Chapter 1

Type de données abstrait

1.1 Quelques définitions

Type de données abstrait ou structure de données abstraite: . C'est la description d'un ensemble organisé d'objets et des opérations de manipulation sur cet ensemble. Pour spécifier un type de données abstrait, on annonce quelles sont les données et on précise comment les manipuler, quelles sont les actions autorisées: c'est en quelque sorte un contrat à respecter qui fournit une **interface** à l'utilisateur. **Celui-ci utilisera les opérations permises de l'interface *sans se soucier de l'implémentation.*** Contrairement à l'implémentation, il ne dépend pas du langage utilisé.

Structure de données: Implémentation explicite d'un ensemble organisé d'objets, avec la réalisation des opérations d'accès, de construction et de modification.

Primitives: Pour chaque type de données, c'est un "petit" nombre d'actions, dites **primitives**, censées permettre d'obtenir par "combinaison" toutes les opérations utiles. On précise la complexité de ces primitives. Ce petit nombre n'est pas toujours minimal, par exemple les opérations arithmétiques pour le type "entier" où on inclut en général la multiplication parmi les primitives (alors qu'elle peut être obtenue à l'aide de l'addition !). On donne aussi les propriétés de ces opérations, c'est ce qu'on appelle la **sémantique**. L'ensemble de ces primitives est appelé **interface** ou API (Application Programming Interface). Spécifier un type abstrait, c'est définir son interface. Implémenter ce type abstrait, c'est donner le code en utilisant les types déjà existants.

L'intérêt de cette notion de type abstrait de données est de raisonner en s'affranchissant de tout langage de programmation. Les concepteurs des langages dits de haut niveau s'attachent à implémenter le stockage des valeurs et les primitives associées pour les types de données les plus classiques. En POO, une structure de données correspond à une classe: nous pourrons donc utiliser des classes pour implémenter un type abstrait.

Exemple Le type booléen est implémenté dans tous les langages, donnons sa spécification pour comprendre:

Notation : Vrai, Faux (par exemple).

Primitives : Opérateurs OU, ET, NON etc...

Sémantique : Le paragraphe sémantique précise les propriétés des opérations.

Pour tout a, b des booléens :

1. $\neg$ vrai = faux
2. $\neg \neg$ a = a
3. vrai $\wedge$ a = a

4. faux $\wedge$ a = faux
5. $a \vee b = \neg(\neg a \wedge \neg b)$

1.2 Les différents types de données abstraits

Une structure dont la taille est fixée et ne peut plus être modifiée est qualifiée de **statique**. Si elle peut varier suivant les besoins, elle est dite **dynamique**. Les structures de données classiques appartiennent le plus souvent aux familles suivantes :

- Les structures linéaires : il s'agit essentiellement des structures représentables par des suites finies ordonnées. Si chaque élément possède un indice par lequel on peut accéder à son emplacement, on parle de *tableau*. Si chaque place a un successeur ou un prédécesseur, on parle de liste chaînées mais dans ce cas l'accès n'est plus direct.

  On y trouve également les piles, les files.
- Les matrices ou tableaux multidimensionnels;
- Les structures arborescentes (en particulier les arbres binaires) ;
- Les structures relationnelles (bases de données ou graphes pour les relations binaire).

Nous étudions dans ce chapitre les structures linéaires.

1.3 Pile

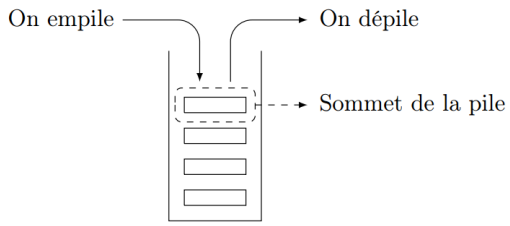
On retrouve pour les piles une partie des propriétés vues sur les listes. Dans les piles, il est uniquement possible de manipuler le dernier élément introduit dans la pile. On prend souvent l'analogie avec une pile d'assiettes : dans une pile d'assiettes la seule assiette directement accessible et la dernière assiette qui a été déposée sur la pile.

Les piles sont basées sur le principe LIFO (Last In First Out : le dernier rentré sera le premier à sortir). On retrouve souvent ce principe LIFO en informatique.

En plus de `creer_pile_vide`, voici les opérations que l'on peut réaliser sur une pile :

- `est_vide(pile)` renvoie `True` si `pile` est vide, `False` sinon.
- `empiler(pile, elem)` ou `push(pile, elem)`.
- `depiler(pile)` (ou `pop(pile)`) la valeur au sommet est supprimée et renvoyée.

Ces trois primitives sont en temps constant $O(1)$.



Une pile

On peut ajouter éventuellement `taille` pour connaitre le nombre d'éléments, et `sommet(pile)` pour récupérer la valeur en haut de `pile`.

Exemples : Soit une pile P composée des éléments suivants : 12, 14, 8, 7, 19 et 22 (le sommet de la pile est 22) Pour chaque exemple ci-dessous on repart de la pile d'origine :

- On effectue un `pop(P)`. Donner la composition de la pile `P`.
- On effectue `push(P,42)`. Que renvoie `sommet(P)` ?
- Si on applique `pop(P)` 5 fois de suite, que retourne `est_vide(P)` ?.


:Selon l'implémentation d'une pile P, on écrira `empiler(P, element)` ou `P.empiler(element)`.

Exercices

1. On applique les séquences suivantes:

```
Q = creer_pile_vide()

while not est_vide(P):

    empiler(Q, depiler(P))
```

Donner la composition de la pile `Q`.

```
>>> p = creer_pile_vide()
>>> for i in range(5):
>>>     empiler(i, p)
>>> depiler p
>>> sommet(p)
```

Qu'affiche le SHELL ?

2. Utiliser l'interface des piles pour calculer la somme des valeurs des éléments d'une pile `p` d'entiers. La pile p ne doit pas être modifiée après utilsaiton du script.
3. Le type `list` permet de respecter la complexité des primitives pour implémenter une pile: écrivez les fonctions correspondant aux primitives à l'aide d'une liste.
4. Implémenter une classe `Pile()`.
5. Une expression bien parenthésée est une expression qui contient, pour toute parenthèse ouvrante la parenthèse fermante associée. Par exemple: `(()())` est bien parenthésée mais `(()(()))` ne l'est pas. En utilisant une pile, écrivez une fonction `est_parenthese(chaine)` qui prend une chaine de caractère en paramètre et retourne `True` si celle-ci est bien parenthésée.

1.4 File

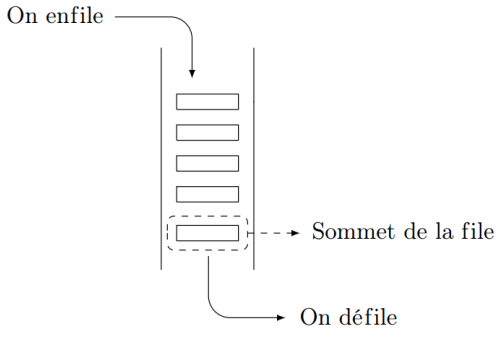
Une file se comporte comme une file d'attente: le premier arrivé est le premier sorti, c'est la sémantique FIFO: First In First Out parfois appelée LILO: Last in Last Out.

Quelles primitives peut-on envisager pour une file ?

Exercices

1. Implémentez une file à l'aide du type `list` de python.

   : Le type `list` ne permet pas en revanche d'implémenter une file en respectant la complexité d'ajout ou de suppression qui doit *normalement* s'effectuer en temps constant $O(1)$. La classe `deque` (double-ended queue) du module `collections` remédie à ce problème.
2. `https://alainbusser.frama.io/NSI-IREMI-974/queuesortable.html`
3. Utiliser deux piles pour implémenter une file remédie également au problème de complexité en temps constant. Implémentez une classe `file` en utilisant deux piles.



Une file

Note historique

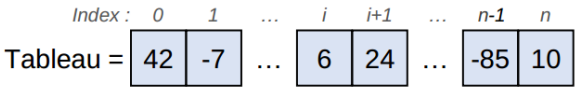
L'origine des piles et des files comme structures de données informatiques n'est pas certaine, ces notions existaient déjà en mathématiques et dans les entreprises avant l'introduction des ordinateurs. Donald Knuth attribue à Alan Turing le développement de piles pour gérer les liens entre sous-programmes en 1947.

1.4.1 Tableaux

Chaque élément d'un tableau est repéré par son index (nombre entier de 0 à longueur de liste -1). Le tableau est de taille fixe et connue d'avance.

Avantages: Taille en mémoire fixe. Accès facile à un élément du tableau avec son index .

Inconvénients: Nécessite de créer un nouveau tableau pour insérer ou supprimer un élément.



1.5 Listes

Une liste est une collection finie d'éléments qui se suivent, qu'il est possible de manipuler individuellement. C'est donc une structure de données **séquentielle** ou **linéaire**.

Une liste L est composée de 2 parties : sa tête (historiquement notée `car`), qui correspond au dernier élément ajouté à la liste, et sa queue (notée `cdr`) qui correspond au reste de la liste. Le langage de programmation **Lisp** (inventé par John McCarthy en 1958) a été un des premiers langages de programmation à introduire cette notion de liste (Lisp signifie "list processing"). Voici les opérations qui peuvent être effectuées sur une liste:

Primitives

- **créer** une liste vide (L=vide() on a créé une liste L vide)
- `est_vide(L)` renvoie vrai si la liste L est vide.
- `ajouter(elem, L)` un élément en tête de liste.
- `supprimer()` la tête `x` d'une liste L et éventuellement renvoyer cette tête x.
- `taille()`: compter le nombre d'éléments présents dans une liste.

Exemples : Voici une série d'instructions avec des commentaires (les instructions ci-dessous s'enchaînent) :

1. `L = vide()` on a créé une liste vide.
2. `est_vide(L)` : renvoie vrai.
3. `ajouter(3,L)` : La liste L contient maintenant l élément 3.
4. `est_vide(L)` : renvoie faux.
5. `ajouter(5,L)` : la tête de la liste L correspond à 5, la queue contient l'élément 3.
6. `ajouter(8,L)` : la tête de la liste L correspond à 8, la queue contient les éléments 3 et 5.
7. `t = supprimer(L)` : la variable t vaut 8, la tête de L correspond à 5 et la queue contient l élément 3.
8. L1 = vide()

9. La fonction `cons` permet de construire une liste à partir d'une tête et d'une queue. L2 = cons(8, cons(5, cons(3, L1))) : La tête de L2 correspond à 8 et la queue contient les éléments 3 et 5.

Note sur Python Contrairement à ce que pourrait laisser croire son nom, la classe `list` de Python n'est pas une liste au sens qu'on lui donne traditionnellement, mais une structure de donnée plus complexe qui cherche à concilier les avantages des tableaux et des listes chaînées, à savoir :

Etre une structure de donnée **dynamique** dans laquelle les éléments sont **accessibles à coût constant**.

Ainsi, ce que la documentation de Python appelle *lists* s'utilise comme des tableaux, au sens où Python permet l'accès aléatoire aux valeurs stockées, grâce à l'indexation. Cela est dû à la méthode `__getitem__` des listes Python. Et comme une liste chaînée au sens où le stockage est bien géré de façon **dynamique: on peut faire varier la taille en cours d'exécution d'un programme.**

1.5.1 Implémentation

Liste chainées

Chaque maillon ou cellule est un *objet* dans lequel on stocke l'élément ainsi qu'un pointeur vers le maillon suivant. Le dernier pointeur pointe sur la valeur *null*. Contrairement au tableau pour lequel l'ordre linéaire est déterminé par les indices, l'ordre d'une liste chaînée est déterminée par un pointeur dans chaque objet.

Une implémentation objet est détaillée dans le paragraphe suivant.

Avantages: Ajout ou suppression facile d'un maillon .

Inconvénients: Seul le premier maillon est accessible directement, il faut donc parcourir les maillons un à un pour accéder à un élément

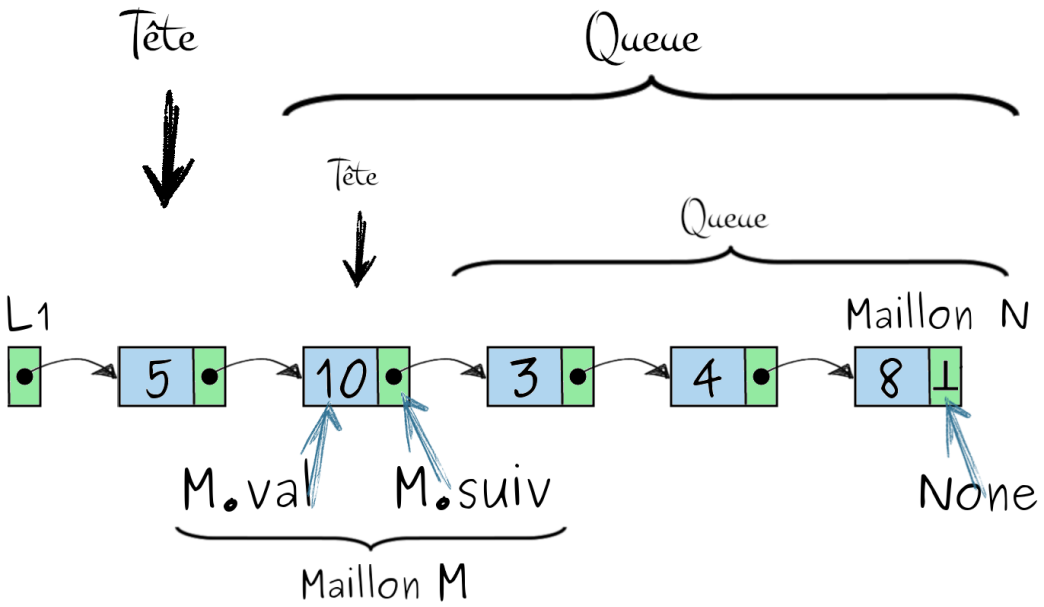


Figure 1.1: Les listes chainées sont des structures de données récursives.

Complexité des primitives

Complexité de chaque Opération	Tableau	Liste chainée
Accès à un élément	O(1)	O(n)
ajouter un élément en tête de liste	O(n)	O(1)
supprimer la tête d'une liste et renvoyer cette tête	O(n)	O(1)
compter le nombre d'éléments présents dans une liste	O(1)	O(n)
ajouter un élément en fin de liste	O(n)*	O(n)
supprimer la fin d'une liste et renvoyer cette fin	O(n)*	O(n)
Ajouter ou supprimer un élément à une position quelconque (L = position)	O(n)	O(L)

* O(1) en python

1.5.2 Exemple de la liste chainée : implémentation objet

Lorsque l'implémentation de la liste fait apparaître une chaîne de valeurs, chacune pointant vers la suivante, on dit que la liste est une liste **simplement** chaînée. Une liste peut être **doublement chaînée** si on pointe également vers l'élément précédent ou encore **cylique** si le dernier élément pointe vers le premier.

Exemple: On crée une variable à partir de **maillons** ou **cellules** : chacun contient une valeur et pointe vers le suivant.

```
>>> L1 = Maillon(5, Maillon(10, Maillon(3, Maillon(4, Maillon(8, None)))))
```

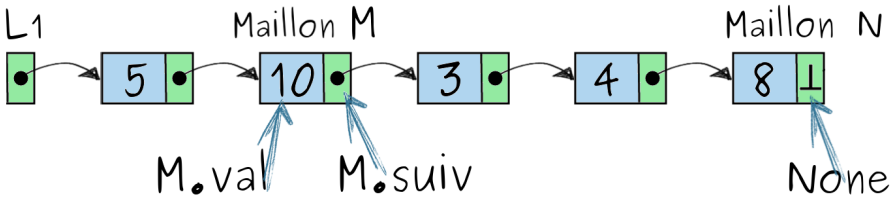


Figure 1 – Représentation d'une liste en mémoire.

- Une liste est caractérisée par un ensemble de cellules. Une liste chainée L est entièrement définie par son maillon de tête, c'est à dire l'adresse de son premier maillon.
- Le lien (on dira souvent le *pointeur*) vers la variable est un lien vers la première cellule, qui renverra elle-même sur la deuxième, etc.
- Chaque cellule contient donc une valeur et un lien vers la cellule suivante.

Une liste peut être vide (la liste vide est notée `X` ou bien `None` sur les schémas)

Temps d'accès non constant

Une conséquence de cette implémentation sous forme de liste chaînée est la non-constance du temps d'accès à un élément de liste : pour accéder au troisième élément, il faut obligatoirement passer par les deux précédents. **Dans une liste chaînée, le temps d'accès aux éléments n'est pas constant.**

Il est en effet impossible de connaître à l'avance l'adresse d'une case en particulier, à l'exception de la première. Pour accéder à la n-ième case, il faut donc parcourir les n-1 précédentes: **le coût de l'accès à une case est proportionnelle à la distance qui la sépare de la tête de la liste**.

Structure de donnée dynamique

En contrepartie, **ce type de structure est dynamique** : une fois la liste crée, il est toujours possible de modifier un pointeur pour insérer une case supplémentaire. En résumé :

- une liste chaînée est une structure de donnée dynamique ;
- le n-ième élément d'une liste chaînée est accessible en temps proportionnel à n.

 Contrairement à ce que pourrait laisser croire son nom, la classe `list` de Python n'est pas une liste chaînée au sens qu'on vient de lui donner, mais une structure de donnée plus complexe qui cherche à concilier les avantages des tableaux et des listes chaînées, à savoir : être une structure de donnée dynamique dans laquelle les éléments sont accessibles à coût constant.

Implémentation

On utilise deux classes `Maillon` et `ListeC` :

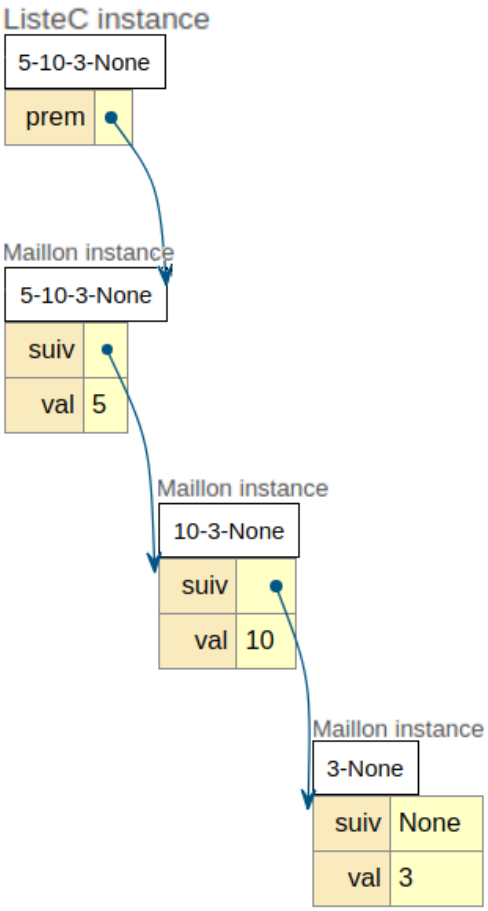
```
class Maillon:
     def __init__(self, valeur = None, suivant = None):
        self.val = valeur
        self.suiv = suivant

     def __str__(self):
         if  self.val is not None:
             return str(self.val) + "-" + str(self.suiv)
         else:
             return str(self.val)
```

Pour construire une liste, il suffirait d'appliquer le constructeur de la classe `Maillon` de façon **récursive** :

```
>>> liste = Maillon(1, Maillon(2, Maillon(3, None)))
```

On écrit toutefois une classe `ListeC` (à compléter): Ci-dessous la représentation en mémoire avec https://pythontutor.com/ des instructions suivantes:

```
>>>L = ListeC(5)
>>>L.ajoute(10)
>>>L.ajoute(3)
```




```
class ListeC:

    def __init__(self,premier=None,prem_suiv=None):
       self.prem = Maillon(premier, prem_suiv )

    def __str__(self):
       return str(self.prem)

     def est_vide(self):
        pass

    def tete(self):
        """
        renvoie le premier élément de la liste
        """

    def ajoute_debut(self, valeur):
        """
        ajoute un élement en debut de liste
        """

    def ajoute(self, valeur):
        """
        ajoute un élement en fin de liste
        """

    def insere(self, valeur, indice):
        """
        insère valeur à indice
        """
```

Exercice En utilisant la classe `Maillon`, implémenter une classe `PileC` avec les primitives habituelles.

1.6 Dictionnaires

Un dictionnaire est en réalité un tableau associatif. C'est un type abstrait qui associe des valeurs à des clés. Ils sont utilisés pour des applications où la recherche d'une valeur est prioritaire et doit se faire le plus rapidement possible.

Contrairement à la liste qui est accessible par un index, l'extraction de la valeur se fera grâce à la clé, peu importe l'ordre dans lequel sont rangées ou insérées les données.

1.6.1 Primitives du dictionnaire abstrait

- ajouter un couple clé : valeur,
- modifier la valeur d'une clé,
- supprimer une clé (et bien entendu sa valeur),
- rechercher la valeur correspondante à une clé.

1.6.2 Implémentation d'un dictionnaire

La principale implémentation d'un tableau associatif est l'utilisation d'une table de hachage.

L'implémentation que vous connaissez déjà est le type construit `dict` de python qui est une implémentation avec table de hachage parfaitement optimisée.

En complément: Vidéo d'Arnaud Legout (Hors programme NSI): `https://www.youtube.com/watch?v=IhJo8sXLfVw&list=PL2CXLryTKuwyOT6o3EEHL2ZUJS70o_NbT&index=4`.

Exercices Un bref retour sur les dictionnaires en python.

1. Ecrire une fonction `text2_ascii(mot)` utilisant un dictionnaire dont les clés sont les lettres de l'alphabet en majuscule et les valeurs le code décimal correspondant dans la table `ascii`.

   Exemple: La variable ascii pour trois lettres et une conversion:

   ```
   >>>ascii = {"A": 65, "B": 66, "C": 67 }
   >>>text2ascii('Rick')
   [82, 105, 99, 107]
   ```

2. Améliorer ce dictionnaire en remplaçant la valeur décimale par un tuple contenant la valeur décimale, hexadécimale, binaire. *Exemple pour trois lettres*:

   ```
   ascii = {'A': (65, '41', '1000001'), 'B': (66, '42', '1000010'),
               'C': (67, '43', '1000011')}
   ```

3. Voici les points pour quelques lettres au `Scrabble`:

Lettre	A	B	C	D	E	F	G	R	I	O	K	J	T
Points	1	3	3	2	1	4	2	1	1	1	10	8	1

   (a) Utiliser une variable `scrabble` de type `dict` pour représenter ce tableau.

   (b) Ecrire une fonction `points(mot)` qui renvoie sa valeur en points pour le Scrabble.

   ```
   >>>points('ATARI')
   5
   >>>points('RICK')
   15
   ```

1.7 Arbres et graphes

Les arbres et les graphes sont des structures de données importantes. Ce sont des structures de données hiérarchiques : les élements n'y sont pas rangés linéairement. Donald Knuth, professeur émérite à Stanford, auteur de l'ouvrage de référence sur l'algorithmique, en plusieurs volumes, intitulé *The Art of Computer Programming* (L'art de la programmation informatique), considère d'ailleurs les arbres comme la structure la plus fondamentale de toute l'informatique.

Ces deux structures de données feront l'objet de chapitres spécifiques.

1.8 Références

1. *http://pixees.fr*
2. `https://www.fil.univ-lille1.fr/~L2S3API/CoursTP/listes.html`
3. *Numerique et Sciences Informatiques-T.Balabonski,Kim Nguyen, S.Conchon, JC.Filliâtre, Edition Ellipses*

4. *Spécialité Numerique et Sciences Informatiques-S. Bays, Editions Ellipses*

5. *Algorithmique: Cormen, Leiserson, Rivest, Stein. Editions DUNOD*