Algorithms and Data Structures for Data Science
Binary Search Tree

CS 277
Brad Solomon

April 9, 2025

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Learning Objectives

Review understanding of Binary Trees

Extend ADT to Binary Search Trees

Implement BST operations

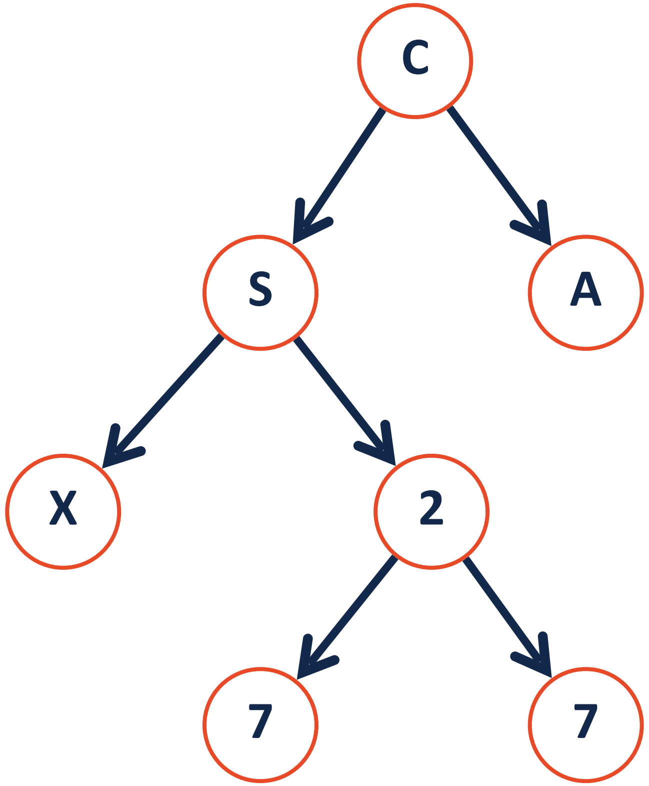
(Binary) Tree Recursion

A **binary tree** is a tree T such that:

$$T = None$$

or

$$T = treeNode(val, T_L, T_R)$$




```
class treeNode:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
class binaryTree:
    def __init__(self):
        self.root = None
```

Tree ADT

Constructor: Build a new (empty) tree

Insert: Add an object into tree

Remove: Remove a specific object from tree

Traverse: Visit every node in tree (all objects)

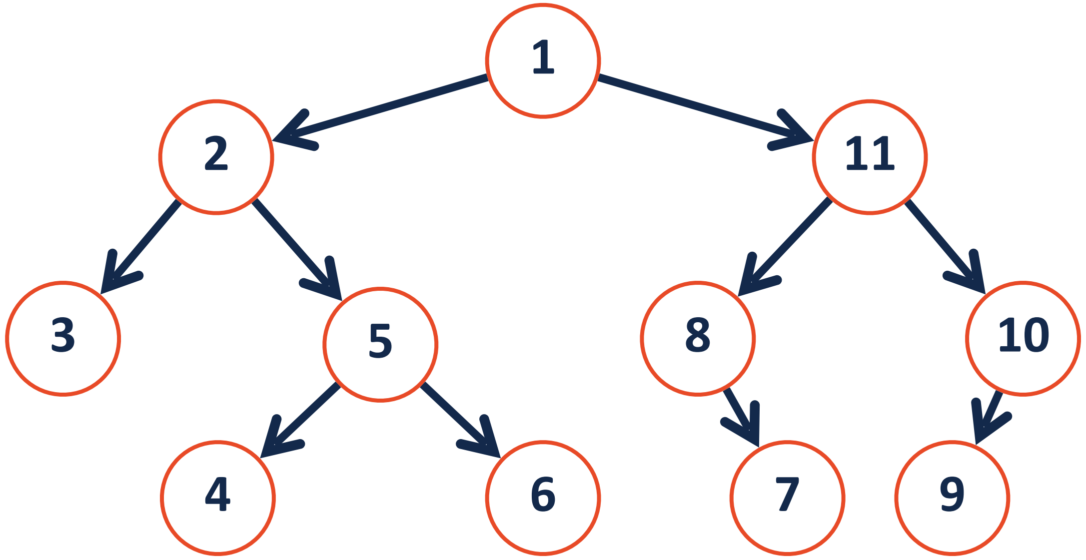
Search: Find a specific object in the tree

Binary Tree Traversal

Last class we implemented traversals using recursion, stacks, and queues.

What implementations led to a **depth first search traversal**?

Which lead to **breadth first search?**



Binary Tree Utility

This week we will deep dive into useful implementations of binary trees

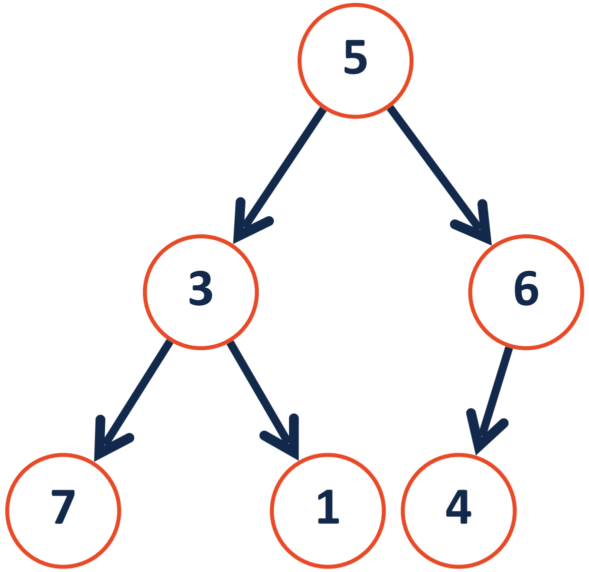
Binary Search Tree: An efficient implementation of a dictionary

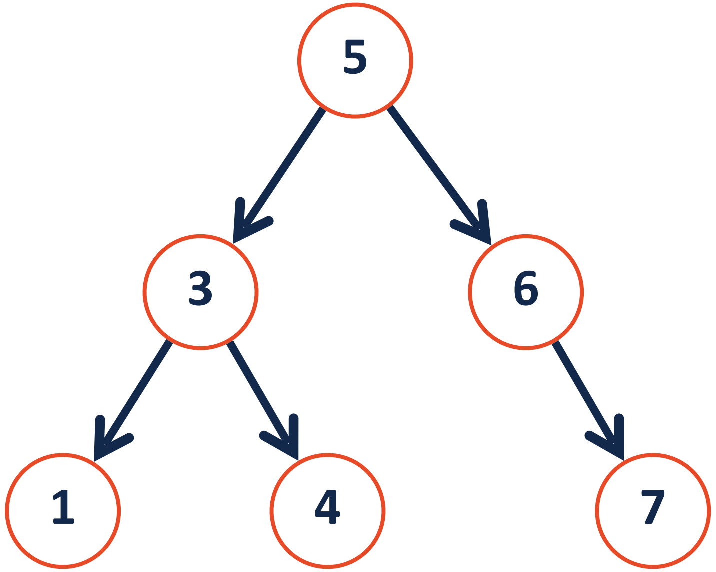
Huffman Tree: A binary tree used to define an optimal text encoding

Improved search on a binary tree

5	3	6	7	1	4

1	3	4	5	6	7



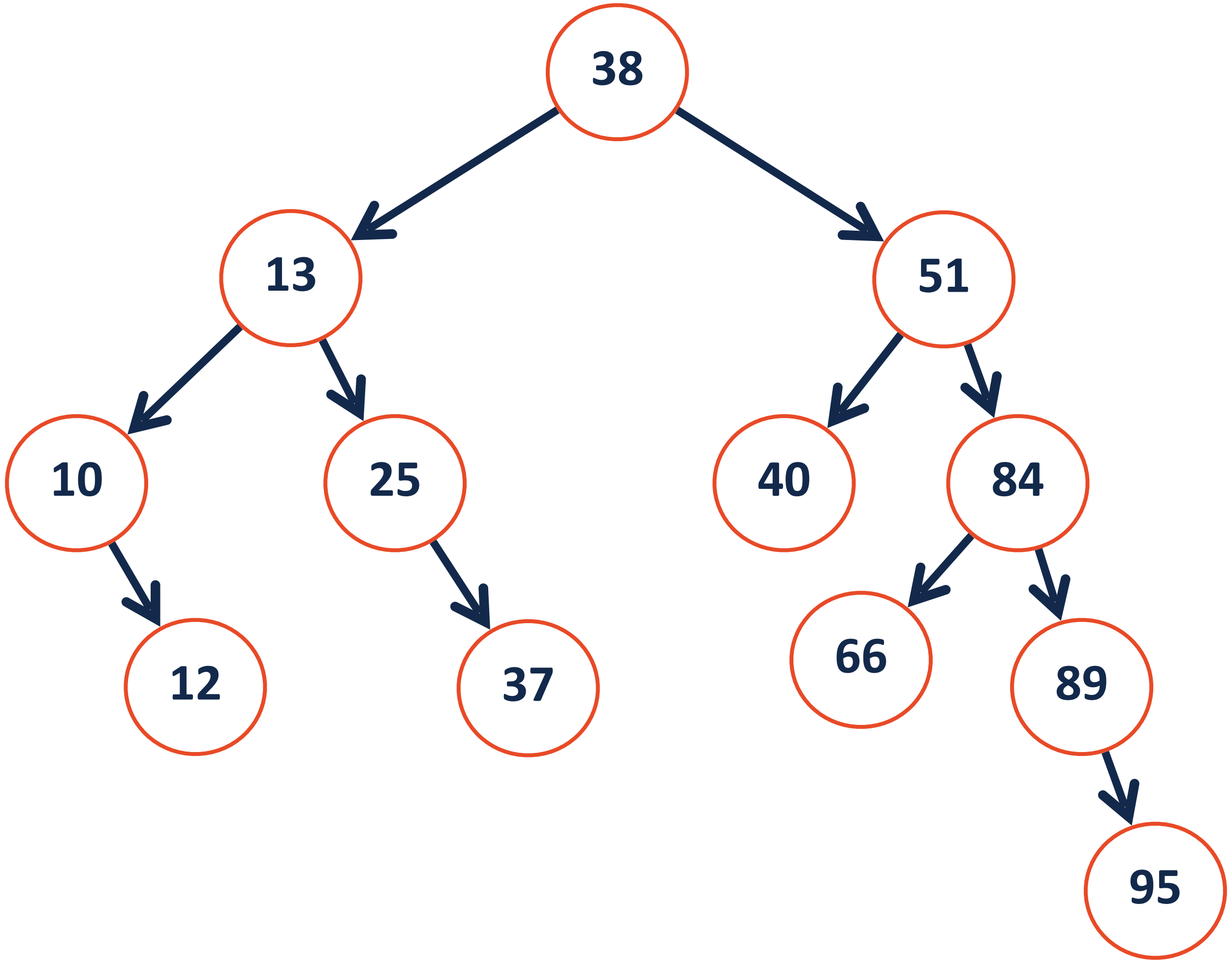


Binary Search Tree (BST)

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

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

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



Dictionary ADT

Data is often organized into key/value pairs:

Word ➔ Definition

Course Number ➔ Lecture/Lab Schedule

Node ➔ Edges

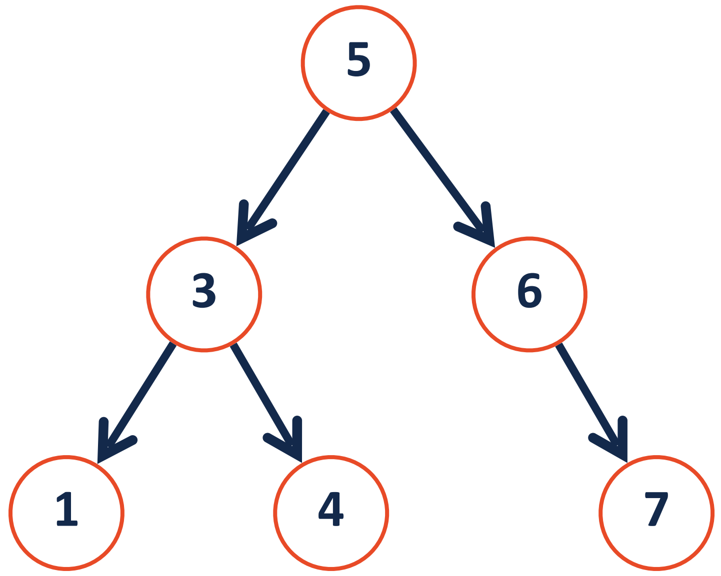
Flight Number ➔ Arrival Information

URL ➔ HTML Page

Average Image Color ➔ File Location of Image

Dictionary as a Binary Search Tree

```
class bstNode:
    def __init__(self, key, val, left=None, right=None):
        self.key = key
        self.val = val
        self.left = left
        self.right = right
```



Key	5	3	6	7	1	4
Value	A	B	C	D	E	F

Binary **Search** Tree ADT — **what changed?**

Constructor: Build a new (empty) tree

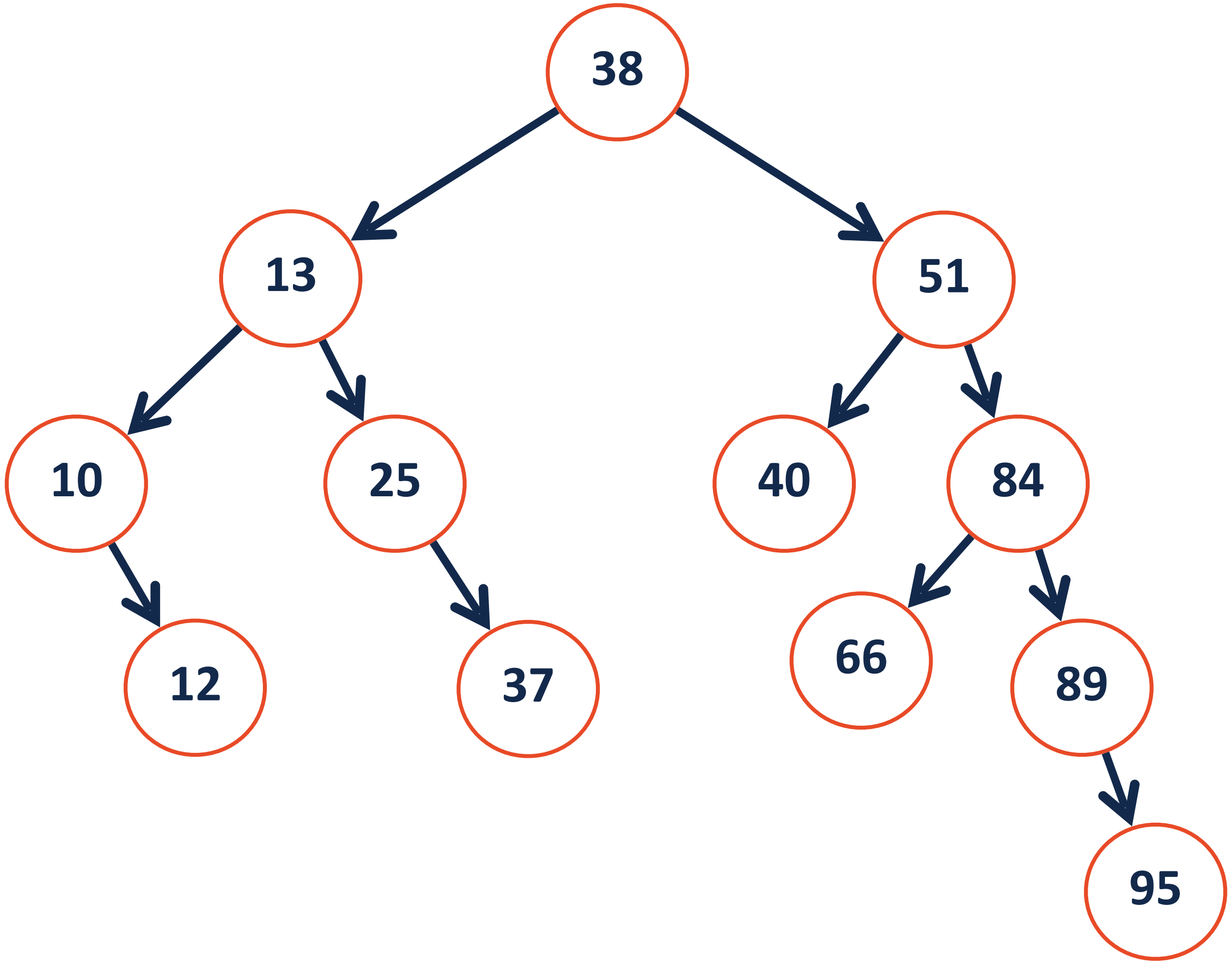
Insert: Add an object into tree

Remove: Remove a specific object from tree

Traverse: Visit every node in tree (all objects)

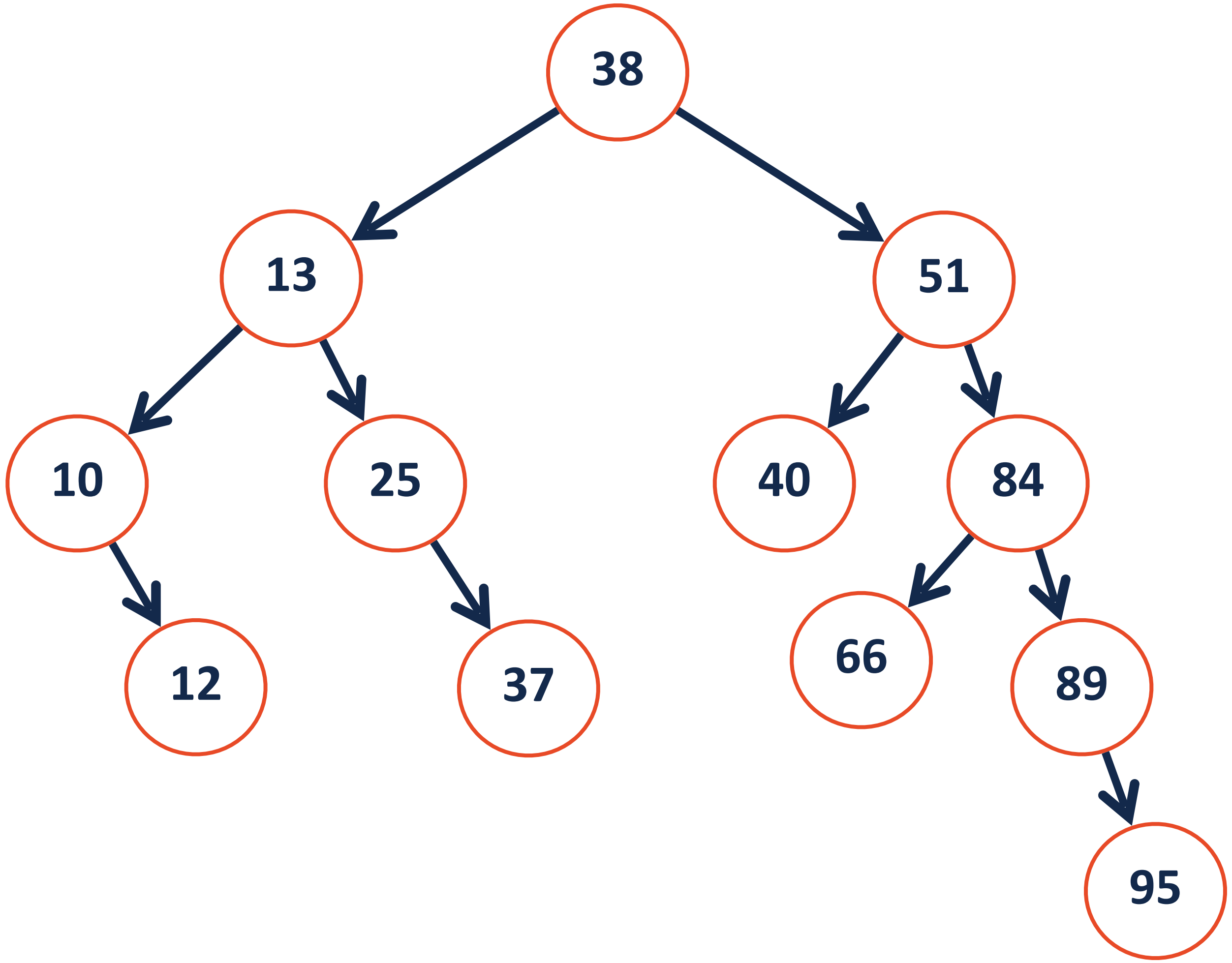
Find
~~Search~~: Find a specific **key** in the tree, **return value**

BST In-Order Traversal



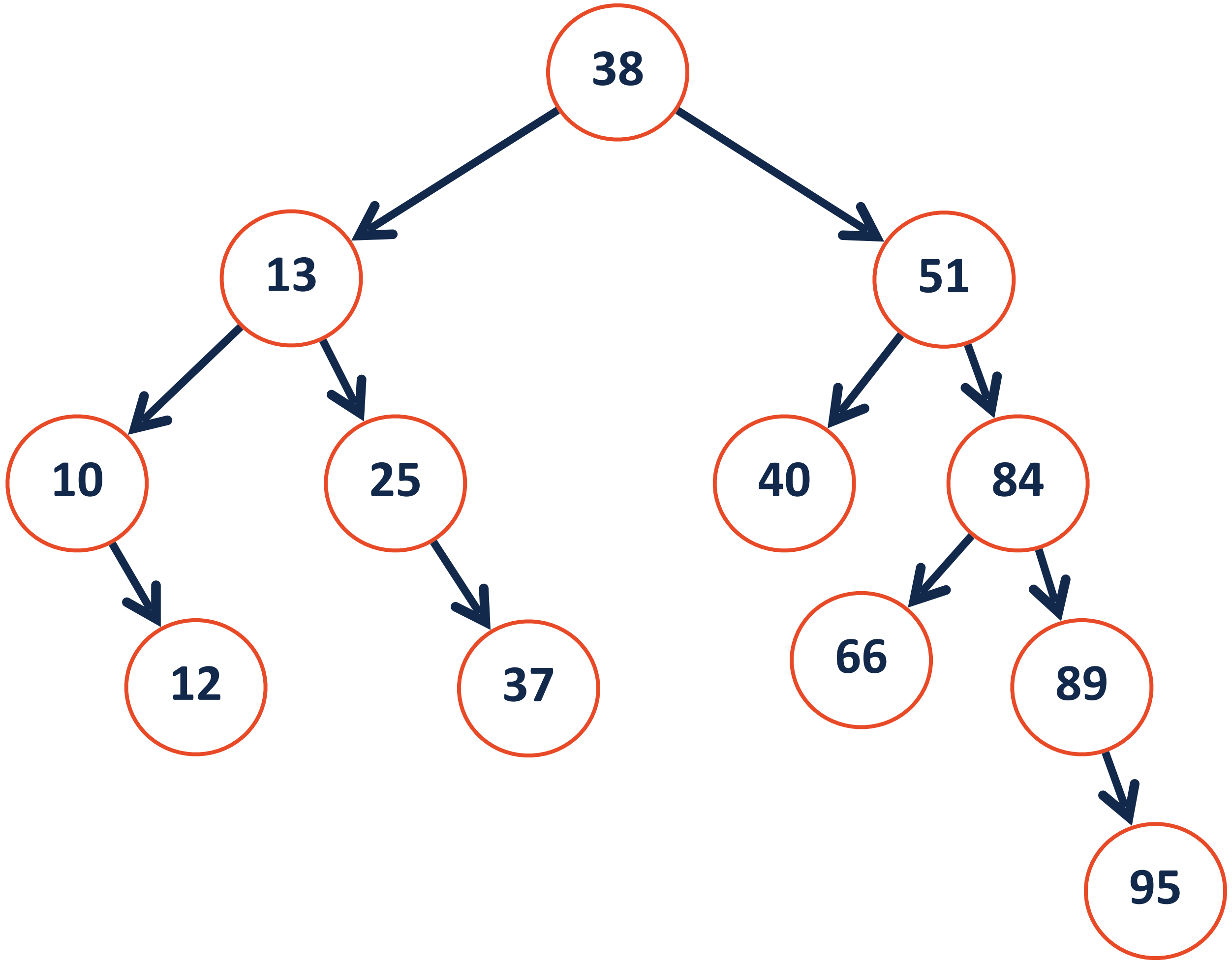
BST Find

`find(66)`

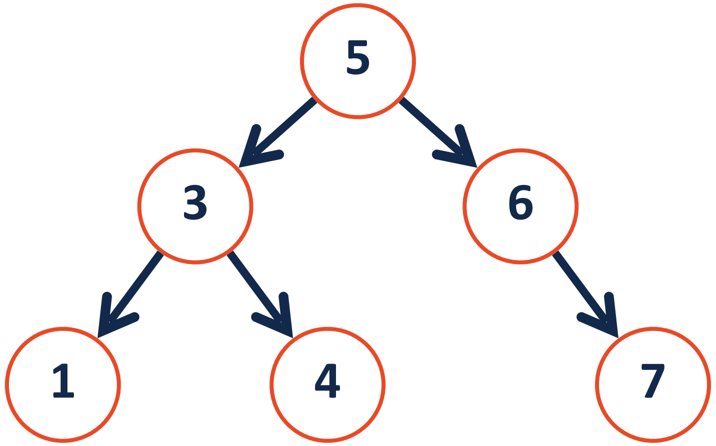


BST Find

`find(9)`



BST Find



Base Case:

Recursive Step:

Combining:

BST Find

A recursive function based around value of root:

Base Case: If root is None, return root

If root.key is query, return root

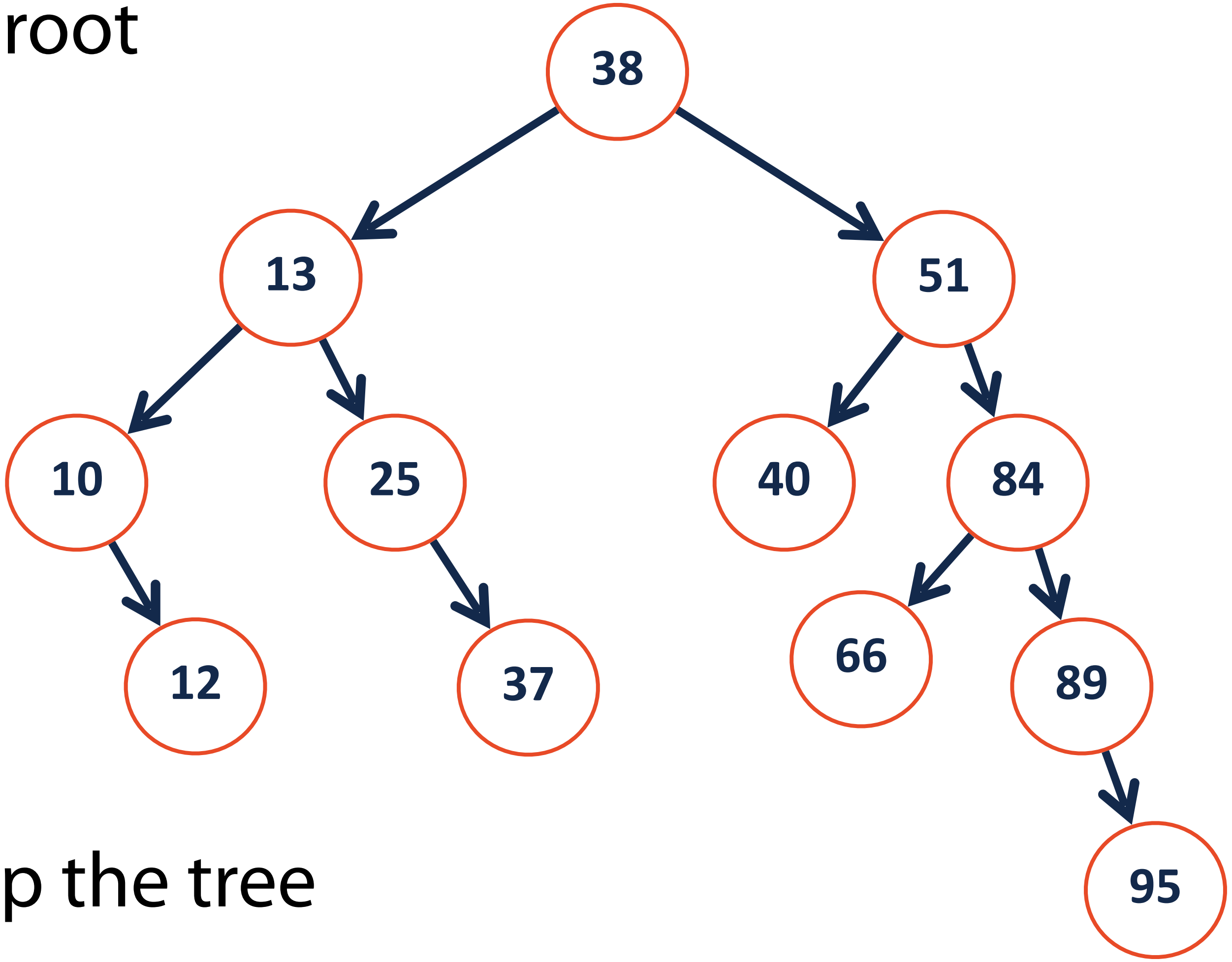
Recursion:

root.key ___ query, recurse right

root.key ___ query, recurse left

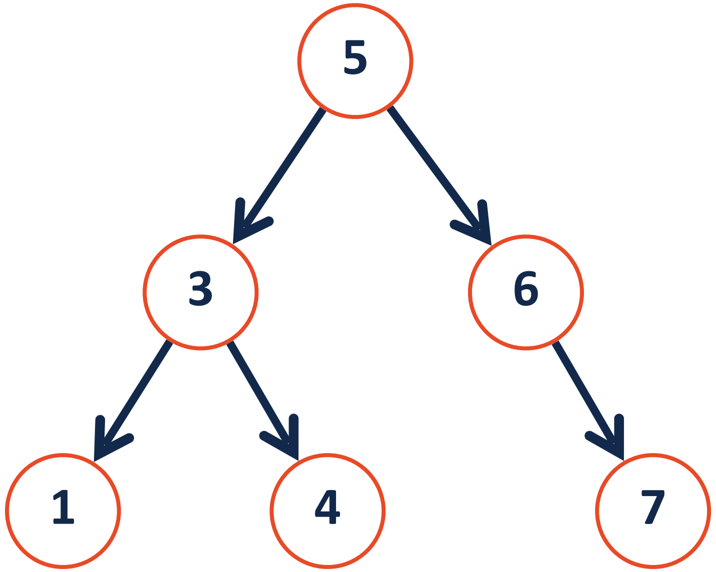
Combining:

Return the recursive value back up the tree

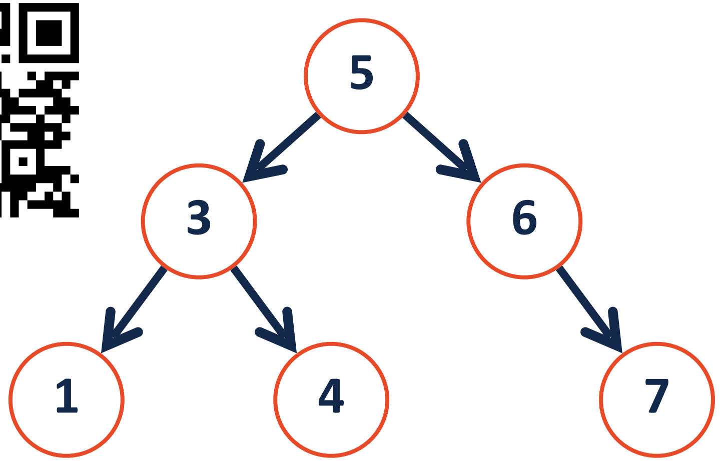


BST Find

```
#inside class bst
def find(self, key):






def find_helper(self, node, key):
```



BST Insert



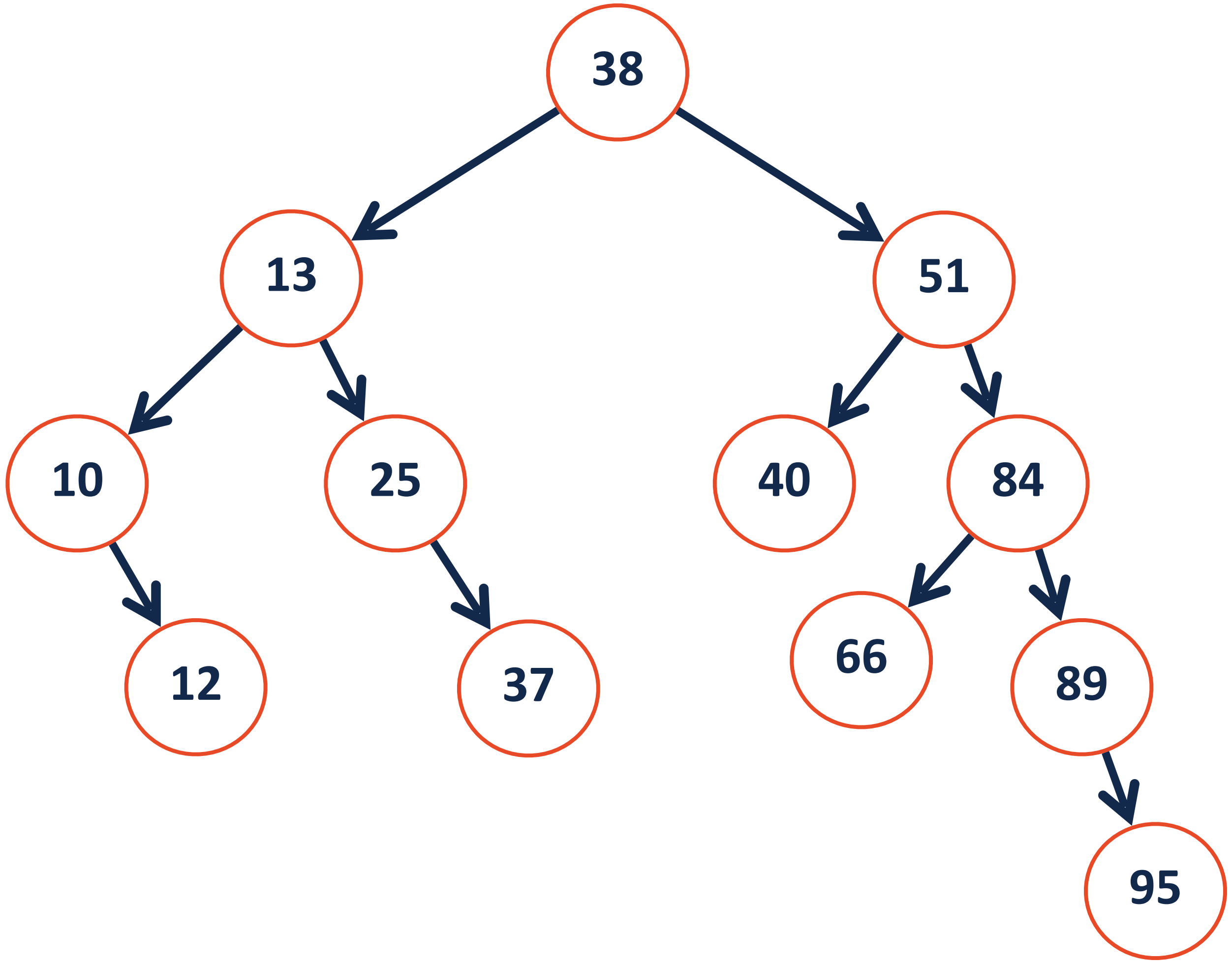
Base Case:

Recursive Step:

Combining:

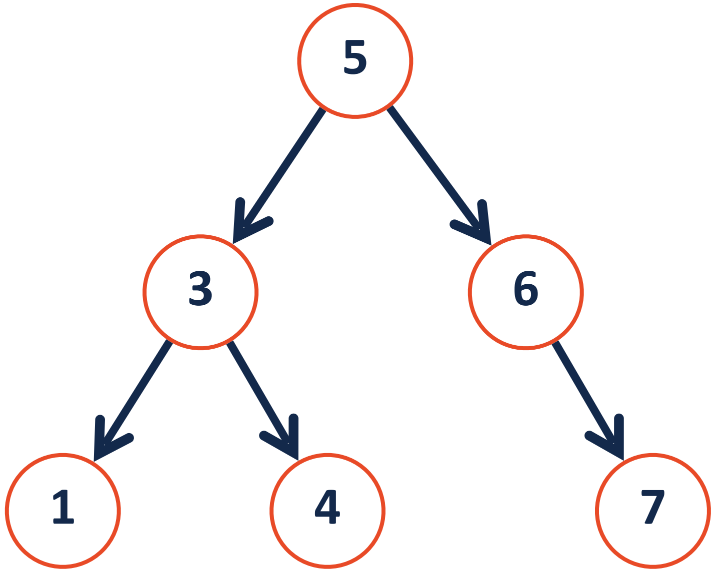
BST Insert

`insert(33)`



BST Insert

```
# inside class bst
def insert(self, key, val):
    self.root = self.insert_helper(self.root, key, val)

def insert_helper(self, node, key, val):
```

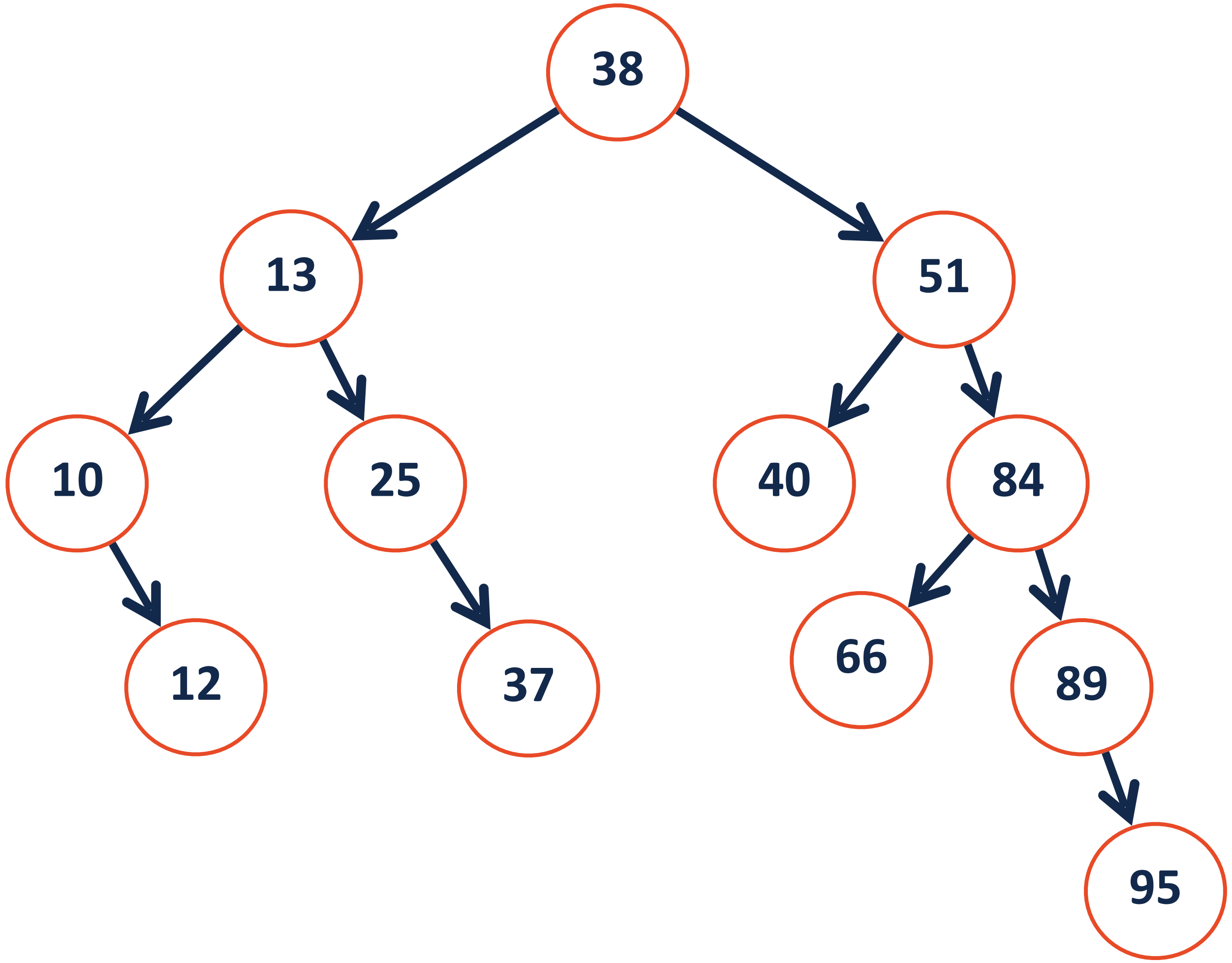


BST Insert

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

BST Remove

`remove(40)`



BST Remove

`remove(12)`

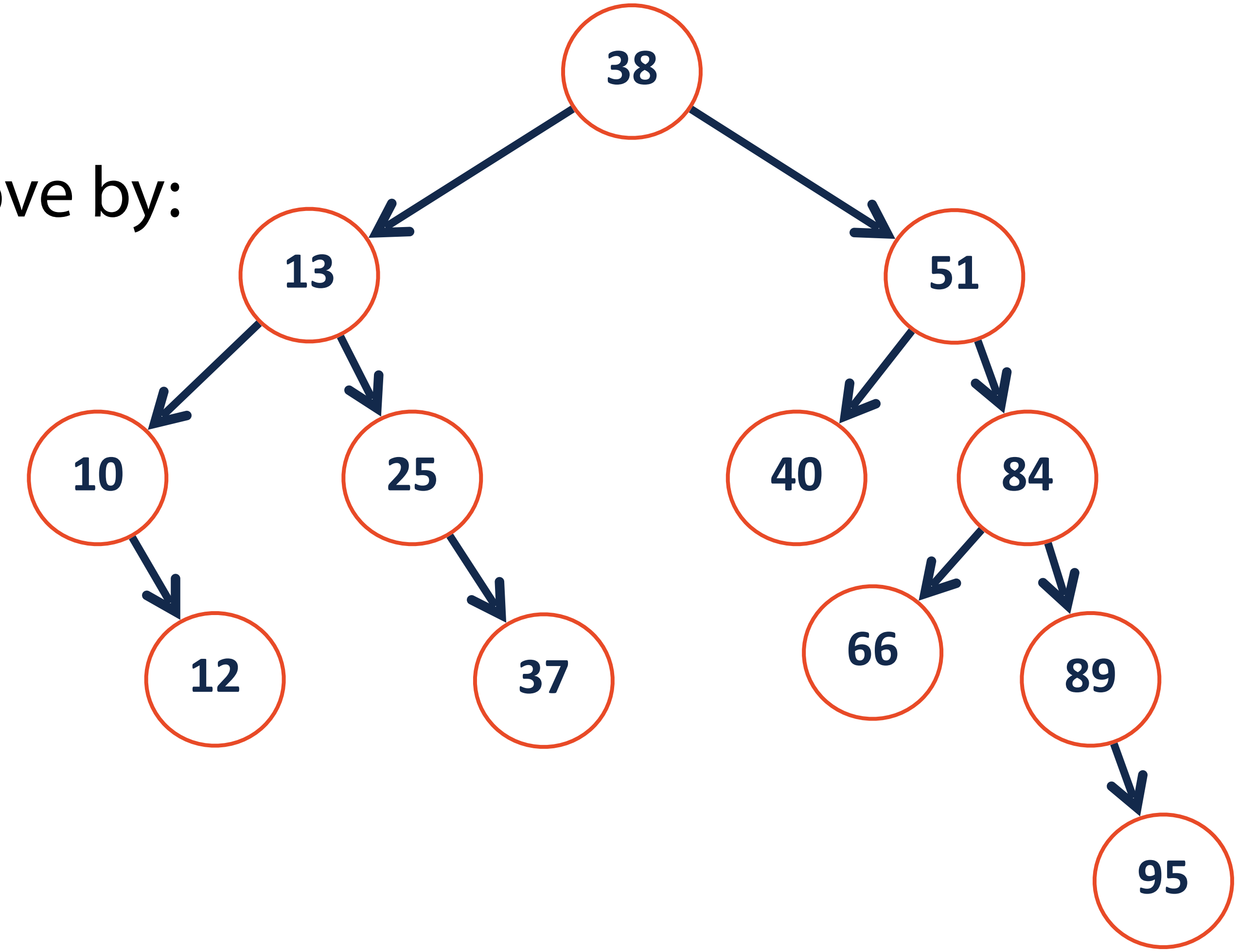
1) Find item being removed

2) Identify number of children

When we have zero children, remove by:

Set parent.**next** = None

next is either **left** or **right**



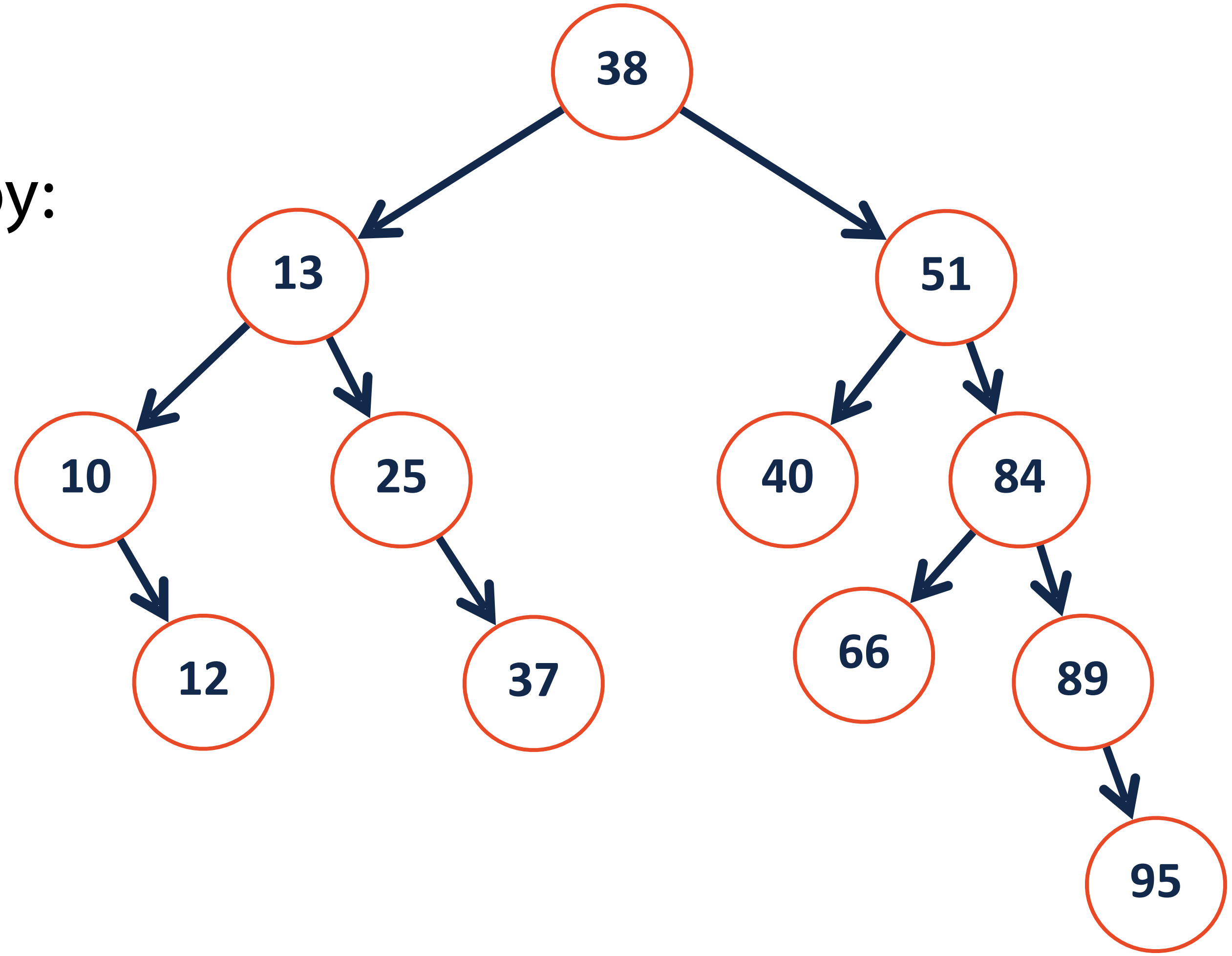
BST Remove

`remove(25)`

1) Find item being removed

2) Identify number of children

When we have one child, remove by:



BST Remove

`remove(10)`

1) Find item being removed

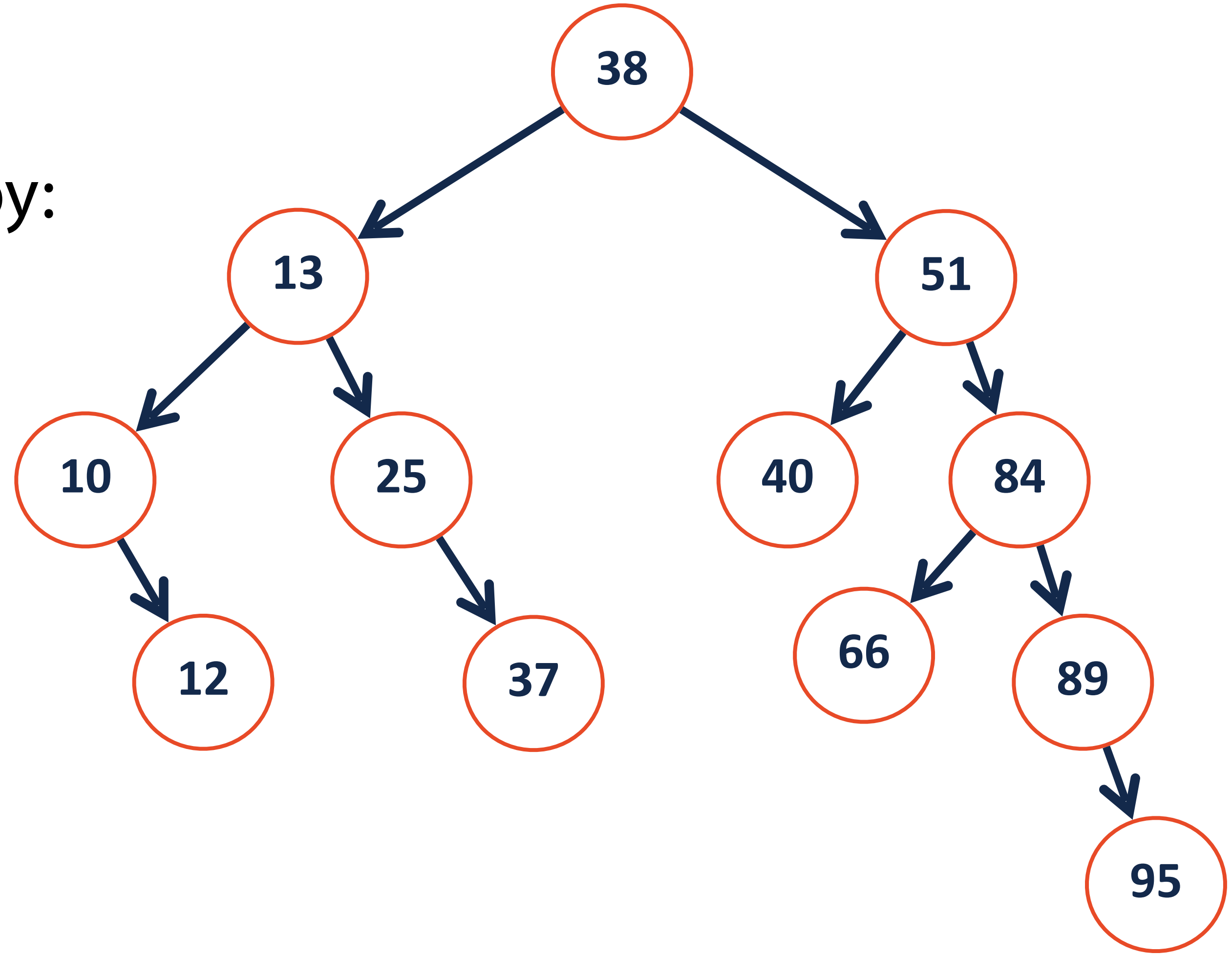
2) Identify number of children

When we have one child, remove by:

Set parent.**next1** = target.**next2**

next1 is either **left** or **right**

next2 is either **left** or **right**



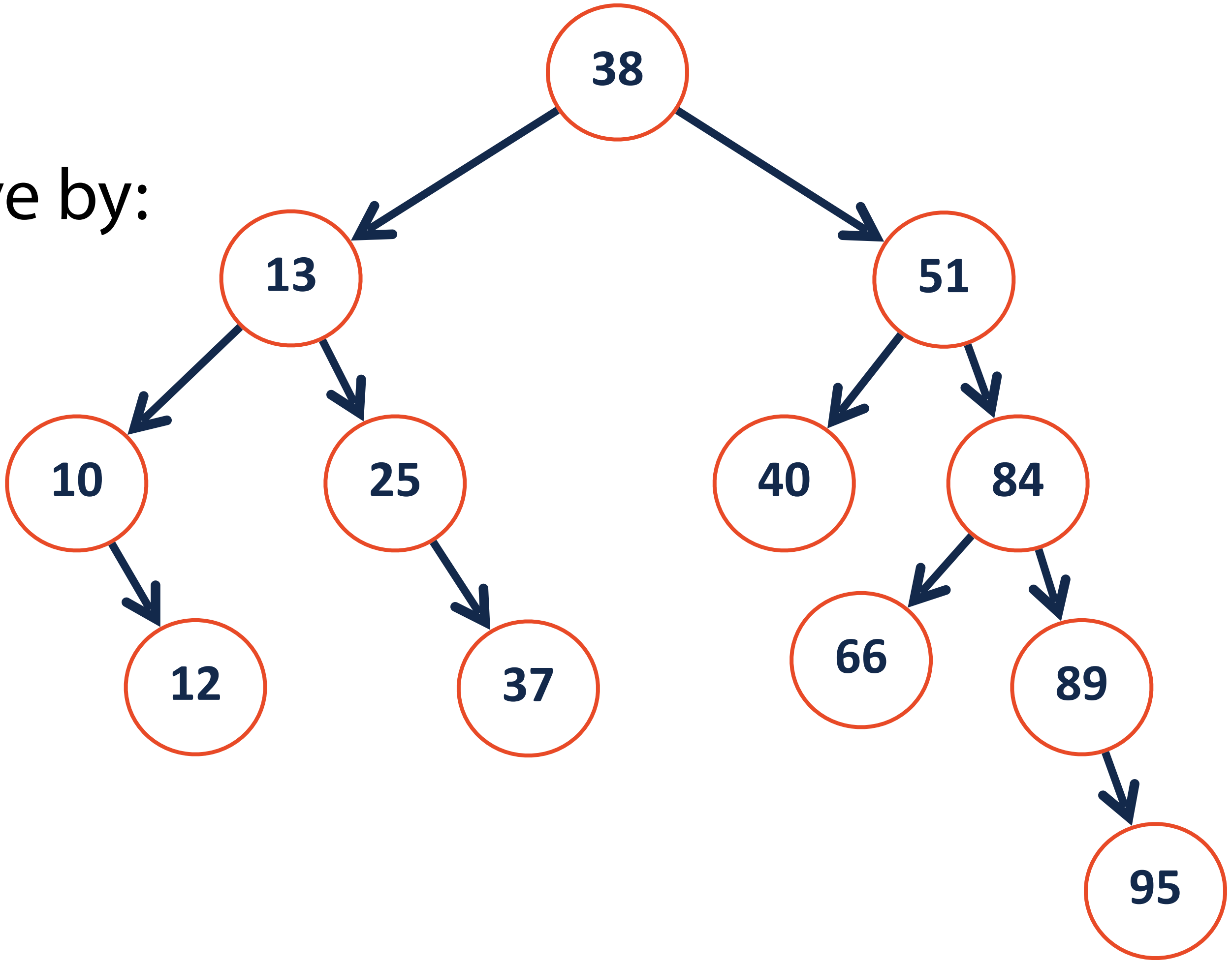
BST Remove

`remove(13)`

1) Find item being removed

2) Identify number of children

When we have two children, remove by:



BST In-Order ______

In-Order Predecessor

Rightmost left child

IOP(38) =

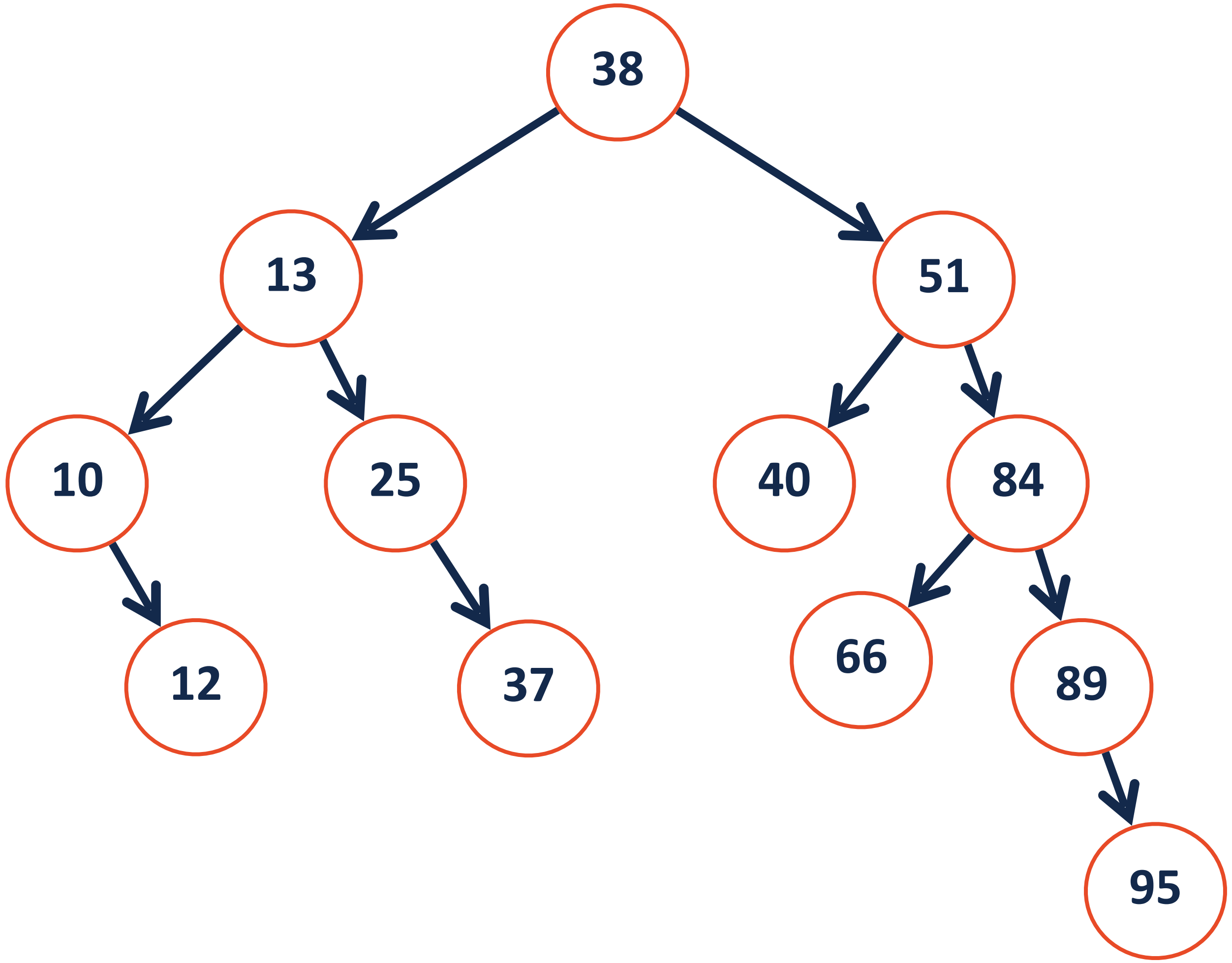
IOP(84) =

In-Order Successor

Leftmost right child

IOS(38) =

IOS(84) =



BST Remove

`remove(13)`

1) Find item being removed

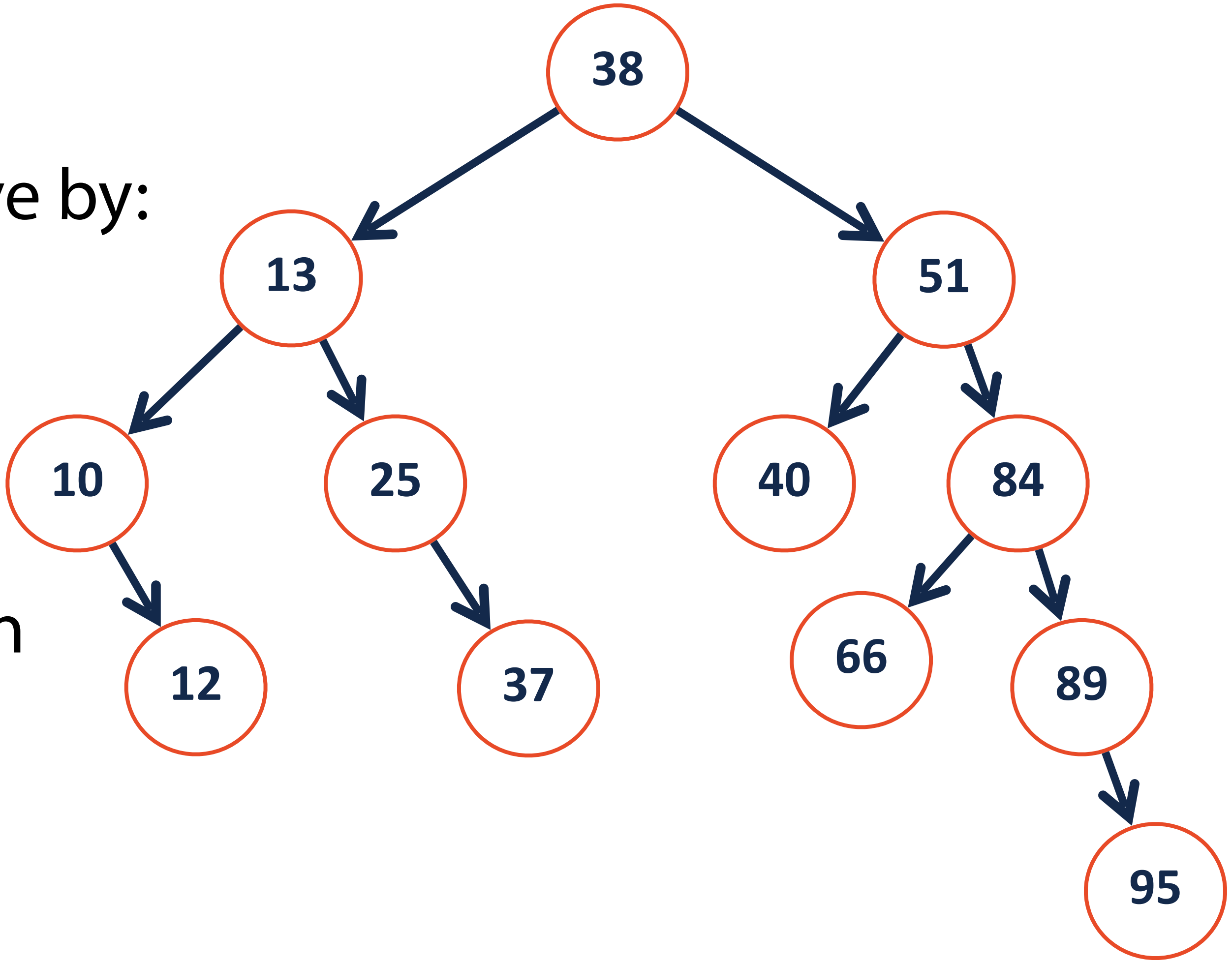
2) Identify number of children

When we have two children, remove by:

3) Find the IOP / IOS

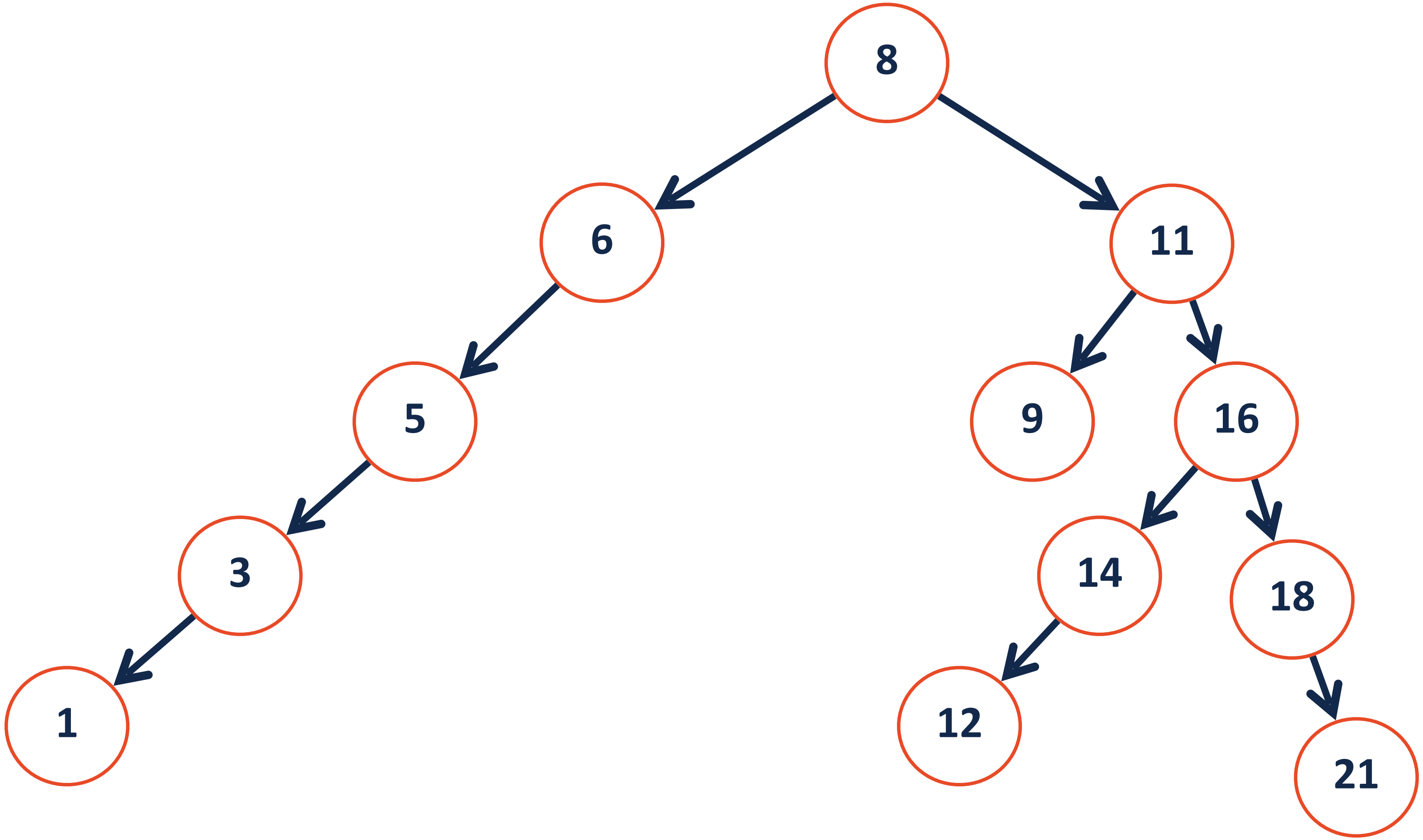
4) Swap target with IOP / IOS

5) Remove target at its new location

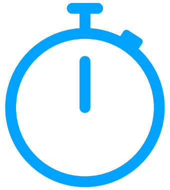


BST Remove

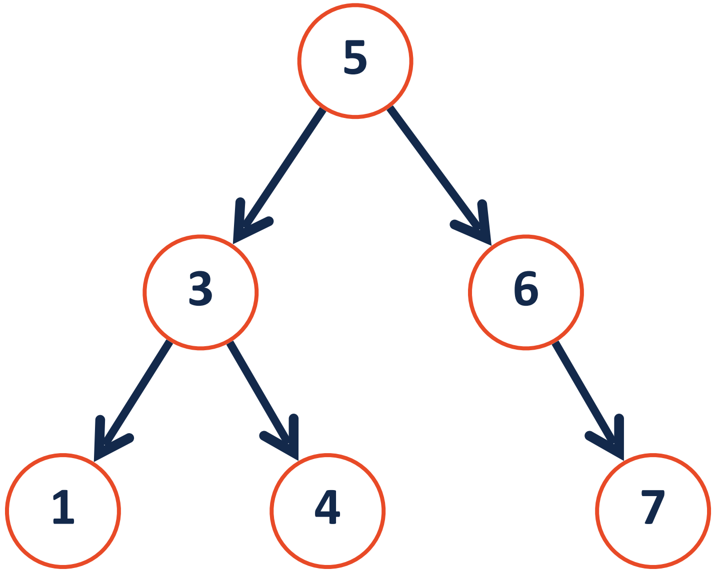
What will the tree structure look like if we remove node 16 using IOS?



BST Remove



```
def remove(self, key):
    self.root = self.remove_helper(self.root, key)

def remove_helper(self, node, key):
```



BST Analysis – Running Time

Operation	BST Worst Case
find	
insert	
delete	
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 ?

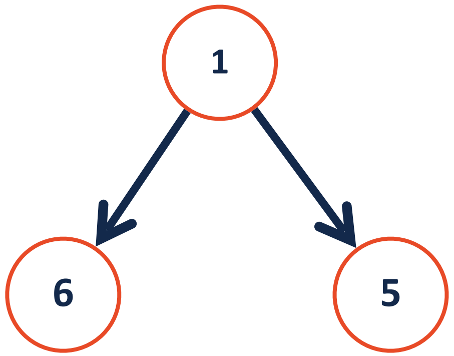
BST Analysis

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

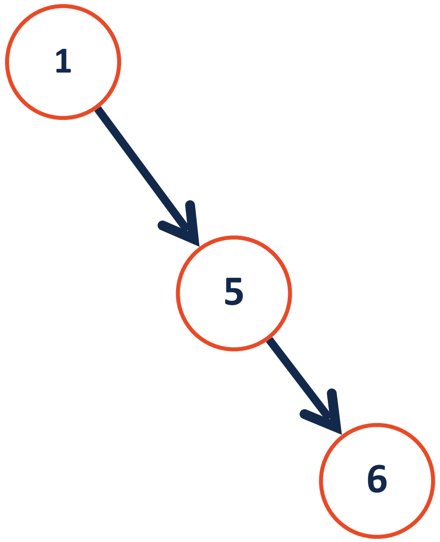
BST Analysis

A BST of n nodes has a height between:

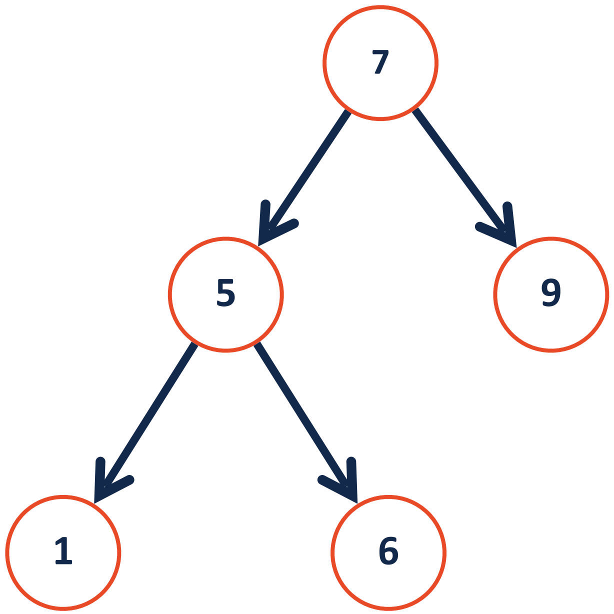
Lower-bound: $O(\log n)$

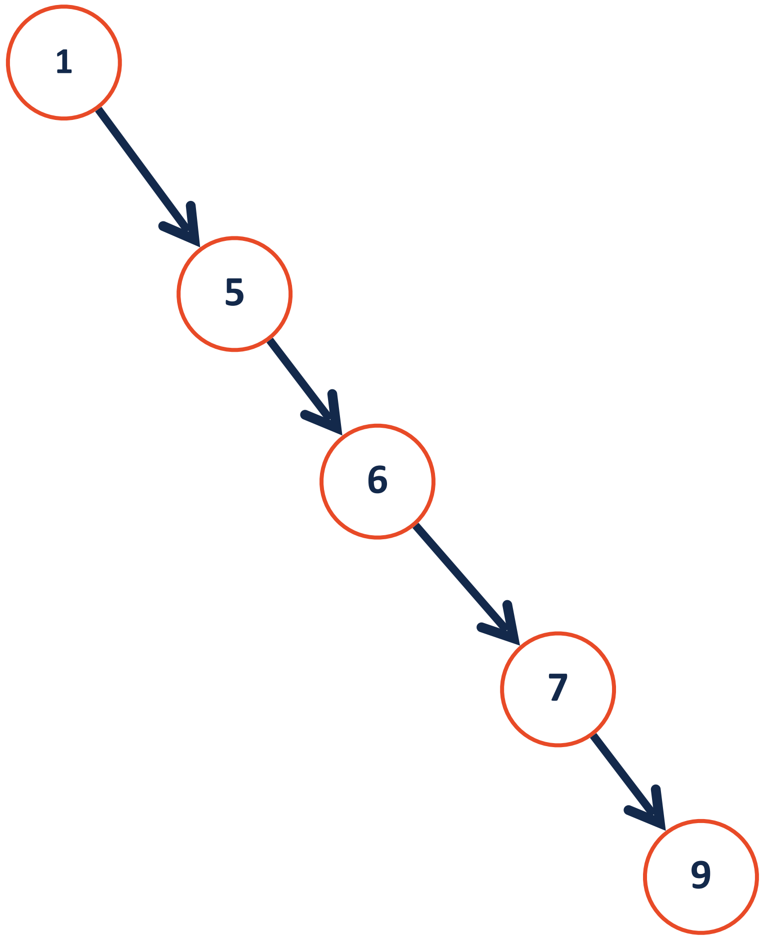


Upper-bound: $O(n)$



Limiting the height of a tree





Option A: Correcting bad insert order

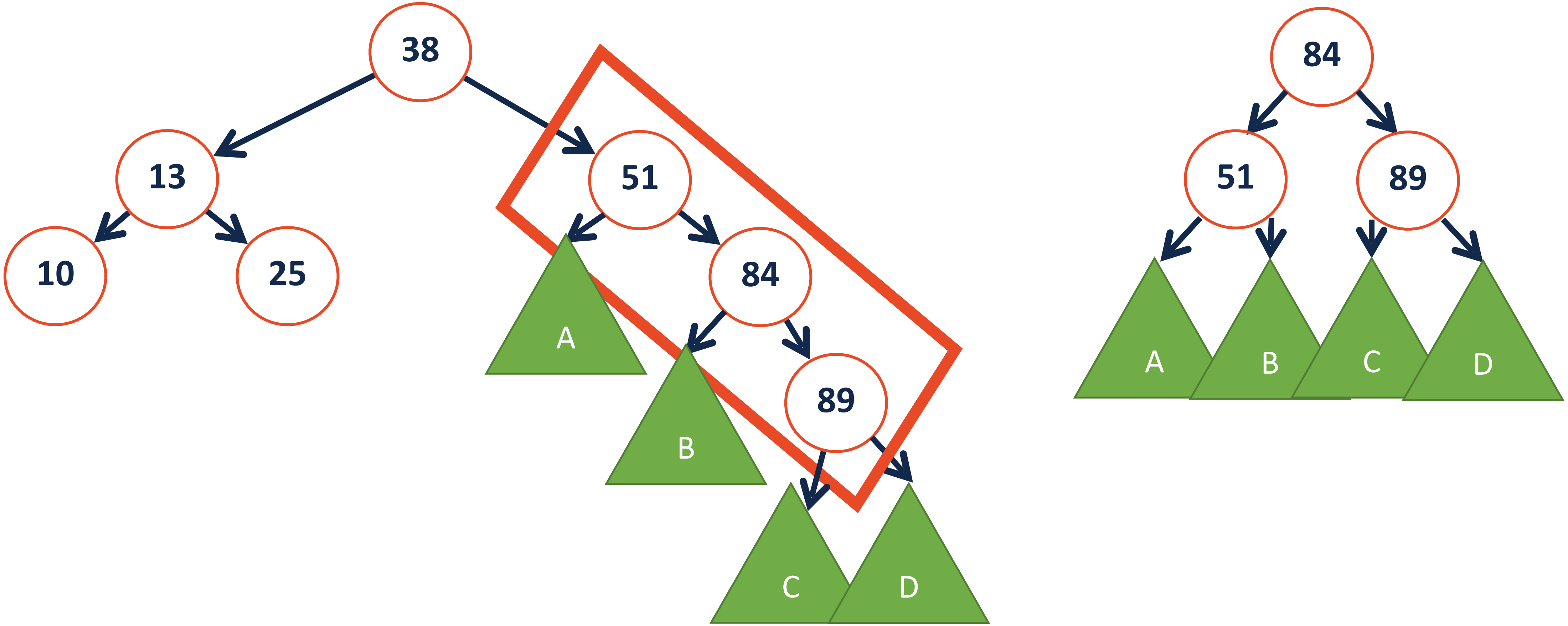
The height of a BST depends on the order in which the data was inserted

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

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

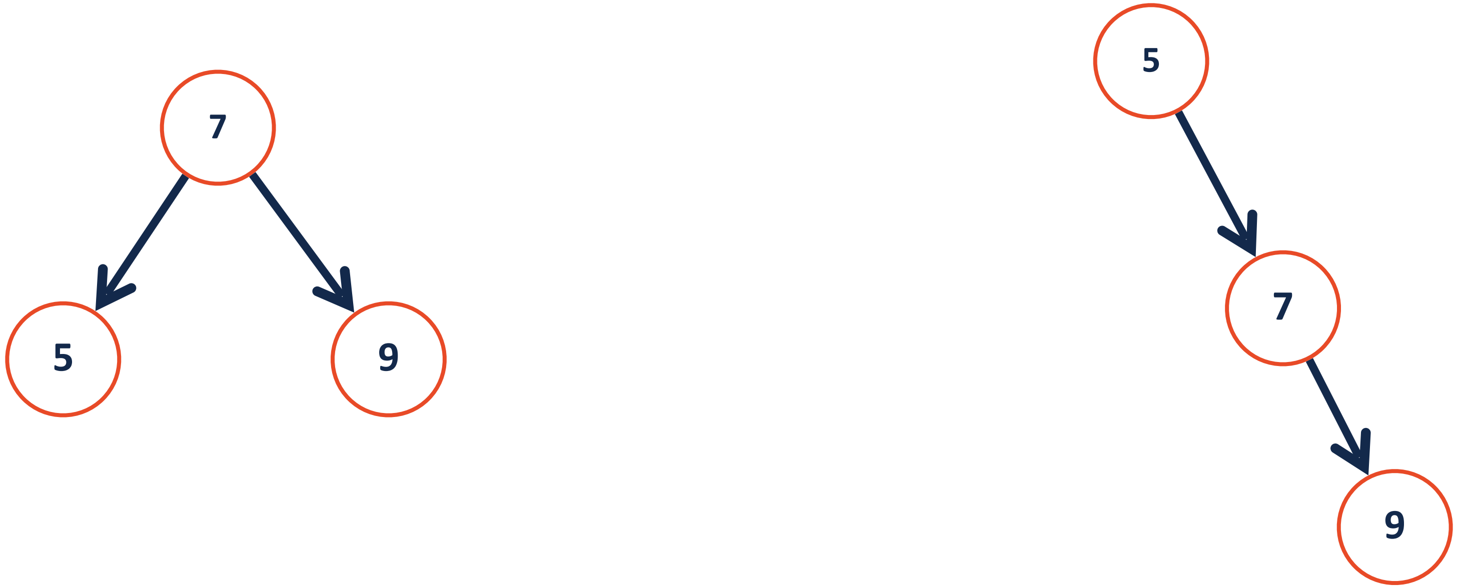
AVL-Tree: A self-balancing binary search tree

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



Height-Balanced Tree

What tree is better?

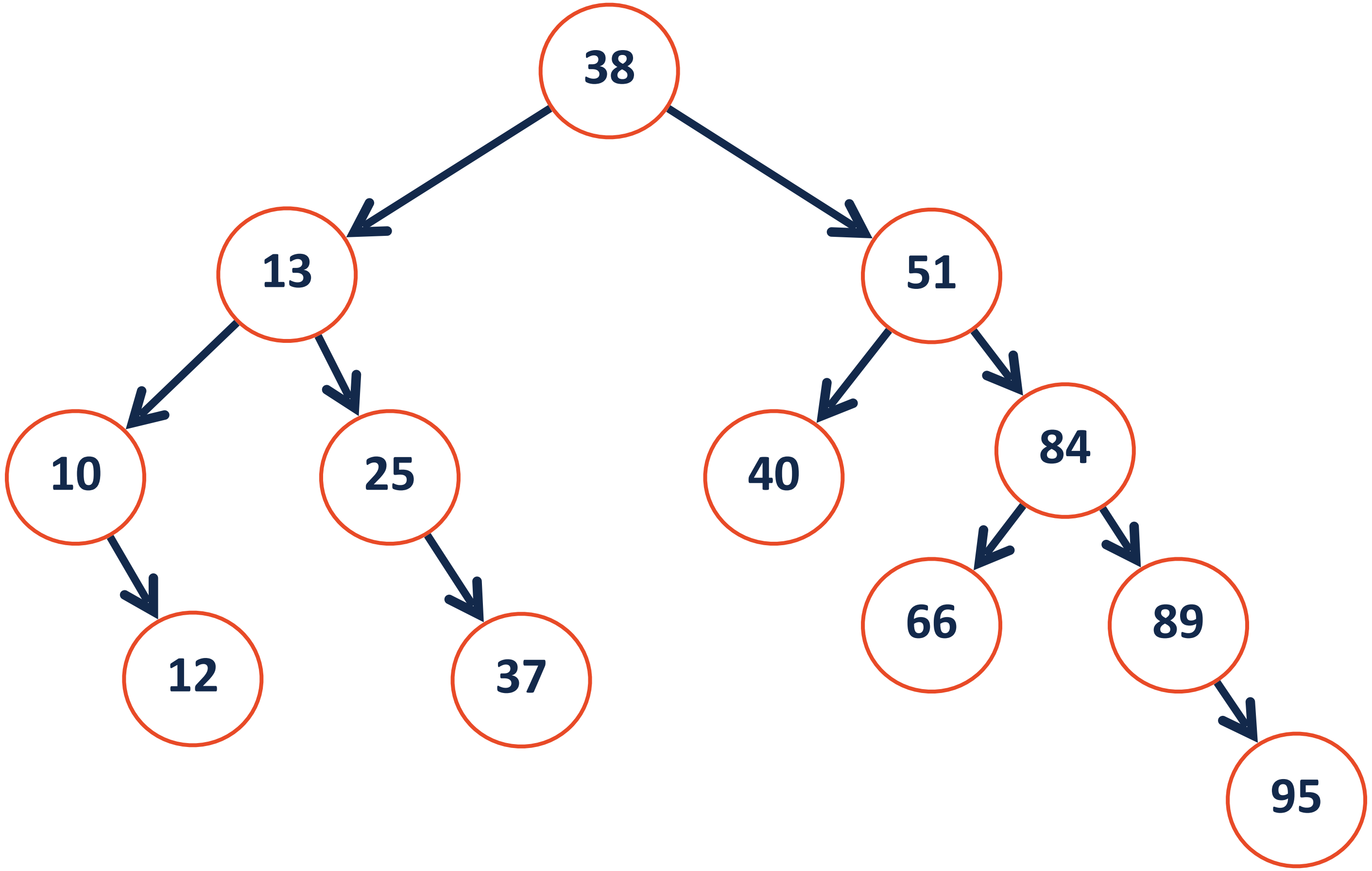


Height balance: $b = height(T_R) - height(T_L)$

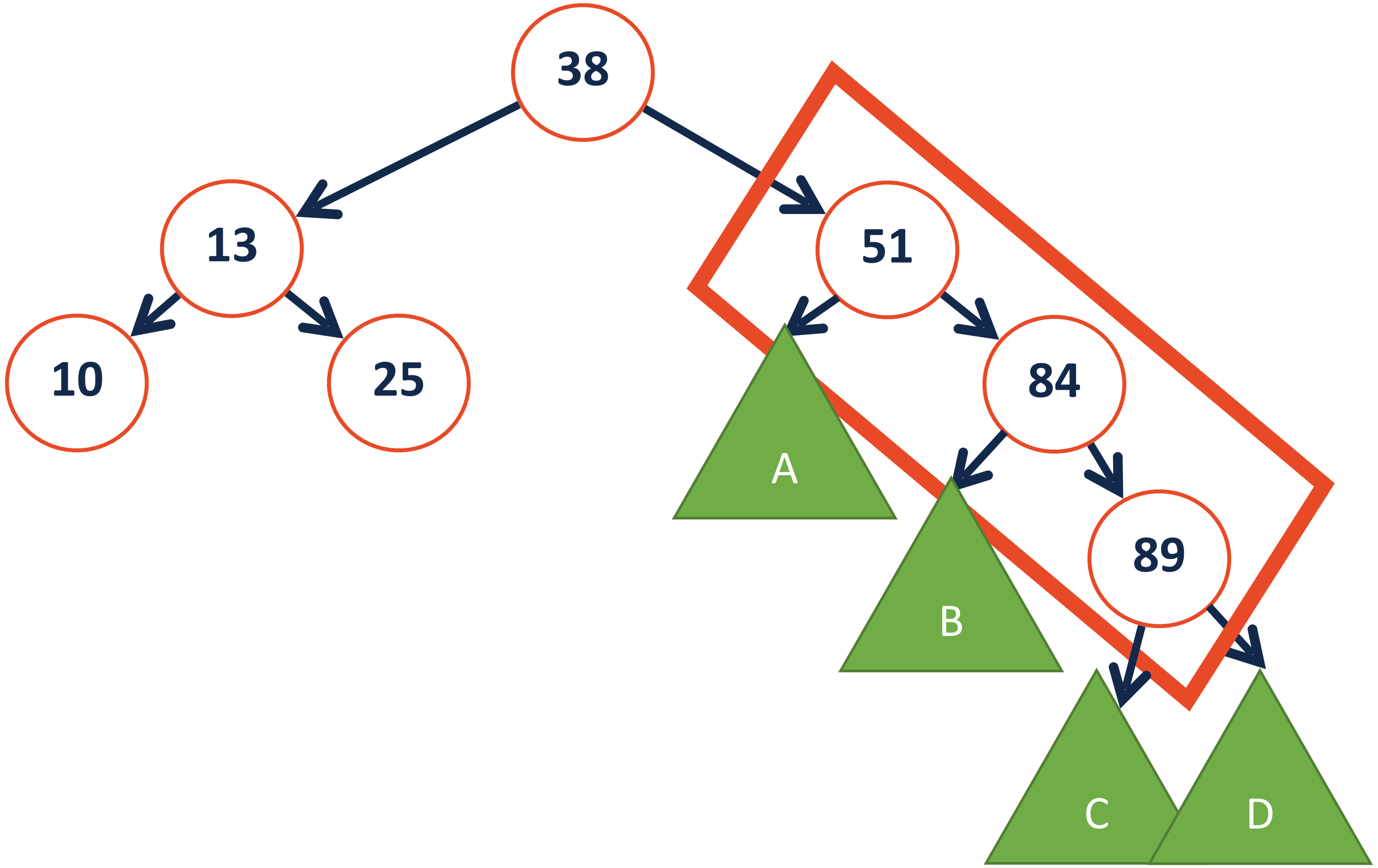
A tree is "balanced" if:

BST Rotations (The AVL Tree)

We can adjust the BST structure by performing **rotations**.



Left Rotation



Left Rotation