CMP(N)302: Design and Analysis of Algorithms

Lecture 03: Binary Search Trees

Ahmed Hamdy

Computer Engineering Department

Cairo University

Fall 2019

Binary search trees (BST)

- Binary Tree (BT) vs Binary search trees (BST)

- Balanced vs unbalanced

- A node in a BT/BST has:
  - Parent pointer
  - Left pointer
  - Right pointer
  - Data

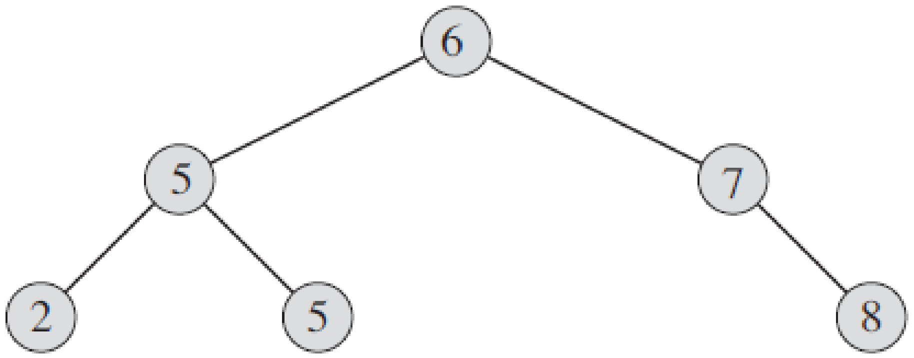
Binary search trees (BST)

- For all nodes y in left subtree of x,

$$y.key \leq x.key$$

- For all nodes y in right subtree of x,

$$y.key \geq x.key$$



Why BST?

- Hash tables has $O(1)$ for insertion, deletion and search. Then why BST?
- BST advantages over hash tables (HTs):
  - Can get sorted data through in-order traversal
  - Easy for operations such as min, max, predecessor, and successor
  - Easy for range query and order statistic (how?)
  - Easier to implement compared to HTs
  - Self-balancing BSTs have complexity guarantees of $O(\log n)$ in contrast to cases of table-resizing for HTs

BST operations

- Operations:
  - Search
  - Minimum
  - Maximum
  - Predecessor
  - Successor
  - Insert
  - Delete
- Complexity: $\mathrm{O}(h)$
  - Complete/balanced tree: $\mathrm{O}(\log n)$
  - Linear chain: $\mathrm{O}(n)$

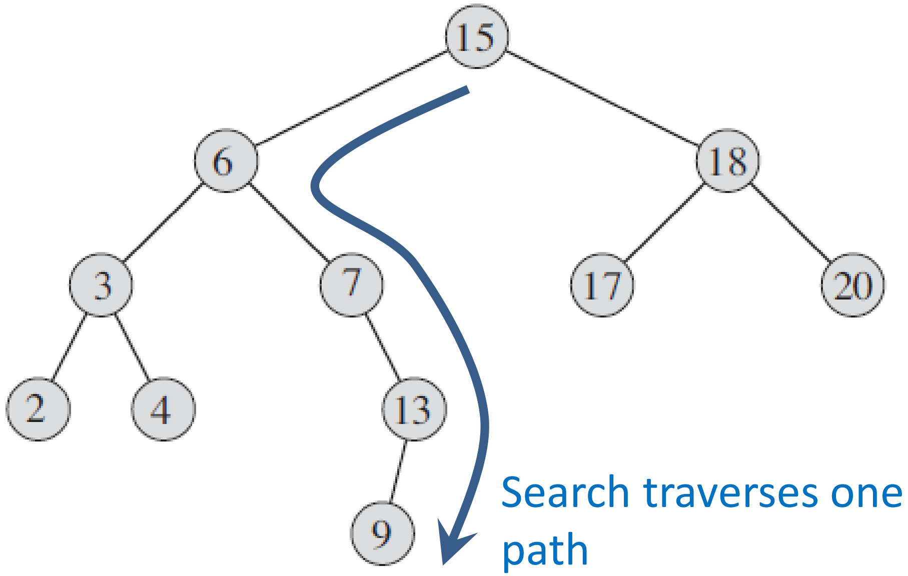
BST traversal

- Inorder tree walk:

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

- Preorder tree walk: visit root first
- Postorder tree walk: visit root last
- Complexity: $\Theta(n)$

BST search




```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

Simplify

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

BST operations

Minimum

TREE-MINIMUM(x)

while $x.left \neq$ NIL
  $x = x.left$
return x

Maximum

TREE-MAXIMUM(x)

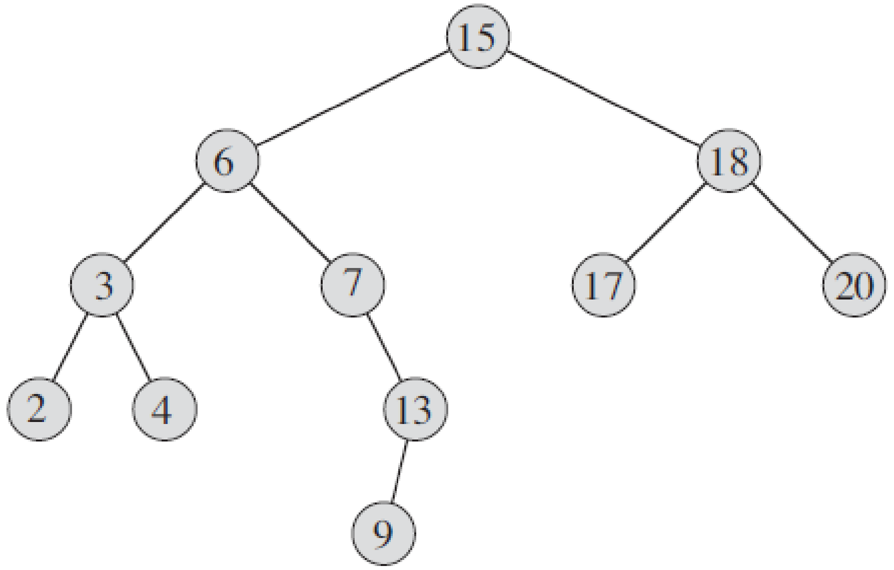
while $x.right \neq$ NIL
  $x = x.right$
return x

BST operations

Successor:

- Successor(node15)
- Successor(node13)
- Successor(node20)

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```



BST operations

Insert

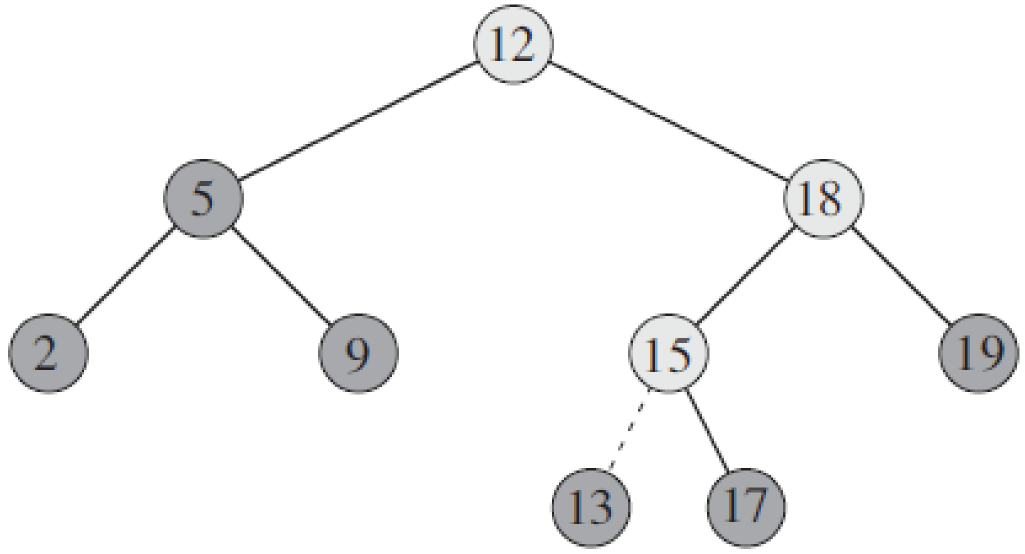


Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

BST operations

Insert

TREE-INSERT(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

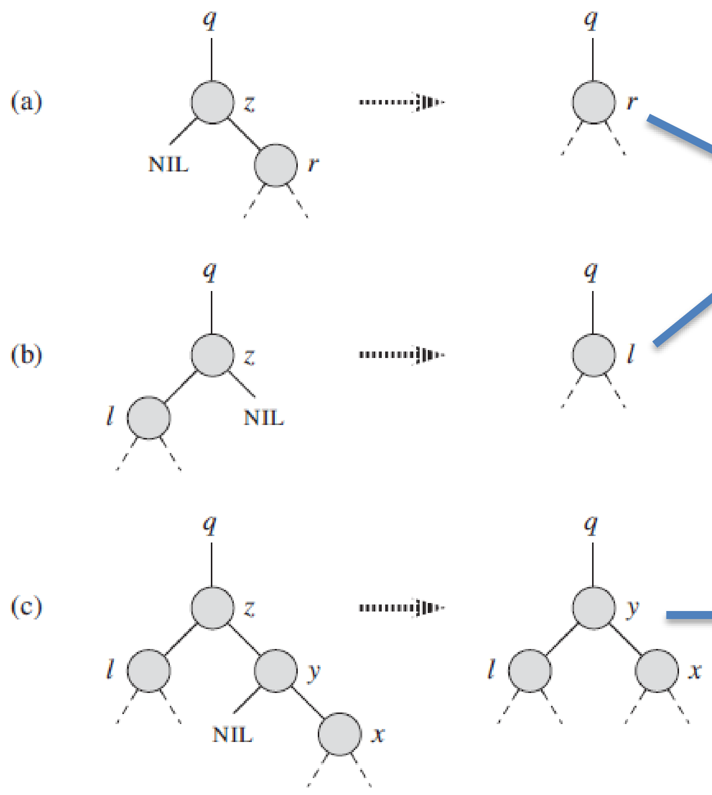
Search

BST operations

Delete

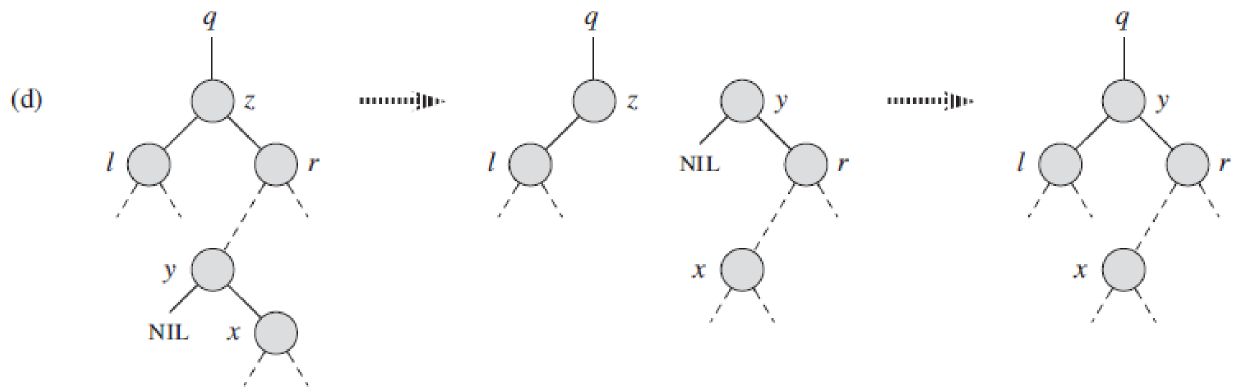
TRANSPLANT(T, u, v)

```
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```



TREE-DELETE(T, z)

```
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

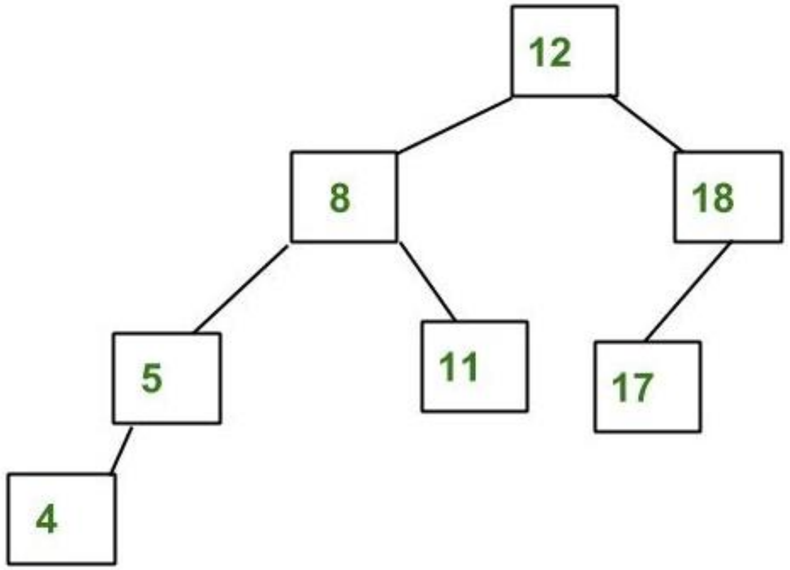


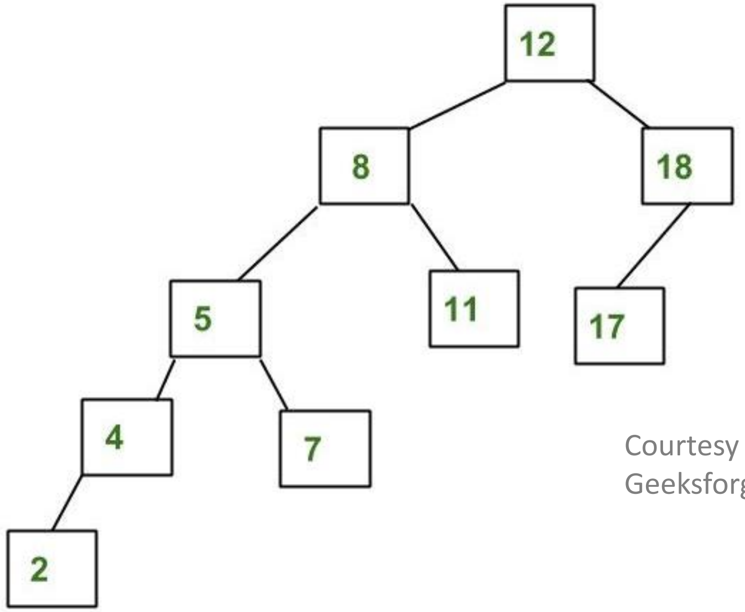
BST building

- Linked-list trees cost $O(n)$ for operations
  - Caused by insertion of sorted elements
- To minimize worst case: randomized insertion
  - Average case height is $O(\log n)$
  - Randomization is not always possible if data is not entirely present
- To have $O(\log n)$ height in the worst case
  - Use self-balancing trees, i.e. AVL and red-black trees

AVL trees

- Heights of the two child subtrees of any node differ by at most one.
- If after insertion or deletion, the difference is more than one, then rebalancing takes place.





Courtesy of
Geeksforgeeks.com

What is the height of each node??

What is the difference in height of each node??

AVL: Insertion

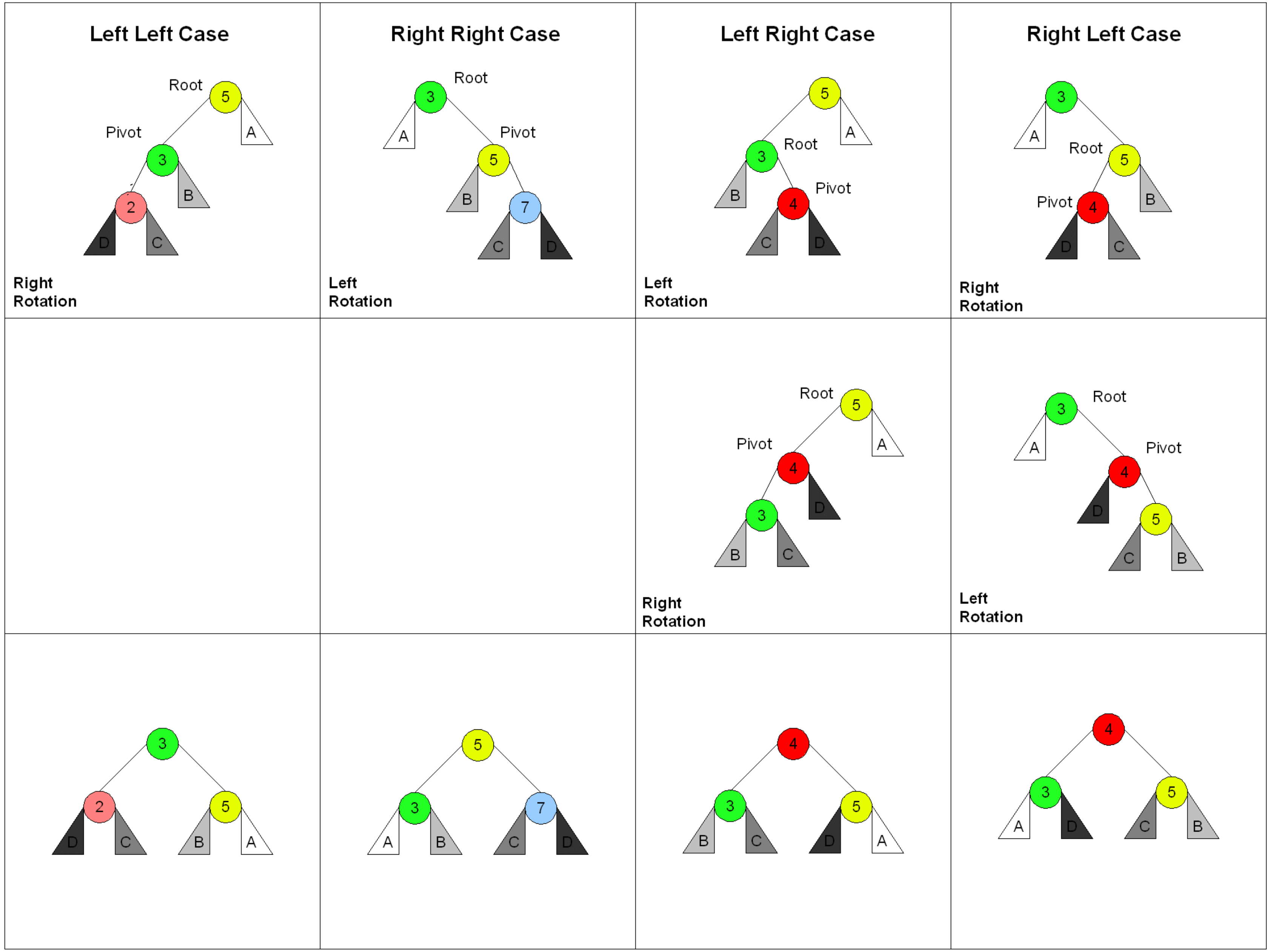
- Insert in the same away as in standard BST
- Now, there might be one of 4 cases to resolve:
  - 2 resolved by single rotation
    - Inserting the new node in left subtree to a parent who is a left child to his parent (LL)
    - Inserting the new node in right subtree to a parent who is a right child to his parent (RR)
  - 2 resolved by double rotations
    - Inserting the new node in right subtree to a parent who is a left child to his parent (RL)
    - Inserting the new node in left subtree to a parent who is a right child to his parent (LR)

AVL: Insertion cases

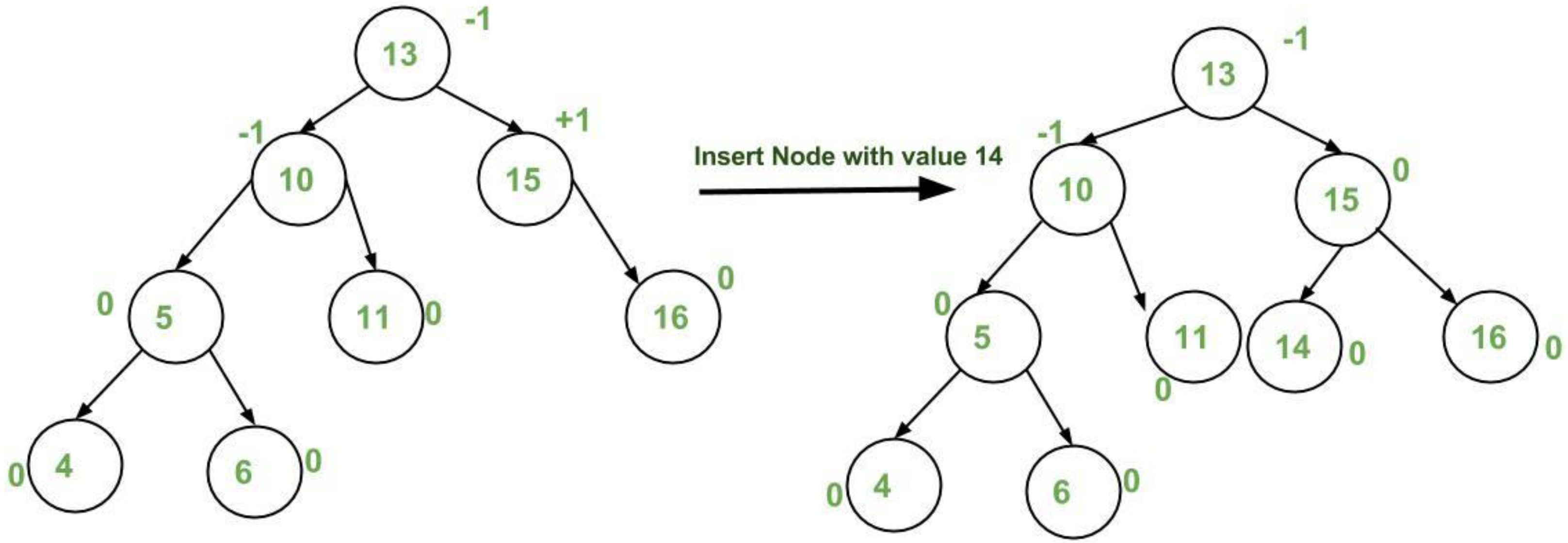
There are 4 cases in all, choosing which one is made by seeing the direction of the first 2 nodes from the unbalanced node to the newly inserted node and matching them to the top most row.

Root is the initial parent before a rotation and **Pivot** is the child to take the root's place.

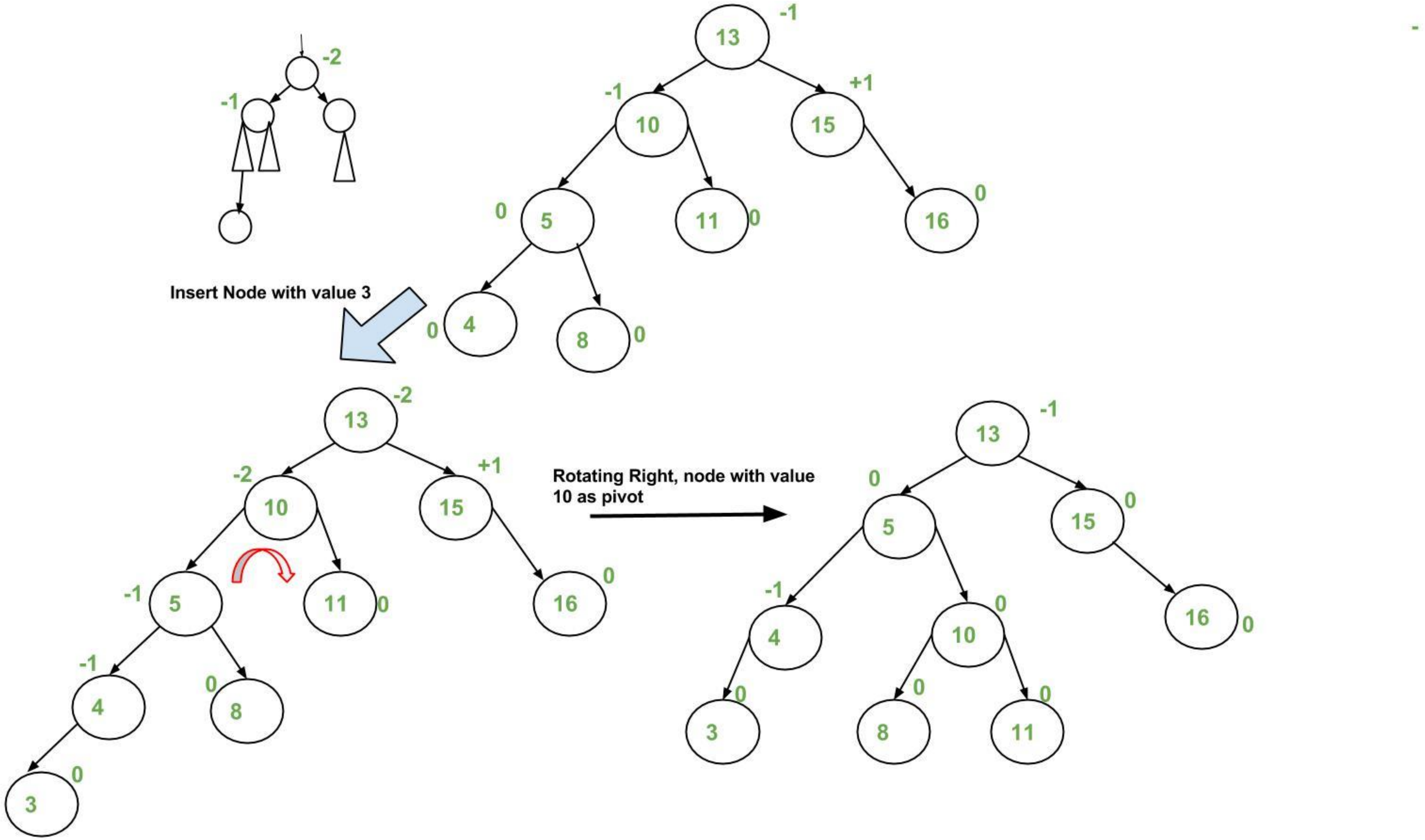
Courtesy of Wikipedia



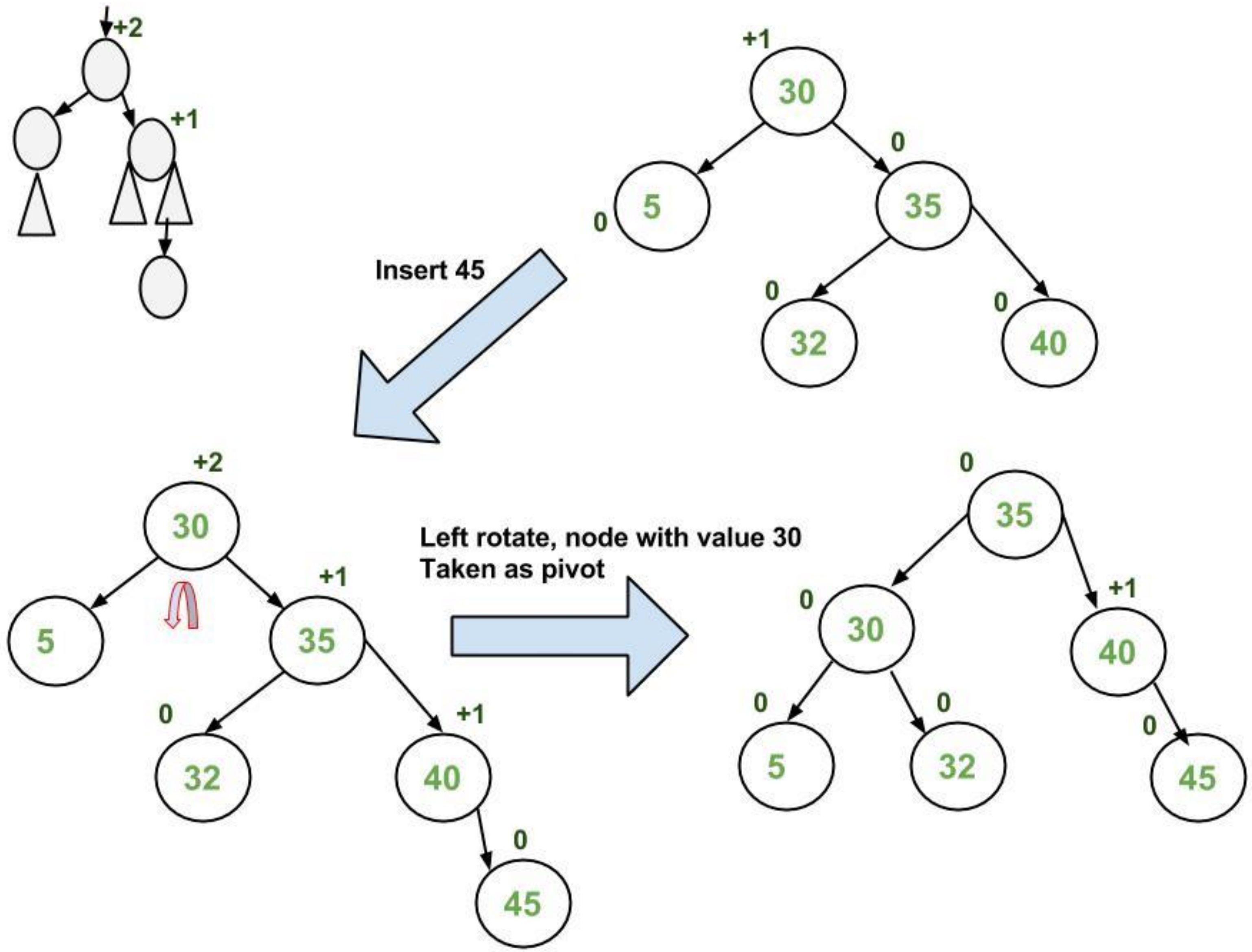
AVL: Insertion examples



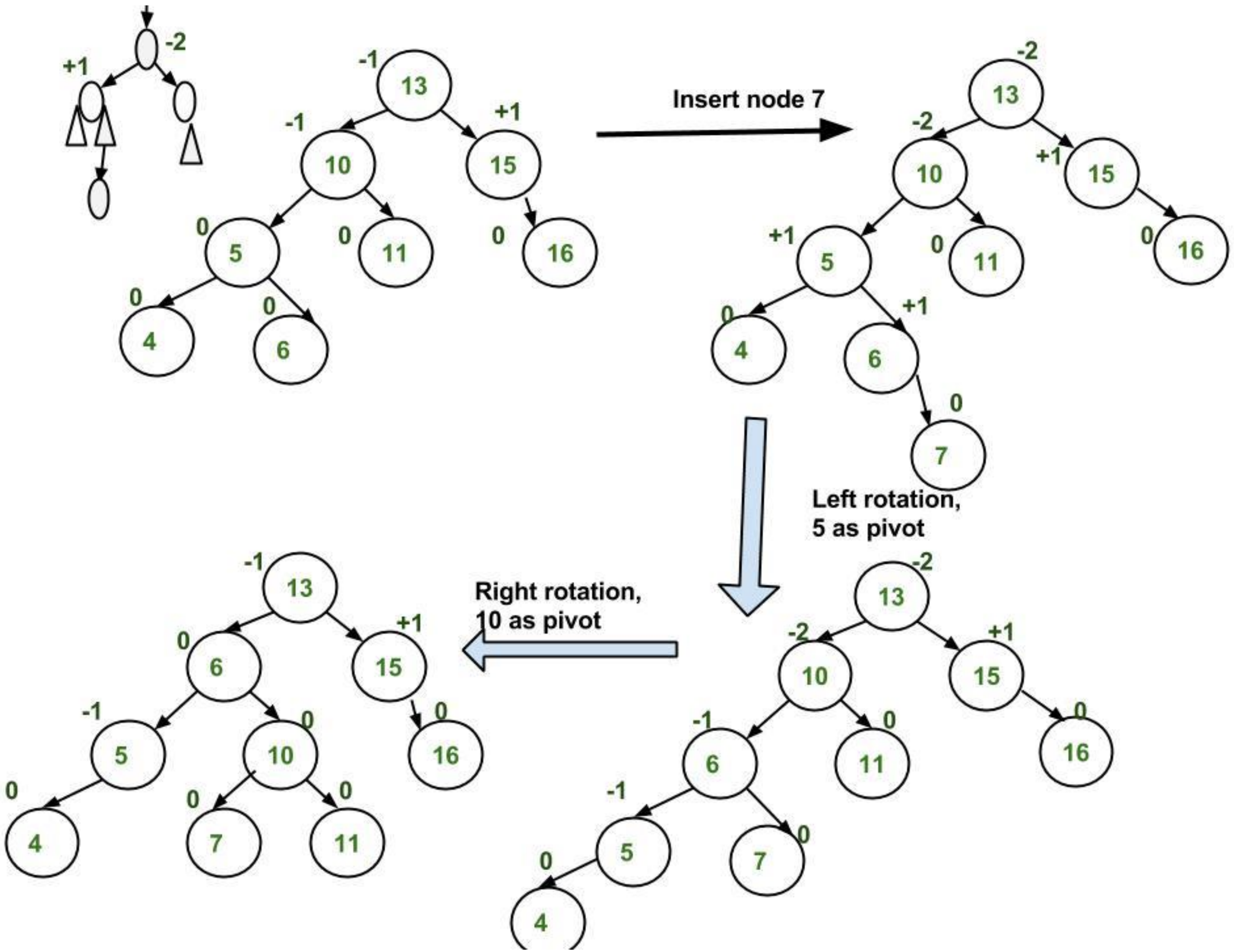
AVL: Insertion examples



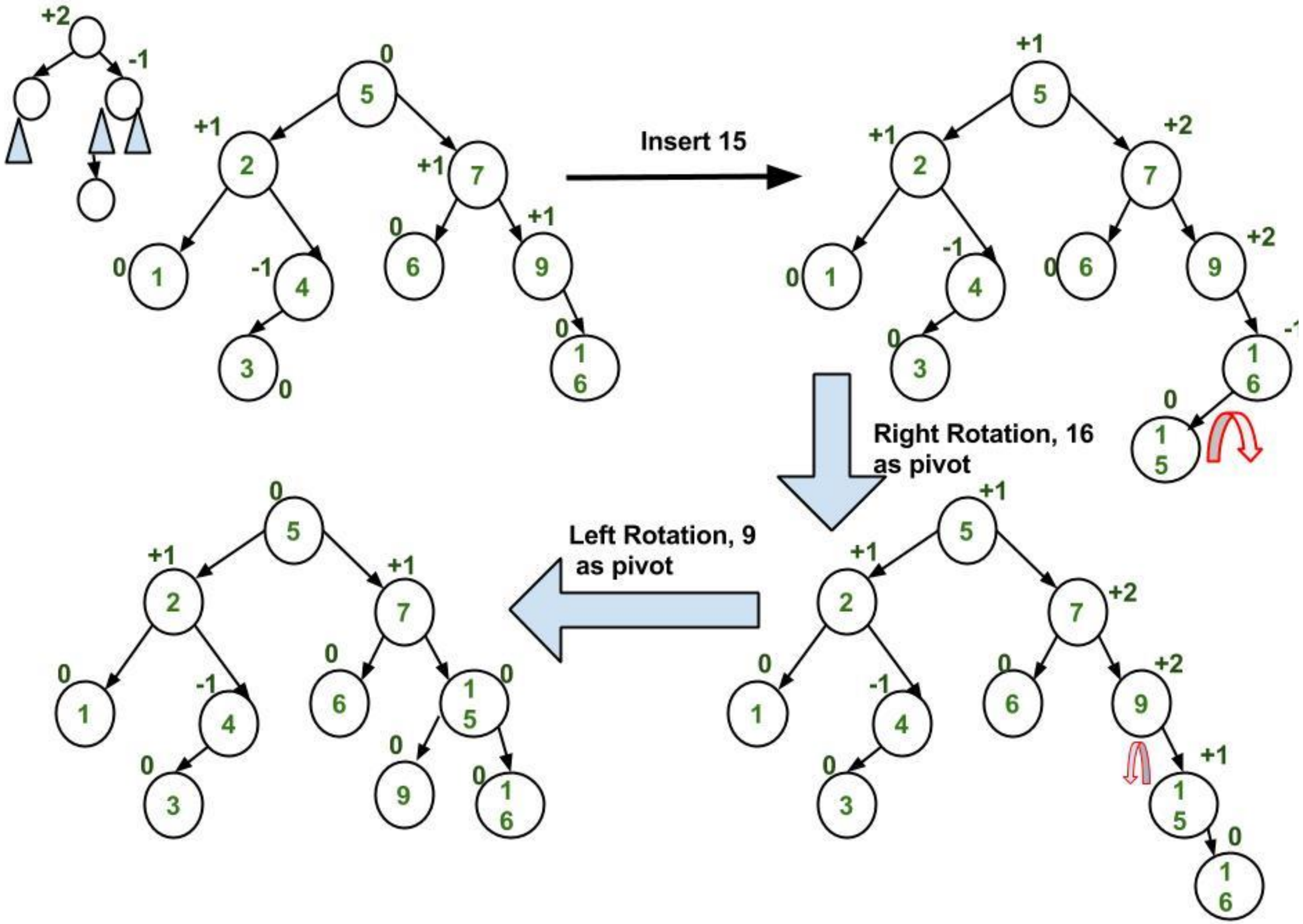
AVL: Insertion examples



AVL: Insertion examples



AVL: Insertion examples



AVL: Insertion examples

Courtesy of Wikipedia

AVL: Deletion

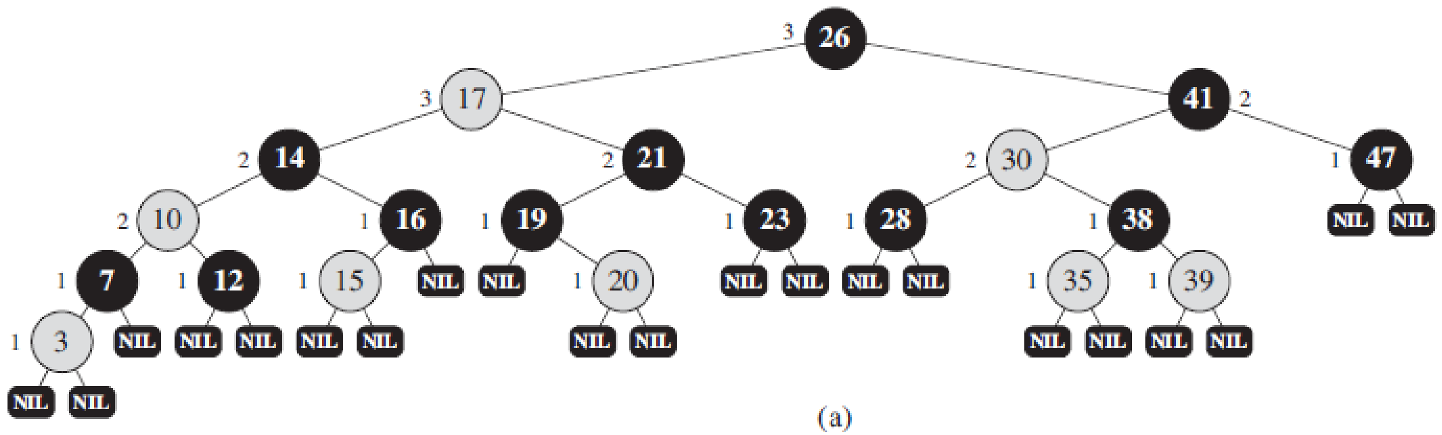
- Delete in the same way as in standard BST.
- After deletion, the same four cases may take place but now the 3 nodes to rotate will not remain the same as in the case of insertion.
- When a node is deleted, the subtree containing this node is reduced in height, so the sibling subtree will relatively exceed in height.
- The three nodes will be consist of the first unbalanced node in the way up, its child with the largest height, and the next child in the way down with the largest height.

AVL sorting

Simply:

1. Insert n nodes in AVL tree in $O(n \log n)$.
2. Perform in-order traversal in $O(n)$.

Red-Black trees



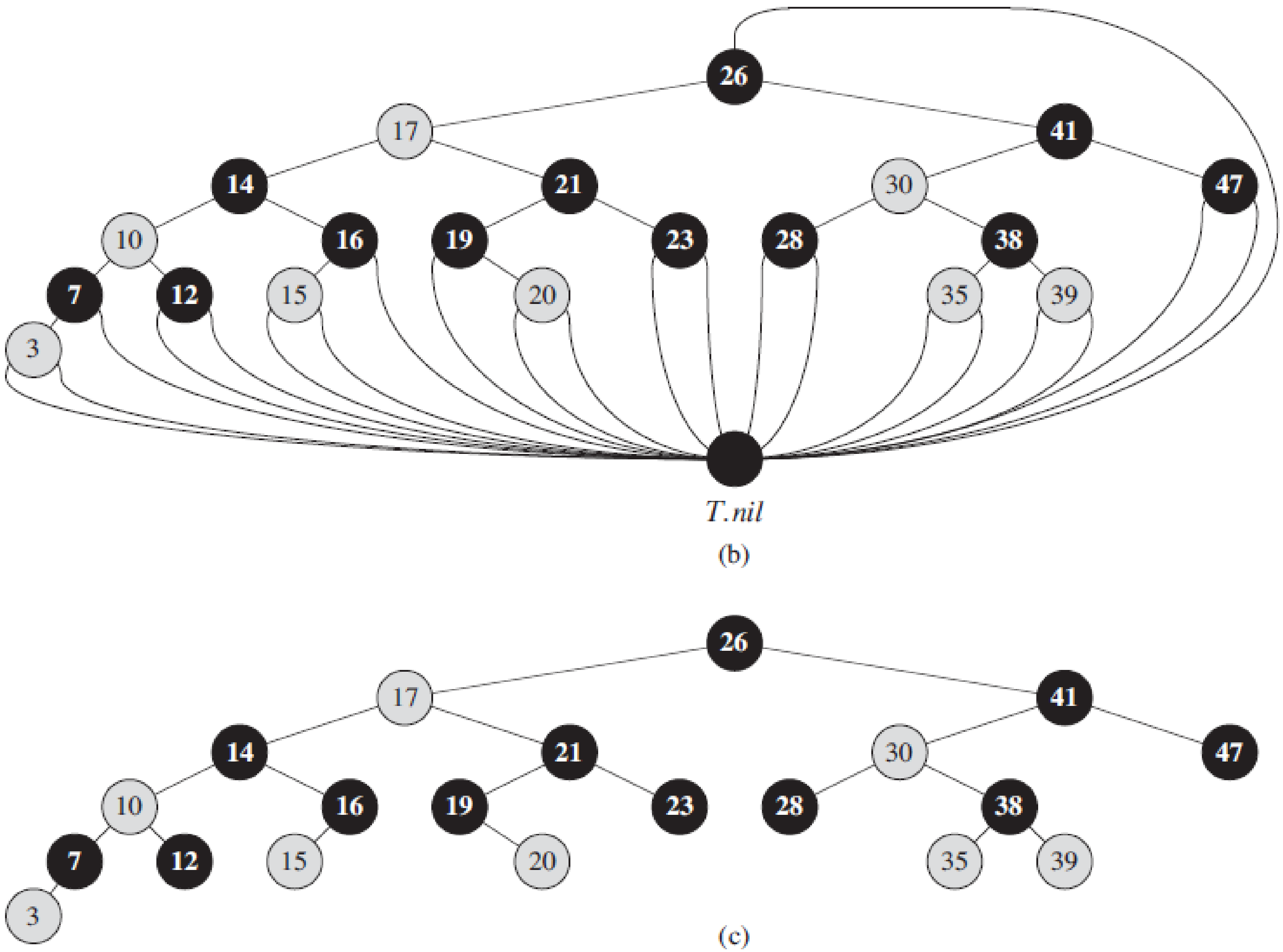
(a)

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Lemma 13.1

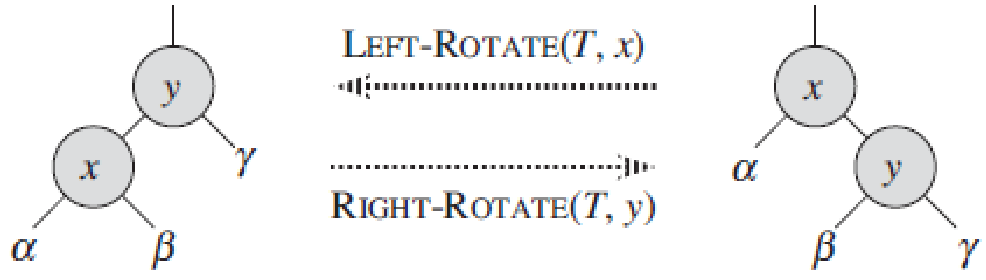
A red-black tree with n internal nodes has height at most $2\lg(n+1)$.

Red-Black trees



Red-Black trees

- Rotations

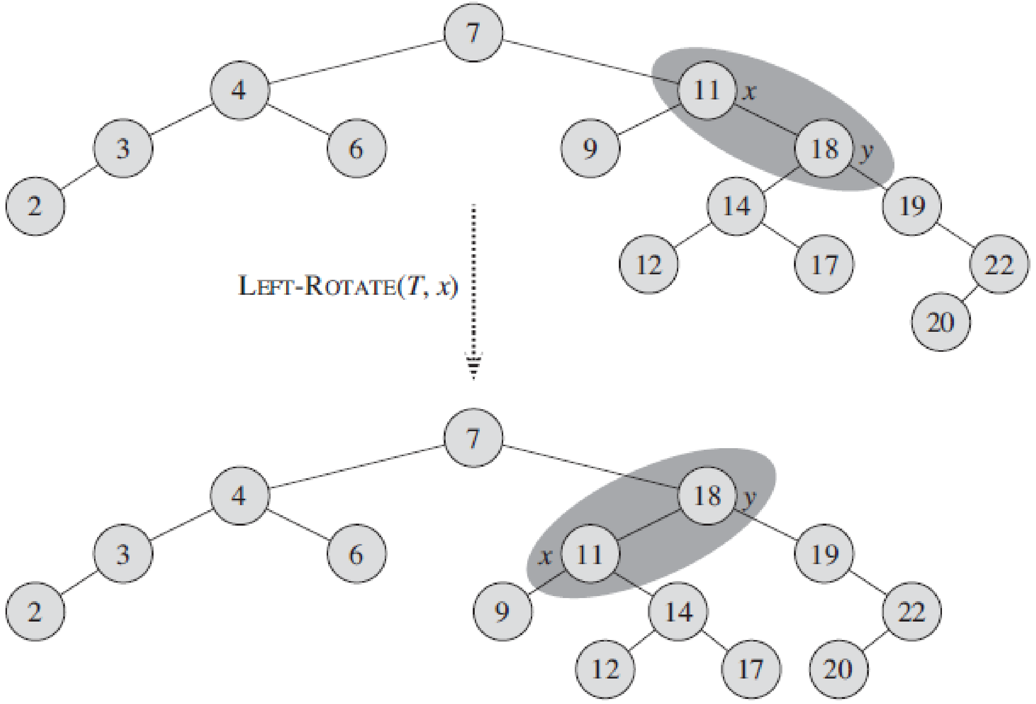


LEFT-ROTATE(T, x)

```
y = x.right                 // set y
x.right = y.left            // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                   // link x's parent to y
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x                  // put x on y's left
x.p = y
```

Red-Black trees

- Rotations



Red-Black trees

- Insertion

Standard BST insertion

```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

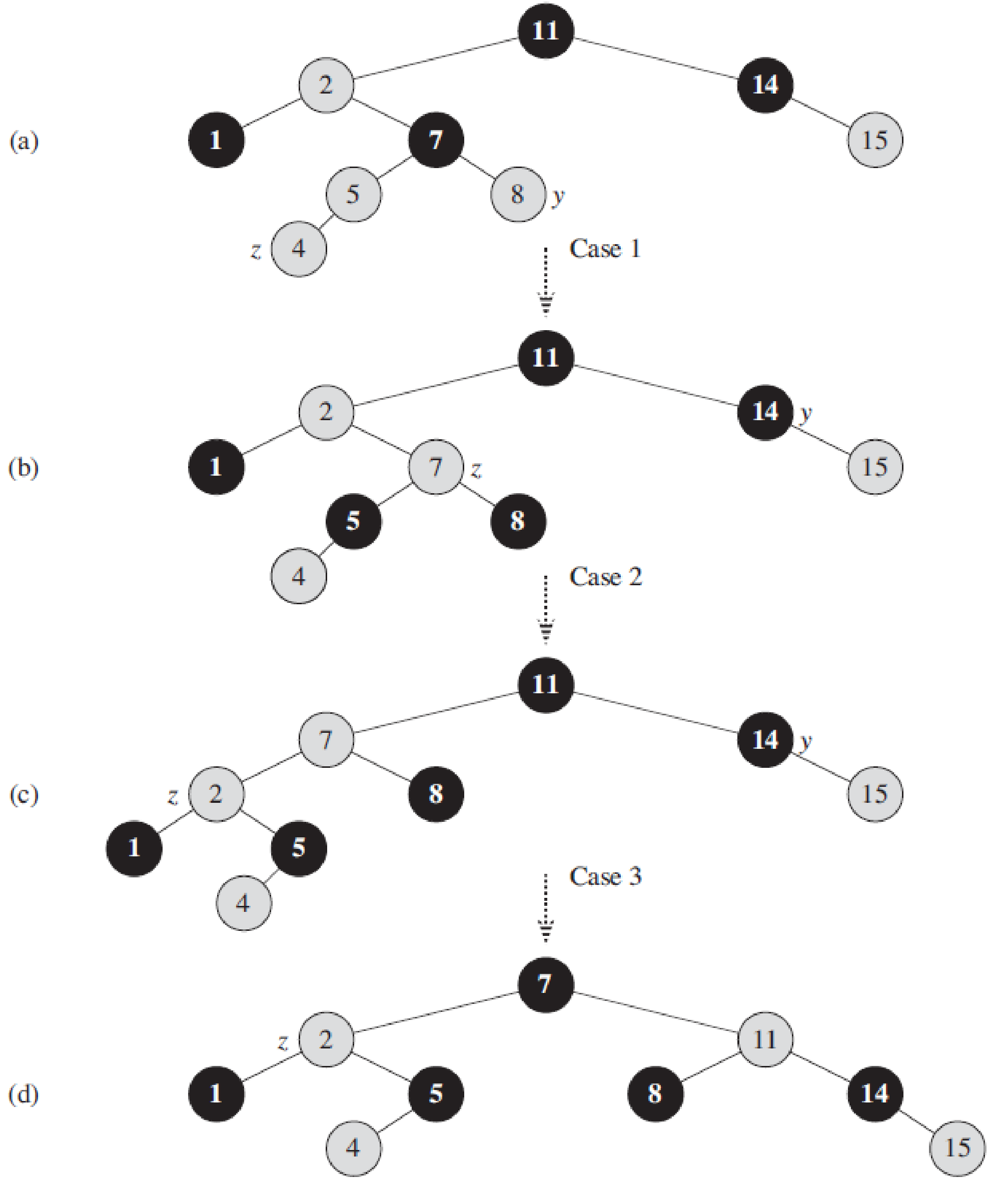
1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Red-Black trees

- Insertion

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK            // case 1
            y.color = BLACK              // case 1
            z.p.p.color = RED            // case 1
            z = z.p.p                    // case 1
        else if z == z.p.right
                z = z.p                  // case 2
                LEFT-ROTATE(T, z)        // case 2
            z.p.color = BLACK            // case 3
            z.p.p.color = RED            // case 3
            RIGHT-ROTATE(T, z.p.p)       // case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```



AVL vs Red-Black Trees

- AVL is faster in search because it is more balanced (thus better for DB)
- RBT is faster in insertion and deletion because it relaxes the condition for balancing compared to AVL.
- AVL requires more space to store the height of every node compared to RBT which stores 1-bit per node.
- RBT is implemented in C++ as map, multimap and multiset.