



# CS 225

## Data Structures

# AVL - Awful AVL Trees

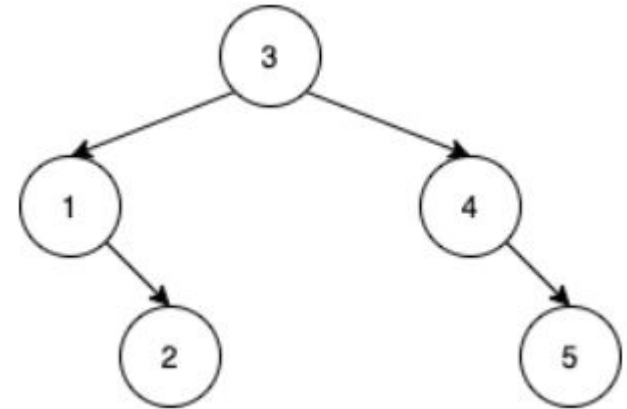
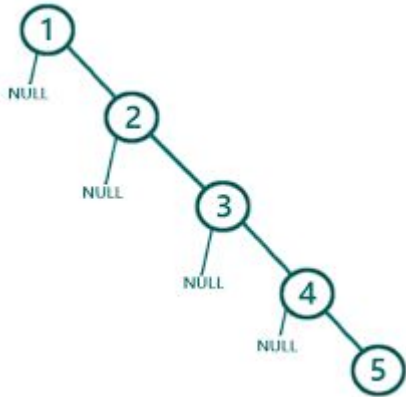
- AVL Tree Rotations
- Insertion
- Remove

# lab\_bst (Binary Search Trees)

■ Worst case

Vs

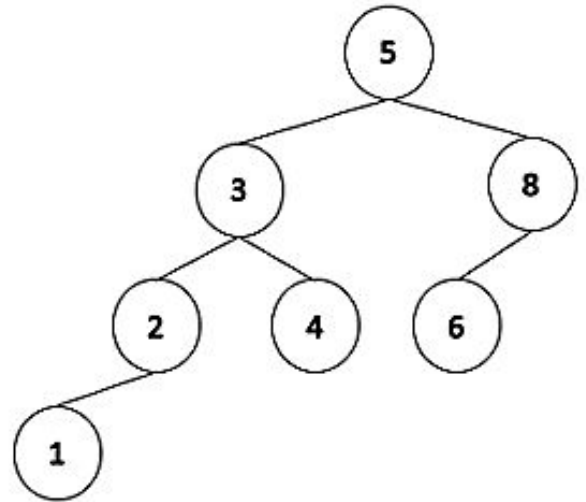
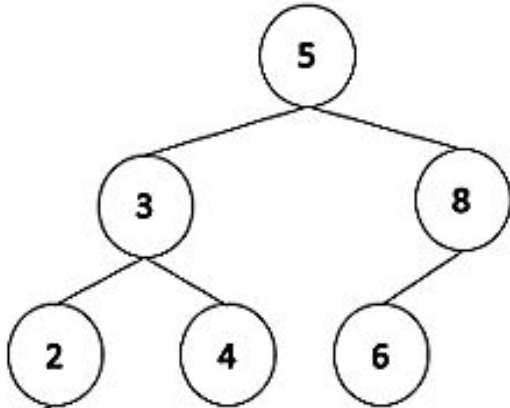
Expected case



How could this be improved?

# AVL Trees are balanced Binary search trees:

- The sub-trees of every node differ in height by at most one.
- Every sub-tree is an AVL tree

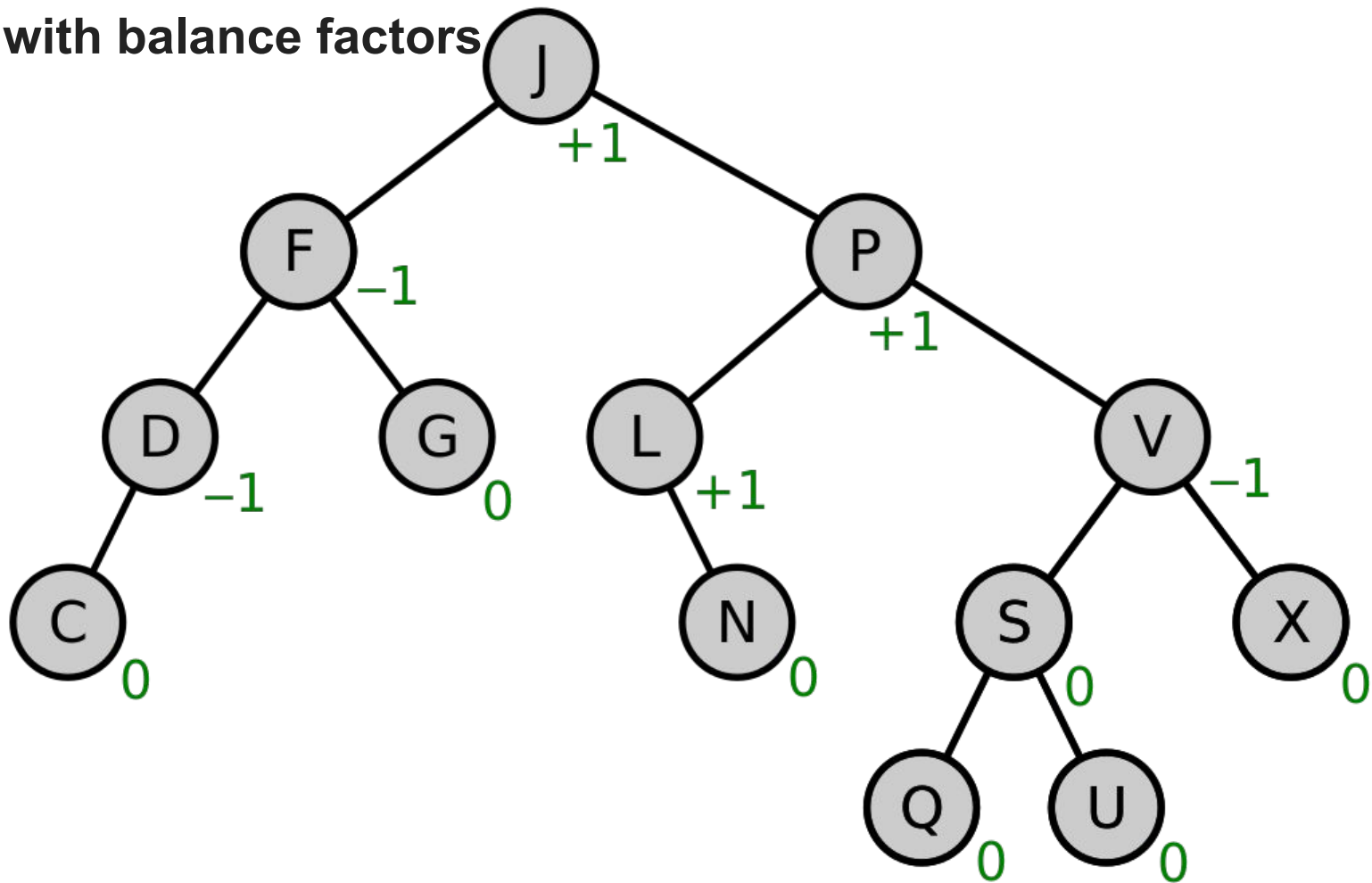


# Balance Factor:

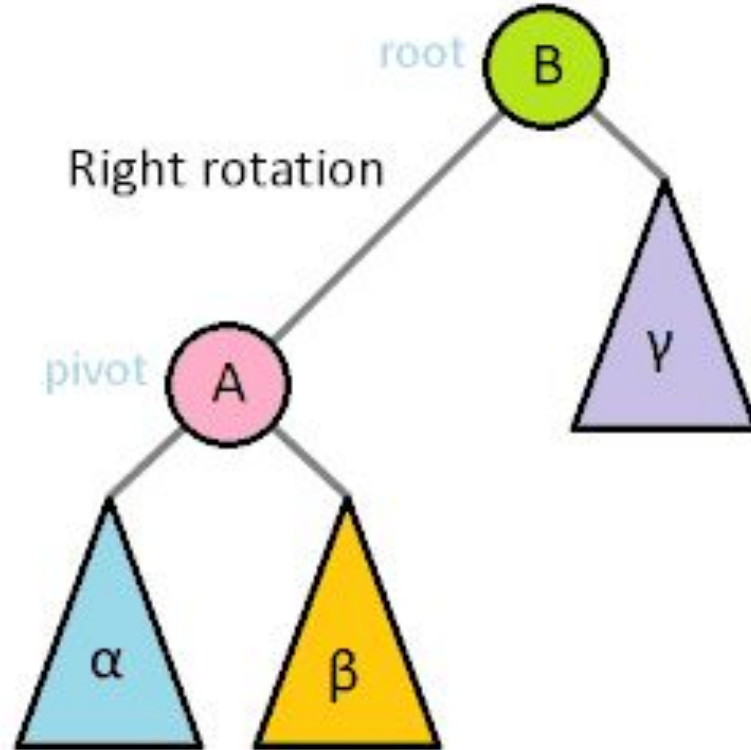
$$b = \text{height}(T_R) - \text{height}(T_L)$$

- By definition, a tree is balanced if the absolute value of the balance factor is less than or equal to 1  $\rightarrow |\mathbf{b}| \leq 1$
- Balance is determined locally (for each node).

## AVL tree with balance factors

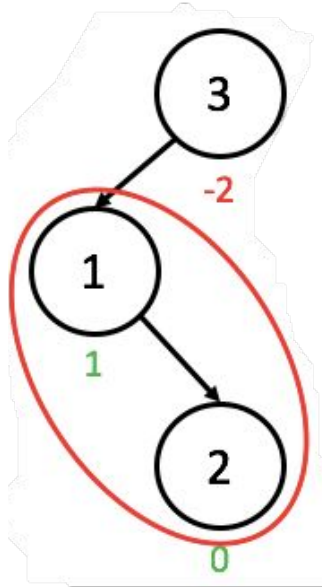


# Single Rotations:



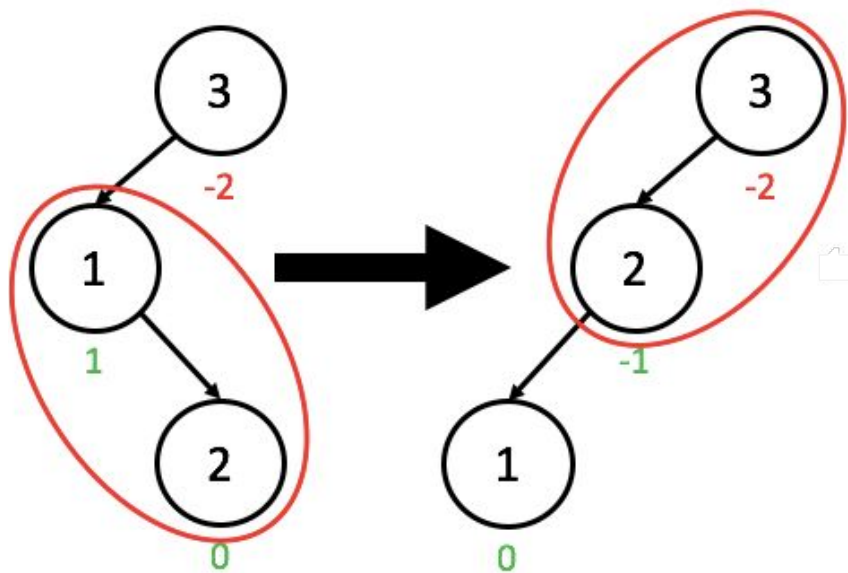
Link: <https://wkdjtjsgur100.github.io/avl-tree/>

# Double Rotation: An example

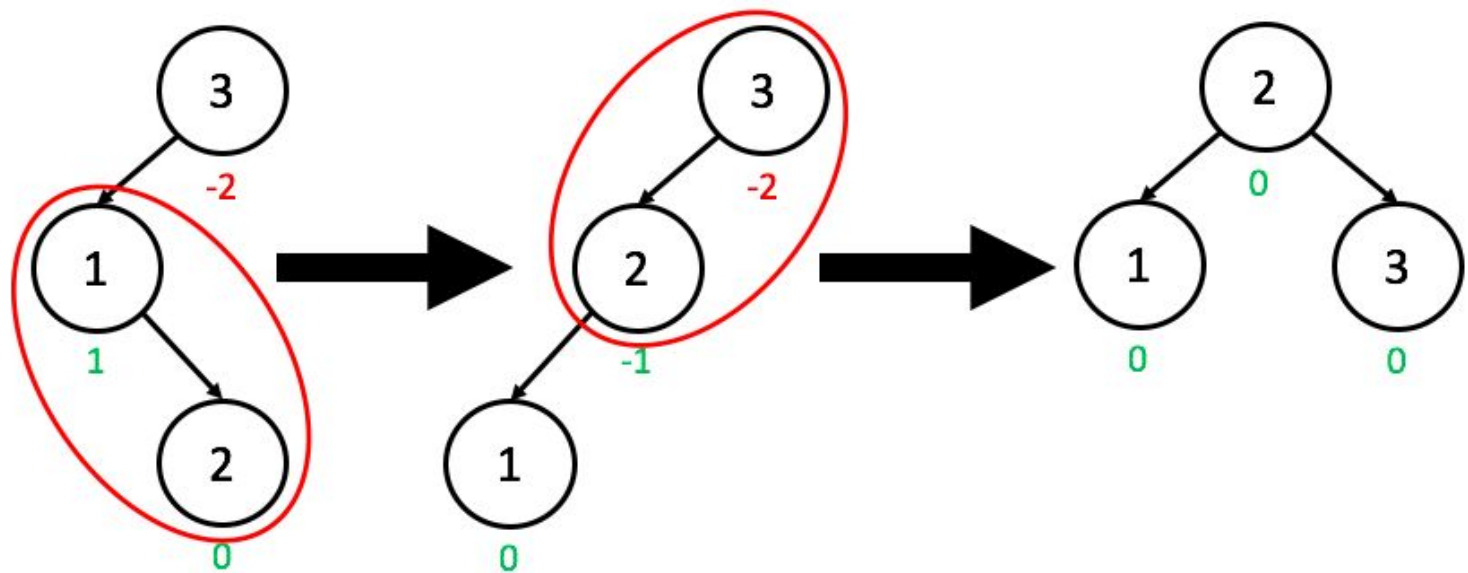




# Double Rotation

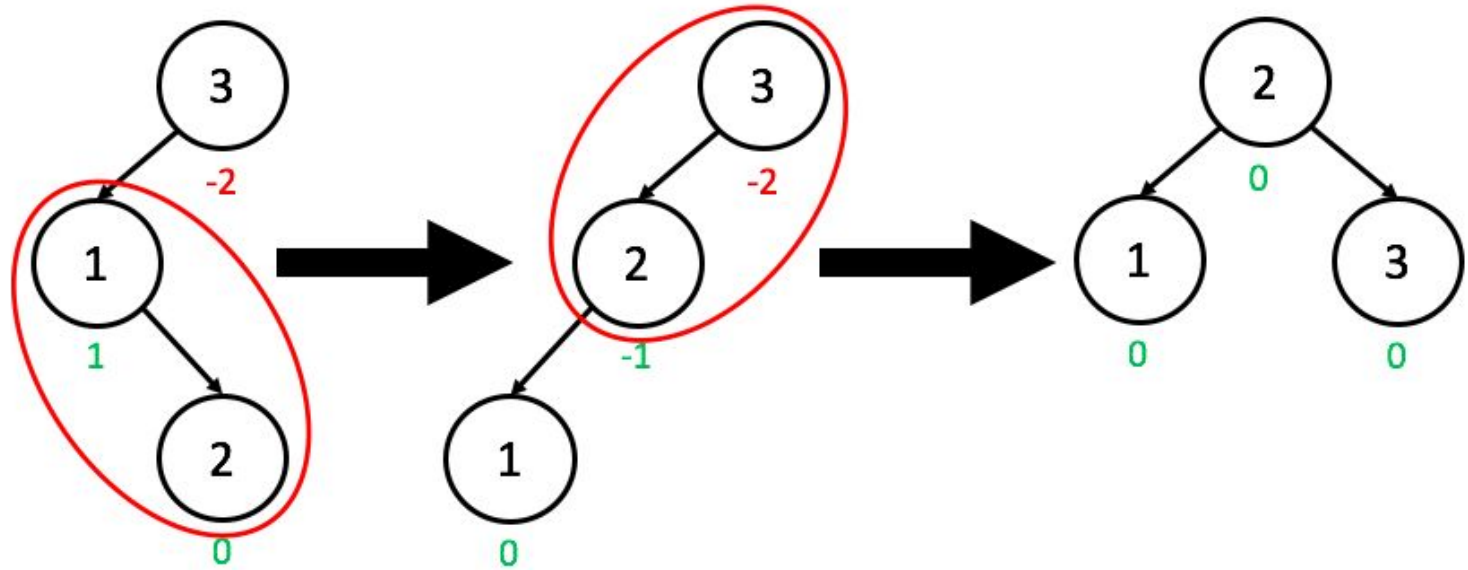


# Double Rotation



*What type of rotation is this?*

# Double Rotation

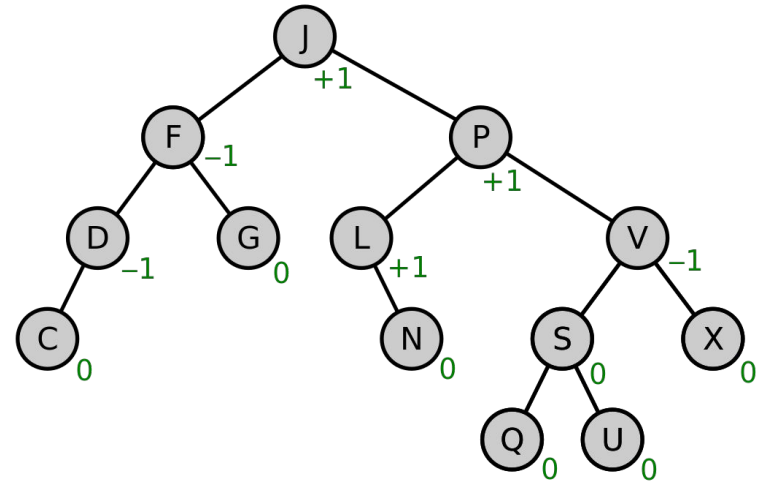


Left-Right rotation.

*Implementation provided in the lab*

# Rebalancing:

- Simple rotation:
  - ✓ Left
  - ✓ Right
- Double rotation
  - ✓ Left-Right
  - ✓ Right-Left



When do I rebalance left? right? left-right? right-left?

# Rebalancing:

- Simple rotation:

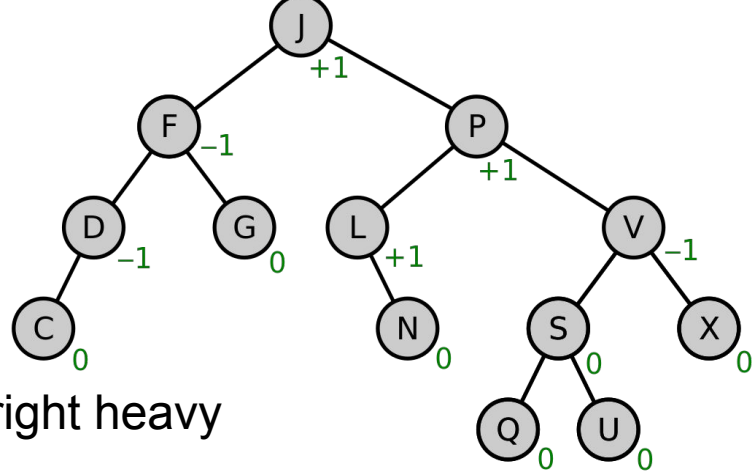
- ✓ Left      Root and Right child are right heavy

- ✓ Right      Root and Left child are left heavy

- Double rotation

- ✓ Left-Right      Root is left heavy and left child is right heavy

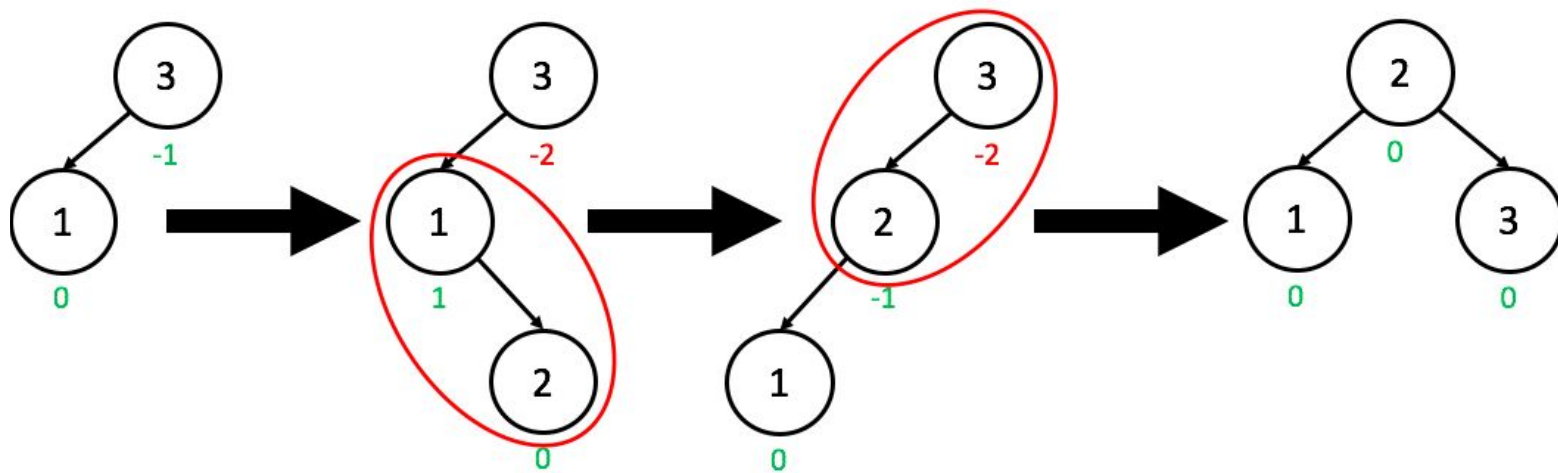
- ✓ Right-Left      Root is right heavy and right child is left heavy



Where can I insert a node to cause a left rotation? right? left-right?

# Insertion

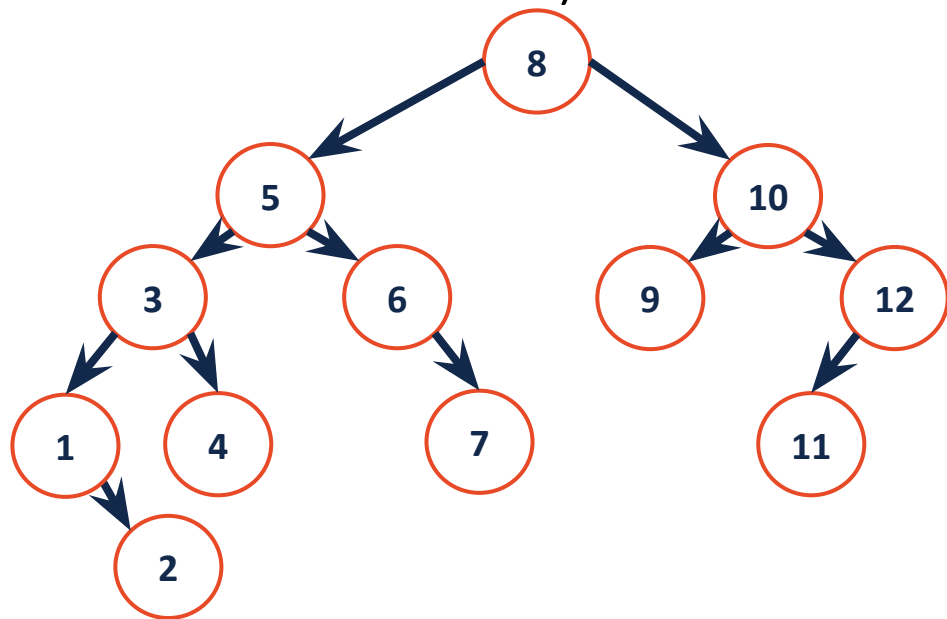
- Perform standard BST insertion for  $w$ ;
- Re-balance the tree if necessary – start from  $w$  travel up to  $root$ , find first unbalanced node and perform appropriate rotation.



## Insertion into an AVL Tree

### Insert (pseudo code):

- 1: Insert at proper place (use recursive calls to walk down the path)
- 2: Check for imbalance (when you unwind the recursion)
- 3: Rotate, if necessary
- 4: Update height



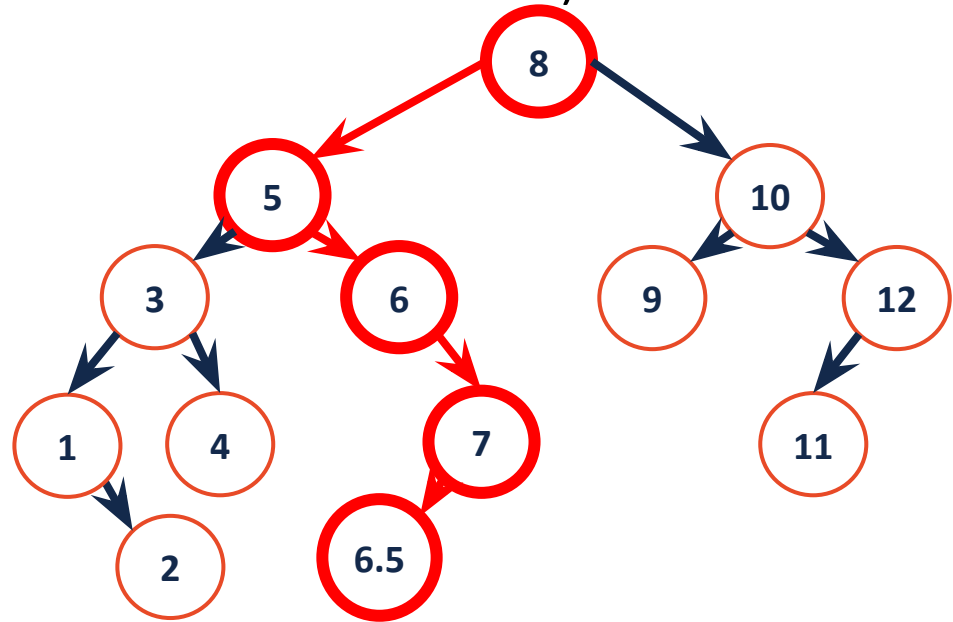
```
1 struct TreeNode {  
2     T key;  
3     unsigned height;  
4     TreeNode *left;  
5     TreeNode *right;  
6 };
```

## Insertion into an AVL Tree

### Insert (pseudo code):

- 1: Insert at proper place (use recursive calls to walk down the path)
- 2: Check for imbalance (when you unwind the recursion)
- 3: Rotate, if necessary
- 4: Update height

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





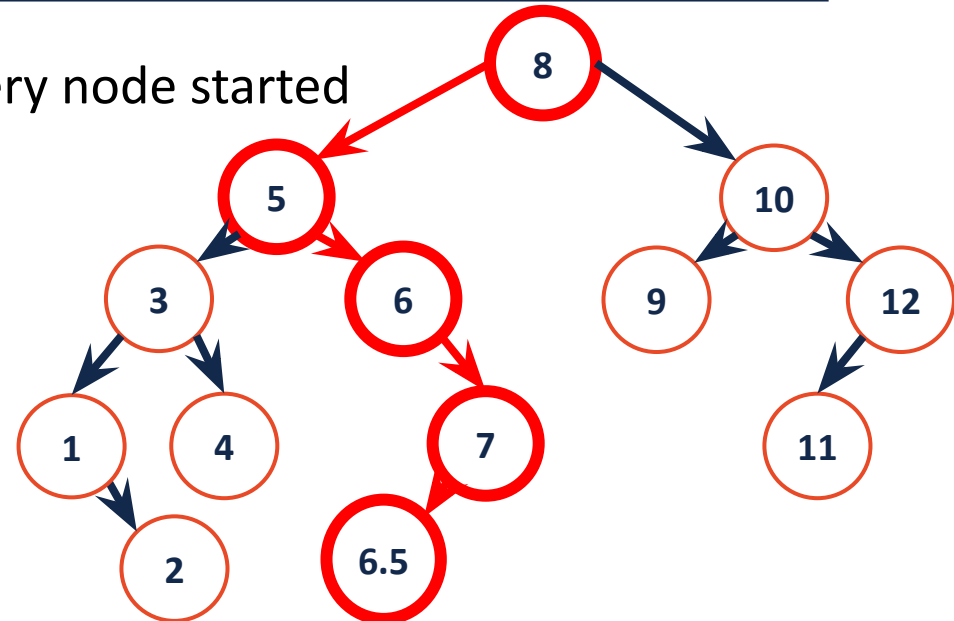
```

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 }

```

- Insertion calls itself h time (for every node started from root to the leaf node);
- Each insert takes  $O(1)$  time;
- After insertion we need max one rotation to balance the tree:  $O(1)$

✓ Running time of insert is  $O(h)$



# Remove from an AVL Tree

AVL tree remove works same as removing element from BST, except we have to rebalance tree if necessary.

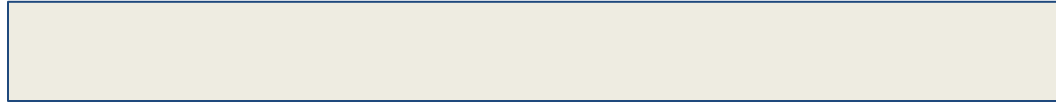
- Perform standard BST delete for the given node  $n$ ;
- Check the nodes for the imbalance once you unwind the recursion.

*Fixing the first lowest point of imbalance does not balance the whole tree!  
You need to check every node, going up from deleted node to the root node;*

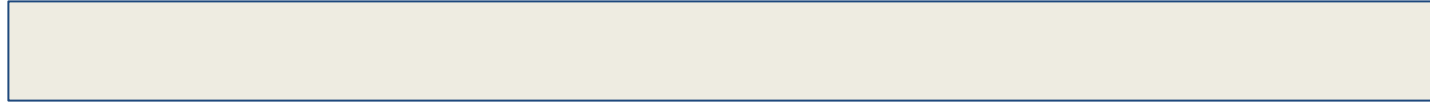
# Remove from an AVL Tree

Find the element  $n$  and if:


- has no child:

A light beige rectangular box with a thin blue border, intended for a diagram of a node with no children.

- has one child

A light beige rectangular box with a thin blue border, intended for a diagram of a node with one child.

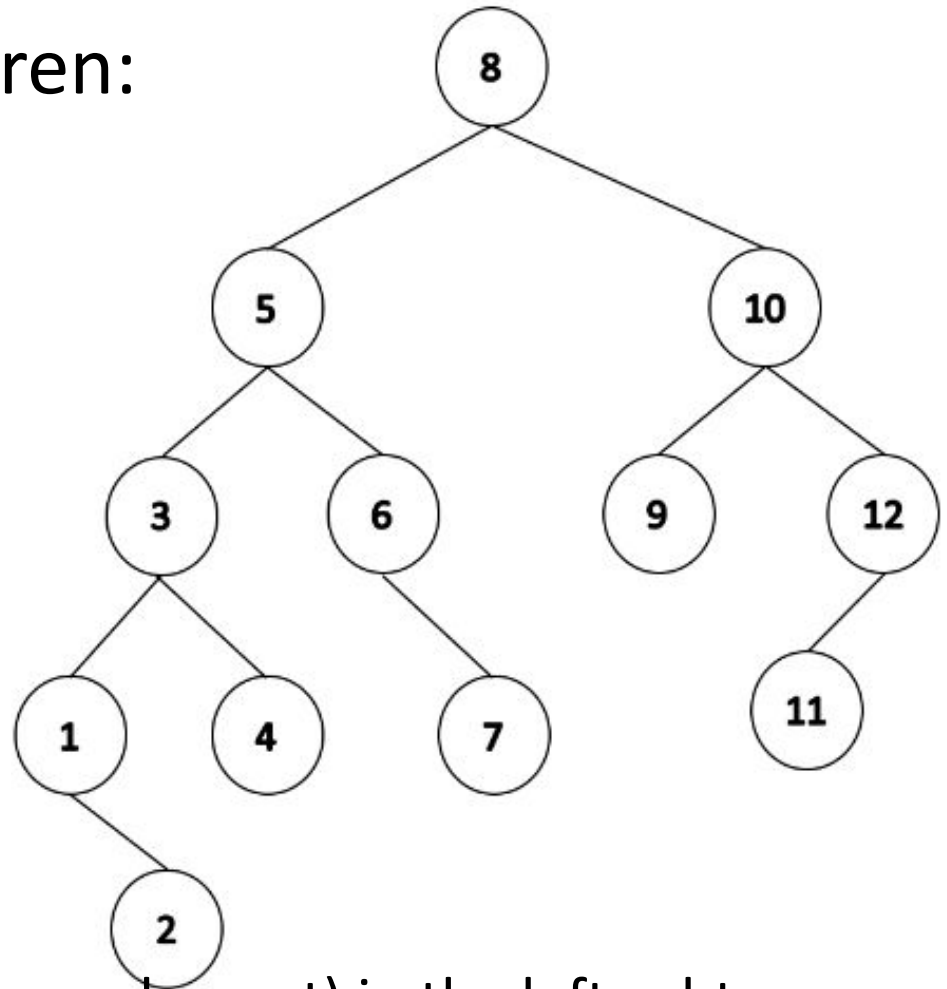
- has two children

A light beige rectangular box with a thin blue border, intended for a diagram of a node with two children.

# Remove with two children:

- Find the element:
  - ✓ Replace it with IOP
  - ✓ Delete the element
  - ✓ Rebalance the tree

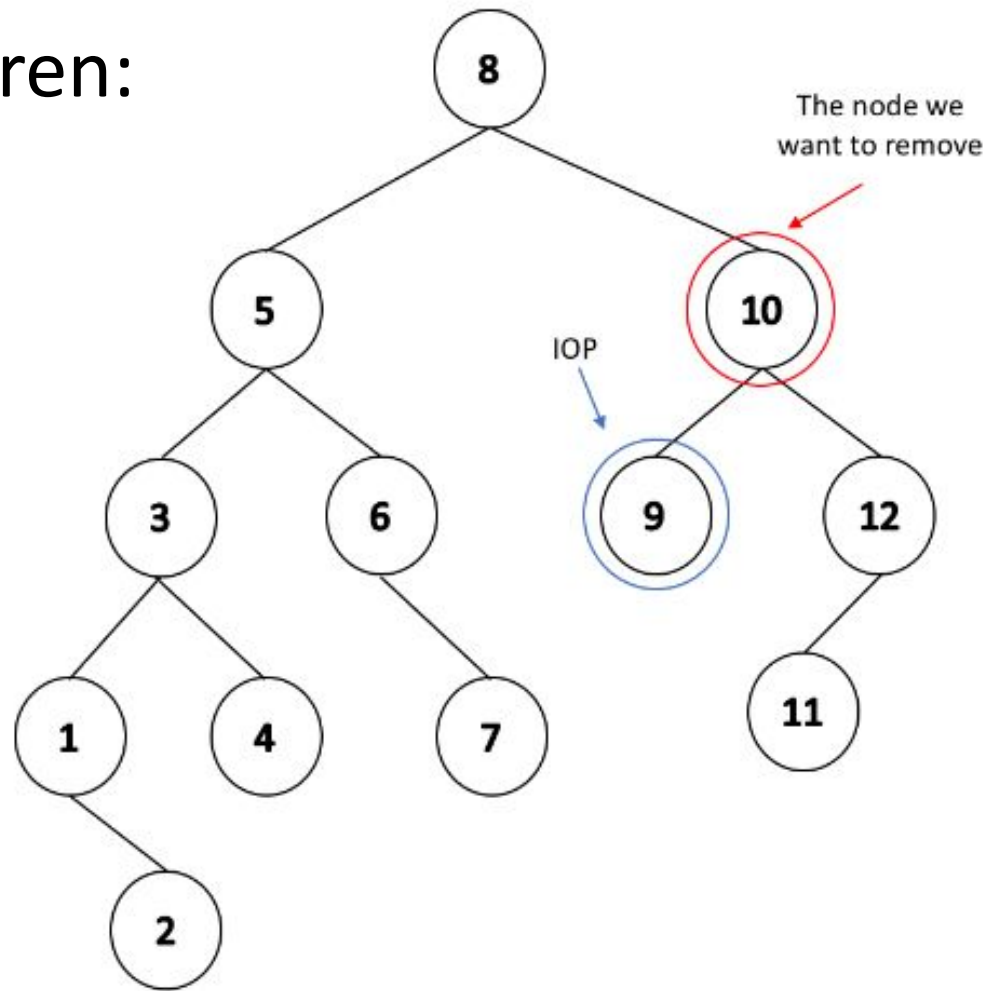
What's IOP for 5? 10? 8?



IOP: Right most element (maximum element) in the left subtree;

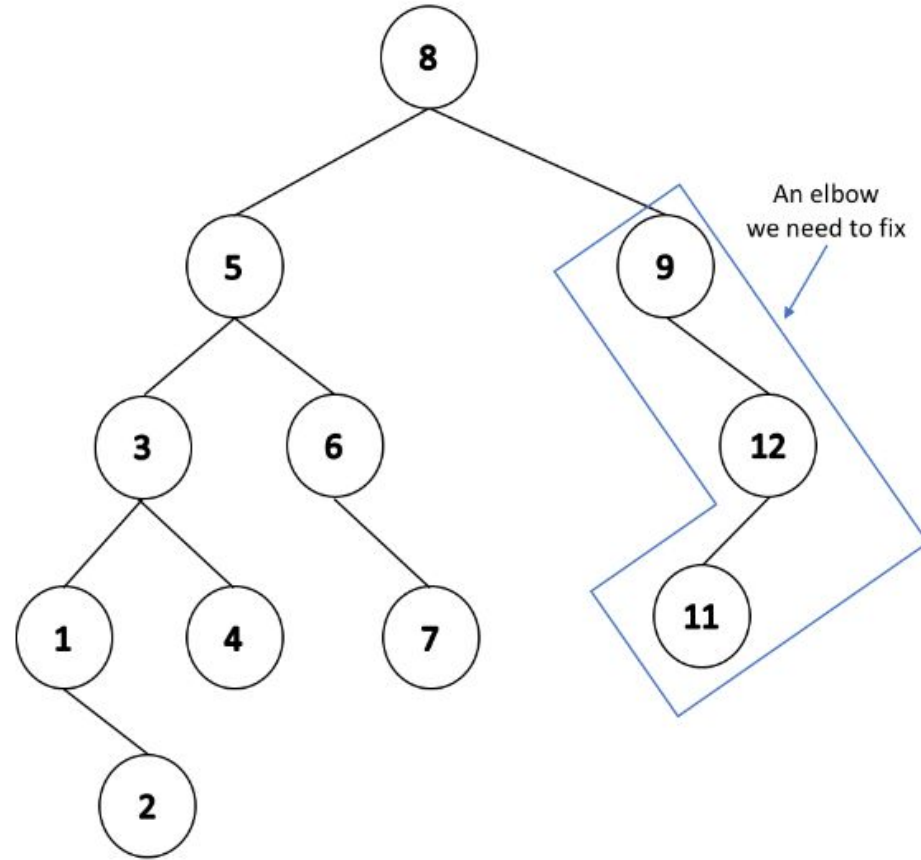
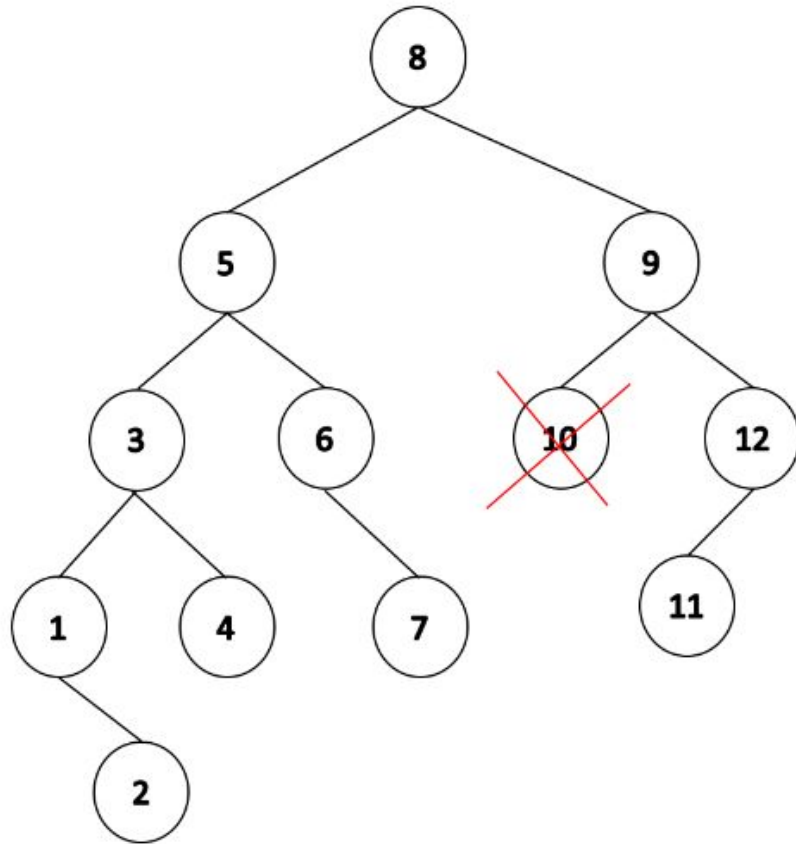
# Remove with two children:

- Find the element:
  - ✓ Replace it with IOP
  - ✓ Delete the element
  - ✓ Rebalance the tree



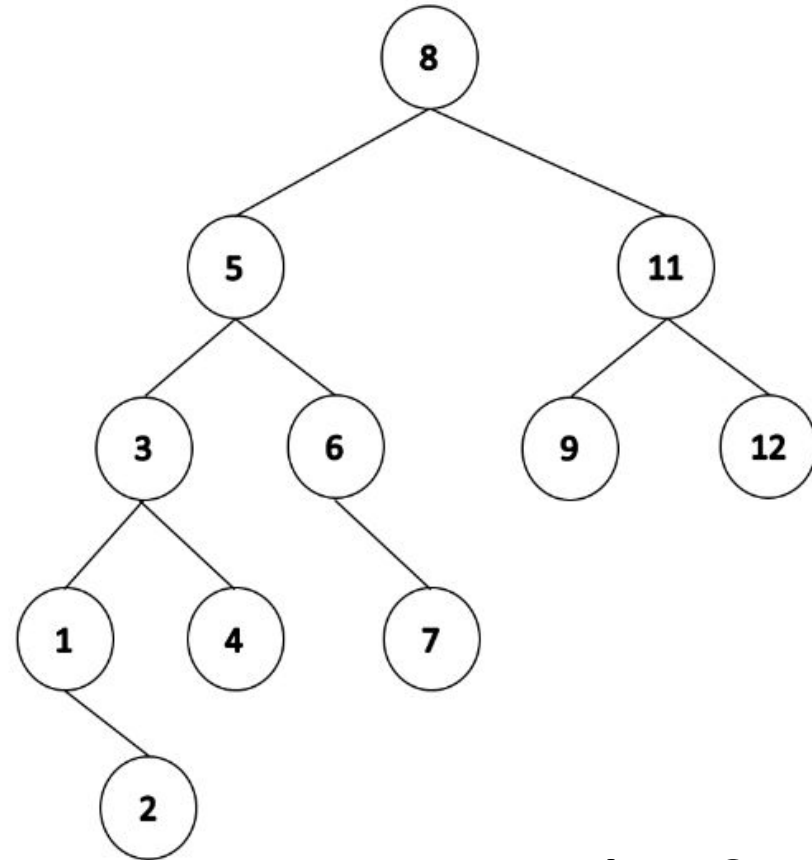
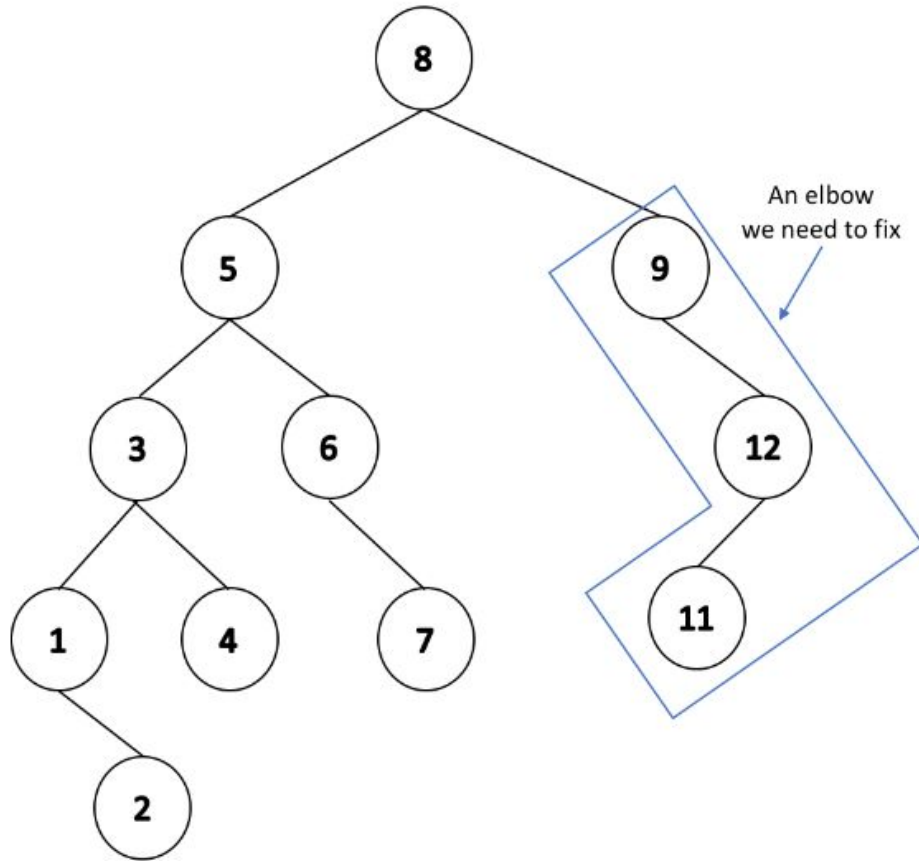
## Rebalance the tree:

Find lowest point of imbalance

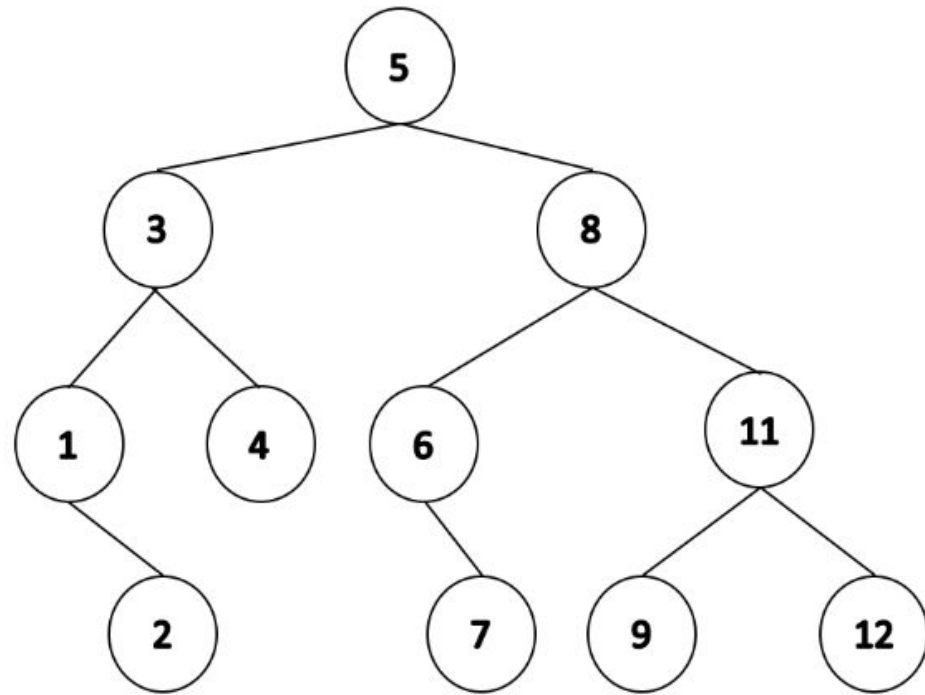
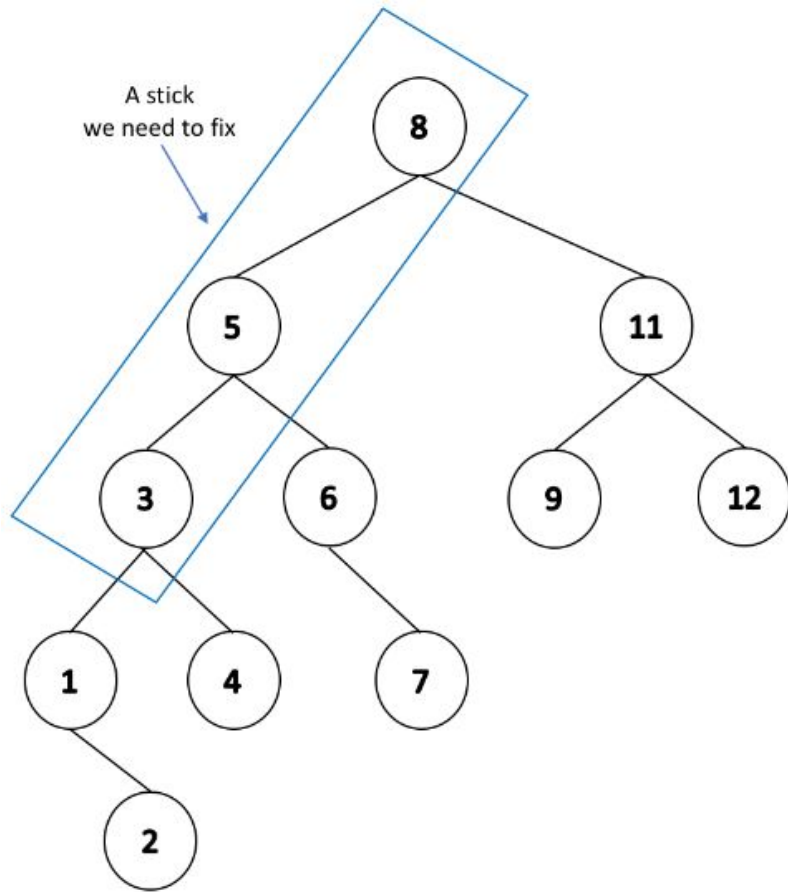


## Rebalance the tree:

Find lowest point of imbalance



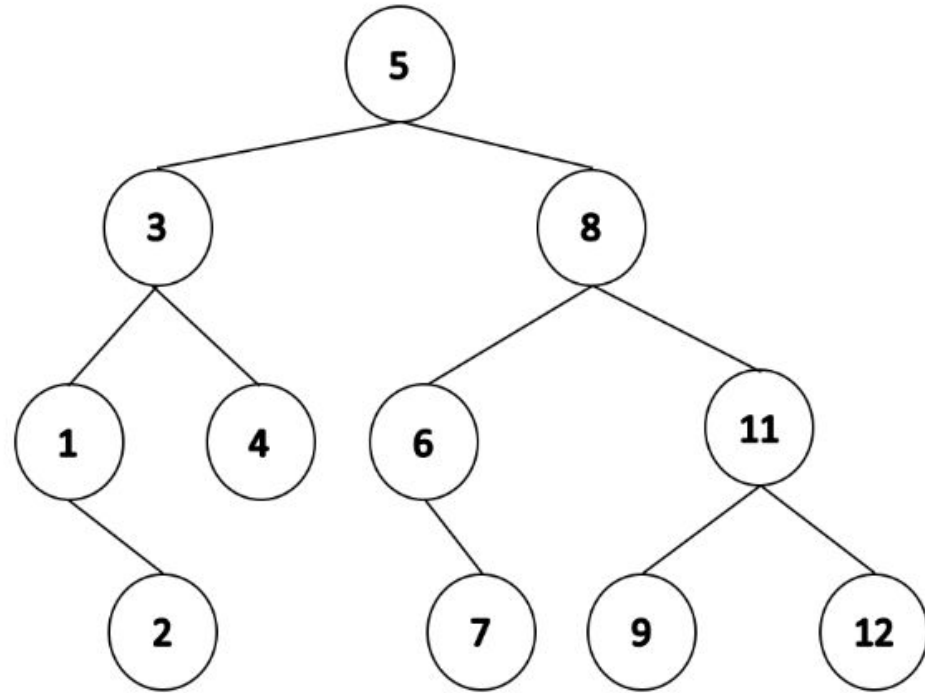
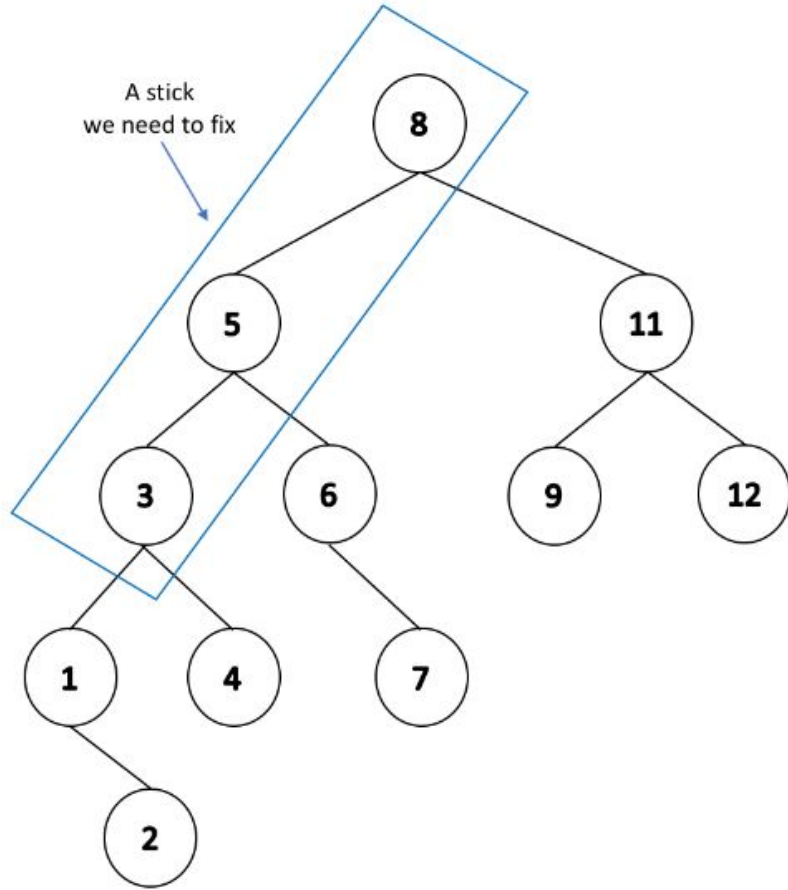
Are we done?



**Are we done?**



A stick  
we need to fix



- $\text{find}(\dots) \rightarrow O(h) + 0 \text{ rotation}$
- $\text{insert}(\dots) \rightarrow O(h) + \text{up to } 1 \text{ rotation}$
- $\text{remove}(\dots) \rightarrow O(h) + \text{up to } h \text{ rotations}$ 
  - Each rotation is  $O(1)$ .
  - Doing  $h$  rotations is  $h * O(1) = O(h)$
  - $O(h) + O(h) = 2 * O(h) = O(h)$

**All operations take  $O(h)$  and  $O(h) = O(\log n)$ !**

Join the Lab AYH Queue if you have questions!



University of San Francisco

<https://www.cs.usfca.edu> > ~galles > visualization > AVLt... ⋮

## AVL Tree Visualization

AVL Tree. Animation Speed. w: h: Algorithm Visualizations.