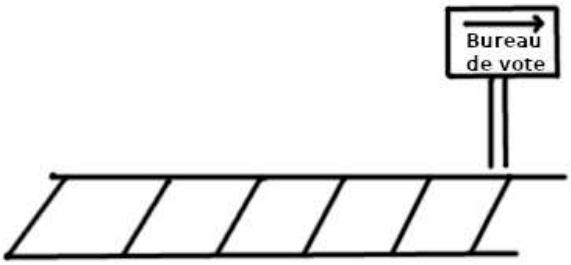
Structures de données abstraites 1/2

Les structures linéaires sont un modèle élémentaire de structure de données fréquemment utilisé dans les programmes informatique. Elles permettent d'organiser les données sous forme de suite d'éléments accessibles de manière séquentielle. Tout élément, sauf le dernier, d'une séquence possède un successeur. Les opérations d'ajout et de suppression d'éléments sont des opérations de base des structures linéaires. La façon dont sont définies ces opérations caractérise le type de structures linéaires : **pile, file ou liste**.

1.1 Les files structure abstraite ou interface

a. Exemple : le bureau de vote

Dans un bureau de vote, les électeurs se présentent les uns après les autres pour voter. Il se crée alors ce que l'on appelle une file d'attente. L'électeur présent depuis le plus longtemps sera le premier à voter. Cette file a une structure dite « **FIFO** » (First In, First Out) « premier entré premier sorti ».



b. Opérations primitives d'une file :

Cette file sera constituée de données d'un certain type (entiers, chaînes de caractères ou encore flottants ...).
Toute implémentation d'une file devra proposer d'une manière ou d'une autre les opérations primitives ci-dessous :

- Créer une file vide (`creer_file`)
- Tester si la file est vide (`est_vide_file`)
- Ajouter un élément dans la file (`ajouter_file`)
- Retirer et renvoyer le premier élément de la file (`retirer_file`)

Ce type de structure de données est très répandu en informatique, elle est utile dans de nombreuses situations comme par exemple pour gérer l'accès à une imprimante.

c. Exemple d'une implémentation d'une file.

Concrètement une file peut être implémentée à l'aide d'un tableau (une suite finie et contiguë de cases mémoires) muni d'une tête mobile. Lorsqu'on arrive en fin de tableau on n'arrive pas forcément en fin de file, on continue le parcours en se plaçant en début de tableau. On décide donc de représenter une file par un triplet (tableau, position de la tête, nombre d'éléments).
Il s'agit d'une des façons d'implémenter une file d'autres implémentations sont possibles.

Illustration

Pour illustrer cette implémentation, on peut reprendre l'exemple de la file d'attente à un bureau de vote.
Soit F une file d'attente à un bureau de vote. La file F est constituée :

d'un tableau de taille 10 (la taille de la file est un choix arbitraire à fixer une fois pour toute),

de la position de la tête de la file,

ainsi que du nombre d'éléments présents dans la file.

On considère une file dans un certain état à instant t :

sens de parcours de la tête $\longrightarrow$

Rang	0	1	2	3	4	5	6	7	8	9
tableau de la file F	elec3	elec4							elec1	elec2

Position de la tête : 8
Nombre d'élement(s) dans la file : 4

L'état de la file F représenté ci-dessus s'interprète ainsi :

- Il y a quatre électeurs dans la file comme l'indique le nombre d'éléments dans la file ;
- l'électeur sur le point de voter est en position 8 comme l'indique la position de la tête de lecture ;
- le suivant est en position 9, le suivant du suivant est en position 0, le dernier est en position 1 et les autres positions sont vides.

Considérons que l'électeur en tête de file vient de voter et sort donc de la file.
Comment évolue l'état de la file F en cas de sortie de l'électeur en tête de file ?
La sortie de l'électeur en tête de la file entraînera la libération de la place occupée par cet électeur, le déplacement de la position de la tête de 8 à 9, ainsi que la diminution d'une unité du nombre d'éléments présents dans la file. L'état de file peut-être représenté ainsi :

sens de parcours de la tête $\longrightarrow$

Rang	0	1	2	3	4	5	6	7	8	9
tableau de la file F	elec3	elec4								elec2

Position de la tête : 9
Nombre d'élement(s) dans la file : 3

Considérons maintenant qu'un électeur se présente dans la file.
Comment évolue l'état de la file F en cas d'arrivée d'un électeur ?
L'arrivée d'un électeur, s'il reste de la place dans la file, se fera en queue de file, soit à la position 2 et entraînera l'incrémentation d'une unité du nombre d'éléments dans la file.
L'état de file peut-être représenté ainsi :

sens de parcours de la tête $\longrightarrow$

Rang	0	1	2	3	4	5	6	7	8	9
tableau de la file F	elec3	elec4	elec5							elec2

Position de la tête : 9
Nombre d'élement(s) dans la file : 4

▶ Exercice 1

En partant de l'état de la file F décrire l'état de la file lorsque les évènements ci-dessous surviennent successivement.

1. L'arrivée d'un électeur supplémentaire.

2. La sortie de l'électeur venant de voter.

3. L'arrivée d'un nouvel électeur.

d. Implémentation en Python d'une file

On a vu que pour travailler avec des files il faut disposer des opérations primitives. Dans le modèle choisi elles sont au nombre de 5 :

- Créer une file vide (`creer_file`)
- Tester si la file est vide (`est_vide_file`)
- Ajouter un élément dans la file (`ajouter_file`)

- Retirer et renvoyer le premier élément de la file (`retirer_file`)

Il va falloir donc écrire en Python 5 fonctions. Dans ce cours, l'implémentation choisie pour une file est un triplet (tableau, position de la tête, nombre d'éléments) encore une fois il est important de noter que d'autres implémentations sont possibles.
L'implémentation concrète en Python de ce triplet peut se faire par exemple à l'aide d'un dictionnaire.
Ci-dessous la fonction `creer_file` qui permet de construire une file vide de taille fixe `LEN_FILE`.

```
LEN_FILE = 10

def creer_file():
    return  {"tableau" : ['-'] * LEN_FILE,
             "position_tete" : 0,
             "nombre_element" : 0}
```

Comme on peut le constater ce dictionnaire est constitué :

- d'un tableau de longueur `LEN_FILE` (ici `LEN_FILE = 10`), on convient que le caractère `'_'` « tiret du bas » correspond à vide,
- d'un entier compris entre 0 et `LEN_FILE` représentant la position de la tête de la file,
- d'un entier compris entre 0 et `LEN_FILE` représentant le nombre d'éléments présents dans la file.

En ce qui concerne la fonction `est_vide_file`, c'est une fonction qui prend en argument une file et retourne un booléen : vrai si la file est vide et faux sinon.

```
def est_vide_file(file):
    return file["nombre_element"] == 0
```

▶ Exercice 2

1. Implémenter en Python les primitives ci-dessous. On pourra pour ces trois fonctions définir des préconditions à l'aide d'« assert ».
   - `premier_file` qui retourne l'élément en tête de file,
   - `retirer_file` qui retire l'élément en tête de file et retourne la file mise à jour,
   - `ajouter_file` qui ajoute un élément en queue de file et retourne la file mise à jour.
2. Écrire une fonction `etat_file` supplémentaire permettant d'afficher l'état de file

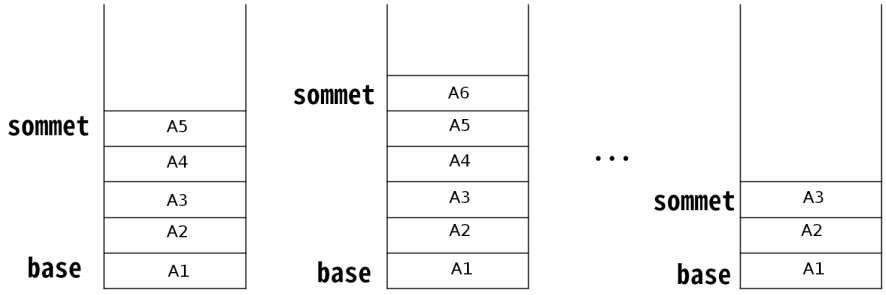
▶ Exercice 3

À l'aide des primitives implémentées en Python, tester la structure de file avec les exemples du bureau de vote de ce cours. Vérifier alors les états à l'issue des exercices 1 et 2.

1.2 Les piles structure abstraite ou interface

a. Exemple un pile d'assiettes

Une pile est une structure de données linéaire telle qu'on passe d'un état à l'autre de la pile par ajout ou suppression d'un élément en tête. Une pile a une structure « **LIFO** » c'est-à-dire « **Last In First Out** » dernier entré premier sorti. Pour schématiser une pile, on représente la suite chronologique de ses états :



Le dernier élément de la pile est appelé base de la pile, le premier élément de la pile est appelé sommet de la pile.
Remarque : toutes les mises à jour d'une pile se font depuis le sommet.
Reprenons l'exemple d'une pile d'assiettes du placard.

- On range les assiettes propres dans le placard depuis le lave-vaisselle au « sommet de la pile d'assiettes ». On dit que l'on **empile** un élément de la pile
- On sort une assiette du placard également depuis le « sommet de la pile d'assiette ». On dit que l'on **dépile** un élément de la pile.

b. Opérations primitives d'une pile

Les opérations primitives d'une pile sont :

- Créer une pile vide (`creer_pile`)
- Tester si une pile est vide (`est_vide_pile`)
- Ajouter un élément à la pile (`empiler`)
- Retirer et renvoyer un élément à la pile (`depiler`)

N.B : il n'y a pas de véritablement de consensus sur la liste des opérations primitives. Toutes ne sont pas indispensables, certaines pouvant être déduites de combinaisons d'autres.

c. Exemple d'une implémentation d'une Pile

Concrètement une pile peut être implémentée à l'aide d'un tableau (une suite finie et contiguë de cases mémoires) muni d'un sommet mobile. On décide donc de représenter une pile par un couple (tableau, position du sommet).

Soit P une pile, bien qu'en théorie une pile soit de taille infinie lors d'une implémentation concrète elle ne pourra contenir qu'un nombre fini d'éléments. On convient donc d'une taille maximale de la pile, ici on choisit arbitrairement 10, ce qui est raisonnable pour une pile d'assiettes. La pile P sera implémentée à l'aide d'un tableau de taille 10 et d'un entier représentant la position du sommet.

Ci-dessous on a représenté l'état de la pile P à un instant donné

Rang	0	1	2	3	4	5	6	7	8	9
tableau de la pile P	a1	a2	a3	a4	a5					

Position du sommet : 4

Ce qui s'interprète ainsi :

- Il y a cinq assiettes empilées dans l'ordre a1, a2, a3, a4 et a5 ;
- le sommet est en position 4 et l'assiette accessible par le sommet de la pile est l'assiette a5 ;
- un éventuel ajout d'une assiette (empilement) se ferait en position 5.

Comment évolue la pile lors de l'arrivée d'une assiette supplémentaire ?
Comme on l'a vu l'ajout ou empilement se fait par le sommet, si la pile n'est pas pleine la nouvelle assiette occupera la position sommet + 1 et la position du sommet est incrémentée d'une unité.
Compléter l'état de la pile ci-dessous.

Rang	0	1	2	3	4	5	6	7	8	9
tableau de la pile P										

Position du sommet : ...

d. Implémentation en Python

En Python, on devra écrire une fonction pour chacune de ces opérations primitives. Dans ce cours l'implémentation choisie pour une pile est un couple (tableau, position du sommet). L'implémentation concrète en Python de ce couple peut se faire par exemple à l'aide d'un dictionnaire.
Ci-dessous la fonction `creer_pile` permet de construire une file de taille fixe `LEN_PILE`.

```
LEN_PILE = 10

def creer_pile():
    return {"tableau" : ['_'] * LEN_PILE,
            "position_sommet" : -1}
```

Comme on peut le constater ce dictionnaire est constitué :

- d'un tableau de longueur `LEN_PILE` (ici `LEN_PILE = 10`), on convient que le caractère `'_'` « tiret du bas » correspond à vide,
- d'un entier compris entre -1 et `LEN_PILE` représentant la position du sommet de la pile,
- par convention la position du sommet à -1 représente le cas de la pile vide.

En ce qui concerne la fonction `est_vide_pile` il s'agit d'une fonction qui prend en argument une pile et retourne un booléen, vrai si la pile est vide et faux sinon.

```
def est_vide_pile(pile):
    return pile[position_sommet] == -1
```

▶ Exercice 4

Implémenter en python les fonctions :

- Ajouter un élément à la pile (empiler)
- Retirer un élément à la pile (depiler)

On pourra pour ces trois fonctions définir des préconditions à l'aide d'« assert ».

e. Aplications des piles

Une application simple des piles serait d'écrire un mot à l'envers. On empile chacune de ses lettres puis on dépile pour obtenir le mot à l'envers. Si le mot obtenu est identique alors il s'agit d'un palindrome.

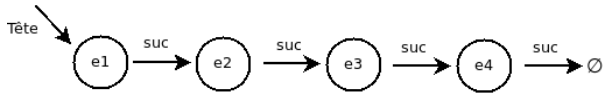
Une des principales applications est le mécanisme d'annulation d'action « undo » dans divers logiciels comme des éditeurs de textes, ou de manipulation d'images. Ce genre d'opération est réalisé en maintenant dans une pile les changements effectués.

Dans le même ordre d'idée la fonctionnalité « back », reculer d'une page des navigateurs web, utilise une pile, ou encore les sauvegardes automatiques ou « check point » dans un jeu vidéo de type plateforme.

La technique dite du Backtracking ou retour en arrière est très utilisée en intelligence artificielle. Cette technique permet d'explorer les possibilités offertes dans le but de déterminer une solution à un problème. Lors du parcours d'un arbre de différentes possibilités, on procède à un retour en arrière lorsqu'on atteint un "cul-de-sac" un chemin qui n'aboutit pas à une solution. On peut citer par exemple la recherche de la sortie dans un labyrinthe.

1.3 Liste

Une liste est une séquence ordonnée finie d'éléments de même type repérés selon leur rang dans la liste. On accède séquentiellement à un élément à partir du premier. Il est indispensable de disposer d'une fonction « suc » permettant d'accéder à l'élément suivant rendant accessible toute la liste en appliquant successivement cette fonction. On peut représenter sous forme de schéma une liste ainsi :



a. Notion de structure de données abstraites ou Interface

Pour définir formellement une structure de données, on définit tout d'abord l'ensemble des valeurs de la structure de données et ensuite on donne une description de ses opérations primitives.
C'est ce que l'on appelle l'interface de la structure de données ou la structure de données abstraite.
Cette façon de définir une structure de données ne présume en rien de la façon dont elle va être implémentée, du moment que l'implémentation choisie respecte l'ensemble des valeurs et les opérations primitives.

Dans le cas des listes :
Soit V un ensemble des valeurs d'éléments (comme des entiers, ou des chaînes de caractères par exemple).
On appelle type **Liste de valeurs de V** et on le note Liste(V) l'ensemble des listes dont les éléments sont dans V.

Ensemble définis et utilisés : V, Liste(V), Rang (ensemble des rangs entier y compris `nil` qui est une position fictive).
Il est à noter qu'il n'existe pas de normalisation pour les primitives de manipulation de listes.

Description fonctionnelle des opérations primitives de base communément rencontrées :

- `creer_liste : ()→ Liste(V)`
  fonction retournant la liste vide.
- `ajouter_en_tete : VxListe(V) → Liste(V)`
  insérer un élément en tête de liste.
- `est_vide : Liste(V) → Booléen`
  fonction retournant vrai si la liste est vide et faux sinon
- `tête : Liste(V) → V`
  fonction retournant la tête de liste
- `queue : Liste(V) → Liste(V)`
  fonction retournant la liste privée de son élément de tête.
- `supprimer_en_tete : Liste(V) → Liste(V)`
  supprime l'élément en tête de liste

Remarque : les piles et les files sont des cas particulier de listes.

Exemple d'utilisation dans un environnement théorique de développement :

```
>>>L = creer_liste()  #On crée une liste vide L = ∅
>>> est_vide(L) #On obtient
True
>>> ajouter_en_ tete(1,L) # On a  L = < 1 >
>>> ajouter_en_tete(3, ajouter_en_tete(1, ajouter_en_tete(4,L)))
>>> L #on obtient
< 3 ; <1 ; <4 ; <1; <∅>>>>
>>> queue(L) #on a
>>> < 1 ; <4 ; <1 ; <∅ > >>
>>> tête(L) # on a
>>> 3
```

▶ Exercice 5

Écrire les instructions permettant de créer une liste contenant la liste des voyelles françaises dans l'ordre alphabétique `L2 = < ‘a’ ; <‘e’ ; <‘i’ ; <‘o’ ; <‘u’ ; <‘y’ ; ∅ >>>>>>`.

Quelle suite d'instructions permet d'accéder à la voyelle 'i' ?

b. Implémentation en python

Il est important de noter que le type List proposé nativement par le langage Python n'est pas une implémentation de la structure de données abstraite de liste. Les langages de programmation dit fonctionnelles comme SCHEME, LISP, OCAML ou Haskell intégrent nativement la structure de données liste.
Pour implémenter la structure de données liste en python on peut par exemple utiliser des tuples `(x, L)` ou `x` une valeur et `L` une liste de valeurs

```
def creer_liste():
    return None


def est_vide(L)
    return L == None


def ajouter_en_tete(x, L):
    return (x, L)


def tete(L):
    return L[0]


def queue(L):
    return L[1]
```

▶ Exercice 6

1. À partir de l'implémentation ci-dessous, écrire une fonction `nombre_element` qui prend en argument une liste et retourne son nombre d'éléments.
2. Écrire une fonction `val(i, L)` qui parcourt une liste `L` et retourne l'élément de `L` au rang `i` avec `i` compris entre 0 et le nombre d'éléments de la liste moins 1 (on pourra utiliser des « assert » pour tester cette pré-condition).
3. Combien d'étapes sont nécessaires pour parcourir la liste jusqu'à l'élément de rang `i` ?
4. On donne la fonction `est_dans` définie ci-dessous

```
def est_dans(x, L):
    if est_vide(L):
        return False
    else:
        if x == tete(L):
            return True
        else:
        return False or est_dans(x, queue(L))
```

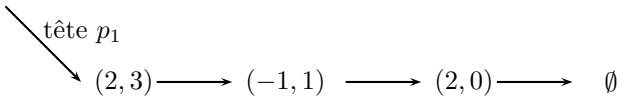
qui retourne vrai si `x` est un élément de `L` et faux sinon. Décrire comment cette fonction procède pour déterminer si une valeur est dans la liste ou non ?
Combien d'étapes dans le pire des cas sont nécessaires pour obtenir une réponse de la fonction `est_dans(x, L)` ?

1.4 Conclusion

De ce premier contact avec les structures de données abstraites ou interface, on peut retenir qu'une interface est définie par ses données, ses opérations et leurs sémantiques.
Cette façon de conceptualiser une interface est à rapprocher des principes de la programmation dite orientée objet du prochain chapitre.

▶ Exercice 7 type bac

L'objectif de cet exercice est de proposer une structure de données permettant de réprésenter des polynômes d'une variables réelle. Un polynôme sera représenté par une liste de monônes. Un monône étant réprésenter par un couple (coeffcient,exposant).
Considérons par exemple le polynôme définie pour tout réel x par $p_1(x) = 2x^3 - x + 2$ il sera représenté ainsi :



On dispose de la structure de données abstraites de liste muni des opérations primitives ci-dessous :

- `creer_liste` qui retourne retourne la liste vide.
- `est_vide` qui retourne vrai si la liste est vide et faux sinon.
- `ajouter_en_tete` qui prend en argument un élement puis une liste et modifie cette liste en y ajoutant cet élément en tête.
- `supprimer_en_tete` qui prend en argument une liste et modifie la liste en supprimant l'élement en tête.
- `tete` qui prend en argument une liste et retourne la tête de celle-ci
- `queue` qui prend en argument une liste et retourne la queue de la liste.

1. À l'aide de ces opérations primitives écrire les instructions permettant de créer une liste noté `p1` représentant le polynôme p_1.

2. Que retourne l'instruction `tete(p1)` ? Même question pour `queue(p1)` ?

3. L'image de 2 par le polynôme p_1 est $p_1(2) = 2 \times 2^3 - \times 2 + 2$ soit $p_1(2) = 16$.
À l'aide des opérations primitives écrire un algorithme `valuation` en langage naturel d'une fonction prenant en argument une liste réprésentant un polynôme et un flottant retournant l'image de ce flottant par ce polynôme.

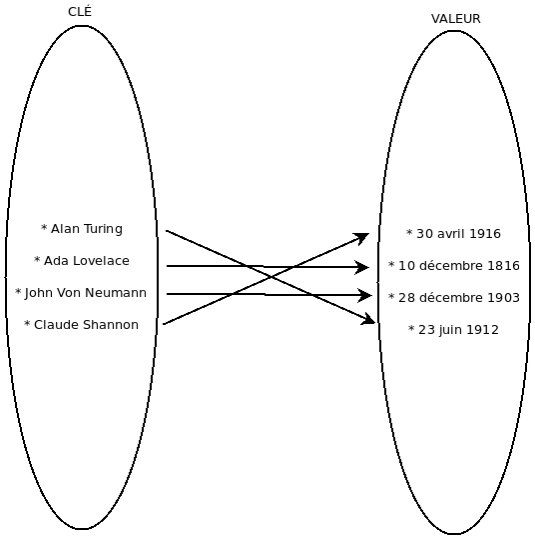
4. *Facultatif :* Écrire un algorithme en langage naturel d'une fonction qui étant donnée deux listes représentant des polynômes retourne la somme de ces deux polynomes.
Exemple : soit p_2 le polynôme défini pour tout réel x par $p_2(x) = 5x^4 + 3x^2 + 1$ on a :
$p_1 + p_2 : x \mapsto 5x^4 + 2x^3 + 3x^2 + 3$.

1.5 Rappels : LES DICTIONNAIRES

Les dictionnaires en tant que structures de données sont nativement présentes en Python et elles ont été étudiées en classe de première. On peut maintenant étudier la structure de données abstraite ou interface à l'origine des dictionnaires implémentés par le langage Python.

a. Définition de la structure abstraite dictionnaire

Contrairement aux listes ou tableaux ou l'on utilise un indice de position pour accéder à un élément dans un dictionnaire, on accède à un élément à l'aide d'une clé. À la manière d'un annuaire ou répertoire téléphonique, le nom d'un contact est la clé permettant d'accéder aux données de ce dernier.



Un dictionnaire est un ensemble constitué de couples clé-valeur dont chaque clé est unique.

Voici les opérations que l'on peut effectuer sur le type abstrait dictionnaire :

- `creer` : on creer un dictionnaire vide
- `ajouter` : on associe une nouvelle valeur à une nouvelle clé
- `modifier` : on modifie un couple clé :valeur en remplaçant la valeur courante par une autre valeur (la clé restant identique)
- `supprimer` : on supprime une clé (et donc la valeur qui lui est associée)
- `rechercher` : on recherche une valeur à l'aide de la clé associée à cette valeur.

b. Implémentation en Python

La structure de dictionnaire est native dans le langage Python, comme on peut le lire dans la documentation officielle `https://docs.python.org/fr/3/tutorial/datastructures.html#dictionaries`
L'implémentation des dictionnaires en python exploite des fonctions dites de hachage. Il s'agit de fonction qui à une chaîne de caractères (la clé) associe une valeur « unique » (unique en théorie mais pas en pratique). Cette valeur « unique » constituera un identifiant qui permet d'associer la clé directement à une adresse en mémoire qui contiendra la valeur correspondant à la clé. Cette implémentation a déjà été étudiée l'année dernière, si besoin se référer aux ressources proposées l'an passé en classe de première.

c. Recherche d'une valeur dans une liste vs dans un dictionnaire

La recherche d'une valeur d'une liste nécessite le parcours de cette liste. Durant ce parcours on compare la valeur recherchée aux valeurs de la liste et dans le pire des cas le parcours de la liste dans son intégralité. Si considère une liste de taille n la recherche d'une valeur dans celle-ci nécessitera un nombre d'opérations de l'ordre de n ce que l'on note $O(n)$.

La recherche d'une valeur dans un dictionnaire se fait à l'aide d'une clé. La fonction de hachage sous-jacente au dictionnaire donne à partir de la clé une adresse qui donne accès directement à la valeur et cela en un nombre d'opération indépendant de la taille du dictionnaire ce que l'on note O(1).

Le coût en O(1) de la recherche de valeur dans un dictionnaire est un des intérêts de la structure dictionnaire.