Data Structures
AVL Trees - 2

CS 225
Harsha Tirumala

October 3, 2025

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Learning Objectives

Review why we need balanced trees

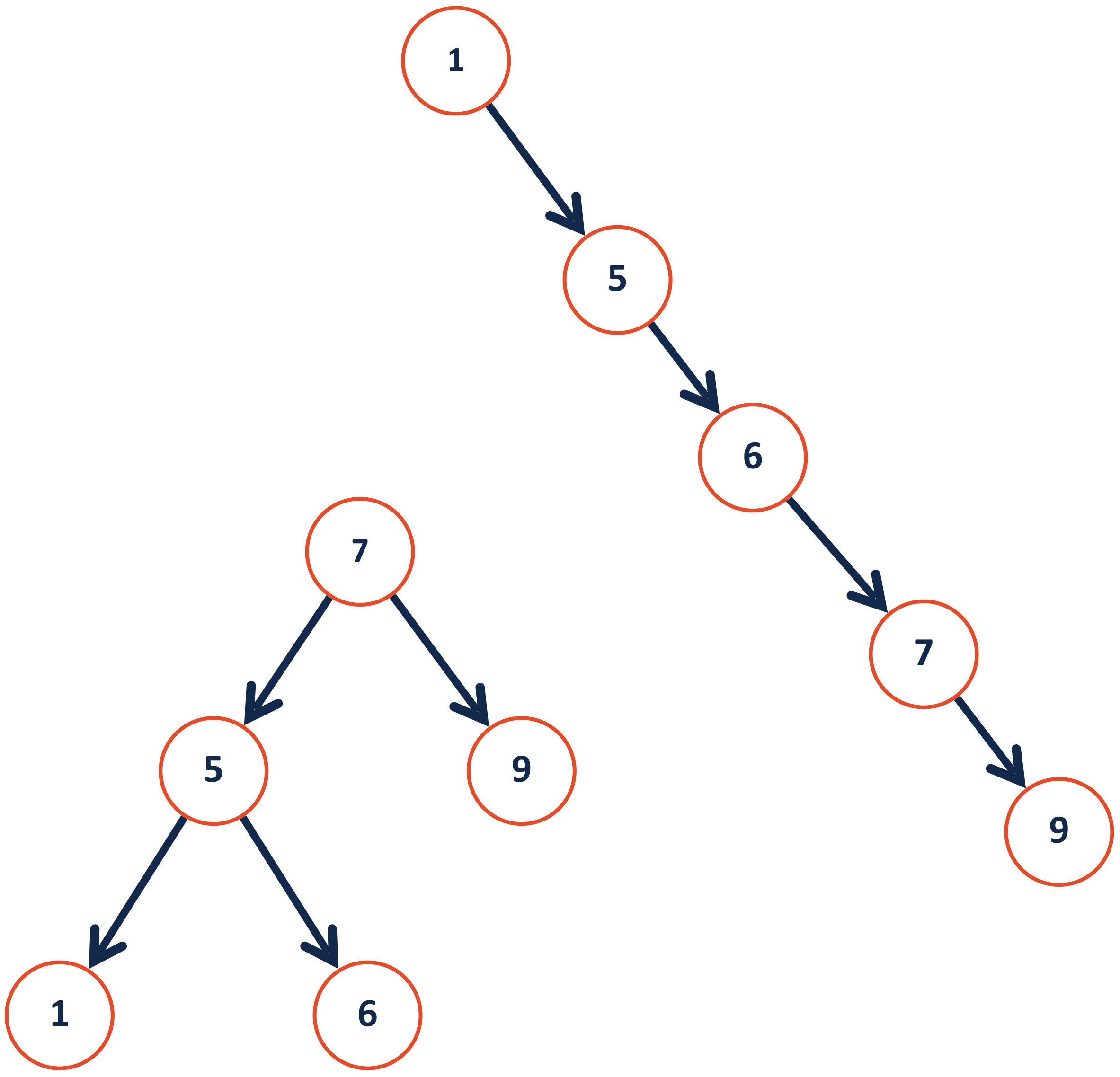
Review what an AVL rotation does

Review the four possible rotations for an AVL tree

Explore the implementation of AVL Tree

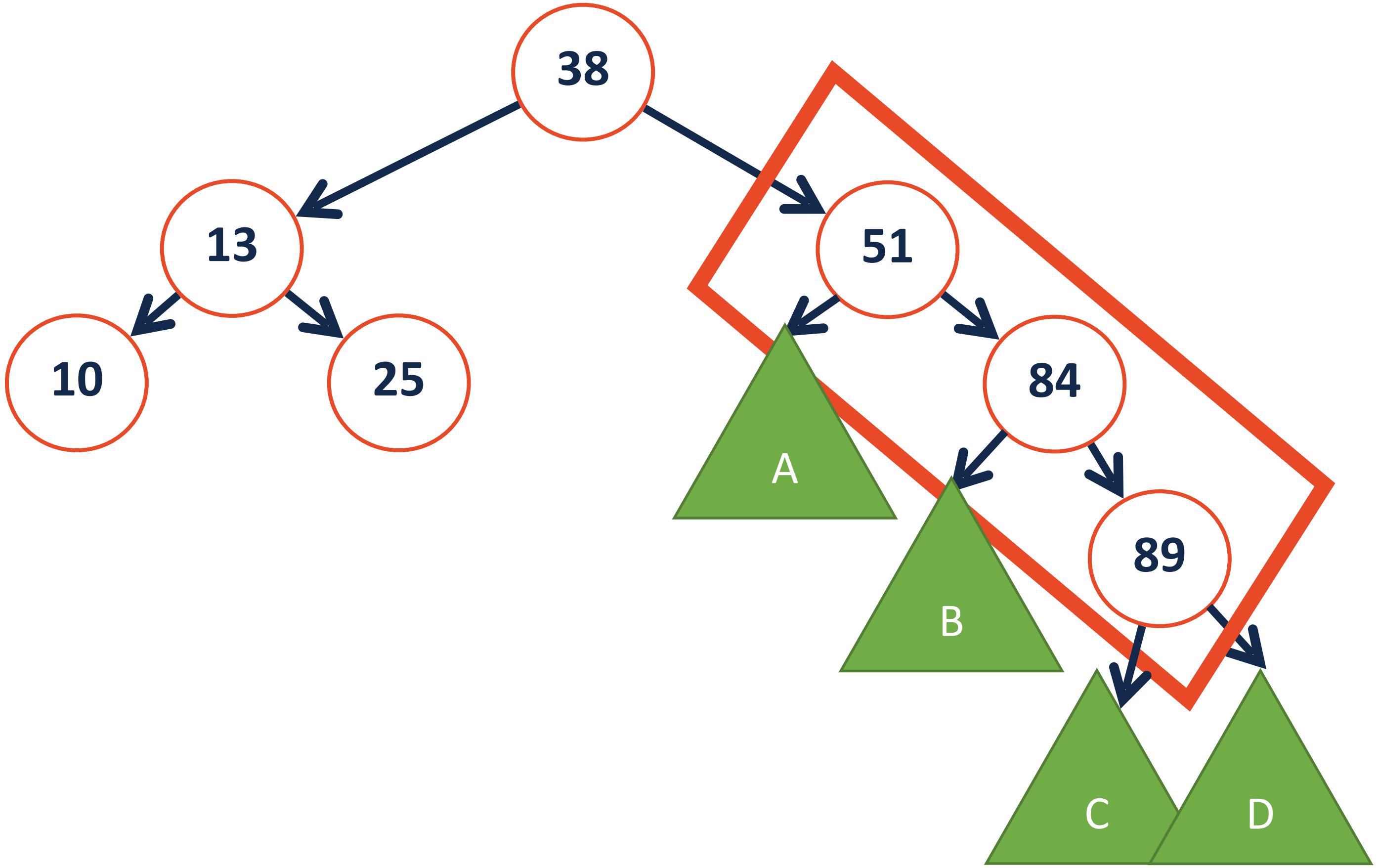
BST Analysis – Running Time

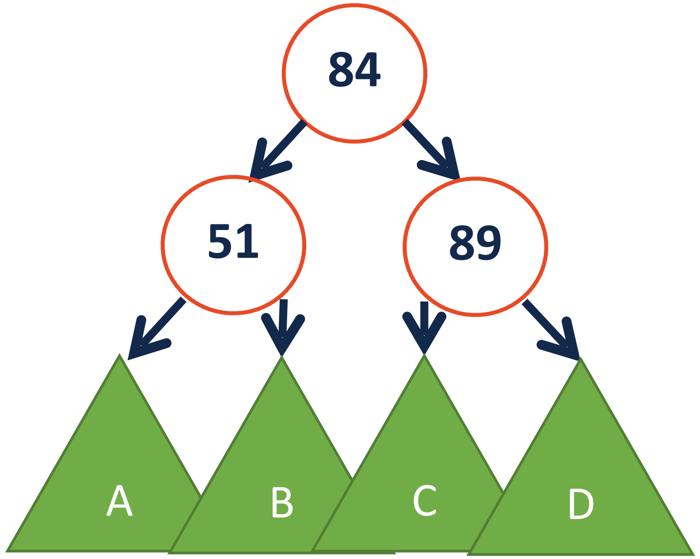
	BST Worst Case
find	O(h)
insert	O(h)
delete	O(h)
traverse	O(n)



AVL-Tree: A self-balancing binary search tree

Rather than fixing an insertion order, just correct the tree as needed!





Adelson **V**elsky,
Evgenii **L**andis 1962

BST Rotations (The AVL Tree)

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

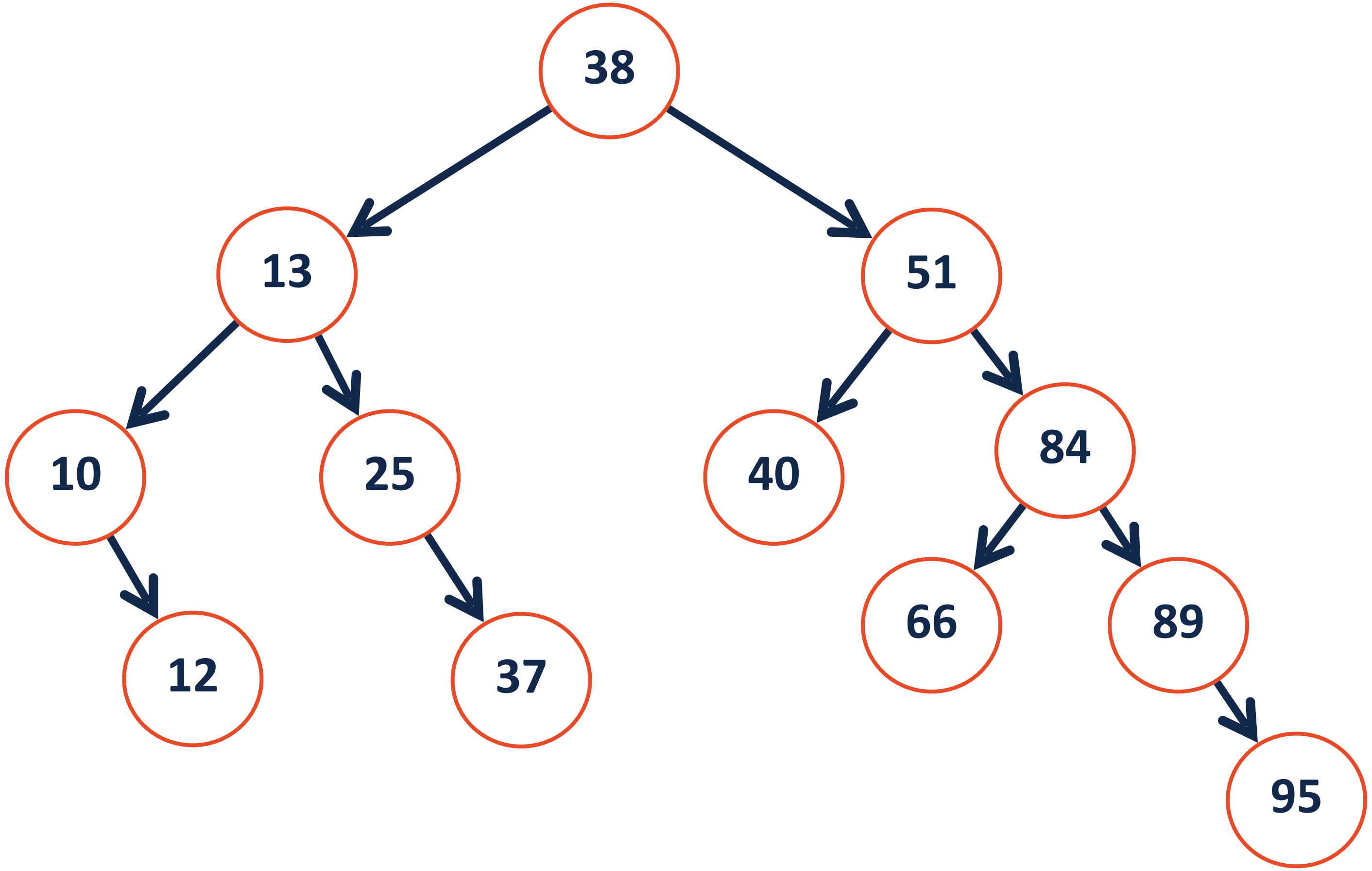
These rotations, when used correctly:

1. Modify the arrangement of nodes while <u>preserving BST property</u>

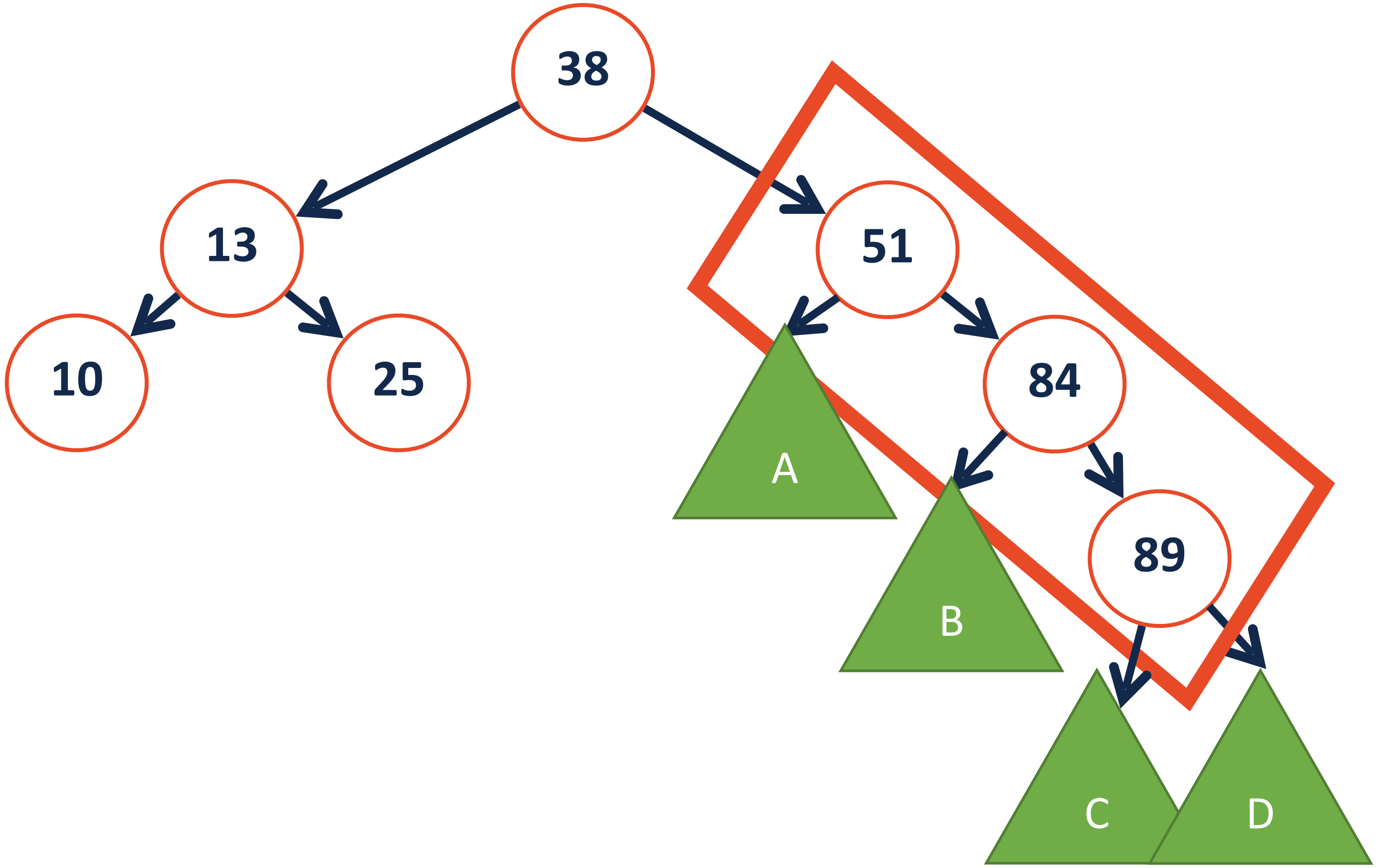
2. <u>Reduce tree height by one</u>

BST Rotations (The AVL Tree)

To begin, lets find the imbalance in the following tree:

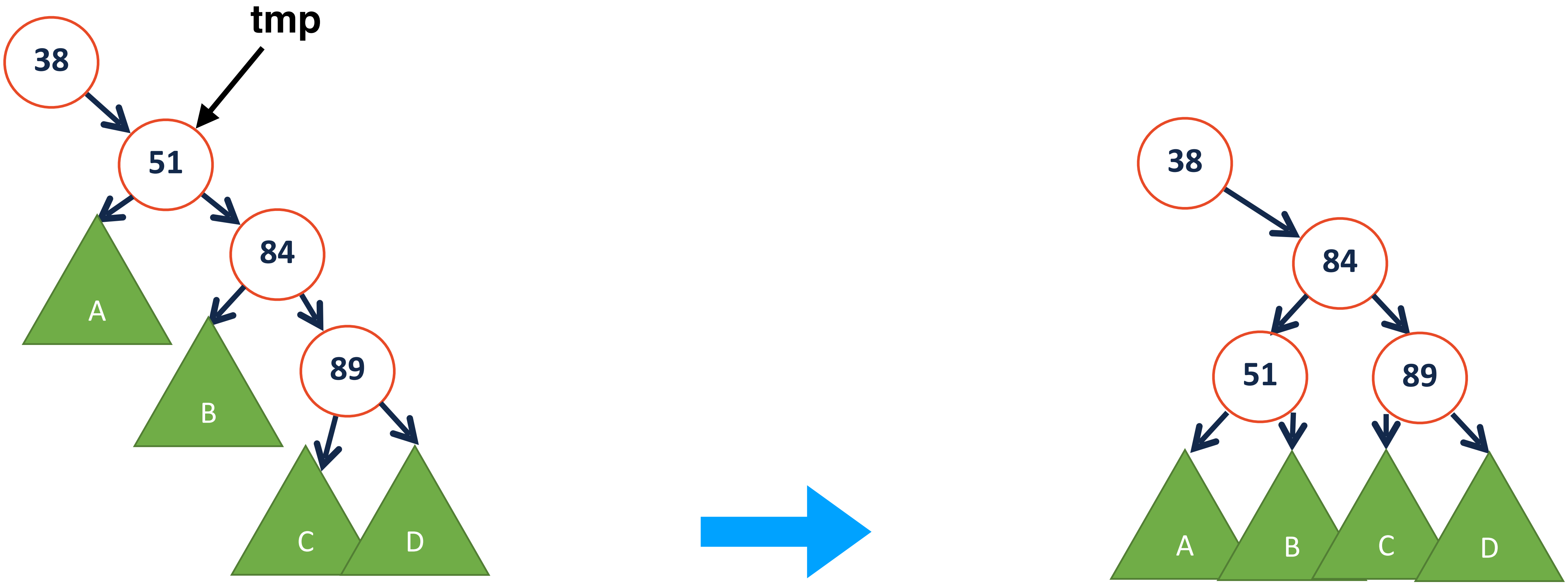


Left Rotation



Left Rotation

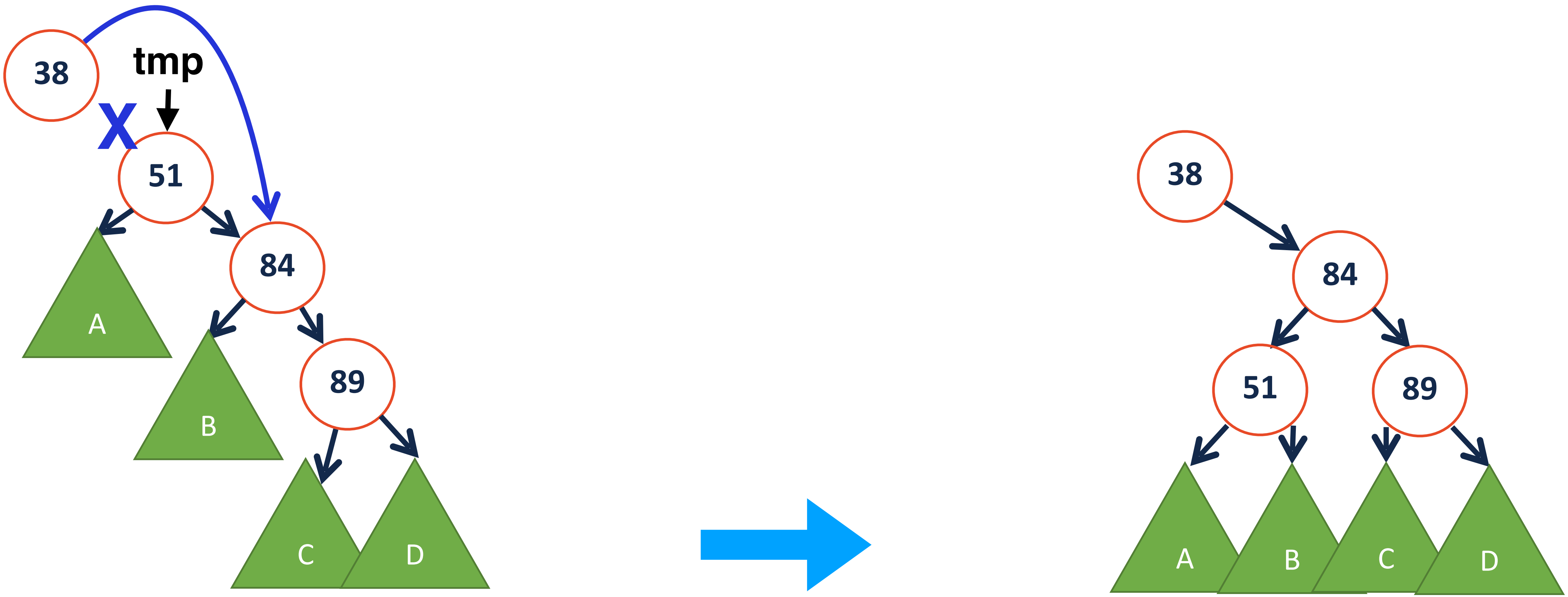
1) Create a tmp pointer to root



Left Rotation

1) Create a tmp pointer to root

2) Update root to point to mid

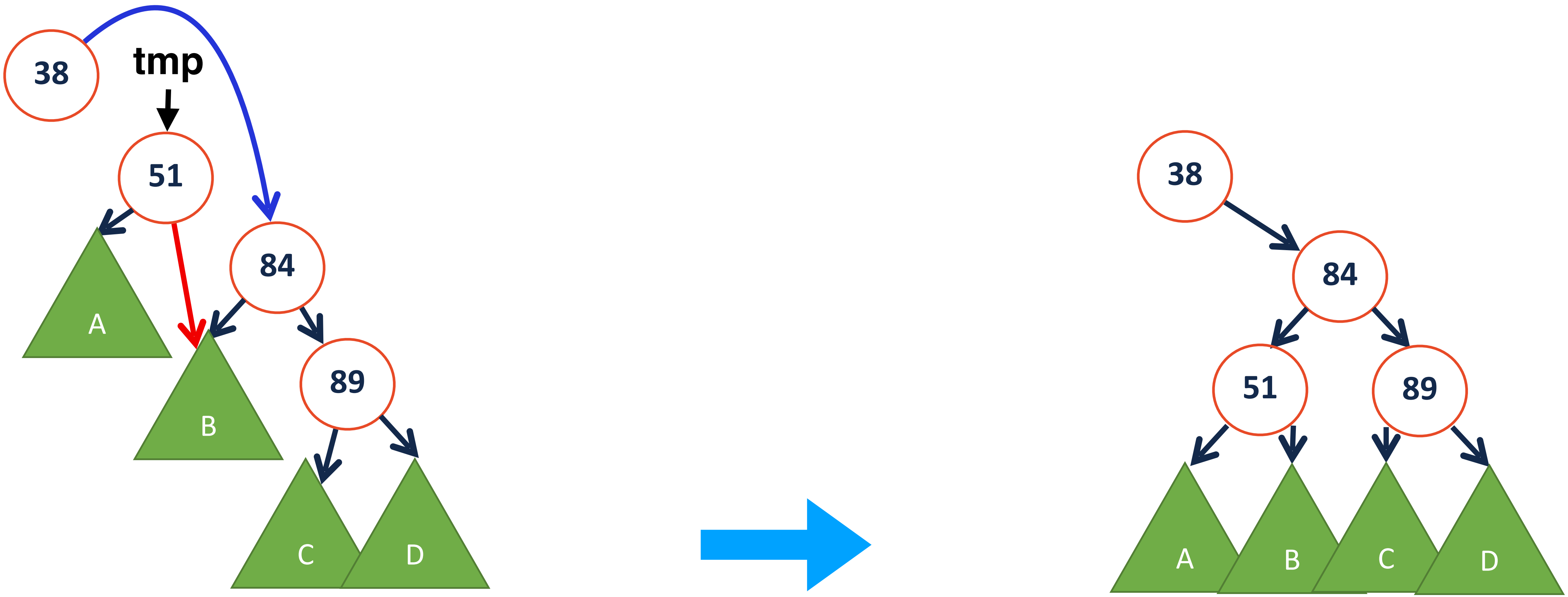


Left Rotation

1) Create a tmp pointer to root

2) Update root to point to mid

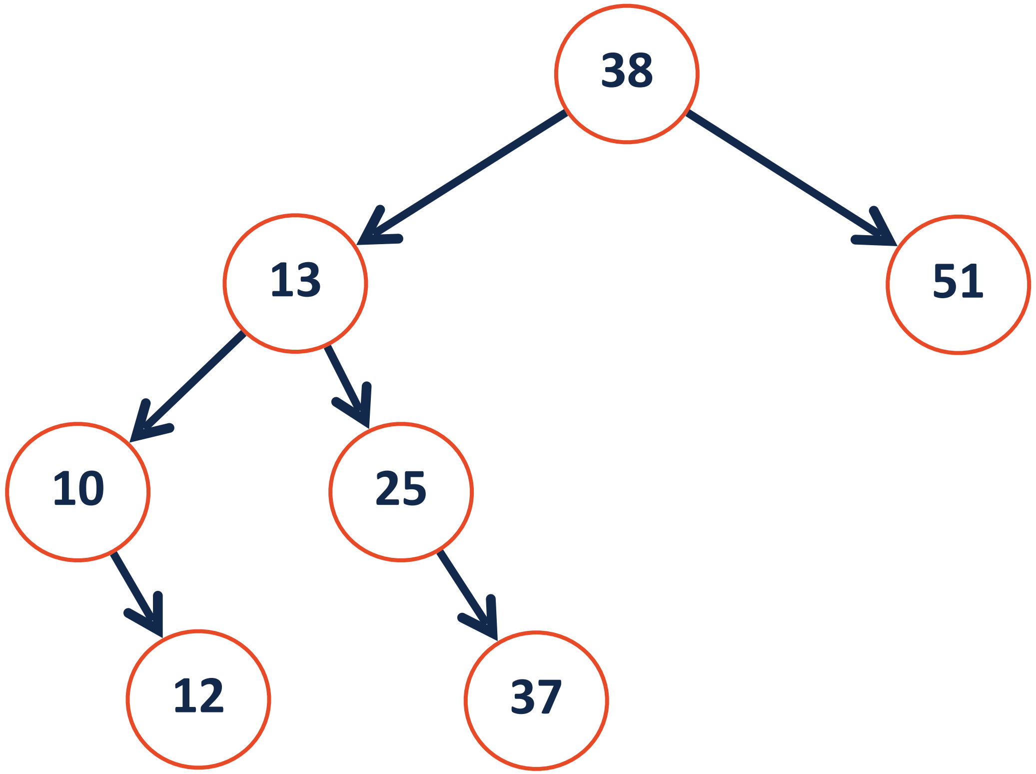
3) tmp->right = root->left



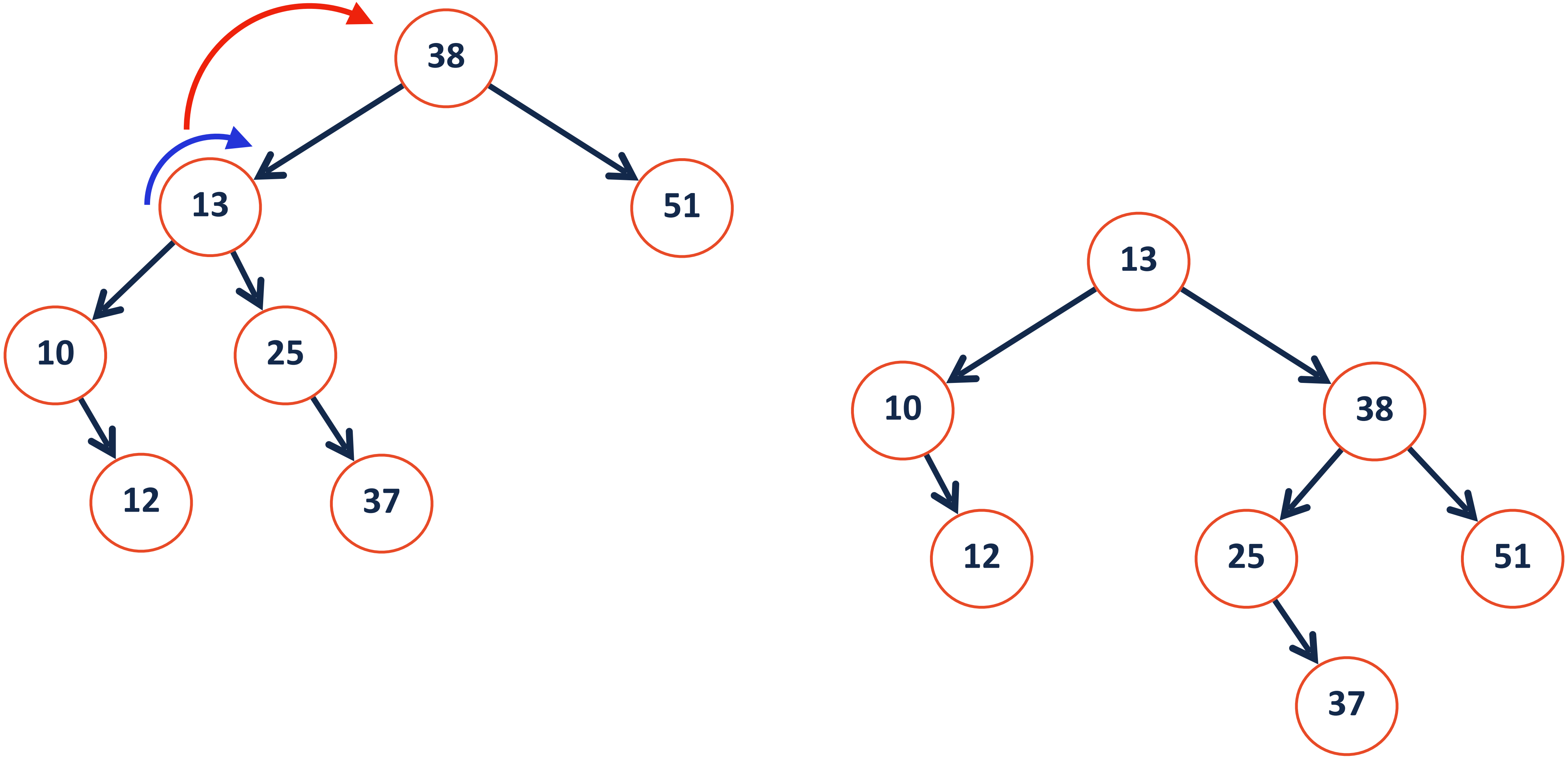
Left Rotation

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

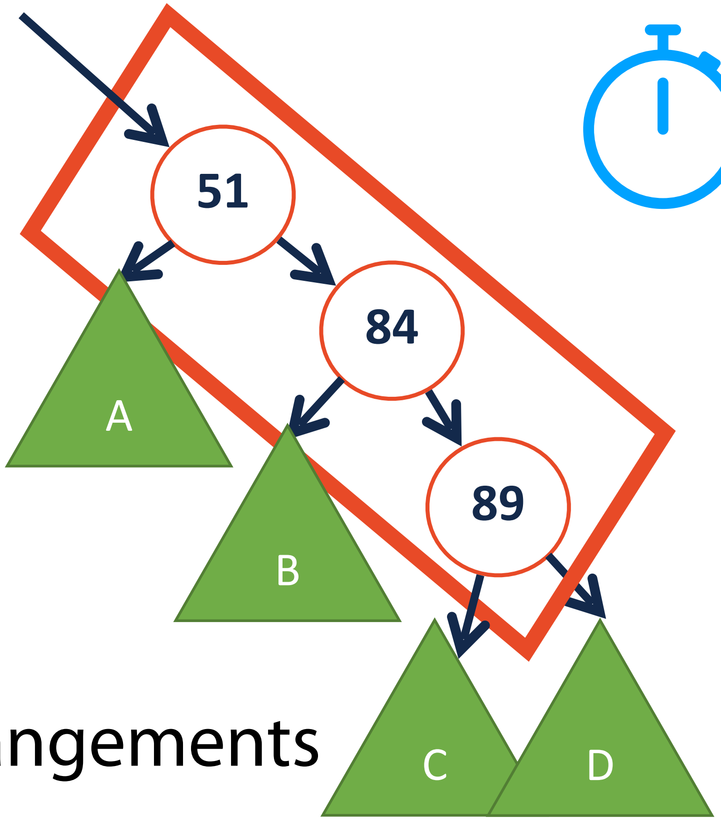
Right Rotation



Right Rotation



Coding AVL Rotations



Two ways of visualizing:

1) Think of an arrow 'rotating' around the center

2) Recognize that there's a concrete order for rearrangements

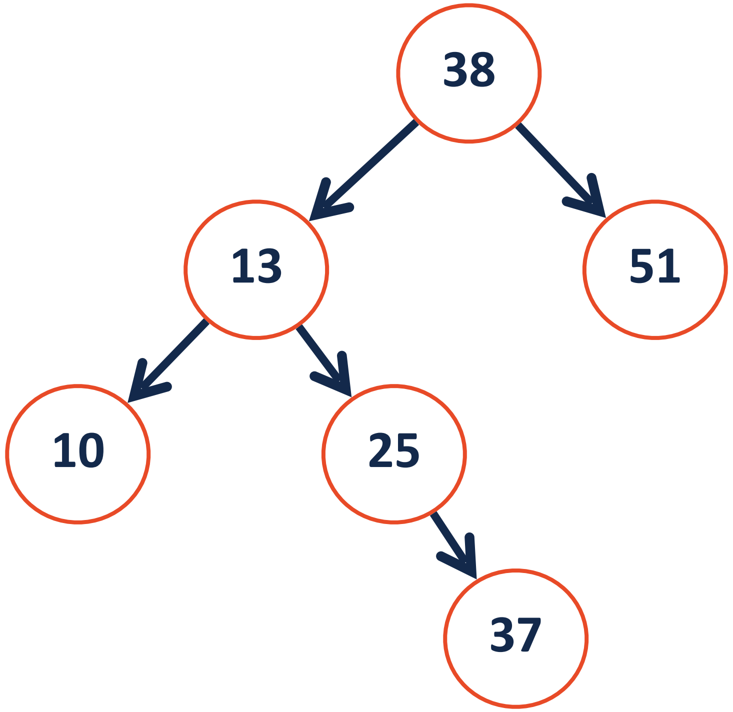
Ex: Unbalanced at current (root) node and need to *rotateLeft*?

Replace current (root) node with it's right child.

Set the right child's left child to be the current node's right

Make the current node the right child's left child

AVL Rotation Practice

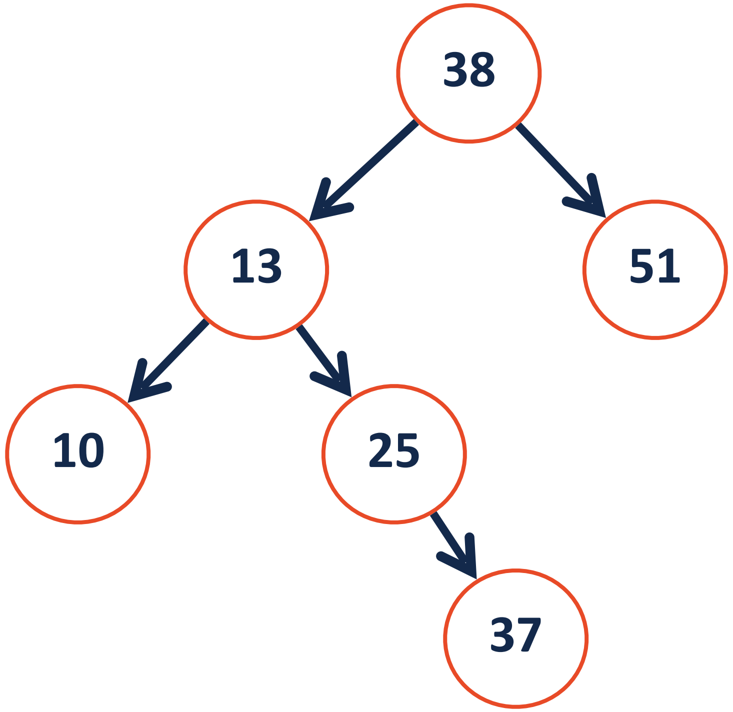


AVL Rotation - Problems

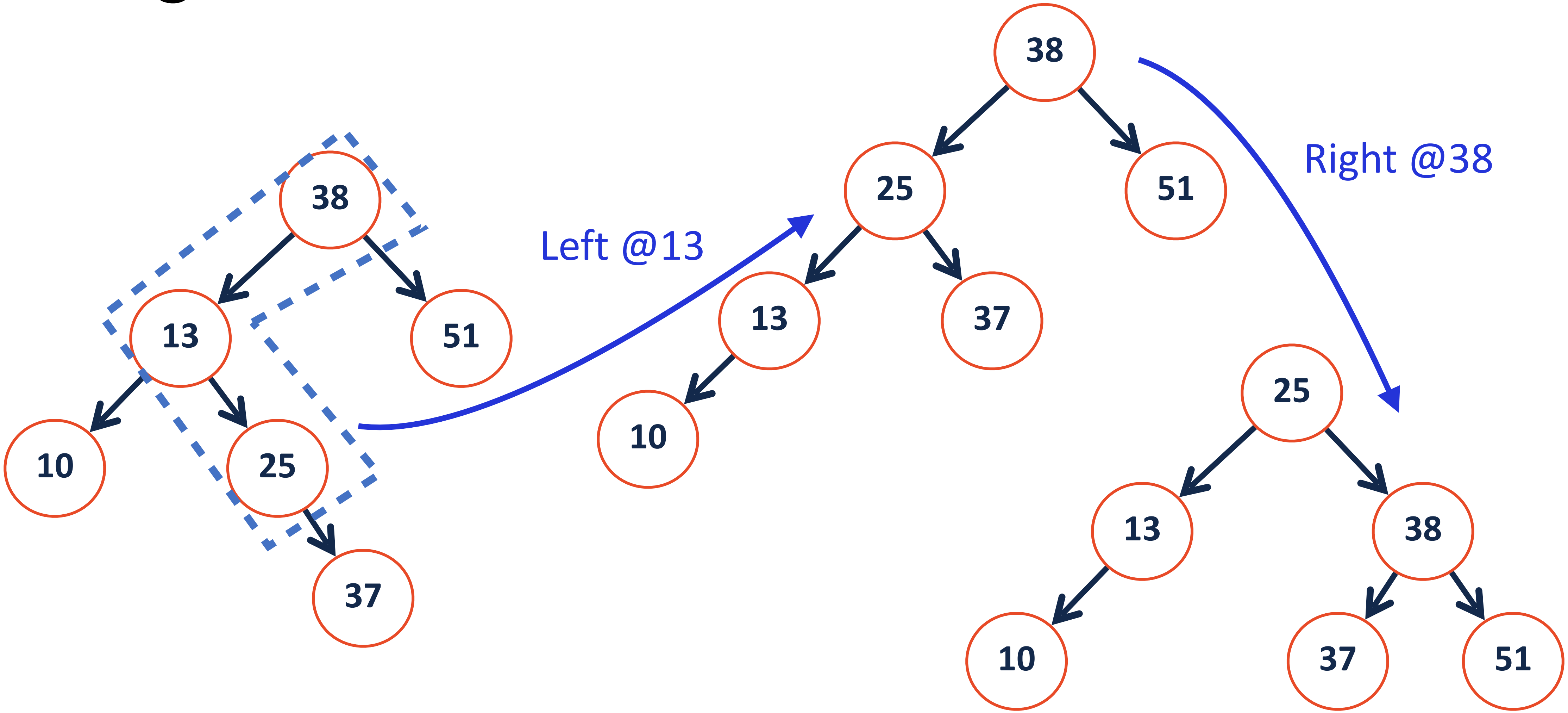
Somethings not quite right…

LeftRight Rotation

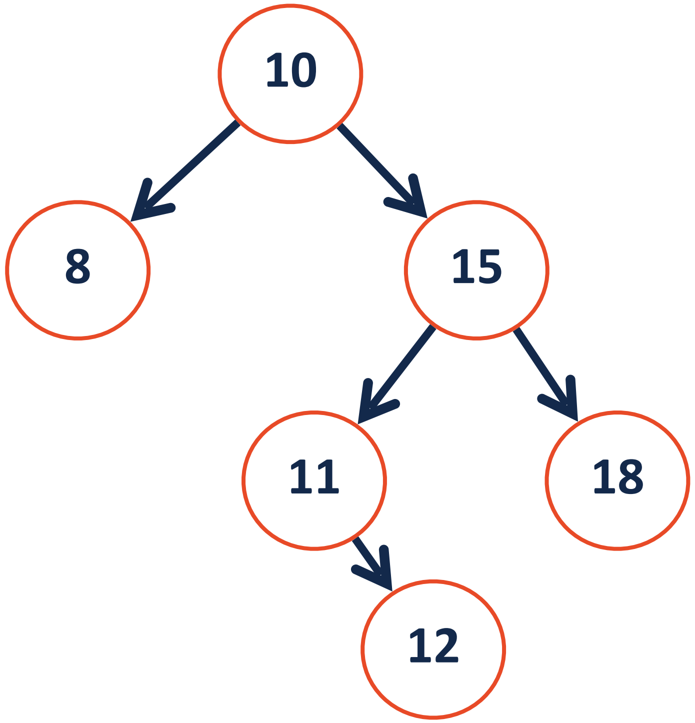
<u>right heavy left imbalance</u>



LeftRight Rotation

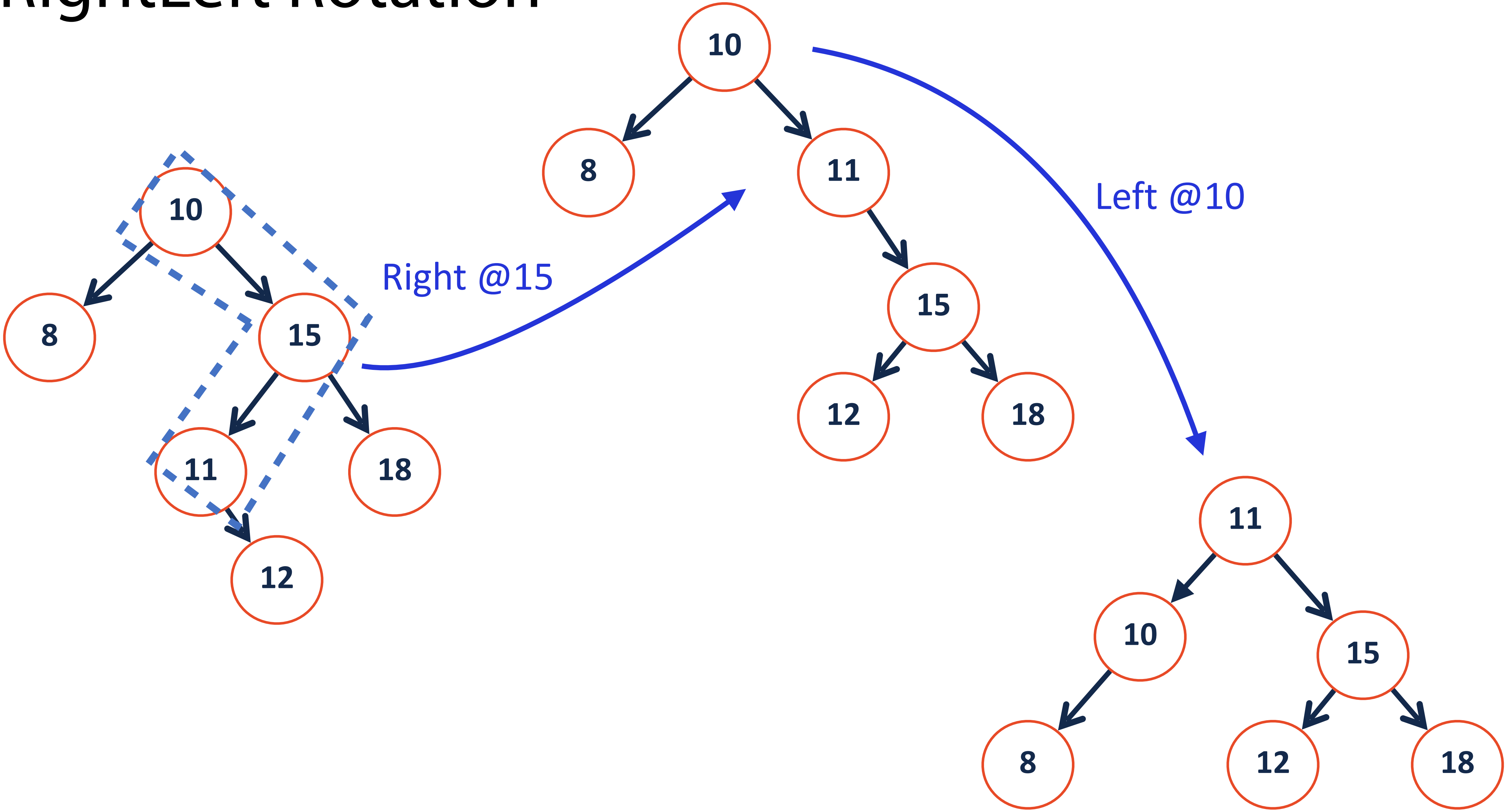


RightLeft Rotation

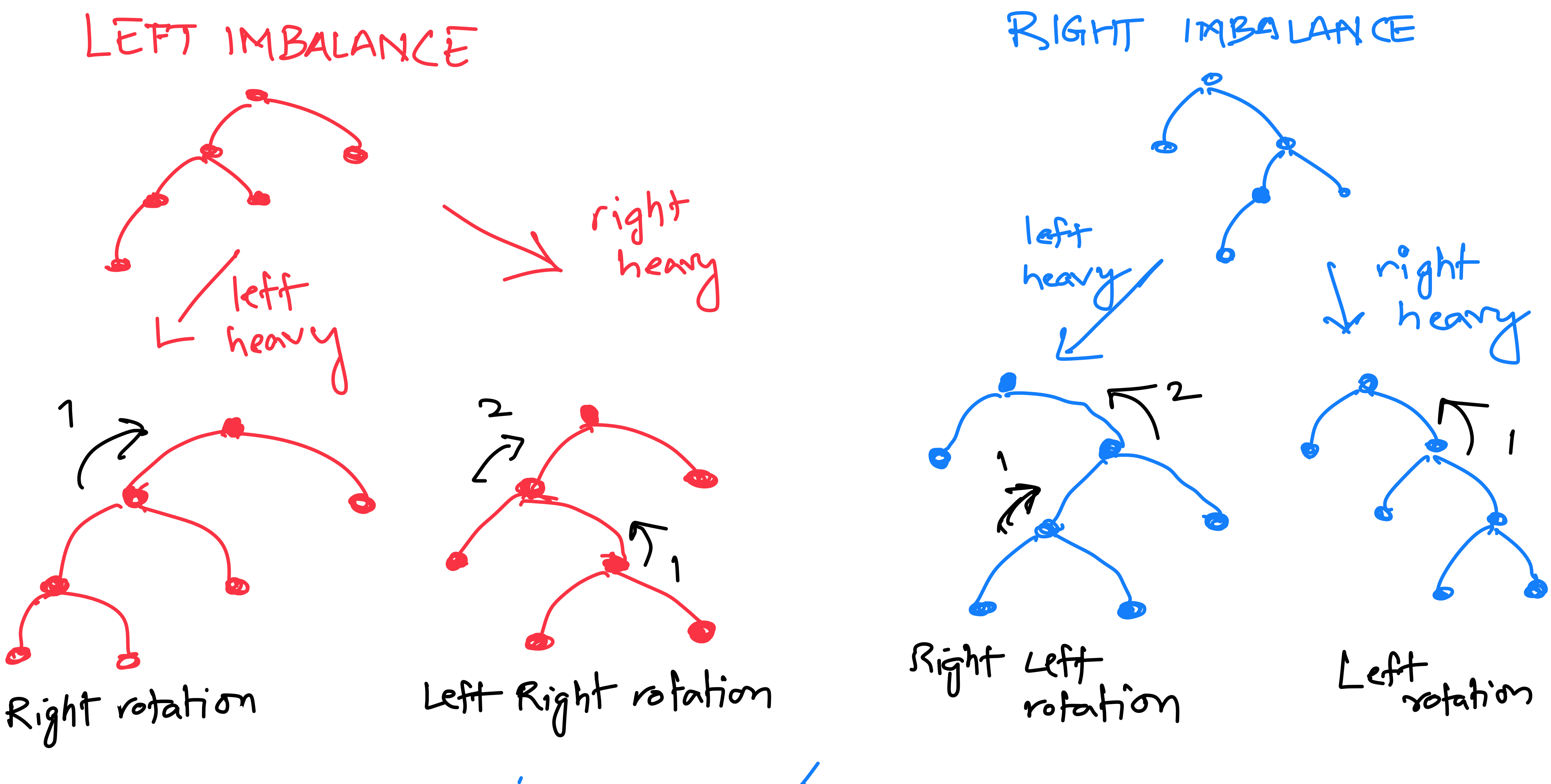


Left heavy right imbalance

RightLeft Rotation



AVL Rotations - types (diagram)



AVL Rotations - types

1. Right Rotation

2. Left Rotation

3. Left-Right Rotation

4. Right-Left Rotation

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:

Produce trees of height = O(log n) so that all our major operations (find, insert, remove) are O(log n) time.

AVL Rotations - steps

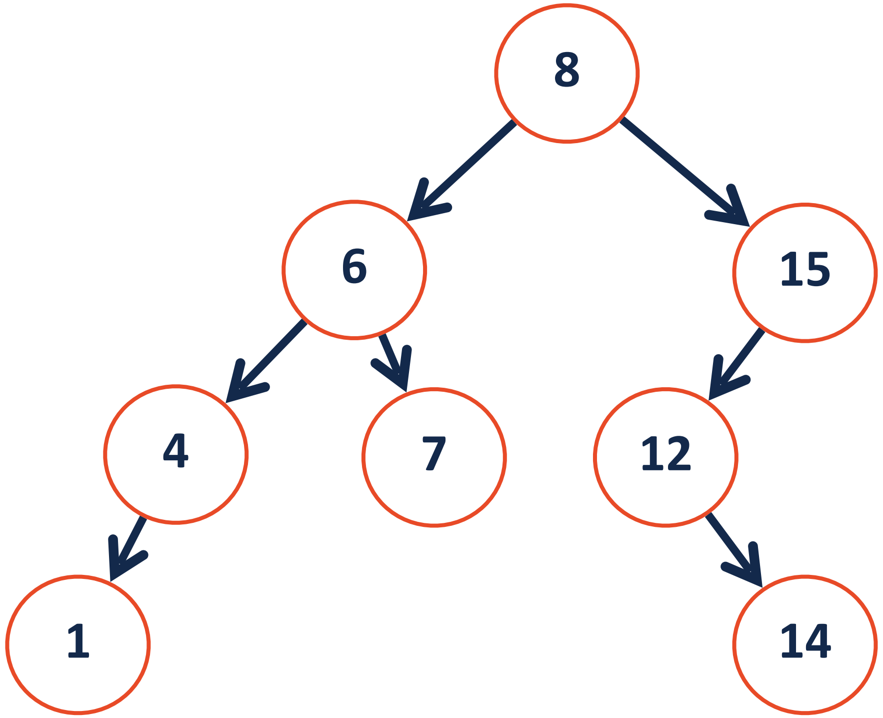
1. Identify nodes with |height balance| >= 2

2. In a bottom up manner, fix nodes with |hb| >= 2

3. Identify the type of rotation to apply - by considering heights and balance of (heavier) child.

4. Execute rotation and return to parent in a bottom up manner until the entire tree is balanced.

AVL Rotation Practice



AVL vs BST ADT

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

How does the constraint on balance affect the core functions?

Operation	BST h = O(n)	AVL tree h = O(log n)
Find	O(h)	O(log n)
Insert	O(h)	O(log n)
Remove	O(h)	O(log n)

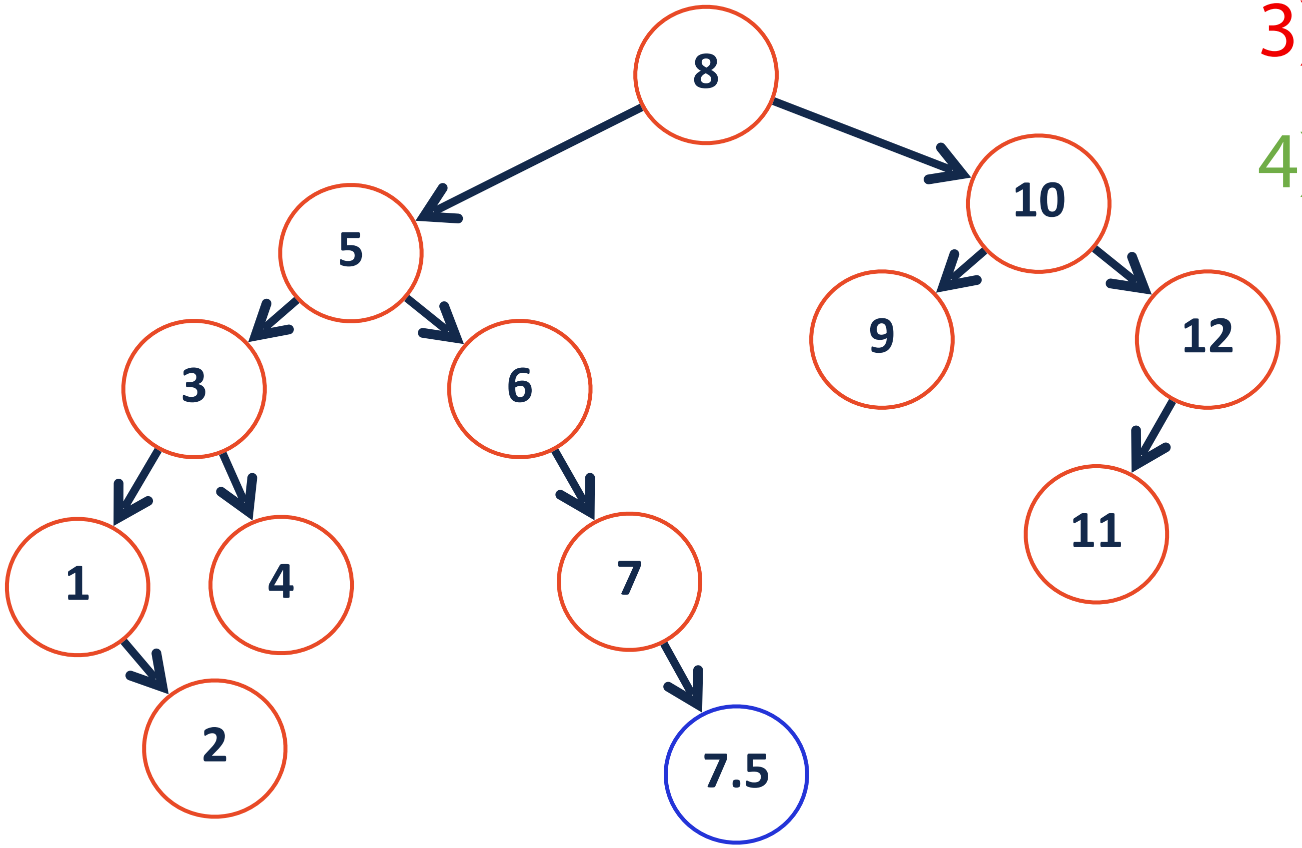
Left Rotation

1) Create a tmp pointer to root

2) Update root to point to mid

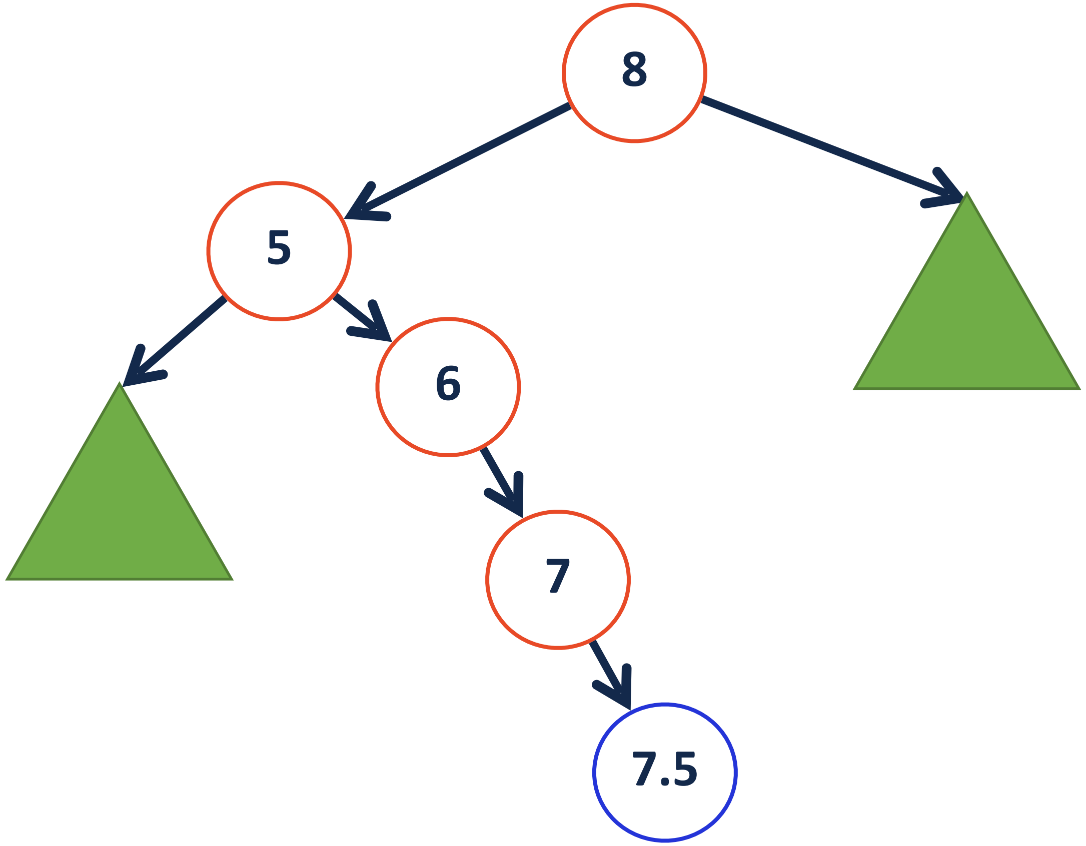
3) tmp->right = root->left

4) root->left = tmp



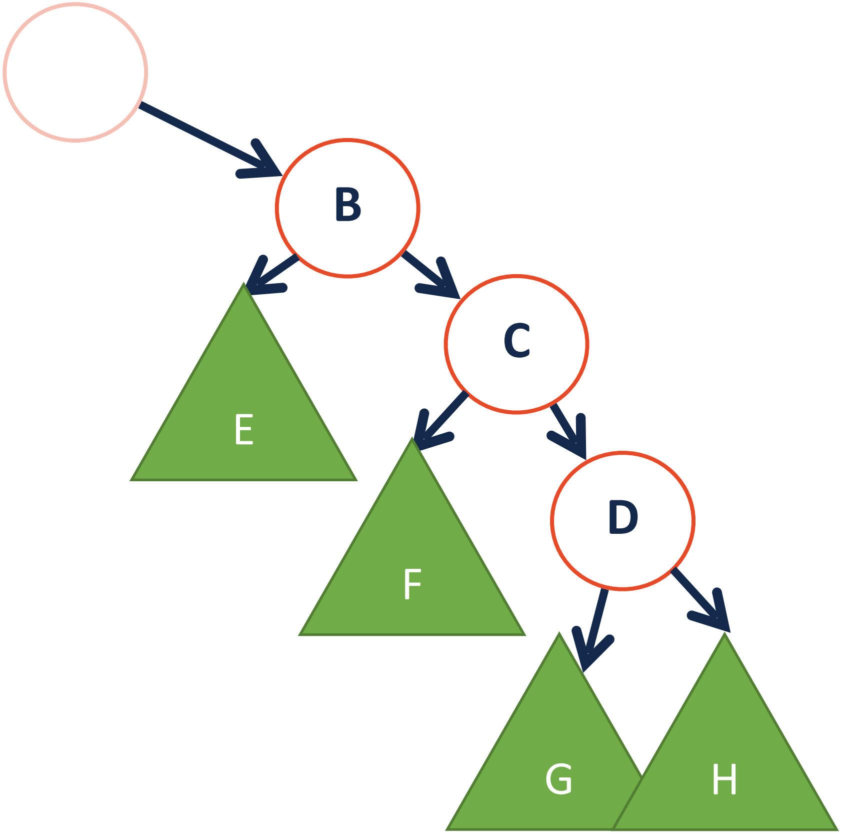
Left Rotation

All rotations are local (subtrees are not impacted)

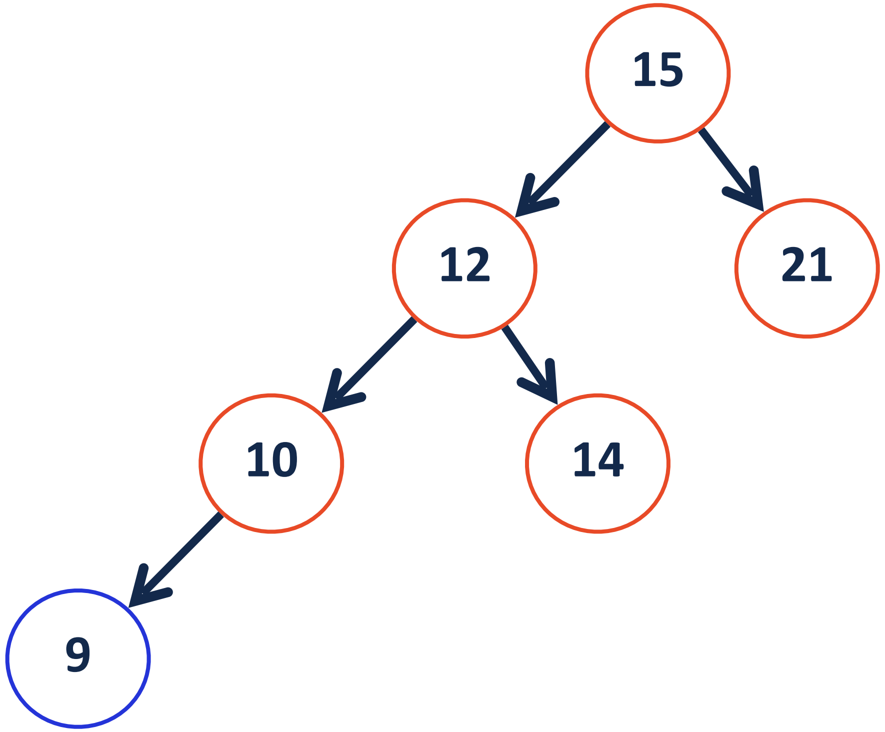


Left Rotation

All rotations preserve BST property



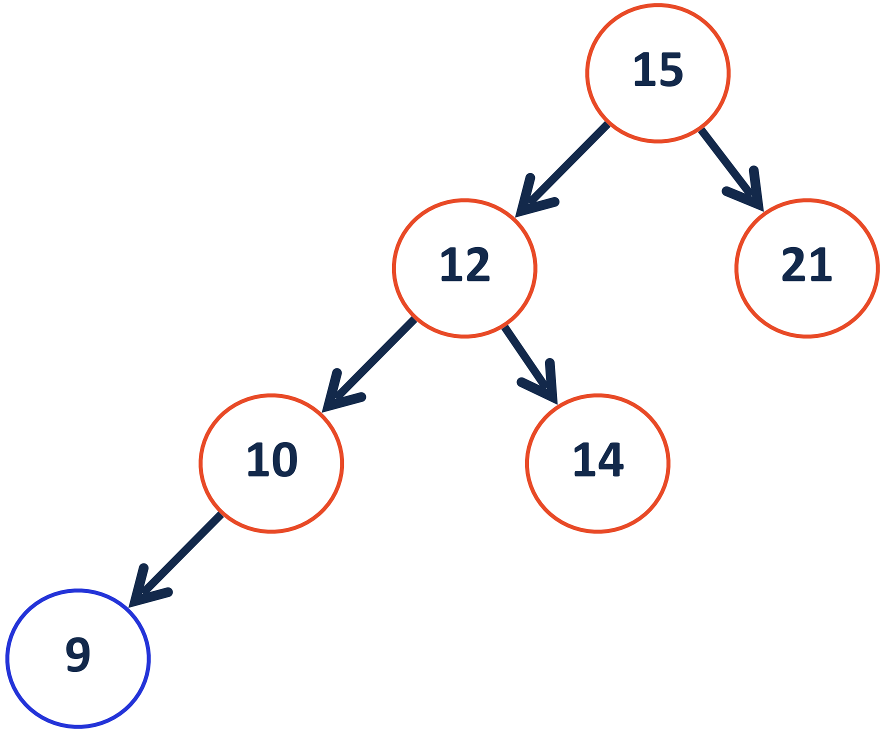
Right Rotation

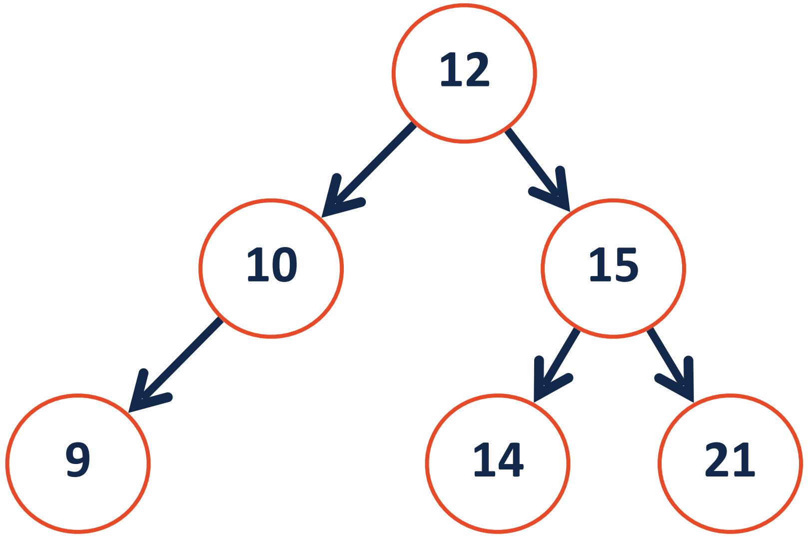


1) Create a tmp pointer to root
2) Update root to point to mid
3) tmp->left = root->right
4) root->right = tmp

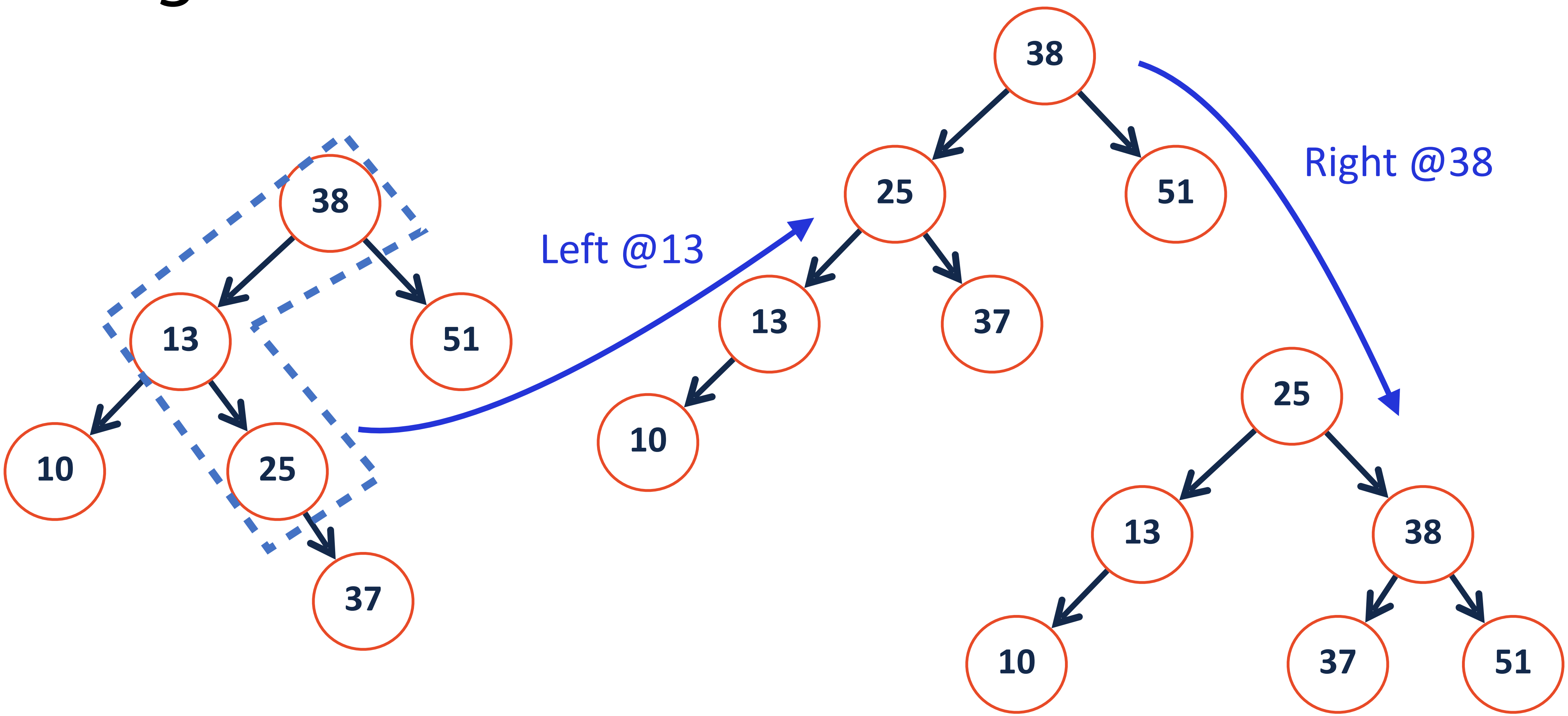
Right Rotation

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

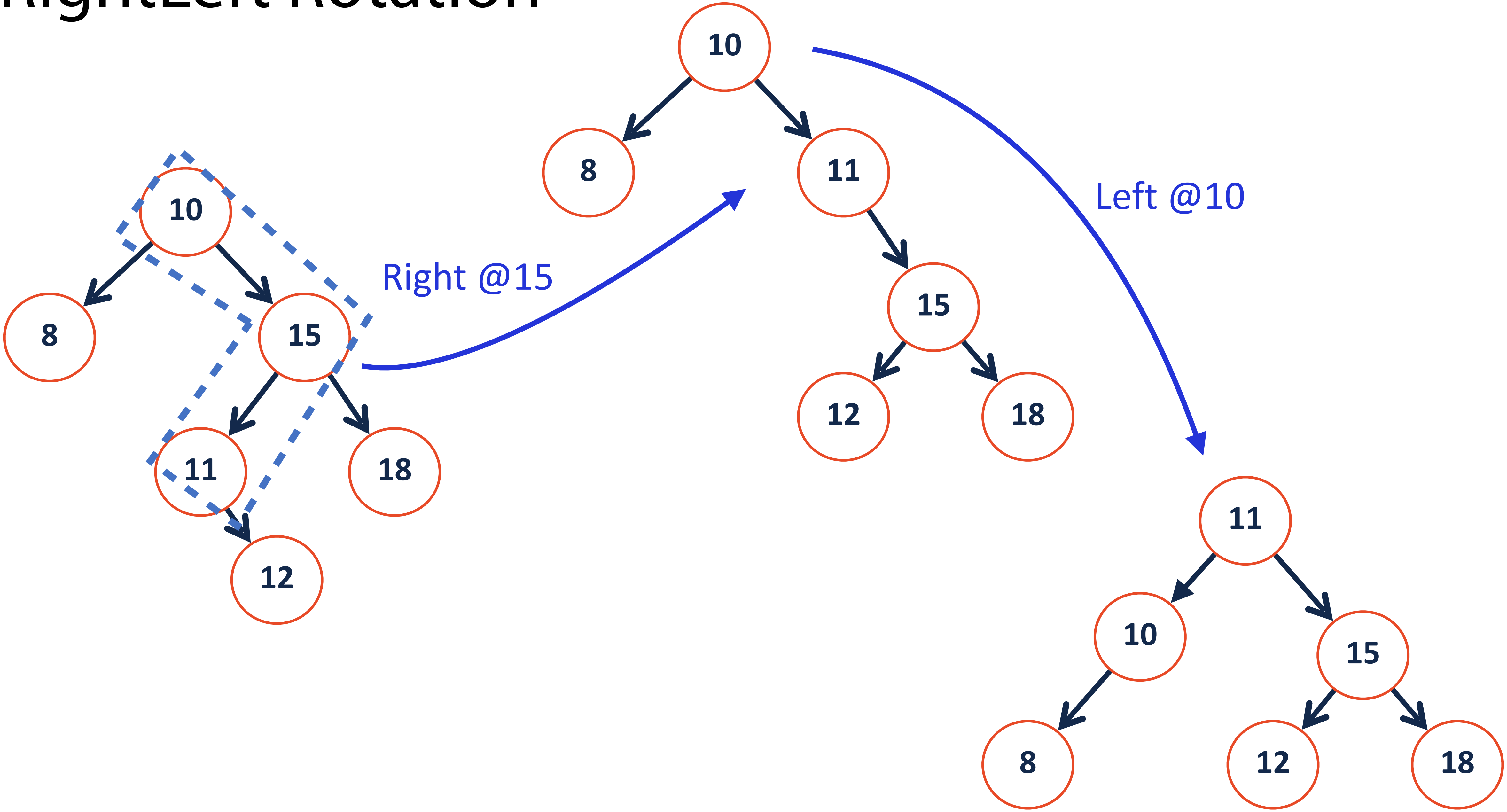




LeftRight Rotation



RightLeft Rotation



AVL Rotations

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

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

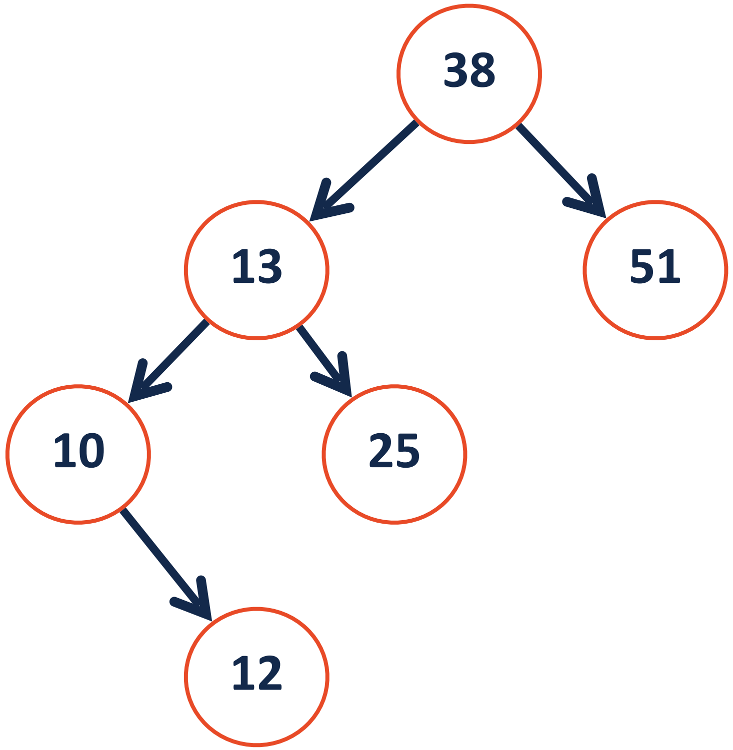
2. The running time of rotations are constant

3. The rotations maintain BST property

Goal:

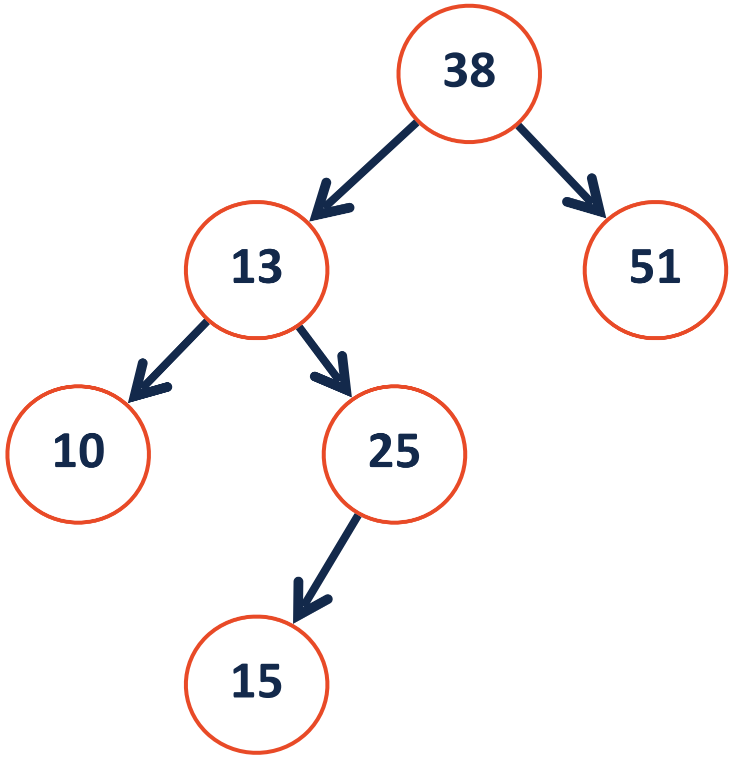
AVL Rotations

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

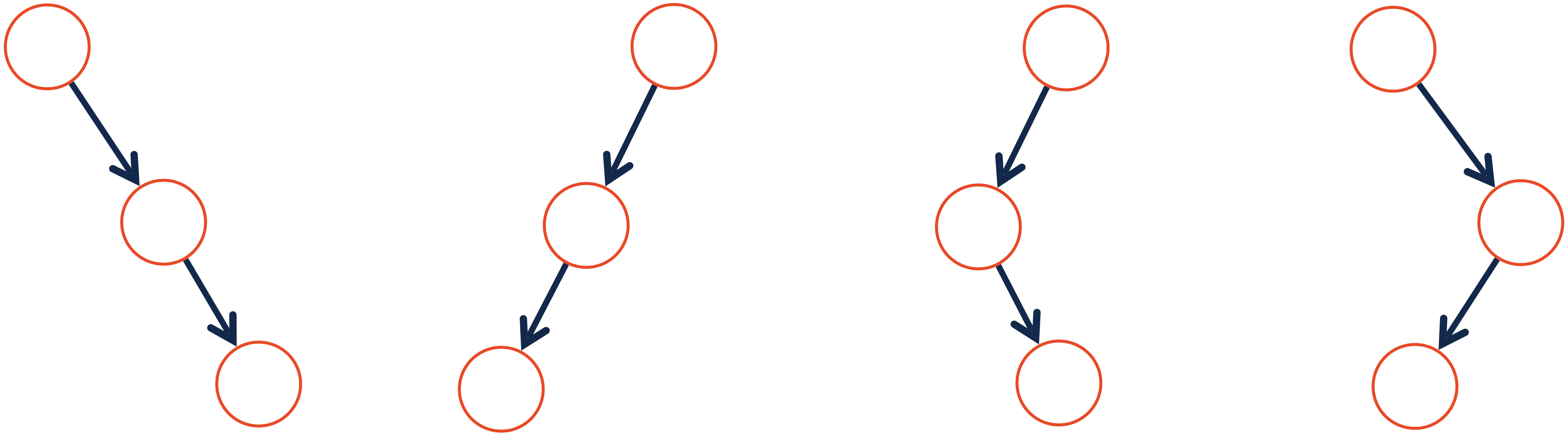


AVL Rotations

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

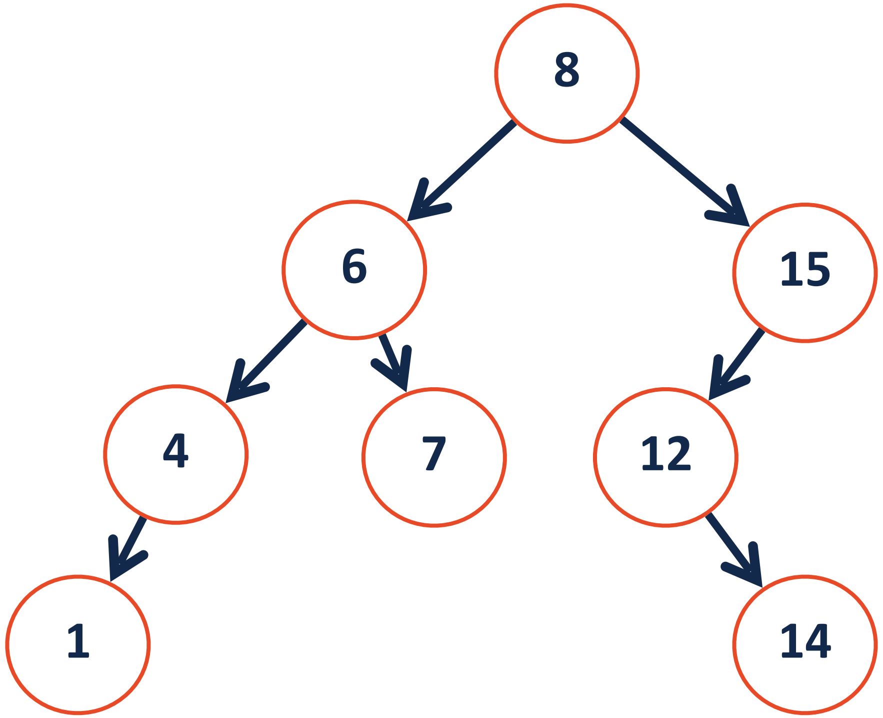


AVL Rotations



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

AVL Rotation Practice



AVL vs BST ADT

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

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

How does the constraint on balance affect the core functions?

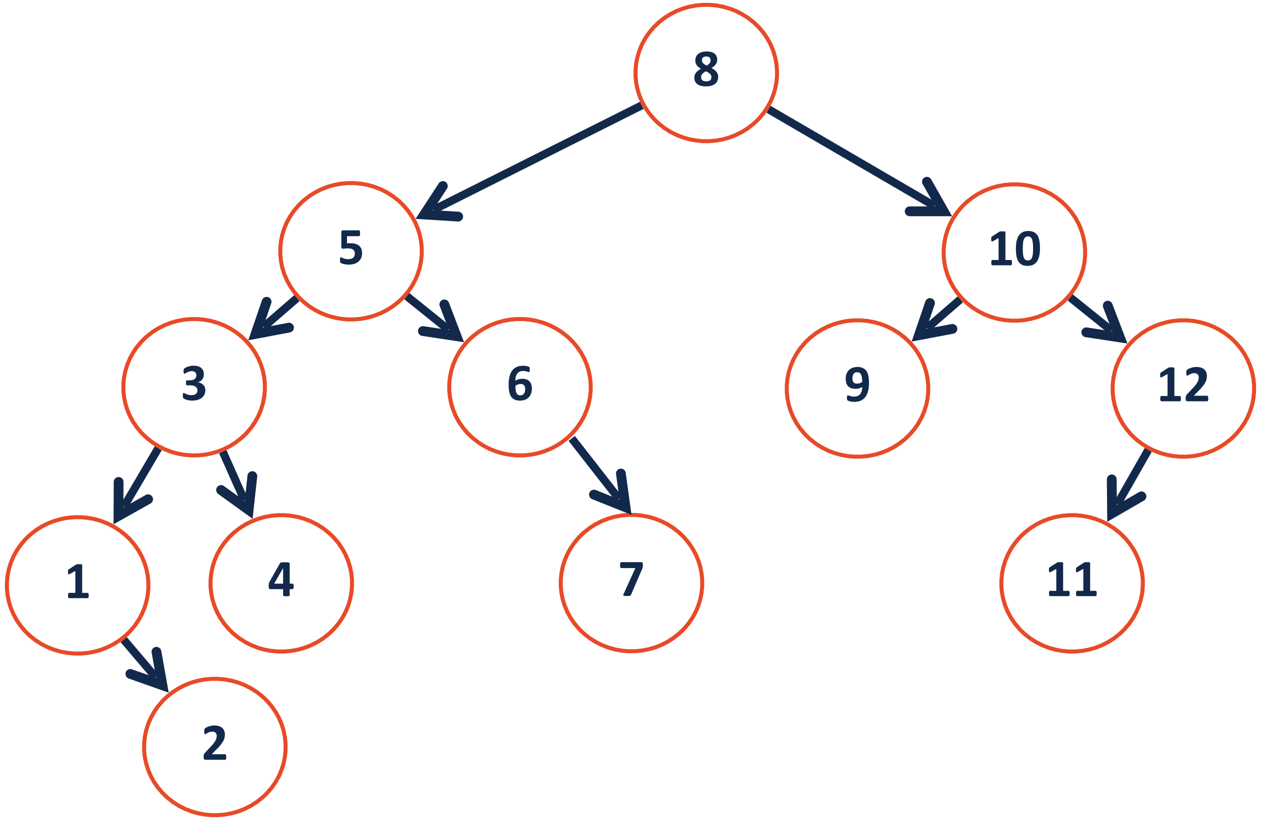
Find

Insert

Remove

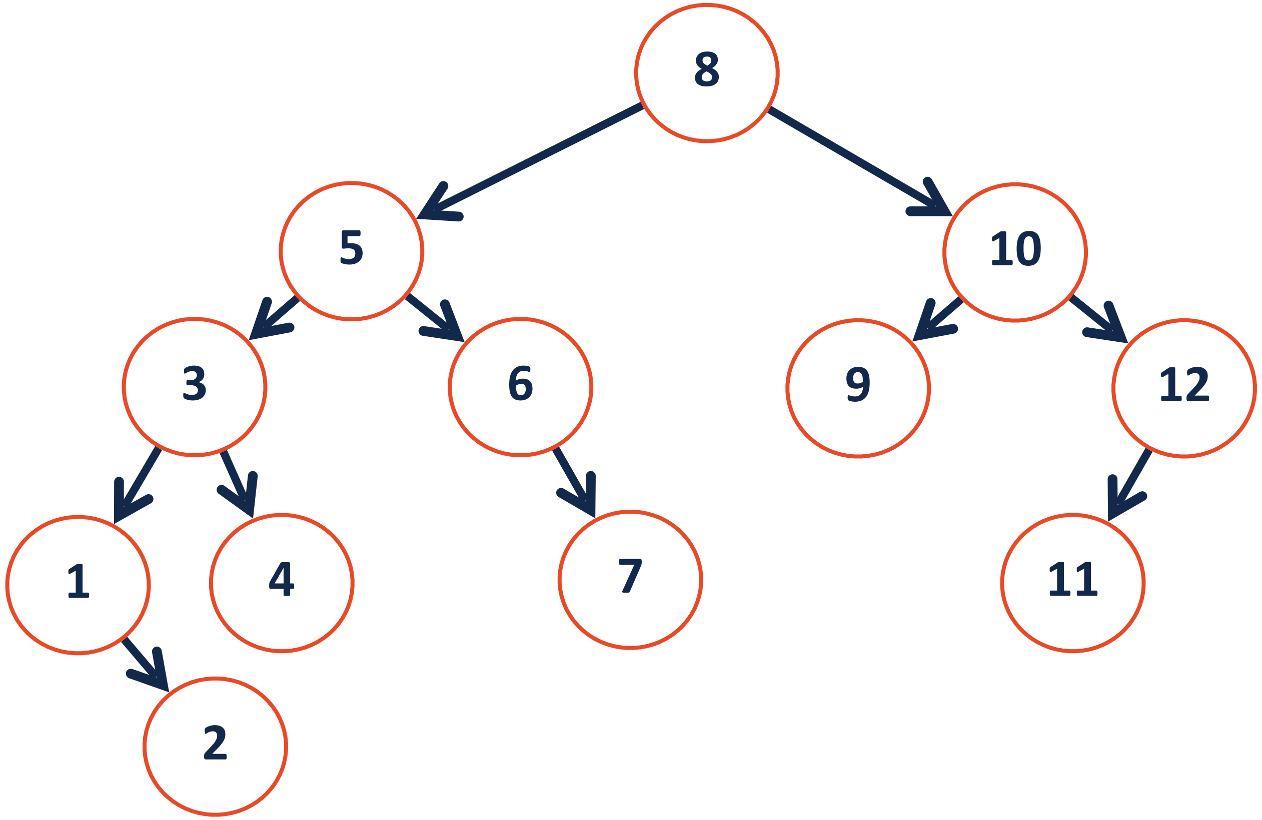
AVL Find

`_find(7)`



AVL Insertion

`_insert(6.5)`



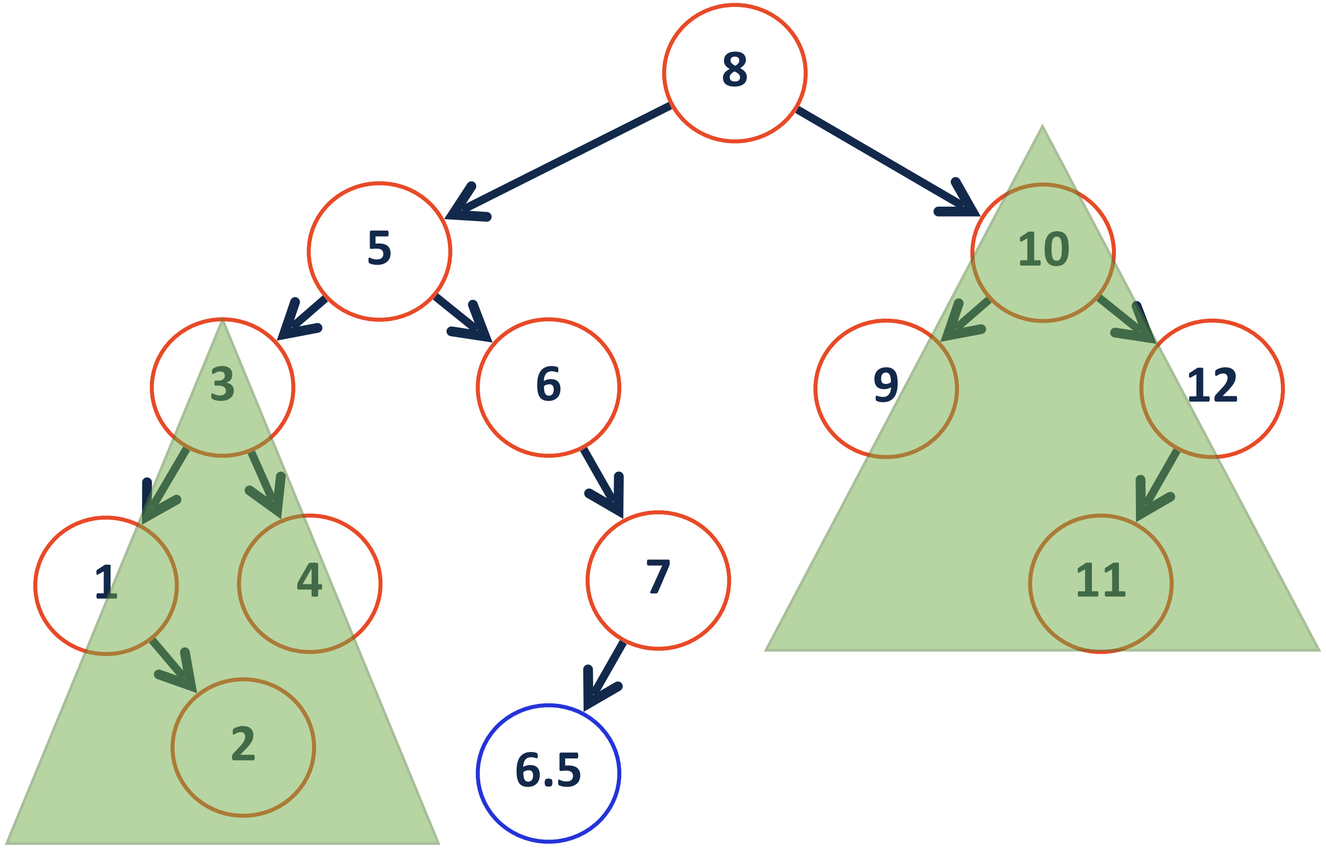

```
struct TreeNode {
  T key;
  unsigned height;
  TreeNode *left;
  TreeNode *right;
};
```

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);
}
```

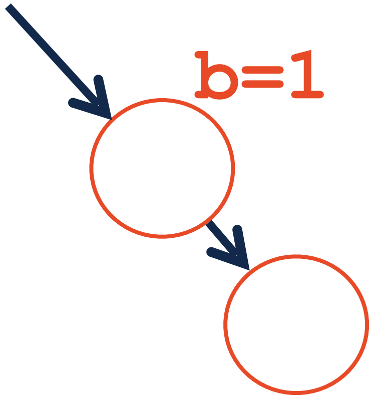
```
template <typename K, typename V>
void AVL<K, D>::_ensureBalance(TreeNode *& cur) {
  // Calculate the balance factor:
  int balance = height(cur->right) - height(cur->left);

  // Check if the node is current not in balance:
  if ( balance == -2 ) {
    int l_balance =
          height(cur->left->right) - height(cur->left->left);
    if ( l_balance == -1 ) { ____________________________; }
    else                   { ____________________________; }
  } else if ( balance == 2 ) {
    int r_balance =
          height(cur->right->right) - height(cur->right->left);
    if( r_balance == 1 ) { ____________________________; }
    else                 { ____________________________; }
  }

  _updateHeight(cur);
};
```

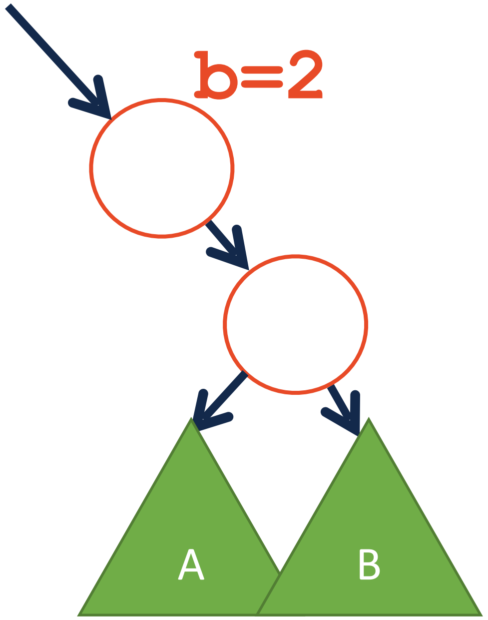
AVL Insertion

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



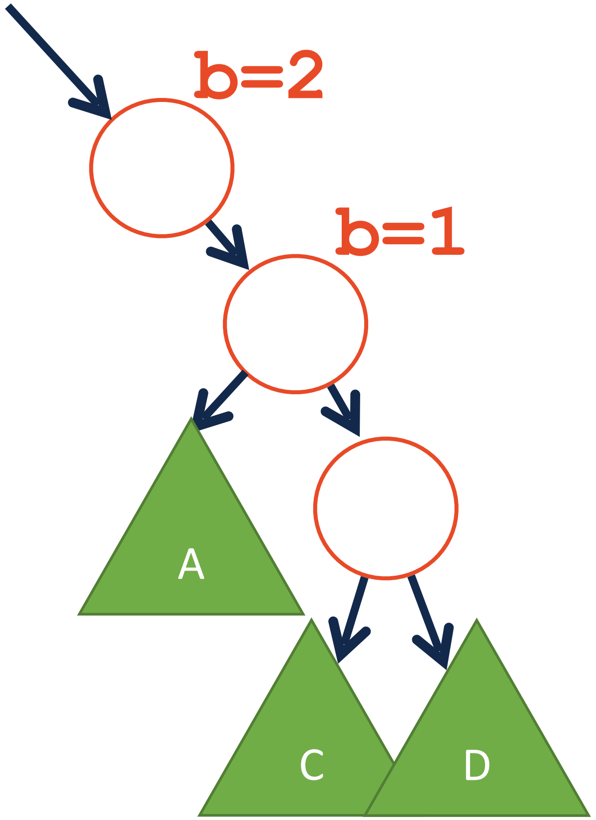
AVL Insertion

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



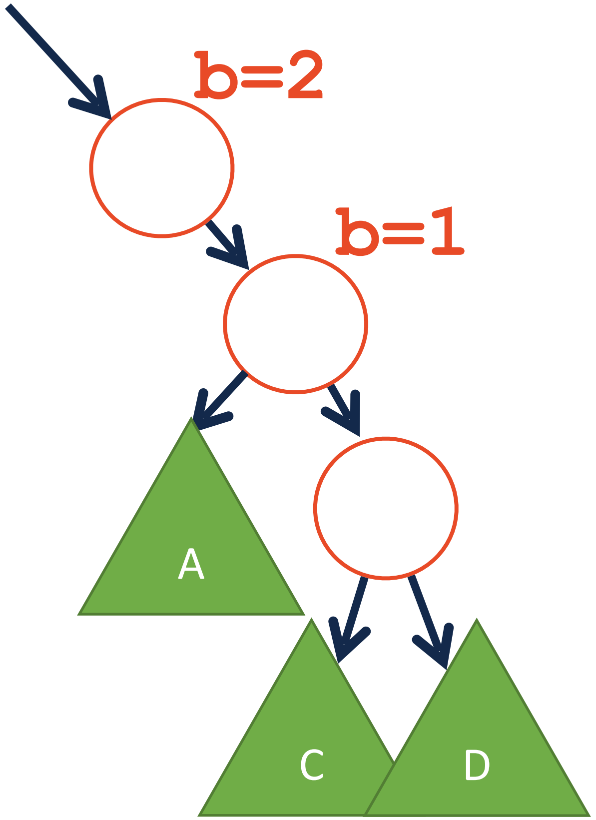
AVL Insertion

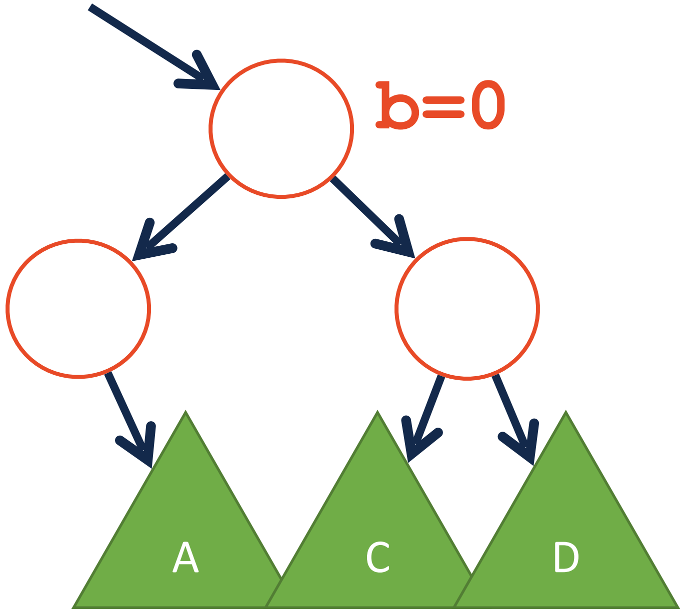
If we insert in B, I must have a balance pattern of **2, 1**



AVL Insertion

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

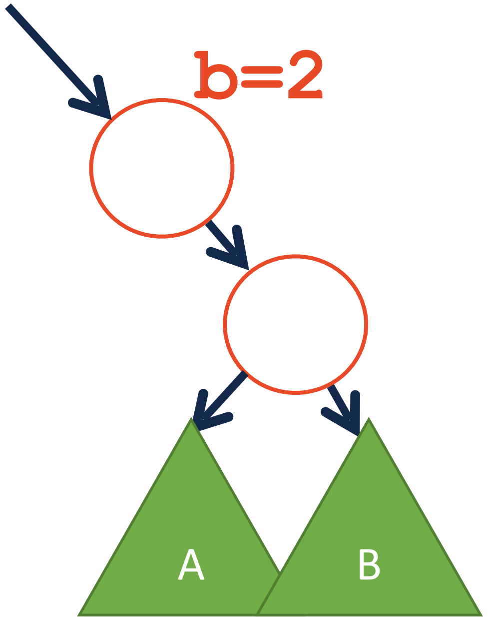




After rotation, subtree has **pre-insert height**. (Overall tree is balanced)

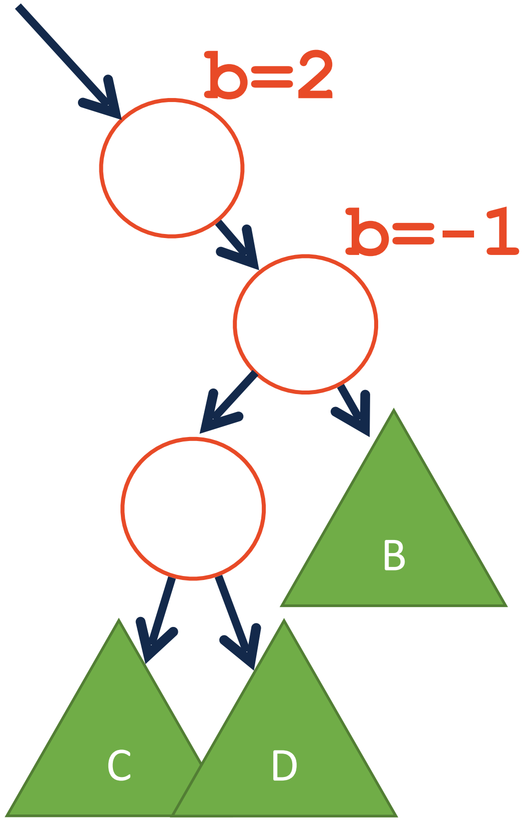
AVL Insertion

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



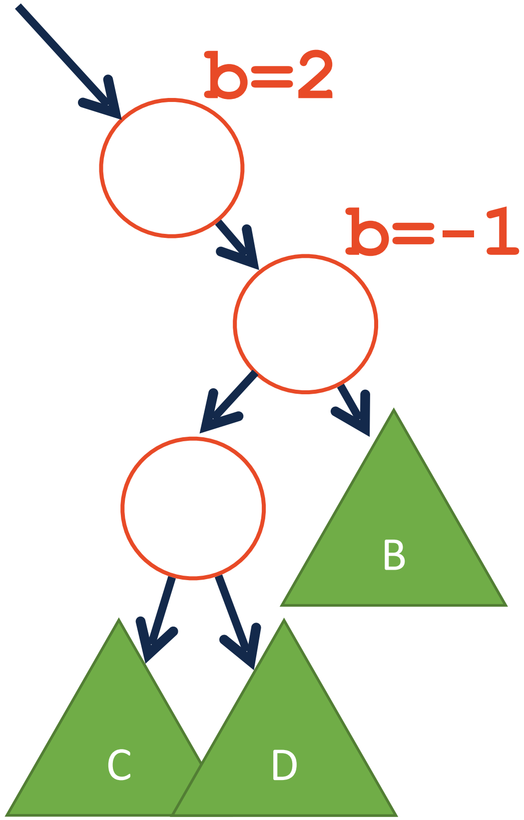
AVL Insertion

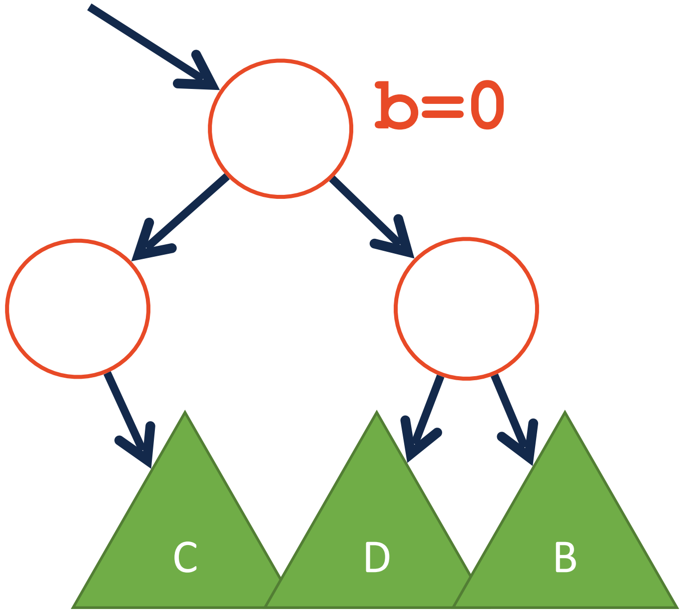
If we insert in A, I must have a balance pattern of **2, -1**



AVL Insertion

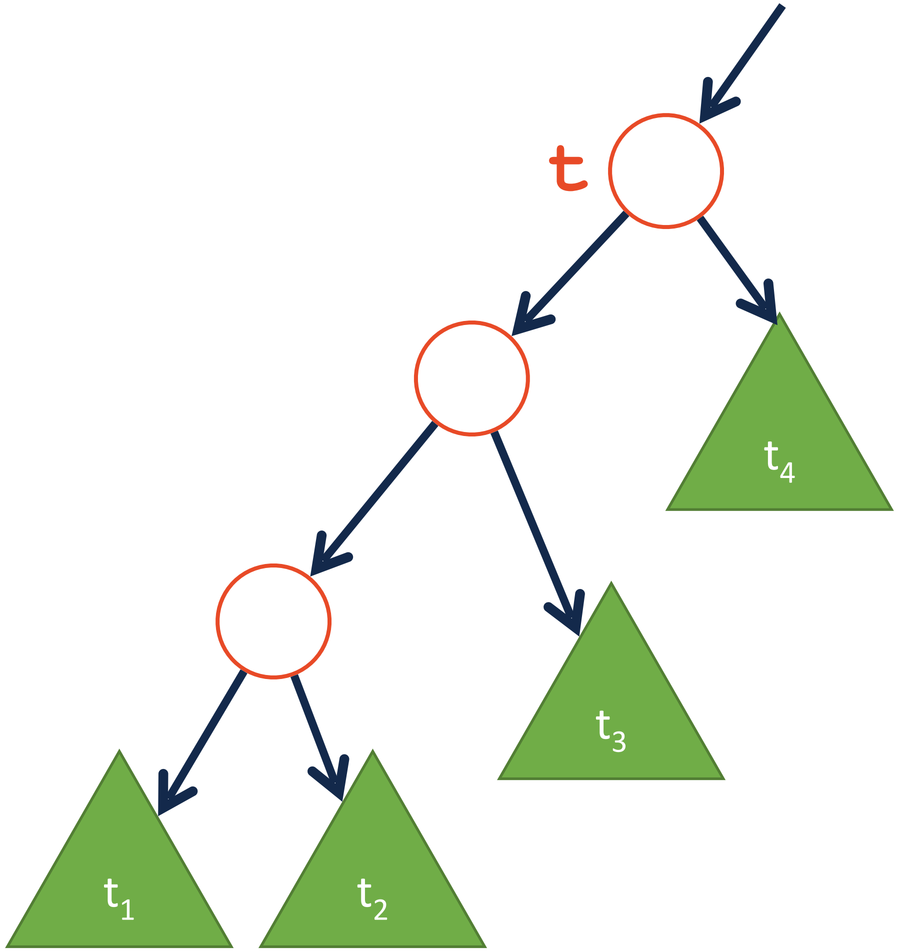
A **rightLeft** rotation fixes our imbalance in our local tree.





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

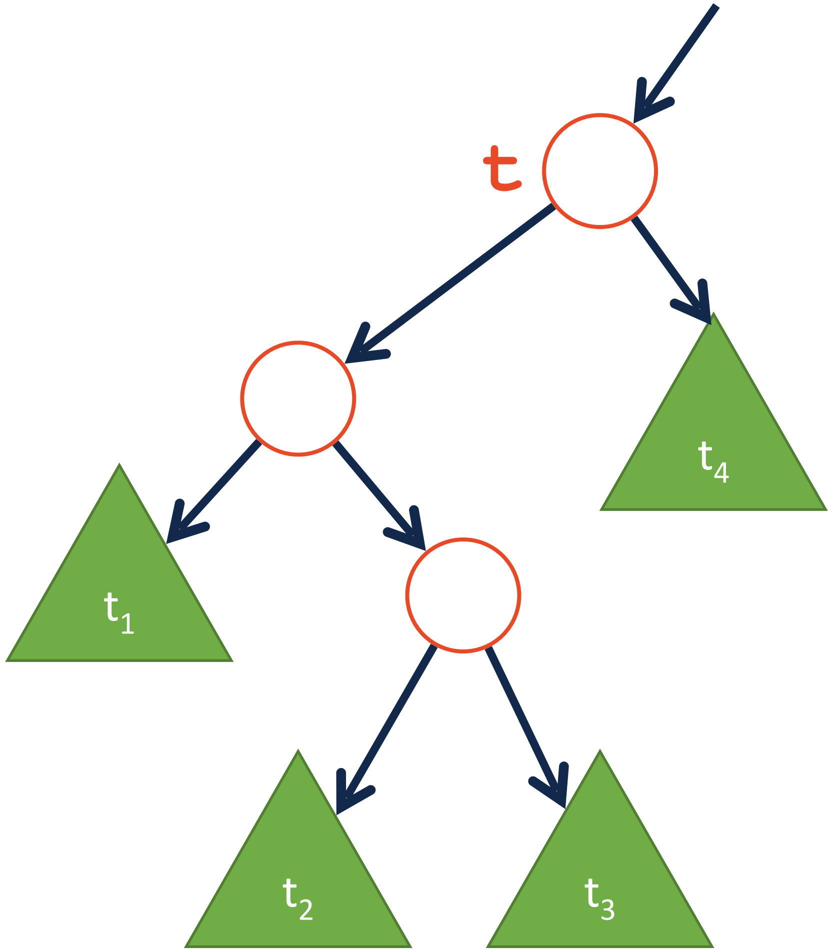
AVL Insertion



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

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

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 _______________ rotation about **t** restores the balance of the tree.

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

AVL Insertion

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

Insert *may* increase height by at most:

A rotation reduces the height of the subtree by:

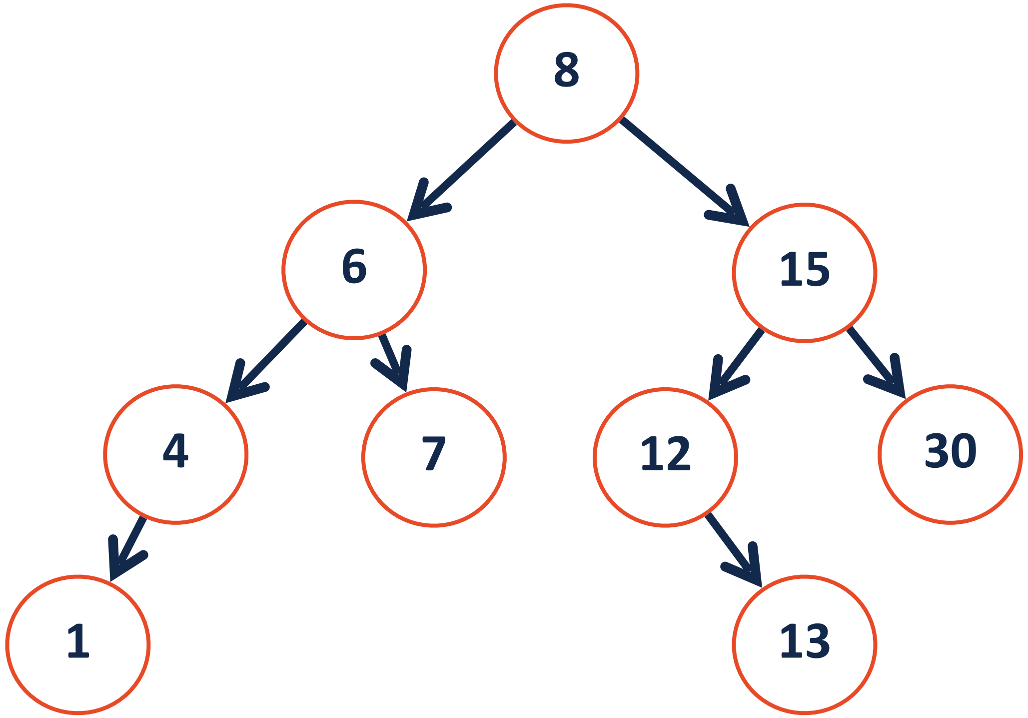
A single* rotation restores balance and corrects height!

What is the Big O of performing our rotation?

What is the Big O of insert?

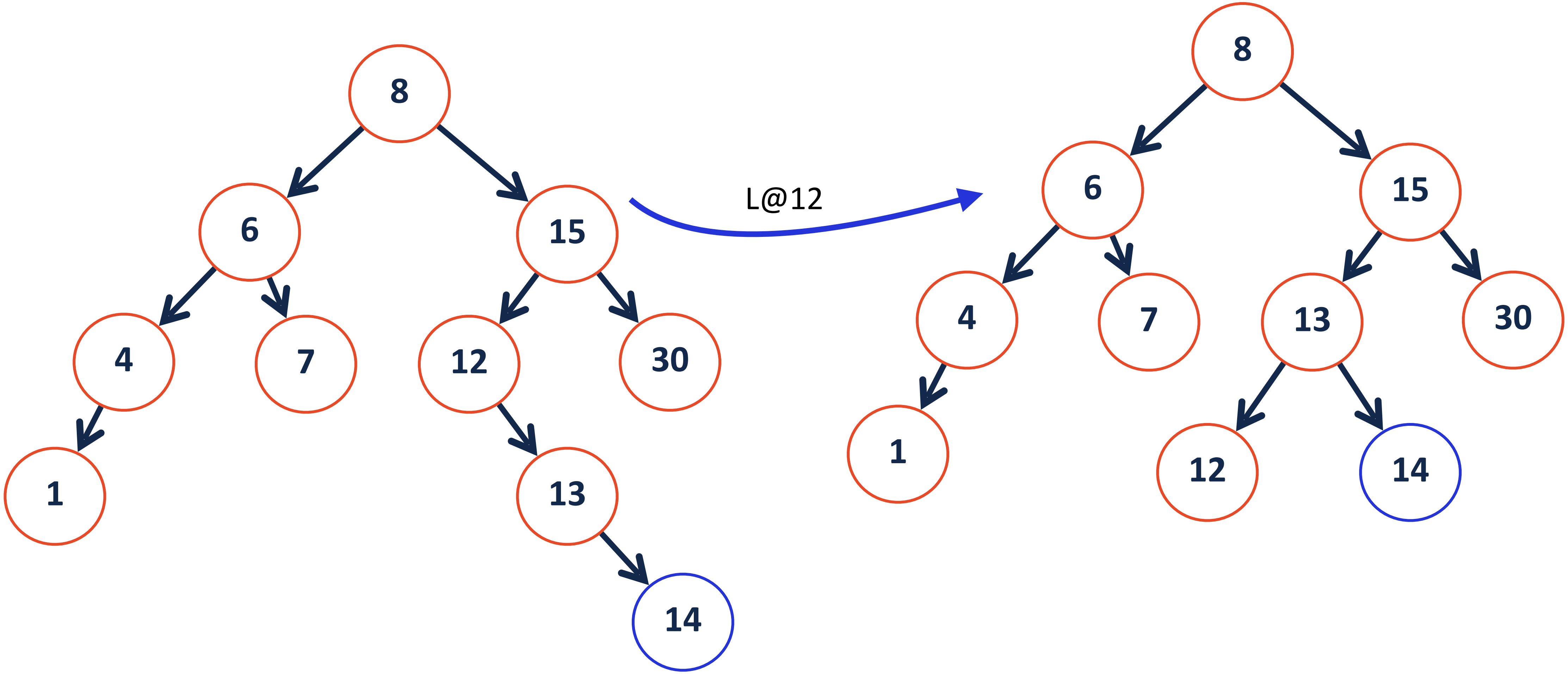
AVL Insertion Practice

`_insert(14)`



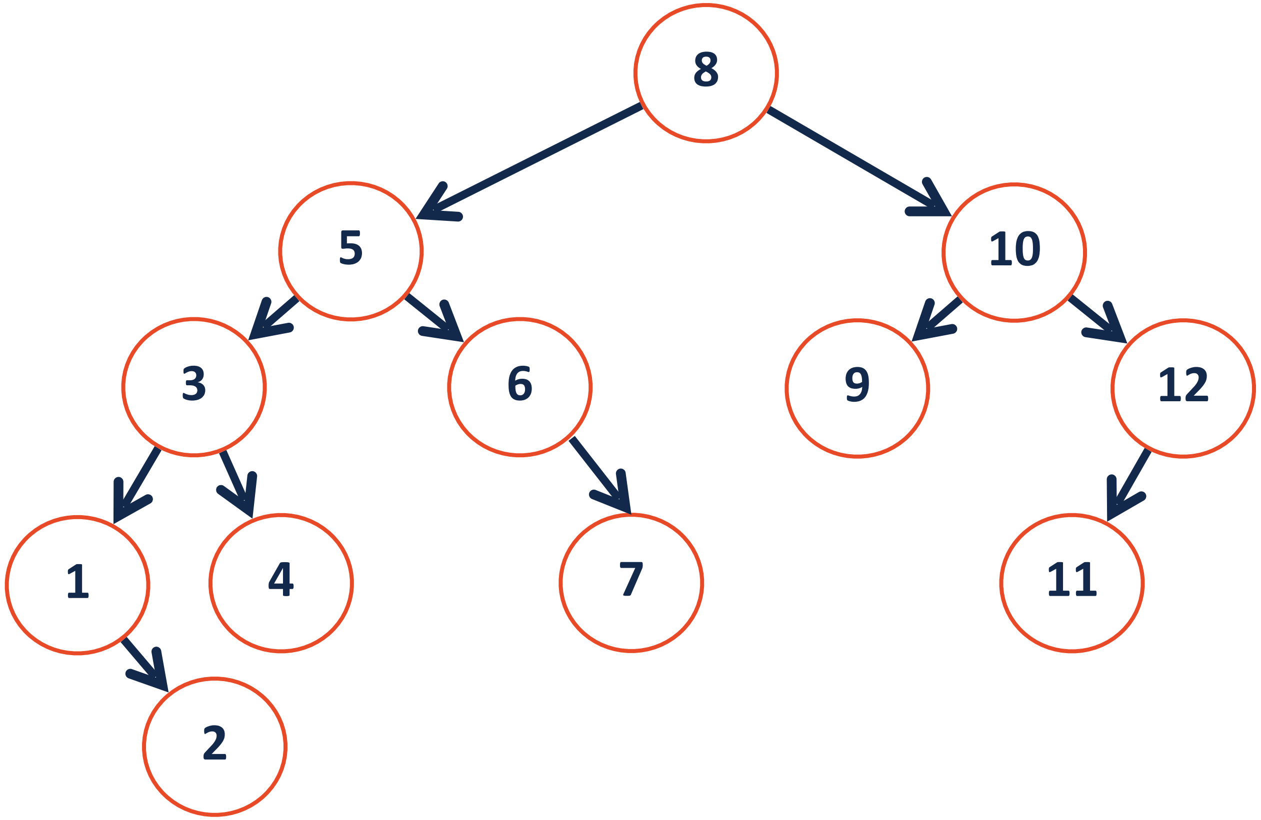
AVL Insertion Practice

`_insert(14)`



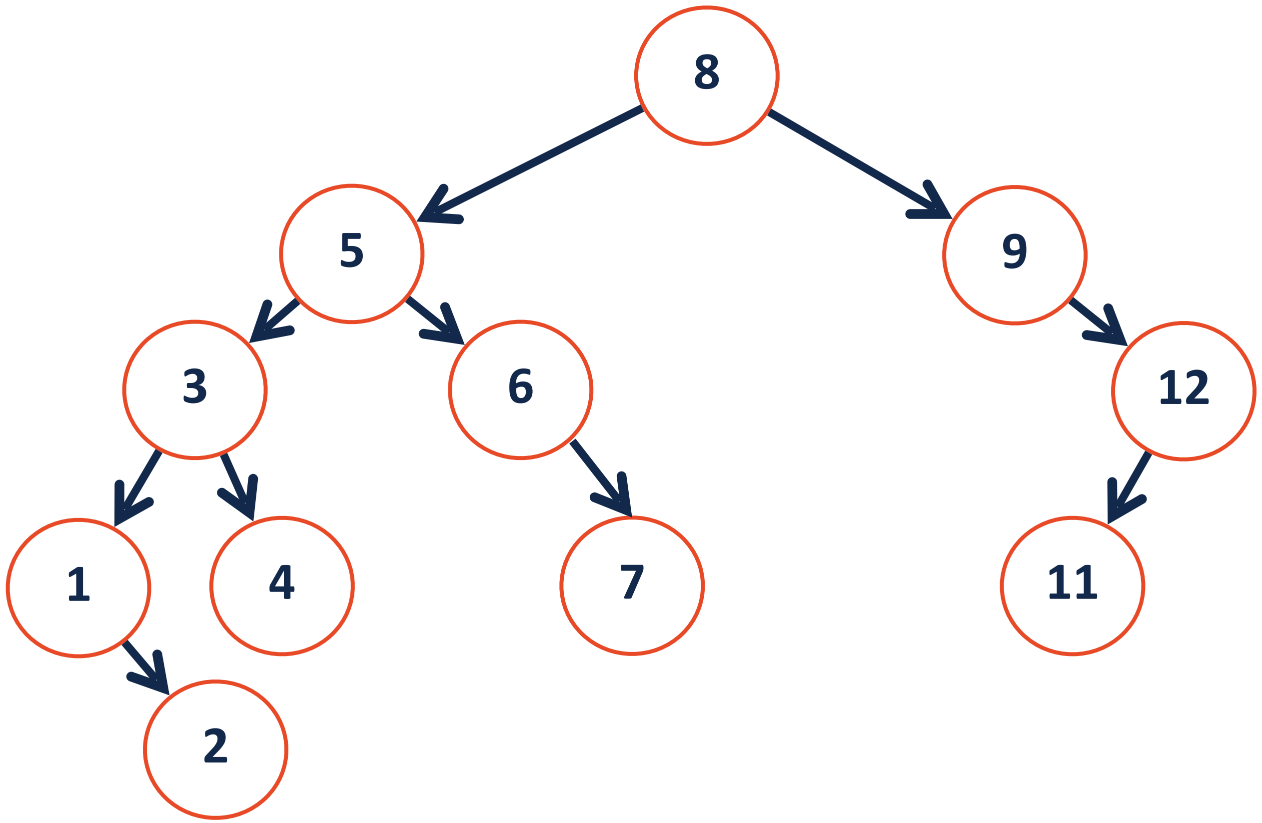
AVL Remove

`_remove(10)`



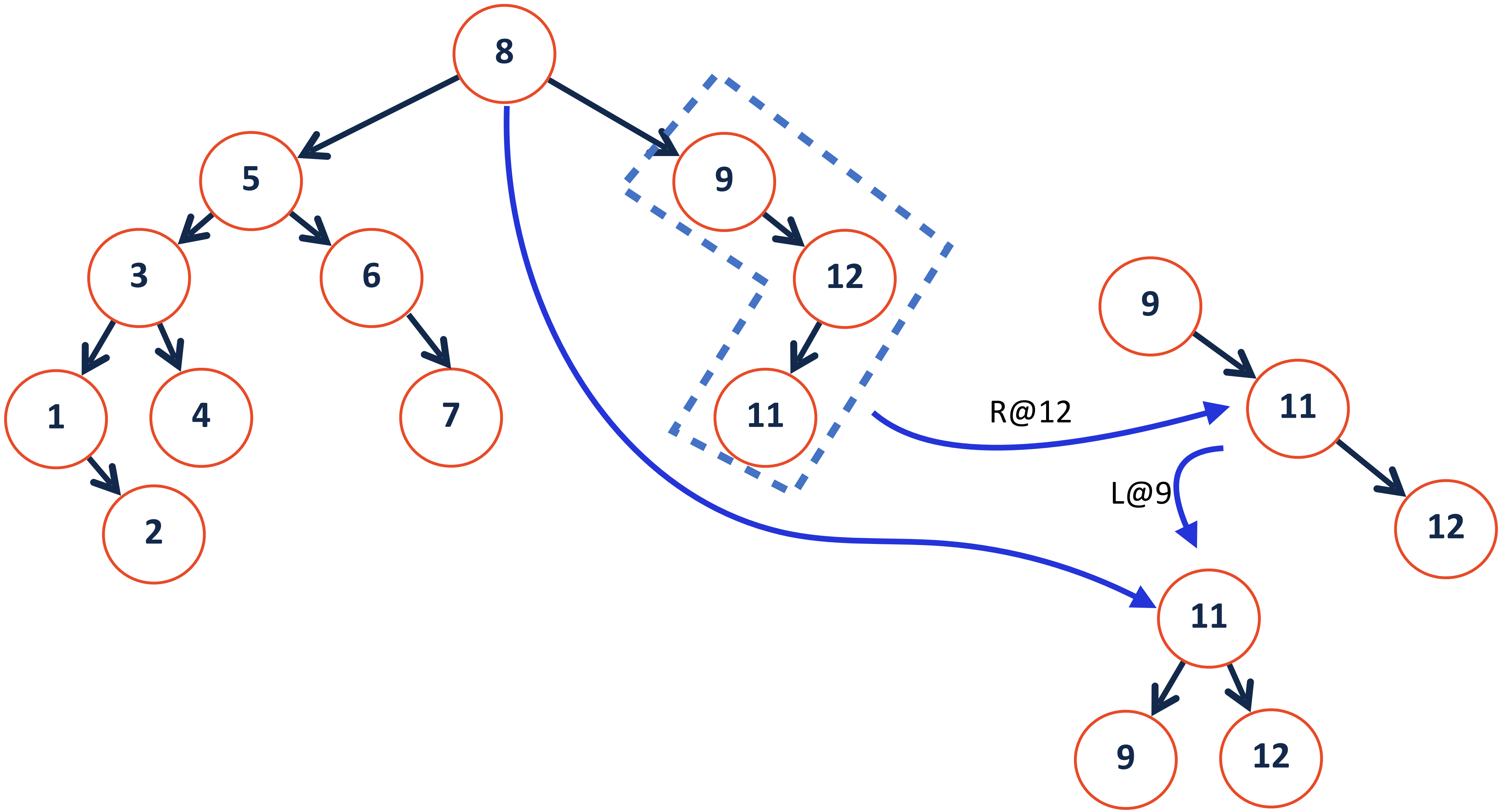
AVL Remove

`_remove(10)`



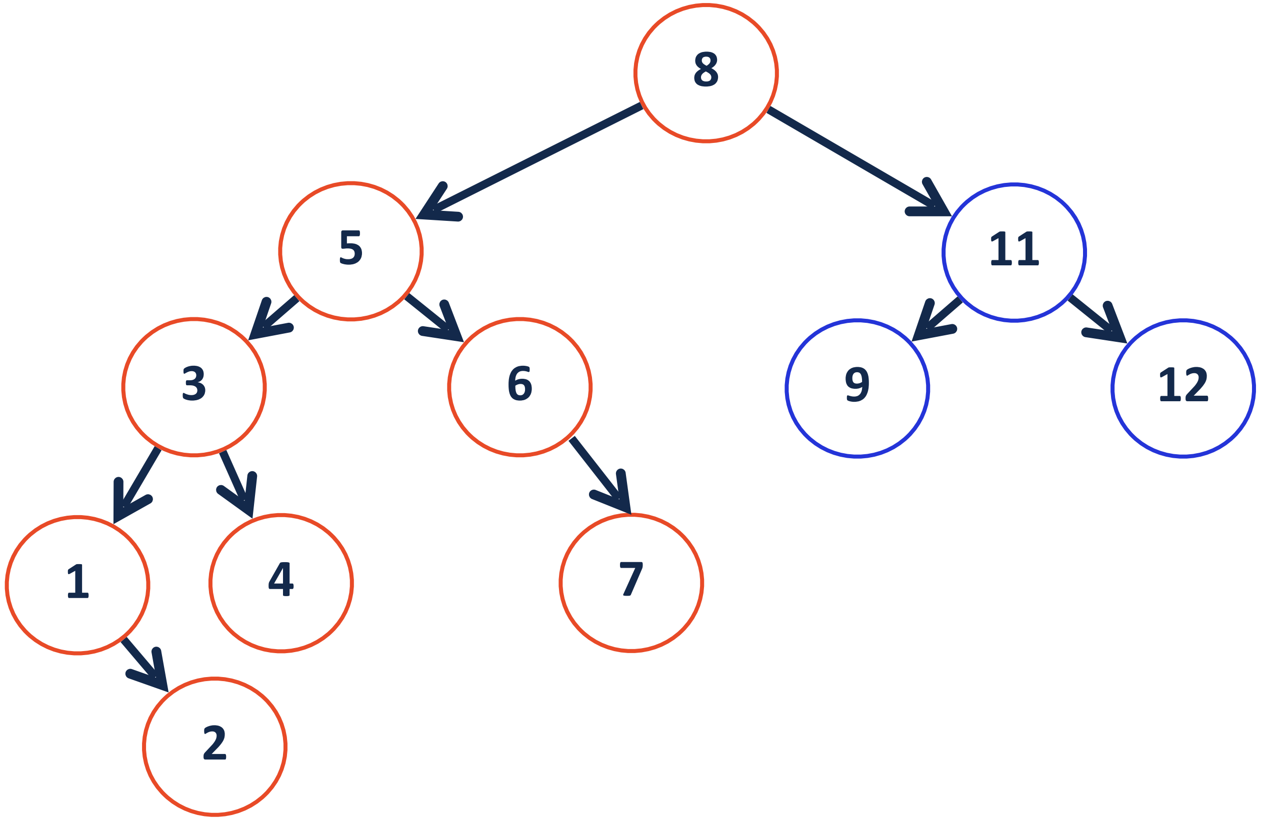
AVL Remove

`_remove(10)`



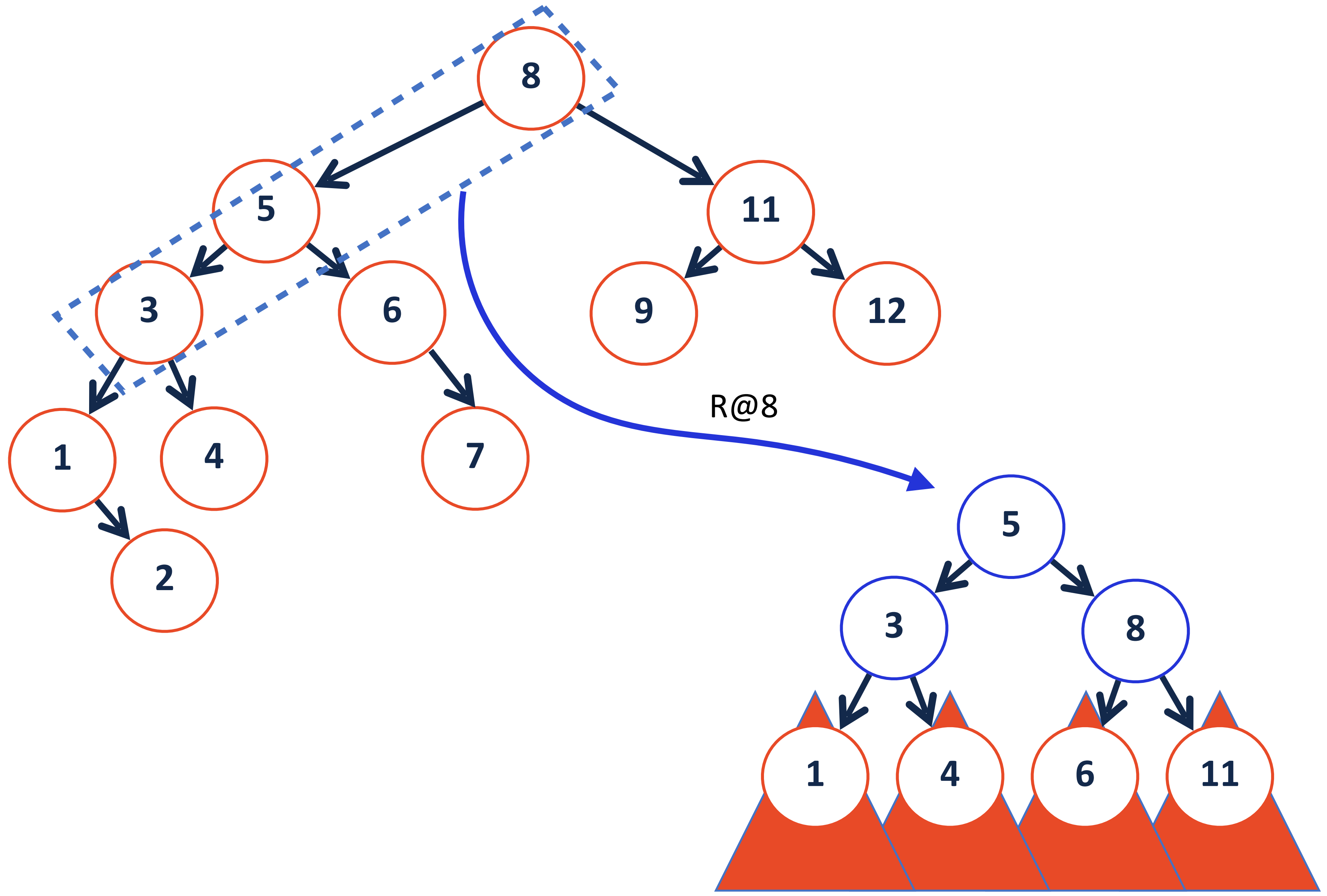
AVL Remove

`_remove(10)`



AVL Remove

`_remove(10)`



AVL Remove

`_remove(10)`

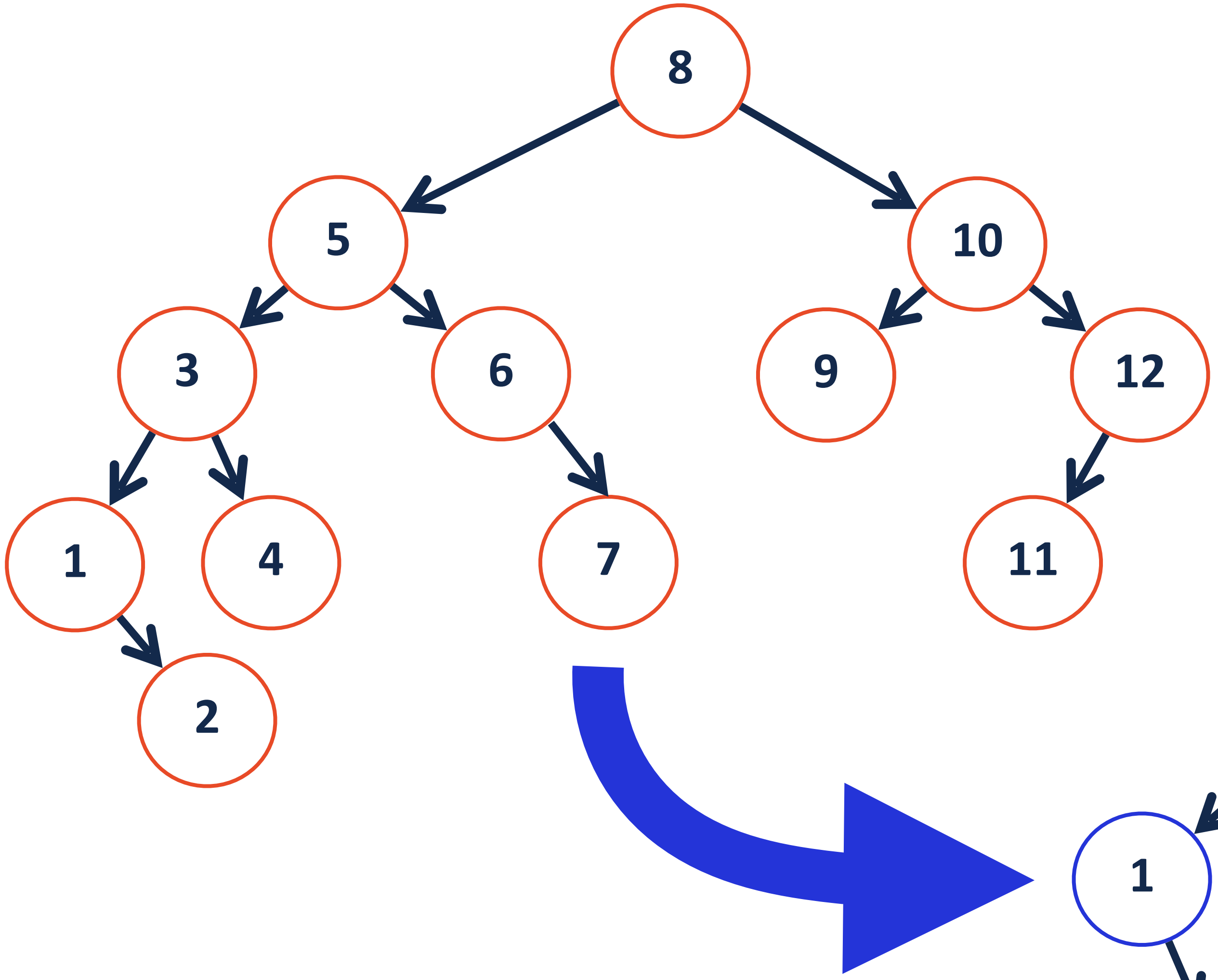
Remove (pseudo code):

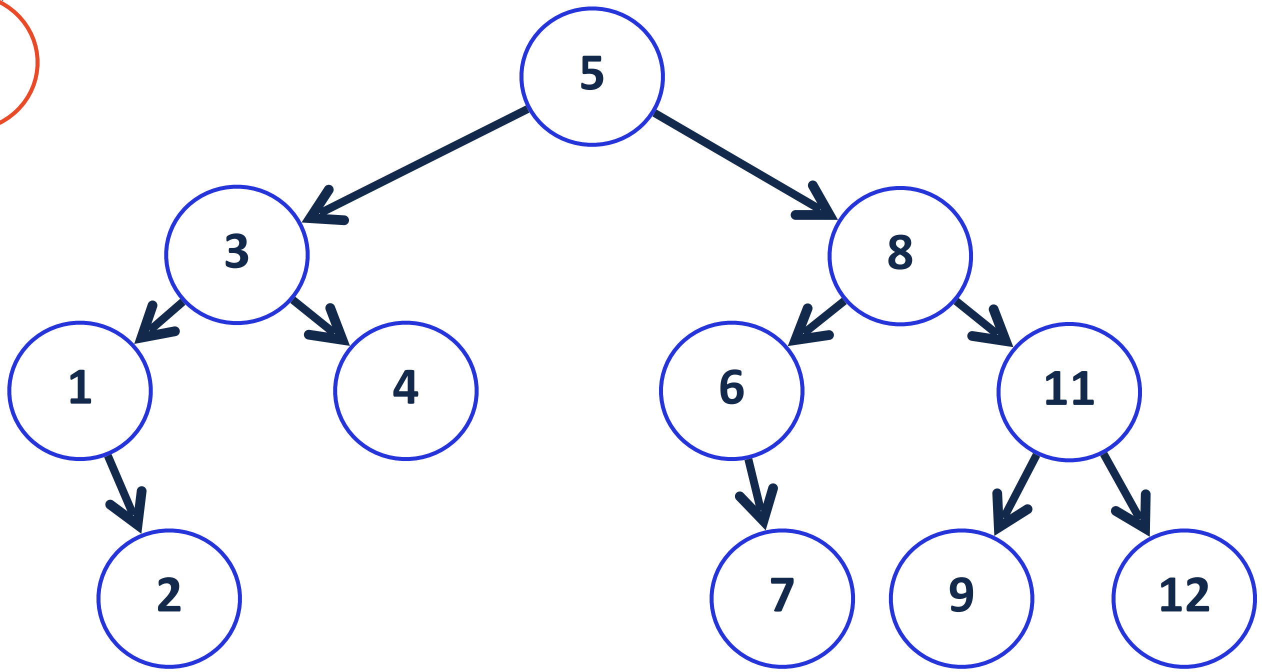
1: Remove at proper place

2: Check for imbalance

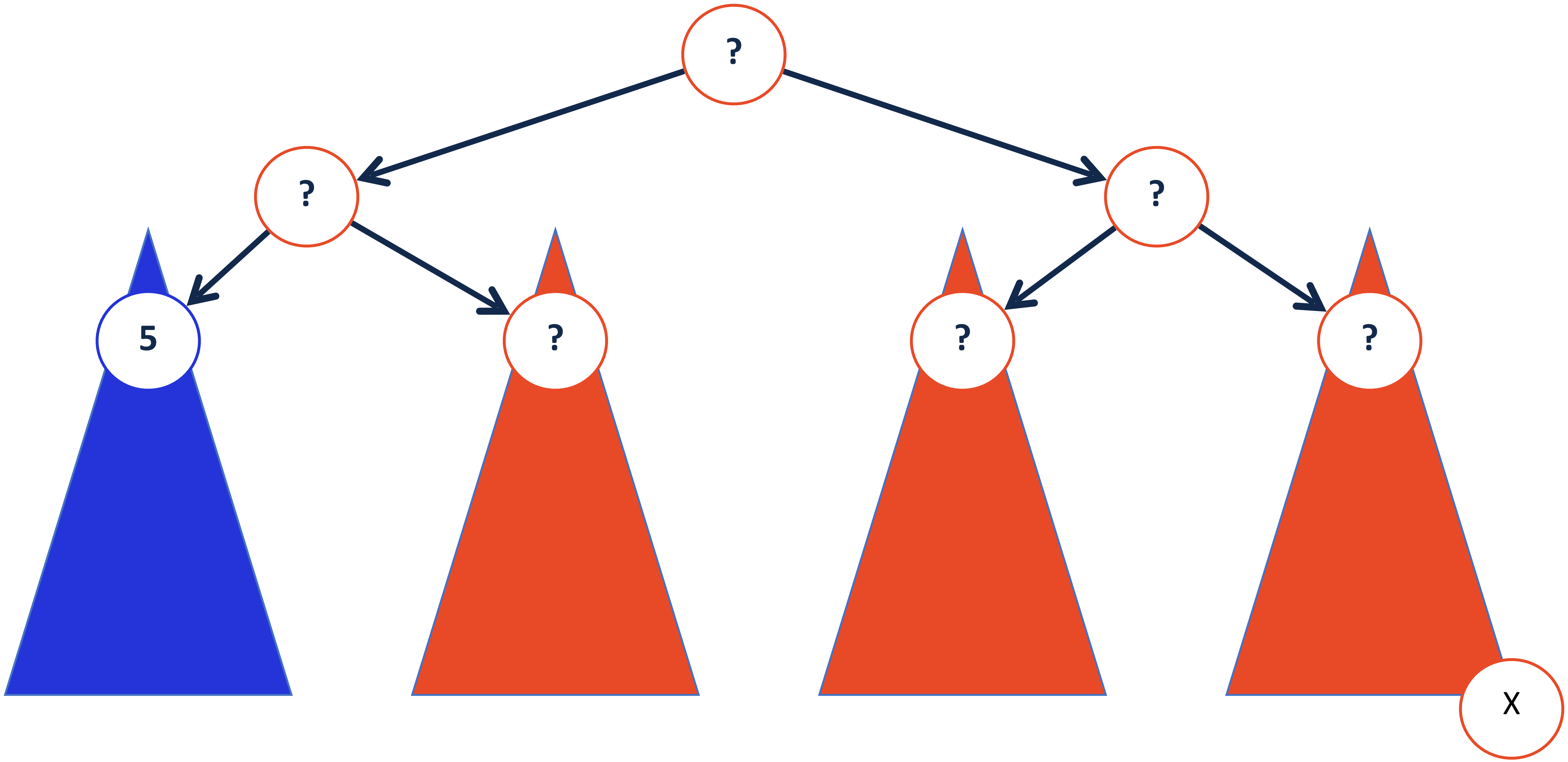
3: Rotate, if necessary

4: Update height





AVL Remove



AVL Remove

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

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: ___________.

Insert runs in: ___________.

Remove runs in: ___________.

Claim: The height of the AVL tree with n nodes is: **___________.**