Structures de données linéaires

Listes chaînées, piles et files

1 Introduction

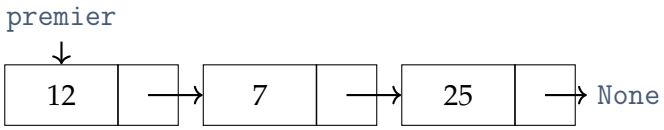
En programmation, le choix de la structure de données est essentiel pour résoudre efficacement un problème. Les tableaux (listes Python) permettent un accès direct aux éléments par leur indice, mais l'insertion ou la suppression en début de liste nécessite de décaler tous les éléments.

Dans ce chapitre, nous étudions trois structures de données fondamentales : les **listes chaînées**, les **piles** et les **files**. Chacune répond à des besoins spécifiques et offre des opérations optimisées pour certains usages.

2 Les listes chaînées

2.1 Définition et principe

Définition 1 **(Liste chaînée)** Une **liste chaînée** est une structure de données composée d'une suite de nœuds. Chaque nœud contient une valeur (donnée) et une référence (pointeur) vers le nœud suivant. Le dernier nœud pointe vers `None` (ou `null`).



2.2 Implémentation en Python

Exemple 1 (Structure d'un nœud). En Python, un nœud est représenté par une classe contenant l'élément et la référence vers le suivant :

```
class Noeud:
    def __init__(self, element,
                 suivant=None):
        self.element = element
        self.suivant = suivant
```

Exemple 2 (Classe Liste Chaînée). La liste chaînée est gérée par une classe dont l'attribut `premier` désigne le premier nœud, ou vaut `None` si la liste est vide :

```
class ListeChainee:
    def __init__(self):
        self.premier = None

    def est_vide(self):
        return self.premier == None
```

2.3 Opérations fondamentales

Pour une liste chaînée simple de n éléments :

- Insertion en tête : $O(1)$
- Insertion en queue : $O(n)$
- Suppression en tête : $O(1)$
- Recherche d'un élément : $O(n)$
- Accès au k-ième élément : $O(k)$

Exemple 3 (Insertion en tête). Pour insérer un nouvel élément en tête de liste :

```
def ajout_en_tete(self, element):
    nouveau = Noeud(element)
    nouveau.suivant = self.premier
    self.premier = nouveau
```

Exemple 4 (Parcours de la liste). Pour parcourir tous les éléments d'une liste chaînée :

```
def parcourir(self):
    courant = self.premier
    while courant:
        print(courant.element)
        courant = courant.suivant
```

2.4 Types de listes chaînées

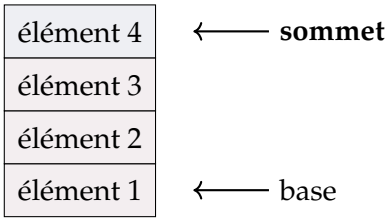
Remarque 1 (Variantes). Il existe plusieurs variantes de listes chaînées :

— **Liste simplement chaînée** : chaque nœud pointe vers le suivant uniquement.
— **Liste doublement chaînée** : chaque nœud possède un pointeur vers le suivant *et* vers le précédent.
— **Liste circulaire** : le dernier nœud pointe vers le premier.

③ Les piles (Stack)

3.1 Principe LIFO

Définition 2 **(Pile)** Une **pile** (en anglais *stack*) est une structure de données linéaire suivant le principe **LIFO** (*Last In, First Out*) : le dernier élément ajouté est le premier à être retiré.



empiler/dépiler par le sommet

3.2 Opérations sur les piles

Une pile supporte les opérations suivantes :

— `empiler(x)` : ajoute x au sommet de la pile.
— `depiler()` : retire et renvoie l'élément au sommet.
— `sommet()` : renvoie l'élément au sommet sans le retirer.
— `est_vide()` : teste si la pile est vide.

Toutes ces opérations s'effectuent en temps constant O(1).

Exemple 5 (Implémentation avec une liste Python).

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

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

    def depiler(self):
        if self.est_vide():
            raise IndexError("Pile vide")
        return self.elements.pop()

    def sommet(self):
        if self.est_vide():
            raise IndexError("Pile vide")
        return self.elements[-1]

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

3.3 Applications des piles

Exemple 6 (Vérification de parenthèses). Les piles sont utilisées pour vérifier l'équilibrage des parenthèses dans une expression :

```python
def parentheses_equilibrees(expression):
    pile = Pile()
    correspondances = {')': '(', ']': '[',
    '}': '{'}

    for c in expression:
        if c in '([{':
            pile.empiler(c)
        elif c in ')]}':
            if pile.est_vide():
                return False
            if pile.depiler() !=
    correspondances[c]:
                return False

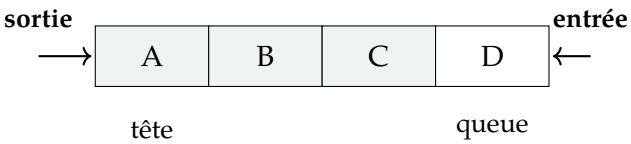
    return pile.est_vide()
```

Remarque 2 (Autres applications). Les piles sont également utilisées pour :
— la gestion des appels de fonctions (pile d'exécution) ;
— l'évaluation d'expressions arithmétiques (notation polonaise inverse) ;
— la fonction « annuler » (*undo*) dans les éditeurs de texte ;
— l'historique de navigation dans un navigateur web.

④ Les files (Queue)

4.1 Principe FIFO

Définition 3 (File) Une **file** (en anglais *queue*) est une structure de données linéaire suivant le principe **FIFO** (*First In, First Out*) : le premier élément ajouté est le premier à être retiré.



4.2 Opérations sur les files

Une file supporte les opérations suivantes :

- `enfiler(x)` : ajoute x à la fin de la file.
- `defiler()` : retire et renvoie l'élément en tête.
- `tete()` : renvoie l'élément en tête sans le retirer.
- `est_vide()` : teste si la file est vide.

Exemple 7 (Implémentation avec une liste Python). Une implémentation simple utilisant une liste Python :

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

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

    def defiler(self):
        if self.est_vide():
            raise IndexError("File vide")
        return self.elements.pop(0)

    def tete(self):
        if self.est_vide():
            raise IndexError("File vide")
        return self.elements[0]

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

Remarque 3 (Complexité de pop(0)). L'opération `pop(0)` a une complexité $O(n)$ car tous les éléments doivent être décalés vers la gauche. Pour de petites files, cela reste acceptable. Pour des applications nécessitant de meilleures performances, d'autres implémentations existent (avec deux piles ou avec une liste chaînée).

4.3 Applications des files

Exemple 8 (Simulation d'une file d'attente). Les files sont utilisées pour simuler des files d'attente, par exemple dans un guichet :

```python
def simulation_guichet():
    file = File()

    # Arrivée de clients
    file.enfiler("Client 1")
    file.enfiler("Client 2")
    file.enfiler("Client 3")

    # Traitement des clients (dans l'ordre d'arrivée)
    while not file.est_vide():
        client = file.defiler()
        print(f"Service de {client}")
```

Remarque 4 (Autres applications). Les files sont utilisées dans de nombreux contextes :

- gestion des processus dans un système d'exploitation ;
- files d'attente (imprimantes, serveurs web) ;
- gestion des tâches à traiter dans l'ordre d'arrivée ;
- buffers de communication (stockage temporaire de données).

⑤ Comparaison et synthèse

Critère	Liste chaînée	Pile	File
Principe	Séquentiel	LIFO	FIFO
Insertion	O(1) en tête	O(1) au sommet	O(1) en queue
Suppression	O(1) en tête	O(1) au sommet	$O(n)$ en tête
Accès	$O(n)$	Sommet seul	Tête seule
Usage	Général	Historique, Ctrl-Z	File d'attente

Remarque 5 (Choix de la structure). Le choix entre ces structures dépend du problème :

- besoin d'accès séquentiel flexible → **liste chaînée** ;
- besoin de traiter le dernier arrivé en premier → **pile** ;
- besoin de traiter dans l'ordre d'arrivée → **file**.

6 Exercices

Exercice 1 Implémenter la méthode spéciale `__len__` de la classe `ListeChainee`, qui renvoie le nombre d'éléments de la liste : Python l'appelle quand on écrit `len(liste)`.

Corrigé de l'exercice 1.

```
def __len__(self):
    courant = self.premier
    cpt = 0
    while courant:
        courant = courant.suivant
        cpt += 1
    return cpt
```

Exercice 2 Implémenter une méthode `contient(element)` pour la classe `ListeChainee` qui renvoie `True` si `element` est présent dans la liste, `False` sinon.

Corrigé de l'exercice 2.

```
def contient(self, element):
    courant = self.premier
    while courant:
        if courant.element ==
            element:
            return True
        courant = courant.suivant
    return False
```

Exercice 3 Écrire une fonction `inverser_chaine(chaine)` qui utilise une pile pour inverser une chaîne de caractères. Par exemple, `inverser_chaine("hello")` renvoie `"olleh"`.

Corrigé de l'exercice 3.

```
def inverser_chaine(chaine):
    pile = Pile()
    for c in chaine:
        pile.empiler(c)
    resultat = ""
    while not pile.est_vide():
        resultat = resultat + pile.
            depiler()
    return resultat
```

Exercice 4 Implémenter une file à l'aide de deux piles. Indice : une pile pour enfiler, une pile pour défiler.

Corrigé de l'exercice 4.

```
class FileAvecDeuxPiles:
    def __init__(self):
        self.pile_entree = Pile()
        self.pile_sortie = Pile()

    def enfiler(self, x):
        self.pile_entree.empiler(x)

    def defiler(self):
        if self.pile_sortie.est_vide
            ():
            while not self.
                pile_entree.est_vide():
                self.pile_sortie.
                    empiler(self.
                    pile_entree.depiler())
        return self.pile_sortie.
            depiler()
```

Exercice 5 Écrire une fonction qui détermine si une chaîne de caractères est un palindrome en utilisant une pile.

Corrigé de l'exercice 5.

```
def est_palindrome(chaine):
    pile = Pile()
    for c in chaine:
        pile.empiler(c)
    for c in chaine:
        if c != pile.depiler():
            return False
    return True
```