I. Les listes

Cours

A. La notion de liste

!!! info "Définition" Une **liste** est un ensemble séquentiel (les éléments sont les uns à la suite des autres) de données de même type. Elle est classiquement implémentée par :
- un **tableau**,
- une **liste chaînée**.

!!! success "Rappel" Un tableau est une suite indicée d'éléments de même type.

L'avantage d'un tableau est que l'on peut accéder directement (avec une complexité ***constante***) à chacun de ses éléments, grâce à leur indice.
Son inconvénient est que sa **taille est définie à l'avance**, et ne peut être modifiée.

B. La structure de liste chaînée

!!! info "Définition" Une **liste chaînée** est une suite d'éléments composée :
- d'une **tête** : le premier élément auquel on peut accéder,
- d'une **queue** : le reste des éléments.

Historiquement, le langage LISP ("list processing") est un des premiers à introduire cette structure de données en 1958. Dans ce langage, la tête est nommée car pour "content of address register", et la queue cdr pour "content of decrement register".

Sur une liste, on peut effectuer les opérations dites "primitives" suivantes :

- *créer* une liste vide (`listeCree()`),
- *obtenir la tête* de la liste (`listeTete(liste)`),
- *obtenir la queue* de la liste (`listeQueue(liste)`),
- renvoyer si la liste est *vide* (`True`) ou non (`False`) (`listeEstVide()`),
- *ajouter* un élément en tête d'une liste (`listeAjoute(element, liste)`).

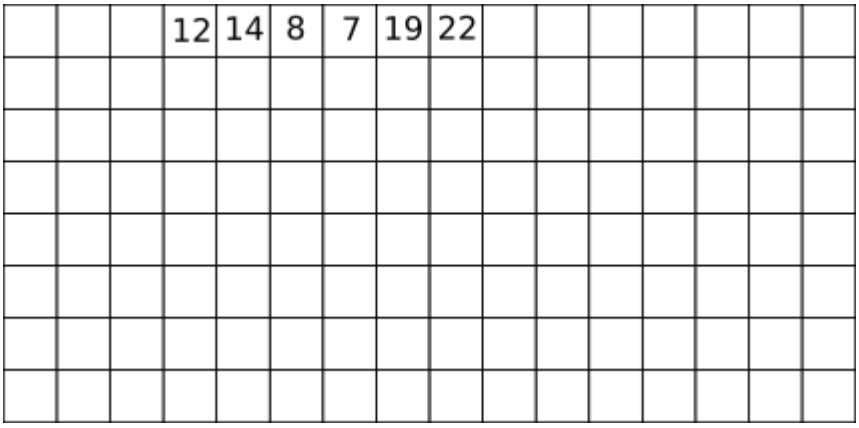
!!! abstract "Exercice" En utilisant les fonctions définie ci-dessus, créer une liste chaînée initialement vide, à laquelle on ajoute successivement les entiers 15, 2 et 3.

````
??? note "Solution"
    ```python
    L = listeCree()
    L = listeAjoute(15, L)
    L = listeAjoute(2, L)
    L = listeAjoute(3, L)
    # ou L = listeAjoute(3,listeAjoute(2,listeAjoute(15, listeCree())))
    ```
````

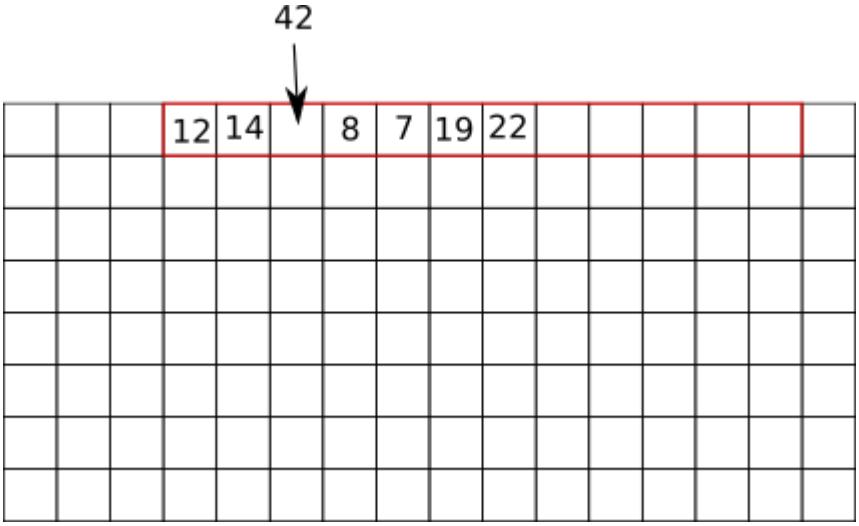
??? note "En quoi une liste chaînée est-elle une structure récursive ?"
Une liste est une structure récursive car sa queue est elle-même une liste.

C. Stockage en mémoire

Tableaux et **listes chaînées** se distinguent par leur utilisation de la mémoire. Un **tableau** est stocké dans une suite contigüe de cases mémoires. Sa taille est fixe : l'ajout d'un nouvel élément, qui agrandirait la taille du tableau, n'est pas possible.

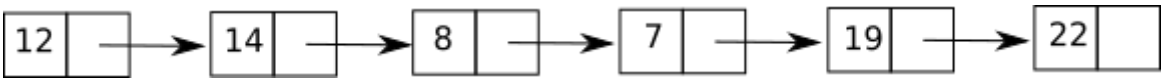


En Python, la structure de liste correspond à un **tableau dynamique** : un tableau auquel on peut ajouter des éléments. Pour insérer un élément au milieu, il faut décaler tous les suivants. Et s'il n'y a pas de place pour ajouter ces données dans les cases mémoires contigües... il faut réécrire le tableau ailleurs.

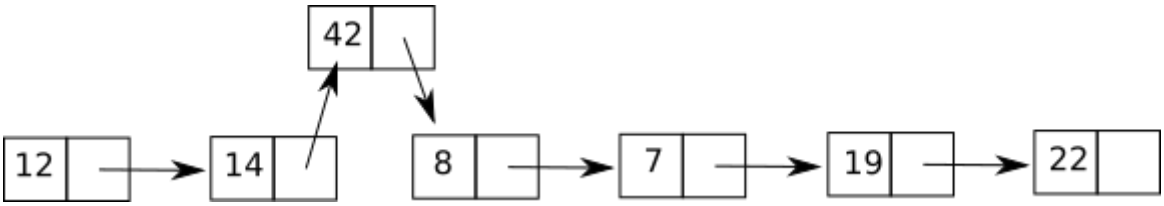


Les différents éléments d'une liste chaînée ne sont pas stockés de manière contigüe : chaque élément est associé à 2 cases mémoires :

- une dans laquelle on stocke **la valeur** de l'élément,
- une dans laquelle on stocke **l'adresse mémoire** de l'élément **suivant**.



Il est donc très facile d'ajouter des éléments à une liste chaînée :



??? note "Quelle opération faut-il faire ?" Il faut lier la queue de l'élément d'avant au nouvel élément, et la queue du nouvel élément à l'élément qui était à sa place.

!!! abstract "Exercice" Ecrire une fonction `elementListe` prenant comme entrées une liste `L` et un indice `i` et retournant l'élement `i` de la liste. Quelle est la complexité de cette fonction ?

````
??? note "Solution"
    ```python
    def elementListe(L, i):
        if i == 1:
            return listeTete(L)
        else:
            return elementListe(listeQueue(L),i-1)
    ```
    Il faut parcourir tous les éléments précédents avant d'accéder à l'élément i :
la complexité est linéaire (en $O(n)$).
````

D. Interface et implémentation d'une structure de données

Nous avons vu en TD que l'on pouvait implémenter de différentes manières la même structure de **liste chaînée**. Le **type abstrait**, ou définition de la structure ne change pas. On manipule la structure grâce à son ***interface*** : la description des opérations qui peuvent être faites sur ces données. La manière de les implémenter peut varier, on aura utilisé la même structure.

TP : Implémentations

A. Première implémentation

On propose une première implémentation d'une liste chaînée, qui est représentée par un tuple `(tete, queue)` :

```
def listeCree():
    return None

def listeEstVide(l):
    return l == None

def listeAjoute(x,l):
    return (x,l)
```

```
def listeTete(l):
    return l[0]

def listeQueue(l):
    return l[1]
```

1. Exécuter les instructions suivantes :
    - Créer une liste vide.
    - Vérifier qu'elle est vide.
    - Ajouter successivement les entiers de 0 à 4.
    - Vérifier qu'elle n'est pas vide.
2. Créer une fonction `listeAffiche(l)` qui affiche successivement les différents éléments de la liste de manière **récursive**.
3. Créer une fonction `listeCompte(l)` qui compte le nombre d'éléments dans une liste de manière ***récursive***.
4. Reprendre la liste de la question 1., et :
    - Ajouter l'entier 5.
    - Compter le nombre d'éléments et les afficher.
    - Supprimer la tête de la liste.
    - Compter à nouveau le nombre d'éléments et les afficher.

B. Deuxième implémentation (POO)

On propose une deuxième implémentation qui utilise la POO. Elle se base sur deux classes : une classe `Cellule` qui représente un élément de la liste, et une classe `Liste` composée d'instances de la classe `Cellule` et implémentant les primitives d'une liste :

```
class Cellule:

    def __init__(self, tete, queue):
        self.car = tete
        self.cdr = queue

class Liste:

    def __init__(self, c):
        self.cellule = c

    def isnil(self):
        return self.cellule is None

    def car(self):
        assert(not(self.cellule is None)), 'Liste vide !'
        return self.cellule.car
```

```
    def cdr(self):
        assert(not(self.cellule is None))
        return self.cellule.cdr

    def cons(self, e):
        return Liste(Cellule(e, self))
```

> **Remarque :** On utilise ici les noms utilisés par le langage Lisp pour implémenter les primitives.

Pour construire une liste, on utilise la méthode `cons` qui ajoute un élément en tête :

```
nil = Liste(None)
L = nil.cons(5).cons(4).cons(3).cons(2).cons(1)
```

1. Représenter les éléments de la liste `L` en utilisant un schéma comme ceux du cours.
2. Identifier le rôle des différentes méthodes et l'ajouter en commentaires.
3. Tester le code suivant :

   ```
   print(L.isnil())
   print(L.car())
   print(L.cdr().car())
   ```

4. Ecrire l'instruction permettant d'afficher le dernier élément de la liste.
5. Créer les fonctions `lengthList(l)` et `displayList(l)` qui, respectivement, compte le nombre d'éléments dans la liste, et affiche chacun d'entre eux. On implémentera ces fonctions de manière ***récursive***.
6. Créer la fonction `removeList(l)`, qui supprime un élément en tête de liste et le retourne.

Pour aller plus loin...

La **liste chaînée** vue en cours est une liste **simplement chaînée** : chaque élément ne possède qu'un lien vers le suivant. Avec une liste **doublement chaînée**, chaque élément possède un lien avec le suivant et avec le précédent. Cela permet les parcours de liste dans les deux sens. On propose les classes ci-dessous pour implémenter cette structure.

Ajouter les méthodes `insereDebut` et `insereFin` à la classe `ListeDoublementChainee`, permettant d'insérer des éléments au début ou à la fin de la liste.

```
class Maillon():
    def __init__(self, e):
        self.element = e
```

```
        self.suivant = None
        self.precedent = None


class ListeDoublementChainee():
    def __init__(self, m):
        self.maillon_debut = m
```