Arrays II

Insertion:

Insertion in an array is typically needed in various scenarios where you need to add elements to the array at a specific position. Here are some common cases where array insertion is necessary:

Dynamic Data Storage:

- Arrays provide a way to store and manage data dynamically. Insertion allows for expanding the array dynamically by adding new elements at specific positions.

Ordered Data:

- When maintaining an ordered collection of data, you might need to insert elements while keeping the order intact. For example, maintaining a sorted list of names and inserting a new name in its correct position.

User Input:

- When dealing with user input, you may need to insert elements at specific positions based on user interactions or requests. For instance, inserting an item at a user-defined position in a shopping cart.

Moving Data:

- Shifting elements by inserting can be required when reorganizing or restructuring data in an array, such as when performing sorting or rearranging elements for efficient searching.

Efficient Memory Management:

- Sometimes, when working with limited memory, you may need to carefully manage data by inserting elements at specific positions to optimize memory usage and access patterns.

These cases highlight the need for array insertion, whether for managing data dynamically or maintaining order. The ability to insert elements into an array is fundamental for solving a wide range of programming problems effectively.

Let's Look at one of the problem statements:

Problem Statement: We need to insert an element at a specified position b in the array by shifting the existing elements to the right.

- We start by checking if the given position b is valid. If it's less than 0 or greater than the array size, it's an invalid position.
- We then iterate through the array in reverse starting from the last element up to the position b. For each element, we shift it one position to the right.
- After creating space for the new element, we insert the new value at the specified position b.

Now, let's look at the pseudo-code:

```
public class ArrayInsertion {
    public static void insertElement(int[] arr, int n, int b, int
newValue) {
        if (b < 0 || b > n) {
            System.out.println("Invalid position for insertion.");
            return;
        }
        for (int i = n - 1; i >= b; i--) {
            arr[i + 1] = arr[i];
        }
        arr[b] = newValue;
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = 5;  // Current size of the array
        int b = 2;  // Position to insert the new value
        int newValue = 10;  // Value to insert
        insertElement(arr, n, b, newValue);
        // Print the updated array
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

Reversing an Array:

Reversing an array is a versatile operation that finds applications in various domains. It allows for efficient handling of data and enables different operations based on the reversed order of the elements.

Problem Statement:

The goal is to reverse an array using two different approaches: one by using extra space to store the reversed array and the other by using the two-pointer approach to reverse the array in place.

Approach 1: Using Extra Space

```
public class ArrayReversal {
    public static int[] reverseArrayWithExtraSpace(int[] arr, int n) {
        int[] reversedArray = new int[n];

        for (int i = n - 1; i >= 0; i--) {
            reversedArray[n - 1 - i] = arr[i];
        }
        return reversedArray;
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = arr.length;
        int[] reversedArray = reverseArrayWithExtraSpace(arr, n);
        // Print the original and reversed arrays
        System.out.println("Original array: " + Arrays.toString(arr));
        System.out.println("Reversed array:"+
Arrays.toString(reversedArray));
    }
}
```

Pseudo Code Explanation:

- We create a new array called reversedArray to store the reversed elements.
- We iterate through the original array in reverse order and copy each element to the reversedArray.
- Finally, we return the reversedArray.

Approach 2: Using Two-Pointers

```java
public class ArrayReversal {

    public static void reverseArrayTwoPointers(int[] arr, int n) {
        int start = 0;
        int end = n - 1;

        while (start < end) {
            // Swap arr[start] and arr[end]
            int temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;
            // Move the pointers towards the center of the array
            start++;
            end--;
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = arr.length;

        // Reverse the array using two pointers
        reverseArrayTwoPointers(arr, n);

        // Print the reversed array
        System.out.println("Reversed array: " + Arrays.toString(arr));
    }
}
```

Pseudo Code Explanation:

- We use two pointers, start and end, initially pointing to the beginning and end of the array, respectively.
- We swap elements at the start and end positions and then move the pointers towards the center until they meet.
- This process effectively reverses the array in place.

Push Zeroes

Pushing zeroes to the end of an array is a fundamental operation that has many applications in data. It allows for efficient handling of data by segregating zero and non-zero elements, enabling various operations based on this segregation.

Problem Statement:

You are given an array arr of integers. Your task is to push all the zeros in the array to the end while maintaining the order of non-zero elements.

```
public class ArrayZeroesToEnd {

    public static void pushZeroesToEnd(int[] arr, int n) {
        int leftPointer = 0;
        int rightPointer = 0;

        while (rightPointer < n) {
            if (arr[rightPointer] != 0) {
                // If the element is non-zero, swap arr[leftPointer]
with arr[rightPointer]
                int temp = arr[leftPointer];
                arr[leftPointer] = arr[rightPointer];
                arr[rightPointer] = temp;
                leftPointer++;
            }
            rightPointer++;
        }
    }

    public static void main(String[] args) {
        int[] arr = {0, 1, 0, 3, 12};
        int n = arr.length;
        // Push zeroes to the end of the array
        pushZeroesToEnd(arr, n);
        // Print the modified array
        System.out.println("Array after pushing zeroes to the end: " +
Arrays.toString(arr));
    }
}
```

Explanation:

Initialization:

- leftPointer = 0: Initialize a variable left pointer to 0, which will point to the current position where a non-zero element can be placed.
- rightPointer = 0: Initialize a variable rightPointer to 0, which will traverse the array.

Traverse the Array using a Single Loop with Two Pointers:

- while rightPointer < n do: Continue the loop while rightPointer is less than the array size.
- if arr[rightPointer] != 0 then: Check if the current element is not zero (a non-zero element).
- swap arr[leftPointer] with arr[rightPointer]: Swap the element at leftPointer with the non-zero element at rightPointer, effectively moving all non-zero elements to the front of the array.
- leftPointer = leftPointer + 1: Increment the leftPointer to indicate the addition of a non-zero element.
- rightPointer = rightPointer + 1: Increment rightPointer to move both pointers together.

Array - Rotation:

Good Going !!
Now, Let's cover the problem statement, pseudo code, and explanation for array rotation using the brute force approach, both left and right rotations.

Problem Statement:

You are given an array arr of integers and an integer x. Your task is to rotate the array by x positions either to the left or to the right.

Right Rotation:

- For right rotation, use the formula: (i + x) % length, where i is the current index and length is the length of the array.

```
public class ArrayRotation {

    public static void rotateRight(int[] arr, int n, int x) {
```

```
        x = x % n;  // Handle cases where x > n
        int[] rotatedArr = new int[n];
        for (int i = 0; i < n; i++) {
            rotatedArr[(i + x) % n] = arr[i];
        }
        // Copy rotated array back to the original array
        for (int i = 0; i < n; i++) {
            arr[i] = rotatedArr[i];
        }
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = arr.length;
        int x = 2;  // Number of positions to rotate
        // Rotate the array to the right by x positions
        rotateRight(arr, n, x);
        // Print the rotated array
        System.out.println("Array after rotating " + x + " positions to the
right: " + Arrays.toString(arr));
    }
}
```

Explanation:

- x = x % n: Normalize the rotation amount x to ensure it's less than the array n's length.
- rotatedArr[(i + x) % n] = arr[i]: For each element in the original array, place it at the (i + x) % n position in the rotated array, effectively performing a right rotation.
- Copying Back to Original Array: (Optional): After rotation, copy the elements from the rotated array back to the original array to reflect the rotation.

Left Rotation:

For left rotation, use the formula: (i + x - length) % length, where i is the current index and length is the length of the array.

```java
import java.util.Arrays;

public class ArrayRotation {

    public static void rotateLeft(int[] arr, int n, int x) {
        x = x % n;  // Handle cases where x > n
        int[] rotatedArr = new int[n];
        for (int i = 0; i < n; i++) {
            rotatedArr[(i - x + n) % n] = arr[i];
        }

        // Copy rotated array back to the original array
        for (int i = 0; i < n; i++) {
            arr[i] = rotatedArr[i];
        }
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = arr.length;
        int x = 2;  // Number of positions to rotate
        // Rotate the array to the left by x positions
        rotateLeft(arr, n, x);
        // Print the rotated array
        System.out.println("Array after rotating " + x + " positions to the
left: " + Arrays.toString(arr));
    }
}
```

Explanation

- x = x % n: Normalize the rotation amount x to ensure it's less than the array n's length.
- rotatedArr[(i + x - n) % n] = arr[i]: For each element in the original array, place it at the (i + x - n) % n position in the rotated array, effectively performing a left rotation.
- Copying Back to Original Array: (Optional): After rotation, copy the elements from the rotated array back to the original array to reflect the rotation.

These pseudocodes outline the steps for rotating an array to the right and to the left using the brute force approach. The rotation is achieved by placing the elements in the new positions calculated based on the given rotation amount x.

Rotate Array: Optimized Approach:

The optimized approach for array rotation involves a reversal technique. Let's cover the problem statement, pseudo code, and explanation for array rotation using this optimized approach, both left and right rotations.

Problem Statement:

You are given an array arr of integers and an integer x. Your task is to rotate the array by x positions either to the left or to the right.

Optimized Approach for Array Rotation:

Right Rotation:

- Reverse the entire array.
- Reverse the subarray from index 0 to x-1.
- Reverse the subarray from index x to length-1.

```
public class ArrayRotation {

    public static void rotateRight(int[] arr, int n, int x) {
        x = x % n;  // Handle cases where x > n
        reverseArray(arr, 0, n - 1);
        reverseArray(arr, 0, x - 1);
        reverseArray(arr, x, n - 1);
    }

    public static void reverseArray(int[] arr, int start, int end) {
        while (start < end) {
            // Swap arr[start] and arr[end]
            int temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;
            start++;
            end--;
        }
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = arr.length;
```

```
        int x = 2;  // Number of positions to rotate
        // Rotate the array to the right by x positions
        rotateRight(arr, n, x);
        // Print the rotated array
        System.out.println("Array after rotating " + x + " positions to
the right: " + Arrays.toString(arr));
    }
}
```

Left Rotation:

- Reverse the entire array.
- Reverse the subarray from index 0 to length-x-1.
- Reverse the subarray from index length-x to length-1.

```
public class ArrayRotation {
    public static void rotateLeft(int[] arr, int n, int x) {
        x = x % n;  // Handle cases where x > n
        reverseArray(arr, 0, n - 1);
        reverseArray(arr, 0, n - x - 1);
        reverseArray(arr, n - x, n - 1);
    }

    public static void reverseArray(int[] arr, int start, int end) {
        while (start < end) {
            // Swap arr[start] and arr[end]
            int temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;
            start++;
            end--;
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = arr.length;
        int x = 2;  // Number of positions to rotate

        // Rotate the array to the left by x positions
```

```
        rotateLeft(arr, n, x);

        // Print the rotated array
        System.out.println("Array after rotating " + x + " positions to
the left: " + Arrays.toString(arr));
    }
}
```

Explanation:

- x = x % n: Normalize the rotation amount x to ensure it's less than the array n's length.
- Reverse Entire Array: reverseArray(arr, 0, n-1): Reverse the entire array.

Reverse Subarrays:

- For right rotation:
  - Reverse the subarray from index 0 to x-1.
  - Reverse the subarray from index x to length-1.
- For left rotation:
  - Reverse the subarray from index 0 to length-x-1.
  - Reverse the subarray from index length-x to length-1.

Reverse Array Procedure:

- reverseArray procedure swaps elements in the given range to reverse the array or subarray.

By reversing the entire array and then reversing appropriate subarrays, we achieve the desired rotation in an optimized manner. This approach avoids multiple rotations and provides a more efficient solution.