

Sujet n°4

Algorithmique et Structure de Données Structures de données linéaires

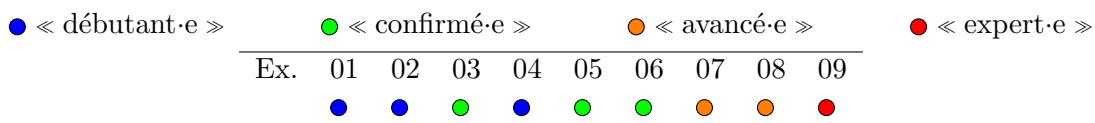
Temps de réalisation: 1h30mn

L'objectif de ce TP est de :

- se familiariser avec les listes chaînées, dans une version avec double chaînage circulaire
- concevoir une implémentation alternative des piles et des files d'attente
- appréhender les files d'attente à deux extrémités
- évaluer le coût des opérations sur les listes chaînées
- comparer les listes et les tableaux
- résoudre des problèmes en mobilisant des structures de données linéaires et leurs opérations

Pro Tip : il est super-méga-extra-recommandé de se munir d'un papier et d'un crayon pour dessiner des tonnes de cases et de flèches qui représentent les listes chaînées en mémoire et leur évolution en fonction des scénarios étudiés.

Échelle de progression :



Pour acquérir un niveau de compétence donné, il fautachever **tous** les exercices de ce niveau et des niveaux inférieurs. L'objectif est d'atteindre le niveau *confirmé·e* (●) à l'issue du TP.

Table des matières

1 Les listes chaînées	2
Exercice 1 : Définition d'une liste doublement chaînée circulaire	2
Exercice 2 : Les opérations élémentaires	3
Exercice 3 : Topswops	4
2 Files, piles et déques	5
Exercice 4 : Implémentation de déques à l'aide de listes chaînées	5
Exercice 5 : La valeur maximale d'une fenêtre glissante	5
Exercice 6 : Retour sur les piles et les files	5
3 Pour aller plus loin...	6
Exercice 7 : Algorithme de Graham	6
3.1 Les listes à enjambements	6
Exercice 8 : Première implémentation	7
Exercice 9 : La médiane mobile	7

Introduction

Les structures de données linéaires offrent la possibilité de mémoriser des variables qui sont des collections finies totalement ordonnées d'éléments pouvant présenter des répétitions. L'idée simple et fondamentale sous-jacente est que chaque élément de la collection vient avant ou après un autre élément.

Elles permettent donc d'implémenter certains types abstraits « linéaires » très utiles en informatique tels que les séquences et des formes restreintes comme les piles et les files d'attente à une ou deux extrémités. Nous avons d'ailleurs déjà rencontré une structure de données linéaire : le tableau. Il s'agit en fait d'une implémentation de « séquence » présentant des opérations et des performances spécifiques, obtenues grâce à une allocation contigüe qui permet un accès direct aux éléments à partir du rang (ou indice).

Les structures de données linéaires sont également fondamentales pour certaines implémentations de types abstraits non linéaires, tels que les files de priorités, les partitions d'ensembles voire certains arbres et graphes !

Dans la suite de cet énoncé, nous nous intéressons à nouveau au type abstrait *séquence* (ou *liste*). Cependant, nous mettons l'accent sur la propriété d'accès séquentiel aux éléments, plutôt que sur l'accès direct par l'indice. En d'autres termes il s'agit, à partir d'un voisinage donné, de parcourir la liste de proche en proche pour accéder à l'élément visé. En effet, dans certaines situations, il est utile de pouvoir désigner la « place » d'un élément sans pour autant connaître son rang : par exemple, lorsque l'on s'écarte temporairement d'une file d'attente, on souhaite pouvoir s'y insérer à nouveau « au même endroit » même si on ne connaît pas son rang ou si celui-ci évolue. Les repères sont alors les voisins de devant et de derrière.

1 Les listes chaînées

Afin de traduire cette idée d'accès séquentiel, il faut disposer d'un moyen pour désigner une « place » dans la séquence. En informatique, les structures de données vont permettre de créer des objets « place » et leurs *références* vont être employées pour réaliser un chaînage entre places consécutives et représenter la séquence : on parle alors de *liste chaînée*.

Il existe en pratique de multiples variantes de listes chaînées, répondant toutes au type abstrait *séquence* et proposant des fonctionnalités supplémentaires et des performances variées :

- les listes simplement chaînées ;
- les listes doublement chaînées ;
- les listes chaînées circulaires ;
- les listes doublement chaînées circulaires ;
- toute variante de chaînage propre à un usage particulier...

Exercice 1: Définition d'une liste doublement chaînée circulaire

Dans cet exercice, vous allez créer à l'aide d'un classe les structures de données et les opérations nécessaires à la gestion d'une liste de nombres entiers.

QUESTION 1. Créer les types `LinkedList` et `Cell` pour représenter une **liste doublement chaînée circulaire** (*Circular Doubly Linked List*) de maillons¹ (`cell`) contenant des nombres entiers.

La définition de la classe `Cell` requiert le symbole `Cell` lui-même : elle est récursive. En Python, à partir de la version 3.7, ce mécanisme est disponible.

QUESTION 2. Écrire le constructeur, et le test de liste vide :

1. Les maillons représentent les « places » dans la liste.

QUESTION 3. Ajouter 3 « observateurs », i.e., des méthodes d'accès aux informations de la liste : la taille, la tête et la queue. qui donnent respectivement, la taille, la première et la dernière cellule de la liste. Les fonctions `head()` et `tail()` retournent `None` lorsque la liste est vide.

Exercice 2: Les opérations élémentaires

QUESTION 1. Écrire la méthode `insert`

```
def insert(item: int,  
          neighbor: Cell,  
          after: bool = True) -> LinkedList
```

qui insère l'élément `item` dans une cellule adjacente à `neighbor`. Si l'indicateur `after` est à *vrai*, alors l'insertion a lieu après `neighbor`, sinon elle a lieu avant.

QUESTION 2. Décliner les méthodes `insert_head` et `insert_tail`, cas particuliers de la fonction d'insertion, qui ajoutent un élément respectivement en tête et en queue de liste.

QUESTION 3. Puis proposer une méthode de sérialisation des listes :

```
def __str__() -> str
```

qui retourne la représentation sérialisée (textuelle) de la liste ℓ , principalement en vue d'un affichage sur la sortie standard (Terminal) :

```
>>> print(liste)  
[1, 2, 3, 2, 1]
```

L'affichage de la liste doit reproduire l'affichage des listes natives de Python, e.g. [1, 2, 3, 2, 1].

Remarque: La méthode de sérialisation, qui nécessite un parcours de la liste, peut s'écrire de manière itérative ou récursive, au choix.

QUESTION 4. Définir un test de validité :

```
def isChained() -> bool
```

qui vérifie que le chaînage et la taille de la liste sont corrects.

QUESTION 5. Écrire les méthodes de recherche par valeur et par rang :

```
def find(item: int) -> Optional[Cell]  
def findAt(i: int) -> Cell
```

La méthode `find()` recherche la première occurrence de l'élément `item` et retourne—une référence vers—le maillon si la recherche aboutit, `None` sinon. La méthode `findAt()` fournit le maillon en position i .

QUESTION 6. Écrire les méthodes de lecture et d'écriture suivantes :

```
def getValue(idx: ???) -> int  
def setValue(idx: ???, item: int) -> LinkedList
```

respectivement, qui retourne l'élément (la valeur entière) donné à la place `idx` de la liste chaînée ℓ , et qui affecte la valeur `item` à la place `idx` de la liste. Il vous appartient de spécifier le type `(???)` d'une place.

QUESTION 7. Ajouter la méthode

```
def remove(cell: Cell) -> LinkedList
```

qui supprime la cellule `cell` de la chaîne.

QUESTION 8. Écrire la méthode

```
def extend(l1: LinkedList) -> LinkedList
```

qui concatène deux listes chaînées. Les données de la liste ℓ_2 sont ajoutées à la fin de la liste objet.

QUESTION 9.

Exercice 3: Topswops

Le défi *Topswops* figure dans la liste des *Al Zimmermann's Programming Contests* et consiste à trouver la permutation des entiers de 1 à n qui maximise le nombre d'itérations nécessaires pour atteindre une permutation commençant par 1, en appliquant une opération unique : inverser la sous-séquence préfixe, de taille égale à la valeur du premier élément. On réitère jusqu'à observer la valeur 1 en première position !

Par exemple, pour $n = 6$, la permutation (3, 6, 5, 1, 4, 2) requiert une inversion des **3** premiers éléments—la sous-séquence (3, 6, 5) est transformée en (5, 6, 3)—dans la première itération. L'intégralité du procédé donne lieu à 10 itérations :

0. [3, 6, 5], 1, 4, 2
1. [5, 6, 3, 1, 4], 2
2. [4, 1, 3, 6], 5, 2
3. [6, 3, 1, 4, 5, 2]
4. [2, 5], 4, 1, 3, 6
5. [5, 2, 4, 1, 3], 6
6. [3, 1, 4], 2, 5, 6
7. [4, 1, 3, 2], 5, 6
8. [2, 3], 1, 4, 5, 6
9. [3, 2, 1], 4, 5, 6
10. 1, 2, 3, 4, 5, 6

La sous-séquence entre crochets indique les éléments à inverser à chaque itération. Après la dixième itération, le chiffre 1 se trouve en première position. L'algorithme s'arrête. Donc pour $n = 6$, il existe une permutation, (3, 6, 5, 1, 4, 2), qui génère 10 itérations du mécanisme proposé. Le défi est de trouver, pour n donné, la permutation qui génère le maximum d'itérations !

QUESTION 1. Pour commencer, il faut être capable d'inverser les éléments dans une liste. Ajouter à la bibliothèque d'opérations sur les listes circulaires doublement chaînées, une fonction d'inversion des k premiers éléments.

Remarque: Il est possible soit d'inverser les valeurs des cellules, soit d'inverser les cellules elles-mêmes. C'est cette dernière approche qui est préconisée.

QUESTION 2. Ensuite, nous avons besoin d'une fonction qui énumère une à une, toutes les permutations d'une liste circulaire doublement chaînée. Écrire cette fonction.

Remarque: L'énumération des permutations s'écrit aisément de manière récursive : l'étape d'induction considère successivement l'insertion de l'élément de tête à toutes les positions de chaque permutation du reste de la liste.

QUESTION 3. Résoudre le problème *Topswops* par énumération exhaustive des permutations, pour $n \in \llbracket 4, 9 \rrbracket$.

Remarque: La solution a été trouvée pour $n = 97$. N'essayez pas !

2 Files, piles et dèques

Au cours du TP précédent, nous avons vu comment implémenter les types abstraits *pile* et *file d'attente* de plusieurs manières. En fait, les files et les piles sont des cas particuliers de la *file d'attente à deux extrémités* nommée *dèque*.

Dans cette partie, nous allons donc nous intéresser à une implémentation alternative des piles et des files d'attente, à partir d'un nouveau type *dèque*.

Exercice 4: Implémentation de dèques à l'aide de listes chaînées

Un dèque est une file d'attente dans laquelle il est possible d'entrer et de sortir aux deux extrémités. En cela, il combine les opérations sur les files d'attente et sur les piles.

Créer une classe `Deque` de file d'attente à deux extrémités pour des nombres entiers, par composition à partir des listes circulaires doublement chaînées `LinkedList`.

Proposer l'interface de programmation pour la classe `Deque` qui regroupe toutes les primitives pour dèques vues en cours sous forme de méthodes d'instance :

Exercice 5: La valeur maximale d'une fenêtre glissante

Étant donné une séquence d'entiers, on souhaite fournir la valeur maximale de toute sous-séquence de k entiers consécutifs. Il s'agit en pratique de calculer un agrégat (le max) d'une fenêtre glissante de taille k sur un flux de données.

La séquence initiale est de type `list[int]` et peut être construite à l'aide de la fonction `remplir_tableau` du module `tp1.util`.

QUESTION 1. Écrire une première version très simple du programme à l'aide d'une double boucle sur les éléments de la séquence et sur ceux de la fenêtre courante.

QUESTION 2. Écrire une version plus astucieuse, à l'aide d'un dèque (type `Deque`) en charge de mémoriser la valeur maximale courante et les éventuelles valeurs maximales suivantes.

Remarque: Le dèque contient toujours la valeur maximale en tête et insère les valeurs suivantes par la queue. Il est à noter que toute valeur du dèque qui serait moins grande que la valeur courante n'aurait aucune chance de figurer dans les réponses (le max) des fenêtres suivantes !

QUESTION 3. Comparer les deux résolutions, en terme de coût.

Exercice 6: Retour sur les piles et les files

QUESTION 1. Proposer un implémentation alternative des types `Stack` et `Queue` à partir de `Deque`. Tous les opérateurs sur les files et les piles seront redéfinis.

QUESTION 2. Vérifier que les programmes qui utilisent les types `Stack` et `Queue` fonctionnent parfaitement avec cette implémentation alternative.

3 Pour aller plus loin...

Exercice 7: Algorithme de Graham

L'enveloppe convexe d'un ensemble de points est la plus petite forme convexe qui les contient tous. Pour un ensemble fini de points, cette enveloppe est un polygone. Il existe de nombreux algorithmes d'extraction d'enveloppe convexe. Celui de GRAHAM repose sur l'idée que les points d'un polygone convexe tournent dans le même sens. Une fois les points triés (ici, dans le sens trigonométrique), il suffit de supprimer ceux qui provoquent des concavités.

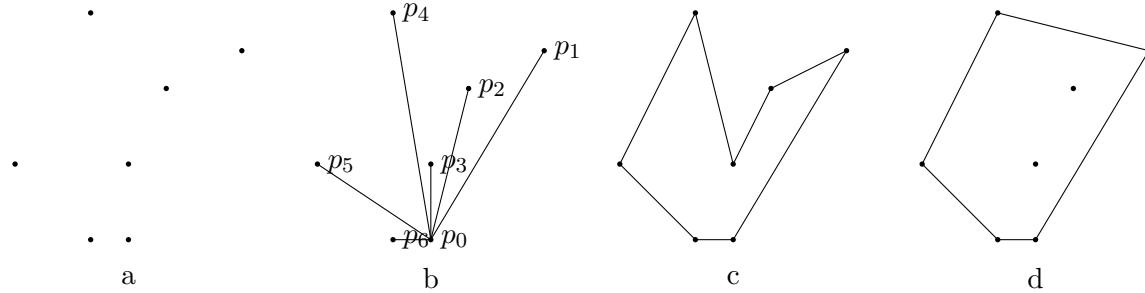


FIGURE 1 – Illustration du principe de l'algorithme de GRAHAM. (a) : un ensemble de points. (b) : choix du point pivot p_0 et tri angulaire des autres points. (c) : polygone non convexe formé par les points ordonnés. (d) : enveloppe convexe obtenue après suppression des concavités.

Cet algorithme met en jeu la recherche dans une liste, le tri d'une liste et la structure de pile.

QUESTION 1. On dispose de n points représentés par une liste de tuples (x, y) . Déterminer le point pivot p_0 , situé en bas à droite de tous les autres (celui qui a la plus petite ordonnée ou, à ordonnée minimale, celui qui a la plus grande abscisse).

QUESTION 2. Trier les autres points selon le sens trigonométrique. p_i et p_j sont dans l'ordre trigonométrique si p_j est à gauche de $\overrightarrow{p_0p_i}$, autrement dit si le déterminant $\det(\overrightarrow{p_0p_i}, \overrightarrow{p_0p_j}) = (x_i - x_0)(y_j - y_0) - (y_i - y_0)(x_j - x_0)$ est positif. Écrire une fonction intermédiaire `determinant`, qui servira également à la question suivante.

QUESTION 3. Supprimer les concavités selon l'algorithme suivant. Les points p_0 et p_1 sont placés dans une pile qui contiendra finalement les points de l'enveloppe convexe. Par construction, p_0 et p_1 appartiennent nécessairement à l'enveloppe convexe. On considère le point p_k , k étant initialisé à 2. Tant que k est inférieur à n ($n = 7$ sur la Figure 1), on considère à chaque étape les deux points p_i et p_j en haut de la pile. Ils forment la dernière arête du polygone en construction. Si p_k est à gauche de $\overrightarrow{p_ip_j}$ alors on empile p_k et on passe au point suivant (on incrémente k), sinon, on enlève le point qui crée une concavité (on dépile p_j sans changer k). On arrête quand on a empilé le dernier point (à cette étape, $k = n - 1$).

3.1 Les listes à enjambements

Les listes à enjambements (ou *skip list* en anglais) sont des listes chaînées triées multi-couches. La première couche consiste en une liste chaînée usuelle dont les éléments ont été ordonnés, tandis que chaque couche supérieure offre un « raccourci » pour la liste immédiatement inférieure :

```

couche 3:   A
couche 2:   A---C-----G
couche 1:   A---C-D----G---I
couche 0:   A-B-C-D-E-F-G-H-I-J
  
```

En pratique, les couches supérieures ne sont constituées que de références vers les cellules de la première couche. La recherche dans une liste à enjambements s'opère de la couche la plus haute vers la plus basse, en se déplaçant au plus près de la valeur recherchée dans chaque couche. Par exemple, le chemin de recherche de la valeur H est :

```
A[3]
-> A[2] -> C[2] -> G[2]
      -> G[1]
      -> G[0] -> H[0]
```

L'insertion procède de la même manière pour établir le rang du nouvel élément au sein de la liste ordonnée. On insère toujours l'élément dans la première couche. La décision de faire figurer un élément dans la couche supérieure est totalement aléatoire ! Les *skip list* sont en effet des structures probabilistes. Le paramètre clé d'une structure de *skip list* est donc la probabilité p de figurer dans la couche supérieure. En pratique, on utilise souvent la valeur $p = 0,5$ qui suggère que la moitié des éléments d'une couche est supposée constituer la couche supérieure. Le nombre de couches est alors variable et déterminé par les tirages aléatoires successifs lors de l'insertion de nouveaux éléments. Il est toutefois d'usage de fixer une borne supérieure ℓ au nombre de couches.

Pour une présentation détaillée des *skip list*, vous pouvez consulter [ce diaporama](#), ou commencer simplement par l'[entrée Wikipedia](#). Pour les plus hardis, il est possible de consulter l'[article original de W. Pugh \(1990\)](#) qui présente les *skip list*.

Exercice 8: Première implémentation

QUESTION 1. Implémenter un type **SkipList** de nombres entiers, muni de ses opérations élémentaires : **new**, **is_empty**, **insert**, **delete**, **lookup**.

QUESTION 2. Établir les performances—complexités—des opérations sur les *skip list*, en tenant compte du paramètre p .

Exercice 9: La médiane mobile

Il s'agit de calculer la valeur médiane pour chaque fenêtre glissante de taille w sur un tableau de valeurs entières de taille n .

Avec une liste à enjambements pour stocker les éléments de la fenêtre courante de manière ordonnée, le procédé requiert une recherche par index (recherche de l'élément au rang $w//2$). Or, cette recherche est a priori en $O(n)$, sauf si l'on ajoute une information de « rang relatif » avec toute référence vers une cellule.

L'exemple précédent devient :

```
couche 3:  A/0
couche 2:  A/0----C/2-----G/4
couche 1:  A/0----C/2-D/1-----G/3----I/2
couche 0:  A/0-B/1-C/1-D/1-E/1-F/1-G/1-H/1-I/1-J/1
```

Il s'interprète comme suit : le parcours de la couche 2 nous renseigne sur les rangs de A, C et G. Le rang de A est obtenu par un saut de 0 à partir du début de la liste et vaut donc 0 ! Le rang de C est alors $\text{rang}(A) + 2$ soit 2. Le rang de G est quant à lui $\text{rang}(C) + 4 = 6$. Ce procédé fonctionne également le long de chemins descendants. Par exemple, chercher l'élément de rang 7 consiste à parcourir le chemin :

```
A[3] (0)
-> A[2] (0) -> C[2] (0+2=2) -> G[2] (2+4=6)
      -> G[1] (6)
      -> G[0] (6) -> H[0] (6+1=7!)
```

QUESTION 1. Modifier le type `SkipList` pour y intégrer l'information de rang relatif.

QUESTION 2. Proposer une résolution pour le calcul de la médiane mobile.