

# Conteneurs élémentaires

## CPES2: Algorithmique et Applications

Pierre Senellart



1<sup>er</sup> octobre 2020

# Plan

Introduction

Conteneurs de base

Tableau dynamique

Remarques

Références

# Conteneurs

- Structures de données stockant une **séquence**  $S = (s_0, \dots, s_{n-1})$  de  $n$  éléments (des entiers, des nombres à virgule flottante, des objets complexes, etc.)
- **Différentes spécifications** de telles structures, permettant différentes opérations, avec une efficacité différente :
  - **Accès aléatoire** : étant donné  $i$ , récupérer  $s_i$  ; toujours possible en  $O(n)$
  - **Accès** au début ( $s_0$ ), à la fin ( $s_{n-1}$ )
  - **Insertion** au début (avant  $s_0$ ), à une position aléatoire (entre  $s_i$  et  $s_{i+1}$ ), à la fin (après  $s_{n-1}$ )
  - **Suppression** du premier élément ( $s_0$ ), d'un élément aléatoire ( $s_i$ ), du dernier élément ( $s_n$ )
  - **Recherche** d'une occurrence d'un élément  $s$  (renvoyer un  $i$  tel que  $s_i = s$ ) ; toujours possible en  $O(n)$

## Dans ce cours

- Deux grandes **classes de conteneurs** :
  - Tableaux (de taille fixe ou dynamique) : stockage contigu en mémoire
  - Listes (simplement ou doublement) chaînées : stockage non contigu, pointeurs reliant les éléments les uns aux autres
- Des structures de données **abstraites** utilisant ces conteneurs : pile, file, file à deux bouts

# Plan

Introduction

Conteneurs de base

Tableau fixe

Liste chaînée

Pile et file

Tableau dynamique

Remarques

Références

## Tableau de taille fixe

- **Zone de mémoire contiguë**, de taille fixe, allouée à la construction
- **Accès aléatoire** en  $\Theta(1)$  : si le tableau est à la position  $t$  en mémoire et contient des éléments de  $k$  octets chacun, il suffit d'accéder à l'élément à la position  $t + ki$
- Insertion, suppression **impossible**
- Aussi **compact** que possible, pas de place perdue

## Tableau de taille fixe en Python

- N'existe pas dans la bibliothèque standard (mais possibilité d'utiliser des tableaux dynamiques, voir plus loin)
- `numpy.array` dans le module tiers (mais très utilisé) numpy
- Stocke des éléments **homogènes** : types élémentaires (par exemple, entier entre  $-2^{31}$  et  $2^{31} - 1$  ; nombre à virgule flottante sur 64 bits ; etc.) ou types plus complexes formés à partir des types de base
- Nombreuses fonctionnalités particulièrement adaptées au stockage et au traitement efficace de **valeurs numériques**
- Cas particulier d'un `numpy.ndarray` : tableau **multi-dimensionnel**

# Plan

Introduction

Conteneurs de base

Tableau fixe

Liste chaînée

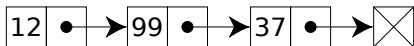
Pile et file

Tableau dynamique

Remarques

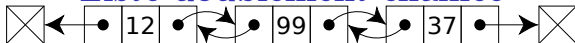
Références

## Liste simplement chaînée



- Chaque élément est stocké dans un **chaînon**, qui contient également un pointeur vers l'élément suivant
- On garde également un pointeur vers le **premier chaînon**
- **Accès au premier élément** en  $\Theta(1)$
- Accès aléatoire ou accès au dernier élément en  $O(n)$
- **Insertion à une position aléatoire** (après accès aléatoire) en  $\Theta(1)$
- **Suppression du premier élément** en  $\Theta(1)$
- **Suppression** en  $\Theta(1)$  si l'élément précédent est connu, en  $O(n)$  sinon

## Liste doublement chaînée



- Chaque élément est stocké dans un **chaînon**, qui contient également un pointeur vers les éléments précédent et suivant
- On garde également un pointeur vers les **premier et dernier chaînons**
- **Accès au premier ou dernier élément** en  $\Theta(1)$
- Accès aléatoire en  $O(n)$
- **Insertion à une position aléatoire** (après accès aléatoire) en  $\Theta(1)$
- **Suppression du premier ou dernier élément** en  $\Theta(1)$
- **Suppression** (après accès aléatoire) en  $\Theta(1)$
- **Plus puissant, moins compact** que la liste simplement chaînée

## Listes chaînées en Python

- Pas de liste simplement chaînée dans la bibliothèque standard
- `collections.deque` est une variante de liste doublement chaînée dans laquelle on ne stocke pas un élément par chaînon, mais un nombre fixe d'éléments (actuellement, 64)
- Sert surtout à implémenter des `files à deux bouts` (voir plus loin)

# Plan

Introduction

Conteneurs de base

Tableau fixe

Liste chaînée

**Pile et file**

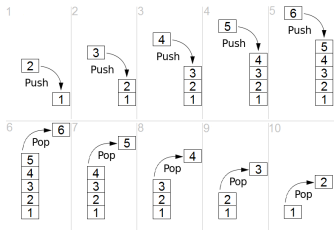
Tableau dynamique

Remarques

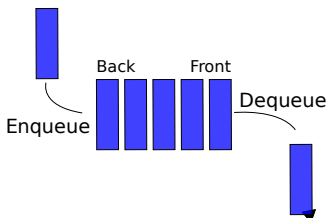
Références

## Pile (*Stack* ou *LIFO* pour *last-in-first-out*)

- Structure de donnée **abstraite**, qui peut être implémentée de différentes manières, p. ex., avec une liste simplement chaînée ou un tableau dynamique
- Seules opérations possibles :
  - Accès au **premier** élément en  $\Theta(1)$
  - Insertion **au début** en  $\Theta(1)$  (**empiler** ou **push**)
  - Suppression du **premier** élément en  $\Theta(1)$  (**dépiler** ou **pop**)



## File (*Queue* or *FIFO* pour *first-in-first-out*)



- Structure de données **abstraite**, qui peut être implémentée de différentes manières, p. ex., par une liste doublement chaînée ou un tableau dynamique à deux bouts
- Seules opérations possibles :
  - Accès au **premier** élément en  $\Theta(1)$
  - Insertion **à la fin** en  $\Theta(1)$  (**enfiler** ou **enqueue**)
  - Suppression du **premier** élément en  $\Theta(1)$  (**défiler** ou **dequeue**)

## File à deux bouts (deque)

- Structure de données **abstraite**, qui peut être implémentée de différentes manières, p. ex., par une liste doublement chaînée ou un tableau dynamique à deux bouts
- **Combine** les opérations des piles et des files
- **Seules opérations possibles** :
  - Accès au **premier** ou au **dernier** élément en  $\Theta(1)$
  - Insertion **au début** ou **à la fin** en  $\Theta(1)$
  - Suppression du **premier** ou du **dernier** élément en  $\Theta(1)$
- Variante parfois utile : file à deux bouts avec **taille maximale** – quand on la file à deux bouts est pleine et qu'on ajoute un élément, on enlève un élément de l'autre côté

## Piles et files en Python

- Pas de pile ou de file simples dans la bibliothèque standard
- Mais `collections.deque` implémente une file à deux bouts avec une liste doublement chaînée (avec plusieurs éléments par maillon)
- Opérations : `append`, `appendleft`, `pop`, `popleft` – on peut donc par exemple utiliser `append` et `pop` pour une pile, et `appendleft` et `pop` pour une file
- On peut spécifier un paramètre `maxlen` à la création pour indiquer une **taille maximale** – par défaut, pas de taille maximale
- Pour les piles, aussi possible d'utiliser des **tableaux dynamiques**

# Plan

Introduction

Conteneurs de base

**Tableau dynamique**  
Complexité amortie  
Tableau dynamique

Remarques

Références

## Complexité amortie

- Jusqu'ici, la complexité algorithmique (en temps) est mesurée séparément pour **chaque** opération d'une structure de données
- Parfois, impossible de fournir une bonne borne sur le temps individuel de chaque opération, mais possible de borner le temps **moyen** d'une **suite finie** d'opérations
- Soit une suite de  **$n$  opérations**  $o_1, \dots, o_n$  sur une structure de données, avec coûts  $c_1, \dots, c_n$
- La complexité moyenne d'une opération  $o_i$  est

$$\frac{1}{n} \sum_{i=1}^n c_i$$

- On appelle ça la **complexité amortie**
- N'a de sens que si on définit précisément le **type de suite d'opérations** considéré (par exemple, insertions)

## Complexité amortie, complexité en moyenne

- Deux notions **orthogonales** :

**Complexité amortie** En moyenne, combien de temps prend une opération **au sein d'une suite finie d'opérations** d'un type donné

**Complexité en moyenne** En moyenne, combien de temps prend une opération, **parmi toutes les entrées possibles** (par opposition au pire des cas)

- On peut donc parler de **complexité amortie dans le pire des cas** (et c'est la notion habituelle) et de complexité amortie en moyenne

## Calcul de la complexité amortie

- **Manuellement**, en calculant le coût de chaque opération au sein d'une séquence et en moyennant le tout
- Parfois **difficile** parce qu'il faut être précis dans ce calcul et considérer l'impact de chaque opération sur les opérations suivantes
- **Astuce** : associer un **potentiel** à la structure de données sur laquelle les opérations ont lieu. Quand l'opération est bon marché, on augmente le potentiel ; quand l'opération est chère, on compense en réduisant le potentiel.
- La méthode du potentiel permet de considérer **chaque opération individuellement** : l'impact de l'opération sur les opérations suivantes est remplacé par l'impact de l'opération sur le potentiel
- Mais parfois difficile de trouver la bonne fonction de potentiel

## Méthode du potentiel

- On associe un **potentiel**  $\Phi(X)$  (nombre réel) à une structure de données  $X$
- Soit une suite de  **$n$  opérations**  $o_1, \dots, o_n$ , de coûts réels  $c_1, \dots, c_n$ , et les structures de données correspondantes  $X_0, \dots, X_n$  ( $X_i = o_i(X_{i-1})$ )
- Soit  $\hat{c}_i := c_i + \Phi(X_i) - \Phi(X_{i-1})$
- Alors :  $\frac{1}{n} \sum_{i=1}^n \hat{c}_i = \frac{1}{n} \sum_{i=1}^n c_i + \frac{1}{n} (\Phi(X_n) - \Phi(X_0))$
- Si  $\hat{c}_i = O(f(|X_n|))$ , alors  $\frac{1}{n} \sum_{i=1}^n \hat{c}_i = O(f(|X_n|))$
- Donc  $\frac{1}{n} \sum_{i=1}^n c_i = O(f(|X_n|))$  dès que, pour tout  $n$ ,  $\Phi(X_n) - \Phi(X_0) \geq 0$
- On prend souvent  $X_0$  la structure de données vide et  $\Phi(X_0) := 0$ , ce qui donne la condition  $\Phi(X_n) \geq 0$

# Plan

Introduction

Conteneurs de base

Tableau dynamique

Complexité amortie

Tableau dynamique

Remarques

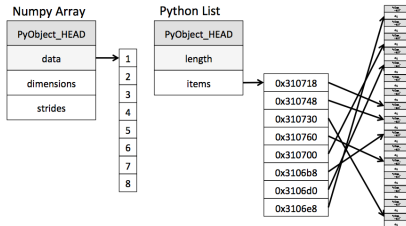
Références

## Application : tableau dynamique

- Pointeur vers un tableau de taille fixe  $cap$  + entier  $n \leq cap$  stockant le nombre d'éléments réellement utilisé dans ce tableau
- **Accès aléatoire** en  $\Theta(1)$  comme pour un tableau de taille fixe
- Suppression du dernier élément en  $\Theta(1)$  en décrémentant  $n$
- Insertion à la fin en **complexité amortie**  $\Theta(1)$  (voir plus loin)
- Suppression ou insertion ailleurs qu'à la fin en  $\Theta(n)$  : on recopie le tableau

## Tableau dynamique en Python

- Les **listes Python** sont des tableaux dynamiques
- De même pour **array.array**, structure de données de la bibliothèque standard
- Les listes Python sont des structures de données **hétérogènes** (n'importe quelle valeur Python peut être ajoutée à une liste existante)
- Les `array.array` (et les `numpy.array` de taille fixe) sont des structures de données **homogènes** : bien plus compact en mémoire, traitement des données bien plus efficaces



## Insertion à la fin d'un tableau dynamique

**Entrée:** tableau  $T$  de capacité  $cap$  et de taille  $n$ , élément  $x$

**Sortie:** tableau  $T$  de taille  $n + 1$  avec  $T[n] = x$

if  $n = cap$  then

    créer un nouveau tableau  $T'$  de capacité  $\max(2 \times cap, 1)$

    for  $i \leftarrow 0$  to  $n - 1$  do

$T'[i] \leftarrow T[i]$

    end for

    détruire le tableau référencé par  $T$

    faire pointer  $T$  vers  $T'$

$cap \leftarrow \max(2 \times cap, 1)$

end if

$T[n] \leftarrow x$

$n \leftarrow n + 1$

## Complexité amortie de $n$ insertions – direct (1/2)

- On part du tableau vide  $T_0$ , on **insère  $n$  éléments** (en obtenant successivement les tableaux  $T_1, \dots, T_n$  de taille  $1 \dots n$  et de capacité  $cap(T_1), \dots, cap(T_n)$ )
- On calcule  $c_i$  pour  $1 \leq i \leq N$
- Pour les  $i$  tels que  $i - 1 \neq cap(T_{i-1})$ ,  $c_i = \Theta(1)$
- Pour les  $i$  tels que  $i - 1 = cap(T_{i-1})$ ,  $c_i = \Theta(i)$
- Quels sont les  $i$  tels que  $i = cap(T_i)$ ? 0, 1, 2, 4, 8, 16, ... ce sont les **puissances de 2** (plus 0)
- Il y a  $E[\log_2(n - 1)] + 1 = \Theta(\log n)$  puissances de 2 entre 0 et  $n - 1$  (plus 0)

Complexité amortie de  $n$  insertions – direct (2/2)

$$\frac{1}{n} \sum_{i=1}^n c_i \leq \frac{1}{n} \left( \sum_{k=0}^{\mathbb{E}[\log_2(n-1)]} \Theta(2^k) + O(n) \right)$$

Rappel :  $\sum_{k=0}^m \alpha^k = \frac{\alpha^{m+1} - 1}{\alpha - 1}$

$$\frac{1}{n} \sum_{i=1}^n c_i \leq \frac{1}{n} O \left( 2^{\mathbb{E}[\log_2(n-1)]+1} + O(n) \right) = \frac{1}{n} (O(n) + O(n)) = O(1)$$

## Complexité amortie de $n$ insertions – potentiel (1/2)

- On part du tableau vide  $T_0$ , on **insère  $n$  éléments** (en obtenant successivement les tableaux  $T_1, \dots, T_n$  de taille  $1 \dots n$  et de capacité  $cap(T_1), \dots, cap(T_n)$ )
- On définit le potentiel d'un tableau  $T_i$  de taille  $i$  et de capacité  $cap(T_i)$  par  **$\Phi(T_i) := (2i - cap(T_i)) \times \alpha$**  pour une certaine constante  $\alpha$  définie plus tard
- On a bien  $\Phi(T_0) = 0$  et  $\Phi(T_i) \geq 0$  pour tout  $i$
- On calcule la complexité  $\hat{c}_i$  amortie par le potentiel  $\Phi$  :  
 **$\hat{c}_i := c_i + \Phi(X_i) - \Phi(X_{i-1})$**

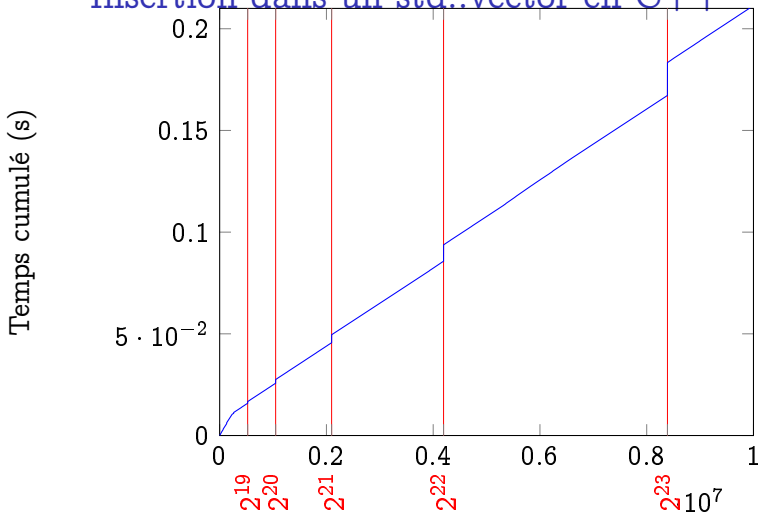
## Complexité amortie de $n$ insertions – potentiel (2/2)

- Pour les  $i$  tels que  $i - 1 \neq \text{cap}(T_{i-1})$ ,  $c_i = O(1)$  et  $\Phi(X_i) - \Phi(X_{i-1}) = 2\alpha$ ; on prend  $\alpha$  tel que  $c_i \leq \alpha$  pour  $i$  suffisamment grand
- Pour les  $i$  tels que  $i - 1 = \text{cap}(T_{i-1})$ ,  $c_i = O(i)$  et  $\Phi(X_i) - \Phi(X_{i-1}) = (2 - n)\alpha$ ; on prend  $\alpha$  tel que  $c_i \leq \alpha i$  pour  $i$  suffisamment grand
- On a alors

$$\hat{c}_i \leq \max(\alpha + 2\alpha, \alpha i + (2 - n)\alpha) = 3\alpha = O(1)$$

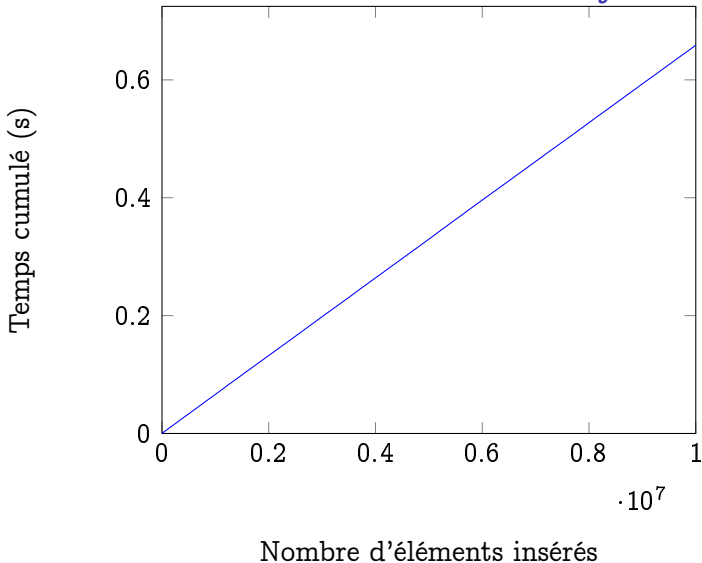
## Suppression dans les tableaux dynamiques

- Tel que présenté : suppression en  $O(1)$  à la fin du tableau (on décrémente  $n$ )
- Mais ça veut dire qu'**on ne libère jamais d'espace mémoire**, même si le tableau diminue beaucoup : complexité en espace excessive
- Aussi : analyse par la méthode du potentiel d'une combinaison d'insertions et suppressions invalide si  $2n - cap < 0$
- En pratique : on diminue la capacité du tableau **quand la taille est suffisamment diminuée**
- On ne prend pas la même borne pour éviter de constamment créer/détruire des tableaux si on ajoute/enlève successivement un élément proche de la capacité
- Analyse complète : chap. 17 de Cormen et al. [2010, 2009]

Insertion dans un `std::vector` en C++

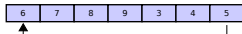
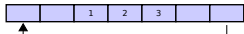
Nombre d'éléments insérés

## Insertion dans une liste en Python



## Tableau dynamique à deux bouts

- **But** : avoir une complexité amorti en  $O(1)$  pour l'ajout et la suppression **au début comme à la fin d'un tableau** ; on conserve l'accès aléatoire en  $O(1)$
- Utile pour implémenter une file, une file à deux bouts
- Deux possibilités pour l'implémenter :
  - Un tableau dynamique que l'on remplit **à partir du milieu** – quand on atteint le début ou la fin du tableau alloué, on crée un nouveau tableau plus grand
  - Un tableau dynamique implémentant un **buffer circulaire** : même concept mais si on arrive à un bout du tableau et qu'il reste de la place à l'autre bout, on y met les éléments jusqu'à ce que le tableau soit entièrement rempli



# Plan

Introduction

Conteneurs de base

Tableau dynamique

**Remarques**

Références

## Nombre d'éléments

- Les conteneurs peuvent (ou non) stocker en plus des éléments le **nombre d'éléments** pour pouvoir l'obtenir en  $\Theta(1)$
- Tableau dynamique : **obligatoire**
- Tableau de taille fixe : optionnel, mais **dangereux** de ne pas le stocker (accès à de la mémoire hors bornes)
- Liste chaînée : **optionnel** ; si non stocké, peut être calculé en  $\Theta(n)$
- **Très peu coûteux** à maintenir, et utile à avoir, donc le plus souvent stocké

## Variante : conteneur trié

- Les éléments de la séquence ne sont pas en positions arbitraires mais on suppose  $s_0 \leq \dots \leq s_{n-1}$  pour un certain opérateur de comparaison  $\leq$
- On n'insère pas à une position arbitraire, mais on insère en suivant l'ordre
- Tableaux (fixes et dynamiques) : recherche en  $O(\log n)$  par recherche dichotomique
- Tableau dynamique : insertion en  $O(n)$
- Liste chaînée : recherche et insertion en  $O(n)$  – mais des listes chaînées triées peuvent être utiles dans certains cas, par exemple pour implémenter des algorithmes de tri

# Plan

Introduction

Conteneurs de base

Tableau dynamique

Remarques

Références

## Références

- **Conteneurs de bases** : chap. 10 de [Cormen et al., 2009, 2010]
- **Complexité amortie et tableaux dynamiques** : chap. 17 de [Cormen et al., 2009, 2010]
- Utilisation des `numpy.array` et bien d'autre chose pour la **science des données** en Python : VanderPlas [2016] (en anglais, librement accessible en ligne)

## Bibliographie

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009. ISBN 978-0-262-03384-8. URL

<http://mitpress.mit.edu/books/introduction-algorithms>.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Algorithmique*. Dunod, 3rd edition, 2010. ISBN 978-2-100-54526-1. URL

<https://www.dunod.com/sciences-techniques/algorithmique-cours-avec-957-exercices-et-158-problemes>.

Jake VanderPlas. *Python Data Science Handbook*. O'Reilly, 2016. <https://jakevdp.github.io/PythonDataScienceHandbook/>.

## Ressources utilisées et licence

- L'image d'une file est due à Vegpuff (Wikimedia), CC-BY-SA-3.0.
- L'image du stockage mémoire des tableaux Numpy vs les listes Python est due à Jake VanderPlas et extraite de VanderPlas [2016], CC-BY-NC-ND-3.0.
- L'image des buffers circulaires est due à Cburnett (Wikimedia), CC-BY-SA-3.0.

L'ensemble du contenu de cette présentation est sous licence CC-BY-4.0, à l'exception des éléments ci-dessus qui conservent une licence plus restrictive.