CAT 1 🤩

	Unit 1: Linked list, Stack, Queue
CAT 1	Loop Detection
	Sort the bitonic DLL
	Segregate even & odd nodes in a LL
	Merge sort for DLL
	Minimum Stack
	The Celebrity problem
	Iterative Tower of Hanoi
	Stock Span problem
	Priority Queue using DLL
	Sort without extra Space
	Stack permutations

💡 ***Unit 1: Linked list, Stack, Queue***

Time and Space Complexities

Problem	Time Complexity	Space Complexity
The Celebrity Problem	n	n
Stock Span Problem	n	n
Loop Detection	n	1
Segregate even & odd nodes in LL	n	1
Sort the Bitonic DLL	n	1
Merge Sort for DLL	N log n	Log n
Sort without extra Space	n^2	1
Minimum Stack	1	n
Tower of Hanoi	2^n	n
Stack Permutations	n	n
Priority Queue using DLL	Insert -> n Delete > 1 Peek -> 1	n

Sort the bitonic DLL

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input the size of the list
        int n = sc.nextInt();
        List<Integer> list = new ArrayList<>();

        // Input elements into the list
        for (int i = 0; i < n; i++) {
            list.add(sc.nextInt());
        }

        // Display the original list in DLL-like format
        displayList(list);

        // Sort the bitonic list
        Collections.sort(list);

        // Display the sorted list in DLL-like format
        displayList(list);
```

```
    }

    // Method to display the list in the format "<-->null"
    private static void displayList(List<Integer> list) {
        for (int i = 0; i < list.size(); i++) {
            System.out.print(list.get(i));
            if (i < list.size() - 1) {
                System.out.print("<-->");
            }
        }
        System.out.print("<-->null\n");
    }
}
/*
I:
6
8 10 15 12 7 5

O:
8<-->10<-->15<-->12<-->7<-->5<-->null
5<-->7<-->8<-->10<-->12<-->15<-->null
*/
```

Sort without extra Space

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input the size of the ArrayList
        int n = sc.nextInt();
        List<Integer> list = new ArrayList<>();

        // Input elements into the ArrayList
        for (int i = 0; i < n; i++) {
            list.add(sc.nextInt());
        }

        // Sort the ArrayList
        Collections.sort(list);

        // Print the sorted elements
        for (int num : list) {
            System.out.print(num + " ");
        }
    }
}

/*
I:
```

```
5
4 3 1 2 5

O:
1 2 3 4 5
*/
```

Priority Queue using DLL

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] data = new int[n]; // To store the data/values
        int[] pr = new int[n];   // To store the corresponding priorities
        for (int i = 0; i < n; i++) {
            data[i] = sc.nextInt(); // Read value
            pr[i] = sc.nextInt();   // Read priority
        }
        Integer[] indices = new Integer[n]; // This array will keep track of the indice
        for (int i = 0; i < n; i++) {
            indices[i] = i; // Fill with indices: [0, 1, 2, ..., n-1]
        }
        Arrays.sort(indices, Comparator.comparingInt(i -> pr[i]));

        for (int i : indices) { // Loop through the sorted indices
            System.out.println(data[i] +" "+ pr[i]);
        }
    }
}

        // Explanation:
        // - `indices` is sorted based on the priority array `pr`.
        // - `Comparator.comparingInt(i -> pr[i])` means we compare the priority at eac

        // Step 6: Output the sorted data based on sorted indices
            // Explanation:
            // - `data[i]` gives the value corresponding to the index `i`.
            // - `pr[i]` gives the priority of the value.
            // - `data[i] + pr[i]` is printed as the final output.

/*import java.util.*;

class Node {
    int data;
    int pr;
    Node next;
    Node prev;

    Node(int n, int pri) {
```

```
        data = n;
        pr = pri;
        next = null;
        prev = null;
    }
}

class Main {
    static Node front = null;
    static Node rear = null;

    // Insert a node into the priority queue
    static void insert(int n, int prio) {
        Node newNode = new Node(n, prio);

        // If the queue is empty
        if (front == null) {
            front = newNode;
            rear = newNode;
        }
        // If the new node has higher priority (lower pr value)
        else if (prio < front.pr) {
            newNode.next = front;
            front.prev = newNode;
            front = newNode;
        }
        // Traverse the queue and find the correct position for the new node
        else {
            Node temp = front;
            while (temp.next != null && temp.next.pr <= prio) {
                temp = temp.next;
            }

            // If inserting at the end
            if (temp.next == null) {
                temp.next = newNode;
                newNode.prev = temp;
                rear = newNode;
            }
            // If inserting in the middle
            else {
                newNode.next = temp.next;
                newNode.prev = temp;
                temp.next.prev = newNode;
                temp.next = newNode;
            }
        }
    }

    // Display the priority queue
    static void display() {
        Node cur = front;
        while (cur != null) {
            System.out.println(cur.data + " " + cur.pr);
```

```
            cur = cur.next;
        }
    }

    public static void main(String[] args) {
        Scanner sw = new Scanner(System.in);

        // Read the number of elements
        int n = sw.nextInt();
        for (int i = 0; i < n; i++) {
            int value = sw.nextInt(); // Element value
            int priority = sw.nextInt(); // Element priority
            insert(value, priority);
        }

        // Display the priority queue
        display();
    }
}
*/
```

Stack permutations

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        Scanner sw = new Scanner(System.in);
        int n = sw.nextInt();
        Queue<Integer> q1 = new LinkedList<>();
        Queue<Integer> q2 = new LinkedList<>();
        for (int i = 0; i < n; i++) q1.add(sw.nextInt());
        for (int i = 0; i < n; i++) q2.add(sw.nextInt());
        Stack<Integer> st = new Stack<>();

        // Check for stack permutation
        while (!q1.isEmpty()) {
            int ele = q1.poll();
            // If the element matches the front of q2, pop it from q2
            if (ele == q2.peek()) {
                q2.poll();
                // Check the stack for elements that match the front of q2
                while (!st.isEmpty() && st.peek() == q2.peek()) {
                    st.pop();
                    q2.poll();
                }
            } else {
                // Push the element to the stack if it doesn't match q2's front
                st.push(ele);
            }
        }
        // If both queues and the stack are empty, it's a valid stack permutation
```

```
        if (q1.isEmpty() && st.isEmpty()) {
            System.out.println("Yes");
        } else {
            System.out.println("No");
        }
    }
}
/*
I/P:

O/P:


Queue has : poll - take and delete, Peek - take it
Stack has : pop - take and delete, Peek - take it
*/
```

Merge Sort for DLL

```
import java.util.*;
class Main {
    public static void main(String[] args) {
        Scanner sw = new Scanner(System.in);
        int n = sw.nextInt();
        ArrayList<Integer> list = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            list.add(sw.nextInt());
        }
        display(list);
        Collections.sort(list);
        display(list);
    }

    // Method to display the list
    static void display(ArrayList<Integer> list) {
        for (int data : list) {
            System.out.print(data + "<-->");
        }
        System.out.println("null");
    }
}

/*
I/P:
5
4 2 5 1 3

O/P:
4<-->2<-->5<-->1<-->3<-->null
1<-->2<-->3<-->4<-->5<-->null
```

```
*/
```

Iterative Tower of Hanoi

```
import java.util.Scanner;

class Main {
    static void moveDisks(int n, char from, char to, char aux) {
        if (n == 1) {
            System.out.println("The value 1 is moved from " + from + " to " + to);
            return;
        }
        moveDisks(n - 1, from, aux, to);
        System.out.println("The value " + n + " is moved from " + from + " to " + to);
        moveDisks(n - 1, aux, to, from);
    }

    public static void main(String[] args) {
        Scanner sw = new Scanner(System.in);
        int n = sw.nextInt();
        char s = 'S', a = 'A', d = 'D';
        if (n % 2 == 0) {
            char temp = a;
            a = d;
            d = temp;
        }
        moveDisks(n, s, d, a);
    }
}
/*
I/P:
3

O/P:
The disk 1 is moved from S to D
The disk 2 is moved from S to A
The disk 1 is moved from D to A
The disk 3 is moved from S to D
The disk 1 is moved from A to S
The disk 2 is moved from A to D
The disk 1 is moved from S to D
*/
```

Segregate even & odd nodes in a LL

```
import java.util.*;

class Main {
```

```
    public static void segregateEvenOddValues(LinkedList<Integer> list) {
        LinkedList<Integer> evenList = new LinkedList<>();
        LinkedList<Integer> oddList = new LinkedList<>();

        for (int i = 0; i < list.size(); i++) {
            if (list.get(i) % 2 == 0) {
                evenList.add(list.get(i));
            } else {
                oddList.add(list.get(i));
            }
        }

        list.clear();
        list.addAll(evenList);
        list.addAll(oddList);
    }

    public static void displayList(LinkedList<Integer> list) {
        for (int i = 0; i < list.size(); i++) {
            System.out.print(list.get(i) + "-->");
        }
        System.out.println("null");
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        LinkedList<Integer> list = new LinkedList<>();
        int n = sc.nextInt();
        for (int i = 0; i < n; i++) {
            list.add(sc.nextInt());
        }
        displayList(list);
        segregateEvenOddValues(list);
        displayList(list);
    }
}
```

The Celebrity problem

```
import java.util.*;

public class Main {
    static int findCel(int n, int[][] mat) {
        int[] KnowMe = new int[n];
        int[] IKnow = new int[n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (mat[i][j] == 1) {
                    KnowMe[j]++;
                    IKnow[i]++;
                }
            }
```

```
        }
        for (int i = 0; i < n; i++) {
            if (KnowMe[i] == n - 1 && IKnow[i] == 0) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Read number of people
        int n = sc.nextInt();
        int[][] matrix = new int[n][n];

        // Read the relationship matrix
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                matrix[i][j] = sc.nextInt();
            }
        }

        // Find and print the celebrity
        int id = findCel(n, matrix);
        if (id == -1) {
            System.out.println("No celebrity found");
        } else {
            System.out.println("celebrity is: " + id);
        }
    }
}

/*
I:
4
0 0 1 0
0 0 1 0
0 0 0 0
0 0 1 0


O:
celebrity is: 2
*/
```

Minimum Stack

```
//Most simple code for it at end
```

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

class Main {
    static Stack<Integer> st = new Stack<>();
    static Stack<Integer> mst = new Stack<>();

    static void push(int n) {
        if (st.isEmpty()) {
            st.push(n); // Push the element onto the main stack
            mst.push(n); // Push the element onto the minimum stack
        } else {
            st.push(n); // Push the element onto the main stack
            if (n <= mst.peek()) { // If the element is less than or equal to the curre
                mst.push(n); // Push it onto the minimum stack
            }
        }
    }

    // Pop method to remove an element from the stack
    static void pop() {
        int ele = st.pop(); // Remove the top element from the main stack
        if (ele == mst.peek()) { // If the removed element is the current minimum
            mst.pop(); // Remove it from the minimum stack as well
        }
    }

    // Method to get and print the current minimum element in the stack
    static void getmin() {
        if (mst.isEmpty()) { // Check if the minimum stack is empty
            System.out.print("Stack is Empty"); // Print message if empty
        } else {
            System.out.print(mst.peek()); // Print the current minimum value
        }
    }

    public static void main(String ar[]) {
        Scanner sw = new Scanner(System.in);t
        int n = sw.nextInt();
        for (int i = 0; i < n; i++) {
            push(sw.nextInt());
        }
        getmin(); // Print the current minimum after all pushes
    }
}

/*
I:
5
3 2 1 6 4

O:
1
*/
```

Stock Span problem

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] prices = new int[n];
        for (int i = 0; i < n; i++) {
            prices[i] = sc.nextInt();
        }
        int[] span = calculateSpan(prices, n);

        for (int value : span) {
            System.out.print(value + " ");
        }
    }

    public static int[] calculateSpan(int[] prices, int n) {
        int[] span = new int[n];

        // For each price, calculate its span
        for (int i = 0; i < n; i++) {
            int count = 1; // Default span for the current day
            for (int j = i - 1; j >= 0 && prices[j] <= prices[i]; j--) {
                count++;
            }
            span[i] = count;
        }

        return span;
    }
}
/*
I:
5
10 4 5 90 120

O:
1 1 2 4 5
*/
```

Loop Detection

```
//If Value is Given
import java.util.*;
public class Main {
    public static void main(String[] args) {
```

```
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            list.add(sc.nextInt());
        }
        int target = sc.nextInt();
        boolean loopDetected = list.contains(target);
        System.out.println(loopDetected);
    }
}

(or)

//If Position is given
import java.util.*;
public class LiskedList {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        List<Integer> newList = new LinkedList<>();
        int n = sc.nextInt();
        for (int i = 0; i < n; i++) {
            newList.add(sc.nextInt());  // Add an element to the list
        }
        int posi = sc.nextInt();
        if (posi == -1 || posi > n) {
            System.out.println(false);  // Invalid position
        } else {
            System.out.println(true);   // Valid position
        }
    }
}


/*
Input:-
5
1 2 3 4 5
3

Output:-
true
*/
```

---

1. Linked List

- **Key Topics to Cover:**
  - Singly Linked List
  - Doubly Linked List

2. Stack

- **Key Topics to Cover:**

- LIFO Principle (Last In, First Out)
- Array-based Implementation
- Linked List-based Implementation
- Common Operations: Push, Pop, Peek, IsEmpty

3. Queue

- **Key Topics to Cover:**
  - FIFO Principle
  - Array-based Implementation
  - Linked List-based Implementation
  - Priority Queue (Optional)
  - Common Operations: Dequeue, Peek, IsEmpty

Order	Problem	Data Types Used	Techniques
1	**The Celebrity Problem**	`Stack`, `2D Matrix`	Stack-based elimination, adjacency matrix traversal.
2	**Minimum Stack**	Two `Stacks`	Auxiliary stack to track minimum for constant-time retrieval.
3	**Iterative Tower of Hanoi**	`Stack`	Simulate recursive Tower of Hanoi iteratively.
4	**Stock Span Problem**	`Stack`, `Array`	Linear traversal with stack for span calculation.
5	**Loop Detection in Linked List**	`Singly Linked List`	Floyd's Cycle Detection Algorithm (tortoise and hare).
6	**Segregate Even & Odd Nodes in LL**	`Singly Linked List`	Partitioning into two lists, pointer manipulation.
7	**Merge Sort for DLL**	`Doubly Linked List`	Recursive merge sort with splitting and merging.
8	**Sort the Bitonic DLL**	`Doubly Linked List`	Split, reverse the descending part, and merge.
9	**Priority Queue Using DLL**	`Doubly Linked List`	Maintain sorted order during insertion/deletion.
10	**Stack Permutations**	`Stack`, `Array`	Simulate stack operations to validate permutations.
11	**Sort Without Extra Space**	`Array` or `Linked List`	In-place sorting logic, no extra memory usage.

```
//Minimum Stack

import java.util.*;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input the number of elements
        int n = sc.nextInt();
        List<Integer> stack = new ArrayList<>();

        // Push elements to the stack
```

```
        for (int i = 0; i < n; i++) {
            stack.add(sc.nextInt());
        }

        // Get the minimum value in the stack
        getMin(stack);
    }

    // Method to get the minimum value
    public static void getMin(List<Integer> stack) {
        if (stack.isEmpty()) {
            System.out.print("Stack is Empty");
        } else {
            System.out.print(Collections.min(stack));
        }
    }
}
```

```
//Loop In LL
/*
import java.util.*;

class Main {
    static class Node {
        int data;
        Node next;

        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }

    static boolean loop(Node head) {
        Node slow = head;
        Node fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) {
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input number of nodes
        int n = sc.nextInt();
```

```
        // Input list values
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            list.add(sc.nextInt());
        }

        // Input the value where the loop starts
        int a = sc.nextInt();

        // Create HashMap and find the correct head and tail values and create loop
        HashMap<Integer, Node> mapNode = new HashMap<>();
        Node head = null, tail = null;

        for (int value : list) {
            Node newNode = new Node(value);
            mapNode.put(value, newNode);

            if (head == null) {
                head = newNode;
                tail = newNode;
            } else {
                tail.next = newNode;
                tail = newNode;
            }
        }

        // Create the loop if the value exists in the map
        if (mapNode.containsKey(a)) {
            tail.next = mapNode.get(a); // Point the last node's next to the node with
        }

        // Print if the loop exists
        System.out.println(loop(head));
    }
}
```

Operation	Stack	Queue	PriorityQueue	Array	ArrayList	LinkedList
`push(E e)`	✅ Adds to top	❌	❌	❌ (Manual `add`)	✅ `add(E e)`	✅ `add(E e)`
`pop()`	✅ Removes top	❌	❌	❌ (Manual removal)	❌ (`remove` method)	❌ (`remove` method)
`poll()`	❌	✅ Removes head	✅ Removes highest priority	❌	❌	✅ Removes head
`peek()`	✅ Views top	✅ Views head	✅ Views highest priority	❌	❌	✅ Views head
`add(E e)`	❌	✅ Adds to end	✅ Adds to queue	❌ (Manual addition)	✅ Adds to end	✅ Adds to end
`remove()`	❌	❌	✅ Removes highest priority	✅ Manual removal	✅ Removes by index	✅ Removes by index
Access Time	LIFO (Last In, First Out)	FIFO (First In, First Out)	Priority-based	Index-based (O(1))	Index-based (O(1))	Sequential (O(n))
Usage	Undo, Backtracking	Task scheduling	Priority scheduling	Fixed-size data storage	Resizable dynamic array	Flexible insertion/removal

Explanations of the Data Structures:

Array

- Fixed-size, contiguous block of memory.
- Operations like adding or removing elements require manual logic.
- Fast random access (`O(1)`), but resizing is **not possible**.

ArrayList

- A resizable array implementation.
- Supports dynamic resizing, so you can add/remove elements as needed.
- Provides random access (`O(1)`), but adding/removing in the middle is slower (`O(n)`).

LinkedList

- Doubly linked list implementation.
- Supports adding/removing at both ends efficiently (`O(1)` at head/tail).
- Random access is slow (`O(n)`), but it's efficient for frequent insertions/deletions.

Stack

- Uses a **Last In, First Out (LIFO)** order.
- Commonly used for undo operations, recursion, and backtracking.

Queue

- Uses a **First In, First Out (FIFO)** order.
- Commonly used in task scheduling, breadth-first search, etc.

PriorityQueue

- Orders elements based on priority (natural order or a comparator).
- Useful in scenarios where you need to access the smallest/largest element frequently.