5.2 Implémentation en Python des listes

NSI terminale - JB Duthoit

5.2.1 Utiliser directement un tableau dynamique python (ou un tuple...) ou bien utiliser des listes chaînées ?

Utiliser un tableau dynamique python comme liste

Dans certains langages de programmation(comme dans le langage Python), 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)
Les tableaux de Python permettent par exemple d'insérer ou de supprimer efficacement des éléments à la fin d'un tableau, avec les opérations append et pop. Il permettent aussi d'insérer un élément avec :

```
t.insert(0,4)
```

⚠ En python, les listes sont en réalité des tableaux dynamiques, à mi-chemin entre les listes et les tableaux !

Utilisation de listes chaînées

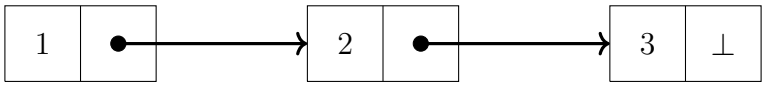
Autre structure qui permet d'implémenter des listes : les listes chaînées

Définition

Une liste chaînée sert à représenter une liste, c'est-à-dire une séquence finie de valeurs, par exemple des entiers.

Chaque élément est stocké dans un petit bloc alloué quelque part dans la mémoire, que l'on pourra appeler maillon ou cellule, et y est accompagné d'une deuxième information : l'adresse mémoire où se trouve la cellule contenant l'élément suivant de la liste.

Considérons la liste contenant trois éléments, respectivement 1, 2 et 3.



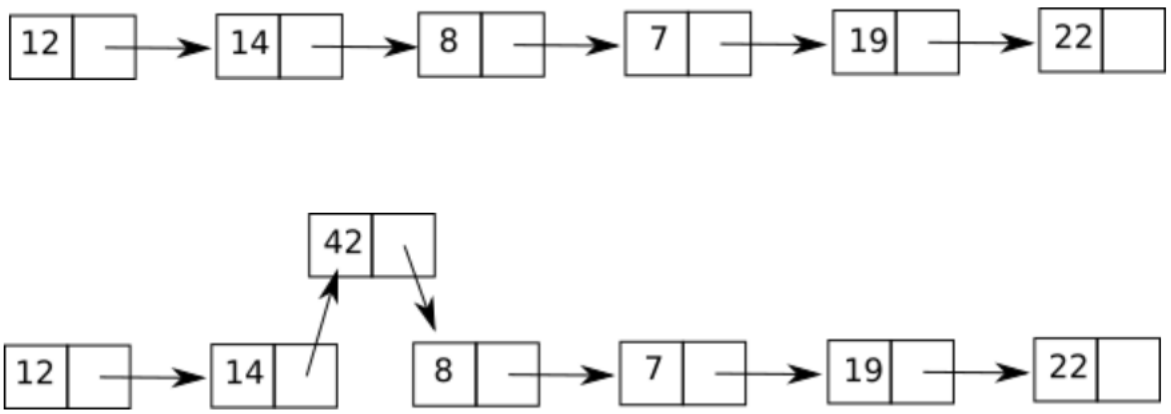
Chaque élément de la liste est matérialisé par un emplacement en mémoire contenant :

- d'une part sa valeur (dans la case de gauche)
- d'autre part l'adresse mémoire de la valeur suivante (dans la case de droite).

Remarque

Pour le dernier élément, qui ne possède pas de valeur suivante, on utilise une valeur spéciale désignée ici par le symbole ⊥ et marquant la fin de la liste

Il est assez facile d'insérer un élément lorsque l'on travaille avec une liste chaînée : Supposons que l'on veuille insérer l'élément "42" en troisième position dans cette liste :



Histoire

Le premier interpréteur fonctionnait sur un ordinateur IBM 704 et deux instructions de cette machine devinrent les deux opérations primitives de Lisp pour décomposer les listes :

- car (contents of address register) : le premier élément de la liste.
- cdr (contents of decrement register) : le reste de la liste.
- L'opération qui consiste à fabriquer une liste à partir d'un premier élément et d'une liste est notée cons.

Complexité

Pourquoi implémenter plusieurs structures ? Après tout, on peut tout faire avec des listes Python !

Parce que l'efficacité est fondamentale. Certaines structures sont plus adaptées à certains problèmes.

Accéder à un élément : tableau Pour accéder à l'élément 2 du tableau T = ['a', 'b', 'd'], On se rend à l'adresse où débute T
On se déplace de deux positions. On lit : 'd' Le temps est constant : Accéder se fait en complexité O(1).

Accéder à un élément : liste chaînée Considérons la liste chaînée suivante :

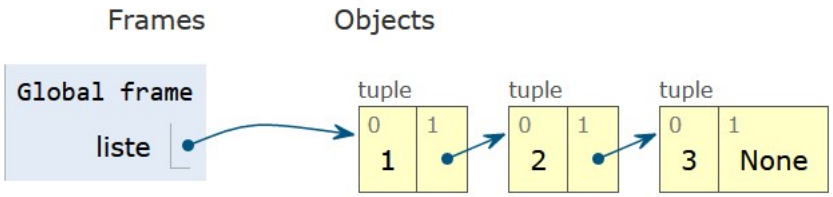
```
d = ['d',None]
b = ['b',d]
a = ['a',b]
```

Pour accéder à l'élément `'d'` de la liste On se rend à l'adresse où débute L On suit le lien jusque l'adresse de la queue du premier élément On suit le lien jusque l'adresse de la queue du second élément On lit la valeur de la tête : 'c' Le temps est linéaire : Accéder se fait en complexité O(n).

Insérer un élément Comme on l'a vu plus haut, c'est le contraire !
Cette opération est plus rapide pour les listes chaînées que pour les tableaux.

5.2.2 Implémenter une liste chainée avec des tuples

On peut utiliser un tuple, et dans ce cas écrire `(1,(2,(3,None)))`.



5.2.3 Implémenter une liste chaînée avec des tableaux

On peut aussi encore un tableau à deux éléments, et dans ce cas écrire `[1,[2,[3,None]]]`.

Une liste peut être encore être représentée par deux tableaux, l'un (contenu) contenant des valeurs et l'autre (suivant) contenant des indices. Le chaînage sera effectué de la façon suivante : l'élément suivant contenu[k] aura suivant[k] comme indice dans le tableau contenu.

	contenu	suivant
0	'2'	1
1	'3'	-1
2	'1'	0

Exercice 5.1
Considérons la liste chaînée suivante :

```
d = ['d',None]
b = ['b',d]
a = ['a',b]
```

Proposer un script python qui permet d'insérer la valeur `'c'` entre `'b'` et `'d'` dans la liste chaînée précédente.

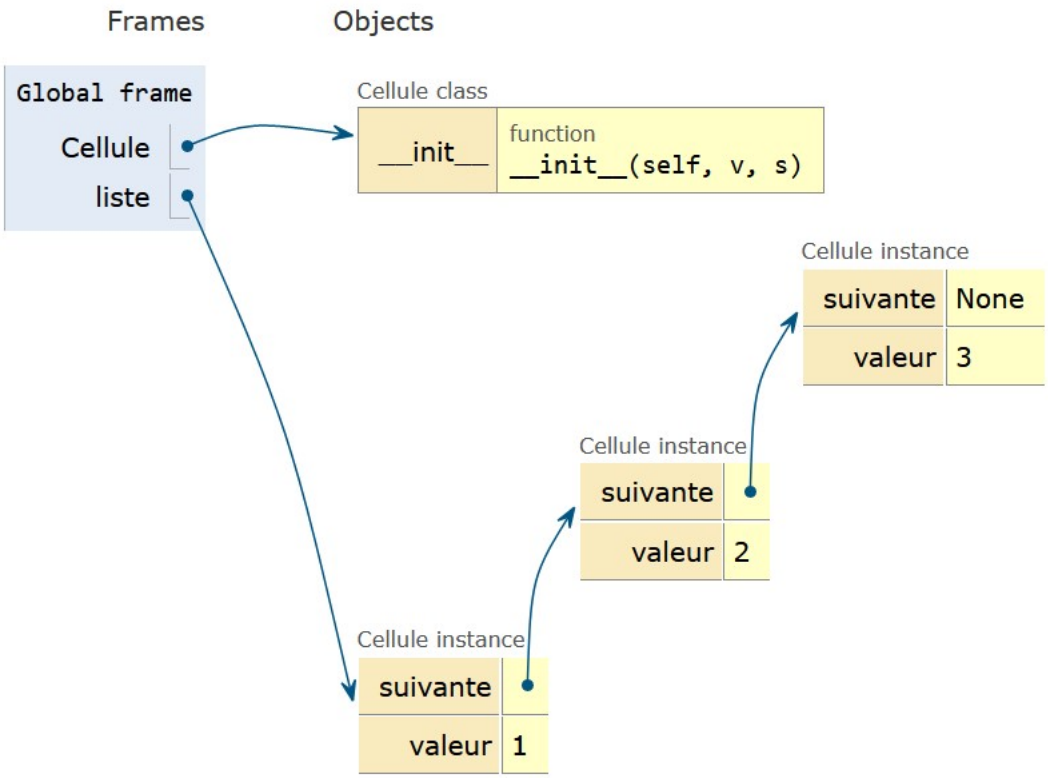
5.2.4 Implémenter une liste chainée avec des objets

Une façon d'utiliser des listes chaînées avec Python, est d'utiliser une classe :

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

Ainsi la liste 1,2,3 est possible avec l'instruction :

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



Remarque

None remplace $\perp$.

Définition

Une liste chaînée est une structure de données pour représenter une séquence finie d'éléments. Chaque élément est contenu dans une cellule, qui fournit par ailleurs un moyen d'accéder à la cellule suivante. Les opérations sur les listes chaînées se programment sous la forme de parcours qui suivent ces liaisons, en utilisant une fonction récursive ou une boucle

Exercice 5.2

Pour cet exercice, on considère la classe suivante, qui permet de fabriquer des listes chaînées :

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

On écrira `lst = None` lorsque `lst` est une liste vide.
Écrire une fonction `listeN(n)` qui reçoit un argument entier `n` et qui renvoie la liste chaînée des entiers 1,2,3,...,n dans cet ordre. Si n = 0, la liste renvoyée est vide.

Exercice 5.3
Pour cet exercice, on considère la classe suivante, qui permet de fabriquer des listes chaînées :

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

On écrira `lst = None` lorsque `lst` est une liste vide.
Écrire une fonction `longueur(lst)` qui reçoit pour argument une liste chaînées `lst` et qui retourne la longueur de la liste chaînées.

1. L'écrire avec une boucle While
2. L'écrire comme fonction récursive

Exercice 5.4
Pour cet exercice, on considère la classe suivante, qui permet de fabriquer des listes chaînées :

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

On écrira `lst = None` lorsque `lst` est une liste vide.
Écrire une fonction `affiche_ liste(lst)` qui reçoit en argument une liste chaînée `lst` et qui affiche, en utilisant print, tous les éléments de la liste `lst`, séparés par des espaces, suivies d'un retour chariot.

1. L'écrire avec une boucle While
2. L'écrire comme fonction récursive

Exercice 5.5
Pour cet exercice, on considère la classe suivante, qui permet de fabriquer des listes chaînées :

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

On écrira `lst = None` lorsque `lst` est une liste vide.
Écrire une fonction `n_ ieme_ element(lst,n)` de paramètres `n` et `lst` une liste chaînée, et qui renvoie le n-ième élément de la liste

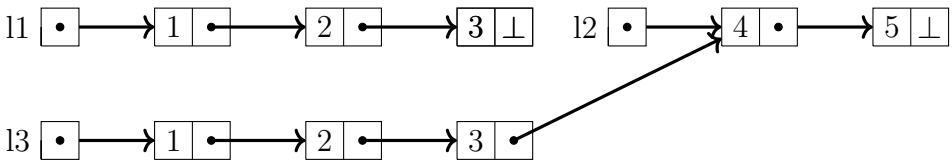
1. L'écrire de façon récursive
2. L'écrire avec une boucle While

Exercice 5.6
Pour cet exercice, on considère la classe suivante, qui permet de fabriquer des listes chaînées :

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

On écrira `lst = None` lorsque `lst` est une liste vide.
On souhaite maintenant mettre bout à bout deux listes. On appelle cela une **concaténation**. Écrire une fonction concaténer qui reçoit deux listes en arguments et renvoie une troisième liste contenant la concaténation.

```
def concatenerr(l1,l2):
    ''' concatène l1 et l2 ,
    sous la forme d'une nouvelle liste
    si l1 est vide, alors on renvoie l2
    si l2 est vide, alors on renvoie l1
    sinon la concaténation est obtenue en concaténant la tête de l1 et la concaténation du reste de
    l1=Cellule(1,Cellule(2,Ceellule(3,None)))
    l2= Cellule(4,Cellule(5,None))
    l3 = concatener(l1,l2)
    '''
```



Remarque

On voit que les cellules de l1 ont été dupliquée tandis que les cellules de l2 partagées.Les cellules l2 permettent de constituer la liste l2 et la fin de la liste l3.Une alternative consisterait à dupliquer également les cellules de l2, mais ceci n'est pas forcément nécessaire.

Exercice 5.7
Pour cet exercice, on considère la classe suivante, qui permet de fabriquer des listes chaînées :

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

On écrira `lst = None` lorsque `lst` est une liste vide.
Écrire une fonction `derniere_cellule(lst)` qui renvoie la dernière cellule de la liste `lst`.

Exercice 5.8
Pour cet exercice, on considère la classe suivante, qui permet de fabriquer des listes chaînées :

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

On écrira `lst = None` lorsque `lst` est une liste vide.
Écrire une fonction `liste_de_tableau(t)` qui renvoie une liste qui contient les éléments du tableau `t`, dans le même ordre. On pourra utiliser une boucle `for`.

Exercice 5.9

Créer la classe `ListeChainee` qui permet de créer une liste chaînée en utilisant l'objet `cellule`.
Cette classe aura les **primitives** suivantes (ou l'**interface** suivante) :

- Création d'une liste vide avec le constructeur.
- La liste chaînée est-elle vide ?
- Ajout d'une valeur en tête de liste
- Suppression d'une valeur en tête de liste
- Nombre d'éléments de la liste (longueur)
- Accès à un élément en fonction de sa position, son indice
- Affichage de la liste

5.2.5 Opérations avec les listes chaînées

Longueur de la liste

```
def longueur(l):
    if lst is None:
        return 0
    else:
        return 1 + longueur(l.suivante)
```

Complexité : la complexité du calcul de la longueur est directement proportionnelle à la longueur elle-même, puisqu'on réalise un nombre constant d'opérations pour chaque cellule de la liste.
Ainsi, pour une liste l de mille cellules, longueur(l) va effectuer mille tests, mille appels récursifs et mille additions.

Renvoyer le n-ème élément d'une liste

```
def nieme_element(n, l):
    if l is None:
        raise IndexError("indice invalide")
    if n == 0:
        return l.valeur
    else:
        return nieme_element(n-1,l.suivante)
```

Concaténer deux listes

```
def concatener(l1, l2):
"""concatène les listes l1 et l2,
sous la forme d'une nouvelle liste"""
i f l1 i s None:
return l2
else:
return Cellule(l1.valeur, concatener(l1.suivante,l2))
```