Complete code on github click here

1. Penjelasan Metode yang Digunakan

a. Operasi pada Array (`ArrayOperations`)

- **Traversal**: Menggunakan `Arrays.toString()` untuk menampilkan isi array.
- **Searching**:
  - **Linear search**: Melakukan iterasi elemen satu per satu dari indeks 0 hingga `length - 1`.
  - **Binary search**: Hanya valid jika array **sudah sorted**; kompleksitas waktu O(log n).
- **Insertion**: Karena array memiliki **fixed size**, operasi insertion dilakukan dengan:
  1. Membuat array baru dengan ukuran `n + 1`.
  2. Menggunakan `System.arraycopy()` untuk menyalin bagian sebelum dan sesudah `index`.
- **Deletion**: Serupa dengan insertion—membuat array baru berukuran `n - 1` dan menyalin elemen yang tidak dihapus.

> ⚠ Semua operasi **mutative** (insert/delete) pada array **not in-place**, sehingga memerlukan alokasi memori tambahan dan menghasilkan overhead waktu.

b. Operasi pada ArrayList (`ArrayListOperations`)

- **Add/Remove**: Menggunakan method bawaan `add(index, element)` dan `remove(index)`, yang secara internal menggeser elemen dan menangani resizing.
- **Searching**: Menggunakan `indexOf()`, yang melakukan **linear search** pada internal `Object[]`.
- **Sorting**: Menggunakan `Collections.sort()`, yang mengimplementasikan **Timsort**—algoritma hybrid berbasis merge sort dan insertion sort, stabil dan adaptif.

c. Performance Measurement (`Comparison`)

- Waktu eksekusi diukur menggunakan `System.nanoTime()` untuk presisi tinggi.
- Dataset diinisialisasi identik di kedua struktur: nilai `0` hingga `size - 1`.
- Operasi diuji pada skenario yang setara (misal: insertion di `index = size/2`).
- Pengujian dilakukan pada berbagai ukuran data (`1,000`, `10,000`, `100,000`) untuk mengamati tren performa.

---

2. Analisis Hasil Eksperimen

Berdasarkan pengujian berulang, pola performa sebagai berikut:

Operasi	Array	ArrayList	Analisis
Traversal	Lebih cepat	Sedikit lebih lambat	Array menggunakan `int[]` (primitive), sedangkan ArrayList menyimpan `Integer` (object), sehingga terjadi **autoboxing overhead**.
Linear search	Cepat	Sedikit lebih lambat	Sama-sama O(n), tetapi ArrayList memiliki overhead method call dan object comparison.

Operasi	Array	ArrayList	Analisis
Insertion (middle)	Sangat lambat	Relatif lebih cepat	Keduanya O(n), tetapi ArrayList menggunakan `System.arraycopy()` internal yang teroptimasi; array memerlukan alokasi baru.
Deletion (middle)	Sangat lambat	Lebih efisien	ArrayList hanya menggeser elemen; array harus membuat salinan penuh.
Sorting	Tidak diuji (asumsi already sorted)	Sangat efisien	`Collections.sort()` pada ArrayList sangat optimal untuk data nyata.
Memory usage	Lebih hemat	Lebih boros	Array menyimpan data primitif; ArrayList menyimpan reference objects + internal capacity overhead.

> **Catatan**:
>
> - **Array** unggul dalam **random access (O(1))** dan **memory efficiency**.
> - **ArrayList** unggul dalam **dynamic resizing** dan **developer productivity**.

---

3. Kesimpulan: Kelebihan dan Kekurangan

Array

Kelebihan:

- **Random access** dalam O(1).
- **Memory efficient** untuk tipe primitif (`int`, `double`, dll).
- Cocok untuk **read-heavy** workloads dan data **fixed-size**.
- Tidak ada **autoboxing** atau **object overhead**.

Kekurangan:

- **Fixed size** — tidak mendukung dynamic resizing.
- Operasi **insert/delete** memerlukan **O(n) time** dan **O(n) extra space**.
- Tidak memiliki built-in methods seperti `sort()`, `contains()`, dll.

ArrayList

Kelebihan:

- **Dynamic size** — kapasitas otomatis bertambah saat diperlukan.
- Menyediakan banyak **built-in methods**: `add()`, `remove()`, `indexOf()`, `sort()`, dsb.
- Integrasi sempurna dengan **Java Collections Framework**.
- Lebih aman dan mudah digunakan dalam pengembangan aplikasi.

Kekurangan:

- **Memory overhead** karena penyimpanan sebagai `Object[]` (misal: `Integer` menggantikan `int`).

- Sedikit **performance penalty** pada traversal dan search karena autoboxing.
- Operasi **insert/delete di tengah** tetap O(n) karena perlu **shifting elements**.

Rekomendasi Penggunaan

Use Case	Recommended Structure
Fixed-size data, high-performance access	**Array**
Frequent resizing or unknown final size	**ArrayList**
Primitive data with minimal memory footprint	**Array**
Rapid development & code maintainability	**ArrayList**
Need sorting, searching, or collection ops	**ArrayList**

Kesimpulan Akhir

Tidak ada struktur data yang "terbaik secara universal". **Array** lebih cocok untuk skenario **low-level**, **performance-critical**, dan **memory-constrained**. Sementara itu, **ArrayList** lebih ideal untuk aplikasi umum yang memprioritaskan **fleksibilitas**, **kemudahan penggunaan**, dan **integrasi dengan ekosistem Java**. Pemilihan harus didasarkan pada **requirements spesifik**, **trade-off antara performa dan produktivitas**, serta **karakteristik data**.