

Chapitre 5

STRUCTURES DE DONNÉES (S1)

Notions	Commentaires
Définition d'une structure de données abstraite comme un type muni d'opérations.	On parle de constructeur pour l'initialisation d'une structure, d'accessor pour récupérer une valeur et de transformateur pour modifier l'état de la structure. On montre l'intérêt d'une structure de données abstraite en terme de modularité. On distingue la notion de structure de données abstraite de son implémentation. Plusieurs implémentations concrètes sont interchangeables. La notion de classe et la programmation orientée objet sont hors programme.
Distinction entre structure de données mutable et immuable.	Illustrée en langage OCaml.

Extrait de la section 3.1 du programme officiel de MP2I : « Algorithmes et programmes ».

Notions	Commentaires
Structure de liste. Implémentation par un tableau, par des maillons chaînés.	On insiste sur le coût des opérations selon le choix de l'implémentation. Pour l'implémentation par un tableau, on se fixe une taille maximale. On peut évoquer le problème du redimensionnement d'un tableau.
Structure de pile. Structure de file. Implémentation par un tableau, par des maillons chaînés.	

Extrait de la section 3.2 du programme officiel de MP2I : « Structures de données séquentielles ».

SOMMAIRE

0. Tableaux dynamiques	82
0. Tableaux C vs listes OCaml	82
1. Fonctionnement	83
<i>Ajout dans un tableau dynamique (p. 84). Création, suppression (p. 85).</i>	
2. Analyse de complexité	85
3. Complément mi hors-programme : RETRAIT	85
1. Interfaces et types abstraits.....	87
0. Aspects pratiques	87
<i>Librairies (p. 87). Interface (p. 88). Étapes de la compilation (p. 90).</i>	
1. Aspects théoriques	91
2. Structures de données séquentielles	93
0. Types de base	93
1. Listes linéaire	93
<i>Implémentation par tableaux de longueur fixe (p. 93). Implémentation par tableaux dynamiques (p. 94). Implémentation par listes simplement chaînées (p. 94). Variantes des listes chaînées (p. 97).</i>	
2. Piles	97
<i>Implémentations par tableaux (p. 98). Implémentation par listes simplement chaînées (p. 99).</i>	

3. Files	100
<i>Implémentations par tableaux circulaires (p. 100). (Hors-programme) Amélioration avec des tableaux dynamiques (p. 103). Implémentations par listes simplement chaînées (p. 103). Implémentation par listes doublement chaînées (p. 104). Par double pile (p. 104).</i>	
4. Variantes des files	105
3. Aperçu des structures de données S2	105

0 Tableaux dynamiques

0.0 Tableaux C vs listes OCaml

Les tableaux sont un outil très utile de la programmation impérative¹. Ils permettent de stocker beaucoup d'éléments tout en ayant un accès à chacun de ces éléments en temps constant ; mais ont un inconvénient : ils ne sont pas redimensionnables. Si une fonction utilise un tableau à 200 cases, et qu'elle découvre durant l'exécution que sur certaines entrées il en fallait en fait plus, on est bloqués². Similairement, si on se rend compte qu'il en fallait beaucoup moins, on a gaspillé de la mémoire (et à force de gaspiller de la mémoire, on n'en a plus...)

Exemple.

- On veut faire une fonction qui prend en entrée un entier `n` et renvoie le tableau de ses diviseurs. Combien de cases réserver ? Pas clair. Il est suffisant d'en réserver `n`, bien sûr, mais on sent bien que c'est beaucoup trop.
- On veut faire une fonction prend en entrée un entier `n` et calcule le tableau des nombres premiers inférieurs ou égaux à `n`. On a exactement le même problème qu'avec le point précédent...

Et pourtant, en OCaml, ces exemples ne posent pas de difficulté particulière avec des listes. Voici par exemple pour les diviseurs de `n` :

```

3  (** Construit la liste des diviseurs positifs et >=k de n .
4     * On suppose n >= 0 . *)
5  let rec construit_lst_diviseurs n k =
6      if k > n then
7          []
8      else if n mod k = 0 then
9          k :: (construit_lst_diviseurs n (k+1))
10     else
11         construit_lst_diviseurs n (k+1)
12
13
14  (** Renvoie la liste des diviseurs positifs de n .
15     * On suppose n >= 0 . *)
16  let lst_diviseurs n =
17      construit_lst_diviseurs n 1

```



Les fonctions `lst_diviseurs` et `lst_preiers` renvoient exactement les éléments attendus, et ne surallouent aucune case : les listes créées ont exactement la même taille.

Mais à l'inverse, l'inconvénient des listes est que l'on ne peut pas accéder à un élément par son indice... et quand bien même on recoderait une fonction qui calcule le *i*ème élément d'une liste, elle serait en temps $\Theta(i)$ et non en temps constant.

1. Rappel : la **programmation impérative** est la méthode de programmation consistant à écrire un programme en décrivant une succession de modifications de la mémoire. C'est ce que l'on fait en C, par exemple.

2. Ici j'utilise 200 comme exemple, et on pourrait m'objecter que `malloc` permet de créer des "tableaux" à `n` cases. C'est vrai, mais si la fonction crée un tableau à `n` cases et se rend compte qu'il en fallait n^2 sur certaines entrées spécifiques, le problème reste le même.

Les **tableaux dynamiques** sont une structure de données qui offre le meilleur des deux mondes : ils permettent un accès à chaque élément en temps constant, et sont redimensionnables ce qui permet de les agrandir quand ils ne sont plus assez grand !

Remarque. Si vous connaissez les listes Python, ce sont des tableaux dynamiques³ !

0.1 Fonctionnement

L'objectif de nos tableaux dynamiques est de :

- Pouvoir accéder aux éléments par leur indice et les modifier en temps constant.
- Pouvoir ajouter un élément *à la fin* en temps constant⁴
- (Et aussi pouvoir créer et supprimer un tableau dynamique)

L'idée est de faire un tableau volontairement trop grand et de n'en utiliser que les premières cases. Comme ça, pour ajouter un élément à la fin, il suffit d'utiliser la prochaine case libre :

FIGURE 5.1 – Ajout dans un tableau dynamique, cas facile

Définition 1 (Tableaux dynamiques, structure de base).

Un tableau dynamique est composé de 3 données :

- le tableau en lui-même. Dans ce cours, on le nommera `arr` .
- son nombre réel de cases, c'est à dire le nombre de cases du tableau qui existent dans la mémoire. Dans ce cours, on le nommera `len_max` .
- son nombre de cases utilisées. Dans ce cours, on le nommera `len` .

Les `len` éléments stockés dans le tableau dynamique sont stockés, dans l'ordre, dans les `len` premières cases de `arr` .

Remarque. En C, `arr` ne sera pas un tableau mais une zone mémoire créée avec `malloc`. Dans cette partie, j'utilise le terme « tableau » pour simplifier. De plus, mis à part la création/destruction qui est différente, un tableau et une zone allouée s'utilisent de la même façon...

Cette façon de faire répond à presque toutes nos contraintes : on peut accéder à un élément par son indice en temps constant, modifier l'élément qui se trouve à un indice donné en temps constant ; on peut aussi ajouter un élément en temps constant lorsque `len < len_max` . Reste à définir comment ajouter un élément lorsqu'il n'y a plus de cases vides.

3. En un peu plus compliqué. Les listes Python sont une petite pépite de bonnes idées mises bout à bout pour que ce soit très efficace.

4. Ou au début, c'est symétrique.

0.1.0 Ajout dans un tableau dynamique

Une première idée serait de créer un nouveau tableau ayant une case de plus, de recopier l'ancien dedans, et d'ajouter l'élément à la fin. Hélas, cela n'est pas très efficace en temps :

FIGURE 5.2 – Ne faites pas l'AJOUT comme ça !!

Avec cette méthode, à partir du moment où le tableau est rempli, chaque ajout se fait en temps $O(\text{len_max})$. Ce n'est clairement pas efficace. On pourrait essayer de l'améliorer en créant un nouveau tableau non de longueur $\text{len_max}+1$ mais de longueur $\text{len_max}+10$, cependant cela ferait qu'1 opération sur 10 serait en temps linéaire (les 9 autres en temps constant), et donc chaque paquet de 10 opérations seraient en temps $O(\text{len_max} + 9.O(1) = O(\text{len_max}))$. Bref, si l'on prévoit de faire plusieurs ajouts, cela ne fonctionne pas très bien. Idem pour d'autres valeurs fixées de 10 : 1000, 10000, etc ont tous le même problème⁵.

La solution est de ne pas agrandir en rajoutant un nombre fixe de cases, mais un nombre variable : en fait, on va simplement *doubler* la longueur maximale !

FIGURE 5.3 – AJOUT successifs, avec doublements du tableau

5. Des petit-es malin-es objecteront que avec 10^9 on est à peu près bons, car il est rare que l'on fasse 10^9 ajouts. C'est vrai, mais : 1) cela peut arriver, et 2) vous voulez vraiment que chaque tableau dynamique demande au moins 1Go de mémoire ?

La fonction AJOUT qui ajoute un élément à un tableau dynamique est donc la suivante :

Fonction Ajout

Entrées :

- un tableau dynamique composé de `arr`, `len` et `len_max`. On les considère passés par référence.
- `x` un élément à ajouter à la fin du tableau

Sorties : rien, mais a comme effet secondaire de modifier le tableau pour y ajouter `x`

```

1 si len < len_max alors
2   arr[len] ← x
3   len ← len + 1
4 sinon
5   new_arr ← tableau de longueur 2*len_max
6   // Recopier arr dans new_arr
7   pour chaque i ∈ [0; len[ faire
8     new_arr[i] ← arr[i]
9   // Remplacer arr par new_arr
10  arr ← new_arr
11  len_max ← 2*len_max
12  // Ajouter x
13  arr[len] ← x
14  len ← len + 1

```

Remarque. Notez que les lignes 2-3 et 10-11 sont les mêmes : on peut réécrire ce code sous la forme « si `arr` est entièrement rempli, le remplacer par `new_arr` ». Ensuite, dans tous les cas, ajouter `x` ».

0.1.1 Création, suppression

Pour supprimer un tableau dynamique, il suffit de supprimer `arr`, `len` et `len_max`.

Pour créer un tableau, il suffit de créer ces trois données, avec quand même un point précis : même si l'on crée un tableau dynamique initialement vide, il faut tout de même initialiser `len_max` à au moins 1 (et donc créer `arr` comme ayant au moins 1 case). En effet, si on prend 0, comme AJOUT double la longueur maximale... deux fois zéro ça fait zéro.

0.2 Analyse de complexité

Analyser la complexité de cette version de AJOUT n'est pas aisé : la plupart des appels à la fonction coutent $O(1)$ (quand on effectue uniquement les lignes 2-3), mais certains $O(\text{len})$ (les lignes 5-11). On va en fait analyser le coût d'une *succession* d'appels à AJOUT. Faire cette analyse est l'objet de la **complexité amortie** : cf cours sur la complexité, section complexité amortie.

On y prouve que :

Proposition 2 (Complexité amortie de AJOUT).

Une succession de n ajouts coûte dans le pire des cas, au total, $O(n)$. Autrement dit, le coût amorti de AJOUT est $O(1)$.

(Précisons que cette complexité amortie est bien celle d'un pire des cas d'une succession quelconque de AJOUTE.)

Démonstration. Cf cours complexité amortie (3 preuves différentes y sont données). □

0.3 Complément mi hors-programme : RETRAIT

Notre opération AJOUT est l'équivalent du `append` de Python. Si vous connaissez Python, vous savez qu'une autre opération est souvent utilisée : `pop`, qui supprime et renvoie le dernier élément. Dans ce

cours, on la nommera RETRAIT.

Il est en soi très facile de l'implémenter : enlever l'élément d'indice $\text{len}-1$ (et faire une erreur s'il n'y a aucun élément). Mais on voudrait faire mieux : pouvoir de temps en temps réduire `len_max`, afin de récupérer la mémoire qui ne sert plus.

La première idée, qui ne marche pas, est de faire le symétrique de AJOUT : lorsque la longueur est inférieure à la moitié de la longueur maximale, créer un tableau deux fois plus petit, etc etc. En soi, l'idée est très bonne : mais elle s'agence mal avec AJOUT. En effet, juste après un AJOUT qui « double » le tableau, on obtient un tableau presque à moitié vide. Ainsi, faire un RETRAIT juste après cet AJOUT mène à une contraction du tableau, refaire un AJOUT ensuite une extension, etc : on a une suite d'opération au sein de la quelle AJOUT et RETRAIT ne seraient *pas* en temps constant amorti. Bref, ça va pas.

FIGURE 5.4 – Pourquoi il ne faut pas contracter le tableau à la moitié

Il faut en fait « éloigner » le pire cas de RETRAIT (la contraction, qui crée un tableau plus petit) du pire cas de AJOUT. Pour ce faire, une solution simple est de **diviser par 2 la longueur maximale du tableau lorsqu'il est aux 3/4 vide**, et non lorsqu'il est à moitié vide :

FIGURE 5.5 – Fonctionnement de la suppression

Fonction Retrait**Entrées :**

- un tableau dynamique composé de `arr`, `len` et `len_max`. On les considère passés par référence.

Sorties :

- si `len = 0`, indique une erreur.
- sinon, le dernier élément de `arr` (c'est à dire le plus à droite). A également pour effet secondaire modifier le tableau pour y enlever cet élément.

```

1 si len = 0 alors renvoyer une erreur
   // Retirer le dernier élément x
2 sortie ← arr[len]
3 len ← len-1

   // Si besoin, contracter arr
4 si len ≤ len_max/4 et len_max ≥ 4 alors
5   new_arr ← tableau de longueur len_max/2
6   pour chaque i ∈ [0; len] faire
7     new_arr[i] ← arr[i]
8   arr ← new_arr
9   len_max ← len_max/2

   // Ne pas oublier de renvoyer la sortie
10 renvoyer sortie

```

Proposition 3 (Complexité amortie de RETRAIT).

En codant RETRAIT et AJOUT comme présentés ici, on obtient pour ces deux opérations une complexité amortie $O(1)$.
(Précisons que cette complexité amortie est bien celle d'un pire des cas d'une succession quelconque de RETRAIT et AJOUT.)

Démonstration. Cf TD. □

1 Interfaces et types abstraits

1.0 Aspects pratiques

Remarque. Cette sous-partie du cours contient de nombreuses approximations. Ces concepts seront définis de manière plus précise et plus approfondie en cours de génie logiciel (en école).

1.0.0 Librairies

Lorsque l'on programme, il est usuel de découper son code en plusieurs fichiers : l'idée est que chaque fichier implémente « un paquet » cohérent de fonctions.

Définition 4 (Librairie et code client).

Un fichier constitué de plusieurs fonctions ayant pour but d'être utilisées dans d'autres fichiers est appelé une **librairie**.

Un code qui utilise une librairie est appelé un **code client**.

On appelle un tel paquet une **librairie**.

Exemple.

- `stdio` est une librairie C. Son nom signifie **standard in-out** : c'est la librairie standard (c'est à dire la librairie « officielle ») qui contient les fonctions de lecture (par exemple `scanf`) et d'écriture/affichage (par exemple `printf`) (`out`).
- `stdlib` (librairie standard généraliste qui contient notamment `malloc` et `free`), `stdbool` (librairie standard qui contient le type `bool`), `stdint` (librairie standard qui contient des types d'entiers sur 8/16/32/48/64 bits) ou encore `assert` (contient la fonction `assert`) sont des librairies C.

Remarque.

- Le terme « librairie » est un anglicisme qui s'est imposé. Vous croiserez peut-être à la place la traduction **bibliothèque logicielle**.
- En OCaml, on parle plutôt de **module**. En réalité, un module est un légèrement différent d'une librairie (c'en est une version évoluée), mais je ne ferai pas la différence à notre niveau.

Exemple.

- `List` est le module des listes en OCaml.
- `Printf` est le module qui contient la fonction `printf` (et ses parentes) en OCaml.
- (et nous en verrons d'autres cette année et l'an prochain)

Pourquoi programmer en plusieurs fichiers ?

- Ne pas tout recoder à chaque fois : une fois que `stdlib` est codée, pas besoin de la recoder ! Il suffit de l'inclure pour pouvoir s'en servir. Et ça tombe bien, recoder `malloc` n'est vraiment pas un exercice facile...
- Accélérer la compilation : notre code n'appelle que les librairies dont il a besoin, et rien de plus. Ainsi, il y a moins de choses à compiler, et on compile donc plus vite.
- Garder chacun des fichiers relativement courts : il est plus simple de trouver un bogue si on sait qu'il est dans tel fichier de 2000 lignes plutôt que dans tel fichier de 50000 lignes. Idem si l'on veut aller relire ou modifier une fonction particulière.
- Segmenter et organiser le travail : on programme une chose à la fois.
On peut également donner chacun des fichiers à une équipe de programmeurs/programmeuses différentes : chaque équipe se spécialise alors dans son fichier, avance efficacement dessus (pendant que les autres avancent efficacement sur le leur), et le tout est terminé plus rapidement.

En résumé, c'est très confortable !

1.0.1 Interface

Dans votre ordinateur, vous *n'avez pas* le fichier `stdlib.c`, c'est à dire le code source de `stdlib`. Vous avez à la place uniquement une version compilée de lui-ci nommée `stdlib.o`, ainsi que son interface `stdlib.h` (sounds familiar ?).

Définition 5 (Interface).


L'**interface** (en anglais : **A**pplication **P**rogramming **I**nterface, ou **API**) d'une librairie est un fichier qui décrit et abstrait son contenu. On y trouve les signatures des variables/fonctions/types déclarés la librairie et la spécification de ceux-ci ; ainsi aussi la liste des dépendances de la librairie (c'est à dire la liste des autres librairies qu'elle utilise).

Définition 6 (Organisation des fichiers d'une librairie).

En C, les interfaces sont appelées des *headers*. Si le fichier source s'appelle `truc.c`, l'interface s'appelle `truc.h` et la version compilée du code source `truc.o`.

En OCaml, `truc.ml` a pour interface `truc.mli` et pour version compilée `truc.o`.

Exemple. Voici un extrait de `dynArray.h`, une interface pour une librairie de tableaux dynamiques que nous coderons en TP :

 **dynArray.h**

```

6  #include <stdlib.h>
7  #include <stdbool.h>
8  #include <stdio.h>
9
10
11 /* Structure d'un tableau dynamique */
12 struct dynArray_s {
13     int* arr;           // pointeur vers la zone contenant les cases
14     unsigned len;       // nombre de cases utilisées de arr
15     unsigned len_max;   // nombre de cases de arr
16 };
17 typedef struct dynArray_s dynArray;
18
19
20 /* Constructeurs */
21
22 /** Crée un tableau dynamique de longueur n, dont les cases sont initialisées
23     ↪ à x */
24 dynArray dyn_create(unsigned len, int x);
25
26 /** Libère le contenu de d */
27 void dyn_free(dynArray* d);
28
29 /** Crée un tableau dynamique qui contient les len valeurs pointées par ptr
30     ↪ */
31 dynArray dynarray_of_array(unsigned len, int const* ptr);

```

Dans cet exemple, on peut voir que la librairie `dynArray` :

- utilise les librairies `stdlib`, `stdbool` et `stdio`.
- définit un type nommé `struct dynArray_s` (aussi nommé simplement `dynArray`).
- et définit les fonctions `dyn_create`, `dyn_free` et `dynarray_of_array` dont les prototypes sont indiqués. Il y a également une documentation de ces fonctions.

Pourquoi faire une interface ?

- Lorsque l'on veut utiliser une librairie, savoir *comment* la librairie fonctionne ne nous intéresse guère. Ce que l'on veut savoir, c'est quelles sont les fonctions que l'on peut appeler et ce qu'elles font : exactement ce que l'interface nous dit.

Par exemple, vous n'êtes jamais allés voir le code source de `printf` ... et heureusement.⁶

- Une interface permet de cacher certaines fonctions, en les omettant de l'interface. C'est très utile quand le code source utilise des fonctions « auxiliaires ».

Par exemple, le code source ma fonction `dyn_create` utilise une fonction `max`. Cependant, coder `max` n'est pas le but de cette librairie, et je ne veux pas alourdir l'interface en y mettant cette fonction : je ne la mets donc pas.

C'est d'autant plus utile lorsque la fonction auxiliaire est un peu incompréhensible, car elle réalise une tâche extrêmement spécifique que l'on ne comprend qu'avec le code source du tout sous les yeux.

6. Si vous êtes curieux-se, `printf` appelle `vprintf` qui appelle `vfprintf` qui fait tout le travail. Le code source de celle-ci est disponible ici : <https://github.com/lattera/glibc/blob/master/stdio-common/vfprintf.c> (la fonction commence à la ligne 1236 et fait environ 400 lignes, globalement très hors-programme).

- Une interface décrit uniquement la signature des fonctions : en particulier, on peut changer complètement la façon précise de les implémenter sans que cela n'impacte les codes clients. En d'autres termes, on peut faire une mise à jour de la librairie sans qu'il n'y ait à adapter les codes clients : c'est plutôt très bien.

Remarque. Certaines interfaces indiquent la complexité des fonctions, d'autres noms : c'est un débat. Certain-es informaticien-n-es demandent à ce que les complexité apparaissent car c'est important pour évaluer la performance du code client ; tandis d'autres demandent à ne pas les indiquer car les mettre revient généralement à forcer une implémentation spécifique et donc à s'interdire de changer le fonctionnement « sous le capot » plus tard.⁷

1.0.2 Étapes de la compilation

Cette sous-sous-partie est volontairement très vague/flou. Vous en comprendre plus les termes avec de l'expérience.

Définition 7 (Étapes de compilations).

Un compilateur effectue les étapes suivantes quand il compile :

- Pré-compilation : une première passe qui a pour but de préparer les fichiers. C'est notamment dans cette phase qu'en C les `sizeof(type)` sont remplacés par leur valeur, et (hors-programme) que les `#define` sont résolus.
- Analyse lexicale, syntaxique et sémantique du code : il s'agit de « lire » le code et de comprendre son « organisation interne ». Cf cours de MPI pour l'analyse lexicale et syntaxique (l'analyse sémantique est hors-programme).
- Génération de code intermédiaire puis de code objet : chacun des fichiers du code est, individuellement, transformé en une version compilée de ses fonctions. Cette version compilée est le fichier `.o`. Cependant, les appels de fonctions ne sont pas encore fonctionnels dans ces `.o` : lorsqu'une fonction A appelle une fonction B, l'appel ne fonctionne pas encore : il y a à la place quelque chose comme « TODO : ici insérer un appel à B ».
- Édition de liens : les appels de fonctions sont rendus fonctionnels, et le fichier est rendu exécutable.

C'est notamment l'étape où le compilateur doit « relier » différents codes sources entre eux pour la première fois.

FIGURE 5.6 – Résumé du rôle des différents fichiers

7. À titre personnel, je suis plutôt de celles et ceux qui veulent voir les complexités ; quitte à ce que ce soit avec la mention « cette complexité pourra changer à l'avenir ». Mon avis personnel n'a cependant que très peu de valeur.

Remarque. (Vous n’avez pas à comprendre le détail de chaque étape pour comprendre ce qui suit :)

- les librairies sont généralement stockées dans votre ordinateur avec uniquement le fichier objet `.o` et l’interface `.h` (ou `.mli`). Ainsi, quand on veut utiliser une librairie, le compilateur n’a pas besoin de refaire toutes les premières étapes sur la librairie ! Il n’aura à faire que l’édition de liens, et la compilation est donc grandement accélérée.
- (Hors-programme) Vous avez peut-être déjà entendu parler des fichiers *shared object* `.so` sous Linux, ou des `.dll` sous Windows. Ce sont une sorte particulière de fichiers objets, c’est à dire une version compilée d’une librairie. C’est en réalité sous cette forme que sont distribuées les librairies, plutôt que comme des simples `.o`.

1.1 Aspects théoriques

Définition 8 (Structure de données).

Une **structure de données** est une façon d’organiser des données et de les manipuler efficacement.

Exemple. En décrivant les tableaux dynamiques en section 1, nous avons décrit une structure de donnée.

Définition 9 (Type abstrait, signature).

Un **type abstrait**, aussi appelé **structure de donnée abstraite**, est une considération abstraite d’une structure de donnée. Elle est décrite par sa **signature**, c’est à dire par la donnée de :

- l’identifiant (le nom) du type abstrait.
- les autres types abstraits dont il dépend.
- les identifiants de constantes du type.
- la signature des opérations que l’on peut effectuer sur le type.

Exemple. Voici deux signatures :

(a) Interface du type abstrait Bool

(b) Interface du type abstrait Tableau Dynamique

FIGURE 5.7 – Deux exemples d’interface

Définition 10 (Signature vs documentation (syntaxe vs sémantique)).

La signature explique quelles sont les « choses » que l'on peut écrire à l'aide du type (quelles fonctions on peut appliquer et dans quels cas). Mais elle n'explique pas ce que *font* ces « choses » ! En termes scientifiques, on dit qu'une signature décrit uniquement la **syntaxe**, c'est à dire les règles d'écritures.

Pour savoir ce que font ces « choses », il faut donner de la **sémantique**, c'est à dire **documenter** les constantes et les opérations du type.

Exemple. Voici des exemples de documentation pour les 3 opérations des booléens :

- **non** : renvoie la négation d'un booléen, c'est à dire que $\text{Non}(\text{vrai}) = \text{faux}$ et réciproquement.
- **ou** : renvoie vrai si et seulement si au moins l'un de ses deux arguments vaut vrai .
- **et** : renvoie vrai si et seulement si ses deux arguments valent vrai .

Remarque.

- La documentation peut aussi prendre une forme très mathématique. C'est notamment utile lorsque l'on veut automatiser les preuves de programme. Voici par exemple un axiome vérifié par Longueur :
 $\forall n : \text{Entier}, \forall x : \text{Élément}, \text{longueur}(\text{creer}(n, x)) = n$.
 Dans la plupart des cas, sauf ambiguïté, on préfère donner une documentation en français plutôt qu'en mathématique pour des raisons évidentes de lisibilité.
- D'un point de vue pratique, une erreur de *syntaxe* est quelque chose qui est détecté à la compilation (par exemple une erreur de type), tandis qu'une erreur de *sémantique* crée un bogue dont on ne se rend compte qu'en testant le code.
- En langage naturel (en français si vous êtes francophone), une erreur de syntaxe correspond à une erreur d'orthographe/grammaire, tandis qu'une erreur de sémantique correspond à une phrase qui ne veut rien dire.

Définition 11 (Constructeur, accesseur, transformateur).

n classe les différentes opérations d'un type abstrait en :

- **constructeur** : ce sont les opérations qui créent ou détruisent une de données.
- **accesseur** : ce sont les opérations qui renvoient une information sur la structure de données (sans la modifier).
- **transformateur** : ce sont les opérations qui modifient une structure de donnée existante.

Exemple. Dans le type abstrait `Tableau dynamique`, `Créer` est un constructeur, `Longueur` un accesseur et `Ajout` un transformateur.

Remarque. Les signatures des transformateurs peuvent être écrites de deux façons : soit d'une façon impérative (ils ne renvoient pas de structure de donnée, mais modifient la structure de donnée passée en argument), soit de façon fonctionnelle (ils ne modifient pas leur argument mais renvoient le nouvel état de la structure de données).

Exemple.

- « `Ajout : Tableau Dynamique × Élément → Rien` » est une signature impérative (et la documentation doit préciser que `Ajout` modifie celui passé en argument).
- « `Ajout : Tableau Dynamique × Élément → Tableau Dynamique` » est une signature fonctionnelle (et la documentation doit préciser l'absence d'effets secondaires).

2 Structures de données séquentielles

Une structure de donnée est dite **séquentielle** lorsqu'elle range les éléments les uns après les autres, dans un certain ordre.

Exemple. Les tableaux C ou les listes OCaml sont des structures de données séquentielles.

L'objectif de cette section est de présenter différents types abstraits de structures de données séquentielles, et une ou plusieurs implémentations.

2.0 Types de base

On a déjà vu comment sont implémentés les entiers, flottants, booléens : dans ce cours, je les considérerai comme des types de base.

On a également déjà vu certains types construits à l'aide d'autres types : par exemple les pointeurs (on utilise des pointeurs vers un élément d'un autre type) ou les tableaux (des tableaux d'un autre type).

Remarque. Notez qu'il existe en fait de nombreux types de tableaux : les tableaux d'entiers, les tableaux de booléens, les tableaux de etc... On dit que le type tableau est **paramétré** par le type de son contenu.

2.1 Listes linéaire

Avant-propos : le terme « liste » est victime d'une forte polyésmie⁸. Il veut dire des choses différentes ici et là. Dans ce cours, par « liste » je désigne spécifiquement une liste linéaire :

Définition 12 (Liste linéaire).

Une **liste linéaire** est une suite finie, éventuellement vide, d'éléments repérés selon leur rang :

$\ell = [\ell_0; \dots; \ell_{n-1}]$.

Si E est l'ensemble des éléments, l'ensemble \mathbb{L} est défini récursivement par :

$$\mathbb{L} = \emptyset + E \times \mathbb{L}$$

Définition 13 (Signature de Liste linéaire).

La signature (fonctionnelle) du type abstrait Liste Linéaire est :

- Type : Liste Linéaire (abrégié List ci-dessous)
- Utilis : Entier, Élément
- Constantes : `[]` est une liste linéaire
- Opérations :
 - `longueur` : `List` → Entier
 - `acces_ieme` : `List` × Entier → Élément
 - `supprime_ieme` : `List` × Entier → List
 - `insere_ieme` : `List` × Entier × Élément → List

Les trois dernières fonctions, respectivement : renvoient l'élément d'indice pris en argument, insère un élément de sorte à ce qu'il soit à l'indice pris en argument, ou supprime l'élément d'indice pris en argument.

2.1.0 Implémentation par tableaux de longueur fixe

Pour cette implémentation, on fixe un majorant `len_max` de la longueur de toute liste linéaire.

8. Notamment à cause de Python, qui utilise le terme pour désigner ses tableaux dynamiques... et encore, le terme était déjà polyésmique avant Python.

On représente alors une liste linéaire par un tableau à `len_max` cases, dont on n'utilise que les premières pour stocker les éléments les uns après les autres. Il faut également stocker `len` la longueur réelle de la liste. Autrement dit, on fait comme des tableaux dynamiques, mais sans redimensionner :

FIGURE 5.8 – Liste linéaire dans un tableau de longueur fixe

Proposition 14 (Complexités des listes linéaires par tableaux).

ne telle implémentation permet d'obtenir les complexités suivantes :

- `[]` : se calcule en $O(1)$ s'il ne faut pas initialiser les cases du tableau, et en $O(\text{len_max})$ sinon.
- `longueur` : $O(1)$
- `acces_ieme` : $O(1)$
- `insere_ieme` : $O(\text{len} - i)$ avec i l'indice auquel on insère.
- `supprime_ieme` : $O(\text{len} - i)$ avec i l'indice auquel on supprime.

Démonstration. Les seules complexités non triviales sont les deux dernières. Je présente ici uniquement la première, l'autre étant similaire.

Pour insérer en position i , on « décale » d'une case tous les éléments d'indice $> i$: il y a $\text{len} - i$ éléments à décaler, chaque décalage se fait en temps constant (1 écriture dans une case du tableau + 1 calcul d'indice), donc tout ce décalage est en $O(\text{len} - i)$. Il ne reste qu'à écrire l'élément dans la case voulue, ce qui se fait en temps constant. \square

2.1.1 Implémentation par tableaux dynamiques

En utilisant les idées des tableaux dynamiques, on peut utiliser des tableaux "redimensionnables" pour améliorer l'implémentation précédente. Cela permet de manipuler des listes linéaires dont la valeur finit par dépasser le `len_max` initial.

Comme on l'a vu dans le chapitre sur les tableaux dynamique, en amorti les complexités restent les mêmes.

2.1.2 Implémentation par listes simplement chaînées

Définition 15 (Liste simplement chaînée).

Une **liste simplement chaînée** est une structure de données composée de **cellules** (aussi appelées **maillons**, ou parfois **noeuds**) reliées entre elles. Chaque cellule stocke deux informations :

- L'élément de la liste stockée dans cette cellule.
- Un pointeur vers la cellule de l'élément suivant.

FIGURE 5.9 – Organisation d'une liste simplement chaînée

Exemple.

FIGURE 5.10 – Une liste chaînée particulière

Remarque.

- Sur mes schémas, je ne représente pas les cellules comme étant « proprement les unes après les autres ». C'est pour vous rappeler qu'en pratique, les cellules sont allouées sur le tas mémoire, et que l'on a aucune garantie de position dans le tas mémoire.
- On sait que l'on a atteint la fin d'une liste lorsque le pointeur vers la prochaine cellule a une valeur particulière. En C, c'est généralement NULL (le pointeur qui ne pointe sur rien).
- On type souvent une liste simplement chaînée comme étant un pointeur vers une cellule. Ainsi, pour donner une liste simplement chaînée, il suffit de donner un pointeur vers la cellule de tête. Pour donner la liste vide, on donne le pointeur NULL (ou autre valeur particulière fixée).
- C'est ainsi que sont implémentées les listes en OCaml !! Cela correspond d'ailleurs à ce que l'on a vu en TP, où l'on a dit qu'une liste correspond à l'un de ces deux motifs :

```
1 | []      (* liste vide *)
2 | t :: q  (* valeur de tête suivi d'un accès à la queue *)
```



En particulier, en OCaml, passer une `'a list` en argument revient exactement à passer un pointeur, c'est à dire que cela se fait en temps constant.

Proposition 16 (Complexité des opérations sur une liste linéaire).

Cette implémentation permet d'obtenir les complexités suivantes pour une liste linéaire à len éléments :

- `[]` : $O(1)$
- `longueur` : $O(\text{len})$
- `acces_ieme` : $O(i)$
- `insere_ieme` : $O(i)$ avec i l'indice auquel on insère.
- `supprime_ieme` : $O(i)$ avec i l'indice auquel on supprime.

Démonstration. Respectivement :

- il suffit de renvoyer le pointeur NULL
- il faut parcourir toutes les cellules, en mémorisant le nombre de cellules. Chaque cellule se traite en temps constant (incrémenter le nombre de cellule vues puis aller à la suivante).
- idem, mais on ne les parcourt que jusqu'à atteindre la cellule d'indice i (où l'on renvoie alors la valeur stockée, en temps constant).
- Similaire au précédent, à ceci près que :
 - 0) On accède à la cellule d'indice i .
 - 1) On crée une nouvelle cellule, que l'on fait pointer vers la queue de la cellule d'indice i .
 - 2) on modifie le chainage de la cellule d'avant. Si l'on ne veut pas modifier la liste prise en argument, on doit recréer tout le début de la liste.

FIGURE 5.11 – Insertion dans une liste chaînée, sans modifier l'originale

- De même que le précédent.

□

Remarque. Un gros avantage des listes simplement chaînées est qu'elles réduisent la duplication en mémoire lorsque l'on « étend » de différentes façons des listes. Par exemple :

```

1 let lst = [4; 5; 6; 7; 8; 9]
2 let adora = 3 :: lst
3 let catra = 10 :: 20 :: lst
4 let glimmer = 55 :: lst

```



correspond en mémoire à :

FIGURE 5.12 – Illustration de la non-duplication des queues des listes OCaml

Cette propriété est notamment utile lorsque l'on fait un algorithme qui essaye différentes façons d'étendre une liste ; par exemple les algorithmes de retour sur trace (ou *backtracking* en anglais, que nous verrons au S2).

Remarque. (Petit complément au programme) De même, la fonction `List.append` qui concatène deux listes (c'est à dire les « colle ») est en temps et espace linéaire en la longueur de la première liste. En particulier, si on peut éviter d'y faire appel plusieurs fois, c'est (bien) mieux.

Pro-tip (au programme aussi) : `List.append l0 l1` se note aussi `l0 @ l1` .

Exemple.

```
1 let l0 = [1; 2]
2 let l1 = [2; 3; 4; 5]
3 let l2 = l0 @ l1
```



correspond en mémoire à :

FIGURE 5.13 – Illustration de la concaténation de listes OCaml

2.1.3 Variantes des listes chaînées

Voici quelques variantes qui existent (il en existe bien d'autres) :

- On peut mémoriser une « meta-donnée » en plus de la liste linéaire, typiquement un pointeur vers le début et la fin :

- La liste peut-être doublement chaînée, c'est à dire que chaque cellule retient la cellule d'avant et celle d'après.
- Le chaînage peut-être circulaire, c'est à dire que la dernière cellule pointe sur la première.
- Et bien d'autres que vous croiserez peut-être.⁹

2.2 Piles

Une pile est un cas particulier de liste linéaire. On abandonne l'idée d'insérer ou de supprimer n'importe où, et même d'accéder n'importe où. En échange, on demande une *garantie* sur l'ordre des éléments :

Définition 17 (Pile).

Une **pile** (anglais : **stack**) est une liste linéaire qui garantit que le dernier élément inséré sera le premier élément supprimé. En particulier, on ne peut pas insérer ou supprimer n'importe où. On dit qu'une pile vérifie le principe **LIFO** : Last In First Out (dernier entré, premier sorti).

9. Techniquement, on peut voir les arbres (S2) comme le cas où il peut y avoir plusieurs cellules suivantes... je préfère voir les listes chaînées comme un cas particulier des arbres.

Exemple. Une pile de linge à trier, ou une pile de copies à corriger, ou n'importe quel autre empilement de la vie réelle.

Exemple.

FIGURE 5.14 – Empilages et dépilages successifs sur une pile

Remarque. La pile mémoire est une pile.

Définition 18 (Signature des piles).

Voici la signature (impérative) des piles, un peu documentée :

- Type : Pile
- Utilise : Élément, Booléen
- Constantes : `pile_vide`
- Opérations :

<ul style="list-style-type: none"> – <code>empile</code> : Pile \times Élément \rightarrow rien – <code>depile</code> : Pile \rightarrow Élément – <code>sommet</code> : Pile \rightarrow Élément – <code>est_vide</code> : Pile \rightarrow Bool 	<p><i>Cette fonction ajoute un élément sur la pile (et la modifie donc).</i></p> <p><i>Cette fonction retire le dernier élément de la Pile (et la modifie donc) et le renvoie.</i></p> <p><i>Renvoie l'élément au « sommet » de la Pile, c'est à dire le prochain élément à sortir de la pile.</i></p>
--	--

Remarque.

- On aurait aussi pu donner une signature fonctionnelle.
- Cette propriété **LIFO** rend les piles indispensables dans la résolution de nombreux problèmes (à noter que la récursion peut fréquemment remplacer une pile).

2.2.0 Implémentations par tableaux

L'idée est de faire un tableau dynamique, où l'on empile et depile à la fin : `empiler` correspond alors à `AJOUT`, et `depiler` à `RETRAIT`.

FIGURE 5.15 – Succession d'opération sur des piles implémentées par tableaux dynamiques

Proposition 19 (Complexité des piles par tableaux dynamiques).

Cette implémentation permet d'obtenir les complexités suivantes pour une pile éléments :

- `pile_vide` : $O(1)$
- `empile` : $O(1)$ amorti
- `depile` : $O(1)$ amorti
- `sommet` : $O(1)$
- `est_vide` : $O(1)$

Démonstration. Cf la section sur les tableaux dynamiques. Pour `sommet`, il suffit de renvoyer l'élément d'indice `len-1`, et pour `est_vide` il suffit de tester si `len = 0`. □

2.2.1 Implémentation par listes simplement chaînées

L'idée est de faire une liste simplement chaînée, où l'on empile et dépile en tête : `empiler` correspond alors à une insertion en indice 0, et `depiler` à une suppression en indice 0 (plus le fait de renvoyer l'élément).

FIGURE 5.16 – Succession d'opération sur des piles implémentées par listes simplement chaînées

Remarque. C'est cette implémentation qui est utilisée par le module `Stack` de OCaml, où le type `'a stack` est défini comme étant une `'a list ref` (c'est à dire un pointeur vers une `'a list`).

Proposition 20 (Complexité des piles par listes simplement chaînées).

Cette implémentation permet d'obtenir les complexités suivantes pour une pile :

- `pile_vide` : $O(1)$
- `empile` : $O(1)$
- `depile` : $O(1)$
- `sommet` : $O(1)$
- `est_vide` : $O(1)$

Démonstration. Cf la sous-sous-section sur les listes simplement chaînées. Pour `empiler`, `depiler` et `sommet`, ces opérations travaillent sur la cellule de tête et donc se font bien en temps $O(1)$. □

2.3 Files

Comme les piles, les Files sont des listes linéaires particulières où l'on restreint insère, supprime et accés pour obtenir une garantie d'ordre :

Définition 21 (File).

Une **file** (anglais : **queue**) est une liste linéaire qui garantie que le premier élément inséré sera le premier élément supprimé. En particulier, on ne peut pas insérer ou supprimer n'importe où. On dit qu'une file vérifie le principe **FIFO** : **F**irst **I**n **F**irst **O**ut (premier entré, premier sorti).

Exemple. Une file d'attente dans une boulangerie^{10 11}.

Exemple.

FIGURE 5.17 – Enfilages et défilages successifs dans une file

Définition 22 (Signature des files).

Voici la signature (impérative) des files, un peu documentée :

- Type : File
- Utilise : Élément, Booléen
- Constantes : file_vide
- Opérations :

– enfile : File × Élément → rien	<i>Cette fonction fait entrer un élément dans la file (et la modifie donc).</i>
– defile : File → Élément	<i>Cette fonction fait sortir le premier élément de la File (et la modifie donc) et le renvoie.</i>
– prochain : File → Élément	<i>Renvoie le prochain élément à sortir de la file.</i>
– est_vide : File → Bool	

Remarque.

- Là encore, on aurait aussi pu donner une signature fonctionnelle.
- Cette propriété **FIFO** rend les piles indispensables dans la résolution de nombreux problèmes.

2.3.0 Implémentations par tableaux circulaires

On borne le nombre d'éléments de la file par une certaine constante `len_max`.

Pour utiliser des tableaux, la première idée est de faire presque comme avec des piles par tableaux dynamiques : pour enfiler on ajoute à droite, et pour défiler on retire à gauche. Cette méthode fonctionne, mais atteint vite un problème :

¹⁰. Boulangerie de Lamport =D

¹¹. Vous comprendrez la blague de la note de bas-de-page précédente en MPI. Je vous assure qu'elle est drôle.

FIGURE 5.18 – Problème de l'implémentation naïve des files dans un tableau

Pour pouvoir utiliser l'espace disponible, l'astuce est de rendre le tableau **circulaire** :

Définition 23 (Implémentation des files par tableaux circulaires).

Pour représenter une file dans un tableau circulaire à len_max cases, on mémorise :

- le tableau `arr` et sa longueur `len_max`
- `entree`, un indice du tableau qui stocke l'entrée de la file (c'est là qu'auront lieu les enfilages).
- `sortie`, un indice du tableau qui stocke la sortie de la file (c'est là qu'auront lieu les défilages).

Les éléments de la file sont alors stockés à partir de l'indice `debut` et jusqu'à l'indice `fin`, en prenant en compte la circularité (çad en calculant le prochain indice modulo len_max) :

FIGURE 5.19 – Organisation d'un tel tableau circulaire

FIGURE 5.20 – Un autre exemple où l'on voit l'aspect circulaire

Remarque. Les indices `entrée` et `sortie` sont parfois des indices de la file *inclus*, c'est à dire qu'on y trouve bien des éléments de la file, et parfois *exclus* (c'est à dire que le premier/dernier élément se trouve juste avant/après). Il faut bien lire la description de l'énoncé pour savoir !

Exemple.

FIGURE 5.21 – Succession d'enfilage et de défilage dans une file par tableaux circulaires

Remarque. Distinguer la file vide d'autres files n'est pas évident, et la façon de le faire dépend de l'implémentation. Il faut bien lire la description de l'implémentation pour savoir !

(a) File vide ou à 1 élément ?

(b) File vide ou pleine ?

FIGURE 5.22 – File vide ou non ?

Proposition 24 (Complexité des files par tableaux circulaires).

Cette implémentation permet d'obtenir les complexités suivantes pour une file :

- `file_vide` : $O(1)$ ou $O(\text{len_max})$ s'il faut initialiser le contenu des cases
- `enfile` : $O(1)$
- `defile` : $O(1)$
- `prochain` : $O(1)$
- `est_vide` : $O(1)$

Démonstration. Respectivement :

- Il suffit de créer le tableau.
- Il suffit d'écrire dans la bonne case, qui est connue grâce à `entrée`, puis de mettre à jour `entree`.
- Idem mais pour la sortie.
- Il suffit de lire la bonne case grâce à l'indice `sortie`.
- La longueur d'une file soit se déduit de l'écart entre `entrée` et `sortie`, soit est stockée (en plus de stocker les deux indices) : dans tous les cas, c'est en temps constant.

□

Proposition 25 (Longueur d'une file par tableau circulaire).

Le nombre d'éléments d'une file implémentée par tableaux circulaires peut se calculer à l'aide de l'écart entre les indices.

La formule est généralement de la forme $(\text{sortie} - \text{entree}) \bmod \text{len_max}$, avec parfois un ± 1 à rajouter au tout selon l'implémentation précise utilisée.

Exemple. Dans les exemples ci-dessous, *sortie* est l'indice (inclus) du prochain élément à sortir, et *entrée* l'indice (inclus) de l'entrée de la file (c'est à dire du dernier élément inséré).

(a) Une première file à 4 éléments

(b) Une seconde file à 4 éléments (où la circularité se voit)

FIGURE 5.23 – Exemples de calculs de longueur de file dans des tableaux circulaires

2.3.1 (Hors-programme) Amélioration avec des tableaux dynamiques

Pour échapper au fait que les files dans des tableaux circulaires sont de longueur au plus len_max , on peut les implémenter dans des tableaux dynamiques. Il faut alors faire attention lors du doublement du tableau à correctement mettre à jour *debut* et *fin*, notamment lorsque $\text{fin} < \text{debut}$.

2.3.2 Implémentations par listes simplement chaînées

Une autre implémentation possible utilise des listes simplement chaînées dont on mémorise un pointeur vers le début et la fin. Ici, on utilisera une signature impérative : on modifie la liste simplement chaînée passée en argument¹²

Exemple.

FIGURE 5.24 – Enfilages et défilages successifs dans une file par listes simplement chaînées

12. Cela sera donc différent des exemples de listes OCaml précédents, donc.

Définition 26 (Implémentation des files par listes simplement chaînée).

Pour implémenter des files à l'aide de listes simplement chaînées, on utilise :

- une liste simplement chaînée. Le prochain élément à sortir est en tête, le dernier élément inséré dans la cellule de fin de liste.
- un pointeur sortie vers la tête de la liste. Il permet de défiler.
- un pointeur entree vers la dernière cellule de fin. Il permet d'enfiler.

Pour défiler, il suffit alors de lire l'élément du maillon de tête, et de modifier le pointeur sortie pour qu'il pointe sur la queue.

Pour enfiler, on crée un nouveau maillon de fin contenant l'élément à enfiler, puis on modifie le chaînage de l'ancien maillon de fin (accessible via entree) pour le faire pointer sur le nouveau. On met ensuite à jour entree en conséquence.

Proposition 27 (Complexités des files par listes simplement chaînées).

ne telle implémentation permet d'obtenir les complexités suivantes :

- `file_vide` : $O(1)$
- `enfile` : $O(1)$
- `defile` : $O(1)$
- `prochain` : $O(1)$
- `est_vide` : $O(1)$

Démonstration. Toutes ces complexités devraient être évidentes à l'aide des descriptions des listes simplement chaînées et du fonctionnement de `enfile` et `defile`.

Précisons que pour `est_vide`, il suffit de tester si `sortie` pointe sur un maillon ou sur rien. \square

2.3.3 Implémentation par listes doublement chaînées

On peut faire l'implémentation précédente avec des listes doublement chaînées (et toujours un pointeur vers chacune de deux extrémités). Cela permet de parcourir la file dans les deux sens, et non simplement de la sortie vers l'entrée.

Remarque. C'est une version très connue, au point où plusieurs livres sur le sujet ne mentionnent pas la version simplement chaînée. Elle est pourtant très pratique ; parcourir la file dans le sens entrée->sortie ne sert pas si souvent que ça¹³.

2.3.4 Par double pile

On peut implémenter une file à l'aide de deux piles. L'idée est d'avoir une pile « d'entrée » où l'on empile les éléments enfilés, une pile de sortie où l'on dépile les éléments défilés, et lorsque cette dernière est vide on renverse la pile d'entrée sur la pile de sortie.

FIGURE 5.25 – File par double pile

13. Ne me faites pas dire ce que je n'ai pas dit : ça sert quand même. Mais à mon humble avis, pas assez souvent pour que cela justifie de ne pas présenter la version simplement chaînée.

Les opérations de la signature des files sont alors en temps constant (amorti). Plus de détails en TD.

Remarque. On peut aussi implémenter une pile à l'aide de deux files, mais c'est plus compliqué.

2.4 Variantes des files

Plusieurs variantes des files existent :

- Files à double entrée : on peut enfiler ou défiler sur les deux extrémités de la file. En anglais, on parle de **double ended queue**, souvent abrégé **dequeue**.
Pour les implémenter, on peut utiliser des tableaux circulaires ou des listes doublement chaînées.
- File de priorité : les éléments ne sortent pas de la file selon leur ordre d'entrée, mais selon une priorité qui leur est assignée. C'est typiquement ce que fait l'ordonnanceur de tâches de votre ordinateur (certains processus sont plus prioritaires à « faire avancer » que d'autres), un site internet (on priorise certaines requêtes sur d'autres¹⁴), ou un hôpital (on priorise les patient-es dont l'état est le plus critique). Nous verrons au S2 comment les implémenter.

3 Aperçu des structures de données S2

Au semestre 2, nous verrons d'autres de structure de données qui permettent de résoudre d'autres problèmes :

- Une nouvelle structure séquentielle, les dictionnaires.
- Les structures hiérarchiques (où les éléments ne sont pas à la suite mais au-dessus/dessous d'autres éléments) : les arbres (sous plusieurs formes), dont notamment les tas qui permettent de faire des files de priorité.
- Les structures relationnelles (où on stocke des relations entre éléments) : les graphes, qui permettent de faire à peu près l'entièreté de l'informatique^{15 16}

Nous verrons également et surtout plus de méthodes algorithmiques, c'est à dire de chapitres de « conception de solution pour résoudre un problème » (comme on a cherché des solutions pour Hanoï et autres problèmes dans le chapitre sur la récursion). Ces chapitres s'appuieront souvent sur des structures de données pour résoudre efficacement les problèmes.

14. De manière générale, l'organisation des réseaux informatiques ou téléphoniques sont basées sur des files d'attente et des files de priorité extrêmement optimisées; ainsi que sur des modélisations et études probabilistes poussées.

15. J'exagère un peu.

16. Mais pas tant que ça.