Listes chaînées

Sommaire

1. Introduction
2. La structure de liste chaînée
3. Opérations sur les listes
4. Exercices

Sommaire

1. Introduction
2. La structure de liste chaînée
3. Opérations sur les listes
4. Exercices

Introduction

- Pour traiter des séquences d'éléments, la structure de tableau est proposée par la majorité des langages de programmation.
- Cette structure de données n'est pas efficace pour des opérations d'insertion ou de suppression, lorsque celles-ci ne sont pas faites sur le dernier élément du tableau.
- En Python, de telles opérations sur les tableaux sont proposées : `t.insert(i,v)` insère la valeur `v` dans la case d'indice `i` du tableau `t`.
- Exemple : pour insérer un nouvel élément au début d'un tableau `t` il faut :
  1. augmenter d'une case la taille de `t` ;
  2. décaler d'une case vers la droite tous les éléments de `t` (sans écraser de valeur) ;
  3. écrire le nouvel élément dans la première case de `t`.

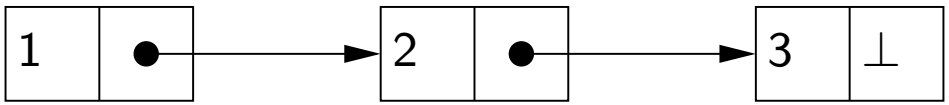
- le nombre d'opérations de l'insertion donnée en exemple est proportionnel à la taille du tableau. Pour insérer un nouvel élément un tête d'un tableau contenant 1 million d'éléments, il faut un million d'opérations.
- On a la même complexité pour l'opération de suppression d'un élément en tête d'un tableau.
- Les listes chaînées vont :
  - permettre de réduire la complexité de ces opérations d'insertion/suppression
  - seront réutilisées pour construire d'autres structures de données (arbres, graphes) qui seront étudiées plus tard.
- Remarque : dans une majorité de cas, on utilisera une liste chaînée pour représenter des listes homogènes, c.à.d d'éléments tous du même type (des entiers par exemple).

Sommaire

1. Introduction
2. La structure de liste chaînée
3. Opérations sur les listes
4. Exercices

La structure de liste chaînée

- Comme son nom l'indique, cette structure est composée d'éléments *chaînés* entre eux, de telle sorte qu'on puisse accéder, à partir de n'importe quel élément, à son successeur dans la liste (sauf pour le dernier).
- La figure ci-dessous est une représentation possible d'une liste chaînée :



- Chaque élément de la liste est une cellule contenant :
  - la valeur de l'élément ;
  - une référence [1] vers l'élément suivant de la liste.
- on peut implémenter une telle structure de cellule à l'aide d'une classe, comme ci-dessous :

```
class Cellule:
    """une cellule d'une liste chaînée"""
    def __init__(self, v, s):
        self.valeur = v
        self.suivante = s
```

1. C'est à dire une adresse mémoire pointant vers cet élément.

- Dans cette classe, l'attribut `valeur` contient la valeur de l'élément de la liste, et `suivante` est l'adresse de la cellule dans laquelle se trouve l'élément suivant de la liste.
- On peut construire la liste de la figure précédente avec cette classe en tapant l'instruction suivante :

```
lst=Cellule(1, Cellule(2, Cellule(3, None)))
```

- la valeur prédéfinie `None` est utilisée pour représenter la fin de la liste.
- On peut remarquer que dans sa définition même, une liste est soit la valeur `None` (qui représente la liste vide), soit un objet de la classe `Cellule` dont l'attribut `suivante` contient l'adresse d'une liste : c'est donc une définition *récursive*.

Sommaire

1. Introduction
2. La structure de liste chaînée
3. Opérations sur les listes
4. Exercices

Opérations sur les listes

On montre, dans ce paragraphe, comment utiliser la classe `Cellule` pour écrire quelques méthodes permettant de réaliser des opérations de base sur les listes.

Longueur d'une liste

- On souhaite écrire une fonction `longueur(lst)`, qui prend en entrée une variable `lst` représentant une liste chaînée et qui renvoie en sortie la longueur de cette liste.
- on propose de montrer comment écrire cette fonction de façon récursive, puis itérative.
- Pour le calcul récursif :
  1. le cas de base correspond à une liste vide (égale à `None`). Dans ce cas il faut renvoyer 0.
  2. Pour le cas récursif : la liste n'étant pas vide, sa longueur est égale à 1 + `longueur(lst.suivante)`.

- On en déduit le code suivant :

```
def longueur(lst):
    """renvoie la longueur de la liste chaînée lst
    version récursive"""
    if lst is None:
        return 0
    else:
        return 1 + longueur(lst.suivante)
```

Ex. 1 Écrire une version itérative de la fonction `longueur(lst)`.

N^e élément d'une liste

- On souhaite écrire une fonction `nieme_element(n, lst)` qui renvoie le n^e élément de la liste `lst`, lorsque celui-ci existe, et qui lève une exception `IndexError` sinon. Par convention, on numérote les éléments à partir de 0.
- On propose d'écrire une version itérative de cette fonction. Il s'agit de *parcourir* la liste à l'aide d'une variable `l`. Un compteur `i` permettra lui de mémoriser l'avancement à l'intérieur de la liste.
- On doit prendre garde de ne pas tenter de dépasser la fin de liste, ce qui donne une double condition d'arrêt dans la boucle : on avance dans la liste tant que le compteur `i` est strictement inférieur à `n` et qu'on est pas arrivé en fin de liste.

```python
def nieme_element(n, lst):
    """renvoie le n-ième élément de la liste lst
    les éléments sont numérotés à partir de 0"""
    if lst is None or n < 0:
        raise IndexError("indice invalide")
    i = 0
    l = lst
    while i < n and l is not None:
        i += 1
        l = l.suivante
    if l is None:
        raise IndexError("indice invalide")
    return l.valeur
```

Ex. 2 Écrire une version récursive de la fonction `nieme_element(n, lst)`.

Concaténation de deux listes

- On souhaite maintenant réaliser la concaténation de deux listes, c'est à dire si la première contient 1,2,3 et la deuxième 4,5 obtenir la liste 1,2,3,4,5.
- La fonction à écrire se nomme `concatener(l1, l2)` : elle prend deux listes chaînées `l1` et `l2` en entrée et renvoie la liste obtenue par concaténation de `l1` et `l2`.
- On part de l'hypothèse importante que toute liste, une fois qu'elle a été créée, n'est plus modifiée.
- Il est assez facile de trouver une implémentation récursive de `concatener(l1, l2)`, en se concentrant sur la structure de `l1` :
  1. le cas de base est obtenu dans le cas où `l1` est vide, auquel cas il faut renvoyer `l2` ;
  2. sinon, on renvoie la liste qui commence par le premier élément de `l1` et qui est suivie de la concaténation de la suite de `l1` avec `l2`, c'est à dire on renvoie `Cellule(l1.valeur, concatener(l1.suivante, l2))`.

```
def concatener(l1, l2):
    """fonction qui renvoie la liste résultant de la concaténation
    des listes l1 et l2"""
    if l1 is None:
        return l2
    else:
        return Cellule(l1.valeur, concatener(l1.suivante, l2))
```

- Ex. 3 Écrire une fonction `affiche_liste(lst)` qui affiche, en utilisant la fonction `print`, tous les éléments de la liste `lst`, séparés par des espaces, suivis d'un retour chariot. L'écrire comme une fonction récursive, puis avec une boucle `while`.
- Ex. 4 Écrire une fonction `listeN(n)` qui reçoit en argument un entier `n`, supposé positif ou nul, et renvoie la liste des entiers 1,2,...,`n` dans cet ordre. Si `n` = 0, la liste renvoyée est vide.
- Ex. 5 Écrire une fonction `occurrences(x, lst)` qui renvoie le nombre d'occurrences de la valeur `x` dans la liste `lst`. L'écrire comme une fonction récursive, puis avec une boucle `while`.
- Ex. 6 Écrire une fonction `trouve(x, lst)` qui renvoie le rang de la première occurrence de `x` dans `lst` le cas échéant, et `None` sinon. L'écrire comme une fonction récursive, puis avec une boucle `while`.

Renverser une liste

On souhaite maintenant écrire une fonction `renverse(lst)` qui renverse la liste `lst`, c'est à dire, si `lst` contient 1,2,3, qui renvoie la liste 3,2,1. Une approche récursive est possible :

1. Le cas de base correspond à une liste `l` vide, auquel cas on renvoit `None` ;
2. Le cas récursif montre que le premier élément de `l` doit devenir le dernier de la liste renversée et que le début de la liste renversée est le renversement de la queue de la liste `l` (liste qui commence au deuxième élément de `l`). On peut donc utiliser la fonction `concatener(l1, l2)` :

```
return concatener(renverser(lst.suivante), Cellule(lst.valeur, None)
```

On obtient le code donné page suivante.

Sommaire

1. Introduction
2. La structure de liste chaînée
3. Opérations sur les listes
4. Exercices

Exercices

```
def renverser(lst):
    """renvoie la liste lst renversée"""
    if lst is None:
        return None
    else:
        return concatener(renverser(lst.suivante), Cellule(lst.valeur, None))
```

Remarque : si on essaye d'évaluer la complexité de cette fonction écrite récursivement, on va observer qu'elle n'est pas très bonne, car elle n'est pas linéaire (proportionnelle à la taille de la liste). Par exemple, pour une liste de 1000 éléments, le coût est de l'ordre de 500 000 opérations : il faut d'abord concaténer (le renversement d') une liste de 999 éléments, ce qui coûte 999 opérations[2], puis de 998,..., puis de 1. Au total : $999 + 998 + \cdots + 1 = 499\ 500$ opérations.

2. Car le coût de la concaténation de `l1` avec `l2` est directement proportionnel à la taille de `l1`, donc ici 999. Pour s'en convaincre, relire le code-source de la fonction `concatener` à la page 13.

- Ex. 7 Écrire une version itérative de la fonction `renverser(lst)` dont la complexité soit linéaire.
- Ex. 8 Écrire une fonction `identiques(l1, l2)` qui renvoie un booléen indiquant si les listes chaînées `l1` et `l2` sont identiques, c'est à dire contiennent exactement les mêmes éléments, dans le même ordre. On suppose que l'on peut comparer les éléments de `l1` et `l2` avec le test logique d'égalité `==` de Python.
- Ex. 9 Écrire une fonction `inserer(x, lst)` qui prend en argument un entier `x` et une liste chaînée d'entiers `lst`, supposée triée par ordre croissant, et qui renvoie une nouvelle liste chaînée dans laquelle `x` a été inséré à sa place. Ainsi, insérer 3 dans la liste 1,2,5,8 renvoie la liste chaînée contenant dans l'ordre 1,2,3,5,8. On pourra écrire `inserer` de façon récursive.
- Ex. 10 Écrire une fonction `liste_de_tableau(t)` qui renvoie une liste chaînée contenant les éléments du tableau `t`, dans le même ordre. On pourra écrire `liste_de_tableau` avec une boucle `for`.

Ex. 11 Écrire une fonction `copie(lst)` qui renvoie une copie de la liste chaînée `lst`. Utiliser cette fonction `copie` pour écrire une nouvelle version de la fonction `concatener(l1, l2)`. Cette nouvelle version renvoie une nouvelle liste chaînée dont tous les éléments sont des cellules distinctes des cellules de `l1` et `l2`, mais dont le contenu est la concaténation de `l1` et `l2`.

Ex. 12 Écrire une classe `Liste` qui permet de manipuler des listes construites avec des objets de la classe `Cellule`. On pourra utiliser un constructeur comme celui donné dans le code ci-dessous :

```
class Liste:
    """une liste chaînée"""
    def __init__(self, t=None):
        self.tete = t
```

Écrire les méthodes suivantes dans cette classe :

`est_vide(self)` qui renvoie `True` si la liste est vide, `False` sinon.

`ajoute(self, x)` qui ajoute à la liste la valeur `x` en tête de la liste.

`__len__(self)` qui renvoie la longueur de la liste (réutiliser, en l'adaptant à la classe `Liste`, l'algorithme de la fonction `longueur` précédemment codée, mais ne pas l'appeler).

`__str__(self)` qui renvoie la chaîne de caractères contenant les éléments de la liste, dans l'ordre et séparés d'un espace.

`__getitem__(self, n)` qui renvoie le nième élément de la liste (réutiliser l'algorithme de la fonction `nieme_element` précédemment codée).

`reverse(self)` qui renverse les éléments de la liste (réutiliser l'algorithme de la fonction `renverse` précédemment codée).

`__add__(self, lst)` qui renvoie la liste obtenue en concaténant la liste à la liste `lst` (réutiliser l'algorithme de la fonction `concatener` précédemment codée).

Tester ces méthodes sur plusieurs listes.