v1.3

INFORMATIQUE 3

I. LES LISTES CHAINÉES

Eva Ansermin & Romuald Grignon

CY TECH

Un point sur l'informatique 3

- Sujets abordés dans la matière:
  - **Algorithmie :** listes chaînées, piles, files, arbres, …
  - UNIX et Script Shell
- Organisation:
  - 6 CM (algo) + 2 CM (Unix/Shell)
  - TD sur 13 semaines (Algo : 7 / Unix-Shell : 3 / Projet : 3)
- Evaluation :
  - 3 DS (2 x 90min + 1 x 120min)
  - Moyenne des quiz qui compte pour 1 DS.
  - 1 Projet en fin de semestre
  - Moyenne = 0.7 * max(moy(DS1, DS2, DS3, QUIZ), DS3) + 0.3 * PROJET

Page de cours + Quiz

- Des quiz auront lieu régulièrement à partir d'octobre en début de séance de TD, en ligne sur la page du cours : https://cours.cyu.fr/course/view.php?id=338

- Vous trouverez aussi des support de cours, TD, quiz d'entrainement, archives de DS, liste des intervenants par groupe, et les modalités d'évaluation.

- **Inscrivez-vous sur ce site avec votre groupe pour accéder aux évaluations !**

I. Rappels

Pointeur et pseudo-code

- Un **pointeur** est :
  - Une variable dont la valeur est une adresse mémoire…
  - L'adresse mémoire d'une autre variable !

```
int  a;
int* p1 = NULL;
a  = 5;
p1 = &a;
printf(" a =%d \n",  a);
printf("*p1=%d \n",*p1);
*p1 = 10;
printf(" a =%d \n",  a);
```

Adresse	Valeur
1154	
1158	
…	
5340	
…	
…	
10 032	
…	
…	
22 222	

Pointeur et pseudo-code

- Un **pointeur** est :
  - Une variable dont la valeur est une adresse mémoire…
  - L'adresse mémoire d'une autre variable !

```
int  a;
int* p1 = NULL;
a  = 5;
p1 = &a;
printf(" a =%d \n",  a);
printf("*p1=%d \n",*p1);
*p1 = 10;
printf(" a =%d \n",  a);
```

Adresse	Valeur
1154	
1158	
…	
5340	
…	
…	
10 032	
…	
…	
22 222	

Pointeur et pseudo-code

- Un **pointeur** est :
  - Une variable dont la valeur est une adresse mémoire…
  - L'adresse mémoire d'une autre variable !

```
int  a;
int* p1 = NULL;
a  = 5;
p1 = &a;
printf(" a =%d \n",  a);
printf("*p1=%d \n",*p1);
*p1 = 10;
printf(" a =%d \n",  a);
```

Adresse	Valeur
1154	???
1158	
…	
5340	
…	
…	
10 032	
…	
…	
22 222	

a

Pointeur et pseudo-code

- Un **pointeur** est :
  - Une variable dont la valeur est une adresse mémoire…
  - L'adresse mémoire d'une autre variable !

```
int  a;
int* p1 = NULL;
a  = 5;
p1 = &a;
printf(" a =%d \n",  a);
printf("*p1=%d \n",*p1);
*p1 = 10;
printf(" a =%d \n",  a);
```

Adresse	Valeur	
1154	???	a
1158		
…		
5340		
…		
…		
10 032	0 (NULL)	p1
…		
…		
22 222		

Pointeur et pseudo-code

- Un **pointeur** est :
  - Une variable dont la valeur est une adresse mémoire…
  - L'adresse mémoire d'une autre variable !

```
int  a;
int* p1 = NULL;
a  = 5;
p1 = &a;
printf(" a =%d \n",  a);
printf("*p1=%d \n",*p1);
*p1 = 10;
printf(" a =%d \n",  a);
```

Adresse	Valeur	
1154	5	a
1158		
…		
5340		
…		
…		
10 032	0 (NULL)	p1
…		
…		
22 222		

Pointeur et pseudo-code

- Un **pointeur** est :
  - Une variable dont la valeur est une adresse mémoire…
  - L'adresse mémoire d'une autre variable !

```
int  a;
int* p1 = NULL;
a  = 5;
p1 = &a;
printf(" a =%d \n",  a);
printf("*p1=%d \n",*p1);
*p1 = 10;
printf(" a =%d \n",  a);
```

Adresse	Valeur	
1154	5	a
1158		
…		
5340		
…		
…		
10 032	1154	p1
…		
…		
22 222		

Pointeur et pseudo-code

- Un **pointeur** est :
  - Une variable dont la valeur est une adresse mémoire…
  - L'adresse mémoire d'une autre variable !

```
int  a;
int* p1 = NULL;
a  = 5;
p1 = &a;
printf(" a =%d \n",  a);
printf("*p1=%d \n",*p1);
*p1 = 10;
printf(" a =%d \n",  a);
```

Adresse	Valeur	
1154	5	a
1158		
…		
5340		
…		
…		
10 032	1154	p1
…		
…		
22 222		

a=5

Pointeur et pseudo-code

- Un **pointeur** est :
  - Une variable dont la valeur est une adresse mémoire…
  - L'adresse mémoire d'une autre variable !

```
int  a;
int* p1 = NULL;
a  = 5;
p1 = &a;
printf(" a =%d \n",  a);
printf("*p1=%d \n",*p1);
*p1 = 10;
printf(" a =%d \n",  a);
```

Adresse	Valeur	
1154	5	a
1158		
…		
5340		
…		
…		
10 032	1154	p1
…		
…		
22 222		

a=5
p1=5

Pointeur et pseudo-code

- Un **pointeur** est :
  - Une variable dont la valeur est une adresse mémoire…
  - L'adresse mémoire d'une autre variable !

```
int  a;
int* p1 = NULL;
a  = 5;
p1 = &a;
printf(" a =%d \n",  a);
printf("*p1=%d \n",*p1);
*p1 = 10;
printf(" a =%d \n",  a);
```

Adresse	Valeur	
1154	10	a
1158		
…		
5340		
…		
…		
10 032	1154	p1
…		
…		
22 222		

a=5
p1=5

Pointeur et pseudo-code

- Un **pointeur** est :
  - Une variable dont la valeur est une adresse mémoire…
  - L'adresse mémoire d'une autre variable !

```
int  a;
int* p1 = NULL;
a  = 5;
p1 = &a;
printf(" a =%d \n",  a);
printf("*p1=%d \n",*p1);
*p1 = 10;
printf(" a =%d \n",  a);
```

Adresse	Valeur	
1154	10	a
1158		
…		
5340		
…		
…		
10 032	1154	p1
…		
…		
22 222		

a=5
p1=5
a=10

Eva ANSERMIN & Romuald GRIGNON

Pointeur et pseudo-code

- Un **pointeur** est :
  - Une variable dont la valeur est une adresse mémoire…
  - L'adresse mémoire d'une autre variable !

```
int main(){
    int  a;
    int* p1 = NULL;
    a  = 5;
    p1 = &a;
    printf(" a =%d \n",  a);
    printf("*p1=%d \n",*p1);
    *p1 = 10;
    printf(" a =%d \n",  a);
    return 0;
    }
```

```
VARIABLE
        a : entier
        p1 : pointeur sur entier
DEBUT
        a ← 5
        p1 ← adresse de a
        ECRIRE(«a=»+a)
        ECRIRE(«*p1 =»+*p1)
        *p1 ← 10
        ECRIRE(« a =»+a)
FIN
```

Les tableaux

- Les valeurs d'un tableau sont à la suite dans la mémoire.
- Lorsqu'un tableau est déclaré :
  - Un espace mémoire de la bonne taille est réservé.
  - Un pointeur constant portant le nom du tableau est créé. Il pointe sur la première case du tableau.

```
VARIABLE
        tab[5] : tableau d'entiers
DEBUT
        tab <- {1,3}
FIN
```

	Adresse	Valeur
tab[0]	1154	1
tab[1]		3
tab[2]		0
tab[3]		0
tab[4]		0
	…	
	…	
	…	
	…	
tab	22 220	1154

Les tableaux : contraintes

- Il est obligatoire de connaître la taille d'un tableau lors de sa déclaration.
- Même lors d'une allocation dynamique, la taille de l'espace mémoire à allouer doit être donnée.

- Solutions possibles :
  - Déclarer une taille de tableau suffisamment grande :
    - ➢ Il faut être certain qu'on ne dépassera pas.
    - ➢ Risque de mémoire « gâchée ».
  - Imposer une limite de taille à l'utilisateur
    - ➢ Application limitée.

  - Utiliser une liste chaînée.

II. Listes chaînées : principe

Les listes chaînées

- Une liste chaînée est un objet informatique permettant de stocker des éléments dans un ordre précis (comme un tableau).

- Contrairement à un tableau, les éléments d'une liste chaînée ne sont pas à la suite dans la mémoire.

- Même lors d'une allocation dynamique, la taille de l'espace mémoire à allouer doit être donnée.

- Un élément stocké dans une liste chaînée est toujours accompagné d'un **pointeur indiquant où se trouve l'élément suivant de la liste dans la mémoire.**

  ✓ Les éléments de la liste ne sont pas à la suite dans la mémoire mais on sait toujours où se trouve l'élément suivant grâce à un pointeur !

Constitution d'une liste chaînée

- Une liste chaînée est constituée de chaînons, une structure qui contient :
  - L' élément à stocker (entier, flottant, tableau de caractères etc...)
  - Un pointeur vers l'occurrence suivante.

Constitution d'une liste chaînée

- Une liste chaînée est constituée de chaînons, une structure qui contient :
  - L' élément à stocker (entier, flottant, tableau de caractères etc...)
  - Un pointeur vers l'occurrence suivante.

```
Structure Chainon :
        elmt : Element
        suivant : pointeur sur structure Chainon
```

Constitution d'une liste chaînée

- Une liste chaînée est constituée de chaînons, une structure qui contient :
  - L' élément à stocker (entier, flottant, tableau de caractères etc...)
  - Un pointeur vers l'occurrence suivante.

```
Structure Chainon :
        elmt : Element
        suivant : pointeur sur structure Chainon
```

  - Elément de type entier

```
Structure Chainon :
        elmt : entier
        suivant : pointeur sur structure Chainon
```

Constitution d'une liste chaînée

- Une liste chaînée est constituée de chaînons, une structure qui contient :
  - L' élément à stocker (entier, flottant, tableau de caractères etc...)
  - Un pointeur vers l'occurrence suivante.

```
Structure Chainon :
        elmt : Element
        suivant : pointeur sur structure Chainon
```

  - Elément de type réel

```
Structure Chainon :
        elmt : réel
        suivant : pointeur sur structure Chainon
```

Constitution d'une liste chaînée

- Une liste chaînée est constituée de chaînons, une structure qui contient :
  - L' élément à stocker (entier, flottant, tableau de caractères etc...)
  - Un pointeur vers l'occurrence suivante.

```
Structure Chainon :
        elmt : Element
        suivant : pointeur sur structure Chainon
```

  - Elément de type structure

```
Structure Chainon :
        elmt : Etudiant
        suivant : pointeur sur structure Chainon
```

Constitution d'une liste chaînée

- Une liste chaînée est constituée de chaînons, une structure qui contient :
  - L' élément à stocker (entier, flottant, tableau de caractères etc...)
  - Un pointeur vers l'occurrence suivante.

```
Structure Chainon :
        elmt : Element
        suivant : pointeur sur structure Chainon
```

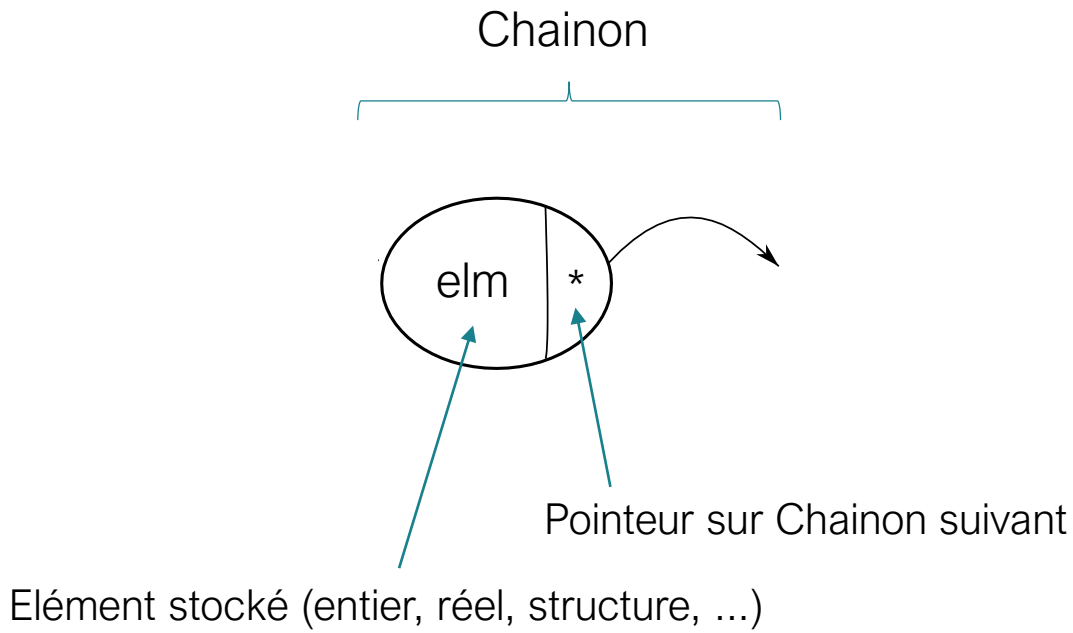
- Exemple en C :

```
struct chainon
{
    int nombre;
    struct chainon* suivant;
};
typedef struct chainon Chainon;
```

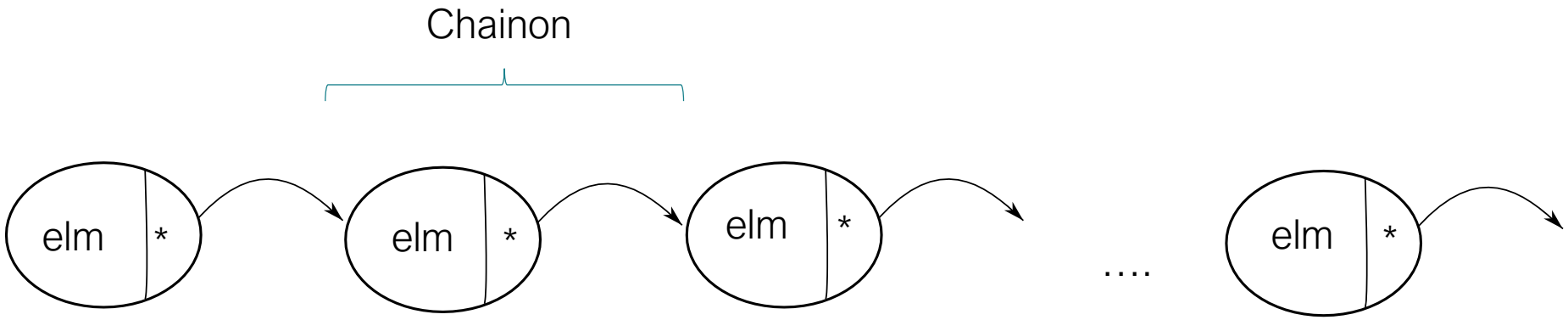
OU

```
typedef struct chainon
{
    int nombre;
    struct chainon* suivant;
} Chainon;
```

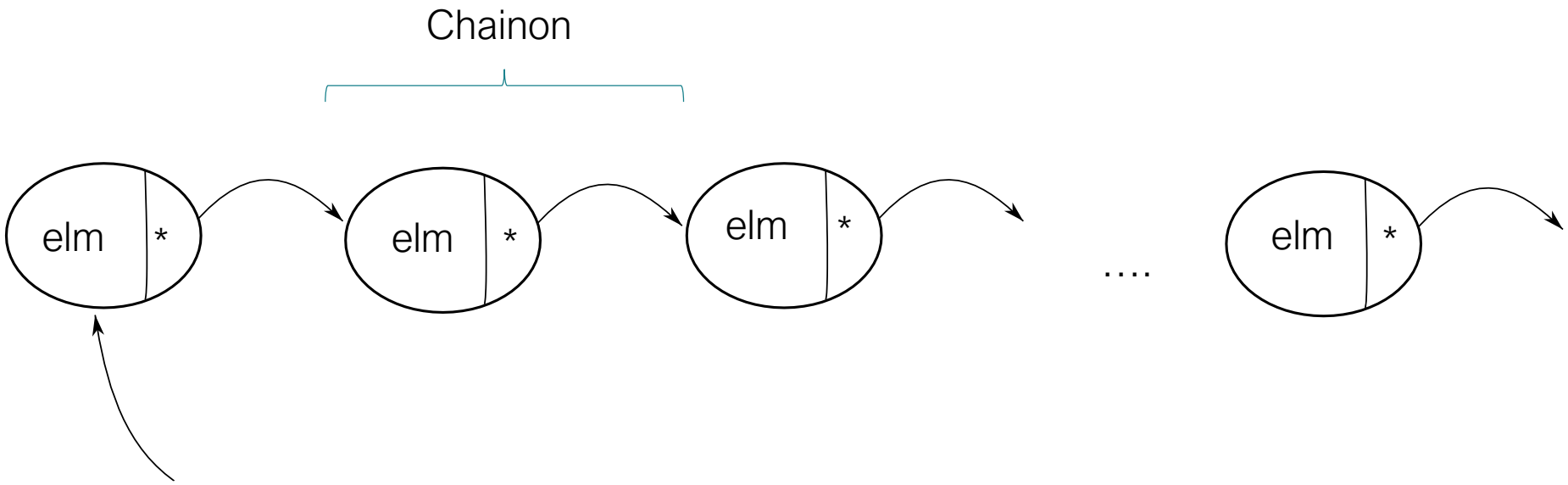
Listes chaînées : schéma



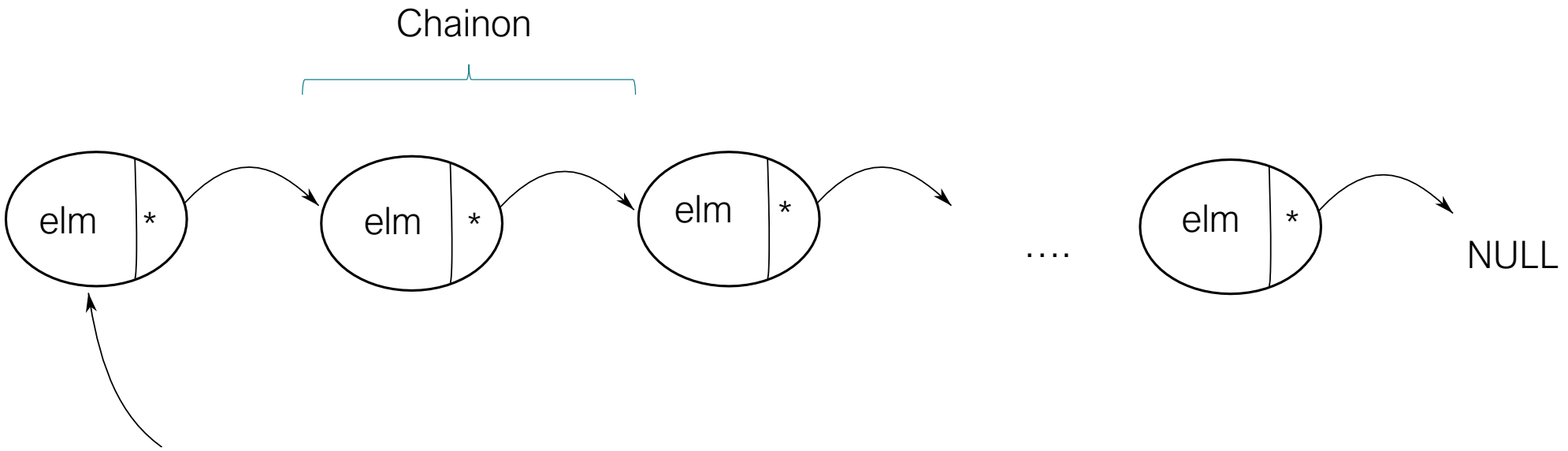
Listes chaînées : schéma



Listes chaînées : schéma



Listes chaînées : schéma



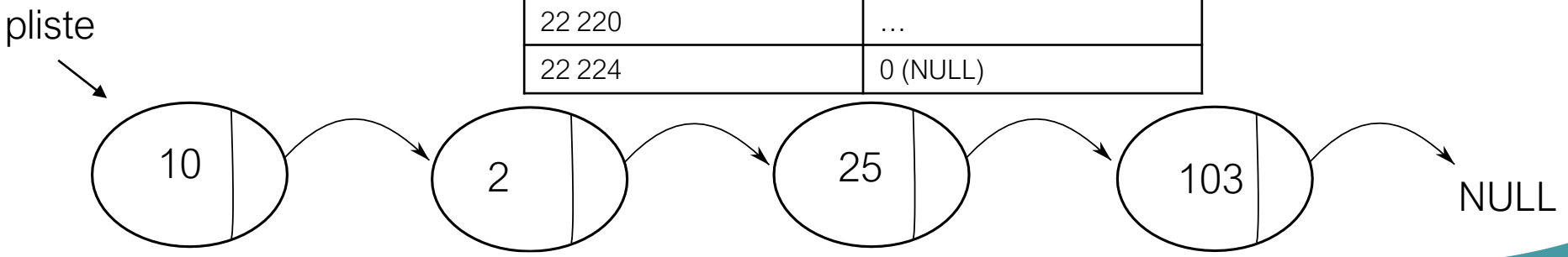
Pointeur sur le premier Chainon de la liste (Tête)

Le dernier Chainon pointe sur NULL

Listes chaînées en mémoire : exemple

pliste

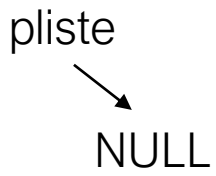
Adresse (décimale)	Valeur (décimale)
1 000	1152
…	…
1152	10
1156	…
1160	5344
…	…
5344	2
5348	…
5352	10 016
…	…
10 016	25
10020	…
10 024	22 222
…	…
22 216	103
22 220	…
22 224	0 (NULL)



II. Listes chaînées : opérations

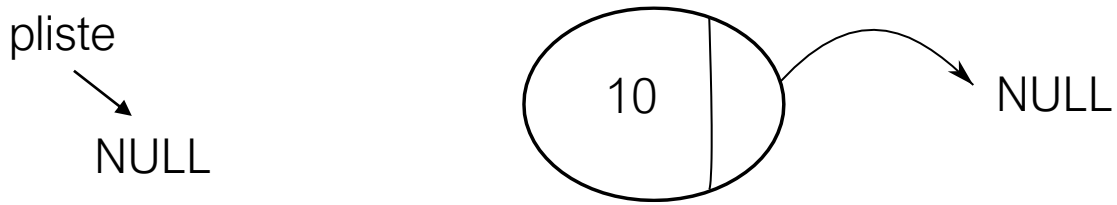
Listes chaînées : construction

1. Initialisation de la chaîne : initialisation pointeur sur Chainon NULL



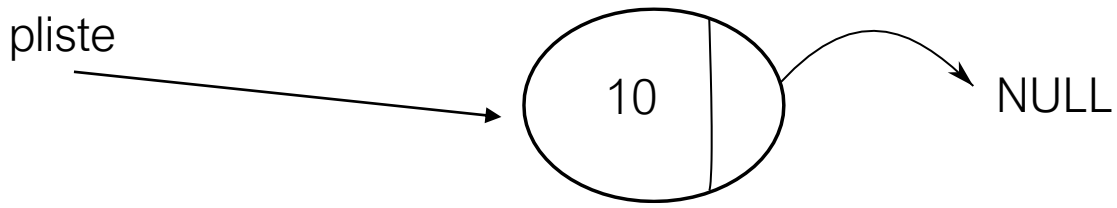
Listes chaînées : construction

1. Initialisation de la chaîne : initialisation pointeur sur Chainon NULL
2. Création d'un Chainon



Listes chaînées : construction

1. Initialisation de la chaîne : initialisation pointeur sur Chainon NULL
2. Création d'un Chainon
3. Faire pointer le pointeur sur le Chainon créé



Liste chaînée : construction

1. Initialisation de la chaîne : initialisation pointeur sur Chainon NULL
2. Création d'un Chainon

```
Chainon* creationChainon(){
        //déclaration du Chainon
        Chainon* c = malloc(sizeof(Chainon));
        if(c==NULL){
           exit(1);
        }
        printf("Entrer la valeur : ");
        if( scanf("%d", &(c->elmt)) != 1 ){
            exit(2);
        }
        c->suivant=NULL;
        return c;
}
```

Remarque : nous verrons plus tard l'intérêt d'utiliser le **malloc**

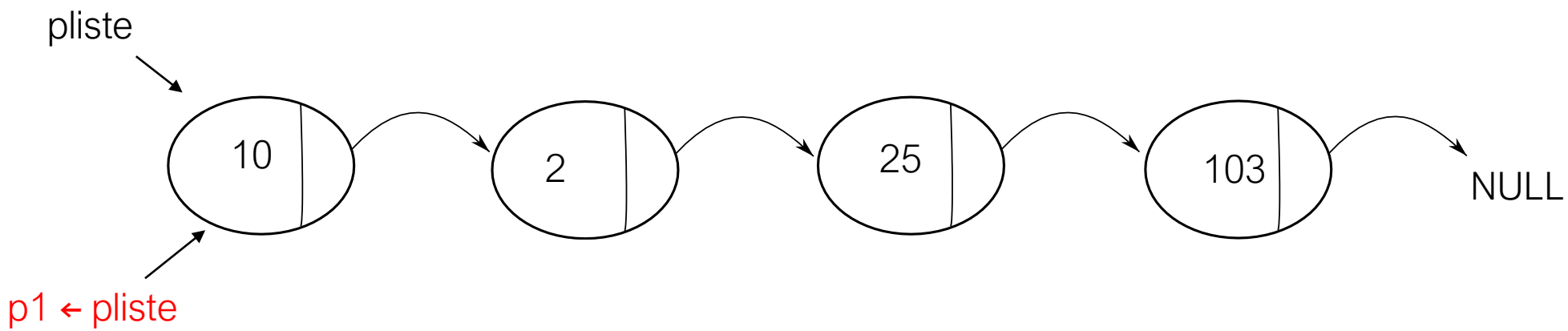
Listes chaînées : construction

1. Initialisation de la chaîne : initialisation pointeur sur Chainon NULL
2. Création d'un Chainon
3. Pointer le pointeur sur le Chainon créé

```
Chainon* creationChainon(){
        //déclaration du Chainon
        Chainon* c = malloc(sizeof(Chainon));
        if(c==NULL){
            exit(1);
        }
        printf("Entrer la valeur : ");
        if( scanf("%d", &(c->elmt)) != 1 ){
            exit(2);
        }
        c->suivant=NULL;
        return c;
}

int main(){
        Chainon* pliste = NULL; //initialisation
        pListe = creationChainon();
        …
        return 0;
}
```

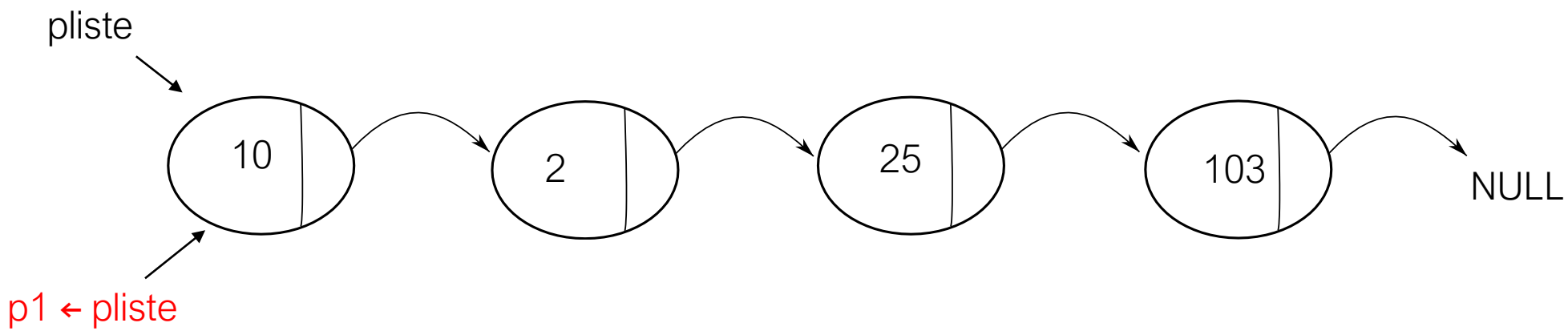
Listes chaînées : parcours



p1 = pliste;

- Pour parcourir une liste chaînée :
  - On déclare un pointeur = pliste. Il pointe donc sur le premier élément.

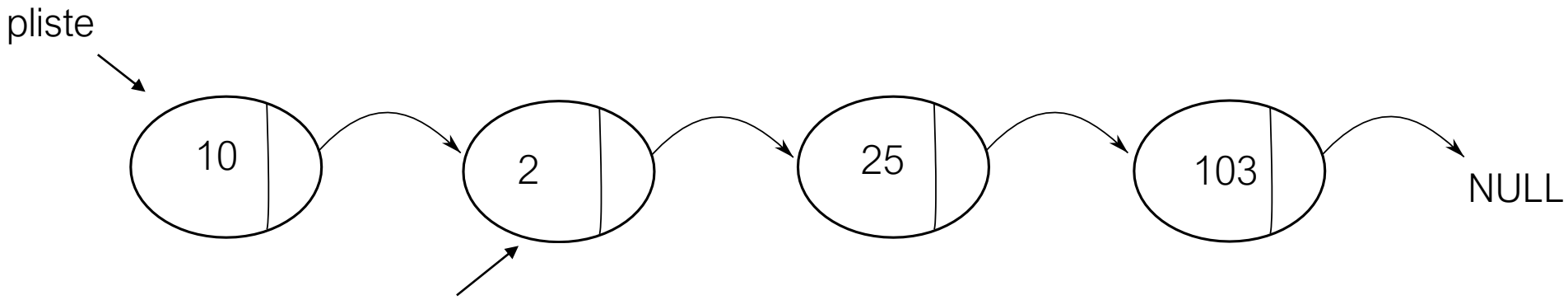
Listes chaînées : parcours



p1 = pliste;

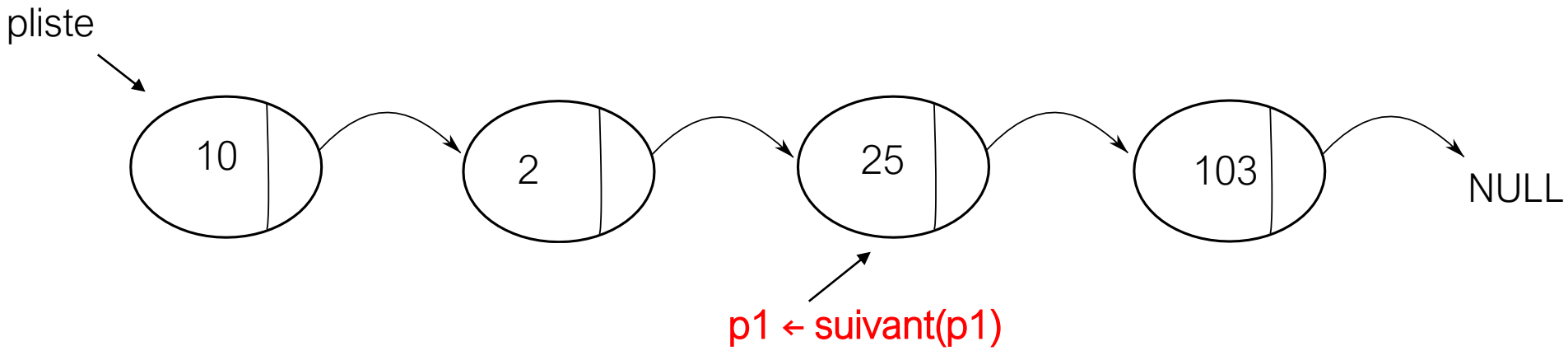
- Pour parcourir une liste chaînée :
  - On déclare un pointeur = pliste. Il pointe donc sur le premier élément.
  - On traite le premier élément.

Listes chaînées : parcours



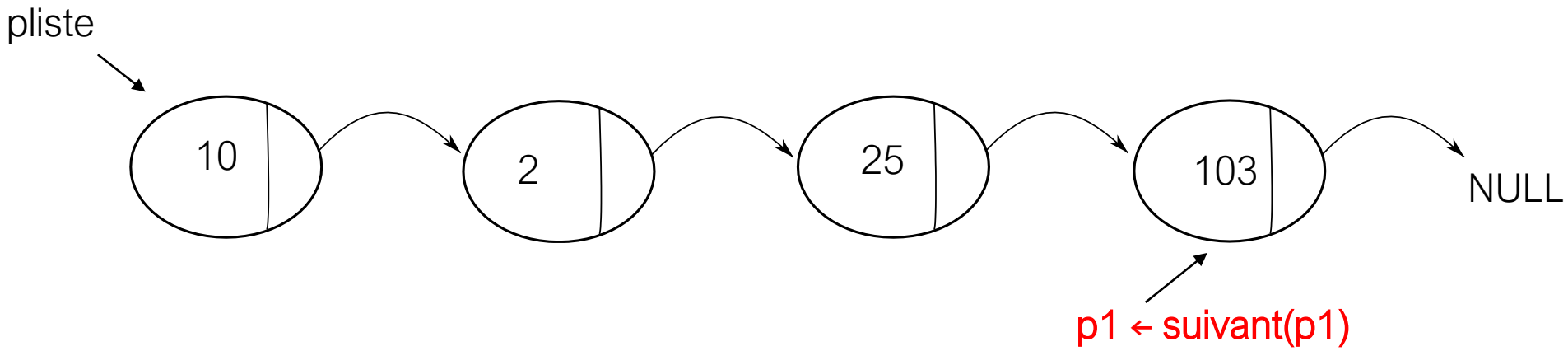
- Pour parcourir une liste chaînée :
  - On déclare un pointeur = pliste. Il pointe donc sur le premier élément.
  - On traite le premier élément.
  - On passe au Chainon suivant et on traite son élément

Listes chaînées : parcours



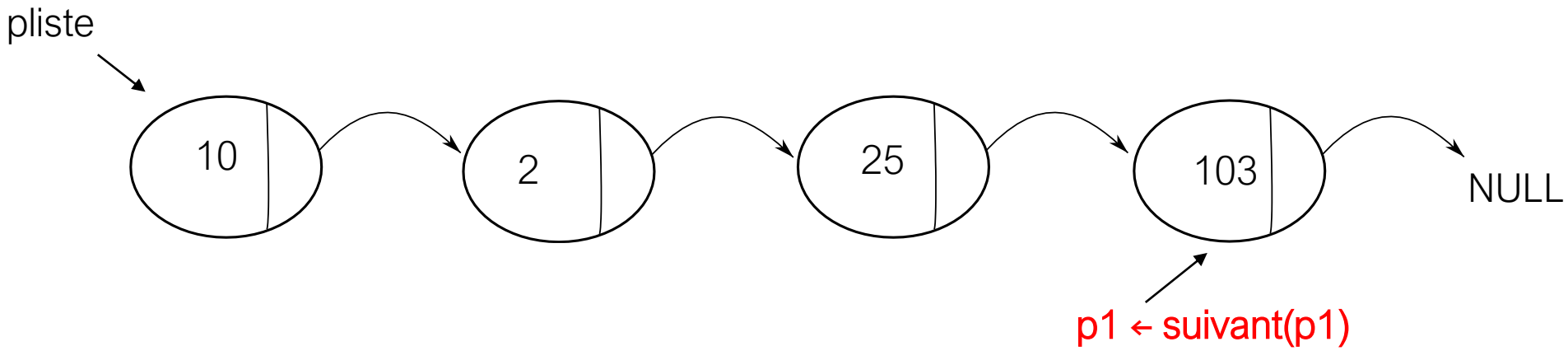
- Pour parcourir une liste chaînée :
  - On déclare un pointeur = pliste. Il pointe donc sur le premier élément.
  - On traite le premier élément.
  - On passe au Chainon suivant et on traite son élément.

Listes chaînées : parcours



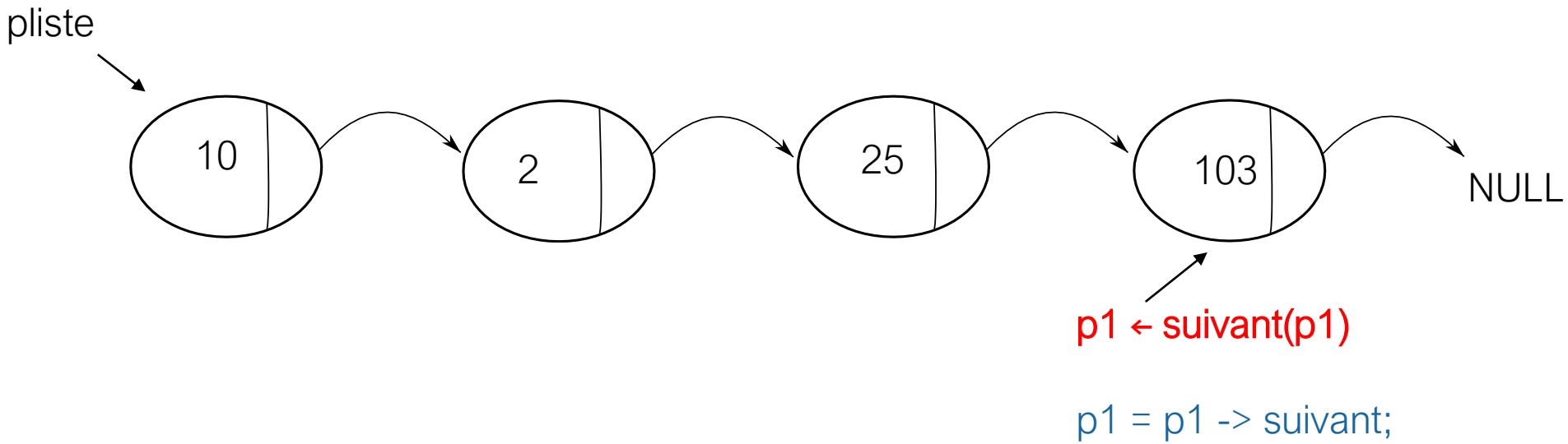
- Pour parcourir une liste chaînée :
  - On déclare un pointeur = pliste. Il pointe donc sur le premier élément.
  - On traite le premier élément.
  - On passe au Chainon suivant et on traite son élément.
  - …

Listes chaînées : parcours



- Pour parcourir une liste chaînée :
  - On déclare un pointeur = pliste. Il pointe donc sur le premier élément.
  - On traite le premier élément.
  - On passe au Chainon suivant et on traite son élément.
  - …
  - Jusqu'à arriver au pointeur NULL

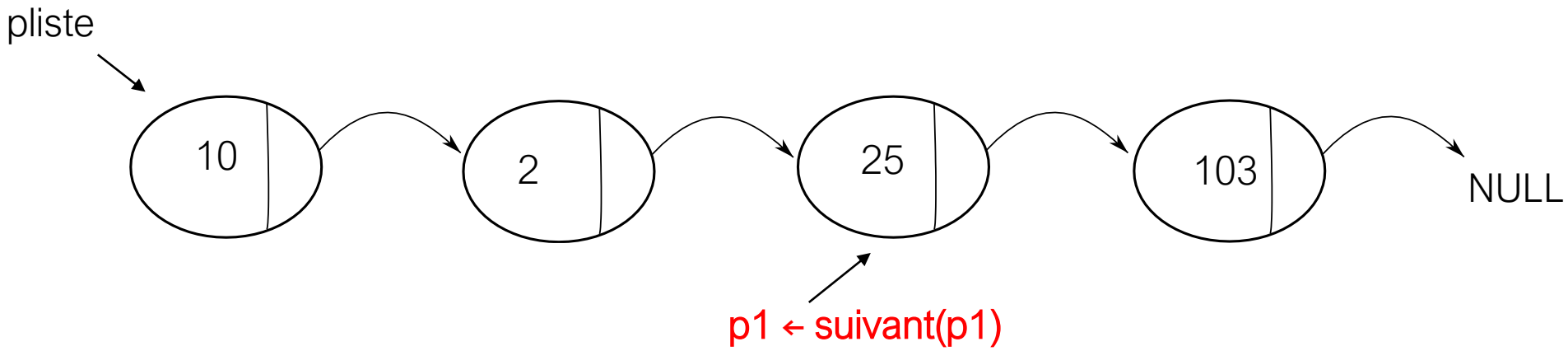
Listes chaînées : parcours



- Pour parcourir une liste chaînée :
  - On déclare un pointeur = pliste. Il pointe donc sur le premier élément.
  - On traite le premier élément.
  - On passe au Chainon suivant et on traite son élément.
  - …
  - Jusqu'à arriver au pointeur NULL

On ne peut pas utiliser d'indice comme pour les tableaux !!

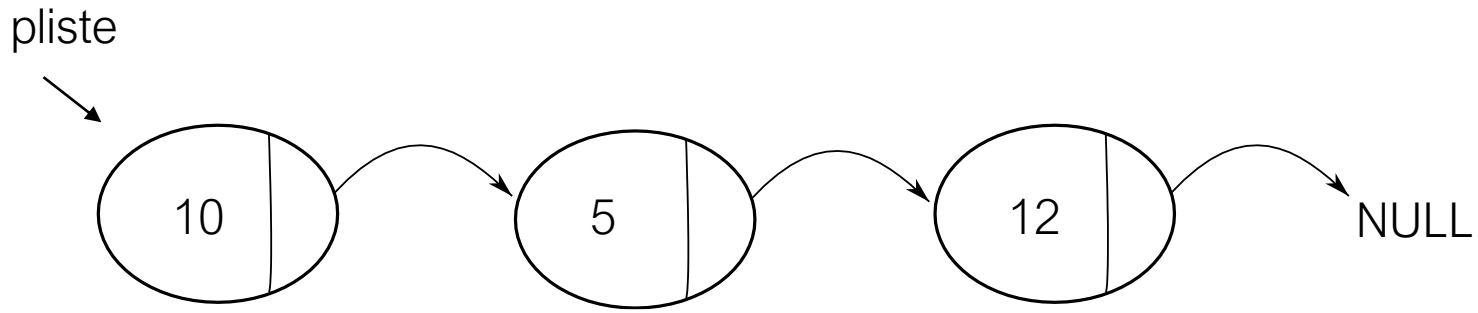
Listes chaînées : parcours




```
procédure traiterListe(pliste : pointeur sur Chainon)
VARIABLE
        p1 : pointeur sur Chainon
DEBUT
        p1 ← pliste;
        TANT QUE p1 ≠ NULL FAIRE
                traiter (element(p1))
                p1 ← suivant(p1)
        FIN TANT QUE
}
```

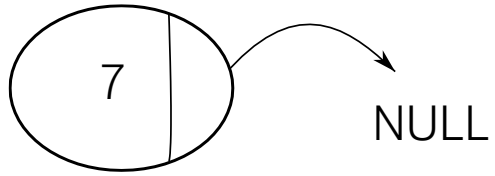
Listes chaînées : ajout de Chainon

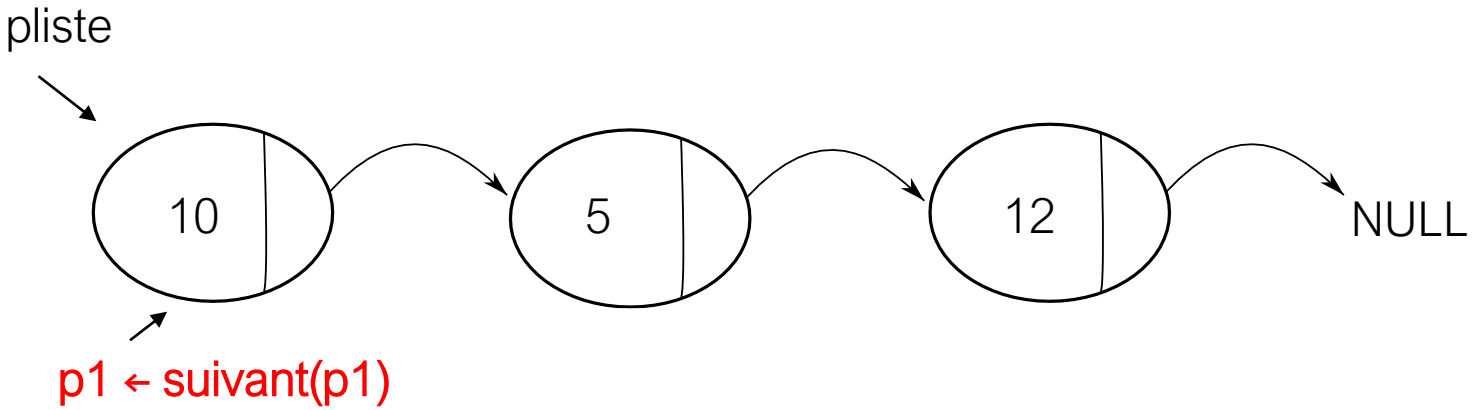
1. Ajout d'un Chainon en **fin de chaîne**



Listes chaînées : ajout de Chainon

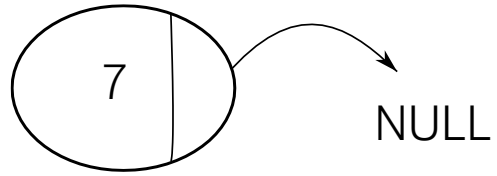
1. Ajout d'un Chainon en **fin de chaîne**
   a. Création du Chainon
   b. Parcours de la liste jusqu'au dernier Chainon

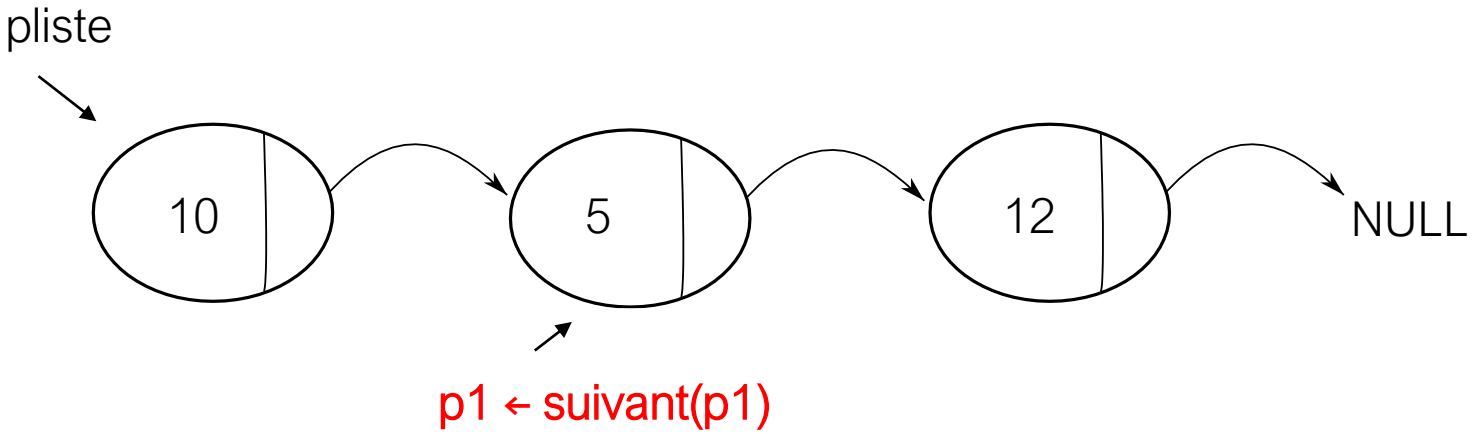




Listes chaînées : ajout de Chainon

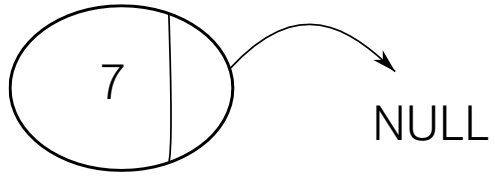
1. Ajout d'un Chainon en **fin de chaîne**
   a. Création du Chainon
   b. Parcours de la liste jusqu'au dernier Chainon

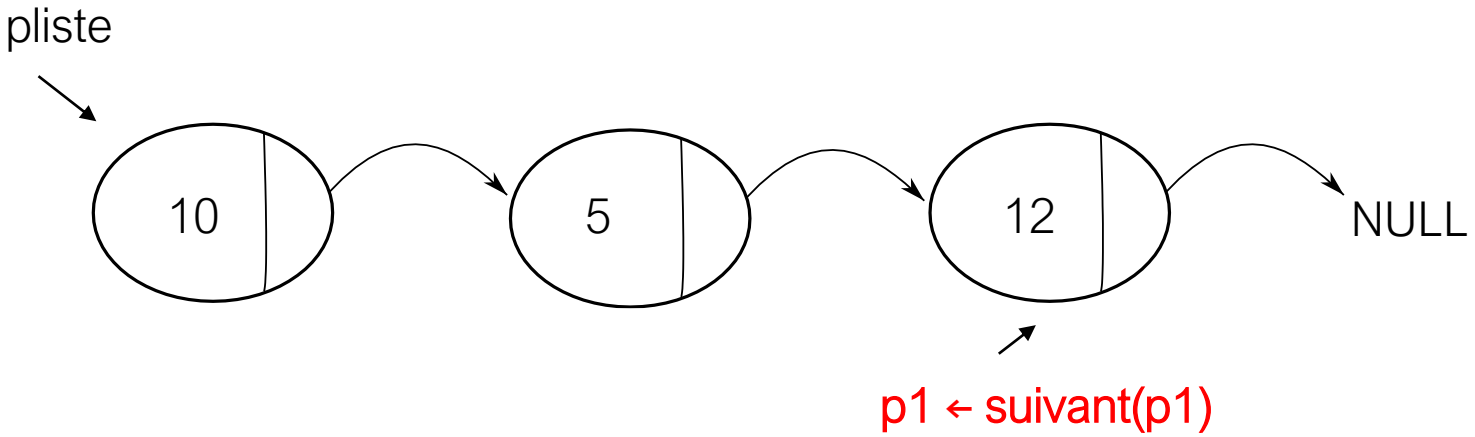




Listes chaînées : ajout de Chainon

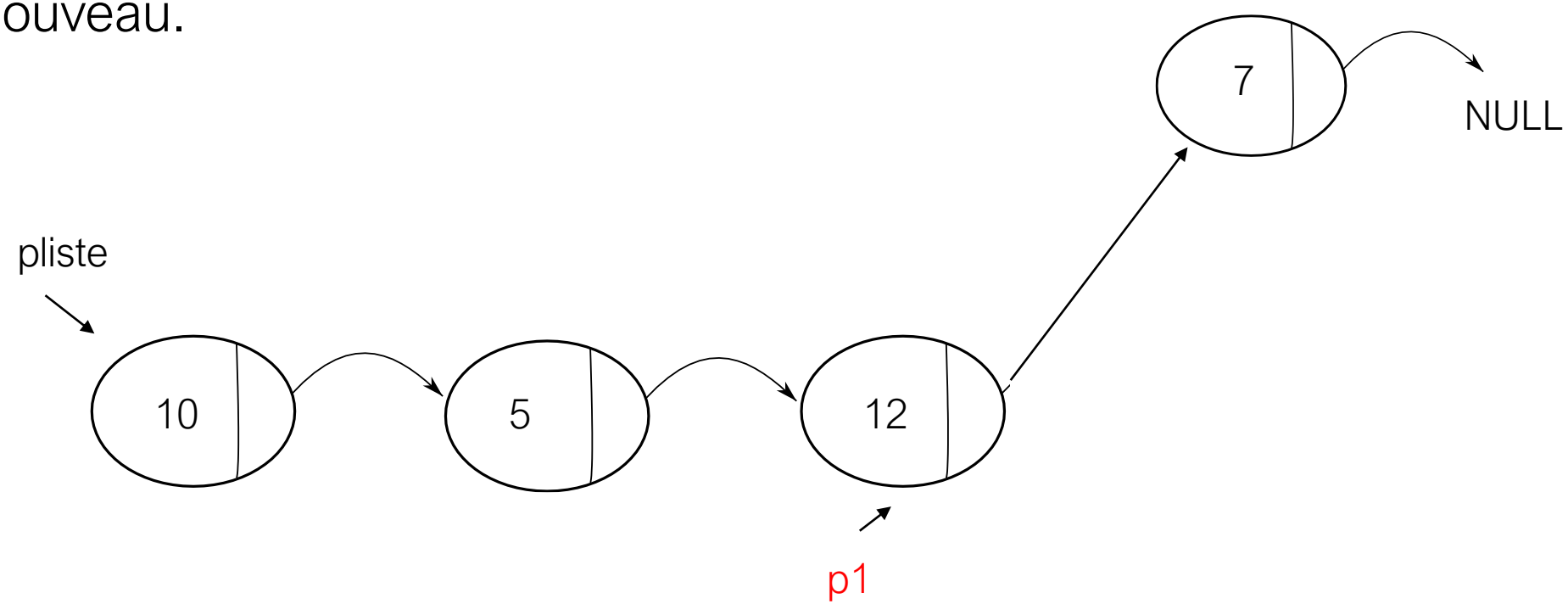
1. Ajout d'un Chainon en **fin de chaîne**
   a. Création du Chainon
   b. Parcours de la liste jusqu'au dernier Chainon





Listes chaînées : ajout de Chainon

1. Ajout d'un Chainon en **fin de chaîne**
   a. Création du Chainon
   b. Parcours de la liste jusqu'au dernier Chainon
   c. Faire pointer le suivant du dernier Chainon sur le nouveau.



Listes chaînées : ajout de Chainon

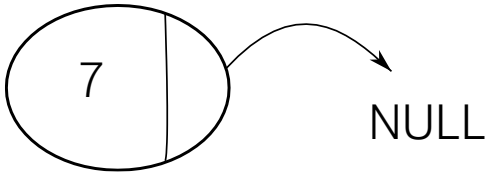
1. Ajout d'un Chainon en **fin de chaîne**
   a. Création du Chainon
   b. Parcours de la liste jusqu'au dernier Chainon
   c. Faire pointer le suivant du dernier Chainon sur le nouveau.

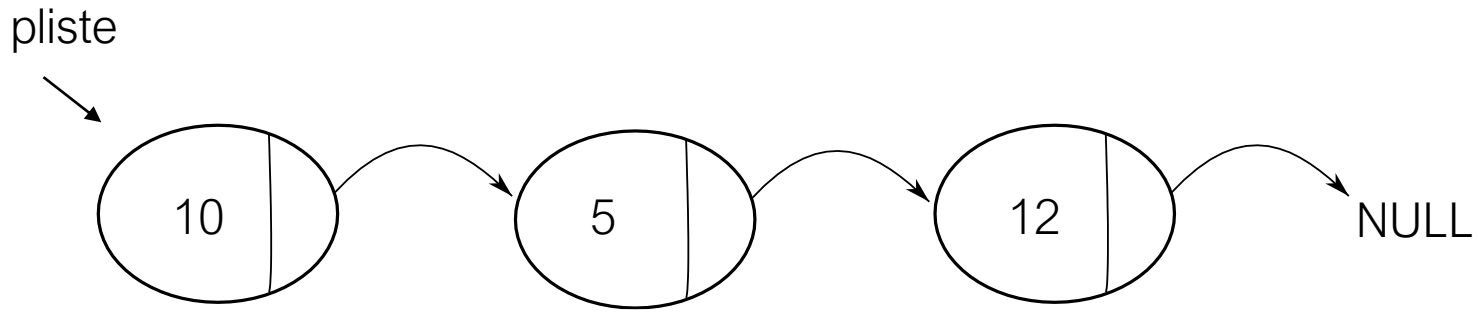
```
fonction insertFin(pliste : pointeur sur Chainon) : pointeur sur Chainon
VARIABLE
        nouveau, p1 : pointeur sur Chainon
DEBUT
        nouveau ← creationChainon()          // etape (a)
        p1 ← pliste
        TANT QUE(suivant(p1) ≠ NULL) FAIRE   // étape (b)
                p1 ← suivant(p1)
        FIN TANT QUE
        suivant(p1) ← nouveau                // étape (c)
        // par convention on retourne le début de liste
        // même si il n'a pas été modifié
        RETOURNER pliste
FIN
```

Complexité temporelle *O(N)*

Listes chaînées : ajout de Chainon

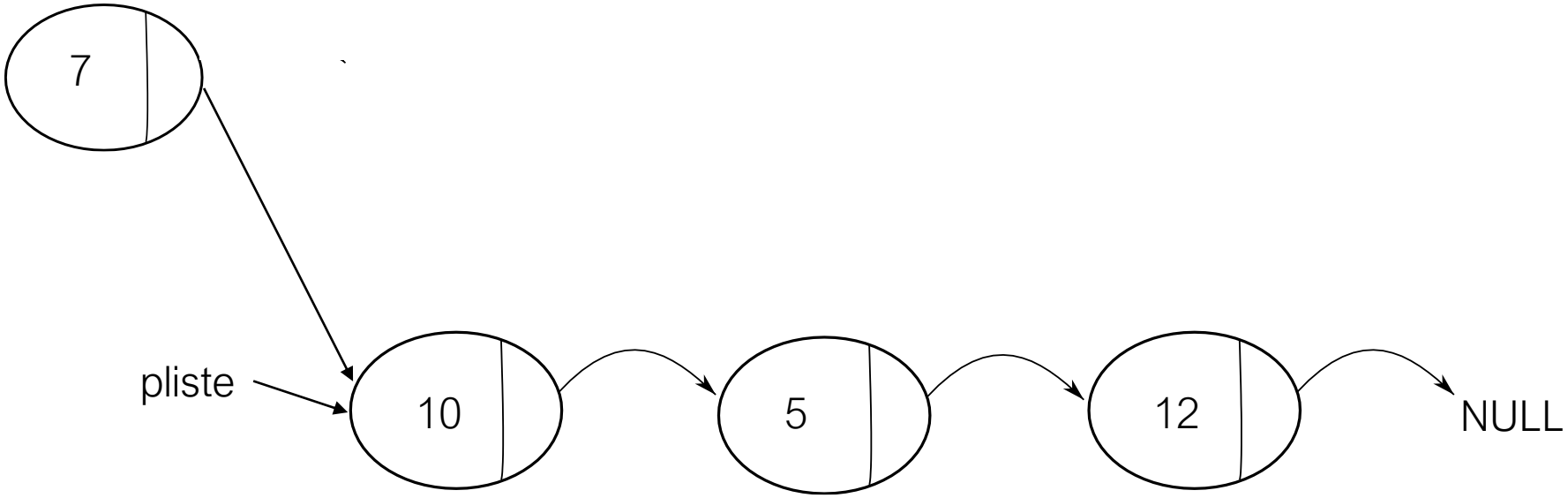
1. Ajout d'un Chainon en fin de chaîne
2. Ajout d'un Chainon en **début de chaîne**
   a. Création Chainon





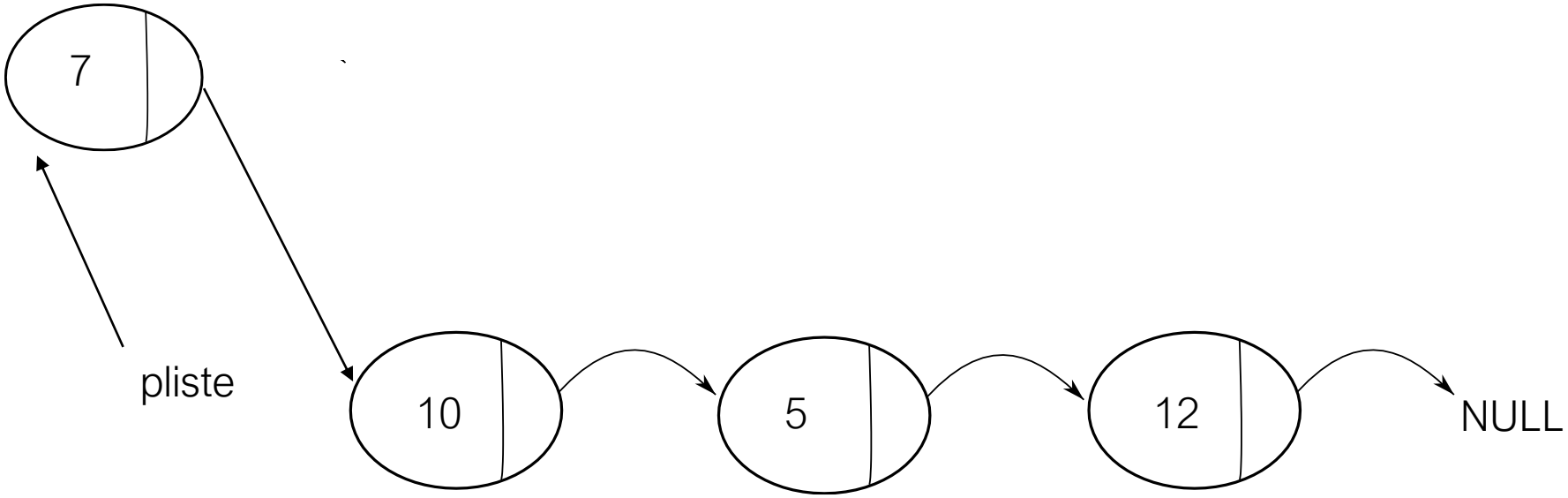
Listes chaînées : ajout de Chainon

1. Ajout d'un Chainon en fin de chaîne
2. Ajout d'un Chainon en **début de chaîne**
   a. Création Chainon
   b. Faire pointer le suivant du nouveau sur la chaîne



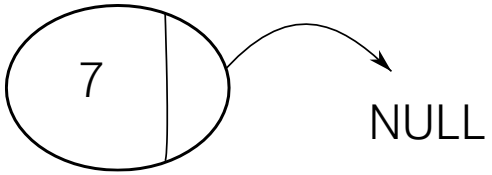
Listes chaînées : ajout de Chainon

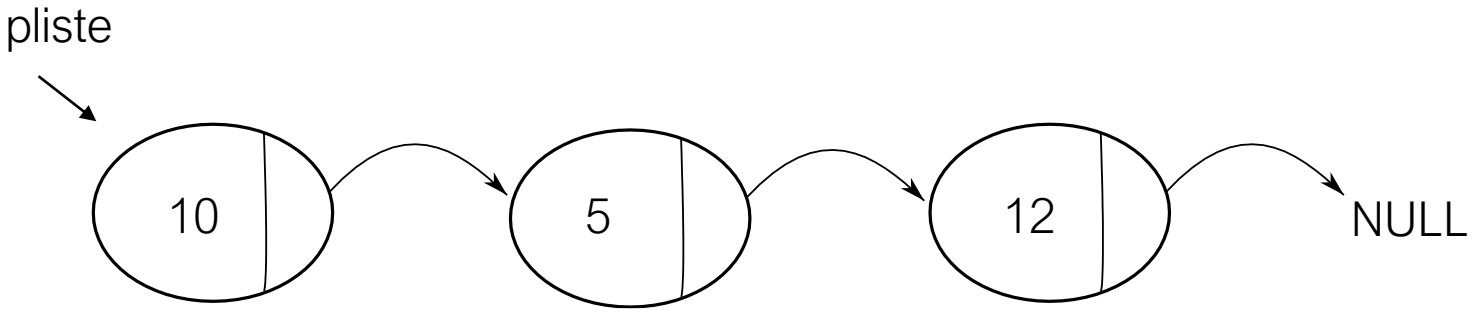
1. Ajout d'un Chainon en fin de chaîne
2. Ajout d'un Chainon en **début de chaîne**
   a. Création Chainon
   b. Faire pointer le suivant du nouveau sur la chaîne
   c. Faire pointer pliste sur le nouveau Chainon



Listes chaînées : ajout de Chainon

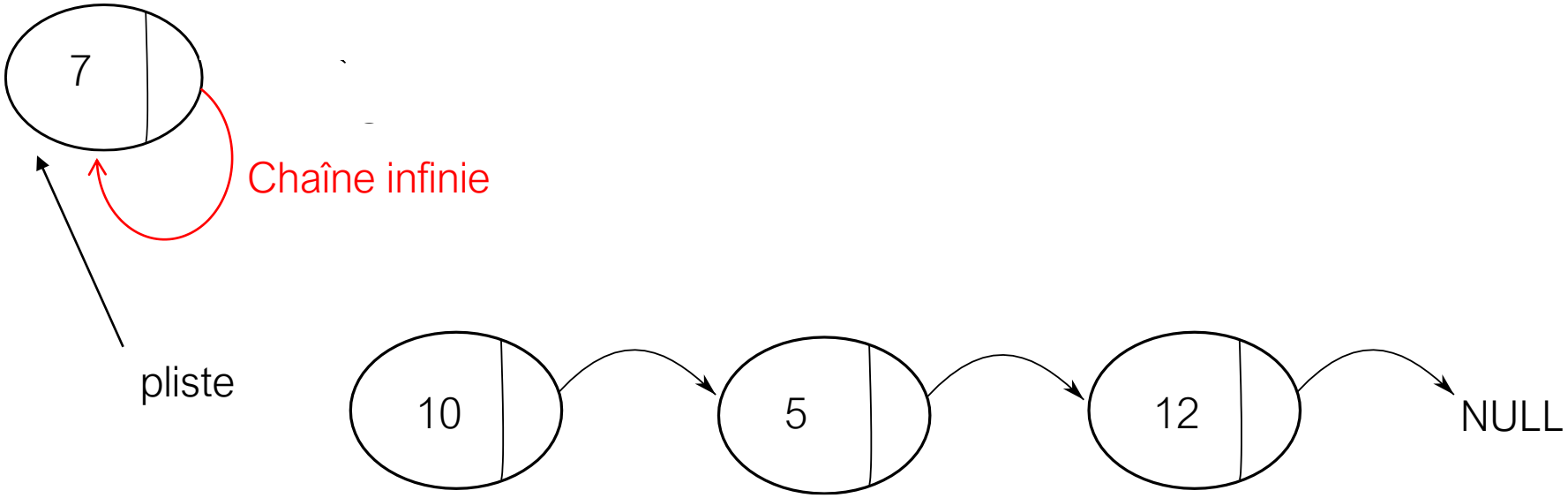
1. Ajout d'un Chainon en fin de chaîne
2. Ajout d'un Chainon en **début de chaîne**
   Attention à l'ordre !





Listes chaînées : ajout de Chainon

1. Ajout d'un Chainon en fin de chaîne
2. Ajout d'un Chainon en **début de chaîne**
   Attention à l'ordre !



Le reste de la liste est perdu !

Listes chaînées : ajout de Chainon

1. Ajout d'un Chainon en fin de chaîne
2. Ajout d'un Chainon en **début de chaîne**
   a. Création Chainon
   b. Faire pointer le suivant du nouveau sur la chaîne
   c. Faire pointer pliste sur le nouveau Chainon

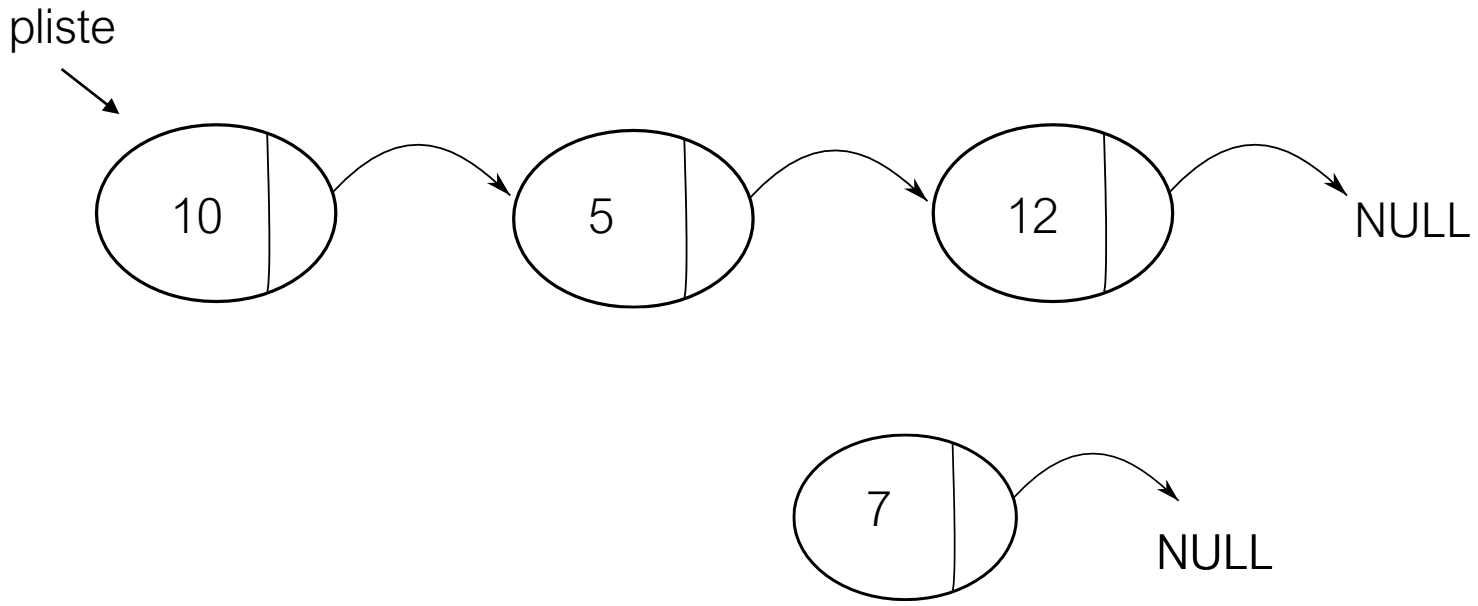
```
Fonction insertDebut( pliste : pointeur sur Chainon) : pointeur sur Chainon
VARIABLE
        nouveau : pointeur sur Chainon
DEBUT
        nouveau ← creationChainon()   // étape (a)
        suivant(nouveau) ← pliste     // étape (b)
        pliste ← nouveau              // étape (c)
        RETOURNER pliste
FIN
```

Complexité temporelle *O(1)*

Le pointeur sur le début de la liste ayant été modifié, il faut le retourner !!

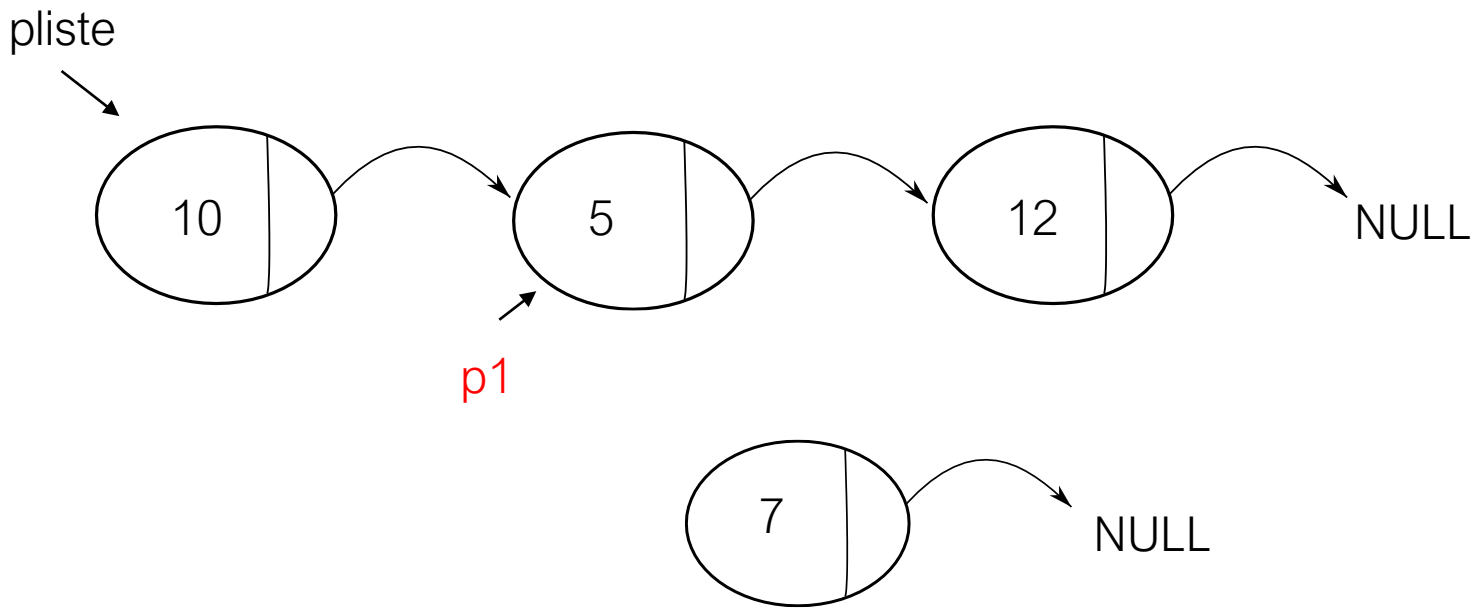
Listes chaînées : ajout de Chainon

1. Ajout d'un Chainon en fin de chaîne
2. Ajout d'un Chainon en début de chaîne
3. Ajout d'un Chainon en **milieu de chaîne**
   a. Création du nouveau Chainon



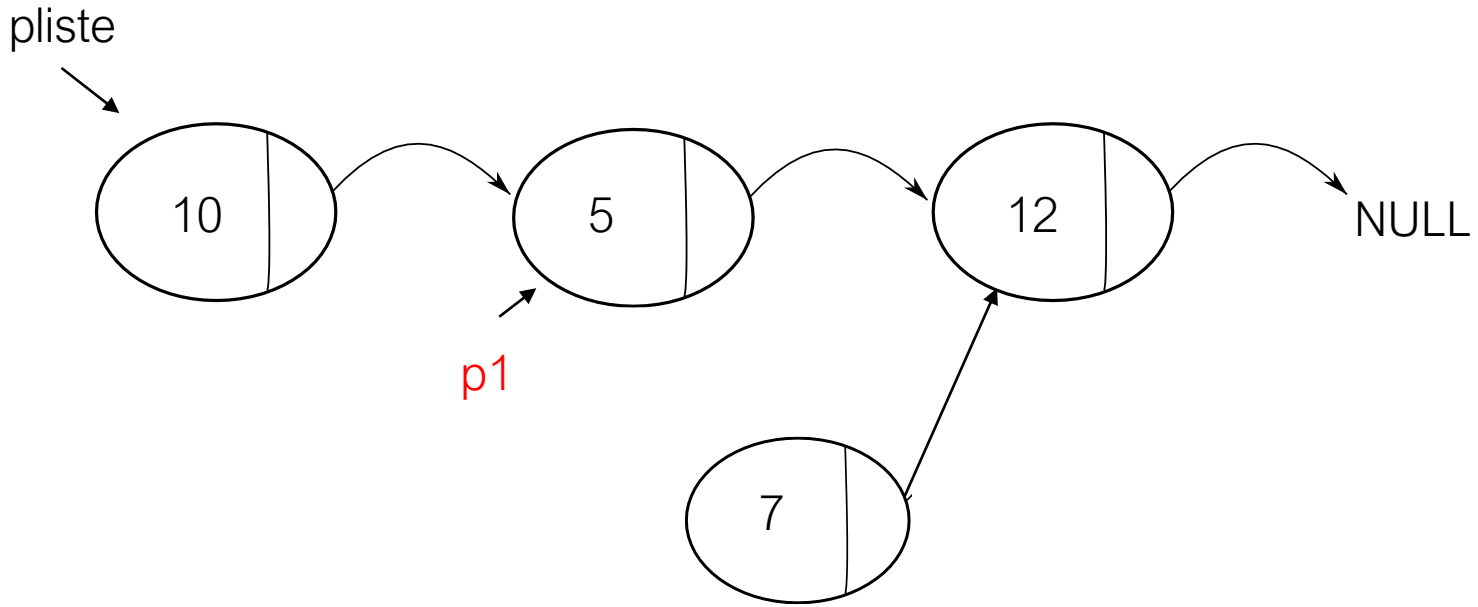
Listes chaînées : ajout de Chainon

1. Ajout d'un Chainon en fin de chaîne
2. Ajout d'un Chainon en début de chaîne
3. Ajout d'un Chainon en **milieu de chaîne**
   a. Création du nouveau Chainon
   b. Parcourir la liste jusqu'à Chainon devant précéder le nouveau



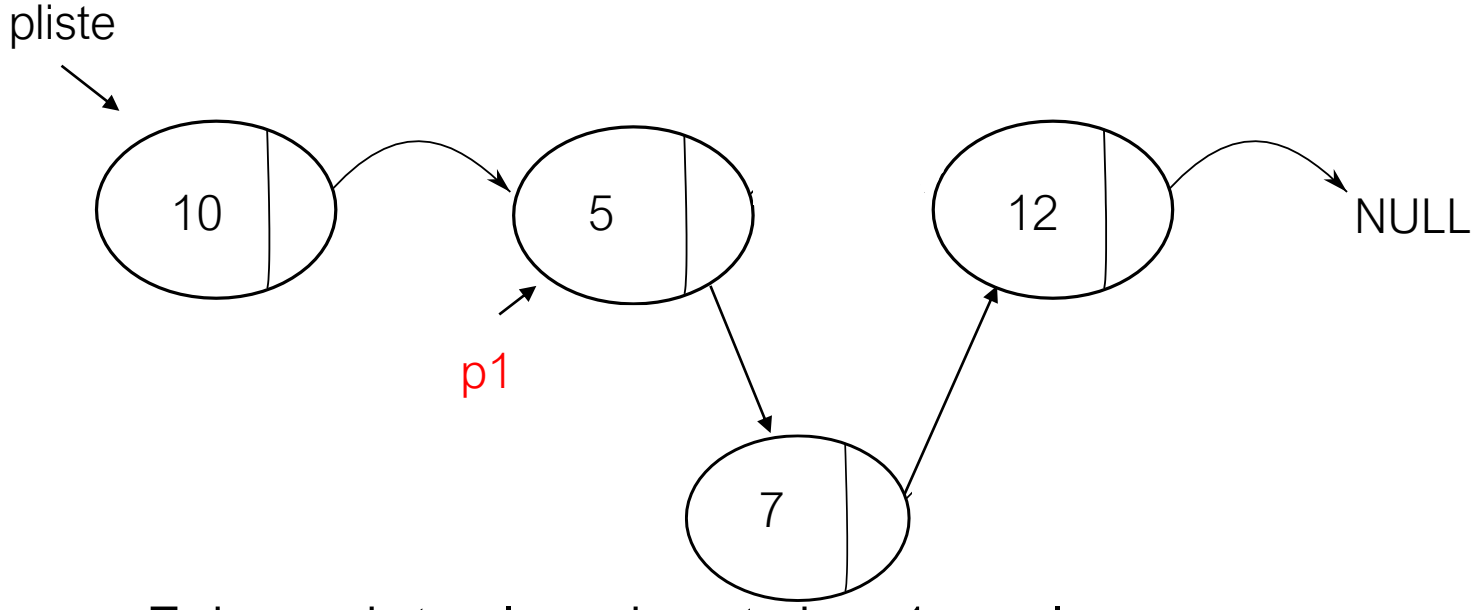
Listes chaînées : ajout de Chainon

1. Ajout d'un Chainon en fin de chaîne
2. Ajout d'un Chainon en début de chaîne
3. Ajout d'un Chainon en **milieu de chaîne**
   a. Création du nouveau Chainon
   b. Parcourir la liste jusqu'à Chainon devant précéder le nouveau
   c. Faire pointer le suivant du nouveau sur le suivant de p1



Listes chaînées : ajout de Chainon

1. Ajout d'un Chainon en fin de chaîne
2. Ajout d'un Chainon en début de chaîne
3. Ajout d'un Chainon en **milieu de chaîne**
   a. Création du nouveau Chainon
   b. Parcourir la liste jusqu'à Chainon devant précéder le nouveau
   c. Faire pointer le suivant du nouveau sur le suivant de p1

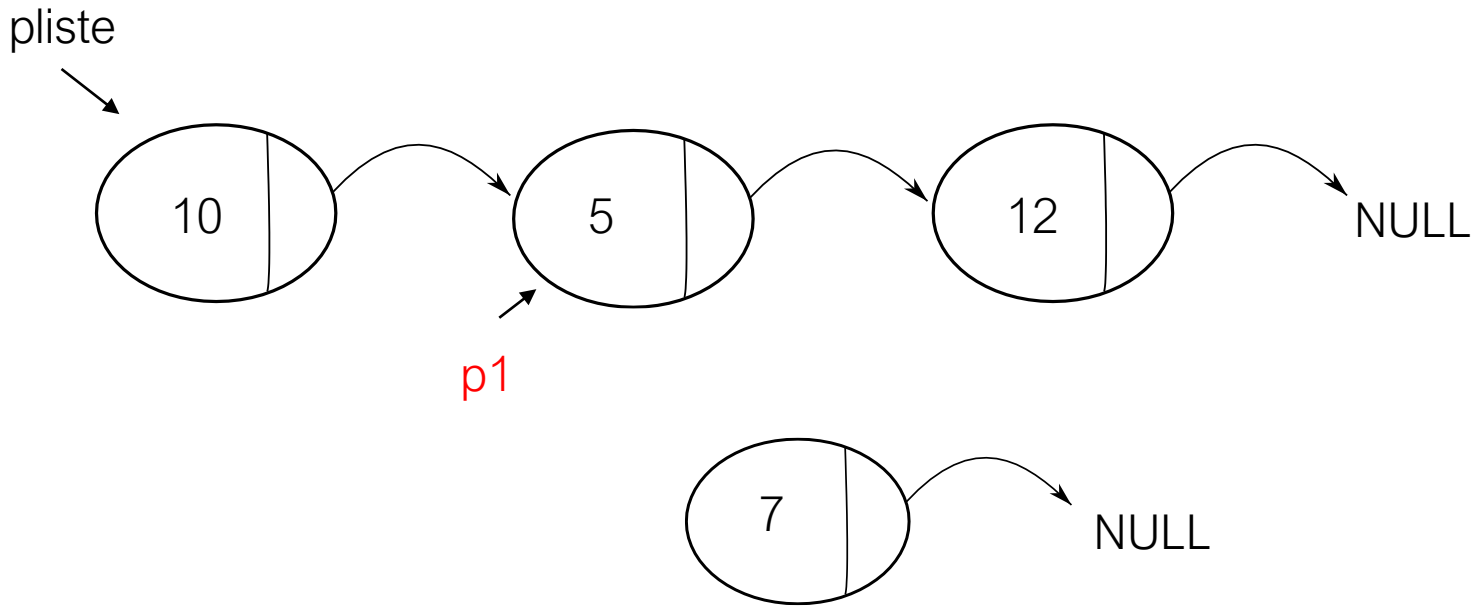


   d. Faire pointer le suivant de p1 sur le nouveau

Listes chaînées : ajout de Chainon

1. Ajout d'un Chainon en fin de chaîne
2. Ajout d'un Chainon en début de chaîne
3. Ajout d'un Chainon en **milieu de chaîne**

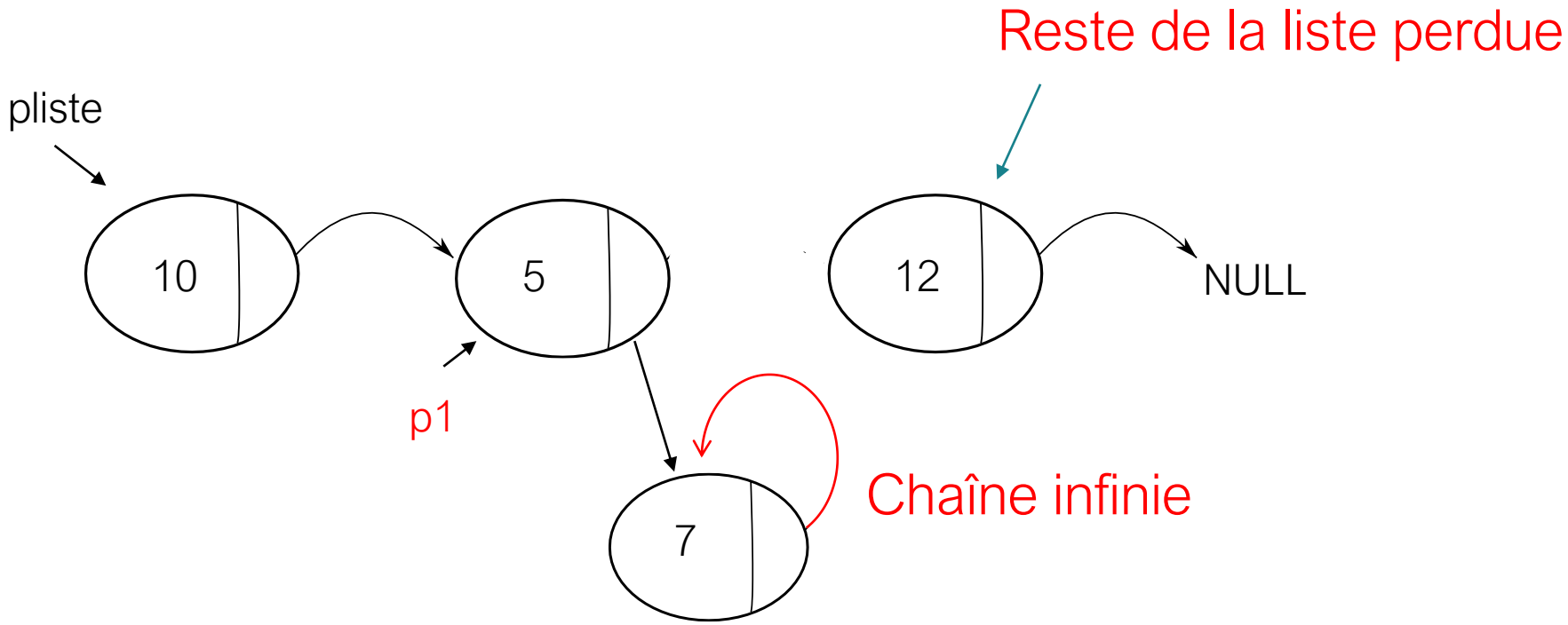
   Attention à l'ordre !



Listes chaînées : ajout de Chainon

1. Ajout d'un Chainon en fin de chaîne
2. Ajout d'un Chainon en début de chaîne
3. Ajout d'un Chainon en **milieu de chaîne**

   Attention à l'ordre !



Listes chaînées : ajout de Chainon

1. Ajout d'un Chainon en fin de chaîne
2. Ajout d'un Chainon en début de chaîne
3. Ajout d'un Chainon en **milieu de chaîne**
   a. Parcourir la liste jusqu'à Chainon devant précéder le nouveau
   b. Création du nouveau Chainon
   c. Faire pointer le suivant du nouveau sur le suivant de p1
   d. Faire pointer le suivant de p1 sur le nouveau

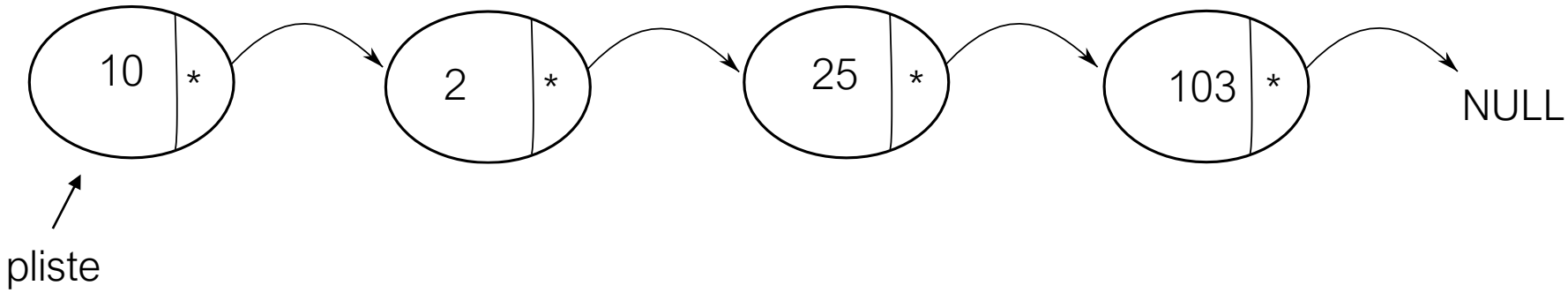
Listes chaînées : ajout de Chainon

a. Parcourir la liste jusqu'à Chainon devant précéder le nouveau
b. Création du nouveau Chainon
c. Faire pointer le suivant du nouveau sur le suivant de p1
d. Faire pointer le suivant de p1 sur le nouveau

```
fonction insertPos( pliste : pointeur sur Chainon, pos : entier) : pointeur sur Chainon
VARIABLE
        nouveau, p1 : pointeurs sur Chainon
        i : entier
DEBUT
        SI (pos EST EGAL A 0) OU (pliste EST EGAL A NULL) ALORS // insertion à l'indice 0
                pliste ← insertDebut(pliste)
        SINON
                p1 ← pliste
                POUR i DE 0 à pos FAIRE                         // étape (a)
                        SI p1 EST EGAL A NULL ALORS
                                ERREUR()                        // pos > nombre de chainons
                        SINON
                                p1 ← suivant(p1)
                        FIN SI
                FIN POUR
                nouveau ← creationChainon()                     // étape (b)
                suivant(nouveau) ← suivant(p1)                  // étape (c)
                suivant(p1) ← nouveau                           // étape (d)
        FIN SI
        RETOURNER pliste
FIN
```

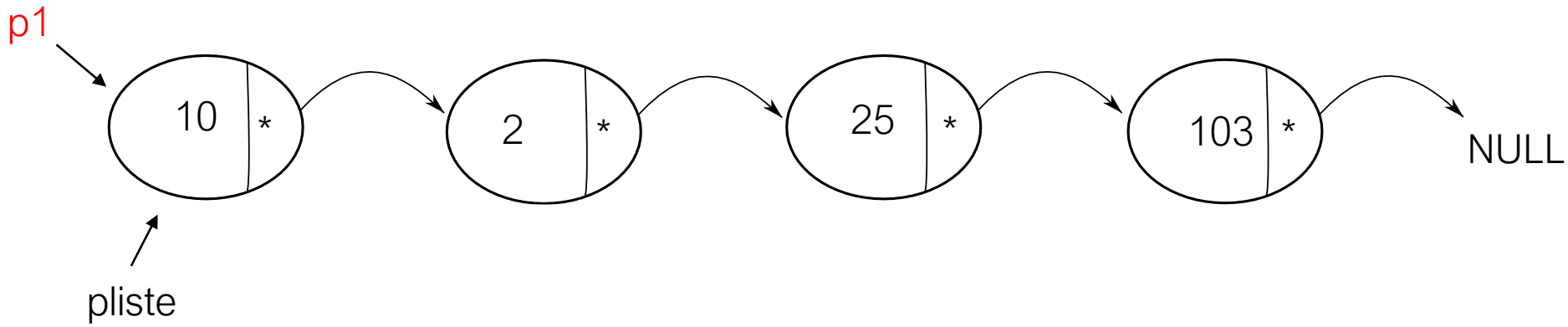
Listes chaînées : suppression de Chainon

1. Suppression d'un Chainon en début de chaîne



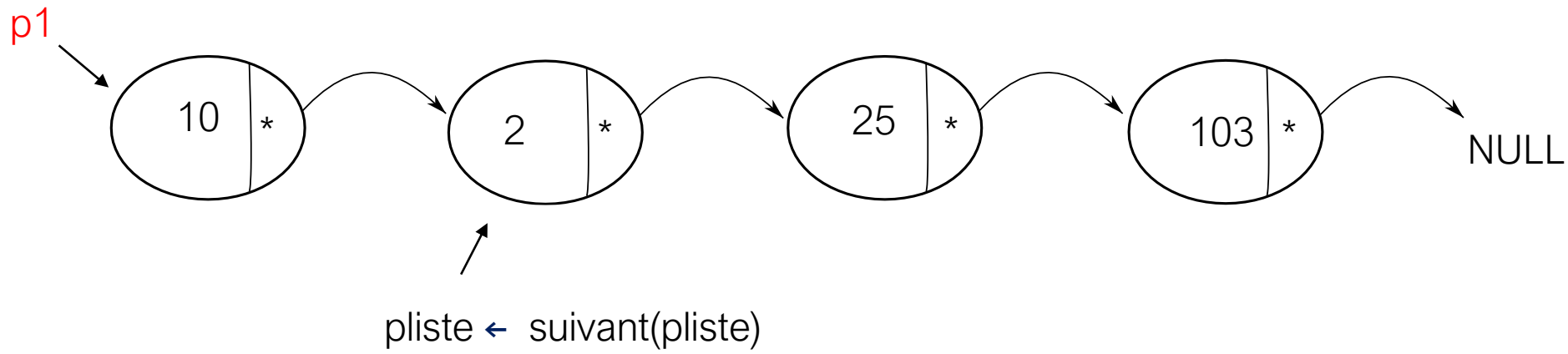
Listes chaînées : suppression de Chainon

1. Suppression d'un Chainon en début de chaîne
   a. Initialiser un pointeur sur le premier chaînon



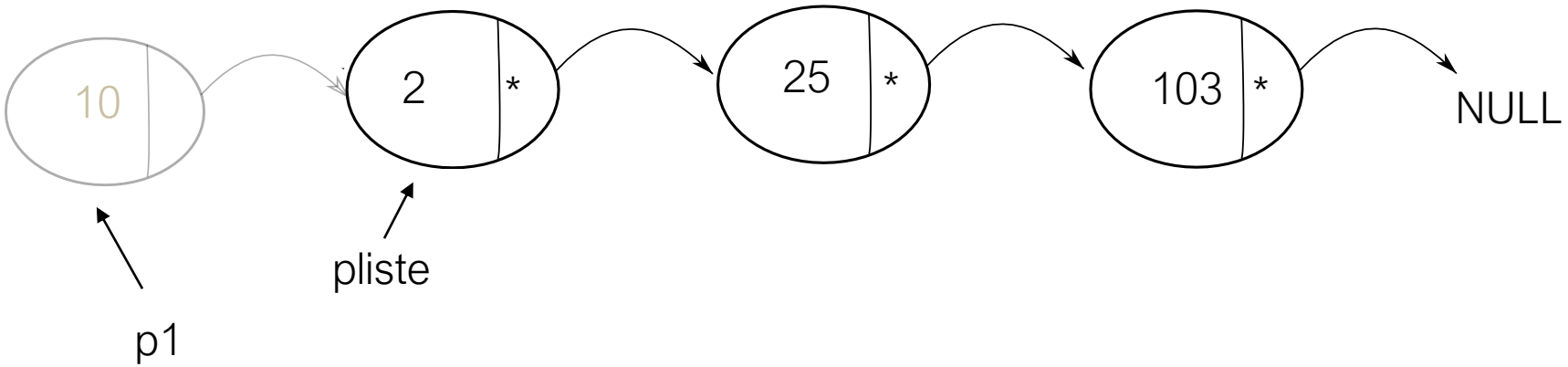
Listes chaînées : suppression de Chainon

1. Suppression d'un Chainon en début de chaîne
   a. Initialiser un pointeur sur le premier chaînon
   b. Faire pointer pliste sur le suivant



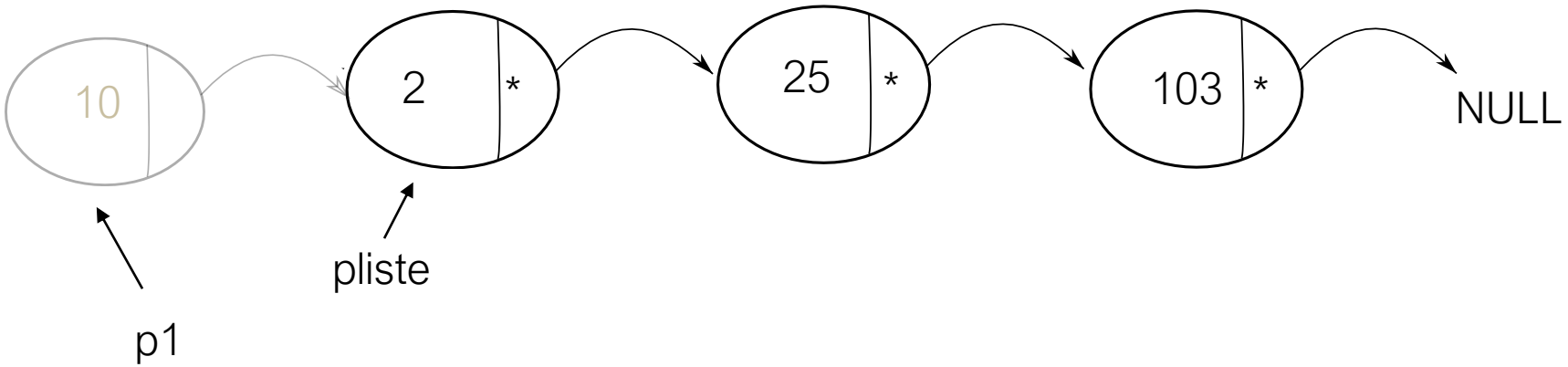
Listes chaînées : suppression de Chainon

1. Suppression d'un Chainon en début de chaîne
   a. Initialiser un pointeur sur le premier chaînon
   b. Faire pointer pliste sur le suivant
   c. Libérer le Chainon à supprimer



Listes chaînées : suppression de Chainon

1. Suppression d'un Chainon en début de chaîne
   a. Initialiser un pointeur sur le premier chaînon
   b. Faire pointer pliste sur le suivant
   c. Libérer le Chainon à supprimer



- Libérer la mémoire allouée au Chainon à supprimer permet d'éviter les fuites mémoires (RAM).

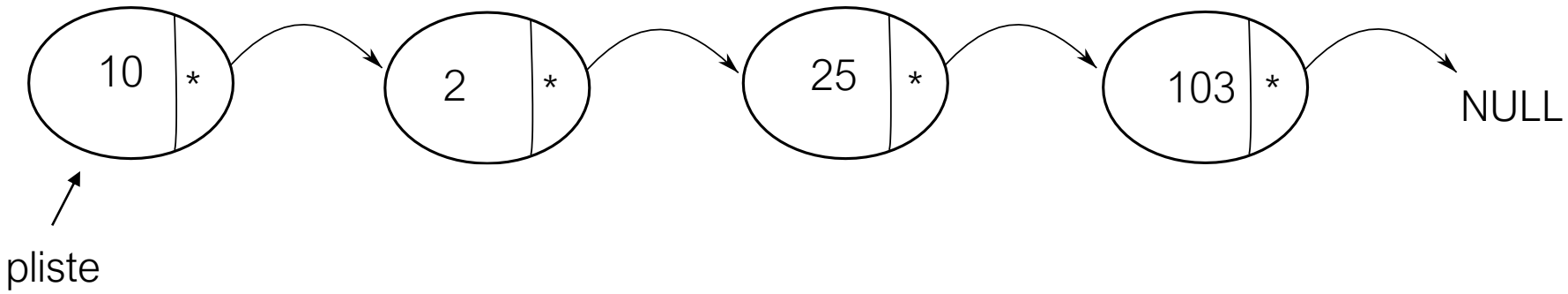
Listes chaînées : suppression de Chainon

1. Suppression d'un Chainon en début de chaîne
   a. Initialiser un pointeur sur le premier chaînon
   b. Faire pointer pliste sur le suivant
   c. Libérer le Chainon à supprimer

```
Chainon* suppDebut(Chainon * pliste){
        Chainon* p1;
        // on vérifie si la liste contient au moins un Chainon
        if(pliste==NULL){
                exit(1);
        }
        p1=pliste;              // étape (a)
        pliste=p1->suivant;     // étape (b)
        free(p1);               // étape (c) : 'libérer(p1)' en pseudo-code
        return pliste;
}
```

Listes chaînées : suppression de Chainon

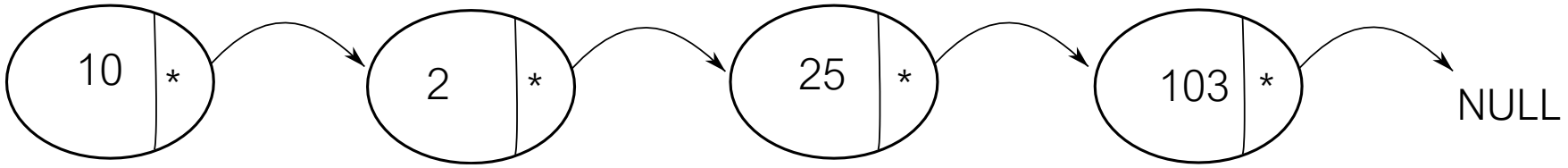
1. Suppression d'un Chainon en début de chaîne
2. Exercice : suppression d'un Chainon en milieu de chaîne



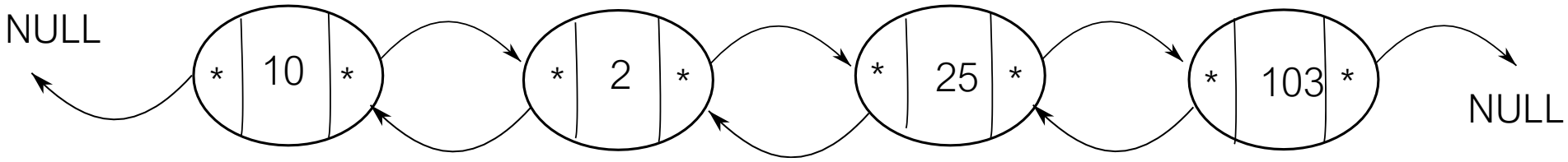
III. Listes doublement chaînées

Listes doublement chaînées

- Les listes que nous venons d'étudier sont dites simplement chaînées.

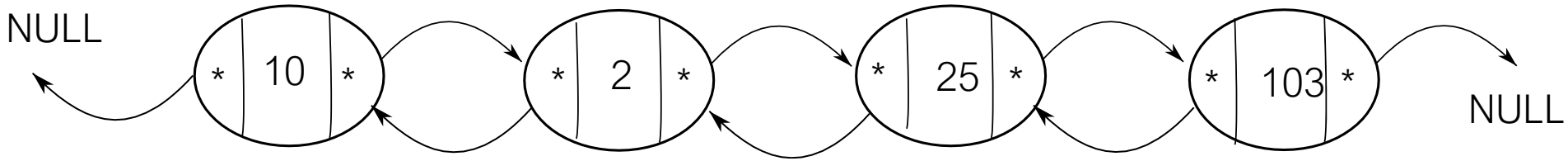


- On peut également construire des listes doublement chaînées.



Listes doublement chaînées

- Les Chainons des listes doublement chaînées possèdent deux pointeurs : l'un indiquant le Chainon suivant, l'autre indiquant le précédent.

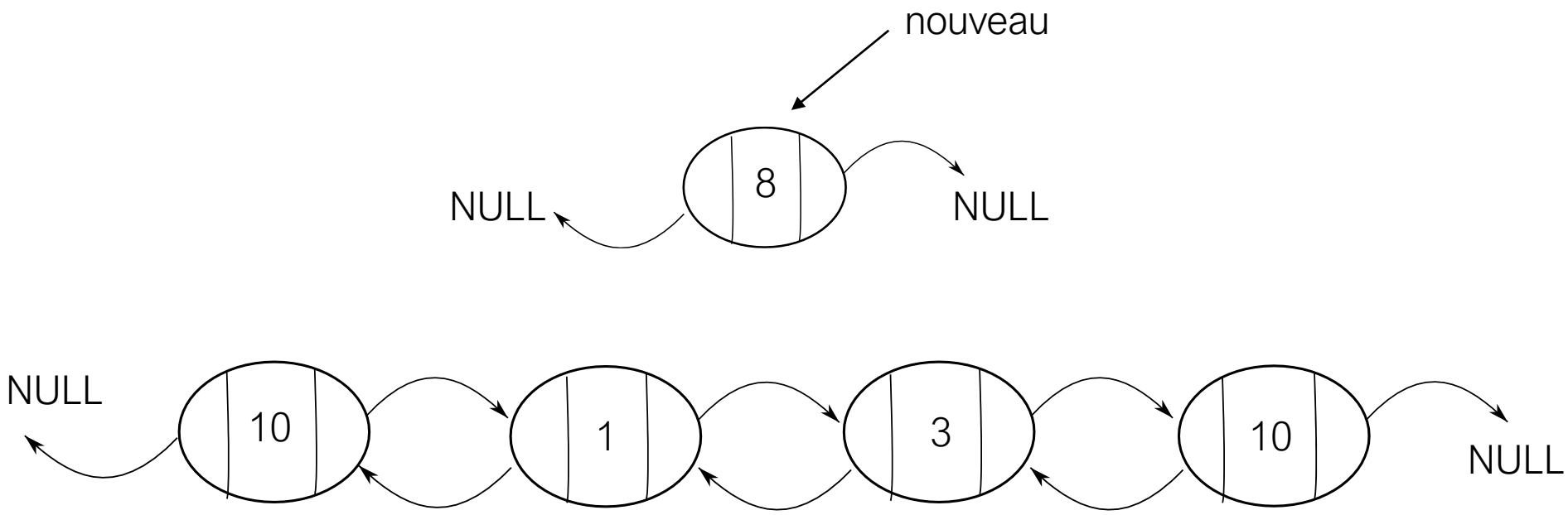



```
Structure Chainon :
        elmt : Element
        suivant : pointeur sur structure Chainon
        precedent : pointeur sur structure Chainon
```

- Ces chaînes sont plus compliquées à implémenter mais permettent de parcourir la liste dans les deux sens.

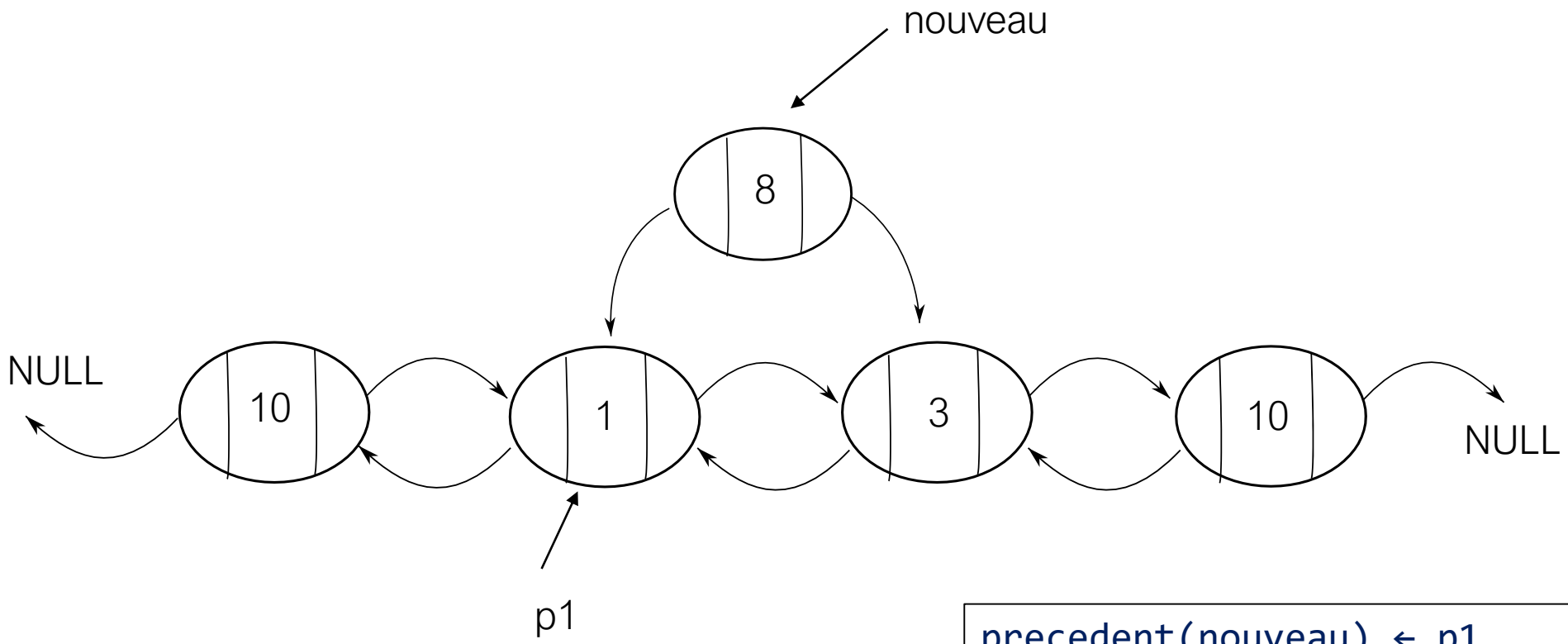
Listes doublement chaînées

- Insertion dans une liste doublement chaînée :



Listes doublement chaînées

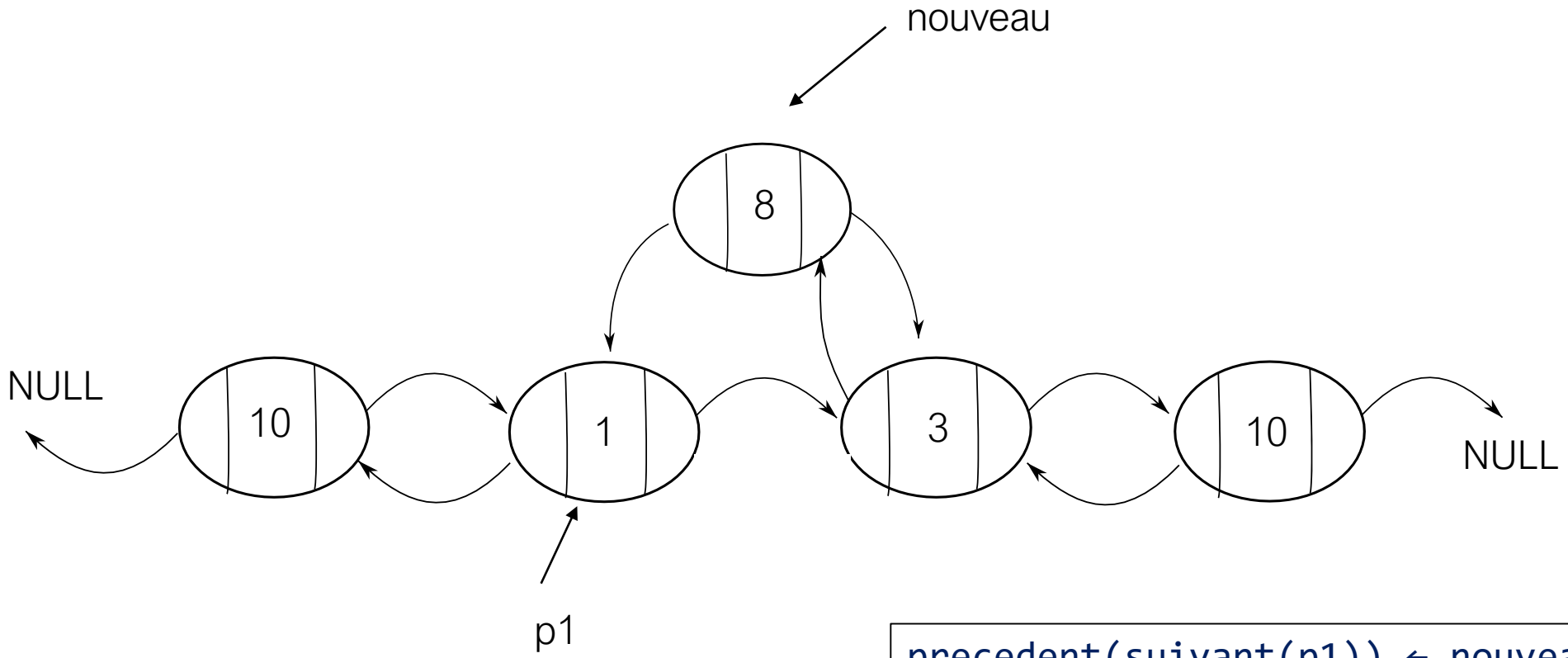
- Insertion dans une liste doublement chaînée :




```
precedent(nouveau) ← p1
suivant(nouveau) ← suivant(p1)
```

Listes doublement chaînées

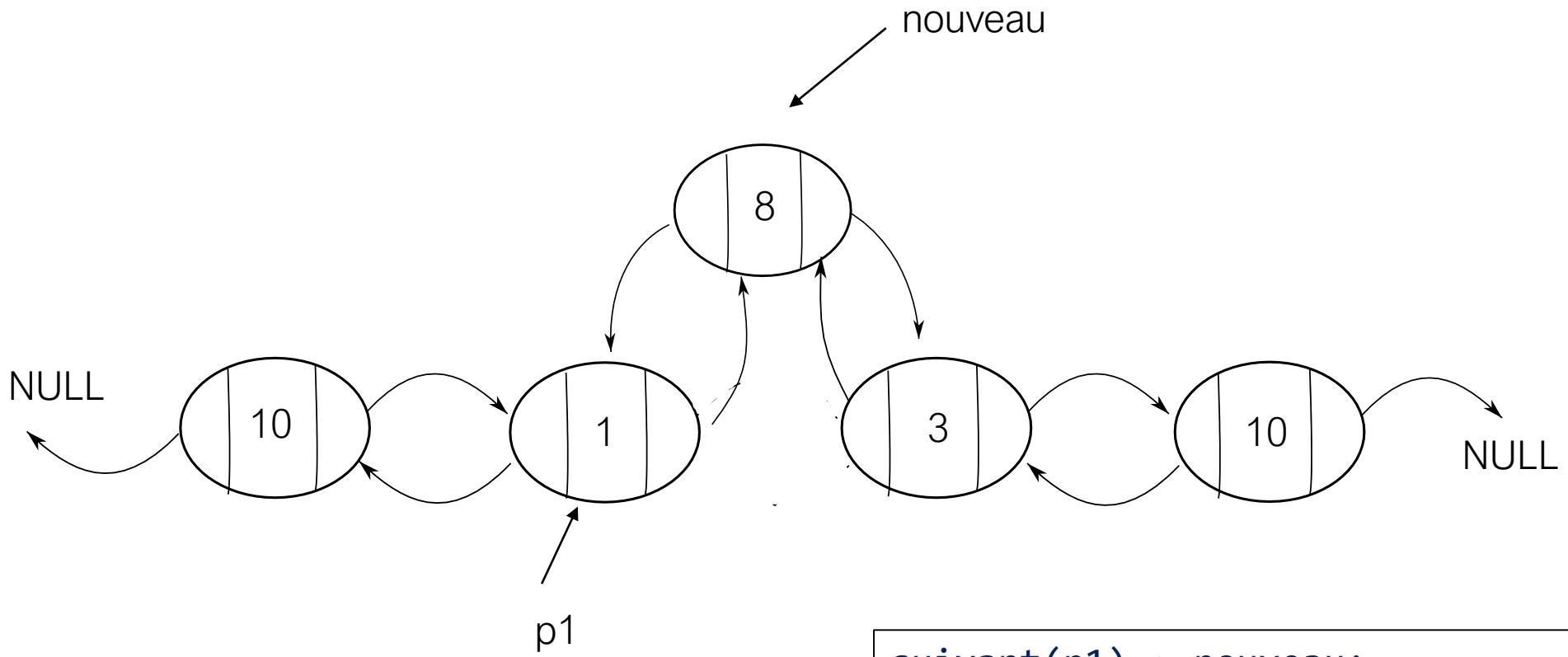
- Insertion dans une liste doublement chaînée :




```
precedent(suivant(p1)) ← nouveau
```

Listes doublement chaînées

- Insertion dans une liste doublement chaînée :




```
suivant(p1) ← nouveau;
```

IV. Listes chaînées ou tableaux ?

Comparaison liste chaînée/tableau

	Tableau	Liste chaînée
Nombre d'éléments	Fixe	Illimité et non fixé

Comparaison liste chaînée/tableau

	Tableau	Liste chaînée
Nombre d'éléments	Fixe	Illimité et non fixé
Données contiguës	Oui	Non

Comparaison liste chaînée/tableau

	Tableau	Liste chaînée
Nombre d'éléments	Fixe	Illimité et non fixé
Données contiguës	Oui	Non
Complexité spatiale :		

Comparaison liste chaînée/tableau

	Tableau	Liste chaînée
Nombre d'élément	Fixe	Illimité et non fixé
Données contiguës	Oui	Non
Complexité spatiale :	Nécessite l'espace pour les éléments + un pointeur	Chaque élément de la liste a un pointeur (ou plus) associé.

Comparaison liste chaînée/tableau

	Tableau	Liste chaînée
Nombre d'élément	Fixe	Illimité et non fixé
Données contiguës	Oui	Non
Complexité spatiale :	Nécessite l'espace pour les éléments + un pointeur	Chaque élément de la liste a un pointeur (ou plus) associé.
Complexité temporelle :		
Ajout/suppression d'un élément		

Comparaison liste chaînée/tableau

	Tableau	Liste chaînée
Nombre d'élément	Fixe	Illimité et non fixé
Données contiguës	Oui	Non
Complexité spatiale :	Nécessite l'espace pour les éléments + un pointeur	Chaque élément de la liste a un pointeur (ou plus) associé.
Complexité temporelle :		
Ajout/suppression d'un élément	Décalage de tous les éléments O(n)	O(1) en bout de chaîne O(n) sinon
Accès à un élément		

Comparaison liste chaînée/tableau

	Tableau	Liste chaînée
Nombre d'élément	Fixe	Illimité et non fixé
Données contiguës	Oui	Non
Complexité spatiale :	Nécessite l'espace pour les éléments + un pointeur	Chaque élément de la liste a un pointeur (ou plus) associé.
Complexité temporelle :		
Ajout/suppression d'un élément	Décalage de tous les éléments O(n)	O(1) en bout de chaîne O(n) sinon
Accès à un élément	Accès direct O(1)	O(1) en bout de chaîne O(n) sinon

Comparaison liste chaînée/tableau

	Tableau	Liste chaînée
Nombre d'élément	Fixe	Illimité et non fixé
Données contiguës	Oui	Non
Complexité spatiale :	Nécessite l'espace pour les éléments + un pointeur	Chaque élément de la liste a un pointeur (ou plus) associé.
Complexité temporelle :		
Ajout/suppression d'un élément	Décalage de tous les éléments O(n)	O(1) en bouts de chaîne O(n) sinon
Accès à un élément	Accès direct O(1)	O(1) en bouts de chaîne O(n) sinon

✓ Les listes chaînées ne sont pas toujours la meilleure solution !

Pour résumer

- Une liste chaînée est constituée d'éléments et de pointeurs associés indiquant l'emplacement mémoire de l'élément suivant (et/ou précédent).
- Il faut donc bien gérer les structures et les pointeurs !!
- Ce principe permet à la liste chaînée de ne pas avoir un nombre fixé d'éléments, contrairement au tableau.
- La liste chaînée simplifie également certaines opérations par rapport au tableau. Elle n'est cependant pas toujours la meilleure solution.
- Le principe des listes chaînées est à la base de nombreux autres objets informatiques … et à la base de votre programme d'algorithmie de deuxième année !