

Structures Linéaires





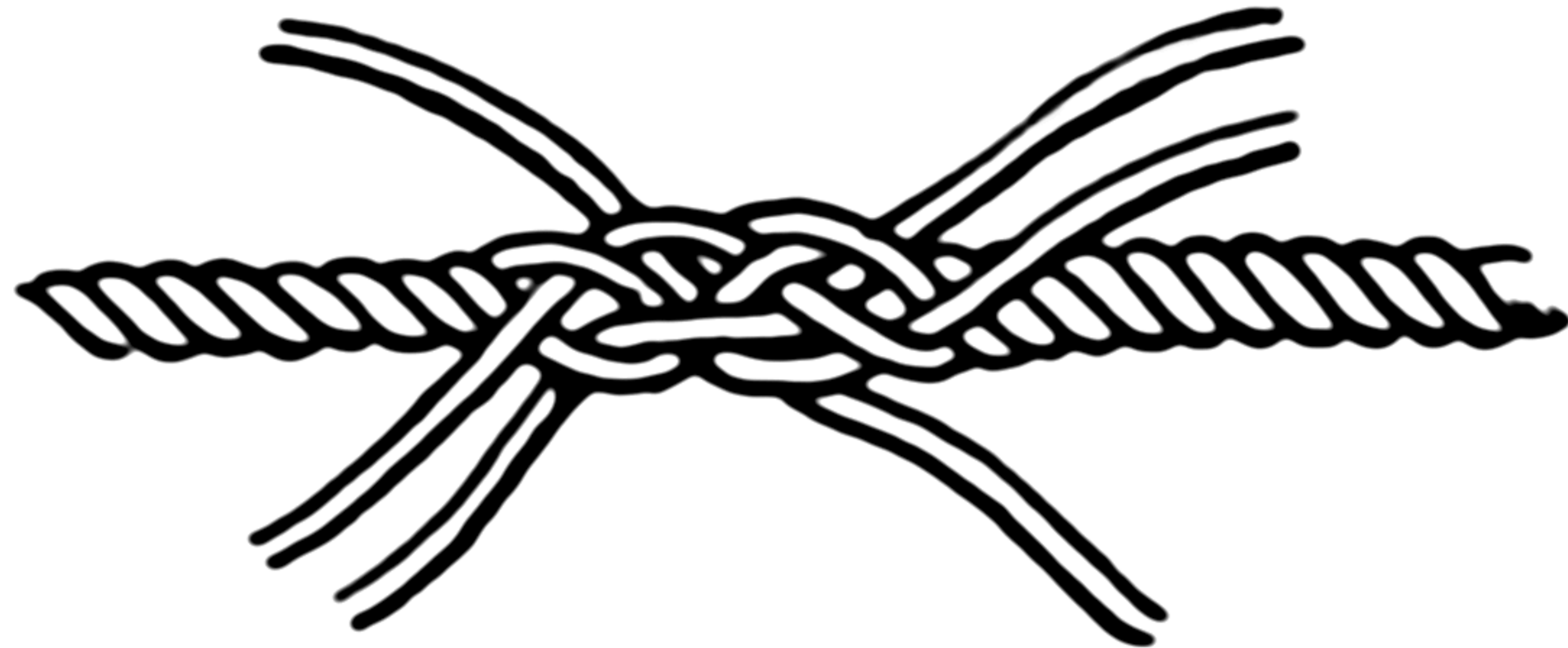
11. Splice

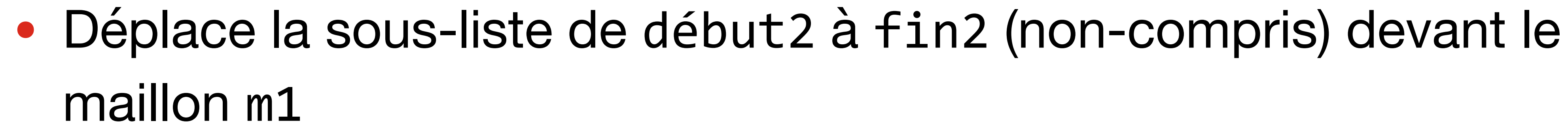
Splice



- En français, épissure, un mot du vocabulaire marin provenant du néerlandais splitsen

assembler deux cordes ou deux câbles mis bout à bout par l'entrelacement de leurs torons.

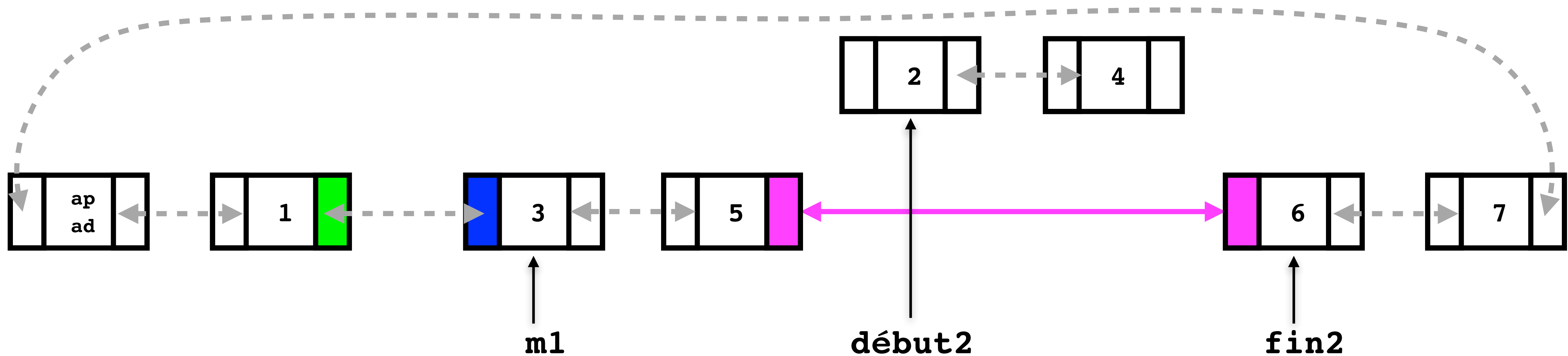




Splice (2)



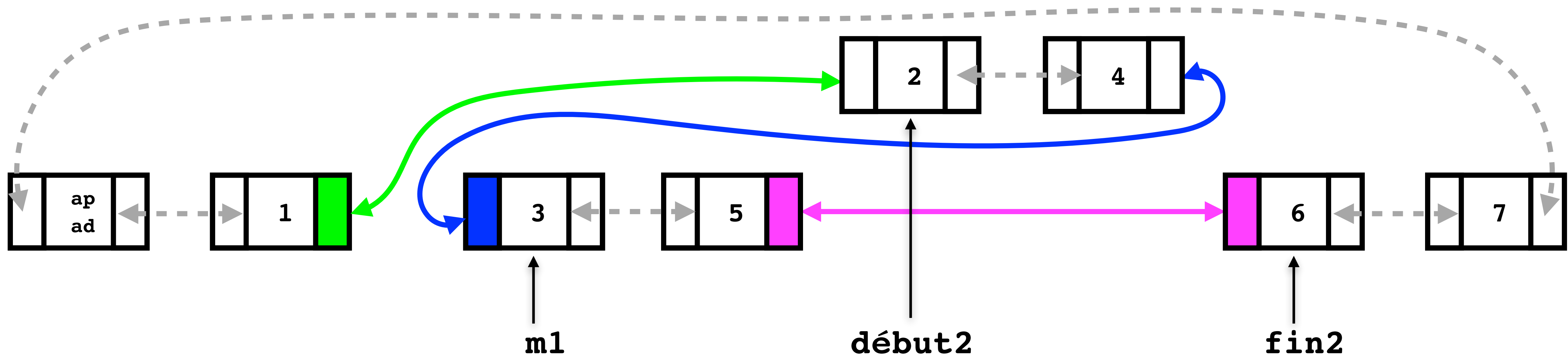
- Déplace la sous-liste de début2 à fin2 (non-compris) devant le maillon m1



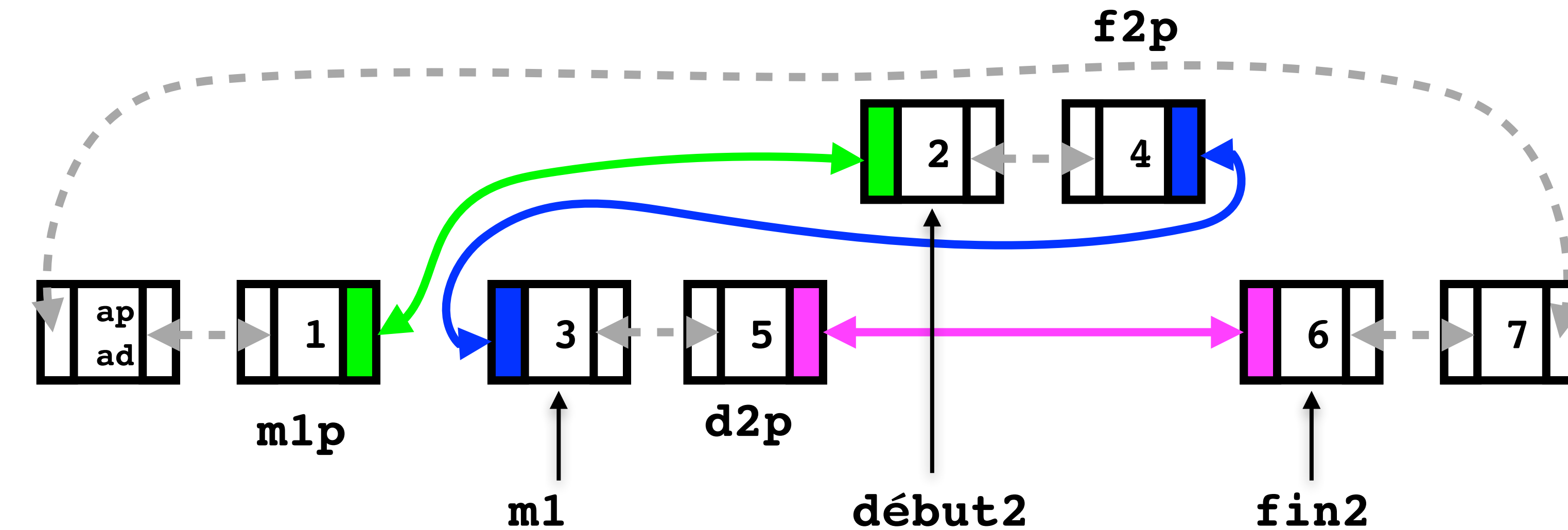
Splice (2)



- Déplace la sous-liste de début2 à fin2 (non-compris) devant le maillon m1



Splice (3)



```

fonction
splice_m(m1,debut2,fin2)

```

```

    si debut2 == fin2

```

```

        sortir

```

```

    m1p ← m1.precedent

```

```

    d2p ← debut2.precedent

```

```

    f2p ← fin2.precedent

```

```

    d2p.suivant ← fin2

```

```

    fin2.precedent ← d2p

```

```

    m1p.suivant ← debut2

```

```

    debut2.precedent ← m1p

```

```

    f2p.suivant ← m1

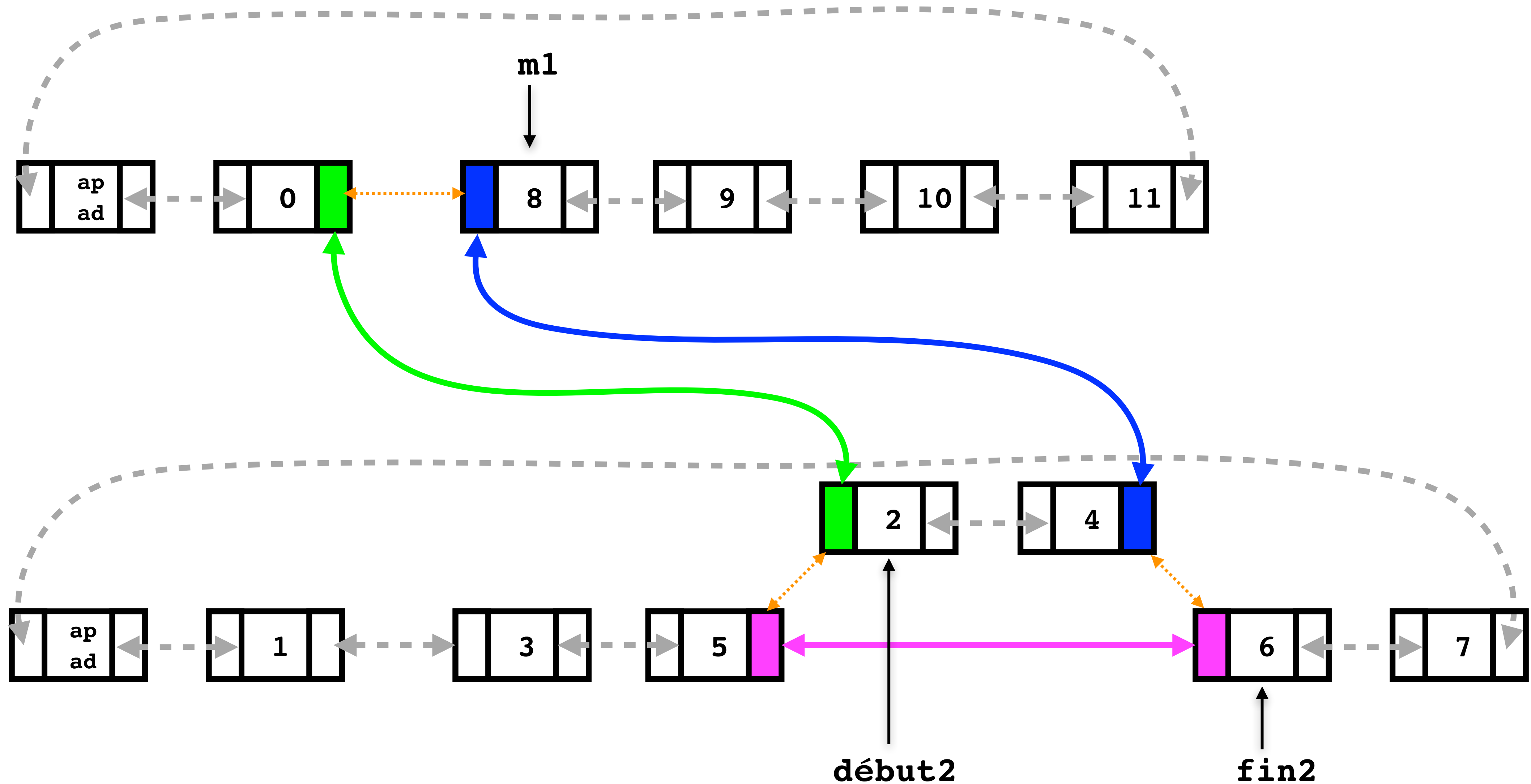
```

```

    m1.precedent ← f2p

```

Splice d'une liste à une autre



Fusion de 2 listes triées par splice



L1: 1 → 2 → 5 → 7 → 8 → 9

L:

L2: 0 → 3 → 4 → 6 → 10

Fusion de 2 listes triées par splice



L1: 1 → 2 → 5 → 7 → 8 → 9

L: 0

L2: 3 → 4 → 6 → 10

Fusion de 2 listes triées par splice



L1: 5 → 7 → 8 → 9

L: 0 → 1 → 2

L2: 3 → 4 → 6 → 10

Fusion de 2 listes triées par splice



L1: 5 → 7 → 8 → 9

L: 0 → 1 → 2 → 3 → 4

L2: 6 → 10

Fusion de 2 listes triées par splice



L1: 7 → 8 → 9

L: 0 → 1 → 2 → 3 → 4 → 5

L2: 6 → 10

Fusion de 2 listes triées par splice



L1: 7 → 8 → 9

L: 0 → 1 → 2 → 3 → 4 → 5 → 6

L2: 10

Fusion de 2 listes triées par splice



L1 :

L : 0 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

L2 : 10

Fusion de 2 listes triées par splice



L1 :

L : 0 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10

L2 :

Tri fusion en place



- La fonction `splice` permet de mettre en oeuvre le tri fusion en place
- Tous les splices peuvent être effectués au sein de la même liste
- Aucun déplacement nécessaire pour une liste déjà triée
- Appels récursifs sans sur-coût si on remplace l'interface `[first, last)` par `{first, N}`.

Et pour les listes simplement chainées ...

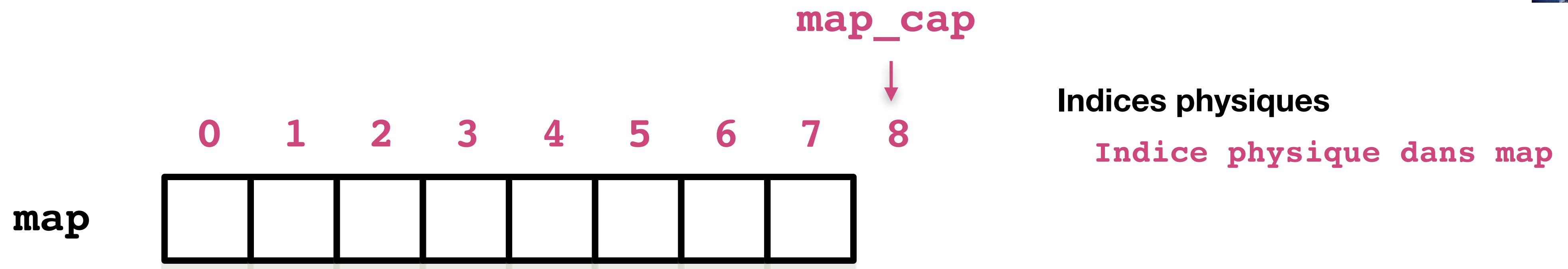


- Fonction `splice_after` prend les maillons précédents en paramètres
- 3 pointeurs et une variable tampon pour 4 affectations
- Tri fusion disponible

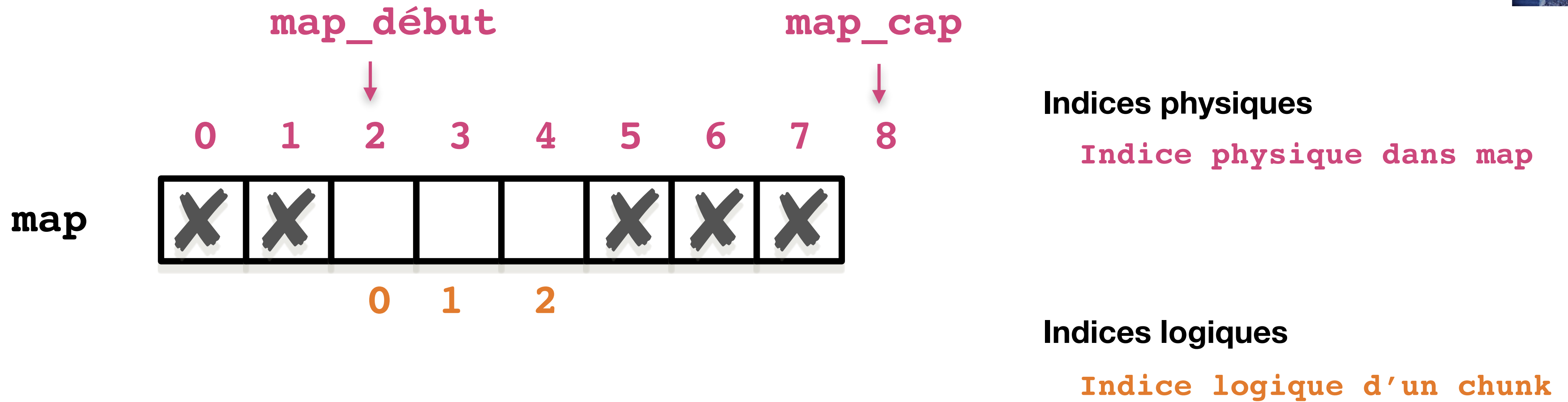


12. Double ended queue

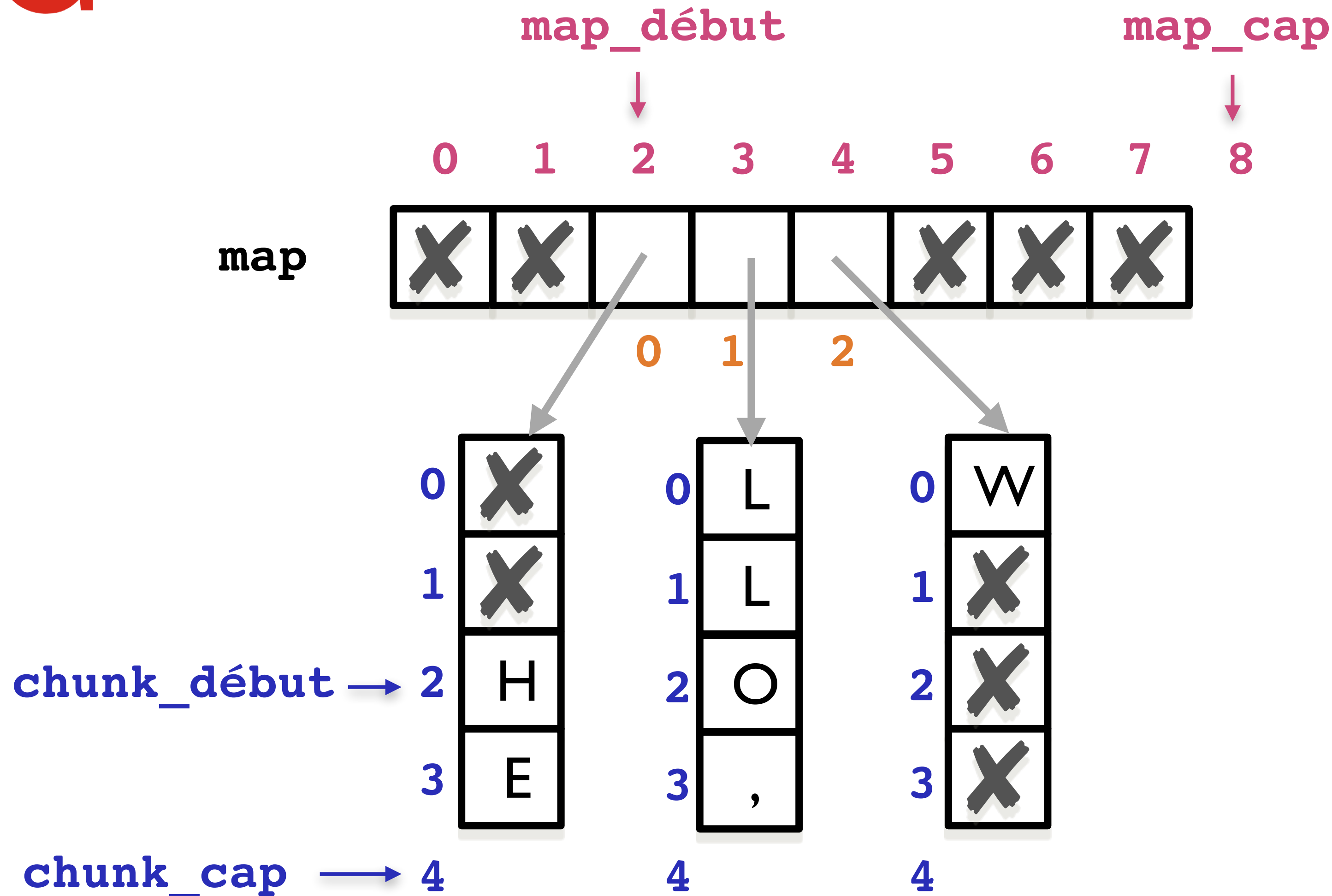
Double ended queue



Double ended queue



Double ended queue



Indices physiques

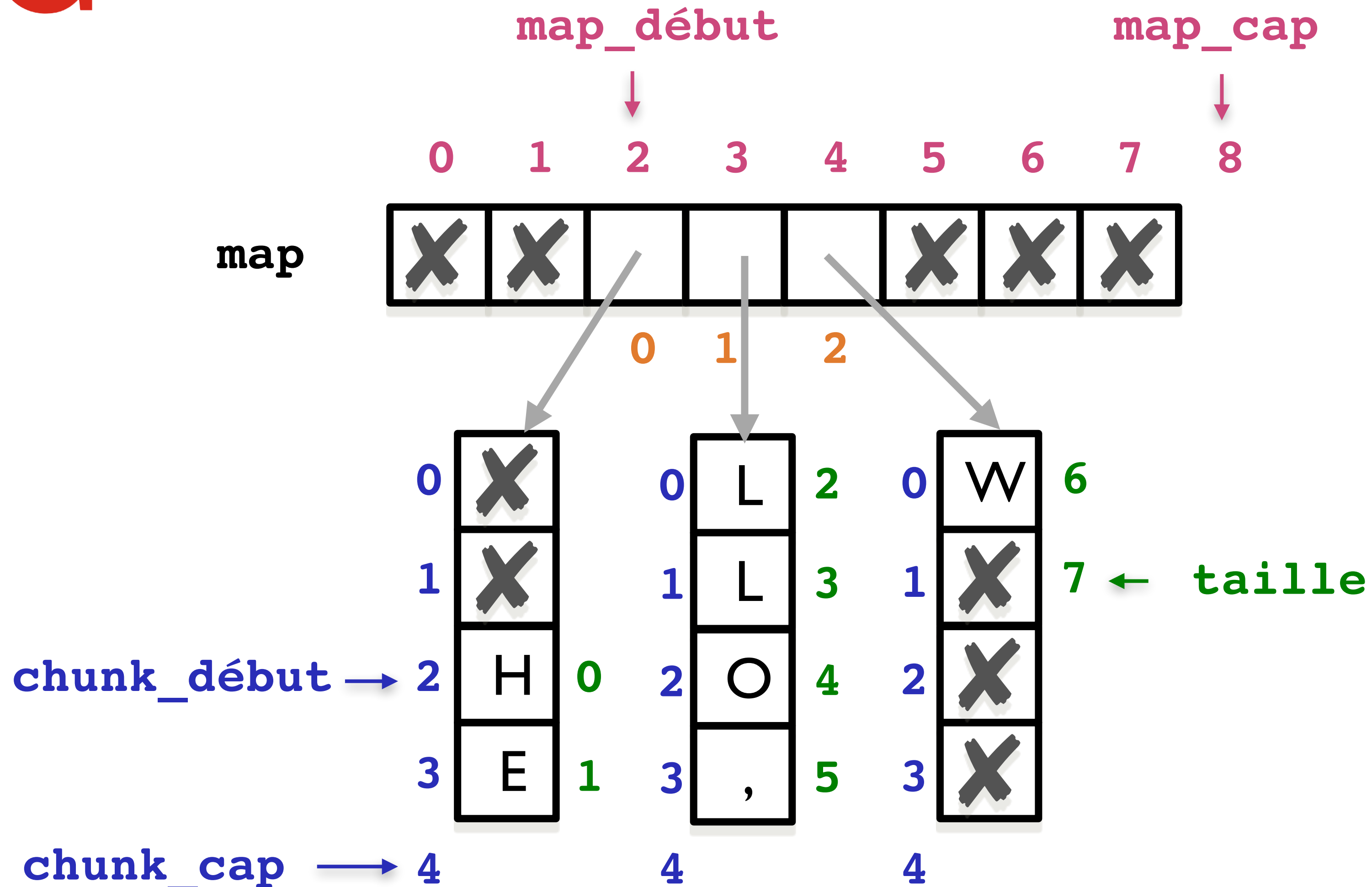
Indice physique dans map

Indice physique d'un élément
au sein d'un chunk

Indices logiques

Indice logique d'un chunk

Double ended queue



Indices physiques

Indice physique dans map

Indice physique d'un élément
au sein d'un chunk

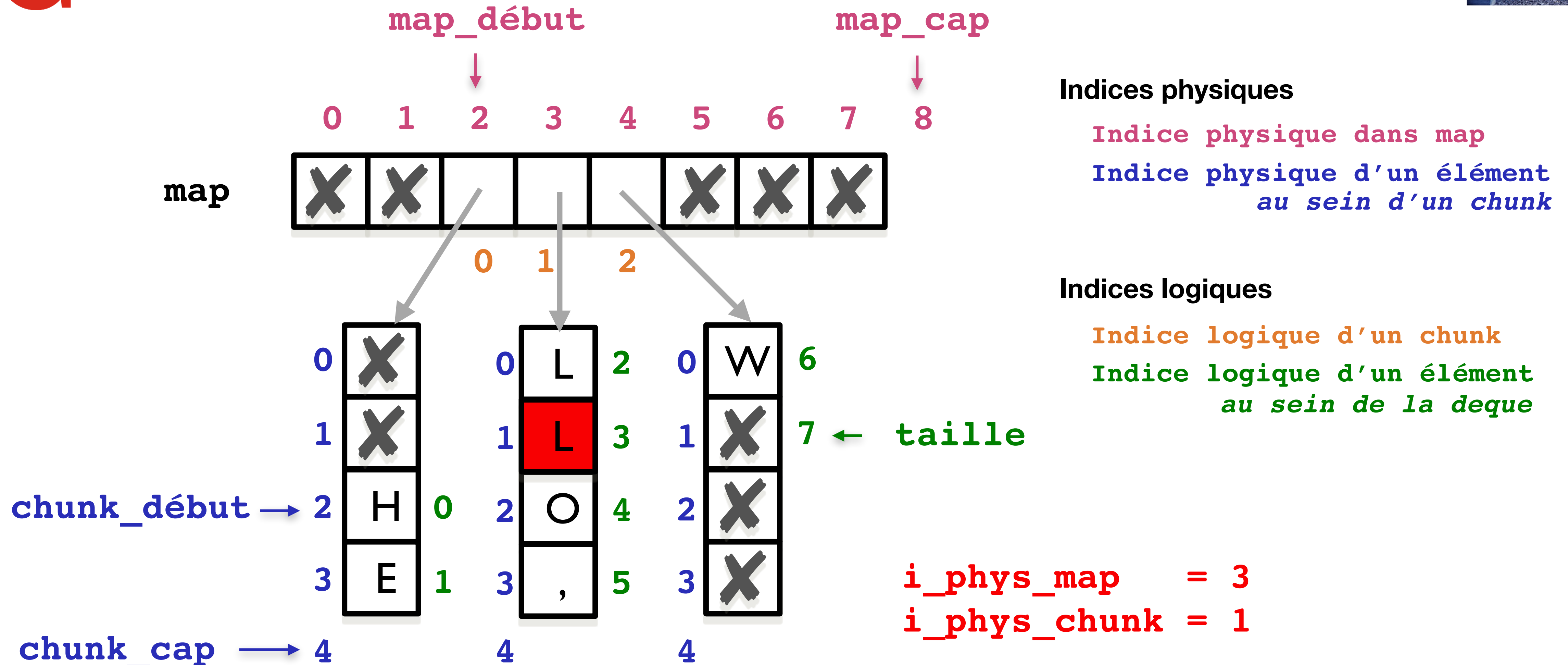
Indices logiques

Indice logique d'un chunk

Indice logique d'un élément
au sein de la deque



Double ended queue



Mise en oeuvre en C++ par `std::deque<T>`

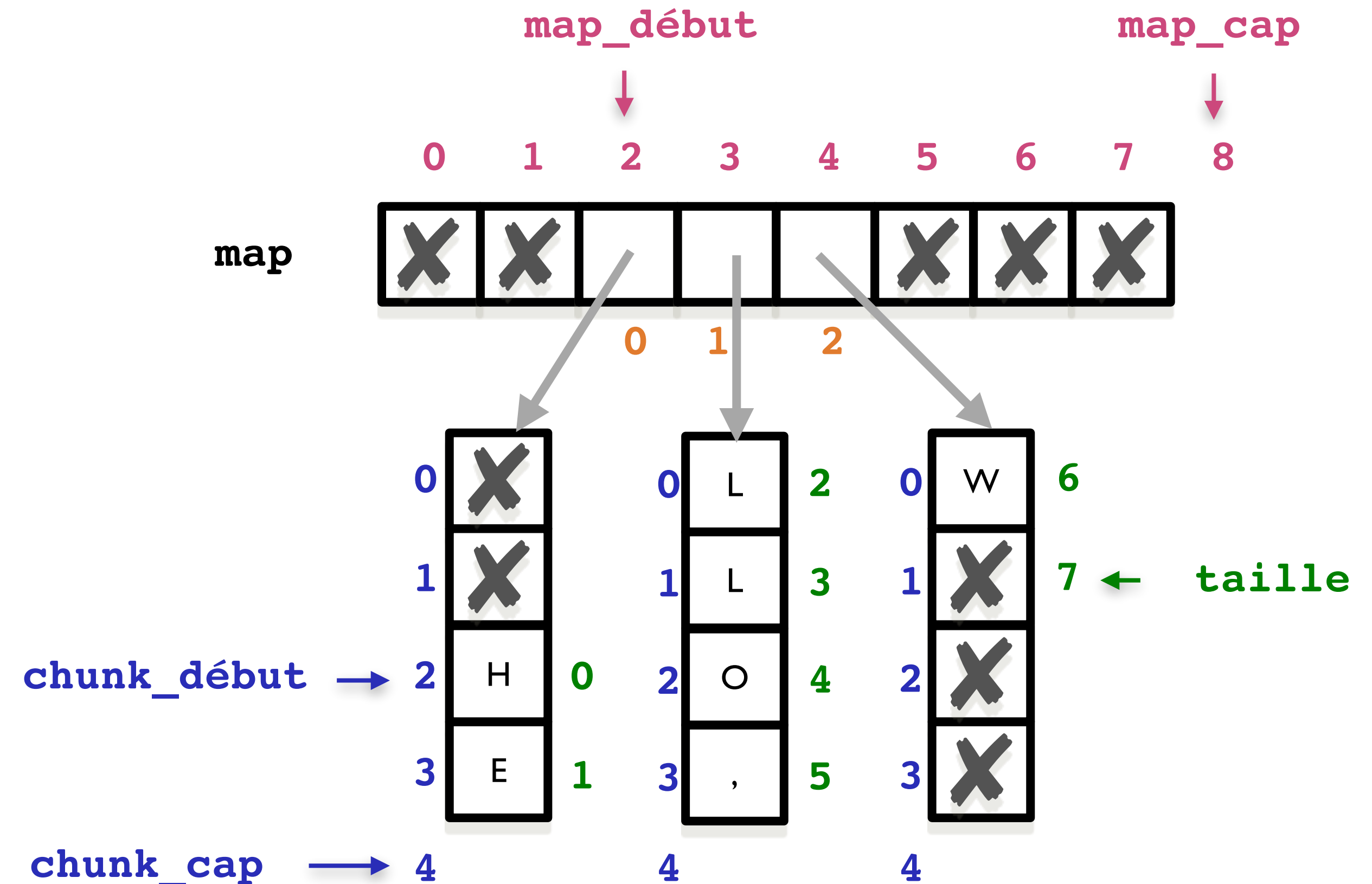


Accès indexé



```
méthode i_chunk(i)
    retourner ( i + chunk_debut ) % chunk_cap
```

```
méthode i_map(i)
    j ← ( i + chunk_debut ) / chunk_cap
    retourner ( j + map_debut ) % map_cap
```



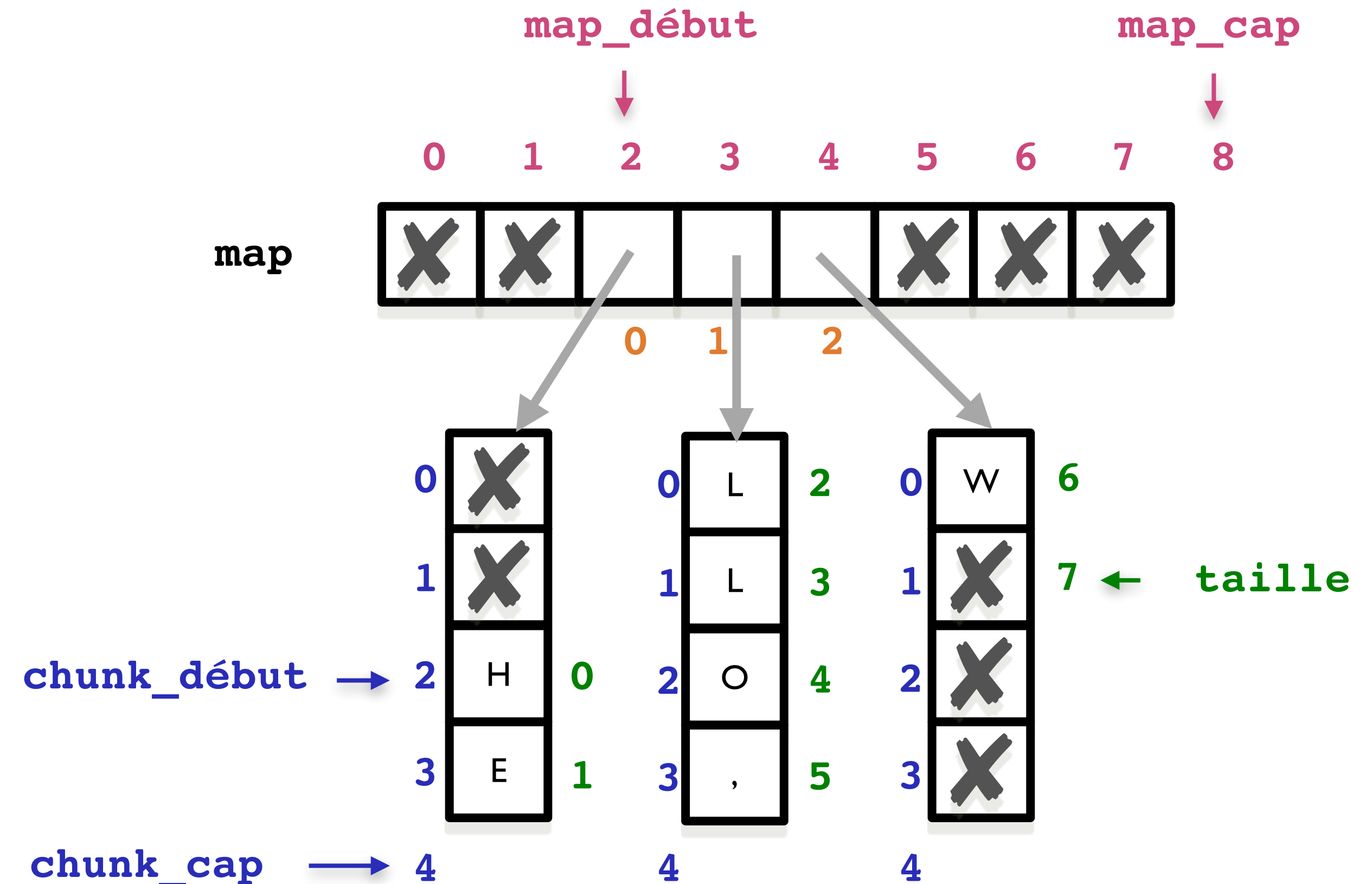
Accès indexé



```
méthode i_chunk(i)
    retourner ( i + chunk_debut ) % chunk_cap
```

```
méthode i_map(i)
    j ← ( i + chunk_debut ) / chunk_cap
    retourner ( j + map_debut ) % map_cap
```

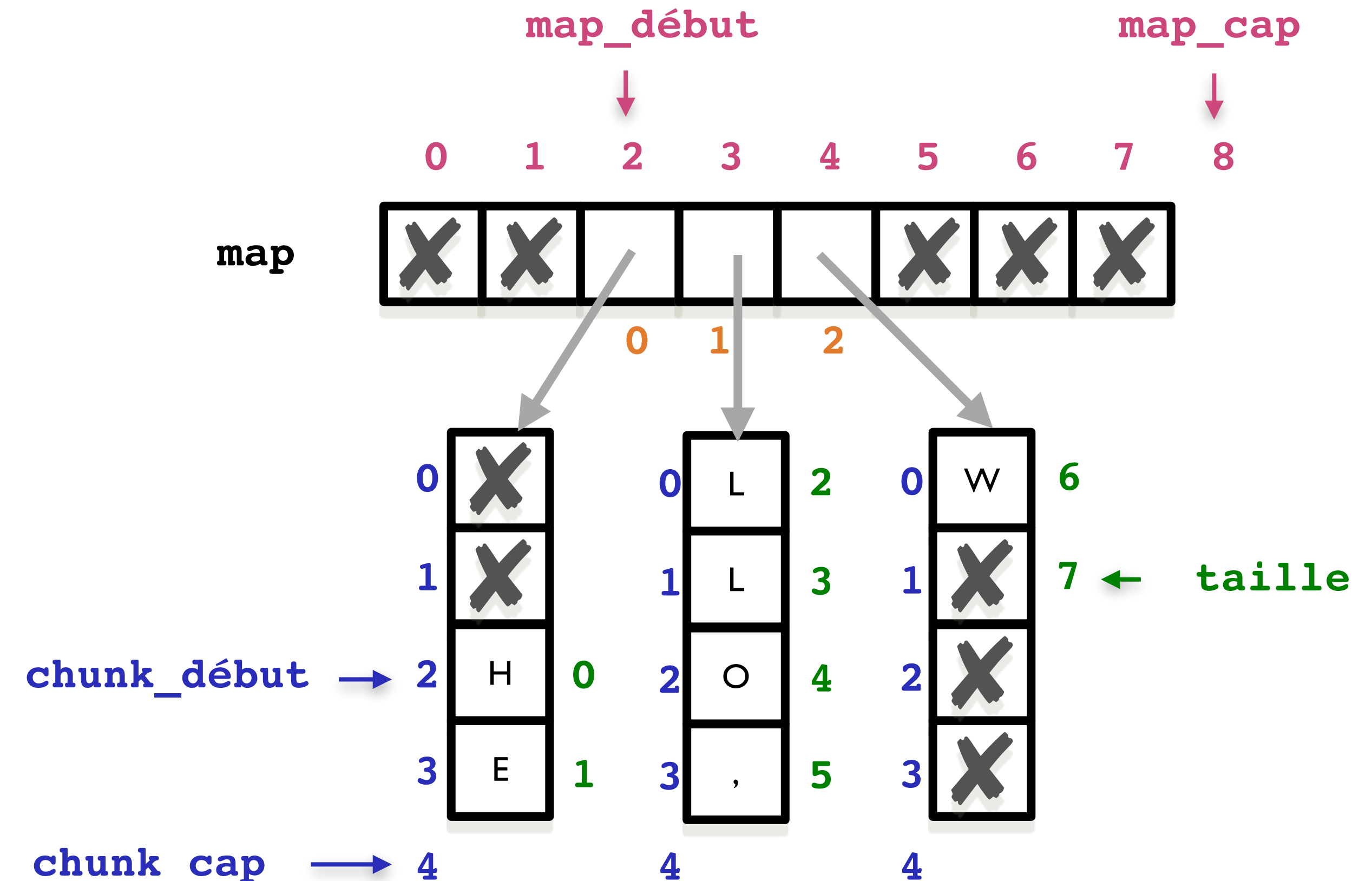
```
méthode at(i)
    retourner map[ i_map(i) ][ i_chunk(i) ]
```



Insérer au début



- Calculer les indices i_chunk et i_map de la position d'insertion -1
- Si nécessaire, allouer un nouveau chunk
- Si nécessaire, augmenter la capacité de map
- Ecrire dans $map[i_map(i)]$ $[i_chunk(i)]$
- Mettre à jour $chunk_debut$, map_debut et $taille$





- Accès indexé : $O(1)$
- Insertion / suppression en début ou fin (amorti) : $O(1)$
- Pire cas d'insertion en début/fin avec n éléments et chunks de capacité c
 - allouer un nouveau chunk en $O(1)$ (ou $O(c)$ s'il faut les initialiser)
 - ré-allouer *et recopier* la map en $O(n/c)$
 - Complexité totale en $O(n/c)$
- Insertion / suppression à l'indice i : amorti $O(i)$ ou $O(n - i)$
pire cas $O(i + n/c)$ ou $O(n - i + n/c)$



13. STL sequence containers

STL sequence containers



```
#include <array>
#include <vector>
#include <forward_list>
#include <list>
#include <deque>

using namespace std;

int main() {
    array<int,10> T;
    vector<double> V;
    forward_list<float> L1;
    list<long> L2;
    deque<char> D;
}
```

Classe	Description
<code>std::array<T,N></code>	tableau de taille fixe de N éléments de type T
<code>std::vector<T></code>	tableau de taille et capacité variables
<code>std::forward_list<T></code>	liste simplement chaînée
<code>std::list<T></code>	liste doublement chaînée
<code>std::deque<T,N></code>	double-ended queue, mise en oeuvre avec un tableau redimensionnable de tableaux de capacité fixe



`std::array<T, N>`

a.k.a. tableau à taille fixe

std::array<T,N>



- Type et taille spécifiés en paramètres génériques
- N doit être connu à la compilation
- Mémoire allouée automatiquement (stack memory)

```
#include <iostream>
#include <array>
using namespace std;

int main() {
    array<int,100> T;
    cout << sizeof(T);    // affiche 400

    for(size_t i = 0; i < T.size(); ++i)
        T.at(i) = rand();

    for(auto it = T.begin();
        it != T.end(); ++it)
        *it = *it % 100 ;
}
```

`std::array<T, N>`



Méthode(s)	Description
<code>size()</code> <code>max_size()</code>	retournent N
<code>empty()</code>	retourne N == 0
<code>operator[i]</code>	référence à l'élément d'indice i
<code>at(i)</code>	idem, mais lève une exception si i n'est pas un indice valable
<code>front()</code>	référence au premier élément: [0]
<code>back()</code>	référence au dernier élément: [N-1]
<code>==</code> <code>!=</code>	opérateurs d'(in)égalité
<code><</code> , <code><=</code> <code>></code> , <code>>=</code>	opérateurs de comparaison lexicographique

`std::array<T, N>` iterators



- Stockent l'adresse de l'élément courant
- Permettent de parcourir l'array de 4 manières

<code>begin() → end()</code>	itère par indice croissant
<code>cbegin() → cend()</code>	idem, mais seulement en lecture
<code>rbegin() → rend()</code>	itère par indice décroissant
<code>crbegin() → crend()</code>	idem, mais seulement en lecture

- Sont des « Random Access Iterators »

Random access iterators



Expression	Propriété
<code>X a; X b(a); b = a;</code>	Constructible par défaut et par copie, assignable et destructible
<code>a == b; a != b;</code>	Comparable (égaux si itèrent sur le même élément)
<code>*a; a->m;</code>	Déréférençable à droite
<code>*a = t;</code>	Déréférençable à gauche (mutable)
<code>++a; a++; *a++;</code>	Incrémentable: passe à l'élément suivant
<code>--a; a--; *a--;</code>	Décrémentable: passe à l'élément précédent
<code>a < b; a <= b; a > b; a >= b;</code>	Comparable: position relative dans la séquence
<code>a + n; n + a; a - n; a - b;</code>	Décalable: passe n éléments plus loin
<code>a += n; a -= n;</code>	Décalable et assignable
<code>a[n];</code>	Décalable et déréférençable



`std::vector<T>`

a.k.a. tableau à capacité variable

std::vector<T>



- Tableau de taille et capacité variable
- Type spécifié en paramètre générique
- Mémoire allouée dynamiquement (heap memory)

```
#include <iostream>
#include <array>
using namespace std;

int main() {
    vector<int> T(1000);
    cout << sizeof(T); // affiche (e.g.) 24
}
```

`std::vector<T>`



Méthode	Description
<code>push_back(t);</code> <code>pop_back();</code>	insère ou supprime un élément en fin
<code>insert(pos,...);</code>	insère un ou plusieurs éléments devant l'itérateur <code>pos</code>
<code>erase(pos);</code> <code>erase(first,last);</code>	supprime un ou plusieurs éléments
<code>emplace_back(...);</code> <code>emplace(pos,...);</code>	comme <code>push_back</code> et <code>insert</code> mais l'objet est construit en place et non copié
<code>clear();</code>	supprime tous les éléments

`std::vector<T>`



Méthode	Description
<code>size();</code>	nombre d'éléments stockés
<code>max_size();</code>	capacité maximale selon les limites du système
<code>empty();</code>	retourne <code>size() == 0</code>
<code>capacity();</code>	nombres d'éléments stockables dans la mémoire allouée
<code>resize(s,...);</code>	change la taille, en construisant ou détruisant des éléments
<code>reserve(s);</code>	augmente si nécessaire la capacité. Ne change pas la taille
<code>shrink_to_fit();</code>	diminue si nécessaire la capacité pour qu'elle égale la taille

std::vector<T> iterators



Comme std::array

- Random Access Iterators
- Stockent l'adresse mémoire de l'élément itéré



Différent de std::array

- Ils peuvent être invalidés
- f1 est valide, f2 pas, si la capacité de v doit être augmentée

Sans itérateurs

```
void f1(vector<int>& v) {  
    for(size_t i = 0; i != v.size(); ++i)  
    {  
        if(v[i] % 2 == 0) {  
            v.insert(v.begin()+i, 42);  
            ++i;  
        }  
    }  
}
```

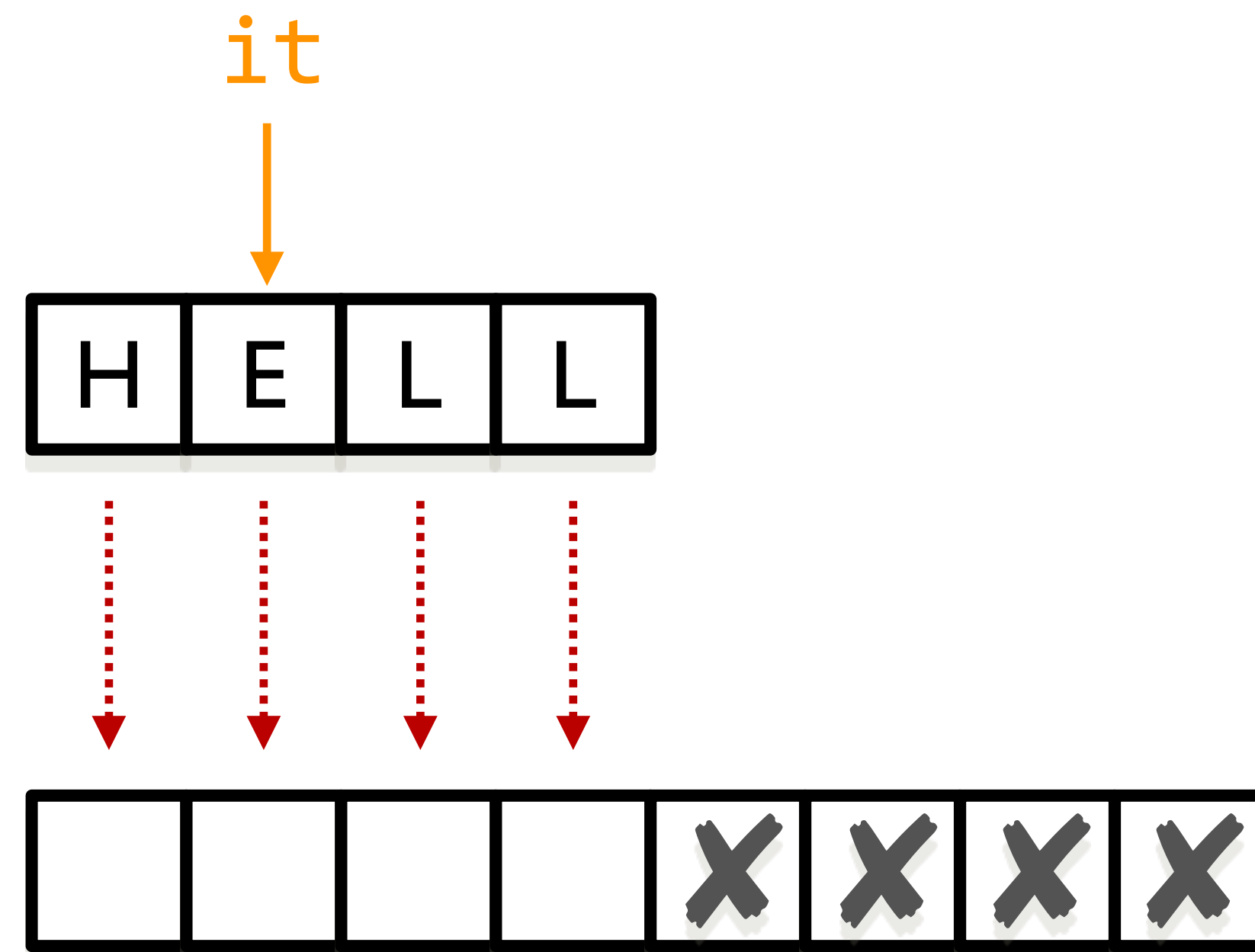


Avec itérateurs

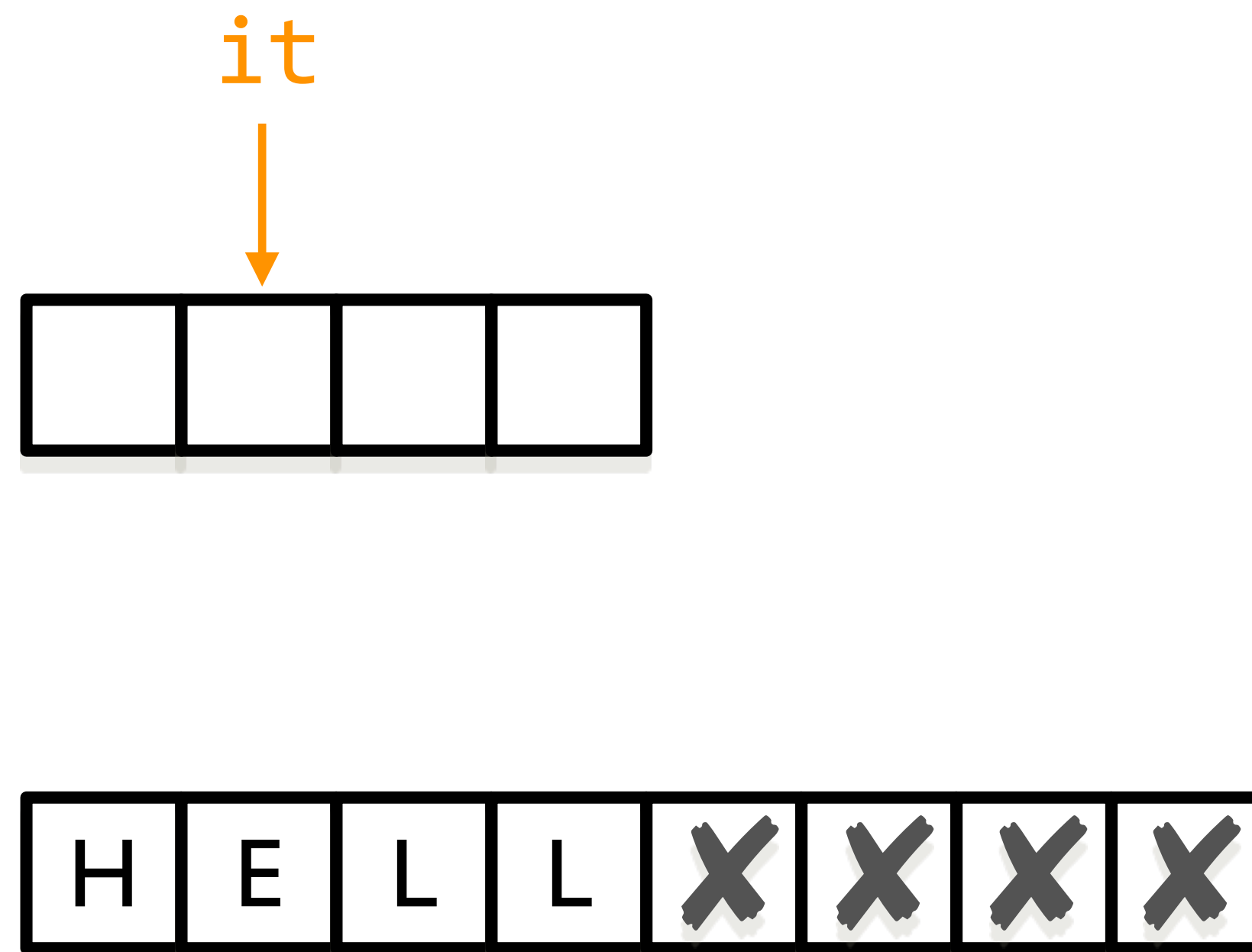
```
void f2(vector<int>& v) {  
    for(auto i = v.begin(); i != v.end(); ++i)  
    {  
        if(*i % 2 == 0) {  
            v.insert(i, 42);  
            ++i;  
        }  
    }  
}
```



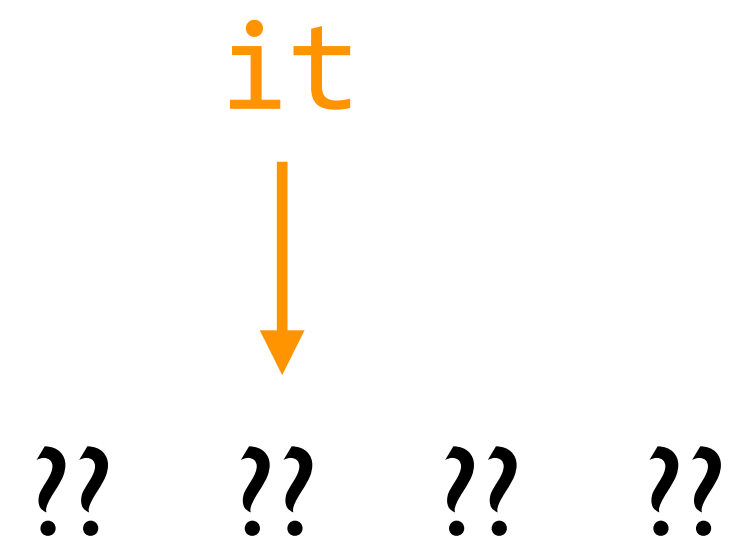
std::vector<T> iterators



std::vector<T> iterators



std::vector<T> iterators





Accès aléatoire : $O(1)$

Suppression en fin : $O(1)$

Insertion en fin (en moyenne) : $O(1)$

Ré-allocation chère en $O(n)$ mais suffisamment rare pour ne pas affecter la complexité moyenne

Insertion/suppression en position i : $O(n - i)$ *(Pourquoi pas $O(i)$?)*
(obligés de décaler vers la droite pour faire de la place ; pas un buffer circulaire)



`std::list<T>`

a.k.a. liste doublement chaînée

`std::list<T>`



- Maintient un attribut taille permettant de la connaître en $O(1)$
- N'offre pas d'accès aléatoire

Expressions non disponibles	
<code>operator[i]</code> <code>at(i)</code>	les opérations d'accès indexé sur la liste
<code>a+n; n+a;</code> <code>a-n; a-b</code>	les propriétés de décalage pour ses itérateurs
<code>a<b; a<=b;</code> <code>a>b; a>=b</code>	les opérateurs de comparaisons de position pour ses itérateurs

std::list<T>



Expression	Description
push_back(t) pop_back()	insère ou supprime un élément en fin
push_front(t) pop_front()	insère ou supprime un élément au début
insert(pos,...)	insère des éléments devant l'itérateur pos
erase(pos) erase(first,last)	supprime des éléments et retourne un itérateur suivant le dernier supprimé
emplace(pos,...) emplace_front() emplace_back()	comme push_back, push_front et insert mais l'objet est construit en place
clear()	supprime tous les éléments

`std::list<T>`



Expression	Description
<code>splice(...)</code>	opération d'épissure
<code>sort()</code>	tri stable en place en $\Theta(n \log(n))$
<code>merge(...)</code>	opération de fusion du tri fusion
<code>remove(...)</code> <code>remove_if(...)</code> <code>unique()</code>	suppression d'éléments selon divers critères
<code>reverse()</code>	inversion de l'ordre des éléments de la liste

std::list<T> iterators



- Bi-directionnal iterators.
 - ⚠ Pas de décalage, d'accès indexé, ni de comparaison
 - ⚠ Fonctions advance(it,n), prev(it,n), next(it,n) et distance(first,last) en O(n)
- Opérateurs ++, --, ==, !=, *
- Stockent un pointeur vers le maillon itéré
 - Valides tant que l'élément est dans la liste
- ⚠ Position dans la liste variable en cas de splice ou de merge

```
list<int> L = { 1, 2, 3, 4, 5, 6 };

auto i4 = find(L.begin(),L.end(),4);

cout << distance(L.begin(),i4) << endl;
// affiche 3

L.splice(L.begin(), L, i4);
// déplace l'élément 4 au début de L

for(auto e : L) cout << e << ' ';
// affiche 4 1 2 3 5 6

cout << distance(L.begin(),i4) << endl;
// affiche 0

cout << *i4 << endl;
// affiche 4
```



`std::forward_list<T>`

a.k.a. liste simplement chaînée

`std::forward_list<T>`



- Pas d'attribut taille, mais uniquement un pointeur vers le premier élément
- N'offre pas d'accès aléatoire, pas d'itération décroissante, pas d'opération en fin de liste

Expressions non disponibles	
<code>--a;</code> <code>a--;</code> <code>*a--;</code>	la décrémentation des itérateurs
<code>rbegin()</code> <code>rend()</code> <code>...</code>	les itérateurs inversés
<code>push_back(t)</code> <code>pop_back()</code> <code>back()</code>	les opérations en fin
<code>size()</code>	la taille, qui n'est pas stockée

`std::forward_list<T>`



- Interface demandant l'élément précédent celui à traiter

Expression	Description
<code>insert_after(it,t)</code> <code>emplace_after(it,...)</code>	insertion après un élément
<code>erase_after(it)</code>	suppression de l'élément suivant
<code>splice_after(it,...)</code>	épissure derrière un élément
<code>before_begin()</code>	itérateur avant le premier élément

`std::forward_list<T>` iterators



- Forward iterators
 - ⚠ Pas de décalage, d'accès indexé, de comparaison, de fonction `prev`, d'operator--
 - ⚠ Fonctions `advance(it,n)`, `next(it,n)` et `distance(first,last)` en $O(n)$
 - Opérateurs `++`, `==`, `!=`, `*`
- Stockent un pointeur vers le maillon itéré
 - Valides tant que l'élément est dans la liste
- ⚠ Position dans la liste *potentiellement* variable en cas de `insert_after`, `erase_after`, `splice_after`, mais le plus souvent non, grâce à l'API `_after`

Supprimer toutes les valeurs (1) - `std::list<T>`



```
template < typename T >
void supprimer(std::list<T>& L, T val)
{
    for(auto it = L.begin(); it != L.end(); )
    {
        if(*it == val) it = L.erase(it);
        else ++it;
    }
}
```

Supprimer toutes les valeurs (2) - `std::forward_list<T>`



```
template < typename T >
void supprimer(std::forward_list<T>& L, T val)
{
    auto prec = L.before_begin();
    for(auto it = next(prec); it != L.end(); it = next(prec))
    {
        if(*it == val) L.erase_after(prec);
        else ++prec;
    }
}
```

Supprimer toutes les valeurs (3) - `std::vector<T>`



```
template < typename T >
void supprimer(std::vector<T>& V, T val)
{
    auto i_write = V.begin();
    for( auto i_read = V.begin(); i_read != V.end(); ++i_read )
    {
        if(*i_read != val) {
            *i_write = std::move(*i_read);
            ++i_write;
        }
    }
    V.erase(i_write, V.end());
}
```



std::deque<T>
a.k.a. double-ended queue

std::deque<T>



- Étend les fonctionnalités de `std::vector` en y ajoutant

Méthode	Description
<code>push_front(t)</code> <code>emplace_front(...)</code>	insère un élément au début
<code>pop_front()</code>	supprime le premier élément

- N'a pas de méthode `capacity()` ⚠

std::deque<T> vs. std::vector<T>



- Accès aux données : $O(1)$ mais un peu plus lent (dû aux indirections)
- Opérations en début : beaucoup plus efficaces en $O(1)$
- Ré-allocations : plus efficaces, en $O(n/c)$ avec c la taille des blocs.

```
// taille des blocs dans la STL de llvm (Mac OS X)
template <class _ValueType, class _DiffType>
struct __deque_block_size
{
    static const _DiffType value =
        sizeof(_ValueType) < 256 ?
        4096 / sizeof(_ValueType) : 16;
};
```

4096 octets par chunk
si objet petit

16 objets
sinon

Mémoire additionnelle



`std::array<T,N>` $\emptyset$

`std::vector<T>` $3*p + O(n)*t$

1 pointeur sur le premier élément

2 `size_t` (taille et capacité)

jusqu'à $O(n)$ éléments alloués non-utilisés

`std::forward_list<T>` $(n+1)*p$

1 pointeur par maillon (n maillons + 1 maillon vide)

`std::list<T>` $(2*n+2)*p + p$

2 pointeur par maillon (n maillons + 1 maillon vide)

1 `size_t` (taille)

`std::deque<T>` $O(n/B)*p + O(B)*t + 6*p$

$O(n/B)$ pointeurs dans la map

jusqu'à $O(B)$ éléments de chunk alloués non-utilisés

1 pointeur sur le premier élément de map

5 `size_t` (début, capacité de map ; début, capacité de chunk ; taille)

Quel conteneur choisir ?



	array	vector	forward_list	list	deque
Mémoire additionnelle	0	$3 \cdot p + O(n) \cdot t$ si taille != capacité	$(n+1) \cdot p$	$(2 \cdot n + 3) \cdot p$	$O(n/B) \cdot p + O(B) \cdot t + 6 \cdot p$
operator[]	$O(1)$	$O(1)$	N/A	N/A	$O(1)$, mais un peu plus lent
push_front / pop_front	N/A	$O(n)$	$O(1)$	$O(1)$	$O(1)$
insert / erase au milieu	N/A	$O(n)$	$O(1)$ si élément connu	$O(1)$ si élément connu	$O(n)$
push_back / pop_back	N/A	$O(1)$ amorti $O(n)$ au pire	N/A	$O(1)$	$O(1)$

$t = \text{sizeof}(T); \quad p = \text{sizeof}(T^*); \quad B = \text{__deque_block_size}\langle T, \text{size_t} \rangle::\text{value};$