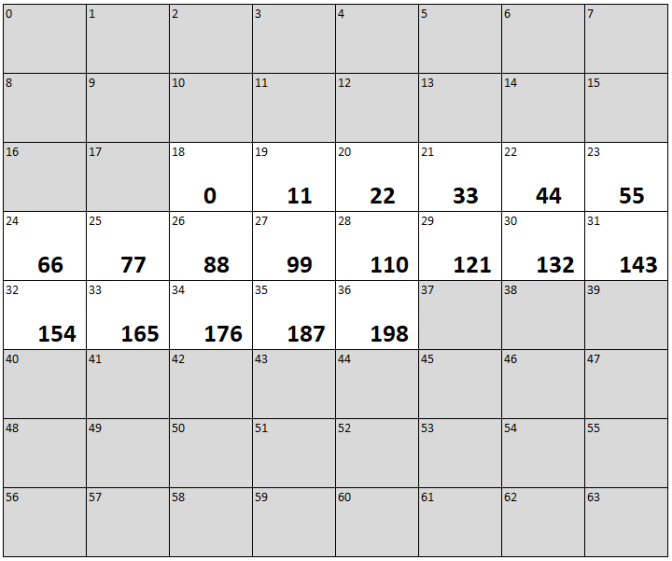
Listes

Introduction

Les schémas qui suivent concernant l'organisation logicielle de la mémoire sont des modèles pour vous permettre de mieux appréhender la distinction entre listes et tableaux, ils ne correspondent donc pas exactement à la réalité.

Organisation en mémoire d'un tableau

Les tableaux de données sont stockés en mémoire par des cases contiguës. Voici un exemple d'affectation de mémoire à un tableau `tab` commençant à la case mémoire `18` et comportant les premiers carrés des entiers positifs ou nuls. Le fait que le tableau soit stocké dans des cases contiguës permet d'accéder en temps constant aux différents éléments du tableau à partir de leur indice. Sur l'exemple ci-contre, si le programme souhaite accéder à `tab[7]`, il suffit de calculer `18+7 = 25` et de regarder la case mémoire `25`. Si on souhaite accéder à `tab[15]` il suffit de calculer `18 + 15 = 33` et d'accéder à la case mémoire `33` etc. On voit donc qu'il suffit d'une somme pour savoir dans quelle case mémoire se trouve l'élément de tel ou tel indice (on parle parfois d'*accès calculé* aux éléments d'un tableau).

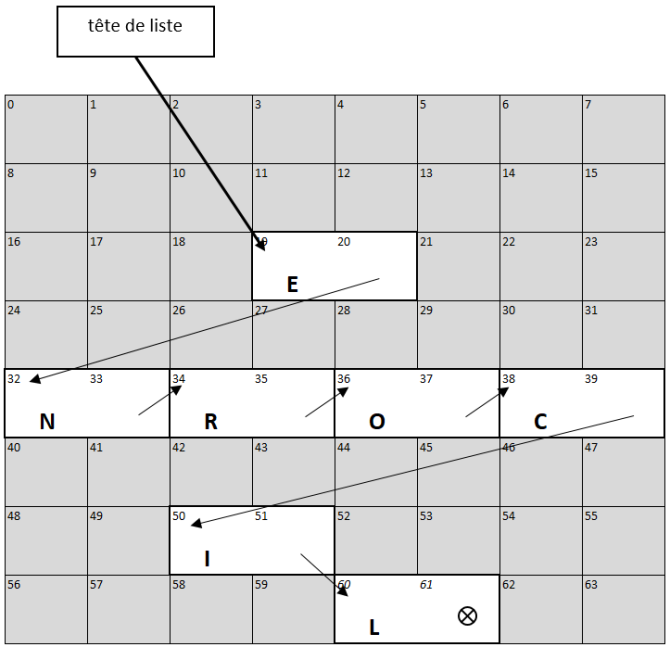


Si elle permet un accès rapide aux éléments à partir de leurs indices, cette organisation de la mémoire n'est en revanche pas souple : l'insertion ou la suppression d'un élément du tableau oblige à décaler d'une case mémoire toutes les données situées plus loin dans le tableau : l'insertion est donc coûteuse.

En résumé un tableau permet un accès calculé rapide à ses éléments mais les insertions ou suppressions d'éléments dans un tableau sont coûteuses.

Organisation en mémoire d'une liste

Les listes sont organisées différemment : les différents éléments sont regroupés dans des *"cellules"* mémoire contenant *deux* informations de nature différente :
- d'une part la valeur que l'on souhaite mémoriser (ci-contre il s'agit de caractères);
- d'autre part l'adresse de la case mémoire de l'élément suivant dans la liste (cette information est souvent représentée par une flèche dans les ouvrages d'informatique).



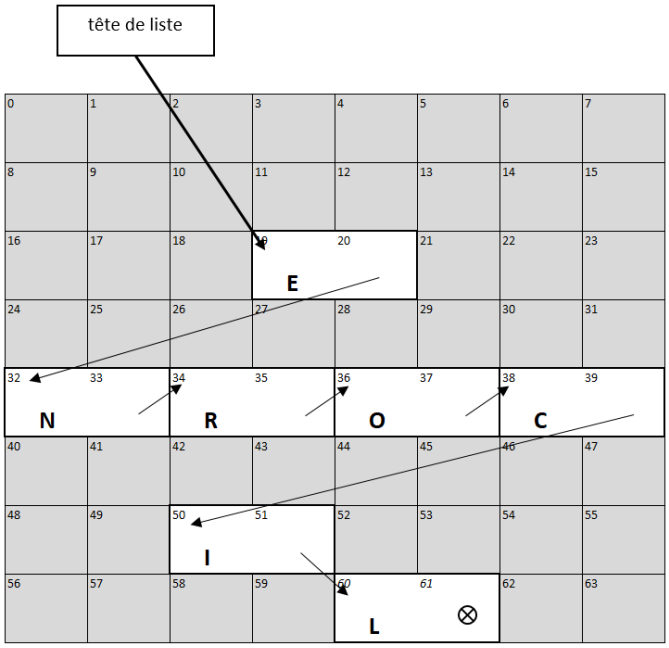
Cela offre un avantage par rapport aux tableaux : si l'on souhaite insérer ou supprimer un élément dans la liste, il suffit essentiellement de rajouter ou supprimer une "cellule" et de modifier les flèches des cellules adjacentes (vous pouvez penser à une chaîne à laquelle on enlève ou rajoute un maillon). Une insertion ou une suppression se fait donc en temps constant : cela ne dépend pas de la taille de la liste.

En revanche cela amène un inconvénient : si vous souhaitez accéder au `k`-ième élément de la liste, vous êtes contraint de commencer par la tête de liste et de suivre le parcours indiqué par les `k` flèches suivantes. L'accès au `k`-ième élément est donc de complexité $O(k)$.

En résumé une liste permet des ajouts ou suppressions d'éléments de façon relativement économique mais l'accès à un élément dont on connait la position dans la liste est quant à lui coûteux.

I . Implémentation d'une liste

I . 1 : Un objet intrinsèquement récursif



Prenons l'exemple ci-contre : un premier point à comprendre est que sur ce schéma il y a … 7 listes plus une :
- la liste contenant `E, N, R, O, C, I et L,`
- la liste contenant `N, R, O, C, I et L,`
- la liste contenant `R, O, C, I et L,`
- la liste contenant `O, C, I et L,`
- la liste contenant `C, I et L,`
- la liste contenant `I et L,`
- la liste contenant `L,`
- plus enfin la liste vide représentée par ⊗.

Par exemple la *liste* contenant les sept lettres ENROCIL est finalement une *cellule* (cases mémoire 19 et 20) contenant :
- une valeur (le caractère E),
- une autre liste (la liste indiquée par la flèche allant de la case 20 à la case 32 et qui contient les six lettres NROCIL).

Il y a un lien très étroit entre liste et cellule. La différence est qu'une liste désigne sa propre cellule ainsi que toutes les cellules suivantes alors qu'une cellule … ce n'est qu'une seule et unique cellule.

Définition :
Une liste `ma_liste` est une structure de données qui est :

- soit la liste vide,
- soit un couple `(t, q)` où :
  - `t` est une donnée à mémoriser de type quelconque, appelée *tête* de `ma_liste`,
  - `q` est une liste (qui contient les données suivantes), appelée *queue* de `ma_liste`.

On voit donc qu'une liste est un objet *récursif* : au même titre que lorsqu'on ouvre une poupée russe on y trouve … une autre poupée russe, lorsqu'on accède à une liste on y trouve une valeur mais aussi … une autre liste.

Compte-tenu de ce qui vient d'être dit, une façon naturelle de représenter une liste en langage Python peut être d'utiliser des 2-uplets emboîtés les uns dans les autres. Par exemple la liste représentée ci-dessus correspondrait à cet objet :

```
( E , ( N , ( R , ( O , ( C , ( I , ( L , None )))))))
```

Cette approche est possible mais il semble plus intéressant de partir de zéro et de construire une structure de liste "à partir de rien" plutôt que "à partir de p-uplets". C'est ce que nous allons faire maintenant.

I . 2 : Implémentation d'une classe `Liste` identifiée à la classe `Cellule`

Cette implémentation est très simple mais, comme indiqué dans le notebook, *elle assimile la notion de liste avec la notion de cellule évoquée plus haut*. Cela a des conséquences : par exemple la liste vide est associée à `None` et non pas à un objet de la classe `Liste`.

```
class Liste:
    def __init__(self, element, succ):
        self._elem = element
        self._succ = succ

    def queue(self):
        return self._succ

    def tete(self):
        return self._elem

    def ajouter(self, element):
        self._succ = Liste(self.tete(), self.queue())
        self._elem = element
```

II . Travailler avec une liste : quelques algorithmes

Parmi les algorithmes travaillés en TP/TD, certains sont à connaître car ils sont par exemple au programme de première (pour les tableaux) . On recopiera dans les encadrés les algorithmes mentionnés :

Calcul de longueur en récursif (à gauche) et en itératif (à droite)

```
def longueur(liste):
    if liste == None:
        return 0
    else:
        return 1 + longueur(liste.queue())
```

```
def longueur_iter(liste):
    lg = 0
    while liste != None:
        lg = lg + 1
        liste = liste.queue()
    return lg
```

Recherche d'un élément en récursif (à gauche) et en itératif (à droite)

```
def recherche(liste, element):
    if liste == None:
        return False
    else:
        if element == liste.tete():
            return True
        else:
            return recherche(liste.queue(),
                                element)
```

```
def recherche_iter(liste, element):
    while liste != None :
        if liste.tete() == element:
            return True
        else:
            liste = liste.queue()
    return False
```

Recherche du maximum en récursif (à gauche) et en itératif (à droite)

```
def maximum(liste):
    if liste.queue() == None:
        return liste.tete()
    else:
        M = maximum(liste.queue())
        if M > liste.tete():
            return M
        else:
            return liste.tete()
```

```
def maximum_iter(liste):
    M = liste.tete()
    while liste.queue() != None:
        liste = liste.queue()
        if liste.tete() > M:
            M = liste.tete()
    return M
```