[SWE2015-41] Introduction to Data Structures (자료구조개론)

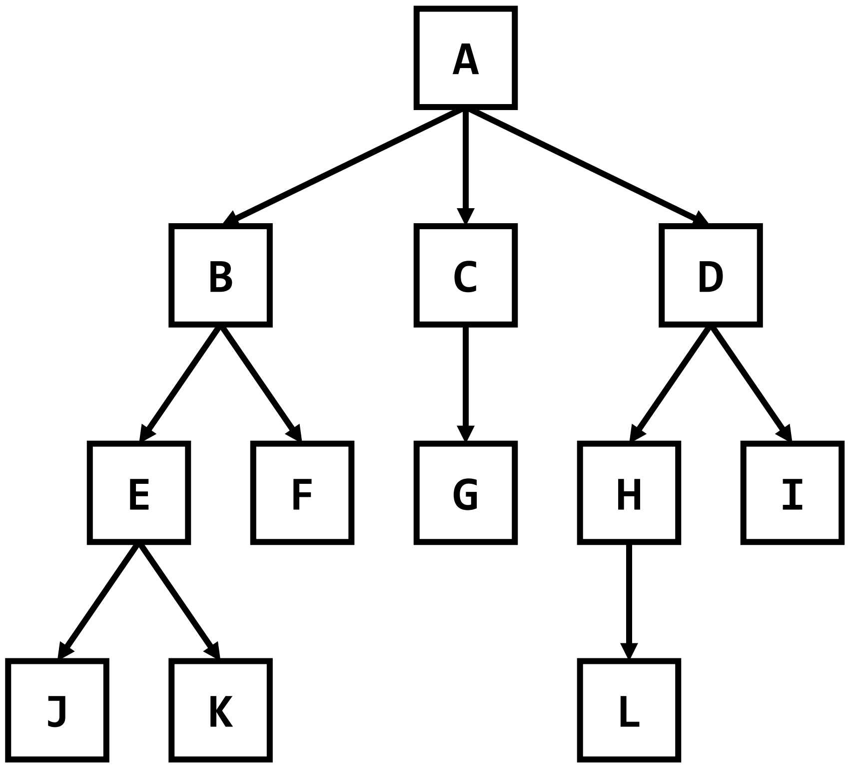
AVL Trees

Department of Computer Science and Engineering

Instructor: Hankook Lee (이한국)

(Recap) What is Tree?

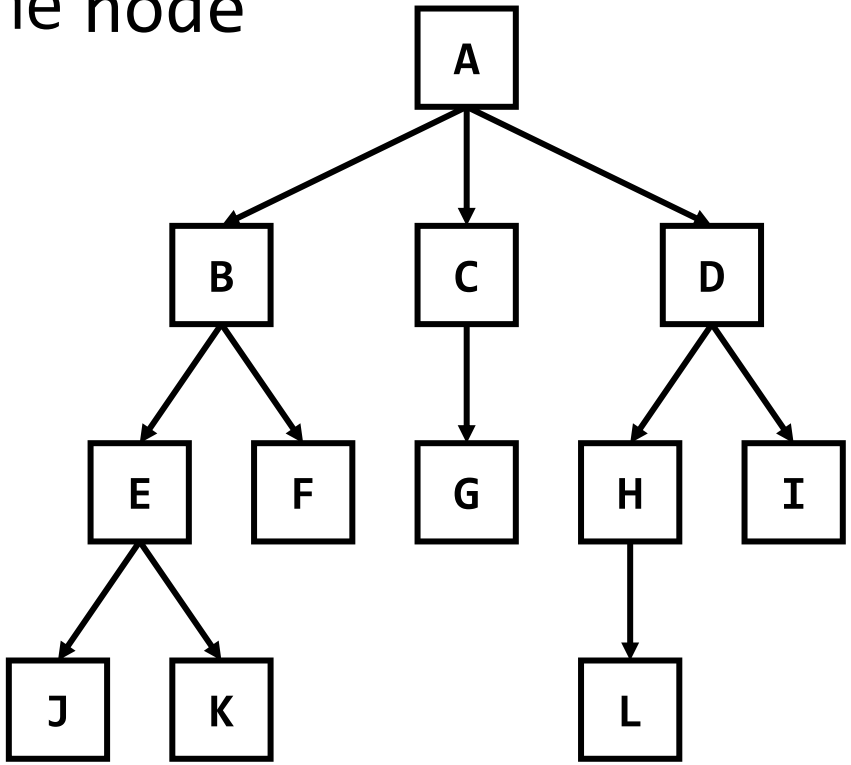
- Tree is a **hierarchical** structure with a set of connected nodes
  - Each node is composed with a parent-children relationship
  - There is no cycle (or loop) in the tree



(Recap) Terminology (Basic)

- **Node** represents an object
- **Edge** represents a connection between two nodes
  - If X → Y, say X is the **parent** of Y and Y is a **child** of X
- **Degree** of a node is the number of children of the node
  - It is equal to the number of outgoing edges

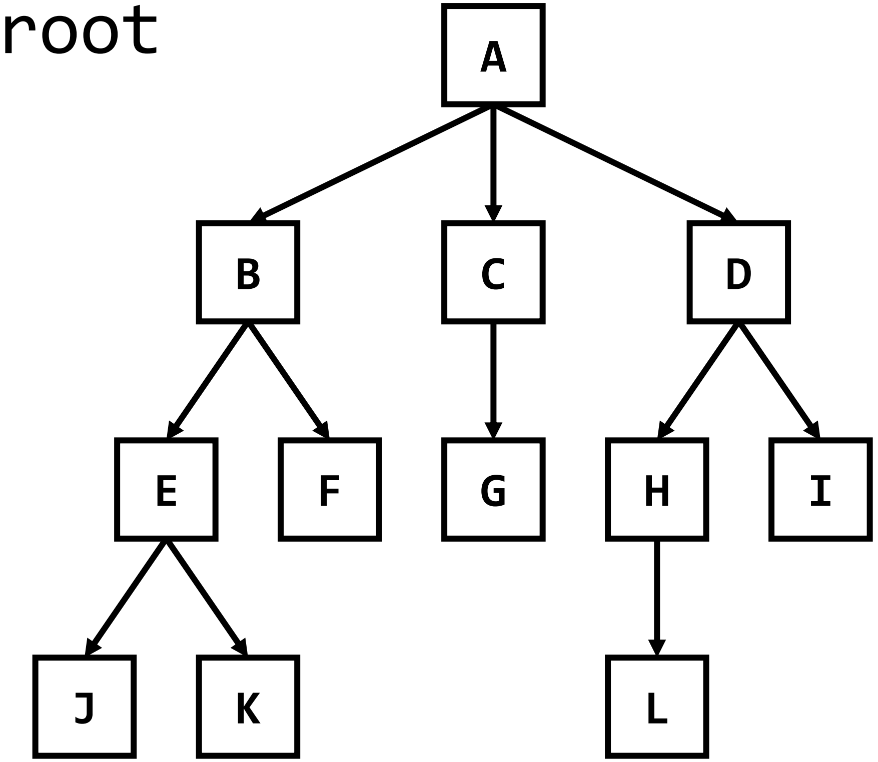
- Examples
  - B is the parent of E and F
  - H is a child of D
  - `degree(D) = 2`
  - `degree(J) = 0`



(Recap) Terminology (Tree-Level)

- **Root** is the top node in a tree
- **Internal** (or non-terminal) node: degree ≥ 1
- **Leaf** (or terminal) node: degree = 0
- **Height** is # of nodes on the longest path from root

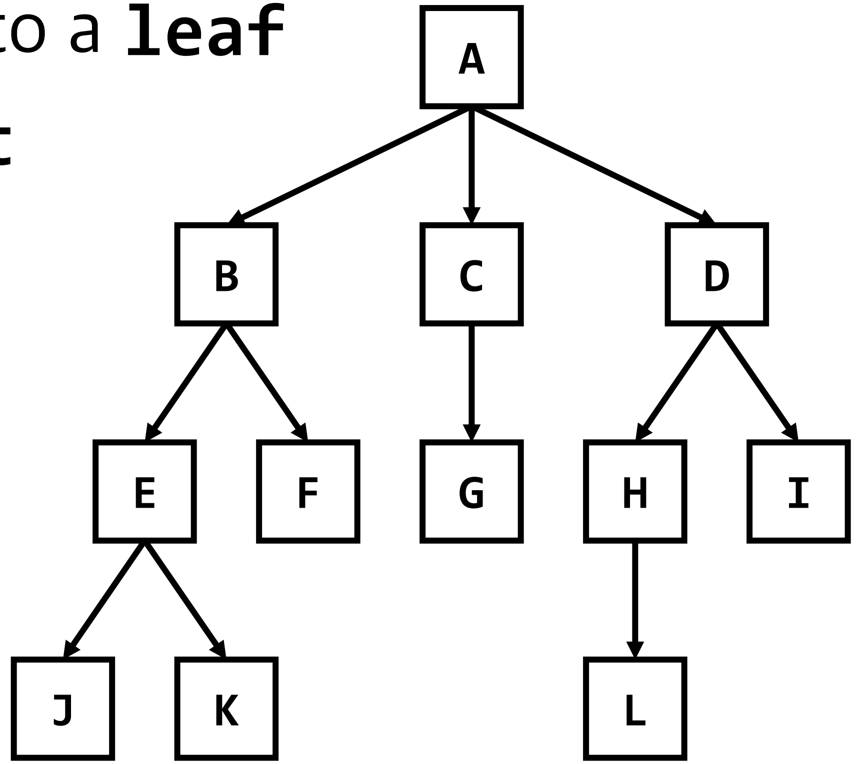
- Examples
  - A is the root of the tree
  - Internal nodes are A, B, C, D, E, H
  - Leaf nodes are F, G, I, J, K, L
  - The height of the tree is 4



(Recap) Terminology (Node-Level)

For a **node X**,

- **`Level`** or **`depth`** is the distance between **`root`** and **`X`**
- **`Ancestor`** is a predecessor on the path from **`root`** to **`X`**
- **`Descendant`** is a successor on any path from **`X`** to a **`leaf`**
- **`Sibling`** is another **`node`** with the same **`parent`**



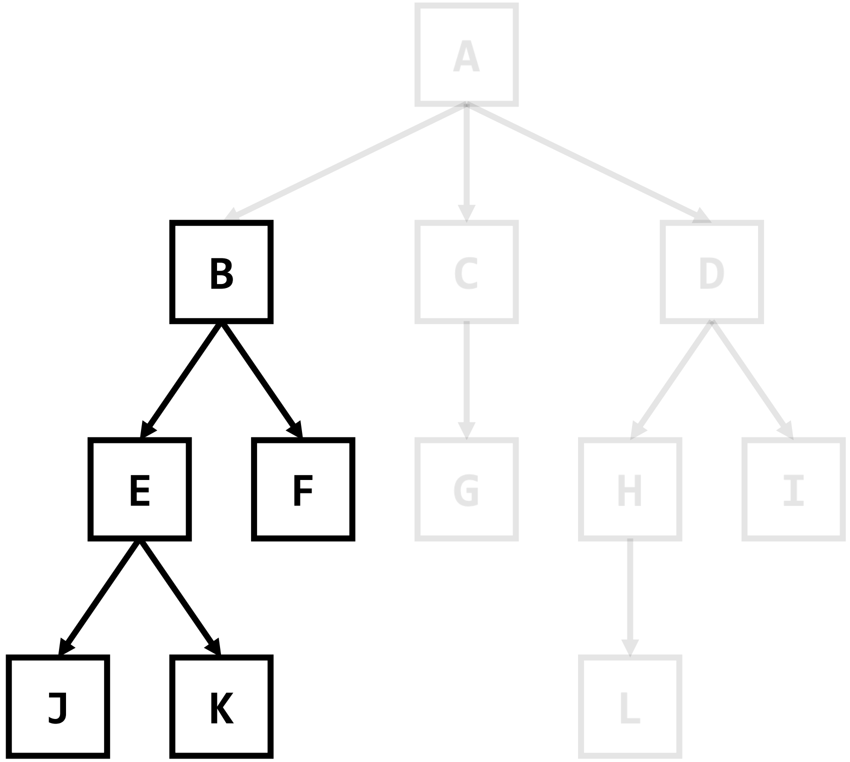
- Examples
  - **A**'s level/depth is 0
  - **F**'s level/depth is 2
  - **A** and **B** are **ancestors** of **E**
  - **E**, **F**, **J**, and **K** are **descendants** of **B**
  - **B** and **D** are **siblings** of **C**

(Recap) Terminology (Node-Level)

Subtree rooted at a **node X**

- Any **node** can be treated as the **root node** of its own **subtree**
- The **subtree** includes **X** and all **descendant**s of **X**

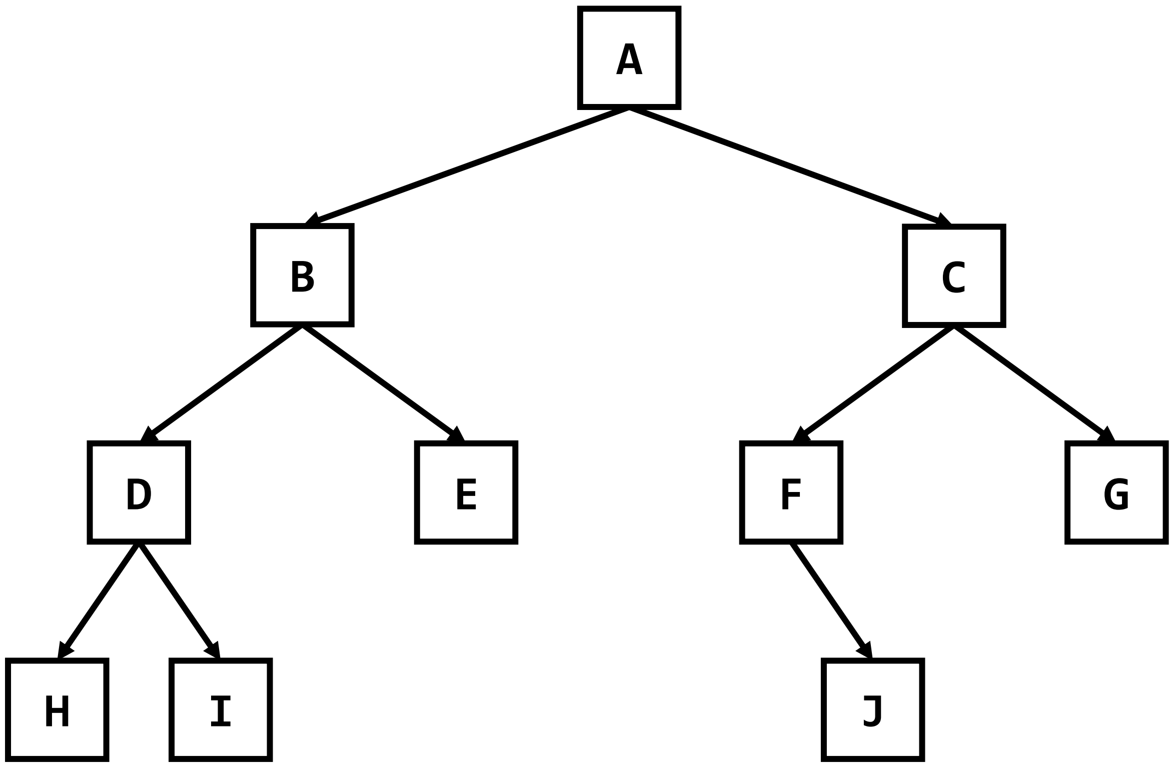
Subtree rooted at **node B**



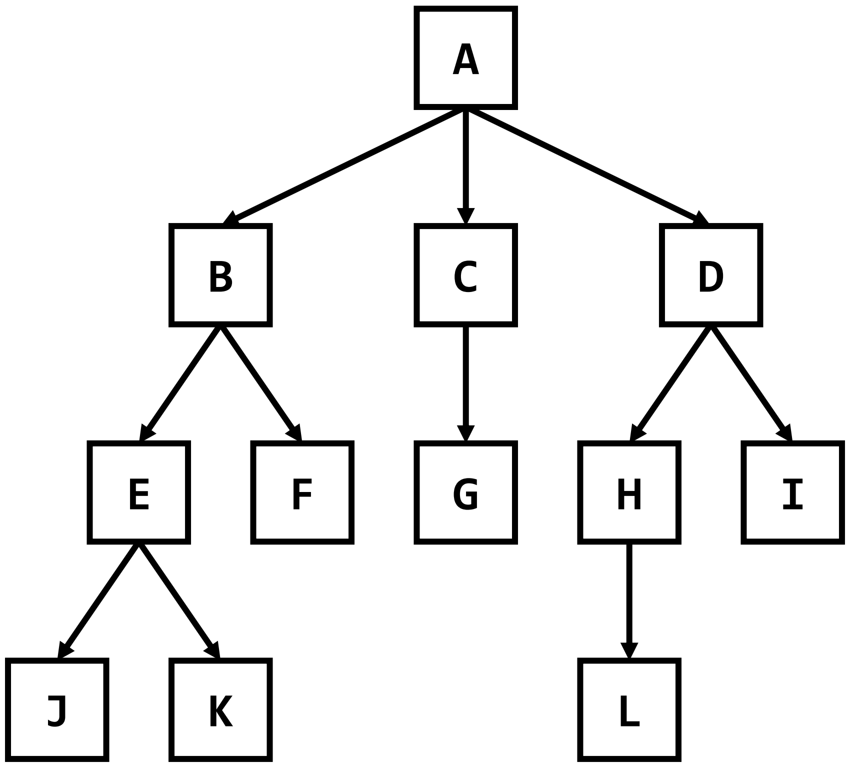
(Recap) Binary Trees

- **Binary Tree** is a tree in which each node has at most two children
  - `degree(X)` ≤ 2 for any `node X` in a binary tree

Binary

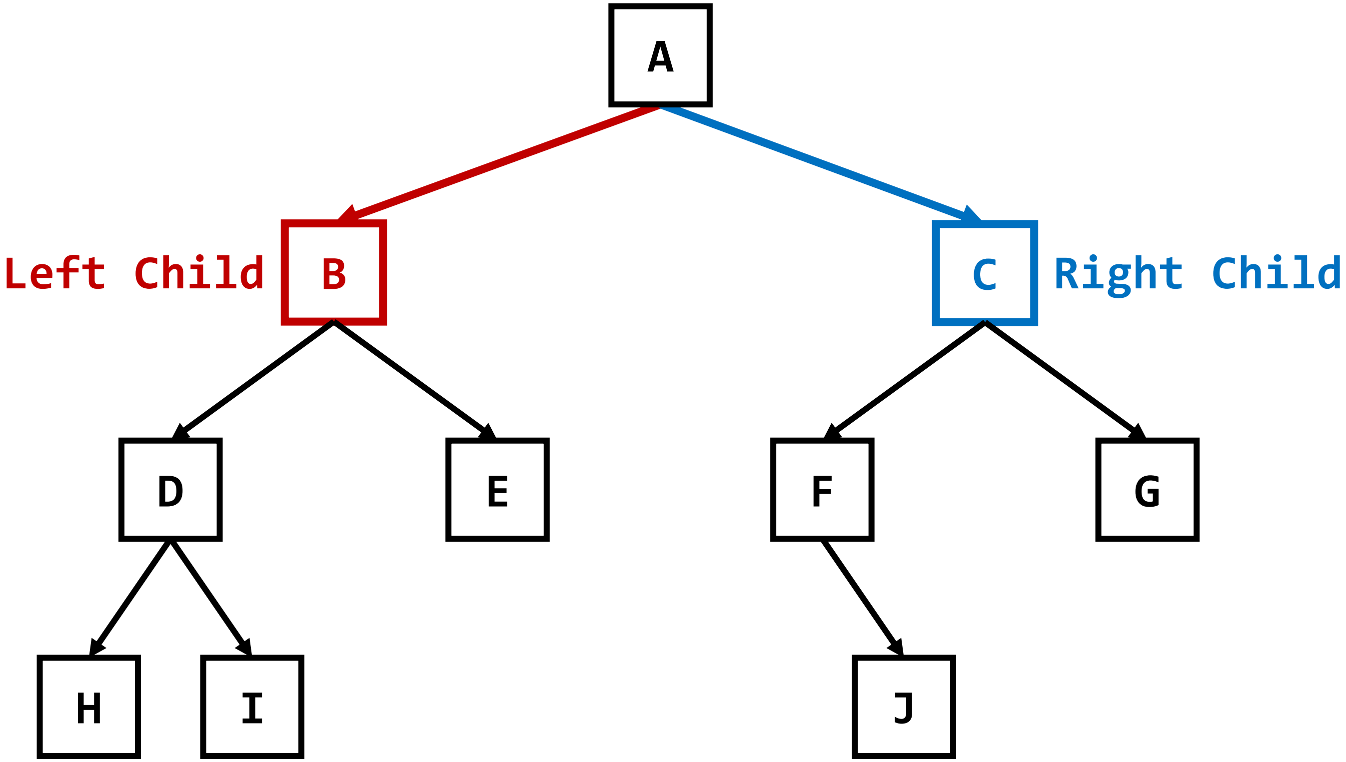


NOT Binary



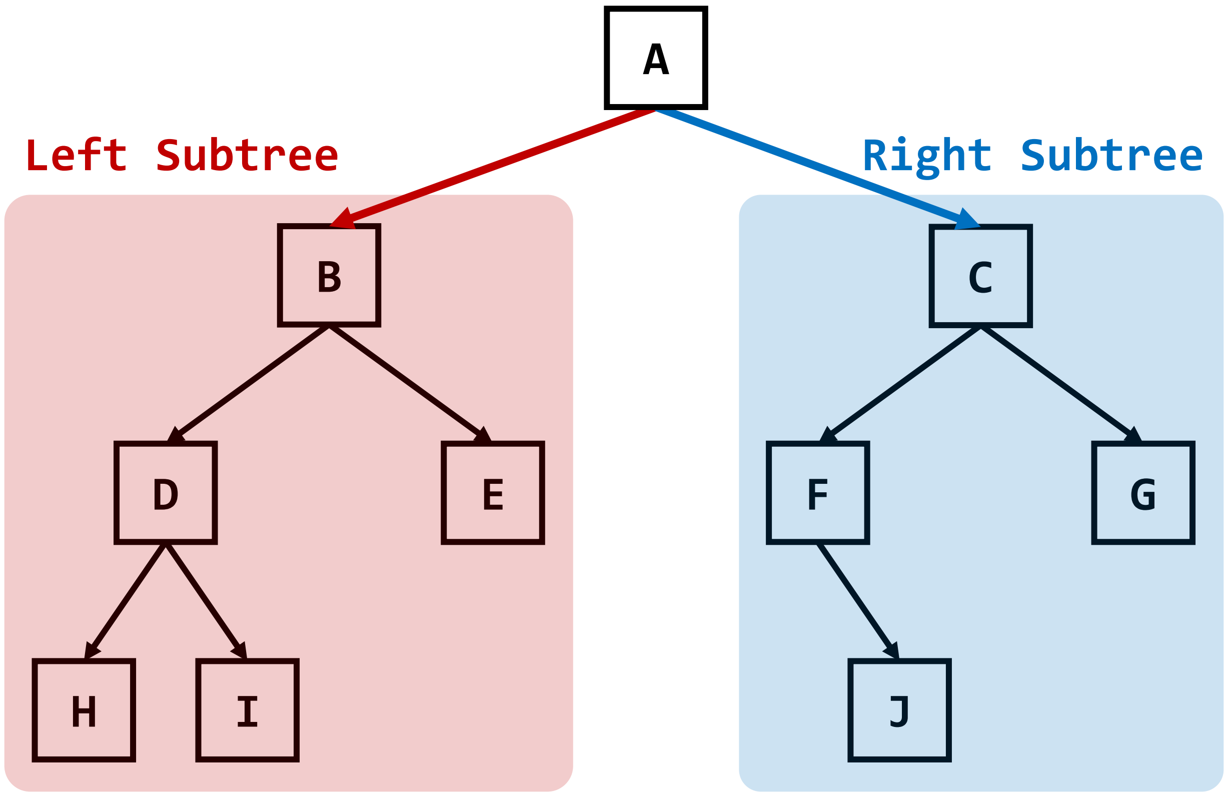
(Recap) Binary Trees

- **Binary Tree** is a tree in which each node has at most two children
  - `degree(X)` ≤ 2 for any `node X` in a binary tree
  - Each node has **`left`** & **`right`** children



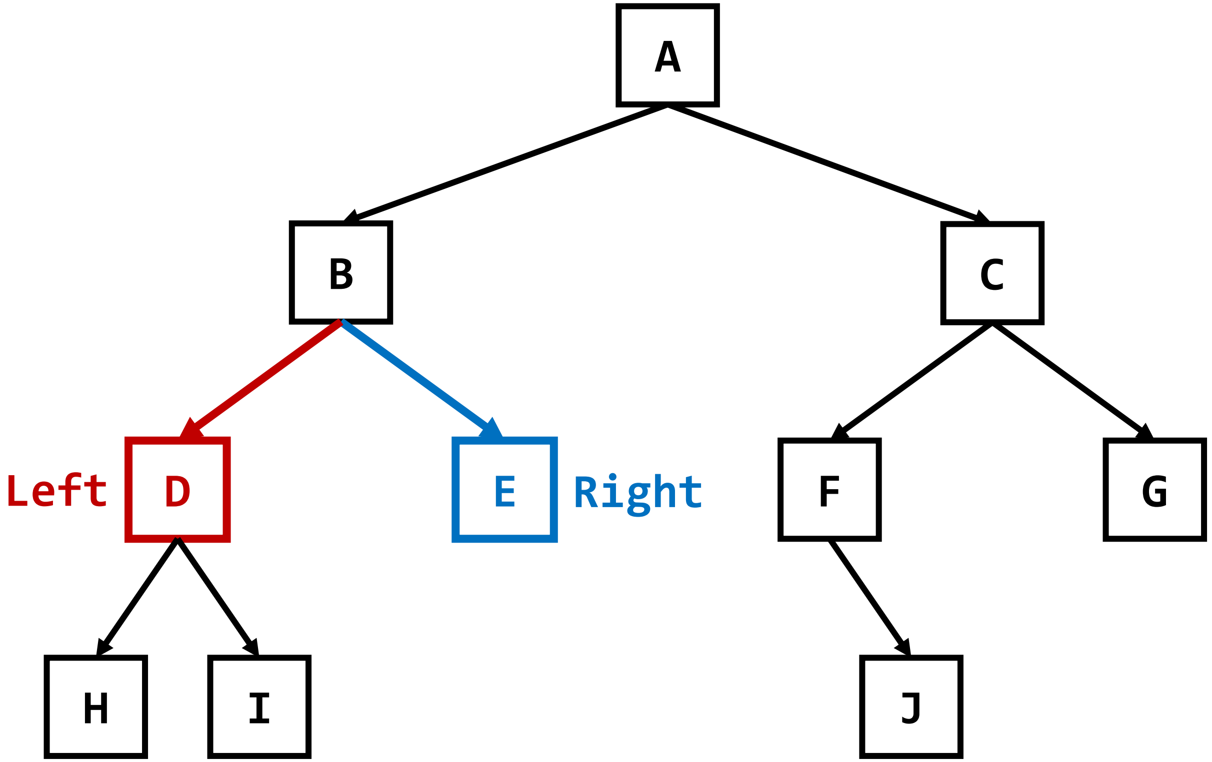
(Recap) Binary Trees

- **Binary Tree** is a tree in which each node has at most two children
  - `degree(X)` ≤ 2 for any `node X` in a binary tree
  - Each node has `left` & `right` children



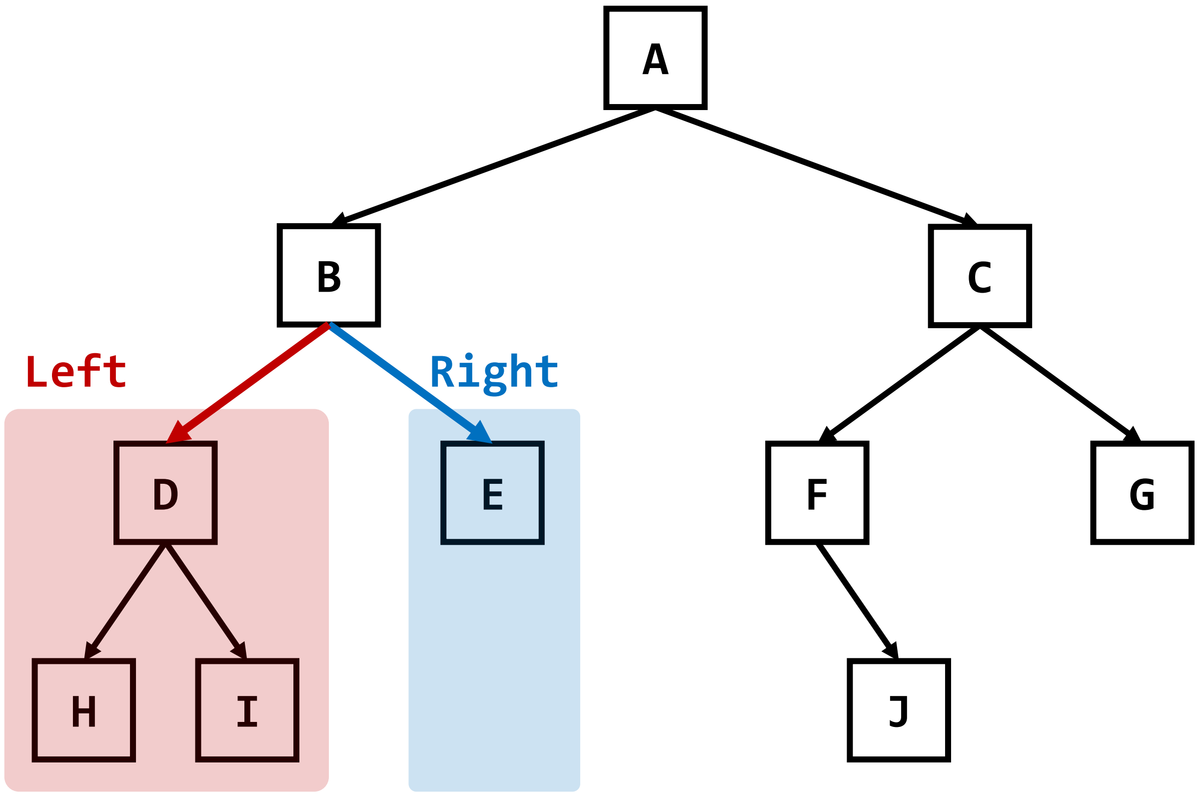
(Recap) Binary Trees

- **Binary Tree** is a tree in which each node has at most two children
  - `degree(X)` ≤ 2 for any `node X` in a binary tree
  - Each node has `left` & `right` children



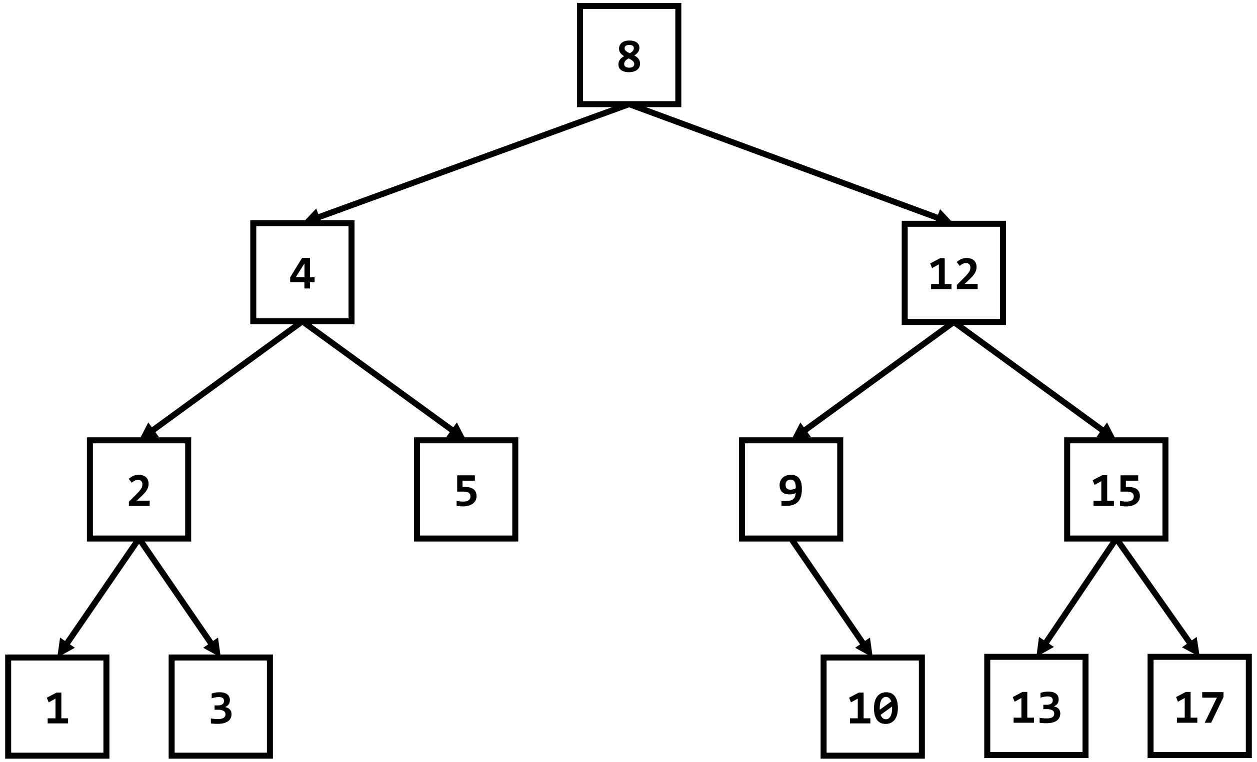
(Recap) Binary Trees

- **Binary Tree** is a tree in which each node has at most two children
  - `degree(X)` ≤ 2 for any `node X` in a binary tree
  - Each node has **`left`** & **`right`** children



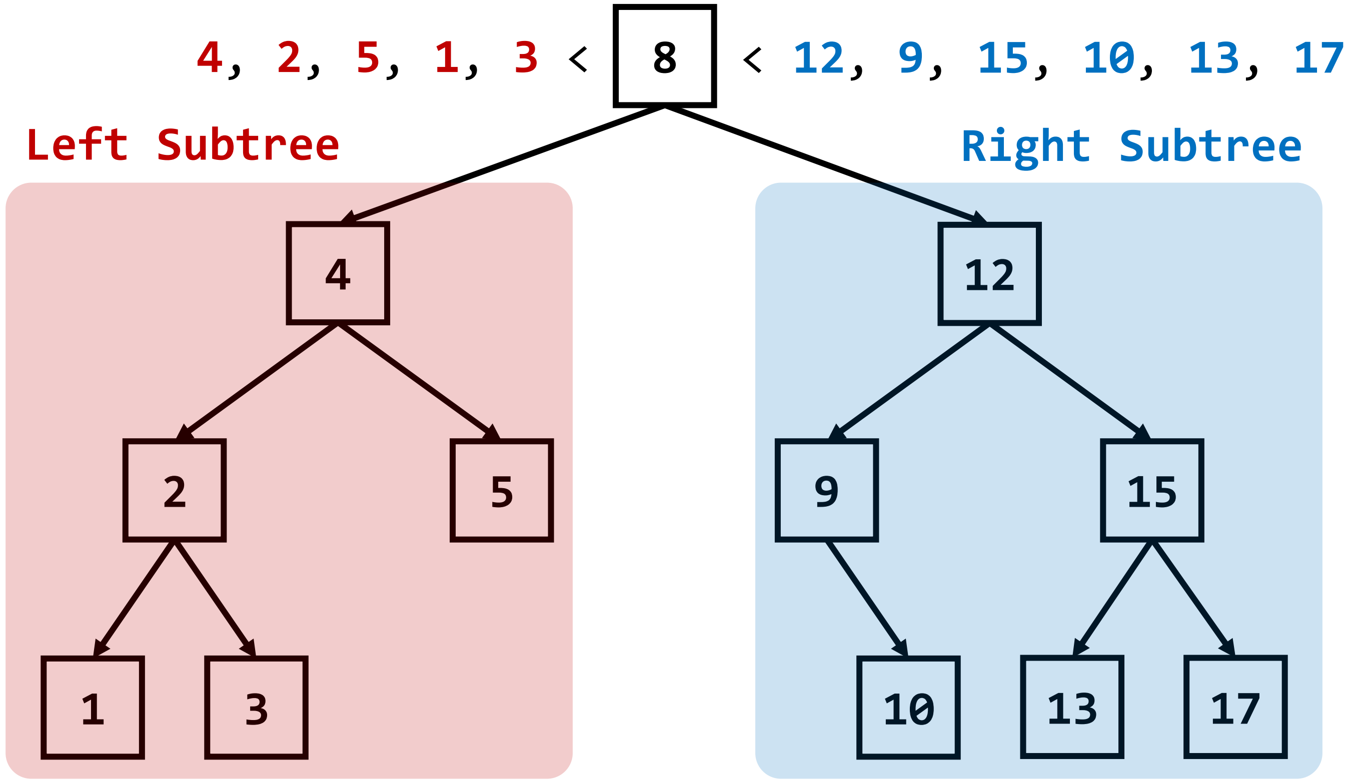
(Recap) Binary Search Trees (BSTs)

- **Binary Search Tree (BST)** satisfies the following conditions:
  1. Any two nodes **A** and **B** are comparable: `A < B`, `A > B`, or `A == B`
     - E.g., you can compare numbers numerically or strings in the alphabetical/dictionary order
     - Such a comparable value of a node is called **KEY** value



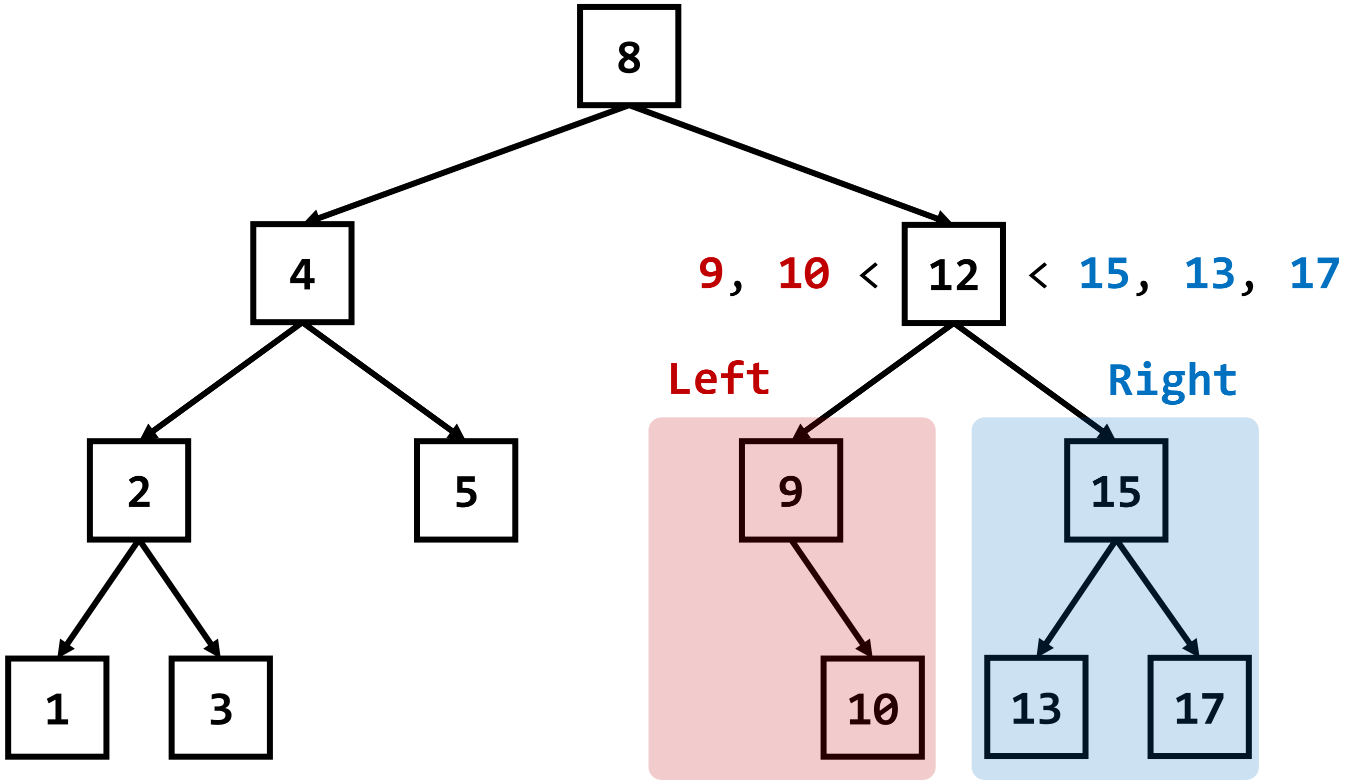
(Recap) Binary Search Trees (BSTs)

- **Binary Search Tree (BST)** satisfies the following conditions:
  2. For any node **X**, all nodes in its **`left subtree`** are less than **X**
  3. For any node **X**, all nodes in its **`right subtree`** are greater than **X**



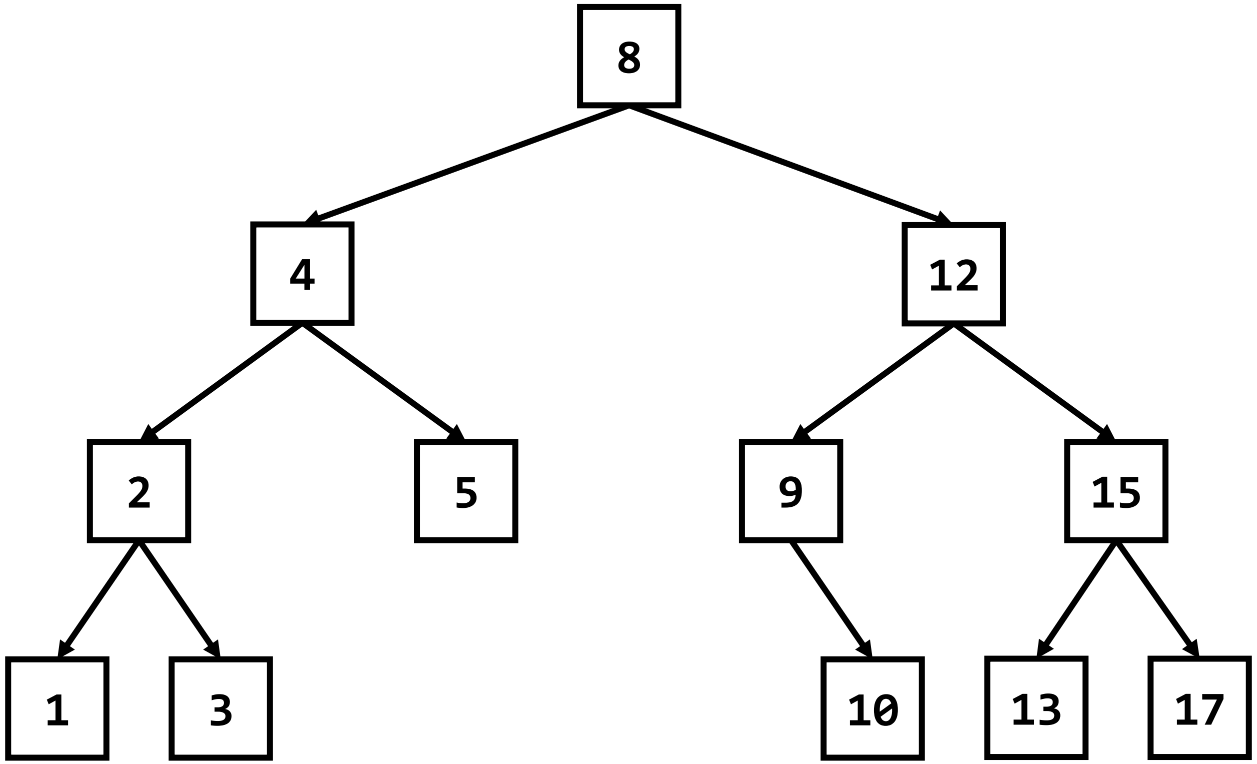
(Recap) Binary Search Trees (BSTs)

- **Binary Search Tree (BST)** satisfies the following conditions:
  2. For any node **X**, all nodes in its `left subtree` are less than **X**
  3. For any node **X**, all nodes in its `right subtree` are greater than **X**



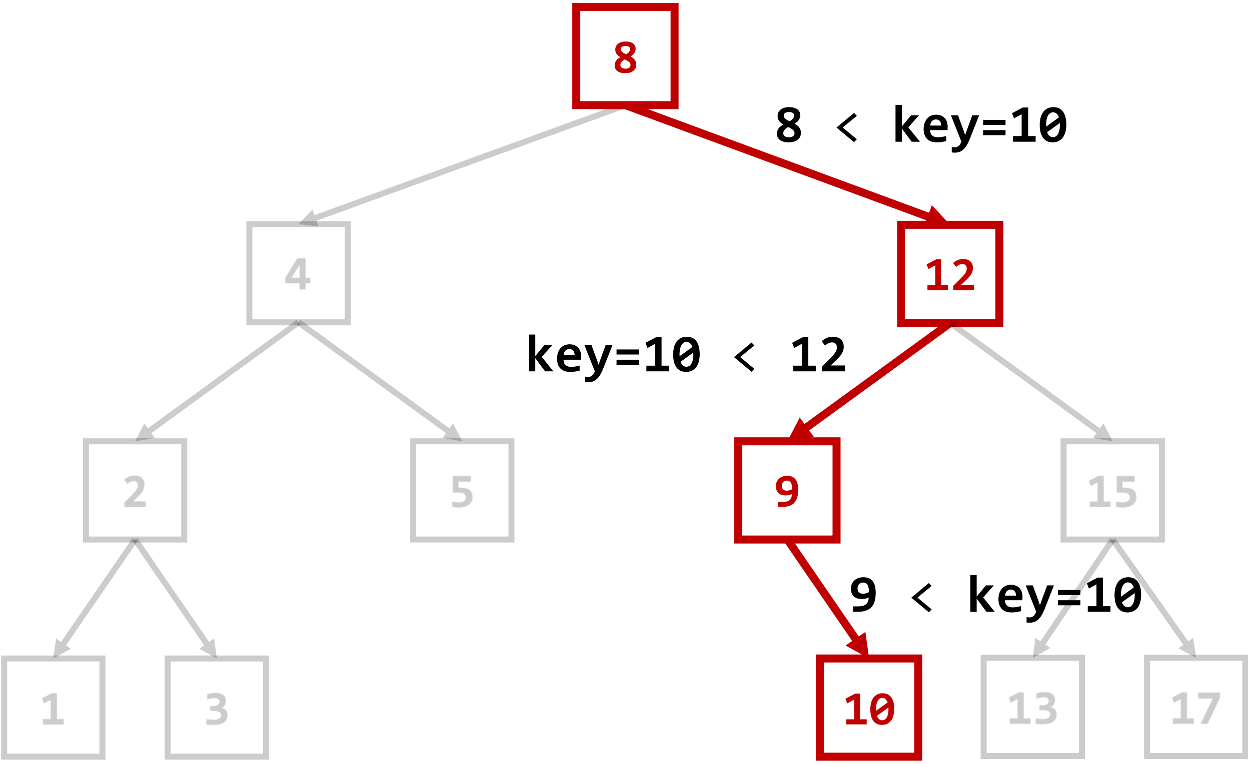
(Recap) BST Operations

- **Validity** - check whether a binary tree is a binary search tree?
- **Search** - find the node of the target **KEY**
- **Insertion/Deletion** - insert/delete the node using **KEY**



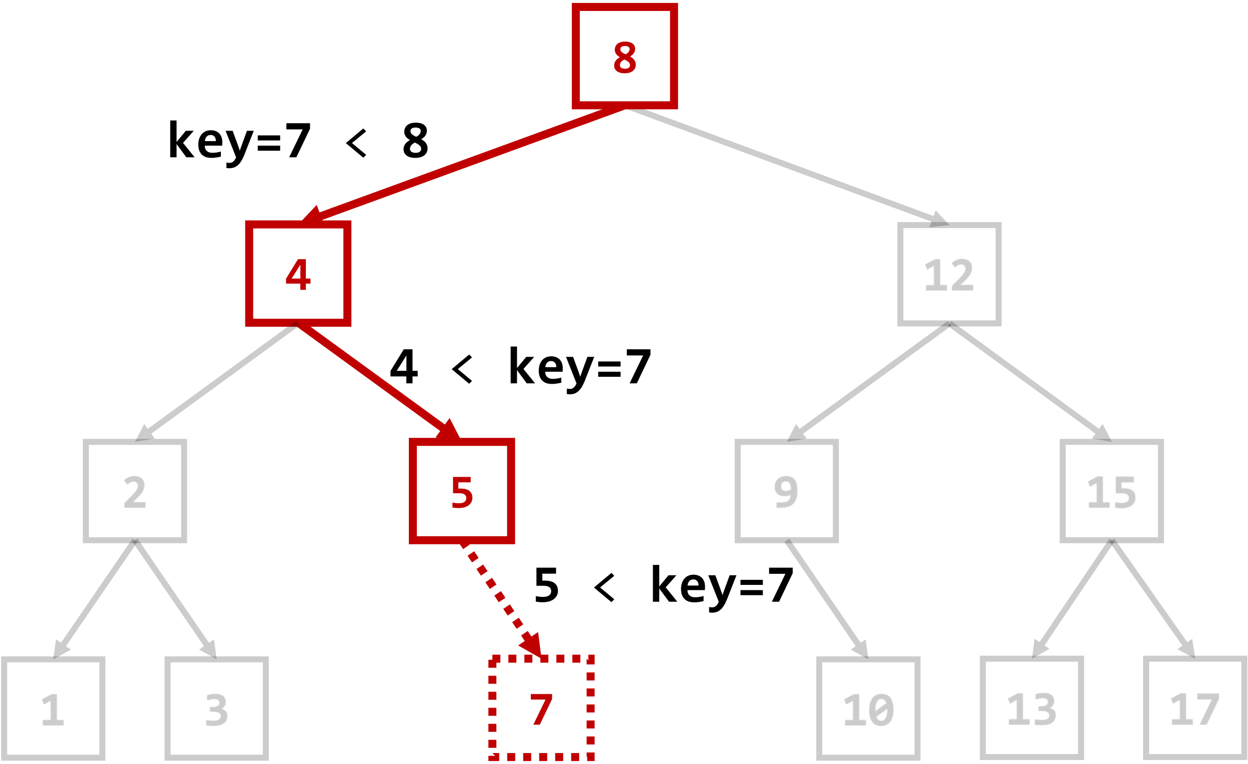
(Recap) BST Operations

- **Validity** - check whether a binary tree is a binary search tree?
- **Search** - find the node of the target **KEY**
- **Insertion/Deletion** - insert/delete the node using **KEY**



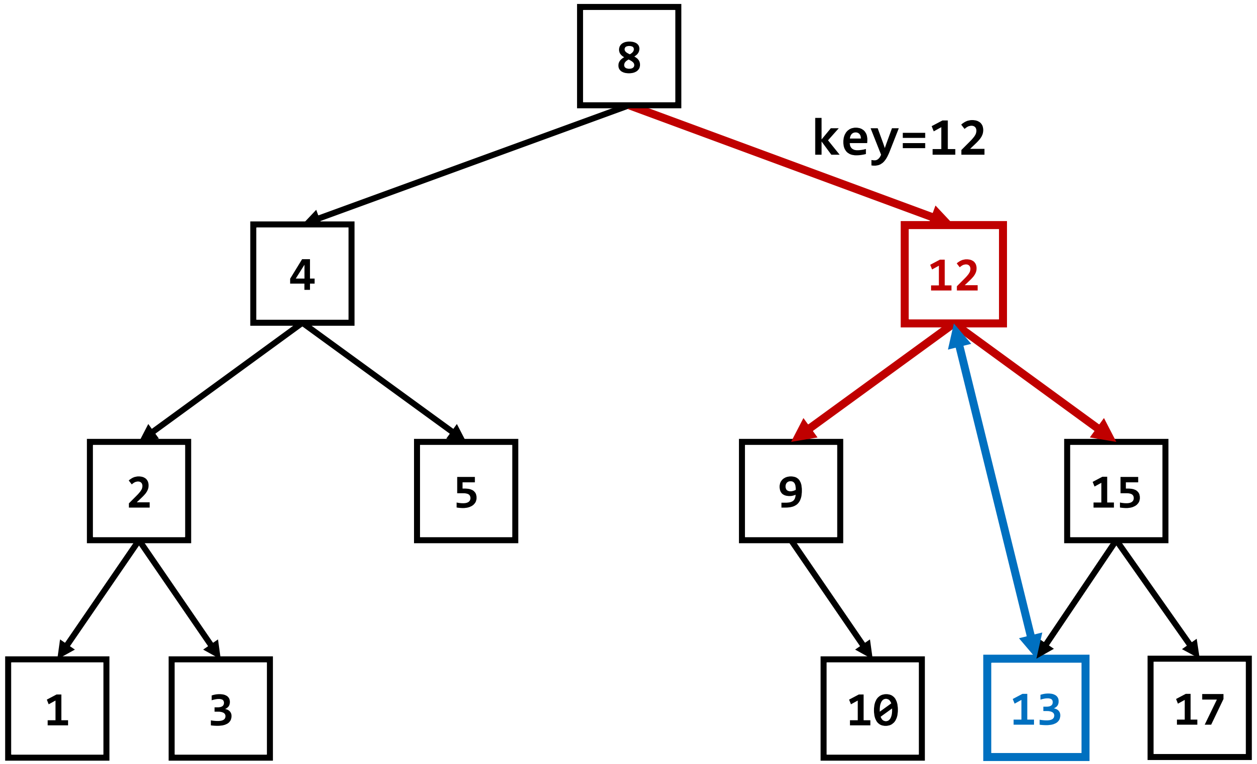
(Recap) BST Operations

- **Validity** - check whether a binary tree is a binary search tree?
- **Search** - find the node of the target **KEY**
- **Insertion**/**Deletion** - insert/delete the node using **KEY**



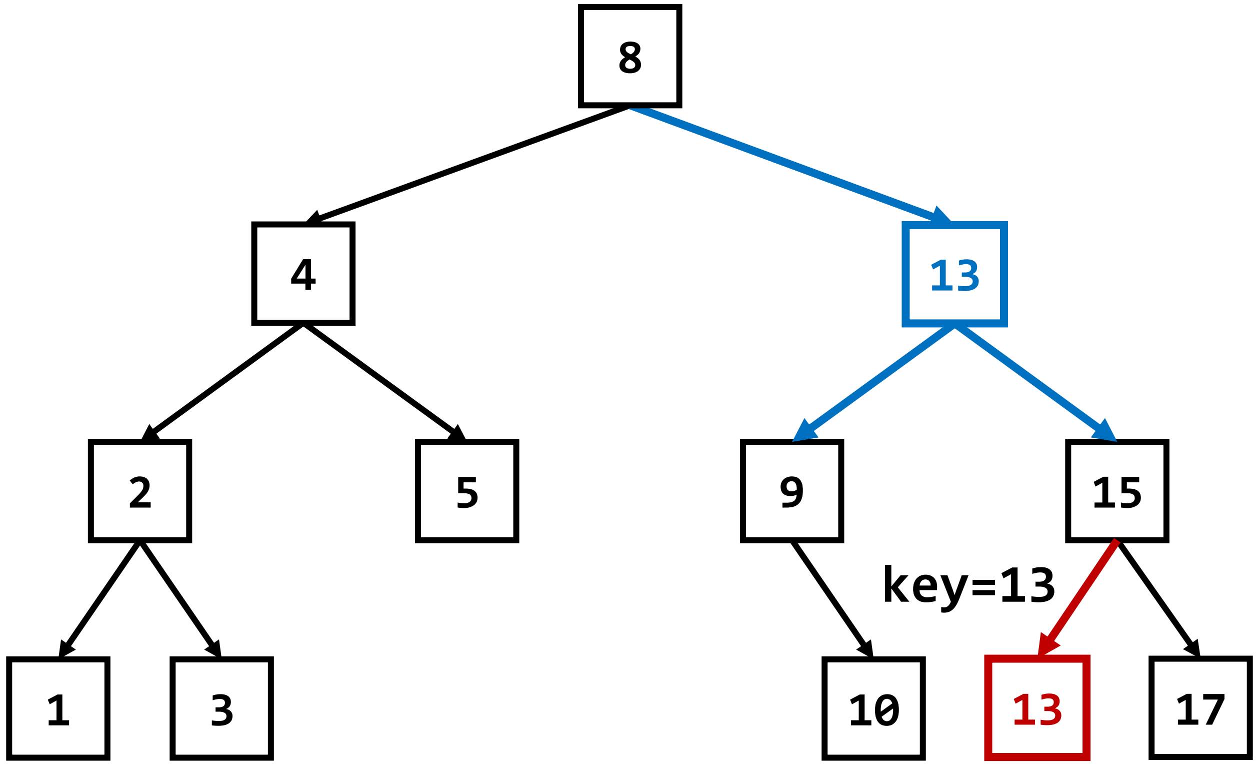
(Recap) BST Operations

- **Validity** - check whether a binary tree is a binary search tree?
- **Search** - find the node of the target **KEY**
- **Insertion**/**Deletion** - insert/delete the node using **KEY**



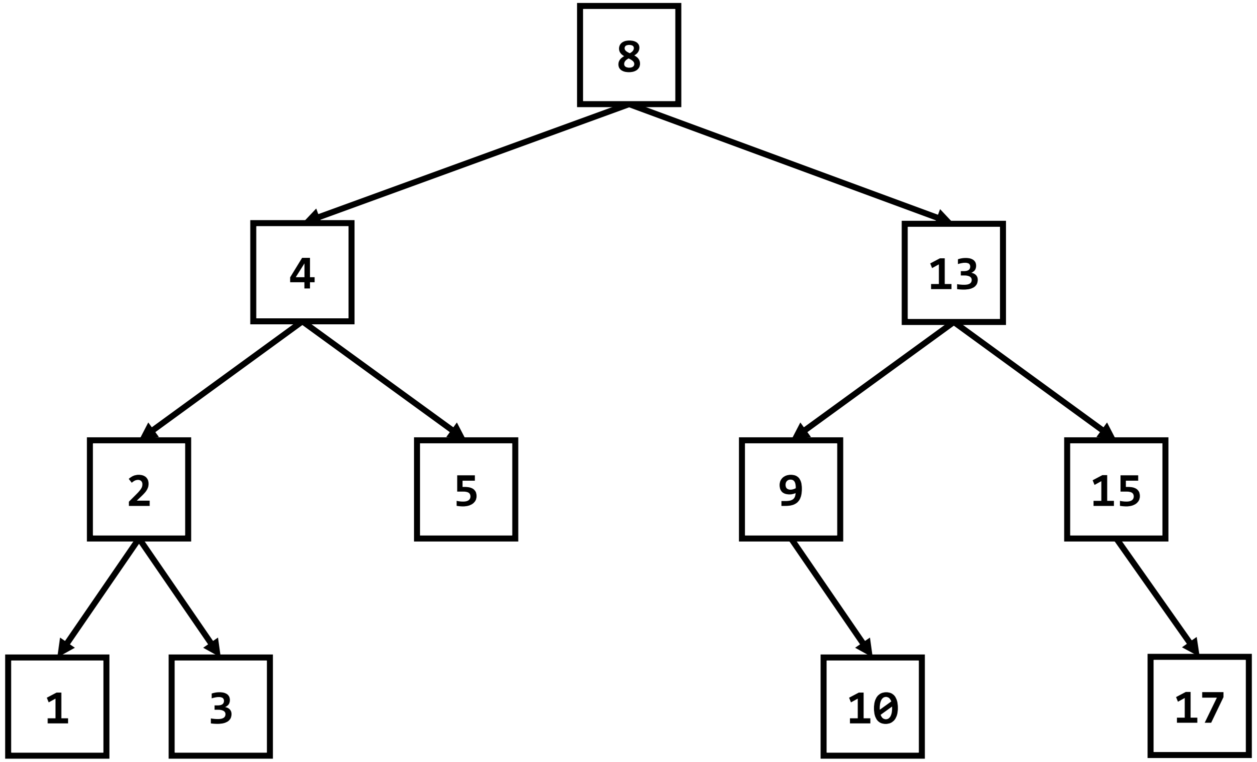
(Recap) BST Operations

- **Validity** - check whether a binary tree is a binary search tree?
- **Search** - find the node of the target **KEY**
- **Insertion/Deletion** - insert/delete the node using **KEY**



(Recap) BST Operations

- **Validity** - check whether a binary tree is a binary search tree?
- **Search** - find the node of the target **KEY**
- **Insertion**/**Deletion** - insert/delete the node using **KEY**

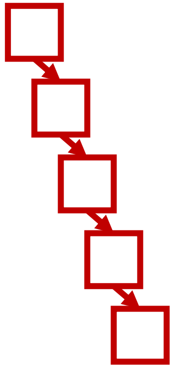


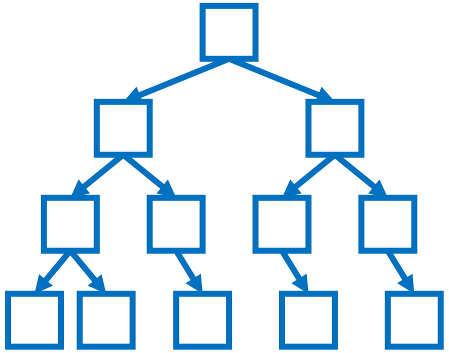
(Recap) BST Operations - Time Complexity

- The time complexities for search, insertion, and deletion are $O(H)$
  - H is the tree height
  - $\log_2 N \leq H \leq N$ where N is the number of nodes in a binary tree

Operation	Balanced Tree	Skewed Tree
Search	$O(\log N)$	$O(N)$
Insertion	$O(\log N)$	$O(N)$
Deletion	$O(\log N)$	$O(N)$

- **Skewed Tree**: each internal node has only one child
- **Balanced Tree**: the left and the right subtrees have similar sizes

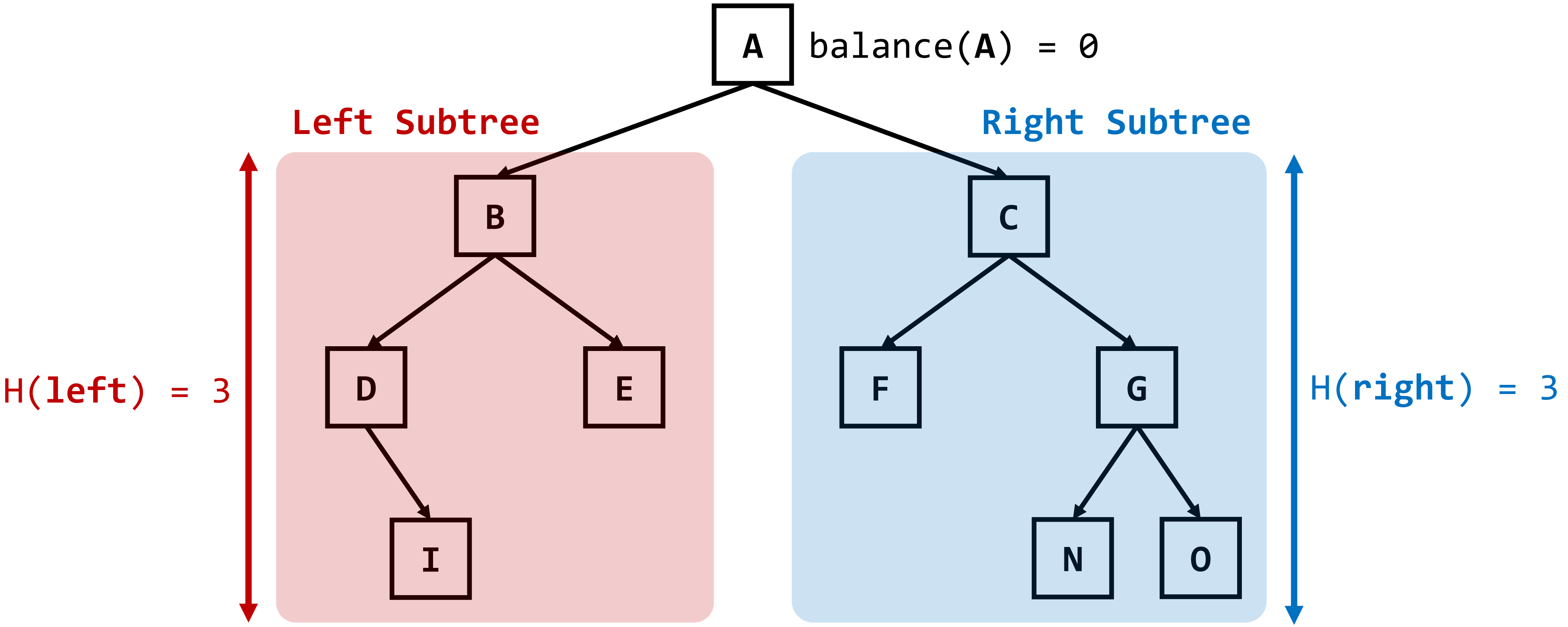




Balanced Binary Trees

- The **balance factor** of a node **X** in a binary tree is defined by

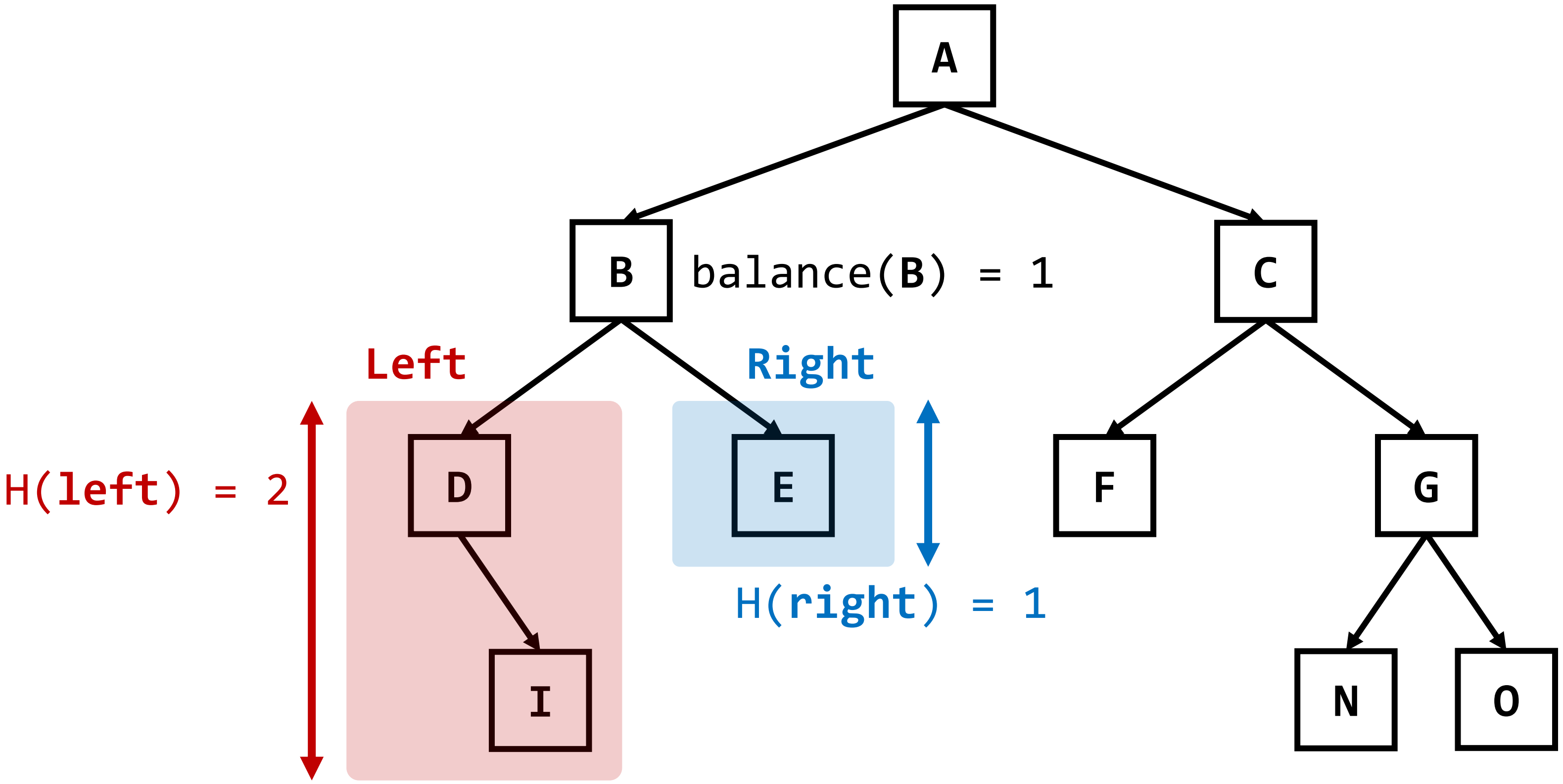
`balance(X) = height(left subtree) – height(right subtree)`



Balanced Binary Trees

- The **balance factor** of a node **X** in a binary tree is defined by

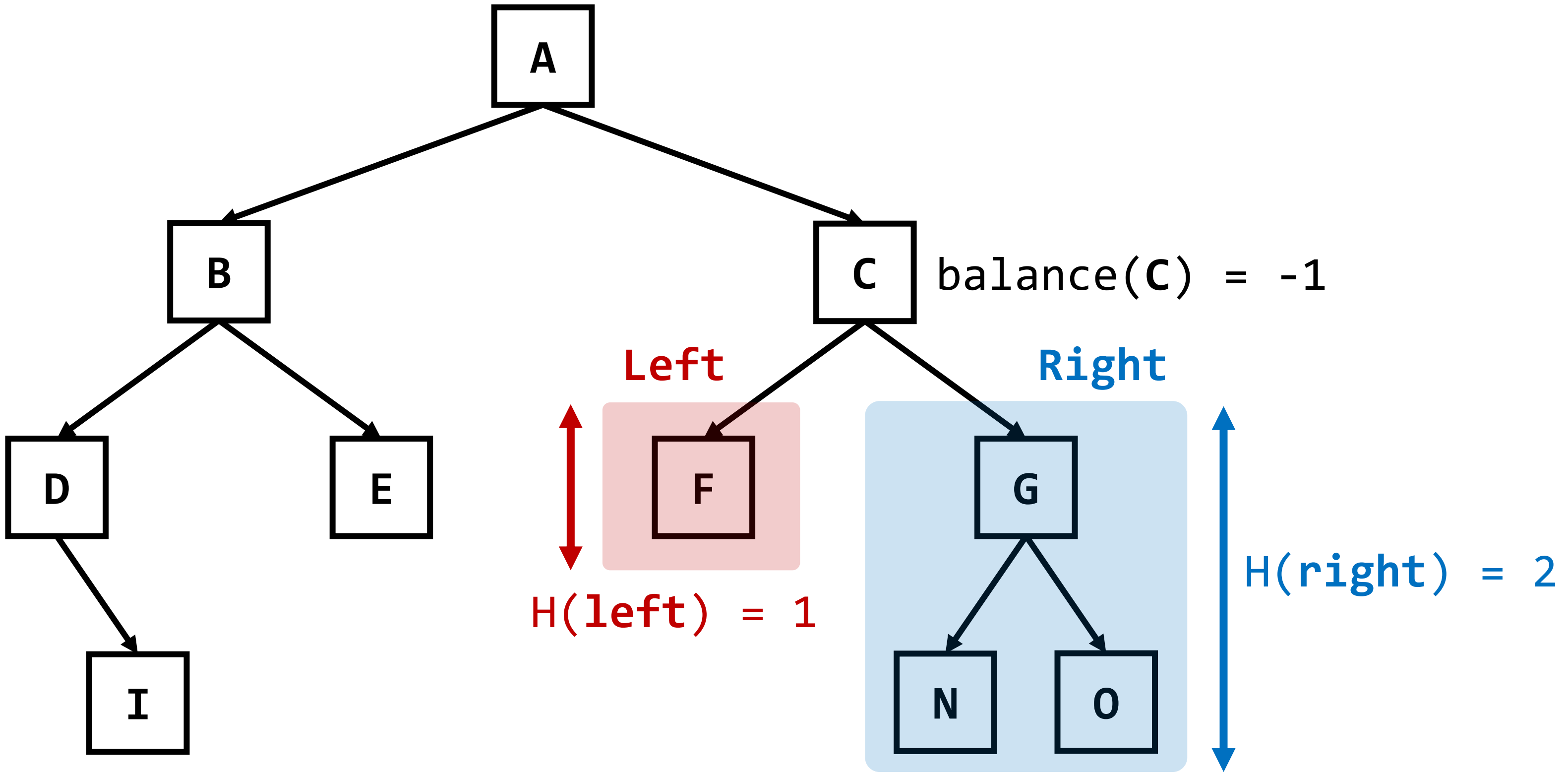
`balance(X) = height(left subtree) – height(right subtree)`



Balanced Binary Trees

- The **balance factor** of a node **X** in a binary tree is defined by

`balance(`**`X`**`) = height(`**`left subtree`**`) – height(`**`right subtree`**`)`

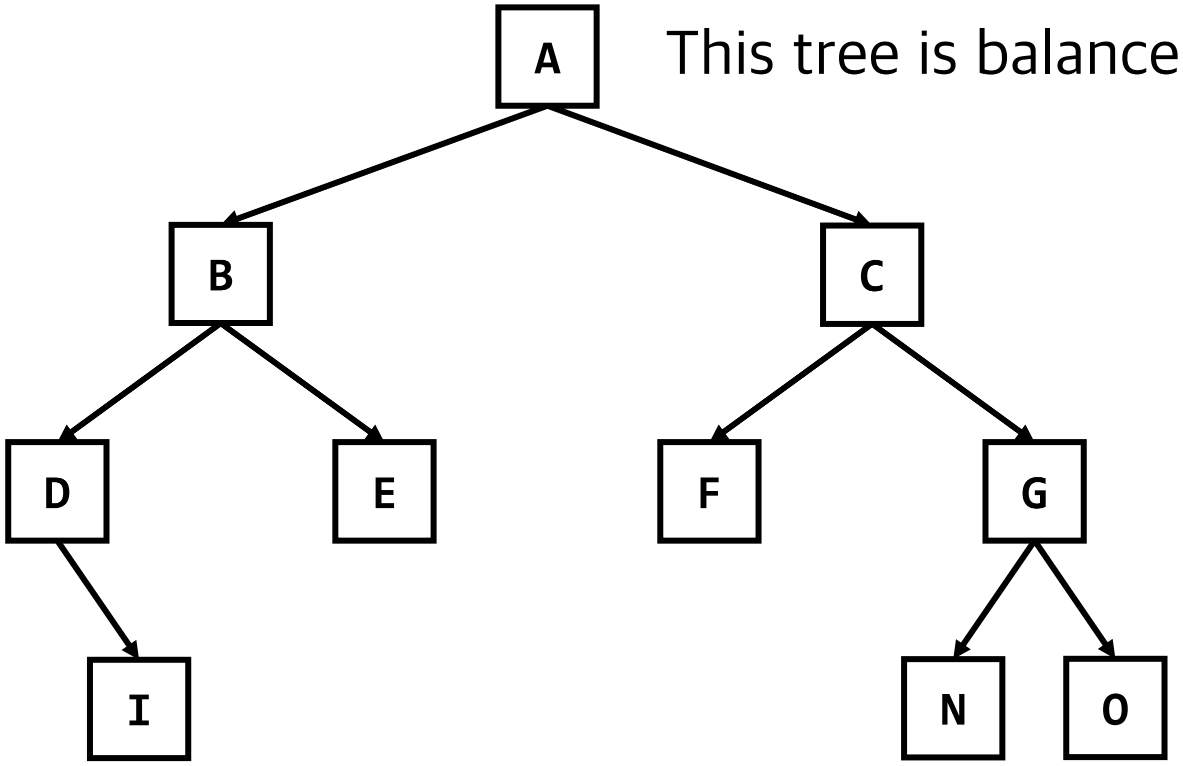


Balanced Binary Trees

- The **balance factor** of a node **X** in a binary tree is defined by

  `balance(X) = height(left subtree) – height(right subtree)`

- A binary tree T is **balanced** if |`balance(X)`| ≤ 1 for any node **X**



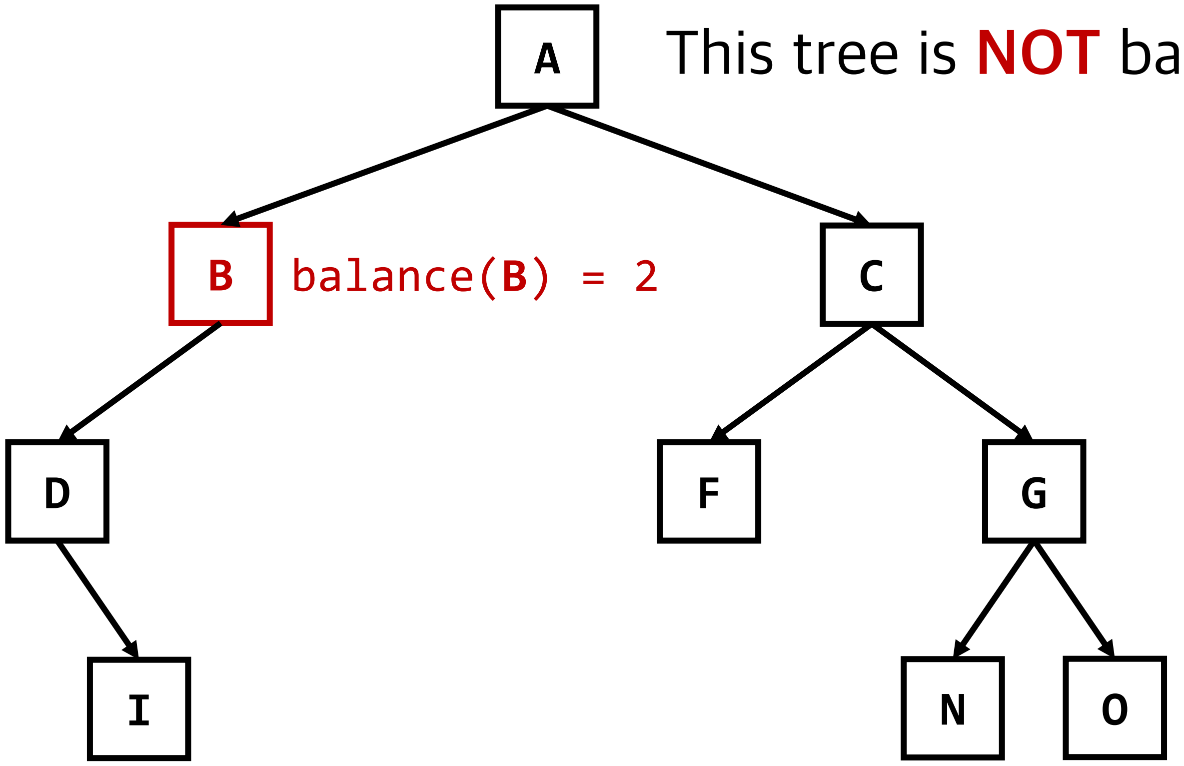
This tree is balanced

Balanced Binary Trees

- The **balance factor** of a node **X** in a binary tree is defined by

  `balance(X) = height(left subtree) – height(right subtree)`

- A binary tree T is **balanced** if |`balance(X)`| ≤ **1** for any node **X**



This tree is **NOT** balanced

Balanced Binary Trees

- The **balance factor** of a node **`X`** in a binary tree is defined by

```
balance(X) = height(left subtree) - height(right subtree)
```

- A binary tree T is **balanced** if |`balance(X)`| ≤ `1` for any node **`X`**
  - If a balanced tree T has N nodes, the height of the tree is $O(\log_2 N)$
  - A balanced BST has $O(\log_2 N)$ time complexity for search!

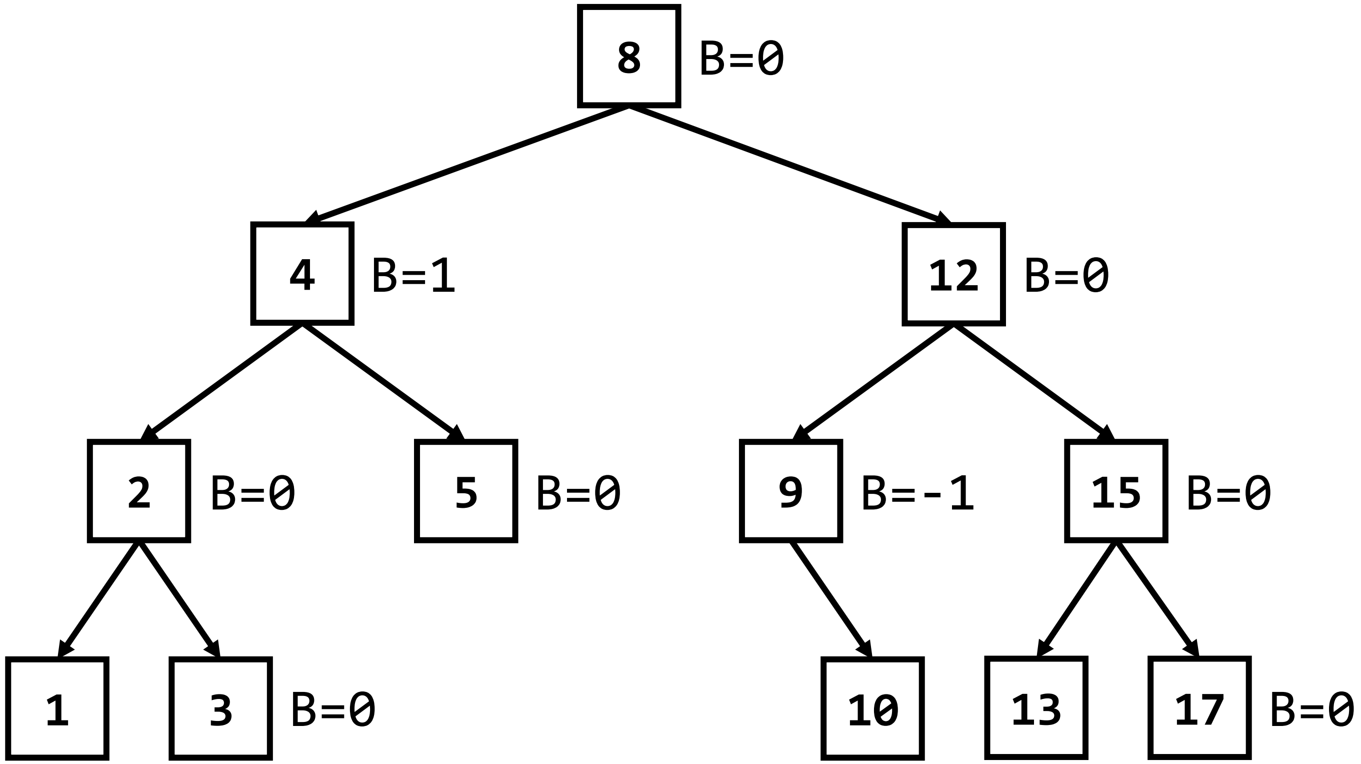
(Q) How does the balance factors change after insertion or deletion?

- After the operations on a balanced BT, will the updated tree still be balanced?
- If not, how to re-balance the tree?

Balanced Binary Trees

(Q) How does the balance factors change after **insertion**?

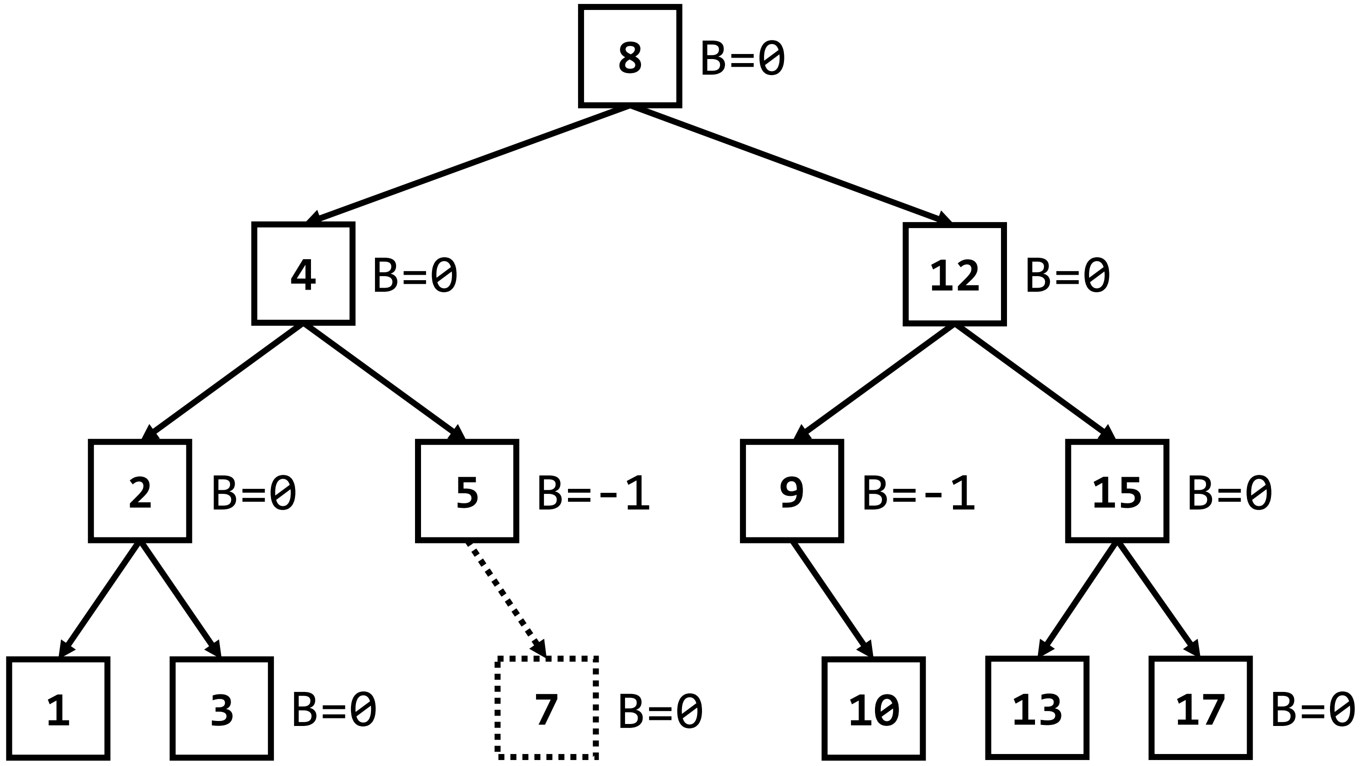
- Insert a node **7** into the below tree …



Balanced Binary Trees

(Q) How does the balance factors change after **insertion**?

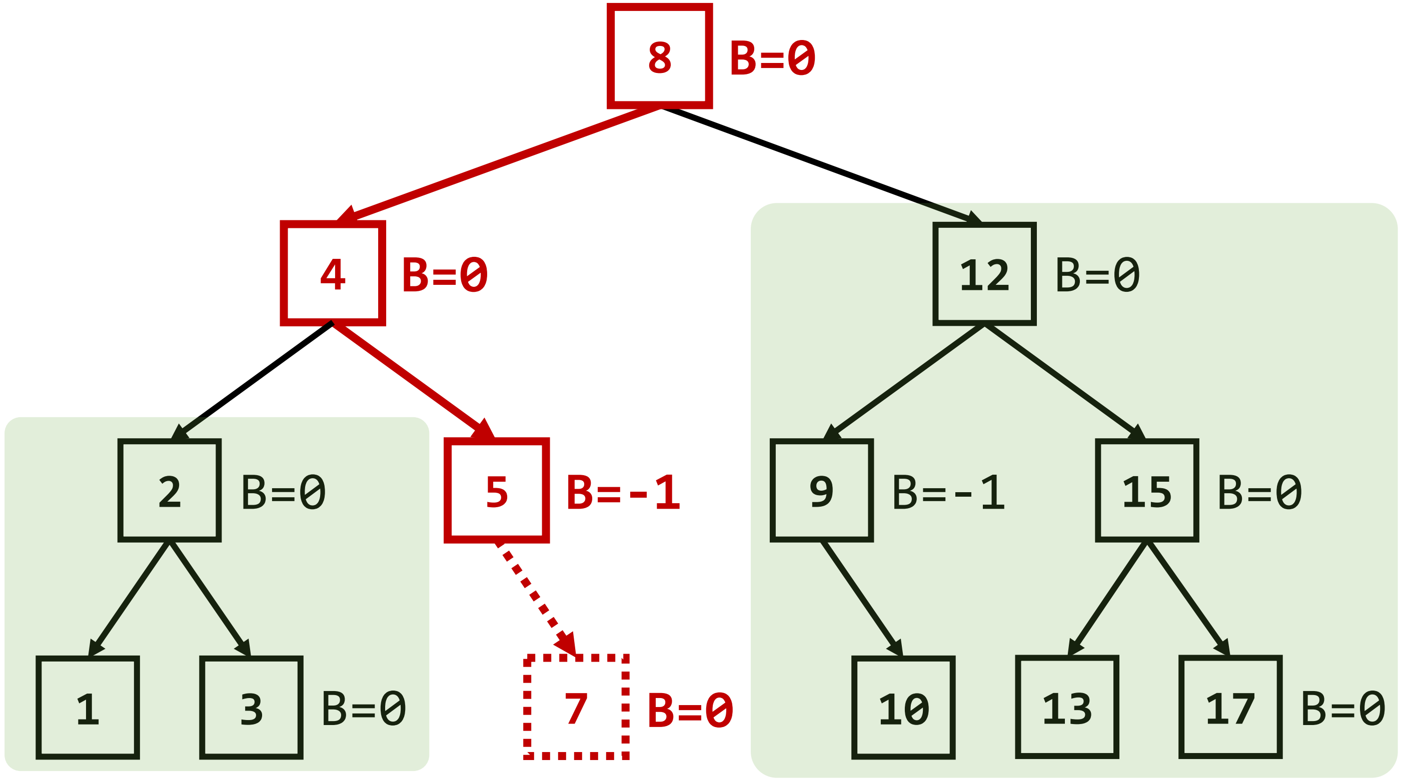
- Insert a node **7** into the below tree …



Balanced Binary Trees

(Q) How does the balance factors change after **insertion**?

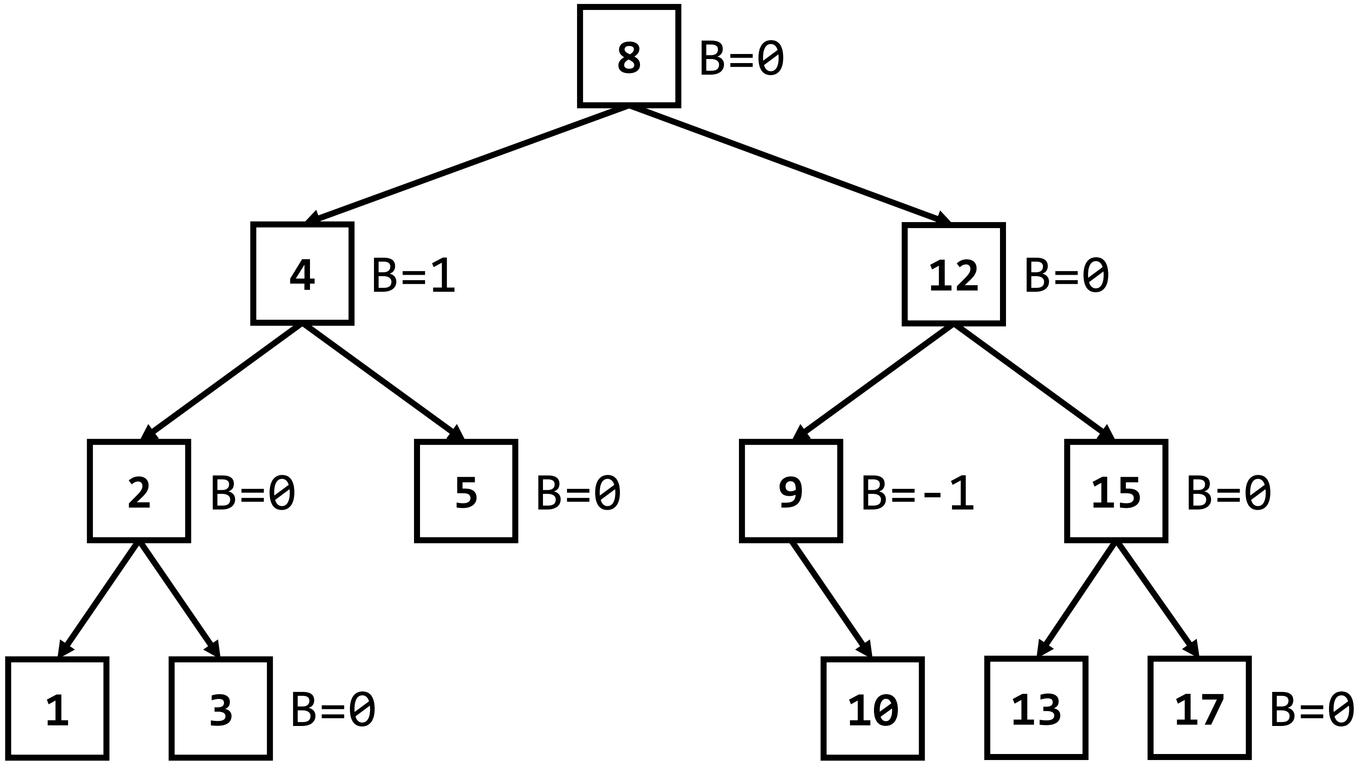
- Insert a node **7** into the below tree …
- **The nodes on the search trajectory** might be changed, other subtrees are not



Balanced Binary Trees

(Q) How does the balance factors change after **deletion**?

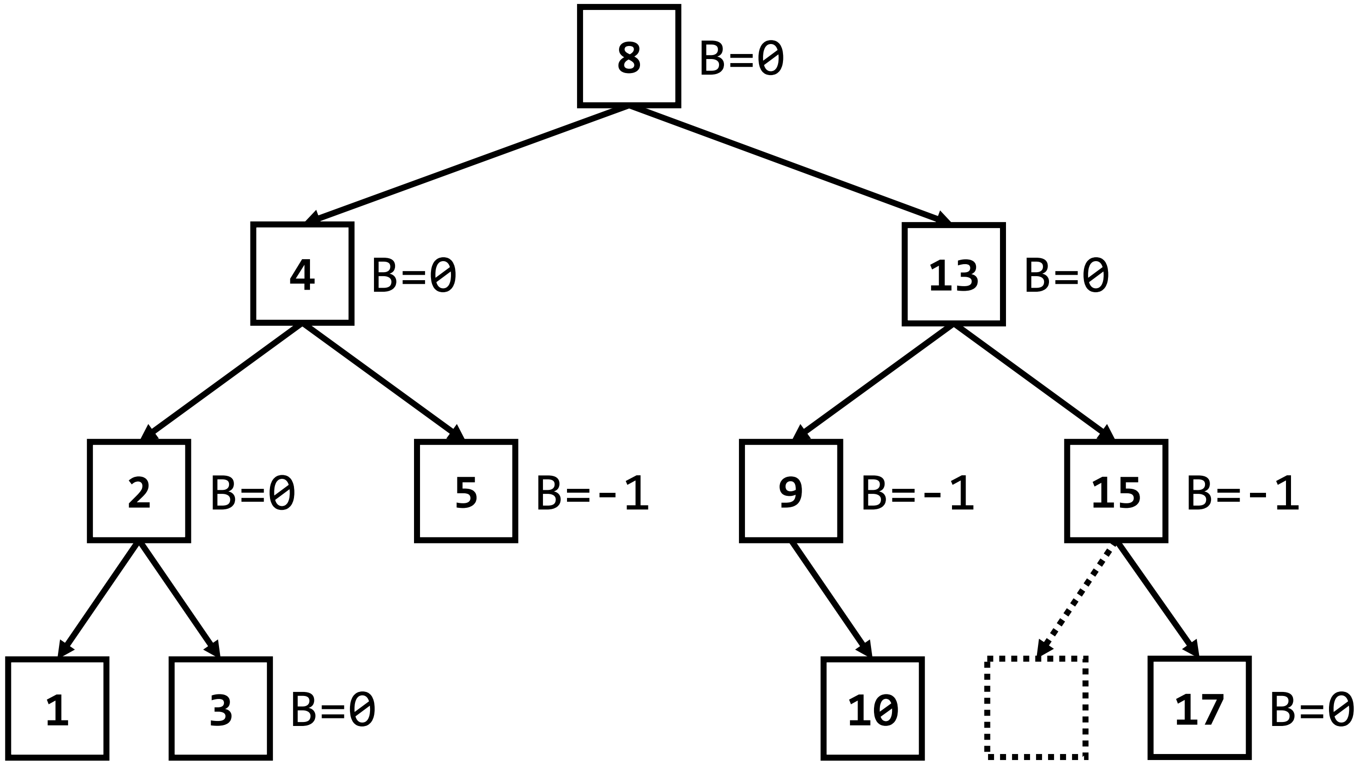
- Delete a node **12** from the below tree …



Balanced Binary Trees

(Q) How does the balance factors change after **deletion**?

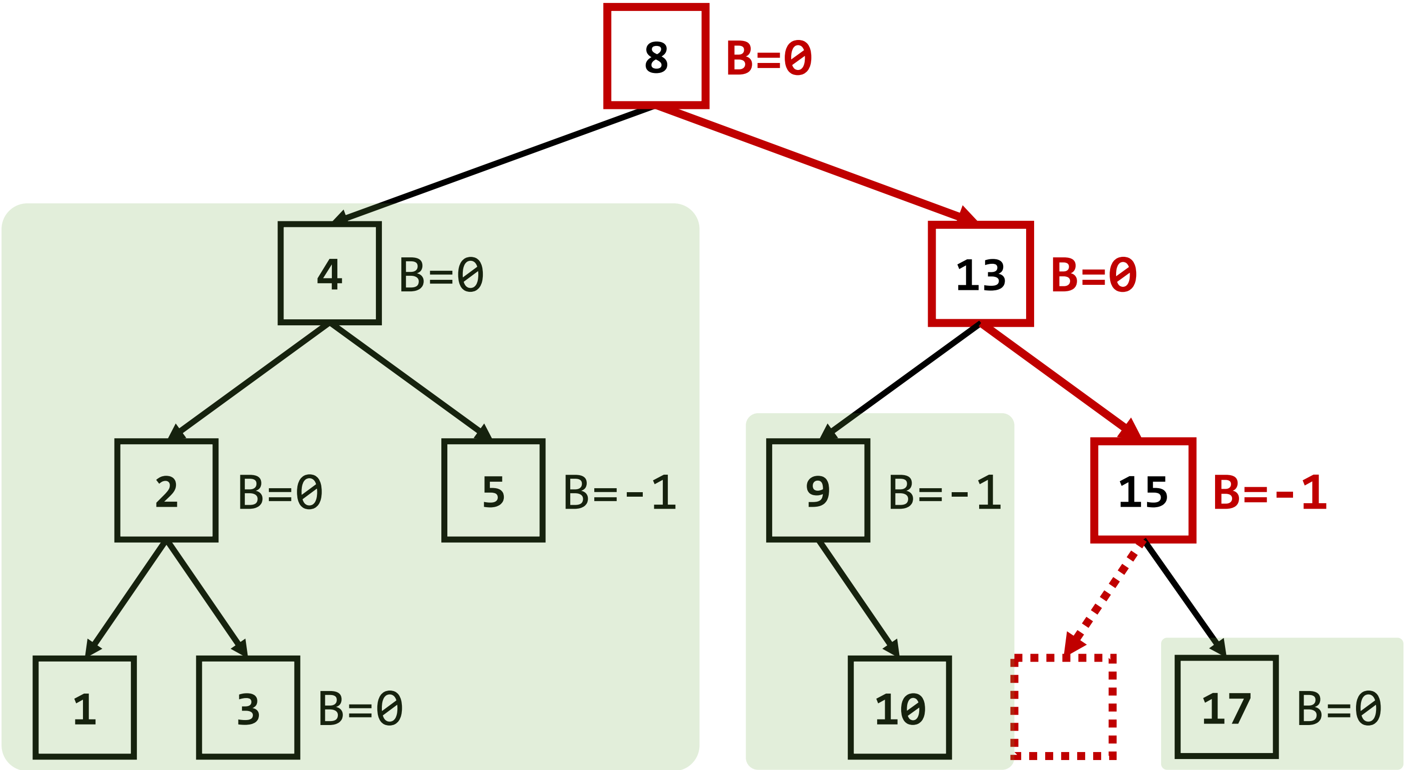
- Delete a node **12** from the below tree …



Balanced Binary Trees

(Q) How does the balance factors change after **deletion**?

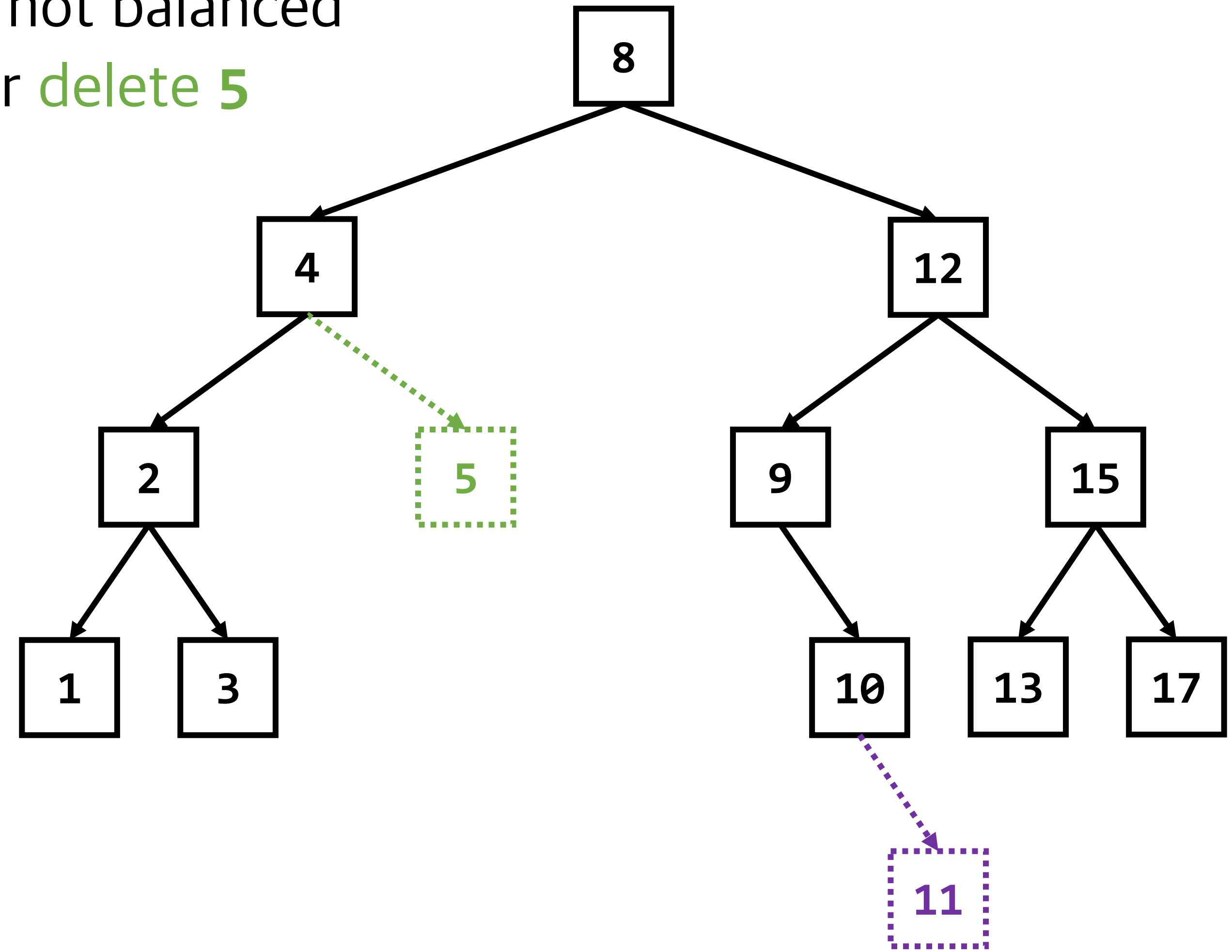
- Delete a node **12** from the below tree …
- **The nodes on the search trajectory** might be changed, other subtrees are not



Balanced Binary Trees

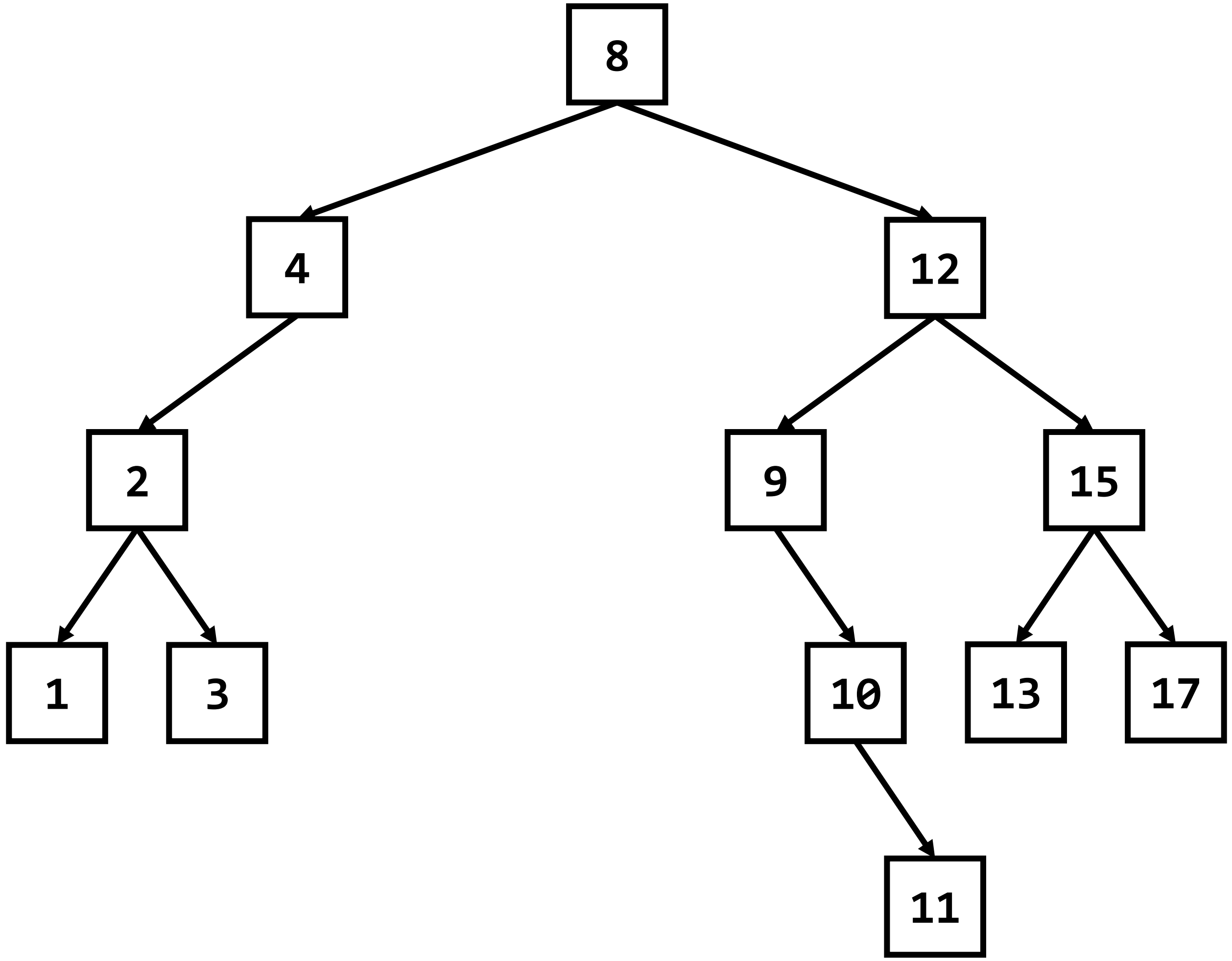
(Q) After the operations on a balanced BT, is the updated tree still balanced?

- **No.** It might be not balanced
- E.g., insert **11** or delete **5**



Balanced Binary Trees

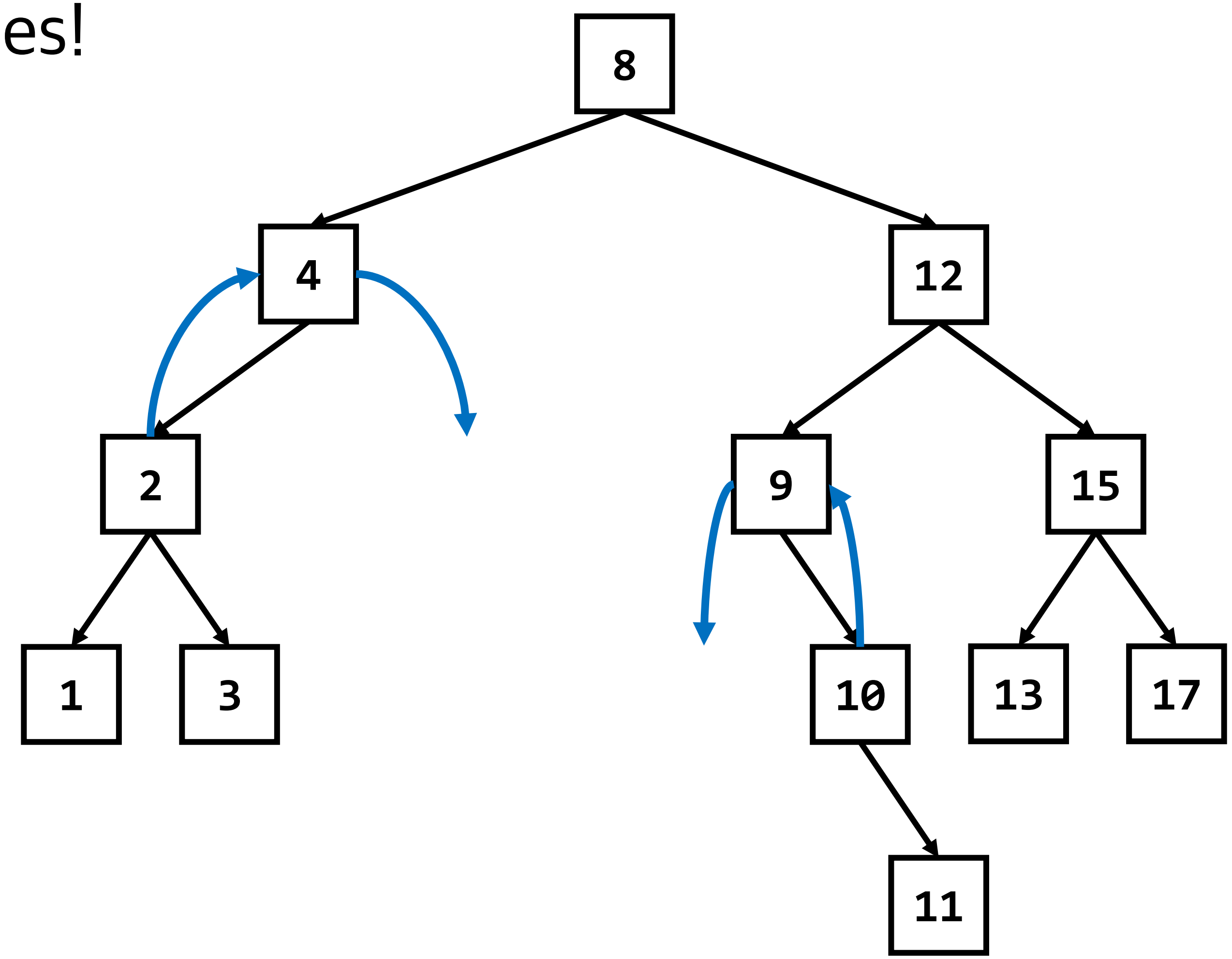
(Q) How to re-balance this tree?



Balanced Binary Trees

(Q) How to re-balance this tree?

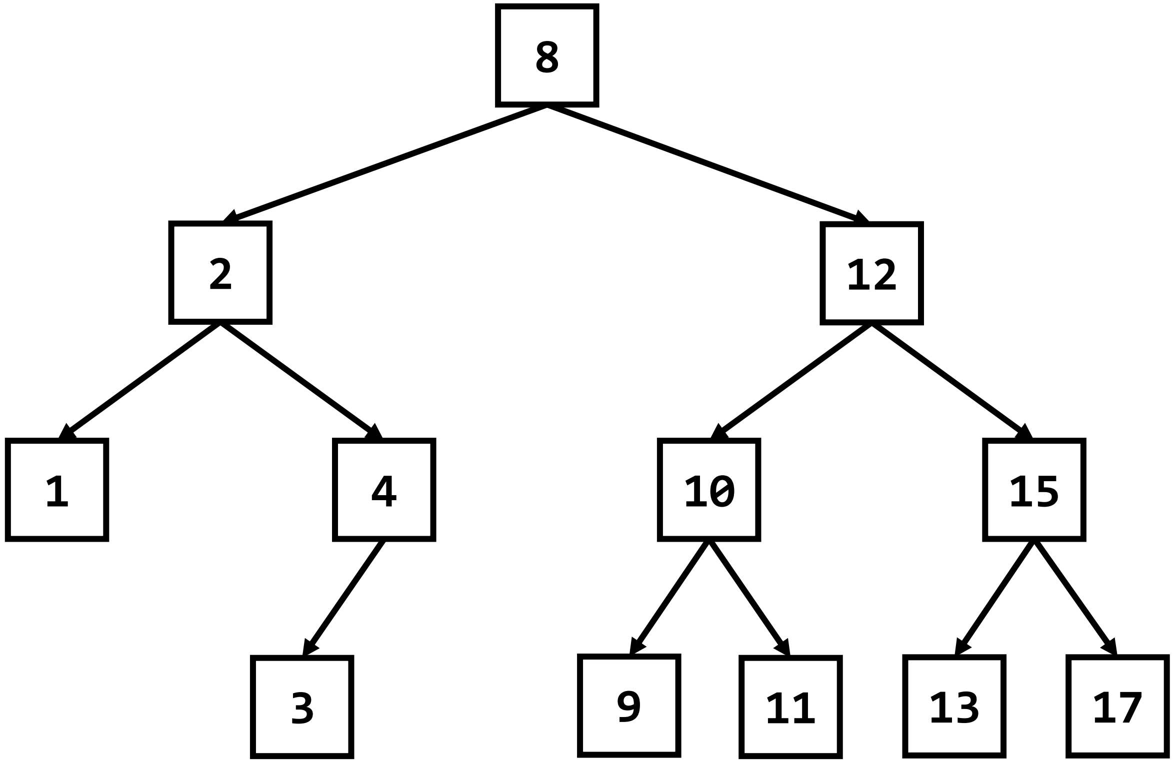
(A) Rotate subtrees!



Balanced Binary Trees

(Q) How to re-balance this tree?

(A) Rotate subtrees!



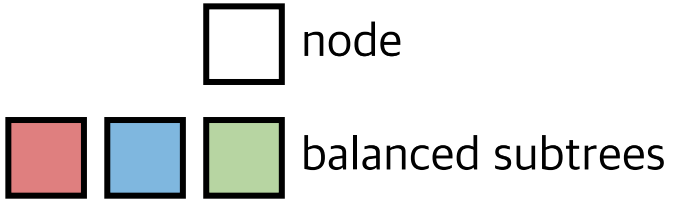
This **self-balancing** BST is called **AVL tree**!

AVL Trees

- **AVL tree** is a **self-balancing** BST invented by G.M. **A**delson-**V**elsky and E.M. **L**andis in 1962
  - AVL tree is always balanced → Its height is $O(\log_2 N)$
  - AVL tree requires $O(\log_2 N)$ time complexity for search, insertion, and deletion

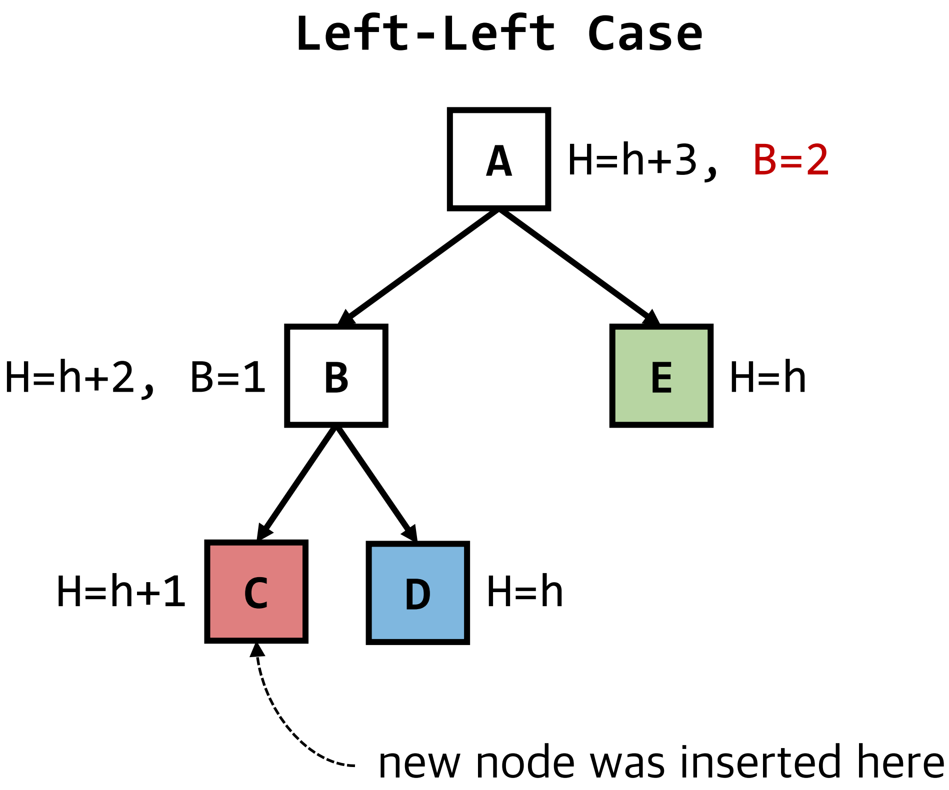
  - AVL tree updates its structure to remain balanced after insertion or deletion
  - **(Q)** How to update?

AVL Trees - Rotations for Insertion



(Q) How to re-balance the tree after insertion?

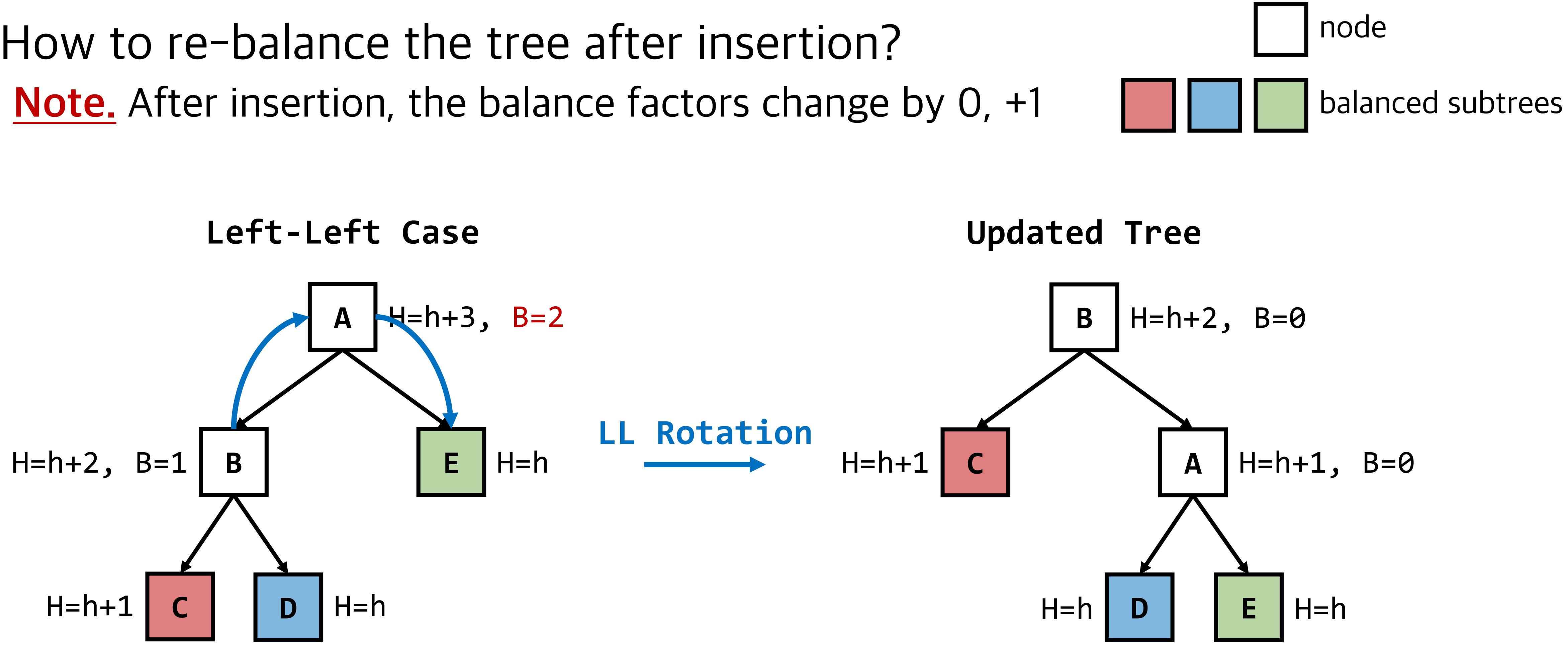
- **Note.** After insertion, the balance factors change by 0, +1



AVL Trees - Rotations for Insertion

(Q) How to re-balance the tree after insertion?

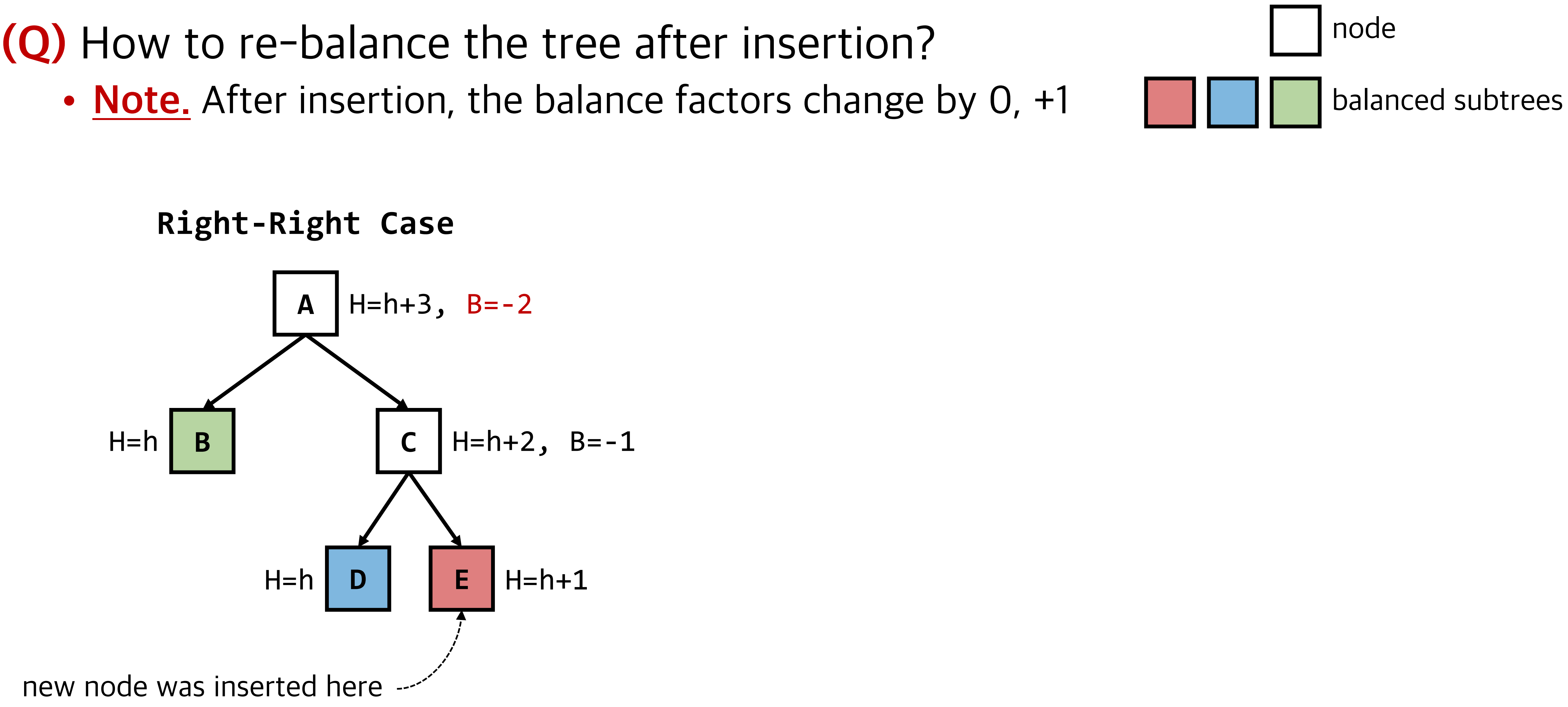
- **Note.** After insertion, the balance factors change by 0, +1



AVL Trees - Rotations for Insertion

(Q) How to re-balance the tree after insertion?

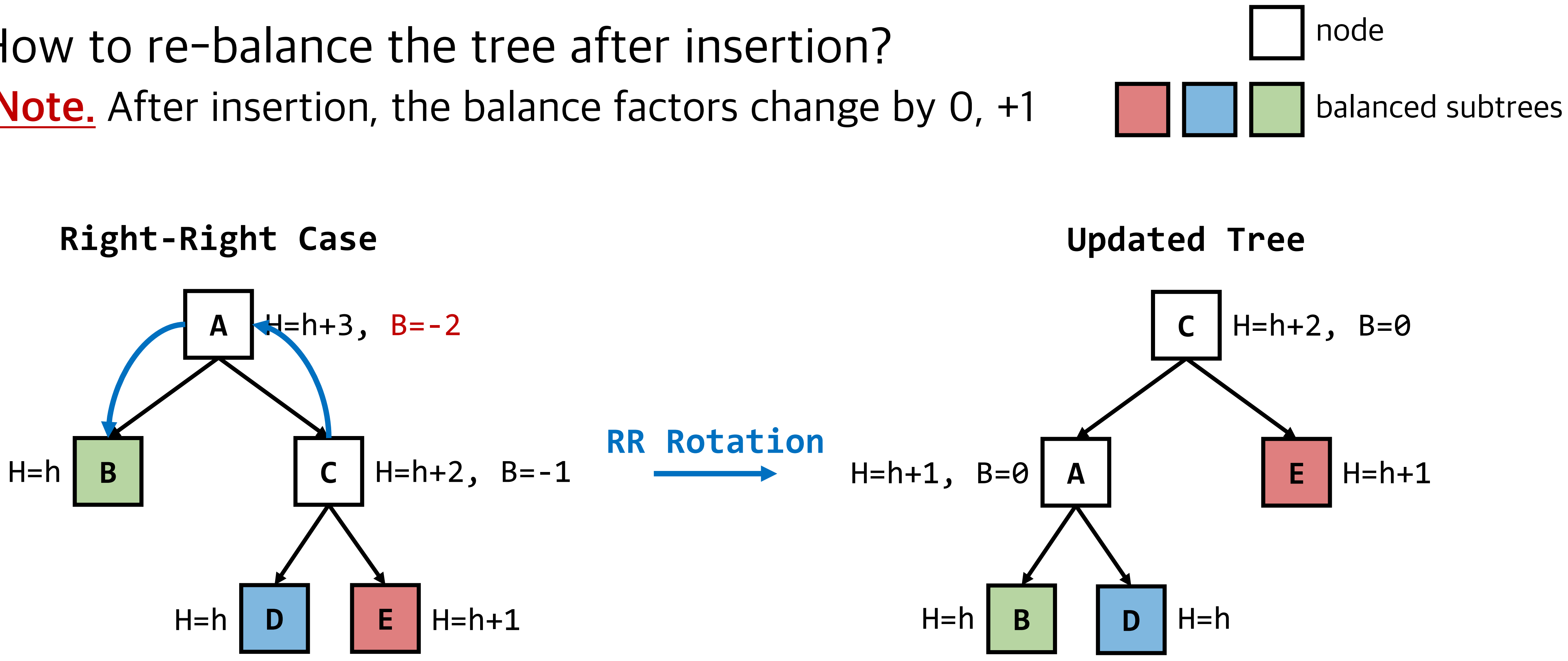
- **Note.** After insertion, the balance factors change by 0, +1



AVL Trees - Rotations for Insertion

(Q) How to re-balance the tree after insertion?

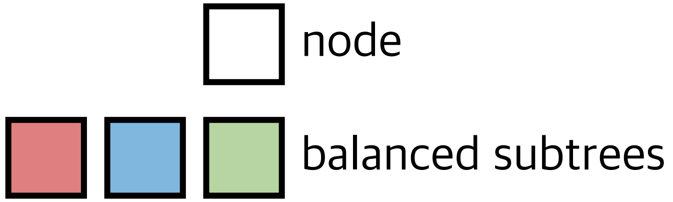
- **Note.** After insertion, the balance factors change by 0, +1

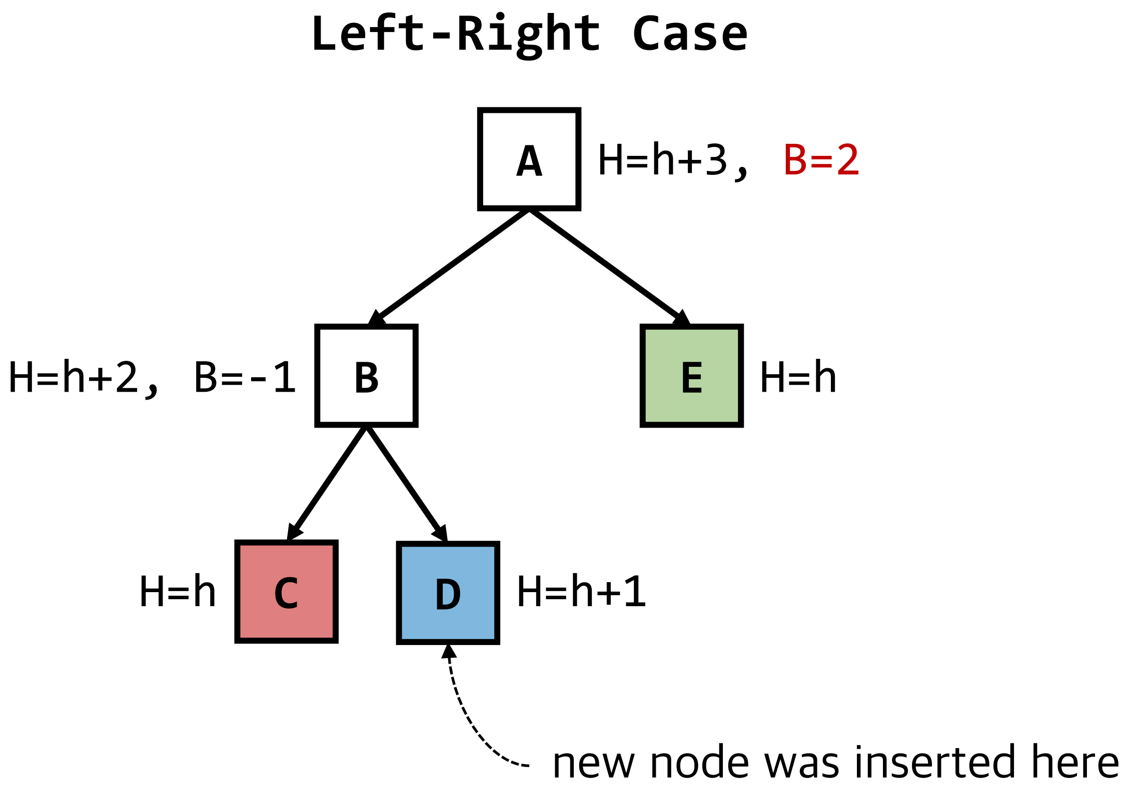


AVL Trees - Rotations for Insertion

(Q) How to re-balance the tree after insertion?

- **Note.** After insertion, the balance factors change by 0, +1

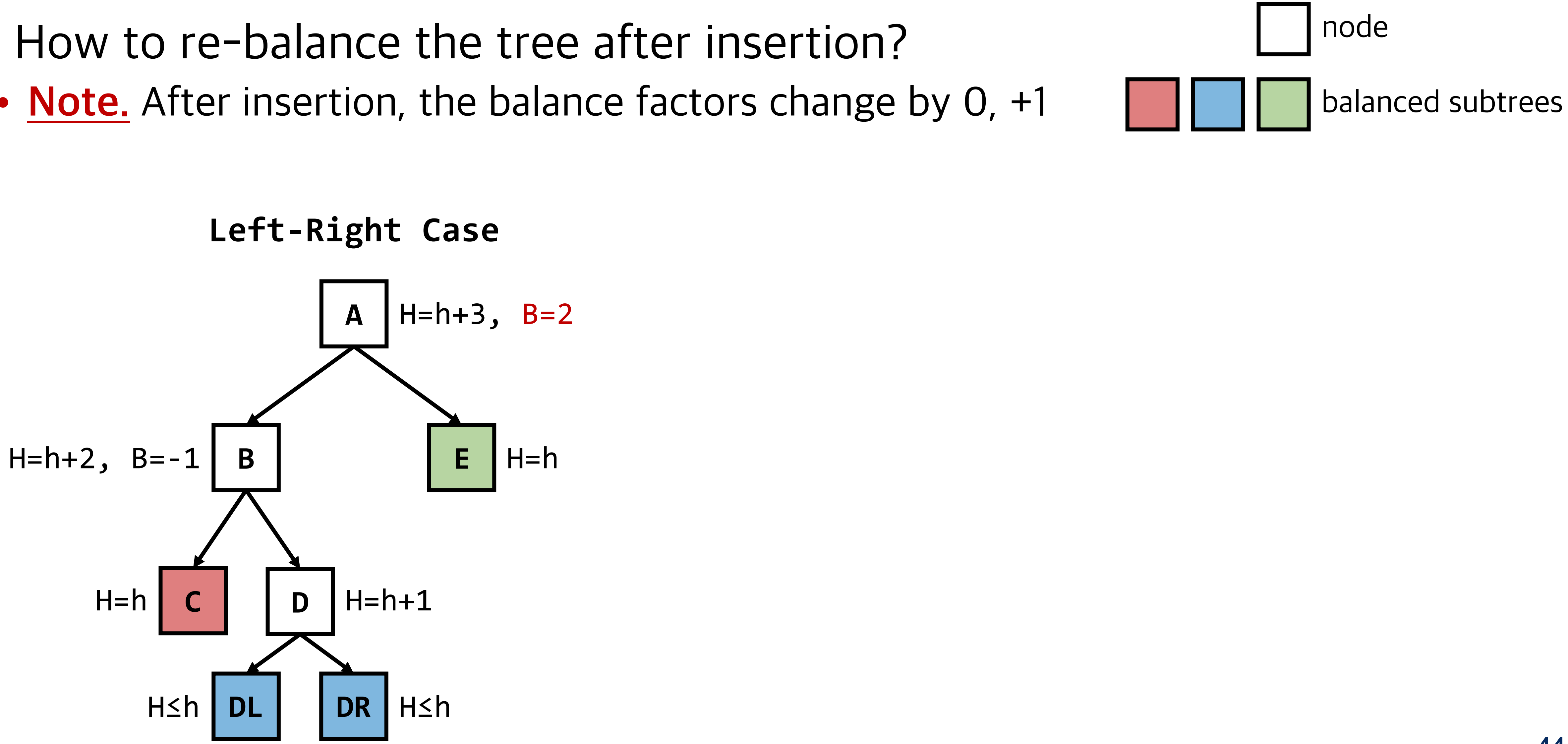




AVL Trees - Rotations for Insertion

(Q) How to re-balance the tree after insertion?

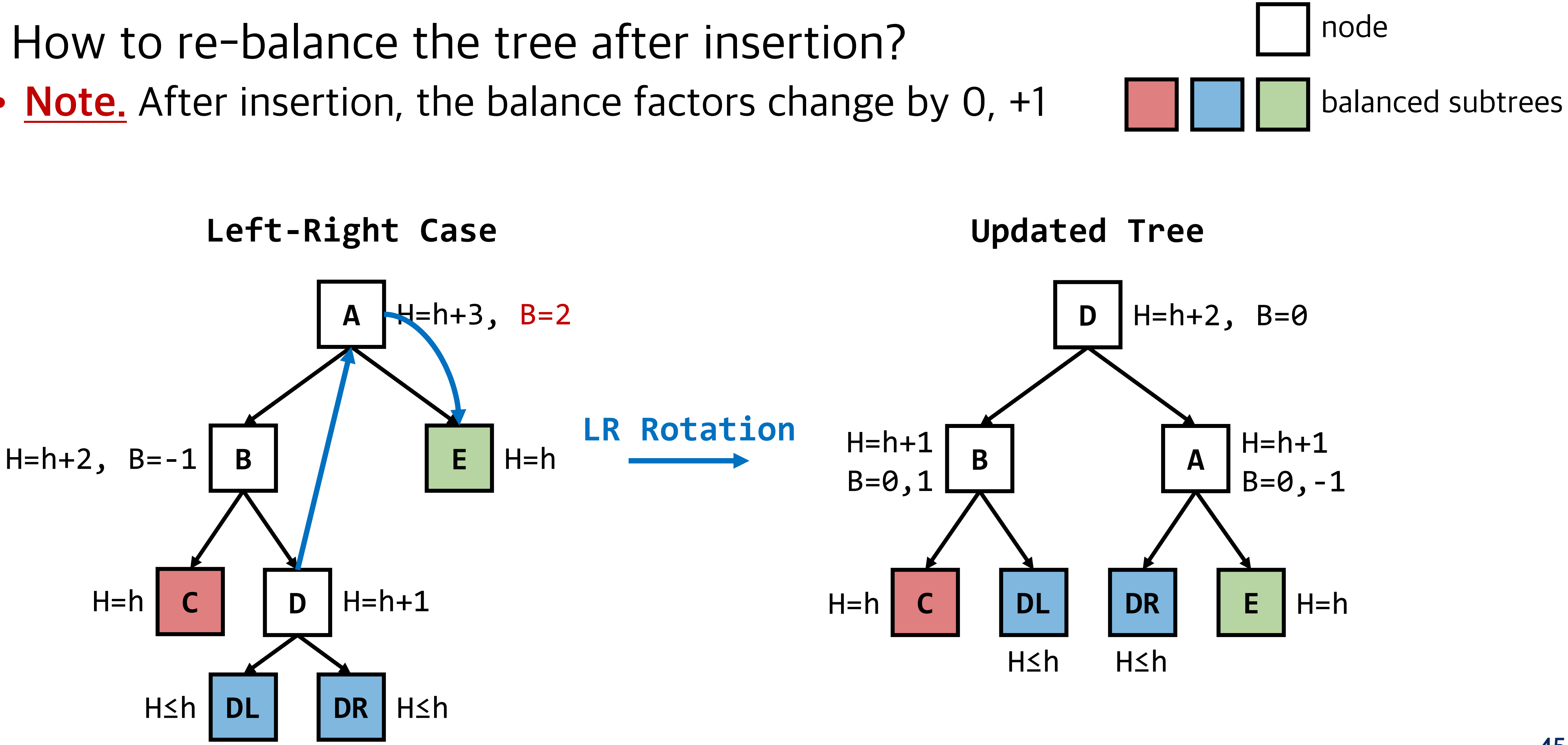
- **Note.** After insertion, the balance factors change by 0, +1



AVL Trees - Rotations for Insertion

(Q) How to re-balance the tree after insertion?

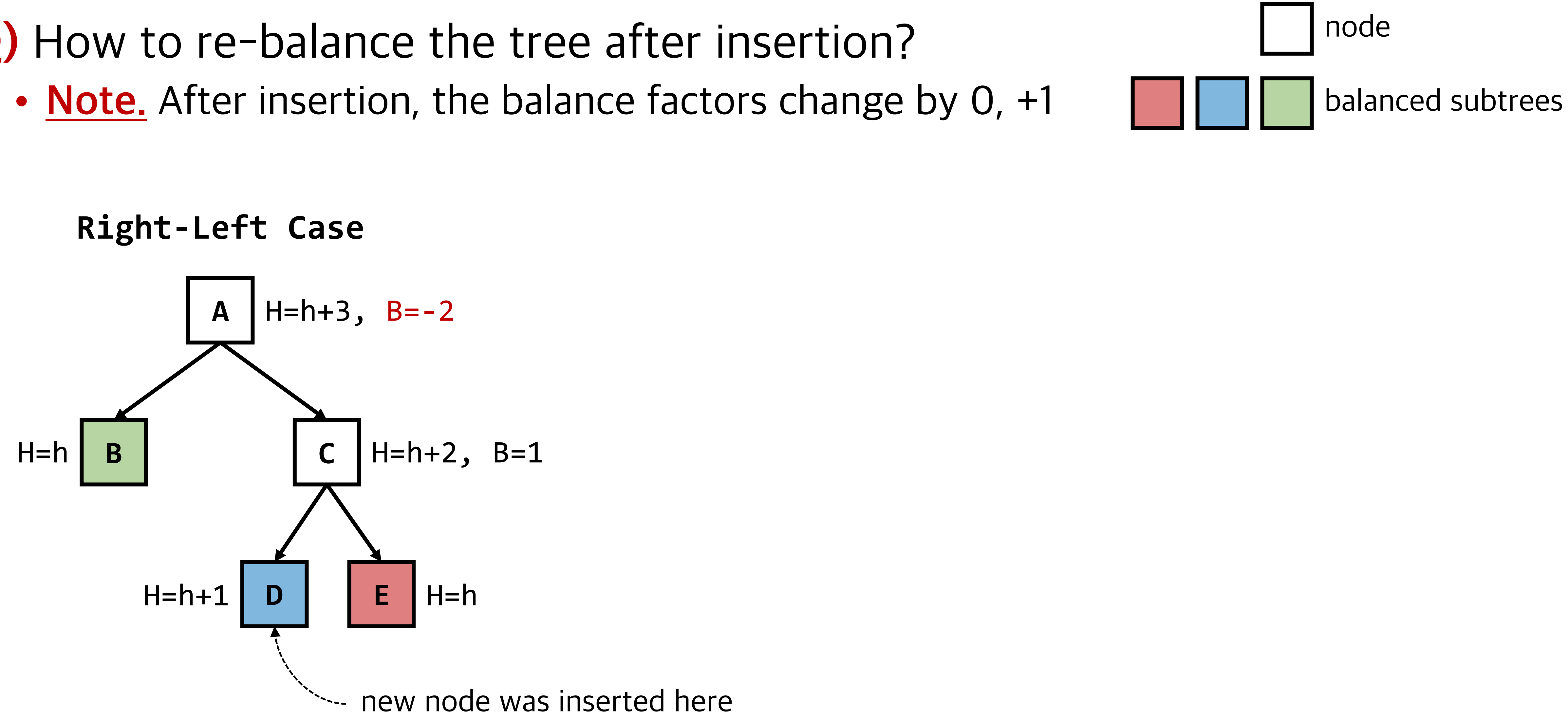
- **Note.** After insertion, the balance factors change by 0, +1



AVL Trees - Rotations for Insertion

(Q) How to re-balance the tree after insertion?

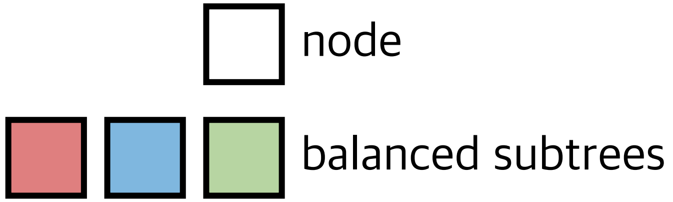
- **Note.** After insertion, the balance factors change by 0, +1

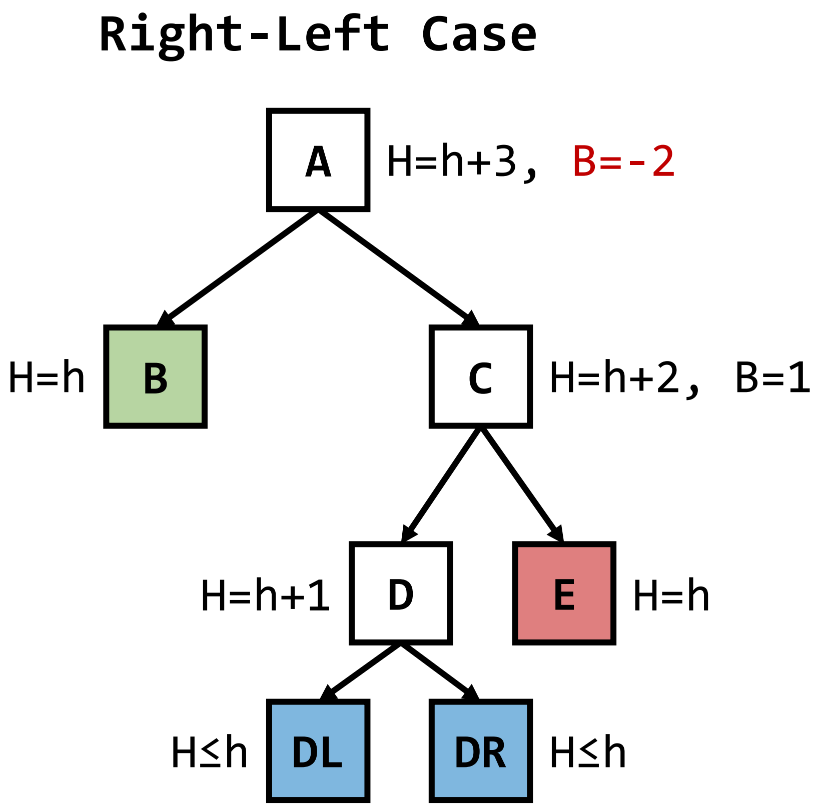


AVL Trees - Rotations for Insertion

(Q) How to re-balance the tree after insertion?

- **Note.** After insertion, the balance factors change by 0, +1

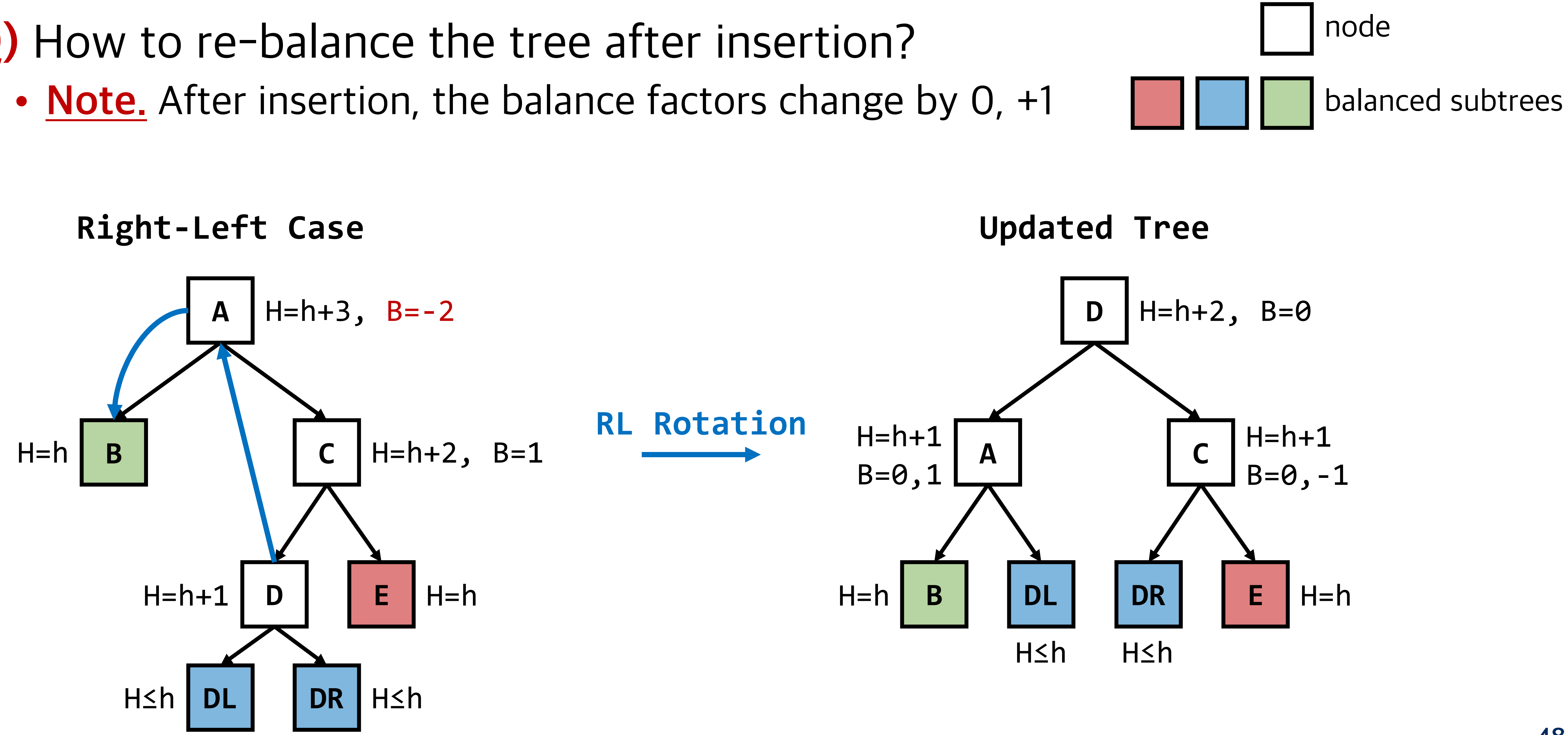




AVL Trees - Rotations for Insertion

(Q) How to re-balance the tree after insertion?

- **Note.** After insertion, the balance factors change by 0, +1



AVL Trees - Rotations for Deletion

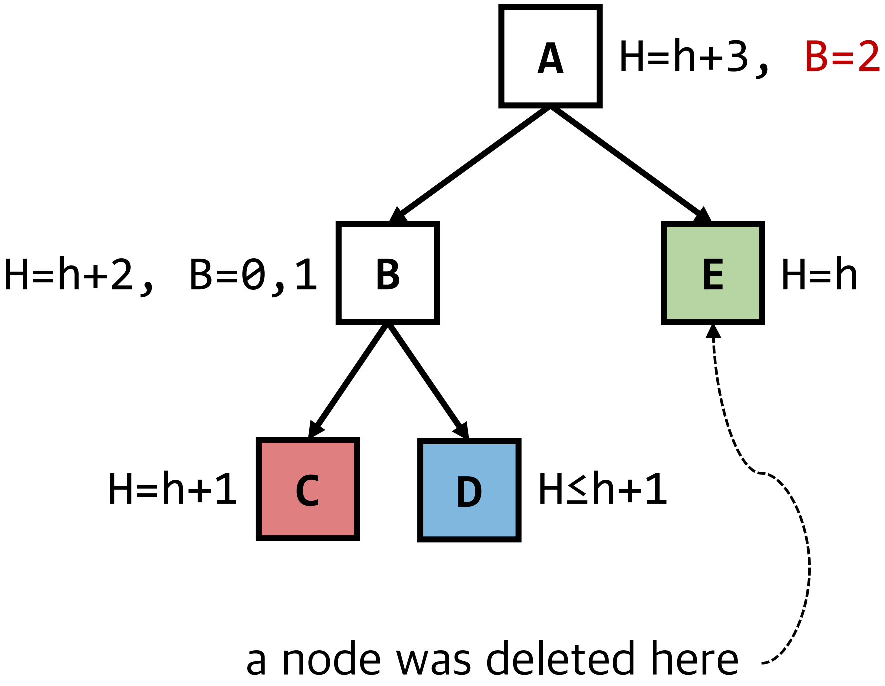
node

balanced subtrees

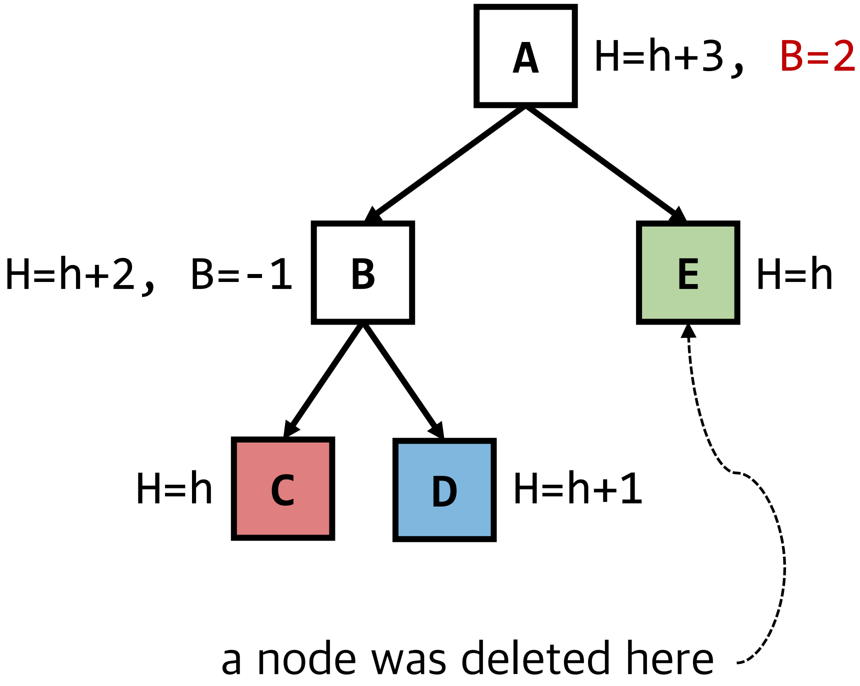
(Q) How to re-balance the tree after deletion?

- **Note.** After deletion, the balance factors change by 0, -1
- Use LL/LR/RR/RL rotation operations

LL Rotation



LR Rotation



AVL Trees - Rotations for Deletion

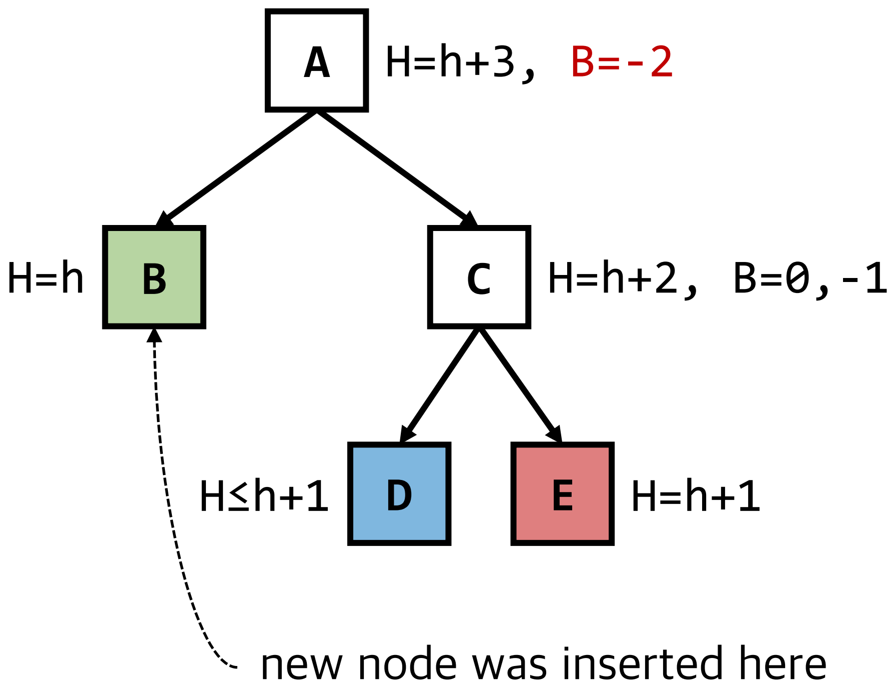
node

balanced subtrees

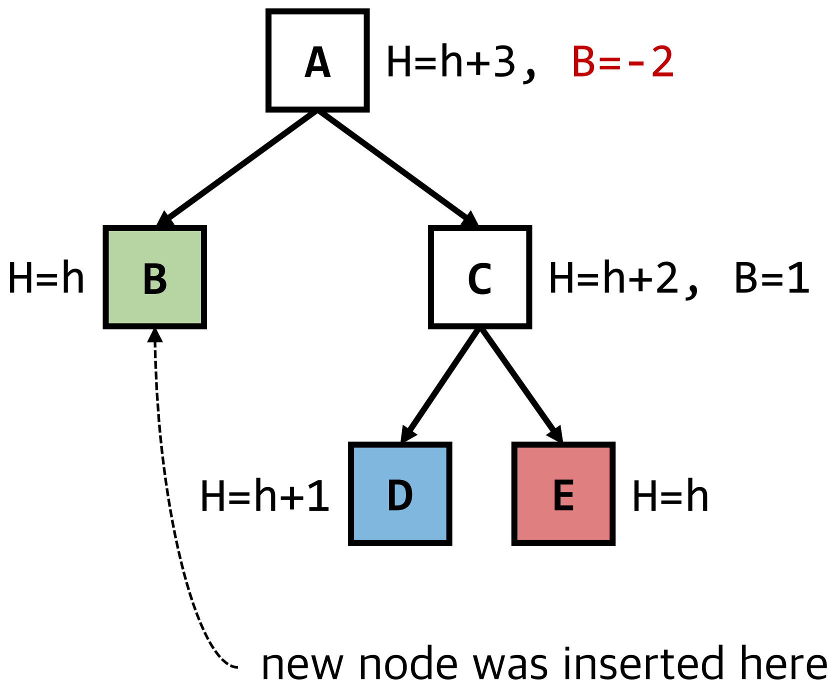
(Q) How to re-balance the tree after deletion?

- **Note.** After deletion, the balance factors change by 0, -1
- Use LL/LR/RR/RL rotation operations

RR Rotation



RL Rotation



AVL Trees - Summary

- **AVL tree** is a **self-balancing** BST
  - AVL tree is always balanced → Its height is $O(\log_2 N)$
  - AVL tree requires $O(\log_2 N)$ time complexity for search, insertion, and deletion

  - AVL tree uses **rotation operations** to remain balanced after insertion or deletion

Any Questions?