LYCÉE INTERNATIONAL COURS LUMIÈRE

Cours d'informatique

Prépas 2

Partie 1

S3

I- Introduction aux Types de Données Abstraites (TDA) - Listes et Piles

Concept clé : Un type de donnée abstrait est une structure de donnée définie par son comportement (opérations permises) plutôt que par son implémentation. L'implémentation réelle est cachée à l'utilisateur.

Exemple : Une **pile** est un ADT. Peu importe si elle est implémentée avec un tableau ou une liste chaînée, ce qui compte, c'est que les opérations de base (push, pop, peek) sont disponibles et fonctionnent comme prévu.

Objectifs du cours :

- Comprendre les types de données abstraites et leur importance dans la programmation.
- Maîtriser les structures de données courantes (listes, piles) et savoir quand les utiliser.
- Implémenter des algorithmes sur ces structures en Python.

1. Définition des Types de Données Abstraites (ADT : Abstract Data Types)

Un **type de données abstrait (TDA)** est une manière de structurer des données afin de permettre une manipulation efficace. Contrairement aux types de données concrets, les TDA définissent les opérations qu'on peut effectuer sur eux sans préciser l'implémentation.

2. Exemples courants :

- **Liste** : Une collection ordonnée d'éléments, où chaque élément a une position spécifique.
- **Pile (Stack)** : Une collection d'éléments où le dernier ajouté est le premier retiré (LIFO - Last In, First Out).
- **File (Queue)** : Les éléments sont retirés dans l'ordre où ils ont été ajoutés (FIFO - First In, First Out).
- **Dictionnaire (Table de hachage)** : Une structure qui associe des clés à des valeurs.

3. Importance des TDA :

Les TDA permettent de choisir la structure de données optimale en fonction du problème à résoudre, ce qui peut significativement améliorer la performance des programmes.

A. Les Listes et tableaux

1. Définition :

Une **liste** est une séquence ordonnée d'éléments où chaque élément est accessible par son index. Les listes permettent :

- Ajout de nouveaux éléments.
- Suppression d'éléments.
- Accès aux éléments via leur index.

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*)).

2. Opérations courantes sur les listes :

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) :append(), insert()
- supprimer la tête x d'une liste L et renvoyer cette tête x (supprEnTete(L)) ; remove(), pop()
- Accès à un élément : Utilisation de l'index [i]
- 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

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. Programmation en python:

```
Notation d'une liste

[1,2,3,4]                    # énumération des objets entre []

[[1,2],[1],1]                # liste dont les deux premiers termes sont des listes

[]                           # liste vide
```

L'accès à un élément d'une liste est direct (contrairement à de nombreux langages, ou l'accès se fait par chaînage, en suivant les pointeurs à partir du premier élément). L'indexation des éléments commence à 0 :

```
Accès aux éléments d'une liste par indexation positive:

>>> li = [1,2,3]
>>> li[0]
1
>>> li[2]
3
>>> li[3]

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

```
IndexError: list index out of range
```

On peut aussi accéder aux éléments par indexation négative. Le dernier élément de la liste est alors numéroté -1. Avec la liste de l'exemple précédent, on obtient par exemple :

```
>>> li[-1]
3
>>> li[-3]
1
>>> li[-4]

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Cette possibilité est surtout intéressante pour accéder aux derniers éléments d'une liste.

Voici une liste des opérations et méthodes les plus utilisées sur les listes. Pour une liste plus exhaustive, utilisez help(list). On indique par un (m) le fait que la méthode modifie directement la liste sur laquelle elle agit. Les autres retourne le résultat en sortie de fonction.

```
Opérations et méthodes applicables à une liste:
len(L)                  # longueur (nombre d'éléments) de L
L1 + L2                 # concaténation des listes
n * L                   # pour n entier: concaténation répétée de L avec elle-même.
L.append(a)             # ajout de l'objet a en fin de liste (m)
L.insert(i,a)           # insertion de l'objet a en position i (m)
L.remove(a)             # retrait de la première occurrence de a (m)
L.pop(i)                # retrait et renvoi de l'élément d'indice i (m)
                       # par défaut, si i non précisé: dernier élément
L.index(a)              # position de la première occurrence de a
                       # ValueError si a n'est par dans la liste
L.count(a)             # nombre d'occurrences de a dans la liste
a in L                 # teste l'appartenance de a à L
L.copy()               # copie simple de L
                       # attention aux problèmes de dépendance des attributs
L.reverse()            # retourne la liste (inversion des indexations) (m)
L.sort()               # trie la liste dans l'ordre croissant (m)
                      # (si composée d'objets comparables)
                       # voir ci-dessous pour des paramètres
```

Le tri d'une liste possède deux paramètres : un qui permet de préciser si le tri est croissant ou décroissant, l'autre qui permet de définir une clé de tri (une fonction : on trie alors la liste suivant les valeurs que la fonction prend sur les éléments de la liste)

```
Paramètres de tri:

li.sort()                          # tri simple

li.sort(reverse=True)              # tri inversé
```

```
li.sort(key = f)                    # tri suivant la clé donnée par la fonction f
```

La fonction donnant la clé de tri peut être une fonction prédéfinie dans un module (par exemple la fonction sin du module math), une fonction définie précédemment par le programmeur (par def), ou une fonction définie sur place (par lambda). Voir plus loin pour plus d'explications à ce propos.

Slicing (tranchage)

On peut créer une nouvelle liste en extrayant certains éléments d'une liste. Pour extraire les éléments d'indice entre d inclus et $j \geq i$ exclu, on utilise `L[i:j]` : la liste obtenue est donc composée des éléments `L[i], L[i+1],..., L[j-1]`. L'un ou l'autre de ces deux indices peut être omis, et même les deux (dans ce cas, i vaut 0 et j vaut la longueur de la liste). Ce mécanisme est tolérant envers les indices trop grands ou trop petits (attention, les indices négatifs entre -1 et $-n$ sont interprétés comme précédemment), et si $i \geq p$, on obtient la liste vide.

```
Technique de slicing:
L[i:j]              # Extraction de la tranche [L[i], ... , L[j-1]]
L[i:j:p]            # De même de p en p à partir de L[i], tant que i+k*p < j
```

À noter que :

- Si le premier indice est omis, il est pris égal à 0 par défaut.
- Si le deuxième indice est omis, il est pris égal à la longueur de la liste par défaut (on extrait la tranche finale)
- Si le troisième indice est omis, il est pris égal à 1 par défaut (cas de la première instruction ci-dessus)
- Un pas négatif permet d'inverser l'ordre des termes
- Le slicing est possible aussi avec des indexations négatives.

Par exemple :

```
>>> M = [0,1,2,3,4,5,6,7,8,9,10]
>>> M[3:6]
[3, 4, 5]
>>> M[2:8:2]
[2, 4, 6]
>>> M[:3]
[0, 1, 2]
>>> M[3::3]
[3, 6, 9]
>>> M[::5]
[0, 5, 10]
>>> M[:2:4]
[0]
>>> M[:5:4]
[0, 4]
>>> M[-3:-1]
[8, 9]
>>> M[2:6:-3]
[]
```

```
>>> M[6:2:-3]
[6, 3]
```

On distingue essentiellement deux types d'objets :

- les tableaux dont la taille est fixe
- les listes (dynamiques) dont le nombre d'objets est variable

De plus, les éléments peuvent être homogènes (tous de même type) ou hétérogènes (types différents et donc chaque élément a une taille mémoire différente).

3. **Tableaux** :

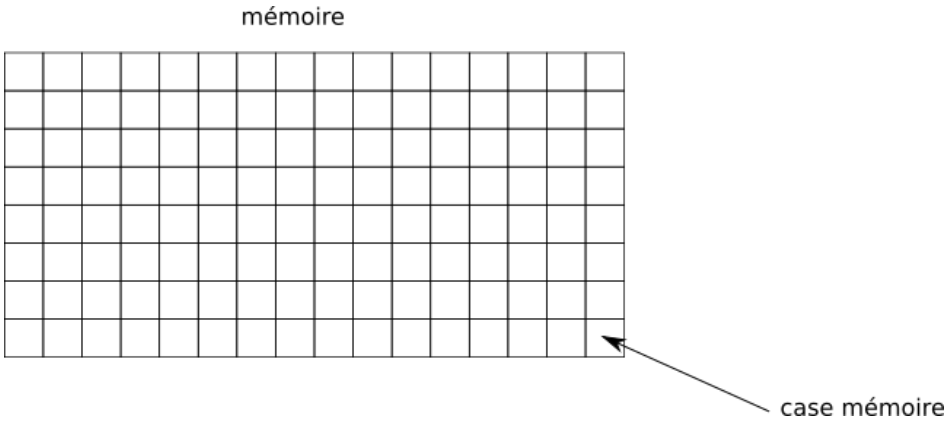
- Les tableaux sont des structures de données statiques de taille fixe. Une fois que la taille d'un tableau est définie, elle ne peut généralement pas être modifiée.

- Les éléments d'un tableau sont stockés de manière contiguë en mémoire, ce qui permet un accès rapide aux éléments en utilisant des indices.

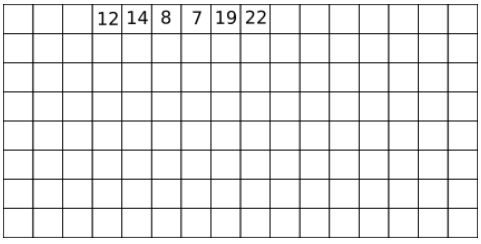
- Les tableaux sont adaptés lorsque vous connaissez la taille maximale de la structure de données à l'avance, et que vous souhaitez un accès rapide aux éléments.

- 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".

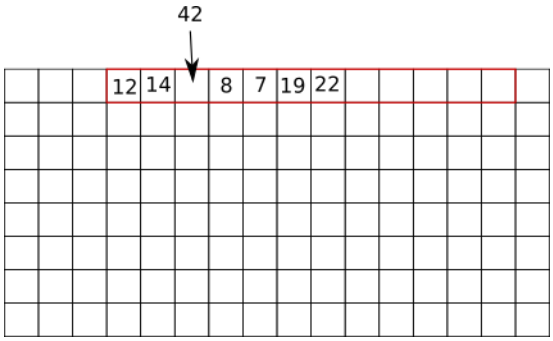


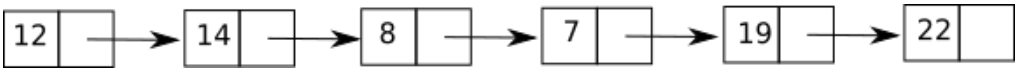
tableau dynamique

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

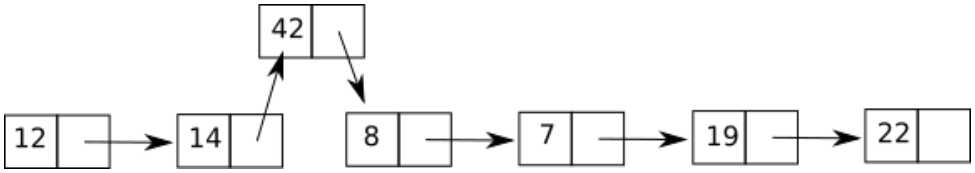
4. Listes Chaînées :

- 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 :



5. En Python

Le type list en Python est dynamique. L'opération essentielle est d'accéder rapidement aux éléments, ce qui est facile pour les tableaux homogènes. Le choix est le suivant :

- on utilise un tableau d'adresses (ainsi les tailles sont fixes) d'une certaine taille. Un élément est alors une adresse vers l'objet correspondant. Cela permet d'accéder en lecture et écriture à un élément quelconque rapidement
- lorsqu'on veut ajouter un élément en fin de liste : soit il reste des cases vides dans le tableau d'adresses, dans ce cas pas de difficulté; soit tout le tableau est rempli et dans ce cas on alloue une zone mémoire plus grande, on recopie le tableau dans cette nouvelle zone et on ajoute l'élément (si on ajoute une fois, il y a des chances qu'on ajoute plusieurs fois. Plutôt que d'ajouter une seule case au tableau, on augmente sa taille suffisamment pour des insertions futures - ni trop pour ne pas gaspiller de mémoire, ni trop peu pour ne pas avoir à le refaire souvent). La complexité de l'insertion est alors presque constante (on dit en temps amorti constant).
- Supprimer le dernier élément est facile, supprimer un élément intermédiaire beaucoup plus long car il faut décaler toutes les cases. . .

Pour concaténer deux listes, on en crée une suffisant longue et on copie les adresses (complexité de l'ordre de la taille totale).

B. Les piles

Définition : *Une classe abstraite est un type, muni d'opérations.*

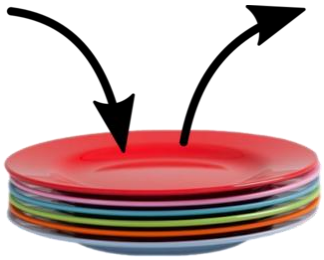
Cette définition est un peu vague, mais elle a le mérite de fixer les idées : pour définir une pile, on a essentiellement besoin de définir les opérations que l'on veut effectuer. L'implémentation effective d'une pile est indépendante de sa définition en tant que classe abstraite.

Le type pile est une structure de données abstraite, munie des opérations suivantes :

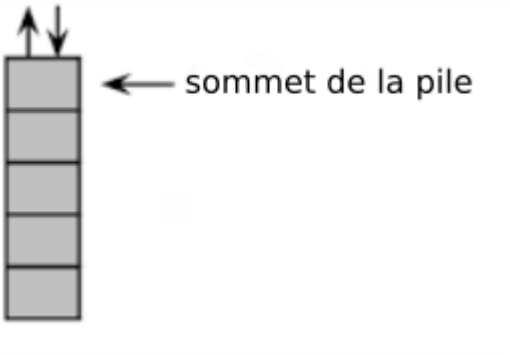
- créer une pile vide,
- savoir si la pile est vide,
- ajouter un objet en sommet de pile,
- récupérer et supprimer l'objet en haut de la pile

Les exemples usuels : une pile d'assiettes, l'historique d'un navigateur. . . La structure de données de liste simplement chaînée se prête parfaitement à la réalisation d'une pile.

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.

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
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)`

En Python basique

On commence par une présentation utilisant les listes Python (l'avantage : c'est simplet et intégré à Python, l'inconvénient : Ce n'est pas la structure optimale adaptée pour gérer les piles)

- on ajoute un élément avec la méthode append : `L.append(x)`
- on dépile le dernier élément avec pop : `x = L.pop()` (retire le sommet de la pile et le renvoie),
- on peut tester si la pile est vide : `L == []` (ou `len(L)==0`)
- éventuellement (ça dépend), on peut lire le sommet de la pile sans le dépiler (mais cela peut se faire avec un dépilement/empilement)
- on interdit tout le reste (retirer un élément quelconque, demander la taille de la liste, modifier un élément en position quelconque)
- len(ma_pile) *# renvoie la longueur de ma_pile*

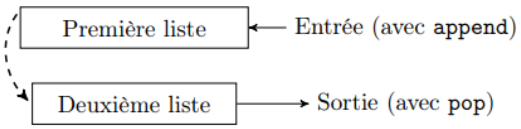
On pourra éventuellement créer des fonctions correspondantes pour des raisons de clarté :

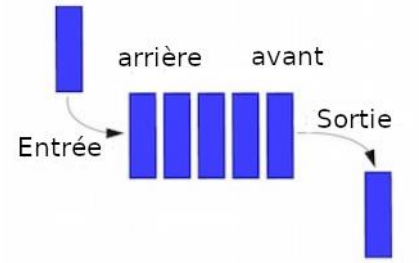
```
def pile_vide () :
    return []
def est_vide ( L ) :
    return ( len ( L ) ==0)
def empiler (L , x ) :
    L . append ( x )
def depiler ( L ) :
    return L . pop ()
```

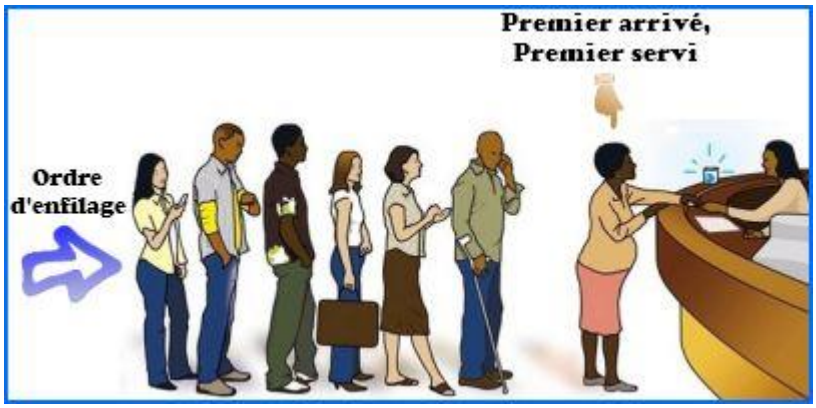
II- Structure de file

Dans cette section, on aborde une autre classe abstraite que celle de pile. La pile fonctionne sur le principe LIFO (last-in, first out), la file fonctionne suivant le principe FIFO (first-in, first-out). Il y a plusieurs réalisations possibles, on en donne une qui utilise deux listes. L'ajout d'un élément à la file se fait uniquement avec append dans la première liste, la suppression se fait avec pop dans la deuxième liste.

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







Lorsque la deuxième liste est vide et qu'on veut défiler (sortir un élément de la file), il est nécessaire de transférer les éléments de la première liste dans la deuxième. Pour conserver l'ordre dans la file, il est nécessaire d'inverser l'ordre des éléments lors du transfert. Voici les opérations de file à écrire :

- creer_file() : construire une file vide.
- file_vide(F) : renvoie un booléen suivant si la file F est vide ou non.
- defiler(F) : défile F en sortant l'élément en tête de file, et le renvoie. Si la file est vide, renvoie une erreur.
- enfiler(F,x) : ajoute l'élément x à la queue de la file F.

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

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)

Cette fois, la structure de données de liste doublement chaînée se prête à la réalisation d'une file : on a besoin de réaliser des opérations à chaque extrémité.

```
def file_vide () :
     return []
def est_vide ( F ) :
     return ( len ( F ) ==0)
def enfiler (F , x ) :
     F . append ( x )
def defiler ( F ) :
      return F . pop ( 0 )                    # ça c'est terrible !!!
```

```
def creer_file():
     return [],[] #couple de listes

def file_vide(F):
     return F[0]==[] and F[1]==[]

def enfiler(F,x):
    F[0].append(x)

def defiler(F):
    assert not file_vide(F), "file vide"
    if F[1]==[]:
         F[1][:]=F[0][::-1] # contenu de la deuxième liste remplacé par celui
de la première, à l'envers.
         F[0][:]=[]         # première liste vidée.
```

```
    return F[1].pop()
```

Bien sûr, une classe serait tout indiquée. En terme de complexité, on peut montrer que toutes les opérations se font en temps constant *amorti*.

III- Algorithmes courants sur les listes

Introduction

Les algorithmes de recherche et de tri sont essentiels lorsqu'on travaille avec des structures de données comme les listes. Ils permettent de localiser des éléments spécifiques (recherche) ou de réorganiser les éléments (tri) pour faciliter d'autres opérations. Ce cours couvre deux types d'algorithmes sur les listes : la **recherche séquentielle** et le **tri par sélection**.

A. Recherche séquentielle

1. Définition

La **recherche séquentielle** est un algorithme simple utilisé pour rechercher un élément dans une liste. Il parcourt chaque élément de la liste, un par un, jusqu'à trouver l'élément recherché ou atteindre la fin de la liste.

2. Principe

- On commence par le premier élément de la liste.
- On compare cet élément à la valeur recherchée.
- Si une correspondance est trouvée, l'algorithme renvoie l'indice de l'élément.
- Si aucun élément ne correspond à la valeur recherchée, l'algorithme renvoie une indication que l'élément est absent (généralement -1 ou None).

3. Implémentation en Python

```
def recherche_sequentielle(liste, valeur):
    for i in range(len(liste)):
        if liste[i] == valeur:
            return i  # Retourne l'indice de l'élément trouvé
    return -1  # Si l'élément n'est pas trouvé
```

4. Complexité

a) Meilleur cas : O(1)

Le meilleur cas survient lorsque l'élément recherché est le **premier élément** de la liste. Voici pourquoi la complexité est O(1) dans ce cas :

- L'algorithme n'a besoin de comparer qu'un seul élément (le premier de la liste) pour trouver l'élément recherché.
- Dès que la condition liste[i] == valeur est vérifiée pour i = 0, l'algorithme retourne immédiatement l'indice et s'arrête.

Exemple :

Si on recherche 3 dans la liste [3, 5, 8, 10], l'algorithme effectue une seule comparaison, donc il s'exécute en **temps constant**.

```
liste = [3, 5, 8, 10]
valeur = 3
# Comparaison unique : liste[0] == 3
# Complexité : O(1)
```

b) Pire cas : O(n)

Le pire cas survient dans deux situations :

1. **L'élément recherché est le dernier élément** de la liste.
2. **L'élément recherché n'est pas présent** dans la liste.

Dans ces deux scénarios, l'algorithme doit parcourir **tous les éléments** de la liste avant de trouver l'élément ou de conclure qu'il est absent. Cela signifie que l'algorithme effectue n comparaisons, où n est la taille de la liste.

Exemple 1 : Élément en dernière position

Si on recherche 10 dans la liste [3, 5, 8, 10], l'algorithme doit comparer chaque élément successivement avant d'arriver au dernier.

```
liste = [3, 5, 8, 10]
valeur = 10
# Comparaison pour chaque élément : liste[0], liste[1], liste[2], puis
liste[3] == 10
# Complexité : O(n)
```

Exemple 2 : Élément absent

Si on recherche 6 dans la liste [3, 5, 8, 10], l'algorithme doit également comparer chaque élément de la liste avant de conclure que l'élément n'est pas présent.

```
liste = [3, 5, 8, 10]
valeur = 6
# Comparaison pour chaque élément : liste[0], liste[1], liste[2], liste[3]
# Élément absent, après n comparaisons.
# Complexité : O(n)
```

c) Complexité moyenne : O(n)

La complexité moyenne de l'algorithme est également O(n), car en moyenne, l'élément recherché sera trouvé après avoir examiné environ la **moitié des éléments** dans une liste non triée. Cela se traduit par environ n / 2 comparaisons. Cependant, dans la notation asymptotique O, les constantes sont ignorées, donc on considère toujours la complexité comme O(n).

Illustration

- Si la taille de la liste est 10 et l'élément recherché se trouve quelque part au milieu, disons à la position 5, l'algorithme fera environ 5 comparaisons en moyenne.
- Cela reste proportionnel à n, et ainsi, même si c'est environ la moitié des comparaisons, la complexité est toujours O(n).

En résumé :

- **Meilleur cas** : O(1), lorsque l'élément est trouvé dès la première comparaison.
- **Pire cas** : O(n), lorsque l'élément est le dernier ou absent.
- **Complexité moyenne** : O(n), car en moyenne, on parcourt environ la moitié des éléments avant de trouver l'élément.

La recherche séquentielle est simple à mettre en œuvre mais peut devenir inefficace pour les grandes listes, en particulier lorsque l'élément est proche de la fin ou absent.

B. Tri par sélection

1. Définition

Le **tri par sélection** est un algorithme de tri qui fonctionne en sélectionnant, à chaque itération, l'élément le plus petit (ou le plus grand) dans la portion non triée de la liste, puis en le plaçant à sa position correcte dans la portion triée.

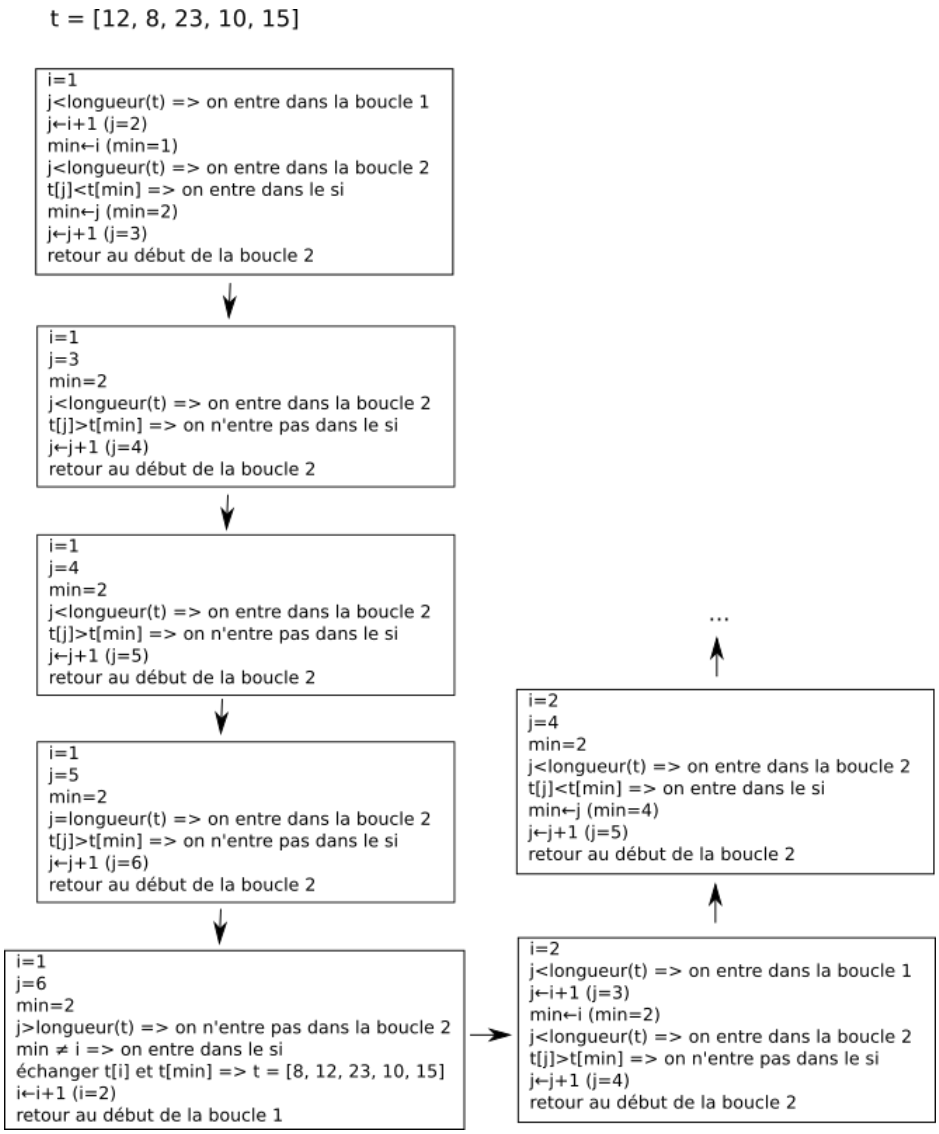
2. Principe

- Diviser la liste en deux parties : une partie triée (initialement vide) et une partie non triée.
- À chaque itération, trouver le plus petit élément de la partie non triée.
- Échanger cet élément avec le premier élément non trié.
- Répéter le processus jusqu'à ce que toute la liste soit triée.

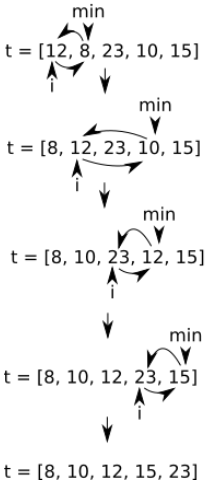
```
VARIABLE
t : tableau d'entiers
i : nombre entier
min : nombre entier
j : nombre entier
DEBUT
i←1
tant que i<longueur(t):   //boucle 1
  j←i+1
  min←i
  tant que j<=longueur(t):   //boucle 2
    si t[j]<t[min]:
      min←j
    fin si
    j←j+1
  fin tant que
  si min≠i :
    échanger t[i] et t[min]
  fin si
  i←i+1
fin tant que
FIN
```

Activité

Poursuivez le travail commencé ci-dessous (attention de bien donner l'état du tableau)



On peut résumer le principe de fonctionnement de l'algorithme de tri par sélection avec le schéma suivant :



Activité

Essayez de produire le même type de schéma explicatif que ci-dessus avec le tableau t = [15, 16, 11, 13, 12]

3. Implémentation en Python

```
def tri_selection(liste):
    n = len(liste)
    for i in range(n):
        # Trouver l'indice du plus petit élément dans la sous-liste [i:n]
        indice_min = i
        for j in range(i+1, n):
            if liste[j] < liste[indice_min]:
                indice_min = j
        # Échanger l'élément à l'indice i avec l'élément à l'indice
indice_min
        liste[i], liste[indice_min] = liste[indice_min], liste[i]
    return liste
```

4. Complexité

Essayons maintenant de déterminer la complexité de l'algorithme de tri par sélection :

Pour établir la complexité de cet algorithme, nous allons comptabiliser les comparaisons entre 2 entiers.

- Si nous nous intéressons à l'étape qui nous permet de passer de

t = [12, 8, 23, 10, 15] à t = [8, 12, 23, 10, 15] (i = 1) nous avons 4 comparaisons : 12 avec 8, puis 8 avec 23, puis 8 avec 10 et enfin 8 avec 15.

- Si nous nous intéressons à l'étape qui nous permet de passer de t = [8, 12, 23, 10, 15] à t = [8, 10, 23, 12, 15] (i = 2) nous avons 3 comparaisons : 12 avec 23, puis 12 avec 10, et enfin 10 avec 15.
- Si nous nous intéressons à l'étape qui nous permet de passer de t = [8, 10, 23, 12, 15] à t = [8, 10, 12, 23, 15] (i = 3) nous avons 2 comparaisons : 23 avec 12 et 12 avec 15
- Si nous nous intéressons à l'étape qui nous permet de passer de t = [8, 10, 12, 23, 15] à t = [8, 10, 12, 15, 23] (i = 4) nous avons 1 comparaison : 23 avec 15

Pour trier un tableau comportant 5 éléments nous avons : 4 + 3 + 2 + 1 = 10 comparaisons

Dans le cas où nous avons un tableau à trier qui contient n éléments, nous aurons :

n-1 + n-2 + n-3 +....+ 3 + 2 + 1 comparaisons. Si vous n'êtes pas convaincu, faites le test avec un tableau de 6 éléments, vous devriez trouver 5 + 4 + 3 + 2 +1 = 15 comparaisons.

Conclusion : l'algorithme de tri par sélection a une complexité en $O(n^2)$ (complexité quadratique).

- **Meilleur cas** : $O(n^2)$ (aucun gain spécifique dans le meilleur cas).
- **Pire cas** : $O(n^2)$.
- **Complexité moyenne** : $O(n^2)$.
- **Note** : Cet algorithme n'est pas très efficace pour les grandes listes, mais il est simple à comprendre et à implémenter.

C. Comparaison des deux algorithmes

- La **recherche séquentielle** est un algorithme de recherche qui fonctionne indépendamment de l'ordre des éléments dans la liste. Il est utile lorsque les données ne sont pas triées ou lorsqu'un algorithme plus complexe comme la recherche binaire ne peut être utilisé.
- Le **tri par sélection** permet de réorganiser les éléments de la liste. Bien qu'il soit simple à comprendre, il est moins performant que d'autres algorithmes de tri comme le tri rapide ou le tri fusion pour des listes de grande taille.

D. Exercice pratique

1. **Recherche séquentielle** : Écrire une fonction qui prend une liste d'entiers et un nombre à rechercher, et qui renvoie un message indiquant si le nombre est présent ou non.
2. **Tri par sélection** : Implémenter le tri par sélection pour une liste de chaînes de caractères (au lieu d'entiers), et comparer les résultats avec le tri intégré de Python `(sorted()).`

Travaux Pratiques :

- Implémentation d'un algorithme de tri (ex : tri par insertion) en Python.
- Comparaison des complexités de différents algorithmes de tri.