

Data Structures

Binary Search Trees

CS 225

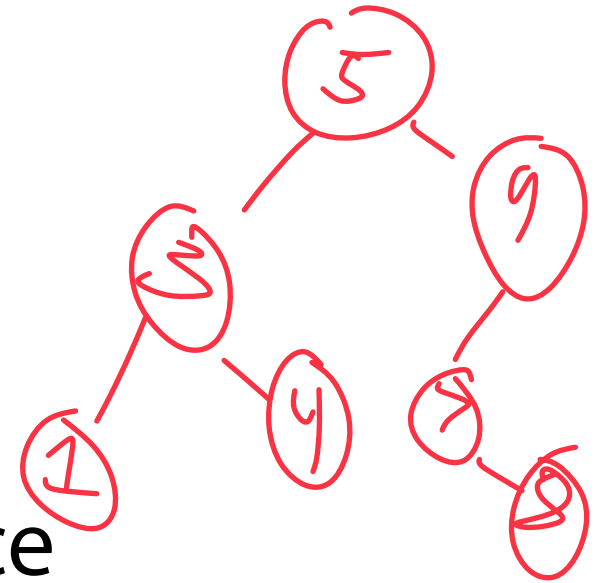
September 21, 2026

Brad Solomon



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science



THE GRAINGER COLLEGE OF ENGINEERING

**SEPTEMBER 23, 2026
430PM-6PM**

UNDERGRADUATE RESEARCH OPEN HOUSE

SIEBEL ATRIUM

TALK WITH FACULTY AND RESEARCH LAB REPS ABOUT OPEN UNDERGRADUATE RESEARCH OPPORTUNITIES IN THE SCHOOL

SIEBEL SCHOOL OF COMPUTING AND DATA SCIENCE



Exam 2 (9/30 — 10/02)

Autograded MC and one coding question

Manually graded short answer prompt

Practice exam will be released on PL on Wednesday

Topics covered can be found on website

Register now

<https://courses.engr.illinois.edu/cs225/exams/>

Exam 2 (9/30 — 10/02)

Lab for this week is **EXAM 2 REVIEW SESSION**

Strongly encouraged to attend! 

Extra credit lab releases this week (lab_trees)! 

 Extended deadline

Learning Objectives

Extend binary trees into binary search trees



Not a lot different

Build conceptual and coding understanding of BST



Introduce tree balance

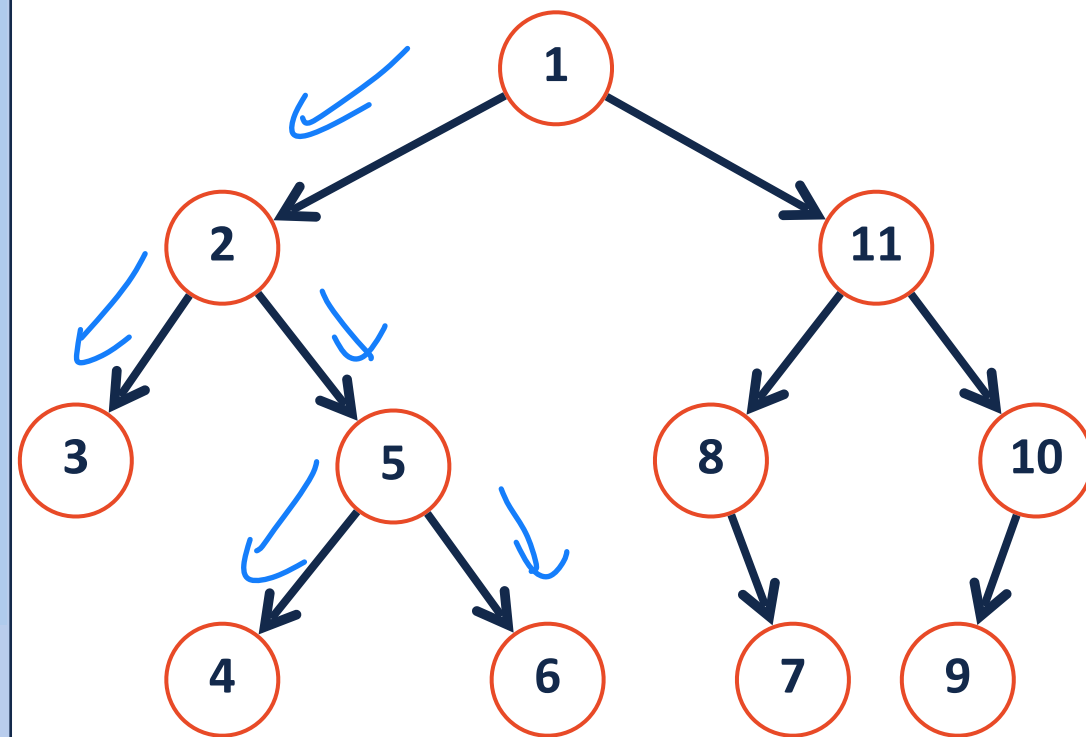


Depth First Search

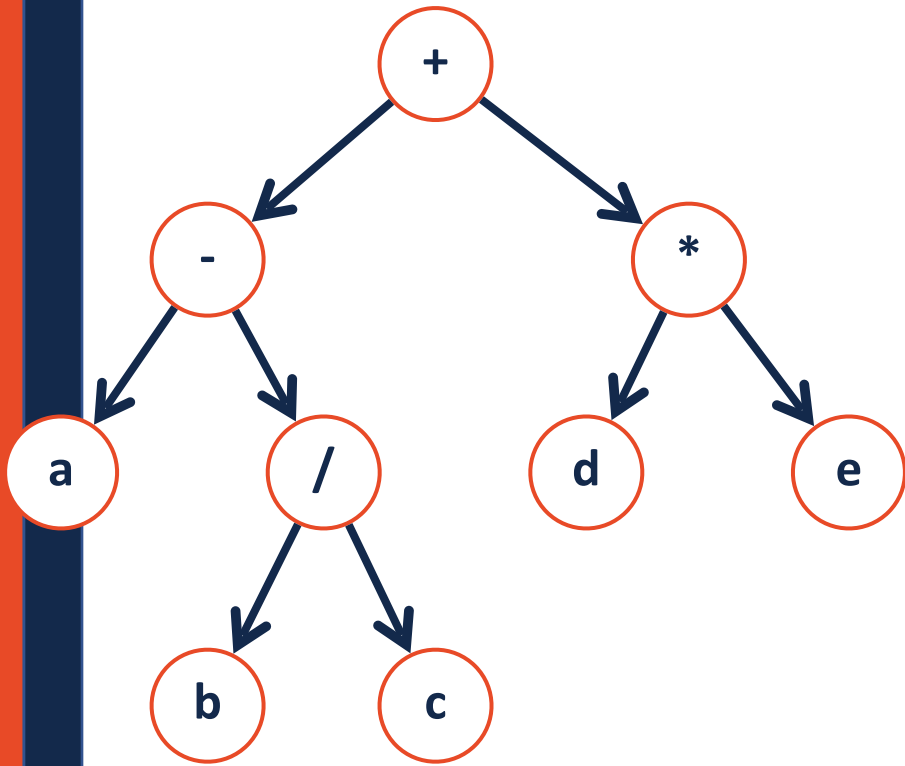
Search is traversal
 $O(n)$

Explore as far along one path as possible before backtracking

```
1 template<class T>
2 bool BinaryTree<T>::DFS_search(TreeNode*
3   root, T& query)
4   {
5       if (!root) {
6           return false;
7       }
8       return process(root->data, query) ||
9             DFS_search(root->left, query) ||
10            DFS_search(root->right, query);
11   }
```



Breadth First Search $\rightarrow$ Level order traversal $O(n)$



```
1 template<class T>
2 bool BinaryTree<T>::BFS_search(TreeNode* root, T&
  query)
3 {
4     if (!root) return false;
5
6     std::queue<TreeNode*> q;
7     q.push(root); // enqueue == push
8
9     while (!q.empty()) {
10         TreeNode* temp = q.front();
11         q.dequeue();
12
13         if (proc(temp->data, query)) return true;
14
15         if (temp->left) q.enqueue(temp->left);
16         if (temp->right) q.enqueue(temp->right);
17     }
18
19     return false;
20 }
```

What search algorithm is best?

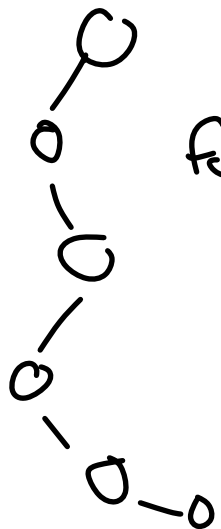
Join Code: 225



The average 'branch factor' for a game of chess is ~ 31 . If you were searching a decision tree for chess, which search algorithm would you use?

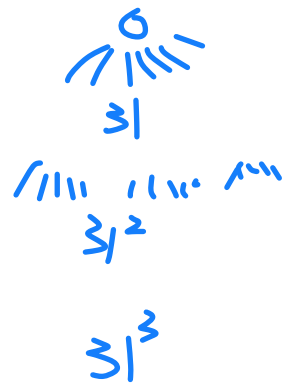
35% DFS / 55% BFS / 10% naive

Doesn't quite work

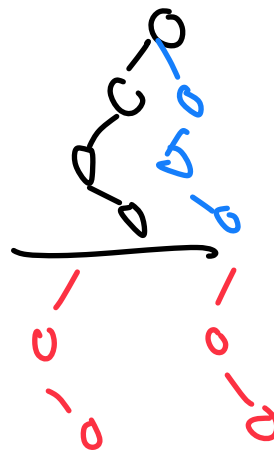


Follows largest path

if no tie end clause

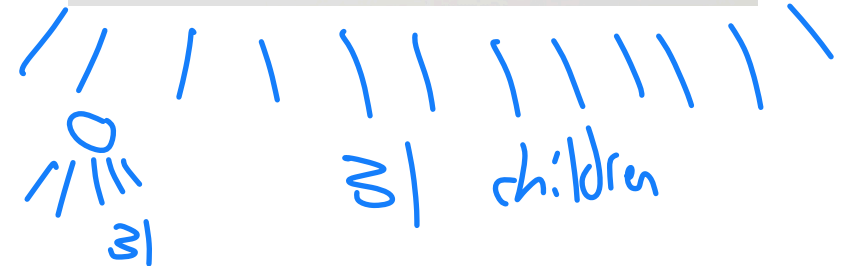


Iterative deepening DFS



DFS to depth limit

increase depth as needed



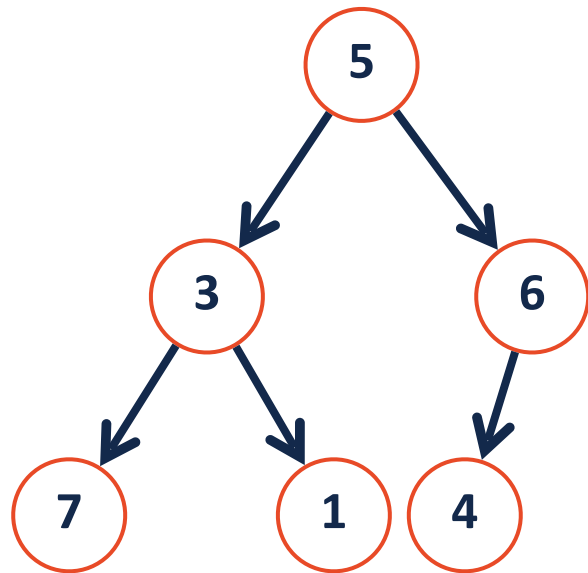
31 children

Trivial Point! ✓

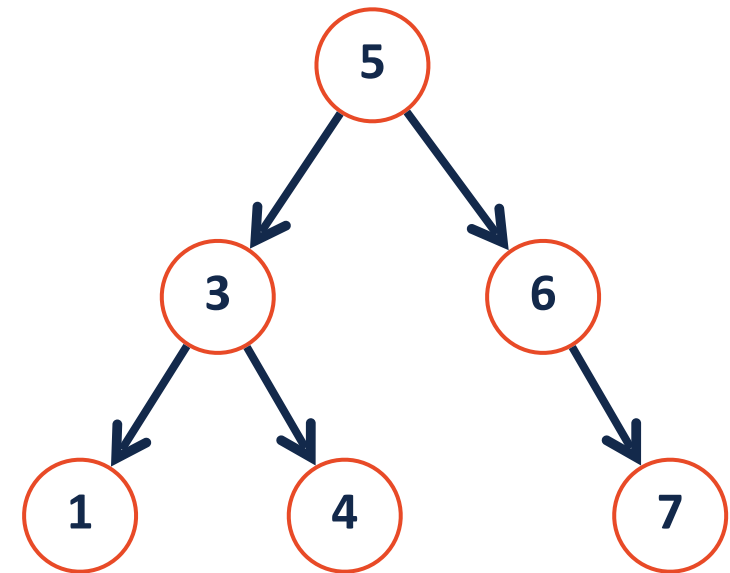
Improved search on a binary tree



Binary Tree



Binary Search Tree

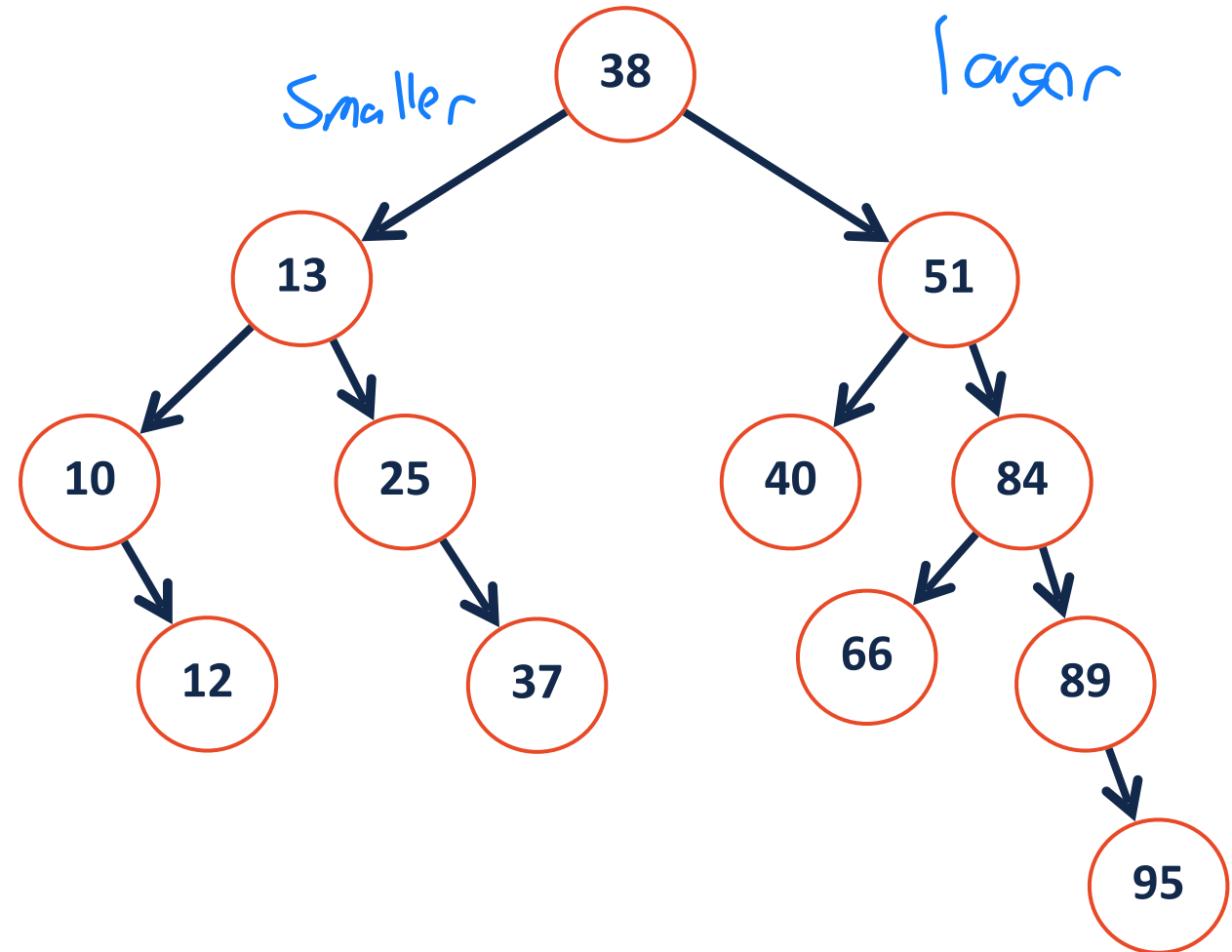


Binary Search Tree (BST)

A **BST** is a binary tree $T = \text{TreeNode}(\text{root}, T_L, T_r)$ such that:

$\forall n \in T_L, n.\text{key} < \text{root}.\text{key}$

$\forall n \in T_R, n.\text{key} > \text{root}.\text{key}$



BST.h

```
1  #pragma once
2
3  template <typename K, typename V>
4  class BST {
5      public:
6          /* ... */
7      private:
8          class TreeNode {
9              K & key;
10             V & value;
11
12             TreeNode *left, *right;
13
14             TreeNode(K & k, V & v) :
15 key(k), value(v),
16 left(NULL), right(NULL) { }
17         };
18
19         TreeNode *root_;
20         /* ... */
21     };
22
23
```

Tree.h

```
1  #pragma once
2
3  template <typename T>
4  class BinaryTree {
5      public:
6          /* ... */
7      private:
8          class TreeNode {
9              T & data;
10
11             TreeNode * left;
12
13             TreeNode * right;
14
15             TreeNode(T & data) :
16 data(data), left(NULL),
17 right(NULL) { }
18         };
19
20         TreeNode *root_;
21         /* ... */
22     };
23
```

Binary Search Tree ADT



Insert

Remove

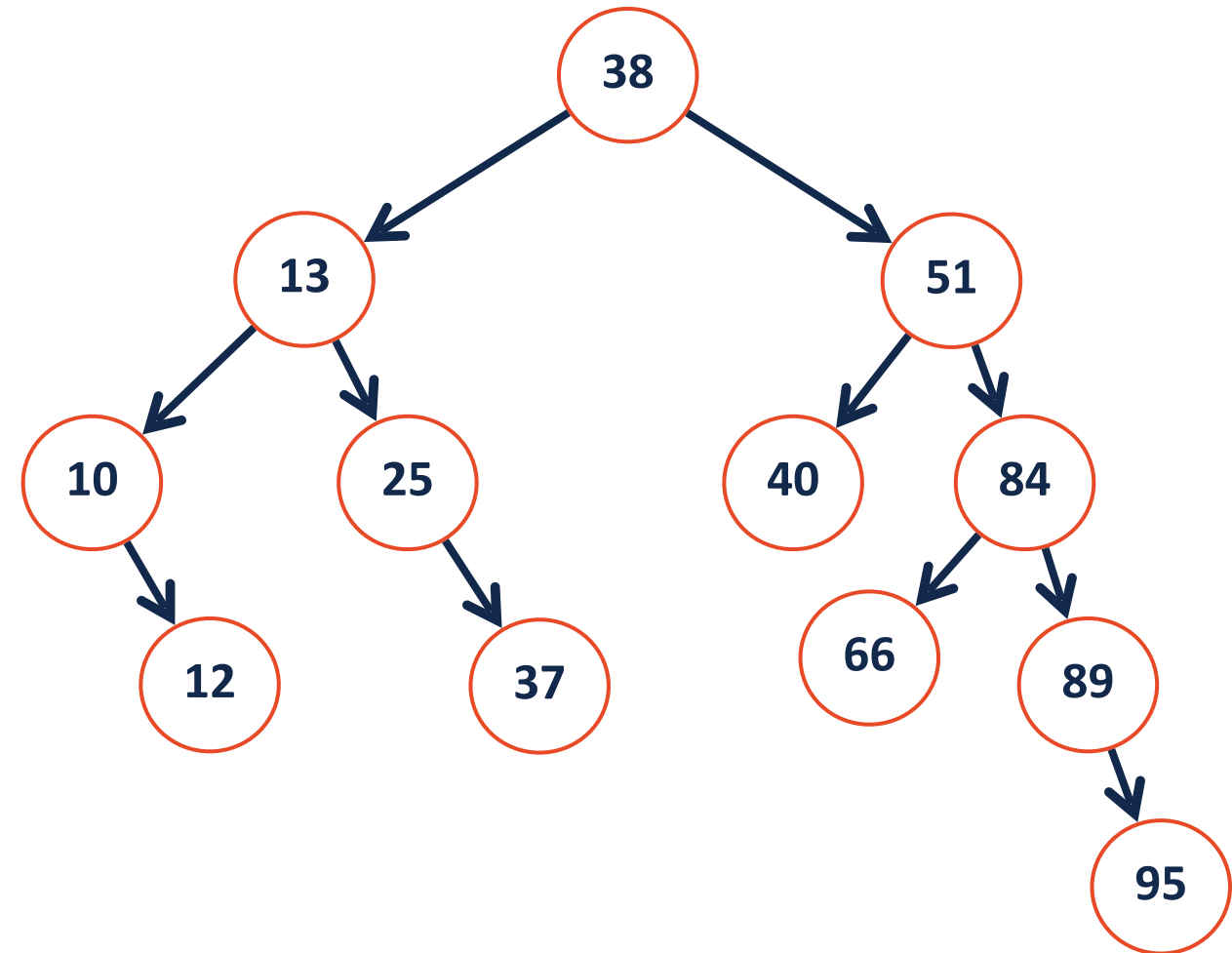
Traverse

Find

Constructor

BST Find

find(66)



BST Find

A recursive function based around value of root:

Base Case: If root is null, return root

Let tmp = root->key()

tmp == query, return root

Recursion: recurse to only one child

tmp < query, recurse right

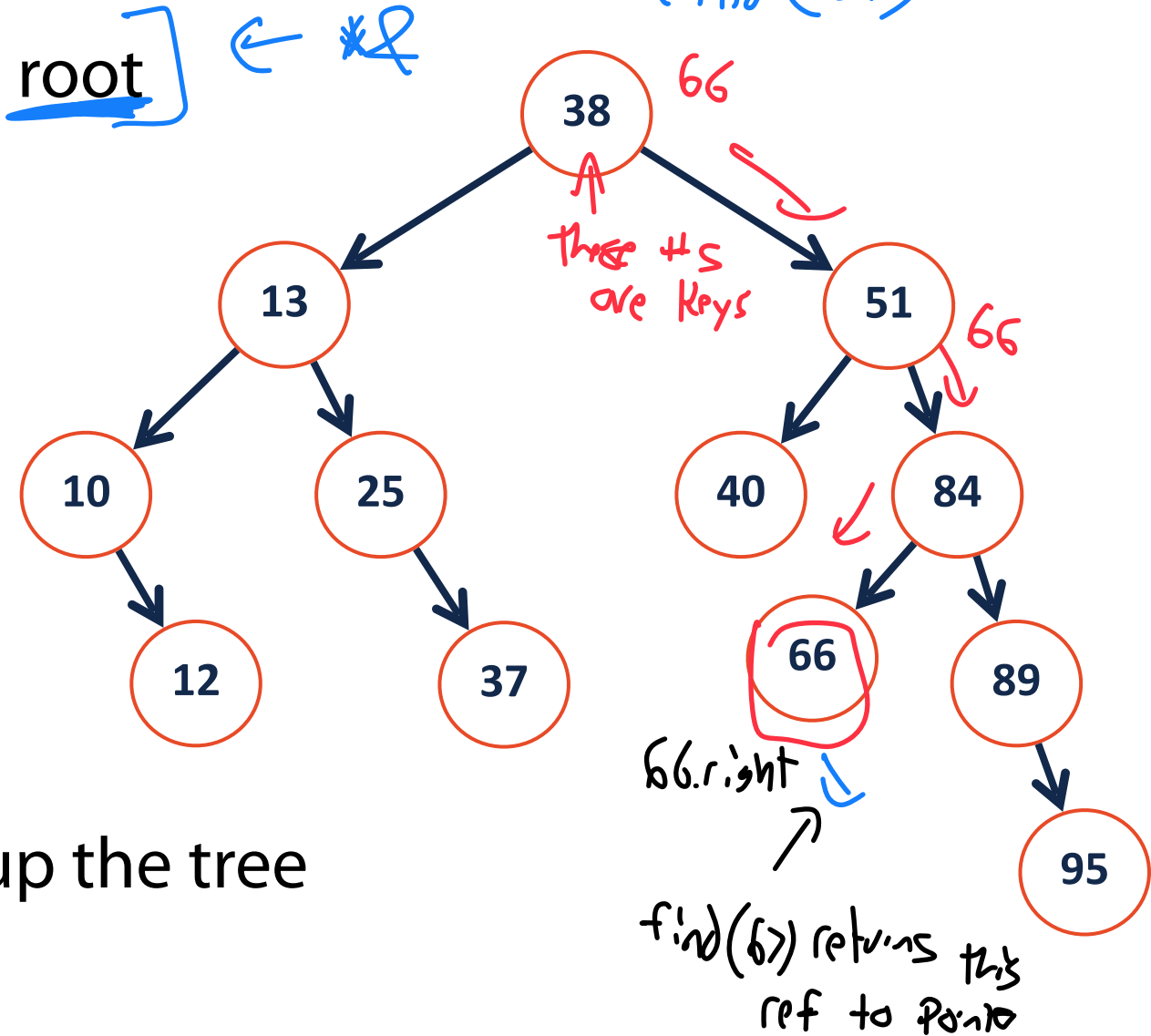
tmp > query, recurse left

Combining:

Return the recursive value back up the tree

find(66)

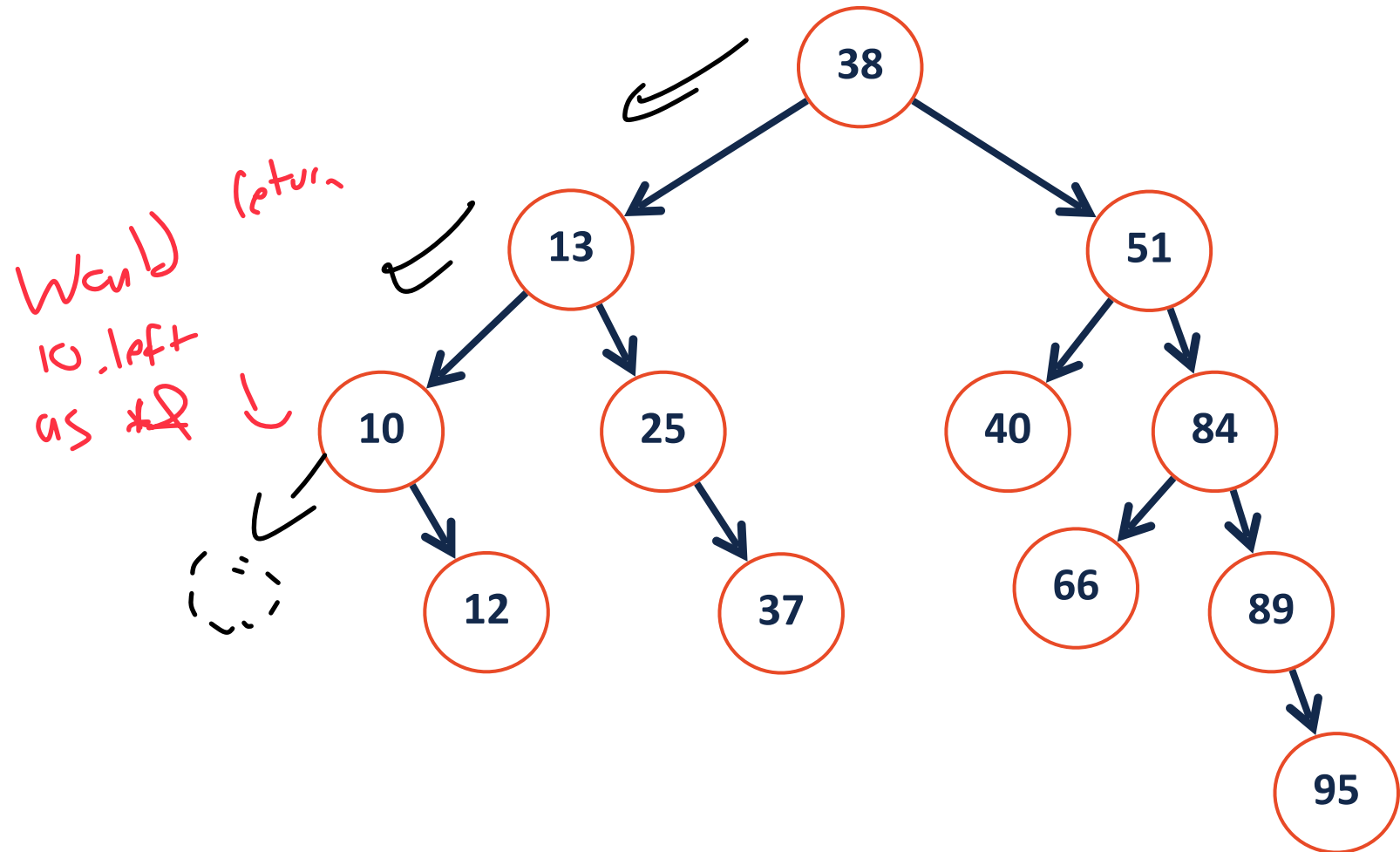
find(67)



BST Find

Make sure you can follow the search path!

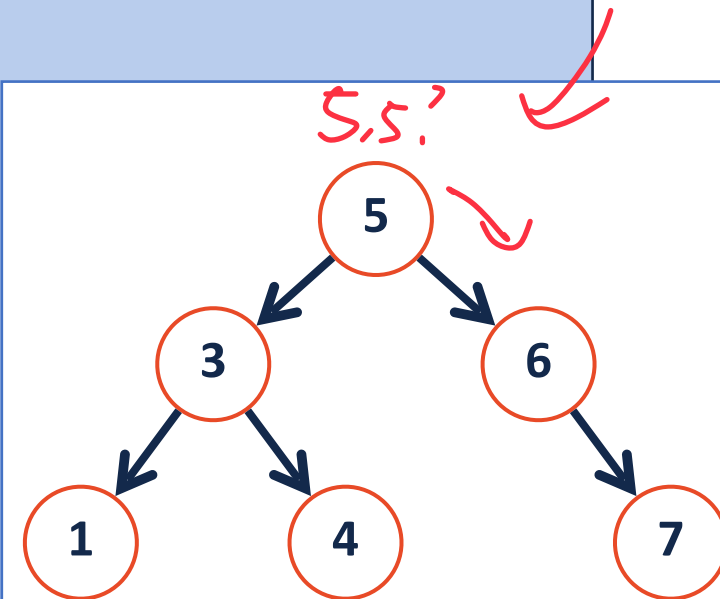
find(9)



```

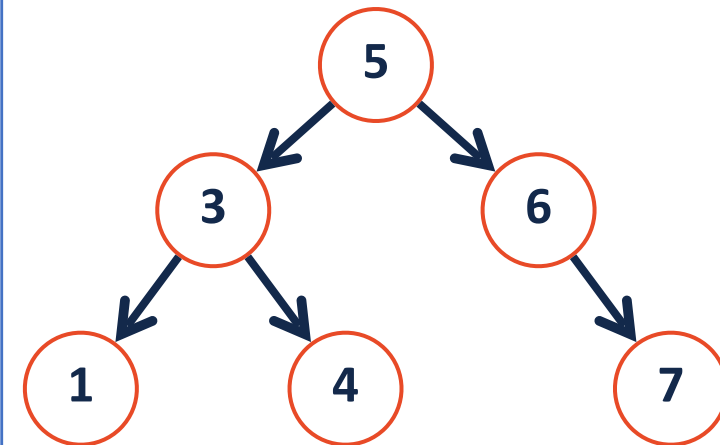
1 template<typename K, typename V>
2   TreeNode & __find(TreeNode *& root, const K & key) {
3   // Base cases
4   if (root == nullptr || root->key == key) {
5       return root;
6   }
7   // Recursive Step
8   if (root->key > key) {
9       return __find(root->left, key);
10  }
11  return __find(root->right, key);
12  }

```





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



BST Insert

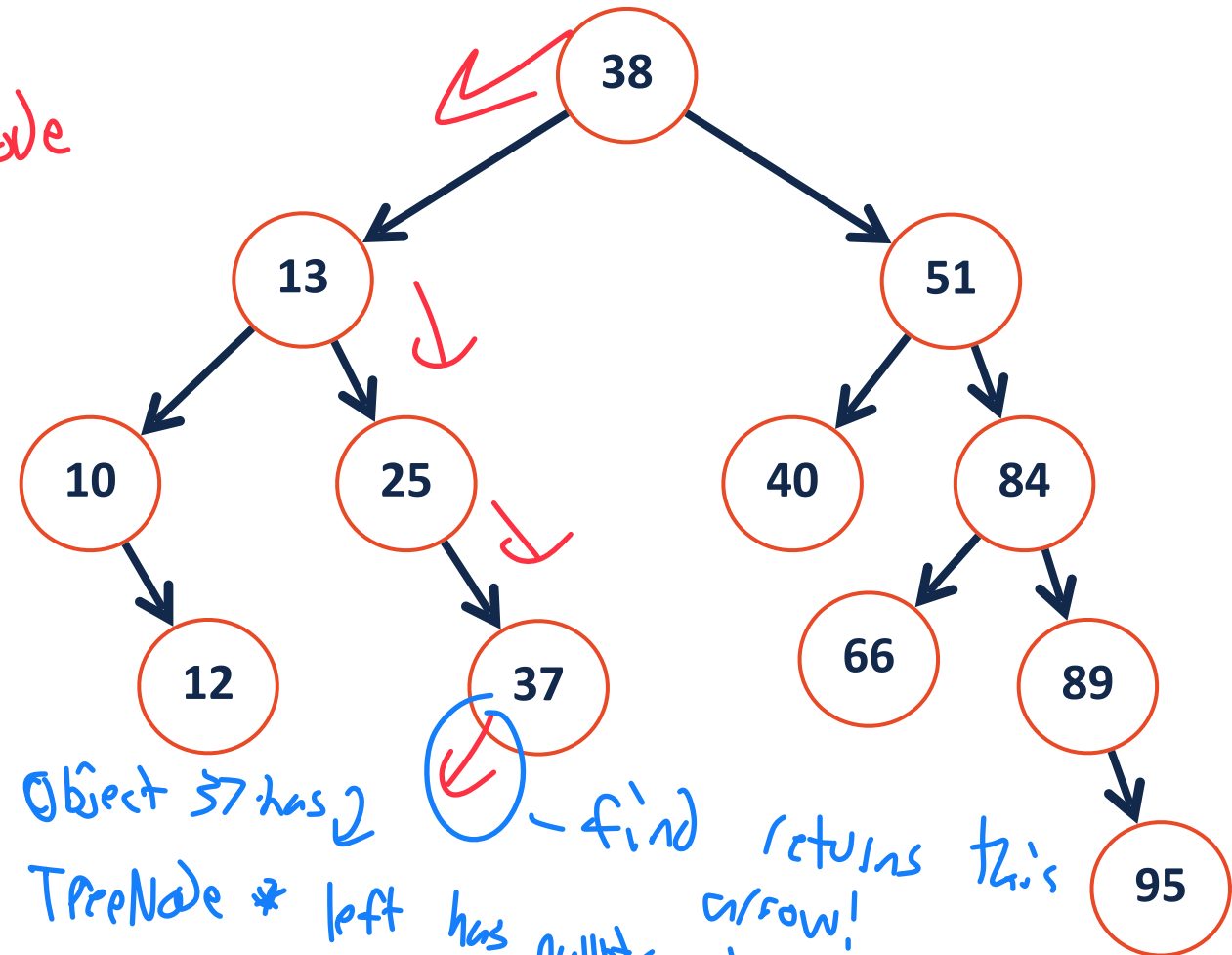
insert(33, v)
key value

1) Find where 33 should go

$tmp = \text{find}(33);$

2) Add (33, v) as new node

$tmp = \text{new TreeNode}(33, v);$



Object 37 has left - find returns this
TreeNode * left has null value

BST Insert

insert(15, v)

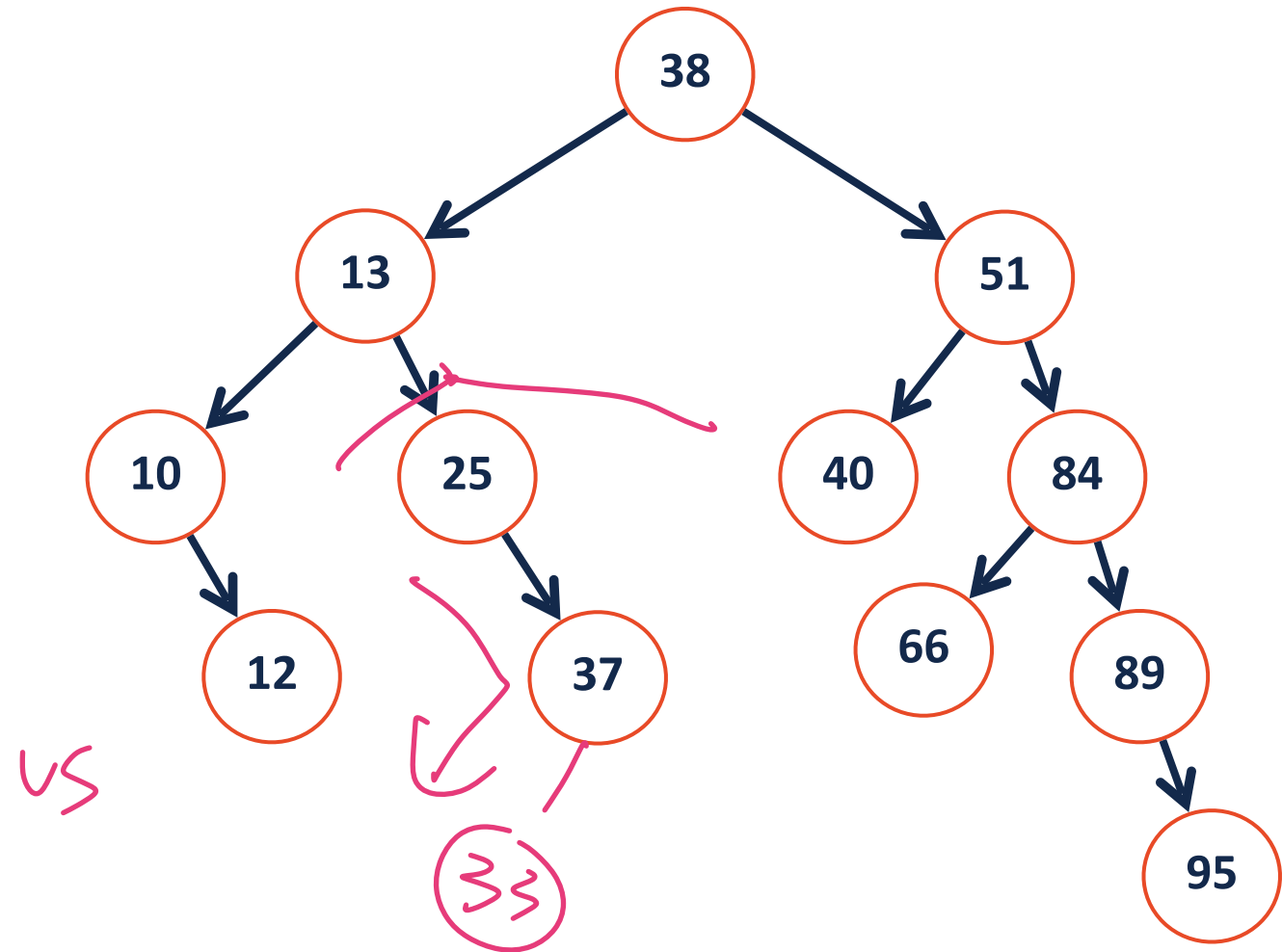
Find the insert location using `_find()`

Trivially insert at location

why not re make tree as

25 >
-33 -37

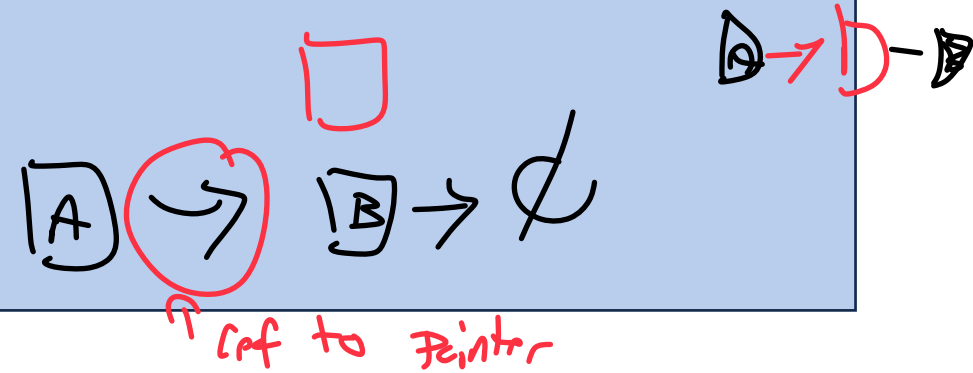
Next week:



```

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
6
7
8
9
10
11
12
13
14
15
16 }

```

"Is this new Node linked properly?"

t = find(k)

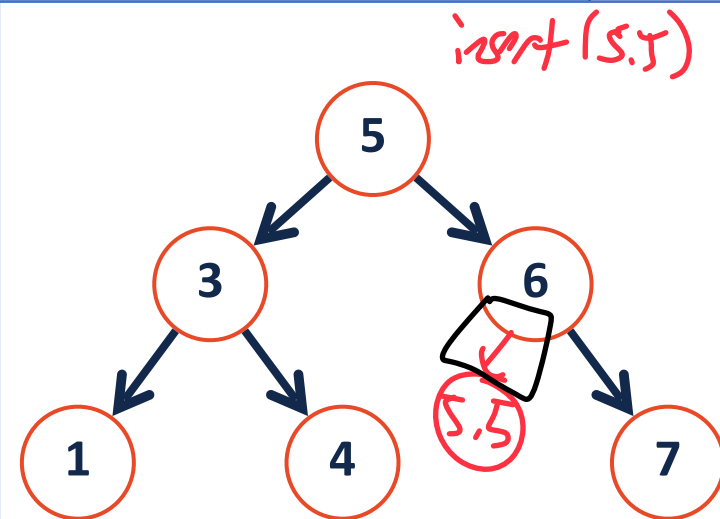
t = new Node(k, v);

← Not nullptr but an existing pointer which has nullptr

← had value nullptr, now has value

(TreeNode *)

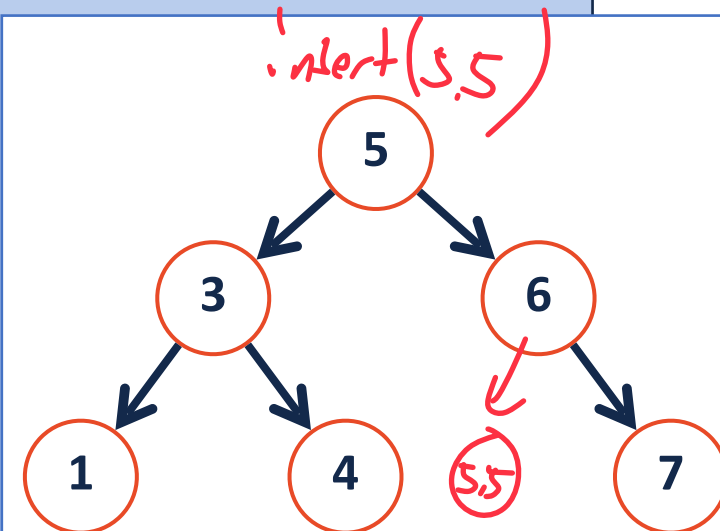
0x10
↑
new heap
address





```
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
```

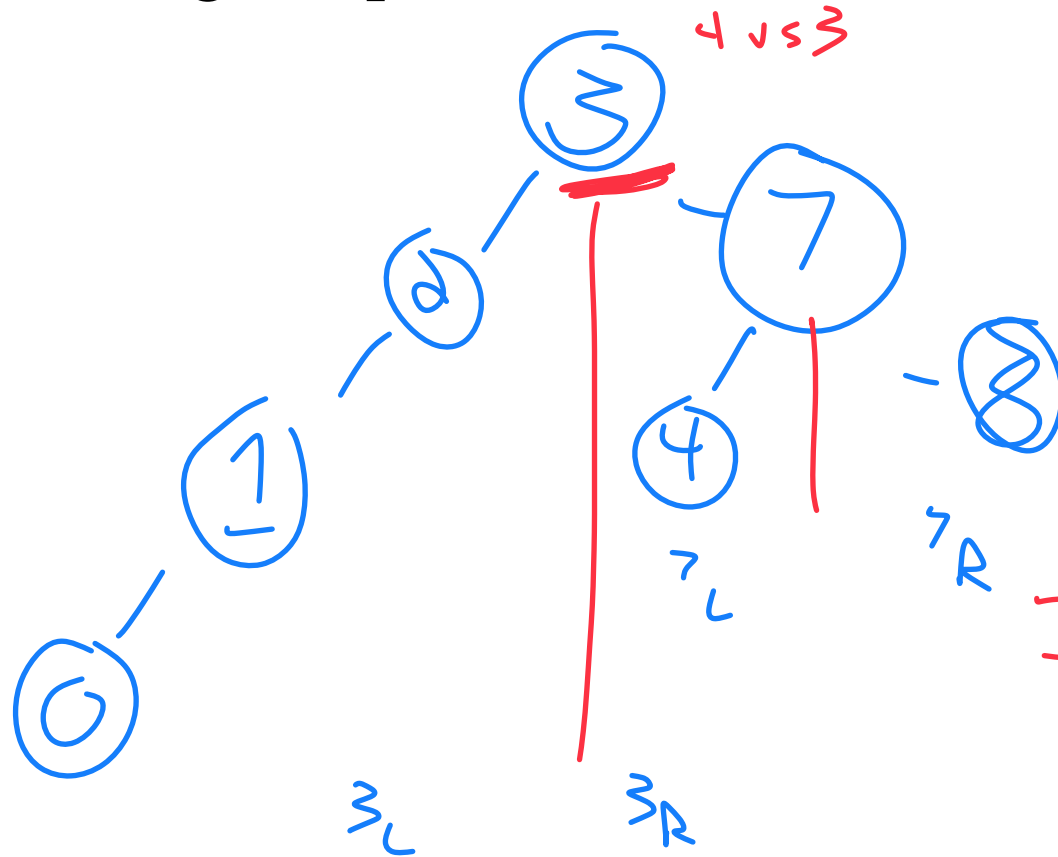
```
1 template<typename K, typename V>
2
3 void _insert(TreeNode *& root, const K & key, const V & val) {
4
5     TreeNode *& tmp = _find(root, key);
6
7
8     tmp = new treeNode(key, val);
9
10
11
12
13 }
14
15
16
```



BST Insert

Tip for exam 2

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



Import trick
→ Order matters!
(lab_bst)

BST Remove

What should our tree look like after the following...

remove (40)

find(40)

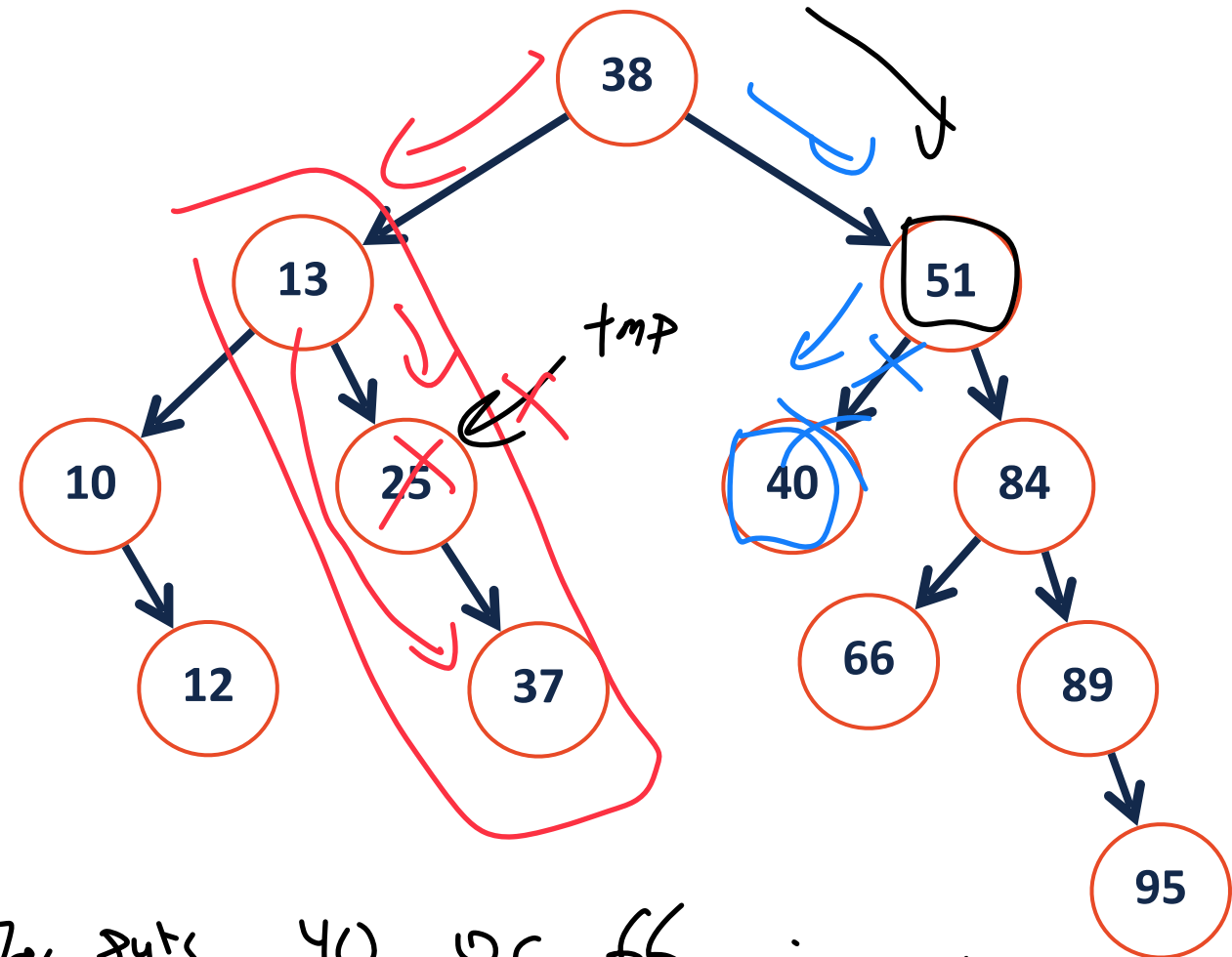
remove it

remove (25)

Linked List removal

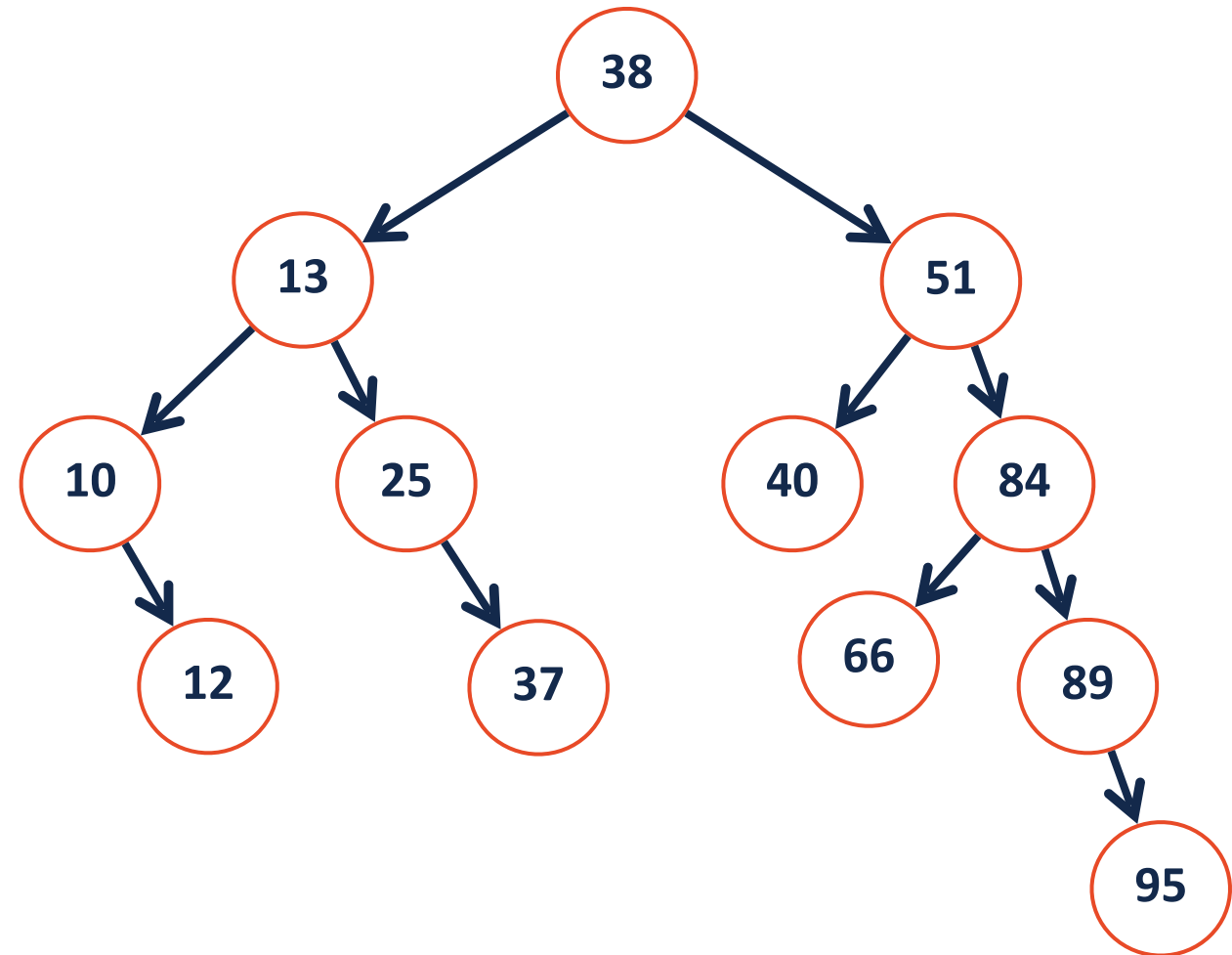
remove (51)

remove 51 either puts 40 or 66 in root position



BST Remove

remove (40)



BST Remove

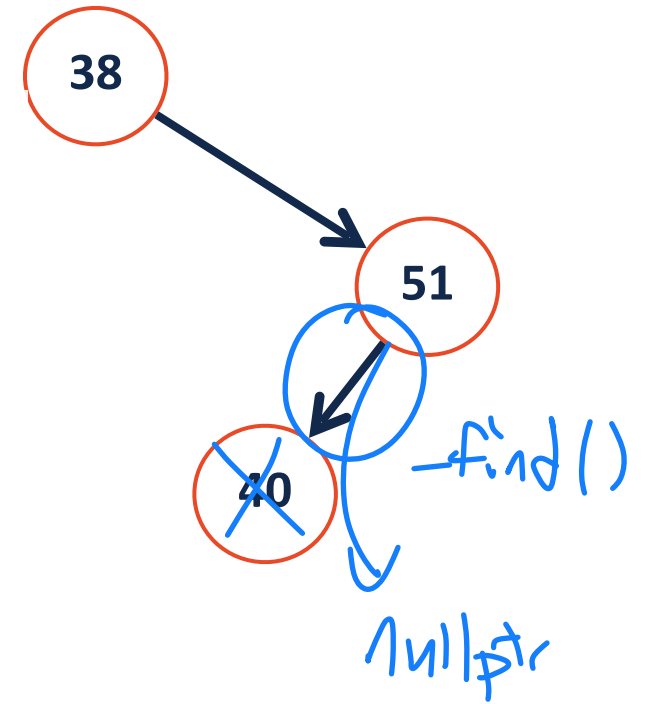
0-Child Case (node being removed has zero children)

```
TreeNode *& t = find(root, 40);
```

```
delete t;
```

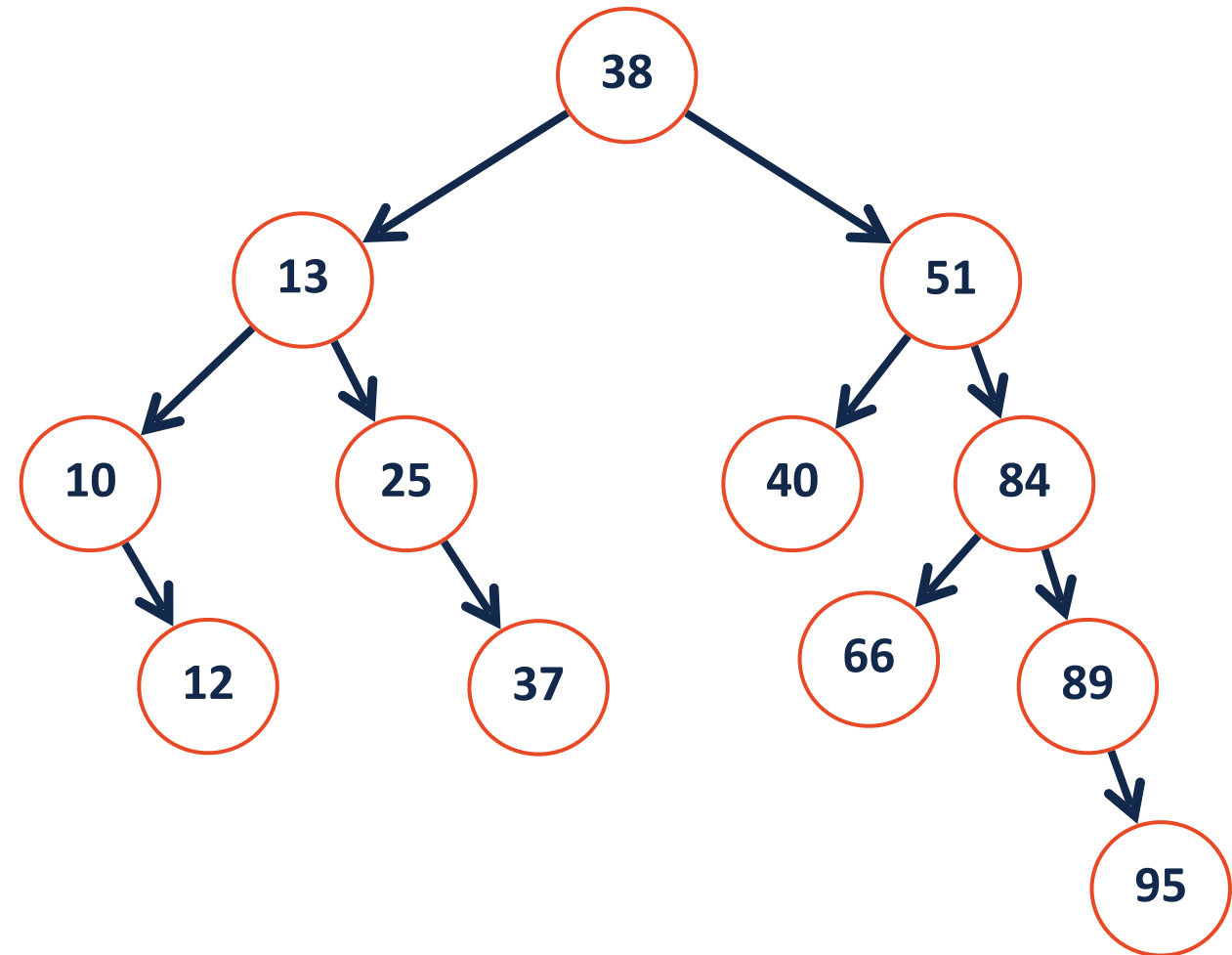
```
t = nullptr;
```

remove(40)



BST Remove

remove (25)



BST Remove

remove (25)

1-Child Case

```
TreeNode *& t = _find(root, 25);
```

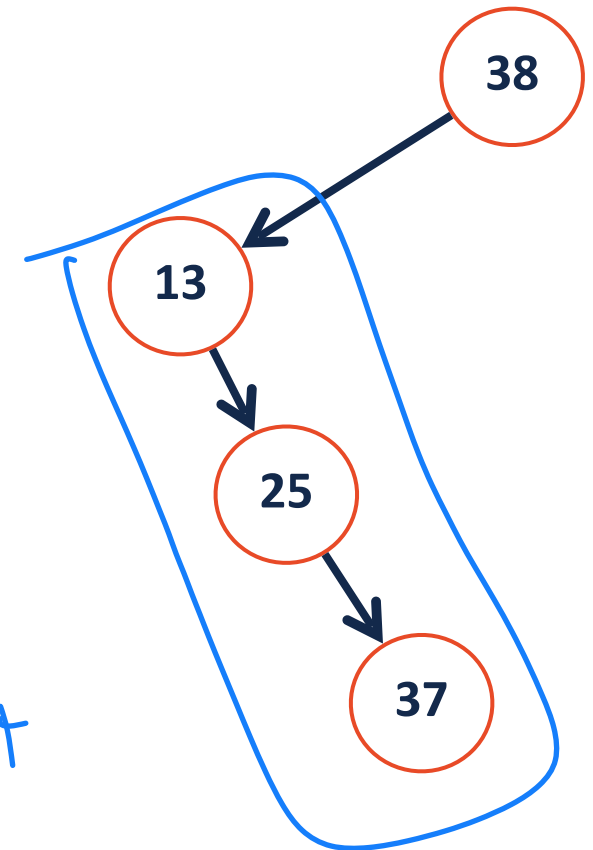
```
TreeNode * tmp = t;
```

```
t = t->right;
```

```
delete tmp;
```

could be $t \rightarrow \text{left}$ or $t \rightarrow \text{right}$

Linked List (removal)



BST Remove

remove (13)

1) find Node to be removed

(recognize two children)

2) Swap node being removed with another val

How to get IOP / IOS?

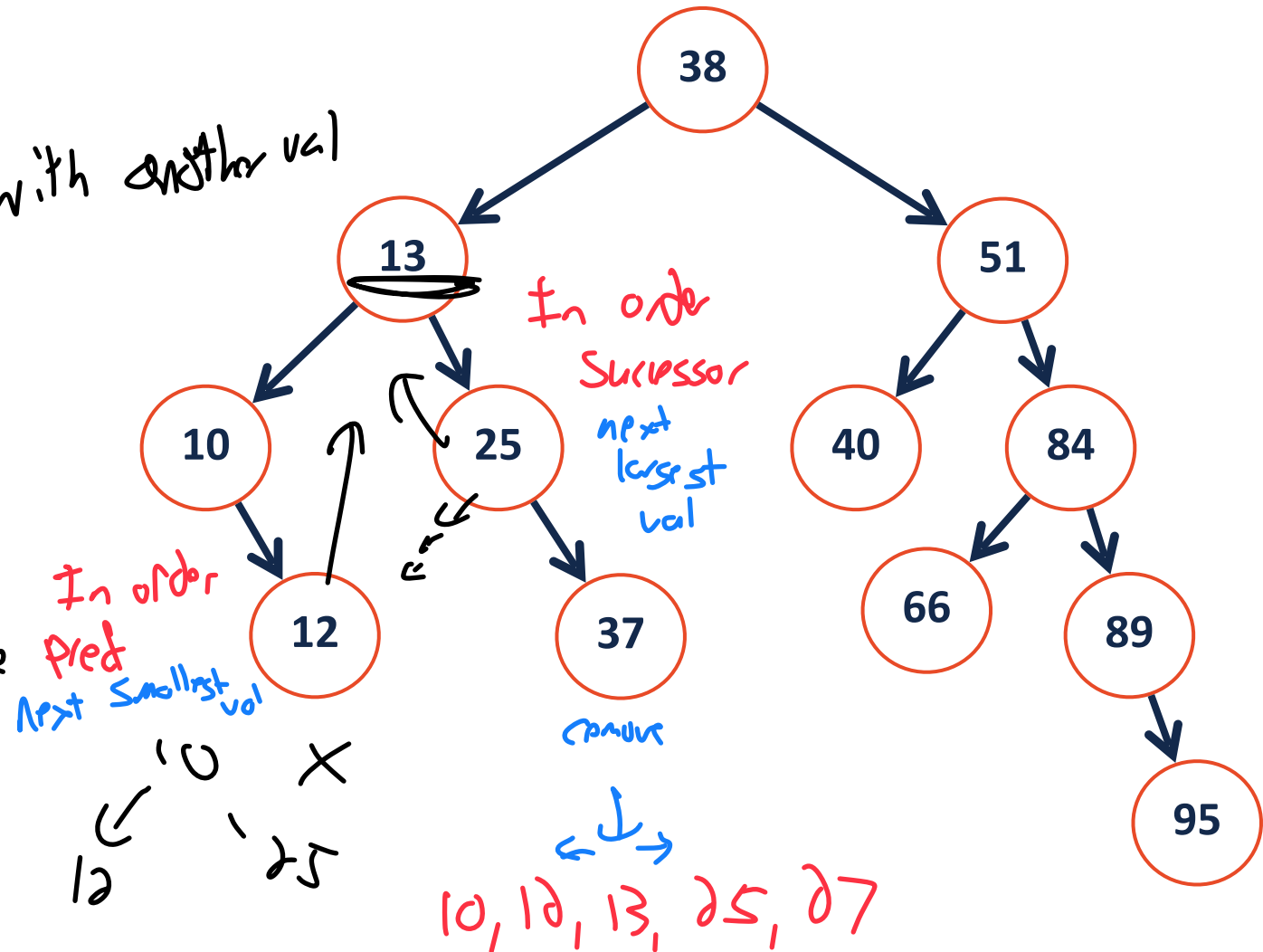
Given root (object being deleted)

IOP: Go left once

Go right as much as possible

IOS: Go right once

Go left as much as possible





BST In-Order _____

In-Order Predecessor

Rightmost left child

$$\text{IOP}(38) = 37$$

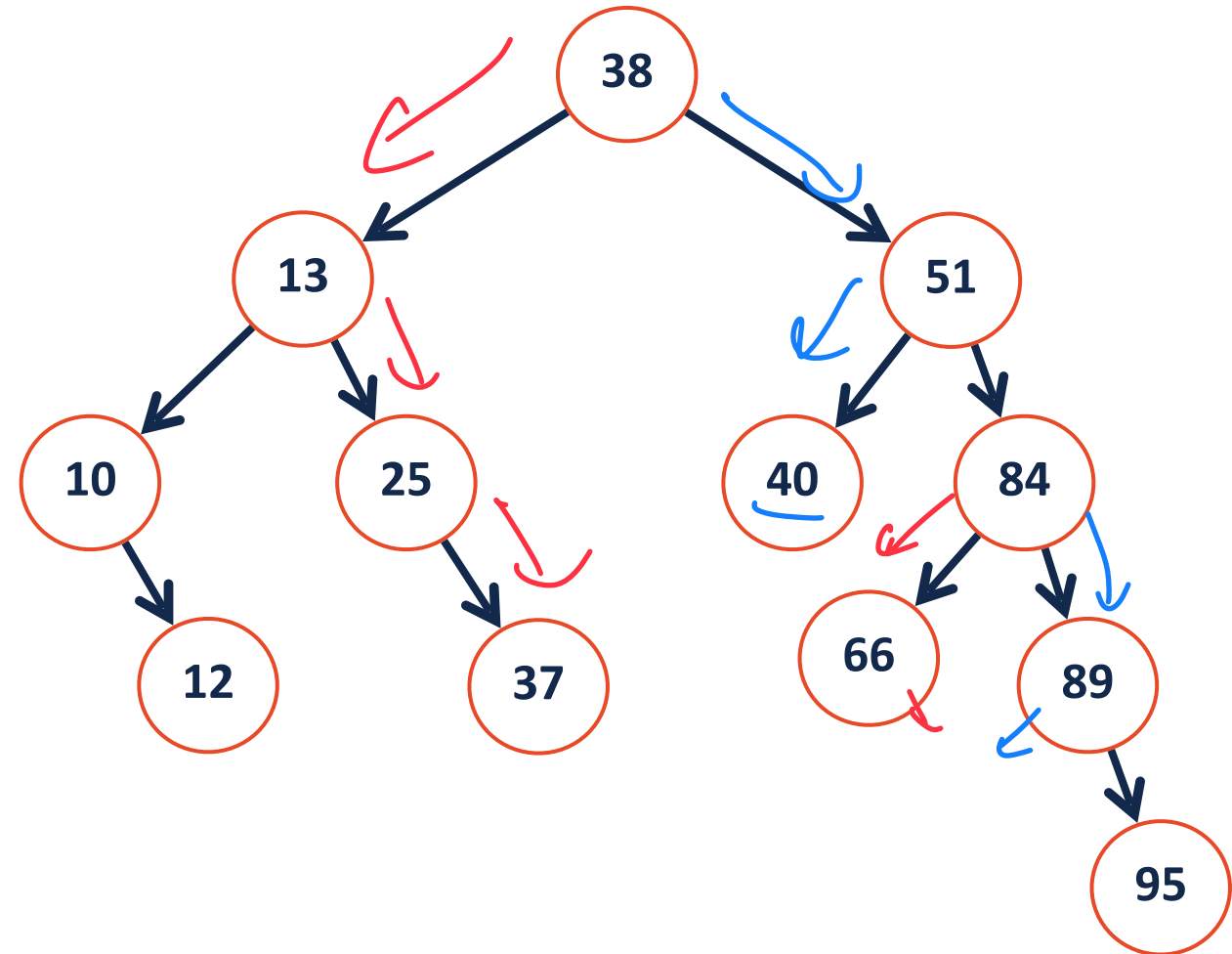
$$\text{IOP}(84) = 66$$

In-Order Successor

Leftmost right child

$$\text{IOS}(38) = 40$$

$$\text{IOS}(84) = 89$$



BST Remove

remove(13) 

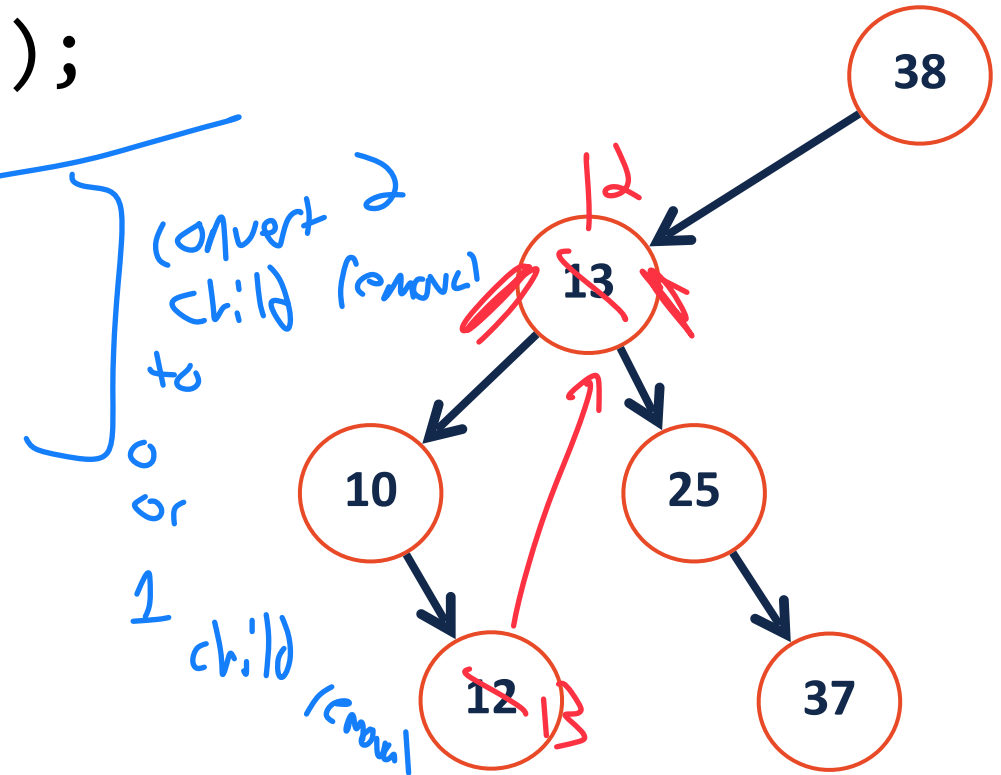
2-Child Case

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

```
TreeNode * IOP = getIOP(t);
```

```
swap(t, iop);
```

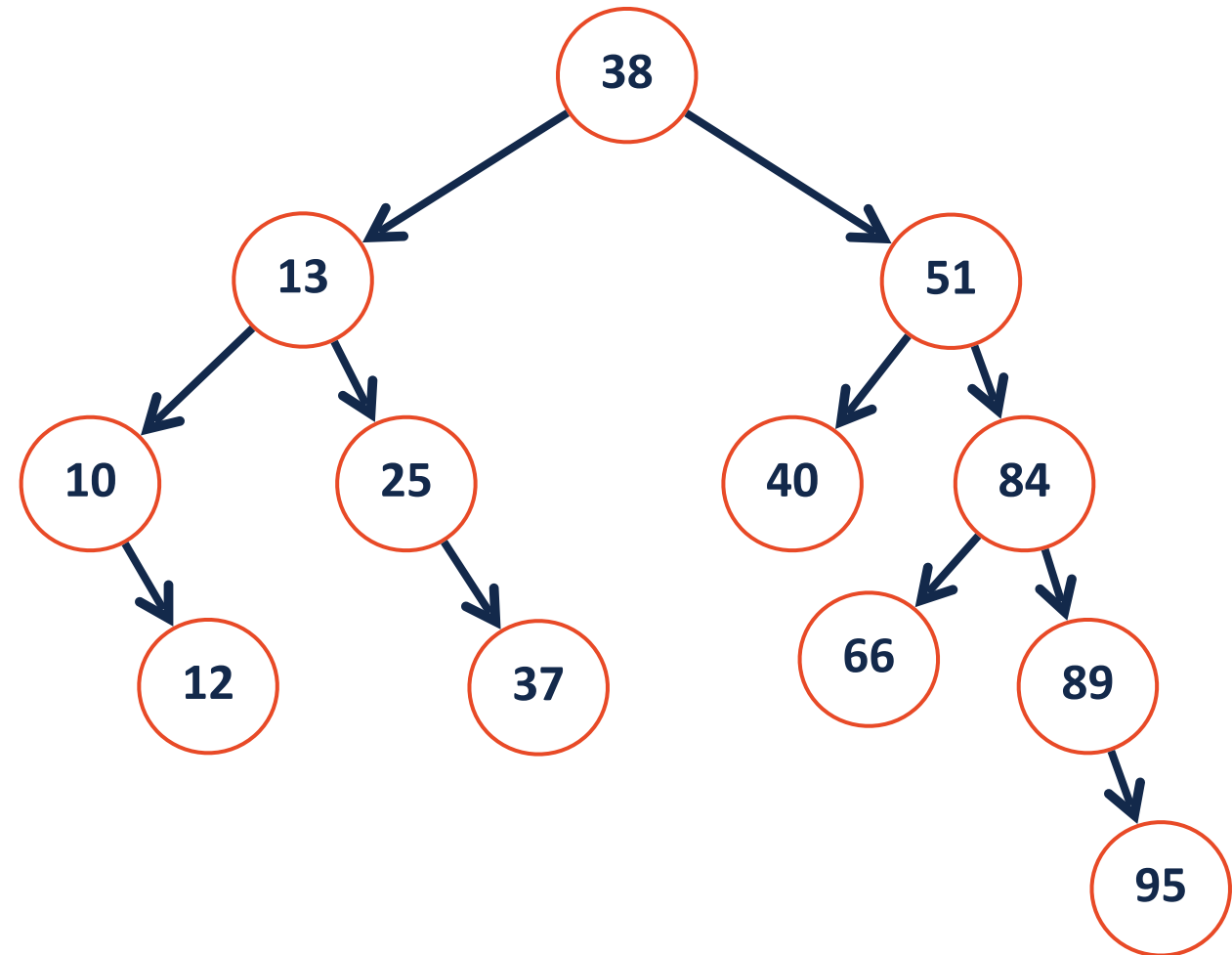
```
remove(13); //starting from t
```



why is IOP / IOS always 0 or 1 child remove?

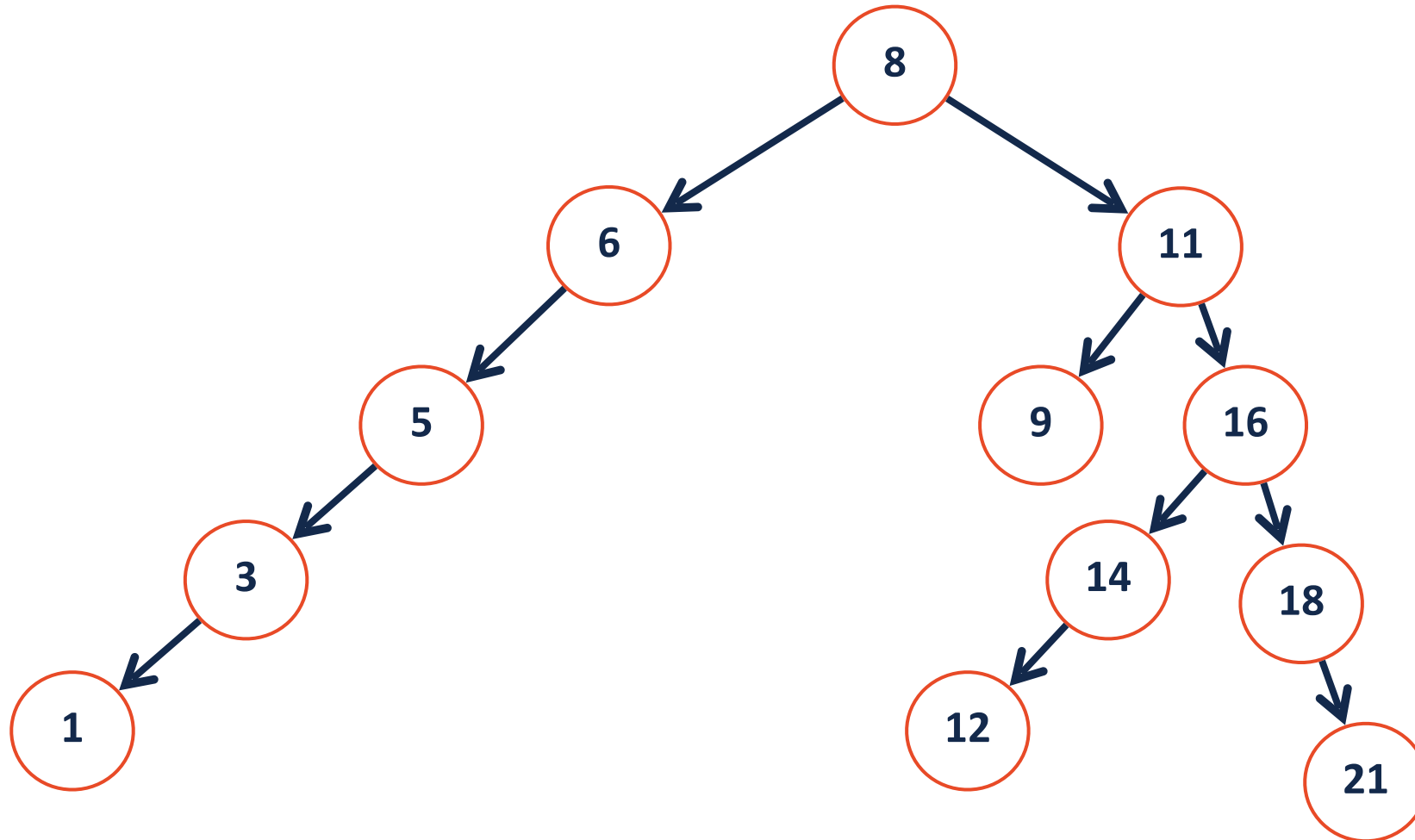
BST Remove

remove (51)



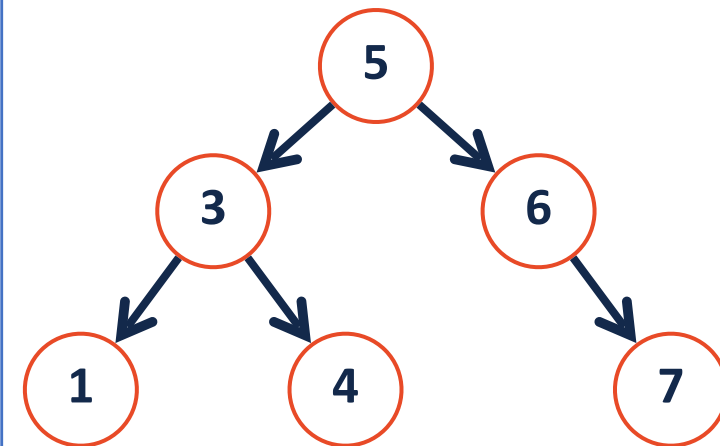
BST Remove

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



```
1 template<typename K, typename V>
2
3 void _remove(TreeNode *& root, const K & key) {
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23 }
```

Lab - BST



BST Analysis – Running Time



Operation	BST Worst Case	
find	$O(h)$	what is bound on height
insert	$O(h)$	
remove	$O(h)$	
traverse		

BST Analysis

Every operation on a BST depends on the **height** of the tree.

... how do we relate $O(h)$ to n , the size of our dataset?

BST Analysis

Join Code: 225



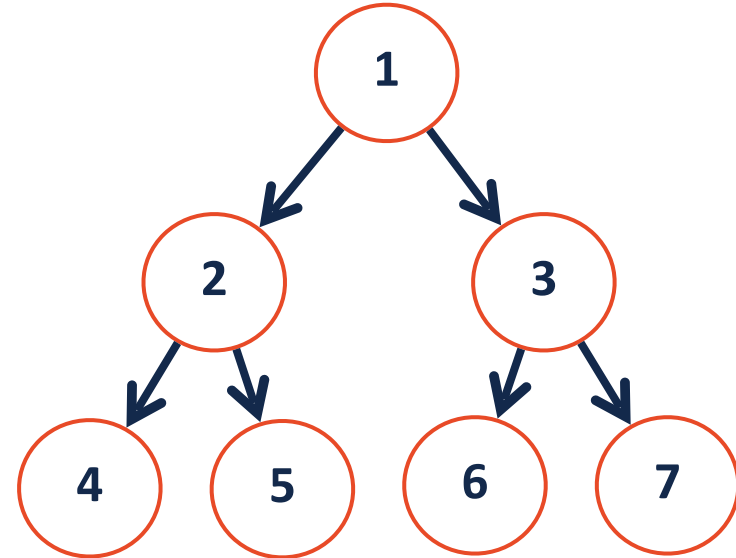
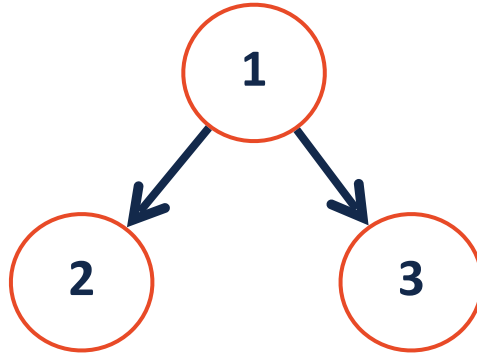
What is the **max** number of nodes in a tree of height h ?

BST Analysis

Join Code: 225



What is the **max** number of nodes in a tree of height h ?



BST Analysis

Join Code: 225



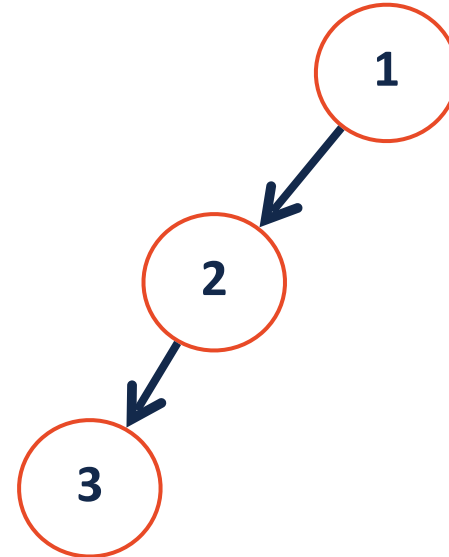
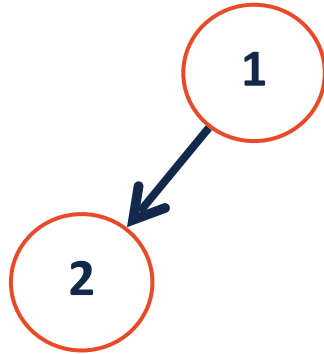
What is the **min** number of nodes in a tree of height h ?

BST Analysis

Join Code: 225



What is the **min** number of nodes in a tree of height h ?

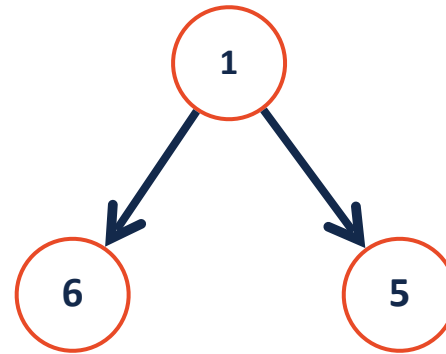


BST Analysis

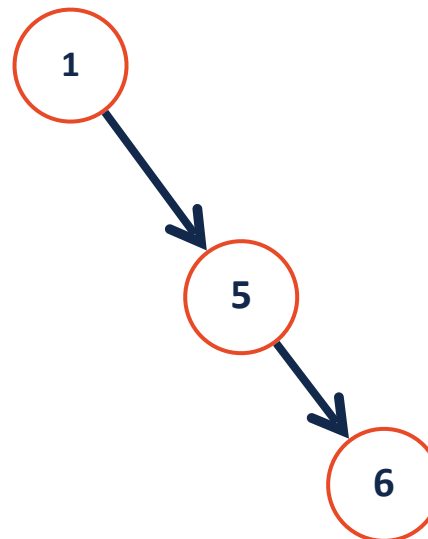


A BST of n nodes has a height between:

Lower-bound: $O(\log n)$



Upper-bound: $O(n)$

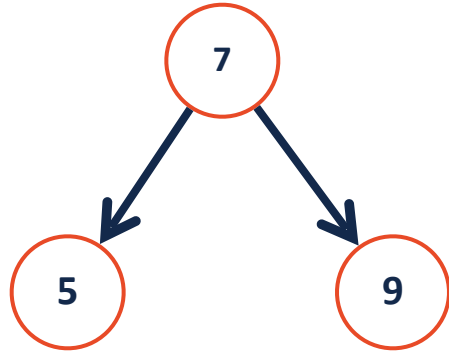


Height-Balanced Tree

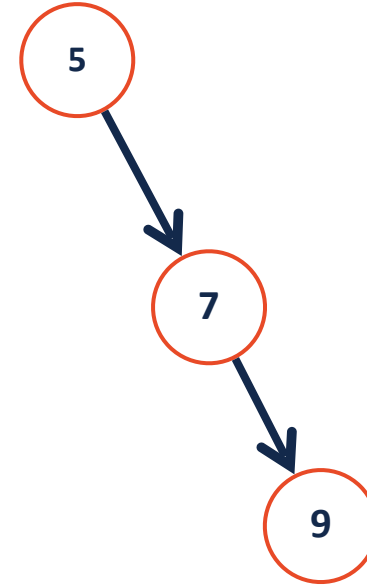
Join Code: 225



What tree is better?



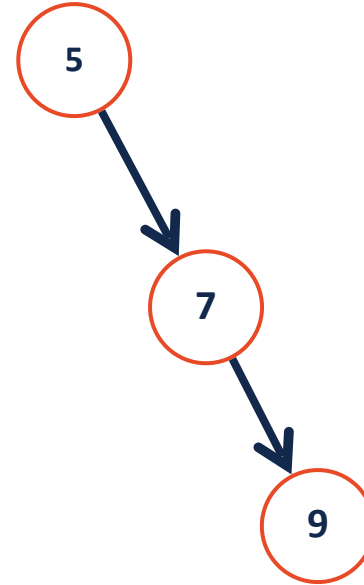
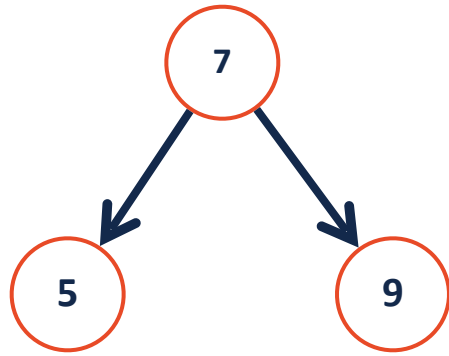
A



B

Height-Balanced Tree

What tree is better?



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

A tree is “balanced” if:



This Week: Creating a multidimensional 'balanced' tree

Next Week: Enforcing a balanced BST