# Tree Concepts

A **Tree** is a hierarchical data structure that consists of nodes connected by edges. Unlike linear structures like arrays and linked lists, trees organize data in a branching manner, making them ideal for representing hierarchical relationships such as file systems, organizational charts, and databases.

**1. Tree Terminology with Examples**

**1.1 Root Node**

The **root** is the topmost node in a tree. It serves as the starting point and has no parent.

**Example**

A <- Root

/ \

B C

/ \ \

D E F

In this example, **A** is the root node.

**1.2 Parent & Child Nodes**

A **parent** node has one or more child nodes connected below it. A **child** node is any node that has a parent.

**Example**

A

/ \

B C

* **A** is the parent of **B** and **C**.
* **B** and **C** are children of **A**.

**1.3 Leaf Nodes**

A **leaf** node has no children.

**Example**

A

/ \

B C

/ \

D E <- Leaf nodes

Nodes **D** and **E** are leaf nodes because they do not have any children.

**1.4 Depth & Height**

* **Depth of a node**: The number of edges from the root to that node.
* **Height of a node**: The longest path from that node to a leaf.

**Example**

A <- Depth: 0

/ \

B C <- Depth: 1

/ \

D E <- Depth: 2

* Depth of **A** = 0
* Depth of **C** = 1
* Depth of **D** = 2

If we measure height:

* Height of **D** = 0 (as it’s a leaf node)
* Height of **C** = 1 (longest path to a leaf is **C → D**)
* Height of **A** = 2 (longest path to a leaf is **A → C → D**)

**2. Implementing a Basic Tree Structure**

A simple tree node can be implemented using a class in Python. Each node contains a value and a list of children.

class TreeNode:

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

self.value = value

self.children = [] # Stores child nodes

def add\_child(self, child):

self.children.append(child)

# Example: Creating a tree

root = TreeNode("A")

child1 = TreeNode("B")

child2 = TreeNode("C")

root.add\_child(child1)

root.add\_child(child2)

print(root.value) # Output: A

print([child.value for child in root.children]) # Output: ['B', 'C']

**3. Tree Traversals**

Traversal means visiting all nodes in a tree in a specific order.

**3.1 Preorder Traversal (Root → Left → Right)**

* Visit the root node first.
* Recursively visit the left subtree.
* Recursively visit the right subtree.

**Example**

A

/ \

B C

/ \

D E

**Preorder Traversal Output:** A → B → D → E → C

def preorder\_traversal(node):

if node:

print(node.value, end=" ") # Visit root

for child in node.children:

preorder\_traversal(child)

preorder\_traversal(root) # Output: A B C

**3.2 Inorder Traversal (Left → Root → Right)**

* Recursively visit the left subtree.
* Visit the root node.
* Recursively visit the right subtree.

💡 **Note:** Inorder traversal is mainly for **binary trees**.

**Example**

A

/ \

B C

/ \

D E

**Inorder Traversal Output:** D → B → E → A → C

class TreeNode:

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

self.value = value

self.left = None

self.right = None

def inorder\_traversal(node):

if node:

inorder\_traversal(node.left) # Visit left subtree

print(node.value, end=" ") # Visit root

inorder\_traversal(node.right) # Visit right subtree

**3.3 Postorder Traversal (Left → Right → Root)**

* Recursively visit the left subtree.
* Recursively visit the right subtree.
* Visit the root node.

**Example**

A

/ \

B C

/ \

D E

**Postorder Traversal Output:** D → E → B → C → A

def postorder\_traversal(node):

if node:

for child in node.children:

postorder\_traversal(child)

print(node.value, end=" ") # Visit root

postorder\_traversal(root) # Output: B C A

**4. Common Tree Operations**

**4.1 Insertion**

Insertion depends on the type of tree. In a **general tree**, a new node is added as a child to a specific node.

class TreeNode:

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

self.value = value

self.children = []

def add\_child(self, child):

self.children.append(child)

root = TreeNode("A")

child1 = TreeNode("B")

root.add\_child(child1)

print(root.children[0].value) # Output: B

**4.2 Deletion**

Deletion involves removing a node and restructuring the tree if necessary. In a **binary tree**, if a node has children, they must be reassigned.

def delete\_node(parent, value):

for i, child in enumerate(parent.children):

if child.value == value:

del parent.children[i] # Delete node

return True

return False

delete\_node(root, "B")

**4.3 Searching**

Searching involves traversing the tree to find a specific value.

def search(node, key):

if node is None:

return False

if node.value == key:

return True

for child in node.children:

if search(child, key):

return True

return False

print(search(root, "B")) # Output: True

**Summary**

1. **Tree Terminology**: Root, Parent, Child, Leaf, Depth, Height.
2. **Basic Tree Implementation**: Using a class with nodes and children.
3. **Tree Traversals**:
   * **Preorder**: Root → Left → Right
   * **Inorder**: Left → Root → Right (only for Binary Trees)
   * **Postorder**: Left → Right → Root
4. **Common Tree Operations**:
   * **Insertion**: Adding a new node.
   * **Deletion**: Removing a node while maintaining structure.
   * **Searching**: Finding a specific node.

# **Binary Search Tree**

A **Binary Search Tree (BST)** is a binary tree where each node follows the **BST property**:

* The left subtree contains nodes with values **less than** the parent node.
* The right subtree contains nodes with values **greater than** the parent node.
* This structure allows for efficient searching, insertion, and deletion in **O(log n)** time on average.

## ****1. Properties of BST****

A **Binary Search Tree** follows these rules:

1. **Each node has at most two children** (left and right).
2. **Left subtree** contains values **smaller** than the node.
3. **Right subtree** contains values **greater** than the node.
4. **No duplicate values** are allowed.

#### ****Example BST****

50

/ \

30 70

/ \ / \

20 40 60 80

* The **left subtree** of 50 (30, 20, 40) contains values < 50.
* The **right subtree** of 50 (70, 60, 80) contains values > 50.

## ****2. Implementing BST****

### ****2.1 BST Node Class****

A BST consists of **nodes**, each containing:

* A value (key).
* A pointer to the **left** child.
* A pointer to the **right** child.

class BSTNode:

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

self.key = key

self.left = None

self.right = None

### ****2.2 Insertion in BST****

To insert a value in BST:

* If the tree is empty, create the root node.
* If the value is **less** than the root, insert it in the **left** subtree.
* If the value is **greater**, insert it in the **right** subtree.

def insert(root, key):

if root is None:

return BSTNode(key)

if key < root.key:

root.left = insert(root.left, key)

else:

root.right = insert(root.right, key)

return root

# Example usage:

root = None

root = insert(root, 50)

insert(root, 30)

insert(root, 70)

insert(root, 20)

insert(root, 40)

insert(root, 60)

insert(root, 80)

**Tree after insertion:**

50

/ \

30 70

/ \ / \

20 40 60 80

### ****2.3 Search (Contains function)****

To search for a value:

* If the node is None, return False.
* If the node’s key matches, return True.
* If the key is **smaller**, search in the **left** subtree.
* If the key is **greater**, search in the **right** subtree.

def search(root, key):

if root is None or root.key == key:

return root is not None

if key < root.key:

return search(root.left, key)

else:

return search(root.right, key)

# Example:

print(search(root, 40)) # Output: True

print(search(root, 100)) # Output: False

### ****2.4 Deletion in BST****

There are **three cases** when deleting a node:

1. **Leaf Node (No Children)** → Simply remove the node.
2. **One Child** → Replace the node with its child.
3. **Two Children** → Replace the node with its **inorder successor** (smallest node in the right subtree).

def find\_min(node):

while node.left:

node = node.left

return node

def delete(root, key):

if root is None:

return root

if key < root.key:

root.left = delete(root.left, key)

elif key > root.key:

root.right = delete(root.right, key)

else:

# Case 1 & 2: One or No Child

if root.left is None:

return root.right

elif root.right is None:

return root.left

# Case 3: Two children

temp = find\_min(root.right)

root.key = temp.key

root.right = delete(root.right, temp.key)

return root

# Example:

root = delete(root, 30) # Deletes 30

## ****3. Tree Traversals****

### ****3.1 Inorder Traversal (Left → Root → Right)****

**Used for:** Sorting elements in increasing order.

def inorder(root):

if root:

inorder(root.left)

print(root.key, end=" ")

inorder(root.right)

inorder(root) # Output: 20 40 50 60 70 80

### ****3.2 Preorder Traversal (Root → Left → Right)****

**Used for:** Copying the tree structure.

def preorder(root):

if root:

print(root.key, end=" ")

preorder(root.left)

preorder(root.right)

preorder(root) # Output: 50 40 20 70 60 80

### ****3.3 Postorder Traversal (Left → Right → Root)****

**Used for:** Deleting a tree (as children are deleted before the parent).

def postorder(root):

if root:

postorder(root.left)

postorder(root.right)

print(root.key, end=" ")

postorder(root) # Output: 20 40 60 80 70 50

## ****4. Find the Closest Value to a Given Number in BST****

To find the closest value to a target:

* Start from the root and track the **closest value** found so far.
* Move **left** or **right** based on the target value.

def find\_closest(root, target, closest=None):

if root is None:

return closest

if closest is None or abs(root.key - target) < abs(closest - target):

closest = root.key

if target < root.key:

return find\_closest(root.left, target, closest)

else:

return find\_closest(root.right, target, closest)

# Example:

print(find\_closest(root, 65)) # Output: 60

print(find\_closest(root, 45)) # Output: 40

## ****5. Validate if a Given Tree is a BST****

A valid BST should have all values in the left subtree **less than** the root and all values in the right subtree **greater than** the root.

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

if root is None:

return True

if not (min\_val < root.key < max\_val):

return False

return (is\_valid\_bst(root.left, min\_val, root.key) and

is\_valid\_bst(root.right, root.key, max\_val))

# Example:

print(is\_valid\_bst(root)) # Output: True

## ****Summary****

### ****Binary Search Tree Properties****

✅ Every node follows the BST ordering rule.  
✅ Left subtree < Parent < Right subtree.  
✅ No duplicate values.

### ****Operations****

1. **Insertion** → Adds a node at the correct position.
2. **Search (Contains function)** → Finds if a value exists.
3. **Deletion** → Handles 3 cases (leaf node, one child, two children).
4. **Tree Traversals**
   * **Inorder** → Left → Root → Right (Sorted order)
   * **Preorder** → Root → Left → Right (Used for copying a tree)
   * **Postorder** → Left → Right → Root (Used for deletion)
5. **Finding the closest value** to a target number.
6. **Validating a BST** to check if it follows BST rules.

# **Heap Concepts**

A **Heap** is a specialized **binary tree-based** data structure that satisfies the **heap property**. It is mainly used for **priority queues**, **heap sort**, and **graph algorithms** like Dijkstra’s shortest path.

## ****1. Understanding Min Heap & Max Heap****

### ****1.1 Min Heap****

A **Min Heap** is a binary tree where:

* The **smallest** element is always at the **root**.
* Each parent node is **less than or equal to** its children.

#### ****Example Min Heap****

10

/ \

20 15

/ \

40 50

* 10 (smallest value) is at the root.
* Every parent is **smaller** than its children (10 < 20, 15, 20 < 40, 50).

### ****1.2 Max Heap****

A **Max Heap** is a binary tree where:

* The **largest** element is always at the **root**.
* Each parent node is **greater than or equal to** its children.

#### ****Example Max Heap****

50

/ \

30 40

/ \

10 20

* 50 (largest value) is at the root.
* Every parent is **greater** than its children (50 > 30, 40, 30 > 10, 20).

## ****2. Heap Operations****

### ****2.1 Build Heap (Heapify Process)****

Heapify ensures that a given array satisfies the heap property. It is done **bottom-up** to efficiently convert an **unordered array** into a **valid heap**.

**Time Complexity:** O(n) (better than inserting one-by-one O(n log n)).

#### ****Implementation****

def heapify(arr, n, i, is\_min\_heap=True):

root = i

left = 2 \* i + 1

right = 2 \* i + 2

if is\_min\_heap:

# Min Heap condition

if left < n and arr[left] < arr[root]:

root = left

if right < n and arr[right] < arr[root]:

root = right

else:

# Max Heap condition

if left < n and arr[left] > arr[root]:

root = left

if right < n and arr[right] > arr[root]:

root = right

if root != i:

arr[i], arr[root] = arr[root], arr[i]

heapify(arr, n, root, is\_min\_heap)

def build\_heap(arr, is\_min\_heap=True):

n = len(arr)

for i in range(n // 2 - 1, -1, -1):

heapify(arr, n, i, is\_min\_heap)

# Example:

arr = [50, 20, 30, 10, 5]

build\_heap(arr, is\_min\_heap=True) # Converts to Min Heap

print(arr) # Output: [5, 10, 30, 20, 50]

### ****2.2 Insert into Heap****

* Add the new element at the end of the heap.
* **Bubble up** (heapify up) to restore heap property.
* **Time Complexity:** O(log n)

#### ****Implementation****

def insert(heap, key, is\_min\_heap=True):

heap.append(key)

i = len(heap) - 1

parent = (i - 1) // 2

while i > 0 and ((is\_min\_heap and heap[i] < heap[parent]) or (not is\_min\_heap and heap[i] > heap[parent])):

heap[i], heap[parent] = heap[parent], heap[i]

i = parent

parent = (i - 1) // 2

# Example:

heap = [10, 20, 30, 40, 50]

insert(heap, 5, is\_min\_heap=True) # Insert into Min Heap

print(heap) # Output: [5, 10, 30, 40, 50, 20]

### ****2.3 Remove (Extract Min/Max from Heap)****

* The **root** (min/max element) is removed.
* The **last element** is moved to the root.
* **Heapify down** is performed to restore heap property.
* **Time Complexity:** O(log n)

#### ****Implementation****

def extract(heap, is\_min\_heap=True):

if len(heap) == 0:

return None

if len(heap) == 1:

return heap.pop()

root = heap[0]

heap[0] = heap.pop() # Move last element to root

heapify(heap, len(heap), 0, is\_min\_heap)

return root

# Example:

heap = [5, 10, 30, 40, 50, 20]

min\_value = extract(heap, is\_min\_heap=True) # Removes min (5)

print(min\_value) # Output: 5

print(heap) # Updated heap after extraction

## ****3. Heap Sort Algorithm****

**Heap Sort** uses the heap property to sort an array.

* Build a **Max Heap** from the array.
* Repeatedly **extract the max** and swap it with the last element.
* **Heapify** the remaining heap.
* **Time Complexity:** O(n log n)

#### ****Implementation****

def heap\_sort(arr):

n = len(arr)

build\_heap(arr, is\_min\_heap=False) # Convert to Max Heap

for i in range(n - 1, 0, -1):

arr[0], arr[i] = arr[i], arr[0] # Swap max with last

heapify(arr, i, 0, is\_min\_heap=False) # Heapify reduced heap

# Example:

arr = [10, 20, 15, 30, 40]

heap\_sort(arr)

print(arr) # Output: [10, 15, 20, 30, 40] (Sorted array)

## ****Summary****

### ****1. Min Heap & Max Heap****

✅ **Min Heap** → Root contains the smallest element.  
✅ **Max Heap** → Root contains the largest element.

### ****2. Heap Operations****

1. **Build Heap** → Converts an array into a heap (O(n)).
2. **Insert** → Adds a new element and restores heap property (O(log n)).
3. **Remove (Extract Min/Max)** → Removes root and restores heap (O(log n)).

### ****3. Heap Sort Algorithm****

✅ Uses **Max Heap** for sorting in O(n log n).

# **Trie Concepts**

A **Trie** (also called a **Prefix Tree**) is a tree-based data structure used to store and retrieve strings efficiently. It is widely used for **dictionary operations**, **autocomplete**, **spell checking**, and **IP routing**.

## ****1. Understanding Trie (Prefix Tree)****

* Each **node** represents a **character** of a word.
* The **root node** is empty ("").
* Each path from the root to a **leaf node** forms a word.
* A **boolean flag** (is\_end\_of\_word) marks the end of a valid word.

### ****Example Trie****

Let's insert "cat", "car", and "dog" into a Trie:

(root)

/ \

c d

/ \ \

a a o

/ \ \

t r g

(✓) (✓) (✓)

* The ✓ marks the **end of a valid word** ("cat", "car", "dog").
* The **common prefix** ("ca") is stored once, making the Trie **memory efficient**.

## ****2. Implement Trie****

### ****2.1 Insert Words into Trie****

Each letter of the word is inserted **one by one** as a **child node**.  
If a character is **already present**, we move to the next level.

**Time Complexity:** O(n), where n is the word length.

#### ****Implementation****

class TrieNode:

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

self.children = {} # Dictionary to store children nodes

self.is\_end\_of\_word = False # Marks the end of a valid word

class Trie:

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

self.root = TrieNode()

def insert(self, word):

node = self.root

for char in word:

if char not in node.children:

node.children[char] = TrieNode()

node = node.children[char]

node.is\_end\_of\_word = True # Marks word completion

# Example:

trie = Trie()

trie.insert("cat")

trie.insert("car")

trie.insert("dog")

### ****2.2 Search for a Word in Trie****

To check if a word exists:

* Traverse each character in the Trie.
* If any character is **missing**, return **False**.
* If all characters exist and the **last node is marked as a word**, return **True**.

**Time Complexity:** O(n), where n is the word length.

#### ****Implementation****

class Trie(Trie):

def search(self, word):

node = self.root

for char in word:

if char not in node.children:

return False # Word not found

node = node.children[char]

return node.is\_end\_of\_word # Check if it's a valid word

# Example:

print(trie.search("cat")) # Output: True

print(trie.search("bat")) # Output: False (Not inserted)

### ****2.3 Delete Words from Trie****

* Use **recursion** to delete characters **only if they are not part of another word**.
* The deletion process must ensure the Trie structure is preserved.

**Time Complexity:** O(n), where n is the word length.

#### ****Implementation****

class Trie(Trie):

def delete(self, word, node=None, depth=0):

if node is None:

node = self.root

if depth == len(word):

if not node.is\_end\_of\_word:

return False # Word doesn't exist

node.is\_end\_of\_word = False

return len(node.children) == 0 # Delete node if it has no children

char = word[depth]

if char not in node.children:

return False # Word doesn't exist

should\_delete = self.delete(word, node.children[char], depth + 1)

if should\_delete:

del node.children[char] # Remove character node

return len(node.children) == 0 and not node.is\_end\_of\_word # Check if parent can be deleted

return False

# Example:

trie.delete("cat") # Deletes "cat" but keeps "car"

print(trie.search("cat")) # Output: False

print(trie.search("car")) # Output: True (Still exists)

## ****3. Applications of Trie****

### ****3.1 Autocomplete Feature****

Trie is used to suggest words based on prefixes, like in **Google search**.

#### ****Implementation****

class Trie(Trie):

def autocomplete(self, prefix):

node = self.root

for char in prefix:

if char not in node.children:

return [] # No words with this prefix

node = node.children[char]

words = []

self.\_dfs(node, prefix, words)

return words

def \_dfs(self, node, prefix, words):

if node.is\_end\_of\_word:

words.append(prefix)

for char, child in node.children.items():

self.\_dfs(child, prefix + char, words)

# Example:

trie.insert("cat")

trie.insert("car")

trie.insert("cartoon")

trie.insert("dog")

print(trie.autocomplete("ca")) # Output: ['cat', 'car', 'cartoon']

### ****3.2 Spell Checking****

* Given a dictionary, Trie can check whether a word exists in **constant time**.
* It can also suggest **similar words** by checking prefixes.

#### ****Implementation****

def spell\_check(trie, word):

return trie.search(word)

# Example:

print(spell\_check(trie, "cat")) # Output: True

print(spell\_check(trie, "bat")) # Output: False

## ****Summary****

### ****1. Trie Overview****

✅ **Trie (Prefix Tree)** stores words efficiently by sharing prefixes.  
✅ Each node contains a **character** and a **dictionary** of children.

### ****2. Trie Operations****

1. **Insert** → Adds words to the Trie (O(n)).
2. **Search** → Checks if a word exists (O(n)).
3. **Delete** → Removes words while maintaining structure (O(n)).

### ****3. Trie Applications****

✅ **Autocomplete** → Suggests words from a given prefix.  
✅ **Spell Checking** → Quickly validates words in a dictionary.

# **Graph Concepts**

A **Graph** is a data structure that consists of **nodes (vertices)** and **edges (connections)**. It is widely used in various applications like **social networks, navigation systems, recommendation engines, and network routing**.

## ****1. Graph Representations****

There are two common ways to represent a graph:

### ****1.1 Adjacency Matrix****

* A **2D matrix** where matrix[i][j] = 1 if there is an edge from vertex i to j, otherwise 0.
* Works well for **dense graphs** (many edges).
* **Space Complexity**: O(V²), where V is the number of vertices.

#### ****Example:****

**Graph:**

(A) --- (B)

| |

(C) --- (D)

**Adjacency Matrix Representation:**

A B C D

A [0, 1, 1, 0]

B [1, 0, 0, 1]

C [1, 0, 0, 1]

D [0, 1, 1, 0]

#### ****Implementation****

class GraphMatrix:

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

self.V = vertices

self.graph = [[0] \* vertices for \_ in range(vertices)]

def add\_edge(self, u, v):

self.graph[u][v] = 1

self.graph[v][u] = 1 # For an undirected graph

def display(self):

for row in self.graph:

print(row)

# Example:

g = GraphMatrix(4)

g.add\_edge(0, 1)

g.add\_edge(0, 2)

g.add\_edge(1, 3)

g.add\_edge(2, 3)

g.display()

### ****1.2 Adjacency List****

* Uses a **dictionary (or list of lists)** where each vertex stores a list of connected vertices.
* Works well for **sparse graphs** (fewer edges).
* **Space Complexity**: O(V + E), where E is the number of edges.

#### ****Example:****

{

A: [B, C],

B: [A, D],

C: [A, D],

D: [B, C]

}

#### ****Implementation****

from collections import defaultdict

class GraphList:

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

self.graph = defaultdict(list)

def add\_edge(self, u, v):

self.graph[u].append(v)

self.graph[v].append(u) # For an undirected graph

def display(self):

for node, neighbors in self.graph.items():

print(f"{node}: {neighbors}")

# Example:

g = GraphList()

g.add\_edge("A", "B")

g.add\_edge("A", "C")

g.add\_edge("B", "D")

g.add\_edge("C", "D")

g.display()

## ****2. Graph Traversals****

### ****2.1 Breadth-First Search (BFS)****

* Explores all **neighboring nodes first**, before moving deeper.
* Uses a **queue (FIFO)** for traversal.
* **Time Complexity:** O(V + E).

#### ****Example:****

A

/ \

B C

\ /

D

**BFS Order (Starting from A):** A → B → C → D

#### ****Implementation****

from collections import deque

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])

# Example:

graph = {

"A": ["B", "C"],

"B": ["A", "D"],

"C": ["A", "D"],

"D": ["B", "C"]

}

bfs(graph, "A") # Output: A B C D

### ****2.2 Depth-First Search (DFS)****

* Explores as **deep as possible** before backtracking.
* Uses a **stack (LIFO)** (or recursion) for traversal.
* **Time Complexity:** O(V + E).

#### ****Example:****

**DFS Order (Starting from A):** A → B → D → C

#### ****Implementation****

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

if visited is None:

visited = set()

if node not in visited:

print(node, end=" ")

visited.add(node)

for neighbor in graph[node]:

dfs(graph, neighbor, visited)

# Example:

dfs(graph, "A") # Output: A B D C

## ****3. Detecting Cycles in a Graph****

* **Undirected Graph**: A cycle exists if we revisit a visited node that is **not the parent**.
* **Directed Graph**: A cycle exists if a node is revisited in the **same DFS path**.

#### ****Implementation (DFS for Undirected Graph)****

def has\_cycle(graph, node, visited, parent):

visited.add(node)

for neighbor in graph[node]:

if neighbor not in visited:

if has\_cycle(graph, neighbor, visited, node):

return True

elif neighbor != parent:

return True

return False

# Example:

visited\_set = set()

print(has\_cycle(graph, "A", visited\_set, None)) # Output: True (if a cycle exists)

## ****4. Shortest Path Algorithms****

### ****4.1 Dijkstra’s Algorithm****

* Finds the **shortest path** from a source to all vertices in a graph.
* Uses a **priority queue (min heap)**.
* **Time Complexity**: O((V + E) log V).

#### ****Implementation****

import heapq

def dijkstra(graph, start):

heap = [(0, start)]

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

distances[start] = 0

while heap:

current\_distance, current\_node = heapq.heappop(heap)

for neighbor, weight in graph[current\_node]:

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(heap, (distance, neighbor))

return distances

# Example:

weighted\_graph = {

"A": [("B", 1), ("C", 4)],

"B": [("A", 1), ("D", 2)],

"C": [("A", 4), ("D", 3)],

"D": [("B", 2), ("C", 3)]

}

print(dijkstra(weighted\_graph, "A")) # Output: {'A': 0, 'B': 1, 'C': 4, 'D': 3}

### ****4.2 Bellman-Ford Algorithm****

* Finds the **shortest path** even with **negative weight edges**.
* Runs in **O(V × E)** time.

#### ****Implementation****

def bellman\_ford(graph, start):

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

distances[start] = 0

for \_ in range(len(graph) - 1):

for node in graph:

for neighbor, weight in graph[node]:

if distances[node] + weight < distances[neighbor]:

distances[neighbor] = distances[node] + weight

return distances

print(bellman\_ford(weighted\_graph, "A")) # Output: {'A': 0, 'B': 1, 'C': 4, 'D': 3}

## ****Summary****

✅ Graph Representations → Adjacency Matrix (O(V²)) & Adjacency List (O(V + E)).  
✅ Graph Traversals → BFS (O(V + E)) & DFS (O(V + E)).  
✅ Cycle Detection → DFS for Undirected & Directed Graphs.  
✅ Shortest Path Algorithms → Dijkstra (O((V + E) log V)) & Bellman-Ford (O(V × E)).  
  
  
Excellent question —  
let me **represent this visually with an example**.

Let's say we have this graph with 5 vertices (0–4) and edges like this:

0 - 1

| |

v v

2 3

|

v

4

This forms following adjacency matrix:

self.matrix = [

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

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

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

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

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

]

**🔹 How BFS with shortest\_path(0, 4) works 🔹:**

Let's see the algorithm’s process:

✅ We start from **0** with path [0].

✅ We visit its neighbors (1, 2).

Queue after adding:

* (1, [0, 1])
* (2, [0, 2])

✅ We **pop (1, [0, 1])**:

* neighbors of 1: (0, 3)
* 0 is already visited
* 3 is not, we add (3, [0, 1, 3])

Queue:

* (2, [0, 2])
* (3, [0, 1, 3])

✅ We **pop (2, [0, 2])**:

* neighbors of 2: (0, 4)
* 0 is visited
* 4 is not, we add (4, [0, 2, 4])

Queue:

* (3, [0, 1, 3])
* (4, [0, 2, 4])

✅ We **pop (3, [0, 1, 3])**:

* neighbors of 3: (1)
* 1 is visited — we ignore.

✅ We **pop (4, [0, 2, 4])**:

* we've reached the destination (4)

✅ We return the path: [0, 2, 4]

**🔹 Summary 🔹:**

✅ The algorithm used **BFS with a queue to explore the graph layer by layer**.  
✅ It kept track of the path while visiting each node.  
✅ The first path to reach the destination is the **shortest path in an unweighted graph**.

✨ If you'd like, I can:

* draw a graph picture
* show pseudocode
* or even produce code with comments.  
  ✅ Just let me know! 🌟

Let's go through these two methods **step by step**.

**🔹 1️⃣ DFS Method 🔹:**

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

"""Performs Depth-First Search (DFS) traversal."""

if visited is None:

visited = set()

✅ dfs performs **depth-first search** starting from a given node.  
✅ visited is a set to track which vertices we've already visited.  
✅ If visited is not provided, we create a new set.

if start in visited:

return

✅ If we've already visited this node, we simply **return** and do not process it again (this prevents infinite loops in case of cycles).

visited.add(start)

print(start, end=' ') # Prints the current node

✅ We **mark the current node as visited** and **print its index**.  
✅ The end=' ' just means we print them with a space instead of a new line.

for neighbor, exists in enumerate(self.matrix[start]):

if exists and neighbor not in visited:

self.dfs(neighbor, visited)

✅ We then **look at its neighbors** in the adjacency matrix.  
✅ If there’s a connection (exists) and we haven't visited that neighbor, we **recursively call DFS** on it.

**🔹 2️⃣ Display Method 🔹:**

def display(self):

"""Displays the adjacency matrix."""

print("\nAdjacency Matrix:")

for row in self.matrix:

print(row)

✅ Prints the adjacency matrix row by row.  
✅ Each row is a list of 0s and 1s, where:

* 1 or a non-zero number means there’s a directed edge from this row’s index to that column’s index.
* 0 means there’s no connection.

**🔹 Summary 🔹:**

✅ **DFS** (depth-first search) is implemented **recursively with a stack-like approach**.  
✅ **Display** simply prints the adjacency matrix in a human-readable format.

✨ If you'd like, I can demonstrate this with:

* An example graph
* The output you’d see after DFS  
  ✅ Just let me know!

Excellent question 👏 —  
let’s go through this **BFS code step by step**.

**🔹 What this method does 🔹:**

def bfs(self, start):

"""Performs Breadth-First Search (BFS) traversal."""

✅ This performs **Breadth-First Search (BFS)** starting from a specified **start node** in the graph.  
✅ BFS is a way to **explore all vertices of a graph layer by layer**.

**🔹 Initialization 🔹:**

visited = set()

queue = [start]

✅ We create:

* **visited** (a set) to track which vertices we've already processed.
* **queue** (a list acting as a FIFO queue) with the starting node in it.

**🔹 Main Loop 🔹:**

print("\nBFS Traversal:")

while queue:

curr = queue.pop(0)

if curr not in visited:

print(curr, end=' ')

visited.add(curr)

for neighbor, exists in enumerate(self.matrix[curr]):

if exists and neighbor not in visited:

queue.append(neighbor)

✅ We continue until the **queue is empty**.

* We **pop** the first element — that's the **current node**.
* If it’s **not visited**:
  + We **print it** (that's the order in which we visit).
  + We **mark it as visited**.
* We then **look at its neighbors** (using self.matrix) and:
  + If there’s a connection (exists) **and the neighbor isn’t visited**, we **enqueue** it.

**🔹 Summary 🔹:**

✅ **BFS** starts from a node and visits its neighbors first, then their neighbors, and then moves forward — **level by level**.

✅ This guarantees:

* We visit **the closest nodes first** before proceeding further.

✨ If you'd like, I can demonstrate this with:

* An example graph
* The order in which the nodes are visited  
  ✅ Just let me know! 🌟

Excellent question 👏 —  
let’s go through this carefully.

**🔹 What this method does 🔹:**

def longest\_path(self, start, end, path=None, visited=None):

"""Finds the longest path using DFS."""

✅ It performs **depth-first search (DFS)** to find **the longest path** from **start** to **end** in the graph.

**🔹 Initial checks 🔹:**

if start >= self.size or end >= self.size:

return []

✅ If the start or the end is invalid (not within range), it directly **returns an empty path**.

**🔹 Initialize path and visited set 🔹:**

if path is None:

path = []

if visited is None:

visited = set()

✅ If we do not provide a path or a set of visited vertices, we create them.  
✅ path maintains the route we've taken so far.  
✅ visited prevents us from visiting the same node multiple times (this avoids infinite loops).

**🔹 Update path and visited 🔹:**

path = path + [start]

visited.add(start)

✅ We **add the current node** to:

* path (so we know which route we've taken).
* visited (so we avoid cycling back).

**🔹 Base case 🔹:**

if start == end:

return path

✅ If we've **reached the destination**, we return the path we've constructed.

**🔹 DFS with backtrack 🔹:**

longest = []

for neighbor, exists in enumerate(self.matrix[start]):

if exists and neighbor not in visited:

new\_path = self.longest\_path(neighbor, end, path, visited)

if len(new\_path) > len(longest):

longest = new\_path

✅ We go through **each neighbor** of the current node.  
✅ If there’s an edge (exists) and we haven't yet visited that neighbor, we recurse with:

* the new path
* the updated set of visited nodes  
  ✅ If we find a path that's **longer than the current longest**, we update longest.

**🔹 Backtrack 🔹:**

visited.remove(start)

return longest

✅ We **remove the current node from visited** — this lets other paths reuse this node in their search.  
✅ We then return the longest path we found.

**🔹 Summary 🔹:**

✅ This algorithm performs **depth-first search with backtracking**.  
✅ It **explores all possible paths from start to end and picks the longest one**.  
✅ It avoids cycles by keeping a set of **visited nodes**.

✨ If you'd like, I can demonstrate this with:

* An example adjacency matrix
* A step-by-step path expansion
* The actual code with debug prints  
  ✅ Just let me know! 🌟