Java Data Structures

Collection Selection Exams

Java Collection Framework — SE

6 exam versions · 4 questions each

Each team must justify their choice of data structure

General Instructions

What should you do?

For each question in your exam you must:

1. Read the scenario carefully.
2. Choose the most suitable data structure from the Java Collections Framework.
3. Justify your choice by addressing: the complexity of key operations (O(1), O(log n), O(n)), why you discard the other options, and the trade-offs of your choice.
4. Write a short Java code snippet demonstrating how you would use the structure.

Grading Rubric

Correctness of choice (25%) — The chosen structure actually solves the problem.

Quality of justification (40%) — Correct complexity analysis, alternatives discarded with technical reasoning.

Java code snippet (25%) — Correct code that compiles and demonstrates the described operations.

Creativity / alternative solution (10%) — Valid proposal combining two structures with solid justification.

★ Excercise 3 ★

Name(s): Clay Gutierrez, Luis Díaz, Emilio Hernández y Alfredo Carmona Team: 43 Date: 28/02/26

Question 1 | Server Error Log

📋 **Scenario:**

A web server needs to maintain a log of recent errors. The system must:

- Append errors constantly to the end (high frequency)
- Read the entire log periodically for analysis
- Keep only the last 10,000 errors (circular buffer)
- No insertions or deletions in the middle

❓ **Question:**

For the underlying collection, ArrayList or LinkedList? Justify considering that all insertions happen at the end and reads are sequential.

💡 *Hint: Which one has better memory locality for sequential reads?*

✏️ **Answer & Justification:**

Chosen structure: ArrayList

Justification:

Aunque el LinkedList ofrece inserciones O(1), se descarta esta opción porque sus nodos dispersos tienen una pobre localidad de memoria, y la creación constante de nodos generaría sobrecarga en el sistema. Por otro lado, un ArrayList almacena sus elementos en ubicaciones de memoria contiguas, lo que proporciona una excelente localidad de memoria y hace que las lecturas secuenciales periódicas (traversal) sean significativamente más rápidas.

Al implementar el ArrayList como un buffer circular de 10,000 elementos, podemos pre-asignar su capacidad. Esto nos permite usar un índice circular para sobrescribir los datos más antiguos, logrando inserciones constantes en tiempo O(1) y evitando el trade-off habitual del ArrayList (el costo de redimensionamiento). Las lecturas secuenciales tomarán un tiempo O(n), pero altamente optimizado por la arquitectura del caché.

Java code snippet:

Java

```java
import java.util.ArrayList;
import java.util.List;


// Clase que maneja la lógica del buffer (5 elementos)
class ErrorLogBuffer {
   private final int MAX_CAPACITY = 5;
   private List<String> errorLog;
   private int currentIndex;
```

```
    private boolean isFull;

    public ErrorLogBuffer() {
        // Pre-asignamos la capacidad
        errorLog = new ArrayList<>(MAX_CAPACITY);
        currentIndex = 0;
        isFull = false;
    }

    // O(1) Inserción constante
    public void addError(String errorMessage) {
        if (!isFull && errorLog.size() < MAX_CAPACITY) {
            // Si no está lleno, simplemente agregamos al final
            errorLog.add(errorMessage);
        } else {
             // Si ya llegamos a MAX_CAPACITY, sobrescribimos el elemento más
antiguo
            isFull = true;
            // Se sobreescribe el dato más antiguo
            errorLog.set(currentIndex, errorMessage);
        }
        // Se actualiza el índice de forma circular
        currentIndex = (currentIndex + 1) % MAX_CAPACITY;
    }
     // O(n) Lectura secuencial rápida gracias a la localidad de memoria del
ArrayList
    public List<String> getSequentialLog() {
        List<String> orderedLog = new ArrayList<>(errorLog.size());

        if (!isFull) {
            return new ArrayList<>(errorLog);
        }
          // Si el buffer ha dado la vuelta, leemos primero desde el índice
actual hasta el final
        for (int i = currentIndex; i < MAX_CAPACITY; i++) {
            orderedLog.add(errorLog.get(i));
```

```
        }
        // Y luego desde el principio hasta el índice actual
        for (int i = 0; i < currentIndex; i++) {
            orderedLog.add(errorLog.get(i));
        }

        return orderedLog;
    }
}

}
```

Question 2 | Profanity Filter for a Chat App

📋 Scenario:

A moderation system needs to check in real time whether a message contains banned words. The system must:

- Load 50,000 banned words at startup
- Check whether a word is banned with the lowest possible latency
- The word list never changes during runtime
- Millions of checks per second

❓ Question:

Which Set implementation gives the best lookup performance? Rule out the other two with clear technical reasoning.

💡 *Hint: 50,000 fixed words, millions of lookups. Order is completely irrelevant.*

✏️ Answer & Justification:

Chosen structure: HashSet

Justification:

La implementación ideal es el HashSet , ya que ofrece el mejor rendimiento de búsqueda con una complejidad de tiempo promedio de O(1) al almacenar elementos únicos donde el orden no importa. Descartamos el TreeSet porque, al mantener los elementos en un orden alfabético, su tiempo de búsqueda es de O(log n), lo cual es matemáticamente más lento y crearía un cuello de botella inaceptable para "millones de revisiones por segundo". También descartamos el LinkedHashSet ; aunque su búsqueda es rápida, está diseñado para preservar el orden de inserción, lo que añade una sobrecarga de memoria y procesamiento totalmente innecesaria dado que el orden de las palabras prohibidas es irrelevante. El principal *trade-off* de la elección es que el HashSet no garantiza ningún orden al iterar y consume más memoria que una simple lista para mantener sus accesos directos, pero es un costo necesario y perfectamente justificado para obtener la latencia ultra baja que exige este sistema en tiempo real.

Java code snippet:

Java

```java
import java.util.HashSet;
import java.util.List;
import java.util.Set;

class ModerationSystem {
   private Set<String> bannedWords;
```

```
    // El constructor recibe la lista de palabras y las carga en el HashSet
al inicio
    public ModerationSystem(List<String> initialWords) {
        bannedWords = new HashSet<>(initialWords);
    }

    // Método con complejidad O(1) para buscar la palabra al instante
    public boolean isBanned(String word) {
        // La función .contains() en un HashSet es extremadamente rápida
        return bannedWords.contains(word.toLowerCase());
    }
}
```

Question 3 | Browser Back & Forward Navigation

📋 Scenario:

You want to implement browser navigation history. The user can visit new pages, go back, and go forward. The system must:

- Remember visited pages
- Navigate backward (undo last navigation)
- Navigate forward (if the user went back)
- Clear the forward history when a new page is visited

❓ Question:

Would you use Stack, Queue, or Deque? Which concrete implementation? Model how you would manage the back and forward history with your chosen structure.

💡 *Hint: You need to operate at both ends — or use two separate structures.*

✏️ Answer & Justification:

Chosen structure: Deque

Justification:

Se utilizaría el modelo de dos estructuras separadas, específicamente usando ArrayDeque (que implementa la interfaz Deque) actuando como Stacks. Se requiere un comportamiento LIFO, por lo que se descarta por completo una Queue, ya que su naturaleza FIFO nos llevaría a la página más antigua en lugar de la anterior. El sistema se modelaría con un backStack y un forwardStack. Al visitar una nueva página, la agregamos al backStack y vaciamos el forwardStack. Al navegar hacia atrás, hacemos 'pop' del backStack y 'push' al forwardStack en tiempo constante O(1). Elegir la implementación concreta de ArrayDeque sobre la clase clásica Stack es un trade-off consciente: se sacrifica la seguridad de hilos que Stack hereda de Vector, a cambio de eliminar la sobrecarga de sincronización, logrando el máximo rendimiento posible para la navegación.

Java code snippet:

Java

```java
import java.util.ArrayDeque;
import java.util.Deque;

class BrowserHistory {
    // Utilizamos Deque como si fueran Stacks para mejor rendimiento
    private Deque<String> backStack;
    private Deque<String> forwardStack;
    private String currentPage;
```

```java
    public BrowserHistory() {
        backStack = new ArrayDeque<>();
        forwardStack = new ArrayDeque<>();
        currentPage = "about:blank"; // Página inicial por defecto
    }

    // O(1) - Visitar una nueva página
    public void visit(String url) {
        System.out.println("Visitando nueva página: " + url);
        if (!currentPage.equals("about:blank")) {
             backStack.push(currentPage); // Guardamos la página actual en el historial hacia atrás
        }
        currentPage = url;
            forwardStack.clear(); // Al visitar nueva página, se borra el historial hacia adelante
    }

    // Ir hacia atrás
    public void goBack() {
        if (backStack.isEmpty()) {
            System.out.println("No hay historial hacia atrás.");
            return;
        }
           forwardStack.push(currentPage); // Guardamos la actual en forward antes de retroceder
        currentPage = backStack.pop();  // Sacamos la última visitada
        System.out.println("<- Navegando hacia atrás a: " + currentPage);
    }

    // Ir hacia adelante
    public void goForward() {
        if (forwardStack.isEmpty()) {
            System.out.println("No hay historial hacia adelante.");
            return;
        }
```

```
        backStack.push(currentPage);      // Guardamos la actual en back antes
de avanzar
        currentPage = forwardStack.pop(); // Recuperamos la página de forward
        System.out.println("-> Navegando hacia adelante a: " + currentPage);
    }

    public void printCurrentPage() {
        System.out.println("Página actual: [" + currentPage + "]");
    }
}
```

Question 4 | Word Frequency Counter

📋 Scenario:

A text analyzer needs to count how many times each word appears in a 1-million-word document. At the end it must:

- Display words sorted alphabetically with their frequency
- Update counts efficiently during the pass
- Handle a vocabulary of 10,000 to 100,000 unique words

❓ Question:

Would you use HashMap or TreeMap to accumulate counts? What if you also needed to display words sorted by frequency (highest to lowest)? Would you change your structure?

💡 *Hint: When you need the order matters: during counting, or only at the end?*

✏️ Answer & Justification:

Chosen structure: HashMap

Justification:

Para acumular los conteos de las palabras, la mejor opción es utilizar un HashMap, ya que esta estructura está diseñada para almacenar pares clave-valor donde se requieren búsquedas rápidas, brindando una complejidad de O(1) para el millón de operaciones de actualización. Se descarta el TreeMap para la fase de conteo porque almacena pares clave-valor donde las claves necesitan estar ordenadas, lo que haría que cada inserción tomara un tiempo de O(log n), creando un cuello de botella masivo dado que el orden solo nos importa hasta el final. El *trade-off* de esta elección es que los datos terminan desordenados, pero ordenar un vocabulario final de 100,000 palabras es mucho más eficiente que ralentizar un millón de lecturas. Si además se necesitara ordenar por frecuencia, no se cambiaría esta estructura inicial; se seguiría usando el HashMap por velocidad de inserción, y al final se pasarían los resultados a un PriorityQueue, el cual asegura que los elementos sean procesados en base a su prioridad (en este caso, la frecuencia de las palabras).

Java code snippet:

Java

```java
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;
import java.util.PriorityQueue;

class TextAnalyzer {
   // HashMap para inserciones y actualizaciones en O(1)
   private Map<String, Integer> wordCounts;

   public TextAnalyzer() {
```

```
        wordCounts = new HashMap<>();
    }

    // Método para simular la lectura del documento
    public void processDocument(String[] document) {
        for (String word : document) {
            word = word.toLowerCase(); // Normalizamos la palabra
            // Si la palabra ya existe, sumamos 1. Si no, la inicializamos en 1.
            wordCounts.put(word, wordCounts.getOrDefault(word, 0) + 1);
        }
    }

    // Imprimir en orden alfabético usando TreeMap (O(k log k))
    public void printAlphabetically() {
        // Al pasar el HashMap al TreeMap, este ordena automáticamente las claves alfabéticamente
        Map<String, Integer> sortedAlphabetically = new TreeMap<>(wordCounts);

        System.out.println("--- Palabras en Orden Alfabético ---");
        for (Map.Entry<String, Integer> entry : sortedAlphabetically.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
    }

    // Imprimir por frecuencia de mayor a menor usando PriorityQueue
    public void printByFrequency() {
        // PriorityQueue ordenará las entradas basándose en el valor (la frecuencia) de mayor a menor
        PriorityQueue<Map.Entry<String, Integer>> queue = new PriorityQueue<>(
            (a, b) -> b.getValue().compareTo(a.getValue())
        );
```

```
        // Agregamos todas las entradas a la cola de prioridad
        queue.addAll(wordCounts.entrySet());

          System.out.println("--- Palabras Ordenadas por Frecuencia (Mayor a
Menor) ---");
        // Extraemos e imprimimos hasta que la cola esté vacía
        while (!queue.isEmpty()) {
            Map.Entry<String, Integer> entry = queue.poll();
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
    }
}
```