# TP N°3 : Les Files

**Avant de commencer** : Créez un dossier *TP_3* dans *« D :\votre_groupe\votre_nom_prenom »* dans lequel vous placerez tous vos scripts que vous créerez durant ce TP.

## 1 Définition

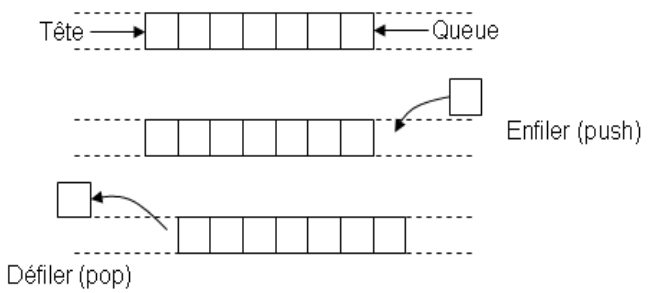
Une **file** est un conteneur dans lequel on peut ajouter des objets, et retirer à tout moment le premier objet ajouté parmi ceux restants. Il correspond à l'idée qu'on se fait d'une file d'attente, où les personnes qui arrivent se placent en **queue** de file, et attendent d'arriver en **tête** avant d'en sortir.

On dit qu'il s'agit d'une structure **FIFO** (**F**irst-**I**n **F**irst-**O**ut), c'est-à-dire que le premier élément qui a été ajouté dans la liste sera aussi le premier qui en sortira.

La première valeur de la file est la *tête* de la file et la dernière est la *queue* de la file.

Les opérations ordinaires avec les files sont :

- Enfiler (enqueue) : Ajouter en fin de la file.
- Défiler (dequeue) : Supprimer le premier élément (sommet) de la file.




Exemples : En Informatique, les files sont utilisées pour

- Mémoriser les données en attente de traitement.
- Ordonnancer les tâches d'impression.
- Ordonnancer les processus au niveau du système d'exploitation.

Il y a beaucoup de façons de concevoir une file : listes, tableaux,.... La plus simple consiste à utiliser un conteneur de type **list**.

## 2 Mise en oeuvre d'une file avec une liste

On peut utiliser les listes pour réaliser une file. Cependant, les listes ne sont pas très efficaces dans ce cas précis (*First In First Out*) : alors que les ajouts et suppressions en fin de liste sont rapides, les opérations d'insertions ou de retraits en début de liste sont lentes (car tous les autres éléments doivent être décalés d'une position).

**Indications pour la réalisation en Python**

Le type `'list'` intègre déjà toutes les méthodes pour réaliser cette structure de données :

- `list.append(x)` qui ajoute un élément à la fin de la liste.
- `list.insert(i, x)` qui insère un élément à la position indiquée. Le premier argument est la position de l'élément courant avant lequel l'insertion doit s'effectuer, donc `a.insert(0, x)` insère l'élément en tête de la liste, et `a.insert(len(a), x)` est équivalent à `a.append(x)`.
- `list.remove(x)` qui supprime de la liste le premier élément dont la valeur est x. Une exception est levée s'il n'existe aucun élément avec cette valeur.
- `list.pop([i])`, qui enlève de la liste l'élément situé à la position indiquée, et le retourne. Si aucune position n'est indiqué, `a.pop()` enlève et retourne le dernier élément de la liste

Programmer le module `file.py` contenant les fonctions suivantes :

1. `creer_file()` : crée et retourne une file vide non bornée (dont la taille maximale n'est pas définie, donc on peut y emfiler autant d'éléments que l'on veut, dans la limite de la mémoire de l'ordinateur).
2. `file_vide(f)` : retourne `True` si la file `f` est vide et `False` sinon.
3. `sommet(f)` : renvoie l'élément du sommet (tête) de la file `f`, sans le supprimer.

4. `taille(f)` : renvoie la taille de la file `f`.
5. `enfiler(f,x)` : ajoute l'élément $x$ au queue de la file `f`.
6. `defiler(f)` : supprime le l'élément située à la tête de la file `f` et le retourne .

# 3 Exercices

**Exercice 1**

On pourra éventuellement utiliser une ou des piles/files temporaires, on utilisera les primitives `creer_pile()` qui renvoie une pile vide et `creer_file()` qui renvoie une file vide.
Écrivez les fonctions suivantes :

1. `afficher(f)` : cette fonction affiche tous les éléments de la file `f`.
2. `defilerJusqua(f,x)` : cette méthode défile la file `f` jusqu'à l'élément $x$. L'élément $x$ n'est pas défilé. Si l'élément $x$ n'appartient pas à la file, alors la méthode défile tous les éléments de la file.
3. `appartient(f,x)` : cette fonction s'applique sur une file `f`. Elle renvoie **True** si l'élément $x$ appartient à la file, et **False** sinon. Attention : il est important que la file ne change pas.
4. `inverser_file(f)` : cette fonction inverse les éléments de la file `f`. On a le droit d'utiliser des files ou des piles temporaires.

**Exercice 2**

Les *nombres de Hamming* sont les entiers naturels non nuls dont les seuls facteurs premiers éventuels sont 2, 3 et 5.
Exemples : 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, ...
Le but de cet exercice est de les générer de manière croissante. Évidemment, on peut parcourir un à un tous les entiers en testant à chaque fois si ceux-ci sont des entiers de Hamming, mais cette démarche montre vite des limites (Le 2000e entier de Hamming est égal à 8 100 000 000 et le 2001e à 8 153 726 976 : il faudrait tester plus de 53 millions de nombres avant d'augmenter notre liste d'un élément !).
On adopte donc la démarche suivante : on utilise trois files `f2` ,`f3` et `f5` contenant initialement le nombre 1, et on suit la démarche suivante :

1. On détermine le plus petit des trois têtes des files, que l'on note $k$ et que l'on imprime à l'écran ;
2. On retire cet élément des files où il se trouve ;
3. On insère en queue des files `f2` ,`f3` et `f5` les entiers $2k$, $3k$ et $5k$.

Cette démarche utilise le fait que tout nombre de Hamming différent de 1 est le produit par 2, 3 ou 5 d'un nombre de Hamming plus petit.
Travail à faire : Rédiger une fonction Python permettant l'affichage des $n$ premiers nombres de Hamming.

**Exercice 3**

Une structure de **File** fonctionne sur le principe **premier entré-premier sorti** (comme les files d'attentes à un guichet).
Une façon d'implémenter une File contenant au plus $n$ éléments est de créer $F = [queue, tete, [0,0,\ldots,0]]$ où la sous-liste $F[2]$ de $F$ est initialisée avec $n$ zéros. Les éléments `tete` et `queue` sont deux entiers.
Le fonctionnement est le suivant :

- À la création de la File `F` : `tete` et `queue` sont initialisées à zéros.
- Quand on insère un élément `x` dans la File `F`, on le met dans `F[2][queue]`, puis `queue` est augmenté d'une unité.
- Si on retire un élément de la File `F`, on le retire de `F[2][tete]`, puis on augmente `tete` d'une unité. La fonction qui gère cela doit retourner la valeur retirée.
- Si `tete` ou `queue` atteignent la fin de `F[2]`, ils doivent pouvoir revenir à 0 : la gestion des indices `tete` et `queue` se fait donc de manière circulaire.
- La File `F` est vide lorsque `tete` égale à `queue`.
- La File `F` est pleine lorsque la différence entre la `tete` et la `queue` égale à $n$.

Écrire les fonctions Python suivantes :

1. `creer_file(n)`, `taille(f)`, `file_vide(f)`, `sommet(f)`.
2. `enfiler(f, x)` qui insère l'élément `x` dans la file `f` si celle-ci n'est pas pleine.
3. `defiler(f)` qui retire l'élément correct (sommet) de la file `f`,si celle-ci n'est pas vide, et le renvoie.