Le type abstrait liste

Table des matières

1) Introduction

1.1) Les structures linéaires

De nombreux algorithmes "classiques" manipulent des structures de données plus complexes que des simples nombres. Nous allons ici voir quelques-unes de ces structures de données. Nous allons commencer par les listes, et deux formes restreintes : les piles et les files. Ces trois types de structures sont qualifiés de linéaires.

1.2) Les autres types de structures

On distingue :
- Les structures par accès par clé : les dictionnaires
- les structures hiérarchiques : les arbres
- les structures relationnelles : les graphes

1.3) Les opérations élémentaires

Une structure de donnée possède un ensemble de routines (procédures ou fonctions) permettant d'ajouter, d'effacer, d'accéder aux données. Cet ensemble de routines est appelé interface.

L'interface est généralement constituée de 4 routines élémentaires dites CRUD :
- **C**reate : ajout d'une donnée
- **R**ead : lecture d'une donnée
- **U**pdate : modification d'une donnée
- **D**elete : suppression d'une donnée

Derrière les opérations de lecture, de modification, ou de suppression d'une donnée se cache une autre routine tout aussi importante : la recherche d'une donnée.

2) Le type abstrait : liste

2.1) Définition

Une liste est une structure de données permettant de regrouper des données. Le langage de programmation Lisp (inventé par John McCarthy en 1958) a été un des premiers langages de programmation à introduire cette notion de liste (Lisp signifie "list processing").

On prendra des listes indicées à partir de 0.

Exemple de liste : L={Buzz ; x ; 1012 ; f9 ; Alan}

Dans cette liste, l'élément 2 est 1012. Cette liste comporte 5 éléments.

2.2) Interface des listes

Voici l'interface d'une liste

- $listeVide$: créer une liste vide.
- $inserer(L, e, i$: ajoute l'élément e à l'index i dans la nouvelle liste L.
- $supprimer(L, i)$ l'élément situé à la position i est supprimé de la liste L
- $modifier(L, e, i)$ l'élément situé à la position i dans la liste L est écrasé par le nouvel élément e
- $longueur(L)$: renvoie le nombre d'éléments dans L.
- $lire(L, i)$: renvoie l'élément d'indice i de L.

Exemple d'utilisation de liste : L={ Buzz ; x ; 1012 ; f9 ; Alan }

- $inserer(L, 25, 0$: nous donne L={ 25 ; Buzz ; x ; 1012 ; f9 ; Alan }
- $supprimer(L, 3)$ nous donne L={ 25 ; Buzz ; x ; f9 ; Alan }.
- $modifier(L, 59, 2)$ nous donne L={ 25 ; Buzz ; 59 ; f9 ; Alan }.
- $longueur(L)$ nous donne 5
- $lire(L, 4)$ nous donne Alan

Exercice 1 :

On donne L={NSI ; plaisir ; jeux ; livres ; π ; E0F1}

Expliciter ce que font chacune des commandes suivantes :

1. $inserer(L, 33, 2)$
2. $supprimer(L, 4)$
3. $modifier(L, programmation, 0)$
4. $longueur(L)$
5. $lire(L, 2)$

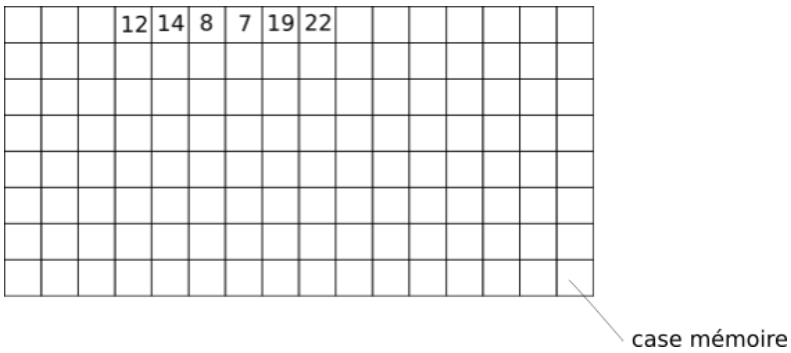
2.3) Implémentation des listes

Pour implémenter les listes (ou les piles et les files), beaucoup de langages de programmation utilisent un mélange de deux structures : les tableaux et les listes chaînées.

a) Les tableaux

- Chaque élément d'un tableau est indicé.
- Si tous les éléments du tableau sont du même type, ils occupent tous la même taille en mémoire, soit t. Il suffit alors de stocker l'adresse du premier élément, soit a et on accède à un élément d'indice k en calculant son adresse en mémoire par $a + k \times t$. Tous les éléments sont donc accessibles avec un coût constant le temps de calcul de l'adresse et l'accès à cette adresse.
- La place du tableau en mémoire est réservé à la création , soit $n \times t$ si n est le nombre d'éléments et t la taille d'un élément.
- Une contrainte est l'impossibilité de remplacer un élément d'un type par un autre élément d'un autre type ou d'agrandir la taille du tableau.

exemple de tableau



b) Les tableaux dynamiques

Dans certains langages de programmation, 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)

À noter que les "listes Python" (listes Python) sont des tableaux dynamiques.

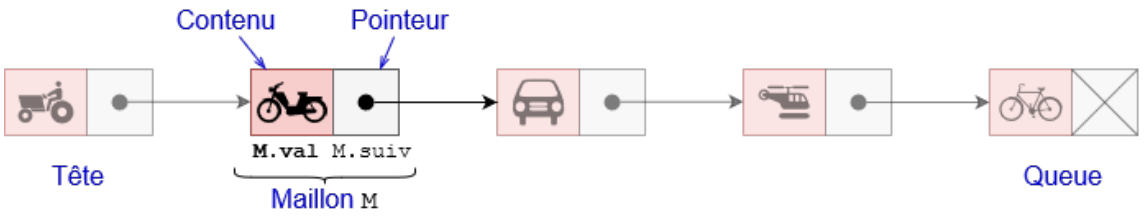
Attention de ne pas confondre avec le type abstrait liste défini ci-dessus, ce sont de "faux amis".

Problème Si on a besoin d'ajouter un élément à un tableau, on crée un nouveau tableau plus grand, on copie les éléments de l'ancien tableau dans le nouveau, on ajoute le nouvel élément à la fin, on remplace l'ancien tableau par le nouveau, et enfin on supprime l'ancien tableau. Ce qui est extrèmement coûteux en nombre d'opération.

c) Les listes chaînées

Une liste chaînée (ou liste liée) est une structure de données composées d'une séquence d'éléments de liste.

Chaque enregistrement d'une liste chaînée est souvent appelé élément, nœud ou maillon.



La tête d'une liste est son premier nœud. La queue d'une liste peut se référer soit au reste de la liste après la tête, soit au dernier nœud de la liste. Le champ de chaque nœud qui contient l'adresse du nœud suivant ou précédent est généralement appelé lien ou pointeur. Le contenu est placé dans un ou plusieurs autres champs appelés données, informations ou valeur.

Chaque élément (ou maillon) M de la liste est composé :

- dun contenu utile `M.val` de nimporte quel type),
- dun pointeur `M.suiv` pointant vers lélément suivant de la séquence.

Le dernier élément de la liste possède un pointeur `M.suiv` vide.

Une liste chainée L est entièrement définie par son maillon de tête `L.tete`, c'est à dire l'adresse de son premier maillon.

Remarque : on peut également lui ajouter un attribut `L.cur` pour mémoriser l'adresse d'un maillon "courant".

On peut implémenter un maillon de liste chaînée en Python à l'aide dune classe Maillon :

```
1    class Maillon:
2    def __init__(self,valeur=None):
3    self.val = valeur
4    self.suiv = None # Pas de maillon suivant
```

Son attribut `suiv` est de type Maillon , ou bien vaut None si le maillon est le dernier de la liste.

Nous pouvons également implémenter une liste chaînée par une classe `ListeC` :

```
1    class ListeC:
2    def __init__(self):
3    self.tete = None # Liste vide
```

Son attribut `tete` est de type Maillon , ou bien vaut `None` si la liste est vide. On peut alors créer une liste ainsi :

```
1    L = ListeC()
2    M1, M2 = Maillon(5), Maillon(8)
3    M1.suiv = M2
4    L.tete = M1
```

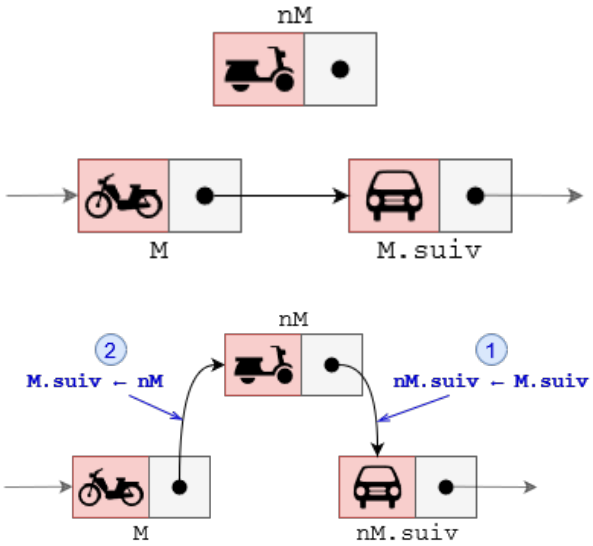
Exercice 2 :

Implémenter en Python les méthodes `estVide()`, `taille()` , `getDernierMaillon()` , et `getMaillonIndice(i)`.

insertion d'un maillon dans une liste

Pour insérer un maillon nM après un maillon M , il faut :

- Faire pointer le pointeur `nM.suiv` vers `M.suiv`
- Faire pointer le pointeur `M.suiv` vers `nM`



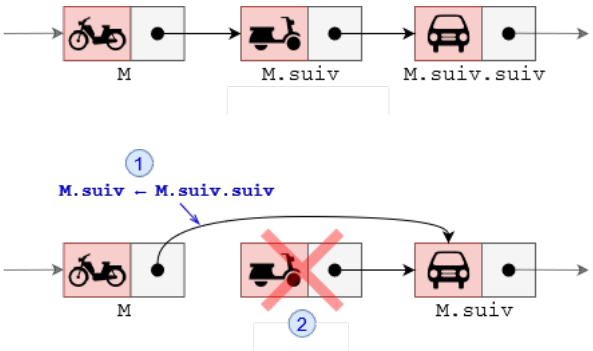
Exercice 3 :

En suivant les schémas des interfaces dinsertion de maillon, implémenter en Python les fonctions `ajouter_debut(M` , `ajouter_fin(M)` et `ajouter_apres(i, M)`

suppression d'un maillon dans une liste

Pour supprimer le maillon suivant un maillon M , il faut :

- Faire pointer le pointeur `M.suiv` vers `M.suiv.suiv`
- Détruire (effacer de la mémoire) le maillon `M.suiv`



Exercice 4 :

En suivant les schémas des interfaces de suppression de maillon, implémenter les méthodes suivantes : `supprimer_debut()` qui supprime le premier élement de la liste, `supprimer_fin()` qui supprime le dernier élément de la liste et enfin `supprimer_apres(i)` qui supprime le maillon de la liste L situé après le maillon d'indice i.

Remarque : Ces dernières méthodes doivent retourner le maillon supprimé.

Exemple : voir la vidéo suivante.