Linear Data Structures

Arrays and their applications

What is an Array?

- A **collection of elements** stored in **contiguous memory locations**.
- All elements are of the **same data type** (e.g., all integers, all floats).
- Each element is accessed using an **index** (starting from 0).

- Example: int[] numbers = {10, 20, 30};

```
numbers[0]= 10
numbers[1]= 20
numbers[2]= 30
```

Features of Arrays

- Fixed size (declared at the time of creation).
- Allows **random access** using index.
- Efficient for **storing and retrieving** large amounts of data.
- Easy to implement and use in most programming languages.

Common Operations on Arrays

- Insertion
- Deletion
- Traversal (visiting each element)
- Searching (Linear Search, Binary Search)
- Sorting (Bubble, Selection, Merge, etc.)

Types of Arrays

- **1D Array**: Linear list of elements.

  Example: `int[] a = {1, 2, 3, 4};`

- **2D Array**: Matrix or table format.

  Example: `int[][] b = { {1,2}, {3,4} };`

- **Multidimensional Array**: More than 2 dimensions

Applications of Arrays

In Real Life / Everyday Tech:

- **Music playlists or photo galleries** → Lists of items accessed by index.
- **Gaming leaderboards** → Player scores stored and updated.
- **Calendars and schedules** → Days, hours stored in 2D arrays.
- **Sensor data storage** → Temperature readings, heart rates.

In Programming:

- **Storing and processing data** (e.g., exam marks, stock prices)
- **Implementing data structures** like:
  - Stacks
  - Queues
  - Heaps
- **Matrix operations** (used in math and physics)
- **Dynamic Programming** (using arrays for memoization)
- **Graph algorithms** (adjacency matrix representation)

In System Design:

- **Memory management** (paging tables)

- **Image processing** (pixels in a 2D array)
- **Database tables** (rows and columns similar to 2D arrays)

Limitations of Arrays

- Fixed size: Can't grow or shrink dynamically.
- Insertion/deletion is **costly** (requires shifting elements).
- Only stores **same data type** elements.

Array Example Java Code

```
public class ArrayExample {
    public static void main(String[] args) {
        // 1. Declare and initialize an array of 5 integers
        int[] numbers = {10, 20, 30, 40, 50};

        // 2. Display all elements
        System.out.println("Array elements:");
        for (int i = 0; i < numbers.length; i++) {
            System.out.println("Element at index " + i + ": "
+ numbers[i]);
        }

        // 3. Find the sum of all elements
        int sum = 0;
        for (int num : numbers) {
            sum += num;
        }
        System.out.println("Sum of array elements: " + sum);

        // 4. Search for an element
        int key = 30;
```

```
        boolean found = false;
        for (int i = 0; i < numbers.length; i++) {
            if (numbers[i] == key) {
                System.out.println(key + " found at index " +
i);
                found = true;
                break;
            }
        }
        if (!found) {
            System.out.println(key + " not found in the
array");
        }
    }
}
```

Output:

```
Array elements:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 4: 50
Sum of array elements: 150
30 found at index 2
```

Question:

Write a Java program that takes an integer array as input and finds the **maximum element** in the array.

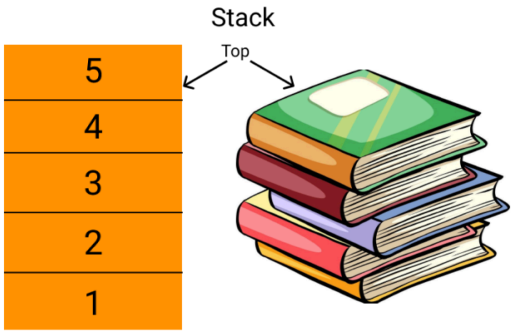
Input: An integer array arr of size n.

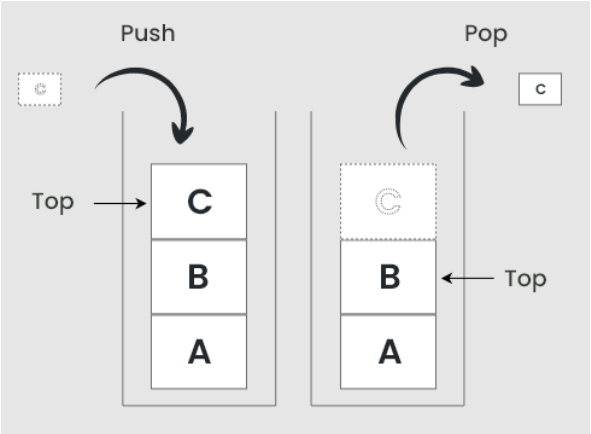
Output: Print the maximum element found in the array.

Input	Output	Explanation
[5, 3, 8, 2, 7]	8	8 is the largest number in the array.
[-1, -5, -3, -4]	-1	-1 is the largest (least negative) number.

Stack

A **stack** is a linear data structure that follows the **LIFO** principle (Last In, First Out). Think of it like a stack of books – the last one placed is the first one removed.

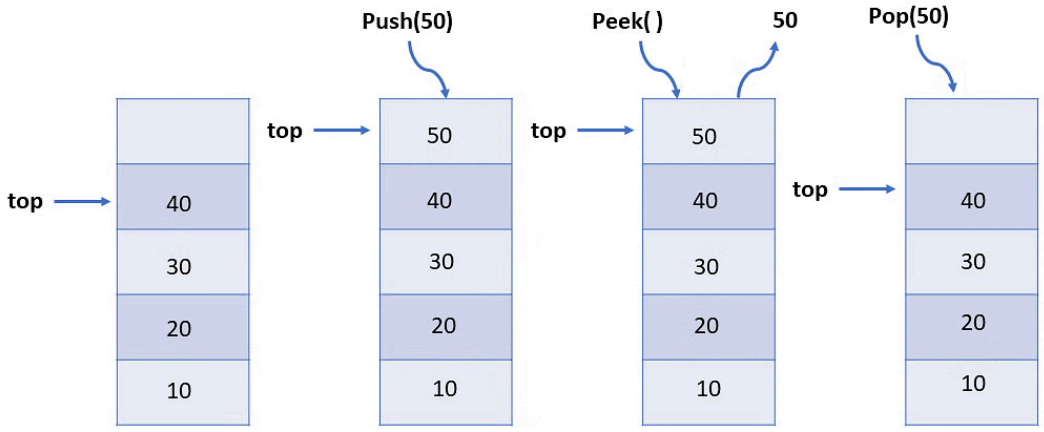




Stack Terminology

- **Top**: The index where insertions and deletions occur.
- **Push**: Operation to insert an element.
- **Pop**: Operation to remove the top element.
- **Peek/Top**: View the top element without removing it.
- **Underflow**: Trying to pop from an empty stack.
- **Overflow**: Trying to push onto a full stack (in fixed-size stacks)

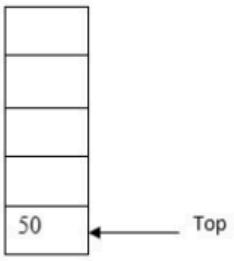
How Stacks Work



a) Consider the following Stack and draw the Stack frames after executing each statement given below.

```
int a = 22, b = 44;
```

i) theStack.push(2);
ii) theStack.push(a);
iii) theStack.push(a + b);
iv) theStack.pop();
v) theStack.push(b);
vi) theStack.push(a –b);



Stack - Java Code

```
public class Stack {
    int[] stack;
    int top;
    int size;

    // Constructor
    public Stack(int size) {
        this.size = size;
```

```
        stack = new int[size];
        top = -1; // Stack is initially empty
    }

    // Push operation
    public void push(int value) {
        if (top == size - 1) {
            System.out.println("Stack Overflow! Cannot push "
+ value);
        } else {
            stack[++top] = value;
            System.out.println(value + " pushed to stack");
        }
    }

    // Pop operation
    public void pop() {
        if (top == -1) {
            System.out.println("Stack Underflow! Cannot pop");
        } else {
            System.out.println(stack[top] + " popped from
stack");
            top--;
        }
    }

    // Peek operation
    public void peek() {
        if (top == -1) {
            System.out.println("Stack is empty");
        } else {
            System.out.println("Top element: " + stack[top]);
        }
    }

    // Display operation
```

```
    public void display() {
        if (top == -1) {
            System.out.println("Stack is empty");
        } else {
            System.out.print("Stack elements: ");
            for (int i = 0; i <= top; i++) {
                System.out.print(stack[i] + " ");
            }
            System.out.println();
        }
    }

    // Main method
    public static void main(String[] args) {
        Stack s = new Stack(5);
        s.push(10);
        s.push(20);
        s.push(30);
        s.display();
        s.peek();
        s.pop();
        s.display();
    }
}
```

Output:

```
10 pushed to stack
20 pushed to stack
30 pushed to stack
Stack elements: 10 20 30
Top element: 30
30 popped from stack
Stack elements: 10 20
```

Applications of Stack

- **Undo/Redo** functionality in software.

- **Function call handling** (call stack in programming).
- **Syntax parsing** (e.g., compilers).
- **Balanced parentheses checking**.
- **Backtracking algorithms** (e.g., maze solving, DFS).
- **Browser history navigation**.

Limitations of Stacks

1. **Restricted Access**
    - Can only access the **top element**.
    - No direct access to elements in the middle or bottom.

2. **Stack Overflow & Underflow**
3. **Not Suitable for All Problems**
4. **No Multi-Directional Traversal**

Question:

You are given a string consisting of different types of brackets:

- Parentheses: `(` and `)`
- Curly braces: `{` and `}`
- Square brackets: `[` and `]`

Write a Java program to determine whether the brackets in the string are balanced. A string is considered balanced if:

1. Every opening bracket has a corresponding closing bracket of the same type.
2. Brackets are closed in the correct order, i.e., inner brackets must be closed before outer brackets.

Your program must use a stack data structure to solve this problem efficiently.

Output:

- Print `"Balanced"` if the input string has properly balanced brackets.

- Print `"Not Balanced"` if the brackets are mismatched or improperly ordered.

Queue

A **queue** is a **linear data structure** that stores elements in a **First In, First Out (FIFO)** order.

That means the **first element added** to the queue will be the **first one to come out**.

It is similar to a real-life queue at a ticket counter or in a line.

Real-Life Example:

Imagine people standing in a queue:

- The first person to join the line is the first person served.
- New people join at the back (rear).
- People are served from the front.

Insertion (enqueue) happens at the **rear**.

Deletion (dequeue) happens at the **front**.

Queue Terminology

- **Front**: Points to the first element.
- **Rear**: Points to the last element.
- **Size**: Number of elements in the queue.
- **Capacity**: Maximum number of elements (for static queues).

Applications of Queues

- CPU scheduling
- Print queue management
- Call center systems
- Data buffering (keyboard input, IO buffers)
- Breadth-First Search (BFS) in Graphs
- Handling requests in web servers

Queue Implementation

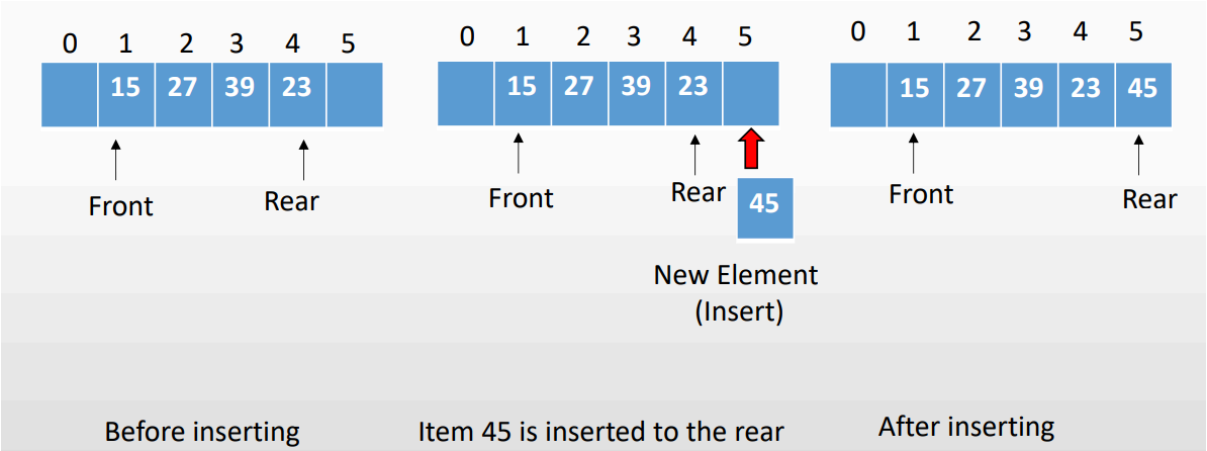
- **Array-based implementation**:
  - Easier to implement
  - Fixed size
- **Linked list-based implementation**:
  - Dynamic memory usage
  - No fixed size limit

Types of Queues

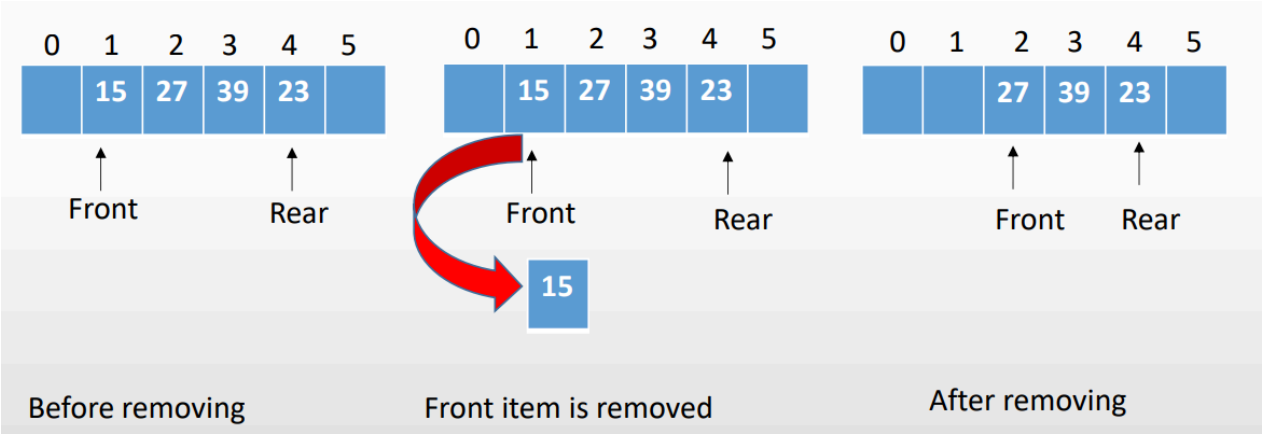
1. Simple Queue

   FIFO, linear structure.

Queue - Insert

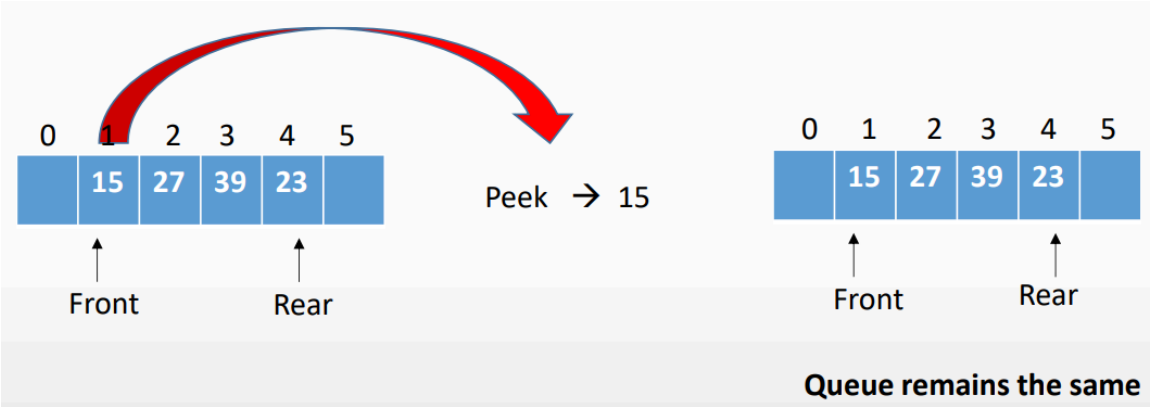


Queue - Remove



Queue - PeekFront

Returns the element at the **front** of the queue **without removing** it.



Question:

A **simple queue** is implemented using an **array of size 5**. Initially, the queue is empty. The following operations are performed in order:

1. Enqueue(10)
2. Enqueue(20)
3. Enqueue(30)
4. Dequeue()
5. Enqueue(40)
6. Enqueue(50)
7. Enqueue(60)

- **Draw an array of size 5** to represent the queue.
- Track and update the `front` **and** `rear` **pointers** (clearly denote the front and rear values for each operation)
- Perform the operations step-by-step.

Answer the following:

a) What will be the **final content** of the queue (in order from front to rear)?

b) What will be the value returned by `peekFront()`?

c) What will happen if you try to **Enqueue(70)** after step 7?
Explain why and what condition this represents.

[Java code for Queue implementation]

```
public class SimpleQueue {
    int[] queue;        // Array to store queue elements
    int front;          // Index of front element
    int rear;           // Index of last element
    int capacity;       // Maximum capacity of the queue
    int size;           // Current number of elements

    // Constructor to initialize the queue
    public SimpleQueue(int capacity) {
        this.capacity = capacity;
        queue = new int[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }

    // Check if queue is empty
    public boolean isEmpty() {
        return size == 0;
    }

    // Check if queue is full
    public boolean isFull() {
        return size == capacity;
    }

    // Add element to the queue
    public void enqueue(int value) {
        if (isFull()) {
            System.out.println("Queue is full! Cannot add " +
value);
            return;
        }
        rear++;                 // Move rear forward
        queue[rear] = value;    // Insert new element
```

```
        size++;                    // Increase size
        System.out.println(value + " added to queue");
    }

    // Remove element from the queue
    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty! Cannot remove
element");
            return -1;   // Return -1 to indicate failure
        }
        int removedValue = queue[front];   // Get front element
        front++;           // Move front forward
        size--;            // Decrease size
        System.out.println(removedValue + " removed from
queue");
        return removedValue;
    }

    // View the front element without removing
    public int peek() {
        if (isEmpty()) {
            System.out.println("Queue is empty!");
            return -1;
        }
        return queue[front];
    }

    // Display all elements in the queue
    public void display() {
        if (isEmpty()) {
            System.out.println("Queue is empty!");
            return;
        }
        System.out.print("Queue elements: ");
        for (int i = front; i <= rear; i++) {
```

```
            System.out.print(queue[i] + " ");
        }
        System.out.println();
    }

    // Main method
    public static void main(String[] args) {
        SimpleQueue q = new SimpleQueue(5);

        q.enqueue(10);
        q.enqueue(20);
        q.enqueue(30);
        q.display();
        System.out.println("Front element is: " + q.peek());
        q.dequeue();
        q.display();
        q.enqueue(40);
        q.enqueue(50);
        q.enqueue(60);
        q.display();
    }
}
```

Output:

```
10 added to queue
20 added to queue
30 added to queue
Queue elements: 10 20 30
Front element is: 10
10 removed from queue
Queue elements: 20 30
40 added to queue
50 added to queue
Queue is full! Cannot add 60
Queue elements: 20 30 40 50
```

Question:

Write a Java class named `SimpleQueue` that implements a queue of integers using an **array** with a fixed size. Your class should include the following methods:

- `void enqueue(int value)` — Adds an element to the rear of the queue. If the queue is full, print `"Queue is full"` and do not add the element.
- `int dequeue()` — Removes and returns the element at the front of the queue. If the queue is empty, print `"Queue is empty"` and return `-1`.
- `int peek()` — Returns the element at the front without removing it. If the queue is empty, print `"Queue is empty"` and return `-1`.
- `boolean isEmpty()` — Returns `true` if the queue is empty; otherwise, returns `false`.
- `boolean isFull()` — Returns `true` if the queue is full; otherwise, returns `false`.
- `void display()` — Prints all elements from front to rear.

Additionally, write a `main` method to demonstrate these operations by:

1. Enqueuing three integers (e.g., 10, 20, 30).
2. Displaying the queue.
3. Dequeuing one element.
4. Displaying the queue again.
5. Peeking at the front element and printing it.

Expected Output:

```
10 enqueued
20 enqueued
30 enqueued
Queue elements: 10 20 30
10 dequeued
Queue elements: 20 30
Front element is: 20
```

✅ Advantages of Simple Queue

1. **Easy to implement**

2. **Maintains order**
   - Elements are processed in **First-In-First-Out (FIFO)** order.

3. **Useful in real-life scenarios**

- Matches real-world processes like customer service lines, task scheduling, etc.

4. **Efficient in processing tasks**
   - Ideal when tasks must be handled in the exact order they arrive.

❌ Disadvantages of Simple Queue

1. **Wasted space (in array-based implementation)**
   - After several dequeues, front moves forward, but empty spaces at the start **cannot be reused**.

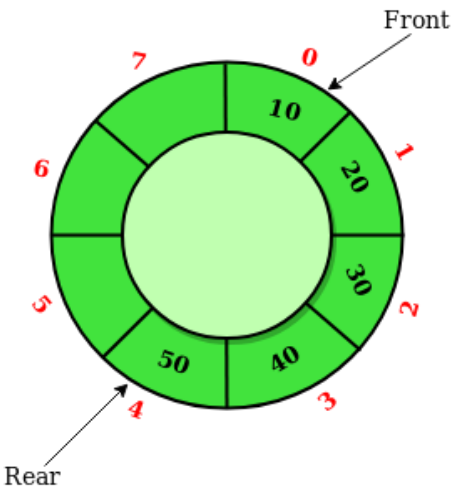
2. **Fixed size (in array implementation)**
   - Limited by predefined array size unless implemented with a dynamic structure like a linked list.

3. **Not suitable for priority handling**
   - All elements are treated equally; you can't prioritize urgent tasks.

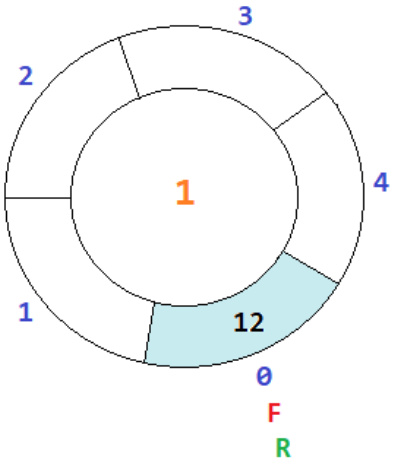
4. **No direct access to middle elements**

2. Circular Queue

A **Circular Queue** overcomes the limitation in simple queues by connecting the end of the queue back to the front — forming a circle.

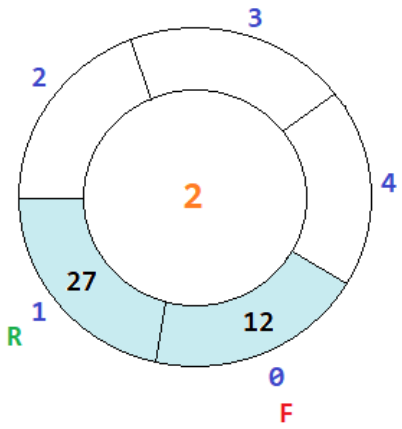


How Circular Queues Work



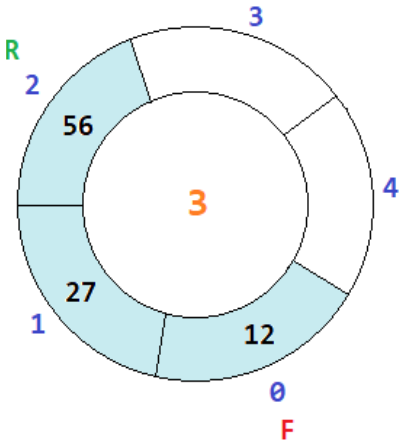
Size = 5
F = 0
R = -1
te = 0 (Total Elements)

Add 12
R = (R+1) % Size = 0
te = 1



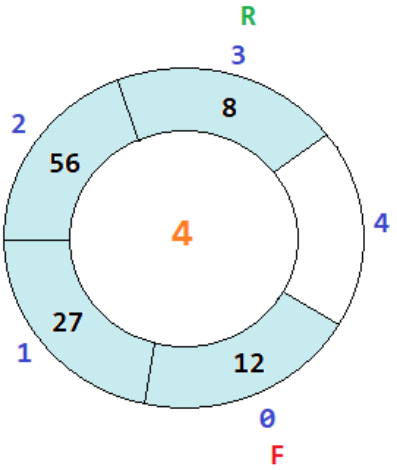
Size = 5
F = 0
R = 0
te = 1 (Total Elements)

Add 27
R = (R+1) % Size = 1
te = 2



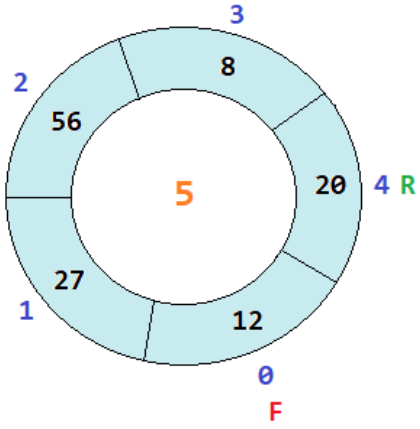
Size = 5
F = 0
R = 1
te = 2 (Total Elements)

Add 56
R = (R+1) % Size = 2
te = 3



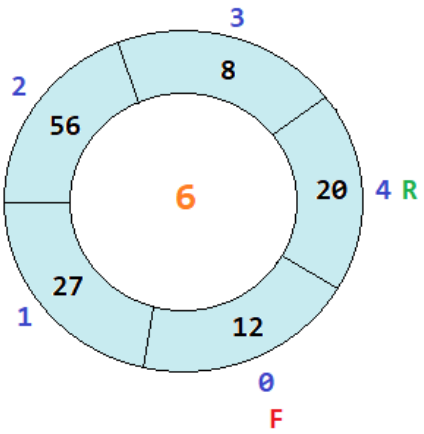
Size = 5
F = 0
R = 2
te = 3 (Total Elements)

Add 8
R = (R+1) % Size = 3
te = 4



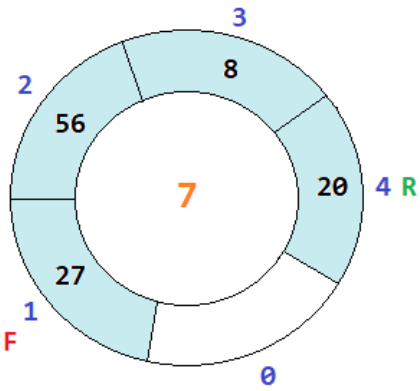
Size = 5
F = 0
R = 3
te = 4 (Total Elements)

Add 20
R = (R+1) % Size = 4
te = 5



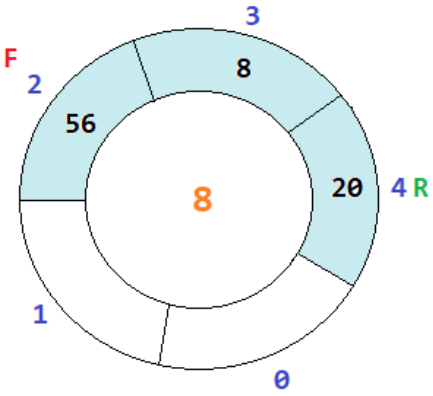
Size = 5
F = 0
R = 4
te = 5 (Total Elements)

Add 15
te == Size
Queue is full



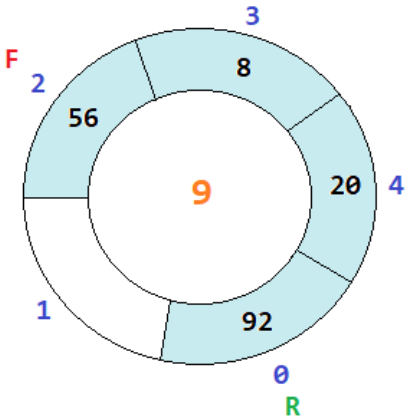
Size = 5
F = 0
R = 4
te = 5 (Total Elements)

Delete 12
F = (F+1) % Size = 1
te = 4



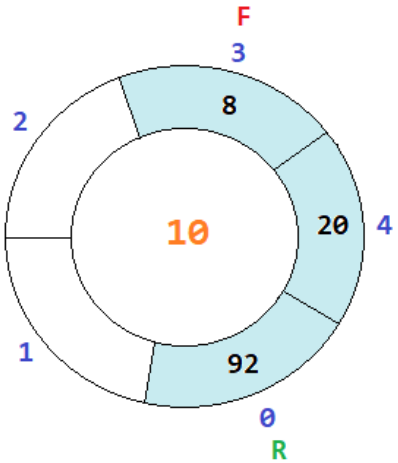
Size = 5
F = 1
R = 4
te = 4 (Total Elements)

Delete 27
F = (F+1) % Size = 2
te = 3



Size = 5
F = 2
R = 4
te = 3 (Total Elements)

Add 92
R = (R+1) % Size = 0
te = 4



Size = 5
F = 2
R = 0
te = 4 (Total Elements)

Delete 56
F = (F+1) % Size = 3
te = 3

Question:

A circular queue has a maximum size of **5** and is initially empty.

Perform the following operations in order:

1. Enqueue(10)
2. Enqueue(20)
3. Enqueue(30)
4. Dequeue()
5. Enqueue(40)
6. Enqueue(50)
7. Enqueue(60)

a) Draw the queue array after all the operations are completed.

b) What are the values of `front` and `rear` at the end?

c) What is the element at the front of the queue (peek)?

d) Can we enqueue another element now? Why or why not?

Java code for Circular Queue

```
public class CircularQueue {
    int[] queue;
    int front, rear, size, capacity;

  // Constructor to initialize queue
    public CircularQueue(int capacity) {
        this.capacity = capacity;
        queue = new int[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }
```

```
    // Check if the queue is full
    public boolean isFull() {
        return size == capacity;
    }

    // Check if the queue is empty
    public boolean isEmpty() {
        return size == 0;
    }

    // Add element to the rear
    public void enqueue(int value) {
        if (isFull()) {
            System.out.println("Queue is full! Cannot add " +
value);
            return;
        }
        rear = (rear + 1) % capacity; // Wrap-around
        queue[rear] = value;
        size++;
        System.out.println(value + " enqueued");
    }

    // Remove element from the front
    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty! Cannot remove
element");
            return -1;
        }
        int removed = queue[front];
        front = (front + 1) % capacity; // Wrap-around
        size--;
        System.out.println(removed + " dequeued");
        return removed;
    }
```

```
// Peek at the front element
public int peek() {
    if (isEmpty()) {
        System.out.println("Queue is empty!");
        return -1;
    }
    return queue[front];
}

// Display all elements in queue
public void display() {
    if (isEmpty()) {
        System.out.println("Queue is empty!");
        return;
    }
    System.out.print("Queue elements: ");
    for (int i = 0; i < size; i++) {
        int index = (front + i) % capacity;
        System.out.print(queue[index] + " ");
    }
    System.out.println();
}

// Main method
public static void main(String[] args) {
    CircularQueue cq = new CircularQueue(5);

    cq.enqueue(10);
    cq.enqueue(20);
    cq.enqueue(30);
    cq.dequeue();
    cq.enqueue(40);
    cq.enqueue(50);
    cq.enqueue(60);
```

```
        cq.display();
        System.out.println("Front element is: " + cq.peek());
    }
}
```

Output:

```
10 enqueued
20 enqueued
30 enqueued
10 dequeued
40 enqueued
50 enqueued
60 enqueued
Queue elements: 20 30 40 50 60
Front element is: 20
```

✅ Advantages of Circular Queue

1. **Efficient Memory Utilization**
    - Unlike simple queues, it **reuses empty spaces** at the front after dequeue operations (wrap-around).

2. **Prevents Overflow in Fixed Size Queues**
    - Space is not wasted when the rear reaches the end but there's room at the beginning.

3. **Faster than Shifting Elements**

4. **Good for Fixed-Size Buffering**

❌ Disadvantages of Circular Queue

1. **More Complex Logic**
    - Requires careful handling of `front` and `rear` pointers using modular arithmetic (`% capacity`).

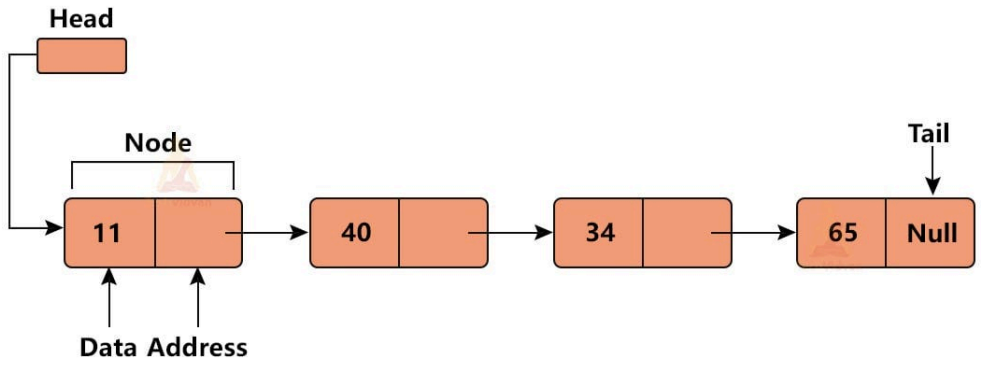
2. **Full vs Empty Detection**
    - Requires extra conditions or a `size` variable to differentiate between a full and an empty queue.

3. **Fixed Capacity**

Linked Lists

Definition:
A linked list is a linear data structure consisting of nodes, where each node contains data and a reference (pointer) to the next node.

Head - the first node **Tail -** the last node



Advantages:

- Dynamic size flexibility.
- Efficient insertion/deletion (especially at the beginning or middle).
- No memory waste due to fixed array size.

Disadvantages:

- No direct access to elements (sequential access only).
- Extra memory needed for pointers.
- Traversal can be slower compared to arrays.

Difference between Arrays and Linked Lists

Feature	**Array**	**Linked List**
Structure	Fixed-size linear structure	Dynamic linear structure made of nodes
Storage	Stored in **contiguous memory locations**	Nodes stored in **non-contiguous memory**
Size	Must be declared **in advance** (static)	Can **grow or shrink** at runtime (dynamic)
Insertion/ Deletion	**Costly** (need to shift elements)	**Efficient** (just update pointers)
Memory Usage	Uses less memory (only data)	More memory (data + pointer for each node)

Implementation	Easy to implement	More complex to implement

Types of Linked Lists:

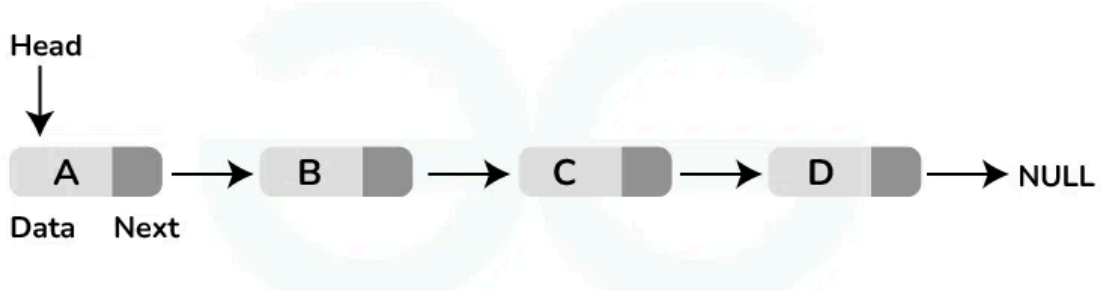
- ***Singly Linked List***
- ***Doubly Linked List***
- ***Circular Linked List***

Singly Linked List

A **Singly Linked List** is a **linear data structure** made up of **nodes**, where each node points to the **next node** in the sequence.

Data – the value stored in the node

Next – a pointer/reference to the next node in the list



Features

- A linear data structure made of **nodes**
- Each node contains **data** and a **pointer** to the next node
- First node is called the **head**
- Last node points to **null** (end of list)
- Nodes are stored **non-contiguously** in memory

Advantages

- **Dynamic size** – easily grow or shrink as needed
- **Efficient insertion/deletion** at beginning or middle
- **No memory waste** (unlike arrays with unused capacity)
- Useful when the number of elements is **unknown or changes frequently**

Limitations

- **No backward traversal** – can only move forward
- **Slow access** – must traverse from head to reach an element
- **More memory** – due to extra pointer in each node
- **Less efficient for random access** (unlike arrays)

Common Use Cases

- **Implementing stacks and queues**
- **Dynamic memory allocation systems**
- **Adjacency lists** in graph representations

Typical Operations of singly linked list

1. Traversal

- Visiting each node in the list from the head to the last node.
- Used to read or display the data stored in each node.

2. Insertion

- **At the beginning:** Add a new node before the current head and update the head pointer.
- **At the end:** Traverse to the last node and link the new node after it.

- **At a specific position:** Traverse to the node just before the desired position and link the new node accordingly.

3. Deletion

- **At the beginning:** Remove the head node and update the head to the next node.
- **At the end:** Traverse to the second-last node and set its next pointer to `null`.
- **At a specific position:** Traverse to the node just before the one to delete and update its next pointer to skip the deleted node.

4. Searching

- Traverse the list to find a node containing a given value.
- Return the position or a reference to the node if found; otherwise, indicate not found.

5. Update/Modify

- Traverse to the desired node and change its data value.

[Java code for Singly Linked List]

```
public class SinglyLinkedList {
    // Node class representing each element in the list
    private static class Node {
        int data;
        Node next;

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

```
private Node head;  // head of the list

public SinglyLinkedList() {
    this.head = null;
}

// 1. Insertion at the end
public void insertAtEnd(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
        return;
    }
    Node temp = head;
    while (temp.next != null)
        temp = temp.next;
    temp.next = newNode;
}

// 2. Insertion at the beginning
public void insertAtBeginning(int data) {
    Node newNode = new Node(data);
    newNode.next = head;
    head = newNode;
}

// 3. Traversal - print all nodes
public void traverse() {
    Node temp = head;
    while (temp != null) {
        System.out.print(temp.data + " -> ");
        temp = temp.next;
    }
    System.out.println("null");
}

// 4. Search for a value
public boolean search(int key) {
```

```
        Node temp = head;
        while (temp != null) {
            if (temp.data == key)
                return true;
            temp = temp.next;
        }
        return false;
    }

    // 5. Delete first occurrence of key
    public void delete(int key) {
        if (head == null)
            return;

        // If head node itself holds the key
        if (head.data == key) {
            head = head.next;
            return;
        }

        Node temp = head;
        Node prev = null;

        while (temp != null && temp.data != key) {
            prev = temp;
            temp = temp.next;
        }

        // If key was not found
        if (temp == null)
            return;

        // Unlink the node to delete
        prev.next = temp.next;
    }

    // 6. Update first occurrence of oldValue to newValue
    public void update(int oldValue, int newValue) {
```

```
        Node temp = head;
        while (temp != null) {
            if (temp.data == oldValue) {
                temp.data = newValue;
                return;
            }
            temp = temp.next;
        }
    }

    // Main method
    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();

        list.insertAtEnd(10);
        list.insertAtEnd(20);
        list.insertAtBeginning(5);
        list.insertAtEnd(30);

        System.out.print("List: ");
        list.traverse();
        System.out.println("Search 20: " + list.search(20));

        System.out.println("Search 40: " + list.search(40));

        list.delete(10);
        System.out.print("After deleting 10: ");
        list.traverse();

        list.update(20, 25);
        System.out.print("After updating 20 to 25: ");
        list.traverse();
    }
}
```

Output:

```
List: 5 -> 10 -> 20 -> 30 -> null
```

```
Search 20: true
Search 40: false
After deleting 10: 5 -> 20 -> 30 -> null
After updating 20 to 25: 5 -> 25 -> 30 -> null
```

Question :

Write a Java program to implement a **singly linked list** with the following operations:

1. **Insert a node at the beginning**
2. **Insert a node at the end**
3. **Delete a node with a given value**
4. **Search for a value in the list**
5. **Display all nodes in the list**

Then, in the `main()` method, perform the following steps:

- Insert the values: `10` at end, `5` at beginning, `20` at end, `15` at beginning.
- Delete the value `10`.
- Search for the value `20`.
- Display the final linked list.

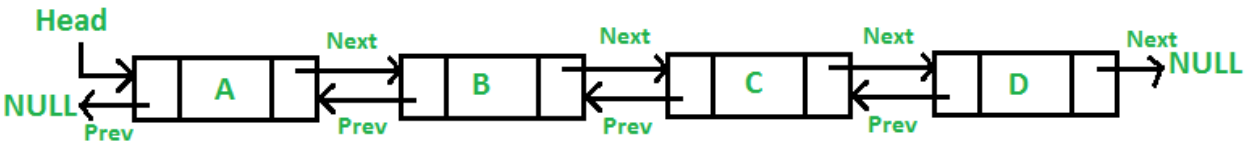
Expected Output:

```
Search for 20: true
Final Linked List: 15 -> 5 -> 20 -> null
```

Doubly Linked List

A **Doubly Linked List** is a type of **linear data structure** where each element (called a **node**) contains **three parts**:

1. **Data** – the actual value you want to store
2. **Pointer to the next node**
3. **Pointer to the previous node**



Features

- Each node has **three fields**: data, next, and **previous**
- Allows **bidirectional traversal** (forward and backward)
- First node's previous is **null**, last node's next is **null**
- More flexible than singly linked list

Advantages

- **Easier deletion** and insertion at both ends
- Supports **backward traversal**
- Ideal for **complex navigation** (undo/redo)
- More suited for applications with frequent two-way movement

Limitations

- Requires **more memory** (extra pointer per node)
- Slightly more complex to implement
- More **pointer updates** during insertion/deletion
- Risk of **memory leaks** if pointers aren't properly managed

Common Use Cases

- **Undo/Redo functionality** in editors and IDEs
- **Browser history** navigation (forward/back)
- **Music or video playlists** with skip forward/back
- **Text editors** and complex buffers

Typical Operations of Doubly Linked List

1. Traversal

- **Forward traversal:** Start from the head and move through next pointers until the end.
- **Backward traversal:** Start from the tail (last node) and move through previous pointers back to the head.

2. Insertion

- **At the beginning:**
  - Create a new node.
  - Set its next pointer to the current head.
  - Update the current head's previous pointer to this new node.
  - Update the head pointer to the new node.
- **At the end:**
  - Traverse to the last node.
  - Set the new node's previous pointer to the last node.
  - Set the last node's next pointer to the new node.
- **At a specific position:**
  - Traverse to the node after which you want to insert.
  - Adjust the previous and next pointers of adjacent nodes and the new node accordingly.

3. Deletion

- **At the beginning:**
  - Update the head pointer to the second node.
  - Set the new head's previous pointer to `null`.
  - Delete the old head node.

- **At the end:**
    - Traverse to the last node.
    - Update the previous node's next pointer to `null`.
    - Delete the last node.
- **At a specific position:**
    - Traverse to the node to delete.
    - Update the previous node's next pointer to the node after the one deleted.
    - Update the next node's previous pointer to the node before the one deleted.
    - Delete the node.

4. Searching

- Traverse forward (or backward) to find a node with a specific value.

5. Update/Modify

- Traverse to the node and change its data value.

[Java code for Doubly Linked List]

```
public class DoublyLinkedList {
    // Node class
    static class Node {
        int data;
        Node prev;
        Node next;

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

```
    }
}

private Node head; // start of the list
private Node tail; // end of the list

// Insert at the beginning
public void insertAtBeginning(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = tail = newNode;
        return;
    }
    newNode.next = head;
    head.prev = newNode;
    head = newNode;
}

// Insert at the end
public void insertAtEnd(int data) {
    Node newNode = new Node(data);
    if (tail == null) {
        head = tail = newNode;
        return;
    }
    tail.next = newNode;
    newNode.prev = tail;
    tail = newNode;
}

// Delete a node with given value
public void delete(int value) {
    Node temp = head;

    while (temp != null) {
        if (temp.data == value) {
            if (temp == head) {
                head = head.next;
```

```
                if (head != null) head.prev = null;
            } else if (temp == tail) {
                tail = tail.prev;
                if (tail != null) tail.next = null;
            } else {
                temp.prev.next = temp.next;
                temp.next.prev = temp.prev;
            }
            return;
        }
        temp = temp.next;
    }
}

// Search for a value
public boolean search(int value) {
    Node temp = head;
    while (temp != null) {
        if (temp.data == value)
            return true;
        temp = temp.next;
    }
    return false;
}

// Update a node value
public void update(int oldVal, int newVal) {
    Node temp = head;
    while (temp != null) {
        if (temp.data == oldVal) {
            temp.data = newVal;
            return;
        }
        temp = temp.next;
    }
}

// Traverse forward
```

```
public void displayForward() {
    Node temp = head;
    while (temp != null) {
        System.out.print(temp.data + " ⇄ ");
        temp = temp.next;
    }
    System.out.println("null");
}

// Traverse backward
public void displayBackward() {
    Node temp = tail;
    while (temp != null) {
        System.out.print(temp.data + " ⇄ ");
        temp = temp.prev;
    }
    System.out.println("null");
}

// Main method
public static void main(String[] args) {
    DoublyLinkedList dll = new DoublyLinkedList();

    dll.insertAtEnd(10);
    dll.insertAtEnd(20);
    dll.insertAtBeginning(5);
    dll.insertAtEnd(30);

    System.out.println("List (Forward):");
    dll.displayForward();

    System.out.println("List (Backward):");
    dll.displayBackward();

    dll.delete(10);
    System.out.println("After deleting 10:");
    dll.displayForward();
```

```
        System.out.println("Search 20: " + dll.search(20));
        System.out.println("Search 100: " + dll.search(100));

        dll.update(20, 25);
        System.out.println("After updating 20 to 25:");
        dll.displayForward();
    }
}
```

Output:

```
List (Forward):
5 ⇄ 10 ⇄ 20 ⇄ 30 ⇄ null
List (Backward):
30 ⇄ 20 ⇄ 10 ⇄ 5 ⇄ null
After deleting 10:
5 ⇄ 20 ⇄ 30 ⇄ null
Search 20: true
Search 100: false
After updating 20 to 25:
5 ⇄ 25 ⇄ 30 ⇄ null
```

Question:

Write a Java program to implement a **Doubly Linked List** with the following operations:

1. **Insert a node at the end**
2. **Display the list in forward direction**
3. **Display the list in reverse direction**
4. **Find and print the middle node of the list**

Then, in the main() method, do the following:

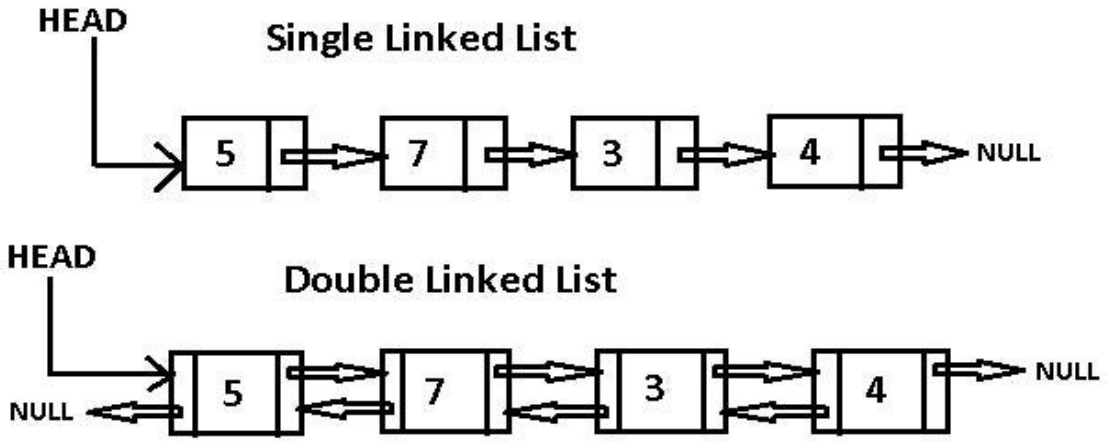
- Insert the values: 11, 22, 33, 44, 55
- Display the list in **forward** direction
- Display the list in **reverse** direction

- Print the value of the **middle node**

Expected Output:

```
Forward List: 11 ⇄ 22 ⇄ 33 ⇄ 44 ⇄ 55 ⇄ null
Backward List: 55 ⇄ 44 ⇄ 33 ⇄ 22 ⇄ 11 ⇄ null
Middle Node: 33
```

Singly Linked List vs Doubly Linked List

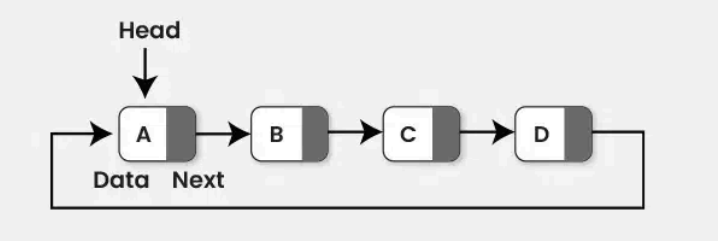


Feature	Singly Linked List	Doubly Linked List
Structure	Each node has `data` and `next` pointer	Each node has `data`, `next`, and `prev`
Traversal	Only **forward**	**Forward and backward**

Memory Usage	Less memory (1 pointer per node)	More memory (2 pointers per node)
Insertion/Deletion	Easy at head, harder at tail	Easy at both head and tail
Reverse Traversal	Not possible	Possible using `prev` pointer
Implementation	Simpler	More complex
Use Cases	Queues, stacks, basic lists	Undo/redo, navigation, playlist editing

Circular Linked List

A **Circular Linked List** is a type of linked list in which the **last node points back to the first node**, forming a continuous loop.



Features

- Last node points to the **first node**, forming a **circle**
- Can be **singly or doubly linked**

- No null pointer at the end
- Can start traversal from any node

Advantages

- Ideal for **circular tasks or continuous loops**
- Easy to **cycle through data repeatedly**
- Efficient for **round-robin scheduling**

Limitations

- Traversal must be handled carefully to avoid **infinite loops**
- More complex to manage insertion/deletion
- Requires **manual breaking** of the loop when needed
- Slightly harder to debug

Common Use Cases

- **Round-robin CPU scheduling**
- **Multiplayer game turns**
- **Circular buffers** in networking
- **Music/slide loop players**

Typical Operations of Circular Linked Lists

1. Traversal

- Start from the head and move forward node by node.
- Stop when you reach the head again (to avoid infinite loop).
- Useful for continuous or repetitive processes.

2. Insertion

- **At the beginning:**
    - Create new node.
    - Set new node's next to current head.
      Update last node's next to point to new node.
    - Update head pointer to new node.

- **At the end:**
    - Create new node.
    - Point current last node's next to new node.
    - New node's next points to head.

- **At a specific position:**
    - Traverse to the node before insertion point.
    - Link new node's next to the next node.
    - Update previous node's next to new node.

3. Deletion

- **At the beginning:**

    - Update head to the next node.
    - Update last node's next to the new head.
    - Delete old head.

- **At the end:**

    - Traverse to the second last node.
    - Update its next to head.
    - Delete the last node.

- **At a specific position:**

    - Traverse to the node before the one to delete.
    - Update its next to skip the deleted node.
    - Delete the node.

4. Searching

- Traverse until you find the value or loop back to head.

5. Update/Modify

- Traverse to the node and change the data.

Java code for Circular Linked List

```
class CircularLinkedList {
    // Node class for circular linked list
    private static class Node {
        int data;
        Node next;

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

    private Node tail;

    // Insert node at the end
    public void insert(int data) {
        Node newNode = new Node(data);
        if (tail == null) {
            tail = newNode;
            tail.next = tail;
        } else {
            // Insert new node after tail and update tail
            newNode.next = tail.next;  // new node points to
head
            tail.next = newNode;       // old tail points to new
node
            tail = newNode;            // new node becomes the
tail
        }
    }

    // Display the circular linked list
    public void display() {
```

```
        if (tail == null) {
            System.out.println("List is empty.");
            return;
        }
        Node current = tail.next;  // Start from head
        do {
            System.out.print(current.data + " -> ");
            current = current.next;
        } while (current != tail.next);
        System.out.println("(back to head)");
    }

    // Delete a node with a given key
    public boolean delete(int key) {
        if (tail == null) return false;

        Node current = tail.next;
        Node prev = tail;
        do {
            if (current.data == key) {
                if (current == tail && current == tail.next) {
                    // Only one node
                    tail = null;
                } else {
                    prev.next = current.next;
                    if (current == tail) {
                        tail = prev;
                    }
                }
                return true; // Node deleted
            }
            prev = current;
            current = current.next;
        } while (current != tail.next);
        return false; // Node not found
    }

    // Search for a value
    public boolean search(int key) {
        if (tail == null) return false;

        Node current = tail.next;
        do {
            if (current.data == key) {
```

```
                return true;
            }
            current = current.next;
        } while (current != tail.next);
        return false;
    }

    // Main method
    public static void main(String[] args) {
        CircularLinkedList cll = new CircularLinkedList();

        cll.insert(10);
        cll.insert(20);
        cll.insert(30);
        cll.insert(40);

        cll.display();
        System.out.println("Search 20: " + cll.search(20));
        System.out.println("Search 50: " + cll.search(50));

        cll.delete(30);
        cll.display();
    }
}
```

Output:

```
10 -> 20 -> 30 -> 40 -> (back to head)
Search 20: true
Search 50: false
10 -> 20 -> 40 -> (back to head)
```

Question:

You are required to implement a **circular singly linked list** in Java that supports the following functionalities:

1. **Insert at the end**
   Add a node with a given integer value at the end of the circular linked list.

2. **Insert at the beginning**
   Add a node with a given integer value at the beginning of the circular linked list.

3. **Delete a node by value**
   Delete the first node in the list that contains the specified value. If the value does not exist, display an appropriate message.

4. **Search for a value**
   Search for a given integer value in the circular linked list and return whether it exists or not.

5. **Display the list**
   Print all the elements of the circular linked list in order, starting from the head node, ending when it loops back to the head.

Expected Output:

```
5 -> 10 -> 20 -> 30 -> (back to head)
Element 20 found in the list.
Element 5 deleted from the list.
10 -> 20 -> 30 -> (back to head)
Element 5 not found in the list.
```