Design and Analysis of Algorithms

CAIRO UNIVERSITY

Lecture 03: Binary Search Trees

Ahmed Hamdy

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

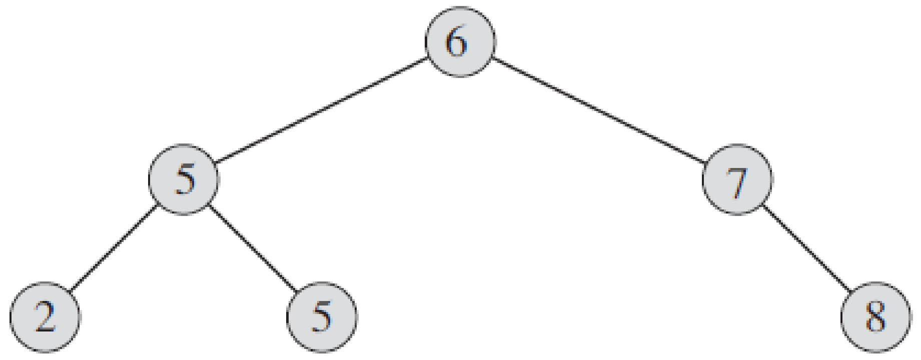
Binary Search Trees (BST)

- 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: $\mathrm{O}(n)$

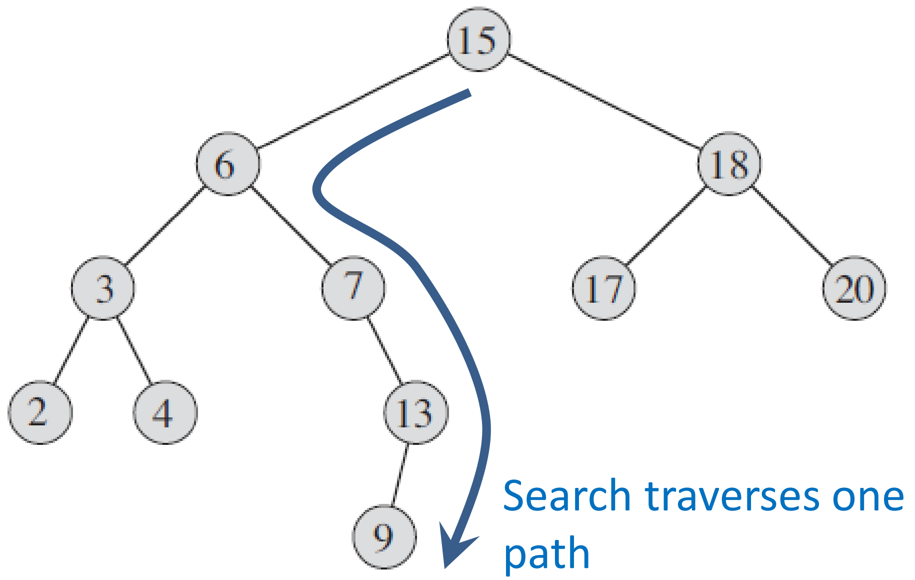
BST traversal

- Inorder tree walk:

```
INORDER-TREE-WALK(x)
1  if $x \neq$ NIL
2      INORDER-TREE-WALK($x.left$)
3      print $x.key$
4      INORDER-TREE-WALK($x.right$)
```

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

BST search



Any tree problem that ends up traversing a single path in the tree can be done iteratively

```
TREE-SEARCH(x, k)
1  if $x$ == NIL or $k$ == $x.key$
2      return $x$
3  if $k < x.key$
4      return TREE-SEARCH($x.left$, $k$)
5  else return TREE-SEARCH($x.right$, $k$)
```

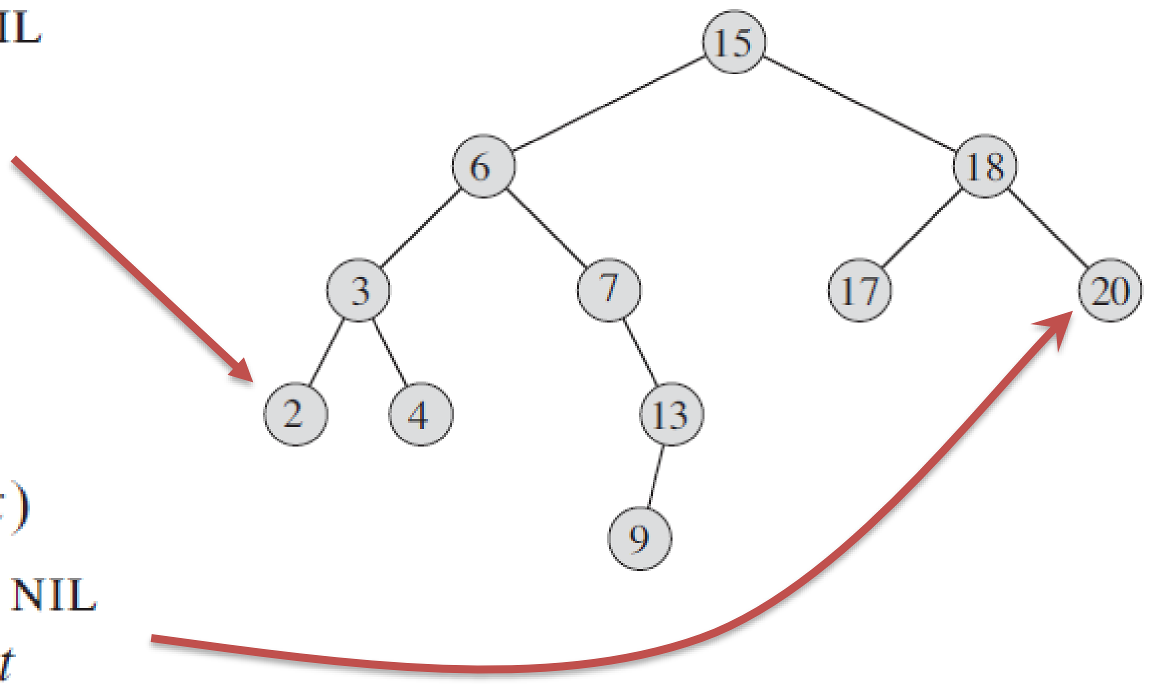
Simplify

```
ITERATIVE-TREE-SEARCH(x, k)
1  while $x \neq$ NIL and $k \neq x.key$
2      if $k < x.key$
3          $x = x.left$
4      else $x = x.right$
5  return $x$
```

BST operations

Minimum

```
TREE-MINIMUM(x)
1  while $x.left \neq$ NIL
2      $x = x.left$
3  return $x$
```



Maximum

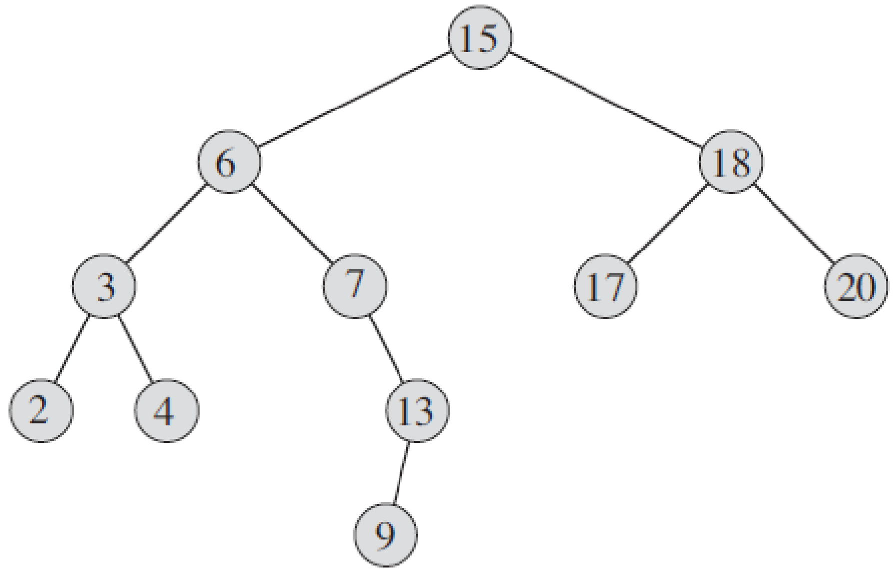
```
TREE-MAXIMUM(x)
1  while $x.right \neq$ NIL
2      $x = x.right$
3  return $x$
```

BST operations

Successor:

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

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```



BST operations

Insert

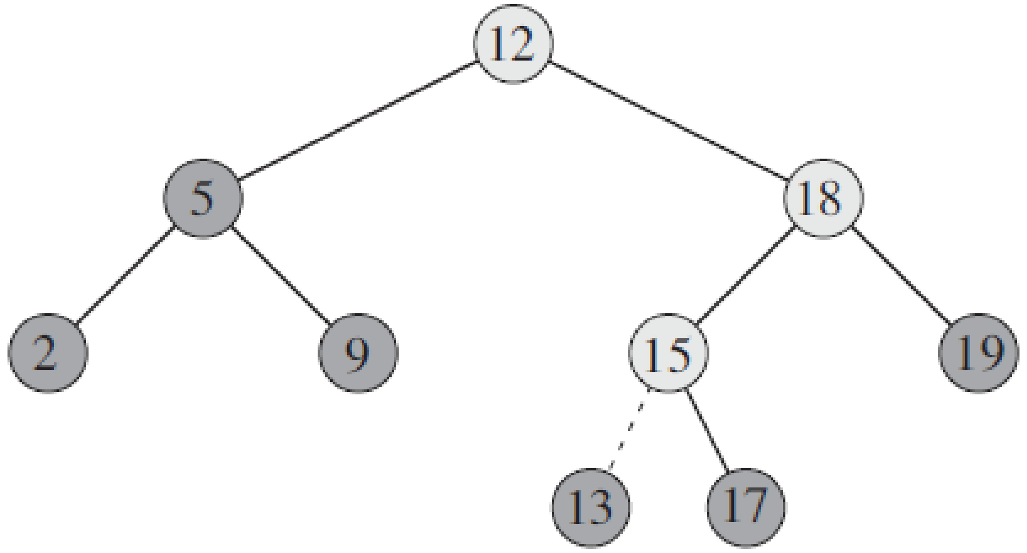


Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

BST operations

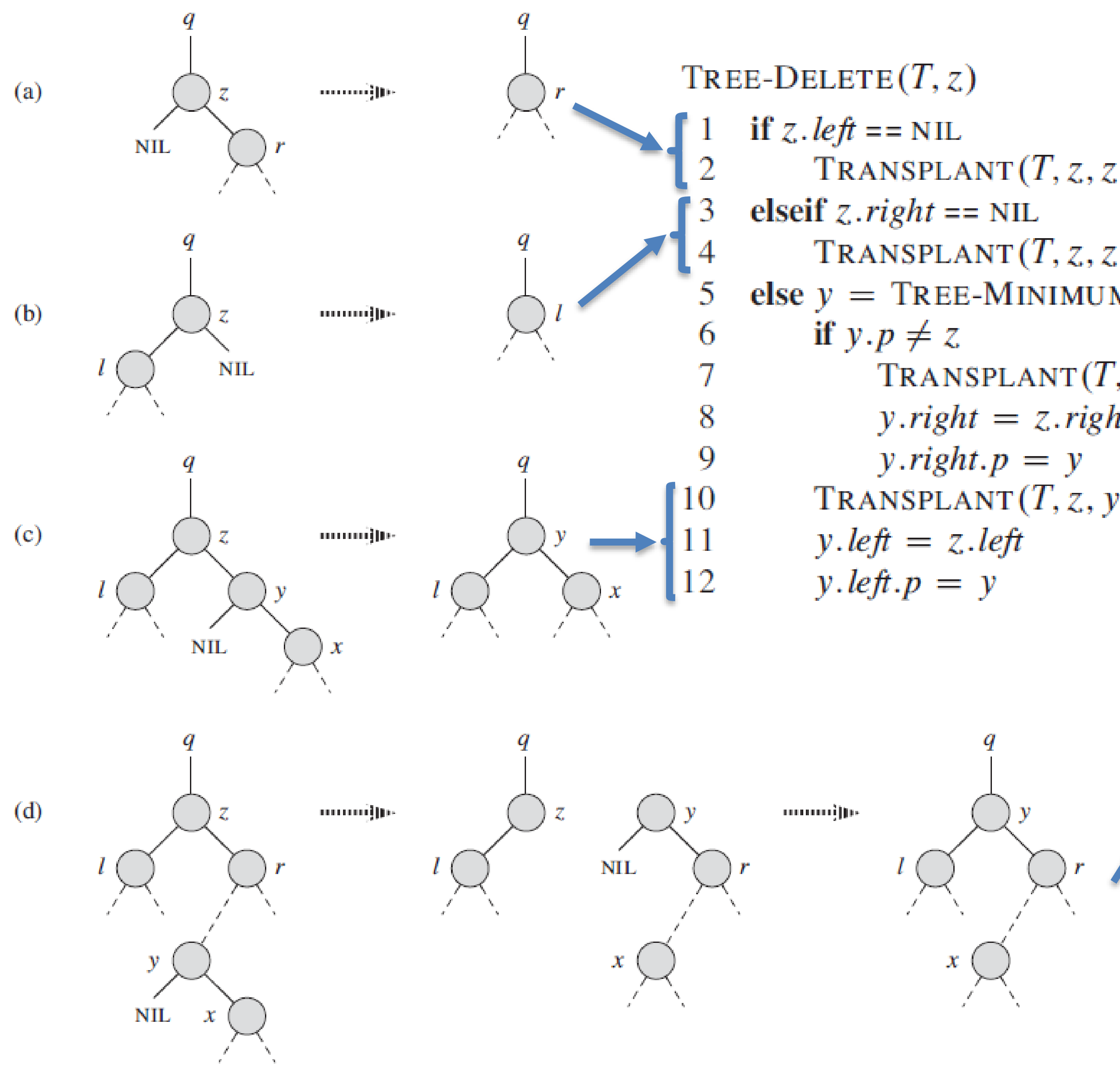
Insert

```
TREE-INSERT($T, z$)            // $z$ node to insert
 1  $y = \text{NIL}$                 // $y$ is parent of $x$
 2  $x = T.root$
 3  while $x \neq \text{NIL}$
 4      $y = x$
 5      if $z.key < x.key$             Search
 6          $x = x.left$
 7      else $x = x.right$
 8  $z.p = y$
 9  if $y == \text{NIL}$
10      $T.root = z$            // tree $T$ was empty
11  elseif $z.key < y.key$
12      $y.left = z$
13  else $y.right = z$
```

BST operations

Delete

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```



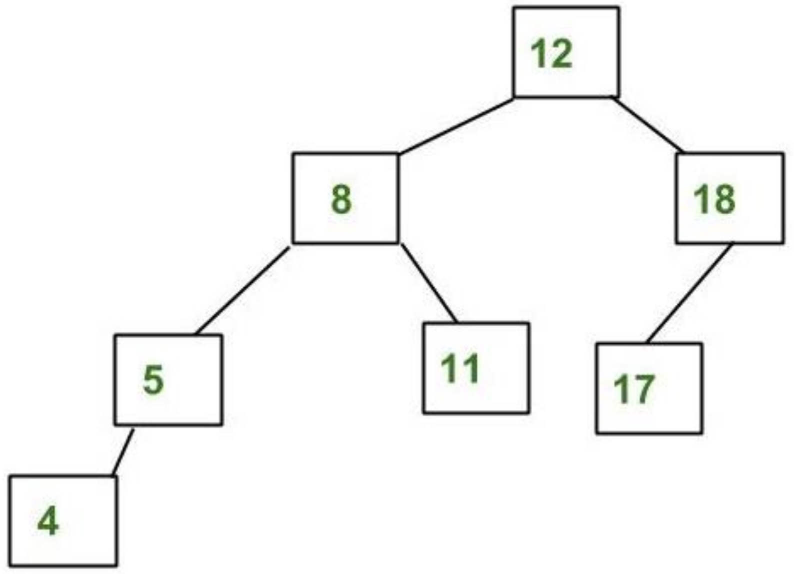
```
TREE-DELETE(T, z)
1   if z.left == NIL
2       TRANSPLANT(T, z, z.right)
3   elseif z.right == NIL
4       TRANSPLANT(T, z, z.left)
5   else y = TREE-MINIMUM(z.right)
6       if y.p ≠ z
7           TRANSPLANT(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

BST building

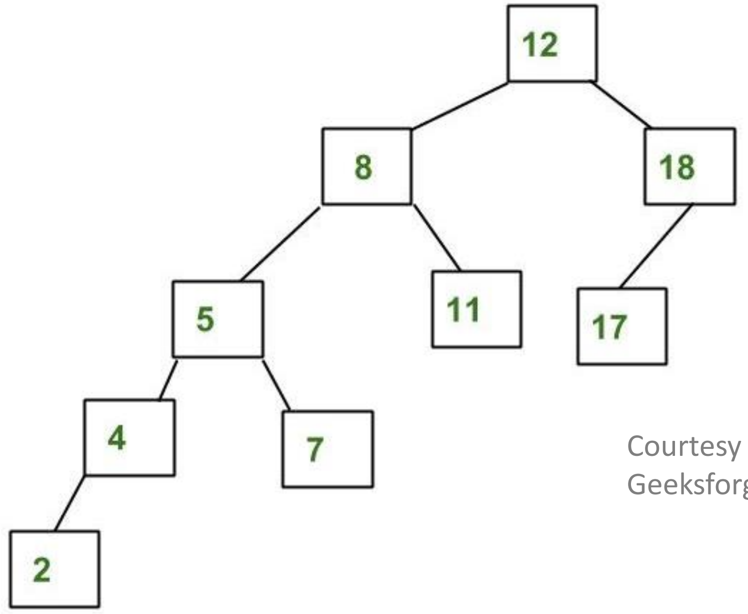
- Linked-list trees cost $O(n)$ for operations
  - Caused by insertion of sorted elements
- To minimize worst case: randomized insertion
  - Average case height is $O(\log n)$
  - Randomization is not always possible if data is not entirely present
- To have $O(\log n)$ height in the worst case
  - Use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



What is the height of each node??



Courtesy of Geeksforgeeks.com

What is the difference in height of each node??

AVL: Insertion

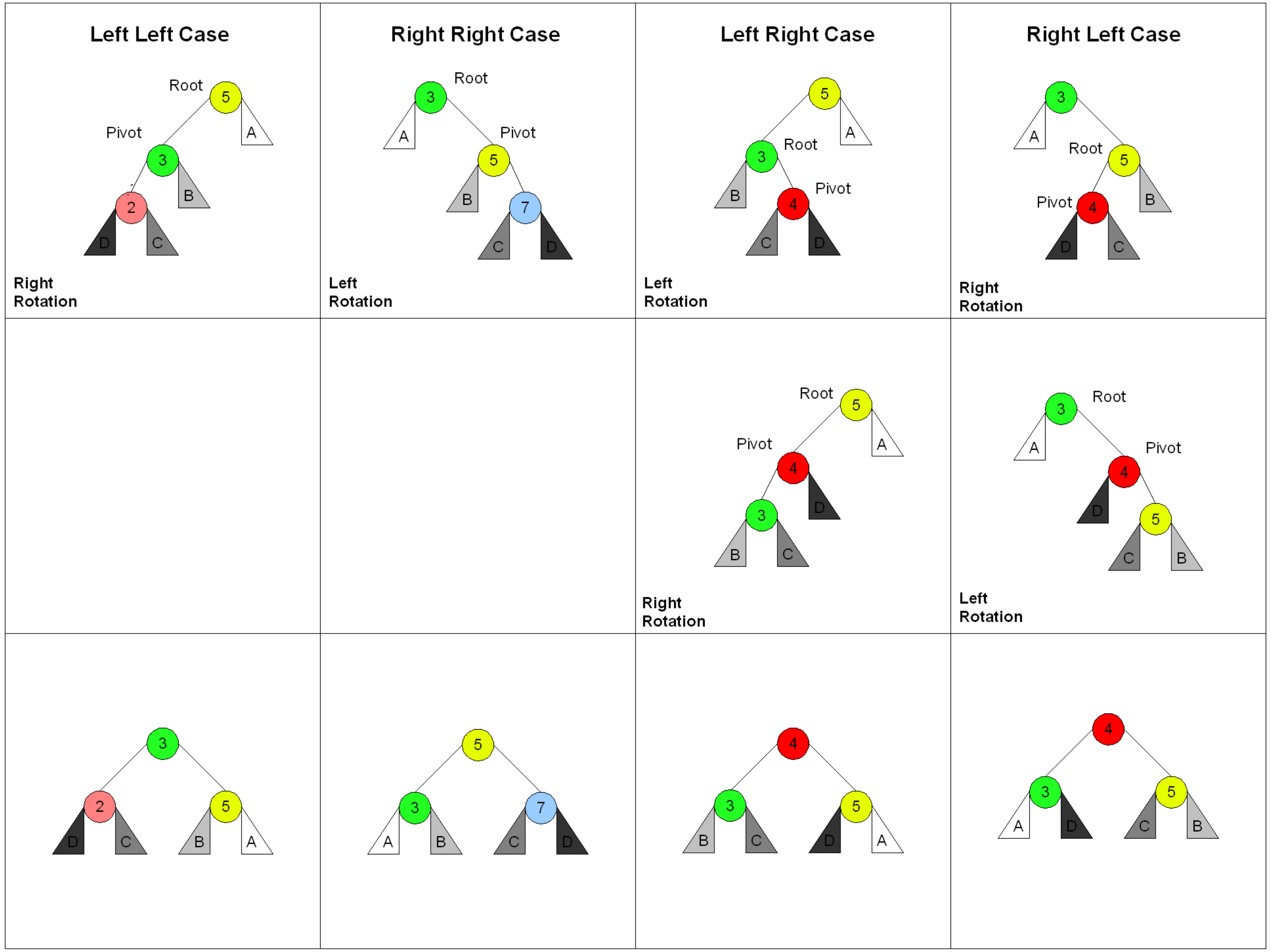
- Insert in the same away as in standard BST
- Now, there might be one of 4 cases to resolve:
  - 2 resolved by single rotation
    - Inserting the new node in left subtree to a parent who is a left child to his parent (LL)
    - Inserting the new node in right subtree to a parent who is a right child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

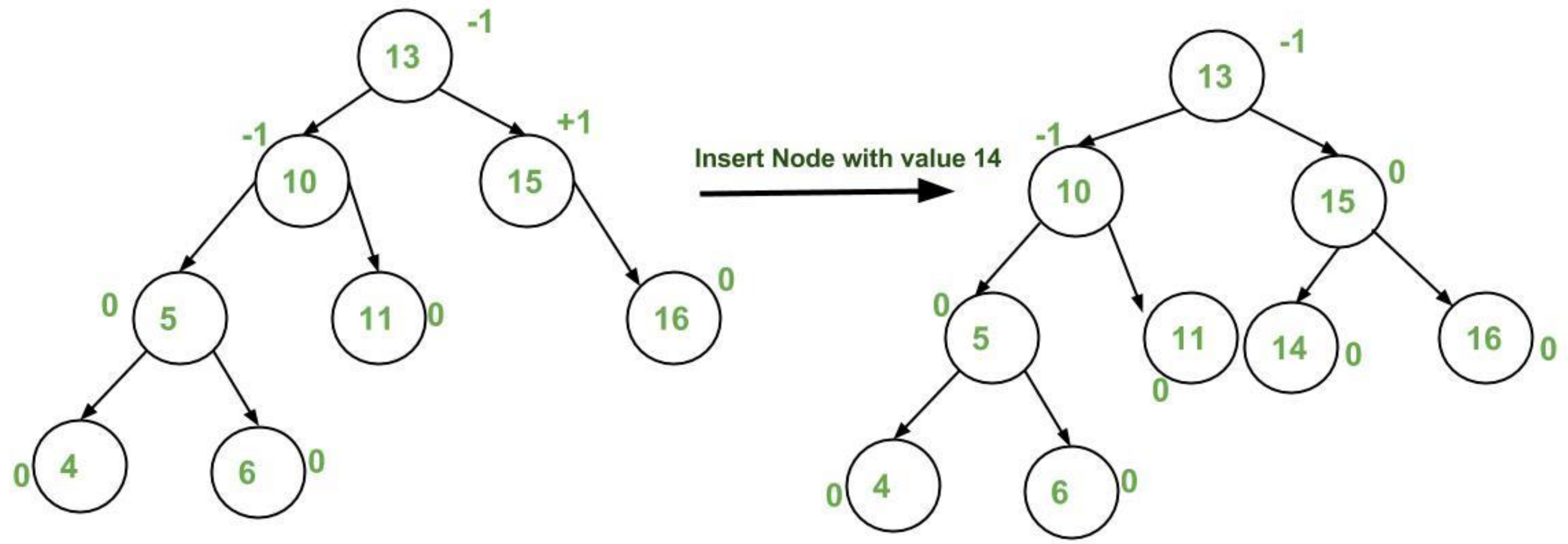
Courtesy of Wikipedia



AVL Insertion Algorithm

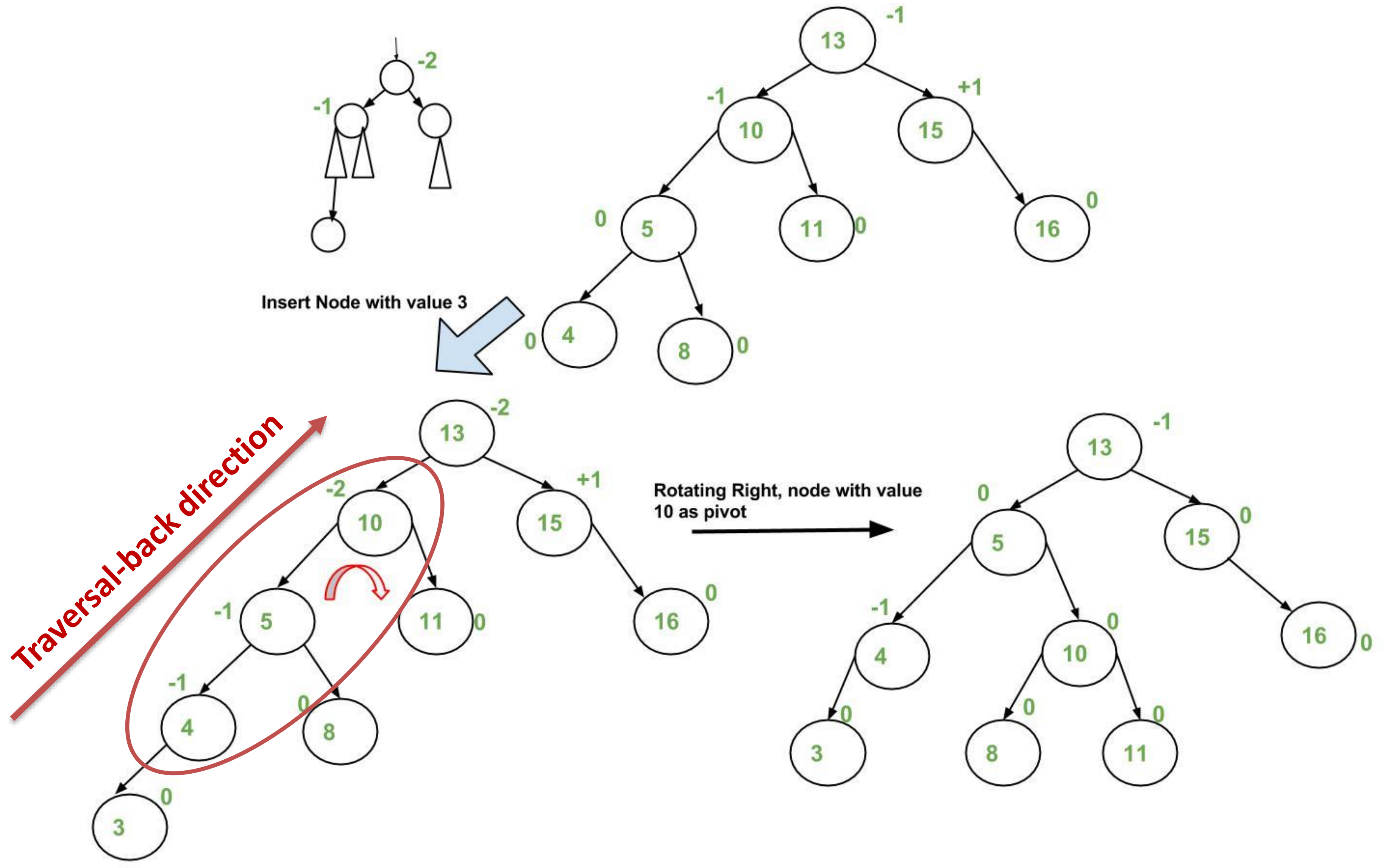
- Insert new node normally as in regular BST
- Traverse up in the tree in the reverse direction of the insertion
  - The first node in this traversal that has a difference of 2 between the heights of the two subtrees is the last of the three nodes needed for rotation.
  - The previous two nodes in this traversal are the other 2 nodes.
- According to the hierarchy of these 3 nodes, apply the corresponding case.

AVL: Insertion examples

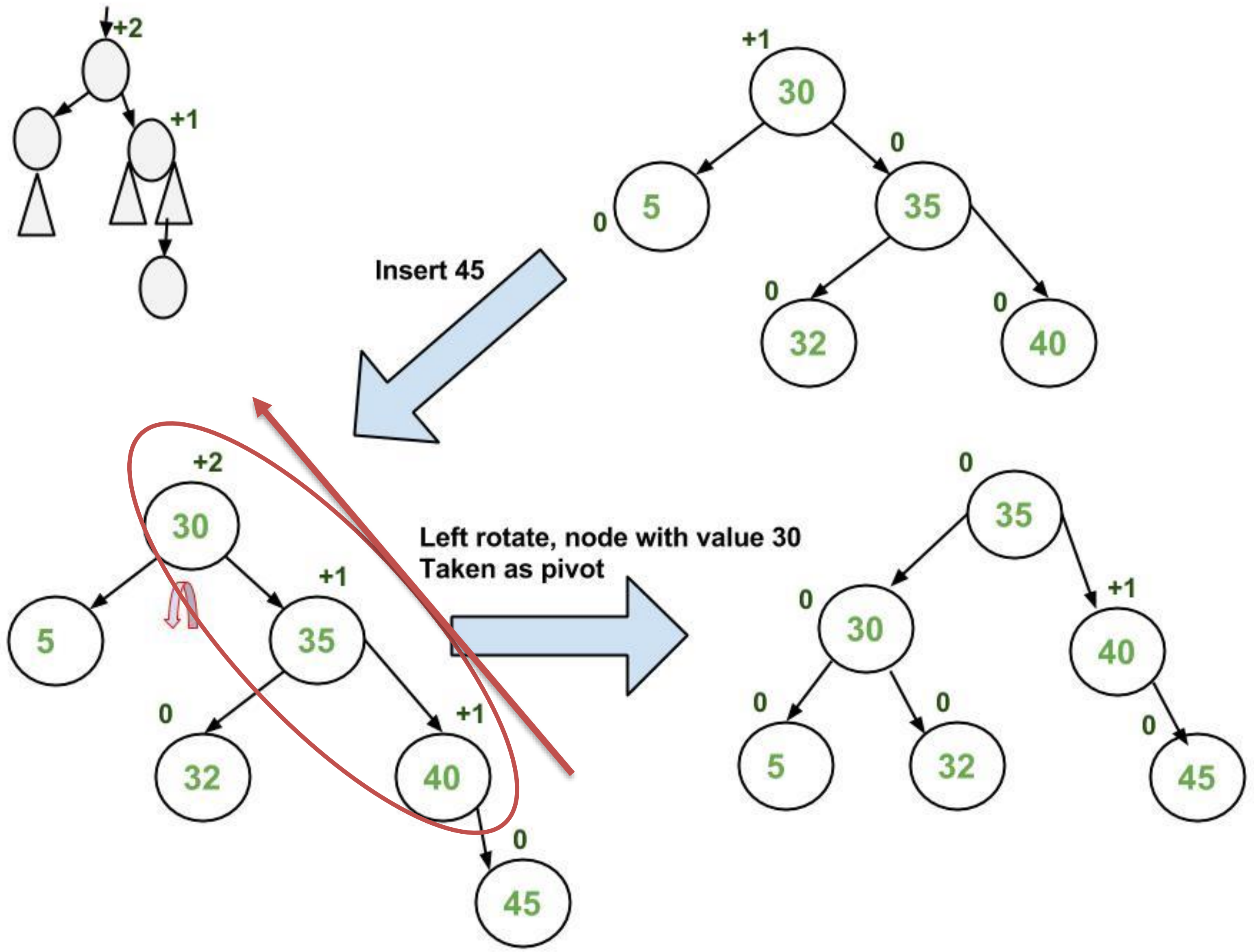


Going up through nodes 15, then 13, no imbalance found.

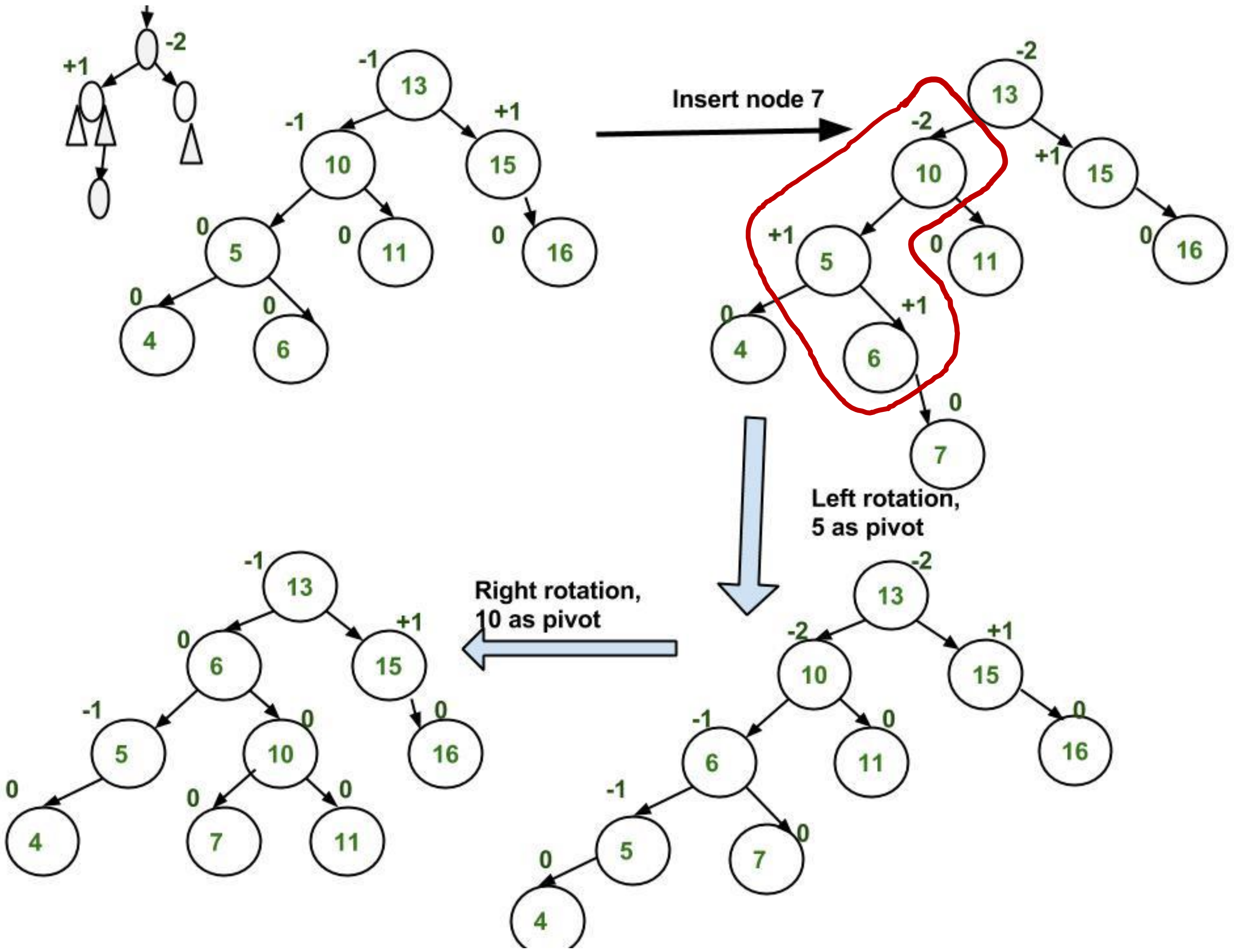
AVL: Insertion examples



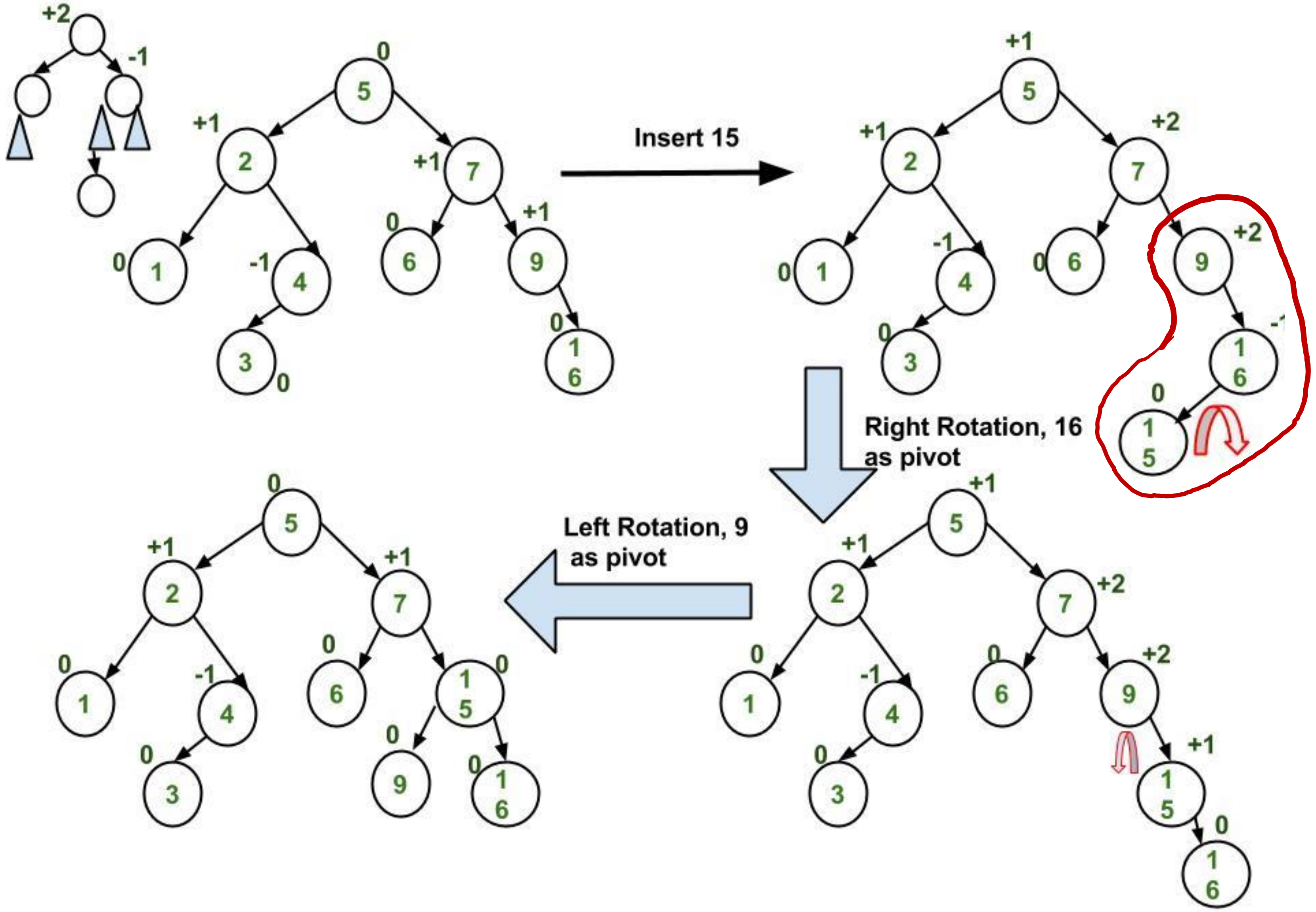
AVL: Insertion examples



AVL: Insertion examples



AVL: Insertion examples



AVL: Insertion examples

Courtesy of Wikipedia

AVL: Deletion

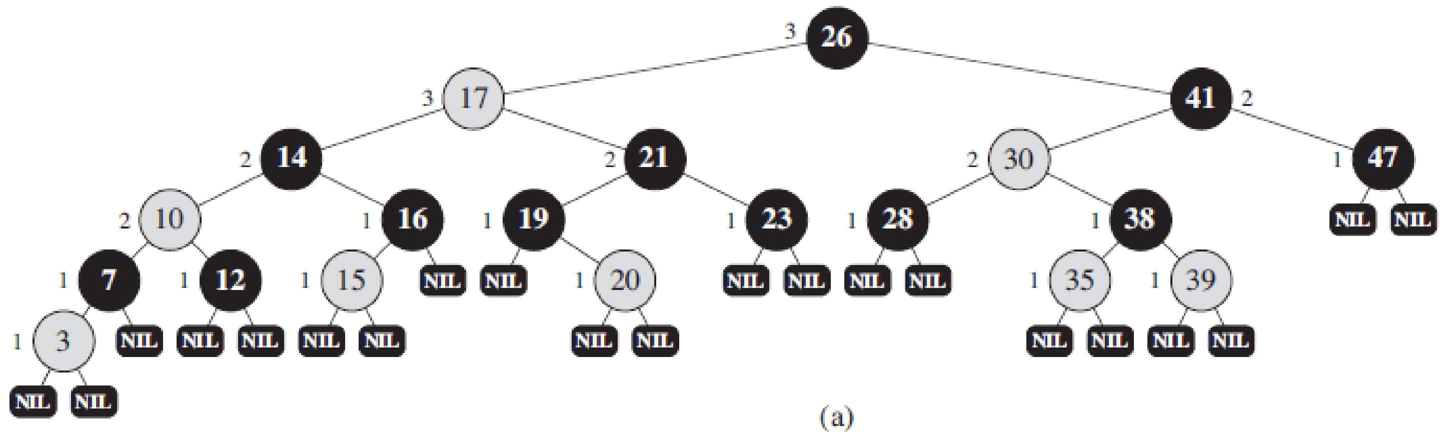
- Delete in the same way as in standard BST.
- After deletion, the same four cases may take place but now the 3 nodes to rotate will not remain the same as in the case of insertion.
- When a node is deleted, the subtree containing this node is reduced in height, so the sibling subtree will relatively exceed in height.
- The three nodes will 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



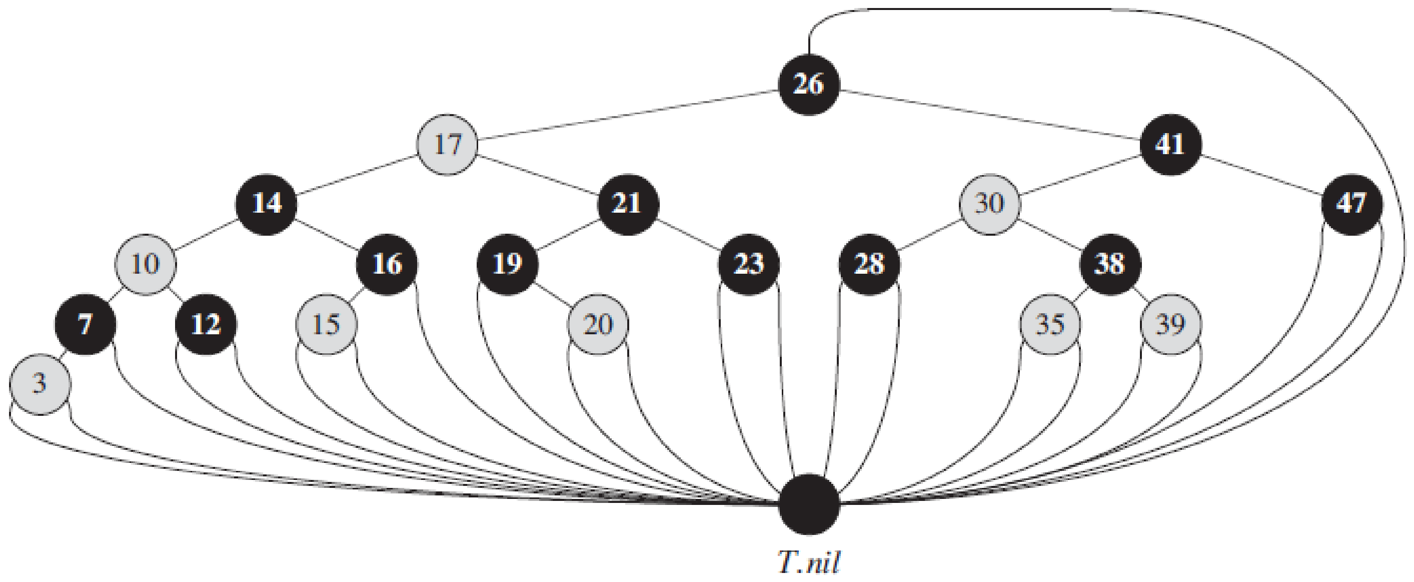
(a)

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.

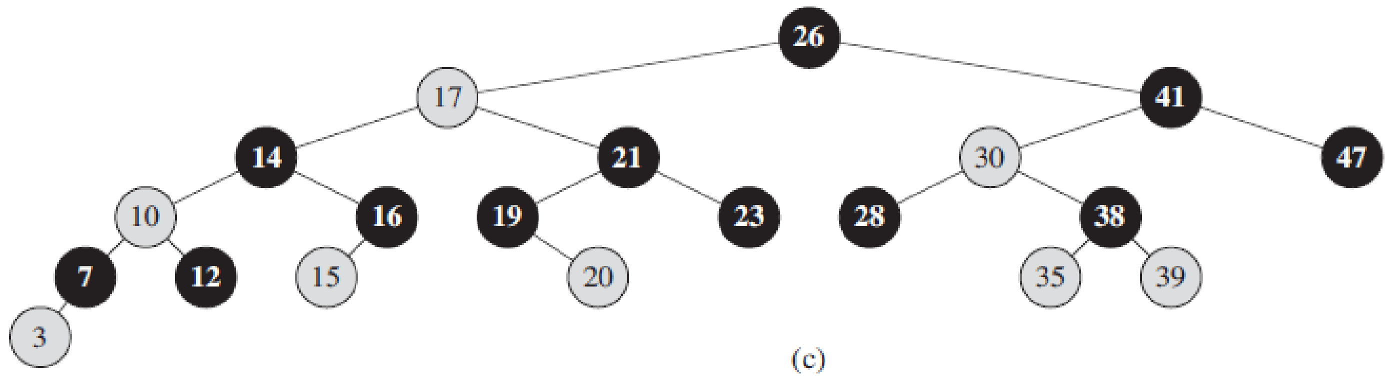
Lemma 13.1
A red-black tree with n internal nodes has height at most $2\lg(n+1)$.

Red-Black trees

- RB-tree typically implemented as having one nil node that all tree leaves point to as well as the root as shown in figure b.
- Later examples won't show the nil but they are assumed to exist as shown in figure c.



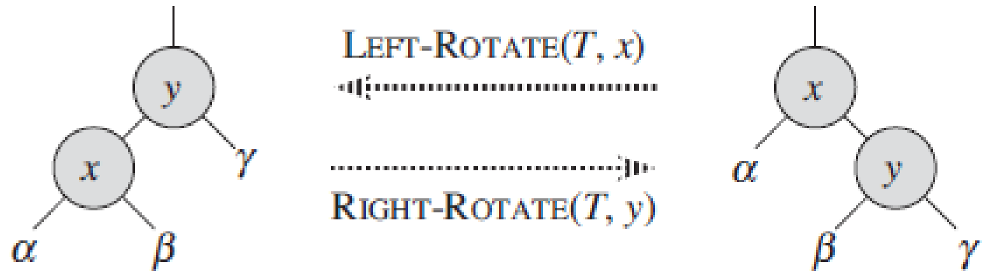
(b)



(c)

Red-Black trees

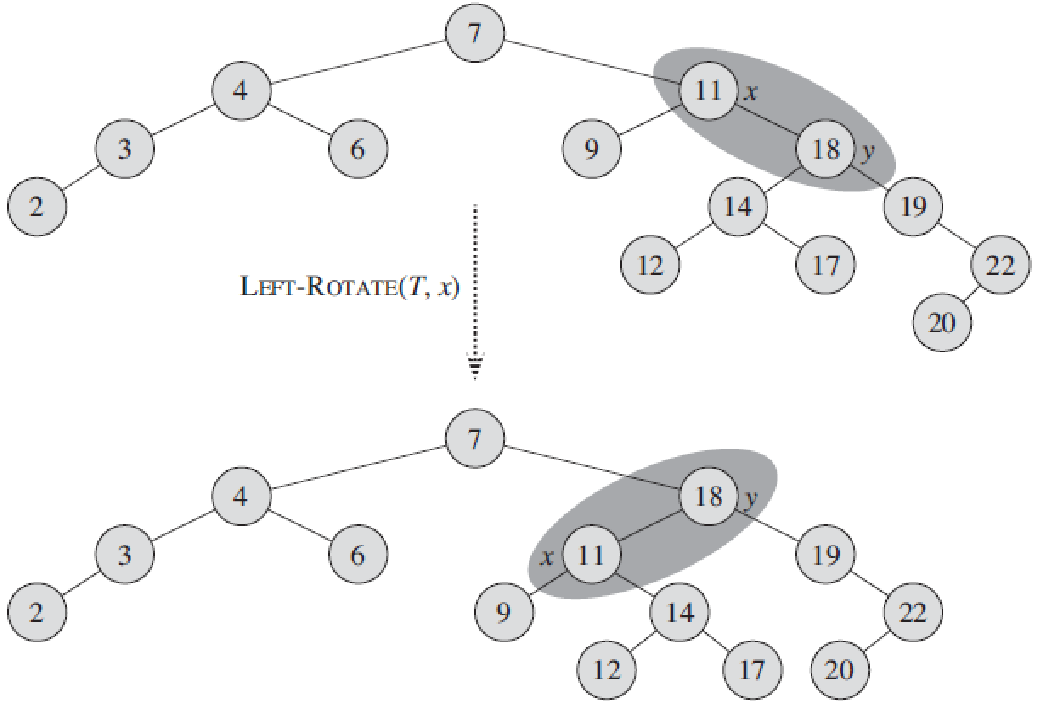
- Rotations



```
LEFT-ROTATE(T, x)
 1  $y = x.right$                          // set $y$
 2  $x.right = y.left$                     // turn $y$'s left subtree into $x$'s right subtree
 3  if $y.left \neq T.nil$
 4      $y.left.p = x$
 5  $y.p = x.p$                            // link $x$'s parent to $y$
 6  if $x.p == T.nil$
 7      $T.root = y$
 8  elseif $x == x.p.left$
 9      $x.p.left = y$
10  else $x.p.right = y$
11  $y.left = x$                           // put $x$ on $y$'s left
12  $x.p = y$
```

Red-Black trees

- Rotations



Red-Black trees

- Insertion

Standard BST insertion

```
RB-INSERT($T, z$)        // $z$: node to insert
y = T.nil
x = T.root
while $x \neq T.nil$
    $y = x$
    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($T, z$)
```

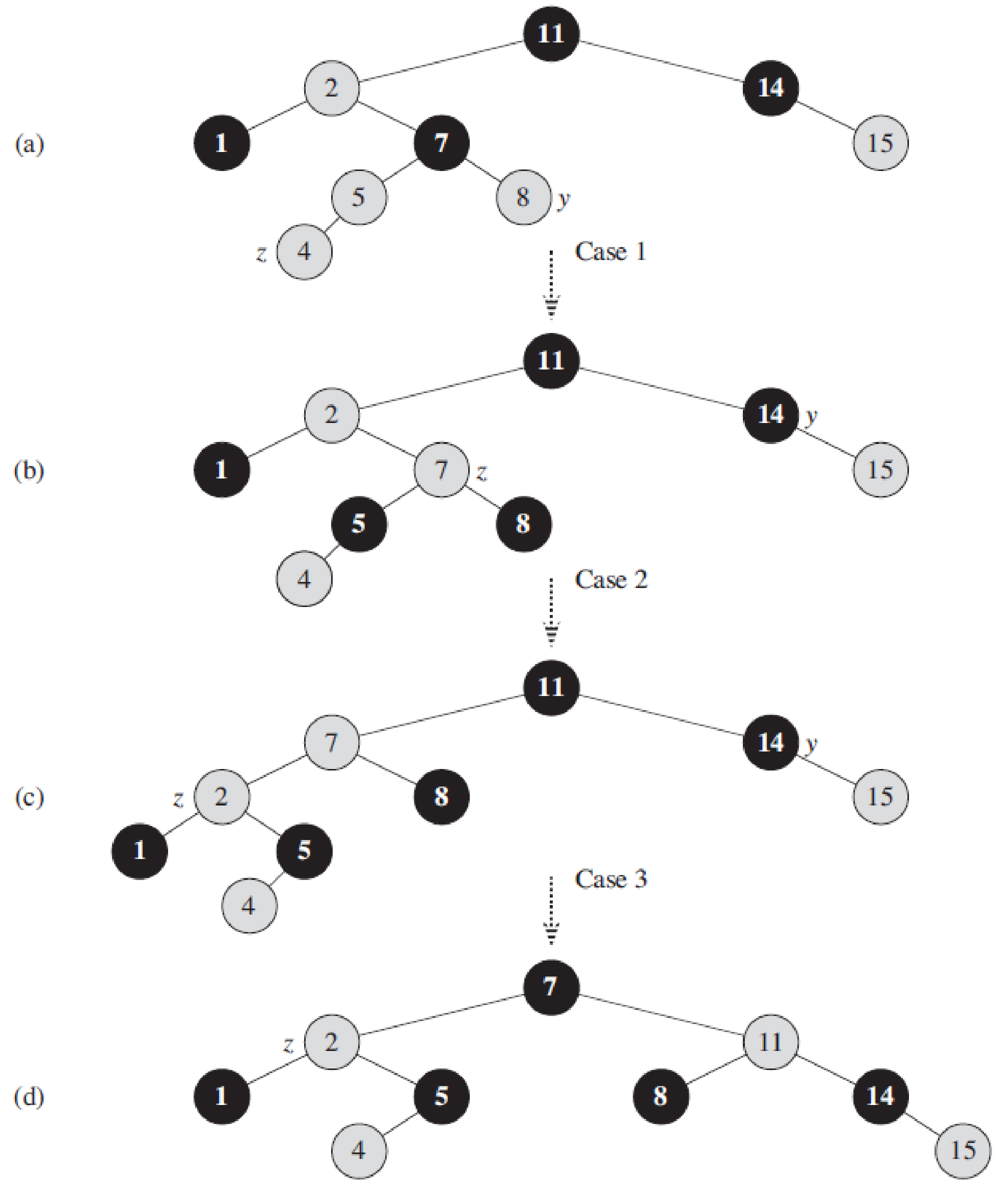
1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

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($T, z$)
 1  while $z.p.color$ == RED          // $z$: current node to process
 2      if $z.p == z.p.p.left$
 3          $y = z.p.p.right$
 4          if $y.color$ == RED
 5              $z.p.color$ = BLACK              // case 1
 6              $y.color$ = BLACK                // case 1
 7              $z.p.p.color$ = RED              // case 1
 8              $z = z.p.p$                      // case 1
 9          else if $z == z.p.right$
10                  $z = z.p$                    // case 2
11                  LEFT-ROTATE($T, z$)          // case 2
12              $z.p.color$ = BLACK              // case 3
13              $z.p.p.color$ = RED              // case 3
14              RIGHT-ROTATE($T, z.p.p$)         // case 3
15      else (same as then clause
                with "right" and "left" exchanged)
16  $T.root.color$ = BLACK
```



AVL vs Red-Black Trees

- AVL is faster in search because it is more balanced (thus better for DB where fast queries are crucial)
- 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.