STRUCTURES ET TYPES ABSTRAITS

À la fin de ce chapitre, je sais :

- ☞ expliquer la notion de type abstrait de données
- ☞ distinguer les différentes structures de données au programme
- ☞ choisir une structure de données adaptée à un algorithme

Écrire un programme optimal en terme de complexité nécessite l'identification des structures de données utilisées très tôt dans le développement : le choix d'une structure de données plutôt qu'une autre, par exemple choisir une entier long plutôt qu'un flottant ou une liste au lieu d'un tableau, peut rendre inefficace un algorithme selon le choix effectué. Le génie logiciel s'appuie à la fois sur :

des types simples comme les (`int, float, bool, char`) sont les éléments de base de l'informaticien, éléments qui représentent une information simple, **atomique**.

et des collections de données comme les listes, les tableaux, les arbres, les files, les piles sont des **collections** de données qui permettent et de manipuler l'information sous la forme d'ensembles ordonnés ou non. Ces collections facilitent la réalisation d'un logiciel et le rendent plus efficace.

Ce chapitre a pour but d'approfondir la définition des structures de données afin de permettre un choix éclairé, c'est-à-dire adapté à un algorithme. C'est pourquoi on définit d'abord ce qu'est un type de données abstrait en illustrant ce concept sur les listes, les tableaux, les piles et les files. Puis on fait le lien avec les implémentations possibles de ces types en structures de données.

A Type abstrait de données et structure de données

■ **Définition 1 — Type abstrait de données (TAD).** Un type de données abstrait est une abstraction d'une structure de données qui ne se préoccupe pas de son implémentation sur une machine : sa structure interne est indiscernable, le type abstrait est vu de l'extérieur comme une boite noire.

Un TAD spécifie le quoi, c'est à dire le type de données contenues ainsi que les opérations qu'on faire dessus. Par contre, il ne spécifie pas comment dont les données sont stockées ni comment les opérations sont implémentées.

■ **Définition 2 — Structure de données.** Une structure de données est une mise en œuvre concrète d'un type abstrait, une implémentation d'un type abstrait sur dans un langage de programmation.

Ⓡ Un type abstrait de données est à une structure de donnée ce qu'un algorithme est à un programme. On spécifie un algorithme ou un type abstrait de données, mais on implémente un programme ou une structure de données.

■ **Exemple 1 — Un entier.** Un entier est un TAD qui :

(données) contient une suite de chiffres [a] éventuellement précédés par un signe − ou +,

(opérations) fournit les opérations $+, -, \times, //, \%$.

Selon le langage, ce TAD entier est implémenté en machine par un type concret différent :

- `int` en Python,
- `Integer` ou `int` en Java,
- `char`, `short`, `int`, `uint`, `long int` en C,
- `int` en OCaml.

a. peu importe la base pour l'instant...

■ **Exemple 2 — Un booléén.** De la même manière, on peut définir un TAD qui désigne un booléen. Un booléen est un TAD qui :

(données) se note Vrai ou Faux,

(opérations) fournit les opérations logiques conjonction, disjonction et négation...

Selon le langage, ce TAD booléen est implémenté en machine par un type concret différent :

- `bool` valant `True` ou `False` en Python,
- `boolean` valant `true` ou `false` en Java,
- `bool` valant `1` ou `0` en C,
- `bool` valant `true` ou `false` en OCaml.

Les exemples précédents de types abstrait de données étaient limités à des types simples. Mais il est possible de définir des types abstraits de données composés.

■ **Exemple 3 — Types abstraits de données composés.** Voici quelques types abstraits composés parmi les plus courants :

- liste,
- file,
- pile,
- arbre binaire,
- dictionnaire ou tableau associatif,
- ensemble,
- graphe.

Ⓡ Il faut faire attention, car si les objets de type `list` en Python implémente le TAD liste, ce n'est pas la seule implémentation possible.

B TAD Tableau

■ **Définition 3 — TAD tableau.** Un TAD tableau représente une structure finie indiçable par des entiers. Cela signifie qu'on peut accéder à la lecture ou à l'écriture de n'importe quel élément directement en utilisant un indice, par exemple `t[3]`.

(données) le plus souvent des nombres, en tout cas des types simples identiques : on appelle la donnée l'élément d'un tableau.

(opérations) deux opérations principales :

- accès à un élément via un indice entier via un opérateur de type `[]`,
- enregistrement de la valeur d'un élément d'après son indice.

Les implémentations du TAD tableau sont la plupart du temps des structures des données linéaires en mémoire : les données d'un tableau sont rangées dans des zones mémoires **continues**, les unes derrières les autres. On peut décliner le TAD tableau de manière :

1. statique : la taille du tableau est fixée, on ne peut pas ajouter ou enlever d'éléments.
2. dynamique : la taille du tableau peut varier, on peut ajouter ou enlever des éléments. Dans ce cas, on parle de tableau dynamique.

C TAD Liste

■ **Définition 4 — TAD liste.** Un TAD liste représente **une séquence finie d'éléments d'un même type** qui possède un **rang** dans la séquence. Les données sont traitées séquentiellement, dans l'ordre du rang.

Un TAD liste est **dynamique**, c'est à dire qu'on peut ajouter ou enlever n'importe quel élément.

La longueur d'une liste est le nombre d'éléments qu'elle contient. On dit qu'une liste est vide si elle ne contient aucun élément, sa longueur vaut zéro. Le début de la liste est désigné par le terme tête de liste (**head**), le dernier élément de la liste par la fin de la liste (**tail**).

(données) de type simple ou composé

(opérations) on peut trouver [a] :

- un constructeur de liste vide,
- un opérateur de test de liste vide,
- un opérateur pour ajouter en tête de liste,
- un opérateur pour ajouter en fin de liste,
- un opérateur pour déterminer la tête de la liste,
- un opérateur pour déterminer la queue de la liste (tout sauf la tête),
- un opérateur pour accéder au ième élément.
- un opérateur pour supprimer en début et/ou en fin de liste,
- un opérateur pour accéder au dernier élément de la liste.

a. Toutes les implémentations ne proposent pas nécessairement toutes ces opérations !

D Implémentations des tableaux

a Implémentation d'un tableau statique

Dans sa version statique, un TAD tableau de taille fixe n est implémenté par un bloc de mémoire contiguë contenant n cases(cf. figure 1). Ces cases sont capables d'accueillir le type d'élément que contient le tableau.

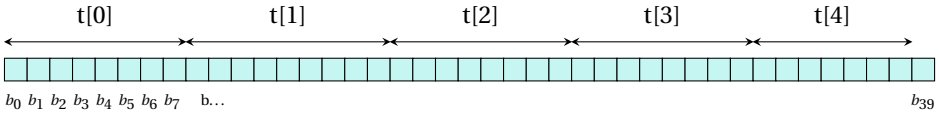

FIGURE 1 – Représentation d'un tableau statique en mémoire. Il peut représenter un tableau t de cinq entiers codés sur huit bits. On accède directement à l'élément i en écrivant $t[i]$.

Par exemple, pour un TAD tableau statique de cinq entiers codés sur huit bits, on alloue

un espace mémoire de 40 bits subdivisés en cinq octets comme indiqué sur la figure 1. Dans la majorité des langages, l'opérateur [] permet alors d'accéder aux éléments [1], par exemple t[3]. Les éléments sont numérotés à partir de zéro : t[0] est le premier élément.

On peut estimer les coûts associés à l'utilisation d'un tableau statique comme le montre le tableau 1.

Opération	**Complexité**	**Raison**
Accès à un élément au début	$O(1)$	
Accès à un élément à la fin	$O(1)$	
Accès à un élément au milieu	$O(1)$	
Ajout d'un élément au début	$O(n)$	créer un nouveau tableau
Ajout d'un élément à la fin	$O(n)$	créer un nouveau tableau
Suppression d'un élément au début	$O(n)$	créer un nouveau tableau
Suppression d'un élément à la fin	$O(n)$	créer un nouveau tableau

TABLE 1 – Complexité des opérations associées à l'utilisation d'un tableau statique.

b Implémentation d'un tableau dynamique

Un tableau dynamique est implémenté par un tableau statique de taille n_{max} supérieure à la taille nécessaire pour stocker les données. Les n données contenues dans un tel tableau le sont donc simplement entre les indices 0 et $n-1$. Si la taille n_{max} n'est plus suffisante pour stocker toutes les données, on crée un nouveau tableau statique plus grand de taille kn_{max} et on recopie les données dedans.

Toute la subtilité des tableaux dynamiques réside dans la manière de gérer les nouvelles allocations mémoires lorsque le tableau doit être modifié.

R Les tableaux dynamiques sont parfois appelés vecteurs.

R Comme le montre le tableau 2, l'intérêt majeur du tableau dynamique est de proposer un accès direct constant comme dans un tableau statique tout en évitant les surcouts liés à l'ajout d'éléments.

R Certaines opérations sont à coût constant ou linéaire : lorsqu'il n'y a plus de place dans le tableau, il faut bien créer la nouvelle structure adaptée au nombre d'éléments et cela a un coût linéaire $O(n)$. Donc le coût **amorti** en $O(1)$ signifie c'est constant la plupart du temps mais que parfois cela peut être linéaire.

1. mais pas en OCaml!

Opération	Complexité	Raison
Accès à un élément au début	$O(1)$	
Accès à un élément à la fin	$O(1)$	
Accès à un élément au milieu	$O(1)$	
Ajout d'un élément au début	$O(n)$	décaler tous les éléments contigus
Ajout d'un élément à la fin	$O(1)$	amorti : il y a de la place ou pas
Suppression d'un élément au début	$O(n)$	décaler tous les éléments contigus
Suppression d'un élément à la fin	$O(1)$	amorti : il y a de la place, parfois trop

TABLE 2 – Complexité des opérations associées à l'utilisation d'un tableau dynamique.

■ **Exemple 4 — Complexité amortie de l'ajout en fun dans un tableau dynamique.** Pour illustrer la notion de complexité amortie, on choisit un tableau dynamique dont la taille est **doublée** à chaque fois qu'on redimensionnne le tableau. Imaginons qu'on a inséré $n = 2^m$ éléments. À la fin des opérations, on a effectué $C(n)$ opérations, n insertions dont le coût est en :

- $O(1)$ si la taille est suffisante
- $O(i)$ si $i-1$, la taille du tableau avant insertion, est une puissance de 2 : dans ce cas, on crée un nouveau tableau et on recopie les $i-1$ premiers éléments plus le ième. D'où un coût linéaire par rapport à la taille du tableau.

$$C(n) = n \times 1 + \sum_{k=0}^{m} 2^k = n + 2\frac{1-2^{m+1}}{1-2} = n + 2(2^{m+1}-1) = n + 4n = O(n) \tag{1}$$

P En python le type `list` est implémenté par un tableau dynamique mais se comporte bien comme un TAD liste !

Cela a pour conséquence que :

- `L.pop()` et `L.append()` sont de complexité $O(1)$, donc supprimer ou ajouter en fin ne coûte pas cher,
- alors que `L.pop(0)` et `L.insert(0,elem)` sont de complexité $O(n)$ et donc supprimer ou ajouter en tête coûte cher.

Lorsqu'un algorithme doit supprimer ou ajouter en tête, il vaut mieux utiliser une autre structure de données qu'une `list` Python. Dans la bibliothèque `collections`, le type `deque` représente une liste sur laquelle les opérations d'ajout et de suppression en tête ou en fin sont en $O(1)$.

R Rechercher un élément dans un tableau statique ou dans un tableau dynamique présente donc une complexité dans le pire des cas linéaire en $O(n)$: il faut nécessairement balayer tous les éléments si l'élément recherché se trouve en dernière position.

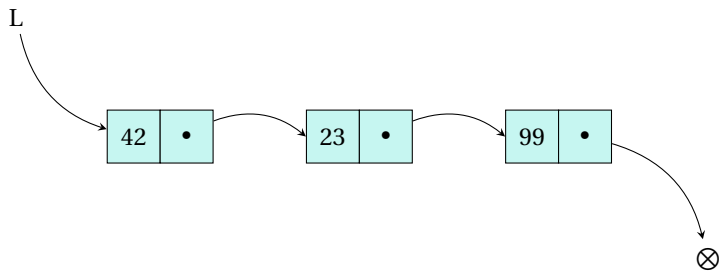

FIGURE 2 – Représentation d'une liste simplement chaînée d'entiers L. L pointe sur le premier élément de la liste. Le dernier pointeur ne pointe sur rien.

E Implémentations des listes

a Listes simplement chaînées

Un élément d'une liste simplement chaînée est une cellule constituée de deux parties :

- la première contient une donnée, par exemple un entier pour une liste d'entiers,
- la seconde contient un pointeur, c'est à dire une adresse mémoire, vers un autre élément (l'élément suivant) ou rien.

Une liste simplement chaînée se présente donc comme une succession d'éléments composites, chacun pointant sur le suivant et le dernier sur rien. En général, la variable associée à une liste simplement chaînée n'est qu'un pointeur vers le premier élément.

Opération	**Complexité**	**Raison**
Accès à un élément au début	$O(1)$	L pointe sur le premier élément
Accès à un élément à la fin	$O(n)$	accès séquentiel
Accès à un élément au milieu	$O(n)$	accès séquentiel
Ajout d'un élément au début	$O(1)$	L pointe sur le premier élément
Ajout d'un élément à la fin	$O(n)$	accès séquentiel
Suppression d'un élément au début	$O(1)$	L pointe sur le premier élément
Suppression d'un élément à la fin	$O(n)$	accès séquentiel

TABLE 3 – Complexité des opérations associées à l'utilisation d'une liste simplement chaînée.

R Rechercher un élément dans une liste chaînée présente donc une complexité dans le pire des cas linéaire en $O(n)$: il faut nécessairement balayer tous les éléments si l'élément recherché se trouve en dernière position.

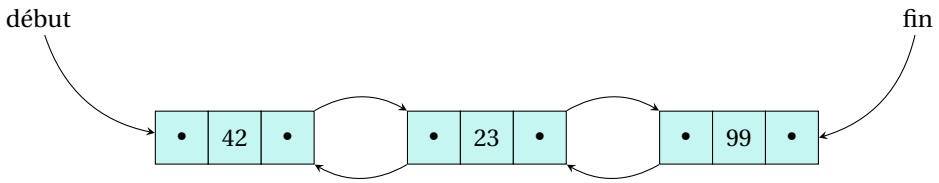

FIGURE 3 – Représentation d'une liste doublement chaînée d'entiers L. On conserve un pointeur sur le premier élément et un autre sur le dernier élément de la liste.

b Listes doublement chaînées

Un élément d'une liste doublement chaînée est une cellule constituée de trois parties :

- la première contient un pointeur vers l'élément précédent,
- la deuxième contient une donnée,
- la troisième contient un pointeur vers l'élément suivant.

Une liste doublement chaînée enregistre dans sa structure un pointeur vers le premier élément et un pointeur vers le dernier élément. Ainsi on peut toujours accéder directement à la tête et à la fin de liste. Par contre, c'est un peu plus lourd en mémoire et plus difficile à implémenter qu'une liste simplement chaînée. Le tableau 4 recense les coûts associés aux opérations sur les listes doublement chaînées.

Opération	**Complexité**	**Raison**
Accès à un élément au début	$O(1)$	pointeur sur le premier élément
Accès à un élément à la fin	$O(1)$	pointeur sur le dernier élément
Accès à un élément au milieu	$O(n)$	accès séquentiel
Ajout d'un élément au début	$O(1)$	pointeur sur le premier élément
Ajout d'un élément à la fin	$O(1)$	pointeur sur le dernier élément
Suppression d'un élément au début	$O(1)$	pointeur sur le premier élément
Suppression d'un élément à la fin	$O(1)$	pointeur sur le dernier élément

TABLE 4 – Complexité des opérations associées à l'utilisation d'une liste doublement chaînée.

F Bilan des opérations sur les structures listes et tableaux

Opération	**Tableau statique**	**Liste chaînée**	**Liste doublement chaînée**	**Tableau dynamique**
Accès à un élément au début	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Accès à un élément à la fin	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Accès à un élément au milieu	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Ajout d'un élément au début	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Ajout d'un élément à la fin	$O(n)$	$O(n)$	$O(1)$	$O(1)$ amorti
Suppression d'un élément au début	$O(1)$	$O(1)$	$O(1)$	$O(n)$
Suppression d'un élément à la fin	$O(n)$	$O(1)$	$O(1)$	$O(1)$ amorti
Recherche d'un élément	$O(n)$	$O(n)$	$O(n)$	$O(n)$

TABLE 5 – Complexité des opérations associées à l'utilisation des listes et des tableaux.

G Demandez le programme

Les chapitres suivants traitent des types abstraits :

- liste,
- tableau,
- arbre,
- pile,
- file,
- tas,
- dictionnaire.