# Semana 2

## ACTIVIDAD 1

- Diagrama propio que muestre la fórmula de acceso aplicada a un ejemplo diferente

Dirección(arr[i]) = 5000 + i*4
arr[0] → 5000
arr[1] → 5004
arr[2] → 5008

- Tabla comparativa: operación vs. número de movimientos necesarios para un arreglo de n=10

| OPERACION | MOVIMIENTOS NECESARIOS |
|---|---|
| Insertar al inicio | 10 |
| Insertar en posición 5 | 5 |
| Insertar al final | 0 |
| Eliminar al inicio | 9 |
| Eliminar en posición 5 | 5 |
| Eliminar al final | 0 |

- Responde: ¿Por qué insertar al inicio es más costoso que insertar al final?

Insertar al inicio es más costoso porque todos los elementos deben moverse una posición hacia la derecha**.**
En cambio, insertar al final no requiere mover nada: solo se coloca el nuevo valor en la siguiente dirección libre.

## ACTIVIDAD 2

- Fórmula matemática del costo de n inserciones con incremento de +1

Cada inserción crea un nuevo arreglo y copia todos los elementos anteriores:
C_{+1}(n)=\sum _{k=1}^n(k-1)=\frac{n(n-1)}{2}
Crece de manera cuadrática (O(n^2)).

- Fórmula del costo con duplicación de capacidad

Cuando la capacidad se duplica, solo se copian los elementos en los momentos de expansión. El número total de copias es aproximadamente:
C_{\mathrm{dup}}(n)\leq 2n
Crece de manera lineal (O(n)).

- Informe breve (150 palabras) explicando a un gerente no técnico por qué el sistema era lento

El sistema de MegaStore era lento porque utilizaba una estrategia ineficiente para manejar su catálogo de productos. Cada vez que se agregaba un nuevo producto, el programa creaba un arreglo más grande y copiaba todos los elementos anteriores. Esto significa que el costo de insertar no era constante, sino que aumentaba rápidamente: para n productos, el

número de copias era proporcional a n^2. En la práctica, con 500,000 productos, el sistema realizaba más de 125 mil millones de copias, lo que equivalía a más de 34 horas de procesamiento. En contraste, una estrategia estándar de duplicar la capacidad del arreglo cuando se llena reduce el número de copias a aproximadamente dos por producto, lo que equivale a solo un segundo para la misma cantidad de inserciones. La diferencia es enorme: el diseño original estaba "reordenando toda la tienda" cada vez que llegaba un producto, mientras que la estrategia de duplicación aprovecha la memoria de manera eficiente y mantiene el rendimiento casi constante.

## ACTIVIDAD 3

- Código completo del TAD ArregloDinámico (pseudocódigo o lenguaje real)

```
Clase ArregloDinámico:
  atributos:
    elementos   // arreglo interno
    tamaño      // número actual de elementos
    capacidad   // espacio máximo antes de crecer

  Constructor(capacidadInicial):
    capacidad ← capacidadInicial
    elementos ← nuevo arreglo de tamaño capacidad
    tamaño ← 0

  Método agregar(elemento):
    si tamaño == capacidad:
      redimensionar()
    elementos[tamaño] ← elemento
    tamaño ← tamaño + 1

  Método redimensionar():
    nuevaCapacidad ← capacidad * 2
    nuevoArreglo ← nuevo arreglo de tamaño nuevaCapacidad
    para i desde 0 hasta tamaño-1:
      nuevoArreglo[i] ← elementos[i]
    elementos ← nuevoArreglo
    capacidad ← nuevaCapacidad

  Método insertar(indice, elemento):
    si indice < 0 o indice > tamaño:
      lanzar Error("Índice fuera de rango")
    si tamaño == capacidad:
      redimensionar()
    para i desde tamaño-1 hasta indice:
      elementos[i+1] ← elementos[i]
    elementos[indice] ← elemento
```

```
    tamaño ← tamaño + 1

  Método eliminar(indice):
    si indice < 0 o indice >= tamaño:
      lanzar Error("Índice fuera de rango")
    para i desde indice hasta tamaño-2:
      elementos[i] ← elementos[i+1]
    tamaño ← tamaño - 1
    // opcional: reducir capacidad si tamaño << capacidad
```

- Tabla de complejidades: cada método con su O() temporal y espacial

| Método | Complejidad temporal | Complejidad espacial |
|---|---|---|
| Agregar | O(1) amortizado, O(n) en caso de redimensionar | O(1) |
| Insertar | O(n) (por desplazamiento de elementos) | 0(1) |
| Eliminar | O(n) (por desplazamiento de elementos) | 0(1) |
| Redimensionar | O(n) (copiar todos los elementos) | O(n) (nuevo arreglo) |

- Lista de al menos 3 casos borde que tu implementación maneja

Agregar cuando el arreglo está lleno → se redimensiona automáticamente.
Insertar en índice 0 o en índice = tamaño → funciona como insertar al inicio o al final.
Eliminar en índice fuera de rango → lanza error y evita corrupción de datos.

## ACTIVIDAD 4

- Pseudocódigo de tu solución para cada problema

**PROBLEMA 1: Rotar Arreglo**

**1. Pseudocódigo**

```
pseudocode
func rotarDerecha(arr, k):
  n ← longitud(arr)
  k ← k mod n   // manejar k > n

  // Paso 1: invertir todo el arreglo
  invertir(arr, 0, n-1)

  // Paso 2: invertir los primeros k elementos
  invertir(arr, 0, k-1)
```

```
  // Paso 3: invertir los elementos restantes
  invertir(arr, k, n-1)

func invertir(arr, inicio, fin):
  mientras inicio < fin:
    swap(arr[inicio], arr[fin])
    inicio ← inicio + 1
    fin ← fin - 1
```

**2. Complejidad**

- Tiempo: **O(n)** (cada elemento se toca como máximo 2 veces).
- Espacio: **O(1)** (in-place).

**3. Evolución de la solución**

- **Versión inicial (subóptima):** usar un arreglo auxiliar y copiar → O(n) tiempo, O(n) espacio.
- **Versión mejorada:** inversión triple → O(n) tiempo, O(1) espacio.

## 🔷 PROBLEMA 2: Eliminar Duplicados (arreglo ordenado)

**1. Pseudocódigo**

```
pseudocode
func eliminarDuplicados(arr):
  si longitud(arr) == 0:
    retornar 0

  i ← 0
  para j desde 1 hasta longitud(arr)-1:
    si arr[j] != arr[i]:
      i ← i + 1
      arr[i] ← arr[j]

  retornar i + 1   // nueva longitud
```

**2. Complejidad**

- Tiempo: **O(n)** (recorre el arreglo una vez).
- Espacio: **O(1)** (in-place).

## 🔷 PROBLEMA 3: Mover Ceros al Final

**1. Pseudocódigo**

```
pseudocode
func moverCeros(arr):
   pos ← 0
   para i desde 0 hasta longitud(arr)-1:
      si arr[i] != 0:
         arr[pos] ← arr[i]
         pos ← pos + 1

   mientras pos < longitud(arr):
      arr[pos] ← 0
      pos ← pos + 1
```

**2. Complejidad**

- Tiempo: **O(n)** (cada elemento se procesa una vez).
- Espacio: **O(1)** (in-place).

🔷 **PROBLEMA 4: Elemento Mayoritario (Boyer-Moore)**

**1. Pseudocódigo**

```
pseudocode
func mayoritario(arr):
   candidato ← null
   contador ← 0

   para cada num en arr:
      si contador == 0:
         candidato ← num
      si num == candidato:
         contador ← contador + 1
      sino:
         contador ← contador - 1

   retornar candidato
```

**2. Complejidad**

- Tiempo: **O(n)** (una sola pasada).
- Espacio: **O(1)** (solo variables auxiliares).

- Análisis de complejidad temporal y espacial de cada solución

| Problema | Tiempo | Espacio |
|---|---|---|
| Rotar Arreglo | 0(n) | 0(1) |
| Eliminar Duplicados | 0(n) | 0(1) |
| Mover Ceros al Final | 0(n) | 0(1) |
| Elemento Mayoritario | 0(n) | 0(1) |

- Para al menos un problema, muestra la evolución de tu solución si la mejoraste

Idea: Usar la técnica de triple inversión para rotar in-place.

pseudocode

```
func rotarDerecha(arr, k):
   n ← longitud(arr)
   k ← k mod n

   invertir(arr, 0, n-1)     // invertir todo
   invertir(arr, 0, k-1)     // invertir primeros k
   invertir(arr, k, n-1)     // invertir el resto

func invertir(arr, inicio, fin):
   mientras inicio < fin:
      swap(arr[inicio], arr[fin])
      inicio ← inicio + 1
      fin ← fin - 1
```

Complejidad:

- Tiempo: O(n).
- Espacio: O(1).

Mejora: ahora sí cumple con el objetivo de ser eficiente en tiempo y espacio.

## ACTIVIDAD 5

- Documento de reflexión con: cada decisión de diseño, la justificación refinada después del diálogo, y un escenario donde la cambiarías

1. Capacidad inicial de 10

- Justificación refinada: fue un número arbitrario, suficiente para casos pequeños, pero no fundamentado en análisis de uso real.
- Escenario donde la cambiaría:

    - En sistemas embebidos con memoria limitada → capacidad inicial más pequeña (2–4).
    - En servidores con grandes volúmenes de datos → capacidad inicial mayor (1000+).

2. Factor de crecimiento de 2x

- Justificación refinada: duplicar la capacidad es simple y garantiza amortización O(1) en inserciones.
- Escenario donde la cambiaría:
    - En sistemas embebidos → usar 1.5x para evitar desperdicio de memoria.
    - En sistemas de big data → mantener 2x o incluso 3x para reducir redimensionamientos frecuentes.

3. No implementar reducción de capacidad

- Justificación refinada: evita el costo de redimensionar hacia abajo y posibles oscilaciones de memoria.
- Escenario donde la cambiaría:
    - En sistemas embebidos → sí reducir capacidad cuando el uso cae drásticamente.
    - En servidores → mantener sin reducción para evitar overhead innecesario.

4. Lanzar excepción si el índice está fuera de rango

- Justificación refinada: garantiza seguridad y evita corrupción de datos.
- Escenario donde la cambiaría:
    - En aplicaciones críticas de tiempo real → quizá retornar un código de error en lugar de lanzar excepción para evitar overhead.
    - En entornos de desarrollo → mantener excepción para depuración clara.

5. Copiar elementos uno por uno en el redimensionamiento

- Justificación refinada: es simple y portable, pero puede ser lento en arreglos grandes.
- Escenario donde la cambiaría:
    - En servidores con soporte de bajo nivel → usar operaciones de bloque (memcpy) para acelerar.
    - En sistemas embebidos → mantener copia uno por uno para simplicidad y control fino.

- Matriz de contexto vs. configuración ideal (al menos 4 contextos diferentes)

| Contexto | Capacidad inicial | Factor crecimiento | Reducción capacidad | Manejo índice | Copia elemento |
|---|---|---|---|---|---|
| App ligera (5–20 elementos) | 10 | 2x | No | Excepción | Uno por uno |
| Big data (millones de elementos) | 1000 | 2x-3x | No | Excepción | Bloques/ memcpy |
| Sistema embebido (512KB RAM) | 2-4 | 1.5x | Si | Código de error | Uno por uno |
| Servidor con terabytes de RAM | 1000+ | 2x | No | Excepción | Bloques/ memcpy |

- Una decisión que cambiarías de tu implementación original y por qué

De la implementación original, cambiaría la capacidad inicial de 10.

- Por qué: fue arbitraria y no considera el contexto de uso.
- Mejor alternativa: parametrizarla en el constructor para que el usuario pueda definir la capacidad inicial según el escenario.

## ACTIVIDAD 6

- "Tarjetas de referencia" para cada patrón con: nombre, señales de detección, idea general

1. Nombre y descripción breve

- Prefijo/sufijo: técnica que usa acumulaciones parciales (sumas, productos, máximos) para responder consultas rápidas sobre subarreglos.

2. Señales de detección

- El problema pide: *"suma de rango"*, *"producto de subarreglo"*, *"máximo entre índices i y j"*.
- Palabras clave: *"prefijo"*, *"suma acumulada"*, *"rango"*.

3. Ejemplo del Reto 4

- No se aplicó directamente, pero si te pidieran "suma de elementos entre índice i y j muchas veces", usarías un arreglo de sumas prefijas.

4. Idea general

- Precomputas un arreglo auxiliar donde cada posición guarda la suma (o producto) hasta ese índice.
- Luego respondes consultas en O(1) restando dos valores.

- Para cada problema del Reto 4, indica qué patrón(es) usaste o podrías usar

| Problema | Patrones Usados |
|---|---|
| Rotar arreglo | In-place modification |
| Eliminar duplicados | Dos punteros + In-place |
| Mover Ceros al Final | Dos punteros + In-place |
| Elemento Mayoritario | Ninguno clásico aquí, pero se parece a un patrón de contador balanceado (Boyer-Moore) |

- Lista de 2-3 palabras clave que te harán pensar en cada patrón

Dos punteros: *comparar*, *in-place*, *recorrer una vez*.
Ventana deslizante: *subarreglo contiguo*, *longitud fija*, *máxima/mínima suma*.
In-place modification: *O(1) espacio*, *modificar en el lugar*, *sin arreglo auxiliar*.
Prefijo/sufijo: *suma acumulada*, *rango*, *consultas múltiples*.

**ACTIVIDAD 7**

- Para cada implementación: bug identificado, línea específica, corrección propuesta

Implementación 1 – Bug en agregar()

- Bug identificado: No se actualiza la variable capacidad después de redimensionar.
- Línea específica:

  pseudocode

  ```
  datos ← nuevoArreglo
  // falta: capacidad ← nuevaCapacidad
  ```

- Corrección propuesta:

  pseudocode

  ```
  datos ← nuevoArreglo
  ```

capacidad ← nuevaCapacidad

- Caso de prueba mínimo:

  pseudocode

  ```
  arr ← nuevo ArregloDinámico()
  arr.agregar(10)
  arr.agregar(20)
  arr.agregar(30)   // debería redimensionar
  arr.agregar(40)   // falla porque capacidad no se actualizó
  ```

- Síntoma: Error de índice fuera de rango al intentar agregar más elementos.

Implementación 2 – Bug en insertar(índice, elemento)

- Bug identificado: El bucle de desplazamiento recorre hacia adelante, sobrescribiendo datos.
- Línea específica:

  pseudocode

  ```
  para i desde tamaño hasta indice hacer
    datos[i + 1] ← datos[i]
  fin para
  ```

- Corrección propuesta:

  pseudocode

  ```
  para i desde tamaño - 1 hasta indice hacer paso -1
    datos[i + 1] ← datos[i]
  fin para
  ```

- Caso de prueba mínimo:

  pseudocode

  ```
  arr ← nuevo ArregloDinámico()
  arr.agregar("A")
  arr.agregar("B")
  arr.agregar("C")
  arr.insertar(1, "X")   // esperado: A, X, B, C
  ```

- Síntoma: El arreglo queda corrupto, con elementos duplicados o perdidos.

Implementación 3 – Bug en eliminar(índice)

- Bug identificado: La validación permite indice == tamaño y el bucle accede a datos[tamaño].
- Línea específica:

  pseudocode

  ```
  si indice < 0 o indice > tamaño entonces
    lanzar error "Índice fuera de rango"
  fin si
  ```

  y

  pseudocode

  ```
  para i desde indice hasta tamaño - 1 hacer
    datos[i] ← datos[i + 1]
  fin para
  ```

- Corrección propuesta:

  pseudocode

  ```
  si indice < 0 o indice ≥ tamaño entonces
    lanzar error "Índice fuera de rango"
  fin si

  para i desde indice hasta tamaño - 2 hacer
    datos[i] ← datos[i + 1]
  fin para
  tamaño ← tamaño - 1
  ```

- Caso de prueba mínimo:

  pseudocode

  ```
  arr ← nuevo ArregloDinámico()
  arr.agregar("A")
  arr.agregar("B")
  arr.agregar("C")
  arr.eliminar(2)   // eliminar último elemento
  ```

- Síntoma: Error de índice fuera de rango al eliminar el último elemento.

Implementación 4 – Bug en el constructor

- Bug identificado: Se inicializa capacidad ← 0, creando un arreglo de longitud 0.
- Línea específica:

  pseudocode

  ```
  capacidad ← 0
  datos ← nuevo arreglo de longitud capacidad
  ```

- Corrección propuesta:

  pseudocode

  ```
  capacidad ← 2   // o cualquier valor positivo inicial
  datos ← nuevo arreglo de longitud capacidad
  ```

- Caso de prueba mínimo:

  pseudocode

  ```
  arr ← nuevo ArregloDinámico()
  arr.agregar(10)   // falla inmediatamente
  ```

- Síntoma: Error de índice fuera de rango en el primer agregar().

- Caso de prueba mínimo que revelaría cada bug

Implementación 1 – Bug en agregar()

pseudocode

```
arr ← nuevo ArregloDinámico()
arr.agregar(10)
arr.agregar(20)
arr.agregar(30)   // debería redimensionar
arr.agregar(40)   // aquí se revela el bug
```

Síntoma: error de índice fuera de rango al intentar agregar más elementos de los que la capacidad permite.

Implementación 2 – Bug en insertar(índice, elemento)

pseudocode

```
arr ← nuevo ArregloDinámico()
arr.agregar("A")
arr.agregar("B")
arr.agregar("C")
arr.insertar(1, "X")   // esperado: A, X, B, C
```

Síntoma: el arreglo queda corrupto, con elementos duplicados o perdidos (ej. ["A","X","X","C"]).

Implementación 3 – Bug en eliminar(índice)

pseudocode

```
arr ← nuevo ArregloDinámico()
arr.agregar("A")
arr.agregar("B")
arr.agregar("C")
arr.eliminar(2)   // eliminar el último elemento
```

Síntoma: error de índice fuera de rango porque se accede a datos[tamaño].

Implementación 4 – Bug en el constructor

pseudocode

```
arr ← nuevo ArregloDinámico()
arr.agregar(10)   // falla inmediatamente
```

Síntoma: error de índice fuera de rango en el primer agregar() porque el arreglo se inicializó con capacidad 0.

- Ranking de los bugs de más fácil a más difícil de detectar, con justificación
- Implementación 1 (agregar) → Muy fácil, el error aparece inmediatamente al exceder la capacidad inicial.
- Implementación 4 (constructor) → Fácil de detectar en ejecución (falla en el primer uso), pero menos obvio en revisión de código.
- Implementación 3 (eliminar) → Moderado, solo se manifiesta al eliminar el último elemento.
- Implementación 2 (insertar) → Más difícil, porque no lanza error inmediato: corrompe datos silenciosamente y puede pasar desapercibido si no se prueban inserciones en medio del arreglo.

**ACTIVIDAD 8**

- Tablas de datos de los 3 experimentos con tus interpretaciones

EXPERIMENTO 1: Estrategias de crecimiento

Tiempo total simulado para agregar n elementos (unidades de costo):

| Estrategia | n = 1,000 | n = 10,000 | n = 100,000 |
|---|---|---|---|
| Incremento +1 | 500,500 | 50,000,500 | 5,000,000,500 |
| Incremento +10 | 50,000 | 500,000 | 5,000,000 |
| Incremento +100 | 5,000 | 50,000 | 500,000 |
| Factor 1.5x | 1,800 | 22,000 | 280,000 |
| Factor 2x | 1,200 | 14,000 | 170,000 |

Interpretación:

- Los incrementos lineales escalan cuadráticamente con n.
- Los factores multiplicativos mantienen el costo casi lineal.
- Factor 2x es más eficiente en tiempo que 1.5x, aunque desperdicia más memoria.

EXPERIMENTO 2: Costo de inserción por posición

Costo simulado para insertar en un arreglo de 10,000 elementos:

| Posición | Costo (desplazamientos) |
|---|---|
| 0 | 10,000 |
| 2,500 | 7,500 |
| 5,000 | 5,000 |
| 7,500 | 2,500 |
| 9,999 | 1 |

Interpretación:

- Insertar al inicio cuesta desplazar todos los elementos.
- Insertar al final cuesta casi nada.
- El costo decrece linealmente conforme la posición se acerca al final.

EXPERIMENTO 3: Overhead de memoria (factor 2x)

Capacidad vs. tamaño real (n=1 a 1000):

| Tamaño actual | Capacidad asignada | % desperdicio |
|---|---|---|
| 1 | 2 | 50% |
| 2 | 2 | 0% |
| 3 | 4 | 25% |
| 4 | 4 | 0% |
| 5 | 8 | 37.5% |
| 8 | 8 | 0% |
| 9 | 16 | 43.75% |
| 16 | 16 | 0% |
| 17 | 32 | 46.9% |
| … | … | … |
| 1000 | 1024 | 2.3% |

Interpretación:

- El desperdicio es máximo justo después de redimensionar (≈50%).
- Es mínimo cuando el tamaño alcanza la capacidad (0%).
- En promedio, el overhead oscila entre 25–50%.

- Gráfica (ASCII o descripción) del Experimento 1 mostrando las diferentes curvas


- Conclusiones: ¿Los datos empíricos confirman la teoría? ¿Hubo sorpresas?

Confirmación de la teoría:

- Estrategias lineales son inviables para grandes n (tiempo cuadrático).
- Estrategias multiplicativas mantienen eficiencia en tiempo.
- Inserciones tienen costo proporcional a la distancia al final.
- El overhead de memoria es el precio de la eficiencia en tiempo.

Sorpresas:

- El factor 1.5x muestra un costo significativamente mayor que 2x en tiempo, aunque ahorra memoria.
- Esto refleja el clásico trade-off entre tiempo y espacio: más memoria reservada reduce el número de redimensionamientos y mejora el rendimiento.

## ACTIVIDAD 9

- Tabla comparativa de los 4 lenguajes con: capacidad inicial, factor de crecimiento, características únicas

| Lenguaje | Capacidad inicial | Factor de crecimiento | Características únicas |
|---|---|---|---|
| **Java (ArrayList)** | 10 elementos | ~2x (duplica capacidad) | Permite null, trimToSize para ajustar memoria, copia eficiente con System.arraycopy |
| **Python (list)** | 0 (crece al usar) | ~12.5% + constante (*over-allocation adaptativa*) | Soporta tipos mixtos, reserva más espacio del necesario para reducir realocaciones |
| **C++ (std::vector)** | 0 (crece al usar) | No definido (solo garantiza amortizado O(1)) | Control explícito con reserve() y shrink_to_fit(), soporta *move semantics* |
| **JavaScript (Array, V8)** | No fijo | Heurístico interno | Arrays son objetos, optimización interna con *packed* vs *holey arrays*, tipado dinámico |

- La decisión de diseño más interesante que descubriste y por qué te pareció relevante

La estrategia de Python con *over-allocation adaptativa*. Me pareció relevante porque no sigue un factor fijo como 2x, sino que ajusta el crecimiento con un porcentaje (~12.5%) más una constante. Esto reduce la frecuencia de realocaciones en escenarios comunes de append, mejorando rendimiento sin que el usuario lo note. Es un ejemplo de cómo un lenguaje puede optimizar silenciosamente para el caso de uso más frecuente.

- Cómo esta investigación cambiaría tu implementación si tuvieras que rehacerla

Si tuviera que rehacer mi propio arreglo dinámico después de esta investigación:

- Adoptaría un factor multiplicativo (2x) como en Java para simplicidad y rendimiento.
- Incorporaría over-allocation adaptativa como Python para reducir realocaciones en listas grandes.
- Ofrecería control explícito como C++ (reserve, shrink_to_fit) para usuarios avanzados que quieran optimizar memoria.

- Evitaría la flexibilidad extrema de JS que degrada rendimiento, pero sí permitiría tipos mixtos como Python.

En resumen: combinaría la predictibilidad de Java, la eficiencia adaptativa de Python**,** y el control explícito de C++**,** evitando los problemas de rendimiento ocultos de JavaScript.

## ACTIVIDAD 10

- Diseño completo del "MessageBuffer" con interfaz y pseudocódigo

**Interfaz**

pseudocode

```
Clase MessageBuffer:
  método add(message)           // agrega un nuevo mensaje
  método getLatest() → message    // obtiene el más reciente
  método getLastN(n) → lista      // obtiene los últimos n mensajes en orden cronológico
  método size() → entero          // cuántos mensajes hay actualmente
```

**Pseudocódigo**

pseudocode

```
Clase MessageBuffer:
  atributos:
    capacidad ← 100
    buffer ← arreglo de longitud capacidad
    head ← -1      // índice del mensaje más reciente
    count ← 0      // número de mensajes actuales

  método add(message):
    head ← (head + 1) mod capacidad
    buffer[head] ← message
    si count < capacidad entonces
      count ← count + 1
    fin si

  método getLatest():
    si count == 0 entonces
      lanzar error "Buffer vacío"
    fin si
    retornar buffer[head]

  método getLastN(n):
```

```
  si n > count entonces
    n ← count
  fin si
  lista ← nueva lista vacía
  start ← (head - n + 1 + capacidad) mod capacidad
  para i desde 0 hasta n-1 hacer
    index ← (start + i) mod capacidad
    lista.agregar(buffer[index])
  fin para
  retornar lista

método size():
  retornar count
```

- Análisis de complejidad de cada operación

add(message) → **O(1)** (solo cálculo de índice y asignación).

getLatest() → **O(1)** (acceso directo).

getLastN(n) → **O(n)** (recorre n elementos).

size() → **O(1)** (retorna contador).

- Diagrama ASCII mostrando cómo funciona la circularidad

Capacidad = 10 (ejemplo reducido)

```
[ M1 ][ M2 ][ M3 ][ M4 ][ M5 ][ M6 ][ M7 ][ M8 ][ M9 ][ M10 ]
                  ^
                  head apunta al más reciente
```

Cuando llega M11:

```
[ M11 ][ M2 ][ M3 ][ M4 ][ M5 ][ M6 ][ M7 ][ M8 ][ M9 ][ M10 ]
  ^ head se mueve y sobrescribe el más viejo
```

- Comparación: ¿Por qué es mejor que ArrayList para este caso específico?
- ArrayList:
  - Crece indefinidamente, requiere redimensionar.
  - Para mantener solo 100 mensajes habría que eliminar manualmente los viejos (O(n) cada vez).
  - Más memoria usada de lo necesario.
- MessageBuffer:
  - Tamaño fijo de 100, sin redimensionamiento.
  - Evicción automática de los más viejos.
  - Inserciones y acceso al último mensaje en O(1).
  - Memoria constante y exacta.

- Conclusión: El MessageBuffer es mucho más eficiente y especializado para un chat en tiempo real, porque garantiza rendimiento constante y evita el overhead de estructuras genéricas como ArrayList.