**Trees and Graphs Using Python for Interviews**

**1. Basic Concepts and Features**

**Trees:**

* **Definition:** A tree is a hierarchical data structure consisting of nodes, with one node designated as the root. Each node can have zero or more child nodes, and there are no cycles.
  + **Binary Tree:** Each node has at most two children.
  + **Binary Search Tree (BST):** A binary tree where the left subtree contains nodes with values smaller than the root, and the right subtree contains nodes with values greater than the root.
  + **AVL Tree:** A self-balancing BST where the height difference between left and right subtrees is at most 1.

**Graphs:**

* **Definition:** A graph consists of vertices (nodes) and edges connecting pairs of vertices.
  + **Types:**
    - **Directed Graph:** Edges have a direction.
    - **Undirected Graph:** Edges do not have a direction.
    - **Weighted Graph:** Edges have weights.
    - **Unweighted Graph:** Edges do not have weights.

**Tree Traversal Techniques:**

* **In-Order Traversal:** Left, Root, Right
* **Pre-Order Traversal:** Root, Left, Right
* **Post-Order Traversal:** Left, Right, Root
* **Level-Order Traversal:** Level by level (BFS on a tree)

**Graph Traversal Techniques:**

* **DFS (Depth-First Search):** Explores as far as possible along a branch before backtracking.
* **BFS (Breadth-First Search):** Explores all neighbors of a node before moving to the next level.

**Graph Representations:**

* **Adjacency Matrix:** A 2D array where matrix[i][j] is 1 if there is an edge from node i to node j.
* **Adjacency List:** A list of lists where list[i] contains the neighbors of node i.
* **Edge List:** A list of pairs representing all edges.

**2. Implementation**

**Binary Tree Traversals:**

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.left = None

self.right = None

def in\_order\_traversal(root):

if root:

in\_order\_traversal(root.left)

print(root.value, end=" ")

in\_order\_traversal(root.right)

def pre\_order\_traversal(root):

if root:

print(root.value, end=" ")

pre\_order\_traversal(root.left)

pre\_order\_traversal(root.right)

def post\_order\_traversal(root):

if root:

post\_order\_traversal(root.left)

post\_order\_traversal(root.right)

print(root.value, end=" ")

**Graph BFS and DFS:**

from collections import deque

# BFS

def bfs(graph, start):

visited = set()

queue = deque([start])

while queue:

node = queue.popleft()

if node not in visited:

print(node, end=" ")

visited.add(node)

queue.extend(graph[node] - visited)

# DFS

def dfs(graph, start, visited=None):

if visited is None:

visited = set()

if start not in visited:

print(start, end=" ")

visited.add(start)

for neighbor in graph[start]:

dfs(graph, neighbor, visited)

**Graph Representations:**

# Adjacency List

graph = {

'A': {'B', 'C'},

'B': {'A', 'D', 'E'},

'C': {'A', 'F'},

'D': {'B'},

'E': {'B', 'F'},

'F': {'C', 'E'}

}

# Adjacency Matrix

adj\_matrix = [

[0, 1, 1, 0, 0, 0],

[1, 0, 0, 1, 1, 0],

[1, 0, 0, 0, 0, 1],

[0, 1, 0, 0, 0, 0],

[0, 1, 0, 0, 0, 1],

[0, 0, 1, 0, 1, 0]

]

**3. Intermediate Topics**

**Dijkstra’s Algorithm:**

import heapq

def dijkstra(graph, start):

distances = {node: float('inf') for node in graph}

distances[start] = 0

priority\_queue = [(0, start)]

while priority\_queue:

current\_distance, current\_node = heapq.heappop(priority\_queue)

if current\_distance > distances[current\_node]:

continue

for neighbor, weight in graph[current\_node].items():

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(priority\_queue, (distance, neighbor))

return distances

**Cycle Detection in Graphs:**

# Using DFS for Directed Graph

def detect\_cycle(graph):

visited = set()

rec\_stack = set()

def dfs(node):

if node not in visited:

visited.add(node)

rec\_stack.add(node)

for neighbor in graph[node]:

if neighbor not in visited and dfs(neighbor):

return True

elif neighbor in rec\_stack:

return True

rec\_stack.remove(node)

return False

for node in graph:

if dfs(node):

return True

return False

**4. Advanced Topics**

* **Trie Data Structure**: Efficient prefix matching.
* **Segment Tree**: Fast range queries and updates.
* **Floyd-Warshall Algorithm**: All-pairs shortest path in weighted graphs.

**5. Interview-Style Questions**

1. **Find the Height of a Binary Tree:**

def height(root):

if root is None:

return 0

return 1 + max(height(root.left), height(root.right))

1. **Check if a Tree is a BST:**

def is\_bst(root, min\_value=float('-inf'), max\_value=float('inf')):

if not root:

return True

if not (min\_value < root.value < max\_value):

return False

return is\_bst(root.left, min\_value, root.value) and is\_bst(root.right, root.value, max\_value)