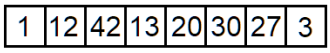
En Python, la classe `list` a été pour l'instant notre structure privilégiée pour représenter des listes d'éléments. Cependant, en accord avec la modularité, on n'a jamais vraiment vu comment cette classe est implémentée.

Sans trop rentrer dans les détails, la classe `list` de Python est implémentée avec une structure de tableau dynamique (cf. ex 19). En pratique, cela signifie que les données stockées dans une `list` le sont de manière contiguë dans la mémoire de l'ordinateur :

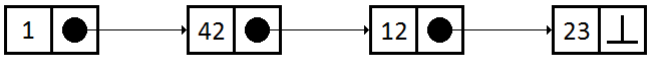


Cette implémentation a des avantages et des inconvénients. Parmi les avantages, on trouve la rapidité en lecture et en écriture des éléments, effectuées en temps constant. En revanche, l'ajout de nouveaux éléments à la fin où à une place précise, bien que nativement possible cachent un coût potentiellement linéaire en la taille du tableau.

Nous allons étudier et implémenter dans ce chapitre une nouvelle structure : la liste chaînée.

1 Structure de liste chaînée

Une liste chaînée représente comme les tableaux une suite finie de valeurs. Sa particularité, suggérée par son nom, est que les éléments de la liste sont chaînés entre eux : chaque élément de la liste contient une valeur mais aussi l'adresse mémoire de l'élément suivant de la liste (s'il existe). On peut ainsi représenter la liste 1, 42, 12, 23 de la manière suivante :



Le symbole $\perp$ est utilisé ici pour signaler qu'il n'y a plus d'élément suivant.

Ainsi, pour accéder aux éléments de la liste, on accède au premier, qui nous donne accès au deuxième, etc. Voyons une première implémentation :

```
class Cellule:
""" classe decrivant les cellules d'une liste chainee """
    def __init__(self, v, s):
        self.valeur = v
        self.suivante = s
```

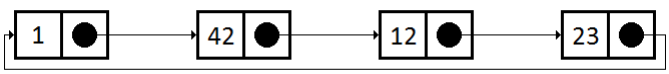
Une cellule d'une liste chaînée possède deux attributs : `valeur` et `suivante`. Cela correspond à la description faite précédemment. Pour créer la liste représentée plus haut on peut alors écrire :

```
lst = Cellule( 1, Cellule( 42, Cellule( 12, Cellule( 23, None))))
```

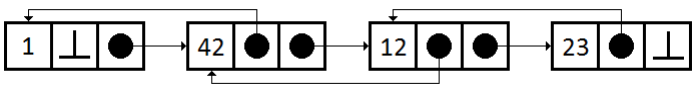
Remarque 1 : Cette définition est une définition récursive car on a :
— un cas de base : la liste vide représentée par `None` ;
— un cas récursif : une cellule contenant une valeur et la suite de la liste.
Cela va nous permettre d'écrire la plupart des opérations sur les listes chaînées de manière récursive. Cependant, pour éviter d'invoquer des méthodes sur `None`, on implémentera des fonctions en dehors de la classe `Cellule`.

Remarque 2 : Bien que cela ne soit pas interdit par la définition ni l'implémentation, on ne fera pas de listes contenant des valeurs de types différents. On s'assurera toujours de l'homogénéité de la liste.

Autres types de listes chaînées : Il est possible de concevoir des listes chaînée plus élaborées que ce que l'on vient de voir en rajoutant des informations dans chaque cellule. Deux variations sont couramment utilisées :
— les listes cycliques, où la dernière cellule renvoie vers la première :



— les listes doublement chaînées, ou chaque cellule peut renvoyer vers la suivante ou la précédente :



2 Opérations sur les listes

2.1 Longueur d'une liste :

Pour chaque opération à définir sur les listes, on commence par reformuler l'opération d'un point de vu récursif avant de l'implémenter. Pour les cas récursif, on parlera souvent de la tête de la liste (la première cellule) et de sa queue (le reste de la liste).

Définissons la longueur d'une liste récursivement :
— cas de base : la liste vide a une longueur de zéro ;
— cas récursif : la longueur d'une liste non-vide est la longueur de sa queue plus un.

D'où l'implémentation :

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

Remarque : le test `lst is None` a ici le même effet que `lst == None`. Cependant, il existe une différence importante entre ces deux formulations nous poussant à plutôt utiliser `is` :
— le `==` teste une égalité structurelle, c'est à dire une égalité dans les valeurs contenues dans les objets et peut être redéfini dans une classe ;
— le `is` teste une égalité physique, c'est à dire une égalité en terme de valeur et d'adresse mémoire.
`None` étant un objet unique ces deux tests coïncident mais afin de ne pas être mis en danger par une éventuelle redéfinition du `==`, il est d'usage d'utiliser `is` pour les comparaisons avec `None`, ce qu'on choisit de faire ici.

Complexité : La complexité est ici linéaire en la taille de la liste.

2.2 Accès au n-ième élément :

Essayons de définir récursivement l'accès au n-ième élément d'une chaîne :
— cas de base :
  — si la liste est vide, il n'y a pas d'élément à renvoyer : c'est une erreur d'indice ;
  — si $n = 0$ on renvoie la tête ;
— cas récursif : le n-ieme élément d'une liste est le $(n-1)$-ième élément de la queue de celle-ci.
D'où l'implémentation :

```
def nieme(lst,n):
    if lst is None:
        raise IndexError("Indice invalide")
    elif n==0:
        return lst.valeur
    else:
        return nieme(lst.suivante, n-1)
```

Complexité : La complexité est linéaire en `min(n,longueur de la liste)`.

2.3 Concaténation de deux listes :

Essayons de définir récursivement (sur la première liste) la concaténation de `lst1` et `lst2` :
— cas de base : si `lst1` est vide, on renvoie `lst2` ;
— cas récursif : la concaténation de `lst1` et `lst2` a pour tête la tête de `lst1` et pour queue la concaténation de la queue de `lst1` avec `lst2`.
D'où l'implémentation :

```
def concatener(lst1, lst2):
    if lst1 is None:
        return lst2
    else:
        return Cellule(lst1.valeur, concatener(lst1.suivante, lst2))
```

Complexité : La complexité est linéaire en `longueur de lst1`.

2.4 Renverser une liste :

Ici, on ne va pas pouvoir facilement utiliser une fonction récursive (même si cela est possible). On va plutôt utiliser une boucle et un accumulateur.

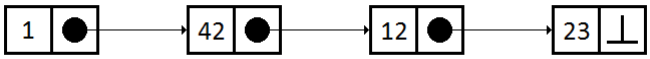
L'idée de l'algorithme est de créer deux listes `tmp` et `res` initialement égales à `lst` et `None`, puis de transvaser les éléments de `tmp` dans `res`. La liste `res` construite sera ainsi renversée. D'où l'implémentation :

```
def renverser(lst):
    tmp = lst
    res = None
    while not(tmp is None):
        res = Cellule(tmp.valeur, res)
        tmp = tmp.suivante
    return res
```

Complexité : La complexité est linéaire en la taille de la liste.

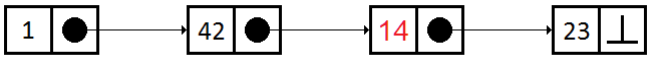
3 Modification d'une liste

Jusqu'à maintenant, aucune de nos fonctions ne modifie les listes, elles en créent et renvoient de nouvelles. S'il est possible de modifier les valeurs des listes et même leur structure, nous allons voir que cela est assez risqué. Reprenons notre liste `lst` (1, 42, 12, 23).



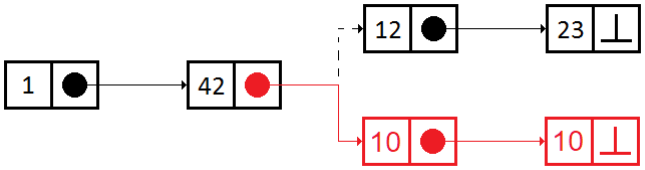
Modification d'une valeur : Supposons que l'on veuille changer le 12 en 14. On peut le faire via l'instruction :

```
lst.suivante.suivante.valeur = 14
```



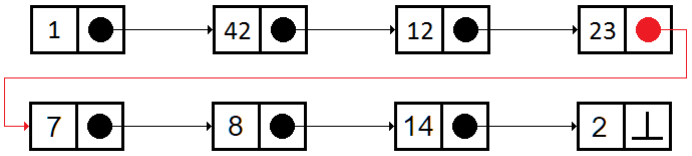
Modification de la structure : Toujours sur le même exmemple, il est possible de changer une partie ou la totalité de la queue d'une liste. Toujours sur `lst` (1,42,12,23), si on veut conserver (1,42) et modifier la fin en (10,10), on peut écrire :

```
lst2 = Cellule(10, Cellule(10, None))
lst.suivante.suivante = lst2
```



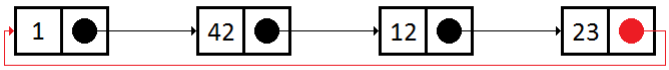
Concaténation par effet de bord : Il est possible de réécrire une fonction de concaténation qui modifie la première liste plutôt que d'en renvoyer une copie résultat. Pour cela, il suffit de modifier l'attribut `suivant` de la dernière cellule de la première liste en la première cellule de la seconde.

Un exemple pour la concaténation de `lst1` (1,42,12,13) et `lst2` (7,8,14,2) :



Ici pas de problème, on a modifié `lst1` comme prévu.

Cependant, une telle fonction est dangereuse à utiliser car elle peut produire des résultats différents de ce à quoi on s'attend. Par exemple, pour la concaténation de `lst1` avec elle même, on s'attend à obtenir deux fois les éléments de `lst1`. Malheureusement, on va avoir le comportement suivant :



Cette fois, on se retrouve avec une structure de liste cyclique, ce qui n'est pas prévu dans l'implémentation de nos fonctions ! Ainsi, appeler les fonctions de nos listes simplement chaînées risque de générer des boucles infinies puisque le cas de base `None` n'est plus présent.

4 Encapsulation dans un objet

On termine par l'encapsulation de nos objets dans une classe définitive `Liste`. L'encapsulation est ici souhaitable pour deux raisons :

— La cellule `None` est un peu problématique car n'étant pas du type `Cellule` et relevant de l'implémentation de la classe (on aurait pu faire d'autres choix).

— On veut éviter de laisser la possibilité à un utilisateur de coder des fonctions mettant à mal la structure, comme la concaténation par effet de bord.

Notre classe finale se présentera donc comme une classe à un attribut `tete` de type `Cellule` et dont les méthodes appellent les fonctions relatives aux cellules (privées).

```
class Liste:
    """ classe liste chainee """
    def __init__(self):
        self.tete = None

    def est_vide(self):
        """ teste si la liste est vide """
        return self.tete is None

    def ajoute(self, x):
        """ ajoute x en tete de la liste """
        self.tete = Cellule(x, self.tete)

    def __getitem__(self, n):
        """ surcharge de getitem, permet d acceder a l element i par self[i] """
        return _nieme(self.tete, n)

    def __len__(self):
        """ surcharge de len """
        return _longueur(self.tete)

    def reverse(self):
        """ renverse la liste par effet de bord """
        self.tete = _renverser(self.tete)

    def __add__(self, lst):
        """ renvoie la concatenation des deux listes, surcharge + """
        r = Liste()
        r.tete = _concatener(self.tete, lst.tete)
        return r
```