Taller 7: Implementación algoritmos de búsqueda

Datos generales

Integrantes **Grupo H**	Wilson Palma Miguel Veintimilla Darwin Correa Jostin Vasquez
Asignatura	Estructura de datos
Ciclo	3 A
Unidad	2
Resultado de aprendizaje de la unidad	Aplica los métodos de ordenación y búsqueda en la resolución de problemas, bajo los principios de solidaridad, transparencia, responsabilidad y honestidad.
Título de la Práctica	Búsqueda en Java: Secuencial y Binaria
Nombre del Docente	Andrés Roberto Navas Castellanos
Fecha	Jueves 27 de noviembre
Horario	07h30 – 10h30
Lugar	Aula
Tiempo planificado en el Sílabo	3 horas

Objetivos

1. Implementar correctamente las variantes canónicas de búsqueda secuencial y búsqueda binaria en Java.
2. Validar con casos borde y justificar cuándo aplicar cada método según la estructura de datos (arreglo vs SLL).
3. Documentar la implementación y presentar evidencias reproducibles (README + tabla de evidencias en Markdown).

Materiales y herramientas

- Conjunto de datasets proporcionados (ver sección “Datasets usados”).
- JDK (OpenJDK 11+ recomendado).
- IDE: Visual Studio Code o IntelliJ IDEA Community.
- Control de versiones: Git / GitHub.
- Documentación: README en Markdown y el respectivo informe.

Metodología (procedimiento)

Se organiza la práctica en pasos numerados que corresponden a los requisitos del taller. Para cada paso se solicita implementación tanto para **arrays** como para **SLL (SimpleLinkedList)** cuando aplique.

Paso 1 — Primera ocurrencia

- **Arrays:** implementar indexOfFirst(int[] a, int key) que retorne el índice de la primera ocurrencia o -1 si no existe.
- **SLL:** implementar Node findFirst(Node head, int key) que retorne el nodo de la primera ocurrencia o null.
- **Casos borde:** arreglo vacío; arreglo con un elemento; duplicados (posición 0, medio, último).

Paso 2 — Última ocurrencia

- **Arrays:** dos alternativas (recorrer de atrás hacia adelante, o una pasada guardando last actualizado). Implementar indexOfLast(int[] a, int key).
- **SLL:** recorrer la lista y mantener last como nodo de la última coincidencia; implementar Node findLast(Node head, int key).
- **Casos borde:** ninguna aparición; todas las posiciones coinciden; tamaños muy pequeños.

Paso 3 — findAll por predicado

- **Arrays:** List<Integer> findAll(int[] a, IntPredicate p) que retorne lista de índices que cumplen p.
- **SLL:** List<Node> findAll(Node head, Predicate<Node> p) que retorne la lista de nodos que cumplen el predicado (por ejemplo: par, ==key, < umbral).
- **Salida esperada:** índices (para arrays) y nodos (para SLL) en orden de aparición.

Paso 4 — Secuencial con centinela (solo arrays)

- Técnica:
  1. Guardar el último elemento en variable last.
  2. Escribir key en a[n-1] (colocar centinela).
  3. Iterar con i = 0; while (a[i] != key) i++;.
  4. Restaurar a[n-1] = last.
  5. Si i < n-1 o last == key entonces fue un hallazgo real; si no, no encontrado.
- **Objetivo:** comparar número de comparaciones contra la variante clásica que comprueba límites en cada iteración.
- **Medición:** instrumentar el conteo de comparaciones y reportarlo.

Paso 5 — Búsqueda binaria (arrays ordenados)

- Implementar int binarySearch(int[] a, int key) de forma iterativa.
- **Precauciones:**
  - Usar mid = low + (high - low) / 2 para evitar overflow.
  - Verificar precondición: el arreglo debe estar ordenado antes de aplicar binaria.
- **Plus:** lowerBound(int[] a, int key) (primera posición >= key) y upperBound(int[] a, int key) (primera posición > key) para tratar duplicados.

Paso 6 — Pruebas y verificación

- Ejecutar SearchDemo con los datasets indicados y las claves pedidas; registrar índices y conteos.

- **Datasets (proporcionados):**
  - A: [8, 3, 6, 3, 9]
  - B: [5, 4, 3, 2, 1] (inverso)
  - C: [1, 2, 3, 4, 5] (ordenado)
  - D: [2, 2, 2, 2] (duplicados)
- **Claves a probar:** {7, 5, 2, 42} (42 no está presente en ningún dataset).
- **SLL de prueba:** 3 -> 1 -> 3 -> 2 — ejecutar findNode(3), findLastNode(3), findAll(val < 3).

Resultados (entregables)

- ZIP con src/ (implementaciones y SearchDemo).
- README en Markdown (cómo compilar/ejecutar; casos borde; precondiciones).
- Tabla de evidencias en Markdown (tabla con las combinaciones: colección, clave/predicado, método, salida).
- Comparación de número de comparaciones entre secuencial clásico y centinela (Ver seccion al final del documento).

Preguntas de control

1. **¿Por qué la binaria no es adecuada para SLL aunque esté ordenada?** Porque la búsqueda binaria requiere acceso aleatorio O(1) a posiciones (p. ej. acceso a a[mid]). En una SLL llegar a la posición media cuesta O(n) por recorrido secuencial, lo que convierte la binaria en una operación con coste total O(n log n) o peor, perdiendo su ventaja teórica.
2. **En primera ocurrencia, ¿por qué se retorna en cuanto se encuentra?** Para minimizar comparaciones y tiempo: una vez hallada la primera coincidencia no tiene sentido seguir buscando — la especificación del problema solicita la *primera* ocurrencia.
3. **¿Qué garantiza la correctitud de la variante centinela?** El mecanismo asegura terminación del bucle sin comprobar límites: al colocar temporalmente key en a[n-1] garantizamos que while (a[i] != key) termine, luego restauramos el último valor y verificamos si el índice i corresponde a una coincidencia real (i < n-1) o si fue la coincidencia con la centinela (i == n-1 y last != key).
4. **¿Cómo adaptarías la binaria para duplicados (primera/última)?** Usar variantes de binaria que muevan high/low para buscar fronteras:
   - lowerBound: buscar la primera posición >= key (si a[mid] >= key entonces high = mid else low = mid + 1).
   - upperBound: buscar la primera posición > key (si a[mid] <= key entonces low = mid + 1 else high = mid). La primera ocurrencia es lowerBound(key) si a[lowerBound] == key; la última es upperBound(key) - 1.
5. **Dos casos borde que detectaron errores en pruebas:**
   1. Arreglo de longitud 0 (vacío) — implementar y comprobar retornos -1 y evitar accesos a a[-1].
   2. Todos los elementos idénticos (duplicados) — verificar que lowerBound y upperBound funcionan correctamente y que la centinela no falsee resultados cuando key coincide con last.

Criterio de evaluación (rúbrica resumida)

Ítem	Descripción	Puntaje máximo
Secuencial (first/last/findAll)	Correcta implementación, manejo de bordes y claridad	3.0

	de código.	
Centinela (arrays)	Implementación y explicación; comparación de comparaciones.	2.0
Binaria (arrays)	Correcta con cuidado en mid y validación de precondición.	2.5
Evidencias (tabla/README)	Completas y reproducibles.	1.5
Calidad de código	Organización y nombres adecuados.	1.0

Bibliografía

1. P. W. Bible and L. Moser, *An Open Guide to Data Structures and Algorithms*. PALNI Open Press, 2023.
2. OpenDSA Project, "Searching and Sorting Modules," Virginia Tech, 2021–2024.
3. Oracle, "Java SE Documentation: Arrays.binarySearch, Comparator y patrones de búsqueda," 2021–2025.

Tabla de evidencias — Búsquedas en Arrays

La tabla siguiente está basada en la ejecución registrada en salida.txt (salida del SearchDemo). Se incluyen índices de primera/última ocurrencia y conteos de comparaciones para las variantes instrumentadas. N/A indica que la búsqueda binaria no fue aplicada porque el arreglo no estaba ordenado.

Dataset	Array	Key	Primera ocurrencia	Última ocurrencia	Secuencial clásico (índice)	Comparaciones clásico	Secuencial centinela (índice)	Comparaciones centinela	Binaria (índice)	Comparaciones binaria
A	[8, 3, 6, 3, 9]	7	-1	-1	-1	10	-1	5	N/A	N/A
A	[8, 3, 6, 3, 9]	5	-1	-1	-1	10	-1	5	N/A	N/A
A	[8, 3, 6, 3, 9]	2	-1	-1	-1	10	-1	5	N/A	N/A
A	[8, 3, 6, 3, 9]	42	-1	-1	-1	10	-1	5	N/A	N/A
B	[5, 4, 3, 2, 1]	7	-1	-1	-1	10	-1	5	N/A	N/A
B	[5, 4, 3, 2, 1]	5	0	0	0	2	0	1	N/A	N/A

B	[5, 4, 3, 2, 1]	2	3	3	3	8	3	4	N/A	N/A
B	[5, 4, 3, 2, 1]	42	-1	-1	-1	10	-1	5	N/A	N/A
C	[1, 2, 3, 4, 5]	7	-1	-1	-1	10	-1	5	-1	6
C	[1, 2, 3, 4, 5]	5	4	4	4	10	4	5	4	5
C	[1, 2, 3, 4, 5]	2	1	1	1	4	1	2	1	5
C	[1, 2, 3, 4, 5]	42	-1	-1	-1	10	-1	5	-1	6
D	[2, 2, 2, 2]	7	-1	-1	-1	8	-1	4	-1	6
D	[2, 2, 2, 2]	5	-1	-1	-1	8	-1	4	-1	6
D	[2, 2, 2, 2]	2	0	3	0	2	0	1	1	1
D	[2, 2, 2, 2]	42	-1	-1	-1	8	-1	4	-1	6

Tabla de evidencias – SimpleLinkedList

Lista usada: 3 -> 1 -> 3 -> 2

Operación	Argumento	Resultado
findNode	3	Node(3)
findLastNode	3	Node(3)
findAll	val < 3	[Node(1), Node(2)]

Comparación de número de comparaciones: Secuencial clásico vs Centinela

Promedios por dataset calculados sobre las 4 claves probadas (7, 5, 2, 42).
Reducción relativa (%) = 100 × (1 − (Centinela / Clásico)).

Dataset	Promedio comparaciones — Secuencial clásico	Promedio comparaciones — Centinela	Reducción relativa (%)
A ([8, 3, 6, 3, 9])	10.00	5.00	50.0%
B ([5, 4, 3, 2, 1])	7.50	3.75	50.0%
C ([1, 2, 3, 4, 5])	8.50	4.25	50.0%

D ([2, 2, 2, 2])	6.50	3.25	50.0%
Global (todos)	**8.13**	**4.06**	**50.0%**

Interpretación breve:

Con los nuevos datos, la variante con centinela mantiene una ventaja consistente, reduciendo en promedio el número de comparaciones en un **50%** respecto al método secuencial clásico en todos los datasets evaluados. Esta reducción se sostiene tanto cuando la clave está presente como cuando está ausente, evidenciando la eficiencia estructural del enfoque con centinela al evitar comprobaciones adicionales de límite en cada iteración.