1 Structure de données : interfaces et implémentations

Les données sont un des 4 piliers de l'informatique (avec les machines, les algorithmes et les langages). Nous avons découvert en classe de 1ère les **types de base**, comme les entiers, les booléens ou les chaînes de caractères, et les **types construits** comme les tableaux ou les p-uplets.
D'autres structures de données existent et nous verrons cette année la notion de **type abstrait de donnée** (TAD).

> Un **type abstrait** définit l'**interface** du type de donnée : elle caractérise de façon théorique la nature des données et les opérations que l'on peut effectuer sur ce type de donnée.

On peut présenter une interface de type abstrait de donnée sous la forme d'un tableau indiquant le nom de la structure de donnée, les types de données qu'elle utilise (comme pour un type construit) et les méthodes qu'on peut appliquer sur le type de données.

Exemple : On peut écrire l'interface d'un tableau, qui contient des données de type data :

TAD : Tableau
Utilise : - entier - data
Opérations : `creation` : $\emptyset \rightarrow$ tableau `lecture` : tableau, entier $\rightarrow$ data `écriture` : tableau, entier, data $\rightarrow \emptyset$ *Remarque* : ne renvoie rien mais modifie le tableau.

Pour chaque type abstrait de données, il peut y a voir plusieurs manières d'envisager sa mise en œuvre concrète. Cela peut dépendre du langage utilisé ou du paradigme de programmation ou de la machine sur laquelle on implémente la structure de donnée.

> L'implémentation est la façon de mettre en œuvre concrètement un type abstrait de données dans un programme.

2 Structures linéaires : position du problème

Dans une administration, on doit gérer de nombreux dossiers de clients. Comment ranger ces dossiers pour faciliter leur gestion ? Plusieurs solutions peuvent s'offrir à nous.

1. **La gestion avec des casiers** :
   On peut imaginer qu'on dispose d'une étagère avec des casiers et on range chaque dossier dans un casier, en remplissant les casiers dans l'ordre (de la gauche vers la droite). Chaque casier est identifié par un numéro (« une adresse »), et on peut ainsi facilement accéder au dossier d'un client en ouvrant le bon casier.

   L'ajout d'un nouveau dossier est assez simple : on le place dans le dernier casier vide. En revanche, lorsque l'étagère est pleine, il faut en racheter une autre ! Et comme on ne sait pas toujours combien de dossiers on aura à traiter, on risque de devoir acheter une étagère surdimensionnée (et donc très peu remplie) ou sous-dimensionnée et il faudra en racheter d'autres plus grandes et re-ranger tous les dossiers dans cette nouvelle étagère...

Par ailleurs, si les dossiers doivent par exemple être classés par ordre alphabétique, l'insertion d'un nouveau dossier nécessite aussi beaucoup de manipulations pour faire une place dans un casier au milieu (il faudra déplacer tous les dossiers de la fin de l'alphabet d'un casier chacun !).

En informatique, une telle étagère à casiers est un **tableau**. Chaque casier est un **espace mémoire dimensionné pour recevoir un certain type de donnée**.

2. **La gestion avec des dossiers chaînés** :
Une autre solution consiste à placer chaque dossier dans une pochette munie d'un mousqueton permettant de l'attacher à une autre pochette similaire.
On peut accéder à cette chaîne de pochettes à partir de la première pochette qu'on ne perd jamais de vue.
L'accès à un dossier quelconque est un peu compliqué car il faut accéder à chaque pochette à partir de la première en suivant le mousqueton vers la suivante jusqu'à atteindre la pochette contenant le bon dossier.
Cependant, l'insertion d'un nouveau dossier est simple : on le place dans une pochette vide et on accroche le mousqueton de cette pochette à la première pochette accessible (qui devient donc désormais la 2ème pochette).
L'insertion en milieu de chaîne (par exemple si les dossiers sont triés par ordre alphabétique) nécessite de parcourir la chaîne de pochettes jusqu'au bon endroit. On peut alors accrocher la pochette contenant le nouveau dossier avec son mousqueton à la pochette suivante, et accrocher le mousqueton de la pochette d'avant à cette nouvelle pochette (on n'a pas besoin de déplacer tous les dossiers comme pour l'étagère à casiers).

   En informatique, une telle structure d'éléments chaînés est une **liste chaînée**. Chaque pochette est une structure de type **maillon contenant une donnée et un lien vers le maillon suivant**. On peut concevoir différentes manières d'*implémenter* ces maillons et les liens entre eux, et cela peut aussi dépendre du langage utilisé. La liste est un **type de données abstrait** (TAD).

3. **La gestion avec une pile de dossiers** :
Voyons maintenant une autre solution consistant à poser les dossiers les uns sur les autres sur le bureau pour former une jolie pile de dossiers.
L'accès au dossier au sommet de la pile est très simple (il suffit de le prendre !) mais les autres dossiers sont inaccessibles, à moins de dépiler temporairement tous les dossiers qui était au-dessus en créant une autre pile intermédiaire à côté, puis de tout rempiler après.
L'ajout d'un dossier est encore très simple : on le pose au sommet de la pile.
L'insertion d'un dossier en milieu de pile ne sera même pas envisagée dans ce type de gestion.
Cette solution permet donc de gérer facilement les derniers dossiers arrivés. Dernier arrivé, premier traité ! (on plaint le dossier au fond de la pile... sera-t-il jamais traité ?)

   En informatique, une telle structure de données est une **pile** ! Là encore, on peut concevoir différentes manières d'implémenter ces piles concrètement. La **pile est aussi un type abstrait de données**. (on pourrait par exemple utiliser une liste chaînée pour créer une pile).

4. **La gestion avec une file de dossiers** :
Envisageons pour finir une solution consistant à ranger les dossiers dans une sorte de distributeur que l'on peut remplir par le dessus et qui donne un accès au dossier au bas du distributeur.
L'accès au dossier du fond du distributeur est donc immédiat : il est toujours accessible directement ! Mais là encore les autres dossiers sont inaccessibles.
L'ajout d'un dossier reste très simple : on le charge au sommet du distributeur.

L'insertion d'un dossier en milieu de distributeur est encore non envisagée dans ce type de gestion.
Cette solution permet donc de gérer facilement les premiers dossiers arrivés. Premier arrivé, premier traité ! (dans notre situation cette solution semble meilleure, mais si nous considérons la pioche d'un jeu de cartes, cela ne semble plus pertinent).

En informatique, une telle structure de données est une **file** ! De nouveau, on peut concevoir différentes manières d'implémenter ces files concrètement. **La file est aussi un type abstrait de données**. (on pourrait par exemple utiliser deux piles pour créer une file).

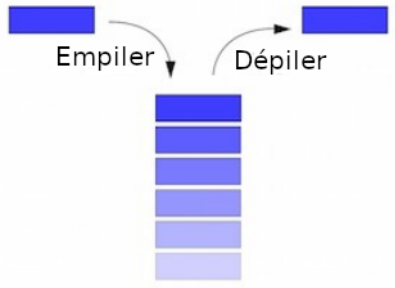
3 Structures de données linéaires : interface

On vient de décrire des types de données abstraits qui correspondent aux structures de données de type **liste linéaire** : ce sont des **structures séquentielles**. Une liste est constituée d'une **suite de places** qui contiennent des **éléments**. Une liste a une place de début et une place de fin. Connaissant une place, on doit pouvoir accéder à la place suivante.

Les piles et les files sont des exemples particuliers de listes.

3.1 Interface de pile (LIFO)

Les éléments sont ajoutés (*push*) à la pile par son sommet et sont aussi retirés (*pop*) par son sommet (exemple : pile d'assiettes).
La sémantique d'un pile est de type « dernier arrivé, premier sorti » : **LIFO** : Last In First Out.



`Pile`	coût
`creer_pile` : $\emptyset \rightarrow$ pile	O(1)
`est_vide` : pile $\rightarrow$ booléen	O(1)
`empiler` : pile, element $\rightarrow$ pile	O(1)
`depiler` : pile $\rightarrow$ pile	O(1)
`sommet` : pile $\rightarrow$ element	O(1)
`taille` : pile $\rightarrow$ entier	O(n)

Nous avons précisé également une notion de coût en temps d'exécution pour chacune des opérations. Voici les notations utilisées :

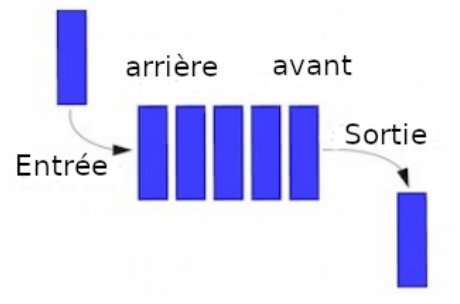
- **O(1)** : signifie un **coût constant**, càd que qu'elle que soit la taille de la pile, le coût en temps d'exécution reste le même.
- **O(n)** : signifie que le **coût est linéaire** : le temps est proportionnel à la taille des données.
- **O(n^2)** : signifie que le **coût est quadratique** : le temps est proportionnel au carré de la taille des données (*donné à titre indicatif car aucune des opérations envisagées n'aura ce coût*).

Exemples d'utilisation :

- Bouton « Annuler » dans un logiciel (Ctrl + Z). Chaque action du logiciel est empilée dans une pile « mémoire » et peut être dépilée (jusqu'à une certaine profondeur).
- Bouton « Retour » d'un navigateur Web. Les page consultées forment la pile « historique ».
- Cette fonctionnalité a déjà été rencontrée dans la pile d'exécution (cf cours récursivité).
- Utiliser dans certaines calculatrices pour effectuer les calculs (les nombres et les opérations sont manipulés dans une pile).

3.2 Interface de file (FIFO)

Les éléments sont ajoutés (*push*) à la file par une extrémité mais sont retirés (*pop*) par l'autre extrémité (exemple : file d'attente à la caisse).
La sémantique d'une file est de type « premier arrivé, premier sorti » : **FIFO** : First In First Out.



`File`	coût
`creer_file` : $\emptyset \rightarrow$ file	O(1)
`est_vide` : file $\rightarrow$ booléen	O(1)
`enfiler` : file, element $\rightarrow$ file	O(1)
`defiler` : file $\rightarrow$ file	O(1)
`premier` : file $\rightarrow$ element	O(1)
`taille` : file $\rightarrow$ entier	O(n)

Exemple d'utilisation :

- File d'impression sur une imprimante. Les documents sont imprimés dans l'ordre où ils sont envoyés à l'imprimante.
- L'ordonnancement des processus par le système d'exploitation peut reposer sur une file.

Remarque :
On trouve parfois des interfaces légèrement différentes de celles présentées dans ce cours.
On pourrait par exemple considérer que l'accès à la taille de ces structures se fassent en temps constant. Cela nécessite alors de conserver en mémoire une trace de la taille au fur et à mesure des modifications effectuée sur la structure. Bien souvent, un coût en temps diminué signifie un coût en mémoire augmenté.

3.3 Interface de liste

3.3.1 Vision itérative

Pour les listes, les opérations applicables sont assez générales (les piles et les files sont par exemple des cas particuliers de listes) :

Liste itérative
`creer_liste` : $\emptyset \rightarrow$ liste
`accès` : liste, entier $\rightarrow$ place
`contenu` : place $\rightarrow$ element
`taille` : liste $\rightarrow$ entier
`supprimer` : liste, entier $\rightarrow$ liste
`inserer` : liste, entier, element $\rightarrow$ liste
`successeur` : place $\rightarrow$ place

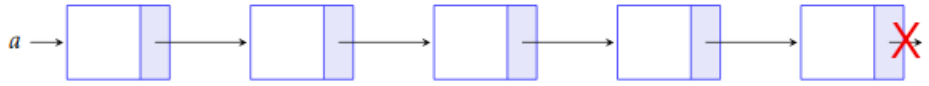
Cette vision des listes correspond bien à la notion de tableau en informatique. Un tableau est en effet une structure de données qui contient une collection d'éléments (de même type) stockés en mémoire dans des cases voisines, et directement accessible par leur adresse mémoire (par rapport à la première case mémoire du tableau).
L'accès à un élément ou l'ajout d'élément en fin de liste est aisé (coût constant en temps), mais l'insertion nécessite un coût linéaire car cela nécessite de recopier tous les éléments à décaler d'une place vers la droite du tableau.
Un autre inconvénient est qu'un tableau a une taille fixe.

3.3.2 Vision récursive

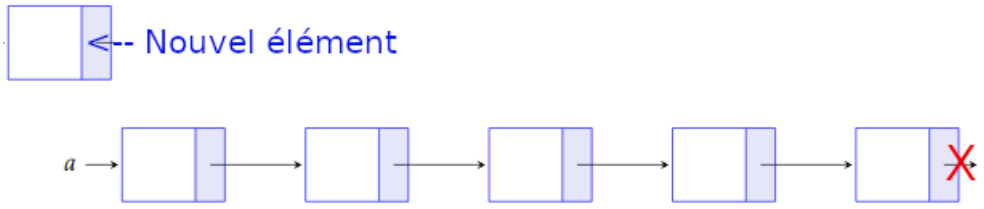
On peut également avoir une vision récursive des listes. Une liste étant alors essentiellement une place de tête et une liste (qui est le reste de la liste privée de sa tête !).

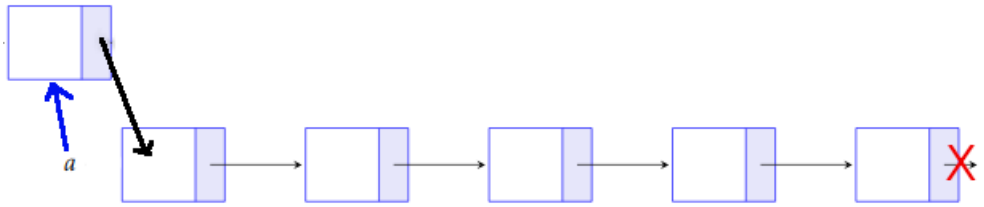
Liste récursive
`creer_liste` : $\emptyset \rightarrow$ liste
`tête` : liste $\rightarrow$ place
`fin` : liste $\rightarrow$ liste
`cons` : liste, element $\rightarrow$ liste
`premier` : liste $\rightarrow$ element
`contenu` : place $\rightarrow$ element
`successeur` : place $\rightarrow$ place

Cette vision des listes quant à elle se prête bien à l'implémentation sous forme de **liste chaînée**. Une liste est ainsi constituée d'une **suite de maillons** qui contiennent chacun un élément et une **référence vers un autre maillon**.



L'insertion d'un élément en début de liste par exemple est alors très simple : on crée un nouveau maillon (qui deviendra la tête) et on fait fait pointer sa référence vers l'ancien maillon de tête.





Remarque : On peut aussi définir des listes doublement chaînées qui permettent un accès à la liste depuis son dernier maillon. Chaque maillon possède alors une référence vers son successeur et son prédécesseur. Les coûts en temps peuvent diminuer au prix du coût en mémoire.

3.3.3 Opérations complémentaires

On peut concevoir aussi classiquement d'autres opérations sur les listes :

Liste itérative
`rechercher` : liste, element → booléen
`concatener` : liste, liste → liste
`scinder` : liste → liste, liste

4 Implémentation

Rappel : l'interface d'un type abstrait de données n'indique absolument rien sur l'**implémentation** de la structure de données.
L'implémentation est la mise en œuvre concrète de la structure. De nombreuses solutions sont souvent possibles, dépendant par exemple :

- du paradigme de programmation. La programmation orientée objet pourra par exemple être utilisé pour implémenter ces structures. On peut s'interroger sur le paradigme de programmation fonctionnelle pour savoir si ces structures seront mutables ou non...
- des algorithmes qui manipuleront ces structures. Certaines implémentations permettront par exemple d'être plus efficace (en temps ou en mémoire).

Implémentation avec des listes Python :
Peut-être pensez-vous aux listes Python comme représentation de ces structures ? C'est une possibilité, mais pas une obligation car l'implémentation n'est pas imposée par la structure de données.
En l'occurrence, le type `list` de Python que vous avez peut-être appelé « tableau » l'an dernier est un type hybride qui ne correspond à aucune de ces structures directement.
Bien entendu, il est possible de facilement implémenter ces structures de données en se reposant sur les listes Python. Cependant, comme déjà évoqué précédemment, le type `list` de Python ne respecte aucune de ces interfaces rigoureusement. On pourra donc toujours facilement et rapidement simuler ces structures avec des listes Python, mais toujours au prix d'un certain manque de rigueur, en particulier en ne respectant pas forcément les coûts en temps pour les opérations à effectuer sur ces structures.
Les listes Python s'apparentent aux *tableaux dynamiques*.

Rappel de quelques méthodes utiles sur les listes Python :

```
liste.append(truc)    # ajout en fin de liste
liste.insert(2, truc) # insertion à l'indice 2
liste.pop()           # suppression avec renvoi du dernier élément
liste.pop(0)          # suppression avec renvoi de l'élément d'indice 0
```