

COMP_SCI 214: Data Structures and Algorithms

Binary Search Trees

PROF. SRUTI BHAGAVATULA

Announcements

- ▶ Project program due today
- ▶ Project document #1 due today
- ▶ HW5 self-evaluation due today

- ▶ Feedback on attempted functions in project coming Friday

Announcements

- ▶ Exam 2 is next Thursday (last day of lecture)
- ▶ Topics:
 - ▶ Heavy focus on **Graphs (Worksheet + Lec 8)** to **Binary Search Trees (Today and possibly Thursday; Lec 15 and 16)**
 - ▶ Older topics are fair game as above topics build on older ones
 - ▶ Lec 16 topic (Amortized Analysis) not in scope
- ▶ I'll release a practice exam later this week

Dictionaries Revisited

Problem of the day

- ▶ Recall our address book problem using a dictionary but with a twist:
 - ▶ Friend's info is stored against how long I've known them
 - ▶ **Key:** when we first met
 - ▶ **Value:** all the info about my friend
- ▶ Goals:
 - ▶ Lookup friends based on how long ago I met them
 - ▶ Lookup oldest friend
 - ▶ Add old friend's info long after I met them

Interface for our address book

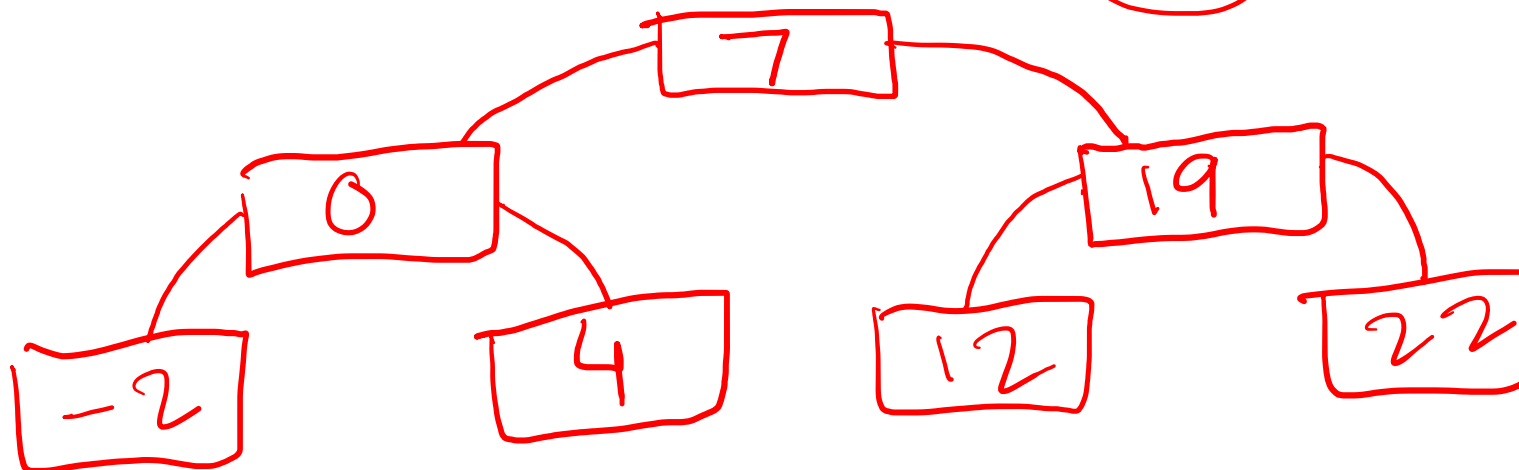
```
interface DICT[K, V]: # key and value types up to client
  def get(self, key: K) -> V
  def put(self, key: K, value: V) -> NoneC
  def find_min(self) -> V
```

Dictionaries so far

| | Unsorted arrays/ Linked lists | Sorted arrays | Hash tables | ?? |
|-----------------|----------------------------------|---------------|---------------------------|-------------|
| lookup | $O(n)$ | $O(\log n)$ | $O(1)$ avg + amortized | $O(\log n)$ |
| insert | $O(n)$ | $O(n)$ | $O(1)$ avg + amortized | $O(\log n)$ |
| find_min | $O(n)$ | $O(1)$ | $O(n)$ | $O(\log n)$ |

Searching for a number

| | | | | | | |
|----|---|---|---|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| -2 | 0 | 4 | 7 | 12 | 19 | 22 |



Binary Search Tree (BST)

- ▶ Deconstructing binary search gave us a Binary Search Tree!
- ▶ A Binary Search Tree is a binary tree data structure whose elements are ordered
 - ▶ We've seen one type of ordering of a binary tree (binary heap)
 - ▶ This is different

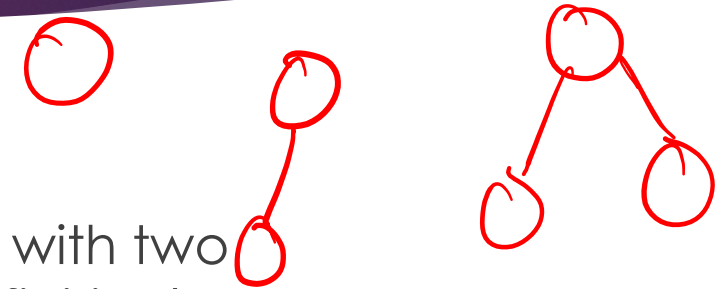
BST data structure invariants

► Valid binary tree invariant:

- The tree is either: (1) an empty tree or (2) a root node with two children, where each tree is a valid tree (recursive definition)

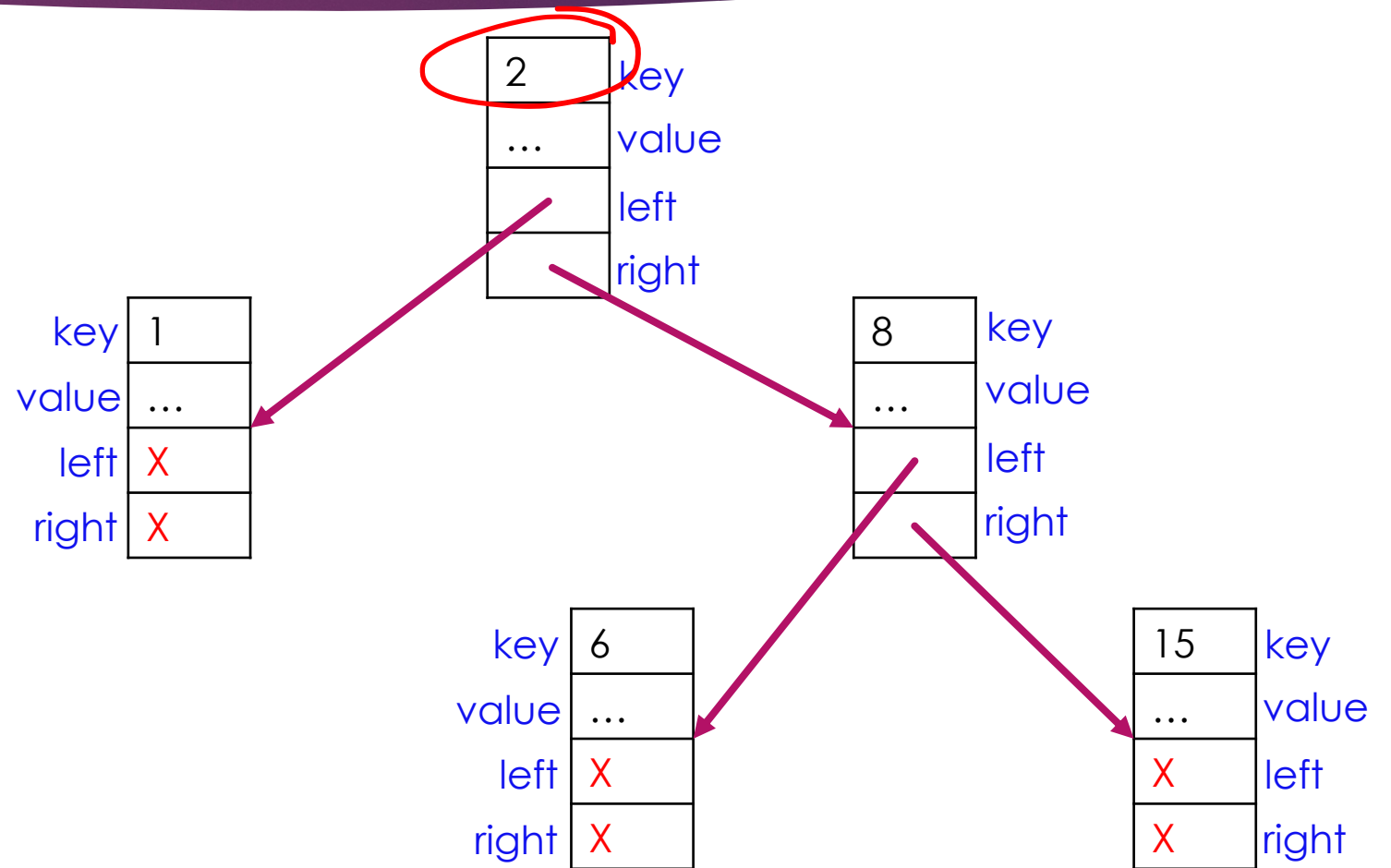
► Ordering invariant:

- Version #1: At any node with key k in a binary search tree, the key of the left child is less than k and the key of the right child is greater than k



BST data structure for a dictionary

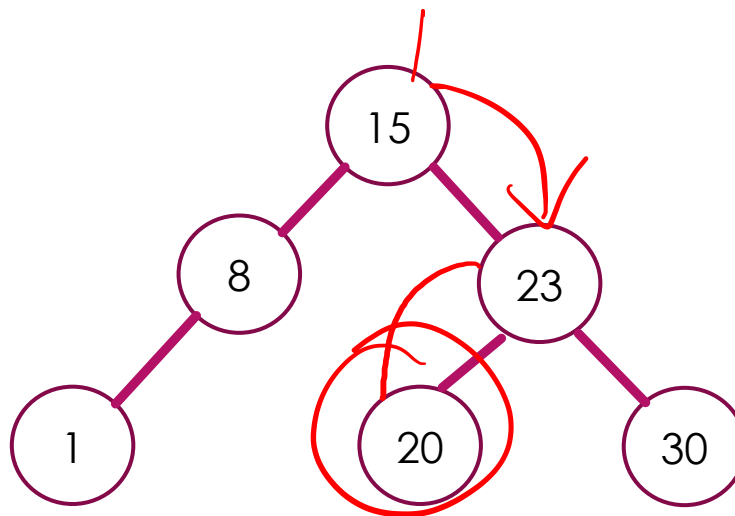
- We represent binary trees with a struct for each node



Dictionary operations with a BST

Lookup (get)

- Look up key 20 in the dictionary



Pseudocode for lookup

Function `get(root, key)`

Input: A node `root` and a key `key`

Output: A value, or nothing

```
curr ← root;
while curr is not None do
  if key < curr.key then
    curr ← curr.left
  else if key > curr.key then
    curr ← curr.right
  else
    return curr.value
  end
end
return None
end
```

Handwritten annotations:

- A red circle around `curr` in the first line.
- A red circle around `key` in the `if` condition.
- A red bracket on the right side of the `while` loop, labeled "success" in red.
- A red bracket on the right side of the `return None` line, labeled "unsuccessful" in red.

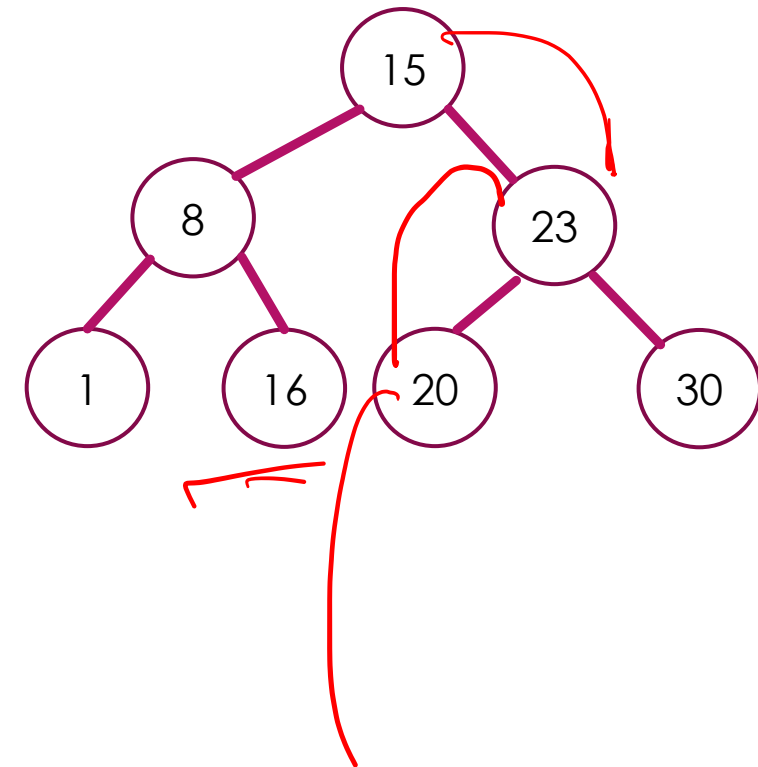
Seem familiar?

Looks like binary search code
But with subtrees instead of intervals!

We cut off one subtree at each step

In-class exercise (5 minutes)

- ▶ This tree satisfies the two invariants from before (tree invariant, ordering invariant version #1). **It contains the key 16.**
- 1. Following the algorithm, will a lookup of 16 be successful?
- 2. If so, what is the sequence of nodes that the algorithm looks at to arrive at the node 16?
If not, what is the issue (either with the invariant or the algorithm)?




BST data structure invariants

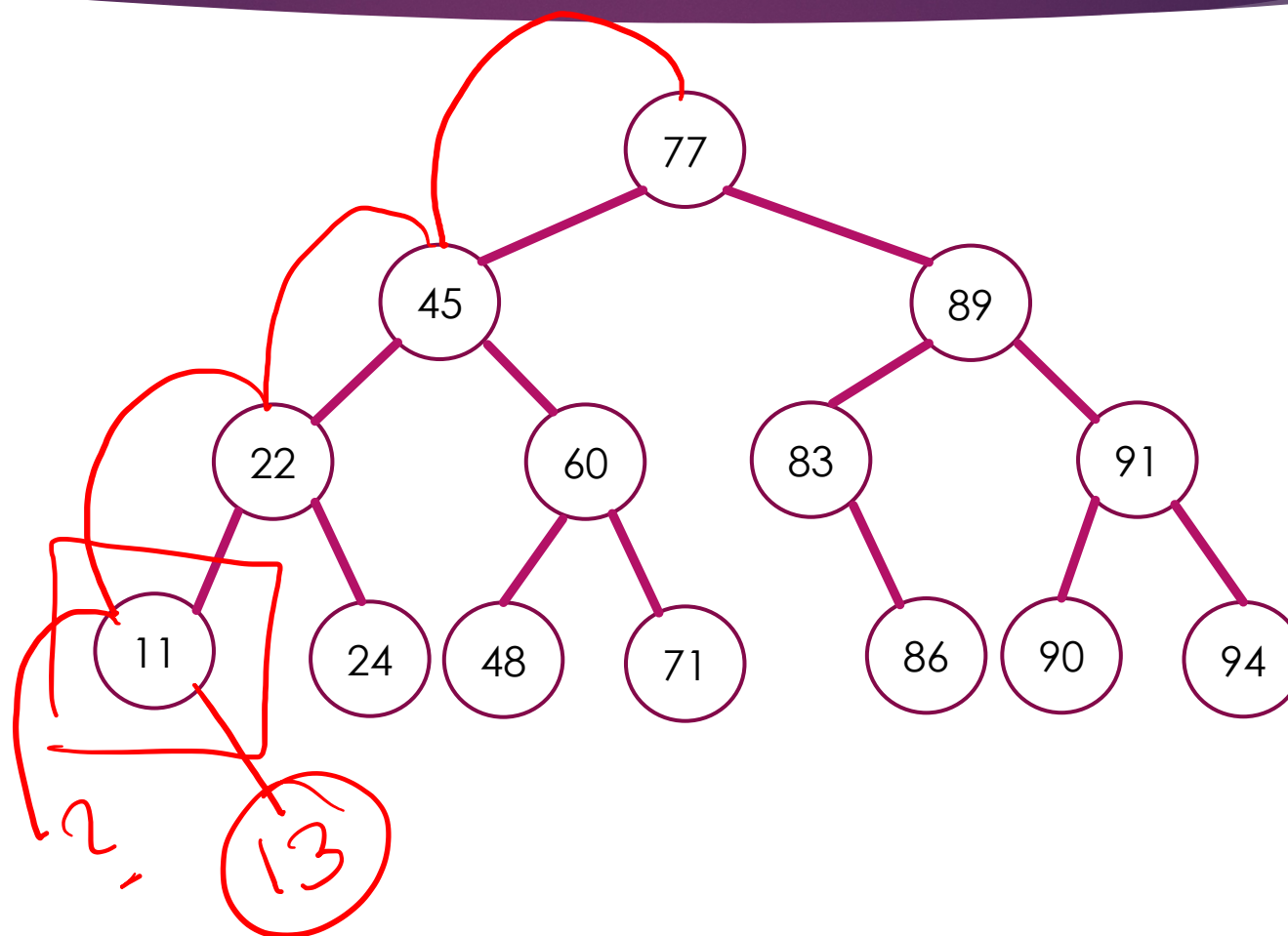
- ▶ **Valid binary tree invariant:**

- ▶ The tree is either: (1) an empty tree or (2) a root node with two children, where each tree is a valid tree (recursive definition)

- ▶ **Ordering invariant:**

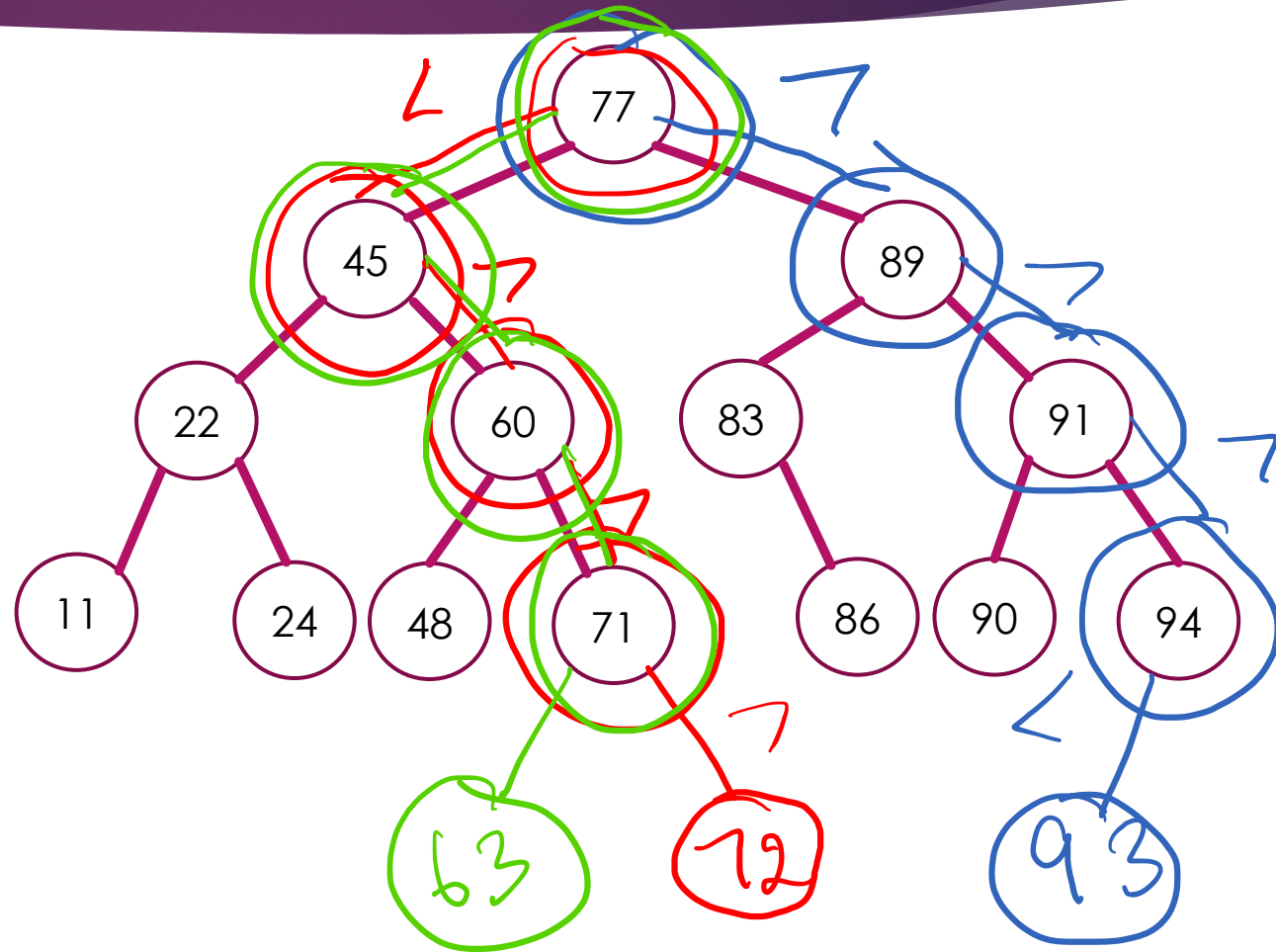
- ▶ Final correct version: At any node with key k in a binary search tree, the key of all the nodes in the left subtree is less than k and the key of all the nodes in the right subtree is greater than k
- 

Finding the min key (`find_min`)



Insert (put)

- Insert 72
- Insert 93
- Insert 63



Pseudocode for insert

Function `put(node, key, value)`

Output: the updated node

```
if node is None then
    return new node with key, value and two None children
else if key < node.key then
    node.left ← put(node.left, key, value);
else if key > node.key then
    node.right ← put(node.right, key, value);
    return node
else
    node.value = value;
    return node
end
```

Insertion always adds
a new leaf

Makes it easy! Right?

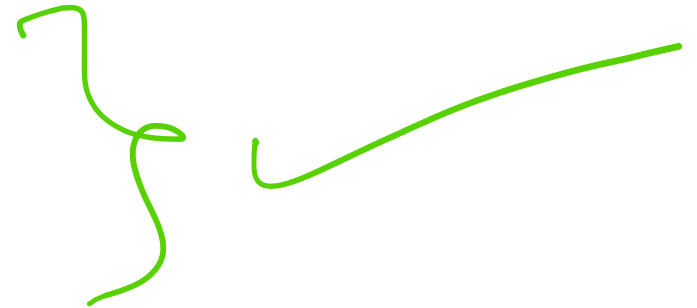
end

Complexity of operations

- ▶ Binary search on an array of size n takes at most $\lceil \log n \rceil$ steps
- ▶ Steps in binary search formed a tree structure where each tree level corresponded to a step
- ▶ Remember, height of a complete tree is $\lceil \log n \rceil$ and is in $O(\log n)$
- ▶ All operations involve traversing only one path of a tree
 - ▶ Insert, lookup, and finding min all have $O(\log n)$ complexity
 - ▶ Just like in a binary heap (tree)! But now we know why a tree gives us this

Good to go for our address book?

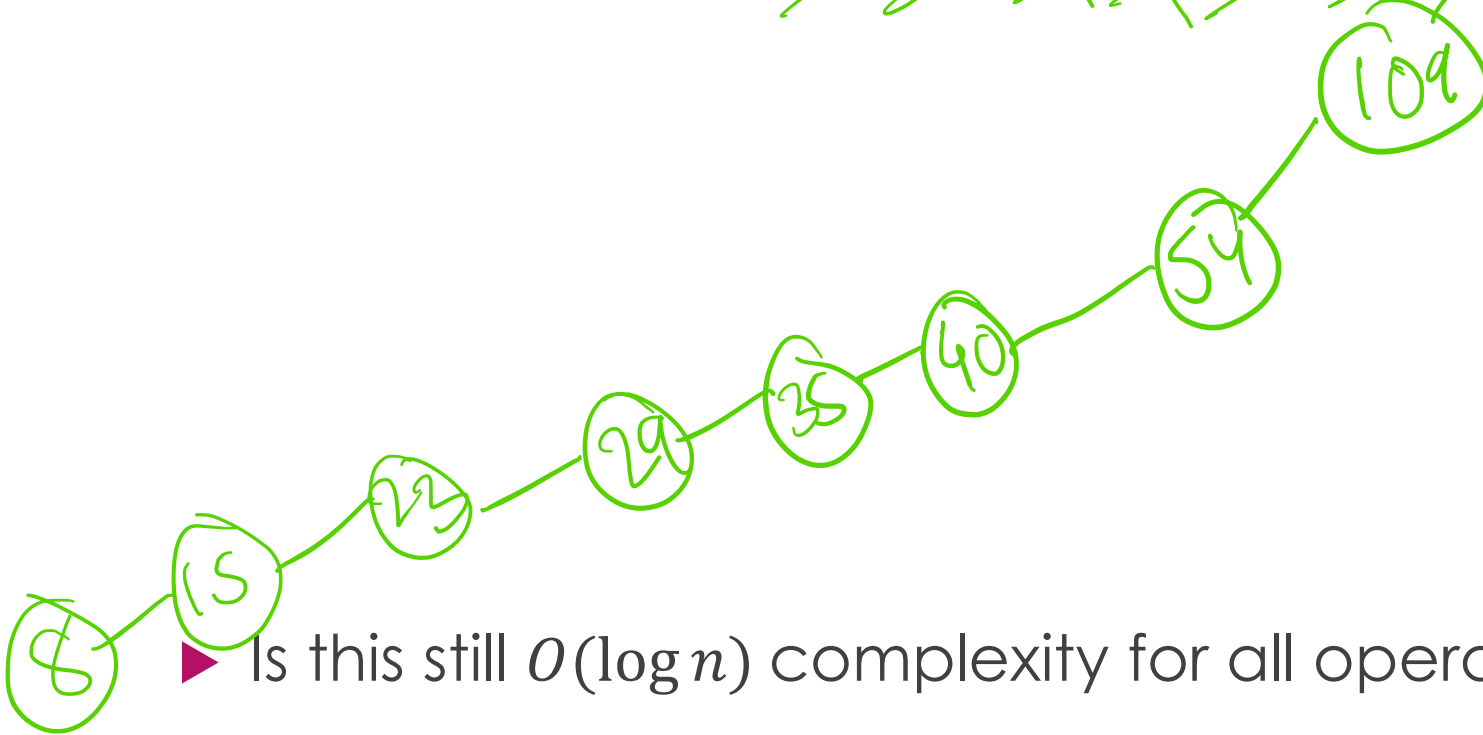
- ▶ Recall our address book problem using a dictionary but with a twist:
 - ▶ Friend's info is stored against how long I've known them
 - ▶ **Key:** when we first met
 - ▶ **Value:** all the info about my friend
- ▶ Goals:
 - ▶ Lookup friends based on how long ago I met them
 - ▶ Lookup oldest friend
 - ▶ Add old friend's info long after I met them



Let's try building a brand new BST

- Build a BST by inserting these elements in order:

~~109, 54, 40, 35, 29, 23, 15, 8~~



- Is this still $O(\log n)$ complexity for all operations?

Requirements for $O(\log n)$

- ▶ Either:
 - ▶ Tree should be complete (like with heaps) --- but BST invariant can't ensure completeness
 - ▶ Tree should be perfectly balanced --- this is just a complete tree as a perfect triangle so as hard as above
- ▶ Instead, tree should be *almost* balanced: no path in the tree is more than 2x long as another
 - ▶ That's enough to get $O(\log n)$ as constants don't matter
 - ▶ We'll consider this to mean **“balanced”**

Complexity of operations

- ▶ Binary search on an array of size n takes at most $\lceil \log n \rceil$ steps
- ▶ Steps in binary search formed a tree structure where each tree level corresponded to a step
- ▶ Height of a tree is $\lceil \log n \rceil$ and is in $O(\log n)$
- ▶ All operations involve traversing only one path of a tree
 - ▶ Insert, lookup, remove, and finding min all have $O(\log n)$ complexity
 - ▶ Only when the tree is balanced

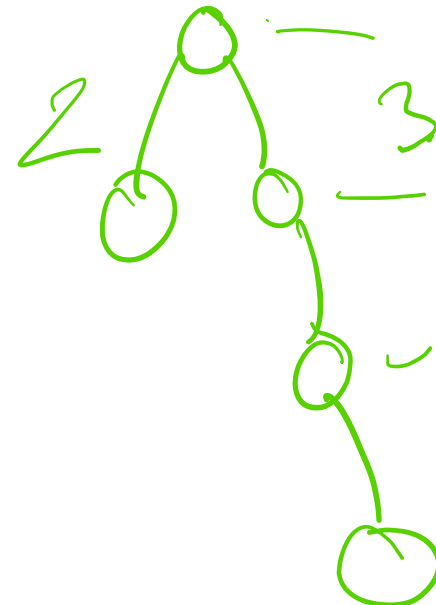
The solution: Self-balancing BST

- ▶ **Observation:** insertions (or deletions) may introduce imbalances (not lookups or updates)
- ▶ **Strategy:**
 - ▶ Introduce another invariant for the trees: a balance invariant *height*
 - ▶ Make sure that the invariant is satisfied at the end of one insertion
 - ▶ Perform insertion as before
 - ▶ If height invariant is broken, repair and restore height invariant while maintaining ordering invariant
- ▶ **Idea:** Fix as we go along, so we only have a small imbalance to fix each time

Think globally, act locally

0 1

- ▶ Balance is a global property about the entire tree
- ▶ But we can maintain this global property by checking and enforcing a balance invariant locally on all sub-trees
- ▶ Each node will store a balance factor: which is the difference between the height of its right subtree and its left one



AVL trees

AVL tree

- ▶ AVL tree is the first proposed self-balancing BST
- ▶ Invented by G.M. Adelson-Velsky & E.M. Landis in 1962

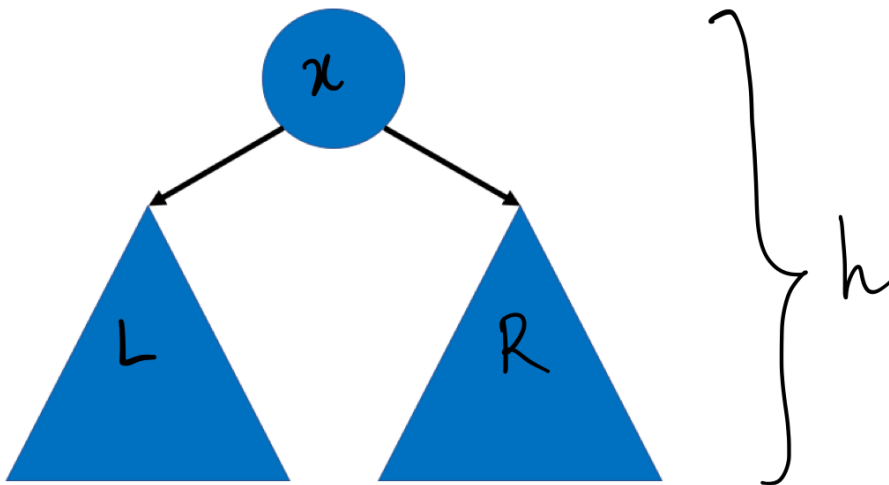
AVL tree invariants

- ▶ **Ordering invariant** (*as before*)

- ▶ **Balance invariant** (*new!*): At every node, the heights of the left and right subtrees differ by at most 1 **or** the balance factor is between -1 and +1

Balance invariant

- At every node, the heights of the left and right subtrees differ by at most 1 **or** the balance factor is between -1 and +1



- **Case 1:** $\text{height}(R) = \text{height}(L) = h-1$
 - Balance factor $(x) = 0$
- **Case 2:** $\text{height}(R) = h-2, \text{height}(L) = h-1$
 - Balance factor $(x) = -1$
- **Case 3:** $\text{height}(R) = h-1, \text{height}(L) = h-2$
 - Balance factor $(x) = 1$

How does it self-balance?

- ▶ If a normal insertion or deletion causes a node's balance factor to go outside the range -1 and $+1$, we have detected imbalance
- ▶ Repair with one or a series of rotations

Recall normal insertion

Function `put(node, key, value)`

Output: the updated node

```
if node is None then
    return new node with key, value and two None children
else if key < node.key then
    node.left ← put(node.left, key, value);
else if key > node.key then
    node.right ← put(node.right, key, value);
    return node
else
    node.value = value;
    return node
end
end
```

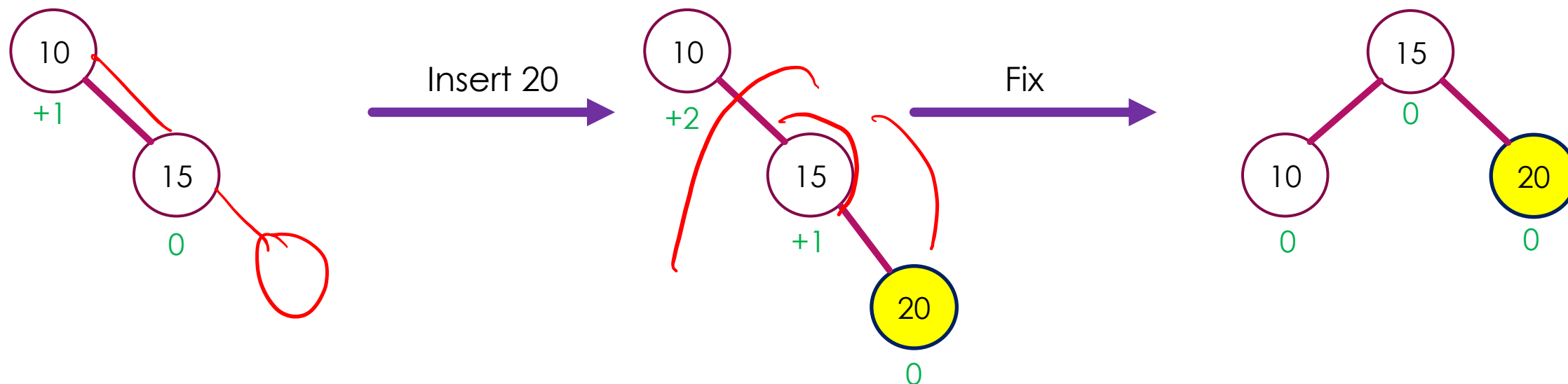
AVL insertion

- ▶ First do a normal leaf insertion (recursively)
- ▶ At each recursive call to insert, insertion returns whether the height of the subtree increased
- ▶ Based on that, adjust balance factors on the way back up (while returning from recursive calls)
- ▶ When adjusting for one node, if the balance factor becomes -2 or $+2$, rotate at this node to restore balance
- ▶ Keep returning all the way up to the root

Types of rotations

- ▶ 4 possible types of rotations to do a repair
 - ▶ Single left rotation
 - ▶ Single right rotation
 - ▶ Right-left double rotation
 - ▶ Left-right double rotation
- ▶ To restore an invariant at one node, we'll do one of these rotations

Example 1

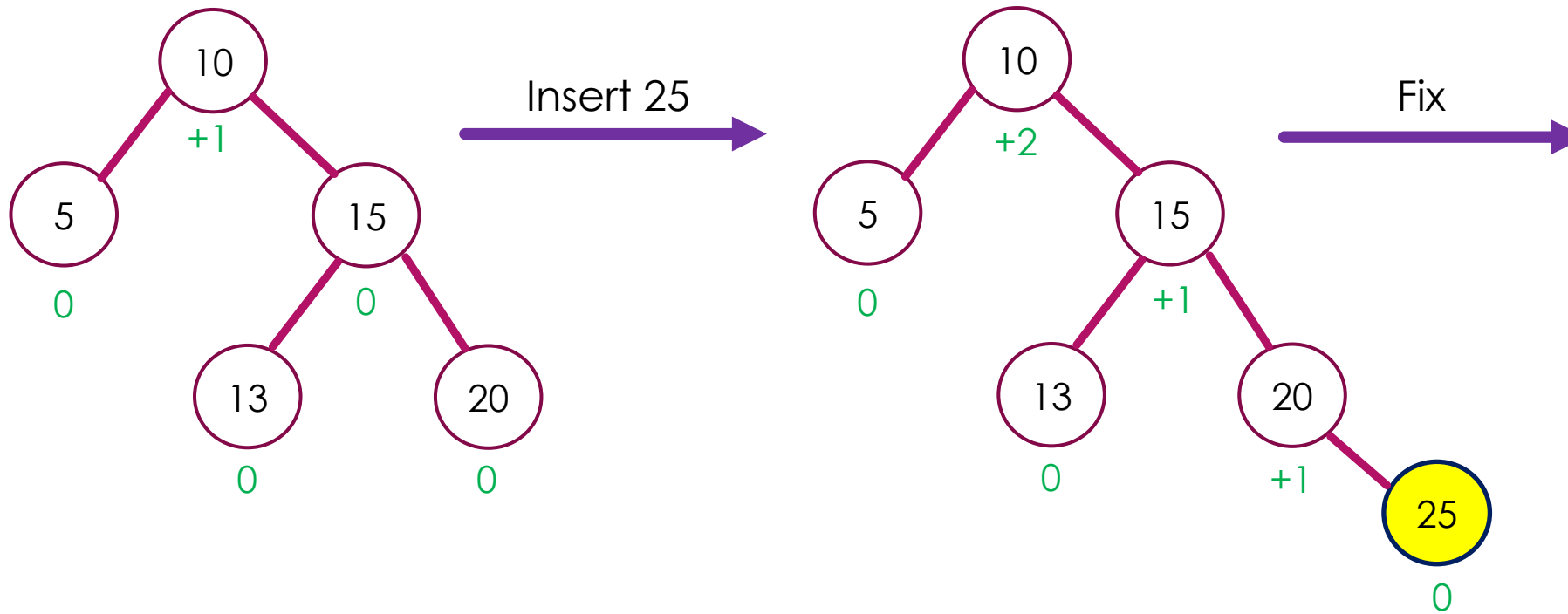


Single left-rotation at 10!

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

This is the only
possible tree that
satisfies both
invariants

Example 2

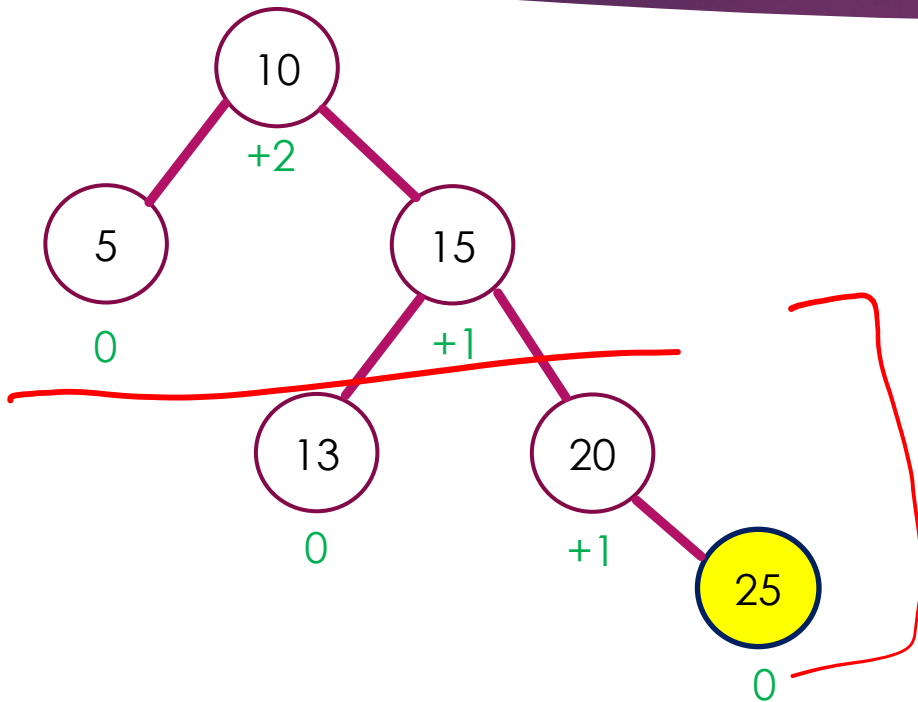


?

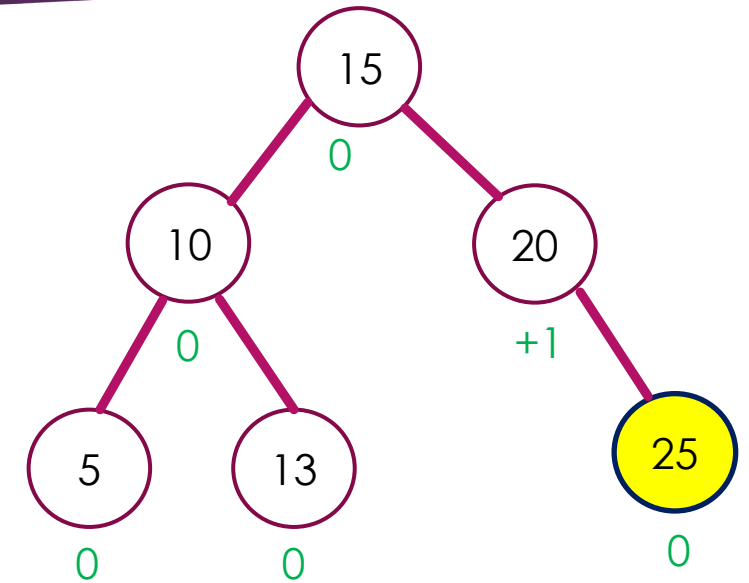
Inserting 25 as in a
BST causes a
violation at node 10

There are a lot of
AVL trees with these
elements:
which to pick?

Example 2



Fix

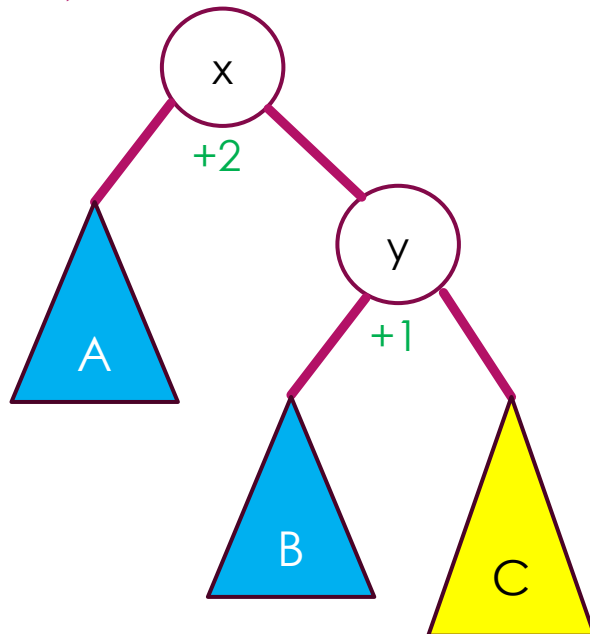


Here is one version

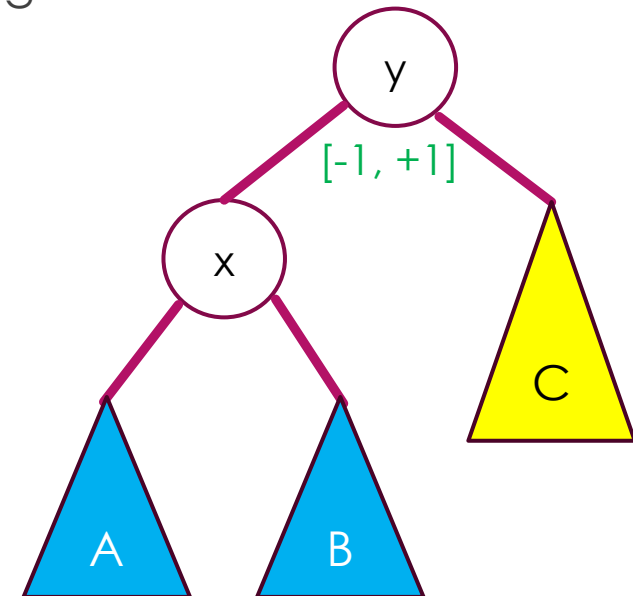
Single left-rotation at 10!

Generalization of left rotation

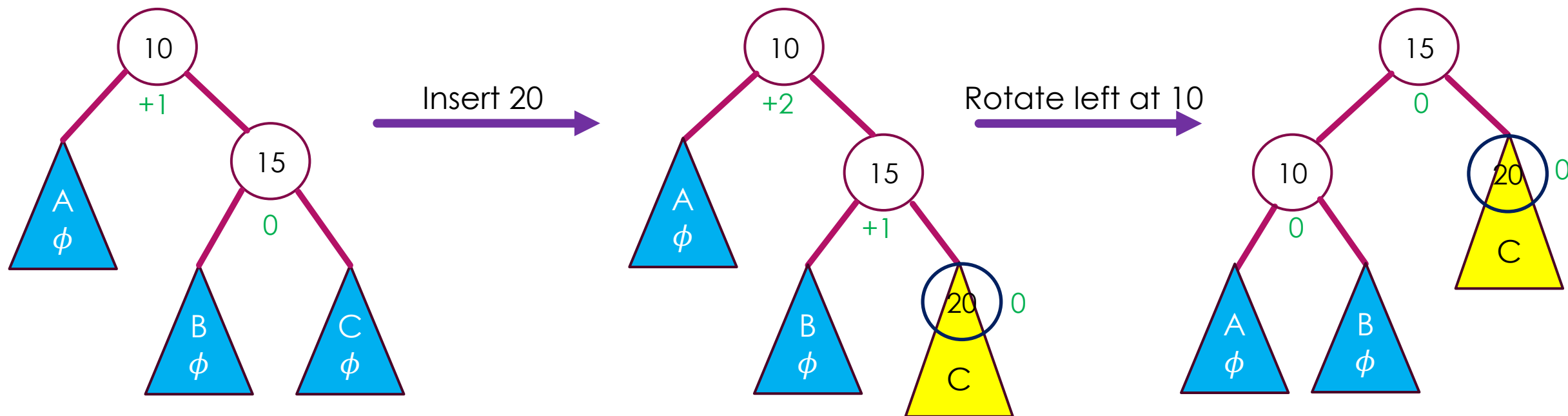
- ▶ If the balance invariant at x is violated because of its **right-right subtree** increasing in height by 1, do a single left rotation on x
 - ▶ If balance factor of x is $+2$, and balance factor of right child is $+1$



Single left rotation
→



Example 1 revisited



Generalization applies in both Examples 1 and 2

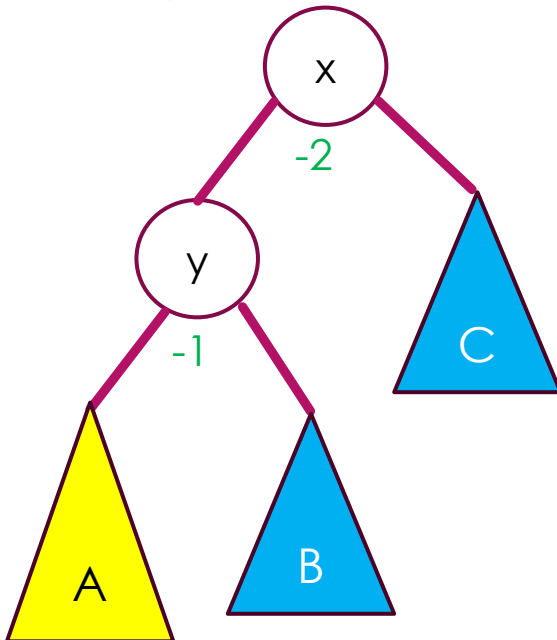
This is the same tree we produced the first time

Single right rotation

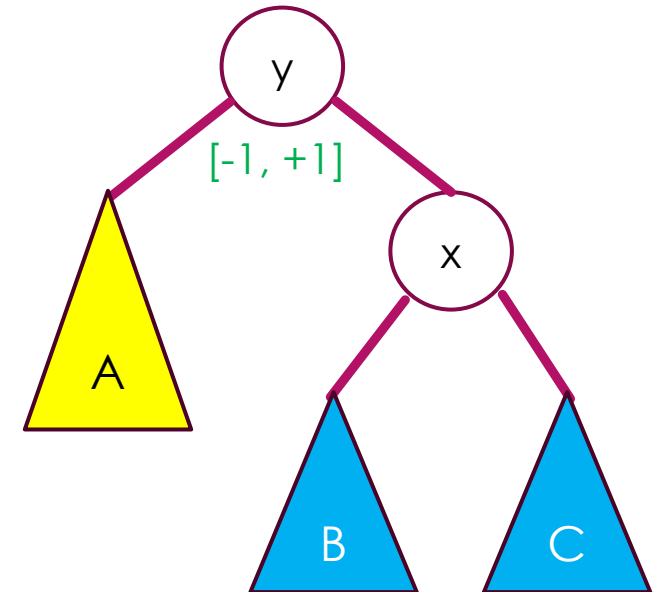
- ▶ The symmetric case warrants a single right rotation

Generalization of right rotation

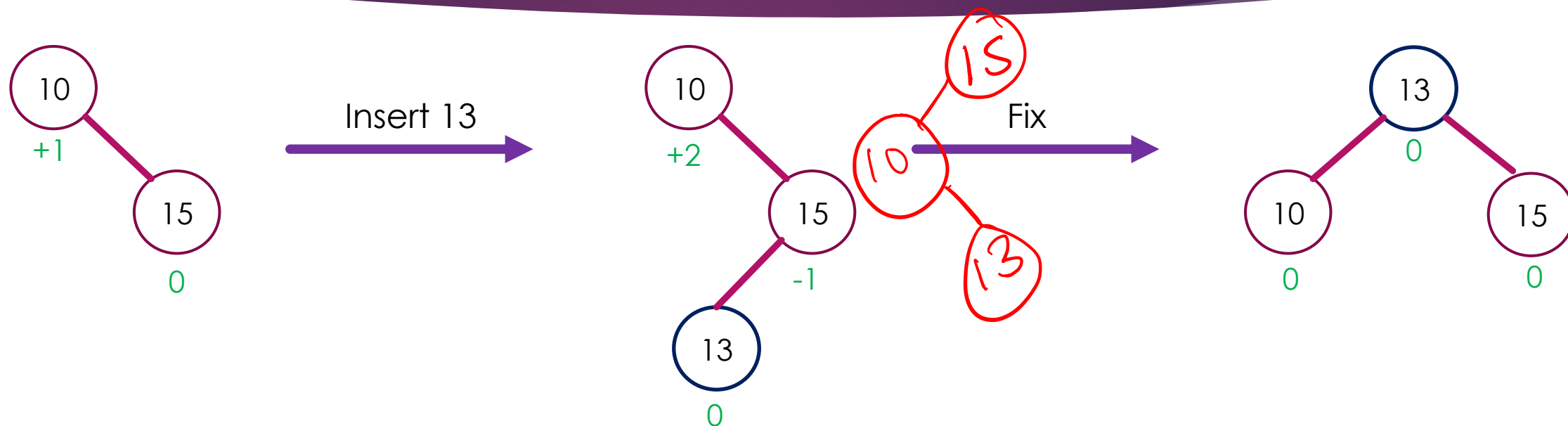
- ▶ If the balance invariant at x is violated because of its **left-left subtree** increasing in height by 1, do a single right rotation on x
 - ▶ If balance factor of x is -2 , and balance factor of left child is -1



Single right rotation
→



Example 3

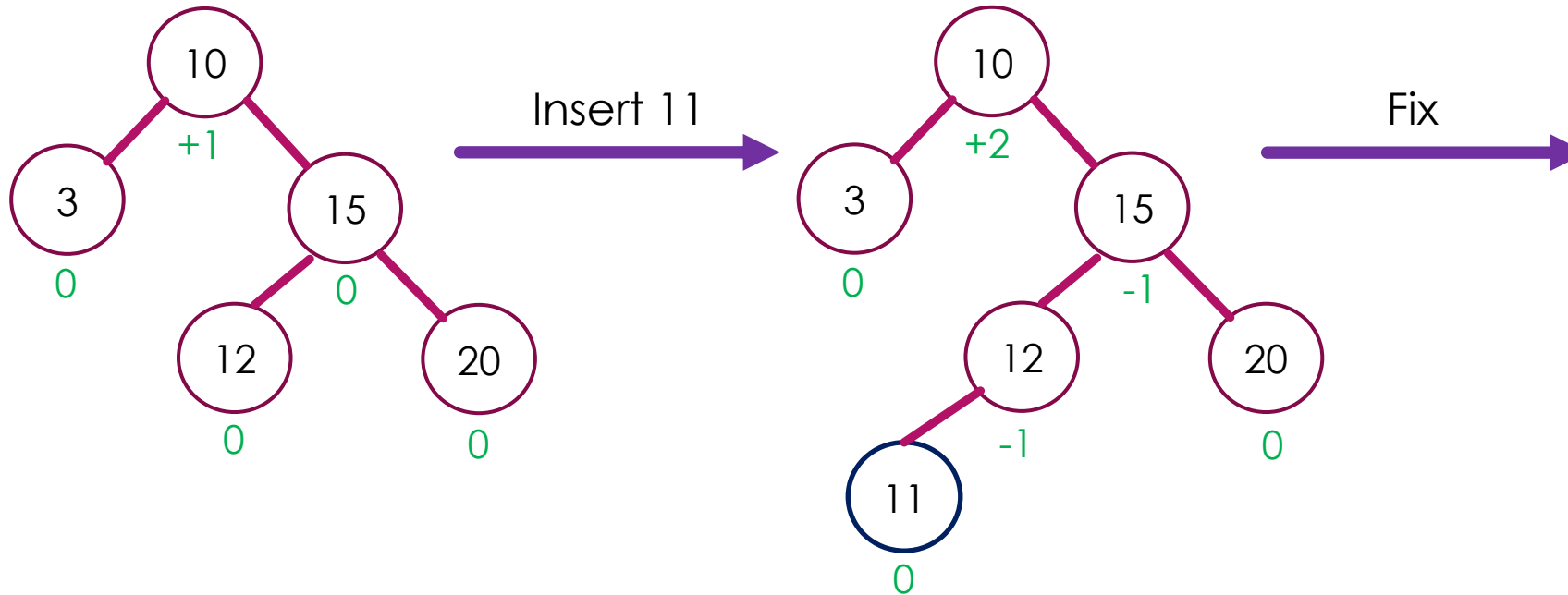


Right-left double rotation at 10!

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

This is the only
possible tree that
satisfies both
invariants

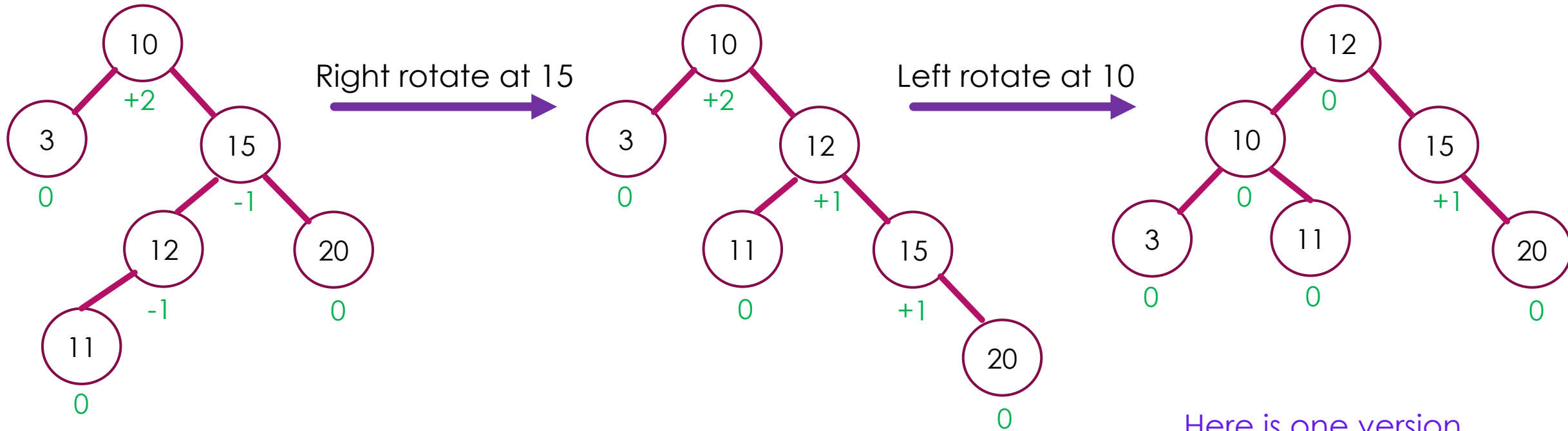
Example 4



Inserting 11 as in a
BST causes a
violation at node 10

There are a lot of
AVL trees with these
elements:
which to pick?

Example 4

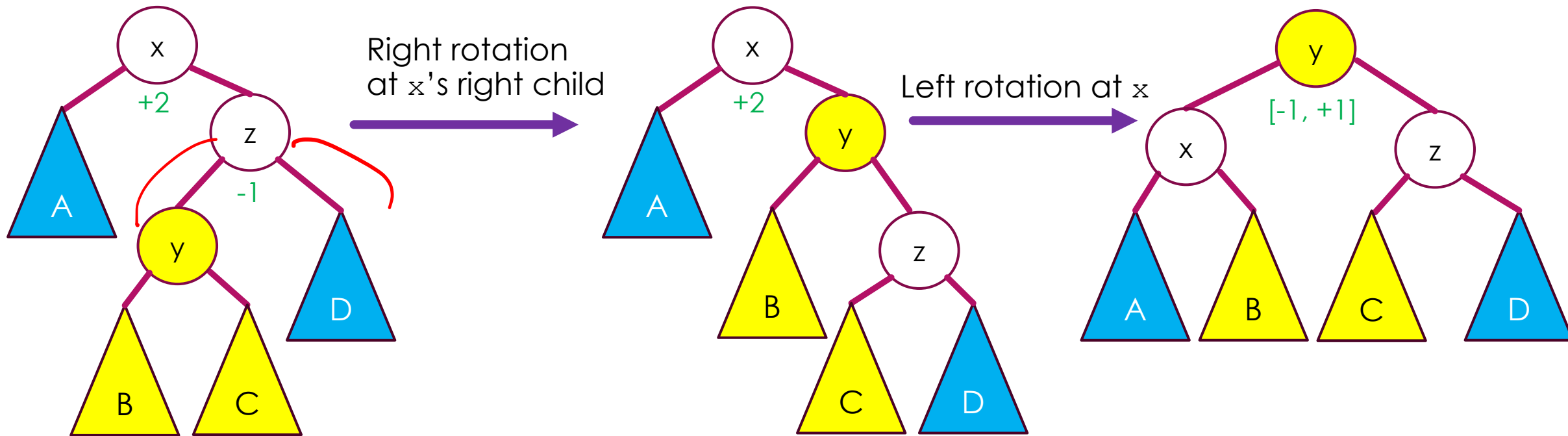


Here is one version

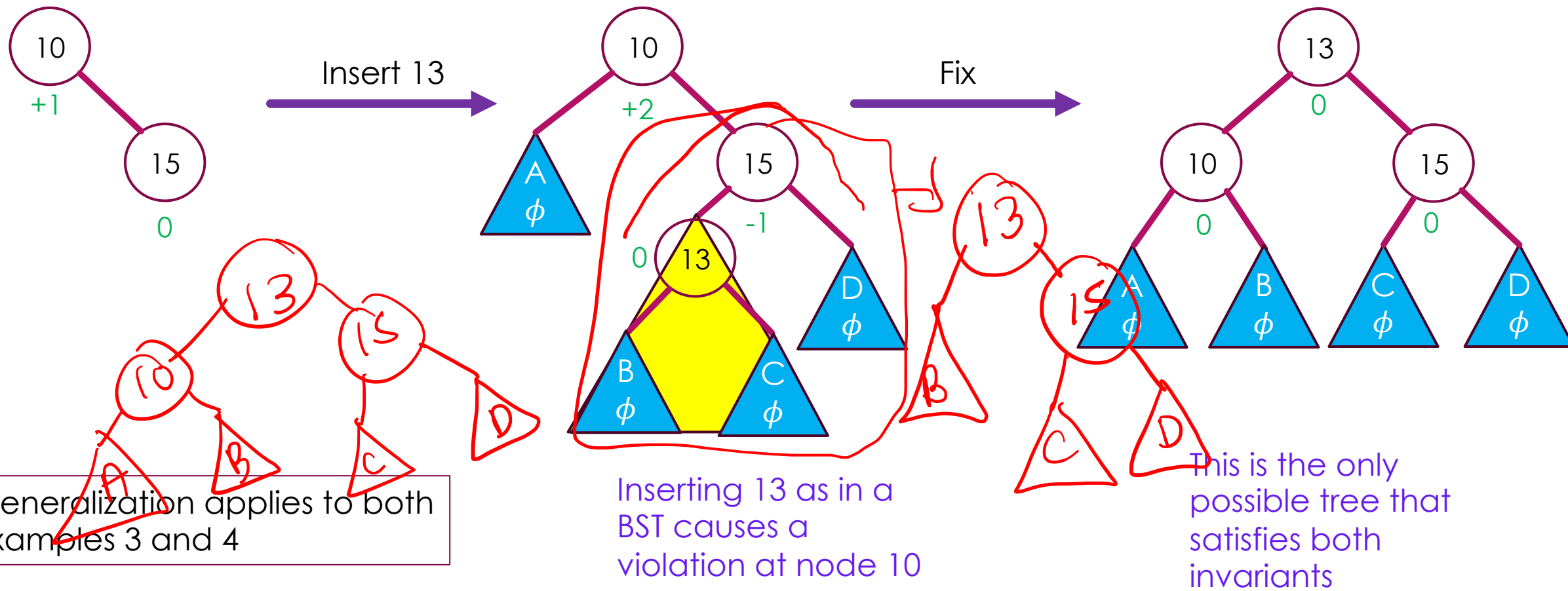
Double right-left rotation at 10!

Generalization of right-left rotation

- If the balance invariant at x is violated because of its **right-left subtree** increasing in height by 1, do a double right-left rotation on x
 - If balance factor of x is $+2$, balance factor of right child is -1



Example 3 revisited

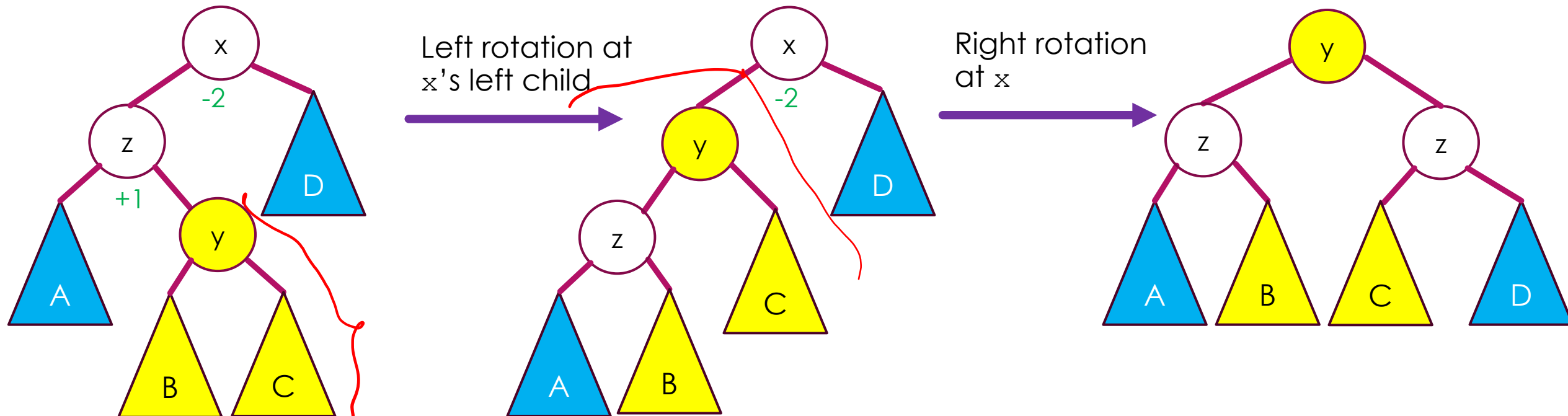


Double left-right rotation

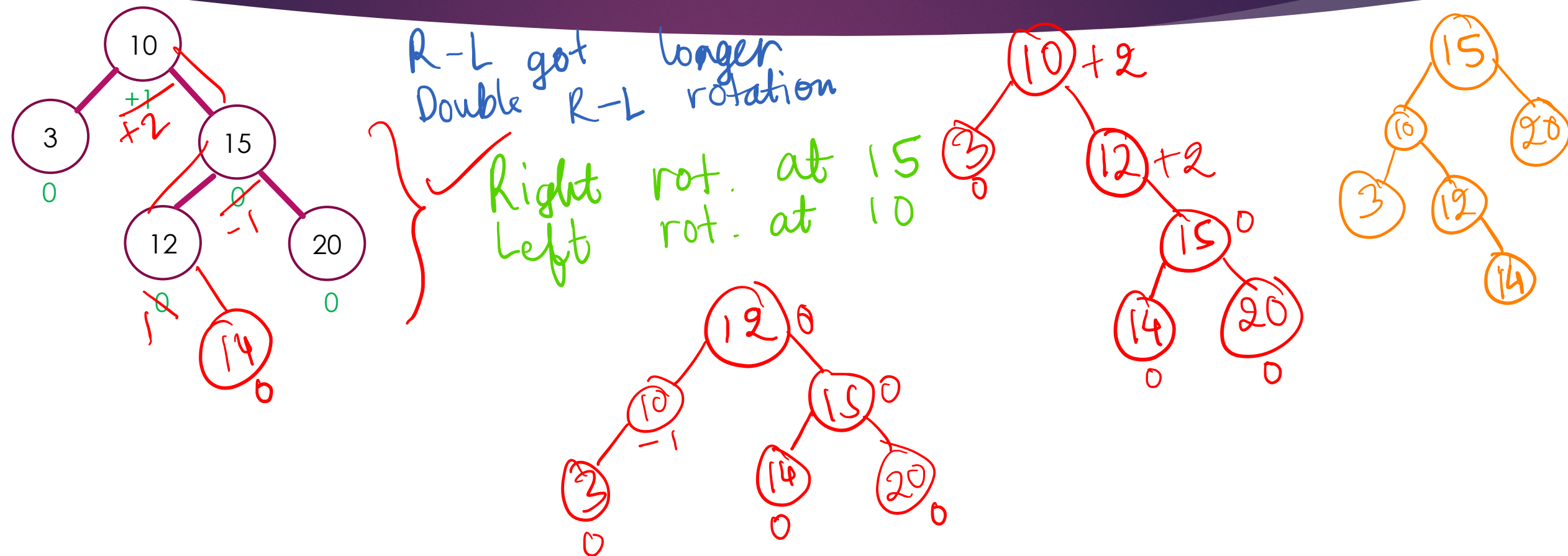
- ▶ Symmetric case warrants double left-right rotation

Generalization of left-right rotation

- ▶ If the balance invariant at x is violated because of its **left-right subtree** increasing in height by 1, do a double left-right rotation on x
 - ▶ If balance factor of x is -2 , balance factor of left child is $+1$



A full example: Insert 14



Exercise

1 ► At which node is the height invariant violated?

42

2 ► Which of the 4 rotations at that node would fix the violation?

Left

