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

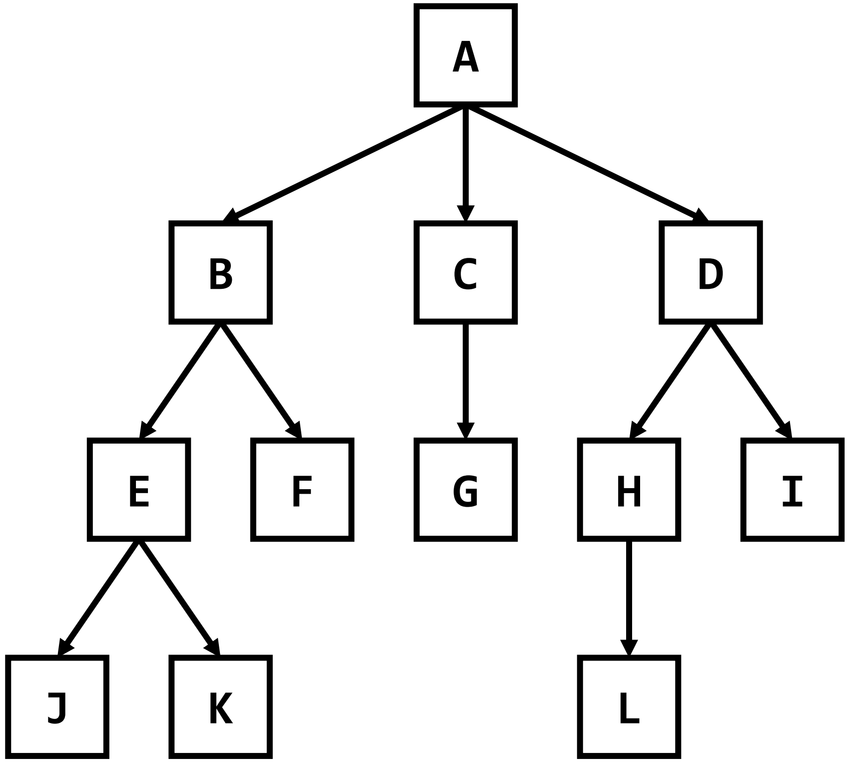
Binary Search Trees

Department of Computer Science and Engineering

Instructor: Hankook Lee (이한국)

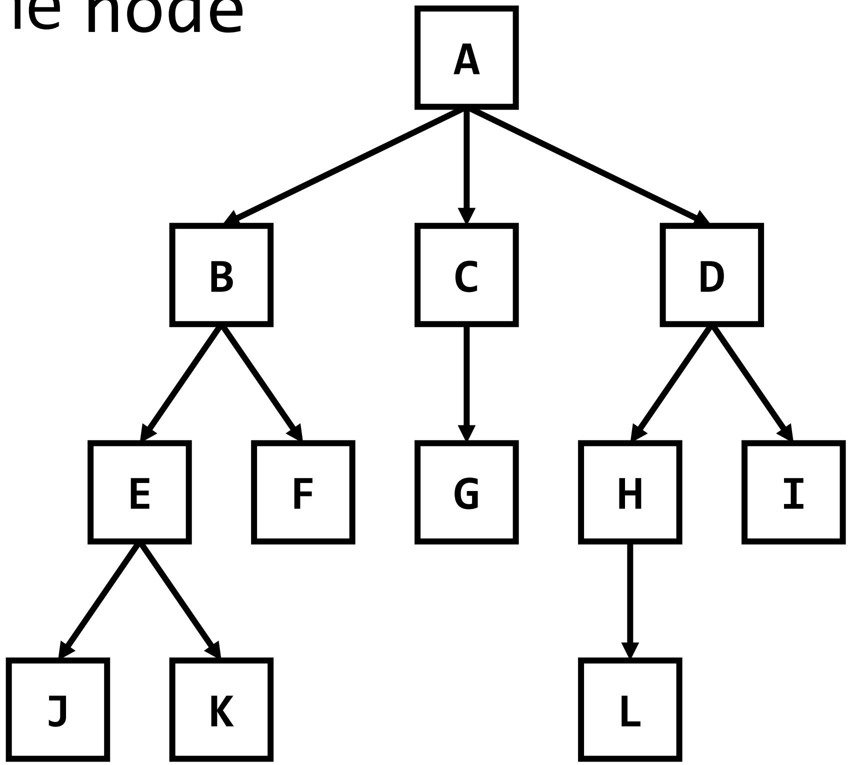
(Recap) What is Tree?

- 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`

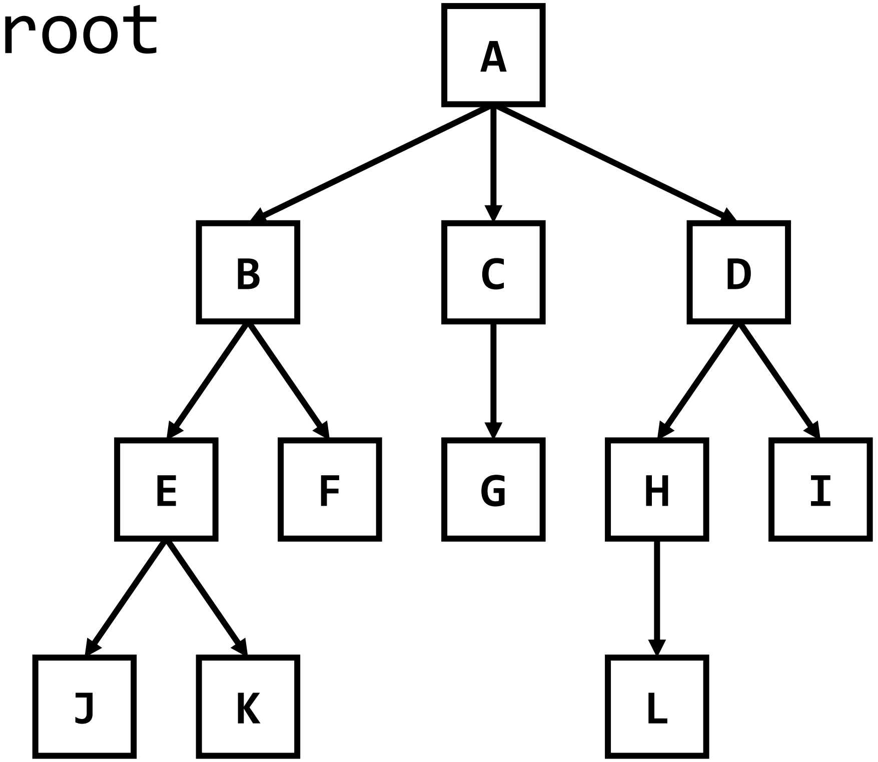


- Examples
  - `B` is the `parent` of `E` and `F`
  - `H` is a `child` of `D`
  - `degree(D) = 2`
  - `degree(J) = 0`

(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

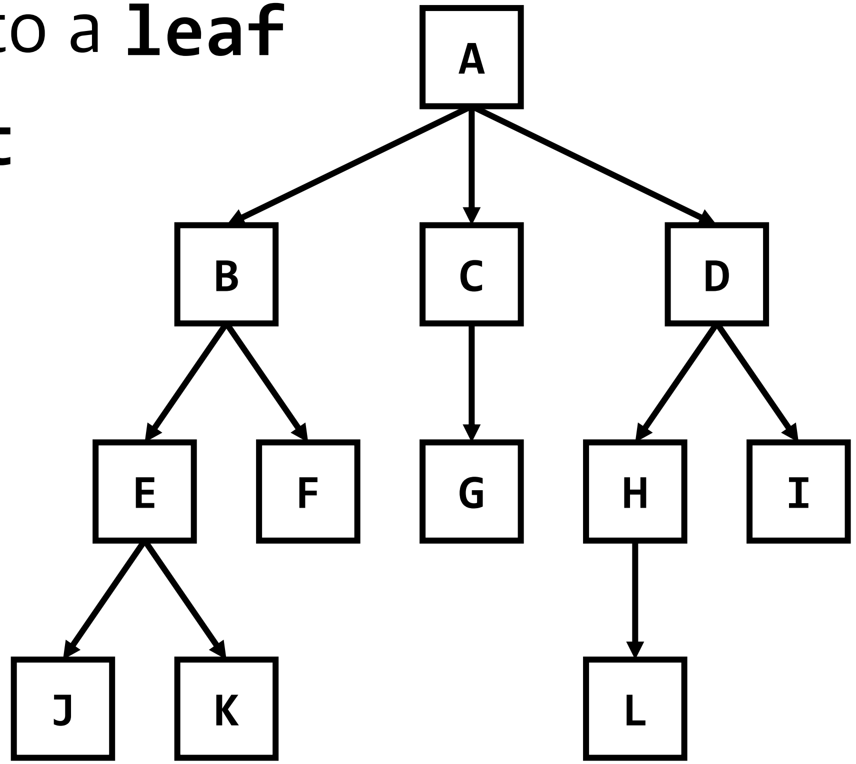


(Recap) Terminology (Node-Level)

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**

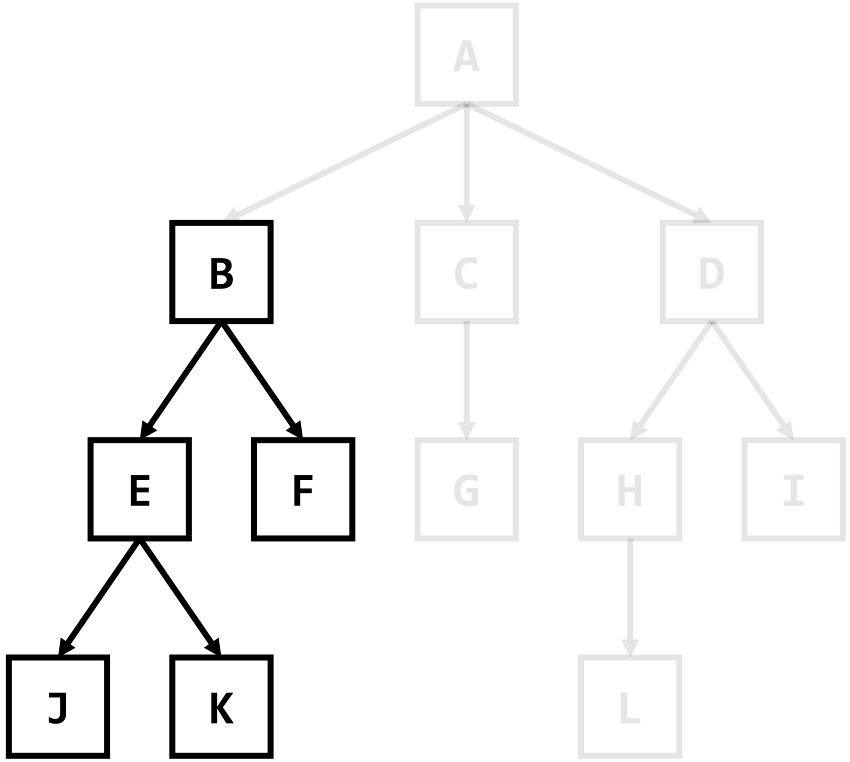


(Recap) Terminology (Node-Level)

Subtree rooted at a **node X**

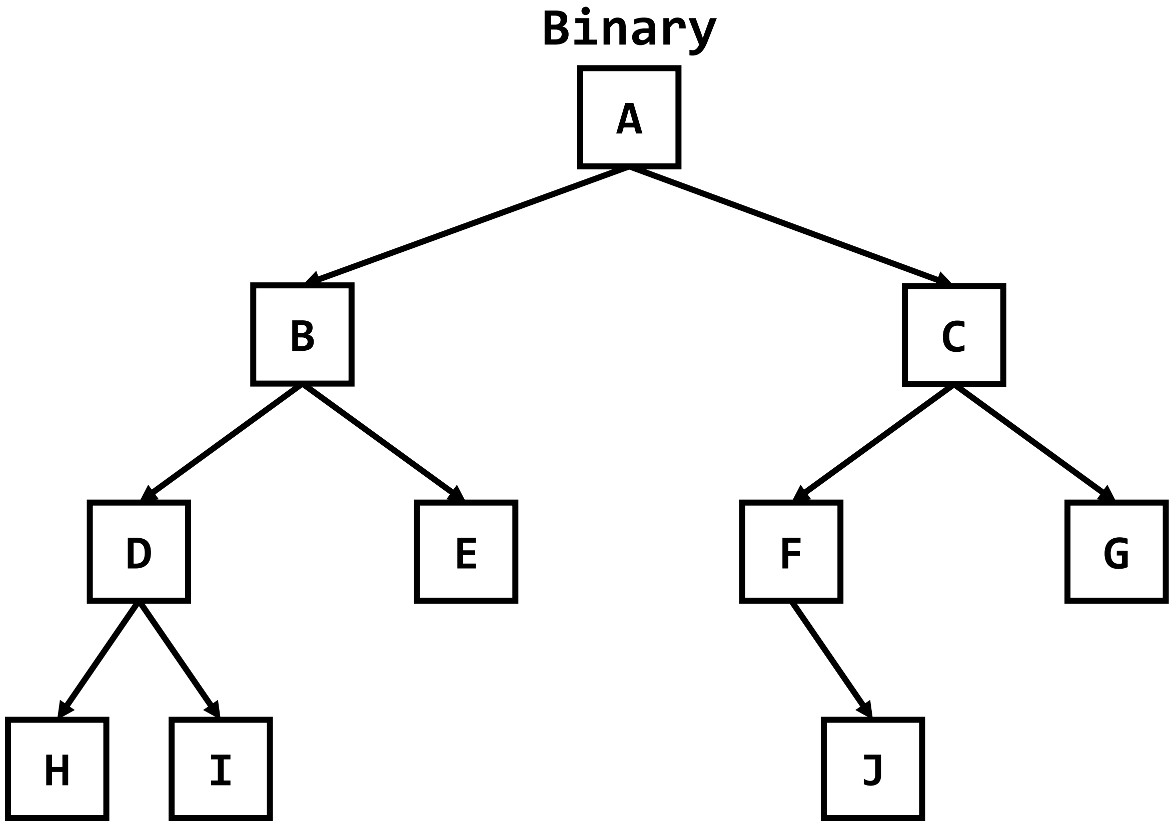
- Any **node** can be treated as the **root node** of its own **subtree**
- The **subtree** includes **X** and all **descendant**s of **X**

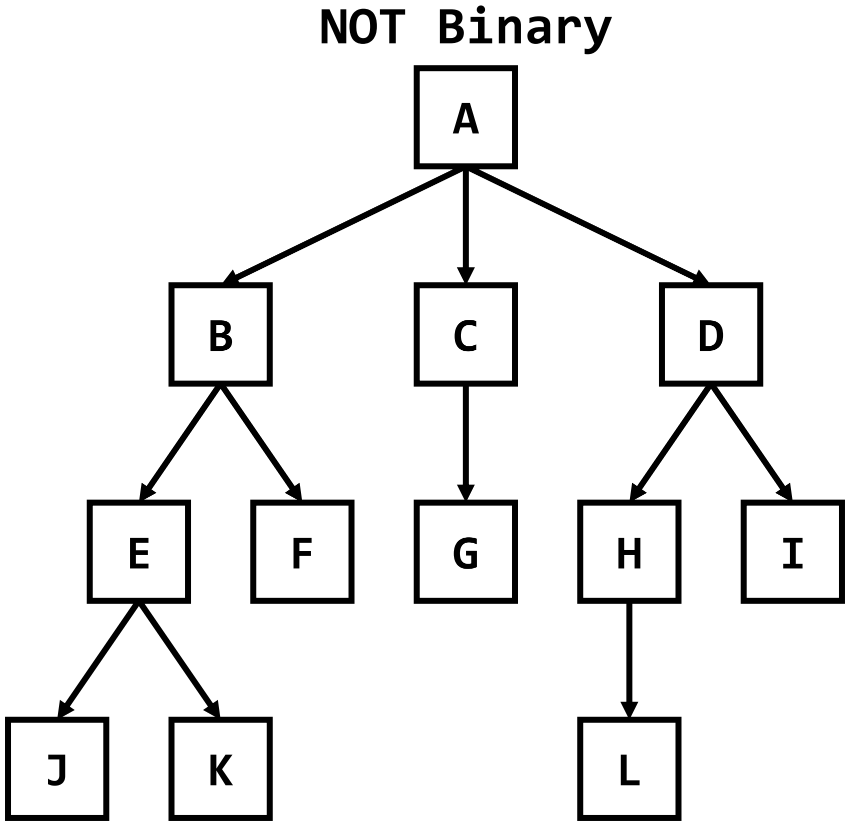
Subtree rooted at **node B**



(Recap) Binary Trees

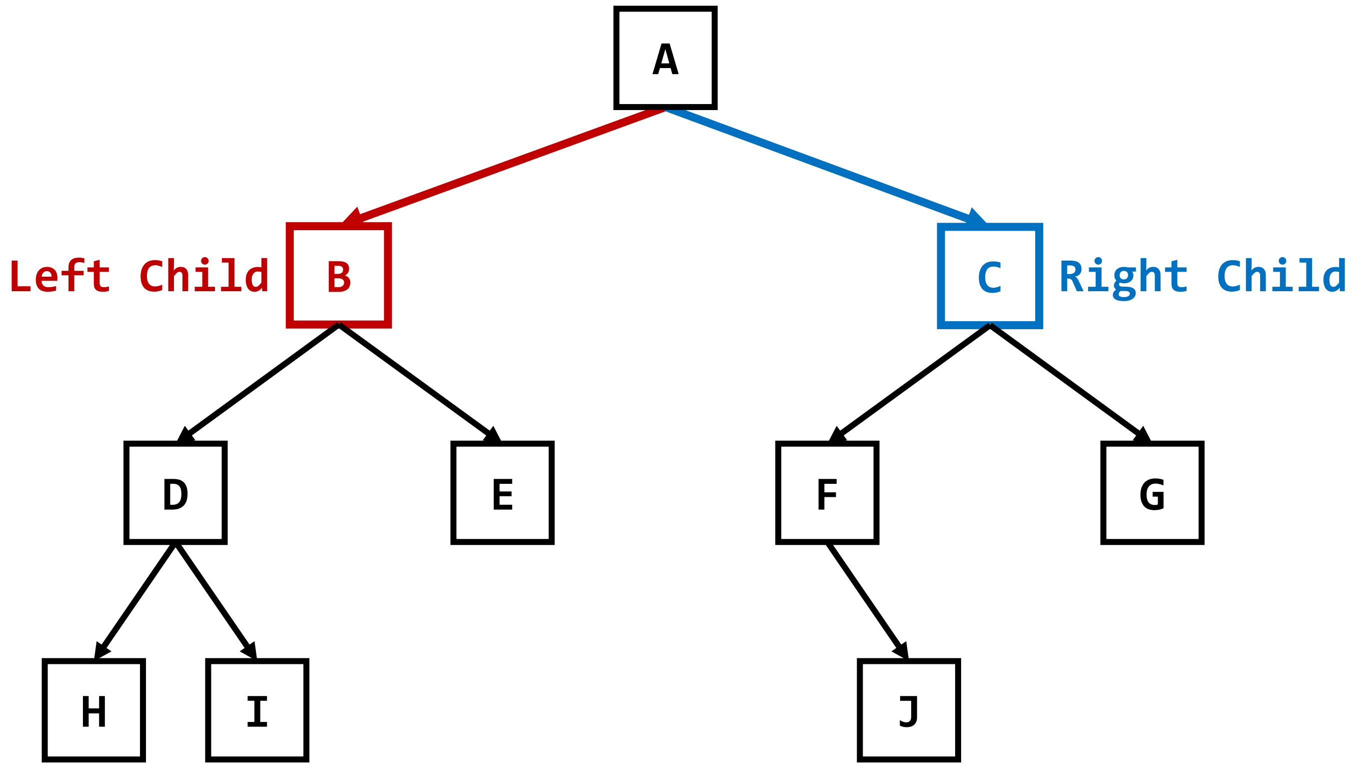
- **Binary Tree** is a tree in which each node has at most two children
  - `degree(X)` ≤ 2 for any `node` **`X`** in a binary tree





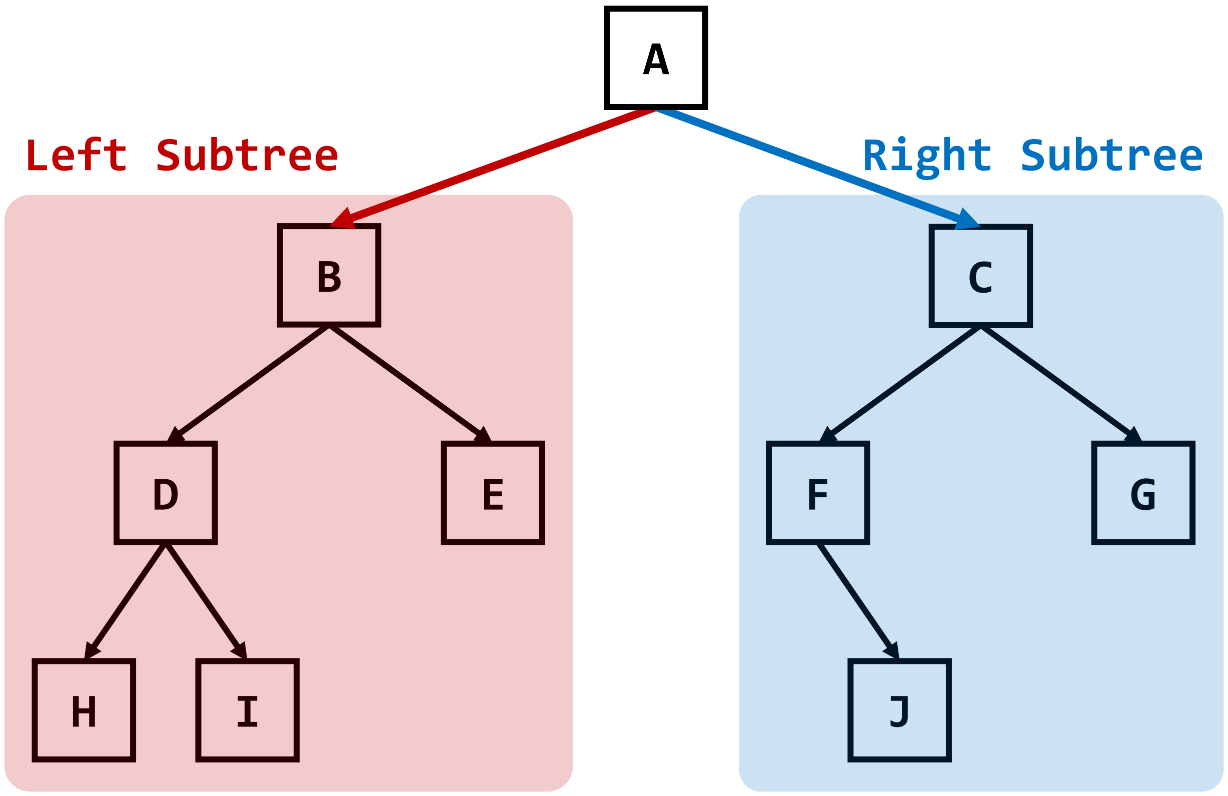
(Recap) Binary Trees

- **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



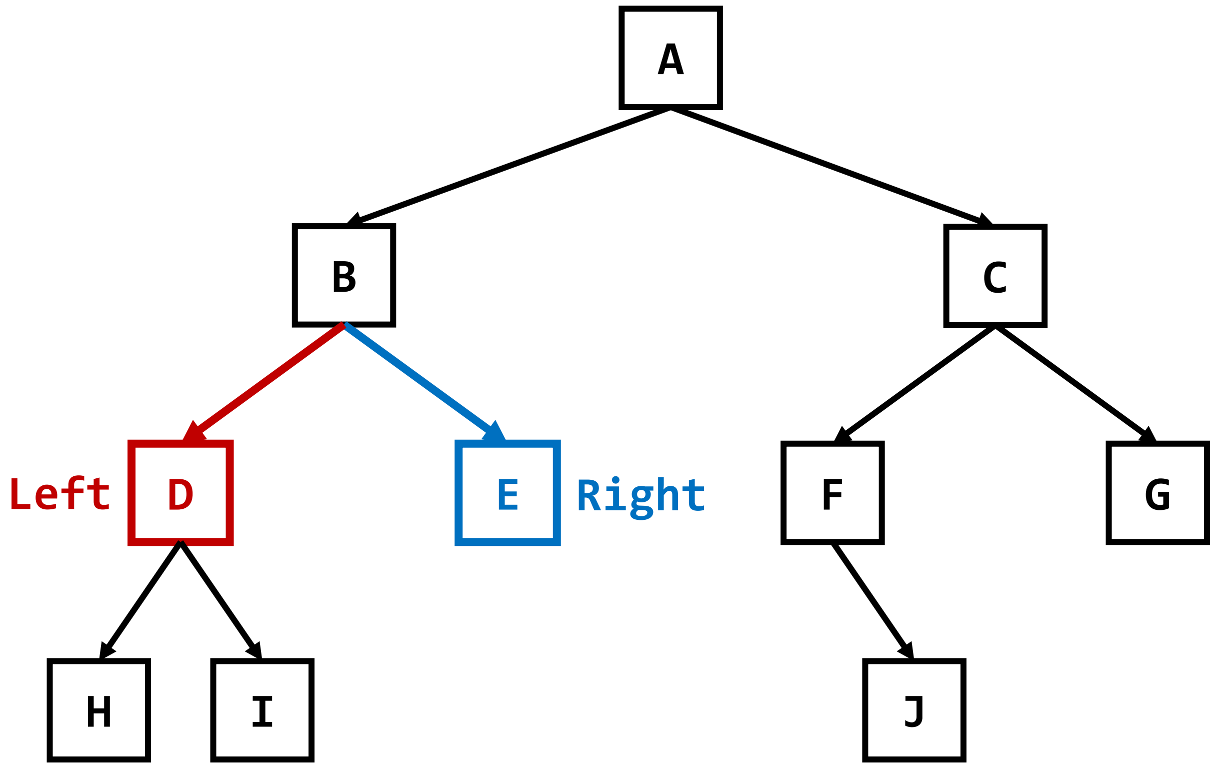
(Recap) Binary Trees

- **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



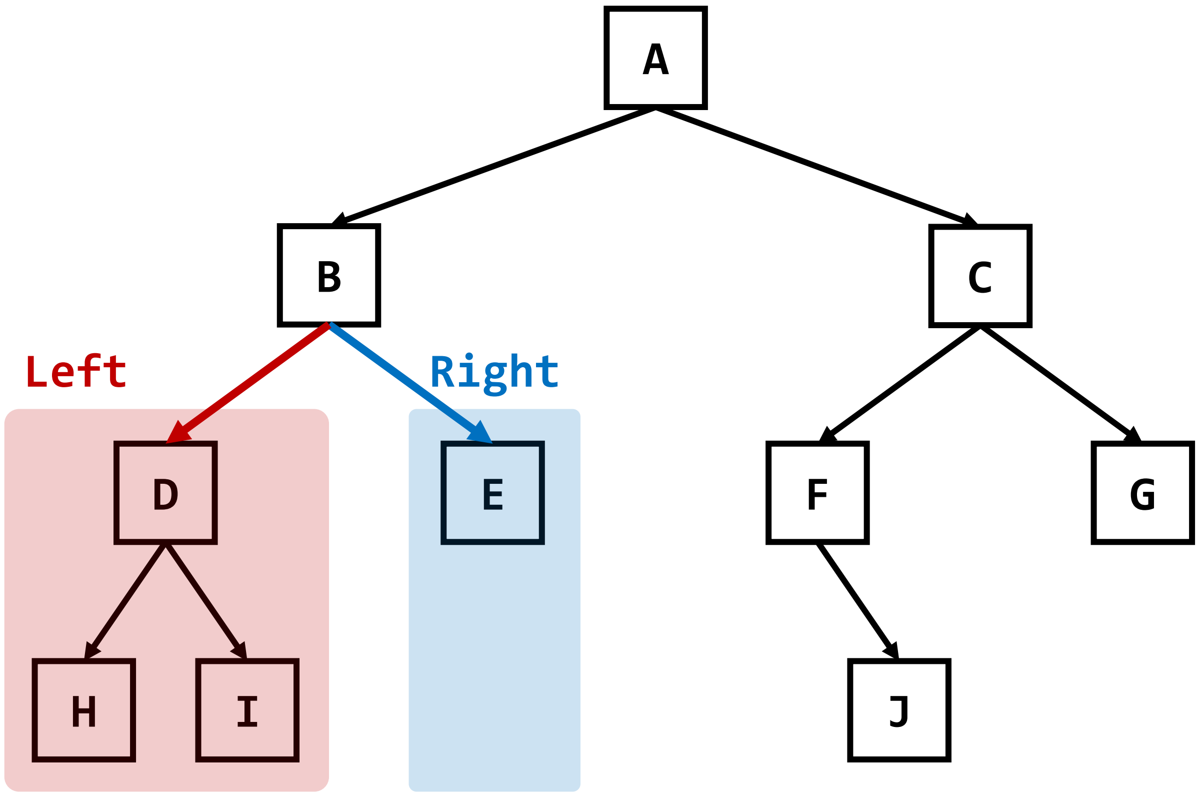
(Recap) Binary Trees

- **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 Tree Implementation

```c
typedef struct _Node {
    int item;
    struct _Node *left, *right;
} Node;
```

- In general, **the (linked-)list structure** is suitable for BT implementation
  - The tree is **non-linear** structure, which is not fit with the array structure
  - Since degree ≤ 2, the node structure can be easily implemented
  - Insertion and deletion are easier to implement

(Recap) Binary Tree Implementation

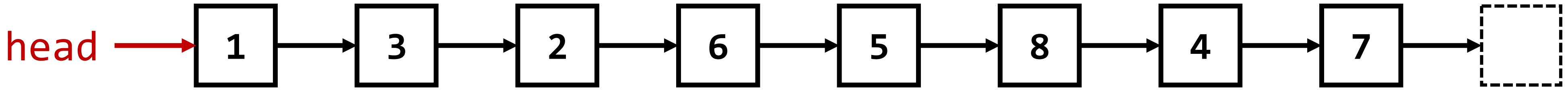
```
typedef struct _Node {
    int item;
    struct _Node *left, *right, *parent;
} Node;
```

- In general, **the (linked-)list structure** is suitable for BT implementation
  - The tree is **non-linear** structure, which is not fit with the array structure
  - Since degree ≤ 2, the node structure can be easily implemented
  - Insertion and deletion are easier to implement

  - The `parent node` pointer is sometimes useful for …
    - tree modification
    - traversal from bottom to top (`child → parent`)

Binary Search Trees (BSTs)

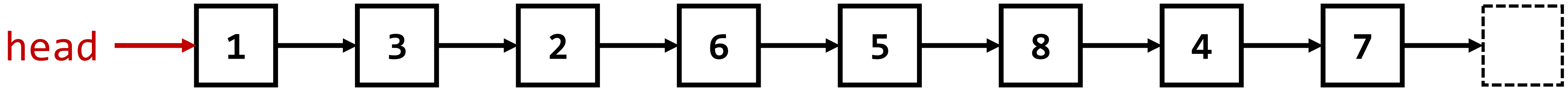
- A linked list is **inefficient in searching** an item



  - **(Q)** How to search the item of 5 in the above list?
  - **(Q)** What is the time complexity of the search?

Binary Search Trees (BSTs)

- A linked list is **inefficient in searching** an item

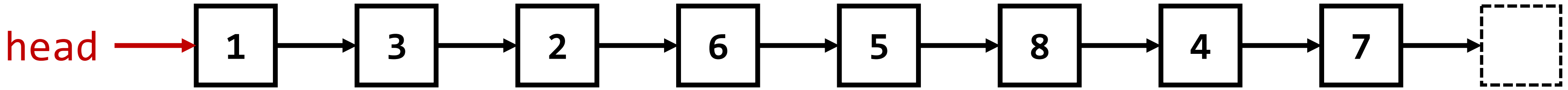


  - **(Q)** How to search the item of 5 in the above list?
  - **(A)** You must traverse the items from the first to the last sequentially

  - **(Q)** What is the time complexity of the search?
  - **(A)** $O(N)$ where N is the number of items

Binary Search Trees (BSTs)

- A linked list is **inefficient in searching** an item



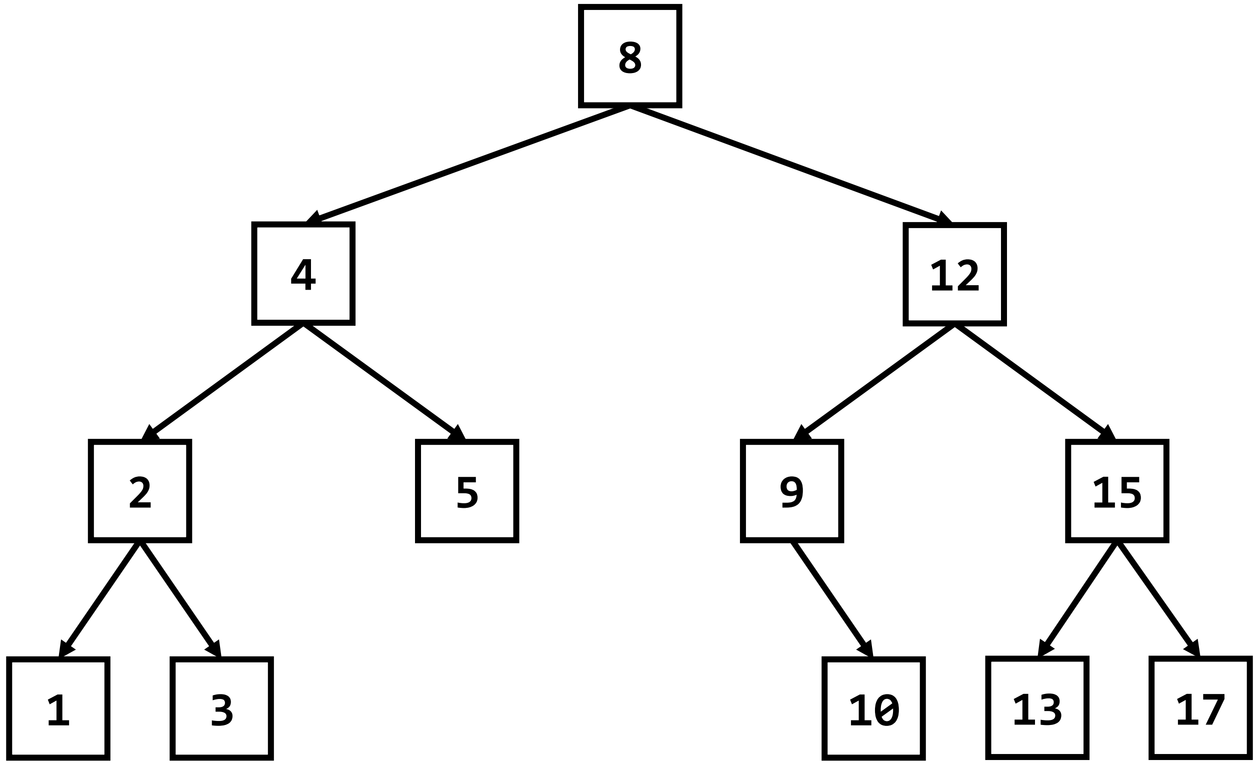
  - **(Q)** How to search the item of 5 in the above list?
  - **(A)** You must traverse the items from the first to the last sequentially

  - **(Q)** What is the time complexity of the search?
  - **(A)** $O(N)$ where N is the number of items

- **Binary Search Tree (BST)** is an **efficient tree structure** for search

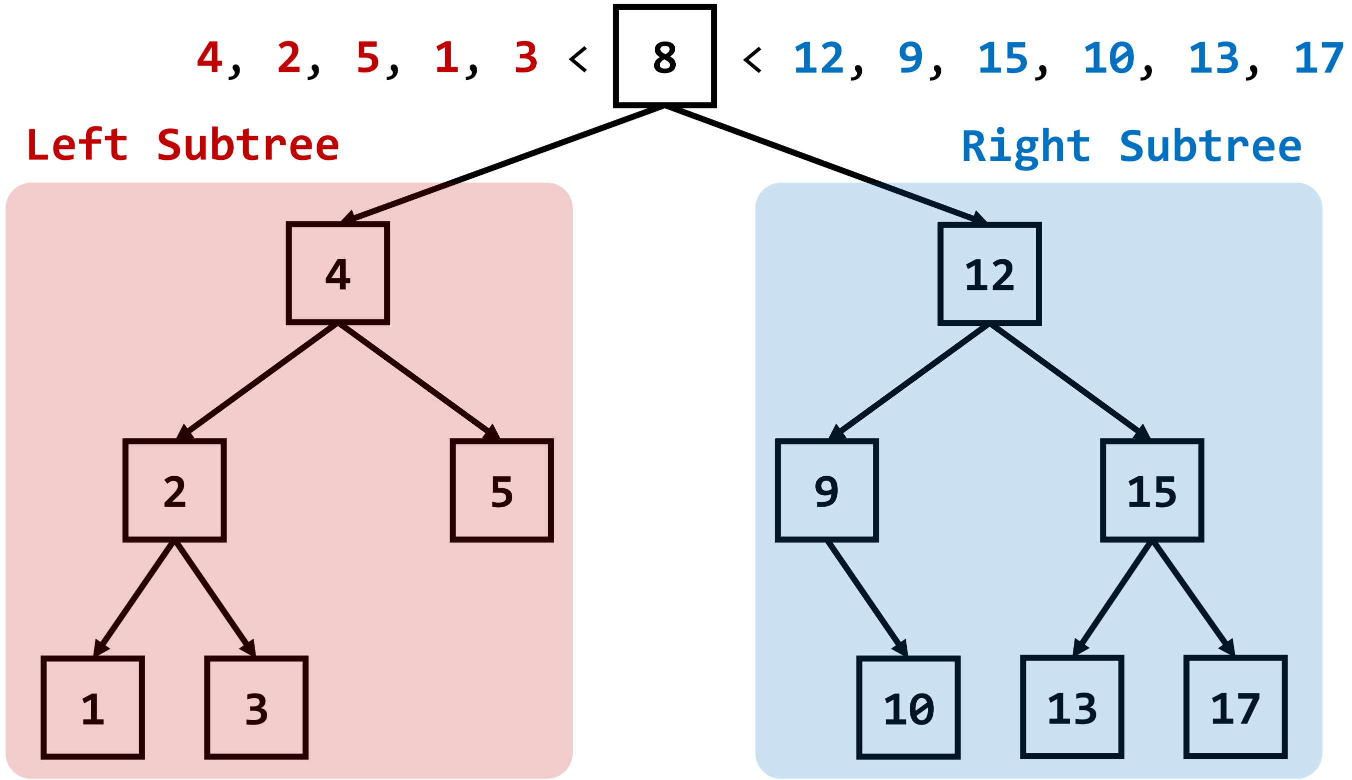
Binary Search Trees (BSTs)

- **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



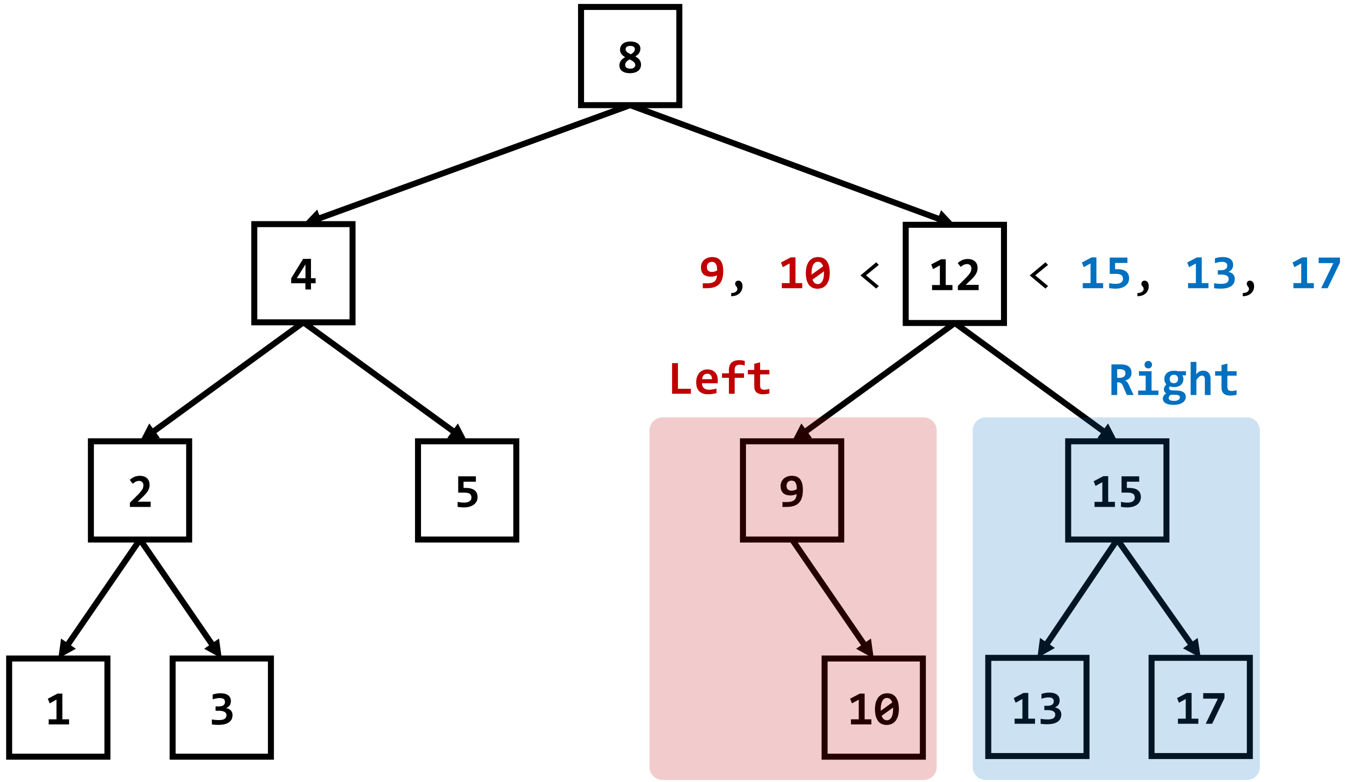
Binary Search Trees (BSTs)

- **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**



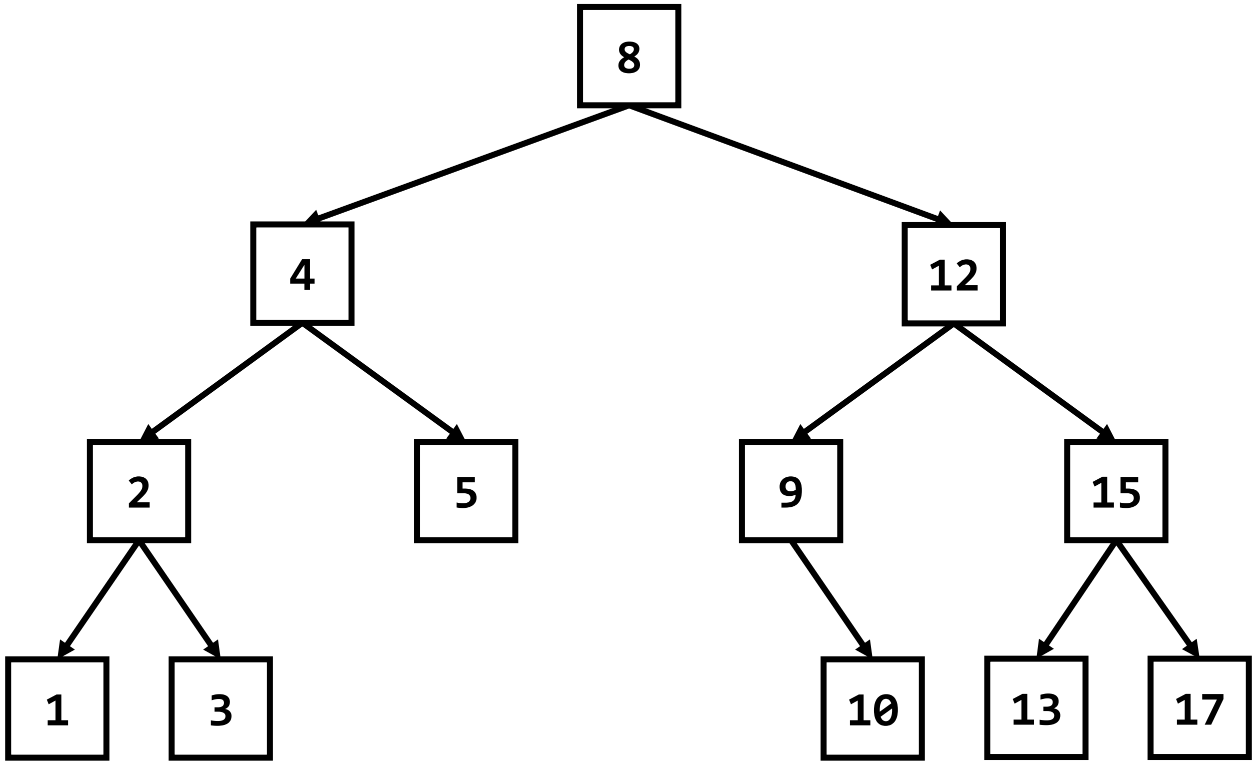
Binary Search Trees (BSTs)

- **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**



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**



BST Operations

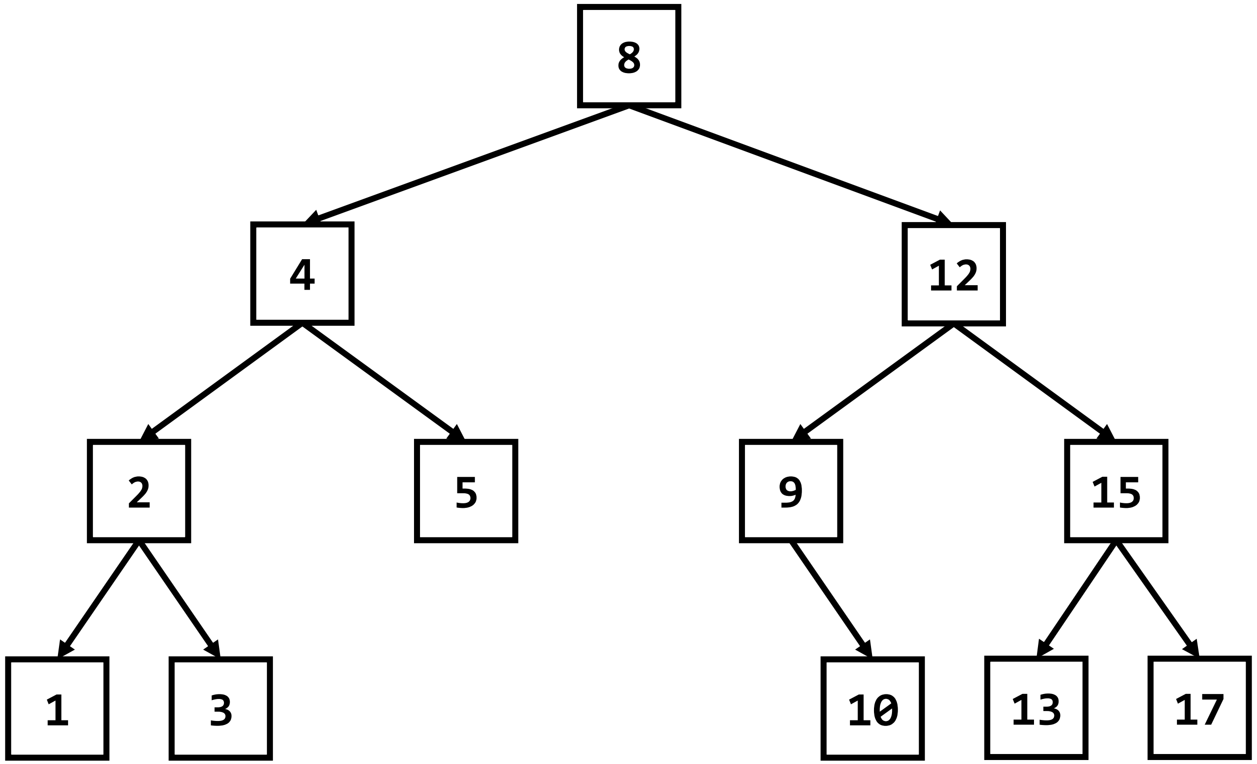
- **Validity** - check whether a binary tree is a binary search tree?
- **Search** - find the node of the target **KEY**
- **Insertion/Deletion** - insert/delete the node using **KEY**

```c
typedef struct _Node {
    int key;
    struct _Node *left, *right;
} Node;

Node* createLeaf(int key); // Create a leaf node with key
void removeTree(Node *tree); // Delete the node and its all descendants
int computeHeight(Node *node); // Compute height of the subtree rooted at the node
void traverse(Node *node); // In-order traversal
bool isBST(Node *node, int min, int max); // Check the BST validity
Node* search(int key, Node *root); // This returns the specific node of the key
Node* insertNode(int key, Node *root); // This returns the root after insertion
Node* deleteNode(int key, Node *root); // This returns the root after deletion
```

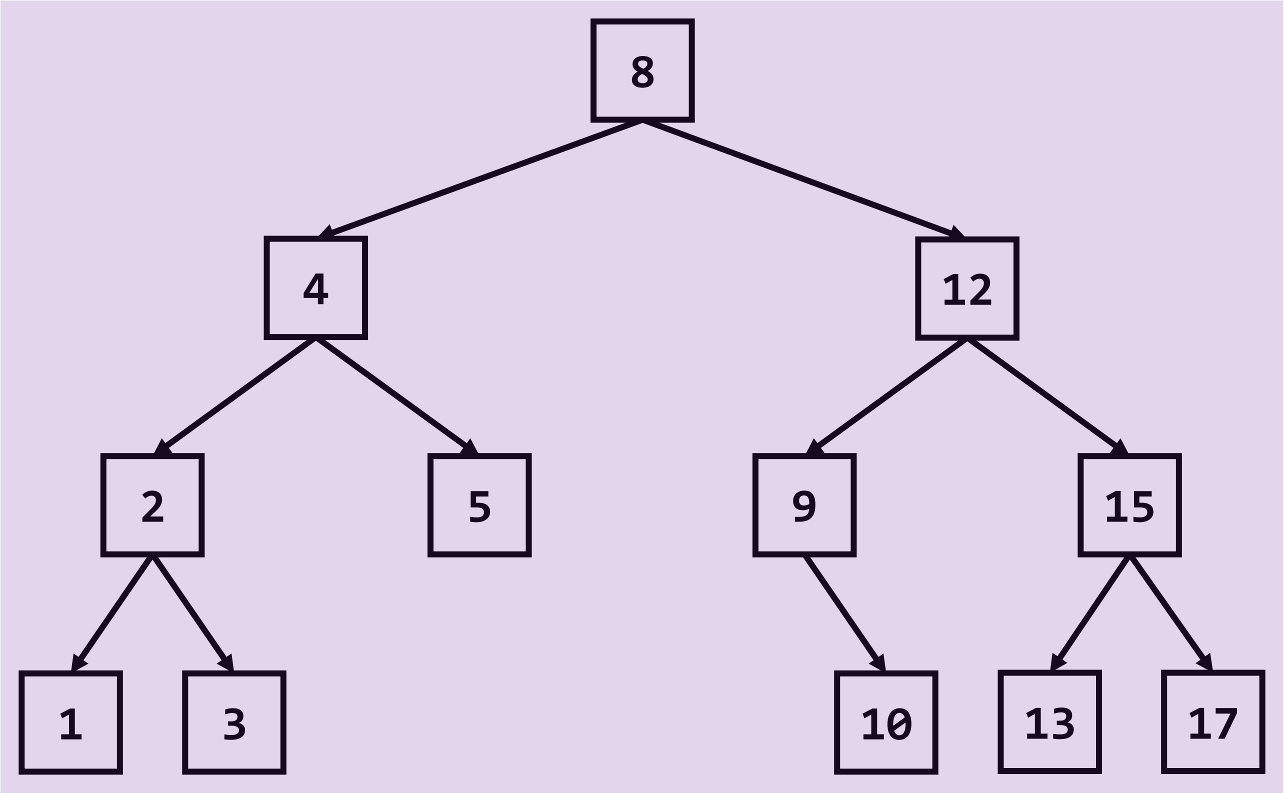
BST Operations - Validity

- How to check whether a binary tree is a binary search tree?
  - Let `(a,b)` is the interval between `a` (exclusive) and `b` (exclusive)
    - Formally, $(a, b) = \{ x \in \mathbb{R} \mid a < x < b \}$
  - **(Q)** What is the set of possible keys for each subtree?



BST Operations - Validity

- How to check whether a binary tree is a binary search tree?
  - Let `(a,b)` is the interval between `a` (exclusive) and `b` (exclusive)
    - Formally, $(a, b) = \{ x \in \mathbb{R} \mid a < x < b \}$
  - **(Q)** What is the set of possible keys for each subtree?

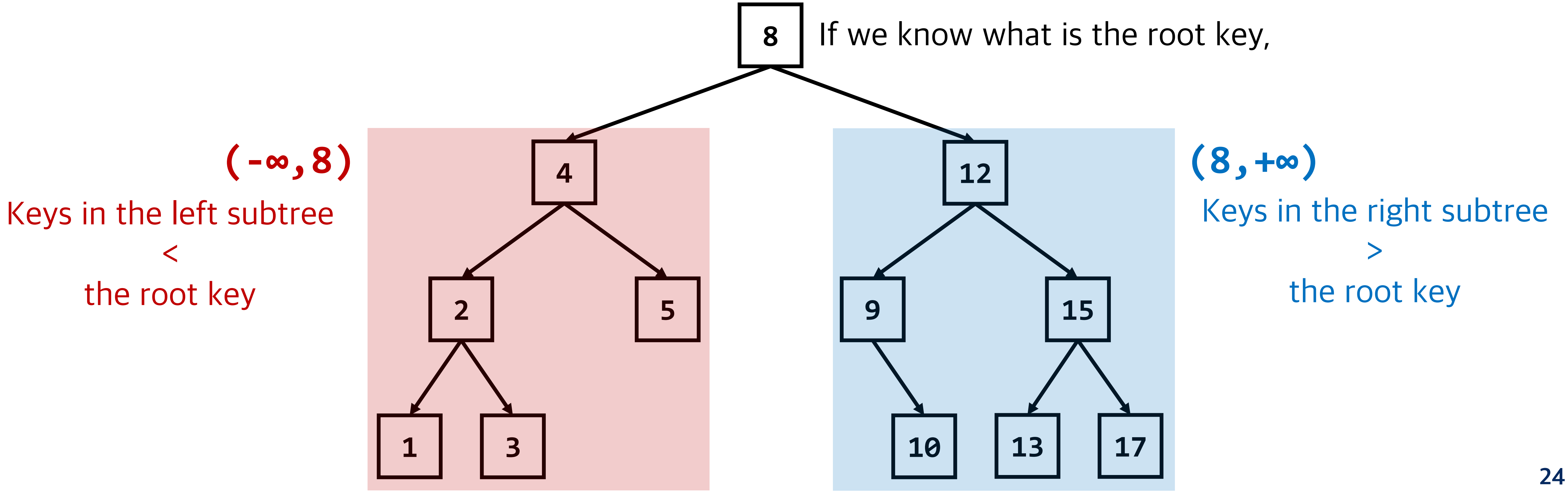


(-∞,+∞)

Any key can exist
in the entire tree

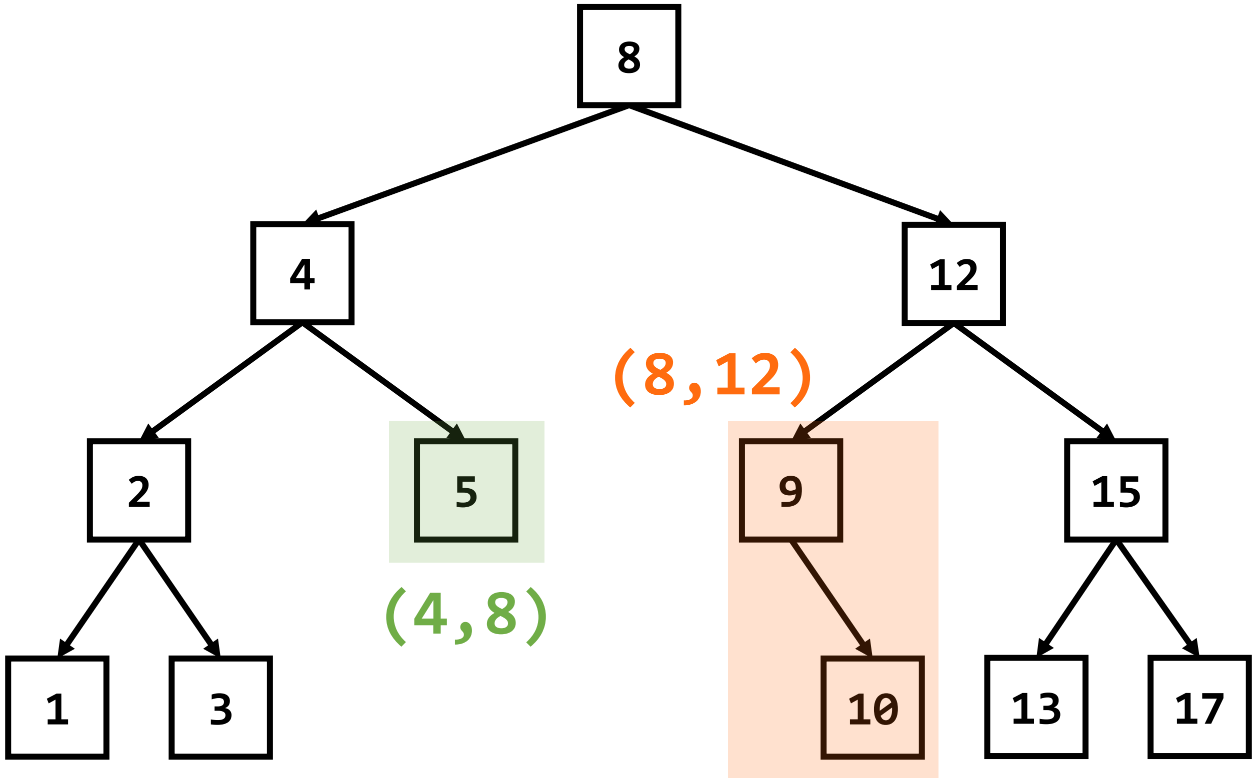
BST Operations - Validity

- How to check whether a binary tree is a binary search tree?
  - Let `(a,b)` is the interval between a (exclusive) and b (exclusive)
    - Formally, $(a, b) = \{\, x \in \mathbb{R} \mid a < x < b \,\}$
  - **(Q)** What is the set of possible keys for each subtree?



BST Operations - Validity

- How to check whether a binary tree is a binary search tree?
  - Let `(a,b)` is the interval between `a` (exclusive) and `b` (exclusive)
    - Formally, $(a, b) = \{\, x \in \mathbb{R} \mid a < x < b \,\}$
  - **(Q)** What is the set of possible keys for each subtree?



BST Operations - Validity

- How to check whether a binary tree is a binary search tree?
  - Let `(a,b)` is the interval between `a` (exclusive) and `b` (exclusive)
    - Formally, $(a, b) = \{\, x \in \mathbb{R} \mid a < x < b \,\}$

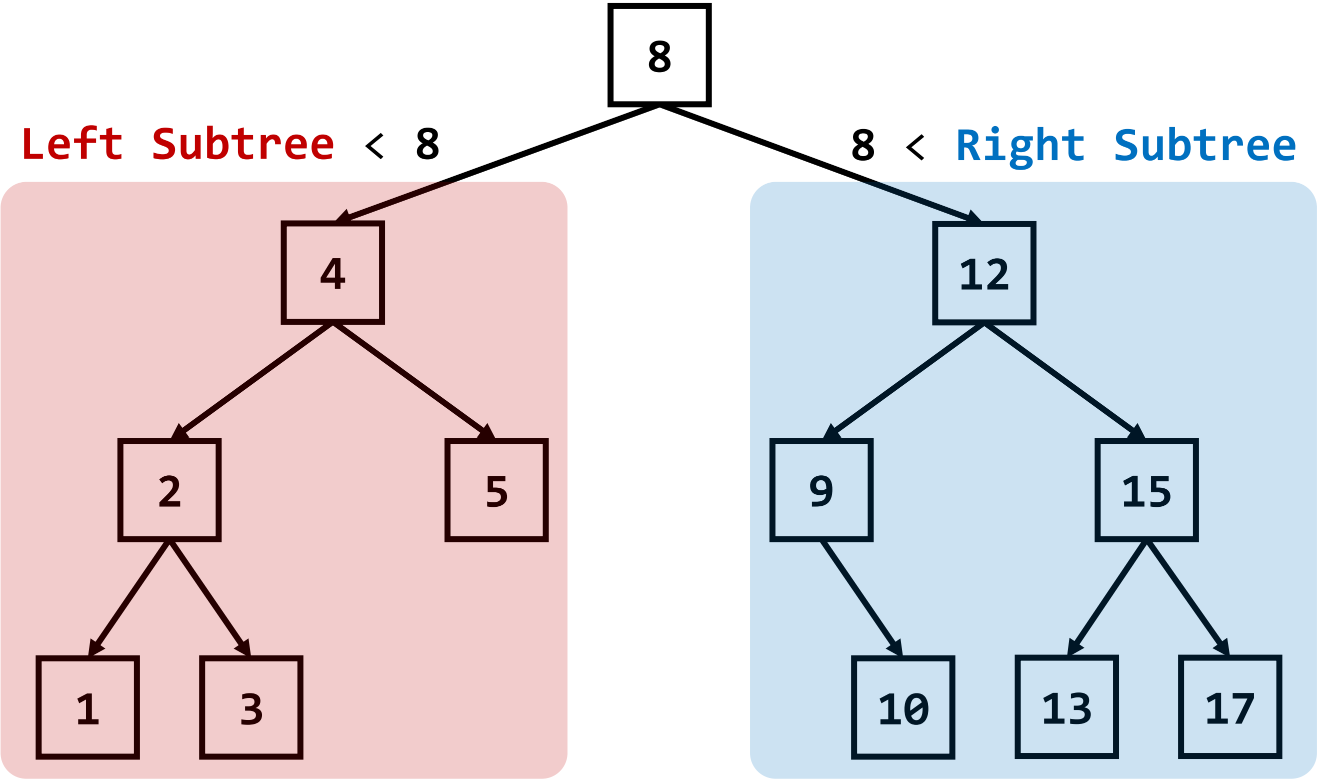
Algorithm: if `(min,max)` is the interval of possible keys of a tree,

1. Check whether its root key **K** is in the set
2. Check whether all keys in its left subtree is in an interval `(min,K)`
3. Check whether all keys in its right subtree is in an interval `(K,max)`
4. If 1 ~ 3 steps are passed, the tree satisfies `min` < `left` < `root` < `right` < `max`

- You can check whether a tree is a BST starting from the root with **`(-∞,+∞)`**

BST Operations - Search

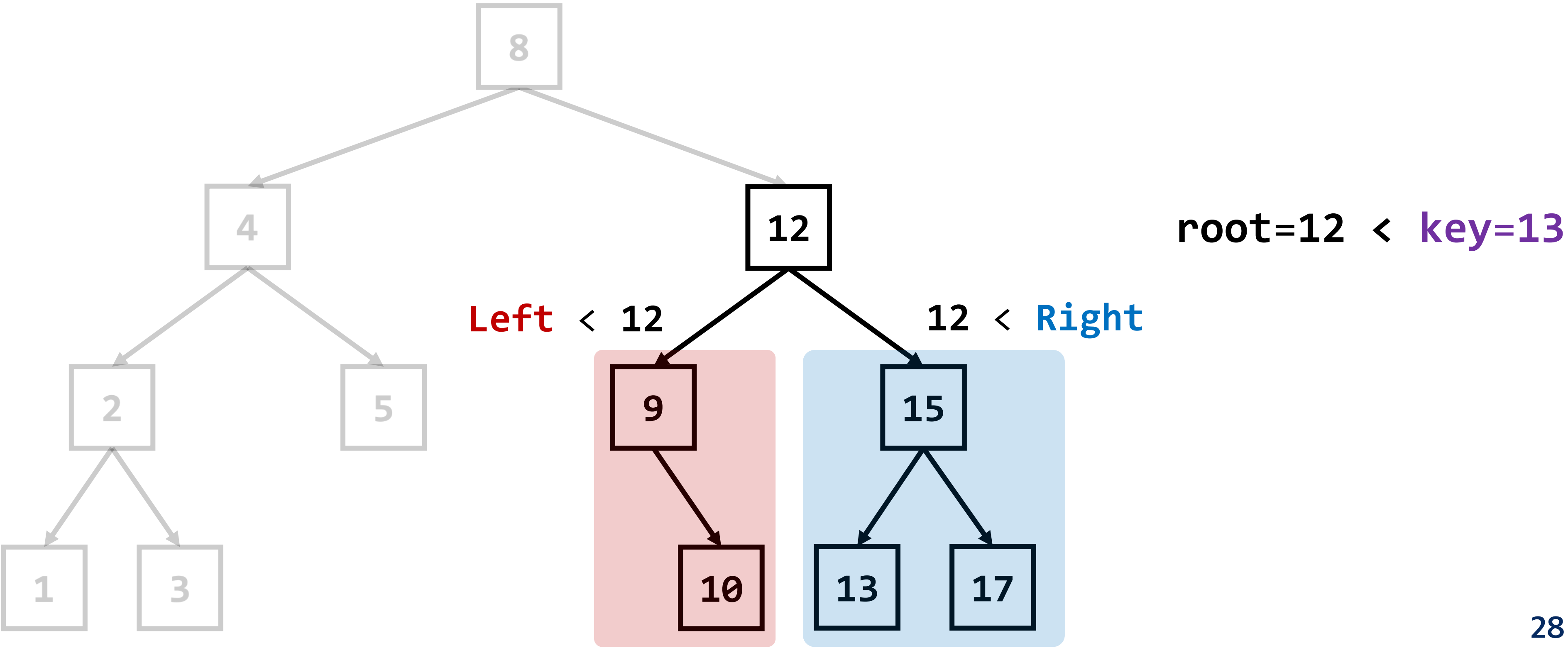
- How to find the specific node that has the target **key**?
  - Use the BST condition: **left subtree** < **root** < **right subtree**
  - **(Q)** Which subtree, left or right, does the node belong to?
    - Compare the target key with the root key



root=8 < key=13

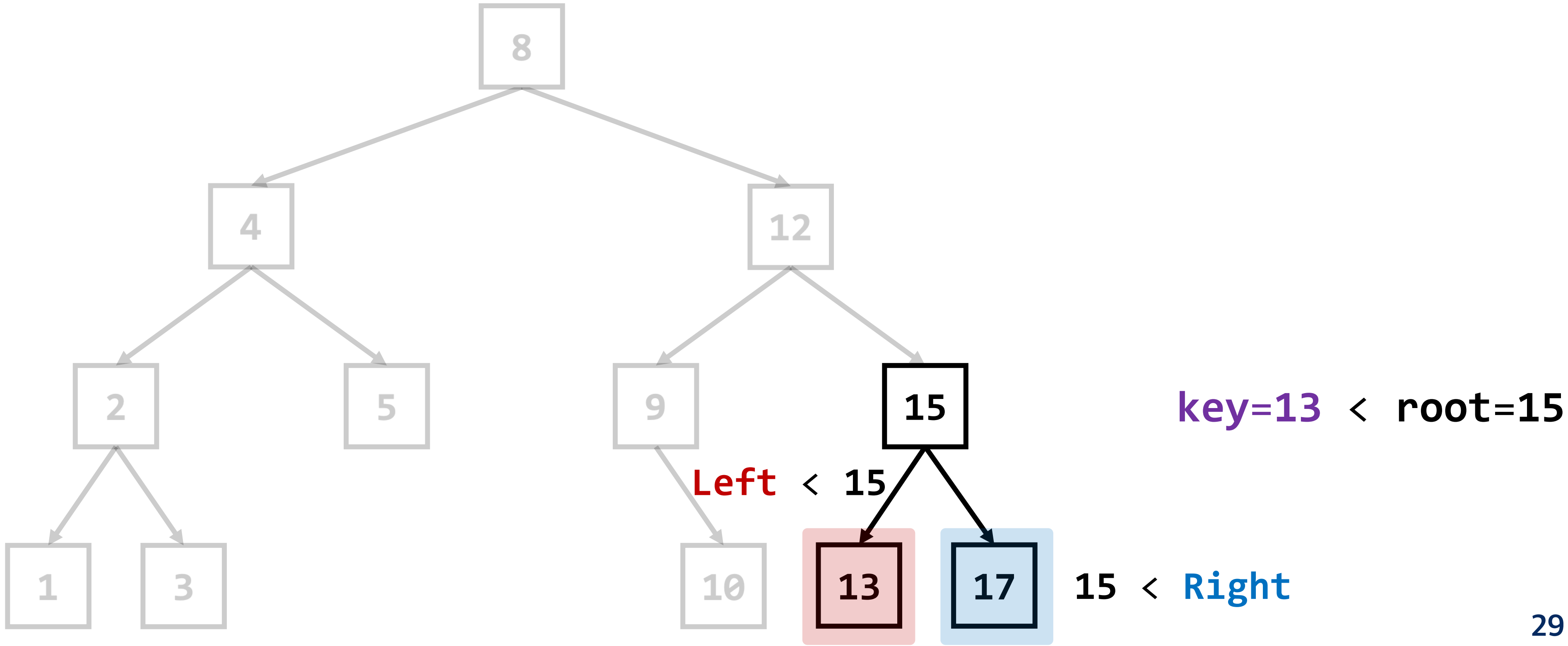
BST Operations - Search

- How to find the specific node that has the target **key**?
  - Use the BST condition: **left subtree** < **root** < **right subtree**
  - **(Q)** Which subtree, left or right, does the node belong to?
    - Compare the target key with the root key



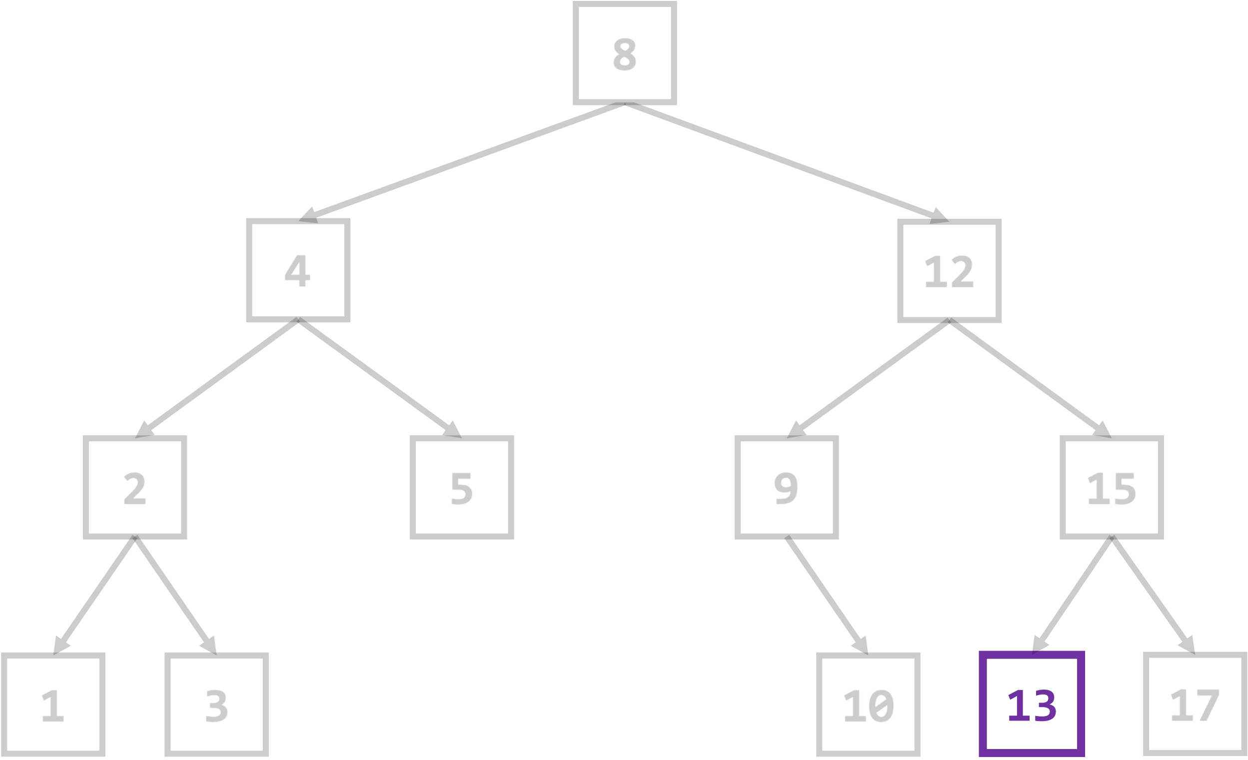
BST Operations - Search

- How to find the specific node that has the target **key**?
  - Use the BST condition: **left subtree** < **root** < **right subtree**
  - **(Q)** Which subtree, left or right, does the node belong to?
    - Compare the target key with the root key



BST Operations - Search

- How to find the specific node that has the target **key**?
  - Use the BST condition: **left subtree** < **root** < **right subtree**
  - **(Q)** Which subtree, left or right, does the node belong to?
    - Compare the target key with the root key



key=13 = root=13

BST Operations - Search

- How to find the specific node that has the target `key`?
  - Use the BST condition: `left subtree` < `root` < `right subtree`

Algorithm: Compare the `target key` with the **`root key`** recursively

1. If they are equal, the root node is what we find
2. If `target key` < **`root key`**, find the node in the `left subtree`
3. If **`root key`** < `target key`, find the node in the `right subtree`

- `(Q)` What is the time complexity of this search algorithm?

BST Operations - Search

- How to find the specific node that has the target `key`?
  - Use the BST condition: `left subtree` < `root` < `right subtree`

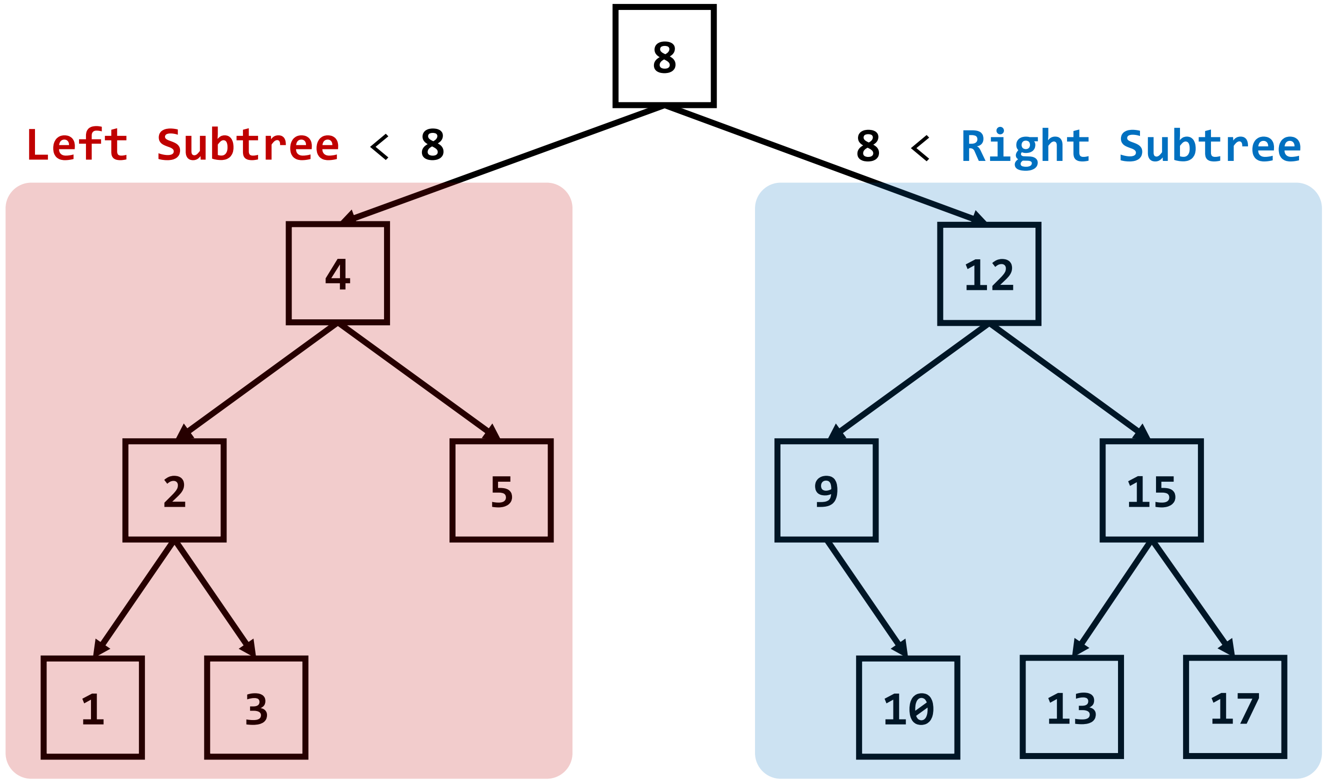
Algorithm: Compare the `target key` with the `root key` recursively

1. If they are equal, the root node is what we find
2. If `target key` < `root key`, find the node in the `left subtree`
3. If `root key` < `target key`, find the node in the `right subtree`

- **(Q)** What is the time complexity of this search algorithm?
- **(A)** The height of the binary search tree, i.e., $O(H)$

BST Operations - Insertion

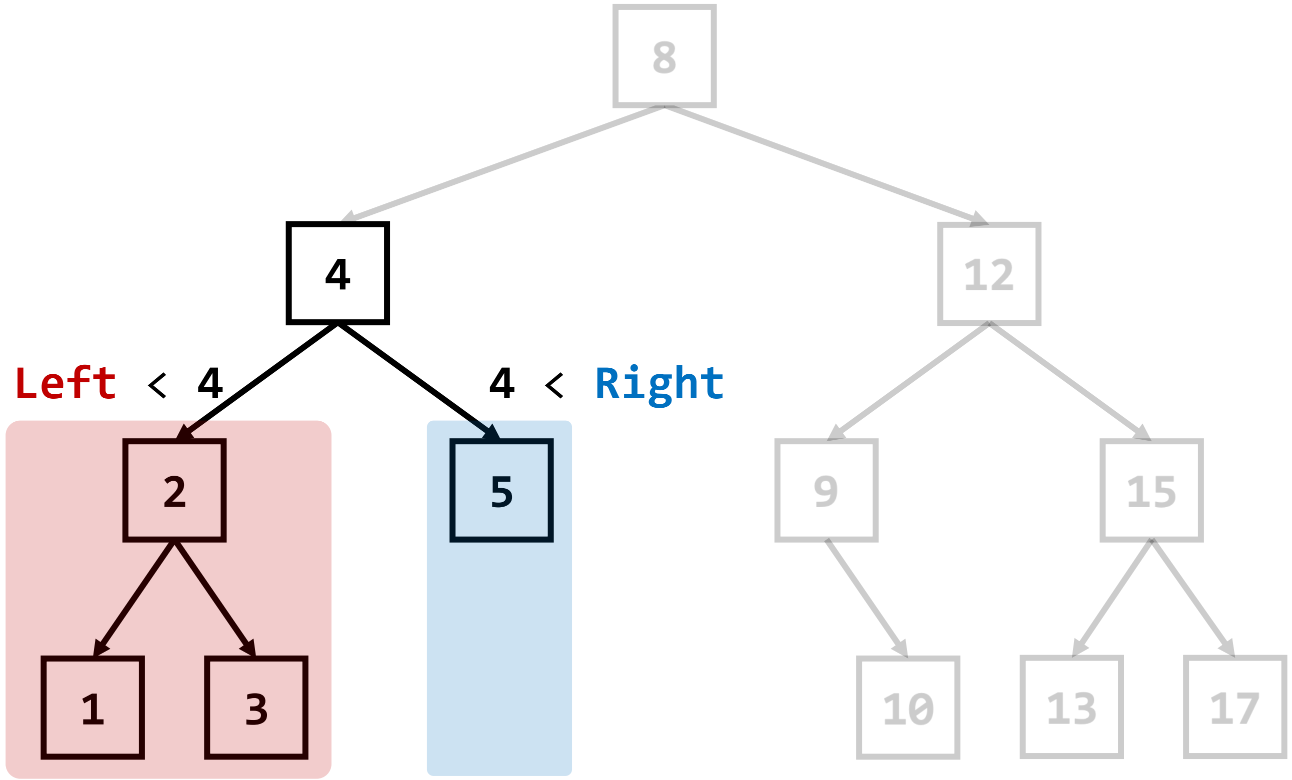
- How to insert a new node with the target **key**?
  - Use the BST condition: **left subtree** < **root** < **right subtree**
  - **(Q)** Which subtree, left or right, does the node belong to?
    - Compare the target key with the root key



key=7 < root=8

BST Operations - Insertion

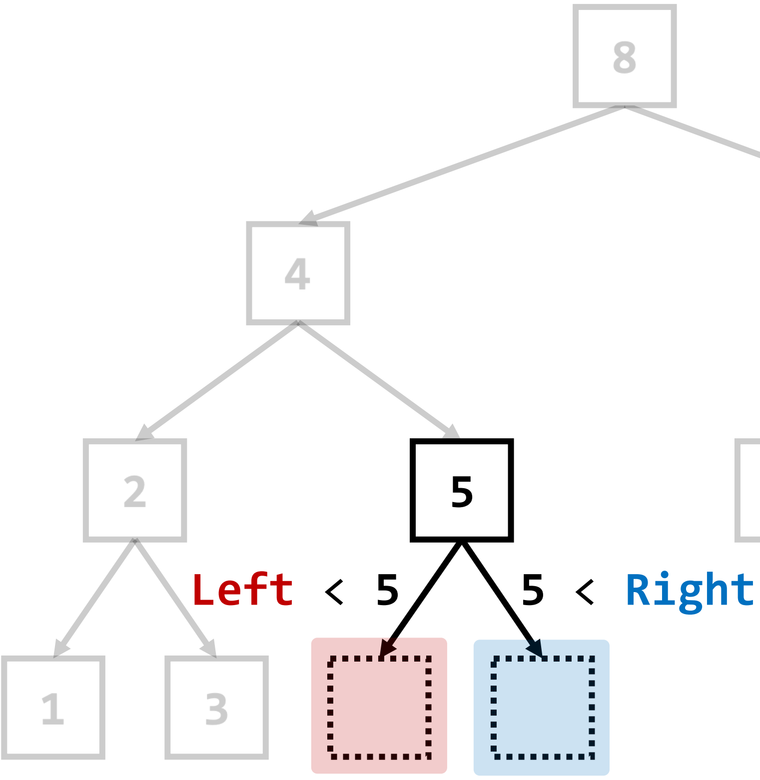
- How to insert a new node with the target **key**?
  - Use the BST condition: **left subtree** < **root** < **right subtree**
  - **(Q)** Which subtree, left or right, does the node belong to?
    - Compare the target key with the root key



root=4 < key=7

BST Operations - Insertion

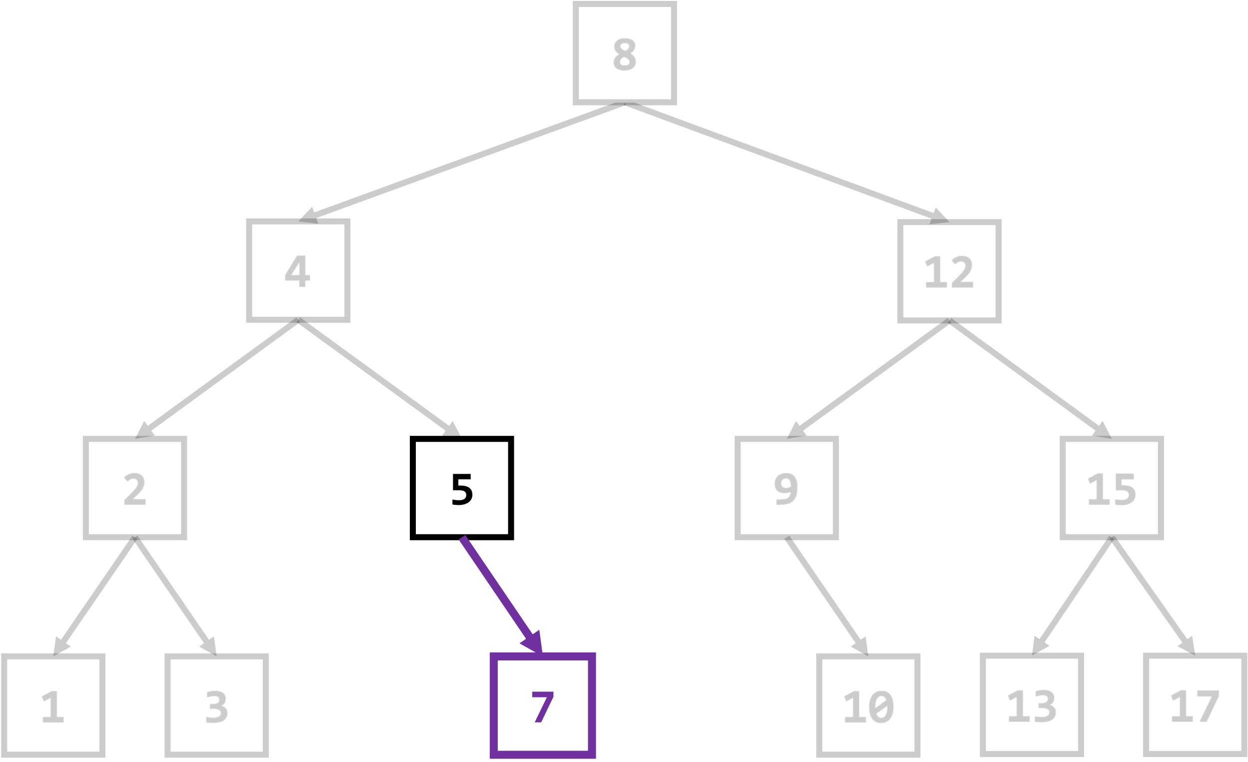
- How to insert a new node with the target **key**?
  - Use the BST condition: **left subtree** < **root** < **right subtree**
  - **(Q)** Which subtree, left or right, does the node belong to?
    - Compare the target key with the root key



root=5 < key=7

BST Operations - Insertion

- How to insert a new node with the target **key**?
  - Use the BST condition: **left subtree** < **root** < **right subtree**
  - **(Q)** Which subtree, left or right, does the node belong to?
    - Compare the target key with the root key



BST Operations - Insertion

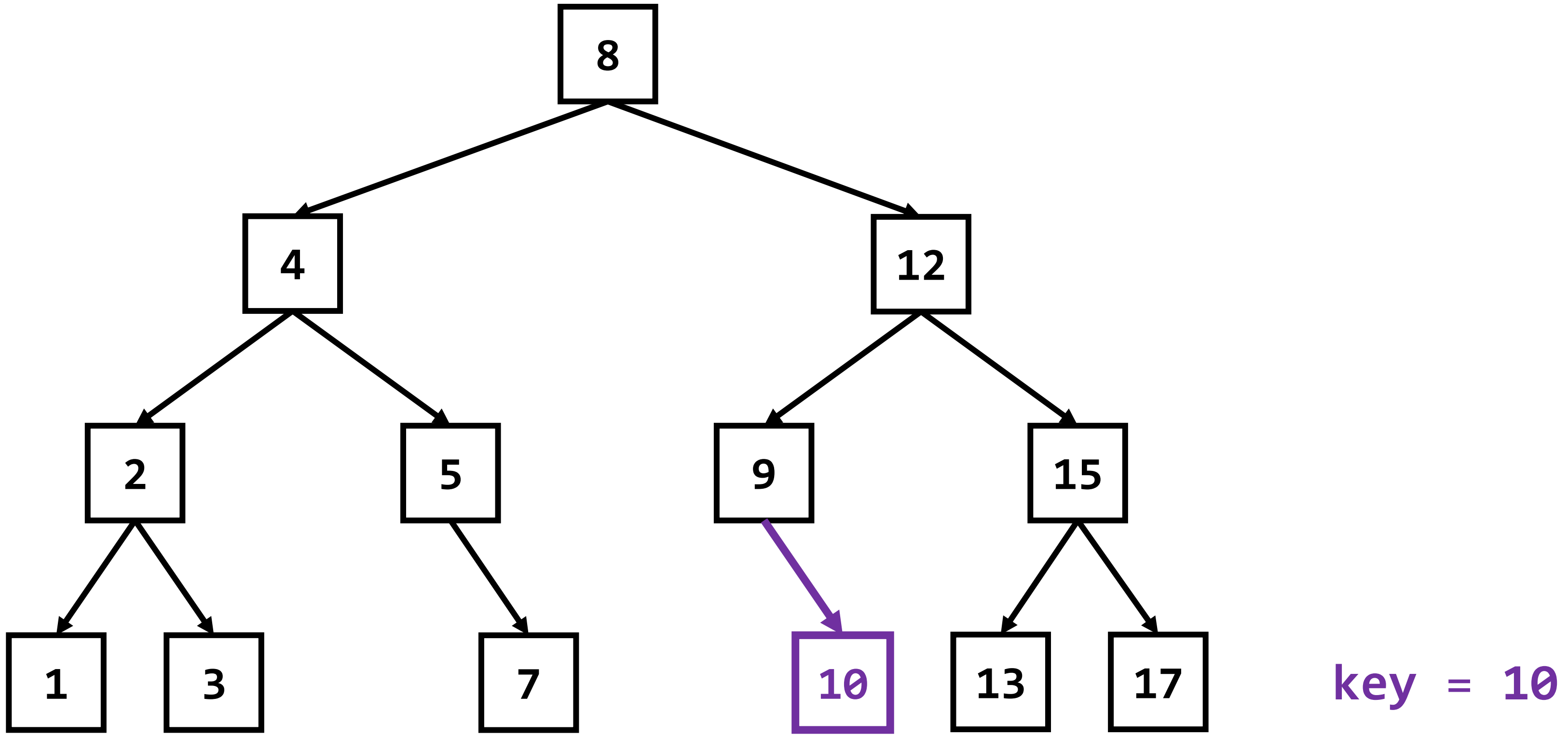
- How to insert a new node with the target **key**?
  - Use the BST condition: `left subtree` < **root** < `right subtree`

Algorithm: Compare the `target key` with the **root key** recursively

1. If `target key` < **root key**, find the insertion position in the `left subtree`
   - If the left subtree does not exist, insert the new node as the leaf child node
2. If **root key** < `target key`, find the insertion position in the `right subtree`
   - If the right subtree does not exist, insert the new node as the leaf child node

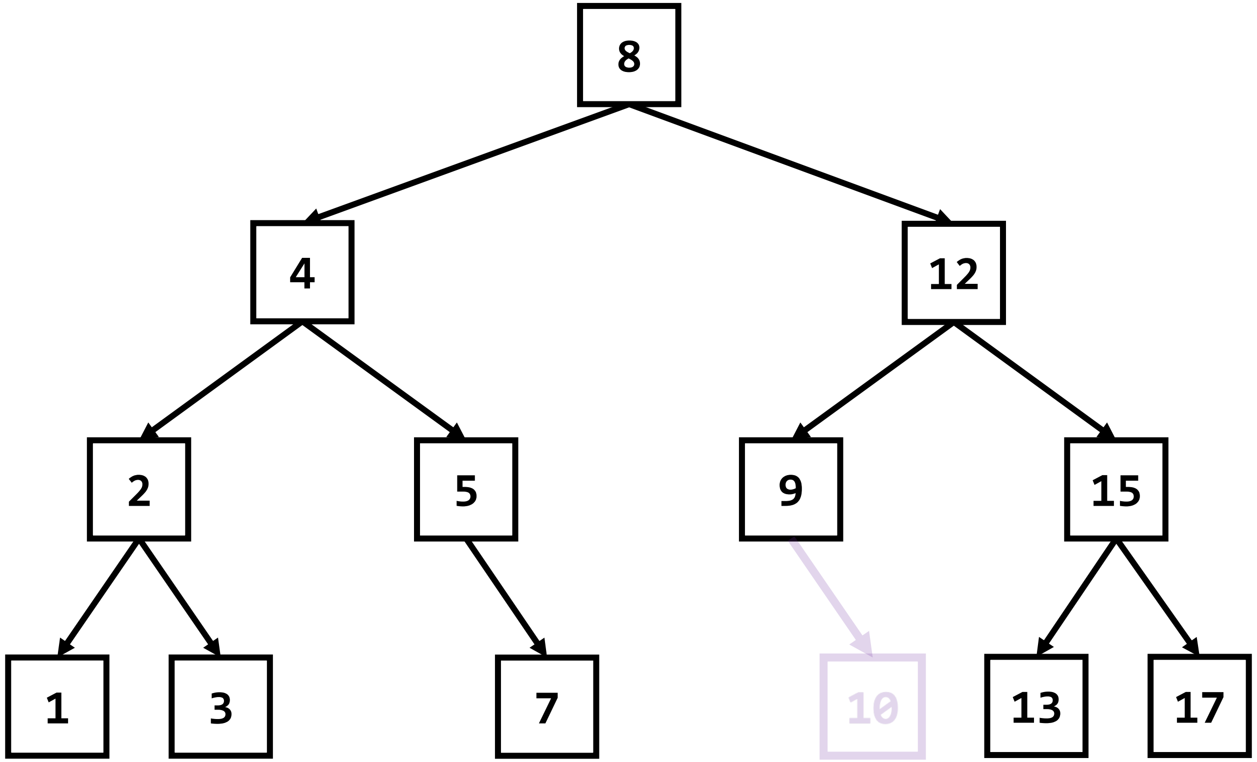
BST Operations - Deletion

- How to delete the node of the target **key** while satisfying BST conditions?
  (Case 1) If the node has no child, it can be simply deleted



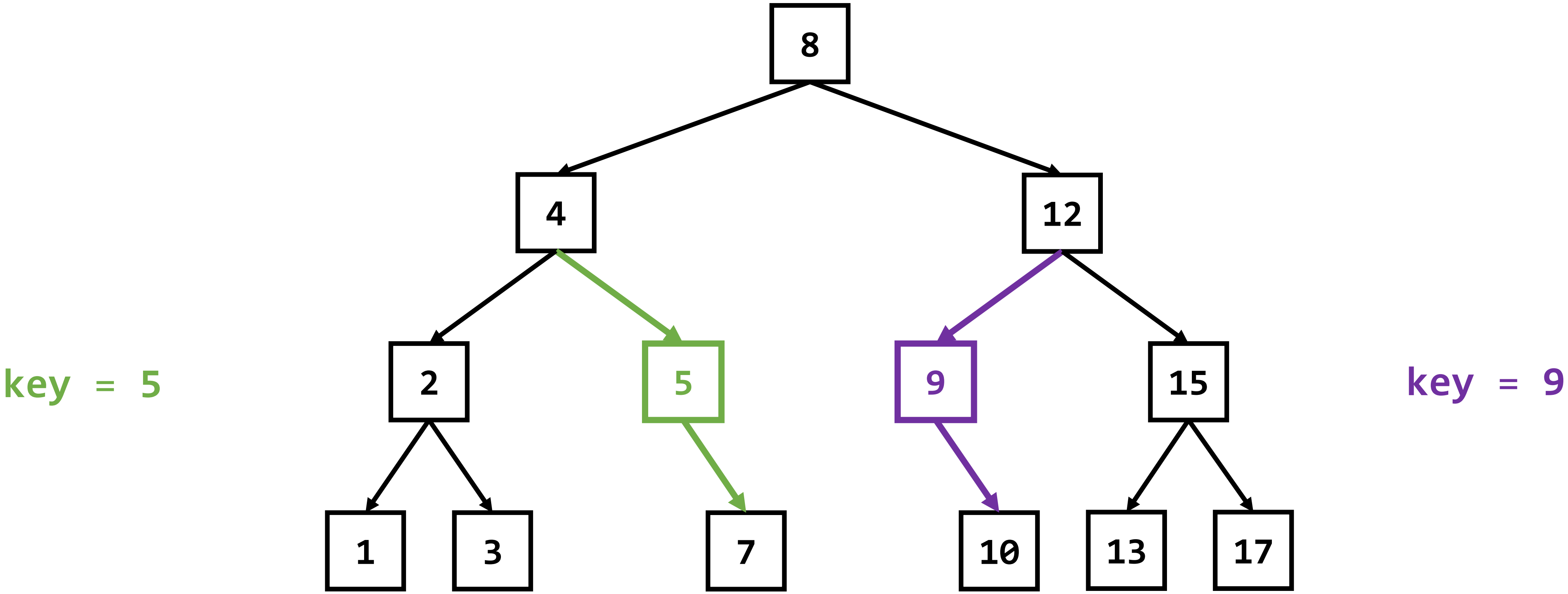
BST Operations - Deletion

- How to delete the node of the target **key** while satisfying BST conditions?
  - **(Case 1)** If the node has no child, it can be simply deleted



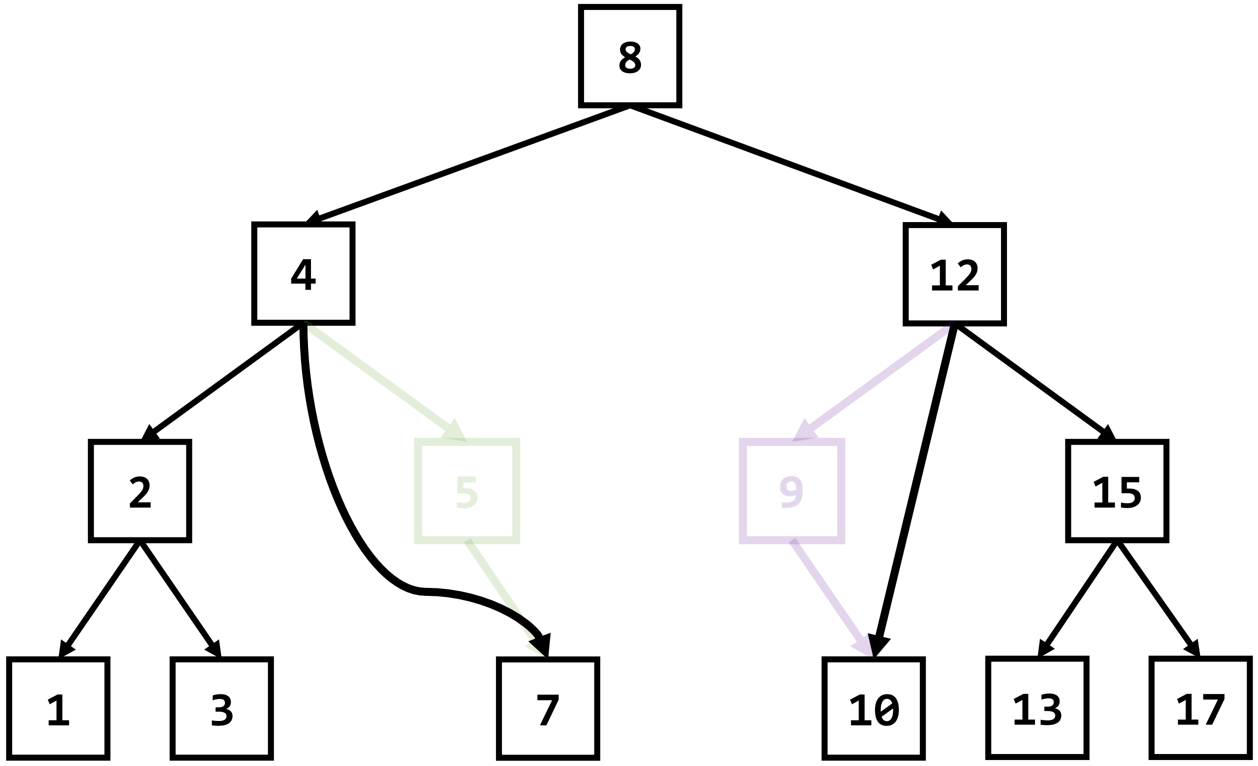
BST Operations - Deletion

- How to delete the node of the target **key** while satisfying BST conditions?
  (Case 2) If the node has one child, it can be deleted like the linked list structure



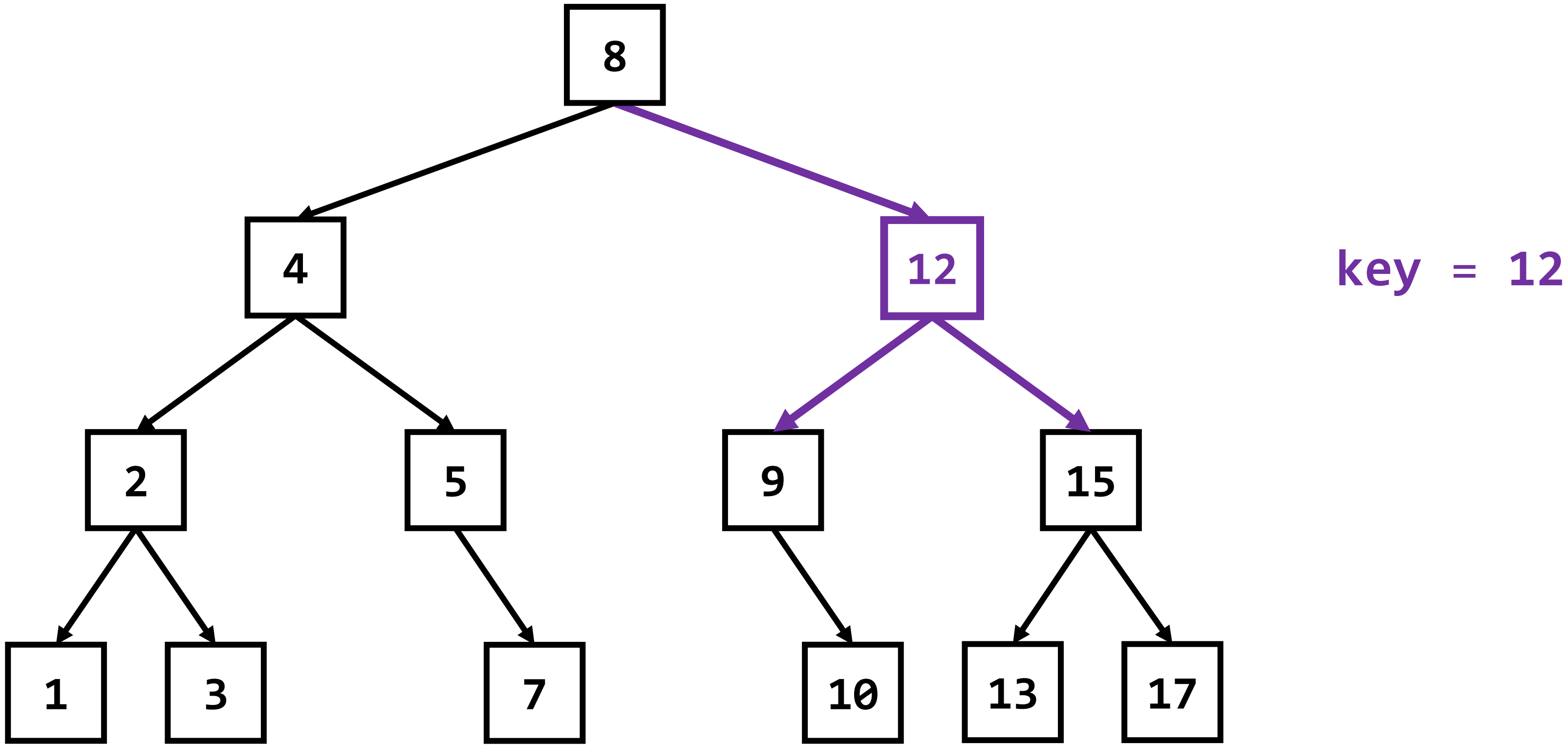
BST Operations - Deletion

- How to delete the node of the target **key** while satisfying BST conditions?
  - **(Case 2)** If the node has one child, it can be deleted like the linked list structure



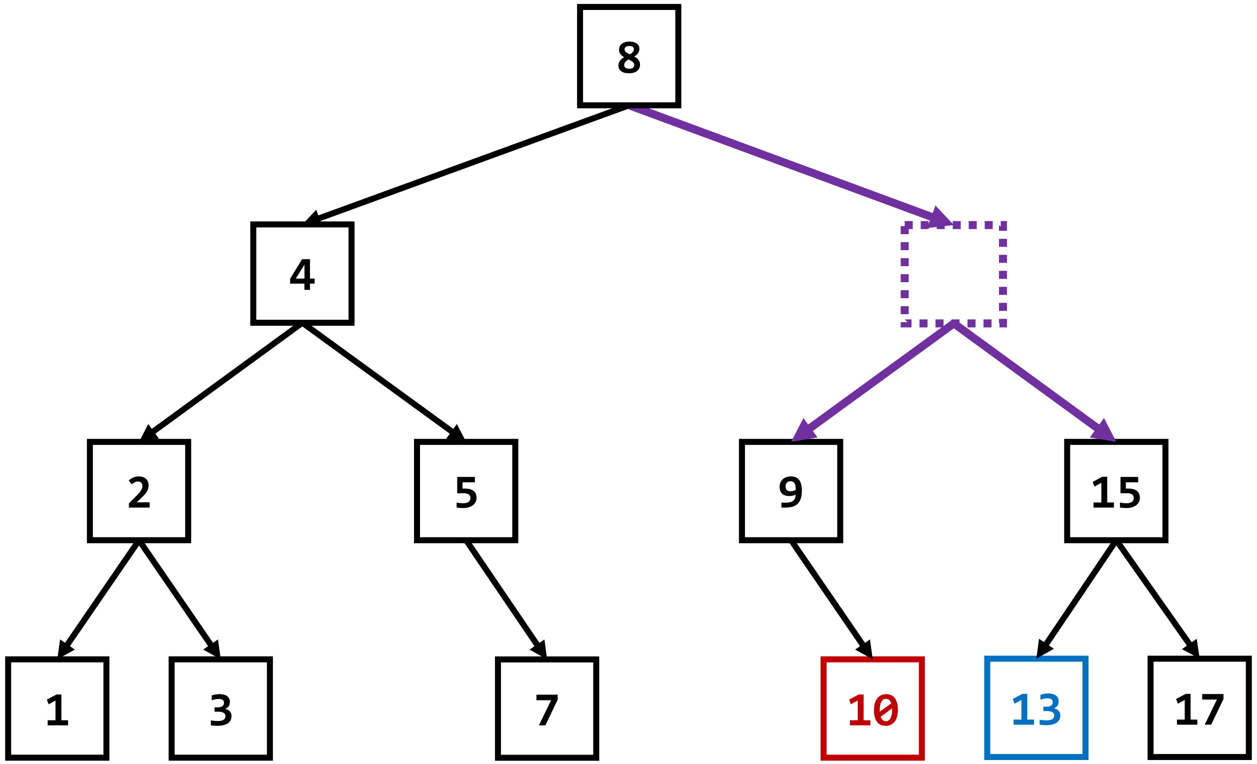
BST Operations - Deletion

- How to delete the node of the target **key** while satisfying BST conditions?
  (Case 3) If the node has two children, must find a replacement node



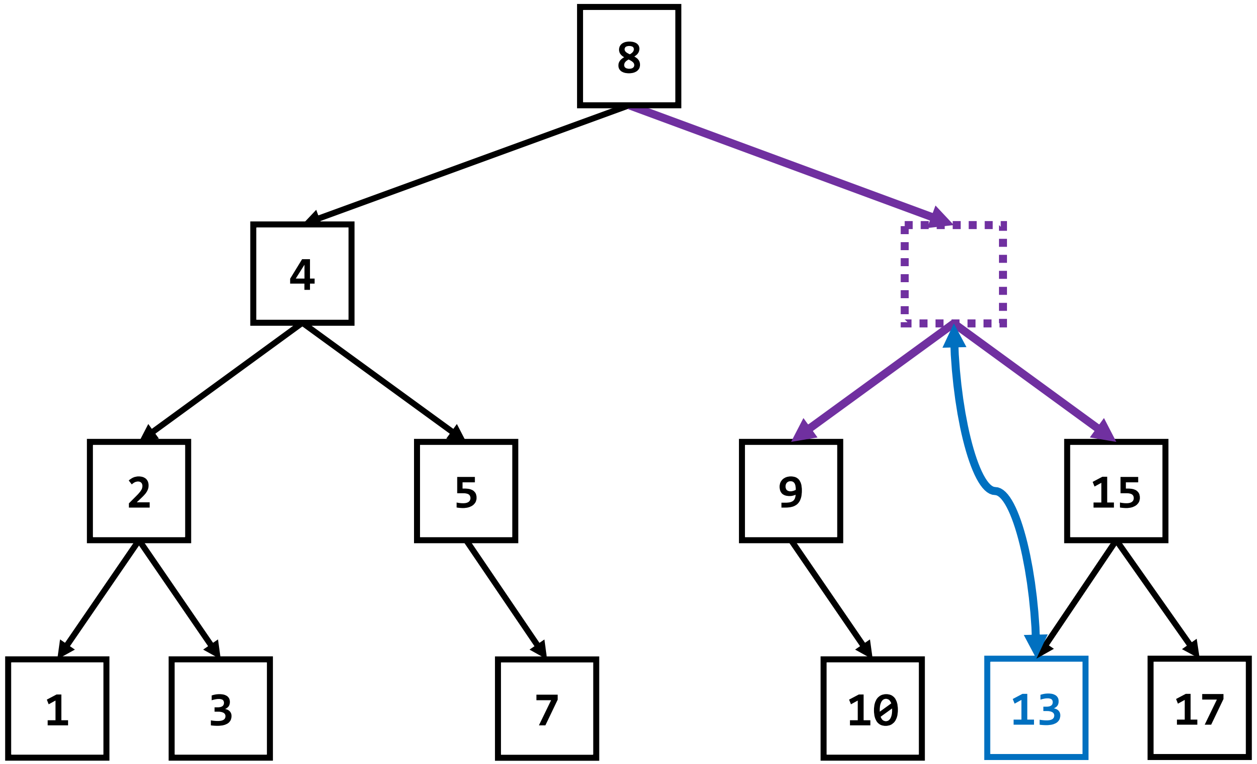
BST Operations - Deletion

- How to delete the node of the target **key** while satisfying BST conditions?
  (Case 3) If the node has two children, must find a replacement node
  - **In-order predecessor** : the largest (right-most) node in the left subtree
  - **In-order successor** : the smallest (left-most) node in the right subtree



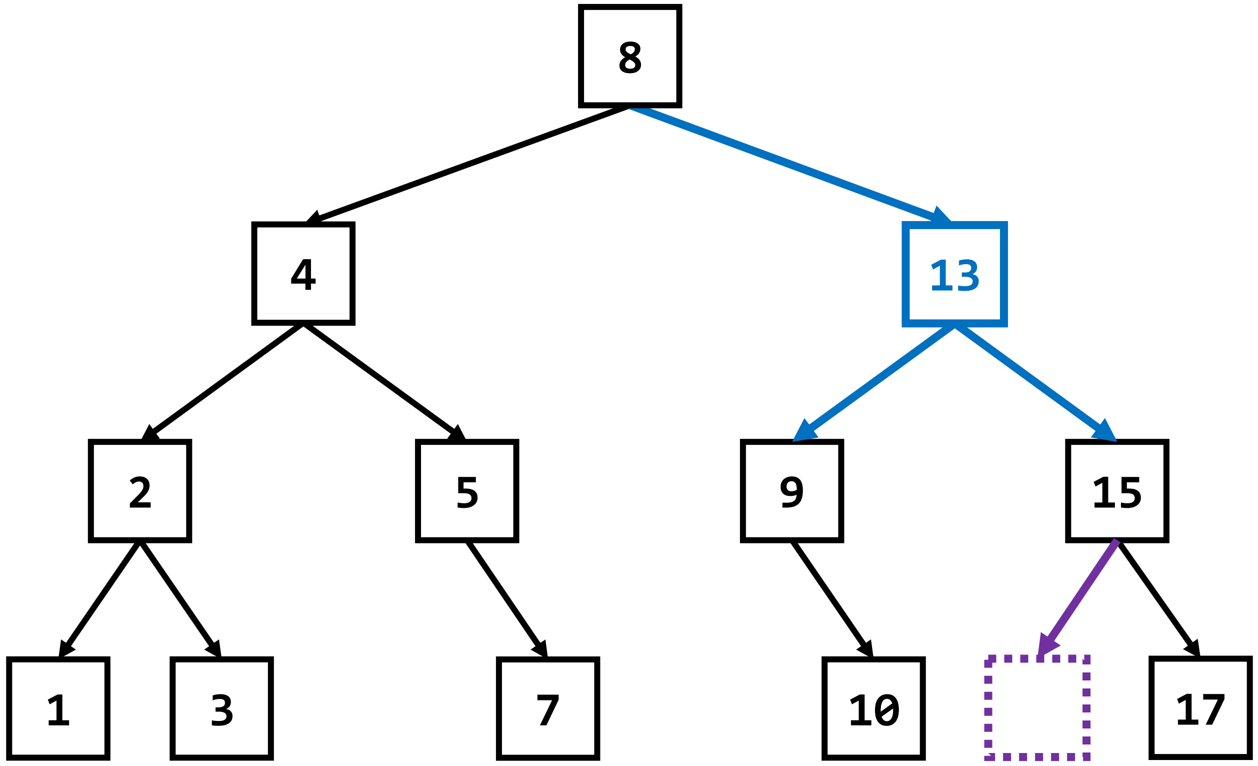
BST Operations - Deletion

- How to delete the node of the target **key** while satisfying BST conditions?
  (Case 3) If the node has two children, must find a replacement node
    - **In-order successor** : the smallest (left-most) node in the right subtree



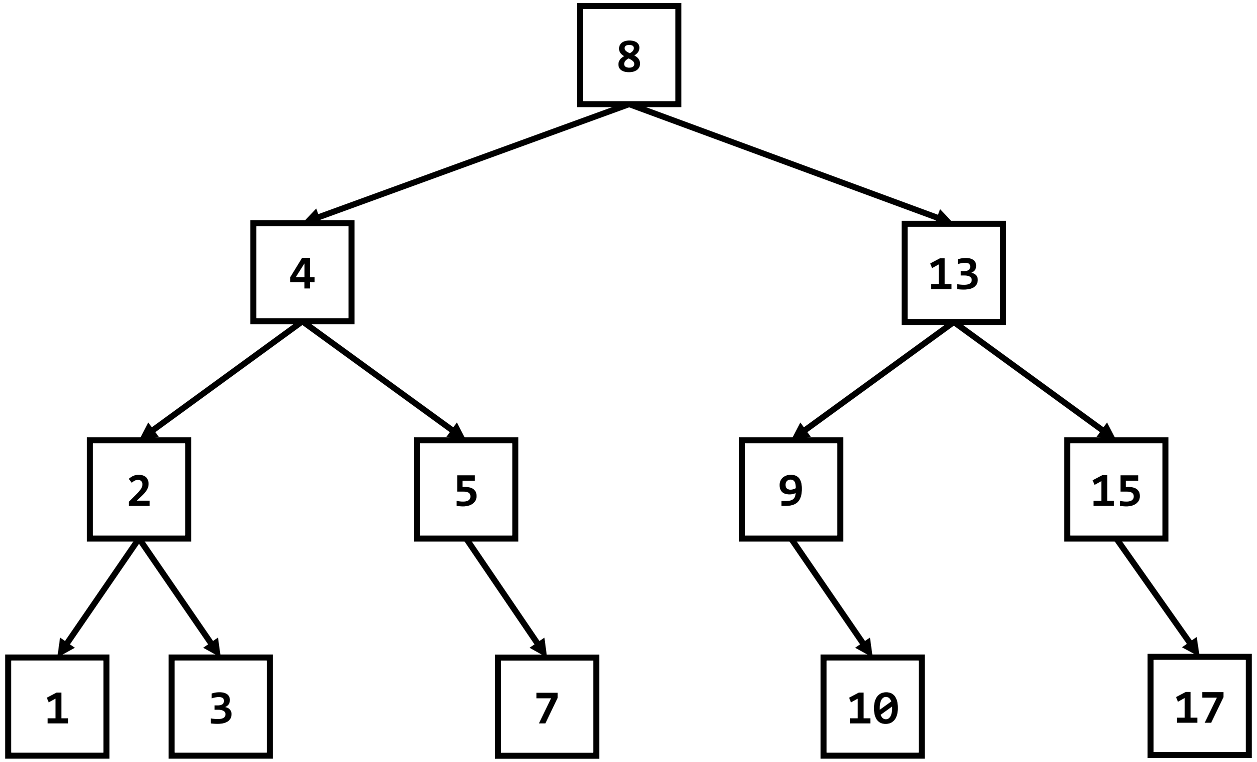
BST Operations - Deletion

- How to delete the node of the target **key** while satisfying BST conditions?
  (Case 3) If the node has two children, must find a replacement node
  - **In-order successor** : the smallest (left-most) node in the right subtree
  - After replacement, continue to delete **the node** at the original successor position



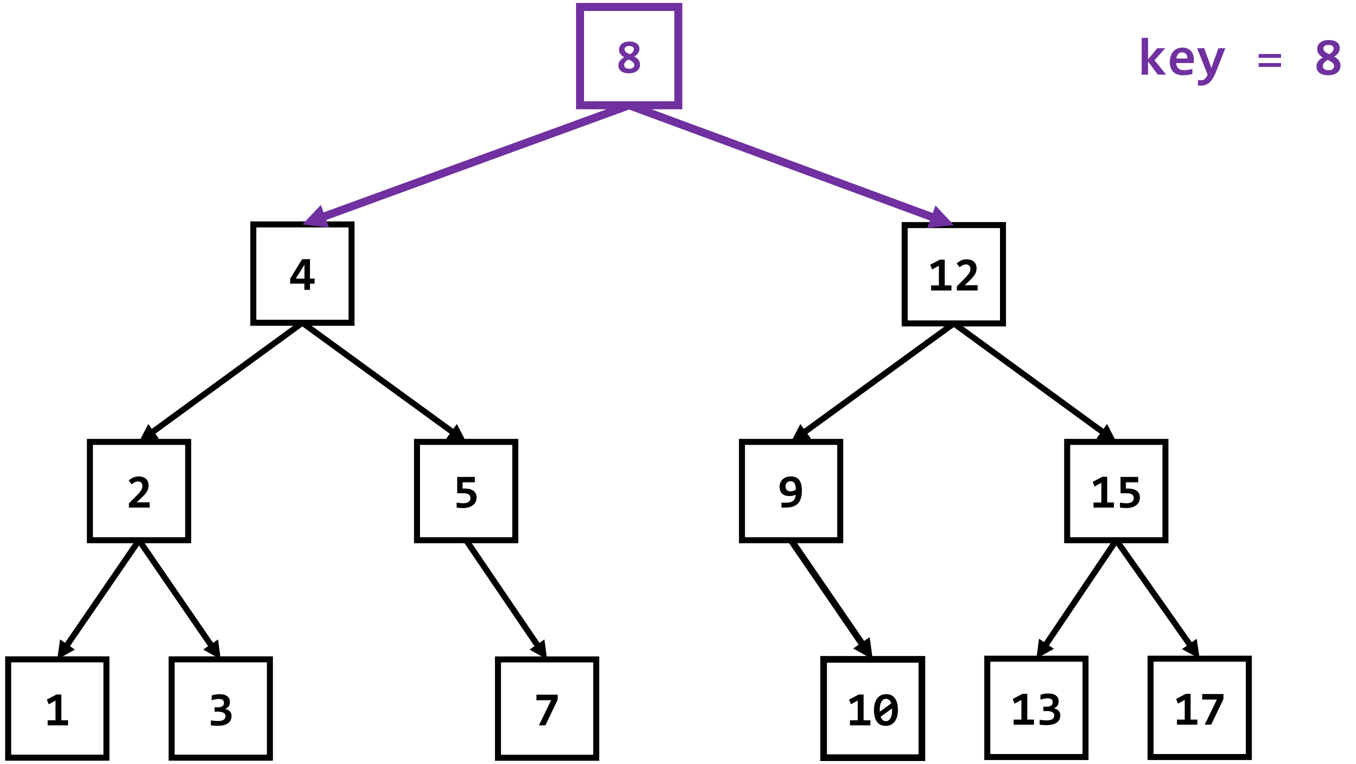
BST Operations - Deletion

- How to delete the node of the target **key** while satisfying BST conditions?
  (Case 3) If the node has two children, must find a replacement node
    - **In-order successor** : the smallest (left-most) node in the right subtree
    - After replacement, continue to delete **the node** at the original successor position



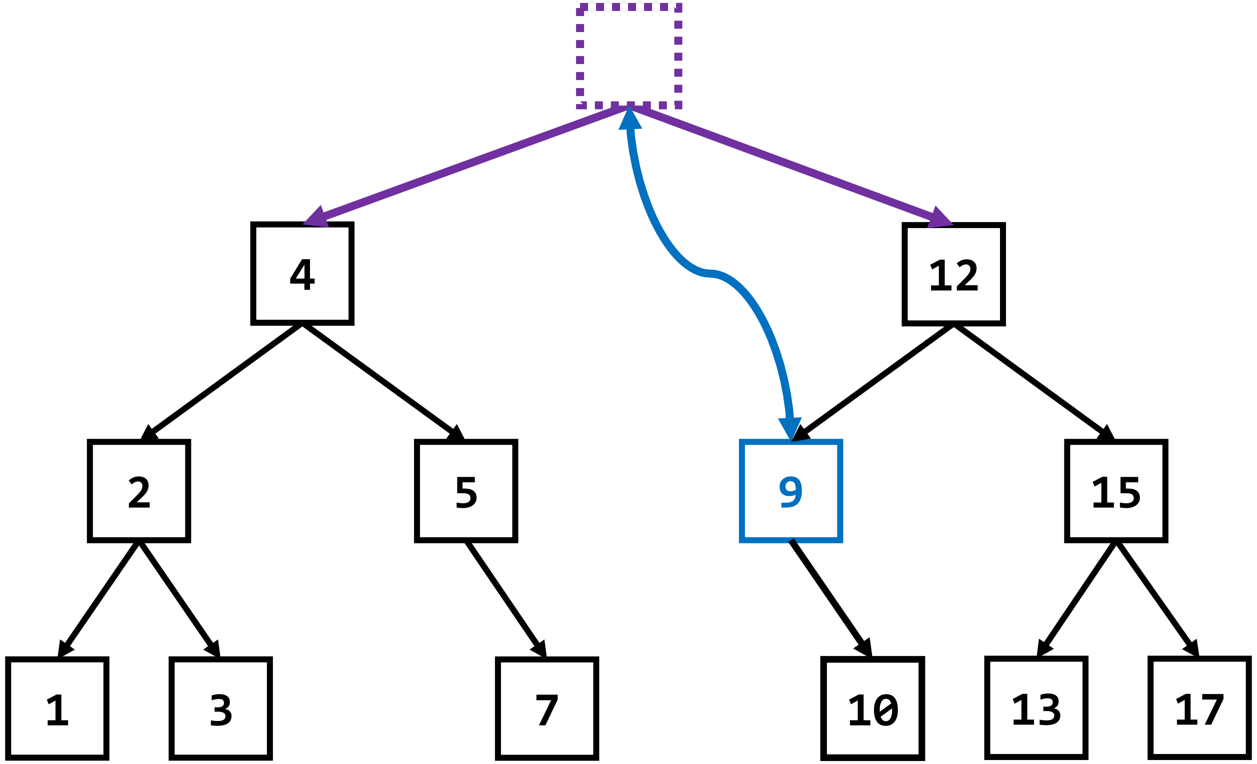
BST Operations - Deletion

- How to delete the node of the target **key** while satisfying BST conditions?
  (Case 3) If the node has two children, must find a replacement node
  - **In-order successor** : the smallest (left-most) node in the right subtree
  - After replacement, continue to delete **the node** at the original successor position



BST Operations - Deletion

- How to delete the node of the target **key** while satisfying BST conditions?
  - **(Case 3)** If the node has two children, must find a replacement node
    - **In-order successor** : the smallest (left-most) node in the right subtree
    - After replacement, continue to delete **the node** at the original successor position

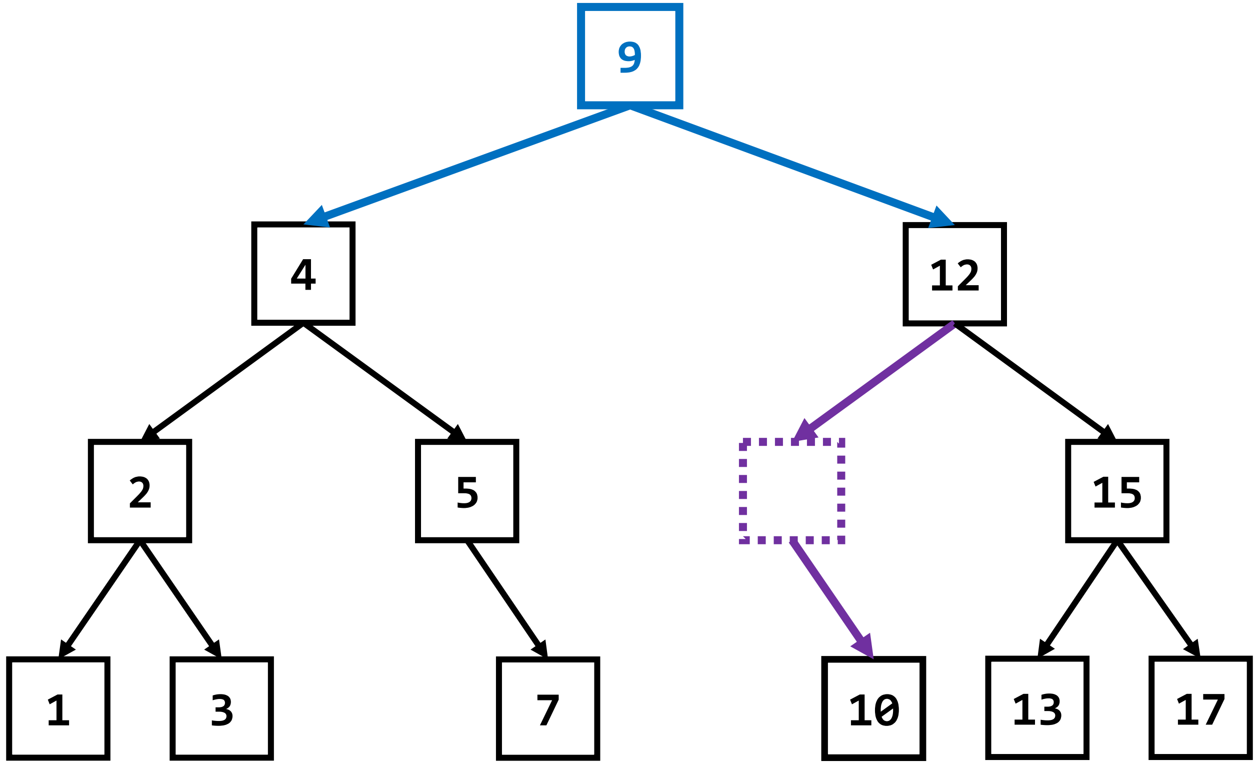


BST Operations - Deletion

- How to delete the node of the target **key** while satisfying BST conditions?

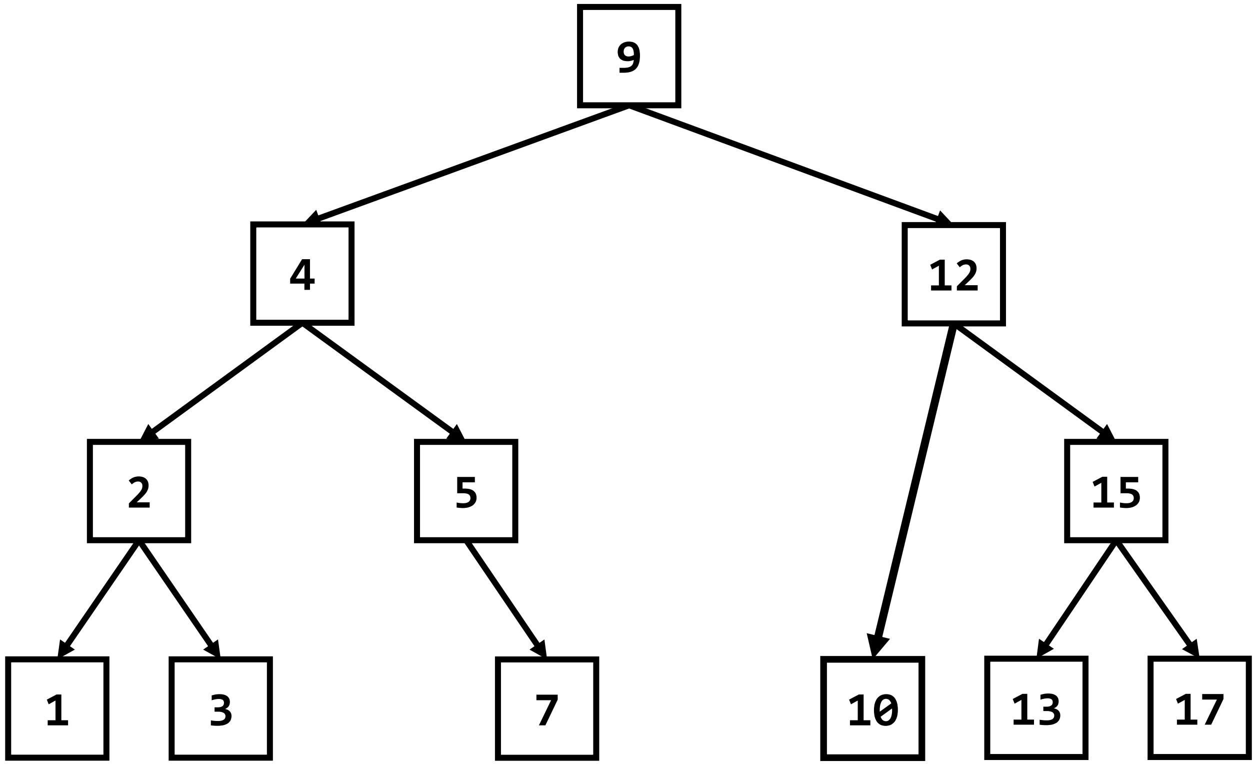
  (Case 3) If the node has two children, must find a replacement node

  - **In-order successor** : the smallest (left-most) node in the right subtree
  - After replacement, continue to delete **the node** at the original successor position



BST Operations - Deletion

- How to delete the node of the target **key** while satisfying BST conditions?
  (Case 3) If the node has two children, must find a replacement node
  - **In-order successor** : the smallest (left-most) node in the right subtree
  - After replacement, continue to delete **the node** at the original successor position

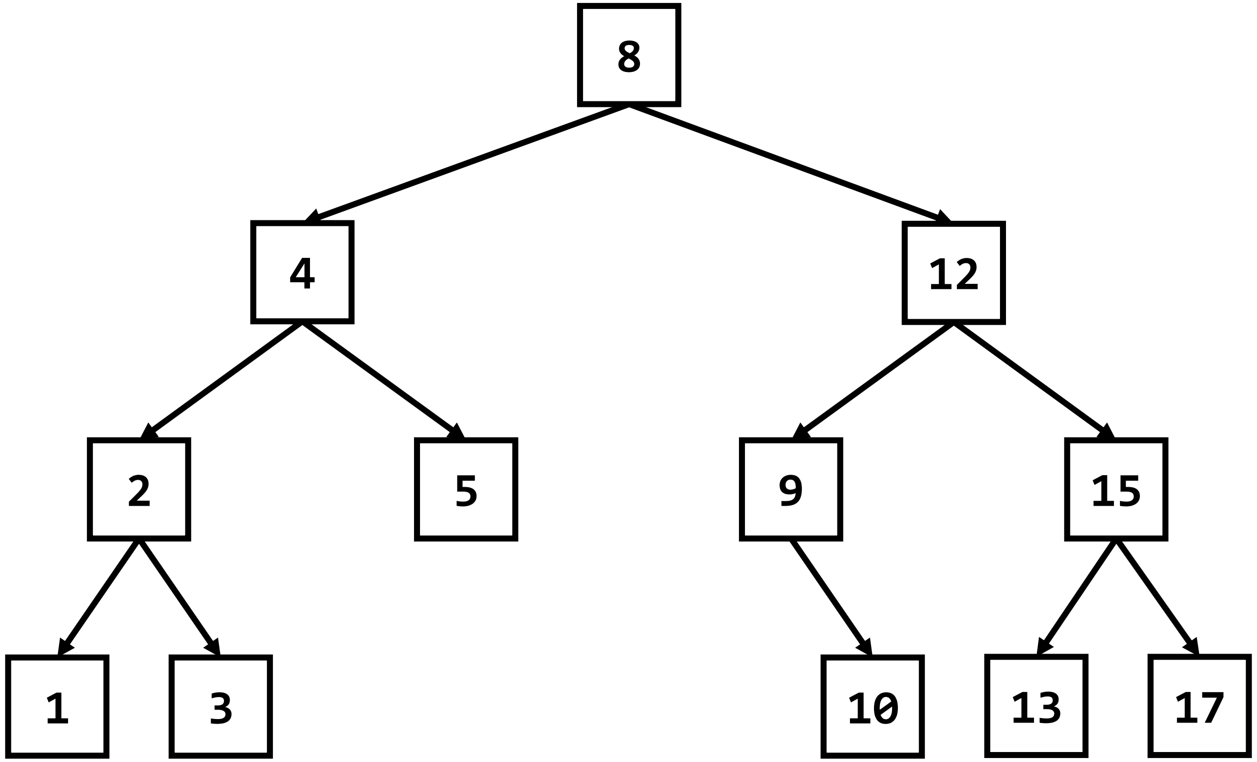


BST Operations - Deletion

- How to delete the node of the target **key** while satisfying BST conditions?
  - **(Case 1)** If **the node** has no child, it can be simply deleted
  - **(Case 2)** If **the node** has one child, it can be deleted like the linked list structure
  - **(Case 3)** If **the node** has two children, must find a replacement node
    - **In-order successor** : the smallest (left-most) node in the right subtree
    - After replacement, continue to delete **the node** at the original successor position

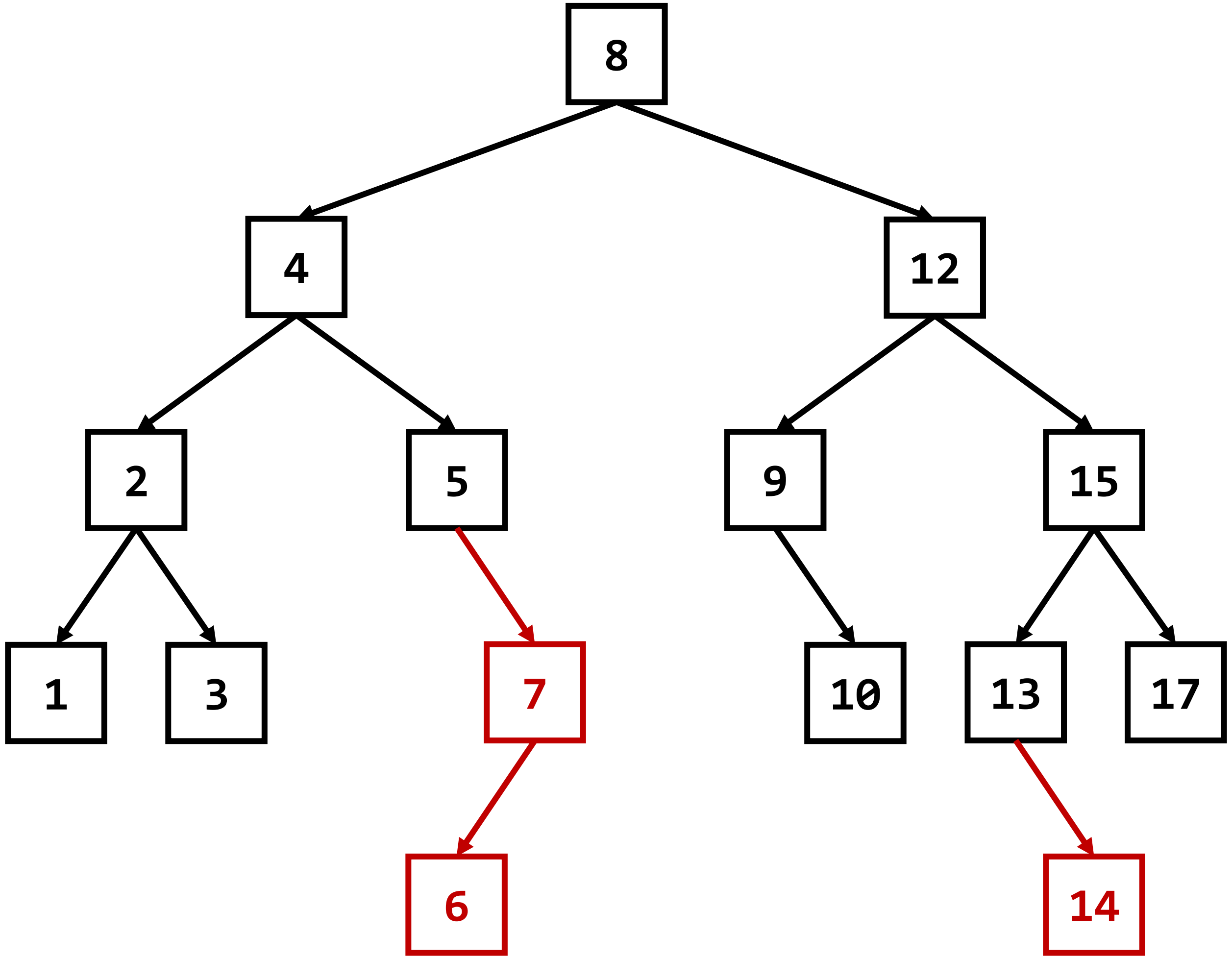
BST Operations - Examples

- What is the updated tree after inserting 7, 6, and 14 sequentially?



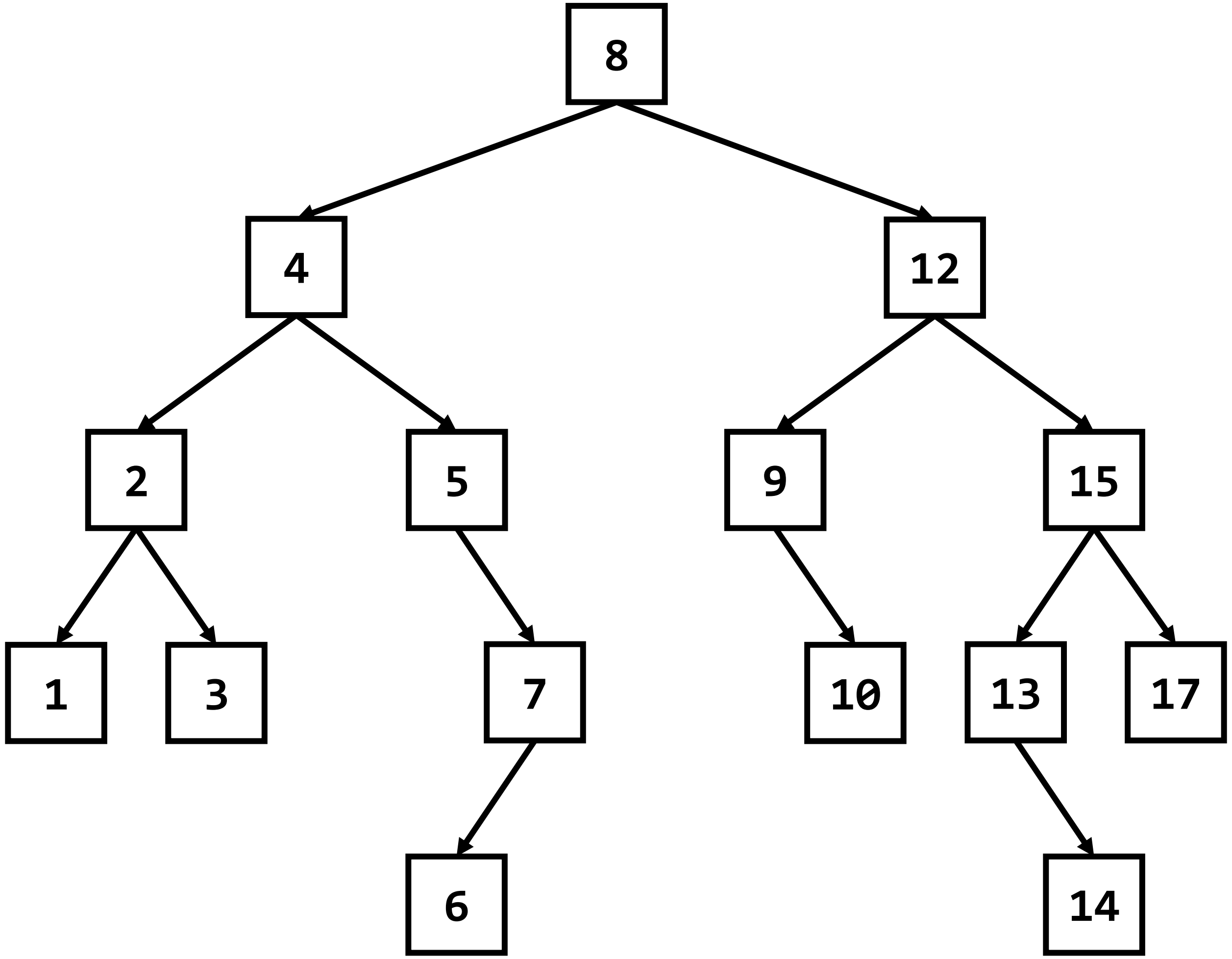
BST Operations - Examples

- What is the updated tree after inserting 7, 6, and 14 sequentially?



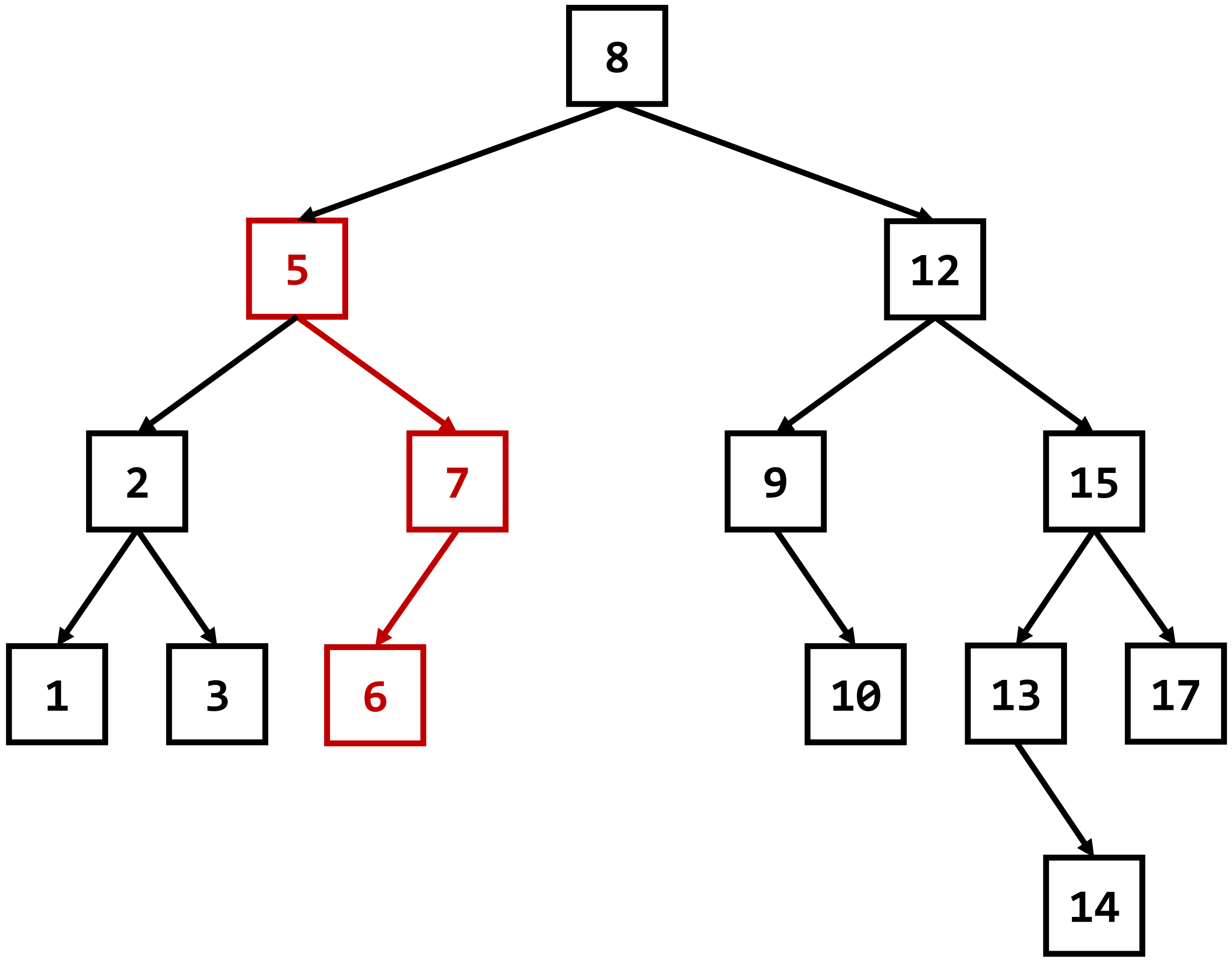
BST Operations - Examples

- What is the updated tree after deleting 4, and 12, and 9 sequentially?



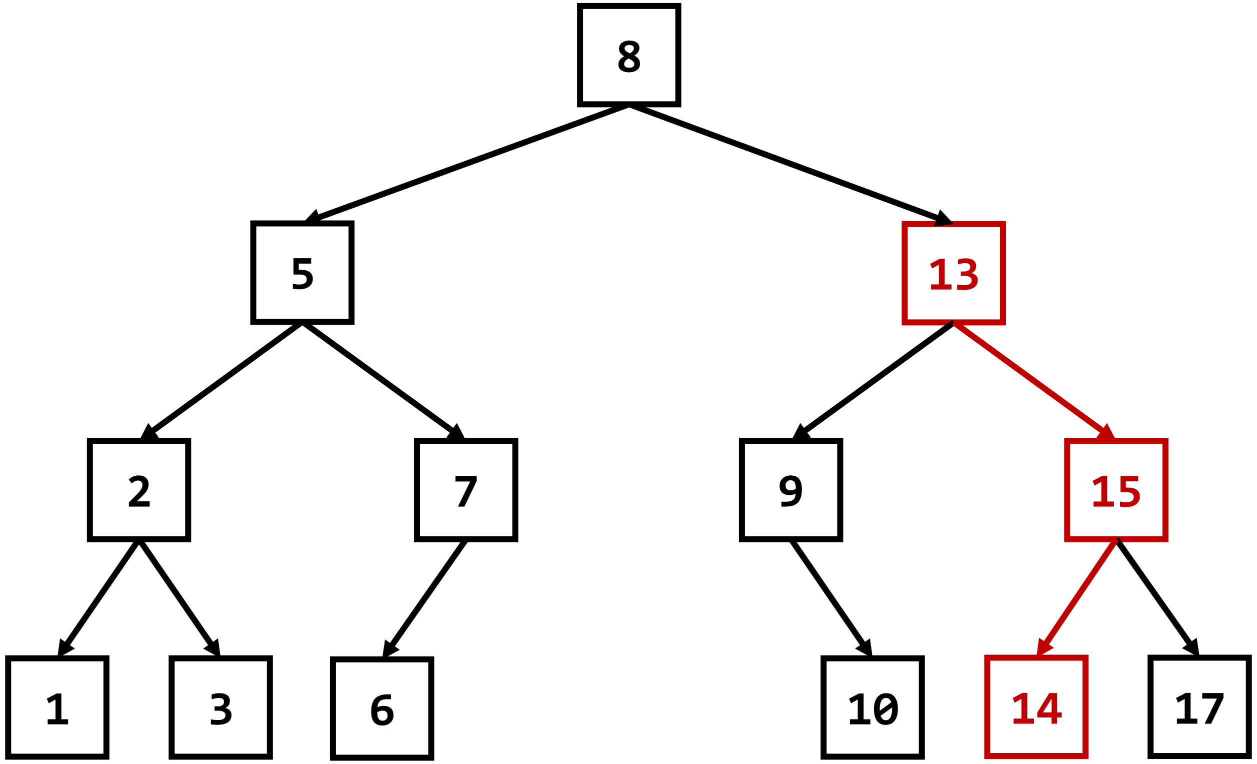
BST Operations - Examples

- What is the updated tree after deleting 4, and 12, and 9 sequentially?



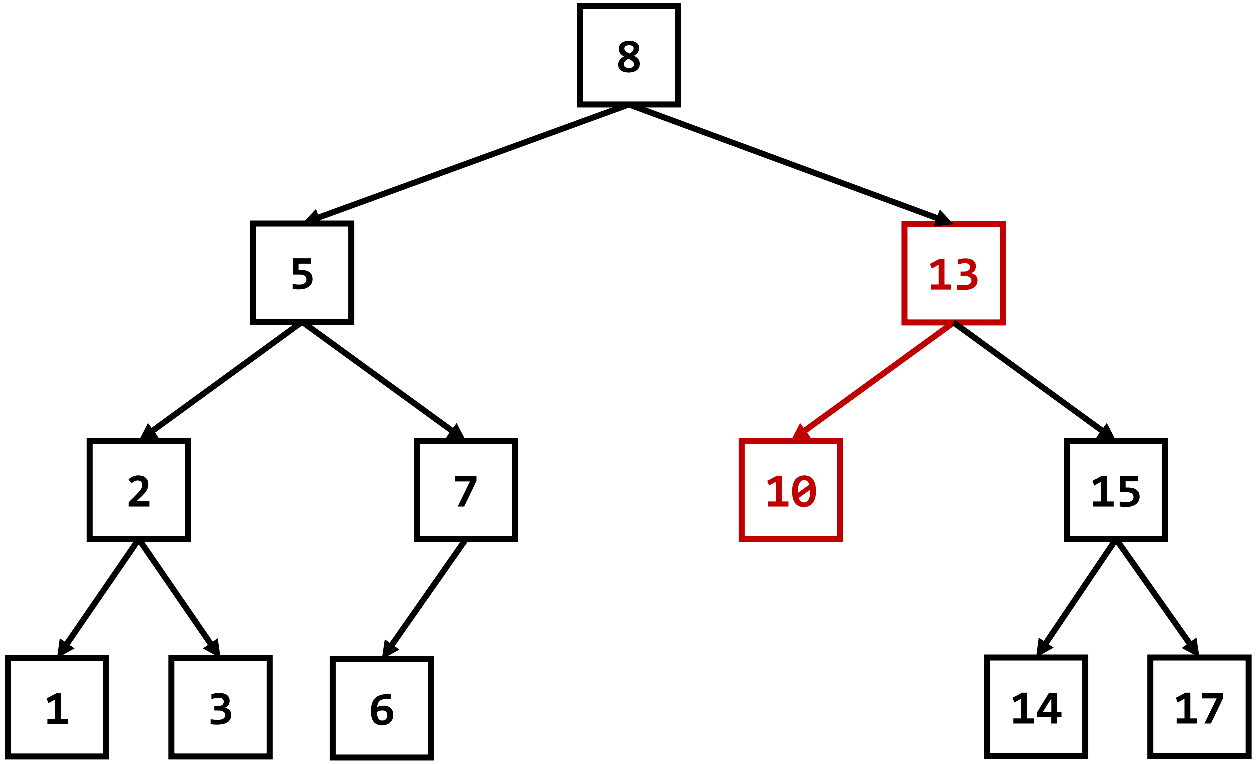
BST Operations - Examples

- What is the updated tree after deleting 4, and 12, and 9 sequentially?



BST Operations - Examples

- What is the updated tree after deleting 4, and 12, and 9 sequentially?

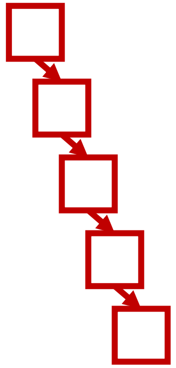


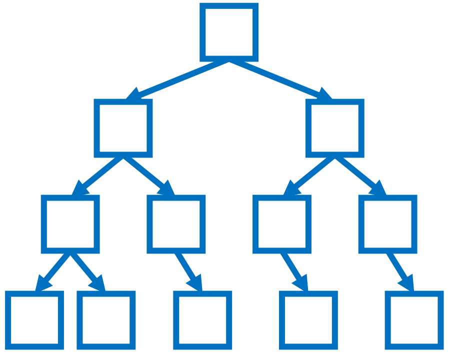
BST Operations - Time Complexity

- 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





Any Questions?