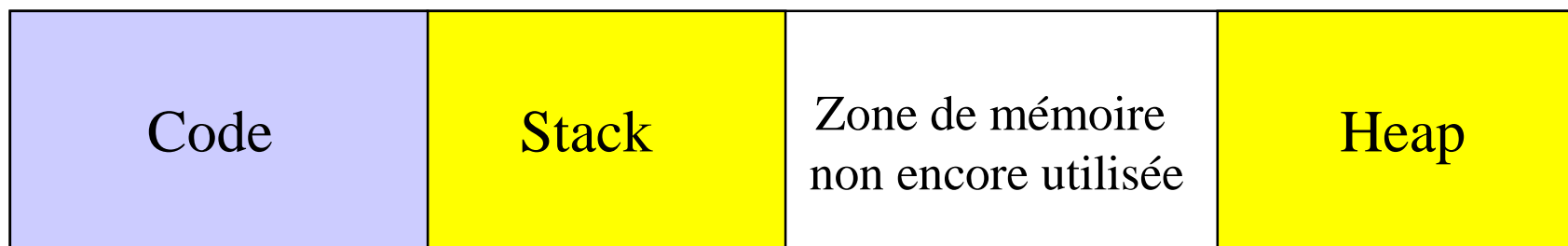


# Chapitre 14: Les listes (structures dynamiques)

- Contenu:
  - Rappel sur la gestion de la mémoire
  - Notion de liste
    - Liste simple
    - Liste circulaire bidirectionnelle
  - Pile et File
    - Concepts
    - Implantation sous forme de liste
  - Liste généralisée
  - Applications

# Rappel: gestion de la mémoire lors de l'exécution

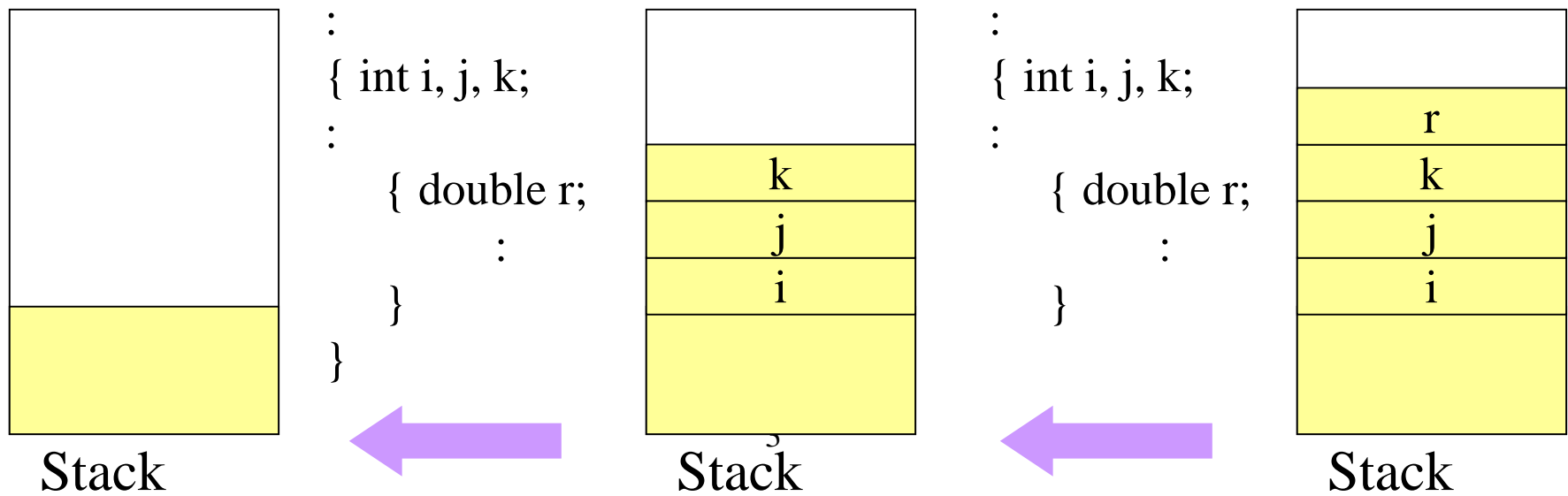
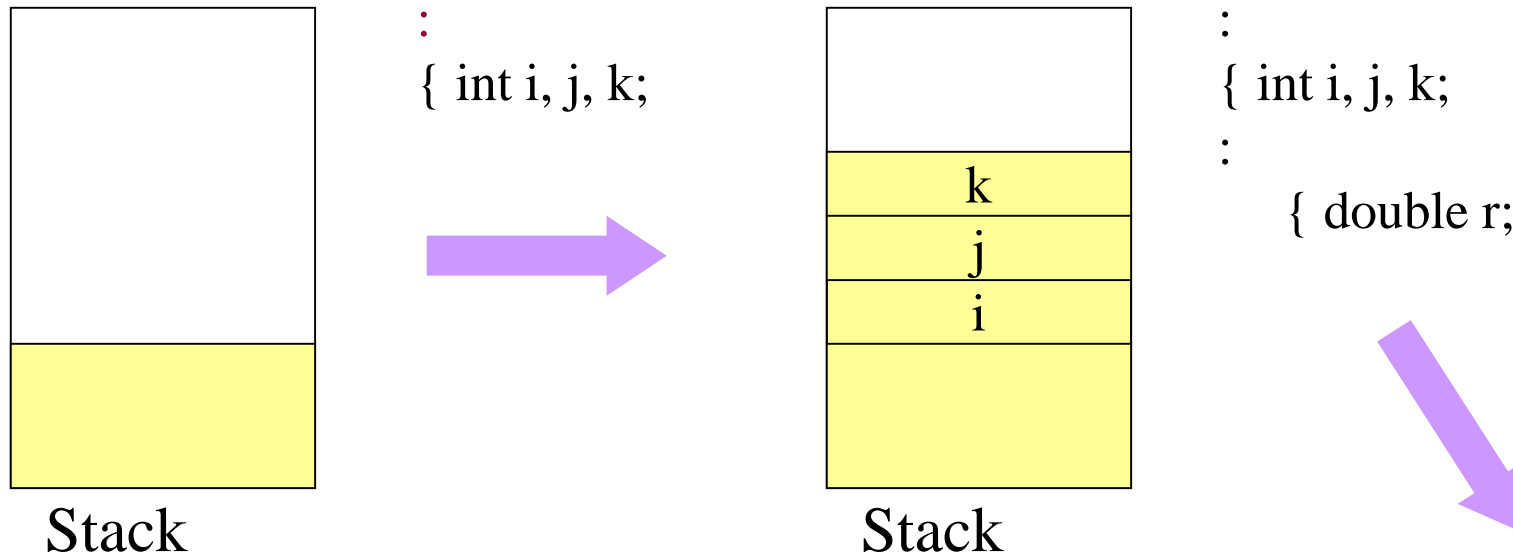
- La mémoire est subdivisée en 3 parties:
  - La **partie statique** contenant principalement le code des fonctions
  - La **Pile (Stack) run time** qui contient principalement les variables créés statiquement
  - Le **Tas (Heap) run time** qui contient les variables créées dynamiquement (par un new)



Sommet du stack

Limite de la zone utilisée par le heap  
(il existe une freelist de zone mémoire allouable)

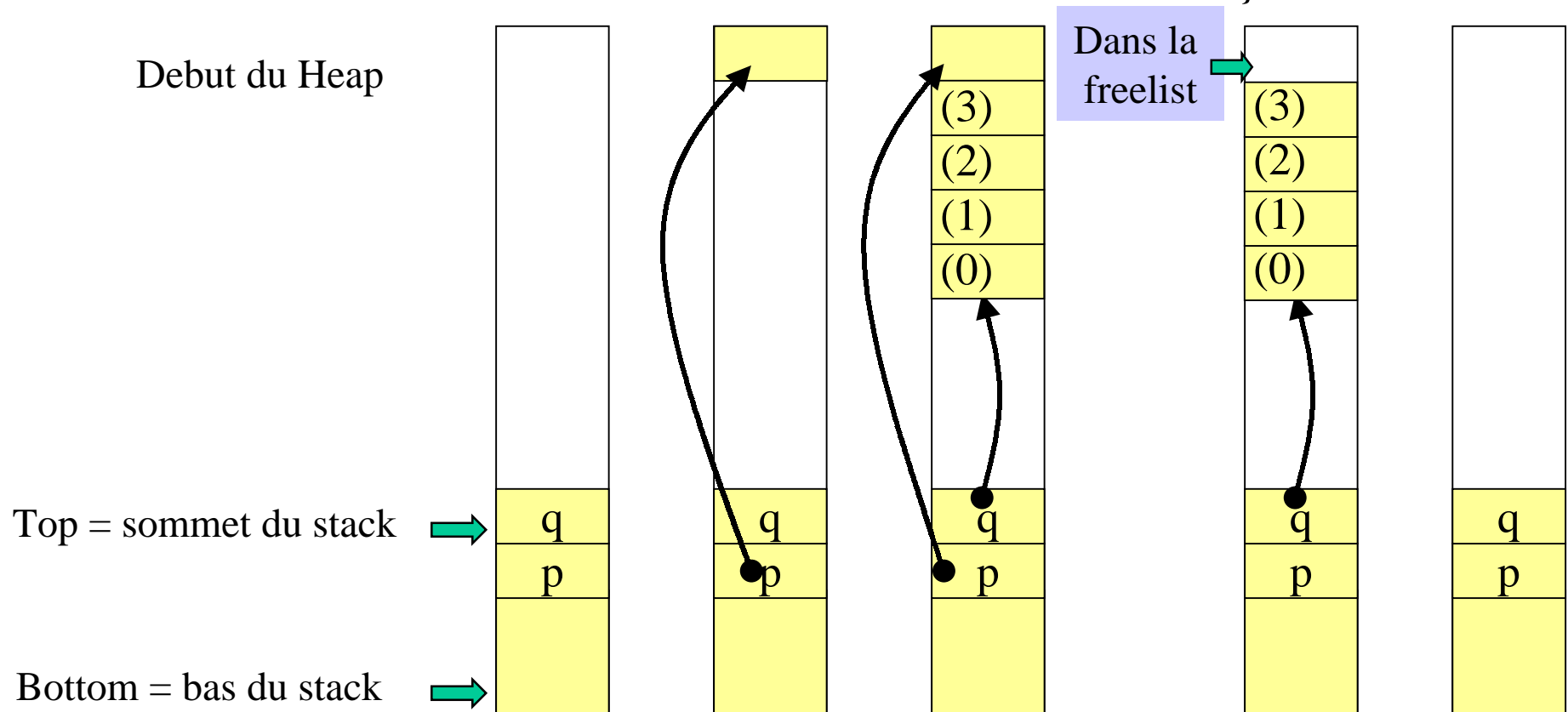
# Gestion mémoire d'entités statiques



# Gestion mémoire d'entités dynamiques

Un petit exemple

```
:  
{ int *p;  
  double *q;  
  p = new int;  
  q = new double[4];  
  delete p;  
  delete[] q;  
}
```



# Gestion de structures dynamiques

- **But:**

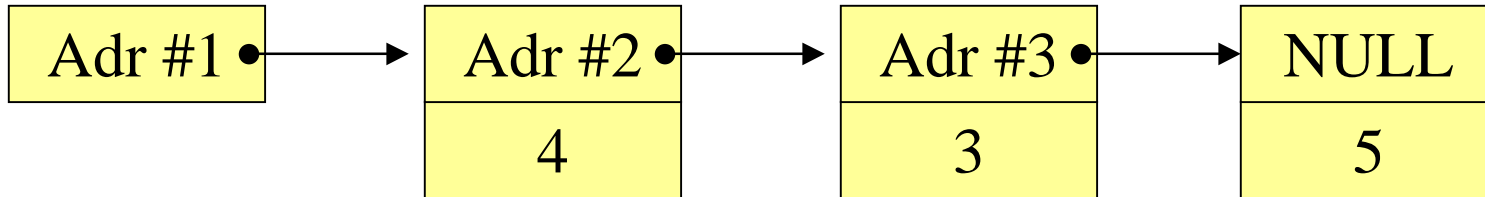
gérer des structures de données contenant plusieurs entités et dont le nombre d'entités varie tout au long de l'exécution du programme

- **Exemple:**

- La liste d'éléments:

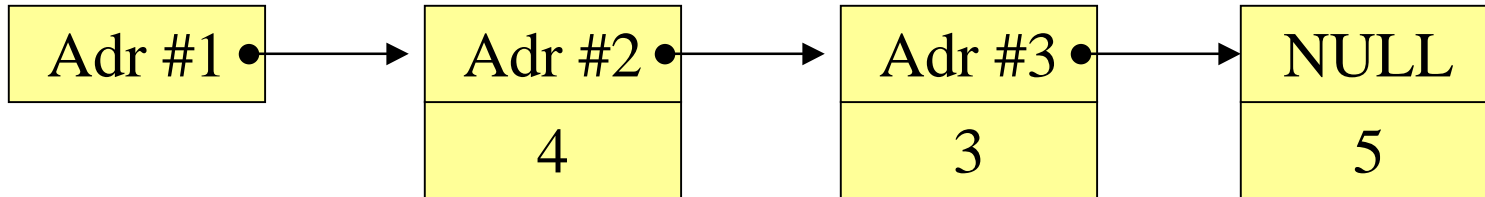
- $\langle \rangle$  = liste vide
    - $\langle 7 \rangle$  : on ajoute 7
    - $\langle 7, 21 \rangle$  : on ajoute 21
    - $\langle 3, 7, 21 \rangle$  : on ajoute 3
    - $\langle 7, 21 \rangle$  : on supprime 3
    - $\langle 5, 7, 21 \rangle$  : on ajoute 5
    - ...

## Implémentation : les listes chaînées



- Qu'est-ce qu'une liste chaînée ? Une collection d'éléments possédant chacun deux composantes : un pointeur vers le prochain élément de la liste et une « valeur » de n'importe quel type.

# Listes chaînées



```
class elem;  
typedef elem* liste;
```

```
class elem {  
public:  
    TypeOfInfo    info;  
    liste    next;  
    elem():info(0),next(NULL){ }  
    elem(TypeOfInfo i, liste n):info(i),next(n) { }  
};  
  
main() {  
    liste first = new elem(5, NULL);  
    first = new elem(3,first);  
    first = new elem(4,first);  
}
```

```
// alternative  
class elem {  
public:  
    TypeOfInfo info;  
    elem*    next;  
    elem():info(0),next(NULL){ }  
    elem(TypeOfInfo i, elem* n):info(i),next(n){ }  
};  
7 typedef elem* liste;  
...
```

# Listes chaînées

- Parcours

```
void Parcours(liste tete)
{
    for(liste p=tete; p != NULL; p= p-> next)
        cout << p->info << endl;
}
```

- Recherche d'un élément (renvoie le vrai/faux)

```
bool Contient(liste tete, TypeOfInfo x) {
    liste p;
    for(p=tete; p!=NULL && p->info != x; p = p-> next);
    return p!=NULL;
}
```



# Listes chaînées

- Recherche d'un élément (renvoie le pointeur)

```
liste Element(liste tete, TypeOfInfo x) {  
    liste p;  
    for(p=tete; p!=NULL && p->info != x; p = p-> next);  
    return p;  
}
```

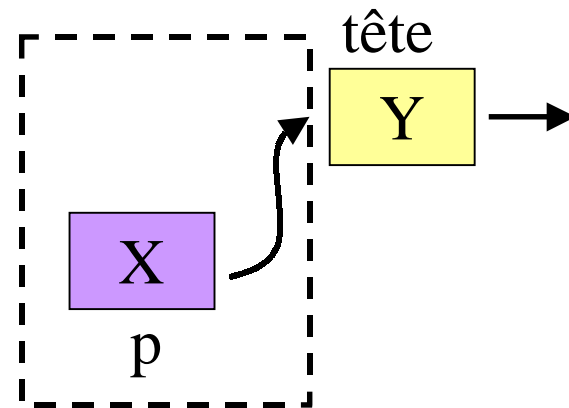
# Listes chaînées

- Insertion en tête

```
void InsertionEnTete(liste &tete, TypeOfInfo x) {  
    tete = new elem(x,tete);  
}
```

- Insertion en tête (2ième version : décomposition des opérations)

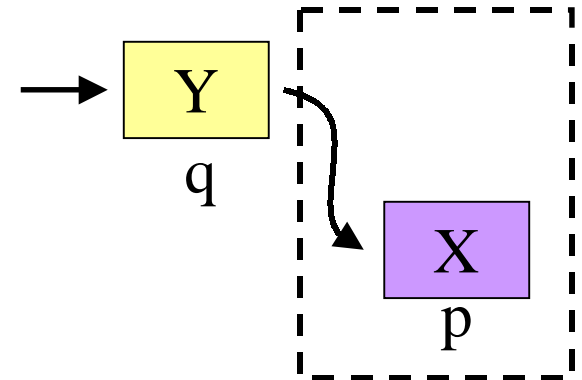
```
void InsertionEnTeteV2(liste& tete, TypeOfInfo x) {  
    liste p = new elem;  
    p -> info = x;  
    p -> next = tete;  
    tete = p;  
}
```



# Listes chaînées

- Insertion en fin de liste (1)

```
void InsertionEnFin(liste &tete, TypeOfInfo x) {  
    if (tete == NULL)    // dans une liste vide  
        tete = new elem(x,NULL);  
    else {                // dans une liste non vide  
        liste q = tete;  
        liste p = tete->next;  
        while (p != NULL) {  
            q = p;  
            p = p -> next;  
        }  
        q -> next = new elem(x,NULL);  
    }  
}
```



# Listes chaînées

- Insertion en fin de liste (2)

```
void InsertionEnFinV2(liste &tete, TypeOfInfo x) {  
    liste q = NULL;  
    liste p = tete;  
    while (p != NULL) {  
        q = p;  
        p = p -> next;  
    }  
    p = new elem(x,NULL);  
    if (q == NULL)  
        tete = p; // liste vide  
    else  
        q -> next = p;  
}
```

# Listes chaînées

- Insertion en fin de liste (3)

```
void InsertionEnFinV3(liste& tete, TypeOfInfo x) {  
    liste p;  
    if (tete == NULL) {  
        tete = new elem;  
        p = tete;  
    }  
    else {  
        for (p = tete; p->next != NULL; p = p->next);  
        p->next = new elem;  
        p = p->next;  
    }  
    p->info = x;  
    p->next = NULL;  
}
```

# Listes chaînées

- Insertion dans une liste triée (1)

```
void InsertionTrie(liste & tete, TypeOfInfo x) {  
    if (tete == NULL || tete -> info > x)  
        tete = new elem(x,tete);  
    else {  
        liste q = tete;  
        liste p = tete->next;  
        while (p != NULL && p->info <= x) {  
            q = p;  
            p = p -> next;  
        }  
        q -> next = new elem(x,p);  
    }  
}
```

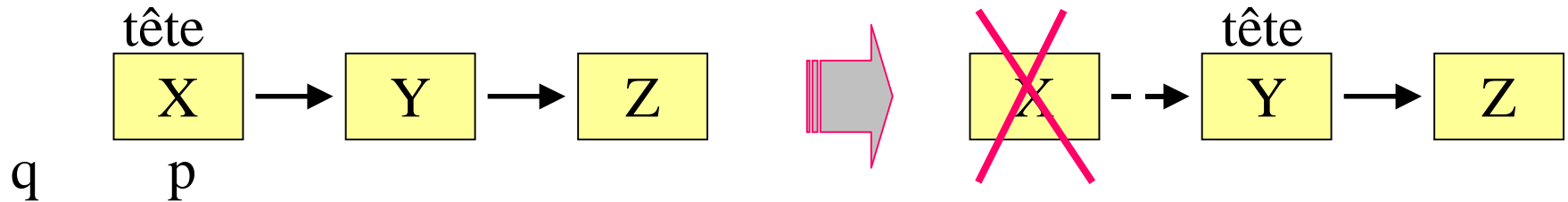
# Listes chaînées

- Insertion dans une liste triée (2)

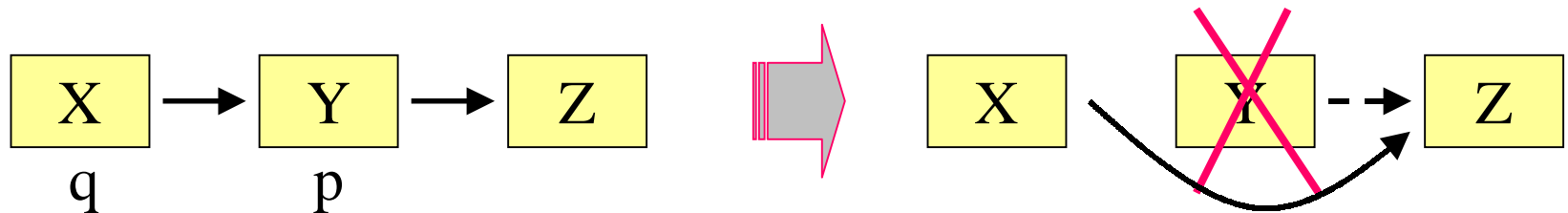
```
void InsertionTrieV2(liste &tete, TypeOfInfo x) {  
    liste q = NULL; liste p = tete;  
    while (p != NULL && p->info <= x) {  
        q = p;  
        p = p -> next;  
    }  
    if (q == NULL) { // insertion en début de liste  
        tete = new elem;  
        q = tete;  
    }  
    else {  
        q -> next = new elem;  
        q = q->next;  
    }  
    q -> info = x;  
    q -> next = p;  
}
```

## Listes chaînées

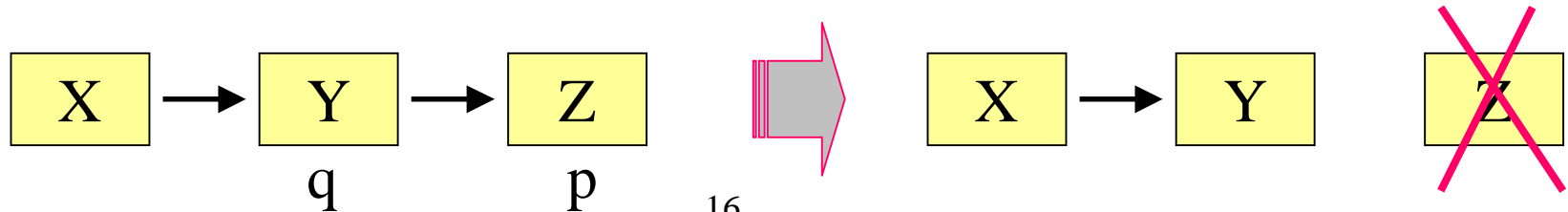
- Suppression d'un élément de la liste
  - L'élément à supprimer est en tête de liste (X)



- L'élément à supprimer est en milieu de liste (Y)



- L'élément à supprimer est en fin de liste (Z)





## Listes chaînées

- Suppression d'un élément de la liste

```
void Suppression(liste & tete, TypeOfInfo x) {  
    liste q = NULL;  
    liste p = tete;  
    while (p != NULL && p->info != x) {  
        q = p;  
        p = p -> next;  
    }  
    if (p != NULL) {  
        if (q == NULL)  
            tete = p->next;  
        else  
            q->next = p->next;  
        delete p;  
    }  
}
```

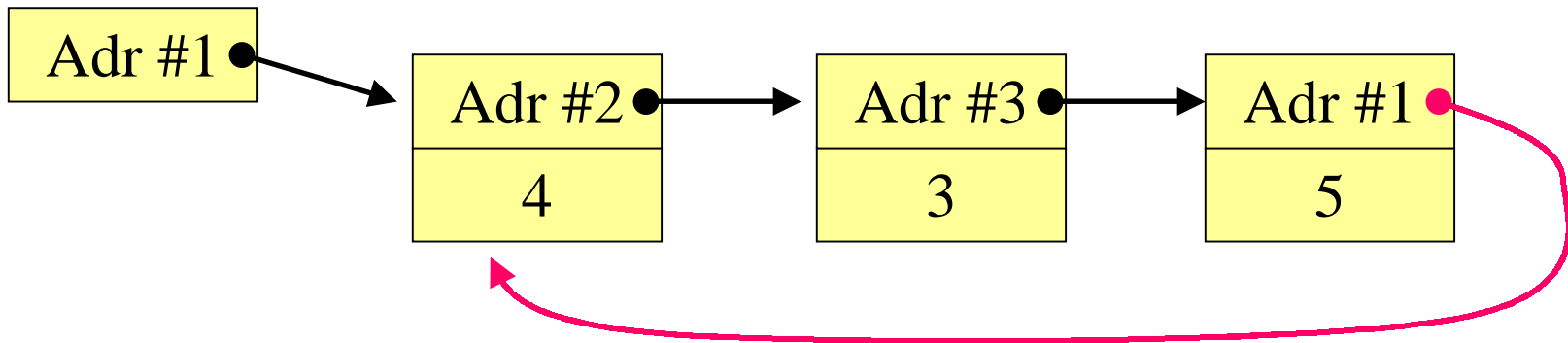
## Listes chaînées

- Suppression d'une liste

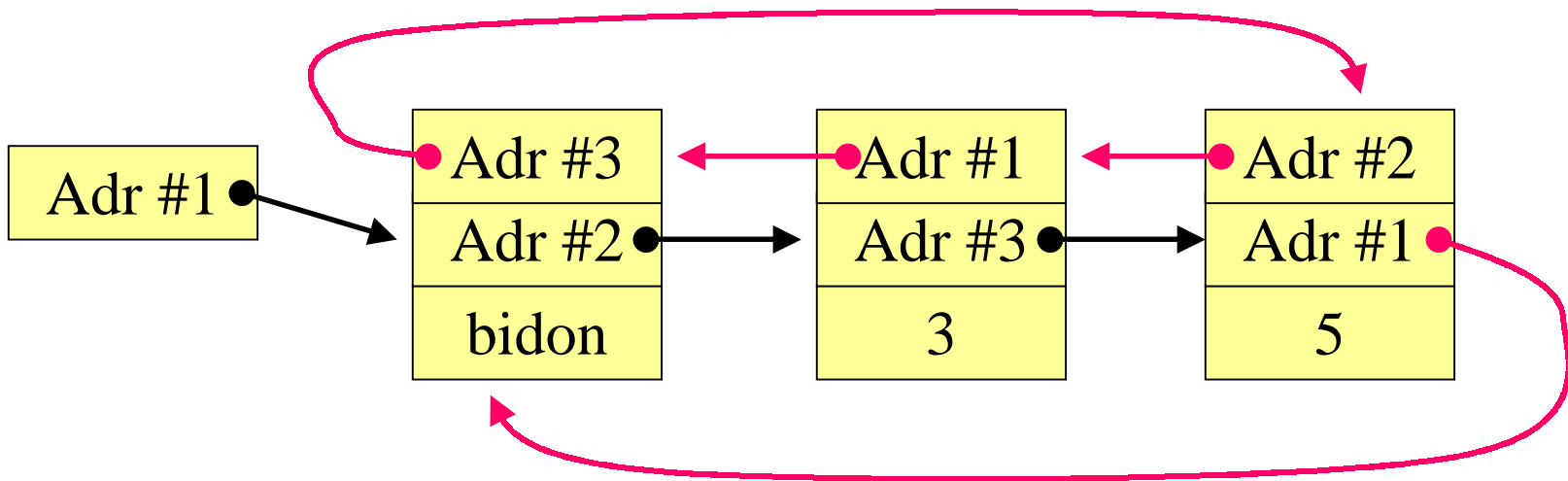
```
void SuppressionListe(liste & tete) {  
    while (tete != NULL) {  
        liste p = tete;  
        tete = tete -> next ;  
        delete p;  
    }  
}
```

# Listes (doublement) chaînées circulaires

## Liste chaînée circulaire

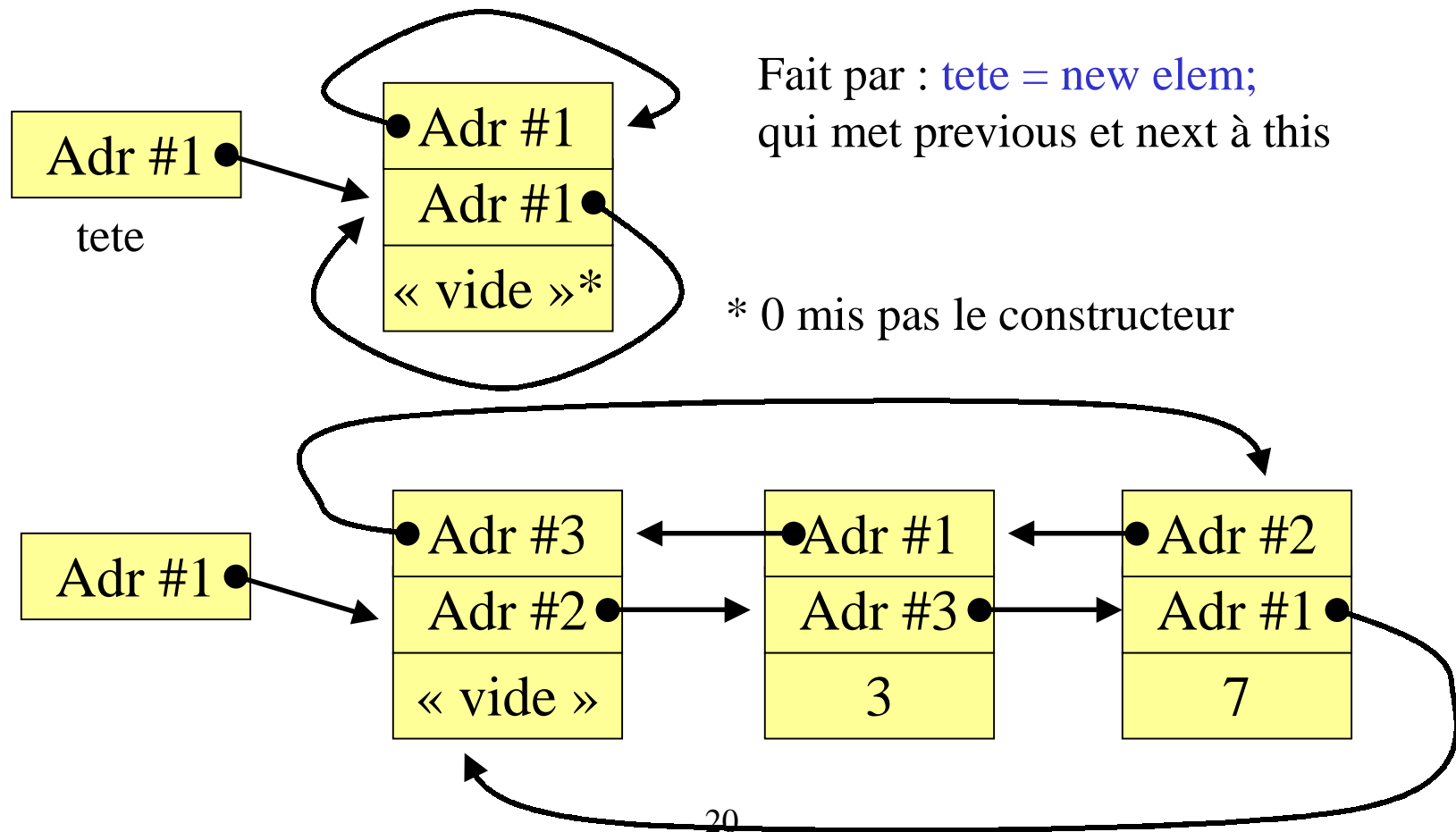


## Liste doublement chaînée circulaire



# Listes doublement chaînées circulaires


- Pour se simplifier la vie, la liste « vide » en fait constituée d'un élément spécial (élément « bidon ») ne contenant pas d'information: on appelle parfois cet élément « prêtête »



# Listes doublement chaînées circulaires

- Déclaration de la classe elem (doublement liée)

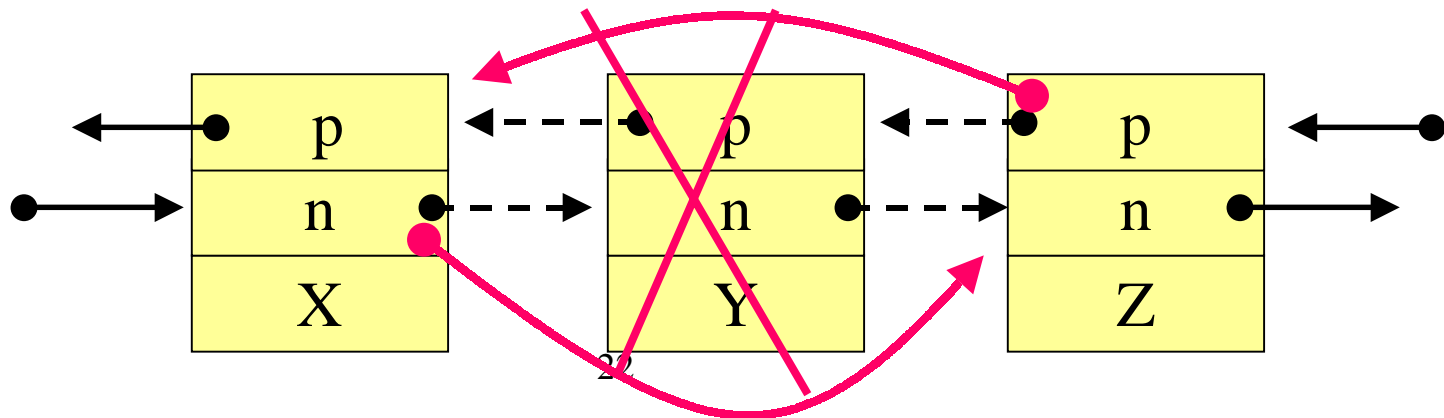
```
class elem;  
typedef elem* liste;  
class elem {  
public:  
    TypeOfInfo    info;  
    liste    previous; // ajout du pointeur vers le précédent  
    liste    next;  
    elem(): info(0),previous(this), next(this){}  
    elem(TypeOfInfo i, liste p, liste n):  
        info(i),previous(p), next(n){}  
};  
...  
liste tete = new elem;
```



# Listes doublement chaînées circulaires

- Supprimer un élément de la liste

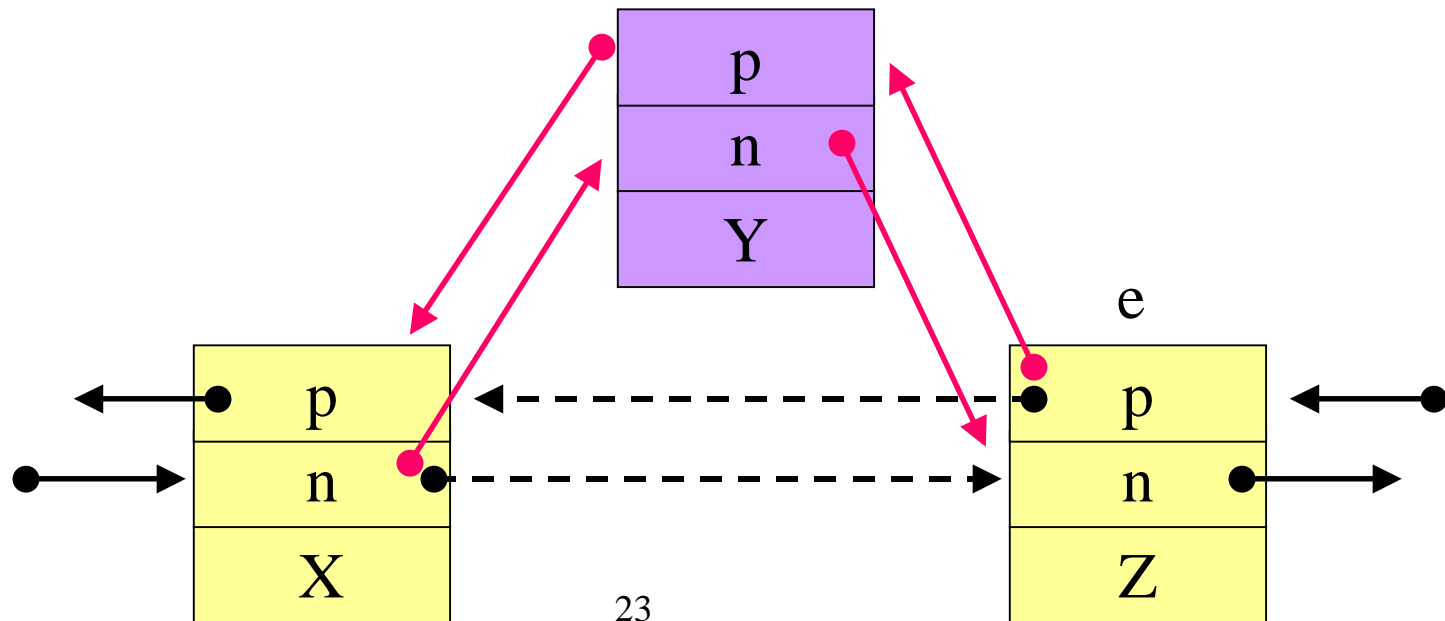
```
void Suppression(liste tete, TypeOfInfo x) {  
    liste p = tete->next;  
    tete->info=x;  
    while (p->info != x)  
        p = p->next;  
    if (p != tete) {  
        // on a donc trouvé x (pas celui dans la « pretete »)  
        p->previous->next = p->next;  
        p->next->previous = p->previous;  
        delete p;  
    }  
}
```



## Listes doublement chaînées circulaires

- Insérer un élément devant un autre élément de la liste

```
void InsAvant(liste e, TypeOfInfo x) {  
    liste r = new elem(x, e->previous, e);  
    e->previous->next = r;  
    e->previous = r;  
}
```



## Listes doublement chaînées circulaires

- Insertion en tête de liste (mais après la prétête)

```
void InsertionEnTete(liste tete, TypeOfInfo x) {  
    InsAvant(tete->next, x);  
}
```

- Insertion après un élément

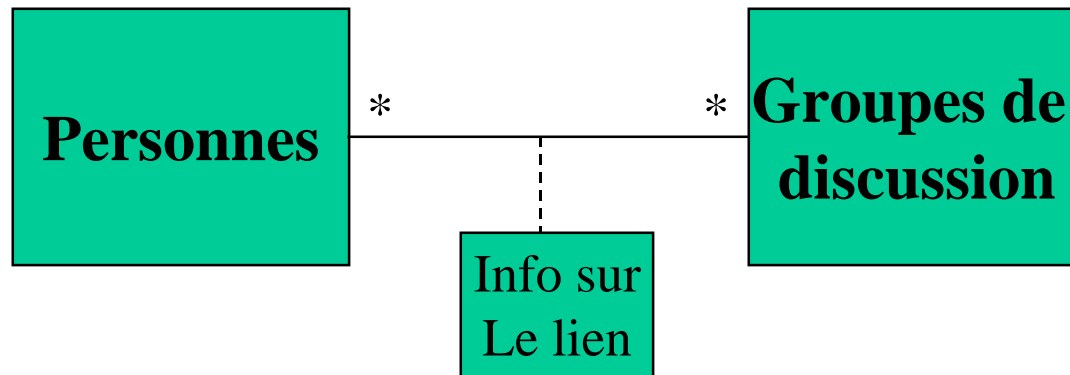
```
void InsApres(liste e, TypeOfInfo x) {  
    InsAvant(e->next, x);  
}
```

- Insertion en fin de liste (donc avant prétête)

```
void InsFin(liste tete, TypeOfInfo x) {  
    InsAvant(tete, x);  
}
```



## Implémentation d'une relation m-n

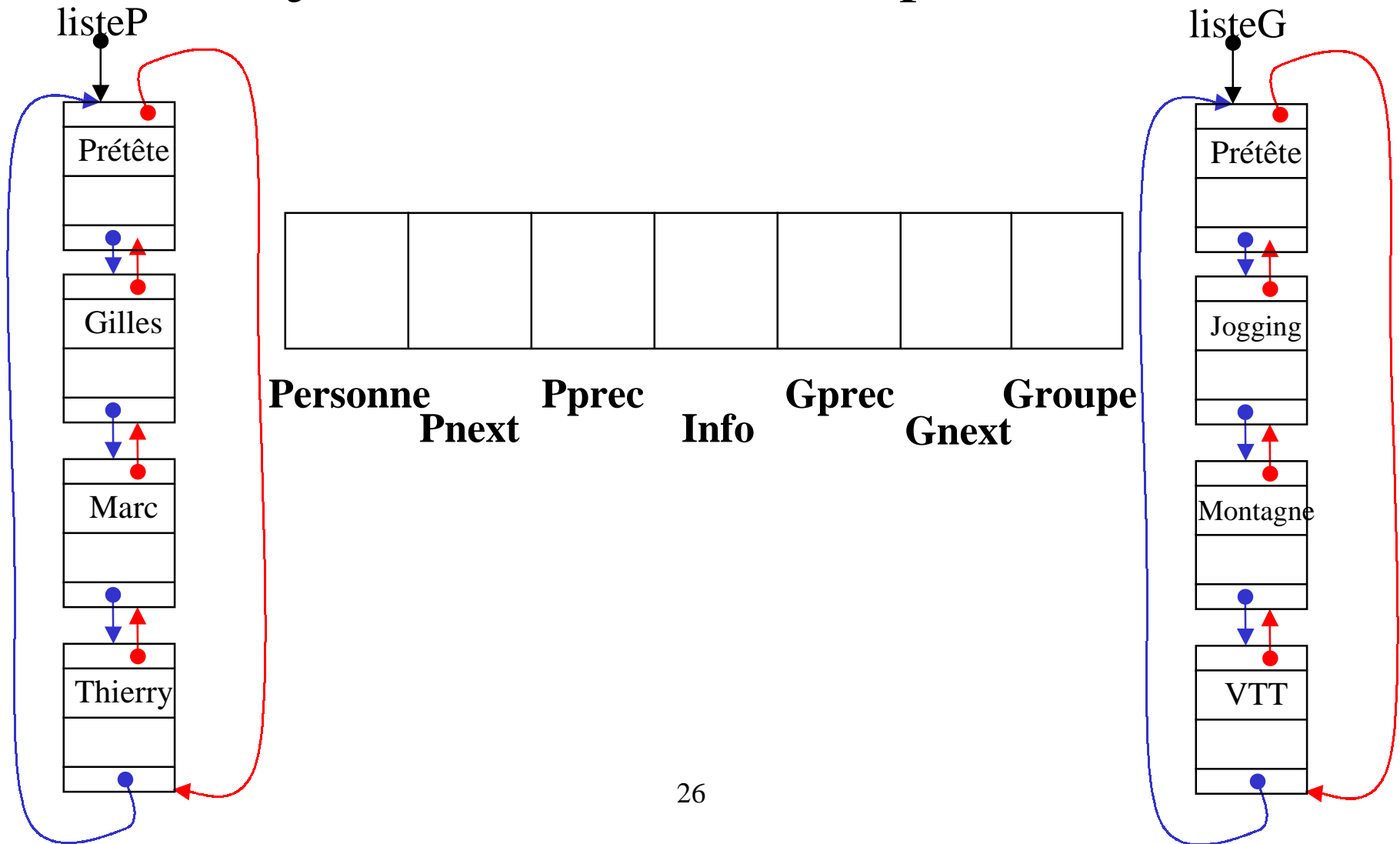


- Représentation graphique (UML) de fait que chaque personne peut être dans plusieurs groupes de discussion et que chaque groupe de discussion « a » plusieurs personnes.

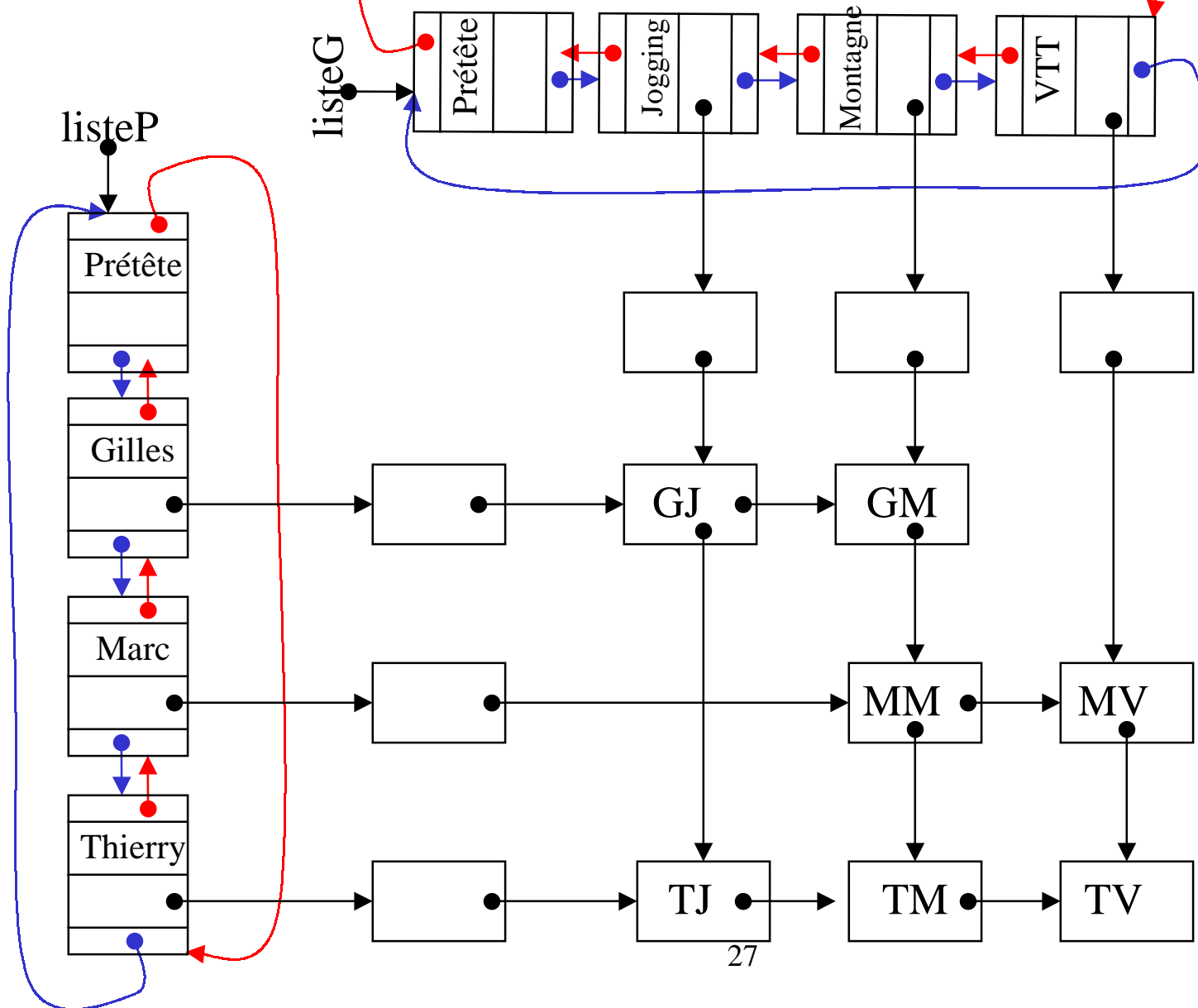
**⇒ Chaque personne a une liste de groupes de discussion**  
**⇒ Chaque groupe de discussion a une liste de personnes**  
**+ on ne veut pas dupliquer les info sur les personnes et groupes**  
**+ on veut pouvoir mettre des info concernant**  
**chaque lien « personne-groupe »**

# Implémentation d'une relation m-n (matrice creuse)

- => Objets « lien » à 7 « champs » de données

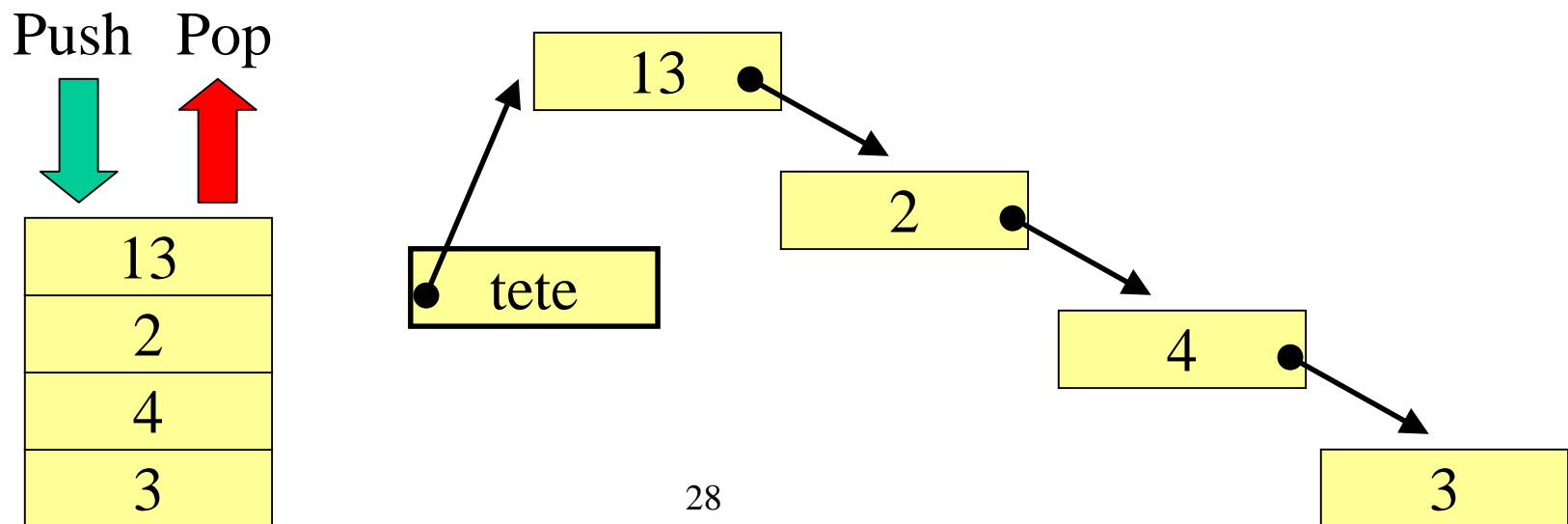


# Implémentation d'une relation m-n (matrice creuse)



# Piles

- Une pile est une structure de donnée dynamique utilisant le principe LIFO (*last in, first out*)
- 4 opérations sont permises
  - **IsEmpty**: test si la pile est vide
  - **Top** : lecture du sommet de pile (sans effet sur la pile)
  - **Pop** : suppression du sommet de pile
  - **Push** : ajout d'un élément sur la pile
- Une pile est équivalente à une simple liste chaînée



## Piles

- La classe pile est définie sur base d 'éléments, comme la liste

```
class Element;  
typedef Element* Stack;  
class Element {  
public:  
    Stack    next;  
    int      data;  
    Element(): next(NULL),data(0){}  
    Element(Stack n, int d) : next(n), data(d) {}  
};
```

# Piles

- Ajouter un élément au sommet de pile

```
void Push(stack &S, int d) {  
    S = new Element (S, d);  
}
```

- Tester si la pile est vide

```
bool IsEmpty(S) {  
    return S == NULL;  
}
```

- Supprimer le sommet de pile

```
void Pop(stack &S) {  
    if (! IsEmpty(S)) {  
        stack T = S;  
        S = S->next;  
        delete T;  
    } // si la pile est vide, il ne se passe rien  
}
```

## Piles

- Lire le sommet de pile (si la pile n'est pas vide)

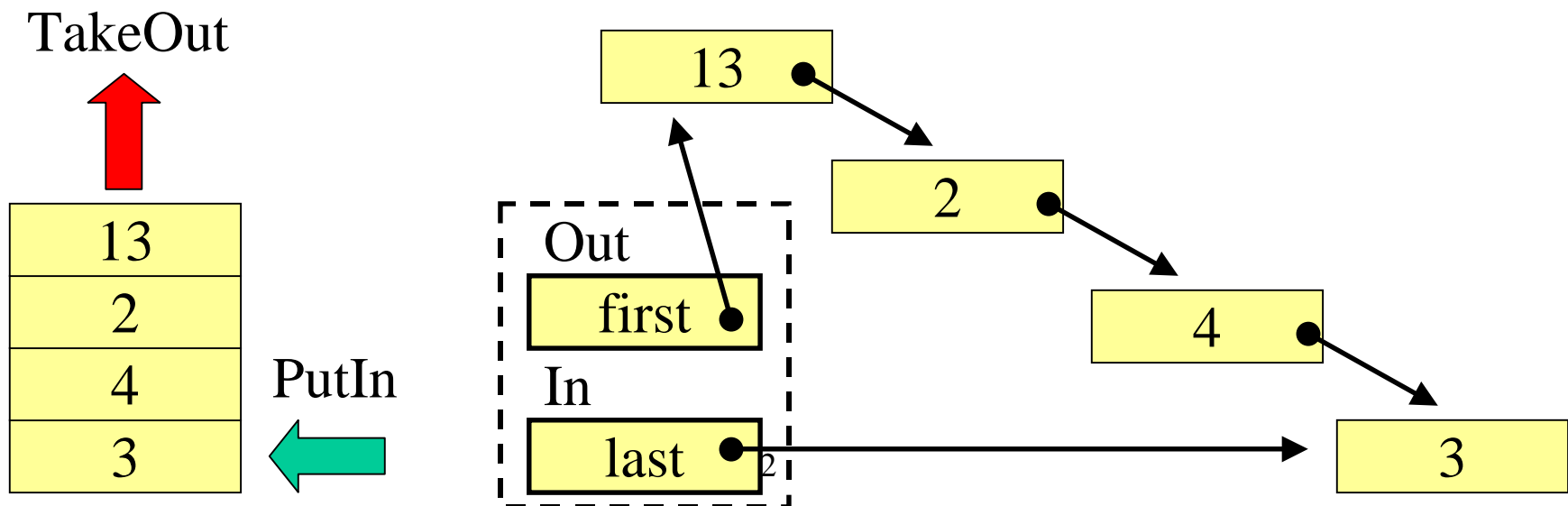
```
int Top(stack S) {  
    return S->data;  
}
```

- Détruire la pile

```
void DeleteStack(stack &S) {  
    while (! IsEmpty(S)) {  
        stack T = S;  
        S = S->next;  
        delete T;  
    }  
}
```

# Files

- Une file est une structure de donnée dynamique utilisant le principe FIFO (*first in, first out*)
- 4 opérations sont permises
  - IsEmpty: test si la file est vide
  - First : lecture du premier de la file (sans effet sur la file)
  - TakeOut : sortie d'un élément de la file
  - PutIn : entrée d'un élément dans la file
- Une file est équivalente à une simple liste chaînée (+ dernier)





## Files

- La classe file est définie sur base d'éléments, comme la liste

```
class Element;  
typedef Element* ptr;  
class Element {  
    ptr      next;  
    int      data;  
    Element():next(NULL),data(0){}  
    Element(ptr n, int d) : next(n), data(d) {}  
};  
  
class Queue {  
public:  
    ptr      first, last;  
  
};
```

# Files

- Entrer un élément dans la file

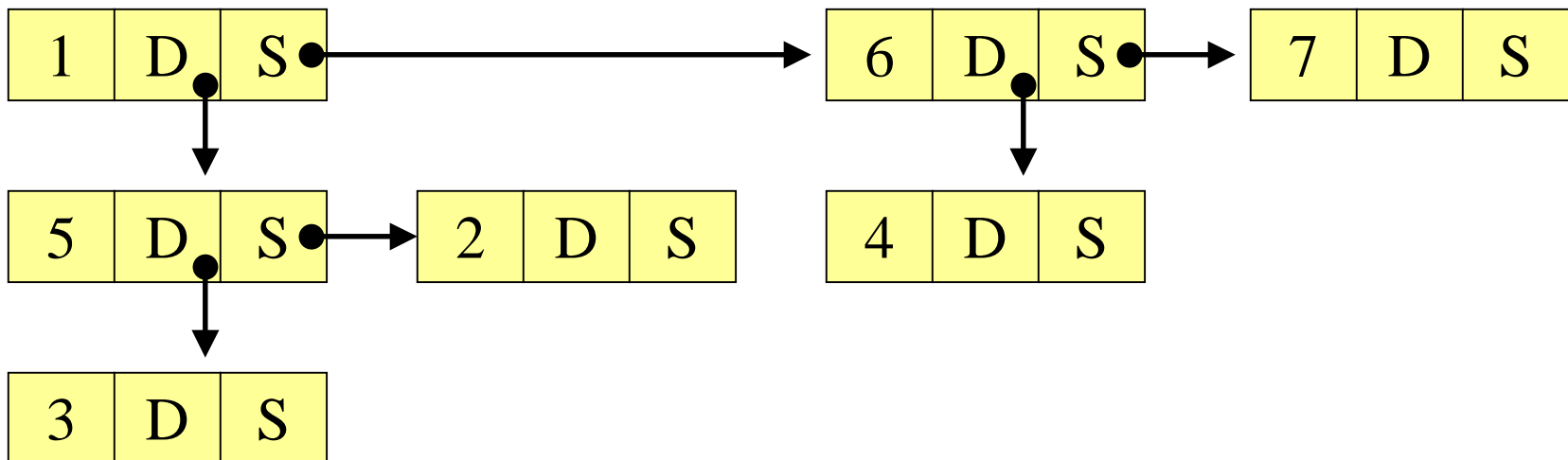
```
void PutIn(Queue &Q, int d) {  
    ptr temp = new Element (NULL, d);  
    if (IsEmpty(Q))  
        Q.first = temp;  
    else  
        Q.last->next = temp;  
    Q.last = temp;  
}
```

- Sortir un élément de la file

```
void TakeOut(Queue &Q) {  
    if (! IsEmpty(Q)) {  
        ptr p = Q.first;  
        Q.first = Q.first->next;  
        if (IsEmpty(Q))  
            last = NULL;  
        delete p;  
    } // si la pile est vide, il ne se passe rien  
}
```

# Listes généralisées

- Une liste généralisée est construite à partir d'éléments possédant chacun deux pointeurs vers d'autres éléments : D (down) pointe vers une liste située à un niveau inférieur, S (same) pointe vers une liste située au même niveau
- Exemple :



- Problème : parcourir la liste généralisée niveau par niveau  
Dans l'exemple : 1 6 7 / 5 2 4 / 3

## Listes généralisées - Les classes

- La classe ListeGen est aussi définie sur base d'éléments

```
class ElGen
typedef ElGen* ListeGen;
class ElGen {
public:
    ListeGen D, S;
    int data;
    ElGen():D(NULL),S(NULL),data(0){}
    ElGen(ListeGen d, ListeGen s, int x) : D(d), S(s), data(x){}
};
```

# Listes généralisées - Parcours par niveau

- ```
void ParcourslisteGen(ListeGen L) {  
    if (L != NULL) {  
        Queue Q;  
        PutIn(Q,L);  
        while (! IsEmpty(Q)) {  
            ListeGen p = First(Q);  
            TakeOut(Q);  
            while (p != NULL) {  
                cout << p->data << endl;  
                if (p->D != NULL) PutIn(Q,p->D);  
                p = p->S;  
            }  
        }  
    }  
}
```

On utilise une file dont les éléments contiennent des pointeurs vers des éléments de la liste généralisée (ElGen)

Exercice : modifier le programme pour que les données s'affichent à raison d'une ligne par niveau

## Application - Notation polonaise inversée

- La notation algébrique directe (NAD) classique peut être traduite en une autre notation appelée polonaise inversée (NPI, Jan Lukasiewicz, 1878-1956), et vice-versa
- Exemple :  $A + (B * C) + (D / E) \equiv A B C * + D E / +$
- Lorsqu'on a sous la main une expression en NPI, il est extrêmement simple de l'évaluer : il suffit, en parcourant l'expression de gauche à droite,
  - d'empiler les opérandes,
  - de les dépiler lorsqu'on rencontre un opérateur et de les remplacer alors par le résultat de l'opération
- Exemple : Push(S,A); Push(S,B); Push(S,C); Multiply(S); Add(S); Push(S,D); Push(S,E); Divide(S); Add(S); et le résultat se trouve au sommet de la pile => cout << Top(S);

# Application - Traduction NAD vers NPI

- On ne traite que les opérateurs binaires  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $^$  et les  $()$
- Les priorités sont ordonnées comme suit :  $() < +, - < *, / < ^$
- En parcourant l'expression NAD de gauche à droite :
  - Opérande : directement transmise en sortie
  - Opérateur : tant que le sommet de la pile a une priorité supérieure ou égale à celle de l'opérateur d'entrée, on le sort de la pile ; ensuite, on empile l'opérateur rencontré à l'entrée
  - Parenthèse ouvrante : empilée
  - Parenthèse fermante : tant que le sommet est différent d'une parenthèse ouvrante, on le sort de la pile ; la parenthèse ouvrante en question est dépilée mais non transmise en sortie
  - Fin de l'expression d'entrée : tout ce qui reste sur la pile est transmis en sortie