

Day 7 and 8:

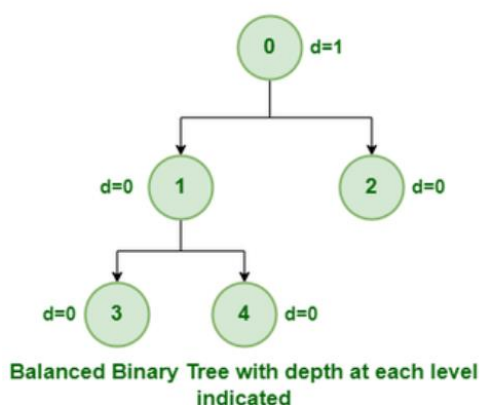
### Task 1: Balanced Binary Tree Check

**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.**

A binary tree is balanced if the height of the tree is  $O(\log n)$  where  $n$  is the number of nodes. For Example, the AVL tree maintains  $O(\log n)$  height by making sure that the difference between the heights of the left and right subtrees is at most 1. Red-Black trees maintain  $O(\log n)$  height by making sure that the number of Black nodes on every root-to-leaf path is the same and that there are no adjacent red nodes. Balanced Binary Search trees are performance-wise good as they provide  $O(\log n)$  time for search, insert and delete.

**A balanced binary tree is a binary tree that follows the 3 conditions:**

- The height of the left and right tree for any node does not differ by more than 1.
- The left subtree of that node is also balanced.
- The right subtree of that node is also balanced.



```
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;  
        }  
  
        int leftHeight = checkHeight(node.left);  
        if (leftHeight == -1) {  
            return -1;  
        }  
  
        int rightHeight = checkHeight(node.right);  
        if (rightHeight == -1) {  
            return -1;  
        }  
  
        if (Math.abs(leftHeight - rightHeight) > 1) {  
            return -1;  
        }  
  
        return Math.max(leftHeight, rightHeight) + 1;  
    }  
}
```

```

public static void main(String[] args) {

    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);

    BalancedBinaryTree treeChecker = new BalancedBinaryTree();
    System.out.println(treeChecker.isBalanced(root));

    TreeNode unbalancedRoot = new TreeNode(1);
    unbalancedRoot.left = new TreeNode(2);
    unbalancedRoot.left.left = new TreeNode(3);

    System.out.println(treeChecker.isBalanced(unbalancedRoot));
}
}

```

**Output:**

**True**

**False**

## **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.**

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.

**using System;**

```
using System.Collections.Generic;
```

```
public class TrieNode
{
    public Dictionary<char, TrieNode> Children { get; private set; }
    public bool IsEndOfWord { get; set; }

    public TrieNode()
    {
        Children = new Dictionary<char, TrieNode>();
        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 currentNode = root;
        foreach (char c in word)
        {
            if (!currentNode.Children.ContainsKey(c))
```

```

    {
        currentNode.Children[c] = new TrieNode();
    }
    currentNode = currentNode.Children[c];
}
currentNode.IsEndOfWord = true;
}

```

// Check if a prefix is present in any word in the trie

```

public bool StartsWith(string prefix)
{
    TrieNode currentNode = root;
    foreach (char c in prefix)
    {
        if (!currentNode.Children.ContainsKey(c))
        {
            return false;
        }
        currentNode = currentNode.Children[c];
    }
    return true;
}

```

// Check if a word is present in the trie

```

public bool Search(string word)
{
    TrieNode currentNode = root;
    foreach (char c in word)
    {

```

```

        if (!currentNode.Children.ContainsKey(c))
        {
            return false;
        }
        currentNode = currentNode.Children[c];
    }
    return currentNode.IsEndOfWord;
}
}

```

**class Program**

```

{
    static void Main(string[] args)
    {
        Trie trie = new Trie();

        trie.Insert("apple");
        trie.Insert("app");

        Console.WriteLine(trie.Search("apple")); // True
        Console.WriteLine(trie.Search("app")); // True
        Console.WriteLine(trie.Search("appl")); // False
        Console.WriteLine(trie.StartsWith("app")); // True
        Console.WriteLine(trie.StartsWith("apl")); // False
    }
}

```

**Explanation**

**TrieNode Class:**

- **Children:** A dictionary that maps each character to the corresponding child node.

- **IsEndOfWord**: A boolean flag to indicate if the node corresponds to the end of a word.

#### Trie Class:

- **root**: The root node of the trie.
- **Insert(string word)**: Inserts a word into the trie by iterating through its characters and creating new nodes if necessary.
- **StartsWith(string prefix)**: Checks if there is any word in the trie that starts with the given prefix.
- **Search(string word)**: Checks if a given word exists in the trie.

#### Main Method:

Demonstrates how to use the Trie class by inserting words and checking for their existence and prefixes.

#### Here's a breakdown of each line in the output:

- **True**
  - **True**
  - **False**
  - **True**
  - **False**
- 
- **trie.Search("apple")**: This returns True because "apple" was inserted into the trie.
  - **trie.Search("app")**: This returns True because "app" was also inserted into the trie.
  - **trie.Search("appl")**: This returns False because "appl" was not inserted into the trie.
  - **trie.StartsWith("app")**: This returns True because both "apple" and "app" start with the prefix "app".
  - **trie.StartsWith("apl")**: This returns False because there is no word in the trie that starts with the prefix "apl".

#### 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.

```
using System;
```

```
using System.Collections.Generic;
```

```
public class MinHeap
{
    private List<int> heap;

    public MinHeap()
    {
        heap = new List<int>();
    }

    // Insert a new element into the heap
    public void Insert(int value)
    {
        heap.Add(value);
        HeapifyUp(heap.Count - 1);
    }

    // Delete the minimum element (the root) from the heap
    public int DeleteMin()
    {
        if (heap.Count == 0)
        {
            throw new InvalidOperationException("Heap is empty");
        }

        int minValue = heap[0];
        heap[0] = heap[heap.Count - 1];
        heap.RemoveAt(heap.Count - 1);
        HeapifyDown(0);
    }
}
```



```

        return minValue;
    }

    // Fetch the minimum element (the root) without deleting it
    public int GetMin()
    {
        if (heap.Count == 0)
        {
            throw new InvalidOperationException("Heap is empty");
        }

        return heap[0];
    }

    // Heapify up to maintain the heap property after insertion
    private void HeapifyUp(int index)
    {
        while (index > 0)
        {
            int parentIndex = (index - 1) / 2;

            if (heap[index] >= heap[parentIndex])
            {
                break;
            }

            Swap(index, parentIndex);
            index = parentIndex;
        }
    }
}

```

// Heapify down to maintain the heap property after deletion

**private void HeapifyDown(int index)**

**{**

**int lastIndex = heap.Count - 1;**

**while (index < lastIndex)**

**{**

**int leftChildIndex = 2 \* index + 1;**

**int rightChildIndex = 2 \* index + 2;**

**int smallestChildIndex = index;**

**if (leftChildIndex <= lastIndex && heap[leftChildIndex] < heap[smallestChildIndex])**

**{**

**smallestChildIndex = leftChildIndex;**

**}**

**if (rightChildIndex <= lastIndex && heap[rightChildIndex] < heap[smallestChildIndex])**

**{**

**smallestChildIndex = rightChildIndex;**

**}**

**if (smallestChildIndex == index)**

**{**

**break;**

**}**

**Swap(index, smallestChildIndex);**

**index = smallestChildIndex;**

**}**

```

    }

    // Swap two elements in the heap
    private void Swap(int index1, int index2)
    {
        int temp = heap[index1];
        heap[index1] = heap[index2];
        heap[index2] = temp;
    }
}

class Program
{
    static void Main(string[] args)
    {
        MinHeap minHeap = new MinHeap();

        // Insert elements
        minHeap.Insert(5);
        minHeap.Insert(3);
        minHeap.Insert(8);
        minHeap.Insert(1);
        minHeap.Insert(2);

        // Get the minimum element
        Console.WriteLine("Min: " + minHeap.GetMin()); // Output: 1
        // Delete the minimum element
        Console.WriteLine("Deleted Min: " + minHeap.DeleteMin());
        Console.WriteLine("New Min: " + minHeap.GetMin());
    }
}

```

```

        // Delete the minimum element

        Console.WriteLine("Deleted Min: " + minHeap.DeleteMin());

        Console.WriteLine("New Min: " + minHeap.GetMin());
    }
}

```

#### Explanation:

##### MinHeap Class:

- **heap**: A list that stores the heap elements.
- **Insert(int value)**: Adds a new element to the heap and ensures the heap property is maintained by calling **HeapifyUp**.
- **DeleteMin()**: Removes and returns the minimum element (the root) from the heap, ensuring the heap property is maintained by calling **HeapifyDown**.
- **GetMin()**: Returns the minimum element (the root) without removing it.
- **HeapifyUp(int index)**: Ensures the heap property is maintained from the given index upwards to the root.
- **HeapifyDown(int index)**: Ensures the heap property is maintained from the given index downwards to the leaves.
- **Swap(int index1, int index2)**: Swaps two elements in the heap.

##### Program Class:

- Demonstrates the use of the **MinHeap** class by inserting elements, fetching the minimum element, and deleting the minimum element.

##### Output:

Min: 1

Deleted Min: 1

New Min: 2

Deleted Min: 2

New Min: 3

##### Insert Elements:

- Elements 5, 3, 8, 1, and 2 are inserted into the min-heap.

##### Get the Minimum Element:

- **minHeap.GetMin()** returns 1 because 1 is the smallest element in the heap.

**Delete the Minimum Element:**

- **minHeap.DeleteMin()** removes 1 (the root) from the heap, and 2 becomes the new root. The heap property is restored by **HeapifyDown**.

**Get the New Minimum Element:**

**minHeap.GetMin()** now returns 2 because 2 is the new smallest element in the heap.

**Delete the Minimum Element Again:**

- **minHeap.DeleteMin()** removes 2 (the root) from the heap, and 3 becomes the new root. The heap property is restored by **HeapifyDown**.

**Get the New Minimum Element Again:**

- **minHeap.GetMin()** now returns 3 because 3 is the new smallest element in the heap.

#### **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.**

Java implementation of a function that adds an edge to a directed graph and checks for cycles. If a cycle is created by adding the edge, the edge is not added.

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

```
public class DirectedGraph {
```

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

```
    public DirectedGraph() {
```

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

```
    }
```

```
    // Add a node to the graph
```

```
    public void addNode(int node) {
```

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

```
    }
```

```
    // Add an edge to the graph and check for cycles
```

```
    public boolean addEdge(int from, int to) {
```

```

addNode(from);
addNode(to);

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

// Helper method to check if the graph has a cycle
private boolean hasCycle() {
    Set<Integer> visited = new HashSet<>();
    Set<Integer> recursionStack = new HashSet<>();

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

    return false;
}

// DFS based utility method to detect cycle

```

```
private boolean hasCycleUtil(int node, Set<Integer> visited, Set<Integer> recursionStack)
{
    if (recursionStack.contains(node)) {
        return true;
    }
    if (visited.contains(node)) {
        return false;
    }

    visited.add(node);
    recursionStack.add(node);

    List<Integer> neighbors = adjList.get(node);
    if (neighbors != null) {
        for (Integer neighbor : neighbors) {
            if (hasCycleUtil(neighbor, visited, recursionStack)) {
                return true;
            }
        }
    }

    recursionStack.remove(node);
    return false;
}
```

```
public static void main(String[] args) {
    DirectedGraph graph = new DirectedGraph();
    System.out.println(graph.addEdge(1, 2));
    System.out.println(graph.addEdge(2, 3));
    System.out.println(graph.addEdge(3, 4));
}
```

```

        System.out.println(graph.addEdge(4, 2));
        // Print the adjacency list to verify edges
        System.out.println(graph.adjList);
    }
}

```

### Explanation:

#### Graph Representation:

- We use an adjacency list (`adjList`) to represent the directed graph.
- Each node maps to a list of its neighbors.

#### Adding Nodes and Edges:

- The **`addNode`** method ensures that a node is added to the graph if it does not already exist.
- The **`addEdge`** method tries to add an edge from `from` to `to` and checks for cycles using DFS. If a cycle is detected, the edge is removed and the method returns false. Otherwise, it returns true.

#### Cycle Detection:

- The **`hasCycle`** method iterates over all nodes to check for cycles using a helper method **`hasCycleUtil`**.
- The **`hasCycleUtil`** method performs a DFS to detect cycles. It uses two sets: `visited` to keep track of all visited nodes, and **`recursionStack`** to keep track of the nodes in the current DFS path.
- If a node is found in the **`recursionStack`**, a cycle is detected.

#### Main Method:

The main method demonstrates adding edges and prints the results. It also prints the adjacency list to verify the graph structure.

#### output

true

true

true

false

{1=[2], 2=[3], 3=[4]}



### **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.**

Breadth-First Search (BFS) for traversing an undirected graph starting from a given node. The implementation includes a graph class that uses an adjacency list to represent the graph, and a BFS function that traverses the graph and prints each node in the order it is visited.

```
import java.util.*;

public class UndirectedGraph {

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

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

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

    // Add an undirected edge to the graph
    public void addEdge(int node1, int node2) {
        addNode(node1);
        addNode(node2);
        adjList.get(node1).add(node2);
        adjList.get(node2).add(node1);
    }

    // BFS traversal starting from a given node
    public void bfs(int startNode) {
        Set<Integer> visited = new HashSet<>();
        Queue<Integer> queue = new LinkedList<>();
```

```

visited.add(startNode);
queue.add(startNode);

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

    List<Integer> neighbors = adjList.get(currentNode);
    if (neighbors != null) {
        for (Integer neighbor : neighbors) {
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);
                queue.add(neighbor);
            }
        }
    }
}

public static void main(String[] args) {
    UndirectedGraph graph = new UndirectedGraph();
    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);
    graph.addEdge(3, 4);
    graph.addEdge(4, 5);

    System.out.println("BFS traversal starting from node 1:");
}

```

```
graph.bfs(1);  
}  
}
```

#### Explanation:

##### Graph Representation:

- The graph is represented using an adjacency list (**adjList**), where each node maps to a list of its neighbors.

##### Adding Nodes and Edges:

- **addNode(int node)**: Adds a node to the graph if it does not already exist.
- **addEdge(int node1, int node2)**: Adds an undirected edge between node1 and node2. It ensures both nodes are present in the graph and then adds each node to the other's adjacency list.

##### BFS Traversal:

- **bfs(int startNode)**: Performs BFS traversal starting from **startNode**.
- Uses a set **visited** to keep track of visited nodes.
- Uses a queue **queue** to manage the nodes to be visited next.
- Visits each node in the order they are dequeued, printing each node as it is visited.
- Adds each unvisited **neighbor** of the current node to the queue and marks them as visited.

##### Main Method:

- The main method demonstrates adding edges to the graph and performing a BFS traversal starting from node 1.
- The expected output of the traversal is printed, showing the order in which the nodes are visited.

##### Output:

##### BFS traversal starting from node 1:

1 2 3 4 5

### 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.

Java implementation of Depth-First Search (DFS) for an undirected graph. The implementation includes a graph class using an adjacency list and a recursive DFS function that visits every node and prints it.

```
import java.util.*;

public class UndirectedGraph {

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

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

    public void addNode(int node) {
        adjList.putIfAbsent(node, new ArrayList<>());
    }

    public void addEdge(int node1, int node2) {
        addNode(node1);
        addNode(node2);
        adjList.get(node1).add(node2);
        adjList.get(node2).add(node1);
    }

    public void dfs(int startNode) {
        Set<Integer> visited = new HashSet<>();
        dfsUtil(startNode, visited);
    }
}
```

```

// Recursive utility function for DFS traversal
private void dfsUtil(int node, Set<Integer> visited) {
    // Mark the current node as visited and print it
    visited.add(node);
    System.out.print(node + " ");

    // Recur for all the vertices adjacent to this vertex
    List<Integer> neighbors = adjList.get(node);
    if (neighbors != null) {
        for (Integer neighbor : neighbors) {
            if (!visited.contains(neighbor)) {
                dfsUtil(neighbor, visited);
            }
        }
    }
}

public static void main(String[] args) {
    UndirectedGraph graph = new UndirectedGraph();
    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);
    graph.addEdge(3, 4);
    graph.addEdge(4, 5);

    System.out.println("DFS traversal starting from node 1:");
    graph.dfs(1);
}
}

```

### Explanation:

#### Graph Representation:

- The graph is represented using an adjacency list (`adjList`), where each node maps to a list of its neighbors.

#### Adding Nodes and Edges:

- `addNode(int node)`: Adds a node to the graph if it does not already exist.
- `addEdge(int node1, int node2)`: Adds an undirected edge between `node1` and `node2`. It ensures both nodes are present in the graph and then adds each node to the other's adjacency list.

#### DFS Traversal:

- **`dfs(int startNode)`**: Initiates DFS traversal starting from `startNode`.
- Uses a set `visited` to keep track of visited nodes.
- Calls the recursive utility function **`dfsUtil`** to perform the traversal.
- **`dfsUtil(int node, Set<Integer> visited)`**: Recursively visits nodes.
- Marks the current node as visited and prints it.
- Recursively visits all unvisited **neighbors** of the current node.

#### Main Method:

- The main method demonstrates adding edges to the graph and performing a DFS traversal starting from node 1.
- The expected output of the traversal is printed, showing the order in which the nodes are visited.

#### Output:

##### DFS traversal starting from node 1:

**1 2 4 5 3**

#### DFS Traversal Path:

- The traversal starts at node 1.
- From node 1, it visits node 2.
- From node 2, it moves to node 4.
- From node 4, it goes to node 5.
- After visiting node 5, it backtracks to node 4 and then moves to node 3.