NANYANG TECHNOLOGICAL UNIVERSITY

Data Structures & Algorithms in Python: Binary Trees

Dr. Owen Noel Newton Fernando

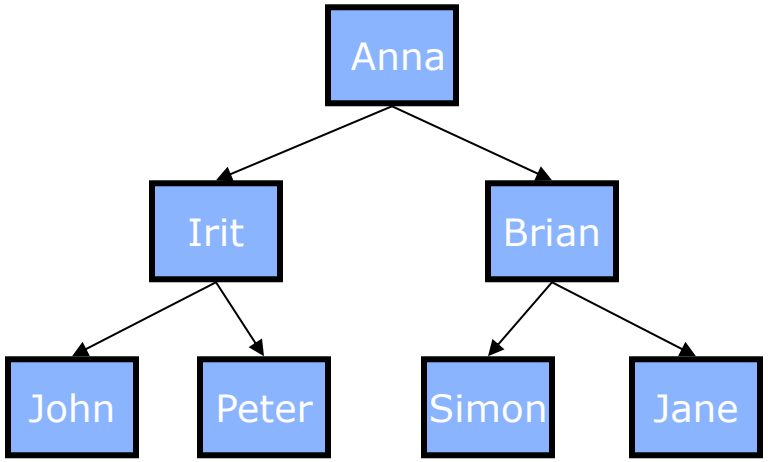
College of Computing and Data Science

Review: Linear data structures

- Here are some of the data structures we have studied so far:
  - Singly-linked lists and doubly-linked lists
  - Stacks, queues, and priority queues
- These all have the property that their elements can be adequately constructed and displayed in a straight line
- Binary trees are one of the simplest nonlinear data structures

What is a tree?

- A tree data structure is a widely used hierarchical data structure in computer science that resembles an inverted tree with a root node at the top and branches extending downward.

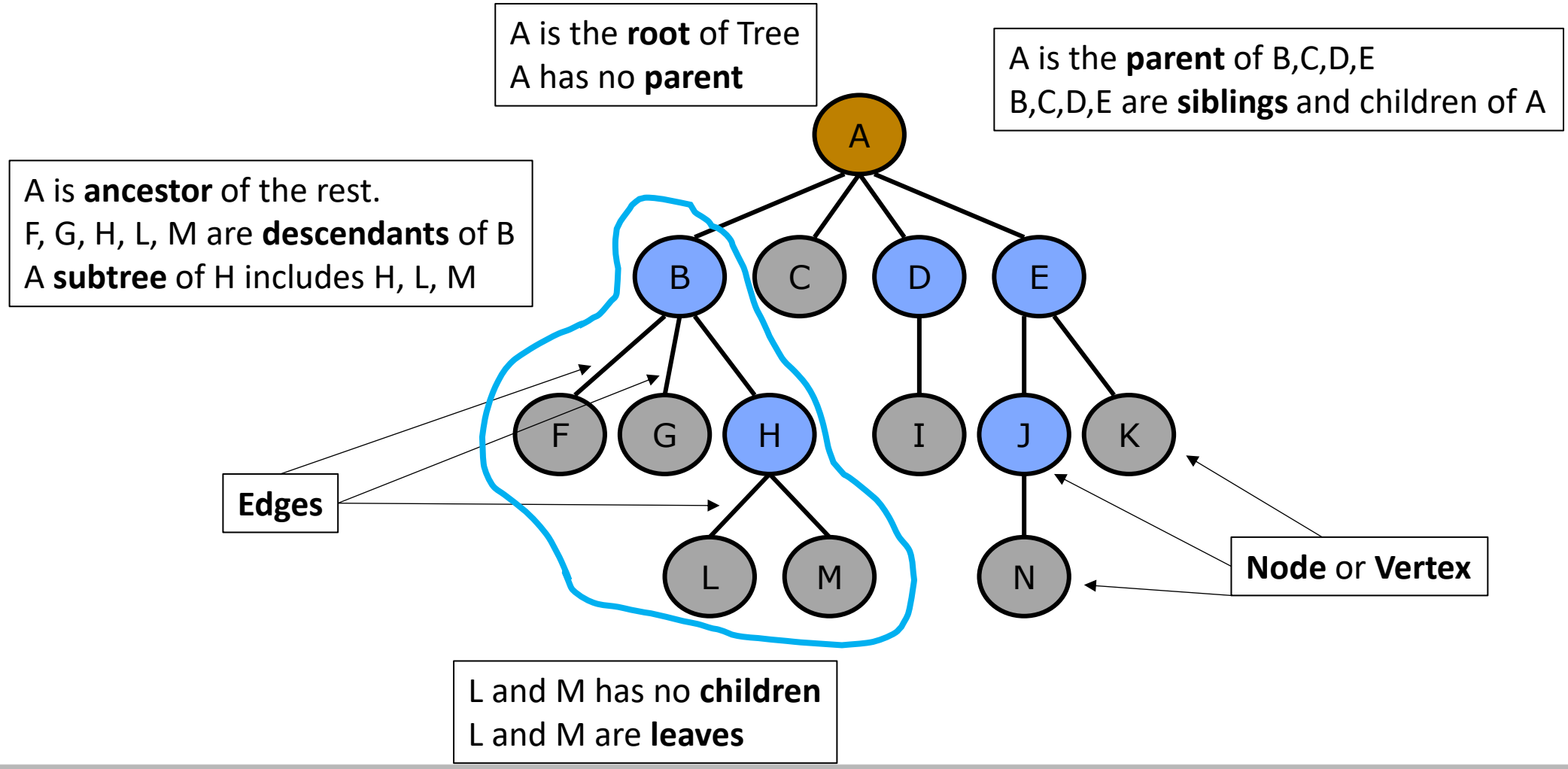


Tree

We define a ***tree*** *T* as a set of ***nodes*** storing elements such that the nodes have a ***parent-child*** relationship that satisfies the following properties:

- If *T* is nonempty, it has a special node, called the ***root*** of *T*, that has no parent.
- Each node *v* of *T* different from the root has a unique ***parent*** node *w*; every node with parent *w* is a ***child*** of *w*.

Terminology in Tree Structure



Terminology in Tree Structure

Ancestor

- **Definition:** An ancestor of a node in a tree is any node that exists on the path from that node to the root. This includes the parent node, the grandparent node, and all the way up to the root node.
- **Example:** In a tree, if **node A** is the **root**, **node B** is a **child** of **node A**, and **node C** is a **child** of **node B**, then **node A** is an **ancestor** of both **node B** and **node C**. **Node B** is also an **ancestor** of **node C**.

Terminology in Tree Structure

Descendant

- **Definition:** A descendant of a node in a tree is any node that exists on the path from that node down to the leaves. This includes the **children**, **grandchildren**, and all nodes further down the tree.
- **Example:** In a tree, if **node A** is the **root**, **node B** is a **child** of **node A**, and **node C** is a **child** of **node B,** then **node C** is a **descendant** of both **node B** and **node A**. **Node B** is also a **descendant** of **node A**.

Terminology in Tree Structure

Ancestor

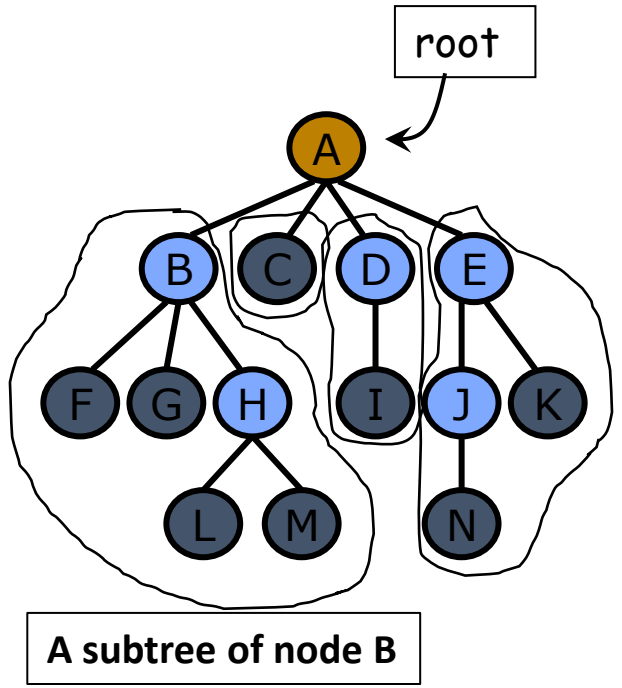
- **Definition:** An ancestor of a node in a tree is any node that exists on the path from that node to the root. This includes the parent node, the grandparent node, and all the way up to the root node.
- **Example:** In a tree, if **node A** is the **root**, **node B** is a **child** of **node A**, and **node C** is a **child** of **node B**, then **node A** is an **ancestor** of both **node B** and **node C**. **Node B** is also an **ancestor** of **node C**.

Descendant

- **Definition:** A descendant of a node in a tree is any node that exists on the path from that node down to the leaves. This includes the **children**, **grandchildren**, and all nodes further down the tree.
- **Example:** In a tree, if **node A** is the **root**, **node B** is a **child** of **node A**, and **node C** is a **child** of **node B,** then **node C** is a **descendant** of both **node B** and **node A**. **Node B** is also a **descendant** of **node A**.

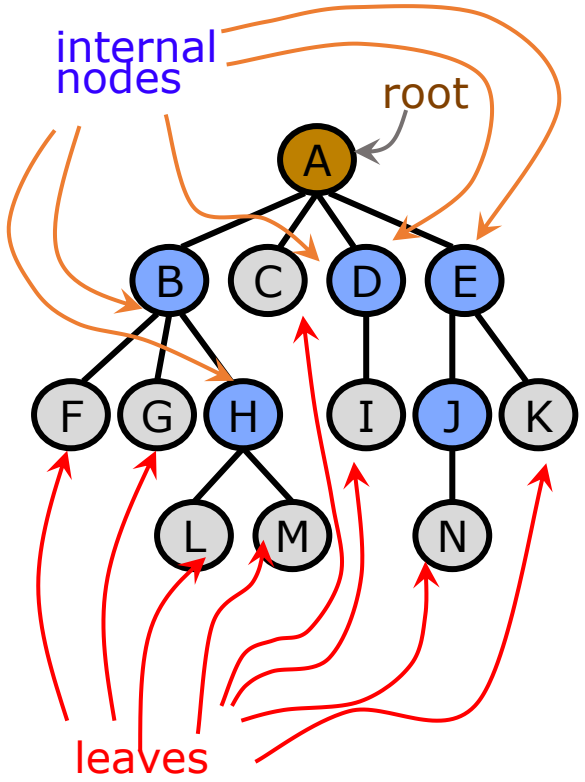
Terminology in Tree Structure

- Similar to family tree concept
- One special node: **root**
- Each node can has many children
  A has four children: B, C, D, E
- Each node (except the root) has a parent node
- **A is the parent of B, C, D, E**
- Other children of your parent are your siblings
- **B, C, D and E are siblings**
- **Subtree**: Any node in the tree together with all of its descendants for a subtree..



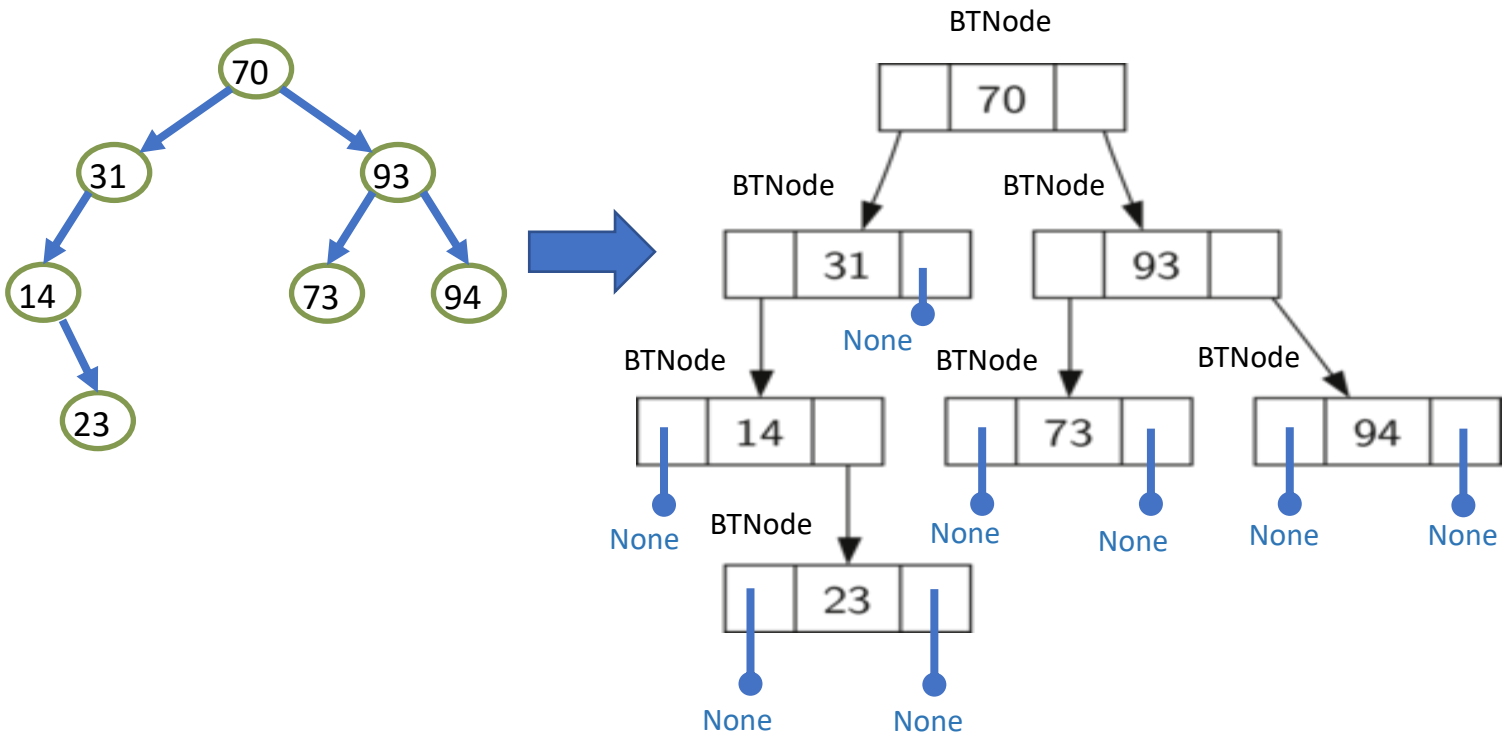
Terminology in Tree Structure

- A tree is composed of nodes
- Types of nodes
  - Root: only one in a tree, has no parent.
  - Internal node(non-leaf):
    Nodes with children are called internal nodes
  - Leaf (External Node):
    nodes without children are called leaves



Binary Trees

Definition: Each node has at most two children.

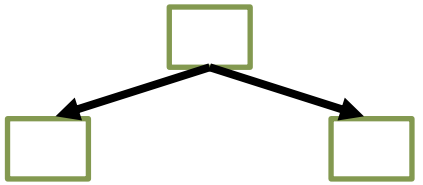


Binary Tree

- The operations for a binary tree :
  - Insert a node
  - Delete a node
  - Search a node
  - Traversal (display all nodes in a certain order)
    - Inorder
    - Preorder
    - Postorder
    - Level-by-level
    - How do we systematically travel each node once in a tree?

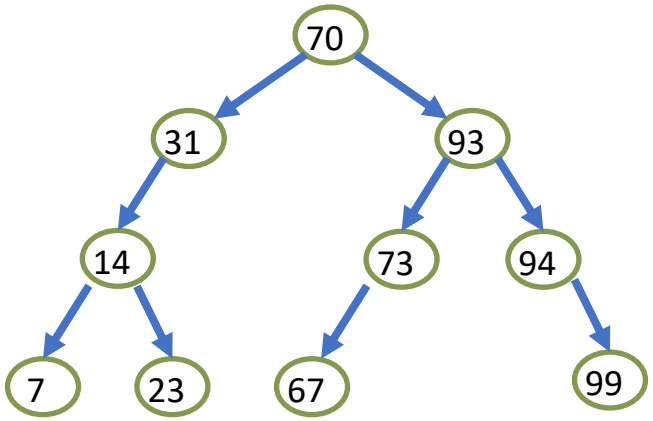
Binary Tree Node Class

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```



Binary Tree Traversal

- Assume that we have a binary tree (constructing the tree will be discussed later)
- How do you systematically visit every nodes once only?
  - Print all contents of a tree
  - Search a node in a tree
  - Find the size of a tree
  - Etc.
- Types of Traversal
  - Depth-first Traversal
  - Breath-first Traversal



Traversal Approaches on A Binary Tree

- **Depth-First Traversal**: From the root of a tree, it explores as far as possible. Then it will do the backtracking. There are three traversal orders:
  - Pre-order
  - In-order
  - Post-order

- **Breadth-First Traversal**: From the root of a tree, it explores each node in level by level.

Depth-first search

Breadth-first search

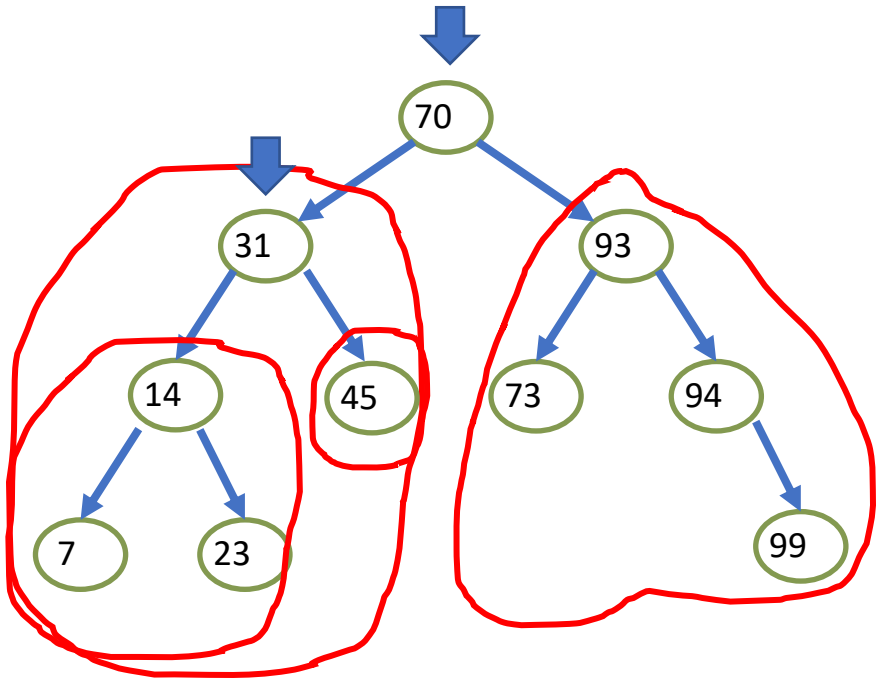
Binary Tree Traversal – Depth-first Traversals

Traversal Problem:

- Visit root + right subtree + left subtree
- Each subtree repeat the same procedure

  visit root + right subtree + left subtree
- Until reach the leave

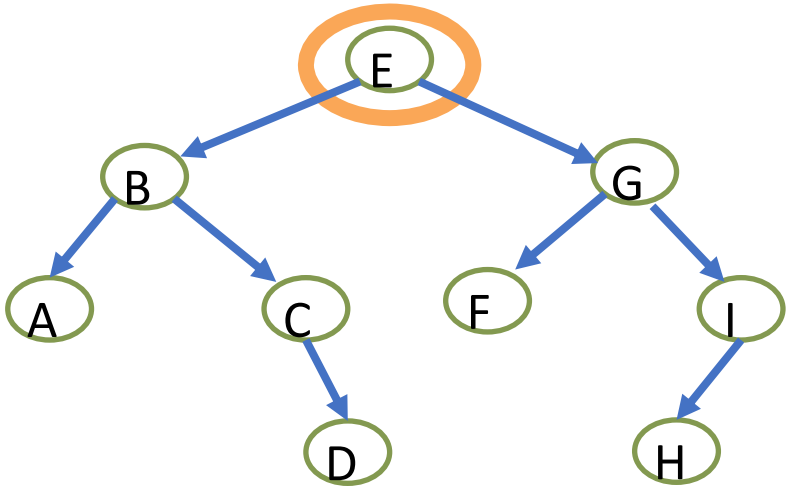
  visit the root (leaf only)
- It is a recursive problem



Pre-order Depth First Traversal

- Pre-order
  - **Process the current node's data**
  - **Visit the left child subtree**
  - **Visit the right child subtree**

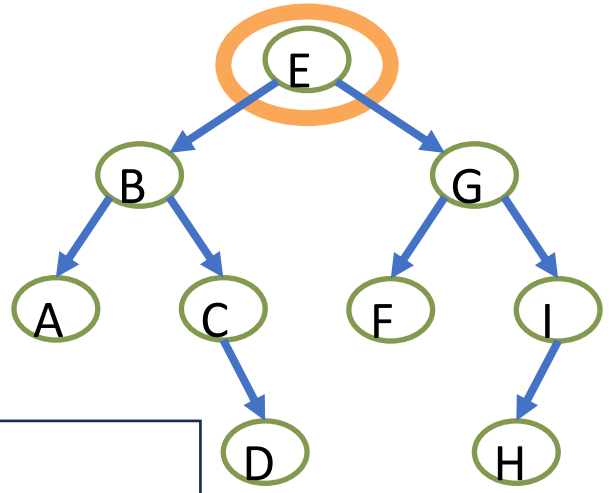
```
TreeTraversal(Node N):
    Visit N;
    If (N has left child)
        TreeTraversal(LeftChild);
    If (N has right child)
        TreeTraversal(RightChild);
    Return; // return to parent
```



Pre-order Depth First Traversal

Output:

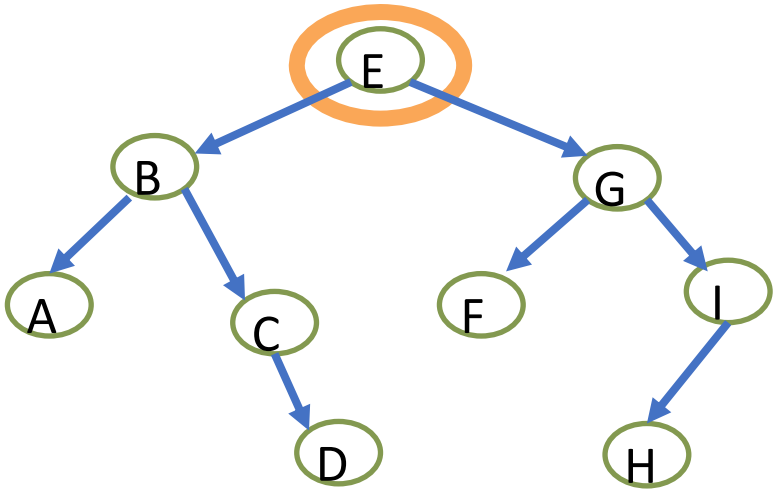
```
E B A C D G F I H
```




```
def pre_order_traversal(self, node):
        if node:
            print(node.data, end=" ")
            self.pre_order_traversal(node.left)
            self.pre_order_traversal(node.right)
```

In-order Depth First Traversal

- Pre-order
  - Process the current node's data
  - Visit the left child subtree
  - Visit the right child subtree
- **In-order**
  - **Visit the left child subtree**
  - **Process the current node's data**
  - **Visit the right child subtree**
- Post-order

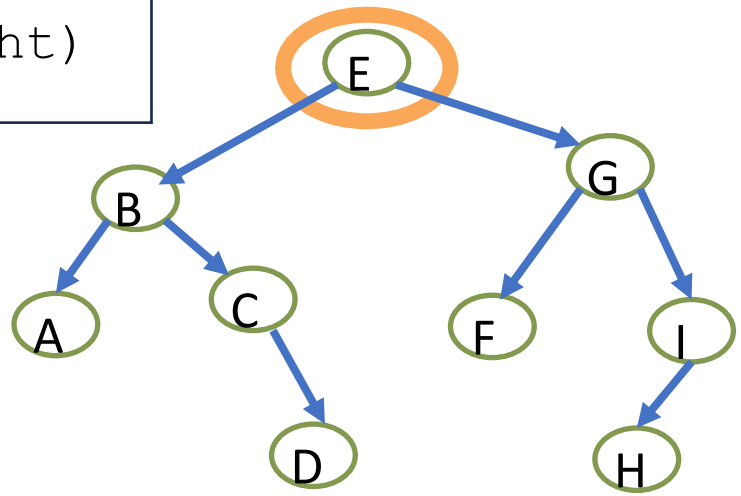


In-order Depth First Traversal

```
def in_order_traversal(self, node):
        if node:
            self.in_order_traversal(node.left)
            print(node.data, end=" ")
            self.in_order_traversal(node.right)
```

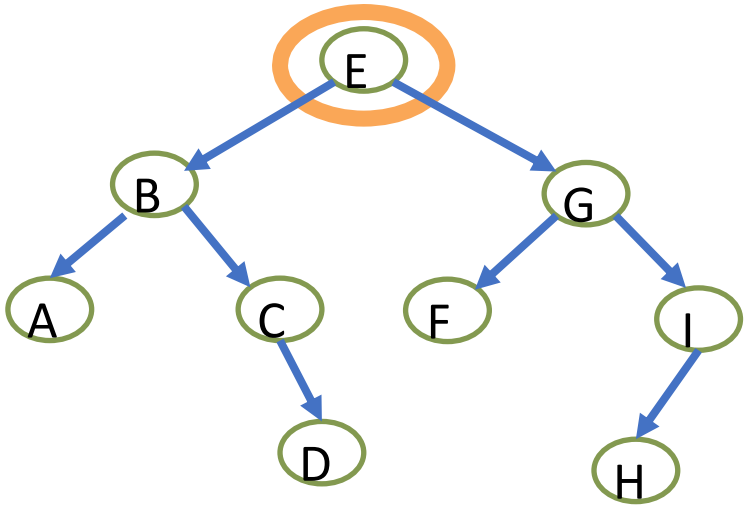
Output:

```
A B C D E F G H I
```



Post-order Depth First Traversal

- Pre-order
  - Process the current node's data
  - Visit the left child subtree
  - Visit the right child subtree
- In-order
  - Visit the left child subtree
  - Process the current node's data
  - Visit the right child subtree
- **Post-order**
  - **Visit the left child subtree**
  - **Visit the right child subtree**
  - **Process the current node's data**

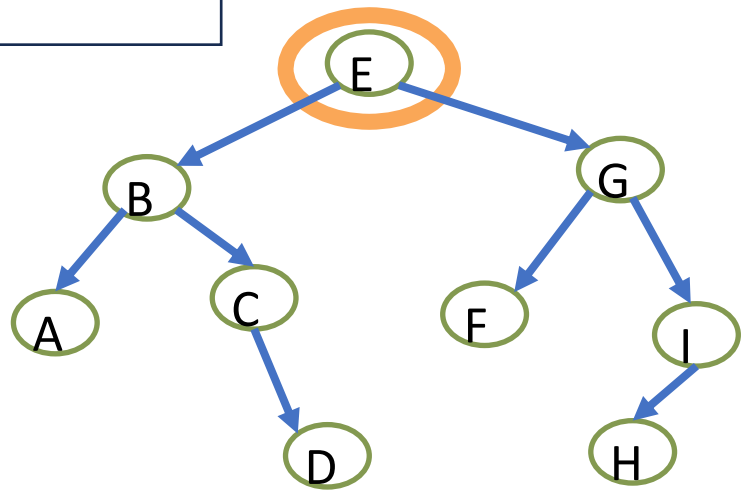


Post-order Depth First Traversal

```
def post_order_traversal(self, node):
        if node:
            self.post_order_traversal(node.left)
            self.post_order_traversal(node.right)
            print(node.data, end=" ")
```

Output:

```
A D C B F H I G E
```



Summary of Depth First Traversal

Pre-Order Traversal

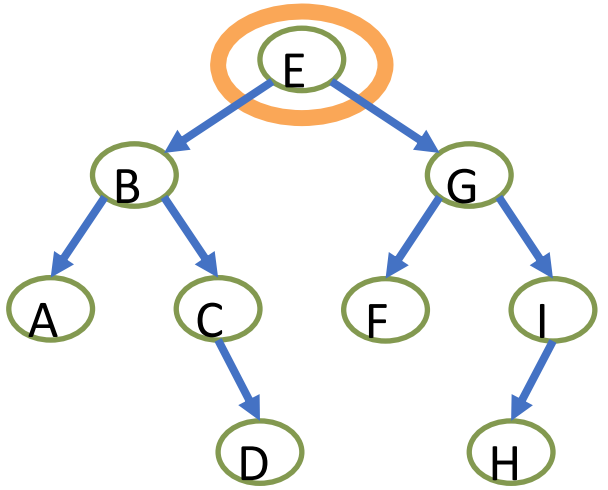
```
E  B  A  C  D  G  F  I  H
```

In-Order Traversal

```
A  B  C  D  E  F  G  H  I
```

Post-Order Traversal

```
A  D  C  B  F  H  I  G  E
```



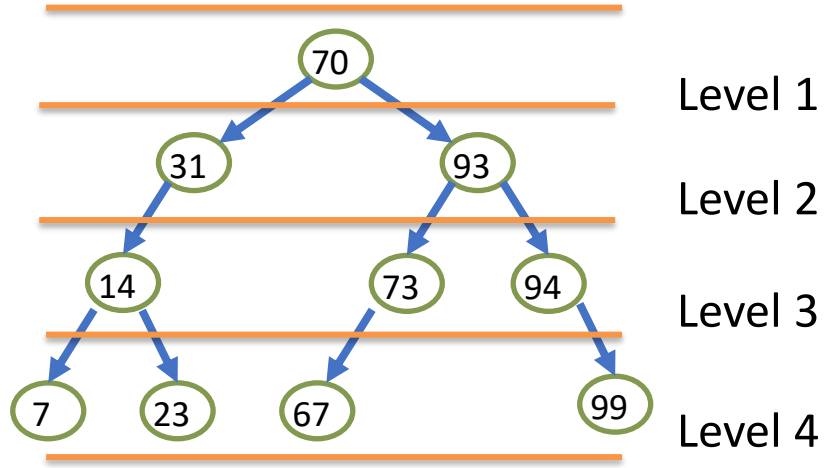
Breath-first Traversal: Level-by-level

Level-By-Level Traversal:

Visit the root (Level 1)

Visit children of the root (Level 2)

Visit grandchildren of the root (Level3) ...



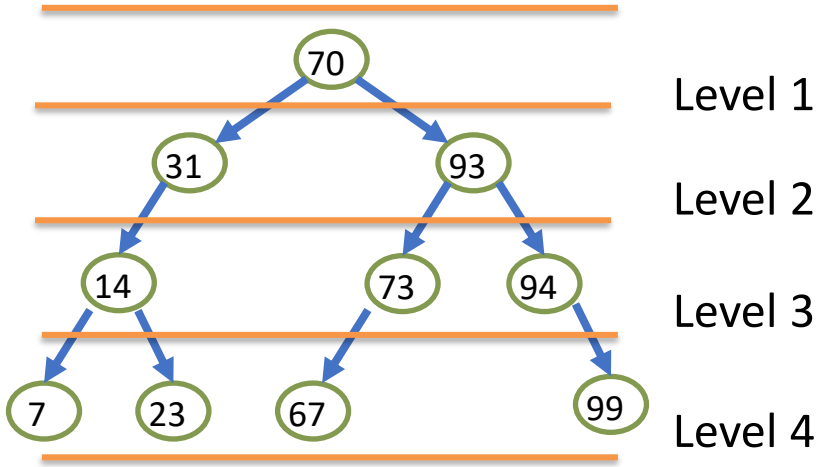
How?

- Visiting a node
- Remember all its children
  - Use a queue (FIFO structure)

Breath-first Traversal: Level-by-level

Level-By-Level Traversal:

- Visiting a node
- Remember all its children
  - Use a queue (FIFO structure)

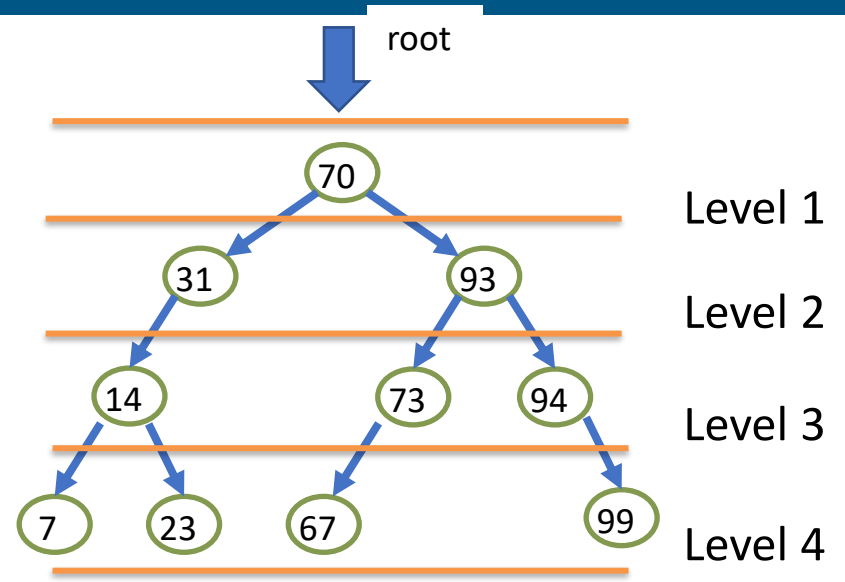


1. Enqueue the current node
2. Dequeue a node
3. Enqueue its children if it is available
4. Repeat Step 2 until the queue is empty

Breath-first Traversal: Level-by-level

Level-By-Level Traversal:

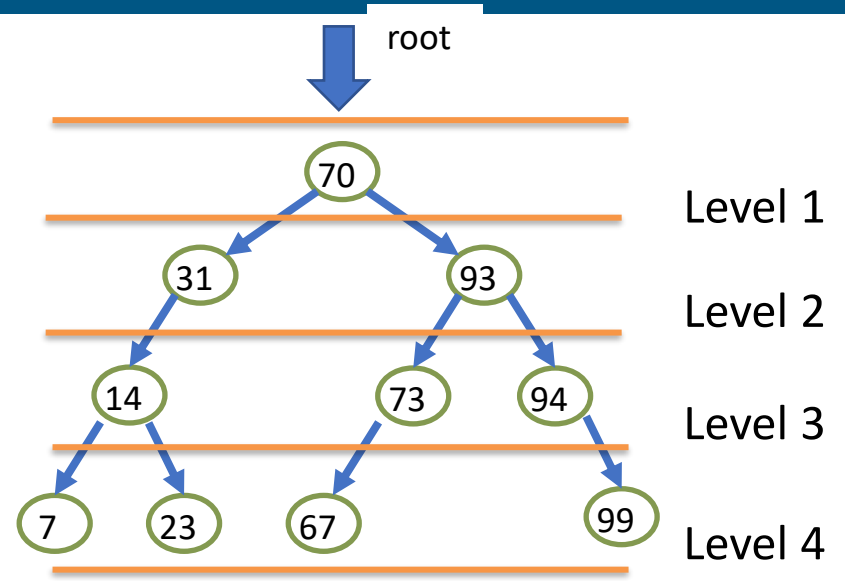
1. **Enqueue the current node**
2. Dequeue a node
3. Enqueue its children if it is available
4. Repeat Step 2 until the queue is empty




```
def level_order_traversal(self):
    if not self.root:
        return

    queue = Queue()
    queue.enqueue(self.root)

    while not queue.is_empty():
        current_node = queue.dequeue()
        print(current_node.data, end=" ")

        if current_node.left:
            queue.enqueue(current_node.left)
        if current_node.right:
            queue.enqueue(current_node.right)
```

Breath-first Traversal: Level-by-level

1. **Initialize the queue:**
   - Create an instance of the Queue class.
   - Add the root node to the queue using **queue.enqueue(self.root).**
2. **Start the traversal loop:**
   - While the queue is not empty:
     - Dequeue the front node using **queue.dequeue()** and print its data.
     - Enqueue the left child if it exists.
     - Enqueue the right child if it exists.
3. **Finish traversal:**
   - Repeat the process until the queue is empty, ensuring level-by-level traversal.




```
def level_order_traversal(self):
    if not self.root:
        return

    queue = Queue()
    queue.enqueue(self.root)

    while not queue.is_empty():
        current_node = queue.dequeue()
        print(current_node.data, end=" ")

        if current_node.left:
            queue.enqueue(current_node.left)
        if current_node.right:
            queue.enqueue(current_node.right)
```

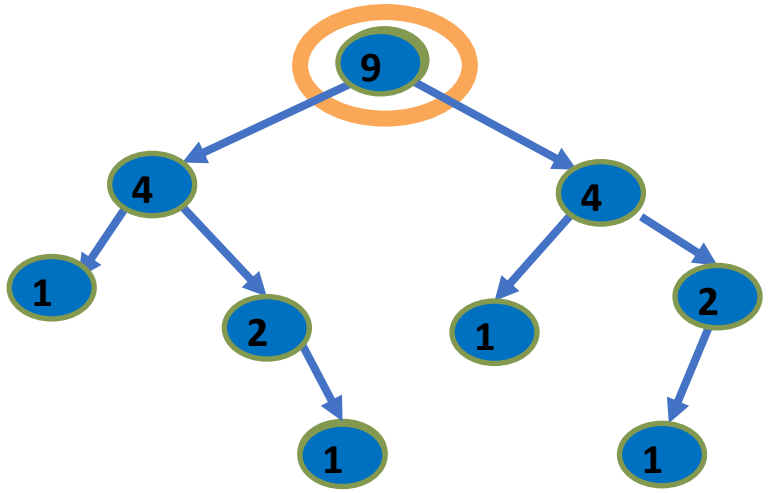
Count Nodes in a Binary Tree (SIZE)

- Recursive definition:
  - Number of nodes in a tree

    = 1

    + number of nodes in left subtree

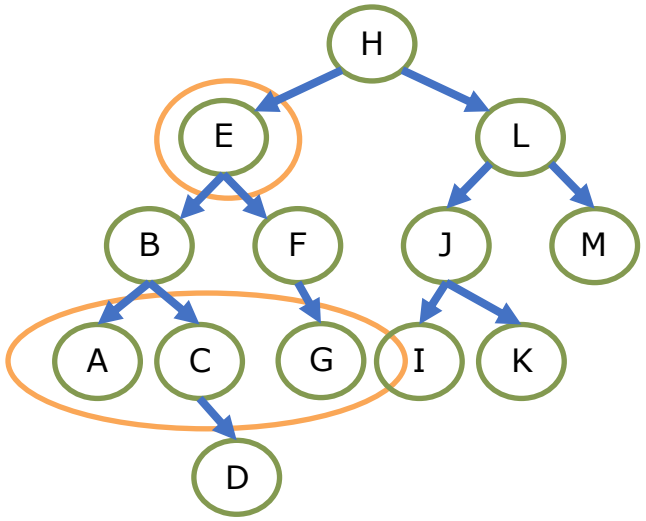
    + number of nodes in right subtree
- Each node returns the number of nodes in its subtree




```
    def count_nodes(self, node):
        if not node:
            return 0
        return 1 + self.count_nodes(node.left) + self.count_nodes(node.right)
```

Find the Kth-level Descendant Nodes

- Given a node X, find all the descendants of X
- Given node E and K=2, we should return its grandchild nodes A, C, and G
- What if we want to find K^{th}-level descendants?
  - **Need a way to keep track of how many levels down we've gone**




```
  def kth_level_descendants(self, node, k):
      if node is None:
          return []

      if k == 0:
          return [node.data]

      left_descendants = self.kth_level_descendants(node.left, k - 1)
      right_descendants = self.kth_level_descendants(node.right, k - 1)

      return left_descendants + right_descendants
...
  print(bt.kth_level_descendants(bt.root, 2))
```

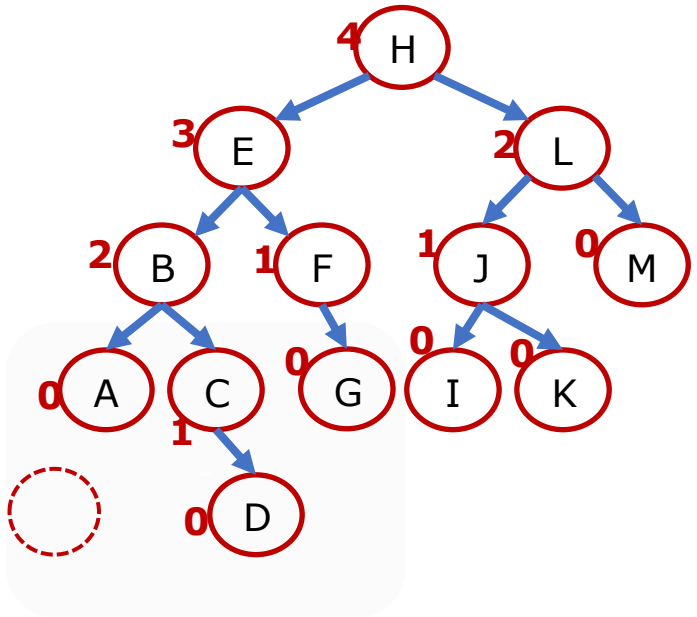
2-level descendants

X->left->left
X->left->right
X->right->left
X->right->right

Height of A Node in A Binary Tree

- The height of a tree: The number of edges on the longest path from the root to a leaf
  - Leaf node returns 0
  - None node (empty) returns -1

```
def calculate_height(self, node):
    if node is None:
        return -1

    left_height = self.calculate_height(node.left)
    right_height = self.calculate_height(node.right)

    return 1 + max(left_height, right_height)
```



Summary

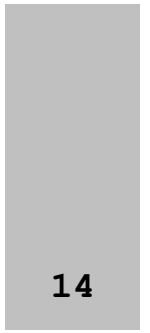
- Non-linear data structures
- Tree data structure
  - Binary trees
- Implement binary tree nodes in C
- Binary Tree Traversal
- Tree traversal order
  - Pre-order
  - In-order
  - Post-order
- Application examples
  - Count nodes in a binary tree
  - Find grandchild nodes
  - Calculate height of every node
- Level-by-level traversal
- **Preorder traversal with a stack**

Pre-order Traversal with Stack

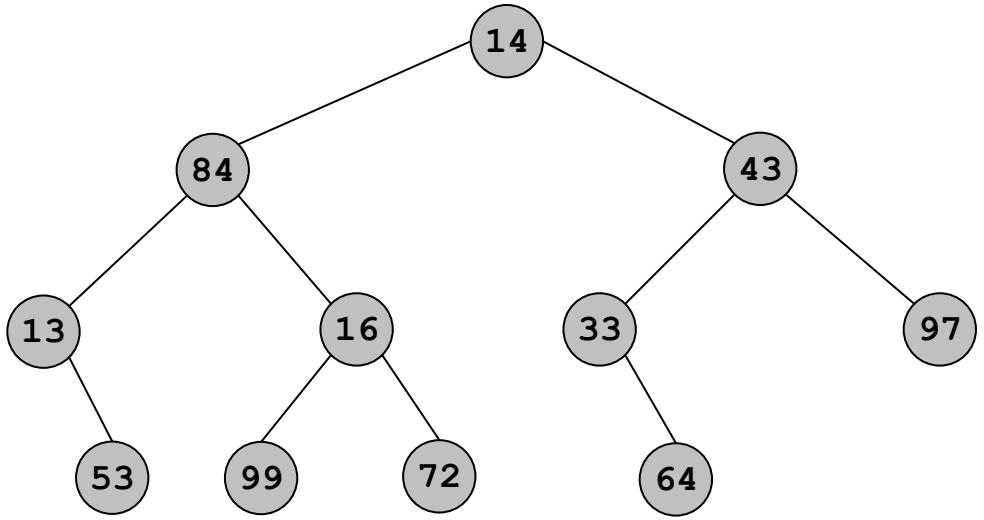
Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children



Stack

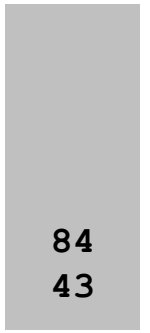


Pre-order Traversal with Stack

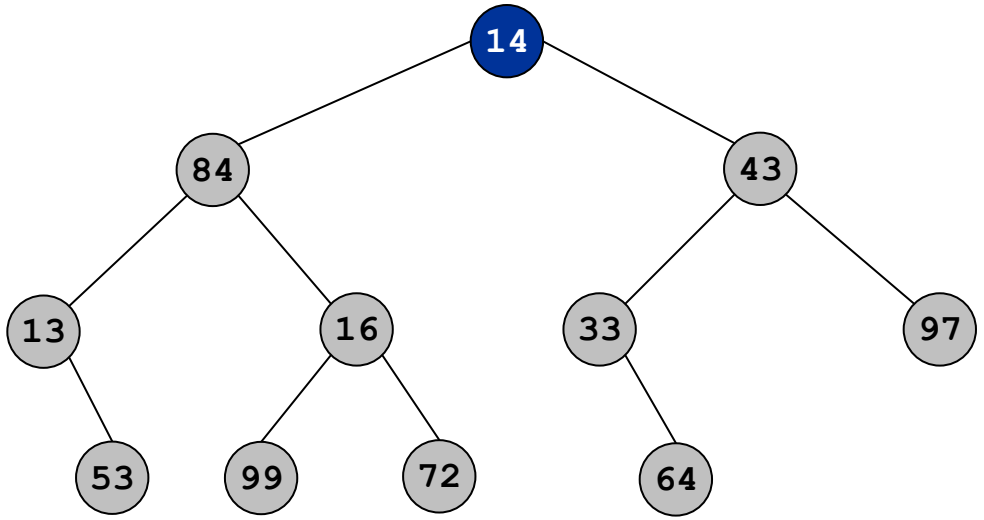
Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children



Stack



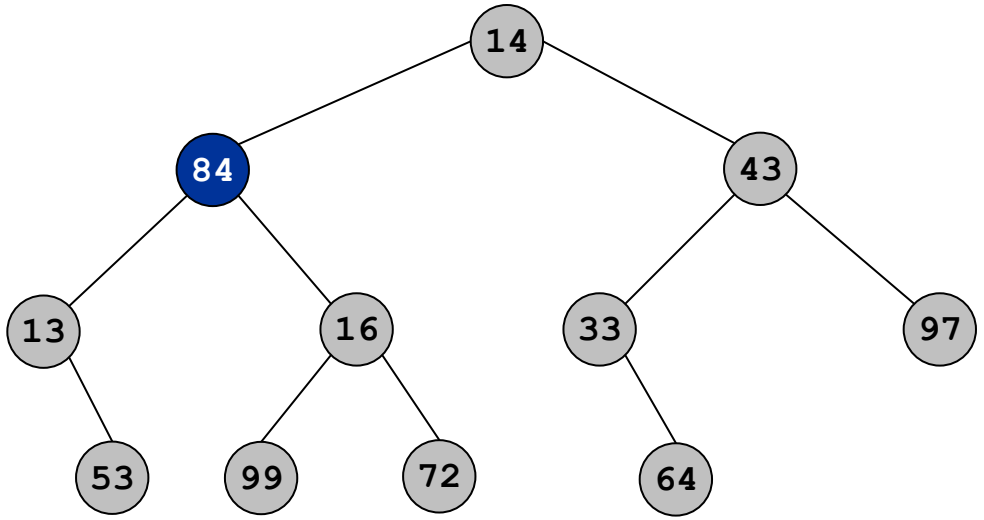
Pre-order Traversal with Stack

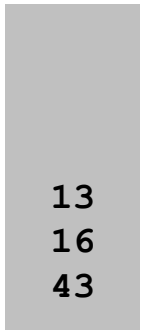
Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children

```
14 84
```





Stack

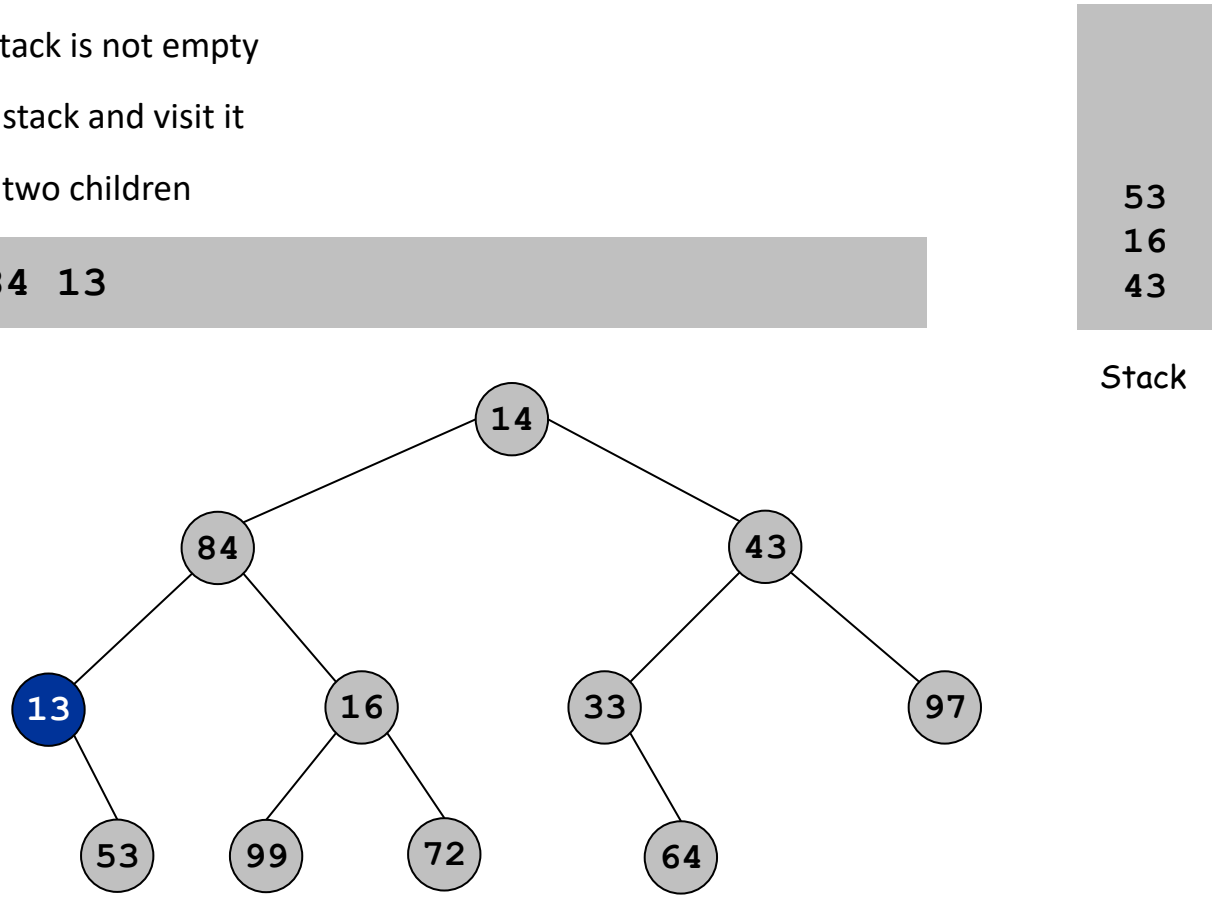
Pre-order Traversal with Stack

Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children

```
14 84 13
```



Stack

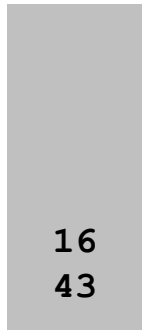
Pre-order Traversal with Stack

Push the root onto the stack.

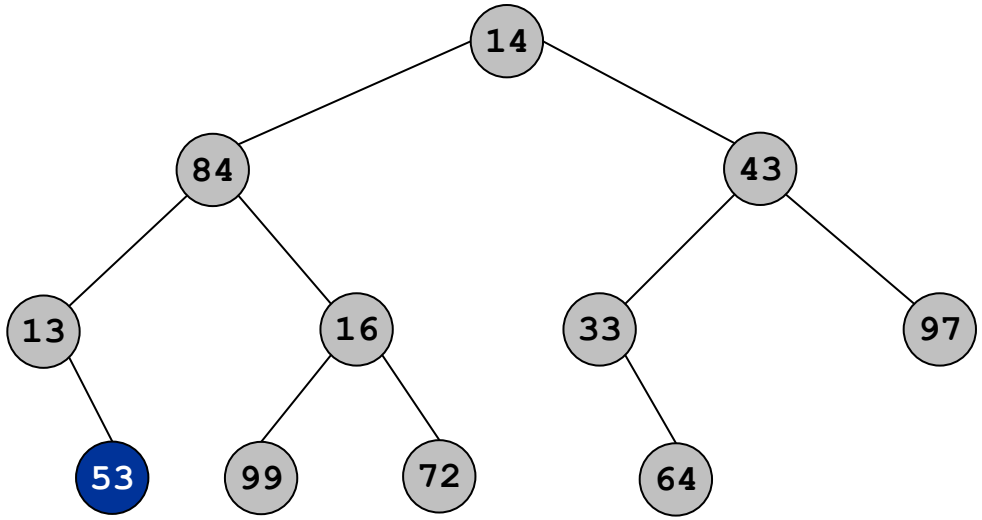
While the stack is not empty

- pop the stack and visit it
- push its two children

```
14 84 13 53
```



Stack



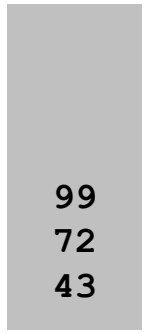
Pre-order Traversal with Stack

Push the root onto the stack.

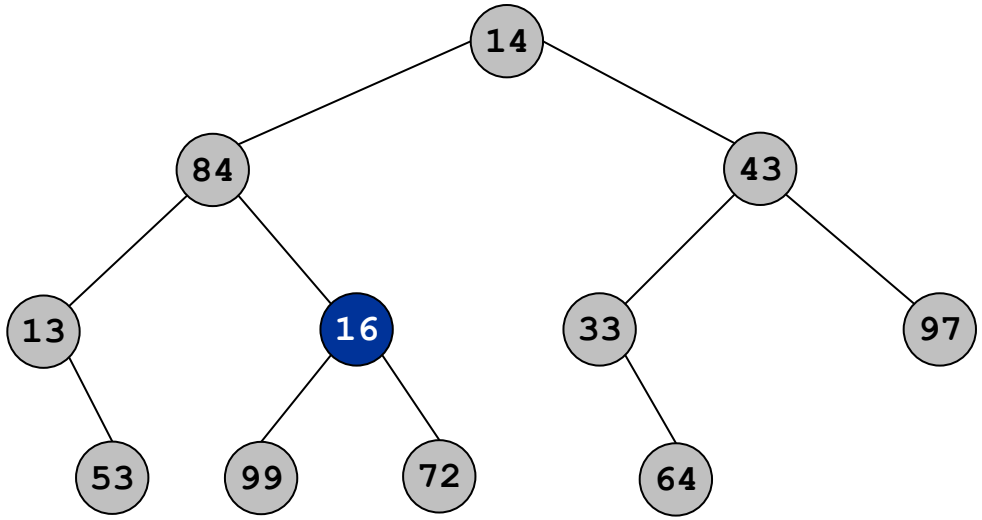
While the stack is not empty

- pop the stack and visit it
- push its two children

```
14 84 13 53 16
```



Stack



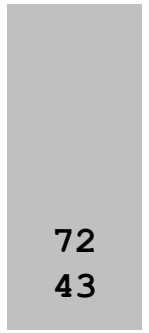
Pre-order Traversal with Stack

Push the root onto the stack.

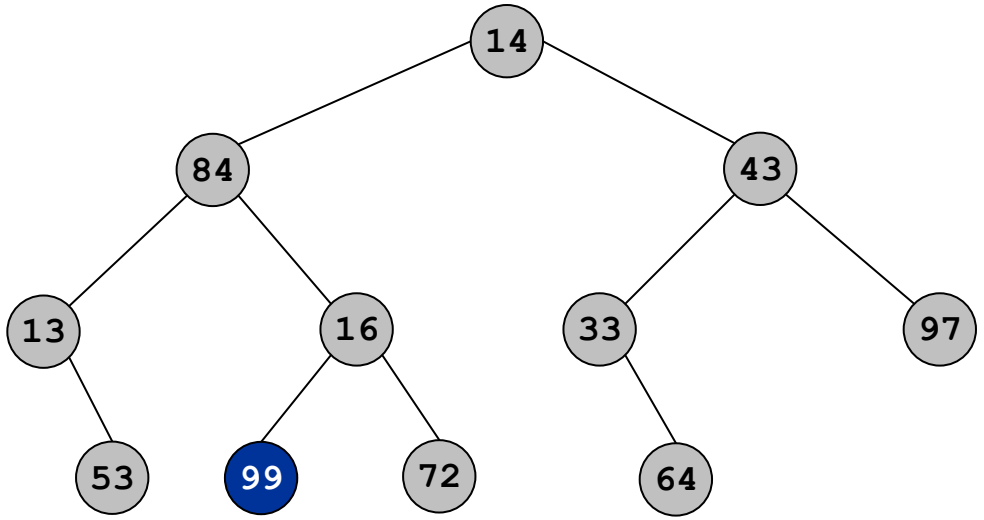
While the stack is not empty

- pop the stack and visit it
- push its two children

```
14 84 13 53 16 99
```



Stack



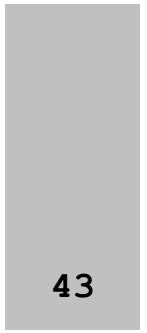
Pre-order Traversal with Stack

Push the root onto the stack.

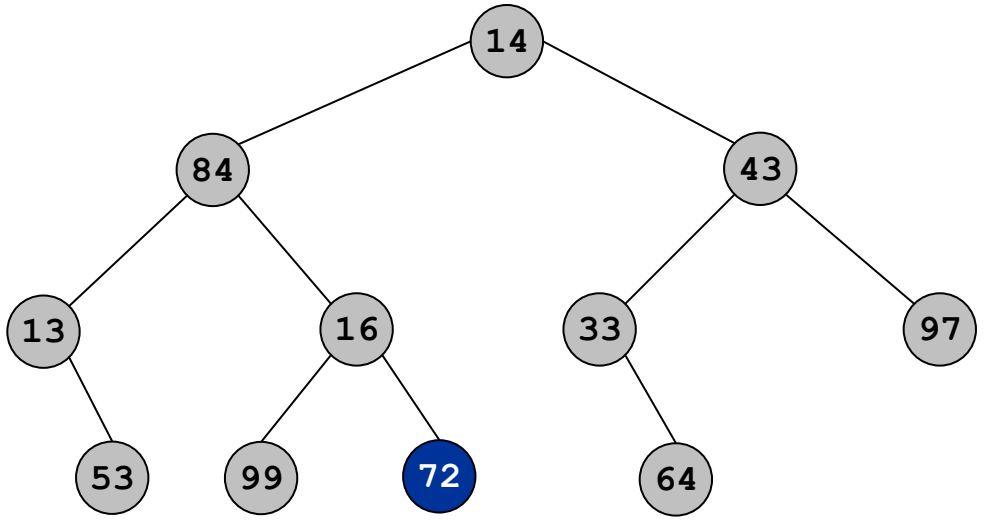
While the stack is not empty

- pop the stack and visit it
- push its two children

```
14 84 13 53 16 99 72
```



Stack



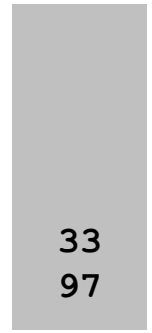
Pre-order Traversal with Stack

Push the root onto the stack.

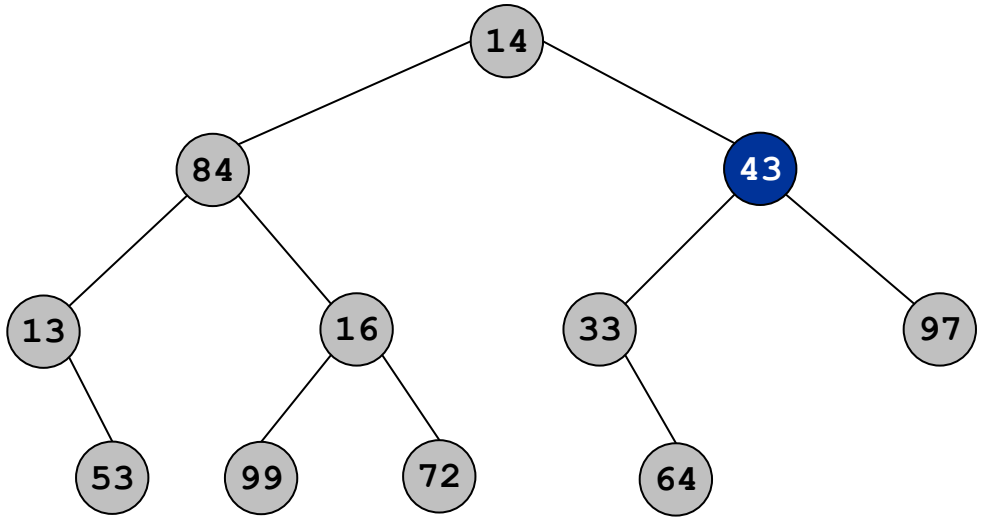
While the stack is not empty

- pop the stack and visit it
- push its two children

```
14 84 13 53 16 99 72 43
```



Stack



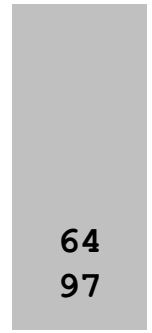
Pre-order Traversal with Stack

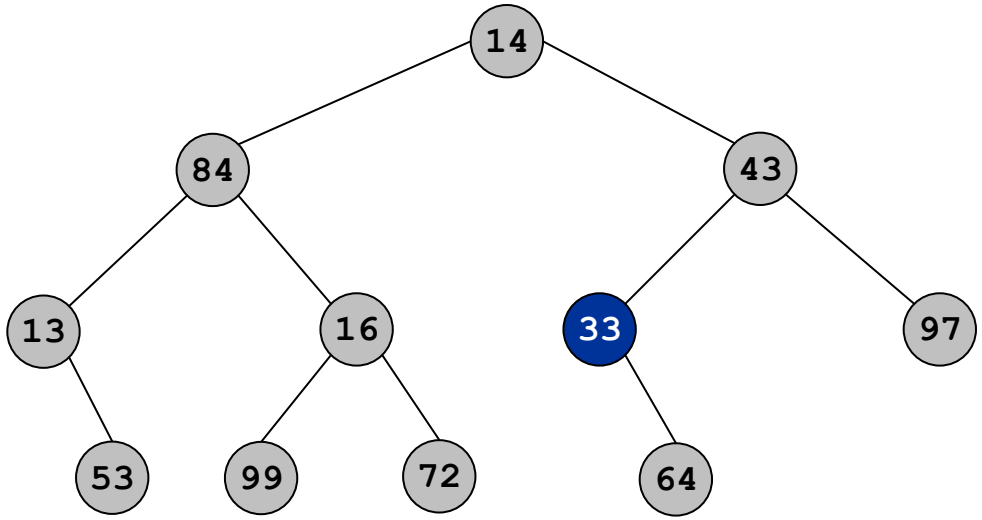
Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children

```
14 84 13 53 16 99 72 43 33
```





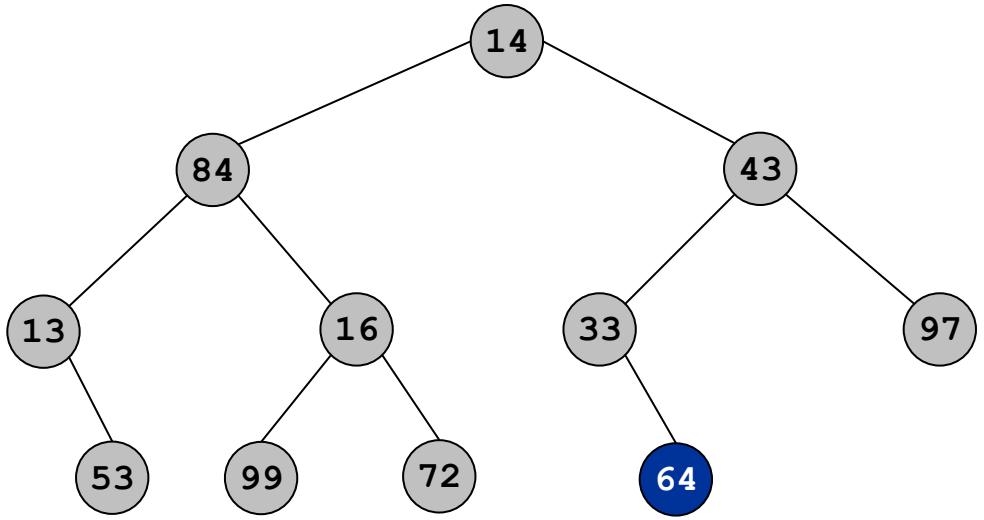
Pre-order Traversal with Stack

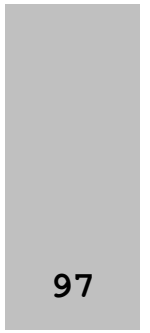
Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children

```
14 84 13 53 16 99 72 43 33 64
```





Stack

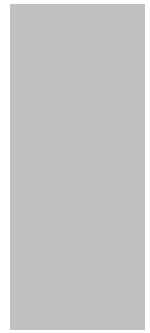
Pre-order Traversal with Stack

Push the root onto the stack.

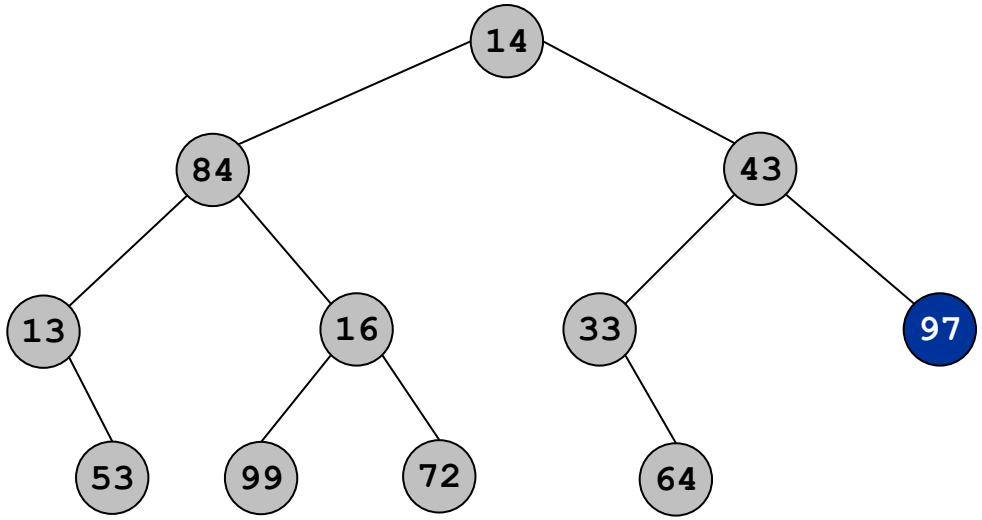
While the stack is not empty

- pop the stack and visit it
- push its two children

```
14 84 13 53 16 99 72 43 33 64 97
```



Stack



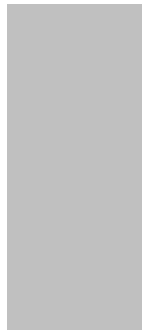
Pre-order Traversal with Stack

Push the root onto the stack.

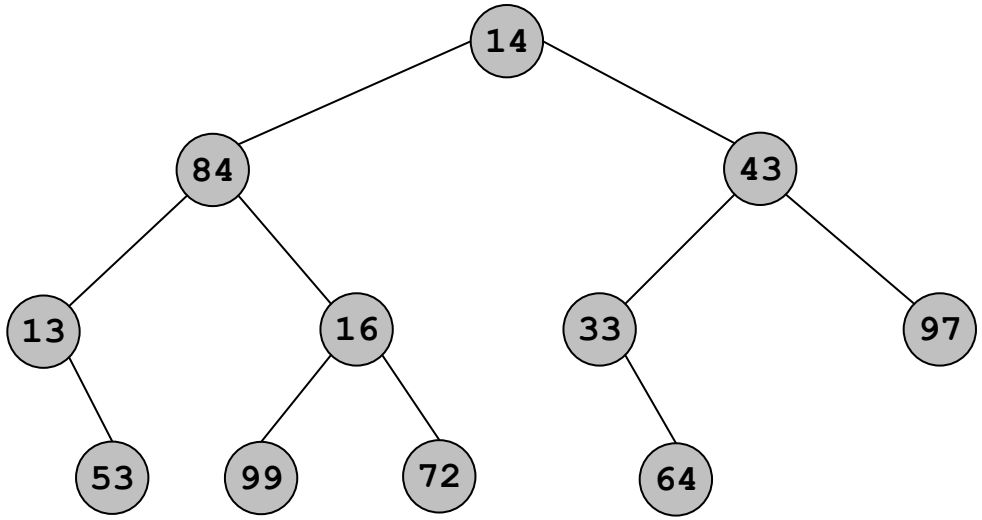
While the stack is not empty

- pop the stack and visit it
- push its two children

```
14 84 13 53 16 99 72 43 33 64 97
```



Stack



Summary

- Non-linear data structures
- Tree data structure
  - Binary trees
- Implement binary tree nodes in C
- Binary Tree Traversal
- Tree traversal order
  - Pre-order
  - In-order
  - Post-order
- Application examples
  - Count nodes in a binary tree
  - Find grandchild nodes
  - Calculate height of every node
- Level-by-level traversal
- Preorder traversal with a stack