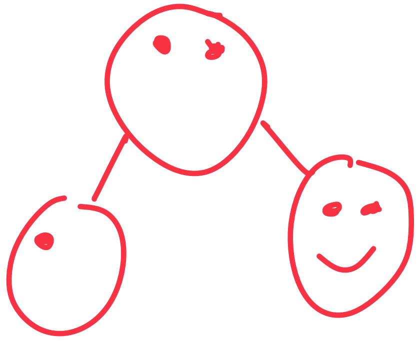
Algorithms and Data Structures for Data Science
Binary Search Tree

CS 277
Brad Solomon

February 28, 2024



University of Illinois Urbana-Champaign
Department of Computer Science

Learning Objectives

Review understanding of Binary Trees

Practice tree traversals

"Binary tree"

???

Introduce the dictionary ADT

"Binary Search tree"

Extend ADT to Binary Search Trees

Practice recursion in the context of trees

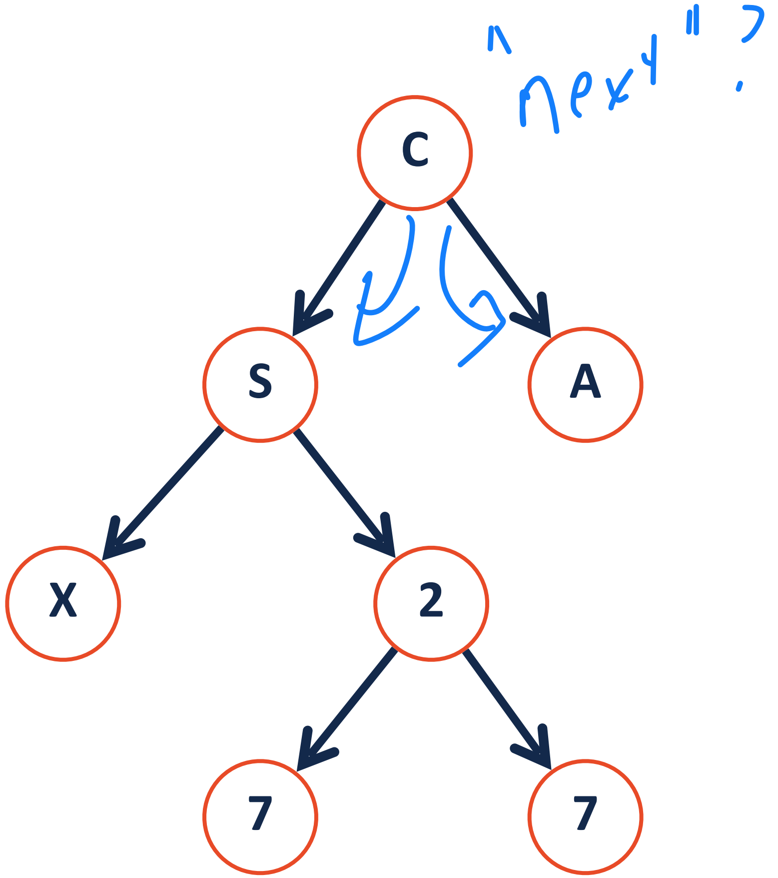
(Binary) Tree Recursion

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

find or get specific values

Binary Tree Insert and Remove

Last class we implemented insert and remove where we are given the parent node and direction (allowing us to reach the node of interest)

Insert was a lot like what previous data structure:

1) the necessary info is very similar

2) the runtime was similar O(1) if given node so I need this node

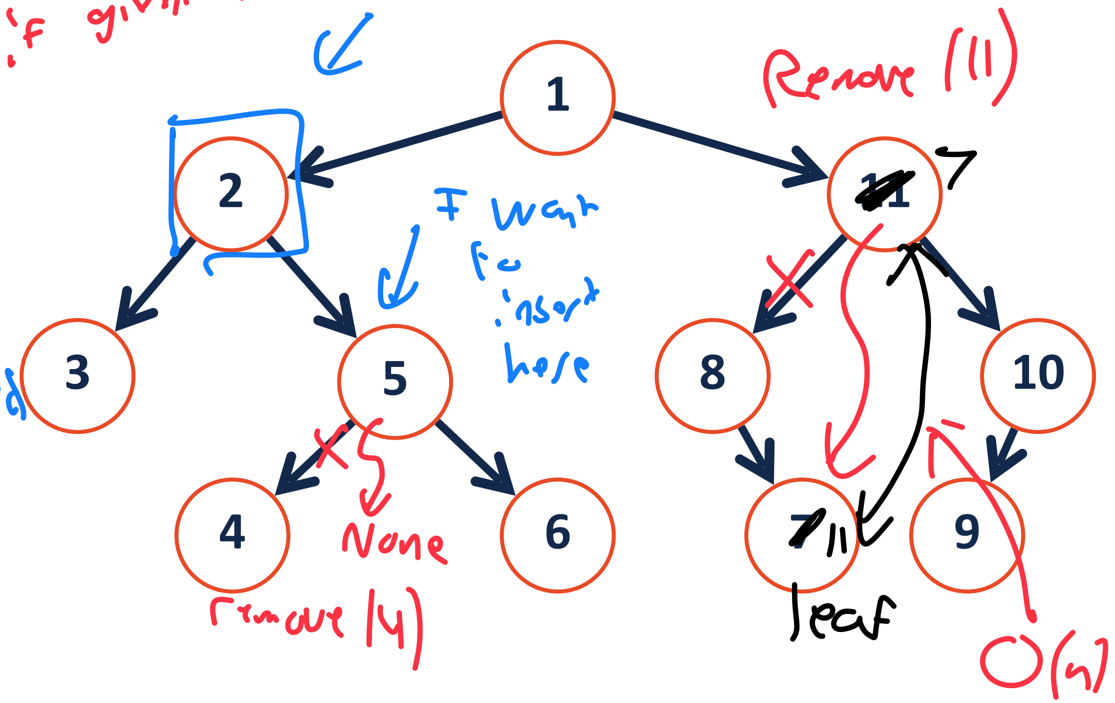
↳ If I need to find parent O(n)

Remove has one bad case, which was:

0 or 1 child at node to be removed

↳ O(1)

2 child removal → O(n)



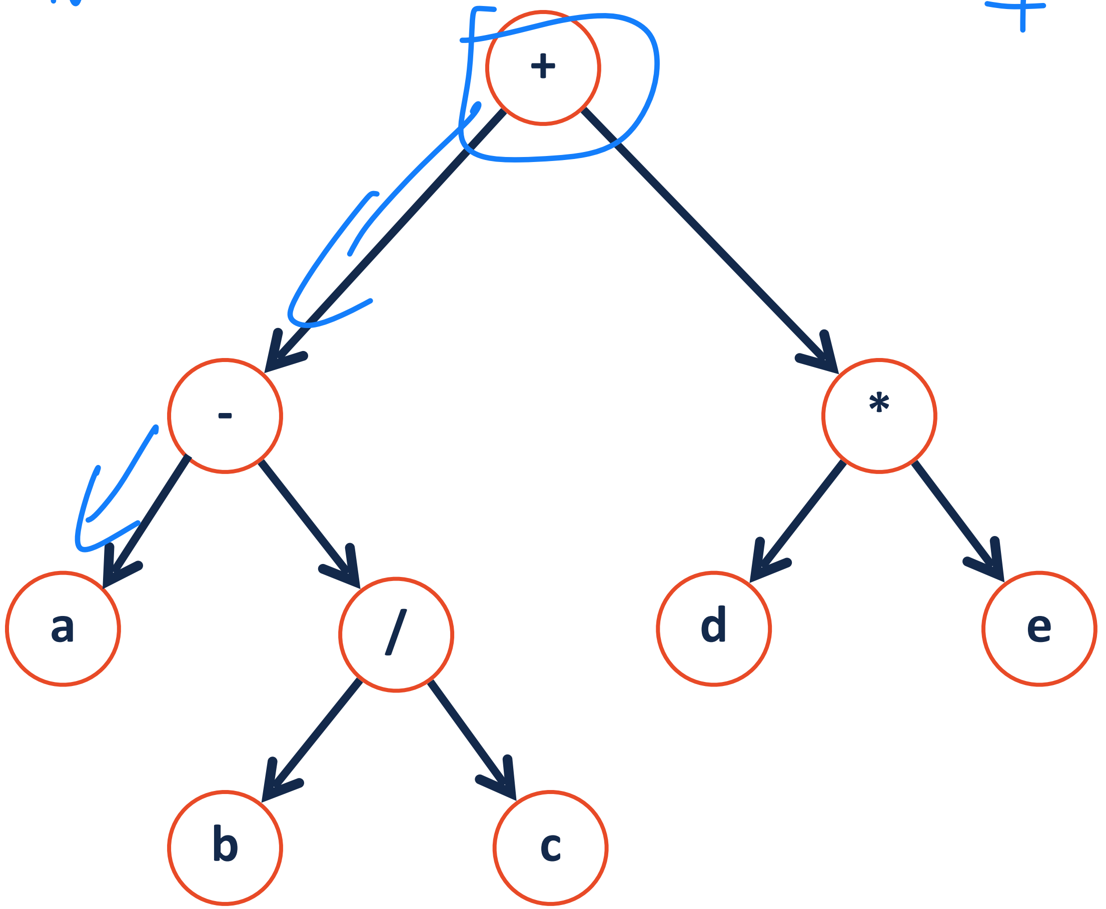
Tree Traversal

A **traversal** of a tree T is an ordered way of visiting every node once.

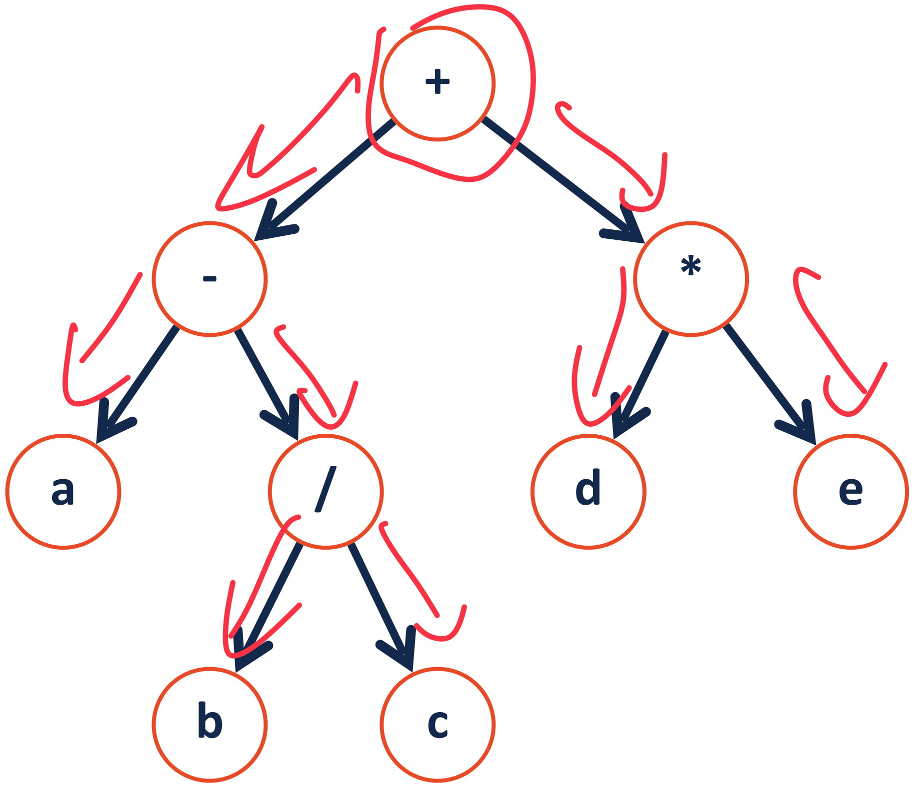
→ O(n)

1) Look at / process Node
   ↳ print
2) Recurse left
3) Recurse right

+ - a



Pre-order Traversal

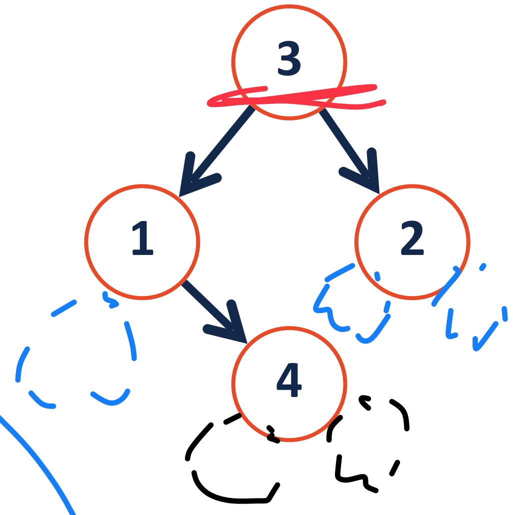



```
def preorderTraversal(node):
    if node:

        print(node.val)

        preorderTraversal(node.left)

        preorderTraversal(node.right)
```

process

Pre-order: + - a / b c * d e

Pre-order Traversal Visualized



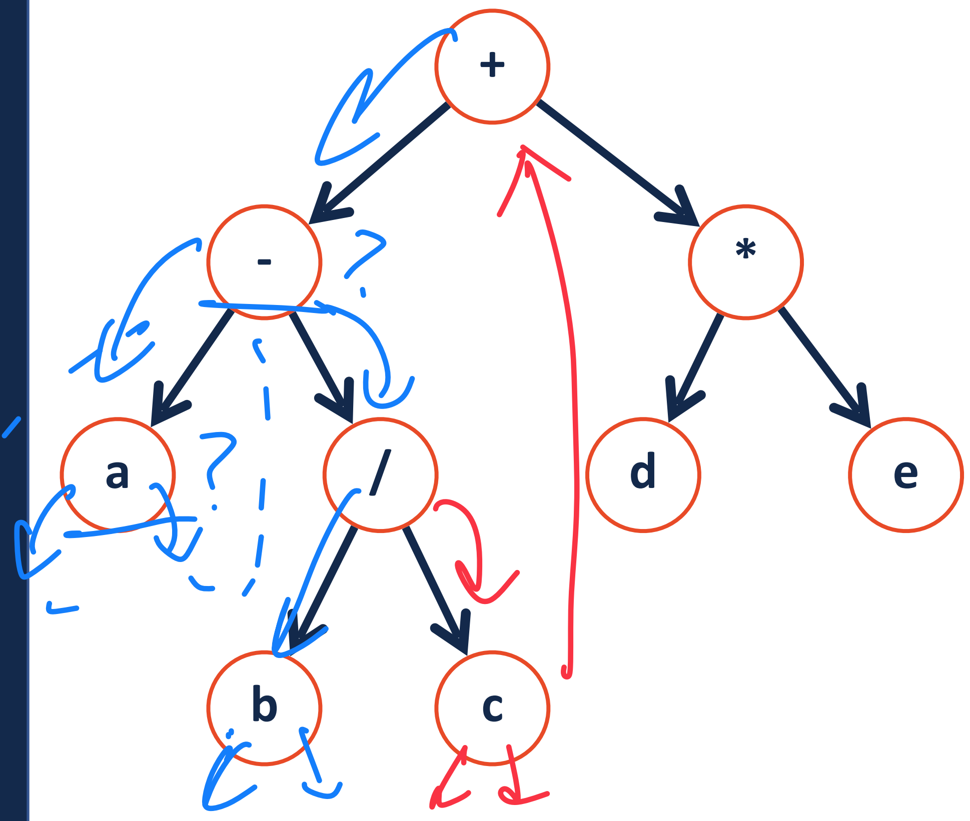

```
def preorderTraversal(<3>):
    if node:
        print(<3>.val)

        preorderTraversal(<3>.left)

        preorderTraversal(<3>.right)
```

```
def preorderTraversal(<1>):
    if node:
        print(<1>.val)

        preorderTraversal(<1>.left)

        preorderTraversal(<1>.right)
```

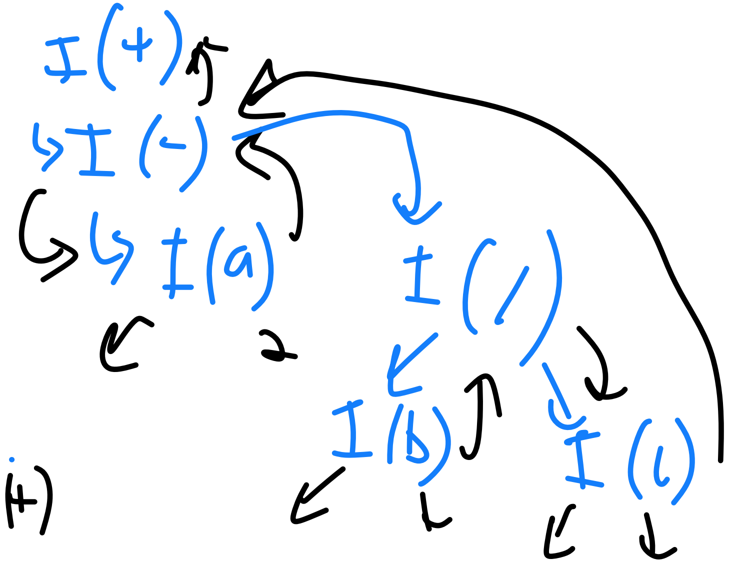
```
def preorderTraversal(<4>):
    if node:
        print(<4>.val)

        preorderTraversal(<4>.left)

        preorderTraversal(<4>.right)
```

```
def preorderTraversal(<2>):
    if node:
        print(<2>.val)

        preorderTraversal(<2>.left)

        preorderTraversal(<2>.right)
```

In-order Traversal



1) Recurse left
2) Process Node   mid/current Node
3) Recurse Right

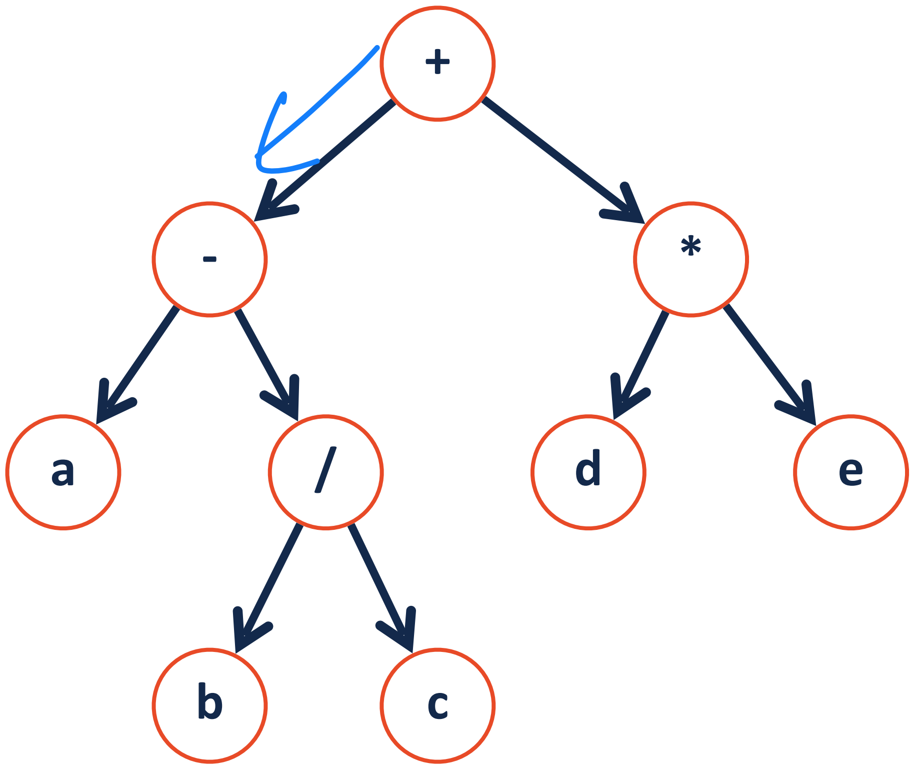


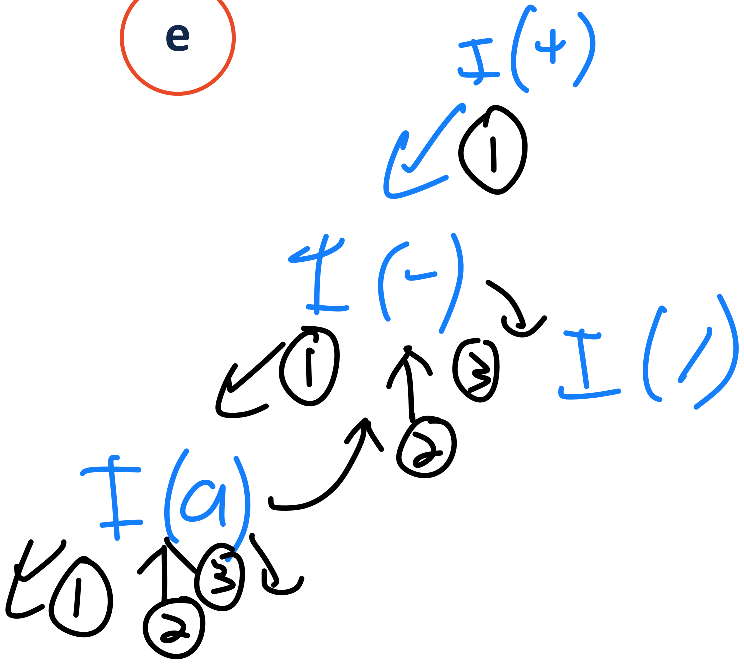
printed all left of (+)

In-order: a – b / c + d * e

In-order Traversal

1) Recurse left
2) Process Node — mid/current Node
3) Recurse Right

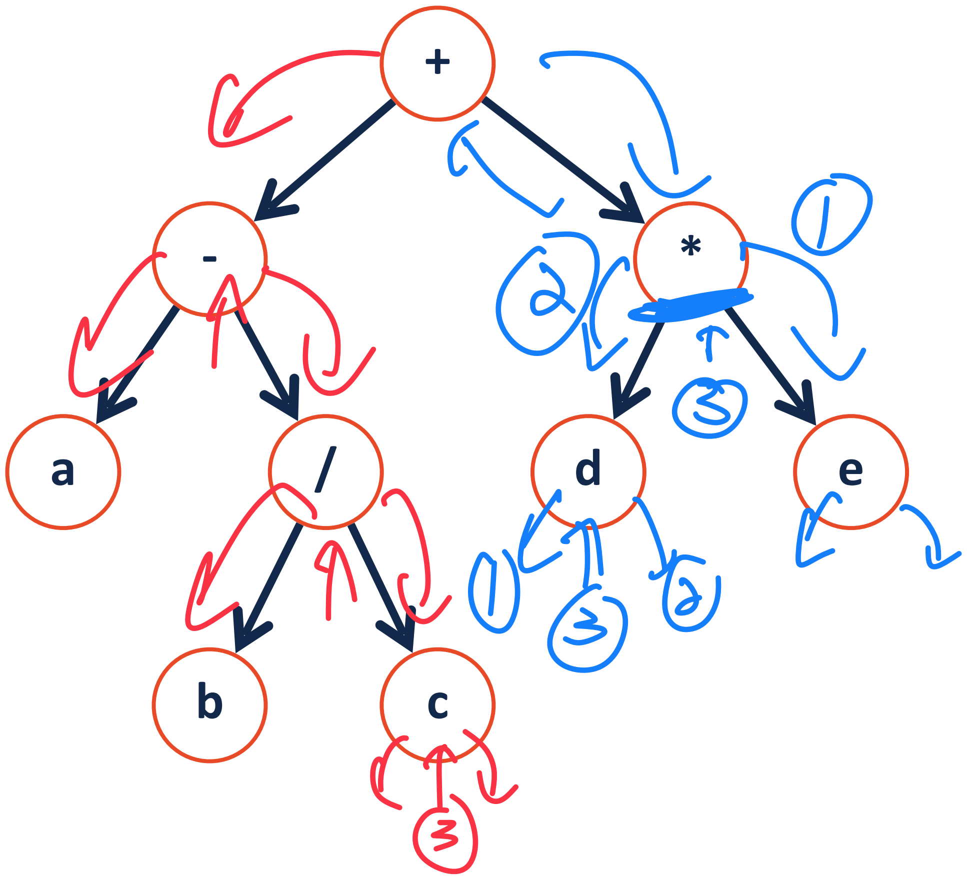


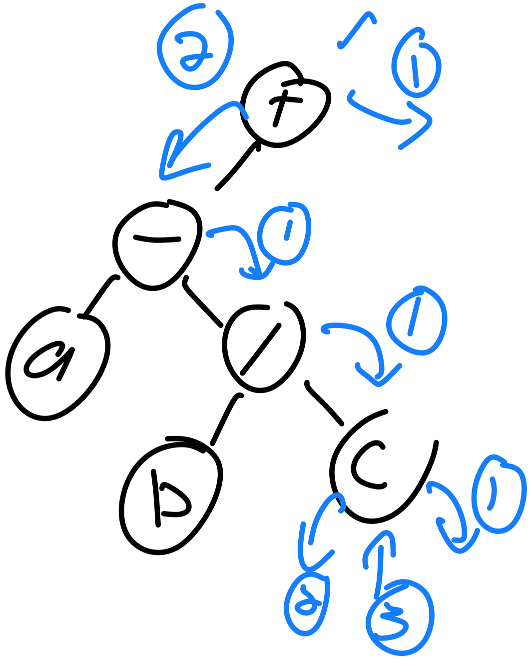


In-order: a —

Post-order Traversal

1) recurse (right)
2) recurse (left)
3) process (current)



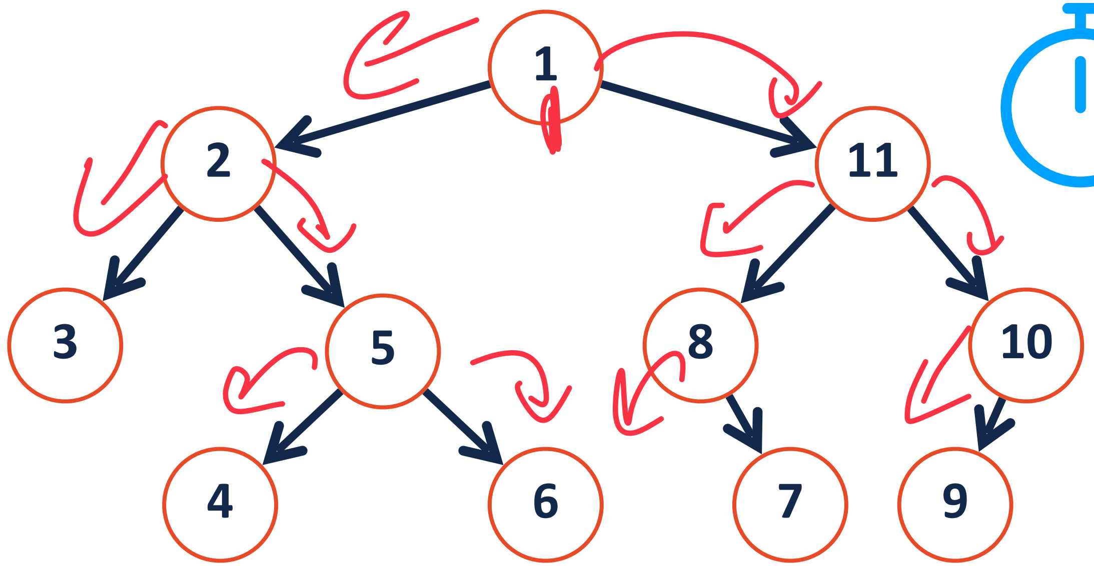


Post-order: e d * c b / a - +

Tree Traversals

Lets practice our traversals!

Left
myself
Right



Pre-order:

In-order: 3 2 4 5 6 1 8 7 11 9 10

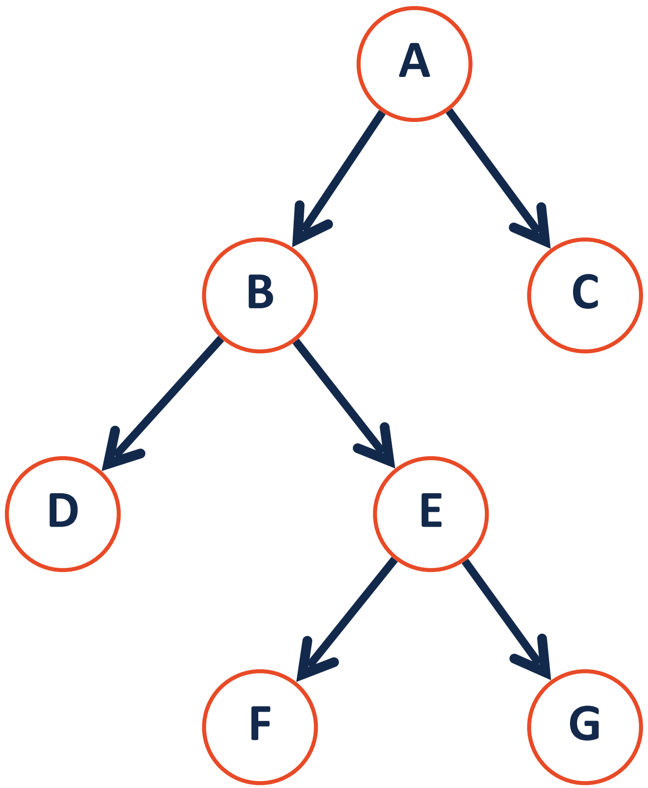
Post-order:

Traversal vs Search

Traversal – Always looks at every Node
↳ O(n)

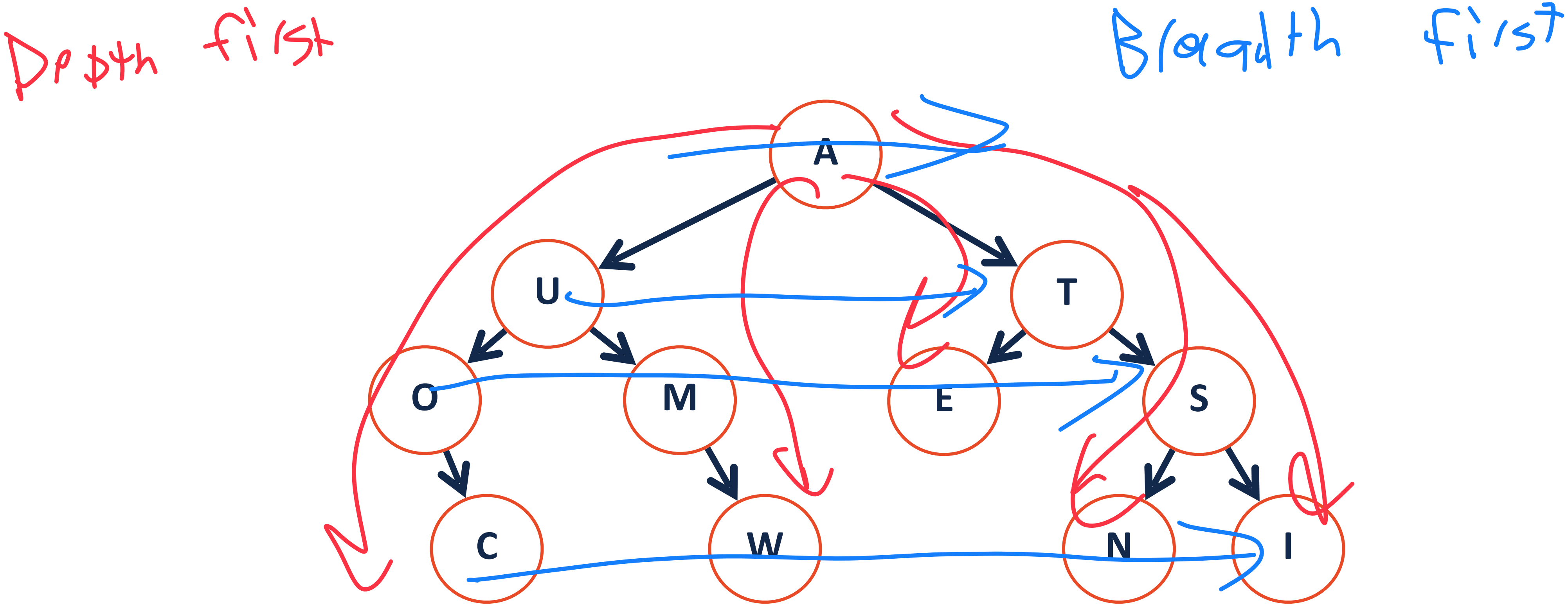
O(n)
worst case
Search – Might look at every Node

D→ D→ D→ D
X



Searching a Binary Tree

There are two main approaches to searching a binary tree:



Depth First Search

1, 11, 10, 9, 8, 7

Explore as far along one path as possible before backtracking

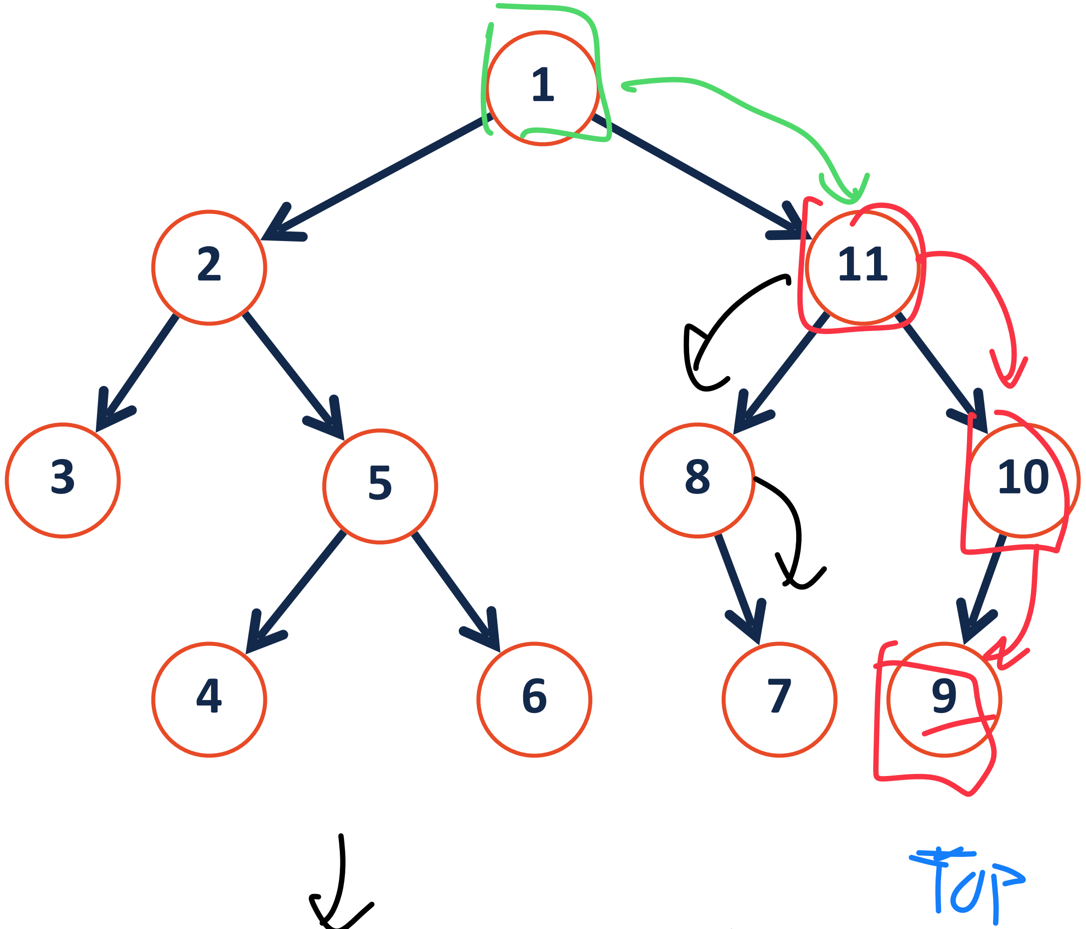
1) Make stack. Stack.push([1])

2) while stack is not empty:

n = stack.pop()

2.5) Add children of n to stack

2.6?) Process n ["print"]



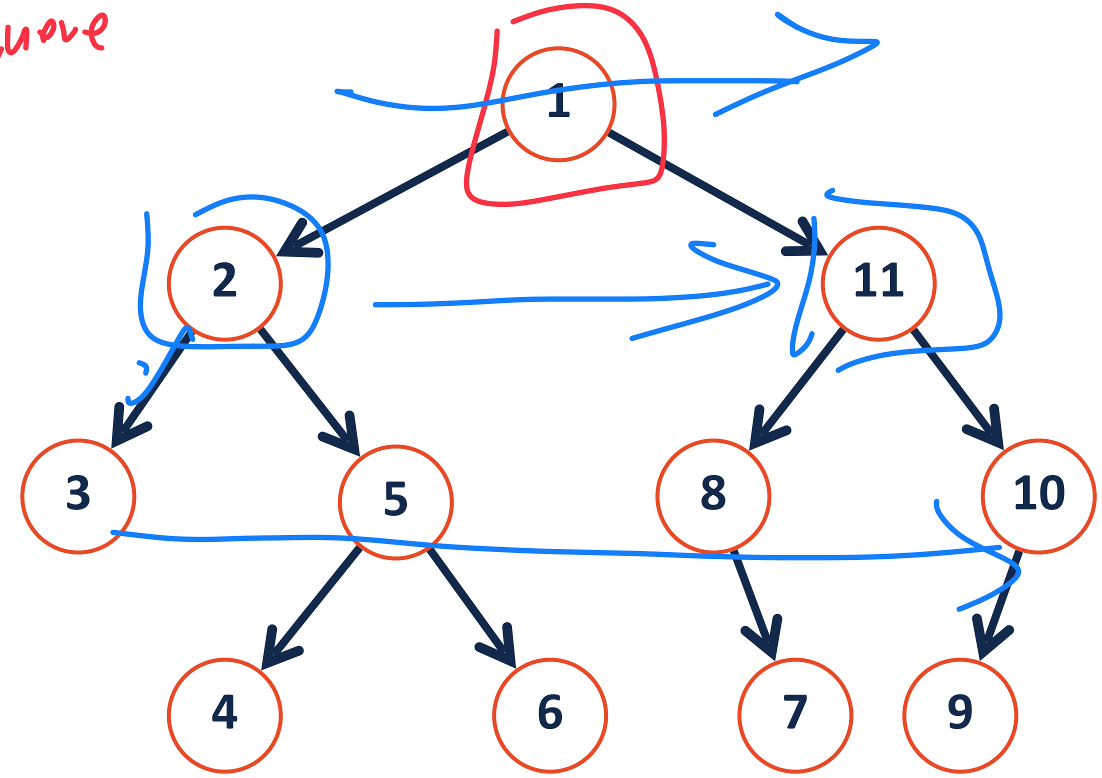
1 2 11 8 10 9 7

Breadth First Search

Fully explore depth i before exploring depth i+1

Is take stack code, replace w/ queue

1) Make queue, enqueue(root)
2) while queue not empty
2.5) n = q.dequeue()
process Node / add children to q



Traversal vs Search II

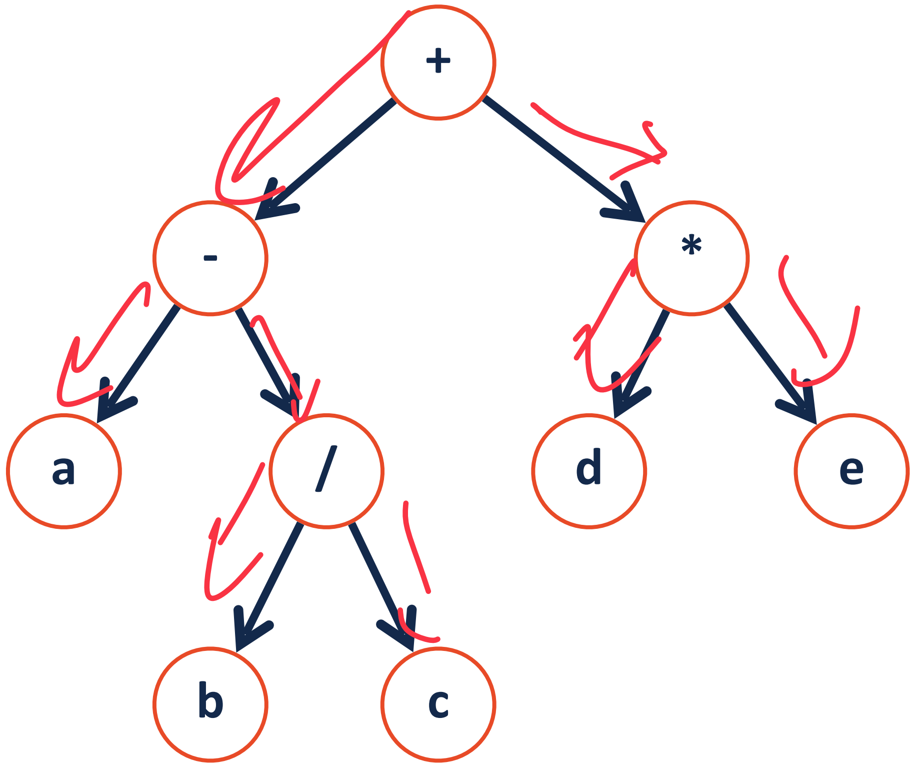
Pre-order, in-order, and post-order are three ways of doing which search?

Depth-first search strategies

Pre-order: + - a / b c * d e

In-order: a - b / c + d * e

Post-order: a b c / - d e * +



Level-Order Traversal

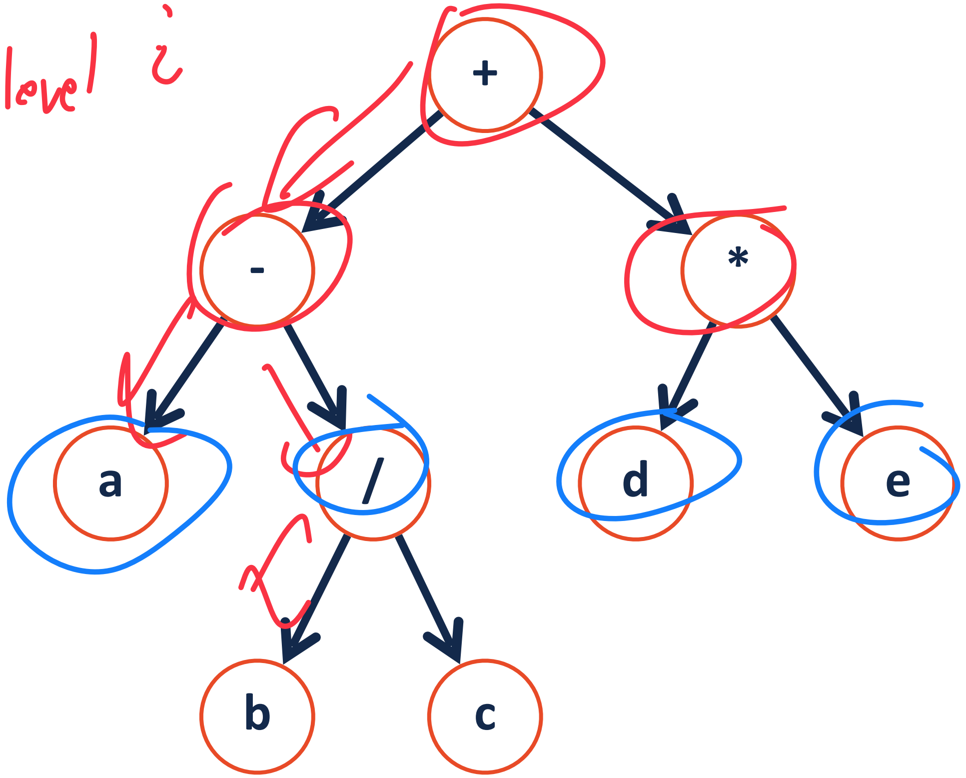
A tricky recursive implementation but an easier queue implementation!

It's possible

1) Recurse through tree but only print level i

i += 1

2) adjust code to return recursive list at depth



Level-order:

What search algorithm is best?

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?

Iterative deepening DFS

↳ DFS but limit height. Expand if necessary

Neither really work!

DFS → ∞ branch

BFS

level 1: 31

level 2: 31^2

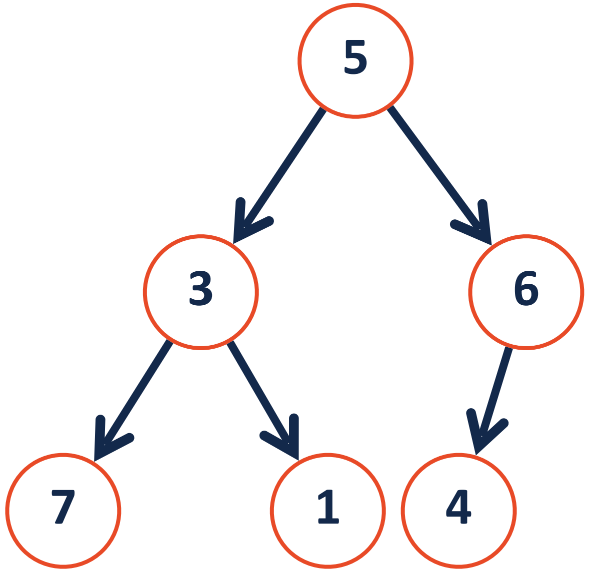
level 3: 31^3

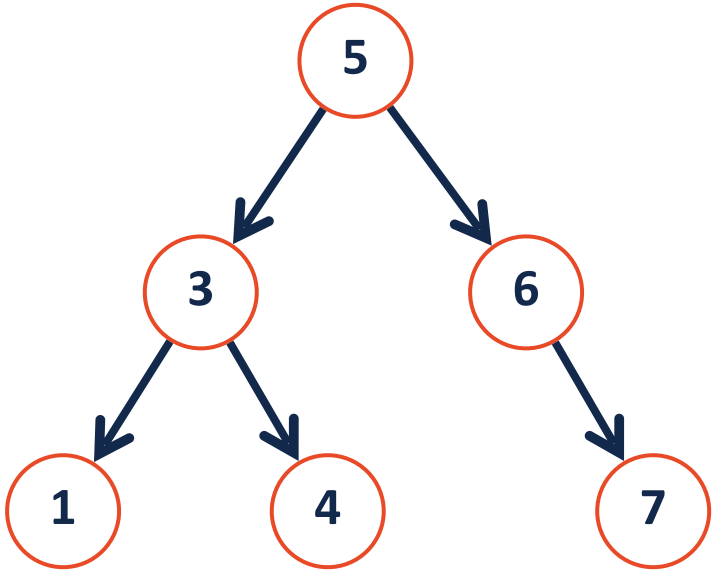
Memory Problems

Improved search on a binary tree

5	3	6	7	1	4

1	3	4	5	6	7



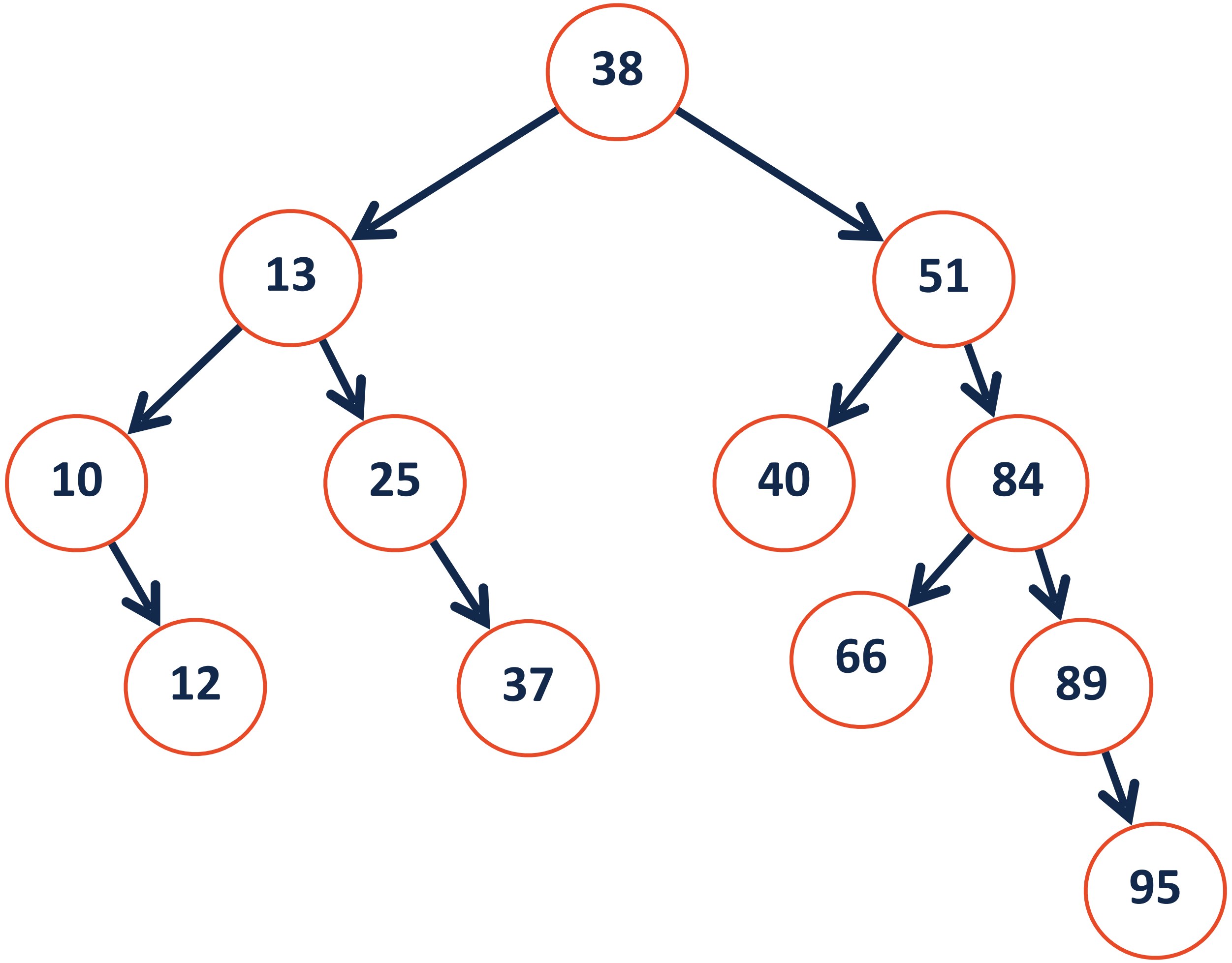


Binary Search Tree (BST)

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

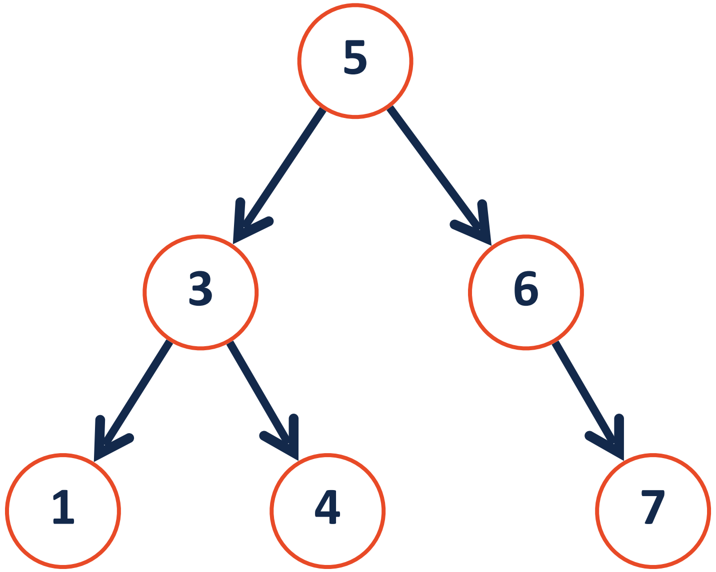
Flight Number → Arrival Information

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

Constructor: Build a new (empty) tree

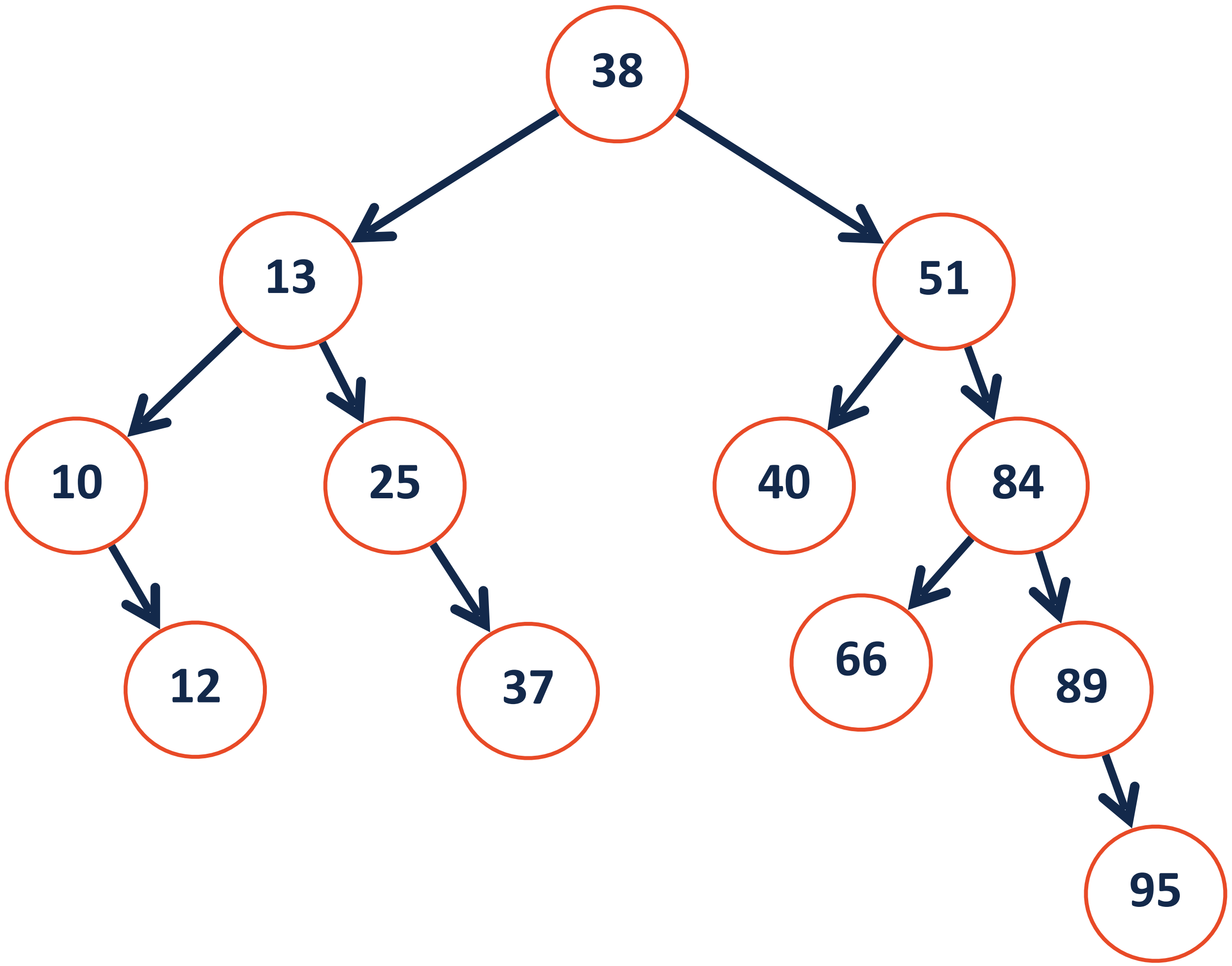
Insert: Add an object into tree

Remove: Remove a specific object from tree

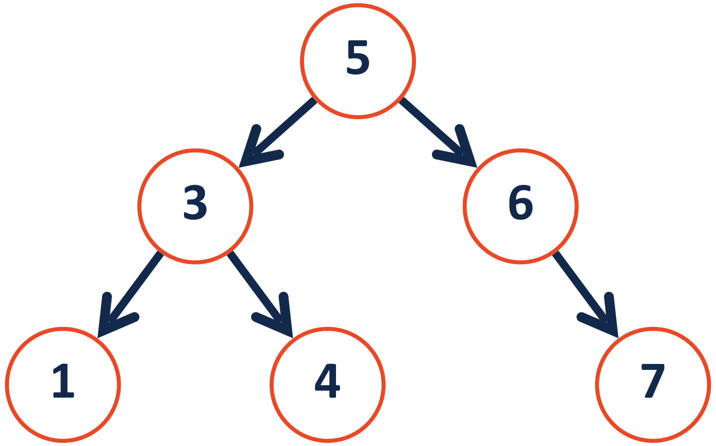
Traverse: Visit every node in tree (all objects)

Search: Find a specific object in the tree

BST In-Order Traversal



BST Insert



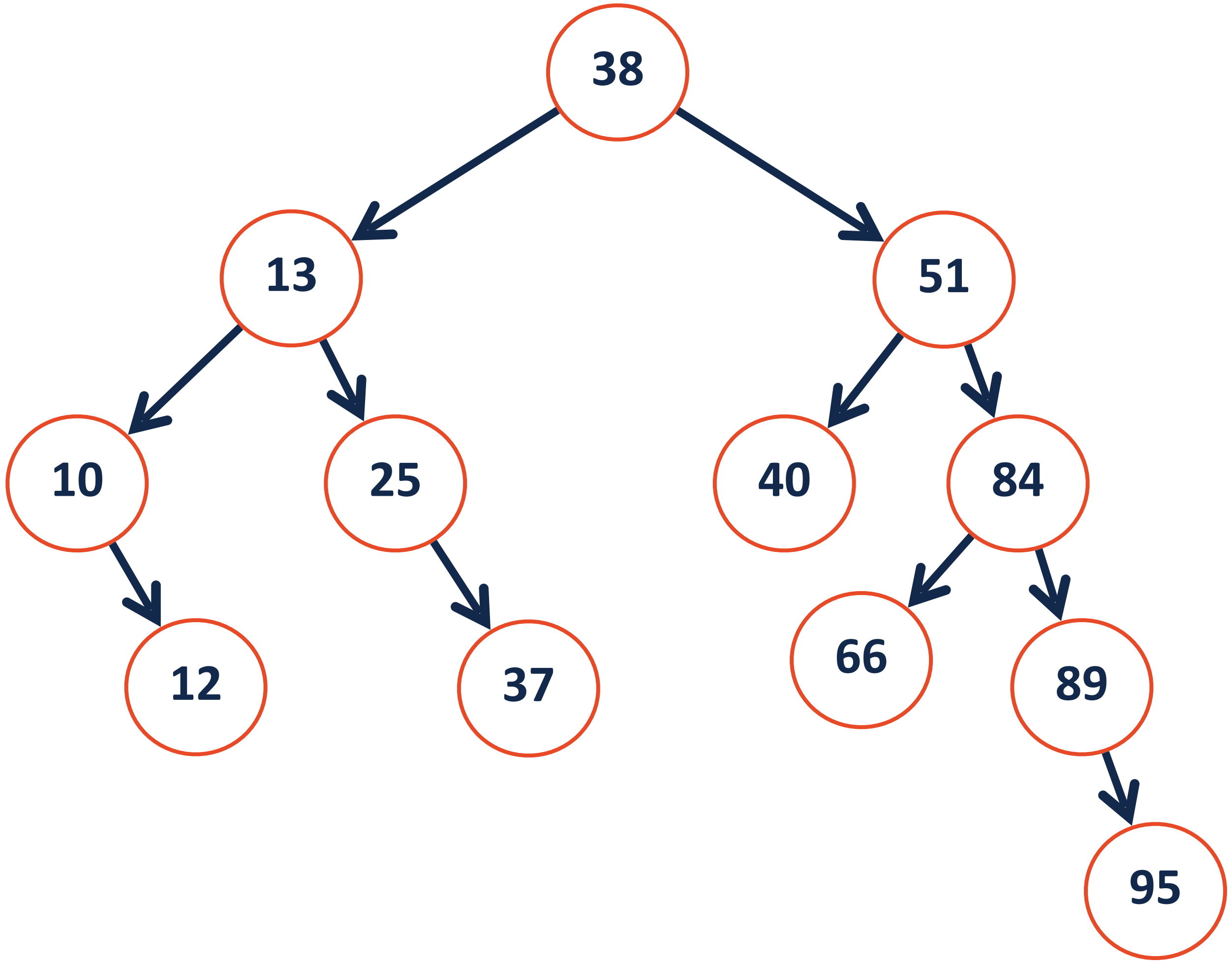
Base Case:

Recursive Step:

Combining:

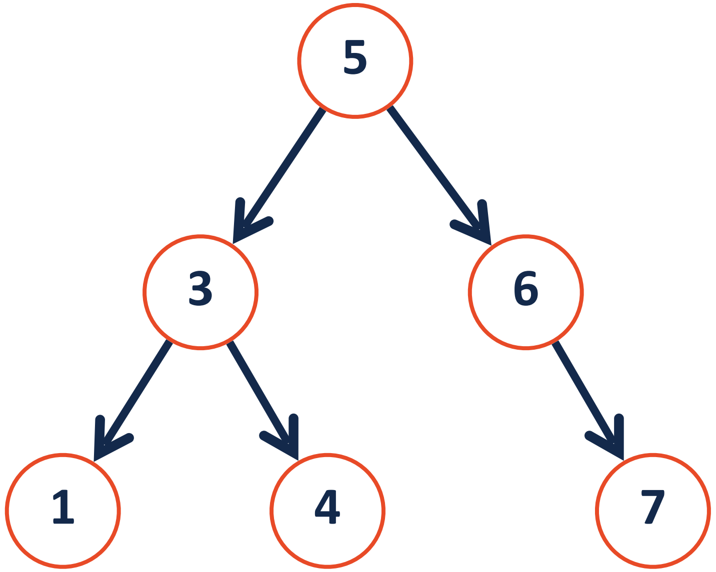
BST Insert

`insert(33)`



BST Insert

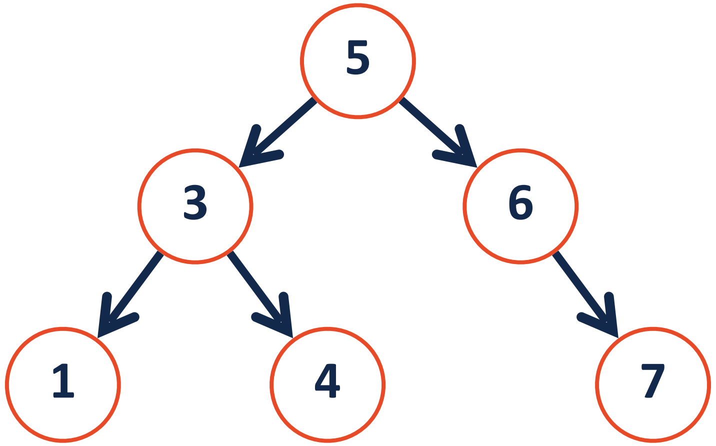
```
# inside class bst
def insert(self, key, val):
    self.root = self.insert_helper(self.root, key, val)

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



BST Insert

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

BST Find



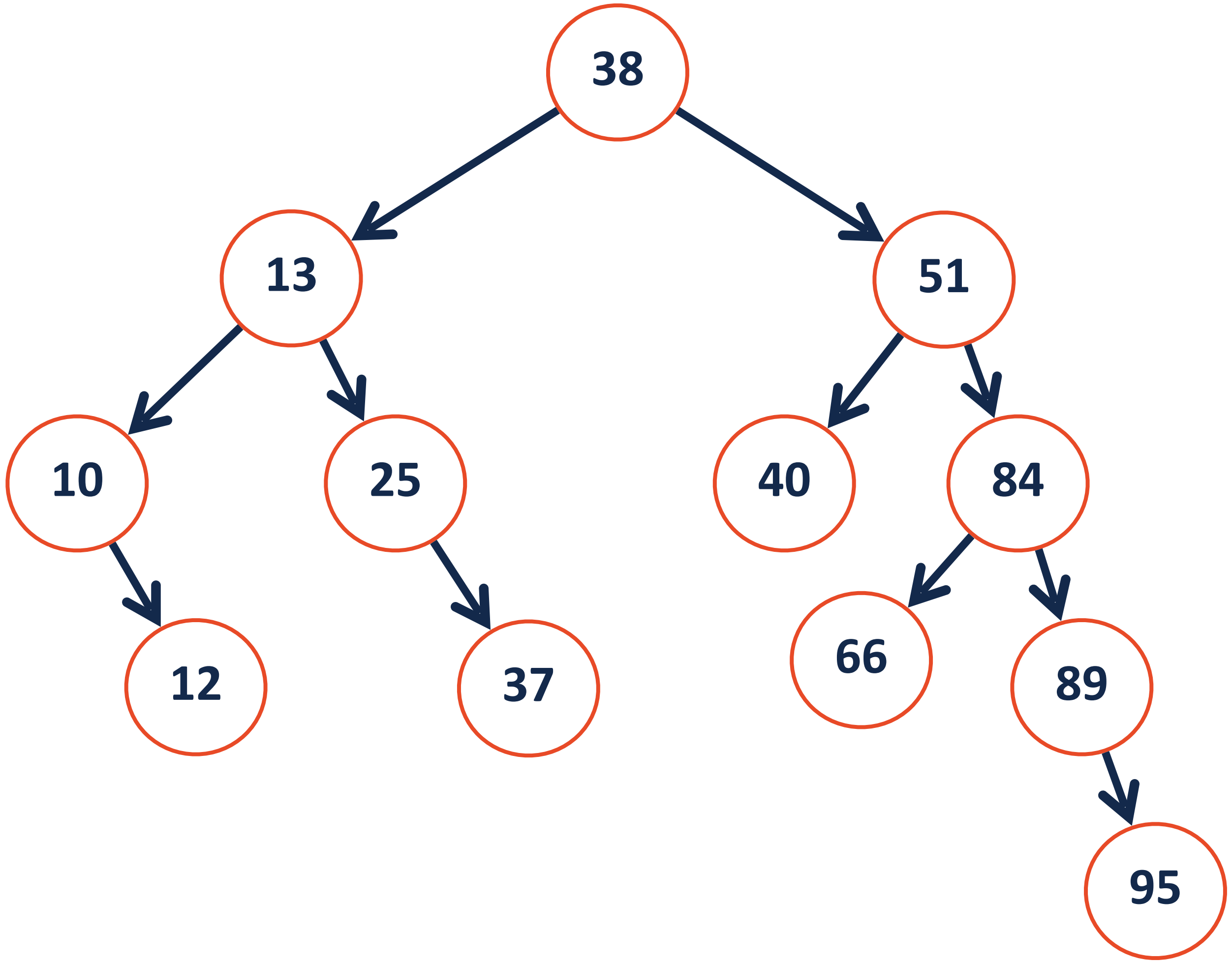
Base Case:

Recursive Step:

Combining:

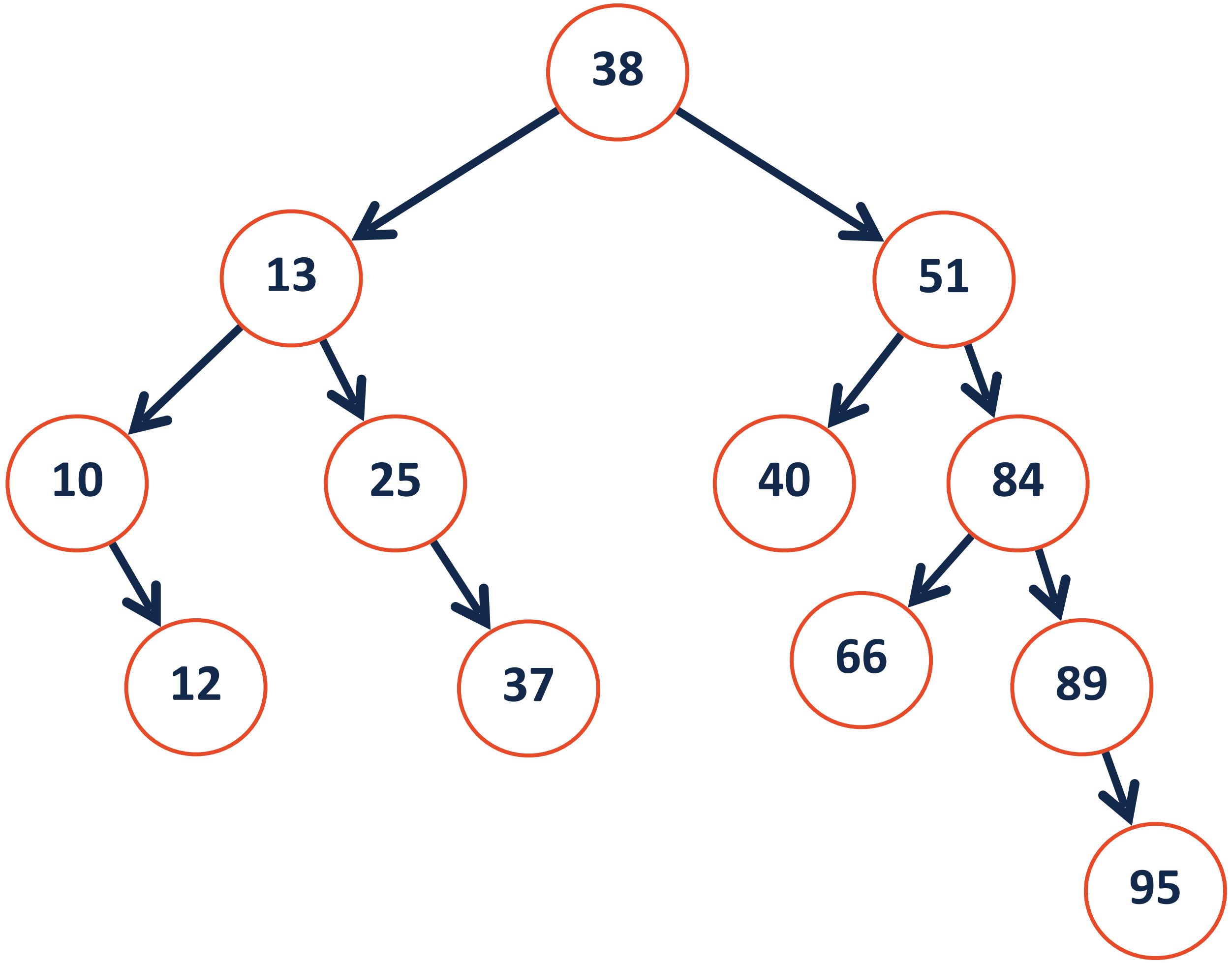
BST Find

`find(66)`



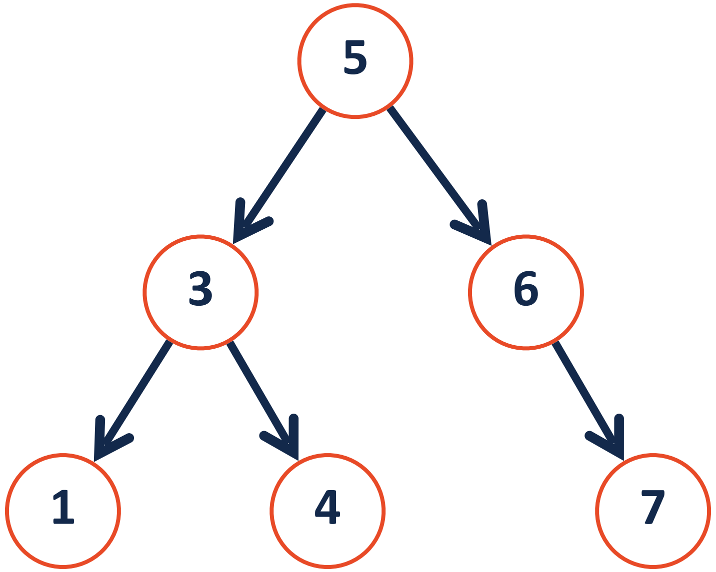
BST Find

find(9)



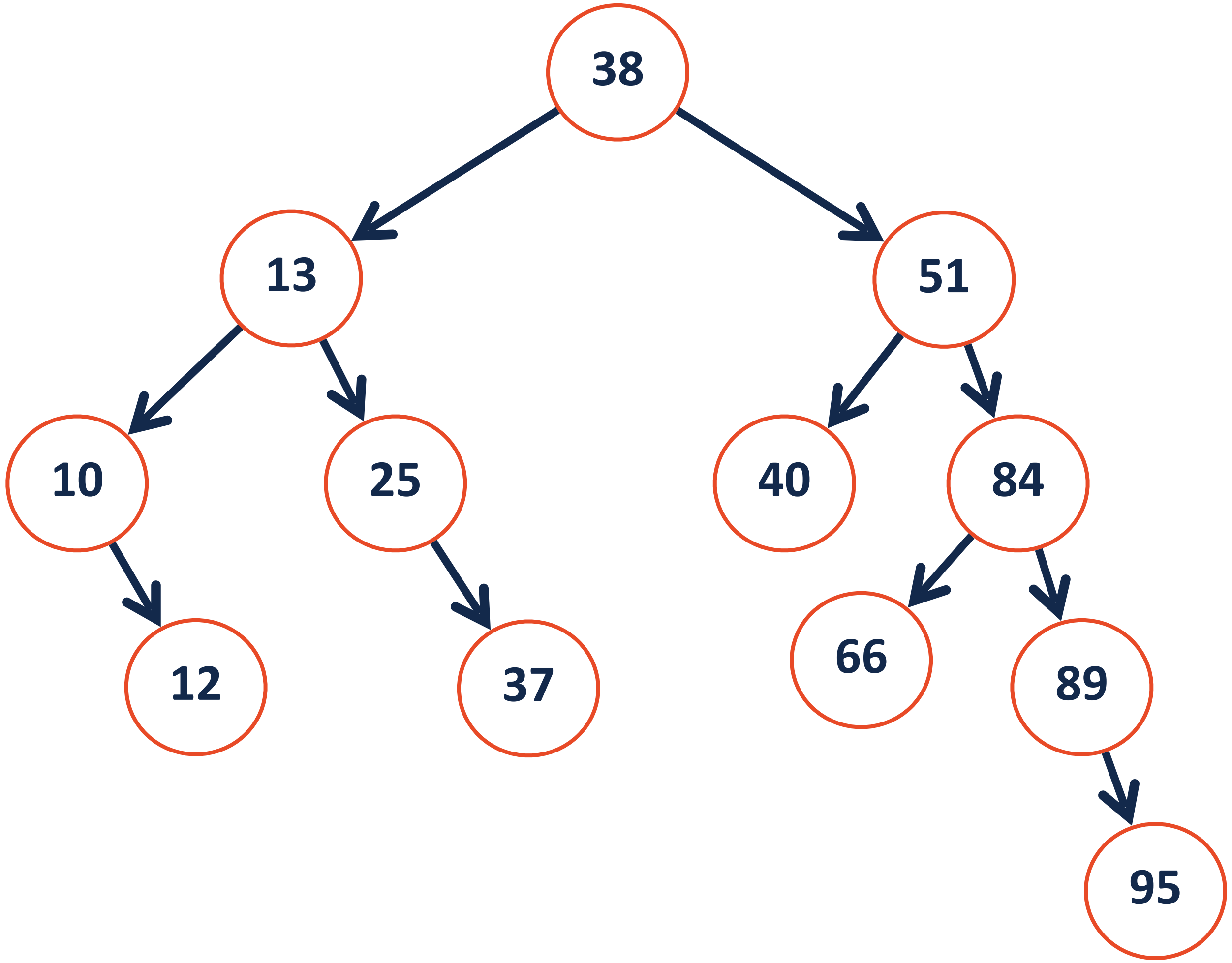
BST Find

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






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



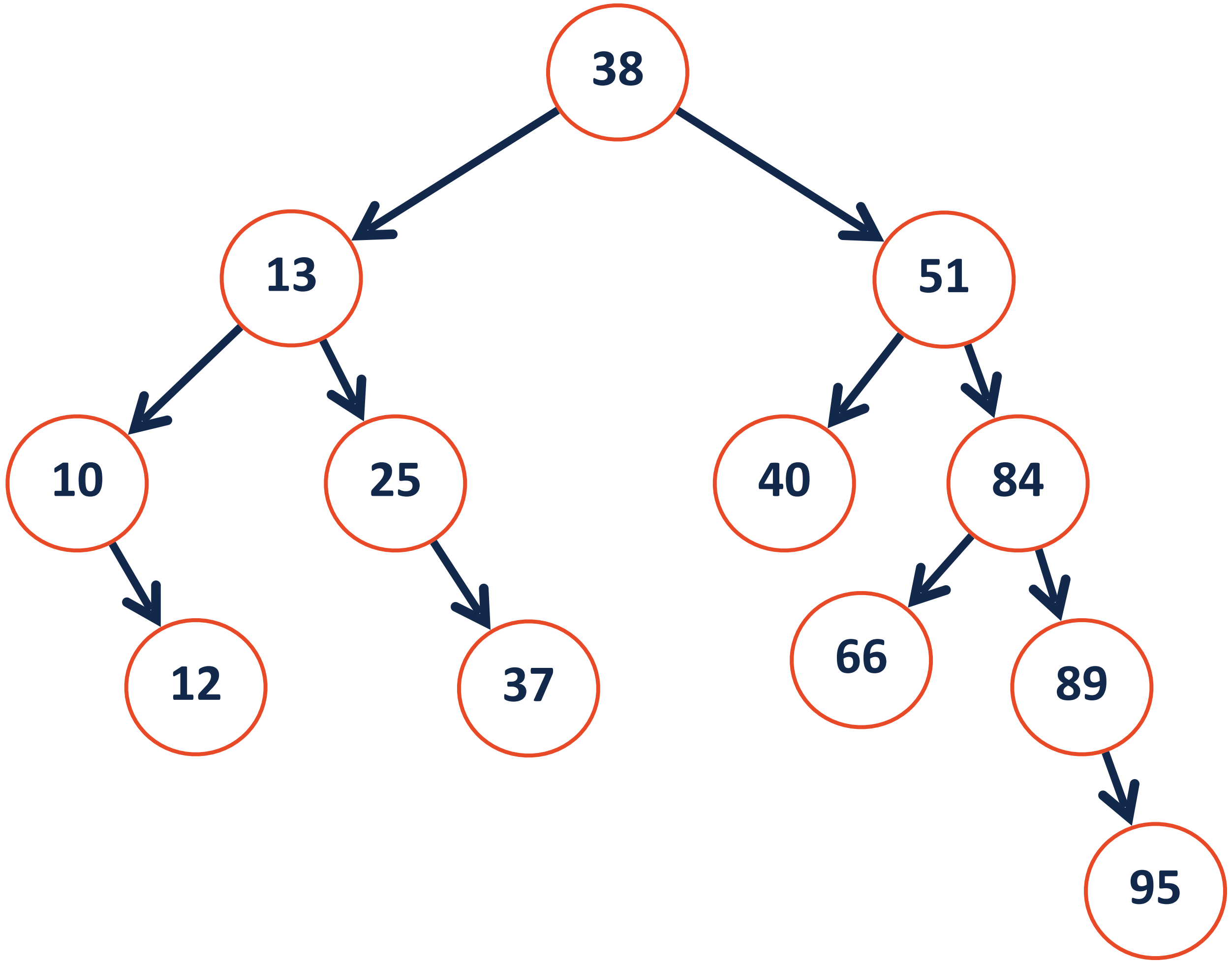
BST Remove

`remove(40)`



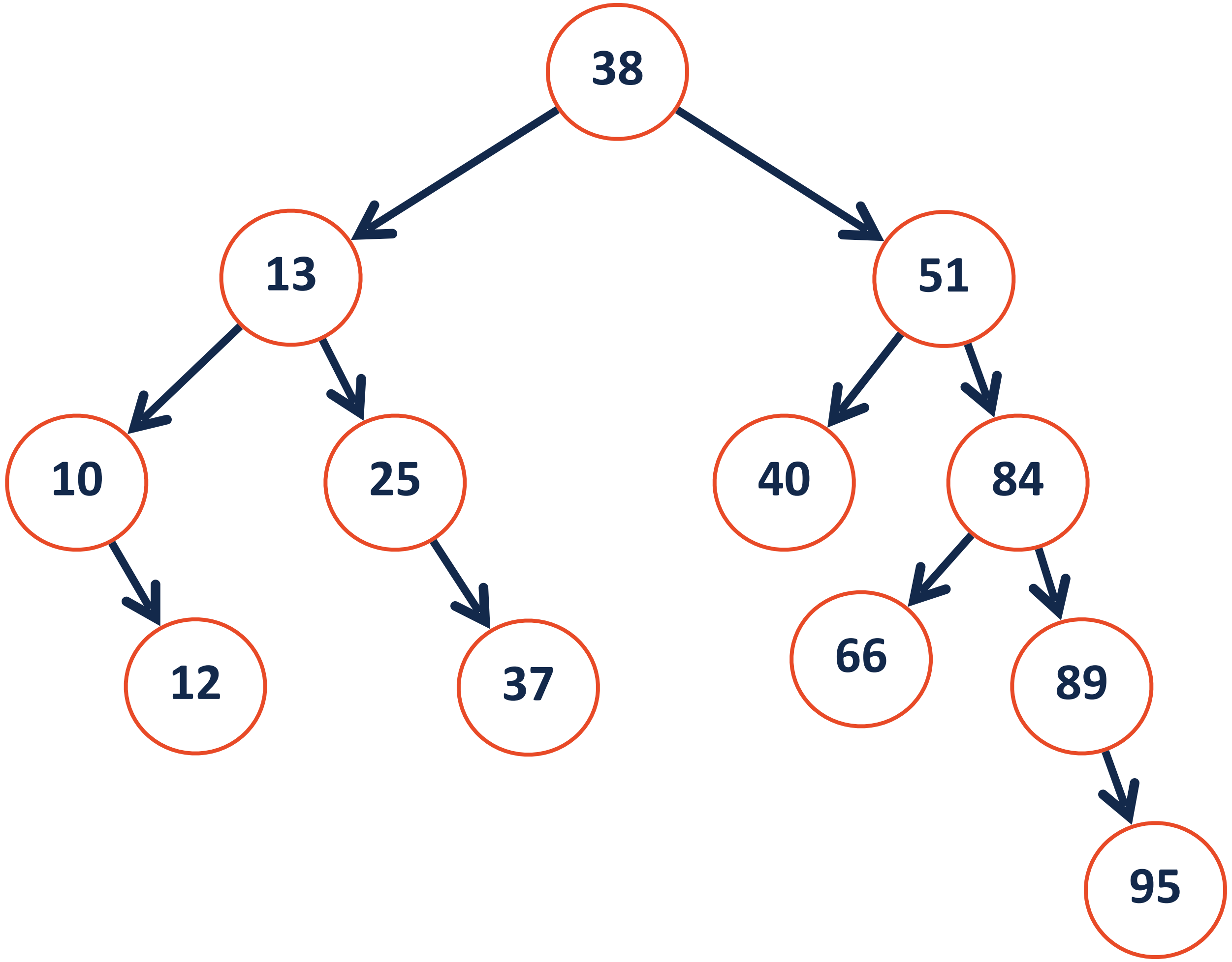
BST Remove

`remove(25)`



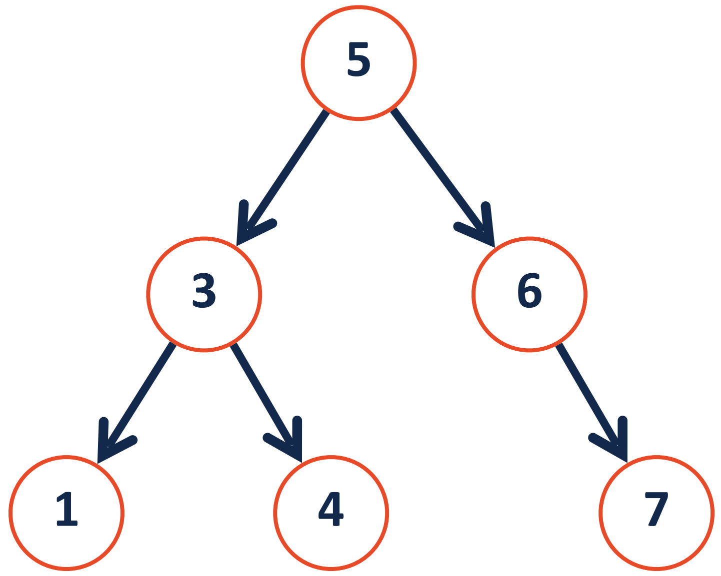
BST Remove

`remove(13)`



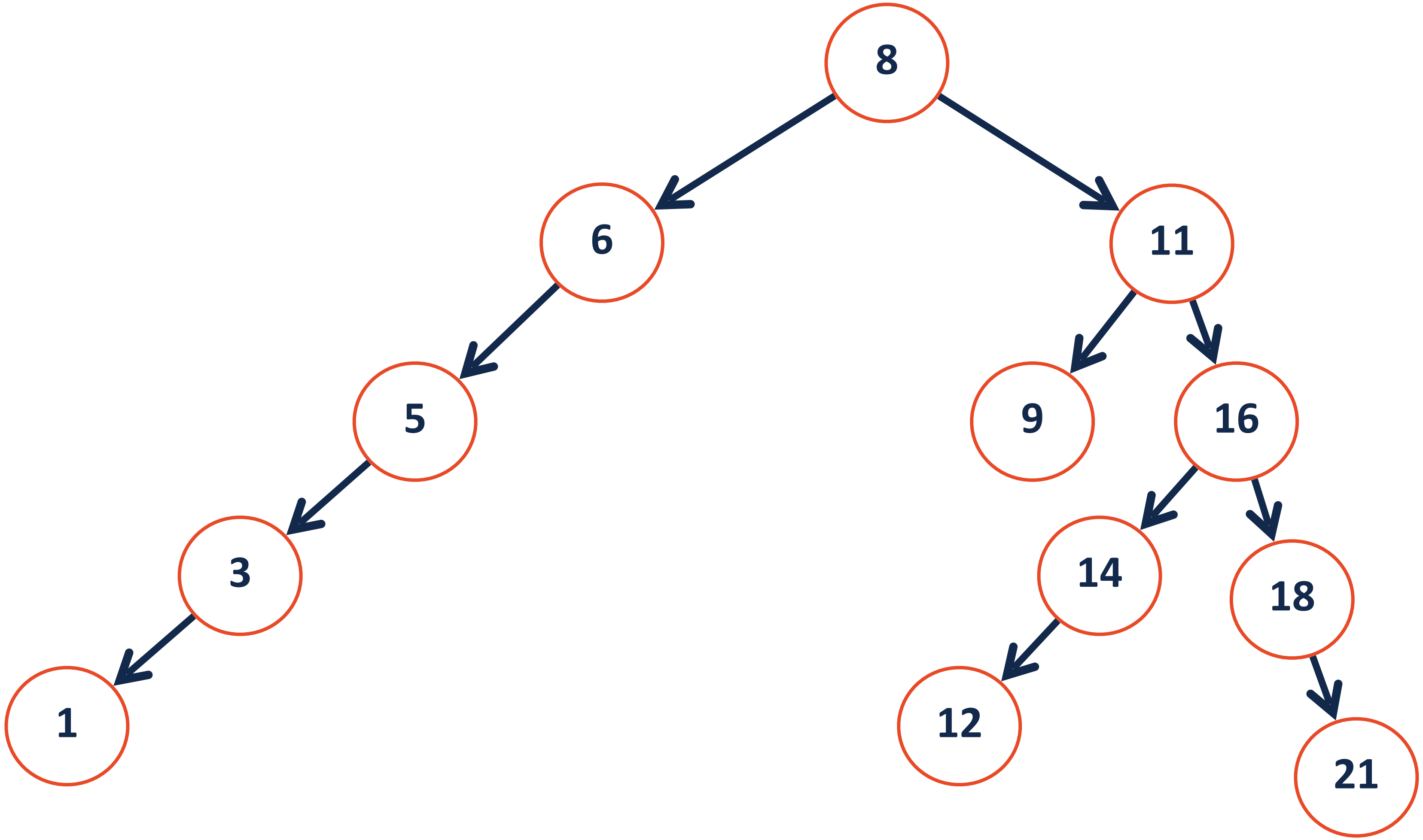
BST Remove

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

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



BST Remove

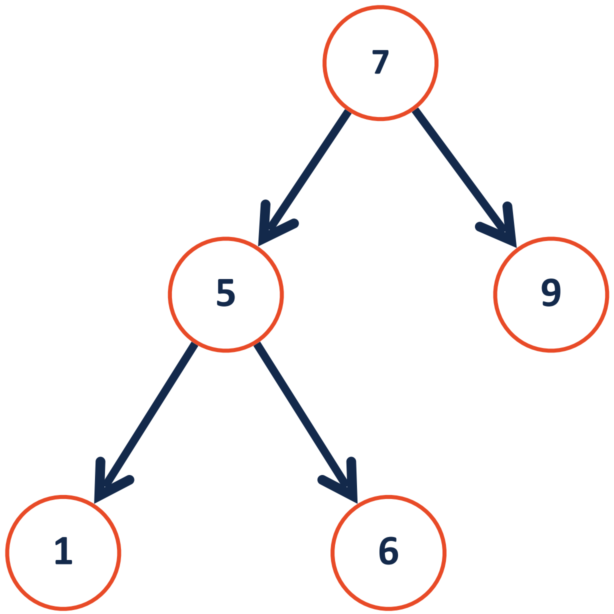
What will the tree structure look like if we remove node 16 using IOS?

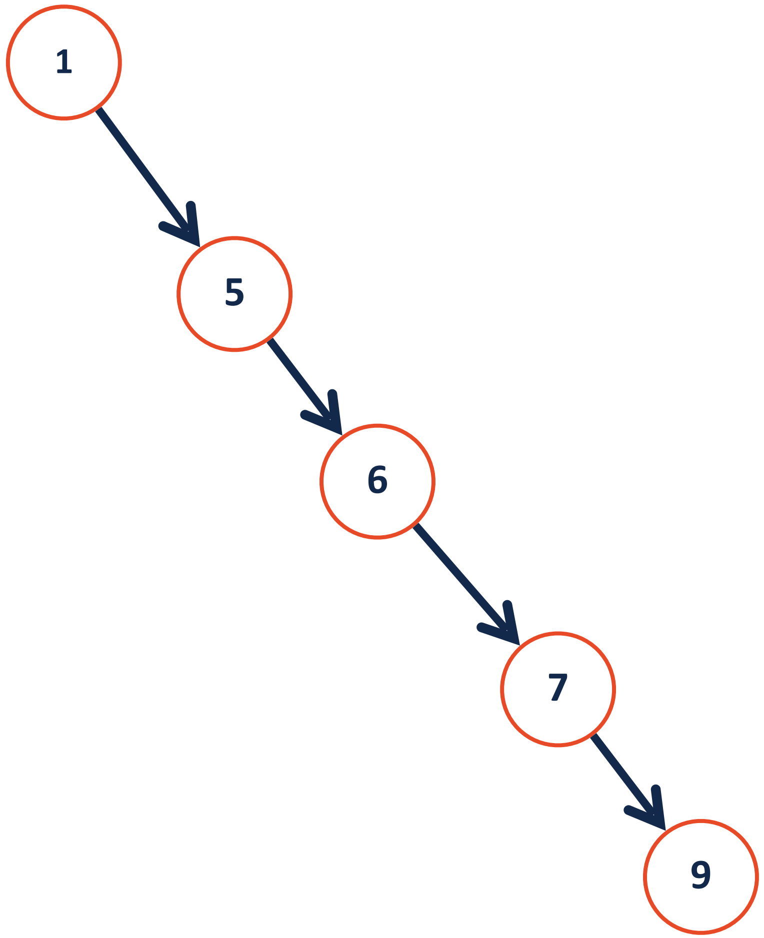


BST Analysis – Running Time

Operation	BST Worst Case
find	
insert	
delete	
traverse	

Limiting the height of a tree





Option A: Correcting bad insert order

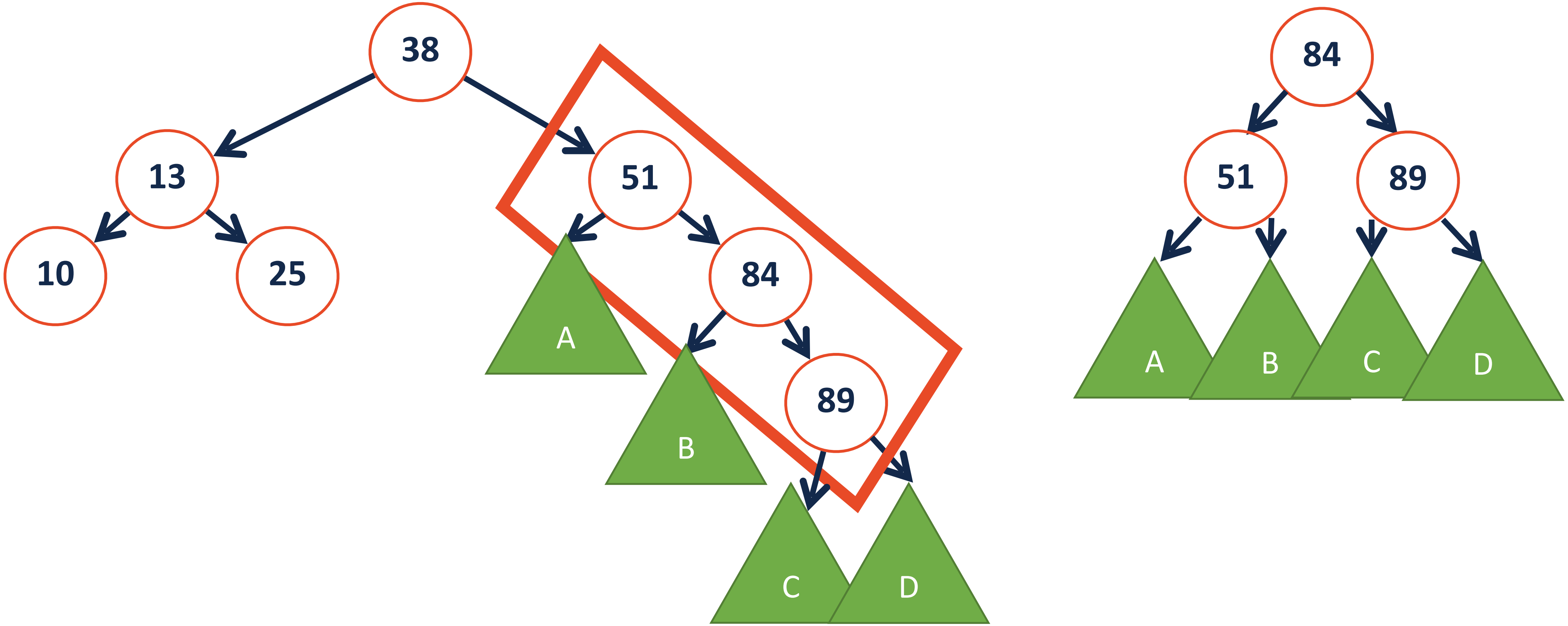
The height of a BST depends on the order in which the data was inserted

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!



When would we use a tree?

Pretend for a moment that we always have an optimal BST.

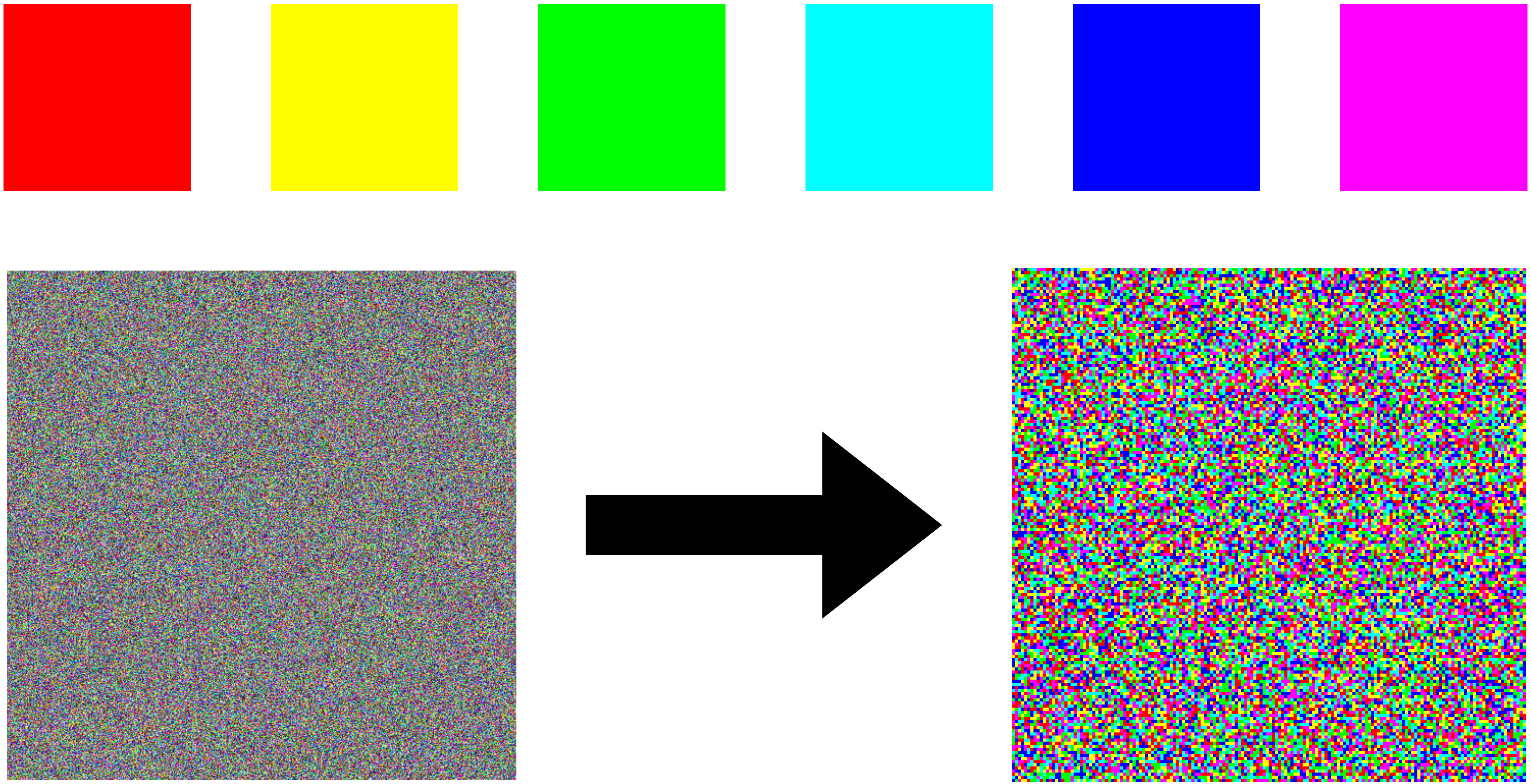
What is the running time of **find**?

What is the running time of **insert**?

What is the running time of **remove**?

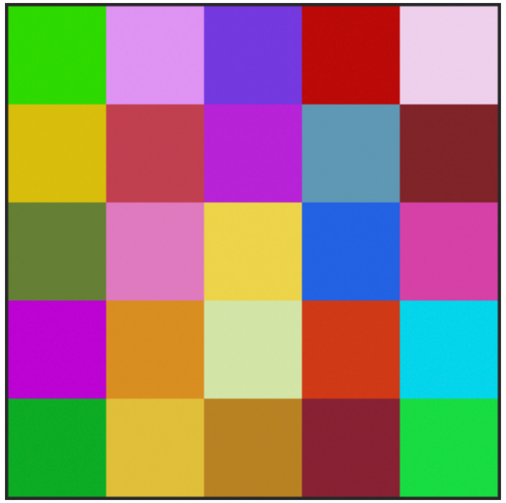
Is there a data structure with a *better* running time for all of these?

Real World Use Case: Nearest neighbor search



Real World Use Case: Nearest neighbor search

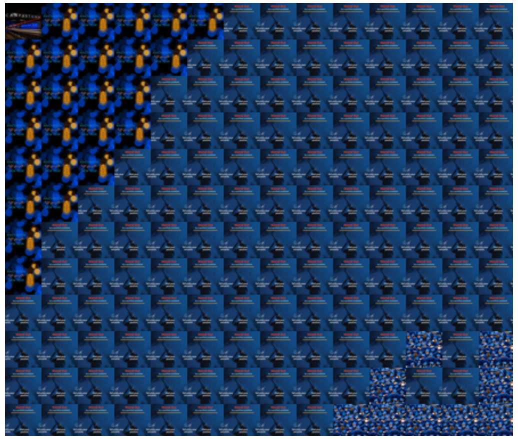
(255, 0, 0)   (255, 255, 0)   (0, 255, 0)   (0, 255, 255)   (0,0,255)   (255,0,255)



```
[[ 45 218   0],[223 147 243],[116  57 223],[187   9   9],[238 208 236]]
[[216 190  15],[193  64  80],[184  35 215],[ 95 152 180],[128  36  41]]
[[101 128  53],[224 122 191],[237 212  74],[ 35  98 227],[214  66 167]]
[[188   3 211],[217 142  33],[210 229 167],[208  57  22],[  3 213 235]]
[[ 11 172  37],[225 191  57],[184 130  34],[136  33  51],[ 26 220  67]]
```

Real World Use Case: Nearest neighbor search

Real World Use Case: Nearest neighbor search



Real World Use Case: Nearest neighbor search

Given an input image, how can we find the closest match from a collection collection of other images?