Listes chaînées

Structures de données

Exercice 1 *QCM : vocabulaire des listes chaînées*

Pour chaque question, une seule réponse est correcte.

1. Dans une liste chaînée, chaque nœud contient :
   - A. uniquement une valeur
   - B. une valeur et une référence vers le nœud suivant
   - C. une valeur, une référence vers le suivant et une référence vers le précédent
   - D. une valeur et son indice dans la liste
2. Pour représenter une liste vide, on utilise :
   - A. un nœud dont la valeur est 0
   - B. un nœud dont la valeur est `None`
   - C. un objet `ListeChainee` dont l'attribut `premier` vaut `None`
   - D. la valeur `[]`
3. Quelle est la complexité de l'ajout en tête d'une liste chaînée ?
   - A. $O(n)$ car il faut décaler tous les éléments
   - B. $O(1)$ car on crée un nœud et on met à jour une référence
   - C. $O(n)$ car il faut parcourir la liste pour trouver la tête
   - D. $O(\log n)$ car on utilise une recherche dichotomique
4. On considère la liste chaînée `5 → 3 → 8 → None`. Quelle expression Python permet d'accéder à la valeur 8 ?
   - A. `ma_liste.premier.element`
   - B. `ma_liste.premier.suivant.element`
   - C. `ma_liste.premier.suivant.suivant.element`
   - D. `ma_liste.ieme_element(3)`
5. Par rapport au type `list` de Python (tableau dynamique), la liste chaînée est plus performante pour :
   - A. l'accès à un élément par son indice
   - B. les insertions et suppressions en tête
   - C. la recherche d'un élément
   - D. le calcul de la longueur

Exercice 2 *Exercice guidé : manipuler une liste chaînée pas à pas*

On rappelle les classes vues en cours :

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

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

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

    def ajout_en_queue(self, element):
        """Insère element en queue de liste"""
        if self.premier == None:
            self.ajout_en_tete(element)
        else:
            courant = self.premier
            while courant.suivant != None:
                courant = courant.suivant
            dernier = Noeud(element)
            courant.suivant = dernier
```

On exécute :

```
ma_liste = ListeChainee()
ma_liste.ajout_en_tete(7)
ma_liste.ajout_en_tete(3)
ma_liste.ajout_en_tete(9)
```

1. Dessiner l'état de la liste chaînée après chaque instruction `ajout_en_tete`.
2. Quelle est la valeur de `ma_liste.premier.element` ?
3. Quelle est la valeur de `ma_liste.premier.suivant.suivant.element` ?
4. On ajoute `ma_liste.ajout_en_tete(1)`. Dessiner le nouvel état.

Exercice 3 *Compléter des méthodes à trous*

Compléter les méthodes suivantes en remplaçant les `...` :

a) Méthode `__len__` qui renvoie le nombre d'éléments :

```
def __len__(self):
    courant = ...           # partir du premier noeud
    cpt = 0
    while courant:
        courant = ...       # passer au noeud suivant
        cpt += 1
    return ...              # renvoyer le compteur
```

b) Méthode `__repr__` qui renvoie `"9 -> 3 -> 7"` :

```
def __repr__(self):
    elements = []
    courant = self.premier
    while courant:
        elements.append(str(...))     # ajouter la valeur
        courant = ...                  # passer au suivant
    return " -> ".join(elements)
```

c) Méthode `contient` qui renvoie `True` si `element` est dans la liste :

```
def contient(self, element):
    courant = self.premier
    while courant:
        if ... == element:   # comparer avec la valeur
            return True
        courant = ...        # passer au suivant
    return ...               # element non trouve
```

Exercice 4 *Accès et suppression en tête*

1. Écrire la méthode `supprime_en_tete(self)` qui supprime le premier élément de la liste et renvoie sa valeur. Si la liste est vide, la méthode doit lever `ValueError("liste vide")`.
2. Tester sur la liste `9 → 3 → 7` :
   - appeler `supprime_en_tete()` une première fois : quelle valeur est renvoyée ? Quel est l'état de la liste ?
   - appeler `supprime_en_tete()` deux fois de plus ;
   - un dernier appel : que se passe-t-il ?

Exercice 5 *Insertion en position quelconque*

Écrire la méthode `inserer(self, element, i)` qui insère `element` en position `i` (à partir de 0).

Indications :

- si `i == 0`, utiliser `ajout_en_tete` ;
- sinon, parcourir jusqu'au nœud de position `i - 1` ;
- créer un nouveau nœud dont le suivant est l'ancien nœud en position `i` ;
- mettre à jour le suivant du nœud en position `i - 1`.

Tester avec la liste `9 → 3 → 7` :

- `inserer(5, 0)` doit donner `5 → 9 → 3 → 7` ;
- `inserer(1, 2)` doit donner `5 → 9 → 1 → 3 → 7`.

Exercice 6 *Comparaison de complexité*

Compléter le tableau suivant en indiquant la complexité de chaque opération :

Opération	`list` Python	Liste chaînée
Accès par indice		
Ajout en tête		
Ajout en queue		
Suppression en tête		
Suppression en queue		
Recherche d'un élément		

Exercice 7 *Synthèse : ajouter en queue en temps constant*

Dans la classe `ListeChainee` de l'exercice 2, la méthode `ajout_en_queue` parcourt toute la liste pour atteindre le dernier nœud. On écrit une nouvelle classe, `ListeAvecDernier`, qui garde en plus une référence `dernier` vers le dernier nœud. Quand la liste est vide, `premier` et `dernier` valent tous les deux `None`.

a) Quelle est la complexité de la méthode `ajout_en_queue` de la classe `ListeChainee` ? Pourquoi ?

b) Compléter les méthodes de la classe `ListeAvecDernier`, pour que `ajout_en_queue` s'exécute en temps constant et que `dernier` désigne toujours le dernier nœud :

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

    def ajout_en_queue(self, element):
        nouveau = Noeud(element)
        if self.dernier == None:
            self.premier = ...
        else:
            self.dernier.suivant = ...
        self.dernier = ...

    def ajout_en_tete(self, element):
        nouveau = Noeud(element, self.premier)
        self.premier = nouveau
        if ...:
            self.dernier = ...

    def supprime_en_tete(self):
        """Retire le premier œnud d'une liste non vide et renvoie son élément."""
        element = self.premier.element
        self.premier = ...
        if ...:
            self.dernier = ...
        return element
```

c) Dans `supprime_en_tete`, que se passerait-il si l'on ne remettait pas `dernier` à `None` quand la liste devient vide ? Dérouler la suite d'opérations suivante : `l = ListeAvecDernier()`, puis `l.ajout_en_queue(5)`, `l.supprime_en_tete()` et `l.ajout_en_queue(8)`.

d) Garder une référence vers le dernier nœud ne rend pas la suppression en queue plus rapide. Pourquoi ?