Day 7 and 8:

Task1: Balanced Binary tree check:

Write a function to check if a given binary tree is balanced. A balance binary tree is one whose the height of two sub tree of any node never differ by more then one.

Implementation of Heap Data Structure:-

The following code shows the implementation of a **max-heap**.

Let's understand the **maxHeapify** function in detail:-

maxHeapify is the function responsible for restoring the property of the Max Heap. It arranges the node **i**, and its subtrees accordingly so that the heap property is maintained.

1. Suppose we are given an array, **arr[]** representing the complete binary tree. The left and the right child of **i**th node are in indices **2*i+1** and **2*i+2**.
2. We set the index of the current element, **i**, as the 'MAXIMUM'.
3. If **arr[2 * i + 1] > arr[i]**, i.e., the left child is larger than the current value, it is set as 'MAXIMUM'.
4. Similarly if **arr[2 * i + 2] > arr[i]**, i.e., the right child is larger than the current value, it is set as 'MAXIMUM'.
5. Swap the 'MAXIMUM' with the current element.
6. Repeat steps **2 to 5** till the property of the heap is restored.

It is the process to rearrange the elements to maintain the property of heap data structure. It is done when a certain node creates an imbalance in the heap due to some operations on that node. It takes **O(log N)** to balance the tree.

- For **max-heap,** it **balances** in such a way that the maximum element is the root of that binary tree and
- For **min-heap,** it balances in such a way that the minimum element is the root of that binary tree.

Insertion:

- If we insert a new element into the heap since we are adding a new element into the heap so it will distort the properties of the heap so we need to perform the **heapify** operation so that it maintains the property of the heap.

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.

Graph Class:

- numVertices: Number of vertices in the graph.
- adjacencyList: An array of linked lists where each list represents the adjacent vertices of a vertex.

Constructor:

- Initializes the adjacencyList array with the given number of vertices.

addEdge Method:

- Adds an edge between vertices v and w in the graph. Since the graph is undirected, it adds w to v's adjacency list and v to w's adjacency list.

BFS Method:

- Takes a starting node as input and performs BFS traversal from that node.
- Uses a boolean array visited to track visited nodes.
- Uses a queue to keep track of the nodes to be visited.
- Marks the starting node as visited, enqueues it, and then processes nodes in the queue by marking their unvisited adjacent nodes as visited and enqueuing them.
- Prints each node as it is dequeued from the queue.

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

 DFS Method:

- Takes a starting node as input and performs DFS traversal from that node.
- Uses a boolean array visited to track visited nodes.
- Calls a helper method DFSUtil to recursively visit nodes.

 DFSUtil Method:

- A recursive method that visits a node, marks it as visited, and then recursively visits all its unvisited adjacent nodes.
- Prints each node as it is visited.

 Main Method:

- Creates an instance of Graph with 6 vertices.
- Adds edges to the graph to form a specific structure.
- Calls the BFS and DFS methods starting from node 0 and prints the order in which nodes are visited.

Task 2: Trie for Prefix Checking

Implement a trie data structure in java that supports insertion of strings and provides a method to check if a given string is a prefix of any word in the trie.

A trie (pronounced "try") is a tree-like data structure used to store a dynamic set of strings, where the keys are usually strings. It is commonly used for tasks like autocomplete and spell-checking. To implement a trie in Java, we need to define two classes:

1. **TrieNode**: Represents a node in the trie. Each node can have multiple child nodes.
2. **Trie**: Contains methods to insert strings and check if a given string is a prefix of any word in the trie.

Task 3: Implementing Heap Operations:

Code a min-heap in java with methods for insertion, deletion, and fetching the minimum element. Ensure that the heap property is maintained after each operation

Explanation:

- **MinHeap Class:**
  - **Fields**: heap is the array to store the heap elements, size is the current number of elements in the heap, and capacity is the maximum capacity of the heap.
  - **Constructor**: Initializes the heap with the given capacity.
  - **insert Method**: Adds a new element to the heap at the end and then performs heapify-up to restore the min-heap property.
  - **deleteMin Method**: Removes and returns the minimum element from the heap (the root). It replaces the root with the last element and performs heapify-down to maintain the min-heap property.
  - **heapifyUp Method**: Restores the min-heap property by bubbling up the element that was just inserted.
  - **heapifyDown Method**: Restores the min-heap property by bubbling down the root element after deletion or replacement.
  - **getMin Method**: Returns the minimum element from the heap (the root) without removing it.
  - **printHeap Method**: Utility method to print the current state of the heap array (useful for testing).
- **main Method**: Demonstrates usage of the MinHeap class by inserting elements, printing the heap, fetching the minimum element, and deleting the minimum element.

```
import java.util.Arrays;

public class MinHeap {

    private int[] heap;
    private int size;
    private int capacity;

    public MinHeap(int capacity) {
```

```
        this.capacity = capacity;
        this.size = 0;
        this.heap = new int[capacity];
    }

    // Insert a new element into the heap
    public void insert(int element) {
        if (size == capacity) {
            System.out.println("Heap overflow, cannot insert more elements.");
            return;
        }

        // Insert the new element at the end of the heap
        heap[size] = element;
        size++;

        // Perform up-heap bubbling (heapify-up)
        heapifyUp(size - 1);
    }

    // Restore heap property upwards
    private void heapifyUp(int index) {
        int parentIndex = (index - 1) / 2;

        while (index > 0 && heap[parentIndex] > heap[index]) {
            // Swap parent and current node
            int temp = heap[parentIndex];
```

```
        heap[parentIndex] = heap[index];
        heap[index] = temp;

        index = parentIndex;
        parentIndex = (index - 1) / 2;
    }
}

// Delete the minimum element from the heap
public int deleteMin() {
    if (size == 0) {
        System.out.println("Heap underflow, cannot delete from an empty heap.");
        return -1;
    }

    int min = heap[0];

    // Replace the root with the last element
    heap[0] = heap[size - 1];
    size--;

    // Perform down-heap bubbling (heapify-down)
    heapifyDown(0);

    return min;
}
```

```java
// Restore heap property downwards
private void heapifyDown(int index) {
   int leftChildIndex = 2 * index + 1;
   int rightChildIndex = 2 * index + 2;
   int smallest = index;

   // Find the smallest element among current node and its children
   if (leftChildIndex < size && heap[leftChildIndex] < heap[smallest]) {
      smallest = leftChildIndex;
   }
   if (rightChildIndex < size && heap[rightChildIndex] < heap[smallest]) {
      smallest = rightChildIndex;
   }

   // If smallest is not the current node, swap and continue heapifying
   if (smallest != index) {
      int temp = heap[index];
      heap[index] = heap[smallest];
      heap[smallest] = temp;
      heapifyDown(smallest);
   }
}

// Return the minimum element from the heap (without removing)
public int getMin() {
   if (size == 0) {
      System.out.println("Heap is empty.");
```

```java
        return -1; // or throw an exception
    }
    return heap[0];
}

// Utility method to print the heap array (for testing)
public void printHeap() {
    System.out.println("Heap array: " + Arrays.toString(heap));
}

public static void main(String[] args) {
    MinHeap minHeap = new MinHeap(10);

    minHeap.insert(3);
    minHeap.insert(2);
    minHeap.insert(1);
    minHeap.insert(7);
    minHeap.insert(8);
    minHeap.insert(4);

    minHeap.printHeap(); // Should print [1, 2, 3, 7, 8, 4, 0, 0, 0, 0]

    System.out.println("Minimum element: " + minHeap.getMin()); // Should print 1

    minHeap.deleteMin();
    minHeap.printHeap(); // Should print [2, 4, 3, 7, 8, 0, 0, 0, 0, 0]
```

```java
        System.out.println("Minimum element after deletion: " + minHeap.getMin()); // Should print 2
    }
}
```

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.

To implement Breadth-First Search (BFS) for traversing an undirected graph starting from a given node and printing each node in the order it is visited, we need to:

1. **Represent the Graph**: Use an adjacency list to store the graph.
2. **Use a Queue**: Utilize a queue to manage the order of exploration (nodes to visit).
3. **Track Visited Nodes**: Maintain a boolean array to keep track of visited nodes to avoid revisiting and infinite loops.

BFS Method:

- Performs BFS traversal starting from the given source vertex.
- Initializes a boolean array visited to track visited vertices.
- Uses a Queue to manage the order of traversal.
- Enqueues the source vertex, marks it as visited, and continues to dequeue and process each vertex.
- For each dequeued vertex, it iterates through its adjacency list:
    - If an adjacent vertex has not been visited, marks it as visited and enqueues it.

```java
import java.util.*;

public class GraphTraversal {

    private int V; // Number of vertices
    private List<List<Integer>> adj; // Adjacency list representation of the graph
```

```
public GraphTraversal(int V) {
    this.V = V;
    adj = new ArrayList<>(V);
    for (int i = 0; i < V; i++) {
        adj.add(new ArrayList<>());
    }
}

// Function to add an edge in an undirected graph
public void addEdge(int u, int v) {
    adj.get(u).add(v);
    adj.get(v).add(u);
}

// Function to perform BFS traversal from a given source vertex
public void BFS(int source) {
    // Array to keep track of visited vertices
    boolean[] visited = new boolean[V];

    // Queue for BFS
    Queue<Integer> queue = new LinkedList<>();

    // Mark the current node as visited and enqueue it
    visited[source] = true;
    queue.offer(source);
```

```java
    while (!queue.isEmpty()) {
        // Dequeue a vertex from queue and print it
        int current = queue.poll();
        System.out.print(current + " ");

        // Get all adjacent vertices of the dequeued vertex current
        // If an adjacent vertex has not been visited, then mark it
        // visited and enqueue it
        for (int neighbor : adj.get(current)) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                queue.offer(neighbor);
            }
        }
    }
}

public static void main(String[] args) {
    // Create a graph
    int V = 6; // Number of vertices
    GraphTraversal graph = new GraphTraversal(V);

    // Add edges
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(1, 4);
```

```
        graph.addEdge(2, 4);
        graph.addEdge(3, 5);
        graph.addEdge(4, 5);

        // Perform BFS traversal starting from vertex 0
        System.out.println("BFS traversal starting from vertex 0:");
        graph.BFS(0); // Output: 0 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.

To implement Depth-First Search (DFS) recursively for an undirected graph and print out each visited node, we can follow these steps:

1. **Graph Representation**: Use an adjacency list to represent the graph.
2. **Visited Array**: Use a boolean array to keep track of visited nodes to avoid cycles and revisiting nodes.
3. **Recursive DFS Function**: Implement a recursive function that:
    - Marks the current node as visited.
    - Prints the current node.
    - Recursively visits all adjacent nodes that have not been visited yet.

Here's how you can implement this in Java: