**PHASE 1 — Basic Array Operations (Interview-style)**

1. Insert at position ✅ (Interview Important)
2. Delete at position ✅
3. Search (Linear search) ✅
4. Left shift / right shift ✅
5. Left rotation / right rotation ✅
6. Reverse array ✅

**PHASE 2 — Built-in Java Array Tools (Warm-up)**

1. Arrays.toString() → print array nicely
2. Arrays.sort() → sorting
3. Arrays.binarySearch() → search in sorted arrays
4. Arrays.copyOf() / copyOfRange() → copying
5. Arrays.equals() → compare arrays
6. Arrays.fill() → fill array with a value

**PHASE 3 — DSA Practice Problems on Arrays**

1. Remove duplicates ✅
2. Merge two arrays ✅
3. Count occurrences of an element ✅
4. Check if array is palindrome ✅
5. Move all zeros to the end ✅

## Insert at position:

"Insert at Position" in arrays means adding a new element **at a specific index** while shifting existing elements to the right to make space.

* Arrays have **fixed size**, so insertion often requires **shifting** elements after the position.
* If the array is full, you’ll need a **new array** with larger size.

**Given Array:**

arr = [10, 20, 30, 40, 50]

**Insert:** 25 at position 2 (0-based indexing).

**Process:**

1. Shift elements from the end to position 2 one step right.  
   Before: [10, 20, 30, 40, 50, \_]  
   After shift: [10, 20, 30, 30, 40, 50]
2. Place new element:  
   Result: [10, 20, 25, 30, 40, 50]

## ****Code in Java****

public class InsertAtPosition {

public static void main(String[] args) {

int[] arr = new int[6]; // 6-size array (extra space for insertion)

arr[0] = 10;

arr[1] = 20;

arr[2] = 30;

arr[3] = 40;

arr[4] = 50;

int n = 5; // current number of elements

int pos = 2; // position to insert

int value = 25;

// Shift elements to the right

for (int i = n; i > pos; i--) {

arr[i] = arr[i - 1];

}

// Insert the new value

arr[pos] = value;

n++; // increase size

// Display

for (int i = 0; i < n; i++) {

System.out.print(arr[i] + " ");

}

}

}

## ****Explanation of the Problem****

* We **started** with an array of 5 elements but allocated size 6 to allow insertion.
* We **shifted** all elements from the insertion point to the right by one step (done in reverse to avoid overwriting).
* We placed the new value at the desired index.

This increases the logical size of the array by 1.  
  
Before shift:

[10, 20, 30, 40, 50, \_]

Shifting:

* i = 5 → arr[5] = arr[4] → [10, 20, 30, 40, 50, 50]
* i = 4 → arr[4] = arr[3] → [10, 20, 30, 40, 40, 50]
* i = 3 → arr[3] = arr[2] → [10, 20, 30, 30, 40, 50]

Now position 2 is free.

## Delete at position

"Delete at Position" means **removing an element from a specific index** in the array and **shifting the remaining elements left** so there are no empty gaps.

* Since arrays in Java have **fixed size**, the last element will usually be set to 0 (or ignored logically).
* Logical size of the array decreases by 1 after deletion.

## ****. Example****

**Given Array:**

arr = [10, 20, 30, 40, 50]

**Delete at Position:** 2 (0-based indexing).

**Process:**

1. Shift elements from position 3 to position 2.  
   Before: [10, 20, 30, 40, 50]  
   After shift: [10, 20, 40, 50, 50]
2. Set last element to 0 for clarity:  
   Result: [10, 20, 40, 50, 0]

public class DeleteAtPosition {

public static void main(String[] args) {

int[] arr = {10, 20, 30, 40, 50};

int n = arr.length; // 5 elements

int pos = 2; // position to delete

// Shift elements to the left

for (int i = pos; i < n - 1; i++) {

arr[i] = arr[i + 1];

}

// Optional: clear last element

arr[n - 1] = 0;

// Display array after deletion

for (int i = 0; i < n; i++) {

System.out.print(arr[i] + " ");

}

}

}  
  
Example run:

| **i** | **arr[i] before** | **arr[i + 1]** | **arr[i] after shift** |
| --- | --- | --- | --- |
| 2 | 30 | 40 | 40 |
| 3 | 40 | 50 | 50 |

[10, 20, 40, 50, 50, 0]

## ****Explanation****

* We **shift left** starting from the deletion position so that each element takes the place of the previous one.
* Last element becomes a duplicate, so we **clear** it (optional in practice but useful for clarity).

Logical size decreases by 1 (though physical array size stays the same).  
  
 i = pos → start at the position you want to remove.

i < n - 1 → stop **one before the last valid element**, because we’ll be copying from i + 1.

Inside the loop:

Copy the element from the **next position** into the **current position**.

This **fills the gap** left by the deleted element.

## Search (Linear search)

**Linear Search** is the simplest searching technique where we **check each element one-by-one** from the beginning until:

* We find the target element (return its index), or
* We reach the end (element not found).
* Works on **unsorted** or **sorted** arrays.

**Example**

**Given Array:**

arr = [15, 8, 23, 42, 4, 16]

**Search for:** 42

**Process:**

1. Compare 15 → not match.
2. Compare 8 → not match.
3. Compare 23 → not match.
4. Compare 42 → match found at index **3**.

public class LinearSearchExample {

public static void main(String[] args) {

int[] arr = {15, 8, 23, 42, 4, 16};

int target = 42;

int n = arr.length;

int position = -1; // default: not found

for (int i = 0; i < n; i++) {

if (arr[i] == target) {

position = i; // store index

break; // stop search

}

}

if (position != -1) {

System.out.println("Element found at index: " + position);

} else {

System.out.println("Element not found.");

}

}

## }

## ****Explanation****

* Start from index 0.
* Compare each element with the target.
* Stop immediately when found.
* If loop ends without finding, return -1.

## Left shift / right shift :

## ****Definition****

* **Left Shift**: Moves every element **one position to the left**.
  + First element is lost (or moved to a temporary variable if needed).
  + Last position is usually filled with 0 (or left empty logically).
* **Right Shift**: Moves every element **one position to the right**.
  + Last element is lost (or stored temporarily).
  + First position is usually filled with 0.

Shifts are often used in array manipulation, rotations, and implementing queues.

**Left Shift Example**

**Before:**

[10, 20, 30, 40, 50]

**After 1 Left Shift:**

[20, 30, 40, 50, 0]

(Process: Remove first element → Shift all others left → Fill last with 0)

**3. Right Shift Example**

**Before:**

[10, 20, 30, 40, 50]

**After 1 Right Shift:**

[0, 10, 20, 30, 40]

(Process: Remove last element → Shift all others right → Fill first with 0)

**4. Java Code**

public class ShiftExample {

public static void main(String[] args) {

int[] arr = {10, 20, 30, 40, 50};

int n = arr.length;

// LEFT SHIFT

for (int i = 0; i < n - 1; i++) {

arr[i] = arr[i + 1];

}

arr[n - 1] = 0; // fill last

System.out.print("After Left Shift: ");

for (int x : arr) System.out.print(x + " ");

System.out.println();

// Reset array for right shift

arr = new int[]{10, 20, 30, 40, 50};

// RIGHT SHIFT

for (int i = n - 1; i > 0; i--) {

arr[i] = arr[i - 1];

}

arr[0] = 0; // fill first

System.out.print("After Right Shift: ");

for (int x : arr) System.out.print(x + " ");

}

}

**Output:**

After Left Shift: 20 30 40 50 0

After Right Shift: 0 10 20 30 40

**5. Explanation**

* **Left shift**: Start shifting from index 0 towards the left end.
* **Right shift**: Start shifting from last index towards the right end.
* Always fill the "vacated" position with a neutral value (like 0) or keep it logically empty.

Left rotation / right rotation :

**Definition**

* **Left Rotation**: Moves every element **one position to the left**, and the **first element goes to the last position** instead of being lost.
* **Right Rotation**: Moves every element **one position to the right**, and the **last element goes to the first position** instead of being lost.

Unlike **shifts**, rotations **preserve all elements**.

**2. Left Rotation Example**

**Before:**

[10, 20, 30, 40, 50]

**After 1 Left Rotation:**

[20, 30, 40, 50, 10]

**3. Right Rotation Example**

**Before:**

[10, 20, 30, 40, 50]

**After 1 Right Rotation:**

[50, 10, 20, 30, 40]

**4. Java Code — Single Rotation**

public class RotationExample {

public static void main(String[] args) {

int[] arr = {10, 20, 30, 40, 50};

int n = arr.length;

// LEFT ROTATION

int first = arr[0]; // store first element

for (int i = 0; i < n - 1; i++) {

arr[i] = arr[i + 1];

}

arr[n - 1] = first;

System.out.print("After Left Rotation: ");

for (int x : arr) System.out.print(x + " ");

System.out.println();

// RIGHT ROTATION

arr = new int[]{10, 20, 30, 40, 50};

int last = arr[n - 1]; // store last element

for (int i = n - 1; i > 0; i--) {

arr[i] = arr[i - 1];

}

arr[0] = last;

System.out.print("After Right Rotation: ");

for (int x : arr) System.out.print(x + " ");

}

}

**Output:**

After Left Rotation: 20 30 40 50 10

After Right Rotation: 50 10 20 30 40

**5. k-Rotation (Multiple Steps)**

If you want to rotate **k times**, use the formula:

k = k % n // optimization for large k

Then perform k single rotations **OR** use a **reverse array trick** for O(n) rotation without repeated shifting.

java.util.Arrays:

It is a special Java **helper class** that comes with **ready-made methods** for doing common array tasks — so you don’t have to write all the logic yourself.

For example:

* **Normally** → if you want to sort an array, you’d have to write a sorting algorithm (bubble sort, etc.).
* **With Arrays.sort()** → you just call one method and it sorts for you.

1. Printing arrays nicely → Arrays.toString()
2. Sorting → Arrays.sort()
3. Searching quickly → Arrays.binarySearch()
4. Copying arrays → Arrays.copyOf(), Arrays.copyOfRange()
5. Comparing arrays → Arrays.equals()
6. Filling an array with a value → Arrays.fill()

These are **important** because:

* They save time in **real projects**.
* They are **asked in interviews** (syntax-based questions like *"How do you copy part of an array in Java?"*).
* They make code **cleaner** and less error-prone.

**1. Arrays.toString() → Print array nicely**

**Definition:** Converts array into a readable string form instead of printing the object reference.

import java.util.Arrays;

int[] arr = {10, 20, 30};

System.out.println(Arrays.toString(arr));

// Output: [10, 20, 30]

Without this, System.out.println(arr) would print something like [I@2a139a55.

**2. Arrays.sort() → Sorting**

**Definition:** Sorts an array in ascending order (default).

int[] arr = {30, 10, 20};

Arrays.sort(arr);

System.out.println(Arrays.toString(arr));

// Output: [10, 20, 30]

You can also sort partially:

Arrays.sort(arr, 0, 2); // sorts only index 0 and 1

**3. Arrays.binarySearch() → Search in sorted arrays**

**Definition:** Searches for a value in a sorted array and returns its index (or negative value if not found).

int[] arr = {10, 20, 30, 40, 50};

int index = Arrays.binarySearch(arr, 30);

System.out.println(index); // Output: 2

int notFound = Arrays.binarySearch(arr, 25);

System.out.println(notFound); // Output: -3 (insertion point = 2, so -(2+1))

⚠ **Array must be sorted** before using binarySearch.

**4. Arrays.copyOf() / Arrays.copyOfRange() → Copying**

**Definition:** Creates a new array by copying data from another array.

int[] arr = {10, 20, 30, 40};

// Full copy with new size

int[] copy1 = Arrays.copyOf(arr, 6);

System.out.println(Arrays.toString(copy1));

// Output: [10, 20, 30, 40, 0, 0]

// Partial copy

int[] copy2 = Arrays.copyOfRange(arr, 1, 3);

System.out.println(Arrays.toString(copy2));

// Output: [20, 30]

**5. Arrays.equals() → Compare arrays**

**Definition:** Checks if two arrays are equal element-by-element.

int[] a1 = {10, 20, 30};

int[] a2 = {10, 20, 30};

System.out.println(Arrays.equals(a1, a2)); // true

⚠ Works for **one-dimensional** arrays; for multi-dimensional arrays use Arrays.deepEquals().

**6. Arrays.fill() → Fill array with a value**

**Definition:** Sets every element of the array to the same value.

int[] arr = new int[5];

Arrays.fill(arr, 99);

System.out.println(Arrays.toString(arr));

// Output: [99, 99, 99, 99, 99]

You can also fill partially:

Arrays.fill(arr, 1, 4, 55); // fills index 1 to 3