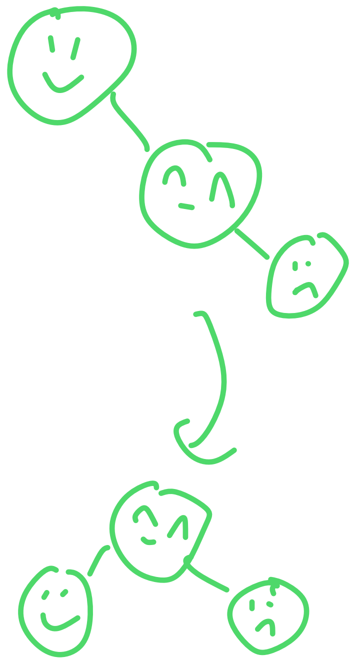
# Data Structures
# AVL Trees

CS 225
Brad Solomon & G Carl Evans

September 25, 2023



Department of Computer Science

# Learning Objectives

Review why we need balanced trees

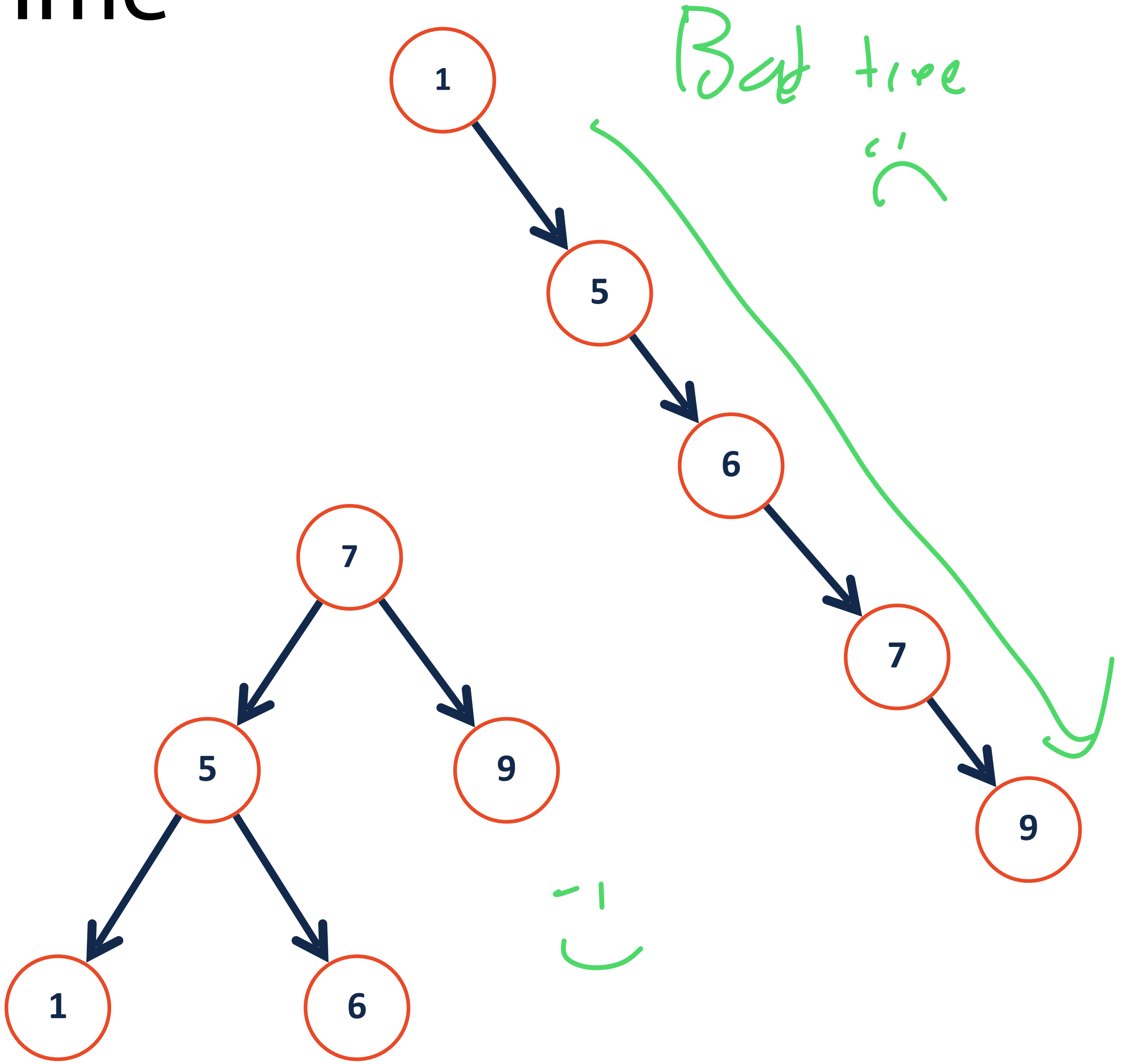
Review what an AVL rotation does

Explore the four possible rotations for an AVL tree

How they modify the ADT

# BST Analysis – Running Time

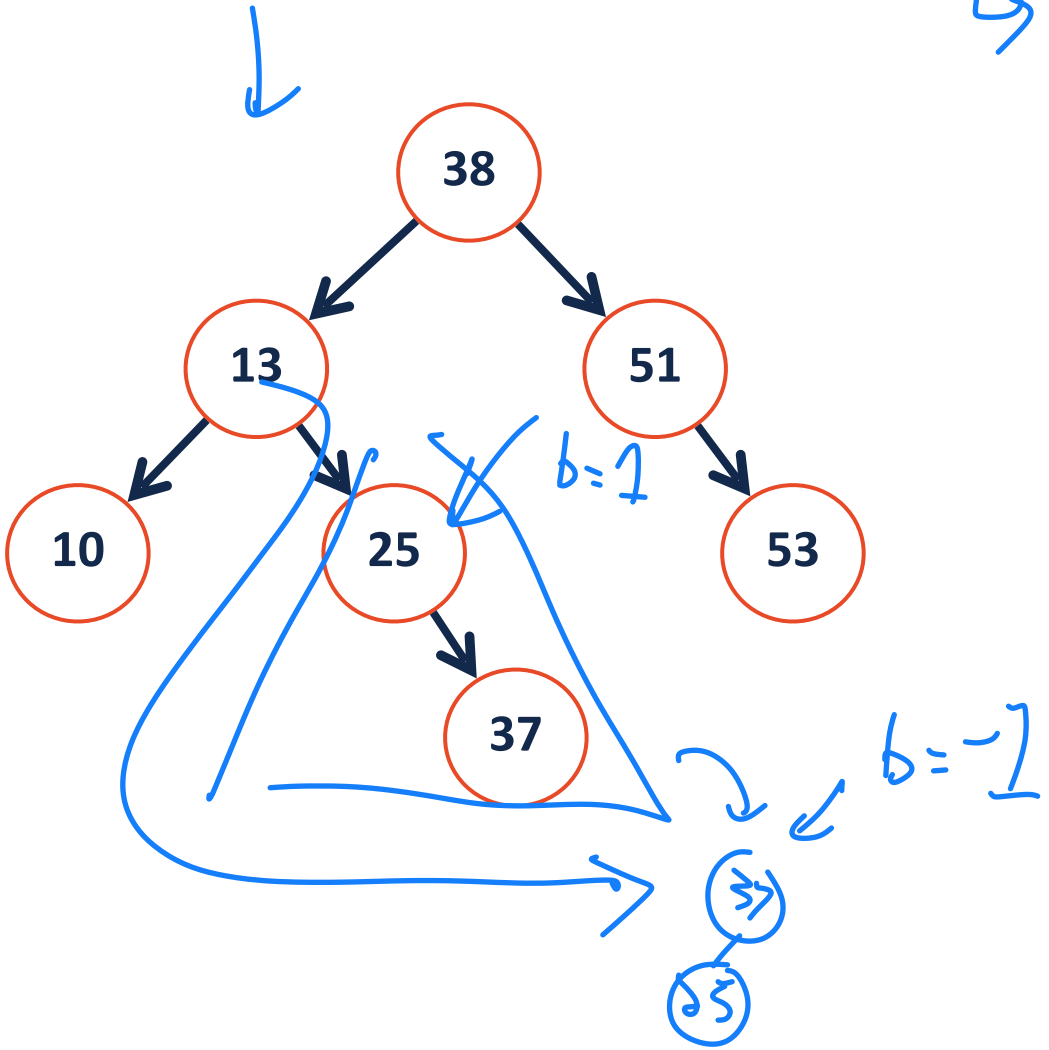
| | BST Worst Case |
|---|---|
| **find** | O(h) |
| **insert** | O(h) |
| **delete** | O(h) |
| **traverse** | O(n) |



# AVL-Tree: A self-balancing binary search tree

Every node in an AVL tree has a balance of: $-1 \leq b \leq 1$

$\hookrightarrow \text{Height}(T_R) - \text{Height}(T_L)$



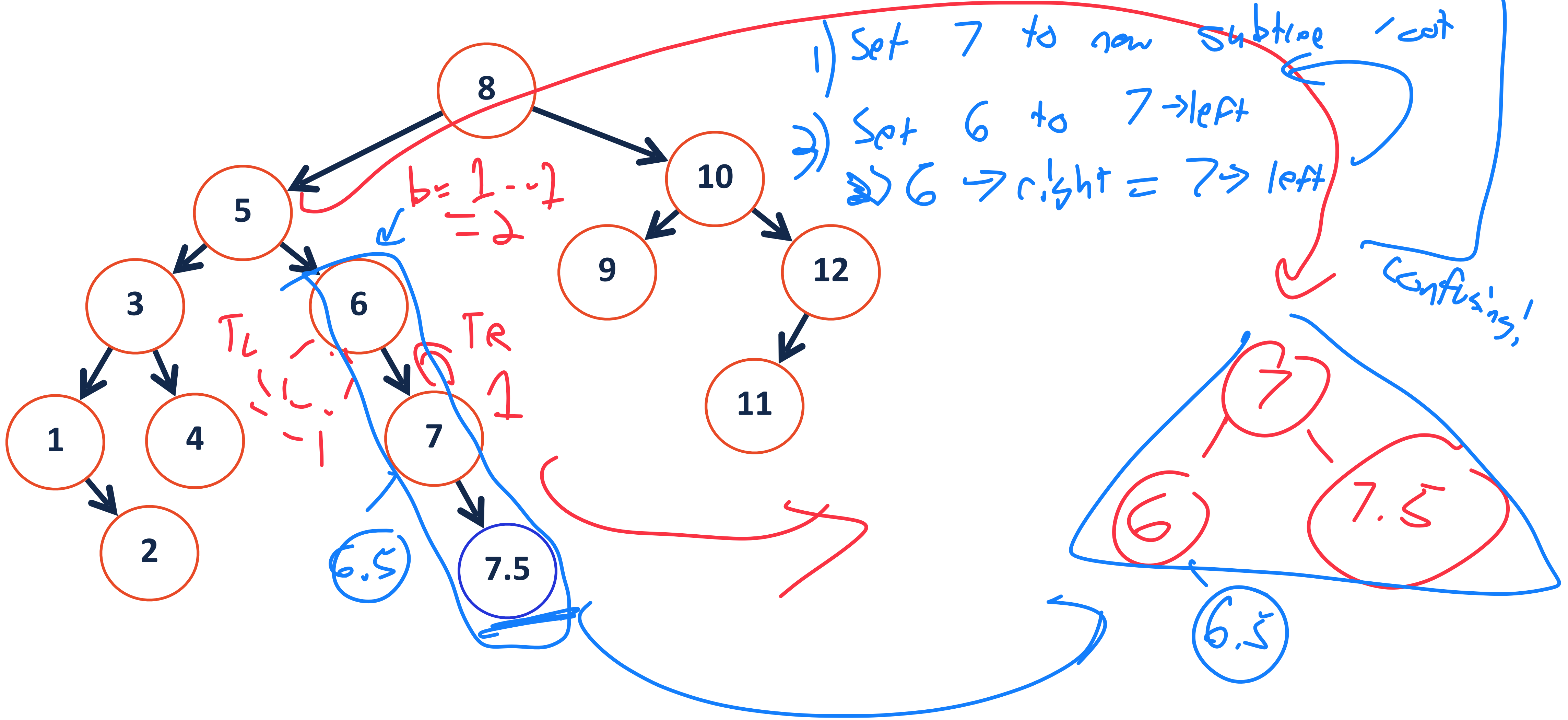
# Left Rotation

balance is (+) I am right heavy

1) Set 7 to new subtree root

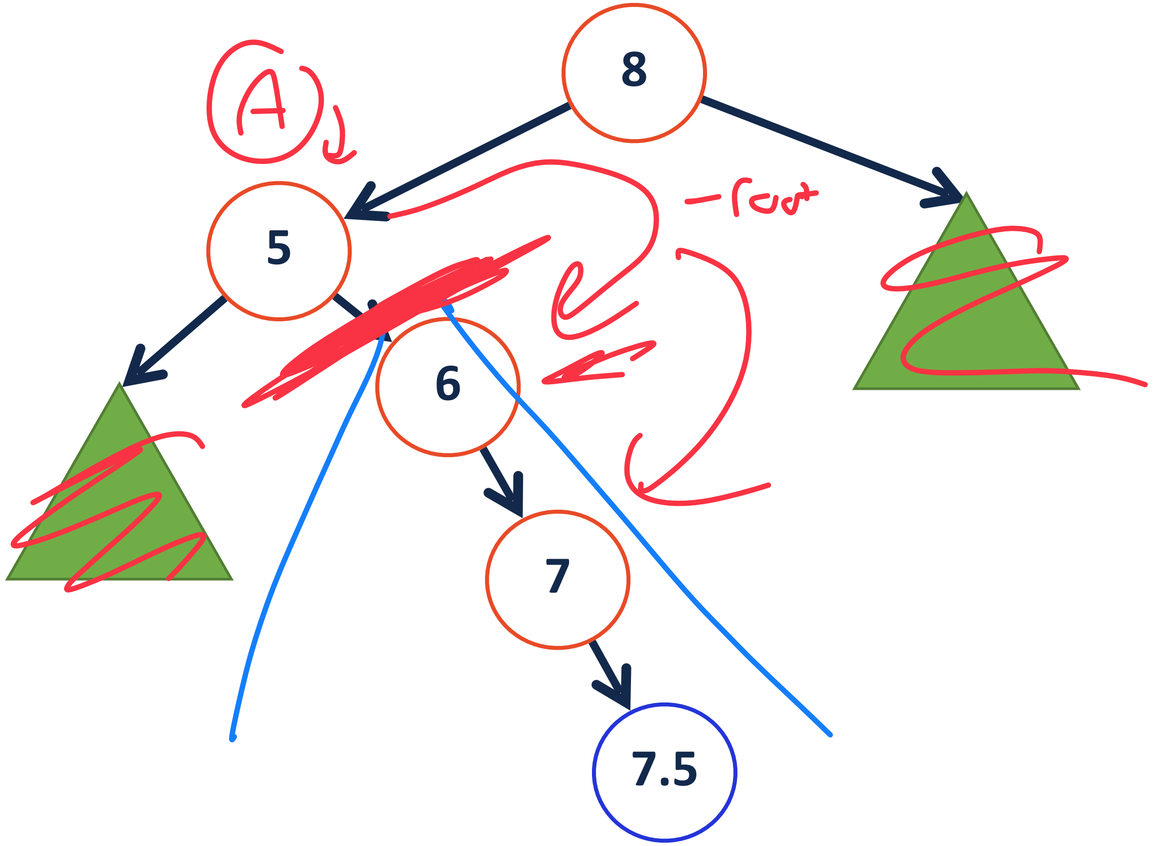
2) Set 6 to 7->left

3) 6->right = 7->left



# Left Rotation

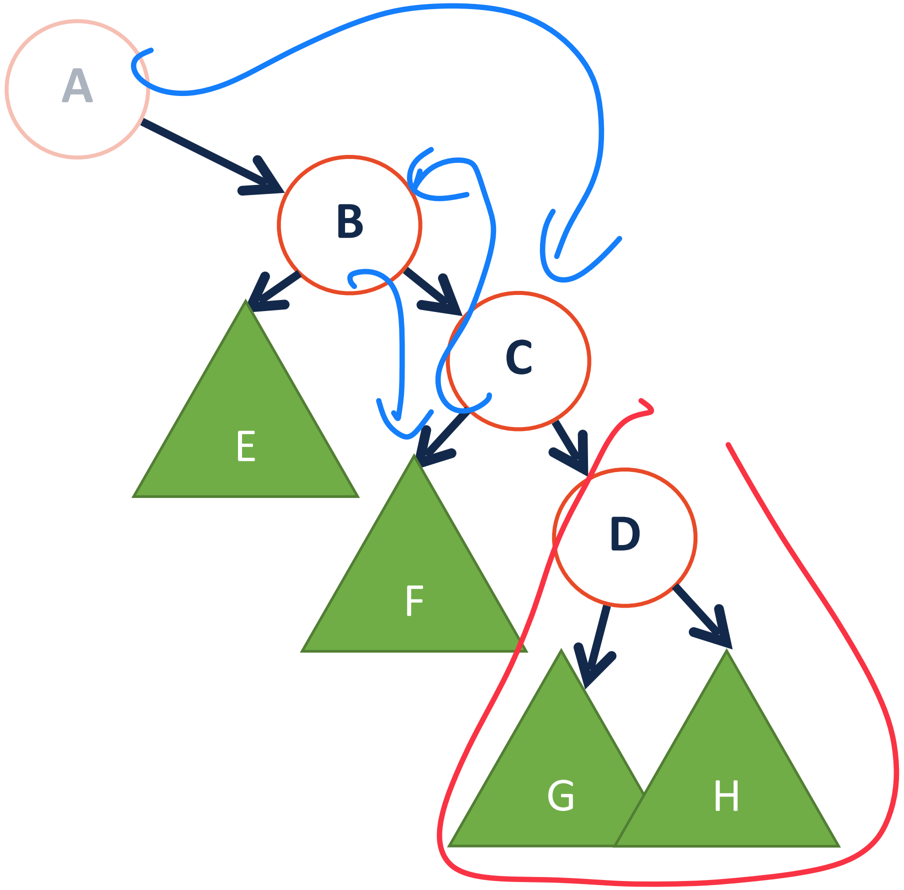
All rotations are local (subtrees are not impacted)

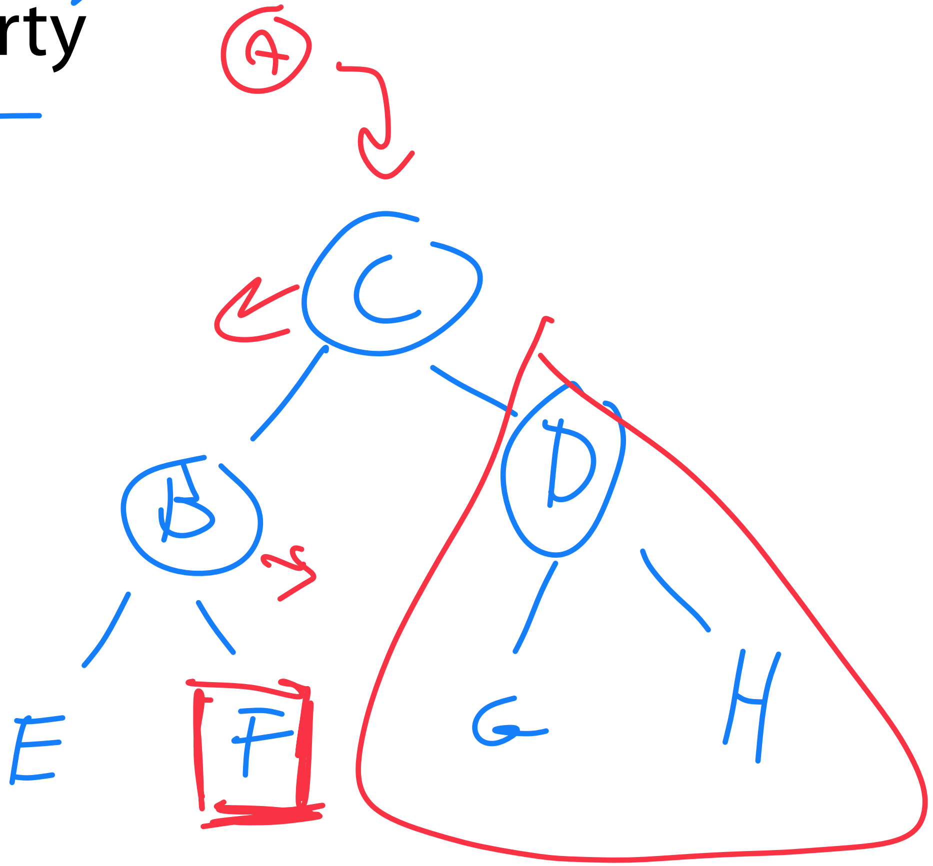


# Left Rotation

All rotations preserve BST property

tree balance



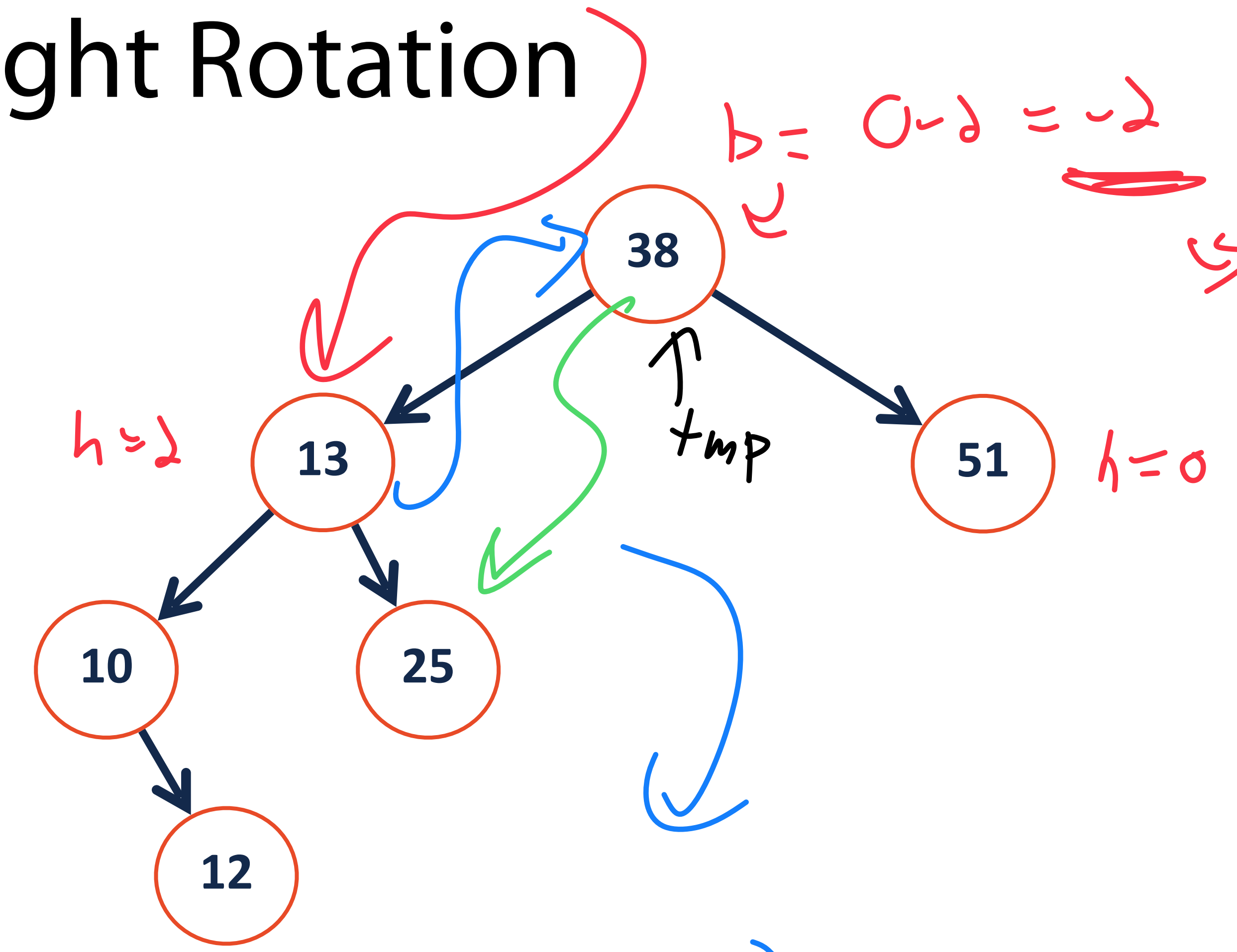

$B < F < C$

# Right Rotation

1) Compute balance

b = 0-2 = -2

→ negative # is left heavy




O(1) each

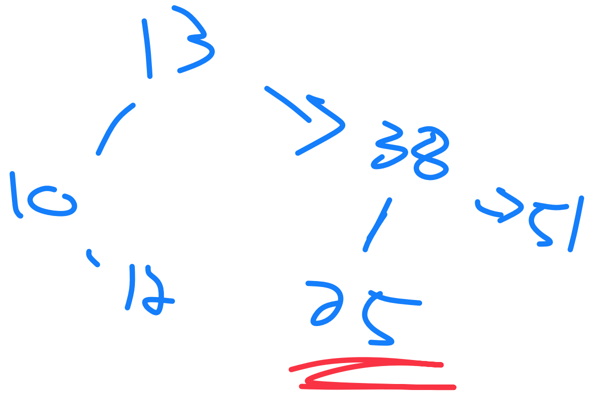
0) tmp = root

1) Set root to 13

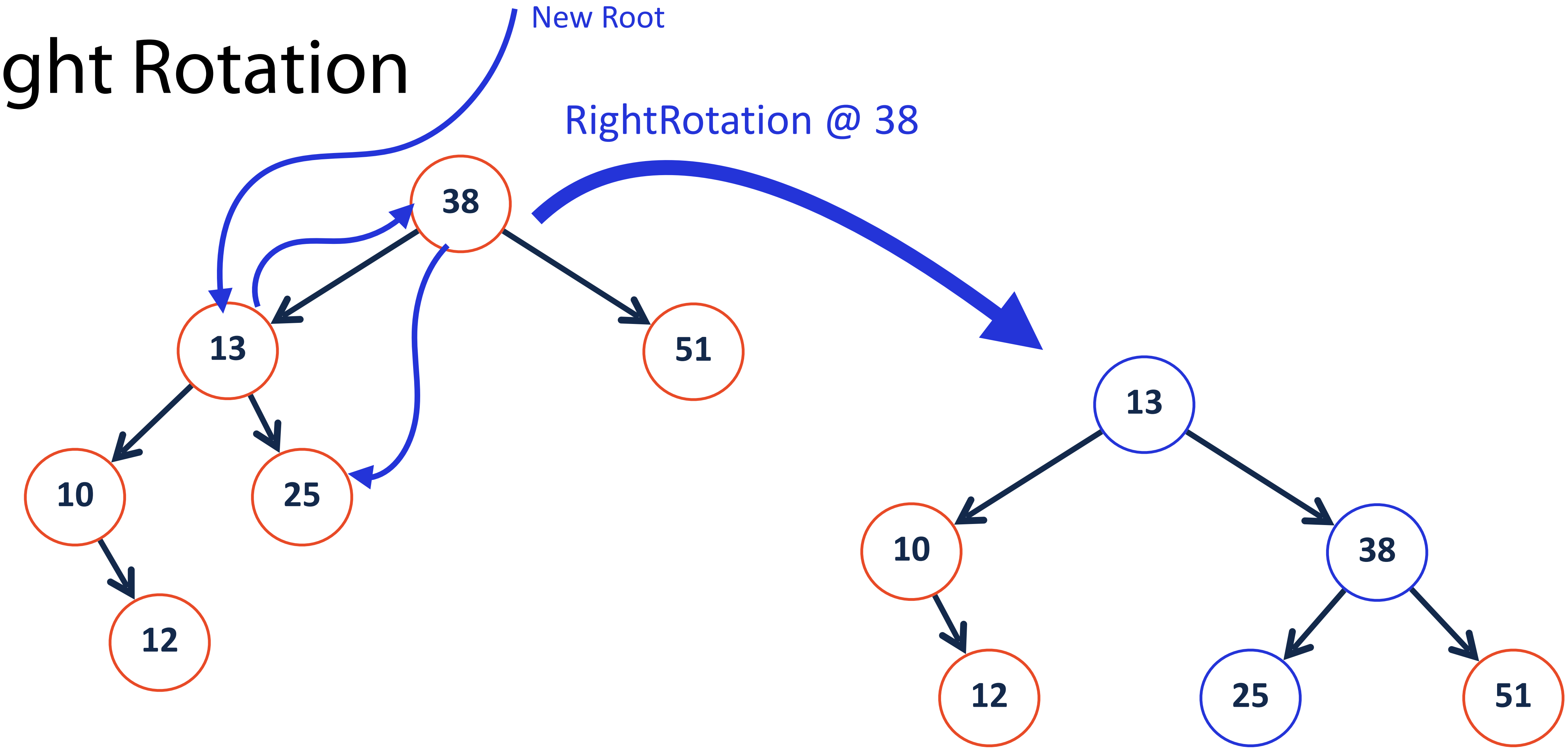
2) 38->left = 13->right

25

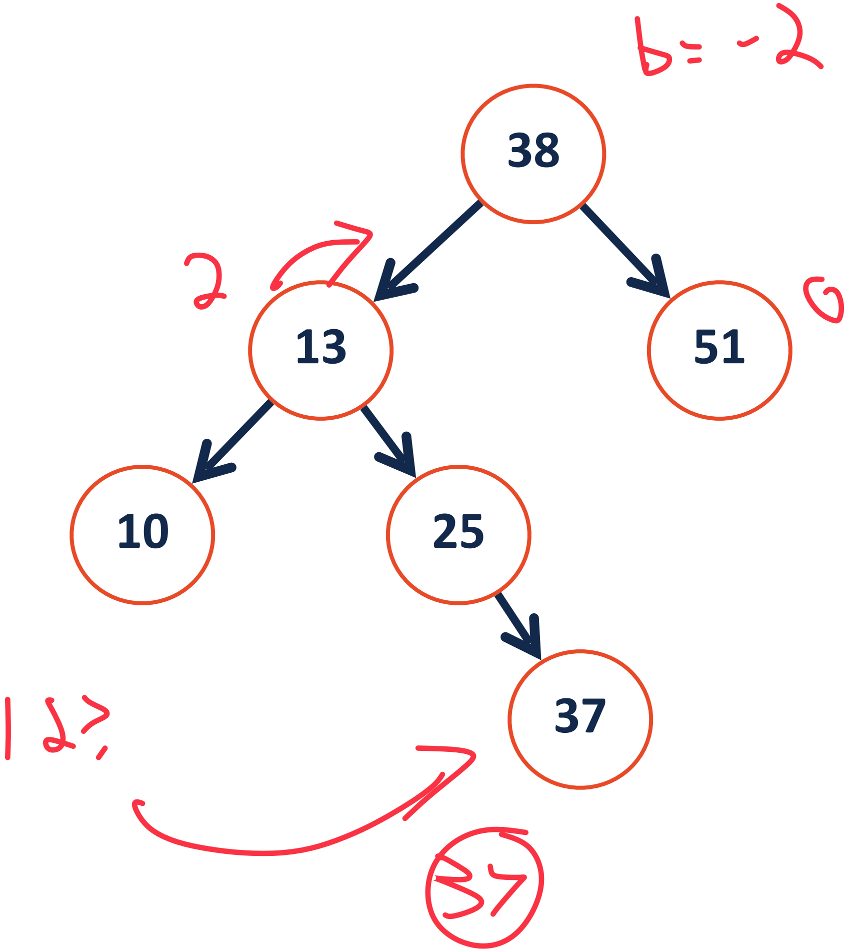
3) 13->right = 38

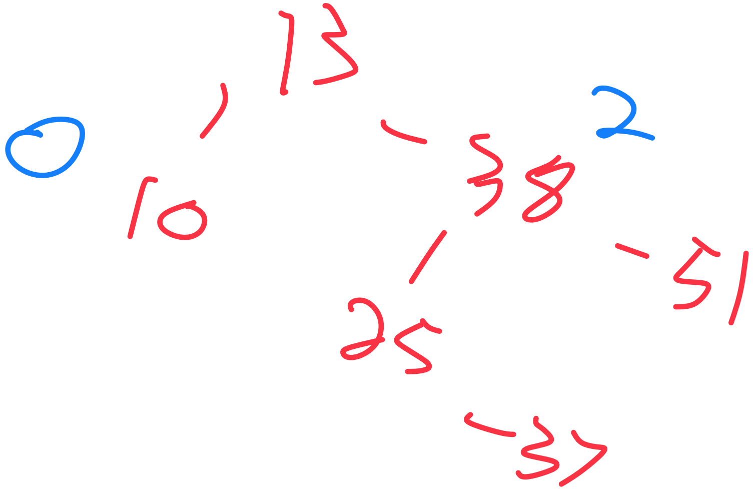


# Right Rotation

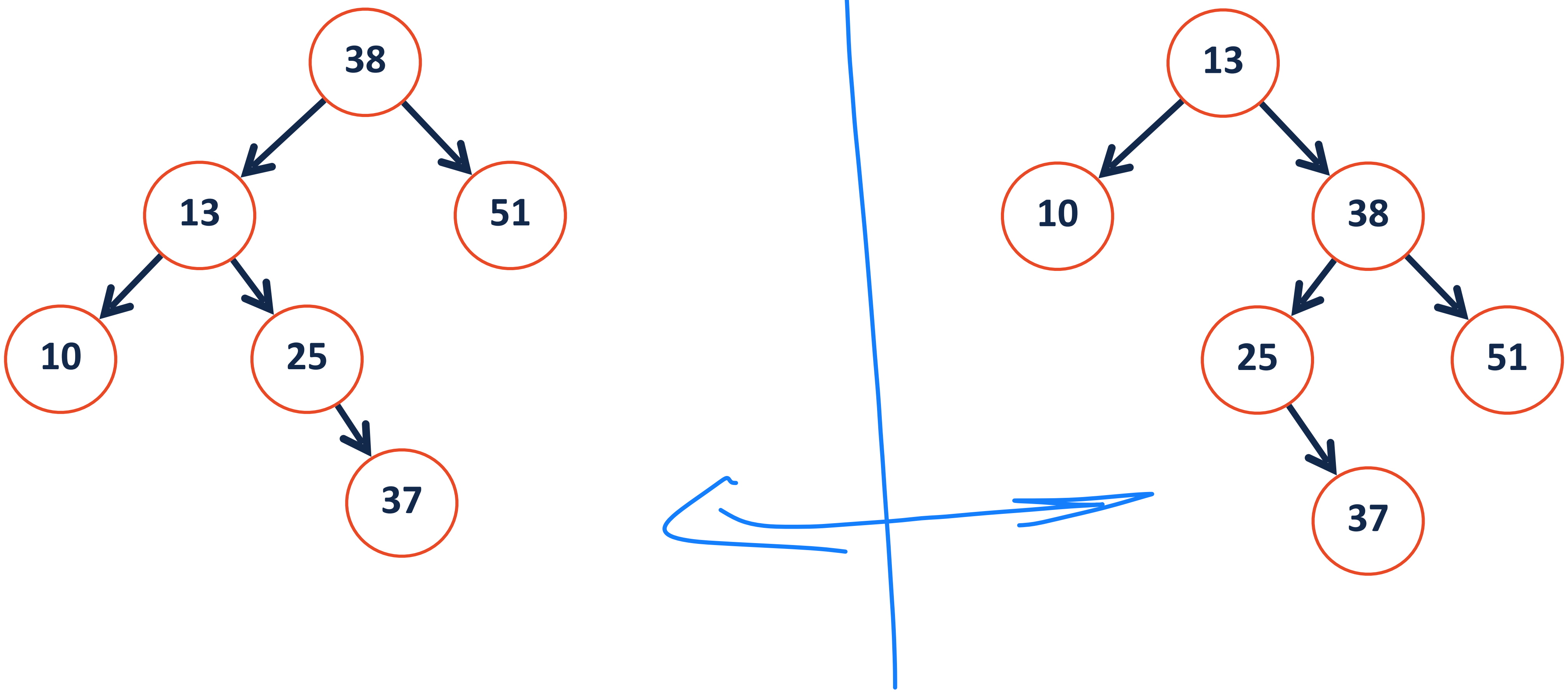


# AVL Rotation Practice



# AVL Rotation Practice

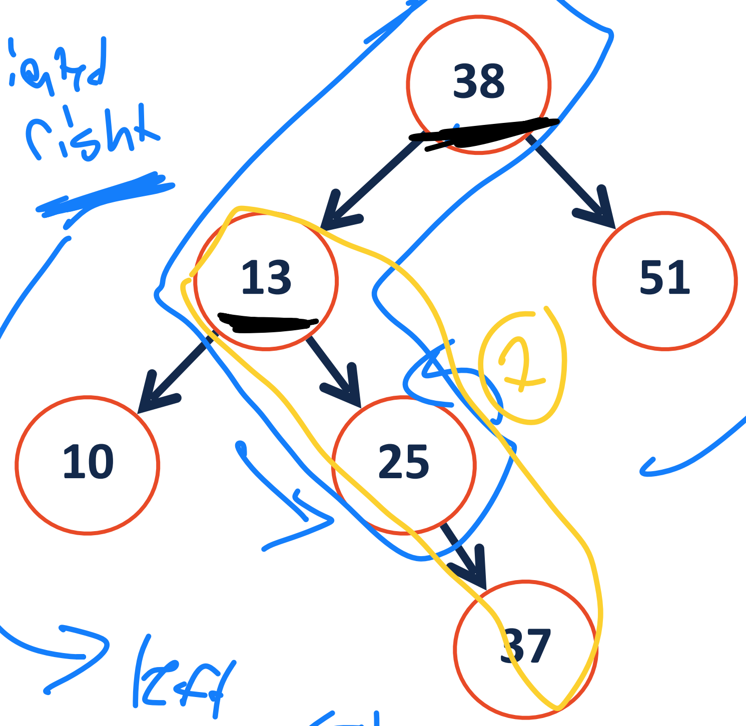



Somethings not quite right…

# LeftRight Rotation

left heavy

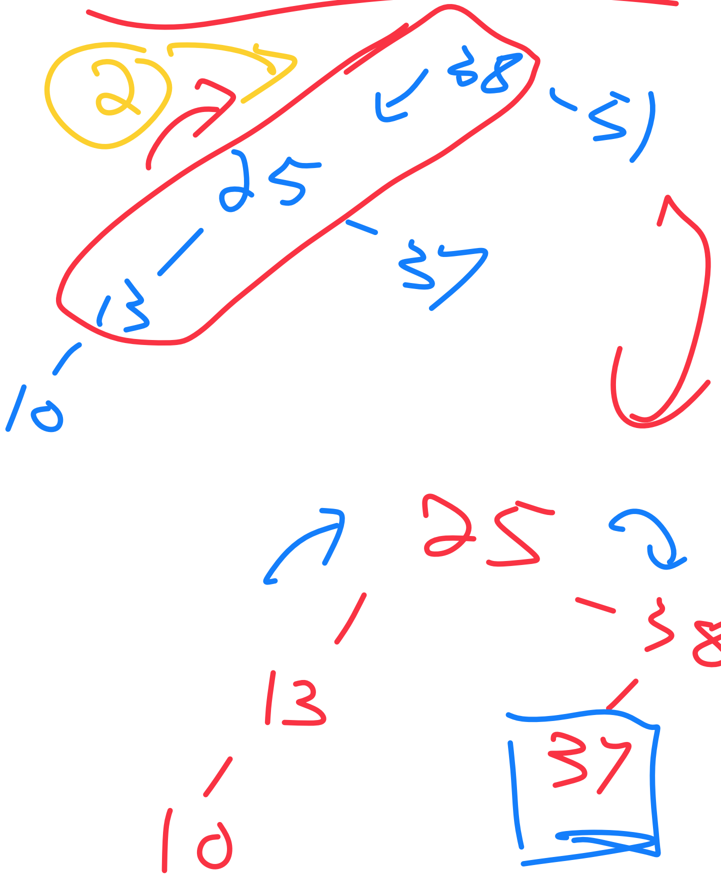
original right




left rotation!

1) Left rotation rooted @ 13

2) Right rotation rooted @ 38



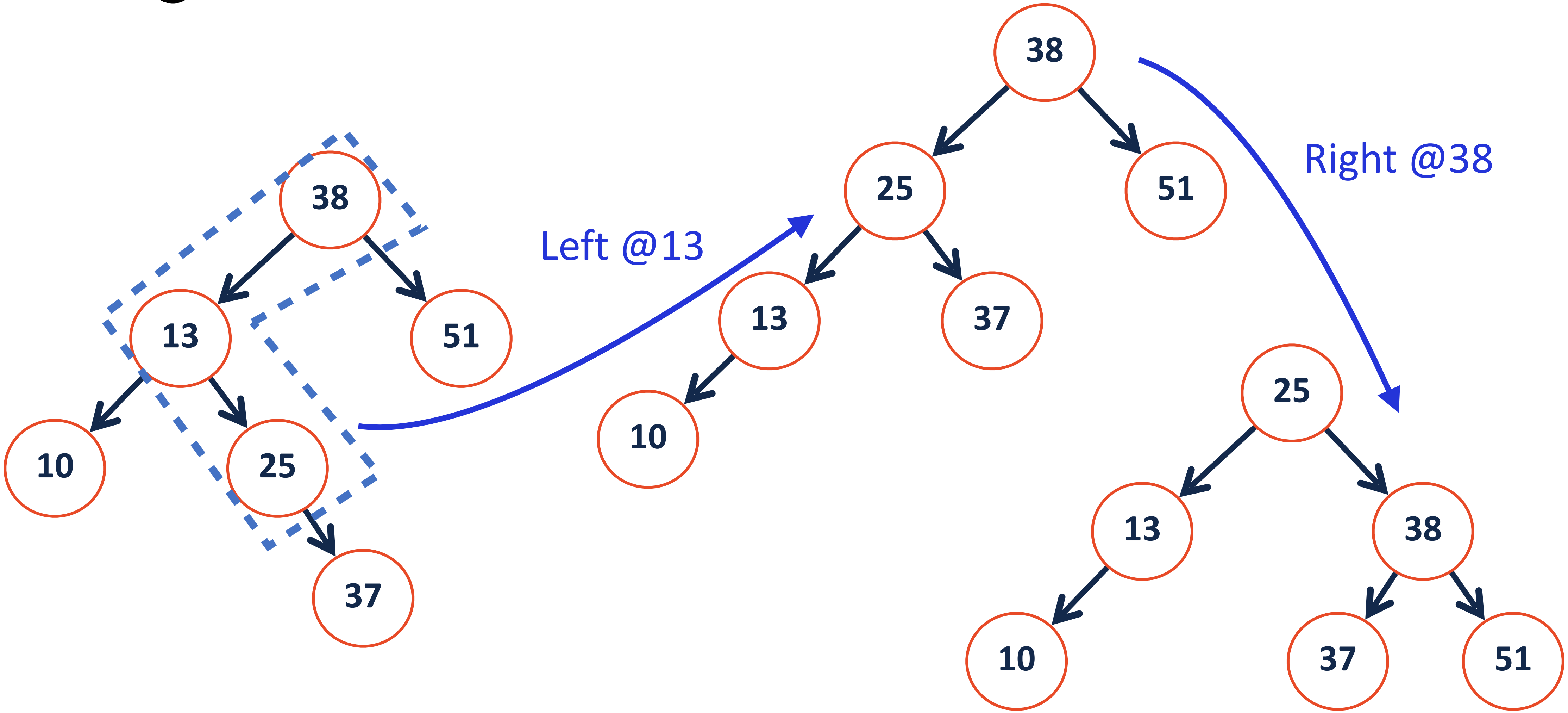

1) 25 become root

2) 38->left = 25->right

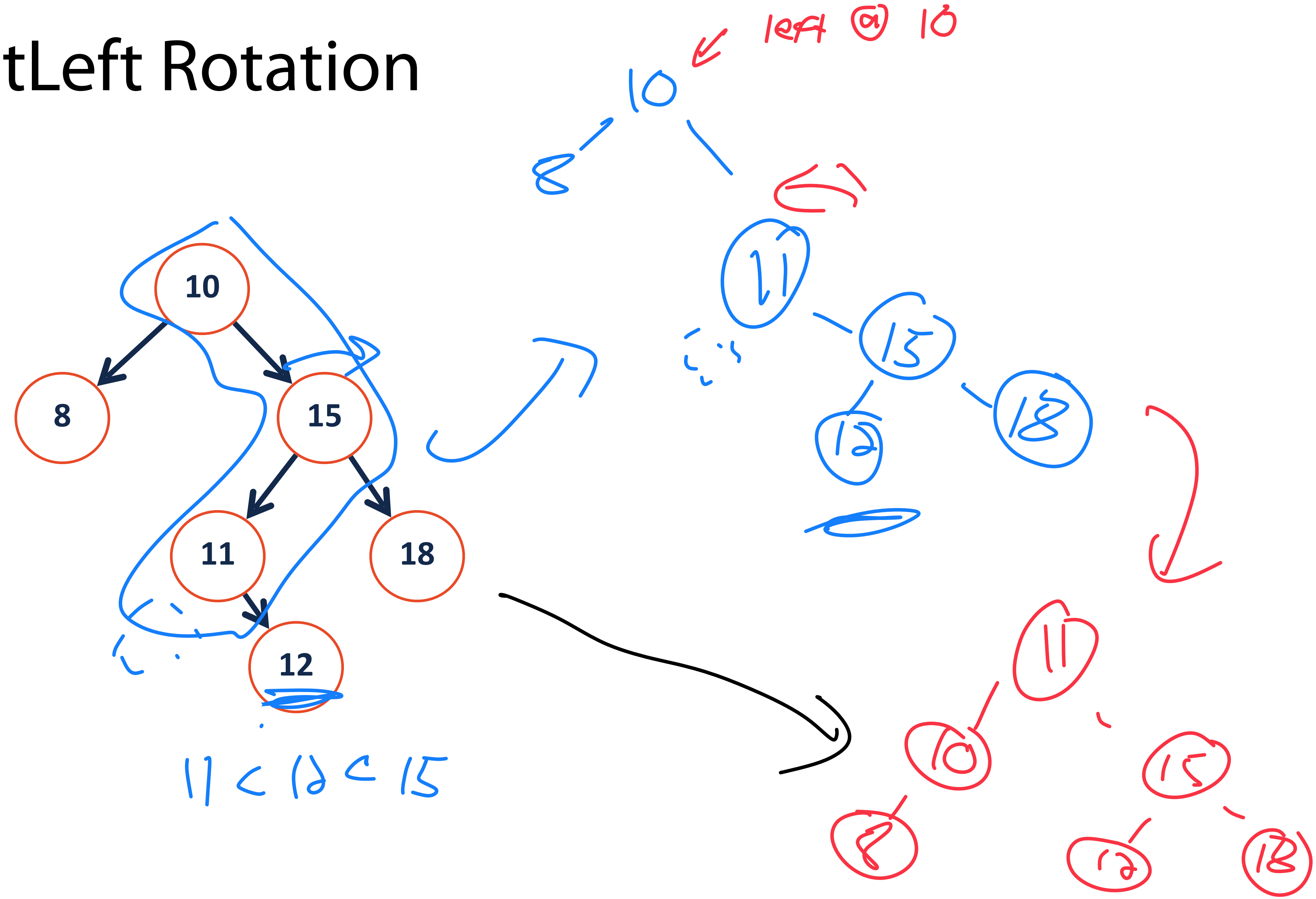
3.) 25->right = 38

has to go here

# LeftRight Rotation



# RightLeft Rotation

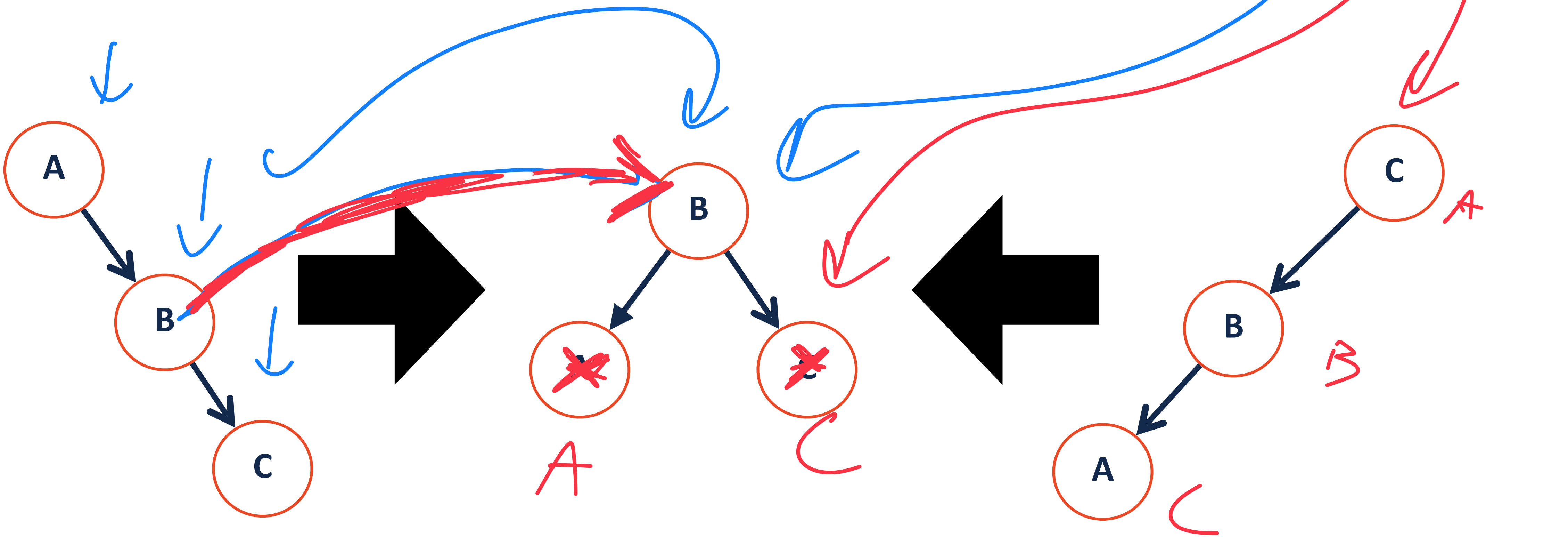


# AVL Rotations

memoize outcome?

A C B C C

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

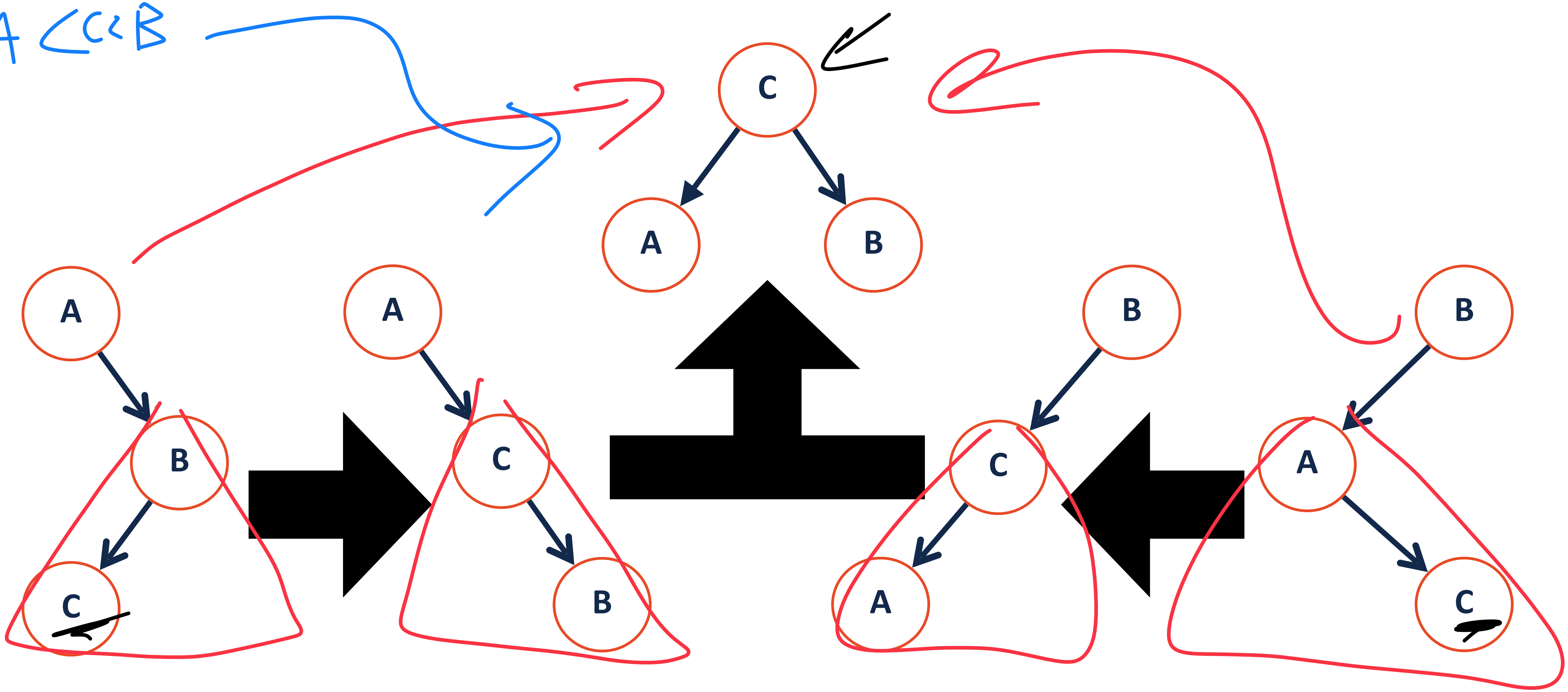



typo on canv slides

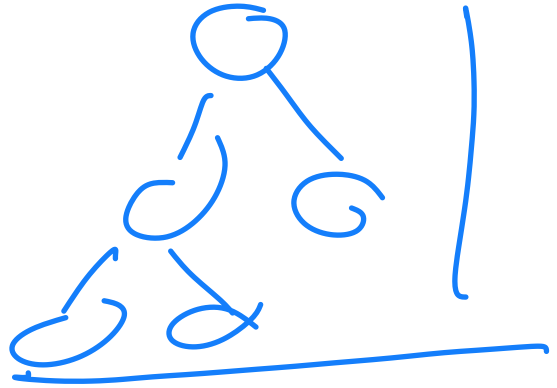
# AVL Rotations

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

A < C < B



# 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:** We want tree height to be $O(\log n)$

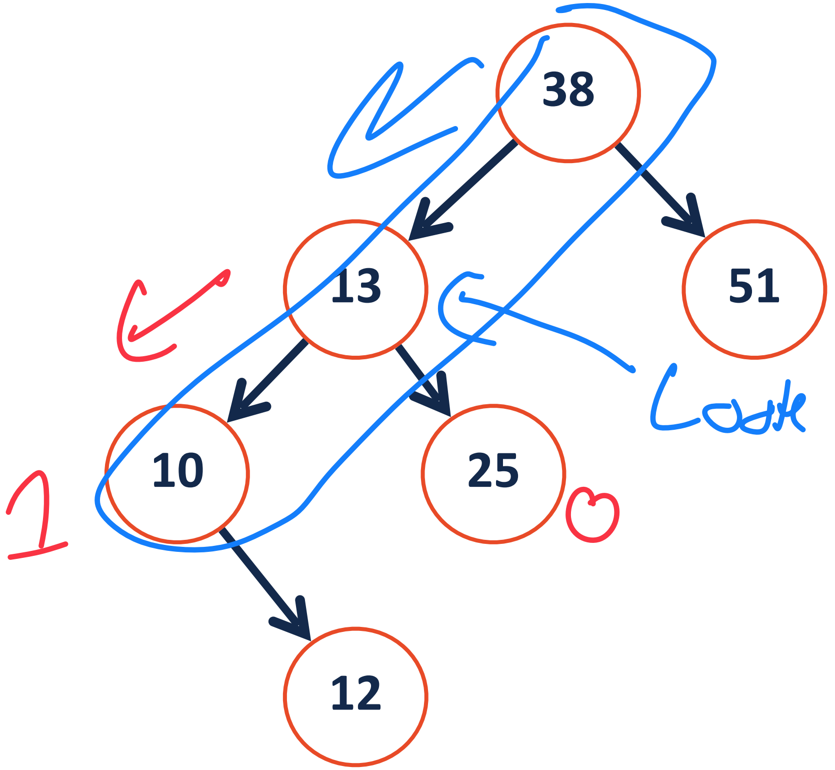
Balance our tree

# AVL Rotations

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

b = -2 ← I am unbalanced

↳ yes so look left



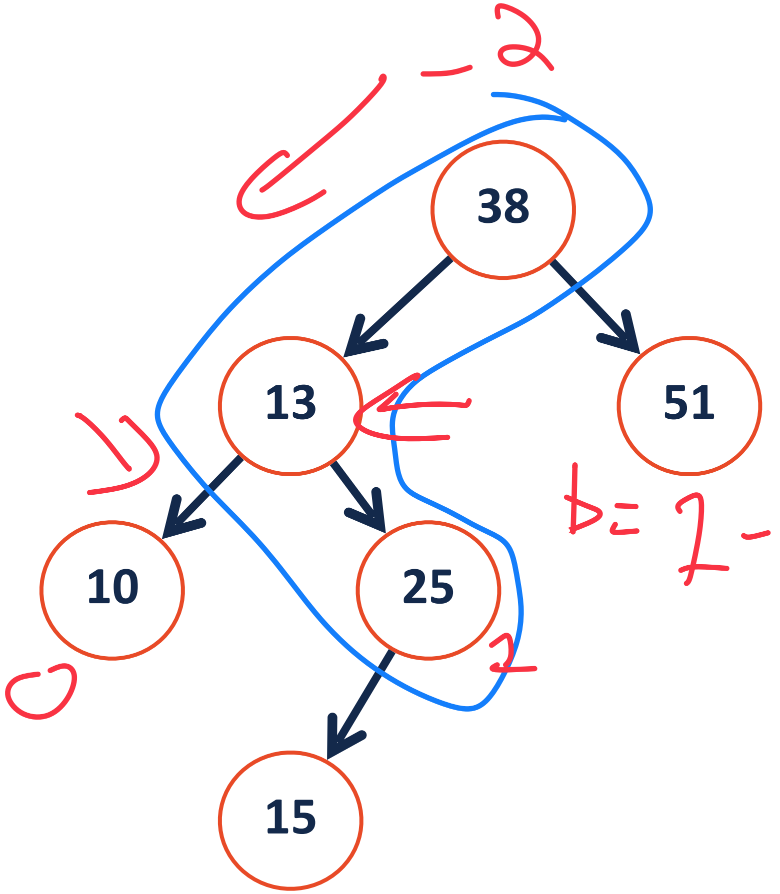

Look at left child

0 - 1 = -1

-2, -1 → right rotation

# AVL Rotations

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




b= 1-0=1

-2, 1

right ☹

left

stick

right ☺

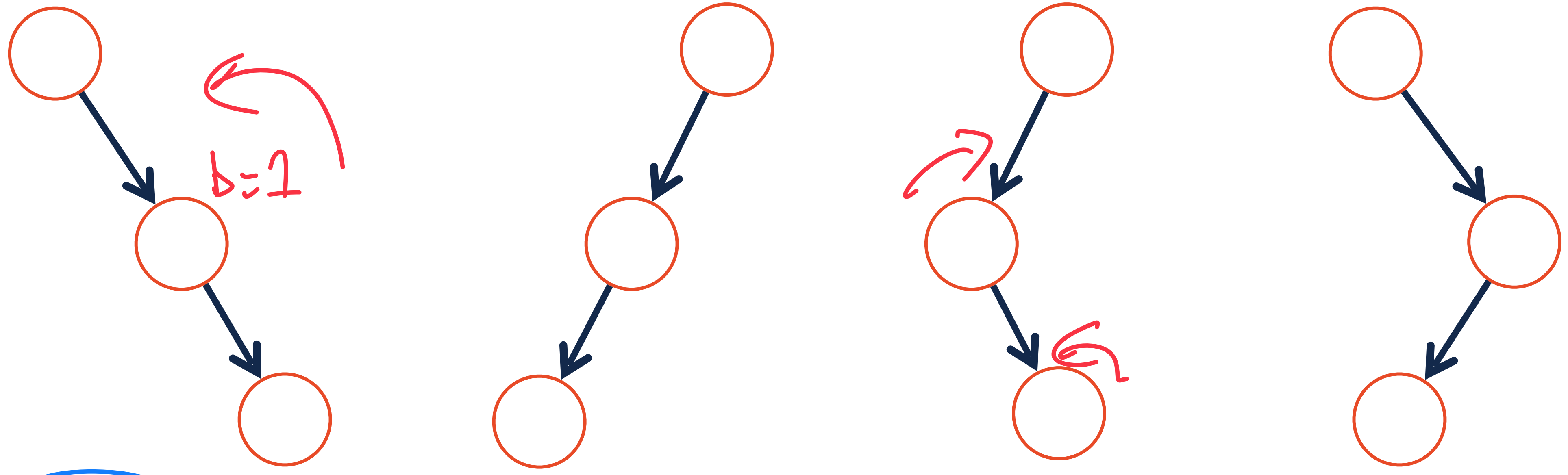
# AVL Rotations

b=2

Simple

Complex

Very important!




2, 1

Left

-2, -1

Right

-2, 1

Left Right

2, -1

Right Left

# AVL Rotation Practice

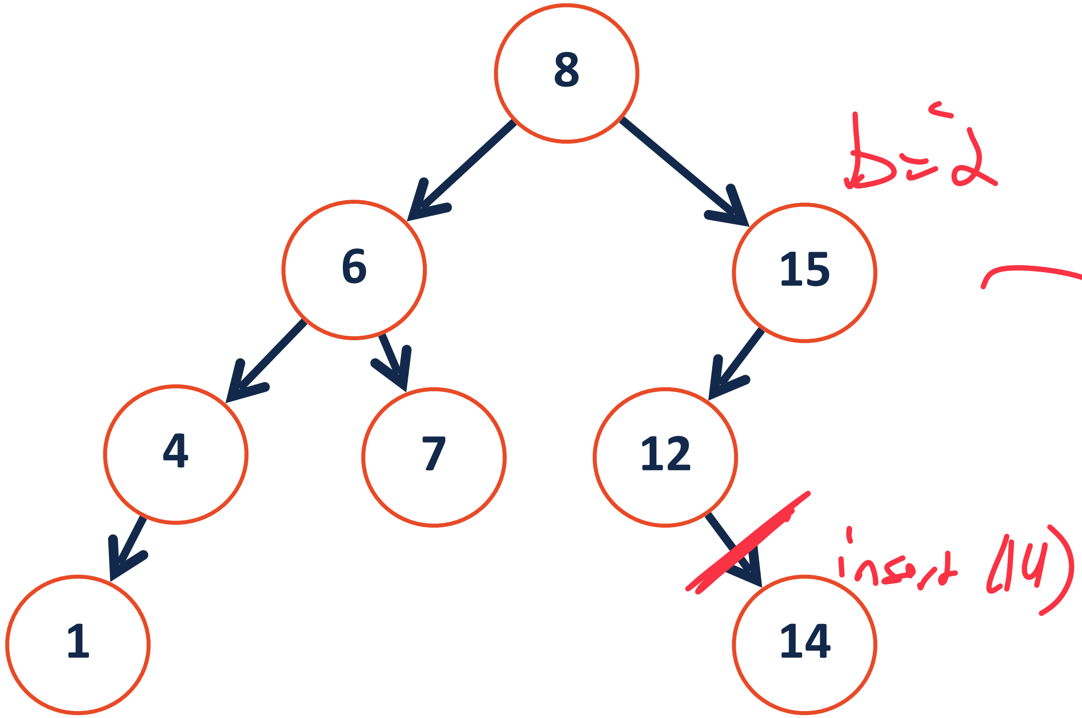
AVL tree is balanced $-1 \le b \le 1$

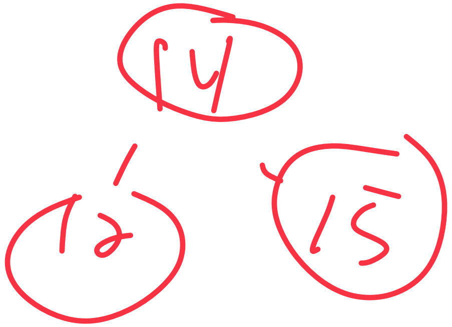
↳ every node was balanced

↳ only see $b = 2$ or $-2$

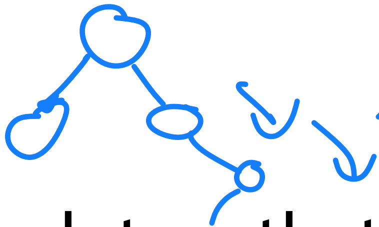
↳ Add one node at a time

↳



# AVL vs BST ADT



h = 1000000....

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?

+1, 0, -1

**Find**

**Insert**

**Remove**

must update height → O(1)

Tradeoff!

Height is slow to calc O(h)

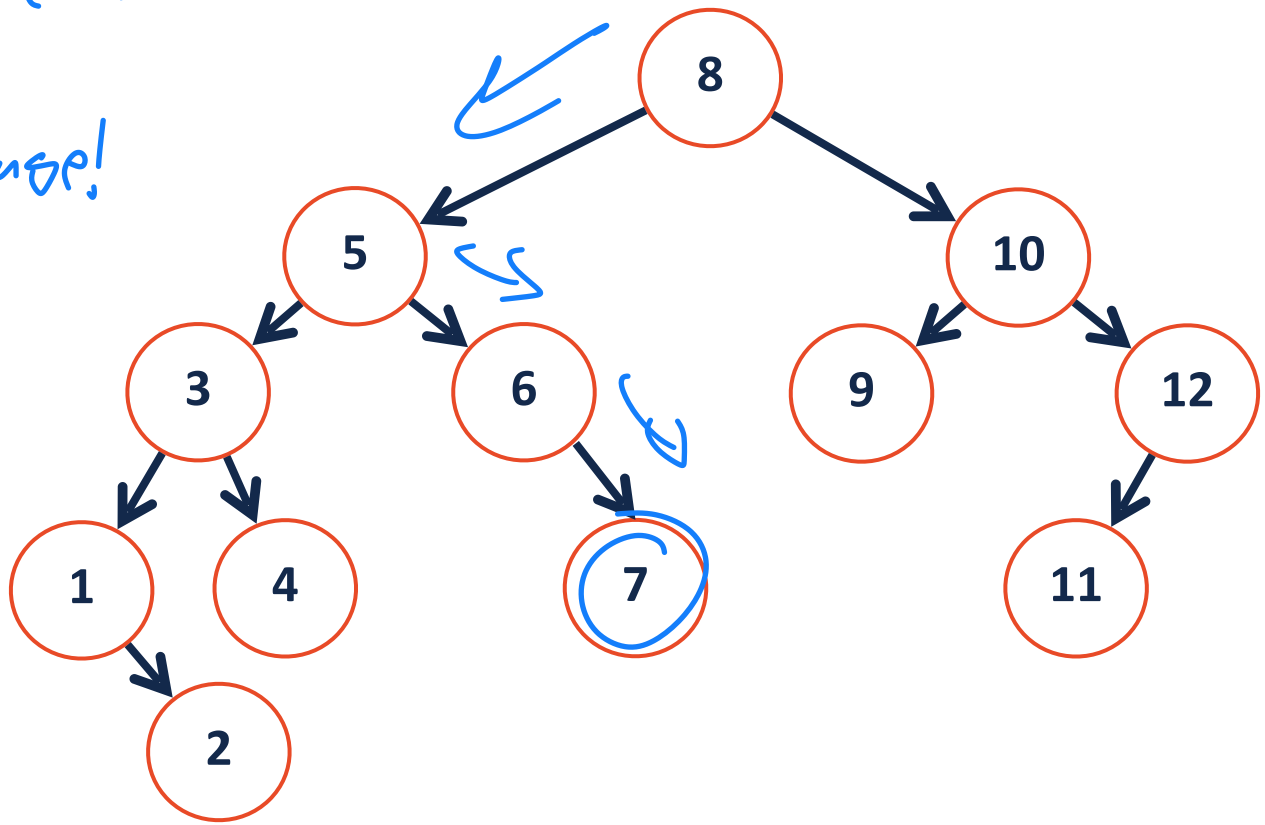
↳ Store it so its O(1)

# AVL Find

`_find(7)`

↳ BST find

↳ No change!



# AVL Insertion

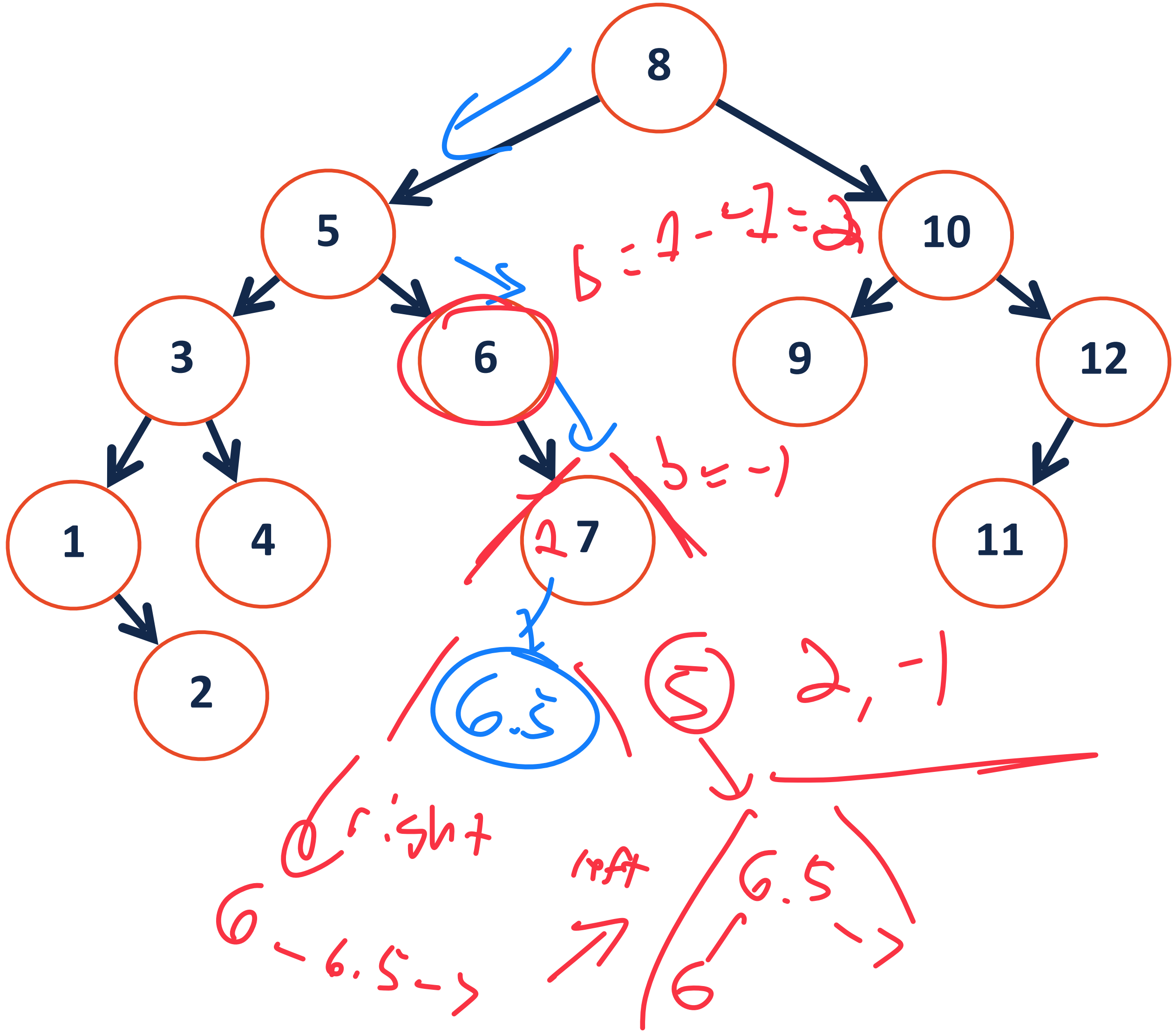
_insert(6.5)

1) Insert at proper place (BST insert)

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)`

Check

**Insert (recursive pseudo code):**

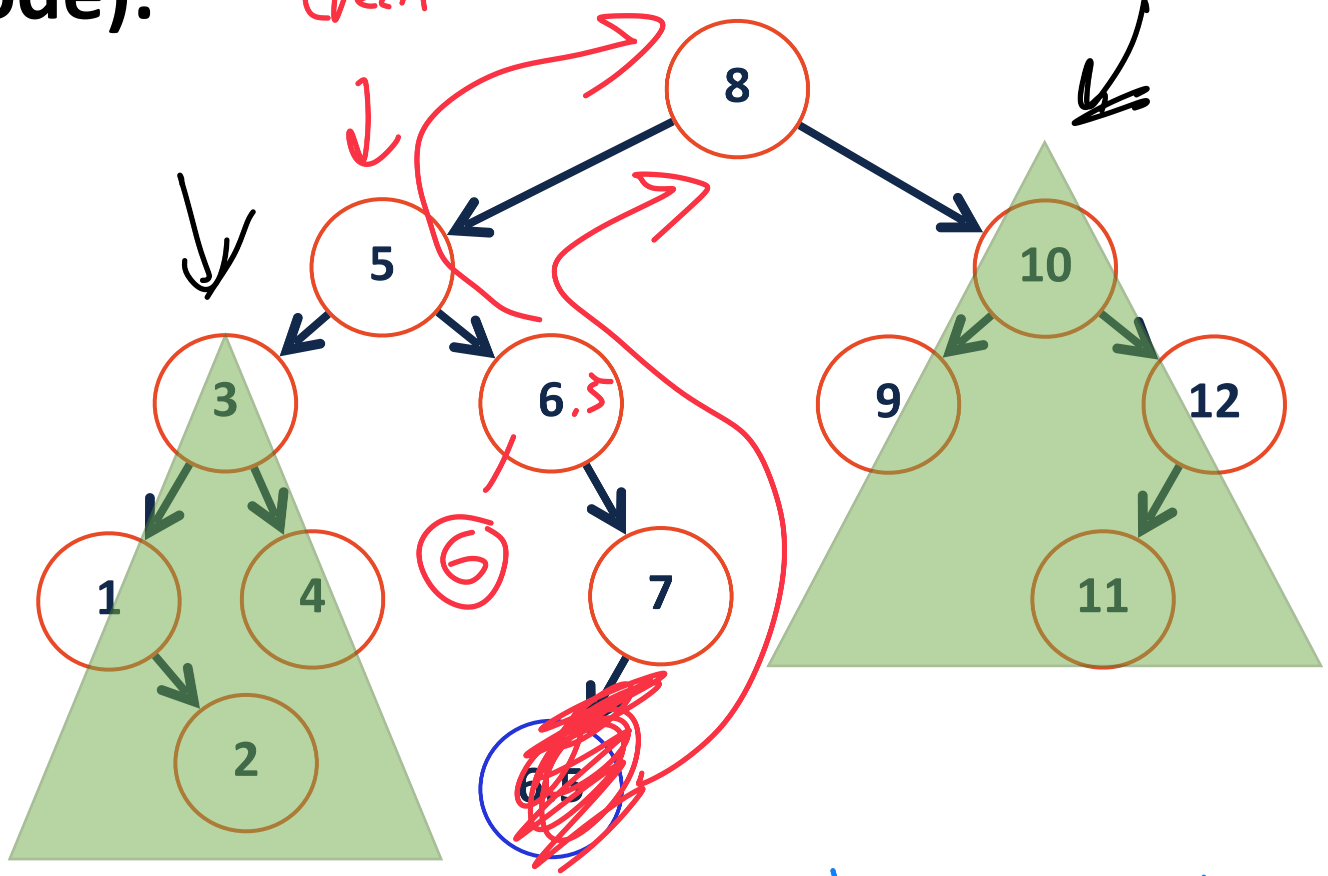
1: Insert at proper place

2: Check for imbalance

3: Rotate, if necessary

4: Update height

check




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

claim

We will only do 1 rotation max!

```
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);
}
```

check balance

```
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 ) { ________right________; }
    else                   { ____left  Right______; }
  } else if ( balance == 2 ) {
    int r_balance =
          height(cur->right->right) - height(cur->right->left);
    if( r_balance == 1 ) { ________left________; }
    else                 { _____Right Left_____; }
  }

  _updateHeight(cur);
};
```

-1 ≤ b ≤ 1