M2103 – Structures de Données

Cours 2
Allocation Dynamique – Listes Chaînées

Plan du Cours

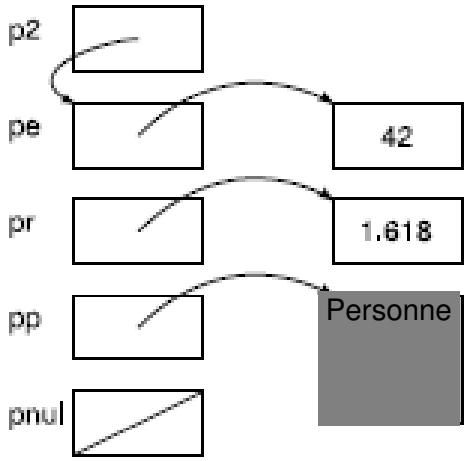
- Allocation Dynamique - Introduction
- Introduction au concept de Liste
  - Implémentation : Classe `TabListe`
  - Problèmes liés à l'utilisation d'un tableau
- Listes Chaînées (Simples)
  - Introduction
  - Ajout
  - Suppression
  - Maillon Tête Factice
- Accès
- Listes Chaînées Doubles
  - Insertion
  - Suppression
- Listes Chaînées Circulaires

Allocation Dynamique

- On connaît les tableaux
  - Structure de données statique
    - Chaque tableau occupe en mémoire la taille maximum envisagée
- Que faire si on ne souhaite pas “perdre de place”?
  - Structure de données dynamique
    - A chaque instant, la place occupée par les données dépend uniquement de la taille de celles-ci (idéalement)
- Mécanisme d'allocation dynamique de la mémoire

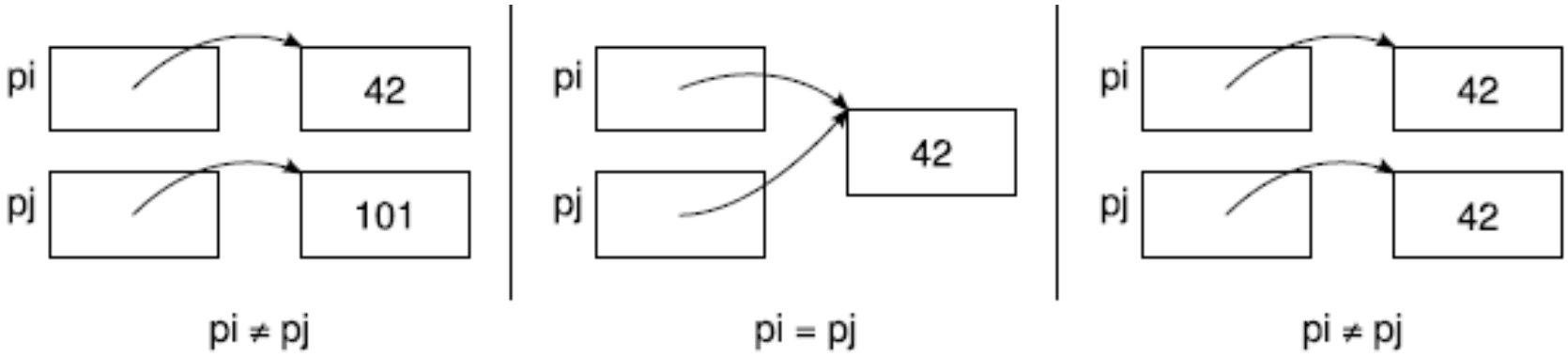
Rappel - Manipulation de variables allouées dynamiquement : le type pointeur

- Variable qui contient l'adresse en memoire d'une autre variable
  - Ne peut pointer que vers des variables d'un seul type
  - Pour déclarer une variable de type pointeur, on donne le type des variables pointées, suivi d'une étoite
    - int* pe //pointeur vers entier
    - float* pr //pointeur vers réel
    - Personne* pp //pointeur vers Personne
    - int** p2 //pointeur vers pointeur
- Un pointeur peut egalement ne pointer vers aucune variable : on l'appelle alors pointeur nul



Opérations sur variables de type pointeur : Comparaison

- Deux pointeurs sont égaux s'ils contiennent la même adresse mémoire
  - i.e. ils pointent vers la même variable
- On notera que deux pointeurs peuvent pointer vers deux variables distinctes mais ayant la même valeur
  - Les pointeurs sont bien différents, puisqu'ils contiennent des adresses mémoire différentes.
  - La valeur des variables pointées n'a aucune influence sur la comparaison des pointeurs



Opérations sur variables de type pointeur : Déréférencement

- Opération permettant, à partir d'un pointeur, d'accéder à la variable pointée (*)
  - Si ***pe*** est un pointeur sur entier, ****pe*** représente l'entier pointé par ***pe***
- Peut être placée à gauche d'une affectation afin de changer la valeur de la variable pointée
  - Example : *pe = 42
  - Le fait d'affecter *pe n'impacte pas le pointeur : il pointe toujours vers la même variable, ce n'est que la valeur de cette variable qui change
  - Modifcation toutefois « visible » depuis d'autres pointeurs pointant vers la même variable
- Attention !
  - Toujours s'assurer lorsqu'on déréférence un pointeur qu'il n'est pas nul
  - Déréférencer un pointeur non initialisé est plus dangereux…

Plan du Cours

- Allocation Dynamique - Introduction
- Introduction au concept de Liste
  - Implémentation : Classe `TabListe`
  - Problèmes liés à l'utilisation d'un tableau
- Listes Chaînées (Simples)
  - Introduction
  - Ajout
  - Suppression
  - Maillon Tête Factice
- Accès
- Listes Chaînées Doubles
  - Insertion
  - Suppression
- Listes Chaînées Circulaires

`Liste` – Définition et Opérations

- Une Liste est une collection d'éléments, chacun avec une position distincte, i.e. les éléments ont un ordre d'apparition

- Opérations
  - Création d'une liste
  - Ajout d'un élément
  - Retourner la position d'un élément donné de la liste
  - Modification d'un élément à une position valide donnée avec récupération de l'ancien élément
  - Suppression d'un élément à une position valide donnée avec récupération de cet élément

Classe `TabListe` : Interface

```
def __init__(self):
 """
 :sortie self:
 :post-cond: tableau déclaré, initialisation nb éléments
 """

def ajout(self,x):
 """
 :entrée-sortie self:
 :entrée x: object
 :pré-cond: le tableau n'est pas plein
 :post-cond: ajout de x à TabListe
 """

def retourner_pos(self,x):
 """
 :entrée self:
 :entrée x: object
 :sortie p: int
 :pré-cond: l'élément x se trouve dans la liste
 :post-cond: p est sa position
 """
```

```
def set(self,id,nouvVal):
 """
 :entrée-sortie self:
 :entrée id: int
 :entrée nouvVal: object
 :sortie vieilleVal: object
 :pré-cond: l'élément à modifier est à la position valide id
 :post-cond: retourne l'ancien élément à la position id
 """

def suppr(self,id):
 """
 :entrée self:
 :entrée id: int
 :sortie supprVal: object
 :pré-cond: l'élément à supprimer est à la position valide id
 :post-cond: retourne l'élément supprimé supprVal
 """
```

Classe `TabListe` : Implémentation

```
import numpy

MAX = 100

class TabListe :

  def __init__ (self) :
    self._tabListe = numpy.empty(MAX, object)
    self._nbElem = 0

  def ajout (self, x) :
    self._tabListe[self._nbElem] = x
    self._nbElem = self._nbElem+1

  def retourner_pos (self, x) :
    I = 0
    while self._tabListe[i] != x :
      I = i+1
    p = i+1
    return p
```

```
def set (self, id, nouvVal) :
  vieilleVal = self._tabListe[id-1]
  self._tabListe[id-1] = nouvVal
  return vieilleVal

def suppr (self, id) :
  supprVal = self._tabListe[id-1]
  for i in range (id-1, self._nbElem-1) :
    self._tabListe[i] = self._tabListe[i+1]
  self._nbElem = self._nbElem-1
  return supprVal
```

Implémentation des opérations de `Liste` avec un tableau

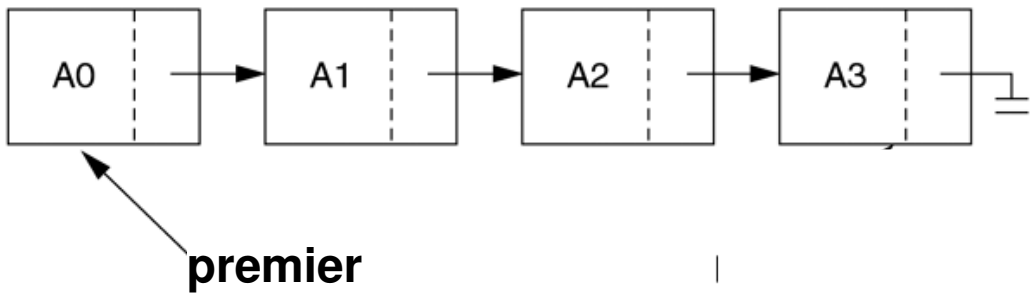
- Problèmes liés à l'utilisation d'un tableau :
  - Suppression (et éventuellement ajout) à l'origine de décalages de données
    - Moitié des éléments en moyenne
  - La taille du tableau arrive à son maximum
    - Nécessaire de déclarer un nouveau tableau (par exemple de taille double)
    - Peut être à l'origine d'un gâchis d'espace mémoire

Plan du Cours

- Allocation Dynamique - Introduction
- Introduction au concept de Liste
  - Implémentation : Classe TabListe
  - Problèmes liés à l'utilisation d'un tableau
- Listes Chaînées (Simples)
  - Introduction
  - Ajout
  - Suppression
  - Maillon Tête Factice
- Accès
- Listes Chaînées Doubles
  - Insertion
  - Suppression
- Listes Chaînées Circulaires

Listes Chaînées : Introduction (1/2)

- Une collection A représentée par une ***liste chaînée*** peut éviter ces problèmes en stockant les éléments sans avoir besoin de mémoire contiguë et en maintenant des liens entre les éléments par ordre de position
- Structure de données dont la taille croît après une insertion et décroît à la suite d'une suppression
- Une ***liste chaînée*** consiste en une suite de maillons
- Chaque maillon contient
  - une valeur stockée
  - un lien référençant le maillon suivant de la liste
- Un lien externe référençant le ***premier*** élément de la liste doit être mis en oeuvre pour accéder à la liste



Implémentation – Classes Internes

- Possibilité de définir une classe `Maillon` à l'intérieur d'une classe `ListeChainee` :

```
class ListeChainee :

   class Maillon:
      def __init__(self, val):
         self._valeur = val
         self._suiv = None

   def __init__( self ) :
      self._premier = None
```

- L'instance de la classe interne 'appartient' à l'objet qui l'instancie
- L'objet du type de la classe englobante a accès à tous les éléments (privés et bien sûr publics) de l'objet de la classe interne (et réciproquement).
- Instanciation d'objets à partir de la classe interne
  - **`nouvMaillon = ListeChainee.Maillon( valeur )`**

Ajout

- Nous considérons l'insertion d'un élément x **après** le maillon référencé par le lien *courant*

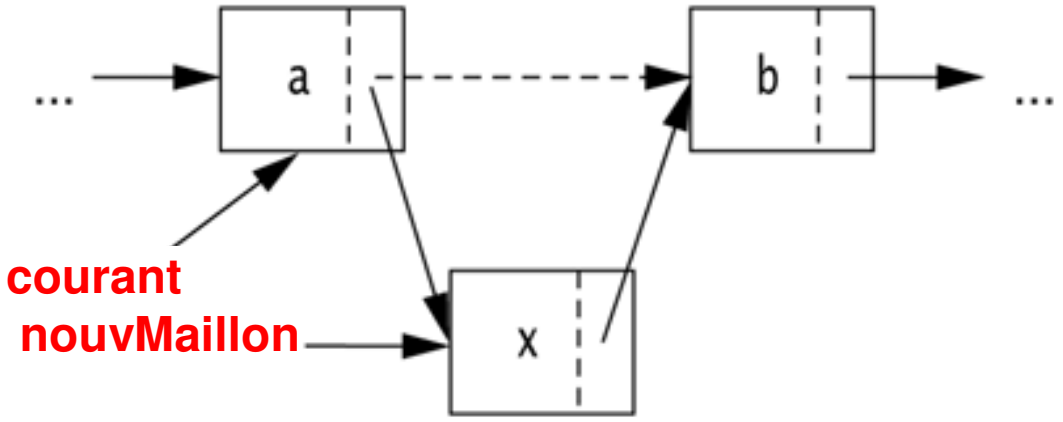
---

nouvMaillon = ListeChainee.Maillon (x) // Création d'un maillon et placement de x

nouvMaillon._suiv = courant._suiv // Le suivant de courant devient le suivant de nouvMaillon

courant._suiv = nouvMaillon // nouvMaillon est placé après le maillon courant

---



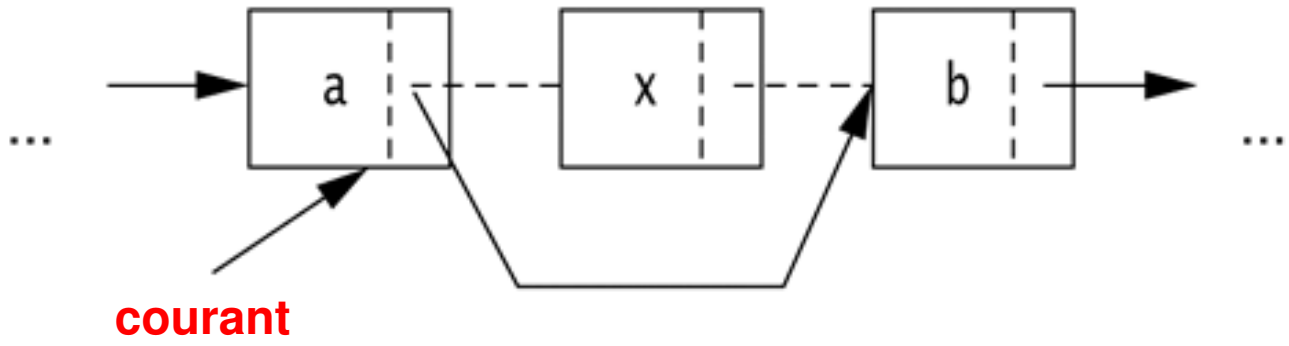
- Pour accéder aux éléments dans la liste, nous utilisons une référence sur le maillon considéré au lieu d'un indice

Suppression

- Suppression de l'élément situé après le maillon référencé par le lien ***courant***

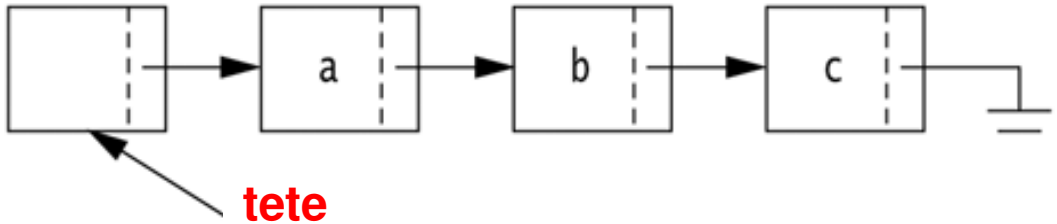
courant._suiv = courant._suiv._suiv

// le maillon situé après le maillon courant est contourné



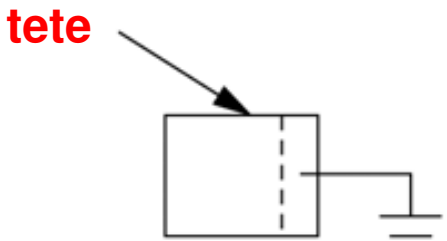
Pour aller plus loin : Maillon Tête Factice

- Insertion/Suppression d'un nouveau premier élément nécessitent de considérer des cas particuliers en l'état
- Au lieu d'implémenter ces cas particuliers
  - Introduction d'un maillon ***tête*** factice sans donnée en tant que premier maillon
    - Tous les maillons avec des données ont donc un maillon précédent



- Une liste chaînée vide n'a donc que le maillon vide initialisé

tete._suiv = None



Illustration

Ecrire les méthodes d'initialisation et d'ajout de la classe `LCListe` (implémentation d'une Liste à l'aide d'une liste chaînée)

- en considérant une liste chaînée simple
- en considérant une liste chaînée avec tête factice
