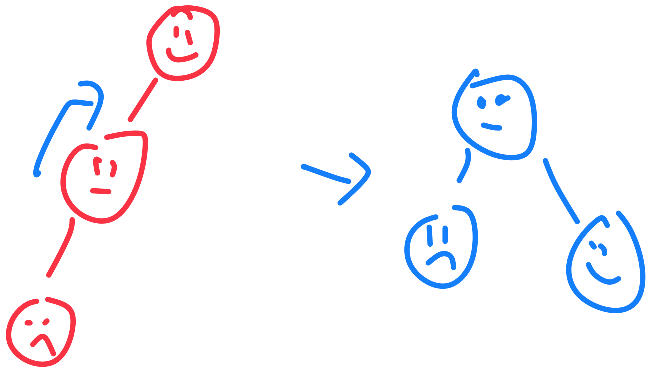
Data Structures
AVL Trees

CS 225
Brad Solomon

September 27, 2024



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Learning Objectives

Review why we need balanced trees

Review what an AVL rotation does

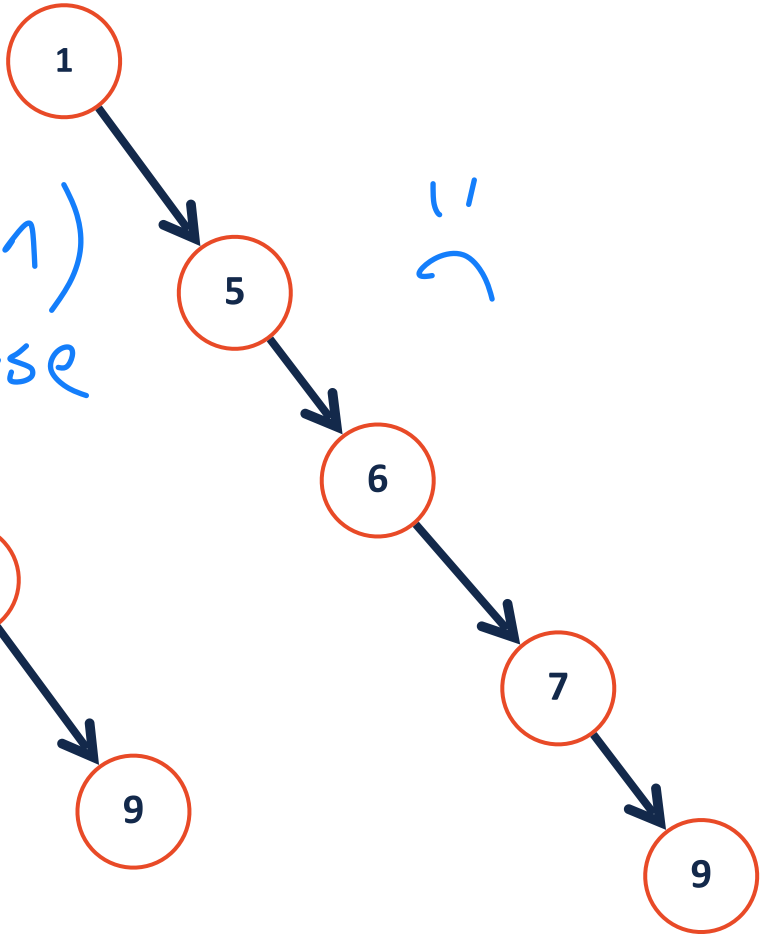
Review the four possible rotations for an AVL tree

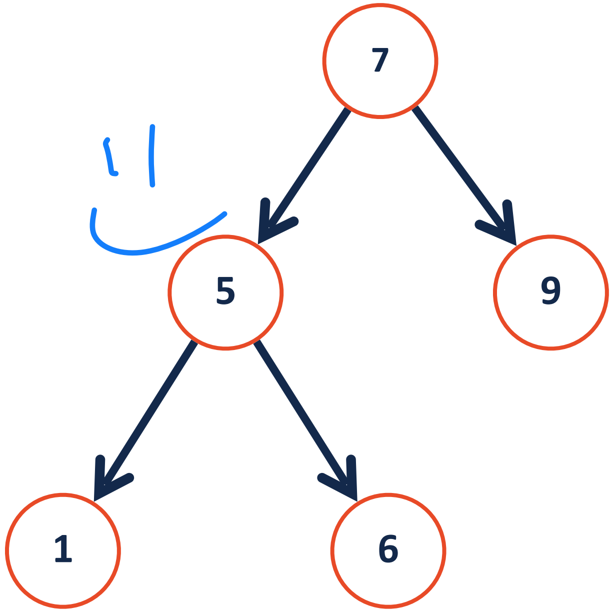
Explore the implementation of AVL Tree

BST Analysis – Running Time

	BST Worst Case
find	O(h)
insert	O(h)
delete	O(h)
traverse	O(n)

$O(h) \sim O(n)$
worst case

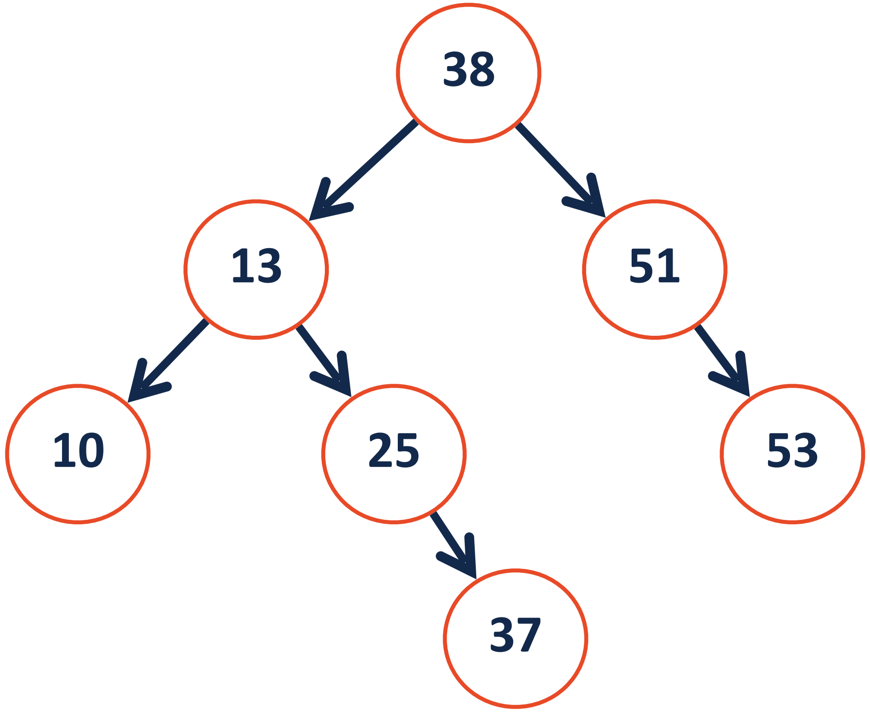




AVL-Tree: A self-balancing binary search tree

Every node in an AVL tree has a balance of:

$B = height(t_R) - height(t_L)$

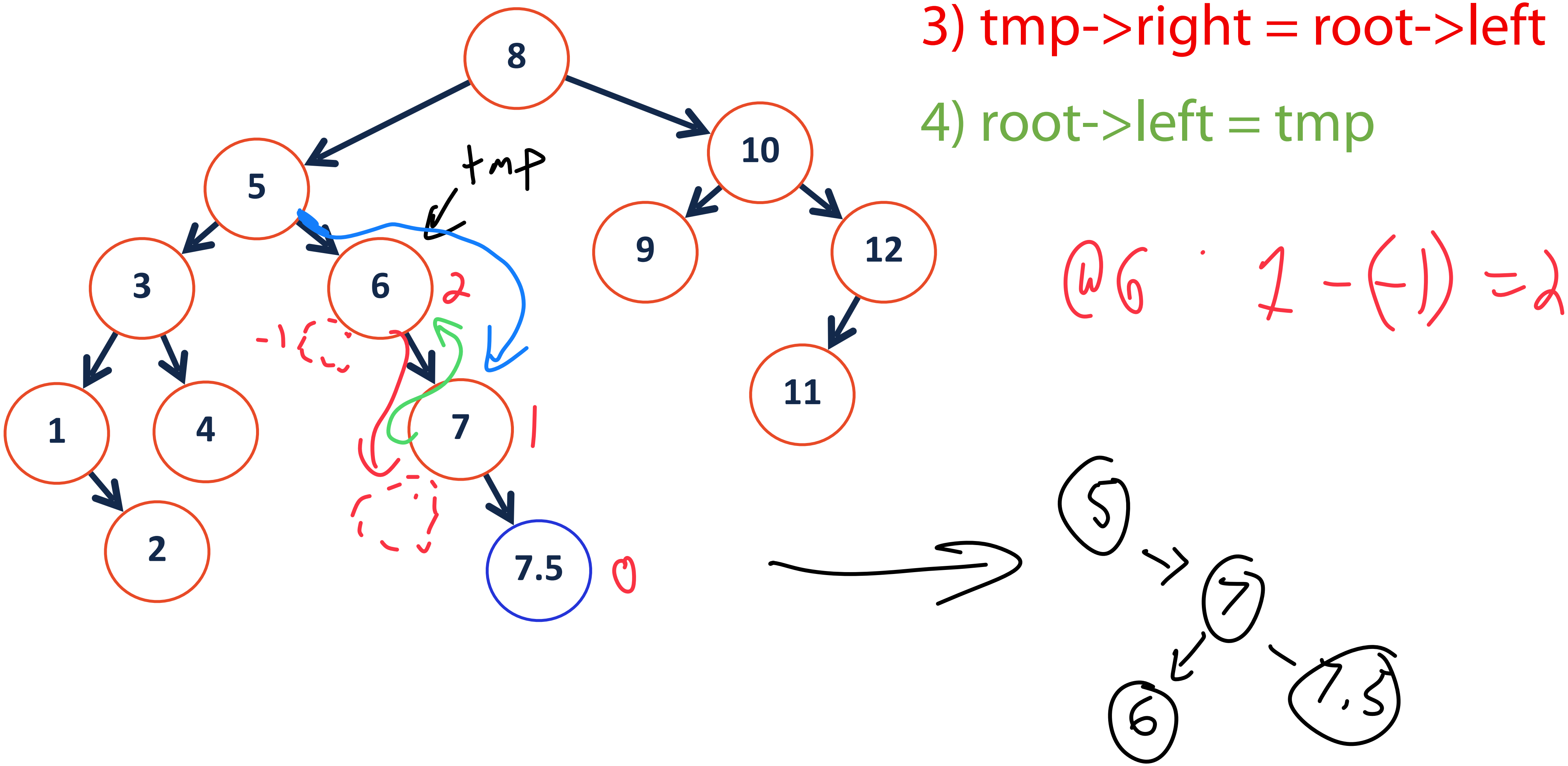


-1, 0, 1 are balanced

2, -2 not balanced

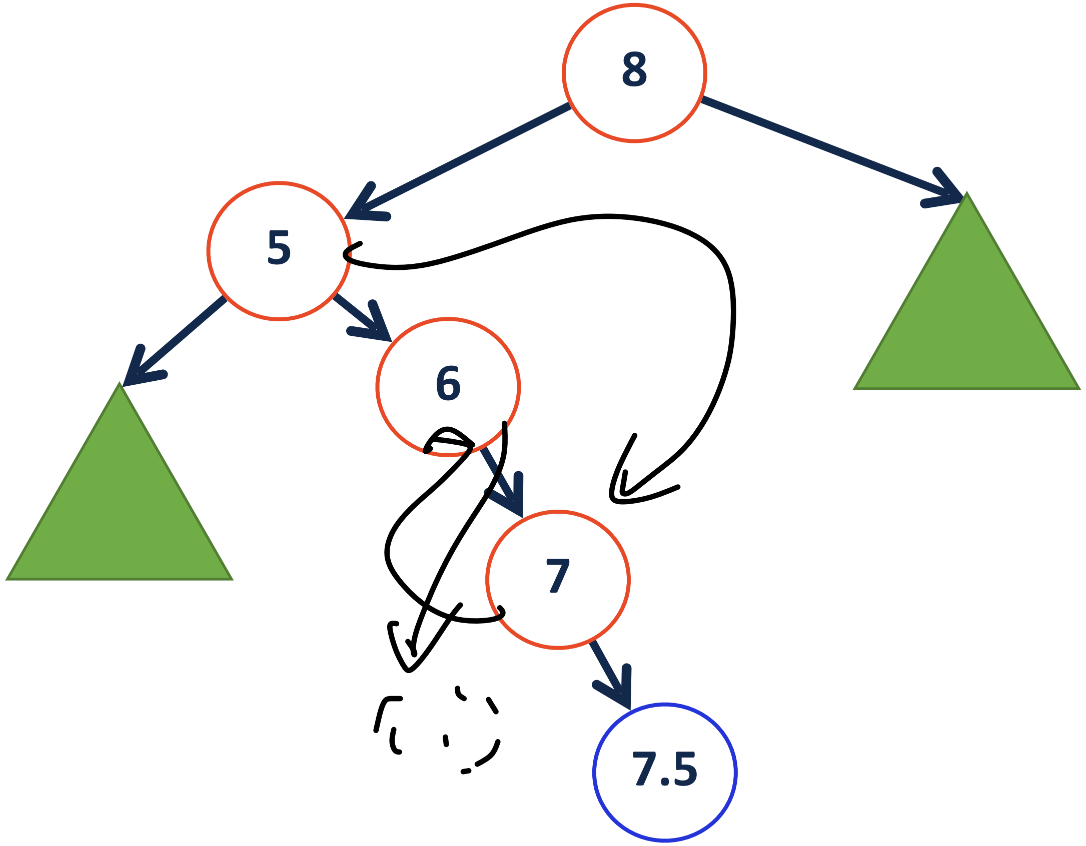
Left Rotation

1) Create a tmp pointer to root
2) Update root to point to mid
3) tmp->right = root->left
4) root->left = tmp



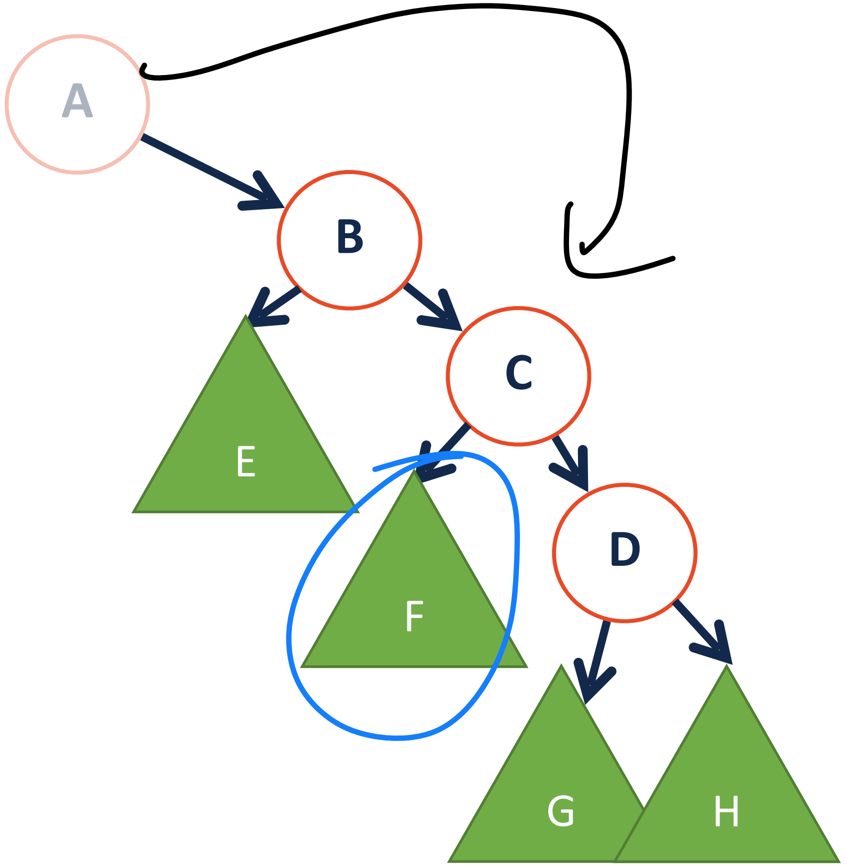
Left Rotation

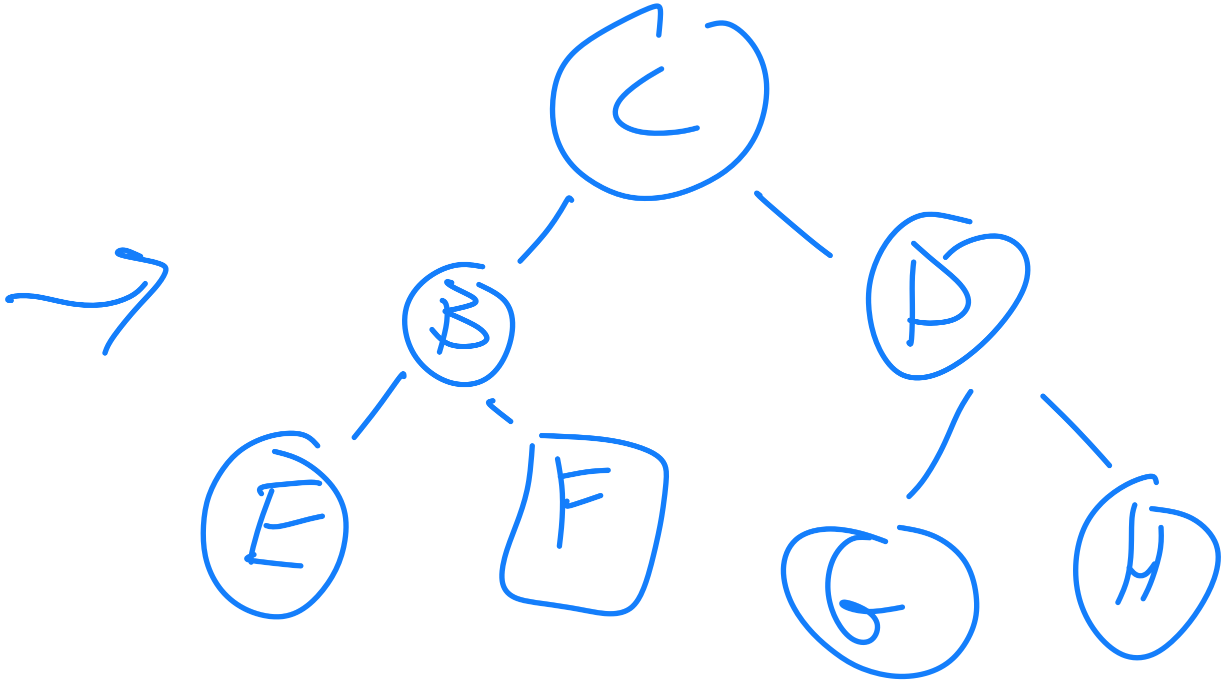
All rotations are local (subtrees are not impacted)



Left Rotation

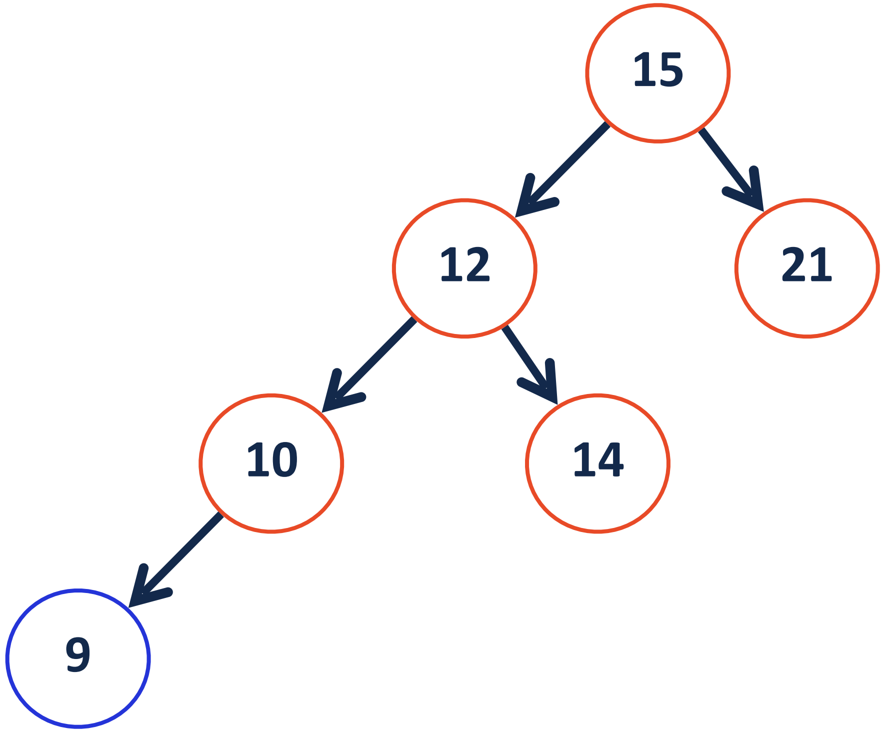
All rotations preserve BST property





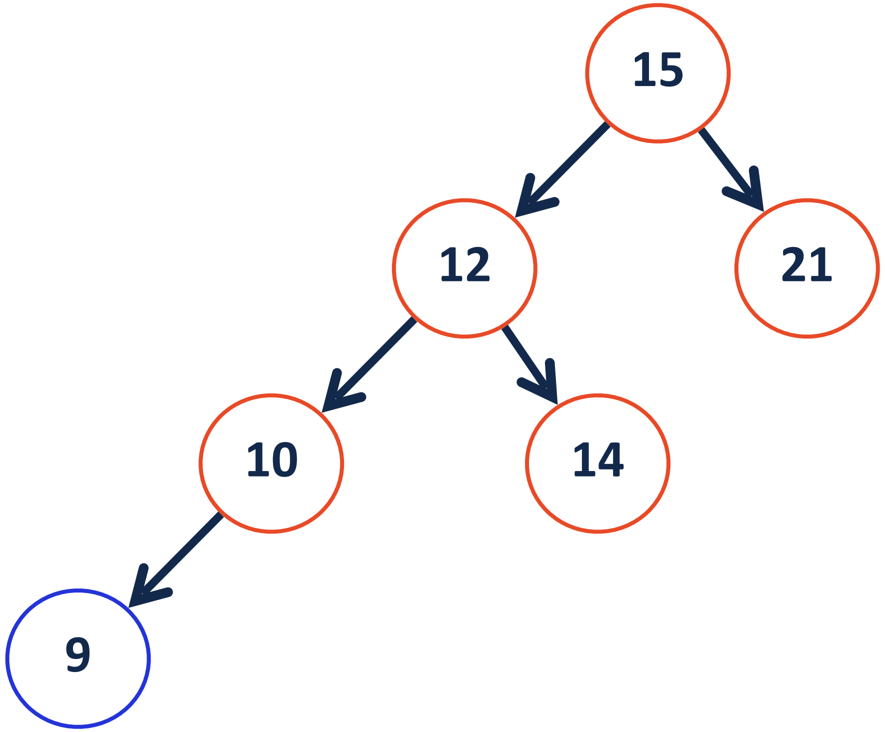
$B < F < C$

Right Rotation

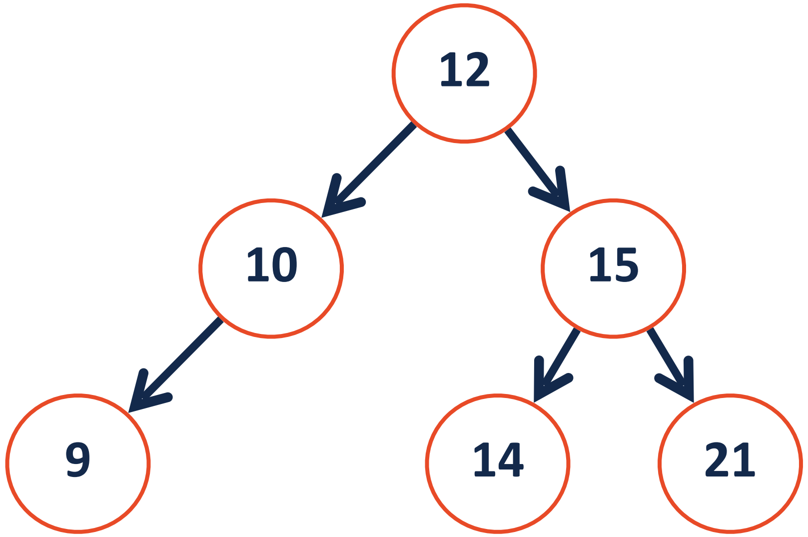


1) Create a tmp pointer to root
2) Update root to point to mid
3) tmp->left = root->right
4) root->right = tmp

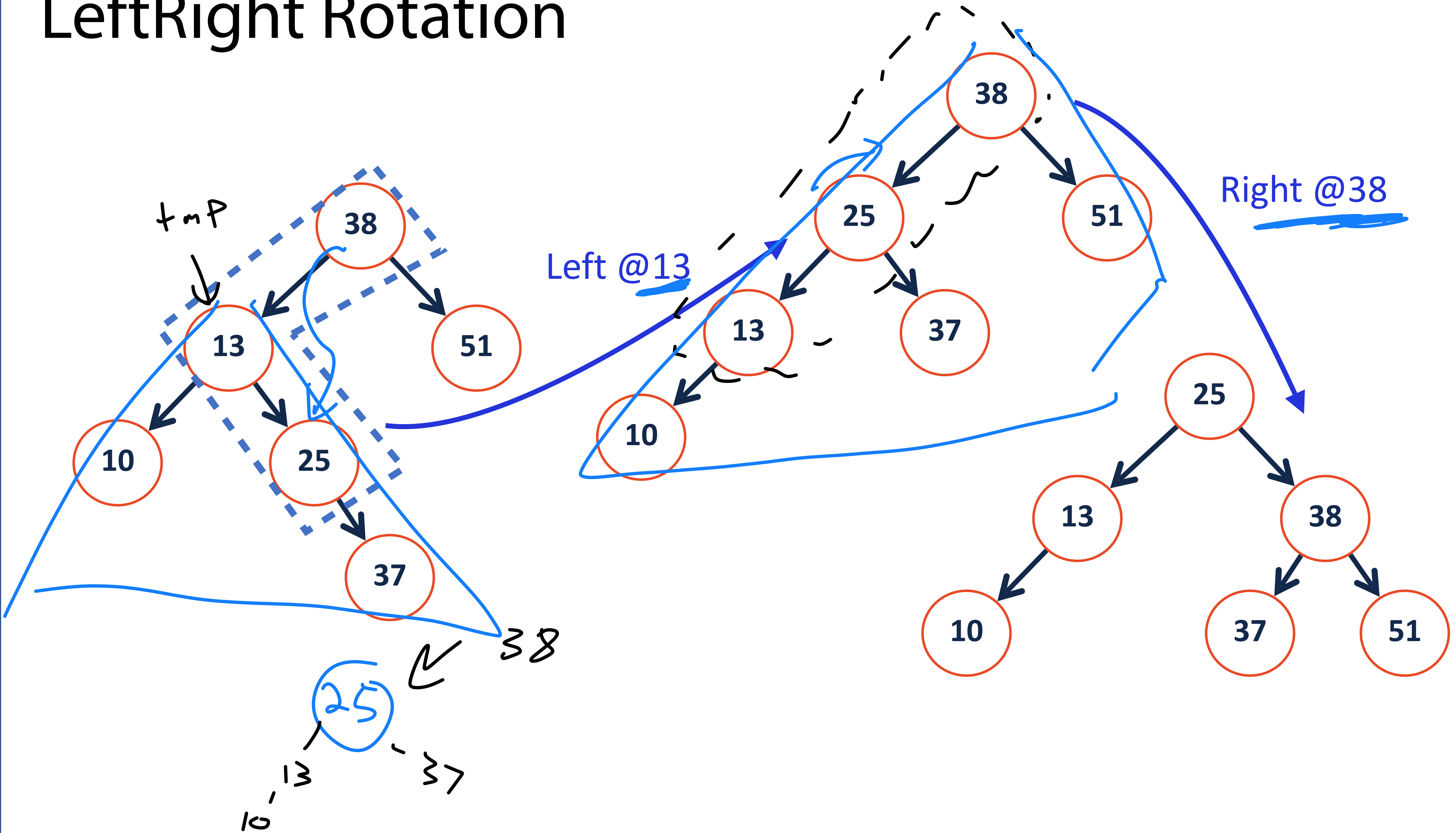
Right Rotation



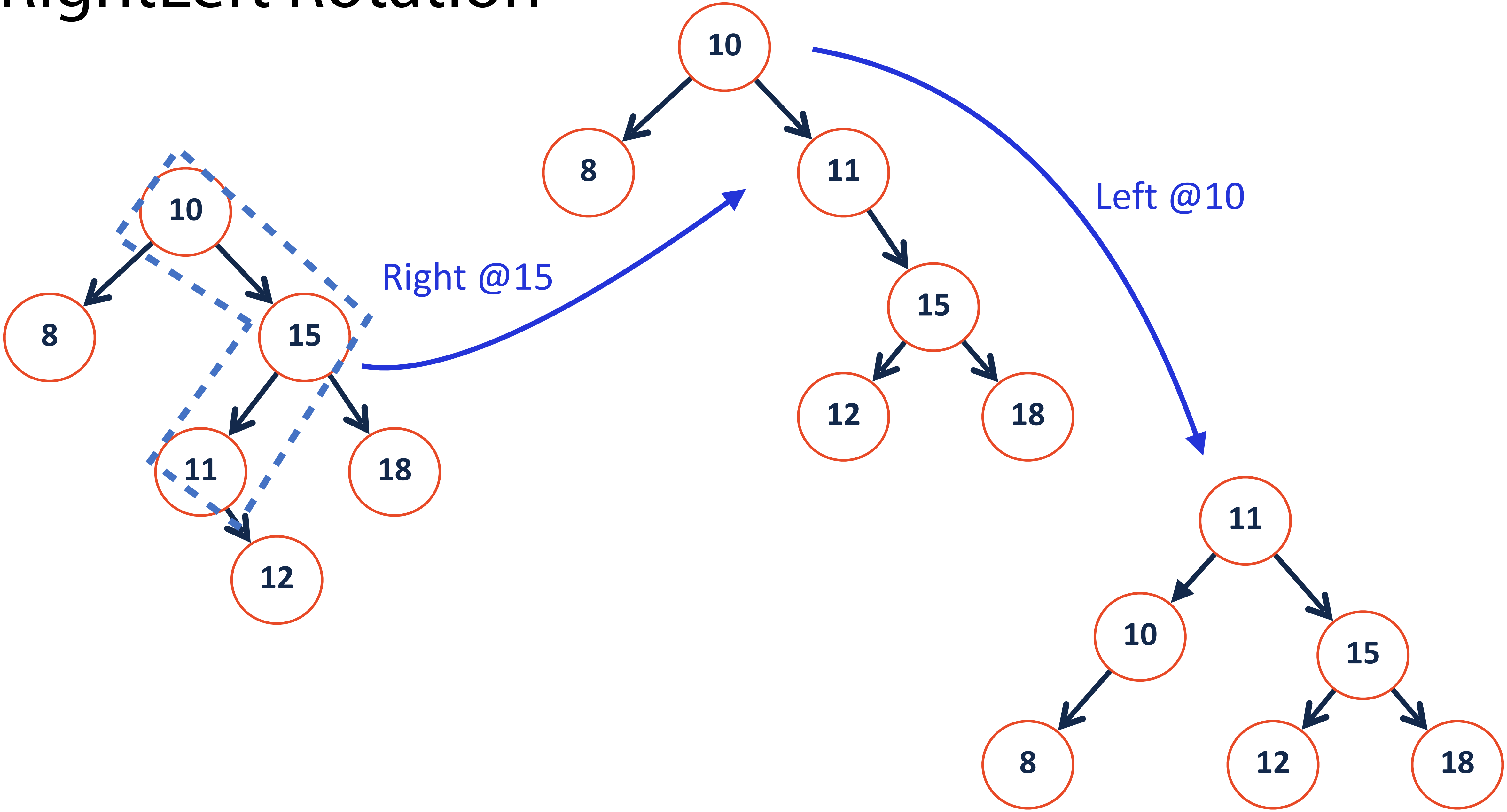
1) Create a tmp pointer to root
2) Update root to point to mid
3) tmp->left = root->right
4) root->right = tmp



LeftRight Rotation



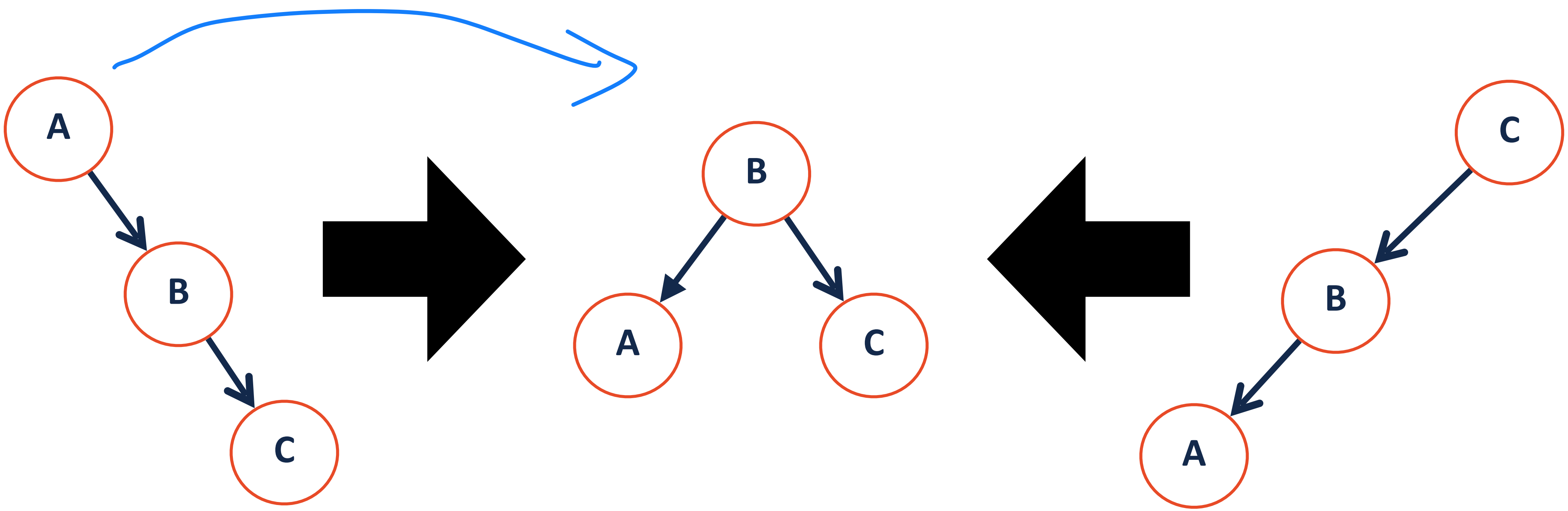
RightLeft Rotation



AVL Rotations

$A < B < C$

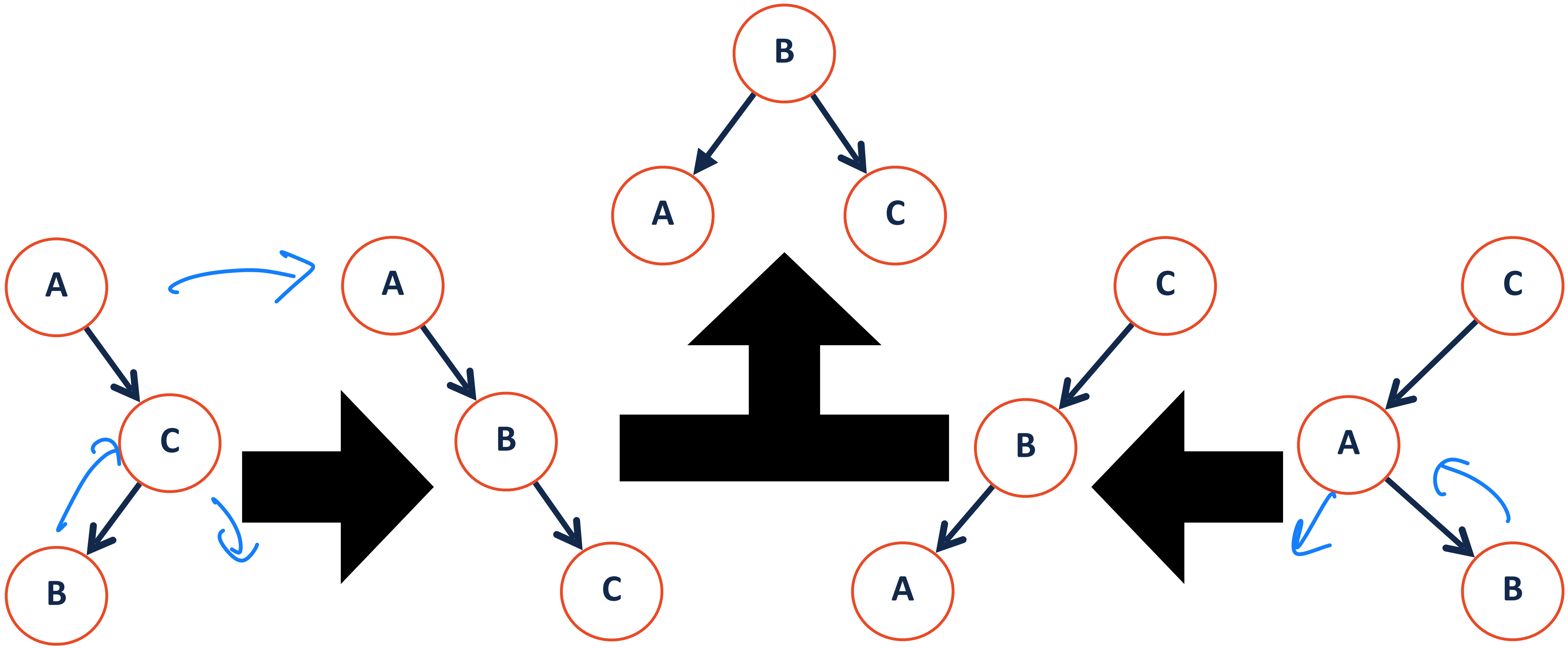
Left and right rotation convert **sticks** into **mountains**



Height one smaller BST

AVL Rotations

LeftRight (RightLeft) convert **elbows** into **sticks** into **mountains**



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 tree will be balanced

↳ This will make height bounded by log(n)

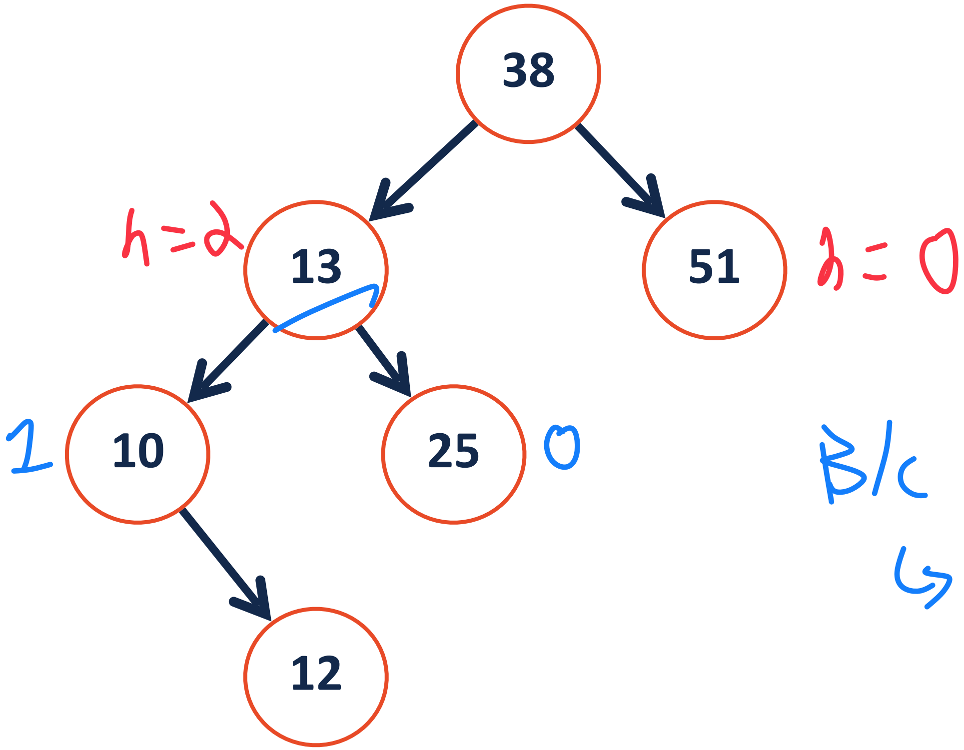
AVL Rotations

↙ a 'stick'

We can identify which rotation to do using **balance**

$B@38 = 0 - 2 = \boxed{-2}$ → Right

↳ Left imbalanced

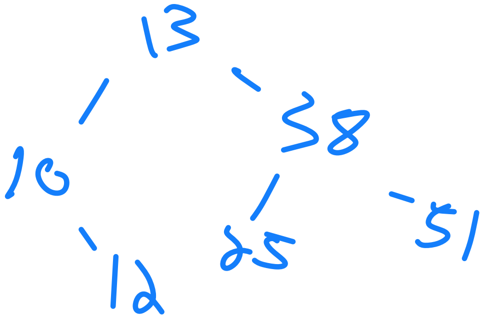


B/c imba is negative

↳ Look at left child

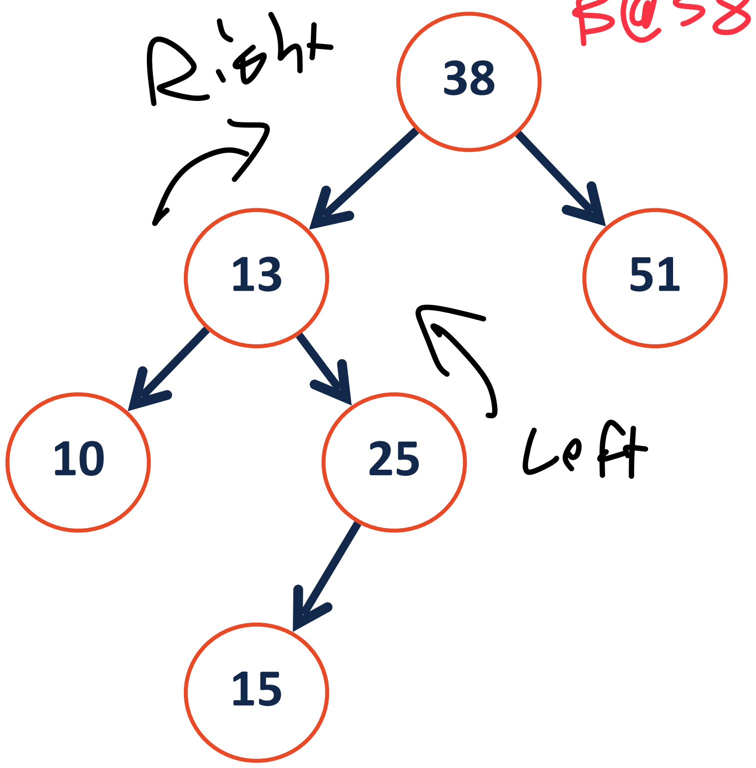
↳ Balance of left child

↳ $B@13 = 0 - 1 = \boxed{-1}$



AVL Rotations

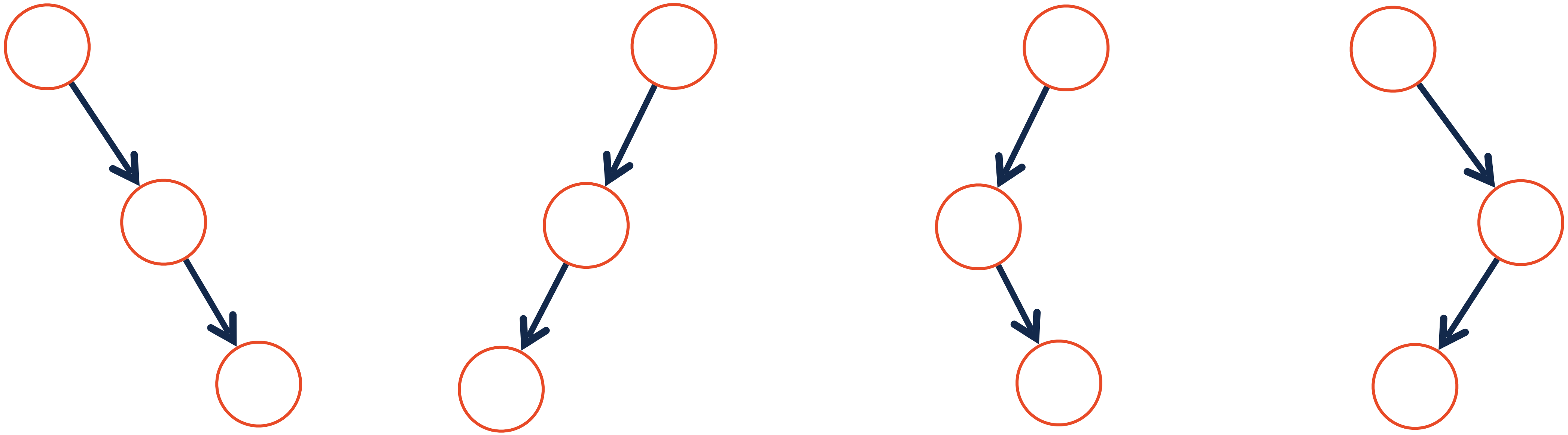
We can identify which rotation to do using **balance**



$B@38 = 0 - 2 = \boxed{-2}$

$B@13 = 1 - 0 = \boxed{1}$

AVL Rotations



	Left	Right	LeftRight	RightLeft
Root Balance:	2	-2	-2	2
Child Balance:	1	-1	1	-1

AVL Rotation Practice

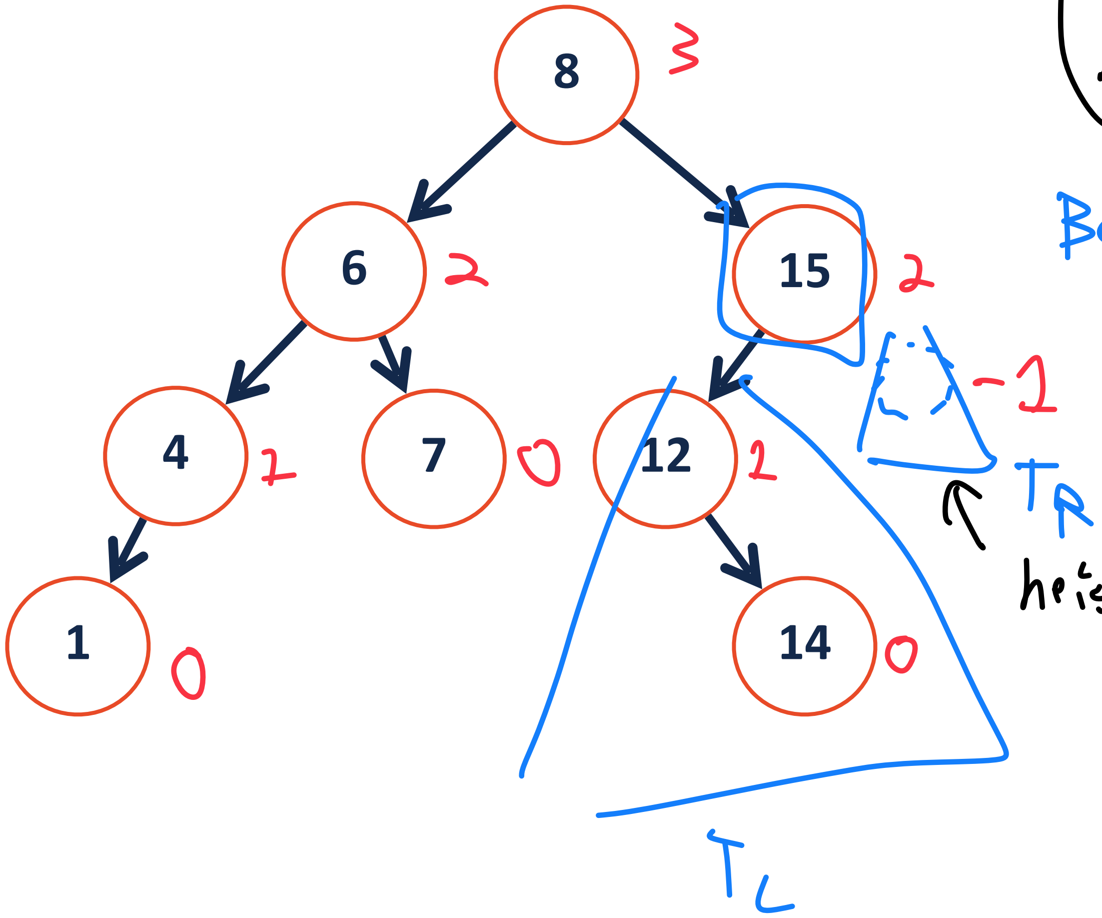
$$H(\text{Root}) = \max\left[H(T_L), H(T_R)\right] + 1$$

R $H = 0$

Base case: $H = -1$

Monday!

Balance @ 15 = $-1 - 1 = \underline{-2}$



height of empty tree is -1

AVL vs BST ADT

BT → BST → AVL

all nodes 'sorted'

The AVL tree is a modified binary search tree that rotates **when necessary**

```
struct TreeNode {
  T key;
  unsigned height;
  TreeNode *left;
  TreeNode *right;
};
```

* Tradeoff!

1

Add cost to store height

Gain O(1) height "calc"

How does the constraint on balance affect the core functions?

Find

↳ Every op must update height (if needed)

Insert

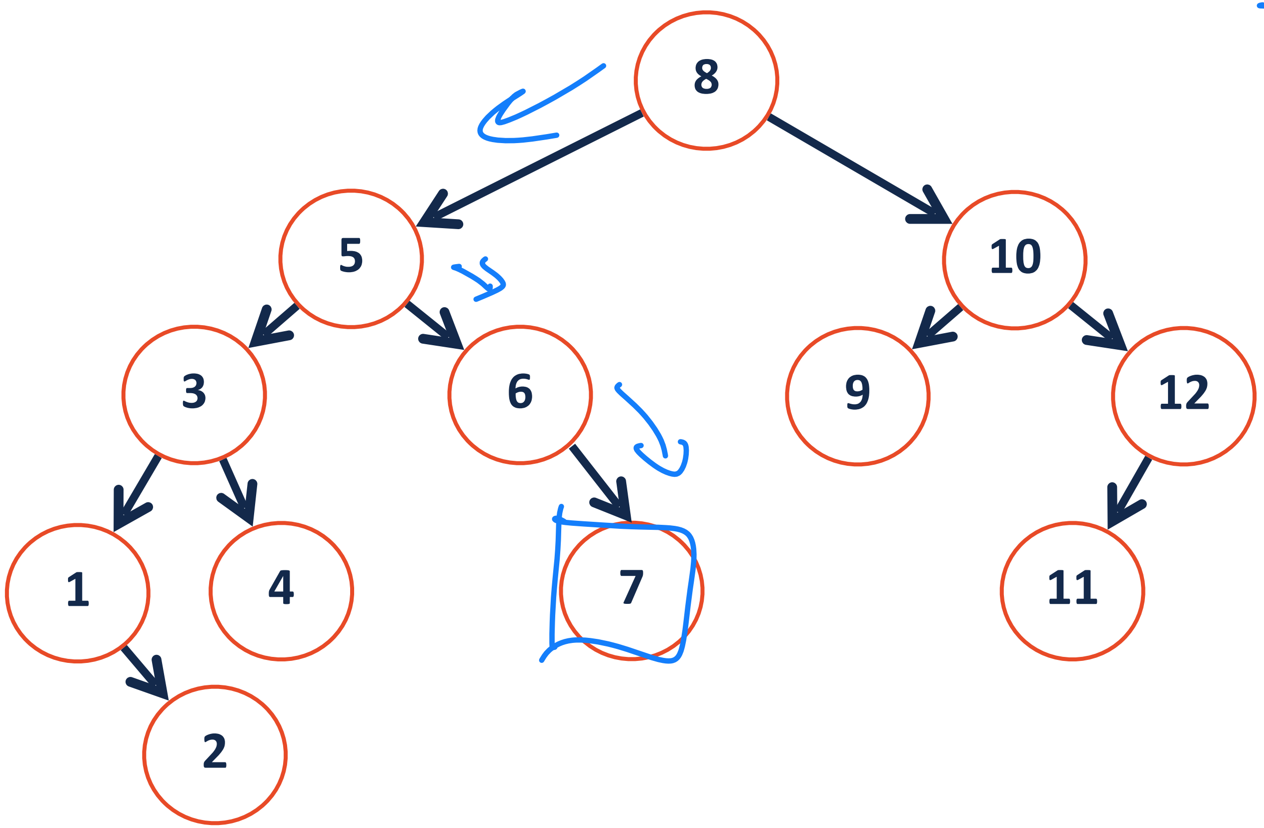
2 ↳ Every op must check balance (if needed)

Remove

AVL Find

`_find(7)`

7 < 8



height is bounded :)

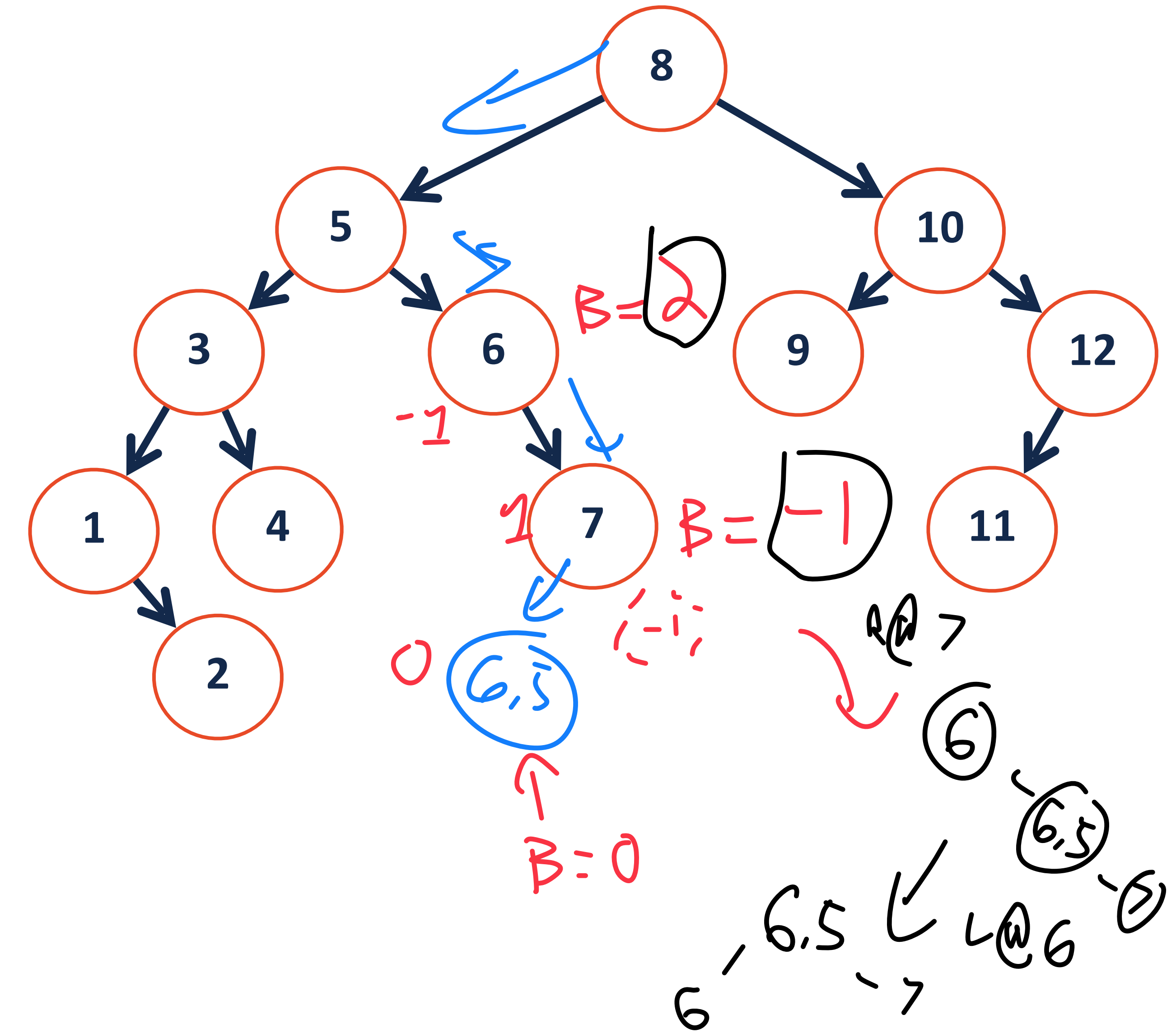
No difference in implementation

BST ⇔ AVL

AVL Insertion

`_insert(6.5)`

1) Find node & insert in place
2) Check for imbalance
3) Rotate if necessary
4) update height



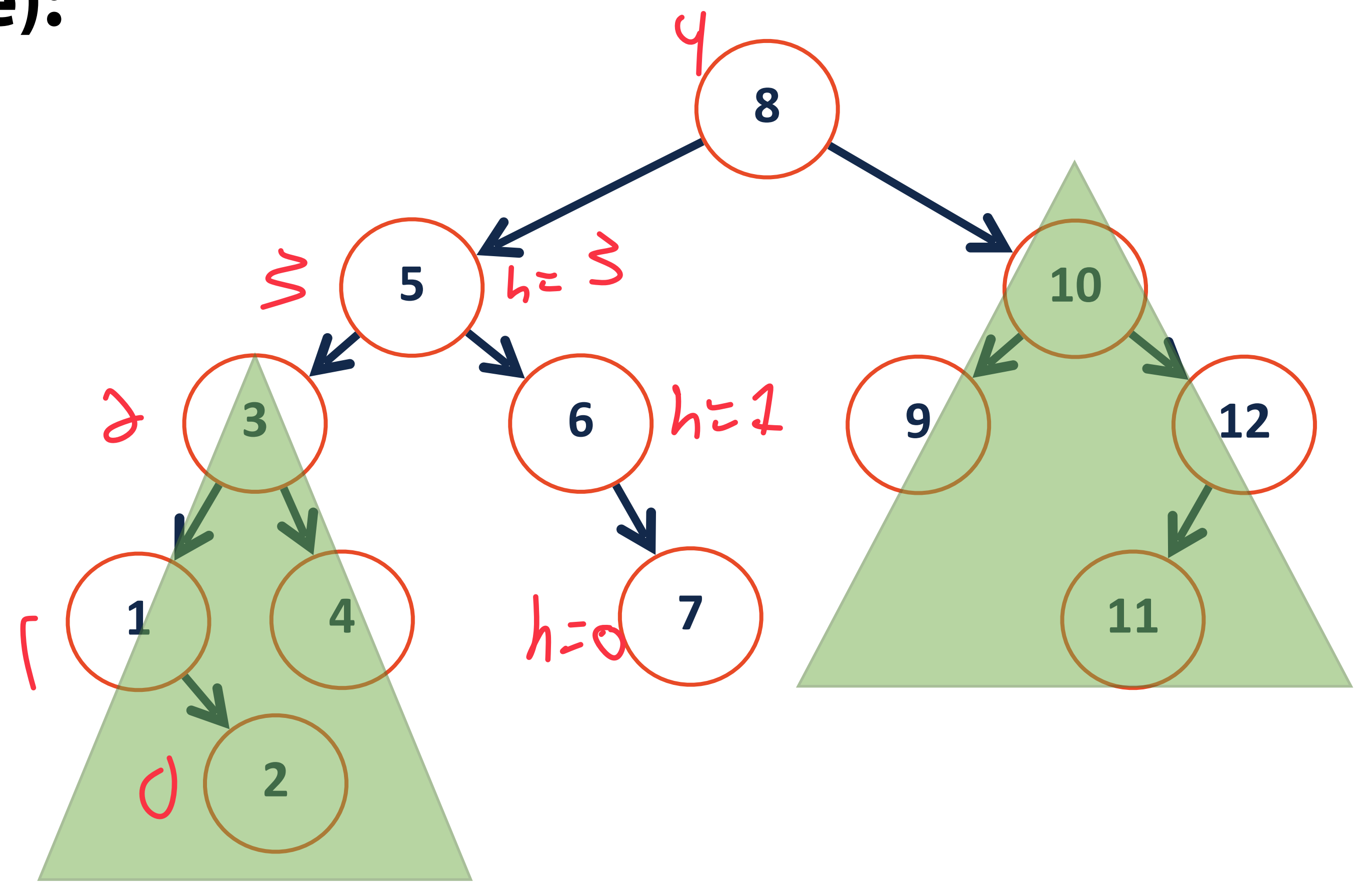

```
struct TreeNode {
  T key;
  unsigned height;
  TreeNode *left;
  TreeNode *right;
};
```

AVL Insertion

`_insert(6.5)`

Insert (recursive pseudocode):

1. Insert at proper place
2. Check for imbalance
3. Rotate, if necessary
4. Update height




```
struct TreeNode {
  T key;
  unsigned height;
  TreeNode *left;
  TreeNode *right;
};
```

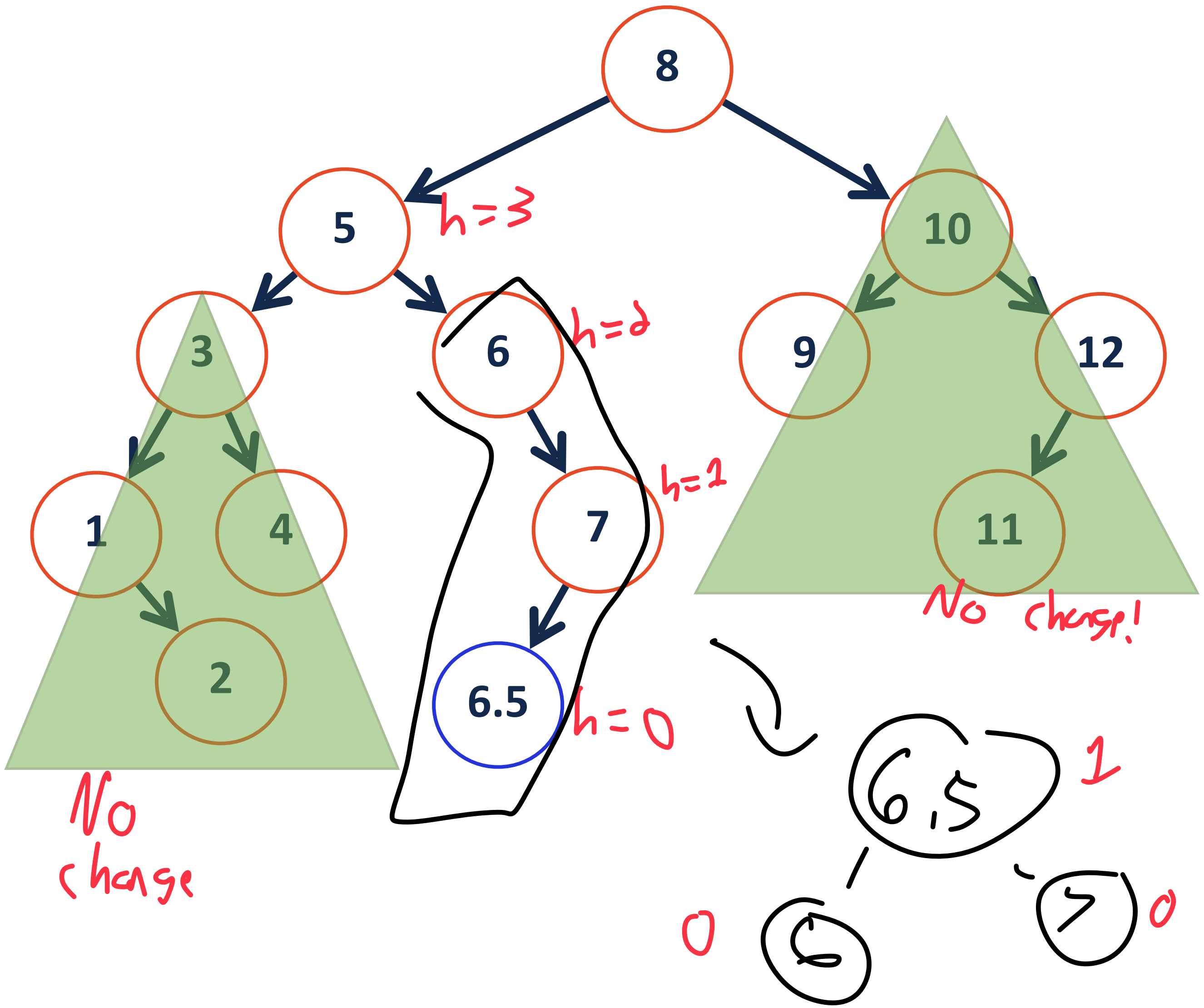
AVL Insertion

`_insert(6.5)`

Insert (recursive pseudocode):

1. Insert at proper place
2. Check for imbalance
3. Rotate, if necessary
4. Update height

```
struct TreeNode {
  T key;
  unsigned height;
  TreeNode *left;
  TreeNode *right;
};
```



```
template <typename K, typename V>
void AVL<K, D>::_insert(const K & key, const V & data, TreeNode *& cur) {
  if (cur == NULL)           { cur = new TreeNode(key, data);    }
  else if (key < cur->key) { _insert( key, data, cur->left ); }
  else if (key > cur->key) { _insert( key, data, cur->right );}
  _ensureBalance(cur);
}
```

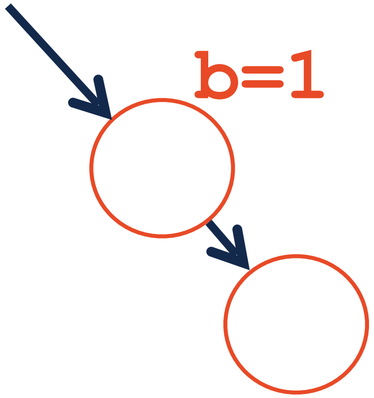
```
template <typename K, typename V>
void AVL<K, D>::_ensureBalance(TreeNode *& cur) {
  // Calculate the balance factor:
  int balance = height(cur->right) - height(cur->left);

  // Check if the node is current not in balance:
  if ( balance == -2 ) { // Right
    int l_balance =
          height(cur->left->right) - height(cur->left->left);
    if ( l_balance == -1 ) { ______rotateRight()______; }
    else        -2, 2      { ______rotateLeftRight()______; }
  } else if ( balance == 2 ) { // Left
    int r_balance =
          height(cur->right->right) - height(cur->right->left);
    if( r_balance == 1 ) { ______rotateLeft()______; }
    else                 { ______rotateRightLeft()______; }
  }

  _updateHeight(cur);  *
};
```

AVL Insertion

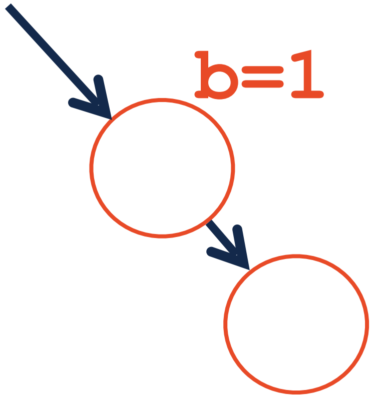
Given an AVL is balanced, insert can create **at most** one imbalance

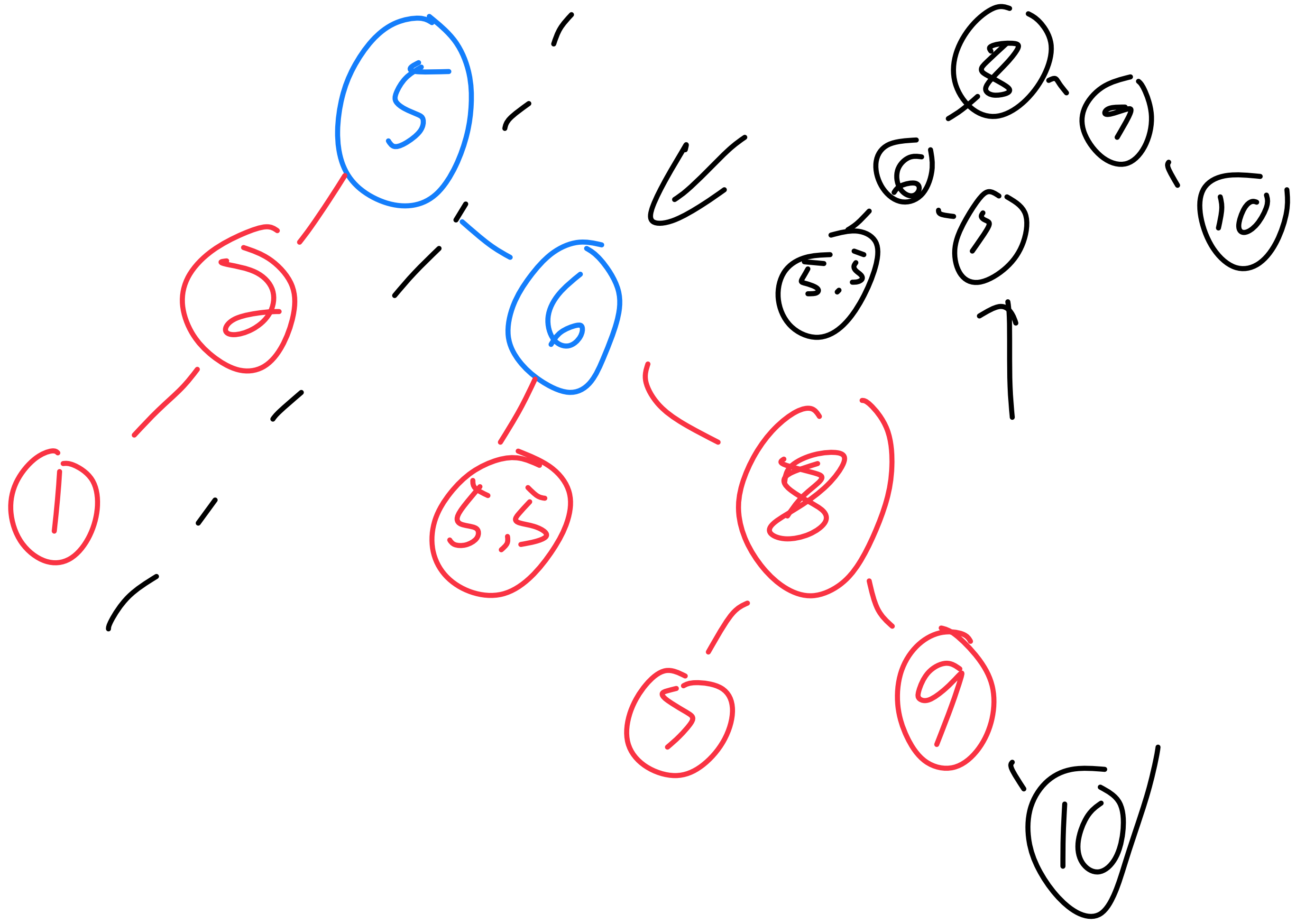


AVL Insertion

Post-class question (Specific example)

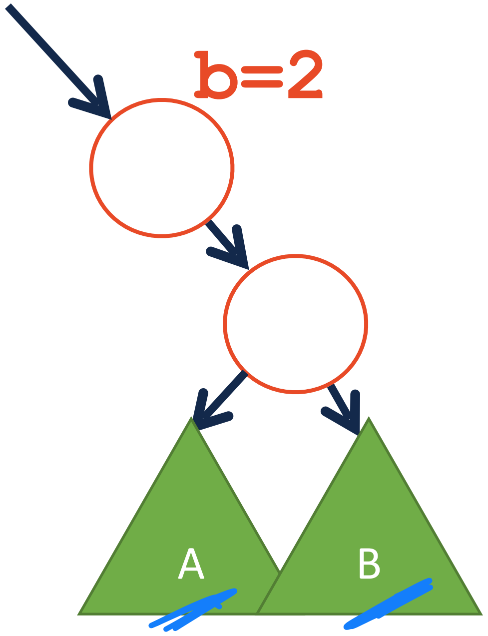
Given an AVL is balanced, insert can create **at most** one imbalance





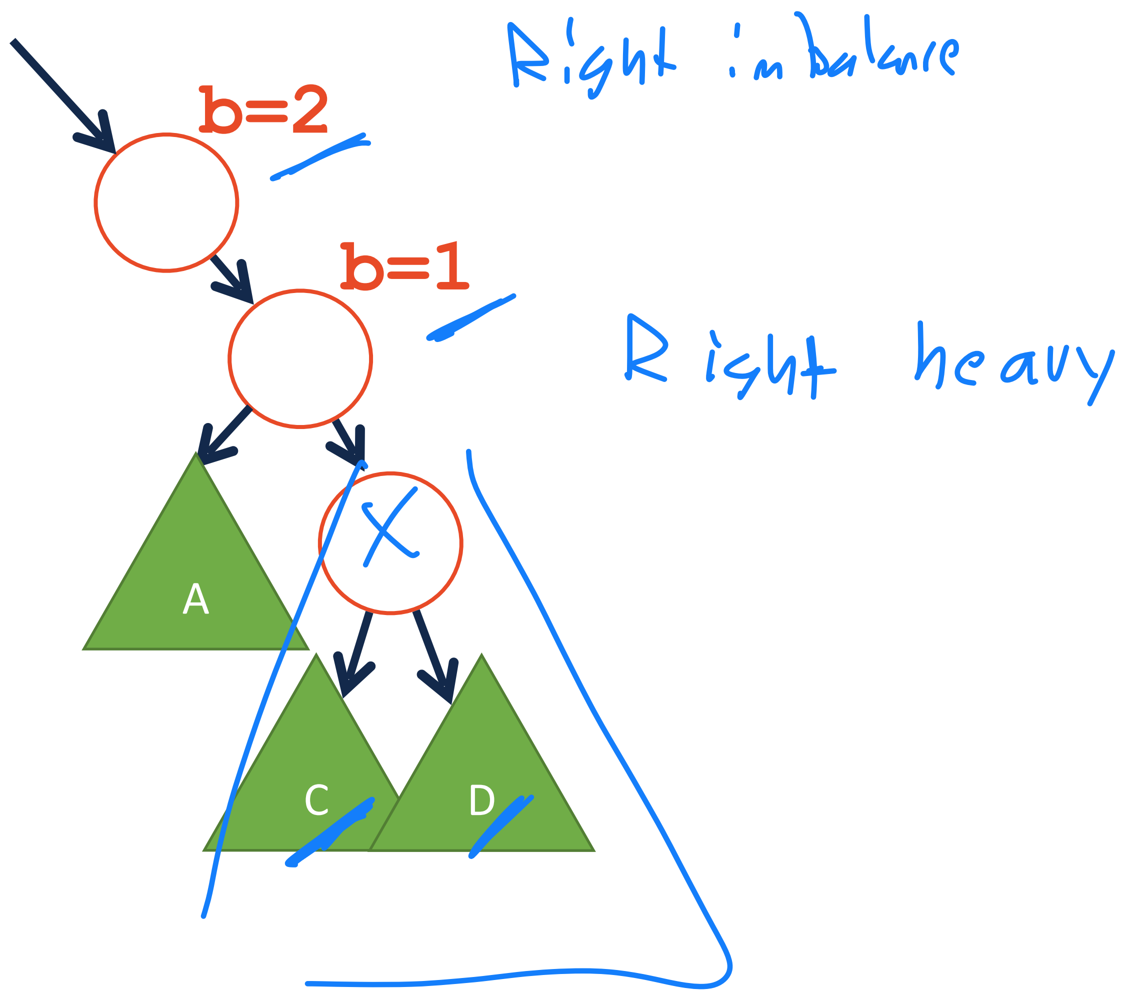
AVL Insertion

Given an AVL is balanced, insert can create **at most** one imbalance



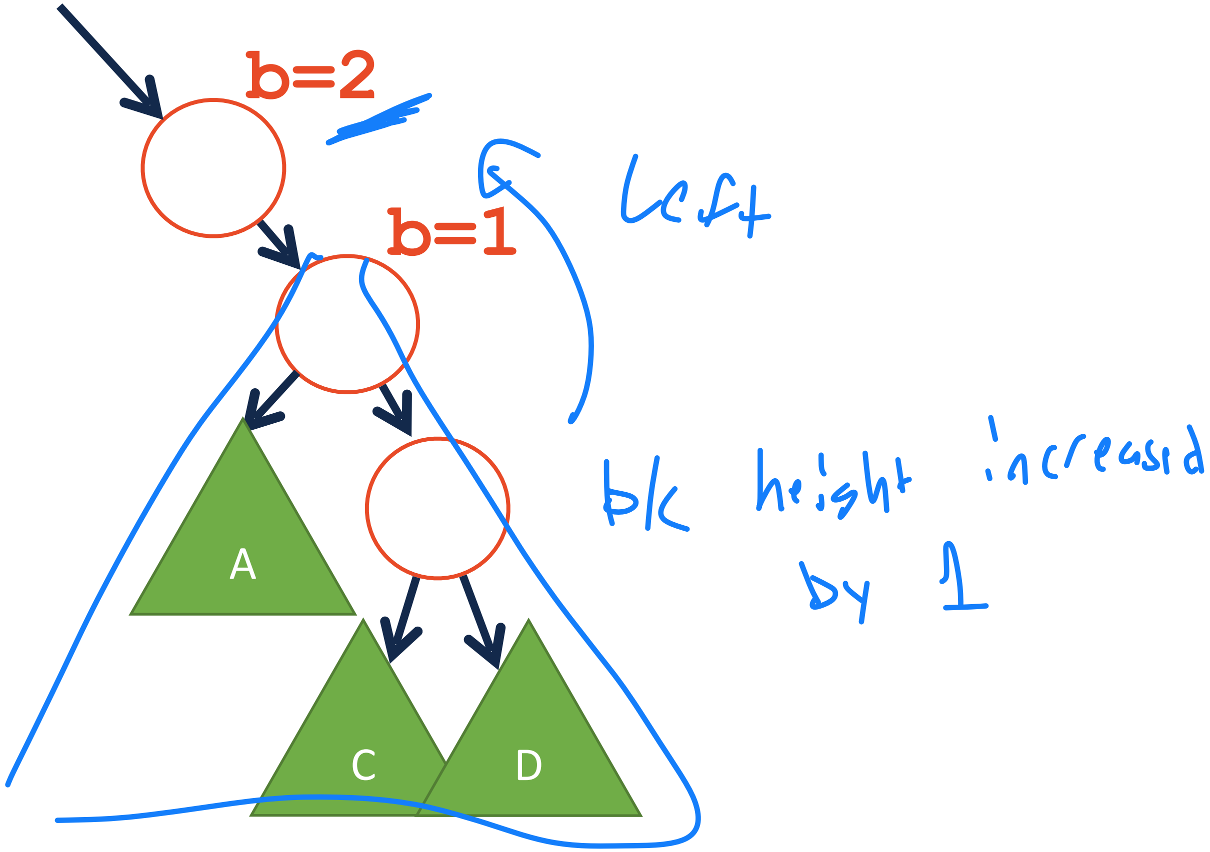
AVL Insertion

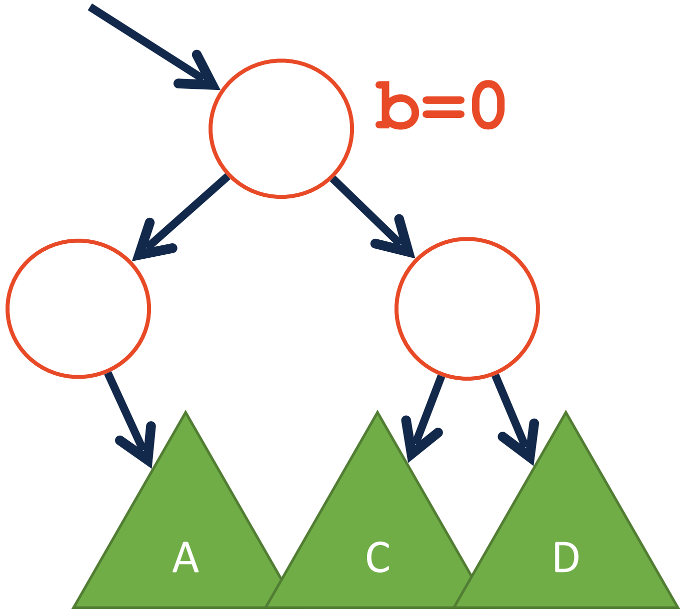
If we insert in B, I must have a balance pattern of **2, 1**



AVL Insertion

A **left** rotation fixes our imbalance in our local tree.

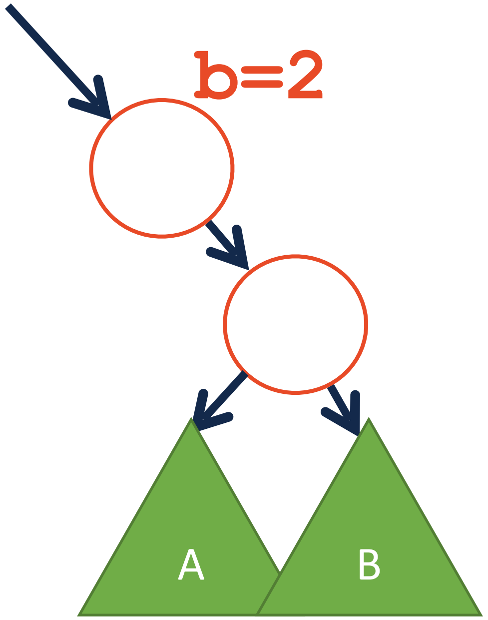




After rotation, subtree has **pre-insert height**. (Overall tree is balanced)

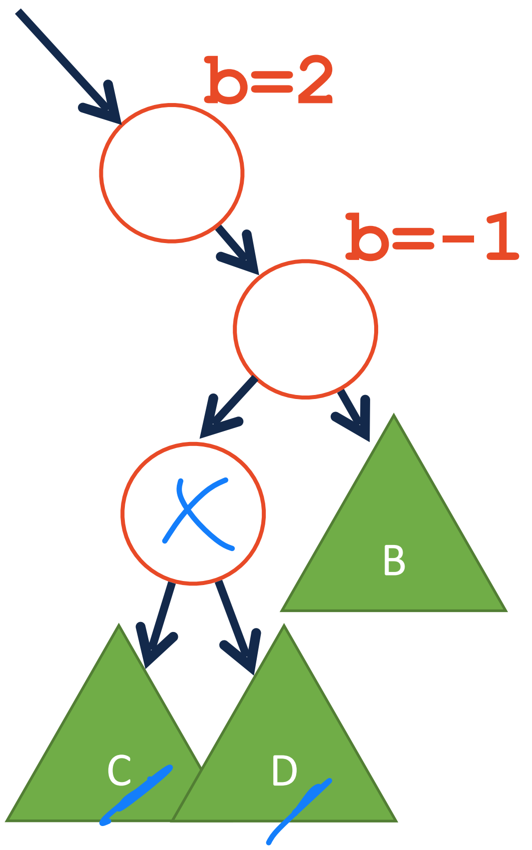
AVL Insertion

If we insert in A, I must have a balance pattern of **2, -1**



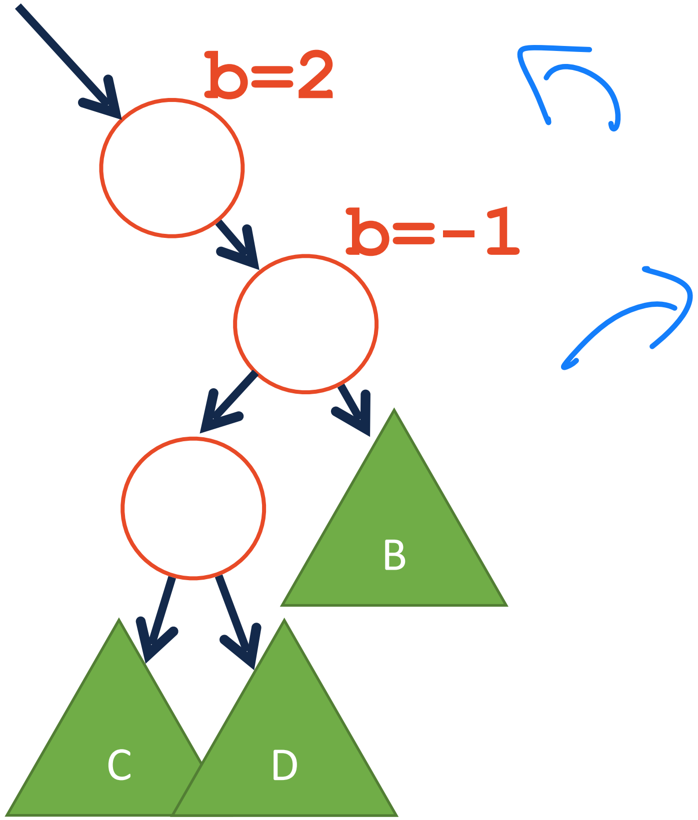
AVL Insertion

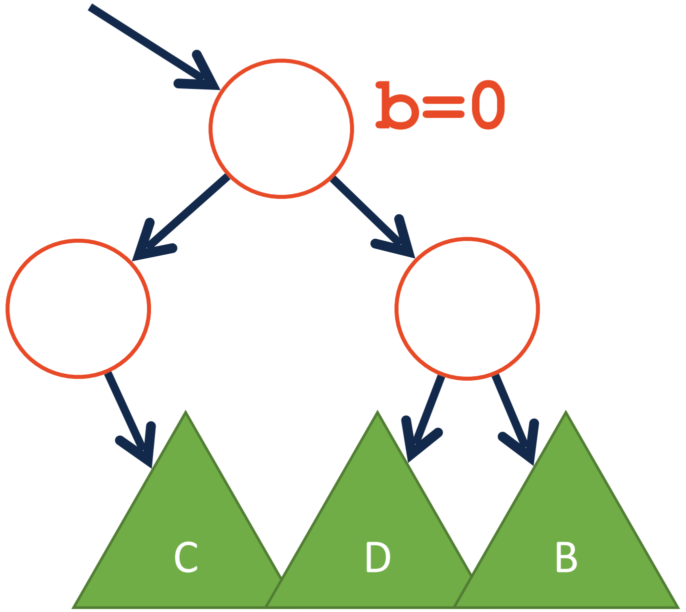
If we insert in A, I must have a balance pattern of **2, -1**



AVL Insertion

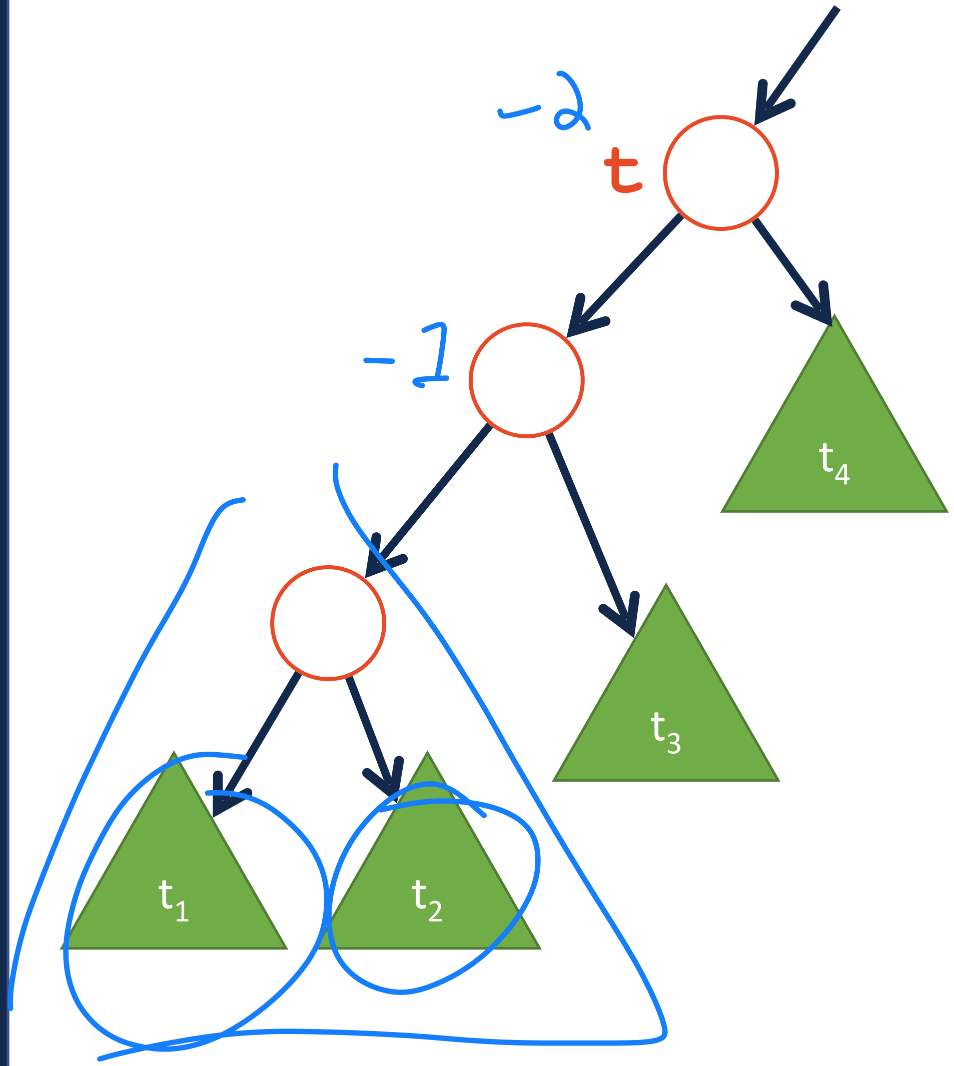
A **rightLeft** rotation fixes our imbalance in our local tree.





After rotation, subtree has **pre-insert height**. (Overall tree is balanced)

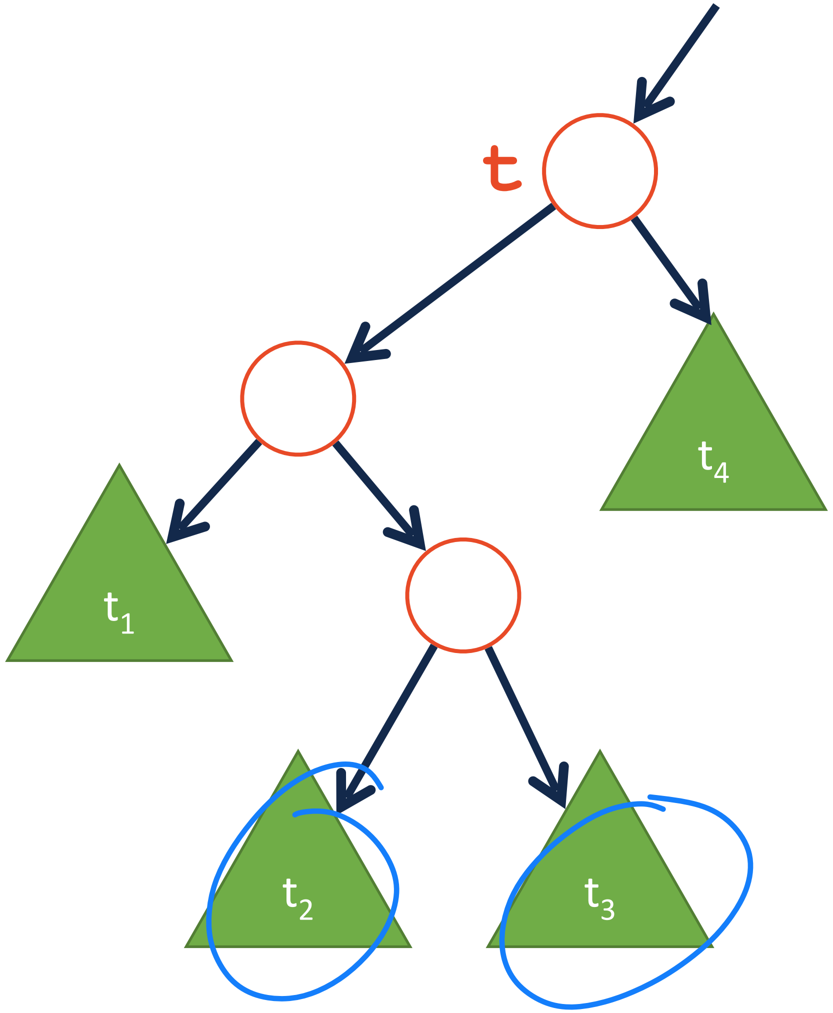
AVL Insertion



Theorem:
If an insertion occurred in subtrees $\mathbf{t_1}$ or $\mathbf{t_2}$ and an imbalance was first detected at **t**, then a ___Right___ rotation about **t** restores the balance of the tree.

We gauge this by noting the balance factor of **t is** ___-2___ and the balance factor of **t->left** is ___-1___.

AVL Insertion



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

We gauge this by noting the balance factor of **t is** ___-2___ and the balance factor of **t->left** is ___1___.

AVL Insertion

We've seen every possible insert that can cause an imbalance

Insert *may* increase height by at most: One

A rotation reduces the height of the subtree by: One

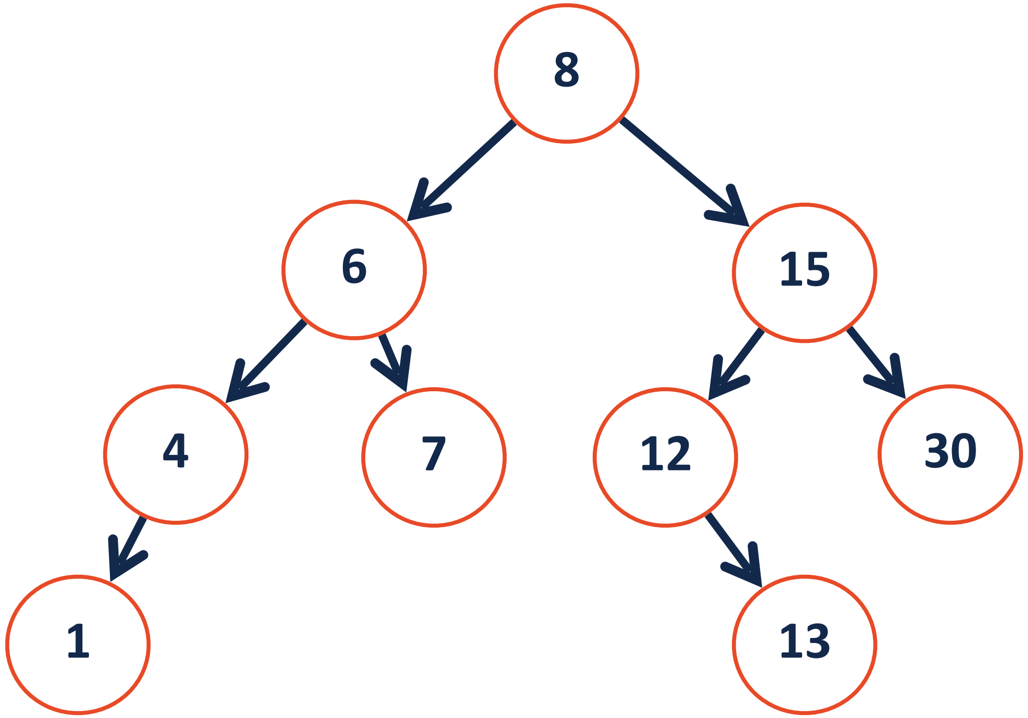
A single* rotation restores balance and corrects height!

What is the Big O of performing our rotation? $O(1)$

What is the Big O of insert? $O(h)$

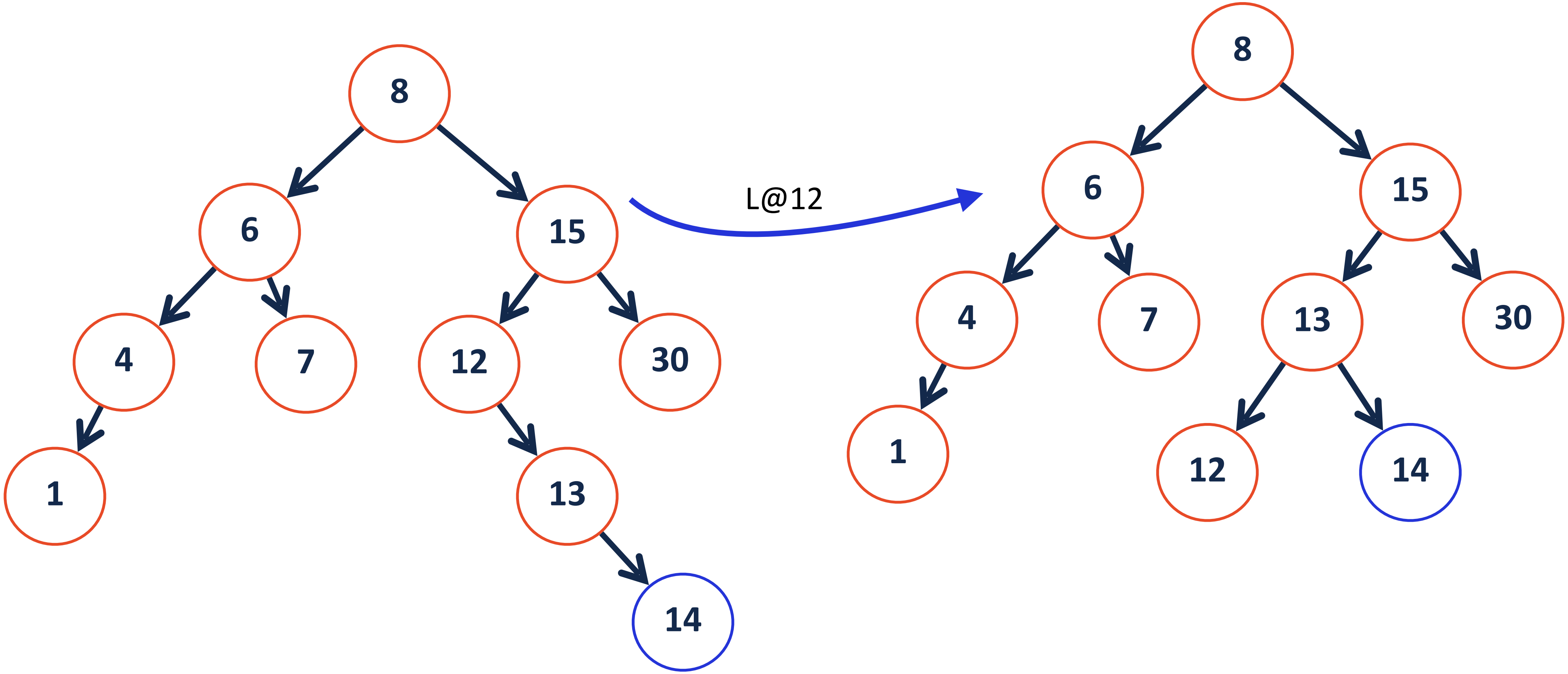
AVL Insertion Practice

`_insert(14)`



AVL Insertion Practice

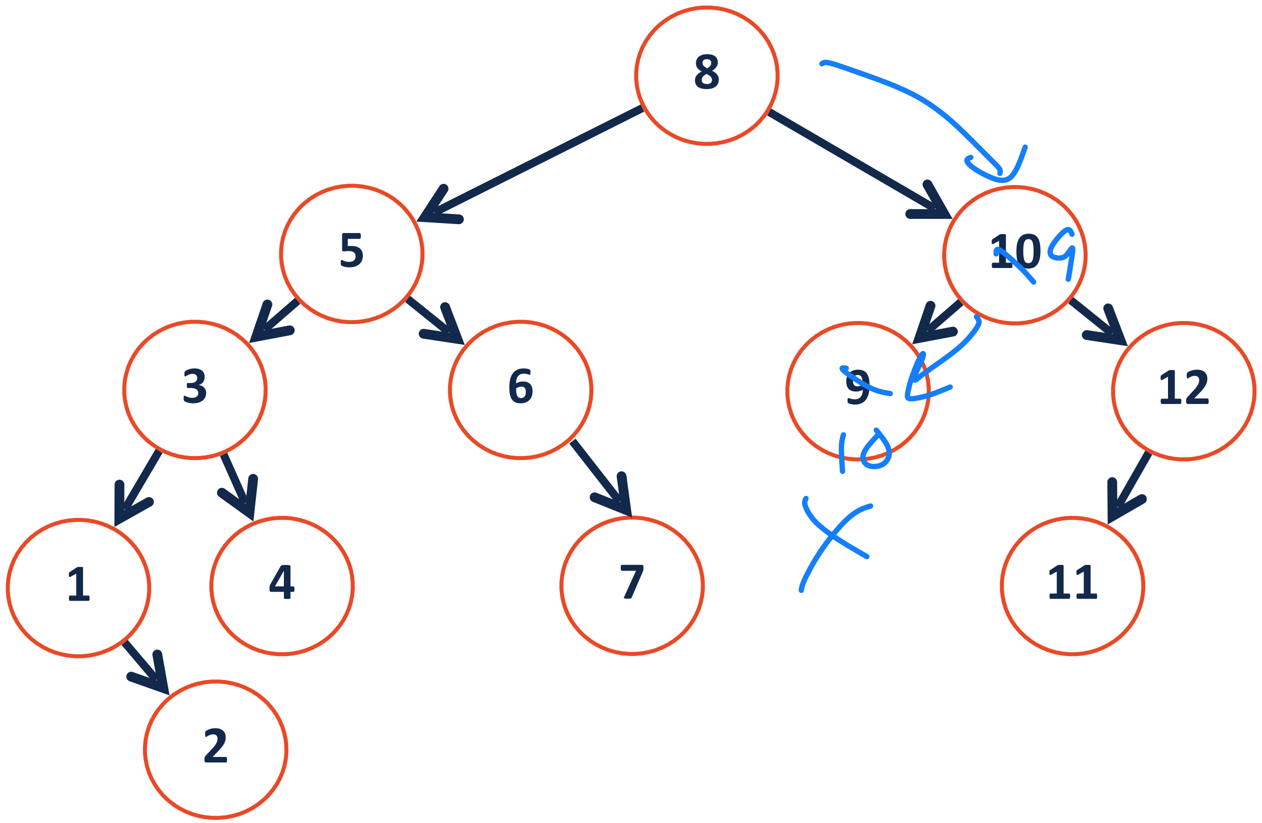
`_insert(14)`



AVL Remove

`_remove(10)`

Find & remove
↳ Find IOP
↳ Swap

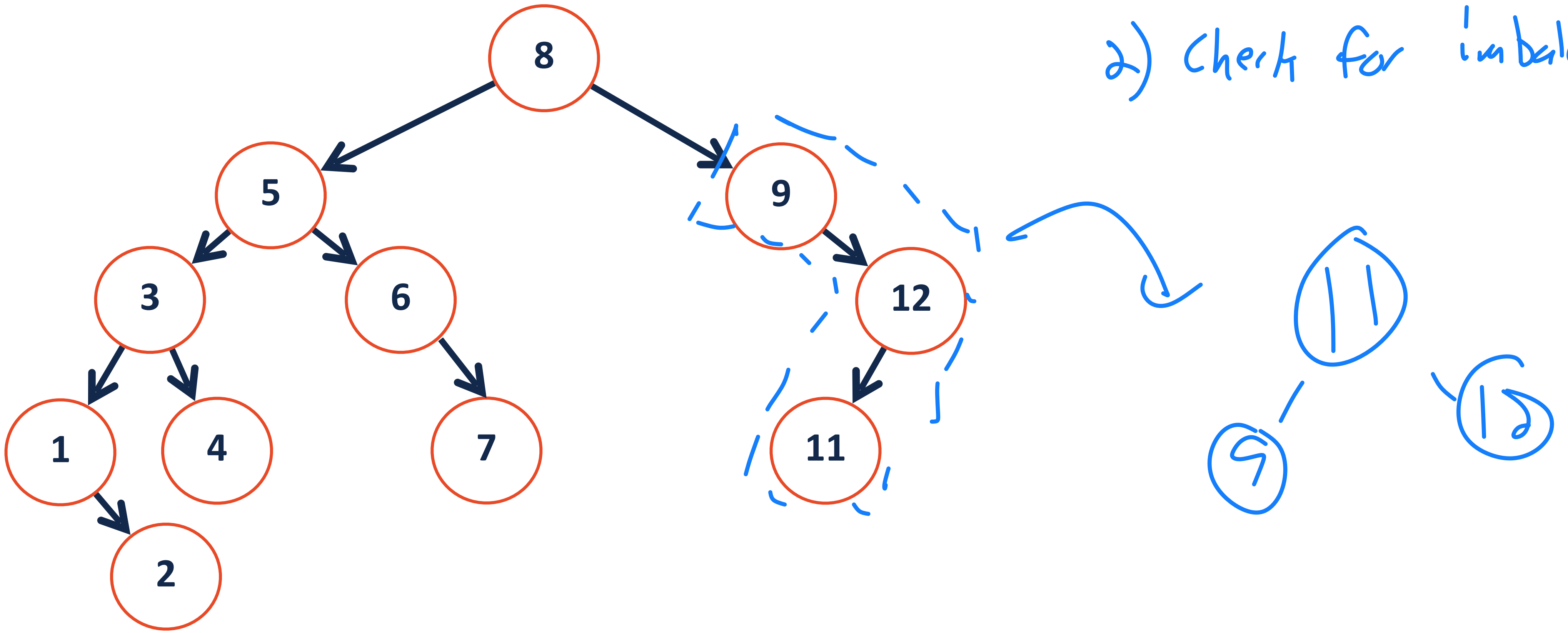


AVL Remove

`_remove(10)`

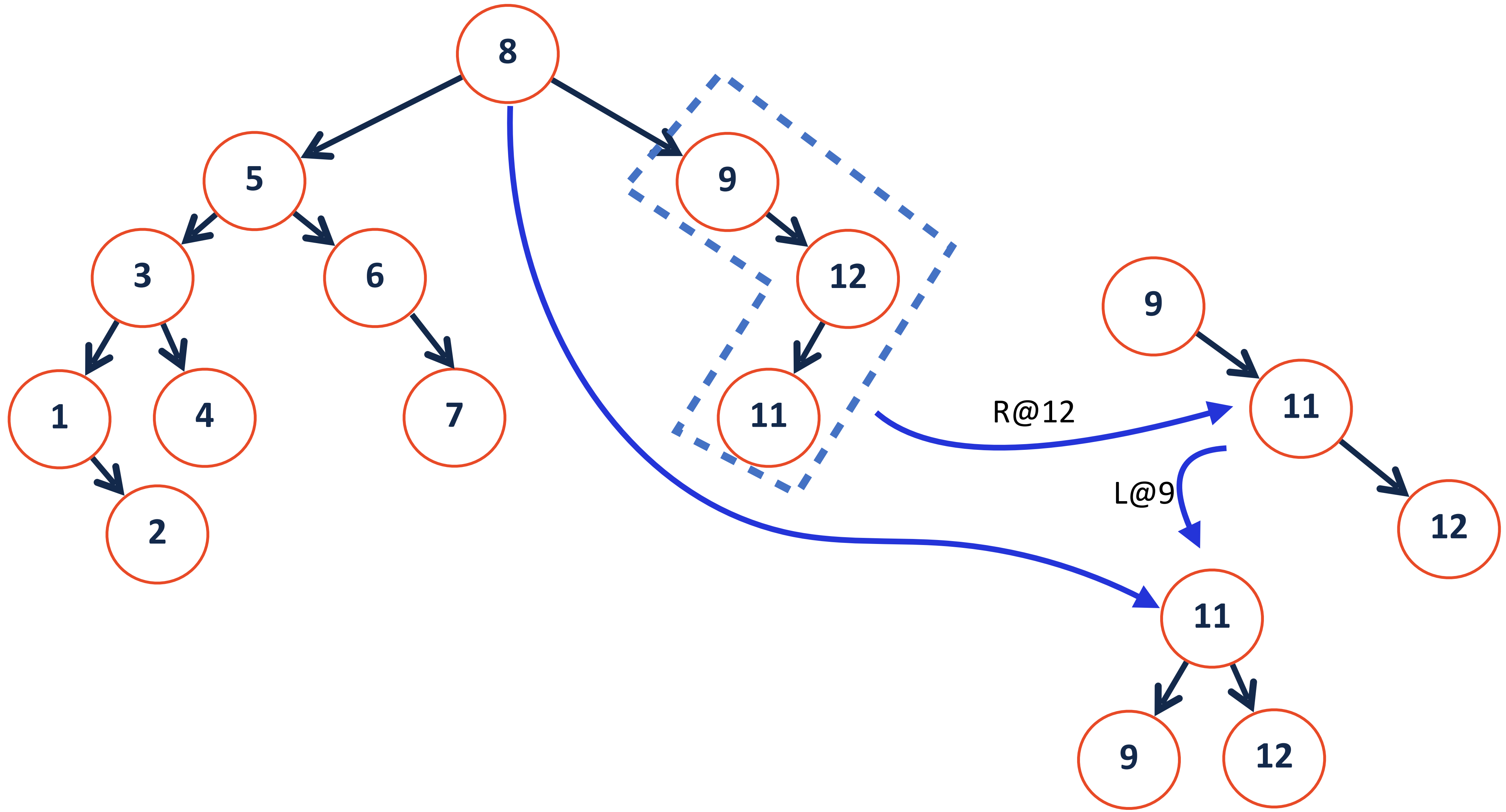
1) Do remove

2) Check for imbalance



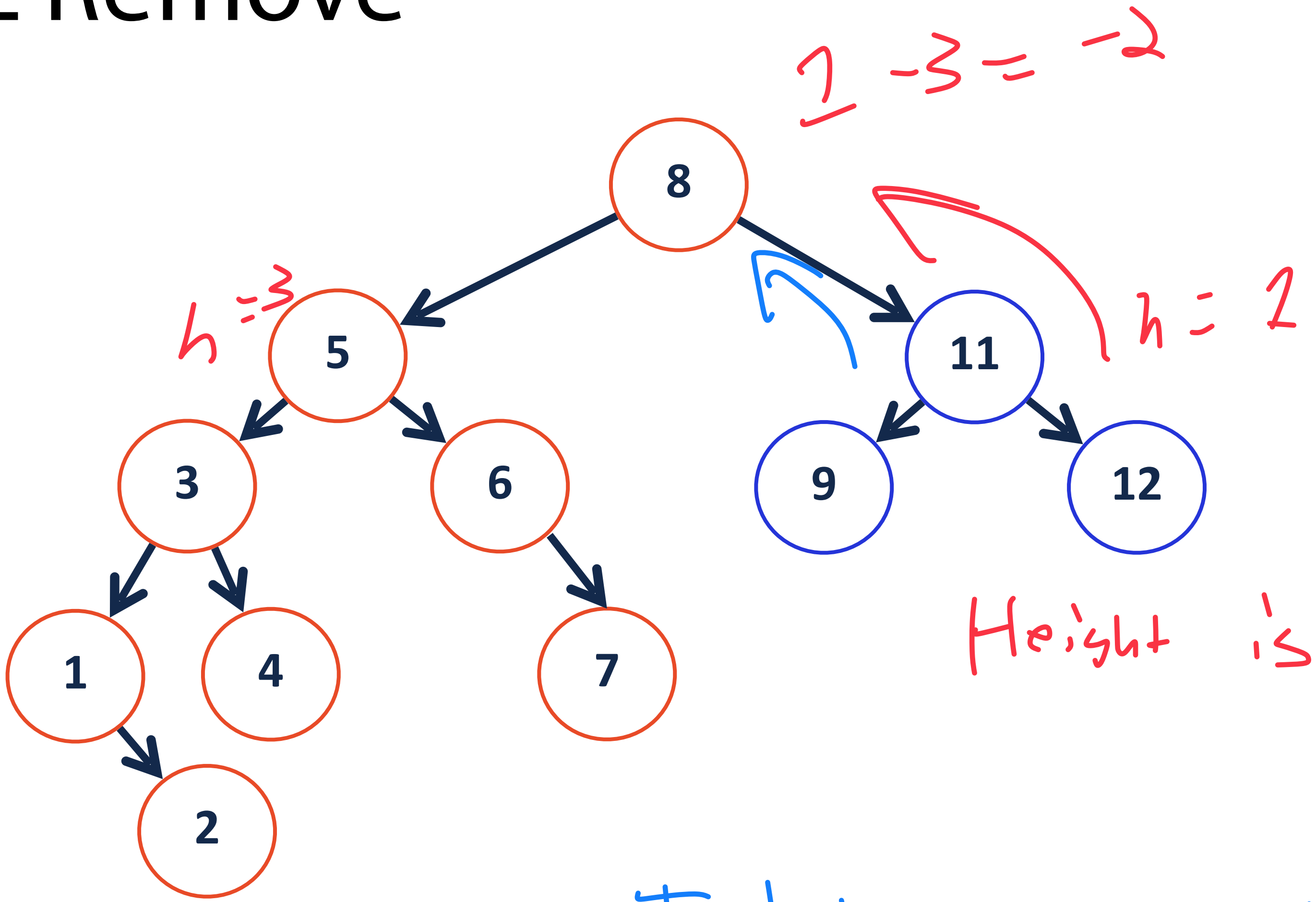
AVL Remove

`_remove(10)`



AVL Remove

`_remove(10)`

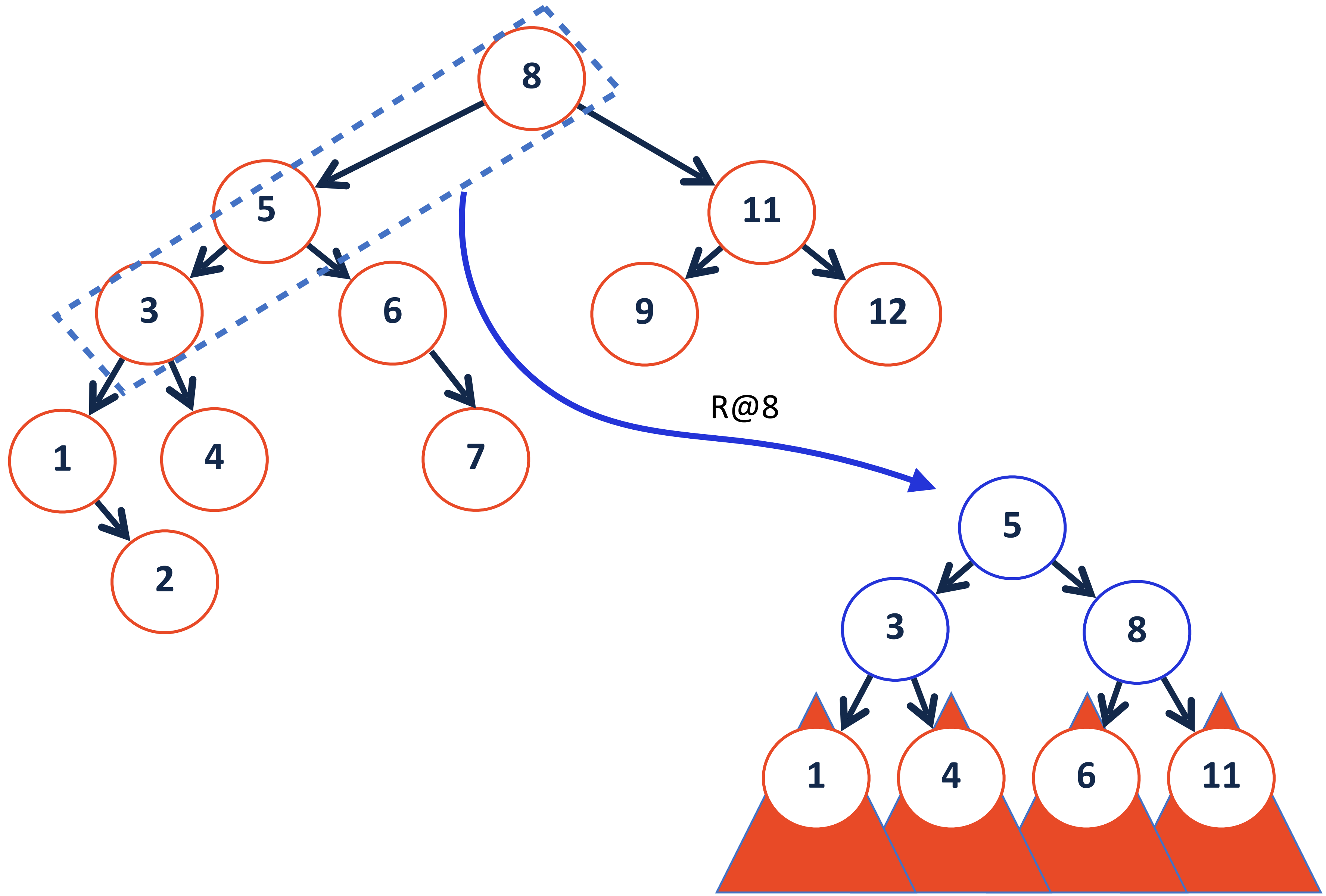


Height is reduced by one!

Imbalanced point tree

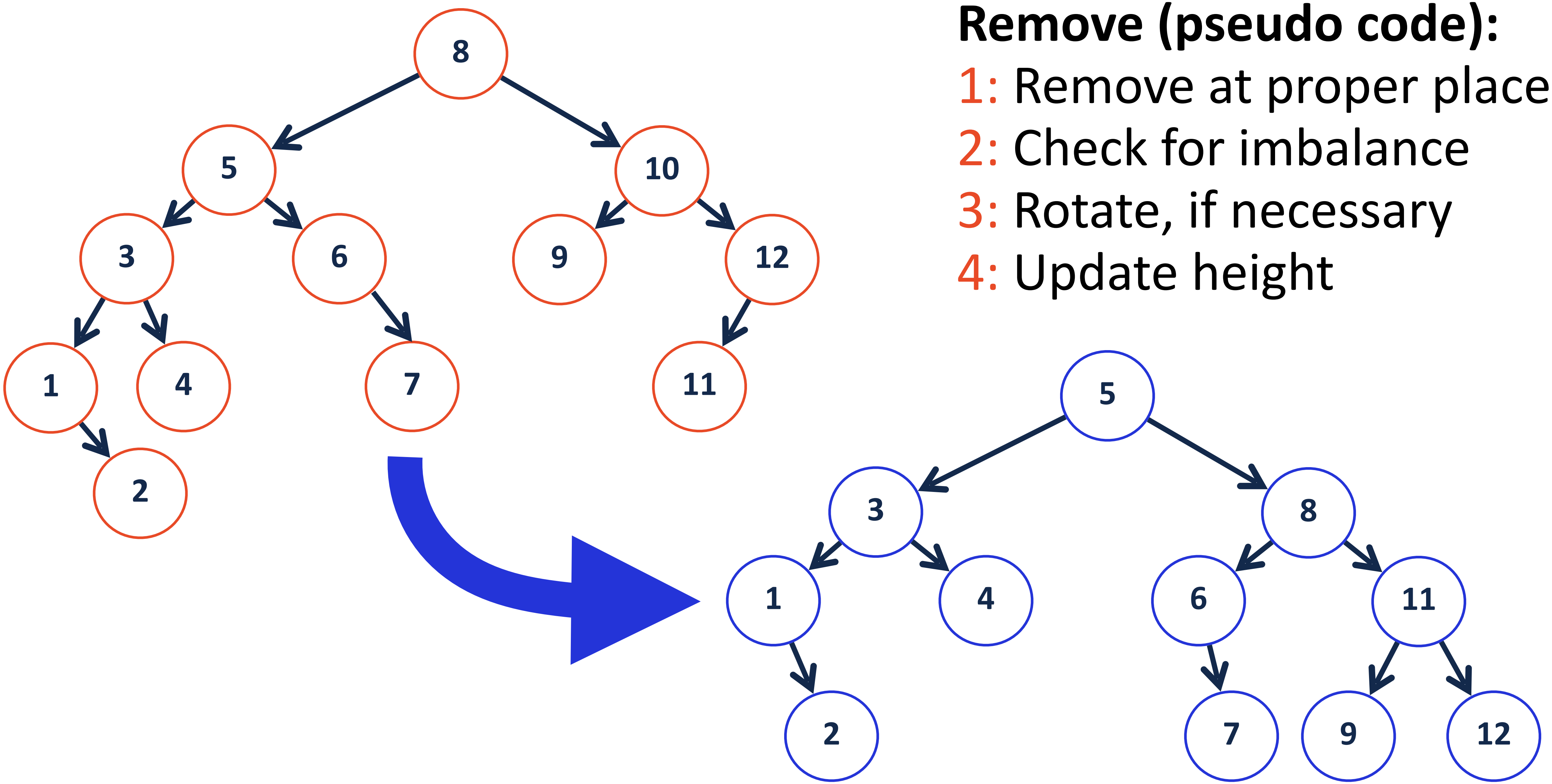
AVL Remove

`_remove(10)`



AVL Remove

`_remove(10)`



Remove (pseudo code):

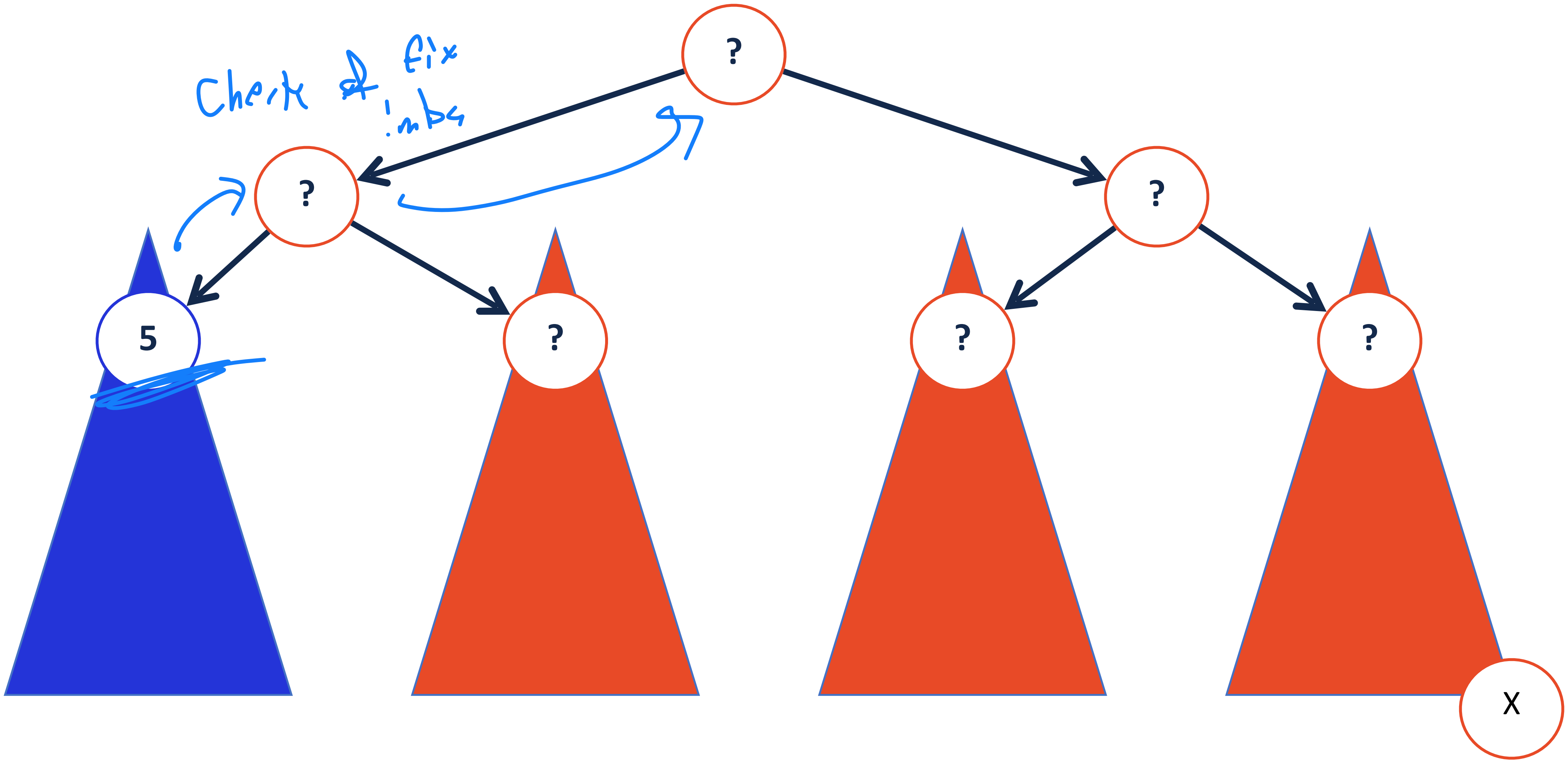
1: Remove at proper place

2: Check for imbalance

3: Rotate, if necessary

4: Update height

AVL Remove



AVL Remove

An AVL remove step can reduce a subtree height by at most: 1

But a rotation ***reduces*** the height of a subtree by one!

We might have to perform a rotation at every level of the tree!

AVL Tree Analysis

For an AVL tree of height h:

Find runs in: $O(h)$.

Insert runs in: $O(h)$.

Remove runs in: $O(h)$.

Claim: The height of the AVL tree with n nodes is: $O(\log n)$.