

Chapitre 1.

Structures de données avancées

Les Files**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 (First-In First-Out)**, c'est-à-dire que le premier élément qui a été ajouté dans la liste sera aussi le premier qui en sortira.

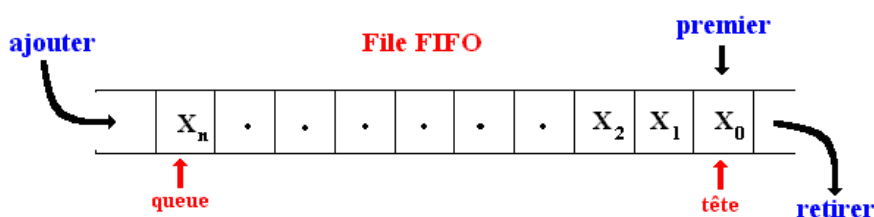
La première valeur de la file est la **tête** de la file et la dernière est la **queue** de la file. On peut utiliser les listes pour réaliser une file.

Exemples : En Informatique, les files sont utilisées pour :

- Ordonnancer les tâches d'impression,
- Ordonnancer les processus au niveau du système d'exploitation, ...

Les opérations ordinaires sont :

- **ENFILER (enqueue/push)** permettant d'ajouter en fin de la file
- **DEFILER (dequeue/pop)** permettant de supprimer le premier élément (sommet/tête) de la file.

**Exercices d'application : Mise en œuvre d'une file avec une liste****i** Indications pour la réalisation en Python :

Le type « **list** » intègre déjà toutes les méthodes pour réaliser ces deux structures de données :

- **L.append(x)** qui ajoute un élément à la fin de la liste.
- **L.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 **L.insert(0, x)** insère l'élément en tête de la liste, et **L.insert(len(a), x)** est équivalent à **L.append(x)**.
- **L.remove(x)** qui supprime de la liste le premier élément dont la valeur est x. Une exception est levée s'il existe aucun élément avec cette valeur.
- **L.pop(i)**, qui enlève de la liste l'élément situé à la position indiquée, et le retourne. Si aucune position n'est indiquée **L.pop()** enlève et retourne le dernier élément de la liste.

Exercice 1 : Mise en œuvre d'une file avec une liste

Créer un module Python « **file.py** » contenant les fonctions suivantes :

1. Ecrire une fonction **creer_file()** permettant de créer et renvoyer une file vide.
2. Ecrire une fonction **file_vide(f)** qui retourne **True** si la file **f** est vide et **False** sinon.

3. Écrire une fonction **sommet(f)** qui renvoie le premier élément (tête) de la file.
4. Ecrire la fonction **enfiler(f, x)** qui ajoute dans la file **f** l'élément **x**, uniquement à l'aide de la méthode **append** la classe **list**.
5. Ecrire la fonction **defiler(f)** qui supprime de la file le premier élément (tête de la file), uniquement à l'aide de la méthode **pop** de la classe **list**. Que se passe-t-il quand on essaye de défiler une file vide ? On y mettra des instructions de sortie brutale du programme.
6. Ecrire la fonction **afficher(f)** qui affiche tous les éléments de la file **f**.
7. Ecrire la fonction **defilerJusqua(f, x)** qui défile la file **f** jusqu'à l'élément **x**. L'élément **x** n'est pas défilé, il sera le sommet de la file. Au cas où l'élément **x** n'appartient pas à la file, la méthode défile tous les éléments de la file.
8. Ecrire la fonction **appartient(f,e)** qui renvoie **True** si l'élément **e** appartient à la file, **False** sinon. Attention : il est important que la file ne change pas à la fin de la fonction.
9. Ecrire la fonction **inverser_file(f)** qui inverse les éléments de la file à laquelle elle est appliquée. On a le droit d'utiliser des files ou des piles temporaires.

Exercice 2 :

Une structure de File fonctionne sur le principe premier entré-premier sorti (comme les files d'attentes à un guichet). On se propose ici de décrire une implémentation d'une file **F** à l'aide d'un tableau de taille **n**. Bien sûr cela implique que la file a une capacité finie. Une file sera constituée d'éléments contigus de ce tableau.

Une façon d'implémenter une file contenant au plus **n** éléments est de créer une liste

F = [queue, tete, [0,0, ...,0]] où la sous-liste **F[2]** de **F** est initialisée avec **n** zéros.

Il suffira donc pour représenter la liste de gérer deux indices entiers **tete** et **queue**: l'indice **tete** marque le début de la file, et **queue** en marque la fin.

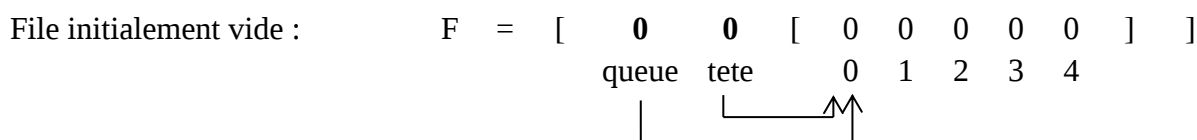
Plus précisément **tete** est l'indice de la case de la sous-liste où on va chercher le prochain élément à défiler, et **queue** est l'indice de la case de la sous-liste où on va enfiler la prochaine valeur.

Le fonctionnement est le suivant :

- À la création de la File **F**, **tete** et **queue** sont initialisés à zéro. Ainsi **F[2]** est initialisée avec **n** zéros.
- Quand on insère un élément 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.

Voici par exemple une représentation de la file **(3, 7, 0, 9)** au sein d'une sous-liste de **5** cases.

L'ordre d'insertion a donc été **3, 7, 0, 9** :



File non pleine : Après avoir enfilé les entiers **3, 7, 0, 9**

F = [4 0 [3 7 0 9 0]]

queue tete 0 1 2 3 4

Attention, on utilise un tableau dit circulaire, ce qui veut dire que si on a besoin de cellule au-delà du bord droit du tableau **F[2]**, on va les chercher à gauche, si elles sont libres.

Il est donc tout à fait possible d'avoir **queue < tete**, comme par exemple dans la représentation suivante de cette même liste :

File non pleine : Après avoir défilé **3** et **7** et enfilé les entiers **8** et **10**

F = [1 2 [10 7 0 9 8]]

queue tete 0 1 2 3 4

Une conséquence de ce choix est que l'égalité **tete = queue** ne permet pas de distinguer entre la file vide et la file pleine ; pour pallier ce problème il suffit de ne pas limiter **tete** et **queue** aux valeurs entre **0** et **n-1** lors des enfilements et des défilements. En revanche pour trouver le contenu des bonnes cases dans le tableau (sous liste) il faudra bien sûr travailler **modulo n**.

Travail à faire : Définir les fonctions primitives suivantes :

1. Écrire une fonction **creerFile(n)** permettant de créer et renvoyer une file vide de taille **n**.
2. Écrire une fonction **estVide(F)** permettant de renvoyer **True** si la file **F** passée en paramètre est vide, sinon elle renvoie **False**.
3. Écrire une fonction **estPleine(F)** permettant de renvoyer **True** si la file **F** passée en paramètre est pleine, sinon elle renvoie **False**.
4. Écrire une fonction **enfiler(F, x)** qui insère l'élément **x** dans la file **F** si celle-ci n'est pas pleine sinon on peut provoquer l'arrêt du programme.
5. Écrire une fonction **defiler(F)** qui retire l'élément correct de la File **F** et renvoie cet élément. Si la file est déjà vide, on provoque un arrêt du programme.
6. Écrire une fonction **sommet(F)** qui renvoie l'élément à la tête de la File **F** sans le retirer de la file. Si la file est déjà vide, on provoque un arrêt du programme.

Exercice 3 : Nombres de Hamming

Les nombres de Hamming sont les entiers naturels non nuls dont les seuls facteurs premiers éventuels sont **2, 3** et **5** : **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 2000^e entier de Hamming est égal à 8 100 000 000 et le 2001^e 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, f5** contenant initialement le nombre **1**, et on suit la démarche suivante :

1. On détermine le plus petit des trois têtes de file, 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 :

En utilisant les fonctions définies dans l'exercice précédent, rédiger une fonction Python permettant l'affichage des **n** premiers nombres de Hamming.