

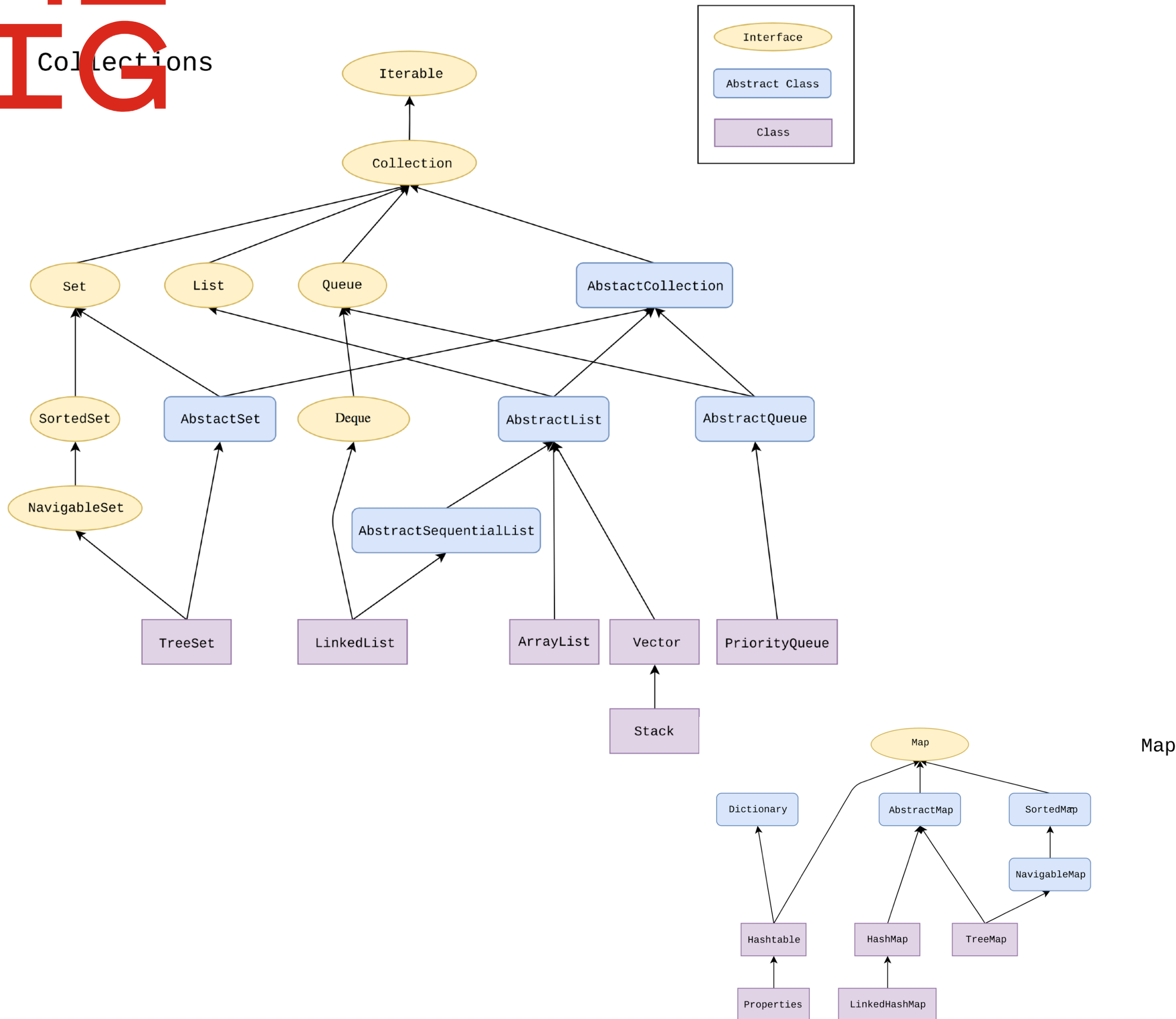
Structures Linéaires





1. Introduction

Java collections vs. C++ containers



Container class templates

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Unordered associative containers:

unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)



Container class templates

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Unordered associative containers:

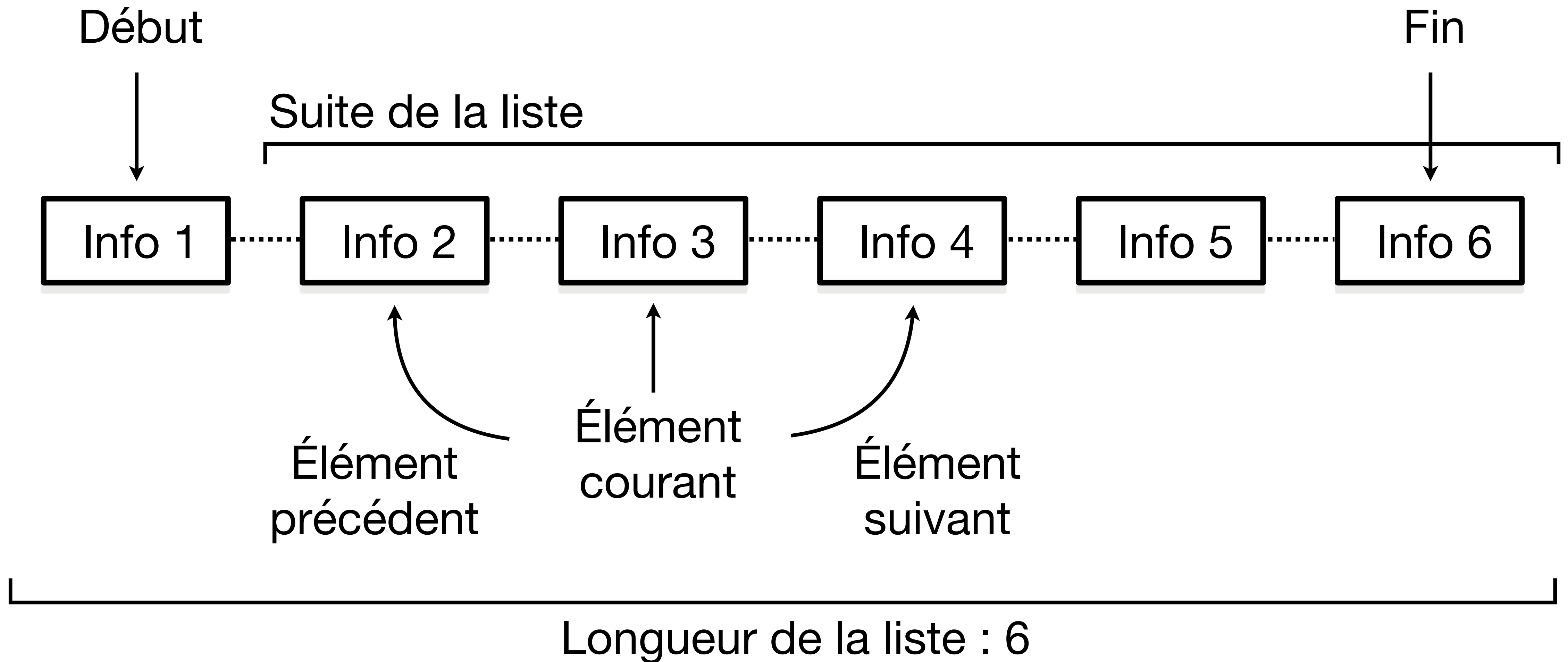
unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

Chapitre 4 :
structures
linéaires

Chapitre 5 : arbres
équilibrés

Chapitre 7 : tables
de hachage

Éléments communs à toutes les structures linéaires





Opérations

- ... au début
- insérer un élément ...



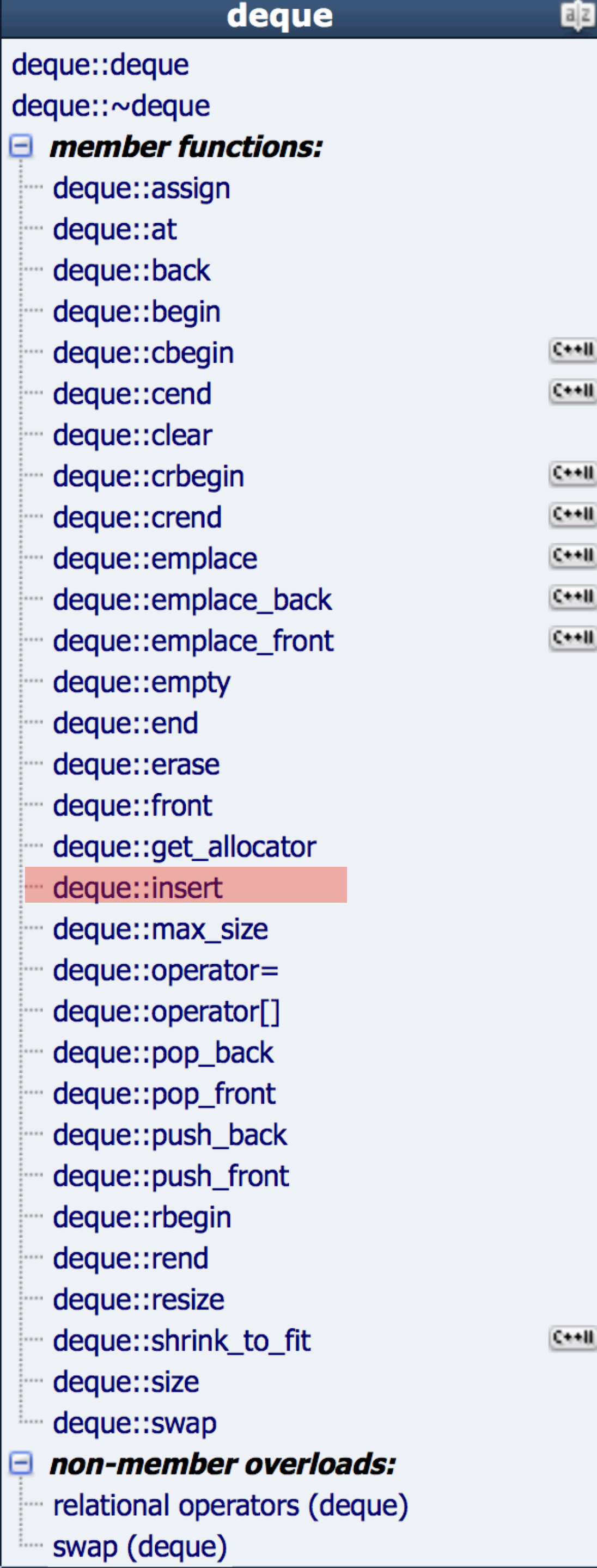
```
deque
deque::deque
deque::~~deque
member functions:
deque::assign
deque::at
deque::back
deque::begin
deque::cbegin
deque::cend
deque::clear
deque::crbegin
deque::crend
deque::emplace
deque::emplace_back
deque::emplace_front
deque::empty
deque::end
deque::erase
deque::front
deque::get_allocator
deque::insert
deque::max_size
deque::operator=
deque::operator[]
deque::pop_back
deque::pop_front
deque::push_back
deque::push_front
deque::rbegin
deque::rend
deque::resize
deque::shrink_to_fit
deque::size
deque::swap
non-member overloads:
relational operators (deque)
swap (deque)
```

Opérations

- insérer un élément ...

- ... au début
- ... en fin





Opérations

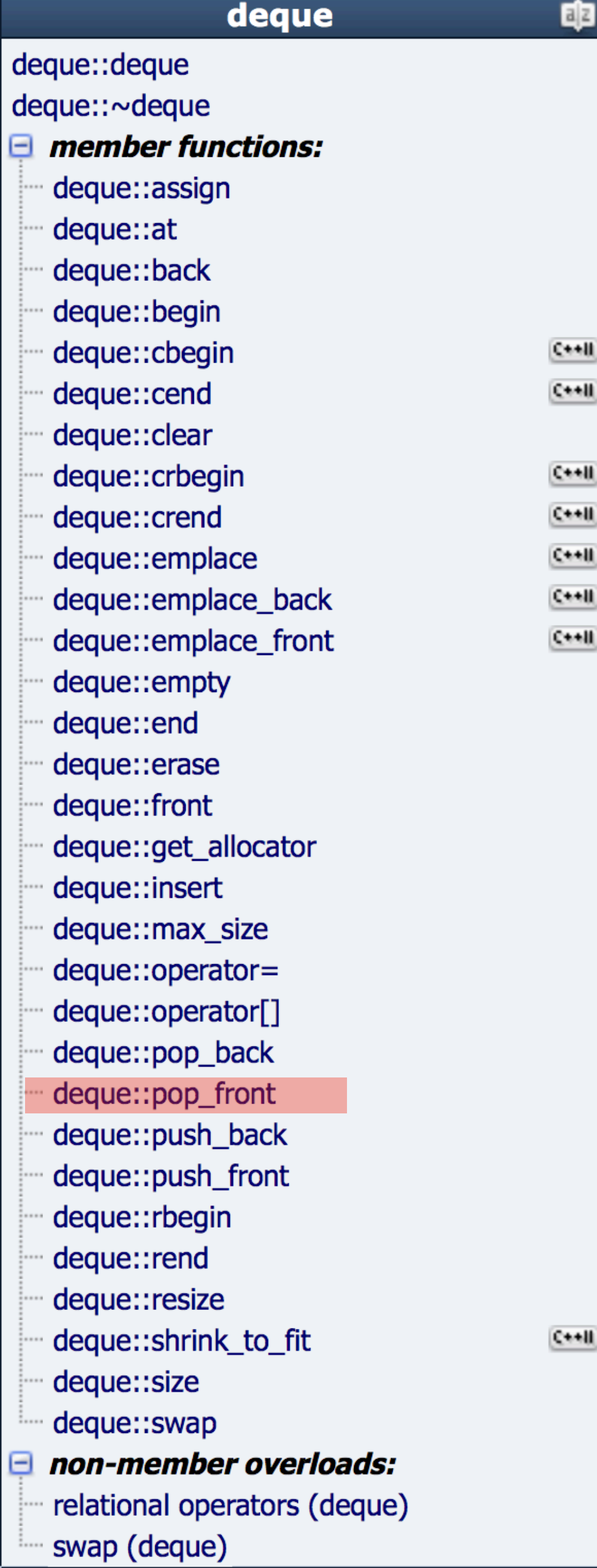
- insérer un élément ...

- ... au début

- ... en fin

- ... en position
quelconque



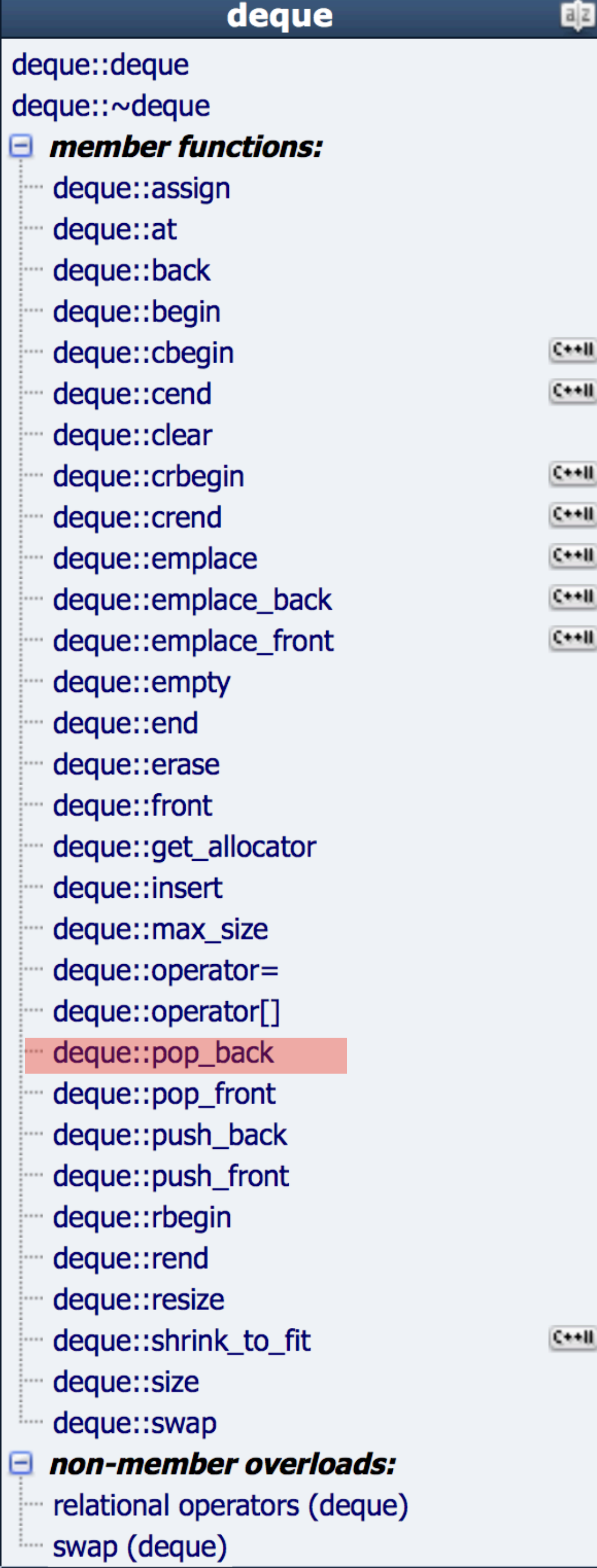


Opérations

- insérer un élément ...
- supprimer un élément ...

- ... au début



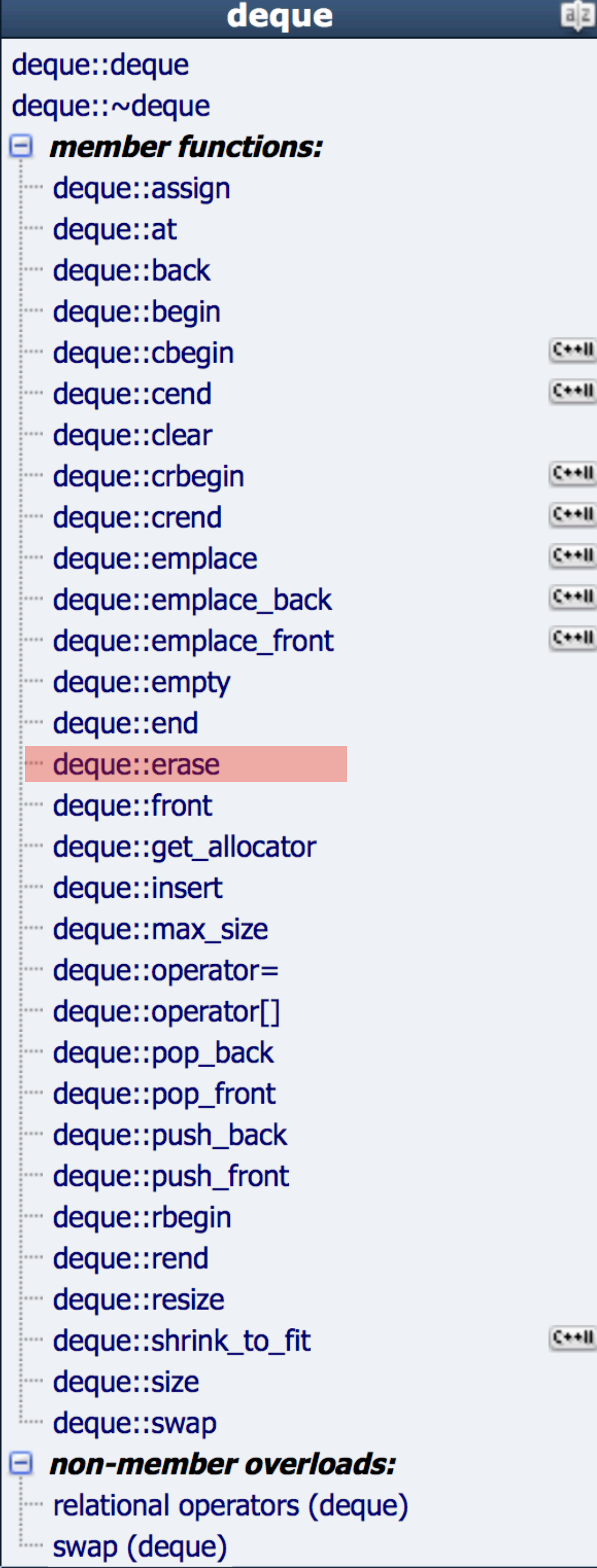


Opérations

- insérer un élément ...
- supprimer un élément ...

- ... au début
- ... en fin





Opérations

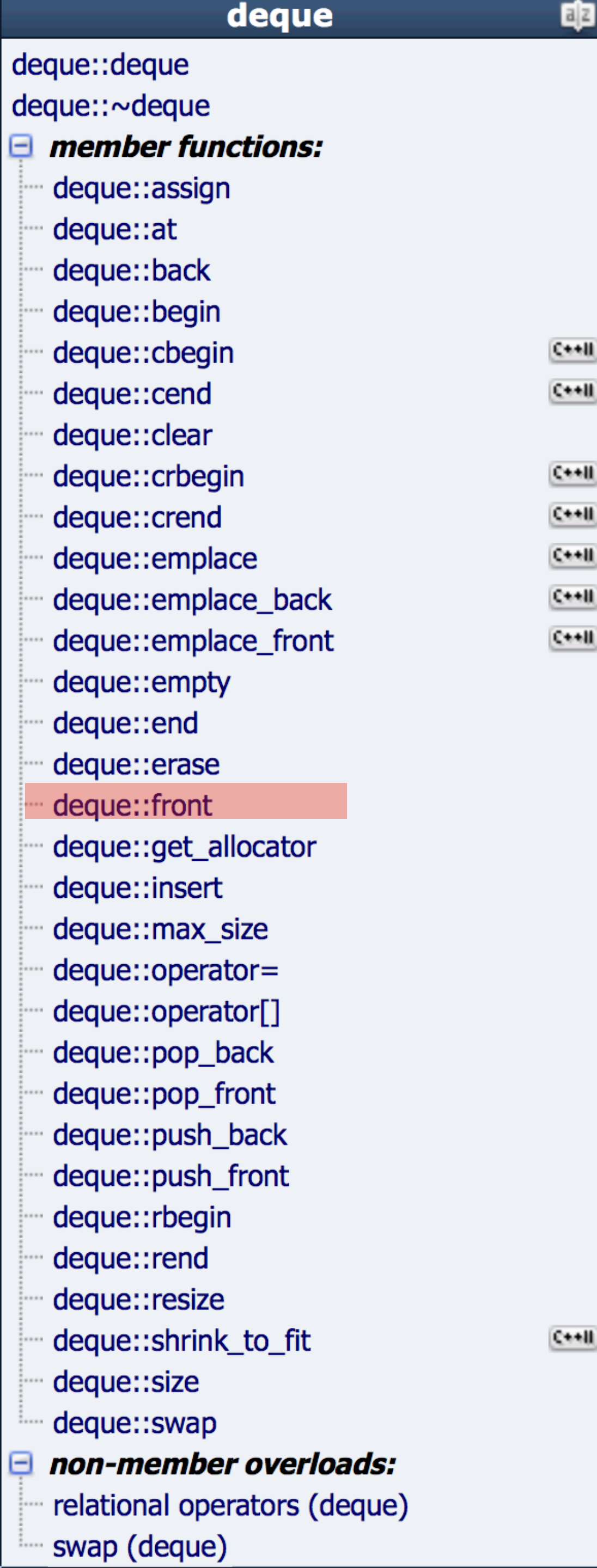
- insérer un élément ...
- supprimer un élément ...

- ... au début

- ... en fin

- ... en position
quelconque





Opérations

- insérer un élément ...
- supprimer un élément ...
- accéder à un élément ...

- ... au début



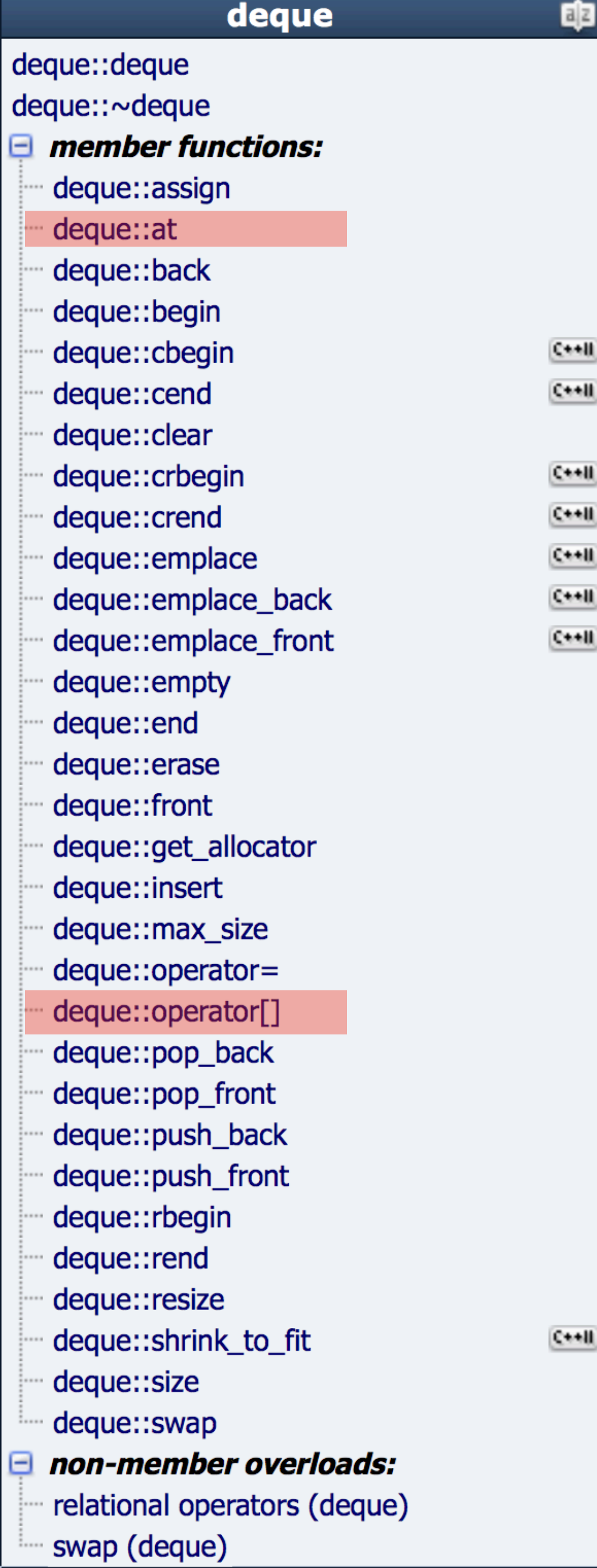


Opérations

- insérer un élément ...
- supprimer un élément ...
- accéder à un élément ...

- ... au début
- ... en fin





Opérations

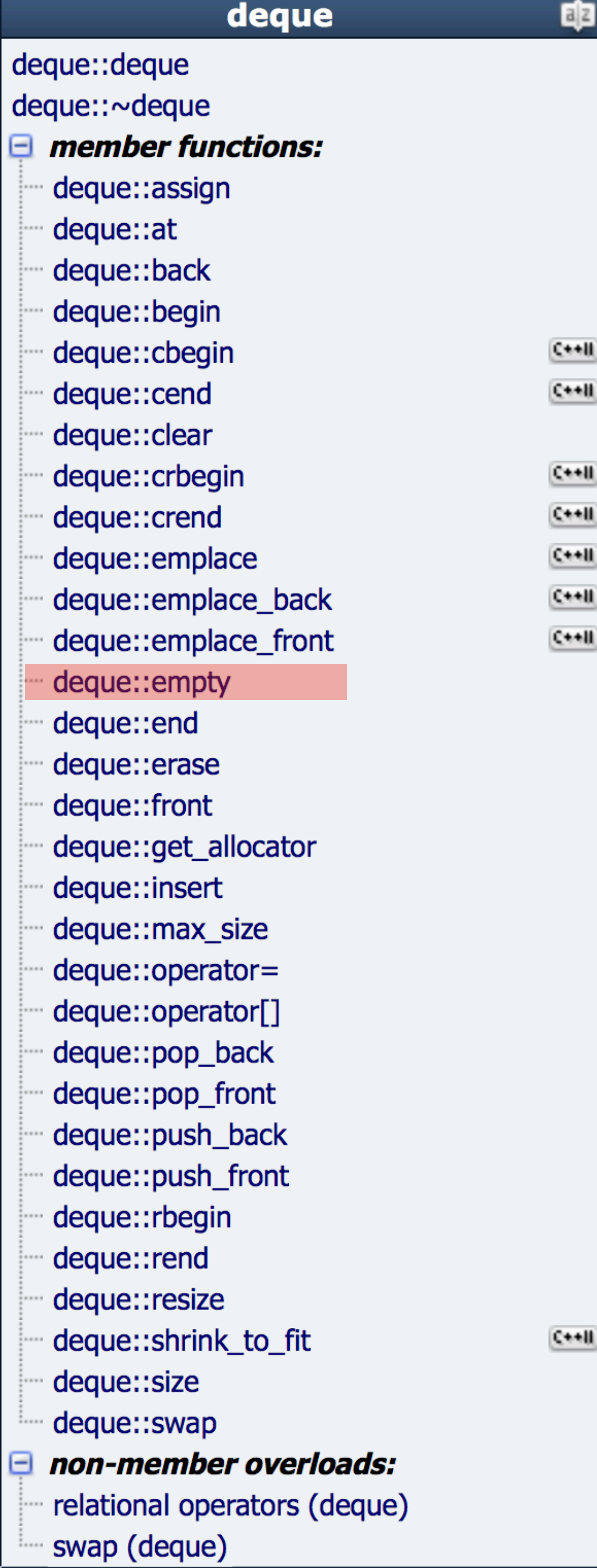
- insérer un élément ...
- supprimer un élément ...
- accéder à un élément ...

- ... au début

- ... en fin

- ... en position quelconque





Opérations

- insérer un élément ...
- supprimer un élément ...
- accéder à un élément ...
- savoir si la liste est vide

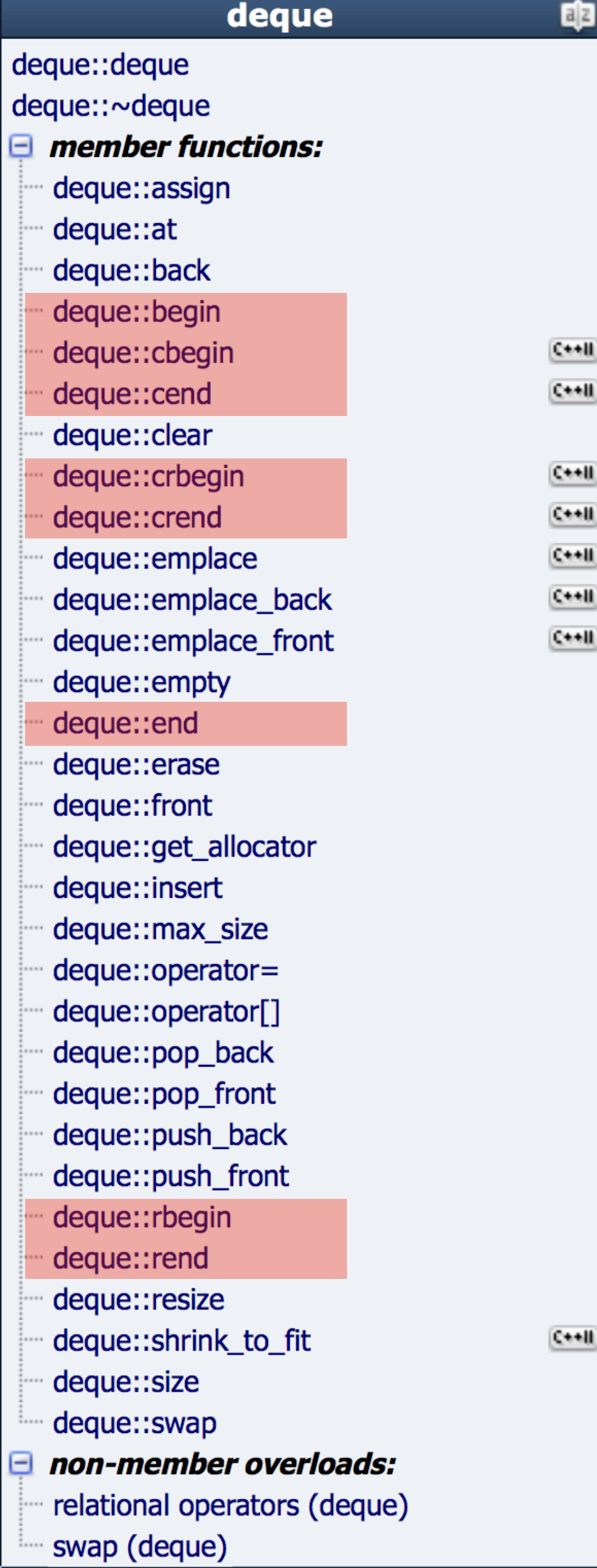




Opérations



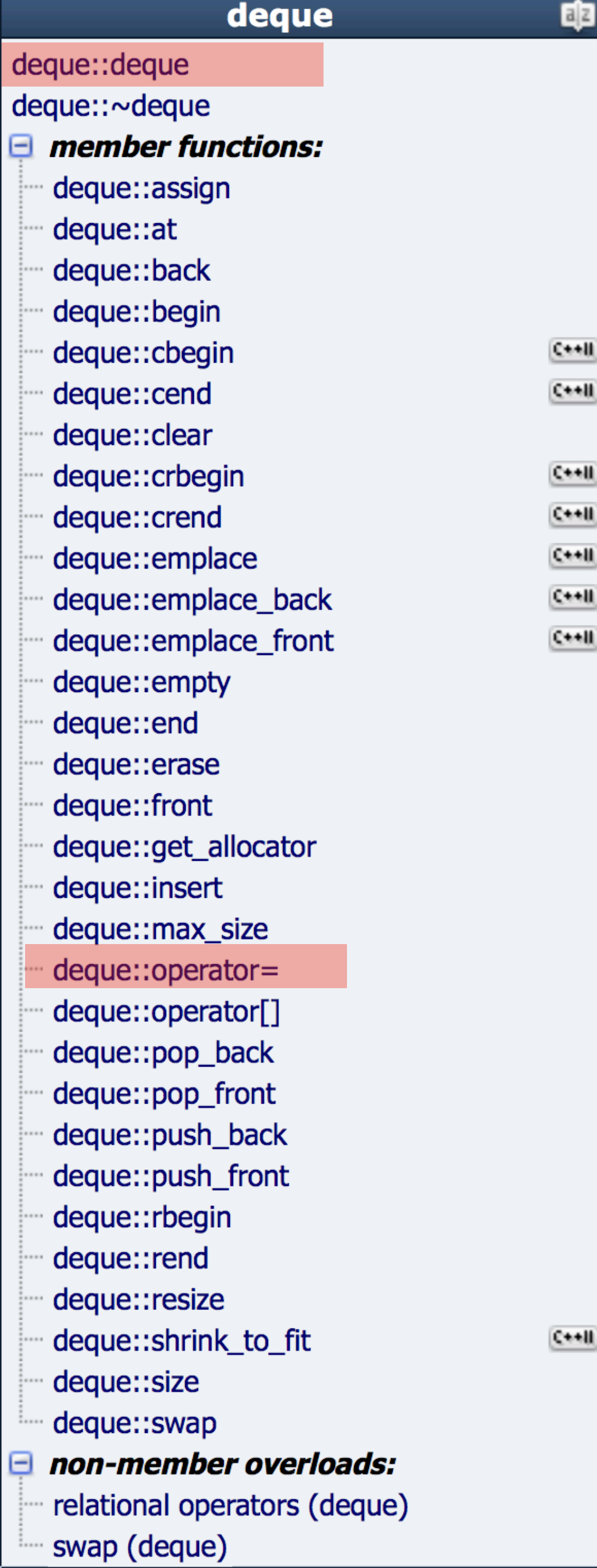
- insérer un élément ...
- supprimer un élément ...
- accéder à un élément ...
- savoir si la liste est vide
- rechercher un élément



Opérations



- insérer un élément ...
- supprimer un élément ...
- accéder à un élément ...
- savoir si la liste est vide
- rechercher un élément
- parcourir la liste et traiter chaque élément, éventuellement avec des itérateurs



Opérations

- insérer un élément ...
- supprimer un élément ...
- accéder à un élément ...
- savoir si la liste est vide
- rechercher un élément
- parcourir la liste et traiter chaque élément, éventuellement avec des itérateurs
- Copier / déplacer la structure

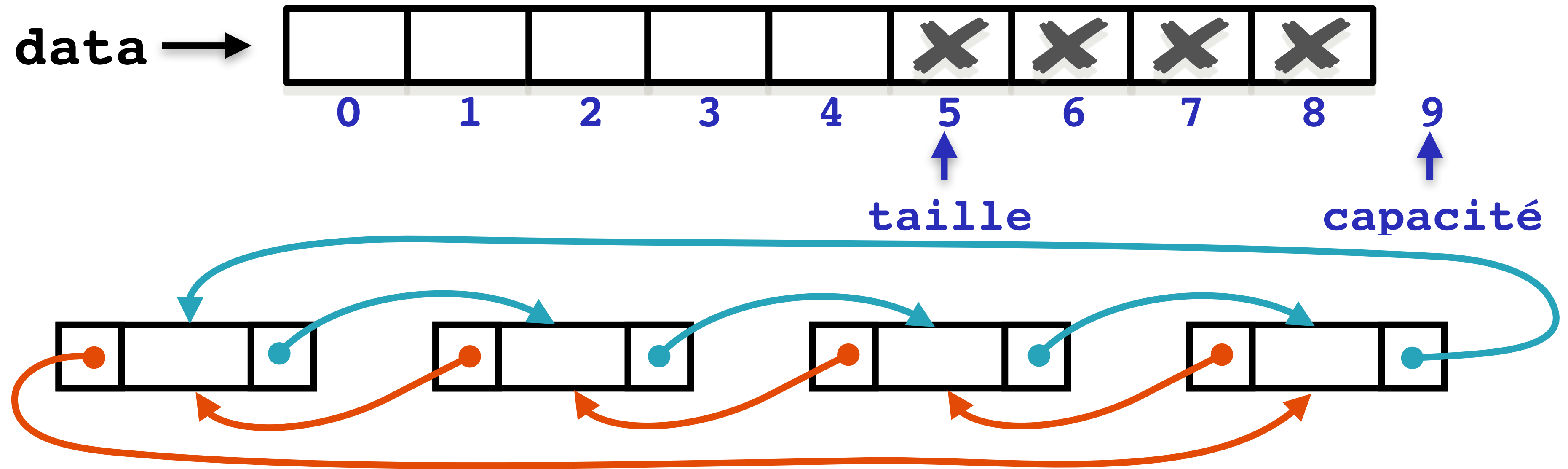


Mises en oeuvres 2 approches principales



Tableau

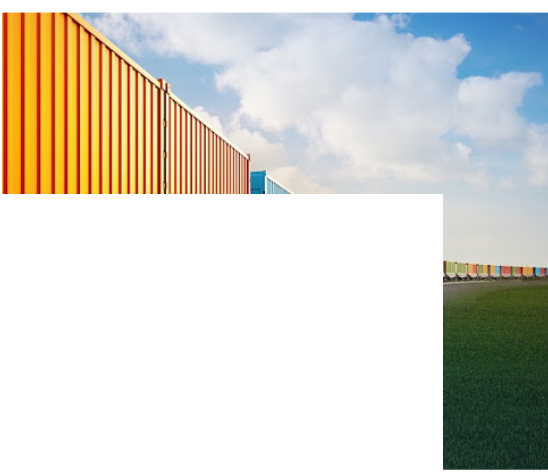
Mémoire consécutive



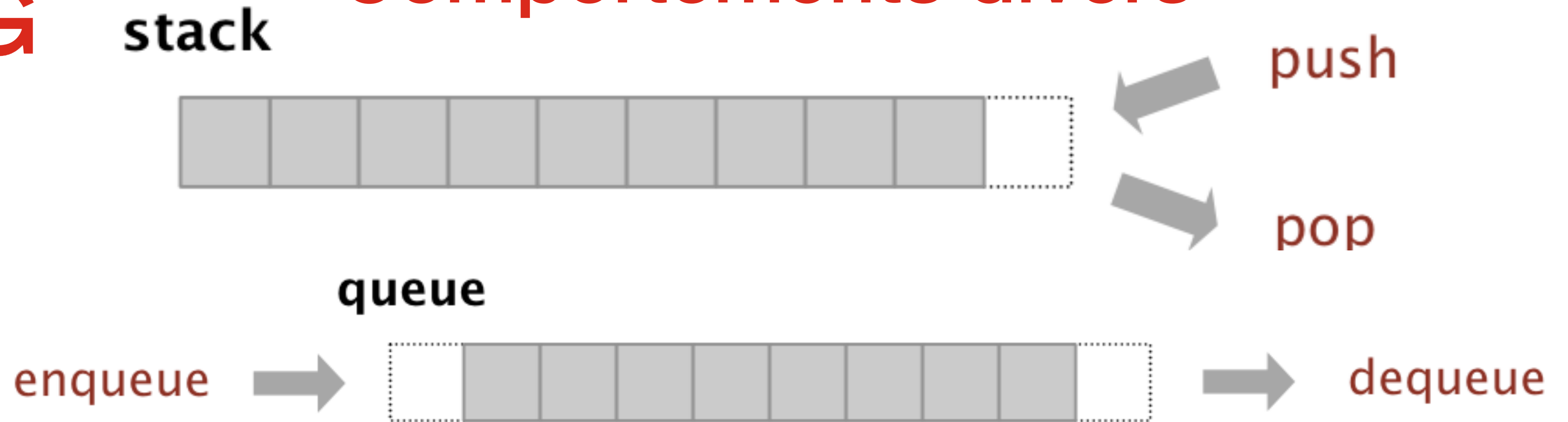
Liste chaînée

*Position arbitraire
en mémoire*

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)



Comportements divers



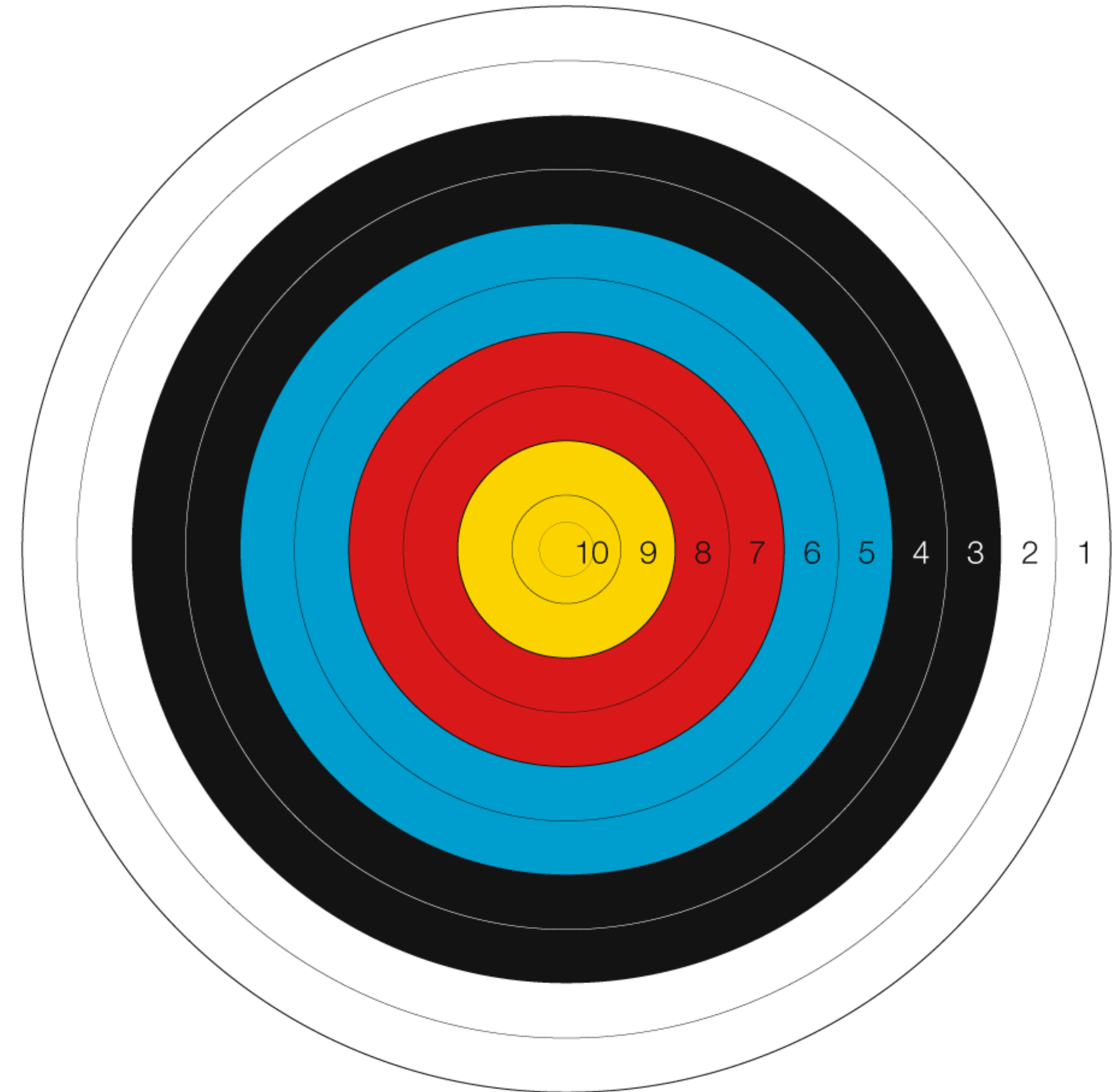
Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Objectifs du chapitre



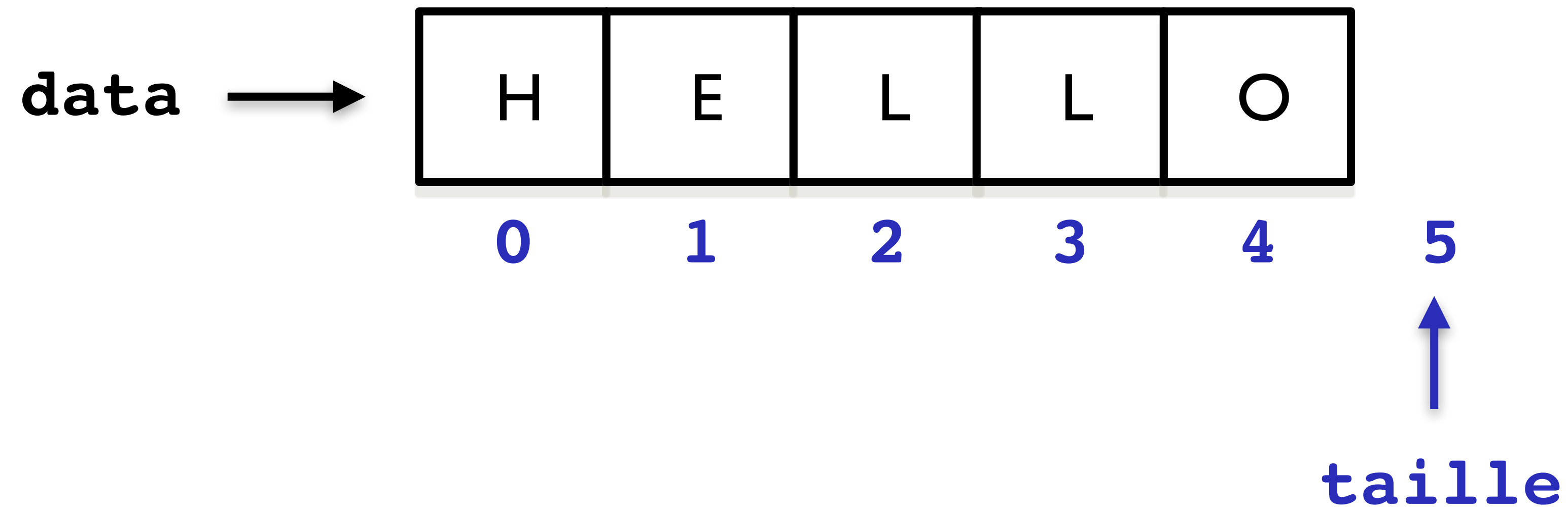
- Etudier les mises en oeuvre standards: complexités temporelle et spatiale, mise oeuvre en C++
- Etudier les comportements standards: opérations disponibles, applications pratiques
- Etre capable de choisir / écrire la bonne structure pour **votre application**





2. Tableaux de taille fixe

Tableau de taille fixe



```
classe TableauTailleFixe  
    Entier constant taille  
    Element[] data[taille]
```

Tableau de taille fixe



- Pas d'insertion / suppression d'éléments
- Accès aux éléments en $O(1)$ en calculant leur adresse.

```
méthode adresse(i)  
    retourner data + i * taille_de(Element)
```

- C et C++ fournissent nativement des tableaux de taille fixée à la **compilation**
- C99 fournit nativement des tableaux de taille fixée à l'**exécution**, pas C++.

Mise en oeuvre (inutile) en C++ d'un tableau de taille fixée à l'exécution



- Tableau générique d'éléments de type `T`
- Données stockées dans un bloc mémoire de taille `*sizeof(T)` octets alloué dynamiquement et pointé par `data`
- L'élément d'indice `i` est à l'adresse `data + i * sizeof(T)`

```
using byte = unsigned char;
```

```
template <typename T>  
class Tableau {
```

```
    size_t taille;  
    byte* data;
```

```
    byte* ptr(size_t i) {  
        return data + i * sizeof(T);  
    }
```

```
    ...
```

Mise en oeuvre (inutile) en C++ d'un tableau de taille fixée à l'exécution



- En C++17, on peut utiliser le type `std::byte`.
- On n'utilise pas le type `void*` parce qu'`operator+(int)` n'est pas défini dans ce cas

```
using std::byte; // C++17
```

```
template <typename T>  
class Tableau {
```

```
    size_t taille;  
    byte* data;
```

```
    byte* ptr(size_t i) {  
        return data + i * sizeof(T);  
    }
```

...

Mise en oeuvre (inutile) en C++ d'un tableau de taille fixée à l'exécution



- Le constructeur
 - initialise `taille`
 - alloue `n*sizeof(T)` octets
 - construit par défaut `n` objets de type `T`
- Le destructeur
 - détruit les `n` objets
 - libère la mémoire
- Bien ... mais qu'advient-il si `new(p) T{}` lève une exception ?

public:

```
Tableau(size_t n) : taille(n) {  
    data = reinterpret_cast<byte*>(  
        ::operator new(n*sizeof(T)));  
  
    for(size_t i = 0; i < taille; ++i)  
        new(ptr(i)) T{};  
}  
  
~Tableau() {  
    for(size_t i = 0; i < taille; ++i)  
        destroy_at(ptr(i));  
    ::operator delete(data);  
}
```

Mise en oeuvre (inutile) en C++ d'un tableau de taille fixée à l'exécution



- En utilisant `taille` comme variable de boucle, sa valeur correspond en tout temps au nombre d'objets construits
- Cela permettrait au destructeur de détruire proprement les objets
- Sauf ... que le destructeur n'est pas appelé par une exception tant que le `Tableau` n'est pas entièrement construit

`public:`

```
Tableau(size_t n) {  
    data = reinterpret_cast<byte*>(  
        ::operator new(n*sizeof(T)));  
  
    for(taille = 0; taille < n; ++taille)  
        new(ptr(taille)) T{};  
}  
  
~Tableau() {  
    for(size_t i = 0; i < taille; ++i)  
        destroy_at(ptr(i));  
    ::operator delete(data);  
}
```

Mise en oeuvre (inutile) en C++ d'un tableau de taille fixée à l'exécution



- On peut forcer l'appel au destructeur en cas de problème en utilisant `try { } catch( )`
- Mais on peut faire mieux...

public:

```
Tableau(size_t n) {  
    data = reinterpret_cast<byte*>(  
        ::operator new(n*sizeof(T)));  
    try {  
        for (taille = 0; taille < n; ++taille)  
            new(ptr(taille)) T{};  
    }  
    catch(...) {  
        this->~Tableau();  
    }  
}
```

Mise en oeuvre (inutile) en C++

d'un tableau de taille fixée à l'exécution



- Le **constructeur par défaut** crée un tableau de zéro élément. Il ne lève jamais d'exception
- Le **constructeur** (`size_t`) appelle le constructeur par défaut dans sa liste d'initialisation
- Comme le `Tableau` est déjà construit par défaut, le **destructeur est appelé** si une exception est levée

`public:`

```
Tableau() noexcept : taille(0),  
                    data(nullptr) {  
}
```

```
Tableau(size_t n) : Tableau() {  
    data = reinterpret_cast<byte*>(  
        ::operator new(n*sizeof(T)));  
  
    for(; taille < n; ++taille)  
        new(ptr(taille)) T{};  
}
```

Mise en oeuvre (inutile) en C++ d'un tableau de taille fixée à l'exécution



- On ne permet pas ici de copier les objets de type `Tableau`
- La mise en oeuvre de l'**accès aux éléments** revient à interpréter le bon emplacement mémoire comme étant de type `T`
- Il faut en fournir 2 versions, selon que `Tableau` est constant ou variable

```
Tableau(const Tableau&) = delete;  
Tableau& operator=(const Tableau&) = delete;
```

```
size_t size() const noexcept {  
    return taille;  
}
```

```
T& at(size_t i) noexcept {  
    return *(reinterpret_cast<T*>(ptr(i)));  
}
```

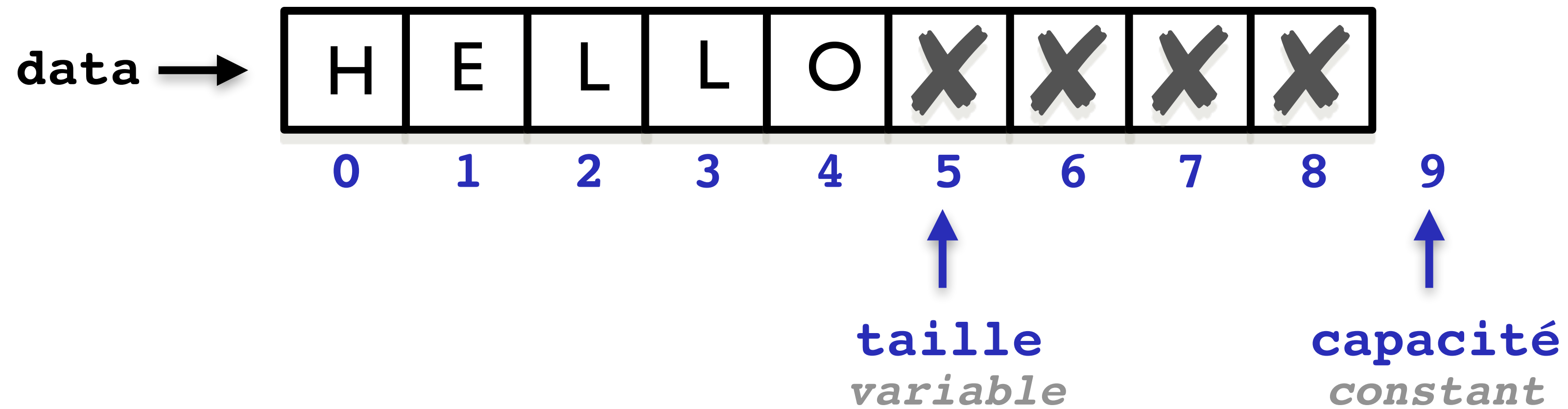
```
const T& at(size_t i) const noexcept {  
    return *(reinterpret_cast<T*>(ptr(i)));  
}
```



Tableaux de capacité fixe

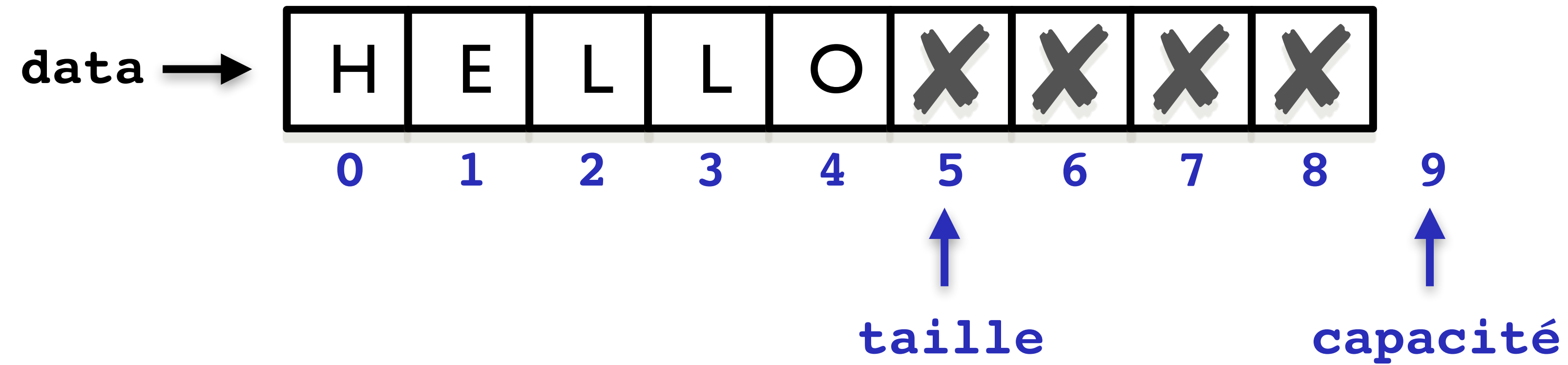
Mais de taille variable

Tableau de capacité fixe

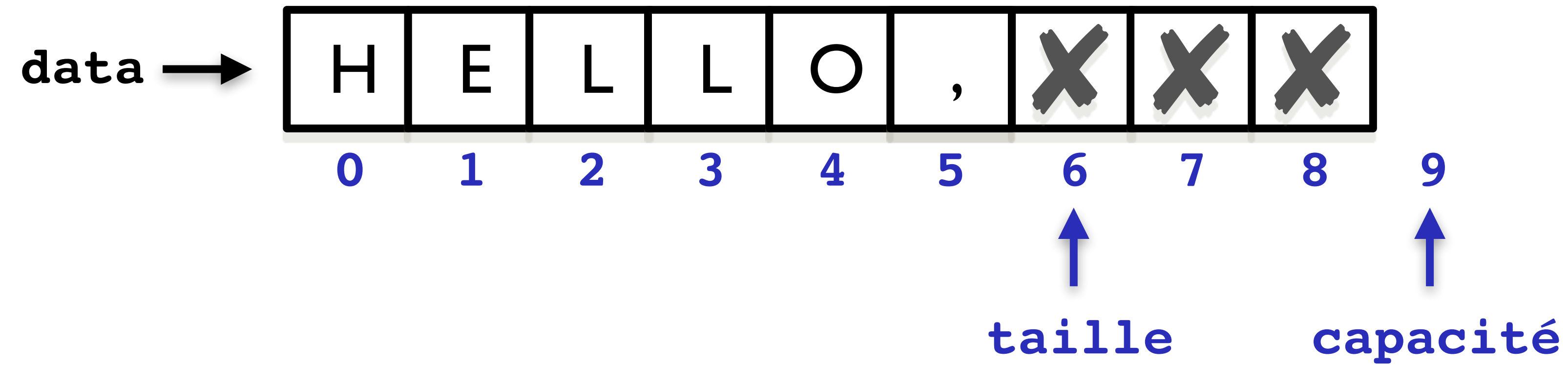


```
classe TableauCapacitéFixe
    Entier constant capacité
    Entier variable taille
    Element[] data[capacité]
```

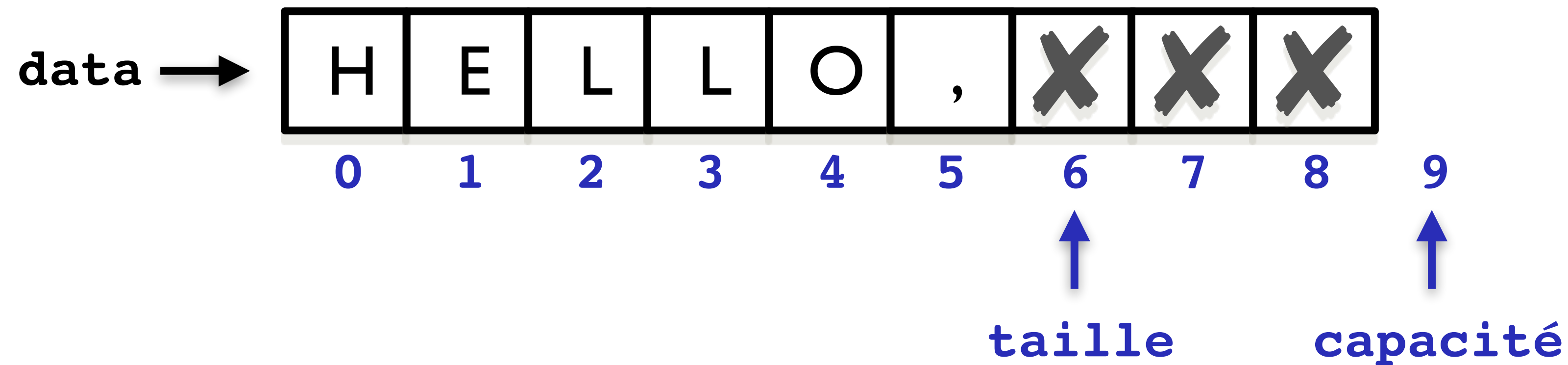
Insertion en fin



Insertion en fin



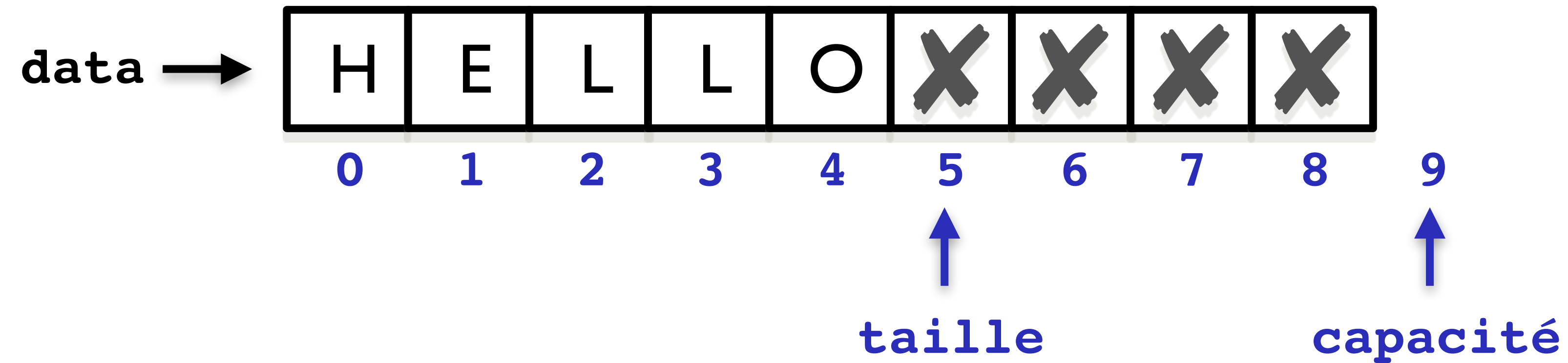
Insertion en fin



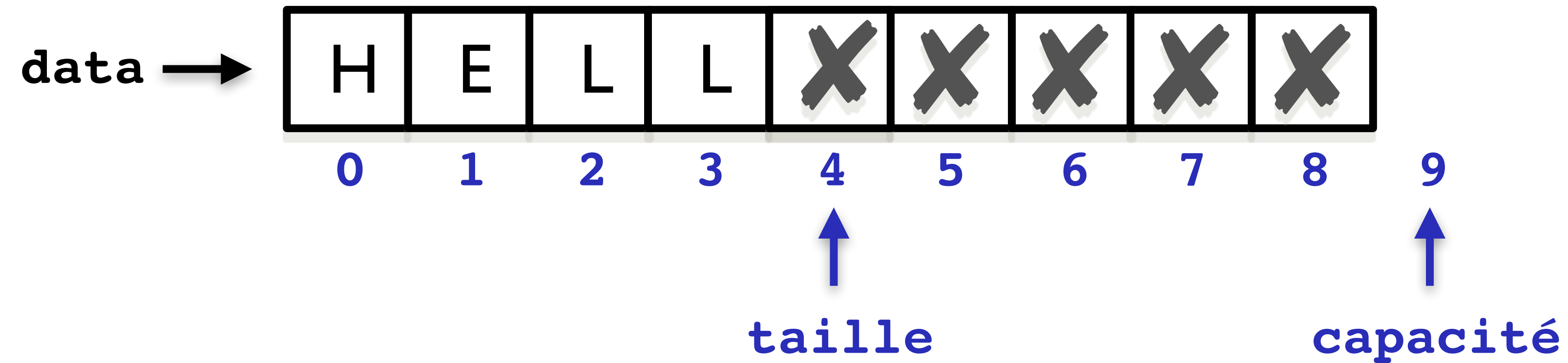
```
méthode insérer_en_fin(v)
  si taille >= capacité
    alerter tableau plein
  construire data[taille] ← v
  taille ← taille + 1
```

- Modifie le contenu et la taille
- Complexité $O(1)$

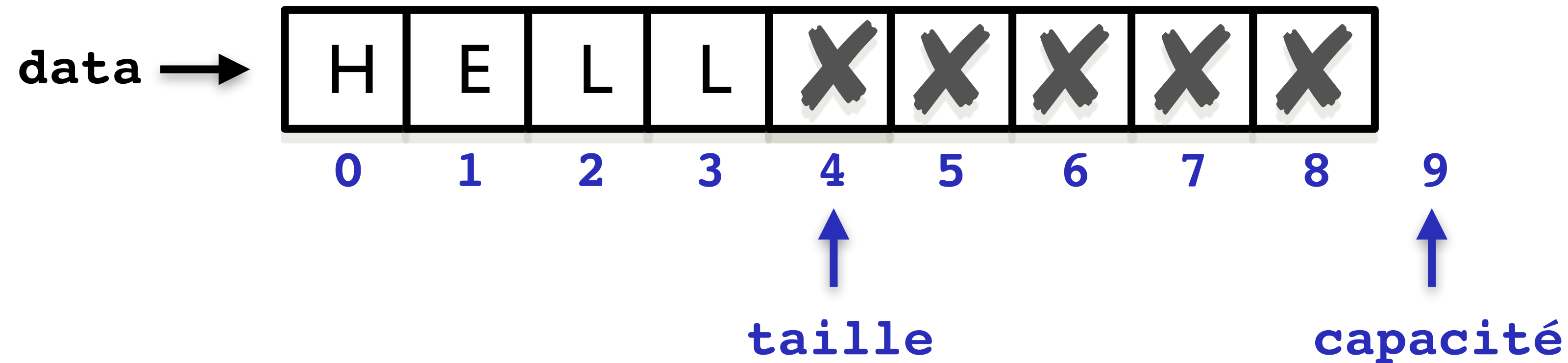
Supprimer en fin



Supprimer en fin



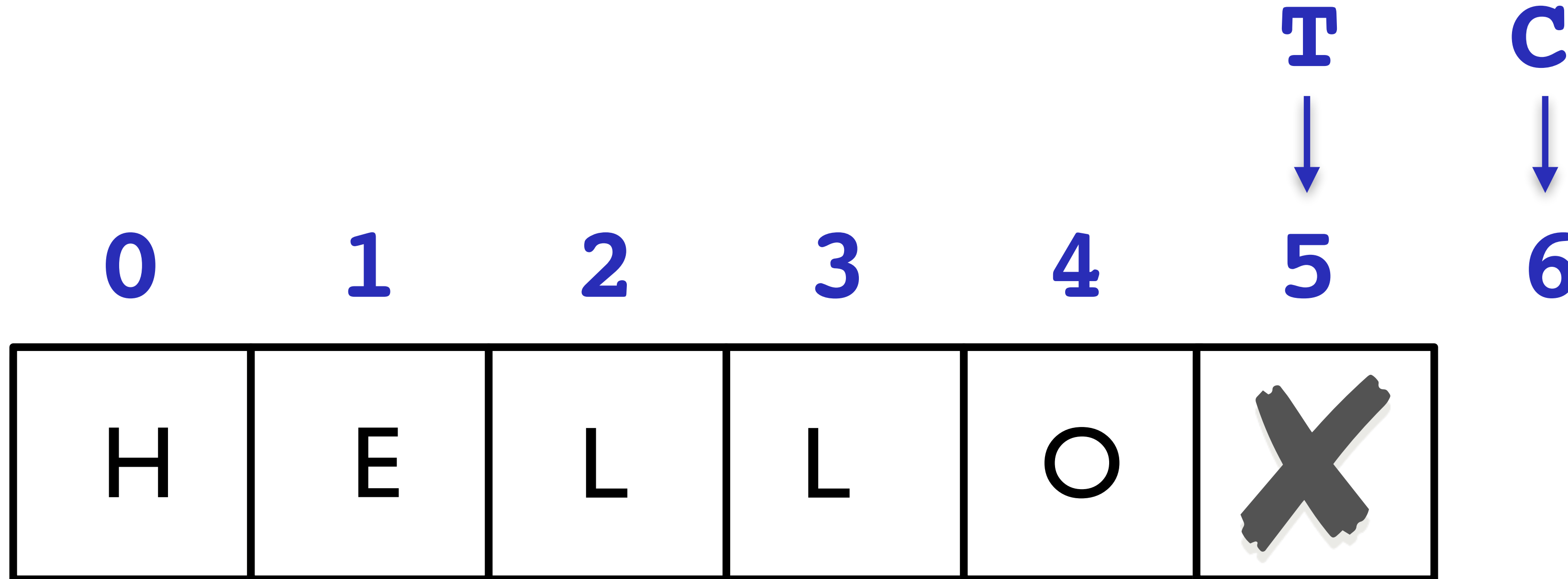
Supprimer en fin



```
méthode supprimer_en_fin()  
  si taille == 0  
    alerter tableau vide  
  détruire data[taille - 1]  
  taille ← taille - 1
```

- Modifie le contenu et la taille
- Complexité $O(1)$

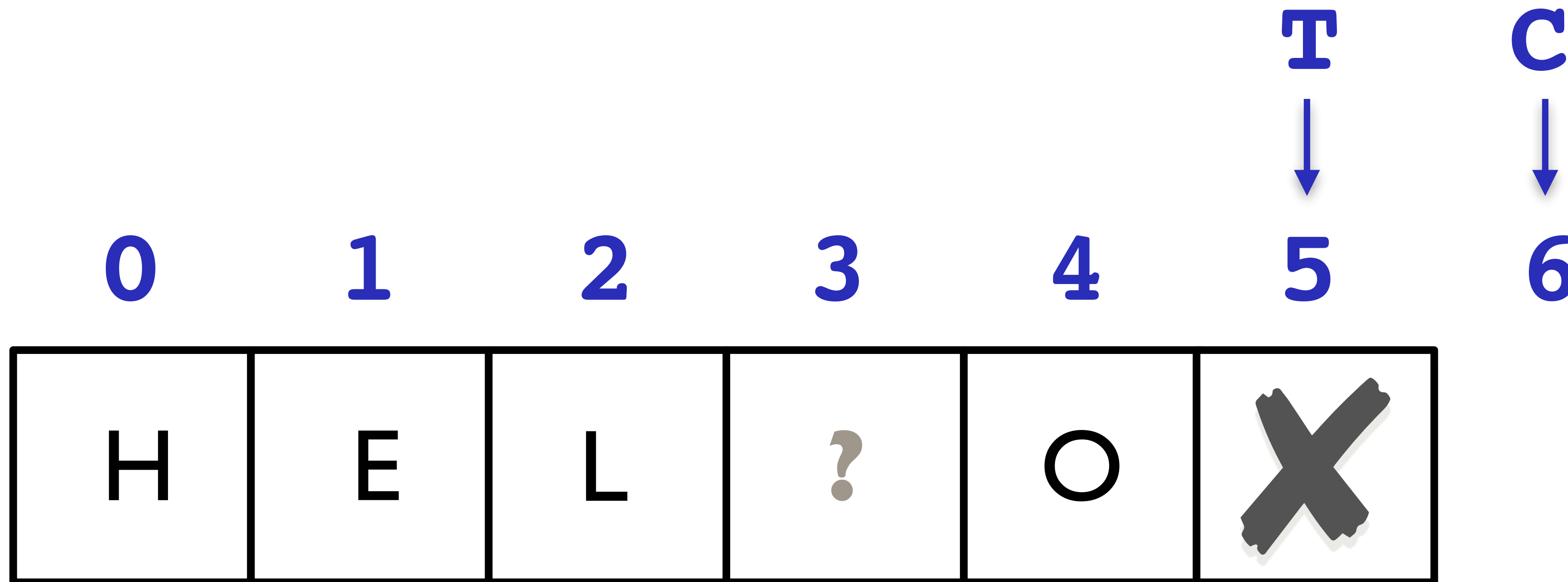
Supprimer en position i



↑
supprimer(2)

- Détruire un élément
- Déplacer les suivants
- Modifier la taille

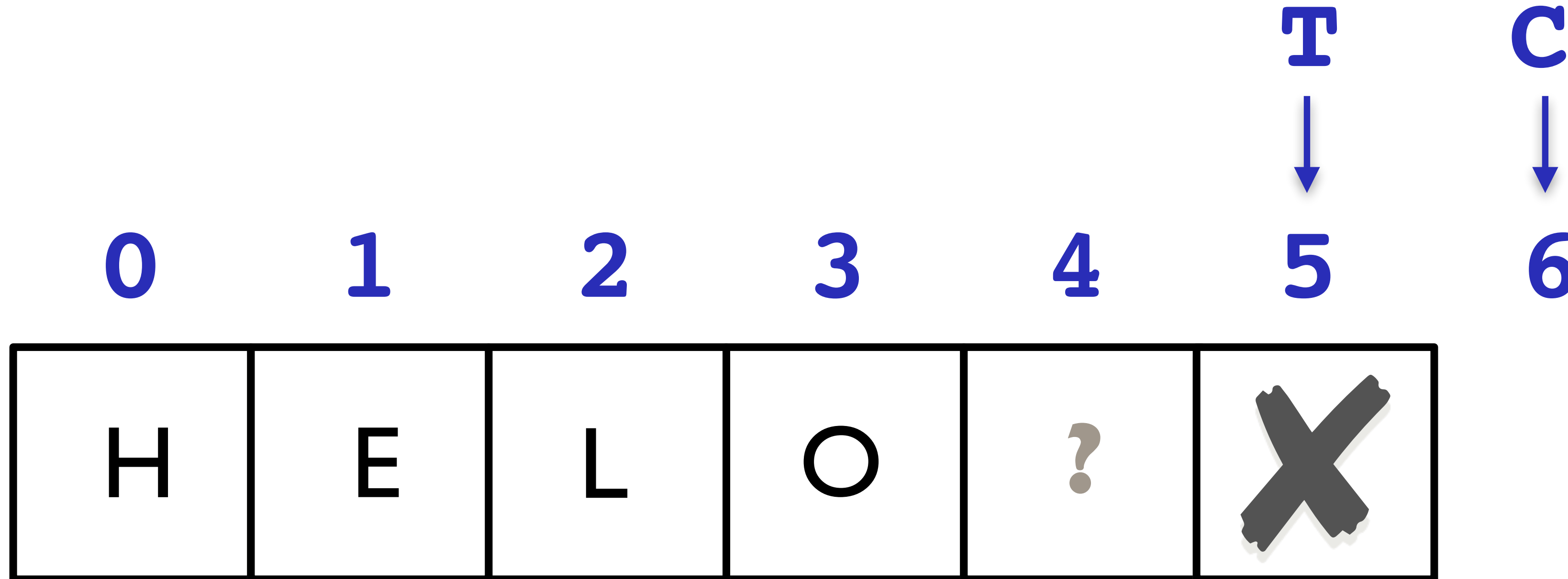
Supprimer en position i



↑
supprimer(2)

- Détruire un élément
- Déplacer les suivants
- Modifier la taille

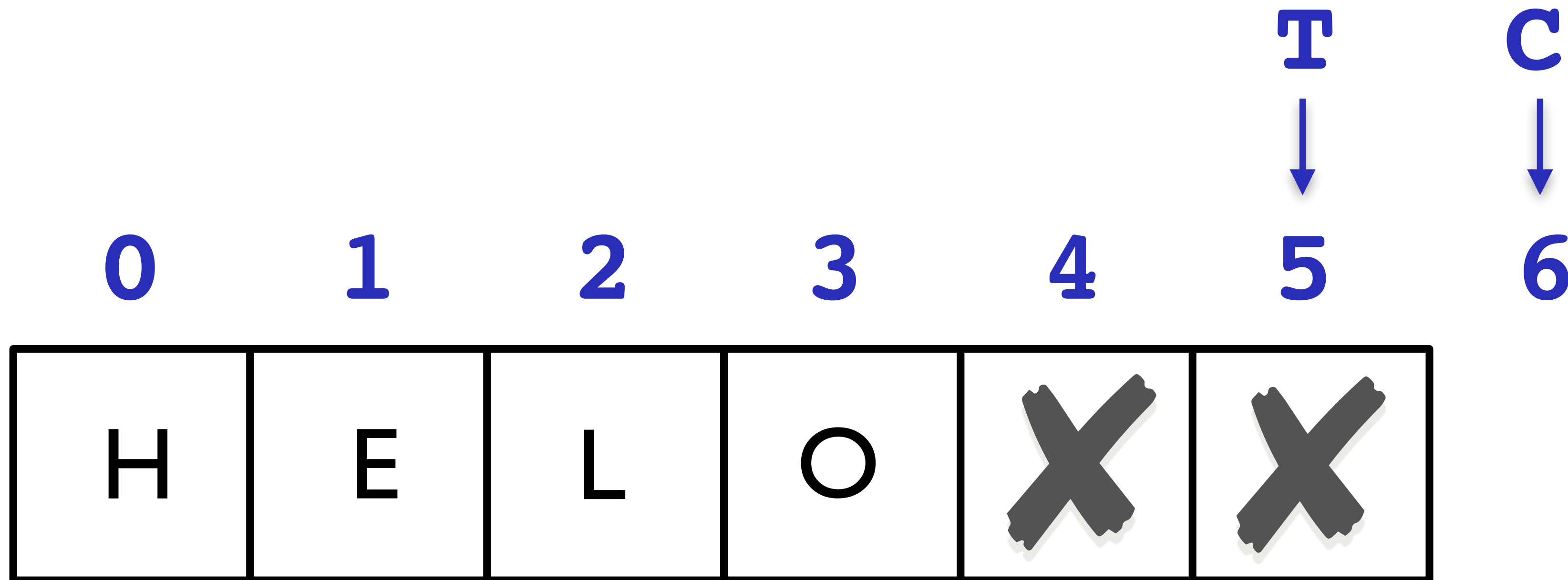
Supprimer en position i



↑
supprimer(2)

- Détruire un élément
- Déplacer les suivants
- Modifier la taille

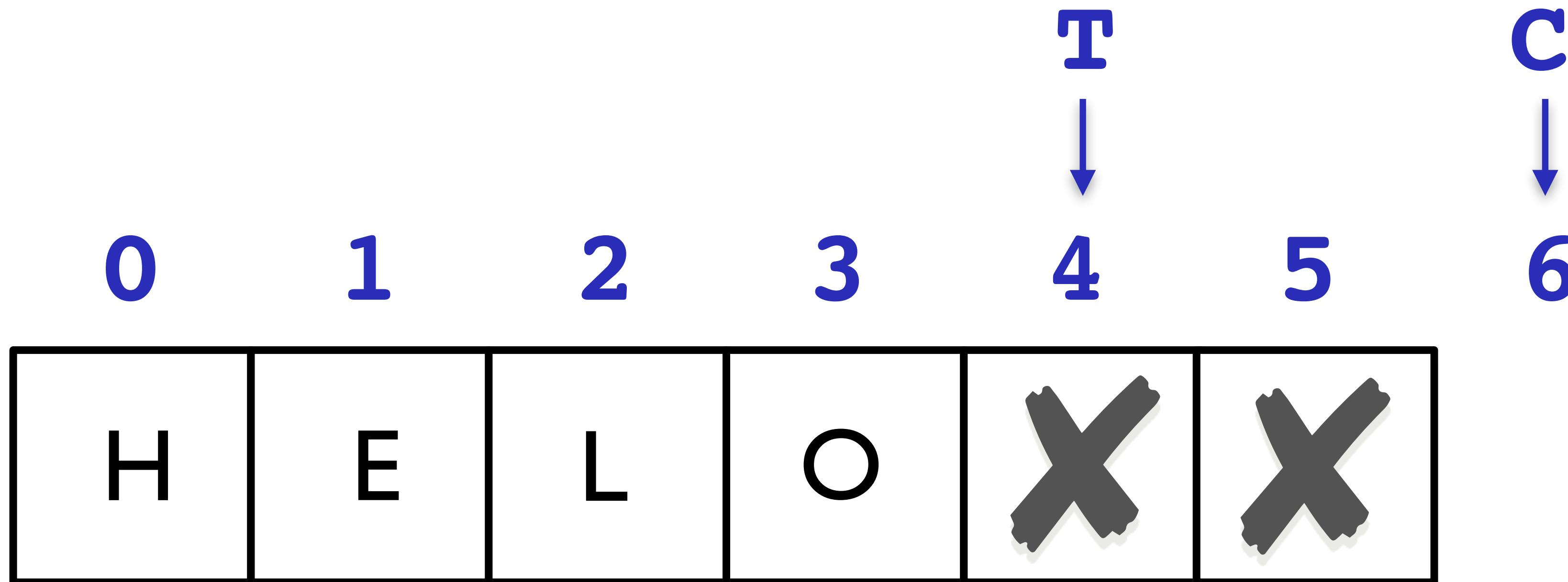
Supprimer en position i



↑
supprimer(2)

- Détruire un élément
- Déplacer les suivants
- Modifier la taille

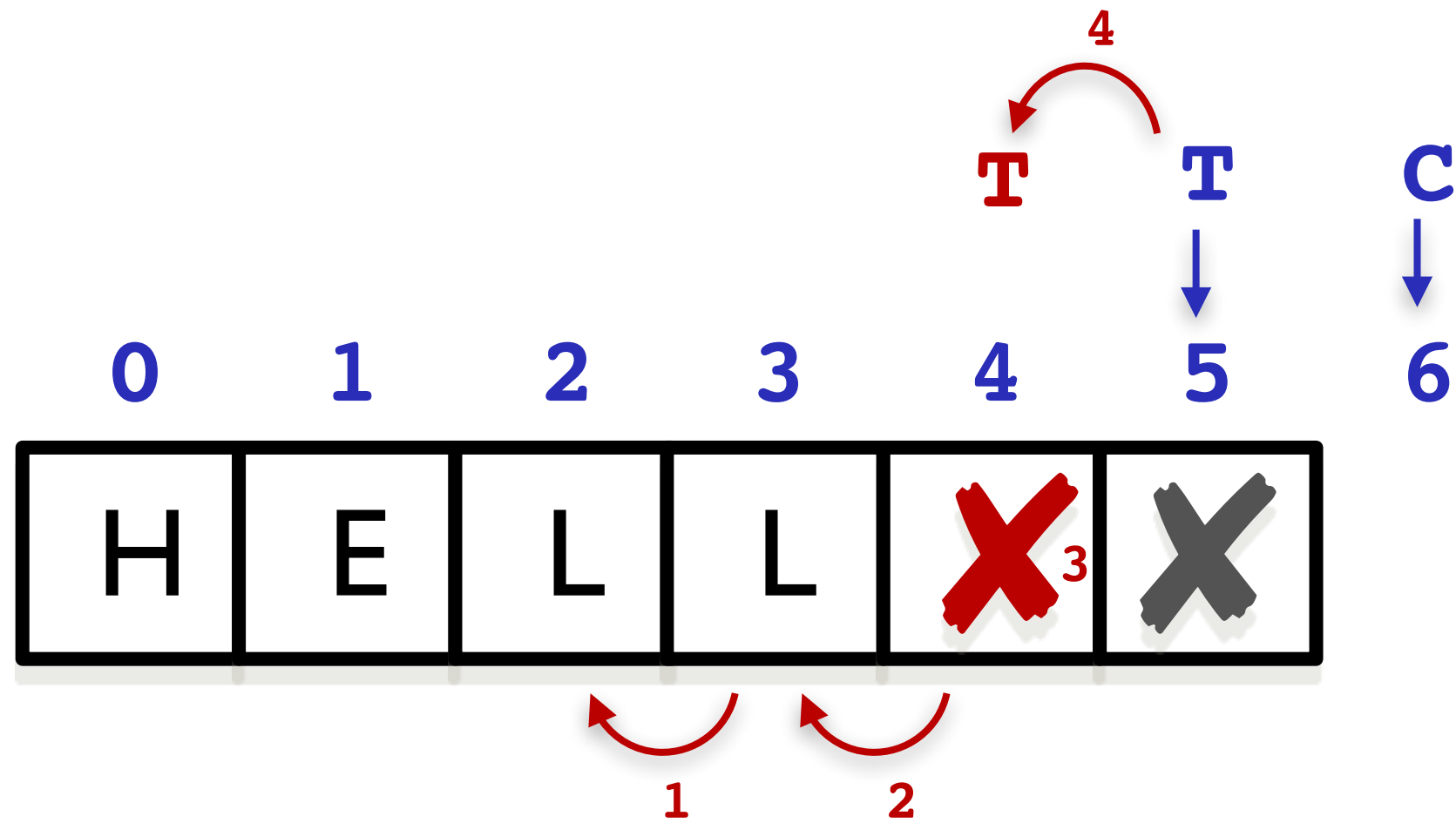
Supprimer en position i



↑
supprimer(2)

- Détruire un élément
- Déplacer les suivants
- Modifier la taille

Supprimer en position i



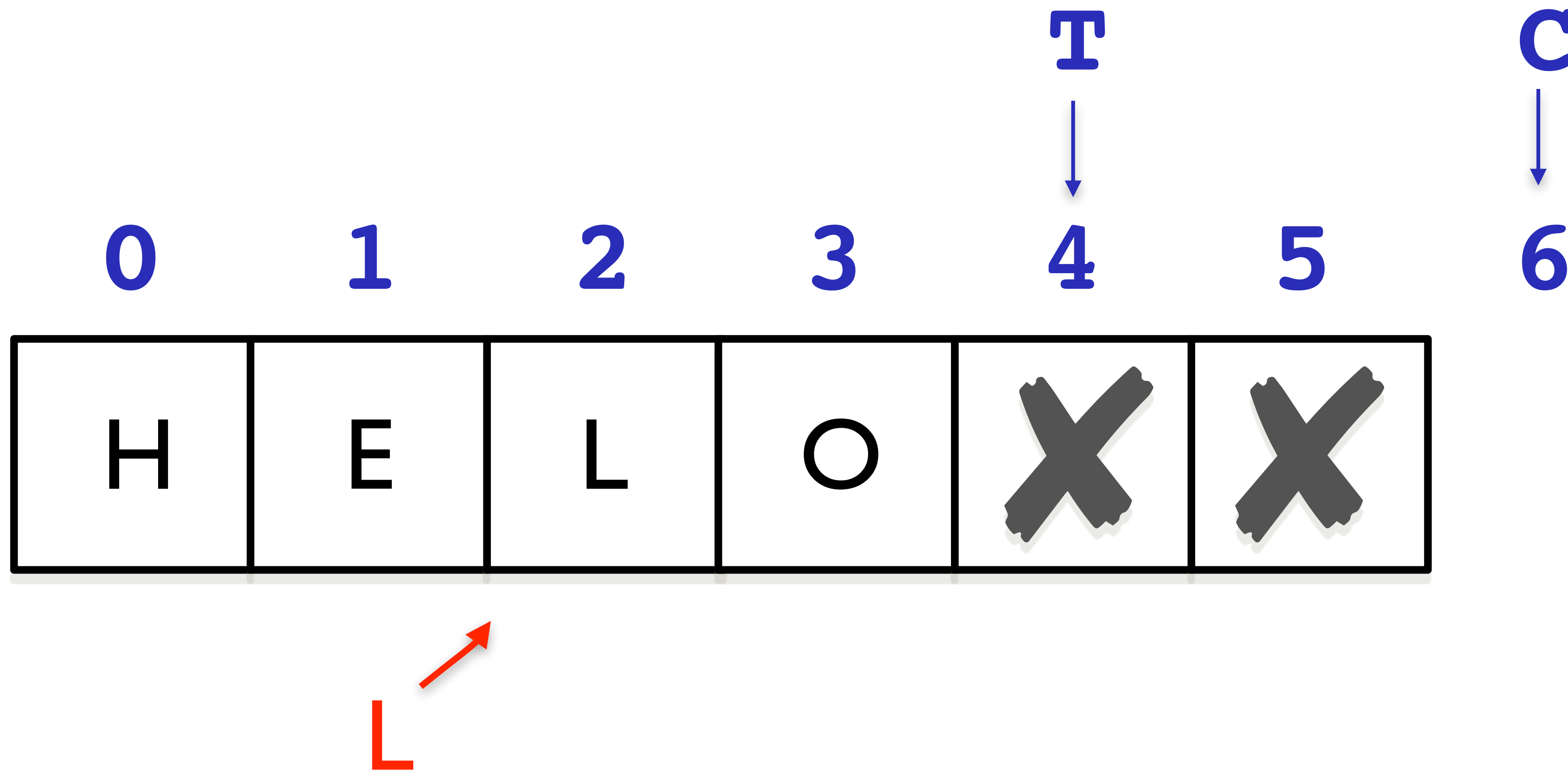
- Complexité dominée par les déplacements
- Linéaire en $O(\text{taille}-i)$

```
méthode supprimer(i)
    si i < 0 ou i >= taille
        alerter indice non valable

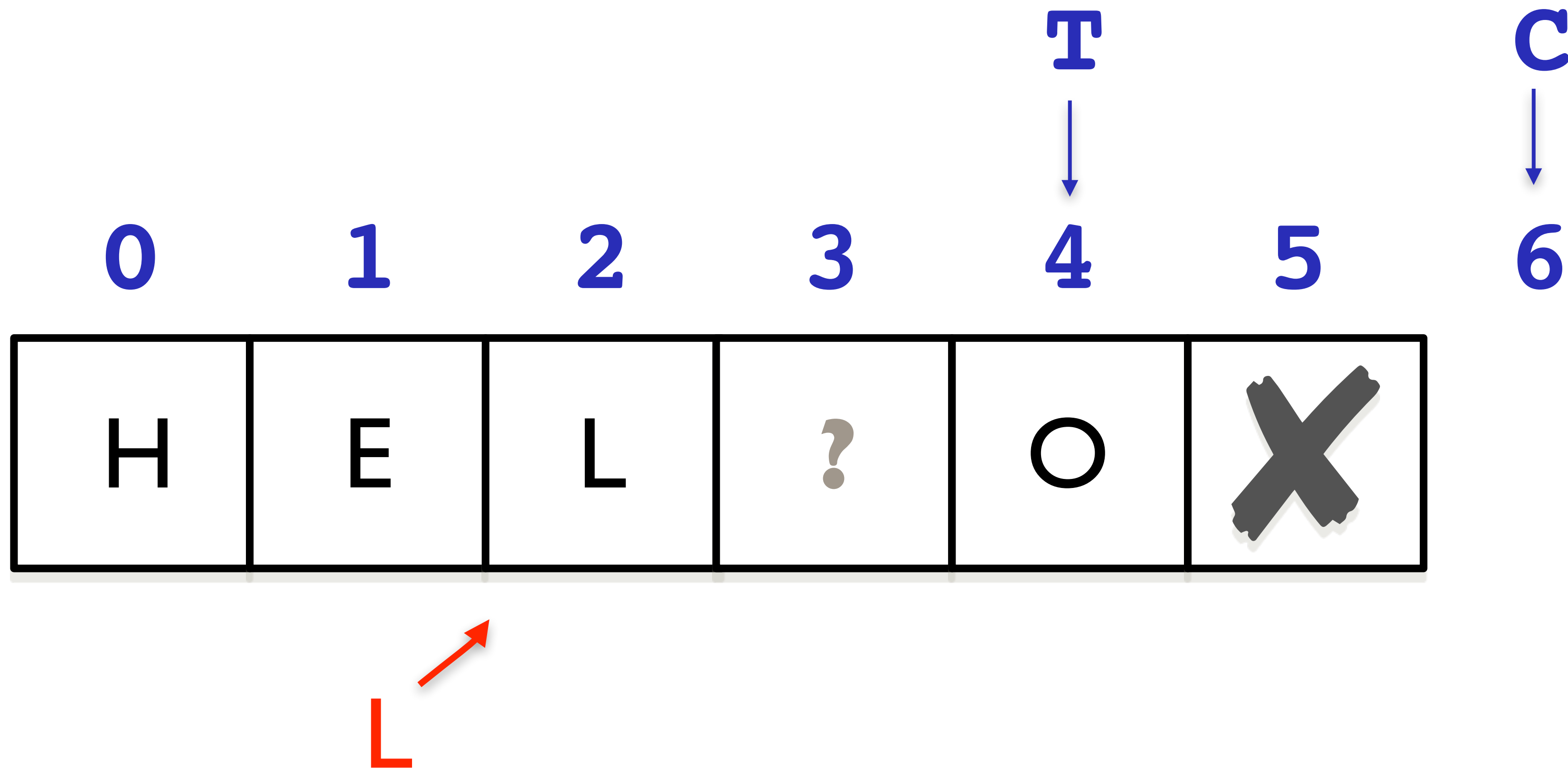
    pour j de i+1 à taille-1
        data[j-1] ← déplacer data[j]

    détruire data[taille-1]
    taille ← taille - 1
```

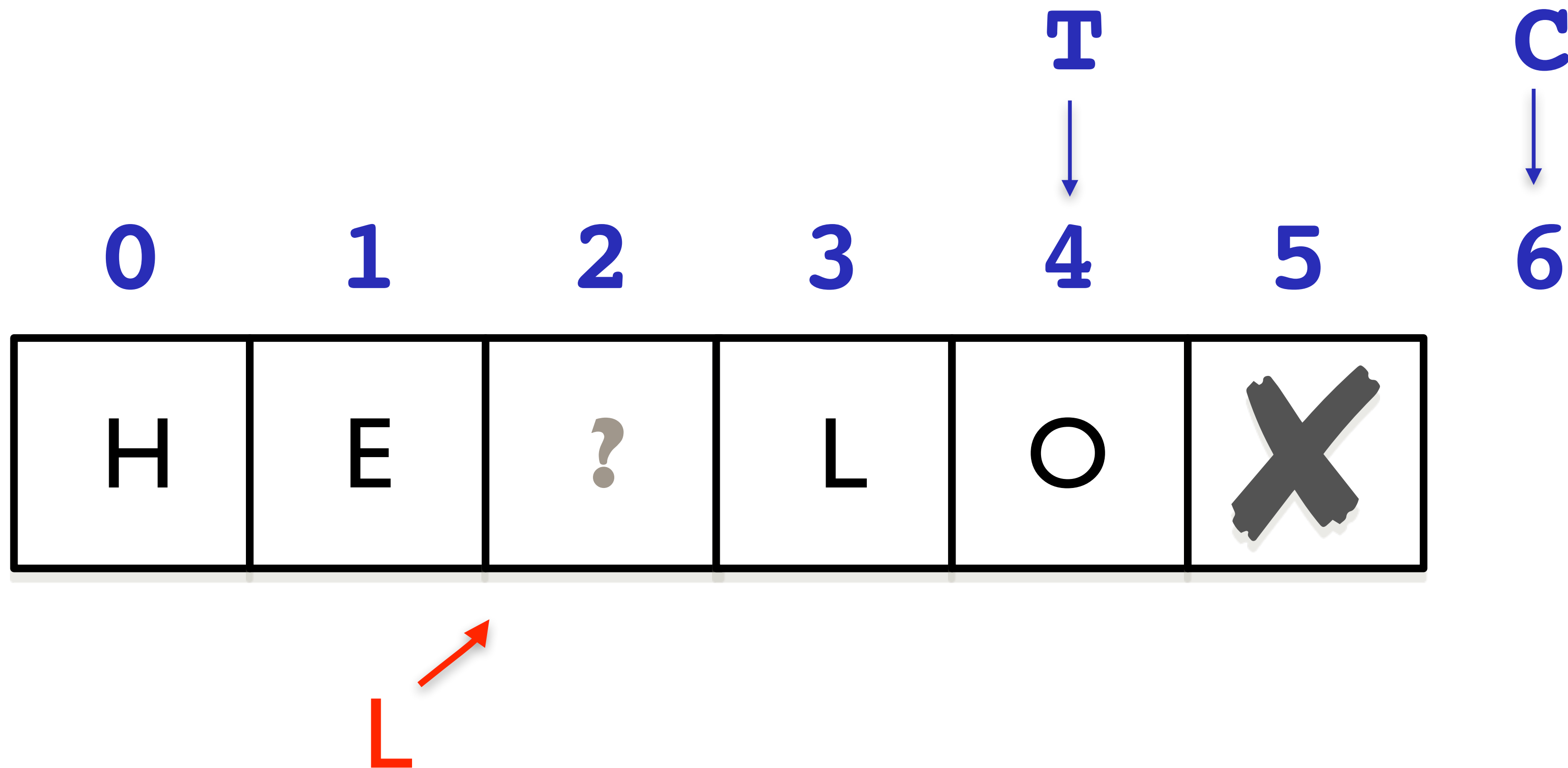
Insérer en position i



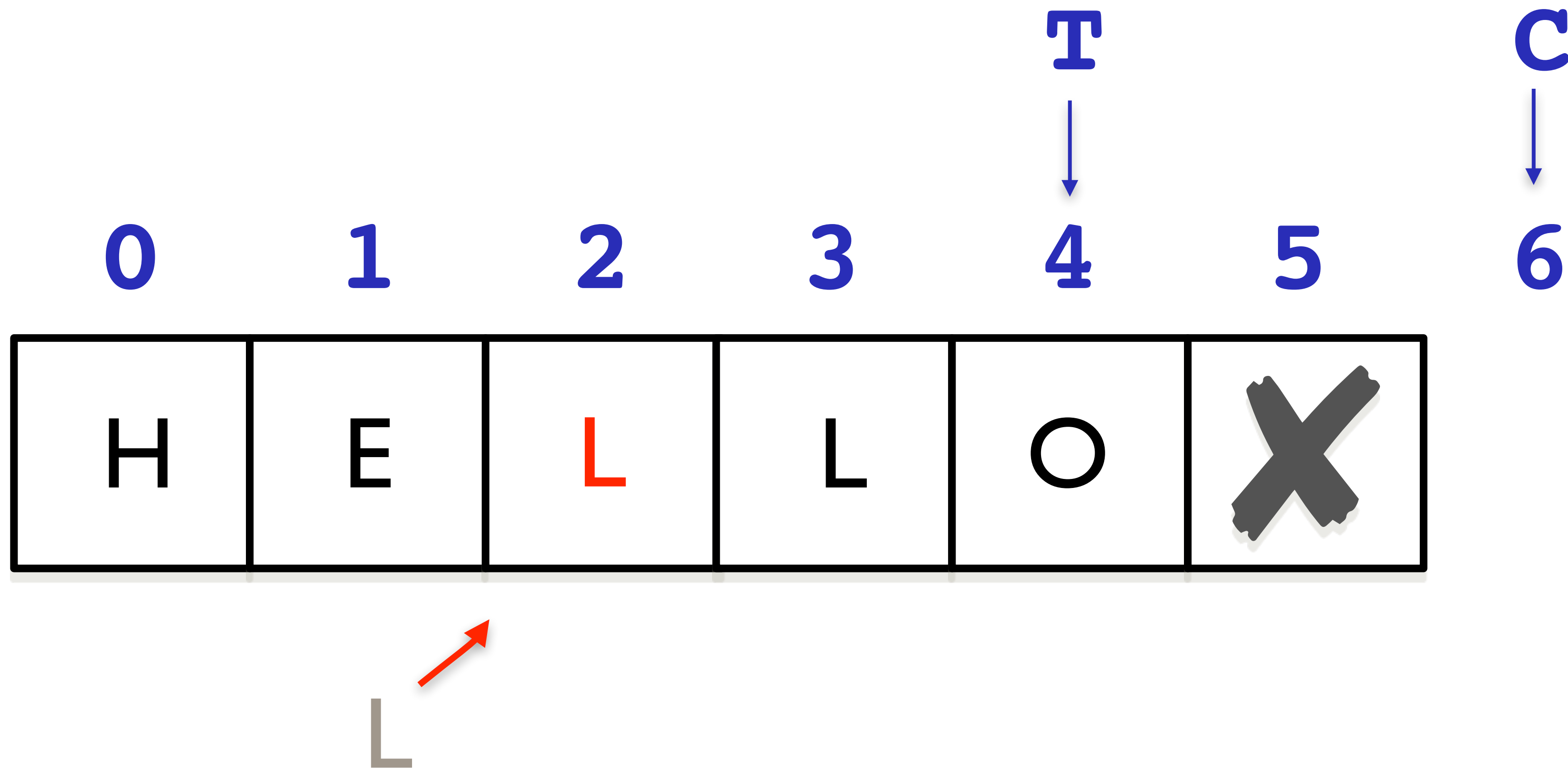
Insérer en position i



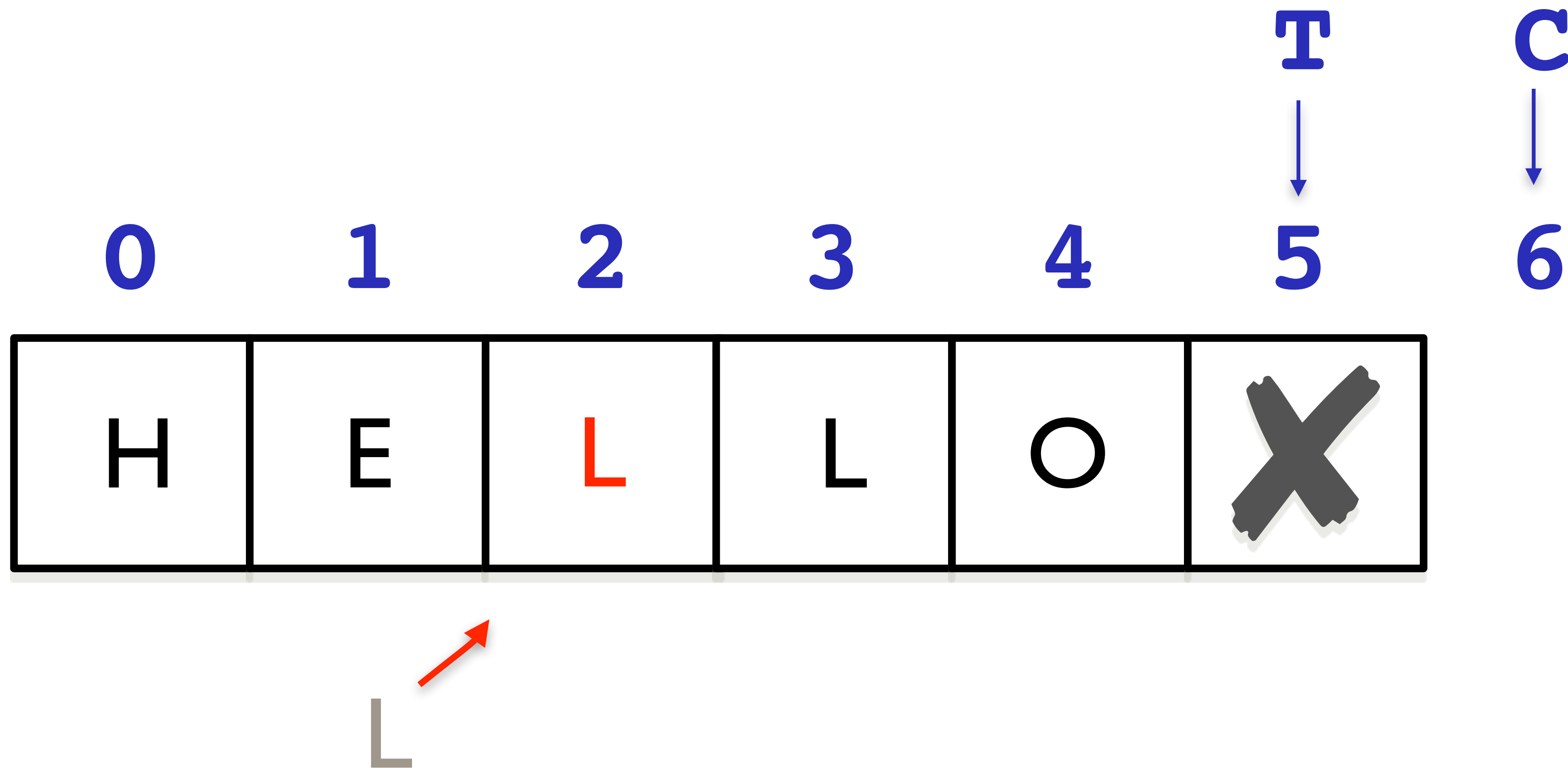
Insérer en position i



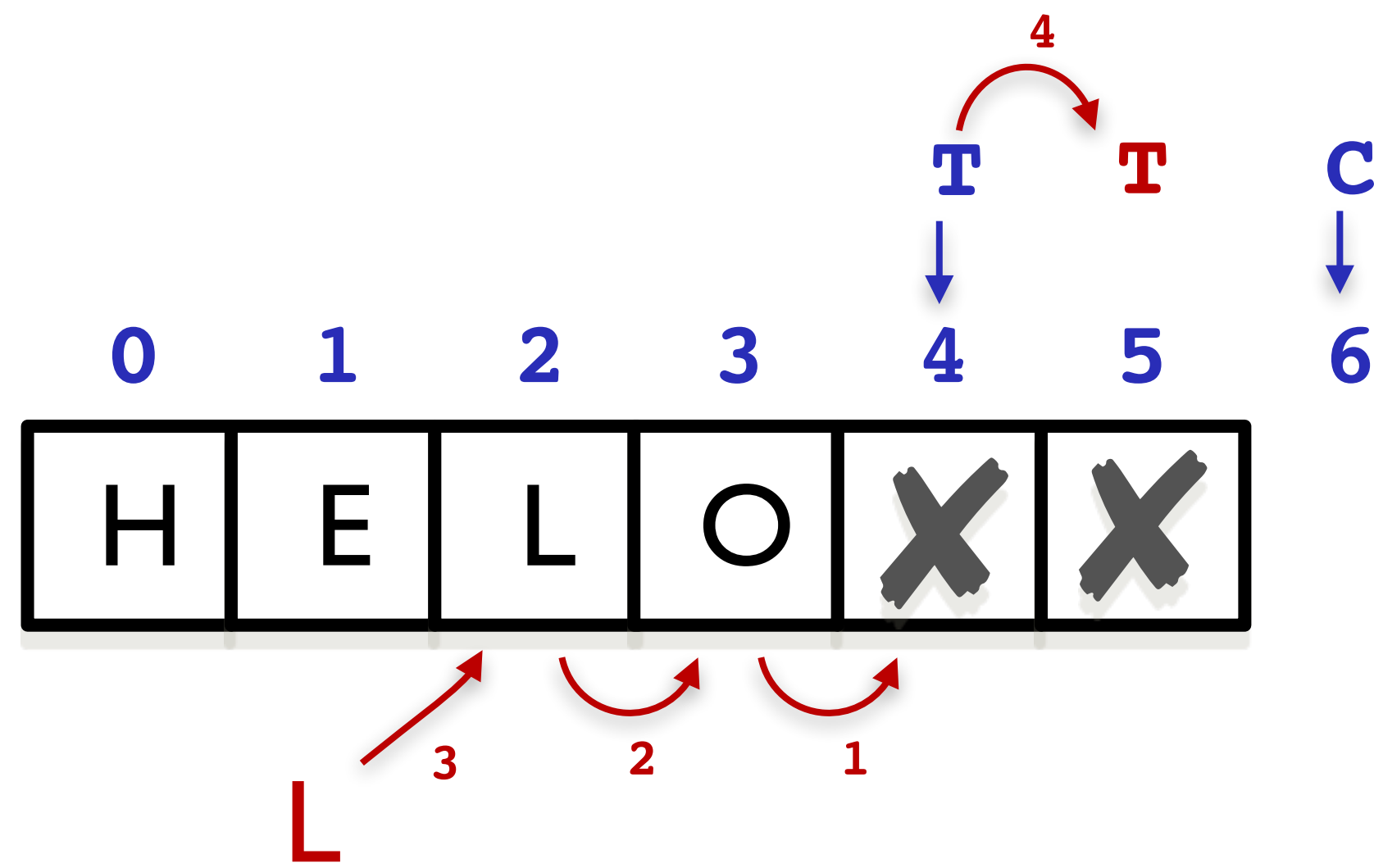
Insérer en position i



Insérer en position i



Insérer en position i



- Complexité dominée par les déplacements
- Linéaire en $O(\text{taille}-i)$

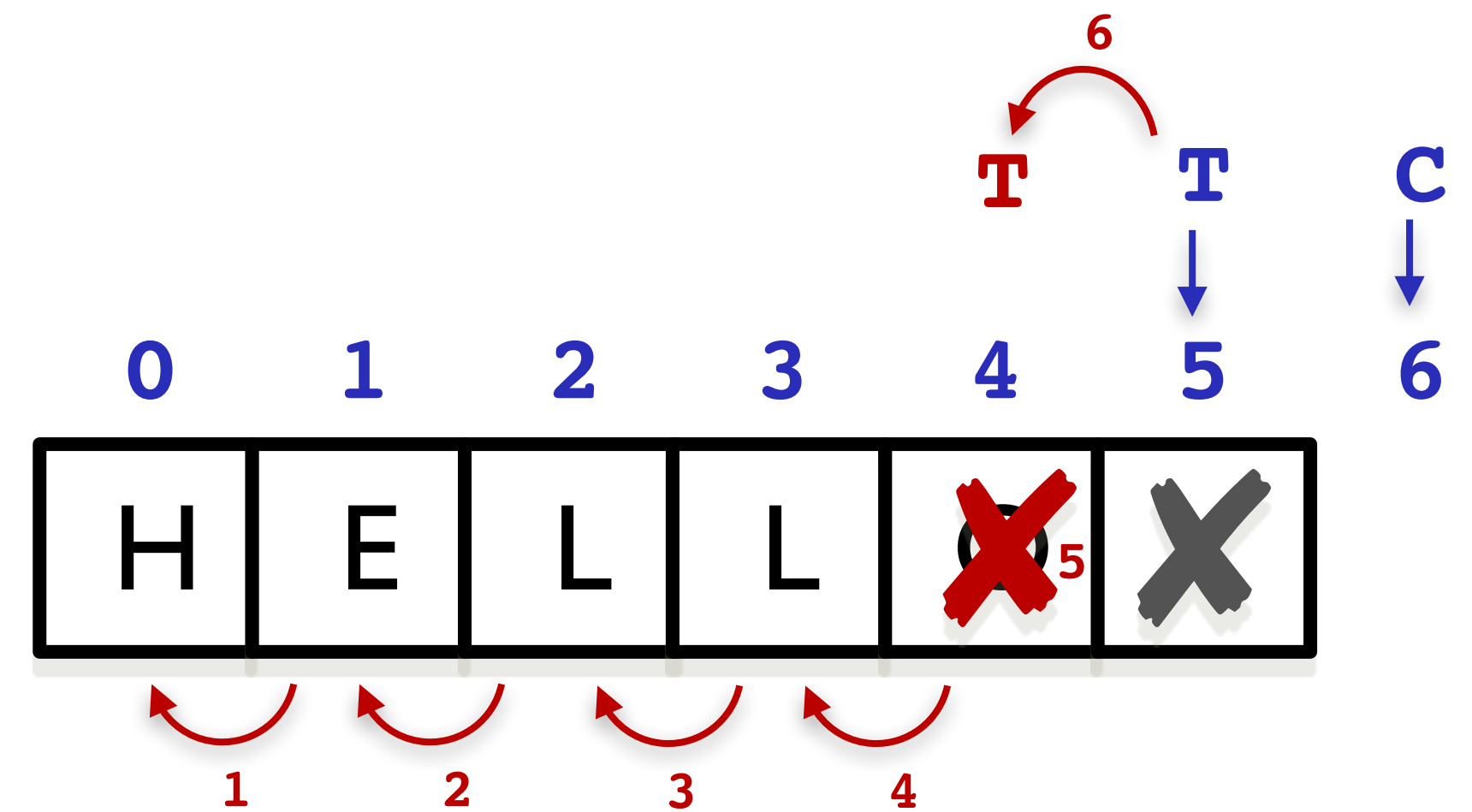
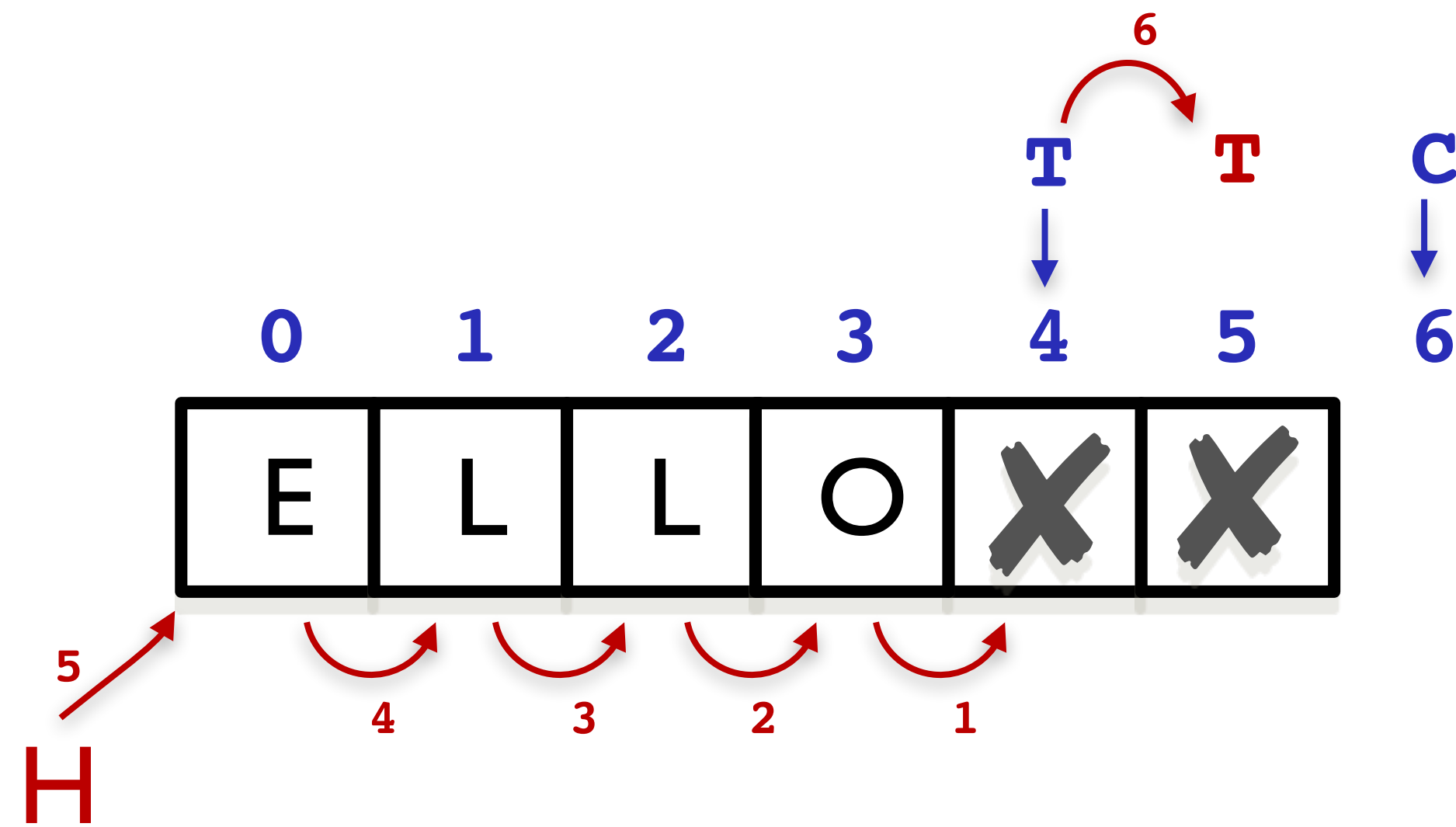
```

méthode insérer(i,v)
    si taille >= capacité,
        alerter tableau plein
    si i < 0 ou i > taille,
        alerter indice non valable

    si i == taille,
        construire data[taille] ← v
    sinon,
        construire data[taille]
            ← déplacer data[taille-1]
        pour j de taille-1 à i+1 par pas de -1
            data[j] ← déplacer data[j-1]
        data[i] ← v

    taille ← taille + 1
    
```

Insérer / supprimer au début



- Pas d'approche efficace
- Mêmes algorithmes que pour une position quelconque
- Linéaire en $O(\text{taille})$



Buffer circulaire

Motivation



- Tableau de taille variable (*indexé de 0 à $taille-1$*) est
 - efficace pour insérer / supprimer en fin
 - **mais** pas efficace pour insérer / supprimer ailleurs qu'en fin

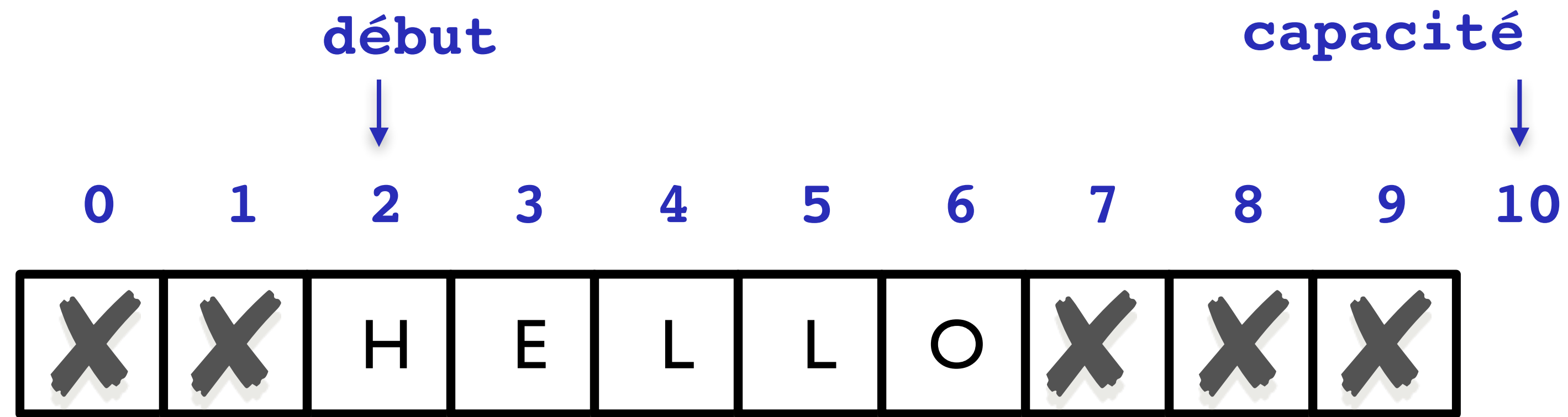
Motivation



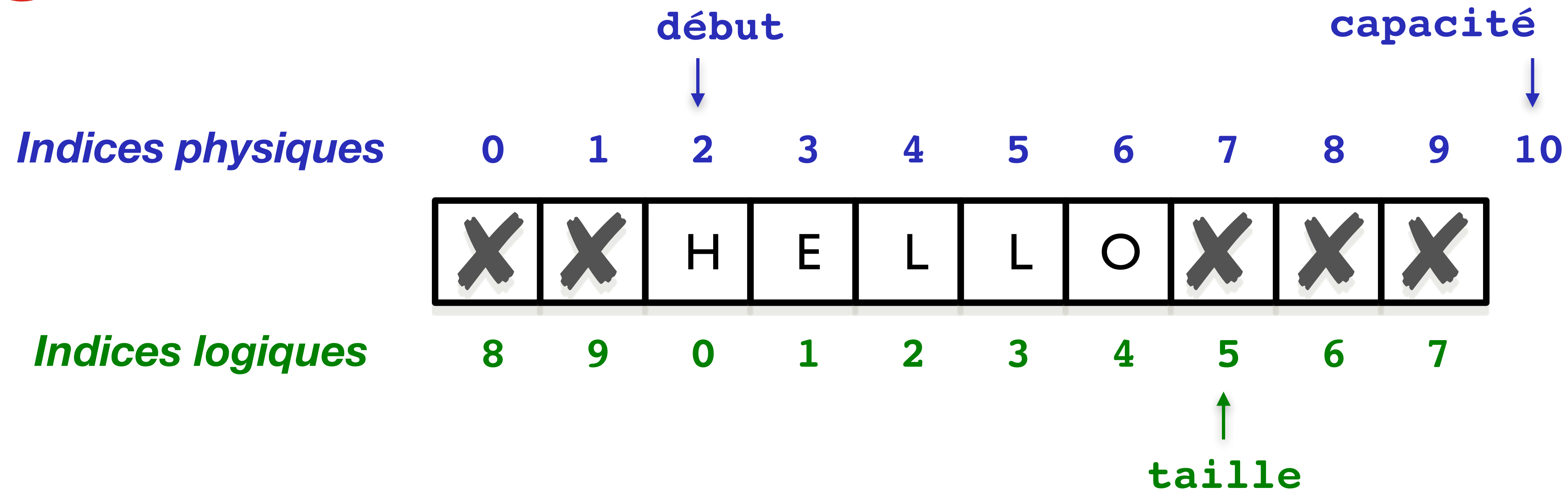
- Tableau de taille variable (*indexé de 0 à $taille-1$*) est
- efficace pour insérer / supprimer en fin
- **mais** pas efficace pour insérer / supprimer ailleurs qu'en fin

- Il existe la file FIFO 
- Comportement très utile.
- Requierd d'insérer au début, et de supprimer à la fin.
- Peut être mise en oeuvre avec un tableau :
 - Le premier élément n'est plus à l'index 0, mais à un index début.

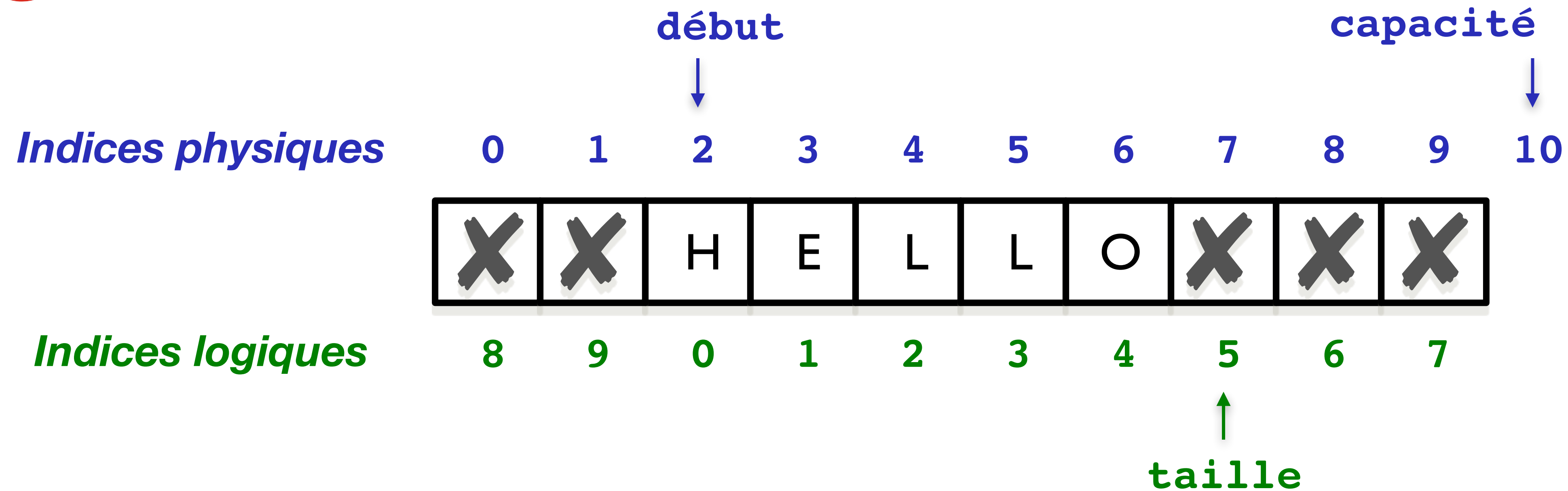
Buffer circulaire



Buffer circulaire



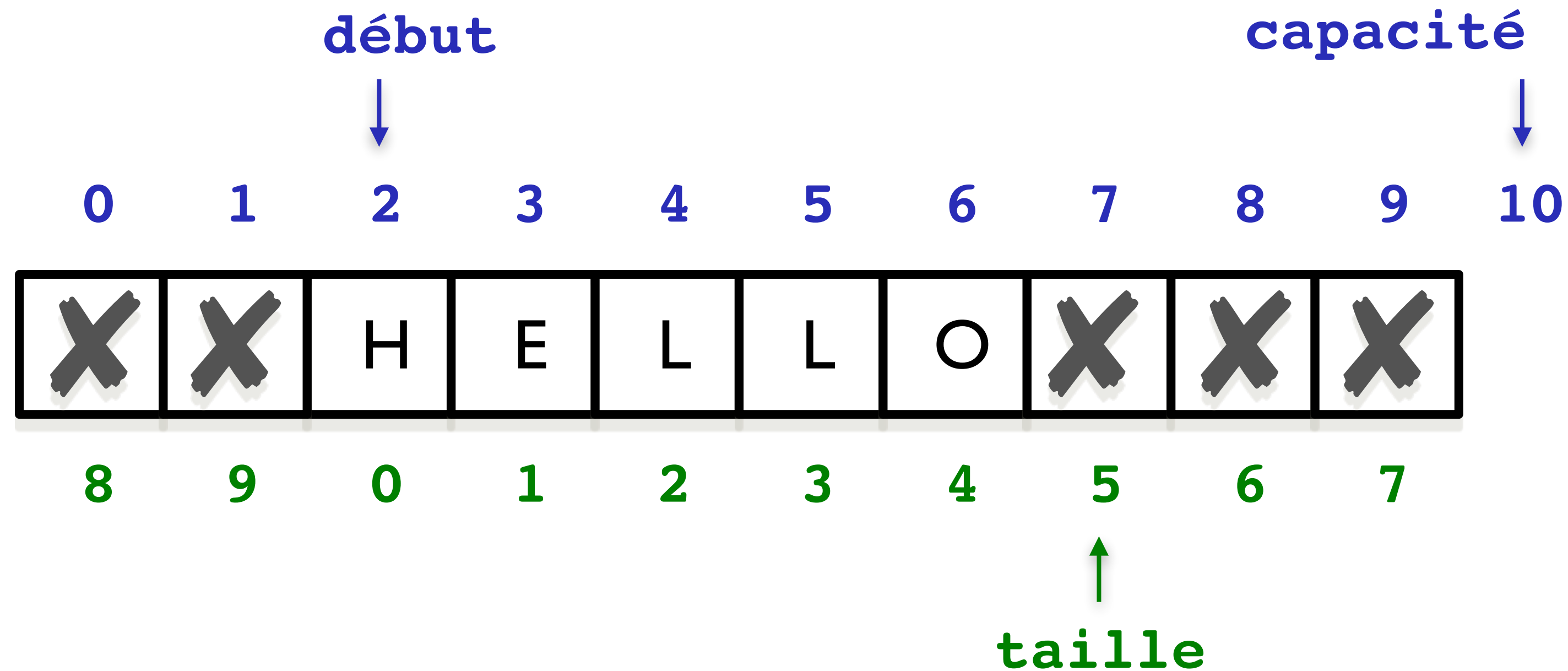
Buffer circulaire



```

classe BufferCirculaire
    Entier constant capacité
    Entier variable début
    Entier variable taille
    Element[] data[capacité]
  
```

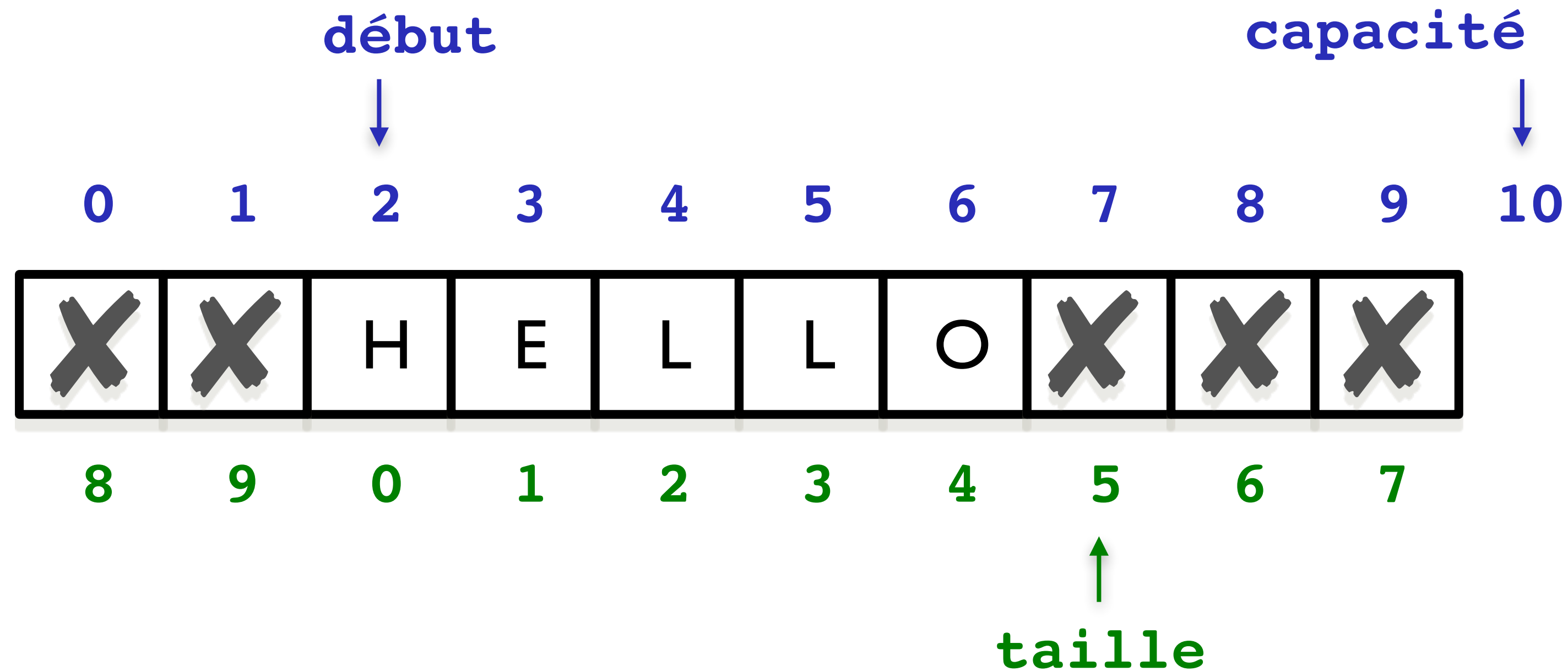
Indices physiques vs. Logiques



- Comment calculer l'indice **physique** en connaissant l'indice **logique** ?
- Ajouter **début**
- Retirer **capacité** si nécessaire

```
méthode i_physique(i_logique)
    retourner (début + i_logique) % capacité
```

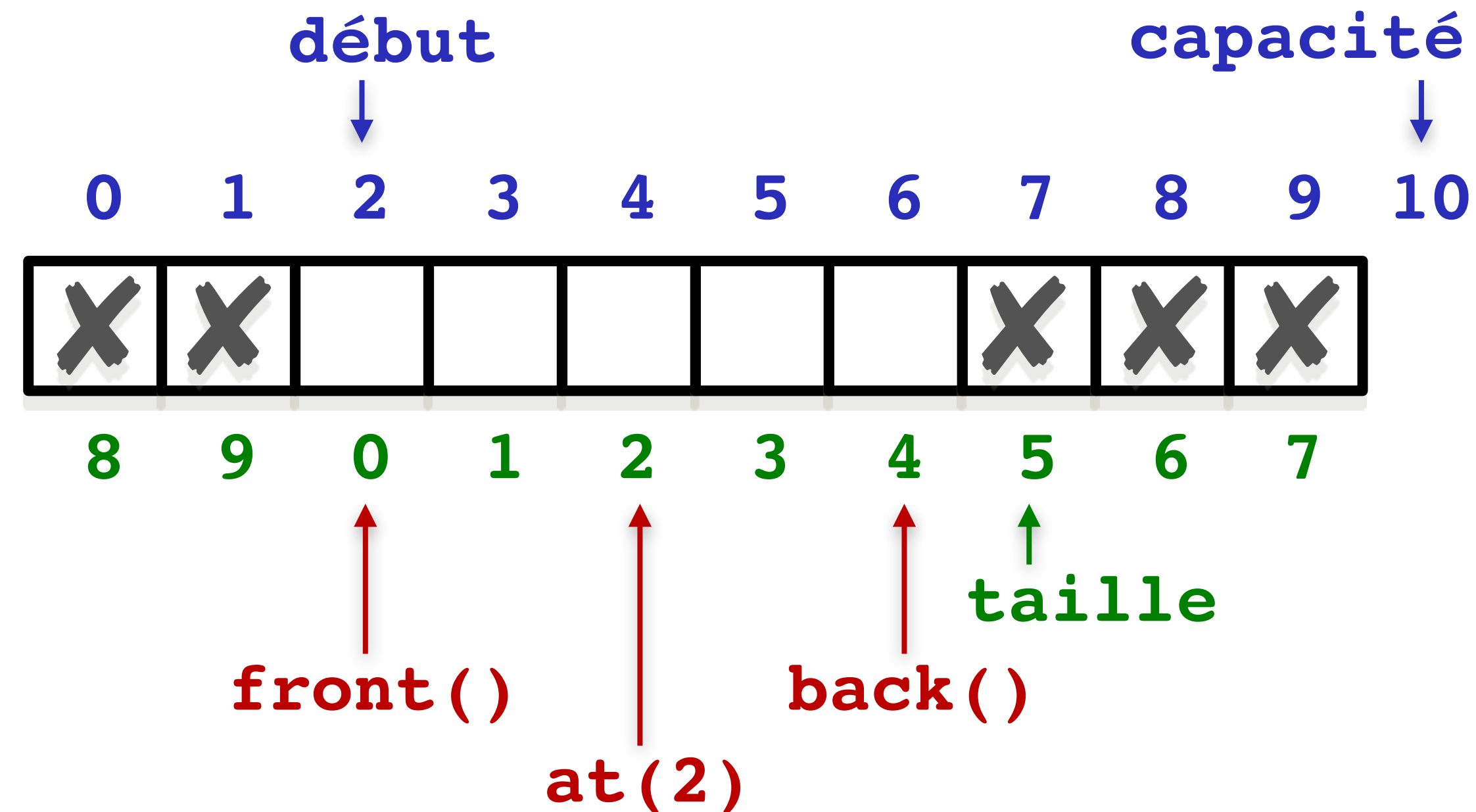
Indices physiques vs. Logiques



- Comment calculer l'indice **physique** en connaissant l'indice **logique** ?
- Ajouter **début**
- Retirer **capacité** si nécessaire

```
méthode i_physique(i_logique)
    retourner (début + i_logique + capacité)
              % capacité
```

Accéder à un élément



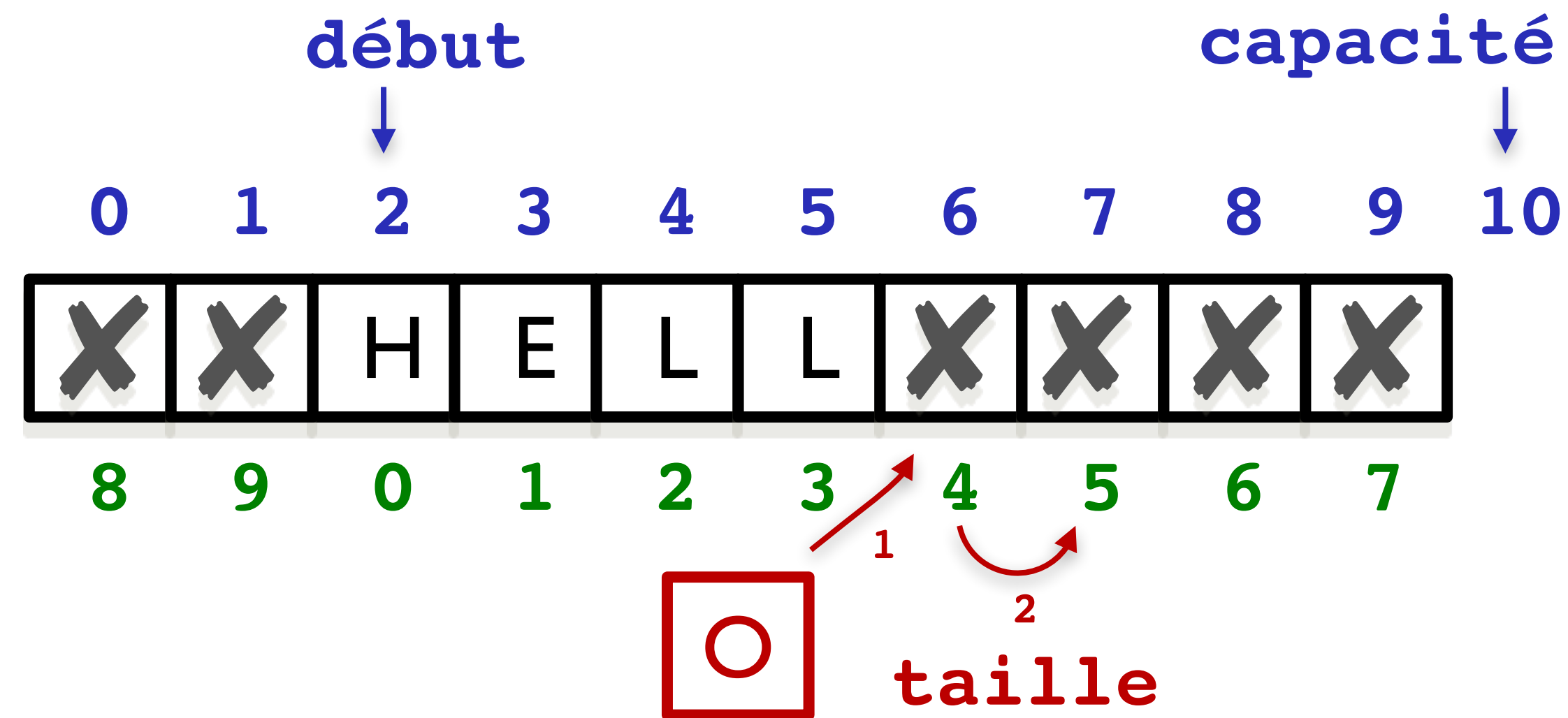
```
méthode at(i)
    si i < 0 ou i >= taille,
        alerter
    retourner data[i_physique(i)]
```

```
méthode back()
    retourner at(taille - 1)
```

```
méthode front()
    retourner at(0)
```



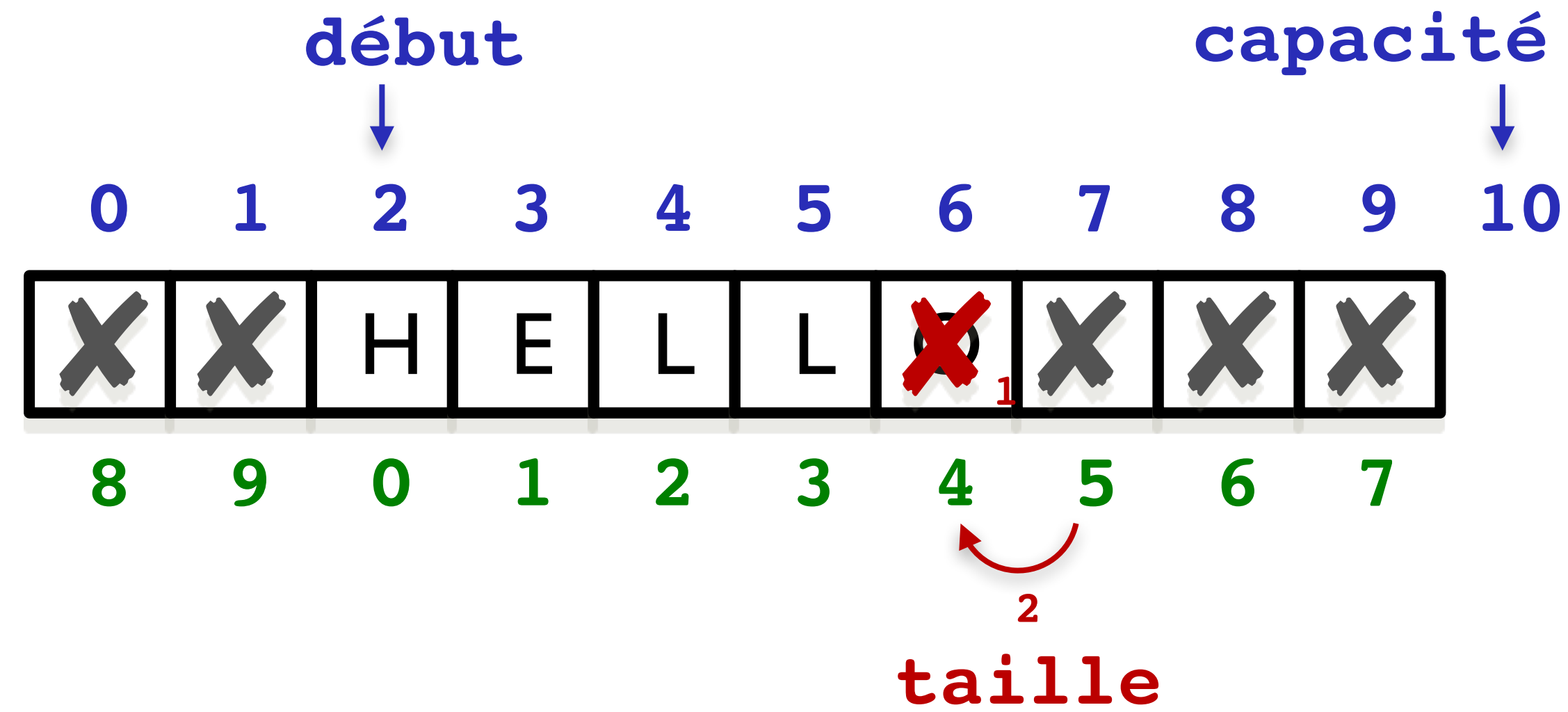
Insérer en fin



```
méthode insérer_en_fin(v)
    si taille >= capacité, alerter
    construire data[i_physique(taille)] ← v
    taille ← taille + 1
```



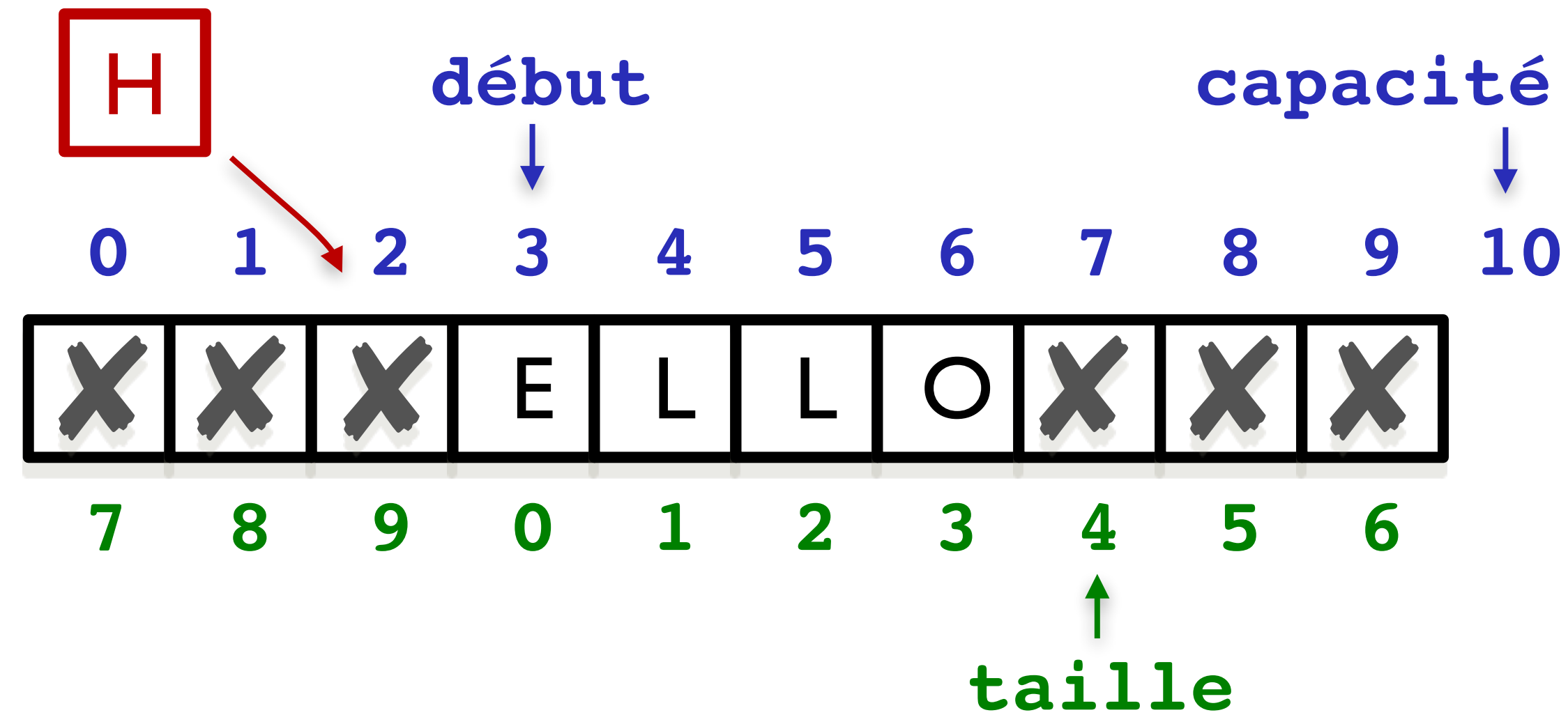
Supprimer en fin



```
méthode supprimer_en_fin()  
    si taille == 0, alerter  
    détruire data[i_physique(taille - 1)]  
    taille ← taille - 1
```



Insérer au début

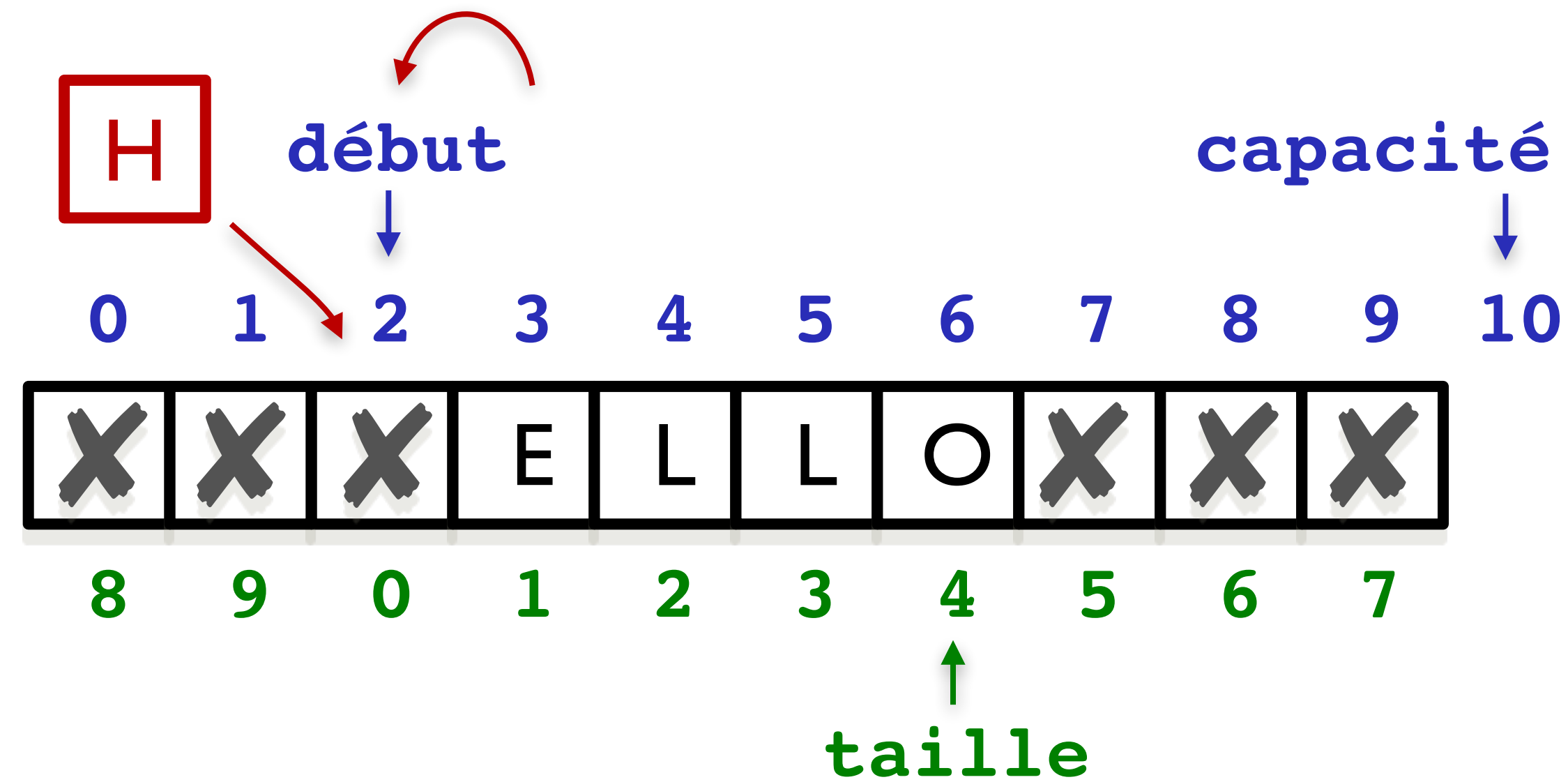


```

méthode insérer_au_début(v)
    si taille >= capacité, alerter
    avant ← i_physique(capacité - 1)
    construire data[avant] ← v
    
```



Insérer au début

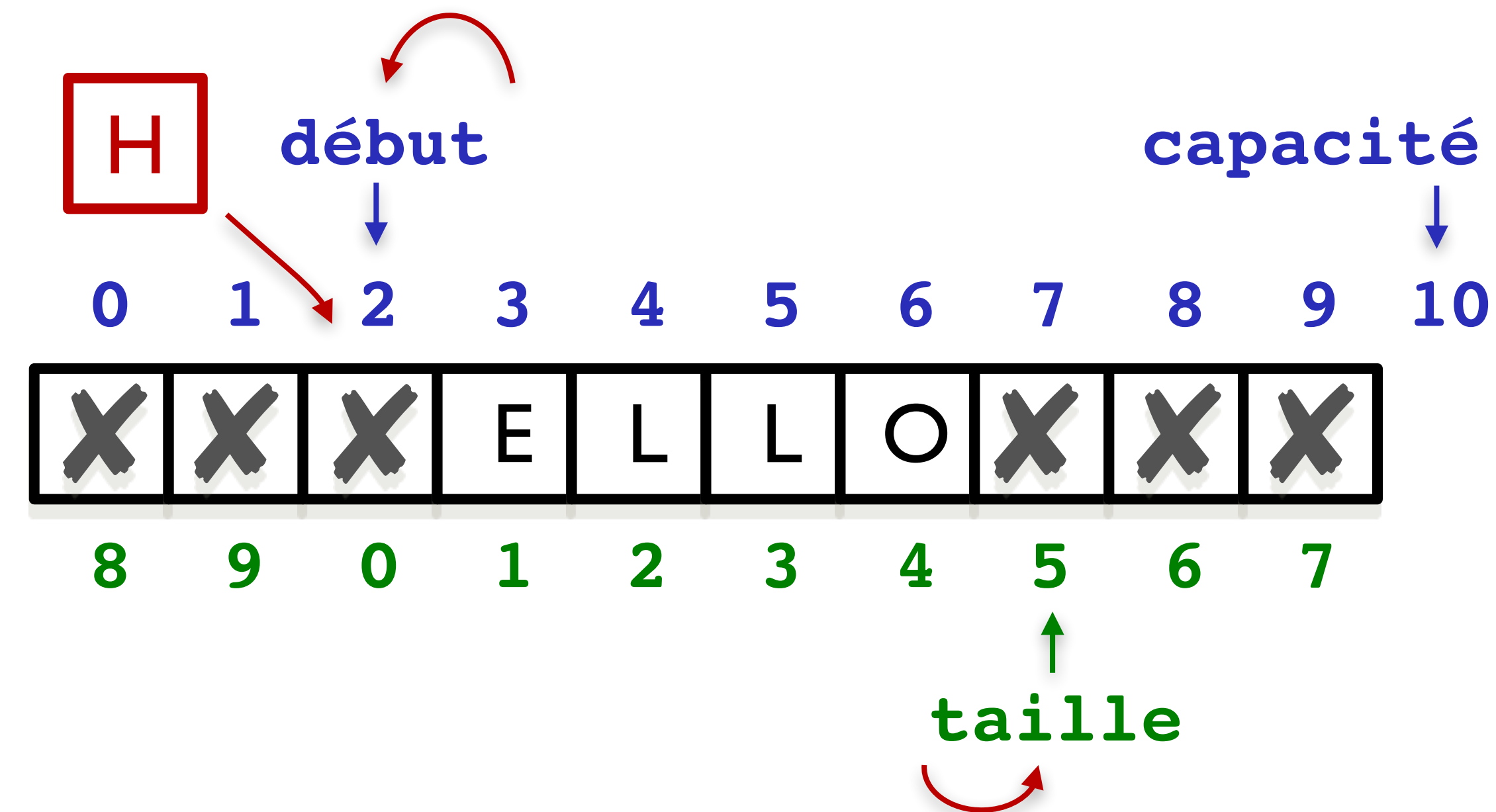


```

méthode insérer_au_début(v)
    si taille >= capacité, alerter
    avant ← i_physique(capacité - 1)
    construire data[avant] ← v
    début ← avant
    
```



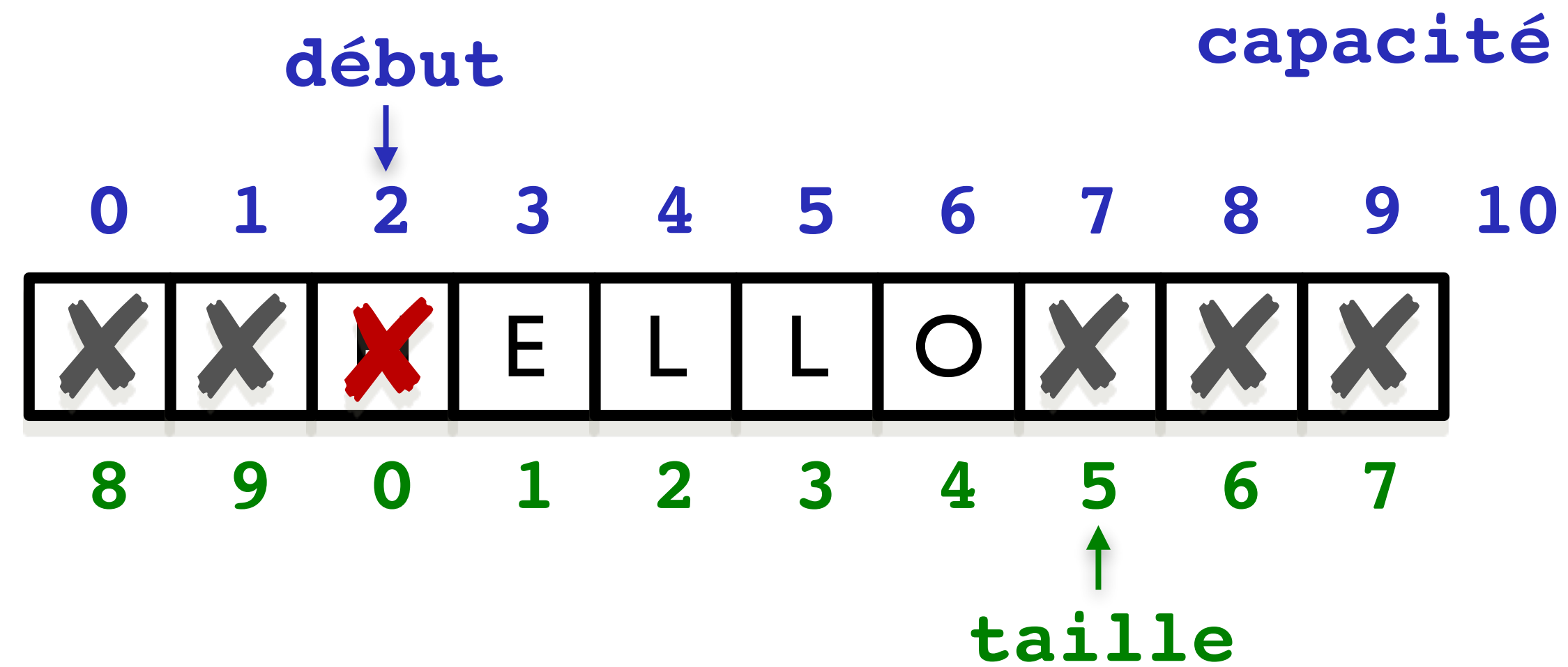
Insérer au début



```

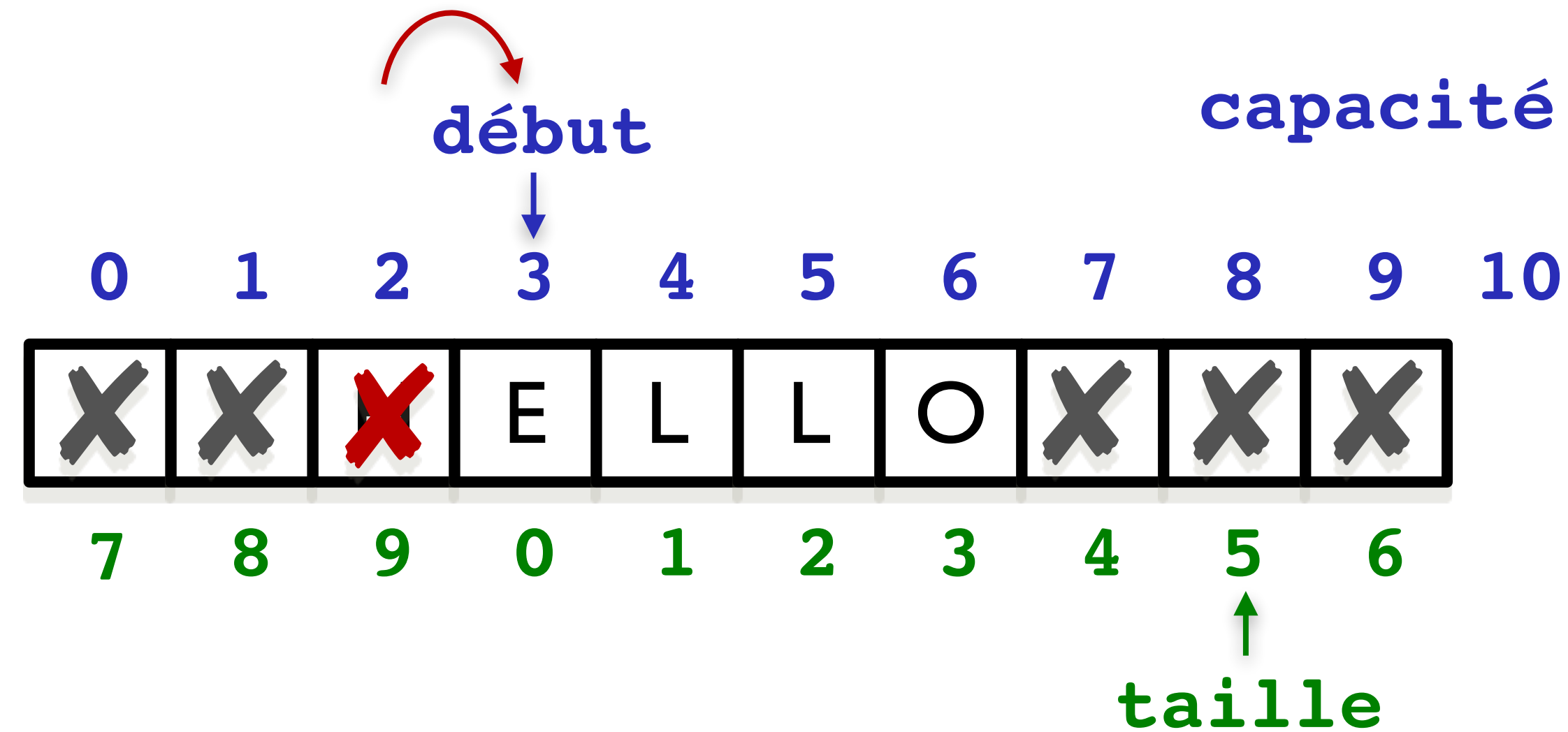
méthode insérer_au_début(v)
    si taille >= capacité, alerter
    avant ← i_physique(capacité - 1)
    construire data[avant] ← v
    début ← avant
    taille ← taille + 1
    
```

Supprimer au début



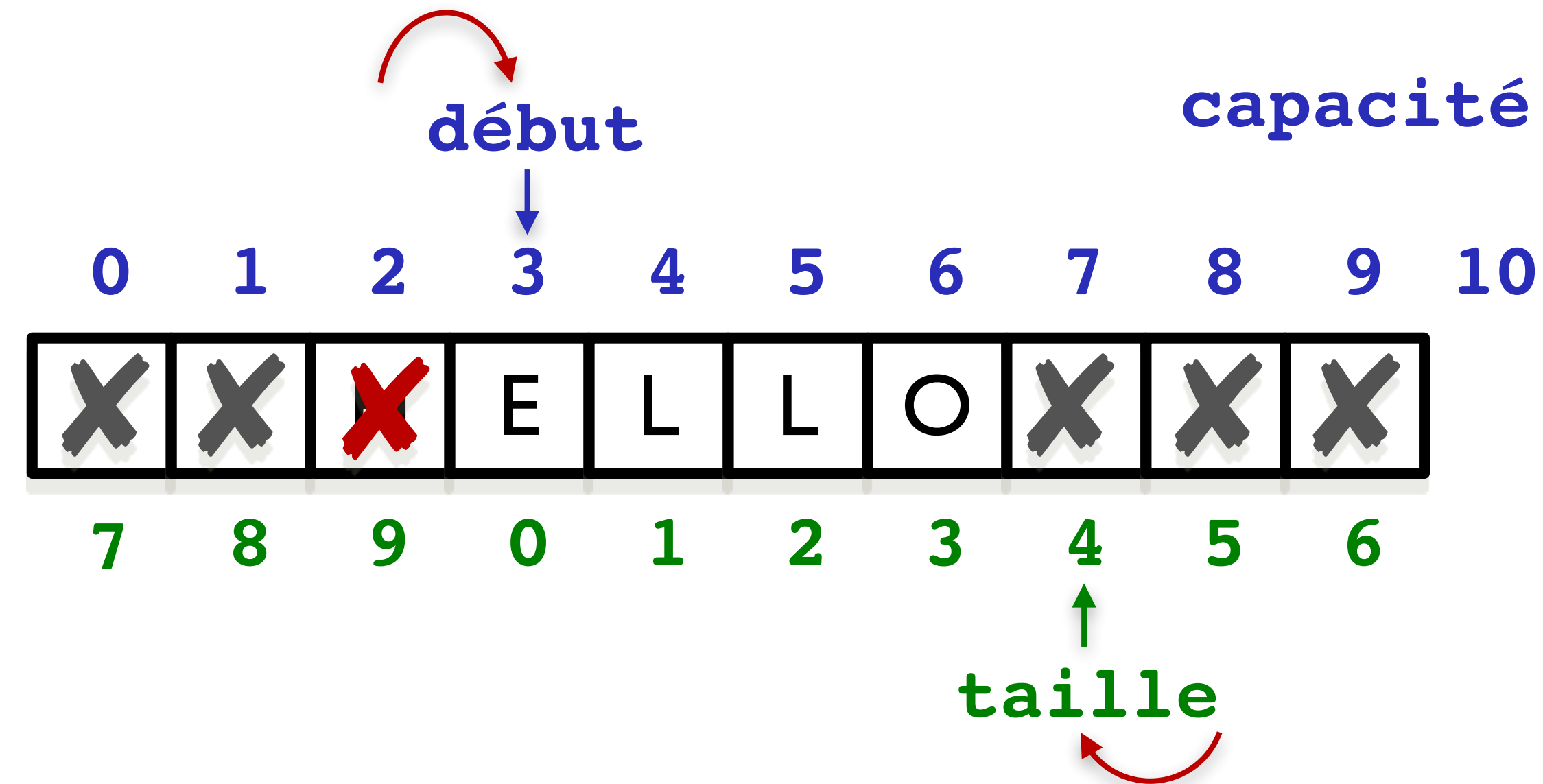
```
méthode supprimer_au_début()
    si taille == 0, alerter
    détruire data[i_physique(0)]
```

Supprimer au début



```
méthode supprimer_au_début()
    si taille == 0, alerter
    détruire data[i_physique(0)]
    début ← i_physique(1)
```

Supprimer au début



```
méthode supprimer_au_début()
    si taille == 0, alerter
    détruire data[i_physique(0)]
    début ← i_physique(1)
    taille ← taille - 1
```

Avantages et inconvénients



+

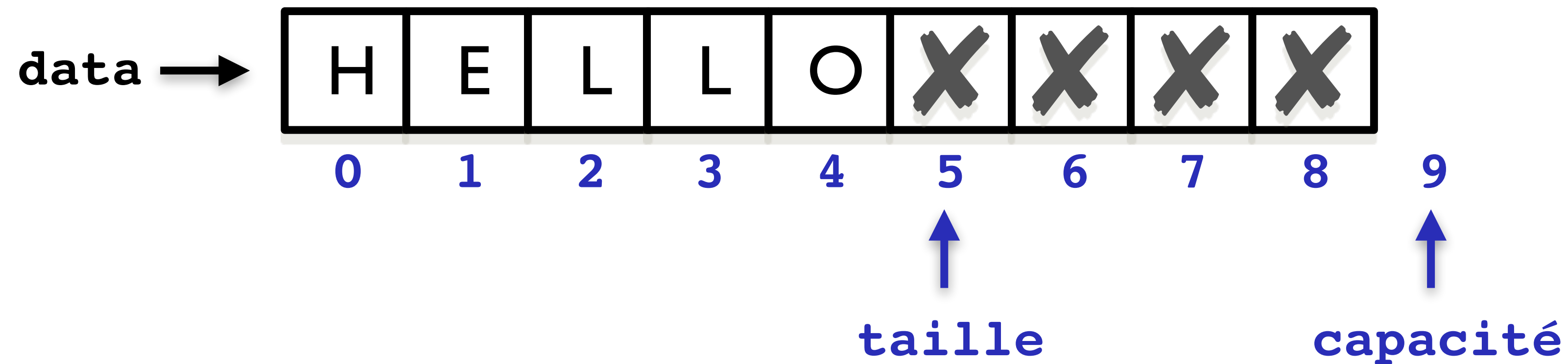
- $O(1)$ pour
 - insérer en début ou en fin
 - supprimer en début ou en fin
 - accéder en toute position

-

- Insérer ou supprimer à l'indice i en $O(\min(i, \text{taille} - i))$
- Capacité fixe



Tableaux de capacité *variable*



```

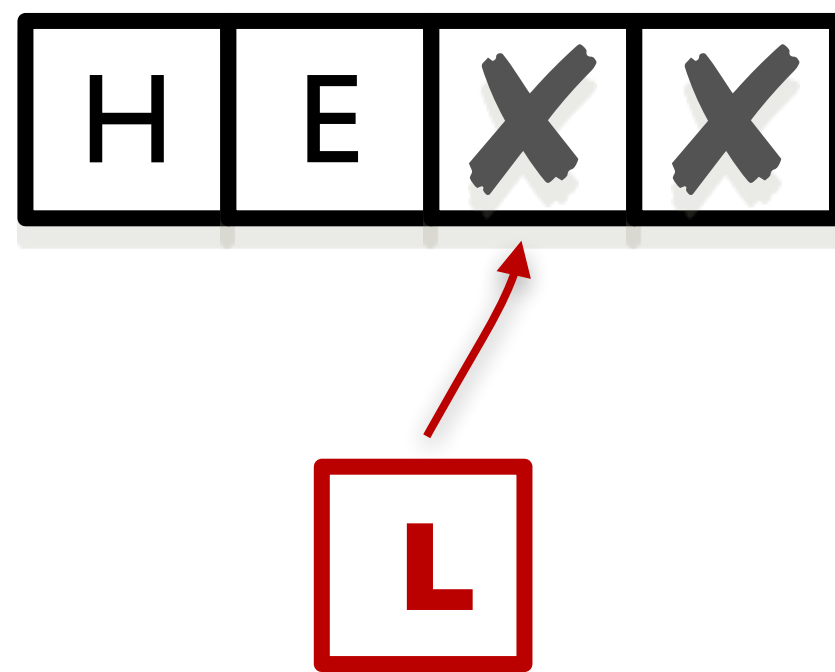
classe TableauCapacitéVariable
    Entier variable capacité
    Entier variable taille
    Element* data

```

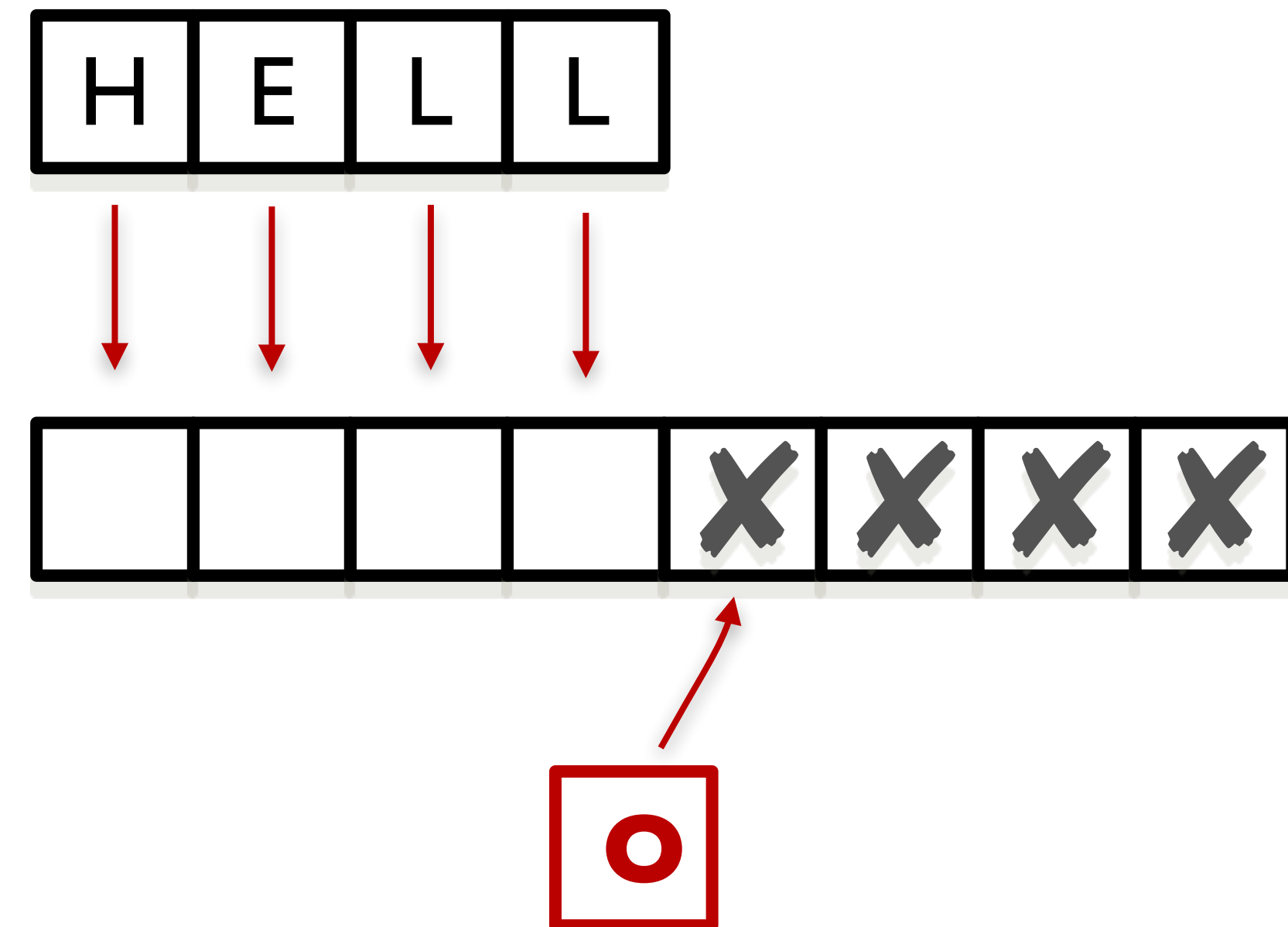
Insertion en fin



- Capacité suffisante



- Capacité à augmenter



Insertion en fin



```
méthode insérer_en_fin(v)
```

H	E	L	L
---	---	---	---

Insertion en fin



```
méthode insérer_en_fin(v)
  si taille >= capacité
    tmp ← allouer nouvelle_capacité()
```

H	E	L	L
---	---	---	---

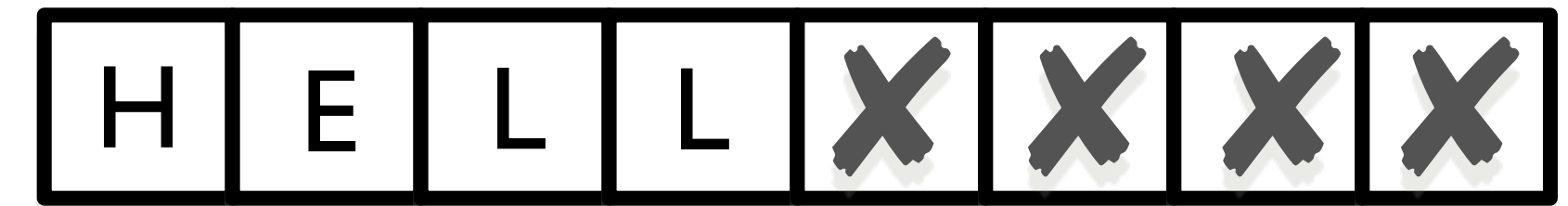
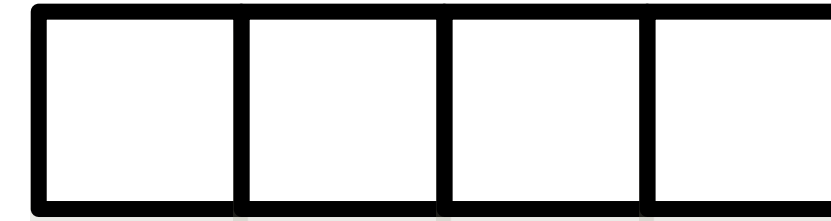
X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---

```
méthode nouvelle_capacité()
  si capacité == 0,
    retourner 1
  sinon,
    retourner 2 * capacité
```

Insertion en fin



```
méthode insérer_en_fin(v)
  si taille >= capacité
    tmp ← allouer nouvelle_capacité()
    pour i de 0 à taille - 1
      construire tmp[i] ← déplacer data[i]
```

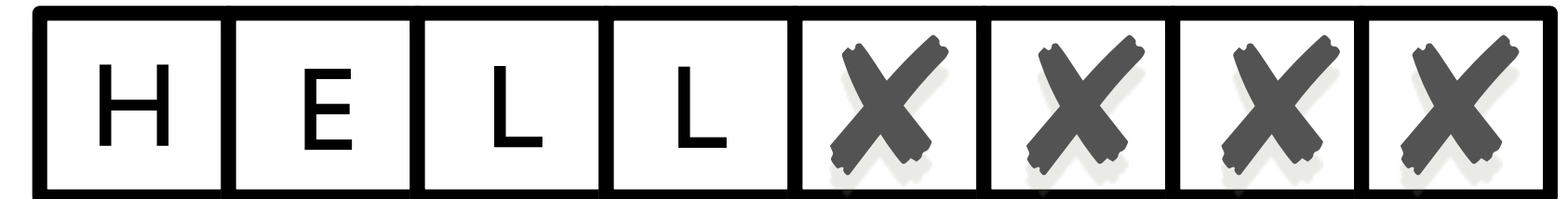
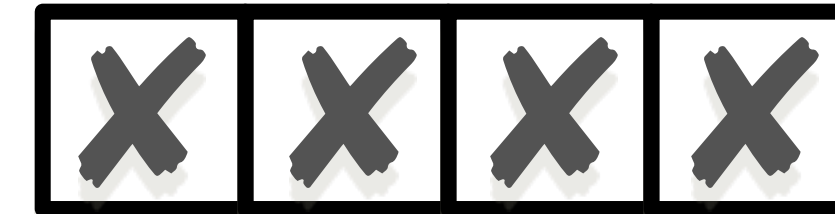


```
méthode nouvelle_capacité()
  si capacité == 0,
    retourner 1
  sinon,
    retourner 2 * capacité
```

Insertion en fin



```
méthode insérer_en_fin(v)
  si taille >= capacité
    tmp ← allouer nouvelle_capacité()
    pour i de 0 à taille - 1
      construire tmp[i] ← déplacer data[i]
    pour i de 0 à taille - 1
      détruire data[i]
```

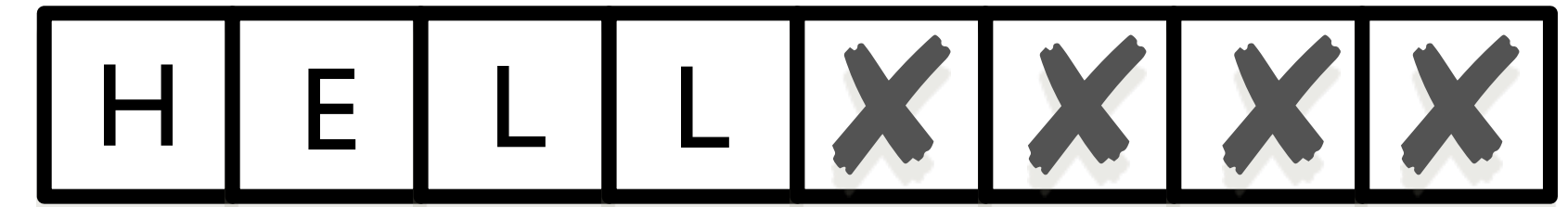


```
méthode nouvelle_capacité()
  si capacité == 0,
    retourner 1
  sinon,
    retourner 2 * capacité
```

Insertion en fin



```
méthode insérer_en_fin(v)
  si taille >= capacité
    tmp ← allouer nouvelle_capacité()
    pour i de 0 à taille - 1
      construire tmp[i] ← déplacer data[i]
    pour i de 0 à taille - 1
      détruire data[i]
    capacité ← nouvelle_capacité()
  libérer data
  data ← tmp
```



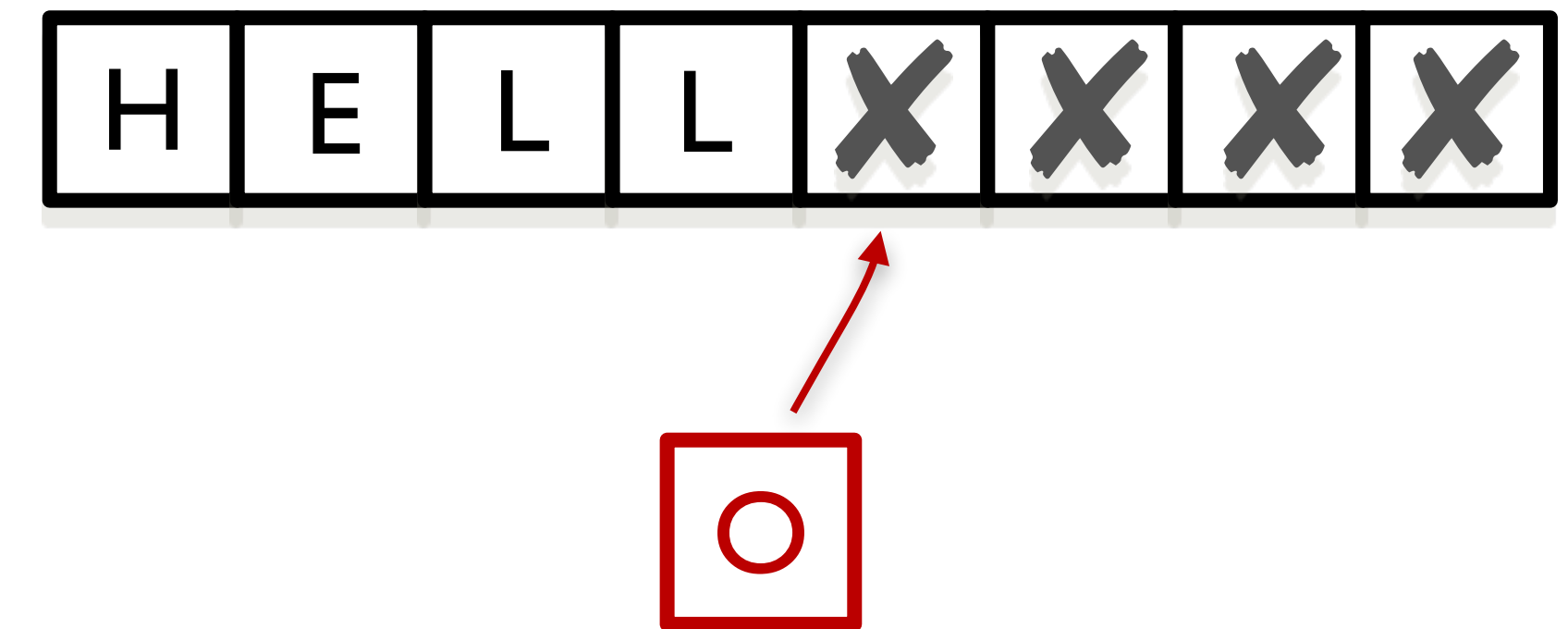
```
méthode nouvelle_capacité()
  si capacité == 0,
    retourner 1
  sinon,
    retourner 2 * capacité
```

Insertion en fin



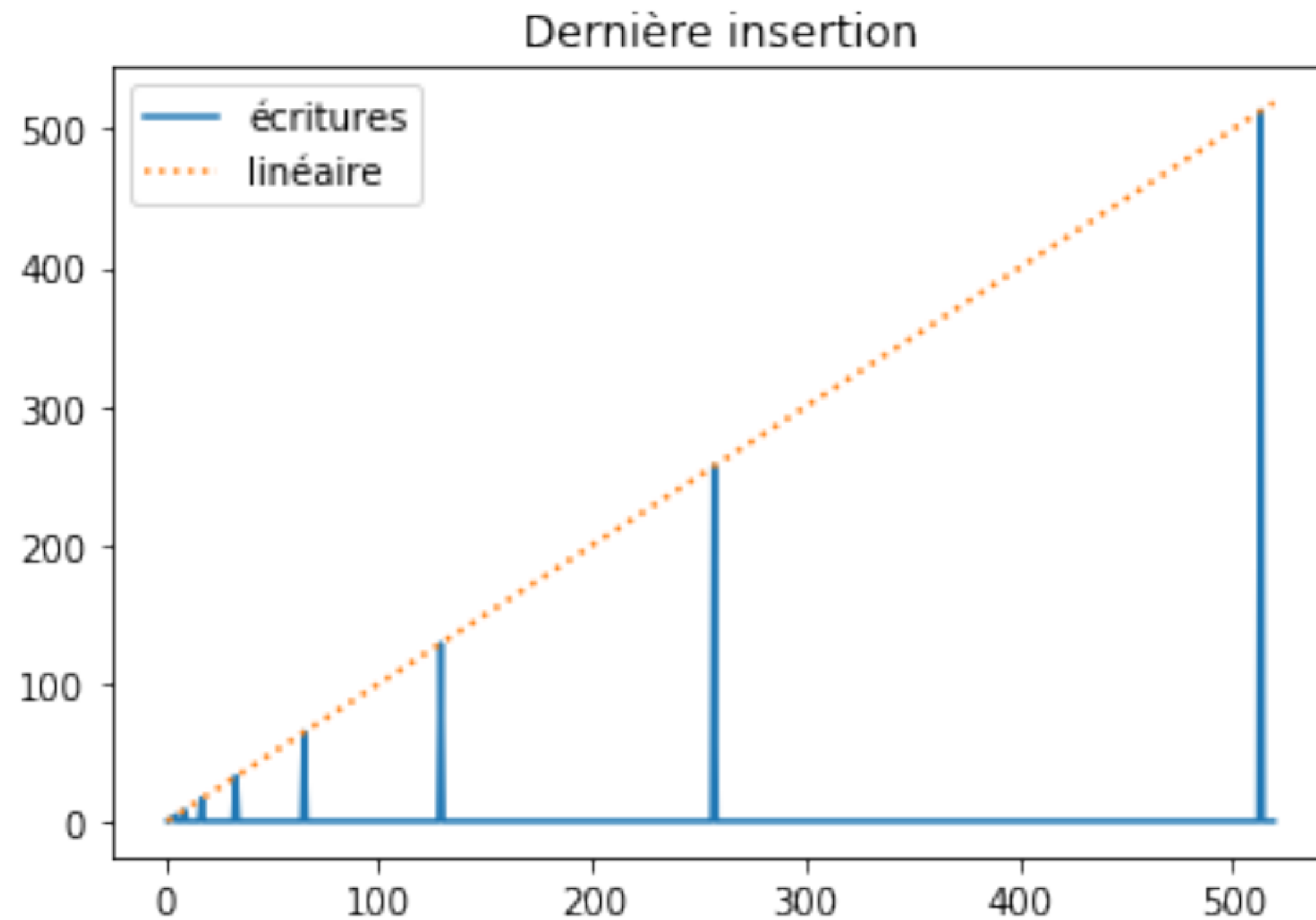
```
méthode insérer_en_fin(v)
  si taille >= capacité
    tmp ← allouer nouvelle_capacité()
    pour i de 0 à taille - 1
      construire tmp[i] ← déplacer data[i]
    pour i de 0 à taille - 1
      détruire data[i]
    capacité ← nouvelle_capacité()
    libérer data
    data ← tmp

  construire data[taille] ← valeur
  taille ← taille + 1
```



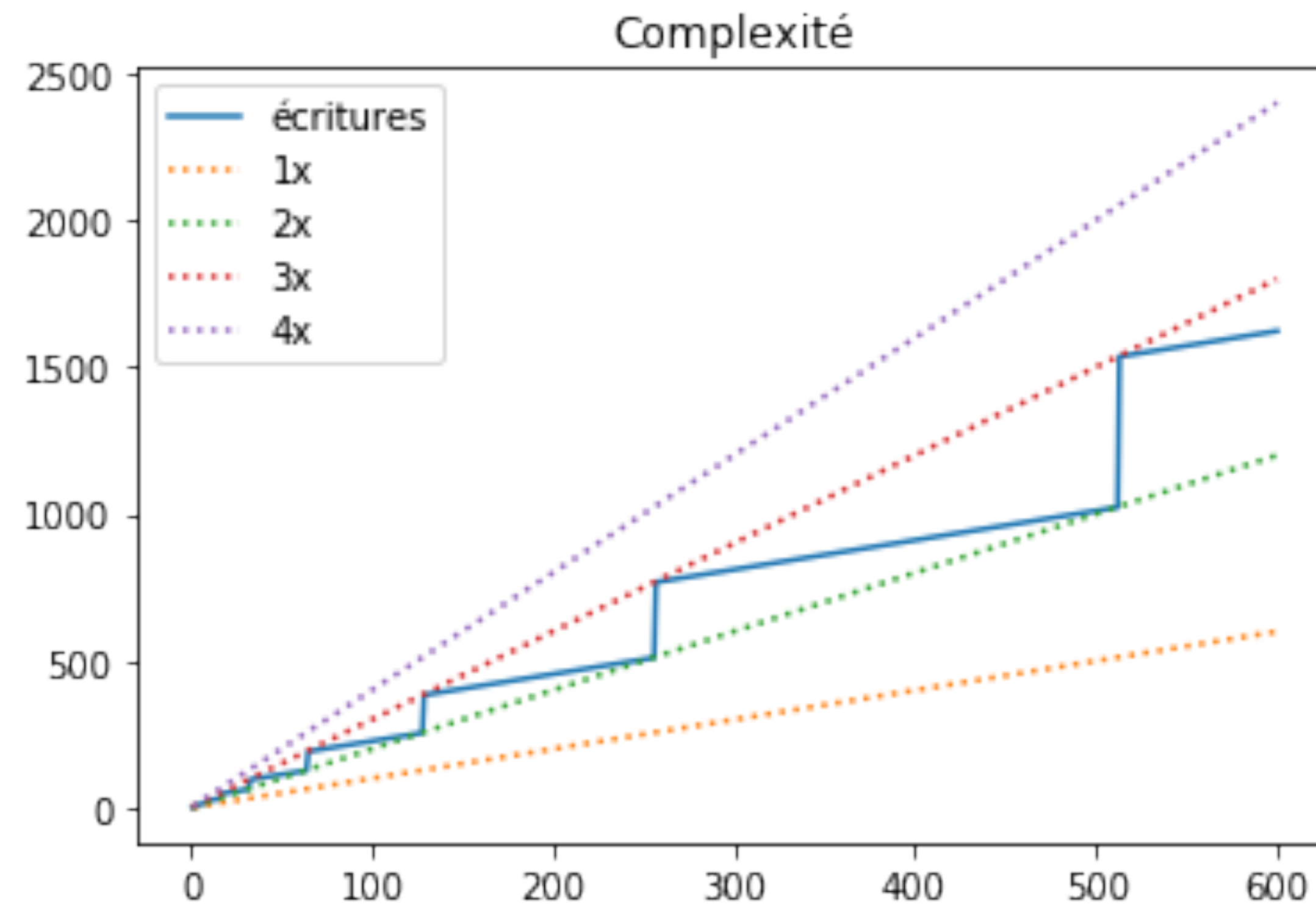
```
méthode nouvelle_capacité()
  si capacité == 0,
    retourner 1
  sinon,
    retourner 2 * capacité
```

Complexité de l'insertion en fin



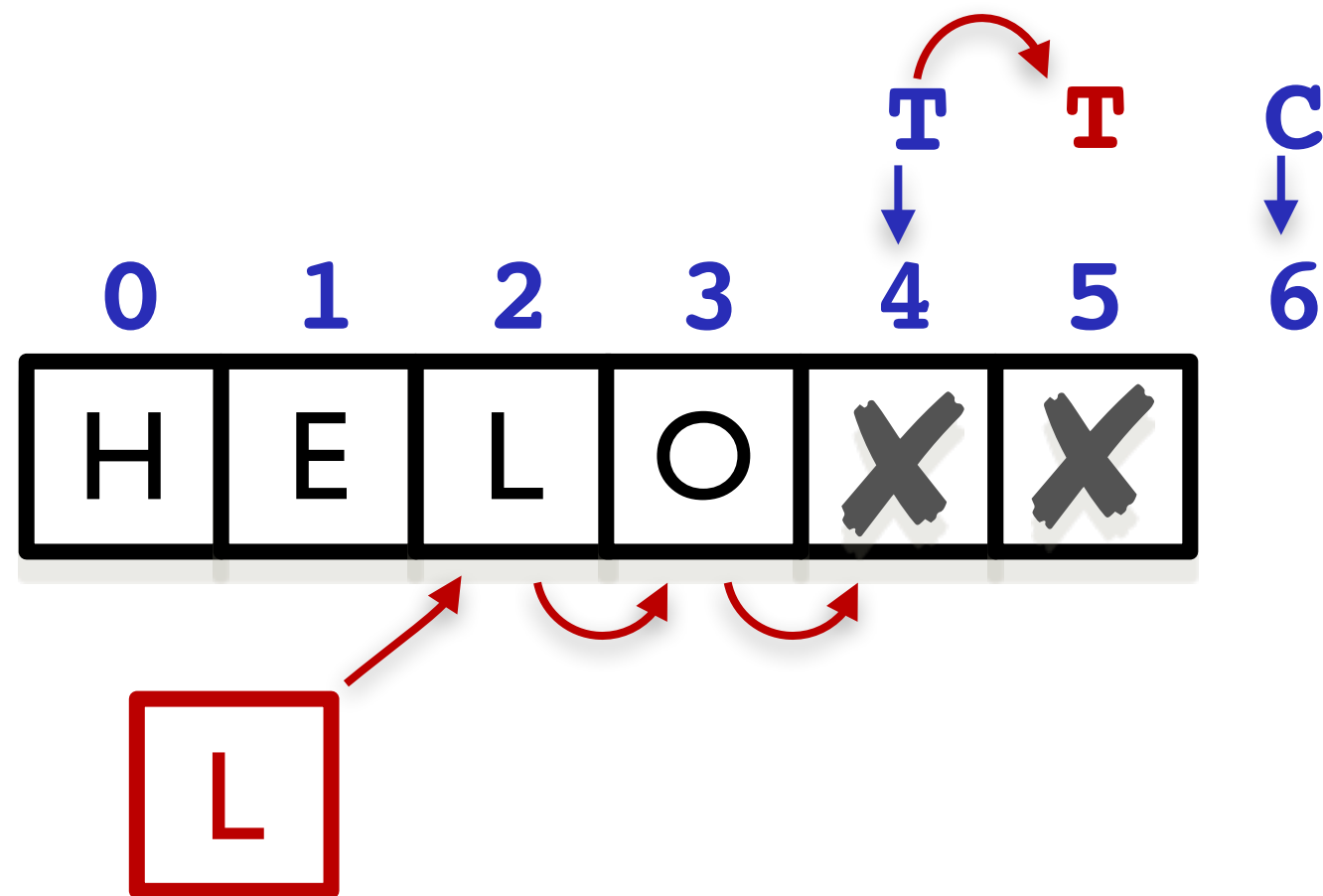
```
v ← tableau vide  
pour i de 0 à 520  
    v.insérer_en_fin(i)  
    Afficher (i, nombre d'écritures de la dernière insertion)
```

Complexité de l'insertion en fin



```

v ← tableau vide
pour i de 0 à 600
    v.insérer_en_fin(i)
    Afficher (i, nombre d'écritures cumulé depuis le début)
    
```



ASD - Algorithmes et Structures de Données - Printemps 2021

Diminuer la capacité ?



H	E	L	L
---	---	---	---

insérer_en_fin('O')

H	E	L	L	O	X	X	X
---	---	---	---	---	---	---	---

supprimer_en_fin()

H	E	L	L
---	---	---	---

insérer_en_fin('O')

H	E	L	L	O	X	X	X
---	---	---	---	---	---	---	---

supprimer_en_fin()

H	E	L	L
---	---	---	---

- Compromis entre temps de calcul et mémoire utilisée
- Ne pas utiliser le même facteur que pour l'augmenter !
- C++ et Java ne le font pas automatiquement
- `vector<T>::shrink_to_fit()` et `ArrayList.trimToSize()`



Opération	<code>std::vector</code>	en moyenne	au pire
Consulter un élément	<code>at, operator[]</code>	$O(1)$	$O(1)$
Modifier un élément	<code>at, operator[]</code>	$O(1)$	$O(1)$
Insertion en fin	<code>push_back</code>	$O(1)$	$O(n)$
... au milieu	<code>insert</code>	$O(n)$	$O(n)$
... au début	<code>insert(begin())</code>	$O(n)$	$O(n)$
Suppression en fin	<code>pop_back</code>	$O(1)$	$O(1)$ ou $O(n)$ *
... au milieu	<code>erase</code>	$O(n)$	$O(n)$
... au début	<code>erase(begin())</code>	$O(n)$	$O(n)$