Algorithmique

Structures de données : Les tableaux

Florent Hivert

Mél : `Florent.Hivert@lri.fr`
Page personnelle : `http://www.lri.fr/~hivert`

Algorithmes et structures de données

La plupart des bons algorithmes fonctionnent grâce à une méthode astucieuse pour organiser les données. On distingue quatre grandes classes de structures de données :

- Les structures de données séquentielles (tableaux) ;
- Les structures de données linéaires (liste chaînées) ;
- Les arbres ;
- Les graphes.

Structures séquentielles : les tableaux

Structure de donnée séquentielle (tableau)

En anglais : array, vector.

Définition

*Un **tableau** est une structure de donnée T qui permet de stocker un certain nombre d'éléments $T[i]$ repérés par un index i. Les tableaux vérifient généralement les propriétés suivantes :*

- *tous les éléments ont le même type de base ;*
- *le nombre d'éléments stockés est fixé ;*
- *l'accès et la modification de l'élément numéro i est en temps constant $\Theta(1)$, indépendant de i et du nombre d'éléments dans le tableau.*

Tableau en C

- On suppose déclaré un type `elem` pour les éléments.
- Espace mémoire nécessaire au stockage d'un élément exprimé en mots mémoire (octets en général) : `sizeof(elem)`.

Retenir

- *définition statique :* `elem t[taille];`
- *définition dynamique en deux temps (déclaration, allocation) :*

```
#include <stdlib.h>
elem *t;
t = (elem*) malloc(taille*sizeof(elem));
```

- *l'adresse de* `t[i]` *est noté* `t + i`*. Calculée par*

$$\mathrm{Addr}(\texttt{t[i]}) = \mathrm{Addr}(\texttt{t[0]}) + \texttt{sizeof(elem)} * \texttt{i}$$

Tableau en C

- On suppose déclaré un type `elem` pour les éléments.
- Espace mémoire nécessaire au stockage d'un élément exprimé en mots mémoire (octets en général) : `sizeof(elem)`.

Retenir

- *définition statique :* `elem t[taille];`
- *définition dynamique en deux temps (déclaration, allocation) :*

```
#include <stdlib.h>
elem *t;
t = (elem*) malloc(taille*sizeof(elem));
```

- *l'adresse de* `t[i]` *est noté* `t + i`*. Calculée par*

$$\mathrm{Addr}(\mathtt{t[i]}) = \mathrm{Addr}(\mathtt{t[0]}) + \mathtt{sizeof(elem)} * \mathtt{i}$$

Tableau en C

- On suppose déclaré un type `elem` pour les éléments.
- Espace mémoire nécessaire au stockage d'un élément exprimé en mots mémoire (octets en général) : `sizeof(elem)`.

Retenir

- *définition statique :* `elem t[taille];`
- *définition dynamique en deux temps (déclaration, allocation) :*

```
#include <stdlib.h>
elem *t;
t = (elem*) malloc(taille*sizeof(elem));
```

- *l'adresse de* `t[i]` *est noté* `t + i`. *Calculée par*

$$\mathrm{Addr}(\texttt{t[i]}) = \mathrm{Addr}(\texttt{t[0]}) + \texttt{sizeof(elem)} * \texttt{i}$$

Tableau en C

- On suppose déclaré un type `elem` pour les éléments.
- Espace mémoire nécessaire au stockage d'un élément exprimé en mots mémoire (octets en général) : `sizeof(elem)`.

Retenir

- *définition statique :* `elem t[taille];`
- *définition dynamique en deux temps (déclaration, allocation) :*

```
#include <stdlib.h>
elem *t;
t = (elem*) malloc(taille*sizeof(elem));
```

- *l'adresse de* `t[i]` *est noté* `t + i`. *Calculée par*

$$\mathrm{Addr}(\texttt{t[i]}) = \mathrm{Addr}(\texttt{t[0]}) + \texttt{sizeof(elem)} * \texttt{i}$$

Opérations de base (1)

Hypothèses :

- tableau de taille max_taille alloué
- éléments $0 \leq i < \text{taille} \leq \text{max\_taille}$ initialisés

Retenir (Opérations de base : accès)

- *accès au premier élément :* $\Theta(1)$
- *accès à l'élément numéro i :* $\Theta(1)$
- *accès au dernier élément :* $\Theta(1)$

Opérations de base (2)

Hypothèses :

- tableau de taille max_taille alloué
- éléments $0 \leq i <$ taille $\leq$ max_taille initialisés

Retenir (Opérations de base : insertions)

- *insertion d'un élément au début :* $\Theta(\text{taille})$
- *insertion d'un élément en position i :* $\Theta(\text{taille} - i) \subset O(\text{taille})$
- *insertion d'un élément à la fin :* $\Theta(1)$

Opérations de base (3)

Hypothèses :

- tableau de taille max_taille alloué
- éléments $0 \leq i < \text{taille} \leq \text{max\_taille}$ initialisés

Retenir (Opérations de base : suppressions)

- *suppression d'un élément au début :* $\Theta(\text{taille})$
- *suppression d'un élément en position i :* $\Theta(\text{taille} - i) \subset O(\text{taille})$
- *suppression d'un élément à la fin :* $\Theta(1)$

Algorithmes de base

Hypothèses :

- tableau de taille taille alloué et initialisé

Retenir (Algorithmes de base)

- *accumulation d'une opération sur les éléments du tableaux (par exemple : somme, produit, maximum, etc.) :* $\Theta(\text{taille})$ *nombre d'opération.*
- *cas particulier ; comptage du nombre d'éléments qui vérifie une certaine condition :* $\Theta(\text{taille})$ *tests de la condition*
- *recherche d'une occurence, d'un élement qui vérifie une certaine condition :* $O(\text{taille})$ *tests de la condition*

Problème de la taille maximum

On essaye d'insérer un élément dans un tableau où

$$\text{taille} = \text{max\_taille}$$

Il n'y a plus de place disponible.

Comportements possibles :

- Erreur (arrêt du programme, exception)
- Ré-allocation du tableau avec recopie, coût : $\Theta(\text{taille})$

Problème de la taille maximum

On essaye d'insérer un élément dans un tableau où

$$\text{taille} = \text{max\_taille}$$

Il n'y a plus de place disponible.

Comportements possibles :

- Erreur (arrêt du programme, exception)
- Ré-allocation du tableau avec recopie, coût : $\Theta(\text{taille})$

Ré-allocation (2)

On ajoute 1 par 1 n éléments. On suppose que l'on réalloue une case supplémentaire à chaque débordement. Coût (nombre de copies d'éléments) :

$$n + \sum_{i=1}^{n-1} i = \frac{n(n+1)}{2} \in \Theta(n^2)$$

Remarque : si on alloue des blocs de taille b, en notant $k = \lceil \frac{n}{b} \rceil$:

$$n + \sum_{i=1}^{k-1} b\, i = n + b\frac{k(k-1)}{2} \approx \frac{(bk)^2}{b} = \frac{n^2}{b} \in \Theta(n^2)$$

La vitesse est divisée par b mais la **complexité reste la même**.

Ré-allocation (2)

On ajoute 1 par 1 n éléments. On suppose que l'on réalloue une case supplémentaire à chaque débordement. Coût (nombre de copies d'éléments) :

$$n + \sum_{i=1}^{n-1} i = \frac{n(n+1)}{2} \in \Theta(n^2)$$

Remarque : si on alloue des blocs de taille b, en notant $k = \lceil \frac{n}{b} \rceil$:

$$n + \sum_{i=1}^{k-1} b\, i = n + b\frac{k(k-1)}{2} \approx \frac{(bk)^2}{b} = \frac{n^2}{b} \in \Theta(n^2)$$

La vitesse est divisée par b mais la **complexité reste la même**.

Ré-allocation par doublement de taille

Retenir (Solution au problème de la réallocation)

À chaque débordement on ré-alloue $\lceil K \cdot \mathrm{max\_taille} \rceil$ *où* $K > 1$ *est une constante fixée (par exemple* $K = 2$*).*

Si l'on veut à la fin un tableaux de n éléments, le nombre de copies d'éléments (en comptant la copie dans le tableau) est environ

$$C = \sum_{i=1}^{m} K^i = \frac{K^{m+1} - 1}{K - 1} \in \Theta(K^m)$$

où m est le nombre de ré-allocations c'est-à-dire le plus petit entier tel que $K^m \geq n$, soit $m = \lceil \log_K(n) \rceil$. On a donc

$$n \leq K^m < Kn$$

Finalement le coût est $\Theta(n)$.

Nombre moyen de copies

Selon la valeur de K, la constante de complexité varie de manière importante : Le nombre de recopies d'éléments est approximativement

$$C = \sum_{i=1}^{m} K^i = \frac{K^{m+1} - 1}{K - 1} \approx \frac{K}{K-1} n$$

Quelques valeurs :

K	1.01	1.1	1.2	1.5	2	3	4	5	10
$\frac{K}{K-1}$	101	11	6	3	2	1.5	1.33	1.25	1.11

Interprétation : Si l'on augmente la taille de 10% à chaque étape, chaque nombre sera recopié en moyenne 11 fois. Si l'on double la taille à chaque étape, chaque nombre sera en moyenne recopié deux fois.

Nombre moyen de copies

Selon la valeur de K, la constante de complexité varie de manière importante : Le nombre de recopies d'éléments est approximativement

$$C = \sum_{i=1}^{m} K^i = \frac{K^{m+1} - 1}{K - 1} \approx \frac{K}{K-1} n$$

Quelques valeurs :

K	1.01	1.1	1.2	1.5	2	3	4	5	10
$\frac{K}{K-1}$	101	11	6	3	2	1.5	1.33	1.25	1.11

Interprétation : Si l'on augmente la taille de 10% à chaque étape, chaque nombre sera recopié en moyenne 11 fois. Si l'on double la taille à chaque étape, chaque nombre sera en moyenne recopié deux fois.

Bilan

Retenir (Nombre de copies)

Dans un tableau de taille n, coût de l'ajout d'un élément ***dans le pire*** *des cas :*

$$\textit{Coût en temps} \approx n, \qquad \textit{Coût en espace} \approx (K-1)n\,.$$

En moyenne*, sur un grand nombre d'éléments ajoutés :*

$$\textit{Coût en temps} \approx \frac{K}{K-1}, \qquad \textit{Coût en espace} \approx K\,.$$

On dit que l'algorithme travaille en ***temps constant amortis*** *(Constant Amortized Time (CAT) en anglais).*

Compromis Espace/Temps

Quand K augmente, la vitesse augmente mais la place mémoire gaspillée ($\approx (K-1)n$) augmente aussi. Le choix de la valeur de K dépend donc du besoin de vitesse par rapport au coût de la mémoire.

Retenir

C'est une situation très classique : dans de nombreux problèmes, il est possible d'aller plus vite en utilisant plus de mémoire.

Exemple : on évite de faire plusieurs fois les même calculs en stockant le résultat.

Compromis Espace/Temps

Quand K augmente, la vitesse augmente mais la place mémoire gaspillée ($\approx (K-1)n$) augmente aussi. Le choix de la valeur de K dépend donc du besoin de vitesse par rapport au coût de la mémoire.

Retenir

C'est une situation très classique : dans de nombreux problèmes, il est possible d'aller plus vite en utilisant plus de mémoire.

Exemple : on évite de faire plusieurs fois les même calculs en stockant le résultat.

Compromis Espace/Temps

Quand K augmente, la vitesse augmente mais la place mémoire gaspillée ($\approx (K-1)n$) augmente aussi. Le choix de la valeur de K dépend donc du besoin de vitesse par rapport au coût de la mémoire.

Retenir

C'est une situation très classique : dans de nombreux problèmes, il est possible d'aller plus vite en utilisant plus de mémoire.

Exemple : on évite de faire plusieurs fois les même calculs en stockant le résultat.

En pratique ...

On utilise `void *realloc(void *ptr, size_t size);`

Extrait des sources du langage Python
Fichier `listobject.c`, ligne 41-91 :

```
/* This over-allocates proportional to the list size, making room
 * for additional growth.  The over-allocation is mild, but is
 * enough to give linear-time amortized behavior over a long
 * sequence of appends() in the presence of a poorly-performing
 * system realloc().
 * The growth pattern is:  0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...
 */
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);
```