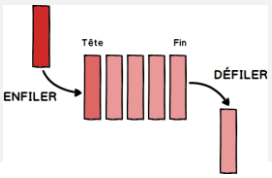
COURS DE NSI (Numérique et Sciences Informatiques)

| | |
|---|---|
| ***Classe : terminale*** | LYCÉE INTERNATIONAL COURS LUMIÈRE |
| ***séquence 2*** | **STRUCTURES DE DONNEES :** **LES LISTES, LES PILES ET LES FILES** |
| ***Mr BANANKO K.*** |   |

**Objectif :**

- Distinguer des structures par le jeu des méthodes qui les caractérisent.
- Choisir une structure de données adaptée à la situation à modéliser.
- Spécifier une structure de données par son interface. Distinguer interface et implémentation.
- Écrire plusieurs implémentations d'une même structure de données.

## 1- INTRODUCTION

De nombreux algorithmes "classiques" manipulent des structures de données plus complexes que des simples nombres (nous aurons l'occasion d'en voir plusieurs cette année). 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 et les files**. Ces trois types de structures sont qualifiés de **linéaires**.

## 2-Les listes

***Une liste*** est une structure de données 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, et
- Sa queue (souvent noté ***cdr***) qui correspond au reste de la liste.

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**"(*traitement de liste*)).

Voici 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)
- obtenir le dernier élément ajouté à la liste (car)
- obtenir une liste contenant tous les éléments d'une liste à l'exception du dernier élément ajouté (cdr)
- 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 (supprEnTete(L))
- Compter le nombre d'éléments présents dans une liste (compte(L) renvoie le nombre d'éléments présents dans la liste L)

La fonction *cons* permet d'obtenir une nouvelle liste à partir d'une liste et d'un élément (L1 = cons(x,L)). Il est possible "d'enchaîner" les *cons* et d'obtenir ce genre de structure : cons(x, cons(y, cons(z,L)))

Exemples :

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

- L=vide() => on a créé une liste vide
- estVide(L) => renvoie vrai
- ajouteEnTete(3,L) => La liste L contient maintenant l'élément 3
- estVide(L) => renvoie faux
- ajouteEnTete(5,L) => la tête de la liste L correspond à 5, la queue contient l'élément 3
- ajouteEnTete(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(8, cons(5, cons(3, L1))) => La tête de L2 correspond à 8 et la queue contient les éléments 3 et 5

**Activité 1**

Voici une série d'instructions (les instructions ci-dessous s'enchaînent), expliquez ce qui se passe à chacune des étapes :

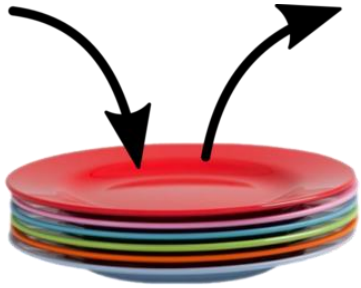
- L = vide()
- ajouteEnTete(10,L)
- ajouteEnTete(9,L)
- ajouteEnTete(7,L)
- L1 = vide()
- L2 = cons(5, cons(4, cons(3, cons (2, cons(1, cons(0,L1))))))

## 3-Les piles

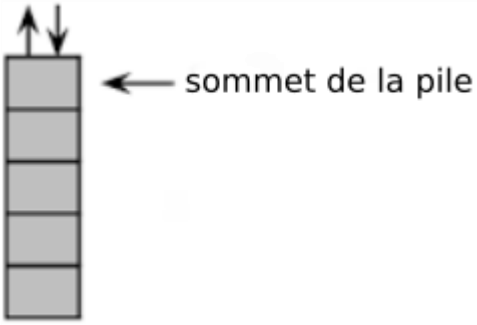
Une pile est un ensemble ordonné d'éléments qui se manipule comme une pile d'assiettes :

- on peut ajouter une assiette au sommet de la pile, c'est l'opération empiler
- on peut tester si une pile est vide, c'est l'opération pile_vide
- si la pile n'est pas vide, on peut retirer l'assiette du sommet de de la pile, c'est l'opération dépiler

Comme dans une liste un seul élément est accessible directement, le sommet de la pile.



On retrouve dans les piles une partie des propriétés vues sur les listes. Dans les piles, il est uniquement possible de manipuler le dernier élément introduit dans la pile.




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.

Voici les opérations que l'on peut réaliser sur une pile :

| Opération | Signature | Description |
|---|---|---|
| creer_pile | creer_pile() | Renvoie une pile vide |
| pile_vide | pile_vide(pile) | Renvoi un booléen indiquant si la pile est vide |
| empiler | empiler(pile, elt) ou push(pile,elt) | Ajoute elt au sommet de la pile |

| Opération | Signature | Description |
|---|---|---|
| depiler | depiler(pile) ou pop(pile) | Retire l'élément au sommet de la pile et le renvoie |
| Taille | taille(pile) | Renvoi le nombre d'éléments présents dans la pile |
| sommet | sommet(pile) | Renvoi l'élément situé au sommet de la pile sans le supprimer de la pile |

Exemples :

Soit une pile P composée des éléments suivants : 12, 14, 8, 7, 19 et 22 (le sommet de la pile est 22) **Pour chaque exemple ci-dessous on repart de la pile d'origine** :

- depiler(P) renvoie 22 et la pile P est maintenant composée des éléments suivants : 12, 14, 8, 7 et 19 (le sommet de la pile est 19)
- empiler(P,42) la pile P est maintenant composée des éléments suivants : 12, 14, 8, 7, 19, 22 et 42
- sommet(P) renvoie 22, la pile P n'est pas modifiée
- si on applique depiler(P) 6 fois de suite, pile_vide(P) renvoie vrai
- Après avoir appliqué depiler(P) une fois, taille(P) renvoie 5

**Activité 2**

Soit une pile P composée des éléments suivants : 15, 11, 32, 45 et 67 (le sommet de la pile est 67). Quel est l'effet de l'instruction depiler(P)

### Implémentation en Python

L'implémentation des piles en python se fait facilement à l'aide des méthodes `append()` et `pop()` du type *list* :

```
ma_pile.append(ma_valeur)   # permet d'empiler une valeur

ma_pile.pop()               # permet de dépiler une valeur

len(ma_pile)                # renvoie la longueur de ma_pile
```

## Exemple : applications des piles

La structure de Pile se retrouve dans de nombreuses situations où on doit gérer un ensemble d'éléments :

- à la fin d'un devoir un enseignant récupère une pile de copies et souvent il les corrige en commençant par le sommet de la pile.
- les couches géologiques forment évidemment une pile

Néanmoins si la pile est une structure naturelle pour stocker des objets car les opérations depiler et empiler sont peu coûteuses, ce n'est pas toujours une solution satisfaisante :

- dans un rayon de supermarché il n'est pas raisonnable de stocker des yaourts sous forme de pile, sinon les plus anciens risquent de n'être jamais achetés
- les personnes qui attendent depuis longtemps devant une salle de concerts avec placement libre ne souhaitent pas que les derniers arrivés soient les premiers entrés !

On retrouve aussi la structure de Pile dans de nombreuses situations en informatique :

- *L'historique* d'un navigateur Web conserve les pages parcourues dans une pile : la flèche gauche permet de revenir en arrière (dépiler) et la flèche droite d'avancer (empiler)
- Lors de l'évaluation d'une *fonction récursive* les contextes des appels récursifs imbriqués sont stockés dans une pile dont la taille est d'ailleurs limitée pour éviter les appels infinis. Lorsque la *pile d'appels* déborde, c'est le fameux *stack overflow* ! Pour éviter le dépassement de capacité de la *pile d'appels*, on peut exprimer une fonction récursive sous forme de boucle en utilisant une pile. Nous verrons une application dans l'algorithme d'exploration de graphe en profondeur.

### Exercice 1

Faire un tableau d'évolution des variables `stack` et `stack2` au cours de l'exécution du code ci-dessous. On considère qu'il s'agit de piles mutables.

```
stack = creer_pile()
empiler(stack, 8)
empiler(stack, 4)
empiler(stack, 3)
stack2 = creer_pile()
while not pile_vide(stack):
        empiler(stack2, depiler(stack))
```

### Exercice 2

On considère une fonction utilisant les opérations du type abstrait Pile :

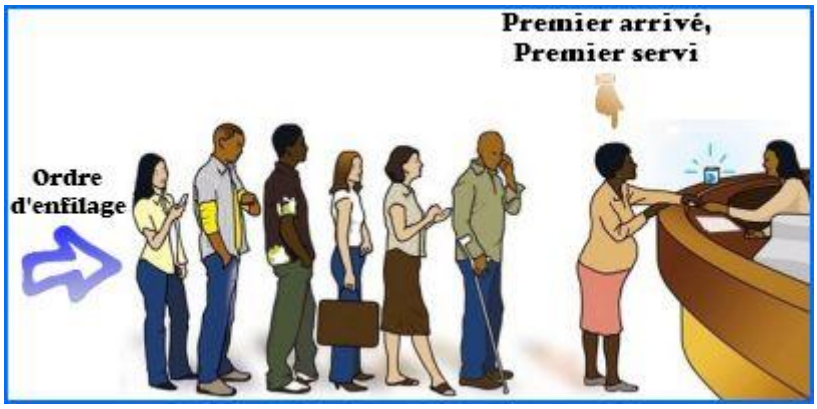
```
def mystere(pile):
    autre = creer_pile()
    k = 0
    while not pile_vide(pile):
        empiler(autre, depiler(pile))
        k = k + 1
    while not pile_vide(autre):
        empiler(pile, depiler(autre))
    return k
```

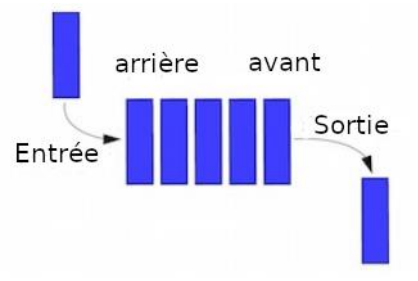
## 4- Les files

Comme les piles, les files ont des points communs avec les listes. Différences majeures : dans une file on ajoute des éléments à une extrémité de la file et on supprime des éléments à l'autre extrémité.

Une file est un ensemble ordonné d'éléments qui se manipule comme une file d'attente

On prend souvent l'analogie de la file d'attente devant un magasin pour décrire une file de données.




Voici les opérations que l'on peut réaliser sur une file :

| Opération | Signature | Description |
|---|---|---|
| creer_file | creer_file() | Renvoie une file vide |
| file_vide | file_vide(pile) | Renvoi un booléen indiquant si la file est vide |
| enfiler | enfiler(file, elt) | Ajoute elt à la fin de la file |

| Opération | Signature | Description |
|---|---|---|
| | taille(file) | Renvoi le nombre d'éléments présents dans la file |
| defiler | defiler(file) | Retire l'élément au début de la file et le renvoie |

On peut accéder à l'élément situé en bout de file sans le supprimer de la file (premier)

Quelques propriétés à retenir :

- Le premier élément qu'on peut retirer d'une file est forcément le premier à y être entré (le premier qui est rentré sera le premier à sortir.), c'est une structure **First In First Out (FIFO).**

| Acronyme anglais | Signification | Structure |
|---|---|---|
| **LIFO** | Dernier entré premier sorti | Pile |
| **FIFO** | Premier entré premier sorti | File |

- En particulier si on défile tous les éléments, l'ordre dans lequel on les retire de la file est le même que leur ordre d'insertion.
- La séquence d'éléments dans la **file** peut être représentée par une liste mais contrairement à une **pile** on a besoin d'accéder à deux éléments qui sont aux extrémités de la liste.
- Le type **File** nécessite un accès aux deux extrémités d'une liste donc une implémentation par une liste chaîné immuable avec des **tuples** n'est pas possible comme pour le type Pile. On proposera des implémentations de ***file mutable*** mais on verra une implémentation du type File avec deux piles, qui peut se décliner en file immuable si on utilise des piles immuables implémentées avec des `tuples` .

## Exemple d'utilisation

La structure de File se retrouve dans de nombreuses situations où on doit gérer un ensemble d'éléments :

- la file d'attente à un guichet de gare, à une caisse de supermarché, pour accéder à une formation sélective sur Parcoursup ...

- le stockage des yaourts dans un rayon de supermarché : le client défile le yaourt qui se trouve devant, le cariste *enfile* les nouveaux produits derrière.

On retrouve aussi la structure de File dans de nombreuses situations en informatique :

- les programmes en cours **d'exécution** ou ***processus*** accédent à tour de rôle à la ressource processeur qui n'exécute qu'un seul programme à la fois, ils sont placés dans une file de priorité : le modèle du tourniquet équivaut à celui d'une file d'attente : le processus qui achève son temps d'accès processeur vient se placer en fin de file et le processus en début de file accède à son tour au processeur.

- les travaux d'impressions lancés sur une imprimante en réseau sont placés dans une file d'impression lors du parcours d'un graphe en largeur, on maintient une file d'attente des prochains sommets à visiter.

Exemples :

Soit une file F composée des éléments suivants : 12, 14, 8, 7, 19 et 22 (le premier élément rentré dans la file est 22 ; le dernier élément rentré dans la file est 12). Pour chaque exemple ci-dessous on repart de la file d'origine :

- enfiler(F,42) la file F est maintenant composée des éléments suivants : 42, 12, 14, 8, 7, 19 et 22 (le premier élément rentré dans la file est 22 ; le dernier élément rentré dans la file est 42)
- défiler(F) la file F est maintenant composée des éléments suivants : 12, 14, 8, 7, et 19 (le premier élément rentré dans la file est 19 ; le dernier élément rentré dans la file est 12)
- premier(F) renvoie 22, la file F n'est pas modifiée
- si on applique défiler(F) 6 fois de suite, file_vide(F) renvoie vrai
- Après avoir appliqué  défiler(F) une fois, taille(F) renvoie 5.

**Activité 3**

Soit une file F composée des éléments suivants : 1, 12, 24, 17, 21 et 72 (le premier élément rentré dans la file est 72 ; le dernier élément rentré dans la file est 1). Quel est l'effet de l'instruction enfiler(F,25)

**Implémentation en Python**

L'implémentation des files en python se fait de manière analogue aux piles :

- ```
  ma_file.append(ma_valeur)   # permet d'emfiler une valeur
  ```
  ```

```
ma_file.pop(0)              # permet de défiler une valeur

len(ma_file)                # renvoie la longueur de ma_file
```

Exercice 1

Faire un tableau d'évolution des variables queue et queue2 au cours de l'exécution du code ci-dessous. On considère qu'il s'agit de files mutables.

```
queue = creer_file()
enfiler(queue 8)
enfiler(queue, 4)
enfiler(queue, 3)
queu2 = creer_file()
while not file_vide(queue):
    enfiler(queue2, defiler(queue))
```

Exercice 2

Écrire une fonction `fusion(file1, file2)` permettant de réaliser la fusion triée de deux files `file1` et `file2` supposées triées avec le maximum en fin de file. On ne demande pas de maintenir les files `file1` et `file2`.

## 5- Types abstraits et représentation concrète des données

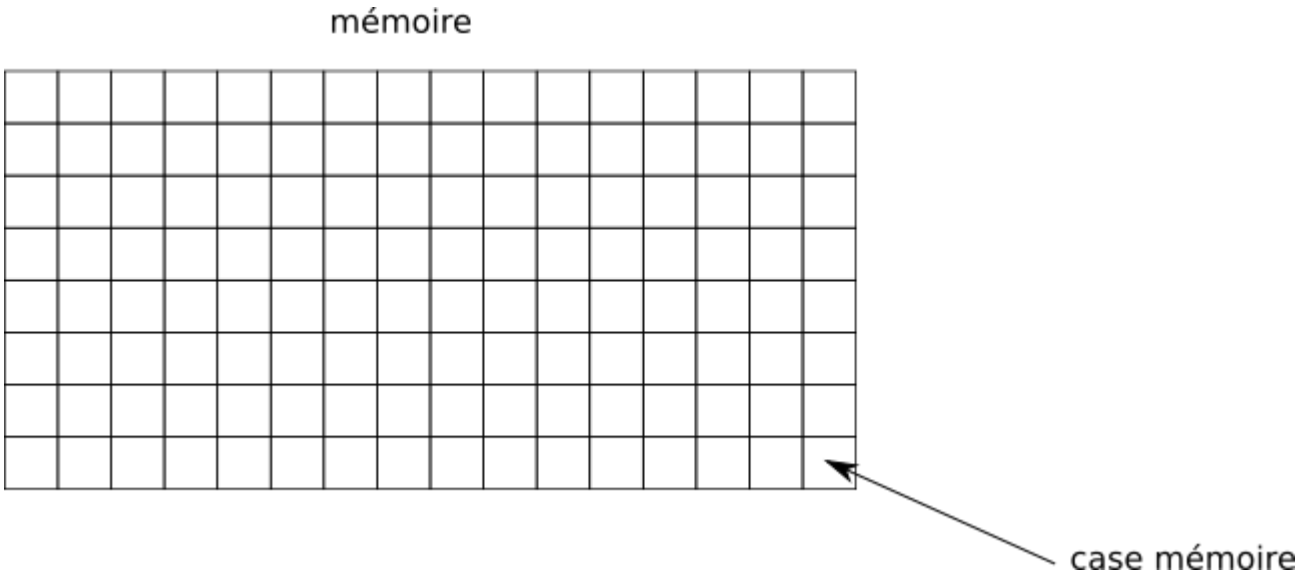
Nous avons évoqué ci-dessus la manipulation des types de données (liste, pile et file) par des algorithmes, mais, au-delà de la beauté intellectuelle de réfléchir sur ces algorithmes, le but de l'opération est souvent, à un moment ou un autre, de "traduire" ces algorithmes dans un langage compréhensible pour un ordinateur (Python, Java, C,...). On dit alors que l'on implémente un algorithme. Il est donc aussi nécessaire d'implémenter les types de données comme les listes, les piles ou les files afin qu'ils soient utilisables par les ordinateurs. Les listes, les piles ou les files sont des "vues de l'esprit" présent uniquement dans la tête des informaticiens, on dit que ce sont des types abstraits de données (ou plus simplement des types abstraits). L'implémentation de ces types abstrait, afin qu'ils soient utilisables par une machine, est loin d'être une chose triviale. L'implémentation d'un type de données dépend du langage de programmation. Il faut, quel que soit le langage utilisé, que le programmeur retrouve les fonctions qui ont été définies pour le type abstrait (pour les listes, les piles et les files cela correspond aux fonctions définies ci-dessus). Certains types abstraits ne sont pas forcément implémentés dans un langage donné, si le programmeur veut utiliser ce type abstrait, il faudra qu'il le programme par lui-même en utilisant les "outils" fournis par son langage de programmation.

Pour implémenter les listes (ou les piles et les files), beaucoup de langages de programmation utilisent 2 structures : **les tableaux et les listes chaînées.**

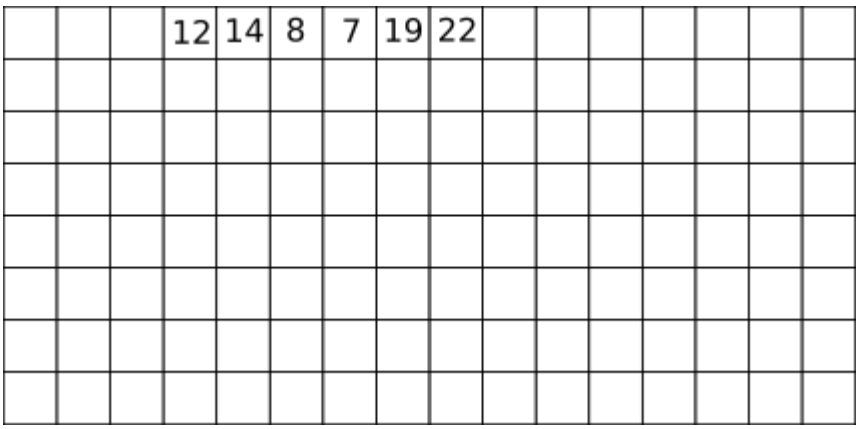
## Tableaux (ou Listes) :

- Les tableaux sont des structures de données statiques de taille fixe. Une fois que la taille d'un tableau est définie, elle ne peut généralement pas être modifiée.
- Les éléments d'un tableau sont stockés de manière contiguë en mémoire, ce qui permet un accès rapide aux éléments en utilisant des indices.
- Les tableaux sont adaptés lorsque vous connaissez la taille maximale de la structure de données à l'avance, et que vous souhaitez un accès rapide aux éléments.
- Cependant, les tableaux peuvent entraîner un gaspillage d'espace si leur taille maximale est beaucoup plus grande que la quantité réelle de données stockées.

Un tableau est une suite contiguë de cases mémoires (les adresses des cases mémoire se suivent) :




Le système réserve une plage d'adresse mémoire afin de stocker des éléments.




La taille d'un tableau est fixe : une fois que l'on a défini le nombre d'éléments que le tableau peut accueillir, il n'est pas possible modifier sa taille. Si l'on veut insérer une donnée, on doit créer un nouveau tableau plus grand et déplacer les éléments du premier tableau vers le second tout en ajoutant la donnée au bon endroit !

Dans certains langages de programmation, on trouve une version "évoluée" des tableaux : les tableaux dynamiques. Les tableaux dynamiques ont une taille qui peut varier. Il est donc relativement simple d'insérer des éléments dans le tableau. Ce type de tableaux permet d'implémenter facilement le type abstrait liste (de même pour les piles et les files)

À noter que les "listes Python" (listes Python) sont des tableaux dynamiques. Attention de ne pas confondre avec le type abstrait liste défini ci-dessus, ce sont de "faux amis".

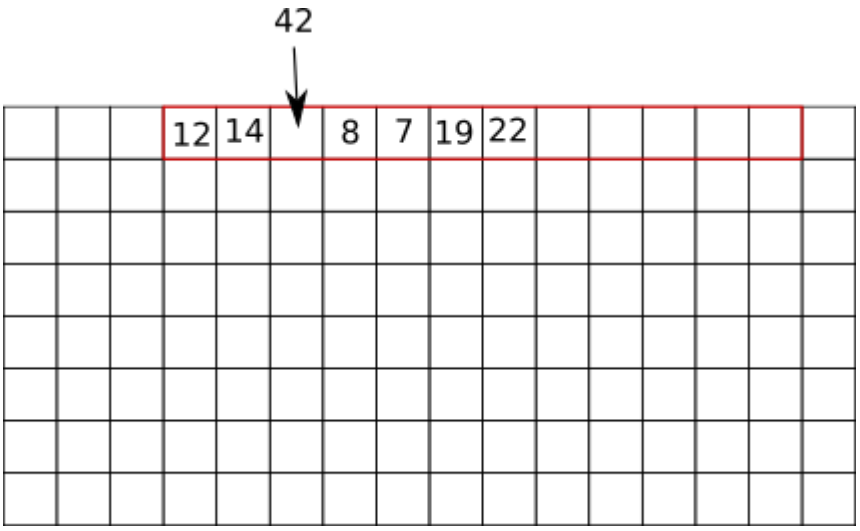


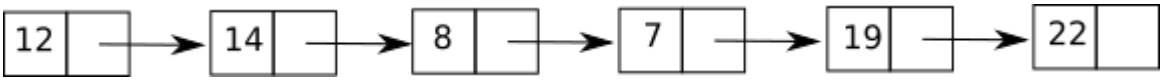

tableau dynamique

Autre type de structure que l'on rencontre souvent et qui permet d'implémenter les listes, les piles et les files : les listes chaînées.

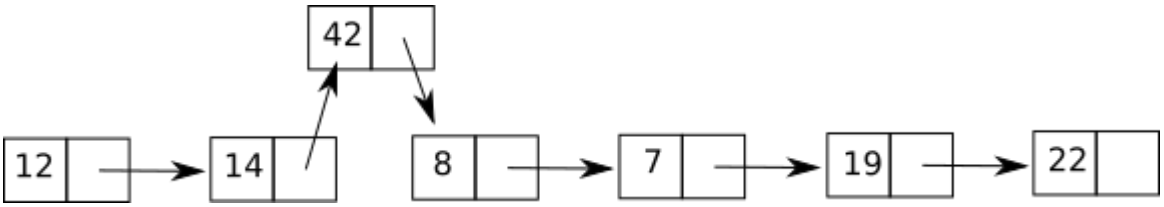
### Listes Chaînées :

- Les listes chaînées sont des structures de données dynamiques qui peuvent grandir ou rétrécir au besoin. Chaque élément est lié au suivant (et parfois au précédent) au lieu d'être stocké en mémoire de manière contiguë.
- Les listes chaînées sont adaptées lorsque la taille de la structure de données est inconnue à l'avance, car elles peuvent être allongées ou raccourcies dynamiquement.
- Les listes chaînées peuvent être plus efficaces en termes d'utilisation de la mémoire, car elles n'ont besoin que de l'espace nécessaire pour stocker les éléments réels.
- Cependant, l'accès aux éléments d'une liste chaînée peut être plus lent que l'accès aux éléments d'un tableau, car vous devez parcourir la liste de manière séquentielle.

Dans une liste chaînée, à chaque élément de la liste on associe 2 cases mémoire : la première case contient l'élément et la deuxième contient l'adresse mémoire de l'élément suivant.




Il est relativement facile d'insérer un élément dans une liste chaînée :




Il est aussi possible d'implémenter les types abstraits en utilisant des structures plus complexes que les tableaux et les listes chaînées. Par exemple, en Python, il est possible d'utiliser les tuples pour implémenter le type abstrait liste :

**Activité 4**

Étudiez attentivement les fonctions suivantes :

```
def vide():
    return None
def cons(x,L):
    return(x,L)
def ajouteEnTete(L,x):
    return cons(x,L)
def supprEnTete(L):
    return (L[0],L[1])
def estVide(L):
    return L is None
def compte(L):
    if estVide(L):
        return 0
    return 1 + compte(L[1])
```

## Correction

Les fonctions que vous avez fournies semblent être liées à la manipulation de listes chaînées. Elles semblent être mises en œuvre en utilisant une structure de liste chaînée cons-cell ou paire (pair en anglais), où chaque élément de la liste est une paire (x, L) composée d'un élément et d'une référence à la liste suivante. Voici une description de chaque fonction :

1. **vide()**: Cette fonction renvoie une liste chaînée vide, représentée par **None**.

2. **cons(x, L)**: Cette fonction crée une nouvelle liste chaînée en ajoutant un élément **x** en tête de la liste **L**. Elle renvoie une nouvelle paire (x, L).

3. **ajouteEnTete(L, x)**: Cette fonction est une version alternative de **cons(x, L)**, elle ajoute un élément **x** en tête de la liste **L**.

4. **supprEnTete(L)**: Cette fonction prend en entrée une liste chaînée **L** et renvoie un tuple contenant le premier élément de la liste (**L[0]**) et le reste de la liste (**L[1]**). Elle supprime donc la tête de la liste.

5. **estVide(L)**: Cette fonction vérifie si la liste chaînée **L** est vide en comparant si elle est égale à **None**.

6. **compte(L)**: Cette fonction récursive compte le nombre d'éléments dans la liste chaînée **L**. Si la liste est vide, elle renvoie 0. Sinon, elle renvoie 1 plus le résultat de l'appel récursif de **compte** sur la liste suivante (**L[1]**).

Ces fonctions sont conçues pour effectuer des opérations de base sur les listes chaînées en utilisant la structure de paire (pair) pour représenter les éléments de la liste et leurs liens. Vous pouvez les utiliser pour créer, modifier et interroger des listes chaînées.

**Activité 5**

Après avoir saisi et exécuté le programme étudié au "À faire vous-même 4", tapez successivement les commandes suivantes dans une console Python :

- L = vide()
- estVide(L)
- L = cons(5, cons(4, cons(3, cons(2, cons(1, cons(0,L))))))
- estVide(L)
- compte(L)
- L = ajouteEnTete(L,6)
- compte(L)
- x, L=supprEnTete(L)
- x
- compte(L)
- x, L=supprEnTete(L)
- x
- compte(L)

**A retenir**

## Implémentation de Piles (Stacks) :

Utilisation de listes :

En Python, vous pouvez utiliser des listes pour implémenter une pile en utilisant les méthodes **append** pour empiler et **pop** pour dépiler. Voici un exemple :

```python
# Création d'une pile vide
stack = []

# Empiler des éléments
stack.append(1)
stack.append(2)
stack.append(3)

# Dépiler un élément
top_element = stack.pop()  # top_element contient 3
```

Utilisation de la bibliothèque **collections.deque** :

La bibliothèque standard Python propose **collections.deque** pour implémenter des piles de manière plus efficace en termes de performances pour les opérations d'empilement et de dépilement :

```python
from collections import deque

# Création d'une pile vide
stack = deque()

# Empiler des éléments
stack.append(1)
stack.append(2)
stack.append(3)

# Dépiler un élément
top_element = stack.pop()  # top_element contient 3
```

## Implémentation de Files (Queues) :

Utilisation de la bibliothèque **queue** :

La bibliothèque standard Python propose le module **queue** pour implémenter des files. Vous pouvez utiliser la classe **Queue** pour créer une file FIFO (premier entré, premier sorti) ou la classe **LifoQueue** pour créer une pile (dernier entré, premier sorti). Voici un exemple d'utilisation de la classe **Queue** :

```python
from queue import Queue

# Création d'une file vide
q = Queue()

# Enfiler des éléments
q.put(1)
q.put(2)
q.put(3)

# Défiler un élément
top_element = q.get()  # top_element contient 1
```

Utilisation de la bibliothèque **collections.deque** :

Vous pouvez également utiliser **collections.deque** pour implémenter une file. Pour une file FIFO, vous pouvez ajouter des éléments à l'une des extrémités et les retirer de l'autre extrémité :

```python
from collections import deque

# Création d'une file vide
queue = deque()

# Enfiler des éléments
queue.append(1)
queue.append(2)
queue.append(3)

# Défiler un élément
front_element = queue.popleft()  # front_element contient 1
```

Ces exemples illustrent différentes façons d'implémenter des piles et des files en Python. Le choix dépendra de vos besoins spécifiques et des performances souhaitées pour vos opérations d'empilement (push) et de dépilement (pop) pour les piles, ou d'enfilage (enqueue) et de défilage (dequeue) pour les files.

## Implémentation d'une Pile (Stack) avec `__init__` :

**`__init__(self, l, tete)`** est le constructeur d'une classe en Python. Le double souligné avant et après "init" indique qu'il s'agit d'une méthode spéciale ou du constructeur par défaut d'une classe.

```python
class Pile:
    def __init__(self):
        self.elements = []

    def empiler(self, element):
        self.elements.append(element)

    def depiler(self):
        if not self.est_vide():
            return self.elements.pop()

    def est_vide(self):
        return len(self.elements) == 0

# Créez une pile et effectuez des opérations
ma_pile = Pile()
ma_pile.empiler(1)
ma_pile.empiler(2)
ma_pile.empiler(3)
sommet = ma_pile.depiler()
print(sommet)  # Résultat : 3
```

## Implémentation d'une File (Queue) avec `__init__` :

```python
class File:
    def __init__(self):
        self.elements = []

    def enfiler(self, element):
        self.elements.append(element)

    def defiler(self):
        if not self.est_vide():
            return self.elements.pop(0)

    def est_vide(self):
        return len(self.elements) == 0

# Créez une file et effectuez des opérations
ma_file = File()
ma_file.enfiler(1)
ma_file.enfiler(2)
ma_file.enfiler(3)
premier_element = ma_file.defiler()
print(premier_element)  # Résultat : 1
```

Dans ces exemples, les classes **Pile** et **File** sont créées avec une méthode **__init__** qui initialise la structure de données respective (une liste vide) lorsque vous créez une instance de la classe. Ensuite, vous pouvez utiliser les méthodes **empiler** (push), **depiler** (pop), **enfiler** (enqueue), et **defiler** (dequeue) pour effectuer les opérations sur la pile ou la file, en fonction de la structure que vous avez choisie.

**Activité 6**

Étudiez l'implémentation des piles et des files en Python en vous aidant de la documentation officielle (5.1.1 et 5.1.2).

Exercice 1

Soit le programme Python suivant :

```
from collections import deque
file = deque([])
tab = [2,78,6,89,3,17]
file.append(5)
file.append(10)
file.append(8)
file.append(15)
for i in tab:
    if i > 50:
        file.popleft()
```

Donnez l'état de la file *file* après l'exécution de ce programme

Exercice 2

- Sujet2_Bordonaro-versionFinale.odt Sujet 1 2022 Exercice 4
- Sujet 1 2021 Exercice 1
- Sujet 3 2021 Exercice 5
- Sujet 6 2021 Exercice 5
- Sujet 7 2021 Exercice 1
- Sujet 8 2021 Exercice 5
-