LinkedList: Stack + Queue + Deque

CIS112

BBBF
Yeditepe University

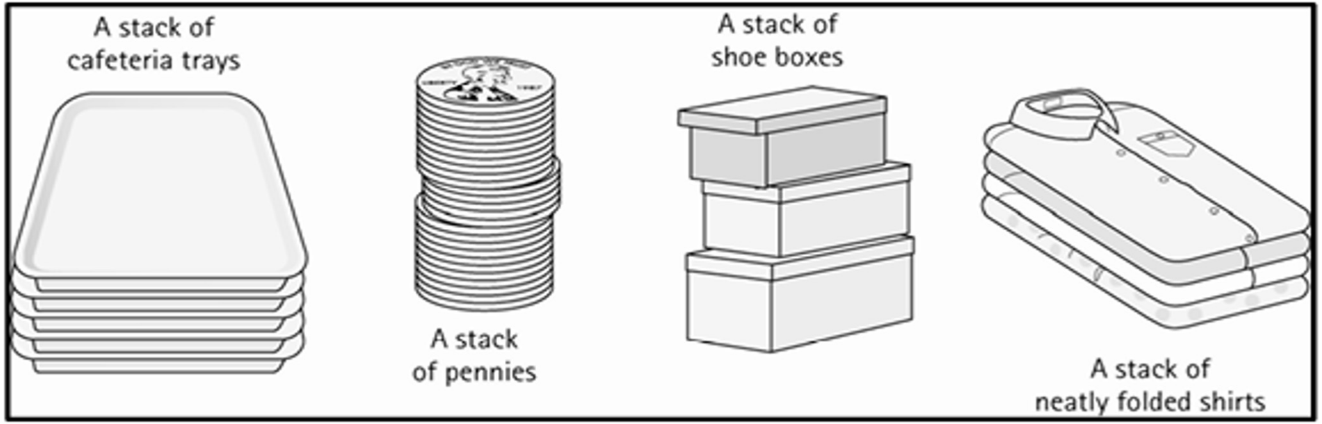
v2024-04-12

Stack

A structure in which elements are added and removed from only one end; a “last in, first out” (LIFO) structure.

Stack principle: LAST IN FIRST OUT = LIFO

It means: the last element inserted is the first one to be removed.



Which is the first element to pick up?

Stack Operations

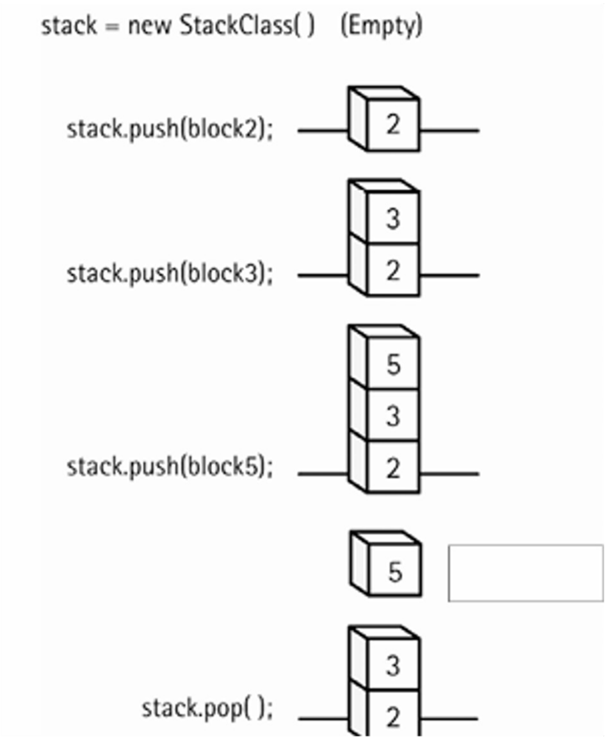
1. **push** adds an element to the top of a stack
2. **pop** removes the top element off the stack
3. **isEmpty** returns a boolean value indicating whether or not the stack is empty

Using LinkedList as Stack

```
class Node {
    int data;
    Node next;


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

```
public class LinkedListStack {
    private Node top; // Points to top of stack


    public LinkedListStack() {
        top = null;
    }
  }
```

Operations : push & pop



```
stack = new StackClass( )   (Empty)

stack.push(block2);

stack.push(block3);

stack.push(block5);

stack.pop( );
```

Operations : push - pop

```
 // Push operation
   public void push(int value) {
       Node newNode = new Node(value);
       newNode.next = top;  // Link new
node
                                         to previous top
       top = newNode;       // Update top
   }
```

```
// Pop operation
    public int pop() {
        if (isEmpty()) {
            throw new RuntimeException("Stack
               Underflow - Stack is empty");}
        int value = top.data;
        top = top.next;  // Move top down
        return value;
    }
```

Operations : peek - isEmpty

```
// Peek operation
  public int peek() {
     if (isEmpty()) {
        throw new RuntimeException("Stack
                        is empty");
     }
     return top.data;
  }
```

```
// Check if stack is empty
  public boolean isEmpty() {
     return top == null;
  }
```

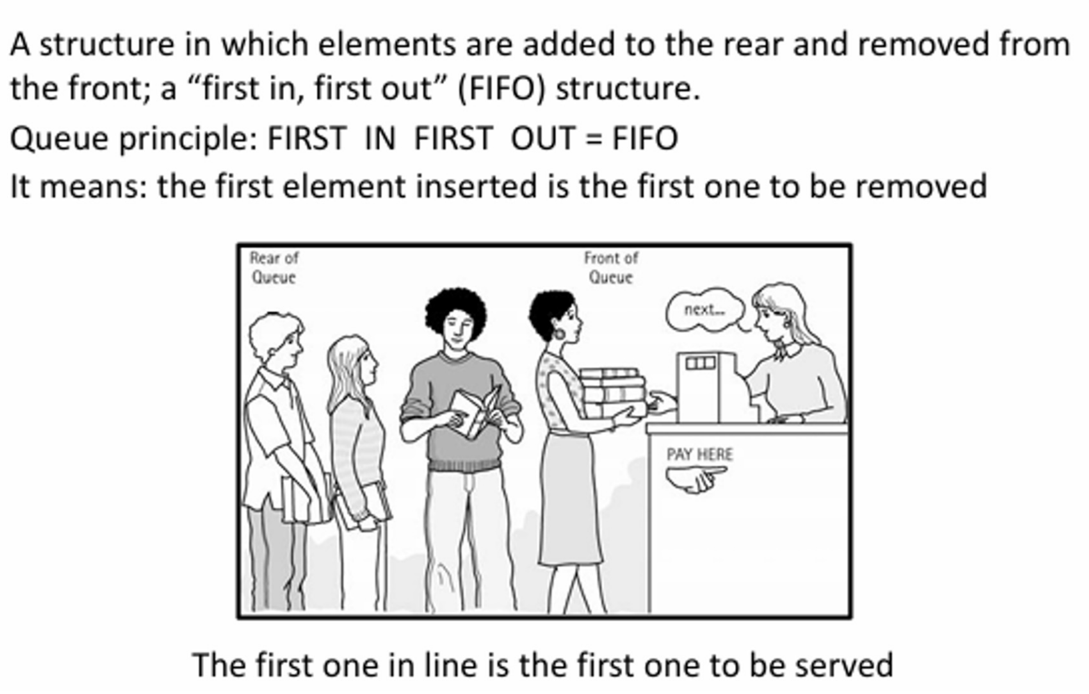
```
// Main method to test
    public static void main(String[] args) {
        LinkedListStack stack = new LinkedListStack();
        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.display(); // Output: Stack: 30 20 10
        System.out.println("Top element is: " +
                                stack.peek()); // 30
        System.out.println("Popped: " + stack.pop()); // 30
        stack.display(); // Output: Stack: 20 10
    }}
```

```
    // Display stack elements
     public void display() {
         Node current = top;
         System.out.print("Stack: ");
         while (current != null) {
             System.out.print(current.data + "
");
             current = current.next;
         }
         System.out.println();
     }
```

Stack Use Case: Undo Feature

- undoStack.push("Step 1");
- undoStack.pop(); // Reverts last step

Queues



A structure in which elements are added to the rear and removed from the front; a “first in, first out” (FIFO) structure.

Queue principle: FIRST IN FIRST OUT = FIFO

It means: the first element inserted is the first one to be removed

Rear of Queue
Front of Queue
next...
PAY HERE

The first one in line is the first one to be served

Queue Operations

1. **enqueue** Adds an element to the rear

2. **dequeue** Removes and returns the front element

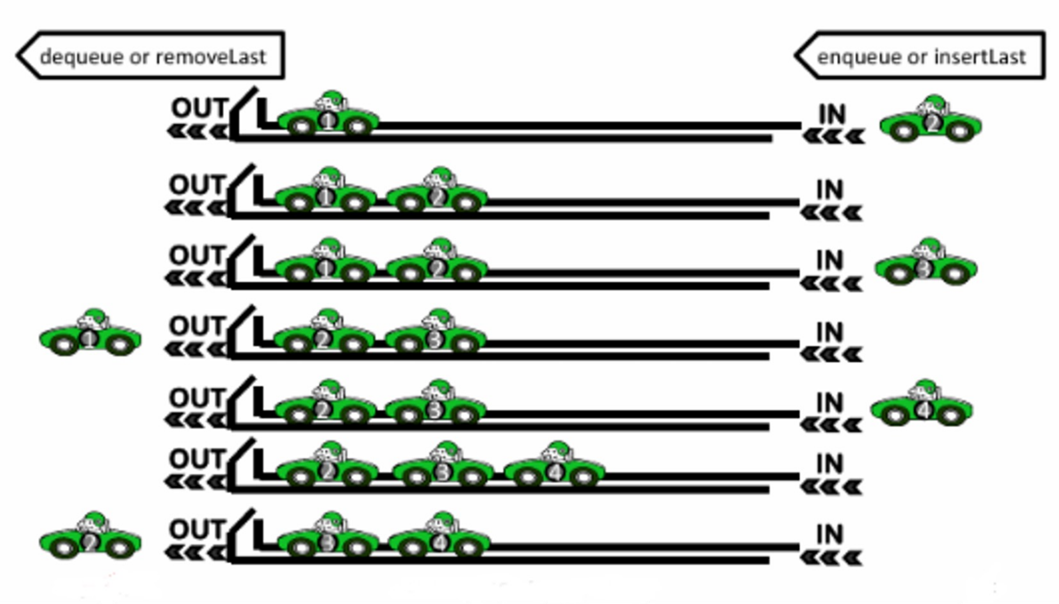
3. **isEmpty** Returns true if the queue is empty and false otherwise

Using LinkedList as Queue

```
// Node class for Linked List
class Node {
    int data;
    Node next;

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

```
// Queue implementation using Linked List
public class LinkedListQueue {
    private Node front, rear;

    public LinkedListQueue() {
        front = rear = null;
    }
}
```

Operations : Enqueue & Dequeue



Operations : Enqueue- Dequeue

```
// Enqueue operation
    public void enqueue(int value) {
        Node newNode = new Node(value);
        if (rear == null) {
            front = rear = newNode;
            return;
        }
        rear.next = newNode;
        rear = newNode;
    }
```

```
// Dequeue operation
    public int dequeue() {
        if (isEmpty()) {
            throw new RuntimeException("Queue
                Underflow - Queue is empty");
        }
        int value = front.data;
        front = front.next;
        if (front == null) {
            rear = null;  // Queue is now empty
        }
```

Operations : peek - isEmpty

```
// Peek operation
    public int peek() {
        if (isEmpty()) {
            throw new
RuntimeException("Queue
                                     is empty");
        }
        return front.data;
    }
```

```
// Check if queue is empty
    public boolean isEmpty() {
        return front == null;
    }
```

```
// Main method to test

  public static void main(String[] args) {

      LinkedListQueue queue = new LinkedListQueue();

      queue.enqueue(1);

      queue.enqueue(2);

      queue.enqueue(3);

      queue.display(); // Output: Queue: 1 2 3

      System.out.println("Front element is: " + queue.peek()); // 1

      System.out.println("Dequeued: " + queue.dequeue()); // 1

      queue.display(); // Output: Queue: 2 3

  }
```

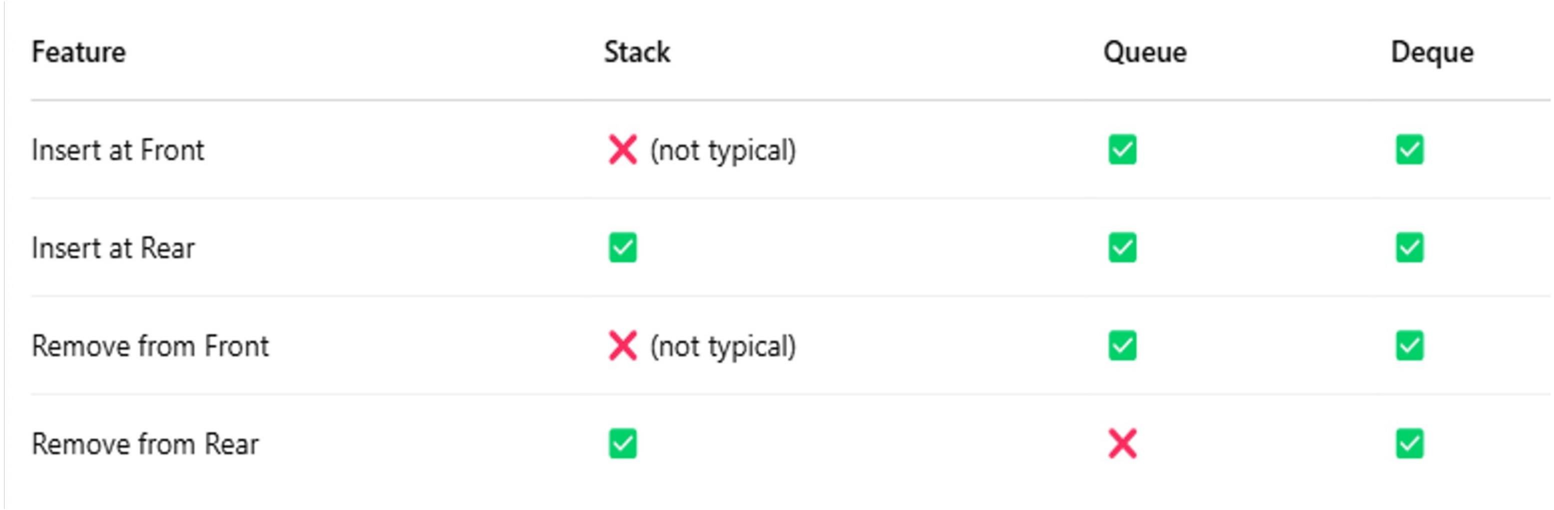
```
// Display queue elements

    public void display() {

        Node current = front;

        System.out.print("Queue: ");

        while (current != null) {

            System.out.print(current.data + " ");

            current = current.next;

        }

        System.out.println();

    }
```

Queue Use Case: Task Management

- queue.enqueue("Task A");
- queue.dequeue(); // Processes Task A

Using Deque for Stack/Queue

Deque stands for **Double-Ended Queue**. It is a **linear data structure** that allows **insertion and deletion at both ends — front and rear**-> Think it as a hybrid of **Stack (LIFO)** and **Queue (FIFO)** — it can act as **either or both** depending on how you use it



Feature	Stack	Queue	Deque
Insert at Front	❌ (not typical)	✅	✅
Insert at Rear	✅	✅	✅
Remove from Front	❌ (not typical)	✅	✅
Remove from Rear	✅	❌	✅

Implementation of a Deque using a Linked List

```
// Node class for doubly linked list
class Node {
    int data;
    Node prev, next;

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

```
// Deque implementation using Doubly Linked List
public class LinkedListDeque {
    private Node front, rear;

    public LinkedListDeque() {
        front = rear = null;
    }
```

Operations : addFront- addRear

```
// Add element to front

  public void addFront(int value) {

    Node newNode = new Node(value);

    if (isEmpty()) {

      front = rear = newNode;

    } else {

      newNode.next = front;

      front.prev = newNode;

      front = newNode;

    }

  }
```

```
// Add element to rear

  public void addRear(int value) {

    Node newNode = new Node(value);

    if (isEmpty()) {

      front = rear = newNode;

    } else {

      newNode.prev = rear;

      rear.next = newNode;

      rear = newNode;

    }

  }
```

Operations : removeFront- removeRear

```
// Remove element from front

  public int removeFront() {

      if (isEmpty()) {

          throw new RuntimeException("Deque Underflow - Empty from
front");

      }

      int val = front.data;

      front = front.next;

      if (front != null) front.prev = null;

      else rear = null;

      return val;

  }
```

```
// Remove element from rear

  public int removeRear() {

      if (isEmpty()) {

          throw new RuntimeException("Deque Underflow - Empty
from rear");

      }

      int val = rear.data;

      rear = rear.prev;

      if (rear != null) rear.next = null;

      else front = null;

      return val;

  }
```

Operations : peek - isEmpty

```
// Peek front

  public int peekFront() {

    if (isEmpty()) throw new RuntimeException("Deque is empty");

    return front.data;

  }

  // Peek rear

  public int peekRear() {

    if (isEmpty()) throw new RuntimeException("Deque is empty");

    return rear.data;

  }
```

```
// Check if deque is empty

  public boolean isEmpty() {

    return front == null;

  }
```

```
// Test in main
    public static void main(String[] args) {
        LinkedListDeque deque = new LinkedListDeque();
        deque.addRear(10);
        deque.addRear(20);
        deque.addFront(5);
        deque.display(); // Output: Deque: 5 10 20
        System.out.println("Front: " + deque.peekFront()); // 5
        System.out.println("Rear: " + deque.peekRear());   // 20
        deque.removeFront(); // Removes 5
        deque.removeRear();  // Removes 20
        deque.display();     // Output: Deque: 10
    }
```

```
// Display deque contents from front to rear
    public void display() {
        Node temp = front;
        System.out.print("Deque: ");
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
```