Documento de reflexión sobre decisiones de diseño

1. Capacidad inicial = 10

Justificación refinada:
Elegir 10 como capacidad inicial es un compromiso razonable para evitar redimensionamientos inmediatos sin desperdiciar demasiada memoria. Funciona bien cuando se espera un número pequeño o moderado de inserciones iniciales.
Cuándo la cambiaría:
La cambiaría si supiera de antemano que el arreglo siempre iniciará vacío y crecerá poco (capacidad menor), o si casi siempre se cargan muchos datos desde el inicio (capacidad mayor).

2. Factor de crecimiento = 2x

Justificación refinada:
Duplicar la capacidad equilibra bien el costo de redimensionar con el desperdicio de memoria y garantiza O(1) amortizado para inserciones. Es un estándar probado en bibliotecas reales.
Cuándo la cambiaría:
En sistemas con memoria muy limitada o patrones de crecimiento predecibles, usaría un factor menor (por ejemplo 1.5x) para reducir memoria desperdiciada.

3. No implementar reducción de capacidad

Justificación refinada:
Evita oscilaciones constantes de tamaño (thrashing) cuando el arreglo crece y decrece repetidamente. Prioriza estabilidad y rendimiento sobre ahorro de memoria.
Cuándo la cambiaría:
Si el patrón típico es “cargar mucho y luego eliminar casi todo”, implementaría reducción con un umbral (por ejemplo, reducir cuando tamaño < capacidad / 4).

4. Lanzar excepción si el índice está fuera de rango

Justificación refinada:
Falla rápido y de forma explícita. Obliga al usuario de la clase a corregir su lógica en lugar de ocultar errores.
Cuándo la cambiaría:
En sistemas críticos o embebidos donde lanzar excepciones es costoso, podría usar códigos de error o validaciones previas.

5. Copiar elementos uno por uno al redimensionar

Justificación refinada:
Es simple, claro y portable. Hace explícito el costo del redimensionamiento y no depende de primitivas del lenguaje.
Cuándo la cambiaría:
En lenguajes o entornos que ofrecen copias de memoria optimizadas (memcpy, System.arraycopy), usaría esas primitivas para mejorar rendimiento.

Escenarios extremos

Escenario donde la implementación es perfecta

Aplicación de negocio o backend general (por ejemplo, manejo de catálogos, listas de usuarios, buffers de datos) donde las inserciones son frecuentes y el tamaño crece de forma gradual.

Escenario donde es la peor opción

Sistema en tiempo real o embebido con memoria muy limitada y patrones de inserción/eliminación impredecibles, donde el desperdicio de memoria y las copias son inaceptables.

Ajustes por contexto

Sistema embebido (512 KB de RAM)

- Capacidad inicial pequeña y ajustada
- Factor de crecimiento menor (1.25x o 1.5x)
- Implementar reducción de capacidad
- Evitar excepciones
- Copias de memoria altamente controladas

Servidor con terabytes de RAM

- Capacidad inicial mayor
- Factor de crecimiento 2x o mayor
- No reducir capacidad
- Priorizar simplicidad y rendimiento
- Usar primitivas de copia rápidas

Matriz: contexto vs configuración ideal

Contexto	Capacidad inicial	Crecimiento	Reducción	Manejo de errores
App académica	10	2x	No	Excepciones
Backend empresarial	50–100	2x	No	Excepciones
Sistema embebido	2–5	1.25x	Sí	Códigos de error
Sistema de alto rendimiento	1000+	2x–4x	No	Excepciones

Decisión que cambiaría de la implementación original

Cambiaría no implementar reducción de capacidad.
Ahora veo que, aunque es una buena decisión general, introducir una reducción con umbral bien definido hace la estructura más adaptable sin sacrificar demasiado rendimiento. Esto la vuelve más robusta frente a patrones de uso reales y no solo al caso promedio.