

Estructuras de Datos

Arreglos dinámicos

Costo amortizado, invalidación y colas sobre un buffer circular

Gerson Garrido

Semana 4

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Tamaño y capacidad: qué pasa cuando se llena
2. Crecer de a k frente a duplicar
3. Costo amortizado
4. Encoger sin oscilar
5. Mover o copiar al crecer: `noexcept`
6. Referencias que se invalidan
7. Un `Vector` propio, y sus límites
8. La cola sobre un buffer circular

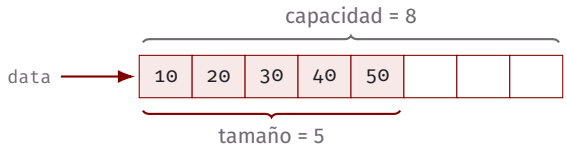
> **Lectura**
>
> Apuntes de la semana 4. Código en `code/week-04`.

SECCIÓN 01

Tamaño y capacidad

01

Dos números distintos



Arreglo dinámico *(dynamic array)*

Un arreglo que cambia de capacidad cuando hace falta. **Tamaño**: cuántos elementos tiene. **Capacidad**: cuántos caben antes de tener que crecer.

- Invariante: $0 \leq$ tamaño $\leq$ capacidad.
- Con espacio libre, `push_back`, `pop_back` y `v[i]` son $\Theta(1)$.

Qué pasa cuando se llena

Lo de al lado del bloque puede estar ocupado, así que no se puede estirar.



La pregunta de hoy

El paso 2 cuesta Θ(tamaño). ¿Cuánto más grande debe ser el bloque nuevo?

Lo que hace `std::vector` aquí

terminal

```
$ build/capacity_growth
size 0: capacity 0
size 1: capacity 1
size 2: capacity 2
size 3: capacity 4
size 5: capacity 8
size 9: capacity 16
size 17: capacity 32
size 33: capacity 64
```

- La capacidad no cambia en cada inserción.
- Con este `g++`, se duplica.

No es una regla del lenguaje

El estándar exige `push_back` en tiempo constante amortizado, pero no fija el factor. Un programa correcto no depende de ese 2.

SECCIÓN 02

Crecer de a k frente a duplicar

02

Primera idea: pedir solo un poco más

Si crece de a k, crece cuando el tamaño llega a $k, 2k, 3k, \ldots$ y cada vez traslada todo.

Tamaño al crecer	Traslados	Capacidad nueva
0	0	k
k	k	$2k$
$2k$	$2k$	$3k$
$\ldots$	$\ldots$	$\ldots$
mk	mk	$(m+1)k$

Para n inserciones, el último crecimiento tiene $mk \leq n-1$, así que $m = \lfloor (n-1)/k \rfloor$.

Sumar emparejando

$$k + 2k + \cdots + mk \;=\; k\,(1 + 2 + \cdots + m)$$

$$\begin{array}{ccccccccc}
1 & + & 2 & + & \ldots & + & m & & \\
m & + & (m-1) & + & \ldots & + & 1 & & \\
\hline
(m+1) & + & (m+1) & + & \ldots & + & (m+1) & \to & m \text{ veces } (m+1)
\end{array}$$

$$\text{traslados} \;=\; k \cdot \frac{m(m+1)}{2} \;=\; \Theta\left(\frac{n^2}{k}\right)$$

$n = 1\,000\,000$, $k = 1000$: $m = 999$, y $1000 \cdot 999 \cdot 1000/2 = 499\,500\,000$ traslados. Casi quinientos por inserción.

Segunda idea: multiplicar

Inserción	Tamaño antes	Cap. antes	Traslados	Cap. después	Acumulado
1	0	0	0	1	0
2	1	1	1	2	1
3	2	2	2	4	3
4	3	4	0	4	3
5	4	4	4	8	7
6 a 8	5 a 7	8	0	8	7
9	8	8	8	16	15
10 a 16	9 a 15	16	0	16	15

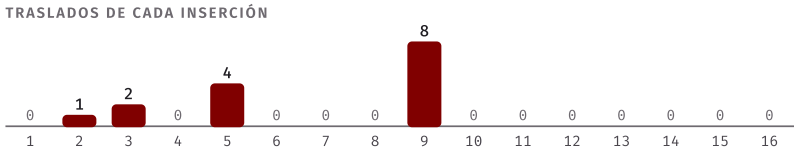
16 inserciones

$1 + 2 + 4 + 8 = 15$ traslados. El elemento 17 traslada 16 más: el total salta a 31.

Duplicar, inserción por inserción

inserción	tamaño	capacidad	traslados de esta inserción	traslados en total
16	16	16	0	15





16 inserciones, 1 + 2 + 4 + 8 = 15 traslados. La inserción 17 trasladaría 16, y el total saltaría a 31

La suma, sin inducción: duplicar y restar

Sea 2^j la mayor potencia de 2 con $2^j \leq n-1$. Entonces $S = 1+2+4+\cdots+2^j$.

$$\begin{array}{rcccccccc} 2S = & & 2 & + & 4 & +\cdots+ & 2^j & + & 2^{j+1} \\ S = & 1 & + & 2 & + & 4+\ldots & + & 2^j & \\ \hline S = & 2^{j+1}-1 & & & & & & & \end{array}$$

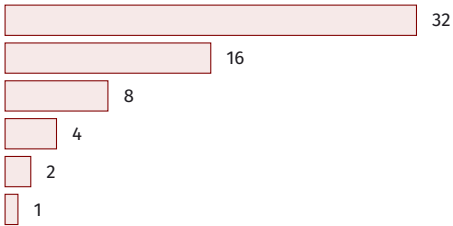
Como $2^j \leq n-1$ y $2^{j+1} \geq n$:

$$n-1 \;\leq\; S \;\leq\; 2n-3 \;<\; 2n \qquad (n \geq 2)$$

Cuidado

"Menos de n copias" es falso: con $n = 17$ hay 31. La cota correcta es **menos de 2n**.

Por qué funciona



Idea clave

Cada bloque es más grande que todos los anteriores juntos. La última copia domina la suma, y la suma completa no llega ni al doble de ella.

El factor no tiene que ser 2. Multiplicar por 3/2 da entre $2n$ y $3n$. Lo que importa es **multiplicar**, no sumar.

Las cuentas exactas

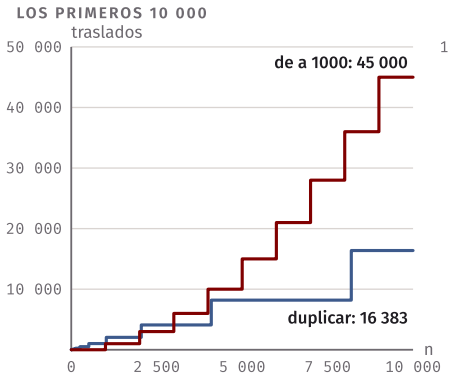
terminal

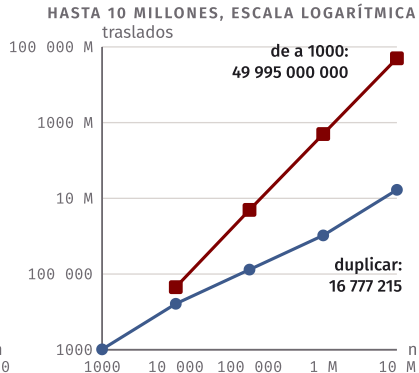
```
$ build/growth_count
         n     duplicar      por 3/2      de a 1000
      1000         1023         2137              0
     10000        16383        24284          45000
   1000000      1048575      2099753      499500000
  10000000     16777215     23917332    49995000000
```

Con $n = 1000$, de a 1000 **gana**: una sola reserva. Con $n = 10\,000\,000$, hace casi 3000 veces más trabajo.

> **¿Por qué sigue siendo cuadrático, si 1000 es grande?**
>
> k está fijo y n crece. Cada crecimiento traslada todo lo que hay, y cada vez hay más.

De a 1000 frente a duplicar





De a 1000 parte ganando, con 0 traslados en n = 1000, y termina casi 3000 veces por encima

Y en el cronómetro

n	duplicar (ms)	de a 1000 (ms)
100 000	0,369	0,951
200 000	0,735	3,126
400 000	1,561	10,970
800 000	3,076	40,678
1 600 000	5,906	155,511

`std::vector<int>`, `-O2`, sin sanitizers, mediana de 5. Una corrida en el entorno del curso, con otros programas en la misma máquina: en su laptop serán otros tiempos.

- Duplicar: cada fila, cerca de $\times 2$.
- De a 1000: $\times 3{,}3$, $\times 3{,}5$, $\times 3{,}7$, $\times 3{,}8$, y se acerca a 4.
- No llega de golpe porque también están las n escrituras, que son lineales.

Lo que se compara

Los factores, no los milisegundos.

SECCIÓN 03

Costo amortizado

03

Costo amortizado

Costo amortizado ***(amortized cost)***

Sea $T(n)$ el mayor costo total posible de una serie de n operaciones que empieza con la estructura vacía. El costo amortizado por operación es $T(n)/n$.

- Toma el **peor caso** de la serie.
- Promedia **dentro de la serie**.
- Parte de un estado conocido: **vacío**.

Sumar todo y dividir es el **método agregado**.

El costo de `push_back`

Una unidad por cada elemento escrito: el valor insertado o un traslado.

$$T(n) < \underbrace{n}_{\text{valores}} + \underbrace{2n}_{\text{traslados}} = 3n \qquad \Rightarrow \qquad \frac{T(n)}{n} < 3$$

Con 16 inserciones

16 escrituras + 15 traslados = 31 unidades. $31/16 \approx 1{,}94$, menos de 3.

Con `new T[c]`, además se construyen las c casillas si T tiene constructor propio. Esas capacidades también son una suma geométrica, menor que $4n$: el total sigue siendo $O(n)$.

Una llamada puede ser cara

¿Una sola llamada a `push_back` puede costar $\Theta(n)$? **Sí.** El elemento 17 traslada 16.

Afirmación	Habla de	Valor
Peor caso de una llamada	La llamada más cara	$\Theta(n)$
Costo amortizado	El total de la serie, entre n	$O(1)$

Error común

Leer "$O(1)$ amortizado" como "cada llamada es rápida". Si deben cumplir un plazo en cada operación, esa llamada cara se lo puede hacer perder. Si conocen el tamaño, usen `reserve`.

Amortizado no es caso promedio

	Caso promedio	Costo amortizado
Promedia sobre	Las entradas posibles	Las operaciones de una serie
¿Probabilidades?	Sí: supone cómo se distribuyen	No, ninguna
¿Depende de los datos?	Sí	No
¿Qué garantiza?	Nada sobre una ejecución concreta	Vale para toda serie

Una garantía

Si hacen n `push_back` desde vacío, el total nunca pasa de $3n$. Sin excepción, y sin suerte.

Una trampa: `reserve` en cada inserción

Pedir `reserve` antes de cada inserción, “para ayudar”, es crecer de a 1:

terminal

```
$ build/vector_facts
...
reserve(size + 1) antes de 1000 push_back: copias 499500
```

Error común

`v.reserve(v.size() + 1)` antes de cada `push_back`: con este `g++`, `reserve` pide lo justo. 499 500 copias, frente a 1023 de duplicar. `reserve` es para cuando conocen el tamaño final: se llama una vez.

SECCIÓN 04

Encoger sin oscilar

04

La regla obvia es mala

Encoger a la mitad cuando el tamaño cae a la mitad, con capacidad 1024 y tamaño 1024:

Operación	Tamaño	Capacidad	Traslados
push	1025	2048	1024 (crece)
pop	1024	1024	1024 (encoge)
push	1025	2048	1024 (crece)
pop	1024	1024	1024 (encoge)

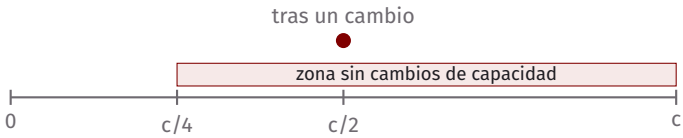
El problema

Cada operación traslada mil elementos. Justo después de crecer, el arreglo queda en el borde de la regla de encoger.

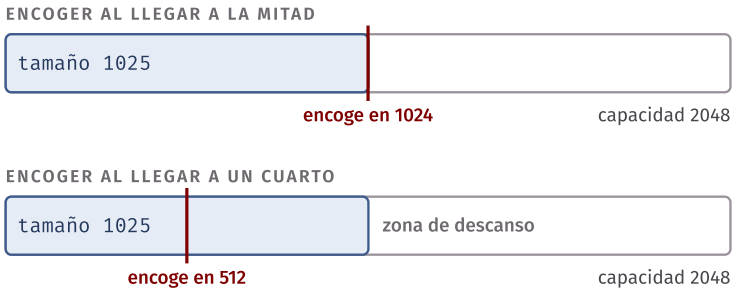
Encoger al llegar a un cuarto

terminal

```
$ build/shrink_policies
Encoger al llegar a la mitad
tras 1025 push: size 1025, capacity 2048, traslados 2047
tras 2000 operaciones más: capacity 2048, traslados nuevos 2048000
Encoger al llegar a un cuarto
tras 1025 push: size 1025, capacity 2048, traslados 2047
tras 2000 operaciones más: capacity 2048, traslados nuevos 0
```



Las dos reglas, operación por operación



Con 1000 pares de pop y push: 2 048 000 traslados frente a 0, como en shrink_policies

Por qué un cuarto funciona: sumar por tramos

Tras un cambio a capacidad c, el tamaño queda en $c/2$ o $c/2 + 1$. Desde ahí:

Siguiente cambio	Traslados	Operaciones mínimas	Por operación
Crecer, con tamaño c	c	$c/2$	≤ 2
Encoger, con tamaño $c/4$	$c/4$	$c/4$	≤ 1

Es el mismo método agregado: se suma tramo por tramo.

$$\text{traslados totales} \;\leq\; 2 \cdot \text{operaciones}$$

En la biblioteca

`std::vector` no encoge solo: 999 `pop_back` dejan la capacidad en 1024. `shrink_to_fit` es solo una solicitud.

SECCIÓN 05

Mover o copiar al crecer

05

Lo mínimo sobre excepciones

- `throw` abandona la función. El programa deshace llamadas, destruyendo los objetos locales, hasta un `catch`.
- `new` no devuelve nulo: lanza `std::bad_alloc`. Copiar una cadena larga puede lanzar.
- `noexcept` es una promesa: si una excepción sale de ahí, `std::terminate`.

Lo que RAII no cubre

Lo que está en manos de un objeto se libera solo. Lo que está en un puntero suelto, no: es una fuga.

El molde para un bloque sin owner

copy_block.cpp

```cpp
std::string* copy_block(const std::string* source, std::size_t count) {
    std::string* fresh = new std::string[count];
    try {
        for (std::size_t i = 0; i < count; i++) {
            fresh[i] = checked_copy(source[i]);
        }
    } catch (...) {
        delete[] fresh;
        throw;
    }
    return fresh;
}
```

`catch (...)` recibe cualquier excepción. `throw;` sola la vuelve a lanzar.

Falla el tercer traslado: ¿qué queda?

Si estaba copiando



viejo, intacto



nuevo

Se libera el nuevo. Nadie perdió nada.

Si estaba moviendo



viejo, vaciado



nuevo

Para deshacer habría que mover de vuelta, y eso también puede fallar.

Garantía fuerte

Si `push_back` falla, el vector queda como estaba. Por eso `std::vector` solo mueve si moverse no puede fallar.

La regla: `std::move_if_noexcept`

El tipo T	Al crecer	¿Garantía fuerte?
Se mueve con `noexcept`	Mueve	Sí
Sin `noexcept`, y se puede copiar	Copia	Sí
Sin `noexcept`, y no se puede copiar	Mueve	No

Es solo un cast

Como `std::move`, no hace nada al ejecutarse. Da algo de lo que se puede mover o una referencia `const`, según lo que prometa la **construcción** por movimiento de T.

Una palabra cambia lo que hace el programa

terminal

```
$ build/move_or_copy
mover con noexcept: capacity 16, copias 0, movimientos 31
mover sin noexcept: capacity 16, copias 15, movimientos 16
```

- 16 movimientos son los argumentos, en los dos casos.
- Los 15 traslados de $1 + 2 + 4 + 8$ son movimientos arriba y copias abajo.

Error común

Escribir el movimiento y olvidar `noexcept`. Compila, funciona y pasa las pruebas, pero `std::vector` copia todo al crecer.

Un destructor vacío también lo cambia

move_or_copy.cpp

```cpp
struct Student {
    std::string name;
};

struct StudentWithDestructor {
    std::string name;
    ~StudentWithDestructor() {}
};
```

terminal

```
$ build/move_or_copy
...
Student: true
StudentWithDestructor: false
```

Declarar un destructor apaga el movimiento automático. “Mover” termina copiando, y copiar una cadena puede lanzar.

Si falla de verdad

terminal

```
$ build/strong_guarantee
excepción: falló una copia
size 4, capacity 4, valores: 1 2 3 4
excepción: falló un movimiento
size 4, capacity 4, valores: -1 -1 3 4
```

- Primero: se puede copiar. Falla una copia, y el vector queda **intacto**.
- Segundo: solo se puede mover, sin `noexcept`. Falla un movimiento, y **se perdieron dos valores**.

En el segundo caso, el estándar deja el efecto sin especificar: esto es lo que hace este `g++`, y no hay que contar con ello.

SECCIÓN 06

Referencias que se invalidan

06

El error común de la semana

errors/stale_reference.cpp

```cpp
std::vector<int> v = {10, 20, 30};
int& first = v[0];
v.push_back(40);
std::cout << first << "\n";
```

Queda invalidada

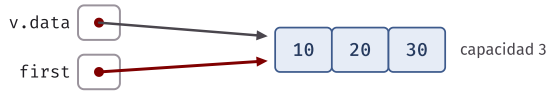
`first` no sabe que el vector creció. Sigue apuntando al bloque viejo. Sin ASan, puede imprimir el número correcto: que funcione hoy no prueba nada.

Lo que pasa en la memoria

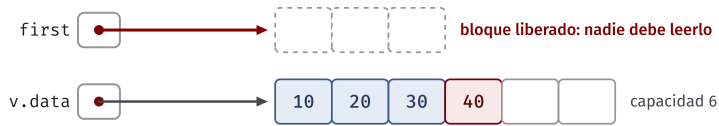
```
6  std::vector<int> v = {10, 20, 30};
7  int& first = v[0];
```

```
8  v.push_back(40);
9  std::cout << first << "\n";
```

DESPUÉS DE LA LÍNEA 7



DESPUÉS DE LA LÍNEA 8



first no cambió, y el vector no sabe que existe. La línea 9 lee memoria liberada

ASan cuenta la historia

terminal

```
$ build/stale_reference
==4921==ERROR: AddressSanitizer: heap-use-after-free on address 0x502000000010 at pc 0xaaaab76a9c04 bp 0xffffcf714ff0 sp 0xffffcf715008
READ of size 4 at 0x502000000010 thread T0
    #0 0xaaaab76a9c00 in main errors/stale_reference.cpp:9
...
freed by thread T0 here:
...
    #8 0xaaaae83c9a7c in main errors/stale_reference.cpp:8
...
previously allocated by thread T0 here:
...
    #7 0xaaaae83c9978 in main errors/stale_reference.cpp:6
```

- Línea 6: creó el bloque.
- Línea 8: `push_back` lo liberó.
- Línea 9: lo leyó a través de `first`.

Lo mismo pasa si se hace `push_back` dentro de un `for` de rango sobre el mismo vector.

Cuándo se invalida y qué hacer

terminal

```
$ build/stable_refs
1000 push_back: capacity 1024, el bloque cambió 11 veces
1000 push_back: capacity 1000, el bloque cambió 0 veces
grades[first] = 10
```

- Si la capacidad **cambia**: se invalida todo.
- Si **no cambia**: siguen válidas las referencias a elementos. Solo se invalida `end()`.

Las dos salidas

Guardar un **índice**, que se recalcula con el bloque actual. O usar **`reserve`** antes de empezar, si conocen el tamaño final.

Una trampa para quien escribe el arreglo

errors/self_push.cpp

```cpp
void push_back(const std::string& value) {
    if (size_ == capacity_) {
        ...
        delete[] data_;
    }
    data_[size_] = value;
```

Con `names.push_back(names[0])`, `value` está **dentro** del bloque liberado.

terminal

```
$ build/self_push
==4938==ERROR: AddressSanitizer: heap-use-after-free on address 0x504000000020 at pc 0xaaaad388d740 bp 0xffffff63f1550 sp 0xffffff63f1568
```

`std::vector` sí lo resuelve; el estándar lo exige. Su `Vector` también debe hacerlo: hay una prueba.

SECCIÓN 07

Un Vector propio

07

`new T[capacity]`: lo que cuesta

`new T[c]` hace dos cosas a la vez: pide memoria y construye c objetos. De ahí salen tres límites.

1. T debe tener **constructor sin argumentos**. Si no, no compila.
2. Hay **`capacity` objetos vivos**, no `size`.
3. `pop_back` **no destruye**: el objeto sigue vivo hasta que se sobrescriba o llegue `delete[]`.

> **Por qué lo hacemos así**
>
> La alternativa es memoria sin construir, construir en una dirección dada y destruir a mano. Eso no cambia lo que hoy importa: tamaño, capacidad, costo, mover o copiar.

Lo que hace `std::vector`, medido

terminal

```
$ build/lifetimes
new Tracked[100]: vivos 100, construidos sin argumentos 100, destruidos 0
delete[]: vivos 0, construidos sin argumentos 100, destruidos 100
reserve(100) de std::vector: vivos 0, construidos sin argumentos 0, destruidos 0
dos push_back: vivos 2, construidos sin argumentos 0, destruidos 2
un pop_back: vivos 1, construidos sin argumentos 0, destruidos 3
fin del vector: vivos 0, construidos sin argumentos 0, destruidos 4
std::vector<NoDefault> compila, y guarda 7
```

- `reserve(100)`: cero objetos vivos, con espacio para cien.
- Los 2 destruidos son los temporales pasados a `push_back`.
- `pop_back` sí destruye: de 2 pasa a 3.

Frente a frente

	`Vector<T>` del laboratorio	`std::vector<T>` en este `g++`
`reserve(100)` en vacío	100 objetos vivos	0 objetos vivos
T sin constructor sin argumentos	No compila	Compila
`pop_back`	El objeto sigue vivo	Destruye el objeto
Trasladar al crecer	Asigna	Construye en la memoria nueva
`push_back(v[0])`	Depende de ustedes	Funciona
Factor de crecimiento	2	2 aquí; el estándar no lo fija

Un detalle honesto

`std::move_if_noexcept` mira la **construcción** por movimiento, y nuestro `Vector` **asigna**. Si un tipo promete una y no la otra, una falla podría dejarlo a medio trasladar. Asumimos que prometen lo mismo, y el código lo dice.

Laboratorio: 2 horas

La otra sesión de esta semana es la **Práctica 1**.

1. Escribir `Vector<T>`: `push_back`, `pop_back`, `operator[]`, `size`, `capacity`, `reserve` y la regla de cinco.
2. Pasar las pruebas con ASan, incluidas las que cuentan copias con `tracked.h`.
3. Medir: duplicando frente a crecer de a 1000, y explicarlo con la cuenta.

Crecer de a 1000 no cambia su clase

El programa de medición lo hace desde fuera, con `reserve`. Basta con que `reserve` deje la capacidad exacta.

SECCIÓN 08

La cola

08

El tipo abstracto cola

Cola ***(queue)***

Lo que entra primero sale primero (FIFO, *first in, first out*), como una fila en el banco.

`enqueue(x)`	Agrega x al final
`dequeue()`	Quita el elemento del frente
`front()`	Devuelve el elemento del frente
`empty()`	Dice si está vacía

`std::queue` es una interfaz sobre otro contenedor: por defecto, `std::deque`.

En un arreglo normal, sacar del frente cuesta $\Theta(n)$

terminal

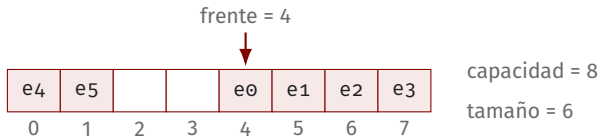
```
$ build/front_removal
vaciar 1000 elementos por el frente: 499500 movimientos
vaciar 2000 elementos por el frente: 1999000 movimientos
vaciar 4000 elementos por el frente: 7998000 movimientos
```

- Vaciar n cuesta $(n-1)+(n-2)+\cdots+1 = n(n-1)/2$. Cuadrático.
- Con un índice `frente` que solo avanza, sacar es $\Theta(1)$, pero las casillas de atrás se pierden para siempre.

La idea

Cuando el final llegue al borde, que dé la vuelta y use esas casillas.

Buffer circular



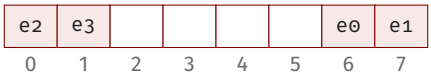
La fórmula

El elemento i de la cola está en la posición **(frente + i) mod capacidad**.

- `enqueue`: escribe en (frente + tamaño) mod capacidad.
- `dequeue`: frente $\leftarrow$ (frente + 1) mod capacidad.
- Nadie se mueve. Todo es $\Theta(1)$, mientras no haya que crecer.

Pregunta

Capacidad 8, frente en 6, tamaño 4. ¿En qué posiciones están los elementos?



terminal

```
$ build/ring_trace
...
capacidad 8, frente 6, tamaño 4: e0->6 e1->7 e2->0 e3->1
```

A mano: $(6+0) \bmod 8 = 6$, $(6+1) \bmod 8 = 7$, $(6+2) \bmod 8 = 0$, $(6+3) \bmod 8 = 1$.

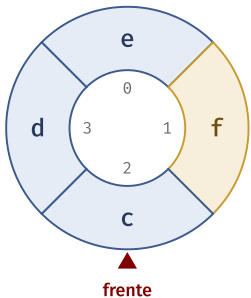
Una traza: capacidad 4

Op.	Posición	Bloque	frente	tamaño
E a	$(0+0)$ mód $4=0$	a . . .	0	1
E b	$(0+1)$ mód $4=1$	a b . .	0	2
E c	$(0+2)$ mód $4=2$	a b c .	0	3
D (sale a)	0	. b c .	1	2
D (sale b)	1	. . c .	2	1
E d	$(2+1)$ mód $4=3$	. . c d	2	2
E e	$(2+2)$ mód $4=0$	e . c d	2	3
E f	$(2+3)$ mód $4=1$	e f c d	2	4

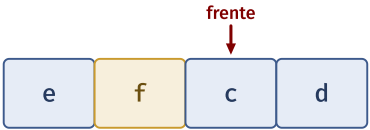
e queda antes que c en la memoria, y después en la cola: el orden lo dicta la fórmula.

La misma traza, en el anillo y en el bloque

COMO ANILLO



EN LA MEMORIA



operación	frente	tamaño	capacidad
E f	2	4	4

orden de la cola: c d e f

E f: (2 + 3) mod 4 = 1. La cola está llena, y el siguiente enqueue ya no cabe

¿Por qué guardar el tamaño?

Con dos índices, frente y final, estos dos estados se ven iguales:



vacía: frente = final = 2

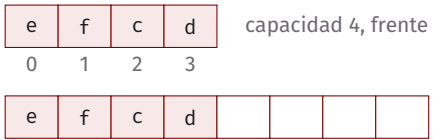


llena: frente = final = 2

Con frente y tamaño no hay ambigüedad

Vacía es tamaño 0. Llena es tamaño = capacidad.

Al crecer hay que desenrollar

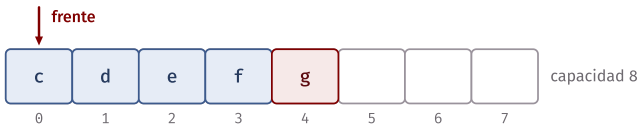


Error común

Según la fórmula, el elemento 2 estaría ahora en $(2 + 2)$ mód $8 = 4$, y esa casilla está vacía: la fórmula depende de la capacidad.

Desenrollar, casilla por casilla

frente	tamaño	capacidad	traslados
0	5	8	4



El elemento que no cabía entra en (0 + 4) mod 8 = 4. Ya no hay vuelta que recordar

El elemento i pasa de (frente + i) mod $\text{capacidad}_{\text{vieja}}$ a la posición i. El frente vuelve a 0.

Las tres implementaciones

Cola sobre	enqueue	dequeue	**Problema**
Arreglo, frente en 0	$O(1)$ amort.	$\Theta(n)$	Vaciar es cuadrático
Arreglo, índice que avanza	$O(1)$ amort.	$\Theta(1)$	Casillas perdidas
Buffer circular	$O(1)$ amort.	$\Theta(1)$	Ninguno de los dos

La cuenta es la misma

Desenrollar traslada cada elemento una vez, igual que crecer. Con duplicación: menos de $2n$ traslados en n enqueue.

Esta semana queda en papel. La próxima lo implementan, ampliado a un deque.

SECCIÓN 09

Cierre

09

Resumen

- Crecer de a k: $k \cdot m(m+1)/2$ traslados. $\Theta(n^2/k)$: cuadrático.
- Duplicar: $1 + 2 + 4 + \cdots < 2n$ traslados. Lineal.
- Amortizado: total de la peor serie, entre n. Una llamada puede costar $\Theta(n)$.
- Amortizado no es caso promedio: no hay probabilidades.
- Encoger al llegar a un cuarto, no a la mitad.
- `noexcept` decide si crecer mueve o copia.
- Crecer invalida referencias e iteradores.
- `new T[c]` tiene límites que `std::vector` no tiene.
- Buffer circular: (frente + i) mod capacidad. Al crecer se desenrolla.

Para la próxima clase

1. Partiendo de capacidad 1 y duplicando, ¿cuántas copias para 16 elementos? ¿Y para 17?
2. ¿Qué diferencia hay entre costo amortizado y caso promedio?
3. ¿Una llamada a `push_back` puede costar $\Theta(n)$? ¿Cómo se concilia con $O(1)$ amortizado?
4. ¿Por qué crecer de a 1000 sigue siendo cuadrático?
5. Capacidad 8, frente 6, tamaño 4: dibujen el bloque y hagan una traza a mano.

Esta semana

Laboratorio: `code/week-04/lab/README.md`, en una sola sesión. La otra sesión es la **Práctica 1**.