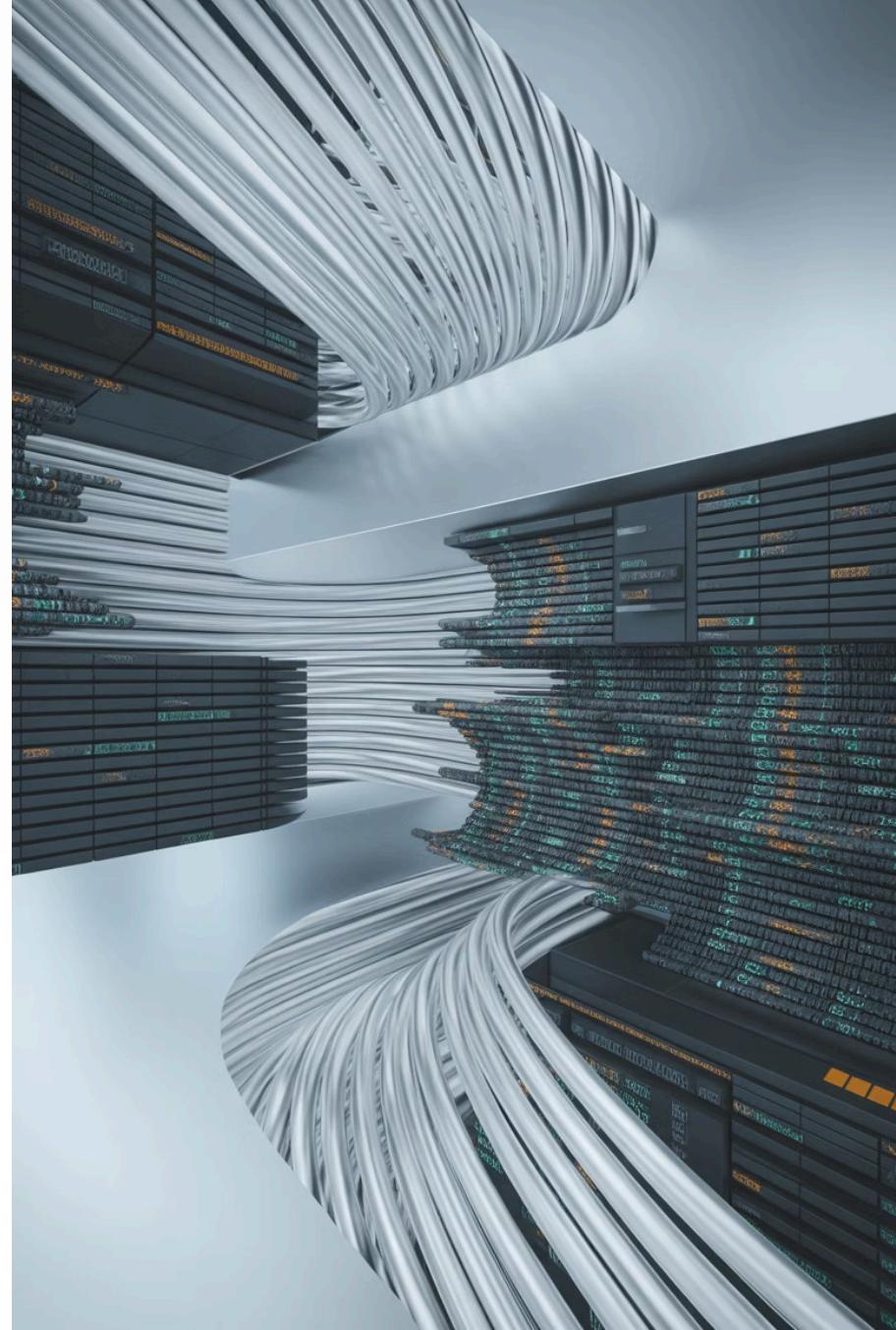


Java : Maîtriser les Collections et les Maps



L'écosystème des Collections Java

Le framework Collections constitue un pilier fondamental de Java, offrant des structures de données puissantes et flexibles.

Architecture hiérarchique claire : Collection comme interface racine, puis List, Set et Queue comme spécialisations.



Les Quatre Piliers



List

Collections ordonnées permettant les doublons. Accès par index.



Set

Collections sans doublons. Garantissent l'unicité des éléments.



Queue

Collections pour traitement FIFO ou prioritaire.



Map

Associations clé-valeur. Accès rapide par clé.

Anatomie d'une List

01

Ordre Préservé

Les éléments maintiennent leur ordre d'insertion.

02

Accès par Index

Récupération directe via position numérique.

03

Doublons Autorisés

Plusieurs occurrences du même élément possibles.





ArrayList : La List par Défaut

Implémentation basée sur un tableau dynamique. Accès en temps constant $O(1)$ pour la lecture par index.

Redimensionnement automatique : nouveau tableau 1.5x plus grand lors du dépassement de capacité.

Insertions/suppressions au milieu : $O(n)$ car déplacement de tous les éléments suivants.

$O(1)$

Accès par index

Temps constant pour `get(i)`

$O(n)$

Insertion milieu

Linéaire pour `add(i, element)`

$O(1)^*$

Ajout en fin

Amorti pour `add(element)`

Bonnes Pratiques ArrayList

```
// Bonnes pratiques ArrayList
```

```
List names = new ArrayList<>();
```

```
// Spécifier la capacité initiale si connue
```

```
List largeList = new ArrayList<>(1000);
```

```
// Utiliser l'interface List comme type
```

```
List products = new ArrayList<>();
```

```
// Éviter de redimensionner fréquemment
```

```
products.ensureCapacity(expectedSize);
```



LinkedList : Quand l'Utiliser

1

Insertions/Suppressions Fréquentes

Excellente quand vous modifiez souvent le milieu de la collection.

2

Pas d'Accès par Index

Éviter si vous avez besoin d'accès aléatoire fréquent.

3

Implémentation Queue/Deque

Idéale pour files d'attente avec opérations aux extrémités.

4

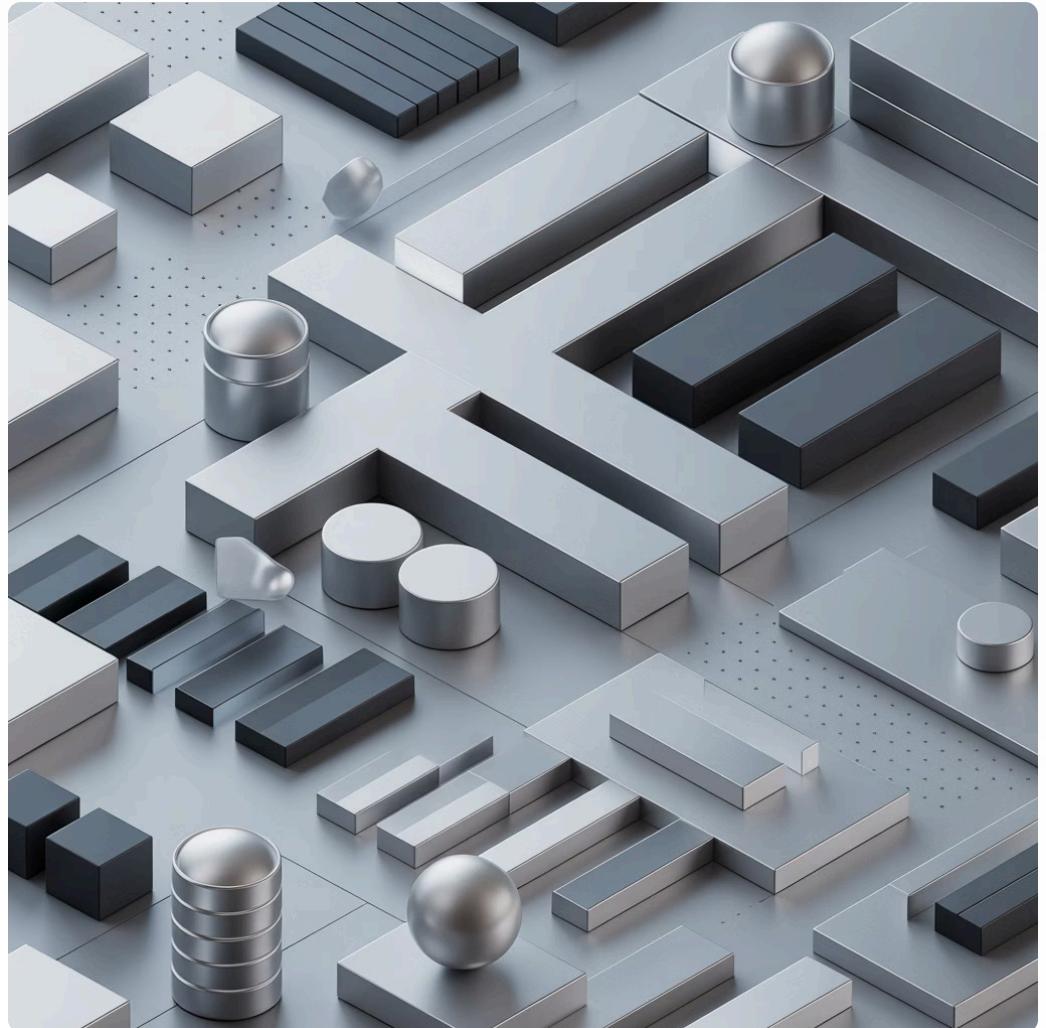
Itération Séquentielle

Parfait pour parcours linéaire complet.

Set : L'Unicité Avant Tout

Caractéristiques Clés

- Aucun élément en double accepté
- Pas d'accès par index
- Opérations ensemblistes efficaces
- Idéal pour tests d'appartenance
- Parfait pour éliminer les doublons





HashSet : Performance et Simplicité

Implémentation la plus performante : opérations en temps constant O(1) moyen pour ajout, suppression et vérification.

O(1)

Ajout moyen

Temps constant pour
add()

O(1)

Contient moyen

Temps constant pour
contains()

O(1)

Suppression
moyenne

Temps constant pour
remove()

HashSet : Code Exemple

```
// Utilisation correcte de HashSet
Set uniqueNames = new HashSet<>();
uniqueNames.add("Alice");
uniqueNames.add("Bob");
uniqueNames.add("Alice"); // N'ajoute rien

// Éliminer les doublons d'une liste
List numbersWithDuplicates =
    Arrays.asList(1, 2, 2, 3, 3, 3);
Set uniqueNumbers =
    new HashSet<>(numbersWithDuplicates);

// Test d'appartenance rapide
if (uniqueNames.contains("Alice")) {
    // O(1) en moyenne
}
```

TreeSet : L'Ordre Naturel



Tri Automatique

Maintient les éléments dans l'ordre naturel ou selon un Comparator personnalisé.

Navigation Efficace

Méthodes first(), last(), lower(), higher() pour naviguer.

Vues de Sous-ensembles

headSet(), tailSet(), subSet() pour créer des vues sur des plages.

LinkedHashSet : Le Meilleur des Deux Mondes

Combine HashSet avec préservation de l'ordre d'insertion. Table de hachage + liste doublement chaînée.



Cas d'Usage Typiques

Élimination de doublons avec ordre préservé



Traitement de Logs

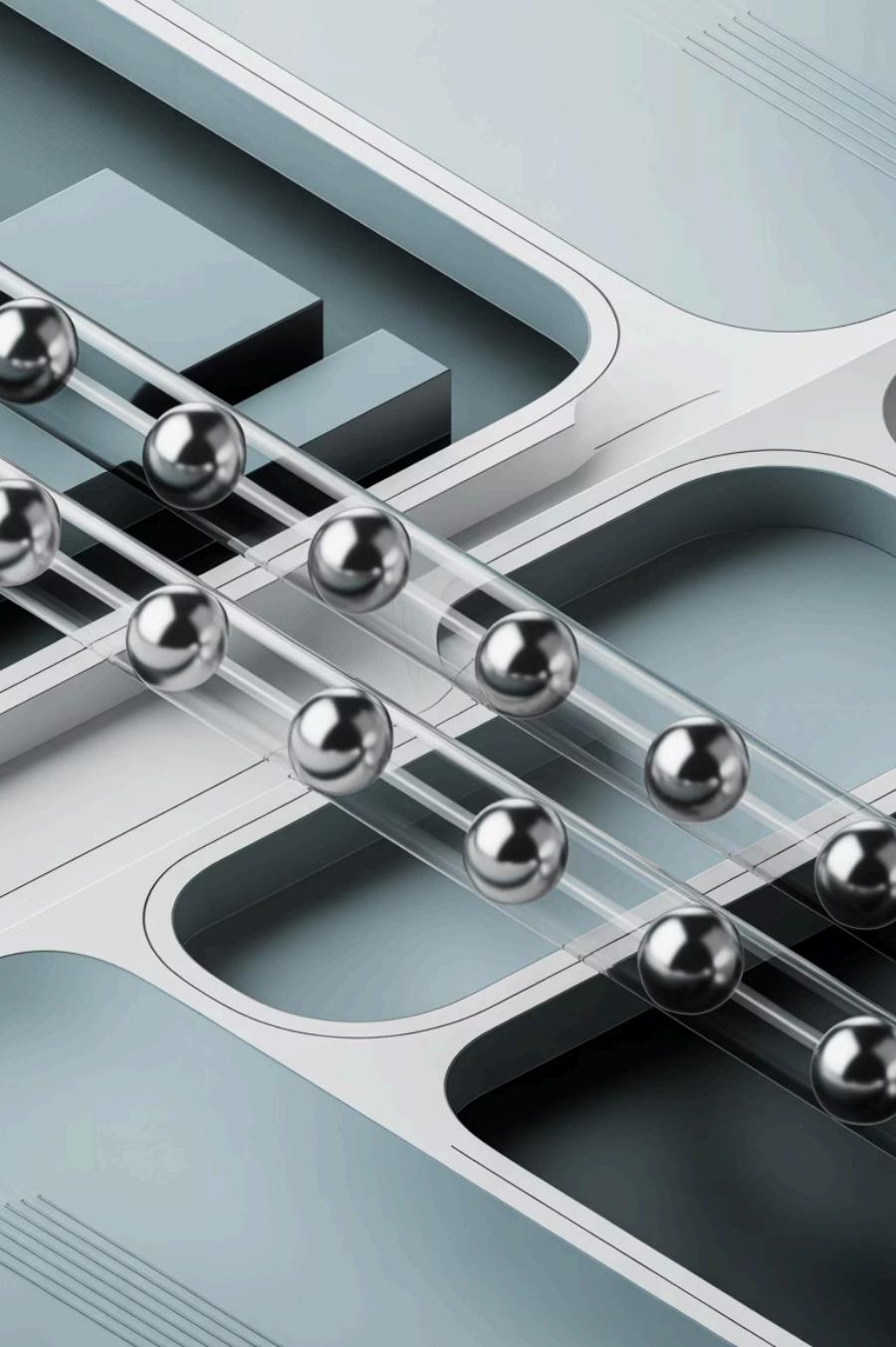
Logs séquentiels et historiques uniques



Cache avec Ordre

Maintien d'ordre d'insertion prévisible

```
// LinkedHashSet préserve l'ordre
Set orderedSet = new LinkedHashSet<>();
orderedSet.add("Premier");
orderedSet.add("Deuxième");
orderedSet.add("Troisième");
// Itération dans l'ordre d'insertion
```



Queue : Gestion de Files d'Attente



Insertion

`add()` / `offer()` ajoutent en queue

Examen

`element()` / `peek()` consultent la tête

Retrait

`remove()` / `poll()` retirent de la tête



PriorityQueue : File à Priorité

File où les éléments sont ordonnés selon leur priorité. Utilise un tas binaire : $O(\log n)$ pour insertion et retrait.

L'élément en tête est toujours le plus petit selon l'ordre de comparaison.

Invaluable pour algorithmes comme Dijkstra, ordonnanceurs de tâches, systèmes d'événements.

$O(\log..)$

Insertion

Logarithmique pour offer()

$O(1)$

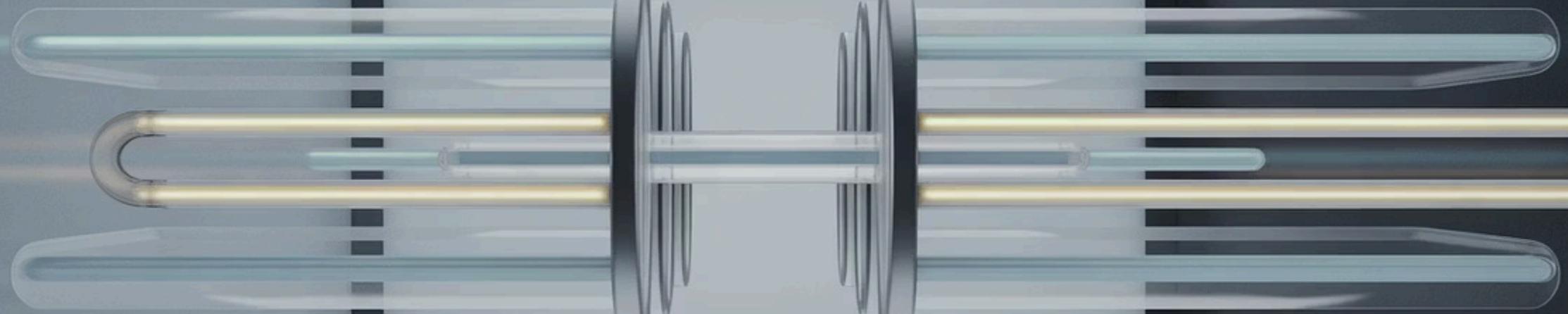
Peek

Constant pour peek()

$O(\log..)$

Retrait

Logarithmique pour poll()



Deque : File à Double Entrée

Pile LIFO

`push()` et `pop()` pour comportement de pile. Alternative moderne à Stack.

Accès Double

Insertion et retrait efficaces aux deux extrémités simultanément.

File FIFO

`offer()` et `poll()` pour comportement de queue classique.

Map : Associations Clé-Valeur

Opérations Essentielles

- `put(key, value)` : Ajouter ou remplacer
- `get(key)` : Récupérer la valeur
- `remove(key)` : Supprimer l'entrée
- `containsKey(key)` : Tester la présence
- `keySet()` : Obtenir toutes les clés
- `values()` : Obtenir toutes les valeurs
- `entrySet()` : Obtenir les paires





HashMap : La Map Haute Performance

Implémentation la plus utilisée et performante. Table de hachage offrant O(1) moyen pour put(), get() et remove().

```
// Déclaration et utilisation de HashMap  
Map userMap = new HashMap<>();  
  
// Ajouter des entrées  
userMap.put("alice", new User("Alice", 30));  
userMap.put("bob", new User("Bob", 25));  
  
// Récupérer une valeur  
User user = userMap.get("alice");  
  
// Utiliser getOrDefault pour éviter null  
User unknown = userMap.getOrDefault(  
    "unknown", DEFAULT_USER);
```

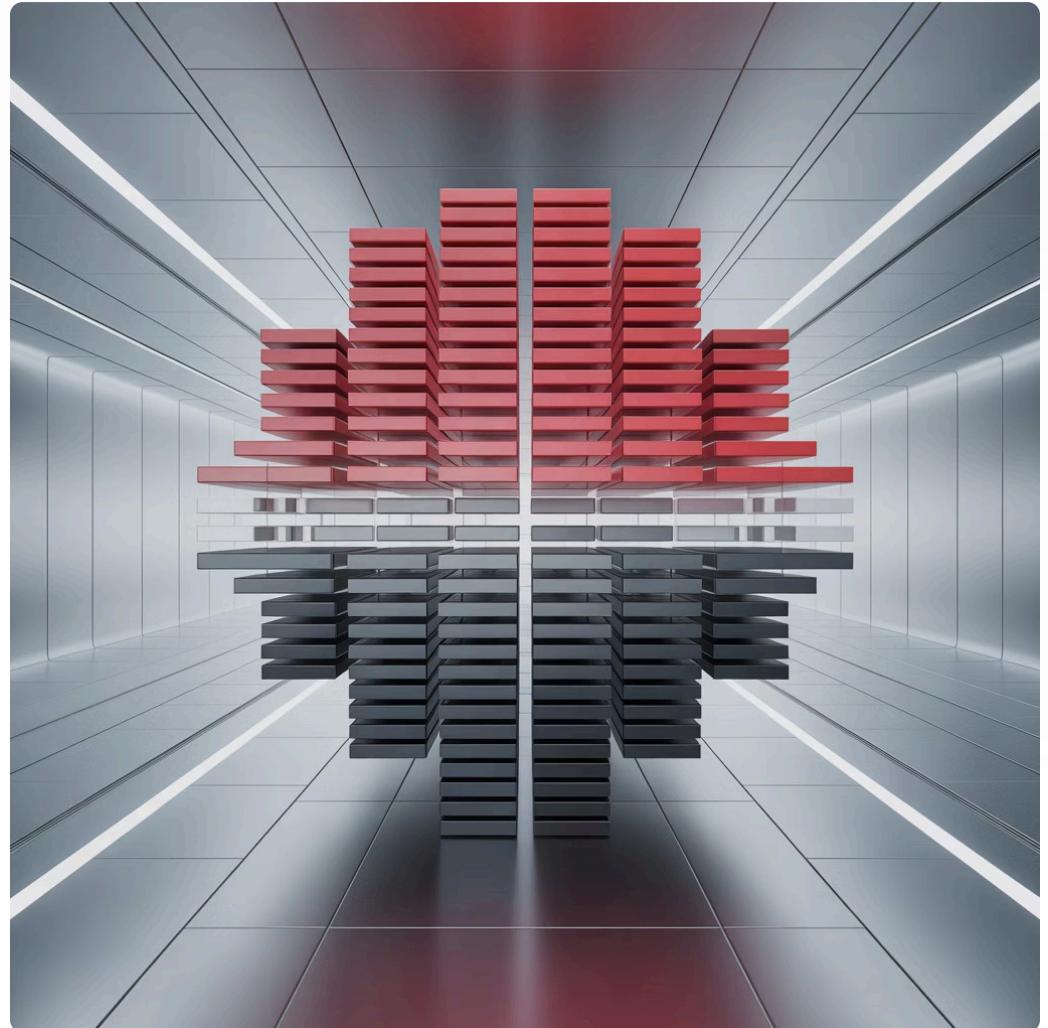
TreeMap : Clés Triées

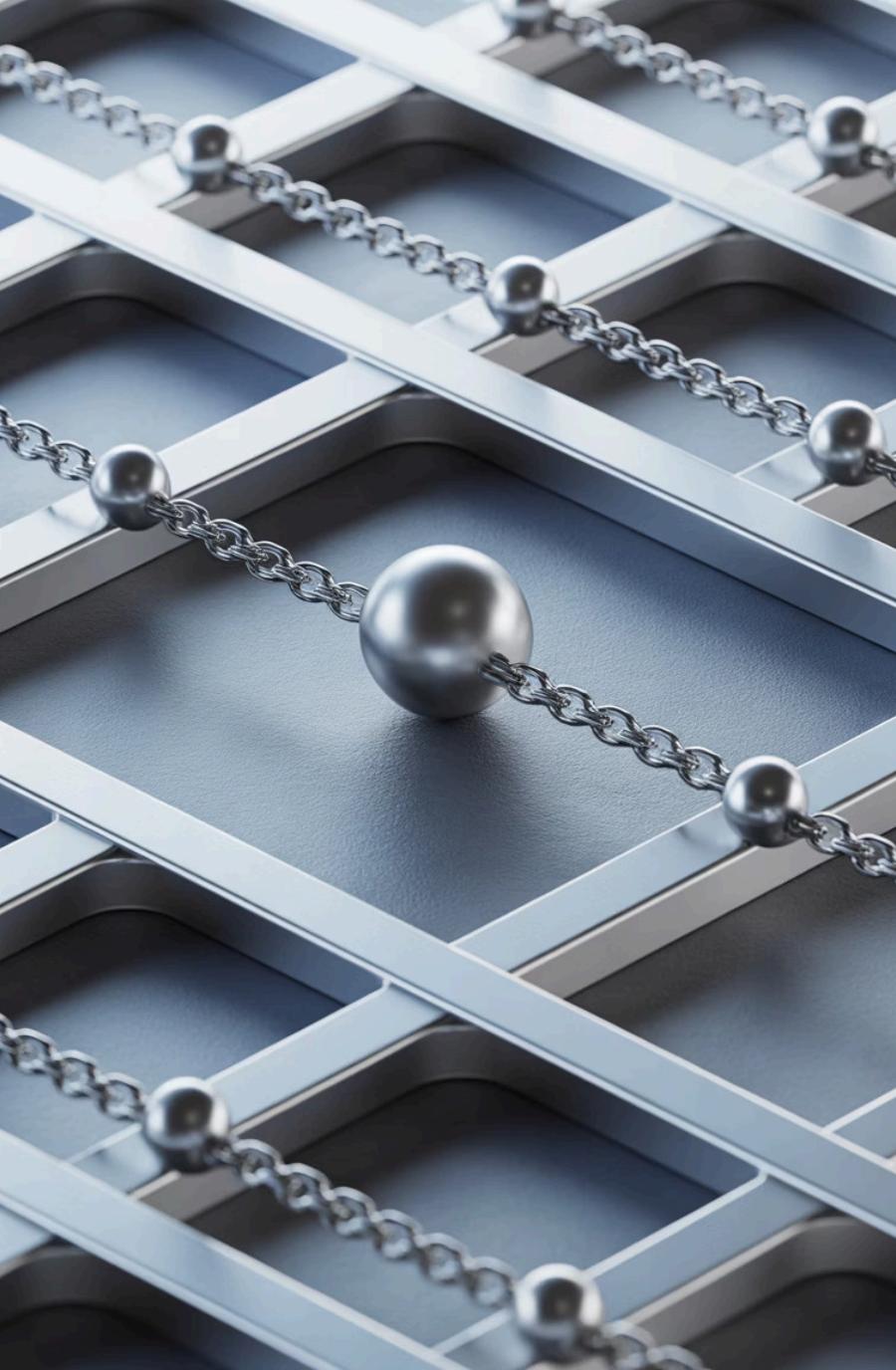
Maintient les clés dans un ordre trié via arbre rouge-noir.

Opérations en $O(\log n)$.

Méthodes de navigation riches : `firstKey()`, `lastKey()`, `lowerKey()`,
`higherKey()`.

Vues sur sous-ensembles : `headMap()`, `tailMap()`, `subMap()`.





LinkedHashMap : Ordre Prévisible

Combine HashMap avec liste doublement chaînée pour préserver l'ordre d'insertion.

Ordre d'Insertion

Itération prévisible dans l'ordre d'ajout des entrées.

Cache LRU

Configuration avec `accessOrder=true` pour implémenter un cache Least Recently Used.

Performance

Quasi- $O(1)$ comme HashMap avec ordre déterministe.

L'Arbre de Décision

Choisir la Bonne Collection



Identifiez Votre Besoin

Clé-valeur → Map. Éléments uniques → Collection. File d'attente → Queue.



Doublons Autorisés ?

Oui → List. Non → Set. Ordre de traitement → Queue.



Ordre Important ?

Insertion → ArrayList/LinkedHashSet. Trié → TreeSet/TreeMap.



Opérations Dominantes

Lectures → ArrayList. Insertions → LinkedList. Lookups → HashSet/HashMap.



Exigences Spéciales

Thread-safety → Collections concurrentes. Priorité → PriorityQueue.

Pattern : Programmation vers l'Interface

✗ Anti-pattern

```
// Trop spécifique  
ArrayList names =  
    new ArrayList<>();  
HashMap users =  
    new HashMap<>();  
  
// Difficile à changer  
public void process(  
    ArrayList items) {  
    // ...  
}
```

✓ Bonne Pratique

```
// Interface générale  
List names =  
    new ArrayList<>();  
Map users =  
    new HashMap<>();  
  
// Flexible  
public void process(  
    List items) {  
    // Accepte toute List  
}
```

Pattern : Collections Immuables

Collections qui ne peuvent être modifiées après création. Thread-safety automatique, absence d'effets de bord.

```
// Collections immuables (Java 9+)
List immutableList =
    List.of("un", "deux", "trois");
Set immutableSet =
    Set.of(1, 2, 3, 4, 5);
Map immutableMap = Map.of(
    "un", 1,
    "deux", 2,
    "trois", 3
);

// Vues immuables de collections existantes
List unmodifiableView =
    Collections.unmodifiableList(mutableList);

// Copie immuable pour véritable immutabilité
List trulyImmutable =
    List.copyOf(mutableList);
```

Pattern : Initialisation avec Capacité

ArrayList

```
// Capacité initiale  
List list =  
    new ArrayList<>(1000);  
  
// Augmenter capacité  
list.ensureCapacity(5000);
```

HashMap

```
// Capacité et load factor  
Map map =  
    new HashMap<>(133, 0.75f);  
  
// Pour 100 éléments:  
// 100 / 0.75 = 133
```

HashSet

```
// Capacité initiale  
Set set =  
    new HashSet<>(133);  
  
// Évite redimensionnement  
// pour ~100 éléments
```

Spécifier la capacité initiale évite des redimensionnements coûteux. Impact notable pour grandes collections.

Pattern : Stream API et Collections

01

Création du Stream

list.stream() ou collection.parallelStream()

02

Opérations Intermédiaires

filter(), map(), sorted(), distinct() - lazy
evaluation

03

Opération Terminale

collect(), forEach(), reduce(), count() -
déclenche le traitement

```
// Filtrage et transformation
List names = users.stream()
    .filter(user -> user.getAge() > 18)
    .map(User::getName)
    .collect(Collectors.toList());
```

```
// Groupement
Map<Employee, List<Employee>> byDept =
    employees.stream()
        .collect(Collectors.groupingBy(
            Employee::getDepartment));
```

Anti-pattern : Mauvais Choix de Collection

ArrayList pour Lookups

Utiliser `contains()` sur une grande `ArrayList` est $O(n)$. Utilisez `HashSet` pour $O(1)$.

List pour Unicité

Vérifier manuellement les doublons dans une `List`. Utilisez `Set` directement.

TreeSet sans Besoin

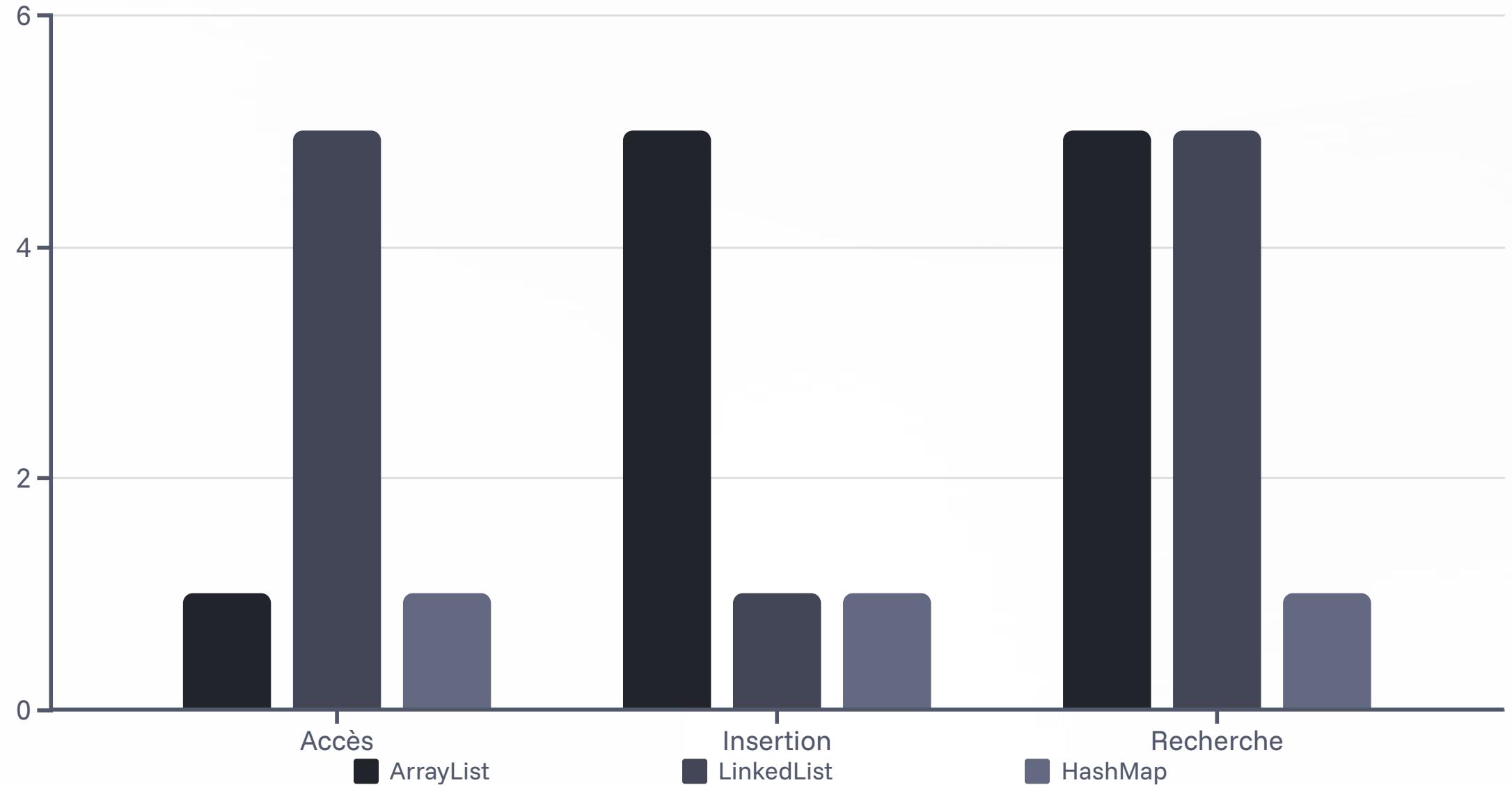
Payer le coût du tri quand l'ordre n'importe pas. `HashSet` est plus rapide.

Vector/Hashtable

Classes legacy synchronisées obsolètes. Utilisez `ArrayList/HashMap`.

Performance : Complexité Algorithmique

Comprendre la complexité Big O est crucial pour écrire du code performant.





Thread-Safety : Collections Concurrentes

Collections standard non thread-safe. Pour environnements multi-threadés, utilisez `java.util.concurrent`.



ConcurrentHashMap

Alternative thread-safe à `HashMap`. Verrouillage fin au niveau des segments.



CopyOnWriteArrayList

Idéale pour lectures fréquentes, écritures rares. Copie à chaque modification.



BlockingQueue

Essentielle pour patterns producteur-consommateur. Opérations bloquantes.

Cas d'Usage : Cache LRU Simple

LinkedHashMap parfaite pour implémenter un cache Least Recently Used avec éviction automatique.

```
// Cache LRU simple avec LinkedHashMap
public class LRUCache
    extends LinkedHashMap {
    private final int maxSize;

    public LRUCache(int maxSize) {
        super(16, 0.75f, true); // accessOrder
        this.maxSize = maxSize;
    }

    @Override
    protected boolean removeEldestEntry(
        Map.Entry eldest) {
        return size() > maxSize;
    }
}

// Utilisation
LRUCache cache =
new LRUCache<>(100);
```

Bonnes Pratiques : Checklist Finale

1

Choisir la Bonne Structure

Analyser opérations dominantes, besoins d'ordre et doublons.

2

Programmer vers l'Interface

Déclarer avec List, Set, Map plutôt que les implémentations.

3

Initialisation Appropriée

Spécifier capacité initiale. Retourner collections vides plutôt que null.

4

Immutabilité et Thread-Safety

Préférer collections immuables. Utiliser collections concurrentes.

5

Itération Sécurisée

Utiliser Iterator.remove() ou removelf(). Privilégier Streams.

6

Clés et Égalité

Objets immuables comme clés. Implémenter hashCode() et equals() ensemble.



Maîtriser les Collections Java



Choix Éclairés

Comprendre les compromis pour sélectionner la structure optimale.



Code Robuste

Éviter les anti-patterns et pièges courants.



Performance

Optimiser avec connaissance des complexités algorithmiques.



Expressivité

Utiliser les collections pour communiquer l'intention.

La maîtrise des collections est un voyage continu. Les fondations solides acquises ici vous serviront tout au long de votre carrière.