Problématique étendue sur ce chapitre et le suivant(pile,file) : représenter la liste des processus dans l'ordonnanceur.

1 Problématique

Quand nous créons un tableau, un espace fixé par la taille du tableau est alloué en mémoire.

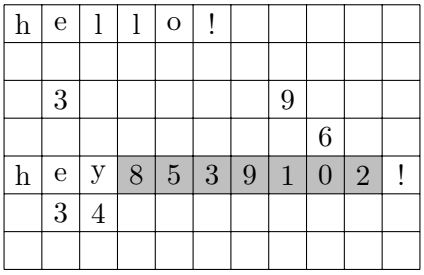

FIGURE 1 – Le tableau est enregistré dans un espace libre

Ce comportement permet d'accéder *en temps constant* à chaque élément du tableau. Cependant insérer un nouvel élément devient problématique : il faut trouver un nouvel espace libre et recopier entièrement le tableau augmenté de la nouvelle valeur.

Peut-on définir un autre type de structure pour représenter les données en mémoire ?

2 Liste chaînée

2.1 Principe

Chaque élément est stocké dans un espace de la mémoire. De plus chaque maillon de la chaîne possède une seconde information : l'adresse du maillon suivant.

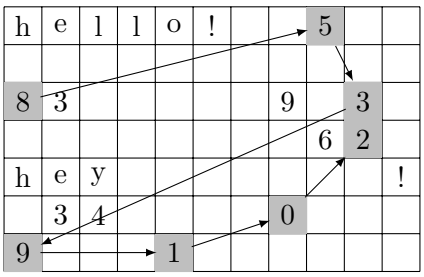

FIGURE 2 – Chaque élément occupe un espace libre

Nous travaillerons sur des listes contenant des entiers.

2.2 Le maillon

Créons un objet *Maillon* qui contiendra la valeur de l'élément et un pointeur vers le maillon suivant.

```
1 class Maillon:
2     """
3     Crée un maillon de la liste chaînée
4     """
```


Christophe Viroulaud

```
5     def __init__(self, val: int, suiv: object)->None:
6         self.valeur = val
7         self.suivant = suiv
```

2.3 La liste

Pour construire une liste il suffit de créer des instances de ce *Maillon* :

```
1 lst = Maillon(3, Maillon(5, Maillon(8, None)))
```

La liste pointe sur le dernier élément ajouté. Le premier élément n'a pas de *suivant*.

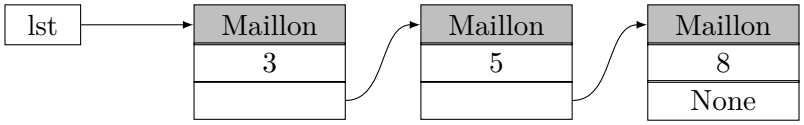

FIGURE 3 – La liste est une succession de maillons

Une seconde approche consiste en la création d'une classe *liste*.

```
1 class Liste:
2     """
3     Crée une liste chaînée
4     """
5     def __init__(self):
6         self.tete: object = None
```

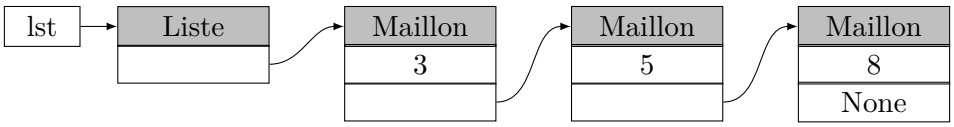

FIGURE 4 – La liste est une succession de maillons

L'attribut *tete* représente le premier *Maillon*. Une liste vide renvoie alors *None*.

Activité 1 :

1. Écrire la méthode **est_vide(self)** → **bool** qui renvoie *True* si la liste est vide, *False* sinon.
2. Écrire la méthode **ajoute(self, val : int)** → **None** qui ajoute un *Maillon* en tête de la liste.

3 Manipuler une liste chaînée

3.1 Longueur de la liste

Pour calculer la taille de la liste il faut obligatoirement la parcourir entièrement. Cette méthode aura donc une complexité en $O(n)$.

Activité 2 :

1. Écrire une méthode **taille(self)** → **int** qui renvoie la taille de la liste. Il sera nécessaire d'écrire une méthode supplémentaire *récursive* **taille_rec(self, maillon : object)** → **int**.

Christophe Viroulaud

méthode (interne) intermédiaire indispensable car on ne peut pas mettre *self.tete* en valeur par défaut pour *maillon*.

2. Il est possible d'effectuer cette opération en programmation impérative. Implémenter alors la méthode **___len___(self)** → **int** qui redéfinit la fonction **len** pour la classe *Liste*.

3.2 N-ième élément

Une fonctionnalité importante qu'on attend d'une liste est de pouvoir renvoyer le n-ième élément.

Activité 3 :

1. Estimer la complexité *dans le pire des cas* de cette opération.
2. En appliquant une méthodologie similaire au paragraphe précédent, écrire la méthode récursive **get_element(self, n : int)** → **int** qui renvoie la valeur du n-ième élément de la liste. Nous considérerons que le premier élément est en *tête* de la liste. La fonction lèvera une *IndexError* si l'indice est négatif ou supérieur à la taille de la liste.
3. Comme pour une *list (au sens Python)* il est possible de récupérer le n-ième élément avec un appel de la forme *lst[n]*. Il faut pour cela redéfinir la méthode **___getitem___(self, n : int)** → **int**. Redéfinir cette méthode en programmation impérative.

Remarque : En toute rigueur, l'élément de rang 0 est en bout de chaîne.

En toute rigueur, la liste est à l'envers. Il faudrait alors $O(n^2)$ pour trouver l'élément de rang n car il faut d'abord calculer la taille de la liste (en $O(n)$)
Des listes doublement chaînées ou circulaires peuvent lever le problème

La structure *list* en Python porte à confusion. Elle allie en fait les avantages des listes chaînées (espace mémoire) et des tableaux (temps d'accès). On peut évoquer la notion de *complexité amortie*.

Christophe Viroulaud