Il existe plusieurs façons de stocker des données linéaires (nous avons déjà vu les tableaux et les listes). Selon l'utilisation que l'on veut faire de nos données, l'une ou l'autre structure sera plus adaptée, notamment si les données sont statiques ou dynamiques ou encore selon l'ordre de traitement des données. Les structures de *pile* et de *file* fournissent elles aussi des opérations permettant d'ajouter ou supprimer des éléments mais pas selon les même règles : pour la pile le dernier arrivé sera le premier sorti (LIFO en anglais pour « Last In First Out ») alors que pour la file c'est le premier arrivé qui sera le premier sorti (FIFO en anglais pour « First In First Out »).

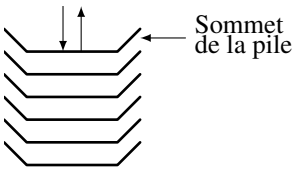
Comme dans le chapitre précédent nous distinguerons le type de données abstrait (interface) des implémentations que nous en ferons.

1 La pile

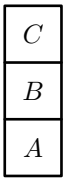
Elle correspond exactement à l'image traditionnelle d'une pile d'assiettes posée sur une table.
On peut donc :

- soit empiler une assiette de plus en haut de la pile,
- soit dépiler les assiettes une par une pour les placer sur la table.

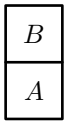
En pratique on ne peut accéder qu'au dernier élément ajouté, appellé *le sommet de la pile*.



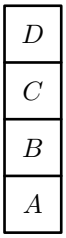
Exemple 1.1 Ainsi, si dans une pile, on a ajouté successivement A, puis B, puis C, on se retrouve dans la situation suivante :



C est empilé sur B, lui-même empilé sur A
Si on veut accéder à l'élément A, il faut d'abord dépiler C, puis B



On peut retirer C de la pile (on dit qu'on *dépile* C),



Ou on peut ajouter un quatrième élément D (on dit qu'on *empile* D).

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 :

- dans les logiciels de traitements de texte : historique des modifications (Ctrl-Z) ;
- dans les navigateurs : historique de navigation ;
- dans la calculatrice HP 15C : notation polonaise inversée ;
- avec les piles d'exécution (vues avec les algorithmes récursifs) qui permettent de garder la trace de l'endroit où chaque fonction active doit retourner à la fin de son exécution (les fonctions actives sont celles qui ont été appelées, mais qui n'ont pas encore terminé leur exécution) ainsi que les valeurs des paramètres et des variables internes au fil de l'exécution.

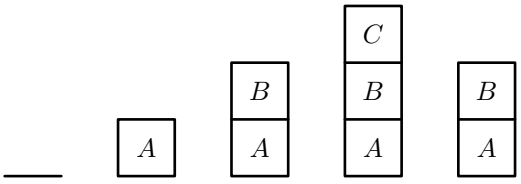
1.1 Opérations caractérisant une pile (interface)

Avant de se pencher sur les différentes façons de réaliser une structure de pile, il faut définir l'ensemble des opérations qu'une telle réalisation doit pouvoir fournir, quelle qu'elle soit. En général, une structure de pile a une interface proposant au moins les quatre opérations suivantes :

- créer une pile vide (fonction `créer_pile()`),
- tester si une pile `p` est vide (fonction `est_vide(p)`),
- ajouter un élément `e` à une pile `p` (fonction `empiler(p,e)`),
- retirer et renvoyer le sommet d'une pile `p` (fonction `dépiler(p)`).

Exemple 1.2 Voici une illustration de l'utilisation de ces opérations :

```
p = creer_pile()
empiler(p, A)
empiler(p, B)
empiler(p, C)
depiler(p)
```



Les opérations `dépiler` et `empiler` modifient le contenu de la pile passée en argument. On peut également définir d'autres opérations, celles-ci ne modifiant pas la pile, comme `taille(p)` qui renvoie le nombre d'éléments contenus dans la pile `p` et `sommet(p)` qui renvoie le sommet de la pile `p`, sans la modifier.

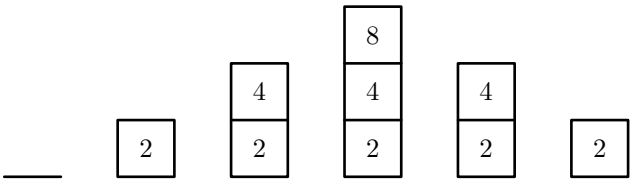
Exercice 1.1 Pour chacune des tâches suivantes, dites si elle est adaptée à une pile.

1. Représenter une répertoire téléphonique.
2. Stocker l'historique des actions effectuées dans un logiciel et disposer d'une commande annuler.
3. Comptabiliser les pièces ramassées et dépensées par un personnage dans un jeu.
4. Ranger des dossiers à traiter sur un bureau.

Exercice 1.2 En s'inspirant des figures précédentes, illustrer le résultat de chacune des opérations de la séquence suivante :

```
p = creer_pile()
empiler(p,4)
empiler(p,1)
depiler(p)
empiler(p,8)
depiler(p)
```

Exercice 1.3 Donner les différentes opérations de la séquence illustrée suivante :



1.2 Implémentation d'une pile avec le type `list` de `Python`

Pour construire la structure de piles, on utilise ici le type `list` de Python. On pourra utiliser la possibilité d'ajouter ou de supprimer des éléments à l'extrémité d'une liste.

Exercice 1.4

1. Quelles sont les méthodes qui permettent de réaliser les opérations décrites ci-dessus avec le type `list` de `Python` ?

2. Compléter les fonctions `créer_pile()`, `dépiler(p)` et `empiler(p,v)`.

```
def créer_pile() :
    ......................
```

```
def dépiler(p) :
    assert len(p)>0
    ......................
```

```
def empiler(p,v) :
    ......................
```

3. Compléter les fonctions `taille(p)`, `est_vide(p)` et `sommet(p)`.

```
def taille(p) :
    ......................
```

```
def est_vide(p) :
    ......................
```

```
def sommet(p) :
    ......................
```

4. Tester vos fonctions avec la séquence de l'exercice 1.2.

1.3 Implémentation d'une pile avec une liste chaînée

La structure de liste chaînée (voir chapitre 1) donne une manière élémentaire de réaliser une pile. Empiler un nouvel élément revient à ajouter une nouvelle cellule en tête de liste, tandis que dépiler un élément revient à supprimer la cellule de tête.

On peut donc construire une classe `Pile` définie par un unique attribut `contenu` associé à l'ensemble des éléments de la pile, stockés sous la forme d'une liste chaînée.

On reprend la classe `Cellule` définie dans le chapitre 1 pour implémenter des liste chaînées ainsi que la classe `Pile` ayant les fonctionnalités décrites ci-dessus :

```
class Cellule :
    """Une cellule d'une liste chaînée"""
    def __init__(self, tete, queue) :
        self.car = tete
        self.cdr = queue

class Pile :
    """Structure de pile"""
    def __init__(self) :
        self.contenu = None
```

Pour pouvoir créer une pile vide il suffit alors simplement de définir la fonction `créer_Pile()` suivante :

```
def créer_Pile() :
    return Pile()
```

Exercice 1.5

1. Avec cette définition, la pile vide est simplement représentée par un contenu valant `None` (une liste chaînée vide). Compléter la méthode `est_vide()` qui renvoie `True` si la pile est vide, `False` sinon :

```
class Pile :
    ...
    def est_vide(self) :

        ..............................
```

2. (a) Pour empiler un nouvel élément `elem` dans une pile, on doit créer un nouvel objet de classe `Cellule` dont la tête (attribut `car`) doit être ..... et la queue (attribut `cdr`) doit être .....

   (b) Compléter la méthode `empiler` de la classe `Pile` :

```
class Pile :
    ...
    def empiler(self,elem) :

        ..............................................
```

3. Pour dépiler un élément d'une pile, quand elle n'est pas vide, il s'agit de récupérer la valeur de la tête de la cellule stockée dans l'attribut `contenu` puis la supprimer en mettant à jour cet attribut. Compléter la méthode `dépiler()` suivante :

```
class Pile :
    ...
    def dépiler(self) :
        assert not self.est_vide(), "on ne peut dépiler une pile vide"

        sommet = ...........

        self.contenu = ..............................

        return ...........
```

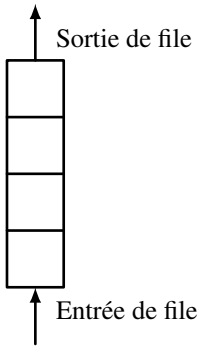
4. Tester vos méthodes avec la séquence de l'exercice 1.2 en adaptant la syntaxe.

2 La file

Dans une structure de file, chaque opération de retrait retire l'élément qui avait été ajouté en dernier.

On associe cette structure à l'image d'un file d'attente dans laquelle les personnes arrivent à tour de rôle, patientent, et sont servies dans leur ordre d'arrivée.

Le premier arrivé est donc le premier servi, c'est le principe *FIFO* (*First In, First Out*).



On retrouve également souvent ce principe FIFO en informatique :

- accès aux ressources d'un réseau ;
- gestion des stocks (base de données des produits d'un supermarché en ligne par exemple) ;
- serveurs d'impression, qui traitent ainsi les requêtes dans l'ordre dans lequel elles arrivent, et les insèrent dans une file d'attente ;
- gestion de processus sans priorité (moteurs multitâches accordant le temps-machine à chaque tâche sans en privilégier aucune).

2.1 Opérations caractérisant une file (interface)

Avant de se pencher sur les différentes façons de réaliser une structure de pile, il faut définir l'ensemble des opérations qu'une telle réalisation doit pouvoir fournir, quelle qu'elle soit.

Les trois opérations principales sont réalisées par les fonctions :

- `créer_file()` renvoie une nouvelle file vide,
- `défiler(f)` défile et renvoie la valeur retirée de la file `f`,
- `enfiler(f,v)` enfile la valeur `v` dans la file `f`.

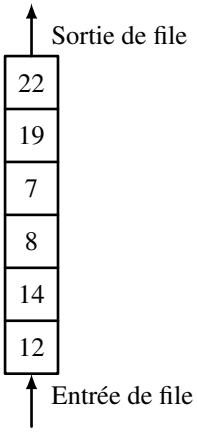
D'autres opérations sont disponibles mais ne modifient pas la file :

- `taille_file(f)` renvoie le nombre d'éléments contenus dans la file `f`,
- `file_vide(f)` indique si la file `f` est vide.

Exemple 2.1 Soit une file `f` composée des éléments suivants telle que :

```
f = créer_file() ; enfiler(f,22) ; enfiler(f,19) ; enfiler(f,7) ; enfiler(f,8) ;
                          enfiler(f,14) ; enfiler(f,12)
```

- ⋆ L'état de la file `f` est illustrée ci-contre. A partir d'une file vide on a enfilé successivement les valeurs `22,19,7,8,14` et `12`. Ainsi le premier arrivé ici est `22` et le dernier arrivé est `12`.
- ⋆ L'opération `taille_file(f)` renvoie `6`, le nombre d'éléments de la file.
- ⋆ L'opération `enfiler(f,42)` ajoute à la file, sous `12`, le nombre `42`.
- ⋆ L'opération `defiler(f)` renvoie `22` qu'elle supprime de la file.
- ⋆ L'opération `file_vide(f)` renvoie `False` mais si on applique `défiler(f)` 6 fois de suite sur la file initiale, l'opération `file_vide(f)` renvoie `True`



Exercice 2.1 En s'inspirant de la figure précédente, illustrer le résultat de chacune des opérations de la séquence suivante sur une file `f` initialement vide.

```
enfiler(f,4)
enfiler(f,1)
défiler(f)
enfiler(f,8)
défiler(f)
```

Quelle instruction permet de construire directement la file résultante ?

2.2 Implémentation d'une file avec le type `list` de `Python`

Pour construire la structure de files, comme pour les piles on peut choisir d'utilise ici le type `list` de Python. On pourra utiliser la possibilité d'ajouter ou de supprimer des éléments à l'extrémité d'une liste.

Exercice 2.2

1. Compléter les fonctions `créer_file()`, `défiler(f)` et `enfiler(f,v)`.

```
def créer_file() :

    ......................
```

```
def défiler(f) :

    assert ...........

    ......................
```

```
def enfiler(f,v) :

    ......................
```

2. Compléter les fonctions `est_vide(f)` et `taille_file(f)`.

```
def est_vide(f) :

    ......................
```

```
def taille_file(f) :

    ......................
```

3. Tester vos fonctions avec la séquence de l'exercice 2.1.

2.3 Implémentation d'une file avec deux piles

Une autre façon de réaliser une file est d'utiliser deux piles. On considère deux piles notée `entrée` et `sortie`. La première va permettre de stocker les éléments à enfiler, qui seront donc empilés dans l'ordre d'arrivée, et la deuxième va permettre de défiler les éléments en respectant le principe FIFO.

Exercice 2.3

1. Quelles sont les opérations sur la pile `entrée` pour stocker les éléments 12, 5, 7 donnés dans l'ordre d'entrée ?

2. On souhaite défiler le premier arrivé. Pour cela on va dépiler de la pile `entrée` et empiler dans la pile `sortie` jusqu'à ce que la pile `entrée` soit vide.
   (a) Quel est l'état de la pile `sortie` après ces opérations ?
   (b) Quel dernière opération doit-on réaliser pour récupérer le premier arrivé ?
3. On doit maintenant enfiler un nouvel arrivé, le nombre 17. Les piles `entree` et `sortie` sont dans l'état obtenu à l'étape précédente. Dans quel pile doit-on ajouter 17 ?
4. A l'aide des questions précédentes, compléter l'implémentation suivante puis tester votre code (séquence de l'exercice 2.1) :

```
class File :
    """Implémentation d'une file avec deux piles"""
    def __init__(self) :
        self.entree = créer_pile()
        self.sortie = créer_pile()

    def est_vide(self) :
        return est_vide(self.entree) and est_vide(self.sortie)

    def enfiler(self,x) :

        ..................................

    def défiler(self) :
        if est_vide(self.sortie) :

            ..........................................

                ..........................................

        assert not est_vide(self.sortie), "on ne peut pas défiler une file vide"

        return ..................................
```

Exercice 2.4 (Réciproquement) Expliquer comment on pourrait implémenter une pile à l'aide de deux files.

2.4 Implémentation d'une file avec une liste chaînée mutable

Comme pour les files, on peut utiliser les liste chaînées pour réaliser une structure de file. Comme pour les piles, on peut retirer la cellule de tête pour défiler un élément, par contre, enfiler un élément revient cette fois à modifier la dernière cellule (une mutation intervient à cet endroit). De plus nous avons maintenant besoin de pouvoir accéder à cette dernière cellule. Il est bien sûr possible de parcourir toute la liste de cellules pour la trouver, mais un moyen plus efficace est d'avoir une attribut `derniere` qui permet de retrouver directement la dernière cellule.

On en propose ici une implémentation :

```
class Cellule :
    """Une cellule d'une liste chaînée"""
    def __init__(self, tete, queue) :
        self.car = tete
        self.cdr = queue

class File :
    """Structure de file avec liste chaînée mutable"""
    def __init__(self) :
        self.premiere = None
        self.derniere = None

    def est_vide(self) :
        return self.premiere is None

    def enfiler(self,x) :
        c = Cellule(x, None)
        if self.est_vide() :
            self.premiere = c
        else :
            self.derniere.cdr = c
        self.derniere = c

    def défiler(self) :
        assert not self.est_vide(), "on ne peut pas défiler une file vide"
        v = self.premiere.car
        self.premiere = self.premiere.cdr
        if self.premiere is None :
            self.derniere = None
        return v
```

Exercice 2.5 Tester cette implémentation des files avec la séquence de l'exercice 2.1 en adaptant la syntaxe.