



# LIFAPSD – Algorithmique, Programmation et Structures de données

Nicolas Pronost



# Chapitre 7

## Liste

# Définition d'une liste

- Une liste est une structure de données à accès séquentiel

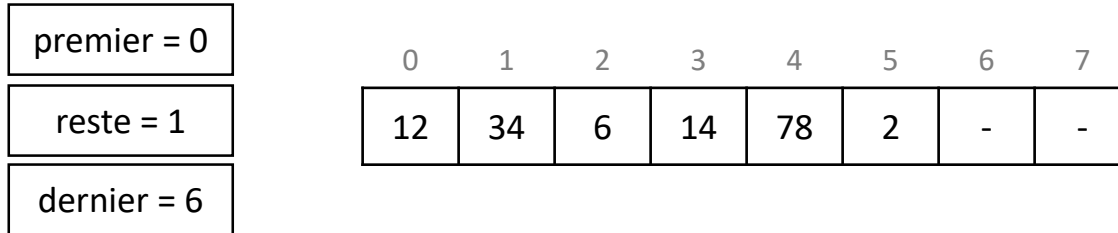
liste = < > (*liste vide*)

liste = < premier (*élément*) | reste (*liste*) >

- Opérations possibles sur une liste
  - créer et détruire
  - vider
  - récupérer le premier élément, le dernier ou un élément quelconque
  - ajouter un nouvel élément en tant que premier élément, dernier élément, ou insérer l'élément dans la liste
  - supprimer un élément
  - etc.

# Gestion par un tableau simple

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



- Opérations sur les listes
  - Initialiser : 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  $\neq -1$ )
  - 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

# 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 un TableauDynamique
- 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

# Les listes chaînées

- Une liste chaînée représente un ensemble d'éléments rangés linéairement
  - mais pas forcément contigus en mémoire
- Contrairement au tableau, l'accès se fait par des **pointeurs**
- Chaque élément, appelé une **cellule**, contient un ensemble d'informations dont un ou plusieurs liens vers d'autres éléments de la liste
- Les liens (pointeurs) 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** : le pointeur NULL

# Les listes chaînées

- Il existe plusieurs listes chaînées dont les 4 classiques
  - La liste simplement chaînée (CM et TD)
  - La liste simplement chaînée circulaire
  - La liste doublement chaînée (TP)
  - 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 C++

```
class Liste {  
    Cellule * adPremiere;  
    // ...  
};
```

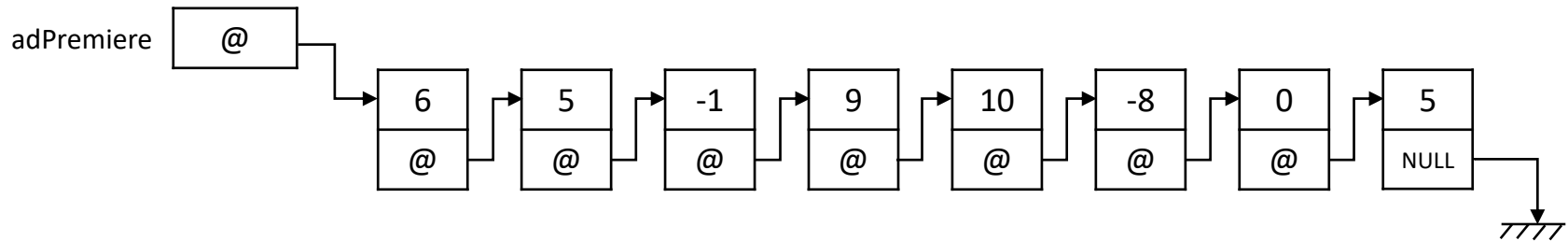
```
struct Cellule {  
    ElementL info;  
    Cellule * suivant;  
};
```

```
typedef int ElementL;
```



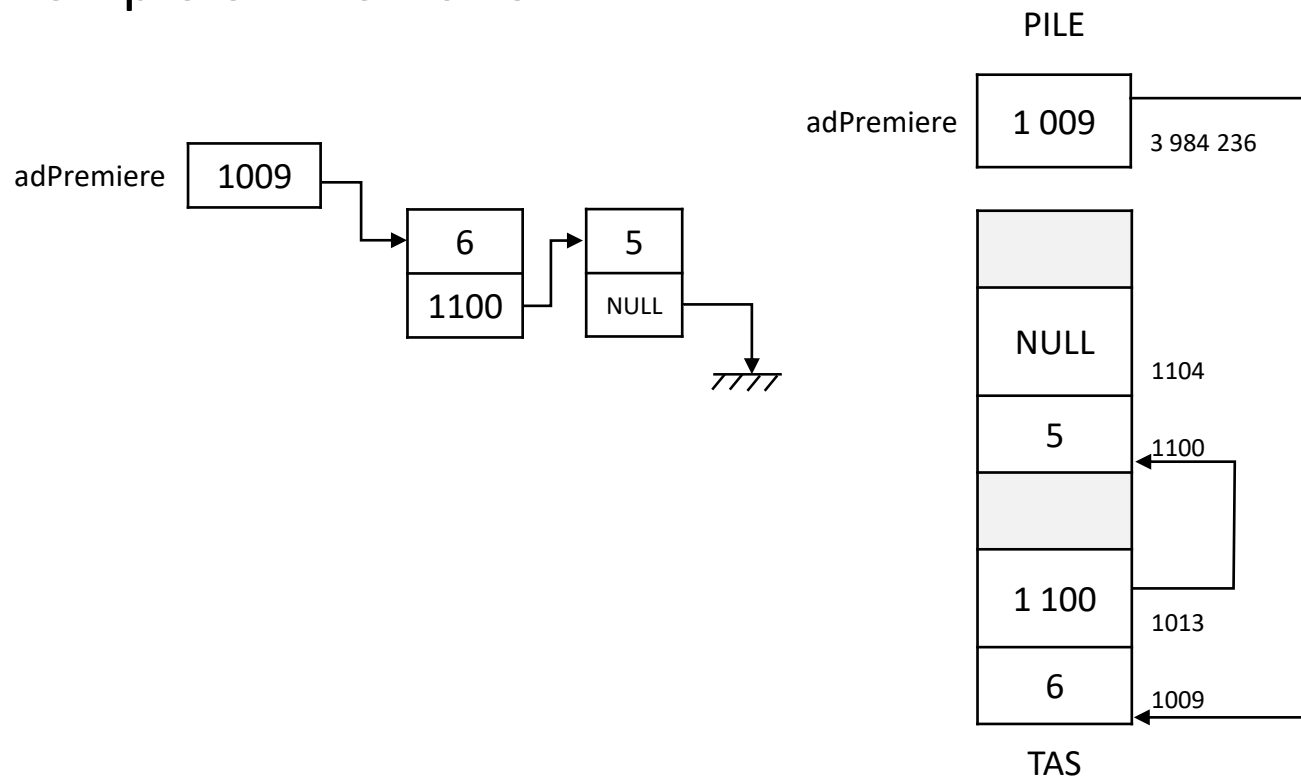
# Liste simplement chaînée

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



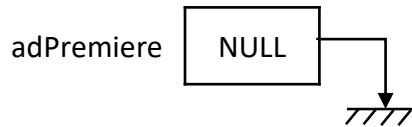
# Liste en mémoire

- Les cellules sont allouées dynamiquement sur le tas
- L'objectif est de créer et de libérer les cellules selon les besoins (pas d'emplacement vide, pas de « trou »)
- Exemple en mémoire

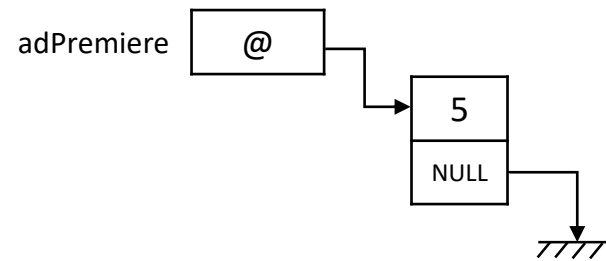


# Evolution d'une liste

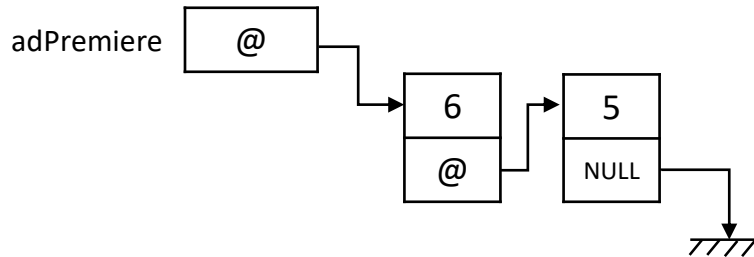
## Liste vide



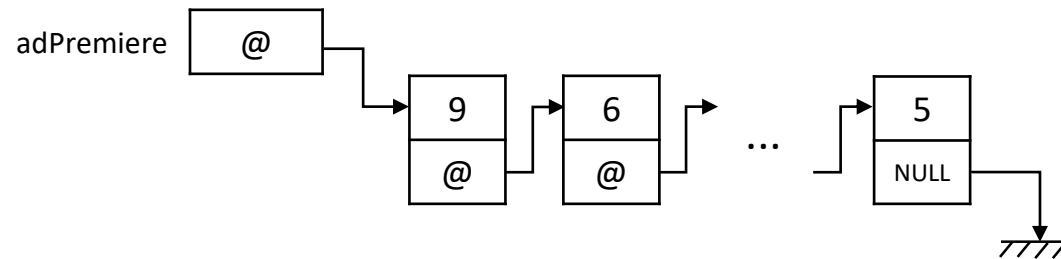
## Liste contenant 1 cellule



## Liste contenant 2 cellules



## Liste contenant plus de 2 cellules

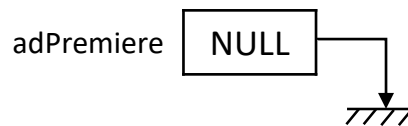


# Opérations sur liste chaînée

- Chaînage d'une cellule créée sur le tas
  - 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

```
Liste::Liste () {  
    adPremiere = NULL;  
}
```

Liste.cpp

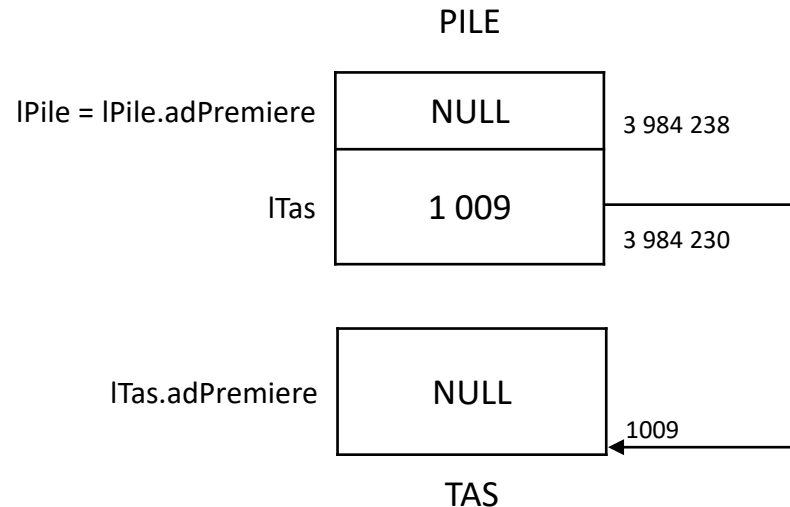
- Ce n'est pas comme le tableau dynamique, il n'y a pas d'emplacement « pré-réservé », on alloue uniquement exactement ce dont on a besoin
- Pas d'espace perdu 🍌 (mais il a fallu ajouter de l'espace mémoire pour les champs « suivant »)

# Création d'une liste chaînée

- L'appel au constructeur se fait comme avec n'importe quel type (primitif ou abstrait)

```
Liste lPile;  
Liste * lTas = new Liste;  
//...  
delete lTas;
```

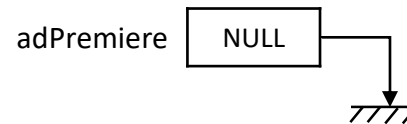
main.cpp



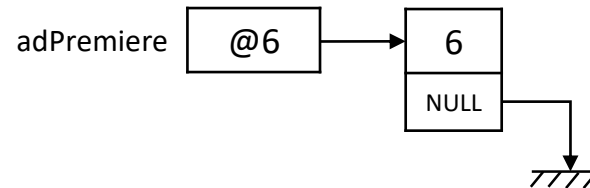
# Ajout en tête de liste

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

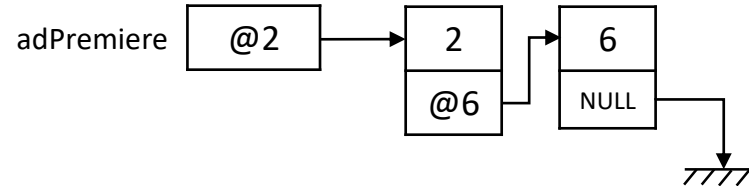
- Etape 1 : initialisation



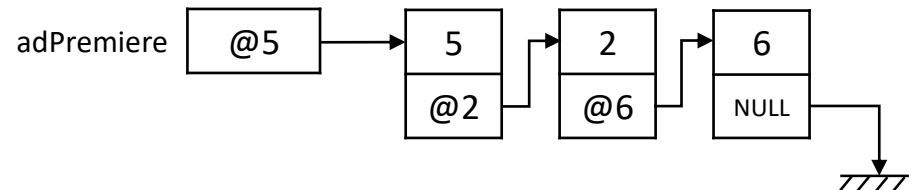
- Etape 2 : ajout en tête de 6



- Etape 3 : ajout en tête de 2

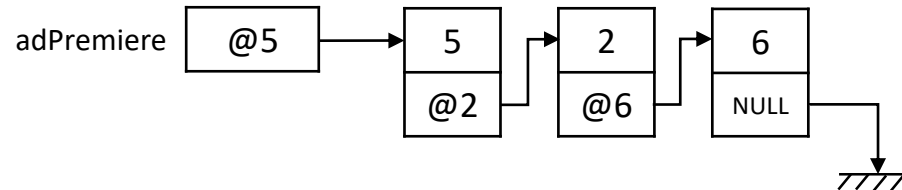
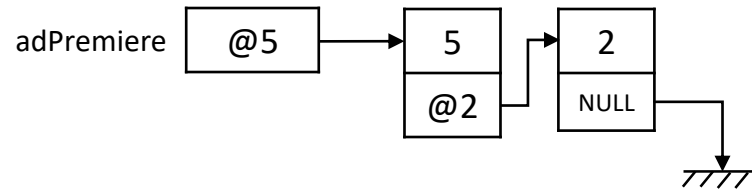
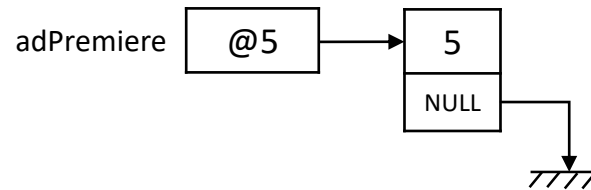
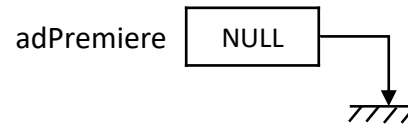


- Etape 4 : ajout en tête de 5



# Ajout en queue de liste

- Ou bien
  - Etape 1 : initialisation
  - 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



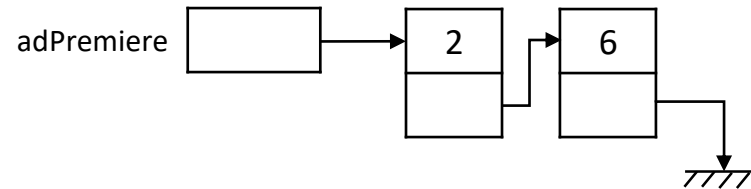
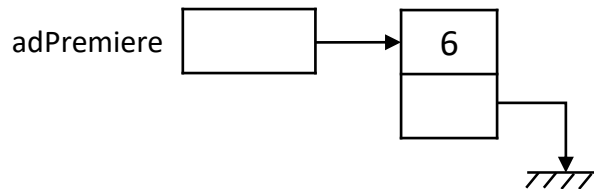
- Ajouter en queue nécessite de parcourir en entier la liste
  - on doit trouver l'élément en queue actuellement pour pouvoir ajouter le nouvel élément derrière
  - mais dans une liste on a accès uniquement à `adPremiere`, l'adresse du premier élément de la liste
  - il faut donc parcourir toute la liste 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 sur le tas
  - affecter son champ « info » à la valeur de l'élément
  - affecter son champ « suivant » à l'adresse de la cellule suivante dans la liste
    - à NULL 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



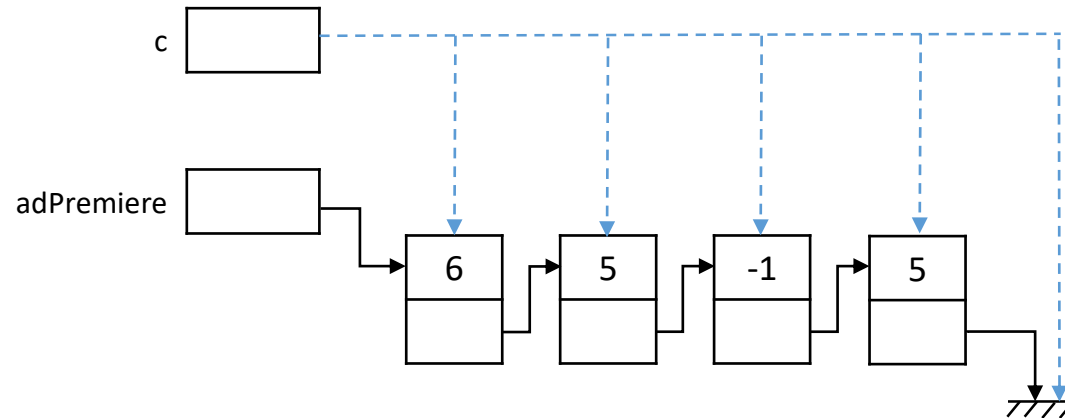
- il faut exécuter les instructions suivantes

```
Cellule * c = new Cellule;  
c->info = 2;  
c->suivant = adPremiere;  
adPremiere = c;
```

- ce qui fonctionne aussi quand la liste est vide (ajout du premier élément) car adPremiere vaut d'abord NULL, est affecté à la donnée membre suivant de la nouvelle cellule qui devient la première cellule

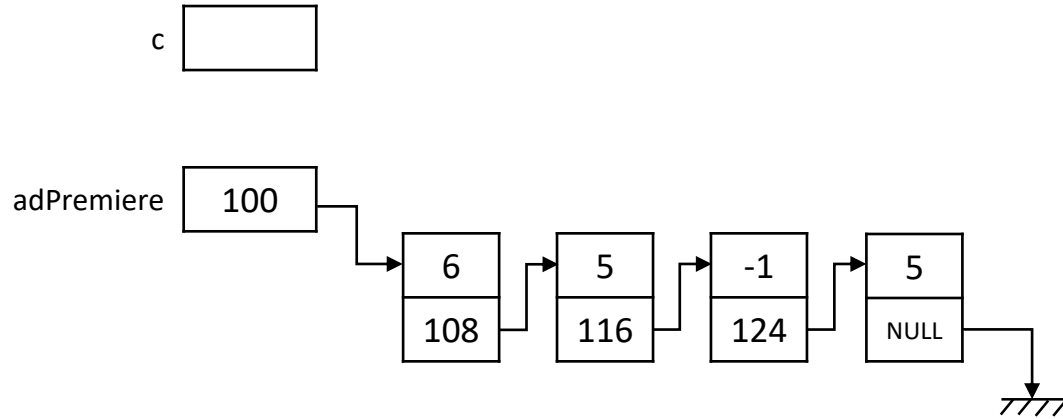
# Parcours des éléments d'une liste

- Beaucoup d'algorithmes nécessitent de parcourir tous les éléments d'une liste une et une seule fois
- On utilise une variable temporaire de type pointeur sur Cellule qui débute sur adPremiere et qui utilise les champs « suivant » jusqu'à ce que l'adresse de la cellule soit NULL



# Parcours des éléments d'une liste

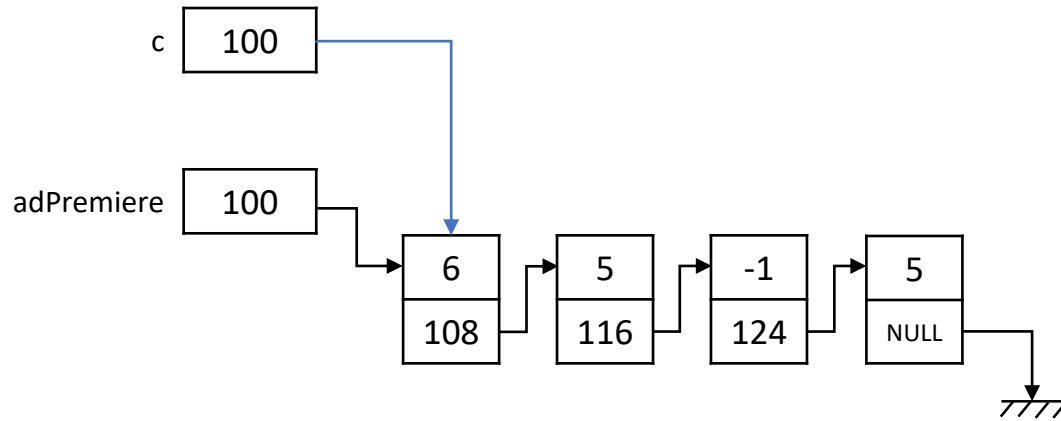
- Exemple pour afficher les éléments de la liste



- Trace écran :

# Parcours des éléments d'une liste

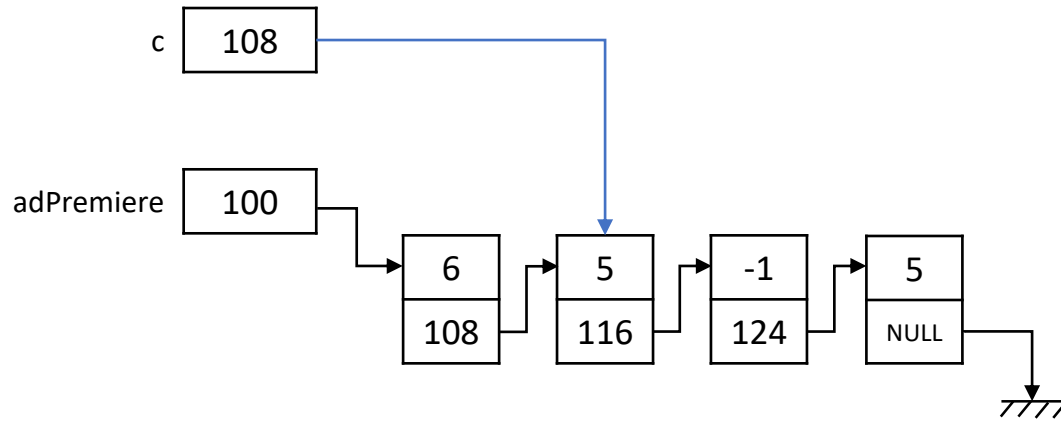
- Exemple pour afficher les éléments de la liste



- Trace écran : 6

# Parcours des éléments d'une liste

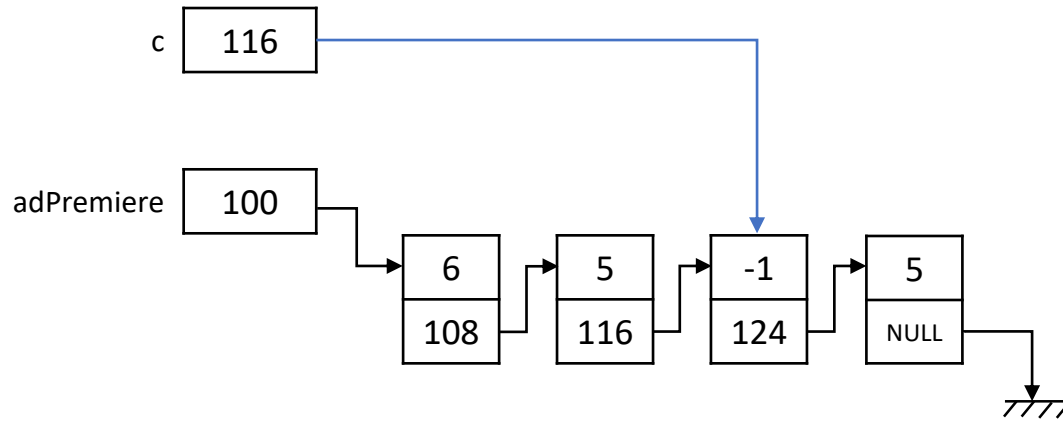
- Exemple pour afficher les éléments de la liste



- Trace écran : 6 5

# Parcours des éléments d'une liste

- Exemple pour afficher les éléments de la liste

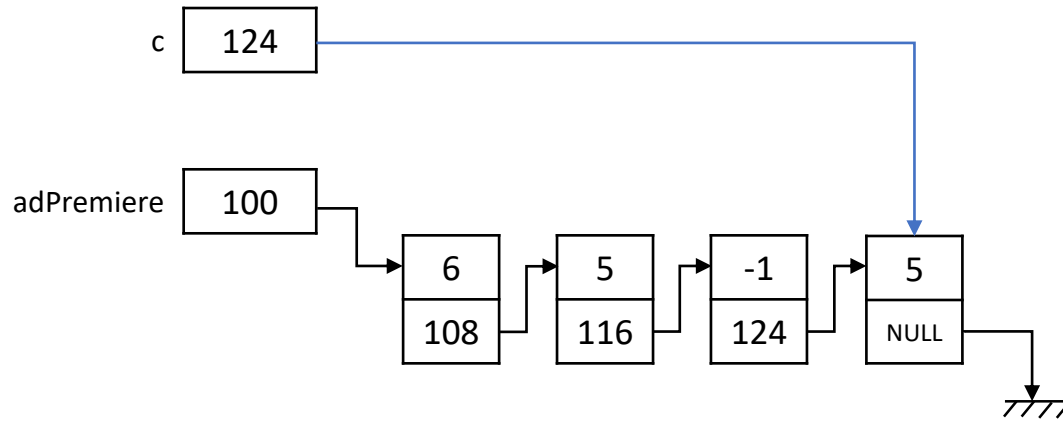


- Trace écran : 6 5 -1



# Parcours des éléments d'une liste

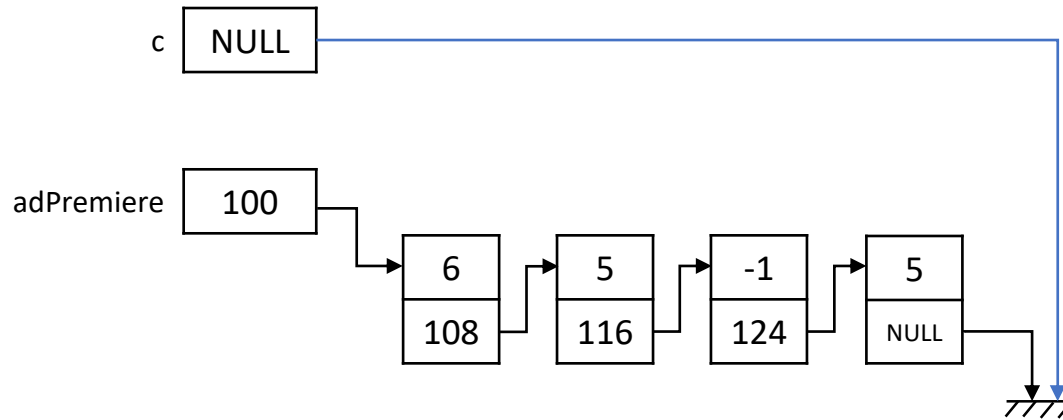
- Exemple pour afficher les éléments de la liste



- Trace écran : 6 5 -1 5

# Parcours des éléments d'une liste

- Exemple pour afficher les éléments de la liste



- Trace écran : 6 5 -1 5

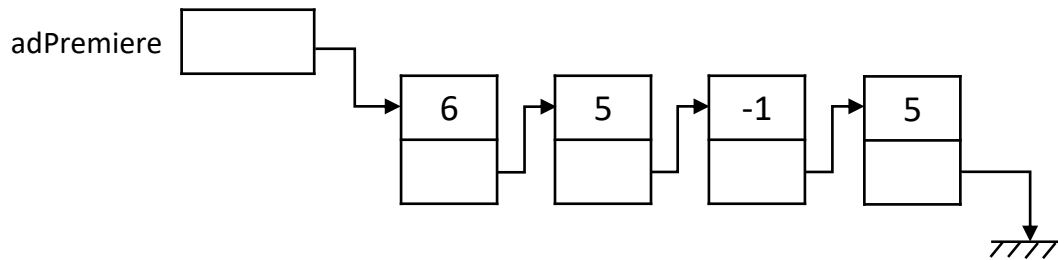
# Parcours des éléments d'une liste

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

```
Cellule * c = l.adPremiere;
while (c != NULL) {
    // faire ce que l'on veut faire sur l'élément pointé par c
    // par exemple : cout << c->info << " ";
    c = c->suivant;
}
```

# Décrochage et libération d'une cellule

- Comment libérer proprement les cellules d'une liste?

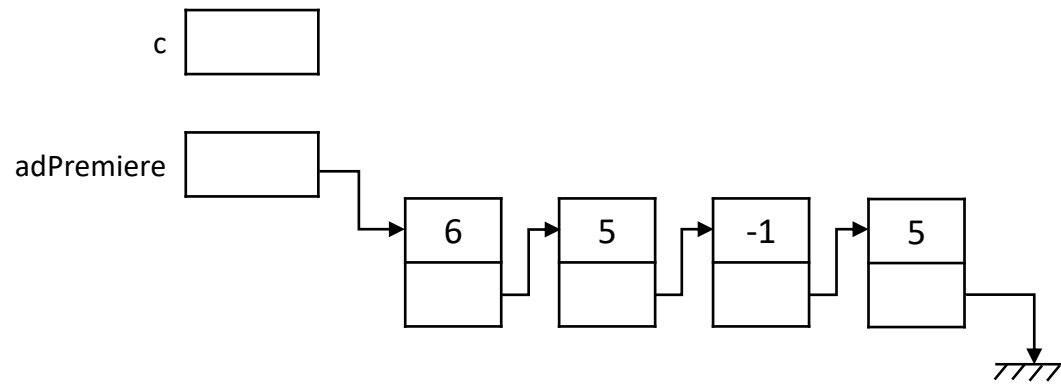


- C'est-à-dire libérer chaque cellule allouée sur le tas, sans en oublier, et obtenir une liste vide

# Décrochage et libération d'une cellule

## 1. Utiliser un pointeur de travail c

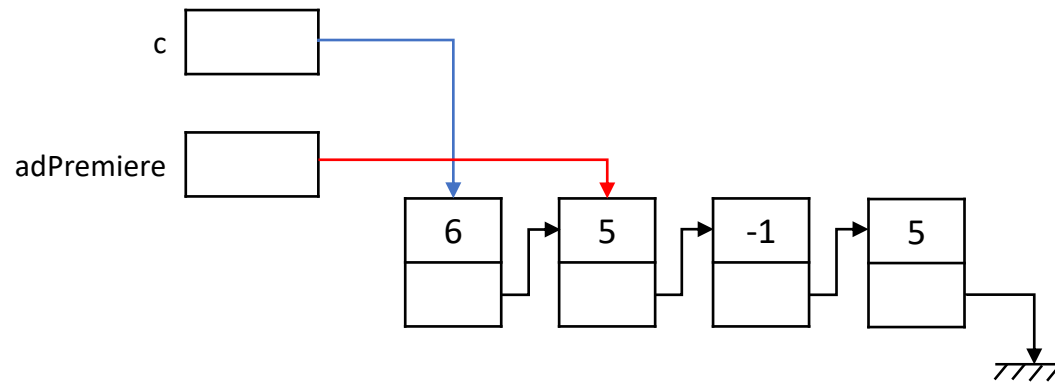
```
Cellule * c;
```



# Décrochage et libération d'une cellule

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

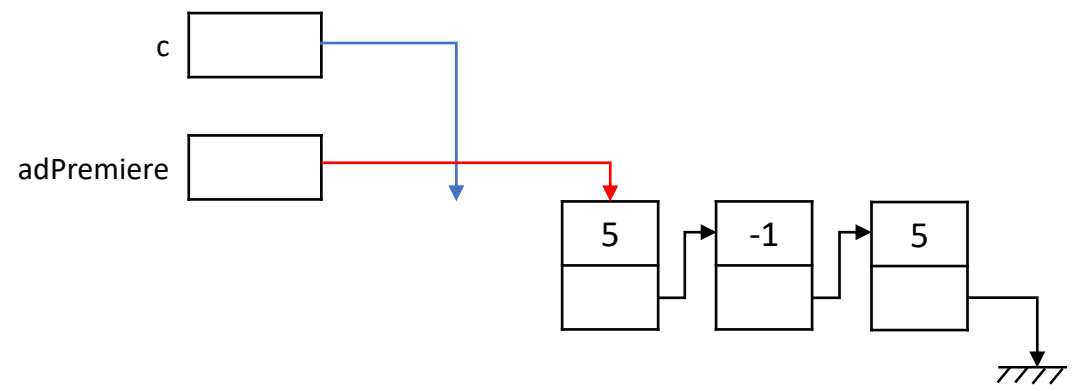
```
c = adPremiere;  
adPremiere = c->suivant;
```



# Décrochage et libération d'une cellule

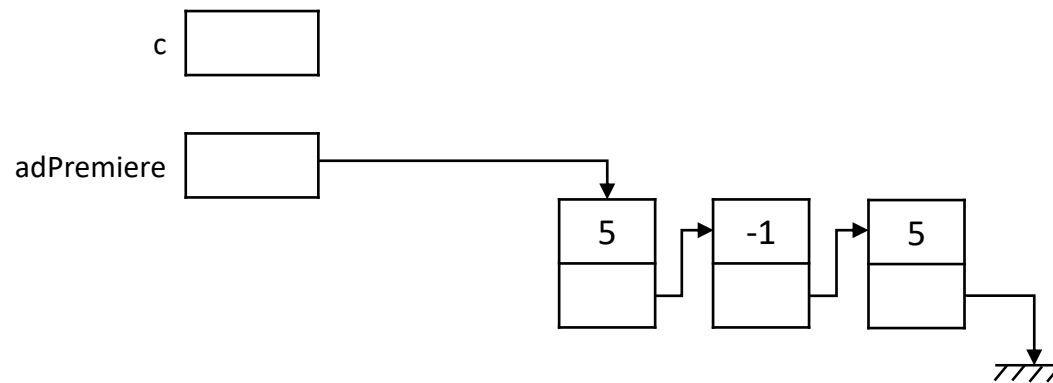
1. Utiliser un pointeur de travail `c`
2. Isoler, à l'aide de `c`, la cellule de tête sans perdre le reste de la liste
3. Libérer la cellule ainsi isolée

```
delete c;
```



# Décrochage et libération d'une cellule

1. Utiliser un pointeur 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)

```
Cellule * c;  
while (adPremiere != NULL) {  
    c = adPremiere;  
    adPremiere = c->suivant;  
    delete c;  
}
```

# Module Liste

**Module** Liste {une liste d'éléments}

- **Importer:**

- `Module ElementL`

- **Exporter:**

- `Type` Cellule
- `Type` Liste
  - `Constructeur` Liste()
    - Postconditions : la liste est une liste vide
  - `Destructeur` ~Liste()
    - Postconditions : libération de la mémoire utilisée sur le tas, la liste est une liste vide
  - `Procédure` ajouterEnTete (e: ElementL)
    - Postcondition : une copie de e est ajoutée en tête de liste
    - Paramètre en mode donnée : e
  - `Procédure` ajouterEnQueue (e: ElementL)
    - Postcondition : une copie de e est ajoutée en queue de liste
    - Paramètre en mode donnée : e
  - ...

# Module Liste

- ...
- **Procédure** vider ()
  - Postcondition : la liste ne contient plus aucune cellule
- **Fonction** estVide () : booléen
  - Résultat : retourne vrai si la liste est vide, faux sinon
- **Fonction** iemeElement (indice: entier positif) : ElementL
  - Précondition :  $0 \leq \text{indice} < \text{nombre d'éléments}$
  - Résultat : retourne l'élément à l'indice passé en paramètre
  - Paramètre en mode donnée : indice
- **Procédure** modifierIemeElement (e: ElementL, indice : entier positif)
  - Précondition :  $0 \leq \text{indice} < \text{nombre d'éléments}$
  - Postcondition : l'élément à l'indice passé en paramètre vaut e
  - Paramètre en mode donnée : e, indice
- **Procédure** afficher ()
  - Postcondition : Les éléments de la liste sont affichés sur la sortie standard
- **Procédure** supprimerTete ()
  - Postcondition : l'élément en tête de liste est supprimé
- **Procédure** insererElement (e: ElementL, indice : entier positif)
  - Précondition :  $0 \leq \text{indice} \leq \text{nombre d'éléments}$
  - Postcondition : une copie de e est insérée de sorte qu'elle occupe la position d'indice en paramètre
  - Paramètre en mode donnée : e, indice

# Module Liste

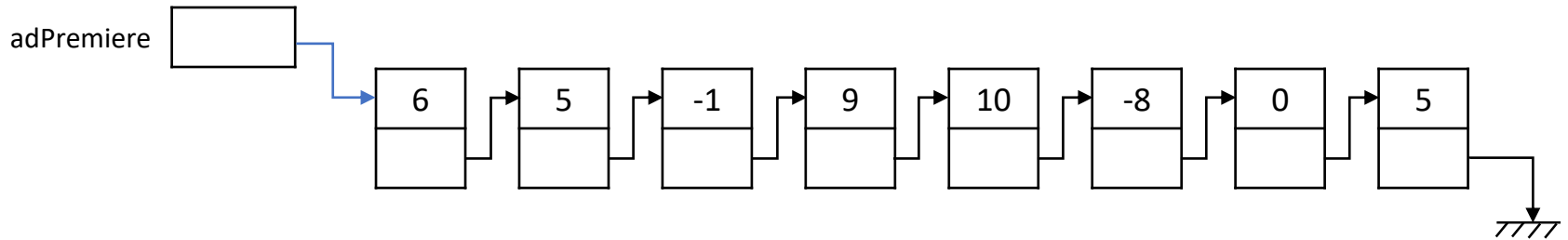
- **Implémentation:**

- **Type** Cellule = Structure
  - info : ElementL
  - suisvant : lien sur CelluleFin Structure
- **Type** Liste = Classe
  - adPremiere : lien sur Cellule
  - **Constructeur** Liste()
    - Début
    - adPremiere ← NULLFin
  - **Fonction** estVide ()
    - Début
    - retourne** adPremiere = NULLFin
  - Etc.Fin Classe

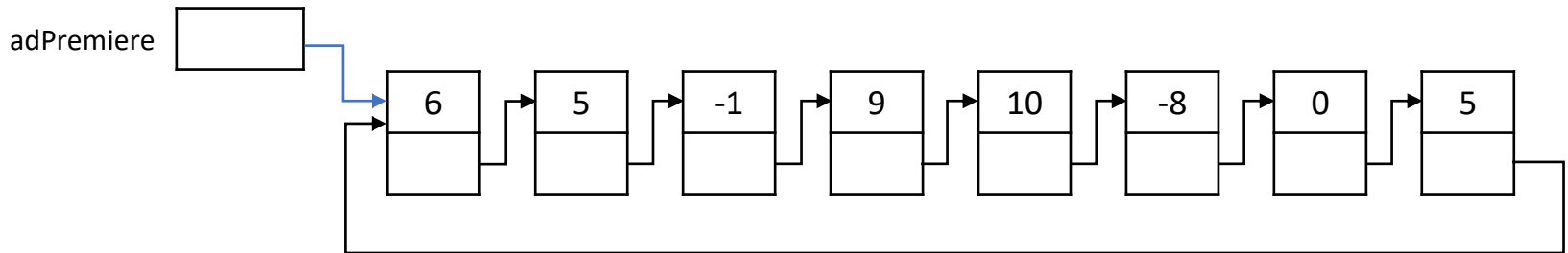
**FinModule** Liste

# Les listes chaînées

- La liste simplement chaînée (CM et TD)

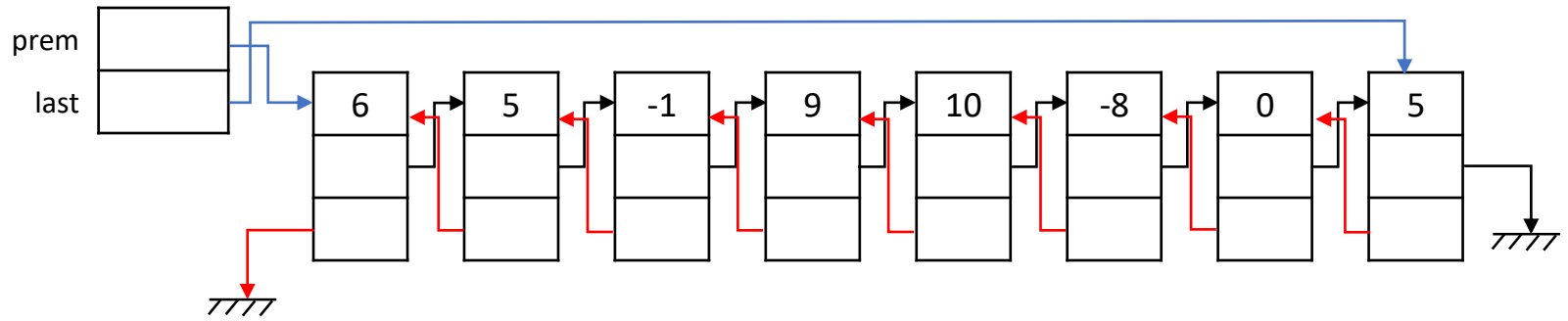


- La liste simplement chaînée circulaire



# Les listes chaînées

- La liste doublement chaînée (TP)



- La liste doublement chaînée circulaire

