

Queue Implementation in Java



Explain the concept of queue in simple but detail. Then implement them using java, give full code and implement all the methods. Then perform all the operation in main() method. Use primitive



Concept of Queue

A queue is a linear data structure that follows the First In, First Out (FIFO) principle. This means that the first element added to the queue will be the first one to be removed. Think of it like a line at a grocery store: the first person in line is the first one to be served.

Key Operations

1. **Enqueue:** Add an element to the end of the queue.
2. **Dequeue:** Remove the element from the front of the queue.
3. **Peek:** View the element at the front of the queue without removing it.
4. **IsEmpty:** Check if the queue is empty.

Java Implementation Using Primitive Array

Here's a simple implementation of a queue using a primitive array in Java. This implementation includes all the key operations.

```
java
```

```
public class Queue {
    private int[] array;
    private int front, rear, size, capacity;

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

    // Add an element to the end of the queue
    public void enqueue(int item) {
        if (isFull()) {
            System.out.println("Queue is full. Cannot enqueue " + item);
            return;
        }
        rear = (rear + 1) % capacity;
        array[rear] = item;
        size++;
    }

    // Remove the element from the front of the queue
    public int dequeue() {
        if (isEmpty()) {
            return -1;
        }
        int item = array[front];
        array[front] = -1;
        front = (front + 1) % capacity;
        size--;
        return item;
    }

    // Check if the queue is full
    public boolean isFull() {
        return (rear == front - 1);
    }

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

```

        throw new RuntimeException("Queue is empty. Cannot dequeue.");
    }
    int item = array[front];
    front = (front + 1) % capacity;
    size--;
    return item;
}

// View the element at the front of the queue without removing it
public int peek() {
    if (isEmpty()) {
        throw new RuntimeException("Queue is empty. Cannot peek.");
    }
    return array[front];
}

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

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

// Get the size of the queue
public int size() {
    return size;
}

// Main method to perform all operations
public static void main(String[] args) {
    Queue queue = new Queue(5); // Create a queue with capacity 5

    System.out.println("Enqueuing elements:");
    queue.enqueue(10);
    queue.enqueue(20);
    queue.enqueue(30);
    queue.enqueue(40);
    queue.enqueue(50);

    System.out.println("Queue size after enqueues: " + queue.size());

    System.out.println("Peeking front element: " + queue.peek());

    System.out.println("Dequeuing elements:");
    System.out.println(queue.dequeue());
    System.out.println(queue.dequeue());

    System.out.println("Queue size after dequeues: " + queue.size());

    System.out.println("Enqueuing more elements:");
    queue.enqueue(60);
    queue.enqueue(70);

    System.out.println("Queue size after more enqueues: " + queue.size());

    System.out.println("Peeking front element: " + queue.peek());

    System.out.println("Dequeuing all elements:");
    while (!queue.isEmpty()) {
        System.out.println(queue.dequeue());
    }

    System.out.println("Queue size after all dequeues: " + queue.size());
}
}

```

Explanation of Code

1. **Constructor**: Initializes the queue with a specified capacity.
2. **enqueue(int item)**: Adds an item to the queue. If the queue is full, it prints a message and doesn't add the item.
3. **dequeue()**: Removes and returns the item at the front of the queue. Throws an exception if the queue is empty.
4. **peek()**: Returns the item at the front of the queue without removing it. Throws an exception if the queue is empty.
5. **isEmpty()**: Checks if the queue is empty.
6. **isFull()**: Checks if the queue is full.
7. **size()**: Returns the number of elements in the queue.

The `main` method demonstrates the usage of all the queue operations.



Now give me 5 most asked coding question on queue and give code as well. First explain the problem statements using example then write code. Also explain about complexity if required. And give multiple solution if more ways can be important to know.



5 Most Asked Coding Questions on Queue

Here are five commonly asked coding questions involving queues, along with explanations, code, and complexities:

1. Reverse the First K Elements of a Queue

Problem Statement: Given a queue of integers and an integer `K`, reverse the order of the first `K` elements of the queue while maintaining the order of the remaining elements.

Example:

- **Input:** Queue: [1, 2, 3, 4, 5], K = 3
- **Output:** [3, 2, 1, 4, 5]

Code:

java

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class QueueProblems {
    public static Queue<Integer> reverseKElements(Queue<Integer> queue, int K) {
        if (K > queue.size() || K < 0) {
            throw new IllegalArgumentException("Invalid value for K");
        }

        Stack<Integer> stack = new Stack<>();
        for (int i = 0; i < K; i++) {
            stack.push(queue.poll());
        }
    }
}
```

```

        while (!stack.isEmpty()) {
            queue.add(stack.pop());
        }

        for (int i = 0; i < queue.size() - K; i++) {
            queue.add(queue.poll());
        }

        return queue;
    }

    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        queue.add(1);
        queue.add(2);
        queue.add(3);
        queue.add(4);
        queue.add(5);

        System.out.println("Original Queue: " + queue);
        queue = reverseKElements(queue, 3);
        System.out.println("Queue after reversing first 3 elements: " + queue);
    }
}

```

Complexity:

- **Time Complexity:** $O(n)$, where n is the number of elements in the queue.
- **Space Complexity:** $O(K)$, due to the stack used for reversal.

2. Generate Binary Numbers from 1 to N Using Queue

Problem Statement: Given an integer `N`, generate binary numbers from 1 to N in ascending order.

Example:

- **Input:** $N = 5$
- **Output:** ["1", "10", "11", "100", "101"]

Code:

java

```

import java.util.LinkedList;
import java.util.Queue;

public class QueueProblems {
    public static String[] generateBinaryNumbers(int N) {
        Queue<String> queue = new LinkedList<>();
        String[] result = new String[N];

        queue.add("1");
        for (int i = 0; i < N; i++) {
            String binary = queue.poll();
            result[i] = binary;
            queue.add(binary + "0");
            queue.add(binary + "1");
        }

        return result;
    }

    public static void main(String[] args) {
        int N = 5;
        String[] binaryNumbers = generateBinaryNumbers(N);
    }
}

```

```

        System.out.println("Binary numbers from 1 to " + N + ":");
        for (String num : binaryNumbers) {
            System.out.println(num);
        }
    }
}

```

Complexity:

- **Time Complexity:** $O(N)$, where N is the number of binary numbers generated.
- **Space Complexity:** $O(N)$, for storing binary numbers.

3. Implement a Queue Using Stacks

Problem Statement: Implement a queue using two stacks. You need to support `enqueue` and `dequeue` operations.

Example:

- **Operations:**
 - **enqueue(1):** Queue = [1]
 - **enqueue(2):** Queue = [1, 2]
 - **dequeue():** Returns 1, Queue = [2]

Code:

java

```

import java.util.Stack;

public class QueueUsingStacks {
    private Stack<Integer> stack1 = new Stack<>();
    private Stack<Integer> stack2 = new Stack<>();

    // Enqueue operation
    public void enqueue(int item) {
        stack1.push(item);
    }

    // Dequeue operation
    public int dequeue() {
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        if (stack2.isEmpty()) {
            throw new RuntimeException("Queue is empty");
        }
        return stack2.pop();
    }

    public static void main(String[] args) {
        QueueUsingStacks queue = new QueueUsingStacks();
        queue.enqueue(1);
        queue.enqueue(2);
        System.out.println("Dequeued: " + queue.dequeue());
        queue.enqueue(3);
        System.out.println("Dequeued: " + queue.dequeue());
    }
}

```

Complexity:

- **Time Complexity:** Amortized $O(1)$ for `enqueue`, $O(1)$ for `dequeue`.
- **Space Complexity:** $O(n)$, where n is the number of elements in the queue.

4. Check for Balanced Parentheses Using Queue

Problem Statement: Given a string of parentheses, determine if the parentheses are balanced (i.e., every opening parenthesis has a corresponding closing parenthesis).

Example:

- **Input:** "(()())"
- **Output:** True

Code:

```
java

import java.util.LinkedList;
import java.util.Queue;

public class BalancedParentheses {
    public static boolean isBalanced(String s) {
        Queue<Character> queue = new LinkedList<>();
        for (char c : s.toCharArray()) {
            if (c == '(') {
                queue.add(c);
            } else if (c == ')') {
                if (queue.isEmpty()) {
                    return false;
                }
                queue.poll();
            }
        }
        return queue.isEmpty();
    }

    public static void main(String[] args) {
        String s = "(()())";
        System.out.println("Is balanced: " + isBalanced(s));
    }
}
```

Complexity:

- **Time Complexity:** $O(n)$, where n is the length of the string.
- **Space Complexity:** $O(n)$, for storing characters in the queue.

5. Sliding Window Maximum

Problem Statement: Given an array of integers and a window size `k`, find the maximum value in each window of size `k` as the window slides from the left to the right.

Example:

- **Input:** Array = [1, 3, -1, -3, 5, 3, 6, 7], $k = 3$
- **Output:** [3, 3, 5, 5, 6, 7]

Code:

java

```
import java.util.Deque;
import java.util.LinkedList;

public class SlidingWindowMaximum {
    public static int[] slidingWindowMaximum(int[] nums, int k) {
        if (nums.length == 0 || k == 0) return new int[0];
        int[] result = new int[nums.length - k + 1];
        Deque<Integer> deque = new LinkedList<>();

        for (int i = 0; i < nums.length; i++) {
            while (!deque.isEmpty() && deque.peekFirst() < i - k + 1) {
                deque.pollFirst();
            }
            while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
                deque.pollLast();
            }
            deque.addLast(i);
            if (i >= k - 1) {
                result[i - k + 1] = nums[deque.peekFirst()];
            }
        }

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1, 3, -1, -3, 5, 3, 6, 7};
        int k = 3;
        int[] maxValues = slidingWindowMaximum(nums, k);
        System.out.print("Sliding window maximums: ");
        for (int value : maxValues) {
            System.out.print(value + " ");
        }
    }
}
```

Complexity:

- **Time Complexity:** $O(n)$, where n is the number of elements in the array.
- **Space Complexity:** $O(k)$, for the deque used to store indices of array elements.

These questions cover a variety of use cases for queues, from basic operations to more complex scenarios involving additional data structures.



Now give list of most commonly asked question on stack in interviews. These have to be non coding questions.



Certainly! Here's a list of commonly asked non-coding interview questions related to stacks:

Common Non-Coding Interview Questions on Stacks

1. What is a Stack Data Structure?

- **Explanation:** Describe the stack as a linear data structure that follows the Last In, First Out (LIFO) principle. Explain its basic operations: ``push`` (adding an element to the top), ``pop`` (removing the element from the top), and ``peek`` (viewing the top element without removing it).

2. What are the Applications of Stacks?

- **Explanation:** Discuss various applications of stacks such as:
 - **Function Call Management:** Used in programming languages to manage function calls and local variables (call stack).
 - **Expression Evaluation:** Evaluating postfix expressions (Reverse Polish Notation) or converting infix expressions to postfix.
 - **Backtracking Algorithms:** Used in algorithms like Depth-First Search (DFS) to keep track of the path.

3. How Does the Stack Data Structure Differ from a Queue?

- **Explanation:** Compare stacks and queues by their order of operation:
 - **Stack:** Follows LIFO (Last



Now give list of most commonly asked question on queue in interviews. These have to be non coding questions.



Certainly! Here are some commonly asked non-coding interview questions related to queues:

Common Non-Coding Interview Questions on Queues

1. What is a Queue Data Structure?

- **Explanation:** Describe the queue as a linear data structure that follows the First In, First Out (FIFO) principle. Explain its basic operations: ``enqueue`` (adding an element to the end), ``dequeue`` (removing the element from the front), and ``peek`` (viewing the front element without removing it).

2. What are the Common Applications of Queues?

- **Explanation:** Discuss various applications of queues such as:
 - **Job Scheduling:** Managing tasks or jobs in operating systems.
 - **Print Queue:** Managing print jobs in printers.
 - **Buffering:** Used in scenarios like IO buffering or streaming data.
 - **Breadth-First Search (BFS):** Utilized in BFS algorithms to explore nodes level by level.

3. How Does a Queue Data Structure Differ from a Stack?

- **Explanation:** Compare queues and stacks by their order of operation:
 - **Queue:** Follows FIFO (First In, First Out) principle.
 - **Stack:** Follows LIFO (Last In, First Out) principle.

4. What is a Circular Queue and How Does It Differ from a Regular Queue?

- **Explanation:** Describe a circular queue as a type of queue where the end of the queue connects back to the front, forming a circle. This prevents the issue of wasted space in a regular queue where the front might be empty but the rear is full.

5. Explain the Concept of Priority Queue. How Does it Work?

- **Explanation:** Discuss a priority queue as a type of queue where each element has a priority. Elements are dequeued based on priority rather than order of insertion. Higher priority elements are served before lower priority elements.

6. What is the Difference Between a Queue and a Deque?

- **Explanation:** Compare a queue and a deque (double-ended queue):
 - **Queue:** Elements are added at the rear and removed from the front (FIFO).
 - **Deque:** Elements can be added or removed from both the front and rear (can function as both a stack and a queue).

7. How is Queue Implemented in Programming Languages?

- **Explanation:** Discuss various implementations of queues:
 - **Array-Based Queue:** Uses an array to manage queue operations.
 - **Linked List-Based Queue:** Uses a linked list to manage queue operations.
 - **Queue Using Stacks:** Utilizes two stacks to simulate queue operations.

8. What Are the Advantages and Disadvantages of Using Queues?

- **Explanation:** Discuss the benefits (e.g., simplicity in managing tasks, effective for real-time systems) and drawbacks (e.g., potential for wasted space in non-circular implementations, complexity in certain implementations) of queues.

9. How Can You Handle Queue Overflow and Underflow?

- **Explanation:** Describe overflow as the condition when trying to enqueue into a full queue and underflow as the condition when trying to dequeue from an empty queue. Discuss strategies to handle these conditions, such as resizing the queue or using flags to check the status.

10. What is a Blocking Queue and How Does it Differ from a Regular Queue?

- **Explanation:** Explain a blocking queue as a type of queue that supports operations that wait for the queue to become non-empty (in case of `dequeue``) or non-full (in case of `enqueue``). It is commonly used in concurrent programming to handle synchronization.

These questions test your understanding of the fundamental concepts, applications, and differences of queues, as well as their practical implications in various scenarios.



Explain in detail about both circular queue and priority queue and also implement them and perform operation. And give use cases and all.



 Memory updated

Circular Queue

Concept

A circular queue is a type of queue where the last position is connected back to the first position to form a circle. This prevents the issue of wasted space in a standard queue where elements are added at the rear and removed from the front. When the rear of the queue reaches the end of the array, it wraps around to the beginning if there is space available.

Operations

1. **Enqueue:** Add an element to the rear of the queue.
2. **Dequeue:** Remove an element from the front of the queue.
3. **Peek:** View the element at the front of the queue without removing it.

4. **IsEmpty**: Check if the queue is empty.

5. **IsFull**: Check if the queue is full.

Implementation

java

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

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

    // Add an element to the rear of the queue
    public void enqueue(int item) {
        if (isFull()) {
            System.out.println("Queue is full. Cannot enqueue " + item);
            return;
        }
        rear = (rear + 1) % capacity;
        array[rear] = item;
        size++;
    }

    // Remove an element from the front of the queue
    public int dequeue() {
        if (isEmpty()) {
            throw new RuntimeException("Queue is empty. Cannot dequeue.");
        }
        int item = array[front];
        front = (front + 1) % capacity;
        size--;
        return item;
    }

    // View the element at the front of the queue without removing it
    public int peek() {
        if (isEmpty()) {
            throw new RuntimeException("Queue is empty. Cannot peek.");
        }
        return array[front];
    }

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

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

    // Get the size of the queue
    public int size() {
        return size;
    }

    // Main method to perform operations
    public static void main(String[] args) {
        CircularQueue queue = new CircularQueue(5); // Create a circular queue with
        capacity 5
    }
}
```

```

System.out.println("Enqueuing elements:");
queue.enqueue(10);
queue.enqueue(20);
queue.enqueue(30);
queue.enqueue(40);
queue.enqueue(50);

System.out.println("Queue size after enqueues: " + queue.size());

System.out.println("Dequeuing elements:");
System.out.println(queue.dequeue());
System.out.println(queue.dequeue());

System.out.println("Queue size after dequeues: " + queue.size());

System.out.println("Enqueuing more elements:");
queue.enqueue(60);
queue.enqueue(70);

System.out.println("Queue size after more enqueues: " + queue.size());

System.out.println("Peeking front element: " + queue.peek());

System.out.println("Dequeuing all elements:");
while (!queue.isEmpty()) {
    System.out.println(queue.dequeue());
}

System.out.println("Queue size after all dequeues: " + queue.size());
}

```

Use Cases

1. **Buffer Management:** Circular queues are ideal for buffering and scheduling tasks where fixed-size buffers are used, such as IO buffers.
2. **Resource Scheduling:** Used in real-time systems where tasks need to be executed in a round-robin manner.

Priority Queue

Concept

A priority queue is a type of queue where each element is associated with a priority. Elements are dequeued based on their priority rather than the order of insertion. Higher priority elements are dequeued before lower priority elements.

Operations

1. **Enqueue:** Add an element with a specified priority.
2. **Dequeue:** Remove the element with the highest priority.
3. **Peek:** View the element with the highest priority without removing it.
4. **IsEmpty:** Check if the queue is empty.

Implementation

Here's a simple implementation using Java's `PriorityQueue` class:

java

```
import java.util.PriorityQueue;

public class PriorityQueueExample {
    public static void main(String[] args) {
        // Creating a priority queue with custom comparator to order integers
        PriorityQueue<Integer> priorityQueue = new PriorityQueue<>((a, b) -> b - a); //
Max heap

        System.out.println("Enqueuing elements:");
        priorityQueue.add(10);
        priorityQueue.add(20);
        priorityQueue.add(15);
        priorityQueue.add(30);
        priorityQueue.add(5);

        System.out.println("Queue size after enqueues: " + priorityQueue.size());

        System.out.println("Peeking highest priority element: " + priorityQueue.peek());

        System.out.println("Dequeuing elements:");
        while (!priorityQueue.isEmpty()) {
            System.out.println(priorityQueue.poll());
        }

        System.out.println("Queue size after all dequeues: " + priorityQueue.size());
    }
}
```

Use Cases

1. **Task Scheduling:** Used in operating systems for managing processes or tasks based on priority.
2. **Graph Algorithms:** Employed in algorithms like Dijkstra's shortest path for priority-based node processing.
3. **Event Management:** Used in simulation systems where events are processed in order of their priority.

Both circular queues and priority queues are essential in various real-world applications and can be implemented and managed using different techniques based on the specific requirements of the application.



Monotonic Stack and Queue what are these



Monotonic Stack and Monotonic Queue

Monotonic Stack and **Monotonic Queue** are specialized data structures used in algorithms to solve specific types of problems efficiently. They are called "monotonic" because they maintain a specific order (monotonicity) of their elements, which can be either non-increasing or non-decreasing.

Monotonic Stack

Concept: A monotonic stack is a stack that maintains its elements in a monotonically increasing or decreasing order.

- **Monotonic Increasing Stack:** The stack maintains elements in increasing order from top to bottom.
- **Monotonic Decreasing Stack:** The stack maintains elements in decreasing order from top to bottom.

Common Use Cases:

1. **Next Greater Element:** Finding the next greater element for each element in an array.
2. **Stock Span Problem:** Finding the span of stock prices (i.e., number of consecutive days before the current day when the price was less than or equal to the current price).

Example: Finding the Next Greater Element

Problem Statement: Given an array, for each element, find the next greater element that appears to its right.

Implementation:

java

```
import java.util.Stack;
import java.util.HashMap;
import java.util.Map;

public class MonotonicStackExample {
    public static int[] nextGreaterElements(int[] nums) {
        int n = nums.length;
        int[] result = new int[n];
        Stack<Integer> stack = new Stack<>();
        Map<Integer, Integer> map = new HashMap<>();

        // Traverse the array twice to handle the circular array scenario
        for (int i = 0; i < 2 * n; i++) {
            int num = nums[i % n];
            while (!stack.isEmpty() && stack.peek() < num) {
                map.put(stack.pop(), num);
            }
            stack.push(num);
        }

        // Fill result array with the next greater elements
        for (int i = 0; i < n; i++) {
            result[i] = map.getOrDefault(nums[i], -1);
        }

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {4, 5, 2, 10, 8};
        int[] result = nextGreaterElements(nums);
        System.out.print("Next Greater Elements: ");
        for (int num : result) {
            System.out.print(num + " ");
        }
    }
}
```

Complexity:

- **Time Complexity:** $O(n)$, where n is the number of elements in the array.

- **Space Complexity:** $O(n)$, for storing the stack and map.

Monotonic Queue

Concept: A monotonic queue maintains its elements in a monotonically increasing or decreasing order. It can be used to solve problems related to sliding windows efficiently.

- **Monotonic Increasing Queue:** Maintains elements in increasing order from front to rear.
- **Monotonic Decreasing Queue:** Maintains elements in decreasing order from front to rear.

Common Use Cases:

1. **Sliding Window Minimum/Maximum:** Finding the minimum or maximum value in a sliding window of size k over an array.

Example: Sliding Window Maximum

Problem Statement: Given an array and a window size k , find the maximum value in each window of size k .

Implementation:

java

```
import java.util.Deque;
import java.util.LinkedList;

public class MonotonicQueueExample {
    public static int[] slidingWindowMaximum(int[] nums, int k) {
        int n = nums.length;
        int[] result = new int[n - k + 1];
        Deque<Integer> deque = new LinkedList<>();

        for (int i = 0; i < n; i++) {
            // Remove elements not within the window
            if (!deque.isEmpty() && deque.peekFirst() == i - k) {
                deque.pollFirst();
            }

            // Remove elements from the deque that are smaller than the current element
            while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
                deque.pollLast();
            }

            deque.addLast(i);

            // The maximum element of the current window
            if (i >= k - 1) {
                result[i - k + 1] = nums[deque.peekFirst()];
            }
        }

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1, 3, -1, -3, 5, 3, 6, 7};
        int k = 3;
        int[] maxValues = slidingWindowMaximum(nums, k);
        System.out.print("Sliding Window Maximums: ");
        for (int value : maxValues) {
            System.out.print(value + " ");
        }
    }
}
```

Complexity:

- **Time Complexity:** $O(n)$, where n is the number of elements in the array.
- **Space Complexity:** $O(k)$, for storing indices in the deque.

In summary, monotonic stacks and queues are useful for efficiently solving problems involving order and sequence. The monotonic stack helps in scenarios where you need to track elements based on their relative order, while the monotonic queue is ideal for problems involving sliding windows and maintaining order over a range of elements.



Stock span problem



The **Stock Span Problem** is a classic problem that involves calculating the span of stock prices. The span of a stock price for a given day is defined as the number of consecutive days (including the current day) just before the given day for which the stock price was less than or equal to the stock price on the given day.

Problem Statement

Given a list of daily stock prices, calculate the span of stock prices for each day. The span for each day is the number of consecutive days prior to the current day where the price was less than or equal to the price on the current day.

Example

Consider the following stock prices for 7 days:

```
makefile
```

Prices: [100, 80, 60, 70, 60, 75, 85]

For each day, the span of the stock price is:

- Day 0: 100 → Span = 1 (only itself)
- Day 1: 80 → Span = 1 (only itself)
- Day 2: 60 → Span = 1 (only itself)
- Day 3: 70 → Span = 2 (60, 70)
- Day 4: 60 → Span = 1 (only itself)
- Day 5: 75 → Span = 4 (60, 70, 60, 75)
- Day 6: 85 → Span = 6 (100, 80, 60, 70, 60, 75, 85)

Solution Using Monotonic Stack

We can efficiently solve this problem using a monotonic stack. The idea is to use a stack to keep track of the indices of the stock prices in a way that ensures the stack remains in decreasing order of stock

prices.

Steps:

1. Traverse each day's price and use the stack to keep track of indices.
2. For each price, pop elements from the stack while the price at those indices is less than or equal to the current price.
3. The span for the current day is the difference between the current day index and the index at the top of the stack (after popping).
4. Push the current day's index onto the stack.

Implementation:

java

```
import java.util.Stack;

public class StockSpan {
    public static int[] calculateSpan(int[] prices) {
        int n = prices.length;
        int[] spans = new int[n];
        Stack<Integer> stack = new Stack<>();

        for (int i = 0; i < n; i++) {
            // Pop elements from the stack while the current price is greater than the
            // price at the index in stack
            while (!stack.isEmpty() && prices[stack.peek()] <= prices[i]) {
                stack.pop();
            }

            // If the stack is empty, it means the current price is the highest so far
            spans[i] = stack.isEmpty() ? (i + 1) : (i - stack.peek());

            // Push the current index onto the stack
            stack.push(i);
        }

        return spans;
    }

    public static void main(String[] args) {
        int[] prices = {100, 80, 60, 70, 60, 75, 85};
        int[] spans = calculateSpan(prices);

        System.out.print("Stock Span: ");
        for (int span : spans) {
            System.out.print(span + " ");
        }
    }
}
```

Complexity Analysis

- **Time Complexity:** $O(n)$, where n is the number of days. Each element is pushed and popped from the stack at most once.
- **Space Complexity:** $O(n)$, for storing the stack and the result array.

Use Cases

- **Stock Market Analysis:** Useful for analyzing stock price movements and determining trends.
- **Financial Data Analysis:** Applied in various financial algorithms for pattern detection and prediction.

This approach efficiently calculates the span for each day using a monotonic stack to ensure that each day's span is computed in constant time on average.

ChatGPT can make mistakes. Check important info.