

Data Structures

AVL Trees

CS 225

September 30, 2026

Brad Solomon



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science

Informal Early Feedback Reminder

Right now we are at ~50% participation

Points are awarded at 70% participation!

If you haven't filled it out yet, fill it out this week!

Learning Objectives

Review why we need balanced trees

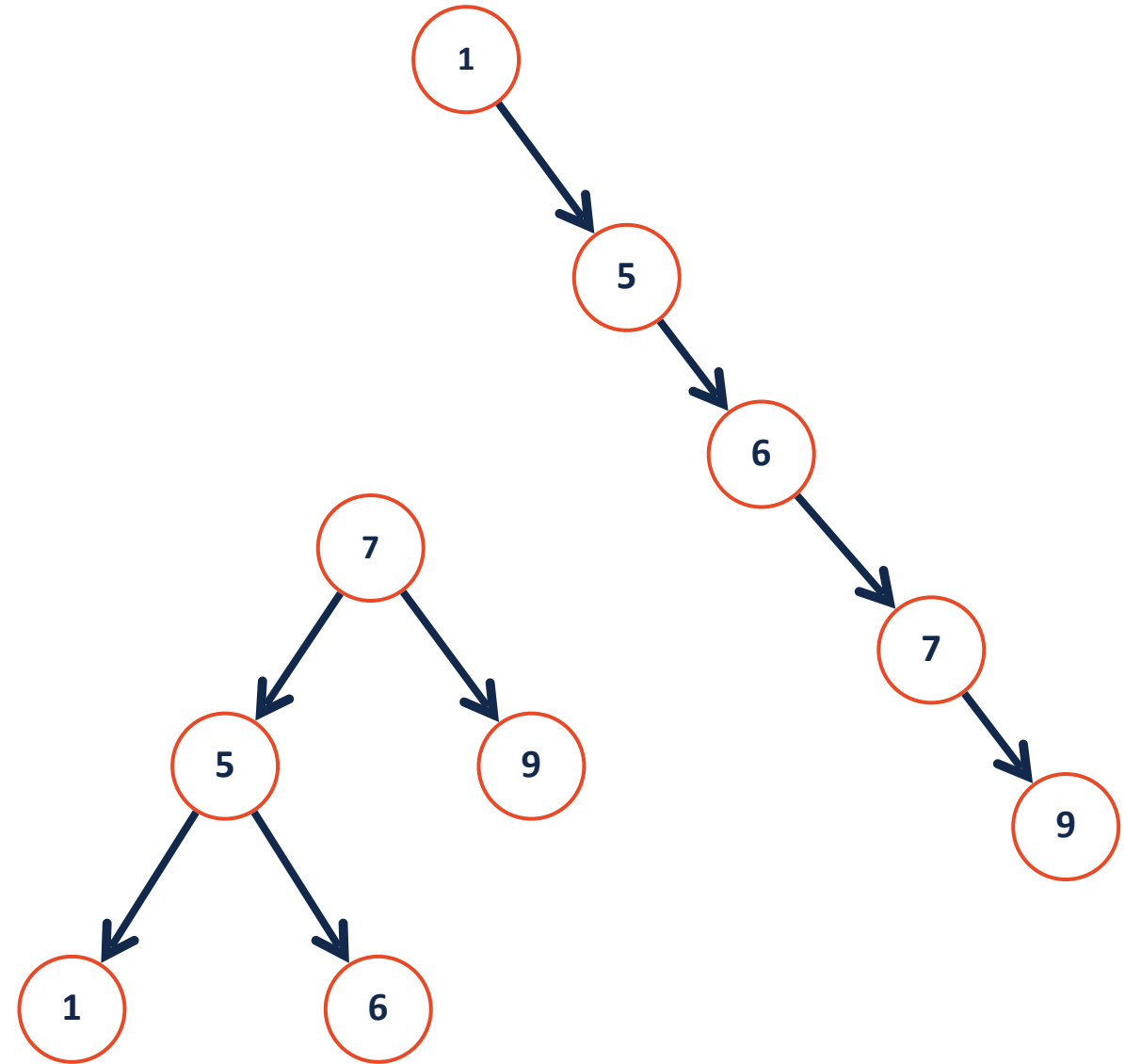
Review what an AVL rotation does

Review the four possible rotations for an AVL tree

Explore the implementation of AVL Tree

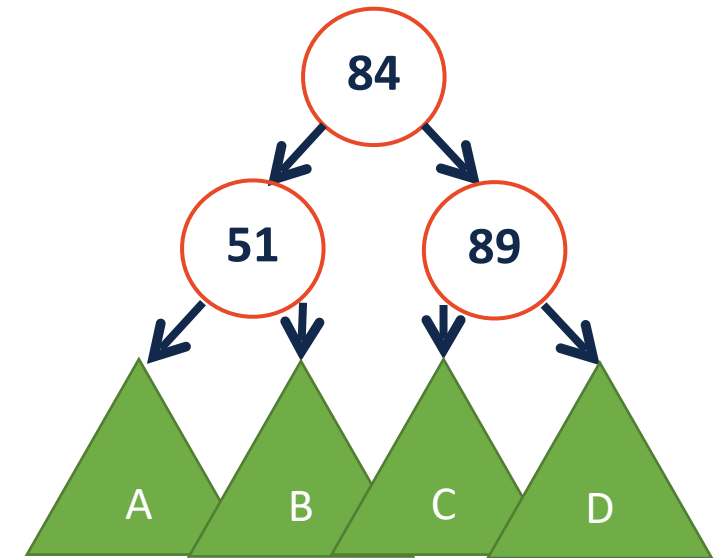
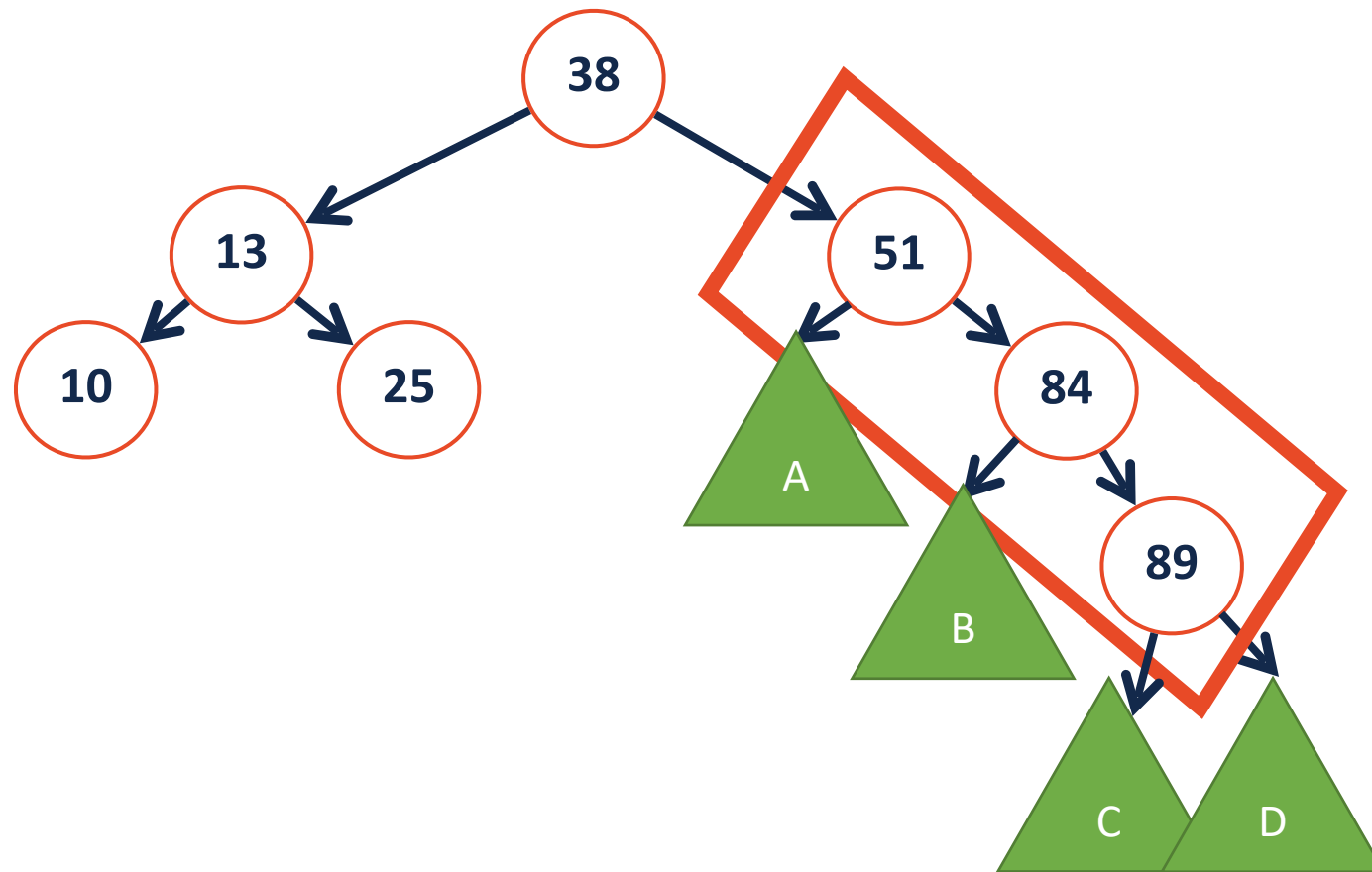
BST Analysis – Running Time

	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!

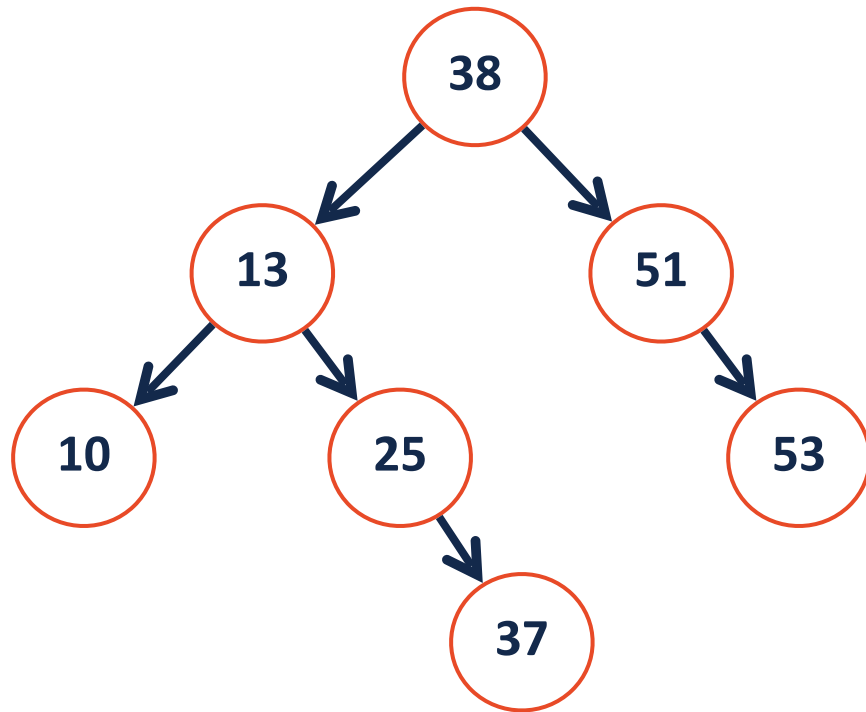


AVL-Tree: A self-balancing binary search tree

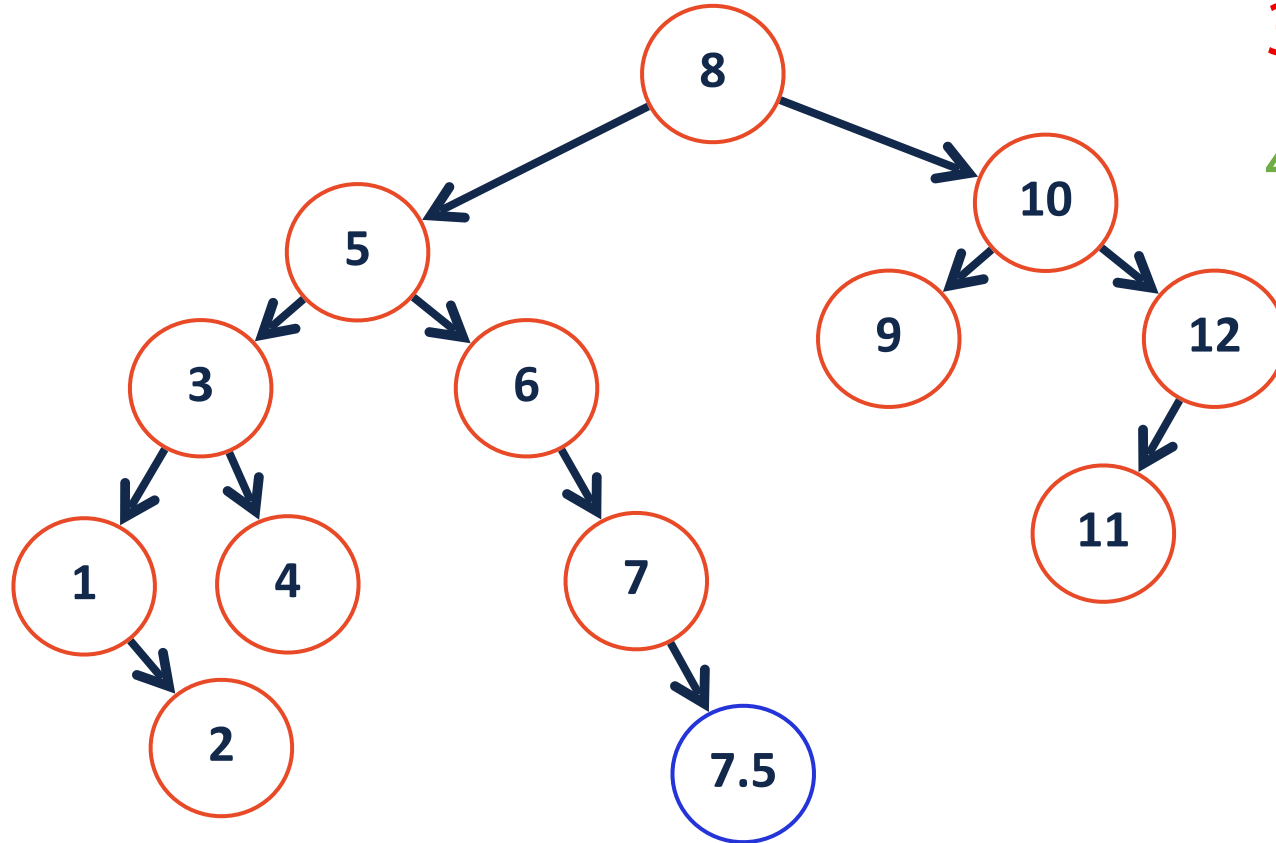


Every node in an AVL tree has a balance of:

Join Code: 225



Left Rotation



1) Create a tmp pointer to root

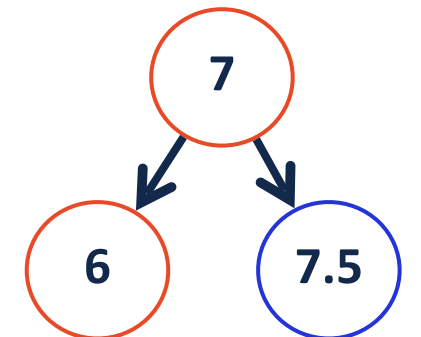
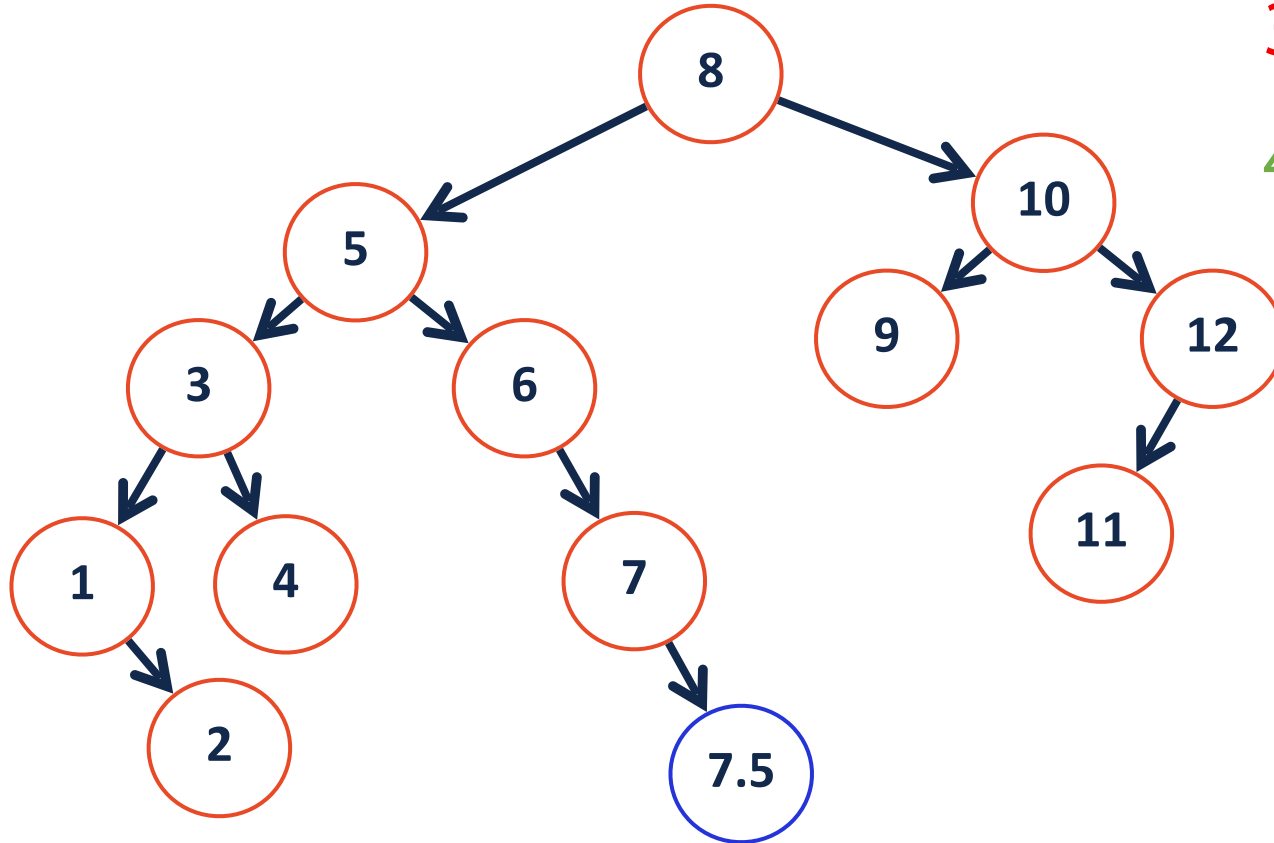
2) Update root to point to mid

3) tmp->right = root->left

4) root->left = tmp

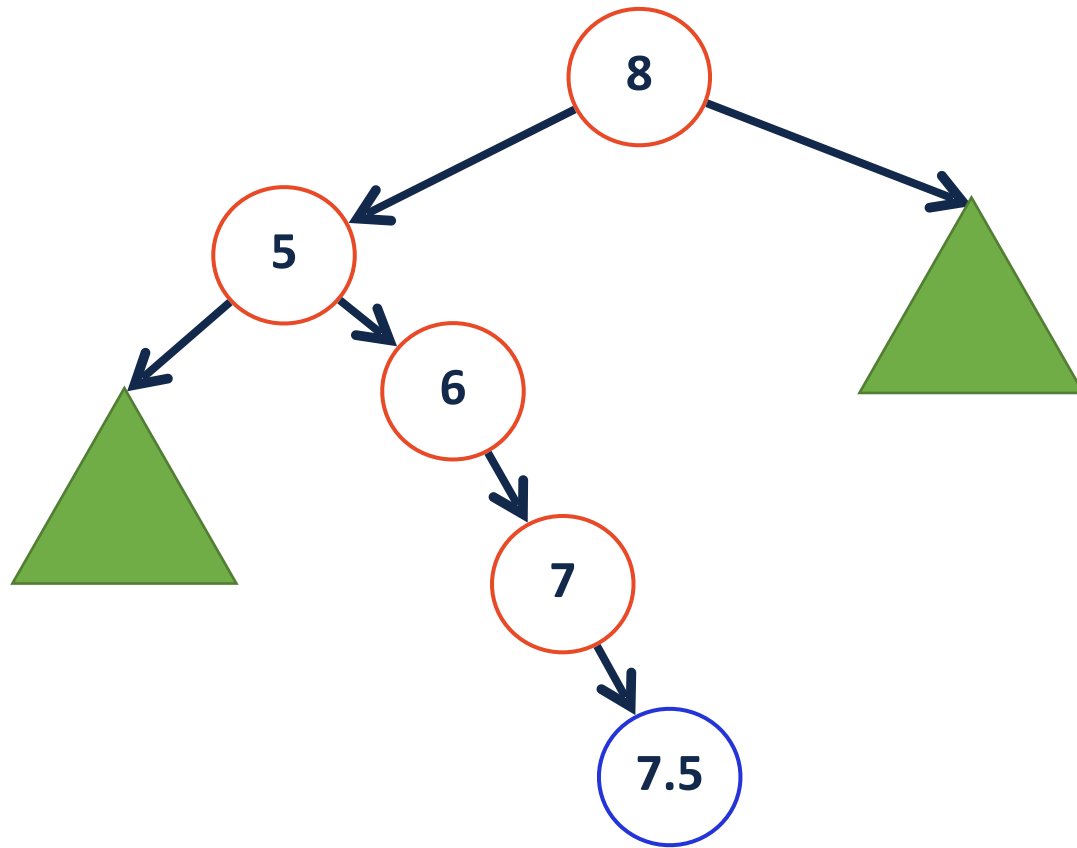
Left Rotation

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



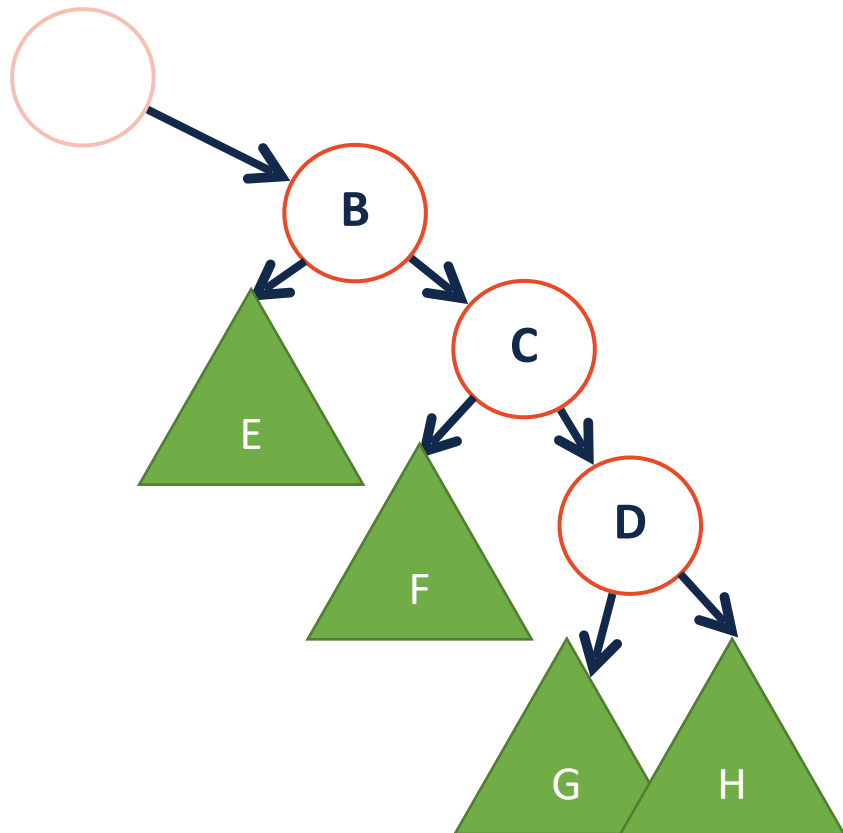
Left Rotation

All rotations are local (subtrees are not impacted)

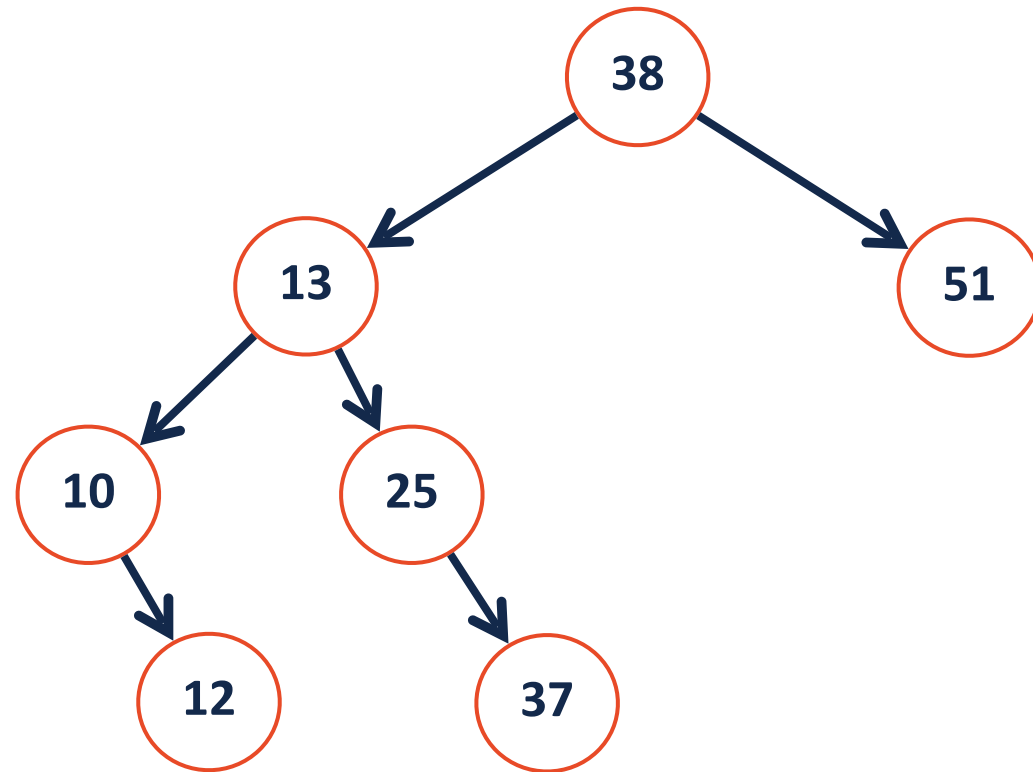


Left Rotation

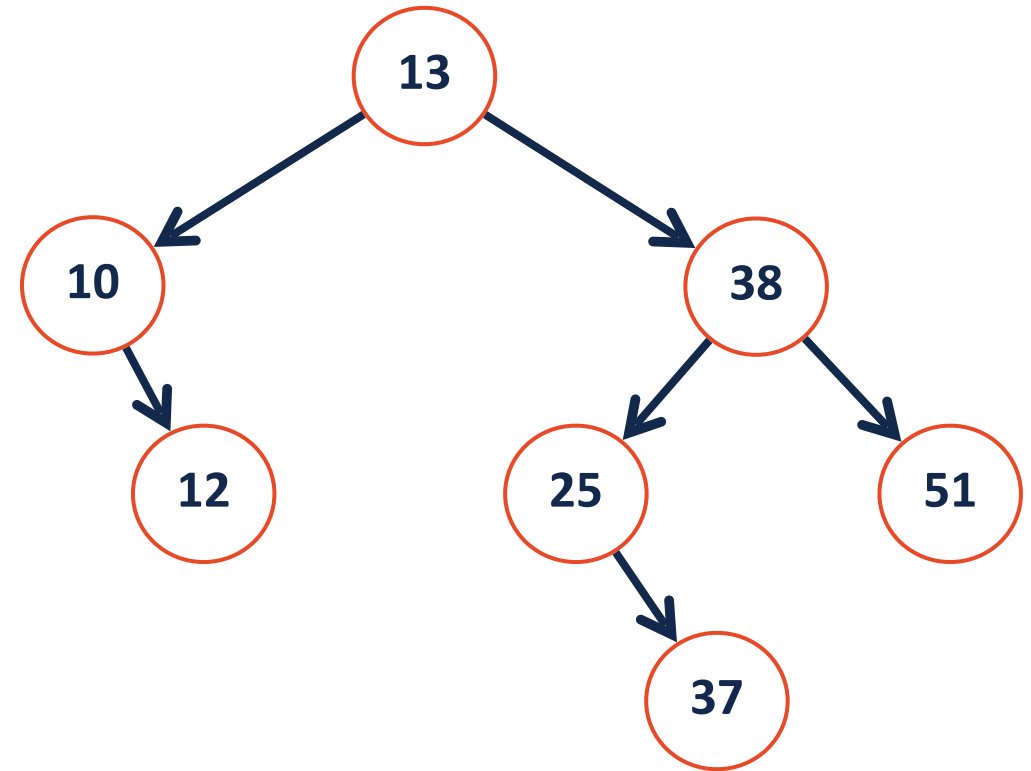
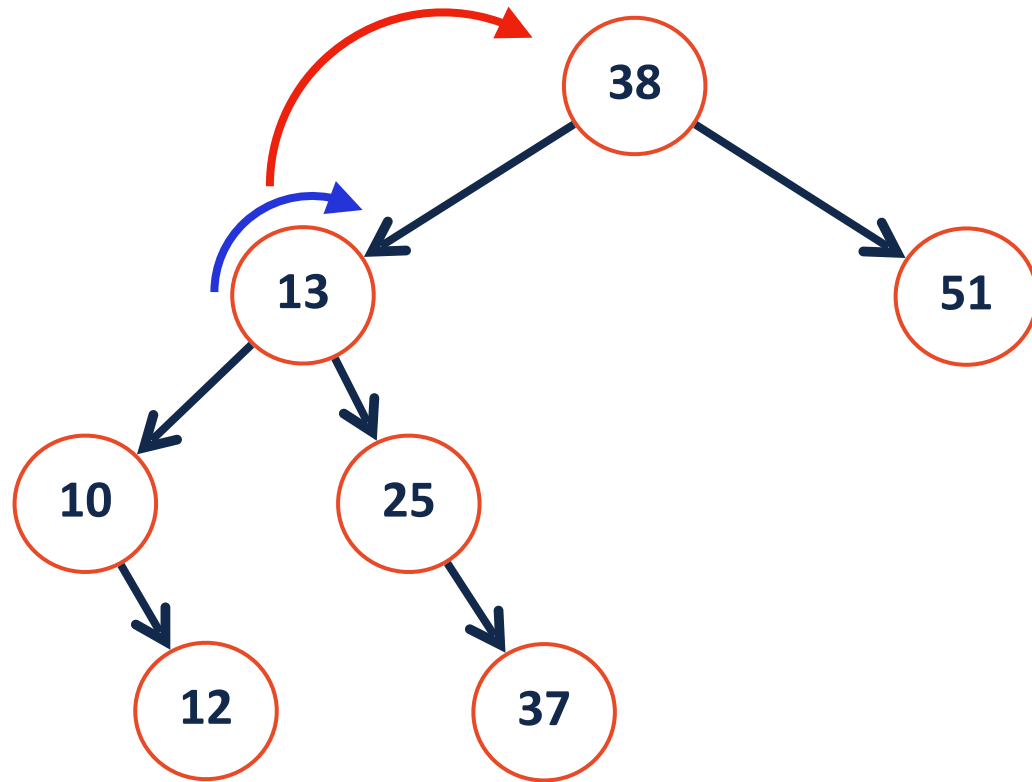
All rotations preserve BST property



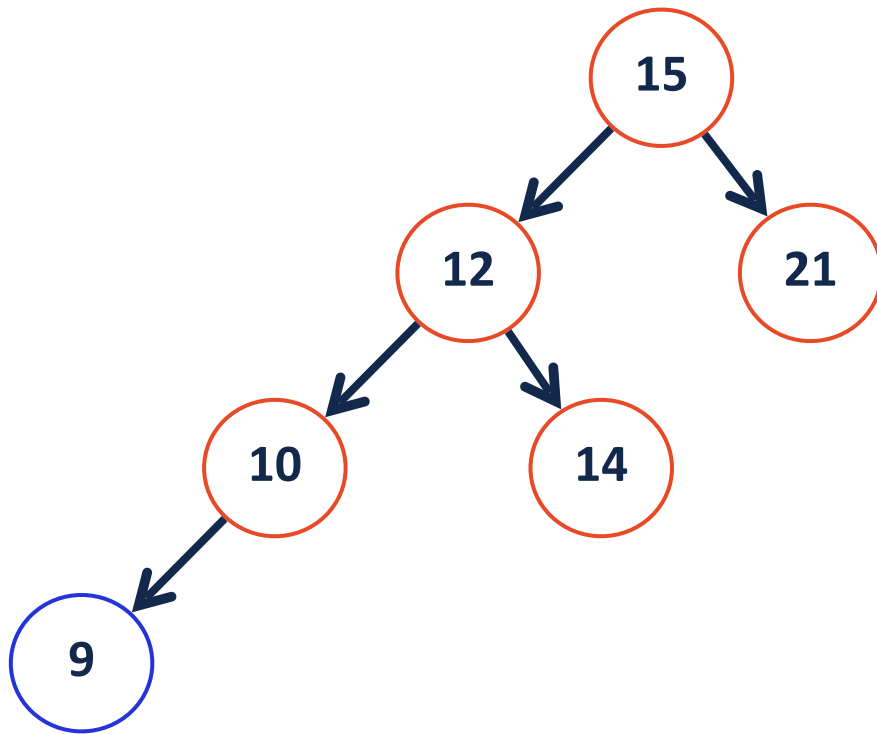
Right Rotation



Right Rotation

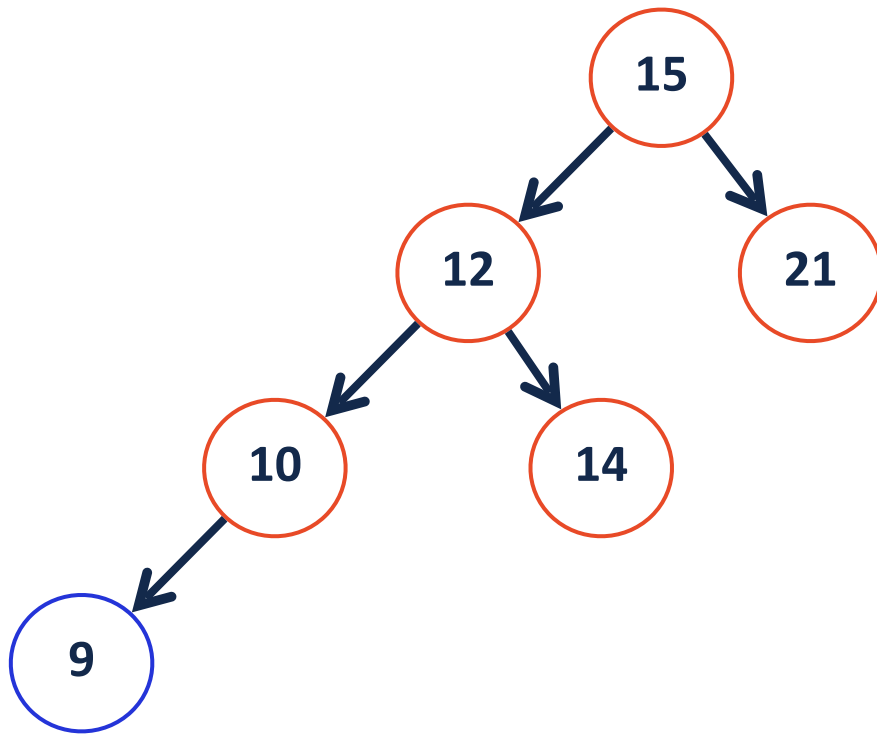


Right Rotation

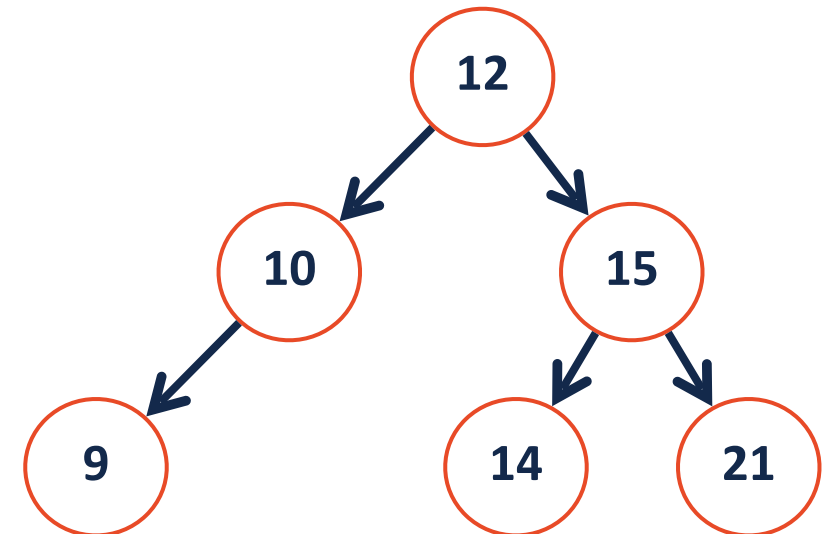


- 1) Create a tmp pointer to root
- 2) Update root to point to mid
- 3) $\text{tmp} \rightarrow \text{left} = \text{root} \rightarrow \text{right}$
- 4) $\text{root} \rightarrow \text{right} = \text{tmp}$

Right Rotation



- 1) Create a tmp pointer to root
- 2) Update root to point to mid
- 3) $\text{tmp} \rightarrow \text{left} = \text{root} \rightarrow \text{right}$
- 4) $\text{root} \rightarrow \text{right} = \text{tmp}$



Coding AVL Rotations

Two ways of visualizing:

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

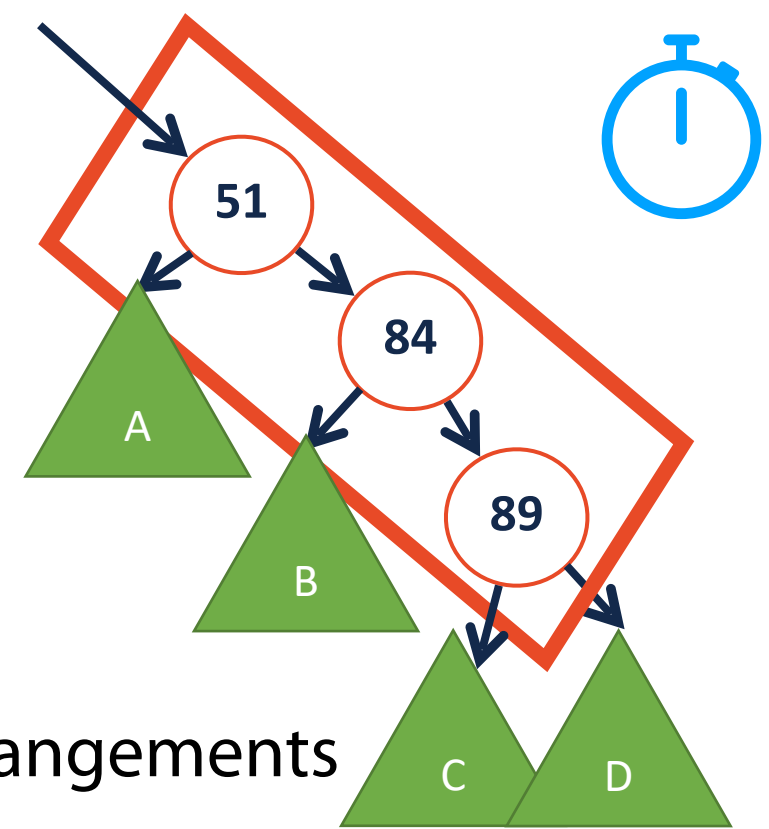
2) Recognize that there's a concrete order for rearrangements

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

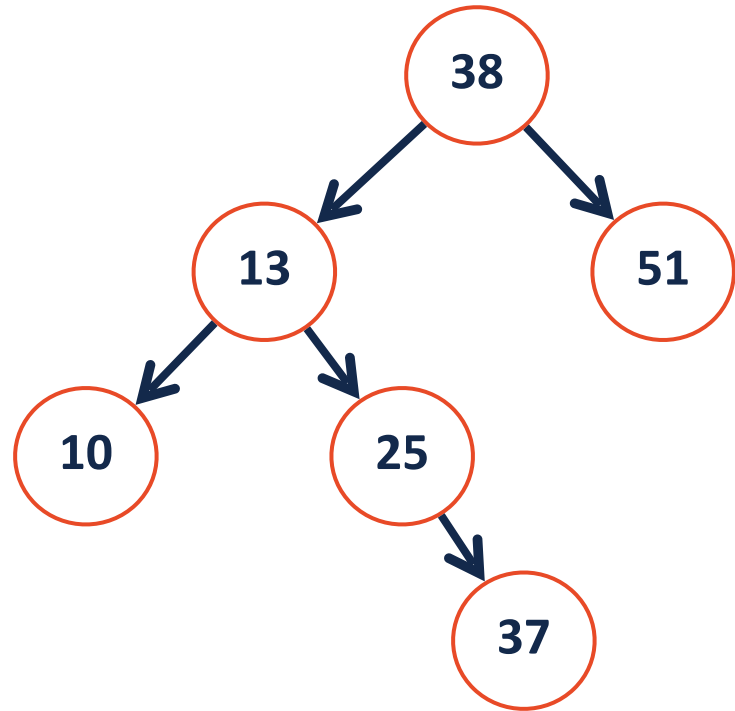
Replace current (root) node with its right child.

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

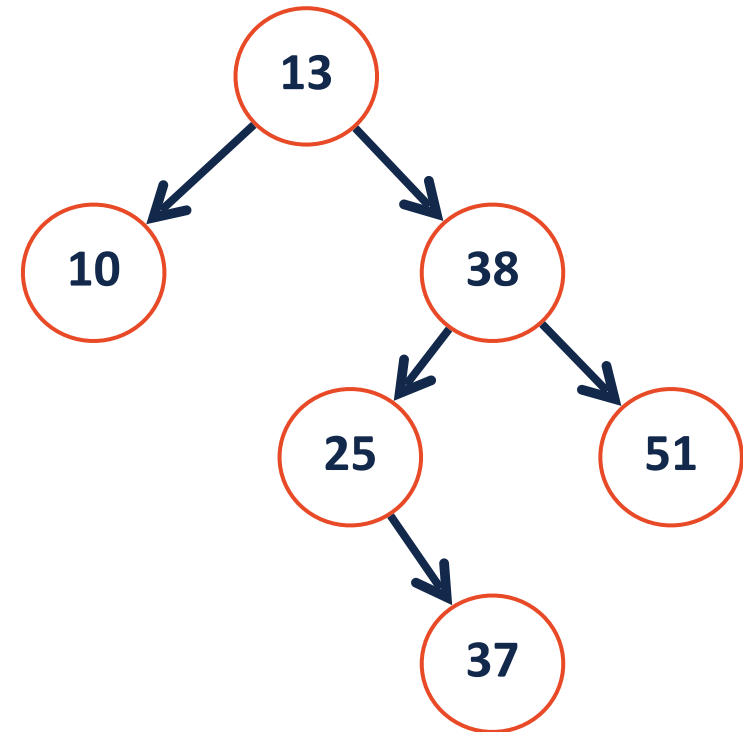
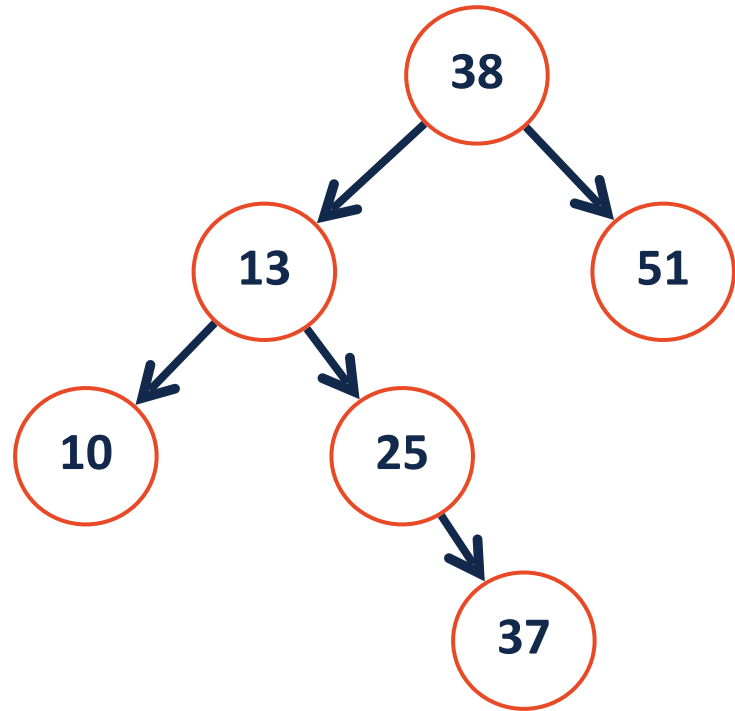
Make the current node the right child's left child



AVL Rotation Practice

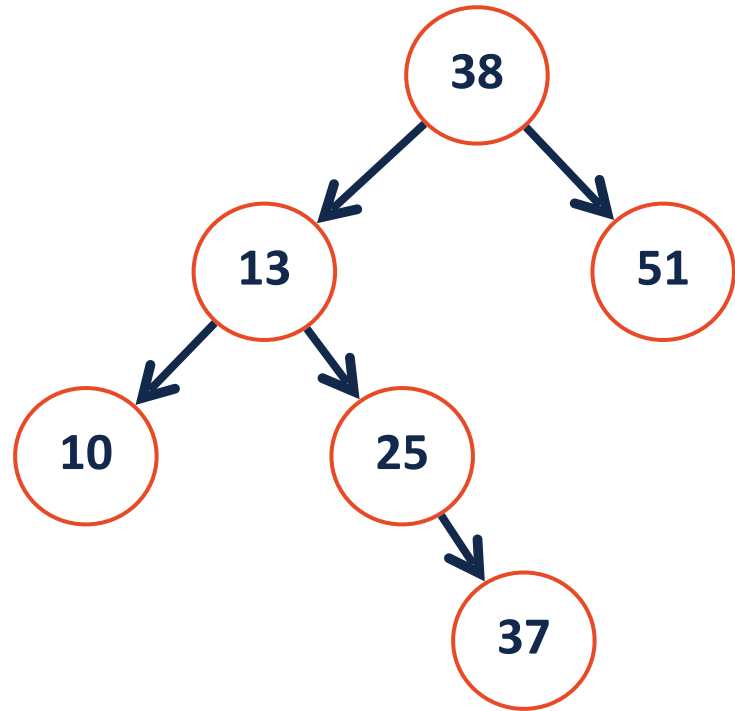


AVL Rotation Practice

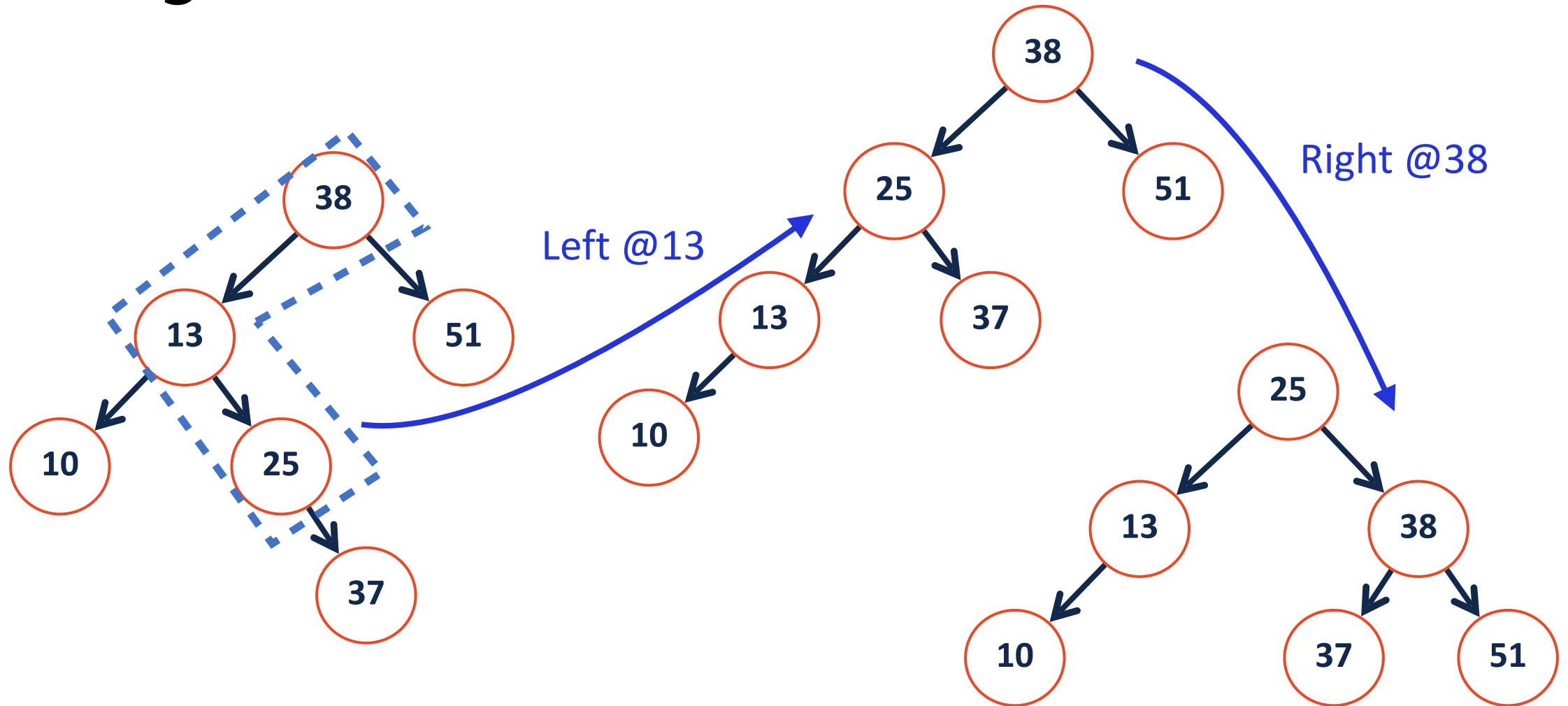


Somethings not quite right...

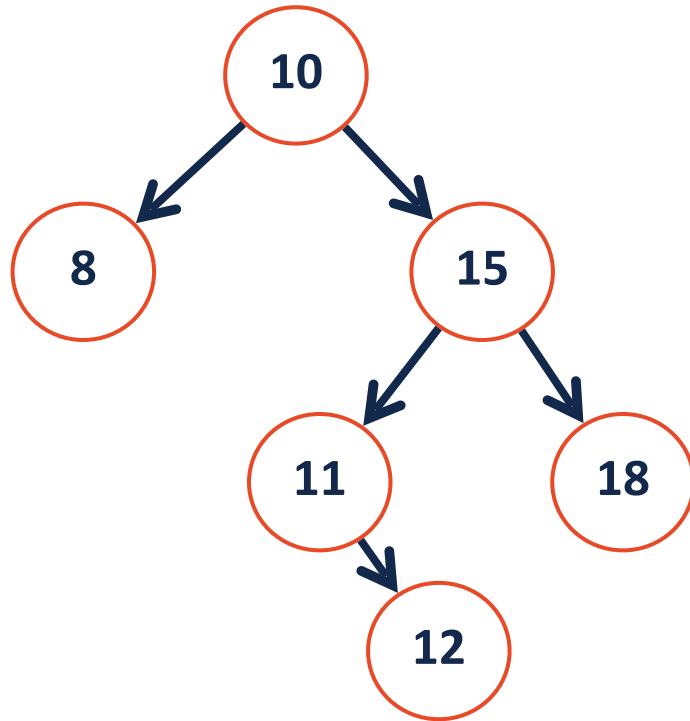
LeftRight Rotation



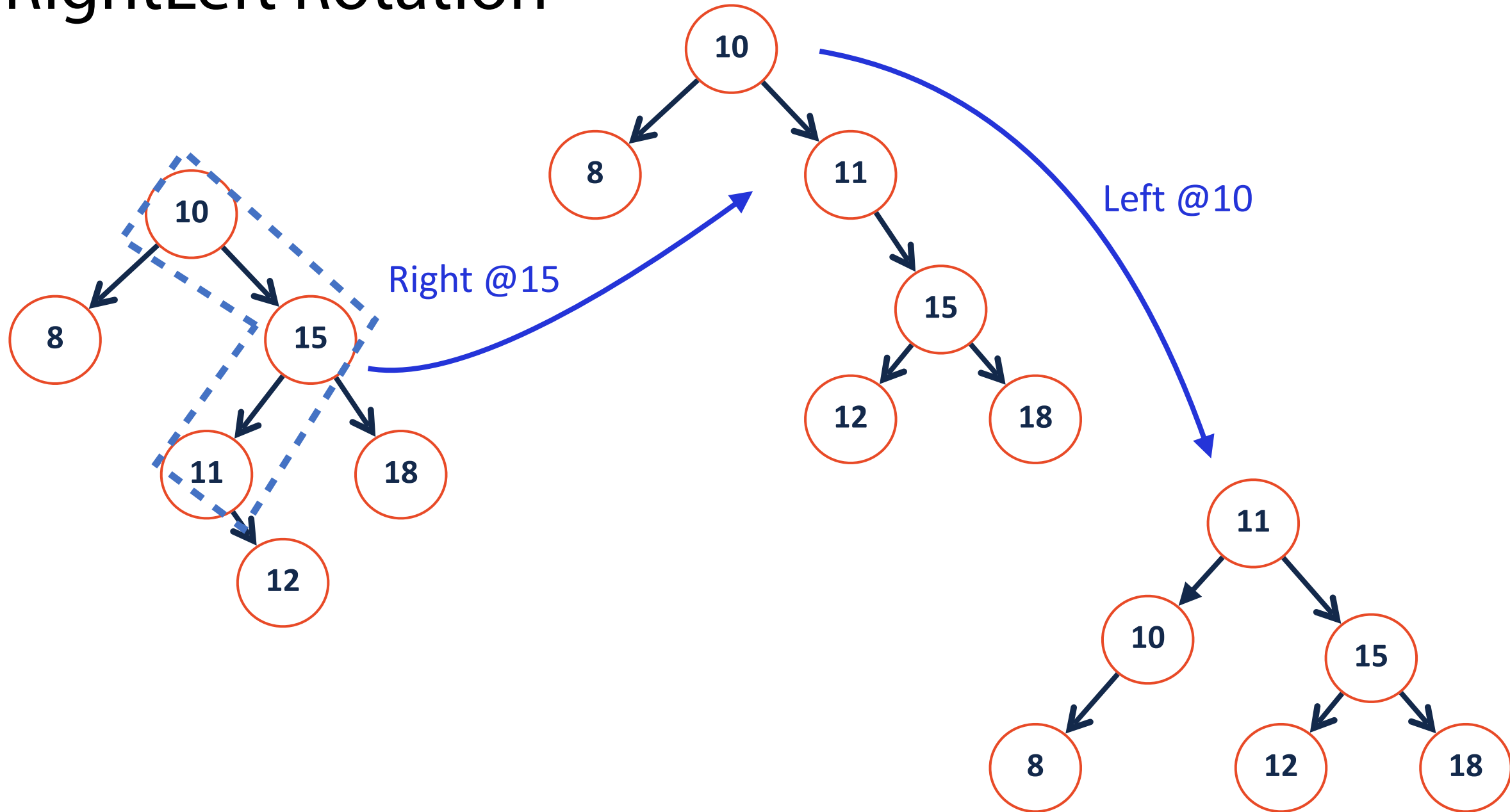
LeftRight Rotation



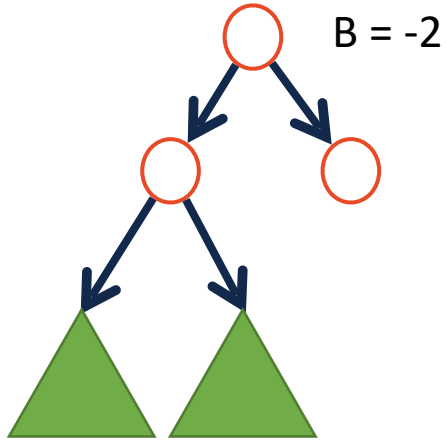
RightLeft Rotation



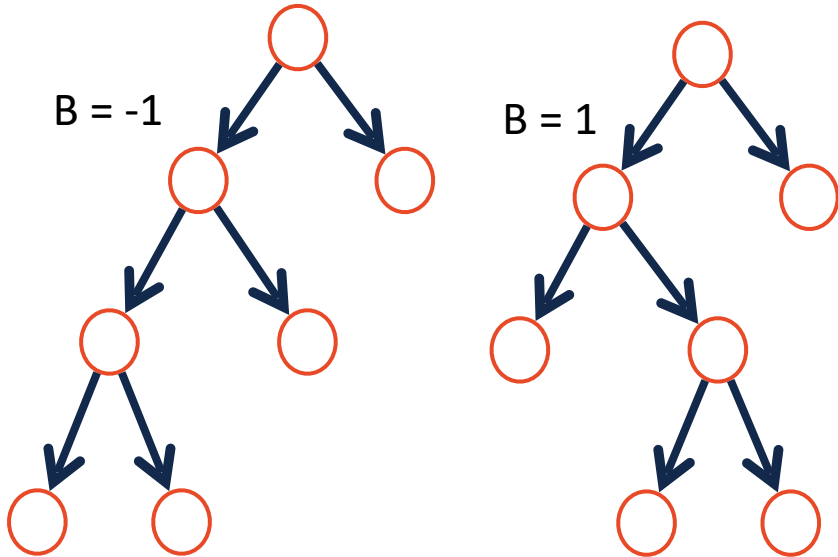
RightLeft Rotation



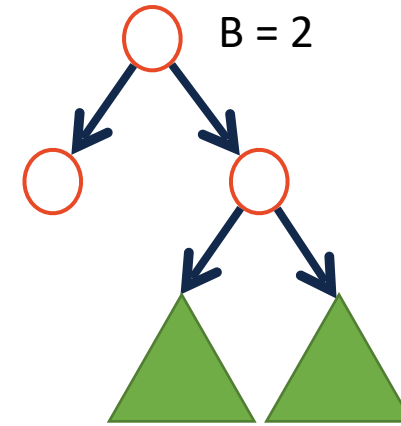
Left Imbalance



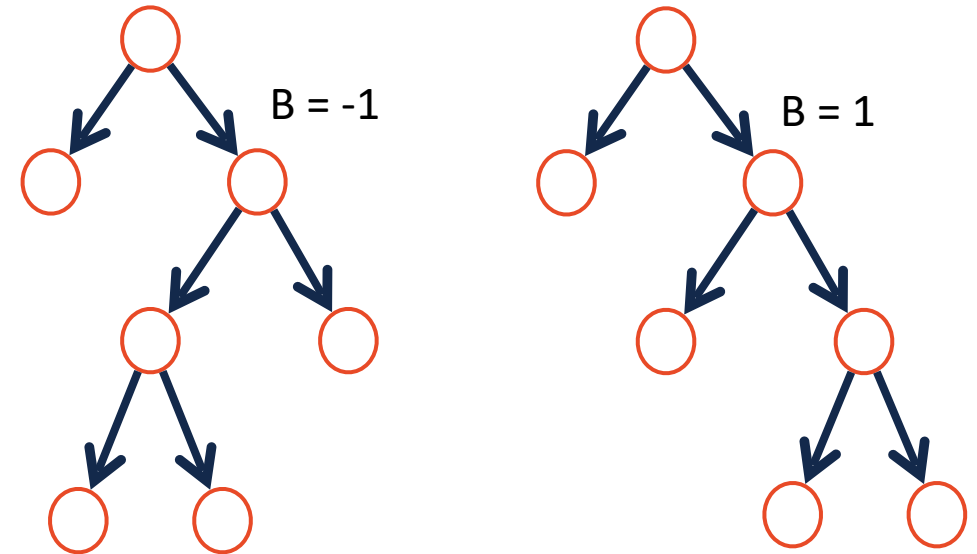
Check left subtree:



Right Imbalance



Check right subtree:





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

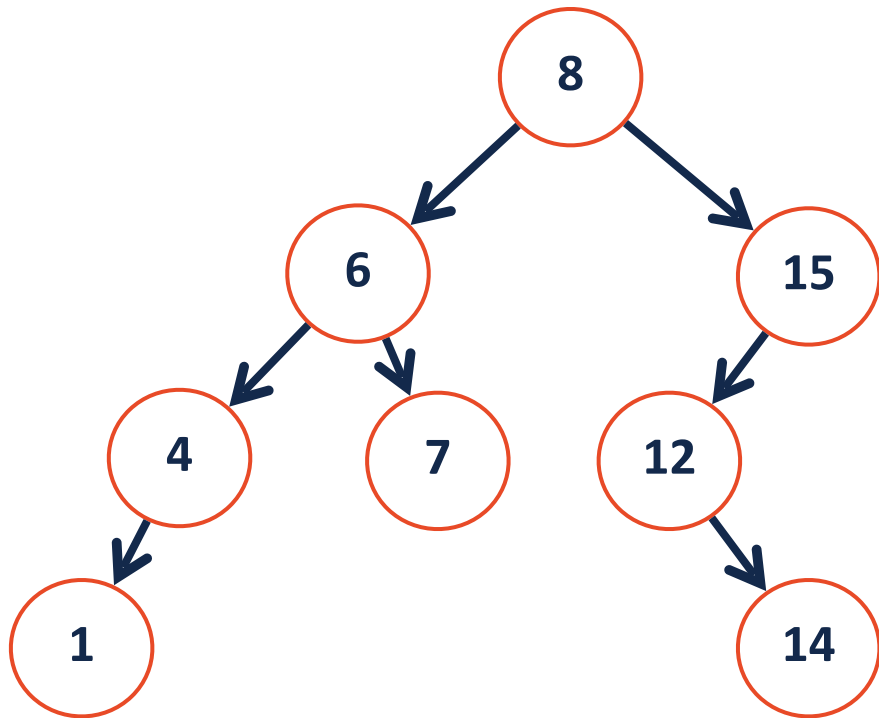
Goal:

AVL Rotation Practice



Where is my imbalance? What rotation?

Join Code: 225



AVL vs BST ADT



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

```
1 struct TreeNode {  
2     T key;  
3     unsigned height;  
4     TreeNode *left;  
5     TreeNode *right;  
6 };
```

How does the constraint on balance affect the core functions?

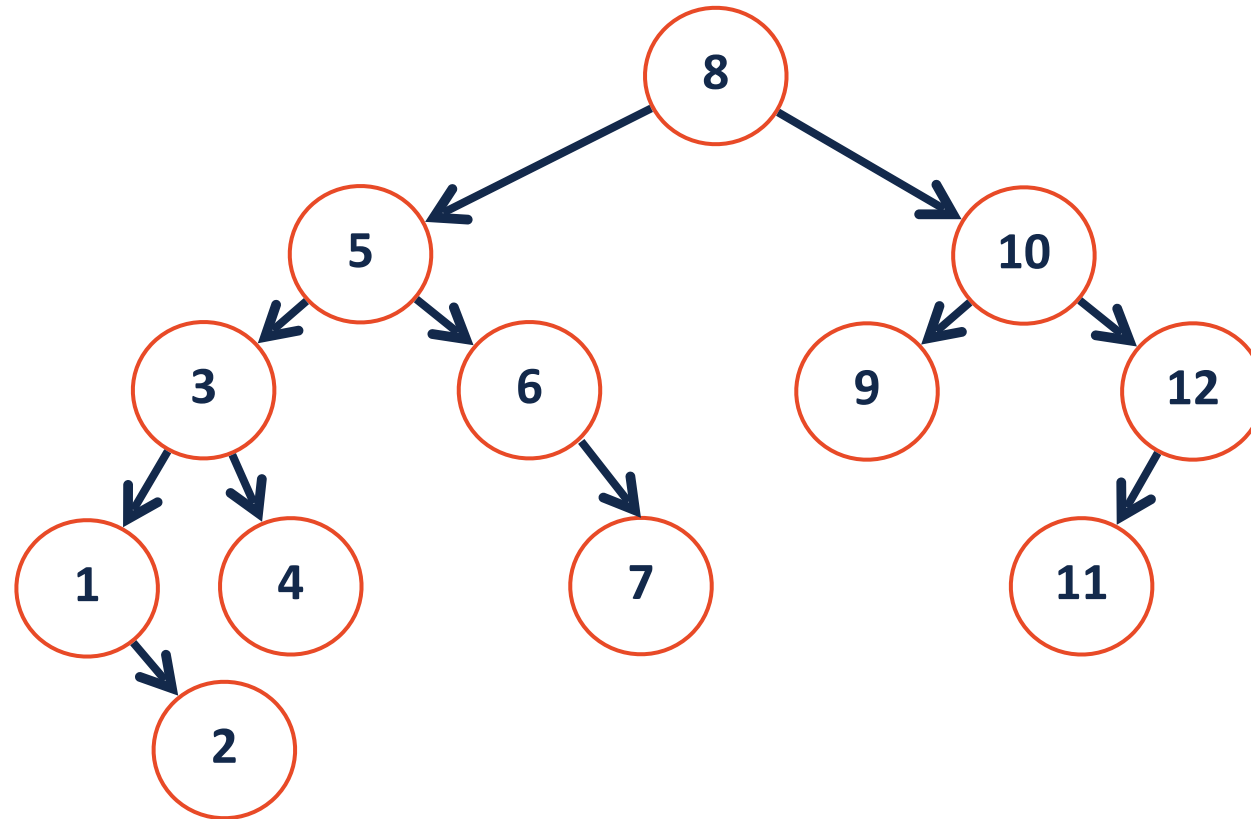
Find

Insert

Remove

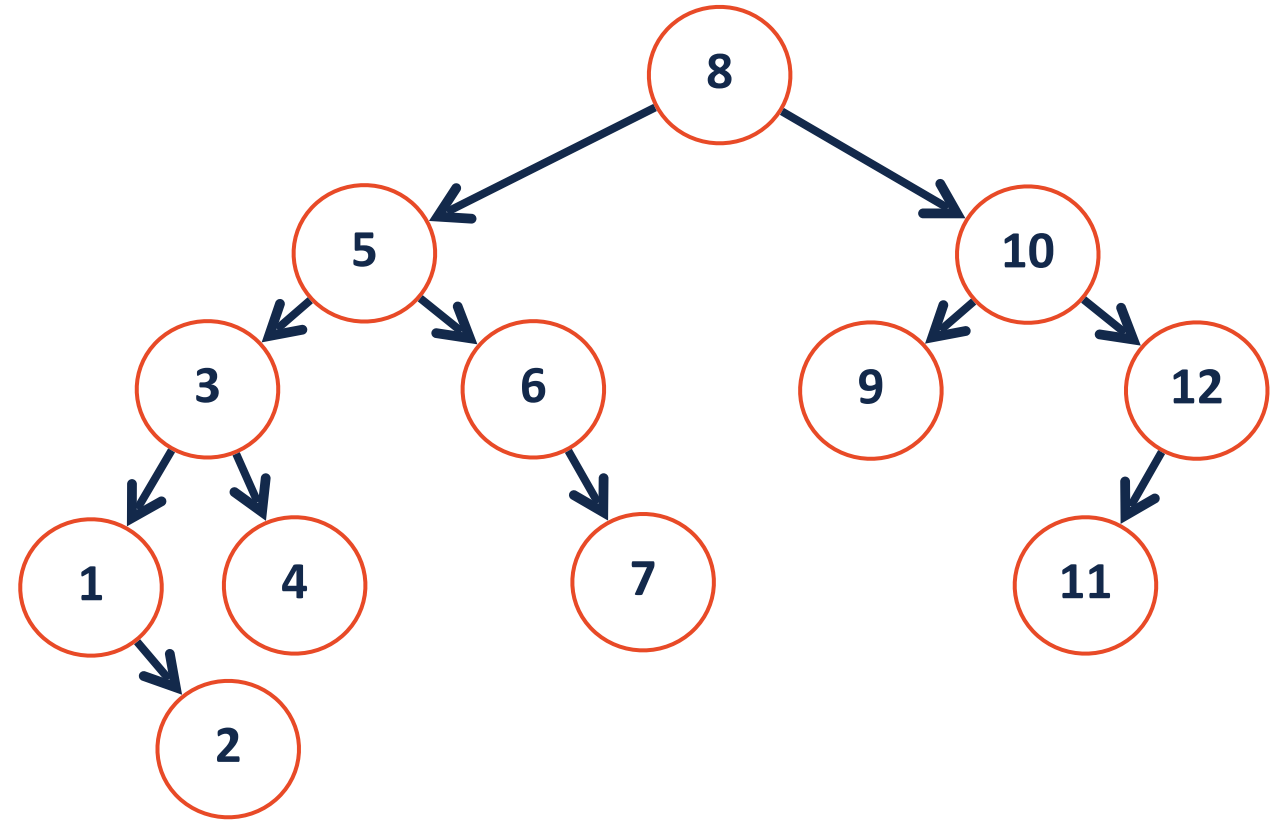
AVL Find

`_find(7)`



AVL Insertion

`_insert(6.5)`



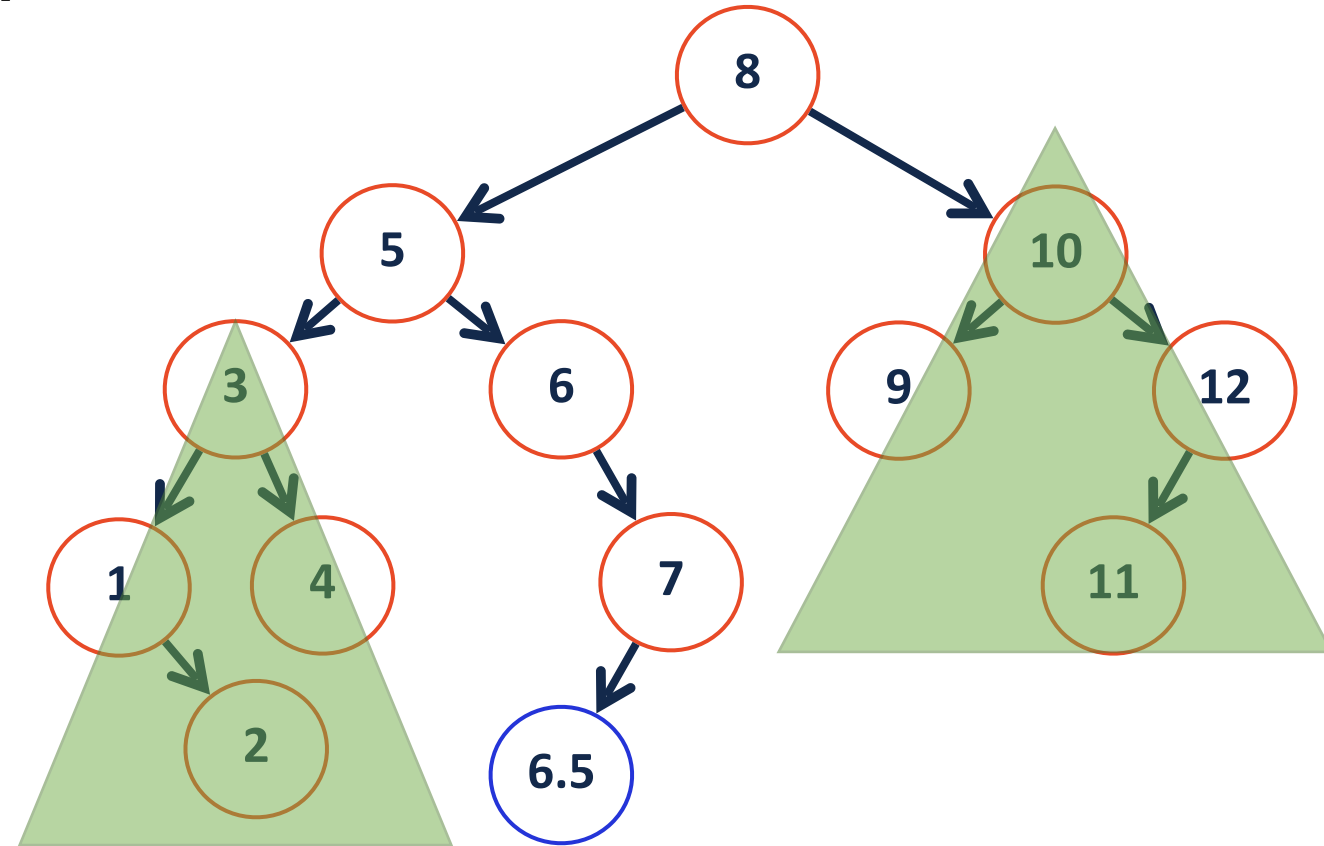
```
1 struct TreeNode {  
2     T key;  
3     unsigned height;  
4     TreeNode *left;  
5     TreeNode *right;  
6 };
```

AVL Insertion

`_insert(6.5)`

Insert (recursive pseudocode):

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



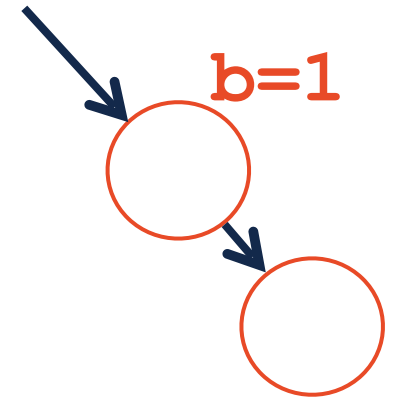
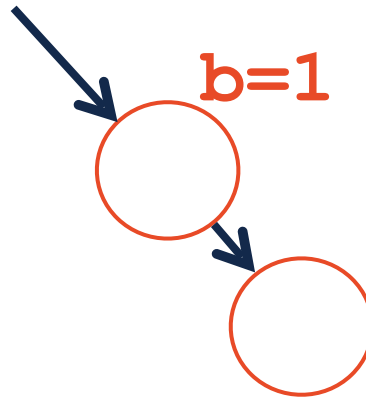
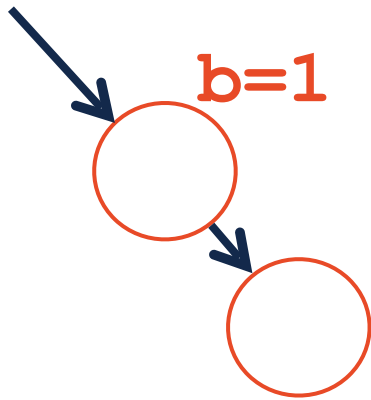
```
1 struct TreeNode {  
2     T key;  
3     unsigned height;  
4     TreeNode *left;  
5     TreeNode *right;  
6 };
```

```
151 template <typename K, typename V>
152 void AVL<K, D>::_insert(const K & key, const V & data, TreeNode
*& cur) {
153     if (cur == NULL)           { cur = new TreeNode(key, data); }
157     else if (key < cur->key) { _insert( key, data, cur->left ); }
160     else if (key > cur->key) { _insert( key, data, cur->right ); }
166     _ensureBalance(cur);
167 }
```

```
119 template <typename K, typename V>
120 void AVL<K, D>::_ensureBalance(TreeNode *& cur) {
121     // Calculate the balance factor:
122     int balance = height(cur->right) - height(cur->left);
123
124     // Check if the node is current not in balance:
125     if ( balance == -2 ) {
126         int l_balance =
127             height(cur->left->right) - height(cur->left->left);
128         if ( l_balance == -1 ) { _____; }
129         else { _____; }
130     } else if ( balance == 2 ) {
131         int r_balance =
132             height(cur->right->right) - height(cur->right->left);
133         if( r_balance == 1 ) { _____; }
134         else { _____; }
135     }
136     _updateHeight(cur);
137 };
```

AVL Insertion

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





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:

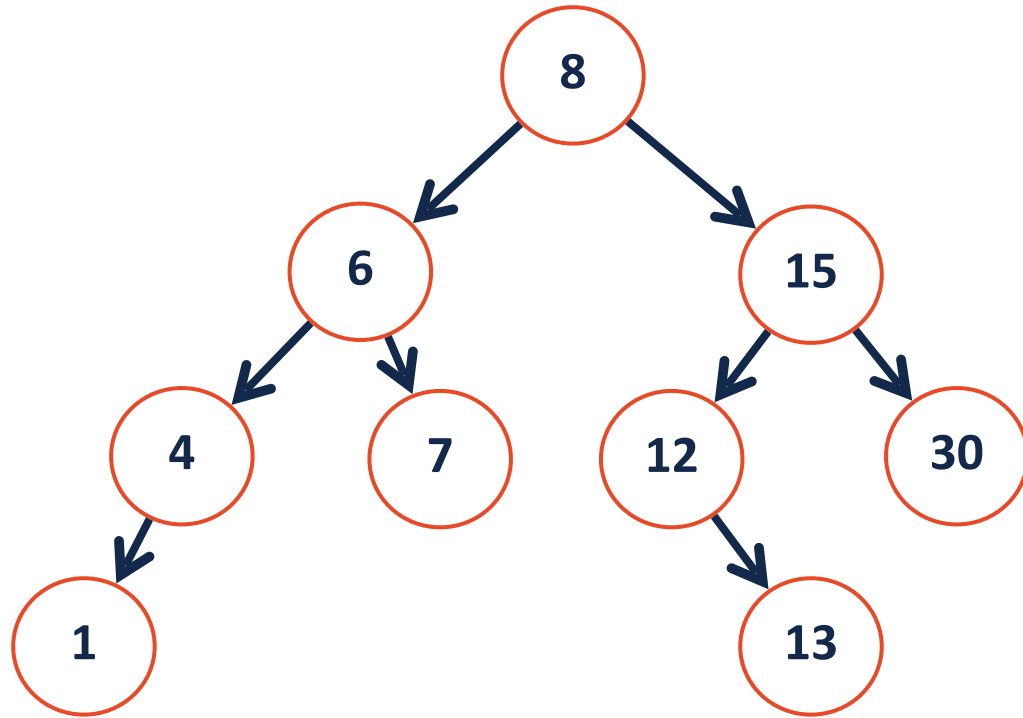
A single* rotation restores balance and corrects height!

What is the Big O of performing our rotation?

What is the Big O of insert?

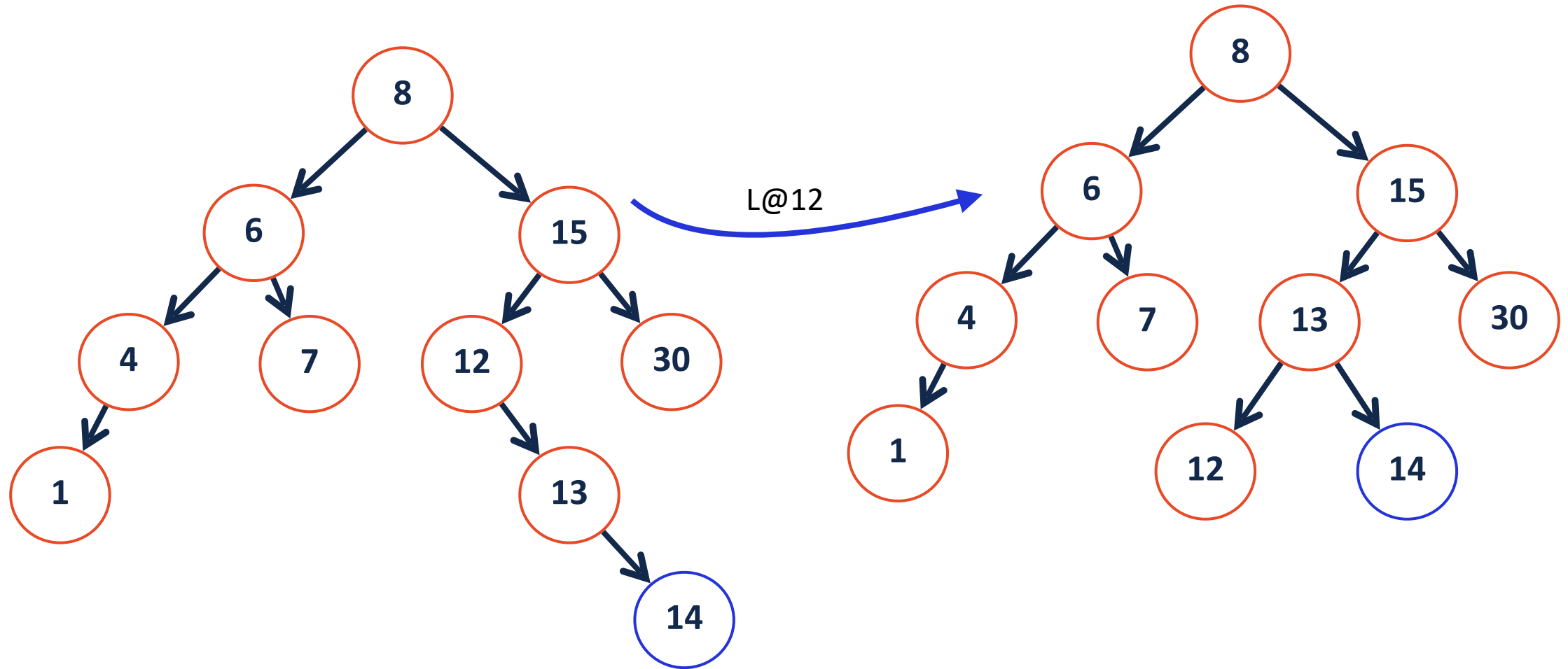
AVL Insertion Practice

`_insert(14)`



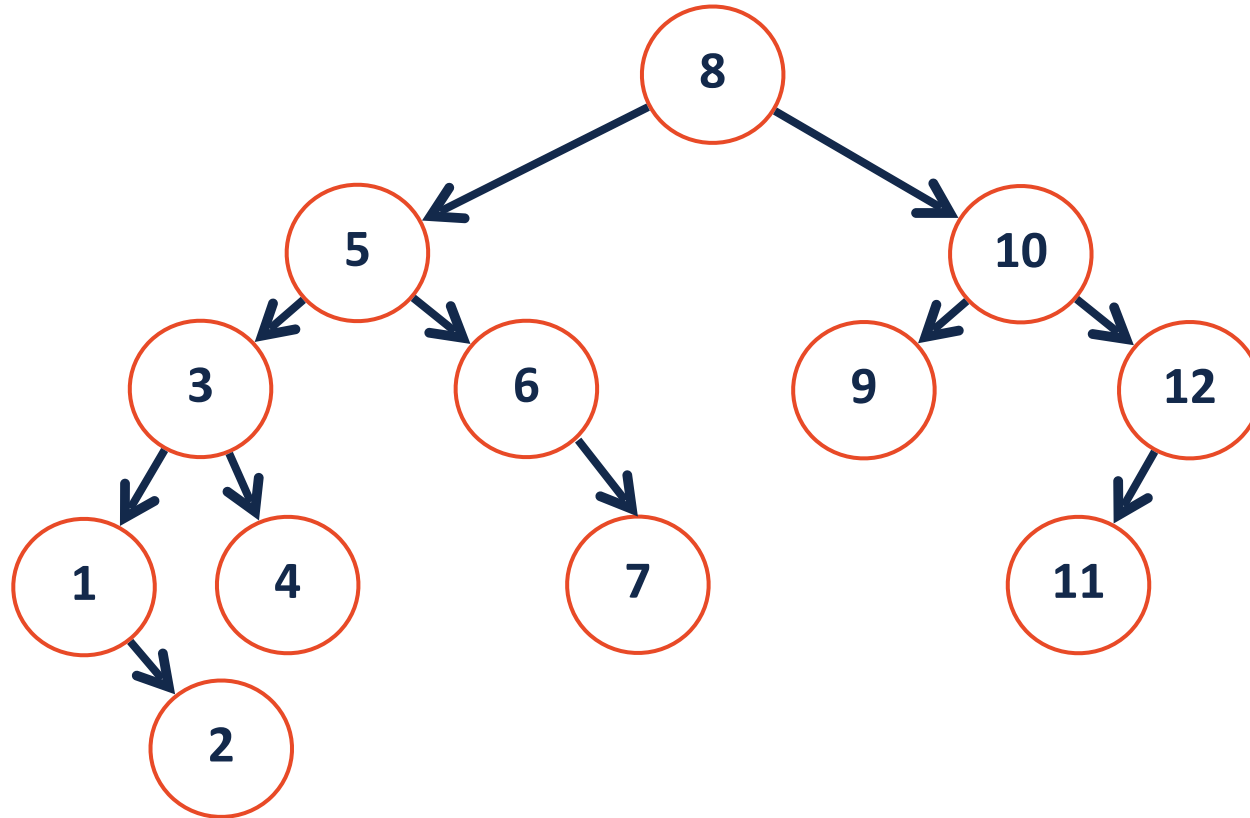
AVL Insertion Practice

`_insert(14)`



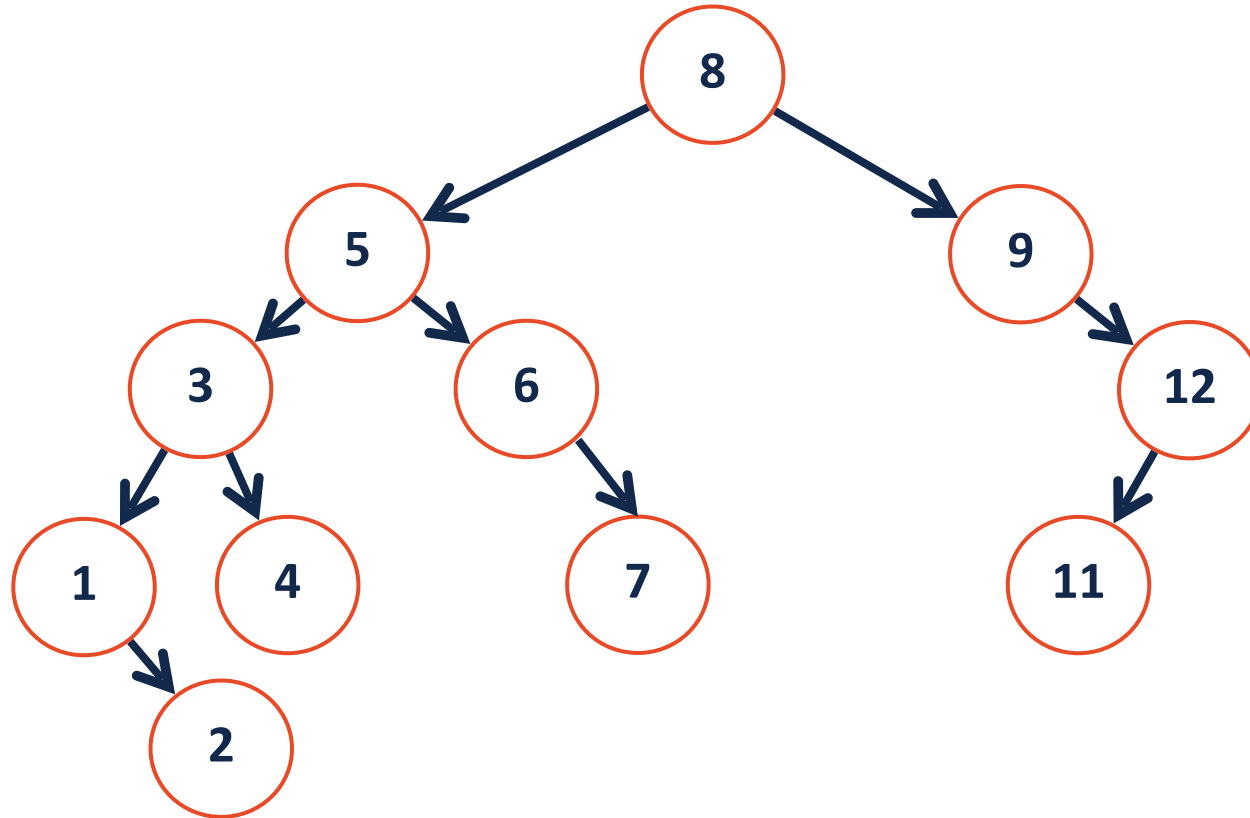
AVL Remove

`_remove(10)`



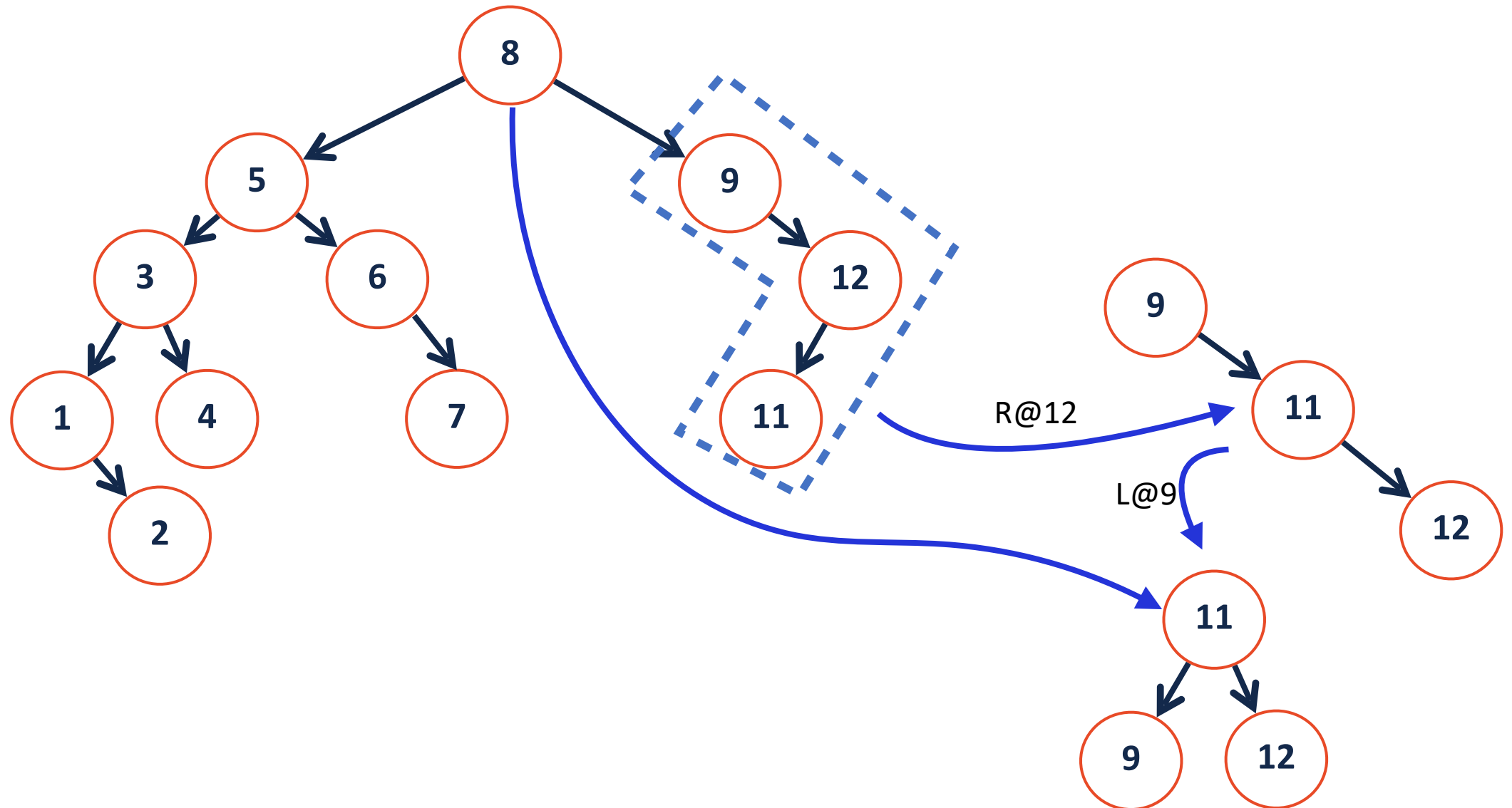
AVL Remove

`_remove(10)`



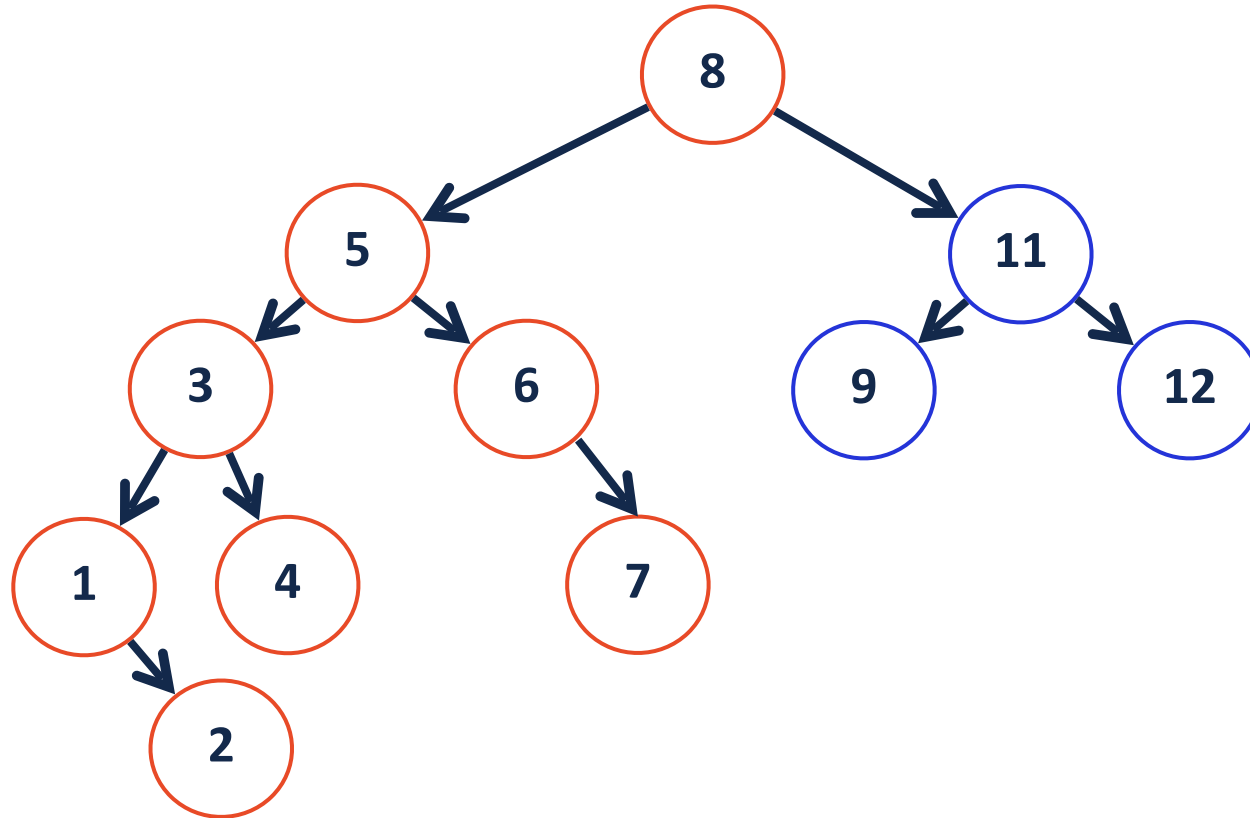
AVL Remove

`_remove(10)`



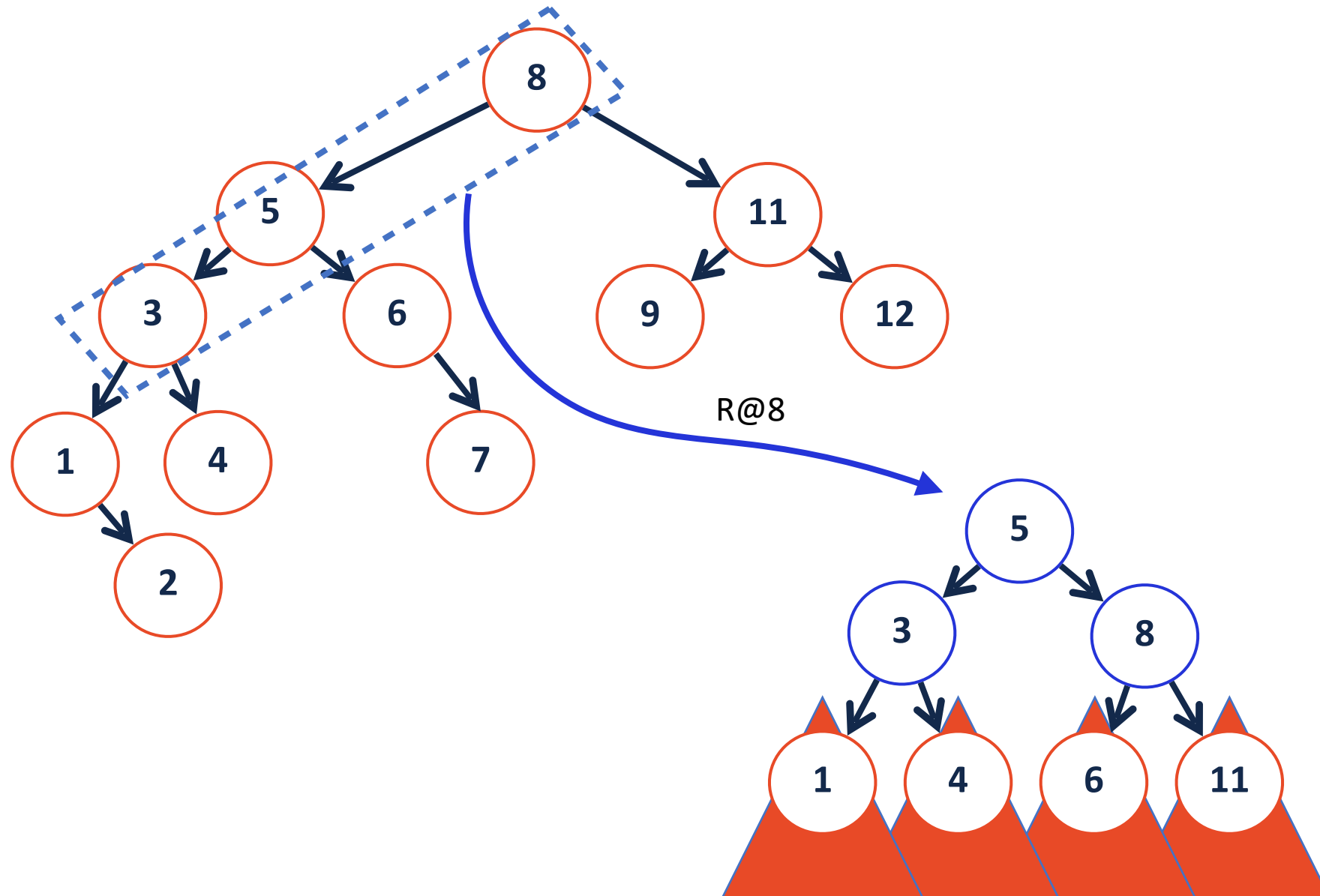
AVL Remove

`_remove(10)`



AVL Remove

`_remove(10)`



AVL Remove

`_remove(10)` 

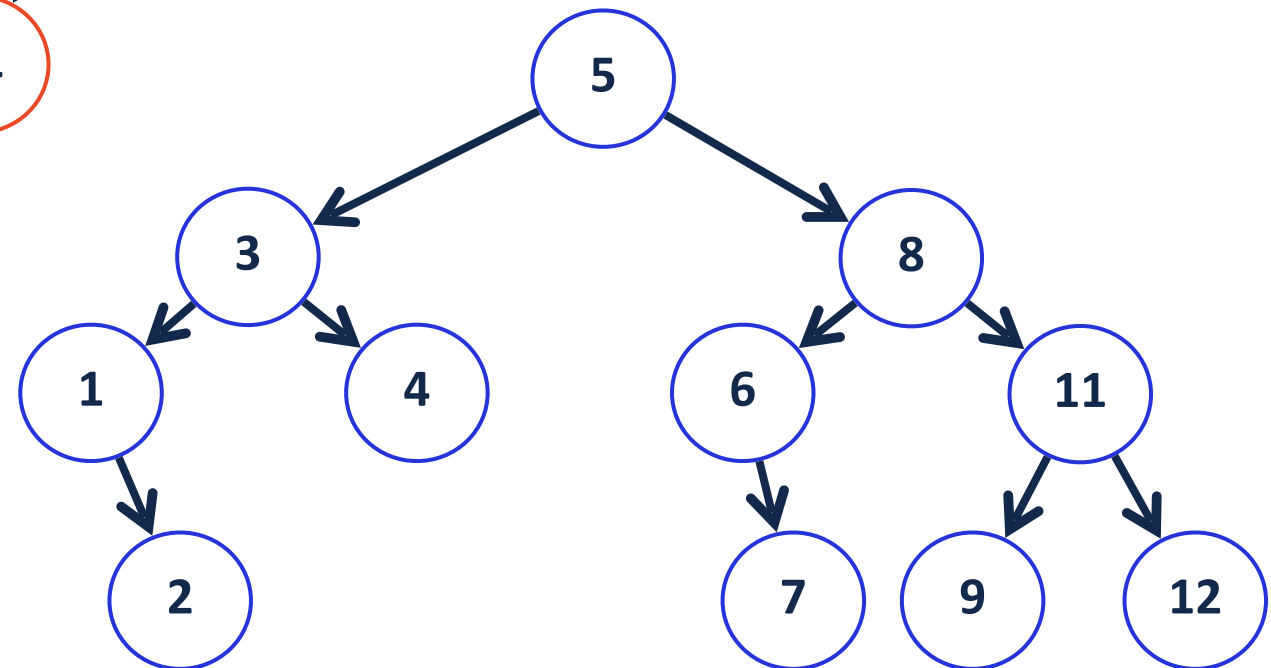
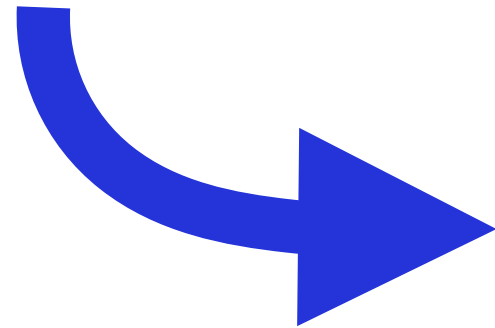
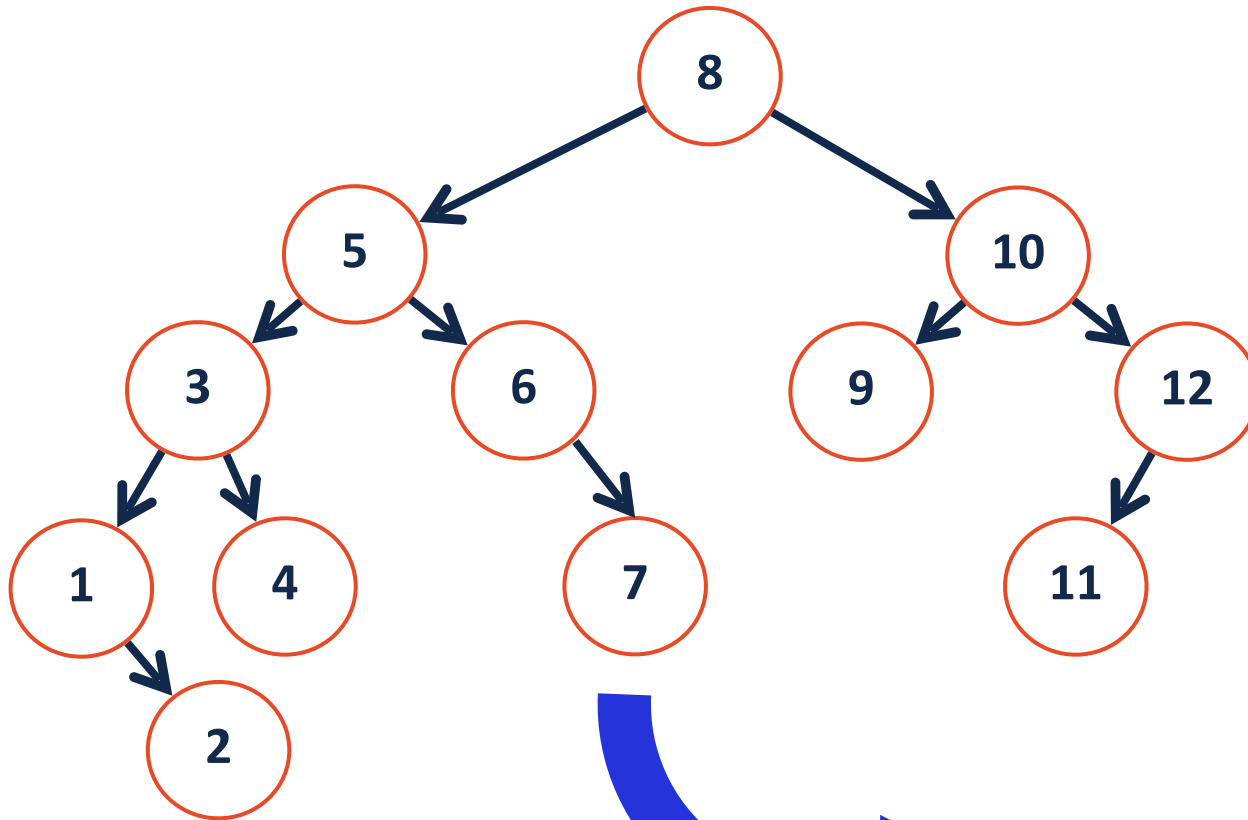
Remove (pseudo code):

1: Remove at proper place

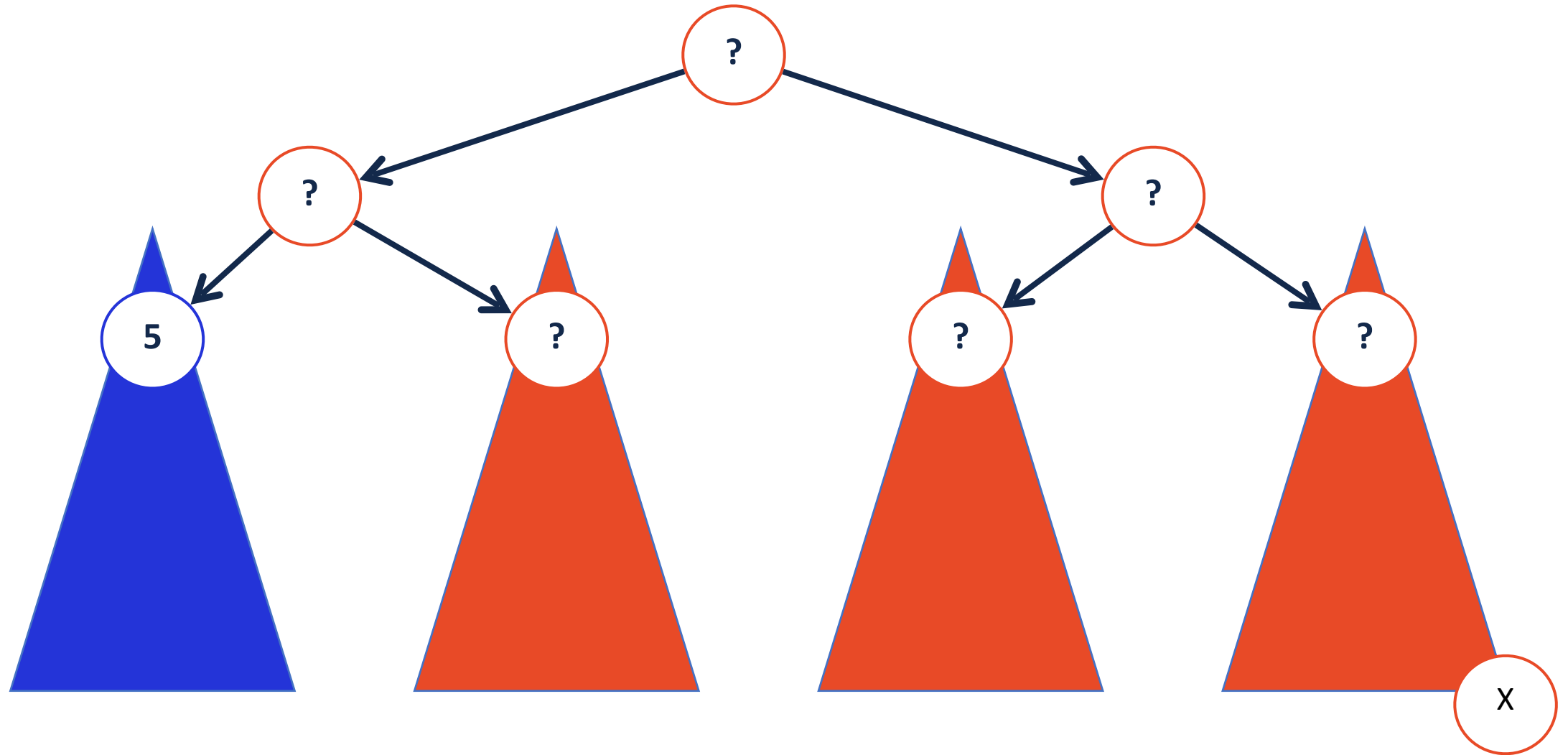
2: Check for imbalance

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: _____.