Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Innopolis University

Binary Search Trees

Binary Search Trees

- A special type of binary tree
  - it represents information in an ordered format
  - A binary search tree is a binary tree in which every node holds a value > every value in its left subtree and < every value in its right subtree

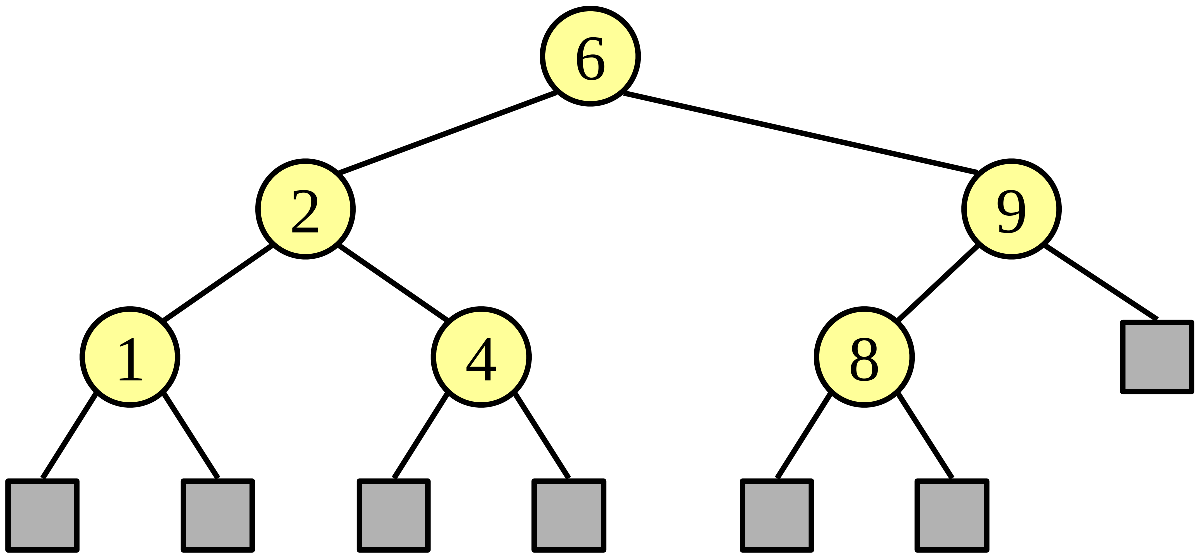
Assuming no duplicates are allowed

Binary Search Trees

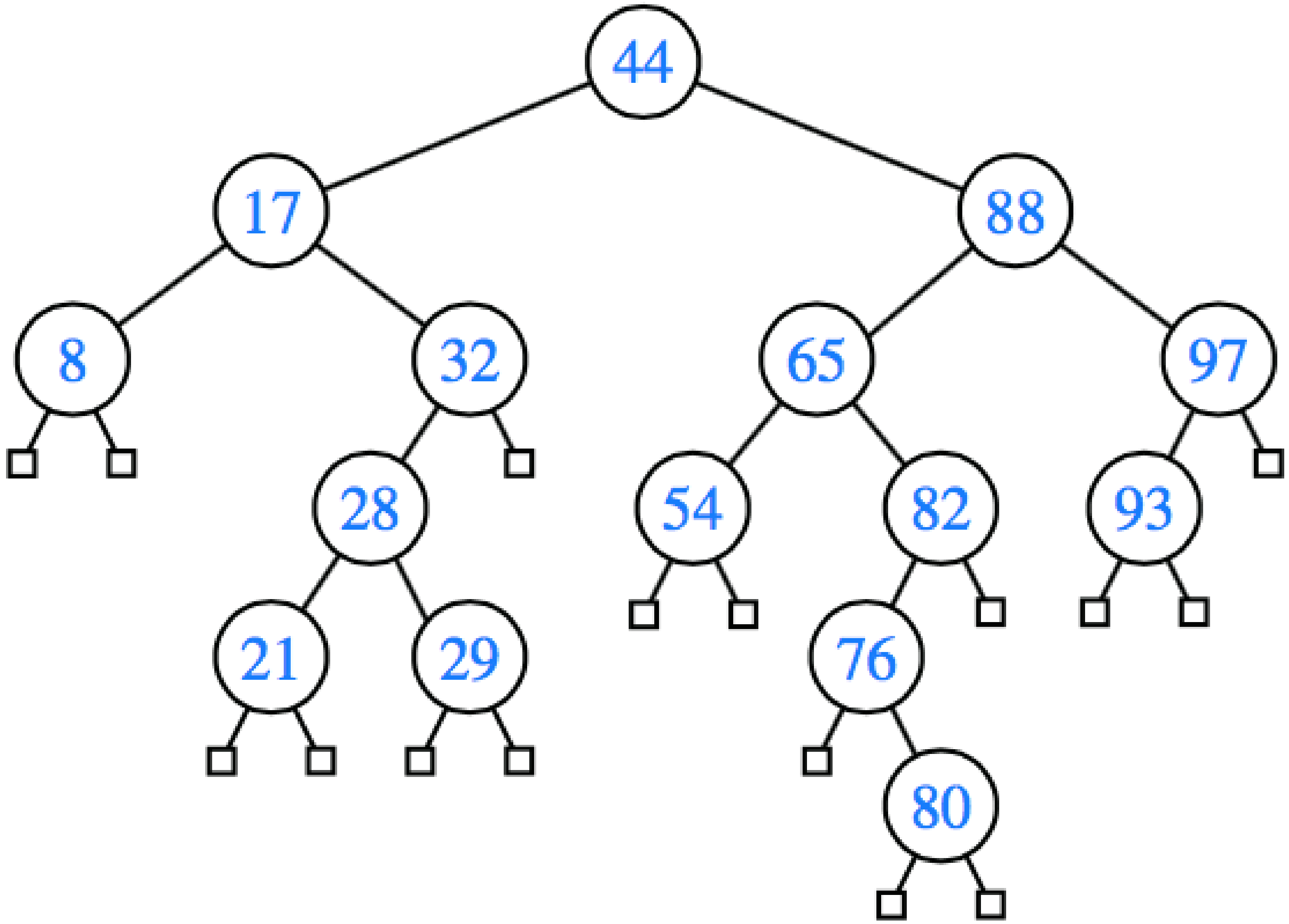
- Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have

key(u) < key(v) < key(w)

- An inorder traversal of a binary search trees visits the keys in increasing order



Example



- What is in the leftmost node?
- What is in the rightmost node?

BST Operations

- A binary search tree is a special case of a binary tree
  - So, it has all the operations of a binary tree

- It also has operations specific to a BST:
  - add an element (requires that the BST property be maintained)
  - remove an element (requires that the BST property be maintained)
  - Remove/find the maximum element
  - Remove/find the minimum element

Searching in a BST

- Why is it called a binary search tree?

  - Data is stored in such a way, that it can be more efficiently found than in an ordinary binary tree

Searching in a BST

- Algorithm to search for an item in a BST

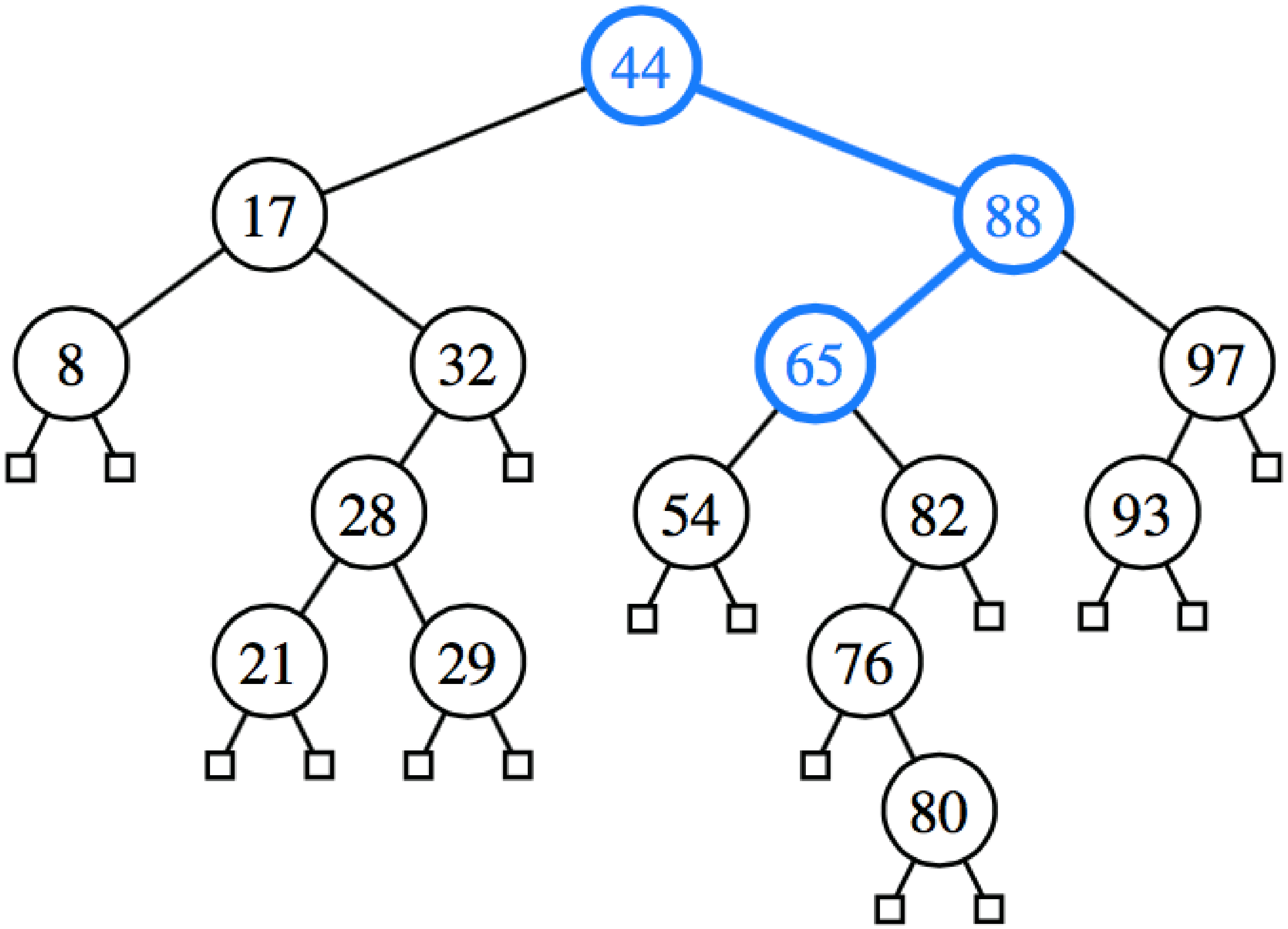
  - Compare data item to the root of the (sub)tree
  - If data item = data at root, found
  - If data item < data at root, go to the left; if there is no left child, data item is not in tree
  - If data item > data at root, go to the right; if there is no right child, data item is not in tree

Searching in a BST

Algorithm TreeSearch(p, k):
  if p is external **then**
    return p {unsuccessful search}
  else if k == key(p) **then**
    return p {successful search}
  else if $k <$ key(p) **then**
    return TreeSearch(left(p), k) {recur on left subtree}
  else {we know that $k >$ key(p)}
    return TreeSearch(right(p), k) {recur on right subtree}

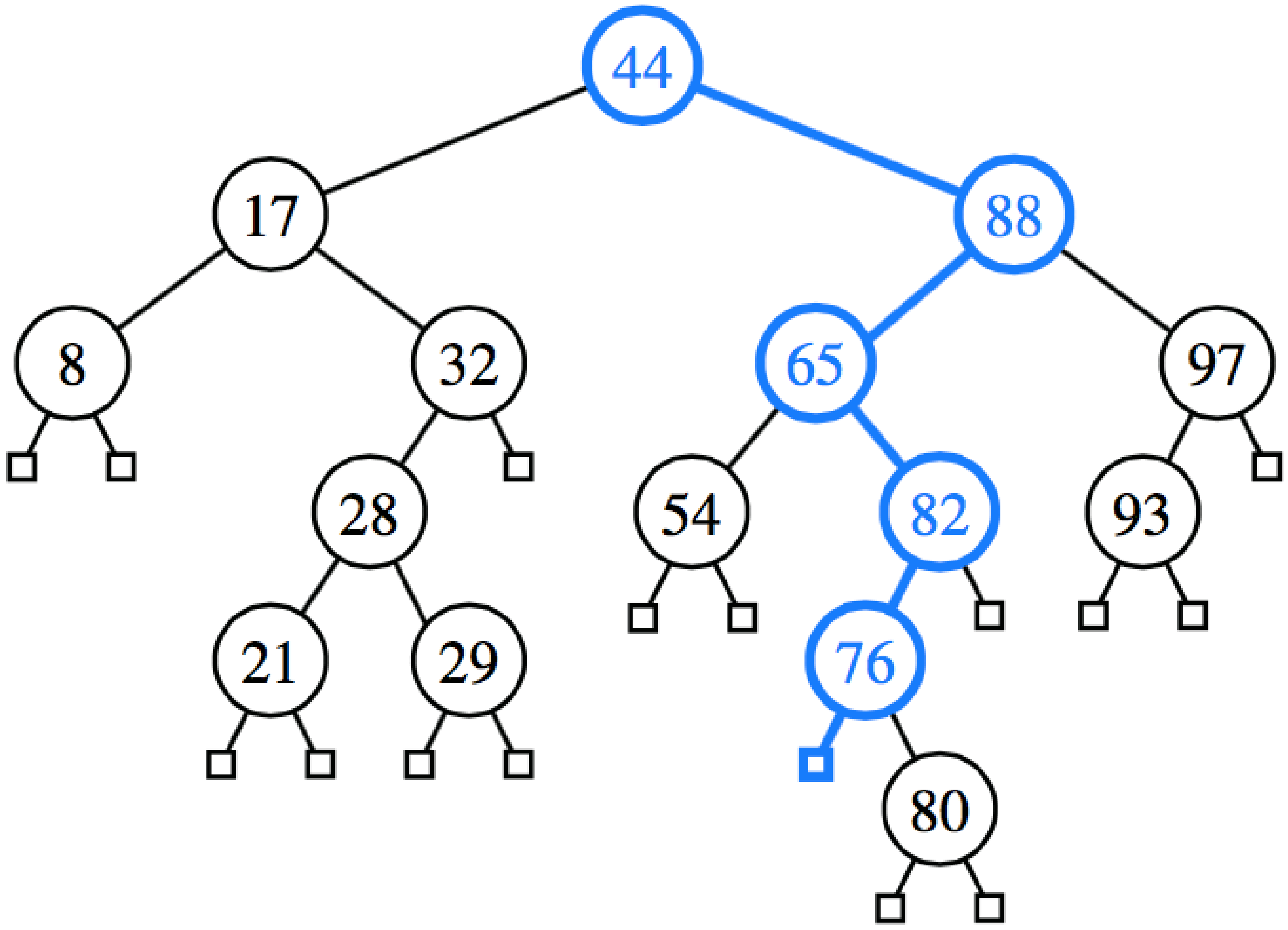
Code Fragment 11.1: Recursive search in a binary search tree.

Searching in a BST



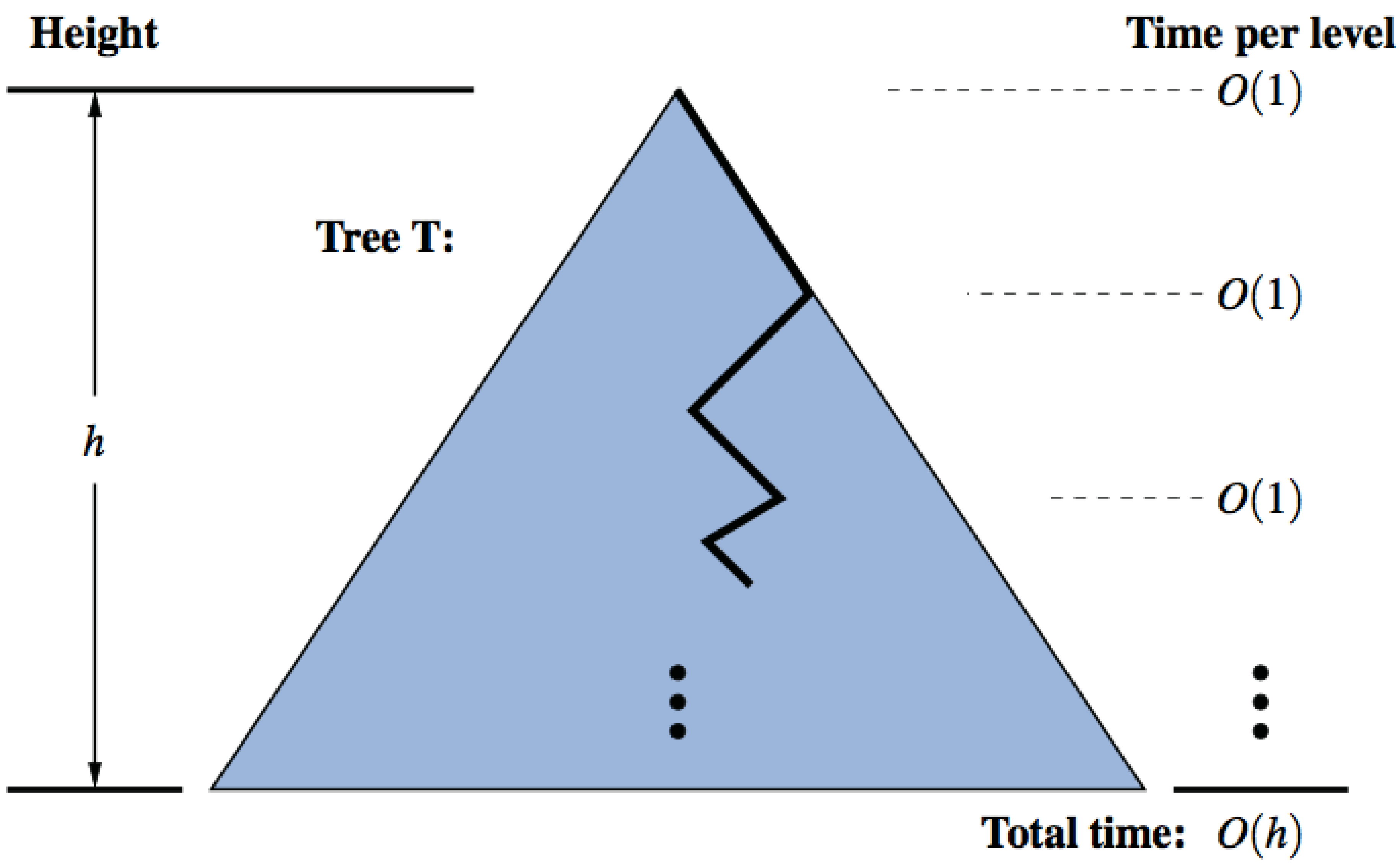
A successful search for key 65 in a binary search tree

Searching in a BST



A successful search for key 68 that terminates at the leaf to the left of key 76

Analysis of BST Searching



Insertions in a BST

- To ***add*** an item to a BST:
  - Follow the algorithm for searching, until there is no child
  - Insert at that point

- So, new node will be added as a leaf

- (We are assuming no duplicates allowed)

Insertions in a BST

Algorithm TreeInsert(k, v):

  Input: A search key k to be associated with value v

  p = TreeSearch(root(), k)

  if k == key(p) **then**

    Change p's value to (v)

  else

    expandExternal(p, (k, v))

Code Fragment 11.2: Algorithm for inserting a key-value pair into a map that is represented as a binary search tree.

Insertions in a BST

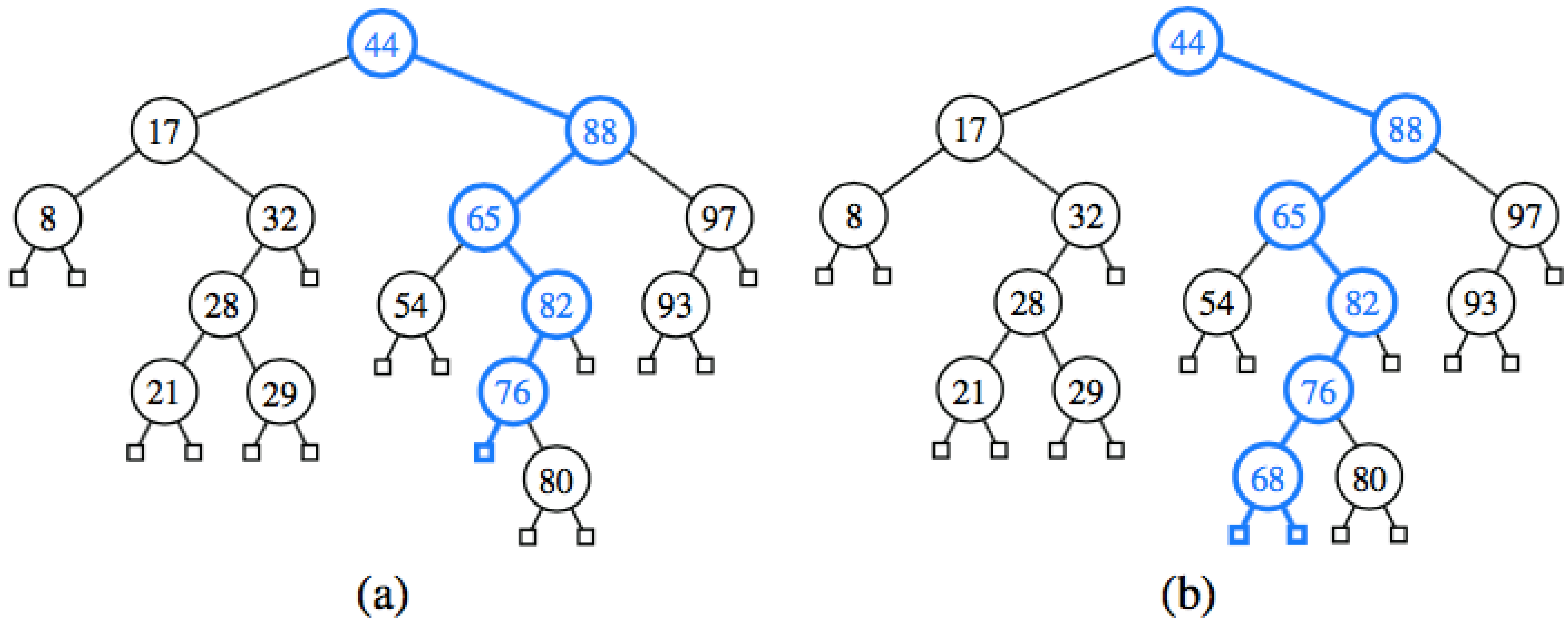


Figure 11.4: Insertion of an entry with key 68 into the search tree of Figure 11.2. Finding the position to insert is shown in (a), and the resulting tree is shown in (b).

Deletions in a BST

method remove (key)

I if the tree is empty return false

II Attempt to locate the node containing the target using the binary
search algorithm
if the target is not found return false
else the target is found, so remove its node:

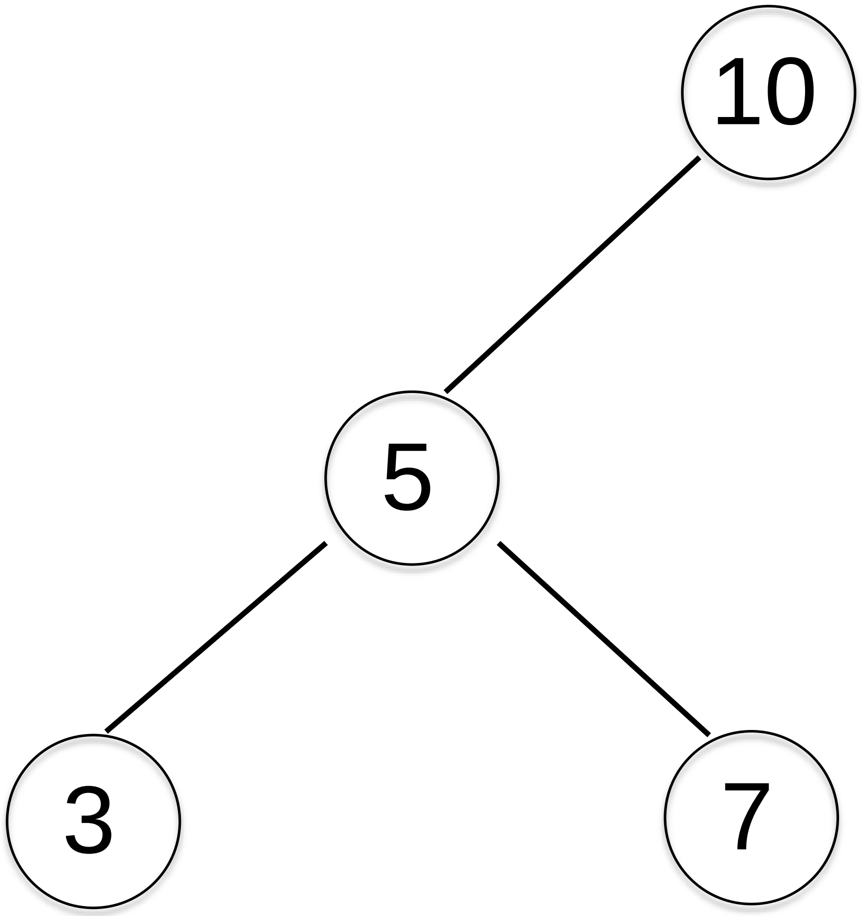
// Now there can be 4 cases

// The easiest case, the node has no children – is a leaf
Case 1: if the node has 2 empty subtrees
replace the link in the parent with null

Deletions in a BST

// The easiest case, the node has no children – is a leaf

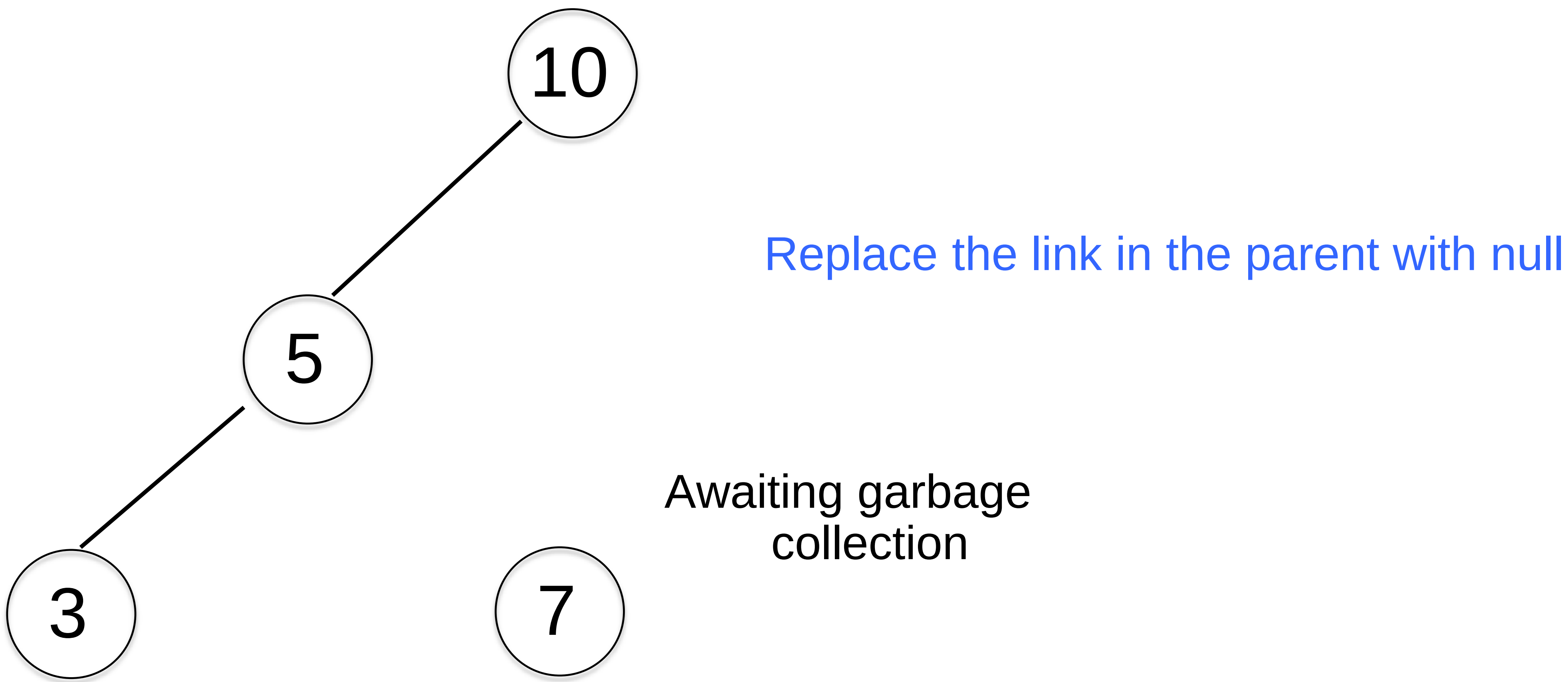
<u>Case 1:</u> the node has no children



Let’s delete the node with key 7

Deletions in a BST

// **Case 1:** **the node has no children**



Deletions in a BST

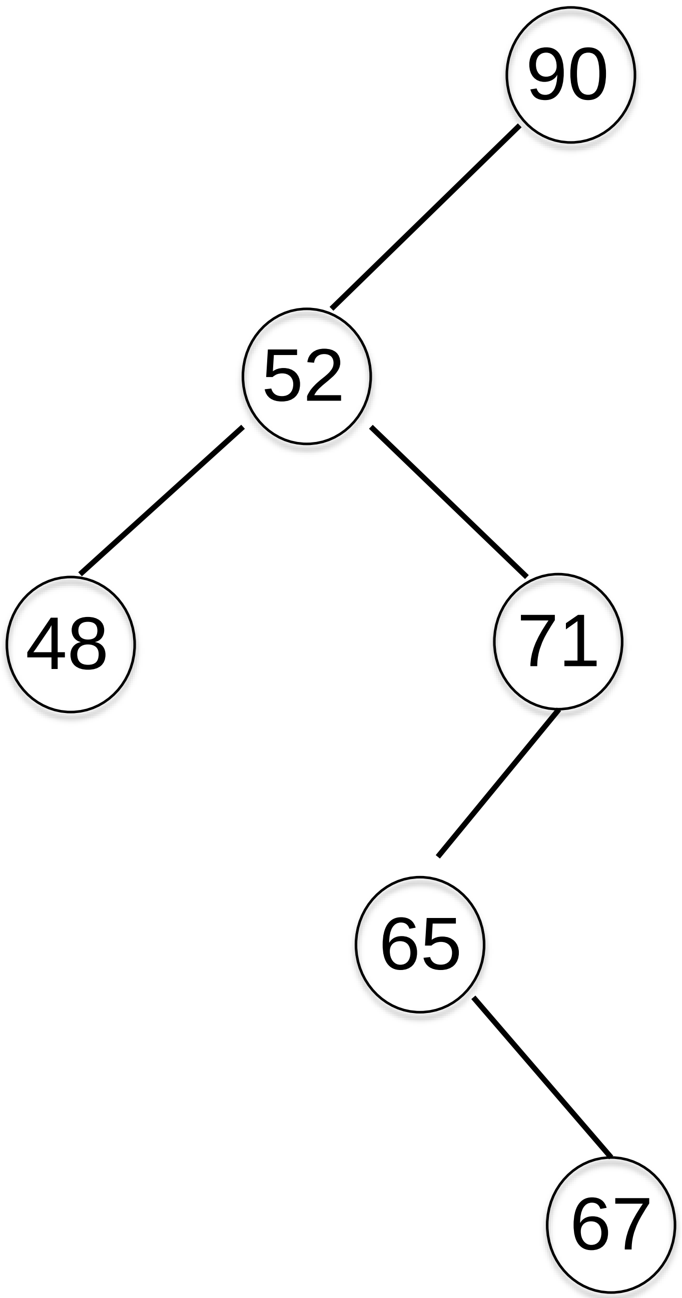
// Next are the cases with one child

Case 2: if the node has no left child
- link the parent of the node
- to the right (non-empty) subtree

Case 3: if the node has no right child
- link the parent of the target
- to the left (non-empty) subtree

Deletions in a BST

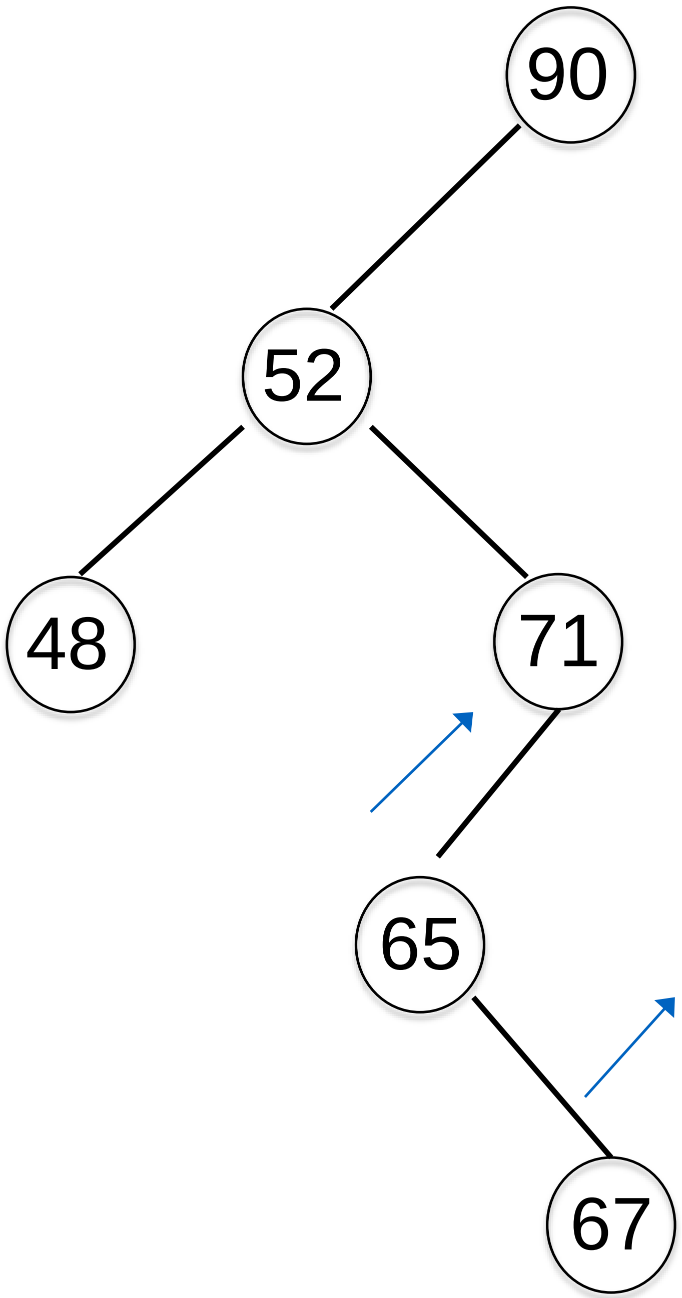
Case 2: **the node has only a left child**



Let’s delete the node with key 71

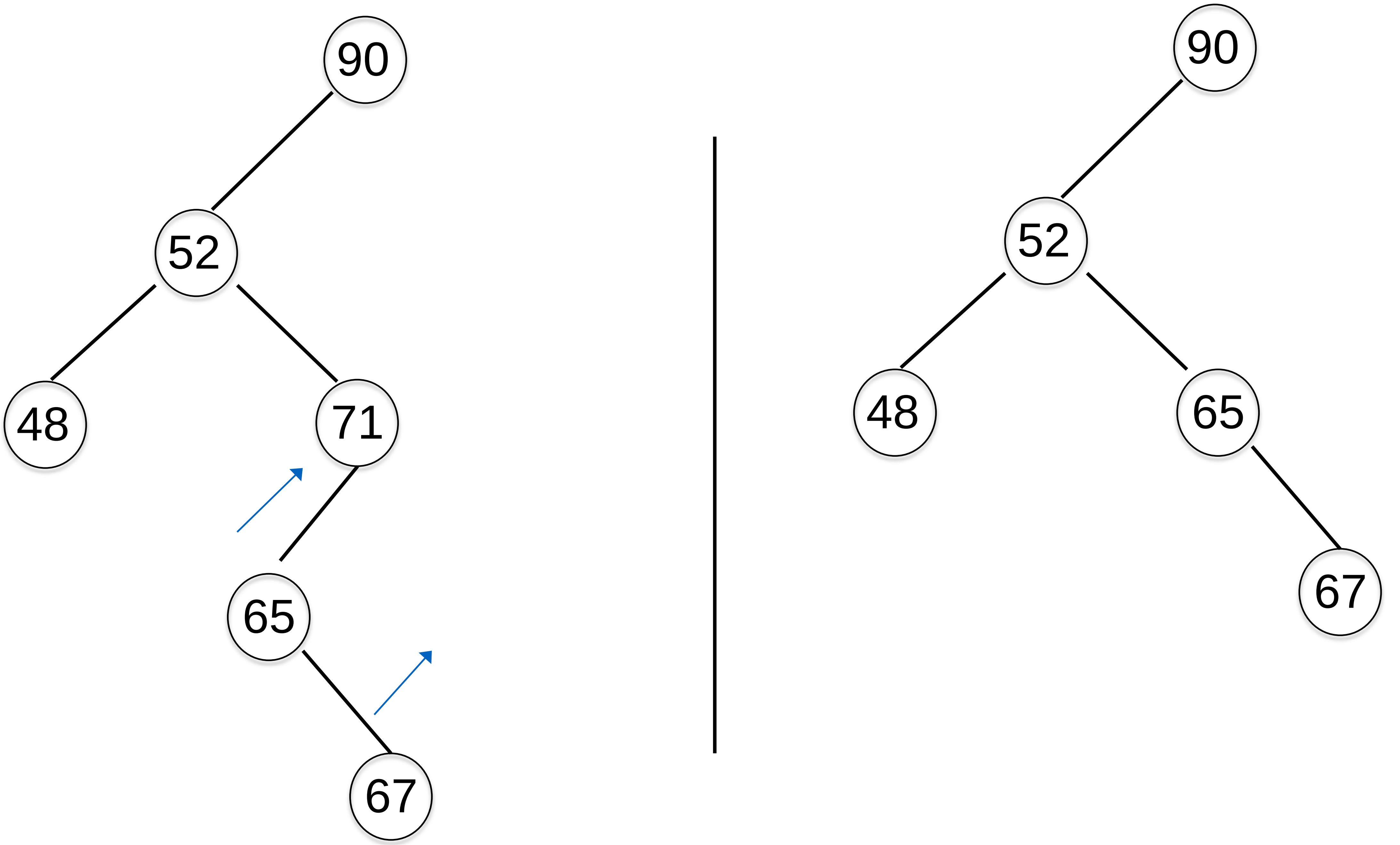
Deletions in a BST

Case 2: the node has only a left child



Deletions in a BST

Case 2: the node has only a left child

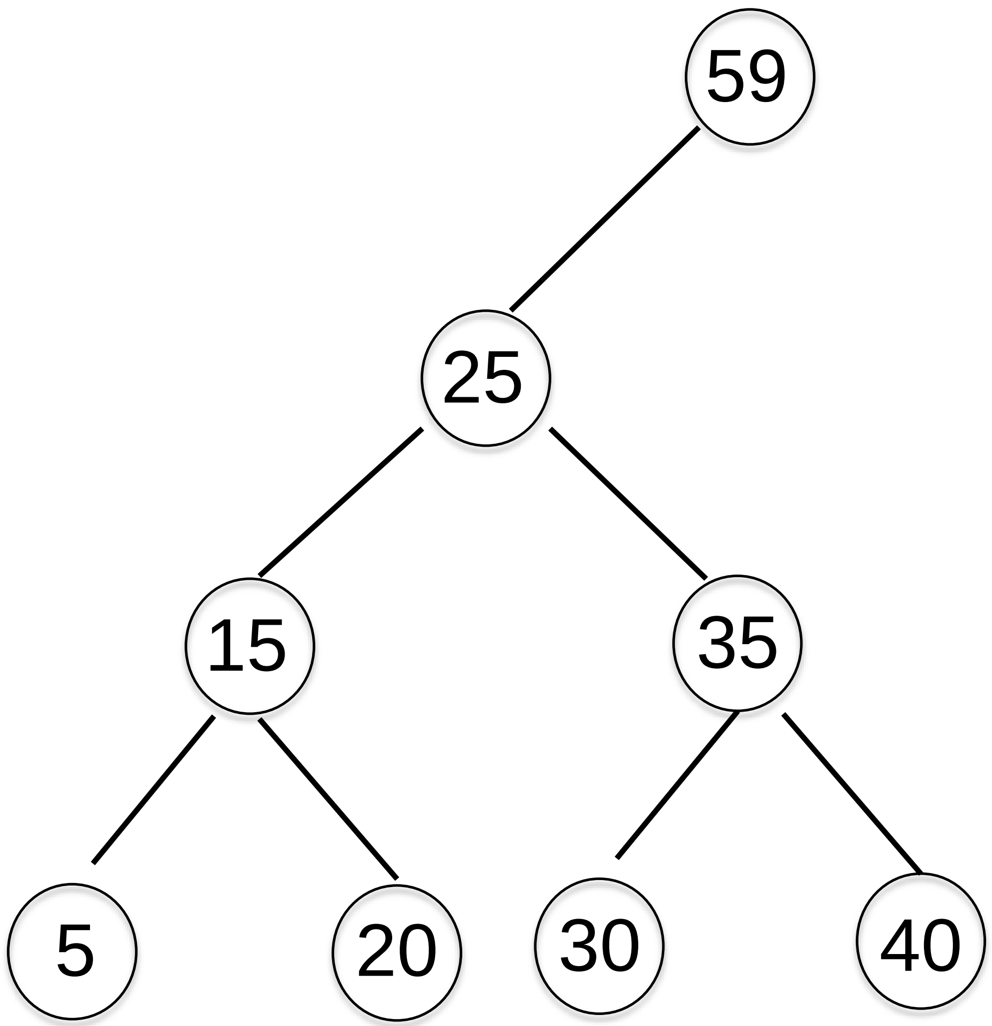


Deletions in a BST

// The most difficult case, the node has two children
// deleting this node will leave two children in trouble

Case 4: if the node has a two children

Deletions in a BST



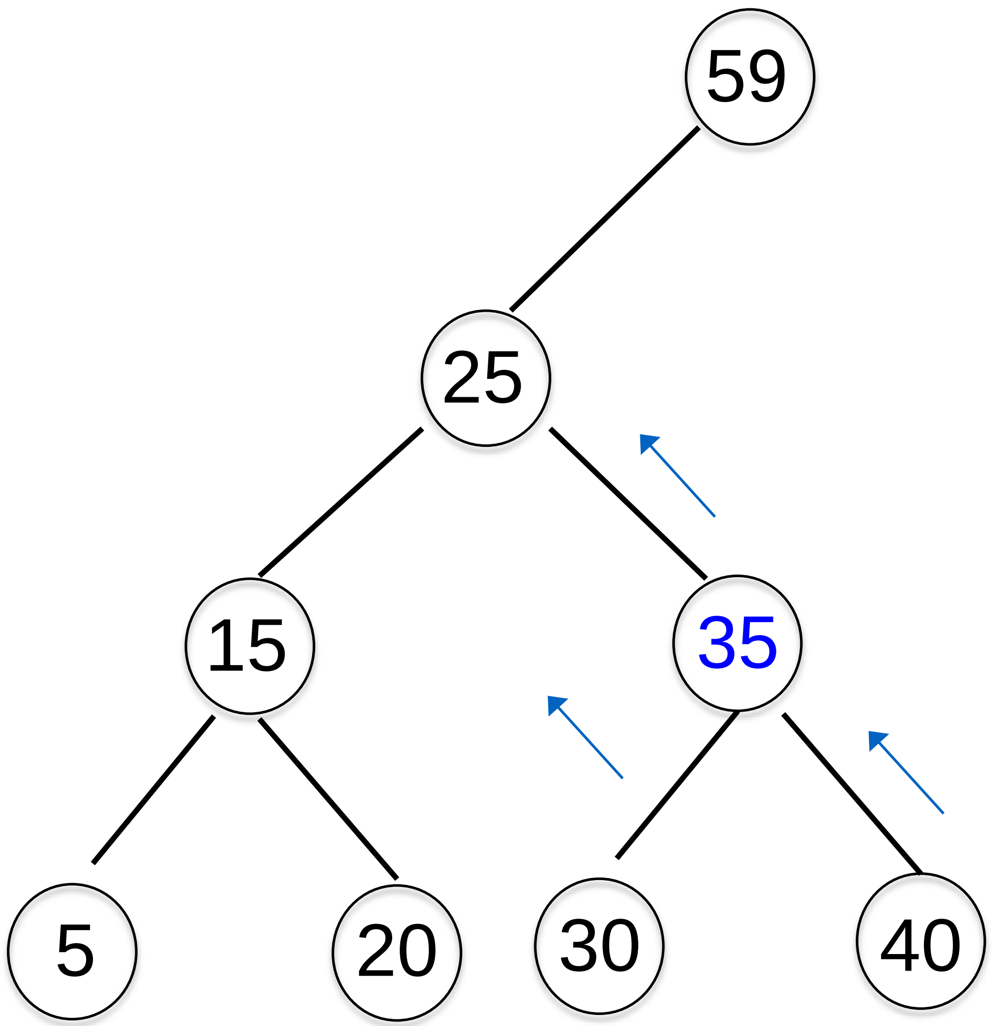
// Why is this a difficult case

Case 4: if the node has a two children

Let's delete 25 – Two choices

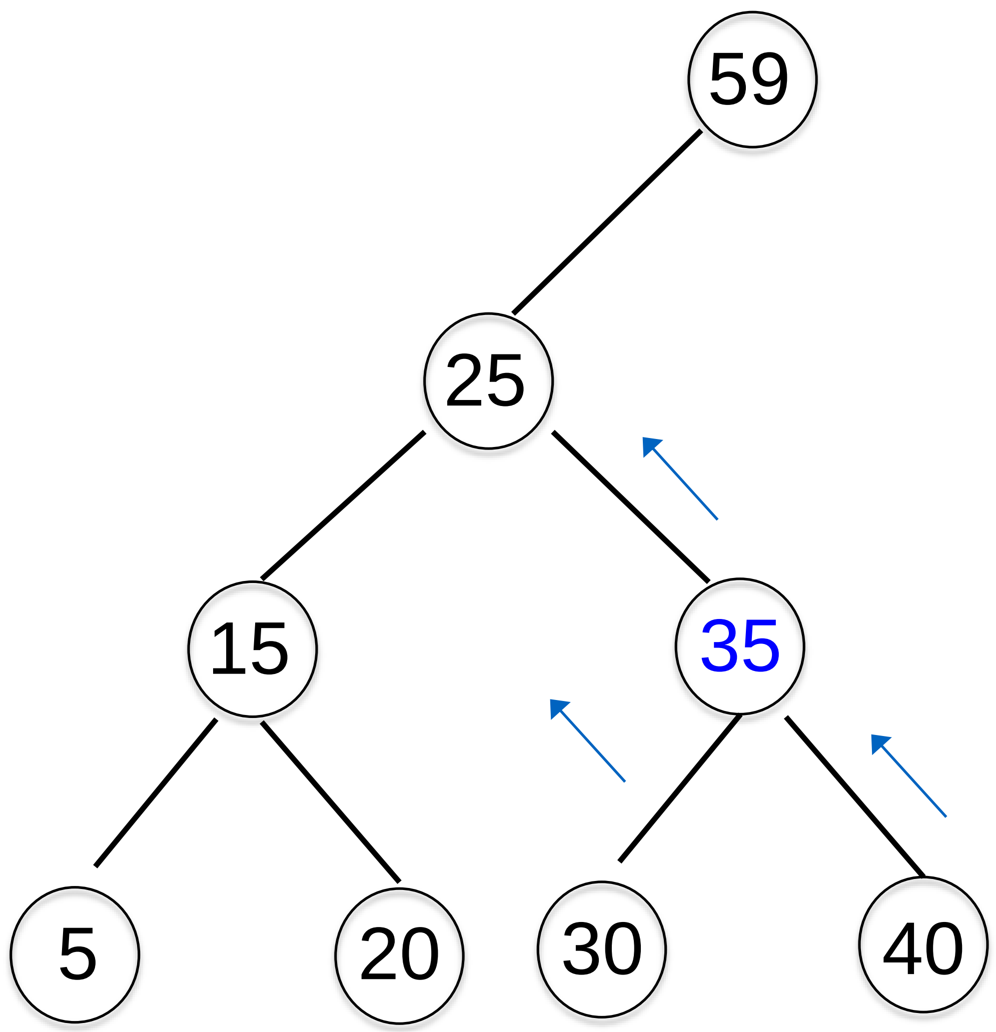
- Replace with root of left sub-tree
- Replace with the root of right sub-tree

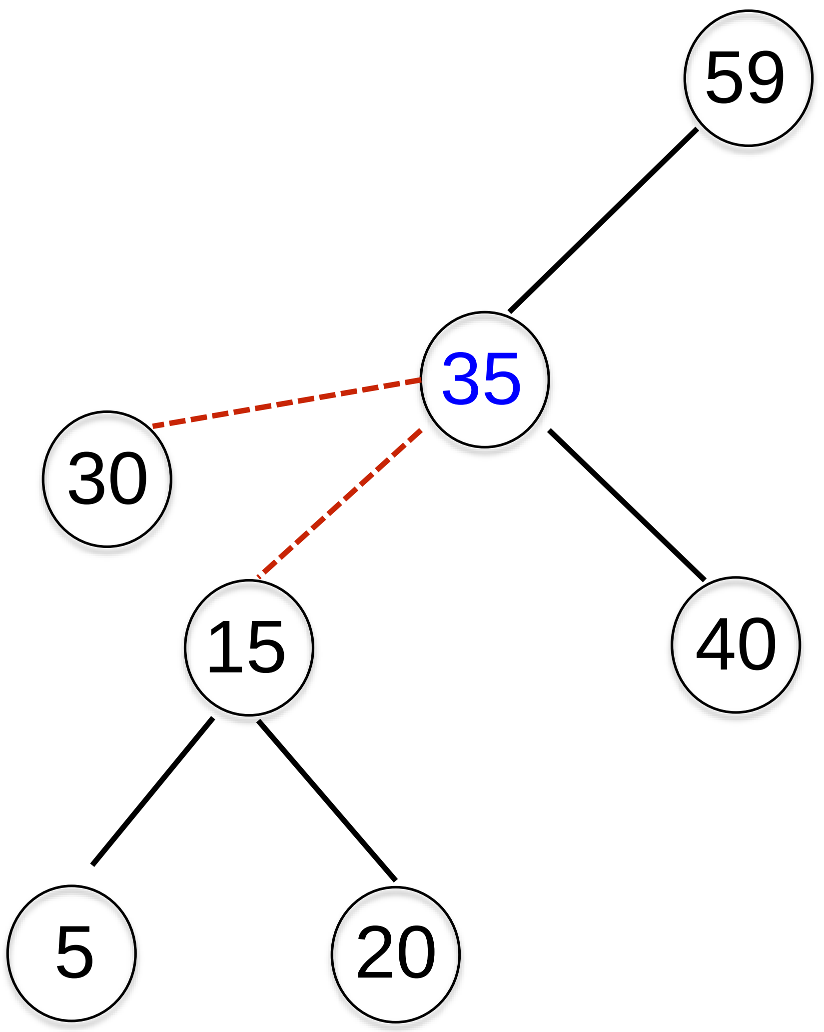
Deletions in a BST



- Replace with the root of right sub-tree

Deletions in a BST





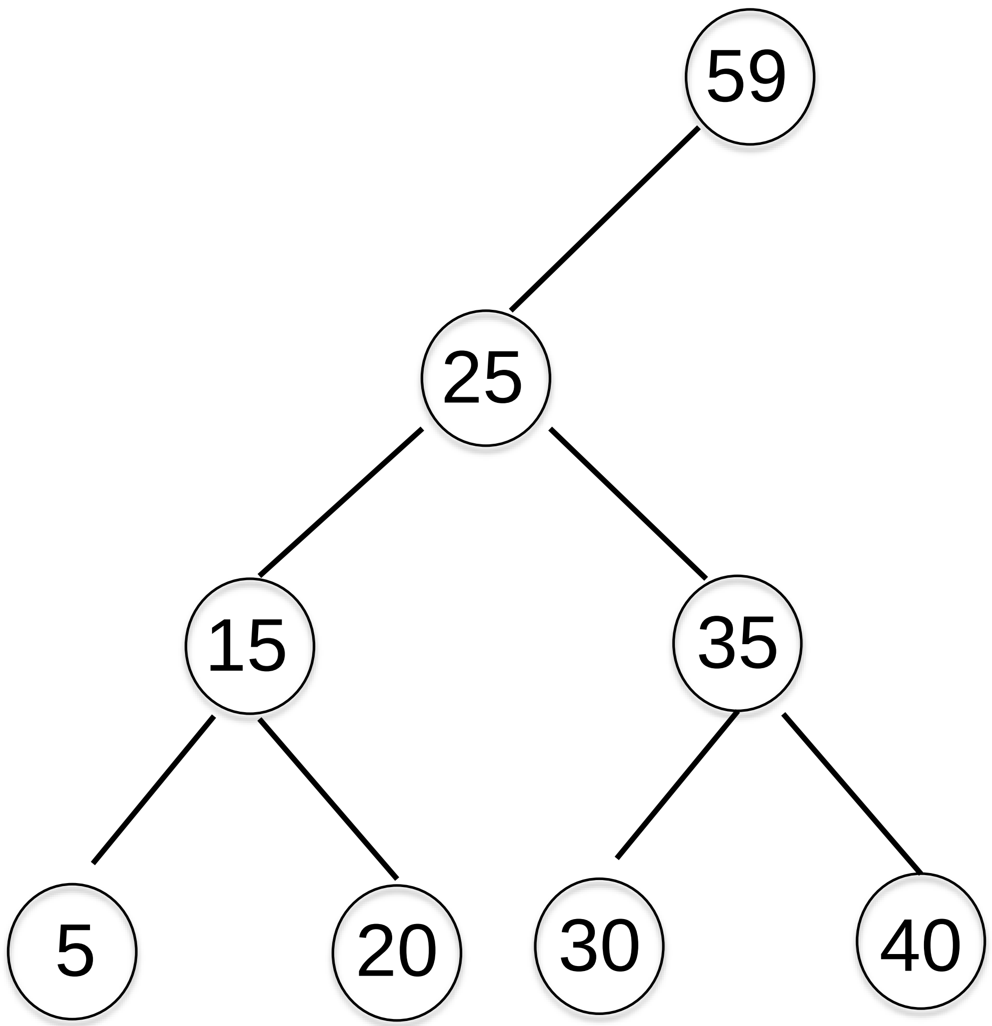
Deletions in a BST

// Thus a trick is needed

Case 4: if the node has a two children

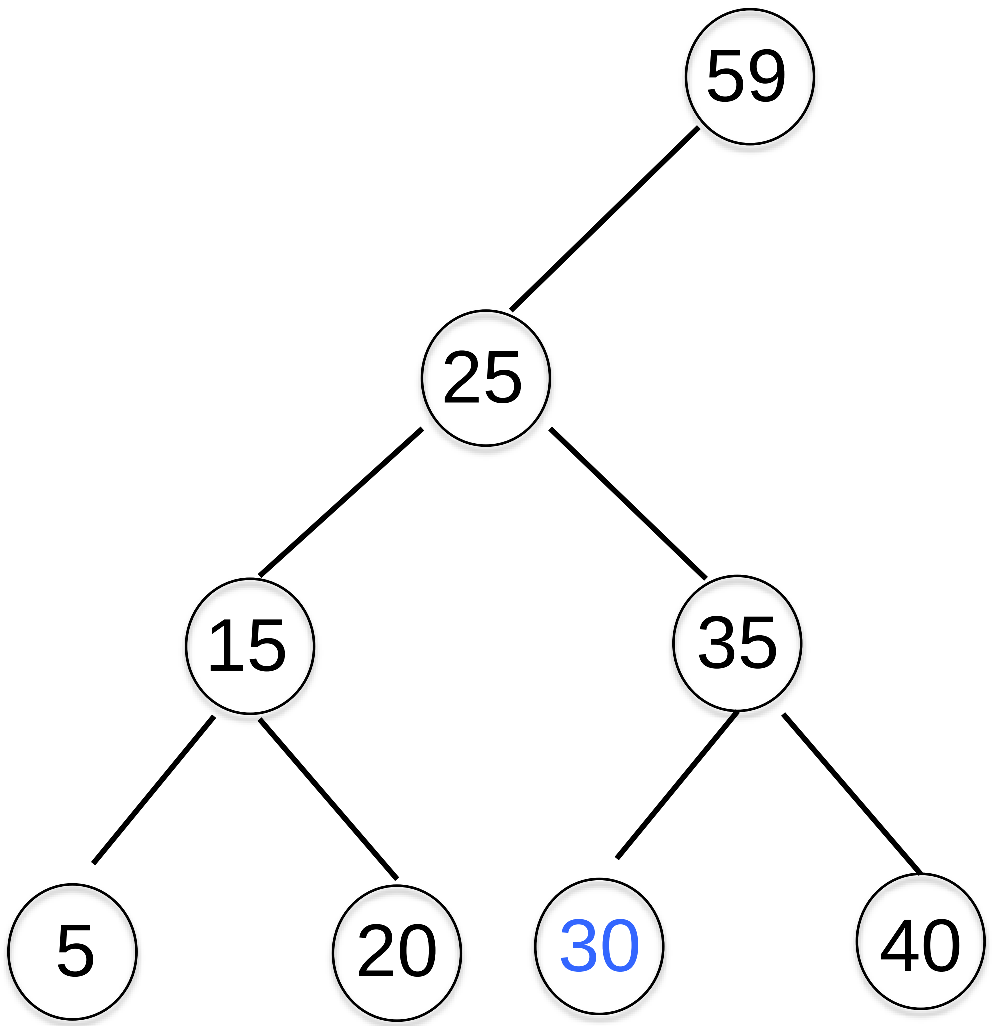
Replace the node with its **predecessor or successor from the inorder traversal** of the tree. and delete that node instead.

Inorder Successor



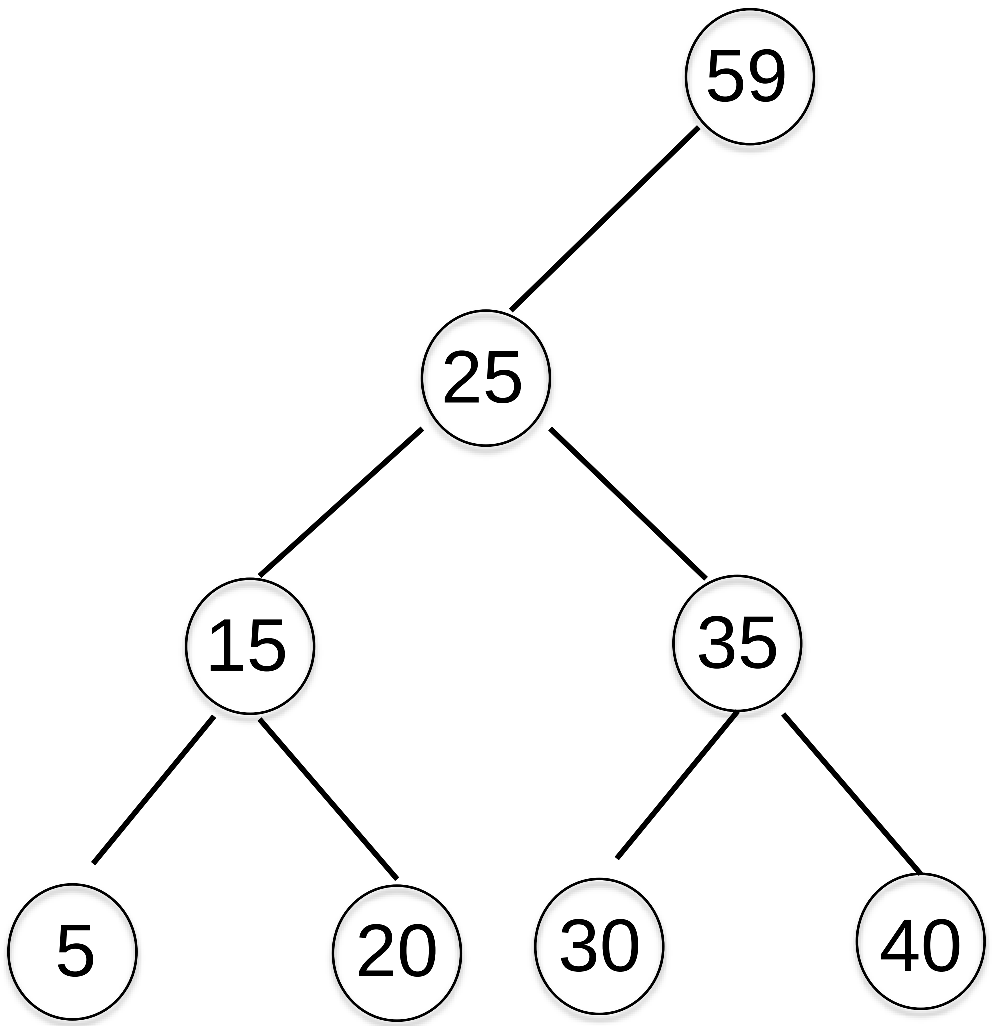
What is the in-order successor of node 25?

Inorder Successor



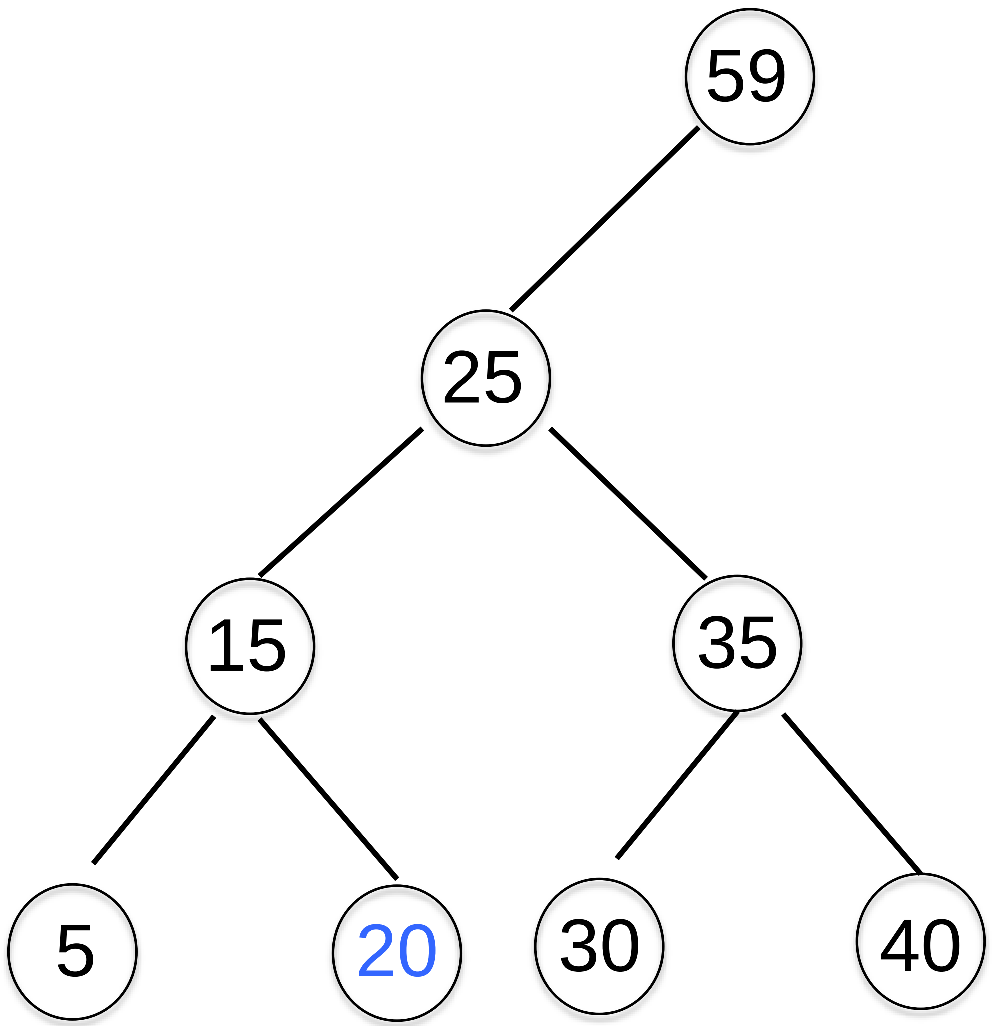
How to find it?

Inorder Predecessor



What is the in-order predecessor of node 25?

Inorder Predecessor



How to find it?

Deletions in a BST

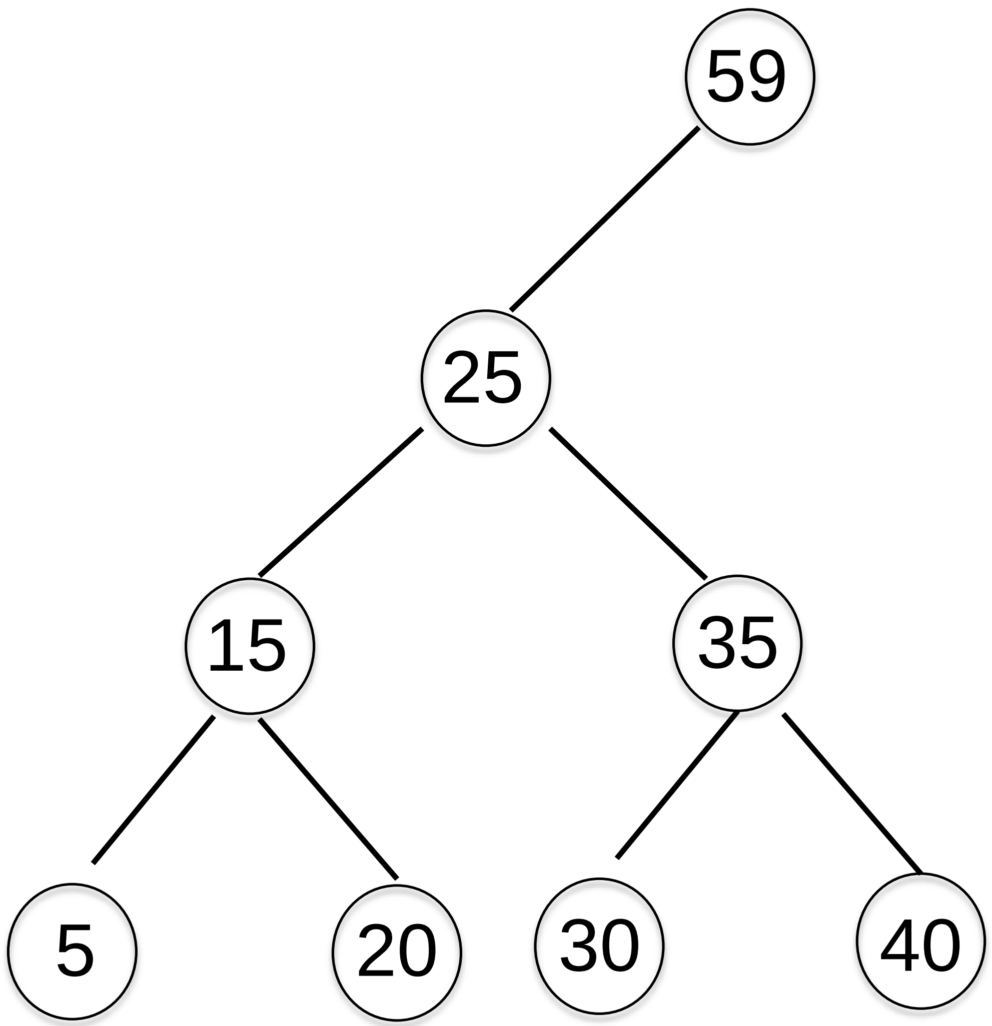
// Thus a trick is needed

<u>**Case 4:**</u> if the node has a two children

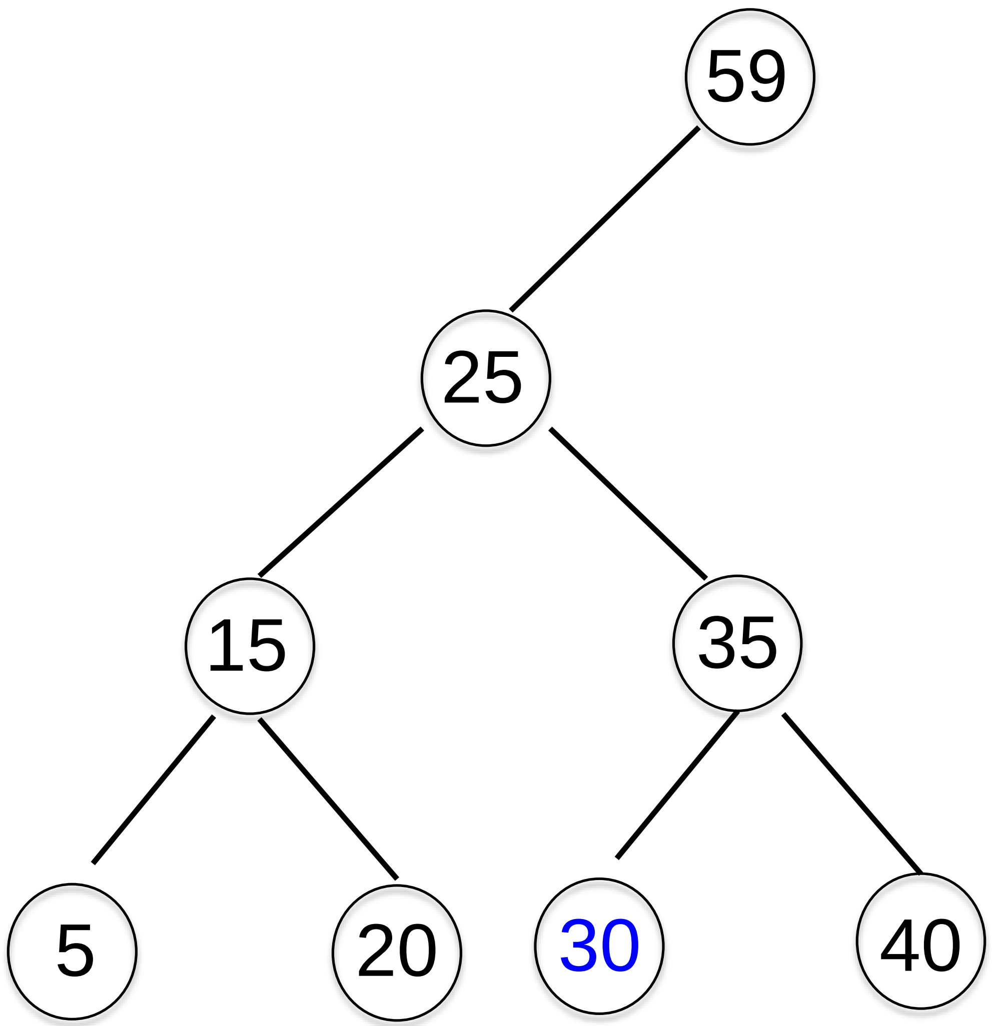
Replace the node with its **predecessor or successor from the inorder traversal** of the tree. and delete that node instead.

Deletions in a BST

Let's delete 25



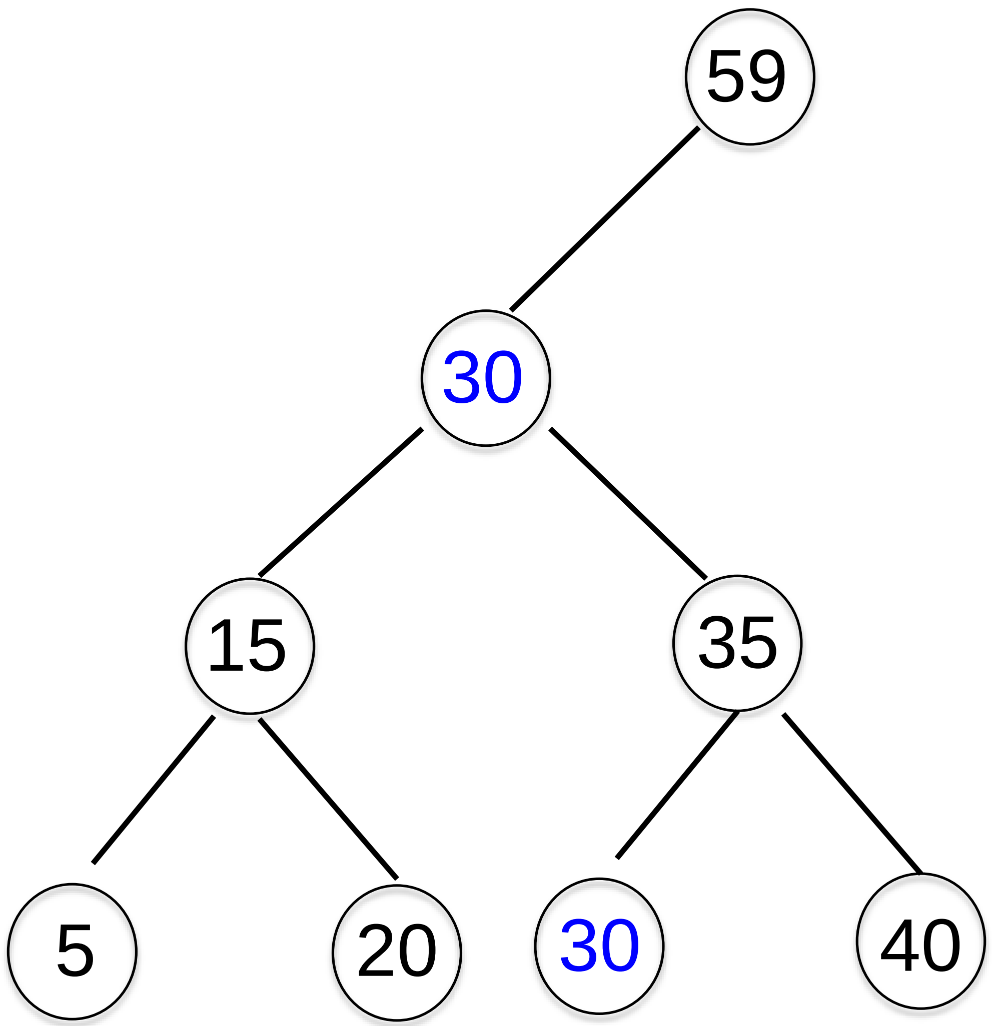
Deletions in a BST



Let's delete 25

- Identify its successor

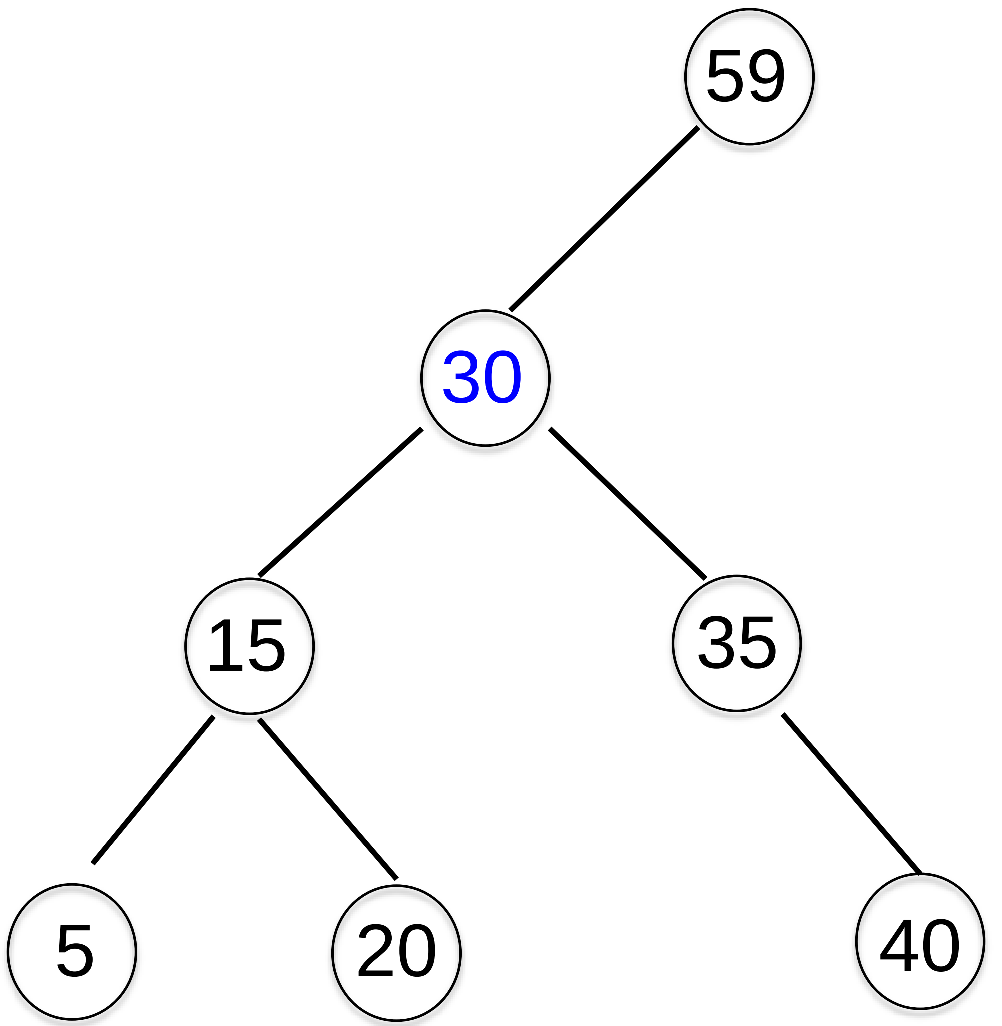
Deletions in a BST



Let's delete 25

- Replace it with its successor

Deletions in a BST



Let's delete 25

- Delete the successor

OR

Deletions in a BST

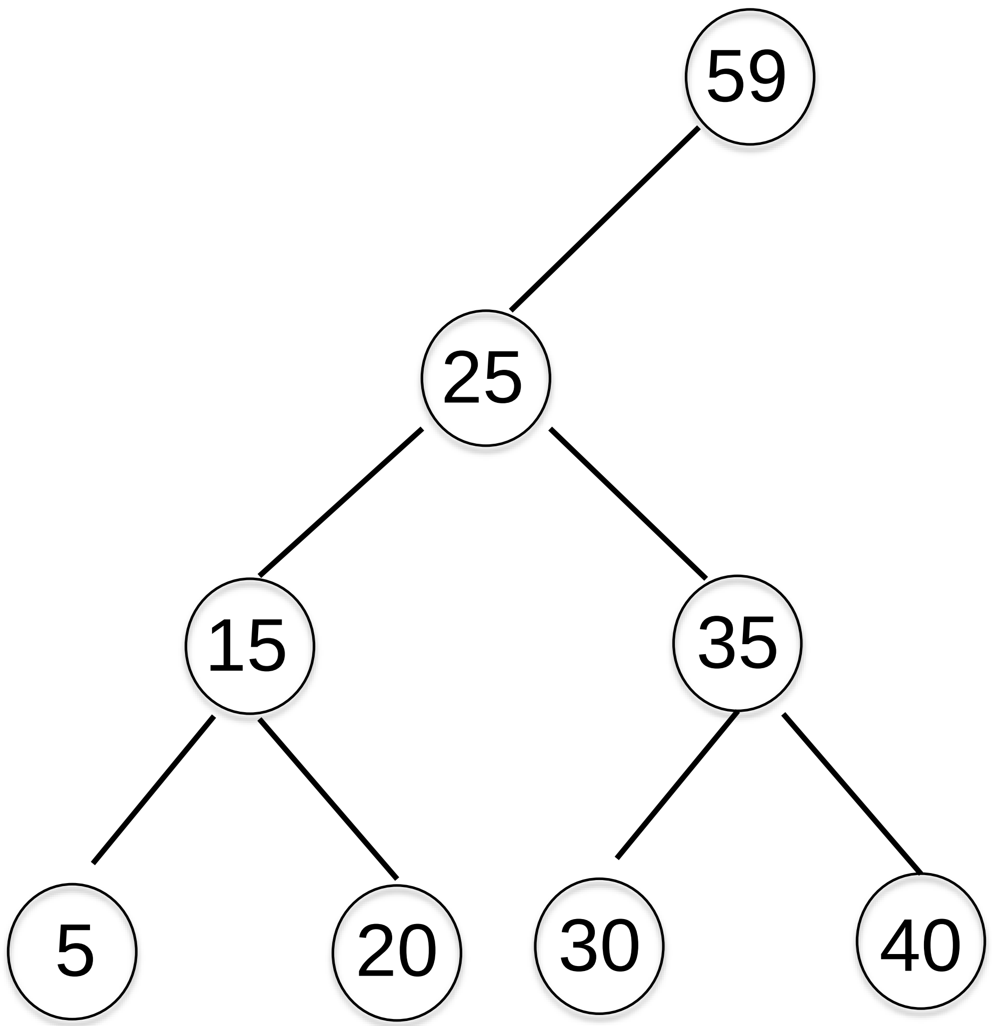
// Thus a trick is needed

Case 4: if the node has a two children

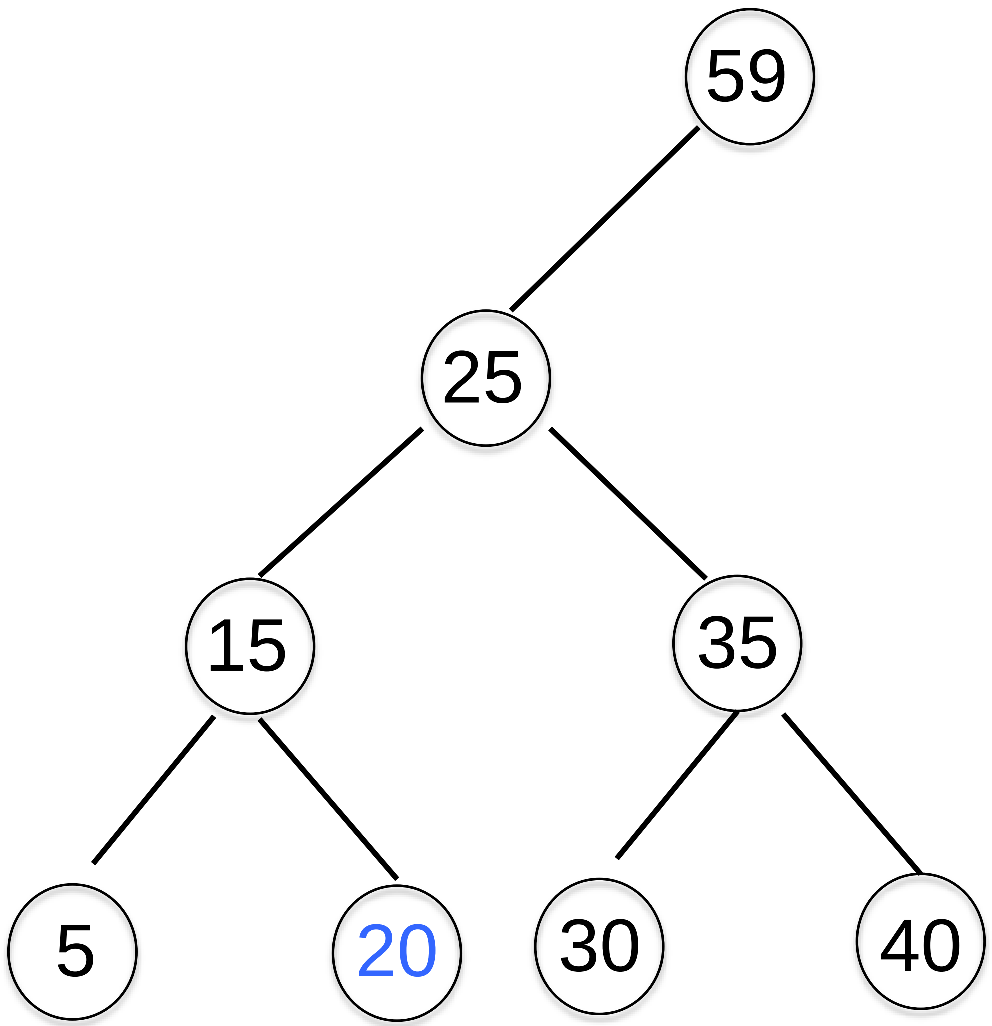
Replace the node with its **predecessor** **or successor from the inorder traversal** of the tree. and delete that node instead.

Deletions in a BST

Let's delete 25



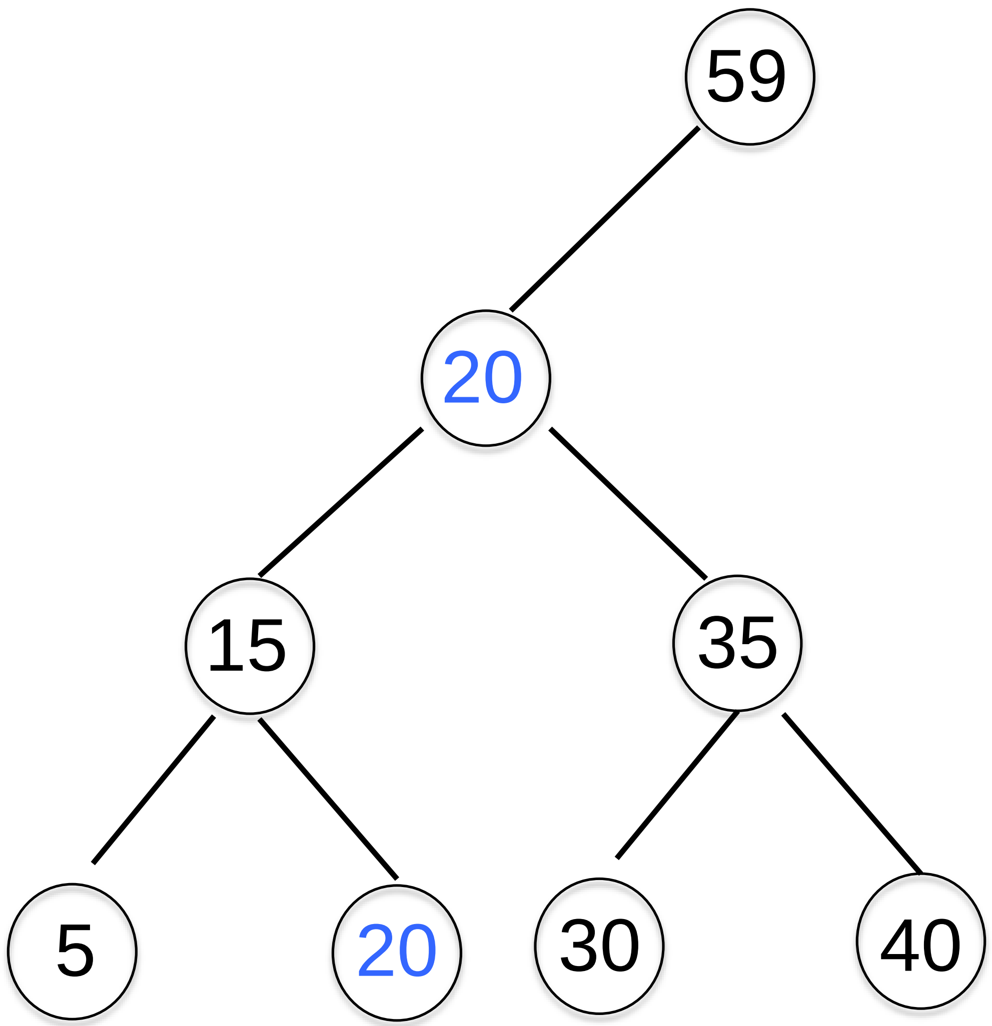
Deletions in a BST



Let's delete 25

- Identify its predecessor

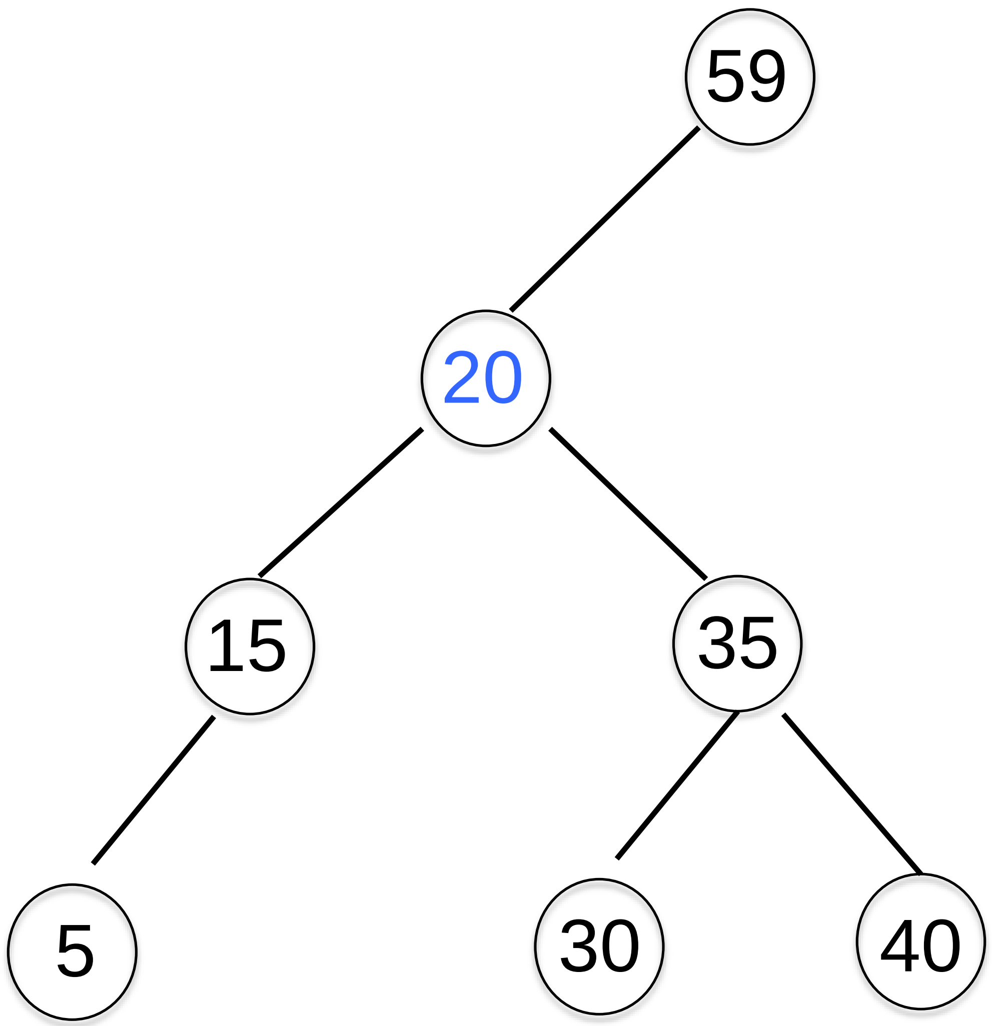
Deletions in a BST



Let's delete 25

- Replace the node with predecessor

Deletions in a BST



Let’s delete 25

- Delete the predecessor

Deletions in a BST

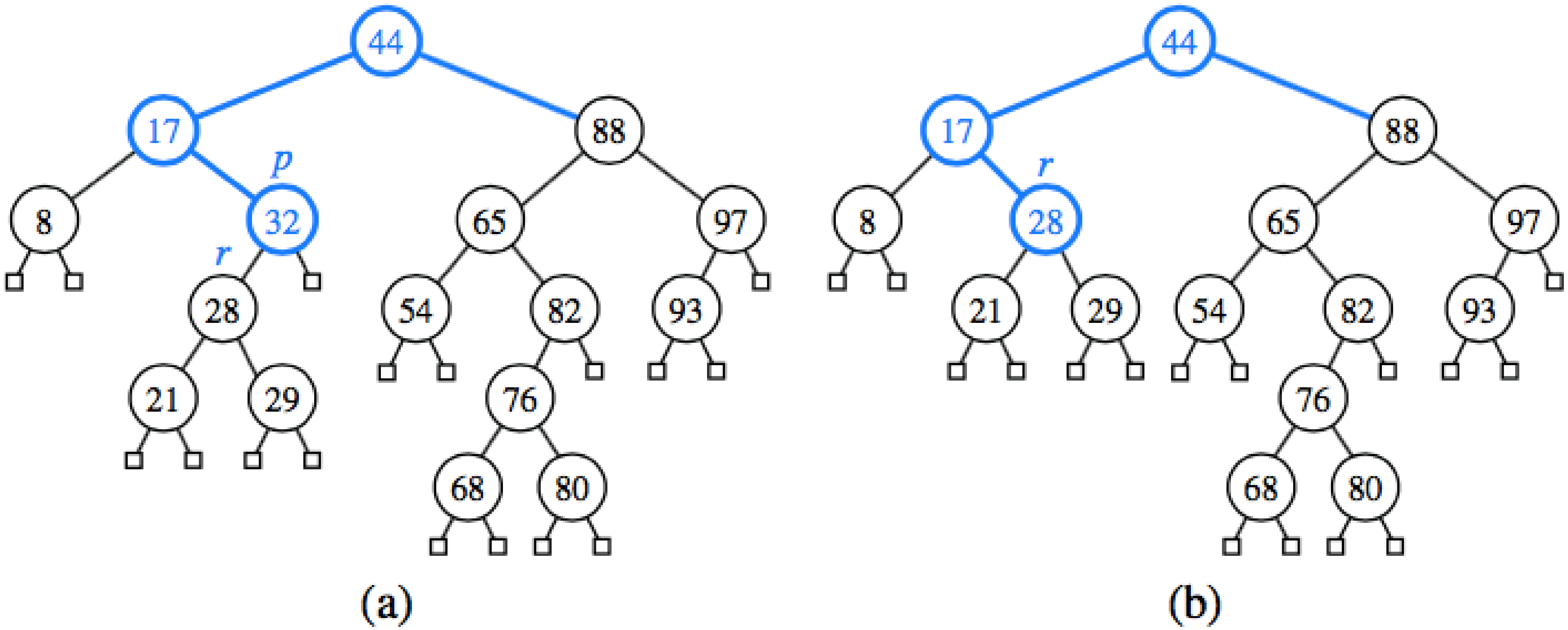


Figure 11.5: Deletion from the binary search tree of Figure 11.4b, where the entry to delete (with key 32) is stored at a position p with one child r: (a) before the deletion; (b) after the deletion.

Deletions in a BST

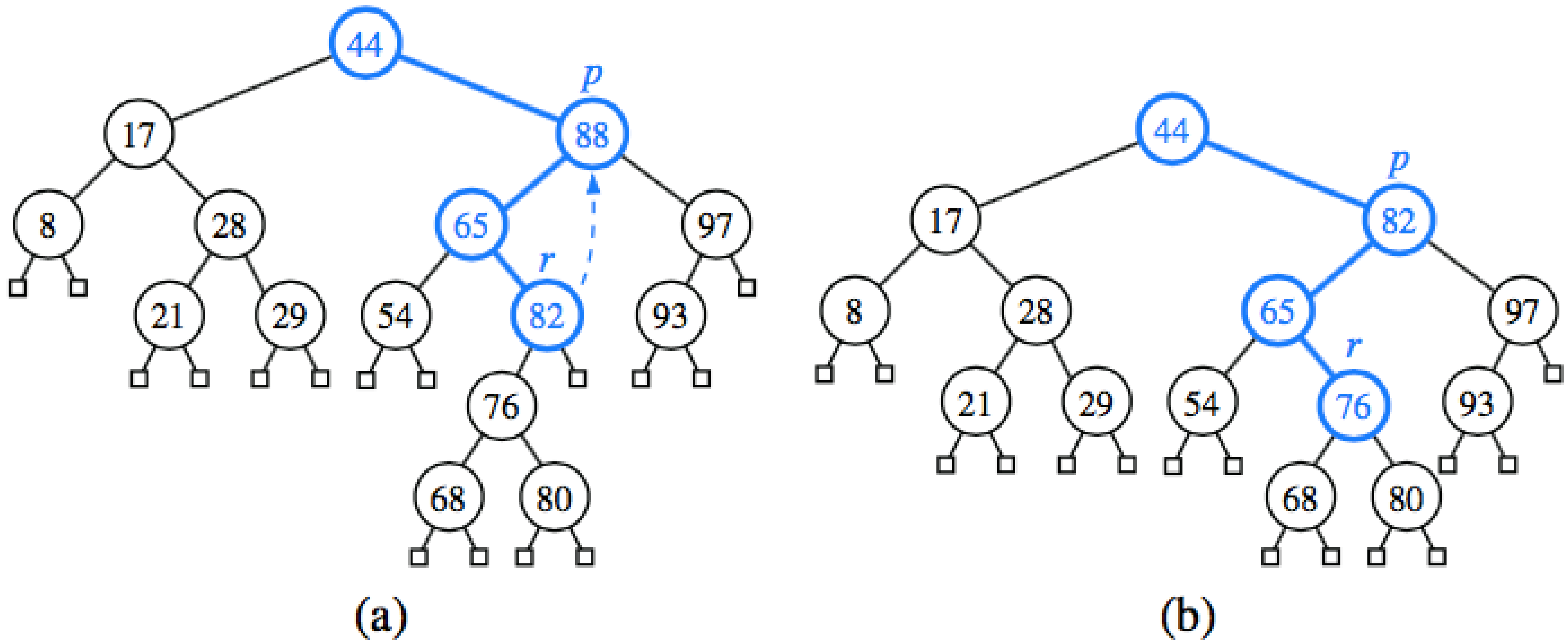
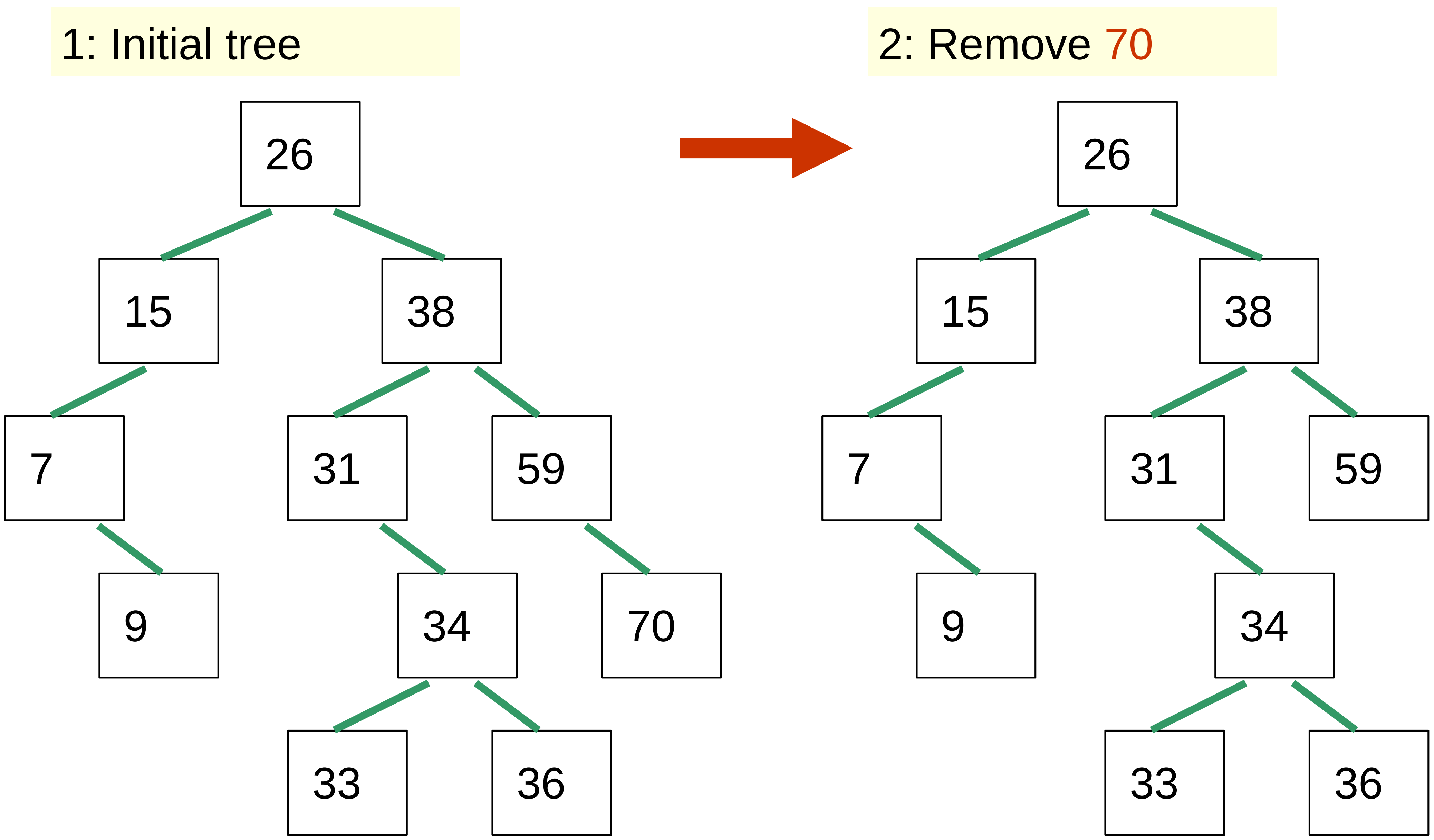
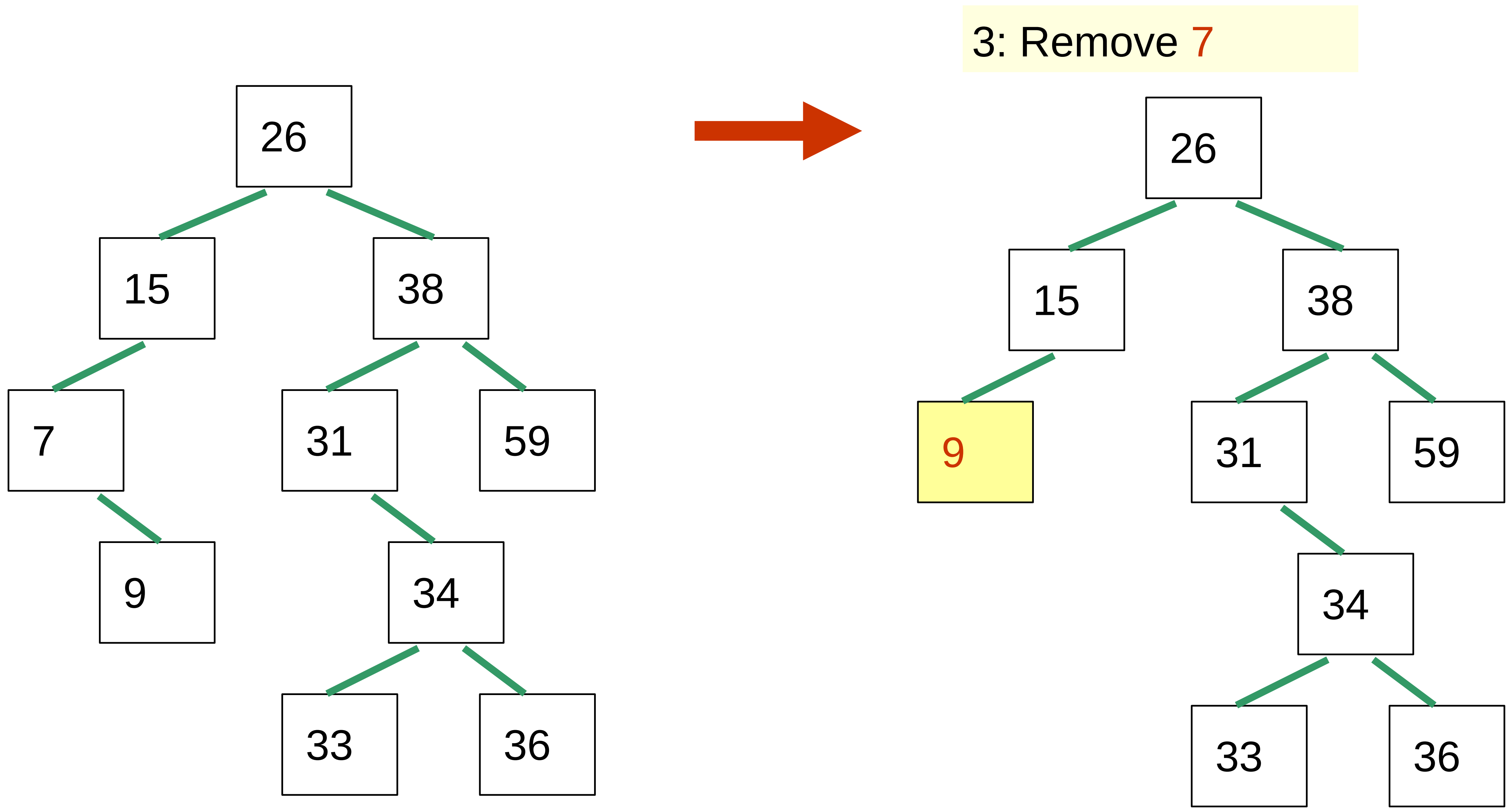


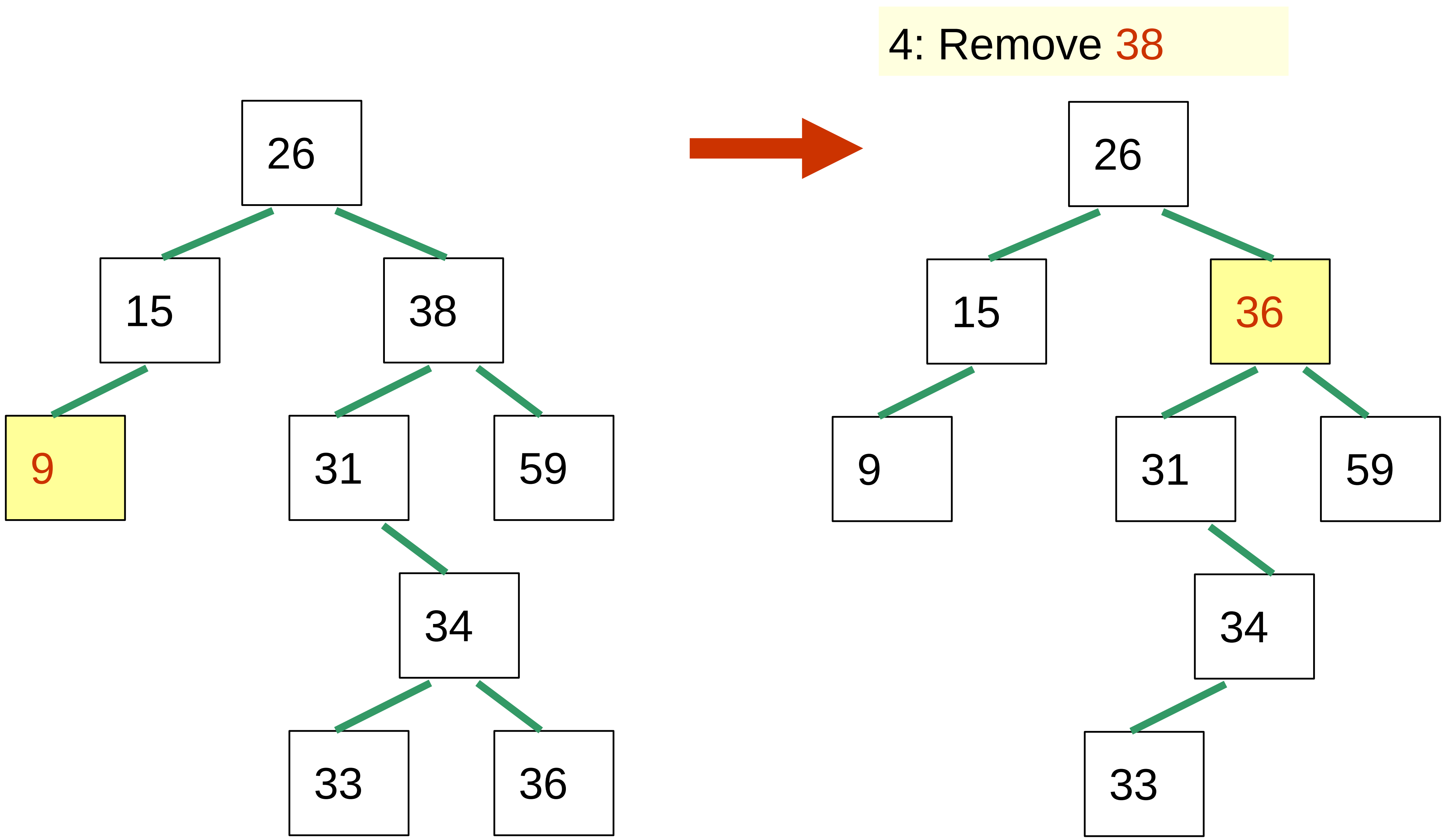
Figure 11.6: Deletion from the binary search tree of Figure 11.5b, where the entry to delete (with key 88) is stored at a position p with two children, and replaced by its predecessor r: (a) before the deletion; (b) after the deletion.

Example: Deleting BST Elements



Example: Deleting BST Elements




4: Remove 38
26
15
38
9
31
59
34
33
36
26
15
36
9
31
59
34
33

Implementation

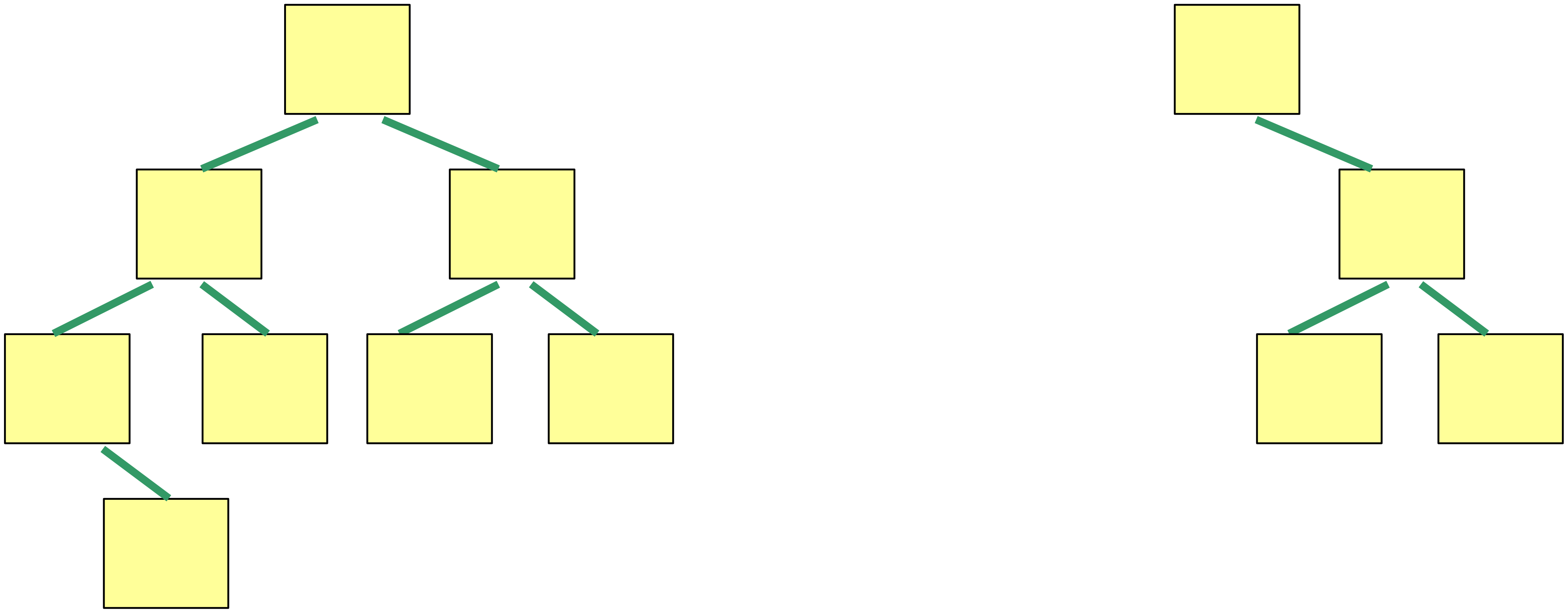
- Take the **LinkedBinaryTree** class from the previous lecture, and extend it to implement **LinkedBinarySearchTree**

Balanced Trees

- A balanced tree has the property that, for any node in the tree, the height of its left and right subtrees can differ by at most 1

  - Note that conventionally the height of an empty subtree is -1

Balanced Trees



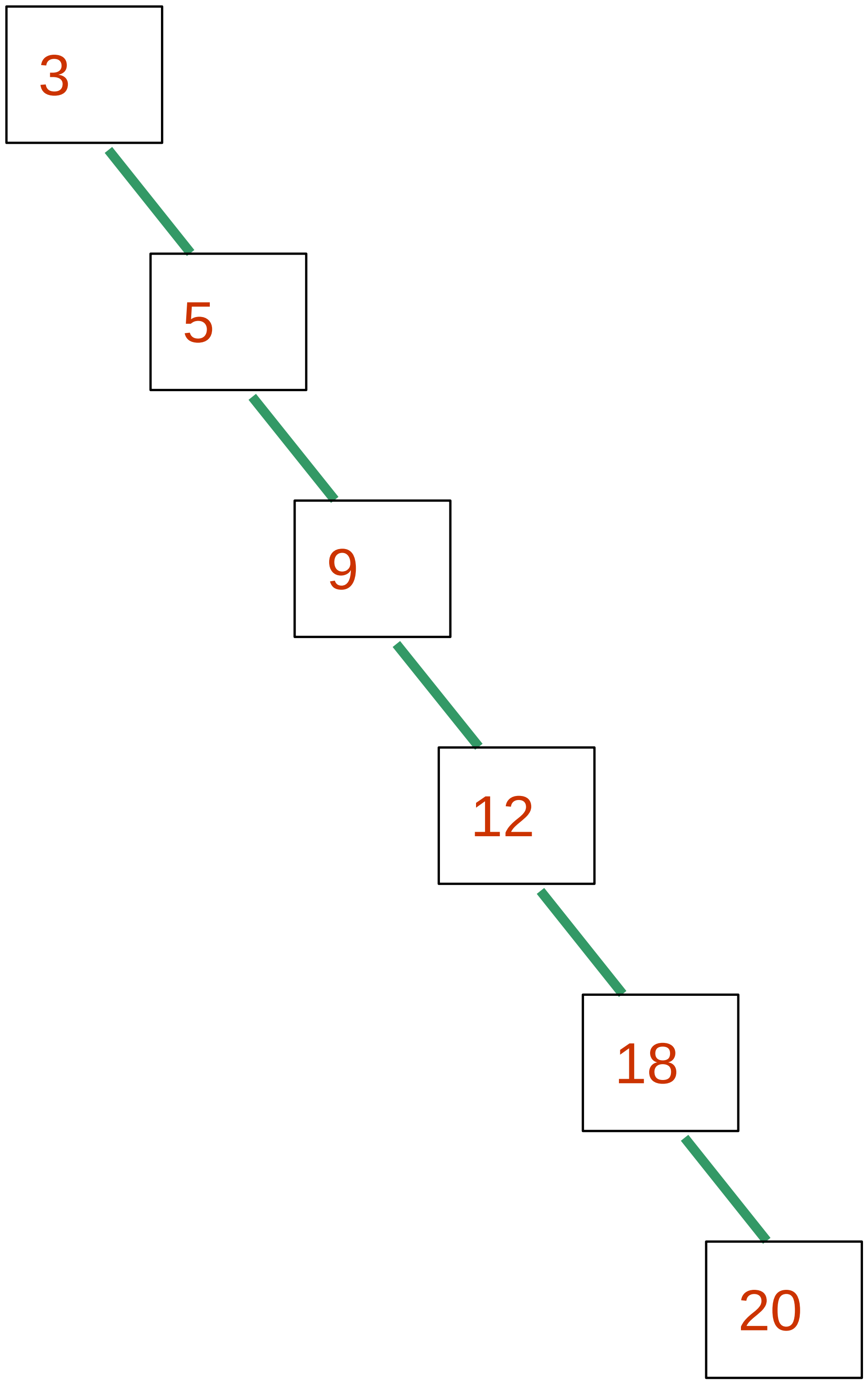
Which of these trees is a balanced tree?

Discussion

- Why is our balance assumption so important?

  - Look at what happens if we insert the following numbers in this order without rebalancing the tree:

3 5 9 12 18 20


3
5
9
12
18
20

Degenerate Binary Trees

- The resulting tree is called a ***degenerate*** binary tree

  - Note that it looks more like a linked list than a tree!
  - But it is actually less efficient than a linked list (***Why?***)