Data Structures

CSCI 2270-202: REC 07

Sanskar Katiyar

Logistics

Make-up for Assignment [Interview Grading]

Assignment 1-4

Sign-up

Midterm 1 Grades

Review your exam in Office Hours

Should be available on Moodle by the end of this week

Logistics

Office Hours at ECAE 128

Wednesday: 3 pm – 5 pm

Thursday: 5 pm – 6 pm

Friday: 3 pm – 5 pm

Recitation Materials *(Notes, Slides, Code, etc.)*

sanskarkatiyar.github.io/CSCI2270

Announcement

Learning Assistant Information Session
TODAY, 5 - 6pm, UMC Room 235

Fall 2020: CSCI 1200, CSCI 1300

Apply: *https://learningassistantalliance.org/*

*Felix Muzny, [**felix.muzny**] & Rachel Cox, [**rachel.cox**]*

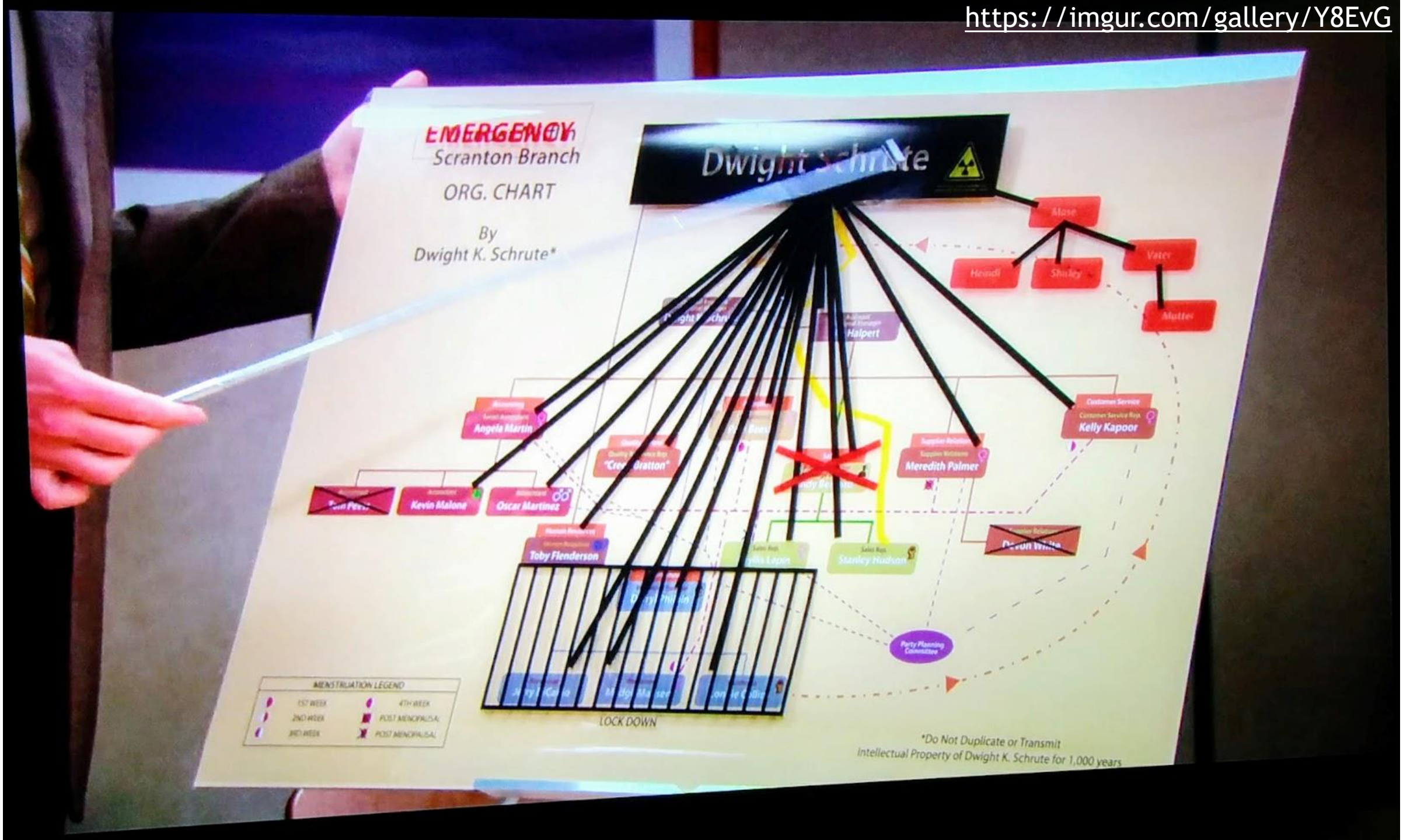
Recitation Outline

1. Tree ADT
2. Binary Tree (BT)
3. Recursion
4. Traversal in a BT
5. Exercise

Tree ADT

https://imgur.com/gallery/Y8EvG

https://imgur.com/gallery/Y8EvG



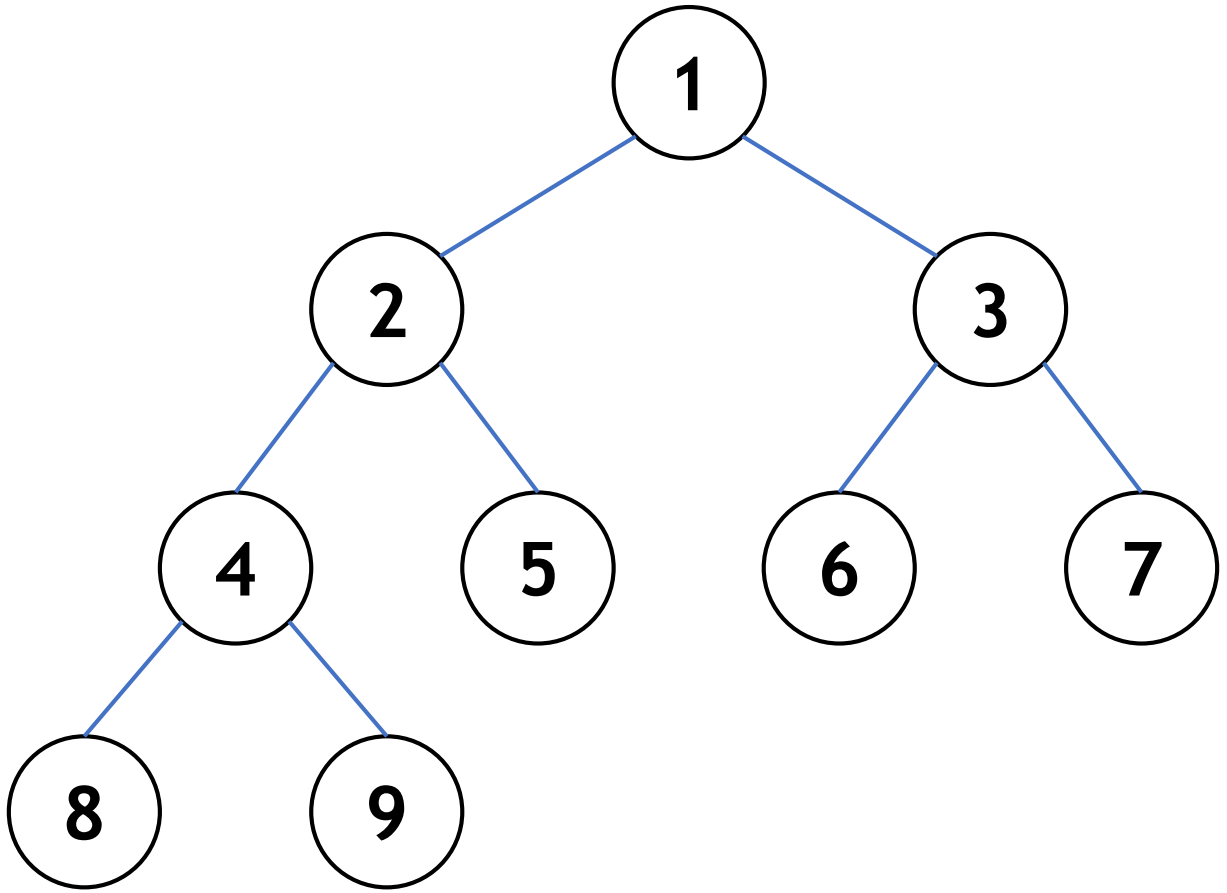
Tree ADT

Composed of **Nodes, Edges**

Non-linear, Hierarchical

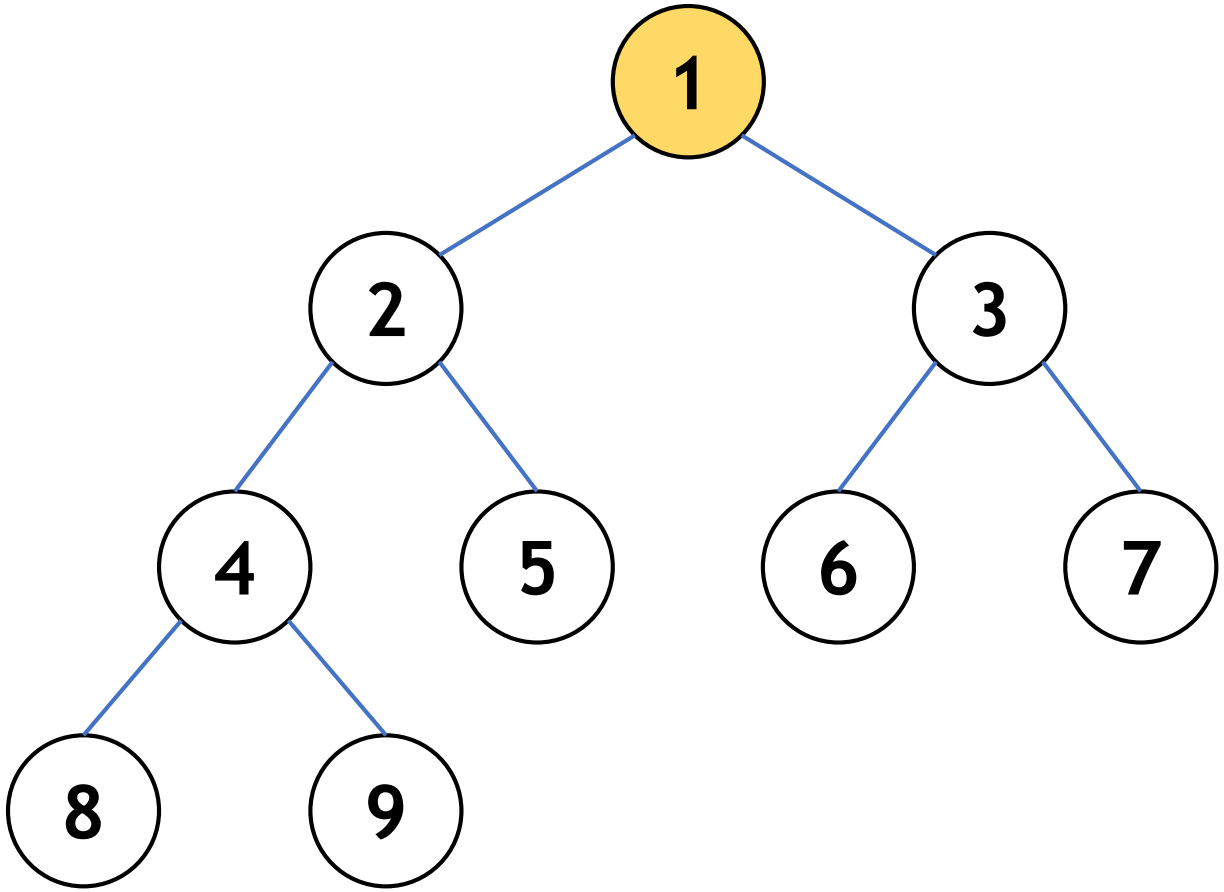
n-ary: Each node can have ***n*** children

Example(s), Application(s):
Search, Expression Trees, Directory Structure



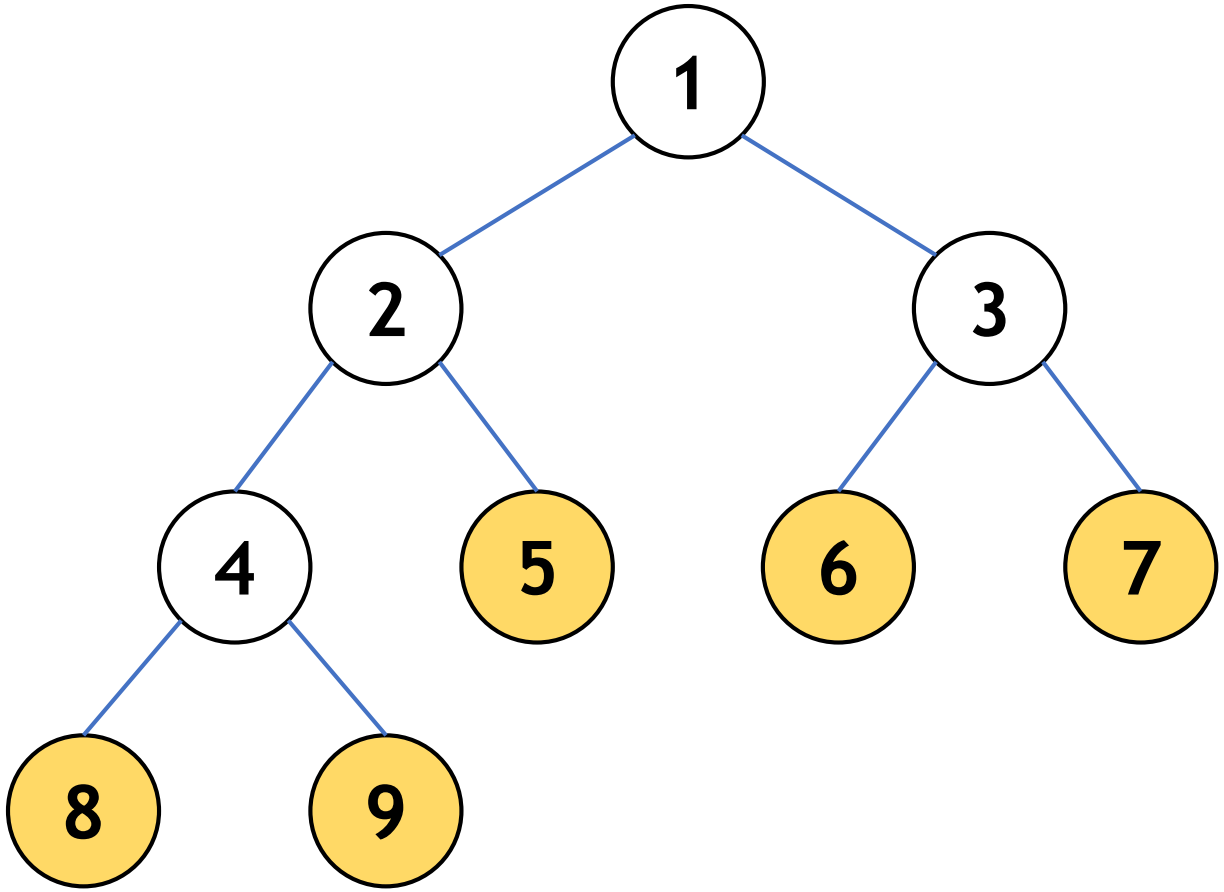
Tree ADT: Root

Recall: *head* in a Linked List



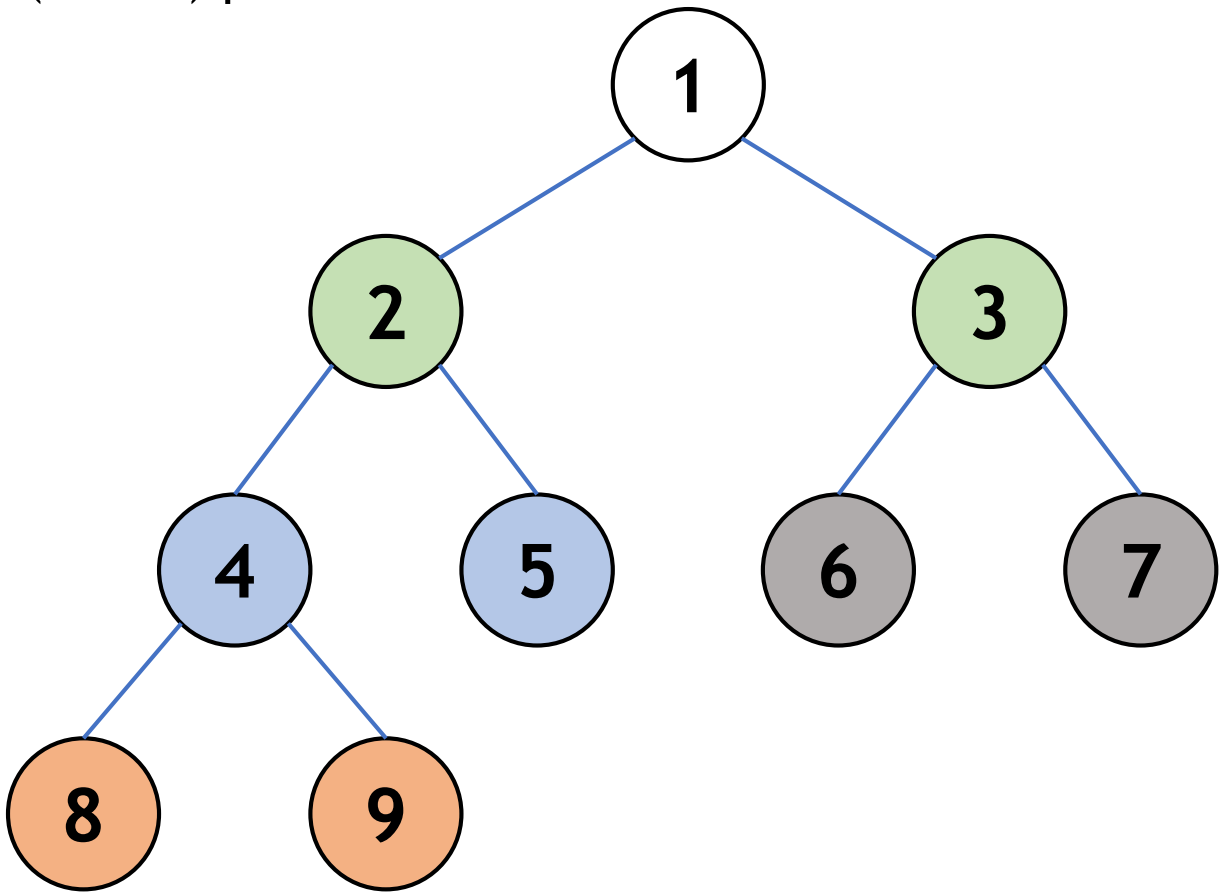
Tree ADT: Leaves

Nodes with no children



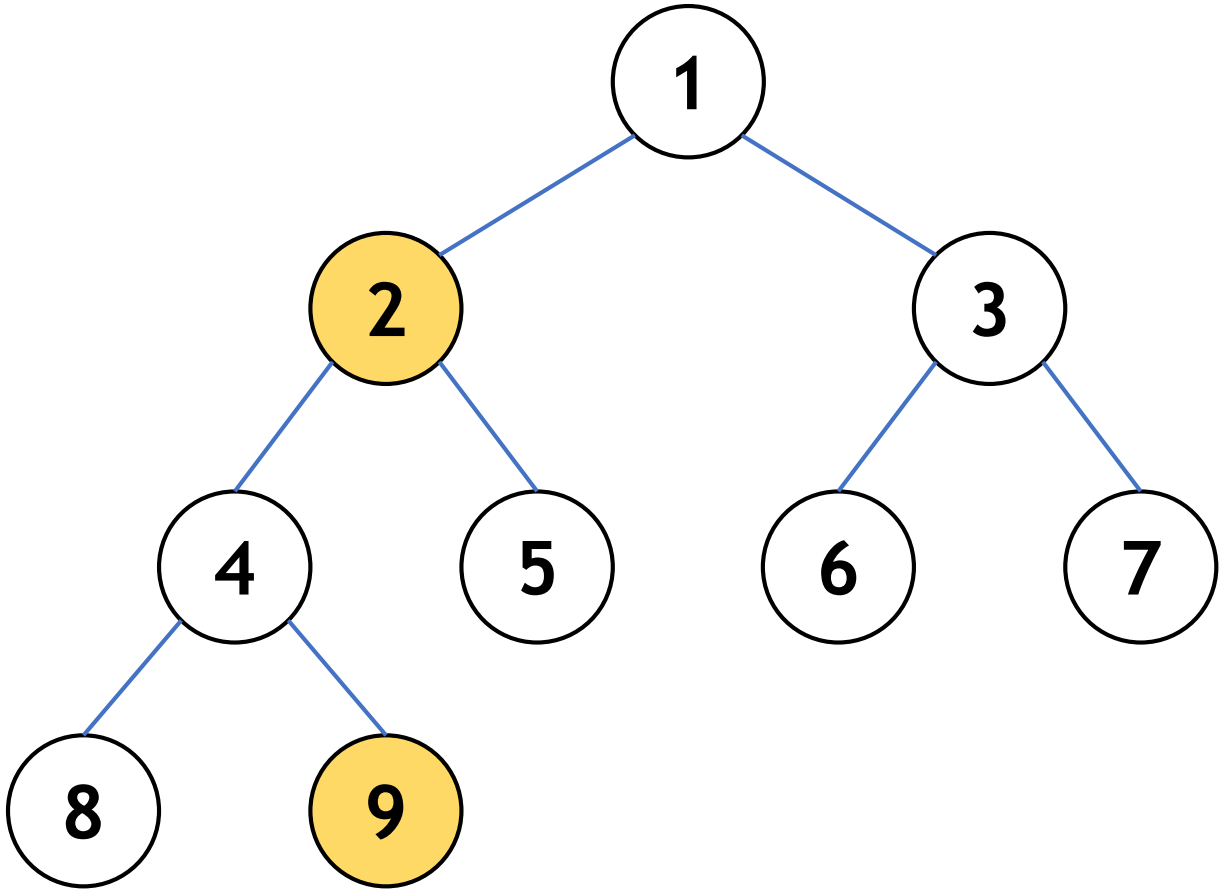
Tree ADT: Siblings

Nodes with common (direct) parent



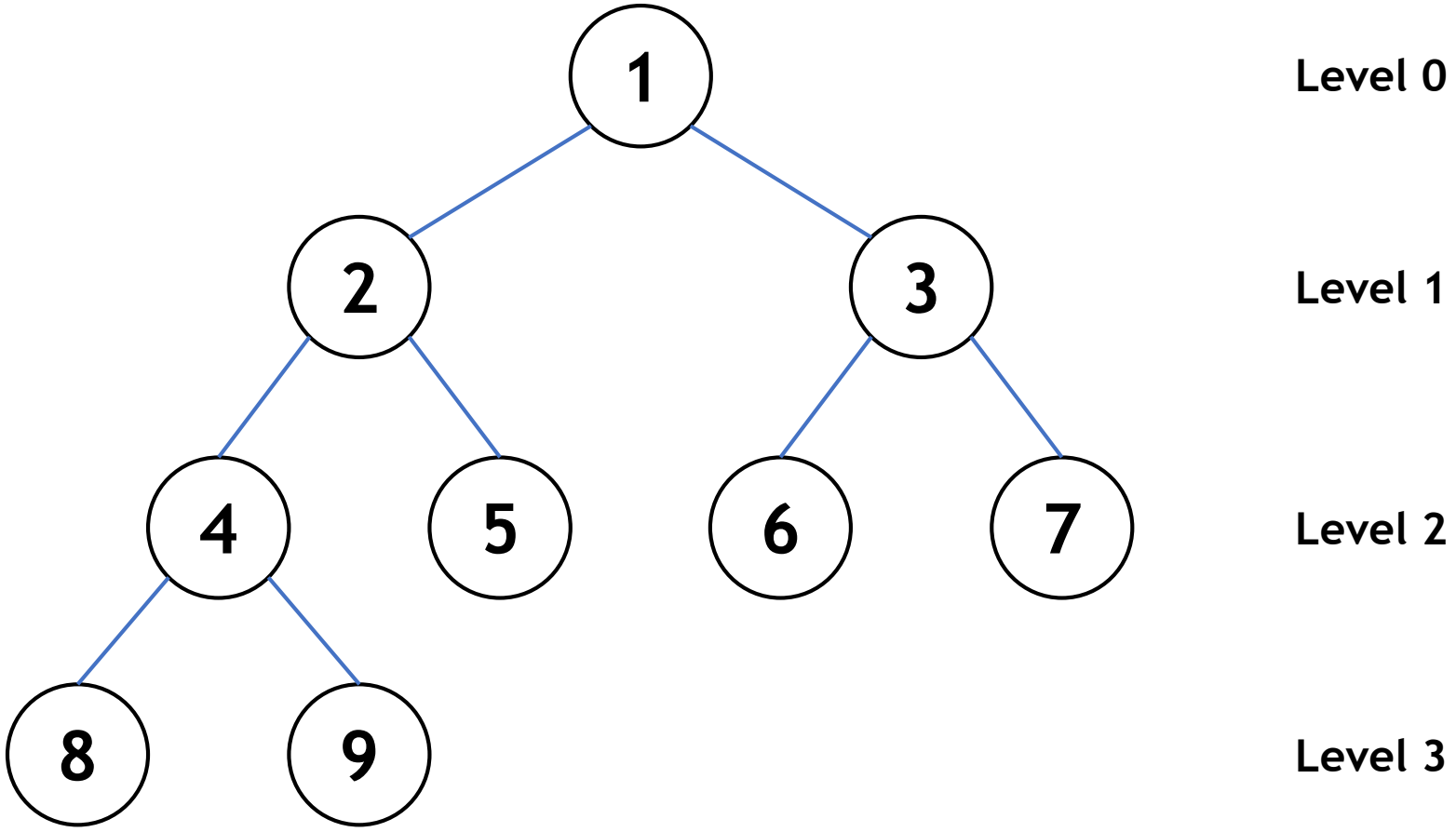
Tree ADT: Ancestor

A node p is an ancestor of node q if there exists a path from root to q and p appears on the path



Tree ADT: Levels

Number of hops from the root



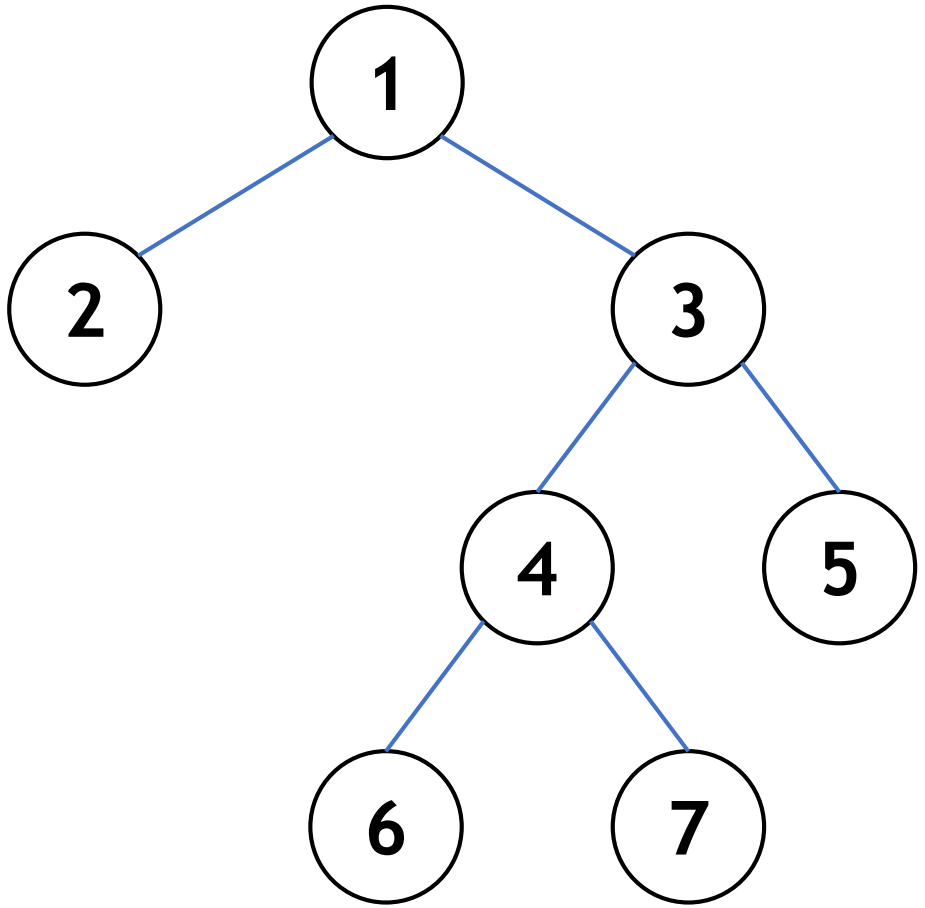
Binary Tree

Each node has {0, 1, 2} children

Binary Tree: Full BT

Proper BT

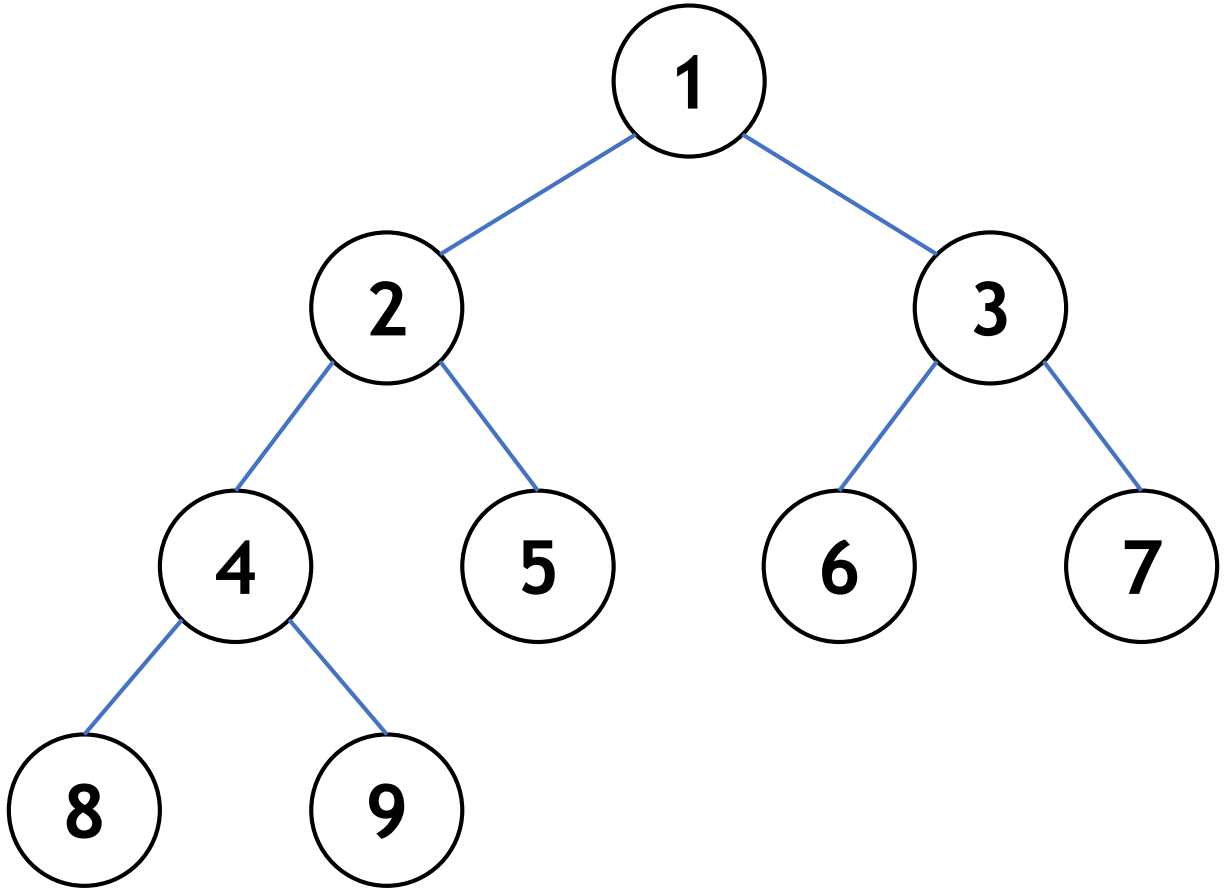
Each non-leaf node has exactly two children



Binary Tree: Complete BT

Each non-leaf level is completely filled

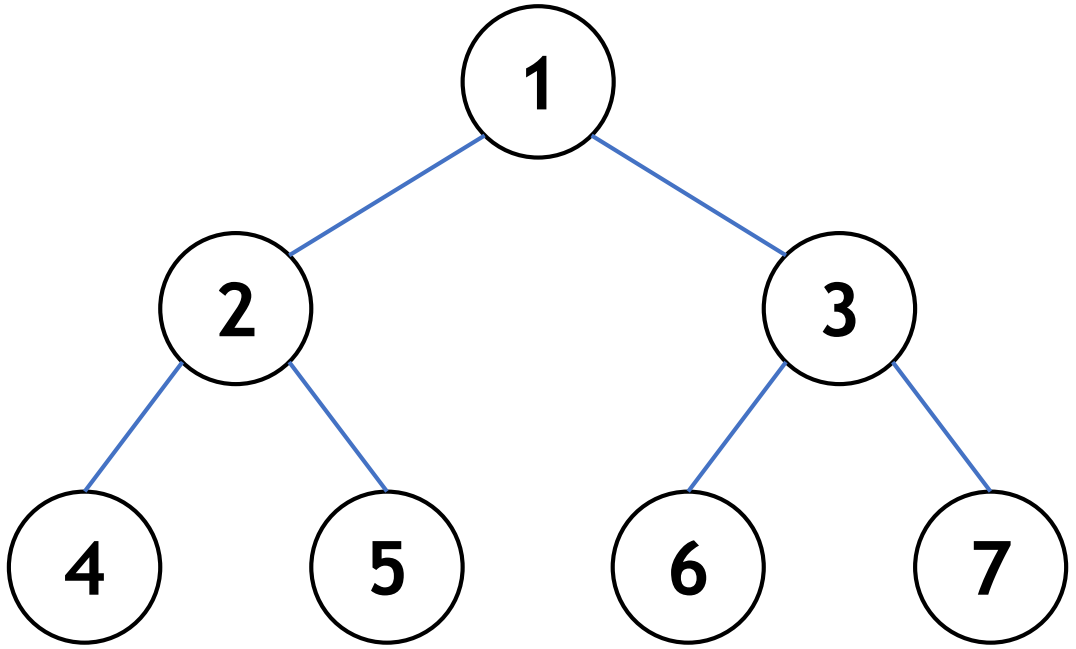
Leaf nodes are as far left as possible



Binary Tree: Perfect BT

Each non-leaf node has exactly two children

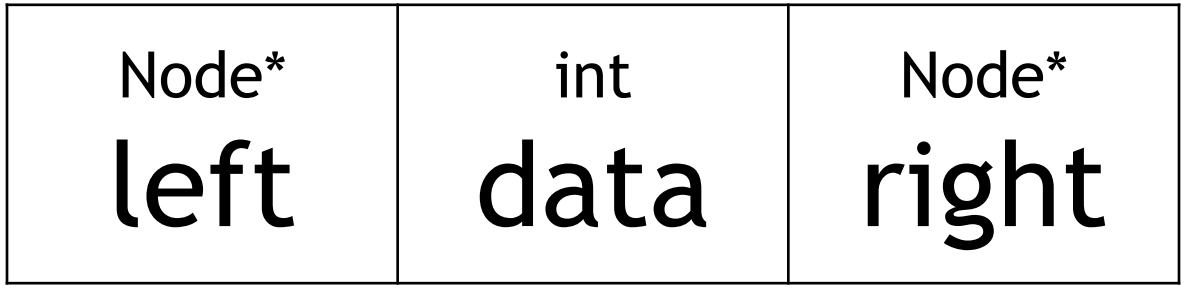
All leaf nodes are at the same level



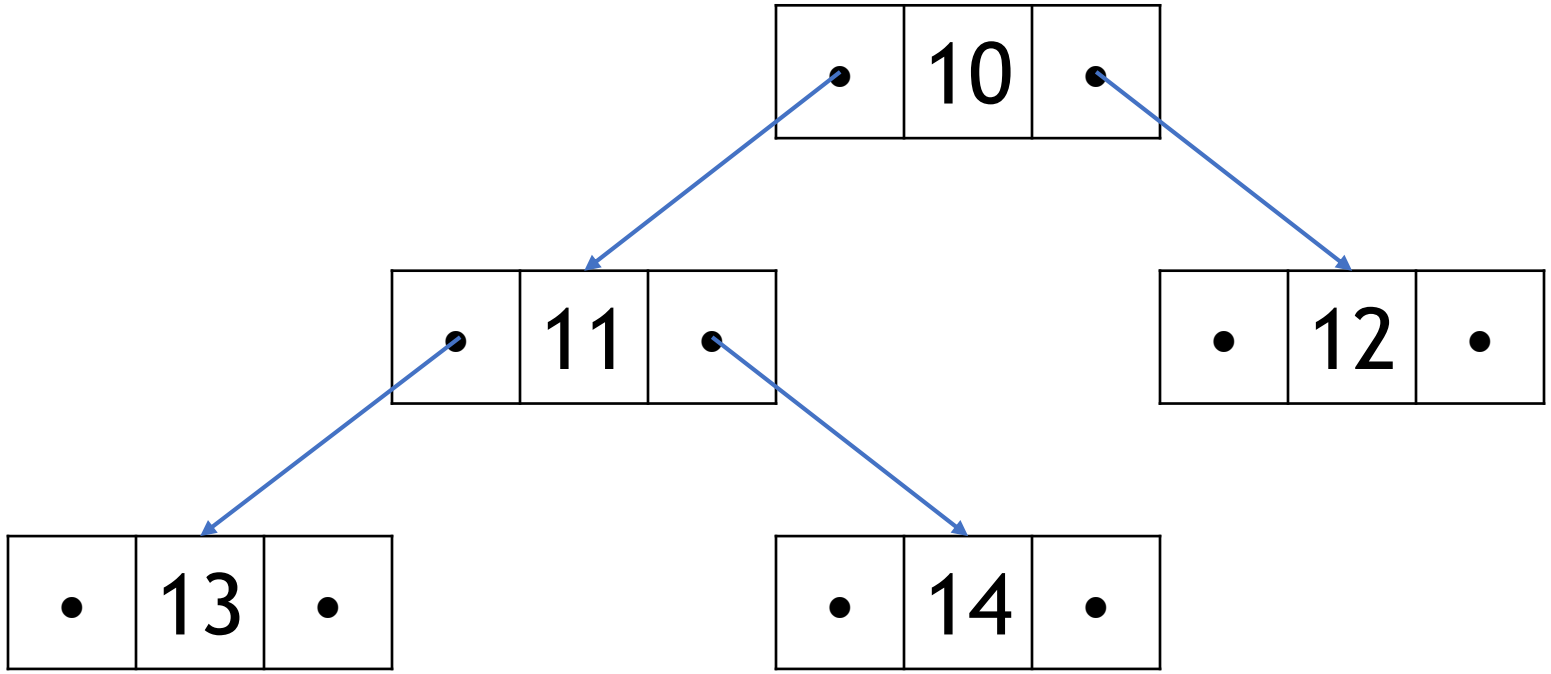
Binary Tree: Implementation

Recall: Implementation of a Node in a Linked List

```
struct Node
{
    int data;
    Node *left;
    Node *right;
};
```



Binary Tree: Implementation



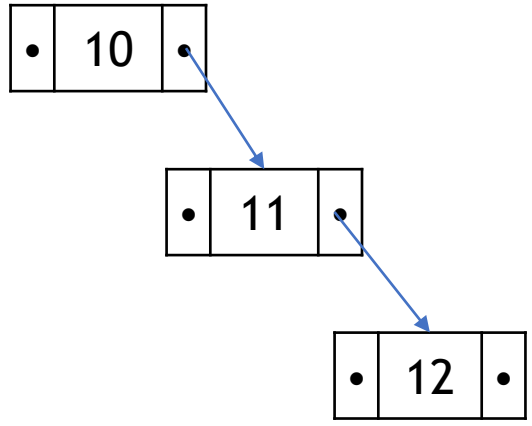
Binary Tree: Pop Quiz

Can a Linked List mimic a tree? (Consider all cases)

No. Tree is non-linear; LL is linear.

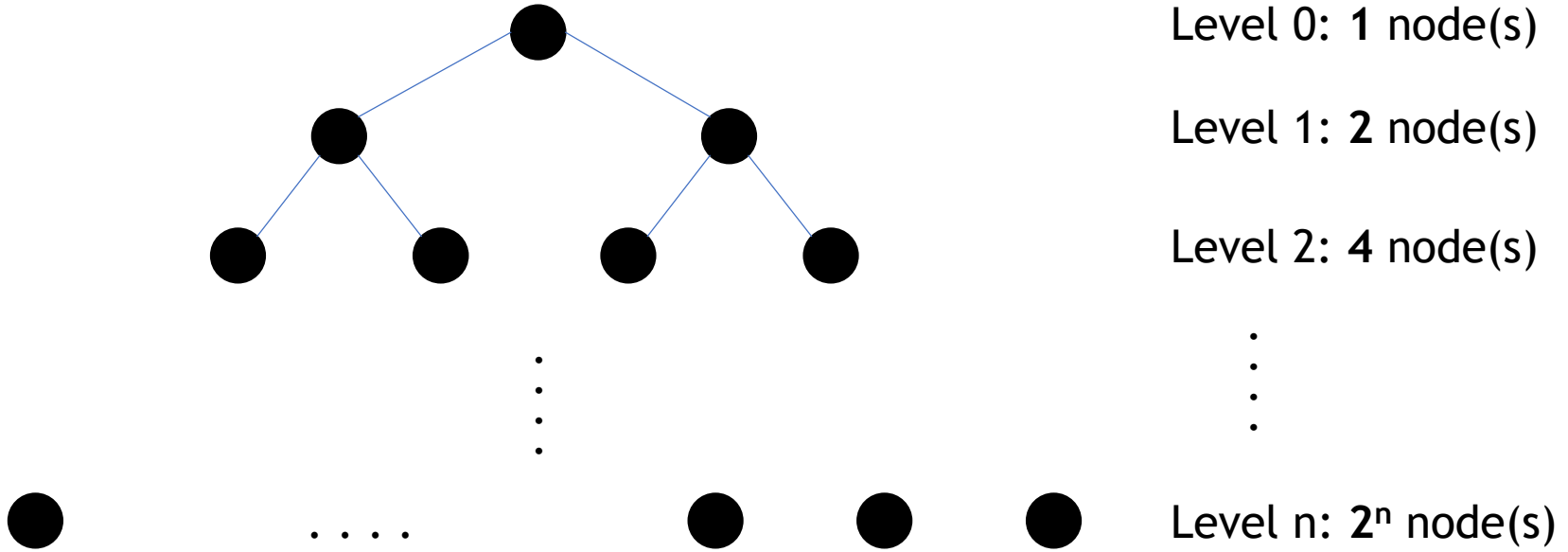
Can a Tree mimic a Linked List?

Yes! How?



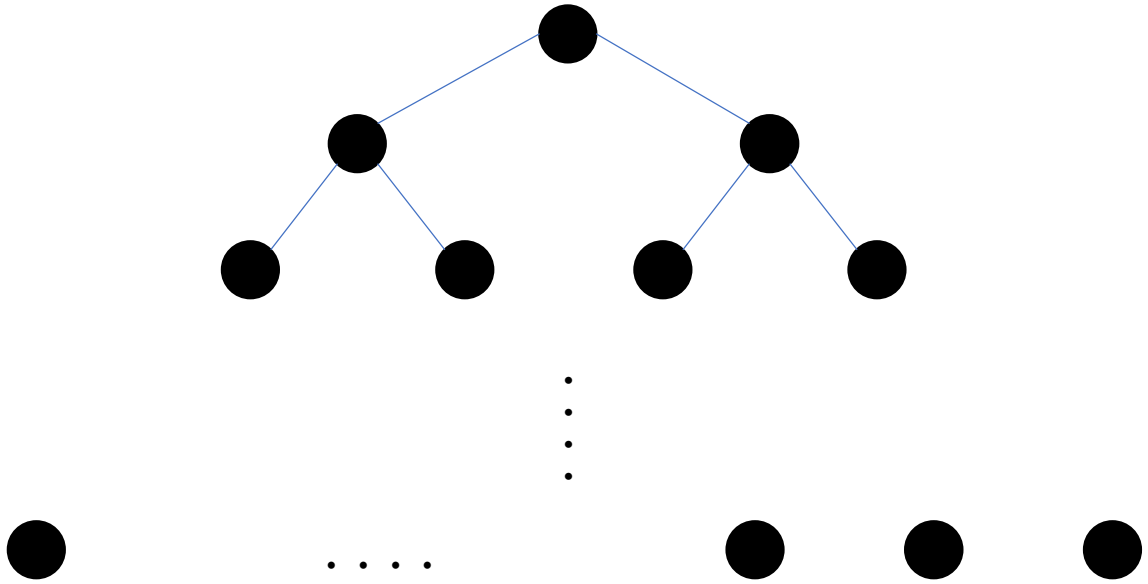
Binary Tree: Nodes, Levels

*Calculate the number of leaf nodes in a level-**n** perfect BT.*



Binary Tree: Nodes, Levels

*Calculate the total number of nodes in a level-**n** perfect BT.*



Level 0: **1** node(s)

Level 1: **2** node(s)

Level 2: **4** node(s)

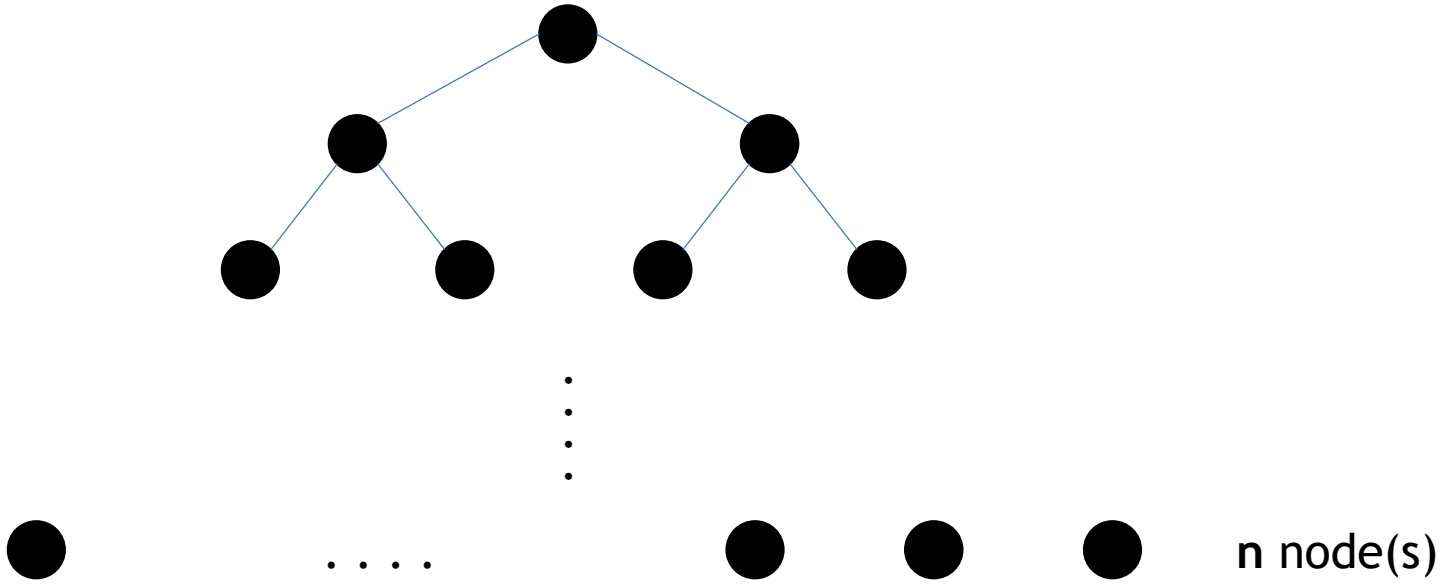
Level n: 2^n node(s)

Sum all the #nodes per level of a Geometric Series:

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

Binary Tree: Nodes, Levels

*Calculate the number of levels in a perfect BT with **n** leaves.*



#times you need to divide by 2, in order to reduce n to 1:

$$\log_2 n$$

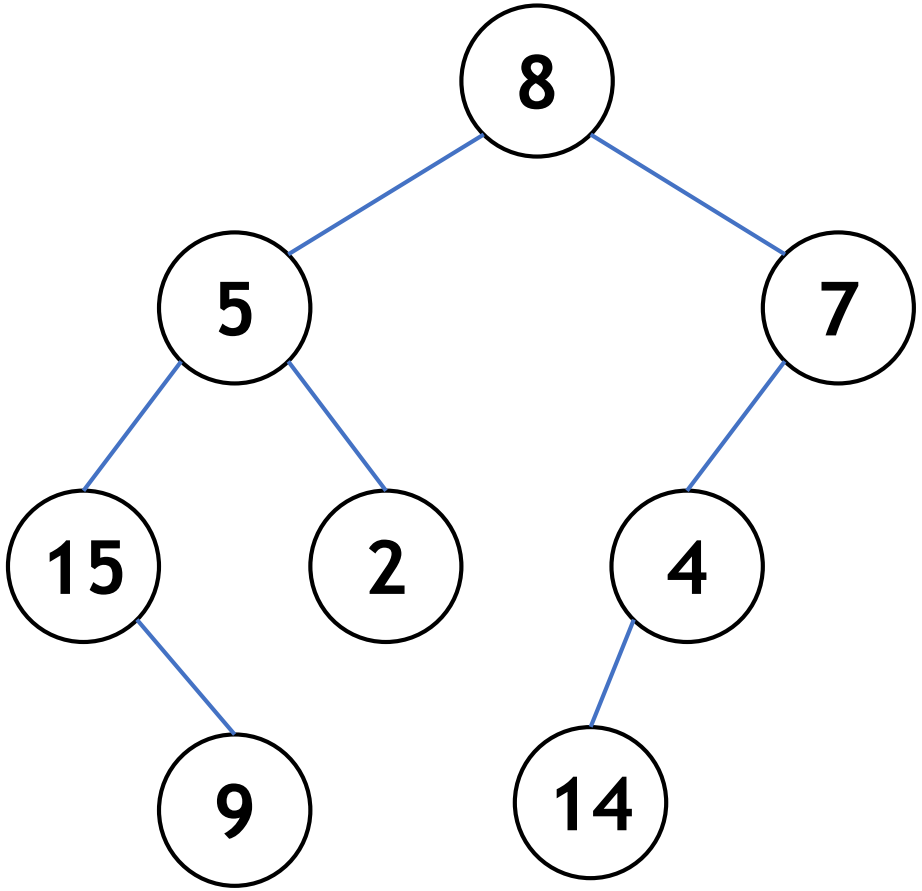
Binary Tree: Pop Quiz

What is the value of root?

8.

Which one are the leaves?

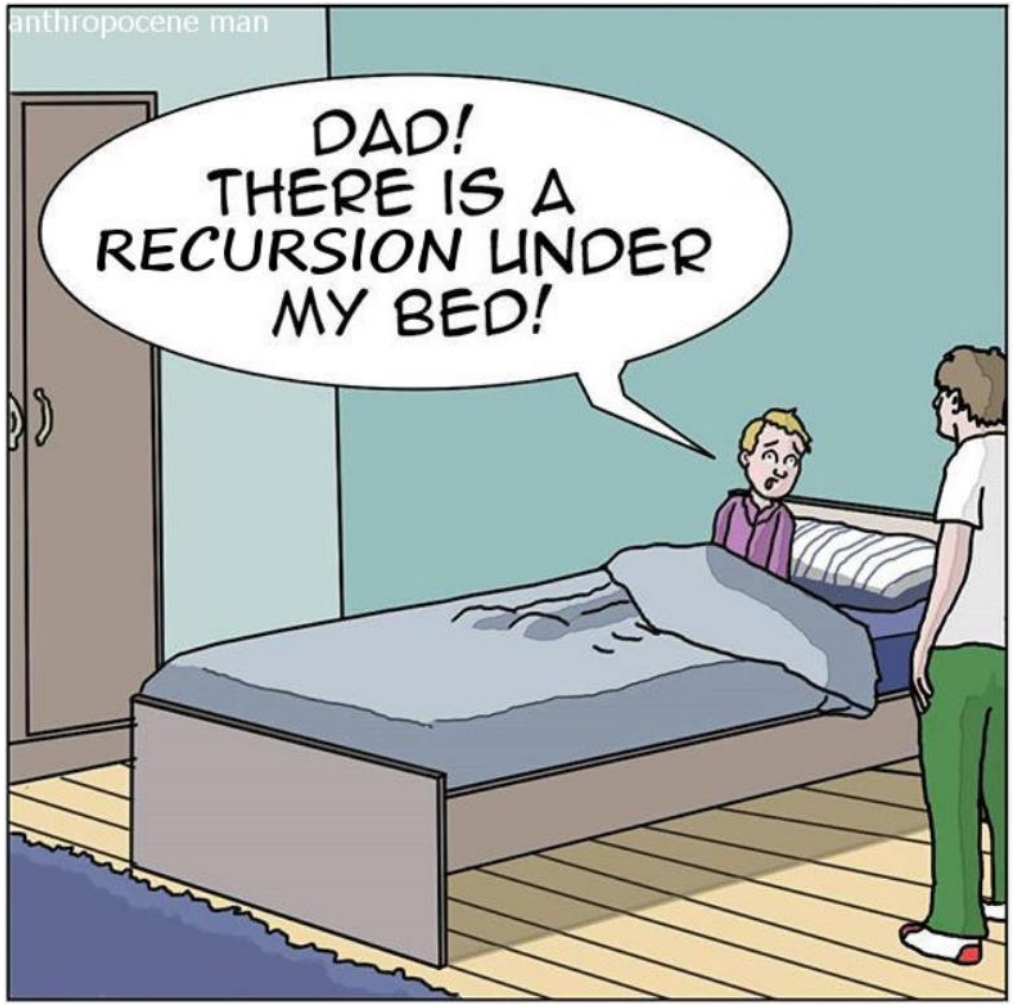
{9, 14, 2}

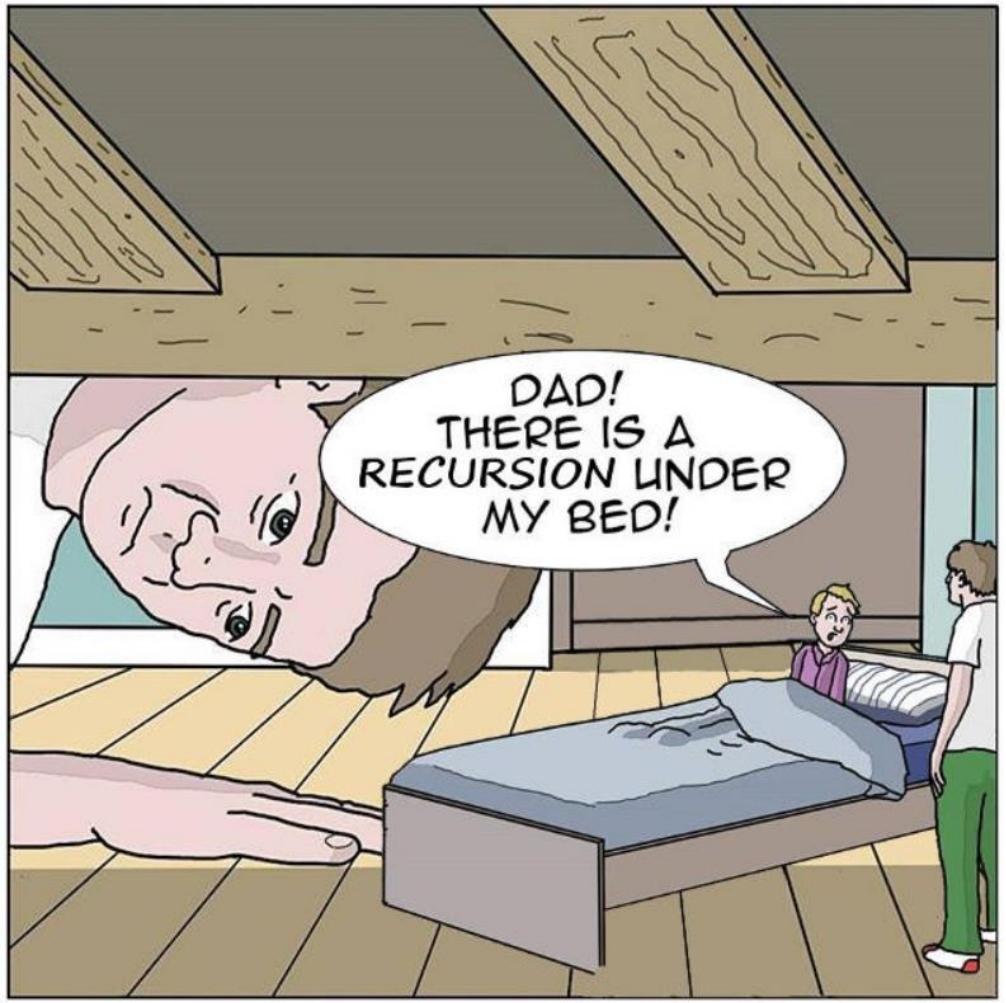


Recursion

Recursion

Medium: @cristy.lucke





Recursion

A function that ***calls itself***.

Requires a ***base case*** and a ***recursive step***.

Examples:

Factorial(x)

Fibonacci(n)

```
Factorial(x):

if x == 1
    return 1;

return x * Factorial(x-1);
```

```
Fib(x):

if x <= 1
    return x;

return Fib(x-1)+Fib(x-2);
```

Recursion: Factorial

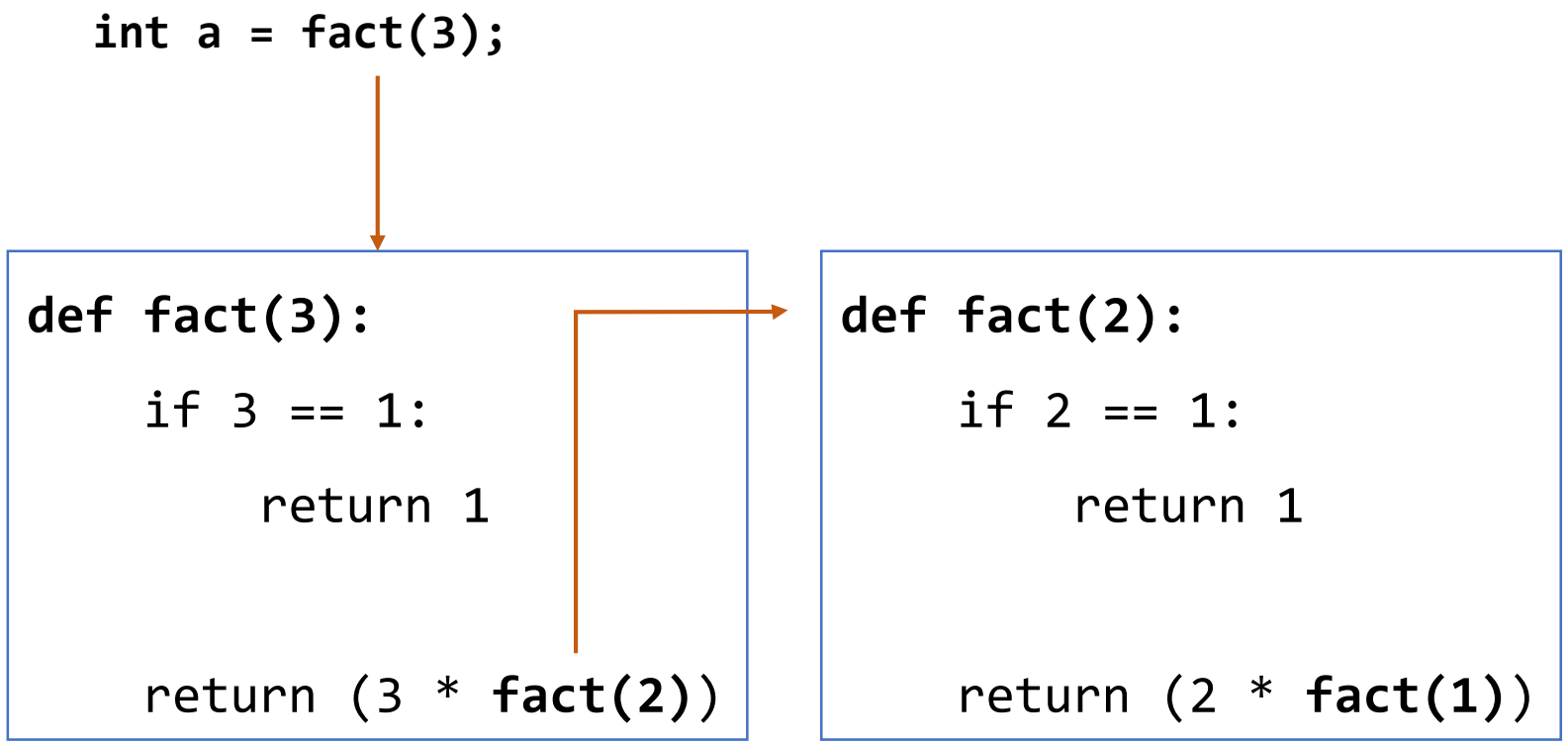
```
int a = fact(3);
```



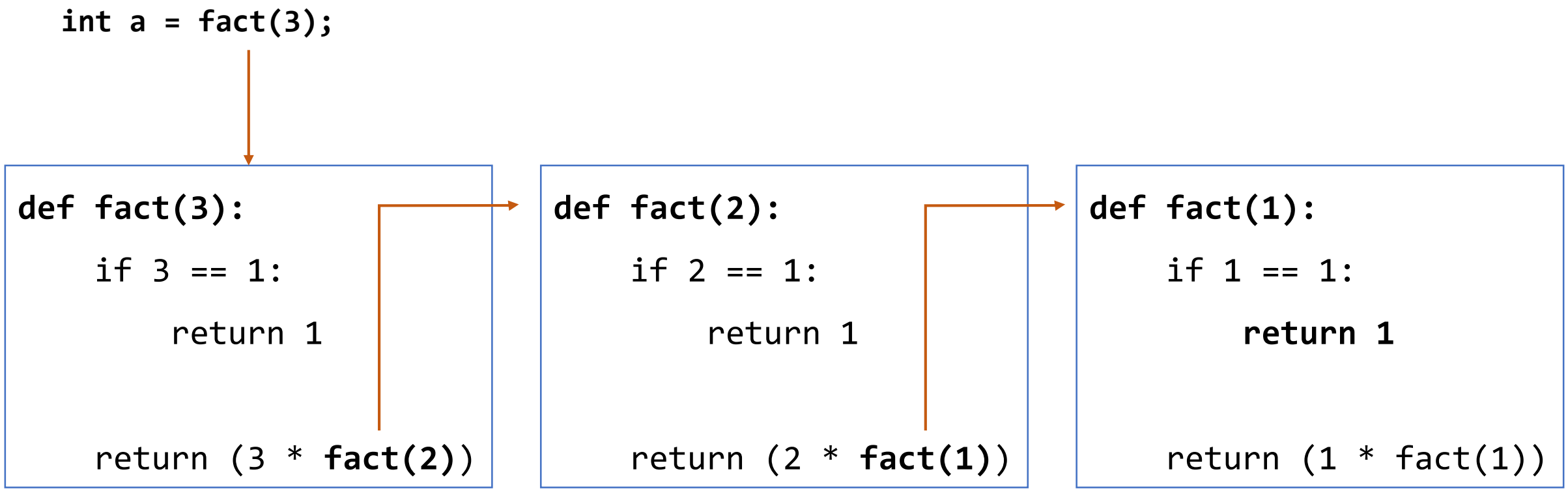
```
def fact(3):
    if 3 == 1:
        return 1

    return (3 * fact(2))
```

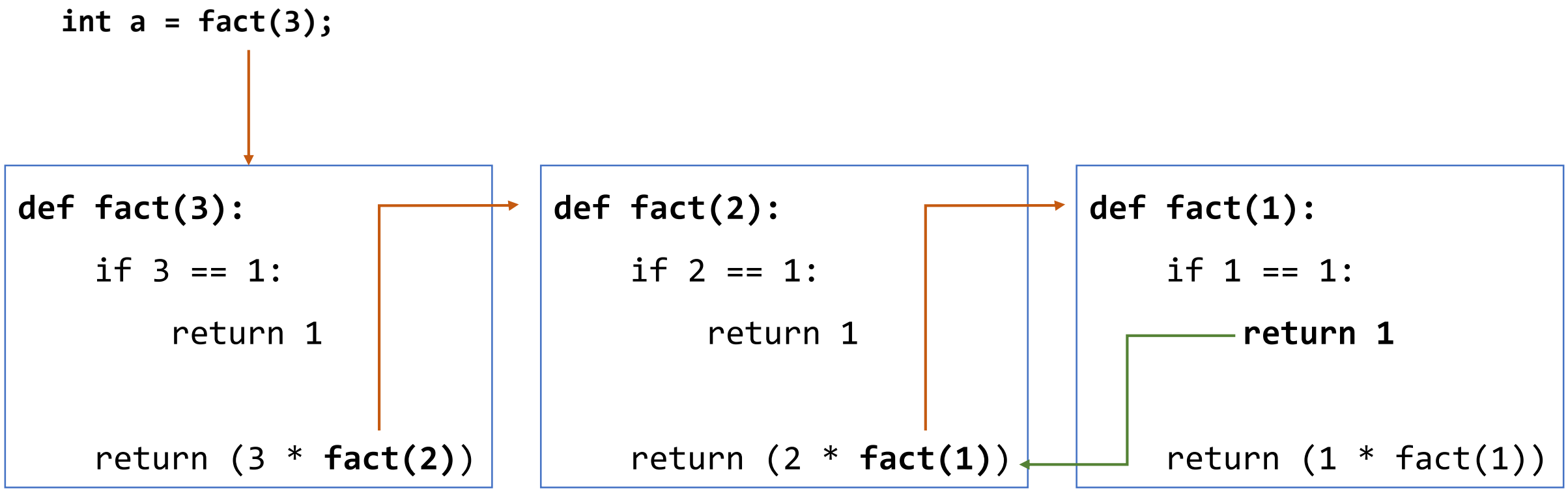
Recursion: Factorial



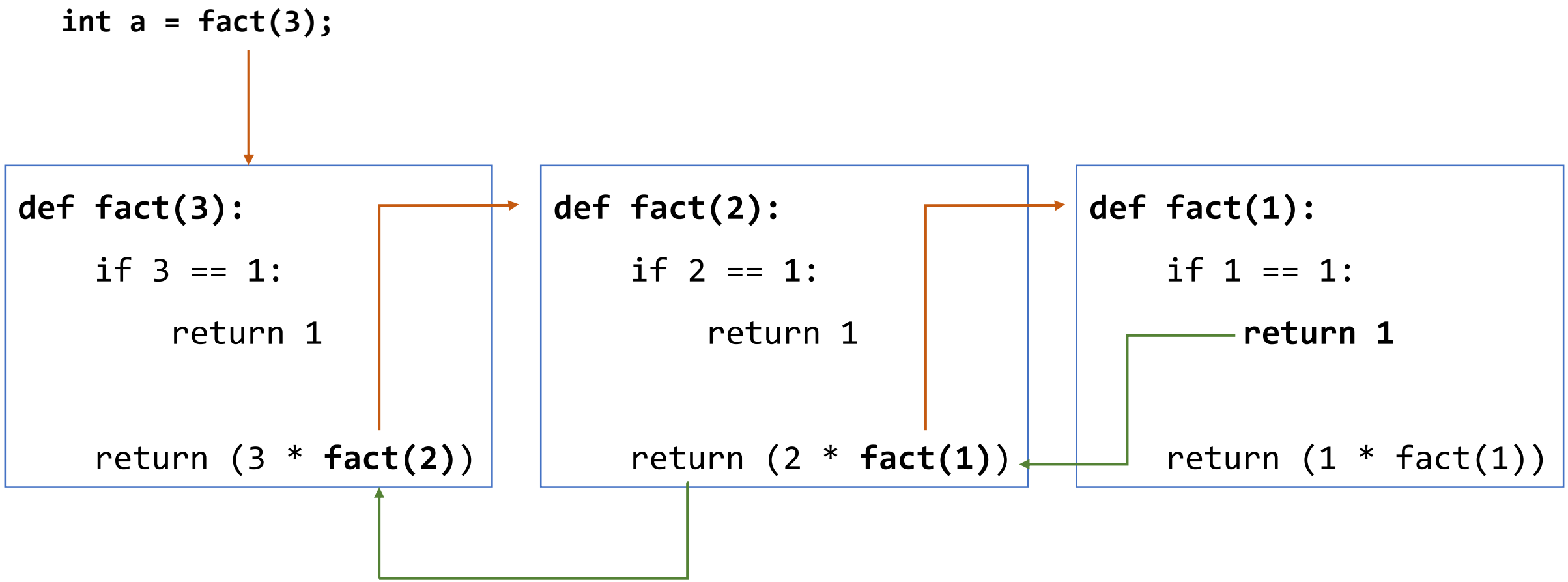
Recursion: Factorial



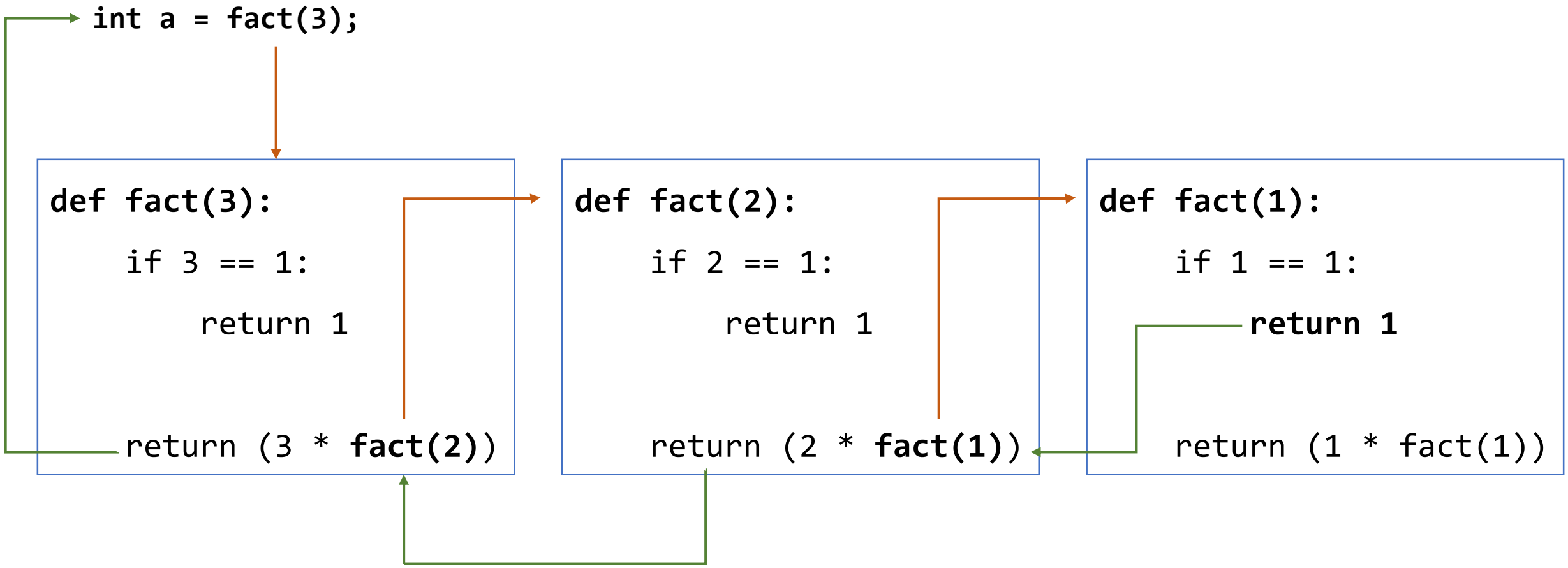
Recursion: Factorial



Recursion: Factorial



Recursion: Factorial



Recursion: Printing a LL in Reverse

Approach #1: Traverse a List and store individual item in an array. Print the array in reverse order. (Keep track of #nodes)

Approach #2: Traverse the List, store individual items in a Stack. Peek, Pop, Print.

Approach #3: Recursive approach!

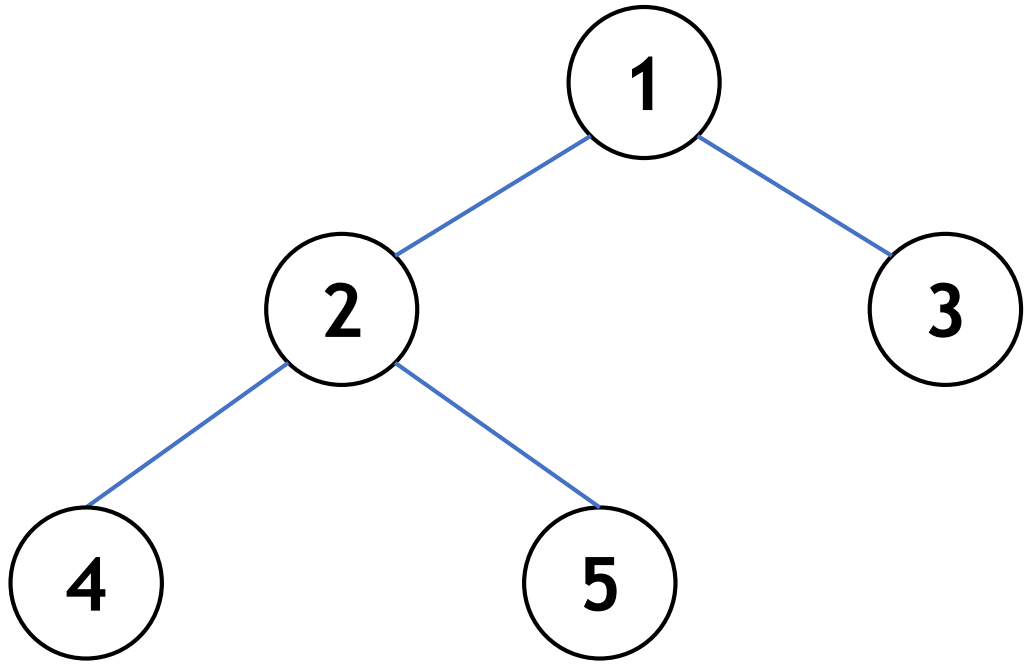
Traversal in a Binary Tree

Classification of traversal on basis of current node

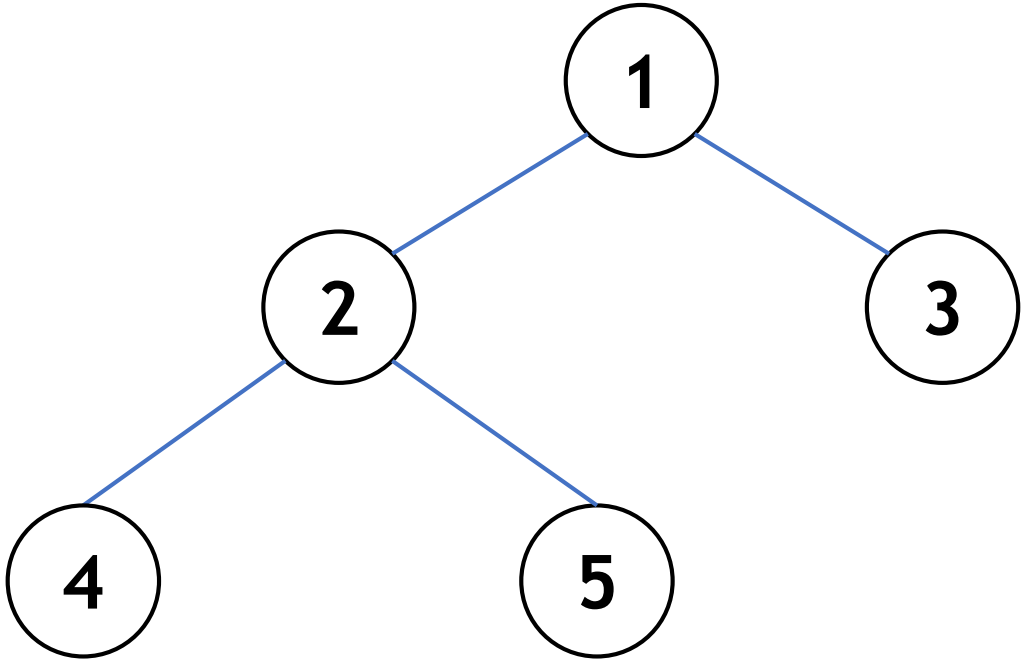
Preorder	DLR	1, 2, 4, 5, 3
Inorder	LDR	4, 2, 5, 1, 3
Postorder	LRD	4, 5, 2, 3, 1

Level order traversal

1, 2, 3, 4, 5



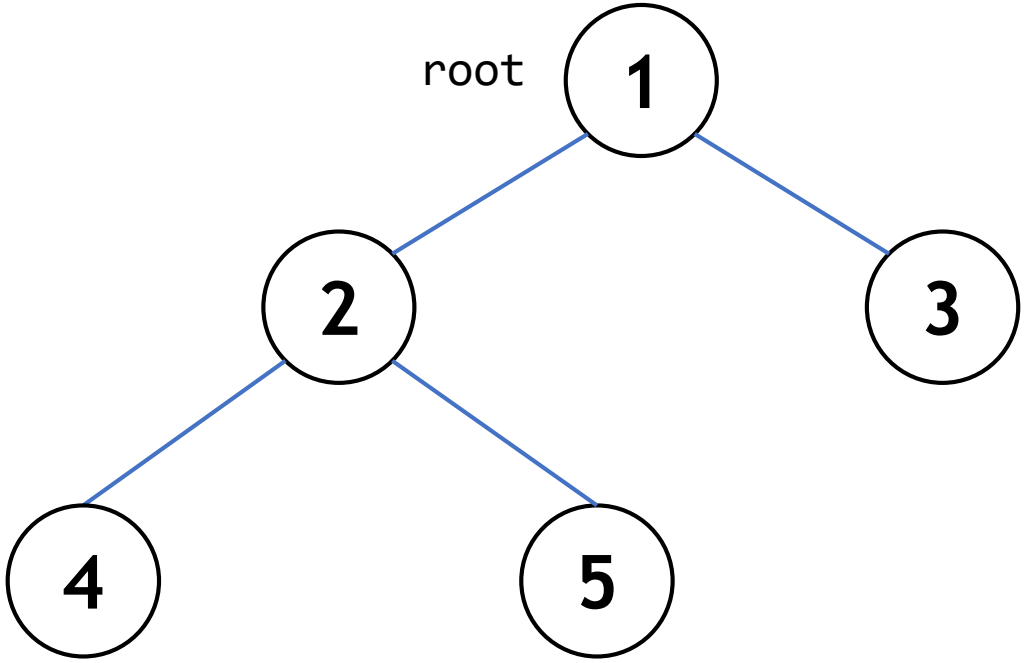
Traversal: Preorder




```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

1. **Print current node**
2. Process left subtree
3. Process right subtree

Traversal: Preorder

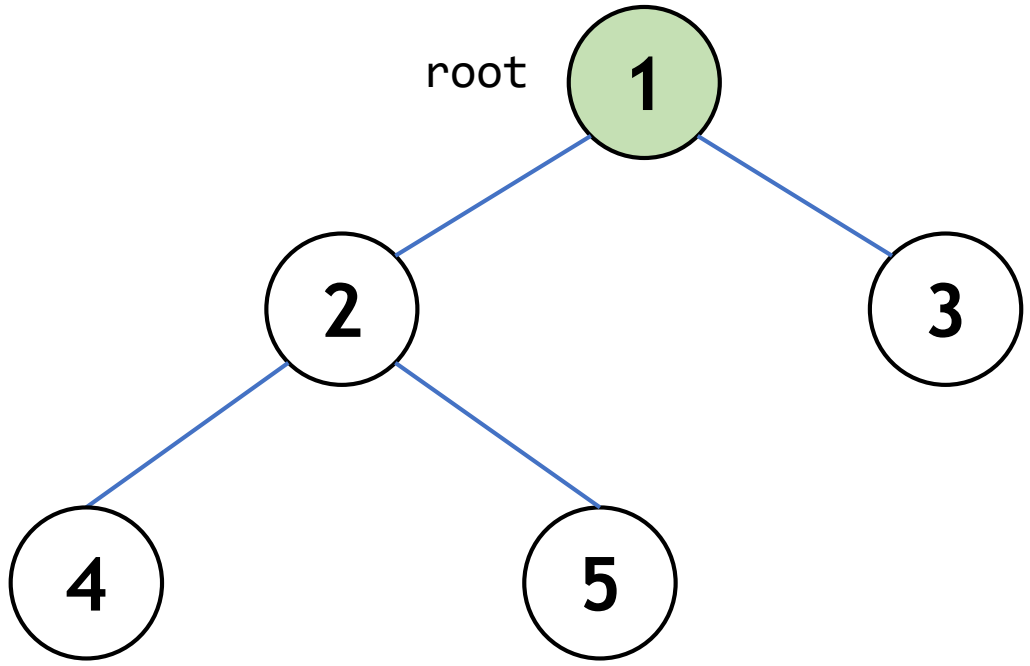


pre(1)

```
pre(root)
→ if root != NULL:
      print(root->data)
      pre(root->left)
      pre(root->right)
```

Output Screen

Traversal: Preorder



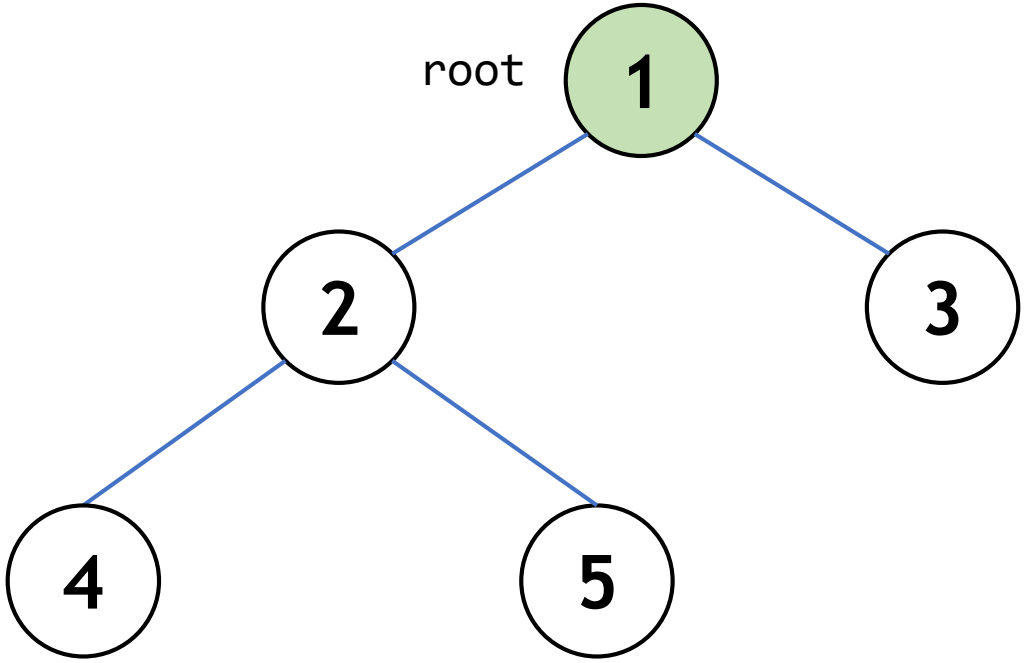
pre(1)

```
pre(root)
   if root != NULL:
      print(root->data)
      pre(root->left)
      pre(root->right)
```

Output Screen

1

Traversal: Preorder



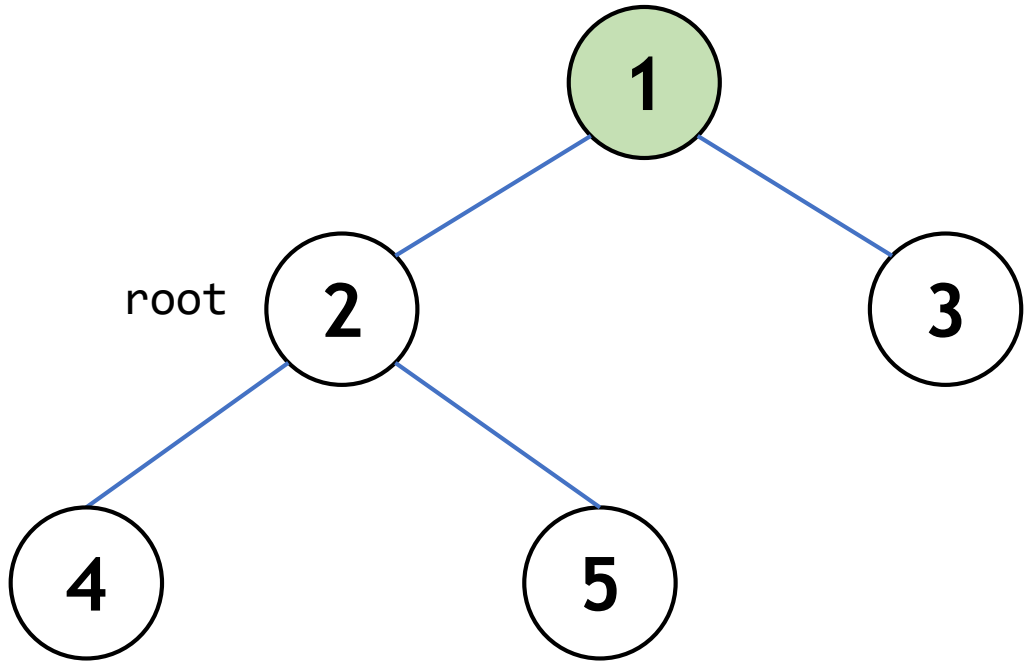
		pre(1)

```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1

Traversal: Preorder

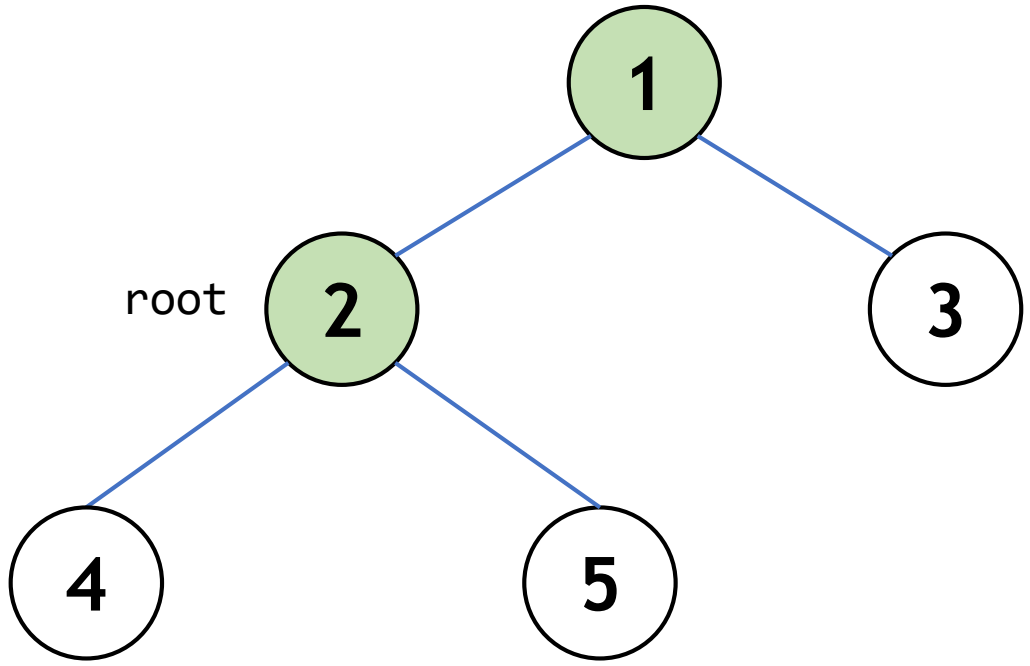



```
pre(root)
→ if root != NULL:
      print(root->data)
      pre(root->left)
      pre(root->right)
```

Output Screen

1

Traversal: Preorder

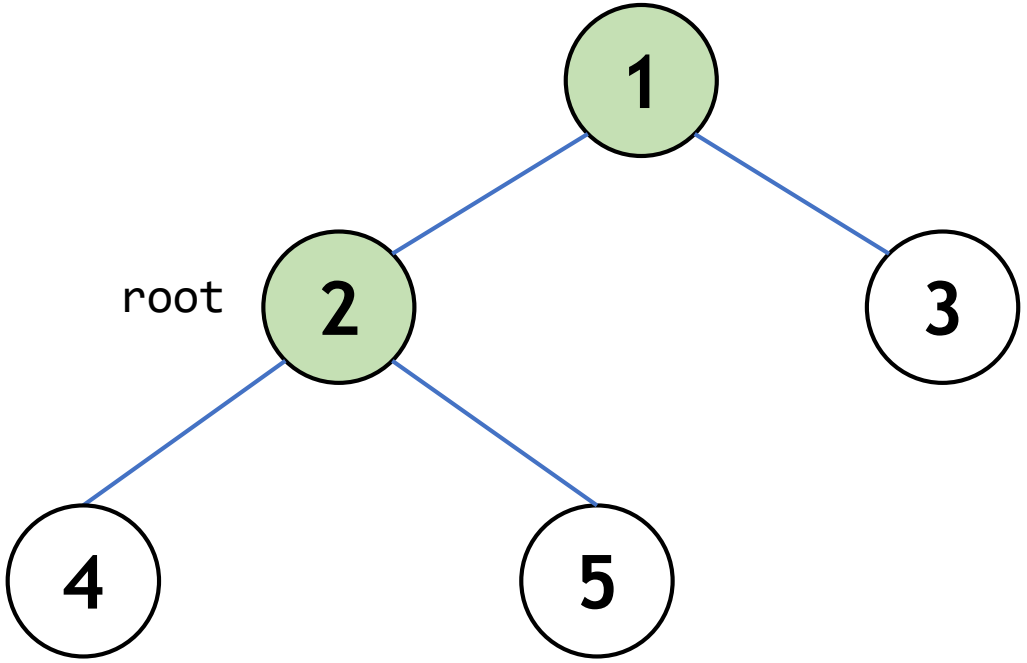



```
pre(root)
   if root != NULL:
      print(root->data)
      pre(root->left)
      pre(root->right)
```

Output Screen

1, 2

Traversal: Preorder



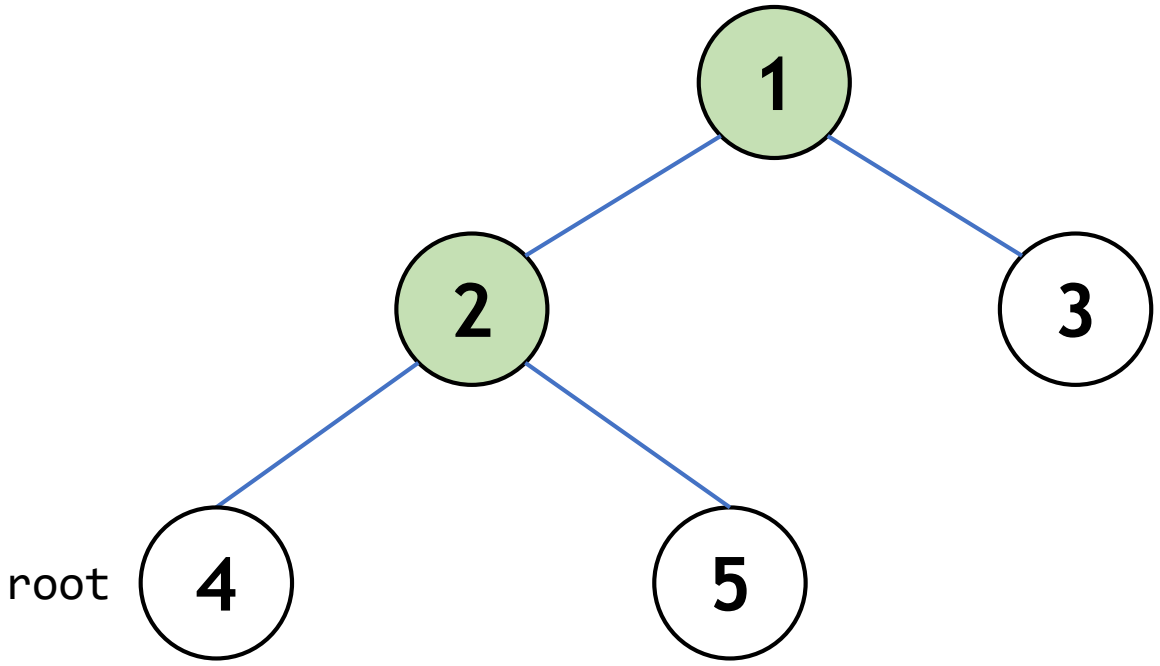
	pre(2)	pre(1)

```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2

Traversal: Preorder



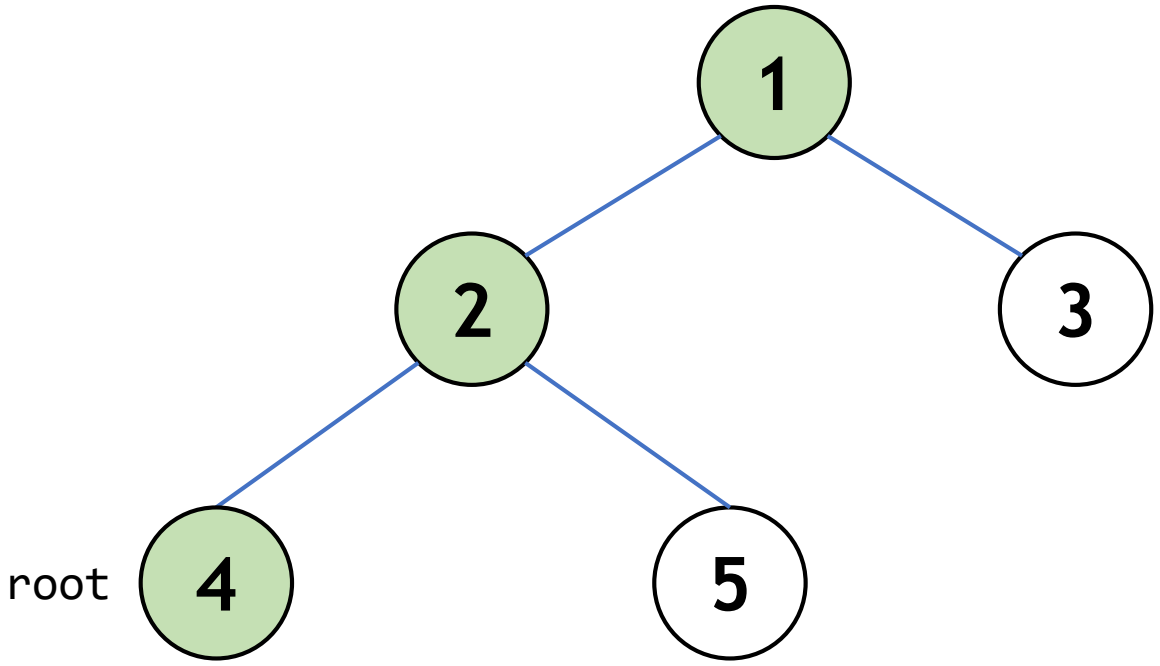
pre(4)		pre(2)	pre(1)

```
pre(root)
→ if root != NULL:
      print(root->data)
      pre(root->left)
      pre(root->right)
```

Output Screen

1, 2

Traversal: Preorder



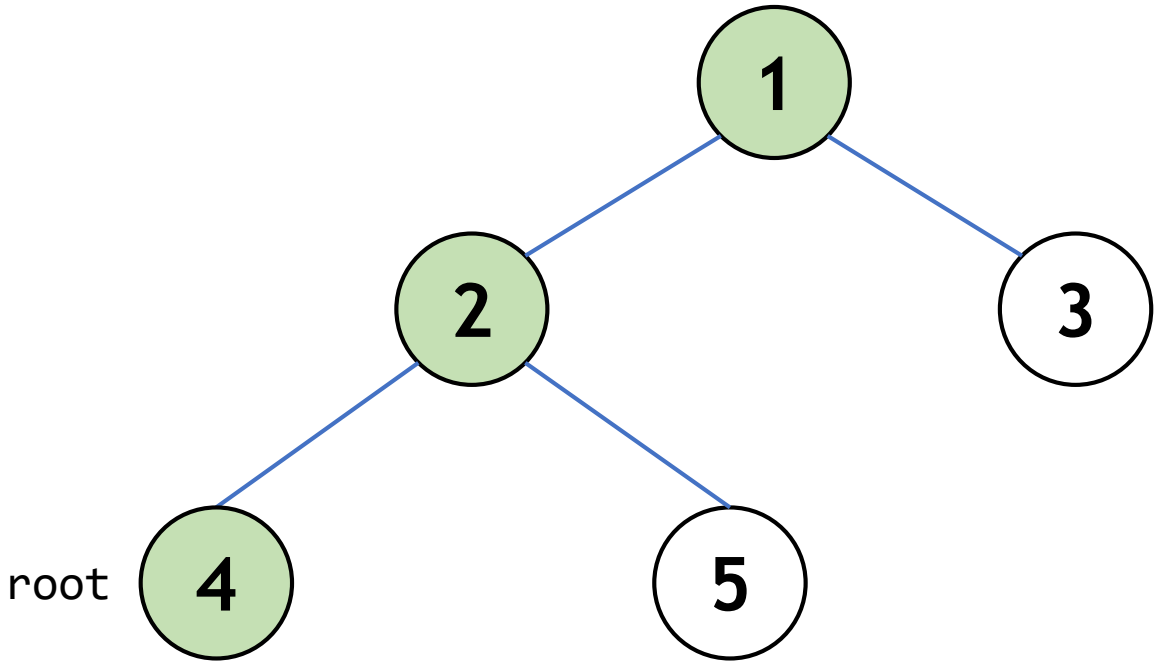
pre(4)		pre(2)	pre(1)

```
pre(root)
   if root != NULL:
      print(root->data)
      pre(root->left)
      pre(root->right)
```

Output Screen

1, 2, 4

Traversal: Preorder



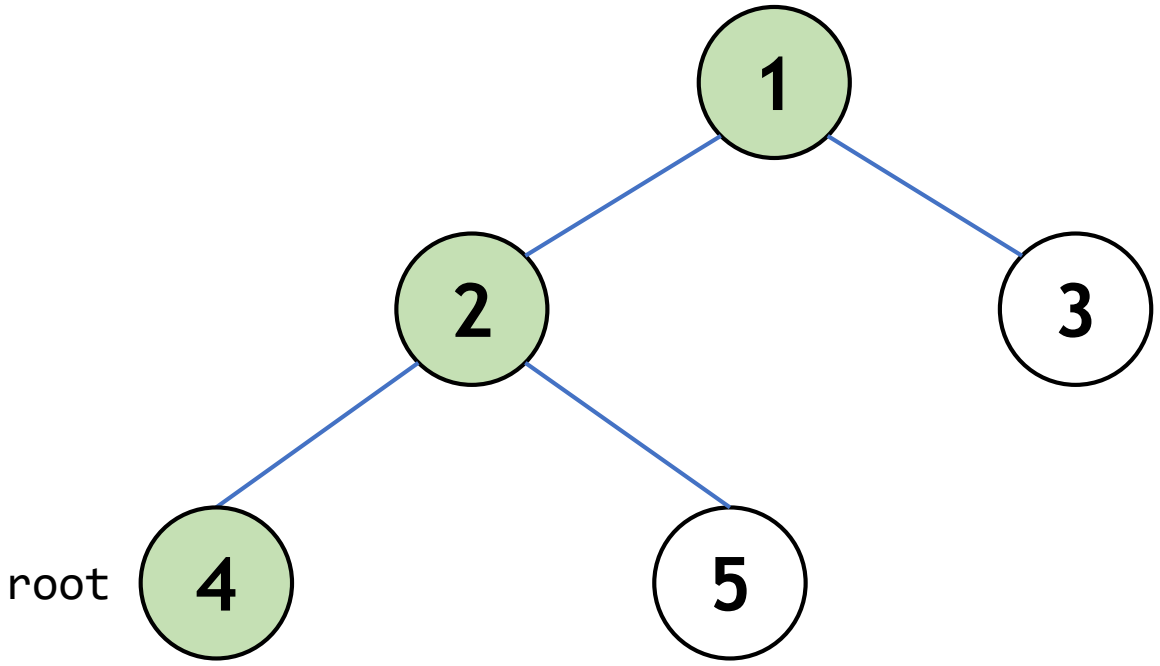
pre(∅)	pre(4)	pre(2)	pre(1)

```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4

Traversal: Preorder



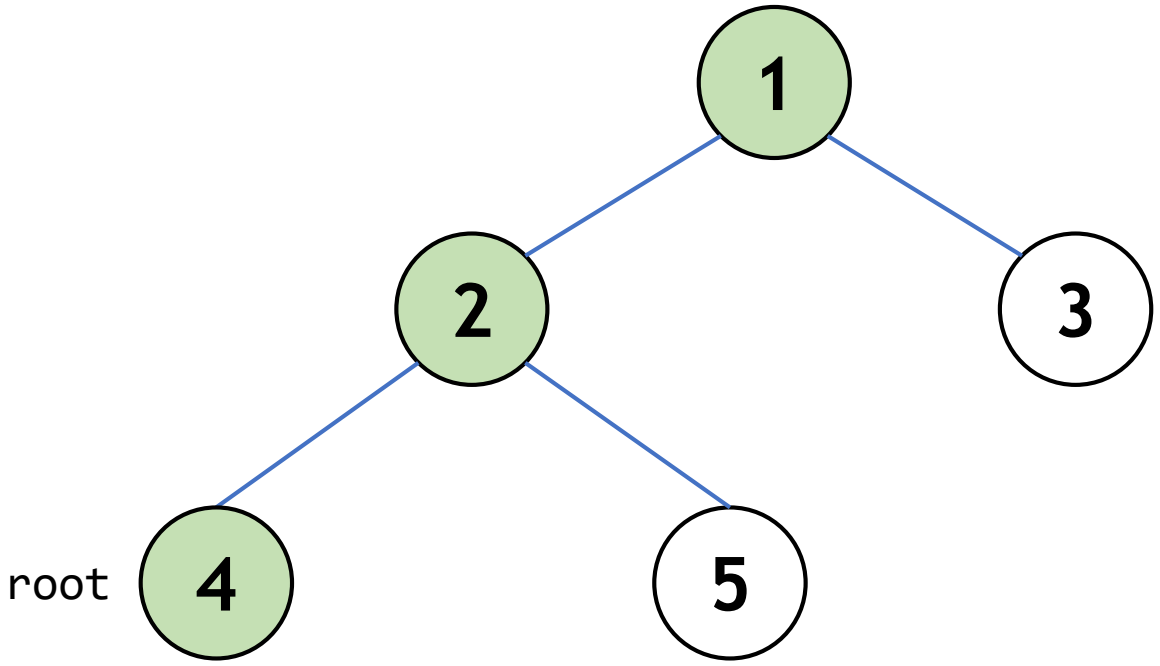
pre(∅)	pre(4)	pre(2)	pre(1)

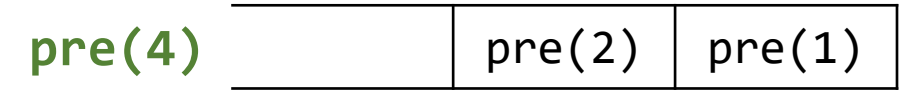
```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4

Traversal: Preorder



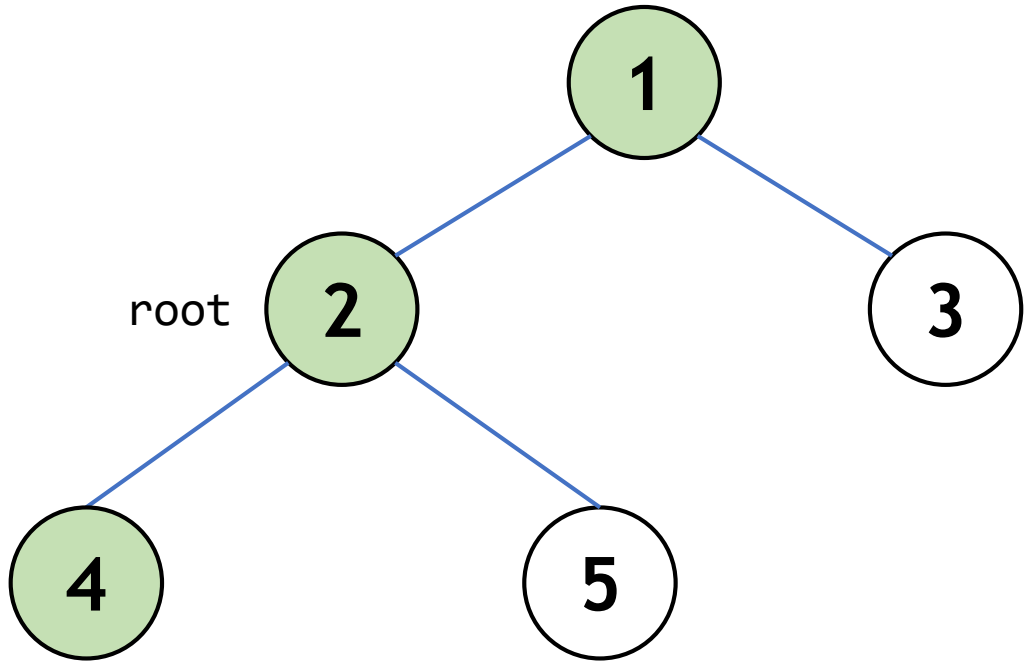



```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4

Traversal: Preorder

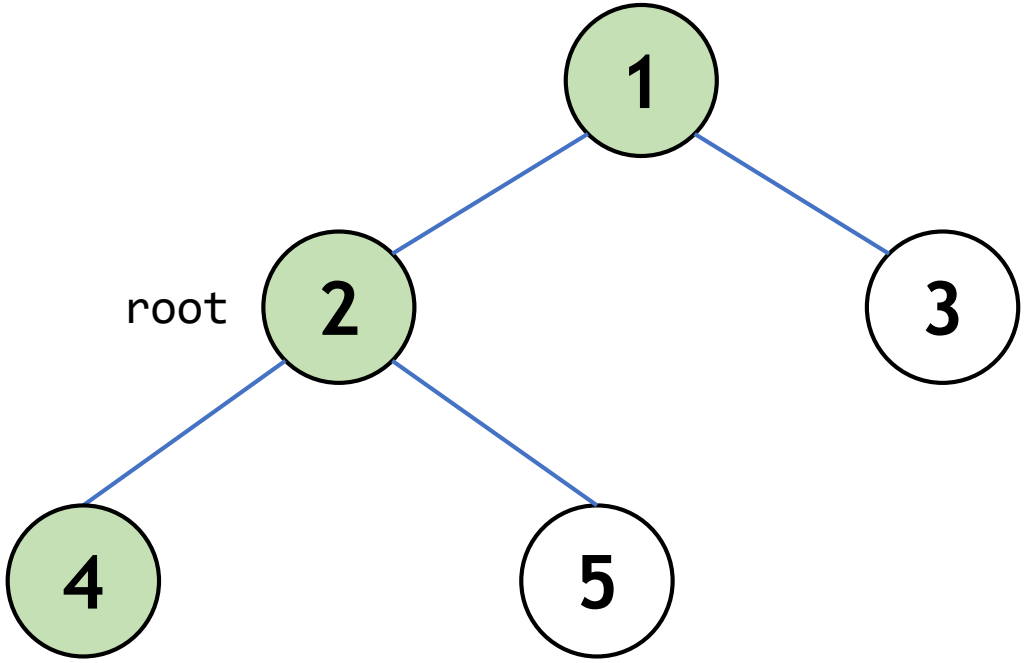



```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4

Traversal: Preorder



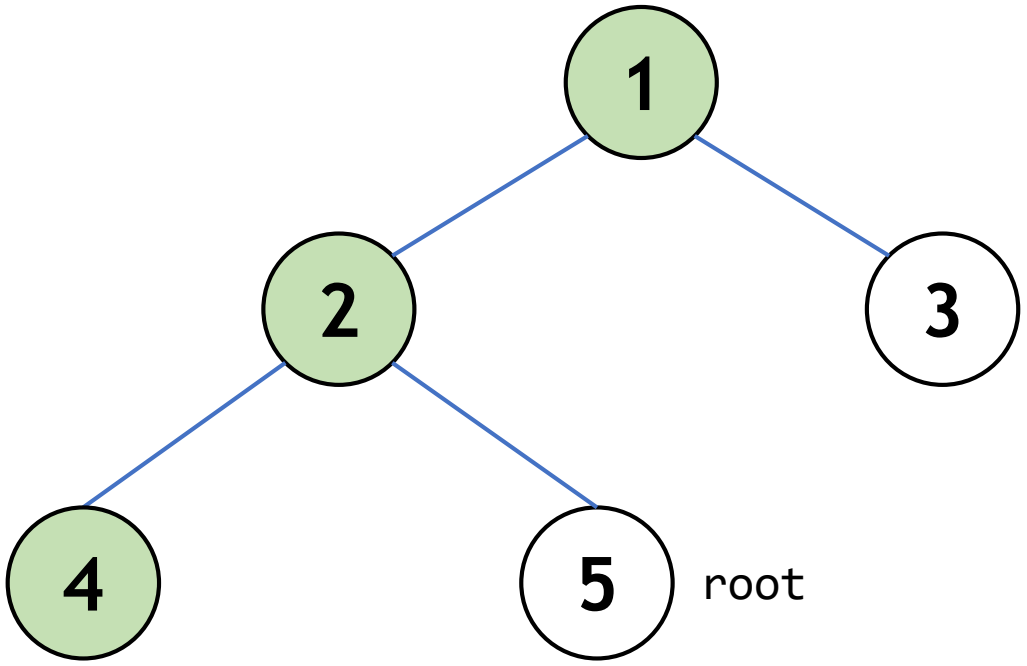
	pre(2)	pre(1)

```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4

Traversal: Preorder



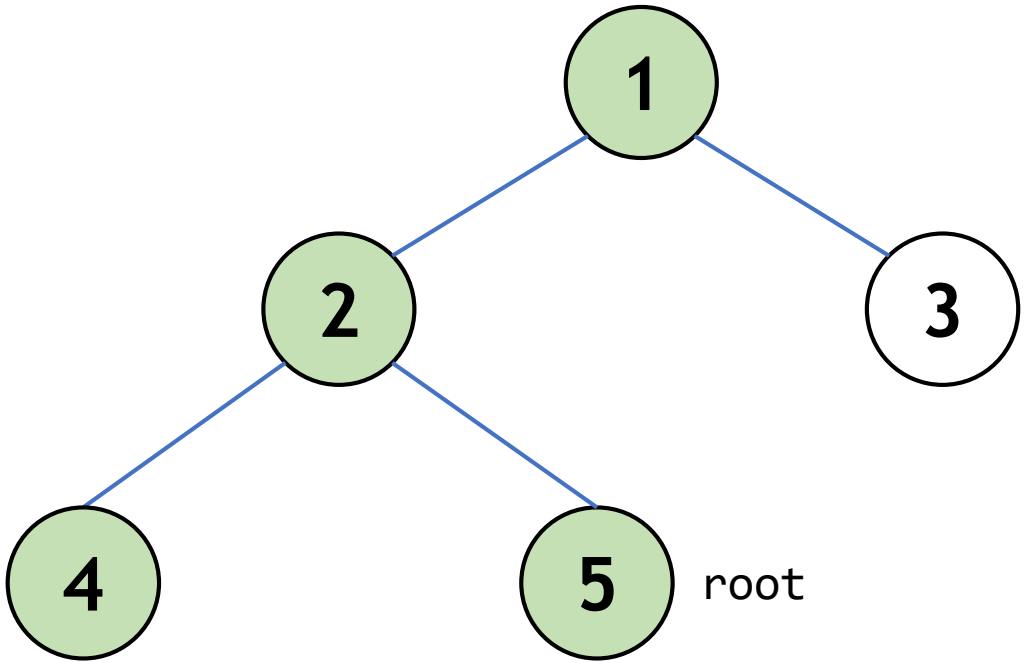
pre(5)		pre(2)	pre(1)

```
pre(root)
→ if root != NULL:
      print(root->data)
      pre(root->left)
      pre(root->right)
```

Output Screen

1, 2, 4

Traversal: Preorder



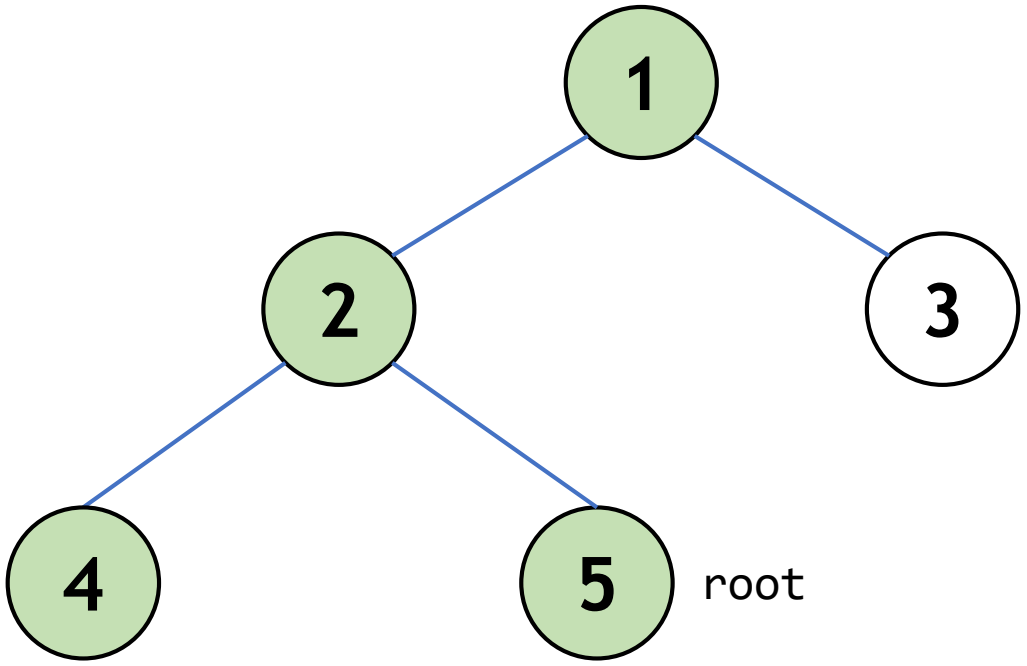
pre(5)		pre(2)	pre(1)

```
pre(root)
   if root != NULL:
      print(root->data)
      pre(root->left)
      pre(root->right)
```

Output Screen

1, 2, 4, 5

Traversal: Preorder



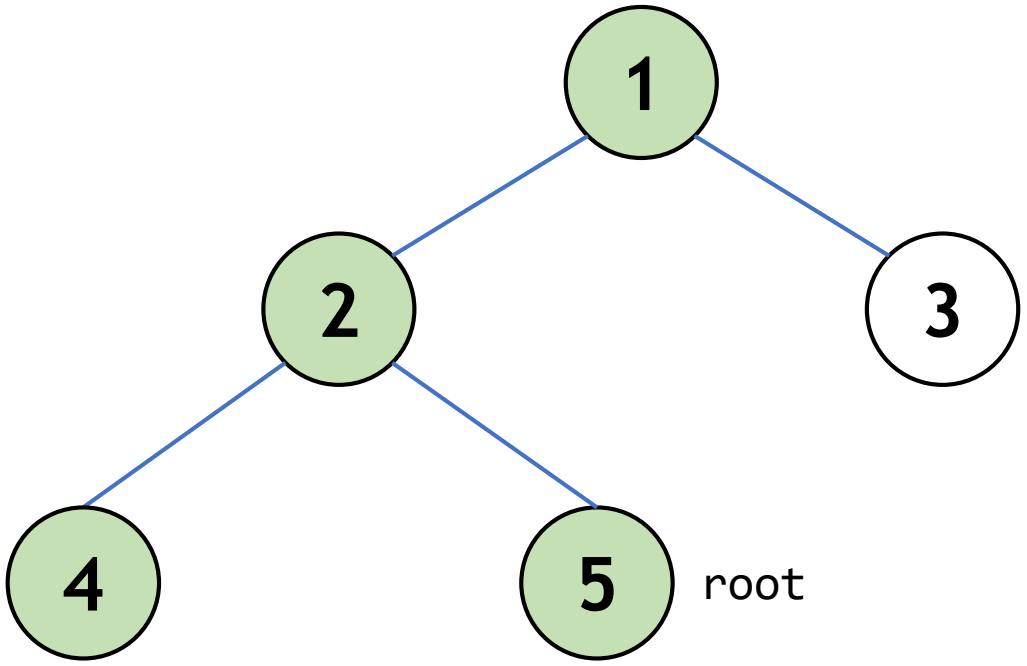
pre(∅)	pre(5)	pre(2)	pre(1)

```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4, 5

Traversal: Preorder



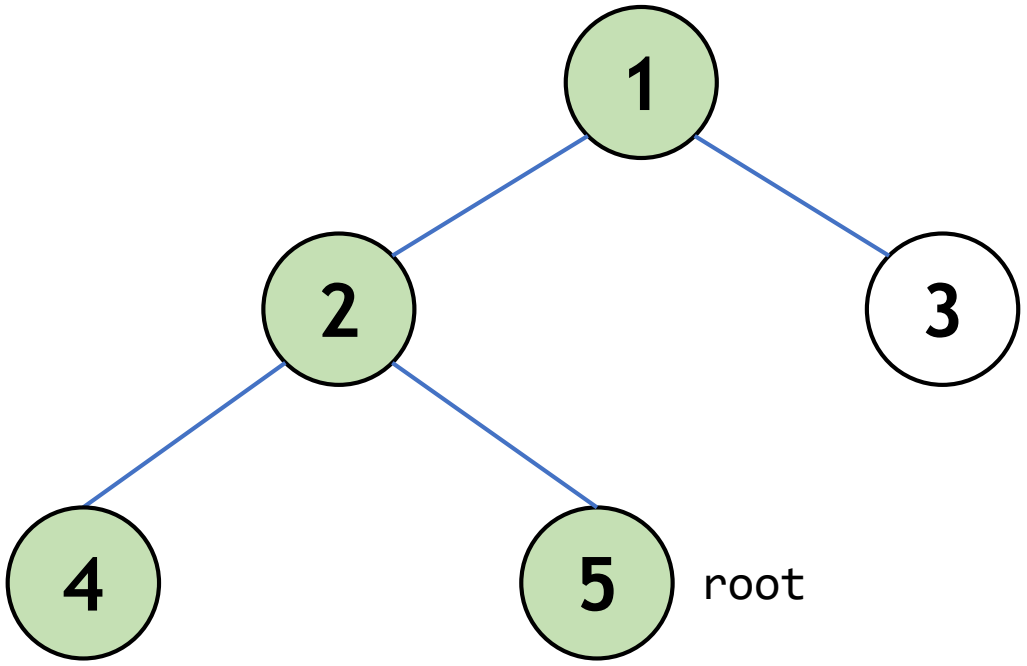
pre(∅)	pre(5)	pre(2)	pre(1)

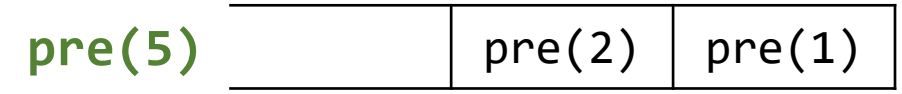
```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4, 5

Traversal: Preorder



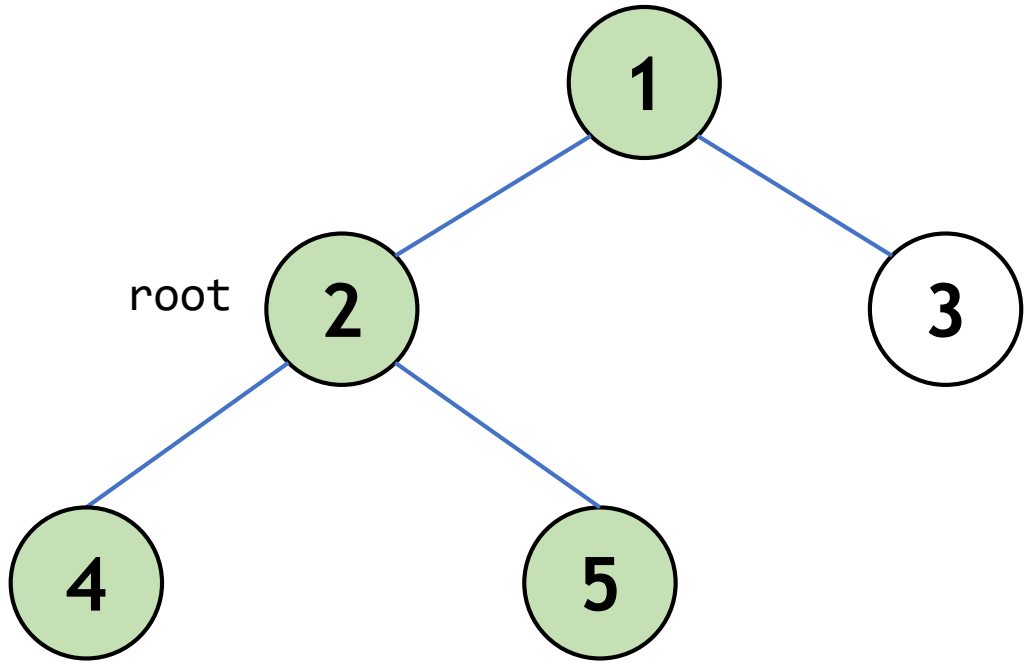



```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4, 5

Traversal: Preorder

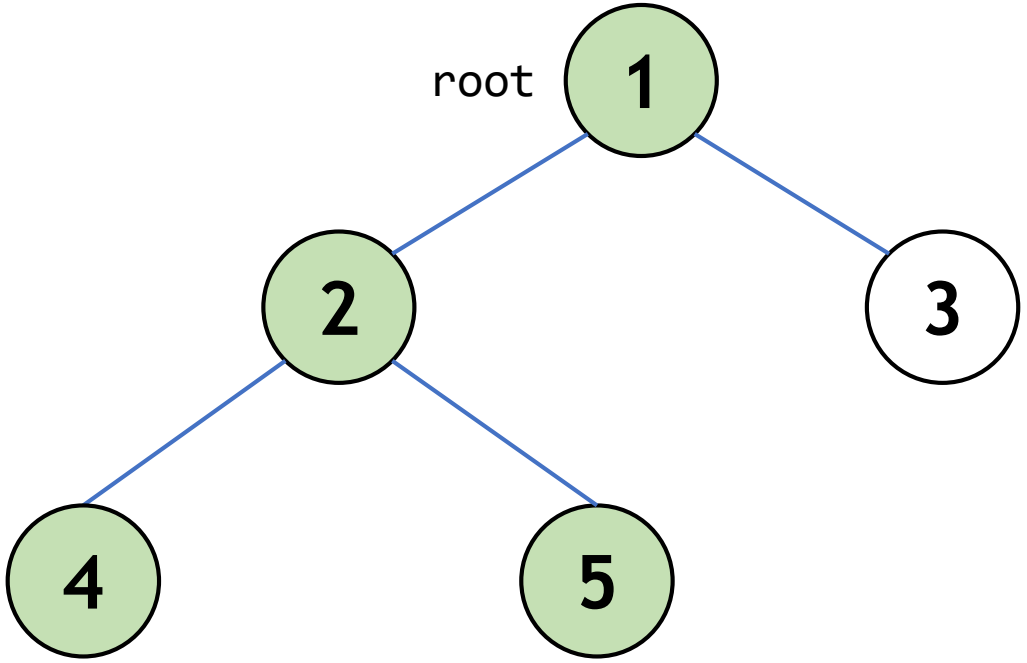



```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4, 5

Traversal: Preorder



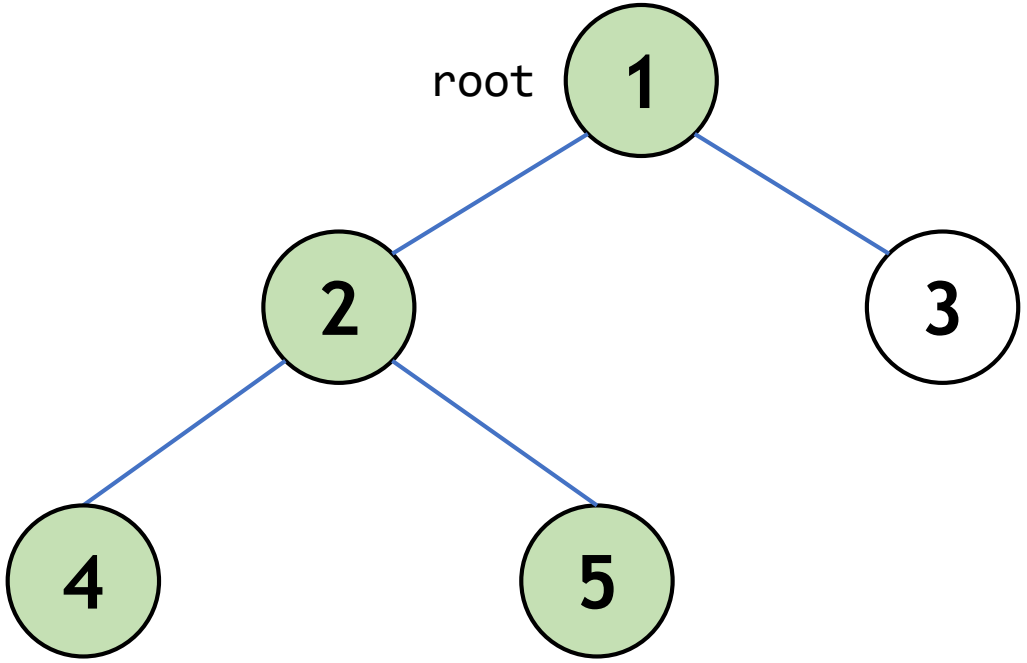
pre(1)

```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4, 5

Traversal: Preorder



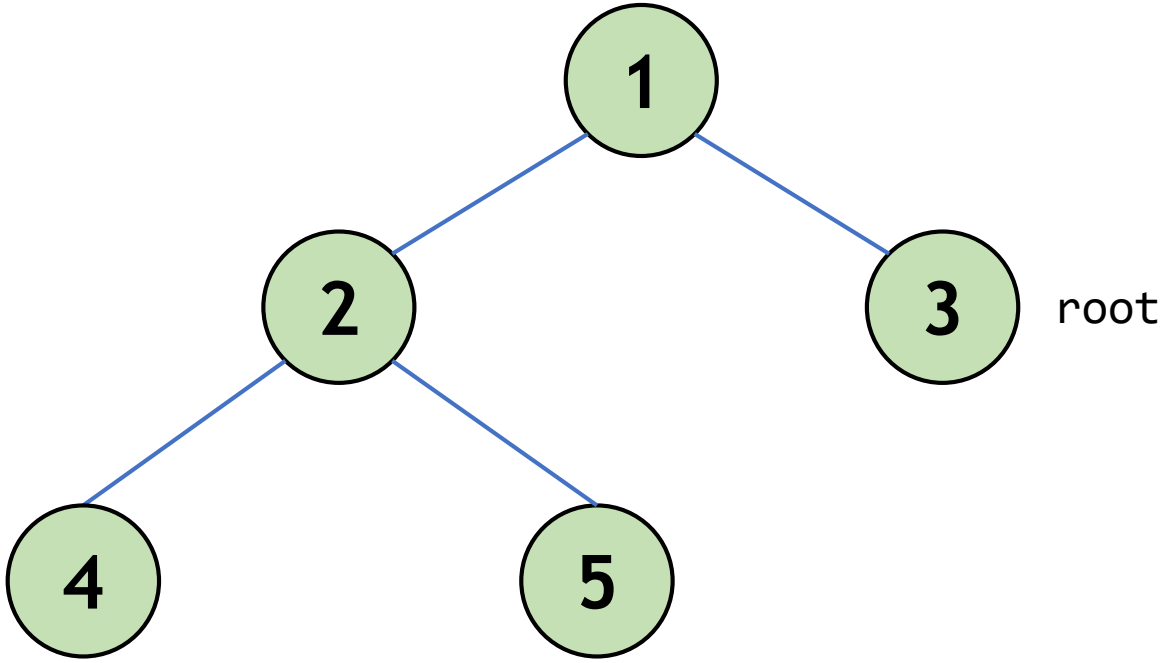
		pre(1)

```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4, 5

Traversal: Preorder

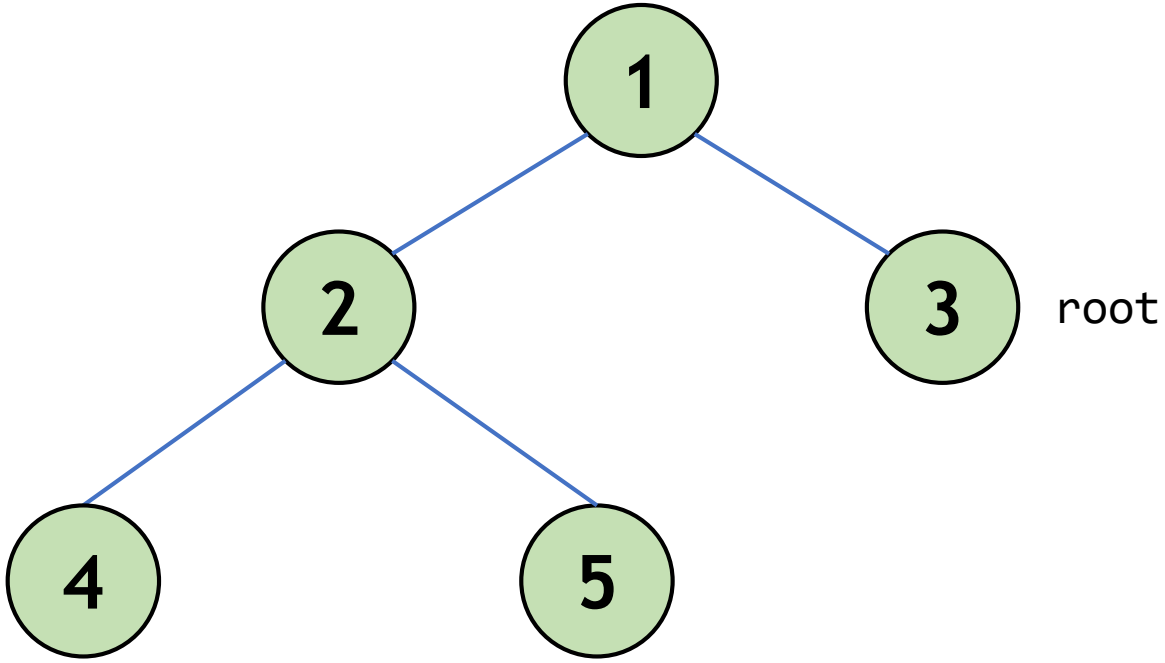



```
pre(root)
   if root != NULL:
      print(root->data)
      pre(root->left)
      pre(root->right)
```

Output Screen

1, 2, 4, 5, 3

Traversal: Preorder



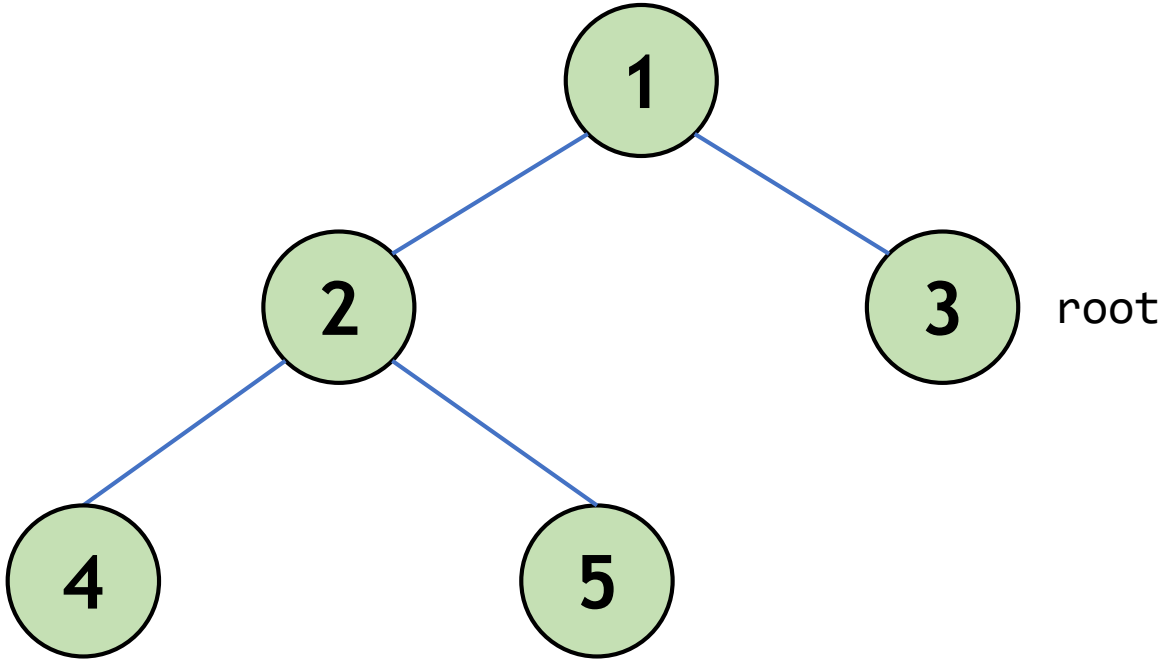
pre(∅)		pre(3)	pre(1)

```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4, 5, 3

Traversal: Preorder



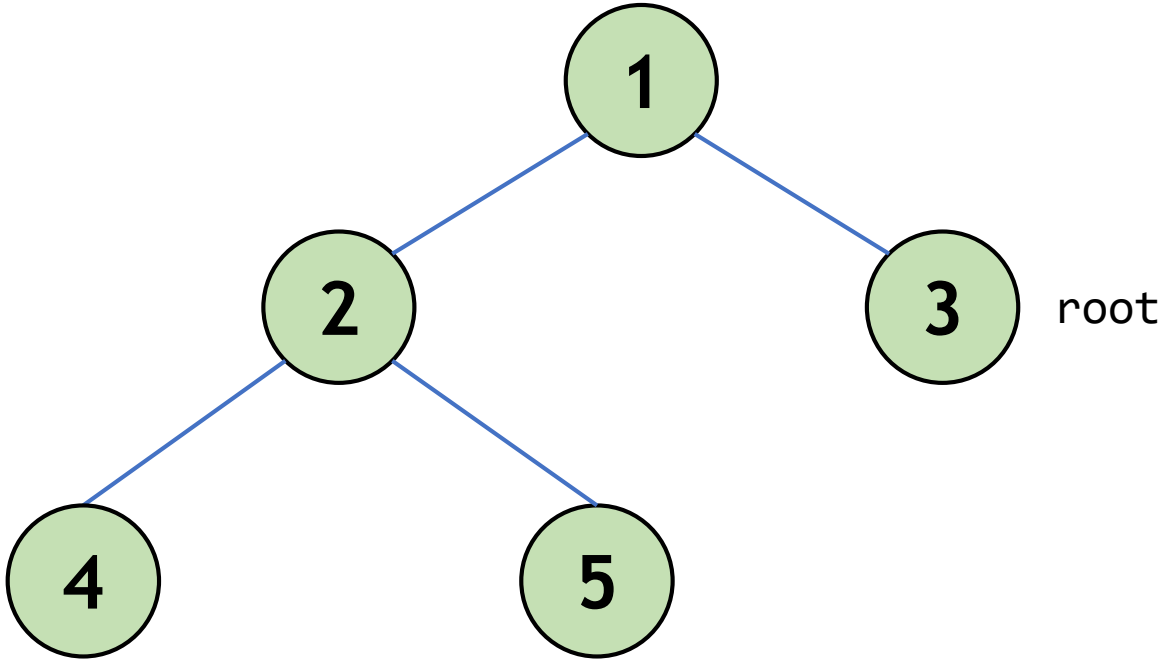
pre(a)		pre(3)	pre(1)

```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4, 5, 3

Traversal: Preorder

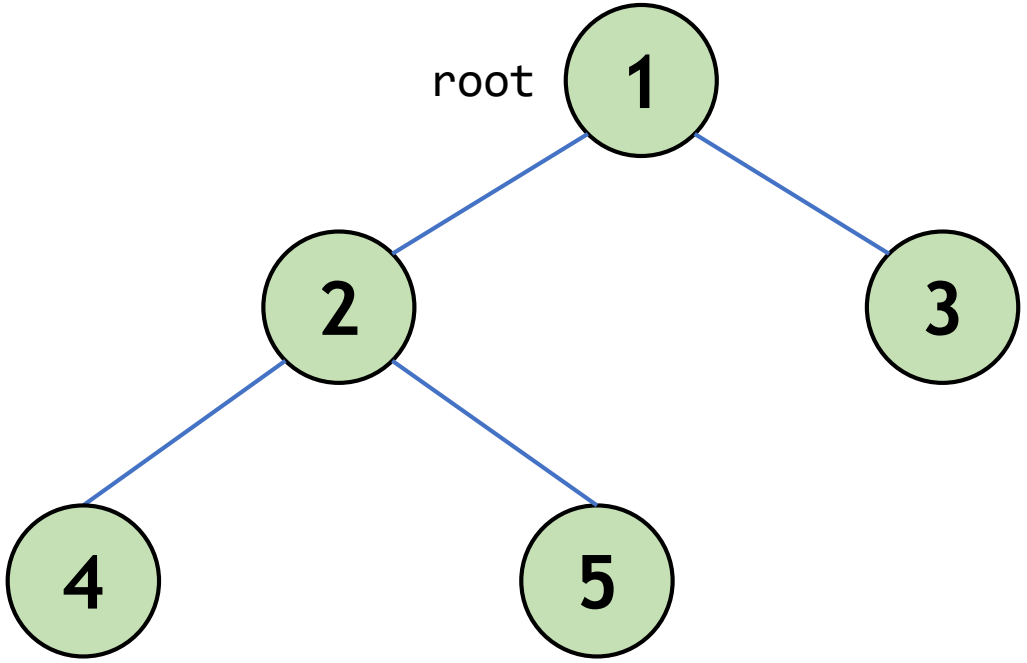



```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4, 5, 3

Traversal: Preorder



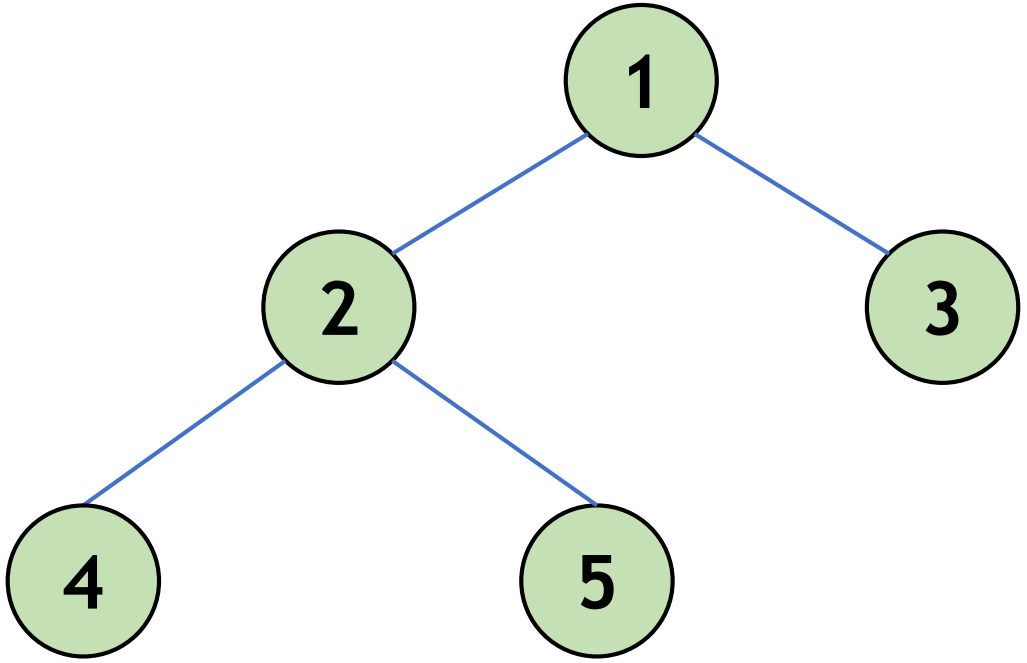
pre(1)

```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4, 5, 3

Traversal: Preorder

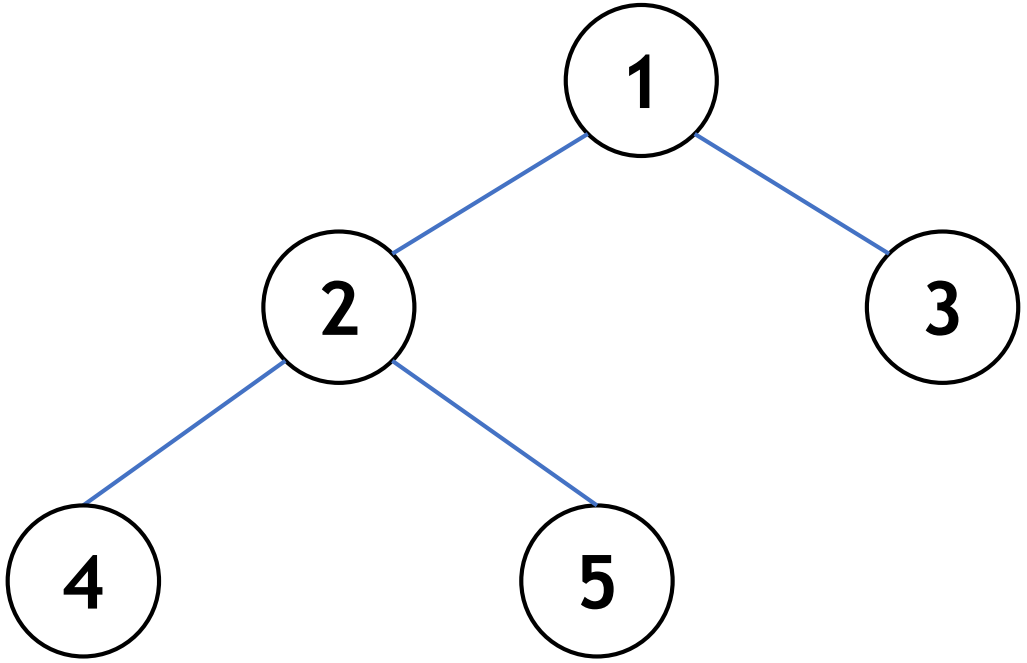



```
pre(root)
   if root != NULL:
        print(root->data)
        pre(root->left)
        pre(root->right)
```

Output Screen

1, 2, 4, 5, 3

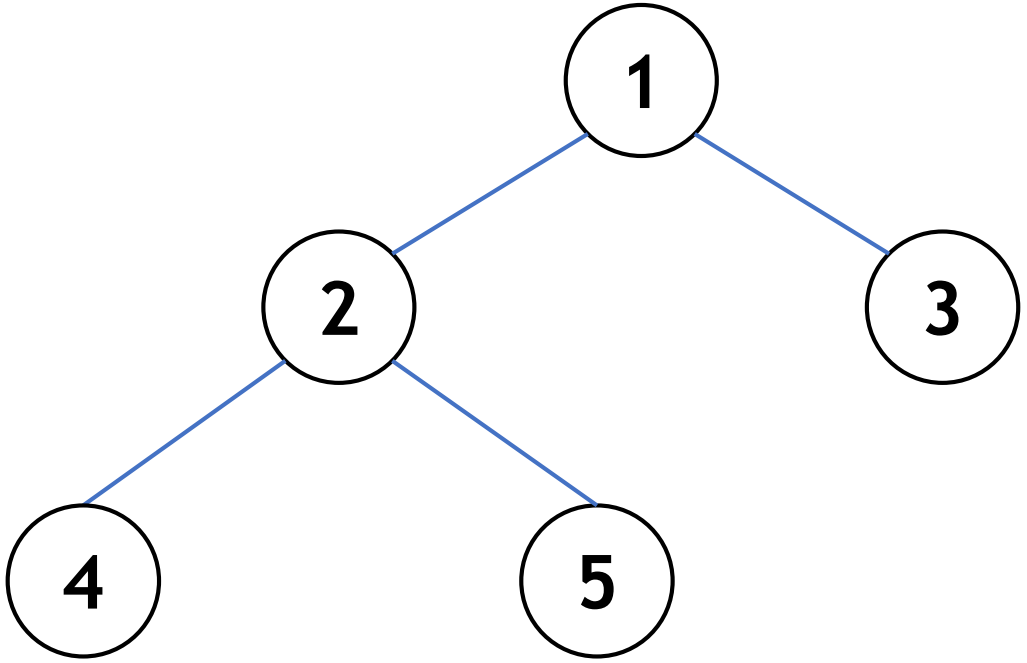
Traversal: Inorder




```
ino(root)
  if root != NULL:
      ino(root->left)
      print(root->data)
      ino(root->right)
```

1. Process left subtree
2. **Print current node**
3. Process right subtree

Traversal: Inorder

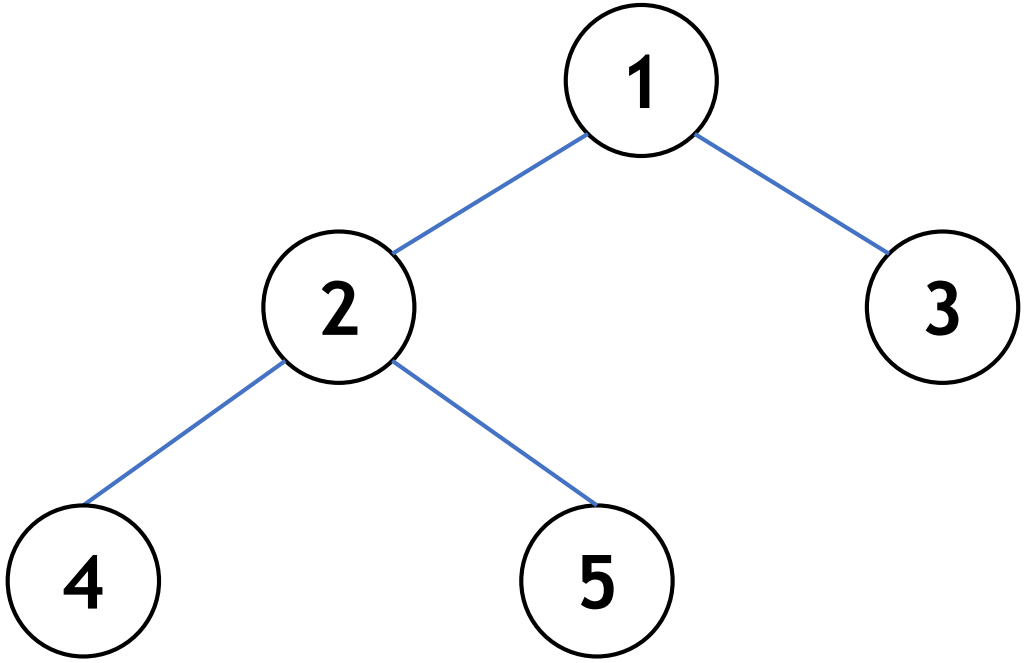



```
ino(root)
  if root != NULL:
      ino(root->left)
      print(root->data)
      ino(root->right)
```

Output Screen

4, 2, 5, 1, 3

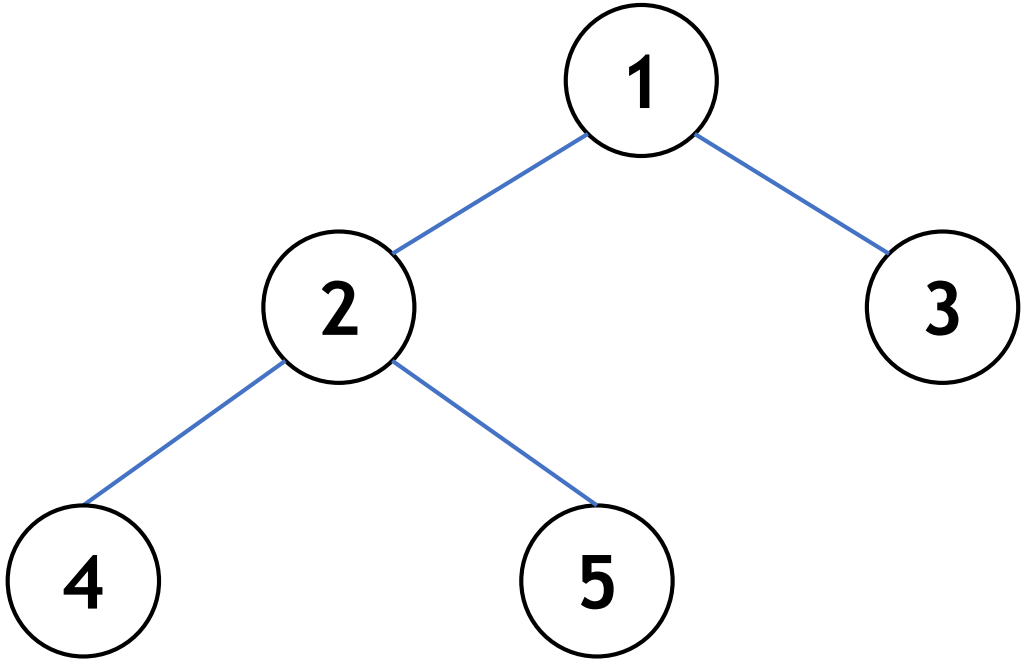
Traversal: Postorder




```
post(root)
   if root != NULL:
        post(root->left)
        post(root->right)
        print(root->data)
```

1. Process left subtree
2. Process right subtree
3. **Print current node**

Traversal: Postorder




```
post(root)
   if root != NULL:
        post(root->left)
        post(root->right)
        print(root->data)
```

Output Screen

4, 5, 2, 3, 1

Traversal: Visualization

OpenDSA (Virginia Tech)

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/BinaryTreeTraversal.html

Binary Tree: Pop Quiz

Given a **Preorder Traversal** of a BT (not perfect), can you identify the **root** of the BT? **[Ex: 1, 2, 4, 5, 3]**

Yes! 1.

Given a **Postorder Traversal** of a BT (not perfect), can you identify the **root** of the BT? **[Ex: 1, 2, 4, 5, 3]**

Yes! 3.

Exercise

Binary Tree

Exercise: Silver

Implement:

`Tree::deleteTree(Node *node)`

3-4 lines of code!

How do you delete a node?

```
Deleting tree

  Deleting node:4
  Deleting node:5
  Deleting node:2
  Deleting node:6
  Deleting node:7
  Deleting node:3
  Deleting node:1
```

Exercise: Gold

```
Sum of all the nodes in tree is:
28
```

Implement:

```
int Tree::sumNodes(Node *node)
```

2 lines of code!

What would be the base case? How to terminate recursion?

What will you return from the function?