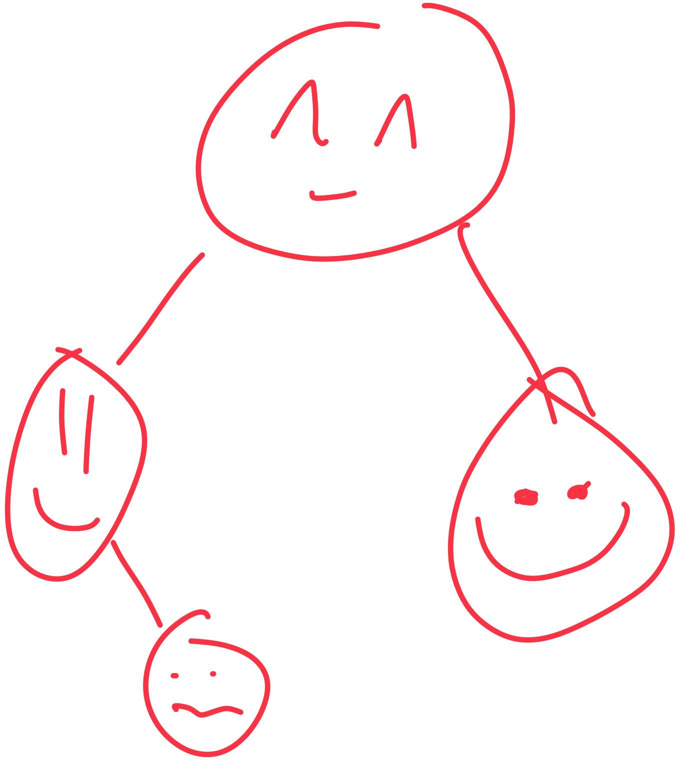
Algorithms and Data Structures for Data Science
Trees

CS 277
Brad Solomon

February 26, 2024



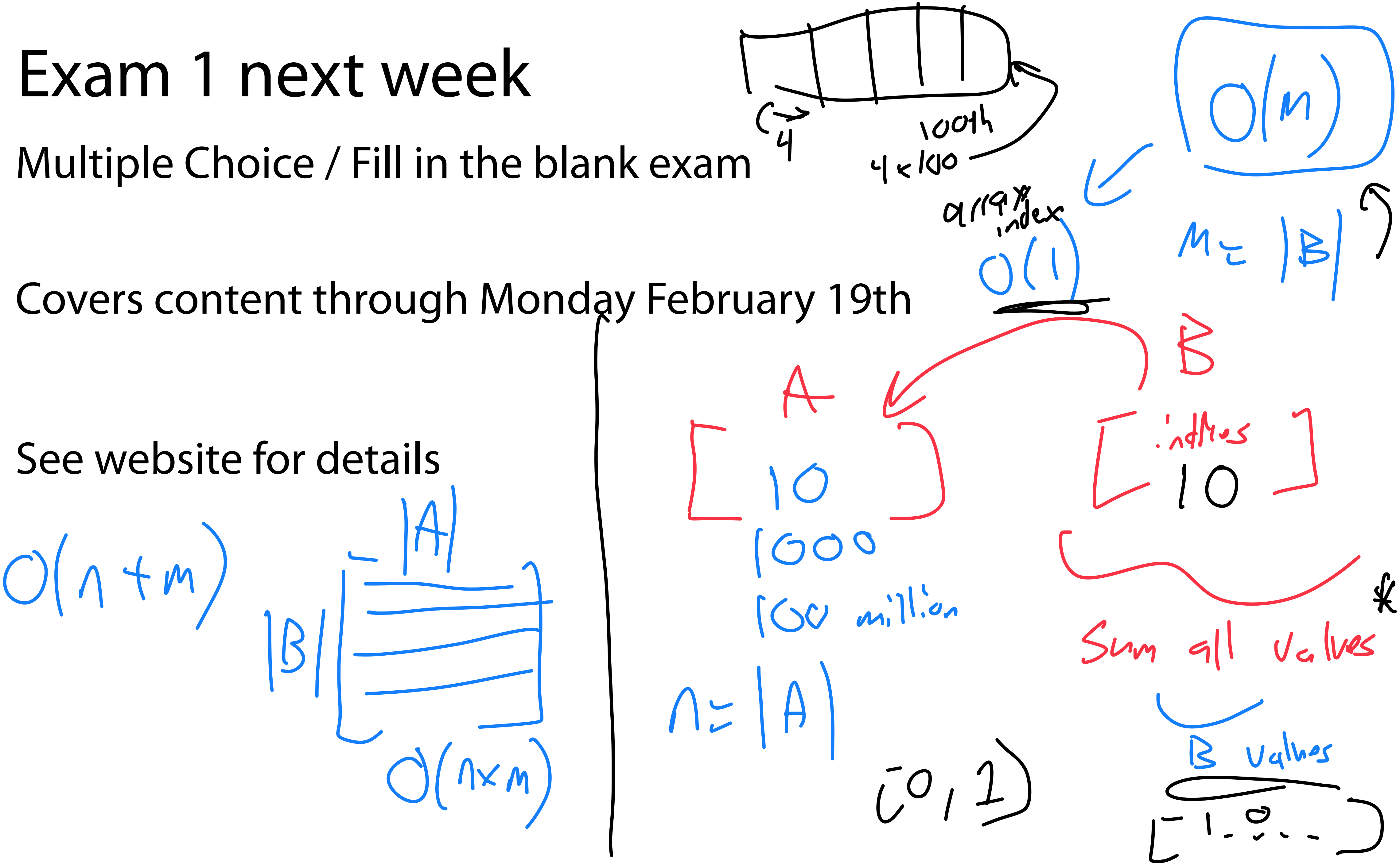
Department of Computer Science

Exam 1 next week

Multiple Choice / Fill in the blank exam

Covers content through Monday February 19th

See website for details



Learning Objectives

Build an understanding of the tree ADT

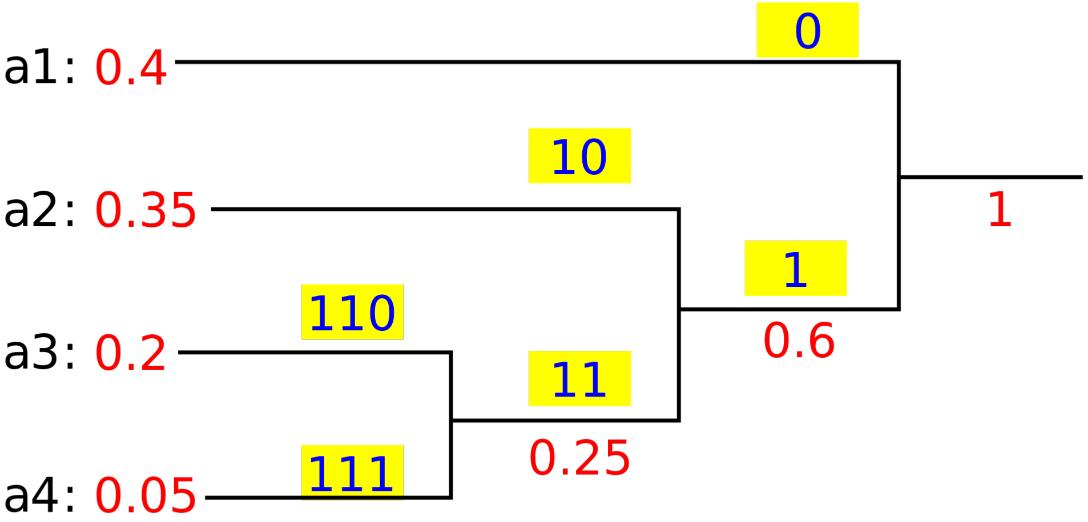
See the implementation details of a binary tree

Practice recursion in the context of trees

There are many *types* of trees

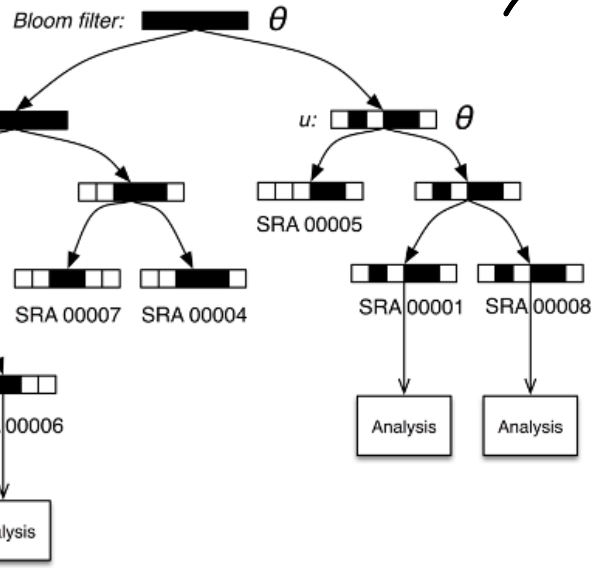
Huffman Trees

0



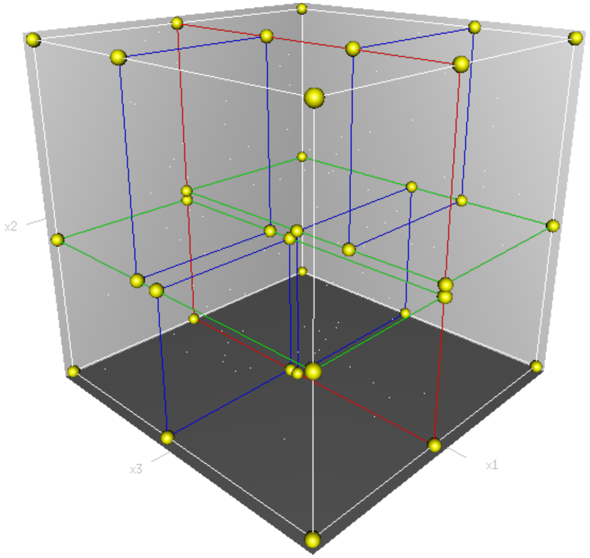
SBT

X ?



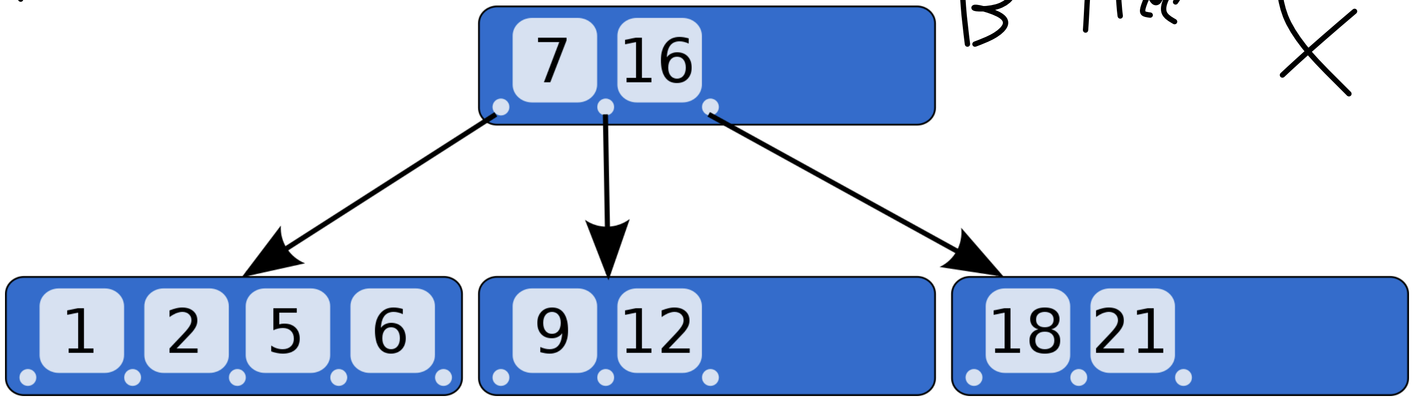
KD tree

*



B tree

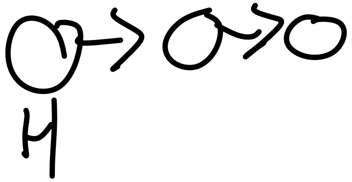
X



(Binary) Tree Recursion

ListNode → Linked List

↳ head

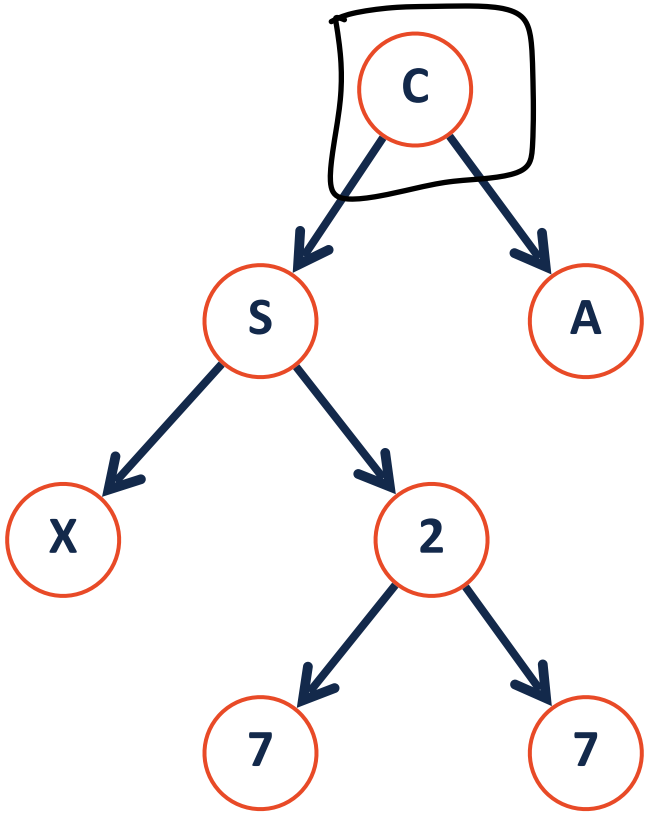


A **binary tree** is a tree T such that:

$$T = None$$

or

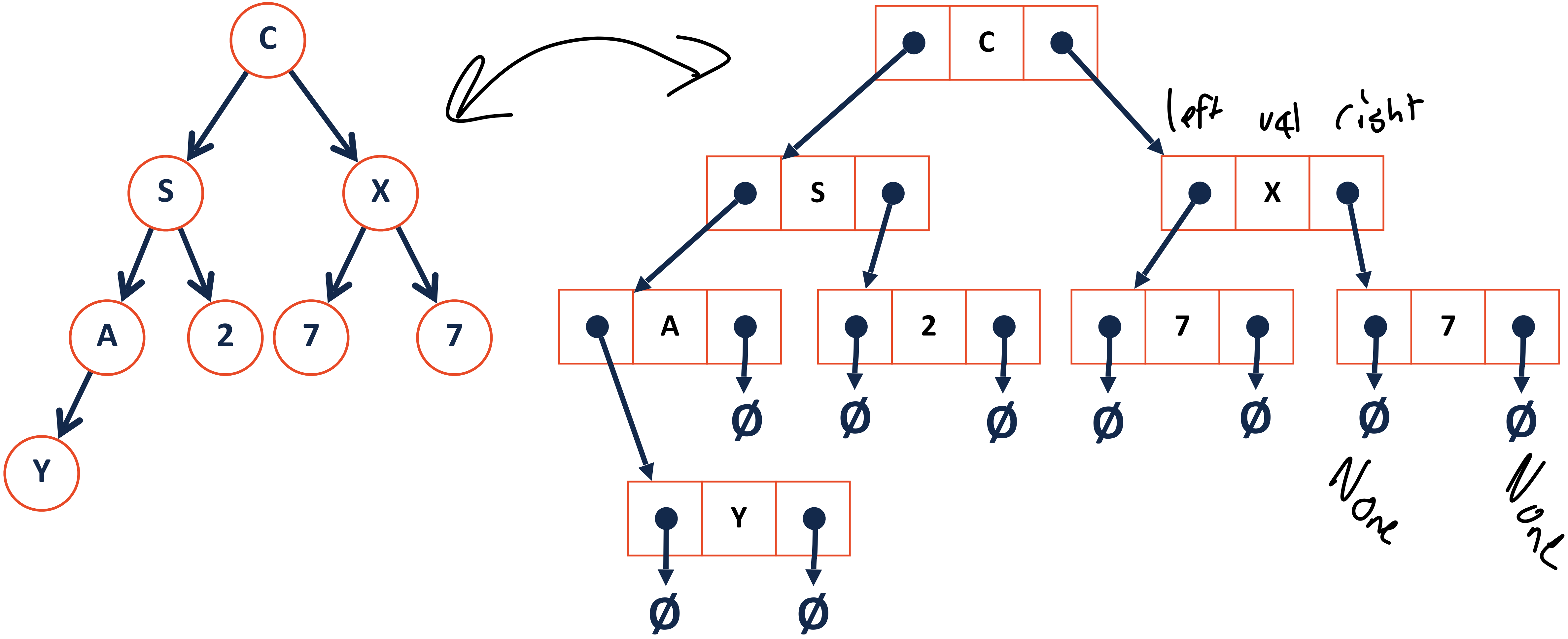
$$T = treeNode(val, T_L, T_R)$$



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

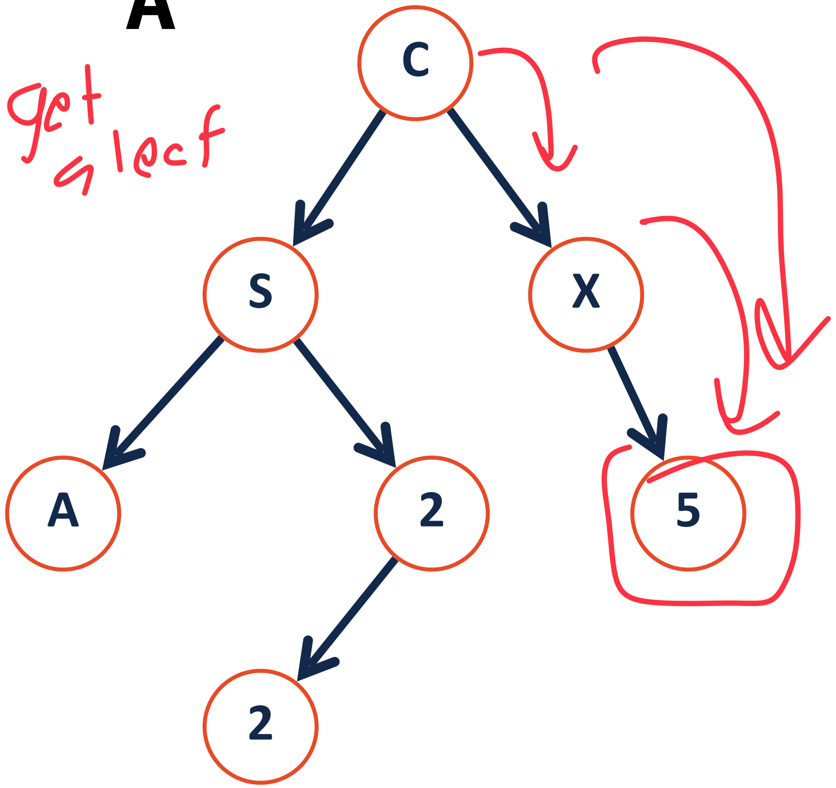
```
class binaryTree:
    def __init__(self):
        self.root = None
```

Visualizing trees



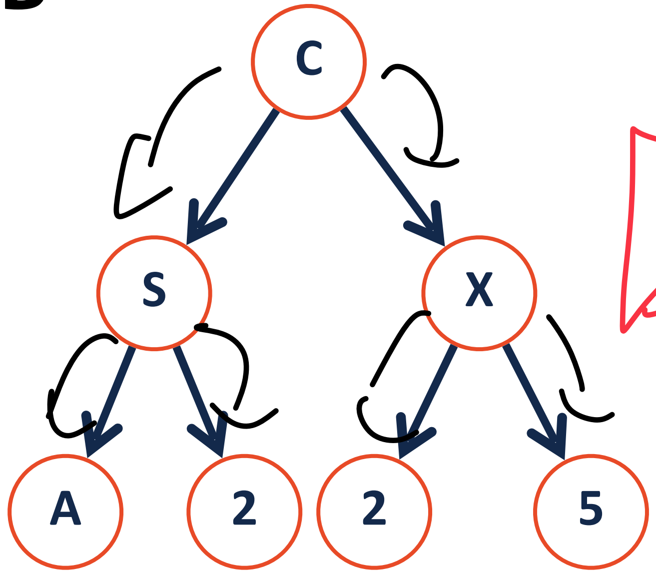
Which of the following are binary trees?

A



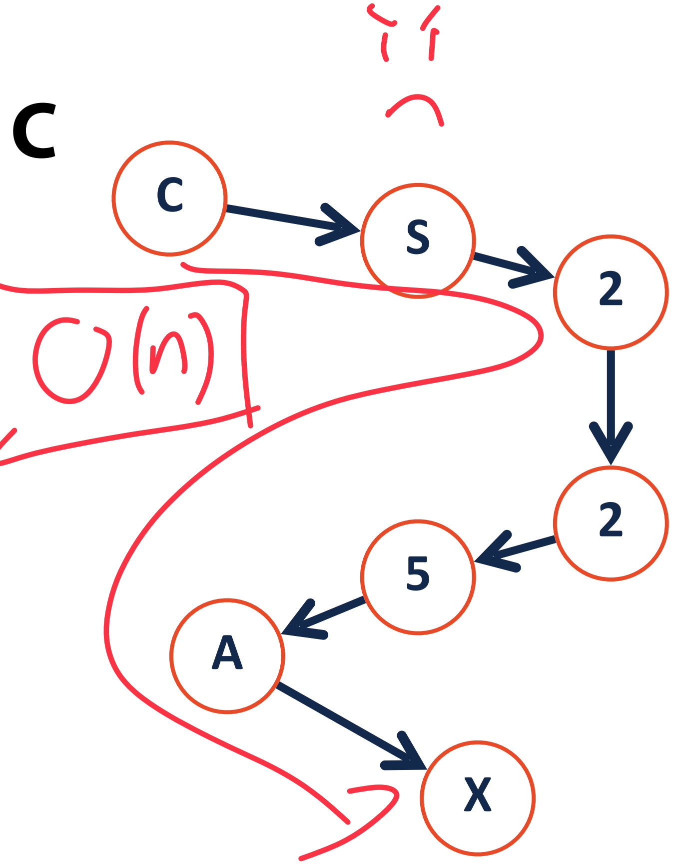
Yes!

B



Yes!

C



O(n)

At most 2 children

Yes!

Tree ADT

Properties

↳ root Node

Functions

↳ Traverse (visit all nodes in an order)

↳ add or insert data

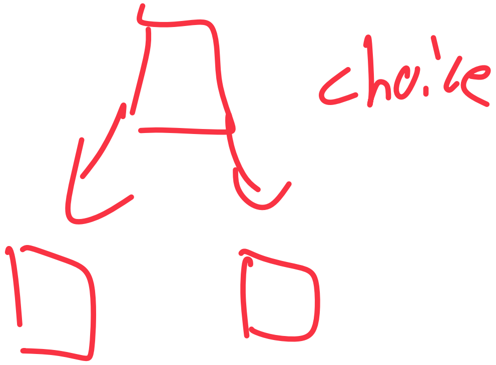
↳ get Node / Search / Lookup

access a node

↳ Remove data

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 (get)

Recursion Practice: build_random_tree()

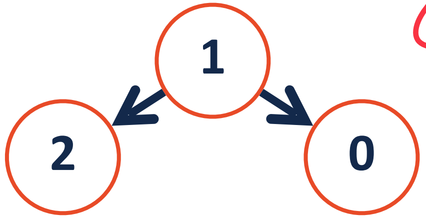
return is root

↳ Have input list of #s

```
def build_random_tree(size, seed=None):
    random.seed(seed)
    keys = list(range(size))
    random.shuffle(keys)

    root = random_tree_helper(keys)
    return root
```

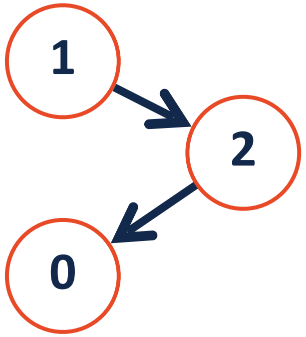
Ex: build_random_tree(3, 1)

1, 2, 0



[0, 1, 2]

Ex: build_random_tree(3, 1001)



[1, 2, 0]

Base Case:

List of size 0 → root = None

size 1 root = treeNode(list[0])

[1] → ①

Reduction step: Remove front item [pop(0)]

Combine step: Single val + (n-1) items in a list

treeNode(val)

recurse

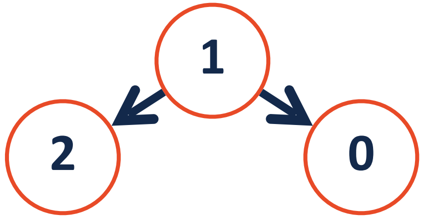
recurse

Recursion Practice: build_random_tree()

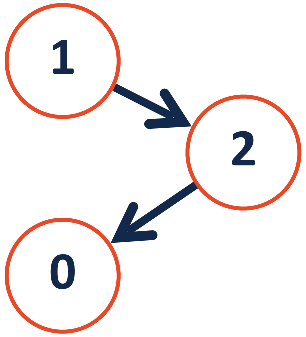
```
def build_random_tree(size, seed=None):
    random.seed(seed)
    keys = list(range(size))
    random.shuffle(keys)

    root = random_tree_helper(keys)
    return root
```

```
def random_tree_helper(keyList):
    # Base Case
    if len(keyList) == 0:
        return None
    if len(keyList) == 1:
        return treeNode(keyList[0])

    # Reduction Step
    node = treeNode(keyList.pop(0))

    # Combining Step
    partition = random.randint(0, len(keyList))
    leftList = keyList[:partition]
    rightList = keyList[partition:]

    node.left = random_tree_helper(leftList)
    node.right = random_tree_helper(rightList)

    return node
```

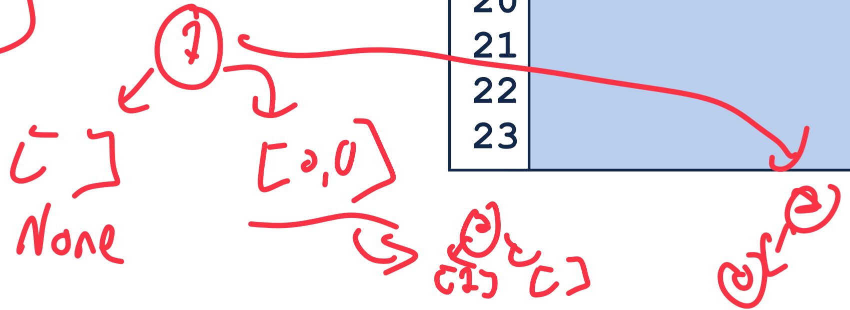
Ex: build_random_tree(3, 1)



Ex: build_random_tree(3, 1001)



[1,2,0]



Binary Tree Insert

insert(val=13, parent = 6, direction="left",

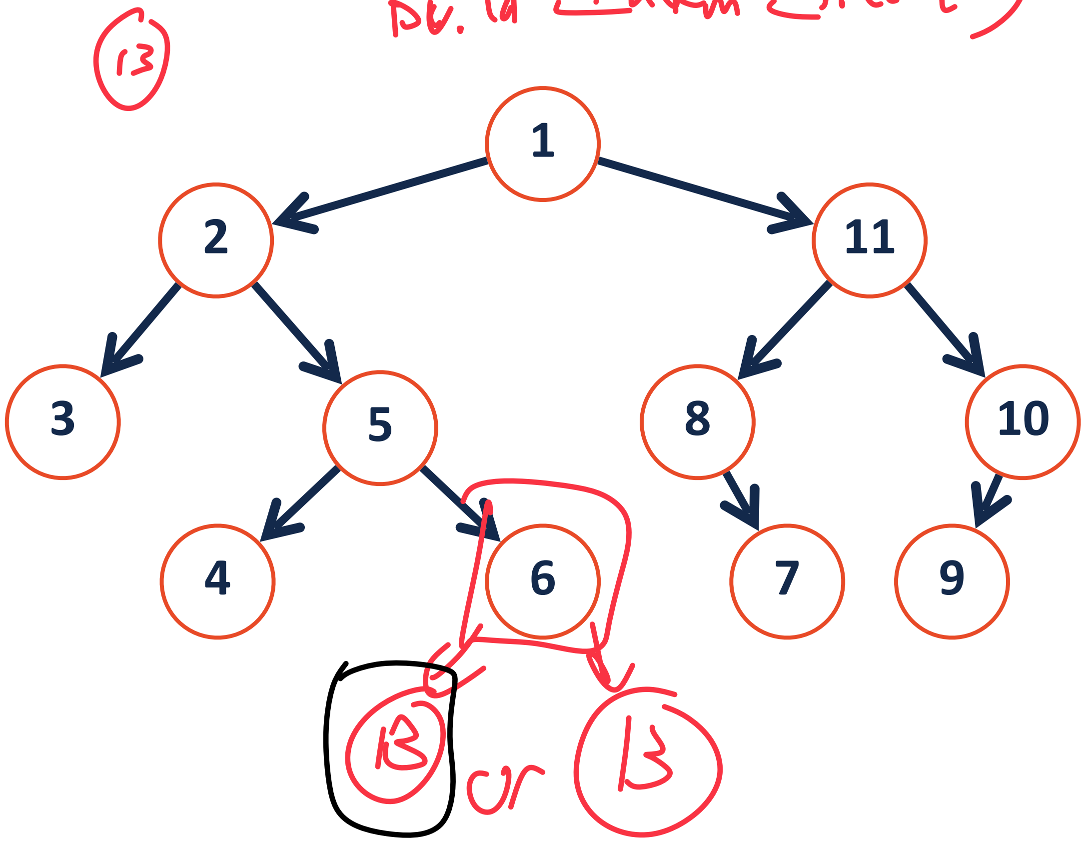
If I want to insert a value into my tree, what information do I need?

Ex: I want to insert the value '13'.

We need to know parent of 13

-> and the child "direction"

build_random_tree()



Steps:

1) Make new Node(13)

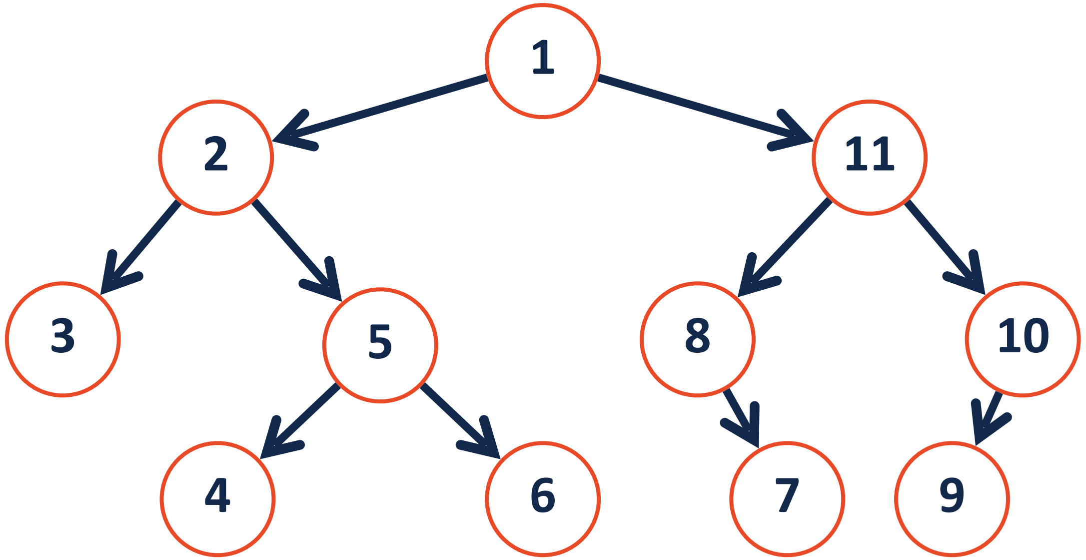
Binary Tree Insert

Different implementations will have very different insert strategies!

In our case, we need to know the following:

1. The exact insert location
   ↳ parent
   ↳ direction

2. The value we want to insert



Binary Tree Insert

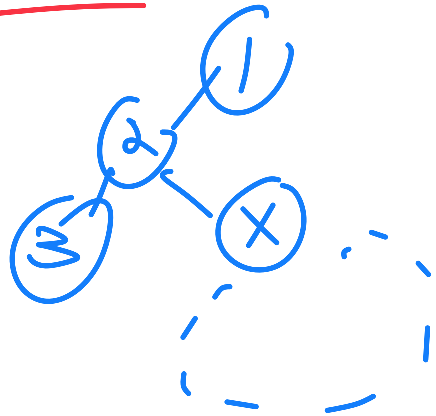
Choice: What happens if a node already exists at our target location?

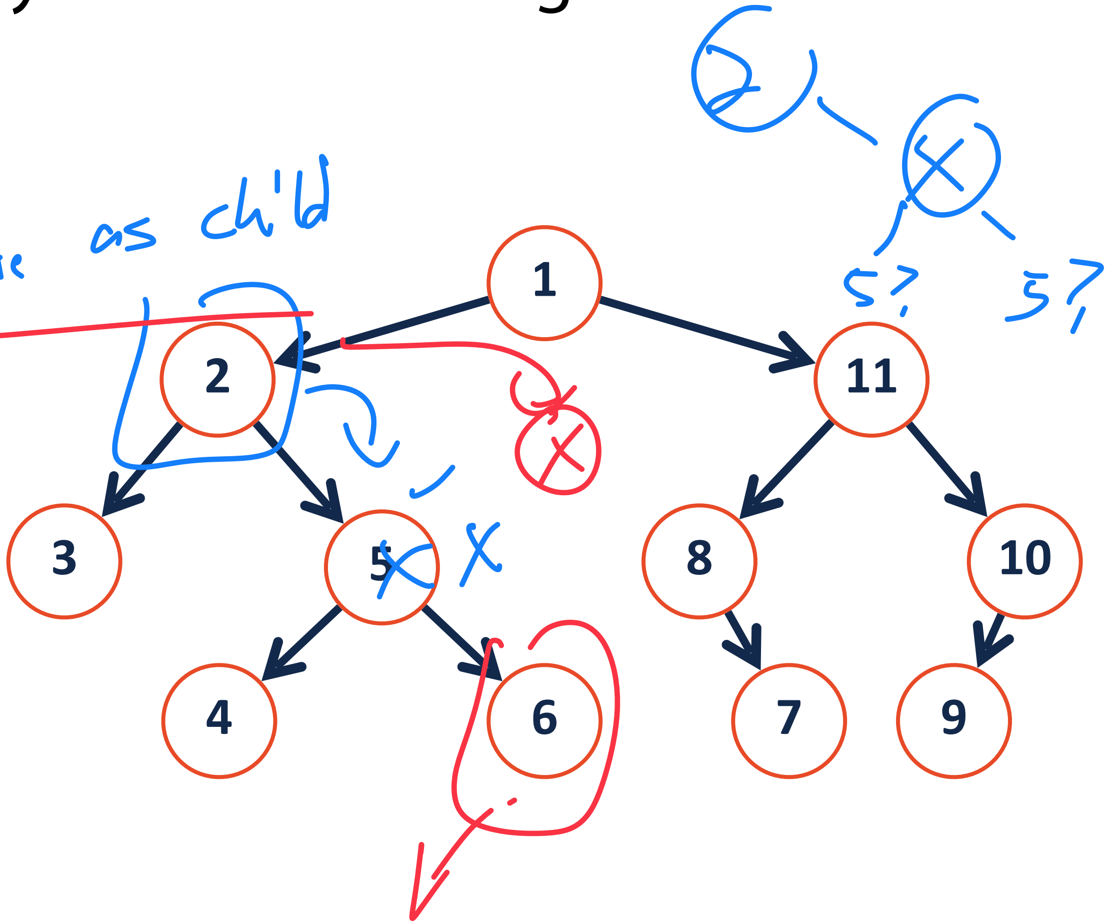
Insert(X, 2, "Right")

choice (1): Add X and put current node as child

choice (2): Delete old branch

(3): Replace value





Lets code up our choice! What is the Big O?

Binary Tree Insert

x = 5

tmp = Val

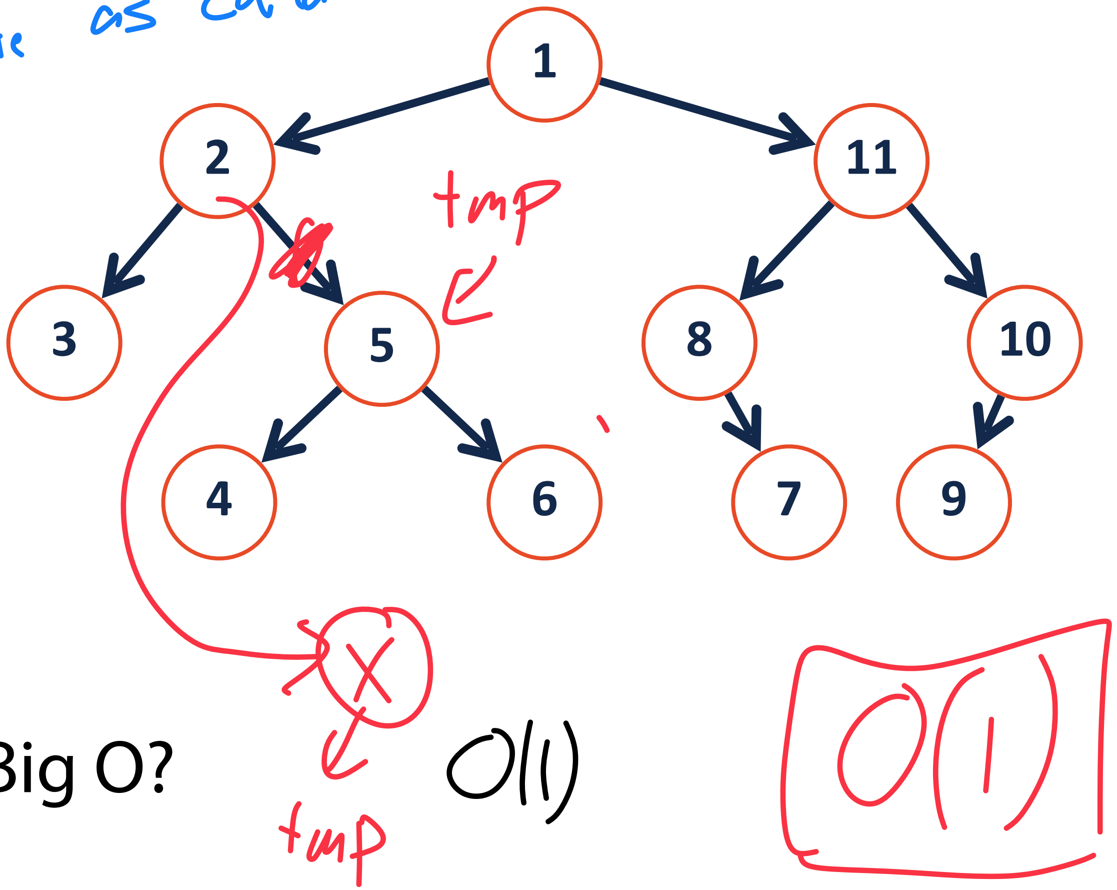
Choice: What happens if a node already exists at our target location?

Insert(X, 2, "Right")

2.right = tN(x)

choice (1): Add x and put current node as child

1) make new treeNode() O(1)
2) Save old branch as 'tmp' O(1)
3) Add new tN as child O(1)
4) Add tmp as child of tN. O(1)

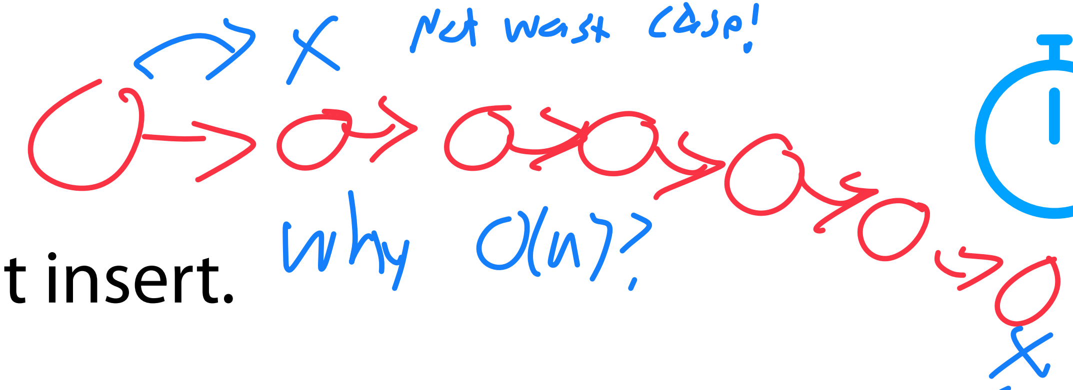


O(1)

$\boxed{O(1)}$

Lets code up our choice! What is the Big O?

Binary Tree Insert Big O



Binary Tree insert is similar to linked list insert.

If we are given the *previous* node (here, the parent node), its O(1).

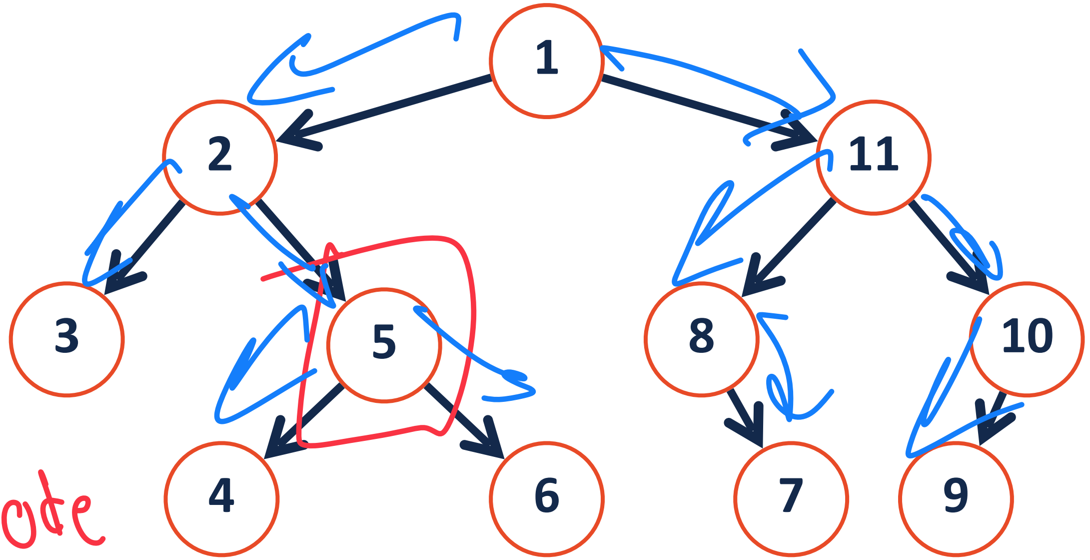
But the act of *finding* a node by value is more complicated (traversal)

insert (TreeNode , ...)

X

insert (parent value , my value , direction)

↳ O(n) if I need to find node

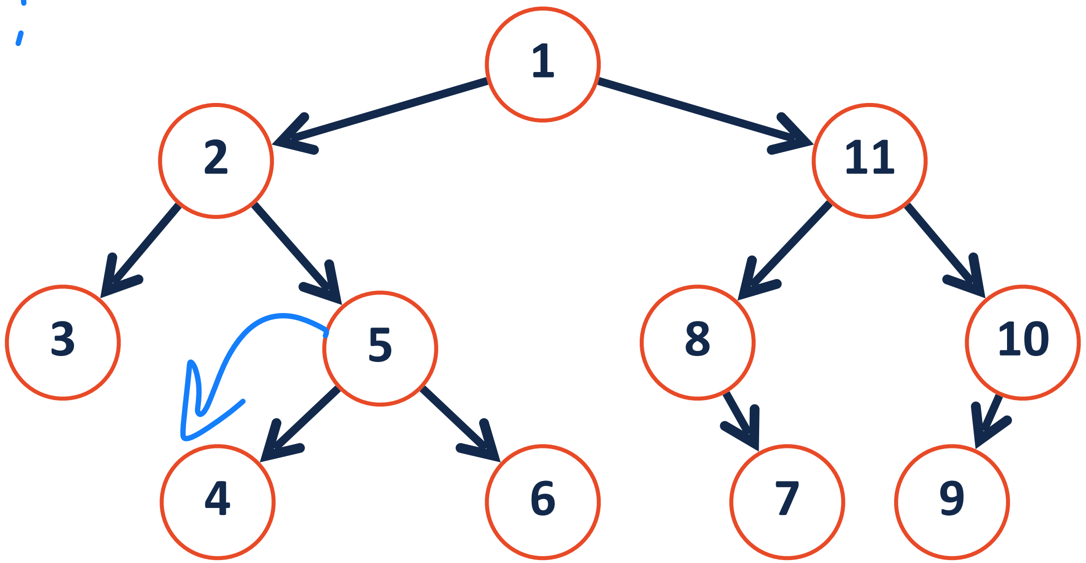


Binary Tree Remove

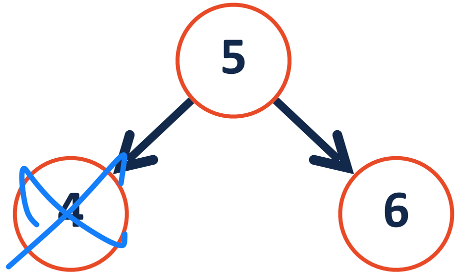
Removing a tree from a binary tree looks deceptively simple…

Ex: I want to remove the value ’4’.

↳ We need parent of removed node!



Binary Tree Remove



Choice: How do we adjust our tree given a removed node?

If the node being removed has 0 children:

remove(5, 'left')

2 info: Parent & direction

parent.direction = None

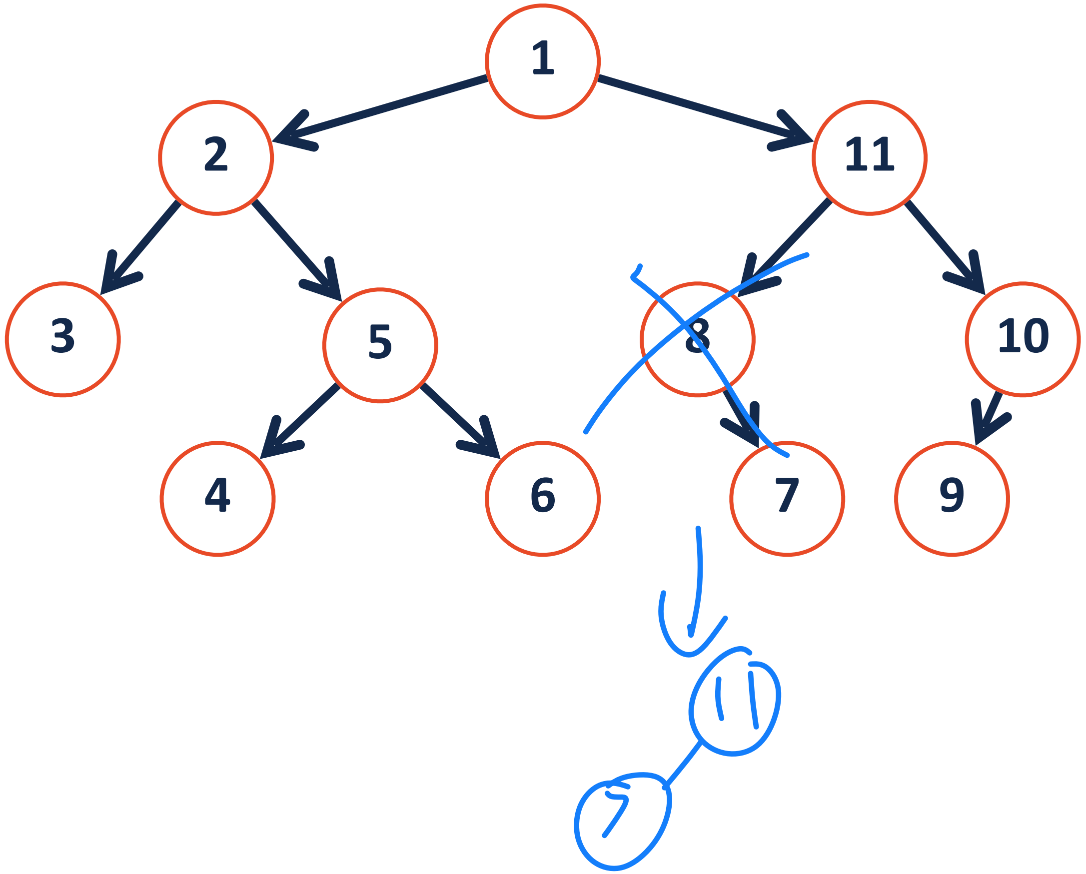
O(1)

Binary Tree Remove

When we remove, we have to be careful not to delete a tree branch!

Ex: I want to remove the value ’8’.

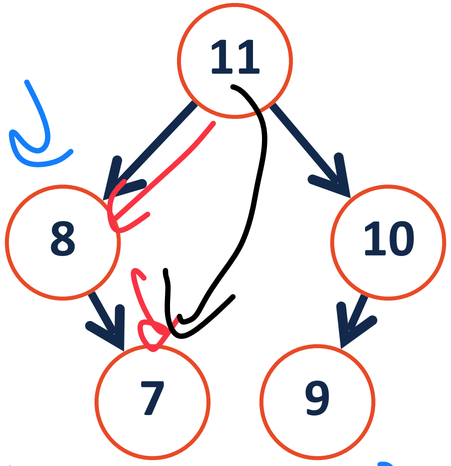
O(1)



Binary Tree Remove

Choice: How do we adjust our tree given a removed node?

If the node being removed has 1 child:



remove(Parent=11, dir=left)

N8=Node(11).left has one child

N8.right exists!

Parent.direction = (child of removed node)

Node(11).left = Node(11).left.right

O(1)

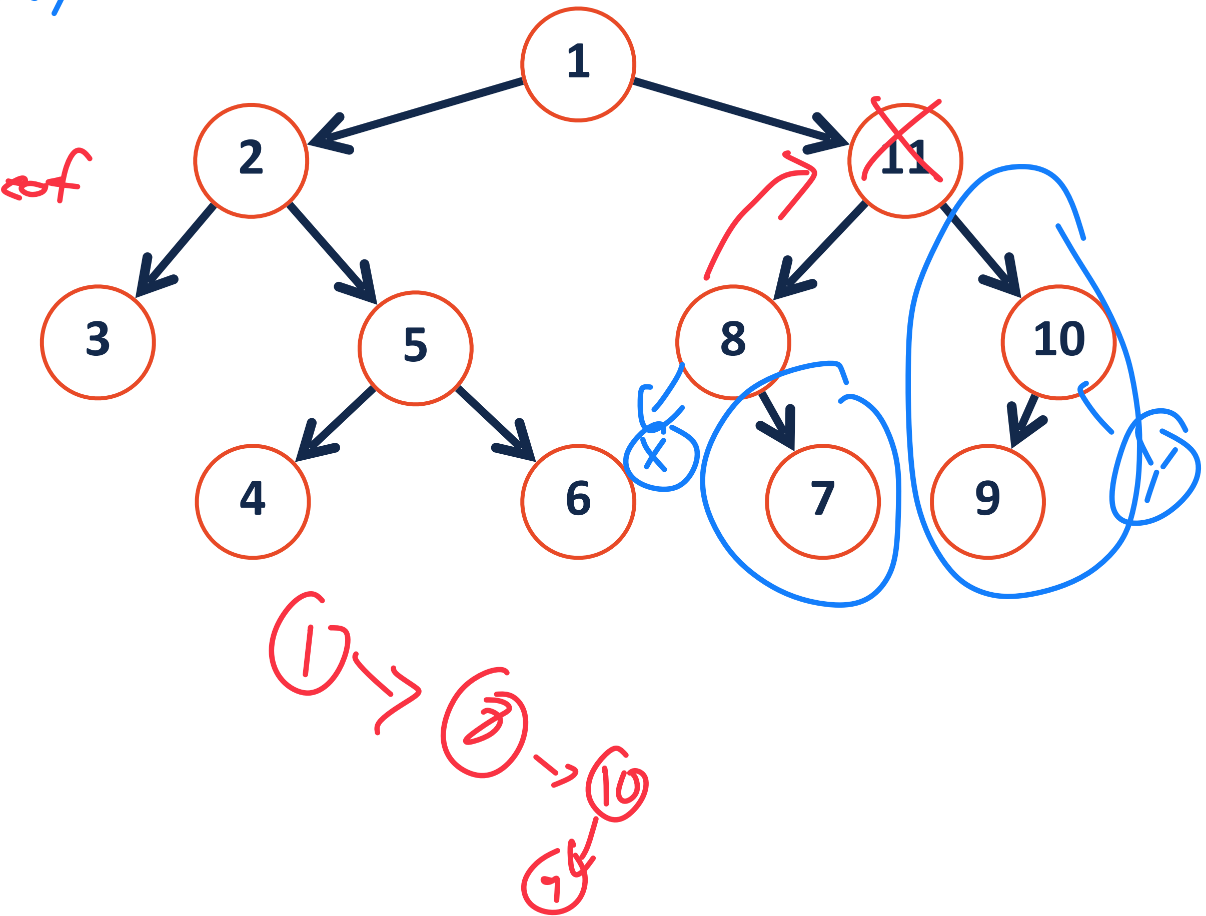
Binary Tree Remove

When we remove, we have to be careful not to delete a tree branch!

Ex: I want to remove the value '11'.

choice (1): Assign order /rules for every combination

choice (2): Swap removed value w/ leaf
↳ then remove it;



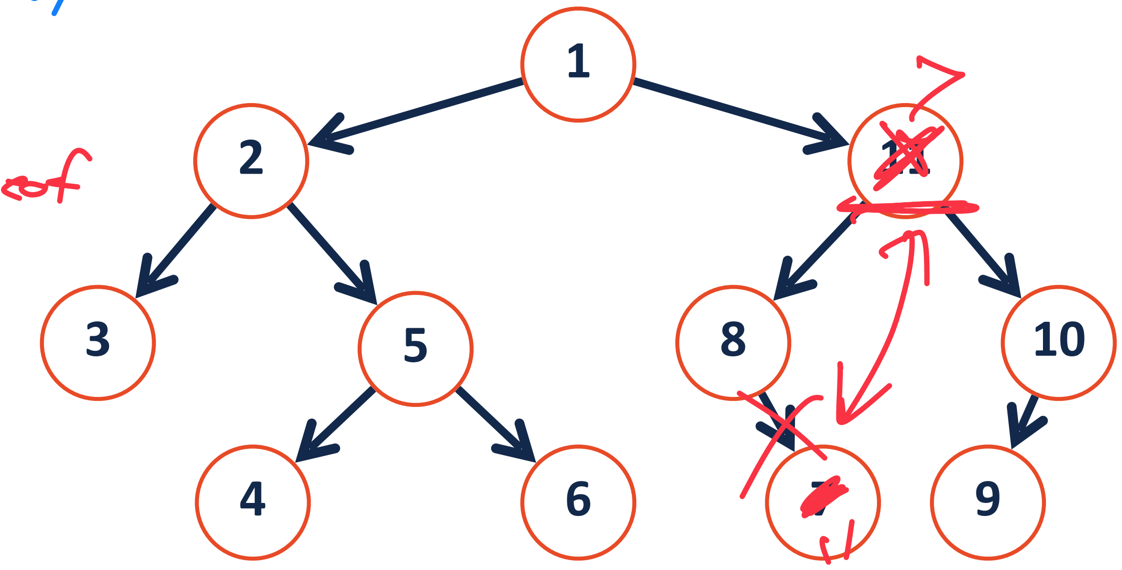
Binary Tree Remove

When we remove, we have to be careful not to delete a tree branch!

Ex: I want to remove the value '11'.

choice (1): Assign order/rules for every combination

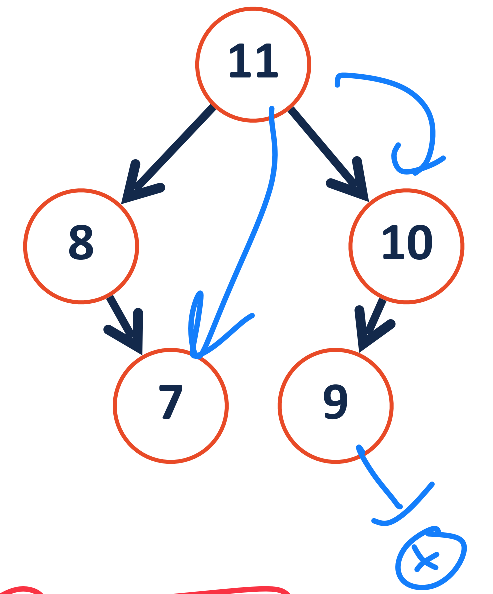
choice (2): Swap removed value w/ leaf
↳ then remove it.



Binary Tree Remove

remove(Node)

Choice: How do we adjust our tree given a removed node?



If the node being removed has 2 children:

1) Descend and find a leaf ← O(n)

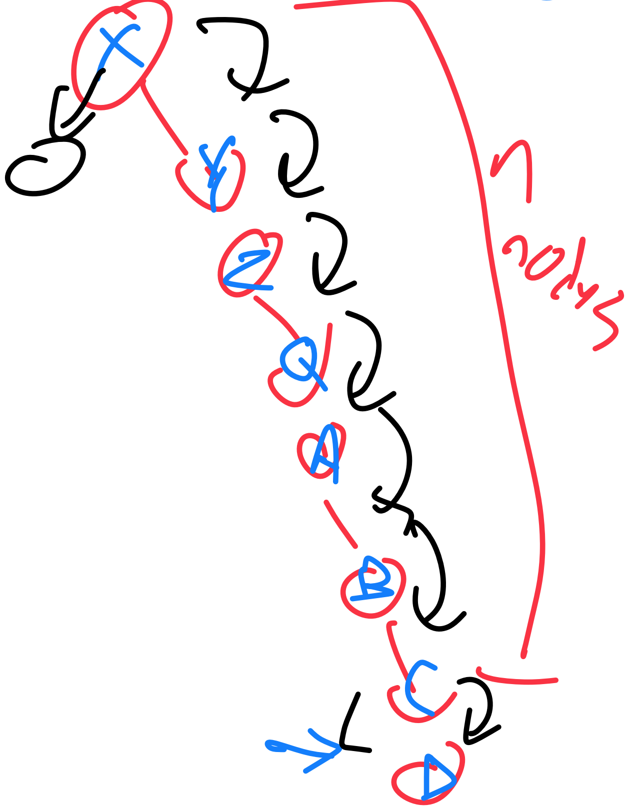
2) Swap value btwn leaf & node ← O(1)

tmpNode(x).val = freeNode(D).val

3) Remove new Node w/ target value

↳ O(1) leaf!

0 child case



Binary Tree Remove Big O

What is the Big O of our removal algorithm on a binary tree?

0 child: $O(1)$

1 child: $O(1)$

2 child: $O(n)$

←

Worst case

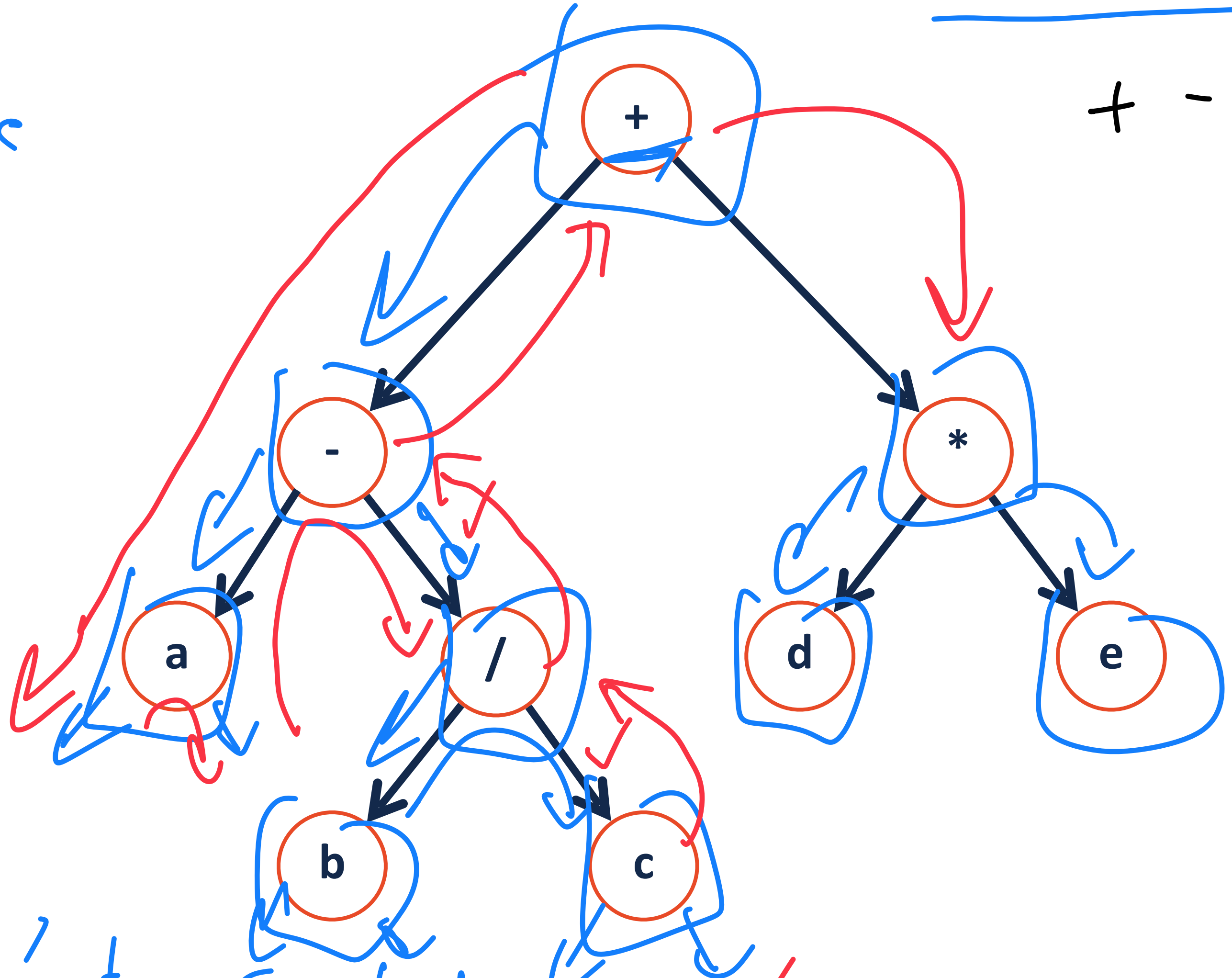
remove is $O(n)$

Tree Traversal

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

At each node
1) Look at Node
2) Recurse left
3) Recurse right

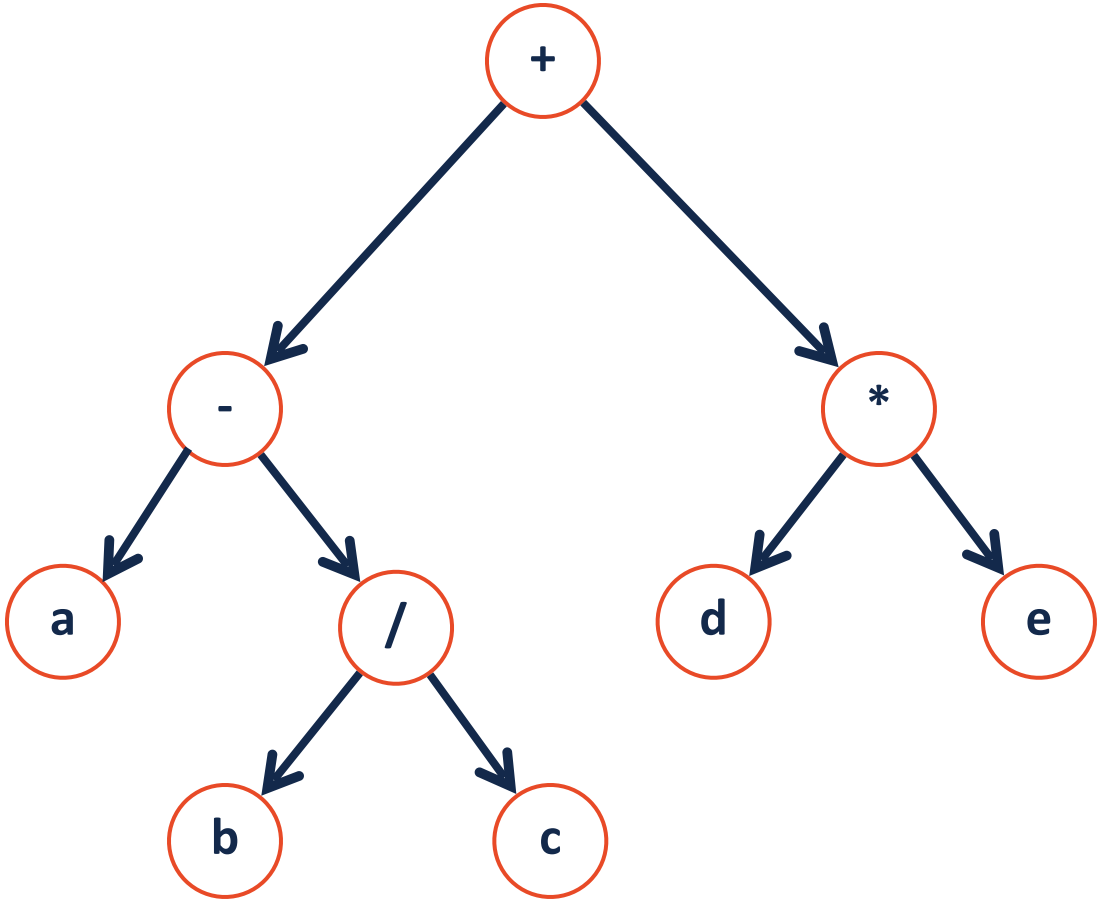
+ - a / b c * d e



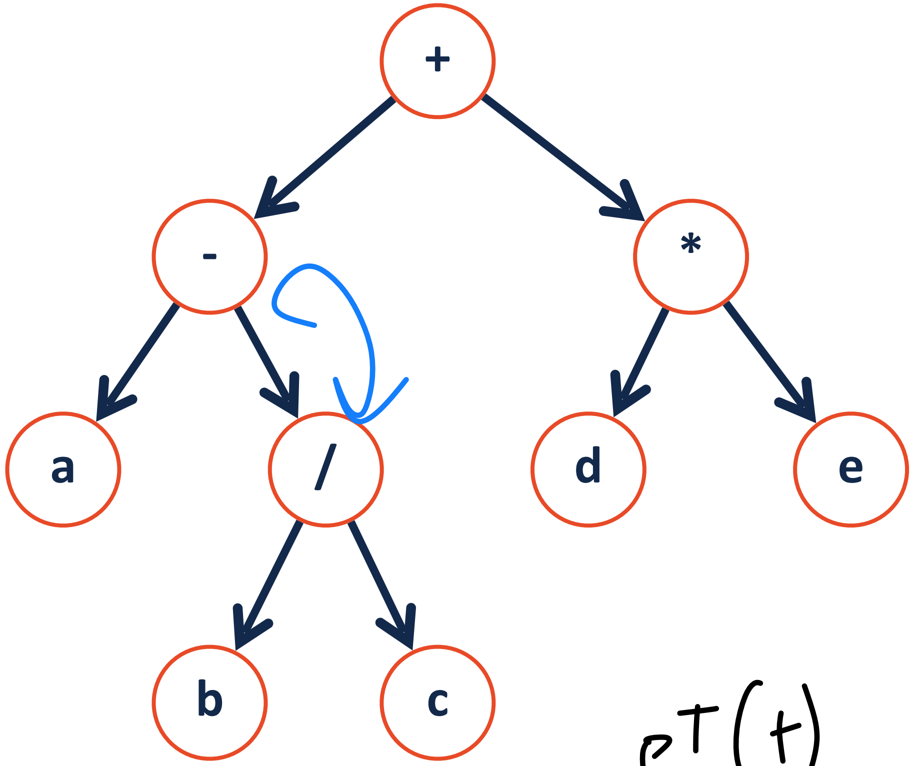
+ - a / b c * d e ✓

Tree Traversal

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



Pre-order Traversal

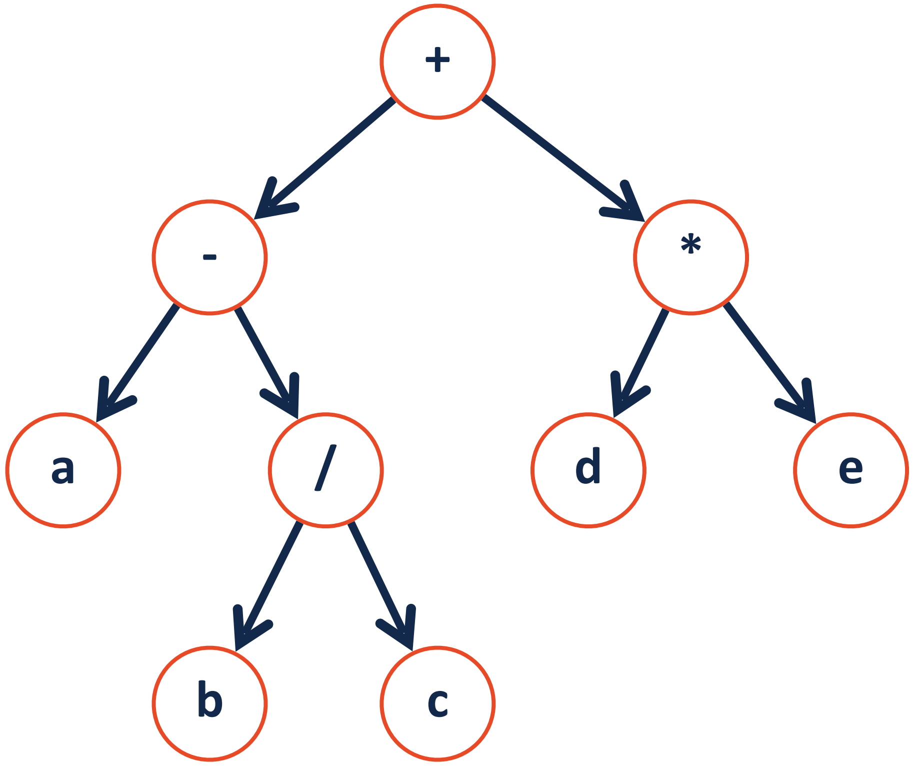


```
def preorderTraversal(node):
    if node:

        print(node.val)


        preorderTraversal(node.left)


        preorderTraversal(node.right)

```

pT(+)
↳ pT(-)
↳ pT(a)
↳ pt(/)

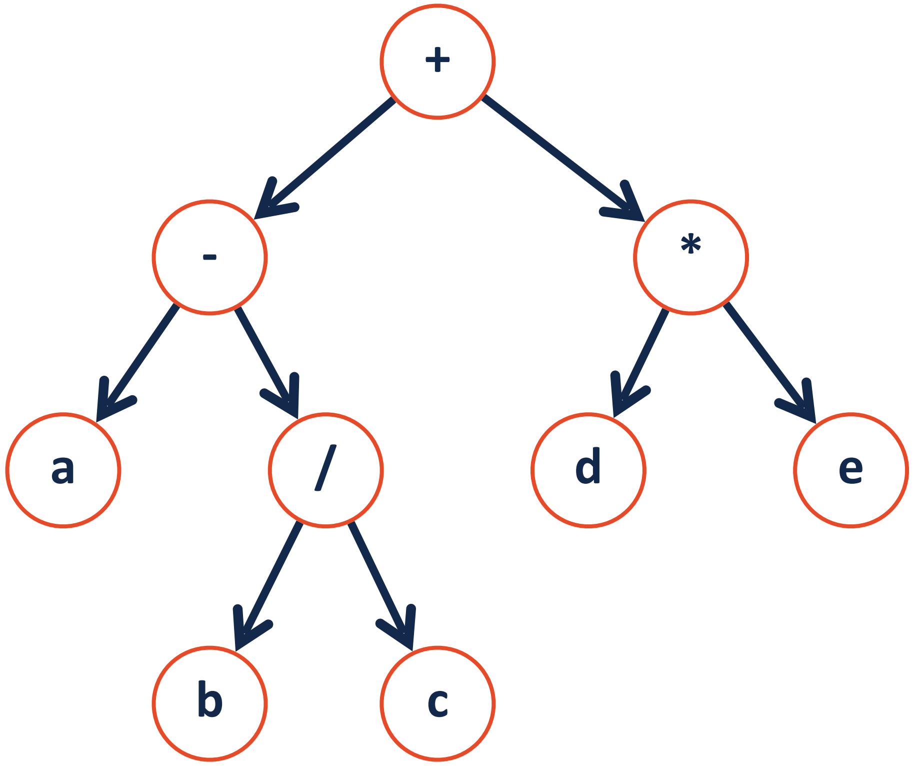
Pre-order: +

In-order Traversal



In-order:

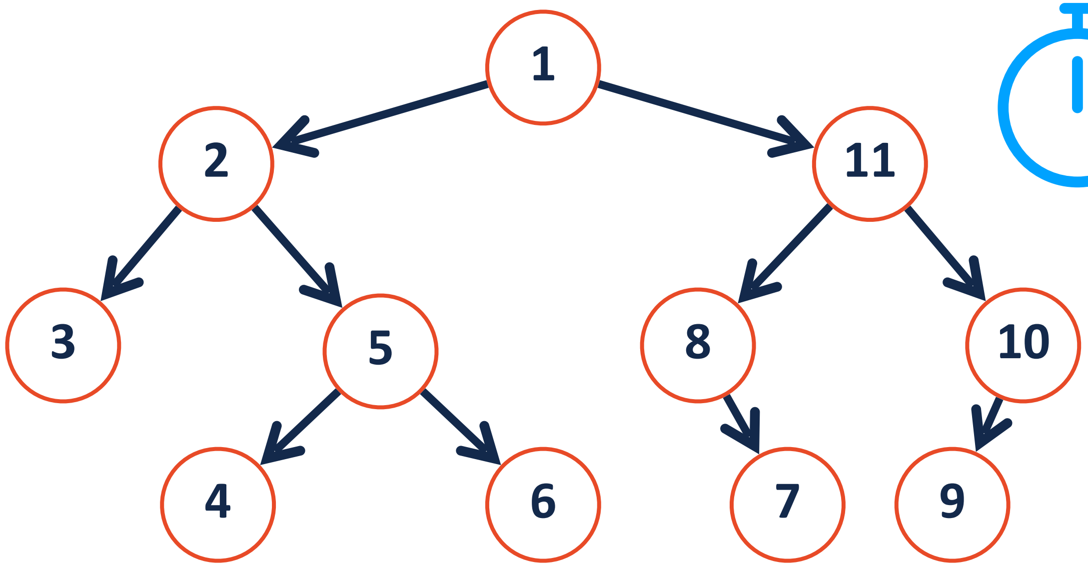
Post-order Traversal



Post-order:

Tree Traversals

Lets practice our traversals!

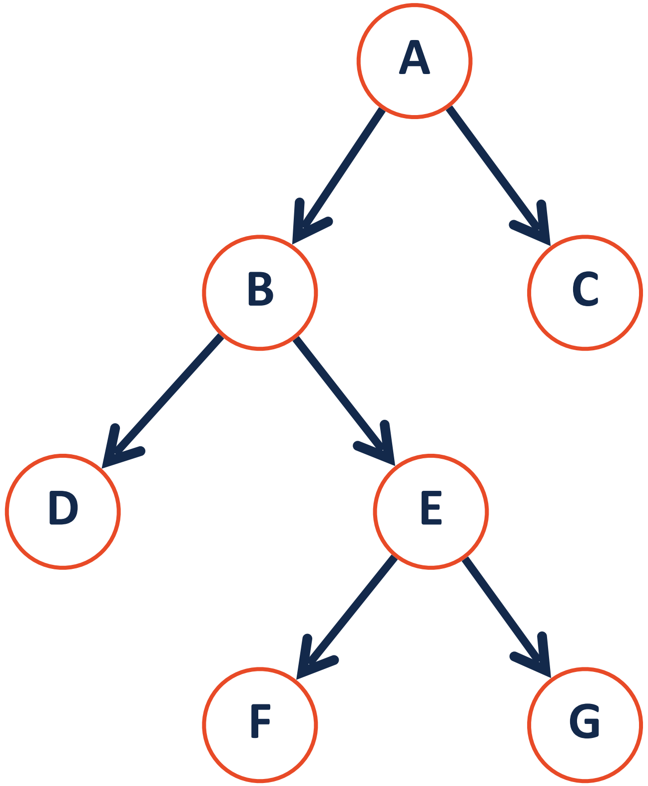


Pre-order:

In-order:

Post-order:

Traversal vs Search

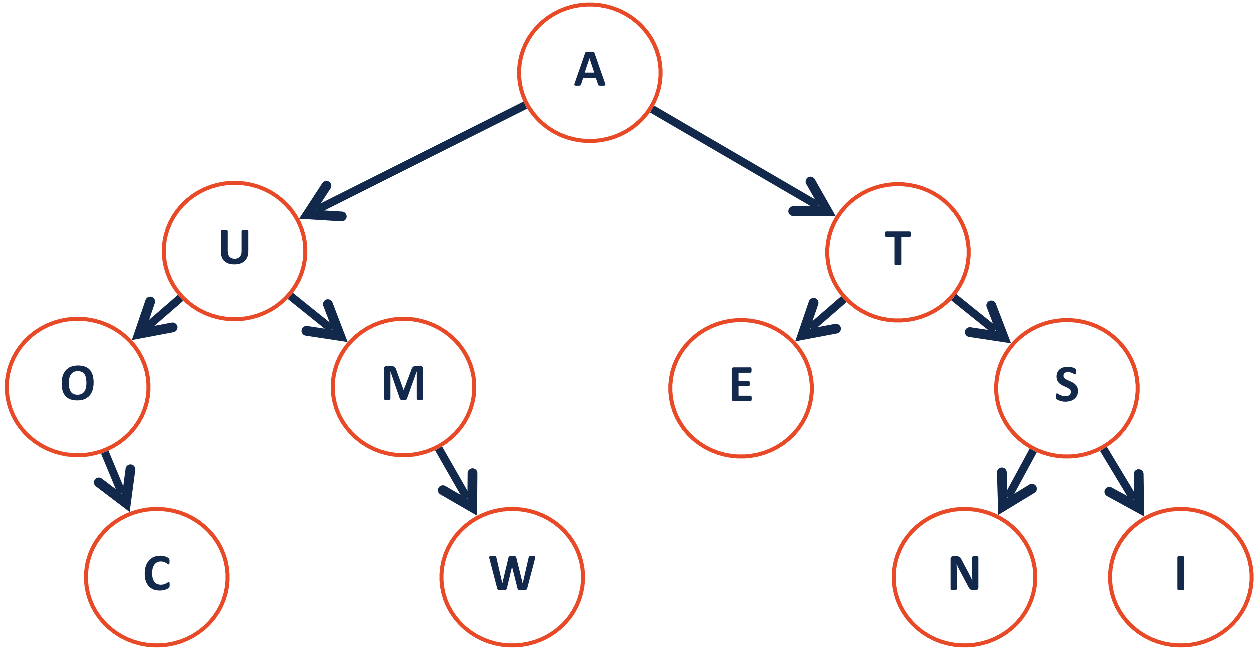


Traversal

Search

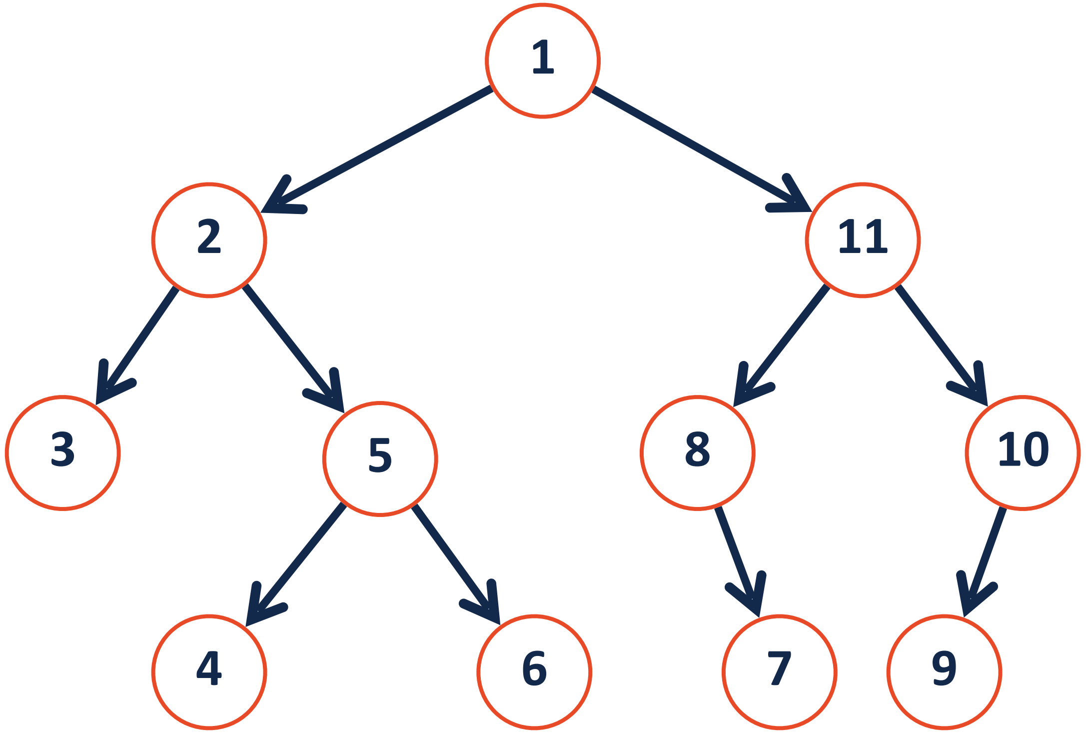
Searching a Binary Tree

There are two main approaches to searching a binary tree:



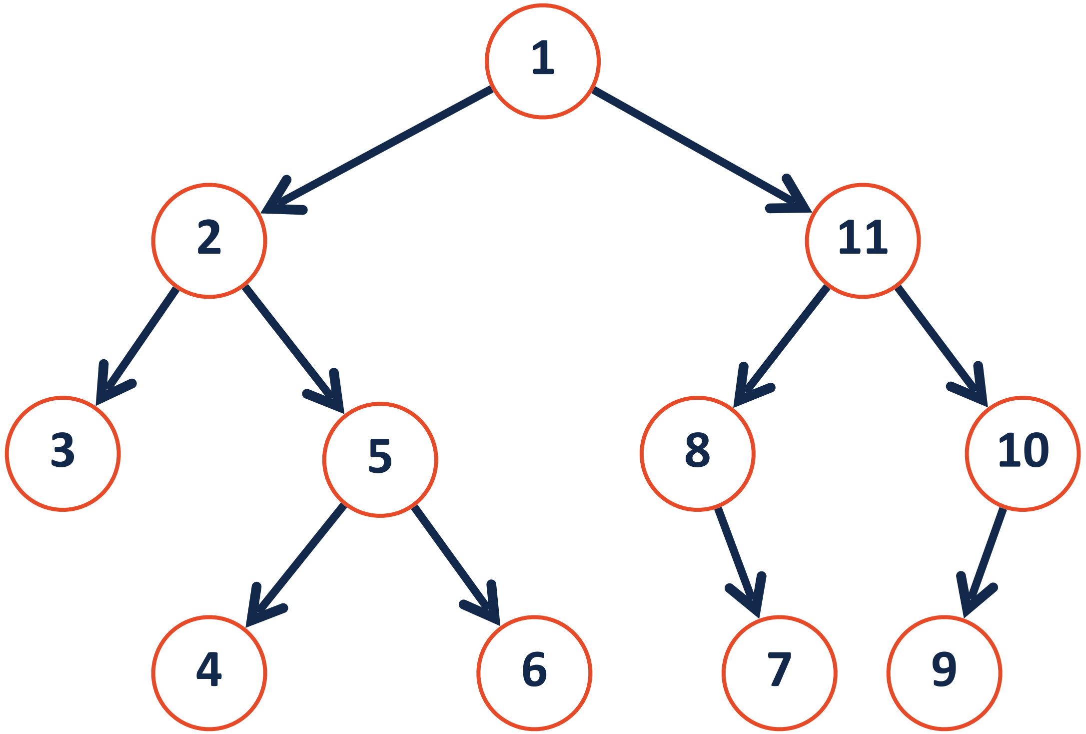
Depth First Search

Explore as far along one path as possible before backtracking



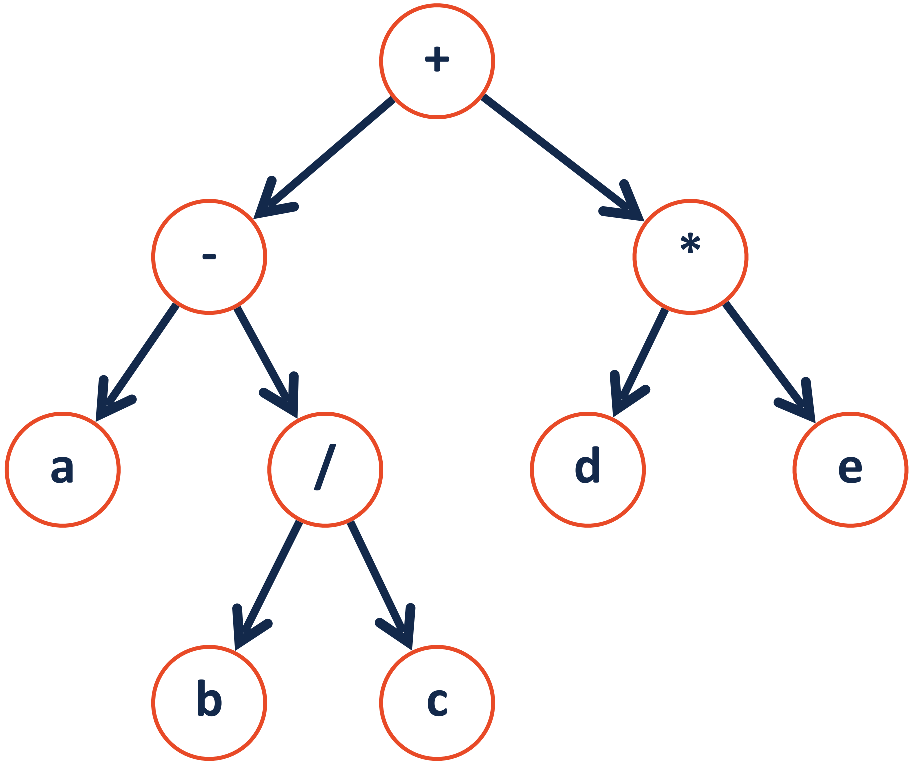
Breadth First Search

Fully explore depth i before exploring depth i+1



Traversal vs Search II

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



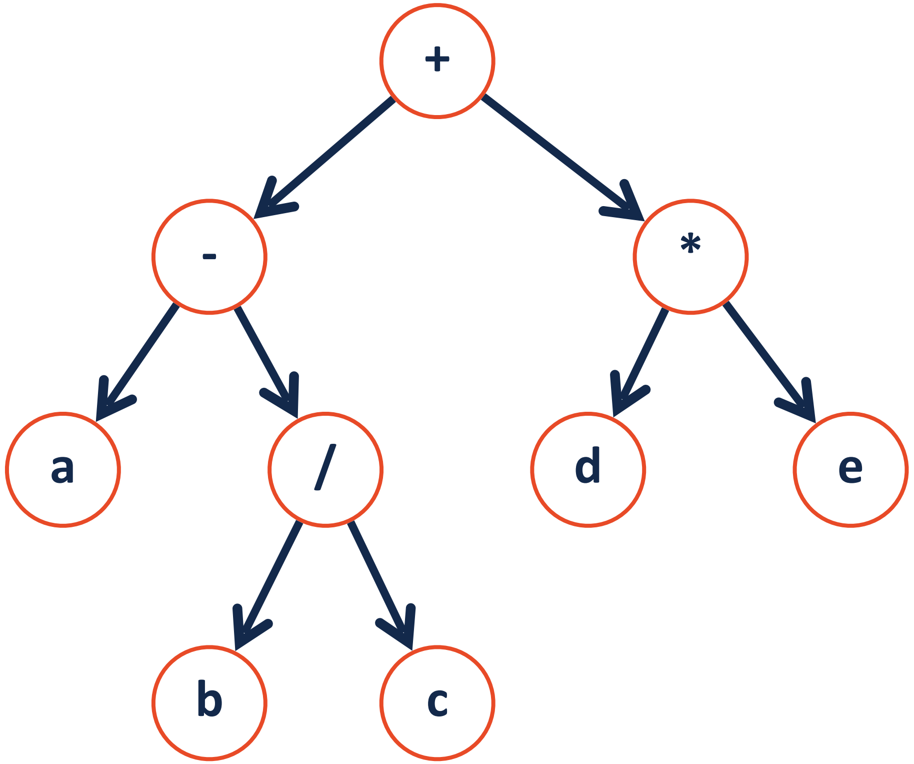
Pre-order: + - a / b c * d e

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

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

Level-Order Traversal

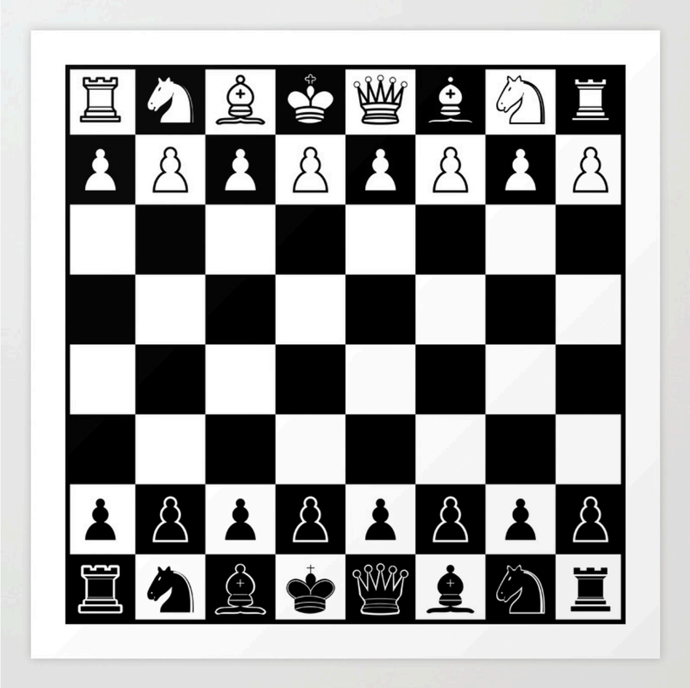
A tricky recursive implementation but an easier queue implementation!



Level-order:

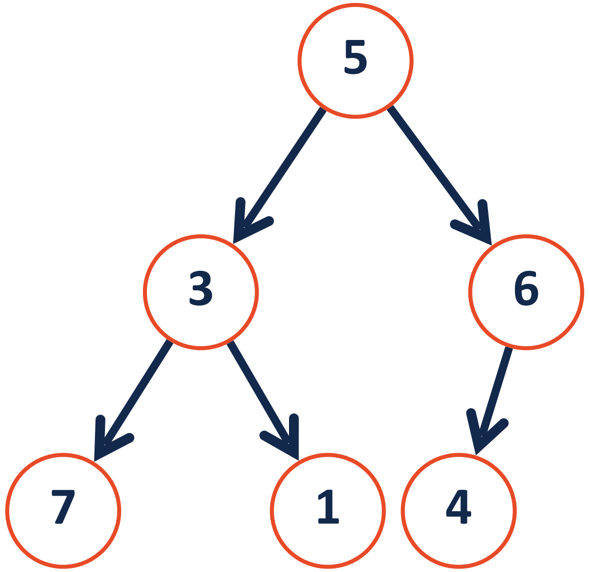
What search algorithm is best?

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?

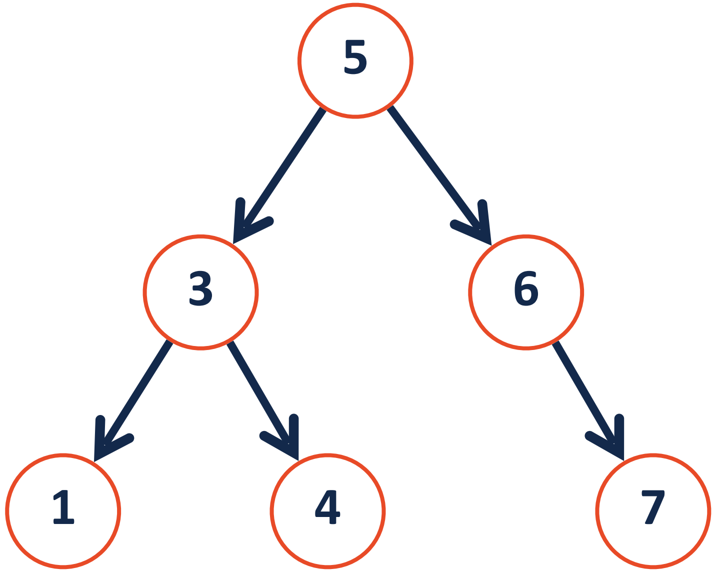


Improved search on a binary tree

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:

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

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