

Estructuras de Datos

Listas enlazadas y deques

Centinelas, iteradores y lo que O no ve

Gerson Garrido

Semana 5

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Nodos en la memoria
2. Lista simple: lo que abarata y lo que encarece
3. Lista doble con centinela
4. Iteradores: qué hace un `for` por rango
5. Cuándo se invalida un iterador
6. El tipo abstracto deque, con pruebas sin clase base
7. Dos representaciones con el mismo O(1)
8. Lo que O no ve: reservas de memoria y cache

> **Lectura**
>
> Apuntes de la semana 5. Código en `code/week-05`.

SECCIÓN 01

Nodos en la memoria

01



La decisión de fondo

Arreglo dinámico

- Elementos uno al lado del otro.
- Elemento i: O(1).
- Insertar al inicio: $\Theta(n)$.
- Al crecer, todo cambia de lugar.

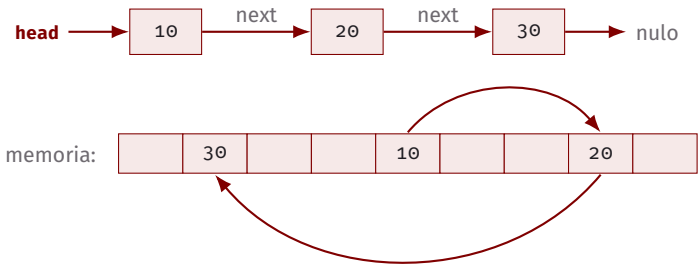
Lista enlazada *(linked list)*

- Cada elemento en su propio bloque.
- Elemento i: $\Theta(i)$.
- Insertar en un punto conocido: O(1).
- Ningún elemento se mueve jamás.

Nodo ***(node)***

Un bloque de memoria dinámica con un elemento y uno o más punteros a otros nodos.

El orden lo dan los punteros, no la memoria



Dibujar antes de programar: qué punteros cambian, y en qué orden.

Lo que hay en una memoria de verdad

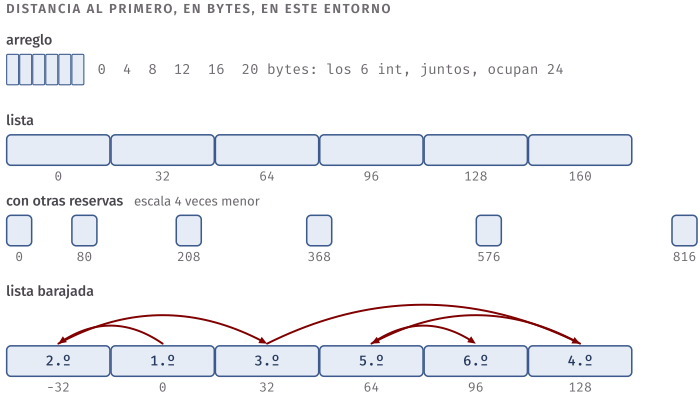
Distancia en bytes de cada elemento al primero. Depende de la biblioteca y de la máquina.

terminal

```
$ build/node_addresses
arreglo: 0 4 8 12 16 20
lista: 0 32 64 96 128 160
con otras reservas: 0 80 208 368 576 816
lista barajada: 0 -32 32 128 64 96
```

- **Arreglo:** contiguos, a `sizeof(int)`: 4 bytes aquí.
- **Lista recién creada:** a 32 bytes. Nadie lo prometió: el programa no reservó nada más. Un `int` ocupa ocho veces más.
- **Con otras reservas:** los nodos ya no están seguidos.
- **Barajada:** los mismos nodos, enlazados en otro orden.

Las cuatro filas, a escala



Los mismos nodos de la lista, enlazados en otro orden: recorrerla salta por la memoria

SECCIÓN 02

Lista simple

02

Una lista simplemente enlazada

slist.h

```
1  struct Node {
2      T value;
3      Node* next;
4  };
5
6  Node* head_ = nullptr;
7  Node* tail_ = nullptr;
8  std::size_t size_ = 0;
```

Invariantes

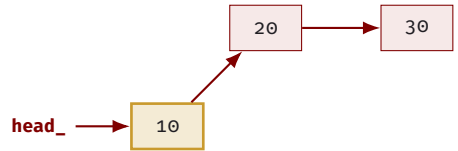
- `head_`, `tail_` nulos $\Leftrightarrow$ `size_ == 0`.
- Desde `head_` se visitan `size_` nodos, y el último es `tail_`.
- `tail_->next` es nulo.

Se comprueban en `check_invariants`, como en la semana 2.

Insertar al inicio: O(1)

slist.h

```
1  void push_front(const T& value) {
2      head_ = new Node{value, head_};
3      if (tail_ == nullptr) {
4          tail_ = head_;
5      }
6      size_++;
7  }
```



- El lado derecho se evalúa primero.
- Ningún elemento se mueve.
- O(1) si `new` cuenta como una operación. `new` no es gratis: lo medimos al final.

Sacar del inicio: el orden importa

slist.h

```
1  void pop_front() {
2      assert(!empty());
3      Node* old_head = head_;
4      head_ = old_head->next;
5      delete old_head;
6      if (head_ == nullptr) {
7          tail_ = nullptr;
8      }
9      size_--;
10 }
```

- Primero avanzar, y después borrar.
- Si se borrara primero, leer `next` sería un acceso a memoria liberada.
- Miren el `if`: la lista vacía obliga a tratar `tail_` aparte.

Casos especiales

Hay uno en `push_front`, uno en `pop_front` y uno en `push_back`. Olvidarlos es un bug clásico.

Llegar al elemento i: Θ(i)

slist.h

```
1  T& at(std::size_t i) {
2      assert(i < size_);
3      Node* node = head_;
4      for (std::size_t step = 0; step < i;
         step++) {
5          node = node->next;
6      }
7      return node->value;
8  }
```

- Sin aritmética de punteros: los nodos no están seguidos. Son i saltos.
- Un ciclo con `at(i)` cuesta $0 + 1 + \ldots + (n-1)$ saltos: $\Theta(n^2)$.
- Sacar del final con `tail_` tampoco funciona: hay que dejar `tail_` en el penúltimo, y un nodo no sabe quién está antes. Θ(n).

Lista contra arreglo

Operación	Lista simple con `tail_`	Arreglo dinámico
Llegar al elemento i	$\Theta(i)$	$O(1)$
Insertar al inicio	$O(1)$	$\Theta(n)$
Sacar del inicio	$O(1)$	$\Theta(n)$
Insertar al final	$O(1)$	$O(1)$ amortizado
Sacar del final	$\Theta(n)$	$O(1)$
Insertar en el medio, ya en ese punto	$O(1)$, con el nodo anterior	$\Theta(n)$ en el peor caso

Idea clave

Ninguna es mejor en general: cada una abarata lo que la otra encarece.

Error común: leer `next` después de borrar

errors/use_after_delete.cpp

```cpp
~BadList() {
    for (Node* node = head_; node != nullptr; node = node->next) {
        delete node;
    }
}
```

El `node = node->next` del `for` se ejecuta después del `delete` del cuerpo. Sin sanitizers puede pasar inadvertido: la memoria recién liberada a menudo conserva sus bytes.

ASan lo señala con precisión

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/use_after_delete.cpp -o build/use_after_delete
$ build/use_after_delete
READ of size 8 at 0x502000000038 thread T0
    #0 0xaaaaac491570 in BadList::~BadList() errors/use_after_delete.cpp:14
...
0x502000000038 is located 8 bytes inside of 16-byte region [0x502000000030,0x502000000040)
freed by thread T0 here:
...
    #1 0xaaaaac4914fc in BadList::~BadList() errors/use_after_delete.cpp:15
```

- Una lectura de 8 bytes, un puntero, en la línea 14.
- Dentro de un bloque de 16 bytes, un nodo, liberado en la línea 15.
- **“8 bytes inside”** es el campo `next`. La corrección: leerlo antes de borrar.

SECCIÓN 03

Lista doble con centinela

03

Un puntero más, y cuatro casos

`prev` permite sacar cualquier nodo sin recorrer nada. Pero con extremos nulos:

Situación de x	Qué hay que hacer
En el medio	`x->prev->next = x->next` y `x->next->prev = x->prev`
Es el primero	Además, actualizar `head`. No existe `x->prev`
Es el último	Además, actualizar `tail`. No existe `x->next`
Es el único	Actualizar los dos

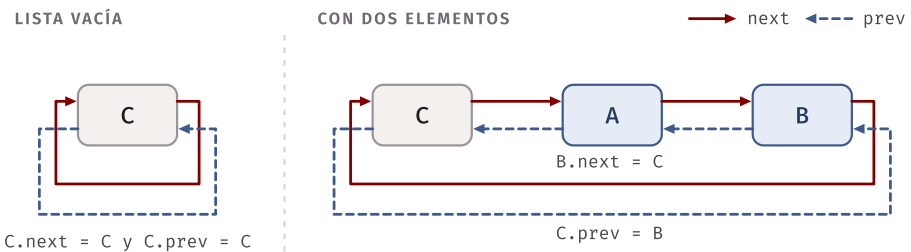
La observación

Todos estos casos existen porque un extremo de la lista apunta a “nada”. La solución: que no haya nada.

El nodo centinela

Nodo centinela *(sentinel node)*

Un nodo que siempre está en la lista y no guarda un elemento. Con él, los nodos forman un ciclo.



El invariante, en código

dlist.h

```
1  struct Node {
2      T value{};
3      Node* prev = nullptr;
4      Node* next = nullptr;
5  };
6
7  DList() : sentinel_(new Node) {
8      sentinel_->prev = sentinel_;
9      sentinel_->next = sentinel_;
10 }
```

Nodo	prev	next
C	B	A
A	C	B
B	A	C

Para todo nodo x

```
x->next->prev == x
x->prev->next == x
```

Insertar antes de un nodo

dlist.h

```
1  iterator insert(iterator pos, const T& value) {
2      Node* after = pos.node_;
3      Node* before = after->prev;
4      Node* node = new Node{value, before, after};
5      before->next = node;
6      after->prev = node;
7      size_++;
8      return iterator(node);
9  }
```

- El nodo nuevo ya sabe quiénes son sus vecinos. La lista no ha cambiado.
- `before->next = node`
- `after->prev = node`. Cambian dos punteros de la lista.

El mismo código, con la lista vacía

Con la lista vacía, `before` y `after` son los dos el centinela C:

Línea	Efecto
`new Node{value, before, after}`	`B.prev = C`, `B.next = C`
`before->next = node`	`C.next = B`
`after->prev = node`	`C.prev = B`

Sin casos

Queda un ciclo C $\leftrightarrow$ B correcto, sin un solo `if`. Insertar al inicio, al final, en el medio y en una lista vacía es la misma función.

Borrar: cambian exactamente dos punteros

dlist.h

```
1  iterator erase(iterator pos) {
2      assert(pos != end());
3      Node* node = pos.node_;
4      Node* after = node->next;
5      node->prev->next = after;
6      after->prev = node->prev;
7      delete node;
8      size_--;
9      return iterator(after);
10 }
```

Sacar A de la lista de dos elementos:

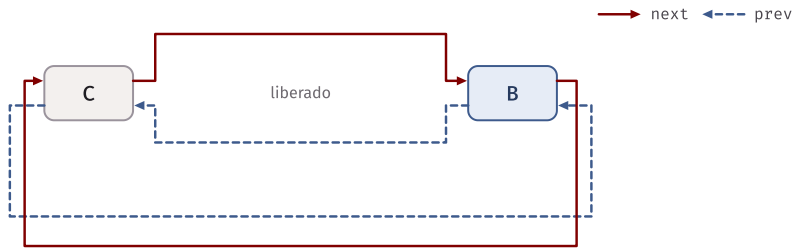
Nodo	`prev`	`next`
C	B	B
B	C	C

`C.next` y `B.prev`. El nodo A se libera y sus punteros dejan de importar.

Insertar y borrar, línea por línea

```
insert
Node* node = new Node{value, before, after};
before->next = node;
after->prev = node;
```

```
erase
node->prev->next = after;
after->prev = node->prev;
delete node;
```



delete node: se libera A. Para sacarlo cambiaron exactamente dos punteros

Los extremos salen en una línea

dlist.h

```
1  void push_front(const T& value) { insert(begin(), value); }
2  void push_back(const T& value) { insert(end(), value); }
3  void pop_front() { erase(begin()); }
4  void pop_back() { erase(iterator(sentinel_->prev)); }
```

terminal

```
$ build/dlist_demo
adelante: 10 20 40
atrás:    40 20 10
adelante: 10 20 30 40
atrás:    40 30 20 10
```

Idea clave

El centinela no agrega funcionalidad: elimina casos. Cada `if` que desaparece es un bug que ya no se puede escribir.

El owner de los nodos

dlist.h

```
1  ~DList() {
2      Node* node = sentinel_->next;
3      while (node != sentinel_) {
4          Node* next = node->next;  // se lee antes de
       borrar
5          delete node;
6          node = next;
7      }
8      delete sentinel_;
9  }
```

Los costos del centinela

- Una lista vacía ya reserva memoria.
- `T` necesita constructor por defecto: el centinela también tiene un `T value{}`.
- `std::list` no tiene esa restricción: en `g++`, sus enlaces van en un tipo aparte del valor.

Mover una lista con centinela

Después de un movimiento, el origen debe seguir siendo válido.

- Si el constructor de movimiento se lleva el centinela, el origen rompe su invariante: `sentinel_` nulo.
- En el laboratorio: el objeto nuevo crea **su propio** centinela e intercambia con el origen.
- Ese constructor reserva memoria, puede lanzar `std::bad_alloc`, y por lo tanto no es `noexcept`.

Recuerden `std::move_if_noexcept`

Un `std::vector<LinkedDeque<int>` **copia** todos los nodos de cada lista cada vez que crece. En `g++`, `std::list<int>` sí es `noexcept` al moverse: guarda su centinela dentro del objeto.

Un bug que recorrer no descubre

errors/forgot_prev.cpp

```cpp
void push_front(int value) {
    Node* first = sentinel_->next;
    Node* node = new Node{value, sentinel_, first};
    sentinel_->next = node;
    // falta: first->prev = node;
}
```

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/forgot_prev.cpp -o build/forgot_prev
$ build/forgot_prev
1 2
forgot_prev: errors/forgot_prev.cpp:48: void BrokenList::check_invariants() const: Assertion 'node->prev->next == node' failed.
```

Adelante se ve bien, y ASan no ve nada: no hay acceso inválido. Fallaría más tarde, lejos de la línea culpable.

SECCIÓN 04

Iteradores

04



¿Cómo se recorre una lista?

- `operator[]` y un índice: $\Theta(n^2)$.
- Dar un `Node*`: expone un detalle privado, y permite romper el ciclo desde afuera.

Iterador ***(iterator)***

Un objeto que señala una posición de un contenedor y sabe avanzar a la siguiente. La posición que sigue a la última es el **fin**, y no se puede leer.

En C, con un puntero	Con un iterador	Qué hace
`int *p = array`	`auto it = list.begin()`	Señala el primero
`array + n`	`list.end()`	Señala el fin
`*p`	`*it`	Da el elemento
`p++`	`++it`	Avanza
`p != array + n`	`it != list.end()`	¿Llegó al fin?

El iterador de la lista

dlist.h

```
1  template <typename V>
2  class Iter {
3  public:
4      V& operator*() const { return node_->value; }
5      Iter& operator++() {
6          node_ = node_->next;
7          return *this;
8      }
9      bool operator!=(const Iter& other) const {
         return node_ != other.node_; }
10 private:
11     friend class DList;
12     explicit Iter(Node* node) : node_(node) {}
13     Node* node_;
14 };
15 using const_iterator = Iter<const T>;
```

- Un wrapper de un `Node*`.
- `V` es `T` o `const T`: un template evita dos clases casi iguales.
- El constructor es privado: solo la lista fabrica iteradores.
- `end()` señala el centinela. Avanzar desde el último llega ahí solo.

Un `for` por rango es una abreviatura

trace_range.cpp

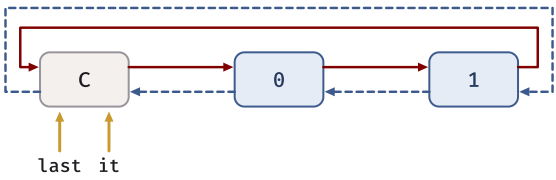
```cpp
TraceRange range(2);
auto it = range.begin();
auto last = range.end();
for (; it != last; ++it) {
    int value = *it;
    std::cout << "valor " << value
     << "\n";
}
```

terminal

```
$ build/trace_range
for por rango:
  begin
  end
  operator!=
  operator*
valor 0
  operator++
  operator!=
  operator*
valor 1
  operator++
  operator!=
```

- Sirve para cualquier clase con `begin` y `end`.
- `end()` se llama una sola vez.
- `int value` es una copia. Para modificar: `int&`.

El mismo ciclo, sobre una lista




```
auto it = range.begin();
auto last = range.end();
for (; it != last; ++it) {        !=
    int value = *it;
    // el cuerpo
}
```

```
SALIDA
  begin
  end
  operator!=
  operator*
valor 0
  operator++
  operator!=
  operator*
valor 1
  operator++
  operator!=
```

it != last es falso, y el ciclo termina: 3 != para 2 elementos

Borrar mientras se recorre

erase_evens.cpp

```cpp
auto it = list.begin();
while (it != list.end()) {
    if (*it % 2 == 0) {
        it = list.erase(it);
    } else {
        ++it;
    }
}
```

terminal

```
$ build/erase_evens
1 3 5 7
```

- Después de borrar, el iterador señala un nodo liberado.
- Por eso `erase` devuelve el siguiente.
- Un `for` por rango no sirve aquí: su `++` oculto siempre se ejecuta.

Error común: borrar dentro de un `for` por rango

errors/erase_in_range_for.cpp

```cpp
for (int value : list) {
    if (value == 1) {
        list.erase(it);  // el ++ oculto del for lee el next del nodo borrado
    }
}
```

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/erase_in_range_for.cpp -o build/erase_in_range_for
$ build/erase_in_range_for
READ of size 8 at 0x503000000080 thread T0
    #0 0xaaaaab114bc8 in DList<int>::Iter<int>::operator++() errors/../dlist.h:32
    #1 0xaaaaab113da4 in main errors/erase_in_range_for.cpp:10
```

La línea 10 es la del `for`. No hay ningún `++` escrito ahí, y ASan lo señala: es el ciclo oculto.

SECCIÓN 05

Invalidación

05

Cuándo se invalida un iterador

Una guía para arreglos y listas

Un iterador a un elemento se invalida cuando ese elemento deja de estar donde estaba.

Operación	**`DList`, `std::list`**	**`std::vector`**
Insertar	Ninguno se invalida	Si crece: todos. Si no crece: los de ese punto en adelante, y `end()`
Borrar	Solo los del elemento borrado	Los de ese punto en adelante, y `end()`

En una lista, los nodos no se mueven nunca. En un arreglo dinámico, crecer traslada todos los elementos a otro bloque.

Comprobado, sin cometer el error

`invalidation.cpp` guarda las direcciones como números: nunca usa un puntero inválido.

terminal

```
$ build/invalidation
lista: el elemento sigue en la misma dirección: sí
lista: *it = 1, *in_list = 1
vector sin reserve: el bloque cambió de lugar 10 veces
vector con reserve: siguió en la misma dirección: sí
```

Un elemento de la lista sobrevive a 2000 inserciones. El `std::vector` creció duplicando; las 10 veces dependen de esa política.

> **Idea clave**
>
> Si otra parte del programa guarda punteros a los elementos, una lista se los conserva. Es mejor razón para elegirla que “insertar es O(1)”.

Y si se usa el iterador viejo

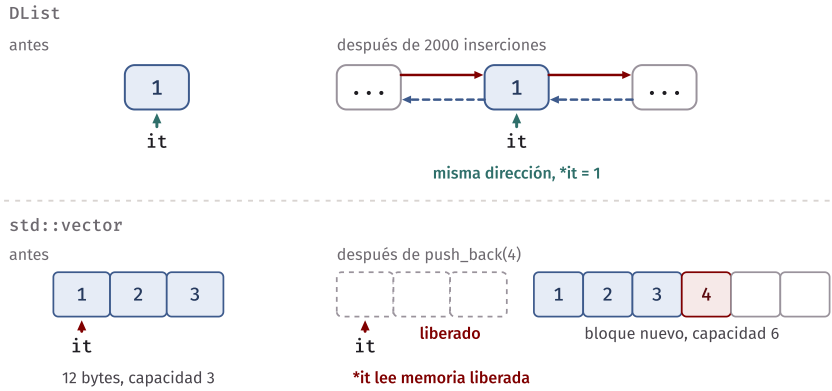
errors/stale_vector_iterator.cpp

```cpp
std::vector<int> values = {1, 2, 3};
auto it = values.begin();
values.push_back(4);  // la capacidad era 3: el bloque cambia de lugar
std::cout << *it << "\n";
```

terminal

```
$ g++ -Wall -Wextra -std=c++20 -g -fsanitize=address,undefined errors/stale_vector_iterator.cpp -o build/stale_vector_iterator
$ build/stale_vector_iterator
READ of size 4 at 0x502000000010 thread T0
    #0 0xaaaae57a9a58 in main errors/stale_vector_iterator.cpp:9
...
0x502000000010 is located 0 bytes inside of 12-byte region [0x502000000010,0x50200000001c)
```

Lista y vector, antes y después



Un iterador se invalida cuando su elemento deja de estar donde estaba

SECCIÓN 06

El tipo abstracto deque

06



Deque *(double-ended queue)*

Operación	Efecto
`push_front(x)`	x pasa a ser el primero
`push_back(x)`	x pasa a ser el último
`pop_front()`	Saca el primero *
`pop_back()`	Saca el último *
`front()`, `back()`	El primero y el último *
`size()`, `empty()`	Tamaño, y si está vacío

* Precondición: no está vacío.

Pila

Un solo extremo: `push_back` y `pop_back`.

Cola

Entra por un extremo y sale por el otro: `push_back` y `pop_front`.

Un tipo abstracto dice qué hacen las operaciones, y no cómo. Quien tiene un deque tiene las dos.

Una interfaz sin clase base

same_interface.cpp

```cpp
template <typename D>
void test_both_ends() {
    D deque;
    assert(deque.empty());
    deque.push_back(2);
    deque.push_back(3);
    deque.push_front(1);
    assert(deque.size() == 3);
    assert(deque.front() == 1);
    assert(deque.back() == 3);
...
```

same_interface.cpp

```cpp
test_both_ends<DList<int>>();
test_both_ends<std::list<int>>();
test_both_ends<std::deque<int>>();
```

terminal

```
$ build/same_interface
DList: ok
std::list: ok
std::deque: ok
```

Las pruebas son del tipo abstracto, no de la representación. El template compila con cualquier `D` que tenga esas operaciones. Así son las del laboratorio.

SECCIÓN 07

Dos representaciones

07

El deque sobre un buffer circular

pseudocódigo

```
algoritmo push_front(deque, x)
    si deque.tamaño = deque.capacidad entonces
        crecer(deque)
    fin si
    deque.frente ← (deque.frente - 1 + deque.
     capacidad)
                        mod deque.capacidad
    deque.datos[deque.frente] ← x
    deque.tamaño ← deque.tamaño + 1
fin algoritmo
```

- El elemento i está en la casilla (frente + i) mod capacidad.
- Sin el `+ capacidad`, con frente 0 la resta da -1.
- En C++, con `std::size_t` da la vuelta, y solo sale bien si la capacidad es potencia de 2; con `int`, `-1 % 4` da `-1`.

Sigamos un buffer de capacidad 4

Operación	Casillas	front	Tamaño	Capacidad
`push_back(a)`	`a . . .`	0	1	4
`push_back(b)`	`a b . .`	0	2	4
`push_back(c)`	`a b c .`	0	3	4
`pop_front()`	`. b c .`	1	2	4
`push_back(d)`	`. b c d`	1	3	4
`push_back(e)`	`e b c d`	1	4	4
`push_front(z)`, crece	`b c d e . . . .`	0	4	8
`push_front(z)`, inserta	`b c d e . . . z`	7	5	8

`e` da la vuelta: (1 + 3) mod 4 = 0. Es el último, y está en la primera casilla. Crecer **desenrolla**: se copia en orden lógico, del elemento 0 al último, y el frente vuelve a 0. Copiar las casillas tal cual dejaría a `e` primero. `z` retrocede hasta el final: (0 − 1 + 8) mod 8 = 7. El elemento 1 es `b`, en la casilla (7 + 1) mod 8 = 0.

Dos decisiones con `T` general

`new T[capacidad]`

Las casillas libres también contienen un `T`, y `T` necesita constructor por defecto. `std::vector` y `std::deque` reservan sin construir, y construyen a mano. Aquí no lo hacemos.

`pop` no destruye

Si no se hace algo más, un `std::string` largo o un `std::shared_ptr` sigue retenido por un elemento que ya no existe. Por eso la casilla libre se reemplaza por un `T{}`.

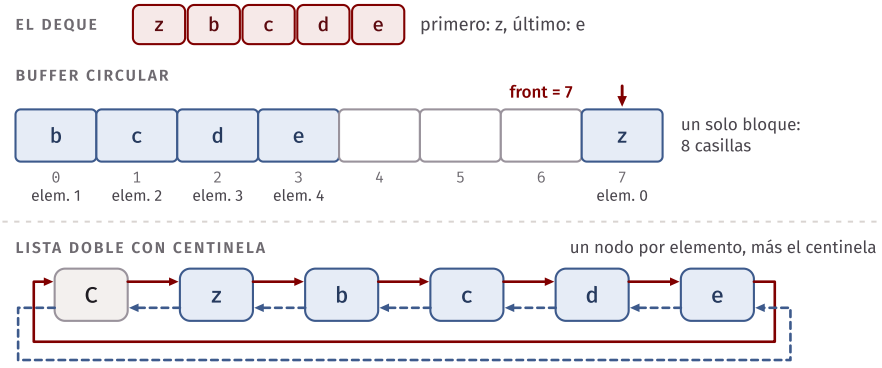
Una trampa de `push_back(const T&)`

Error común

`deque.push_back(deque.front())` con el buffer lleno: `grow` libera el bloque donde vive el argumento, y luego se lee una referencia colgante.

- Hay que sacar la copia **antes** de crecer.
- La lista doble no tiene este problema: crea el nodo sin tocar los demás.
- Las pruebas del laboratorio lo revisan, con ASan.

El mismo deque, dos representaciones



Los mismos elementos en el mismo orden: el orden lo da la fórmula en uno, y los punteros en el otro

Lado a lado

Operación	Buffer circular	Lista doble con centinela
`push_front`, `push_back`	O(1) amortizado, Θ(n) la que crece	O(1)
`pop_front`, `pop_back`	O(1)	O(1)
`front`, `back`	O(1)	O(1)
Elemento i	O(1)	Θ(min(i, n − i))
Recorrer todo	Θ(n)	Θ(n)
Insertar en el medio, con un iterador	Θ(n)	O(1)
Los elementos no se mueven	No: crecer los traslada	Sí, hasta que se borren

Si solo miramos O, las operaciones del deque se ven iguales. Vamos a medir.

SECCIÓN 08

Lo que O no ve

08

Cómo se midió

- Entorno del curso: contenedor Linux en una laptop con procesador Apple (aarch64), con otros procesos corriendo a la vez.
- `g++ -O2`, sin sanitizers.
- Cada tiempo es la mediana de varias repeticiones, con `bench.h`, el medidor de la semana 3.
- Cada programa se corrió **tres veces** completas.

Los tiempos dependen de la máquina

En la suya serán otros. Lo que se puede defender son las razones entre columnas que se repiten en las tres corridas.

Reservas de memoria: un conteo

terminal

```
$ build/alloc_count
reservas para 1000000 push_back
std::vector: 21
std::deque:  7825
std::list:   1000000
DList:       1000001
```

- Un conteo no depende del reloj.
- `std::vector` duplica. `DList` reserva por elemento, más el centinela.
- `std::deque` usa bloques de tamaño fijo en `g++`.
- Son conteos de esta biblioteca. El estándar no los fija.

¿Se nota en el tiempo?

n `push_back` y luego n `pop_back`. Nanosegundos por operación:

terminal

```
$ build/ends_bench
       n std::vector    DList  razón
   10000        0.44     6.65   15.3
  100000        0.43     6.95   16.2
 1000000        0.41     7.55   18.2
```

- Tres corridas: razones entre 15.2 y 15.4, entre 16.0 y 16.2, y entre 16.8 y 18.2.
- Las dos columnas casi no cambian con n: O(1). La diferencia es un factor constante.
- No separa las causas. El conteo de reservas es consistente con el resultado, pero no lo explica por completo.

Una hipótesis: la cache

Cache

Memoria pequeña y rápida donde la CPU guarda lo último que leyó. Trae una **línea** completa de bytes vecinos.

- **Localidad espacial:** leyeron una dirección, probablemente lean sus vecinas.
- **Localidad temporal:** volverán a leer lo que leyeron.
- **Prebúsqueda:** un paso regular se detecta, y se trae lo siguiente por adelantado.

La predicción

Si la cache es la causa, el efecto debe depender de **cuánta memoria ocupa la estructura**. Con pocos elementos, todo cabe, y el orden no debería importar.

Aislar la localidad

dlist.h

```
1  void shuffle_nodes(unsigned seed) {
2      std::vector<Node*> nodes;
3      ...
4      std::mt19937 rng(seed);
5      std::shuffle(nodes.begin(), nodes.end(),
         rng);
6      Node* last = sentinel_;
7      for (Node* node : nodes) {
8          last->next = node;
9          node->prev = last;
10         last = node;
11     }
12     last->next = sentinel_;
13     sentinel_->prev = last;
14 }
```

- Mismo código de recorrido y misma cantidad de nodos.
- Lo que devuelve `new` no está en nuestras manos, pero podemos enlazar los mismos nodos en otro orden.
- Es el orden que tendría una lista cuyos nodos se hubieran reservado al azar.

Sumar n enteros, en ns por elemento

terminal

```
$ build/traversal_bench
       n  arreglo arr.baraj    lista lis.baraj  lis/arr  lb/lista     cont   cont.b
    1024     0.37      0.37     1.11      1.11      3.0       1.0   100.0%     0.2% ok
    4096     0.36      0.36     1.20      1.16      3.4       1.0    99.7%     0.1% ok
   16384     0.35      0.40     1.58      8.80      4.5       5.6    99.9%     0.0% ok
   65536     0.36      0.71     1.73     12.78      4.7       7.4   100.0%     0.0% ok
  262144     0.36      0.96     1.69     17.97      4.7      10.6   100.0%     0.0% ok
 1048576     0.36      1.05     1.67     96.32      4.6      57.7   100.0%     0.0% ok
 4194304     0.36      1.04     1.69    158.50      4.7      93.8   100.0%     0.0% ok
```

- `arr.baraj`: el arreglo, visitado según una permutación al azar de los índices.
- `lis.baraj`: la misma lista, después de `shuffle_nodes`.
- `cont` y `cont.b`: % de saltos de la lista que avanzan exactamente 32 bytes. Es un conteo que comprueba dónde están los nodos.
- `ok`: la suma coincide en las cuatro formas.

Las razones, en las tres corridas

n	Memoria de la lista	lis/arr	lb/lista
1 024	32 KB	3.0, 3.0, 3.0	1.0, 1.0, 1.0
4 096	128 KB	3.4, 3.3, 3.3	1.0, 1.0, 1.0
16 384	512 KB	4.5, 4.5, 4.4	5.6, 5.5, 5.7
65 536	2 MB	4.7, 4.5, 4.6	7.4, 7.7, 7.8
262 144	8 MB	4.7, 4.6, 4.6	10.6, 10.4, 9.6
1 048 576	32 MB	4.6, 4.6, 4.7	57.7, 56.6, 54.4
4 194 304	128 MB	4.7, 4.7, 4.7	93.8, 89.0, 91.4

Con pocos elementos, barajar no cambia nada, aunque los nodos ya están desordenados. Saltar sin orden no es lento por sí mismo. Saltos claros: entre 128 KB y 512 KB, y entre 8 MB y 32 MB. El macOS de la laptop reporta una cache de nivel 1 de 128 KB y una de nivel 2 de 16 MB en sus núcleos de rendimiento.

Qué podemos afirmar, y qué no

Se repite en las tres corridas

- Todos son $\Theta(n)$ y ninguno es igual a otro.
- Con pocos elementos, barajar no cambia nada.
- Con mucha memoria, con el mismo código, multiplica el tiempo por decenas.

Es lo que predijo la hipótesis de la cache.

No lo sabemos

- **No medimos fallos de cache.** Medimos tiempo. Este entorno no trae `perf`.
- Puede no ser la única causa: el TLB también traduce direcciones.
- Los tamaños son orientativos: el contenedor dice líneas de 64 bytes y el macOS, de 128, y no sabemos en qué núcleo corrió.
- En otra máquina, los saltos estarán en otro lugar. Debería repetirse la forma.

Las otras dos columnas

- **`lis/arr` es 3.0 con n = 1 024,** cuando todo debería caber en la cache. Y el compilador no vectorizó el ciclo del arreglo:

```
terminal

$ g++ -Wall -Wextra -std=c++20 -O2 -fopt-info-vec-missed traversal_bench.cpp -o /dev/null 2>&1 | grep 'cpp:69:'
traversal_bench.cpp:69:26: missed: couldn't vectorize loop
traversal_bench.cpp:69:26: missed: Loop costings not worthwhile.
```

- Lo probable: en la lista, cada paso depende de leer el puntero anterior. `arr.baraj` también hace dos lecturas y va igual que el arreglo. No lo medimos por separado.
- **`arr.baraj` sube de 0.37 a cerca de 1.05 ns sin un solo puntero:** lo que cuesta es el desorden en el acceso, no los punteros como tales.
- Sube menos que la lista barajada: ocupa la octava parte, y sabe el índice siguiente sin leer el elemento. No separamos las dos causas.

Entonces, ¿para qué sirve una lista?

- **Referencias estables:** otros guardan punteros a los elementos.
- **Insertar o borrar en un punto que ya tienen,** sin trasladar los demás.
- **Costo sin picos:** una inserción en un arreglo dinámico puede costar $\Theta(n)$.

> **Idea clave**
>
> Dos recorridos $\Theta(n)$ pueden diferir por un factor de cientos, con el mismo código, según dónde estén los datos. O dice cómo crece el costo. La localidad ayuda a explicar cuánto vale cada paso.

Si solo necesitan una secuencia que se recorre y crece en un extremo, empiecen por un arreglo, y midan.

SECCIÓN 09

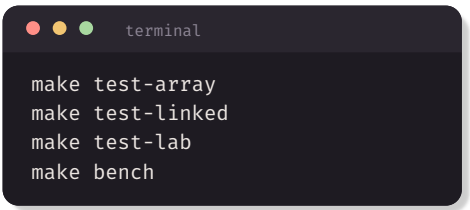
Laboratorio

09

Un deque con dos representaciones

Sesión 1: `ArrayDeque<T>`.
Sesión 2: `LinkedDeque<T>`, y medir.

- Las mismas pruebas, como templates. Cuatro más de cada lado.
- Con ASan y UBSan, y operaciones al azar con semilla fija contra `std::deque`.
- El origen de un movimiento queda vacío y se puede usar: contrato nuestro, no de la biblioteca. Cuidado con `% 0`.
- `lab/conclusion.md`: ¿por qué los tiempos no son iguales? Qué midieron y qué supusieron.



Para la próxima clase

- Leer los apuntes de la semana 6: mergesort, quicksort y búsqueda binaria.
- Laboratorio: un deque con dos representaciones.
- Dibujen la lista doble antes de escribir `LinkedDeque`.

> **Pregunta**
>
> Si barajar los nodos no cambió nada con 1 024 elementos, ¿cuándo esperan que empiece a notarse en su laptop, y cómo lo comprobarían?