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 donc 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 types composés comme les listes, les tableaux, les arbres, les files, les piles. Ce sont des structures composites qui permettent et de manipuler l'information sous la forme d'ensembles ordonnés ou non.

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 abstrait de données en illustrant ce concept sur les tableaux, les dictionnaires et les listes. Puis le lien avec les implémentations possibles de ces types en structures de données met en évidence la diversité des solutions disponibles.

A Type abstrait de données et structure de données

■ **Exemple 1 — Analogie introductive : de la fonction mécanique à l'architecture physique.** Pour réaliser une fonction mécanique, il est courant de disposer de plusieurs solutions concrètes. Par exemple, si l'on considère un vélo, comment convertir le mouvement de rotation du pédalier en mouvement de rotation des roues? La fonction abstraite recherchée, c'est-à-dire le **quoi**, ce que l'on veut pouvoir faire, est un convertisseur de rotation en rotation. Cette **abstraction** peut être réalisée par un système classique (roues dentées et chaîne métallique), par une courroie polyamide et carbone ou par un cardan. Trois réalisations possibles au moins pour une même abstraction.

Tout comme en conception mécanique on distingue l'abstraction à réaliser de sa réalisation, de la même manière, en informatique, on distingue un type de données que l'on désigne par le terme *Type Abstrait de Données* (TAD) de sa réalisation concrète que l'on désigne par le terme *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 possibles. 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. On y décrit donc le **comment**, c'est-à-dire la manière avec laquelle sont codées les données et les opérations en machine.

R 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 2 — 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 3 — 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 4 — 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.

■ **Exemple 5 — Pile.** Une pile est un TAD composé de type LIFO (Last In First Out) qui gère une collection d'éléments et dont les trois principales opérations sont :

- empiler (push) : ajouter un élément à la collection,
- dépiler (pop) : retirer le dernier élément ajouté à la collection qui n'a pas été retiré,
- consulter le sommet (peek) : consulter l'élément sur le sommet de la pile.

Une pile peut être implémentée à l'aide d'une liste ou d'un tableau.

■ **Exemple 6 — File.** Une file est un TAD composé de type FIFO (First In First Out) qui gère une collection d'éléments et dont les deux principales opérations sont :

- enfiler (push) : ajouter un élément à la fin de la file,
- défiler (get) : retirer l'élément en tête de la file.

Une file peut être implémentée à l'aide d'une liste ou d'un tableau.

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 `a = t[3]` pour la lecture et `t[7] =`

`67.6` pour l'écriture.

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

(opérations) on distingue deux opérations principales caractéristiques :

- l'accès à un élément via un indice entier via un opérateur de type `[]`,
- l'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 à la création du tableau. Il n'est pas possible d'ajouter ou d'enlever des éléments.
2. **dynamique** : la taille du tableau peut varier, on peut ajouter ou enlever des éléments. Dans ce cas, on parle de tableau dynamique.

P En Python, il n'existe pas à proprement parlé de type tableau dans le cœur du langage. Cependant, la liste Python est implémentée par un tableau dynamique et permet donc souvent de pallier ce manque. Néanmoins, pour un calcul numérique efficace, il faut absolument privilégier l'usage des tableaux Numpy qui implémentent le TAD tableau statique.

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 des éléments.

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 alors 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 et/ou retirer la tête de la liste,
- un opérateur pour déterminer et/ou retirer la queue de la liste (tout sauf la tête),
- un opérateur pour accéder au ième élément.

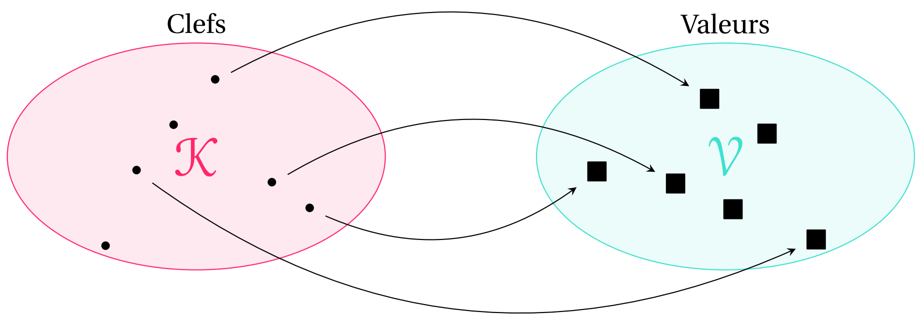

FIGURE 1 – Illustration du concept de dictionnaire

- un opérateur pour accéder au dernier élément de la liste.

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

D TAD Dictonnaire

■ **Définition 5 — TAD Dictionnaire.** Un dictionnaire est une extension du TAD tableau dont les éléments $\mathcal{V}$, au lieu d'être indicés par un entier sont indicés par des clefs appartenant à un ensemble $\mathcal{K}$. Soit $k \in \mathcal{K}$, une clef d'un dictionnaire `D`. Alors `D[k]` est la valeur v de $\mathcal{V}$ qui correspond à la clef k.

On dit qu'un dictionnaire est un **tableau associatif** qui associe une clef `k` à une valeur `v`.

Les opérations sur un dictionnaire sont :

1. rechercher la présence d'une clef dans le dictionnaire,
2. accéder à la valeur correspondant à une clef,
3. insérer une valeur associée à une clef dans le dictionnaire,
4. supprimer une valeur associée à une clef dans le dictionnaire.

R L'intérêt principal d'un dictionnaire est que l'on connaît des implémentation qui permettent de rechercher et d'accéder à un élément en un temps constant $\mathcal{O}(1)$. Rechercher un élément dans une liste est une opération linéaire en $\mathcal{O}(n)$ dans le pire des cas. Dans le cadre d'un tableau, si la recherche par dichotomie est implémentée, alors la recherche d'un élément est logarithmique en $\mathcal{O}(\log(n))$. Si on implémente bien un dictionnaire, tester l'appartenance à un dictionnaire est de complexité constante, ce qui peut accélérer grandement l'exécution d'un algorithme.

P Un dictionnaire relie donc directement une clef, qui n'est pas nécessairement un en-

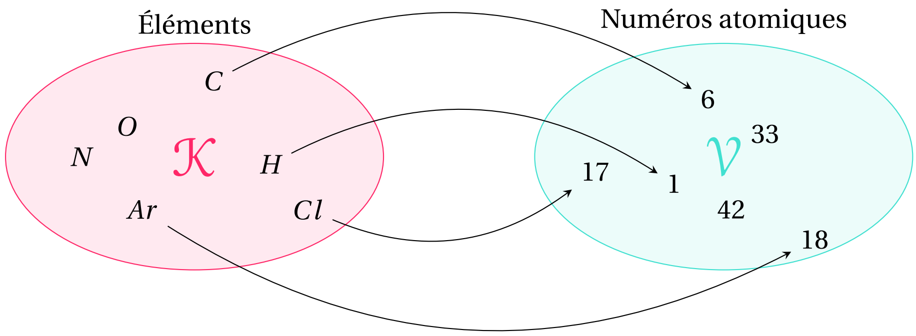

FIGURE 2 – Illustration du concept de dictionnaire, ensembles concrets

tier, à une valeur : pas besoin d'index intermédiaire pour rechercher une valeur comme dans une liste ou un tableau. Par contre, cette clef est nécessairement d'un type immuable. Considérons l'exemple donnée sur l'exemple de la figure 2. On suppose que les éléments chimiques sont enregistrés via une chaîne de caractères : `"C"`, `"O"`, `"H"`, `"Cl"`, `"Ar"`, `"N"`. Soit `d`, un dictionnaire correspondant à la figure 2. Accéder au numéro atomique de l'élément `C` s'écrit : d["C"].

R Un dictionnaire n'est pas une structure ordonnée, à la différence des listes ou des tableaux.

■ **Exemple 7 — Usage des dictionnaires.** Les dictionnaires sont utiles notamment dans le cadre de la programmation dynamique pour la mémoïsation, c'est-à-dire l'enregistrement des valeurs d'une fonction selon ses paramètres d'entrée. Par exemple :

- a-t-on déjà rencontré un sommet lorsqu'on parcourt un graphe ?
- a-t-on déjà calculé la suite de Fibonacci pour $n = 4$?

Répondre à ces questions exige de savoir si pour une clef donnée il existe une valeur.

E 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 3). Ces cases sont capables d'accueillir le type d'élément que contient le tableau.

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 3. Dans la majorité des langages, l'opérateur [] permet alors d'accéder aux éléments[1], par exemple t[3].

1. mais pas en OCaml !

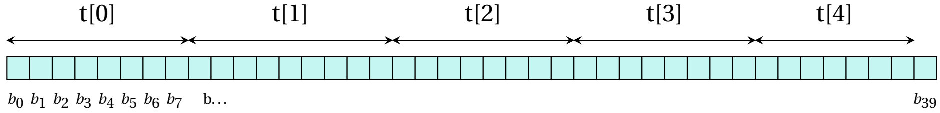

FIGURE 3 – Représentation d'un tableau statique en mémoire. Il peut représenter un tableau t de cinq entiers codés sur huit bits. On accède directement à l'élément i en écrivant $t[i]$.

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.

O En OCaml, les types `Array` sont des tableaux muables, c'est-à-dire les éléments sont modifiables.

■ **Exemple 8 — Tableau statique en OCaml.** En OCaml les tableaux statique sont nommés `Array` et l'API est consultable en ligne. Voici un exemple d'utilisation :

```
let t = [|3;9;0;1;7;4;5;2;6|];;
let n = Array.length t;;
let t1 = Array.make 10 0;; (* construire un tableau de 10 éléments initialisés à 0 *)
print_int t2.(0);;      (* accès au premier élément *)
t2.(3) <- 42;;        (* modification d'un élément *)
let m = Array.make_matrix 3 3 0;;
let t2 = Array.init 10 (fun i -> i);;
```

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 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.

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.

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 qu'il constant la plupart du temps mais que parfois cela peut être linéaire.

■ **Exemple 9 — 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.

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

Cela montre que lorsqu'on insère n éléments dans un tableau dynamique, le coût est proportionnel à n. Donc l'insertion d'un seul élément est en $O(1)$, en complexité amortie.

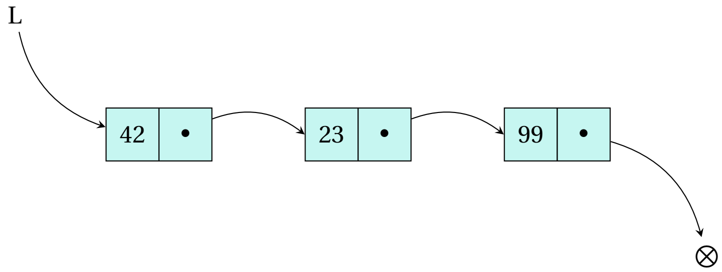

FIGURE 4 – Représentation d'une liste simplement chaînée d'entiers L. L pointe sur le premier élément de la liste. Le dernier pointeur ne pointe sur rien.

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.

F 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.

■ **Exemple 10 — Les listes en OCaml.** Les listes OCaml sont des listes chaînées dont le type est :

```
type 'a list =
        | []
        | (::) of 'a * 'a list
```

Un type `list` est donc soit une liste vide `[]` soit un couple composé d'un élément de type `'a` et d'une liste de `'a`. Le constructeur `::` est noté entre parenthèse car il possède une syntaxe infixe. Ce constructeur permet d'ajouter en tête de liste un élément. On l'utilise ainsi :

```
let l = [1;3;4];;
let l2 = 0::l;; (* l2 vaut [0;1;3;4] *)
```

L'API List OCaml est consultable en ligne. Celle-ci est riche et on y trouve notamment les fonctions ci-dessous :

```
let n = List.length l;;
let head = List.hd l;;
let tail = List.tl l;;
let fourth = List.nth l 3;;
let b = List.mem 3 l;;
```

O En OCaml, les types `List` sont des listes immuables, c'est-à-dire les éléments ne sont pas modifiables, on ne peut pas ajouter, retirer ou modifier un élément d'une liste. Pour réaliser ces opérations, il est nécessaire de construire une autre liste avec un élément en plus, en moins ou un élément différent.

Le filtrage de motif utilise la déconstruction de liste pour parcourir une liste de la tête de liste à la fin comme suit :

```
let rec rm e l = (* supprimer les éléments qui valent e dans l *)
    match l with
        | [] -> []
        | h::t when h = e -> rm e t
        | h::t -> h::(rm e t);;
```

Sur cet exemple, on se rend compte que supprimer un élément d'une liste, c'est en construire une autre identique sans l'élément à supprimer. Pour construire cette autre liste, on en construit en fait plusieurs intermédiaires : le ramasse miettes (Garbage Collecter) d'OCaml efface de la mémoire automatiquement les listes créées qui ne sont plus nécessaires. Sans ce mécanisme, ce processus serait terriblement inefficace d'un point de vue mémoire.

b Listes doublement chaînées

Un élément d'une liste doublement chaînée (cf. figure 5) 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.

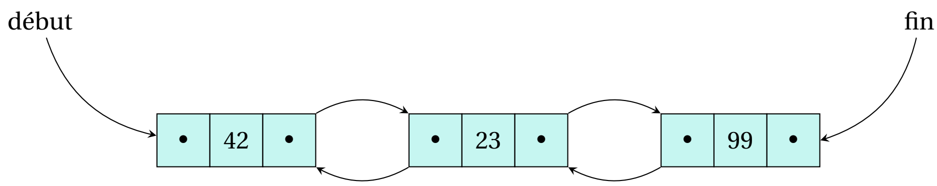

FIGURE 5 – Représentation d'une liste doublement chaînée d'entiers L. On conserve un pointeur sur le premier élément et un autre sur le dernier élément de la liste.

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.

G 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	$O(n)$	$O(1)$	$O(1)$	$O(n)$
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.

H Implémentation d'un TAD dictionnaire

On peut implémenter efficacement un TAD dictionnaire à l'aide

1. des tables de hachage,
2. d'arbres binaires de recherche équilibrés (AVL ou arbres rouges et noirs).

Il existe de nombreuses implémentations possibles du TAD dictionnaire. On peut, par exemple, les implémenter avec des listes, mais l'efficacité n'est pas au rendez-vous. .

Opération	**Table de hachage**	**Arbre de recherche équilibré**
Ajout (pire des cas)	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
Accès (pire des cas)	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
Ajout (en moyenne)	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
Accès (en moyenne)	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

TABLE 6 – Complexité des opérations associées à l'utilisation des tables de hachage ou des arbres pour implémenter un TAD dictionnaire. Les coûts indiqués sont dans le pire des cas ou en moyenne.