Data Structures

Binary Search Trees 2

CS 225

September 20, 2023

Brad Solomon & G Carl Evans

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Extra Credit Project

Propose, implement, and benchmark an algorithm or data structure not covered in CS 225

Teams of two

Extra credit only if you complete the project.

Four key parts:

1. Proposal
2. Development Logs
3. Mid-project Checkin
4. Final Submission

Learning Objectives

Review binary search trees

Continue implementing BST ADT

Discuss pros and cons of BST (and possible improvements)

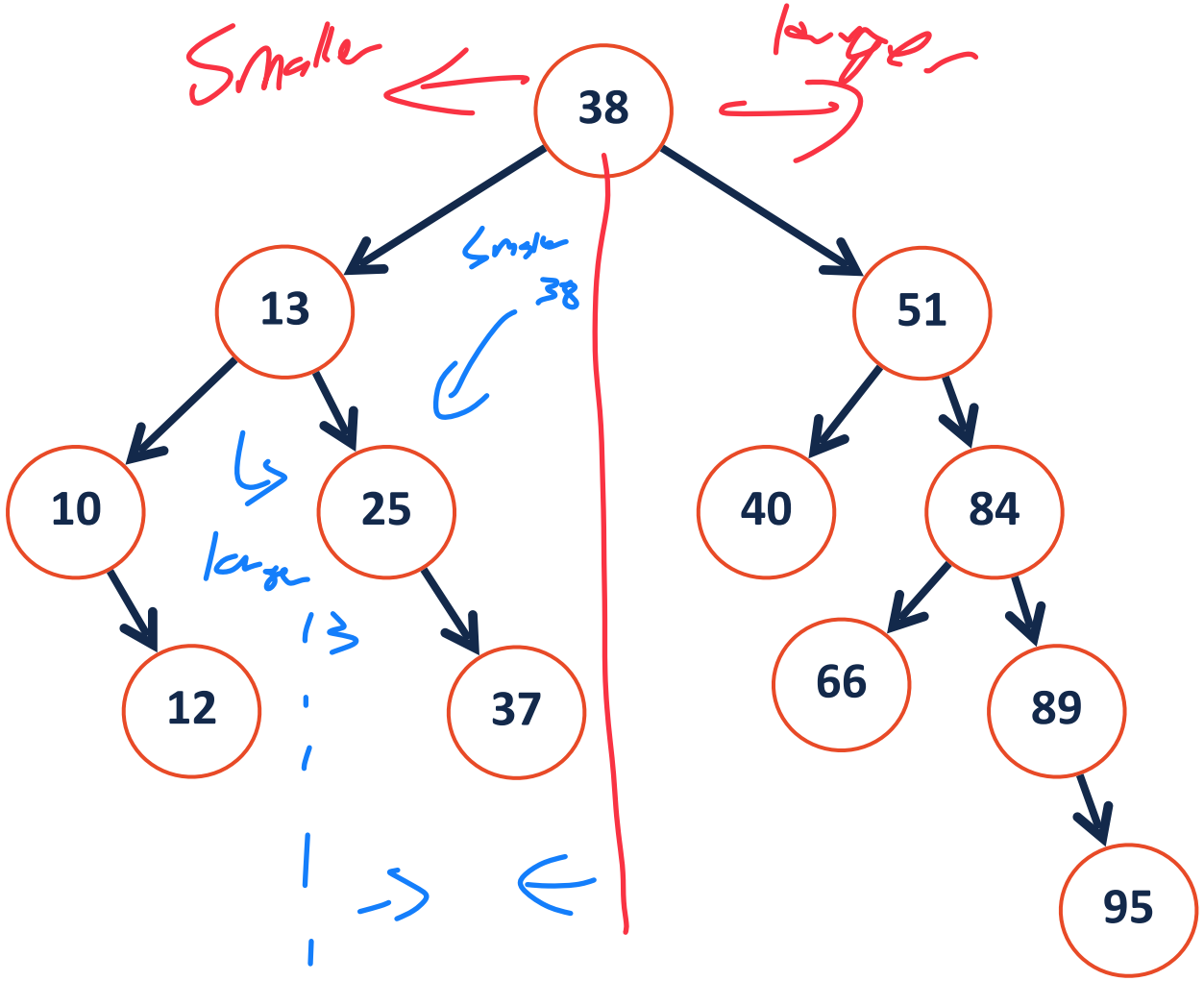
BT → BST → ???

Binary Search Tree (BST)

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

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

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



```
1  template<typename K, typename V>
2
3  TreeNode *& _find(TreeNode *& root, const K & key) {
4
5  // Base case
6     1) root == nullptr -> return root;
7
8     2) root->key == key -> return root;
9
10
11 // Recursive Step
12    Look at root->key
13       |
14       |
15       |
16    |> root->key < key -> go right
17
18    |> root->key > key -> go left
19
20
21
22
23 }
```

* & vs **

"value"

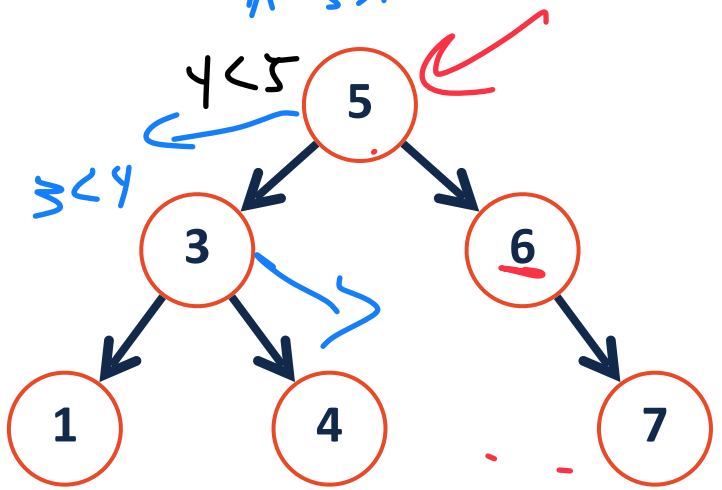
find(4)

nullptr

key

R2

* 5->right



BST Insert

1) use find(33) → TreeNode *p
   tmp =

2) tmp = new TreeNode(33, v);

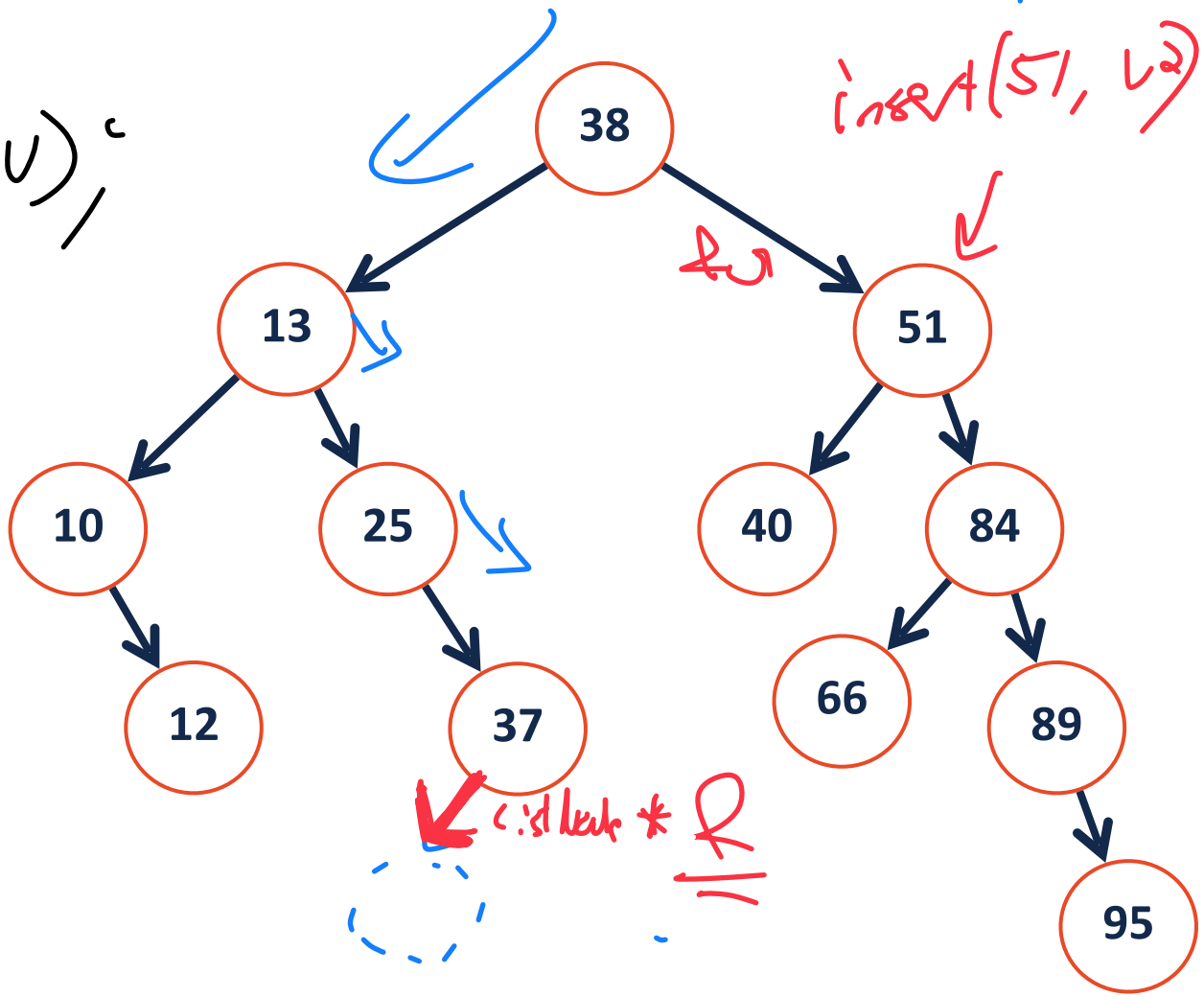
Main Point

Target

Duplicate 'Bob_1', 'Bob_2'
↳ key =
value [ ]

TreeNode
↳ key K
value V

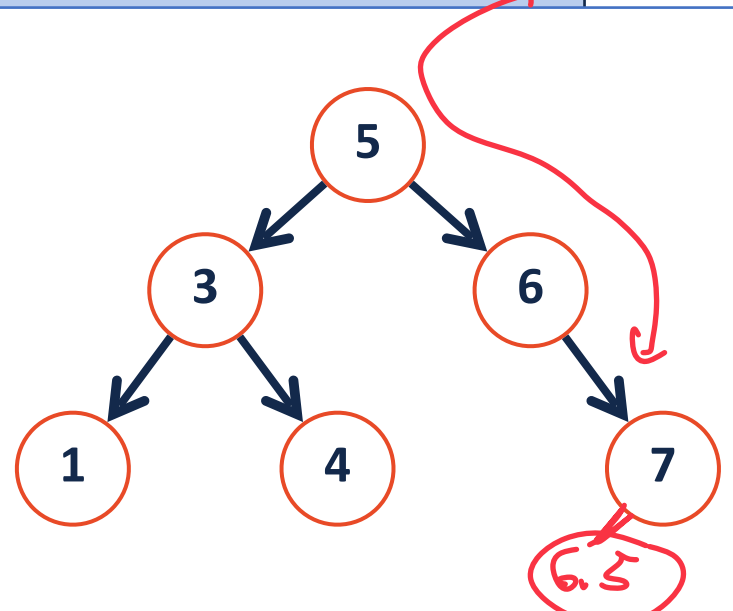
insert(33, v)



```
1  template<typename K, typename V>
2
3  void _insert(const K & key, const V & val) {
4
5      return _insert(root, key, val);
6  }
7
```

```
1  template<typename K, typename V>
2
3  void _insert(TreeNode *& root, const K & key, const V & val) {
4
5      1) Do a find ( )
6
7      2) Insert new Node
8
9
10
11
12
13
14
15
16 }
```

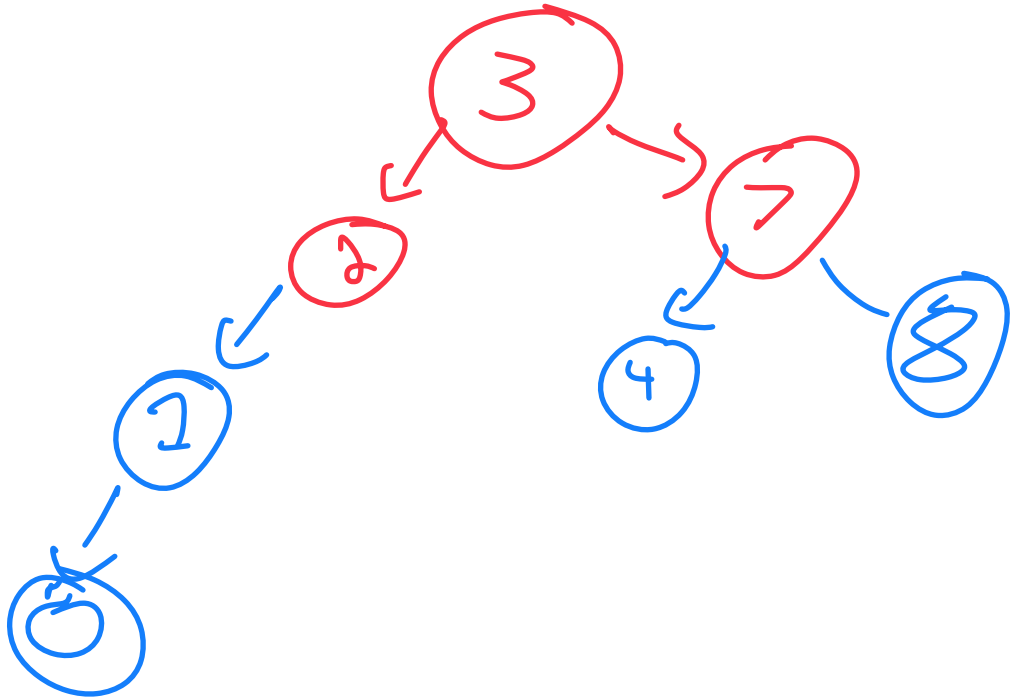
6.5



BST Insert

The order of insert matters!

What binary tree would be formed by inserting the following sequence of integers: `[3, 7, 2, 1, 4, 8, 0]`



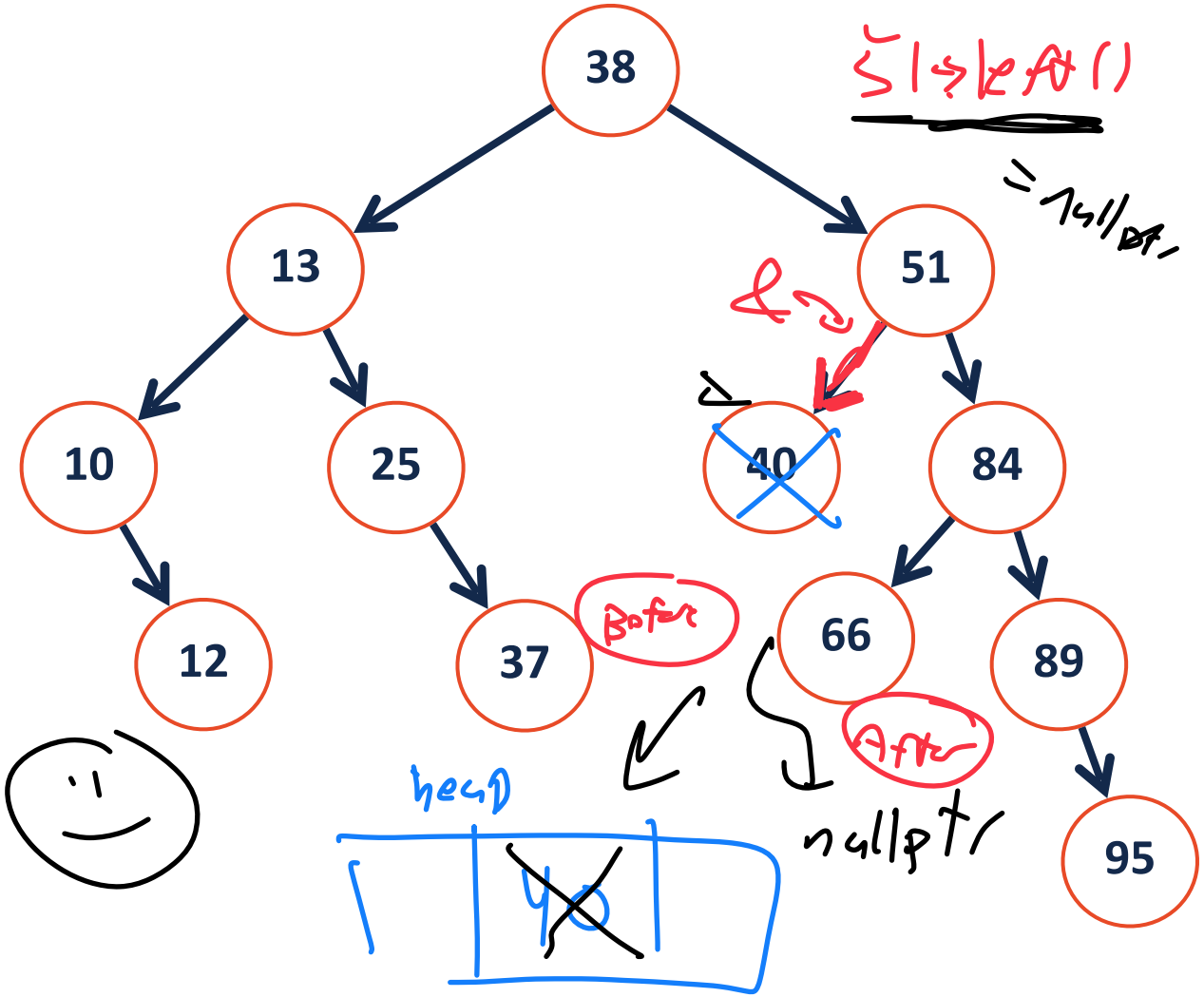
BST Remove

remove(40)

1) tmp = find(40)
2) delete tmp;
3) tmp = nullptr;

Because we are removing a leaf

↳ 0-child remove

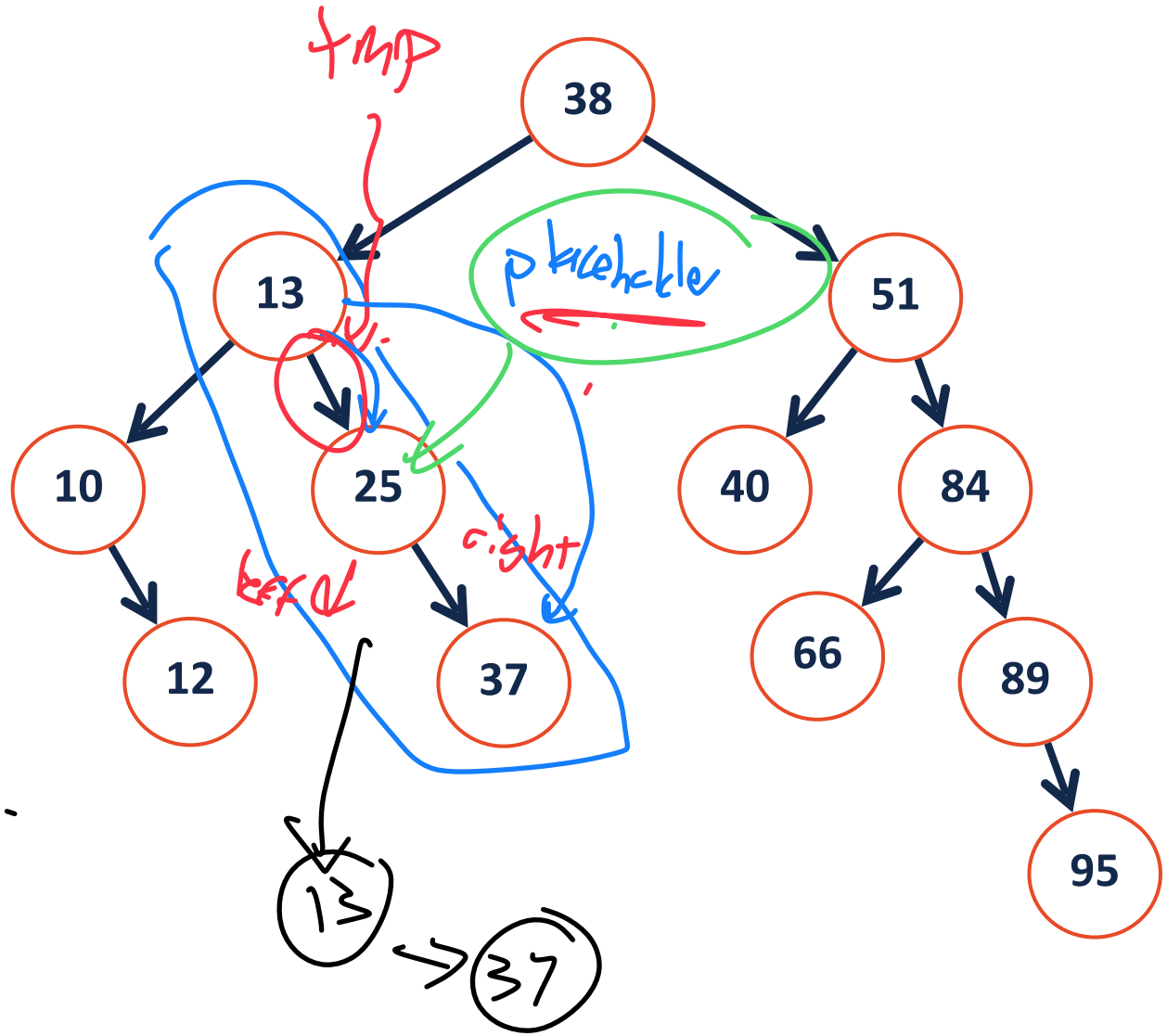


BST Remove

remove(25)

1) tmp = find(25)
2) TreeNode * placeholder = tmp;
3) tmp = tmp->right
4) Delete placeholder

Linked list remove :)



BST Remove

remove(13)

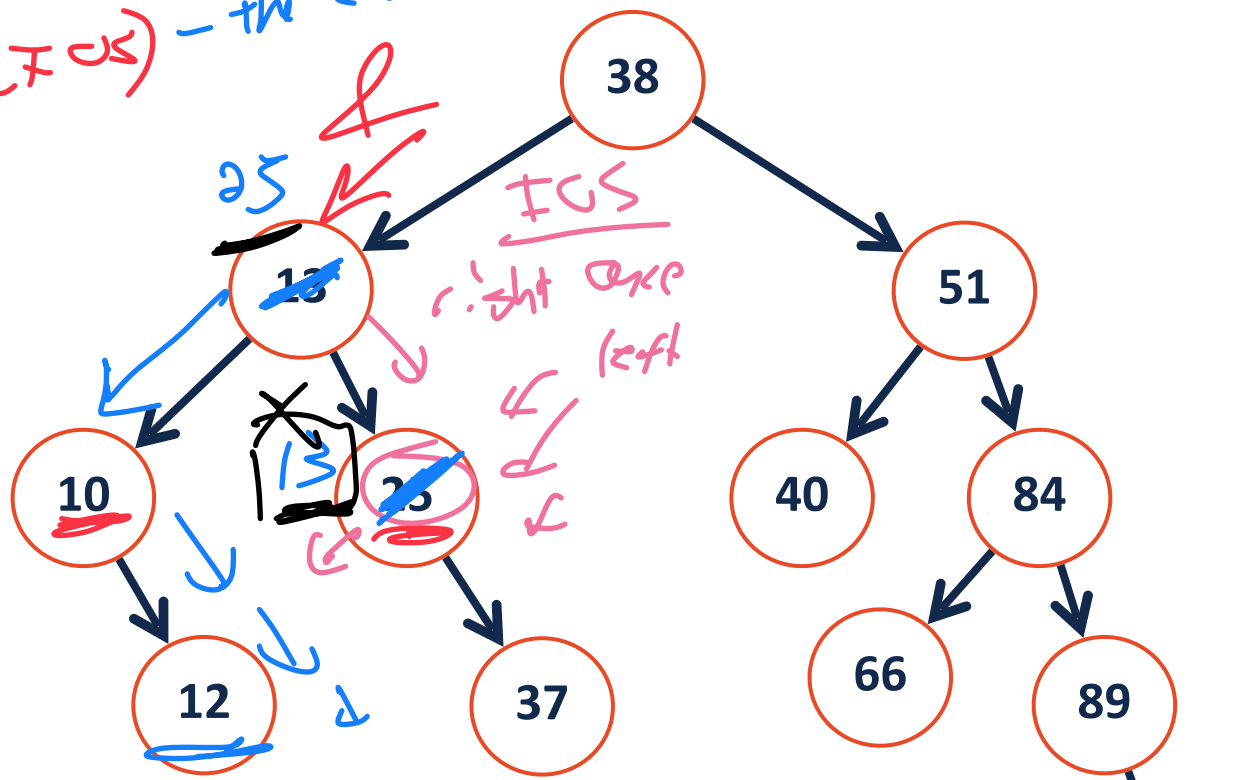
right most left child / go left once, right as much as poss

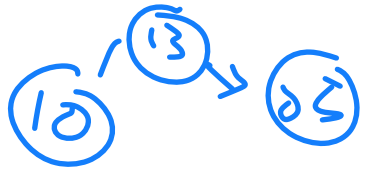
1) find (13) → tmp

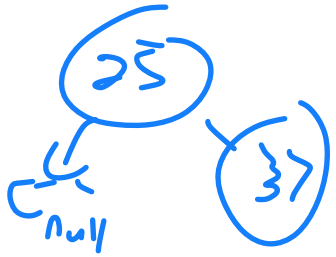
2) find
- In-order predecessor (IOP)
- In-order successor (IOS) – the closest larger

3) Swap tmp w/ IOP / IOS
- ↳ In the are swapped
- ↳ Swap either value (if easy)
- or
- Swap pointers

4) delete (13)
- ↳ This will always be 0 or 1 child remove



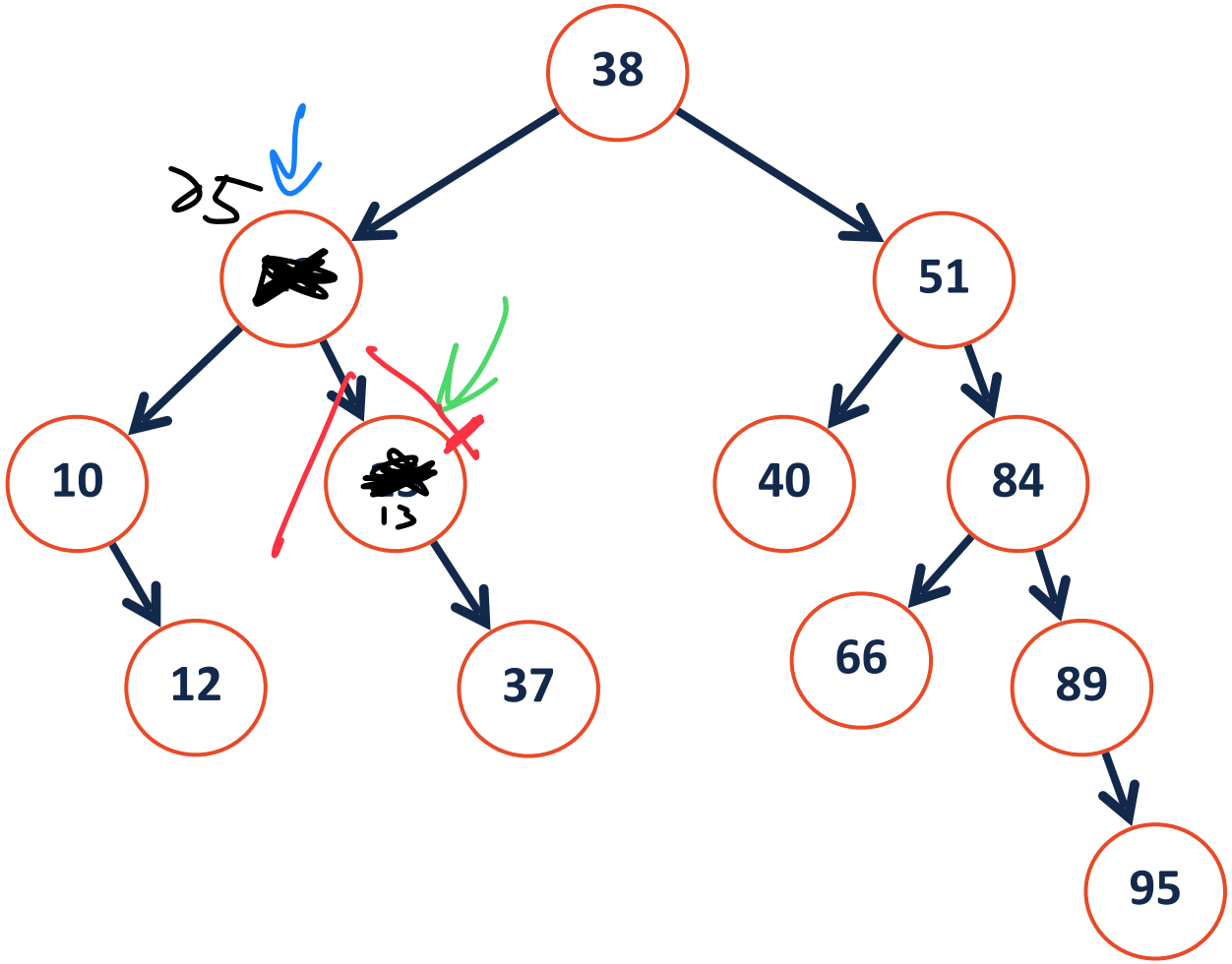




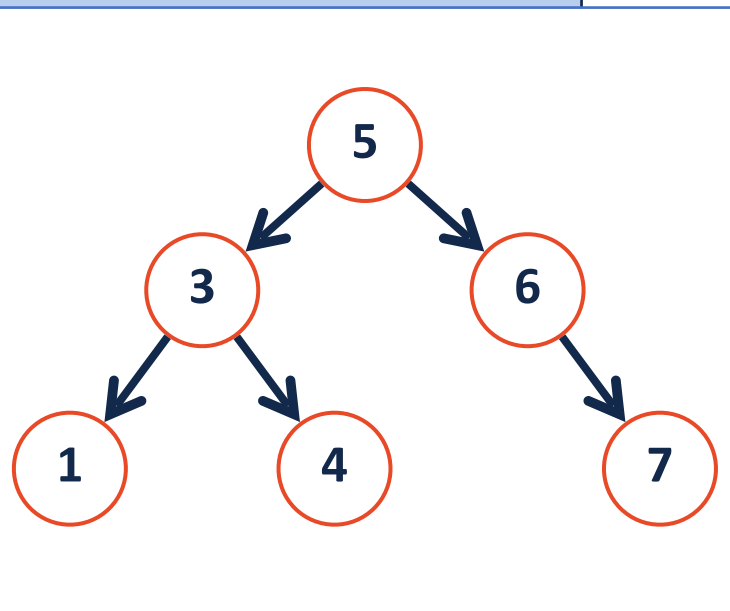
BST Remove

`remove(51)` → `remove(13)`

1) find (13)
2) find IOS
3) swap
4) remove (13)
   ↑ on my subtree



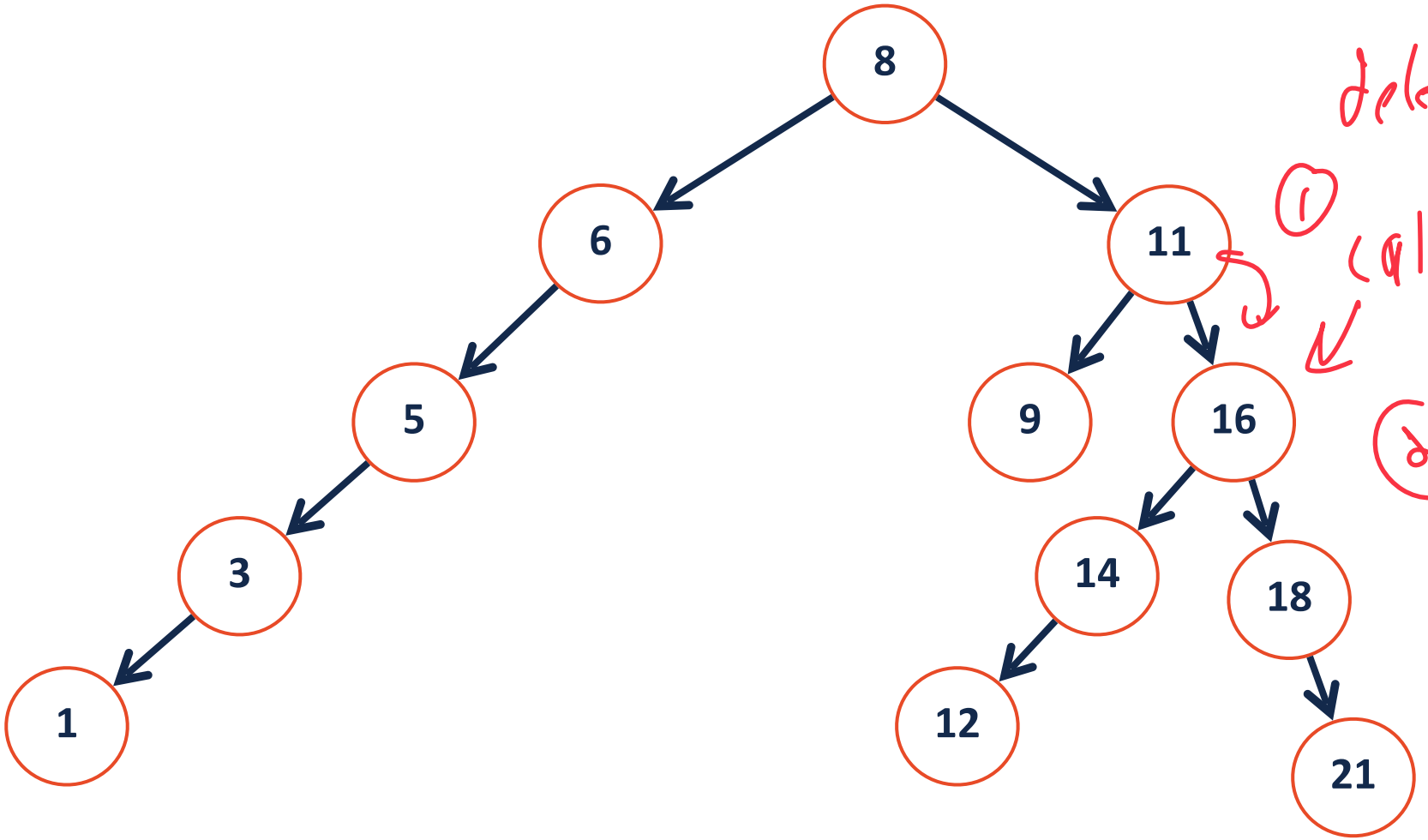
```
template<typename K, typename V>

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









}
```



BST Remove

Skipped => put on Friday!

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

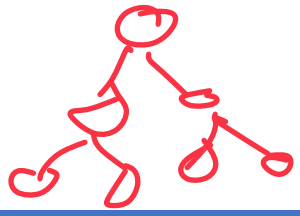


delete whole subtree

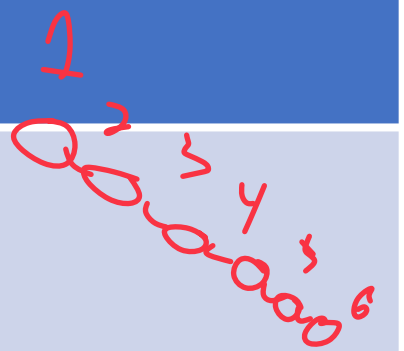
1) call Tree destructor on 16

2) set 11->right = null

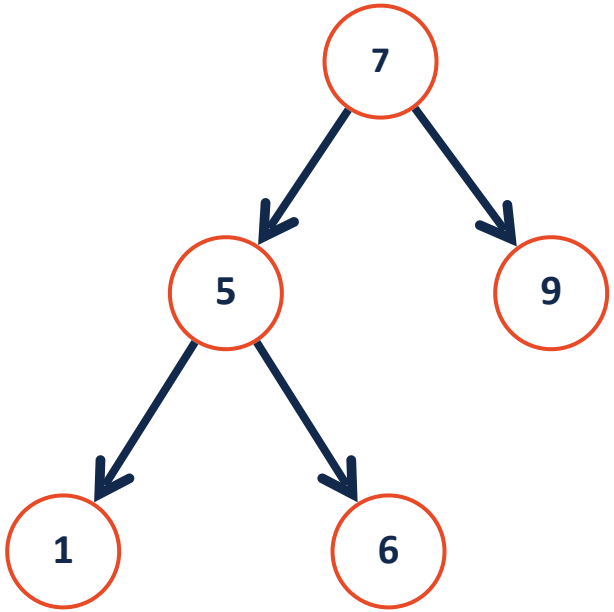
BST Analysis – Running Time

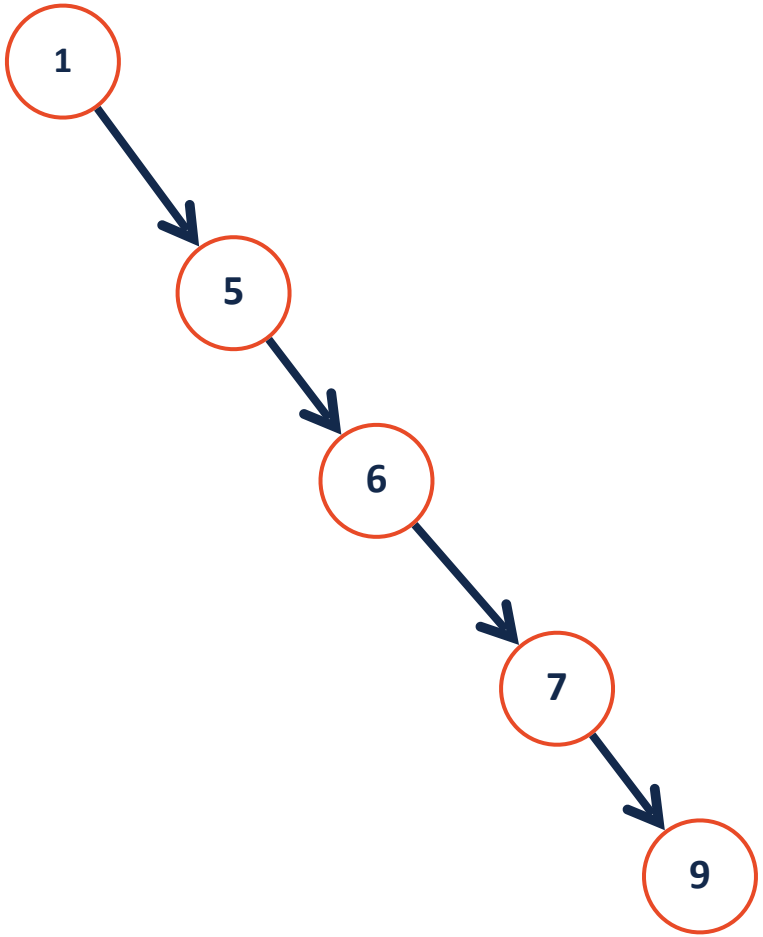


Operation	BST Worst Case
find	O(h) → where h is height h is O(n)
insert	find O(h) insert O(1) → O(h)
remove	find (O(h)) → remove obj + find (O(h)) → IOP/IOS + 1 O(h) ∴
traverse	O(n)



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!