**Deque in Java**

**Introduction**

A **Deque** (Double-Ended Queue) is a data structure that allows insertion and deletion of elements from both ends. It can operate as both a **Queue** (FIFO) and a **Stack** (LIFO). In Java, the Deque interface is part of the java.util package and is implemented by classes like ArrayDeque and LinkedList.

**Key Characteristics of a Deque**

1. **Bidirectional Operations**: Elements can be added or removed from both ends (front and rear).
2. **Flexible Functionality**:
   * Acts as a Queue (FIFO) or a Stack (LIFO).
3. **Dynamic Size**: The size adjusts dynamically as elements are added or removed.

**Deque Interface in Java**

The Deque interface is a generic interface that allows storage of any type of object.

**Declaration:**

Deque<Type> deque = new ArrayDeque<>();

**Example:**

import java.util.Deque;

import java.util.ArrayDeque;

public class Main {

public static void main(String[] args) {

Deque<Integer> deque = new ArrayDeque<>();

// Add elements at the rear

deque.addLast(10);

deque.addLast(20);

deque.addLast(30);

// Add elements at the front

deque.addFirst(0);

// Peek elements

System.out.println("Front element: " + deque.peekFirst());

System.out.println("Rear element: " + deque.peekLast());

// Remove elements

System.out.println("Removed from front: " + deque.pollFirst());

System.out.println("Removed from rear: " + deque.pollLast());

}

}

**Methods of the Deque Interface**

1. **Insertion Methods**:
   * addFirst(E e): Adds an element at the front.
   * addLast(E e): Adds an element at the rear.
   * offerFirst(E e): Adds an element at the front without throwing an exception.
   * offerLast(E e): Adds an element at the rear without throwing an exception.
2. **Removal Methods**:
   * removeFirst(): Removes and returns the front element.
   * removeLast(): Removes and returns the rear element.
   * pollFirst(): Removes and returns the front element, or null if empty.
   * pollLast(): Removes and returns the rear element, or null if empty.
3. **Retrieval Methods**:
   * getFirst(): Returns the front element without removing it.
   * getLast(): Returns the rear element without removing it.
   * peekFirst(): Returns the front element or null if empty.
   * peekLast(): Returns the rear element or null if empty.
4. **Stack-Like Methods**:
   * push(E e): Adds an element at the front.
   * pop(): Removes and returns the front element.
5. **Others**:
   * isEmpty(): Checks if the deque is empty.
   * size(): Returns the number of elements.

**Implementations of Deque in Java**

1. **ArrayDeque**:
   * A resizable array-based implementation of the Deque interface.
   * More efficient than LinkedList for stack and queue operations.
   * Does not allow null elements.

Example:

Deque<Integer> deque = new ArrayDeque<>();

deque.addFirst(10);

deque.addLast(20);

1. **LinkedList**:
   * Implements both List and Deque interfaces.
   * Allows null elements.

Example:

Deque<Integer> deque = new LinkedList<>();

deque.addFirst(10);

deque.addLast(20);

**Applications of Deque**

1. **Sliding Window Problems**: Efficiently find the maximum or minimum in a sliding window.
2. **Palindrome Checking**: Used to check if a sequence is a palindrome.
3. **Expression Parsing**: Useful in evaluating mathematical expressions.
4. **Task Scheduling**: Allows efficient scheduling with priorities.
5. **Undo/Redo Mechanisms**: Maintains the history of operations for reversible actions.

**Variants of Deque**

1. **Input-Restricted Deque**:
   * Insertion allowed only at one end, but deletion allowed from both ends.
2. **Output-Restricted Deque**:
   * Deletion allowed only at one end, but insertion allowed at both ends.

**Limitations of Deque**

1. **Thread Safety**: Default implementations are not thread-safe.
2. **Memory Overhead**: Dynamic resizing may lead to increased memory usage.
3. **Performance**: Linked-based implementations may have higher overhead for random access.

**Best Practices**

1. Use ArrayDeque for most use cases due to its efficiency.
2. Avoid using LinkedList unless null elements or random access are required.
3. For thread-safe operations, use ConcurrentLinkedDeque.

**Comparison with Other Data Structures**

1. **Queue**: Only allows operations at the front and rear.
2. **Stack**: Operates on LIFO; Deque provides more flexibility.
3. **List**: Allows random access, unlike Deque.

**Frequently Asked Questions**

1. **Why use Deque over Queue or Stack?**
   * For problems requiring flexibility in insertion and deletion from both ends.
2. **Which Deque implementation is most efficient?**
   * ArrayDeque is generally the most efficient.
3. **Can Deque store null elements?**
   * LinkedList allows null elements; ArrayDeque does not.

**Conclusion**

The Deque interface in Java provides a versatile and powerful data structure for a wide range of applications. Its ability to operate as both a queue and a stack makes it an essential tool for developers. Choosing the right implementation, such as ArrayDeque or LinkedList, depends on the specific requirements of the application.