# Algorithms and Data Structures for Data Science

## AVL Trees

CS 277
Brad Solomon

October 25, 2021

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

# Final Project Proposals

Please notify me when you have resubmitted

Email or post on Campuswire if you want to schedule a meeting

# Final Project Development Log

Pick a day each week to write up a summary of progress

For full credit, must have an entry each week starting this week

If project is not yet approved, just discuss the progress on proposal

# reflection on lab_trees

Avg: 15/15

# Exam 2 October 29th - October 31st

Exam will be combination multiple choice and coding

Covers all content up until October 20th

Sign up for registration as early as October 14th

Coding question will use new workspace setup.

# Learning Objectives

Identify four AVL rotations

Discuss how AVL functions relate to BST functions

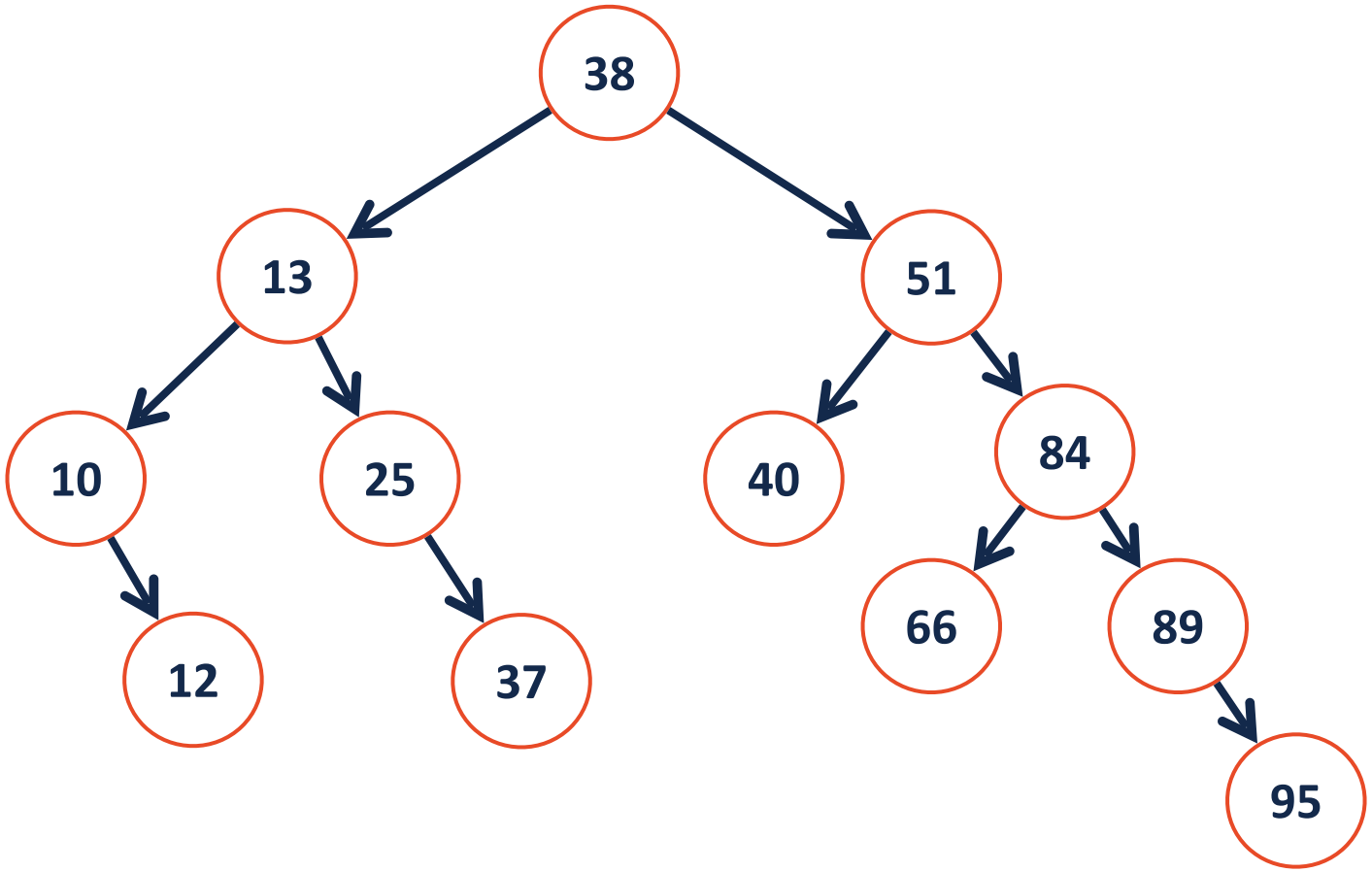
Prove that the AVL tree's height is bounded

# BST Analysis – Running Time

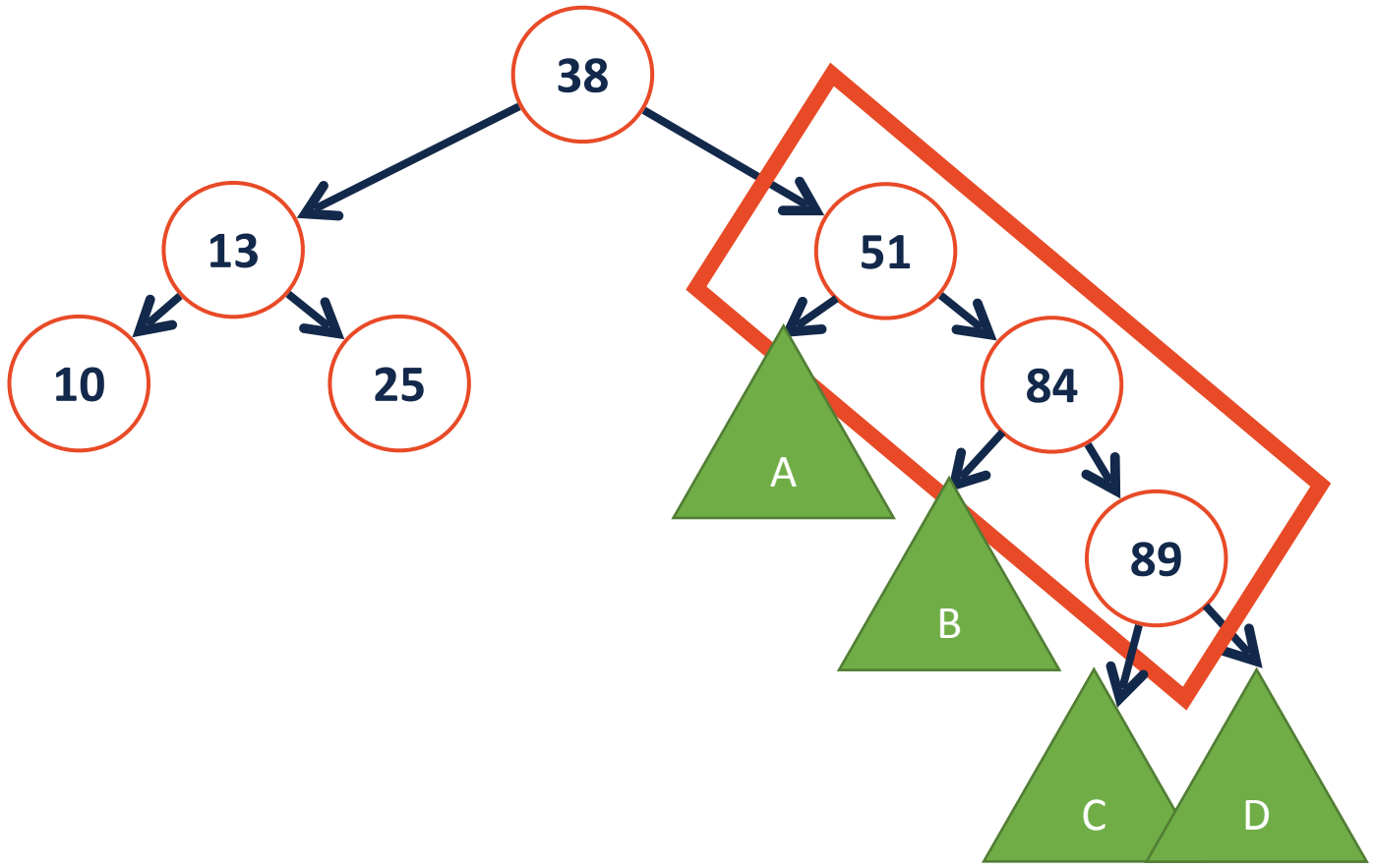
| Operation | BST Worst Case |
|---|---|
| find | |
| insert | |
| remove | |
| traverse | |

# BST Rotations

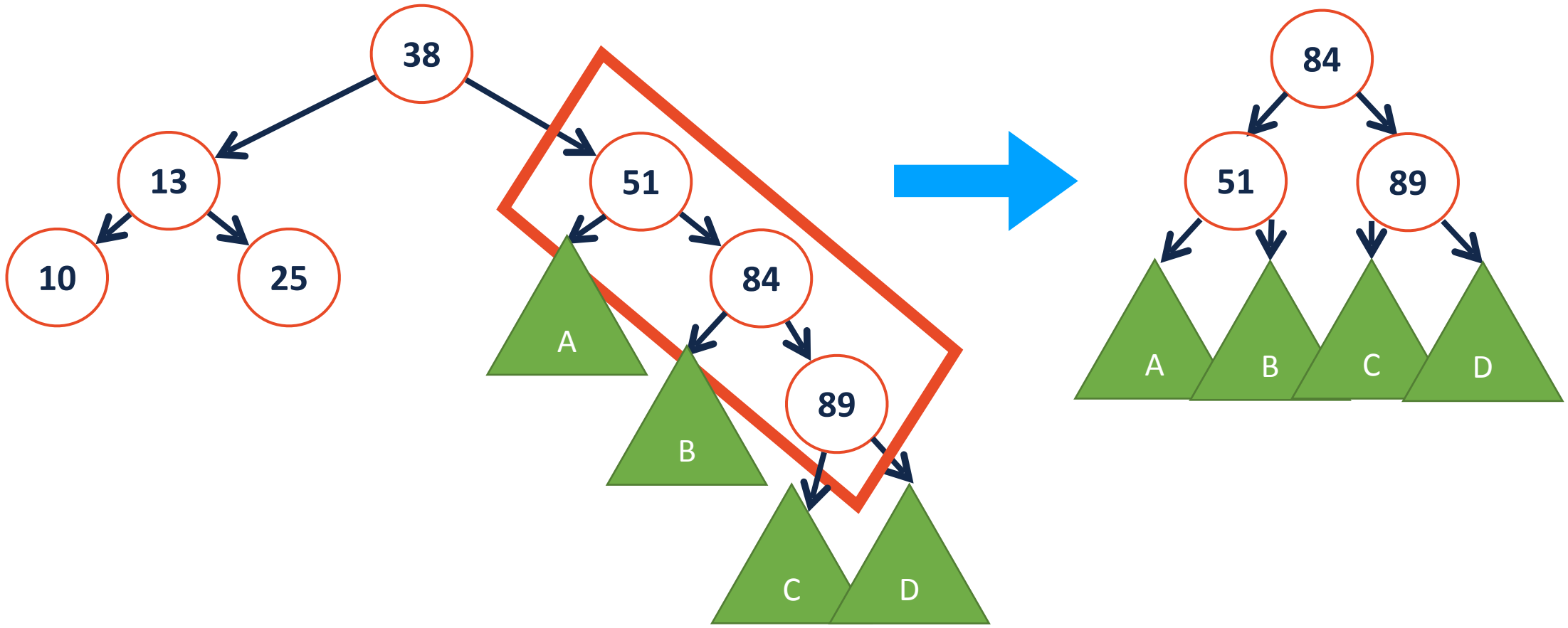
We can adjust the BST structure by performing **rotations**.



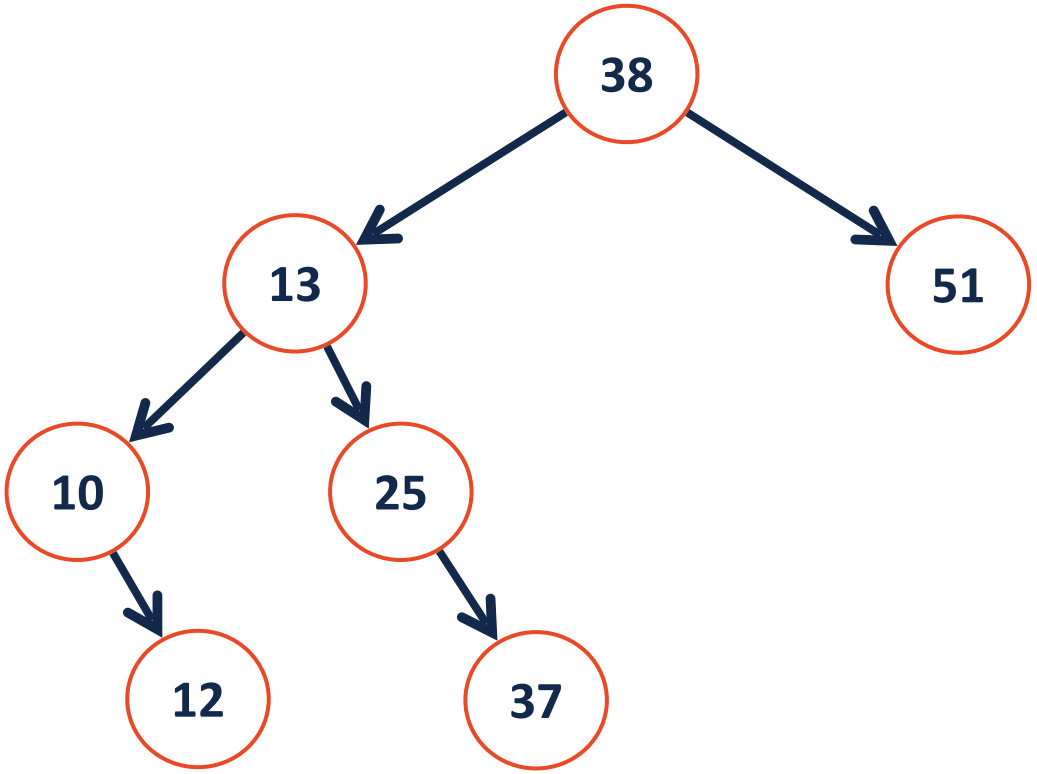
# Left Rotation



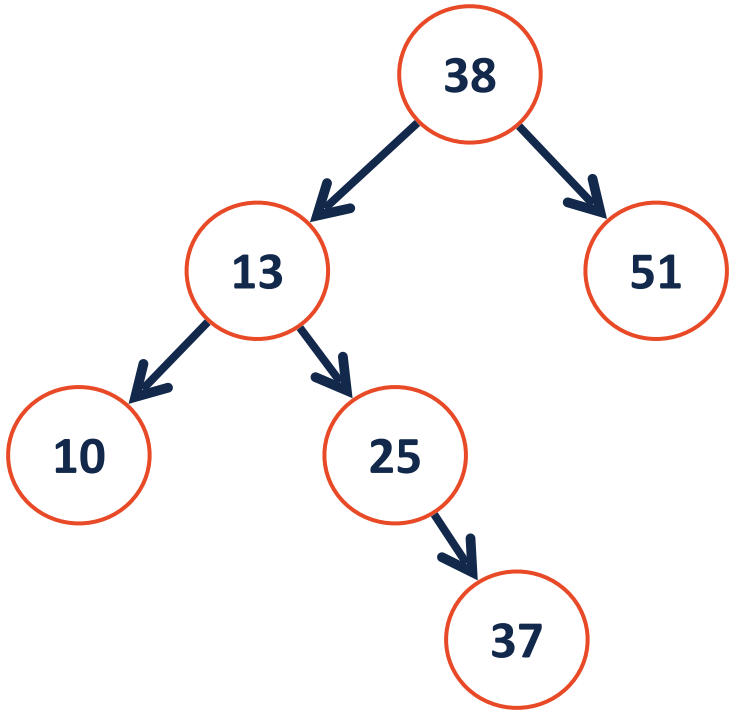
# Left Rotation



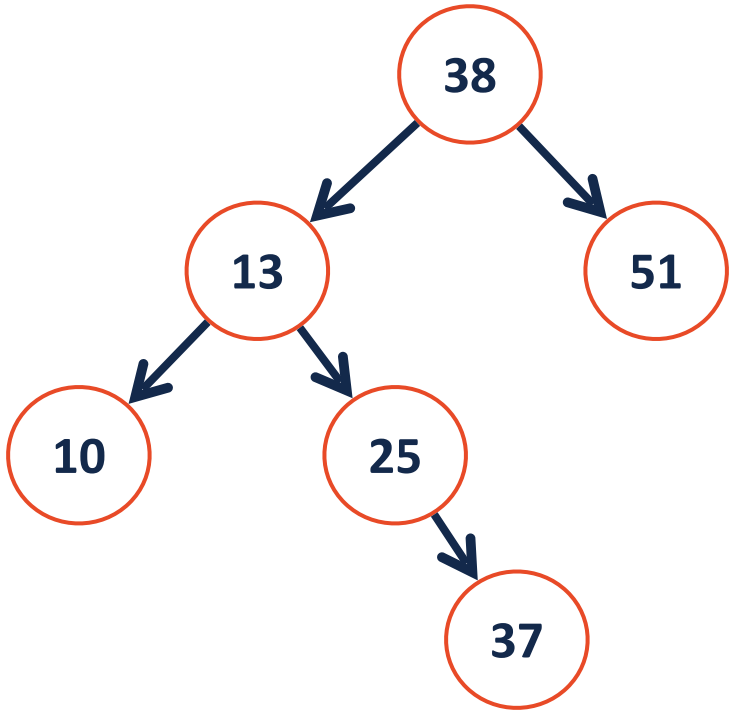
# Right Rotation



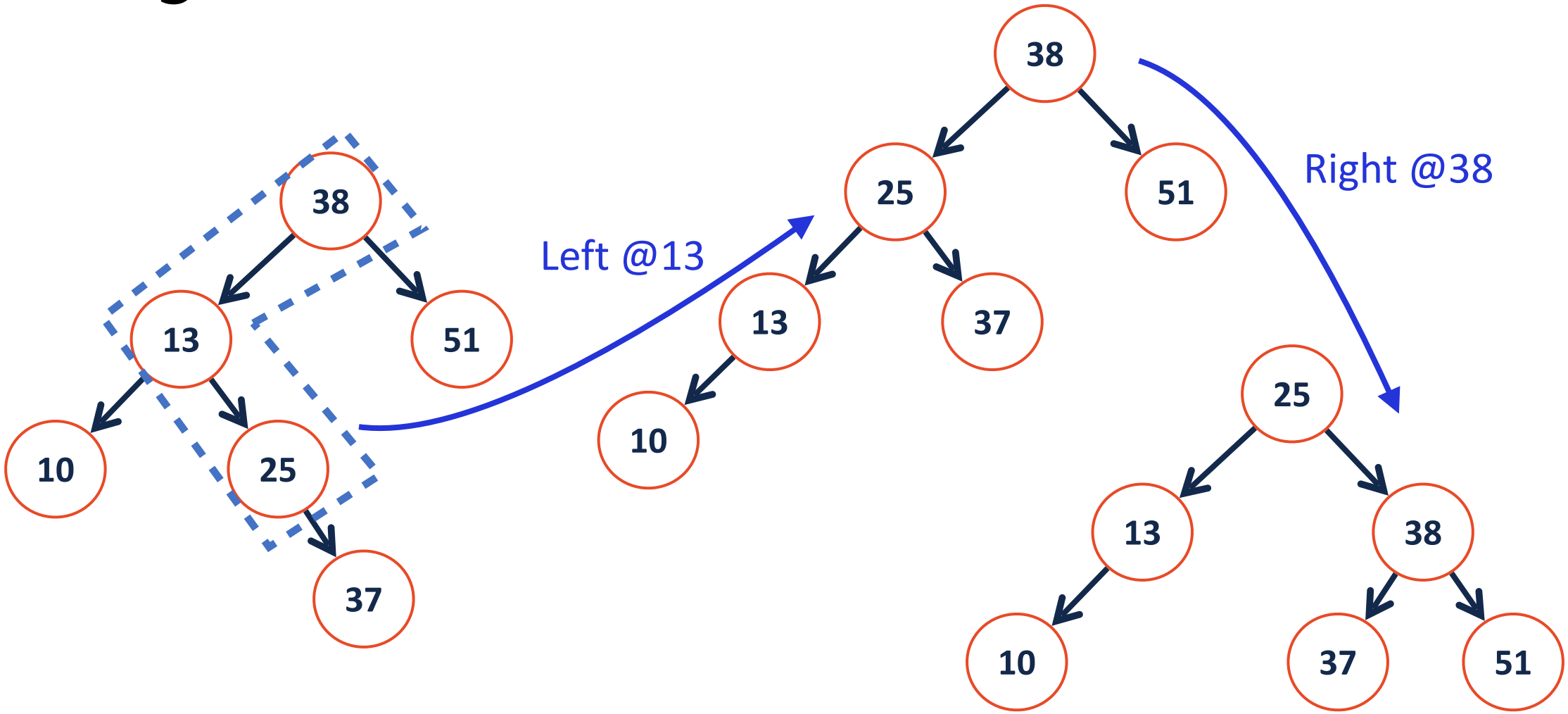
# Complex rotations



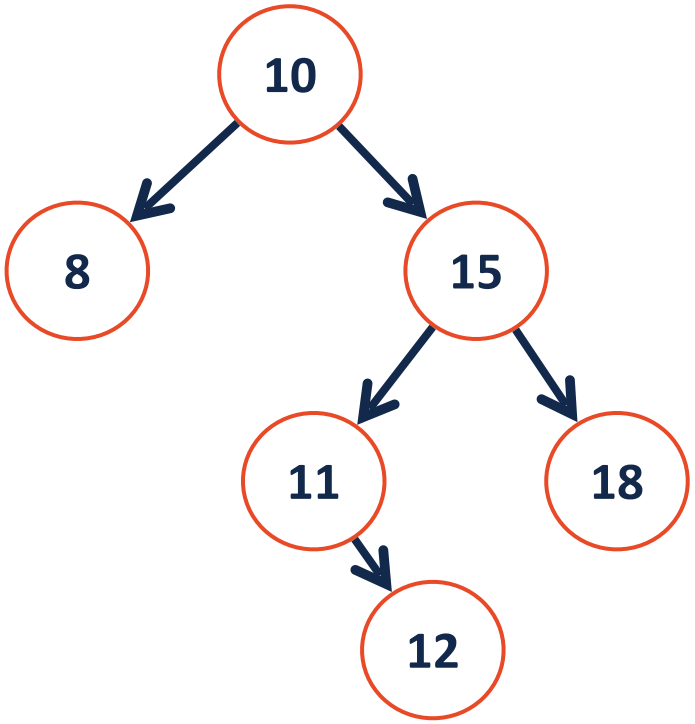
# LeftRight Rotation



# LeftRight Rotation

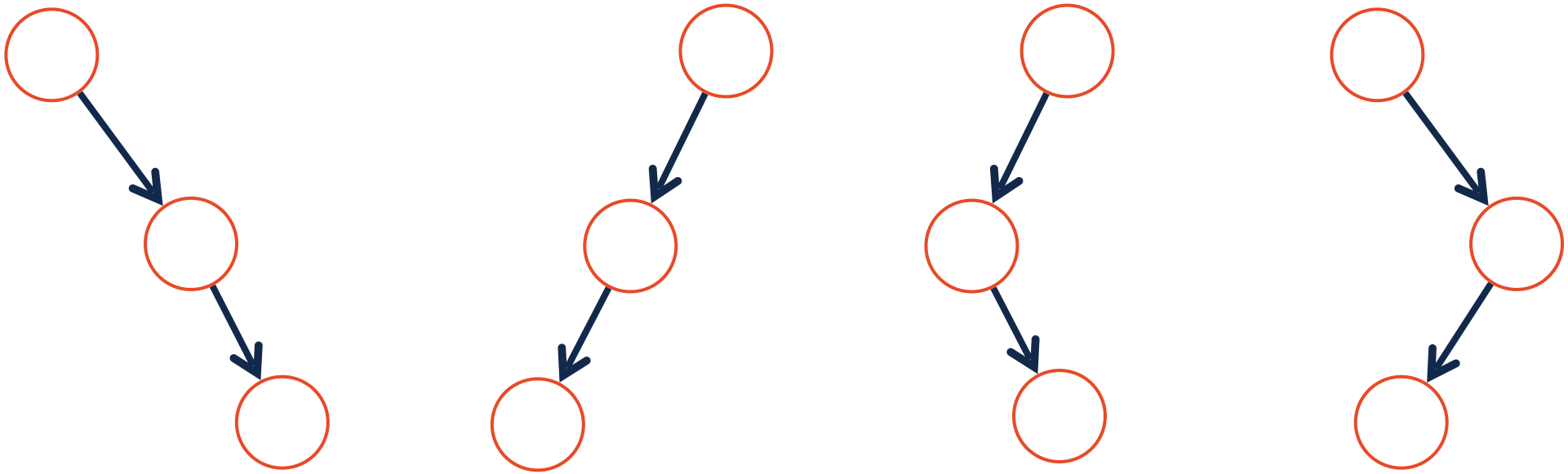


# RightLeft Rotation



# AVL Rotations

Four kinds of rotations:



# AVL Rotations

Four kinds of rotations: (L, R, LR, RL)

1. All rotations are local (subtrees are not impacted)

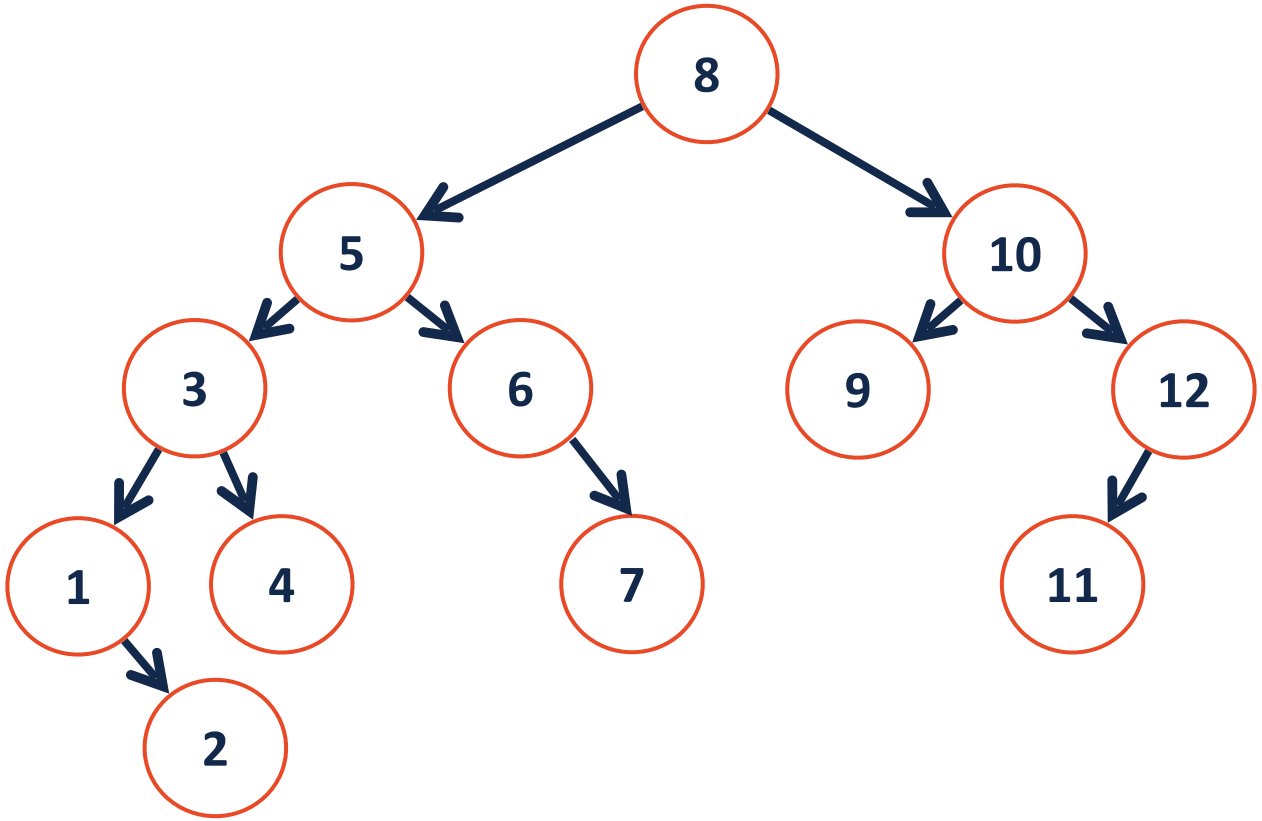
2. The running time of rotations are constant

3. The rotations maintain BST property

**Goal:**

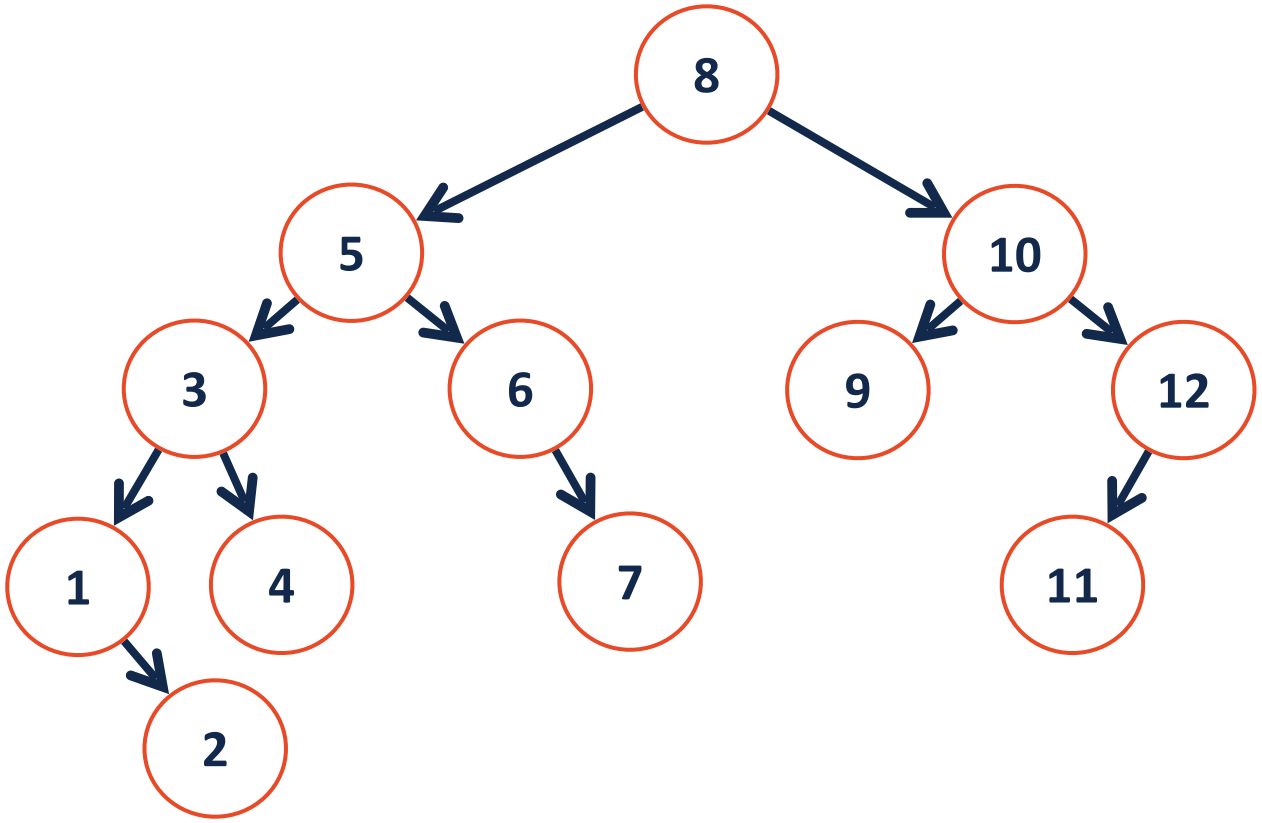
# AVL Find

`_find(7)`



# AVL Insertion

_insert(6.5)




```
class treeNode:
    def __init__(self, key, val, left=None, right=None):
        self.key = key
        self.val = val
        self.left = left
        self.right = right

```

# AVL Insertion

```
class treeNode:
    def __init__(self, key, val, left=None, right=None):
        self.key = key
        self.val = val
        self.left = left
        self.right = right

```

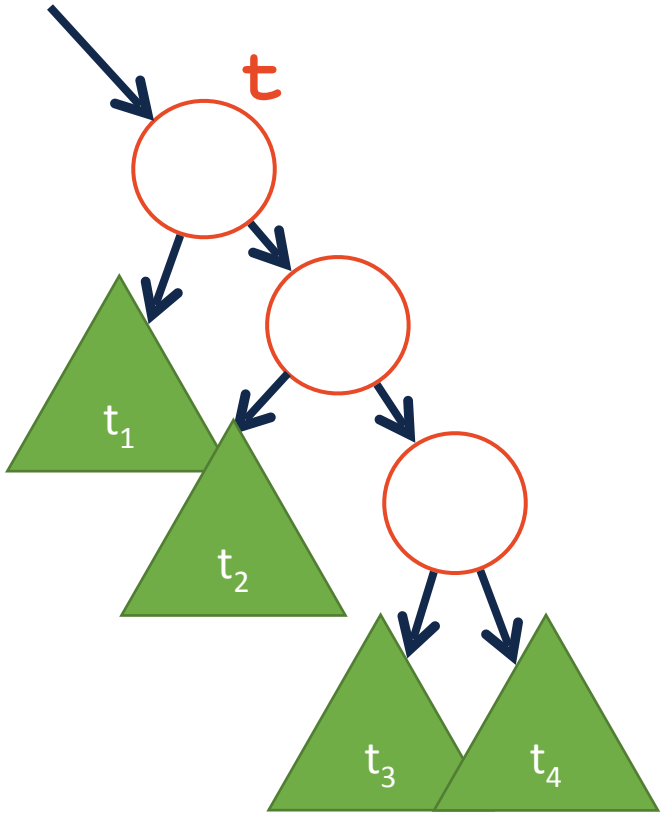
```
def insert(self, key, val):
    self.root = self.insert_helper(self.root, key, val)

def insert_helper(self, node, key, val):
    if node == None:
        return treeNode(key, val)

    if key < node.key:
        node.left = self.insert_helper(node.left, key, val)
    else:
        node.right = self.insert_helper(node.right, key, val)

    return rebalance(node)
```

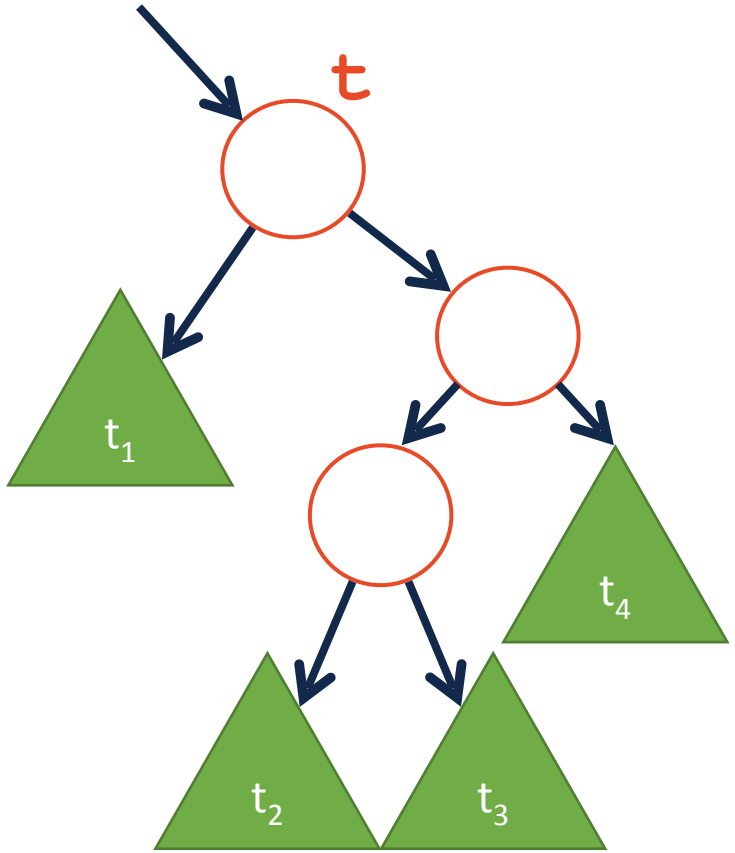
# Rebalancing on insert




**Theorem:**
If an insertion occurred in subtrees $\mathbf{t_3}$ or $\mathbf{t_4}$ and an imbalance was first detected at **t**, then a _____ rotation about **t** restores the balance of the tree.

We gauge this by noting the balance factor of **t->right** is _____.

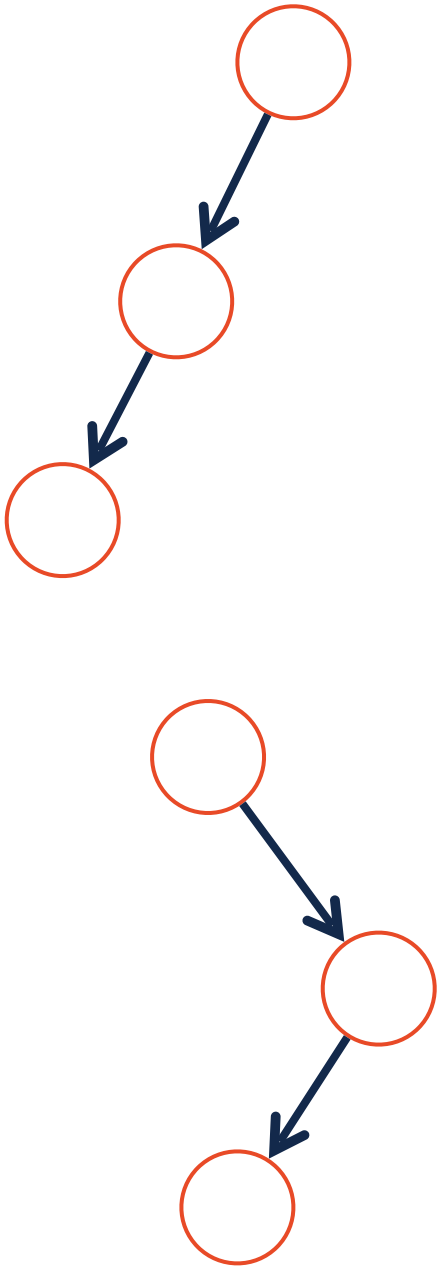
# Rebalancing on insert




**Theorem:**
If an insertion occurred in subtrees $\mathbf{t_2}$ or $\mathbf{t_3}$ and an imbalance was first detected at **t**, then a _____ rotation about **t** restores the balance of the tree.

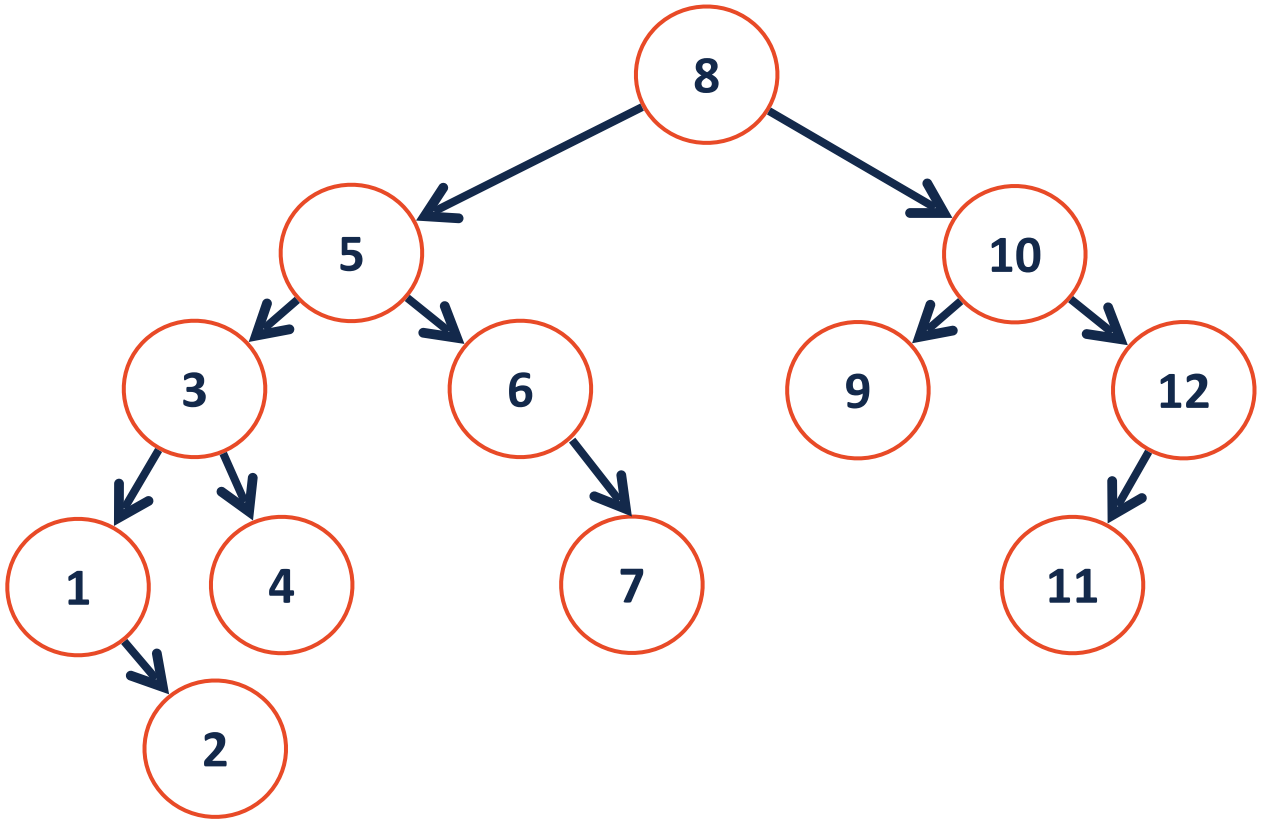
We gauge this by noting the balance factor of **t->right** is _____.

# Rebalancing on insert



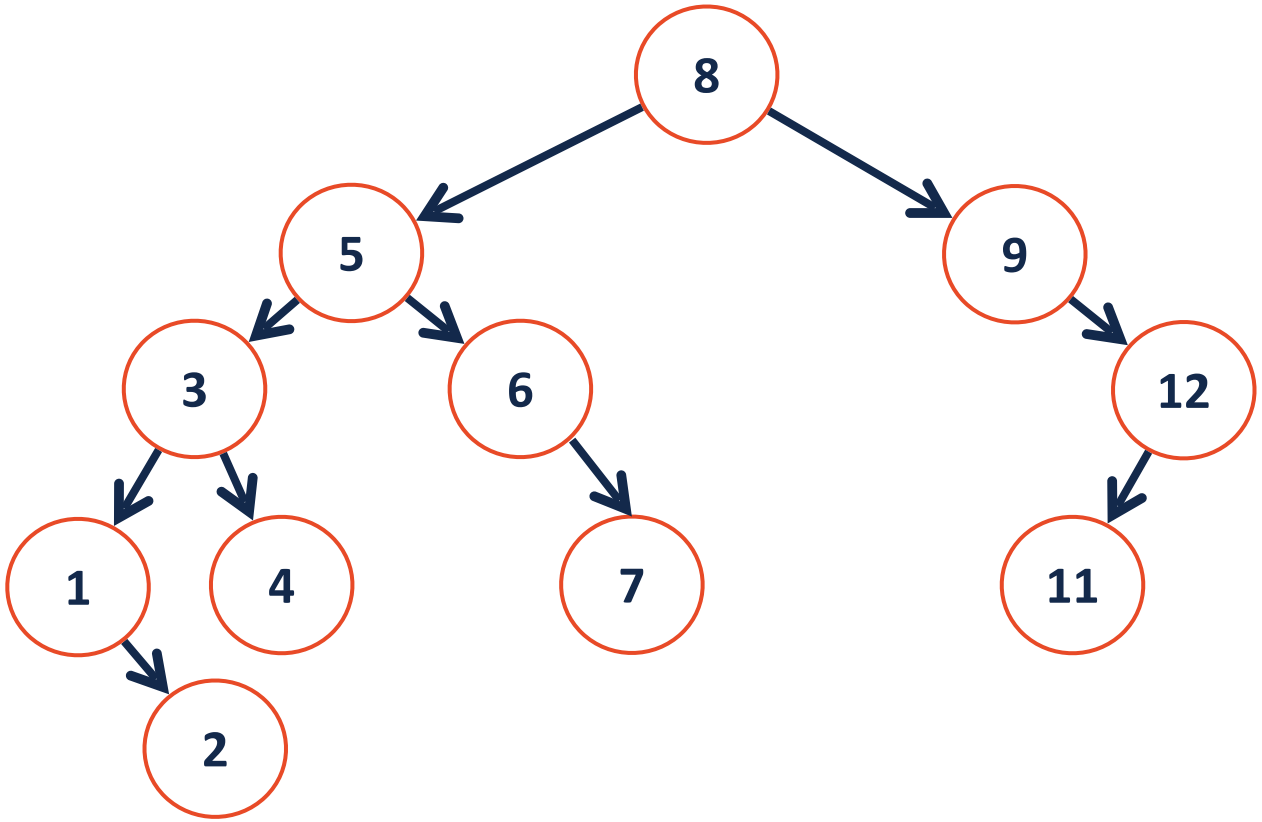
# AVL Remove

`_remove(10)`



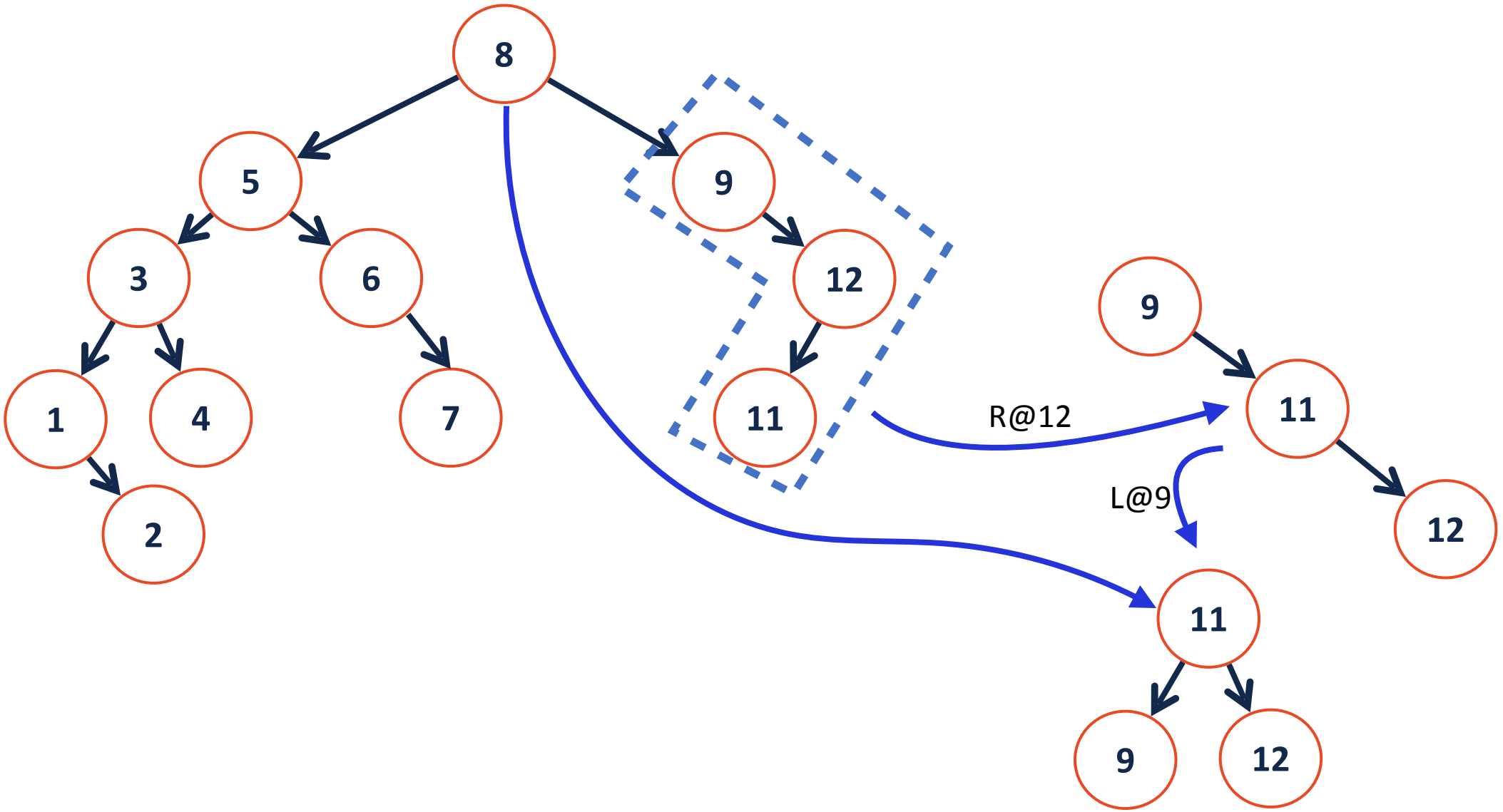
# AVL Remove

`_remove(10)`



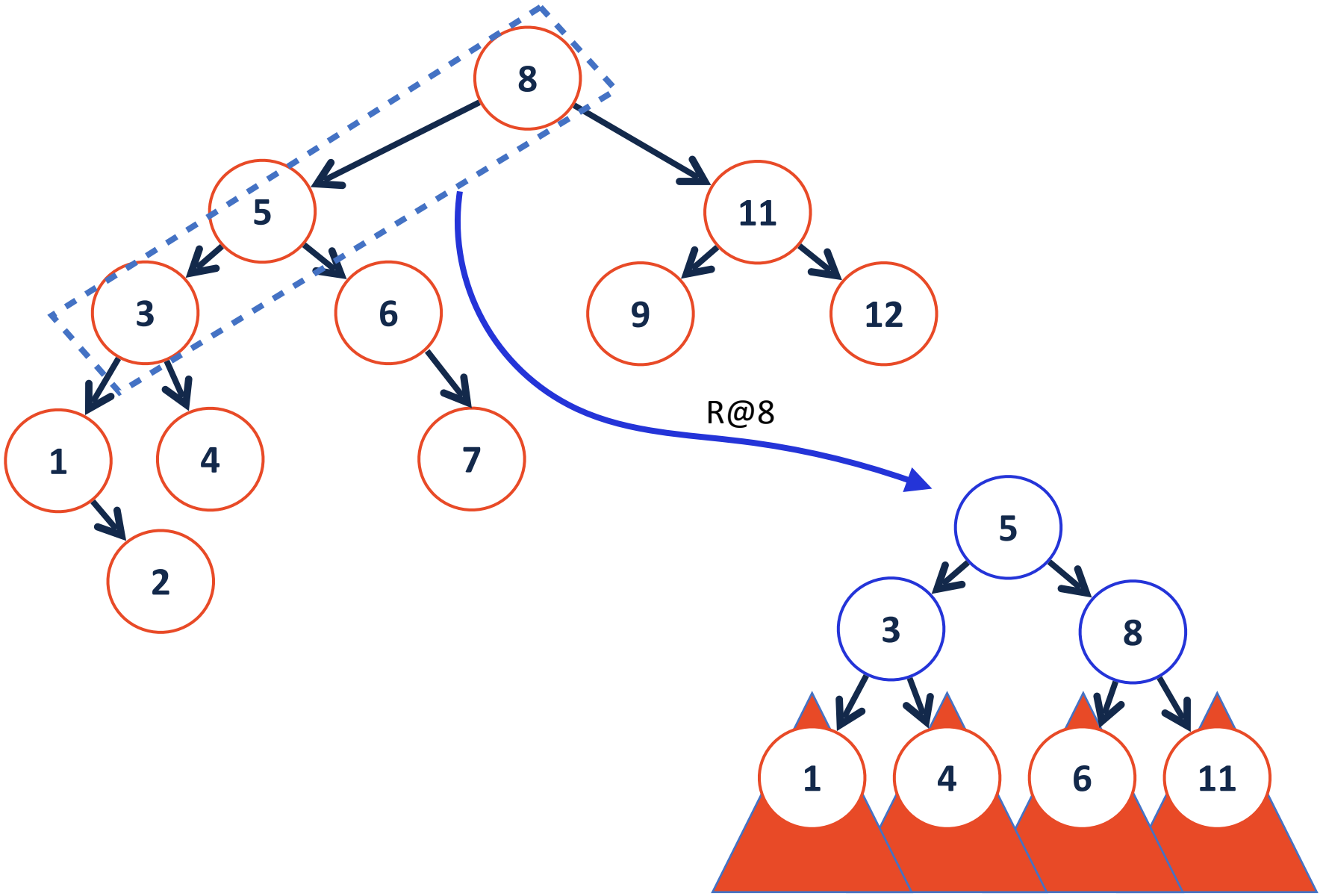
# AVL Remove

`_remove(10)`



# AVL Remove

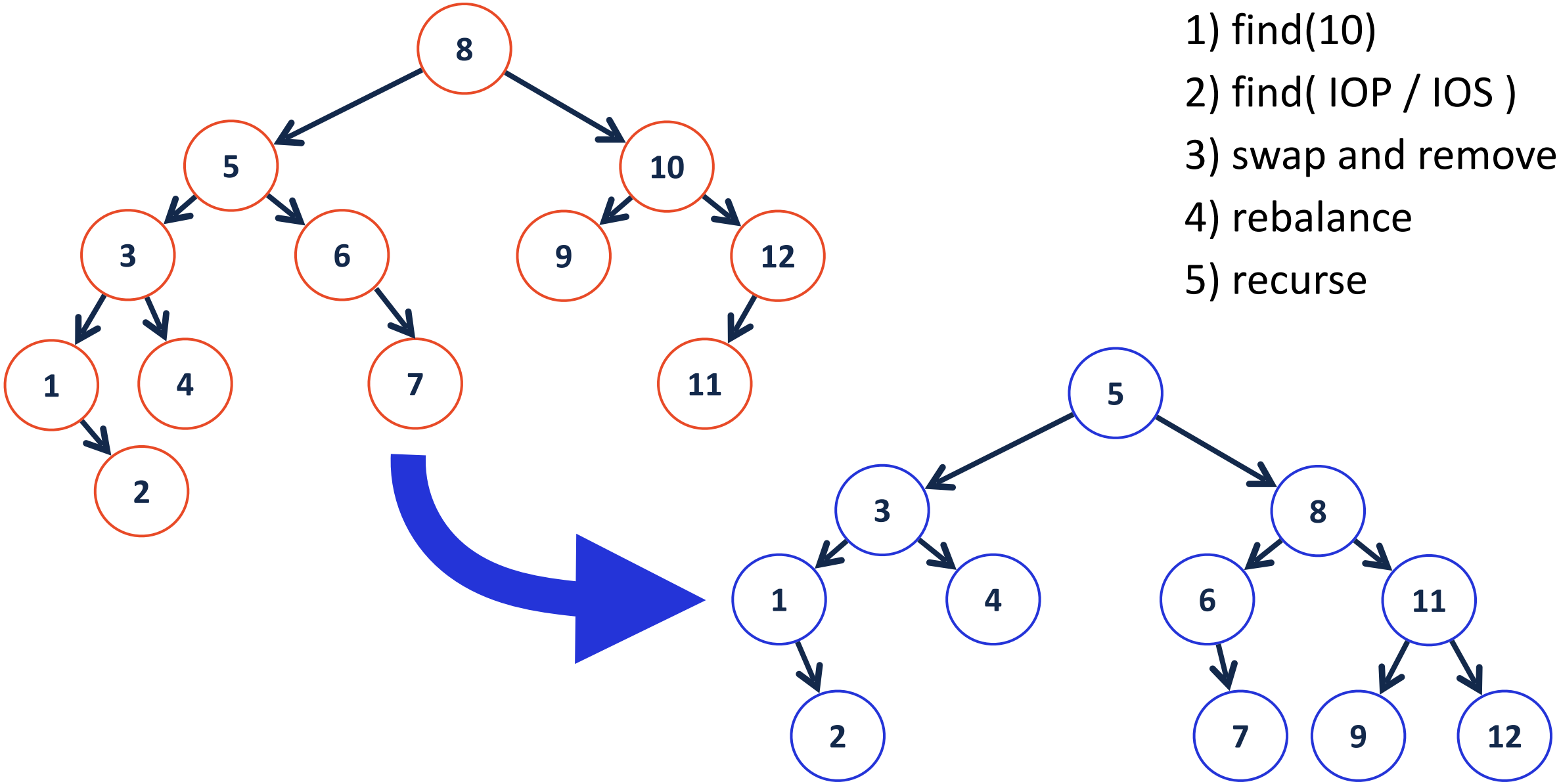
`_remove(10)`



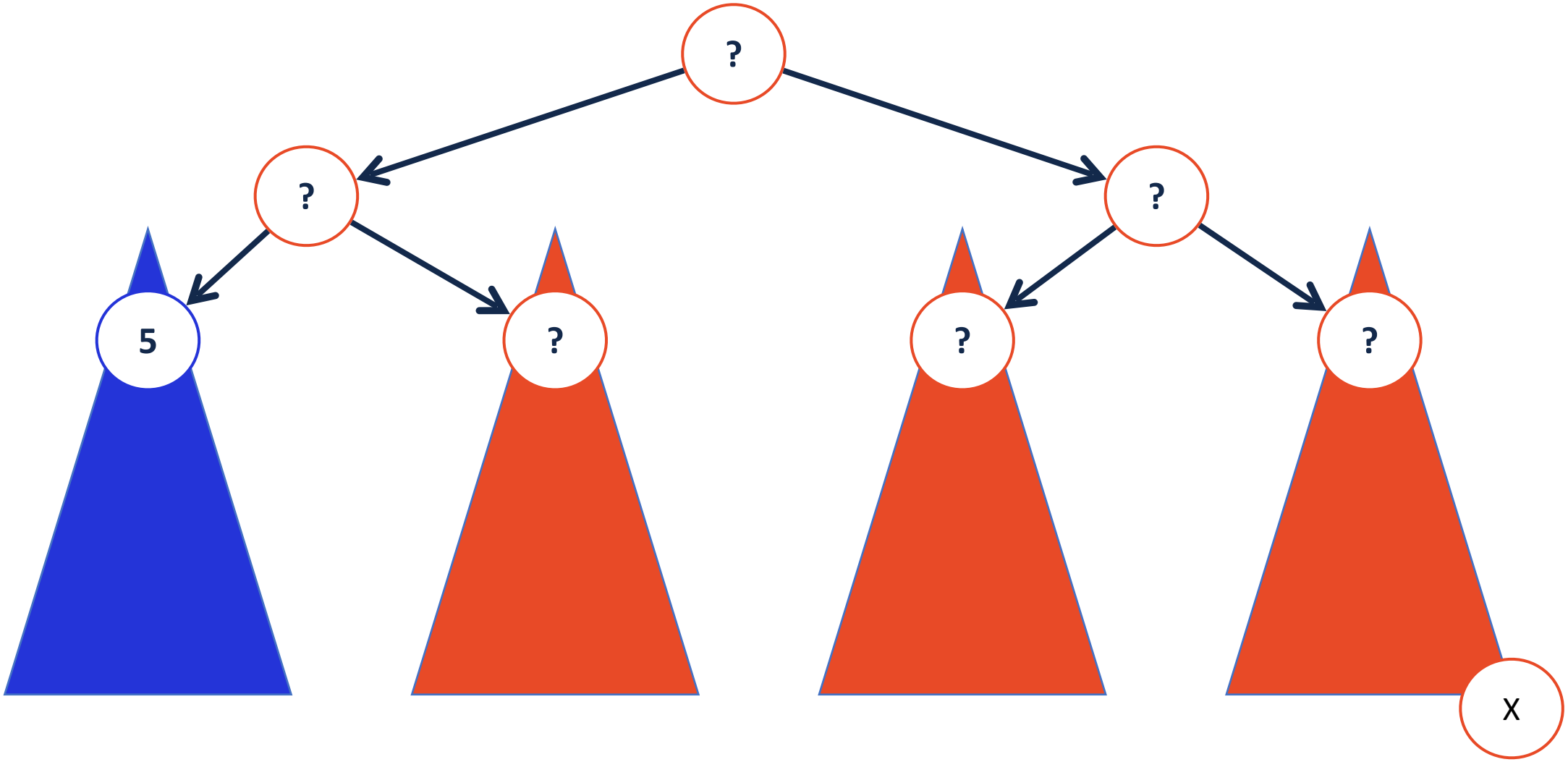
# AVL Remove

`_remove(10)`

1) find(10)
2) find( IOP / IOS )
3) swap and remove
4) rebalance
5) recurse



# AVL Remove



# AVL Tree Analysis

**For AVL tree of height h, we know:**

find runs in: _____.

insert runs in: _____.

remove runs in: _____.

**We will argue that: h** is _____.