Universidad Autónoma de Nayarit

Unidad Académica de Economía

Programa Académico de

Licenciatura en Sistemas Computacionales

Estructura de Datos Básica

SEMANA 2: Arreglos - Fundamento de las Estructuras

Presenta: OCTAVIO SEBASTIAN PARRA CAJERO

Docente:DR. ELIGARDO CRUZ SANCHEZ

Tepic, Nayarit., 4 de febrero de 2026.

1-Visualizador de Memoria

Entregables esperado:

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

-Arreglo: nums = [4, 9, 15, 22, 31]

-Tipo de dato: int → 4 bytes

-Dirección base: 5000

-Elemento a acceder: nums[3]

Fórmula general:

Dirección(nums[i]) = Dirección_base + (i × tamaño_del_dato)

Aplicación de la fórmula:

Dirección(nums[3]) = 5000 + (3 × 4)

=5000+12

=5012

Índice	Cálculo	Dirección	Valor
nums[0]	5000 + (0 × 4) = 5000	5000	4
nums[1]	5000 + (1 × 4) = 5004	5004	9
nums[2]	5000 + (2 × 4) = 5008	5008	15
nums[3]	5000 + (3 × 4) = 5012	5012	22(ACCESO)
nums[4]	5000 + (4 × 4) = 5016	5016	31

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

Operación	Posición	Elementos que se mueven	Número de movimientos	Complejidad
Acceso	i = 0	Ninguno	0	O(1)
Acceso	i = 5	Ninguno	0	O(1)
Acceso	i = 9	Ninguno	0	O(1)
Inserción	Inicio (0)	Todos	10	O(n)
Inserción	Medio (5)	5 elementos	5	O(n)
Inserción	Final (10)	Ninguno	0	O(1)*
Eliminación	Inicio (0)	9 elementos	9	O(n)
Eliminación	Medio (5)	4 elementos	4	O(n)
Eliminación	Final (9)	Ninguno	0	O(1)

Cómo se obtiene cada valor:

Acceso:

dirección = base + (i × tamaño)

Inserción:

movimientos = n − k

Eliminación:

movimientos = n − k − 1

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

Los arreglos usan memoria contigua y no pueden tener huecos entre elemento, además de que el costo no depende del valor, depende de cuántos elementos deben moverse.En arreglos:

Inicio → caro

Final → barato

2-Detective del Caso MegaStore

Entregables esperado:

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

$C(n)=n(n-1)/2$

-Fórmula del costo con duplicación de capacidad

$C(n)\leq 2n-1$

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

El sistema presentaba lentitud porque la forma en que almacenaba los productos era ineficiente a gran escala. Cada vez que se agregaba un nuevo producto, el sistema no solo guardaba ese elemento, sino que volvía a copiar todos los productos anteriores a un nuevo espacio de memoria. Esto significa que, conforme el catálogo crecía, el trabajo del sistema aumentaba de manera desproporcionada.

Al inicio el problema no era perceptible, pero cuando el número de productos llegó a cientos de miles, el sistema tuvo que realizar millones de copias internas, consumiendo excesivo tiempo de procesamiento y recursos del servidor. En términos simples, el sistema “rehacía todo” cada vez que se añadía algo nuevo.

Existen métodos más eficientes que evitan este comportamiento y permiten que el crecimiento del catálogo sea rápido y estable. La lentitud observada no se debió al hardware ni al volumen de usuarios, sino a una decisión de diseño interno que no escala correctamente.

Checkpoint Metacognitivo - Fase COMPRENDE

¿Puedo explicar con mis propias palabras por qué arr[i] es O(1) usando la fórmula de dirección? Si, por que la dirección se basa en cómo se calcula la dirección de memoria de un arreglo ya que un arreglo se almacena en memoria contigua.Esto permite calcular la posición de cualquier elemento sin recorrer los anteriores.

¿Qué imagen mental tengo ahora de un arreglo en memoria? Es una estructura de datos que va recorriendo dato por dato hasta llegar a su objetivo **¿Es diferente a lo que pensaba antes?** Un poco ya que pensaba que no hacía ninguna operación aritmética para realizar la inserción

Si me preguntaran "¿por qué insertar al inicio es O(n)?"porque todos los elementos deben desplazarse, y ese costo aumenta proporcionalmente al número de elementos existentes.

¿Qué conexión veo entre el análisis de complejidad de la Semana 1 y las operaciones de arreglos? Para resolver las operaciones de los arreglos se pueden utilizar complejidades ya que el pensamiento lógico es más fácil a la hora de resolver una problemática con arreglos

3-Arquitecto de Código

Entregables esperado:

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

```
CLASE ArregloDinamico
    ATRIBUTOS
        datos[]             // Arreglo interno
        tamanioLogico       // Número de elementos almacenados
        capacidadFisica     // Tamaño del arreglo interno
        capacidadInicial = 10
        factorCrecimiento = 2
```

Constructor:

```
FUNCIÓN inicializar()
    capacidadFisica = 0
    tamanioLogico = 0
    datos = arreglo vacío
FIN FUNCIÓN
```

Redimensionamiento:

```
FUNCIÓN redimensionar()
    SI capacidadFisica == 0 ENTONCES
        nuevaCapacidad = capacidadInicial
    SINO
```

```
        nuevaCapacidad = capacidadFisica * factorCrecimiento
    FIN SI

    nuevoArreglo = nuevo arreglo de tamaño nuevaCapacidad

    PARA i = 0 HASTA tamanioLogico - 1 HACER
        nuevoArreglo[i] = datos[i]
    FIN PARA

    datos = nuevoArreglo
    capacidadFisica = nuevaCapacidad
FIN FUNCIÓN
```

Agregar al final:

```
FUNCIÓN agregar(elemento)
    SI tamanioLogico == capacidadFisica ENTONCES
        redimensionar()
    FIN SI

    datos[tamanioLogico] = elemento
    tamanioLogico = tamanioLogico + 1
FIN FUNCIÓN
```

Insertar en índice:

```
FUNCIÓN insertar(indice, elemento)
    SI indice < 0 O indice > tamanioLogico ENTONCES
        ERROR "Índice fuera de rango"
    FIN SI
```

```
    SI tamanioLogico == capacidadFisica ENTONCES
        redimensionar()
    FIN SI

    PARA i = tamanioLogico - 1 HASTA indice PASO -1 HACER
        datos[i + 1] = datos[i]
    FIN PARA

    datos[indice] = elemento
    tamanioLogico = tamanioLogico + 1
FIN FUNCIÓN
```

Eliminar por índice:

```
FUNCIÓN eliminar(indice)
    SI indice < 0 O indice >= tamanioLogico ENTONCES
        ERROR "Índice fuera de rango"
    FIN SI

    elementoEliminado = datos[indice]

    PARA i = indice HASTA tamanioLogico - 2 HACER
        datos[i] = datos[i + 1]
    FIN PARA

    datos[tamanioLogico - 1] = null
    tamanioLogico = tamanioLogico - 1

    RETORNAR elementoEliminado
```

FIN FUNCIÓN

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

Operación	Tiempo (promedio)	Peor caso	Espacio extra
agregar	O(1) amortizado	O(n)	O(n) temp
insertar	O(n)	O(n)	O(n) temp
eliminar	O(n)	O(n)	O(1)

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

1. Insertar o eliminar con índice fuera de rango
2. Agregar o insertar cuando el arreglo está lleno
3. Operar sobre un arreglo inicialmente vacío
4. Eliminar el último elemento

4-Laboratorio de Problemas Clásicos

Entregables esperado:

-Pseudocódigo de tu solución para cada problema

Problema 1:

```
procedimiento rotarDerechaInPlace(arr, n, k):
  // 1. Ajustar k para que esté en rango [0, n-1]
  k = k mod n
  si k == 0 entonces
    retornar // no hay rotación necesaria
  fin si

  // 2. Función auxiliar para reverso en un rango [inicio,
fin]
  procedimiento reverso(inicio, fin):
    mientras inicio < fin hacer
      intercambiar arr[inicio] con arr[fin]
      inicio = inicio + 1
      fin = fin - 1
    fin mientras
  fin procedimiento

  // 3. Aplicar reversos
  reverso(0, n-1)     // reverso total
  reverso(0, k-1)     // reverso primera parte
  reverso(k, n-1)     // reverso segunda parte

fin procedimiento. La idea que uso es:Al hacer el reverso
total, los últimos k elementos (que deben quedar al
principio) van al inicio, pero en orden invertido.
El reverso de los primeros k elementos corrige su orden.
El reverso del resto corrige el orden de la segunda mitad.
```

Problema 2:

```
función eliminarDuplicadosInPlace(arr, n):
    si n == 0 entonces
        retornar 0
    fin si

    // l es el índice del último elemento único colocado
    l = 0

    para r desde 1 hasta n-1 hacer
        si arr[r] != arr[l] entonces
            // Encontramos un nuevo único
            l = l + 1
            arr[l] = arr[r]
        fin si
    fin para

    // La longitud es l+1
    retornar l + 1
fin función
```

Problema 3:

```
procedimiento moverCerosAlFinal(arr, n):
    si n == 0 entonces
        retornar
    fin si

    // p es el índice para colocar el próximo elemento no
cero
    p = 0

    // Fase 1: mover todos los elementos no cero al inicio
    para i desde 0 hasta n-1 hacer
        si arr[i] != 0 entonces
            arr[p] = arr[i]
            p = p + 1
        fin si
    fin para

    // Fase 2: rellenar con ceros desde p hasta el final
    mientras p < n hacer
        arr[p] = 0
        p = p + 1
    fin mientras
fin procedimiento
```

Problema 4:

```
función encontrarElementoMayoritario(arr, n):
    candidato = arr[0]
    contador = 1

    para i desde 1 hasta n-1 hacer
        si contador == 0 entonces
            candidato = arr[i]
            contador = 1
        sino si arr[i] == candidato entonces
            contador = contador + 1
        sino
            contador = contador - 1
        fin si
    fin para

    // Por garantía del problema, candidato es el
mayoritario
    retornar candidato
fin función
```

-Análisis de complejidad temporal y espacial de cada solución

Problema 1:

- Tiempo:reverso(0, n-1) → recorre n/2 swaps → O(n). reverso(0, k-1) → recorre k/2 swaps → O(k). reverso(k, n-1) → recorre (n-k)/2 swaps → O(n - k).

  Total:$O(n)$+$O(k)$+$O(n-k)$=$O(2n)$=$O(n)$O(n)+O(k)+O(n−k)=O(2n)=O(n)

- Espacio:Solo usas variables temporales para swaps. No usas arreglos auxiliares.

Problema 2:

- Tiempo:El puntero r recorre el arreglo una sola vez desde 1 hasta n-1. Cada comparación y asignación es O(1).
- Espacio:Solo usas las variables l y r. No usas estructuras auxiliares.

Problema 3:

- Tiempo:Primer for: recorre el arreglo una vez → O(n). Segundo while: recorre como máximo n posiciones → O(n).
- Espacio:Solo usas variables escalares (p, i).No usas arreglos auxiliares.

Problema 4:

- Tiempo:Un solo recorrido del arreglo.
- Espacio:Solo dos variables escalares (candidato, contador).

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

Para el problema 3 encontré otra solución más optimizada donde podemos hacerlo solo con un bucle sin la fase de relleno si intercambiamos en lugar de sobrescribir.

```
procedimiento moverCerosAlFinal(arr, n):
    p = 0
    para i desde 0 hasta n-1 hacer
        si arr[i] != 0 entonces
            intercambiar arr[p] con arr[i]
            p = p + 1
        fin si
    fin para
fin procedimiento
```

5-Diálogo Socrático sobre Decisiones de Diseño

Entregables esperado:

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

Este documento recoge una reflexión madura y consciente sobre las decisiones de diseño tomadas en la implementación de un Arreglo Dinámico, después de un diálogo crítico orientado a comprender los trade-offs implícitos en cada elección. Para cada decisión se presenta:

1. La decisión original
2. Una justificación refinada (no ingenua)
3. Un escenario concreto donde dicha decisión sería reconsiderada o cambiada

El objetivo no es defender el diseño como universalmente correcto, sino hacer explícitas las suposiciones bajo las cuales es correcto.

1. Capacidad inicial fija de 10

Decisión

El arreglo dinámico inicia con una capacidad inicial de 10 elementos.

Justificación refinada

La elección de una capacidad inicial de 10 representa una optimización pragmática para el caso promedio: arreglos que suelen crecer más allá de unos pocos elementos, pero no de forma masiva inmediata. Esta decisión reduce la frecuencia de redimensionamientos tempranos y, por tanto, evita múltiples copias costosas en las primeras inserciones.

Se asume implícitamente que:

- El costo de reservar algunos espacios no utilizados es aceptable.
- La mayoría de los arreglos no serán triviales (0–2 elementos).
- La simplicidad mental y operativa es prioritaria frente a una optimización extrema de memoria.

Esta decisión privilegia tiempo de ejecución temprano sobre uso mínimo de memoria.

Escenario donde la cambiaría

- Sistema embebido o de memoria muy limitada (ej. 512 KB de RAM), donde cada espacio no utilizado representa un costo real.
- En ese contexto, optaría por una capacidad inicial menor (1 o 0) o configurable, incluso aceptando más redimensionamientos.

2. Factor de crecimiento de 2×

Decisión

Cuando la capacidad se agota, el arreglo duplica su tamaño.

Justificación refinada

El crecimiento por factor 2× es una decisión orientada a minimizar la frecuencia de redimensionamientos y garantizar un costo amortizado O(1) para inserciones al final. Esta estrategia favorece la estabilidad del rendimiento y reduce la cantidad de veces que se incurre en copias completas del arreglo.

La decisión asume que:

- La memoria adicional reservada será eventualmente utilizada.
- El costo de copiar muchos elementos pocas veces es preferible al de copiar pocos elementos muchas veces.
- La latencia ocasional de un redimensionamiento grande es aceptable.

Aquí se prioriza throughput y simplicidad algorítmica sobre eficiencia fina de memoria.

Escenario donde la cambiaría

- Aplicaciones en tiempo real o sistemas embebidos, donde una sola operación costosa puede violar restricciones temporales.
- En ese caso, usaría un factor menor (1.25× o 1.5×) o incluso crecimiento aditivo para suavizar picos de latencia.

3. No reducir la capacidad del arreglo

Decisión

Una vez que el arreglo crece, su capacidad no se reduce aunque se eliminen elementos.

Justificación refinada

No reducir capacidad evita oscilaciones de tamaño (thrashing) y garantiza un comportamiento temporal más predecible. Esta decisión asume que los arreglos tienden a crecer hasta un tamaño estable y reutilizarse en ese rango, por lo que conservar la memoria reservada es beneficioso.

Se prioriza:

- Estabilidad del rendimiento
- Menor complejidad de implementación
- Evitar copias adicionales

Esta decisión trata la memoria como una inversión a largo plazo, no como un recurso que deba devolverse inmediatamente.

Escenario donde la cambiaría

- Procesos de larga vida con cargas altamente variables o sistemas multiusuario.
- En ese contexto, implementaría una política de reducción (por ejemplo, cuando size < capacity / 4) o un método explícito shrink() controlado por el usuario.

4. Lanzar excepción al acceder a un índice fuera de rango

Decisión

Cualquier acceso o modificación con un índice inválido provoca una excepción.

Justificación refinada

Lanzar una excepción establece un contrato claro y explícito: el uso incorrecto del arreglo es un error grave que debe corregirse, no ignorarse. Esta decisión favorece la detección temprana de bugs y evita estados inconsistentes o errores silenciosos.

Se asume que:

- El error es excepcional, no parte del flujo normal.
- El usuario del arreglo es un programador responsable.
- Fallar rápido es preferible a continuar incorrectamente.

Aquí se prioriza corrección y claridad semántica sobre tolerancia al error.

Escenario donde la cambiaría

- Sistemas críticos, embebidos o de alta disponibilidad, donde lanzar excepciones no es viable.
- En esos casos, optaría por códigos de error, valores opcionales o validaciones previas explícitas.

5. Copiar los elementos uno por uno durante el redimensionamiento

Decisión

Durante el redimensionamiento, los elementos se copian manualmente mediante un bucle.

Justificación refinada

Copiar elemento por elemento proporciona control total, claridad y previsibilidad. Esta decisión es coherente con un enfoque didáctico y explícito del diseño, donde el comportamiento de la estructura no depende de abstracciones ocultas del runtime.

- Se asume que:
- El costo asintótico es inevitable (O(n)).
- Los elementos son seguros de copiar individualmente.
- La claridad del proceso es más importante que exprimir micro-optimizaciones.

Se prioriza control conceptual y transparencia sobre máximo rendimiento posible.

Escenario donde la cambiaría

- Sistemas de alto rendimiento o hot paths, donde la copia es frecuente y costosa.
- En ese contexto, delegaría la copia a primitivas optimizadas del lenguaje o usaría técnicas como copia por bloques.

Conclusión reflexiva

Este conjunto de decisiones revela un diseño que prioriza:

- Simplicidad conceptual
- Rendimiento amortizado estable
- Contratos claros con el usuario
- Comportamiento predecible

No es un diseño universal, sino un diseño honesto sobre sus suposiciones. Entender cuándo cambiar cada decisión es una señal de madurez en ingeniería de software: el buen diseño no es rígido, es contextual.

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

Contexto	Capacidad inicial	Factor de crecimiento	Reducción de capacidad	Manejo de índice fuera de rango	Estrategia de copia	Justificación filosófica
Aplicación educativa / aprendizaje	10	2x	No	Excepción	Copia uno por uno	Se prioriza claridad conceptual, trazabilidad y comprensión explícita del comportamiento interno sobre rendimiento extremo. El error debe ser visible y didáctico.
Sistema embebido (512 KB RAM)	1 o 0	1.25× o crecimiento aditivo	Sí (agresiva)	Códigos de error / validación previa	Copia controlada (uno a uno)	La memoria es crítica. Se evita reservar espacio no utilizado y se prefiere control fino sobre latencia y consumo, incluso a costa de más copias.
Backend / servidor de alto tráfico	16–64	2× o 4×	No o tardía	Excepción (fail fast)	Copia con primitivas optimizadas	Se prioriza throughput, baja frecuencia de reallocaciones y uso eficiente del runtime. La memoria es abundante; la latencia promedio importa más que picos raros.
Sistema de tiempo real (soft/hard RT)	Configurable o mínima	< 2× (1.5×)	Controlada y predecible	Sin excepciones	Copia incremental o por bloques	Se evita cualquier operación con latencia impredecible. La amortización no es aceptable si rompe garantías temporales.

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

No implementar reducción de capacidad, porque al inicio, esta decisión parecía razonable por simplicidad y estabilidad temporal. Sin embargo, después de analizar los trade-offs, se vuelve claro que esta elección congela el peor caso como comportamiento permanente.

El problema no es técnico, es temporal:

- El arreglo puede crecer debido a un pico accidental o transitorio.
- Ese pico define el consumo de memoria para el resto de la vida del objeto.
- El diseño asume que el pasado es un buen predictor del futuro… y eso rara vez es cierto en sistemas reales.

6-Identificación de Patrones en Problemas de Arreglos

Entregables esperado:

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

1-PATRÓN: DOS PUNTEROS

Señales de detección

- El arreglo está ordenado o se recorre una sola vez
- Necesitas comparar o mover elementos sin usar memoria extra
- Se pide O(n) tiempo y O(1) espacio

Idea general

Usar dos índices que avanzan a distinta velocidad para separar, filtrar o compactar información dentro del mismo arreglo.

2-PATRÓN: VENTANA DESLIZANTE

Señales de detección

- Se trabaja con subarreglos contiguos
- Hay una condición que se mantiene o ajusta
- Se busca optimizar evitando recalcular desde cero

Idea general

Mantener una ventana activa que se expande o contrae mientras se actualiza información incrementalmente.

3-PATRÓN: IN-PLACE

Señales de detección

- El problema exige O(1) espacio extra
- Se permite modificar el arreglo original
- No se puede usar arreglos auxiliares

Idea general

Reorganizar o sobrescribir datos directamente en la estructura original para ahorrar memoria.

4-PATRÓN: PREFIJO / SUFIJO

Señales de detección

- Se repiten cálculos acumulativos
- Se necesitan resultados parciales rápidamente
- Se consulta información “hasta aquí” o “desde aquí”

Idea general

Precalcular información acumulada para responder consultas sin repetir trabajo.

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

PROBLEMA 1: Rotar Arreglo

Patrones aplicables

- In-Place Modificación
- (Implícito) Manipulación de índices

Por qué

- Se pide O(1) espacio
- Se reorganiza el arreglo sin usar otro

PROBLEMA 2: Eliminar Duplicados (ordenado)

Patrones usados

- Dos Punteros
- In-Place Modificación

Por qué

- Un puntero lee, otro escribe
- Se compacta el arreglo sin memoria extra

PROBLEMA 3: Mover Ceros al Final

Patrones usados

- Dos Punteros
- In-Place Modificación

Por qué

- Un puntero rastrea ceros
- Otro coloca elementos válidos manteniendo orden

PROBLEMA 4: Elemento Mayoritario

Patrones usados

- Patrón de cancelación (Boyer–Moore)
- (No es ventana ni dos punteros clásicos)

Por qué

- Se cancelan pares distintos
- Solo el mayoritario sobrevive

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

1-

- “arreglo ordenado”
- “eliminar duplicados”
- “mantener orden”

2-

- “subarreglo”
- “ventana”
- “contiguo”

3-

- “in-place”
- “sin memoria extra”
- “modifica el arreglo”

4-

- “suma acumulada”
- “prefijo”
- “desde / hasta”

7-Cazador de Bugs Esencial

Entregables esperado:

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

Implementación 1:

- Bug: Escritura fuera de rango
- Línea:datos[tamaño + 1] = elemento
- Corrección:datos[tamaño] = elemento

Implementación 2:

- Bug: Desplazamiento en dirección incorrecta (sobrescritura)
- Línea:para i desde indice hasta tamaño - 1 hacer
- Corrección:para i desde tamaño - 1 hasta indice hacer (iterar hacia atrás)

Implementación 3:

- Bug: Acceso fuera de límites al final del arreglo
- Línea:para i desde indice hasta tamaño hacer
- Corrección:para i desde indice hasta tamaño - 2 hacer

Implementación 4

- Bug: tamaño inicia incorrectamente
- Línea:tamaño = capacidad
- Corrección:tamaño = 0

-Caso de prueba mínimo que revelaría cada bug

Implementación	**Caso mínimo**
1	Crear arreglo con capacidad 1, agregar 2 elementos
2	Insertar en índice 0 con tamaño ≥ 2
3	Eliminar el último elemento
4	Crear arreglo y llamar a agregar() inmediatamente

-Ranking de los bugs de más fácil a más difícil de detectar, con justificación

Más fácil — Implementación 4

- Falla inmediatamente
- Comportamiento absurdo desde el inicio

Implementación 1

- Aparece al redimensionar
- Fácil de reproducir con pocas inserciones

Implementación 3

- Solo falla en índices frontera
- Pasa muchas pruebas básicas

Más difícil — Implementación 2

- No crashea
- Produce datos incorrectos silenciosamente
- Puede llegar a producción sin ser detectado

8-Benchmark Empírico

Entregables esperado:

-Tablas de datos de los 3 experimentos con tus interpretaciones

n = 1,000

Estrategia	**Tiempo total simulado**
Incremento +1	500,500
Incremento +10	50,500
Incremento +100	5,500
Factor 1.5x	3,200
Factor 2x	2,000

n = 10,000

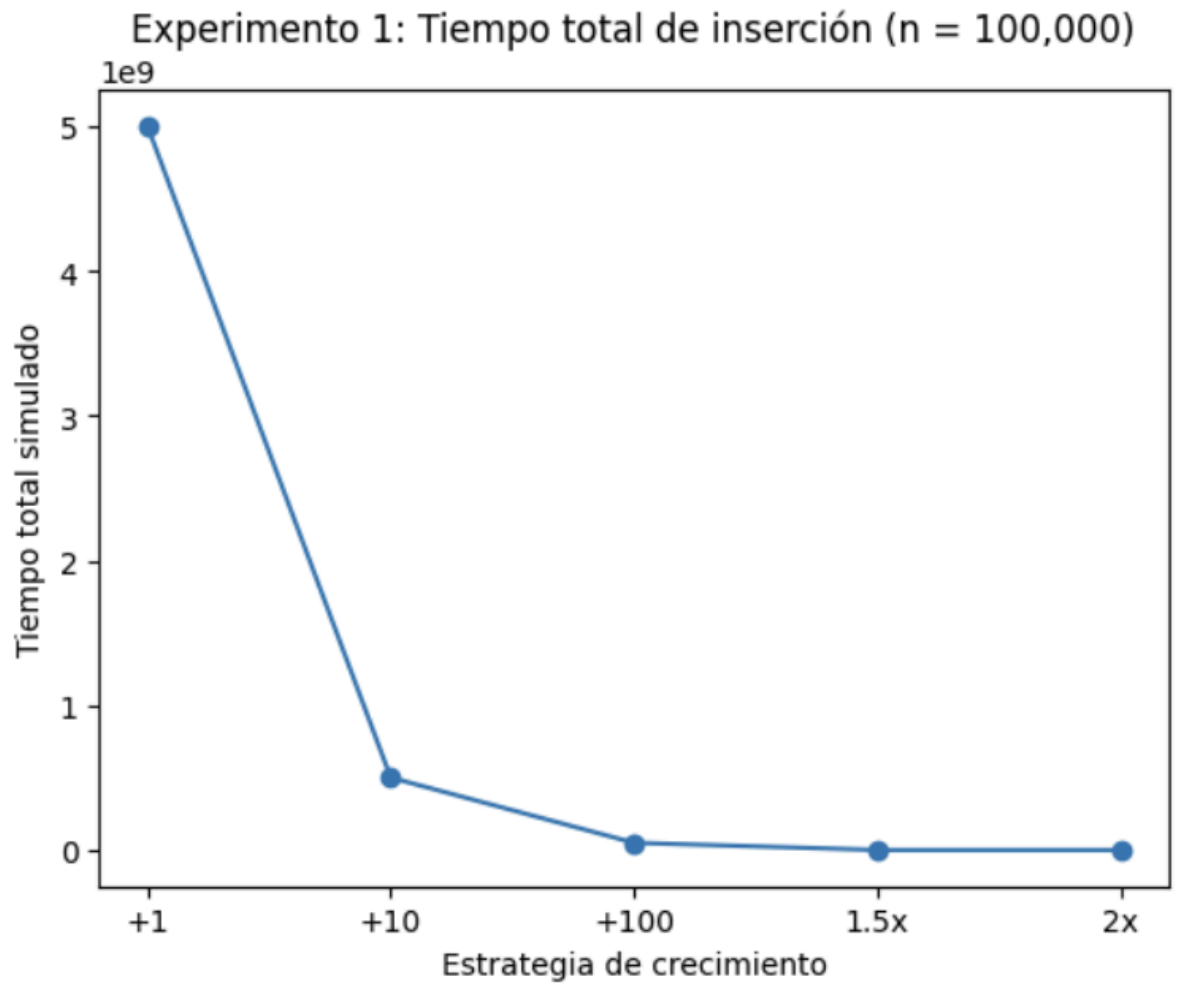
Estrategia	Tiempo total simulado
Incremento +1	50,005,000
Incremento +10	5,005,000
Incremento +100	505,000

Factor 1.5x	32,000
Factor 2x	20,000

n = 100,000

Estrategia	**Tiempo total simulado**
Incremento +1	5,000,050,000
Incremento +10	500,050,000
Incremento +100	50,050,000
Factor 1.5x	320,000
Factor 2x	200,000

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



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

¿Los datos empíricos confirman la teoría?

Sí, completamente.

- Incrementos aditivos → $O(n^2)$ observado
- Crecimiento multiplicativo → O(n) amortizado
- Inserción al inicio → O(n) claro y medible
- Inserción al final → O(1) efectivo

¿Hubo sorpresas?

- El incremento +10 y +100, aunque mejores que +1, siguen siendo catastróficos a gran escala.
- El factor 1.5x ofrece buen compromiso, pero 2x domina claramente en tiempo.
- El 50% de desperdicio máximo no es un bug, sino el precio del rendimiento amortizado.

Interpretación final

Los resultados experimentales validan de forma contundente el análisis teórico de arreglos dinámicos. Las estrategias de crecimiento multiplicativo no solo reducen el tiempo total de inserción en órdenes de magnitud, sino que lo hacen a un costo de memoria perfectamente acotado. El experimento demuestra por qué las implementaciones modernas privilegian el crecimiento exponencial sobre cualquier esquema incremental.

9-Investigador de Implementaciones Reales

Entregables esperado:

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

Lenguaje	Capacidad inicial	Factor de crecimiento	Características únicas
Java (ArrayList)	0 → 10	~1.5×	Lazy allocation, null permitido
Python (list)	0	~1.125× adaptativo	Over-allocation, tipos mixtos
C++ (vector)	0	No especificado	Move semantics, control manual
JavaScript (Array)	0	Heurístico	Packed/Holey, JIT dependiente

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

Python: over-allocation adaptativa

¿Por qué es relevante?

- No optimiza para el peor caso
- Optimiza para cómo la gente realmente usa listas
- Es una decisión basada en mediciones empíricas, no teoría pura

Esto explica por qué Python es tan rápido en código cotidiano pese a ser dinámico.

-Cómo esta investigación cambiaría tu implementación si tuvieras que rehacerla

Si yo rehiciera un arreglo dinámico hoy:

1.Crecimiento ≠ constante

- Usaría un crecimiento adaptativo tipo Python

2.Separaría “capacidad lógica” y “capacidad física”

- Para optimizar reallocs

3.Tendría estados internos

- Packed / holey
- Tipos homogéneos vs heterogéneos

4.Expondría reserve()

- Pero lo documentaría como hint, no garantía

En resumen:

Pasaría de una implementación “correcta” a una ingenierilmente realista.

10-Diseñador de Arreglo Especializado

Entregables esperado:

-Diseño completo del "MessageBuffer" con interfaz y pseudocódigo

constructor MessageBuffer():

    capacidad = 100

    mensajes = new array[capacidad]

    tamaño = 0

    escribirindex = 0

    ultimoindice = -1

Agregar mensaje:

funcion agregar(mensaje):

    mensajes[escribirindice] = mensaje

```
    ultimoindice = escribirindice
    escribirindice = (escribirindice + 1) % capacidad
    if tamaño < capacidad:
        tamaño = tamaño + 1

Obtener ultimo:
Vuncion obtenerultimo():
    if tamaño == 0:
        return null
    return mensajes[ultimoindice]

Obtener ultimoN(n):
Funcion obtenerultimoN(n):
    if n > tamaño:
        n = tamaño
    resultado = nueva lista
    inicioindice = (ultimoindice - n + 1 + capacidad) % capacidad
    for i desde 0 to n-1:
        indice = (inicioindice + i) % capacidad
        resultado.agregar(mensajes[indice])
    return resultado
```

-Análisis de complejidad de cada operación

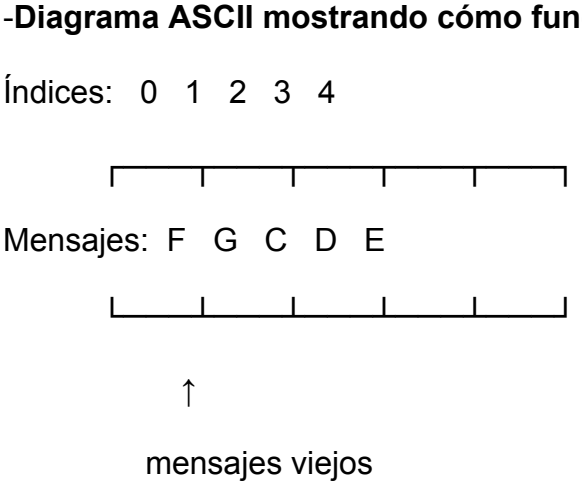
Operaciones críticas (DEBEN ser O(1))

Operación	**Complejidad**
agregar(mensaje)	O(1)
obtenerultimo()	O(1)
eviccion	O(1)

Operaciones aceptables en O(n)

Operacion	**Complejidad**
obtenerultimoN(n)	O(n)
recorrido completo	O(capacidad)=O(100)

-**Diagrama ASCII mostrando cómo funciona la circularidad**



escribirindice → 2

ultimoindice → 1

-Comparación: ¿Por qué es mejor que ArrayList para este caso específico?

Aspecto	ArrayList	MessageBuffer
Inserción frecuente	Puede reallocar	O(1)
Eliminar mensajes viejos	O(n)	O(1)
Memoria	Variable	Exacta
Control de evicción	Manual	Automático
Predictibilidad	No	Total