

Name : Vaishali Ramesh Kale

Email id: kalevaishalir16@gmail.com

Task 1: Balanced Binary Tree Check Write a function to check if a given binary tree is balanced. A balanced tree is one where the height of two subtrees of any node never differs by more than one.

Solution:::

To determine if a binary tree is balanced, we need to ensure that for every node in the tree, the height difference between its left and right subtrees is no more than one. We can implement this check by traversing the tree and simultaneously calculating the height of each subtree while checking the balance condition

CODE:::

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class BalancedBinaryTree {
    public boolean isBalanced(TreeNode root) {
        return checkHeight(root) != -1;
    }

    private int checkHeight(TreeNode node) {
        if (node == null) {
            return 0; // An empty tree is height 0 and balanced
        }

        // Check the height of the left subtree
        int leftHeight = checkHeight(node.left);
```

```

    if (leftHeight == -1) {
        return -1; // Not balanced
    }

    // Check the height of the right subtree
    int rightHeight = checkHeight(node.right);
    if (rightHeight == -1) {
        return -1; // Not balanced
    }

    // If the current node is unbalanced, return -1
    if (Math.abs(leftHeight - rightHeight) > 1) {
        return -1;
    }

    // Return the height of the tree rooted at the current node
    return Math.max(leftHeight, rightHeight) + 1;
}

public static void main(String[] args) {
    // Example usage:
    BalancedBinaryTree treeChecker = new BalancedBinaryTree();

    // Creating a balanced tree
    TreeNode root = new TreeNode(1);
    root.left = new TreeNode(2);
    root.right = new TreeNode(3);
    root.left.left = new TreeNode(4);
    root.left.right = new TreeNode(5);

```

```
root.right.right = new TreeNode(6);
```

```
System.out.println(" it is Balanced :"+treeChecker.isBalanced(root)); // Output: true
```

```
// Creating an unbalanced tree
```

```
TreeNode root2 = new TreeNode(1);
```

```
root2.left = new TreeNode(2);
```

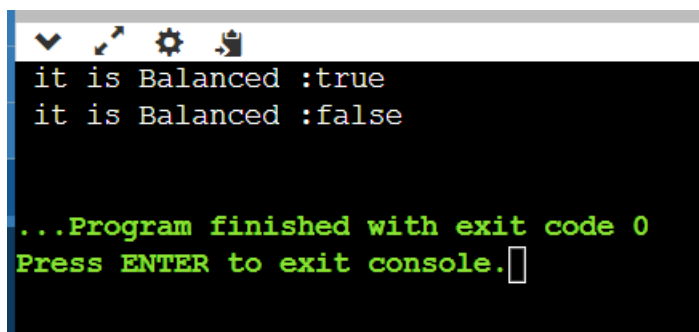
```
root2.left.left = new TreeNode(3);
```

```
System.out.println(" it is Balanced :"+treeChecker.isBalanced(root2)); // Output: false
```

```
}
```

```
}
```

OUTPUT:::



```
it is Balanced :true
it is Balanced :false

...Program finished with exit code 0
Press ENTER to exit console.
```

Explanation

1. **TreeNode Class:** This class represents a node in the binary tree with an integer value (val) and pointers to the left and right children.
2. **BalancedBinaryTree Class:**
 - **isBalanced Method:**
 - This method determines if the tree is balanced by calling the checkHeight method on the root. If checkHeight returns -1, the tree is unbalanced; otherwise, it is balanced.
 - **checkHeight Method:** This is a helper method that recursively calculates the height of the tree and checks the balance condition:
 - If the node is null, it returns 0, indicating the height of an empty tree.

- It recursively checks the heights of the left and right subtrees.
 - If either subtree is unbalanced (indicated by -1), it returns -1.
 - If the height difference between the left and right subtrees is more than 1, it returns -1, indicating the tree is unbalanced.
 - Otherwise, it returns the height of the current node, which is the maximum height of its subtrees plus 1.
3. Main Method:
- This is an example usage of the `BalancedBinaryTree` class. It creates two trees, one balanced and one unbalanced, and checks their balance status by calling the `isBalanced` method.
 - This implementation ensures that each node is visited only once, making the time complexity
 - $O(n)$
 - $O(n)$, where
 - n is the number of nodes in the tree.

Task 2: Trie for Prefix Checking Implement a trie data structure in C# that supports insertion of strings and provides a method to check if a given string is a prefix of any word in the trie.

Solution::::

Explanation:

1. TrieNode Class:

- This class represents a single node in the Trie. Each node contains:
- A Map called `children` that maps characters to their corresponding child nodes.
- A boolean `isEndOfWord` that indicates if the node represents the end of a word.

2. Trie Class:

- This is the main class that contains the root node of the Trie and methods for insertion and prefix checking.
- The `insert` method takes a word, iterates over each character, and inserts it into the Trie. If a node for a character doesn't exist, it creates one.
- The `startsWith` method takes a prefix, iterates over each character, and checks if the prefix exists in the Trie.

3. Usage:

- Insert words into the Trie using the `insert` method.
- Check if a prefix exists using the `startsWith` method.

- This implementation provides a basic and efficient Trie structure for handling prefix checking operations.

CODE:::

```
import java.util.HashMap;
import java.util.Map;

class TrieNode {
    Map<Character, TrieNode> children;
    boolean isEndOfWord;

    public TrieNode() {
        children = new HashMap<>();
        isEndOfWord = false;
    }
}

public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    // Insert a word into the trie
    public void insert(String word) {
        TrieNode current = root;
        for (char ch : word.toCharArray()) {
            current = current.children.computeIfAbsent(ch, c -> new TrieNode());
        }
        current.isEndOfWord = true;
    }
}
```

```

// Check if there is any word in the trie that starts with the given prefix
public boolean startsWith(String prefix) {
    TrieNode current = root;
    for (char ch : prefix.toCharArray()) {
        current = current.children.get(ch);
        if (current == null) {
            return false;
        }
    }
    return true;
}

public static void main(String[] args) {
    Trie trie = new Trie();

    trie.insert("apple");
    trie.insert("app");
    trie.insert("banana");
    trie.insert("band");

    System.out.println(trie.startsWith("app")); // true
    System.out.println(trie.startsWith("ban")); // true
    System.out.println(trie.startsWith("bat")); // false
    System.out.println(trie.startsWith("cat")); // false
}
}

```

OUTPUT:::

```
✓ ↩ ⚙ 🖨
true
true
false
false

...Program finished with exit code 0
Press ENTER to exit console.
```

Task 3: Implementing Heap Operations Code a min-heap in C# with methods for insertion, deletion, and fetching the minimum element. Ensure that the heap property is maintained after each operation.

Solution:::

```
import java.util.ArrayList;
import java.util.List;
import java.util.NoSuchElementException;

public class Heap {

    private List<Integer> heap;

    // Constructor to initialize the heap
    public Heap() {
        this.heap = new ArrayList<>();
    }

    // Method to get the current state of the heap
    public List<Integer> getHeap() {
        return new ArrayList<>(heap);
    }
}
```

```
}
```

```
// Method to get the index of the left child
```

```
private int leftChild(int index) {
```

```
    return (index * 2) + 1;
```

```
}
```

```
// Method to get the index of the right child
```

```
private int rightChild(int index) {
```

```
    return (index * 2) + 2;
```

```
}
```

```
// Method to get the index of the parent
```

```
private int parent(int index) {
```

```
    return (index - 1) / 2;
```

```
}
```

```
// Method to insert a new value into the heap
```

```
public void insert(int value) {
```

```
    heap.add(value); // Add the new value to the end of the heap
```

```
    heapifyUp(heap.size() - 1); // Restore the heap property
```

```
}
```

```
// Method to fetch the minimum element in the heap
```

```
public int getMin() {
```

```
    if (heap.isEmpty()) {
```

```
        throw new NoSuchElementException("Heap is empty");
```

```
    }
```

```
    return heap.get(0); // The minimum element is at the root of the heap
```



```
}
```

```
// Method to delete the minimum element in the heap
```

```
public int deleteMin() {
```

```
    if (heap.isEmpty()) {
```

```
        throw new NoSuchElementException("Heap is empty");
```

```
    }
```

```
    int min = heap.get(0); // Store the minimum element
```

```
    int last = heap.remove(heap.size() - 1); // Remove the last element
```

```
    if (!heap.isEmpty()) {
```

```
        heap.set(0, last); // Move the last element to the root
```

```
        heapifyDown(0); // Restore the heap property
```

```
    }
```

```
    return min; // Return the minimum element
```

```
}
```

```
// Method to restore the heap property by moving the element at index up
```

```
private void heapifyUp(int index) {
```

```
    while (index > 0 && heap.get(index) < heap.get(parent(index))) {
```

```
        swap(index, parent(index)); // Swap the element with its parent
```

```
        index = parent(index); // Move to the parent index
```

```
    }
```

```
}
```

```
// Method to restore the heap property by moving the element at index down
```

```
private void heapifyDown(int index) {
```

```
    int lastIndex = heap.size() - 1;
```

```
    while (index <= lastIndex) {
```

```
        int leftChildIndex = leftChild(index);
```

```

    int rightChildIndex = rightChild(index);

    int smallestChildIndex = index;

    // Find the smallest child
    if (leftChildIndex <= lastIndex && heap.get(leftChildIndex) <
heap.get(smallestChildIndex)) {
        smallestChildIndex = leftChildIndex;
    }

    if (rightChildIndex <= lastIndex && heap.get(rightChildIndex) <
heap.get(smallestChildIndex)) {
        smallestChildIndex = rightChildIndex;
    }

    // If the smallest child is the current index, stop
    if (smallestChildIndex == index) {
        break;
    }

    // Swap the element with the smallest child
    swap(index, smallestChildIndex);

    index = smallestChildIndex; // Move to the smallest child index
}
}

```

// Method to swap two elements in the heap

```

private void swap(int index1, int index2) {
    int temp = heap.get(index1);
    heap.set(index1, heap.get(index2));
    heap.set(index2, temp);
}

```

```

public static void main(String[] args) {

```

```
Heap h = new Heap();

System.out.println("Heap: " + h.getHeap());

// Insert elements into the heap
h.insert(99);
h.insert(66);
h.insert(34);
h.insert(44);
h.insert(50);

System.out.println("Heap after inserts: " + h.getHeap());

// Fetch the minimum element
System.out.println("Min: " + h.getMin());

// Delete the minimum element
System.out.println("Deleted Min: " + h.deleteMin());

// Fetch the minimum element again
System.out.println("Min: " + h.getMin());

// Print the final state of the heap
System.out.println("Heap after deletion: " + h.getHeap());
}
}
```

OUTPUT:::

Output

```
java -cp /tmp/MUSSgp87DH/Heap
Heap: []
Heap after inserts: [34, 44, 66, 99, 50]
Min: 34
Deleted Min: 34
Min: 44
Heap after deletion: [44, 50, 66, 99]

=== Code Execution Successful ===
```

Explanation:

4. Class and Constructor: The constructor initializes the heap as an empty ArrayList.
5. Utility Methods: Methods to get the left child, right child, and parent indices of a given node.
6. Insertion: The insert method adds a new element to the end of the heap and then restores the heap property by calling heapifyUp.
7. Get Minimum: The getMin method returns the root element, which is the minimum element in the heap.
8. Deletion: The deleteMin method removes the root element, replaces it with the last element, and then restores the heap property by calling heapifyDown.
9. Heapify Methods: heapifyUp and heapifyDown methods are used to restore the heap property after insertion and deletion, respectively.
10. Swap Method: The swap method is used to exchange two elements in the heap.
11. Main Method: Demonstrates the usage of the heap by performing insertions, fetching the minimum element, and deleting the minimum element, and then printing the heap state at each step.

Task 4: Graph Edge Addition Validation Given a directed graph, write a function that adds an edge between two nodes and then checks if the graph still has no cycles. If a cycle is created, the edge should not be added.

Solution::::

Explanation:

1. Graph Class:

- Contains a map adjList to store the adjacency list of the graph.
- addVertex method adds a vertex to the graph.
- addEdge method adds a directed edge from from to to. It temporarily adds the edge, checks for a cycle, and if a cycle is detected, removes the edge and returns false. Otherwise, it returns true.
- hasCycle method checks if the graph contains a cycle using DFS.

- hasCycleUtil method is a utility method used by hasCycle to perform DFS and detect cycles.
- printGraph method prints the adjacency list of the graph.

2. Cycle Detection:

- Uses DFS to detect cycles. The recStack set keeps track of nodes in the current recursion stack to detect back edges which indicate a cycle.

3. Usage:

- Add edges using the addEdge method and check the return value to see if the edge was successfully added.
- The graph structure can be printed using printGraph for visualization.
- This implementation ensures that the graph remains acyclic after adding new edges.

CODE:::

```
import java.util.*;

class Graph {

    private final Map<Integer, List<Integer>> adjList;

    public Graph() {
        adjList = new HashMap<>();
    }

    // Add a vertex to the graph
    public void addVertex(int v) {
        adjList.putIfAbsent(v, new ArrayList<>());
    }

    // Add a directed edge to the graph
    public boolean addEdge(int from, int to) {
        addVertex(from);
```

```

addVertex(to);

// Temporarily add the edge
adjList.get(from).add(to);

// Check for cycle
if (hasCycle()) {
    // Remove the edge if a cycle is detected
    adjList.get(from).remove((Integer) to);
    return false;
}

return true;
}

// Check if the graph has a cycle
private boolean hasCycle() {
    Set<Integer> visited = new HashSet<>();
    Set<Integer> recStack = new HashSet<>();

    for (Integer node : adjList.keySet()) {
        if (hasCycleUtil(node, visited, recStack)) {
            return true;
        }
    }

    return false;
}

```

```

// Utility function to detect cycle using DFS
private boolean hasCycleUtil(int node, Set<Integer> visited, Set<Integer> recStack) {
    if (recStack.contains(node)) {
        return true;
    }

    if (visited.contains(node)) {
        return false;
    }

    visited.add(node);
    recStack.add(node);

    List<Integer> children = adjList.get(node);

    if (children != null) {
        for (Integer child : children) {
            if (hasCycleUtil(child, visited, recStack)) {
                return true;
            }
        }
    }

    recStack.remove(node);
    return false;
}

// Print the adjacency list of the graph
public void printGraph() {

```

```

    for (Map.Entry<Integer, List<Integer>> entry : adjList.entrySet()) {
        System.out.print(entry.getKey() + ": ");
        for (Integer v : entry.getValue()) {
            System.out.print(v + " ");
        }
        System.out.println();
    }
}

public static void main(String[] args) {
    Graph graph = new Graph();

    System.out.println("Add edge 1->2: " + graph.addEdge(1, 2));
    System.out.println("Add edge 2->3: " + graph.addEdge(2, 3));
    System.out.println("Add edge 3->4: " + graph.addEdge(3, 4));
    System.out.println("Add edge 4->1 (should fail): " + graph.addEdge(4, 1));
    System.out.println("Add edge 3->1 (should fail): " + graph.addEdge(3, 1));
    System.out.println("Add edge 4->5: " + graph.addEdge(4, 5));

    graph.printGraph();
}
}

```

OUTPUT:::


```
✓ ↗ ⚙ 📄
Add edge 1->2: true
Add edge 2->3: true
Add edge 3->4: true
Add edge 4->1 (should fail): false
Add edge 3->1 (should fail): false
Add edge 4->5: true
1: 2
2: 3
3: 4
4: 5
5:

...Program finished with exit code 0
Press ENTER to exit console.□
```


Task 5: Breadth-First Search (BFS) Implementation For a given undirected graph, implement BFS to traverse the graph starting from a given node and print each node in the order it is visited.

Solution::::

```
import java.util.*;

class Vertex {
    String label;
    Vertex(String label) {
        this.label = label;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Vertex vertex = (Vertex) obj;
        return Objects.equals(label, vertex.label);
    }
}
```

```
}
```

```
@Override
```

```
public int hashCode() {
```

```
    return Objects.hash(label);
```

```
}
```

```
@Override
```

```
public String toString() {
```

```
    return this.label;
```

```
}
```

```
}
```

```
class Graph {
```

```
    private Map<Vertex, List<Vertex>> adjList;
```

```
    Graph() {
```

```
        adjList = new HashMap<>();
```

```
    }
```

```
    public void addVertex(String label) {
```

```
        adjList.putIfAbsent(new Vertex(label), new ArrayList<>());
```

```
    }
```

```
    public void addEdge(String label1, String label2) {
```

```
        Vertex v1 = new Vertex(label1);
```

```
        Vertex v2 = new Vertex(label2);
```

```
        adjList.get(v1).add(v2);
```

```

adjList.get(v2).add(v1); // For undirected graph
}

public void bfs(String startLabel) {
    Vertex startVertex = new Vertex(startLabel);
    if (!adjList.containsKey(startVertex)) {
        System.out.println("Vertex not found in the graph");
        return;
    }

    Set<Vertex> visited = new HashSet<>();
    Queue<Vertex> queue = new LinkedList<>();

    visited.add(startVertex);
    queue.add(startVertex);

    while (!queue.isEmpty()) {
        Vertex current = queue.poll();
        System.out.print(current + " ");

        for (Vertex neighbor : adjList.get(current)) {
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);
                queue.add(neighbor);
            }
        }
    }

    System.out.println();
}

```

```

public static void main(String[] args) {
    Graph graph = new Graph();

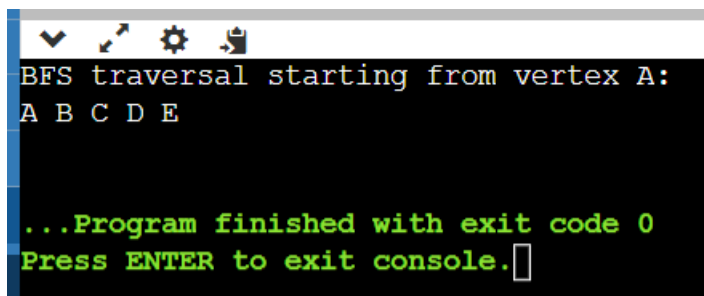
    graph.addVertex("A");
    graph.addVertex("B");
    graph.addVertex("C");
    graph.addVertex("D");
    graph.addVertex("E");
    graph.addVertex("F");

    graph.addEdge("A", "B");
    graph.addEdge("A", "C");
    graph.addEdge("B", "D");
    graph.addEdge("C", "E");
    graph.addEdge("D", "E");

    System.out.println("BFS traversal starting from vertex A:");
    graph.bfs("A");
}
}

```

OUTPUT:....



The screenshot shows a console window with a dark background. At the top, there are several icons: a checkmark, a magnifying glass, a gear, and a trash can. The text in the console is as follows:

```

BFS traversal starting from vertex A:
A B C D E

...Program finished with exit code 0
Press ENTER to exit console.

```

Explanation:

- Vertex Class: This class will represent the nodes in the graph.
- Graph Class: This class will manage the vertices and edges, and provide methods for adding vertices, adding edges, and performing BFS.

1. Vertex Class:

- Represents a vertex with a label.
- Implements equals and hashCode methods to ensure vertices can be compared and used in collections like HashMap and HashSet.
- Implements toString method for easy printing of vertex labels.

2. Graph Class:

- Uses a HashMap to store adjacency lists for each vertex.
- addVertex: Adds a vertex to the graph.
- addEdge: Adds an edge between two vertices (undirected graph).
- bfs: Performs Breadth-First Search starting from a given vertex label.

3. bfs Method:

- Checks if the start vertex exists in the graph.
- Uses a HashSet to keep track of visited vertices.
- Uses a Queue to manage the BFS traversal order.
- Iterates through the graph level by level, printing each vertex as it is visited.

4. main Method:

- Creates a Graph instance.
- Adds vertices and edges to form the graph.
- Calls the bfs method to start BFS traversal from vertex "A".

Task 6: Depth-First Search (DFS) Recursive Write a recursive DFS function for a given undirected graph. The function should visit every node and print it out.

Solution:::

```
import java.util.*;
```

```
// Class to represent a vertex in the graph
```

```
class Vertex {
```

```
    String label; // Label of the vertex
```

```

Vertex(String label) {
    this.label = label;
}

// Override equals method to compare vertices based on their labels
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null || getClass() != obj.getClass()) return false;
    Vertex vertex = (Vertex) obj;
    return Objects.equals(label, vertex.label);
}

// Override hashCode method to ensure vertices can be used in collections like HashMap
and HashSet
@Override
public int hashCode() {
    return Objects.hash(label);
}

// Override toString method to return the label of the vertex
@Override
public String toString() {
    return this.label;
}
}

// Class to represent the graph
class Graph {

```

```
private Map<Vertex, List<Vertex>> adjList; // Adjacency list to store vertices and their edges
```

```
Graph() {  
    adjList = new HashMap<>(); // Initialize the adjacency list  
}
```

```
// Method to add a vertex to the graph
```

```
public void addVertex(String label) {  
    adjList.putIfAbsent(new Vertex(label), new ArrayList<>());  
}
```

```
// Method to add an undirected edge between two vertices
```

```
public void addEdge(String label1, String label2) {  
    Vertex v1 = new Vertex(label1);  
    Vertex v2 = new Vertex(label2);  
  
    adjList.get(v1).add(v2); // Add v2 to the adjacency list of v1  
    adjList.get(v2).add(v1); // Add v1 to the adjacency list of v2 (since the graph is undirected)  
}
```

```
// Recursive DFS method
```

```
private void dfs(Vertex vertex, Set<Vertex> visited) {  
    visited.add(vertex); // Mark the current vertex as visited  
    System.out.print(vertex + " "); // Print the current vertex  
  
    for (Vertex neighbor : adjList.get(vertex)) { // Iterate through all adjacent vertices  
        if (!visited.contains(neighbor)) { // If the neighbor has not been visited  
            dfs(neighbor, visited); // Recursively visit the neighbor  
        }  
    }  
}
```

```
    }  
    }  
}
```

```
// Method to start DFS from a given vertex label
```

```
public void dfs(String startLabel) {  
    Vertex startVertex = new Vertex(startLabel); // Create the starting vertex  
    if (!adjList.containsKey(startVertex)) { // Check if the start vertex exists in the graph  
        System.out.println("Vertex not found in the graph");  
        return;  
    }  
}
```

```
Set<Vertex> visited = new HashSet<>(); // Set to keep track of visited vertices  
dfs(startVertex, visited); // Start the recursive DFS  
System.out.println(); // Print a new line after the traversal  
}
```

```
// Main method to test the DFS implementation
```

```
public static void main(String[] args) {  
    Graph graph = new Graph(); // Create a new graph instance  
  
    // Add vertices to the graph  
    graph.addVertex("A");  
    graph.addVertex("B");  
    graph.addVertex("C");  
    graph.addVertex("D");  
    graph.addVertex("E");  
  
    // Add edges between the vertices
```



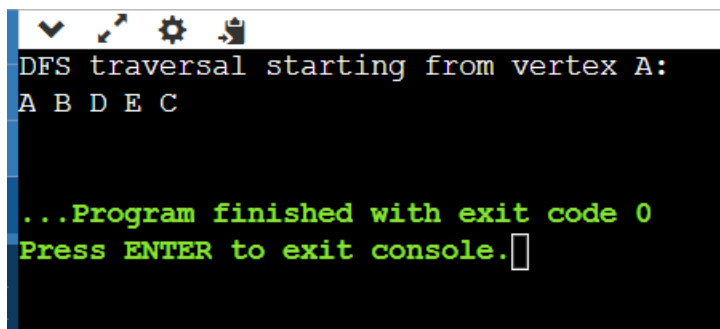
```

graph.addEdge("A", "B");
graph.addEdge("A", "C");
graph.addEdge("B", "D");
graph.addEdge("C", "E");
graph.addEdge("D", "E");

// Perform DFS traversal starting from vertex A
System.out.println("DFS traversal starting from vertex A:");
graph.dfs("A");
}
}

```

OUTPUT:::



```

DFS traversal starting from vertex A:
A B D E C

...Program finished with exit code 0
Press ENTER to exit console.

```

Implementation:

Vertex Class: Represents the nodes in the graph.

Graph Class: Manages vertices, edges, and provides the DFS function

Explanation:

1. Vertex Class:
 - Represents each node in the graph.
 - Implements equals, hashCode, and toString methods for proper comparison and representation.
2. Graph Class:
 - Manages the vertices and edges using an adjacency list.
 - addVertex: Adds a new vertex to the graph.
 - addEdge: Adds an undirected edge between two vertices.
 - dfs (private): Recursive method to perform DFS.
 - Marks the current vertex as visited.

- Prints the current vertex.
- Recursively visits all unvisited neighbors.
- dfs (public): Starts DFS traversal from a given vertex label.
- Checks if the start vertex exists in the graph.
- Initializes a HashSet to track visited vertices.
- Calls the recursive dfs method.

3. main Method:

- Creates a Graph instance.
- Adds vertices and edges to the graph.
- Calls the dfs method to start DFS traversal from vertex "A".