1. Visualizador de memoria

arreglo = [1,2,3,4,5,6]
tamaño de cada elemento = 4
Dirección base de memoria = 4500

Fórmula de acceso

P. ej. arreglo[3]
$4500 + (3 * 4) = 4512$

Dirección	Valor
4500	1
4504	2
4508	3
4512	4
4516	5
4520	6

Tabla comparativa de movimientos

n	Acceso	Inserción	Eliminación
0	0	10	9
1	0	9	8
2	0	8	7
3	0	7	6
4	0	6	5
5	0	5	4
6	0	4	3
7	0	3	2

8	0	2	1
9	0	0	0

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

Porque para llegar al primer elemento se tienen que mover todos los elementos que lo suceden. Y al insertar al final solo se está agregando un elemento.

2. Detective del caso MegaStore

Fórmula n inserciones + 1

$$\frac{n(n-1)}{2}$$

Fórmula de duplicación de capacidad

$$\sum_{i=0}^{k-1} 2^i = 2^k - 1$$

Informe breve

Comparemos el sistema a una libreta en donde un estudiante toma notas. En el momento en el que se le acaban todas las hojas de la libreta, el estudiante consigue una nueva que incluye una hoja adicional. Luego, comienza a transcribir todo el contenido de la libreta anterior a la nueva. Una vez terminado eso, comienza a tomar notas en la hoja adicional. Cuando la llena, repite todo el proceso de nuevo. Es algo demasiado ineficiente y es lo mismo que está haciendo el algoritmo actual.

Checkpoint Metacognitivo - Fase COMPRENDE

¿Puedo explicar con mis propias palabras por qué arr[i] es O(1) usando la fórmula de dirección?

La fórmula de dirección consiste de una división y una suma. No hay bucles y la operación no depende del tamaño de entrada (n). Por lo mismo, la complejidad temporal siempre es la misma.

¿Qué imagen mental tengo ahora de un arreglo en memoria? ¿Es diferente a lo que pensaba antes?

Ya se me había presentado ese tipo de imagen en asignaturas anteriores. Simplemente lo imagino como una especie de lista que se encuentra en la memoria.

Si me preguntaran "¿por qué insertar al inicio es O(n)?", ¿podría dibujarlo paso a paso?

Sí. Solo debo mover cada elemento una posición a la derecha.

¿Qué conexión veo entre el análisis de complejidad de la Semana 1 y las operaciones de arreglos?

Las operaciones suelen ser lineales, probablemente porque la complejidad lineal involucra recorrer toda una lista de elementos, y eso es exactamente lo que es un arreglo.

3. Arquitecto de código

```
#include <iostream>

#include <stdexcept>



class ArregloDinamico {

private:

   int* arreglo;

   int tamaño;

   int capacidad;



   void redimensionar(int nuevaCapacidad) {

       int* nuevoArreglo = new int[nuevaCapacidad];



       for (int i = 0; i < tamaño; i++) {

           nuevoArreglo[i] = arreglo[i];

       }



       delete[] arreglo;

       arreglo = nuevoArreglo;

       capacidad = nuevaCapacidad;

   }
```

```
public:
    // Constructor
    ArregloDinamico(int capacidadInicial = 1) {
        if (capacidadInicial <= 0)
            capacidadInicial = 1;

        capacidad = capacidadInicial;
        tamaño = 0;
        arreglo = new int[capacidad];
    }

    // Destructor
    ~ArregloDinamico() {
        delete[] arreglo;
    }

    // Agregar al final
    void agregar(int elemento) {
        if (tamaño == capacidad) {
            redimensionar(capacidad * 2);
        }
```

```
    arreglo[tamaño] = elemento;
    tamaño++;
}

// Insertar en índice
void insertar(int indice, int elemento) {
    if (indice < 0 || indice > tamaño) {
        throw std::out_of_range("Índice inválido");
    }

    if (tamaño == capacidad) {
        redimensionar(capacidad * 2);
    }

    for (int i = tamaño; i > indice; i--) {
        arreglo[i] = arreglo[i - 1];
    }

    arreglo[indice] = elemento;
    tamaño++;
}
```

```
// Eliminar en índice
void eliminar(int indice) {
    if (indice < 0 || indice >= tamaño) {
        throw std::out_of_range("Índice inválido");
    }

    for (int i = indice; i < tamaño - 1; i++) {
        arreglo[i] = arreglo[i + 1];
    }

    tamaño--;

    if (tamaño > 0 && tamaño <= capacidad / 4) {
        redimensionar(capacidad / 2);
    }
}

// Obtener elemento
int obtener(int indice) const {
    if (indice < 0 || indice >= tamaño) {
        throw std::out_of_range("Índice inválido");
    }
```

```
        return arreglo[indice];
    }

    // Tamaño actual
    int getTamaño() const {
        return tamaño;
    }

    // Capacidad actual
    int getCapacidad() const {
        return capacidad;
    }

    // Mostrar contenido
    void imprimir() const {
        std::cout << "[ ";
        for (int i = 0; i < tamaño; i++) {
            std::cout << arreglo[i] << " ";
        }
        std::cout << "]\n";
    }
};
```

Tabla de complejidades

Método	Complejidad temporal	Complejidad espacial
Agregar	O(1) amortizado	O(n)
Eliminar	O(n)	O(n)
Insertar	O(n)	O(n)

Casos borde

1. Comprueba que el índice sea válido
2. Establece una capacidad inicial para evitar que la capacidad sea 0, lo que haría imposible redimensionar el arreglo
3. Comprueba si el arreglo está lleno. Sí lo está, lo redimensiona.

4. Laboratorio de Problemas Clásicos

1.

Pseudocódigo

```
Función RotarArreglo(nums, k):
    n = longitud(nums)
    k = k % n  // Manejar casos donde k > n

    // Paso 1: Invertir todo el arreglo
    InvertirRango(nums, 0, n - 1)

    // Paso 2: Invertir los primeros k elementos
    InvertirRango(nums, 0, k - 1)

    // Paso 3: Invertir el resto (desde k hasta el final)
    InvertirRango(nums, k, n - 1)

Función InvertirRango(nums, inicio, fin):
    Mientras inicio < fin:
        Intercambiar nums[inicio] por nums[fin]
        inicio = inicio + 1
        fin = fin - 1
```

Análisis de complejidad

Temporal: O(n)
Espacial: O(1)

2.

Pseudocódigo

```
Función EliminarDuplicados(nums):
   Si longitud de nums == 0: retornar 0

   indice_unico = 0  // Puntero de escritura

   Para i desde 1 hasta longitud(nums) - 1:
      Si nums[i] != nums[indice_unico]:  // Encontramos un número nuevo
         indice_unico += 1
         nums[indice_unico] = nums[i]   // Lo movemos a la posición que sigue

   Retornar indice_unico + 1
```

Análisis de complejidad

Temporal: O(n)
Espacial: O(1)

3.

Pseudocódigo

```
Función MoverCeros(nums):
   posicion_escritura = 0

   // Paso 1: Mover todos los no-ceros al frente
   Para i desde 0 hasta longitud(nums) - 1:
      Si nums[i] != 0:
         nums[posicion_escritura] = nums[i]
         posicion_escritura += 1

   // Paso 2: Rellenar el resto con ceros
   Mientras posicion_escritura < longitud(nums):
      nums[posicion_escritura] = 0
      posicion_escritura += 1
```

Análisis de complejidad

Temporal: O(n)
Espacial: O(1)

4.

Pseudocódigo

```
Función ElementoMayoritario(nums):
    candidato = nulo
    votos = 0

    Para cada numero en nums:
        Si votos == 0:
            candidato = numero

        Si numero == candidato:
            votos += 1
        Sino:
            votos -= 1

    Retornar candidato
```

Análisis de complejidad

Temporal: O(n)
Espacial: O(1)

Checkpoint Metacognitivo - Fase APLICA

¿Puedo implementar `agregar()` sin mirar mis notas? ¿Qué parte me cuesta más recordar?

Sí puedo.

Cuando la IA me hizo preguntas socráticas, ¿en qué momento me di cuenta de algo que no había considerado?

Me corrigió algunas confusiones que tenía, como que los elementos que se mueven deben de empezar su movimiento de derecha a izquierda

De los 4 problemas, ¿cuál me costó más trabajo? ¿Qué patrón o técnica me faltaba?

Todos se me hicieron algo complicados. El concepto de funciones es algo que no comprendo muy bien.

¿Cómo cambió mi aproximación a los problemas entre el primero y el último?

Decidí simplificar y pensar en la manera más rápida de resolver el problema

5. Diálogo socrático sobre decisiones de diseño

1. Capacidad inicial de 10

La elección de una capacidad inicial de 10 representa un equilibrio pragmático entre la economía de memoria y la preparación para la carga inicial. En entornos de propósito general, la mayoría de las colecciones de datos (como listas de parámetros o resultados de formularios) contienen pocos elementos. Al asignar 10 espacios, evitas múltiples redimensionamientos en los primeros instantes de vida del objeto sin comprometer significativamente el "heap". Es una decisión que prioriza la simplicidad y el rendimiento en el "caso promedio" de las aplicaciones empresariales, donde el costo de unos pocos bytes desperdiciados es despreciable comparado con la claridad del código.

Escenario donde la cambiaría

Si estuviera diseñando una estructura para representar un grafo masivo (como una red social) donde cada nodo tiene un arreglo de conexiones.

2. Factor de crecimiento de 2x

El factor 2x es una decisión orientada a la eficiencia temporal, garantizando una complejidad amortizada de O(1) para la operación de inserción. Matemáticamente, este crecimiento geométrico asegura que la cantidad de veces que el sistema debe detenerse para reasignar y copiar datos sea logarítmica respecto al tamaño total (log2n). Al duplicar el espacio, se apuesta por la probabilidad de que, si el arreglo ha crecido hasta cierto punto, seguirá necesitando una cantidad proporcional de espacio adicional, minimizando la latencia en aplicaciones que reciben flujos de datos impredecibles o en ráfagas.

Escenario donde la cambiaría

En sistemas con memoria virtual crítica o muy fragmentada; usaría un factor de 1.5x o un crecimiento aritmético para evitar solicitar bloques contiguos gigantescos que el sistema operativo no pueda satisfacer.

3. No implementación de reducción (Shrink)

Al omitir la reducción de capacidad, se prioriza la estabilidad del rendimiento y se evita el fenómeno de la "histéresis" o thrashing (pedir y liberar memoria constantemente en el límite de capacidad). Esta decisión asume que la memoria, una vez reclamada, es un activo del objeto para manejar picos de trabajo futuros sin incurrir nuevamente en el costo de asignación del sistema operativo. Es una postura defensiva en sistemas de alta frecuencia donde la latencia de recolectar basura o reasignar bloques de memoria es más costosa que el almacenamiento pasivo de datos vacíos.

Escenario donde la cambiaría

En aplicaciones de larga ejecución que tienen picos de carga muy esporádicos.

4. Lanzar excepción fuera de rango

Esta decisión refuerza el "Diseño por Contrato", estableciendo una frontera clara de responsabilidad. Al lanzar una excepción, el sistema adopta una postura de Fail-Fast, impidiendo que un error lógico del programador (como un índice mal calculado) se propague y corrompa el estado de la aplicación o produzca resultados silenciosamente erróneos. Es una medida de integridad que facilita la depuración inmediata, asegurando que el flujo de ejecución solo continúe si se cumplen las precondiciones de acceso, lo cual es vital en software donde la corrección de los datos es innegociable.

Escenario donde la cambiaría

En sistemas embebidos de tiempo real donde el manejo de excepciones es demasiado costoso en ciclos de CPU o memoria

5. Copia elemento por elemento

La copia manual es la máxima expresión de transparencia y portabilidad en el diseño. Al iterar uno a uno, el código se mantiene independiente de las optimizaciones específicas de la plataforma o de funciones de bajo nivel que podrían no estar disponibles en todos los entornos. Esta implementación permite, además, una flexibilidad total: si en el futuro se requiere lógica adicional durante el traslado (como notificar a observadores o transformar los datos), el punto de extensión ya existe. Es un diseño "legible por humanos" que prioriza la mantenibilidad y la facilidad de seguimiento del flujo de datos sobre la velocidad bruta.

Escenario donde la cambiaría

En sistemas de alto rendimiento.

Decisión que cambiaría

Implementaría una reducción de capacidad, pero solo en casos en donde sea necesario debido a que se está desperdiciando mucha memoria.

6. Identificación de patrones en problemas de arreglos

Tarjeta de DOS PUNTEROS

Señales

- Deben compararse dos posiciones
- Buscar pares
- Se quiere evitar un doble bucle

Idea general

Usa dos índices que se mueven según una condición para reducir combinaciones.

Tarjeta de VENTANA DESLIZANTE

Señales

- Hay un subarreglo
- Se menciona un segmento continuo
- Se debe sumar o contar

Idea general

Mantiene una “ventana” válida que se expande y se contrae sin reiniciar cálculos desde cero.

Tarjeta de IN-PLACE

Señales

- No se pueden generar más arreglos
- Solo se busca reorganizar el arreglo
- No importa lo que quede al final

Idea general

Reutiliza el arreglo original sobrescribiendo posiciones.

Tarjeta de PREFIJO/SUFIJO

Señales

- Rangos
- Acumulación
- Sumas o productos repetidos

Idea general

Precalcula información acumulada para responder rápido sin repetir trabajo.

Palabras Clave

1. Dos punteros
    - Pares
    - Extremos
2. Ventana deslizante
    - Continuo
    - Subarreglo
3. In-Place
    - Reorganizar
    - Poco espacio
    - Mismo arreglo
4. Prefijo/Sufijo
    - Rango
    - Acumulación

Checkpoint Metacognitivo - Fase REFLEXIONA

¿Qué decisión de diseño me costó más justificar? ¿Qué dice eso sobre mi comprensión?

Copiar elementos uno por uno. No conocía muy bien las otras técnicas.

¿Qué patrón de los explorados era nuevo para mí? ¿Puedo ahora reconocerlo en un problema?

La verdad todos me parecieron nuevos. Debo estudiarlos más para comprenderlos mejor.

Si tuviera que enseñar "arreglos dinámicos" a un compañero, ¿por dónde empezaría?

Los arreglos deben tener un factor de crecimiento para evitar tener que crear muchos nuevos arreglos al momento de agregar un nuevo elemento.

¿Qué conexión inesperada descubrí entre los conceptos de esta semana?

La manera en que los patrones se conectan con los bucles

7. Cazador de bugs

Implementación 1

Bug identificado

No se verifica que la variable capacidad tenga un valor mínimo de 1, lo que puede causar que la capacidad nunca pueda incrementarse si el valor inicial es 0.

Línea específica

```
nuevaCapacidad ← capacidad * 2
```

Corrección propuesta

```
Si capacidad == 0 Entonces
   nuevaCapacidad ← 1
Sino
   nuevaCapacidad ← capacidad * 2
Fin Si
```

Caso de prueba mínimo

```
arr ← nuevo ArregloDinamico(0)
arr.agregar(10)
```

Implementación 2

Bug identificado

El desplazamiento se hace de izquierda a derecha, lo que hace que se sobrescriban elementos.

Línea específica

```
Para i desde indice hasta tamaño - 1
   datos[i + 1] ← datos[i]
Fin Para
```

Corrección propuesta

```
Para i desde tamaño - 1 hasta indice
   datos[i + 1] ← datos[i]
Fin Para
```

Caso de prueba mínimo

```
arr.agregar("A")
arr.agregar("B")
arr.agregar("C")

arr.insertar(1, "X")

["A", "X", "B", "B"]
```

Implementación 3

Bug identificado

El índice de la última iteración está fuera del rango del arreglo.

Línea específica

```
Para i desde indice hasta tamaño - 1
   datos[i] ← datos[i + 1]
Fin Para
```

Corrección propuesta

```
Para i desde indice hasta tamaño - 2
   datos[i] ← datos[i + 1]
Fin Para
```

Caso de prueba mínimo

```
arr.agregar("A")
arr.agregar("B")
arr.agregar("C")

arr.eliminar(2)   // eliminar último elemento
// Se intenta leer datos[3], elemento que está fuera de rango
```

Implementación 4

Bug identificado

La variable tamaño no se inicializa, lo que provoca que pueda tomar cualquier valor almacenado en la dirección de memoria a la que sea asignada.

Línea específica

```
Constructor(capacidadInicial)
   capacidad ← capacidadInicial
   datos ← nuevo arreglo de tamaño capacidad
Fin Constructor
```

Corrección propuesta

```
tamaño ← 0
```

Caso de prueba mínimo

```
arr ← nuevo ArregloDinamico(10)
arr.agregar("A")
```

Ranking de bugs de más fácil a más difícil

1. Bug en eliminar(indice)

Muestra un error de forma inmediata al eliminar el último elemento.

2. Bug en insertar(indice, elemento)

No lanza errores, pero corrompe datos. Es detectable con un test específico de inserción en el medio. Requiere inspeccionar el contenido del arreglo, pero el patrón de duplicación ayuda a identificar la causa.

3. Bug en agregar()

Solo aparece con capacidades iniciales extremas como cero. Muchos tests no cubren ese caso. El código parece correcto y el fallo ocurre en un borde poco considerado, lo que retrasa su detección.

4. Bug en constructor

Como los valores almacenados en memoria pueden variar de máquina a máquina, es muy difícil encontrar el error en producción.

8. Benchmark empírico

Tabla de datos del experimento 1

⏱️ Tiempo total simulado (unidades abstractas)

Estrategia	n = 1,000	n = 10,000	n = 100,000
Incremento +1	500,500	50,005,000	5,000,050,000
Incremento +10	50,500	5,050,000	505,000,000
Incremento +100	5,500	550,000	55,000,000
Factor 1.5×	2,950	29,500	295,000
Factor 2×	1,999	19,999	199,999

Sigue el modelo teórico de crecimiento exponencial y de crecimiento amortizado en el caso del crecimiento por factor.

Tabla de datos del experimento 2

⏱️ Tiempo de inserción según posición

Posición	Elementos movidos	Tiempo simulado
0	10,000	10,001
2,500	7,500	7,501
5,000	5,000	5,001
7,500	2,500	2,501
9,999	0	1

Al instertar un elemento se mueven todos los elementos a la derecha del índice una posición hacia la derecha, es por eso que insertar al inicio es el más tardado, y explica por qué insertar al final es O(1), ya que no hay más elementos después del último elemento

Tabla de datos del experimento 3

📦 Capacidad vs tamaño (ejemplos clave)

Tamaño real	Capacidad	Memoria desperdiciada
1	1	0%
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%
512	512	0%
513	1024	**49.9%**
1000	1024	2.3%

Después de cada duplicación, el desperdicio de memoria se acerca al 50% en el peor de los casos, pero nunca pasa de ese límite. El desperdicio es de 0% cuando el arreglo se llena, y cuando se llena se reinicia el ciclo de crecimiento.

Gráfica ASCII del experimento 1

```
Incremento +1     | ##############################
Incremento +10    | ####################
Incremento +100   | ##########
Factor 1.5×       | #####
Factor 2×         | ####
```

Conclusiones

¿Los datos empíricos confirman la teoría? ¿Hubo sorpresas?

Sí la confirman. No noté ninguna sorpresa o diferencia a la teoría.

Checkpoint Metacognitivo - Fase VALIDA

¿Cuál de los 4 bugs me costó más encontrar? ¿Qué estrategia de búsqueda me funcionó mejor?

El segundo, me tomó trabajo deducir en que parte del método se encontraba el bug.

¿Los datos empíricos me sorprendieron en algo? ¿Por qué sí o por qué no?

No, porque ya habíamos repasado sobre esto en las actividades de la última semana

Si tuviera que diseñar pruebas para el código de otro compañero, ¿qué probaría primero?

Los límites de los bucles que utiliza, es decir, asegurarme de que los índices no salgan del rango del arreglo.

¿Cómo ha cambiado mi confianza en mi propia implementación después de esta fase?

Puede que ahora sea un poco más capaz de escribir mi propio código y de detectar errores comunes como lo es salirse del rango del arreglo.