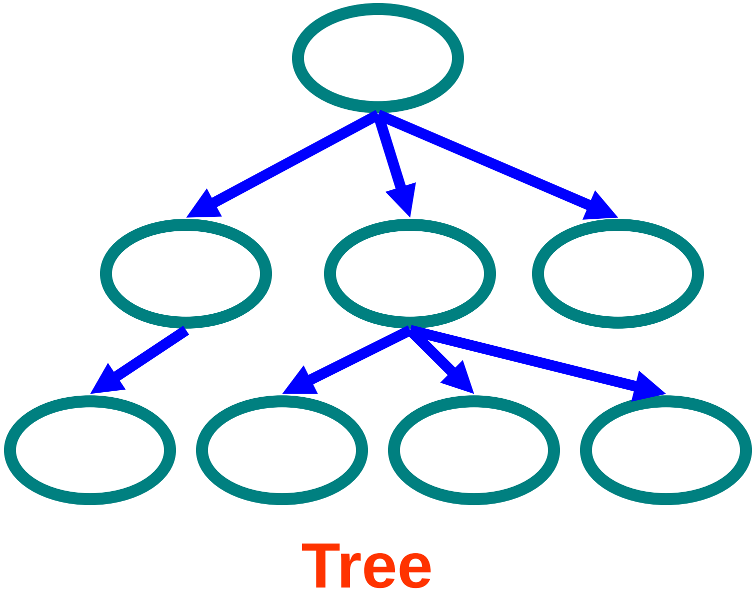
Lectures 14-15.
Binary Search Trees - AVL Trees

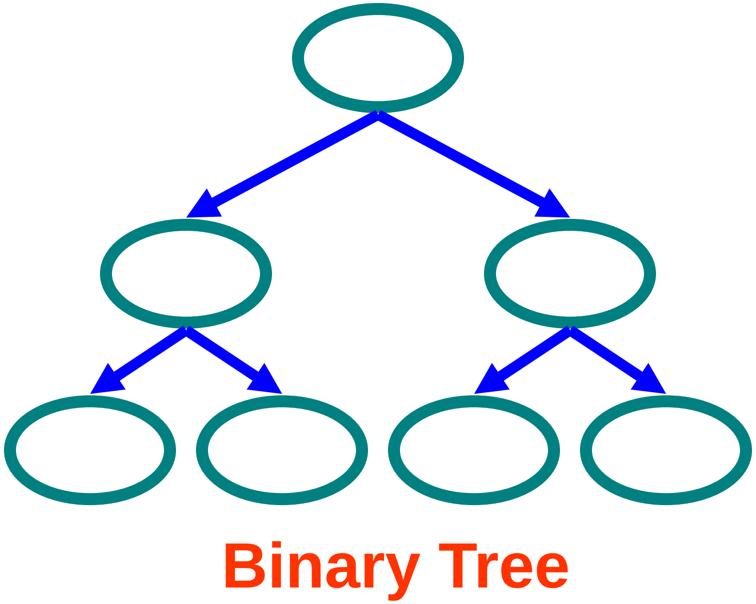
Introduction to Algorithms
Da Nang University of Science and Technology

Dang Thien Binh
dtbinh@dut.udn.vn

Introduction

- Tree
  - Single parent, multiple children
- Binary tree
  - Tree with 0-2 children per node



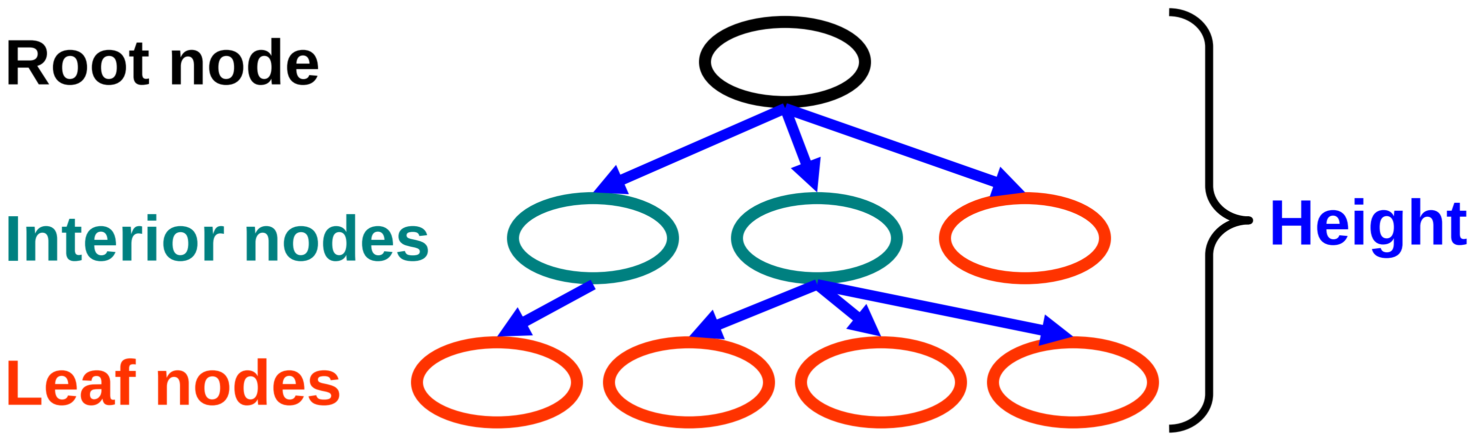


Introduction

- Search trees support
  - SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE
  - Dictionary and priority queue
  - Basic operations take time proportional to the height of the tree
    - Complete binary tree: $\Theta(\lg n)$
    - Linear-chain tree: $\Theta(n)$
    - Expected height of a randomly built binary search tree: $O(\lg n)$

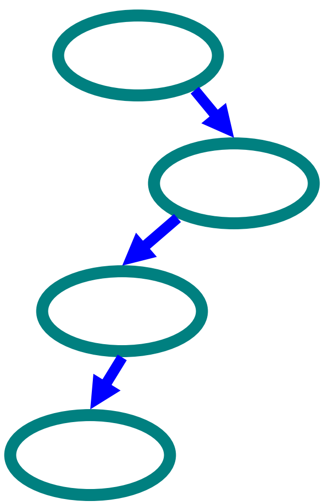
Trees

- Terminology
  - Root ⇒ no predecessor
  - Leaf ⇒ no successor
  - Interior ⇒ non-leaf
  - Height ⇒ distance from root to leaf

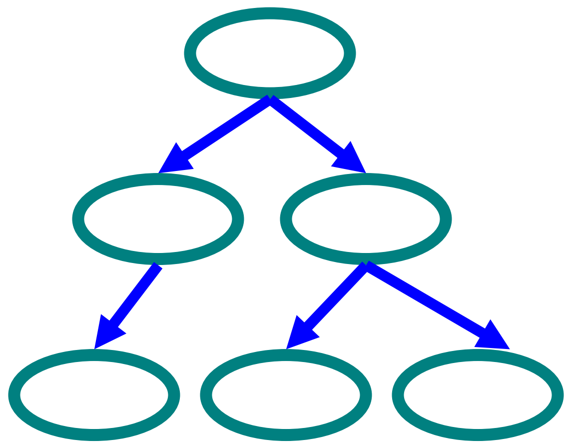


Types of Binary Trees

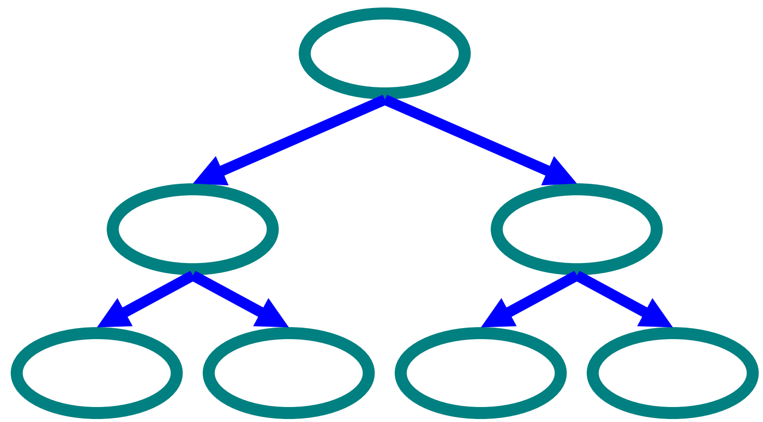
- Degenerate – only one child
- Balanced – mostly two children
- Complete – always two children



Degenerate Binary Tree



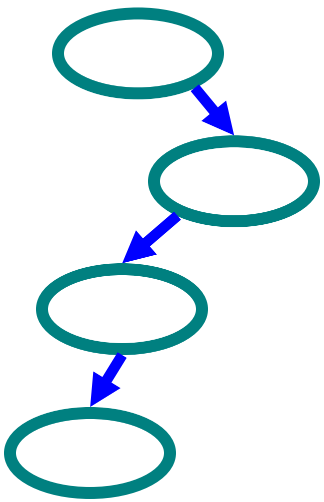
Balanced Binary Tree



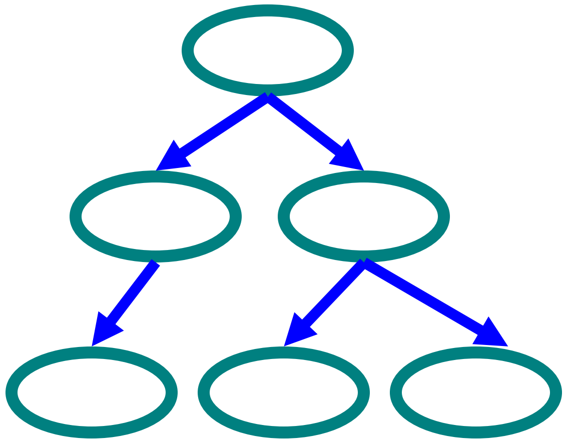
Complete Binary Tree

Binary Trees Properties

- Degenerate
  - Height = O(n) for n nodes
  - Similar to linear list
- Balanced
  - Height = O(lg *n*) for n nodes
  - Useful for searches



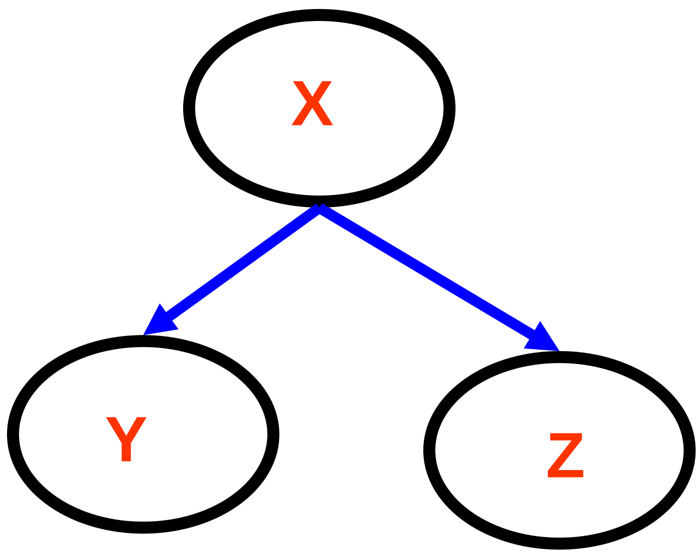
Degenerate Binary Tree



Balanced Binary Tree

Binary Search Trees

- Key property
  - Value at node
    - Smaller values in left subtree
    - Larger values in right subtree
  - Example
    - X > Y
    - X < Z

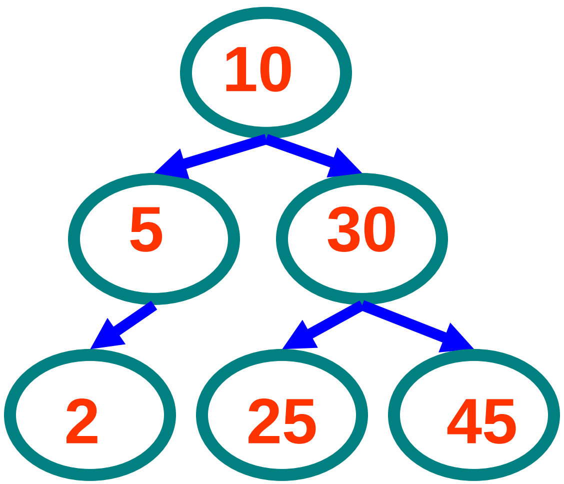


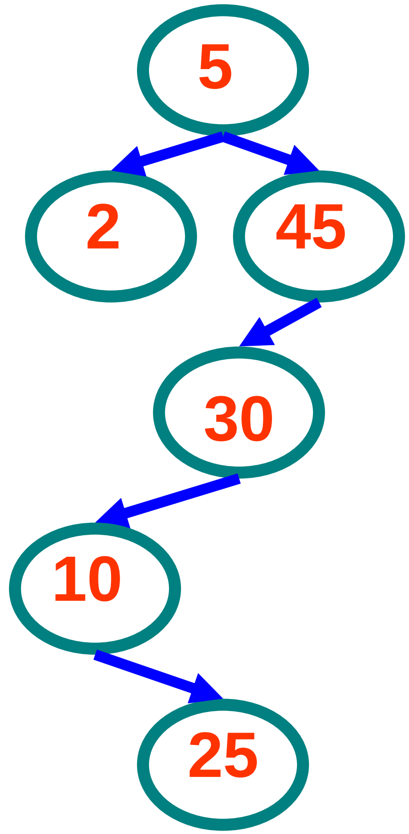
Binary Search Trees

- Binary Search Tree (BST)
  - Binary tree
  - Represented by a linked data structure
    → each node is an object
    - key + satellite data
    - Pointers: *left* → left child, *right* → right child, *p* → parent
- Binary search tree property
  - Let x be a node in a BST
    - If y is a node in the left subtree of x, then $y.key \leq x.key$
    - If y is a node in the right subtree of x, then $y.key \geq x.key$

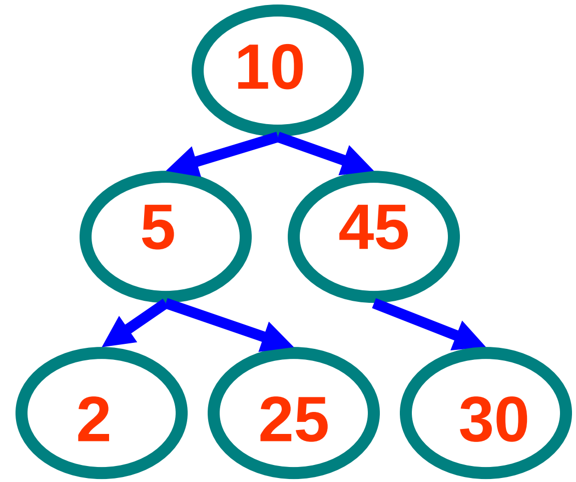
Binary Search Trees

- Examples





Binary Search Trees



Non-Binary Search Tree

Tree Traversal

- Tree Traversal
  - A technique for processing the nodes of a tree in some order
- Preorder traversal
  - Process all nodes of a tree by processing the root, then recursively processing all subtrees
  - Also known as prefix traversal
  -

```
void preorder(tree_ptr ptr) {
    if(ptr) {
        printf("%d", ptr->data);
        preorder(ptr->left_child);
        preorder(ptr->right_child);
    }
}
```

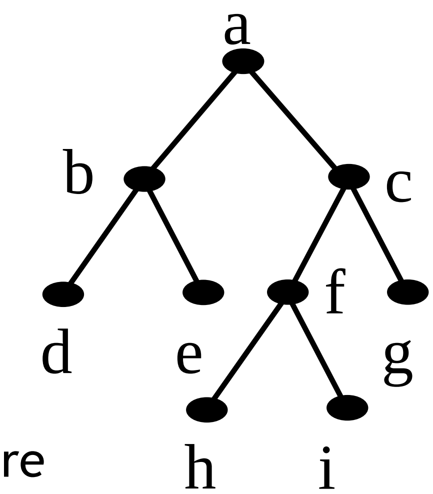
Tree Traversal

- Inorder traversal
  - Process all nodes of a tree by recursively processing the left subtree, then processing the root, and finally, the right subtree
  - ```
    void inorder(tree_ptr ptr) {
        if(ptr) {
            inorder(ptr->left_child);
            printf("%d",ptr->data);
            inorder(ptr->right_child);
        }
    }
    ```
    ```

Tree Traversal

- Postorder traversal
  - Process all nodes of a tree by recursively processing all subtrees, then finally processing the root
  - Also known as postfix traversal
  -
```
void postorder(tree_ptr ptr) {
    if (ptr) {
        postorder(ptr->left_child);
        postorder(ptr->right_child);
        printf("%d", ptr->data);
    }
}
```

Tree Traversal



- Tree Traversal
  - Preorder
    - The key of the root of a subtree is printed before the values in its left subtree and those in its right subtree
    - a, b, d, e, c, f, h, i, g
  - Inorder
    - The key of the root of a subtree is printed between the values in its left subtree and those in its right subtree
    - d, b, e, a, h, f, i, c, g
  - Postorder
    - The key of the root of a subtree is printed after the values in its left subtree and those in its right subtree
    - d, e, b, h, i, f, g, c, a

Computation Time of INORDER

- If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ time
  - Use the substitution method
  - $T(n) = T(k) + T(n-k-1) + d;\ T(0) = c$
  - Show that $T(n) = \Theta(n)$ by proving that $T(n) = (c+d)n + c$
    - For $n = 0 \Rightarrow (c+d) * 0 + c = c$
    - For $n > 0$
      - $T(n) = T(k) + T(n-k-1) + d$
        $= ((c+d) * k + c) + ((c+d) * (n-k-1) + c) + d$
        $= (c+d) * n + c - (c+d) + c + d = (c+d) * n + c$

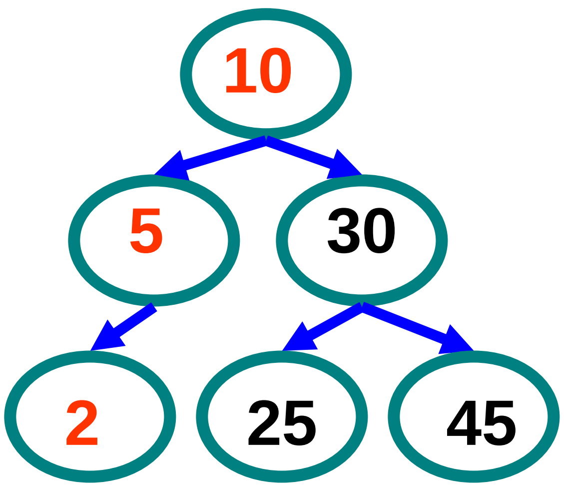
Searching

```
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)
```

```
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
```

Example Binary Searches

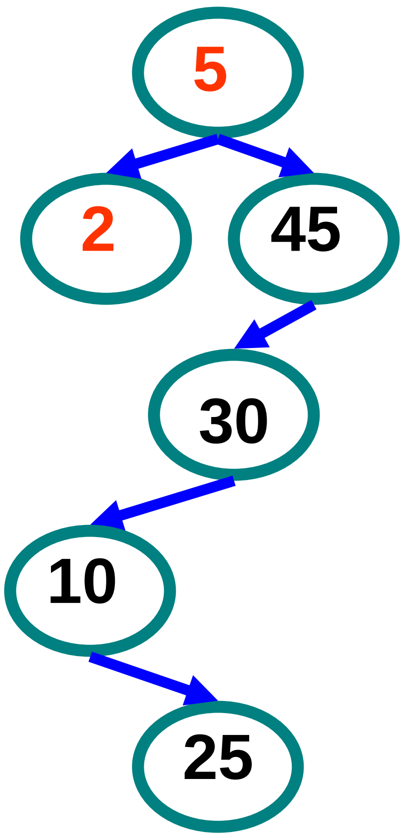
- Find (2)



10 > 2, left

5 > 2, left

2 = 2, found

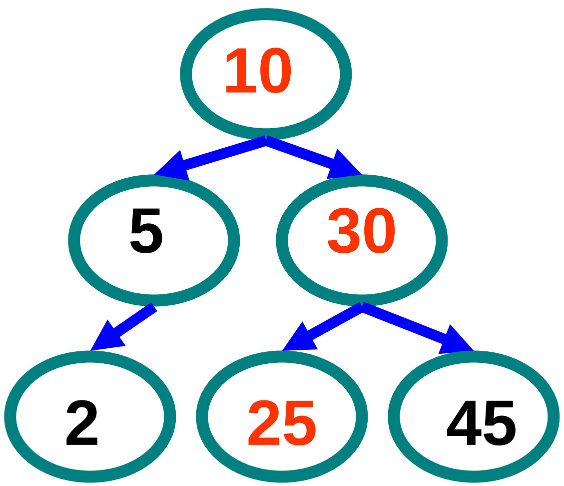


5 > 2, left

2 = 2, found

Example Binary Searches

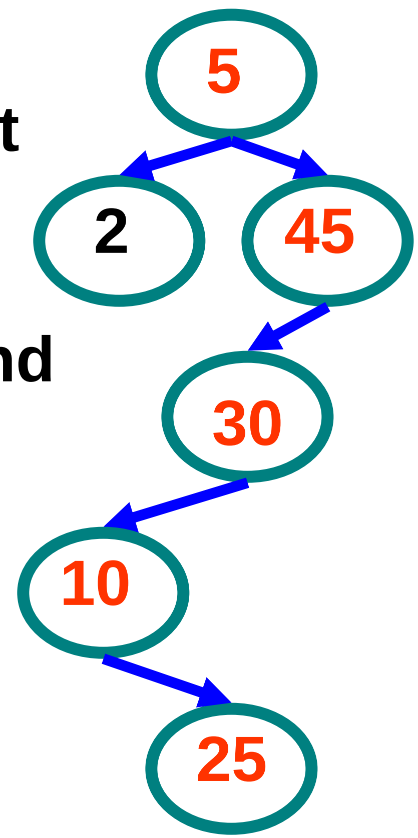
- Find (25)



10 < 25, right

30 > 25, left

25 = 25, found



5 < 25, right

45 > 25, left

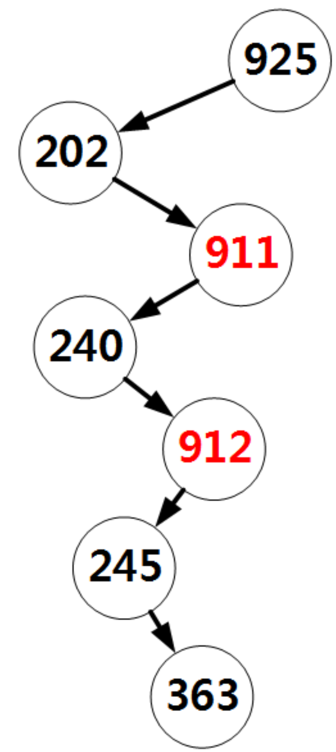
30 > 25, left

10 < 25, right

25 = 25, found

Practice Problems

- Suppose that we have numbers between 1 and 1000 in a binary search tree and want to search for the number 363. Which of the following sequences could NOT be the sequence of nodes examined?
  a. 2, 252, 401, 398, 330, 344, 397, 363.
  b. 924, 220, 911, 244, 898, 258, 362, 363.
  c. 925, 202, 911, 240, 912, 245, 363.



Minimum and Maximum

- The element with the minimum/maximum key can be found by following left/right child pointers from the root until NIL is encountered
  - TREE-MINIMUM(x)

    ```
    while x.left ≠ NIL
        x = x.left
    return x
    ```

  - TREE-MAXIMUM(x)

    ```
    while x.right ≠ NIL
        x = x.right
    return x
    ```

Successor and Predecessor

- Assume distinct keys
- The successor of a node x is the node with the smallest key greater than $x.key$
  - Right subtree nonempty → successor is the leftmost node in the right subtree (Line 2)
  - Otherwise, the successor is the lowest ancestor of x whose left child is also an ancestor of x → go up the tree until we encounter a node that is the left child of its parent (Lines 3-7)

TREE-SUCCESSOR(X)

```
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
```

Examples

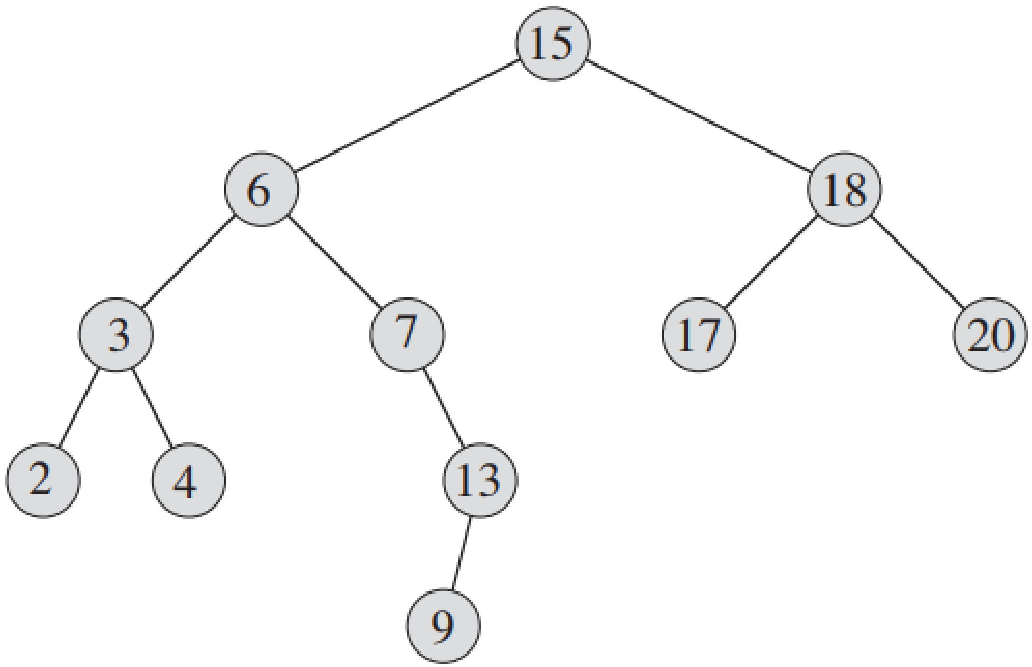


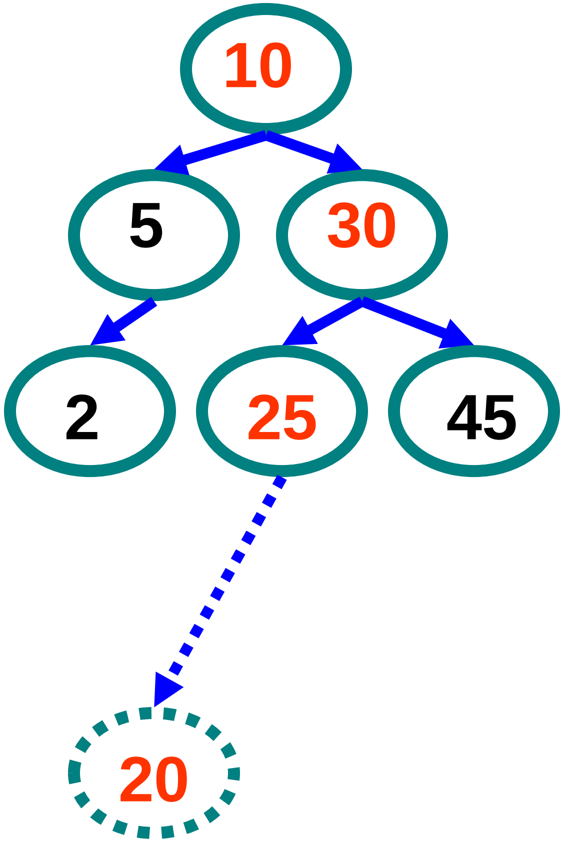
Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \to 6 \to 7 \to 13$ from the root. The minimum key in the tree is 2, which is found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

Insertion

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL                      Trace down the tree until a leaf
    y = x
    if z.key < x.key
        x = x.left                 Should be inserted in the left subtree
    else x = x.right               Should be inserted in the right subtree
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
```

Insertion Example (1/2)

- Insert (20)



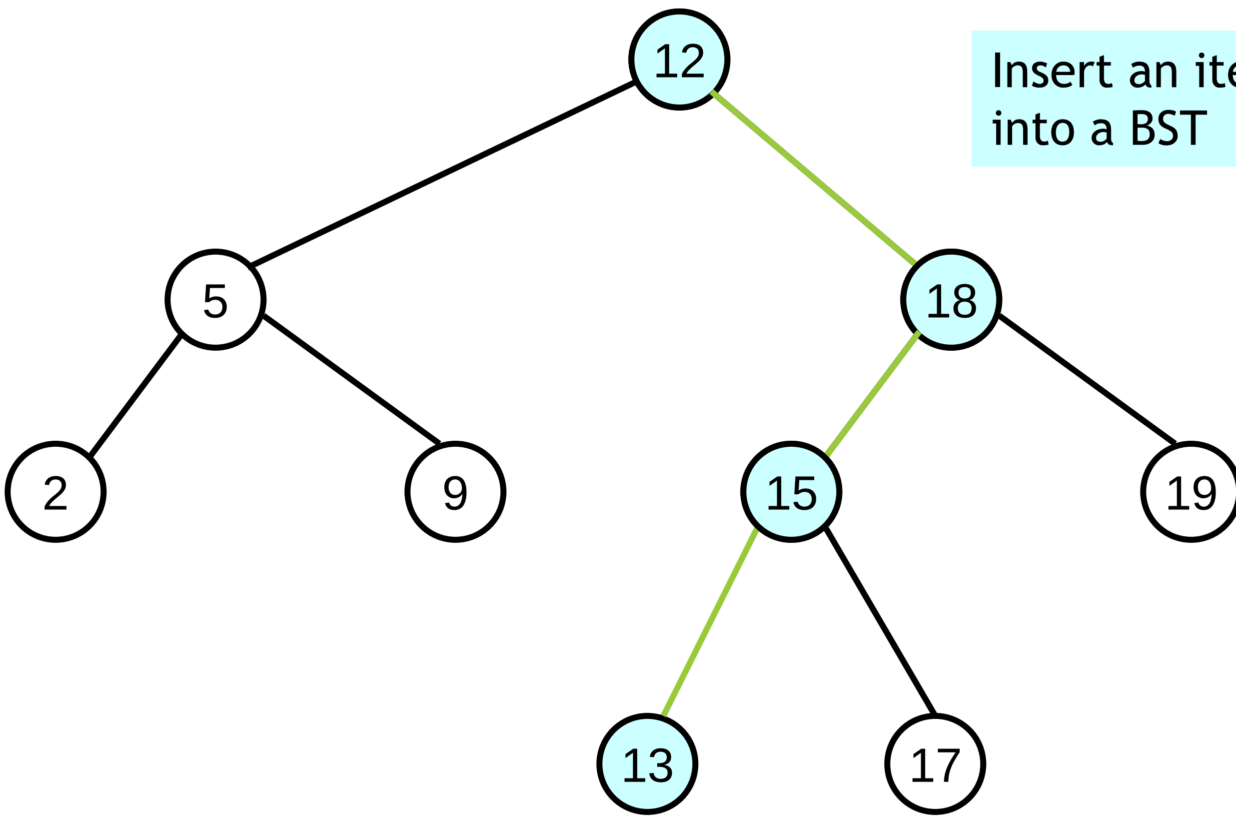
10 < 20, right

30 > 20, left

25 > 20, left

Insert 20 on left

Insertion Example (2/2)

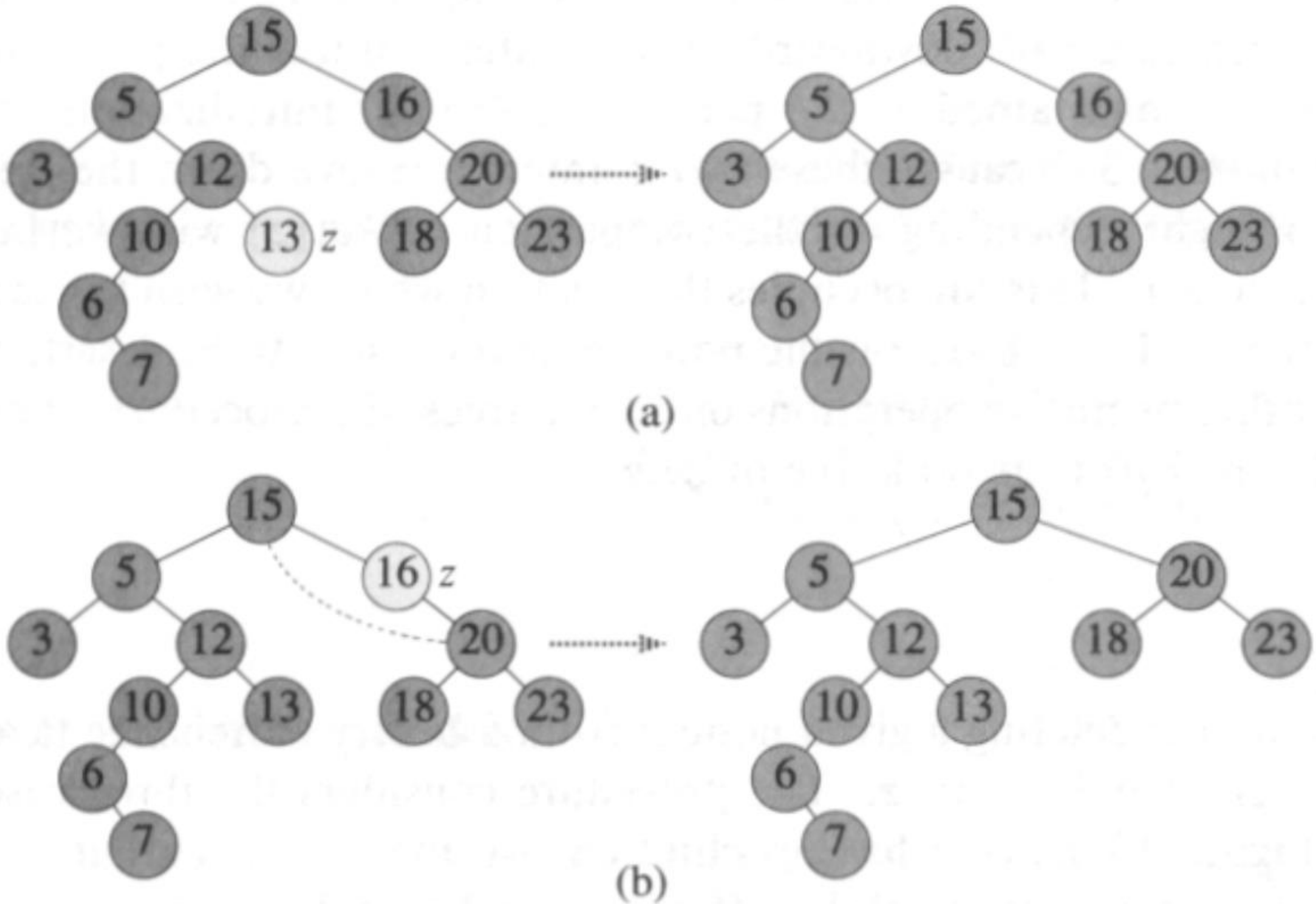


Insert an item with key 13 into a BST

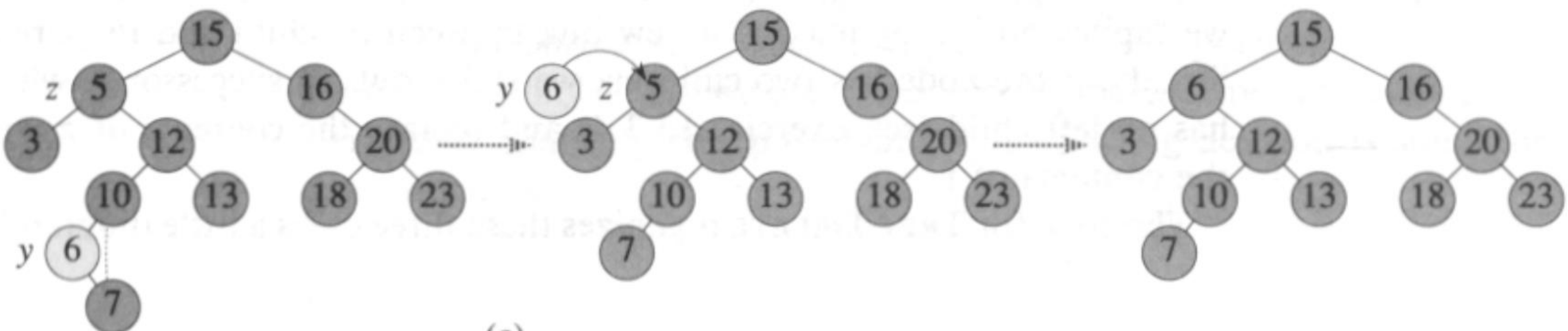
Deletion

- Consider three cases for the node z to be deleted
  - z has no children: modify $z.p$ to replace z with NIL as its child
  - z has only one child: splice out z by making a new link between its child and its parent
  - z has two children: splice z’s successor y, which has no left child and replace z’s key and satellite data with y’s key and satellite

Deletion Example (1/2)



Deletion Example (2/2)



(c)

Figure 12.4 Deleting a node z from a binary search tree. In each case, the node actually removed is lightly shaded. **(a)** If z has no children, we just remove it. **(b)** If z has only one child, we splice out z. **(c)** If z has two children, we splice out its successor y, which has at most one child, and then replace the contents of z with the contents of y.

TREE-DELETE Algorithm

```
TREE-DELETE (T, z)

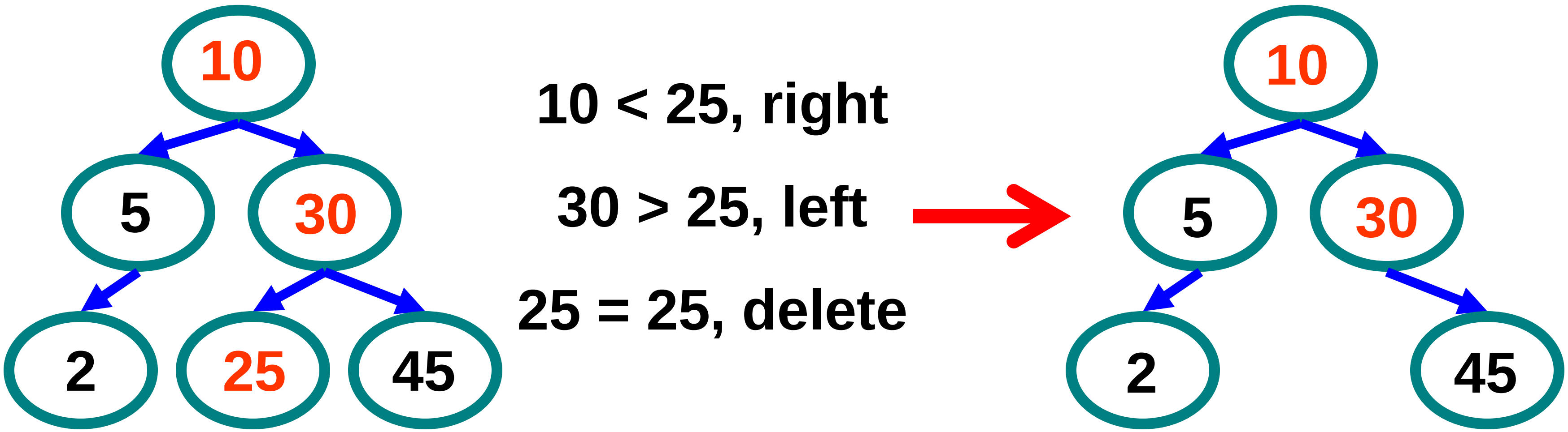
if z.left == NIL or z.right == NIL
    y = z
else y = TREE-SUCCESSOR (z)
if y.left ≠ NIL
    x = y.left
else x = y.right
if x ≠ NIL
    x.p = y.p
if y.p == NIL
    T.root = x
else if y == y.p.left
            y.p.left = x
        else y.p.right = x
if y ≠ z
    z.key = y.key
    copy y's satellite data into z
return y
```

TREE-DELETE Algorithm

- TREE-DELETE Algorithm
  - Lines 1-3: determines a node y to splice out (z itself or z's successor)
  - Lines 4-6: x is set to the non-NIL child of y, or to NIL if y has no children
  - Lines 7-13: splice out y by modifying pointers in $p[y]$ and x
    - Special care for when x=NIL or when y is the root
  - Lines 14-16: if the successor of z was the node spliced out, y's key and satellite data are moved to z, overwriting the previous key and satellite data
  - Line 17: return node y for recycle it via the free list

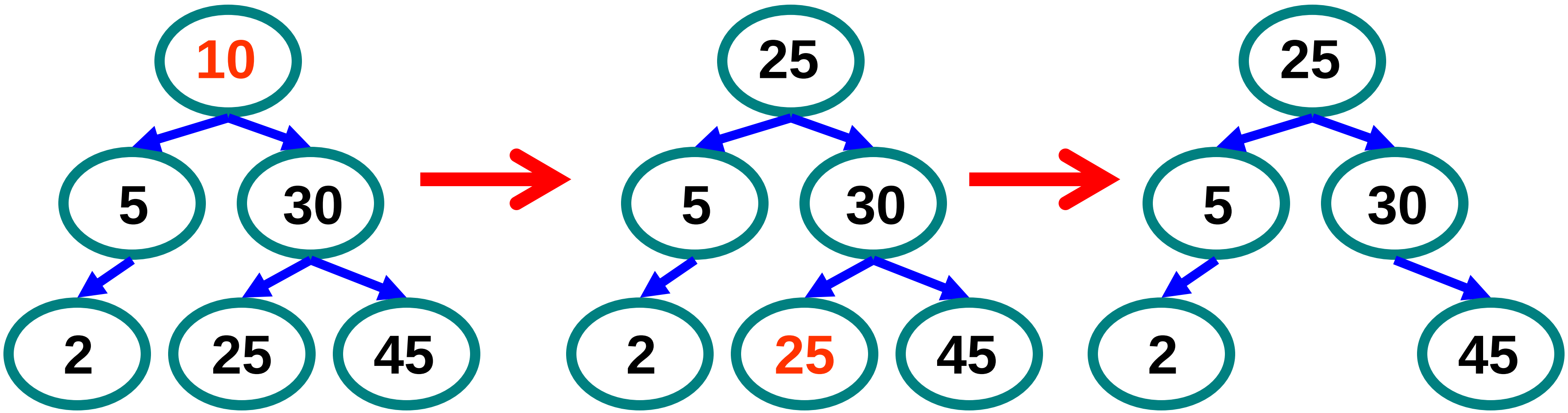
Example Deletion (Leaf)

- Delete (25)



Example Deletion (Internal Node)

- Delete (10)



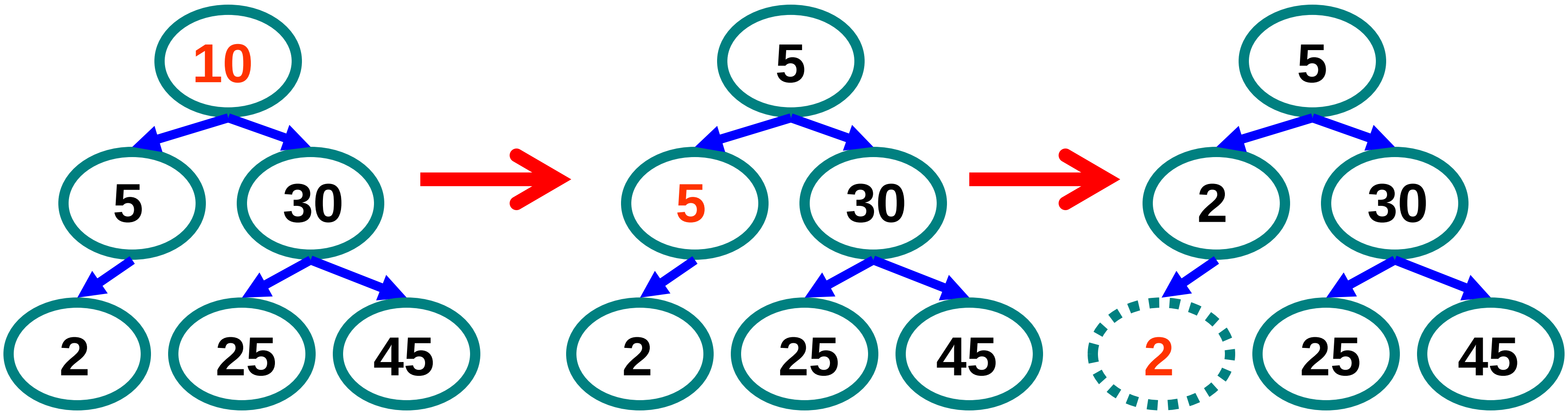
Replacing 10 with smallest value in right subtree

Deleting leaf

Resulting tree

Example Deletion (Internal Node)

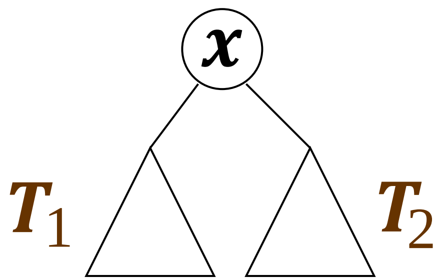
- Delete (10)



Replacing 10 with largest value in left subtree	Replacing 5 with largest value in left subtree	Deleting leaf

AVL Trees

- AVL Trees (Adelson-Velskii and Landis)
  - BST : all values in $T_1 \leq x$ all values in T_2

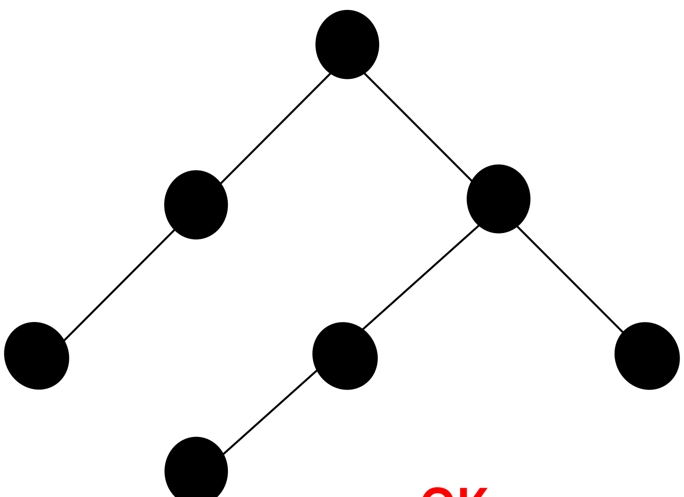


  - T_1, T_2 are AVL trees and $\left|\text{height}(T_1) - \text{height}(T_2)\right| \leq 1$

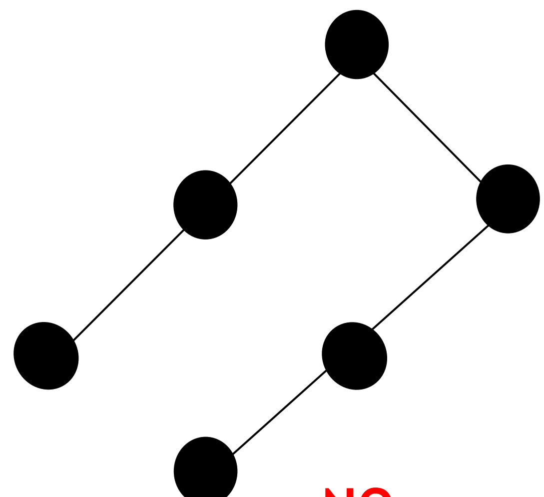
AVL Trees

<u>AVL Tree Property:</u> It is a BST in which the heights of the left and right subtrees of the root differ by at most 1 **and** in which the right and left subtrees are also AVL trees

Example: Select a AVL tree !



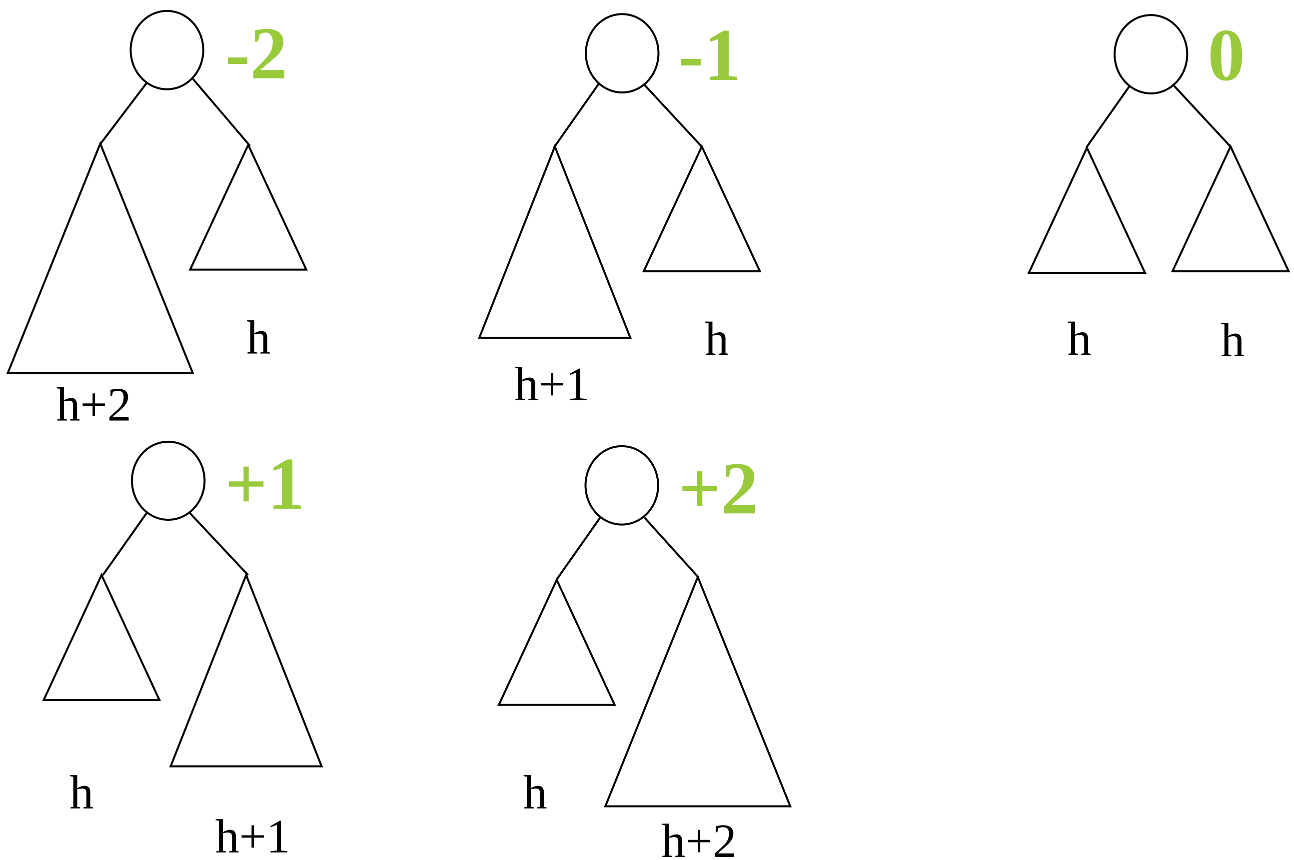
OK



NO

Insertion

- Very much like as in BST with operations to maintain height balance-rotation code associated with each node
  -2, -1, 0, 1, 2

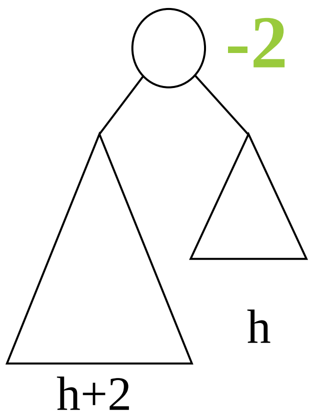


Rotation (1/7)

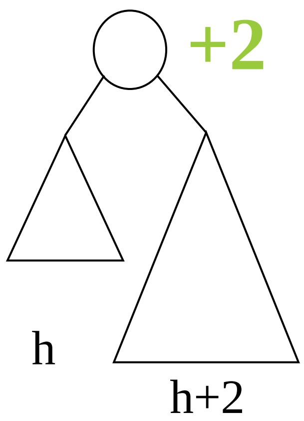
- 4 Types
  - Single Left Rotation
  - Double Left Rotation
  - Single Right Rotation
  - Double Right Rotation
    - SLR, DLR : left (+2)
    - SRR, DRR : right (-2)

Rotation (2/7)

- Case 1

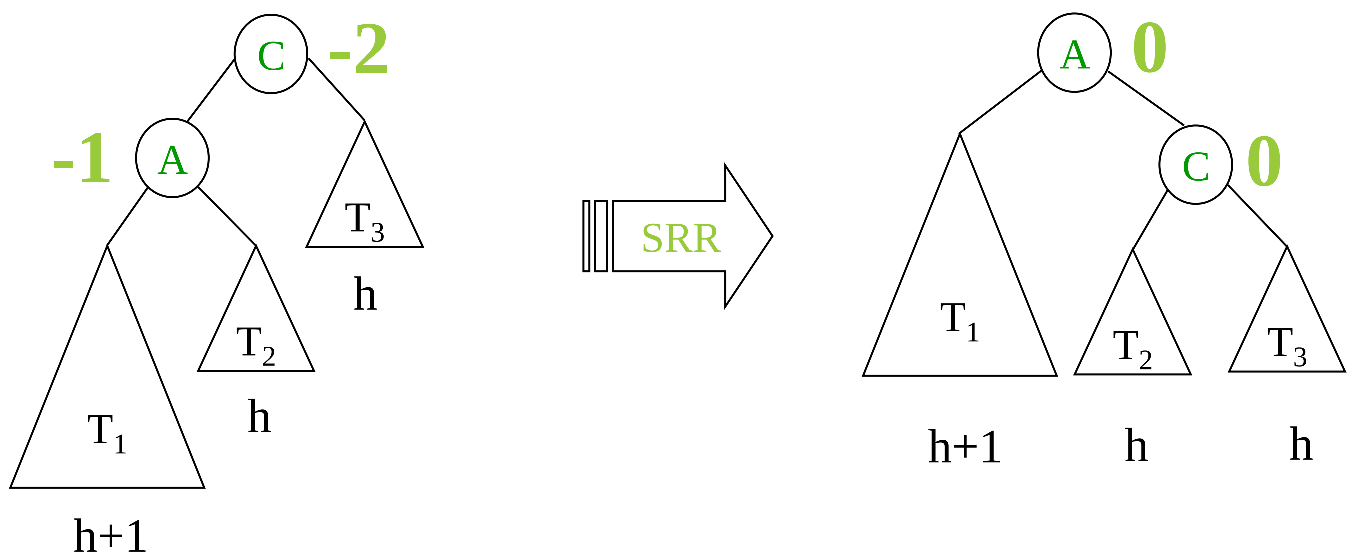


- Case 2



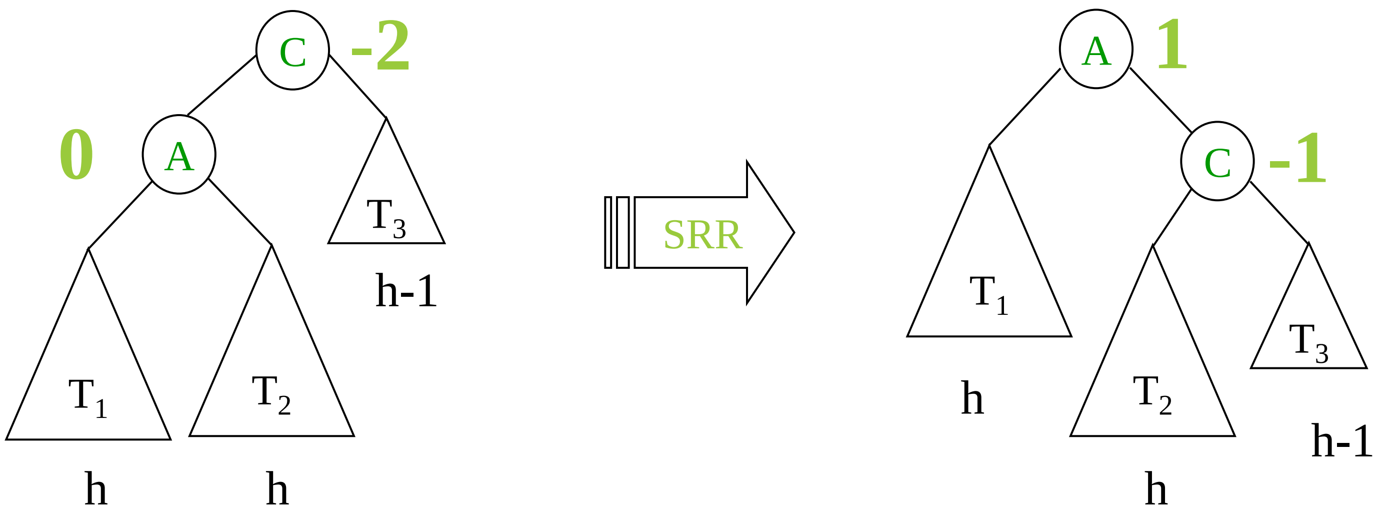
Rotation (3/7)

- Case 1 (a)



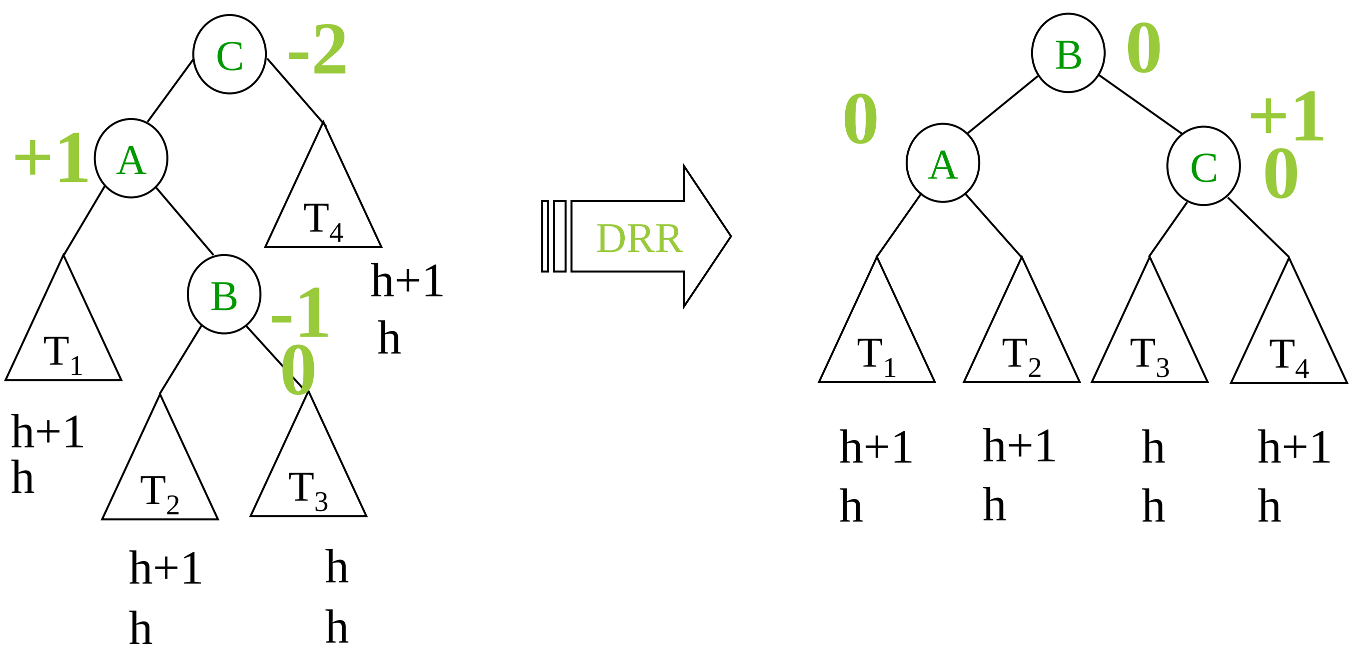
Rotation (4/7)

- Case 1 (a)



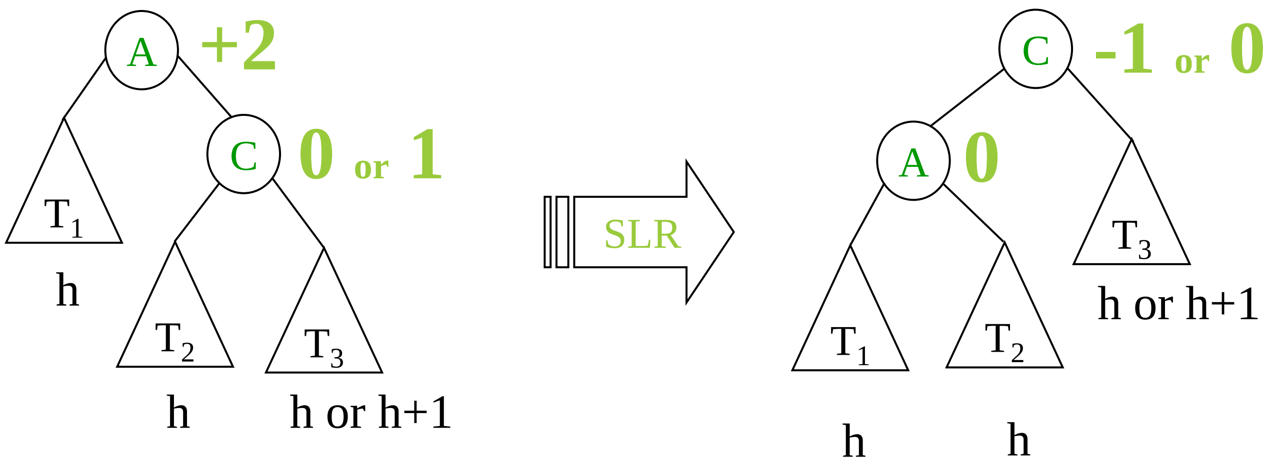
Rotation (5/7)

- Case 1 (b)



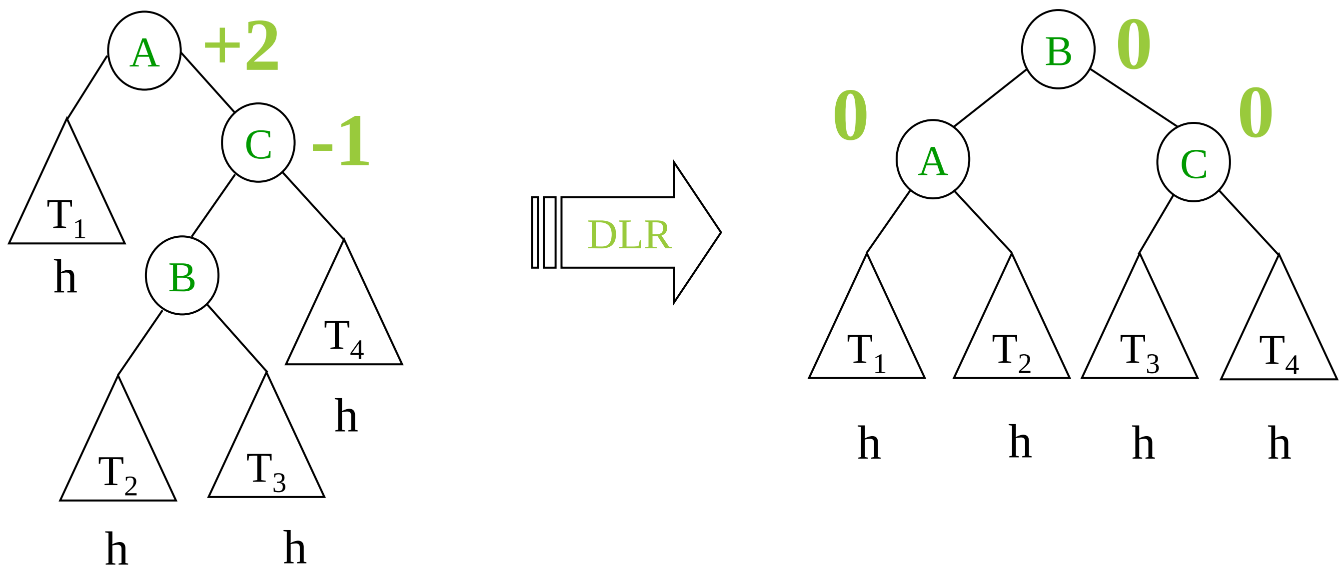
Rotation (6/7)

- Case 2 (a)



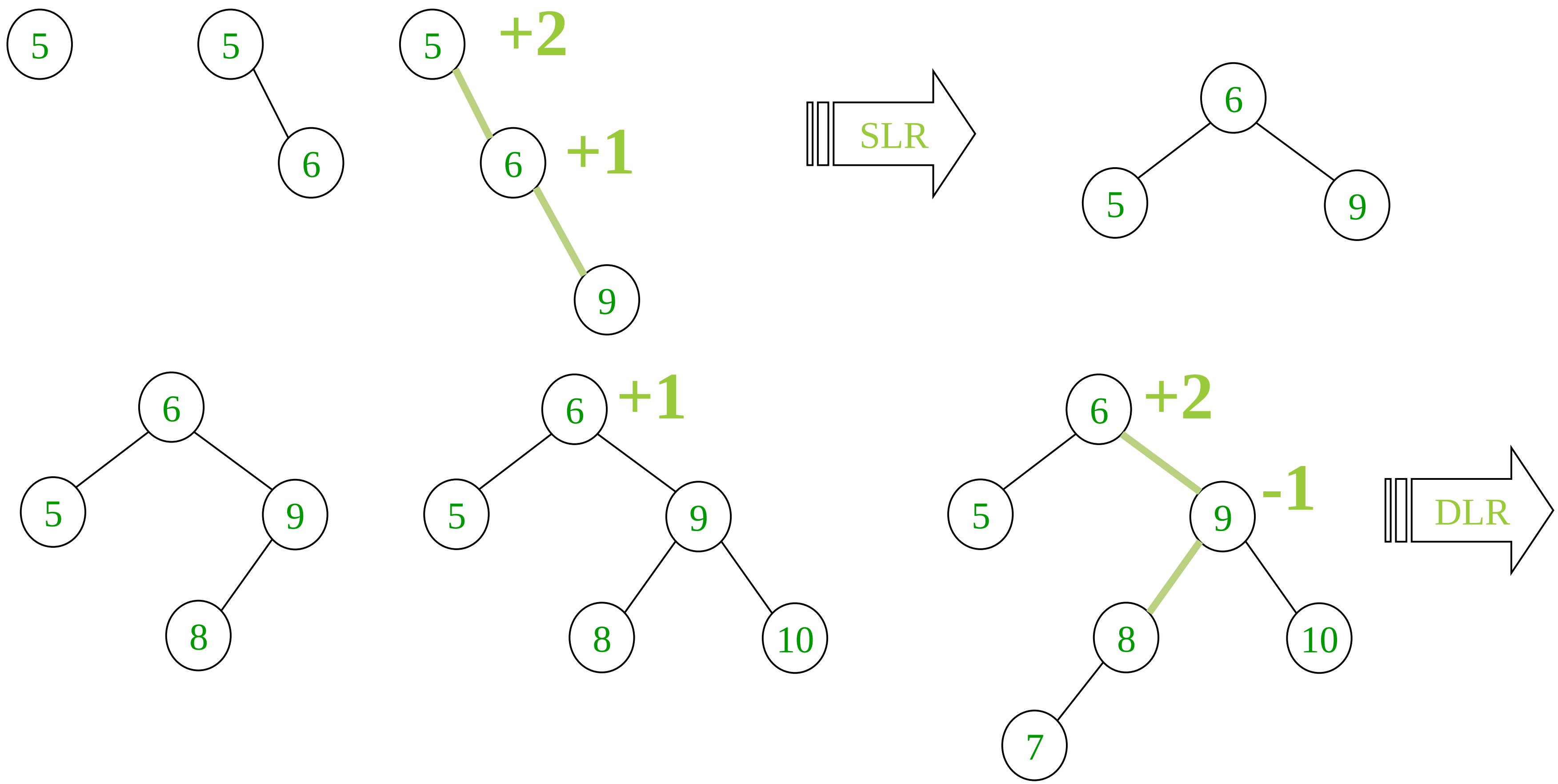
Rotation (7/7)

- Case 2 (b)



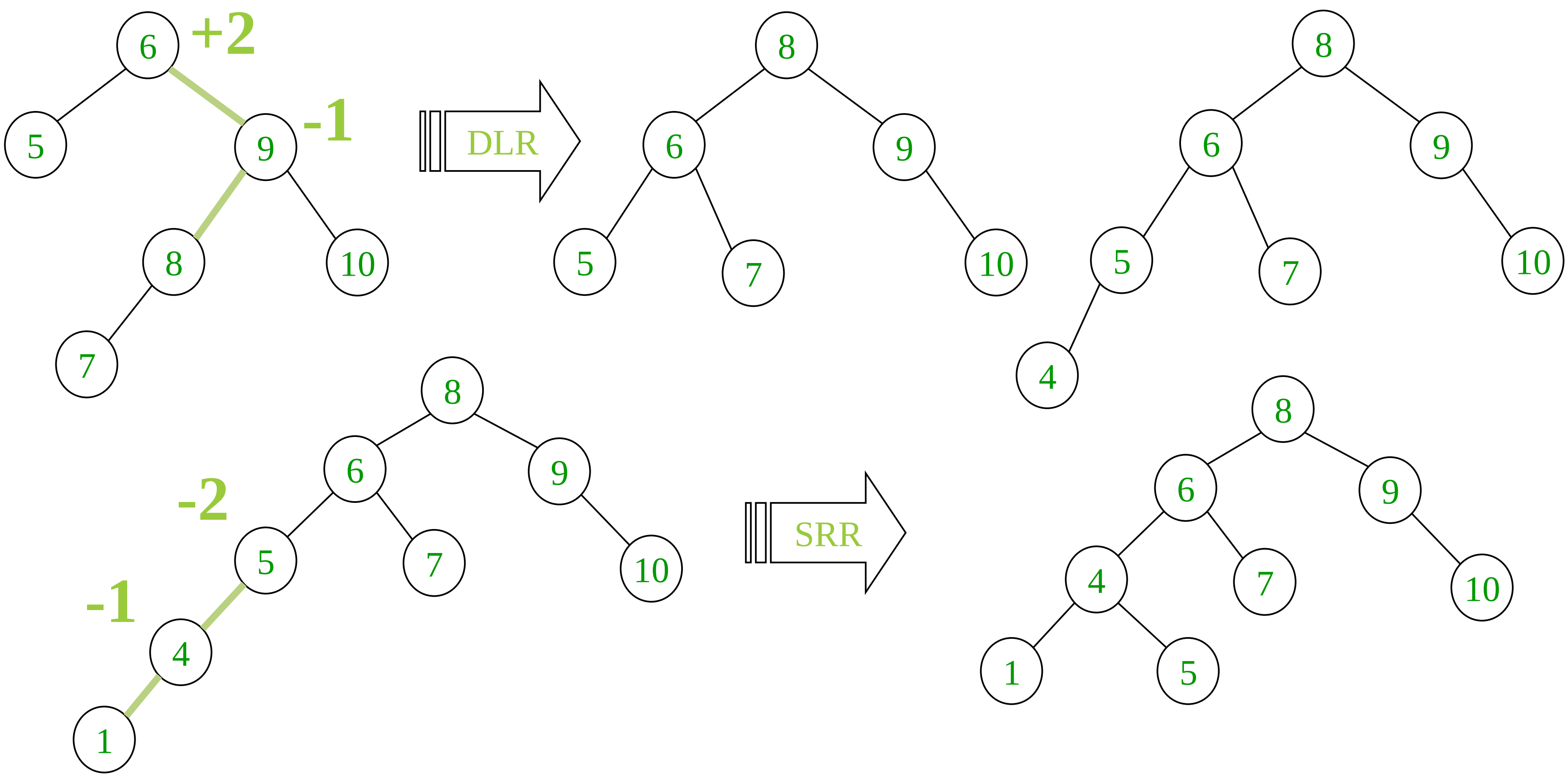
Example (1/4)

- 5 6 9 8 10 7 4 1 3 2



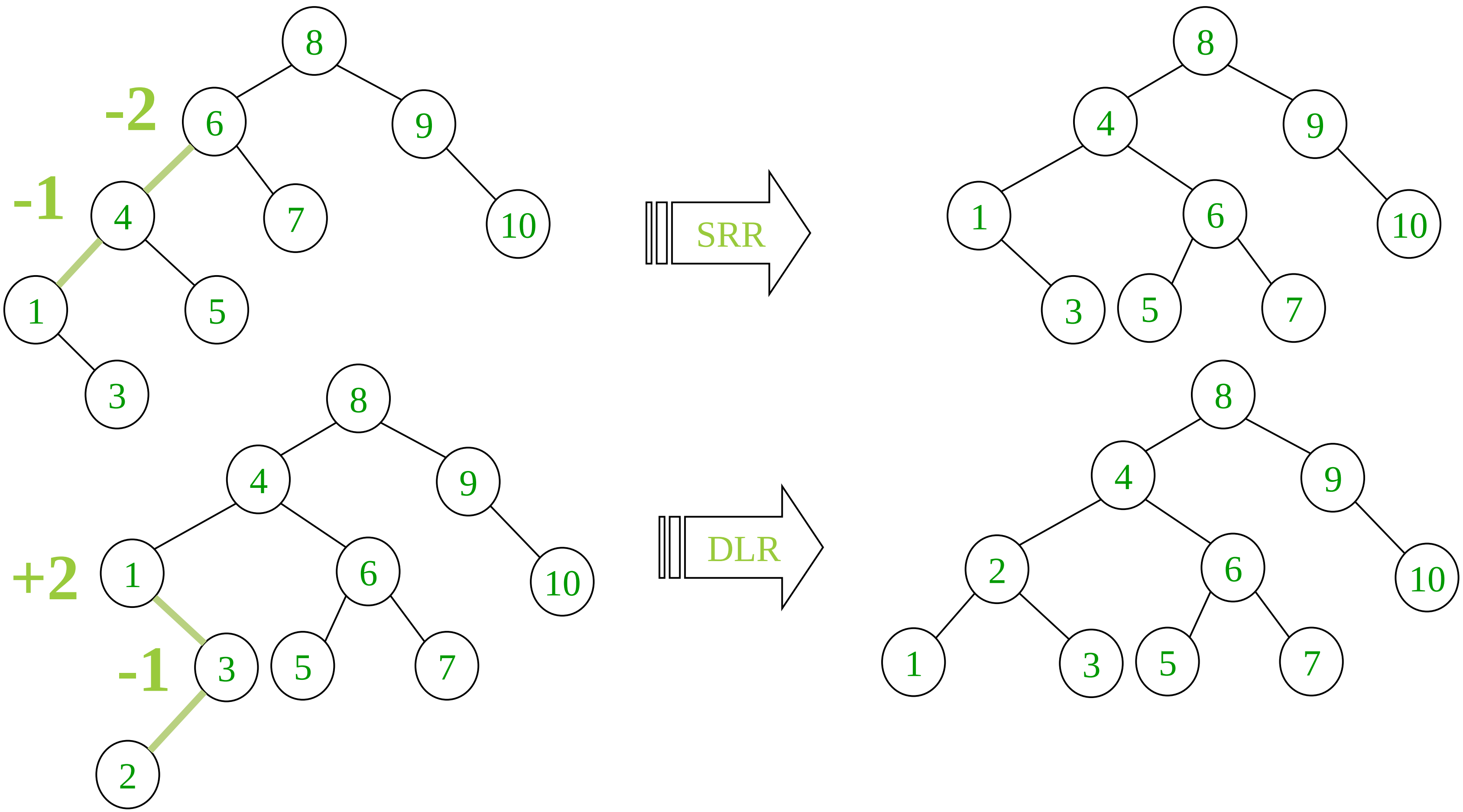
Example (2/4)

- 5 6 9 8 10 7 4 1 3 2

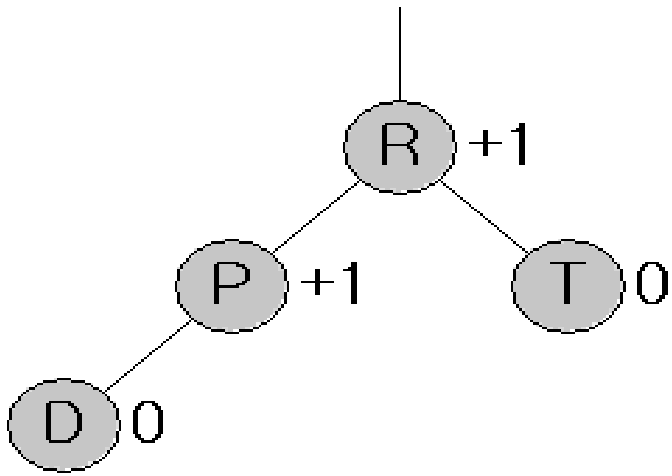


Example (3/4)

- 5 6 9 8 10 7 4 1 3 2

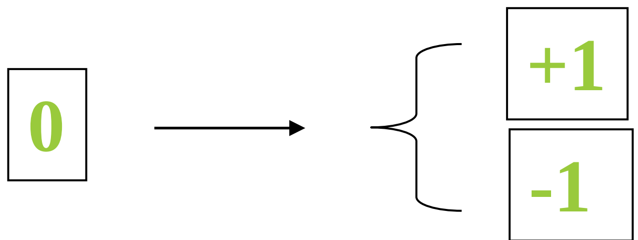


Example (4/4)

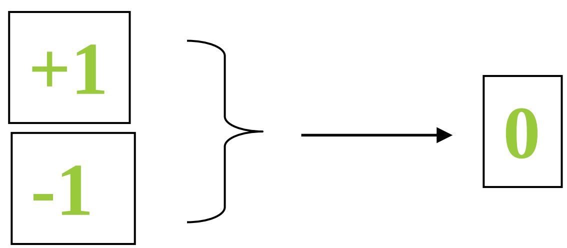


Insertion Strategy

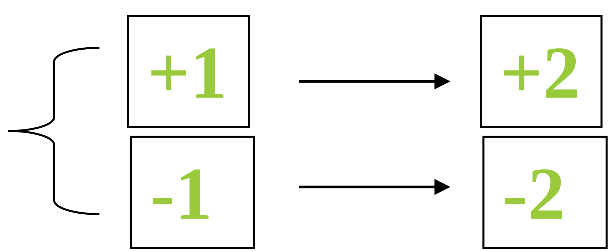
- Insert at leaf $\boxed{0}$
- Keep going upward and update code from leaf to parent if



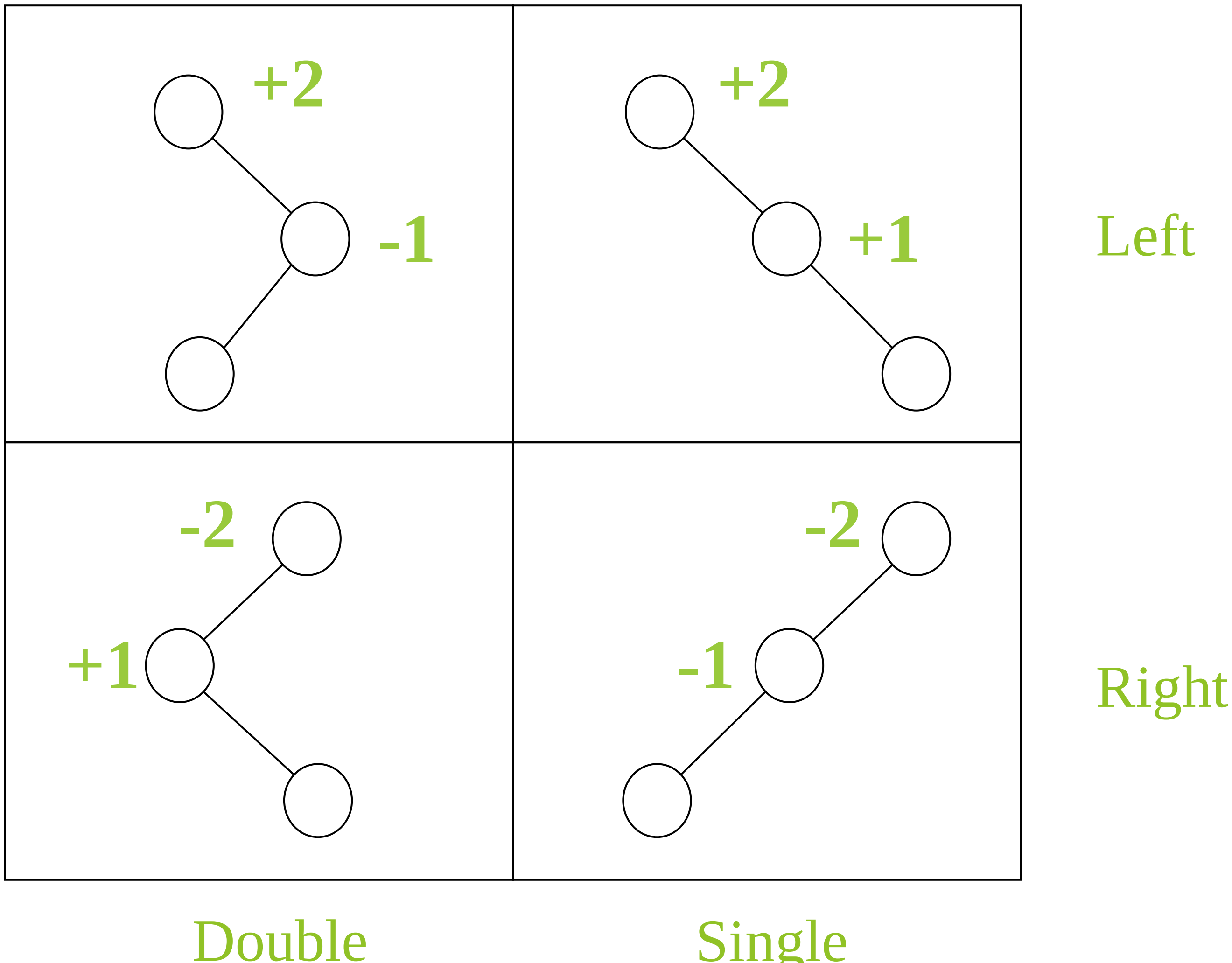
Stop when



Rotate when

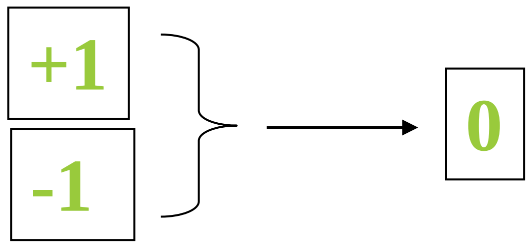


Insertion Strategy

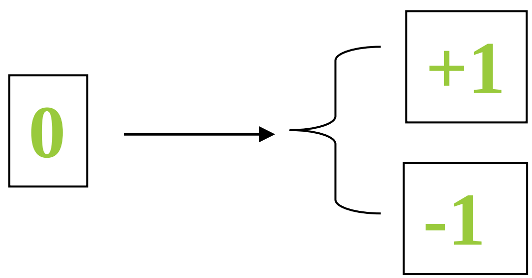


Deletion Strategy

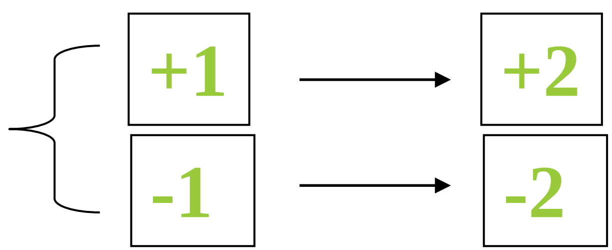
- Delete the node as in BST deletion
- Keep going upward and update code from leaf to parent if



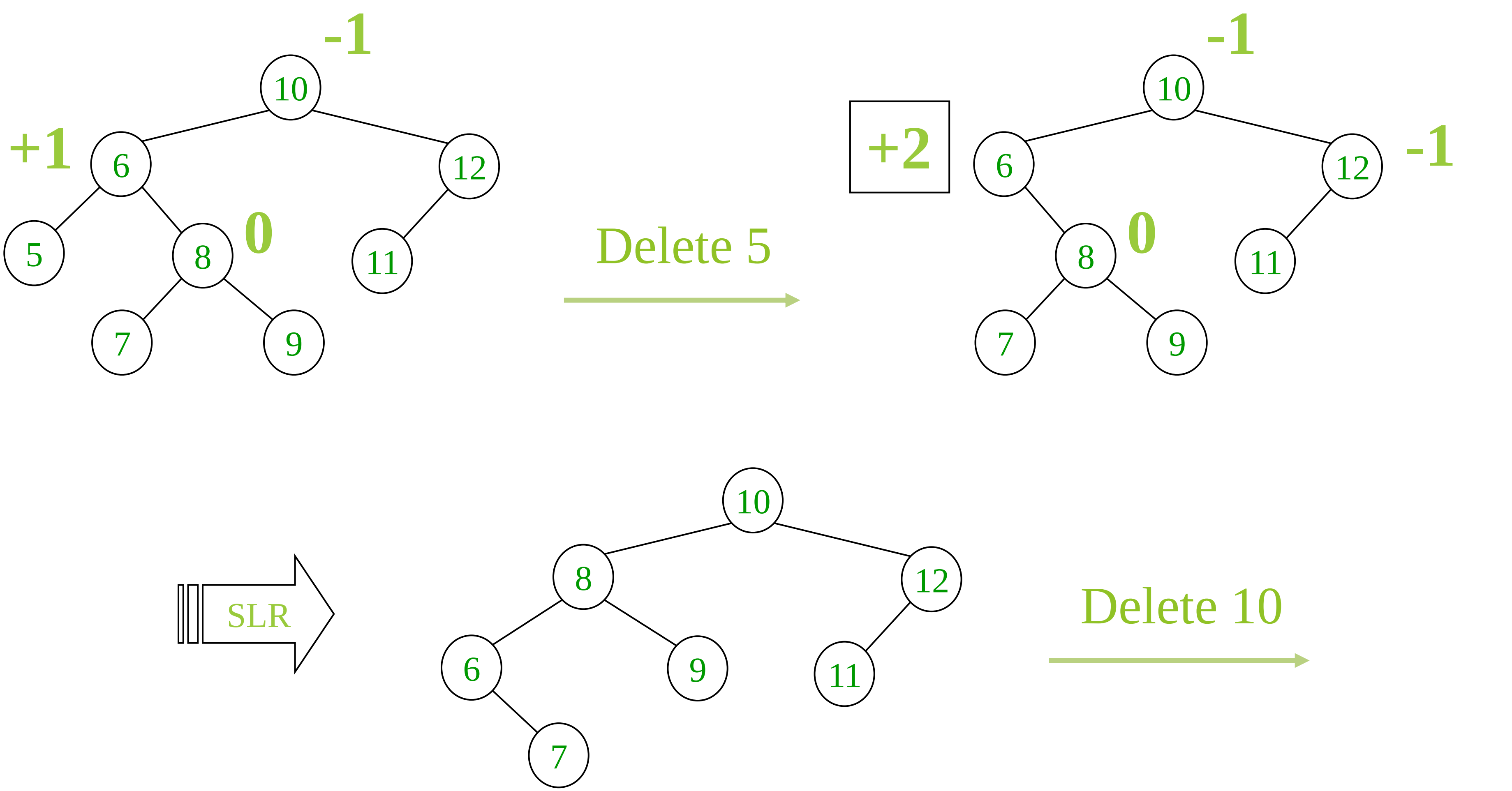
Stop when



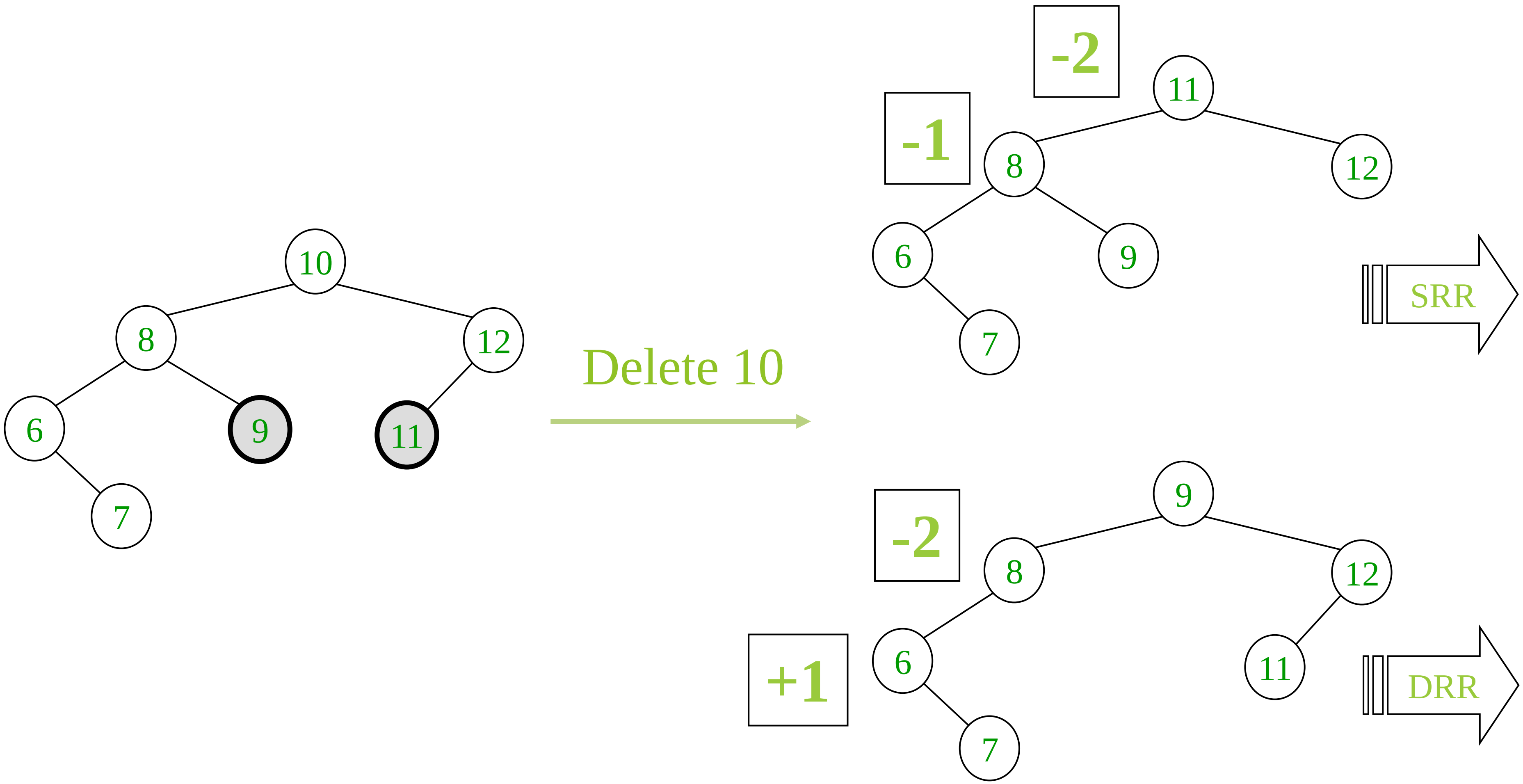
Rotate when



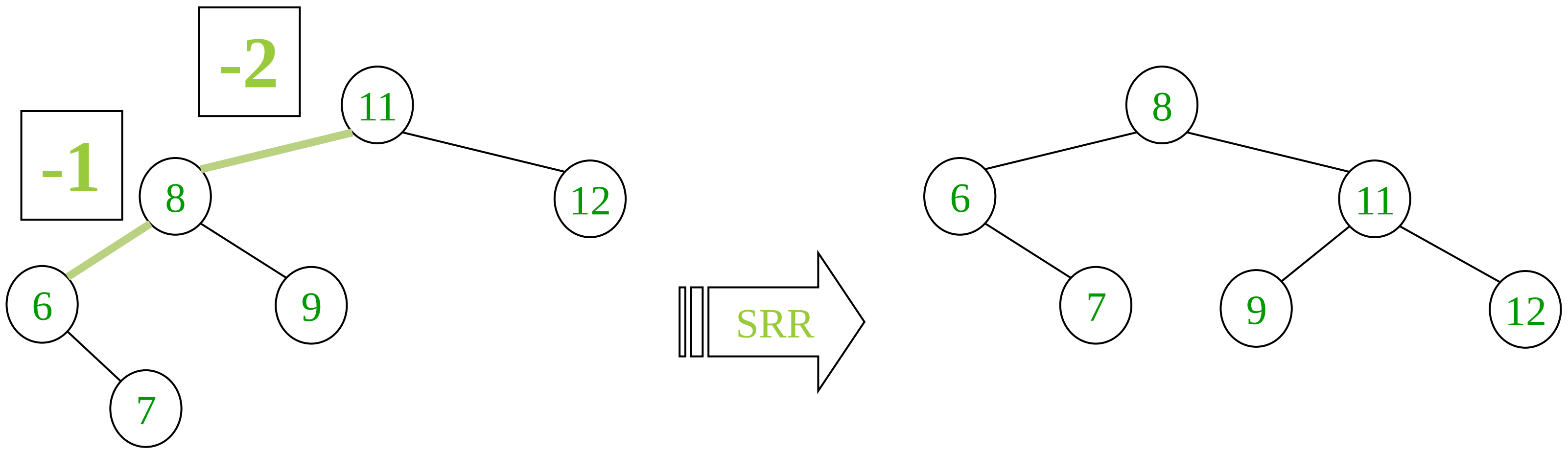
Example (1/3)

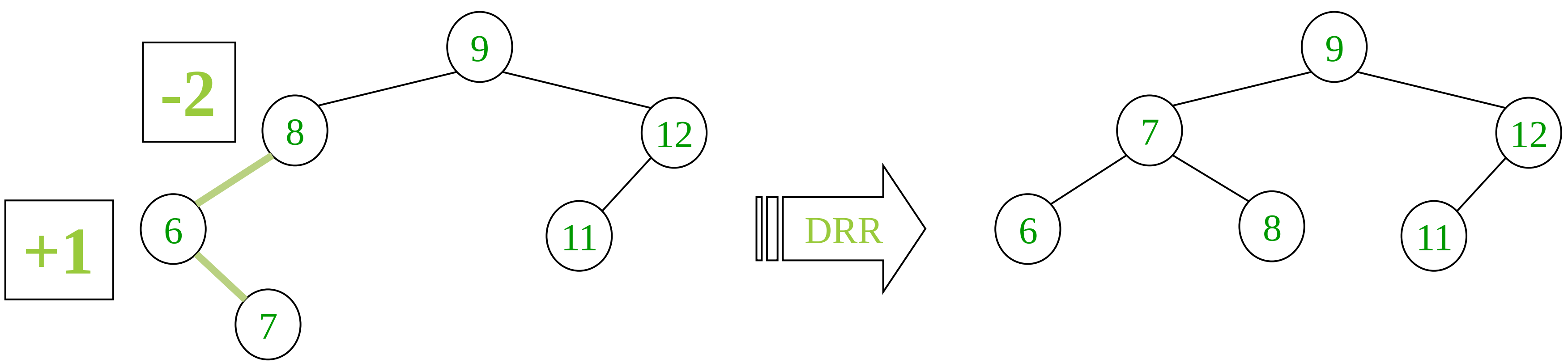

-1
10
+1
6
12
5
8
0
11
7
9
Delete 5
+2
-1
10
6
12
-1
8
0
11
7
9
SLR
10
8
12
6
9
11
7
Delete 10

Example (2/3)



Example (3/3)





Thanks to contributors

Mr. Pham Van Nguyen (2022)

Mr. Phuoc-Nguyen Bui (2022)

Dr. Thien-Binh Dang (2017 – 2022)

Prof. Hyunseung Choo (2001 – 2022)