Université Claude Bernard Lyon 1

DIU EIL – UE 4
Liste chaînée

Nicolas Pronost

LIRIS

Représentation d'une liste par un tableau
(éléments contigus en mémoire)

Liste :

- premier : indice du début de la liste
- reste : indice du début du reste de la liste
- dernier : indice où le prochain élément sera ajouté
  - nombre d'éléments de la liste

- Opérations sur les listes :

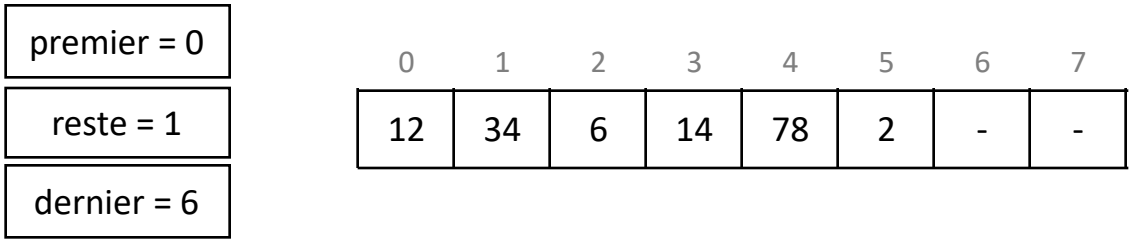
  * Choisir comment représenter une liste vide :
    Par exemple : premier = -1 (et/ou dernier = 0)
    → conséquence sur le test de liste vide

- Ajouter un élément en tête de liste : coûteux à cause du décalage vers la droite de tous les éléments

- Supprimer un élément : coûteux aussi à cause des décalages vers la gauche après le retrait de suppression

Gestion par un tableau simple

- On peut représenter une liste par un tableau et trois données



- Opérations sur les listes
  - Création : premier=-1 (et/ou dernier=0)
  - Test si vide : renvoie vrai si premier==-1 (ou dernier==0)
  - Premier élément : renvoie la valeur en position premier (0 si ≠-1)
  - Ajouter en premier (tête) : tous les éléments sont décalés à droite d'une position, dernier=dernier+1, et le nouvel élément est mis en position 0
  - Supprimer un élément : repérer l'élément (en commençant sur le premier), puis décaler à gauche à partir de la position derrière l'élément, et finalement dernier=dernier-1

* Si on représente une liste par un tableau simple, on a juste besoin du repère dernier, premier sera toujours à 0 et reste à l'. (ou le nombre d'éléments dans le tableau)

* Attention, on parle du type abstrait liste, le type list en Python représente un tableau dynamique (de taille variable)

* Avantage de la représentation par tableau ⟹ accès à un élément en coût constant

* Inconvénient : tous les décalages lors des ajouts (en tête de liste dans le pire des cas) et les suppressions.

Gestion par un tableau simple

- **Avantages** :
  - le premier élément est toujours en 0, le reste toujours en 1, il faut simplement repérer la première place libre (dernier)
    - on peut utiliser une « liste » Python (i.e. un tableau dynamique)
  - on bénéficie des avantages des tableaux, ex. accès en O(1)
- **Inconvénient** : on fait beaucoup de décalages (recopies d'éléments) qui prennent du temps (en O(n))
- Comment éviter de décaler (recopier) les éléments?
  - il faut laisser un « trou » lorsqu'on supprime un élément
  - mais comment gérer l'existence de « trous » dans un tableau?
    - ex: gérer à la main les places « occupées » et les places « libres » mais assez difficile à maintenir et pas optimal en espace mémoire

Comment repérer les "trous" et les "autres"

Les listes chaînées

- Une liste chaînée représente un ensemble d'éléments rangés linéairement
  - mais pas forcément contigus en mémoire
- Chaque élément, appelé une cellule, contient un ensemble d'informations dont un ou plusieurs liens vers d'autres éléments de la liste
- Les liens qui désignent des éléments particuliers (ex. premier et/ou dernier) sont stockés à part
- Les liens ne désignant aucun autre élément ont un code spécial : la valeur **None**

Les listes chaînées

- Il existe plusieurs listes chaînées dont les 4 classiques
  - La liste simplement chaînée
  - La liste simplement chaînée circulaire
  - La liste doublement chaînée
  - La liste doublement chaînée circulaire

Liste simplement chaînée

- Dans une liste simplement chaînée
  - la première cellule est repérée
  - chaque cellule contient l'information de l'élément et le lien vers la cellule suivante dans la liste
- Implémentation en Python

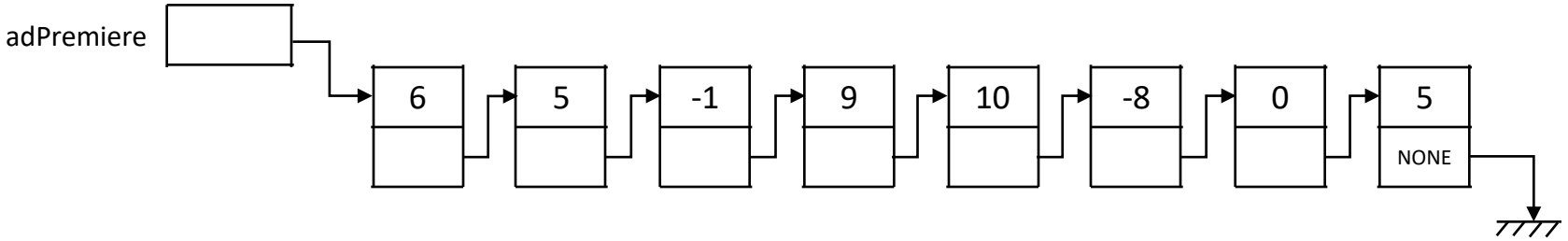
```
class Cellule :
    def __init__(self, info, suivant) :
        self.info = info
        self.suivant = suivant
```

```
class Liste :
    def __init__(self) :
        self.adPremiere = None;

    # ...
```

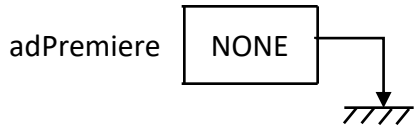
Liste simplement chaînée

- La liste 6,5,-1,9,10,-8,0,5 est représentée graphiquement:

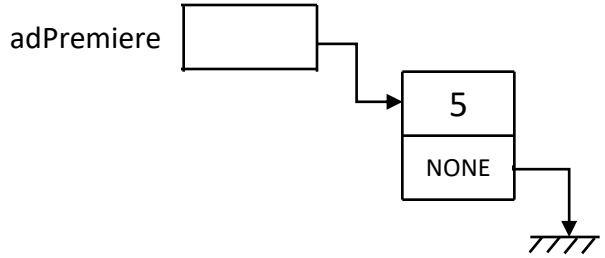


Evolution d'une liste

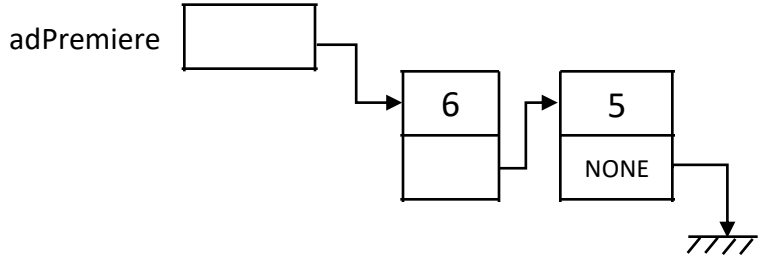
Liste vide



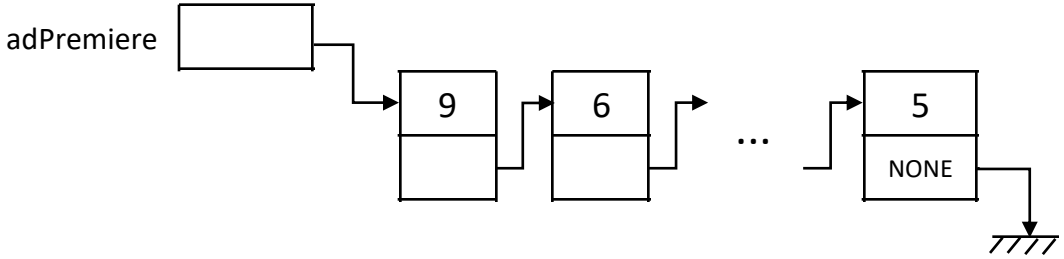
Liste contenant 1 cellule



Liste contenant 2 cellules



Liste contenant plus de 2 cellules

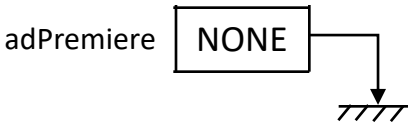


Opérations sur liste chaînée

- Chaînage d'une cellule
  - en tête de liste
  - en queue de liste
  - à un indice donné
- Parcours des cellules d'une liste
  - affichage des éléments
  - recherche d'un élément
  - lecture ou modification d'un élément
  - réorganisation des éléments (ex. tri)
- Décrochage et libération d'une cellule de la liste

Création d'une liste chaînée

- On a vu que l'initialisation d'une liste doit produire l'état de la mémoire suivant :



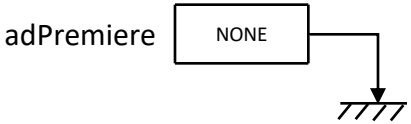
- Le constructeur de la classe Liste est donc simplement

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

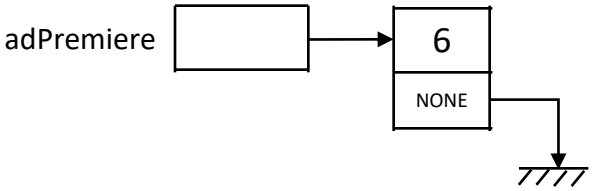
Ajout en tête de liste

On ajoute dans l'ordre inverse des éléments dans la liste

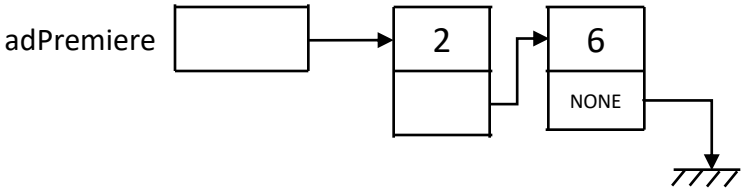
- On veut créer une liste d'entiers dans l'ordre suivant : 5 2 6
  - Etape 1 : création



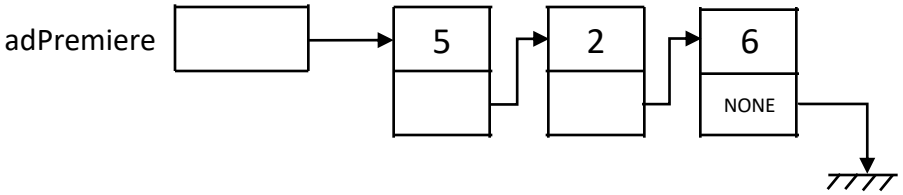
  - Etape 2 : ajout en tête de 6



  - Etape 3 : ajout en tête de 2



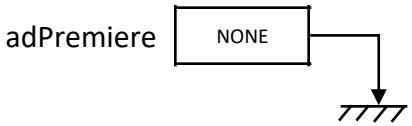
  - Etape 4 : ajout en tête de 5



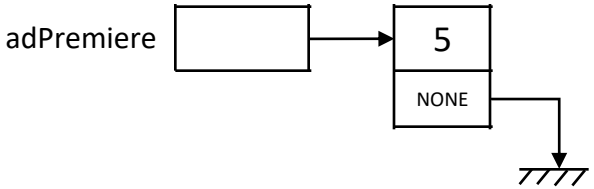
Ajout en queue de liste

On rajoute dans l'ordre de la séquence

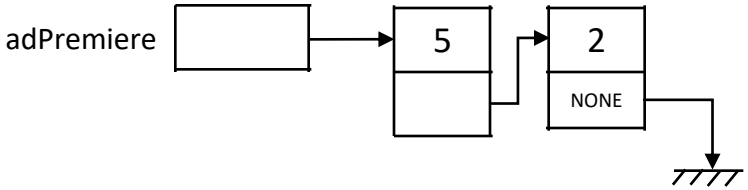
- Ou bien
  - Etape 1 : création



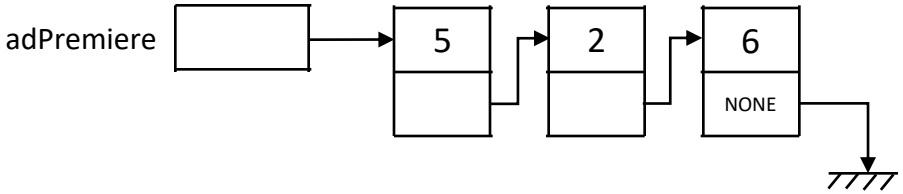
  - Etape 2 : ajout en queue de 5



  - Etape 3 : ajout en queue de 2



  - Etape 4 : ajout en queue de 6



Ajouts en tête et en queue

Ajouter en queue nécessite de parcourir en entier la liste

- on doit trouver l'élément en queue actuellement pour pouvoir ajouter le nouvel élément derrière
- mais dans une liste on a accès uniquement à adPremiere, l'adresse du premier élément de la liste
- il faut donc parcourir toute la liste pour ajouter en queue
- ajouter en queue est donc de complexité linéaire $O(n)$

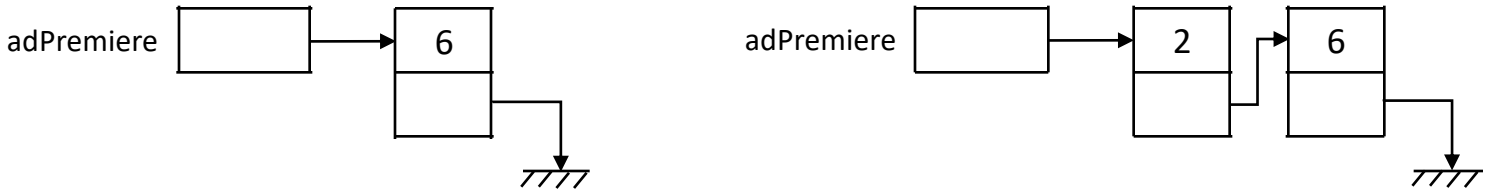
- Par contre, l'ajout en tête est fait indépendamment du nombre d'éléments dans la liste
  - il suffit de remplacer adPremiere par le nouvel élément et de désigner l'ancienne tête comme suivant de la nouvelle
  - ajouter en tête est de complexité constante $O(1)$

Chaînage des cellules

- Que ça soit pour l'ajout en tête ou l'ajout en queue, il faut
  - créer une nouvelle cellule
  - affecter son attribut « info » à la valeur de l'élément
  - affecter son attribut « suivant » à la cellule suivante dans la liste
    - à None si on ajoute en queue
    - à adPremiere si on ajoute en tête
    - au suivant de l'élément précédent sinon
  - éventuellement mettre à jour d'autres liens
    - adPremiere si ajout en tête
    - le suivant de l'élément précédent sinon

Ajout en tête de liste

- Pour passer de cette liste à celle là



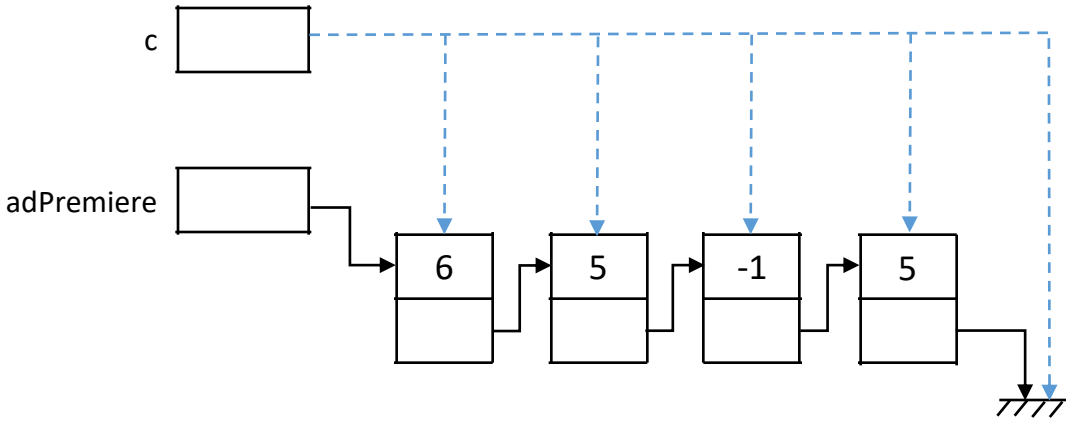
- il faut exécuter l'instruction suivante

```
self.adPremiere = Cellule(2,self.adPremiere)
```

- ce qui fonctionne aussi quand la liste est vide (ajout du premier élément) car adPremiere vaut d'abord None, est affecté à l'attribut « suivant » de la nouvelle cellule qui devient la première et dernière cellule

Parcours des éléments d'une liste

- Beaucoup d'algorithmes nécessitent de parcourir tous les éléments d'une liste une et une seule fois
- On utilise une variable temporaire qui débute sur adPremiere et qui utilise les attributs « suivant » jusqu'à trouver None



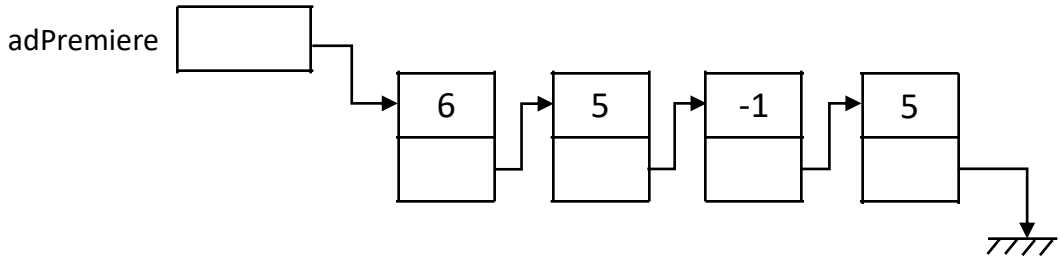
Parcours des éléments d'une liste

- Le code suivant peut donc être utilisé pour itérer sur tous les éléments d'une liste chaînée l, du premier (la tête, le plus à gauche) au dernier (la queue, le plus à droite)

```
c = l.adPremiere
while c : # identique à while c != None
   # faire ce que l'on veut faire sur l'élément c
   # par exemple : print(c.info)
   c = c.suivant
```

Décrochage et libération d'une cellule

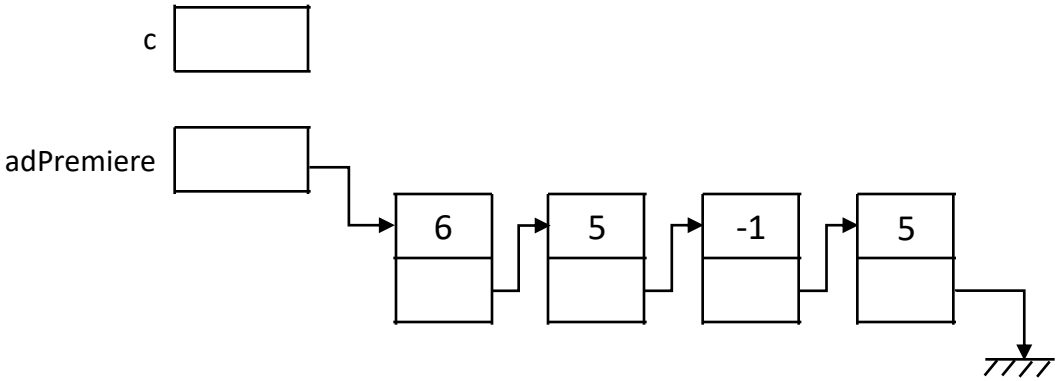
- Comment libérer proprement les cellules d'une liste?



- C'est-à-dire libérer chaque cellule, sans en oublier, et obtenir une liste vide

Décrochage et libération d'une cellule

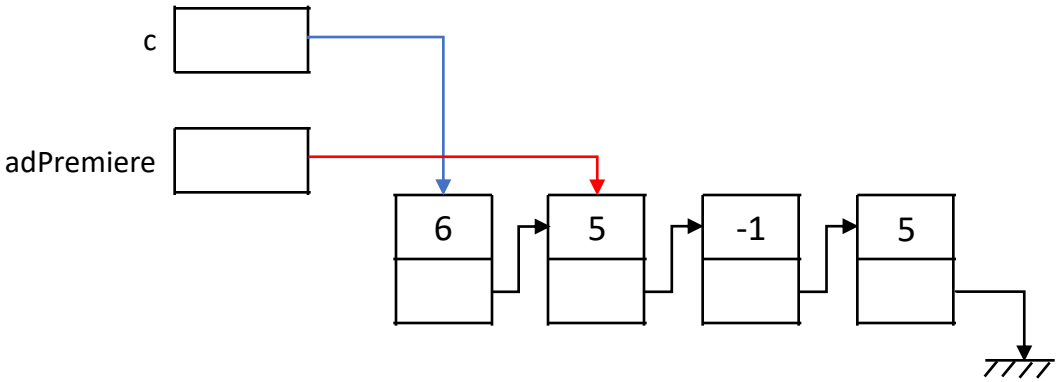
1. Utiliser une variable de travail c



Décrochage et libération d'une cellule

1. Utiliser une variable de travail c
2. Isoler, à l'aide de c, la cellule de tête sans perdre le reste de la liste

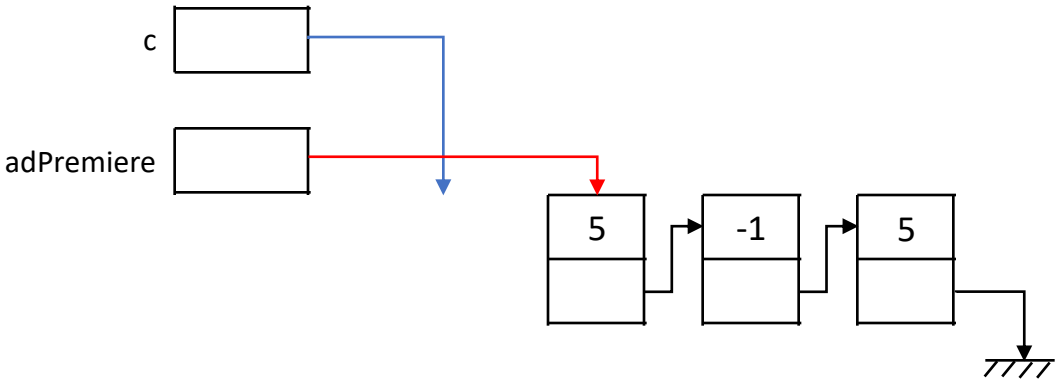
```
c = l.adPremiere;
l.adPremiere = c.suivant;
```



Décrochage et libération d'une cellule

1. Utiliser une variable de travail c
2. Isoler, à l'aide de c, la cellule de tête sans perdre le reste de la liste
3. Libérer la cellule ainsi isolée

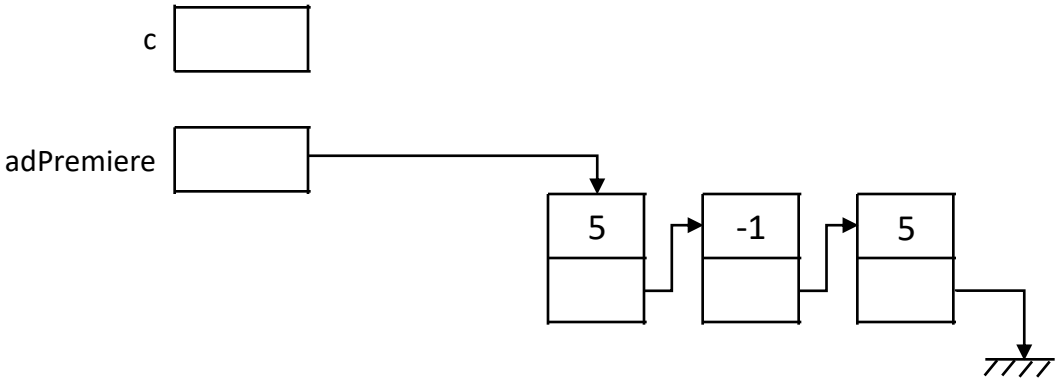
```
del c
```



Décrochage et libération d'une cellule

1. Utiliser une variable de travail c
2. Isoler, à l'aide de c, la cellule de tête sans perdre le reste de la liste
3. Libérer la cellule ainsi isolée

➢ Recommencer les étapes 2 et 3 jusqu'à ce que la liste soit vide



Décrochage et libération d'une cellule

- On obtient l'algorithme de libération suivant (destructeur et procédure vider)

```
while self.adPremiere :
   c = self.adPremiere
   self.adPremiere = c.suivant
   del c
```

Fonctionnalités de la classe Liste

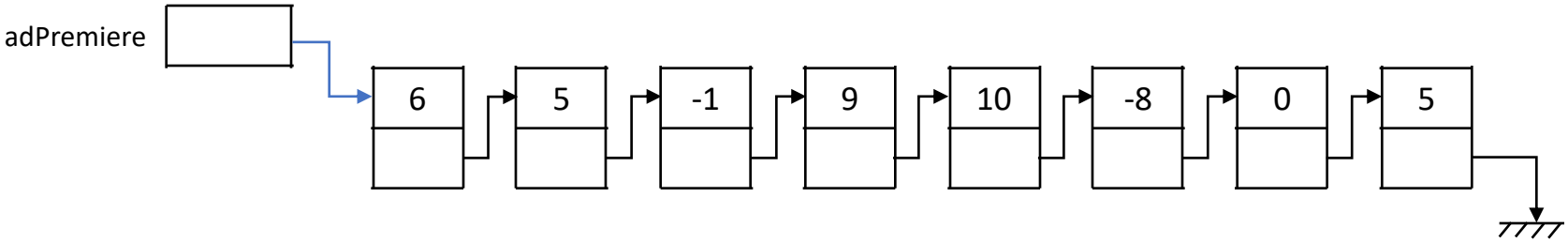
- Constructeur Liste()
  - Postconditions : la liste est une liste vide
- Destructeur ~Liste()
  - Postconditions : libération de la mémoire utilisée, la liste est une liste vide
- Procédure ajouterEnTete (e)
  - Postcondition : l'élément e est ajouté en tête de liste
- Procédure ajouterEnQueue (e)
  - Postcondition : l'élément e est ajouté en queue de liste
- Procédure vider ()
  - Postcondition : la liste ne contient plus aucune cellule
- Fonction estVide () : booléen
  - Résultat : retourne vrai si la liste est vide, faux sinon

Fonctionnalités de la classe Liste

- Fonction iemeElement (indice) : tout type
  - Précondition : 0 <= indice < nombre d'éléments
  - Résultat : retourne l'élément à l'indice passé en paramètre
- Procédure modifierIemeElement (e, indice)
  - Précondition : 0 <= indice < nombre d'éléments
  - Postcondition : l'élément à l'indice passé en paramètre vaut e
- Procédure afficher ()
  - Postcondition : Les éléments de la liste sont affichés à l'écran
- Procédure supprimerTete ()
  - Postcondition : l'élément en tête de liste est supprimé
- Procédure insererElement (e, indice)
  - Précondition : 0 <= indice <= nombre d'éléments
  - Postcondition : e est inséré de sorte qu'il occupe la position d'indice en paramètre
- Fonction rechercheElement (e) : entier
  - Résultat : retourne l'indice de l'élément e dans la liste, ou -1 si l'élément n'est pas présent

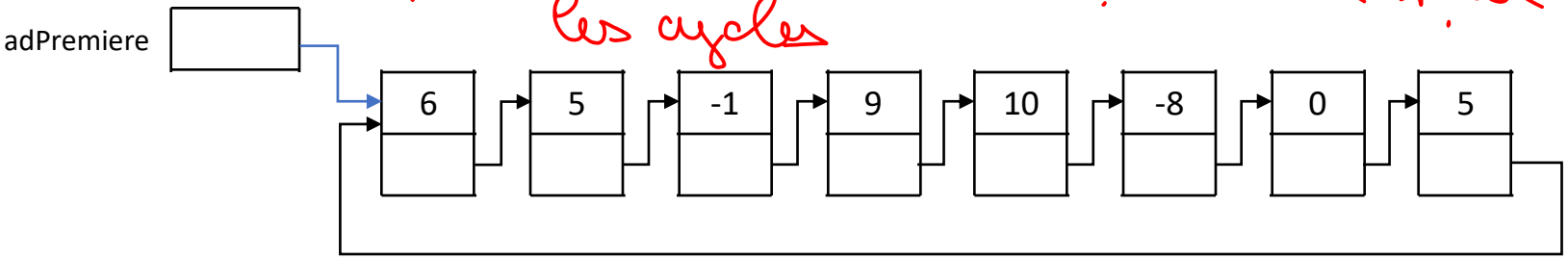
Les listes chaînées

- La liste simplement chaînée



- La liste simplement chaînée circulaire

bien pour les permutations circulaires et pour les cycles

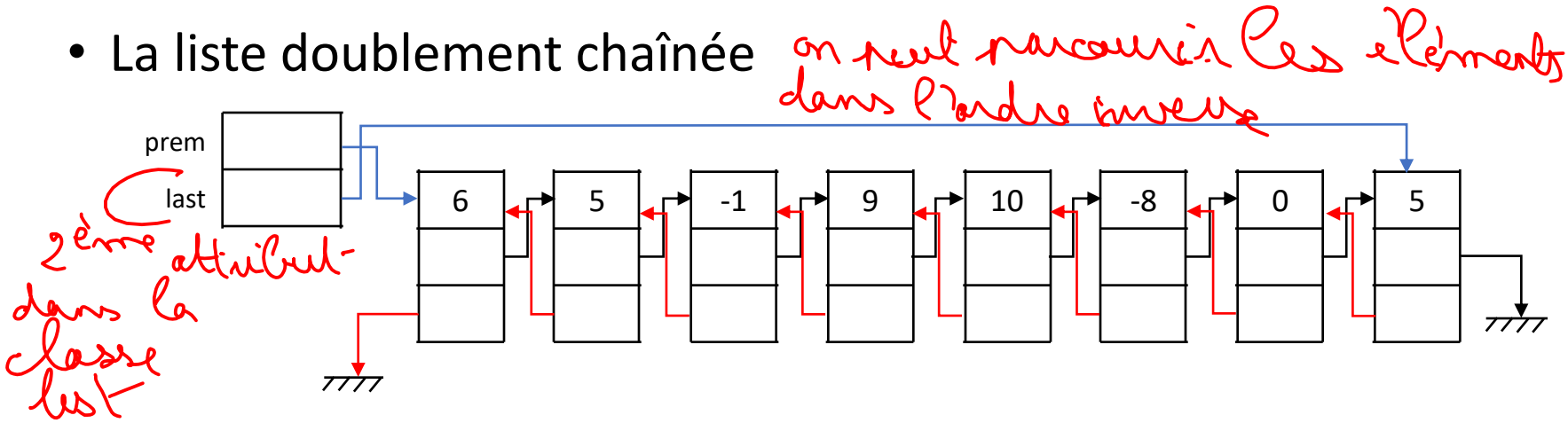


jamais None dans suivant

On détecte qu'on est revenu au début lorsqu'on retrouve le premier

Les listes chaînées

- La liste doublement chaînée



- La liste doublement chaînée circulaire