

# Data Structures

## Binary Search Trees

CS 225

September 21, 2026

Brad Solomon



UNIVERSITY OF  
**ILLINOIS**  
URBANA - CHAMPAIGN

Department of Computer 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)!

# Learning Objectives

Extend binary trees into binary *search* trees

Build conceptual and coding understanding of BST

Introduce tree balance

# What search algorithm is best?

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

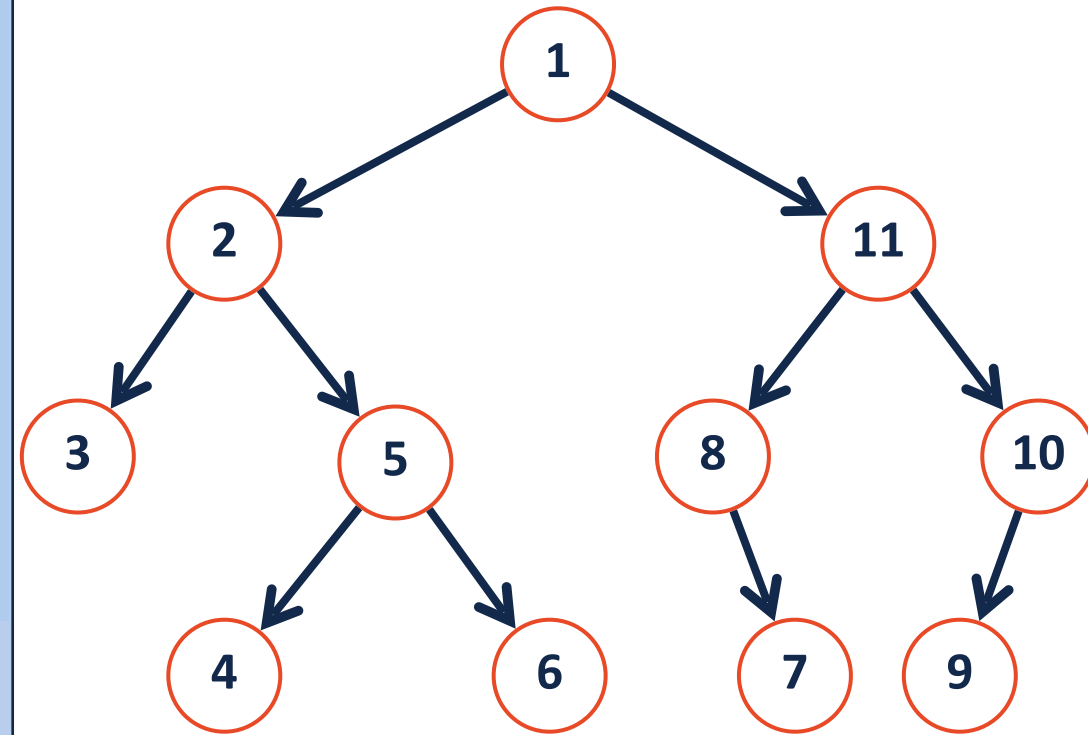
Join Code: 225



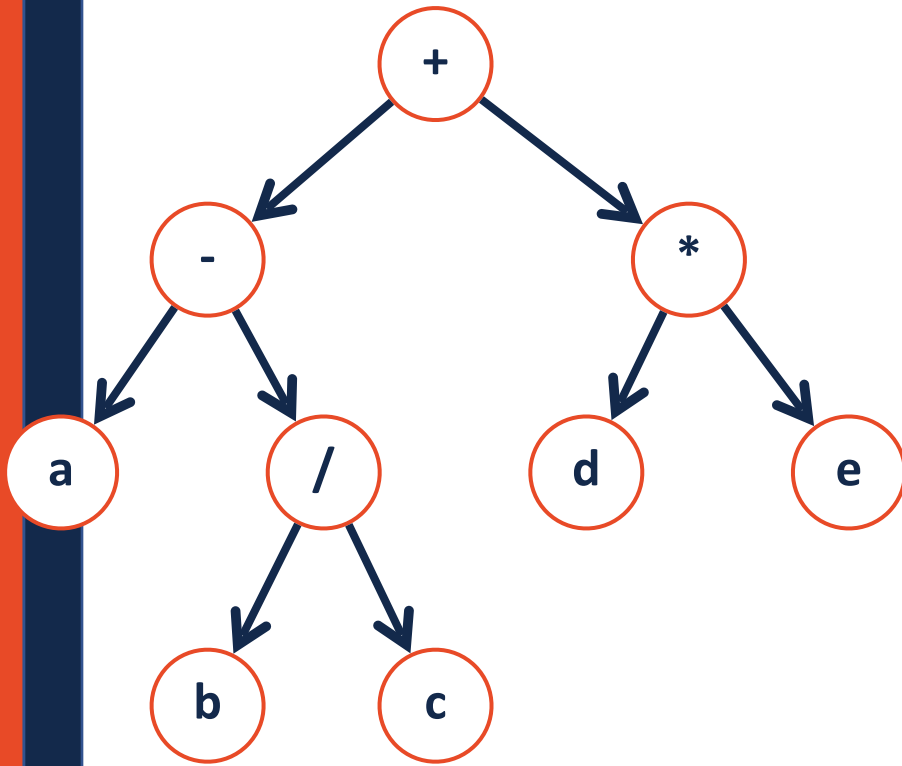
# Depth First Search

**Explore as far along one path as possible before backtracking**

```
1 template<class T>
2 bool BinaryTree<T>::DFS_search(TreeNode*
  root, T& query)
3 {
4     if (!root) {
5         return false;
6     }
7
8     return process(root->data, query) ||
9         DFS_search(root->left, query) ||
10        DFS_search(root->right, query);
11 }
12
13
14
15
16
17
18
```



# Level-Order Traversal

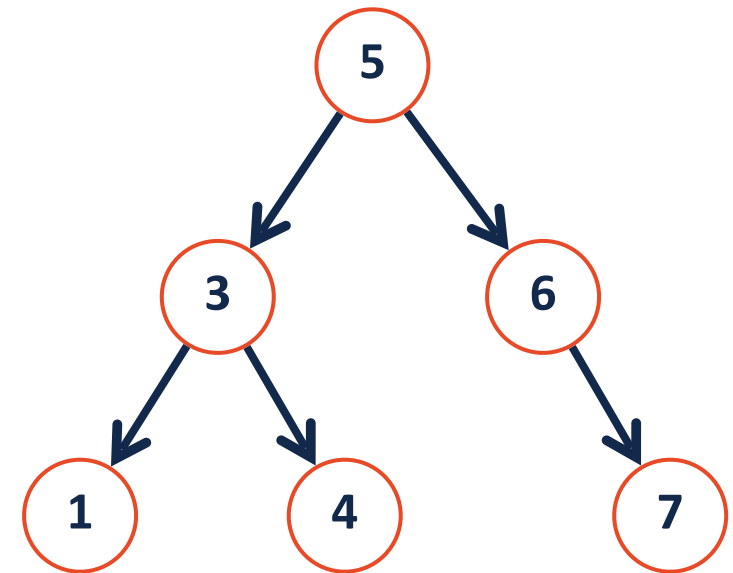
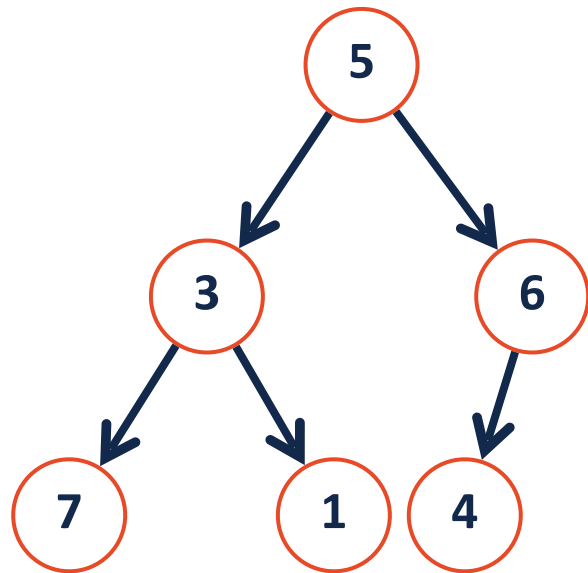


```
1 template<class T>
2 bool BinaryTree<T>::BFS_search(TreeNode* root, T&
  query)
3 {
4     if (!root) return false;
5
6     std::queue<TreeNode*> q;
7     q.push(root); // enqueue == push
8
9     while (!q.empty()) {
10         TreeNode* temp = q.front();
11         q.dequeue();
12
13         if (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 }
```

# Improved search on a binary tree

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 5 | 3 | 6 | 7 | 1 | 4 |
|---|---|---|---|---|---|

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 1 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|

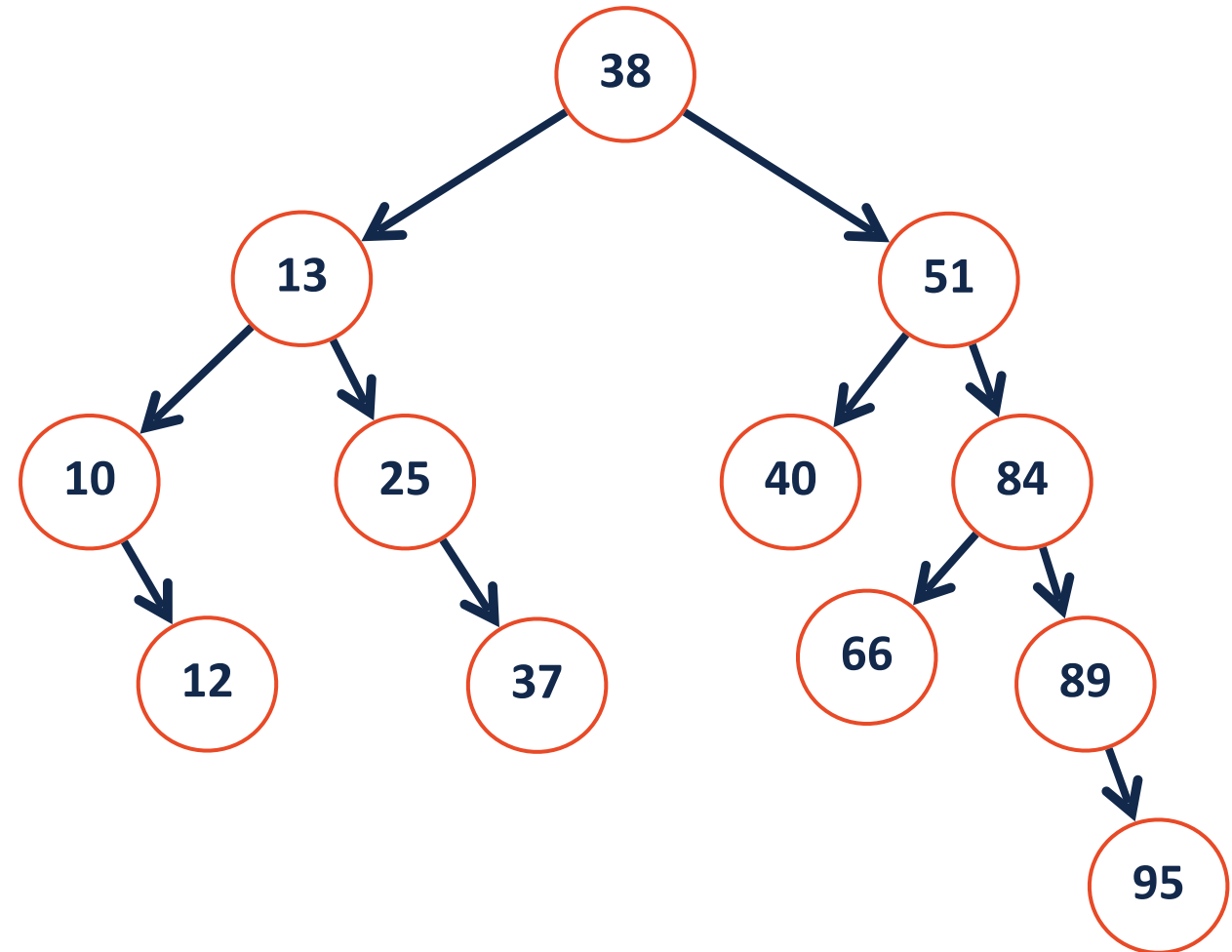


# Binary Search Tree (BST)

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

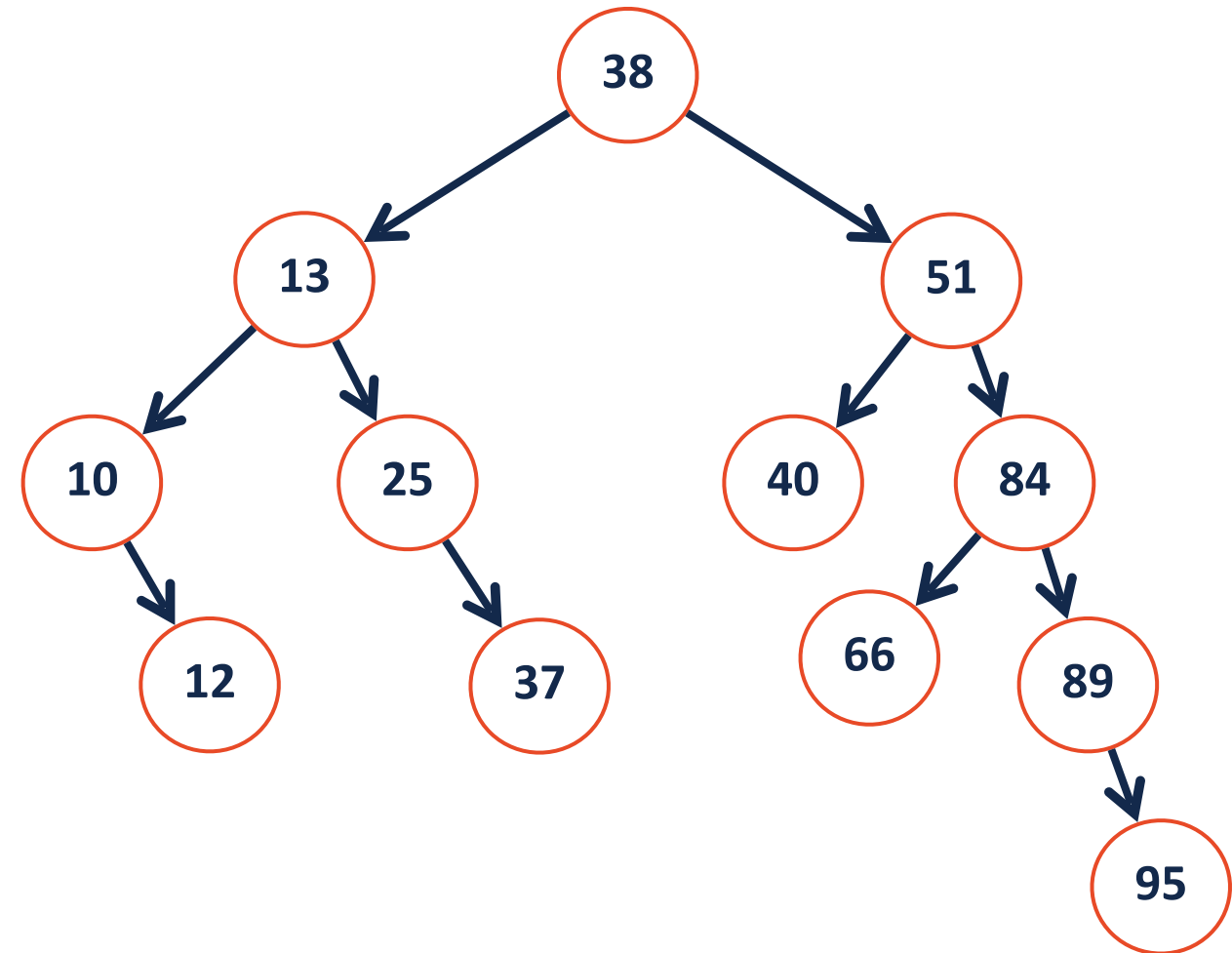
Traverse

Find

Constructor

# BST Find

**find(66)**



# BST Find

**find(66)**

**A recursive function based around value of root:**

**Base Case:** If root is null, return root

Let tmp = root->key()

tmp == query, return root

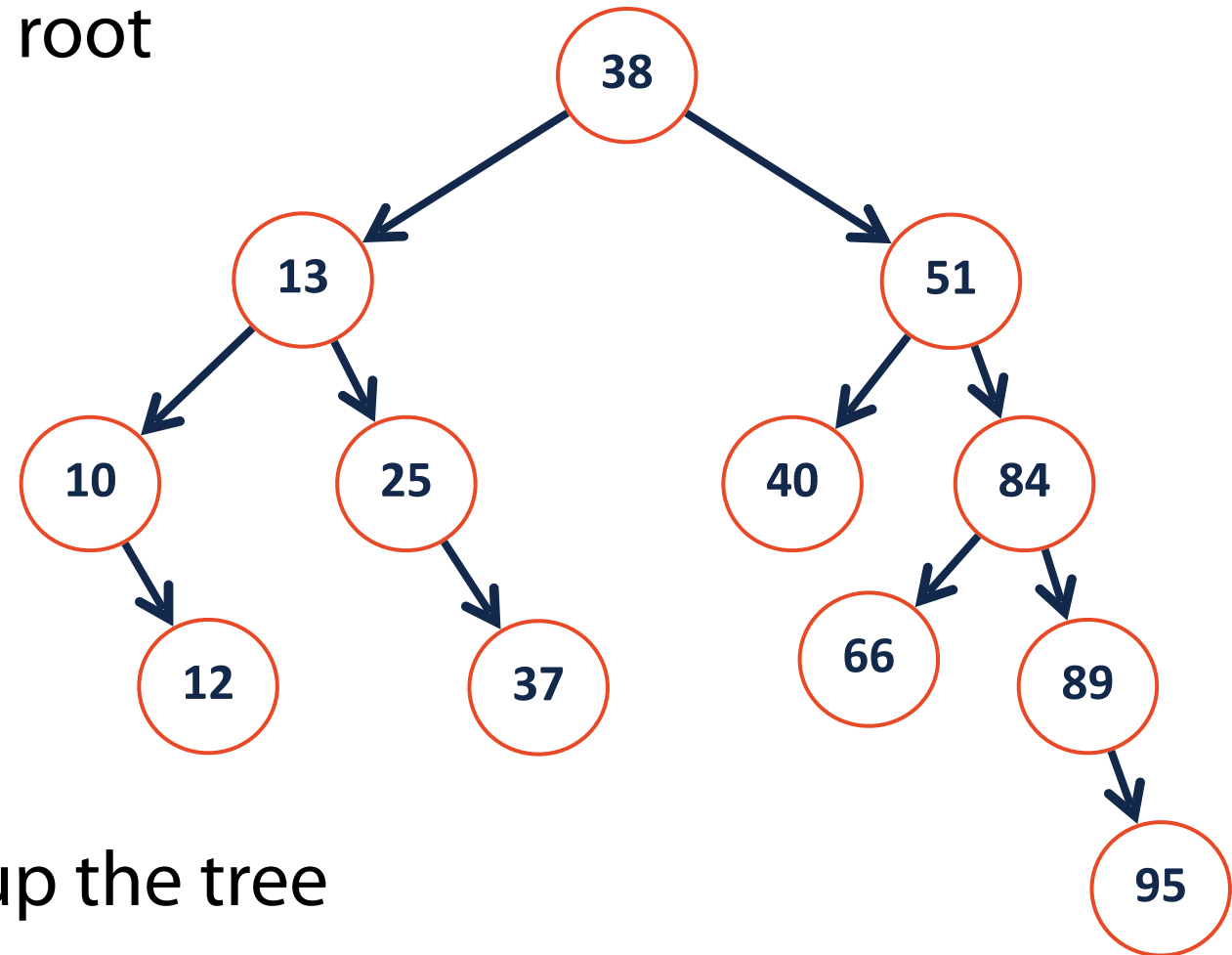
**Recursion:**

tmp < query, recurse right

tmp > query, recurse left

**Combining:**

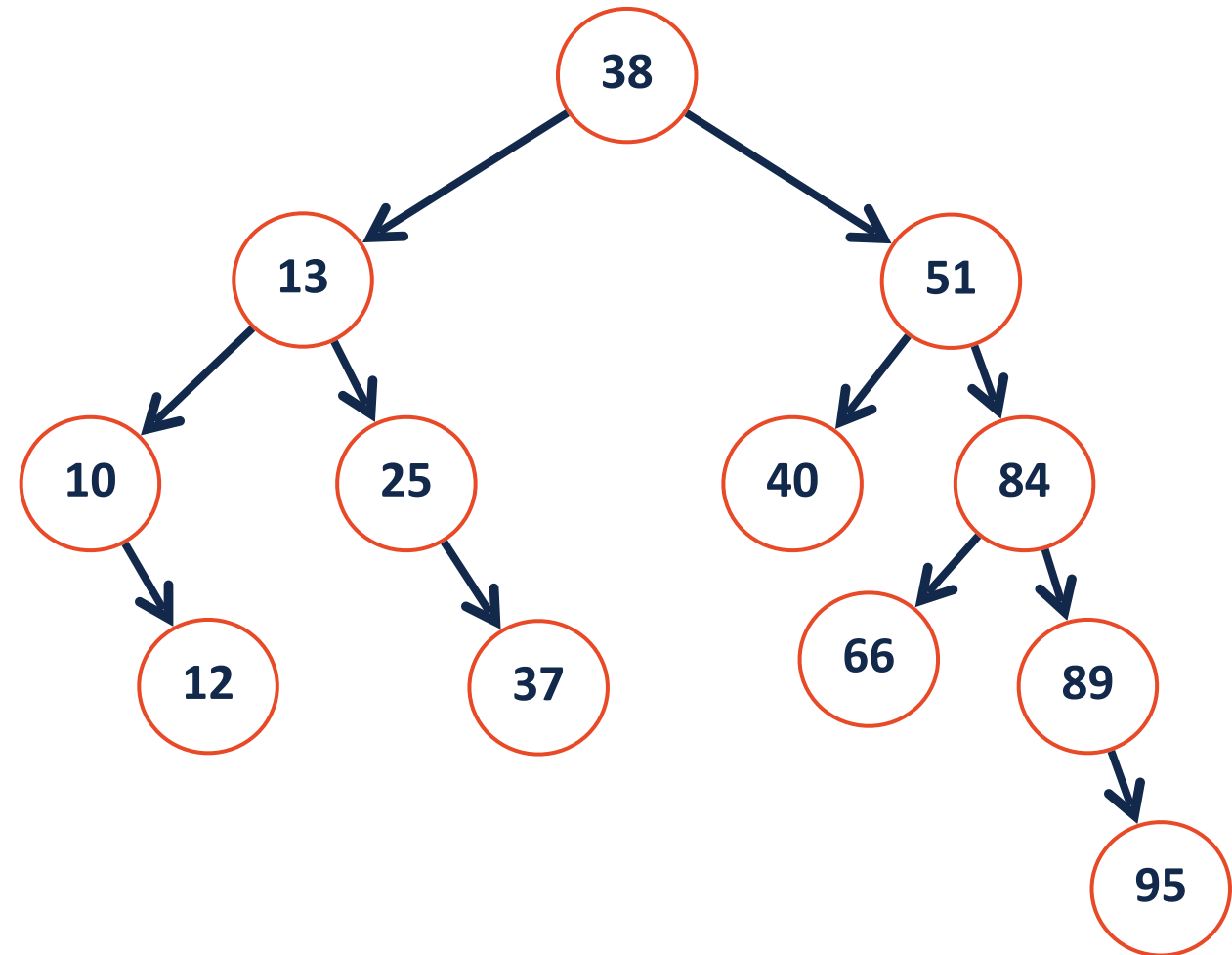
Return the recursive value back up the tree



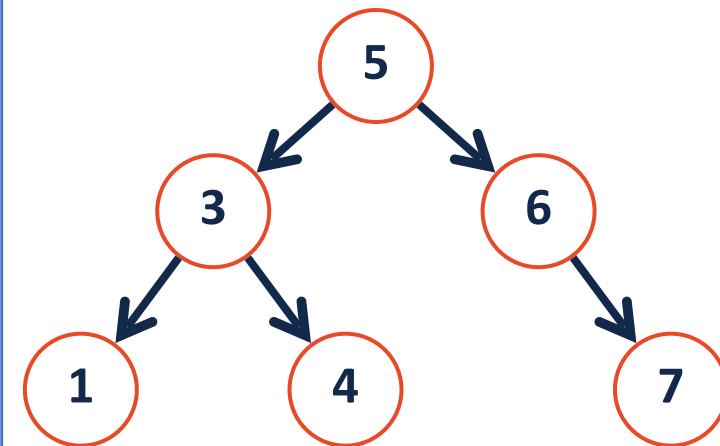
# BST Find

**find(9)**

Make sure you can follow the search path!

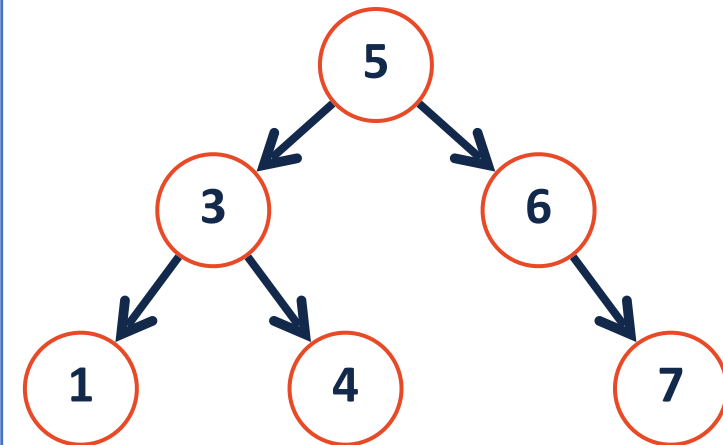


```
1 template<typename K, typename V>
2
3 _____ _find(TreeNode *& root, const K & key) {
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23 }
```



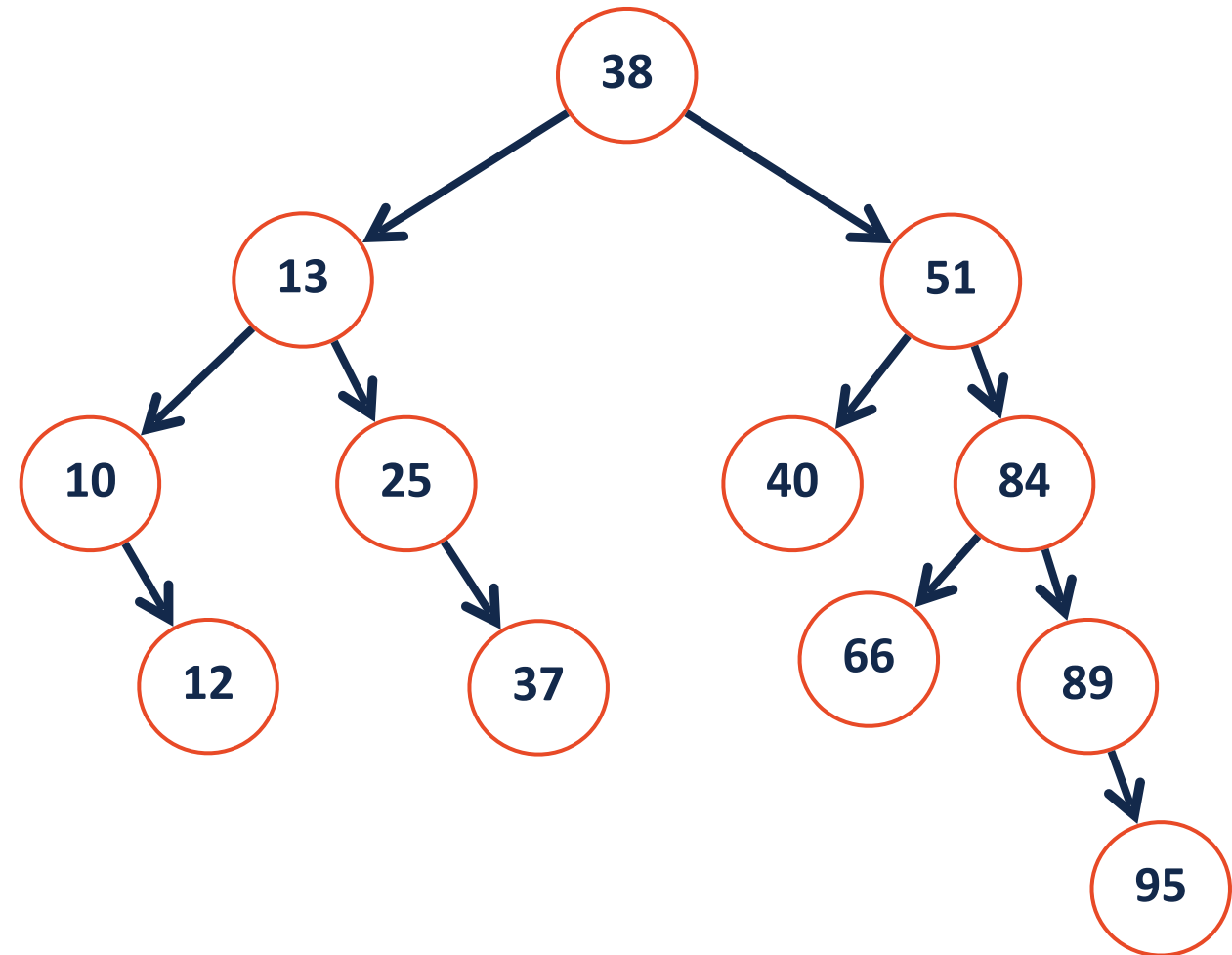


```
1 template<typename K, typename V>
2
3     TreeNode *& __find(TreeNode *& root, const K & key) {
4
5
6     // Base Case
7     if(root == nullptr || root->data == key){
8         return root;
9     }
10
11    // Recursive Step ("Combining step" is 'return')
12    if (root->data > key){
13        return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



# BST Insert

**insert(33, v)**

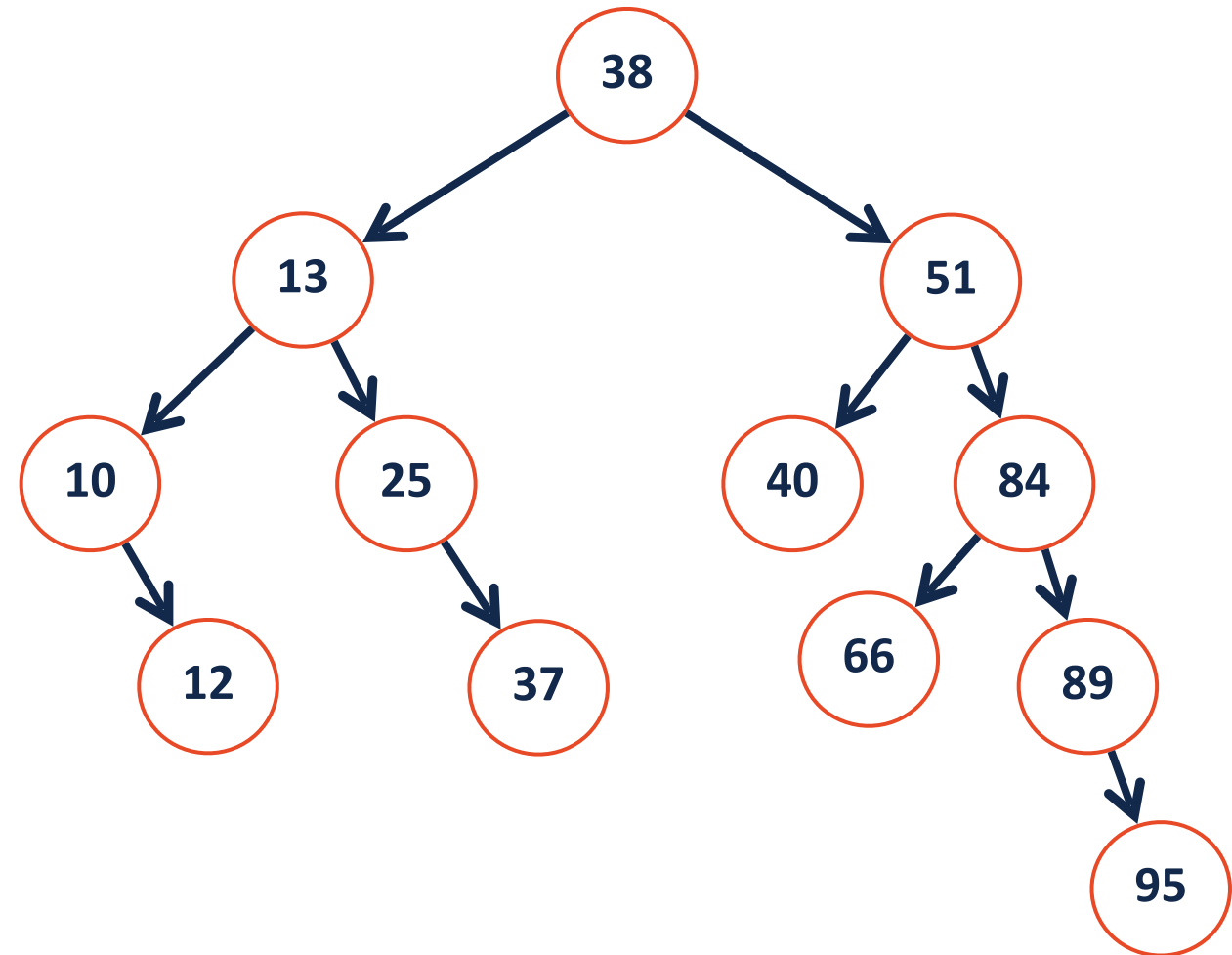


# BST Insert

**insert(15, v)**

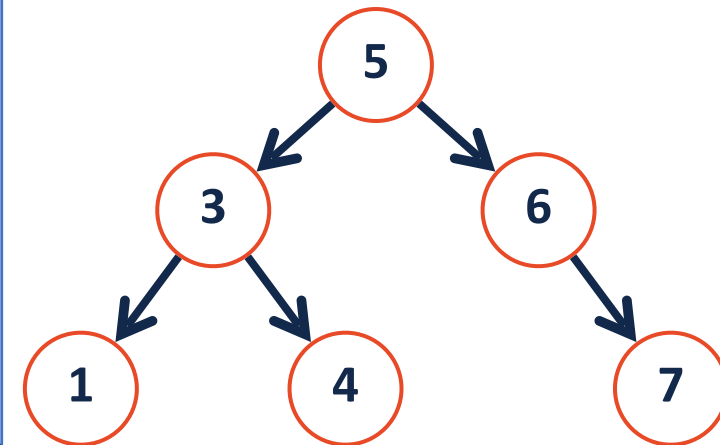
Find the insert location using `_find()`

Trivially insert at location



```
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
```

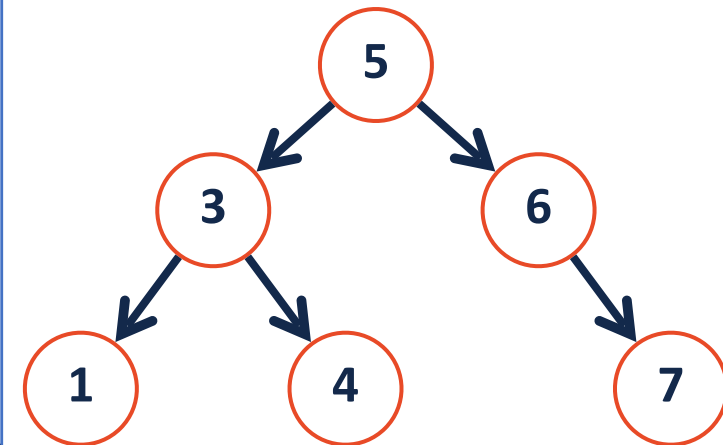
```
1 template<typename K, typename V>
2
3 void _insert(TreeNode *& root, const K & key, const V & val) {
4
5
6
7
8
9
10
11
12
13
14
15
16 }
```





```
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

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

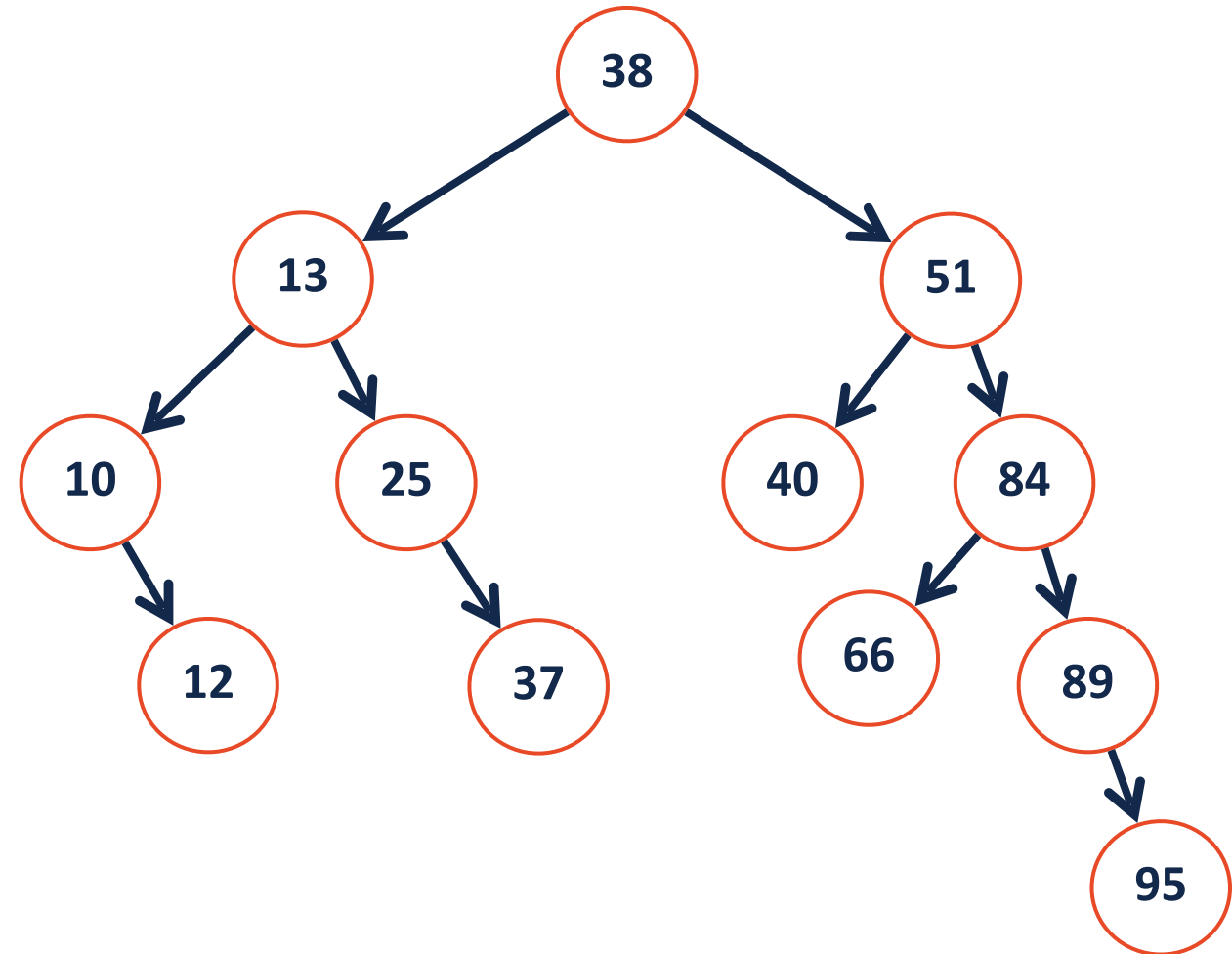
# BST Remove

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

**remove (40)**

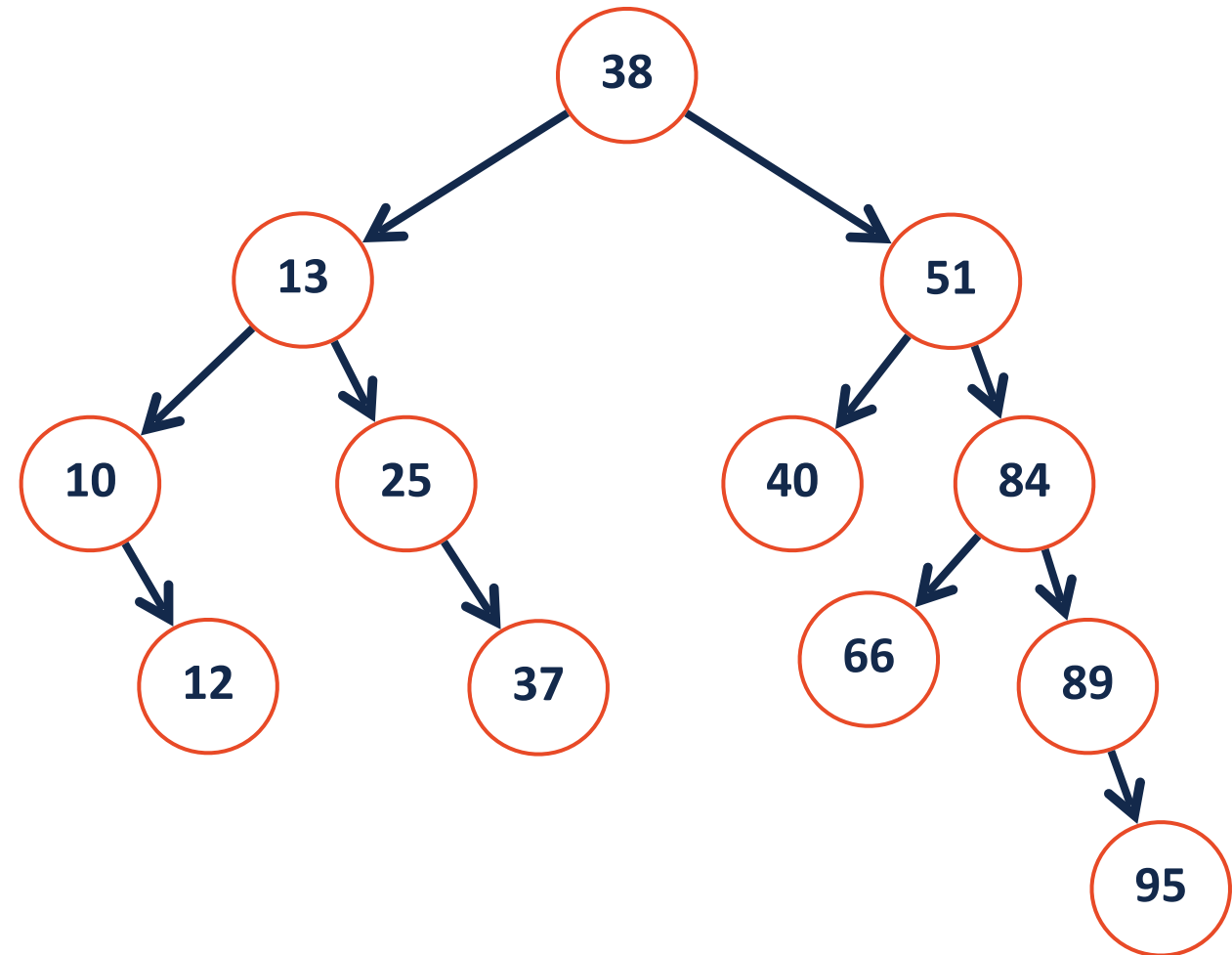
**remove (25)**

**remove (51)**



# BST Remove

**remove (40)**



# BST Remove

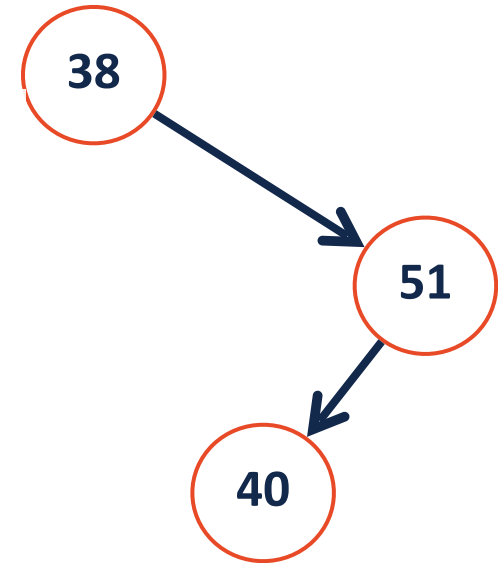
## 0-Child Case

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

```
delete t;
```

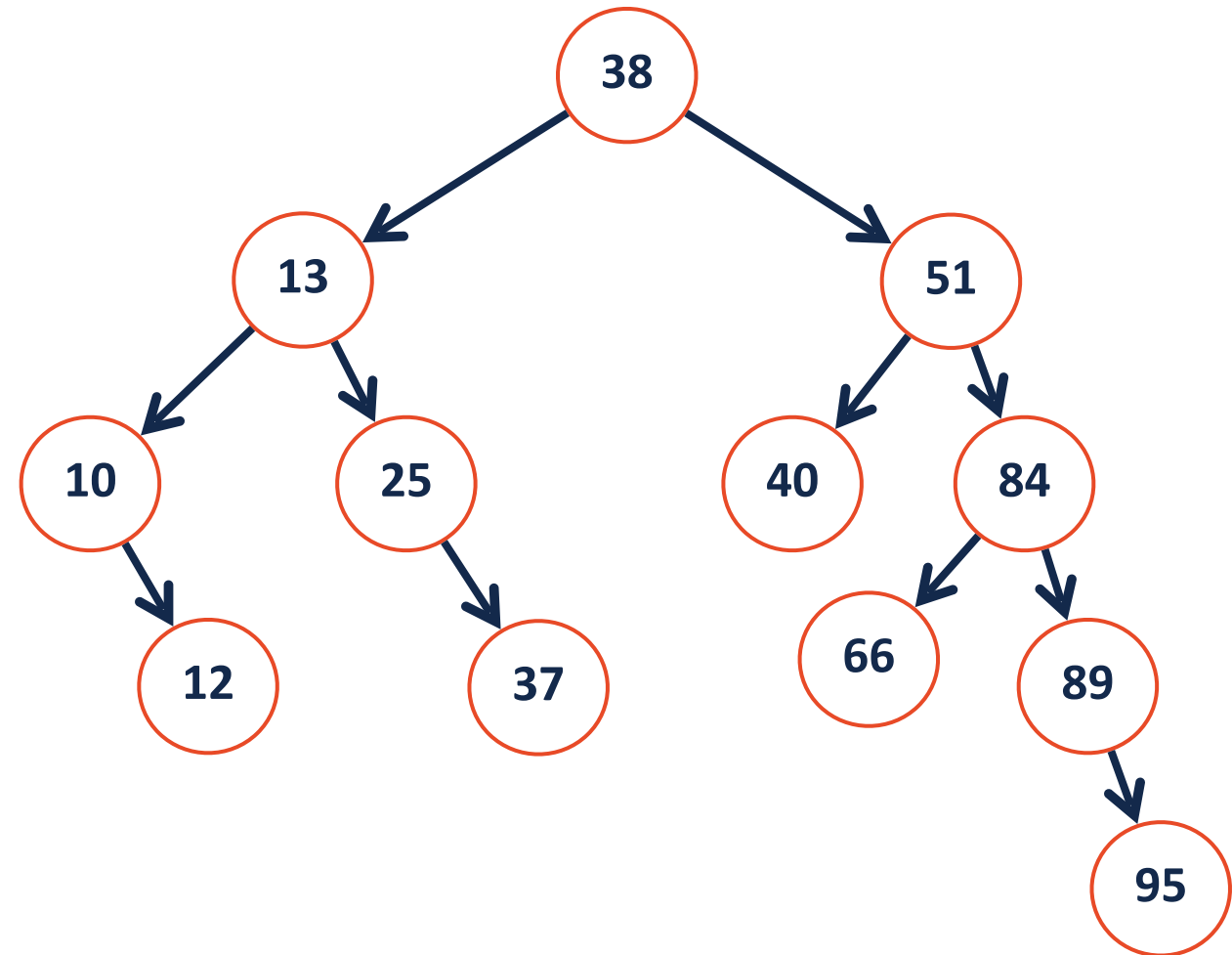
```
t = nullptr;
```

**remove (40)**



# BST Remove

**remove (25)**



# BST Remove

## 1-Child Case

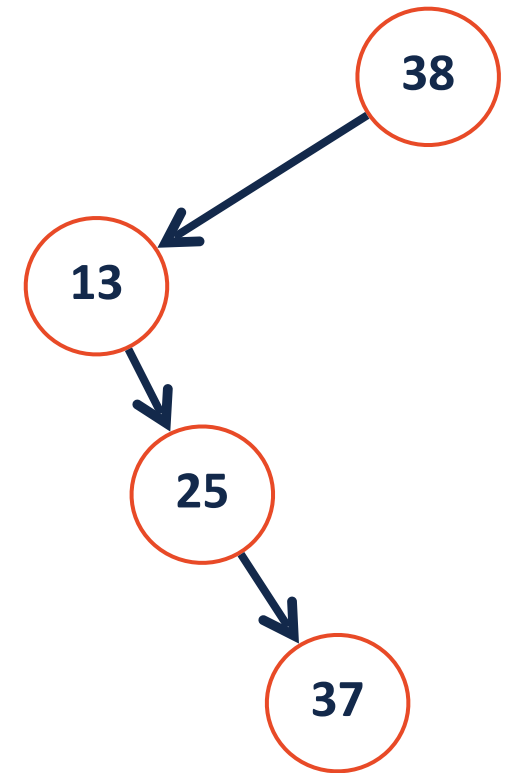
```
TreeNode *& t = _find(root, 25);
```

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

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

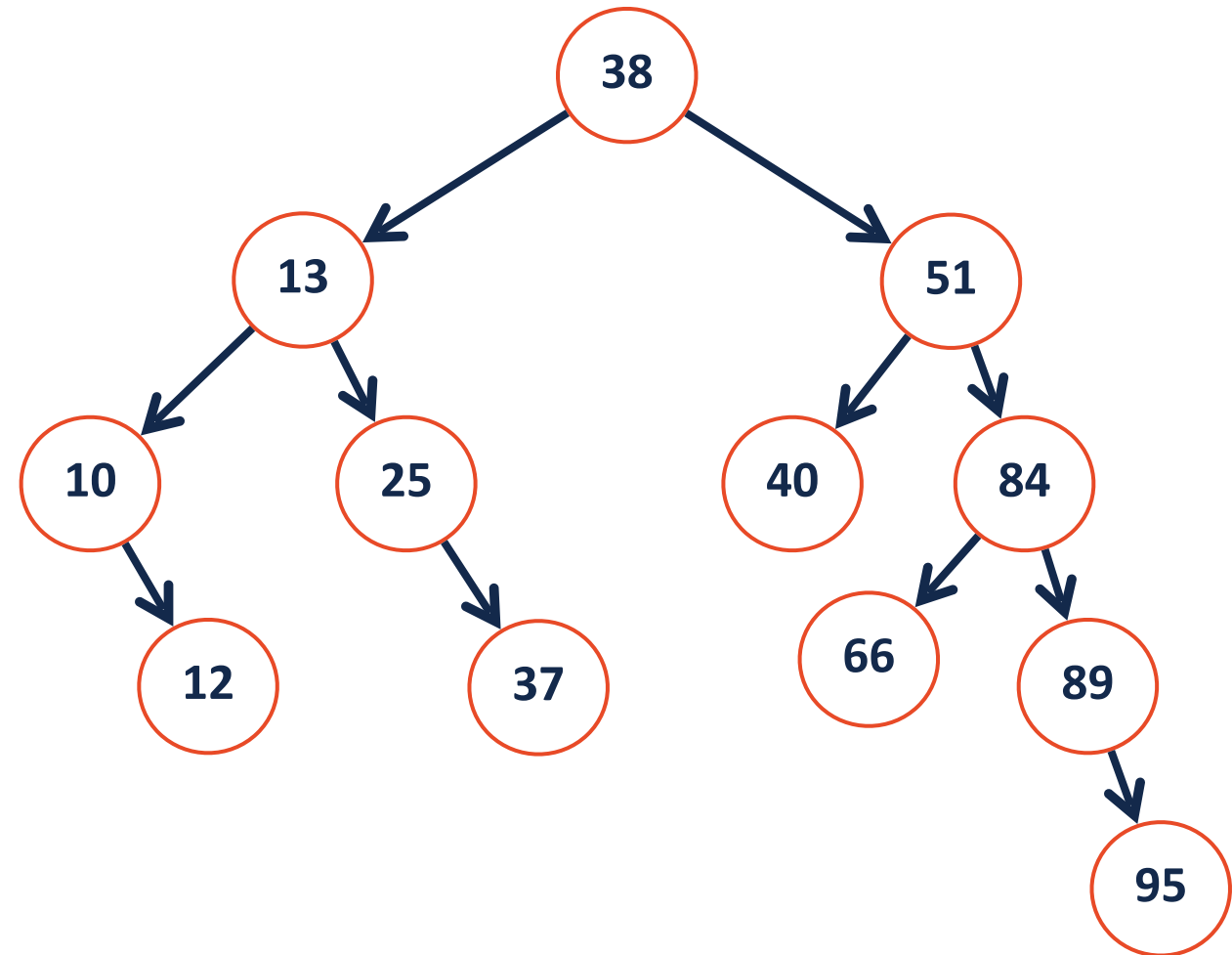
```
delete tmp;
```

**remove (25)**



# BST Remove

**remove (13)**



# BST In-Order \_\_\_\_\_

## In-Order Predecessor

*Rightmost left child*

IOP(38) =

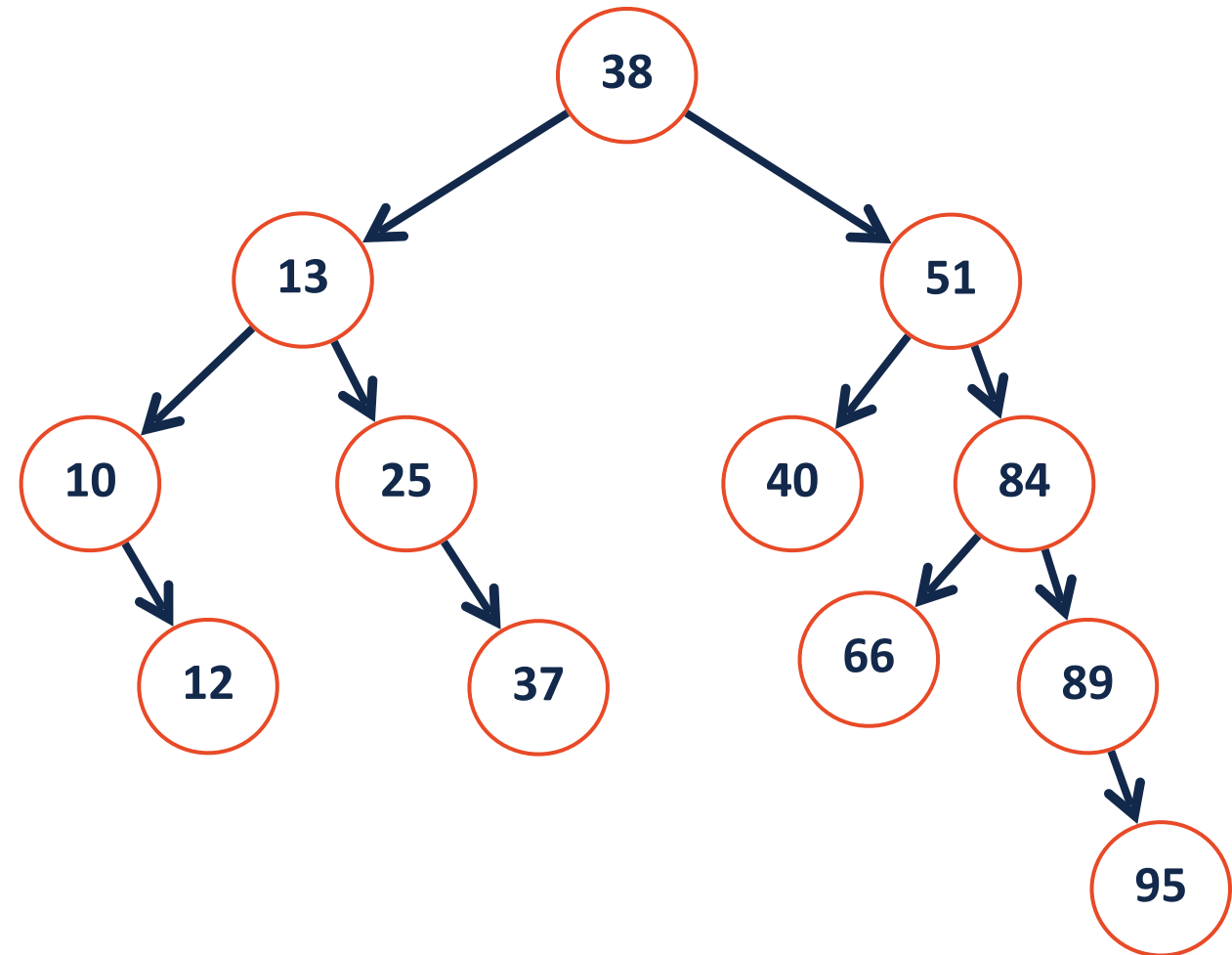
IOP(84) =

## In-Order Successor

*Leftmost right child*

IOS(38) =

IOS(84) =



# BST Remove

**remove(13)** 

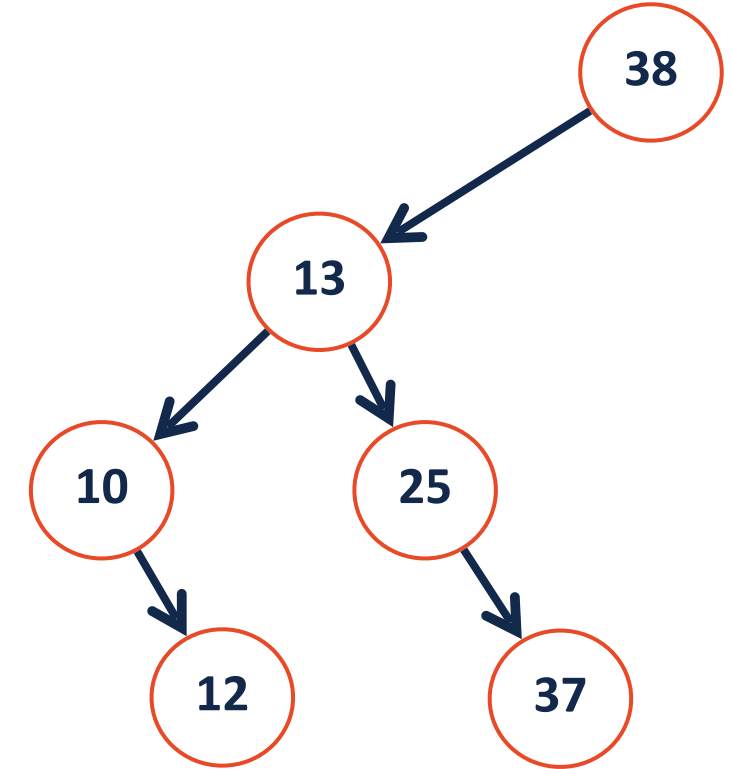
## 2-Child Case

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

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

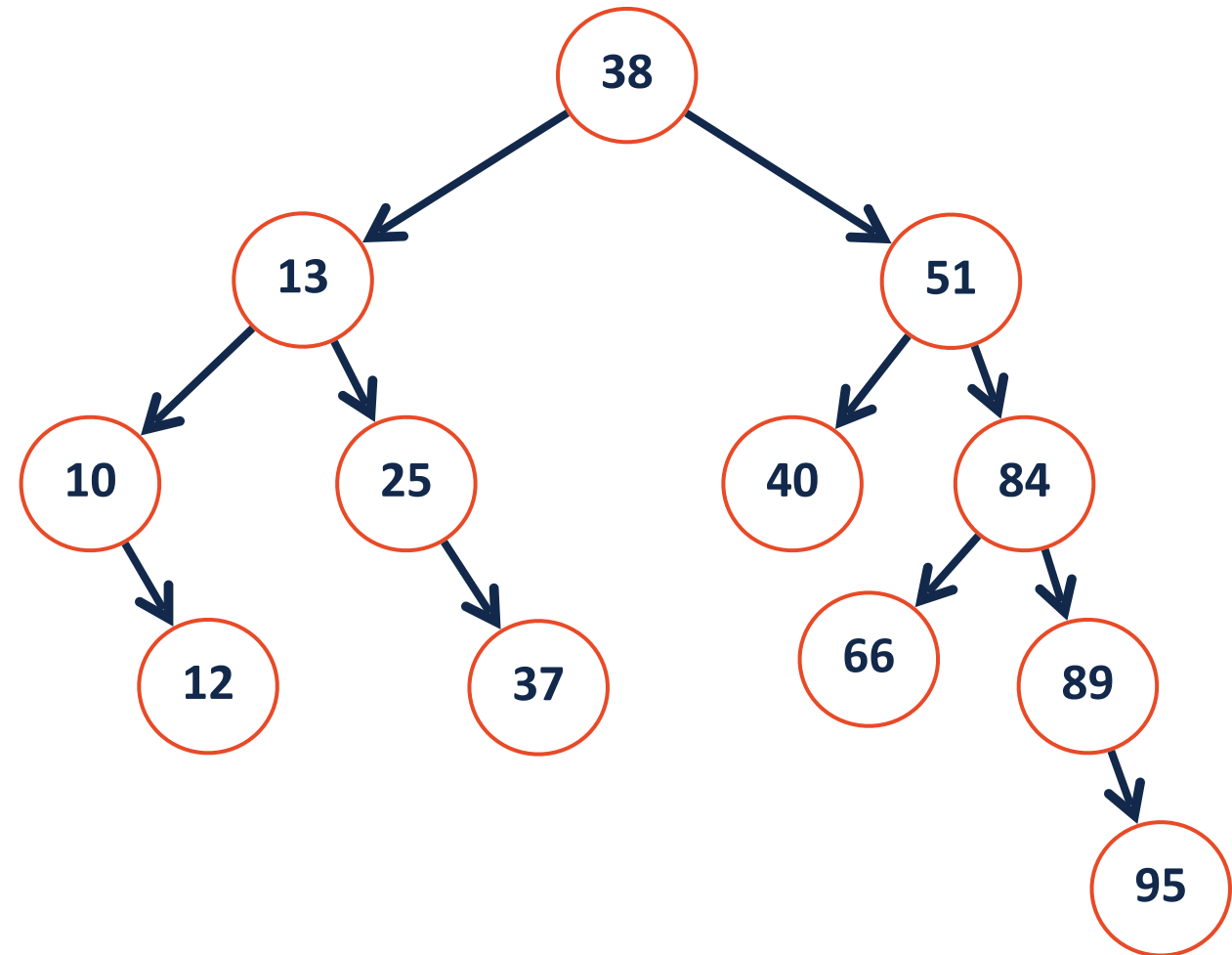
```
swap(t, iop);
```

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



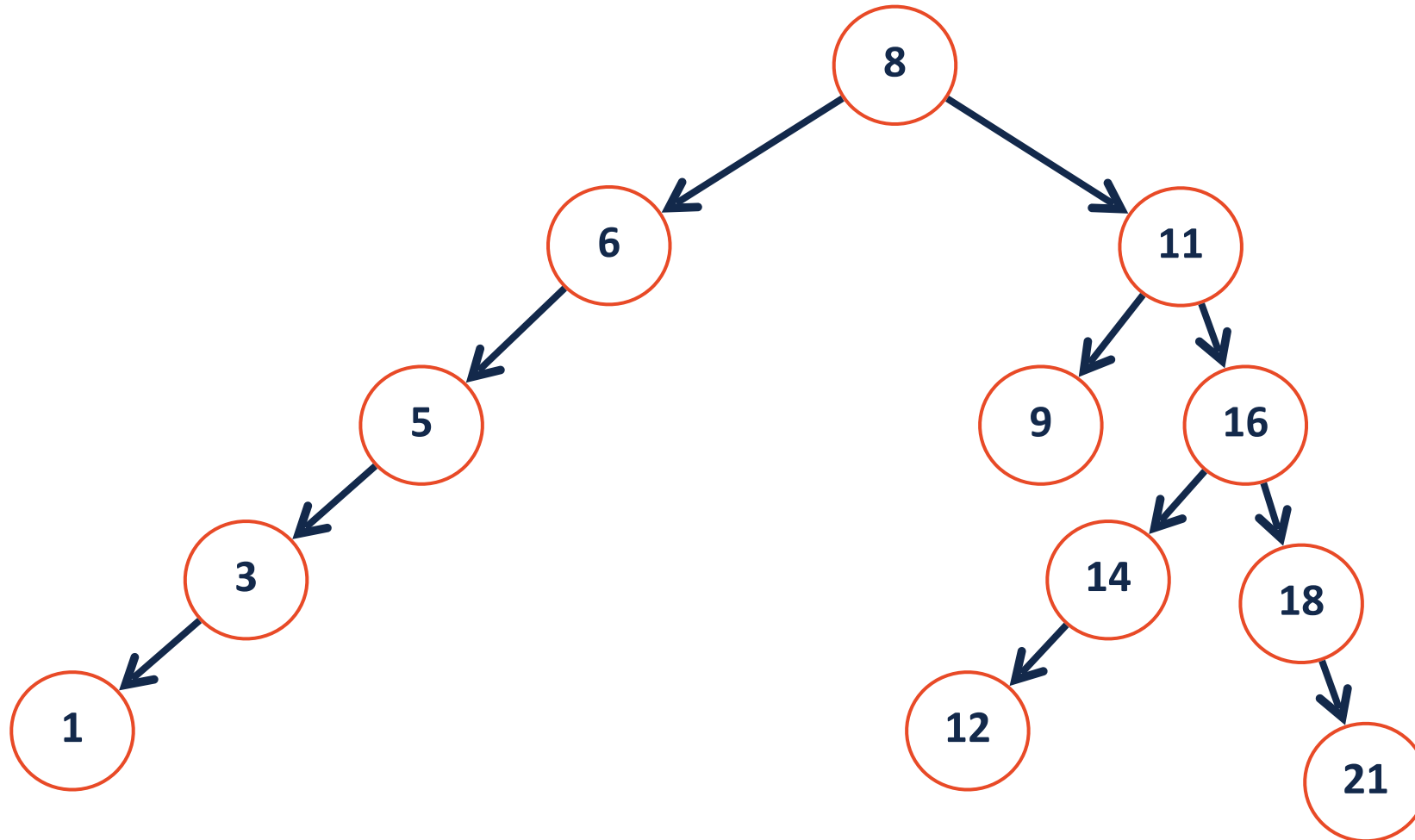
# BST Remove

**remove (51)**

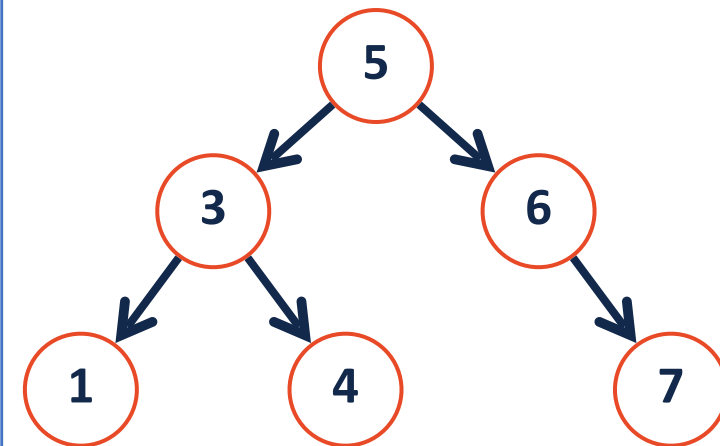


# BST Remove

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 }
```



# BST Analysis – Running Time



| Operation | BST Worst Case |
|-----------|----------------|
| find      |                |
| insert    |                |
| remove    |                |
| traverse  |                |

# BST Analysis

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

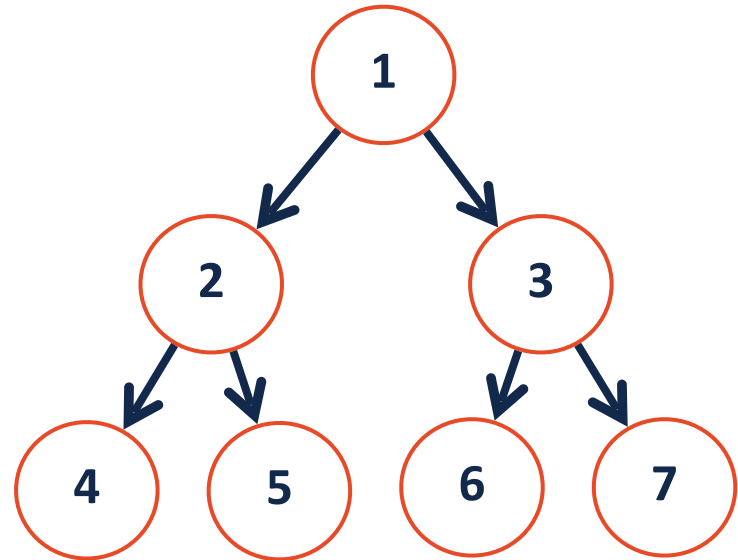
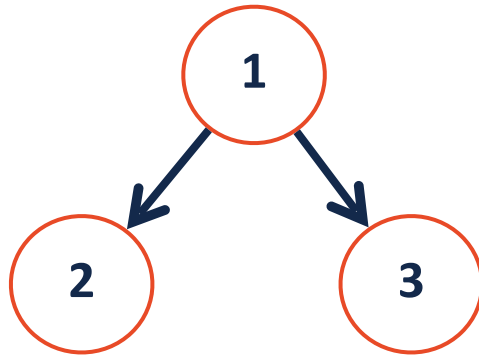
... how do we relate  $O(h)$  to  $n$ , the size of our dataset?

# BST Analysis

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

# BST Analysis

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

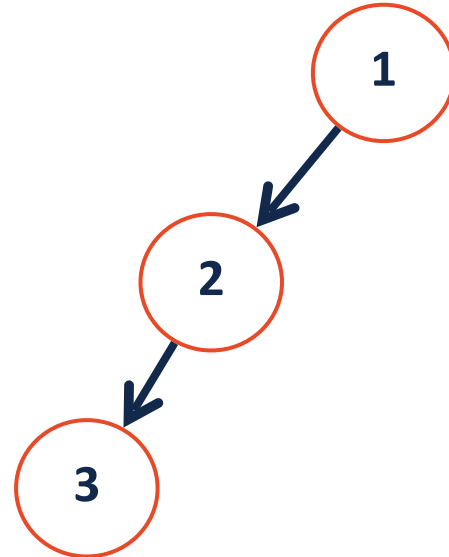
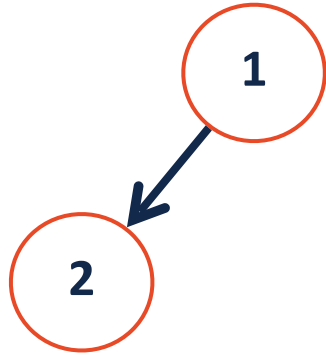


# BST Analysis

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

# BST Analysis

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

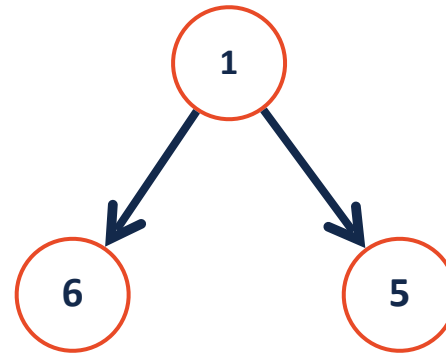


# BST Analysis

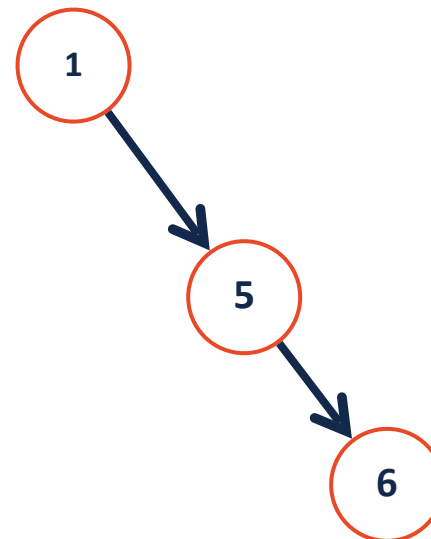


A BST of  $n$  nodes has a height between:

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

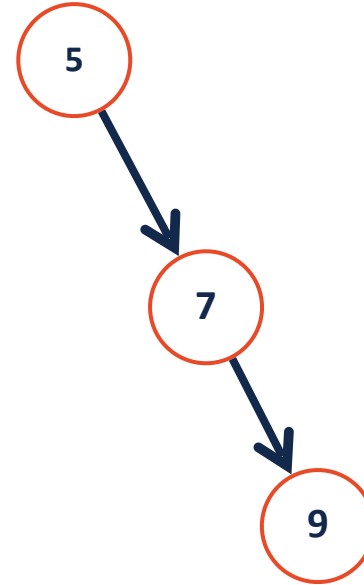
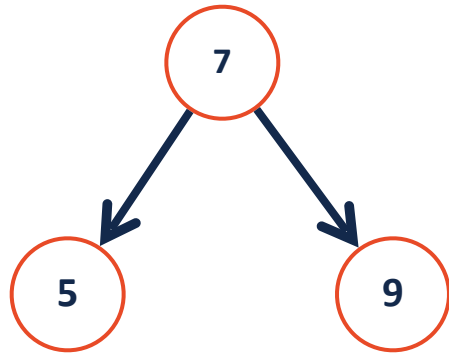


**Upper-bound:**  $O(n)$



# Height-Balanced Tree

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