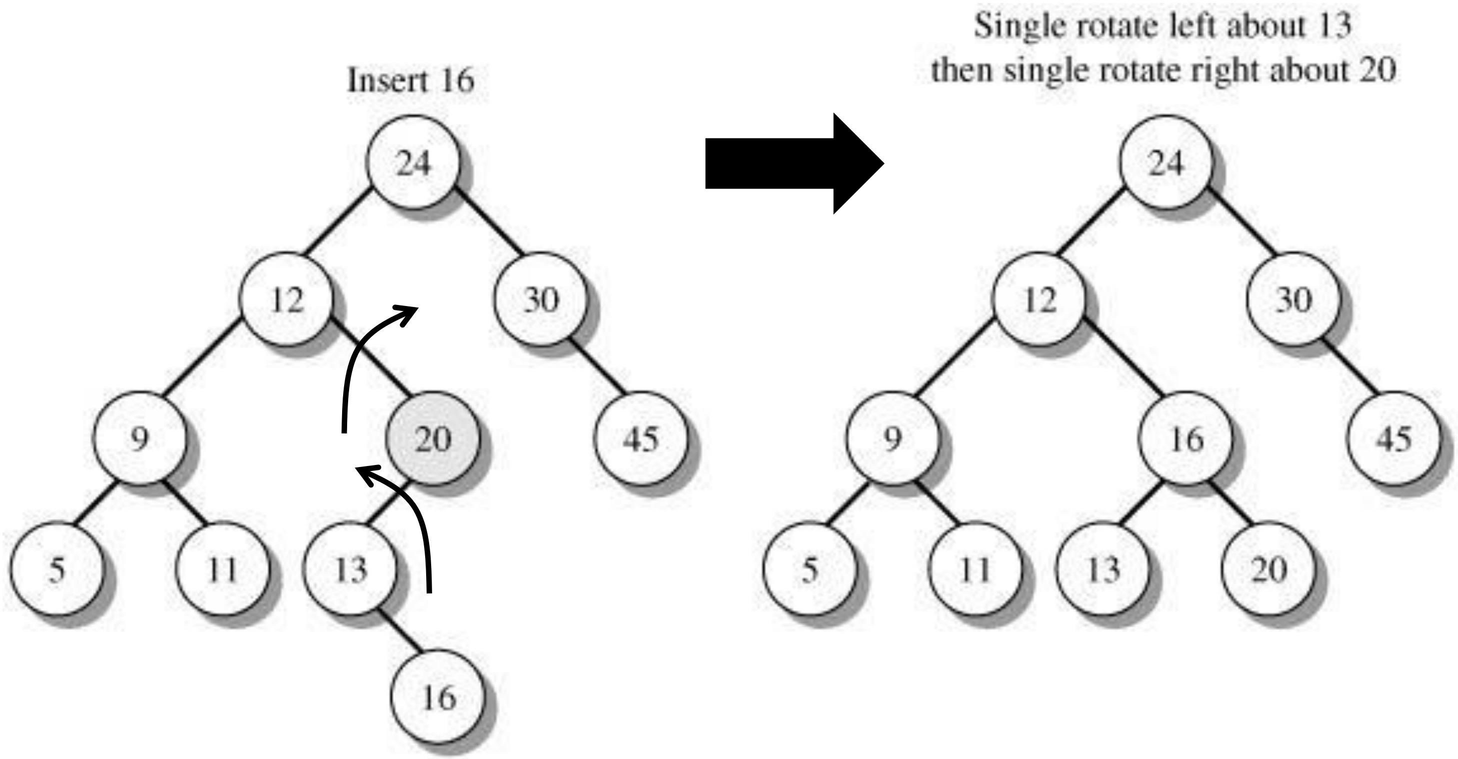
continued



Insert the three elements 11, 13, and 9. At 9, node 5 has balance factor −2.

right inside grandchild (right-left) → double rotate left (right then left rotation)

continued



Insert the last element 16. Node 20 has balance factor +2.

left inside grandchild (left-right) → double rotate right (left then right rotation)

continued

Efficiency of AVL Tree Insertion

- Detailed analysis shows:

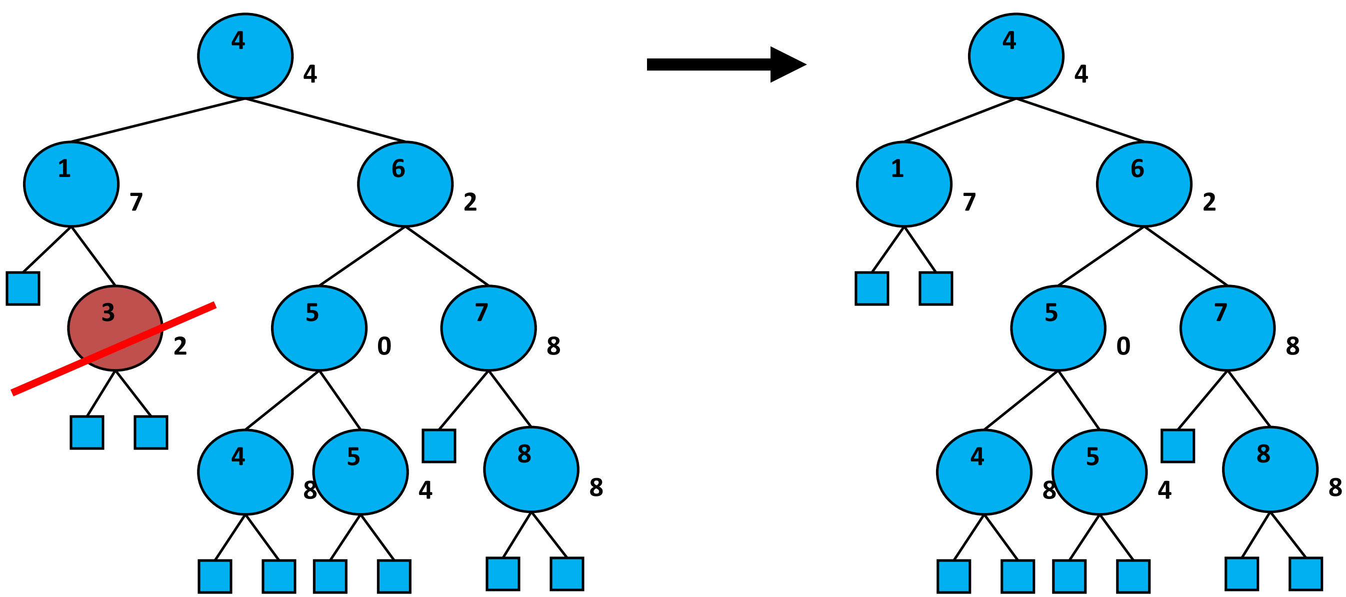
  $\text{int}(\log_2 n) \leq \text{height} < 1.4405 \log_2(n+2) - 1.3277$

- So the worst case running time for insertion is $O(\log_2 n)$.

- The worst case for deletion is also $O(\log_2 n)$.

Deletion in an AVL Tree

- Deletion can easily cause an imbalance
  - e.g delete 3



before deletion of 32

after deletion