ICT-2101
Data Structure

Lecture 11

Binary Search Tree

Md. Mahmudur Rahman
Assistant Professor, IIT

Querying a binary search tree

- SEARCH operation,
- MINIMUM,
- MAXIMUM,
- SUCCESSOR, and
- PREDECESSOR.

Searching in a BST

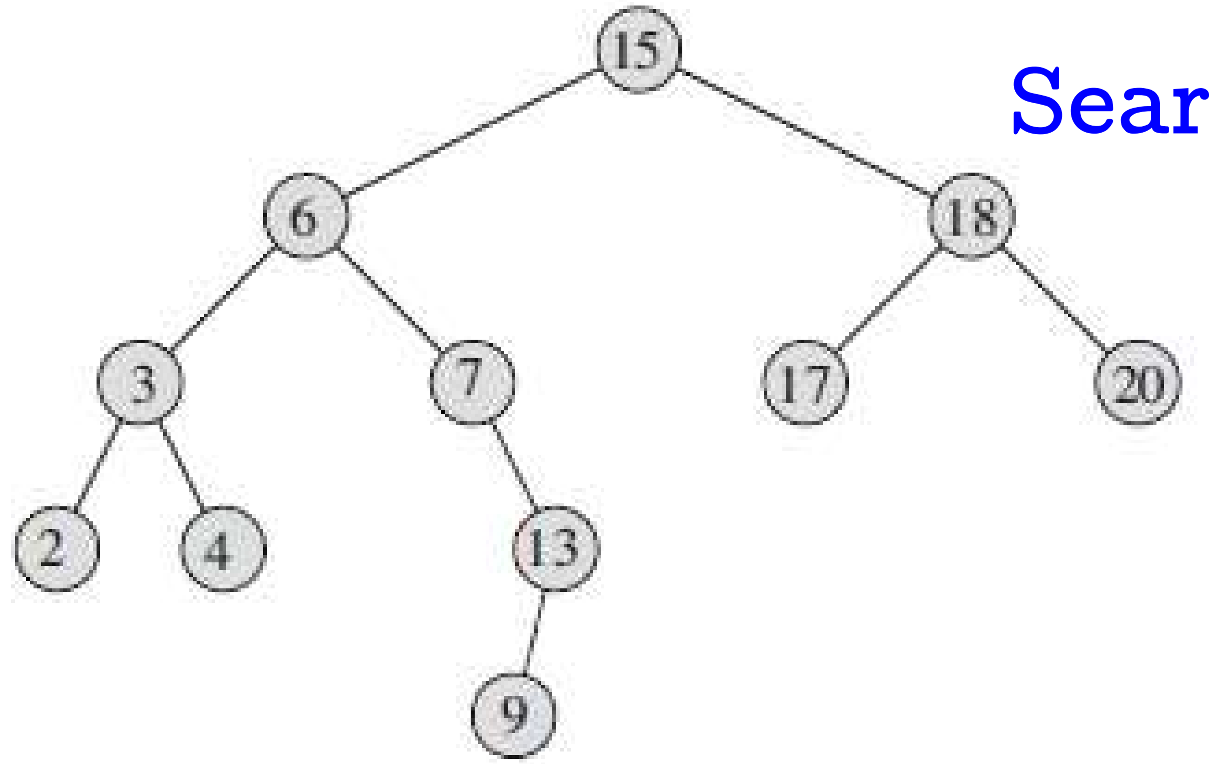
- The following procedure to search for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

```
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($x.right, k$)
```

Searching in a BST

- The procedure begins its search at the root and traces a simple path downward in the tree, as shown in Figure (next slide).
- For each node x it encounters, it compares the key k with **x.key**. *If the two keys are equal, the search terminates.*
- *If k is smaller* than x.*key*, *the search continues in the left sub-tree of x.*
- Symmetrically, if **k** is larger than x.*key*, *the search continues in the* right sub-tree.
- The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of TREE-SEARCH is O(h), where h is the height of the tree.

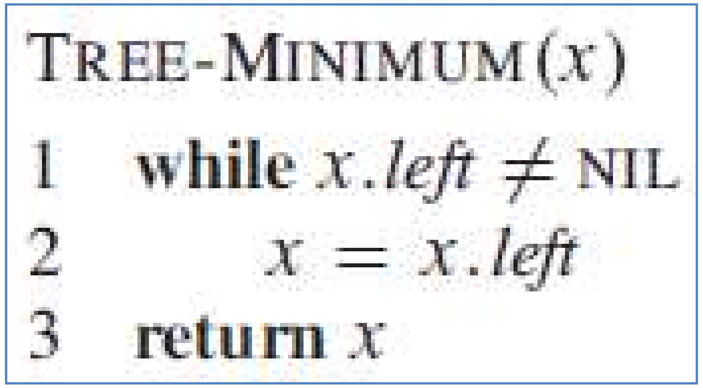
Searching in a BST



To search for the key 13 in the tree, we follow the path 15 →6 →7 →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.

Minimum and Maximum in BST

- The following procedure returns a pointer to the minimum element in the sub-tree rooted at a given node x, which we assume to be non-NIL:



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

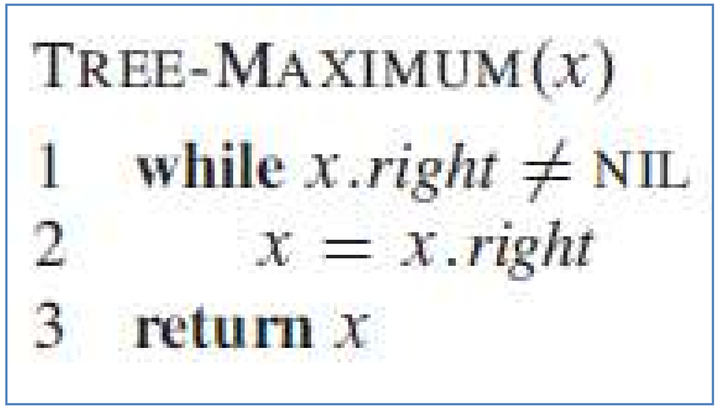
Minimum and Maximum in BST

- If a node x has no left sub-tree, then since every key in the right sub-tree of x is at least as large as x.*key*, *the minimum key in the sub-tree rooted at* x *is* $x.key$.

- *If node* x *has* a left sub-tree, then since no key in the right sub-tree is smaller than x.*key* *and every* key in the left sub-tree is not larger than x.*key*, *the minimum key in the sub-tree* rooted at x resides in the sub-tree rooted at x.*left*.

Minimum and Maximum in BST

- The following procedure returns a pointer to the maximum element in the sub-tree rooted at a given node x, which we assume to be non-NIL:



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

- Both of these procedures run in O(h) time on a tree of height h since, as in TREESEARCH, the sequence of nodes encountered forms a simple path downward from the root.

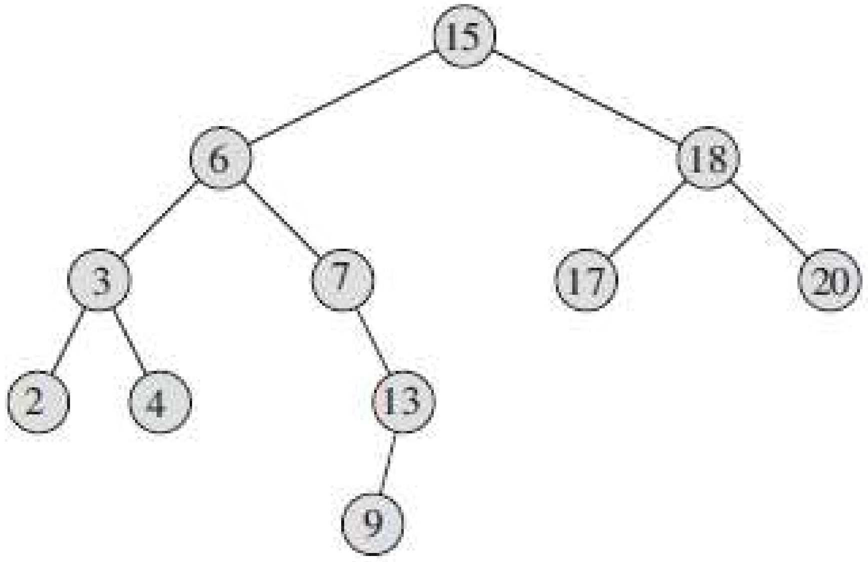
Successor and Predecessor in BST

- Given a node in a binary search tree, sometimes we need to find its successor in the sorted order determined by an in-order tree walk.

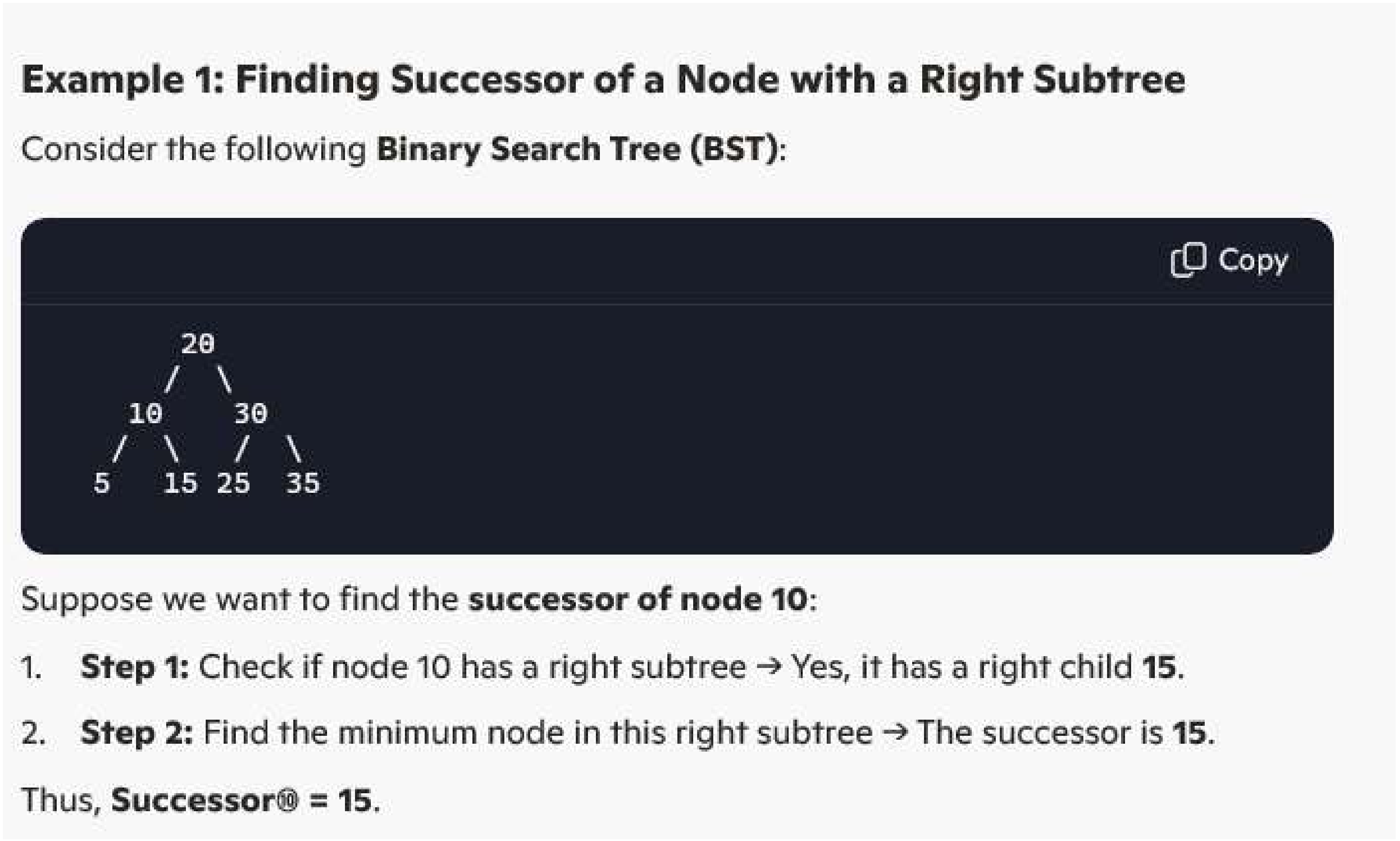
- If all keys are distinct, the successor of a node x is the node with the smallest key greater than x.key.

Successor and Predecessor in BST

- The following procedure returns the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the tree:



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



Example 1: Finding Successor of a Node with a Right Subtree

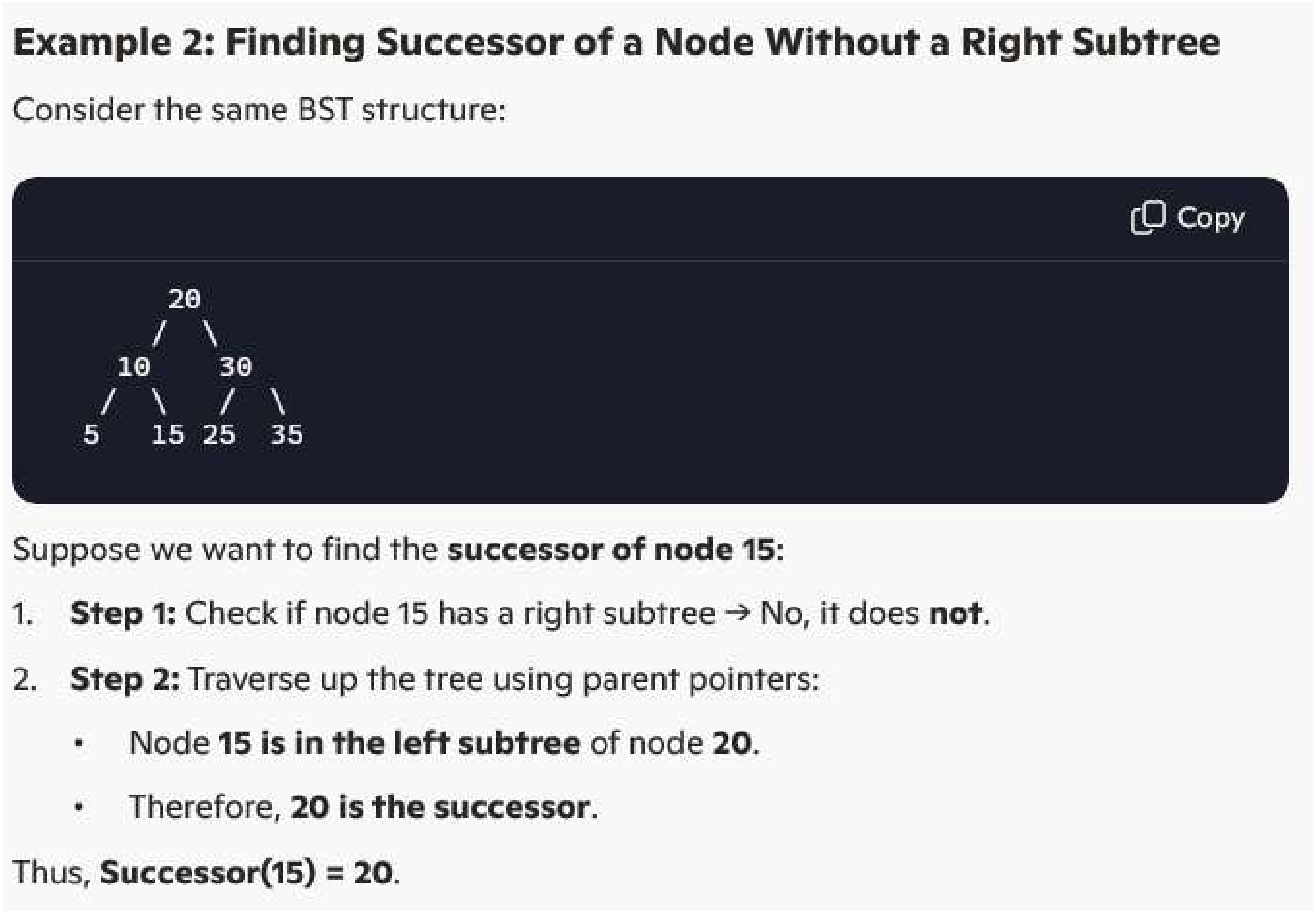
Consider the following **Binary Search Tree (BST)**:

```
       20
      /  \
    10     30
   /  \   /  \
  5   15 25   35
```

Suppose we want to find the **successor of node 10**:

1. **Step 1:** Check if node 10 has a right subtree → Yes, it has a right child **15.**
2. **Step 2:** Find the minimum node in this right subtree → The successor is **15.**

Thus, **Successor⑩ = 15.**



Example 2: Finding Successor of a Node Without a Right Subtree

Consider the same BST structure:

```
        20
       /  \
     10    30
    /  \   /  \
   5   15 25   35
```

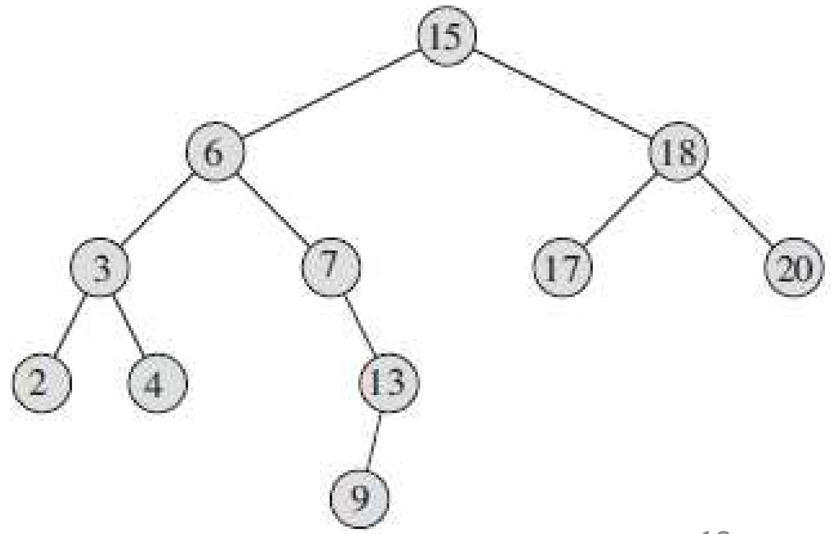
Suppose we want to find the **successor of node 15**:

1. **Step 1:** Check if node 15 has a right subtree → No, it does **not**.
2. **Step 2:** Traverse up the tree using parent pointers:
   - Node **15 is in the left subtree** of node **20**.
   - Therefore, **20 is the successor**.

Thus, **Successor(15) = 20**.

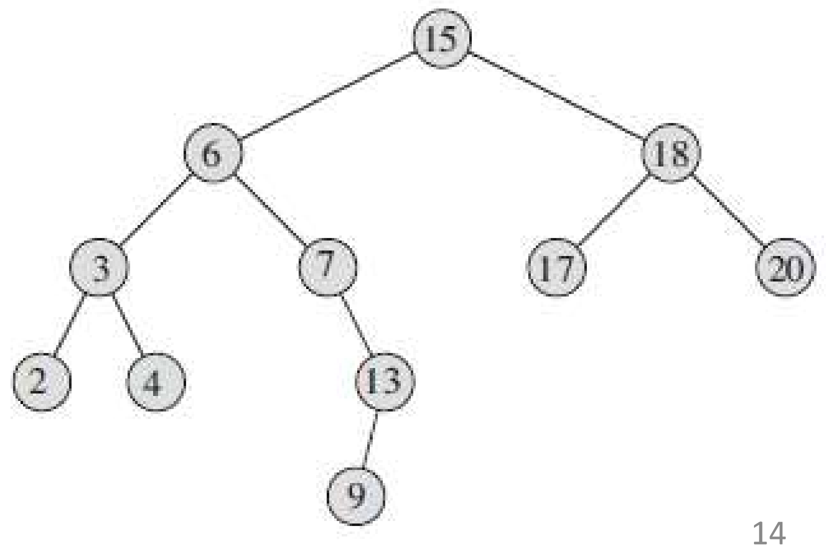
Successor and Predecessor in BST

- We break the code for TREE-SUCCESSOR into two cases. If the right sub-tree of node x is nonempty, then the successor of x is just the leftmost node in x's right sub-tree, which we find in line 2 by calling TREE-MINIMUM(x.*right*).
- *For* example, the successor of the node with key 15 in Figure is the node with key 17.



Successor and Predecessor in BST

- On the other hand, if the right sub-tree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. In the figure, the successor of the node with key 13 is the node with key 15. To find y, we simply go up the tree from x until we encounter a node that is the left child of its parent; lines 3–7 of TREE-SUCCESSOR handle this case.



Successor and Predecessor in BST

- The running time of TREE-SUCCESSOR on a tree of height h is O(h), since we either follow a simple path up the tree or follow a simple path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time O(h).

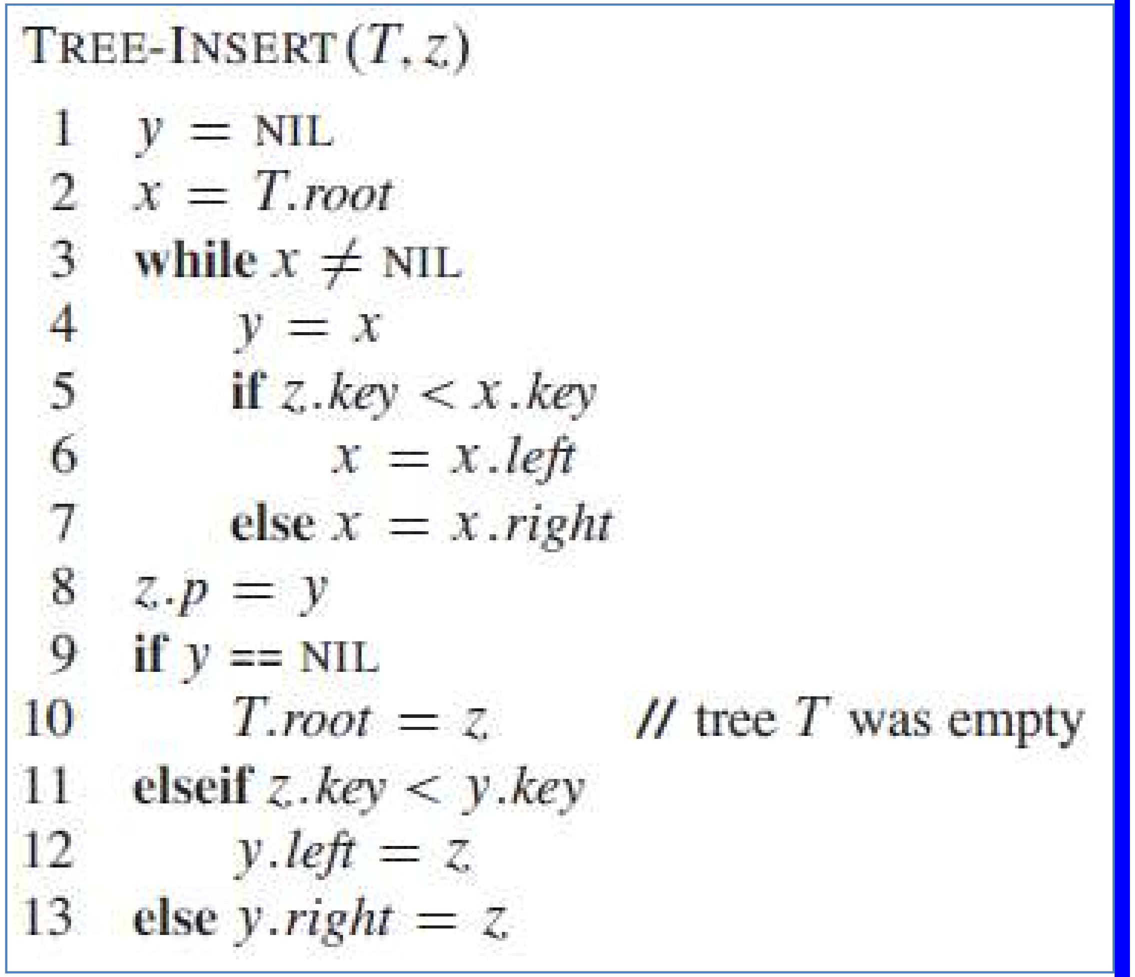
Insertion in BST

- To insert a new value into a binary search tree T , we use the procedure TREE-INSERT.
- The procedure takes a node z for which

  z.key=v ,

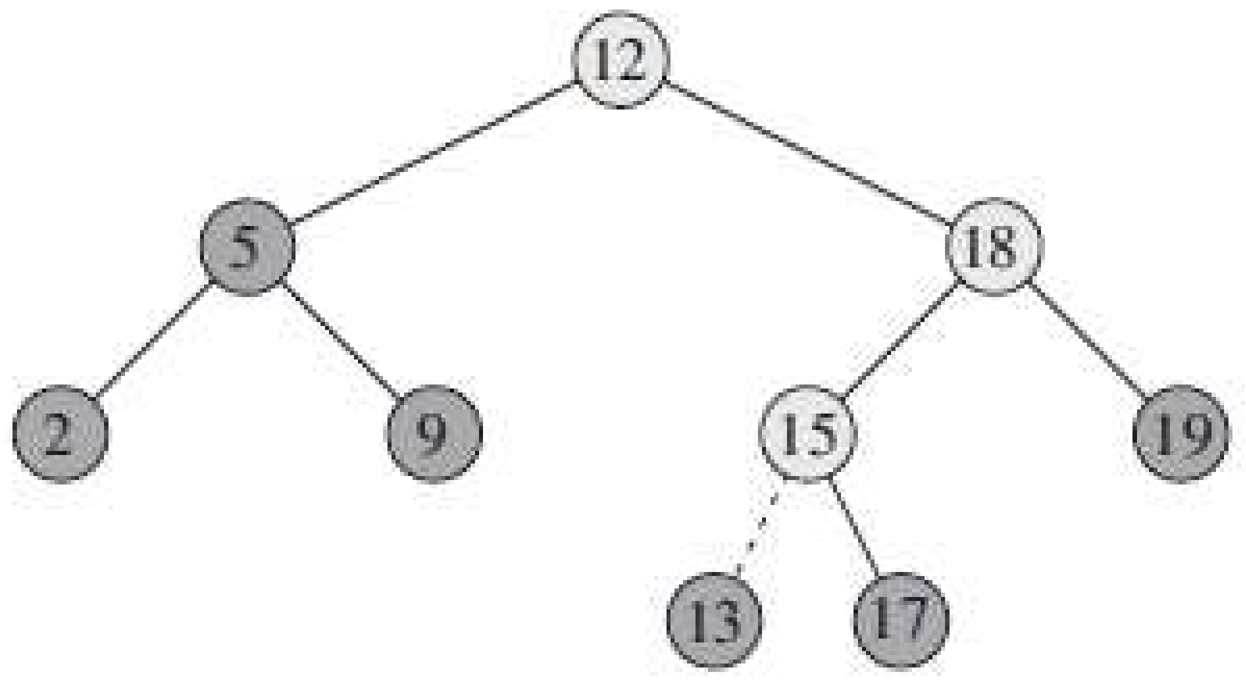
  z.left=NIL,

  z.right=NIL.
- *It modifies T and some of the attributes of z in such a way that* it inserts z into an appropriate position in the tree.



```
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
```

Insertion in BST



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.

Insertion in BST

Explanation:

- TREE-INSERT begins at the root of the tree and the pointer x traces a simple path downward looking for a NIL to replace with the input item z. The procedure maintains the ***trailing pointer y as the parent*** of x.
- After initialization, the **while loop in lines 3–7 causes these two pointers** to move down the tree, going left or right depending on the comparison of z.*key* with x.*key*, *until x becomes NIL. This NIL occupies the position where we wish to* place the input item z. We need the trailing pointer y, because by the time we find the NIL where z belongs, the search has proceeded one step beyond the node that needs to be changed. Lines 8–13 set the pointers that cause ’ to be inserted.

Deletion in BST

"The overall strategy for deleting a node (z) from a binary search tree (T) has three basic cases but, as we shall see, one of the cases is a bit tricky.

•**If (z) has no children**, then we simply remove it by modifying its parent to replace (z) with NIL as its child.

•**If (z) has just one child**, then we elevate that child to take (z)'s position in the tree by modifying (z)'s parent to replace (z) by (z)'s child.

•**If (z) has two children,** then we find (z)'s successor (y)—which must be in (z)'s right subtree—and have (y) take (z)'s position in the tree. The rest of (z)'s original right subtree becomes (y)'s new right subtree, and (z)'s left subtree becomes (y)'s new left subtree. This case is the tricky one because, as we shall see, it matters whether (y) is (z)'s right child."

Deletion in BST

- The procedure for deleting a given node ’ from a binary search tree T takes as arguments pointers to T and z. Consider the four cases to delete z.

Case 1:

- If z has no left child (part (a) of the figure), then we replace z by its right child, which may or may not be NIL. When z’s right child is NIL, this case deals with the situation in which z has no children. When z’s right child is non-NIL, this case handles the situation in which z has just one child, which is its right child.

Case 2:

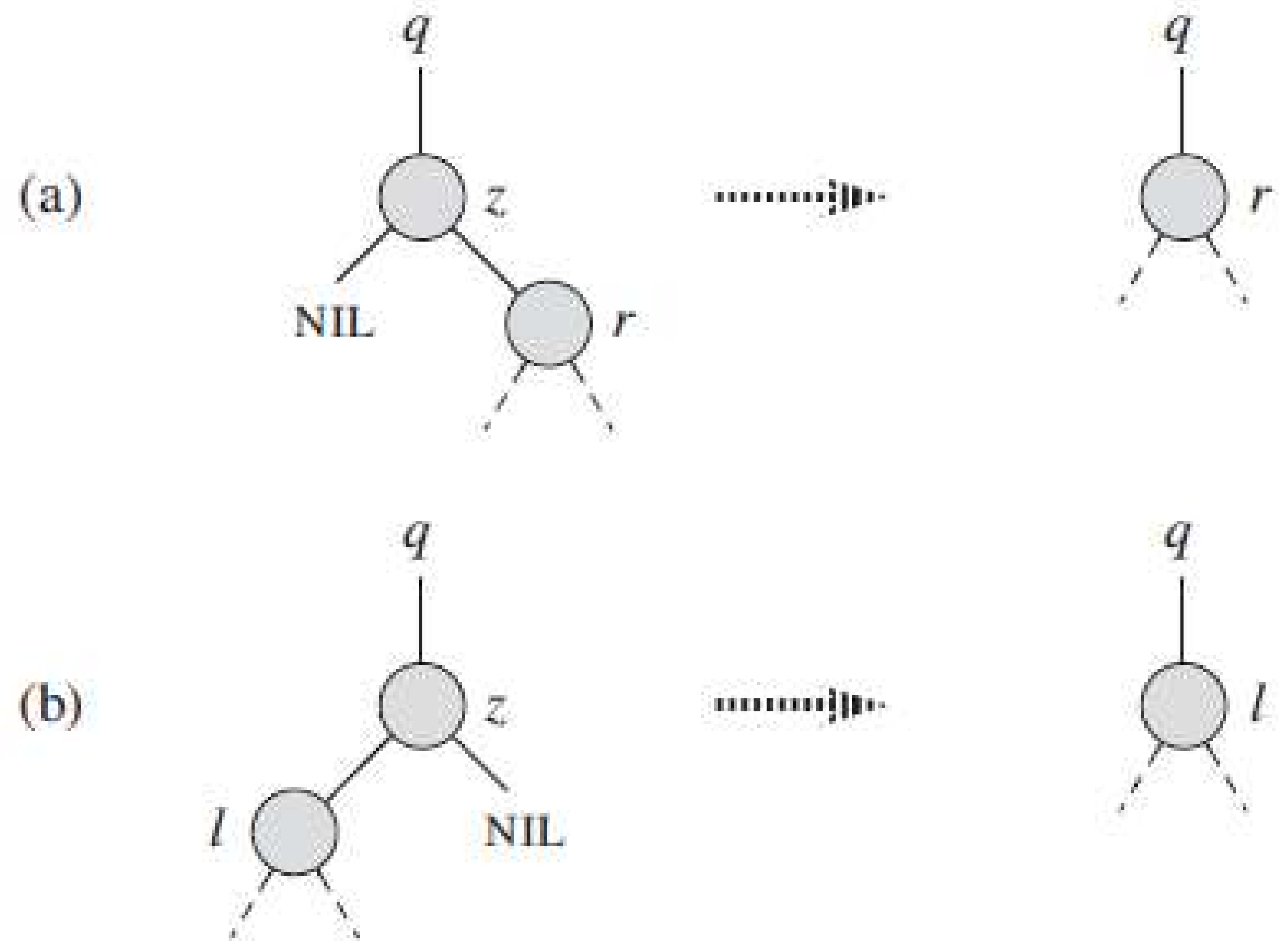
- If z has just one child, which is its left child (part (b) of the figure), then we replace z by its left child.

Deletion in BST

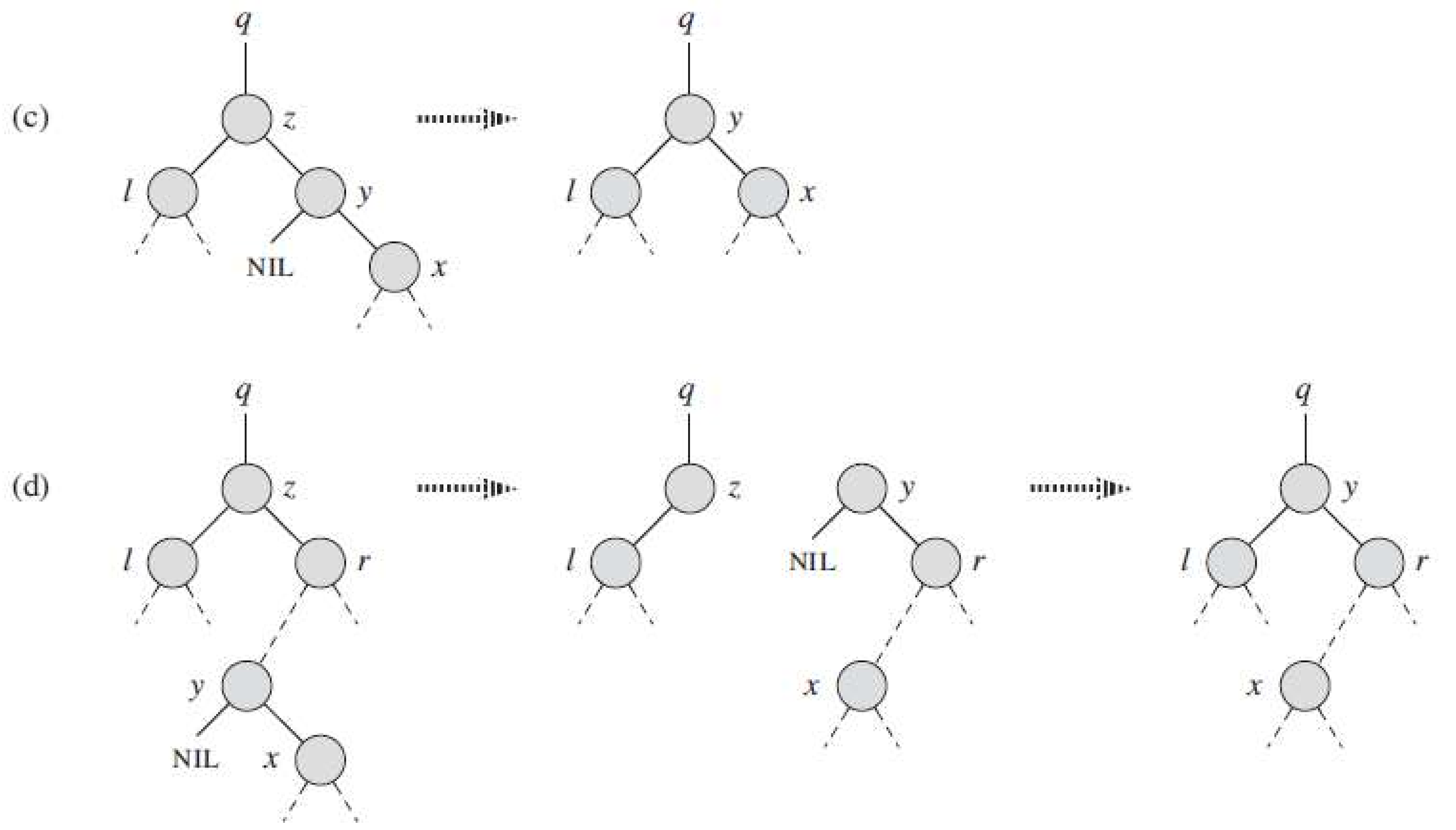
Case 3 and 4:

- Otherwise, z has both a left and a right child. We find z’s successor y, which lies in z’s right sub-tree and has no left child. We want to splice y out of its current location and have it replace ’ in the tree.
- If y is z’s right child (part (c)), then we replace z by y, leaving y’s right child alone.
- Otherwise, y lies within z’s right sub-tree but is not z’s right child (part (d)). In this case, we first replace y by its own right child, and then we replace z by y.

Deletion in BST



Deletion in BST



Deleting a node z from a binary search tree. Node z may be the root, a left child of node q, or a right child of q.

(a) Node z has no left child. We replace z by its right child r, which may or may not be NIL.

(b) Node z has a left child l but no right child. We replace z by l .

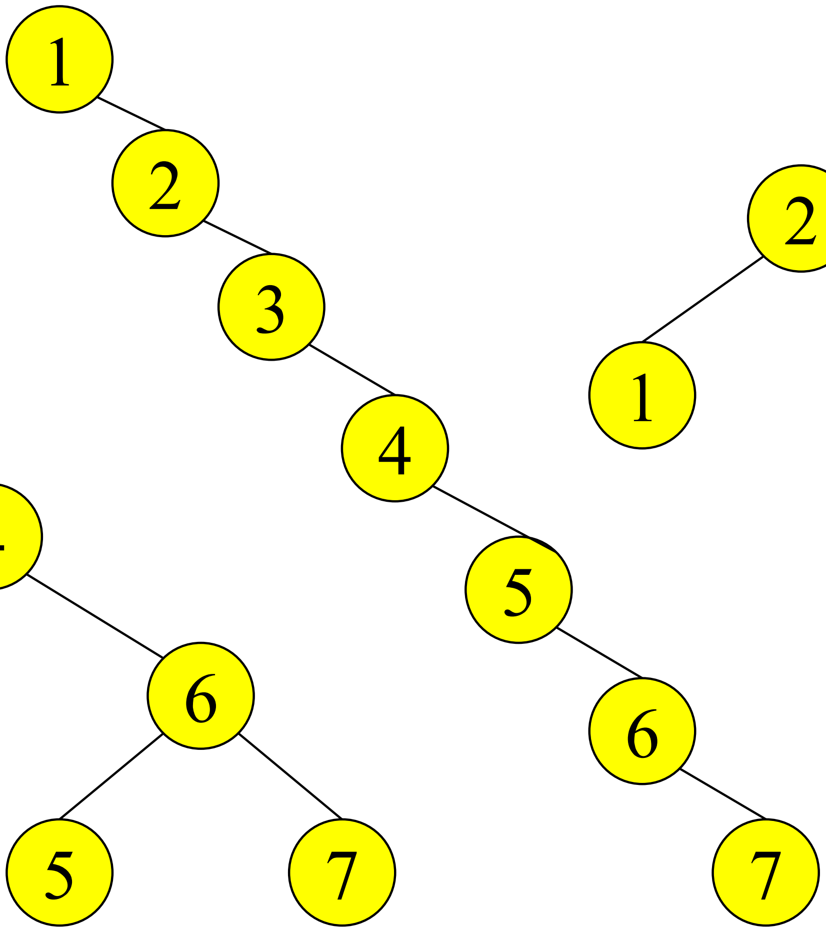
(c) Node z has two children; its left child is node l , its right child is its successor y, and y’s right child is node x. We replace z by y, updating y’s left child to become l, but leaving x as y’s right child.

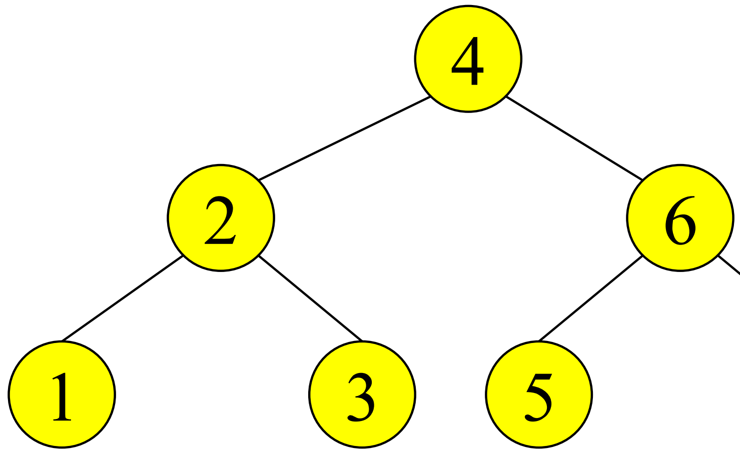
(d) Node z has two children (left child l and right child r), and its successor y≠ r lies within the sub-tree rooted at r. We replace y by its own right child x, and we set y to be r’s parent. Then, we set y to be q’s child and the parent of l .

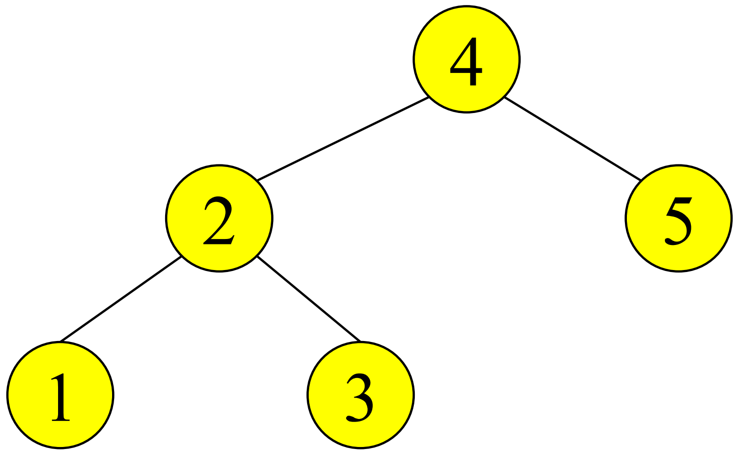
Binary Search Tree - Worst Time

- Worst case running time is O(N)
  - What happens when you Insert elements in ascending order?
    - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
  - Problem: Lack of “balance”:
    - compare depths of left and right sub-tree
  - Unbalanced degenerate tree

Balanced and unbalanced BST







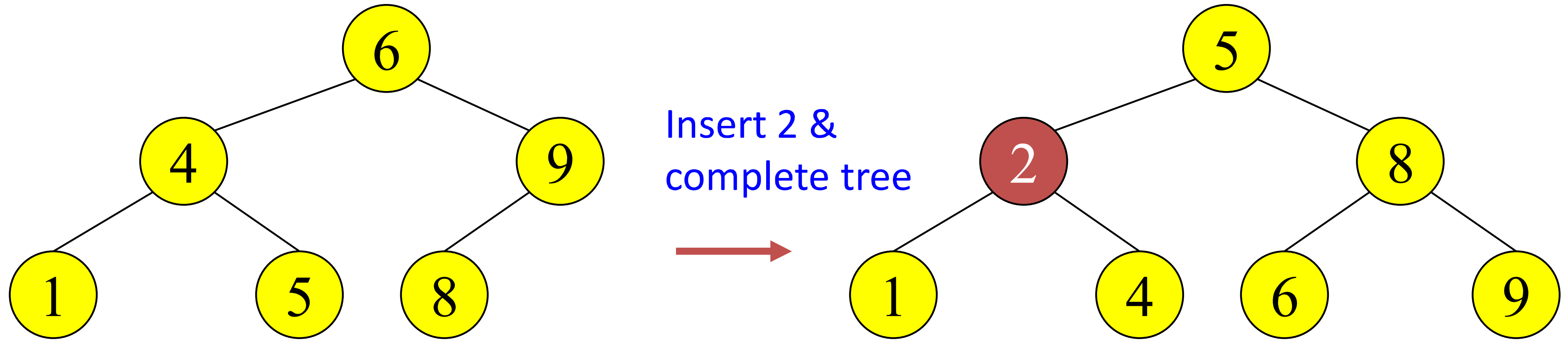
Is this “balanced”?

Balancing Binary Search Trees

- Many algorithms exist for keeping binary search trees balanced
  - Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
  - Splay trees and other self-adjusting trees
  - B-trees and other multiway search trees

Perfect Balance

- Want a complete tree after every operation
  - tree is full except possibly in the lower right
- This is expensive
  - For example, insert 2 in the tree on the left and then rebuild as a complete tree



AVL TREE

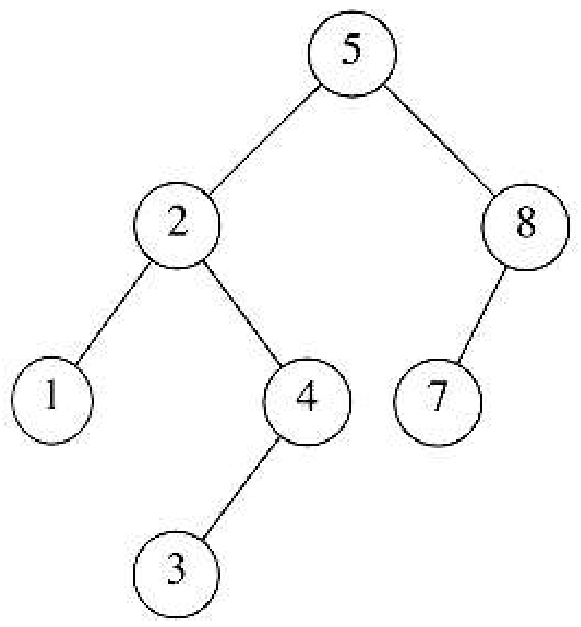
- ✓ An AVL tree is a balanced binary search tree. Named after their inventors, Adelson-Velskii and Landis
- ✓ They are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an O(logn) search time. Addition and deletion operations also take O(logn) time.
- ✓ An AVL tree is a binary search tree which has the following properties:
  1. The sub-trees of every node differ in height by at most one.
  2. Every sub-tree is an AVL tree.
- ✓ What if the input to binary search tree comes in sorted (ascending or descending) manner?

AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - height(left sub-tree) - height(right sub-tree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right sub-tree can differ by no more than 1
  - Store current heights in each node

AVL - Good but not Perfect Balance

- An AVL tree is a binary search tree in which
  - for *every* node in the tree, the height of the left and right subtrees differ by at most 1.



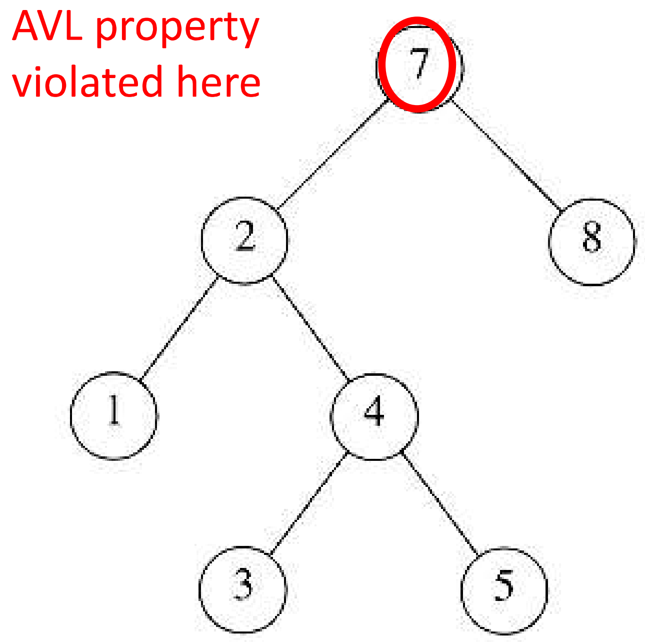
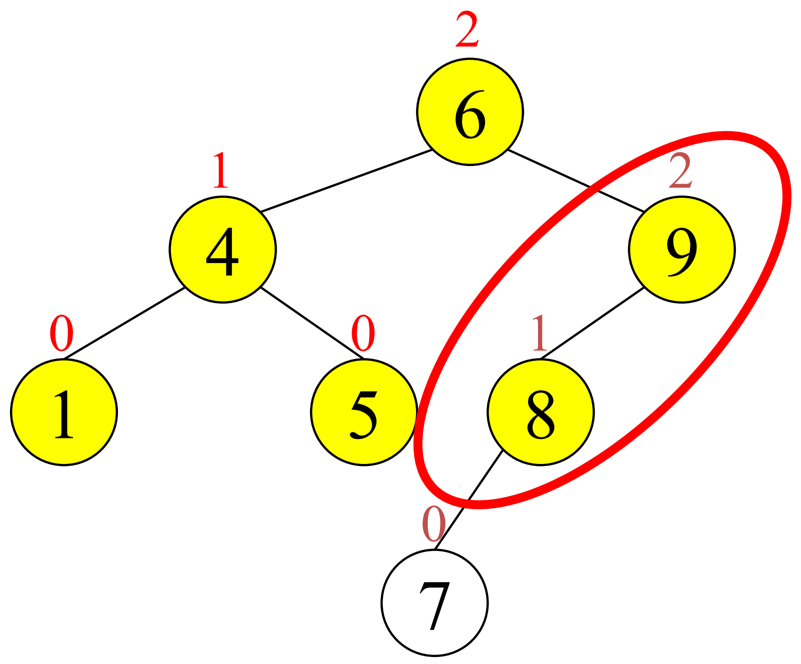


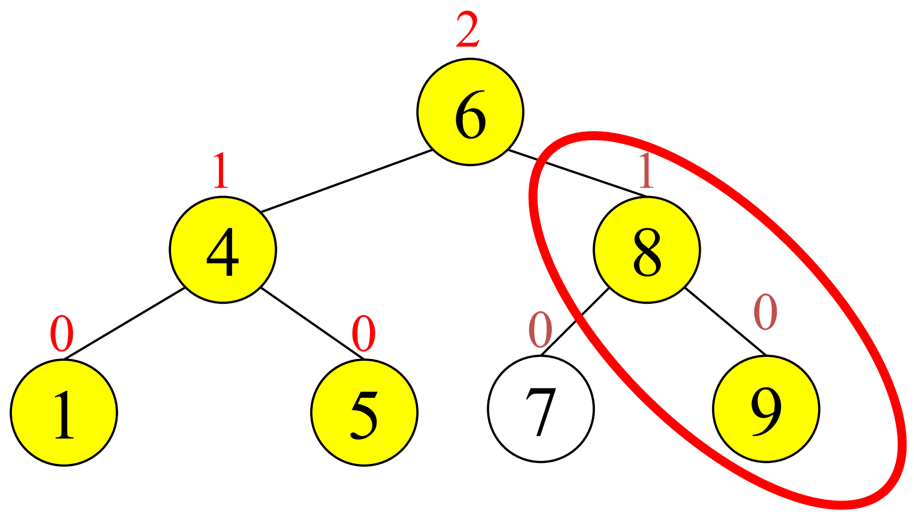
Figure 4.32 Two binary search trees. Only the left tree is AVL.

Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

Single Rotation in an AVL Tree





Single Rotation in an AVL Tree

AVL Rotations

To make itself balanced, an AVL tree may perform four kinds of rotations –

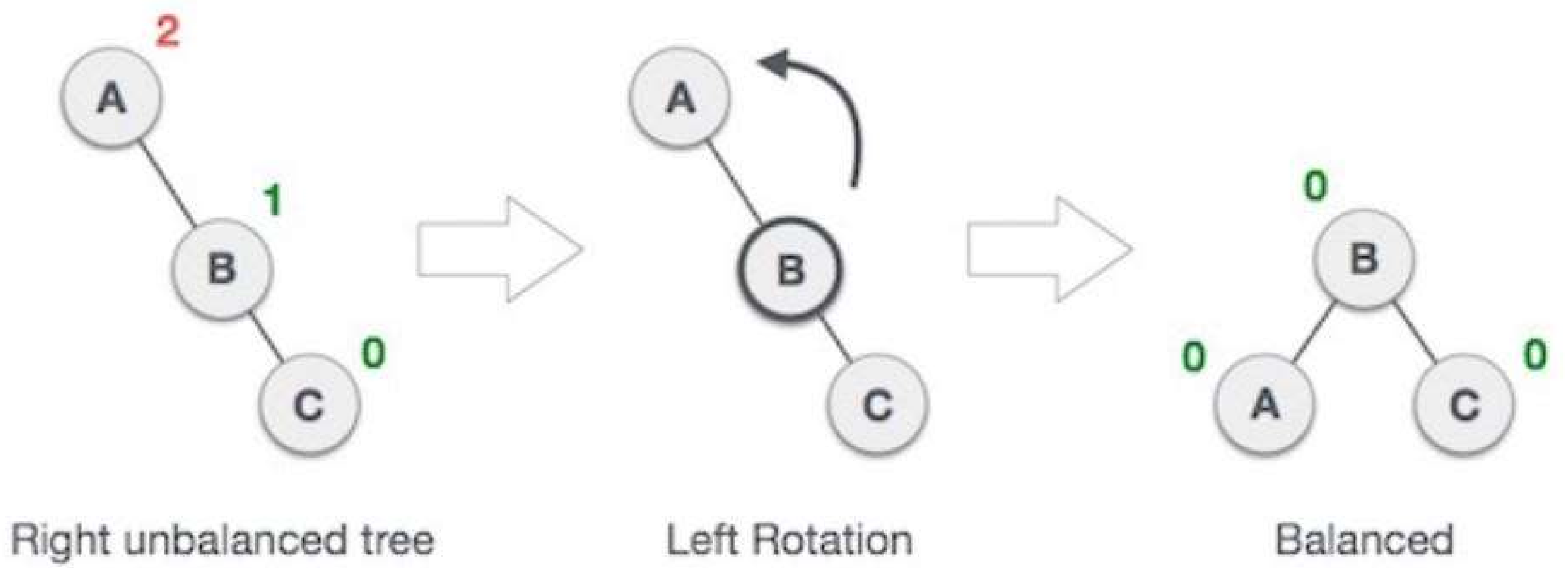
- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

First two rotations are single rotations and next two rotations are double rotations. Two have an unbalanced tree we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Single Rotation in an AVL Tree

Left Rotation

If a tree become unbalanced, when a node is inserted into the right subtree of right subtree, then we perform single left rotation –

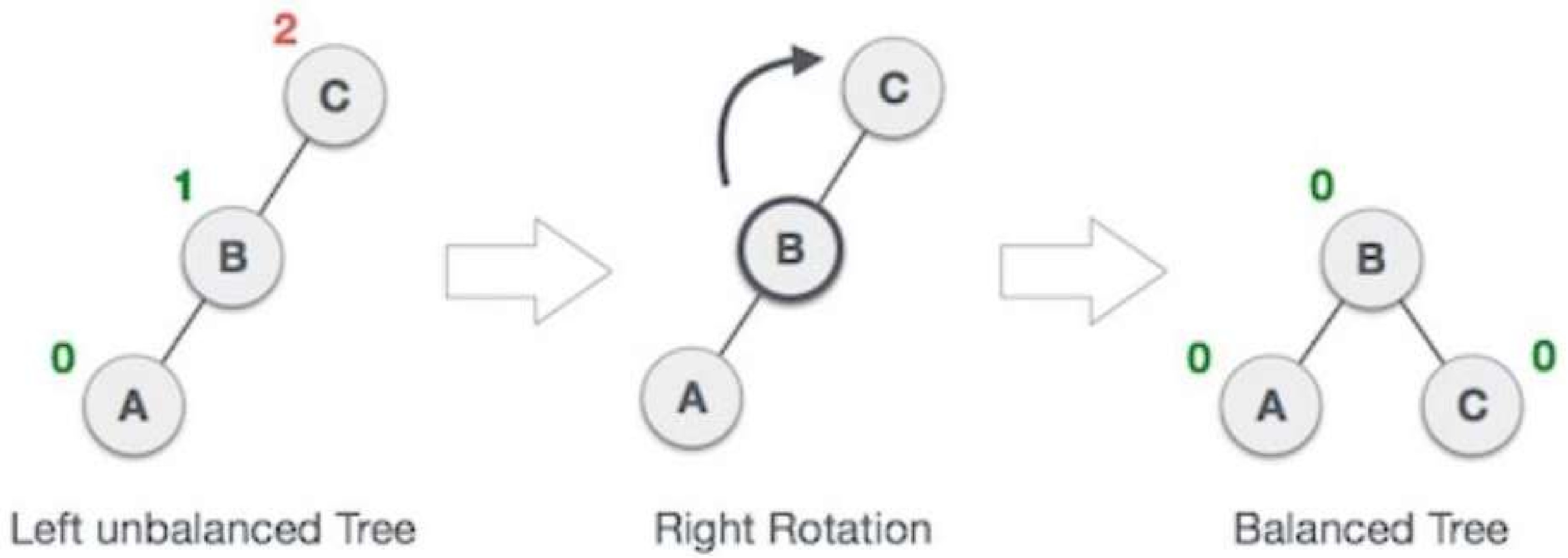


In our example, node **A** has become unbalanced as a node is inserted in right subtree of A's right subtree. We perform left rotation by making **A** left-subtree of B.

Single Rotation in an AVL Tree

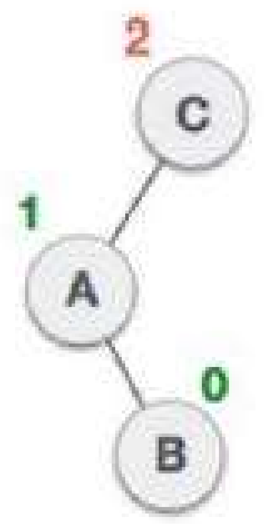
Right Rotation

AVL tree may become unbalanced if a node is inserted in the left subtree of left subtree. The tree then needs a right rotation.



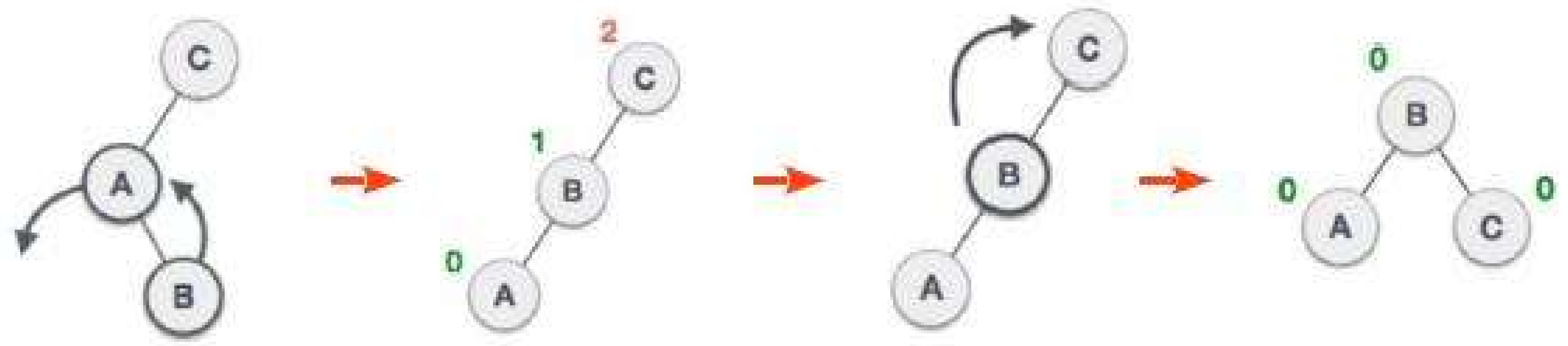
As depicted, the unbalanced node becomes right child of its left child by performing a right rotation.

Single Rotation in an AVL Tree

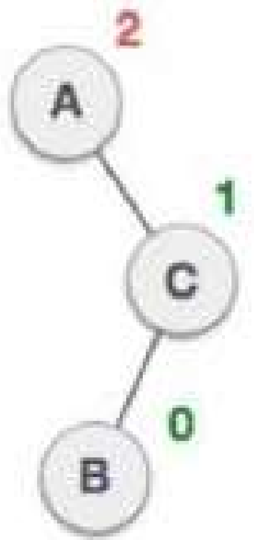


Left-Right Rotation

A node has been inserted into right subtree of left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.

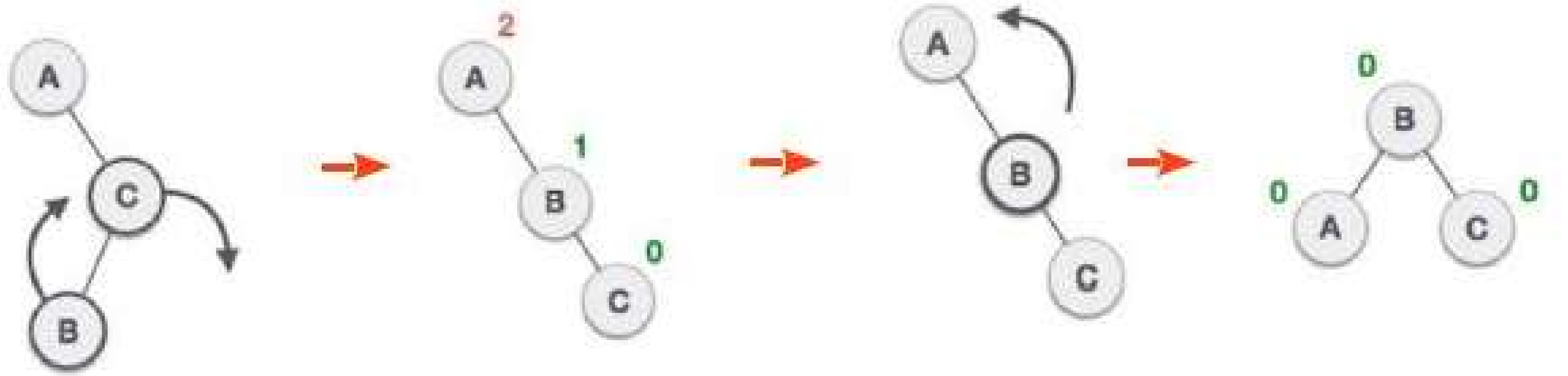


Single Rotation in an AVL Tree



Right-Left Rotation

A node has been inserted into left subtree of right subtree. This makes **A** an unbalanced node, with balance factor 2.



Thank you