Chapitre 8 - Structures linéaires : listes, piles, files

De nombreux algorithmes ”classiques” manipulent des structures de données plus complexes que des simples nombres. Nous allons ici voir quelques-unes de ces structures de données. Nous allons commencer par des types de structures relativement simples :

- les listes
- les piles
- les files

Ces trois types de structures sont qualifiés de **linéaires**.

I - Listes

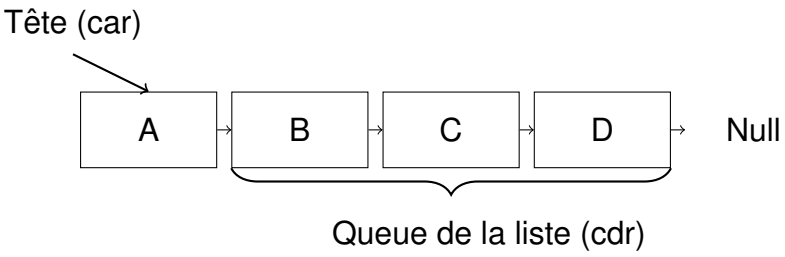
Définition 1

Une **liste** est une structure de données linéaire permettant de regrouper des données. Une liste `L` est composée de 2 parties :

- sa **tête** (souvent noté `car`), qui correspond au **dernier élément** ajouté à la liste.
- sa **queue** (souvent noté **cdr**) qui correspond au **reste de la liste**.
  La queue est également une liste

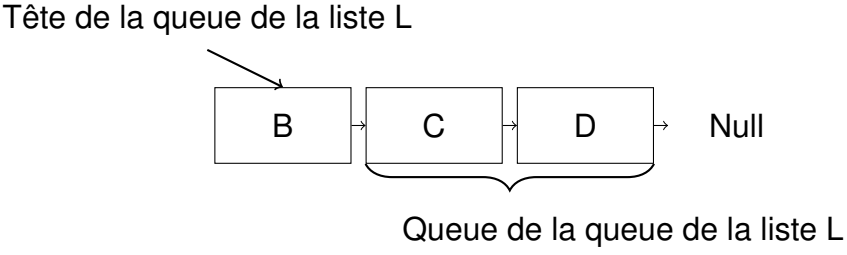
On utilise parfois le terme de *liste chaînée* car les éléments de la liste sont réliés entre et forme une chaîne.
On considère alors que chaque élément possède un élément suivant.
Par définition l'élémént suivant le dernier élément de la liste est `Null` ou `None`

Exemple : liste `L`



Dans cette exemple :

- La **tête** de la liste `L` est l'élément `A`.
- La **queue** de la liste `L` est la liste composée des éléments `B`, `C` et `D`



- L'élément `B` à pour suivant l'élément `C`.
- Le dernier élément de la liste `L` est l'élément `D`. Son suivant est un élément `Null`.

Note : 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”).

Exemple d’interface d’une implémentation du type `Liste`

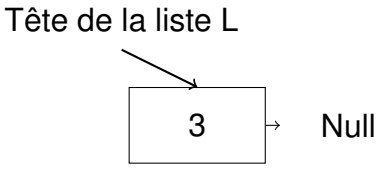
Voici un exemple d’interface proposant les opérations qui peuvent être effectuées sur une liste :

- créer une liste vide : `L=vide()` (on a créé une liste `L` vide).
- tester si une liste est vide : `estVide(L)` renvoie `vrai` si la liste `L` est vide, `faux` sinon.
- construire une liste à partir d’un élément qui sera la tête et d’un liste qui sera la queue : `L1 = cons(x,L)`. Il est possible ”d’enchaîner” les `cons` et d’obtenir ce genre de structure de liste chaînée : `L2 = cons(x, cons(y, cons(z,L)))`.
- ajouter un élément en tête de liste : `ajouteEnTete (x,L)` avec `L` une liste et `x` l’élément à ajouter.
- supprimer la tête `x` d’une liste `L` et renvoyer cette tête `x` : `tete = supprEnTete(L)`.
- `car(L)` : renvoi la tête de la liste `L` sans suppression.
- `cdr(L)` : renvoi la queue de la liste `L`.
- Compter le nombre d’éléments présents dans une liste : `compte(L)` renvoie le nombre d’éléments présents dans la liste `L`.

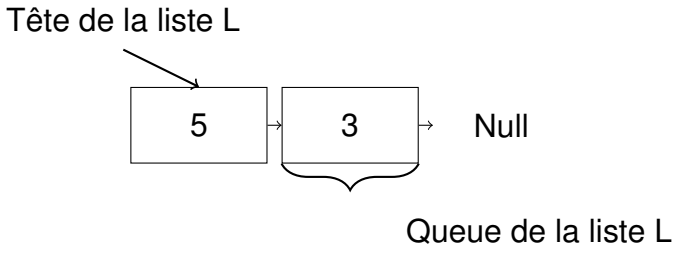
Cas pratique :

Voici une série d’instructions (les instructions ci-dessous s’enchaînent) :

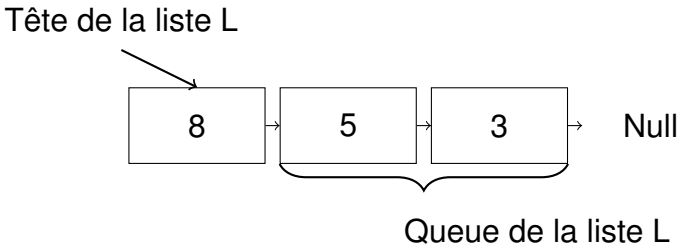
- `L=vide()` ⇒ on a créé une liste vide
- `estVide(L)` ⇒ renvoie vrai
- `ajoutEnTete(3,L)` ⇒ La liste `L` contient maintenant l’élément 3. La queue est vide.



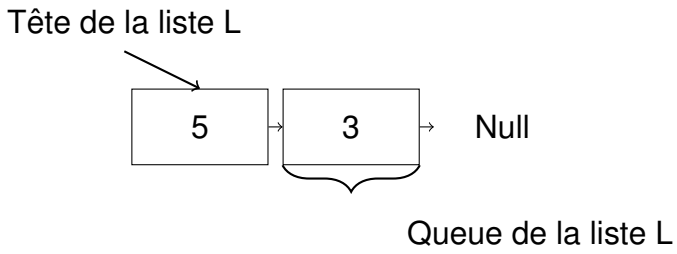
- `estVide(L)` ⇒ renvoie faux
- `ajoutEnTete(5,L)` ⇒ la tête de la liste `L` correspond à 5, la queue contient l’élément 3.



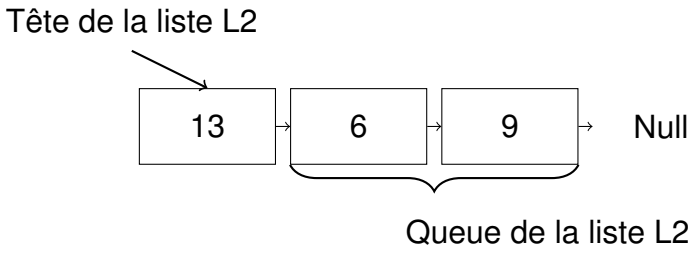
- `ajoutEnTete(8,L)` ⇒ la tête de la liste `L` correspond à 8, la queue contient les éléments 3 et 5.



- `t = supprEnTete(L)` ⇒ la variable `t` vaut 8, la tête de `L` correspond à 5 et la queue contient l'élément 3.



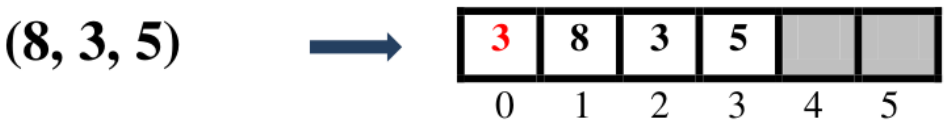
- `L1 = vide()`
- `L2 = cons(13, cons(6, cons(9, L1)))` ⇒ La tête de `L2` correspond à 13 et la queue contient les éléments 6 et 9.



Représentation d'une liste à l'aide d'un tableau

Il existe plusieurs façons de concevoir une liste. La plus simple consiste à utiliser un tableau (de taille fixe) dont chaque élément est identifié par son indice. Tous les langages de programmation permettent de réaliser des tableaux. Pour le langage Python, le plus simple est d'utiliser des listes. On peut par exemple réaliser une liste capable de contenir n éléments avec un tableau `L[0 ... n]` pouvant contenir $n+1$ éléments :

- La première case du tableau d'indice 0 contient le nombre d'éléments présents dans la liste.
- Les cases suivantes du tableau d'indice 1 à n, contiennent les éléments de la liste ou sont vides.



Représentation d'une liste de taille maximale 5 éléments

Remarque

- Si `L[0]==0`, la liste est vide. A chaque fois qu'on insère un élément dans la liste, on augmente `L[0]` d'une unité.
- Lorsque `L[0]==n`, la liste est pleine. De la même façon, lorsque l'on supprime un élément, on diminue `L[0]` d'une unité.

Voici une implémentation python de 2 fonctions `inserer` et `supprimer`. La liste est représentée par un tableau python.

- Insertion d'un élément `x` à l'indice `i` de la liste `L` :
  (a) S'il y a de la place dans la liste et i est un indice valide, il faut décaler les éléments d'indice supérieurs ou égaux à `i` vers la droite.
  (b) On met à jour la case d'indice `i` avec la valeur `x`.
  (c) On incrémente le nombre d'élément de la liste (case 0 du tableau).

```
def inserer(L, x, i) :
  if (L[0]==len(L)-1) or (i - 1>L[0]) or i < 1:
    print("Indice invalide ou liste pleine")
    return False
  else:
    for k in range(L[0]+1,i,-1):
      L[k] = L[k-1]
    L[i]=x
    L[0]=L[0] + 1
    return True
```

- Insertion d'un élément `x` à l'indice `i` de la liste `L` :
  (a) Si la liste n'est pas vide et si l'élément *i* existe, il faut décaler les éléments d'indice strictement supérieurs à `i` vers d'une unité vers la gauche.
  (b) On décrémente le nombre d'élément de la liste (case 0 du tableau).

```
def supprimer(L, i) :
  if (L[0] !=0) and (i <=L[0]) :
    for k in range(i,L[0],1):
      L[k]= L[k+1]
    L[0]= L[0] - 1
    return True
  else:
    return False
```

Exemple d'utilisation de cette implémentation

```
taille = 3
ma_liste = [0]*(taille+1) # +1 pour la premiere case qui contient le nombre d'
   element de la liste
print(ma_liste)          #OUTPUT : [0, 0, 0, 0]
inserer(ma_liste,5,1)
print(ma_liste)          #OUTPUT : [1, 5, 0, 0]
inserer(ma_liste,2,1)
print(ma_liste)          #OUTPUT : [2, 2, 5, 0]
inserer(ma_liste,7,3)
print(ma_liste)          #OUTPUT : [3, 2, 5, 7]
inserer(ma_liste,9,4)    #OUTPUT : Indice invalide ou liste pleine
print(ma_liste)          #OUTPUT : [3, 2, 5, 7]
supprimer(ma_liste, 1)
print(ma_liste)          #OUTPUT : [2, 5, 7, 7]
supprimer(ma_liste, 1)
print(ma_liste)          #OUTPUT : [1, 7, 7, 7]
```

Remarque : Les cases 2 et 3 du tableau python contiennent encore les valeurs 7, mais ces valeurs n'appartiennent pas à la liste `ma_liste`.

II - Piles

Définition 2

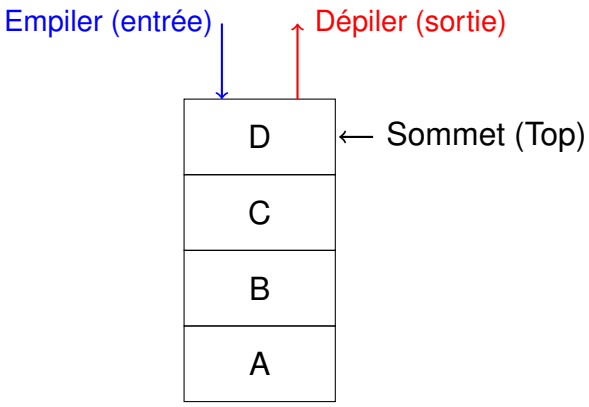
Il s'agit d'une structure de données qui donne accès en priorité aux dernières données ajoutées.
Ainsi, **la dernière information ajoutée sera la première à en sortir**.
Autrement dit, on ne peut accéder qu'à l'objet situé au sommet de la pile.
On décrit souvent ce comportement par l'expression « dernier entré, premier sorti » ou encore **LIFO : Last In, First Out**.
Le dernier élément inséré dans la pile s'appelle le **sommet** de la pile.

- L'action d'ajouter un élément dans une pile s'appelle **empiler**.
- L'action de supprimer un élément dans une pile s'appelle **dépiler**.

Exemple : pile `P`

Dans cet exemple :

- Le somment de la pile est l'élément `D`.
- Le premier élément empilé est l'élément `A`.
- Le porchain élément susceptible d'être dépilé est l'élément `D`.



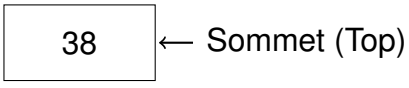
Exemple d'interface d'une implémentation du type `Pile`

Voici un exemple d'interface proposant les opérations qui peuvent être effectuées sur une pile :

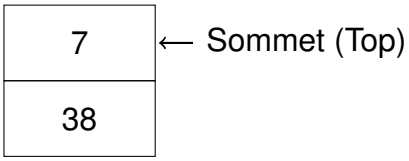
- `pileVide()` : retourne une pile vide.
- `estVide(P)` : retourne vrai si la pile `P` est vide, faux sinon.
- `push(P,x)` : empile l'élément `x` dans la pile `P`.
- `pop(P)` : dépile le sommet de la pile `P`. On récupère l'élément au sommet de la pile `P` tout en le supprimant.
- `sommet(P)` : retourne l'élément situé au sommet de la pile sans le supprimer de la pile.
- `taille(P)` : retourne le nombre d'éléments présents dans la pile `P`.

Cas pratique :

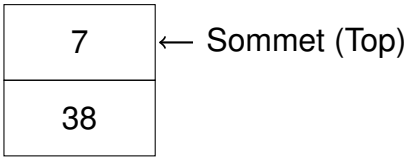
- `P = pileVide()` ⇒ on a créé une pile vide
- `estVide(P)` ⇒ renvoie vrai
- `taille(P)` ⇒ renvoie 0
- `push(P,38)` ⇒ empile l'élémént 7 dans la pile `P`



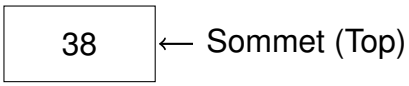
- `push(P,7)` ⇒ empile l'élément 7 dans la pile `P`



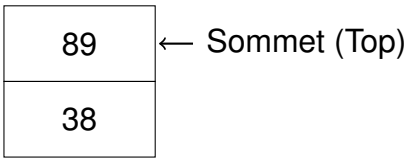
- `estVide(P)` ⇒ renvoie faux
- `taille(P)` ⇒ renvoie 2
- `sommet(P)` ⇒ renvoie 7



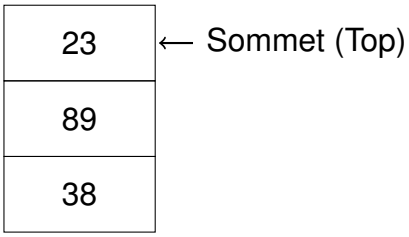
- `pop(P)` ⇒ renvoi 7 et supprime le sommet de la pile `P`



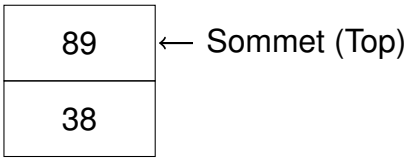
- `push(P,89)` ⇒ empile l'élément 89 dans la pile `P`



- `push(P,23)` ⇒ empile l'élément 23 dans la pile `P`



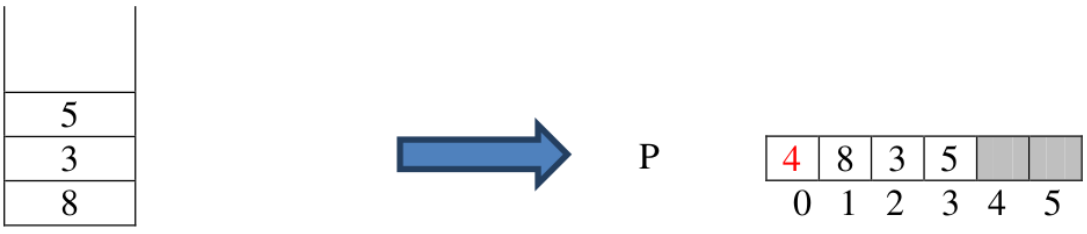
- `pop(P)` ⇒ renvoi 23 et supprime le sommet de la pile `P`



Représentation d'une pile à l'aide d'un tableau

On peut par exemple réaliser une pile capable de contenir `n` éléments avec un tableau `P[0 ... n]` pouvant contenir n + 1 éléments :

- La première case du tableau (d'indice 0) contient l'indice de la prochaine case vide (c'est l'indice qui correspondra au prochain élément à insérer dans la pile).
- Les cases suivantes du tableau (d'indice 1 à n), contiennent les éléments de la pile ou sont vides. La dernière case non vide du tableau est le sommet de la pile.



Exemple avec n = 5

Remarque

- Si `P[0]==1`, la pile est vide. A chaque fois qu'on insère un élément, on augmente `P[0]` d'une unité.
- Lorsque `P[0]==n + 1`, la pile est pleine

Voici une implémentation python de 2 fonctions `empiler` et `depiler`. La pile est représentée par un tableau python.

- Fonction `empiler(P,x)`

```
def empiler(P,x):
    if P[0] == len(P):
        print("La pile est pleine !")
        return False
    else:
        P[P[0]] = x
        P[0] = P[0] + 1
        return True
```

- Fonction `depiler(P)`

```
def depiler(P):
    if P[0] !=1: #la pile n'est pas vide
        P[0] = P[0]-1
        return P[P[0]]
    else:
        print("la pile est vide")
```

Exemple d'utilisation de cette implémentation

```
taille = 3
ma_pile= [0]*(taille+1) # +1 pour la premiere case qui contient l'indice du prochin
    element a empiler
ma_pile[0]=1 # indice du prochain element a empiler
print(ma_pile)        #OUTPUT : [1, 0, 0, 0]
empiler(ma_pile,39)
print(ma_pile)        #OUTPUT : [2, 39, 0, 0]
empiler(ma_pile,45)
print(ma_pile)        #OUTPUT : [3, 39, 45, 0]
empiler(ma_pile,78)
print(ma_pile)        #OUTPUT : [4, 39, 45, 78]
empiler(ma_pile,90)   #OUTPUT : La pile est pleine !
print(ma_pile)        #OUTPUT : [4, 39, 45, 78]
depiler(ma_pile)
print(ma_pile)        #OUTPUT : [3, 39, 45, 78]
depiler(ma_pile)
print(ma_pile)        #OUTPUT : [2, 39, 45, 78]
empiler(ma_pile,99)
print(ma_pile)        #OUTPUT : [3, 39, 99, 78]
```

III - Files

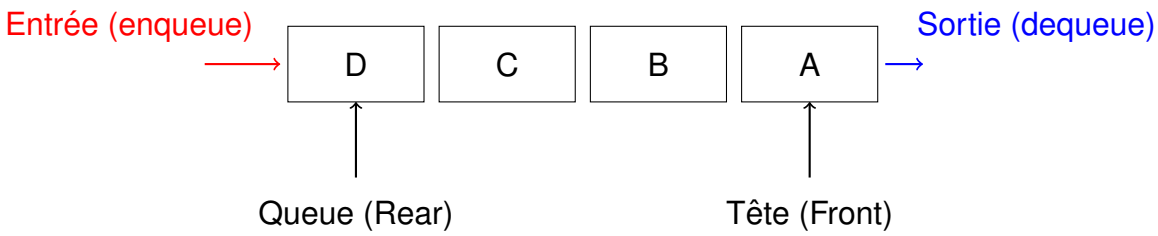
Définition 3

Une **file** est une structure de donnée linéaire dans laquelle :

- on ajoute des éléments à une extrémité de la file (en queue de la file)
- on supprime des éléments à l'autre extrémité (en tête de la file).

On prend souvent l'analogie de la file d'attente devant un magasin pour décrire une file de données.
Les files sont basées sur le principe **FIFO (First In First Out)** : le premier qui est rentré sera le premier à sortir.

Exemple : File `F`



Dans cet exemple :

- le premier élément qui a été entré dans la file `F` est l'élément `A`. Cet élément est également appelé la **tête** de la file `F`. Il sera le premier élément à sortir de la file `F`.
- le dernier élément qui a été entré dans la file `F` est l'élément `D`. Cet élément est également appelé la **queue** de la file `F`. Il sera le dernier élément à sortir de la file `F`.

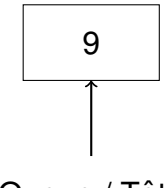
Exemple d'interface d'une implémentation du type `File`

Voici un exemple d'interface proposant les opérations qui peuvent être effectuées sur une file :

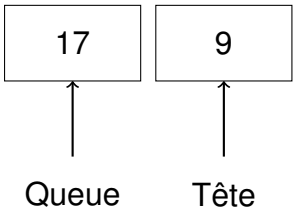
- `fileVide()` : retourne une file vide.
- `estVide(F)` : retourne vrai si la file `F` est vide, faux sinon.
- `enqueue(F,x)` : ajoutte l'élément `x` à la queue de la file `F`.
- `dequeue(F)` : retourne l'élément le plus ancien de la file `F` et le supprime de la file.
- `taille(F)` : retourne le nombre d'éléments présents dans la file `F`.

Cas pratique

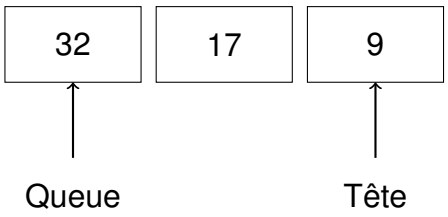
- `F = fileVide()` ⇒ on a créé une file vide
- `estVide(F)` ⇒ renvoie vrai
- `taille(F)` ⇒ renvoie 0
- `enqueue(F,9)` ⇒ ajoute l'élémént 9 dans la file `F`



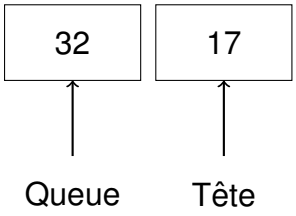
- `enqueue(F,17)` ⇒ ajoute l'élément 17 en queue de la file `F`



- `enqueue(F,32)` ⇒ ajoute l'élément 32 en queue de la file `F`



- `taille(F)` ⇒ renvoie 3
- `dequeue(F)` ⇒ renvoi 9 (élément en tête de la file `F`) et le supprime de la file.



Représentation d'une file à l'aide d'un tableau

On peut par exemple réaliser une file capable de contenir `n` éléments avec un tableau `F[0 ... n]` pouvant contenir n + 1 éléments :

- La première case du tableau (d'indice 0) contient le nombre d'élément de la file, que l'on note `nb_elt`.
- La case d'indice 1 contient la queue de la file.
- La case d'indice `nb_elt` contient la tête de la file.

Remarque

- Si `F[0]==0`, la file est vide. A chaque fois qu'on insère un élément, on augmente `F[0]` d'une unité.
- Lorsque `F[0]==n`, la file est pleine

Voici une implémentation python de 2 fonctions `enqueue` et `dequeue`. La file est représentée par un tableau python.

- Fonction `enqueue(P,x)`
  (a) Si la file n'est pas pleine (`nb_elt < n`, on décale tous les éléments d'indice compris entre 1 et `nb_elt` d'une unité d'indice vers la droite.
  (b) On affecte la valeur `x` à la case d'indice 1 du tableau.
  (c) On incrémente la case d'indice 0.

```
def enqueue(F,x):
    if F[0]<len(F)-1:
        for k in range(F[0],0,-1):
            F[k+1] = F[k]
        F[1]=x
        F[0]=F[0]+1
    else:
        print("la file est pleine !")
```

- Fonction `dequeue(P)`
  (a) Si la file n'est pas vide (`nb_elt < n`, on retourne l'élément d'indice `nb_elt`.
  (b) On décrémente la case d'indice 0.

```
def dequeue(F):
    if F[0]>0:
        x = F[F[0]]
        F[0] = F[0] - 1
        return x
    else:
        print("la file est vide")
```

Exemple d'utilisation de cette implémentation

```
taille = 3
ma_file= [0]*(taille+1) # +1 pour la premiere case qui contient le nombre d'element
    de la file
print(ma_file)           #OUTPUT : [0, 0, 0, 0]
enqueue(ma_file,39)
print(ma_file)           #OUTPUT : [1, 39, 0, 0]
enqueue(ma_file,16)
print(ma_file)           #OUTPUT : [2, 16, 39, 0]
enqueue(ma_file,87)
print(ma_file)           #OUTPUT : [3, 87, 16, 39]
enqueue(ma_file,55)      #OUTPUT : la file est pleine !
print(ma_file)           #OUTPUT : [3, 87, 16, 39]
x = dequeue(ma_file)
print(x)                 #OUTPUT : 39
print(ma_file)           #OUTPUT : [2, 87, 16, 39]
x = dequeue(ma_file)
print(x)                 #OUTPUT : 16
print(ma_file)           #OUTPUT : [1, 87, 16, 39]
x = dequeue(ma_file)
print(x)                 #OUTPUT : 87
print(ma_file)           #OUTPUT : [0, 87, 16, 39]
x = dequeue(ma_file)     #OUTPUT : la file est vide
print(x)                 #OUTPUT : None
print(ma_file)           #OUTPUT : [0, 87, 16, 39]
```