CSC10004: Data Structure and Algorithms

Lecture 8: Trees

Lecturer: Bùi Văn Thạch
TA: Ngô Đình Hy/Lê Thị Thu Hiền
{bvthach,ndhy}@fit.hcmus.edu.vn, lththien@hcmus.edu.vn

Course topics

0. Introduction
1. Algorithm complexity analysis
2. Recurrences
3. Search
4. Sorting
5. Linked list
6. Stack, and Queue, Priority queue

7. Hashing
8. Trees
    1. Binary search trees (BST)
    2. AVL trees
8. Graphs
    1. Graph representation
    2. Graph search
9. Algorithm designs
    1. Greedy algorithm
    2. Divide-and-Conquer
    3. Dynamic programming

fit@hcmus

Goals

1. Understand Hierarchical Data Organization
    1. Learn how data can be organized in a **non-linear** (parent-child) structure.
    2. Recognize the differences between linear structures (like arrays, lists) and trees.
2. Master Tree Terminology and Types
    1. Get comfortable with terms like: root, leaf, node, height, depth, subtree, etc.
    2. Explore common types of trees.
3. Learn Tree Traversals
    1. Learn to visit all nodes in different orders: Level-order (BFS), and In-order, Pre-order, Post-order (DFS strategies).
4. Implement Efficient Operations
    1. Learn how to insert, delete, and search elements efficiently (especially in BSTs and balanced trees).
    2. Understand time complexity ($O(\log n)$ for balanced trees).

Average/Worst-case analysis

	Design	Run time					Space	Collision
		Cost	Search	Insert	Delete	Find_min		
Unsorted array	Det.		$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	
Sorted array by key			$O(\log n)$ $O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	
Singly Linked list			$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	
Hashing	Rnd.		$O(1)$ $O(n)$	$O(1)$ $O(n)$	$O(1)$ $O(n)$	$O(n)$	$O(p)$ $p < n$	YES
(Balanced) Tree	Det.		$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	NO

Outline

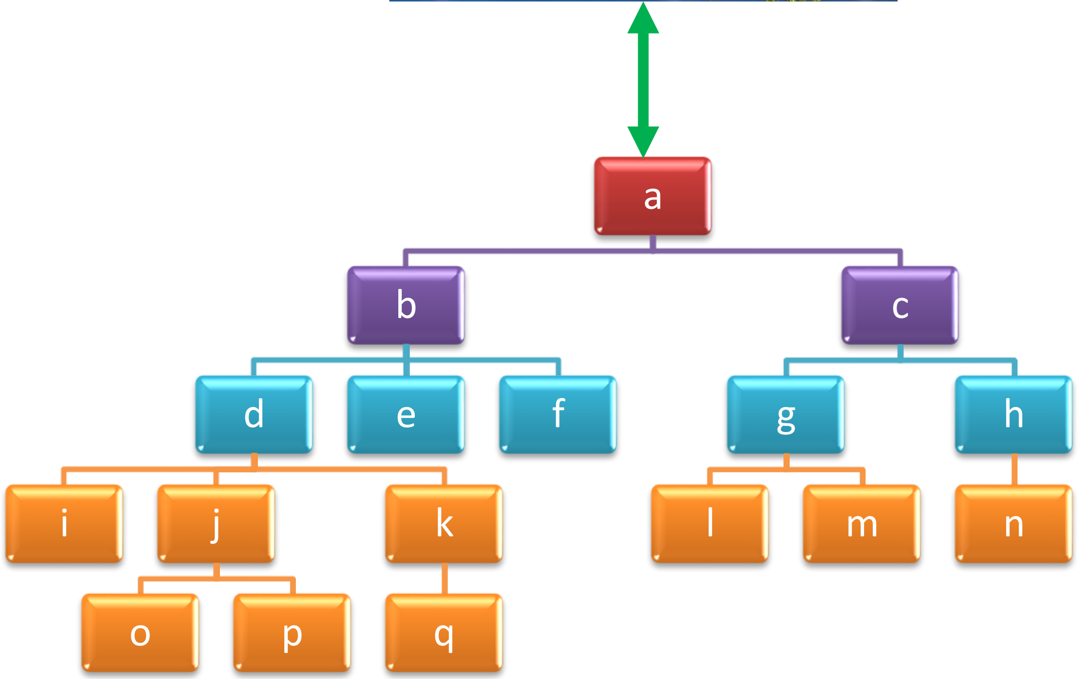
1. Introduction
2. Terminologies
3. Tree traversals
   – Breadth-First Search (BFS)
   – Depth-First Search (DFS)
4. Tree representation
5. Binary trees
6. Binary search trees (BST)
7. Balanced trees
8. AVL trees
9. 2-3, 2-3-4 trees
10. B-trees

Outline

1. Introduction
2. Terminologies
3. Tree traversals
   - Breadth-First Search (BFS)
   - Depth-First Search (DFS)
4. Tree representation
5. Binary trees
6. Binary search trees (BST)
7. Balanced trees
8. AVL trees
9. 2-3, 2-3-4 trees
10. B-trees

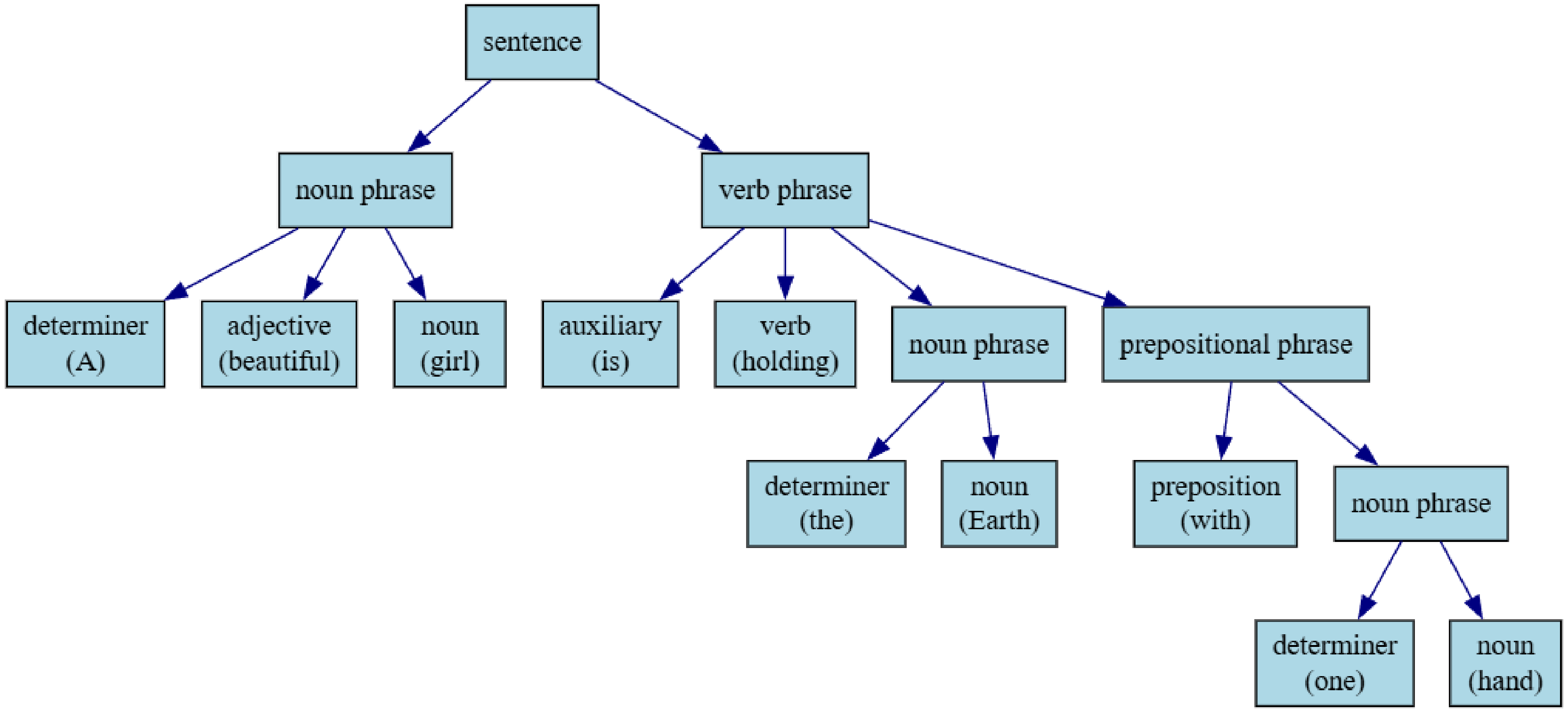
What is tree?

Rotate 180^o

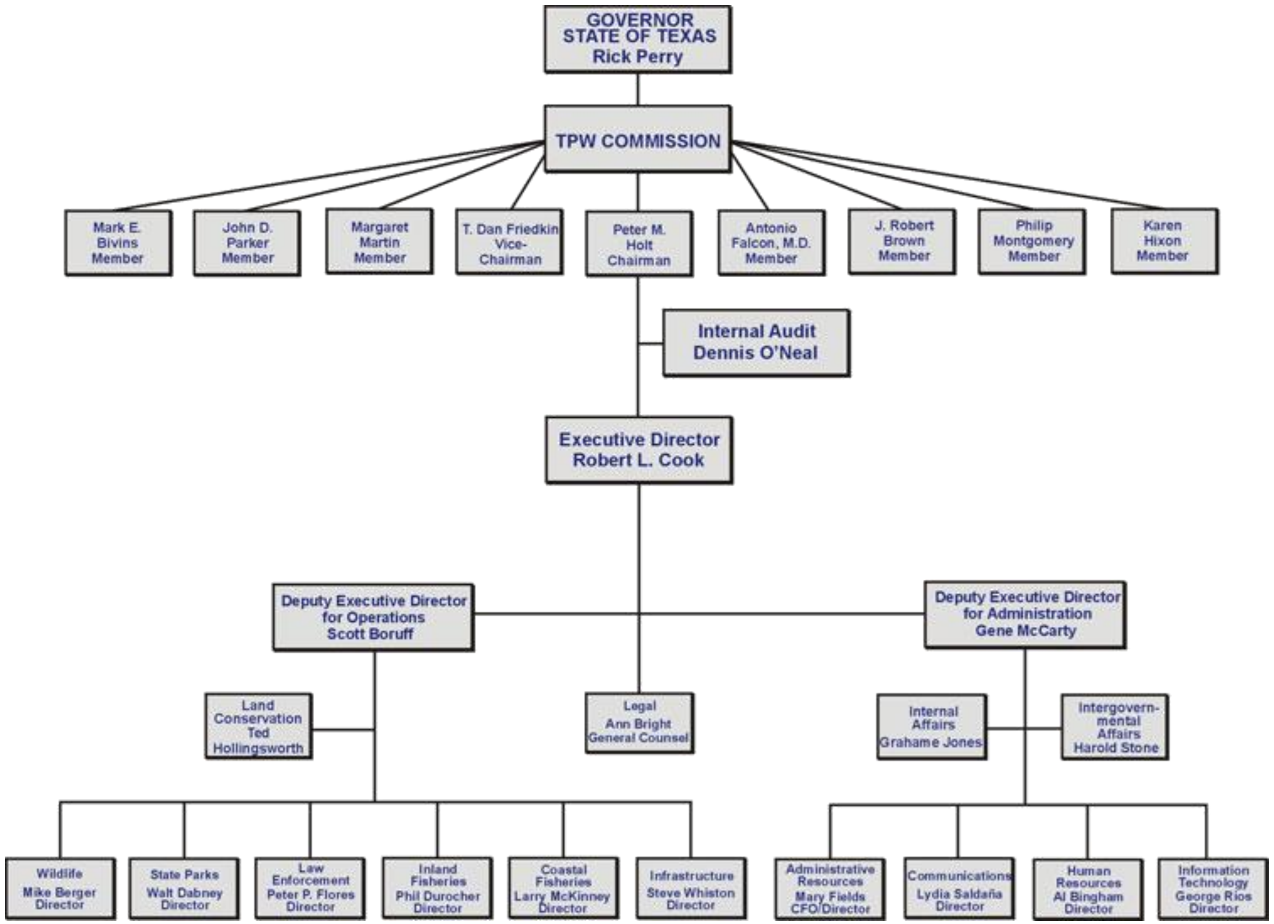


Parse Tree

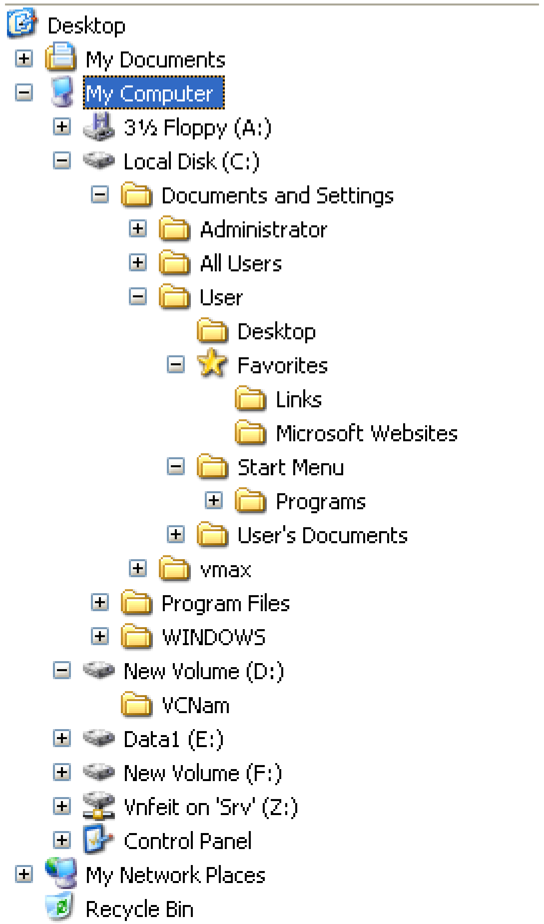
A beautiful girl is holding the Earth with one hand



Organization chart and directory tree

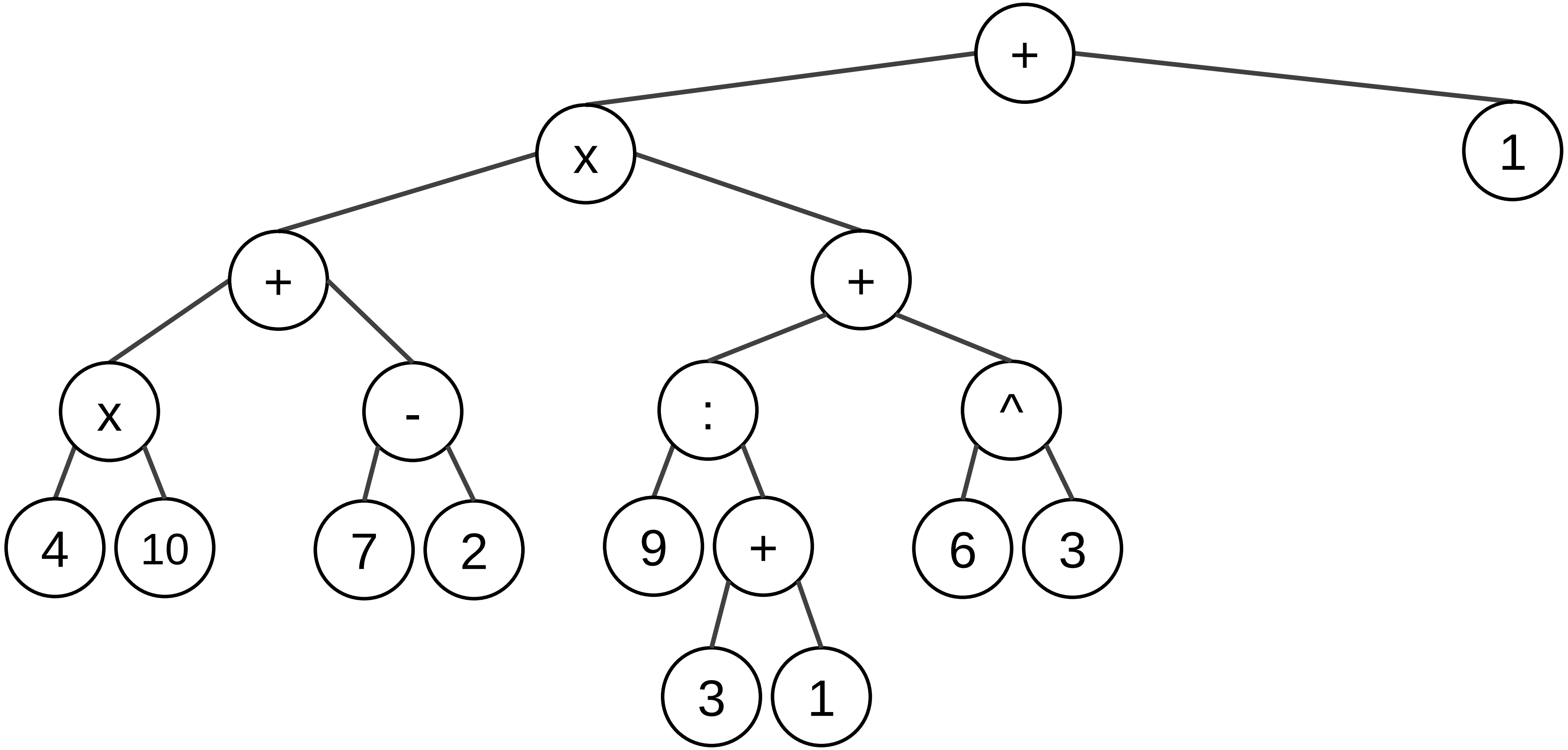


Organizational chart



Directory tree

Mathematical expression



$$\big((4 \times 10) + (7 - 2)\big) \times \left(\frac{9}{3 + 1} + (6^3)\right) + 1$$

Searching Sorted Data

- Consider the following sorted array $a[0, n]$

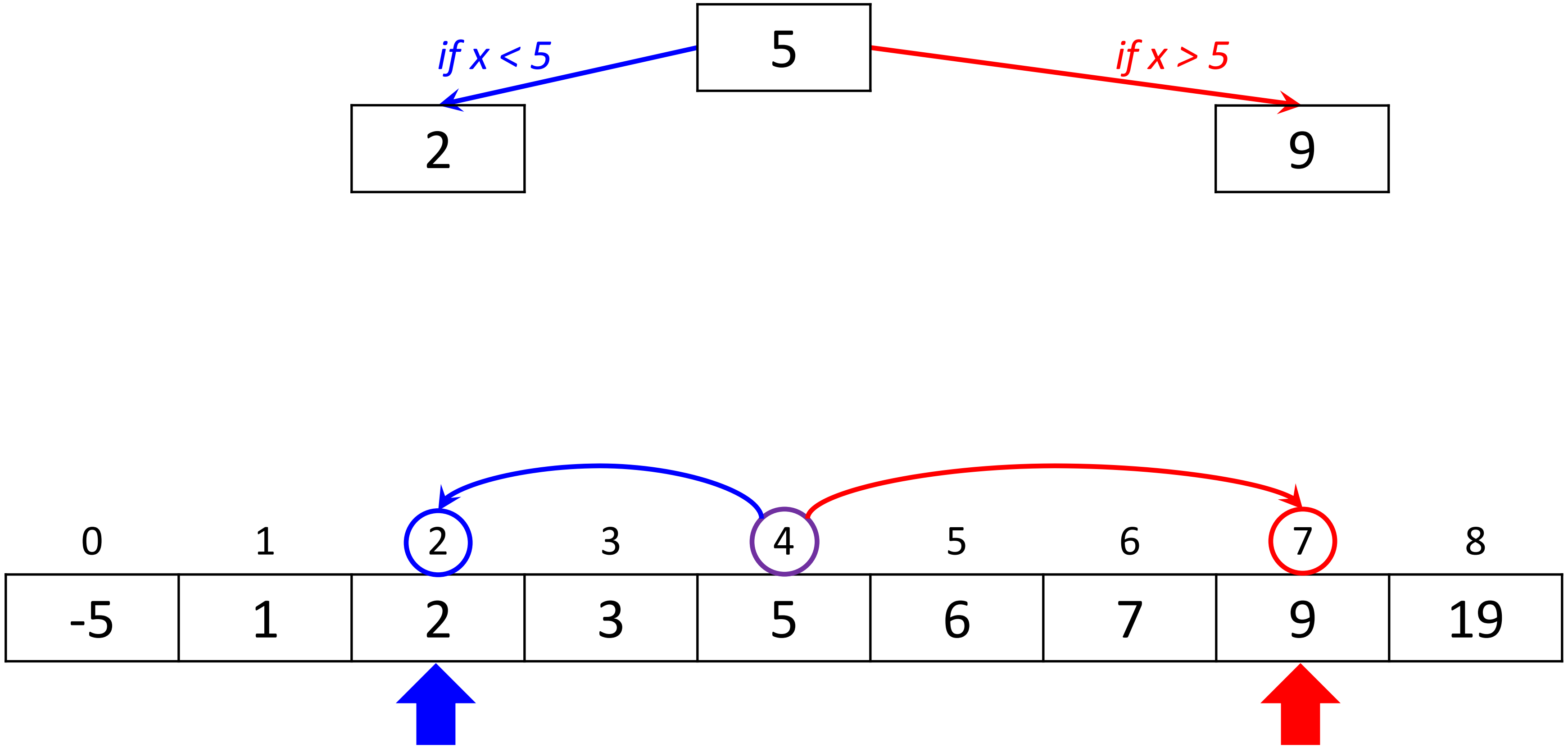
	0	1	2	3	4	5	6	7	$n = 9$
a →	-5	1	2	3	5	6	7	9	19

- When searching for a number x using binary search, we **always** start by looking at the midpoint, index $\left\lceil \frac{n}{2} \right\rceil = 4$.

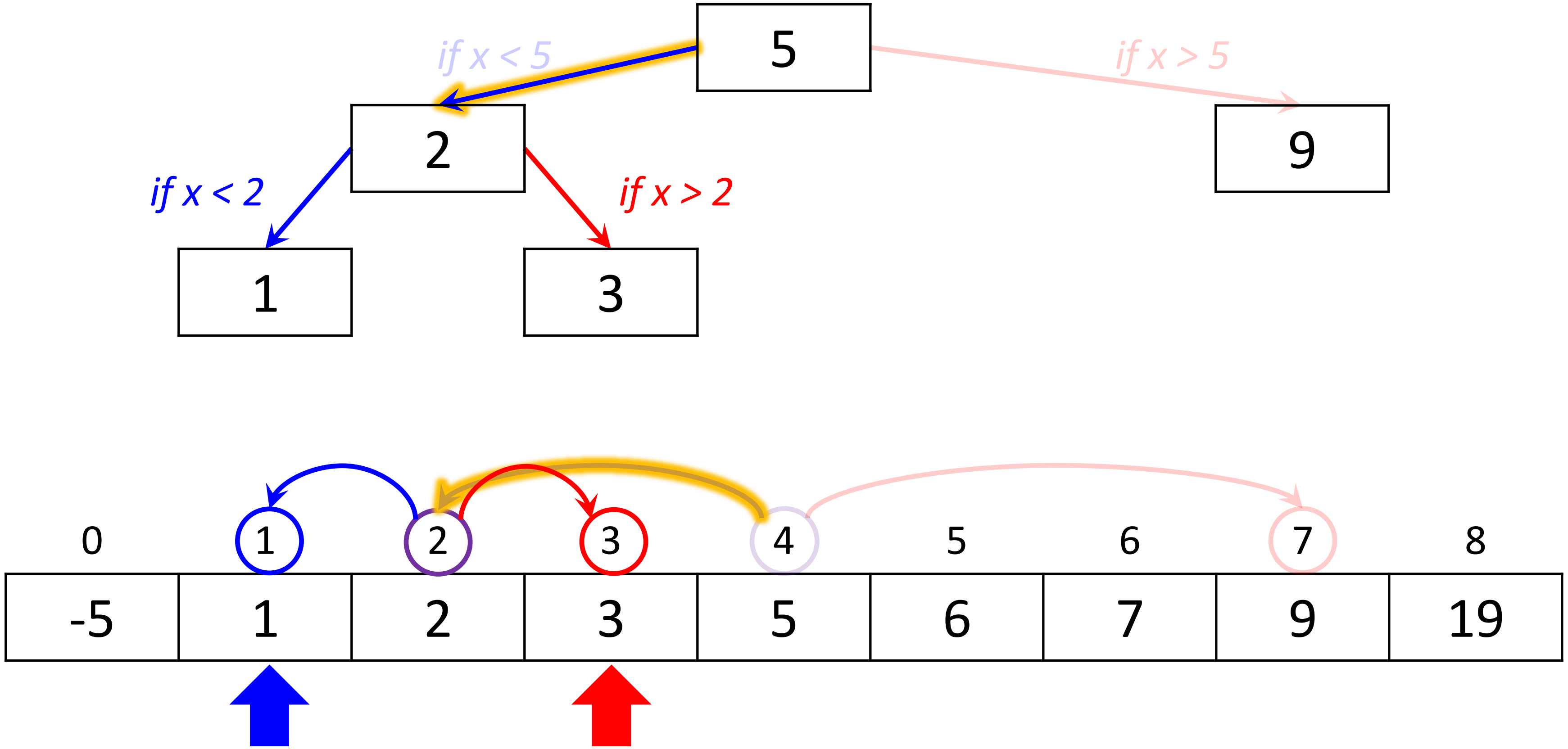
4
5

- Then, 3 things can happen
  - $x = 5$ (and we are done)
  - $x < 5$ then we search in the array $a\left[0, \left\lceil \frac{n}{2} \right\rceil - 1\right]$
  - $x > 5$ then we search in the array $a\left[\left\lceil \frac{n}{2} \right\rceil + 1, n\right]$

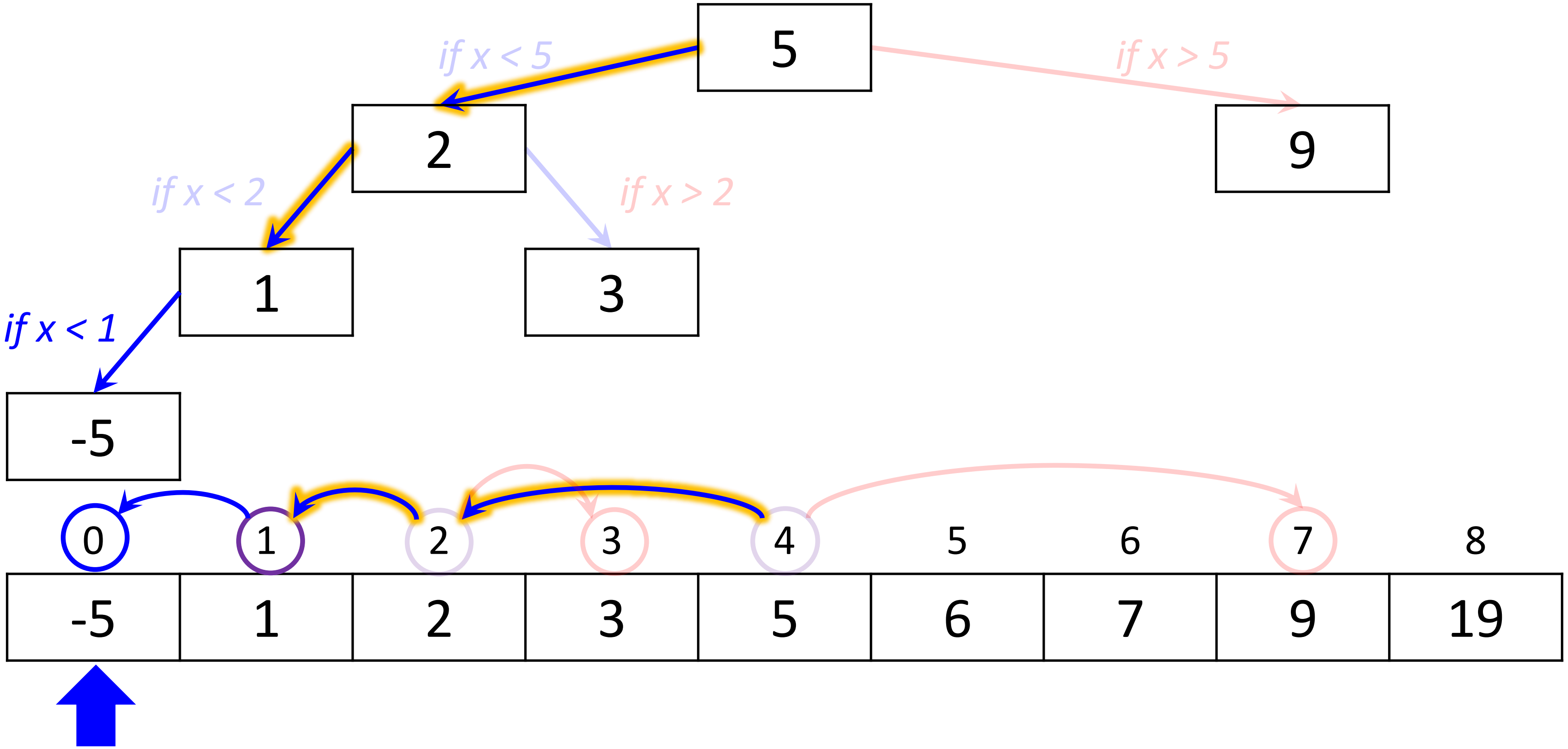
- If x < 5, the next index we look at is **indeed** 2
- If x > 5, the next index we look at is **indeed** 7



- Assume x < 5, so we look at index 2
  - If x = 2, we are done
  - If x < 2, we **indeed** look at index 1
  - If x > 2, we **indeed** look at index 3

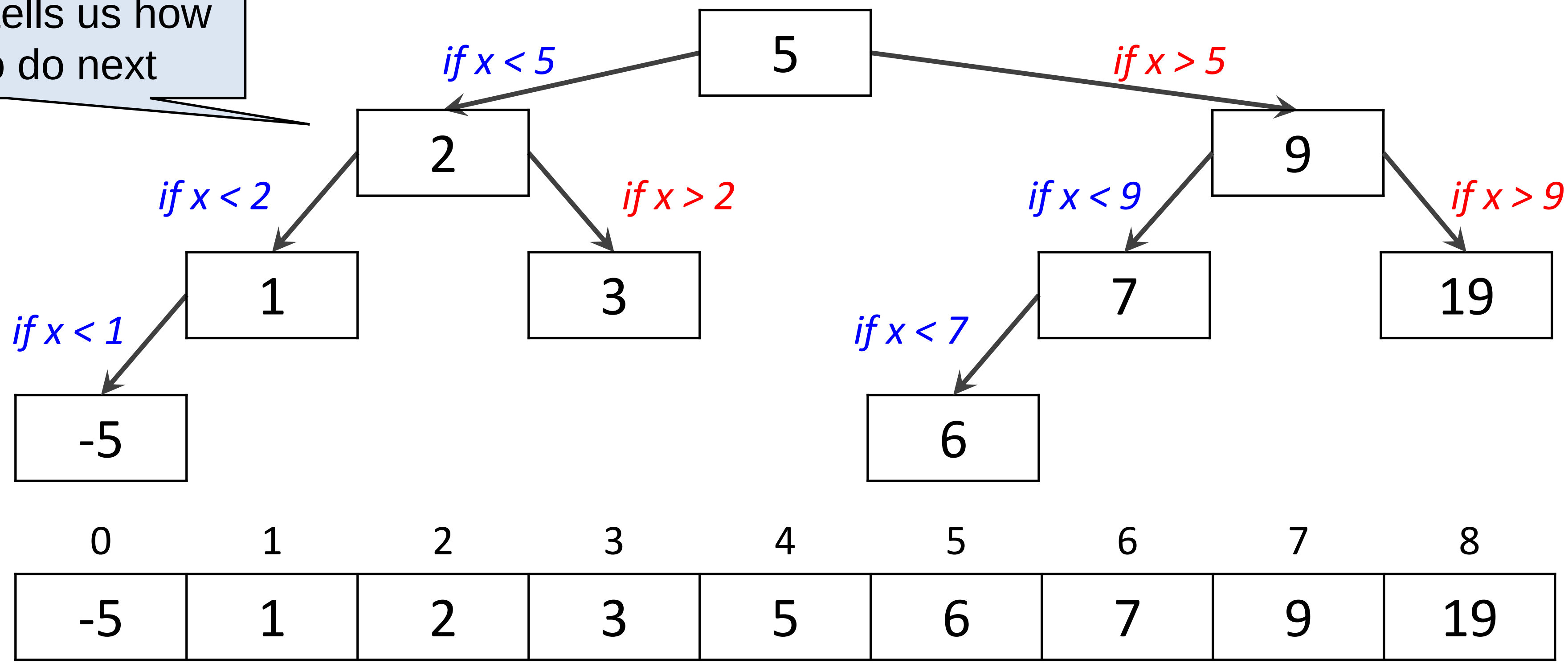


- Assume x < 1, so we look at index 1
  - If x = 1, we are done
  - If x < 1, we **indeed** look at index 0



- We can map out all possible sequences of elements binary search may examine, for any x.

This is called a decision tree: At every step, it tells us how to decide what to do next



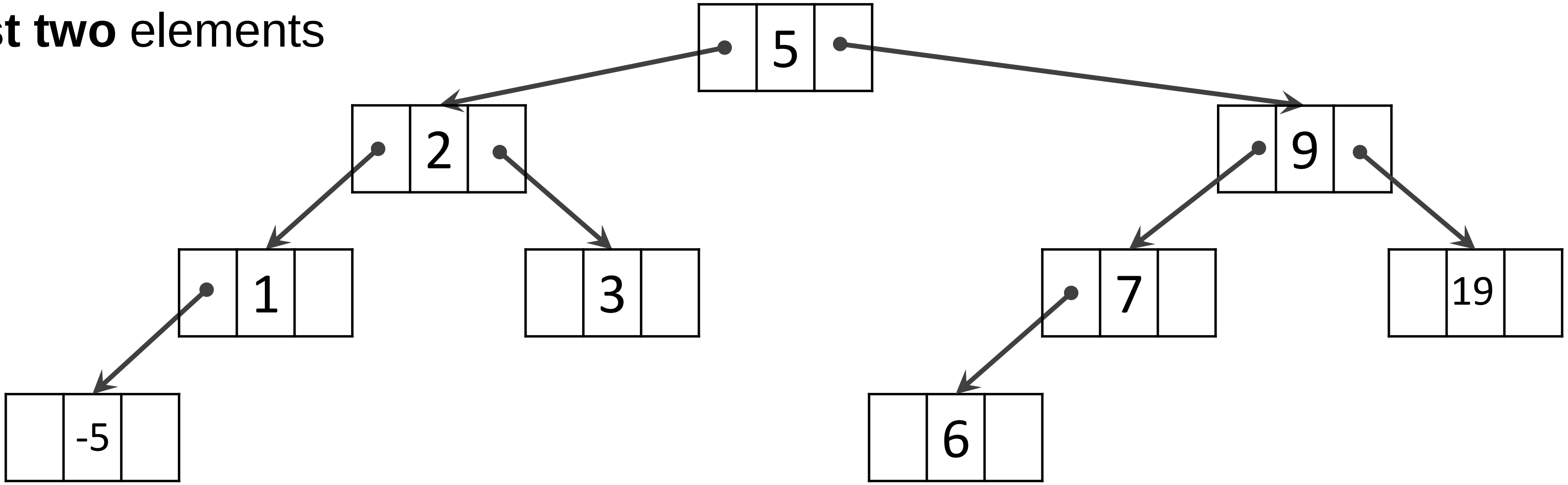
Binary Search Tree representation by a linked list

An array provides direct access to all elements

- This is an overkill for binary search
- At any point, it needs direct access to **at most two** elements

We are losing direct access to arbitrary elements

- But retaining access to the elements that matter to binary search



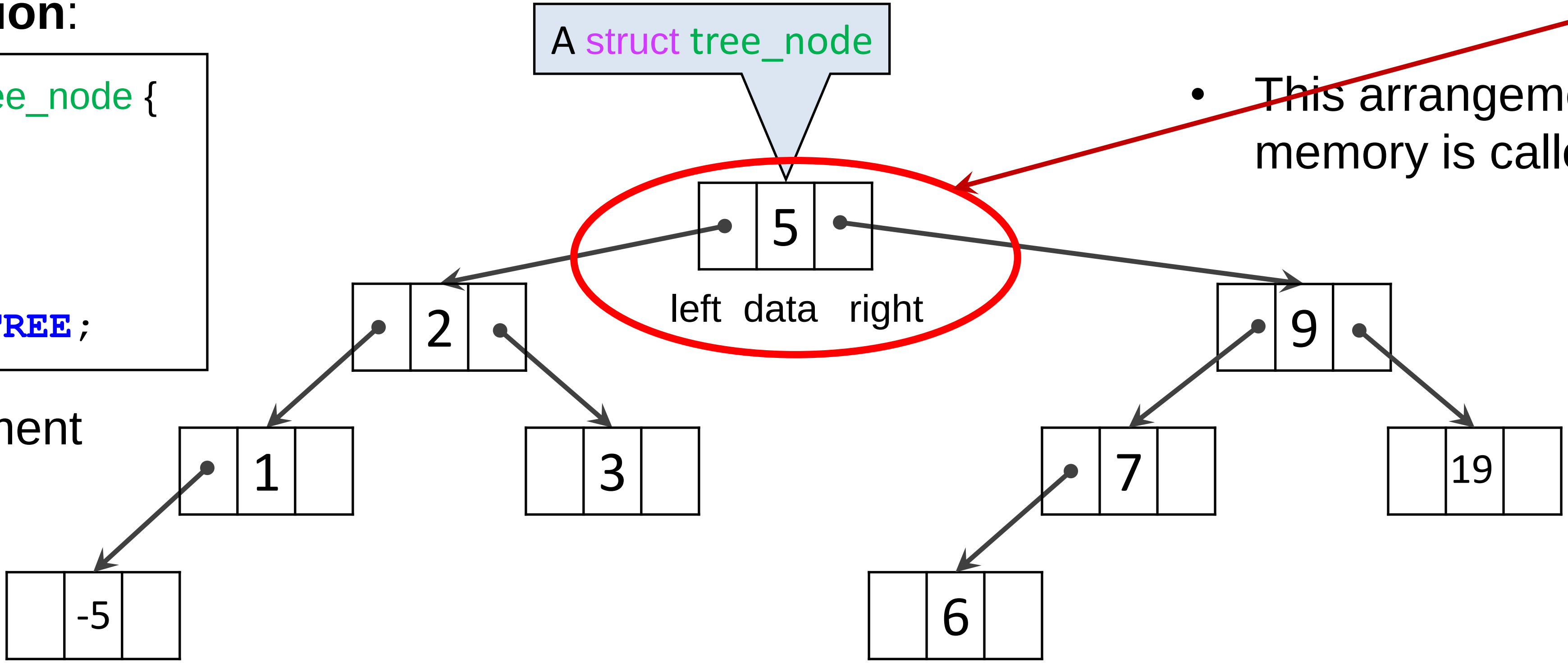
0	1	2	3	4	5	6	7	8
-5	1	2	3	5	6	7	9	19

Implementation: Binary Search Tree (1/2)

We can capture this idea with this **type declaration**:

```
typedef struct tree_node {
  tree* left;
  int data;
  tree* right;
} tree;
typedef  tree  *TREE;
```

- A data element
- Pointers to the 2 elements we may look at next



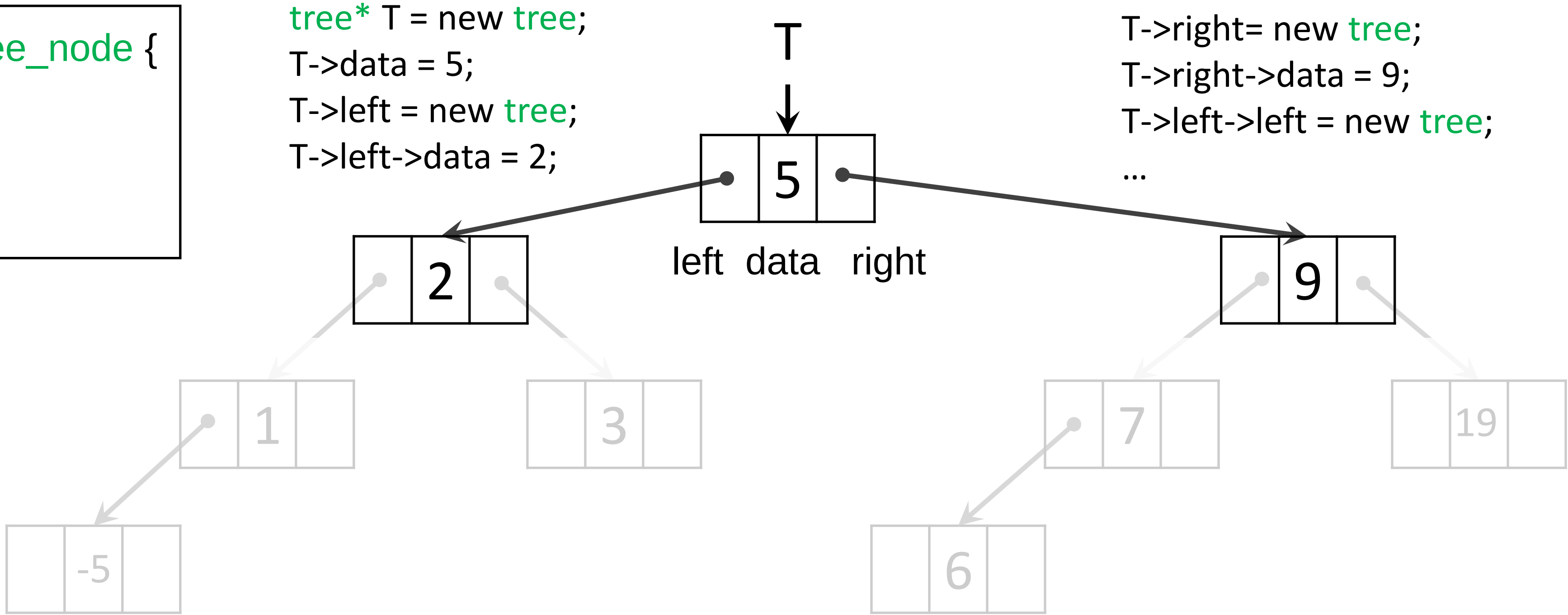
- This struct is called a **node**
- This arrangement of data in memory is called a **tree**

Implementation: Binary Search Tree (2/2)

Let's build the first three nodes of this example

```
typedef struct tree_node {
  tree* left;
  int data;
  tree* right;
} tree;
```

```
tree* T = new tree;
T->data = 5;
T->left = new tree;
T->left->data = 2;
```




```
T->right= new tree;
T->right->data = 9;
T->left->left = new tree;
...
```

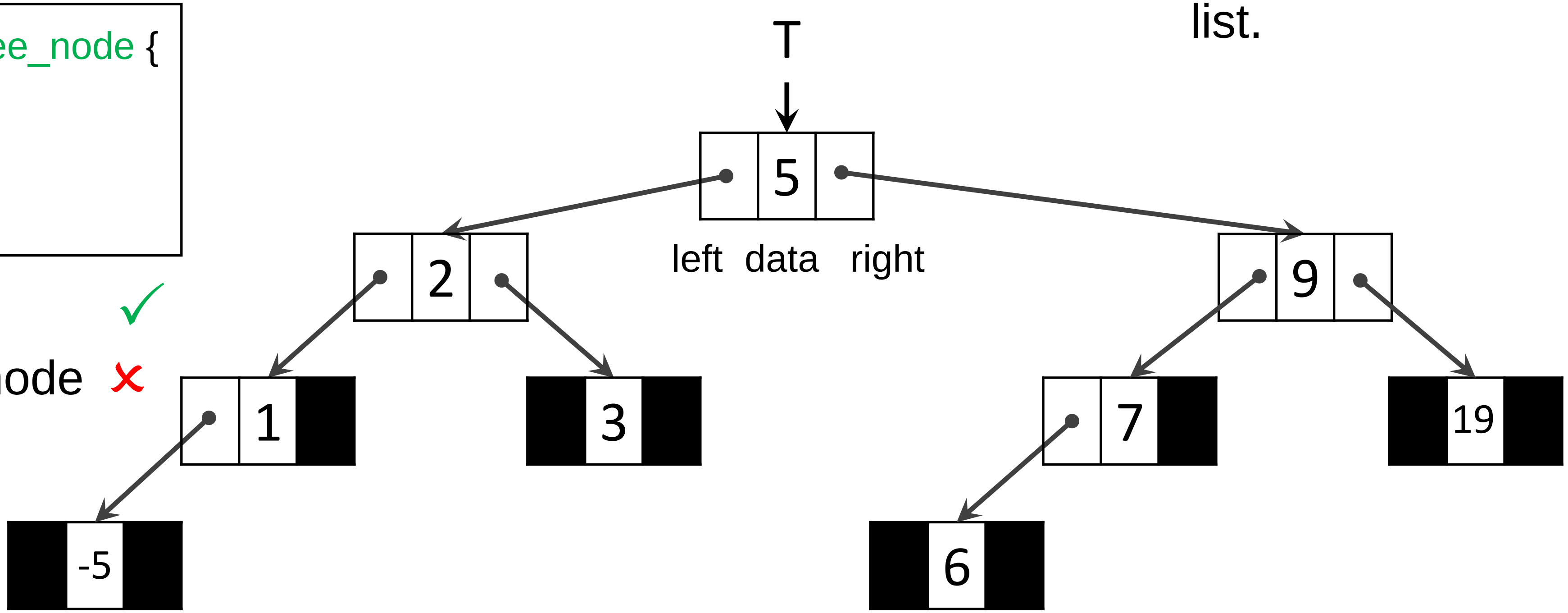
The End of nodes

What should the grey left/right fields point to?

```
typedef struct tree_node {
  tree* left;
  int data;
  tree* right;
} tree;
```

- NULL ✓
- A dummy node ✗

We used dummy nodes to get direct access to the end of a list.



Search (1/2)

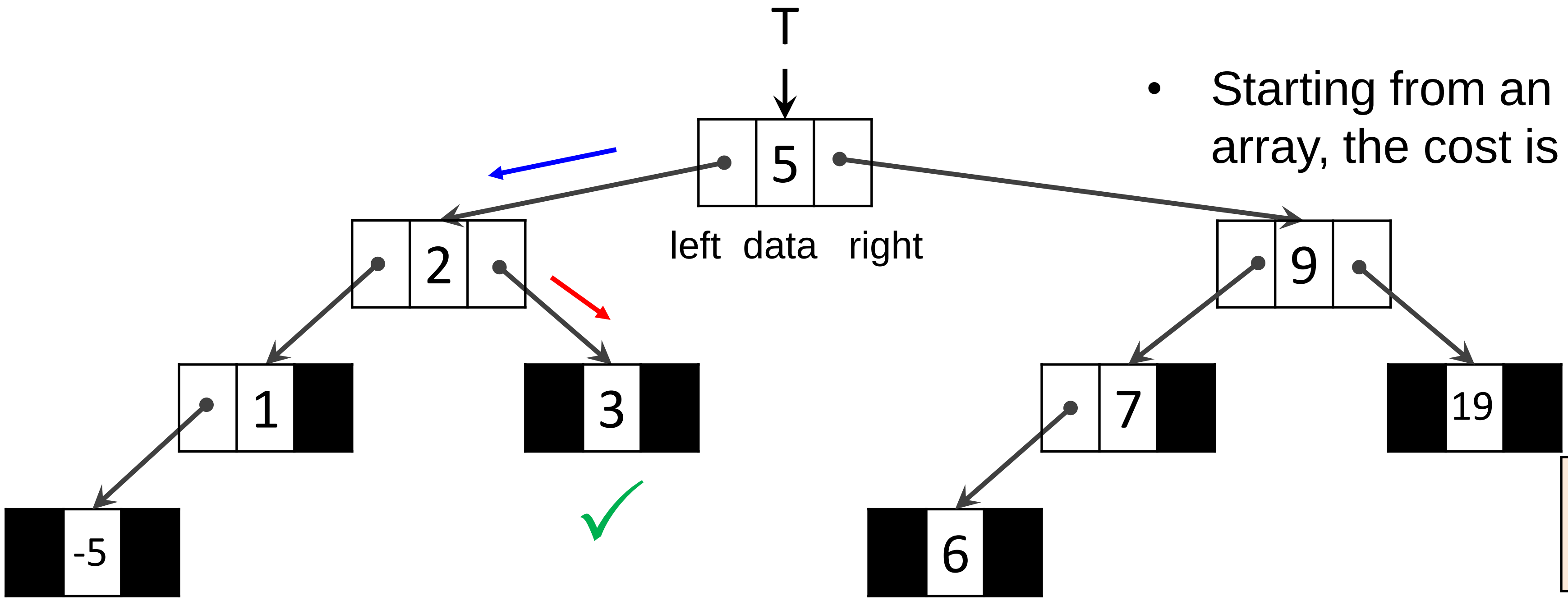
Search for 3:

3 < 5: Go left

3 > 2: Go right

3 = 3: **Found**

- We are doing the **same steps** as binary search
- Starting from an n-element array, the cost is $O(\log n)$



Only applied to this tree!

Search (2/2)

Search for 4:

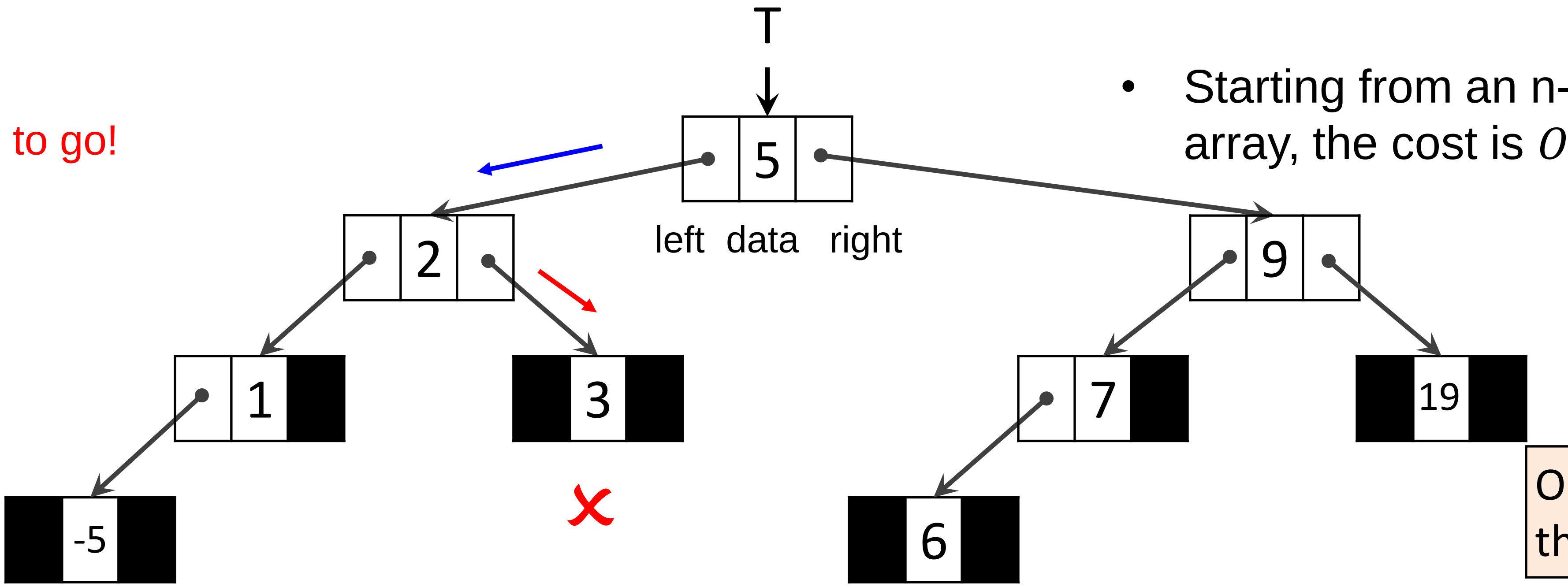
4 < 5: Go left

4 > 2: Go right

4 > 3: Go right

No where to go!

➔ **Not found**



- We are doing the **same steps** as binary search
- Starting from an n-element array, the cost is $O(\log n)$

Only applied to this sorted tree!

Worst-case analysis

	Design	Run time					Space	Collision
		Cost	Search	Insert	Delete	Find_min		
Unsorted array	Det.		$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	
Sorted array by key			$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	
Singly Linked list			$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	
Hashing	Rnd.		$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(p)$ $p < n$	YES
Binary Search Tree	Det.		$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	NO

✓ ? ? ?

Insertion (1/2)

Insert 8:

8 > 5: Go right

8 < 9: Go left

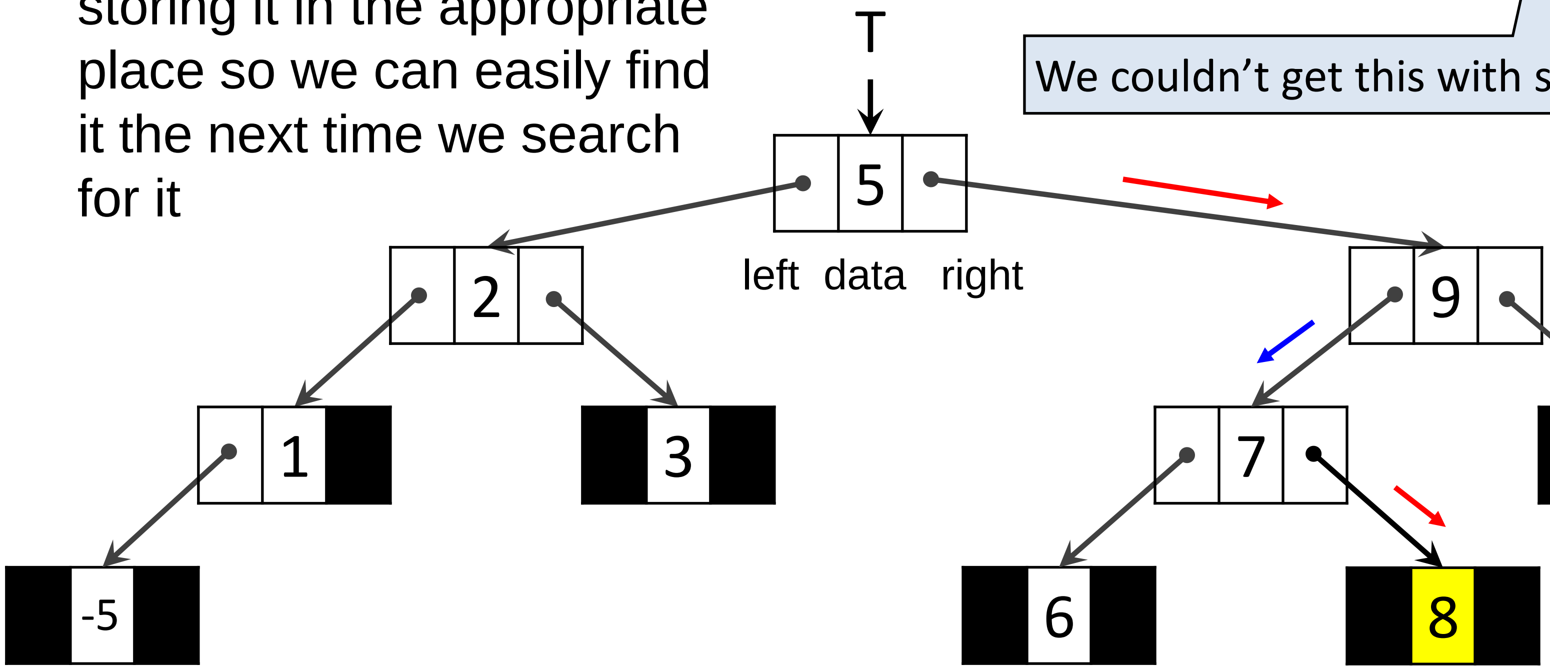
8 > 7: Go right

Put it there

We are following **the same steps** as in a search, then storing it in the appropriate place so we can easily find it the next time we search for it

- Starting from an n-element array, the cost is $O(\log n)$

We couldn't get this with sorted arrays



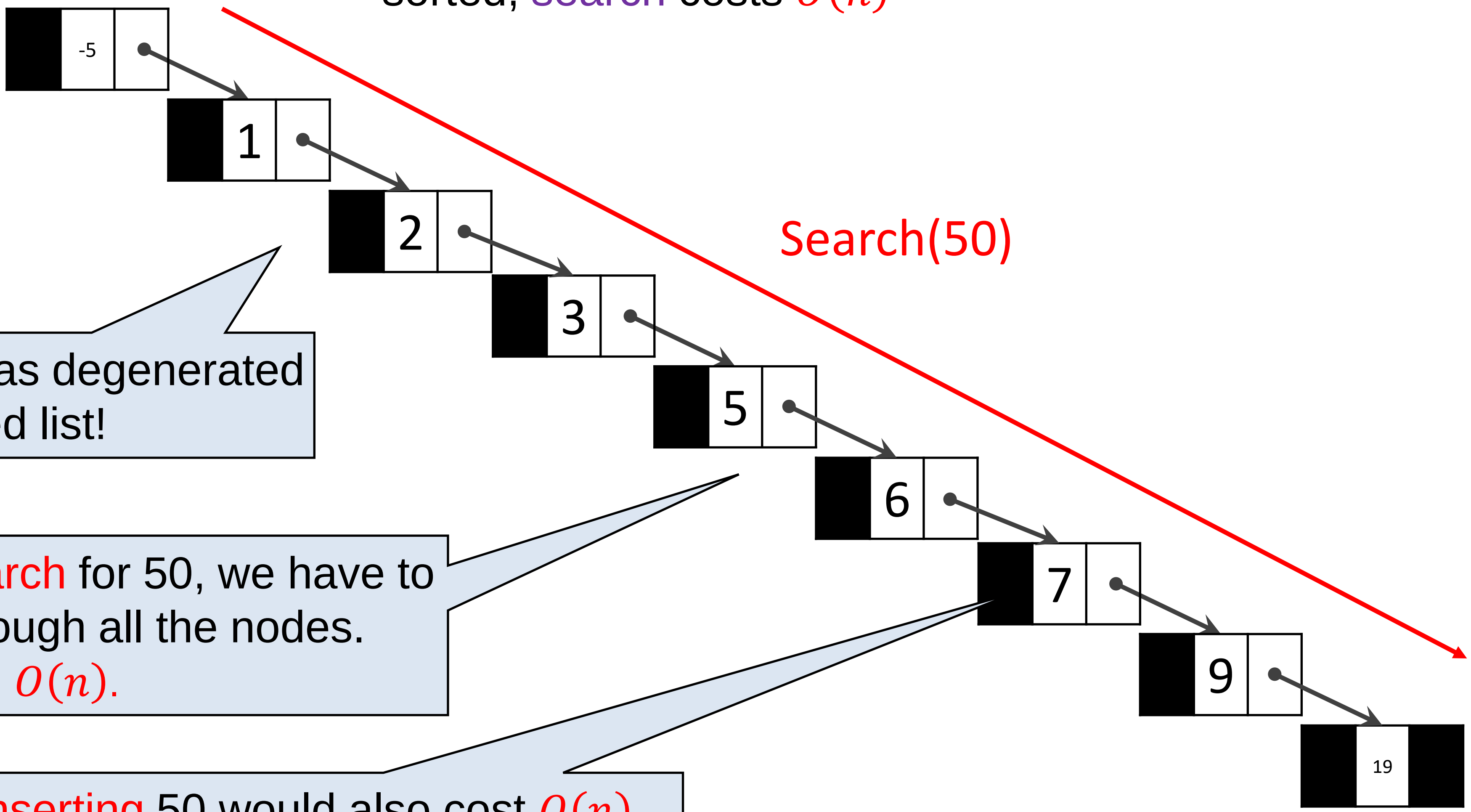
Only applied to this tree!

Insertion (2/2): degeneration

Consider this sequence of insertions into an initially empty BST

Insert -5
Insert 1
Insert 2
Insert 3
Insert 5
Insert 6
Insert 7
Insert 9
Insert 19

If the insertion sequence is sorted, search costs $O(n)$



The tree has degenerated into a linked list!

To search for 50, we have to go through all the nodes. This is $O(n)$.

Inserting 50 would also cost $O(n)$.

Worst-case analysis

	Design	Run time					Space	Collision
		Cost	Search	Insert	Delete	Find_min		
Unsorted array	Det.		$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	
Sorted array by key			$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	
Singly Linked list			$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	
Hashing	Rnd.		$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(p)$ $p < n$	YES
Binary Search Tree	Det.		$O(\log n)$	$O(\log n)$			$O(n)$	NO

CAUTION: A BST **after insertion** may not be able to search in $O(\log n)$!!!

x x ? ?

Deletion (1/2)

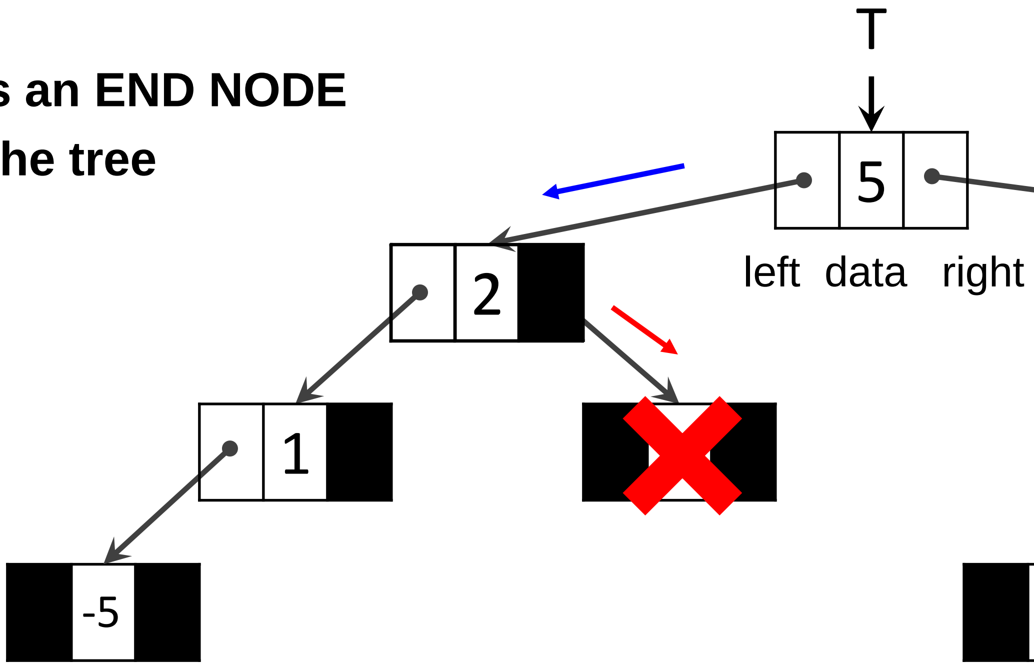
Delete for 3:

3 < 5: Go left

3 > 2: Go right

3 = 3: **Found as an END NODE**

Remove from the tree



- We are doing the **same steps** as binary search
- Starting from an n-element array, the cost is $O(\log n)$

Only applied to this tree!

Deletion (2/2)

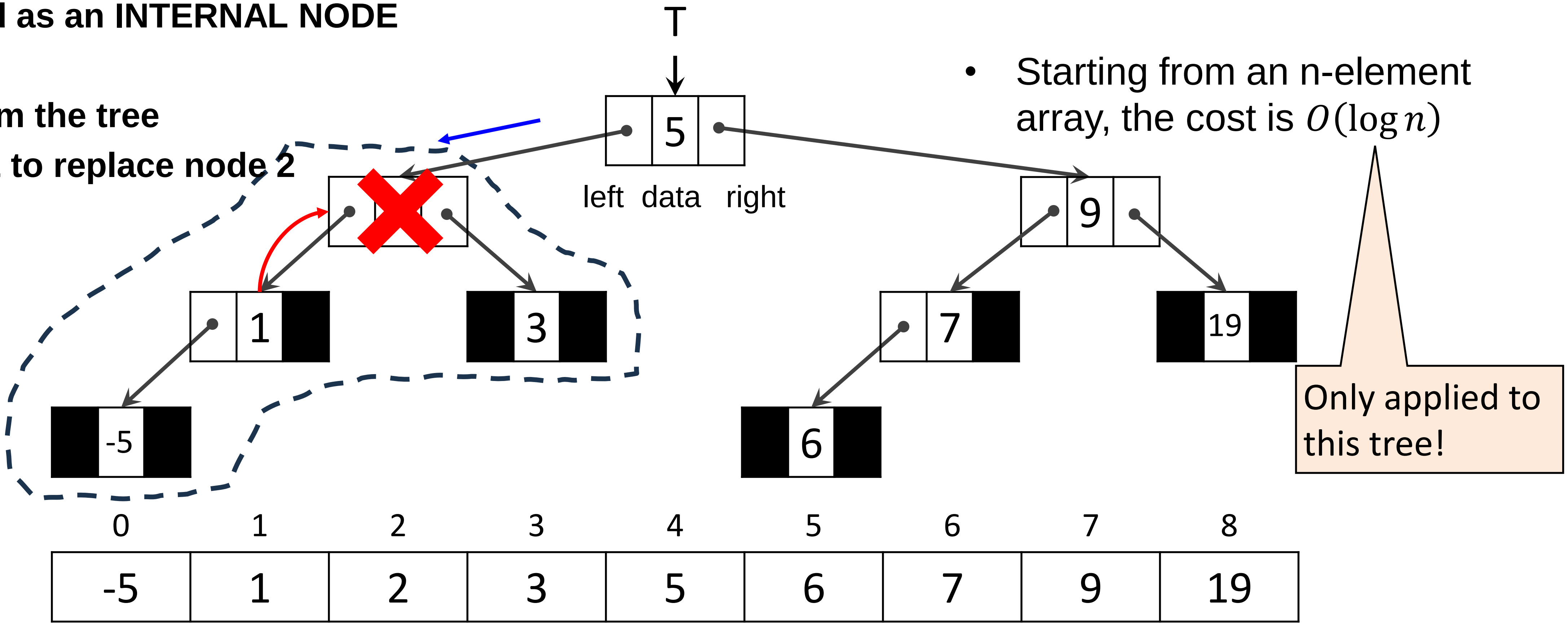
Delete for 2:

2 < 5: Go left

2 = 2: **Found as an INTERNAL NODE**

Remove from the tree

Take node 1 to replace node 2



- We are doing the **same steps** as binary search
- Starting from an n-element array, the cost is $O(\log n)$

Deletion (2/2)

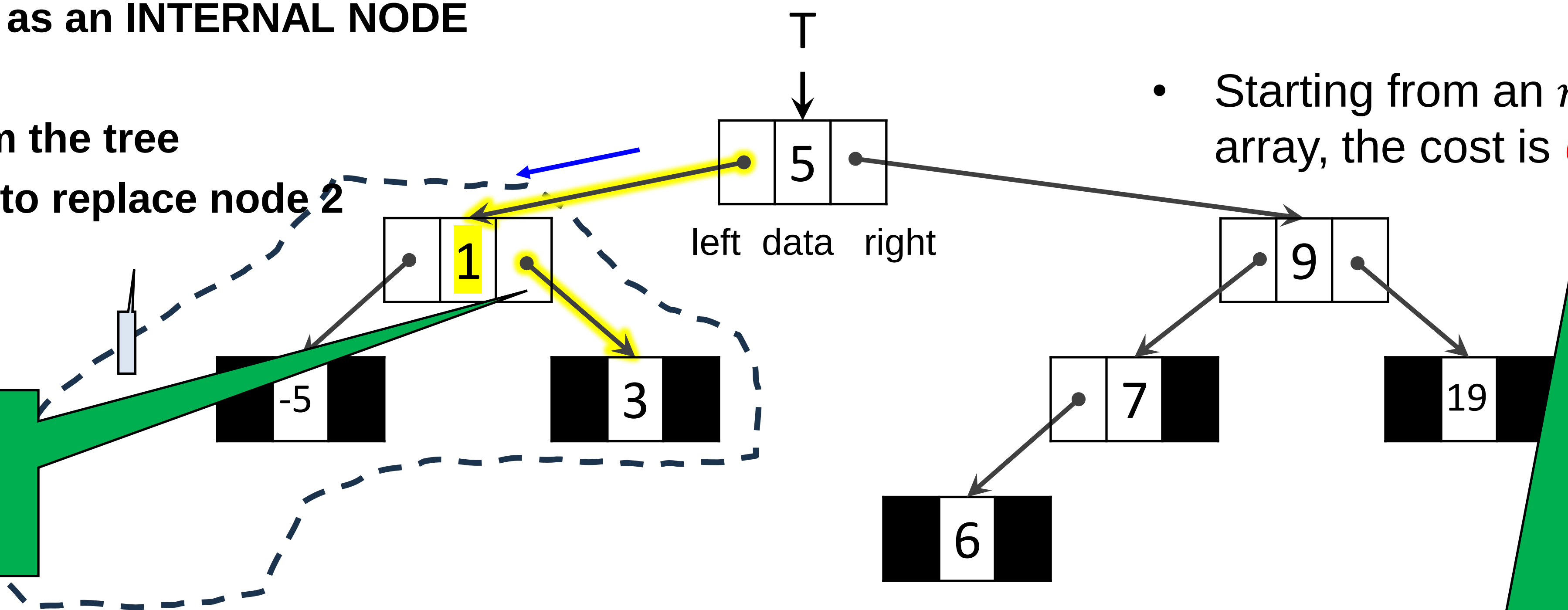
Delete for 2:

2 < 5: Go left

2 = 2: **Found as an INTERNAL NODE**

Remove from the tree

Take node 1 to replace node 2



What happens if node 1 has a right child?

- We are doing the **same steps** as binary search
- Starting from an n-element array, the cost is $O(1)$

changed to $O(n)$ (Why?)

CAUTION: $O(n)$ when the tree has degenerated into a linked list!!!

Worst-case analysis

	Design	Run time					Space	Collision
		Cost	Search	Insert	Delete	Find_min		
Unsorted array	Det.		$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	
Sorted array by key			$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	
Singly Linked list			$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	
Hashing	Rnd.		$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(p)$ $p < n$	YES
Binary Search Tree	Det.		$O(\log n)$	$O(\log n)$	$O(\log n)$		$O(n)$	NO

x x ✓ ?

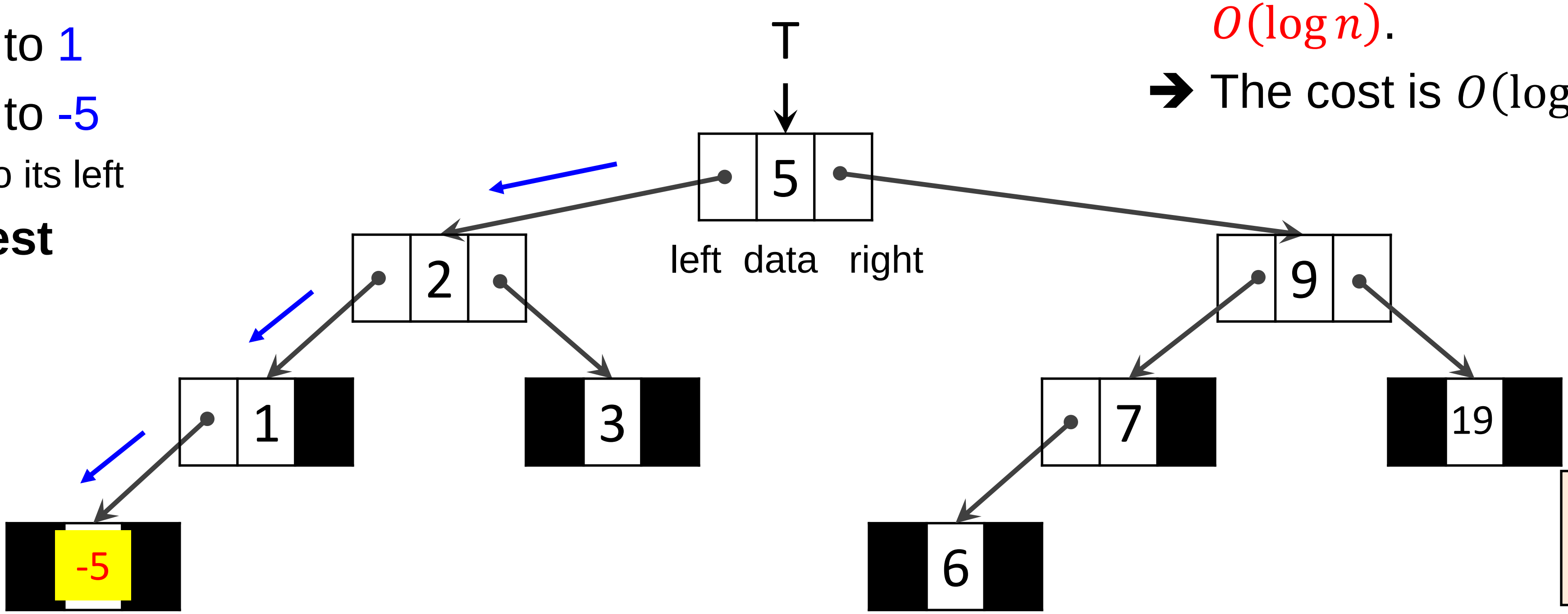
Finding the Smallest Key

Keep going left:

- Left from 5 to 2
- Left from 2 to 1
- Left from 0 to -5
  - Nothing to its left
- **The smallest key is -5**

- Starting from an n-element array, the heigh is at most $O(\log n)$.

➔ The cost is $O(\log n)$.



Only applied to this tree!

CAUTION: $O(n)$ when the tree has degenerated into a linked list!!!

Worst-case analysis

	Design	Run time					Space	Collision
		Cost	Search	Insert	Delete	Find_min		
Unsorted array	Det.		$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	
Sorted array by key			$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	
Singly Linked list			$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	
Hashing	Rnd.		$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(p)$ $p < n$	YES
Binary Search Tree	Det.		$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	NO

CAUTION: A BST **after insertion** may not be able to search, delete or find the min in $O(\log n)$!!!

X X X X

How to keep the best properties of a BST for those operations?

Worst-case analysis

	Design	Run time					Space	Collision
		Cost	Search	Insert	Delete	Find_min		
Unsorted array	Det.		$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	
Sorted array by key			$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	
Singly Linked list			$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	
Hashing	Rnd.		$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(p)$ $p < n$	YES
Binary Search Tree	Det.		$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	NO
?	Det.		$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	NO

Find a data structure that can ALWAYS achieve these complexities!

Outline

1. Introduction
2. Terminologies
3. Tree traversals
   - Breadth-First Search (BFS)
   - Depth-First Search (DFS)
4. Tree representation
5. Binary trees
6. Binary search trees (BST)
7. Balanced trees
8. AVL trees
9. 2-3, 2-3-4 trees
10. B-trees

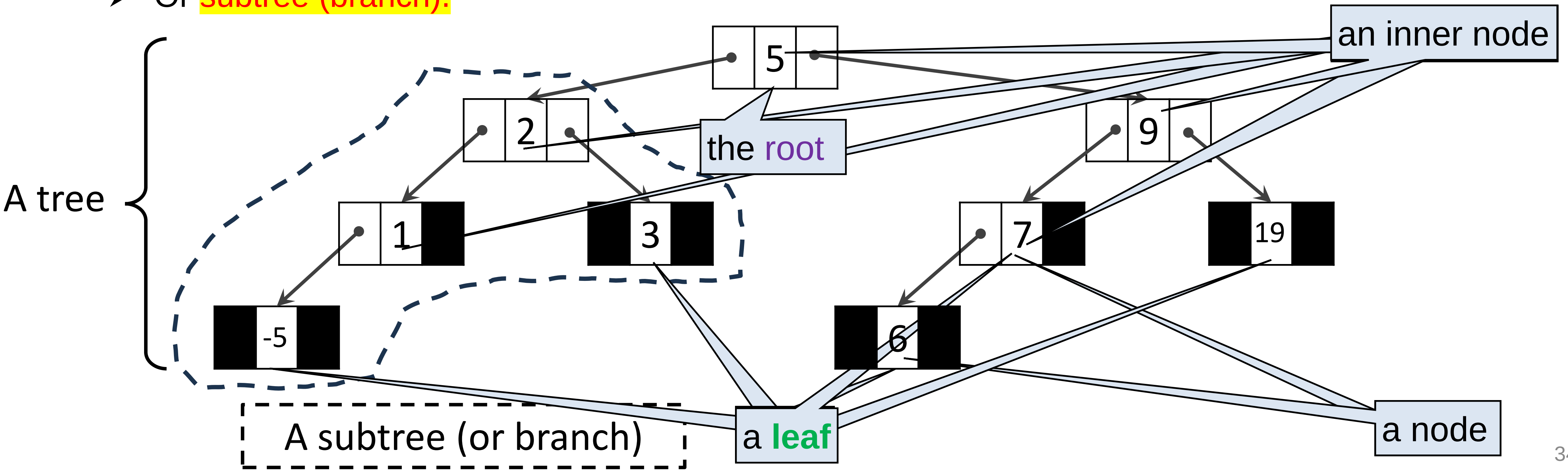
Tree, node, subtree (branch)

A tree structure with base type D is either:

- An empty structure (called the empty tree, or NULL), or
- A node containing:
  - Information of type D.
  - **Links** to a finite number of other tree structures (also of type D).
    - Or subtree (branch).

The node at the top is called the root of the tree

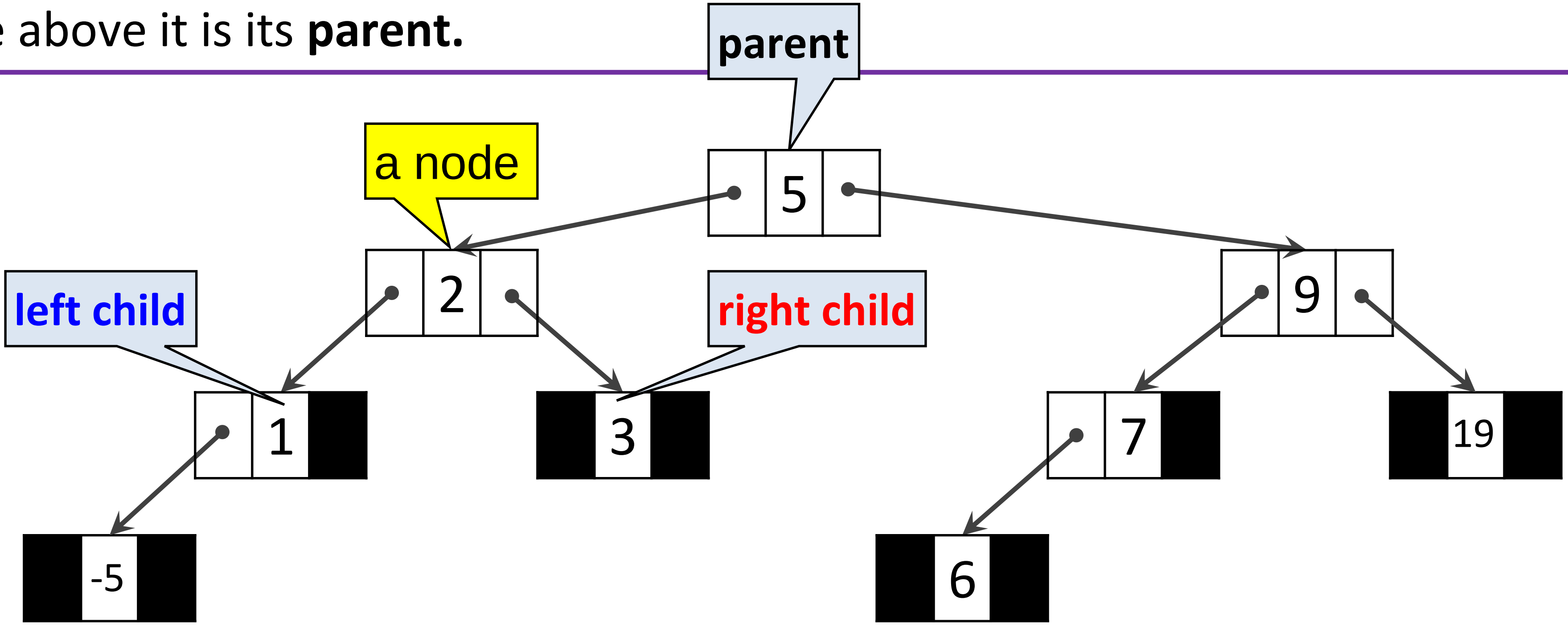
- The nodes at the bottom are the **leaves** of the tree
- The other nodes are called **inner nodes**



Left child, right child, parent

Definition 1. Given any node, we define:

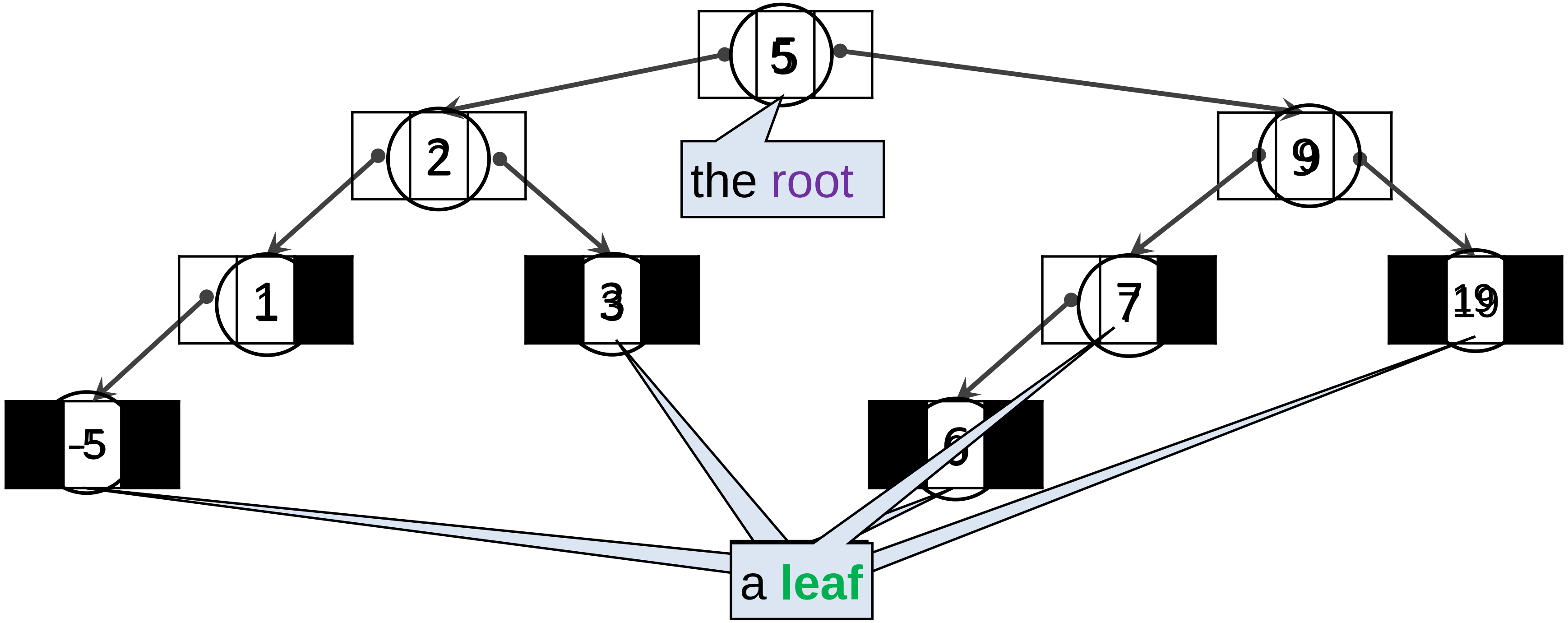
- The node to its left is its **left child.**
- The node to its right is its **right child.**
- The node above it is its **parent.**



Concrete (Simplified) Tree Diagrams

When drawing trees, we usually leave out the details of memory diagrams.

- Show only the data stored in each node, not the pointer fields.
- Indicate the connections to their children.



Walk, Trail, Path

Definition 1. Let G be a graph, and let v and w be vertices of G.

A walk from v to w is a finite alternating sequence of adjacent vertices and edges of G. Thus a walk has the form

$$v_0 e_1 v_1 e_2 \ldots v_{n-1} e_n v_n,$$

where the v's represent vertices, the e's represent edges, $v_0 = v, v_n = w$, and for all $i \in \{1, 2, \ldots, n\}$, v_{i-1} and v_i are the endpoints of e_i.

The trivial walk from v to v consist of the single vertex v.

A trail from v to w is a walk from v to w that does not contain a repeated edge.

A path from v to w is a trail that does not contain a repeated vertex.

Walk, Trail, Path

	Repeated edge?	Repeated vertex?	Starts and Ends at the same point?	Must contain at least one edge?
Walk	ALLOWED	ALLOWED	ALLOWED	NO
Trail	NO	ALLOWED	ALLOWED	NO
Path	NO	NO	NO	NO

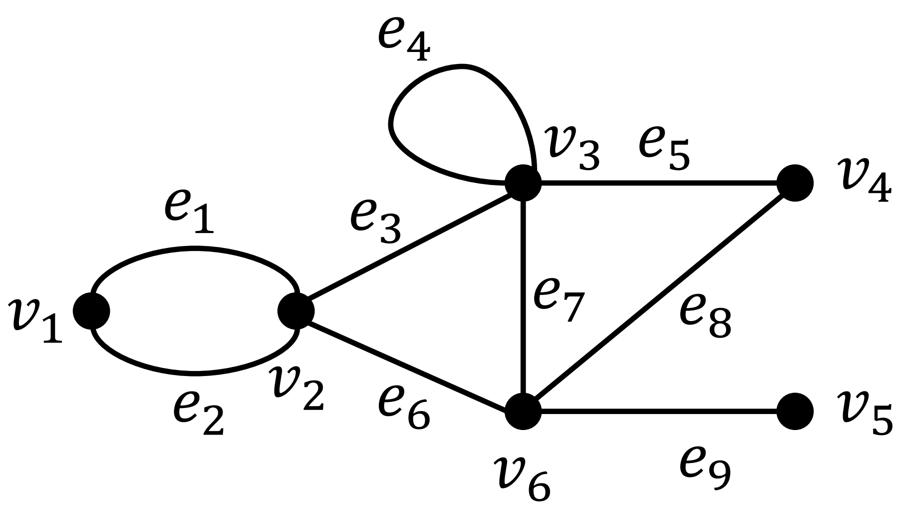
Quiz

In this graph, determine which of the following walks are trails or paths:

a) $v_1 e_1 v_2 e_3 v_3 e_4 v_3 e_5 v_4$.

b) $e_1 e_3 e_5 e_5 e_7$.

c) $v_1 e_1 v_2 e_6 v_6 e_9 v_5$.



Rooted tree, level, height

Definition 2 ([1, p. 122]). A rooted *tree* is a tree in which there is one node, called the root, at the top is distinguished from the others.

The level/depth of a **node** is the number of edges along the unique PATH between it and the root.

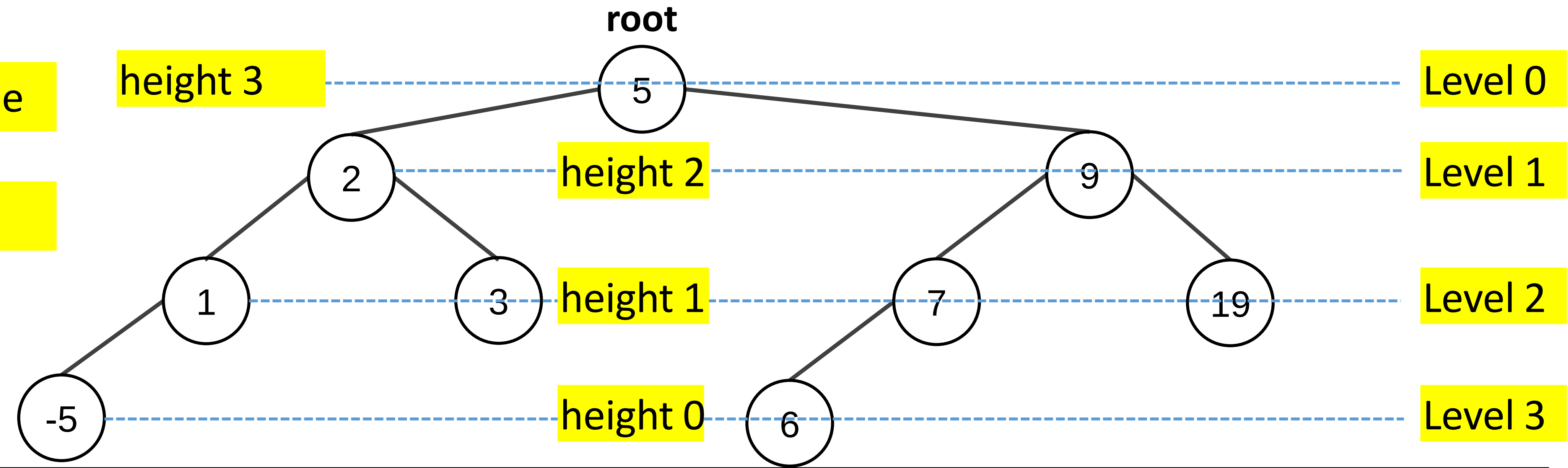
The height of a **node** is the length of the longest path from it to a leaf.

The height of a **rooted tree** is the MAXIMUM level of any node of the tree, i.e. the LONGEST PATH.

Level of a node

height(T) = 3

Empty tree has height -1



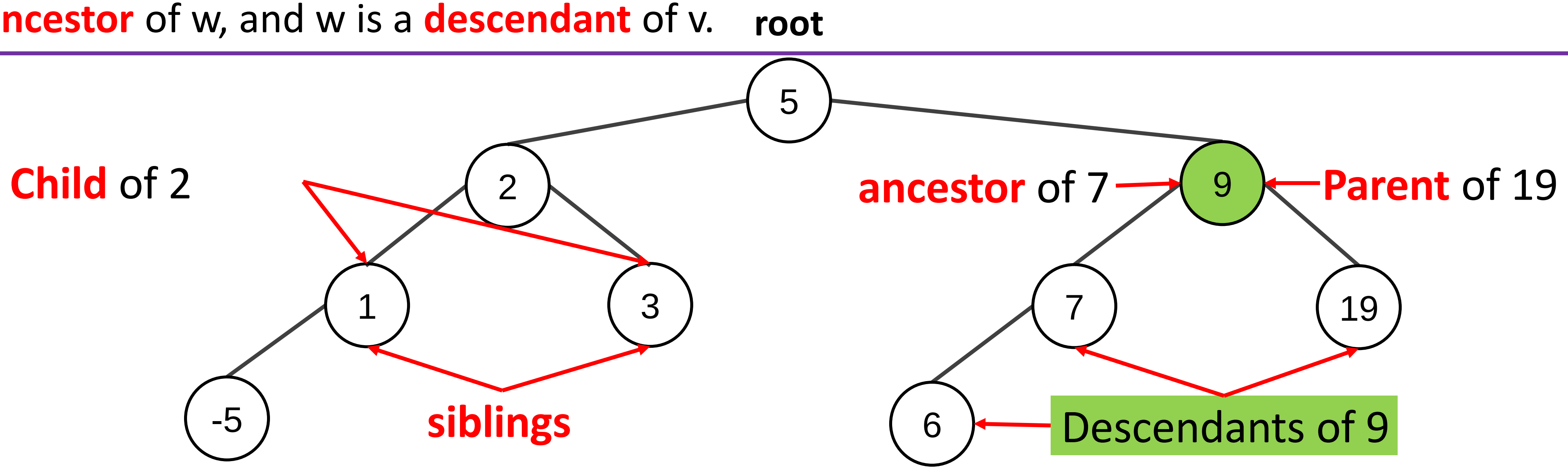
Allen, Weiss Mark. *Data structures and algorithm analysis in C++*. Pearson Education India, 2007.

Parent, Sibling, Ancestor, Descendant

Definition 3. Given the root or any internal vertex v of a rooted tree, the **children** of v are all those vertices that are adjacent to v and are one level farther away from the root than v.

If w is a **child** of v, then v is called the **parent** of w, and **two distinct vertices that are both children of the same parent are called siblings**.

Given two distinct vertices v and w, if v lies on the unique path between w and the root, then v is an **ancestor** of w, and w is a **descendant** of v.



Quiz

Consider the tree with root v_0 shown below.

a. What is the level of v_6?

b. What is the level of v_0?

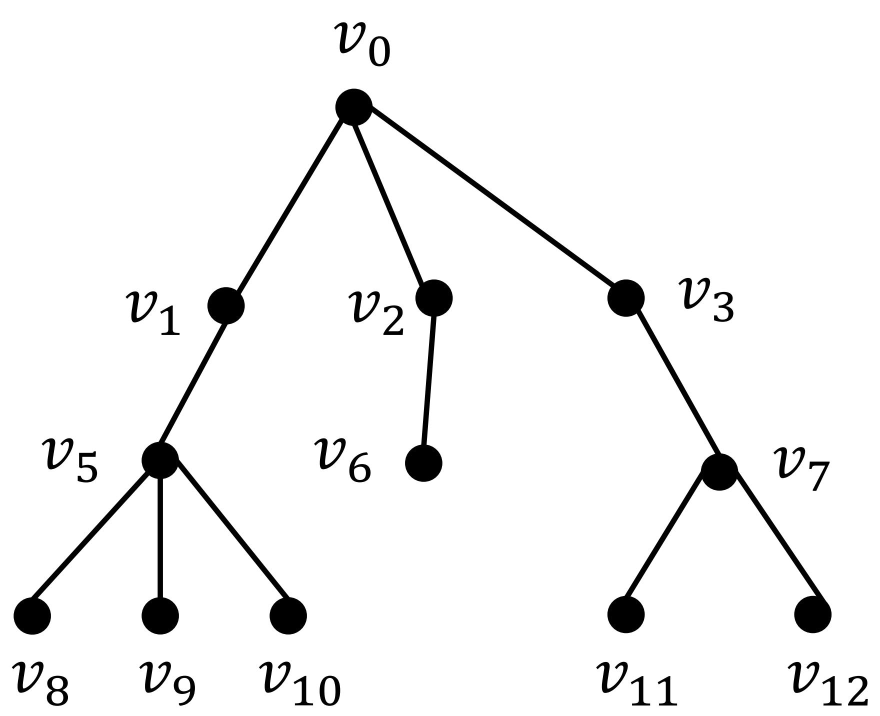
c. What is the height of this rooted tree?

d. What are the children of v_7?

e. What is the parent of v_6?

f. What are the siblings of v_5?

g. What are the descendant of v_1?

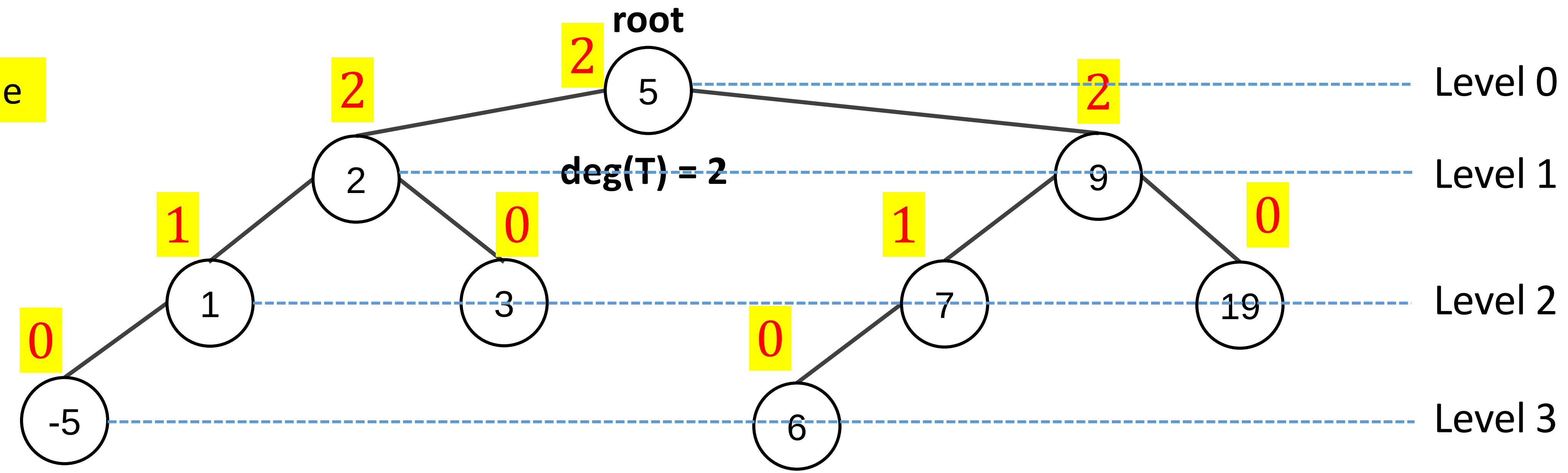


Degree

Definition 4. Degree of a node: the number of subtrees (children) it has.

- Degree of a tree: the largest degree among all nodes in the tree (i.e., the maximum number of children any single node has).
- A tree with degree n is called an n-ary tree.

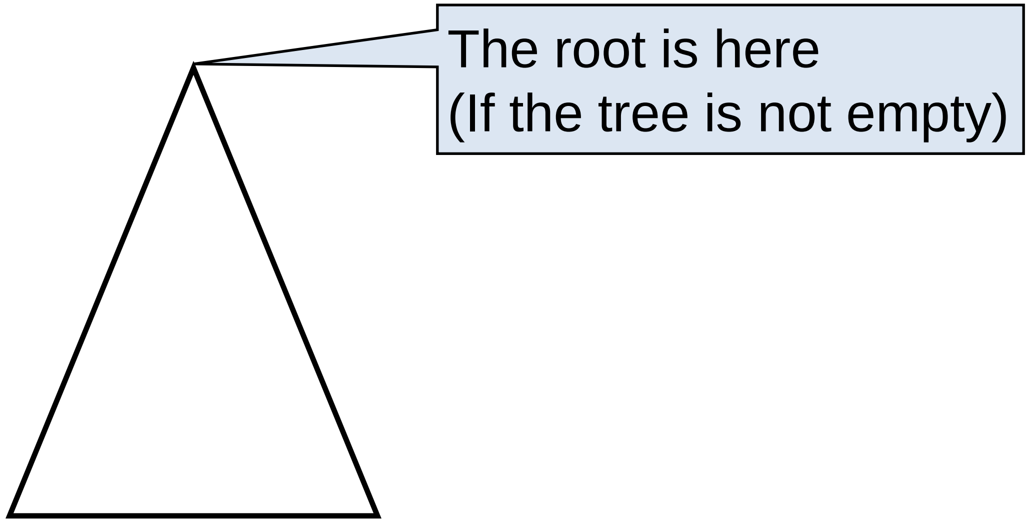
Degree of a node



Abstract illustration

We will often reason about arbitrary trees

- Their actual content is unimportant, so we abstract it away
- We draw a generic tree as a triangle

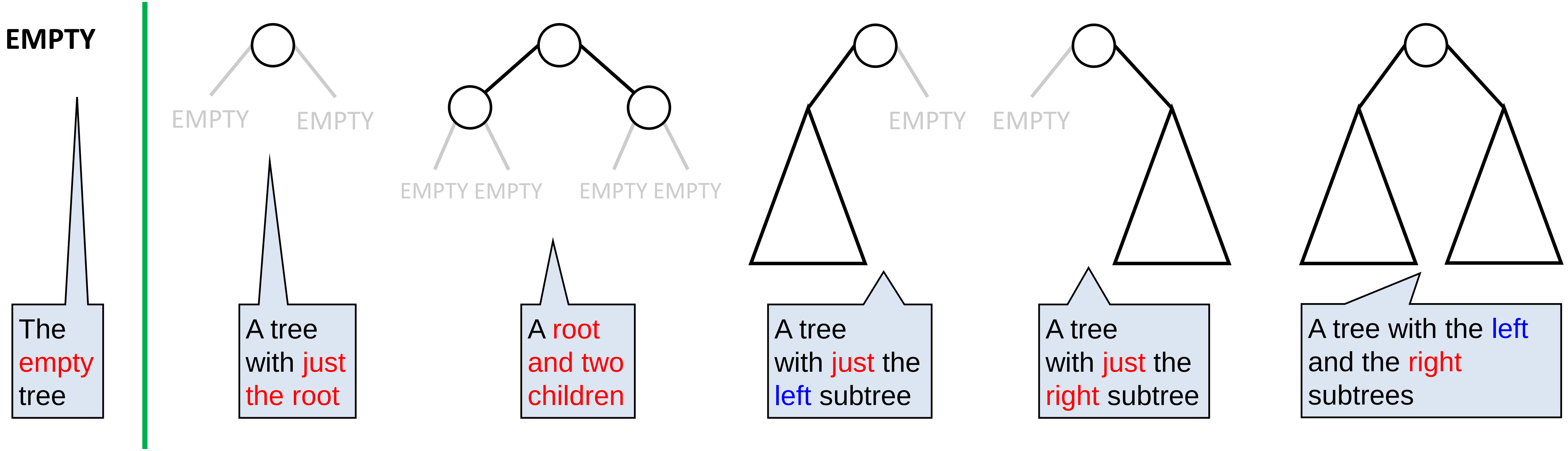


- We represent the empty tree by simply writing “**Empty**”

Empty

Instances of trees

Every tree reduces to TWO instances: either **empty** or a root with a tree on its left and a tree on its right



Outline

1. Introduction
2. Terminologies
3. Tree traversals
   - Breadth-First Search (BFS)
   - Depth-First Search (DFS)
4. Tree representation
5. Binary trees
6. Binary search trees (BST)
7. Balanced trees
8. AVL trees
9. 2-3, 2-3-4 trees
10. B-trees

Binary Tree Traversal

- Tree traversal (also known as tree search) is the process of visiting each node in a tree data structure exactly once in a systematic manner.

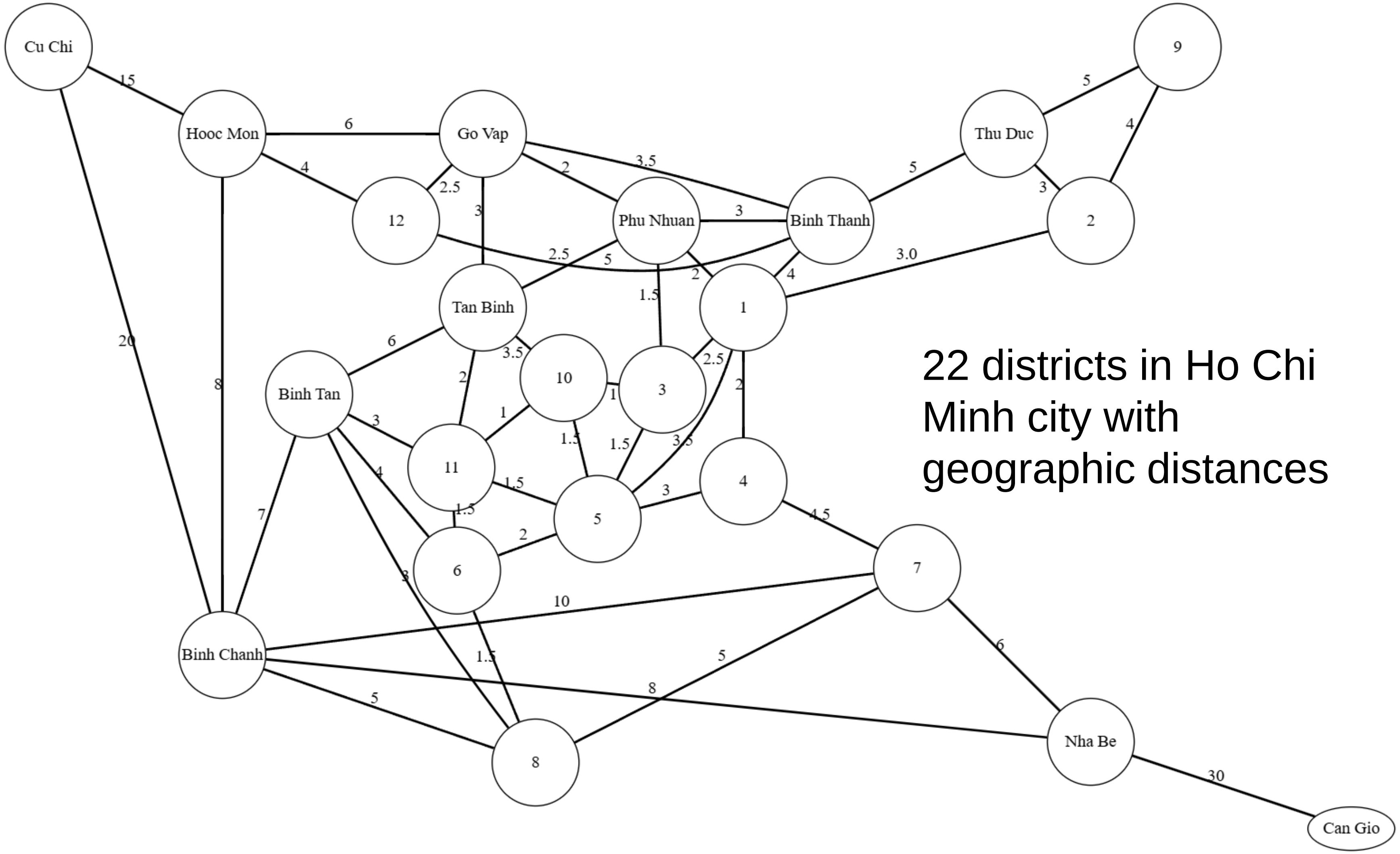
- There are two types of traversal: breadth-first search (BFS) or depth-first search (DFS).

- The following sections describe BFS and DFS on binary trees, but in general, they can be applied to any type of trees, or even graphs.

Breadth-first search (BFS)

- Breadth-first search (BFS) was first invented by Zuse in 1945 [1] in his rejected PhD thesis and later reinvented by E.F. Moore in 1959 [2].

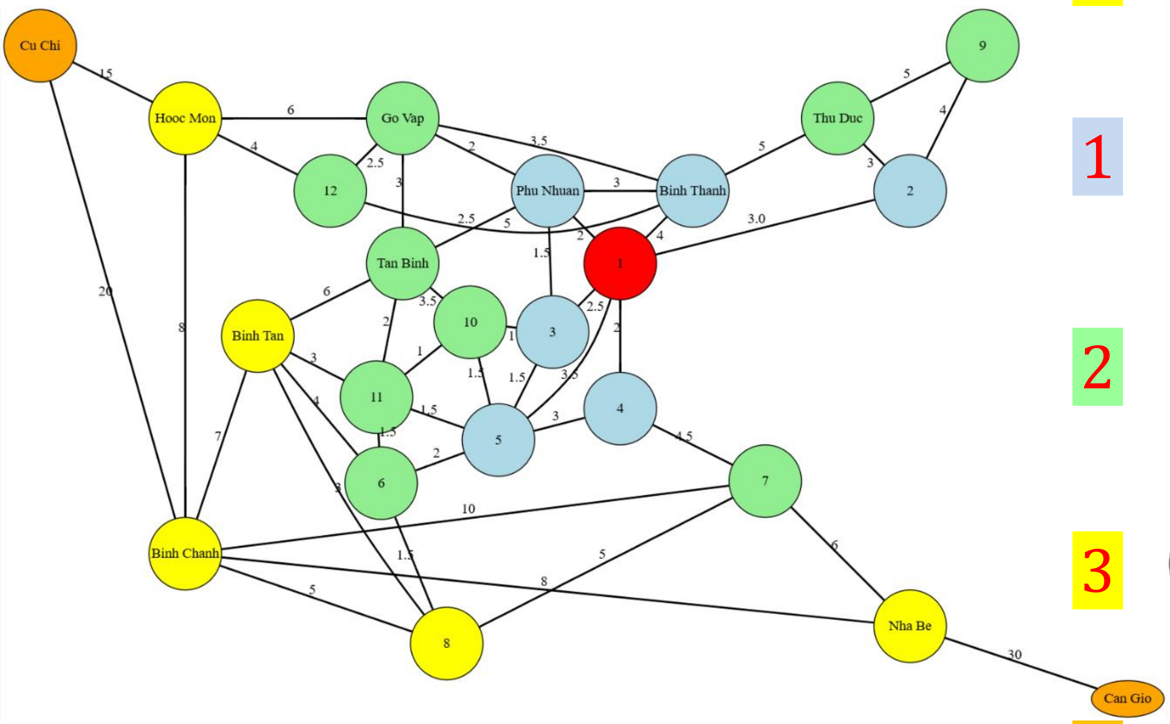
- It starts at the root and visits its adjacent nodes, and then moves to the next level.



22 districts in Ho Chi Minh city with geographic distances

1. Zuse, Konrad. *Der Plankalkül.* (1972).
2. Moore, Edward F. *The shortest path through a maze*. Proc. of the Int. Symp. the Theory of Switching. Harvard Univ. Press, 1959.

BFS for 22 districts in Ho Chi Minh city



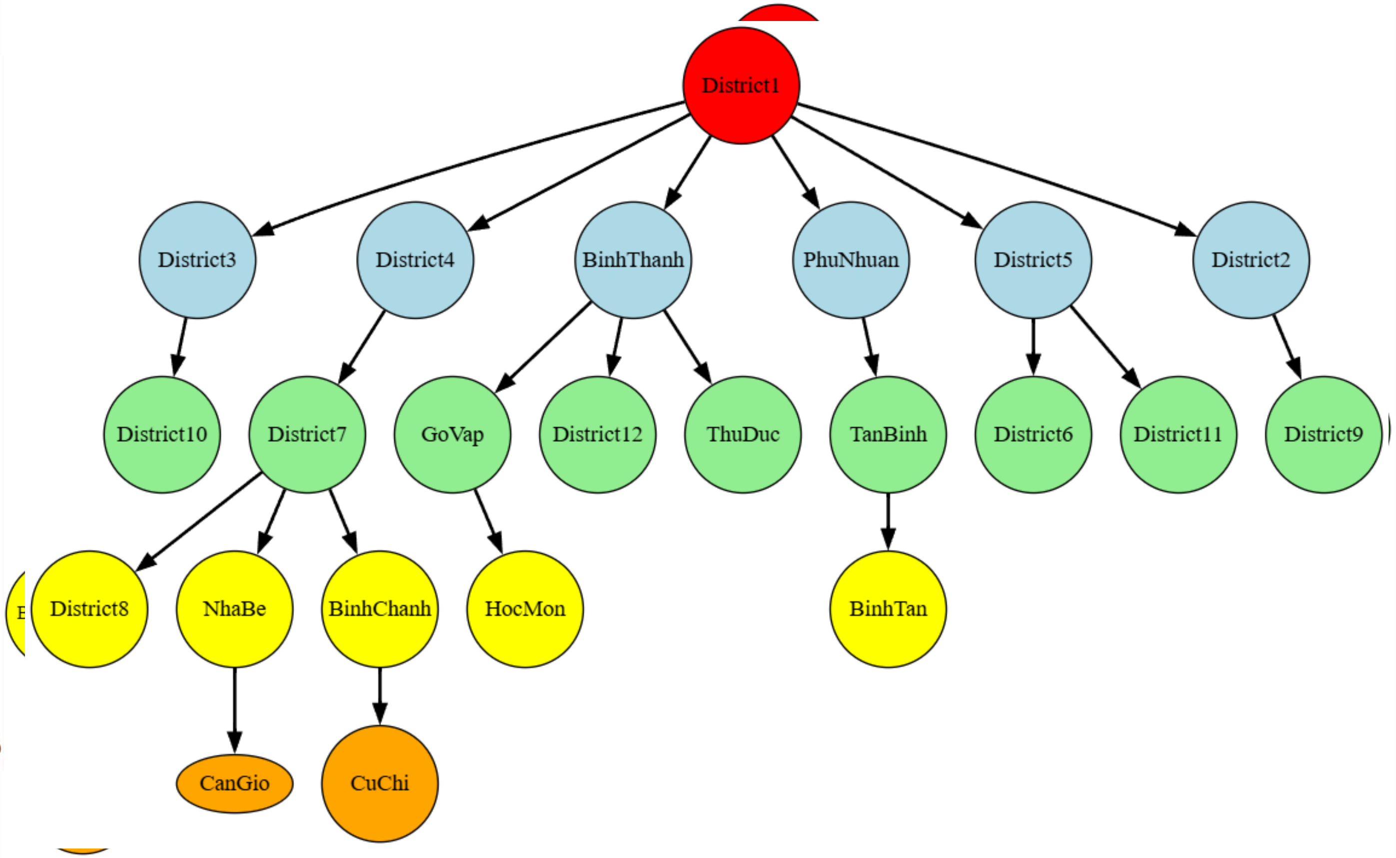
0

1

2

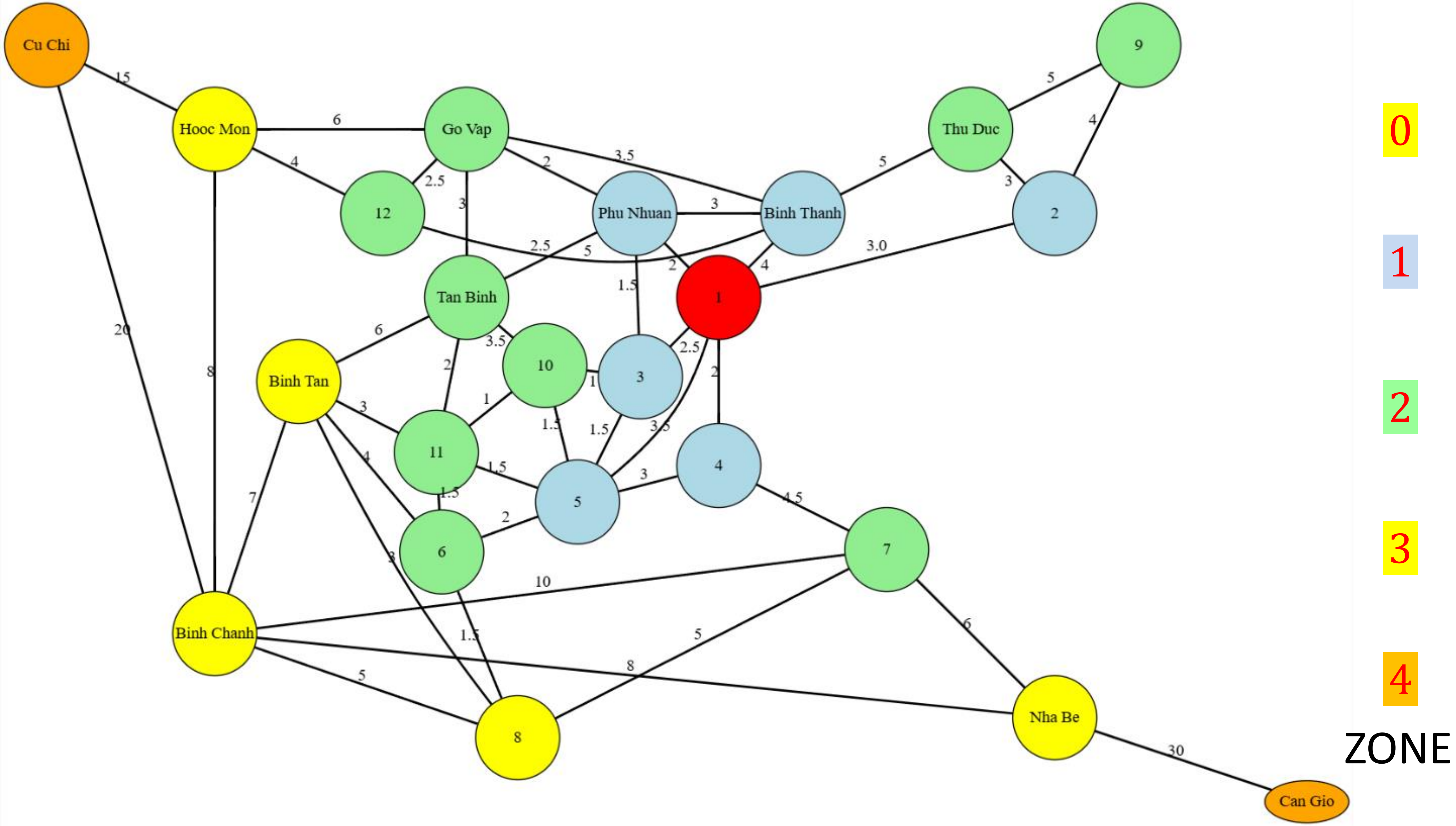
3

4



ZONE

BFS for 22 districts in Ho Chi Minh city



BFS for 22 districts in Ho Chi Minh city

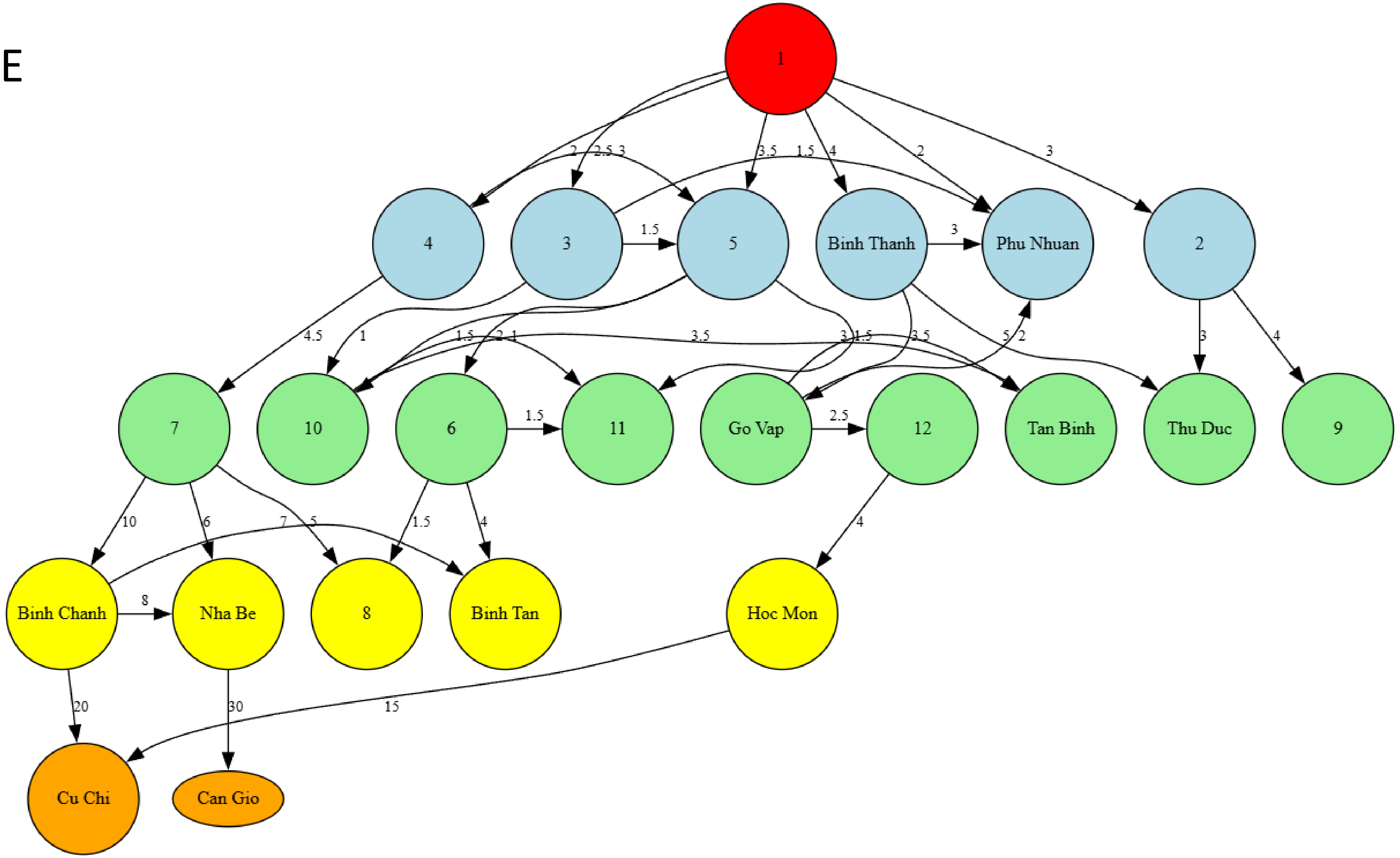
0 ZONE

1

2

3

4



Outline

1. Introduction
2. Terminologies
3. Tree traversals
   – Breadth-First Search (BFS)
   – Depth-First Search (DFS)
4. Tree representation
5. Binary trees
6. Binary search trees (BST)
7. Balanced trees
8. AVL trees
9. 2-3, 2-3-4 trees
10. B-trees

Depth-First Search (DFS)

There are three types of depth-first traversal:

Pre-order (Node-Left-Right)

1. Process the root.
2. Traverse EVERY subtree by recursively calling the pre-order function with the corresponding child.

```
Void NLR(TREE Root) {
  if (Root != NULL) {
    <Process Root>;
    NLR(Root->child_1);
    …
    NLR(Root->child_k);
  }
}
```

In-order (Left-Node-Right)

1. Traverse the MOST LEFT subtree by recursively calling the in-order function with the most left child.
2. Process the root.
3. Traverse the REMAINING subtrees by recursively calling the in-order function with the remaining children.

```
Void LNR(TREE Root) {
  if (Root != NULL) {
    LNR(Root->child_1);
    <Process Root>;
    LNR(Root->child_2);
    …
    LNR(Root->child_k);
  }
}
```

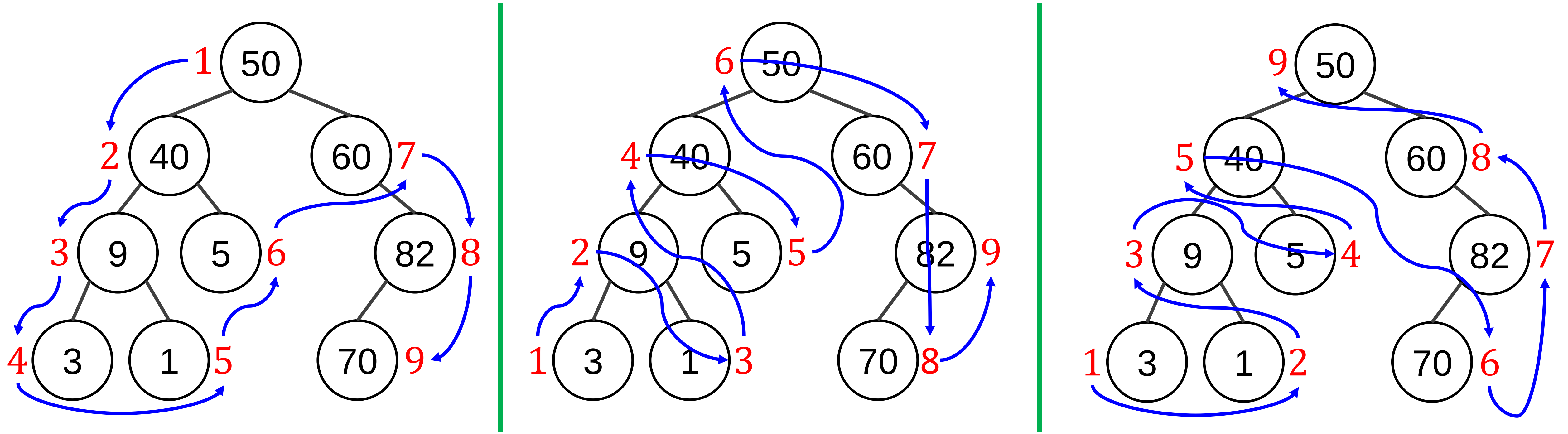
- **Post-order (Left-Right-Node)**

1. Traverse EVERY subtree by recursively calling the post-order function with the corresponding child.
2. Process the root.

```
Void LRN(TREE Root) {
  if (Root != NULL) {
    LRN(Root->child_1);
    …
    LRN(Root->child_k);
    <Process Root>;
  }
}
```

Example (1/2)

Numbers beside nodes indicate traversal order



Pre-order (NLR):
50, 40, 9, 3, 1, 5, 60, 82, 70

In-order (LNR):
3, 9, 1, 40, 5, 50, 60, 70, 82

Post-order (LRN):
3, 1, 9, 5, 40, 70, 82, 60, 50

Example (2/2)

Pre-order

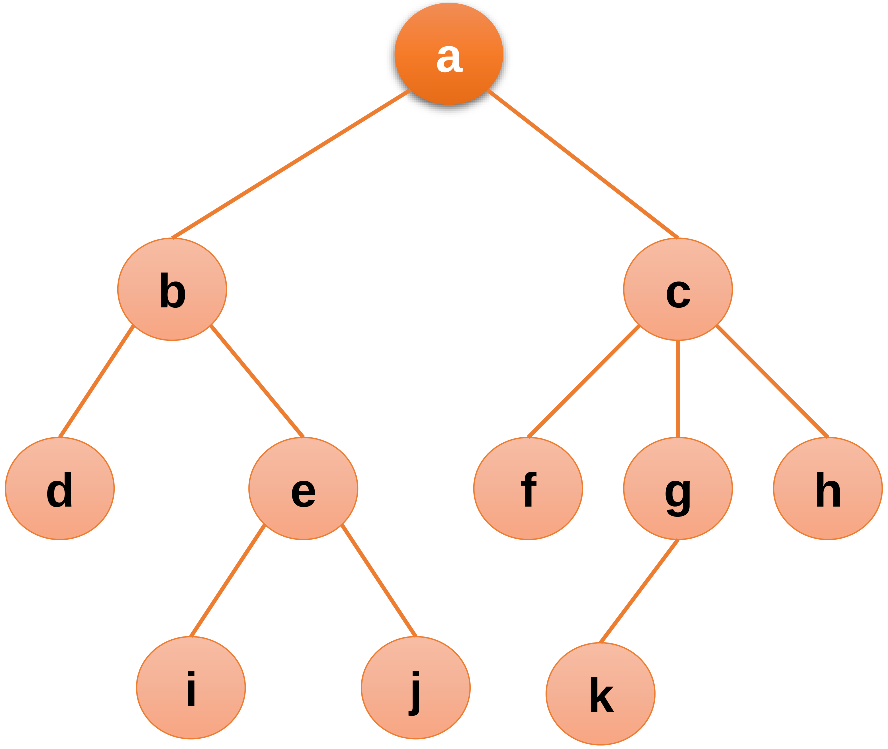
- *a b d e i j c f g k h*

In-order

- d b i e j a f c k g h

Post-order

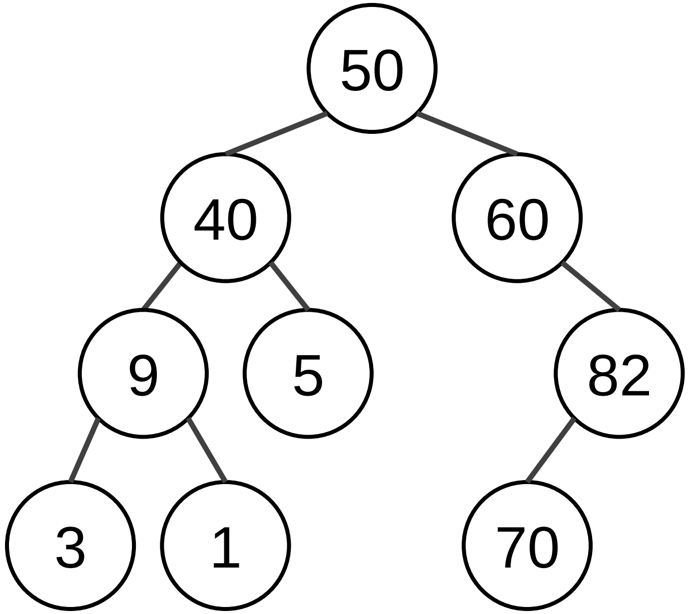
- *d i j e b f k g h c a*



Outline

1. Introduction
2. Terminologies
3. Tree traversals
   – Breadth-First Search (BFS)
   – Depth-First Search (DFS)
4. Tree representation
5. Binary trees
6. Binary search trees (BST)
7. Balanced trees
8. AVL trees
9. 2-3, 2-3-4 trees
10. B-trees

1. Diagram (Visual Tree Drawing)
   - Nodes are circles or boxes.
   - Lines (edges) connect parents to children.
   - Typically drawn top-down.



2. Adjacency List
   - Every node lists its direct children.

- 50: [40, 60]
- 40: [9, 5]
- 60: [82]
- 9: [3, 1]
- 82: [70]
- 3: []
- 1: []
- 5: []
- 70: []

3. Parent Array
   - Store the parent of each node.

- 50: None
- 40: 50
- 60: 50
- 9: 40
- 5: 40
- 82: 60
- 3: 9
- 1: 9
- 70: 82

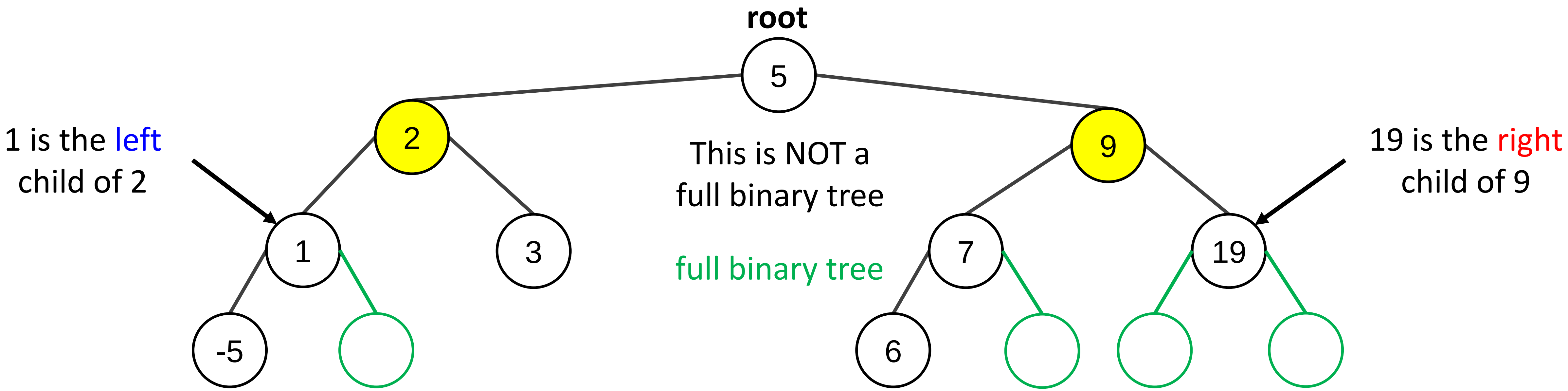
Outline

1. Introduction
2. Terminologies
3. Tree traversals
   - Breadth-First Search (BFS)
   - Depth-First Search (DFS)
4. Tree representation
5. Binary trees
6. Binary search trees (BST)
7. Balanced trees
8. AVL trees
9. 2-3, 2-3-4 trees
10. B-trees

(Full) Binary trees

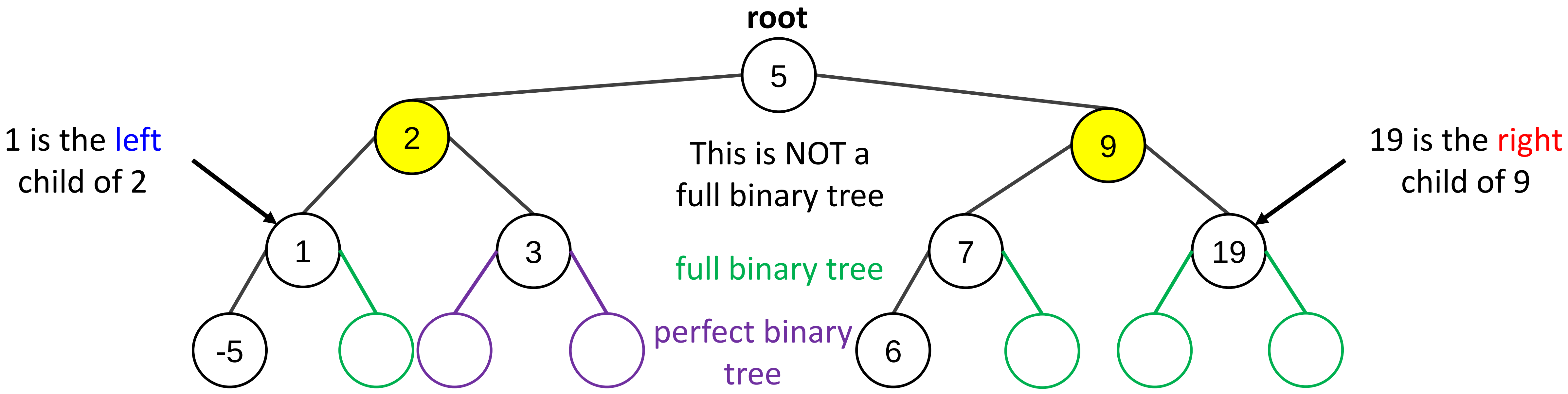
Definition 5 (A (full) binary tree). A **binary tree** is a rooted tree in which every parent has at most two children. Each child is designated either a **left child** or a **right child** (but not both), and every parent has at most one left child and one right child.

A **full binary tree** is a binary tree in which each PARENT has exactly two children.



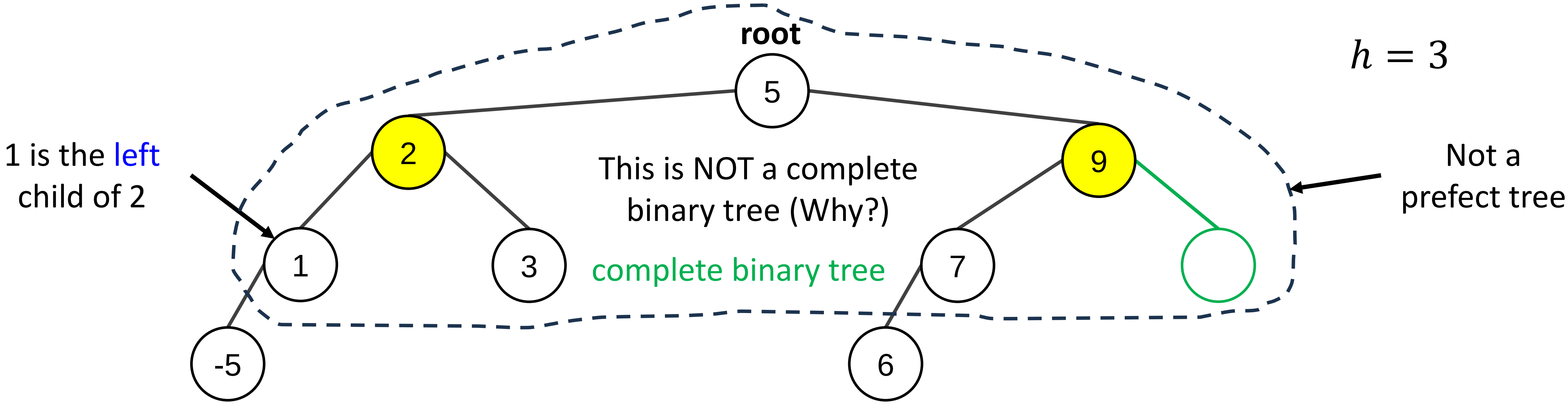
Perfect binary trees

Definition 6 (A perfect binary tree). A **binary tree** is a rooted tree in which every parent has EXACTLY two children and ALL leaves have the same level (depth).



Complete binary trees

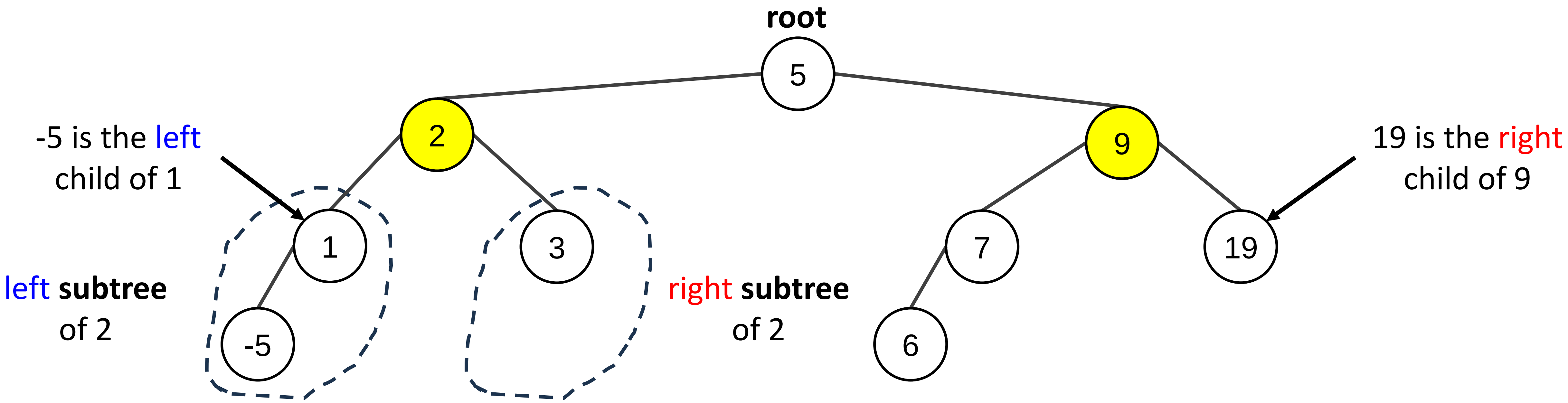
Definition 7 (A complete binary tree). A **complete binary tree** of height h is a rooted tree that is perfect down to level $h-1$.



Left Subtree, Right Subtree

Definition 8. Given any parent v in a binary tree T, if v has a left child, then the left **subtree** of v is the binary tree whose root is the left child of v, whose nodes consist of the left child of v and all its descendants, and whose edges consist of all those edges of T that connect the nodes of the left **subtree**.

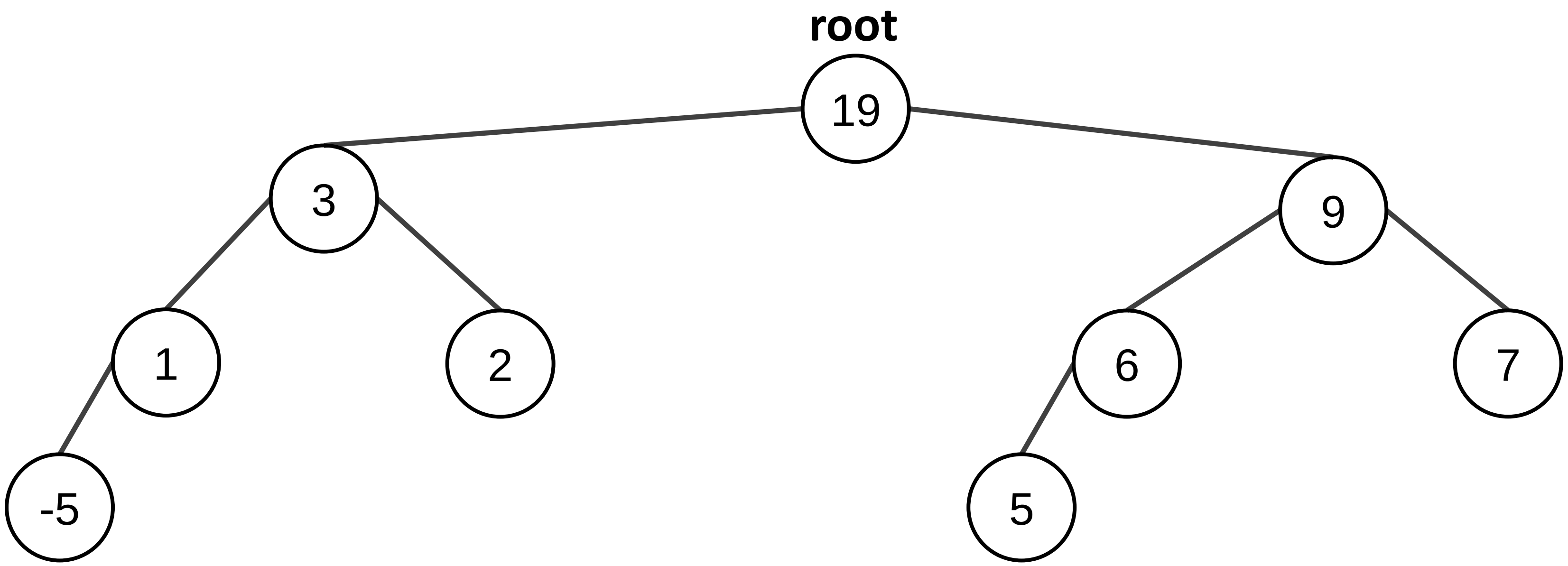
The right **subtree** of v is defined analogously.



Heap as a tree

Definition 9 (A heap (tree)). A **heap** is a complete tree that is either empty or its root:

1. Contains a value greater than or equal to the value in each of its children, and
2. Has heaps as its subtrees



Height and leaves of a Binary Tree

Theorem 1. For non-negative integers h, if T is any binary tree with height h and t leaves, then

$$t \leq 2^h.$$

Equivalently,

$$\log_2 t \leq h.$$

- This theorem says that the maximum number of leaves of a binary tree of height h is 2^h.

  Alternatively, a binary tree with t leaves has height of at least $\log_2 t$.

- **Claim**: The number of nodes at level i is at most 2^i.

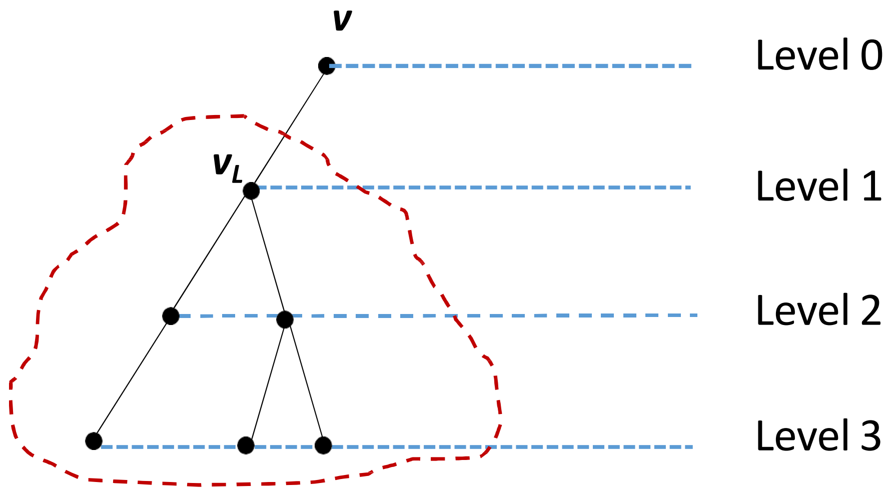
Proof (1/3)

Proof: By mathematical induction

1. Let $P(h)$ be "If T is any binary tree of height h, then the number of leaves of T is at most 2^h.
2. $P(0)$: T consists of one vertex, which is a terminal vertex. Hence $t = 1 = 2^0$.
3. Show that for all integers k □ 0, if $P(i)$ is true for all integers i from 0 through k, then $P(k+1)$ is true.
4. Let T be a binary tree of height $k + 1$, root v, and t leaves.
5. Since $k \geq 0$, hence $k + 1 \geq 1$ and so v has at least one child.
6. We consider two cases: If v has only one child, or if v has two children.

Proof (2/3): Case 1 (*v* has only one child)

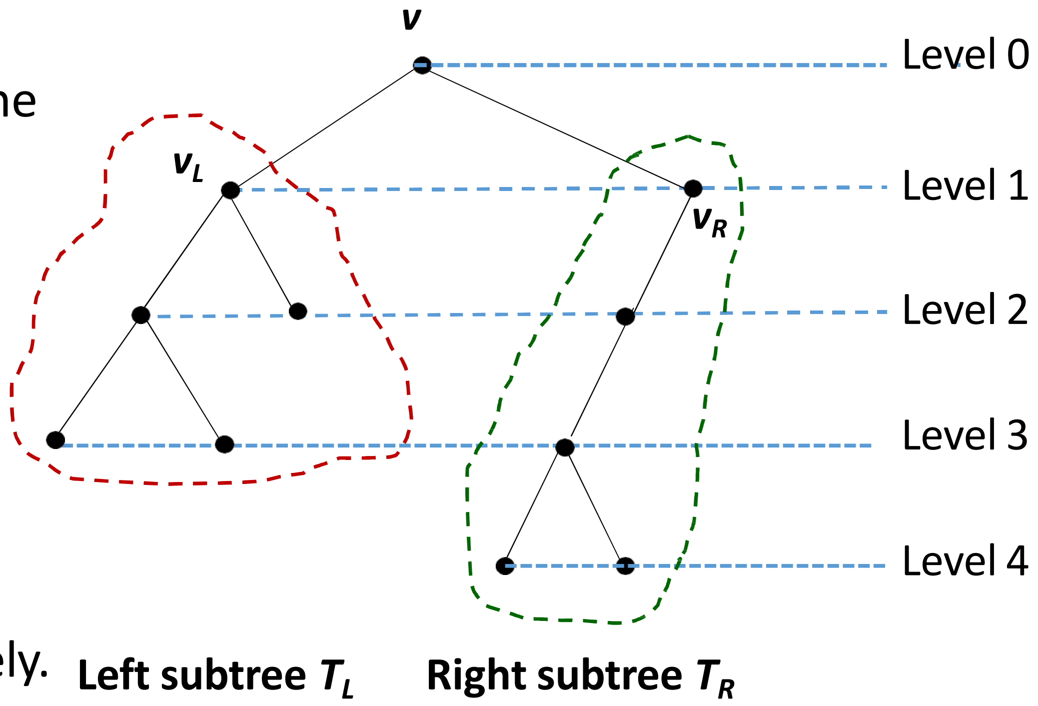
1. Without loss of generality, assume that *v*'s child is a left child and denote it by v_L. Let T_L be the left subtree of *v*.
2. Because *v* has only one child, *v* is a leaf, so the total number of leaves in *T* equals the number of leaves in $T_L + 1$. Thus, if t_L is the number of leaves in T_L, then $t = t_L + 1$.
3. By inductive hypothesis, $t_L \leq 2^k$ because the height of T_L is *k*, one less than the height of *T*.
4. Also, because *v* has a child, $k+1 \geq 1$ and so $2^k \geq 2^0 = 1$.
5. Therefore, $t = t_L + 1 \leq 2^k + 1 \leq 2^k + 2^k = 2^{k+1}$



Left subtree T_L

Proof (3/3): Case 2 (*v* has two children)

1. Now v has a left child v_L and a right child v_R , and they are the roots of a left subtree T_L and a right subtree T_R respectively.
2. Let h_L and h_R be the heights of T_L and T_R respectively.
3. Then $h_L \le k$ and $h_R \le k$ since T is obtained by joining T_L and T_R and adding a level.
4. Let t_L and t_R be the number of leaves of T_L and T_R respectively.
5. Then, since both T_L and T_R have heights less than k + 1, by inductive hypothesis, $t_L \le 2^{hL}$ and $t_R \le 2^{hR}$.
6. Therefore, $t = t_L + t_R \le 2^{hL} + 2^{hR} \le 2^k + 2^k \le 2^{k+1}$



- We proved both cases that $P(k+1)$ is true.
- Hence if T is any binary tree with height h and t terminal vertices (leaves), then $t \le 2^h$.

Quiz

Q: Is there a binary tree that has height 6 and 70 leaves?

No, by Theorem 1, any binary tree *T* with height 6 has at most $2^6 = 64$ leaves, so such a tree cannot have 70 leaves.

Given a binary tree *T* height of h.

- What is the maximum number of nodes?
- What is the minimum number of nodes?

- Given a binary tree *T* with n nodes.
- What is the maximum height of that tree?
- What is the minimum height of that tree?

Outline

1. Introduction
2. Terminologies
3. Tree traversals
   - Breadth-First Search (BFS)
   - Depth-First Search (DFS)
4. Tree representation
5. Binary trees
6. Binary search trees (BST)
7. Balanced trees
8. AVL trees
9. 2-3, 2-3-4 trees
10. B-trees

Expected Searching Time

	Design	Run time					Space	Collision
		Cost	Search	Insert	Delete	Find_min		
Unsorted array	Det.		$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	
Sorted array by key			$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	
Singly Linked list			$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	
Hashing	Rnd.		$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(p)$ $p < n$	YES
Binary Search Tree	Det.		$O(\log n)$					NO

✓ ? ? ?

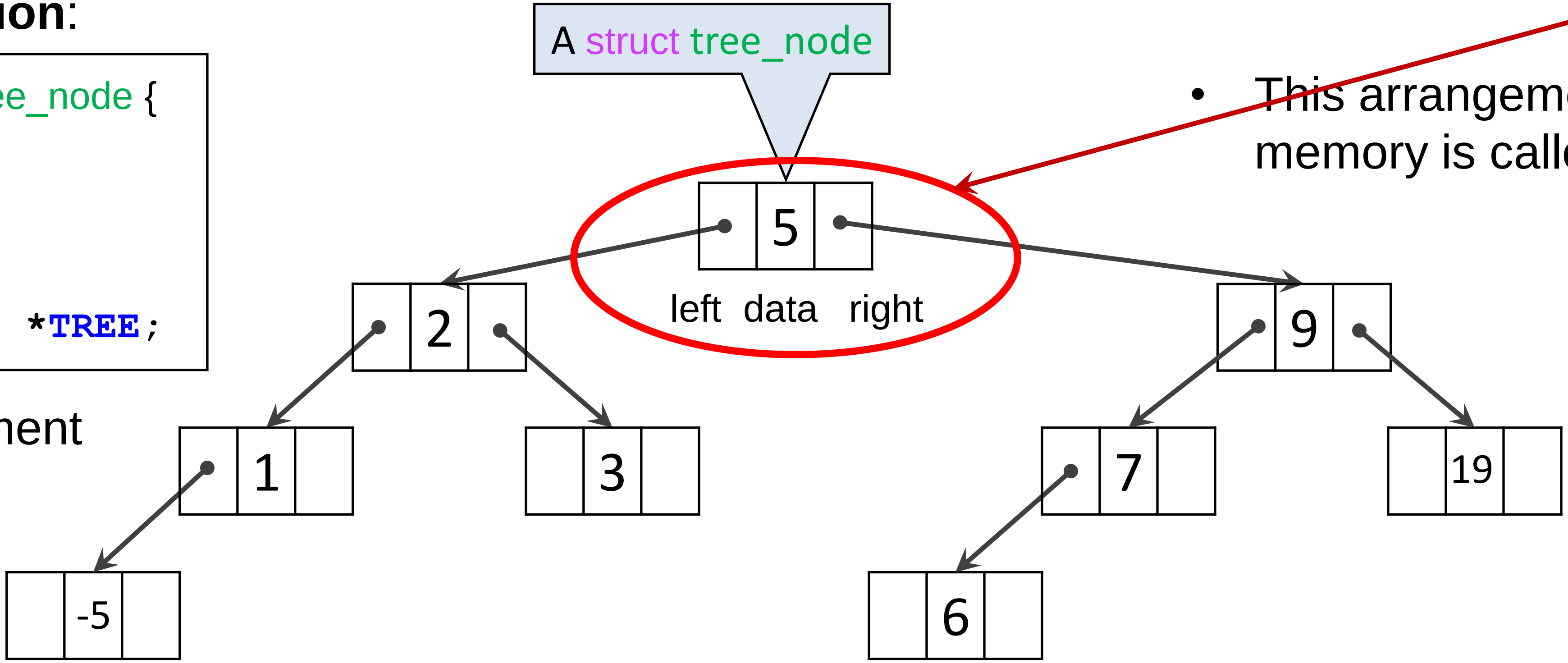
Implementation: Binary Search Tree (1/2)

We can capture this idea with this **type declaration**:

```
typedef struct tree_node {
  tree* left;
  int data;
  tree* right;
} TNODE;
typedef  TNODE  *TREE;
```

- A data element
- Pointers to the 2 elements we may look at next

- This struct is called a **node**
- This arrangement of data in memory is called a **tree**



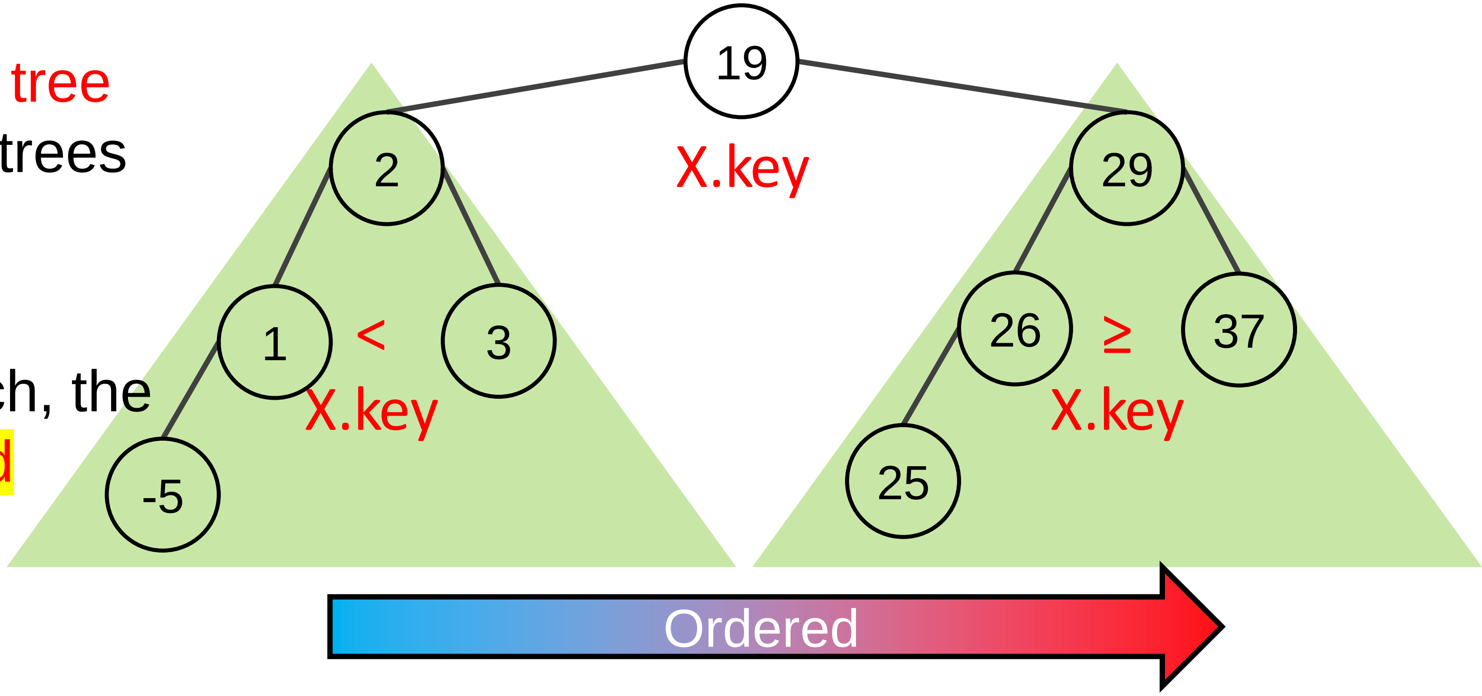
- Last time we transform a sorted array to a BST.
- How can we create a BST from an ARBITRARY array?

Binary Search Tree (BST): definition

Definition 10 (Binary Search Tree (BST)). A tree is called a binary search tree (BST) if for ANY INTERNAL node in the tree:

1. All nodes in the left subtree of a node have a value < than the value of the node itself.
2. All nodes in the right subtree of a node have a value > than the value of the node itself.

- What additional constraints on the tree representation do we need to use trees to implement?
- Because search uses binary search, the data in the tree need to be ordered
  - Smaller values on the left
  - Bigger values on the right



Common operations

- Finding
- Insertion
- Deletion

1. Start at the root.
2. If the current node is X, you're done.
3. If X is smaller, go to the left child.
4. If X is larger, go to the right child.
5. Repeat until you find X or reach an empty node (which means X is not in the tree).

Source code

Find an element X in the tree (recursively)

```
TNODE* searchNode(TREE T, Data X) {
   if (T) {
     if (T->Key == X)
       return T;
     if(T->Key > X)
       return searchNode(T->pLeft, X);
     return searchNode(T->pRight, X);
   }
   return NULL;
}
```

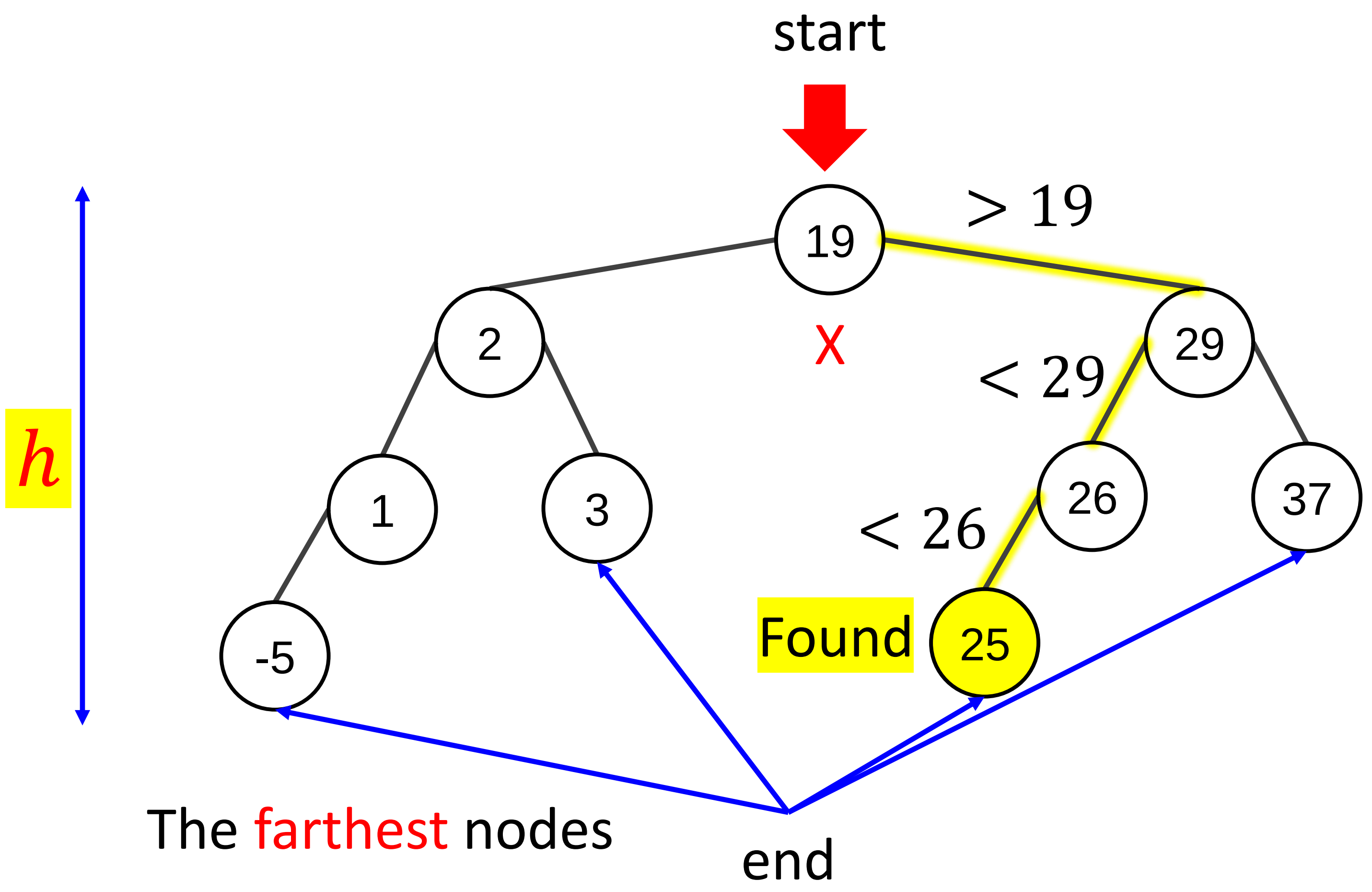
Find an element X in the tree (non-recursively)

```
TNODE* searchNode(TREE T, Data X) {
   if (T) {
     if (T->Key == X)
       return T;
     if(T->Key > X)
       return searchNode(T->pLeft, X);
     return searchNode(T->pRight, X);
   }
   return NULL;
}
```

Complexity

Finding takes $O(h)$, where h is the height of the tree.

Find 25



Common operations

- Finding
- Insertion
- Deletion

1. Start at the root.
2. If X is smaller than the current node, go to the left child.
3. If X is larger, go to the right child.
4. When you reach an empty spot, insert X there as a new node.

Source code

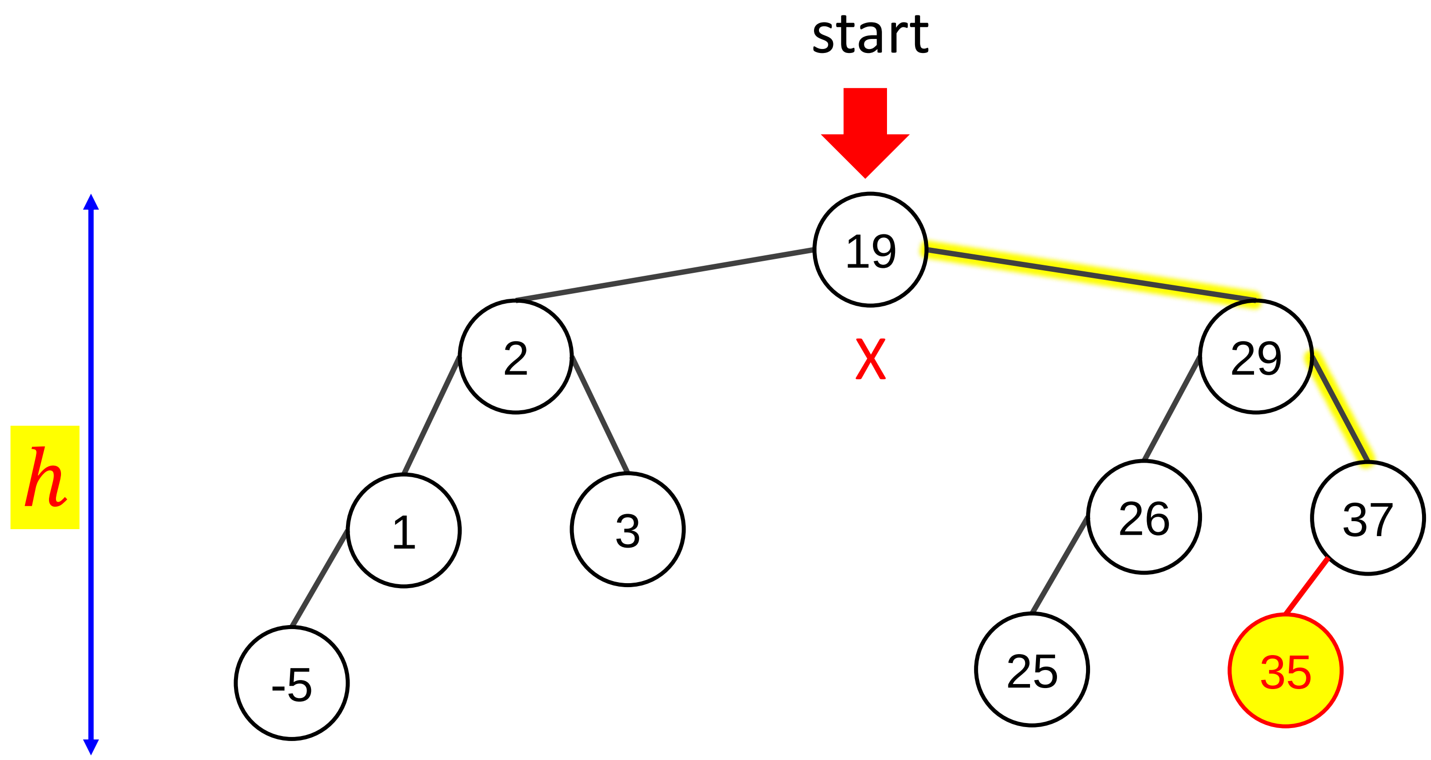
The insert function returns –1, 0, or 1 when there is insufficient memory, a duplicate node is found, or the insertion is successful.

```
int insertNode(TREE &T, Data X)
{ if (T) {
        if(T->Key == X) return 0; // đã có
        if(T->Key >  X)
                return insertNode(T->pLeft, X);
        else
                return insertNode(T->pRight, X);
  }
  T     = new TNode;
  if (T == NULL) return -1; // insufficient memory
  T->Key    = X;
  T->pLeft = T->pRight = NULL;
  return 1;   // success
}
```

Complexity

Insertion takes $O(h)$, where h is the height of the tree.

Insert 35



Exercises

1. Beginning with an empty binary search tree, what binary search tree is formed when you insert the following values in the order given?

   15, 5, 12, 8, 23, 1, 17, 21

2. Beginning with an empty binary search tree, what binary search tree is formed when you insert the following values in the order given?
   - W, T, N, J, E, B, A
   - W, T, N, A, B, E, J
   - A, B, W, J, N, T, E
   - B, T, E, A, N, W, J

Common operations

- Finding
- Insertion
- Deletion

- Removing an element X from the tree must ensure the constraints of the BST are maintained.

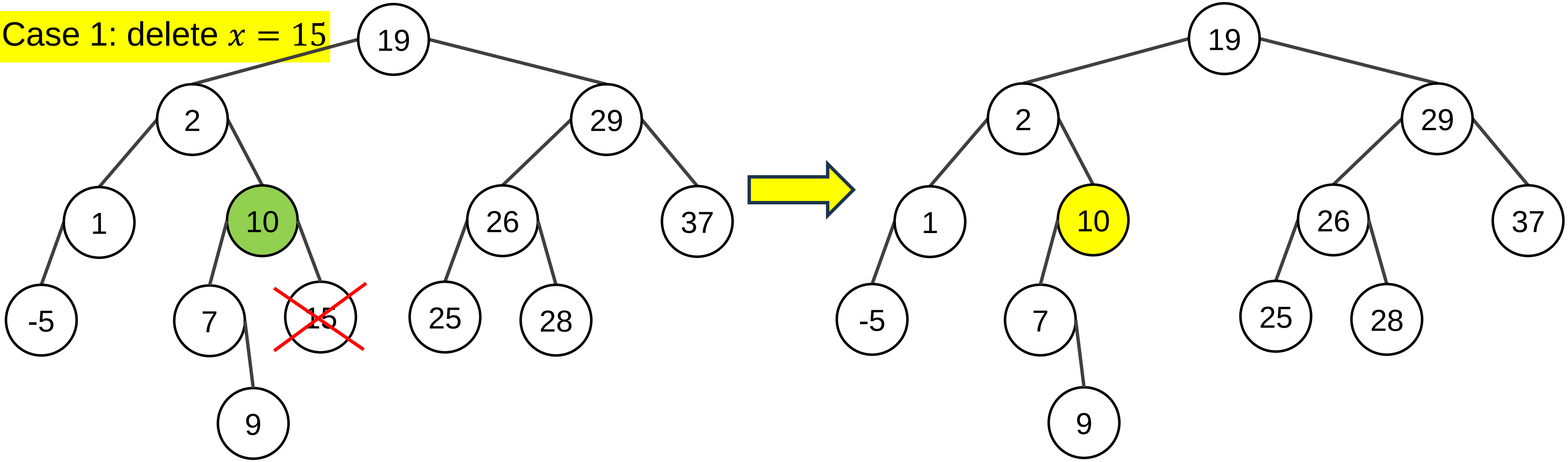
- There are 3 possible cases when deleting node X:

  1. X is a leaf node.
  2. X has only one child (left or right).
  3. X has both left and right children.

Case 1: X is a leaf node

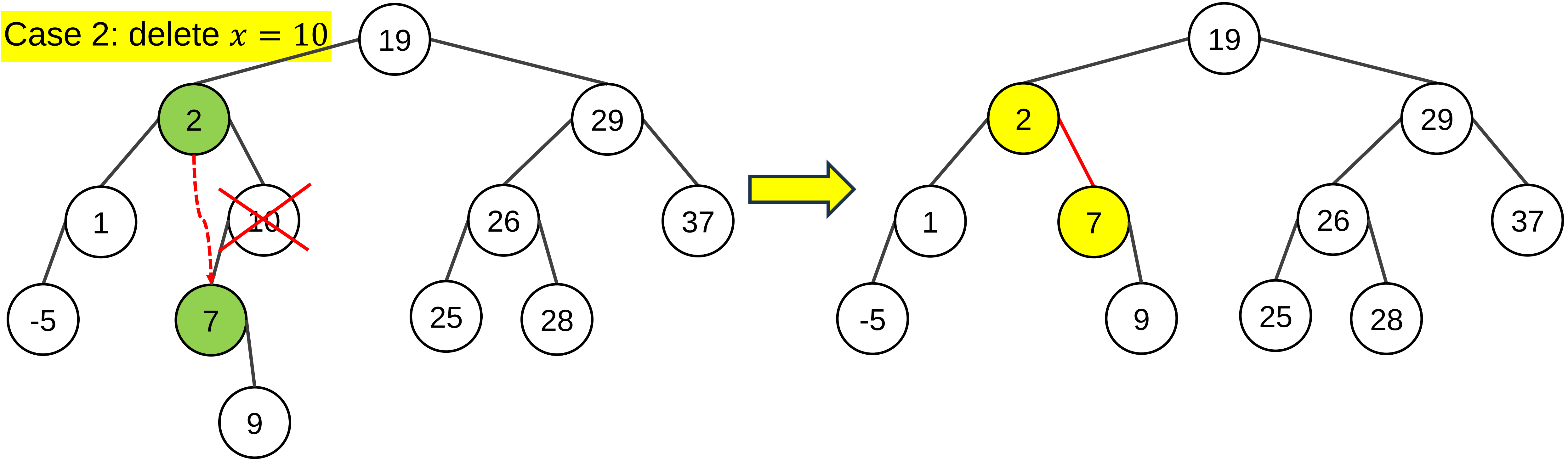
- Simply delete X because it is not connected to any other element

Case 1: delete $x = 15$



Case 2: X has only one child (left or right)

- Before deleting X, we link X's parent to its only child



Case 3: X has both left and right children (1/4)

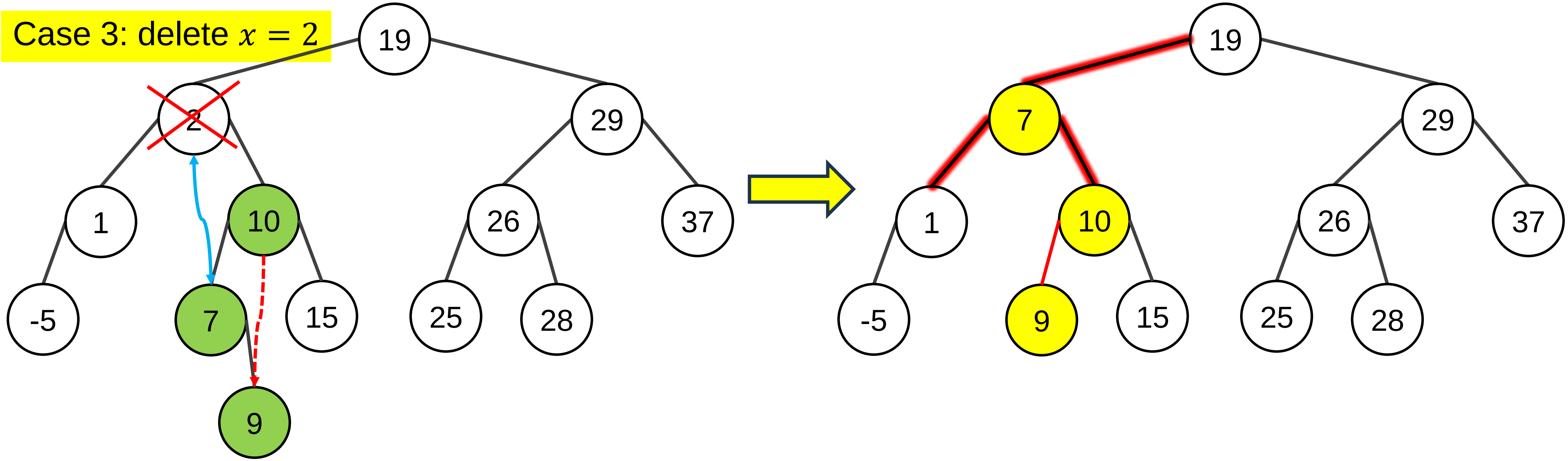
- Cannot delete directly because X has two children.
- Indirect deletion:
  - Instead of deleting X, find a replacement node Y. This node has at most one child.
  - The information stored in Y will be moved up to X.
  - Then, the actual node deleted will be Y, which now falls under one of the two simpler cases (leaf or one child).
- **Issue**: Choose Y so that when Y is placed at X's position, the tree remains a valid BST.

Case 3: X has both left and right children (2/4)

- The issue is to choose Y so that when Y is placed in the position of X, the tree remains a Binary Search Tree (BST).
- There are two elements that satisfy this requirement:
  - The smallest element (leftmost node) in the right subtree.
  - The largest element (rightmost node) in the left subtree.
- Choosing which of these elements to use as the replacement entirely depends on the programmer's preference.
- In this case, we will choose the SMALLEST element in the RIGHT subtree as the replacement.

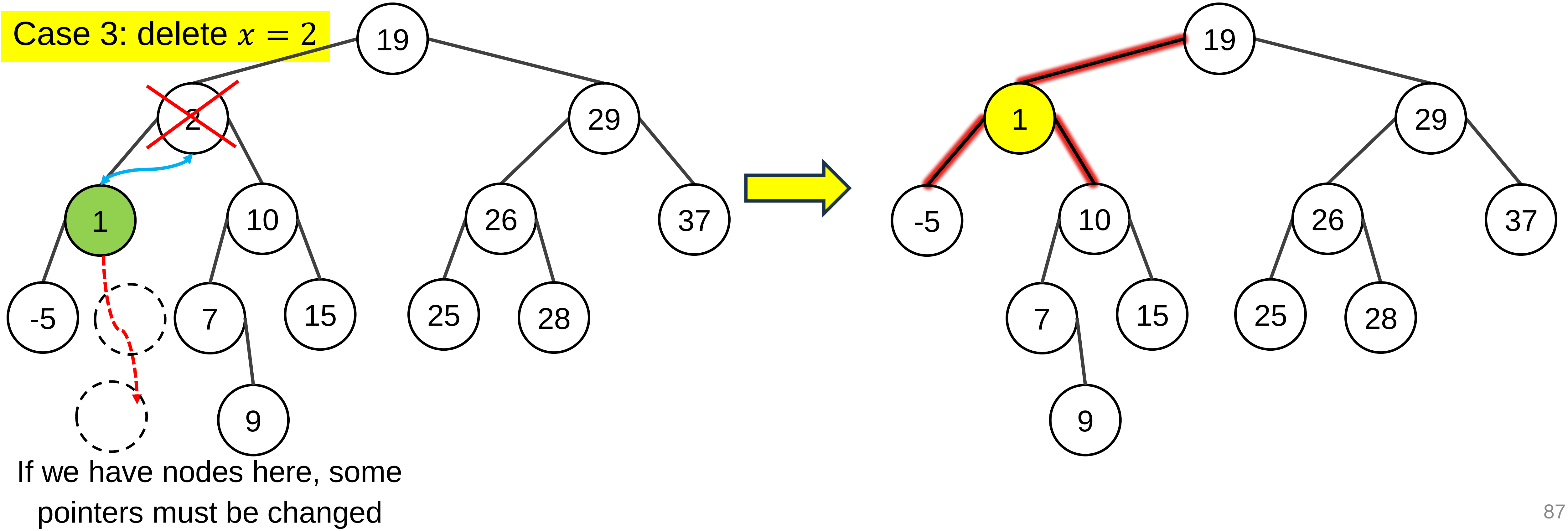
Case 3: X has both left and right children (3/4)

- When deleting the element $x = 2$ from the tree, the element 7 (the leftmost element in the right subtree of node 2) is the replacement element.



Case 3: X has both left and right children (4/4)

- When deleting the element $x = 2$ from the tree, the element 1 (the rightmost element in the left subtree of node 2) is the replacement element.

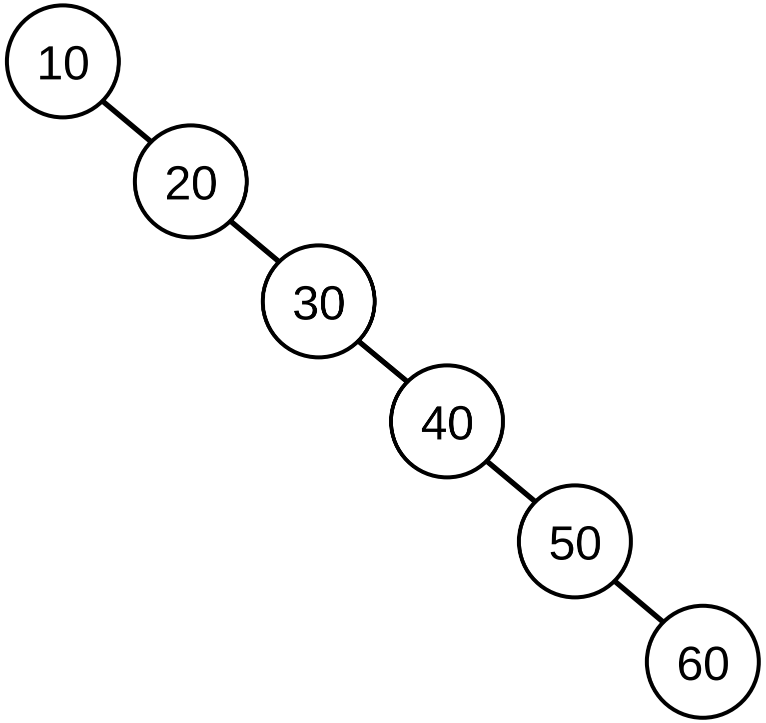


Outline

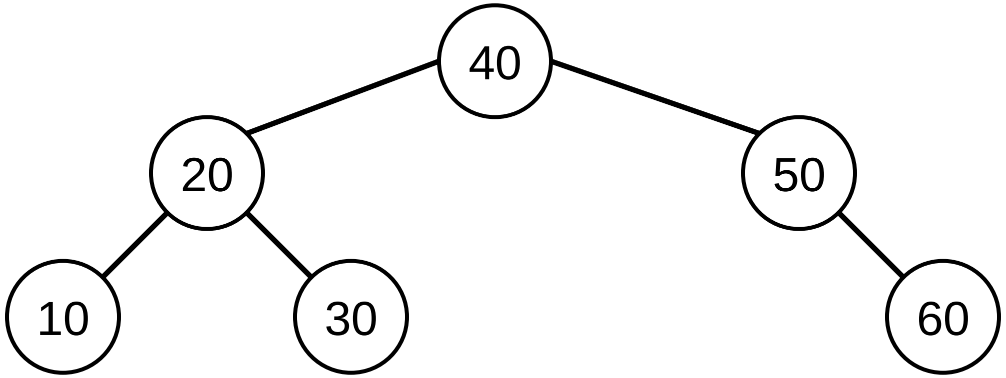
1. Introduction
2. Terminologies
3. Tree traversals
   - Breadth-First Search (BFS)
   - Depth-First Search (DFS)
4. Tree representation
5. Binary trees
6. Binary search trees (BST)
7. Balanced trees
8. AVL trees
9. 2-3, 2-3-4 trees
10. B-trees

An Equivalent Tree

- Is there a BST with the same elements that yields $O(\log n)$ cost?



- How about the following one?

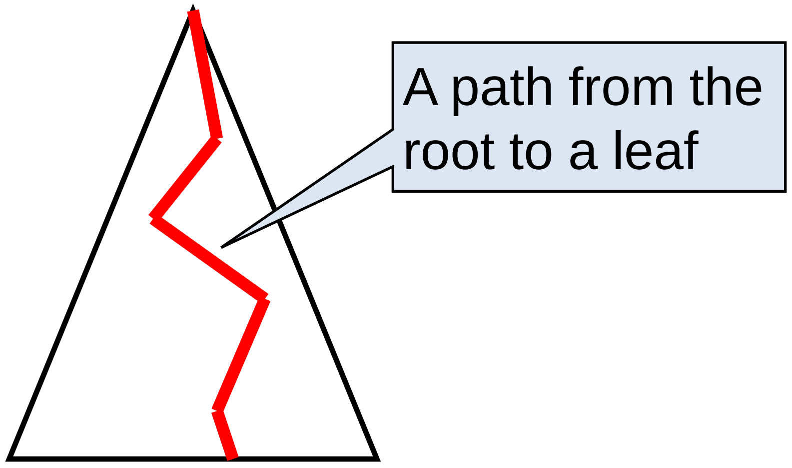


  – It contains the same elements
  – It is sorted
  – But the nodes are arranged differently

Motivation (1/2)

- Depending on the tree, BST search can cost
  - $O(\log n)$
  - Or $O(n)$

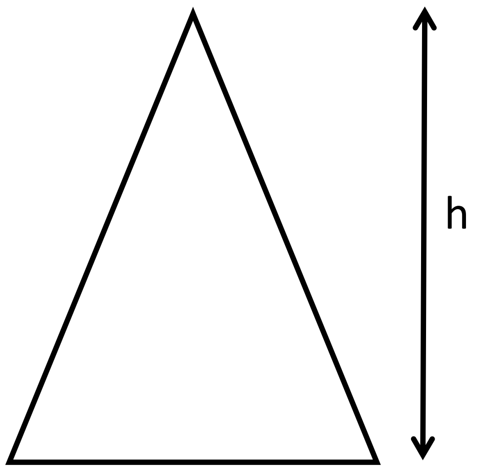
- Can we define a cost metric that yields the same complexity in all cases?
  - The cost of search depends on how deep we need to go in the tree.
    - If the key is present, the worst case occurs when it's in a leaf.
    - If the key is absent, we must still reach a leaf to confirm that.



Motivation (2/2)

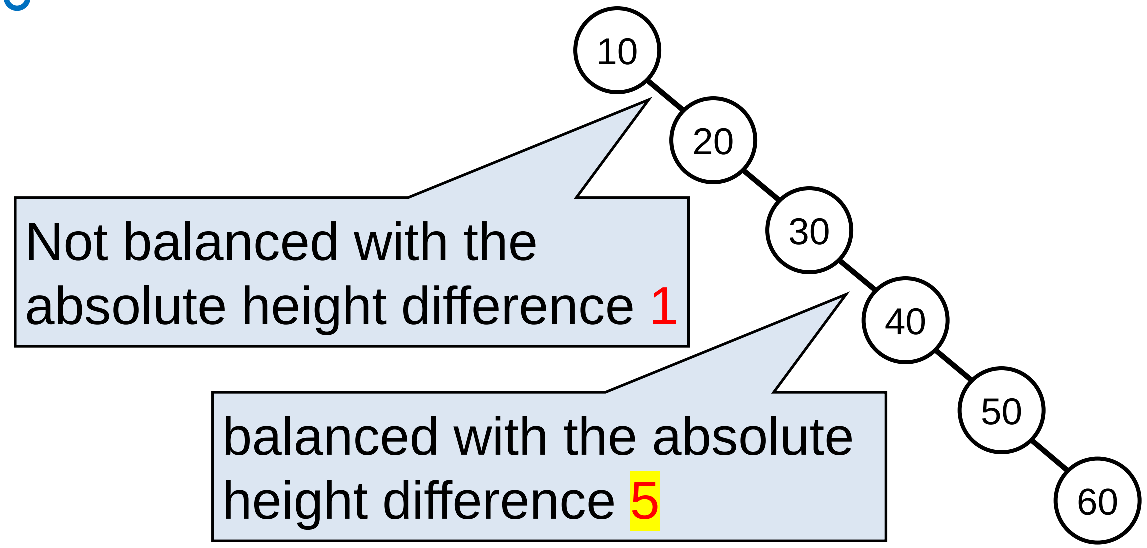
- search for a tree of height h has complexity $O(h)$
  - Always!
  - Same for insert and find_min

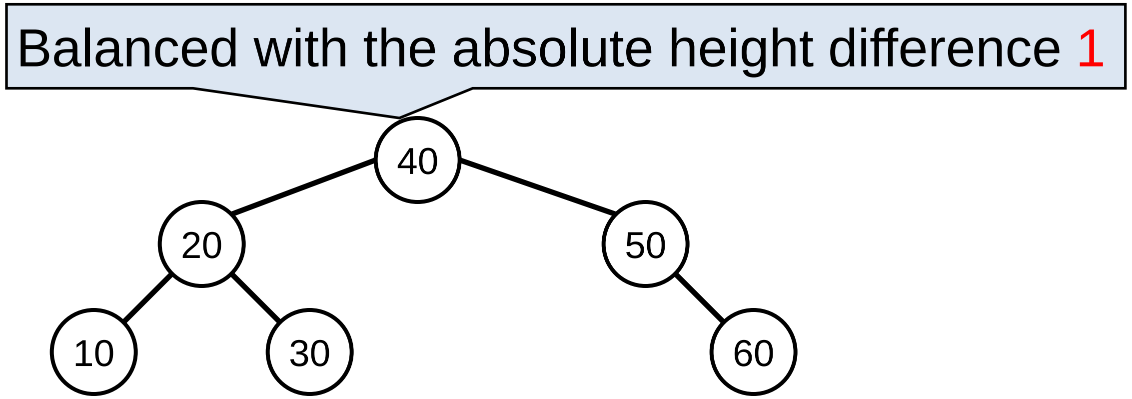
- However …
  - h can be in $O(n)$ or in $O(\log n)$
    - where n is the number of nodes in the tree



Definition 11 (A balanced tree). A **balanced tree** is a rooted tree in which the height difference between the left and right subtrees of any node is bounded by a constant.

Theorem 2. Given a balanced tree with n nodes in which the absolute height difference between the left and right subtrees of any node is at most m, the maximum height of the balanced tree is $O(\log n)$.
Consequently, search, insert, and find_min cost $O(\log n)$.





Height of a balanced tree (1/3)

- Let $N(h)$ be the minimum number of nodes in a balanced tree of height h satisfying the given condition.
- Consider a balanced tree of height h.
- Its root has two subtrees, a left subtree of height h_L and a right subtree of height h_R.
- Since the height of the tree is h, at least one of these subtrees must have height $h - 1$ (assuming a non-empty tree). Without loss of generality, let's say
$$h_L = h - 1.$$
- According to the balancing condition, the height difference between the left and right subtrees is at most m:
$$|h_L - h_R| \leq m.$$
- Since $h_L = h - 1$, we have
$$h - 1 - m \leq h_R \leq h - 1.$$

Height of a balanced tree (2/3)

- To minimize the total number of nodes for a given height h, we should minimize the number of nodes in the right subtree, which occurs when its height is as small as possible while still satisfying the balancing condition.
- Thus, we consider the case where
$$h_R = h - 1 - m.$$
- The minimum number of nodes $N(h)$ in a balanced tree of height h will then be the sum of the nodes in the root, the minimum number of nodes in the left subtree of height $h - 1$, and the minimum number of nodes in the right subtree of height $h - 1 - m$:
$$N(h) = 1 + N(h-1) + N(h-1-m)\ (1).$$

Height of a balanced tree (3/3)

- We will prove $N(h) \geq c\phi^h$ for some positive constants $c, \phi > 0$, where c and ϕ will be chosen later.
- Then by substituting $n = N(h)$, we can get
$$h \leq \log_\phi \frac{n}{c} = O(\log n).$$
- We prove by induction.
- Suppose that $N(h) \geq c\phi^h$ is true until $h = k$. Consider $h = k + 1$. By induction, we have
$$N(k+1) = 1 + N(k) + N(k-m) \geq 1 + c\,\phi^k + c\phi^{k-m}.$$
- We need to prove the following inequality:
$$1 + c\,\phi^k + c\phi^{k-m} \geq c\phi^{k+1}.$$
- Indeed, we have
$$(1) \Leftrightarrow 1 \geq c\phi^{k-m}(\phi^{m+1} - \phi^m - 1) = c\phi^{k-m} f(\phi) = 0.$$
- by choosing ϕ as a root of the equation $f(x) = x^{m+1} - x^m - 1$.
  - Since $f(1)f(2) < 0$ for any $m \geq 1$, $\phi \in (1,2)$.

Special case

- When $m = 1$ (an AVL tree), we can calculate $\phi = \frac{\sqrt{5}-1}{2} \approx 1{,}6180$ is a root of $f(\phi) = x^2 - x - 1$.
- Parameter c can be chosen to be 2.
- Therefore,

$$h \leq \log_{\phi} \frac{n}{c} = \log_{1.618} \frac{n}{2} = \log_{\phi} e \cdot \ln \frac{n}{2} = O(\ln n).$$

Self-balancing Trees

- **New Goal:**
  - Ensure the tree stays balanced as new nodes are inserted.

And continues to be a valid BST

- Trees that maintain this balance automatically are known as self-balancing trees. There are several types, including:
  - AVL trees
  - 2-3 trees
  - B-trees
  - Red-black trees
  - …

We will focus on this one

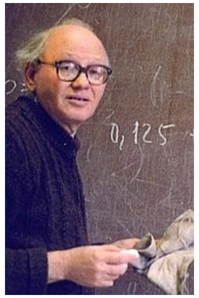
Why so many?

- Because there are multiple strategies for keeping a tree balanced after each insertion.
- Additionally, different tree types offer various useful properties depending on the application.

Outline

1. Introduction
2. Terminologies
3. Tree traversals
   – Breadth-First Search (BFS)
   – Depth-First Search (DFS)
4. Tree representation
5. Binary trees
6. Binary search trees (BST)
7. Balanced trees
8. AVL trees
   1. Definition
   2. Insertion
   3. Correctness (Height analysis)
   4. Deletion
   5. Implementation
9. 2-3, 2-3-4 trees
10. B-trees

Definition



Adelson-Velsky

The first self-balancing trees (1962)

Landis

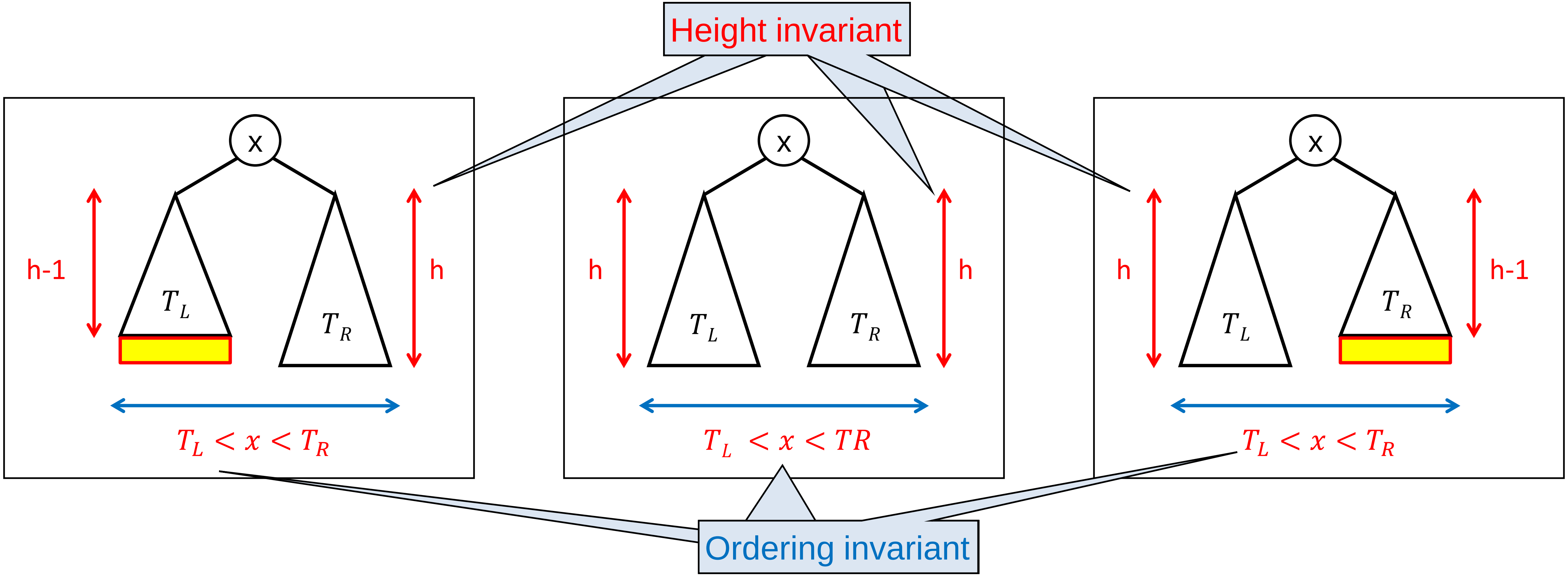
Definition 12 (An AVL tree). An AVL **tree** is a rooted tree in which it is a BST (the ordering invariant) and the absolute height difference between the left and right subtrees of any node is at most 1 (the height invariant).

- An AVL tree satisfies **two invariants:**
  - The ordering invariant.
  - The height invariant.

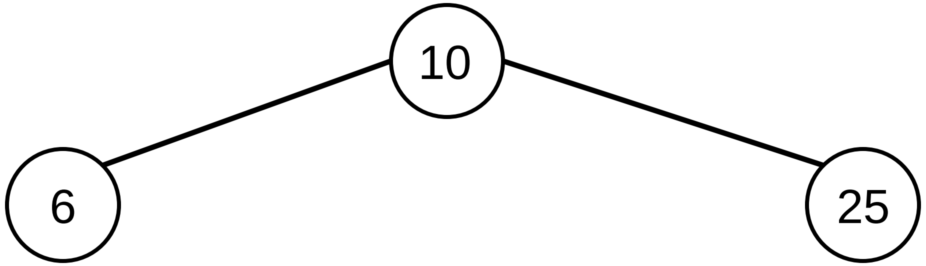
Adel'son-Velskii, Georgii Maksimovich, and Evgenii Mikhailovich Landis. An algorithm for organization of information. *Doklady Akademii Nauk*. Vol. 146. No. 2. Russian Academy of Sciences, 1962.

The Invariants of AVL Trees

- At any node, there are 3 possibilities



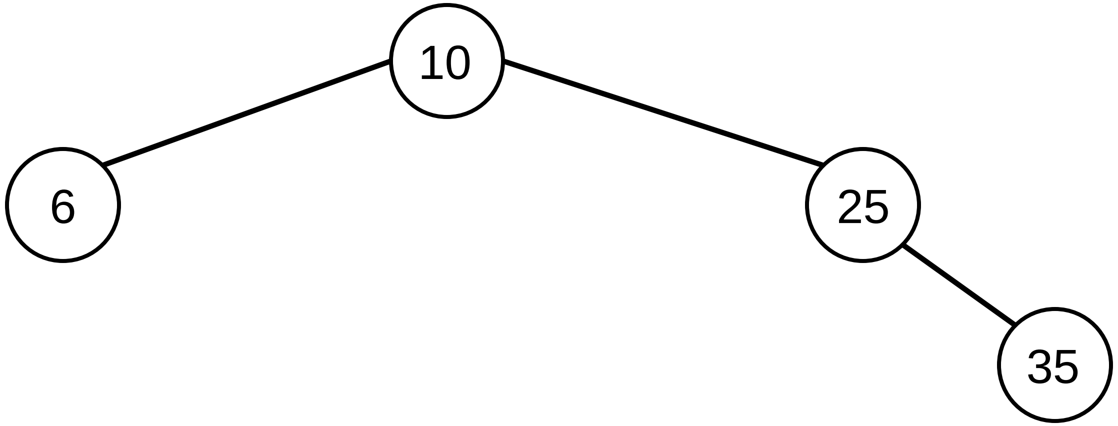
Example (1/6): Is this an AVL Tree?



- Is it ordered (sorted)?
- Is the absolute height difference of the two subtrees of every node at most by 1?

YES

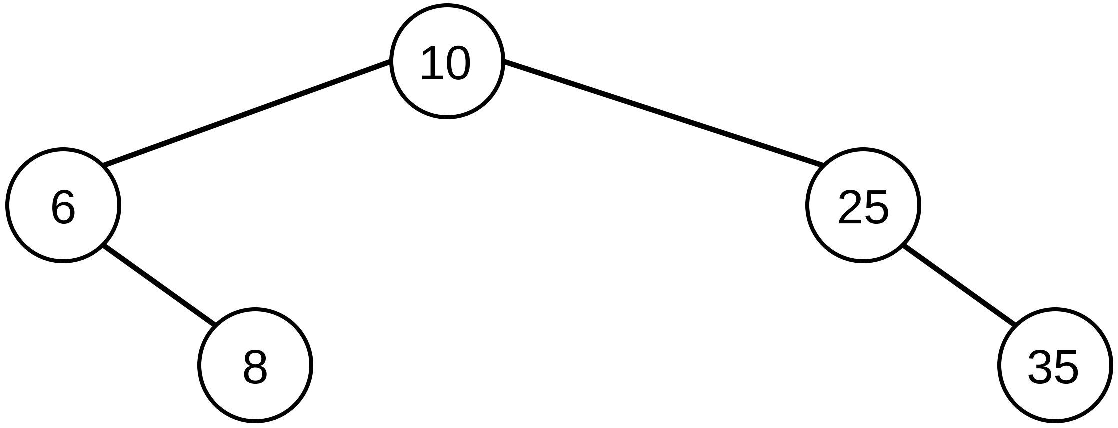
Example (2/6): Is this an AVL Tree?



- Is it ordered (sorted)? ✓
- Is the absolute height difference of the two subtrees of every node at most by 1? ✓

YES

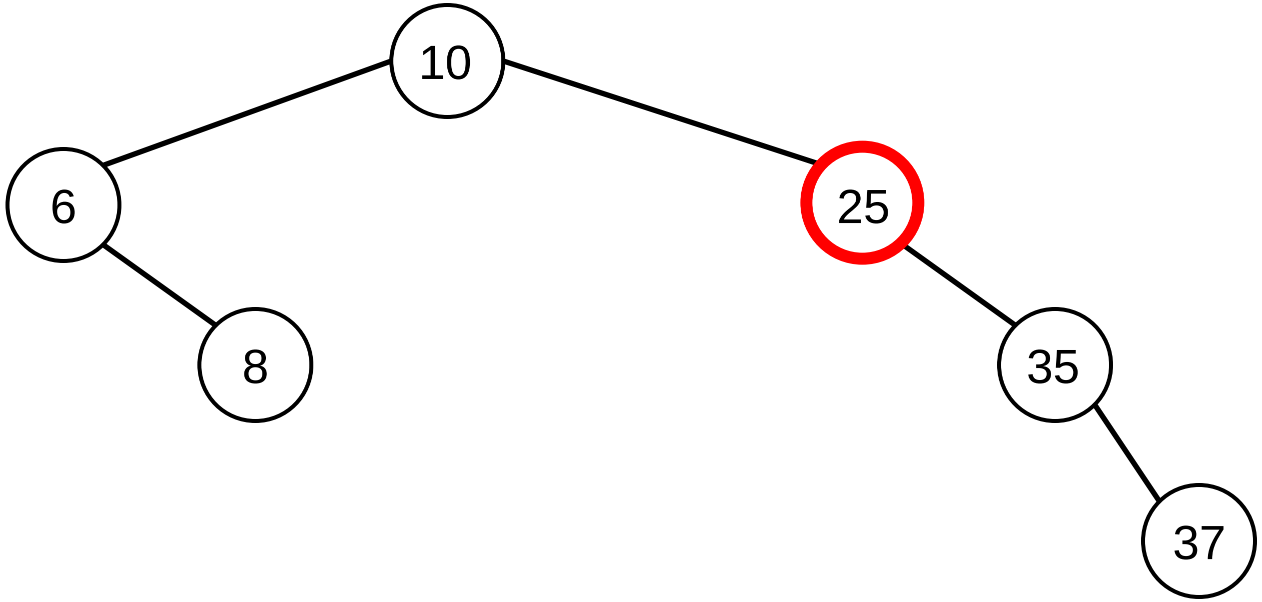
Example (3/6): Is this an AVL Tree?



- Is it ordered (sorted)? ✓
- Is the absolute height difference of the two subtrees of every node at most by 1? ✓

YES

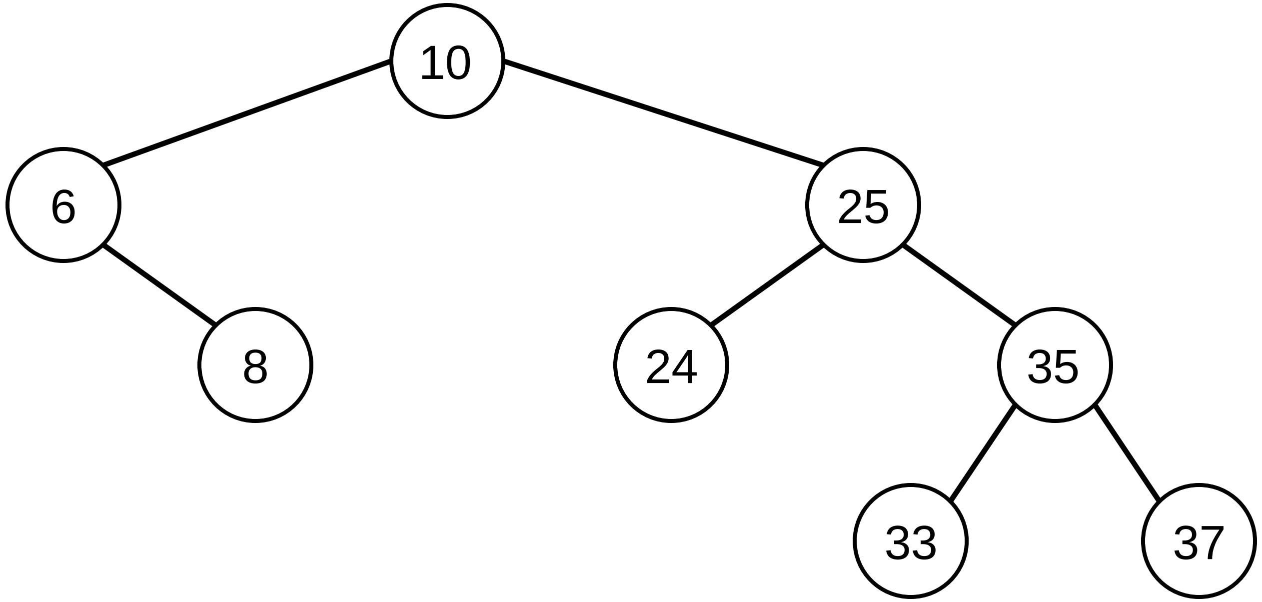
Example (4/6): Is this an AVL Tree?



- Is it ordered (sorted)? ✓
- Is the absolute height difference of the two subtrees of every node at most by 1? ✗

NO

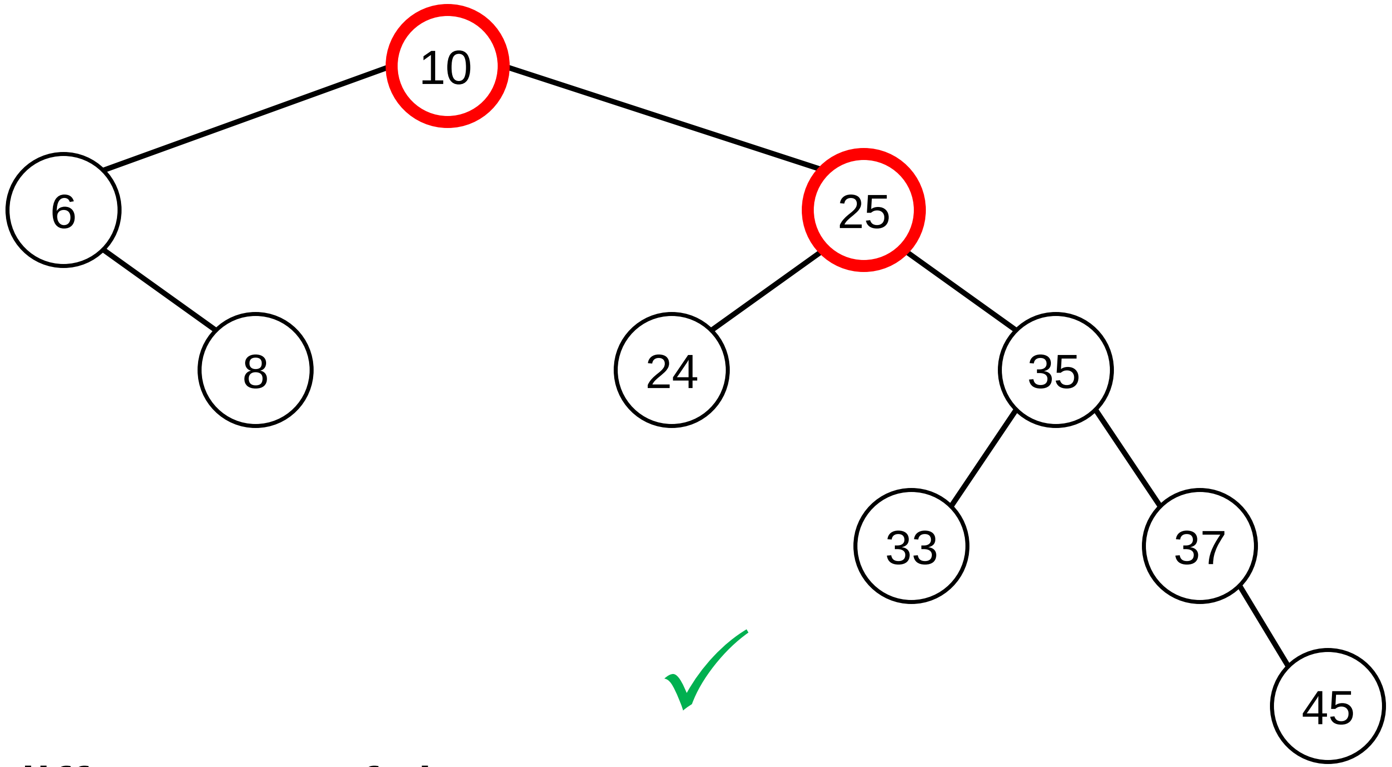
Example (5/6): Is this an AVL Tree?



- Is it ordered (sorted)? ✓
- Is the absolute height difference of the two subtrees of every node at most by 1? ✓

YES

Example (6/6): Is this an AVL Tree?



- Is it ordered (sorted)? ✓
- Is the absolute height difference of the two subtrees of every node at most by 1? ✗
  – There are violations at nodes 10 and 25

NO

Outline

1. Introduction
2. Terminologies
3. Tree traversals
   – Breadth-First Search (BFS)
   – Depth-First Search (DFS)
4. Tree representation
5. Binary trees
6. Binary search trees (BST)
7. Balanced trees
8. AVL trees
   1. Definition
   2. Insertion
   3. Correctness (Height analysis)
   4. Deletion
   5. Implementation
9. 2-3, 2-3-4 trees
10. B-trees

Insertion Strategy

1. Insert the new node as in a BST
    1. This preserves the ordering invariant
    2. But it may break the height invariant

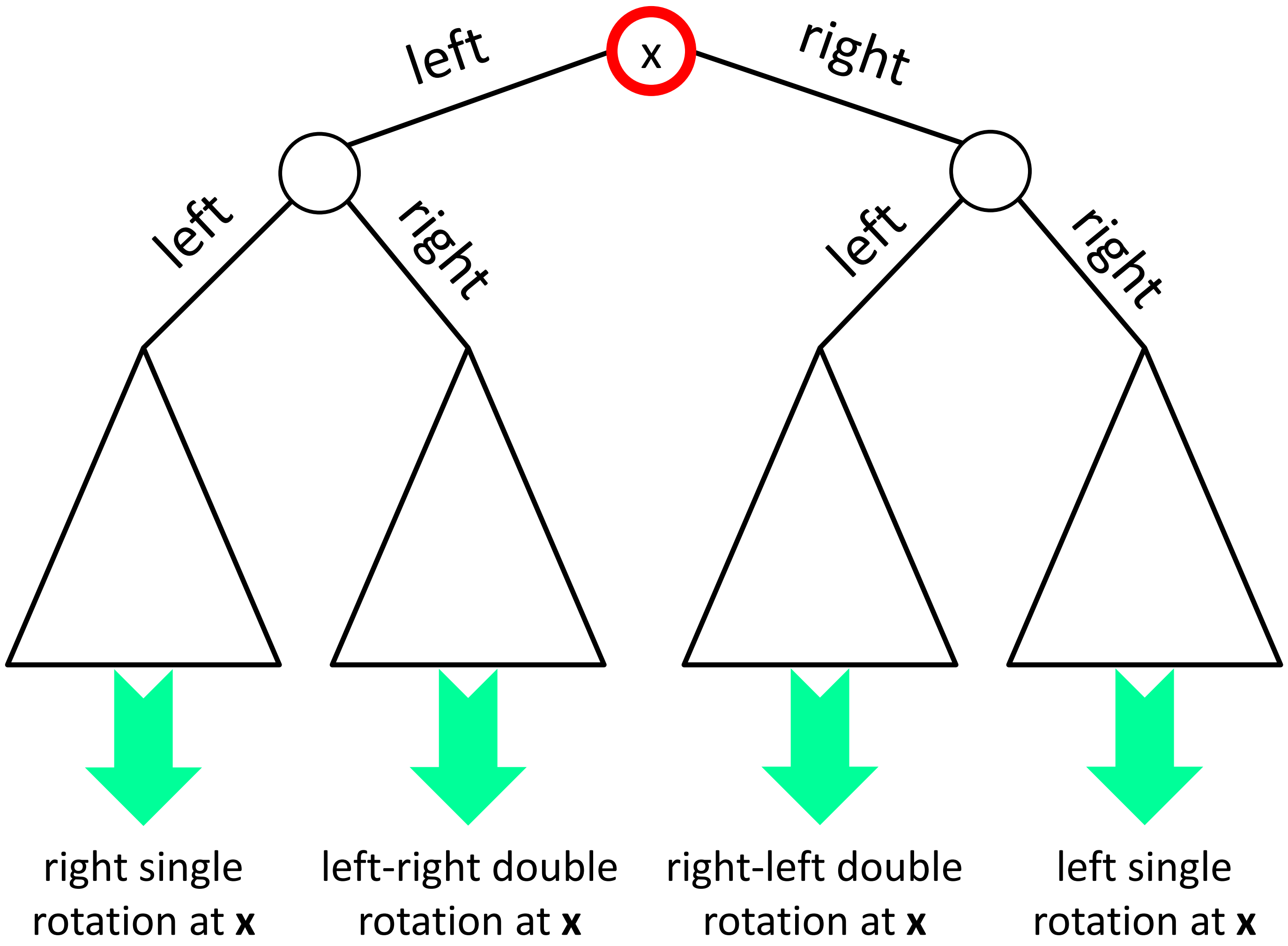
2. Fix any height invariant violation
    – Fix the LOWEST violation
        • This will take care of all other violations (Why?)

3. This is a common approach
    1. Of two invariants, **preserve one** and TEMPORARILY break the other.
    2. Then, **fix** the broken invariant.

AVL Rotation: 4 Cases to ensure the ordering and the height invariants

If the insertion
that caused the lowest
violation x happened …

… then do a …

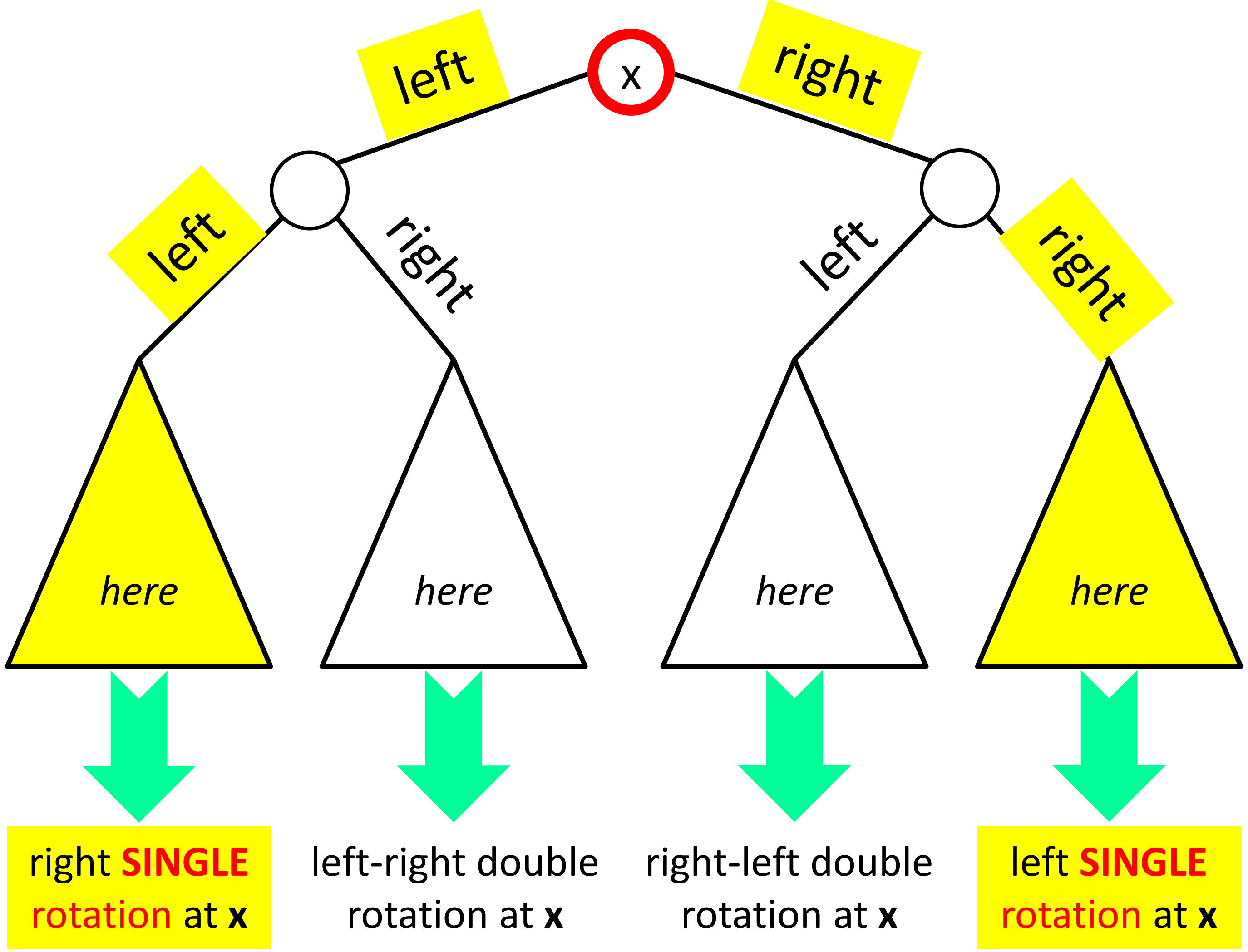


AVL Rotation: Single rotations

Rotate at the node NEXT TO the imbalanced node.

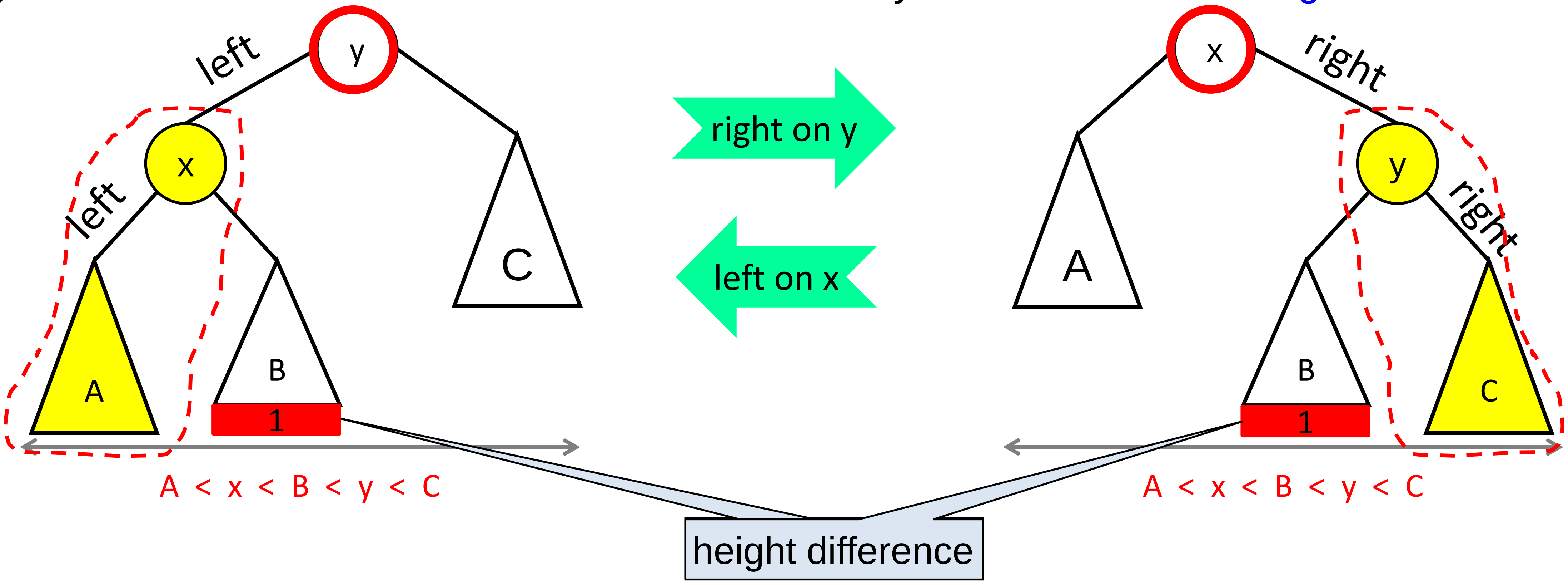
If the insertion that caused the lowest violation x happened …

… then do a …



Single Rotations Summary

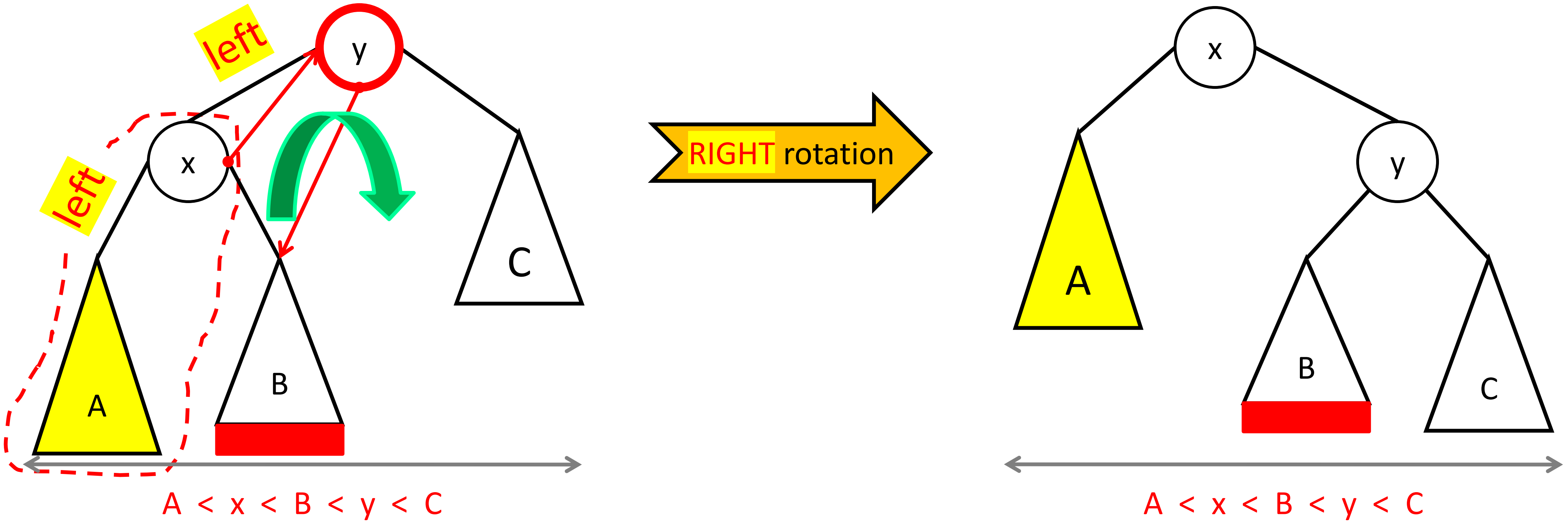
- Right and left rotations are **SINGLE rotations:** They maintain the ordering invariant



- Rotate at the node NEXT TO the imbalanced node. (1 hop)

Right Rotation

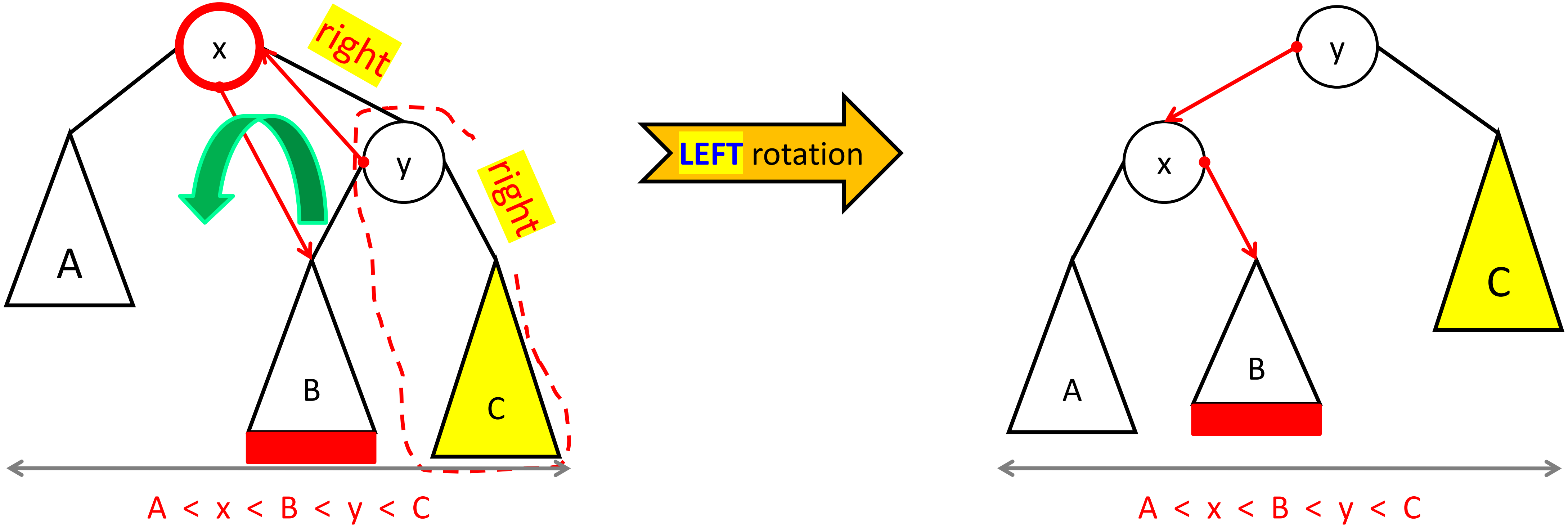
- The symmetric situation is called a **right rotation.**



- We do a right rotation when A has become too tall after an insertion

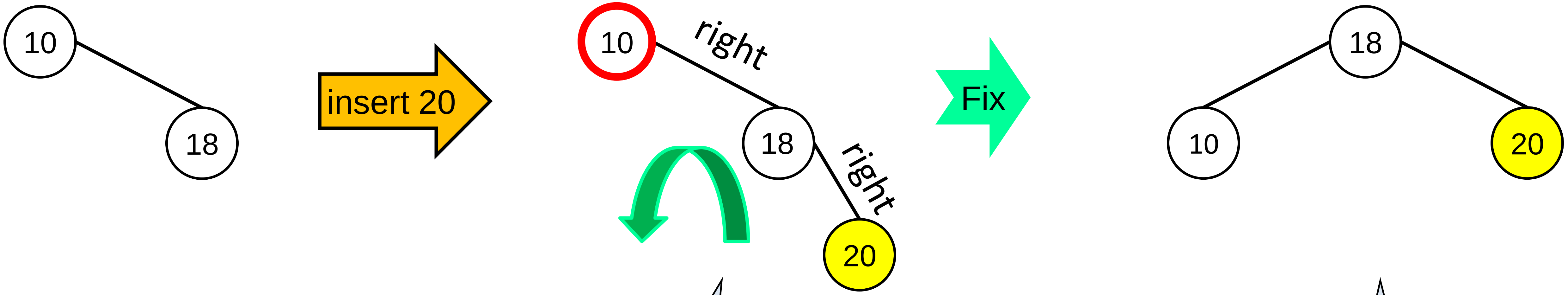
Left Rotation

- This transformation is called a **left rotation**



- We do a right rotation when C has become too tall after an insertion

Example 1

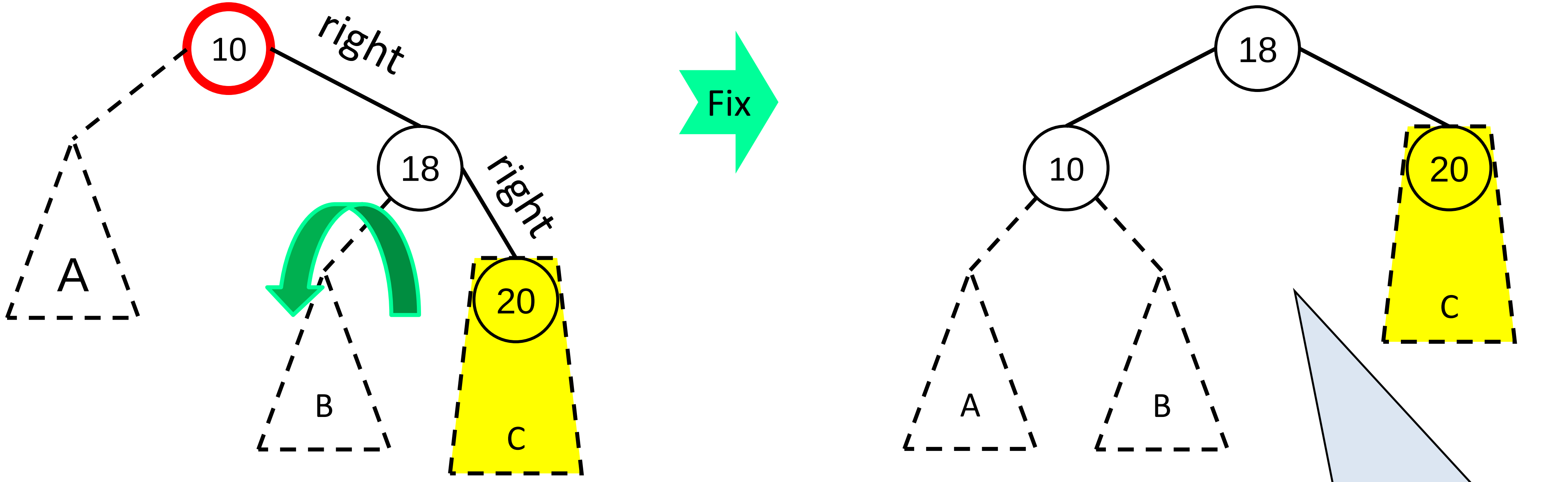


Inserting 20 as in a BST causes the height violation at node 10

This is the only tree with these elements that satisfies both the ordering and the height invariants

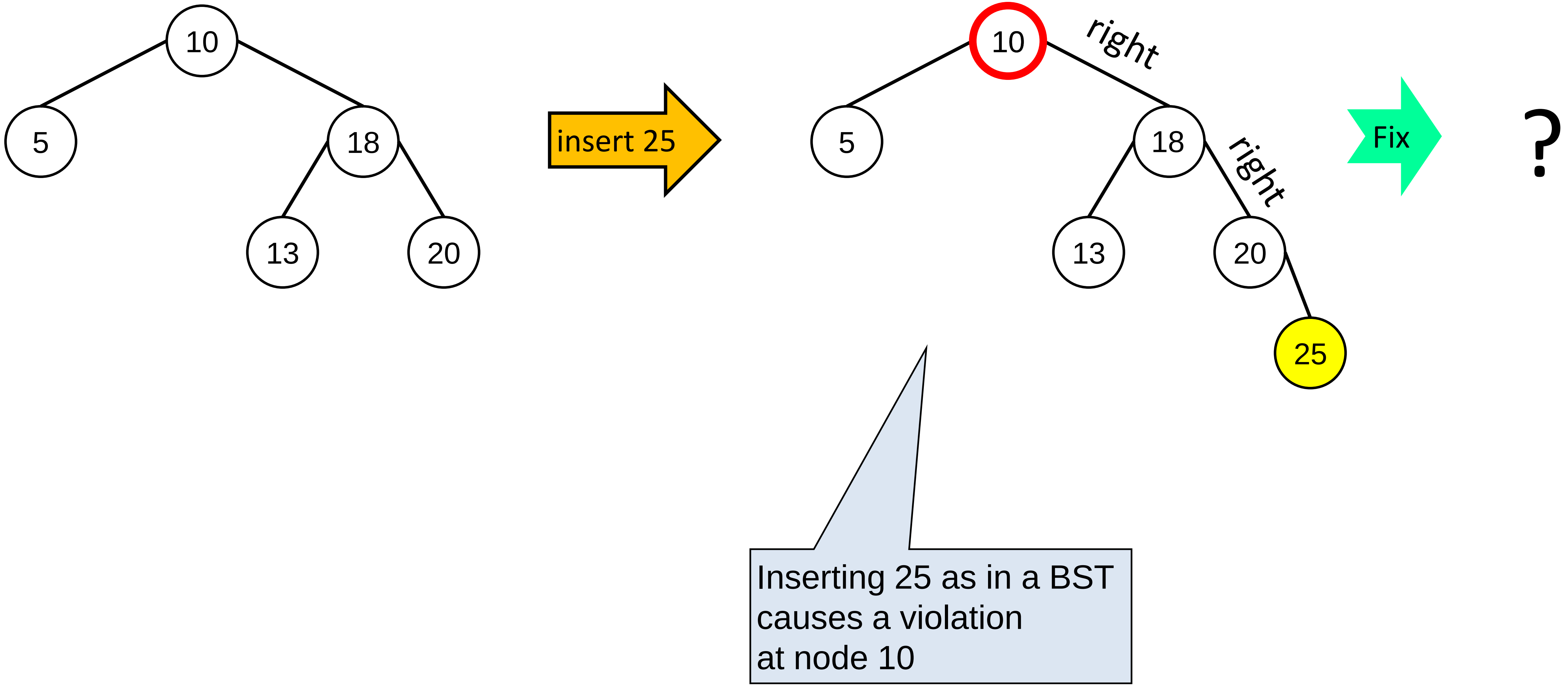
Example 1: General case

- If this example were part of a bigger tree, what would it look like?

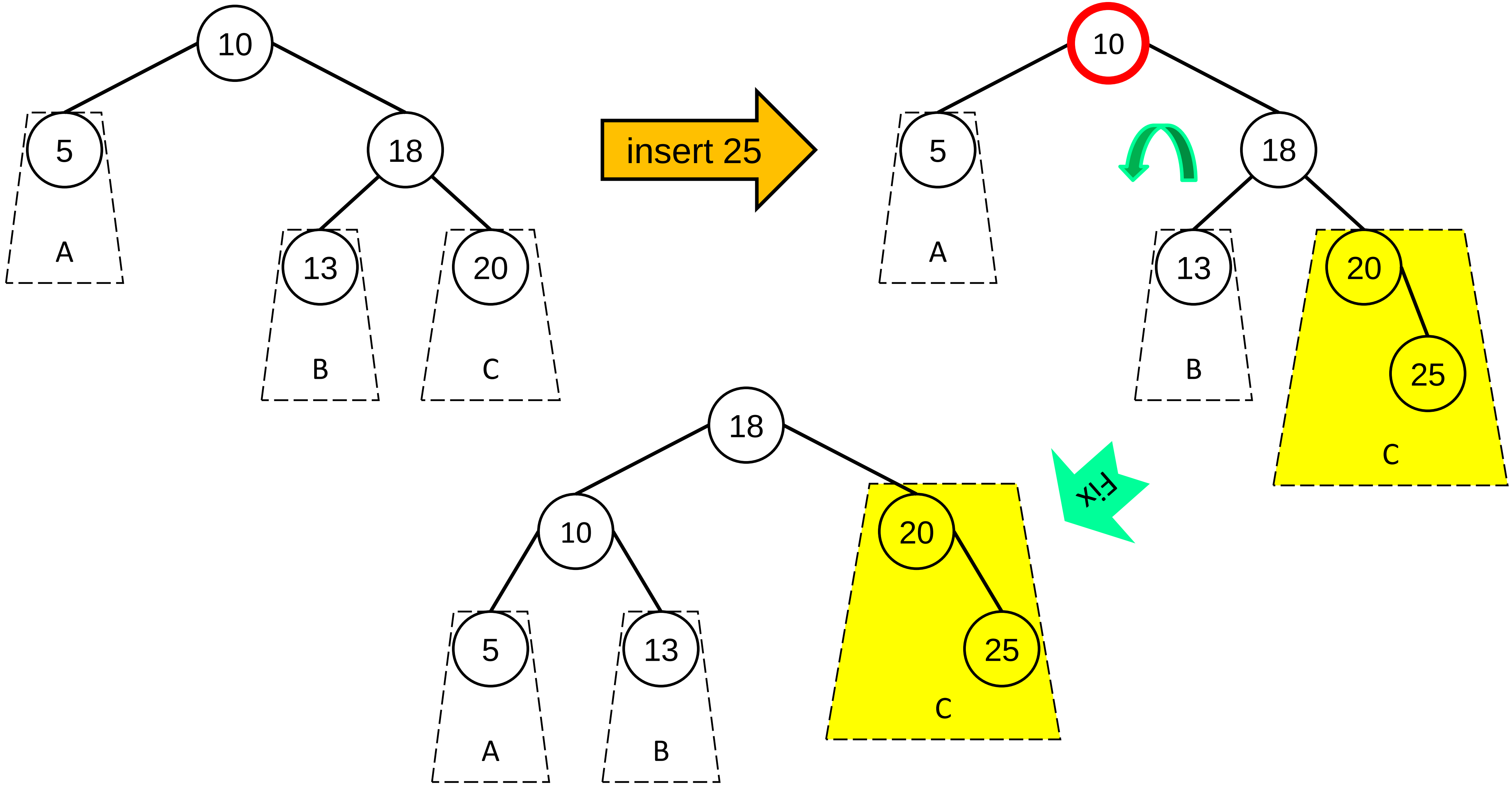


This is where the subtrees A, B and C must go to preserve the ordering invariant

Example 2 (1/2)


10
5
18
13
20
insert 25
10
right
5
18
right
13
20
25
Fix
?
Inserting 25 as in a BST
causes a violation
at node 10

Example 2 (2/2)

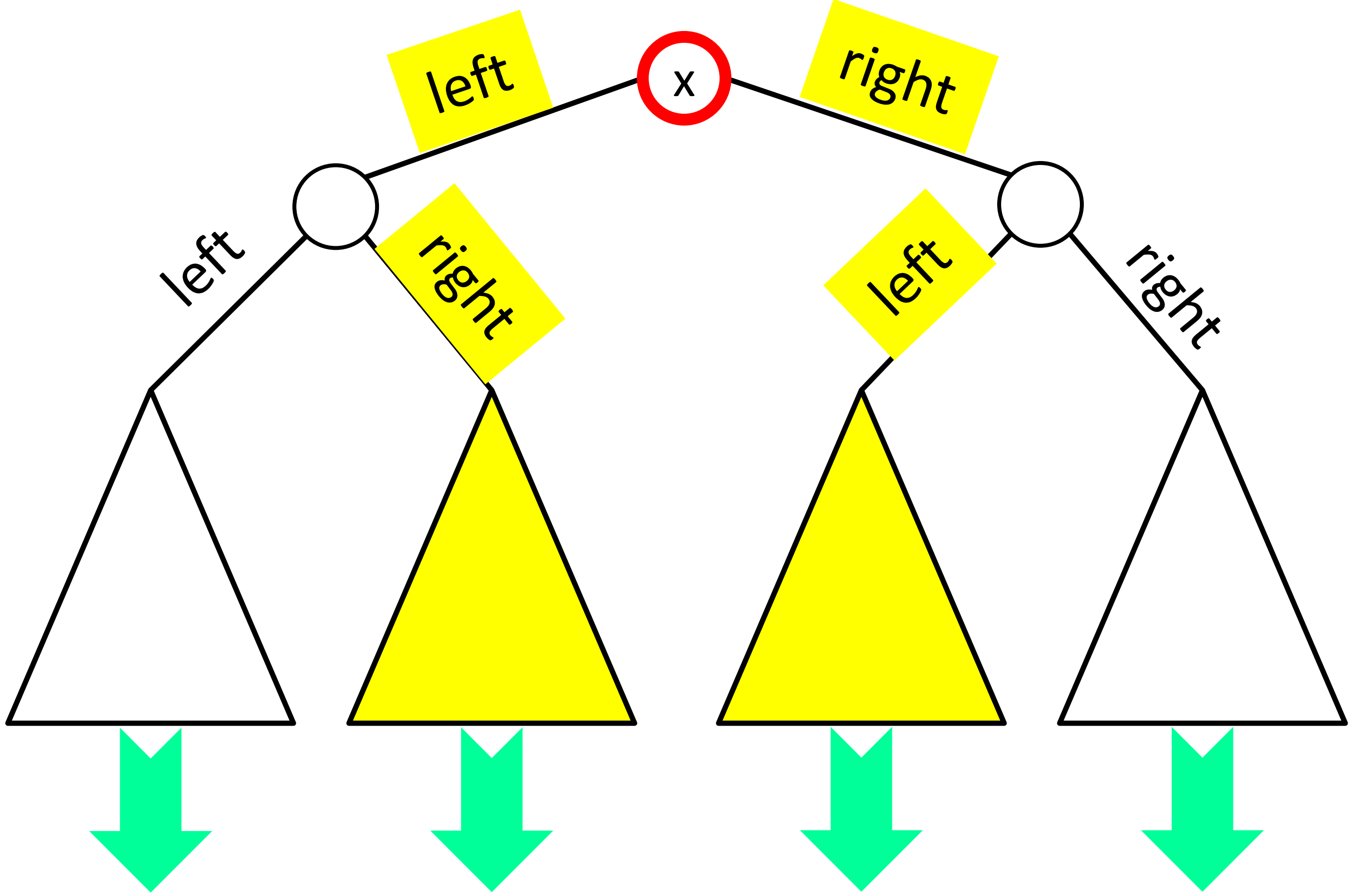

10
5
18
A
13
20
B
C
insert 25
10
5
18
A
13
20
25
B
C
Fix
18
10
20
5
13
25
A
B
C

AVL Rotation: Double rotations

Rotate at the node at the end of the path RIGHT of the imbalanced node.

(2 hops)

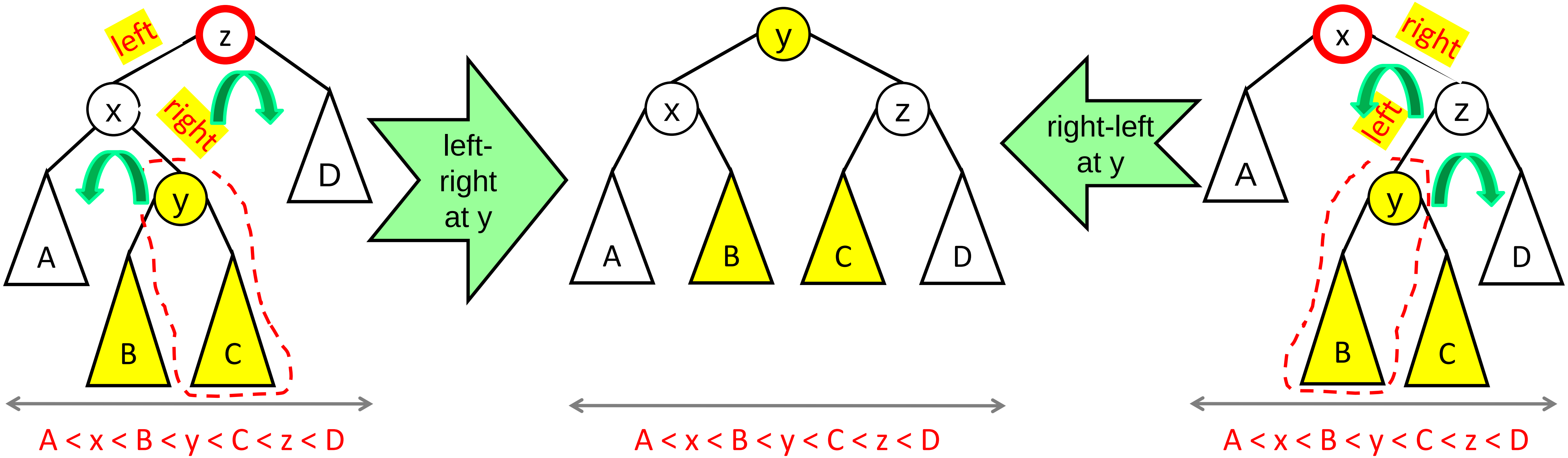
If the insertion that caused the lowest violation x happened …



… then do a …

right single rotation at **x**	left-right **DOUBLE** rotation at **x**	right-left **DOUBLE** rotation at **x**	left single rotation at **x**

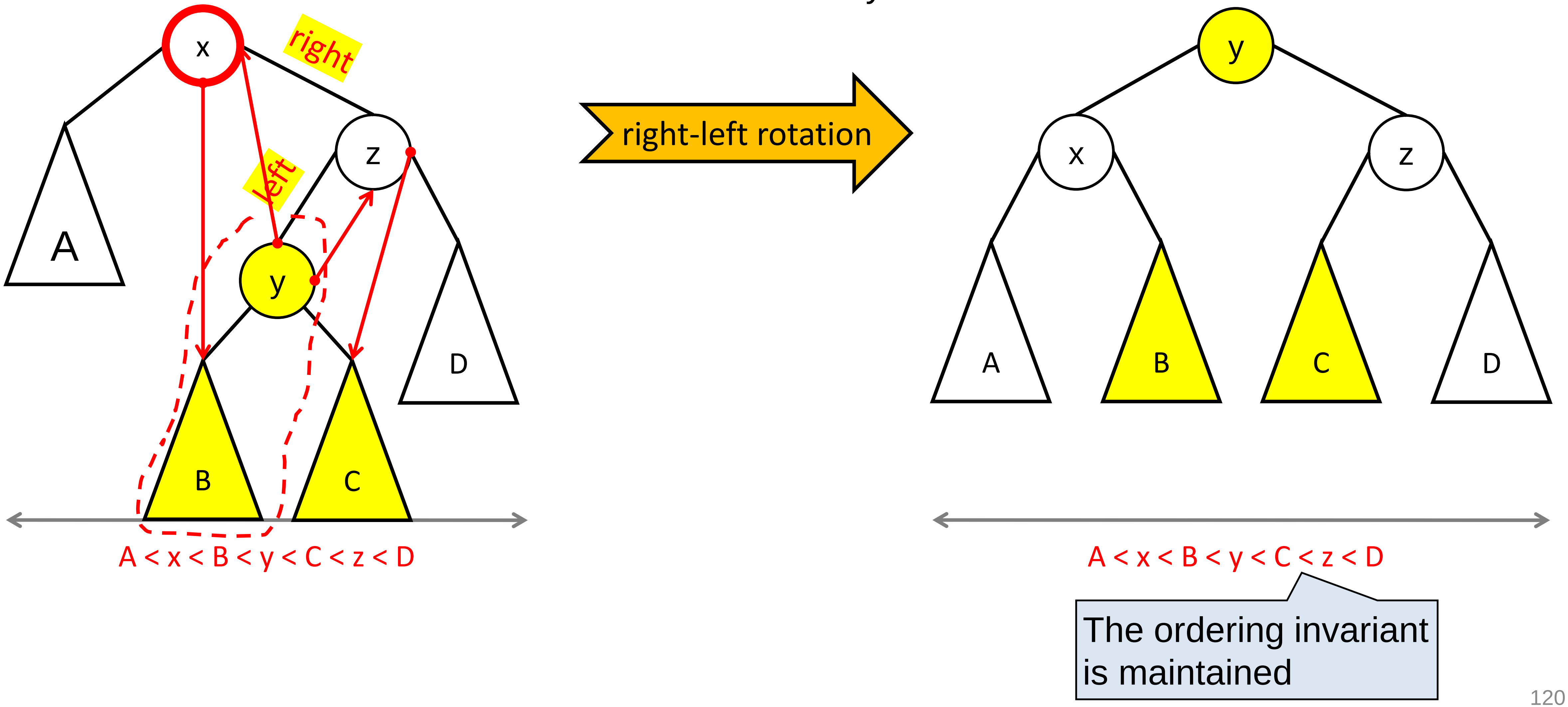
Double Rotations Summary



- Double rotations maintain the ordering invariant.
- Rotate at the node at the end of the path RIGHT of the imbalanced node. (2 hops)

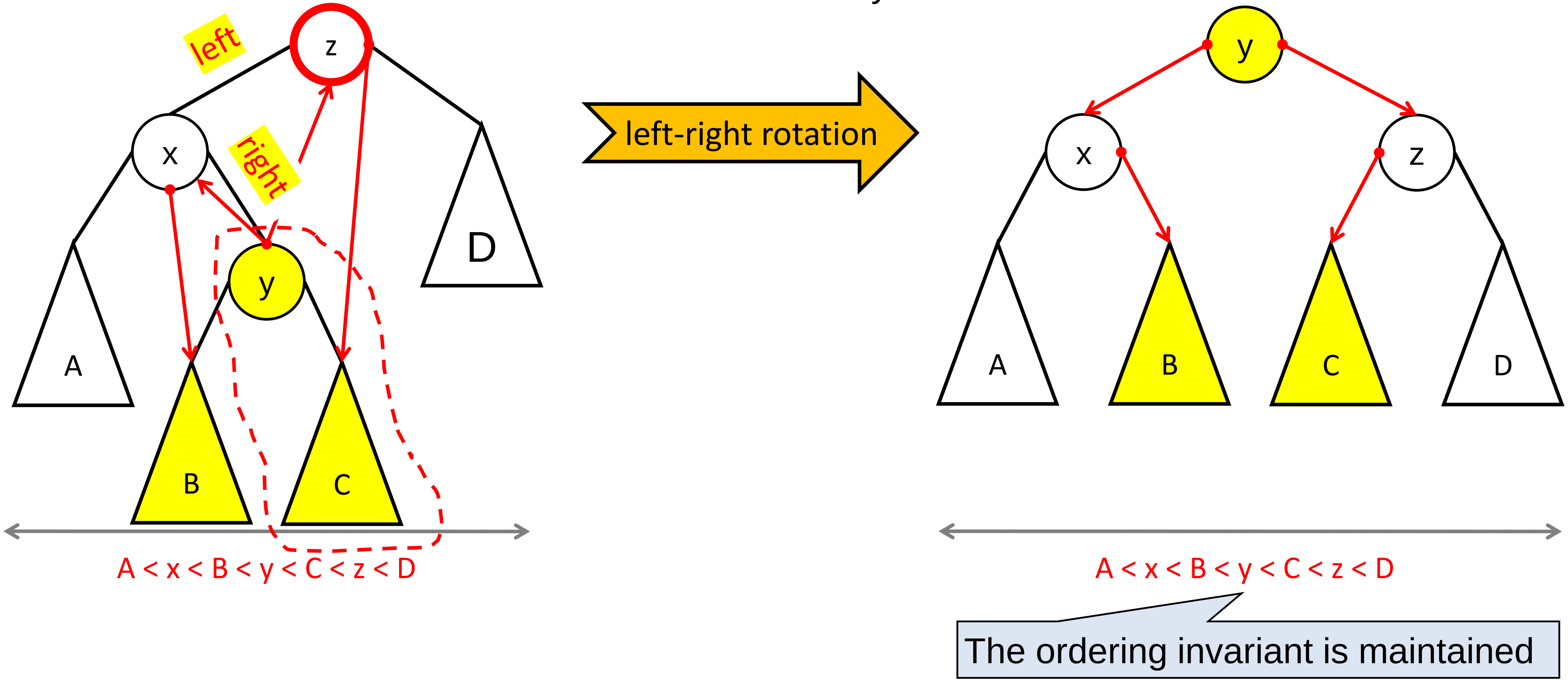
Right-left Double Rotation

- We do this double rotation when the subtree rooted at y has become too tall after an insertion

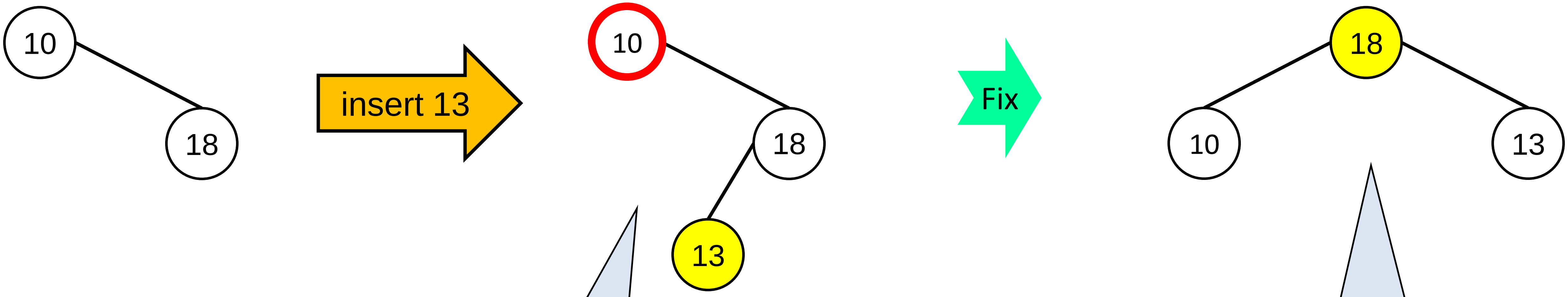


Left-right Double Rotation

- We do this double rotation when the subtree rooted at y has become too tall after an insertion



Example 3



Inserting 13 as in a BST causes a violation at node 10

This is the only tree with these elements that satisfies both the ordering and the height invariants

Self-balancing Requirements

- Does the height constraint satisfy our needs?

1. It guarantees that $h \in O(\log n)$ ✓

2. It is cheap to maintain — at most $O(\log n)$
   - Each type of rotation costs $O(1)$
   - At most one rotation is needed for each insertion (Why?)

So, maintaining the height invariant costs $O(1)$ ✓

Outline

Insertion into an AVL Tree

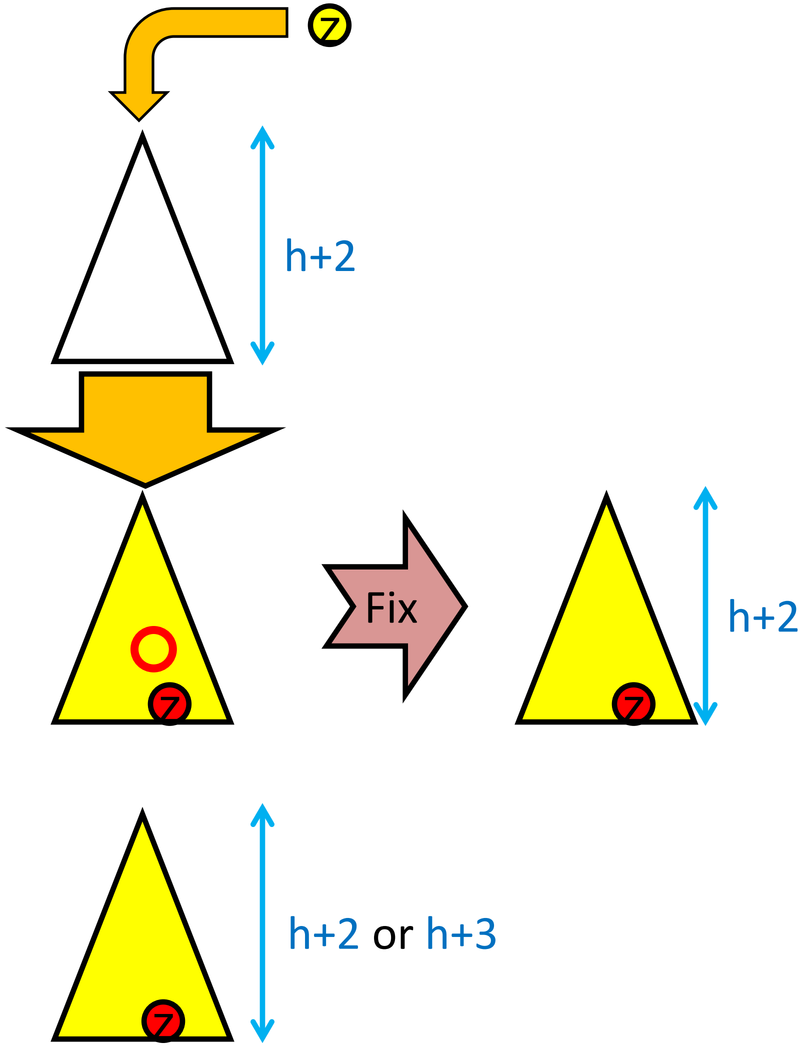
Assuming the original subtree has height $h + 2$, two outcomes are possible after inserting a new node z:

Case 1: Violation Occurs: Insertion creates a height violation.

- We perform a rotation to fix it.
- Importantly: The rotation restores the AVL balance condition.
- The subtree height remains h+2.
- 👉 So, the tree does not grow taller.
- Balance is restored locally.

Case 2: No Violation: Insertion does not cause a violation.

- Depending on structure:
  - Subtree height may become h+3 (if child subtree grows).
  - 👉 In this case, the tree may grow.
    - ➢ But since there's no violation, no rotation is needed.



Fixing the Lowest Violation

- When an **insertion causes a balance violation**, it might create multiple violations up the tree.
- But:

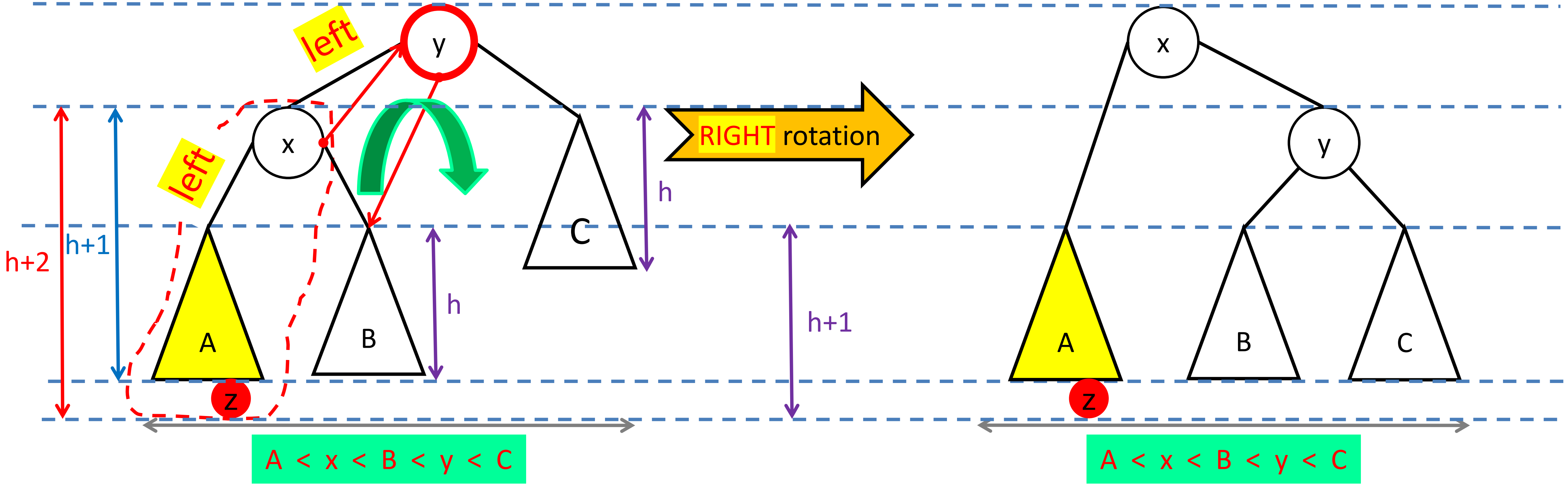
👉 **You only need to fix the lowest (deepest) one.**

- Why?
  - Fixing this violation (via rotation) restores balance locally.
  - The **height** of the modified subtree remains the same.
  - Therefore, no new violation is introduced above, and any previous violations at higher levels are automatically resolved.

Fixing the lowest violation fixes the whole tree

Right Rotation (1/2)

- This transformation is called a right rotation.



- We do a right rotation when A has become too tall after an insertion

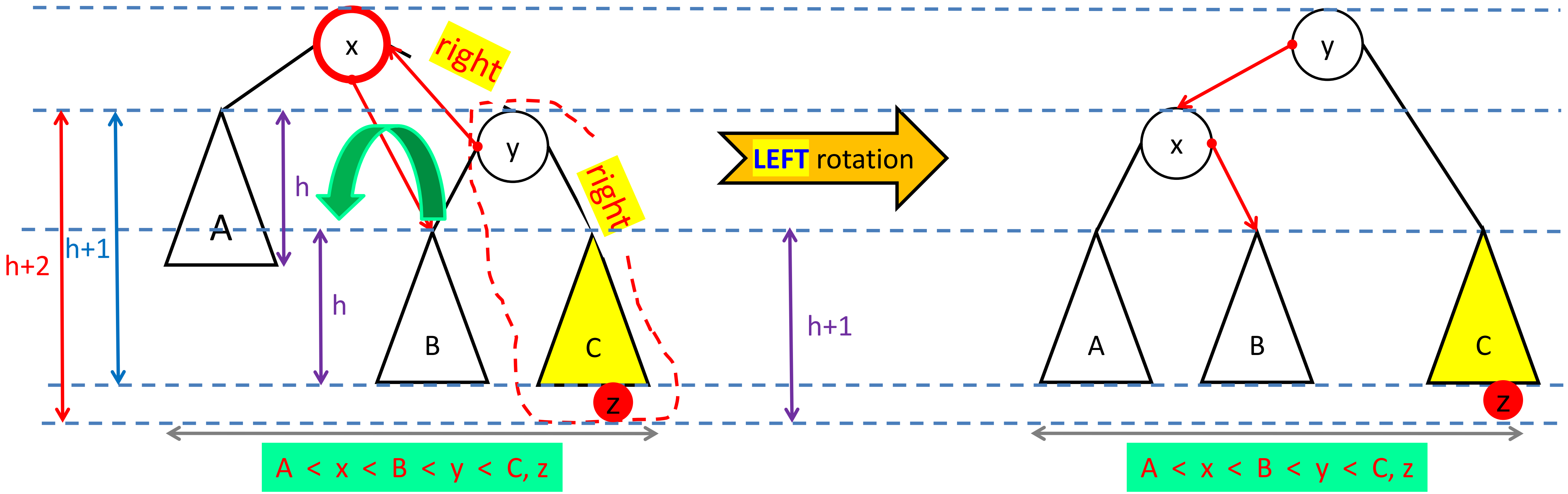
Right Rotation (2/2)

Theorem 3. Suppose node y violates the height invariant after inserting node z with the case left-left violation. Suppose the height of the subtree C is h. Then the heights of subtrees A and B must be h. Consequently, the ordering and the height invariants are restored.

- The heights of A and B must be at most h. Otherwise, y violates the height invariant before inserting node z. This contradicts the assumption that the tree satisfies the height invariant.
- The height of A must be h. Otherwise, y **DOES NOT** violate the height invariant after inserting node z.
- Then the height of B must be h. Otherwise, x violates the height invariant after inserting node z.
- Then, the right rotation makes the tree composed of z, A, x, y, B, and C satisfy the height and the ordering invariant.

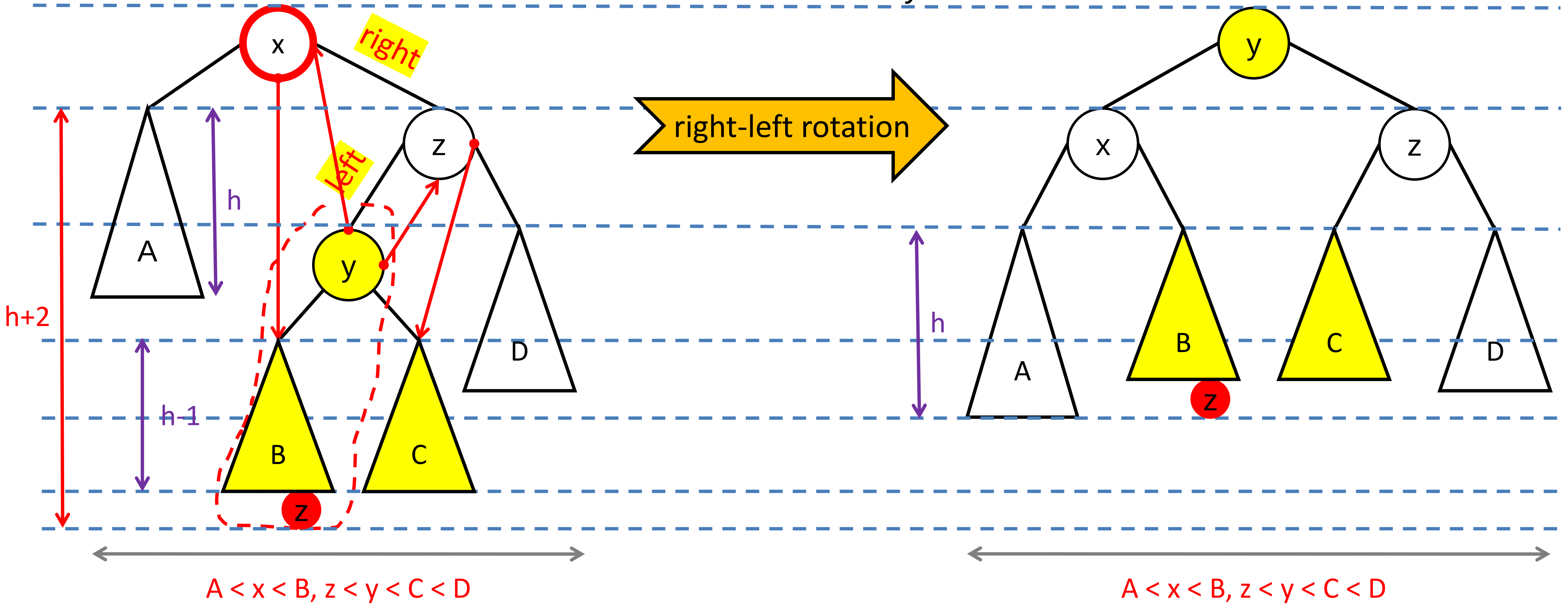
Left Rotation

Theorem 4. Suppose node x violates the height invariant after inserting node z with the case right-right violation. Suppose the height of the subtree A is h. Then the heights of subtrees B and C must be h. Consequently, the ordering and the height invariants are restored.



Right-left Double Rotation (1/3)

- We do this double rotation when the subtree rooted at y has become too tall after an insertion



Right-left Double Rotation (2/3)

Theorem 5. Suppose node x violates the height invariant after inserting node z with the case right-left violation. Suppose the height of the subtree A is h. Then the height of subtrees B and C must be $h-1$ and the height of D must be either h or $h-1$. Consequently, the ordering and the height invariants are restored.

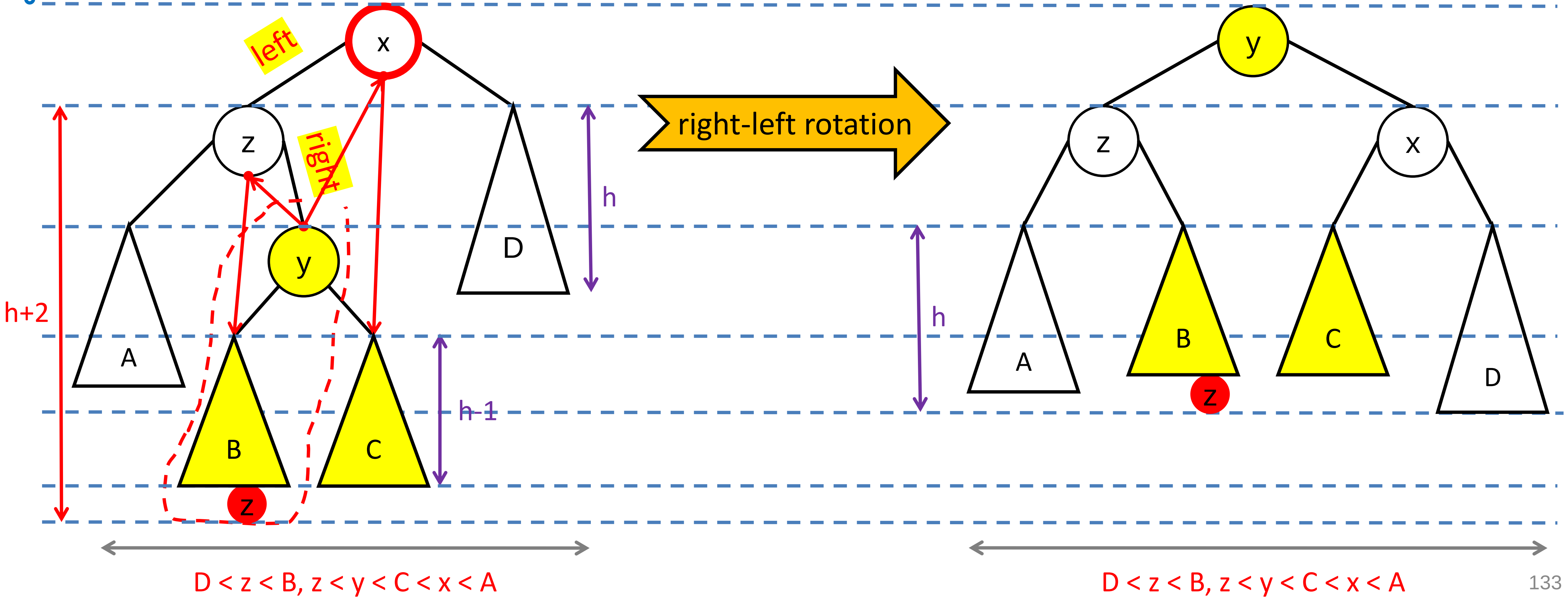
- The heights of B and C must be the same. Indeed, assume that the heights of B and C are NOT THE SAME. WLOG, let B be taller than C.
- Then the node z must be inserted into B. Otherwise, the heights of B and the new subtree formed by C and z are the same. Therefore, it is impossible to make node x violate the height invariant.
- However, in that case, the height of the new tree formed by B and node z is taller than C by 2.
- This implies y is the LOWEST node that violates the height invariant.
- This contradicts the assumption that x is the lowest node.
- Therefore, the heights of B and C must be the same.

Right-left Double Rotation (3/3)

- Since the heights of B and C are the same, WOLG, let z be inserted into B.
- Because there is a height violation at node x, the height of B must be $h-1$.
  - The height cannot be smaller than $h-1$ because x wouldn't violate the height invariant **after** inserting z.
  - The height cannot be larger than $h-1$ because x would violate the height invariant **before** inserting z.
  - (Note that the height of D is either h or $h-1$ to make an AVL tree.)
- Therefore, after the right-left rotation as depicted in the figure, the height difference between every node in the new tree must be at most 1.

Left-right Double Rotation

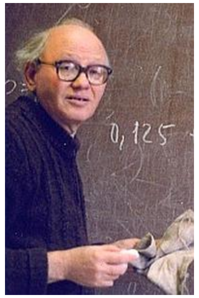
Theorem 6. Suppose node x violates the height invariant after inserting node z with the case left-right violation. Suppose the height of the subtree D is h. Then the height of subtrees B and C must be $h-1$ and the height of A must be either h or $h-1$. Consequently, the ordering and the height invariants are restored.



Outline

1. Introduction
2. Terminologies
3. Tree traversals
   – Breadth-First Search (BFS)
   – Depth-First Search (DFS)
4. Tree representation
5. Binary trees
6. Binary search trees (BST)
7. Balanced trees
8. AVL trees
   1. Definition
   2. Insertion
   3. Correctness (Height analysis)
   4. Deletion
   5. Implementation
9. 2-3, 2-3-4 trees
10. B-trees

Definition



Adelson-Velsky

The first self-balancing trees (1962)

Landis

Definition 13 (An AVL tree). An AVL **tree** is a rooted tree in which it is a BST (the ordering invariant) and the absolute height difference between the left and right subtrees of any node is at most 1 (the height invariant).

- An AVL tree satisfies **two invariants:**
  - The ordering invariant. ✓ — mostly served for deletion
  - The height invariant. ✓ — mostly served for insertion

Adel'son-Velskii, Georgii Maksimovich, and Evgenii Mikhailovich Landis. An algorithm for organization of information. *Doklady Akademii Nauk*. Vol. 146. No. 2. Russian Academy of Sciences, 1962.

General ideas

1
- **Perform** standard BST deletion because an AVL is a BST

2
- **Update** the heights of ancestor nodes

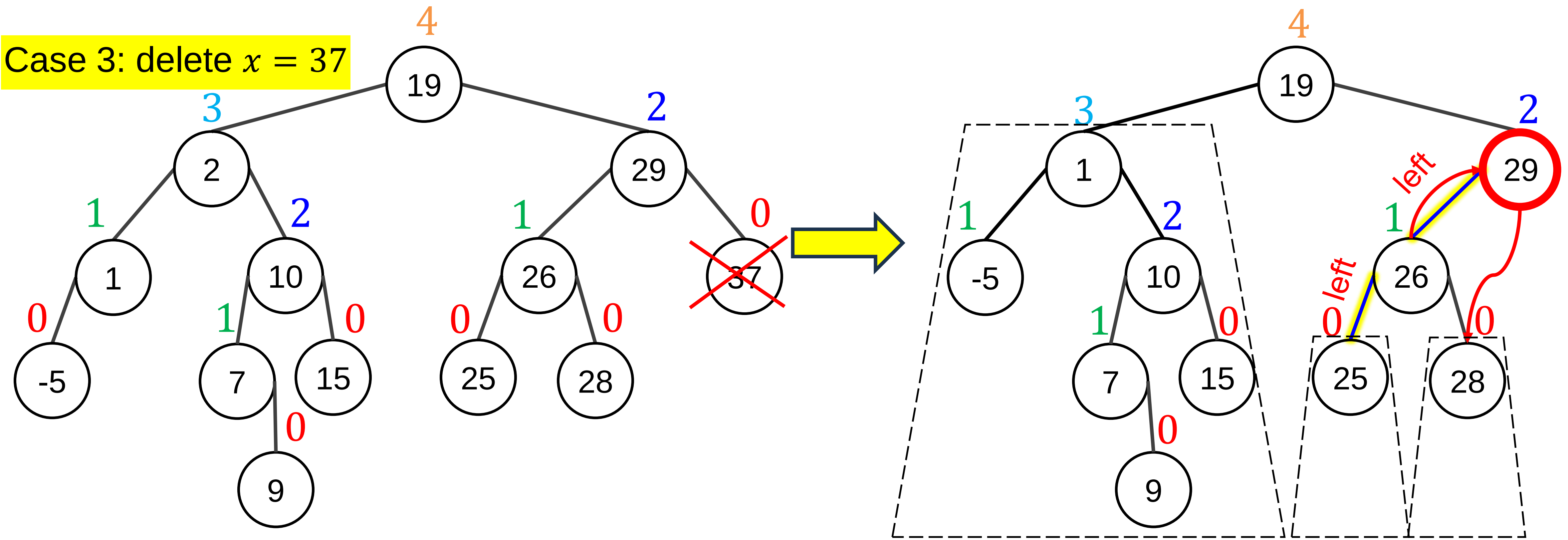
3
- **Rebalance** the new tree by rebalancing the node at the deleted position and its ancestor nodes

There are 3 possible cases when deleting node X:

1. X is a leaf node.
2. X has only one child (left or right).
3. X has both left and right children.

Example 1 (1/2)

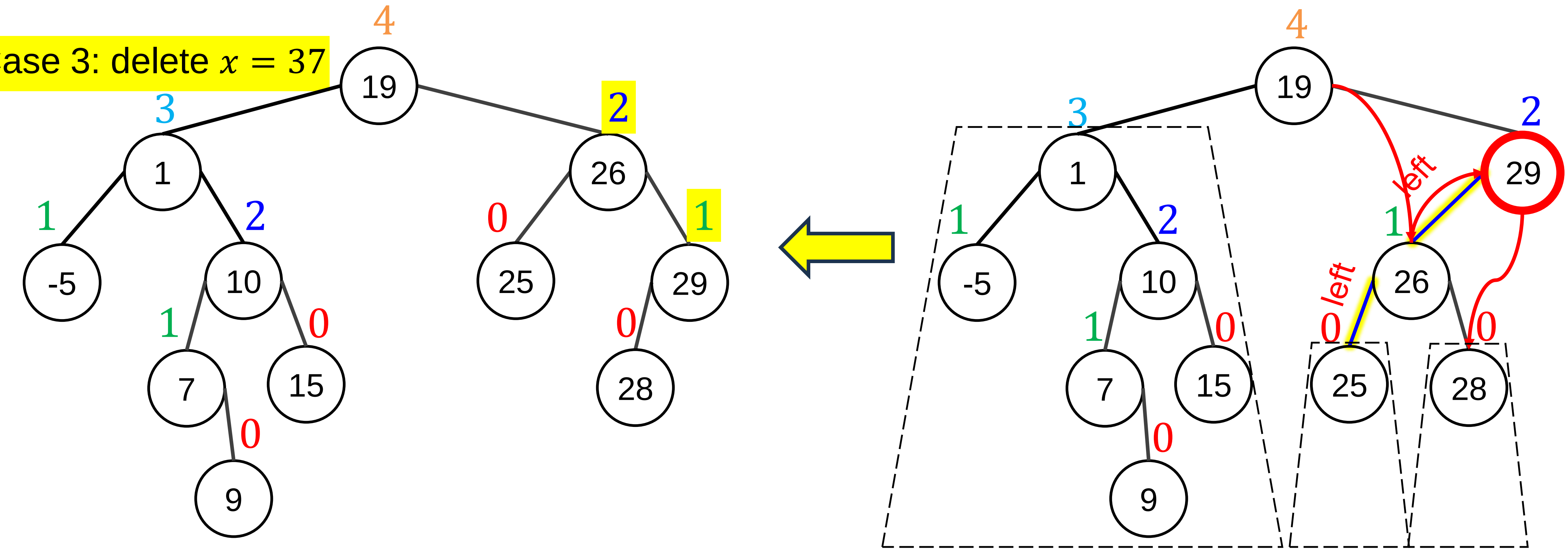
- When deleting the element $x = 37$ from the tree, we simply delete it because it is a leaf.
- The height of each node is beside it.



Example 1 (2/2)

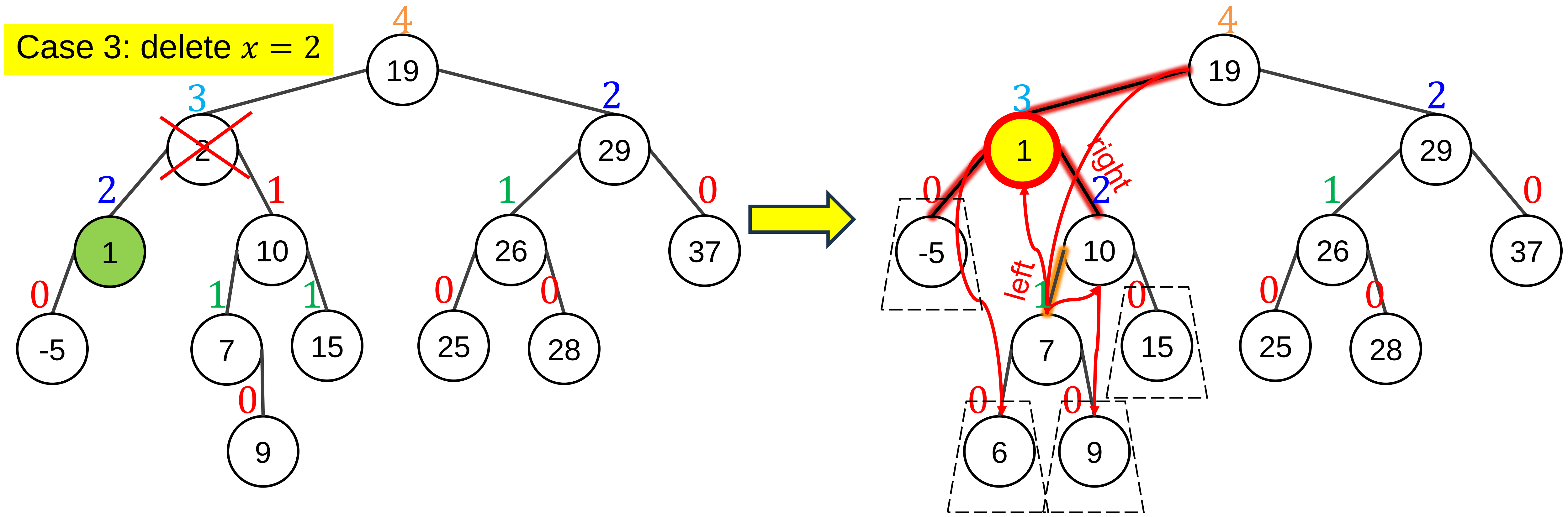
- It causes a left-left imbalance.

Case 3: delete $x = 37$



Example 2 (1/2)

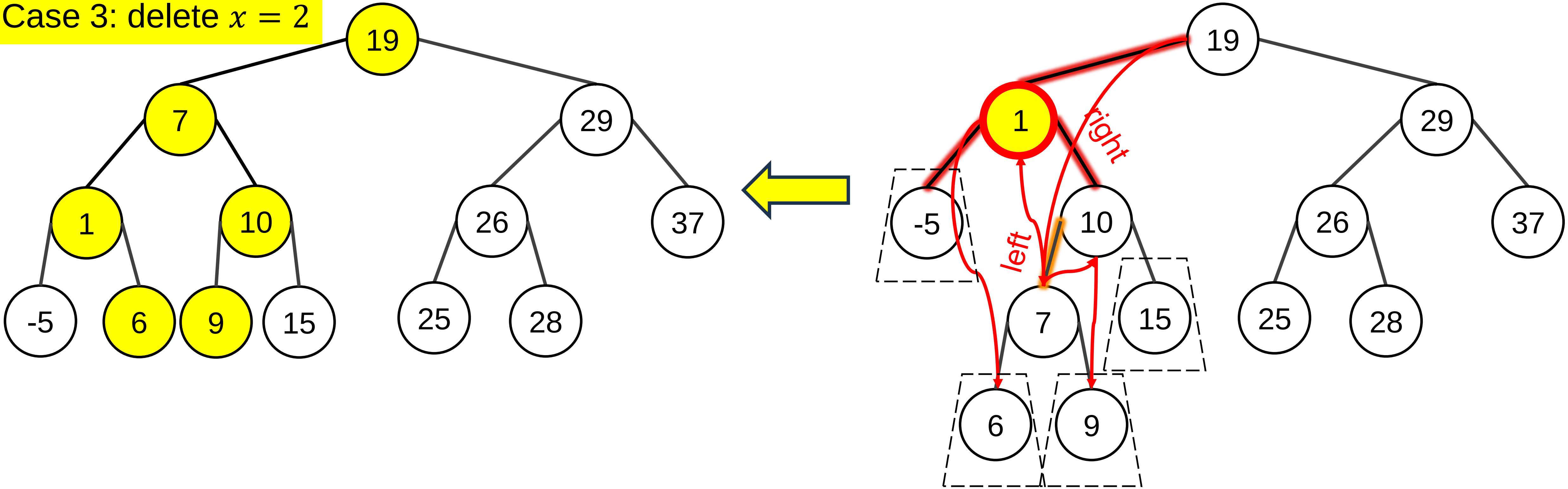
- When deleting the element $x = 2$ from the tree, the element 1 (the rightmost element in the left subtree of node 2) is the replacement element.



Example 2 (2/2)

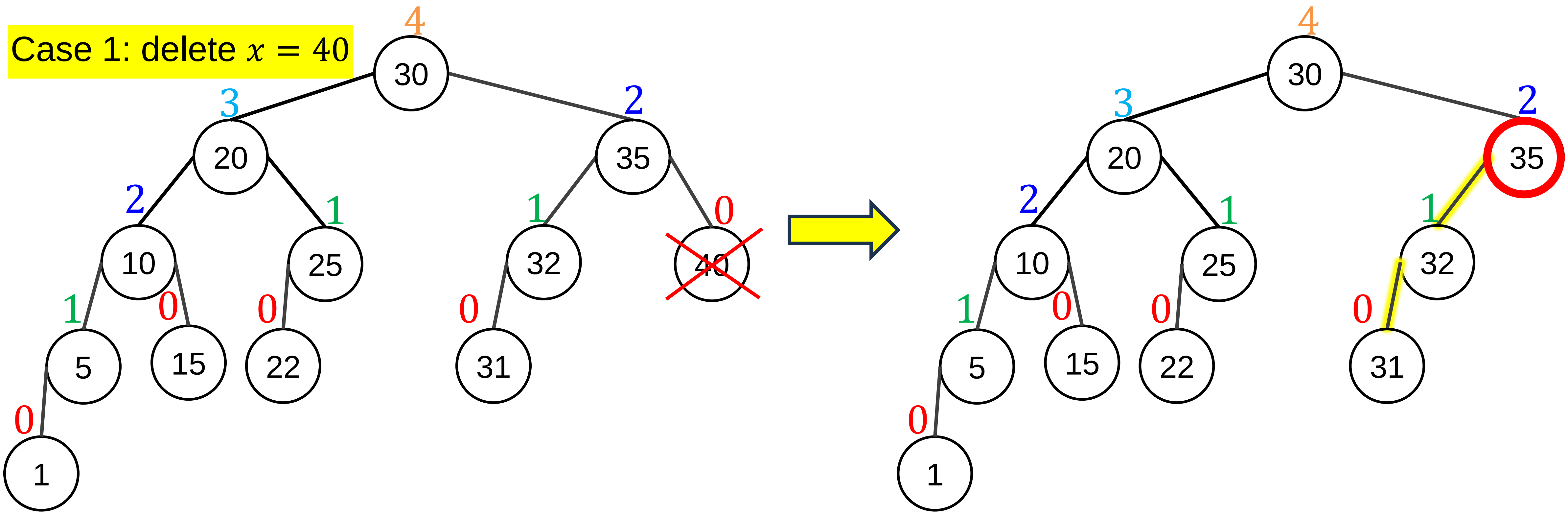
- When deleting the element $x = 2$ from the tree, the element 1 (the rightmost element in the left subtree of node 2) is the replacement element.

Case 3: delete $x = 2$



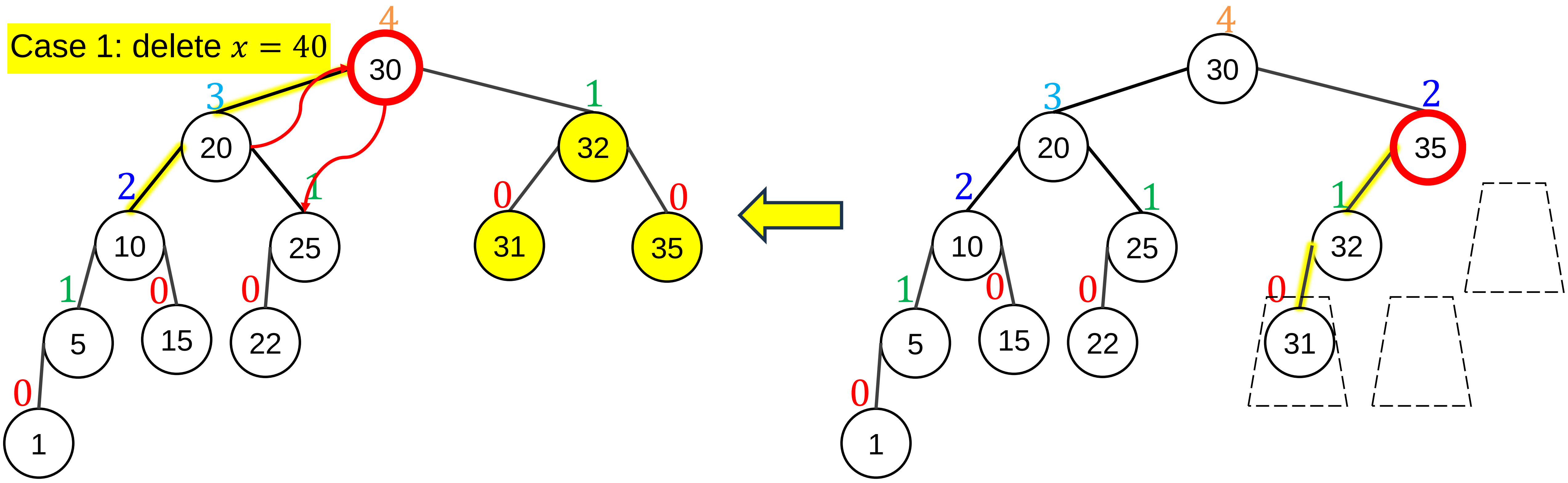
Example 3 (1/3)

- When deleting the element $x = 40$ from the tree, we simply delete it because it is a leaf.
- Node 35 causes a left-left imbalance.



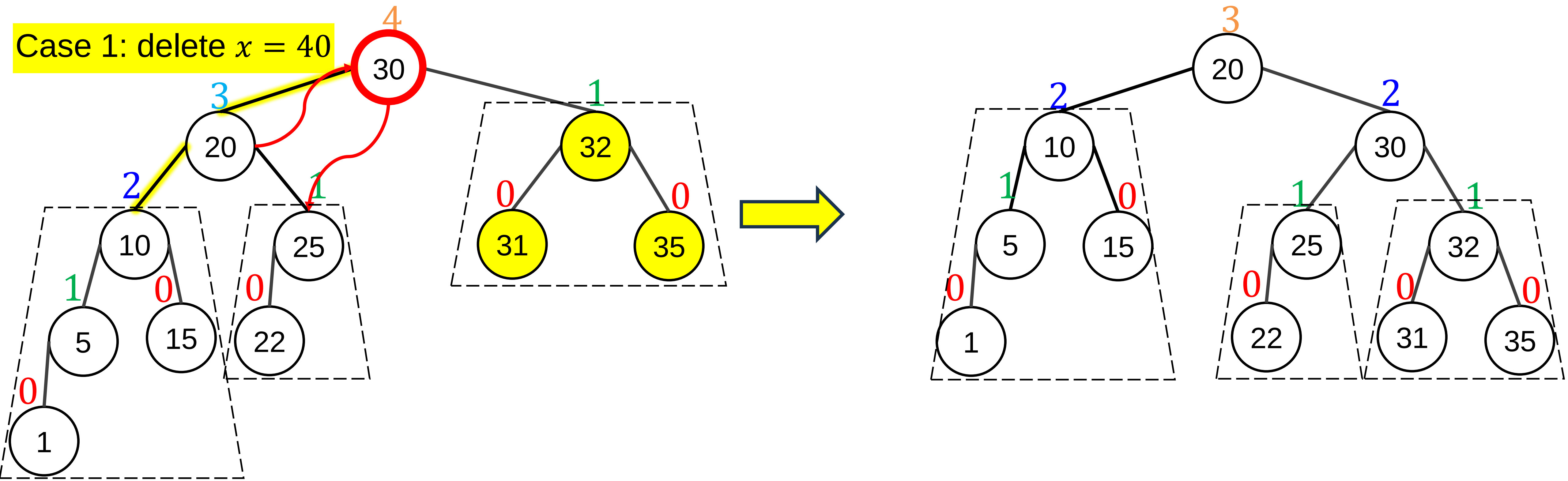
Example 3 (2/3)

- Rebalancing node 35 causes node 30 to be imbalanced in the case left-left imbalance.



Example 3 (3/3)

- Rebalancing node 35 causes node 30 to be imbalanced in the case of left-left imbalance.



Outline

1. Introduction
2. Terminologies
3. Tree traversals
   - Breadth-First Search (BFS)
   - Depth-First Search (DFS)
4. Tree representation
5. Binary trees
6. Binary search trees (BST)
7. Balanced trees
8. AVL trees
   1. Definition
   2. Insertion
   3. Correctness (Height analysis)
   4. Deletion
   5. Implementation
9. 2-3, 2-3-4 trees
10. B-trees

Initialization

```
#define LH -1  // Left High
#define EH  0  // Equal Height
#define RH  1  // Right High

typedef int Data;

typedef struct tagAVLNode {
    char balFactor;
    Data key;
    struct tagAVLNode* pLeft;
    struct tagAVLNode* pRight;
} AVLNode;

typedef AVLNode* AVLTree;
```

```
// Create a new node
AVLTree createNode(Data key) {
    AVLTree node = new AVLNode;
    node->key = key;
    node->balFactor = EH;
    node->pLeft = node->pRight = nullptr;
    return node;
}
```

Right and left rotations

```
// Right rotation
void rotateRight(AVLTree& root) {
    AVLTree leftChild = root->pLeft;
    root->pLeft = leftChild->pRight;
    leftChild->pRight = root;
    root = leftChild;
}
```

```
// Left rotation
void rotateLeft(AVLTree& root) {
    AVLTree rightChild = root->pRight;
    root->pRight = rightChild->pLeft;
    rightChild->pLeft = root;
    root = rightChild;
}
```

Left Balance

```
void leftBalance(AVLTree& root) {
    AVLTree leftChild = root->pLeft;
    if (leftChild->balFactor == LH) {
        root->balFactor = EH;
        leftChild->balFactor = EH;
        rotateRight(root);
    } else {
        AVLTree rightGrandChild =
leftChild->pRight;
        switch (rightGrandChild->balFactor)
{
            case LH:
                root->balFactor = RH;
                leftChild->balFactor = EH;
                break;
            case EH:
                root->balFactor = EH;
                leftChild->balFactor = EH;
                break;
            case RH:
                root->balFactor = EH;
                leftChild->balFactor = LH;
                break;
        }
        rightGrandChild->balFactor = EH;
        rotateLeft(root->pLeft);
        rotateRight(root);
    }
}
```

Right Balance

```
void rightBalance(AVLTree& root) {
    AVLTree rightChild = root->pRight;
    if (rightChild->balFactor == RH) {
        root->balFactor = EH;
        rightChild->balFactor = EH;
        rotateLeft(root);
    } else {
        AVLTree leftGrandChild =
rightChild->pLeft;
        switch (leftGrandChild->balFactor)
{
            case RH:
                root->balFactor = LH;
                rightChild->balFactor = EH;
                break;
            case EH:
                root->balFactor = EH;
                rightChild->balFactor = EH;
                break;
            case LH:
                root->balFactor = EH;
                rightChild->balFactor = RH;
                break;
        }
        leftGrandChild->balFactor = EH;
        rotateRight(root->pRight);
        rotateLeft(root);
    }
}
```

Recursive insertion

```
bool insertAVL(AVLTree& root, Data key, bool& taller) {
    if (!root) {
        root = createNode(key);
        taller = true;
        return true;
    }
    if (key == root->key) {
        taller = false;
        return false;  // No duplicate keys
    }
    if (key < root->key) {
        if (!insertAVL(root->pLeft, key, taller)) return false;
        if (taller) {
            switch (root->balFactor) {
                case LH:
                    leftBalance(root);
                    taller = false;
                    break;
                case EH:
                    root->balFactor = LH;
                    taller = true;
                    break;
                case RH:
                    root->balFactor = EH;
                    taller = false;
                    break;
            }
        }
    } else {
        if (!insertAVL(root->pRight, key, taller)) return false;
        if (taller) {
            switch (root->balFactor) {
                case RH:
                    rightBalance(root);
                    taller = false;
                    break;
                case EH:
                    root->balFactor = RH;
                    taller = true;
                    break;
                case LH:
                    root->balFactor = EH;
                    taller = false;
                    break;
            }
        }
    }
    return true;
}
```

Find and remove minimum

```
AVLTree deleteMin(AVLTree& root, bool&
shorter) {
    AVLTree node;
    if (!root->pLeft) {
        node = root;
        root = root->pRight;
        shorter = true;
    } else {
        node = deleteMin(root->pLeft,
shorter);
        if (shorter) {
            switch (root->balFactor) {
                case LH:
                    root->balFactor = EH;
                    shorter = true;
                    break;
                case EH:
                    root->balFactor = RH;
                    shorter = false;
                    break;
                case RH:
                    rightBalance(root);
                    shorter = true;
                    break;
            }
        }
    }
    return node;
}
```

Recursive deletion (1/2)

```
bool deleteAVL(AVLTree& root, Data key, bool& shorter) {
    if (!root) return false;

    if (key == root->key) {
        AVLTree temp = root;
        if (!root->pLeft) {
            root = root->pRight;
            delete temp;
            shorter = true;
        } else if (!root->pRight) {
            root = root->pLeft;
            delete temp;
            shorter = true;
        } else {
            AVLTree minNode = deleteMin(root->pRight, shorter);
            root->key = minNode->key;
            delete minNode;
            if (shorter) {
                switch (root->balFactor) {
                    case RH:
                        root->balFactor = EH;
                        shorter = true;
                        break;
                    case EH:
                        root->balFactor = LH;
                        shorter = false;
                        break;
                    case LH:
                        leftBalance(root);
                        shorter = true;
                        break;
                }
            }
        }
        return true;
}
```

Recursive deletion (2/2)

```
if (key < root->key) {
        if (!deleteAVL(root->pLeft, key, shorter))
return false;
        if (shorter) {
            switch (root->balFactor) {
                case LH:
                    root->balFactor = EH;
                    shorter = true;
                    break;
                case EH:
                    root->balFactor = RH;
                    shorter = false;
                    break;
                case RH:
                    rightBalance(root);
                    shorter = true;
                    break;
            }
        }
    } else {
        if (!deleteAVL(root->pRight, key, shorter))
return false;
        if (shorter) {
            switch (root->balFactor) {
                case RH:
                    root->balFactor = EH;
                    shorter = true;
                    break;
                case EH:
                    root->balFactor = LH;
                    shorter = false;
                    break;
                case LH:
                    leftBalance(root);
                    shorter = true;
                    break;
            }
        }
    }
    return true;
}
```

Outline

1. Introduction
2. Terminologies
3. Tree traversals
   – Breadth-First Search (BFS)
   – Depth-First Search (DFS)
4. Tree representation
5. Binary trees
6. Binary search trees (BST)
7. Balanced trees
8. AVL trees
9. 2-3, 2-3-4 trees
10. B-trees

Outline

1. Introduction
2. Terminologies
3. Tree traversals
   - Breadth-First Search (BFS)
   - Depth-First Search (DFS)
4. Tree representation
5. Binary trees
6. Binary search trees (BST)
7. Balanced trees
8. AVL trees
9. 2-3, 2-3-4 trees
10. B-trees

Q & A