

Data Structures

AVL Analysis

Analysis probably Monday....

CS 225

Brad Solomon

3

October 2, 2026



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science

Learning Objectives

Review AVL trees

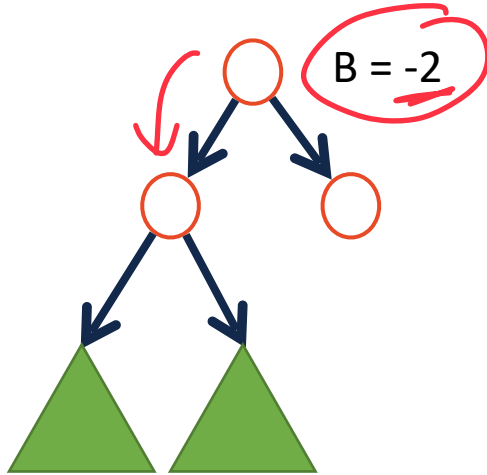
← *Implementation*

Prove that the AVL Tree speeds up all operations

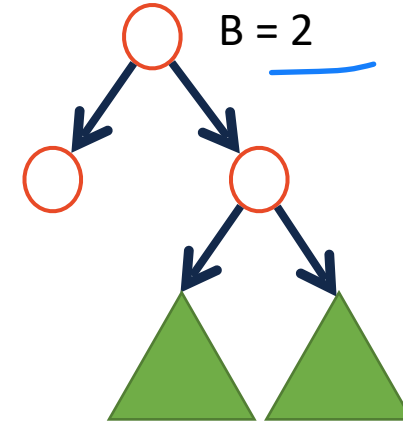
Left Imbalance

If signs match single rotation

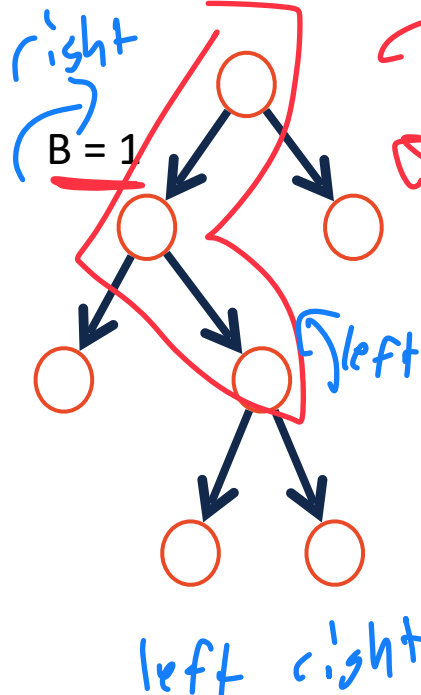
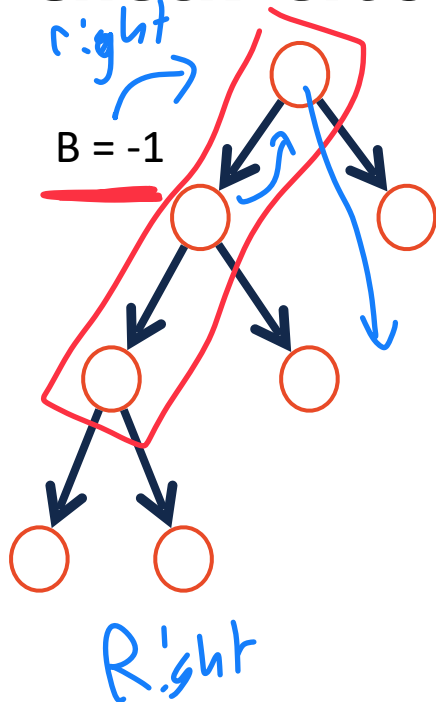
Right Imbalance



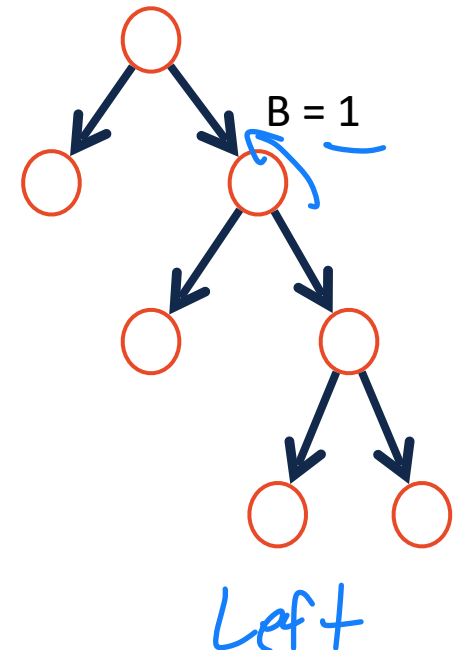
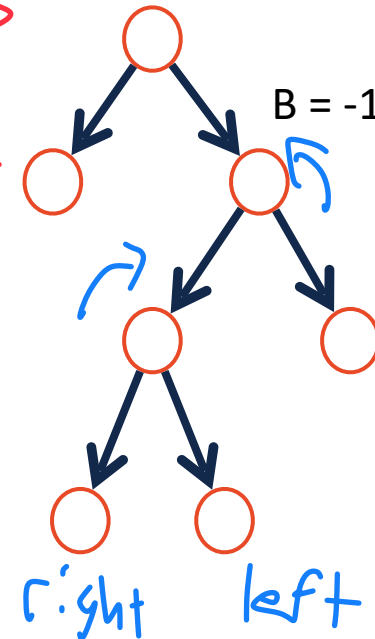
If signs differ double rotate



Check left subtree:

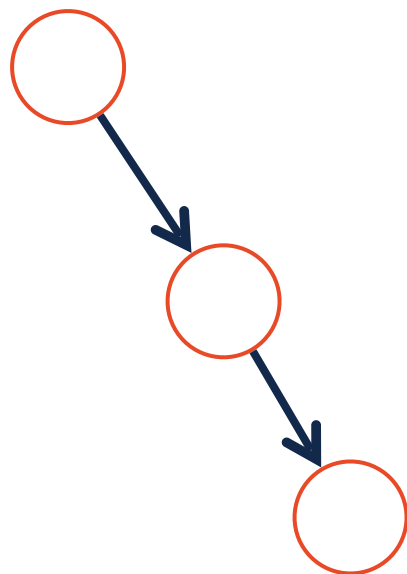


Check right subtree:



AVL Rotations

Left

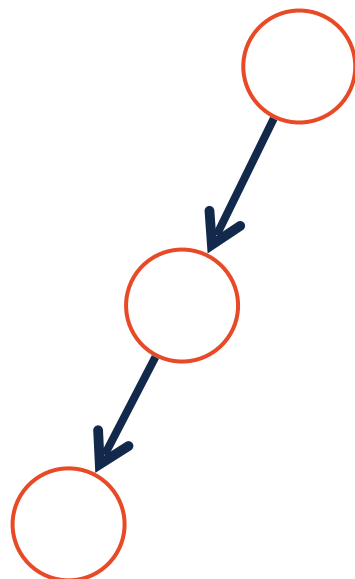


Root Balance: 2

Child Balance: 1

single

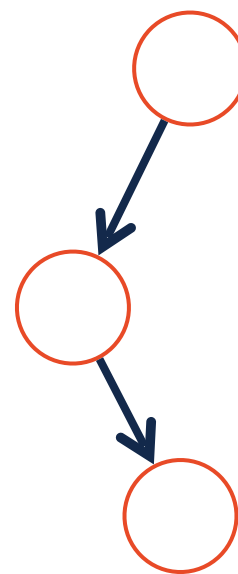
Right



-2

-1

LeftRight

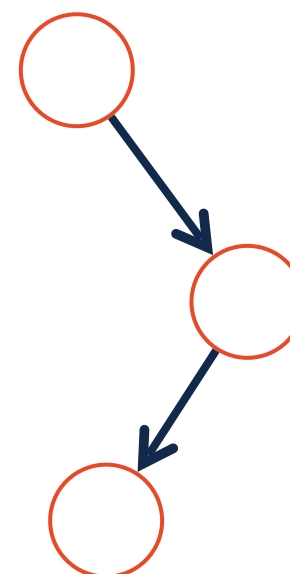


-2

1

double

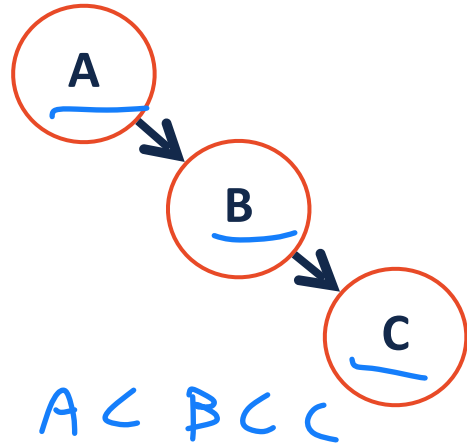
RightLeft



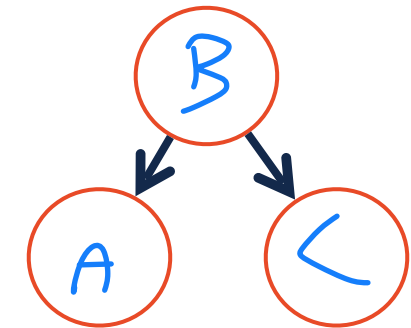
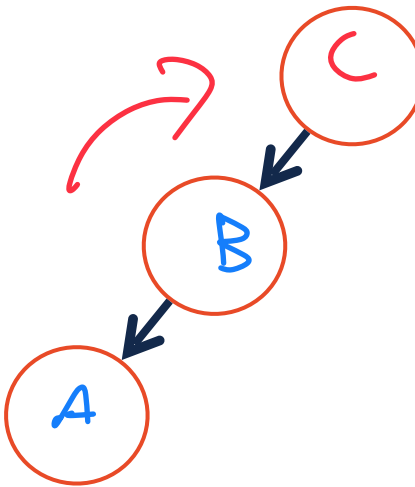
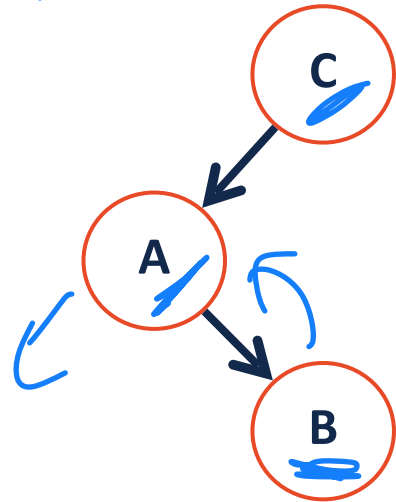
2

-1

AVL Tree Rotations



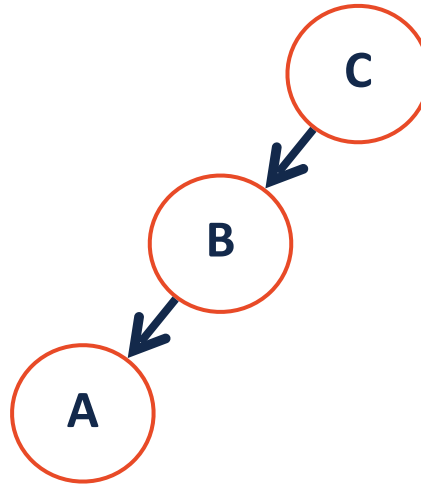
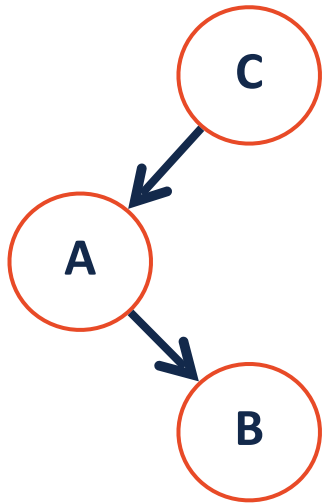
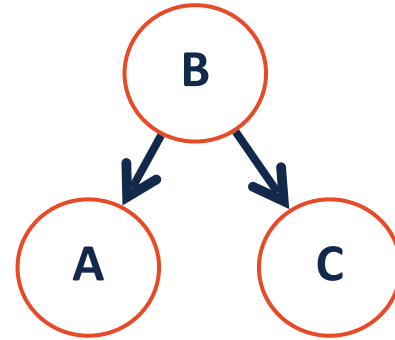
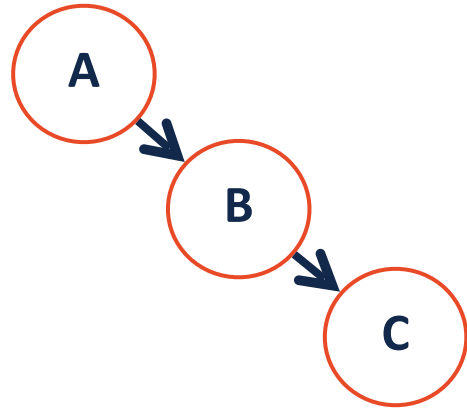
$A < B < C$



All rotations are $O(1)$

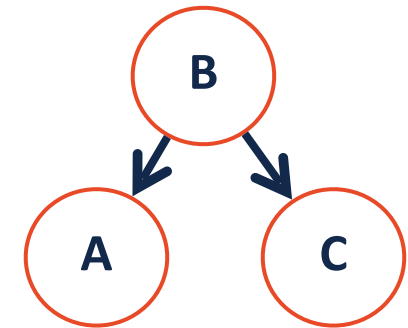
All rotations reduce subtree height by one

AVL Tree Rotations



All rotations are $O(1)$

All rotations reduce subtree height by one



AVL vs BST ADT



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

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

One change to `TreeNode`
Must know own height

Tradeoff $O(1)$ balance calcs

$O(n)$ balance

AS soon as imbalance happens

How does the constraint on balance affect the core functions?

Find

Insert

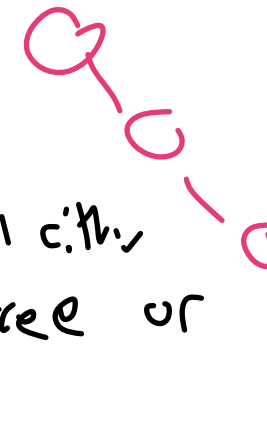
Remove

How to fix a long straight top?

i) AVL can't do this

In general c.t.h. rebuild tree or

rotate repeatedly



AVL Find

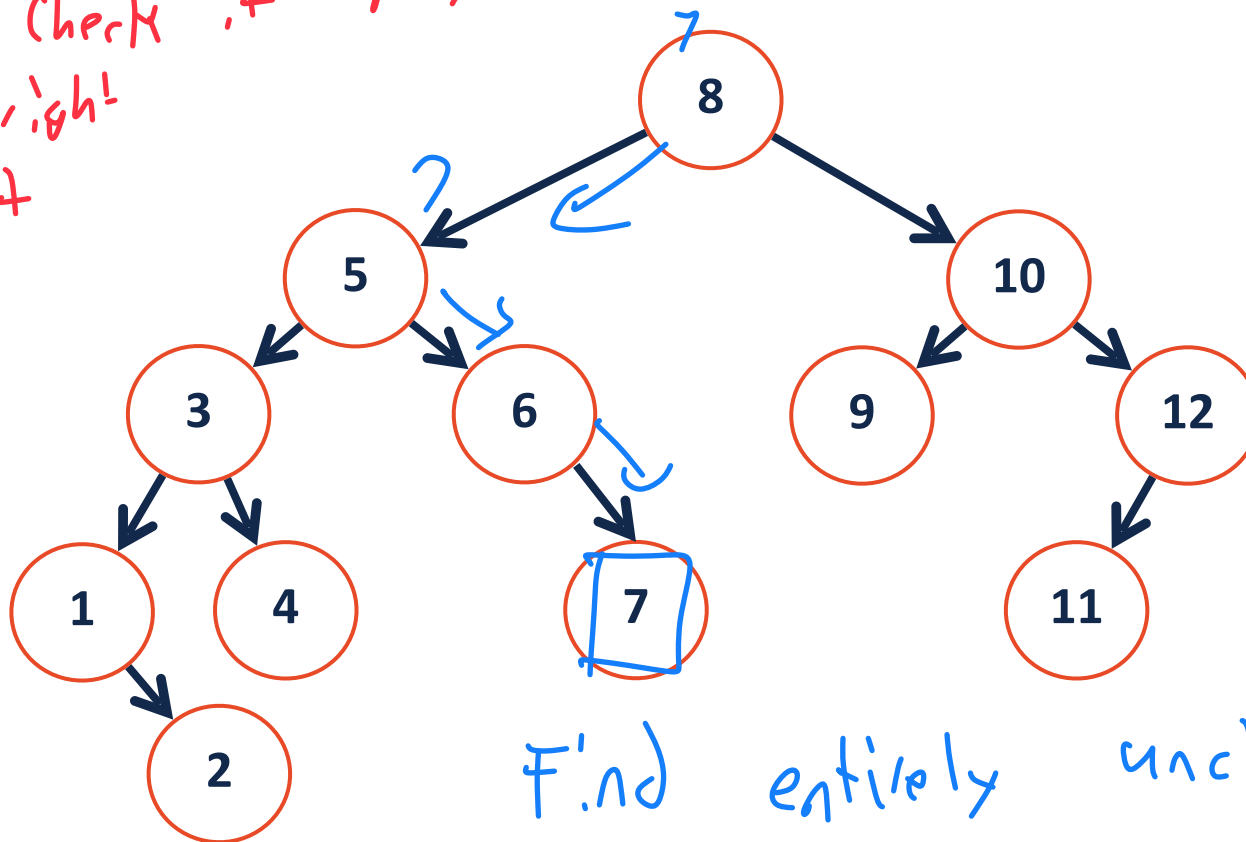
Base case: root is null or query is root
↳ return root;

`_find(7)`

Start from root

recursive step: check if query > root

↳ if yes, go right
else go left



$$h = O(\log n)$$

To be proven

Find entirely unchanged!

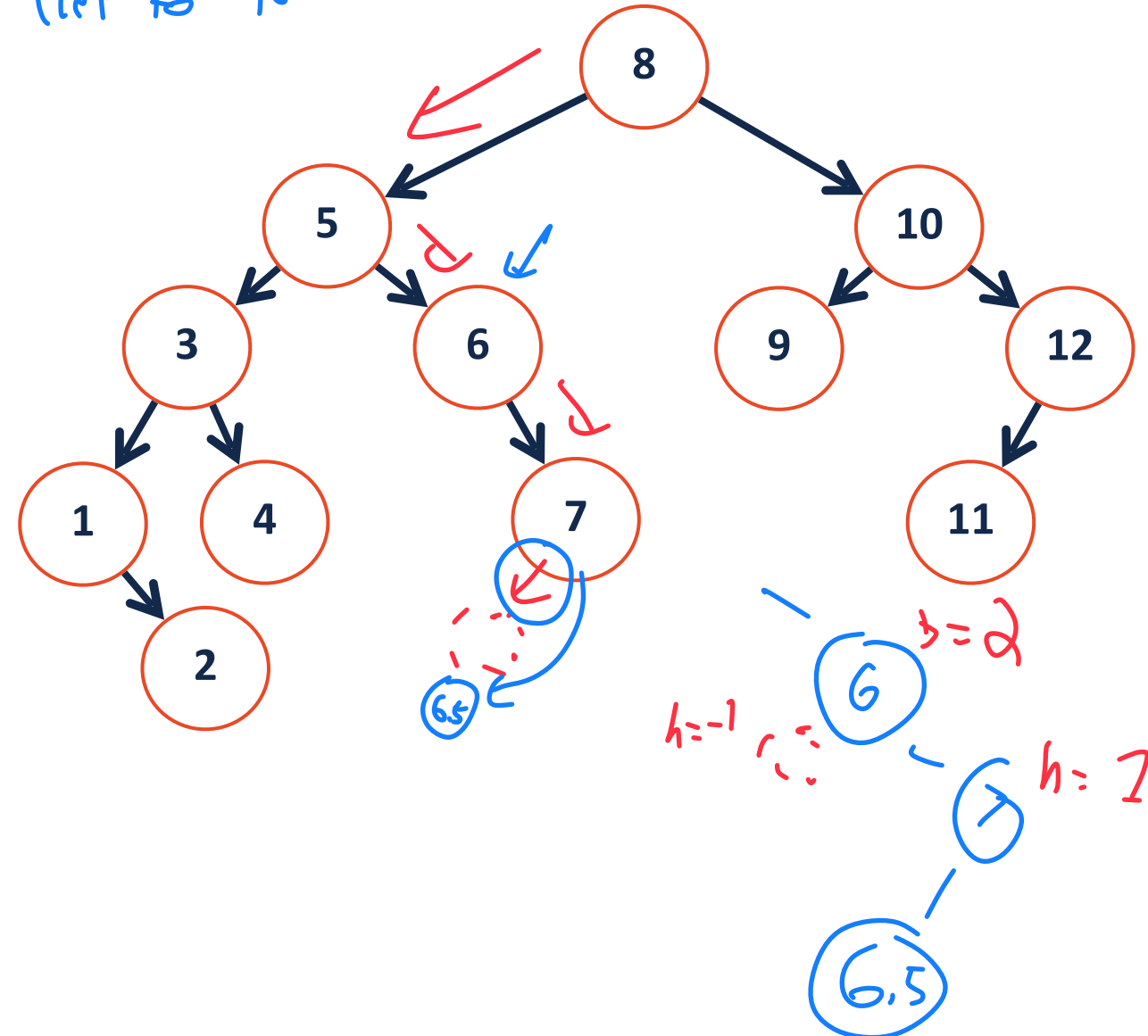
↳ Better runtime though!

$$O(h)$$

AVL Insertion

`_insert(6.5)`

- 1) Find insert location ↴
- 2) Add new Node to output of find (ref to pointer)



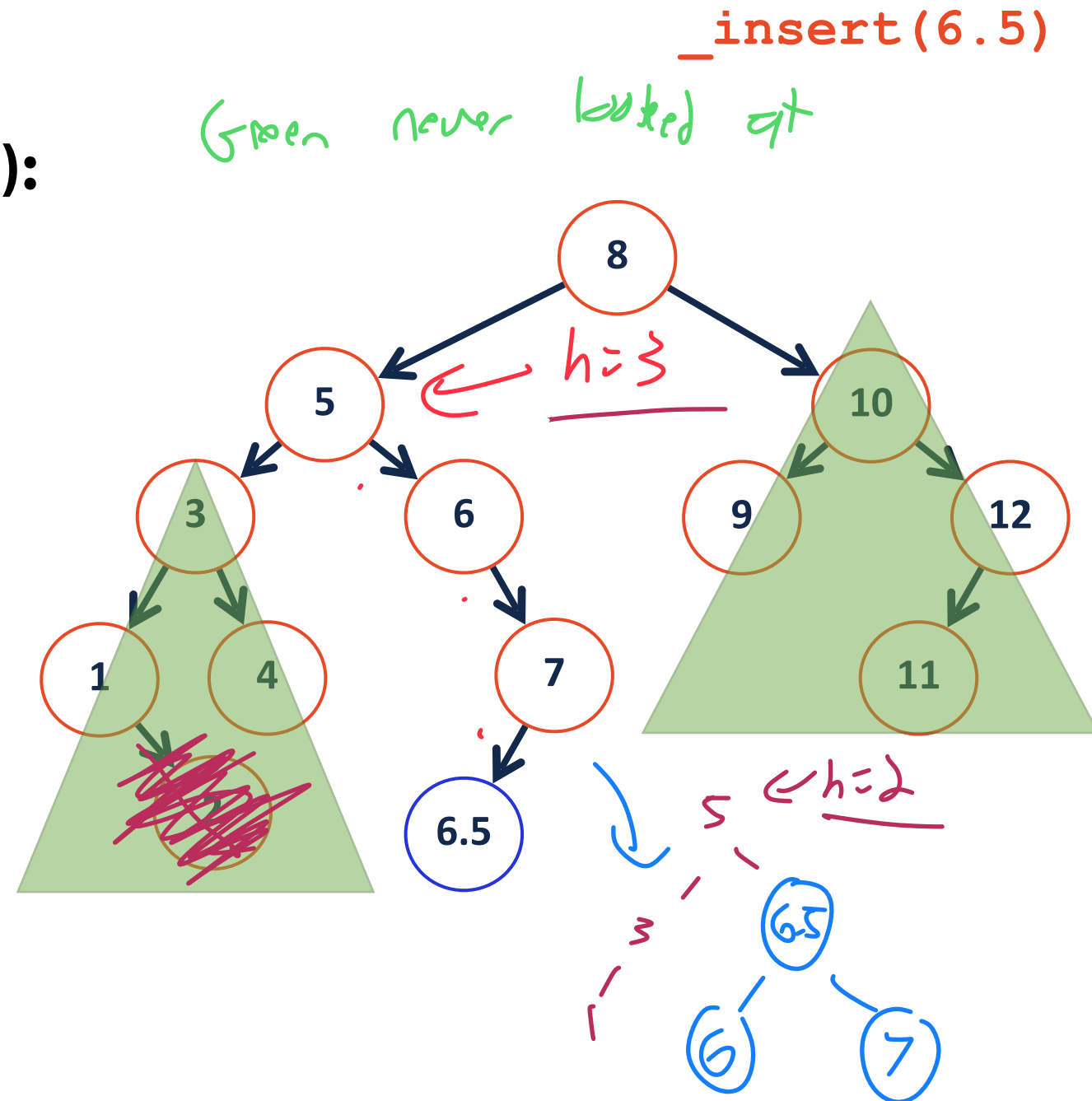
```
1 struct TreeNode {  
2     T key;  
3     unsigned height;  
4     TreeNode *left;  
5     TreeNode *right;  
6 };
```

AVL Insertion

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 };
```




```

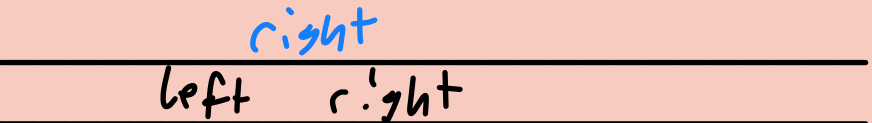
119 template <typename K, typename V>
120 void AVL<K, D>::_ensureBalance(TreeNode *& cur) {
121     // Calculate the balance factor:
122     int balance = height(cur->right) - height(cur->left);
123
124     // Check if the node is current not in balance:
125     if ( balance == -2 ) {
126         int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

```

↖ check new height

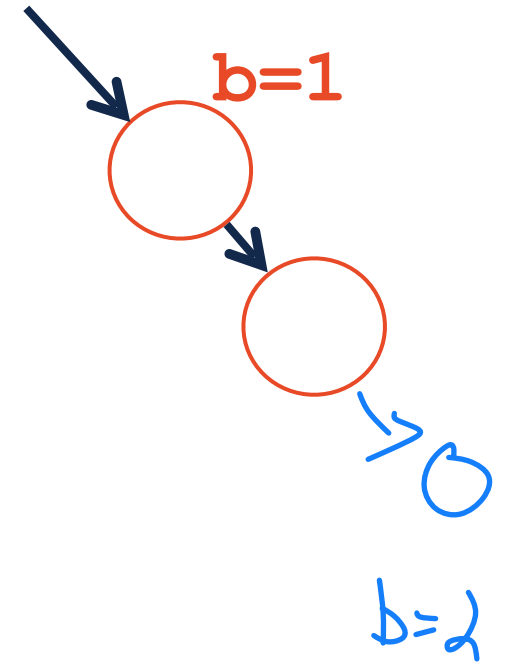
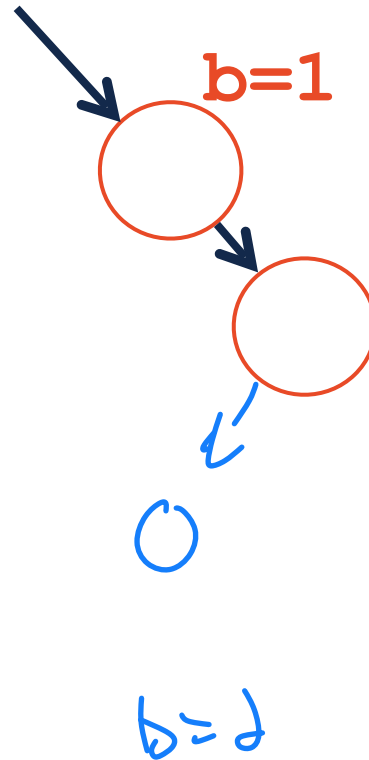
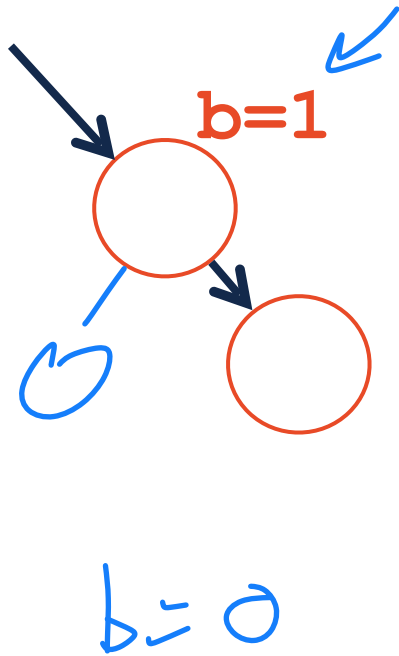
This work
Only happens
if NOT
balanced

O(1)



AVL Insertion

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



✓

AVL Insertion



An AVL tree can never have a balance $|b| > 2$

Insert may increase height by at most:

1

A rotation reduces the height of the subtree by:

1

A single* rotation restores balance and corrects height!

To do the rotation only

What is the Big O of performing our rotation?

$O(1)$

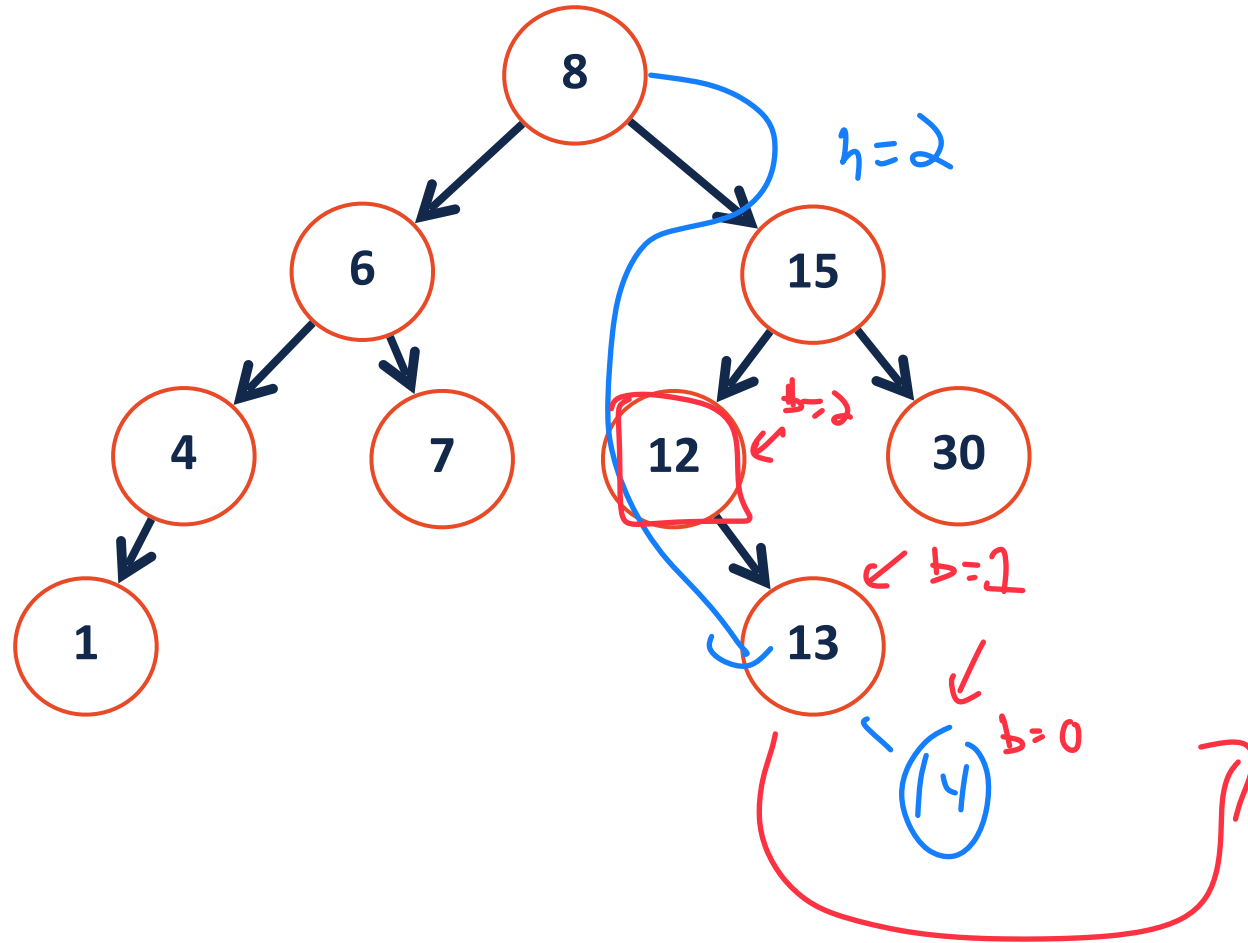
What is the Big O of insert?

$O(h)$

find() is $O(h)$
insert point is $O(h)$
update height() on way back $O(h)$

AVL Insertion Practice

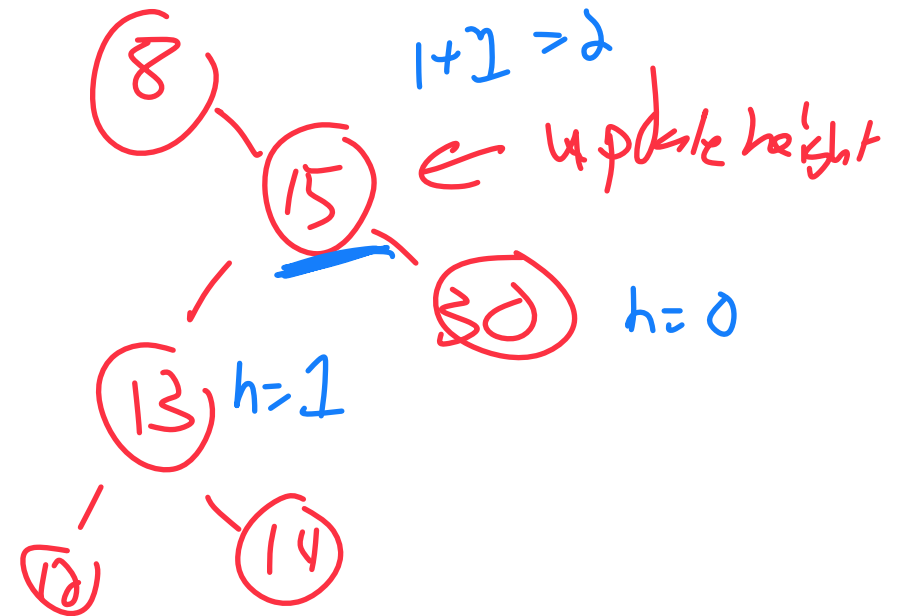
Every node knows own height!



insert(14)

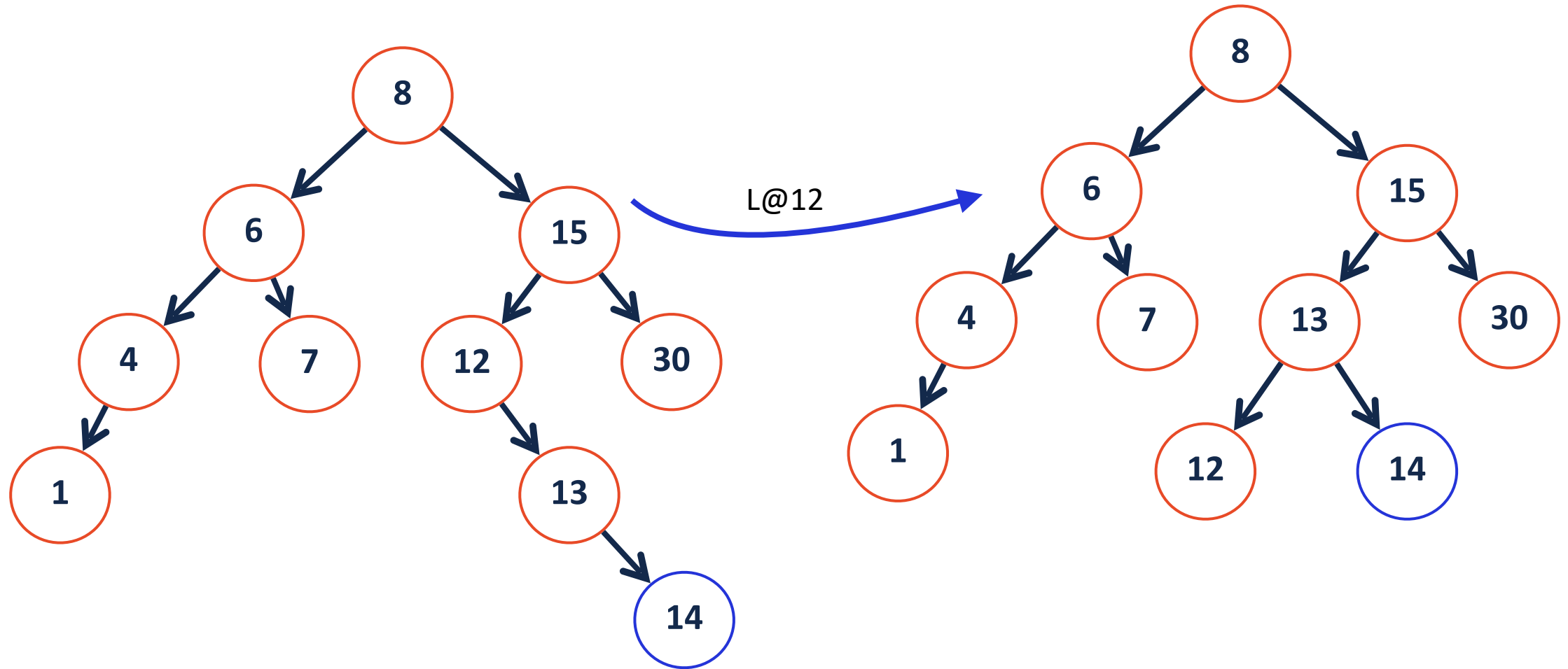
1) Insert

2) Rotate if necessary

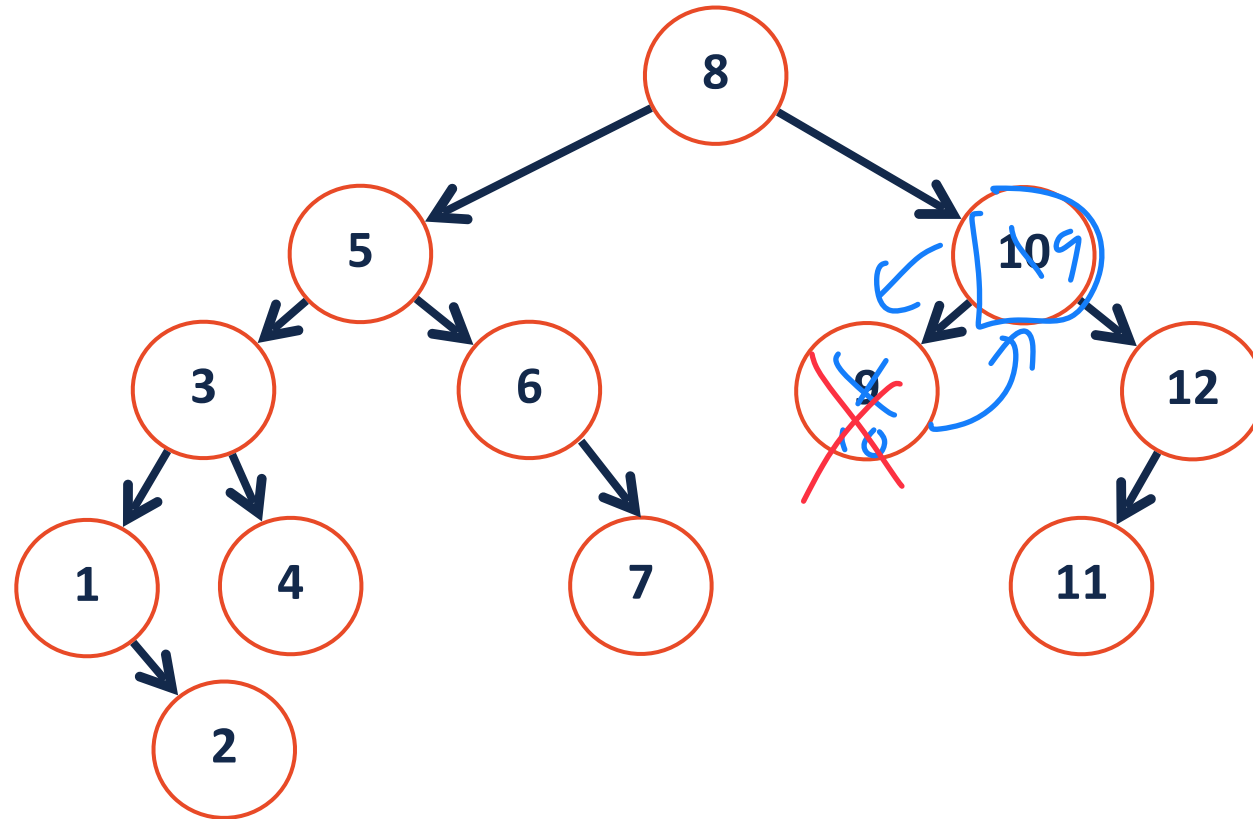


AVL Insertion Practice

`_insert(14)`

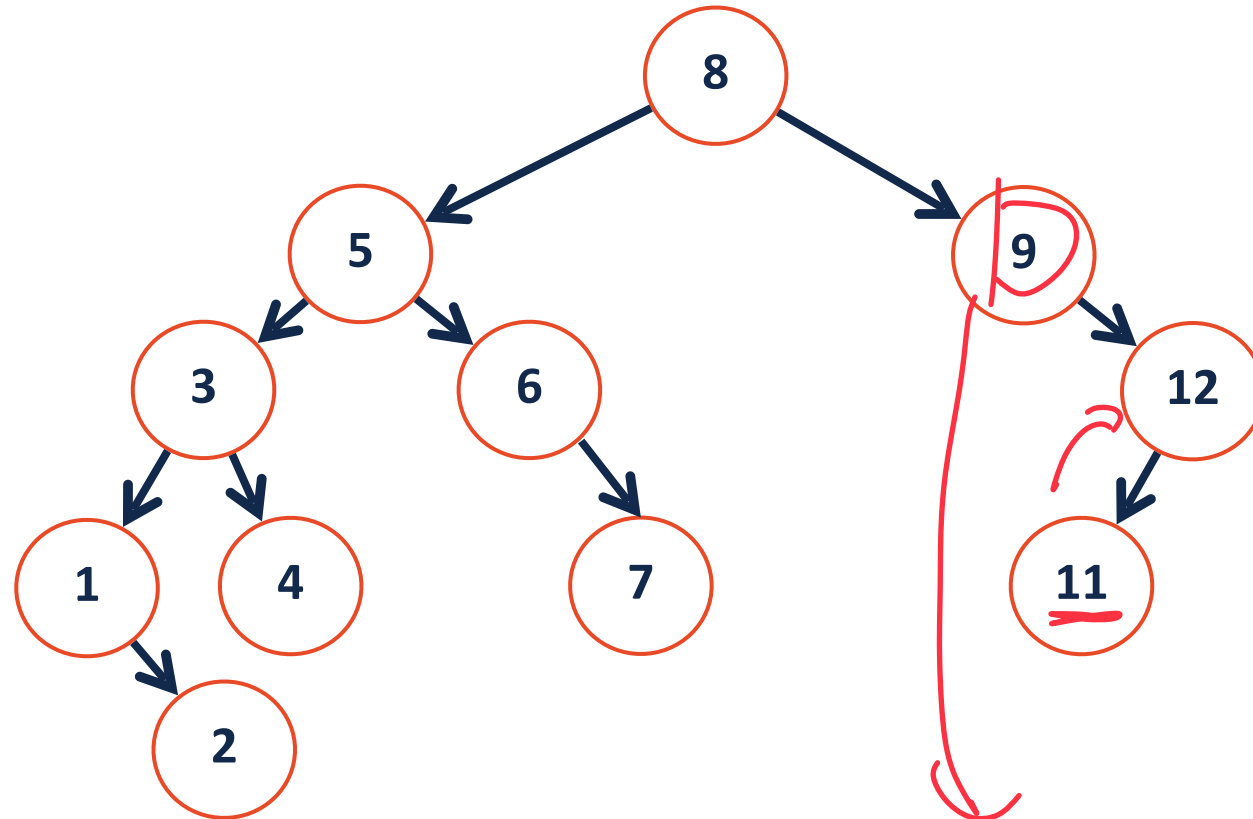


AVL Remove

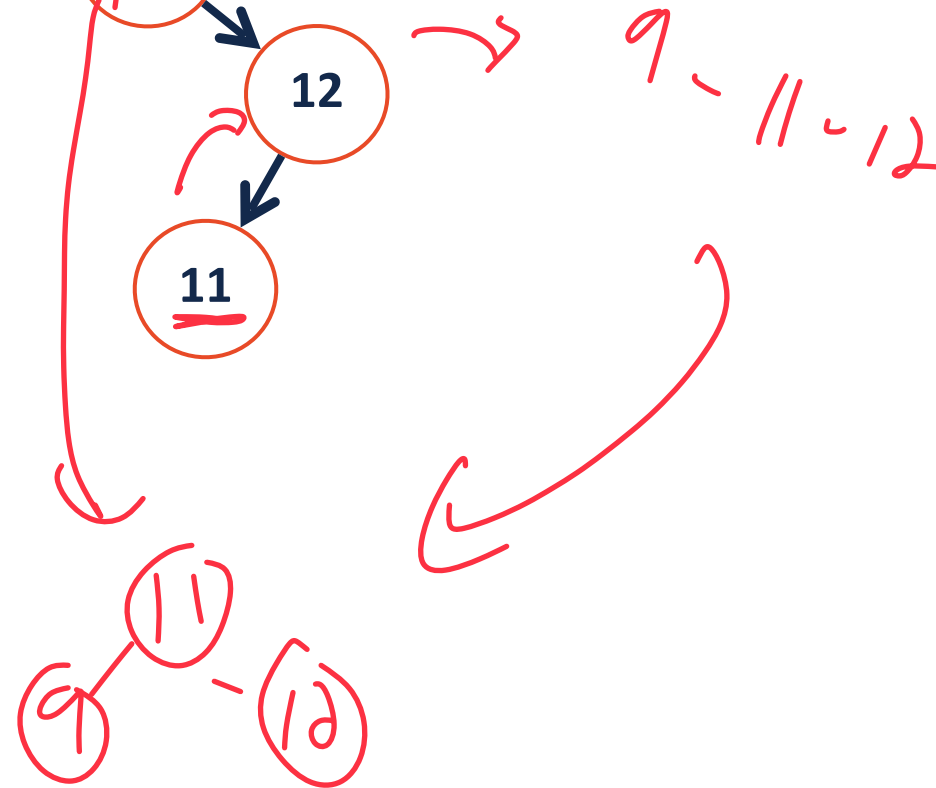


- _remove(10)**
- 1) Find Node to be removed
 - 2) Remove it
↳ child case
↳ IOP & swap
↳ left remove

AVL Remove

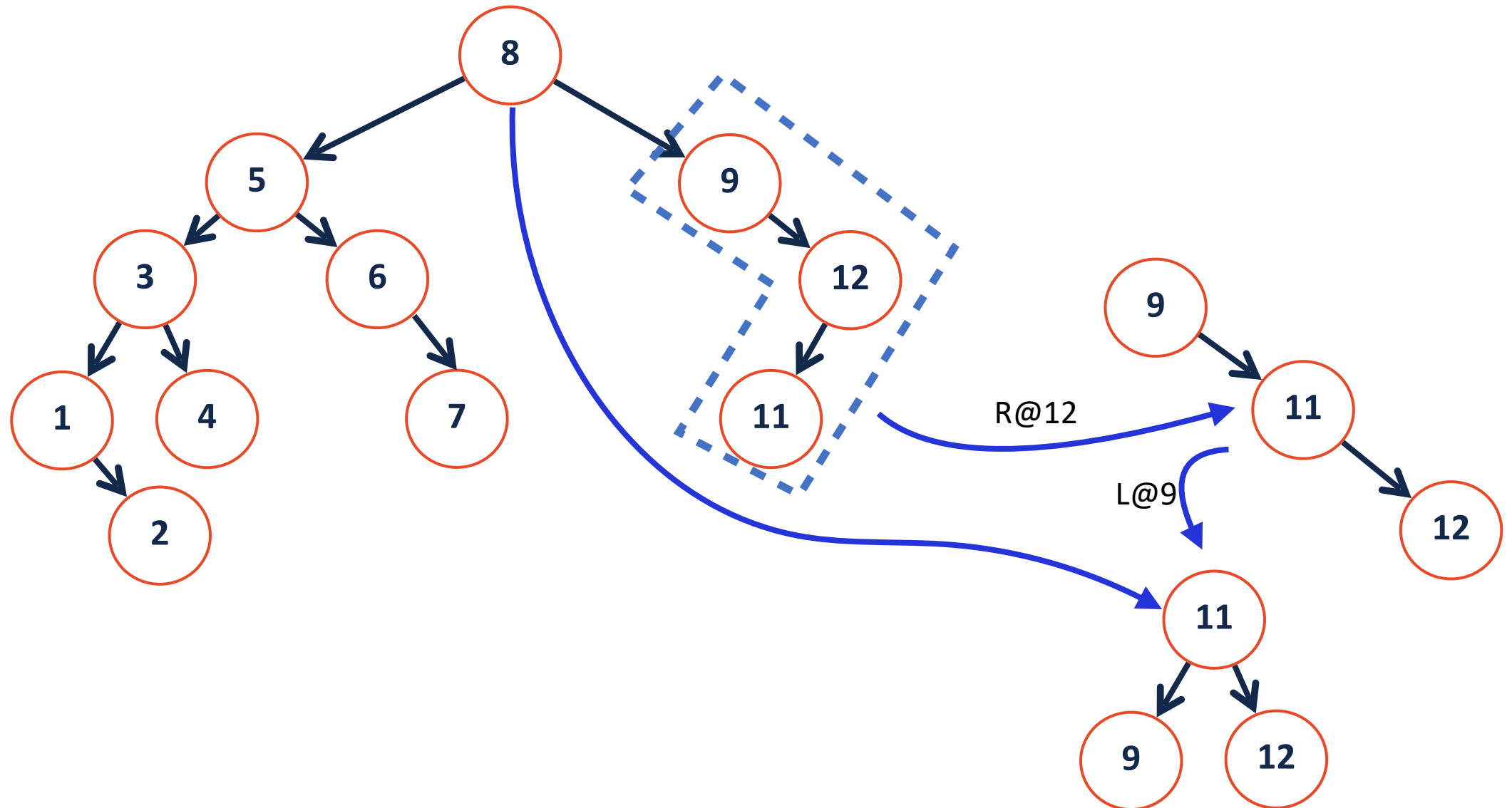


- remove(10)**
- 1) Do BST remove
 - 2) check for imbalance



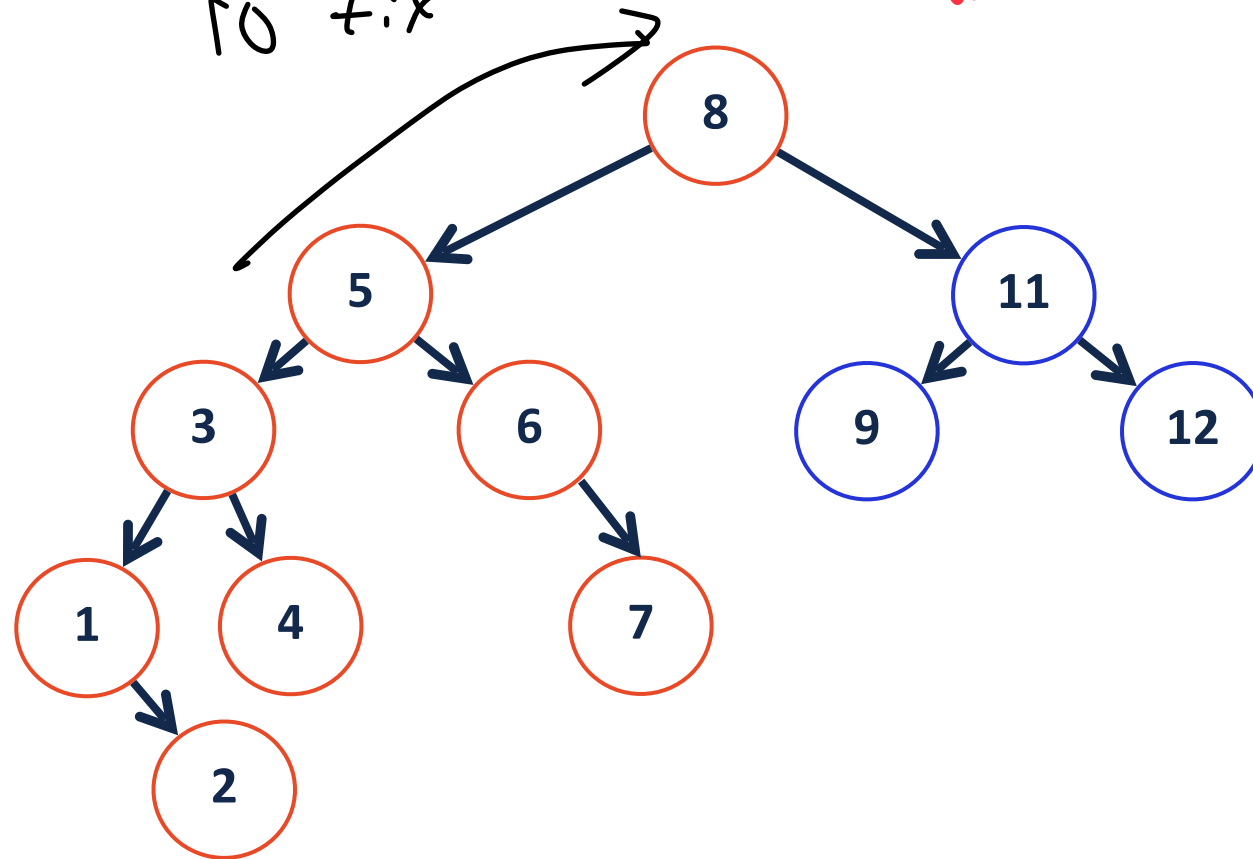
AVL Remove

`_remove(10)`



AVL Remove

to fix



uh oh....

8

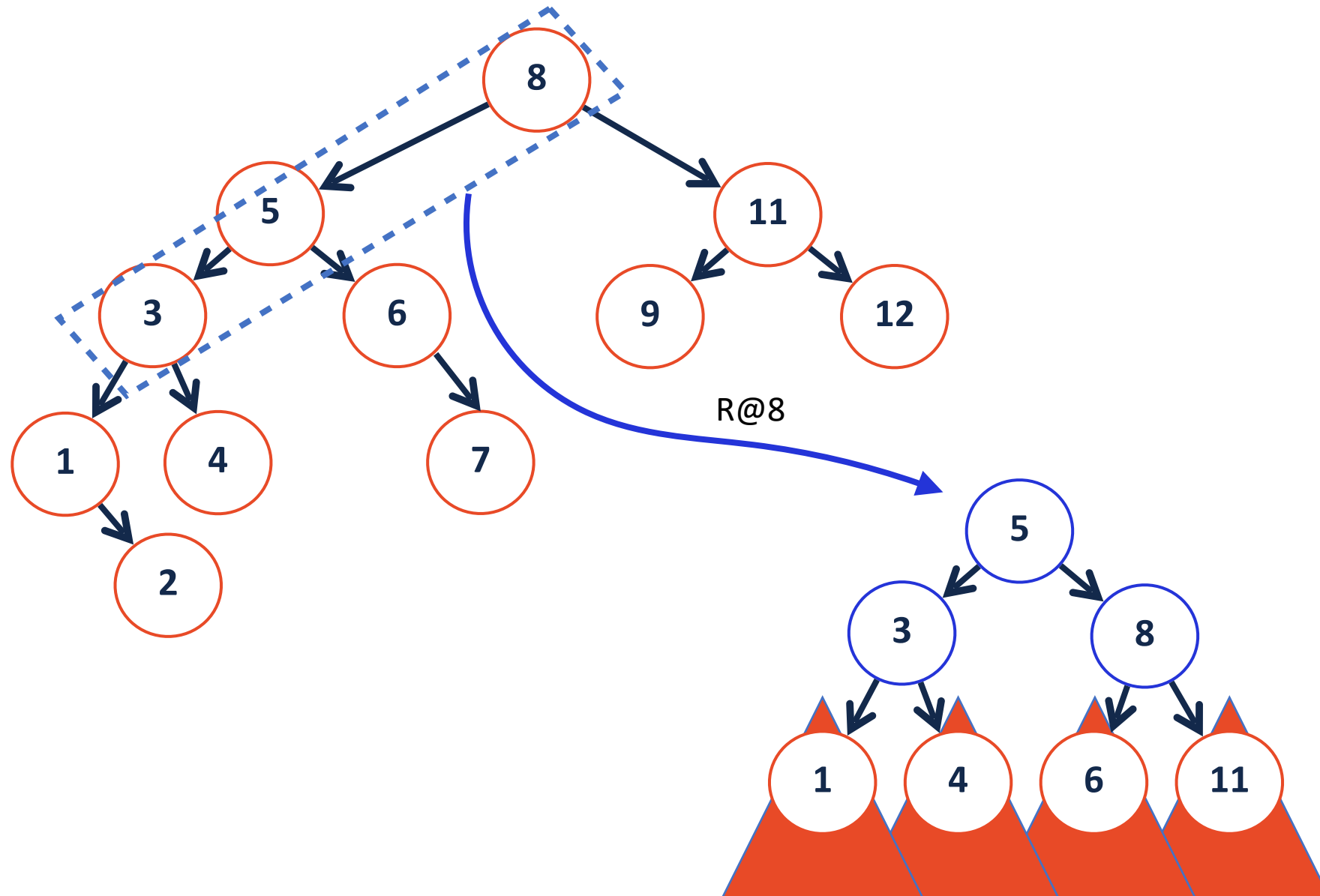
not

balanced!

`_remove(10)`

AVL Remove

`_remove(10)`

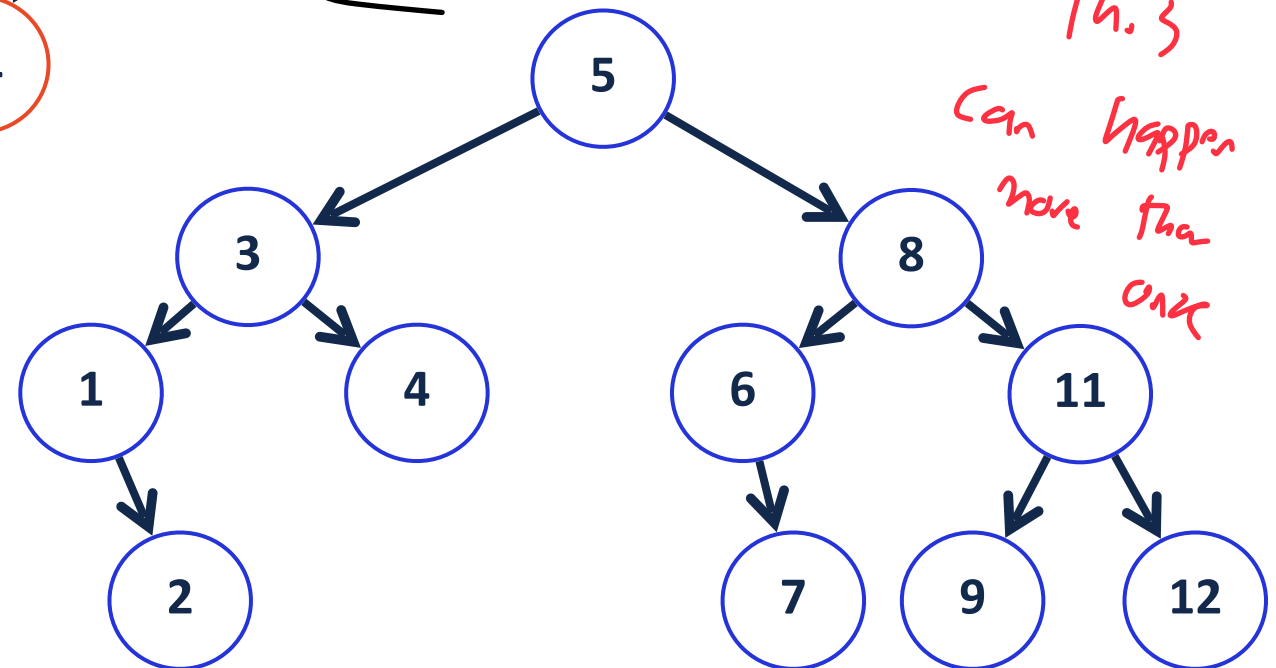
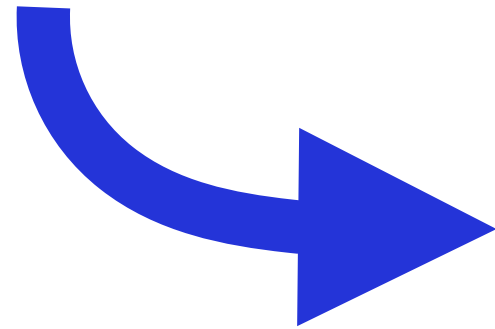
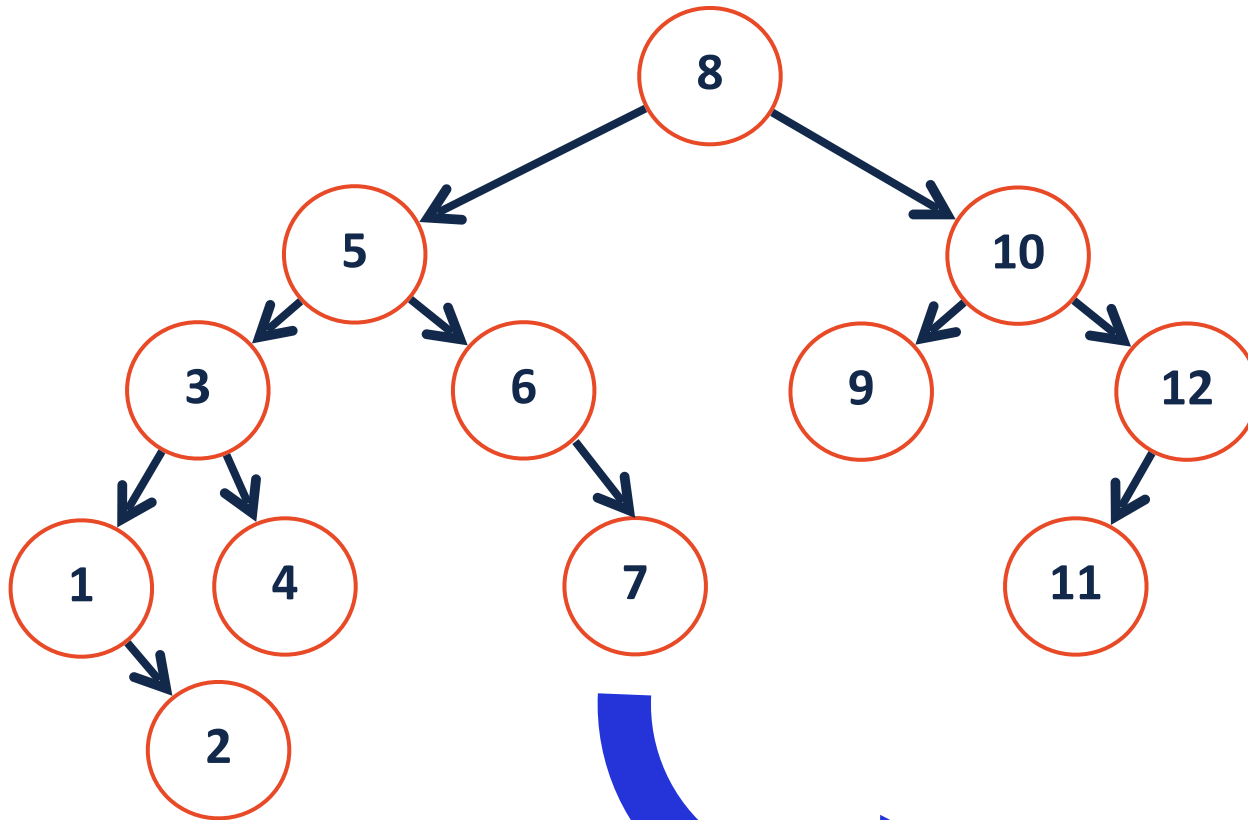


AVL Remove

`_remove(10)` 

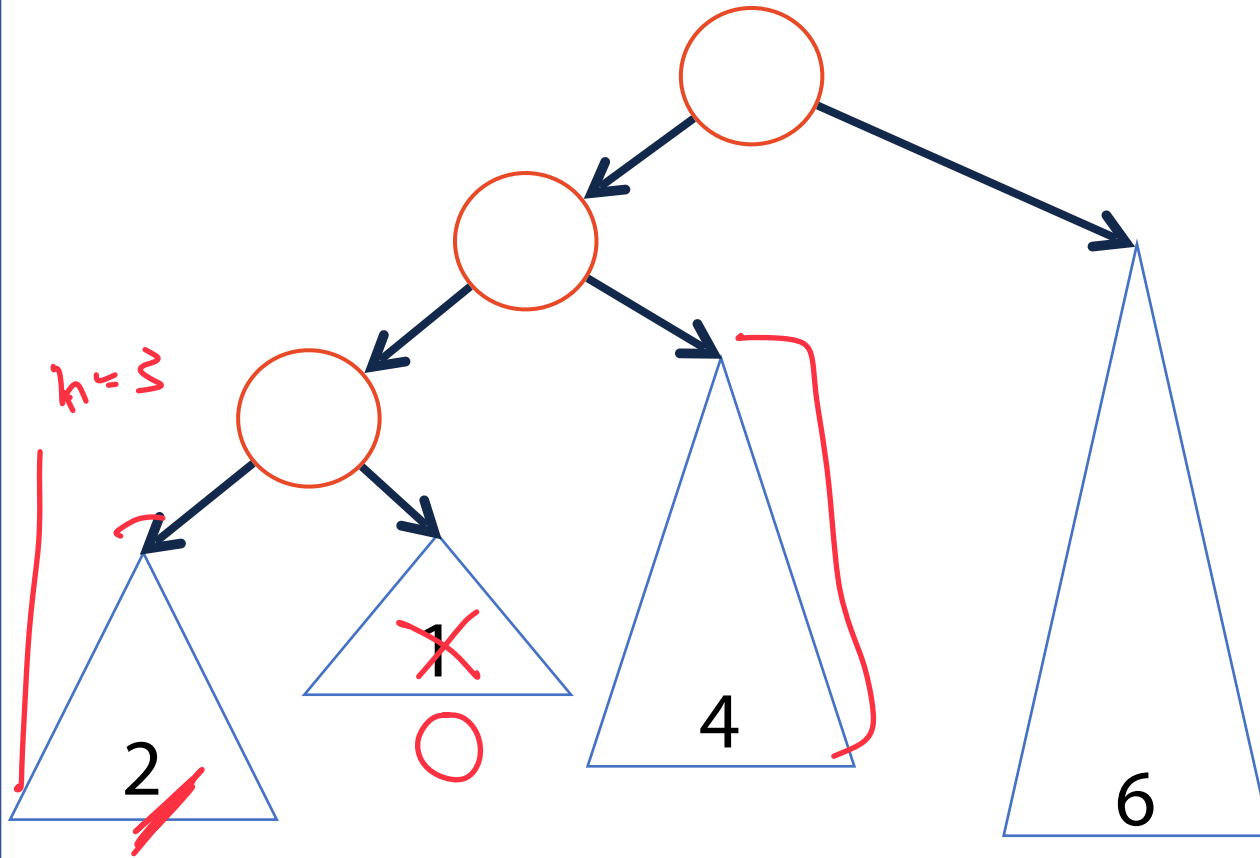
Remove (pseudo code):

- 1: Remove at proper place
- 2: Check for imbalance
- 3: Rotate, if necessary
- 4: Update height



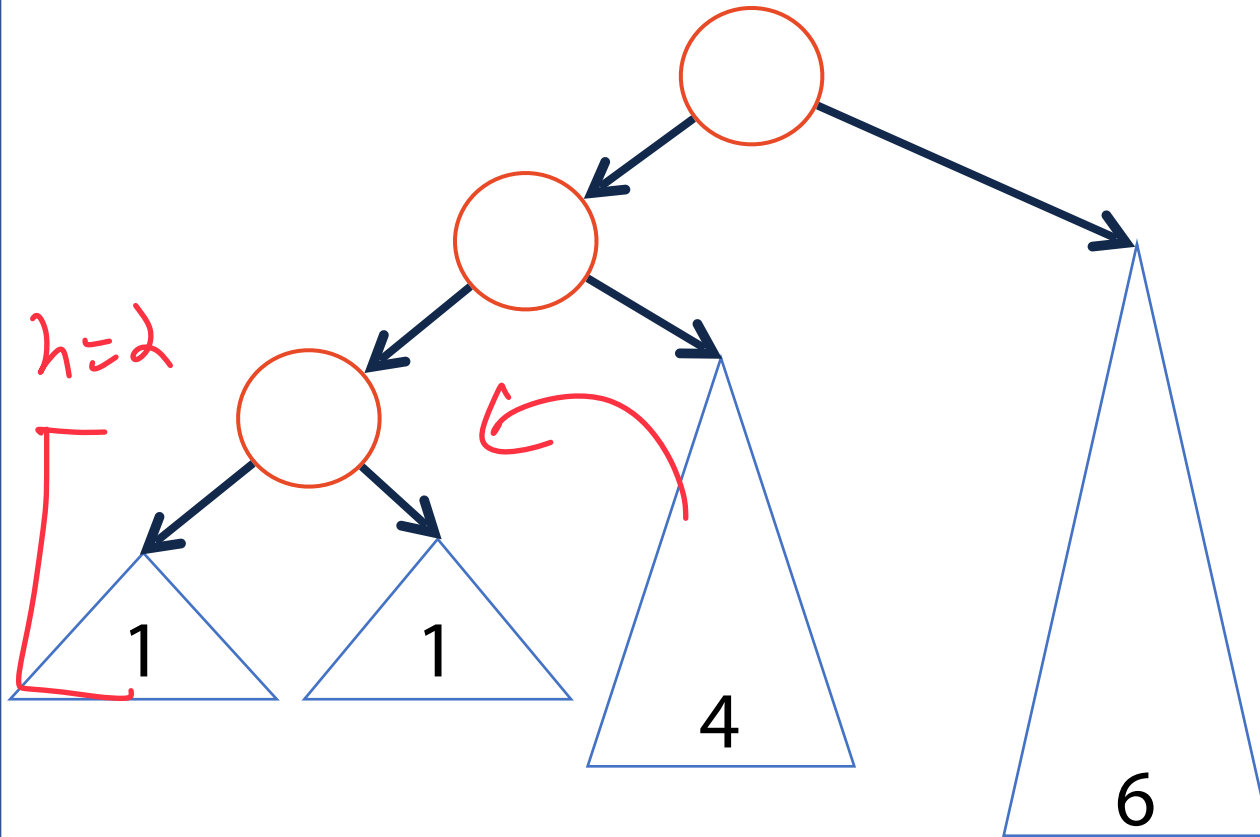
AVL Remove

Remove can cause an imbalance at every level



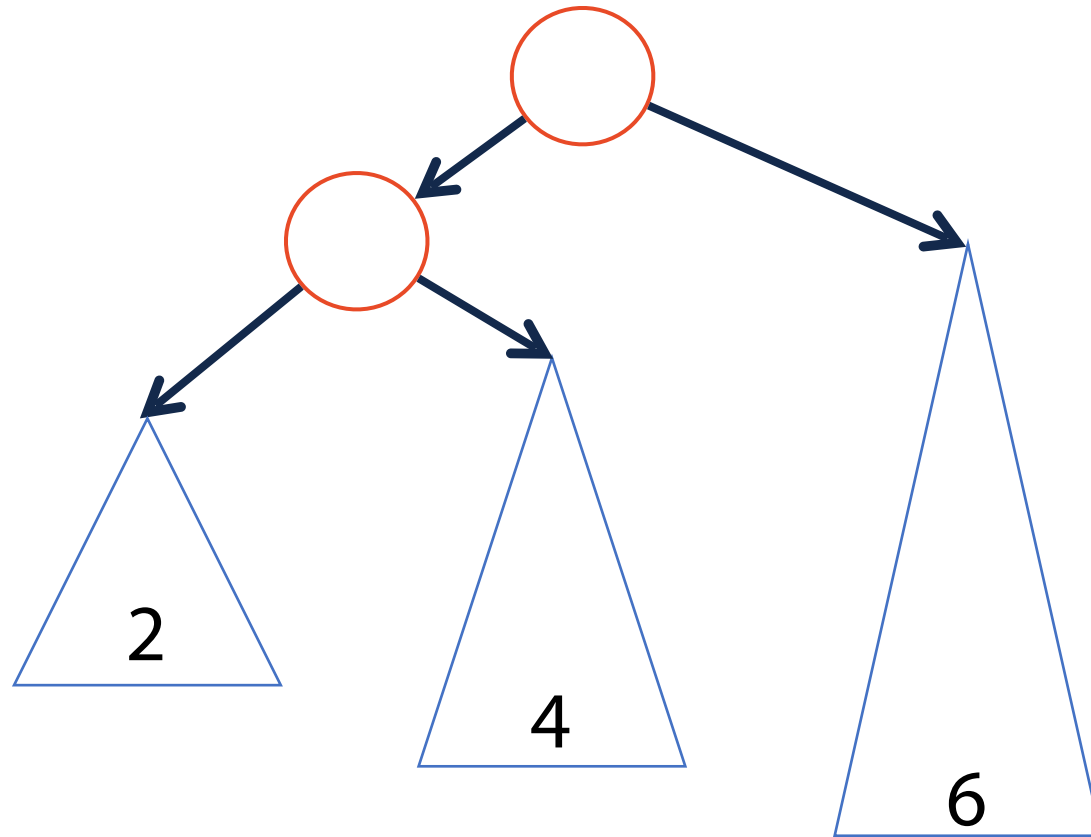
AVL Remove

Remove can cause an imbalance at every level



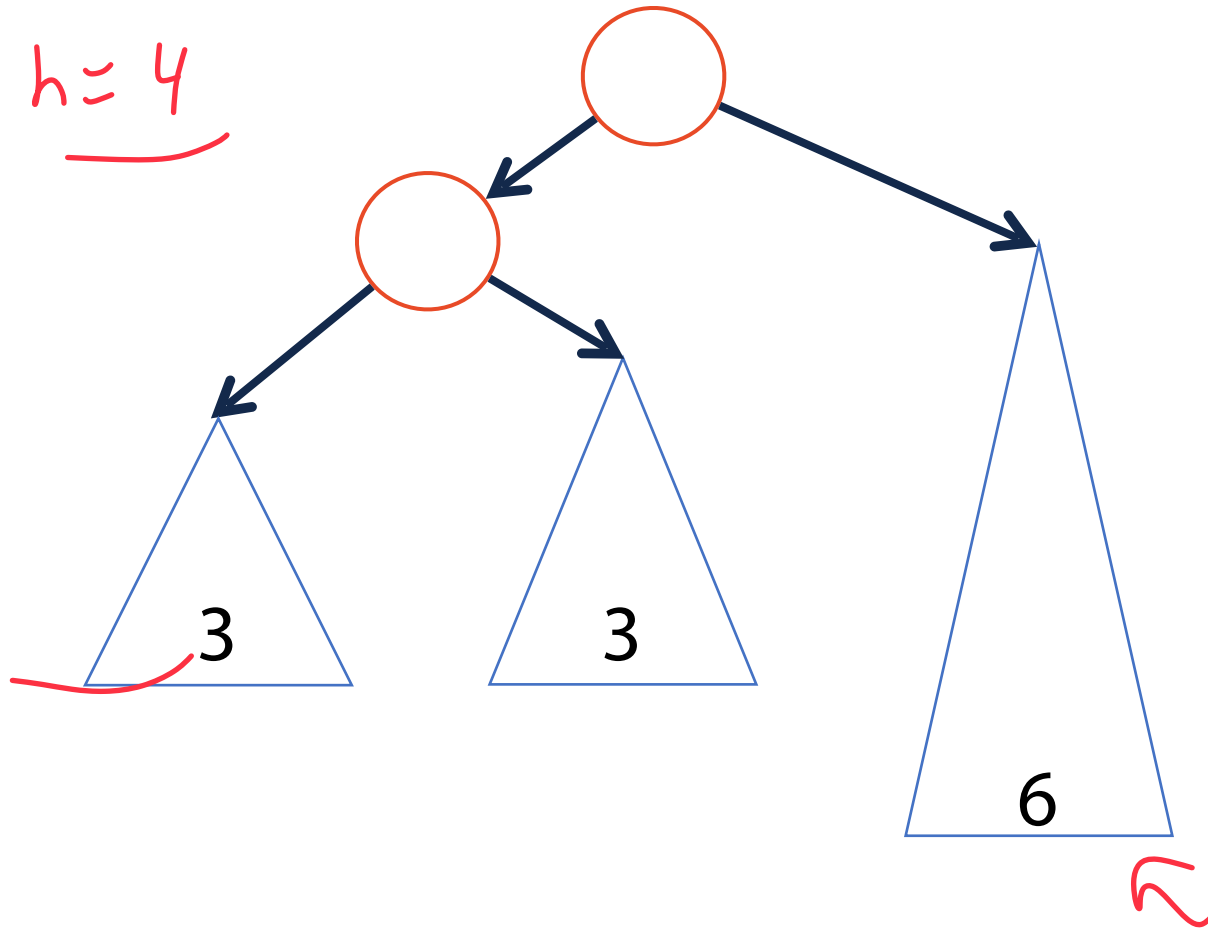
AVL Remove

Remove can cause an imbalance at every level



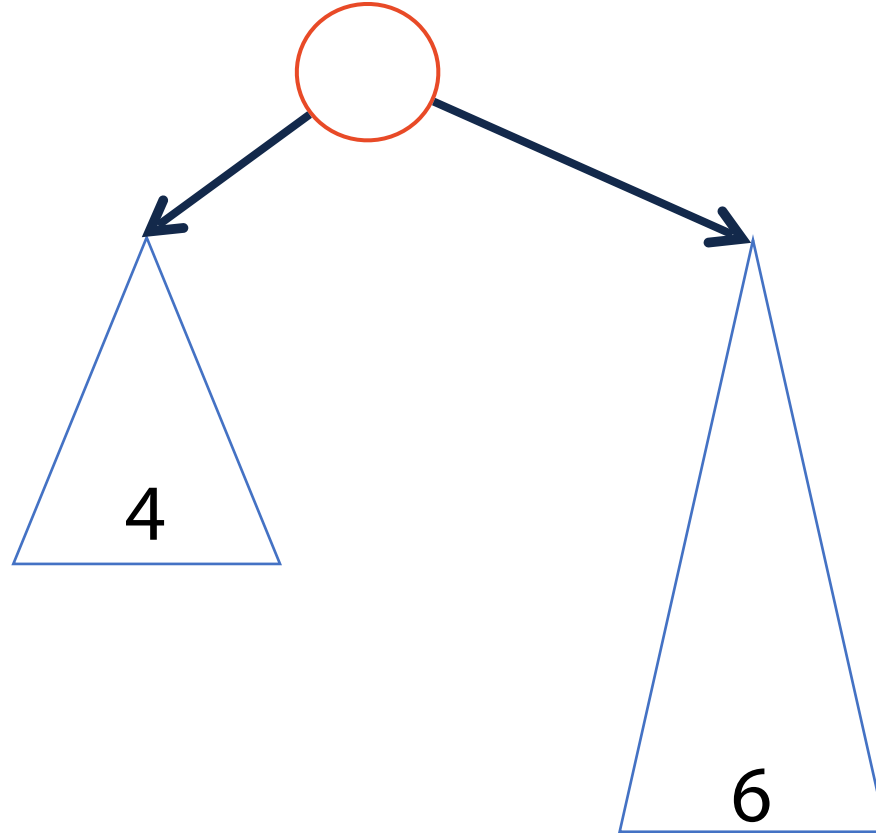
AVL Remove

Remove can cause an imbalance at every level



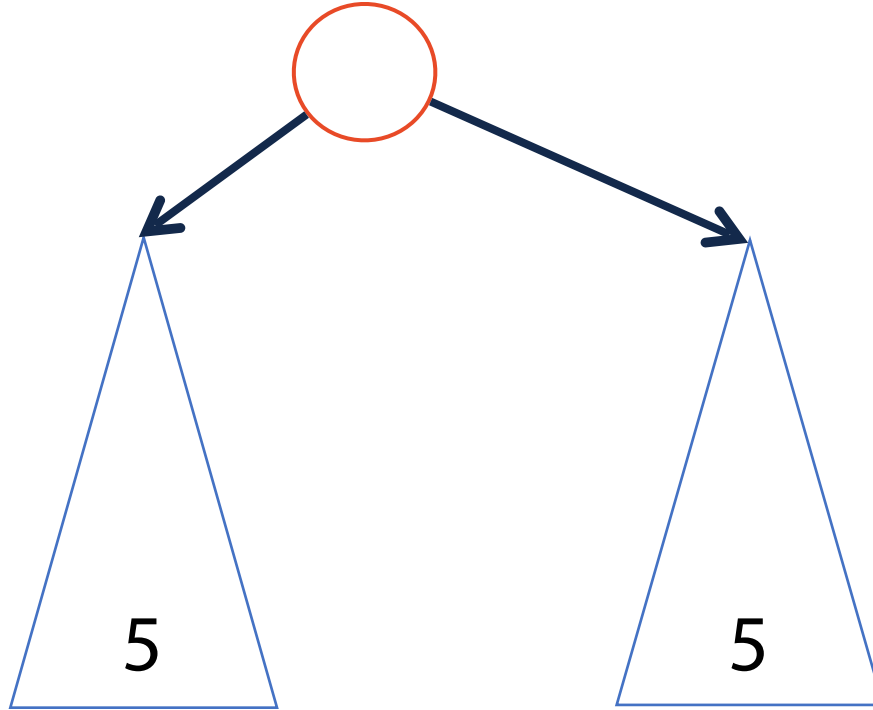
AVL Remove

Remove can cause an imbalance at every level



AVL Remove

Remove can cause an imbalance at every level



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!

$O(h)$	find	}	$O(h)$
$O(h)$	remove		
$O(h)$	rotations		

AVL Tree Analysis

For an AVL tree of height h :

Find runs in: $O(h)$.

Insert runs in: $O(h)$.

Remove runs in: $O(h)$.

Balanced tree
↓ has height

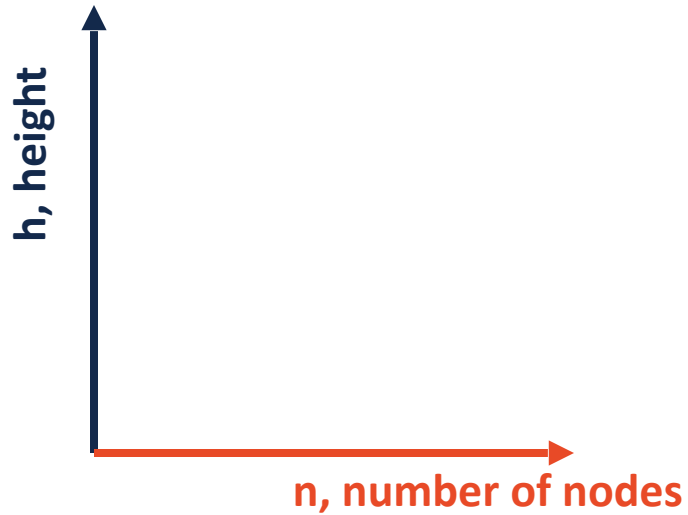
Claim: The height of the AVL tree with n nodes is: $O(\log n)$.

AVL Tree Analysis

Definition of big-O:

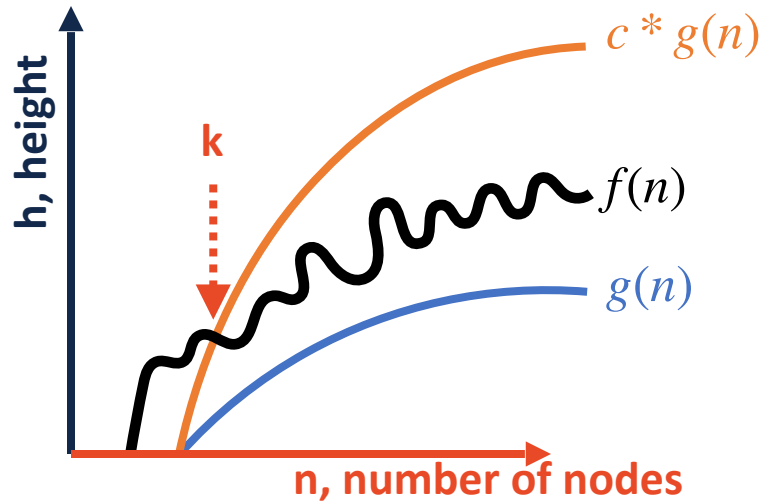
$$f(n) \text{ is } O(g(n)) \text{ iff } \exists c, k \text{ s.t. } f(n) \leq cg(n) \quad \forall n > k$$

...or, with pictures:



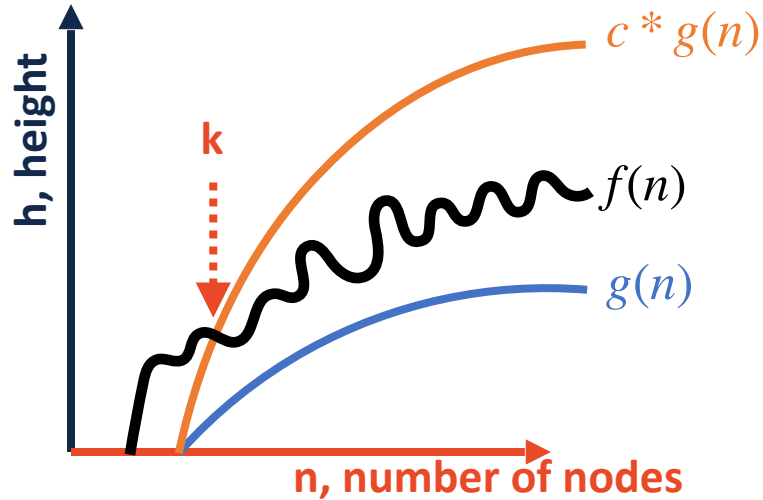
Actual AVL
analysis on Monday!
Friday lecture will be
named "AVL 3"

AVL Tree Analysis

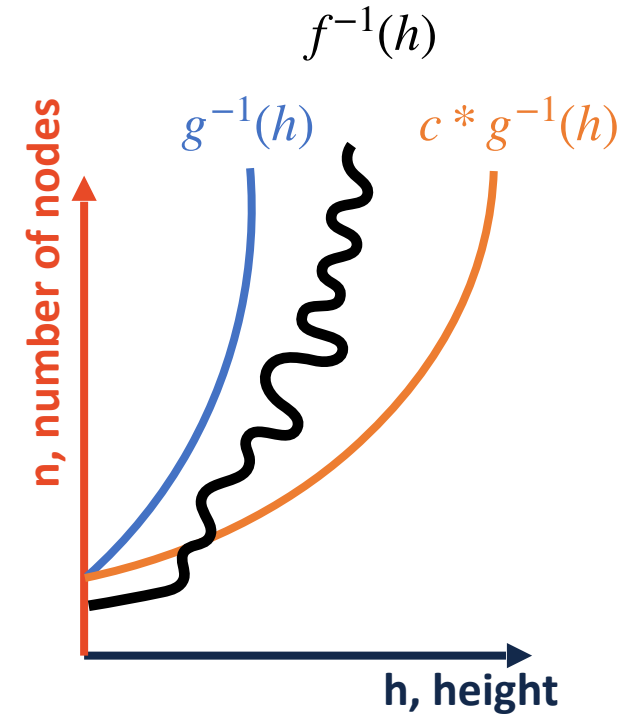


The height of the tree, $f(n)$, will always be less than $c \times g(n)$ for all values where $n > k$.

AVL Tree Analysis



$f(n)$ = "Tree height given nodes"



$f^{-1}(h)$ = "Nodes in tree given height"

The number of nodes in the tree, $f^{-1}(h)$, will always be greater than $c \times g^{-1}(h)$ for all values where $n > k$.

Plan of Action

Since our goal is to find the lower bound on n given h , we can begin by defining a function given h which describes the smallest number of nodes in an AVL tree of height h :

$N(h)$ = minimum number of nodes in an AVL tree of height h

Simplify the Recurrence

$$N(h) = 1 + N(h - 1) + N(h - 2)$$

Simplify the Recurrence

$$N(h) = 1 + N(h - 1) + N(h - 2)$$

$$N(h) > N(h - 1) + N(h - 2)$$

Simplify the Recurrence

$$N(h) = 1 + N(h - 1) + N(h - 2)$$

$$N(h) > N(h - 1) + N(h - 2)$$

$$N(h) > 2N(h - 2)$$

Simplify the Recurrence

$$N(h) = 1 + N(h - 1) + N(h - 2)$$

$$N(h) > N(h - 1) + N(h - 2)$$

$$N(h) > 2N(h - 2)$$

1) Know characteristic equation? Get answer immediately!

Simplify the Recurrence

$$N(h) = 1 + N(h - 1) + N(h - 2)$$

$$N(h) > N(h - 1) + N(h - 2)$$

$$N(h) > 2N(h - 2)$$

$$2) \text{ Unroll: } N(h) > 2N(h - 2) = 2 \left(2(N(h - 4)) \right) = 2^k \left(N(h - 2k) \right)$$

Simplify the Recurrence

$$N(h) = 1 + N(h - 1) + N(h - 2)$$

$$N(h) > N(h - 1) + N(h - 2)$$

$$N(h) > 2N(h - 2)$$

$$2) \text{ Unroll: } N(h) > 2N(h - 2) = 2(2N(h - 4)) = 2^k(N(h - 2k))$$

When $h - 2k = 0$, $k = h/2$. Thus $N(h) > 2^{h/2}$

Simplify the Recurrence

$$N(h) = 1 + N(h - 1) + N(h - 2)$$

$$N(h) > N(h - 1) + N(h - 2)$$

$$N(h) > 2N(h - 2)$$

3) Intuit approximate shape from recursion

Simplify the Recurrence

$$N(h) = 1 + N(h - 1) + N(h - 2)$$

$$N(h) > N(h - 1) + N(h - 2)$$

$$N(h) > 2N(h - 2)$$

By whatever strategy you like: $N(h) > 2^{h/2}$

State a Theorem

Theorem: An AVL tree of height h has at least $N(h) > 2^{h/2}$.

Proof by Induction:

I. Consider an AVL tree and let h denote its height.

II. Base Case: _____

An AVL tree of height _____ has at least _____ nodes.

Prove a Theorem

III. Base Case: _____

An AVL tree of height _____ has at least _____ nodes.

Prove a Theorem

IV. Induction Step: Assume for all heights $i < h$, $N(i) \geq 2^{i/2}$.

Prove that $N(h) \geq 2^{h/2}$

Prove a Theorem

IV. Induction Step: Assume for all heights $i < h$, $N(i) \geq 2^{i/2}$.

Prove that $N(h) \geq 2^{h/2}$

$$N(h) = 1 + N(h-1) + N(h-2)$$

$$N(h) > 2N(h-2)$$

$$N(h) > 2 * 2^{(h-2)/2}$$

$$N(h) > 2 * 2^{h/2-1}$$

$$N(h) > 2^{h/2}$$

Prove a Theorem

V. Using a proof by induction, we have shown that:

Prove a Theorem

V. Using a proof by induction, we have shown that:

$N(h) \geq 2^{h/2}$, where $N(h)$ is the **min # of nodes of a tree of height h**

But we need to know n , the **# of nodes in any tree of height h**



Prove a Theorem

V. Using a proof by induction, we have shown that:

$N(h) \geq 2^{h/2}$, where $N(h)$ is the **min # of nodes of a tree of height h**

But we need to know n , the **# of nodes in any tree of height h**

$$n \geq N(h)$$

$$\log(n) \geq \frac{h}{2}$$

$$h \leq 2 \log(n)$$

AVL Runtime Proof

An upper-bound on the height of an AVL tree is **$O(\lg(n))$** :

$N(h)$:= Minimum # of nodes in an AVL tree of height h

$$N(h) = 1 + N(h-1) + N(h-2)$$

$$> 1 + 2^{(h-1)/2} + 2^{(h-2)/2}$$

$$> 2 \times 2^{(h-2)/2} = 2^{(h-2)/2+1} = 2^{h/2}$$

Theorem #1:

Every AVL tree of height h has at least $2^{h/2}$ nodes.

AVL Runtime Proof

An upper-bound on the height of an AVL tree is $O(\lg(n))$:

$$\text{\# of nodes } (n) \geq N(h) > 2^{h/2}$$

$$n > 2^{h/2}$$

$$\lg(n) > h/2$$

$$2 \times \lg(n) > h$$

$$h < 2 \times \lg(n) \quad , \text{ for } h \geq 1$$

Proved: The maximum number of nodes in an AVL tree of height h is less than $2 \times \lg(n)$.

Summary of Balanced BST

Pros:

Cons:

Summary of Balanced BST

AVL Trees

- Max height: $1.44 * \lg(n)$
- Rotations:

Summary of Balanced BST

AVL Trees

- Max height: $1.44 * \lg(n)$
- Rotations:
 - Zero rotations on find
 - One rotation on insert
 - $O(h) == O(\lg(n))$ rotations on remove

Red-Black Trees

- Max height: $2 * \lg(n)$
- Constant number of rotations on insert (max 2), remove (max 3).

Summary of Balanced BST

Pros:

- Running Time:
 - Improvement Over:
- Great for specific applications:

Summary of Balanced BST

Cons:

- Running Time:
- In-memory Requirement:

Red-Black Trees in C++

C++ provides us a balanced BST as part of the standard library:

```
std::map<K, V> map;
```

```
V & std::map<K, V>::operator[] ( const K & )
```

```
std::map<K, V>::erase( const K & )
```

Red-Black Trees in C++

```
iterator std::map<K, V>::lower_bound( const K & );  
iterator std::map<K, V>::upper_bound( const K & );
```

Every Data Structure So Far

	Unsorted Array	Sorted Array	Unsorted Linked List	Sorted Linked List	Binary Tree	BST	AVL
Find							
Insert							
Remove							
Traverse							