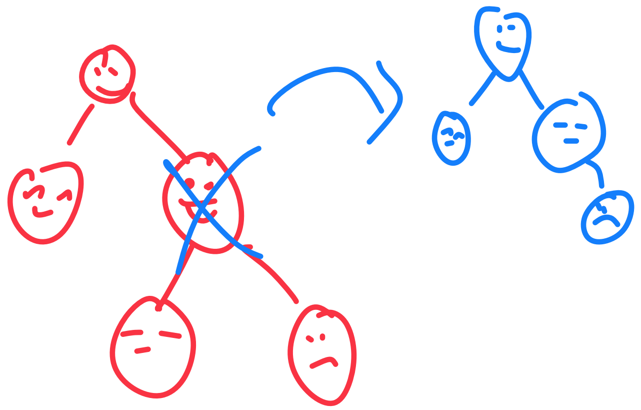
Data Structures

Binary Search Trees 2

CS 225
Brad Solomon

September 23, 2024



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Exam 2 (10/02 — 10/04)

Autograded MC and one coding question

Manually graded short answer prompt

Practice exam will be released on PL

Topics covered can be found on website

Last content on exam is today!

Registration started September 19

https://courses.engr.illinois.edu/cs225/fa2024/exams/

Learning Objectives

Build conceptual and coding understanding of BST

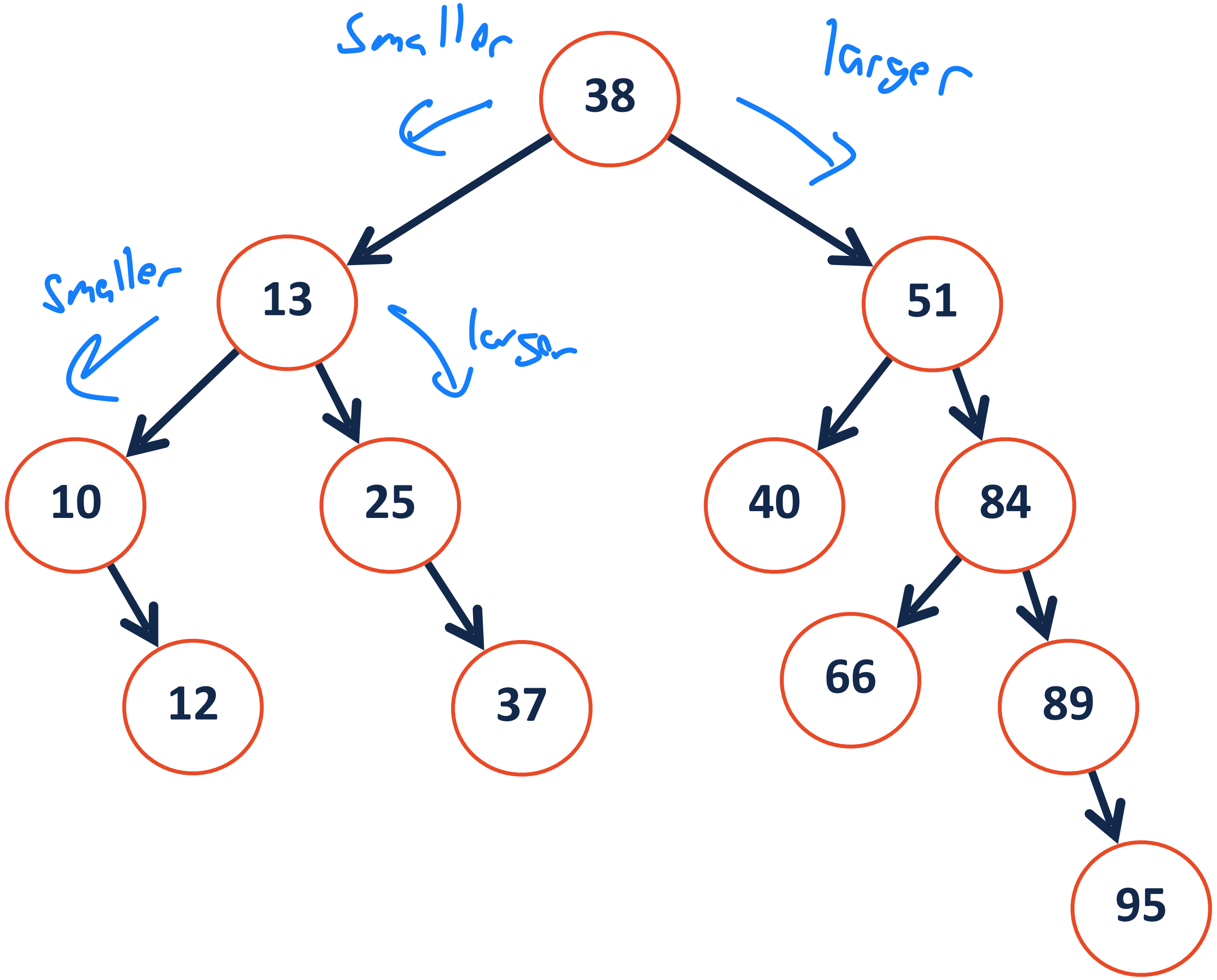
Discuss pros and cons of BST (and possible improvements)

Binary Search Tree (BST)

A **BST** is a binary tree $T = TreeNode(val, T_L, T_r)$ such that:

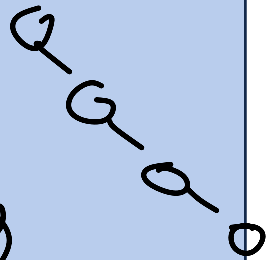
$$\forall n \in T_L,\ n.val < T.val$$

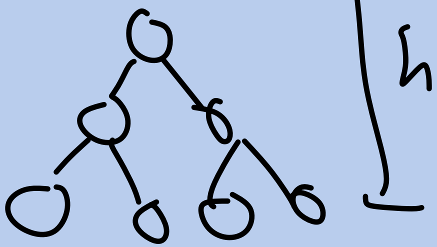
$$\forall n \in T_R,\ n.val > T.val$$



```
template<typename K, typename V>

TreeNode *& _find(TreeNode *& root, const K & key) {


// Base Case
if(root == nullptr || root->key == key){
    return root;
}

// Recursive Step (“Combining step” is ‘return’)
if (root->key > key){
    return _find(root->left, key);
}

return _find(root->right, key);


}
```

$O(n)$

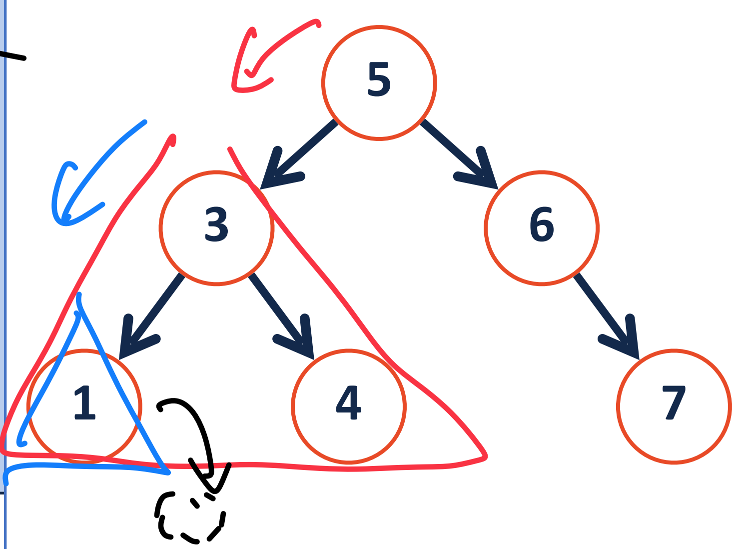




$h \sim \log N$

$\boxed{O(h)}$

Find(2)

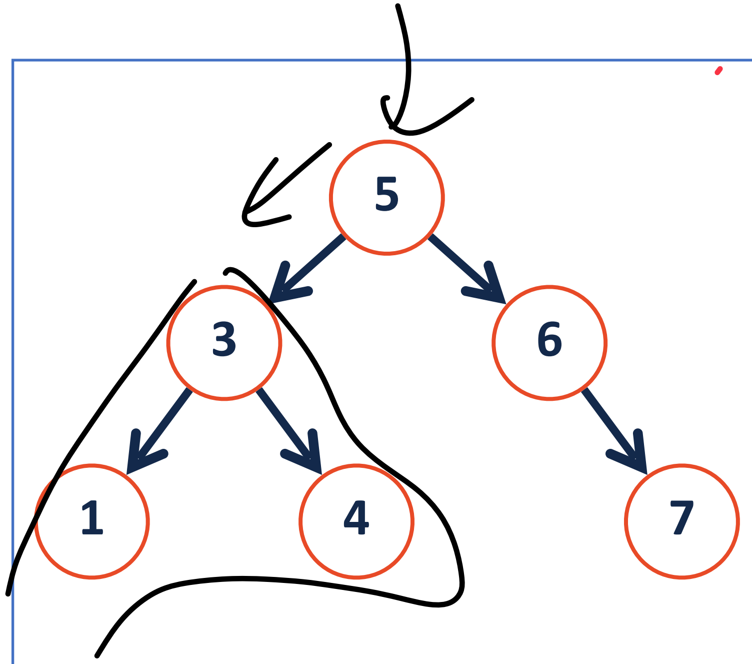


Find (5, 2)

↳ Find (3, 2)

↳ Find (1, 2) ref to pointer (→right)

↳ Find (1→right (nullptr), 2)

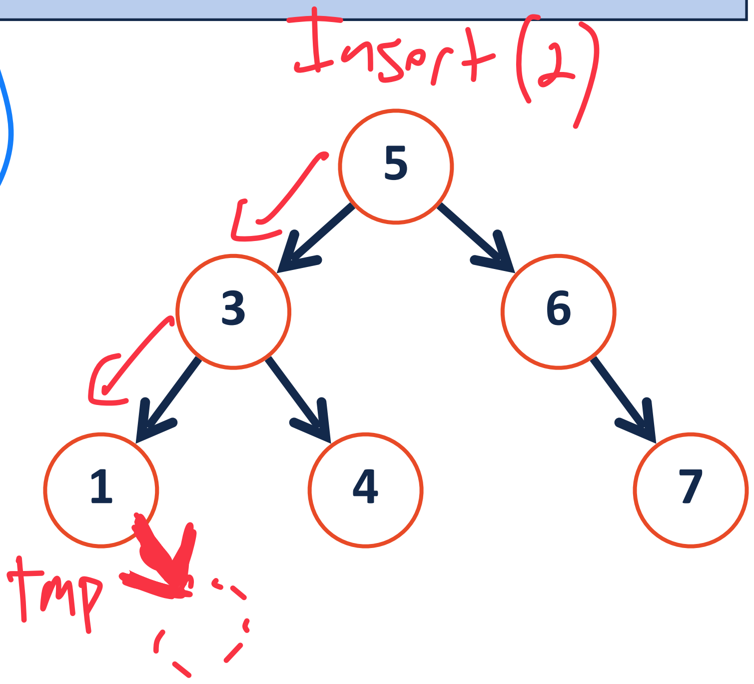


```
template<typename K, typename V>

void _insert(const K & key, const V & val) {

  TreeNode *& tmp = _find(root, key);

  tmp = new treeNode(key, val);


}
```

1) Find location to insert $O(h)$

2) Add new node at location $O(1)$

$\boxed{O(h)}$



BST Remove

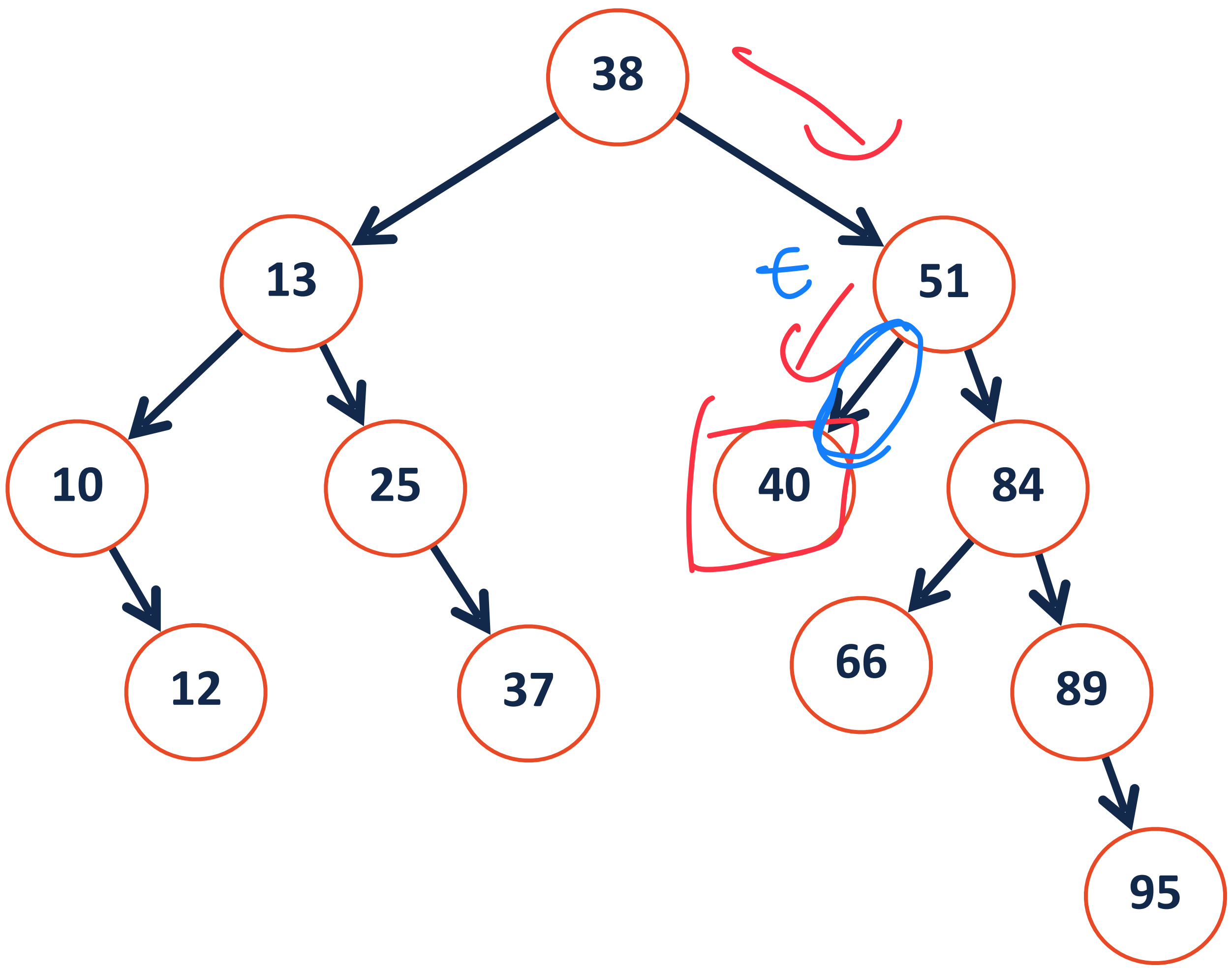
1) Find node to remove

2) Remove it!

↳ delete t

t = nullptr

remove(40)

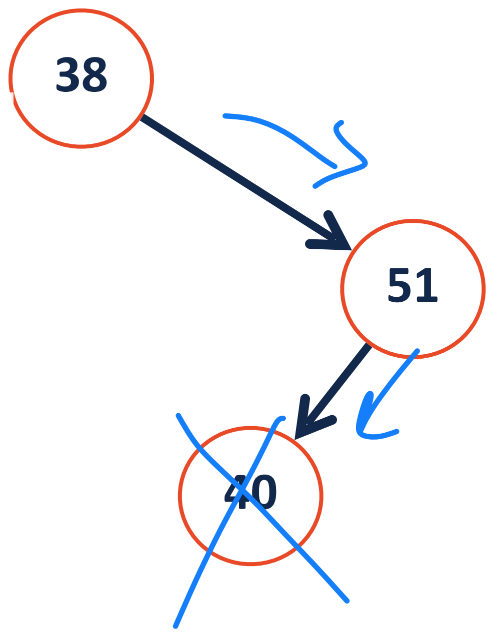


BST Remove

remove(40)

0-Child Case

```
TreeNode *& t = _find(root, 40);
delete t;
t = nullptr;
```



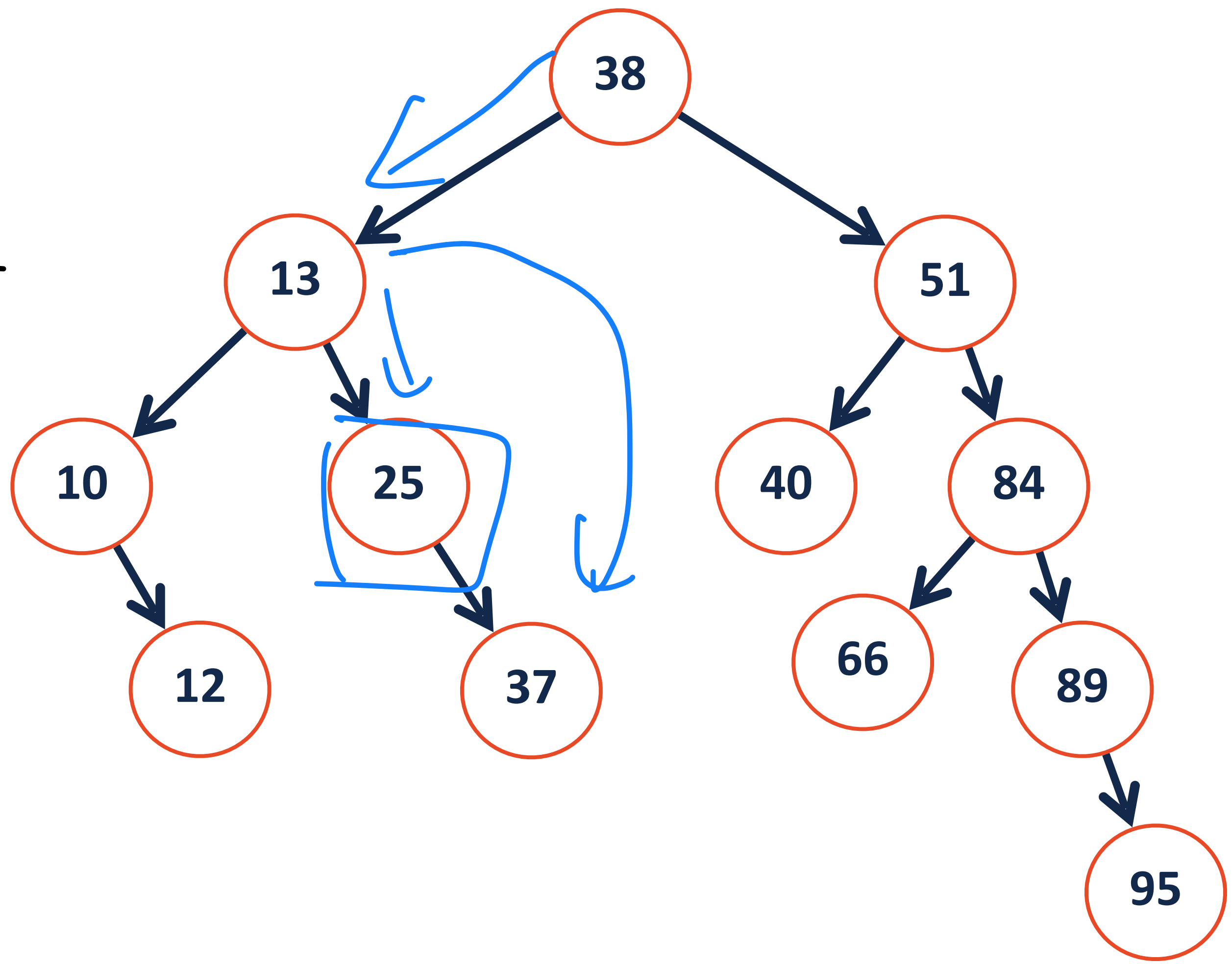
BST Remove

remove(25)

1) Find

2) Remove

    1) make tmp pointer to target
    2) update parent to point to target's child
    3) delete tmp



BST Remove

remove(25)

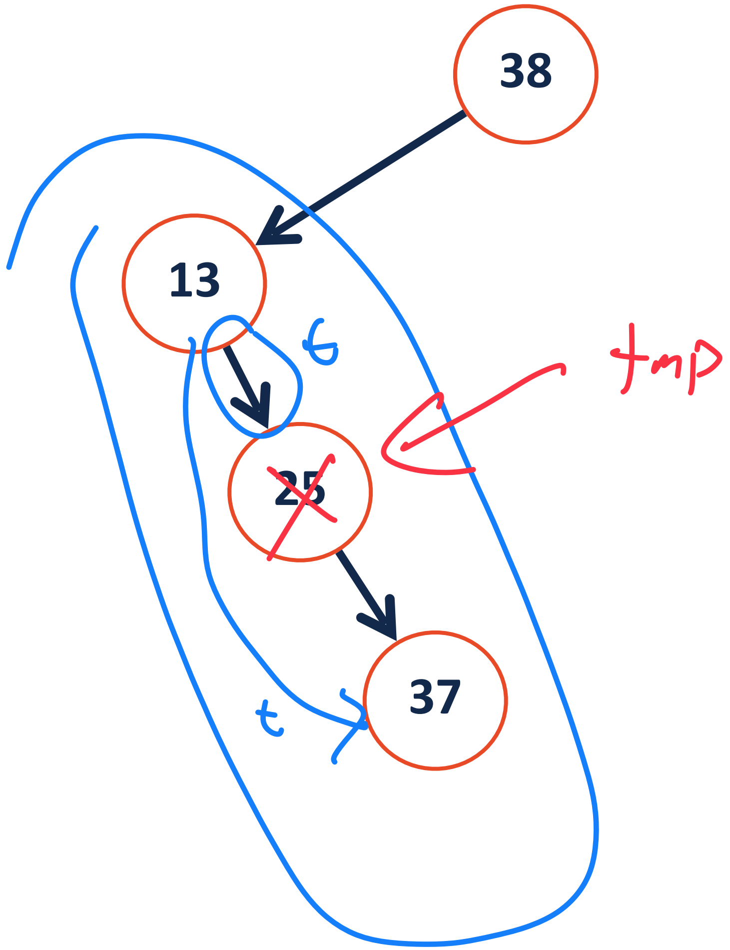
1-Child Case

```
TreeNode *& t = _find(root, 25);
TreeNode * tmp = t;
t = t->right;
delete tmp;
```

LL remove!

right child

if left, use left!



BST Remove

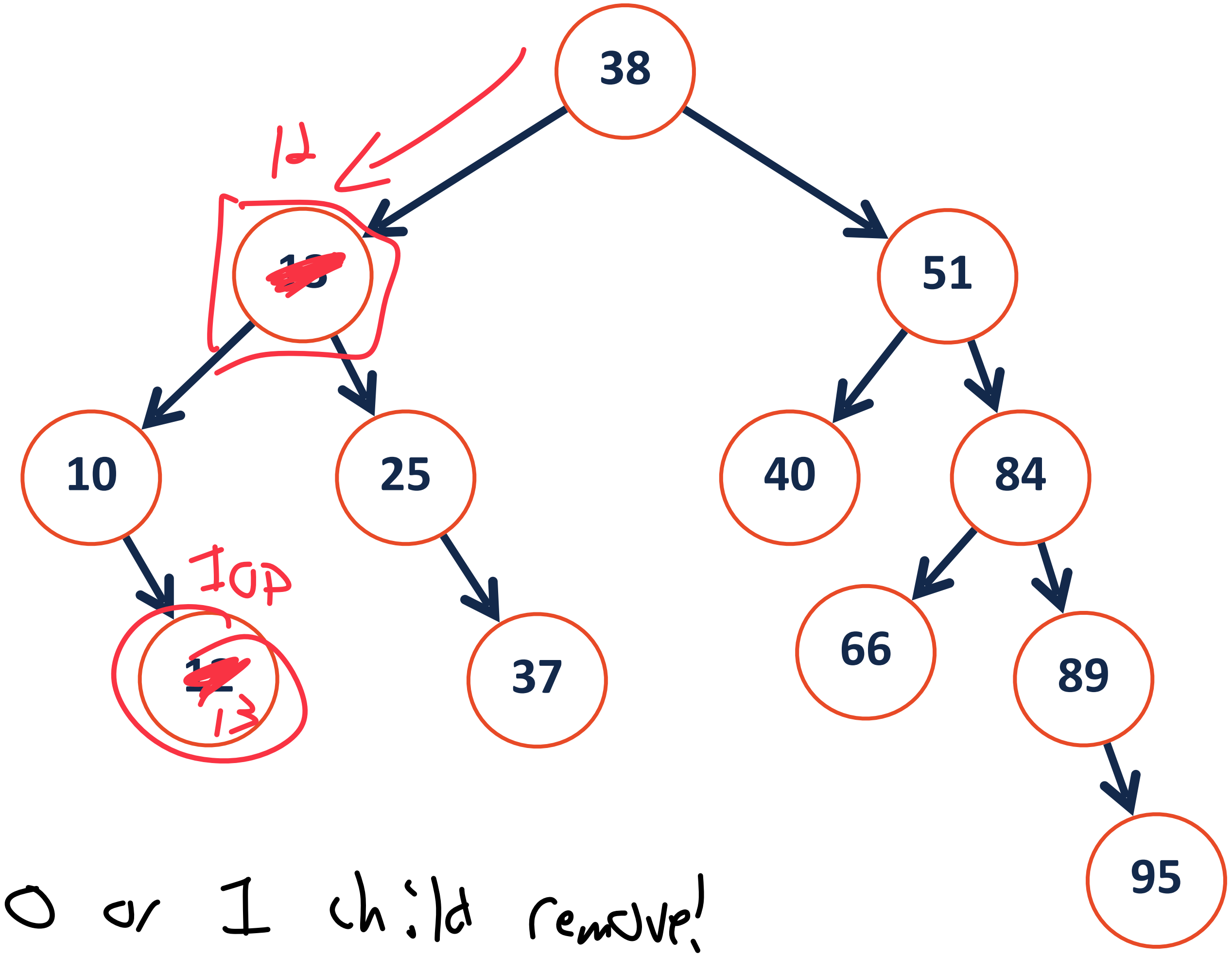
remove(13)

1) Find target
2) Find In-order predecessor
   or
   In-order successor

3) Swap target & IOP

4) Remove(13)
   ↳ But at subtree

↳ This will always be 0 or 1 child remove!



BST In-Order ________

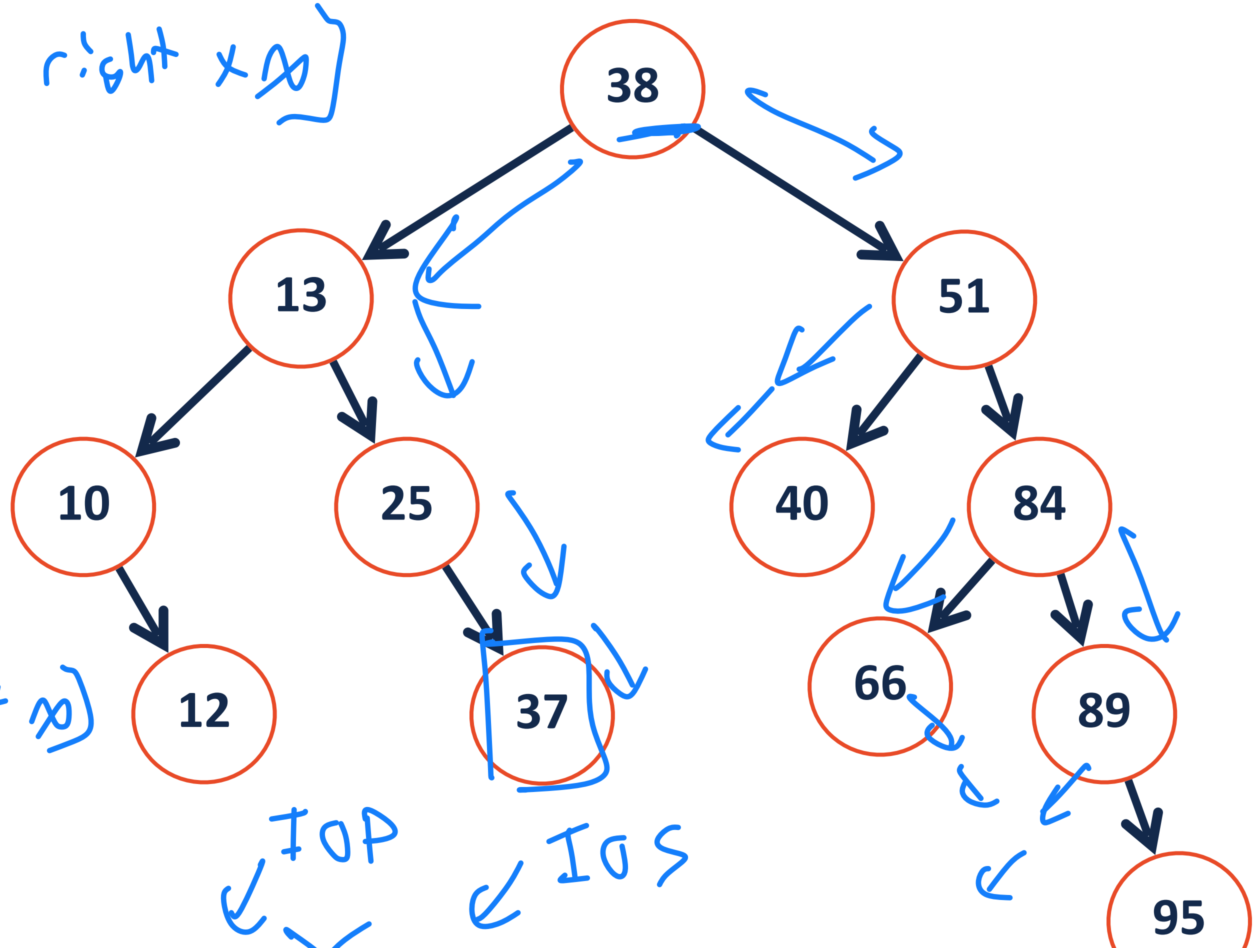
IOP / IOS guaranteed 0 or 1-child

In-Order Predecessor (Next smallest node in subtree)

Rightmost left child [go left once, right x ∞]

IOP(38) = 37

IOP(84) = 66



10, 12, ~~13~~, 25, 37

In-Order Successor (Next Largest)

Leftmost right child [right once, left ∞]

IOS(38) = 40

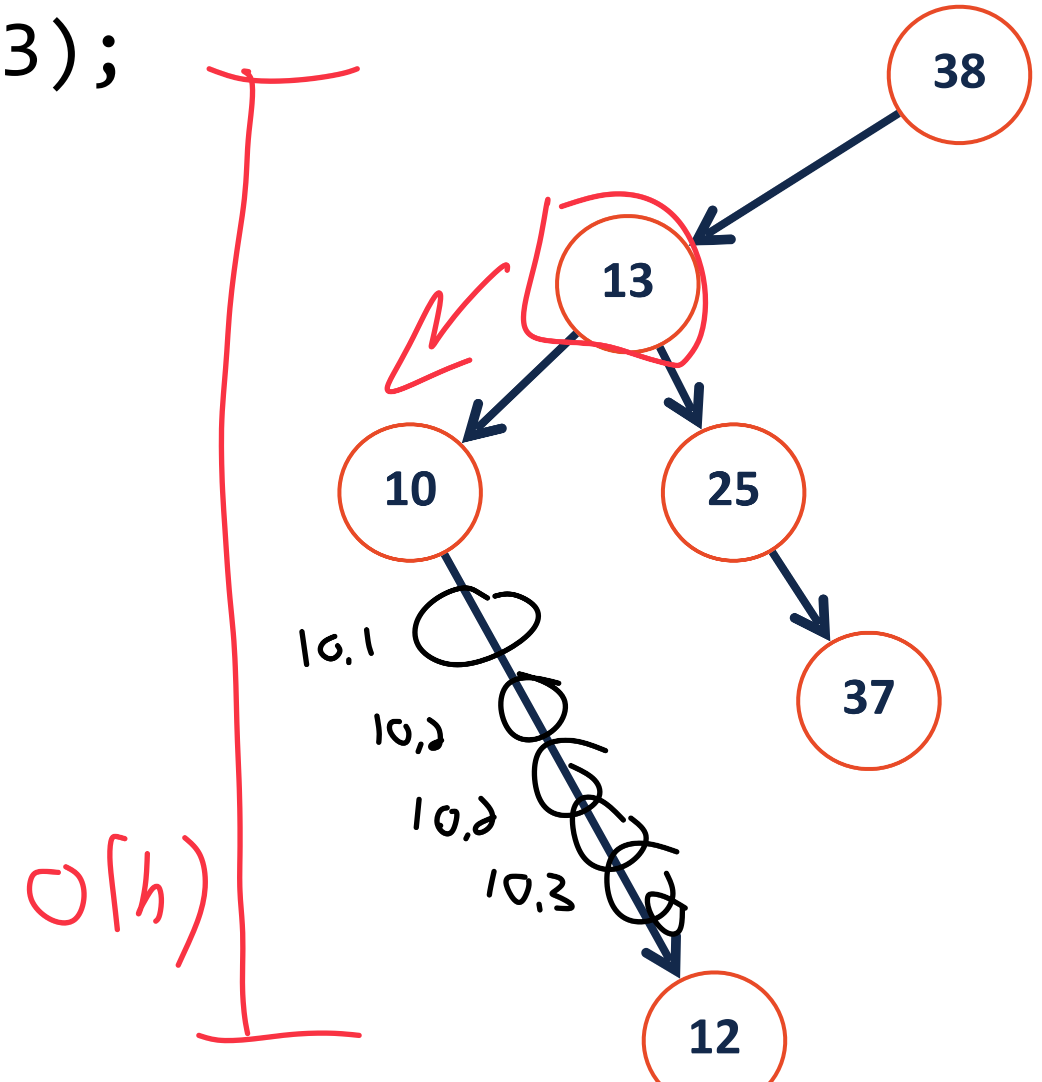
IOS(84) = 89

BST Remove

remove(13)

2-Child Case

```
TreeNode *& t = _find(root, 13);

TreeNode * IOP = getIOP(t);

swap(t, iop);

remove(13); //starting from t
```

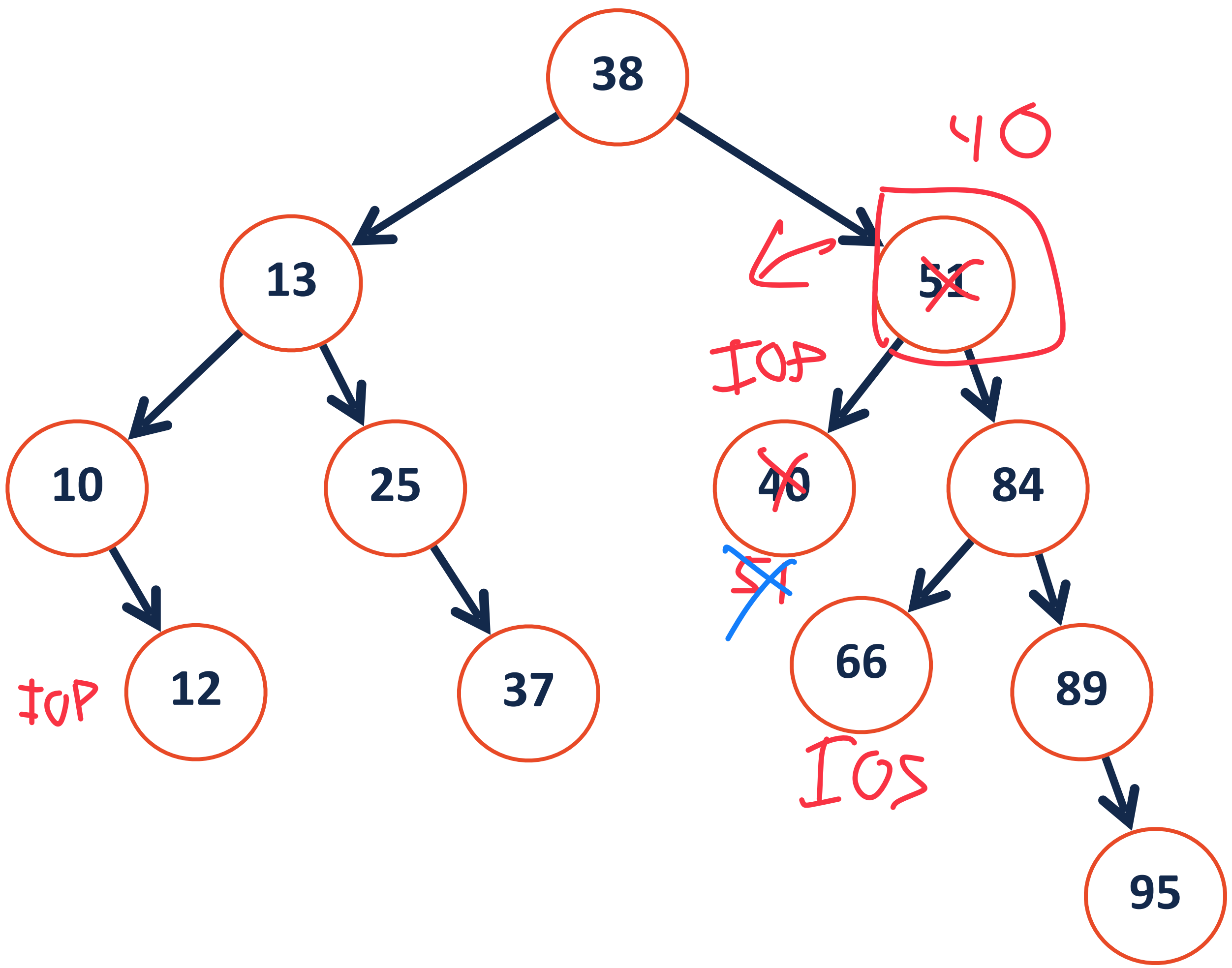


BST Remove

remove(51)

1) Find(51)
2) IOP = 40
3) swap(51, 40)
4) Remove(51) @ target subtree
   ↳ or jump there if
   we keep ref to pointer to IOP

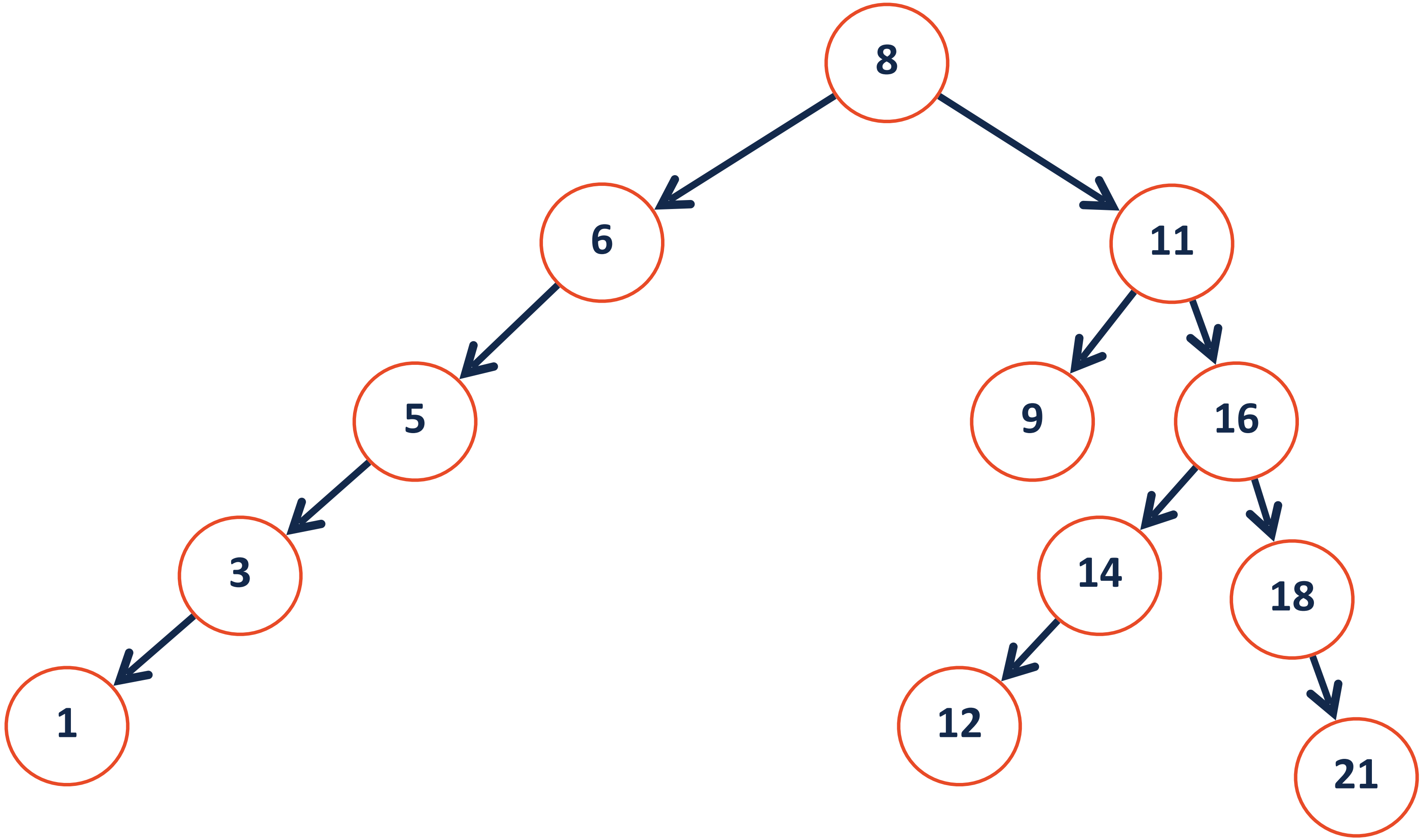
↓ student suggested!



BST Remove

Exercise for you!

What will the tree structure look like if we remove node 16 using IOS?



```
template<typename K, typename V>

void _remove(TreeNode *& root, const K & key) {
```

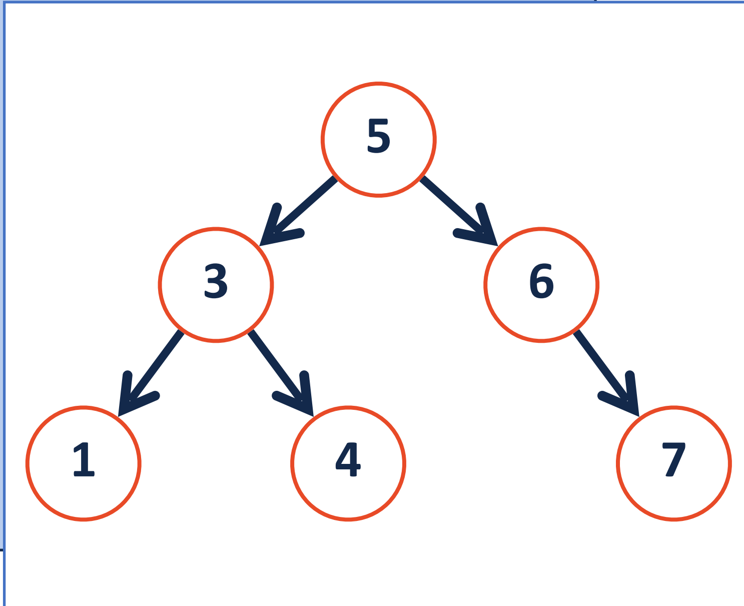
This weeks lab!

0 - child

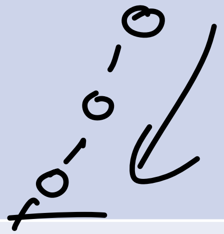
1 - child

2 - child

```
}
```



BST Analysis – Running Time

Operation	BST Worst Case
find	$O(h)$ 
insert	$O(h)$ b/c S
remove	$\underbrace{O(h)}_{\text{find}} + \underbrace{O(h)}_{\text{find (IOP)}} + \underbrace{O(h)}_{\text{remove( )}} = O(h)$
traverse	$O(n)$

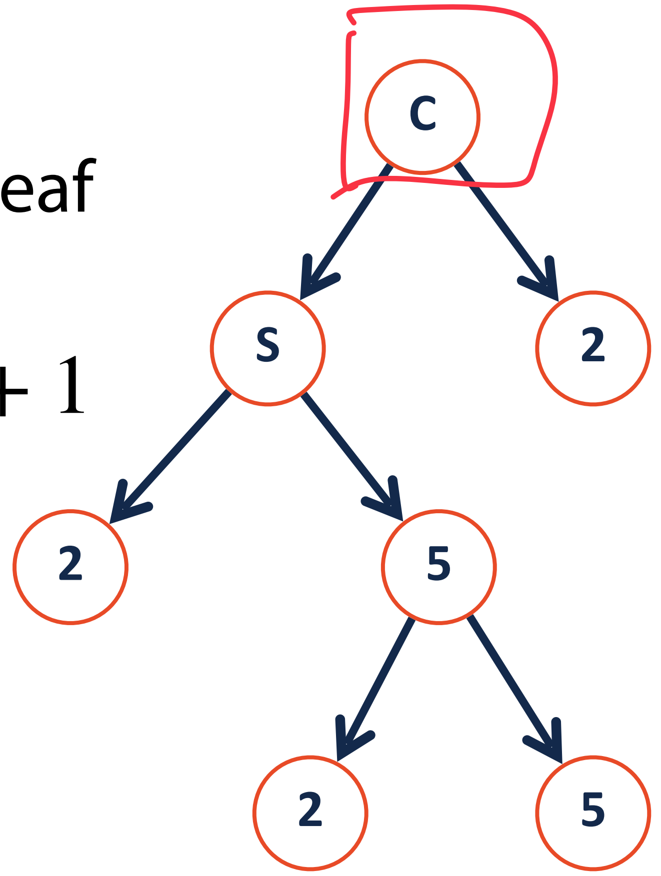
Binary Tree Height

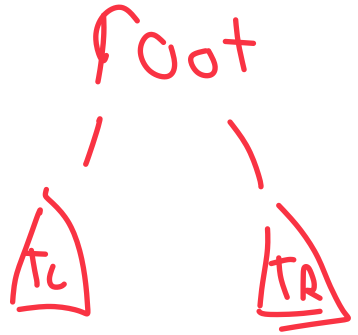
Height: The length of the longest path from root to leaf

$$Height(root) = max\left(\ Height(T_L),\, Height(T_R)\right) + 1$$

Given this recursion, what is base case?

r Height is zero

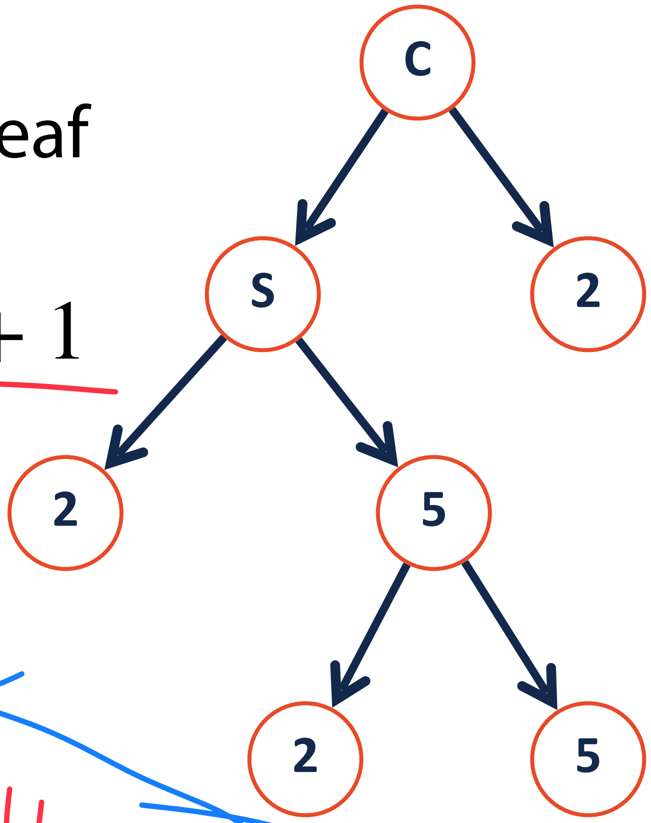




Binary Tree Height

Height: The length of the longest path from root to leaf

$$Height(root) = \underline{max\left(\; Height(T_L), \, Height(T_R)\right) + 1}$$



Given this recursion, what is base case?

$$Height(\varnothing) = -1$$

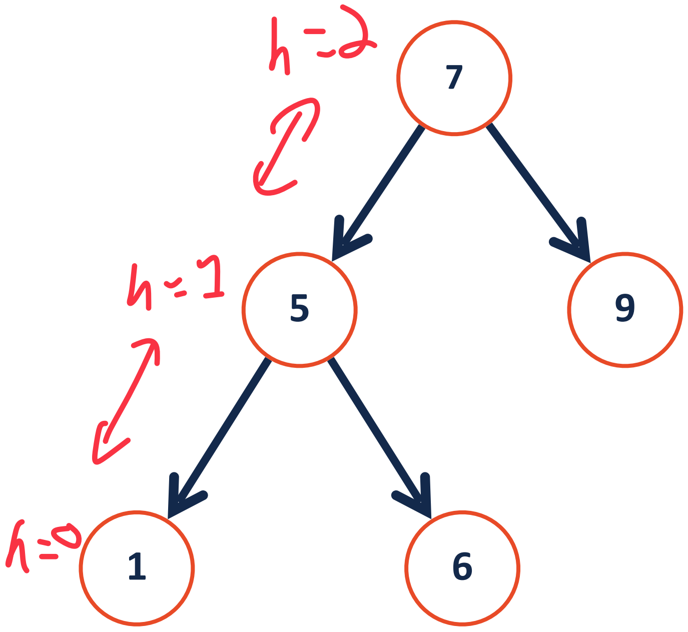
empty tree

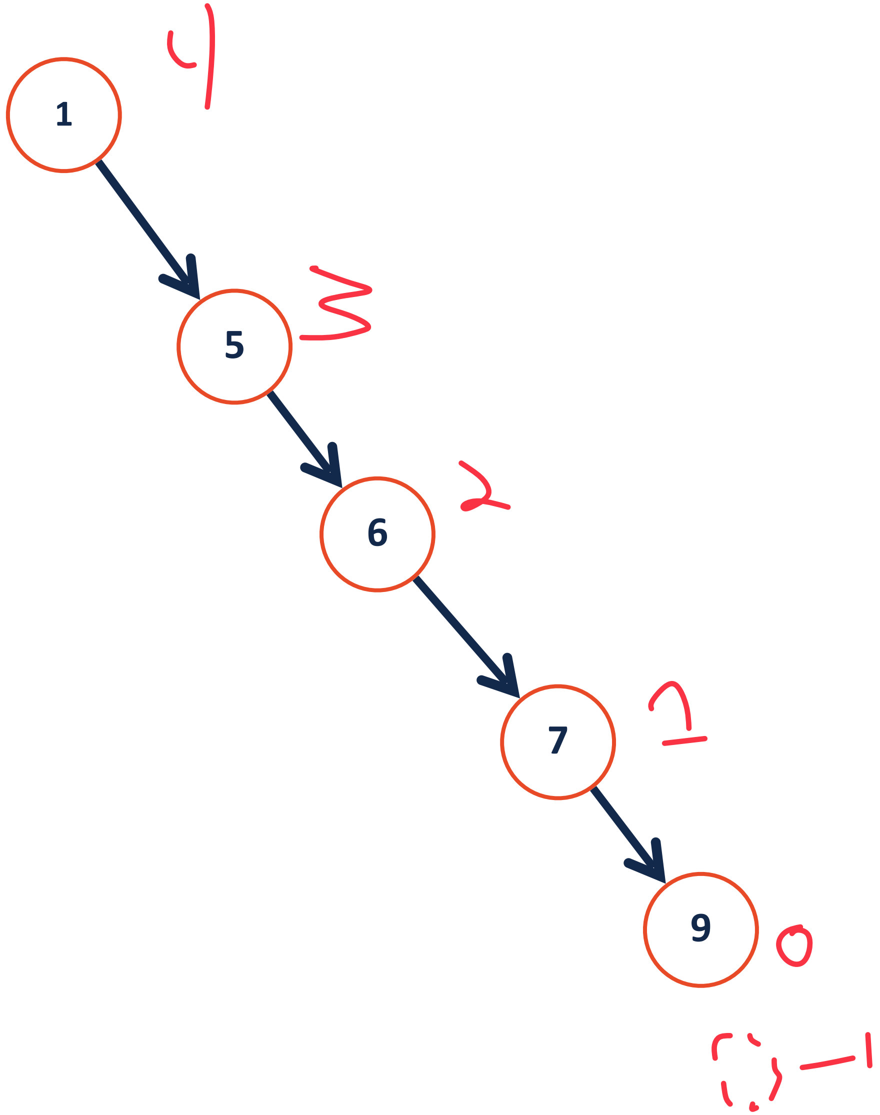
c

H = 0

max(-1 , -1) +1 = 0

Limiting the height of a tree





Option A: Correcting bad insert order

The height of a BST depends on the order in which the data was inserted

Insert Order: [1, 3, 2, 4, 5, 6, 7]

Insert Order: [4, 2, 3, 6, 7, 1, 5]

AVL-Tree: A self-balancing binary search tree

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