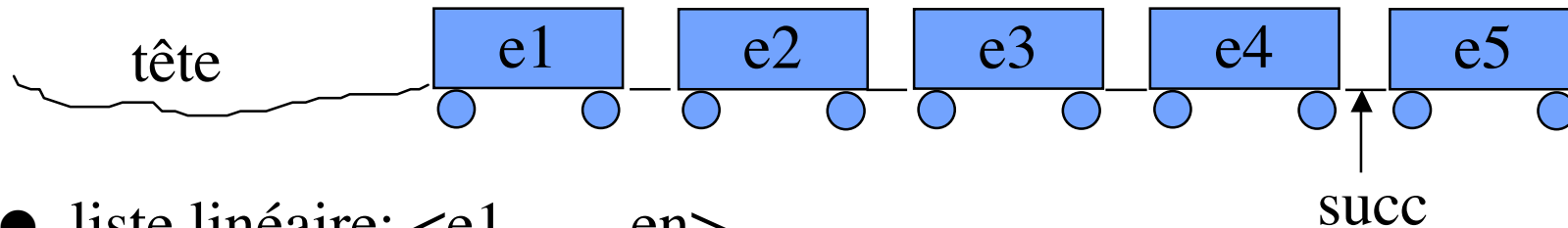


# Structures séquentielles

# liste linéaire (1)



- liste linéaire:  $\langle e1, \dots, en \rangle$

ordre sur les places et non sur les éléments

pour tout  $p$ , il existe  $k \geq 0$  tel que  $p = \text{succ}^k(\text{tete}(L))$

longueur de la liste:  $n$

- traitement séquentiel

$x := \text{tête} ( L );$

traiter (x);

**pour**  $i := 1$  **jusqu'à** longueur (L) - 1 **faire**

$x := \text{succ} (x);$

    traiter (x);

**fait**

# liste linéaire (2)

**type** liste

**paramètre** élément

**utilise** entier, booléen

**opérations**

listevide :  $\square$  liste

insérer : liste  $\square$  entier  $\square$  élément /  $\square$  liste

supprimer : liste  $\square$  entier /  $\square$  liste

longueur : liste  $\square$  entier

ième : liste  $\square$  entier /  $\square$  élément

**préconditions**

insérer(l, k, e):  $1 \leq k \leq \text{longueur}(l) + 1$

supprimer(l, k):  $1 \leq k \leq \text{longueur}(l)$

ième(l, k):  $1 \leq k \leq \text{longueur}(l)$

# liste linéaire (3)

## sémantique

longueur(l) = **si** l = listevide **alors** 0

**sin**si l = insérer(l', k, e) **alors** longueur(l') + 1

**sinon** soit l = supprimer(l', k); longueur(l') - 1 **fsi**

ième(insérer(l, k, e), i) = **si**  $1 \leq i < k$  **alors** ième(l, i)

**sin**si i = k **alors** e

**sinon** ième(l, i - 1) **fsi**

ième(supprimer(l, k), i) = **si**  $1 \leq i < k$  **alors** ième(l, i)

**sinon** ième(l, i + 1) **fsi**

*dire ce que sont les observateurs sur le résultat des opérations*

# listes récursives (1)

**type** liste

**paramètre** élément

**utilise** booléen

**opérations**

listevide	: $\square$ liste
cons	: élément $\square$ liste $\square$ liste
fin	: liste $\rightarrow$ $\square$ liste
estvide	: liste $\square$ booléen
premier	: liste $\rightarrow$ élément

**préconditions**

fin(l):  $\neg$  estvide (l)  
premier(l):  $\neg$  estvide(l)

**sémantique**

$\square$  e: élément, l: liste  
estvide(listevide)  
 $\neg$  estvide(cons(e, l))  
fin(cons(e, l)) = 1  
premier(cons(e, l)) = e

# listes récursives (2)

$\text{cons}(e_1, \text{fin}(\text{fin}(\text{cons}(e_2, \text{fin}(\text{cons}(e_3, \text{cons}(e_4, \text{listevide})))))))$

$= \text{cons}(e_1, \text{fin}(\text{fin}(\text{cons}(e_2, \text{cons}(e_4, \text{listevide})))))$

$= \text{cons}(e_1, \text{fin}(\text{cons}(e_4, \text{listevide})))$

$= \text{cons}(e_1, \text{listevide})$

$\text{fin}(\text{cons}(e, l)) = l$

# relation entre les modèles

- liste itérative vers liste récursive

$\text{fin}(l) = \text{supprimer}(l, 1)$

$\text{cons}(e, l) = \text{insérer}(l, 1, e)$

$\text{premier}(l) = \text{ième}(l, 1)$

$\text{estvide}(l) \equiv \text{longueur}(l) = 0$

- liste récursive vers liste itérative

$\text{longueur}(l) = \text{si } \text{estvide}(l) \text{ alors } 0 \text{ sinon } 1 + \text{longueur}(\text{fin}(l)) \text{ fsi}$

$\text{supprimer}(l, k) = \text{si } k = 1 \text{ alors } \text{fin}(l)$

$\text{sinon } \text{cons}(\text{premier}(l), \text{supprimer}(\text{fin}(l), k-1)) \text{ fsi}$

$\text{insérer}(l, k, e) = \text{si } k = 1 \text{ alors } \text{cons}(e, l)$

$\text{sinon } \text{cons}(\text{premier}(l), \text{insérer}(\text{fin}(l), k-1, e)) \text{ fsi}$

# introduction des places

parcours des listes récursives est "destructeur":

traitement (l) = traiter(premier(l))  $\square$  traitement (fin(l))

**type** place

**paramètre** élément

**utilise** booléen, liste[élément]

**opérations**

tête	: liste $\rightarrow$ place
succ	: place $\rightarrow$ place
dernier	: place $\rightarrow$ booléen
contenu	: place $\rightarrow$ élément

**préconditions**

tête(l):  $\neg$  estvide(l)  
succ(p):  $\neg$  dernier(p)

**sémantique**

succ(tête(l)) = tête(fin(l))  
dernier(tête(l))  $\equiv$  estvide(fin(l))  
contenu(tête(l)) = premier(l)



# simplification d'écriture

**extension** type liste

**opérations**

$[_]$  : élément  $\rightarrow$  liste

$[_; _]$  : élément  $\rightarrow$  élément  $\rightarrow$  liste

$_:: _$  : élément  $\rightarrow$  liste  $\rightarrow$  liste

**sémantique**

$[e] = \text{cons}(e, \text{listevide})$

$[e_1; e_2] = \text{cons}(e_1, \text{cons}(e_2, \text{listevide}))$

$e :: l = \text{cons}(e, l)$

# opérations complémentaires

**extension type** liste

**opérations**

$\_ \& \_$  : liste  $\rightarrow$  liste  $\rightarrow$  liste  
rechercher : liste  $\rightarrow$  élément /  $\rightarrow$  place

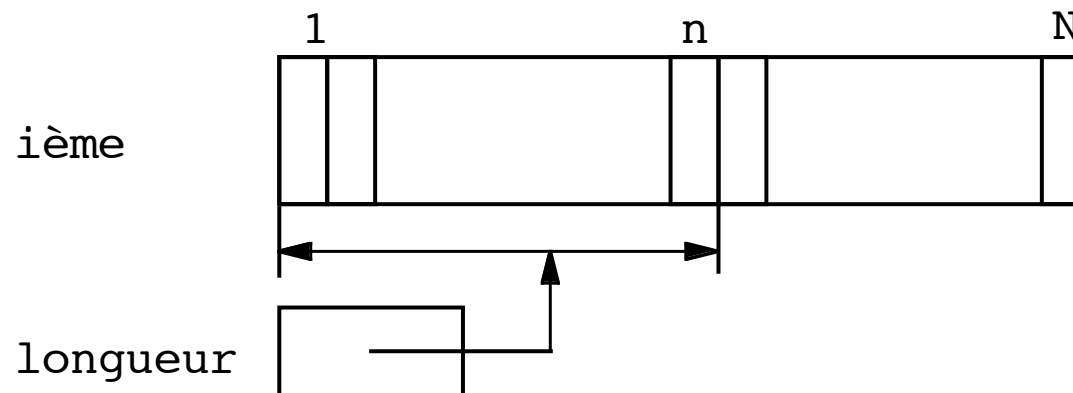
**préconditions**

rechercher (l, e):  $\neg$  estvide(l)

**axiomes**

$l \& l' =$  **si** estvide(l) **alors** l' **sinon** premier(l) :: (fin(l) & l') **fsi**  
rechercher(l, e) = **si** e = premier(l)  $\wedge$  estvide(fin(l)) **alors** tête(l)  
**sinon** rechercher(fin(l), e) **fsi**

# implantation contiguë



```
type Table_Element is array (Positive range<>) of Element
```

```
type Liste (Taille_Max : Positive) is
```

```
  record -- représentation effective du type liste fourni
```

```
    Longueur : Natural := 0; --initialisation à la création
```

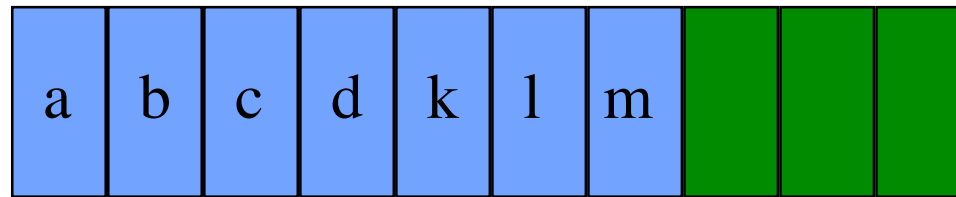
```
    Ieme : Table_Element(1..Taille_Max);
```

```
  end record;
```

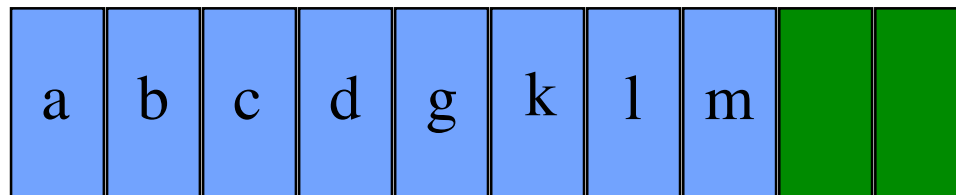
- visibilité: partielle=> garantir  $0 \leq \text{Longueur} \leq \text{Taille\_Max}$

# Exemple d'insertion

- Soit à insérer "g" en 5 dans la liste [a; b ; c; d; k; l; m]
- partant de la liste ci-dessous,

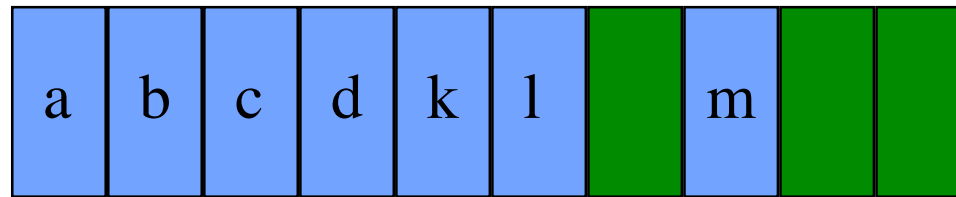


- il faut aboutir à celle-ci (selon les spécifications):



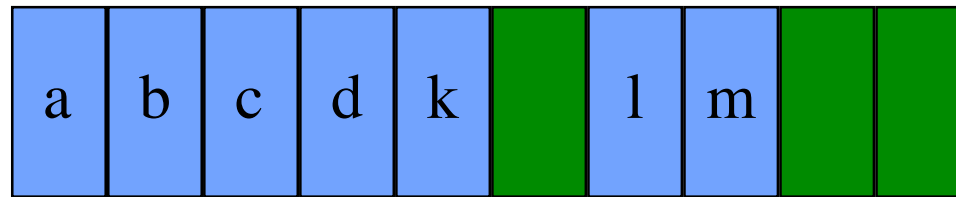
# Exemple d'insertion

- Déplacement du 7ème



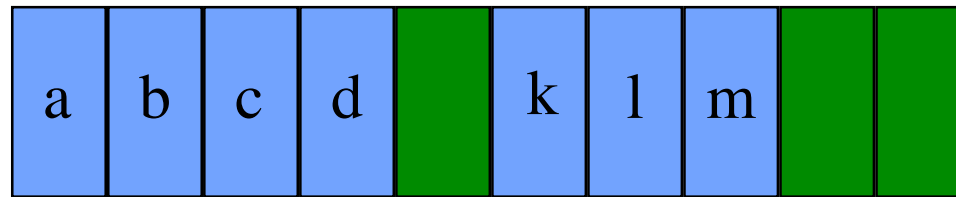
# Exemple d'insertion

- Déplacement du 6ème



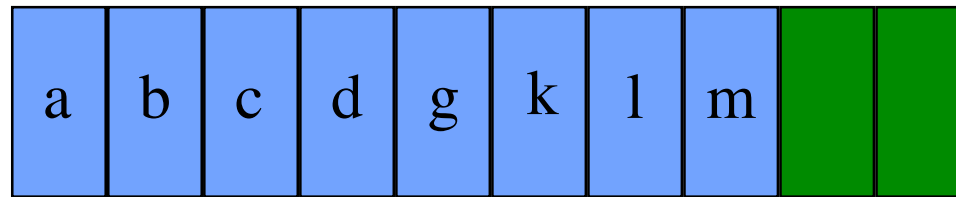
# Exemple d'insertion

- Déplacement du 5ème



# Exemple d'insertion

- Placement de g





# opérations sur listes contiguës (1)

- Implantation simple de Longueur, Ieme
- Implantation en 1,  $n/2$ ,  $n$  pour Insérer:

```
procedure Insérer (Dans : in out Liste; En : in Positive; Val : in Element) is  
begin  
    if En > Dans.Longueur + 1 then raise Erreur_Specification; end if;  
    if Dans.Longueur = Dans.Taille_Max then raise Limite_Implantation; end if;  
    Dans.Longueur := Dans.Longueur + 1;  
    Dans.Ieme (En + 1..Dans.Longueur) := Dans.Ieme (En..Dans.Longueur - 1);  
    Dans.Ieme (En) := Val;  
end Insérer;
```

# opérations sur listes contiguës (2)

- Implantation en 1,  $n/2$ ,  $n$  pour Supprimer:

**procedure** Supprimer (Dans : **in out** Liste; En : **in** Positive) **is**

**begin**

**if** En > Dans.Longueur **then raise** Erreur\_Specification; **end if**;

    Dans.Longueur := Dans.Longueur - 1;

    Dans.Ieme (En..Dans.Longueur) := Dans.Ieme (En + 1..Dans.Longueur + 1);

**end** Supprimer;

- Ajouter des opérations permettant de gagner en efficacité:

Changer\_Ieme => enchaînement de suppression/insertion au même endroit

Tronquer => suppression des derniers

# Implantation contiguë en Java (1)

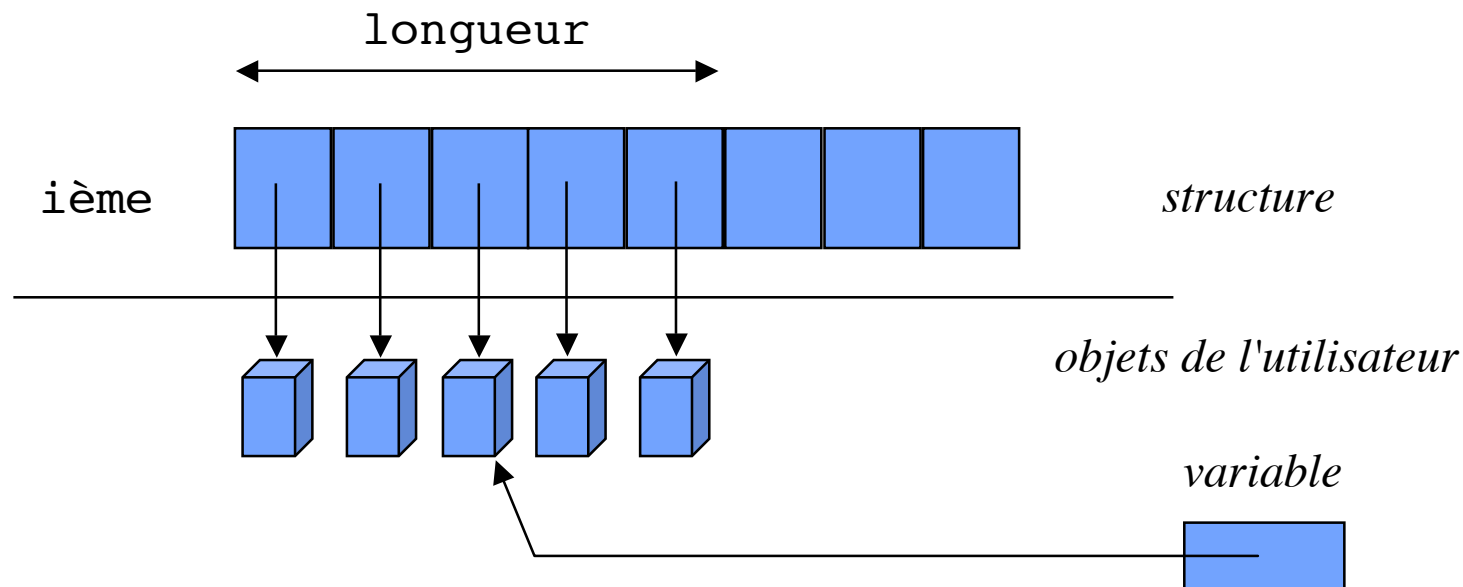
- Liste d'objets

```
package bibSDjava.Les_Listes_Contigues;
import bibSDjava.Erreurs_Types_Abstraits.*;
public class Liste_Contigue_D_Objets {
    protected int longueur = 0;
    protected Object [] ieme;
    public Liste_Contigue_D_Objets (int Taille)
        throws Erreur_Specification {
        if (Taille <= 0) throw new Erreur_Specification(this);
        ieme = new Object [Taille];
    }
    public int Taille_Max () { return ieme.length; }
    public int Longueur () { return longueur; }
}
```

# Implantation contiguë en Java (2)

- En fait les objets dans la liste sont les objets de l'utilisateur, et non des copies

modifier un objet après l'avoir mis dans la liste, modifie l'objet de la liste



# Implantation contiguë en Java (3)

- Liste contiguë

```
package bibSDjava.Les_Listes_Contigues;  
import bibSDjava.Erreurs_Types_Abstraits.*;  
public class Liste_Contigue extends Liste_Contigue_D_Objets {  
    protected Class Type_Element;  
    public Liste_Contigue (int Taille, Cloneable1 Specimen)  
        throws Erreur_Specification, CloneNotSupportedException {  
        super (Taille);  
        Object Obj= Specimen.clone(); Type_Element = Specimen.getClass();  
    }  
    public Object Ieme (int En) throws Erreur_Specification {  
        try { return ((Cloneable1) Super.Ieme (En)).clone (); }  
        catch (CloneNotSupportedException e) {throw new InternalError(); }  
    }  
}
```

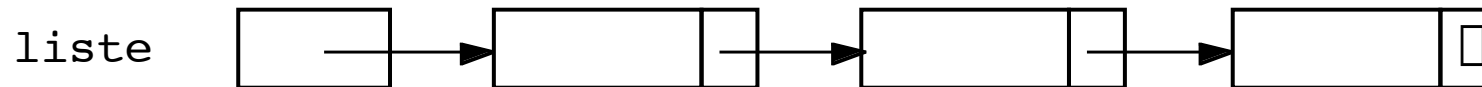
*redéfinition de l'opération*

*retourne un clone de l'objet*

# implantation contiguë en C++

```
template <class Element> class Liste_Contigue {  
    public:  
        Liste_Contigue (const Liste_Contigue<Element> &Copie);  
        ~Liste_Contigue () {delete [] ieme;}  
        const int Taille_Max;  
  
    protected:  
        Element& operator[ ] (const int En) {return ieme[En - 1];}  
        int longueur;  
  
    private:  
        Element *ieme;  
  
};
```

# implantation chaînée (1)



```
type Place; type Pt_Place is access Place;
```

```
type Place is
```

```
  record
```

```
    Contenu : Element;
```

```
    Suivant : Pt_Place := null;
```

```
  end record;
```

```
type Liste is
```

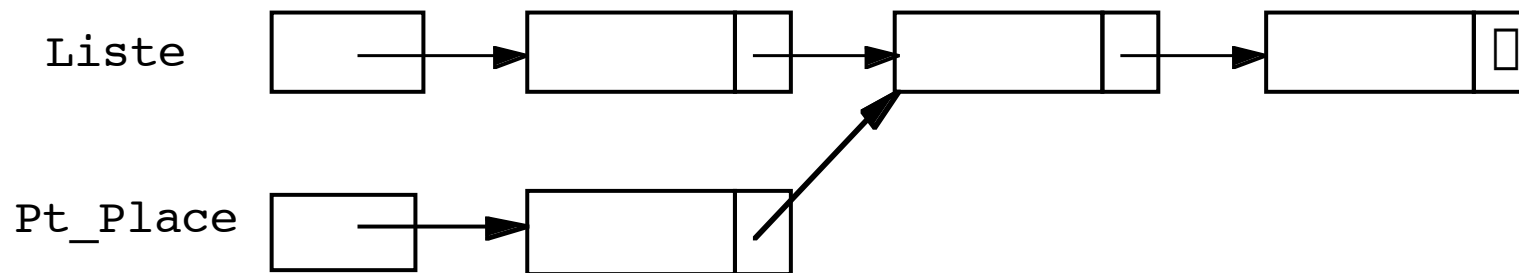
```
  record
```

```
    Tete : Pt_Place := null;
```

```
  end record;
```

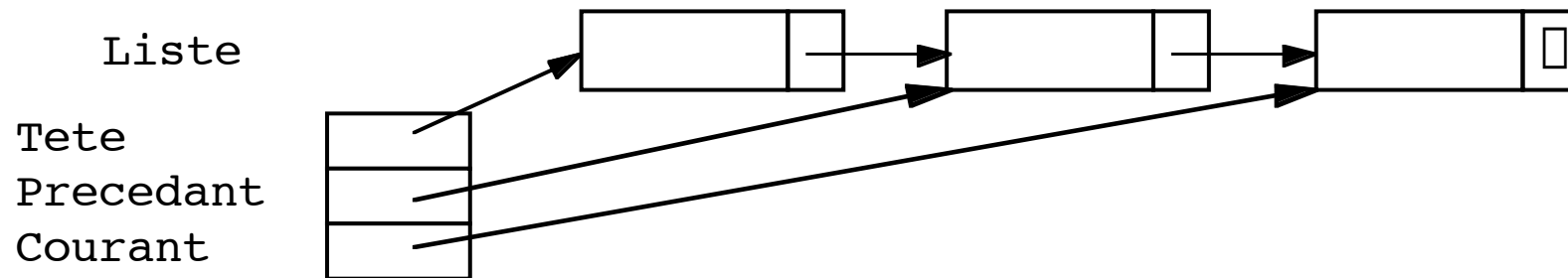
# implantation chaînée (2)

```
procedure Cons (La_Liste : in out Liste; Val : in Element) is  
begin  
  La_Liste.Tete := new Place'(Contenu => Val, Suivant => La_Liste.Tete);  
end Cons;  
  
function Succ (Courant : in Pt_Place) return Pt_Place is  
begin  
  if Courant = null then raise Place_Non_Definie; end if;  
  if Courant.Suivant = null then raise Erreur_Specification; end if;  
  return Courant.Suivant;  
end Succ;
```





# liste chaînée pointée (1)



Courant = **si** Precedant = **null** **alors** Tete **sinon** Precedant.Suivant **fsi**

**type** Liste **is new** Ada.Finalization.Limited\_Controlled **with**

**record**

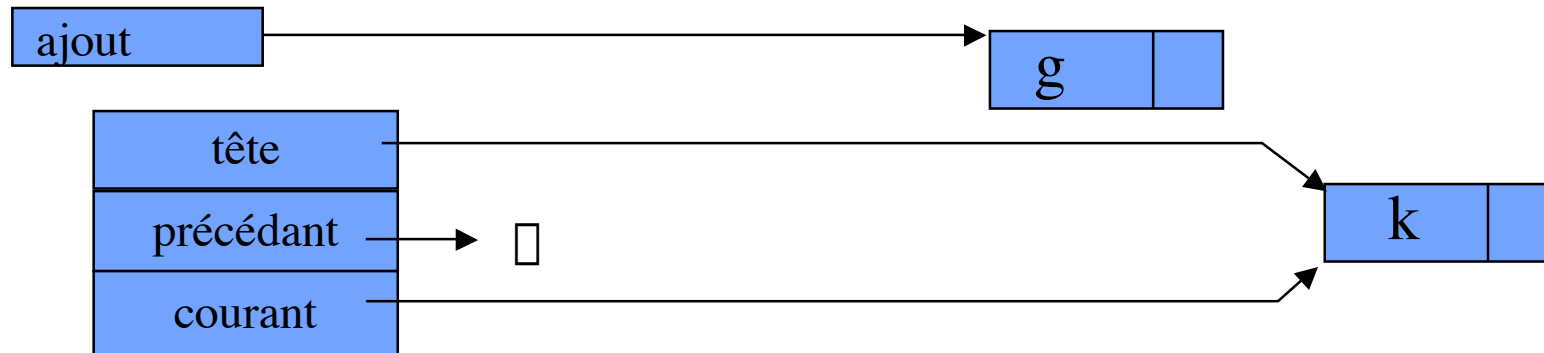
Longueur : Natural := 0;

Tete, Precedant, Courant : Pt\_Place := **null**;

**end record**;

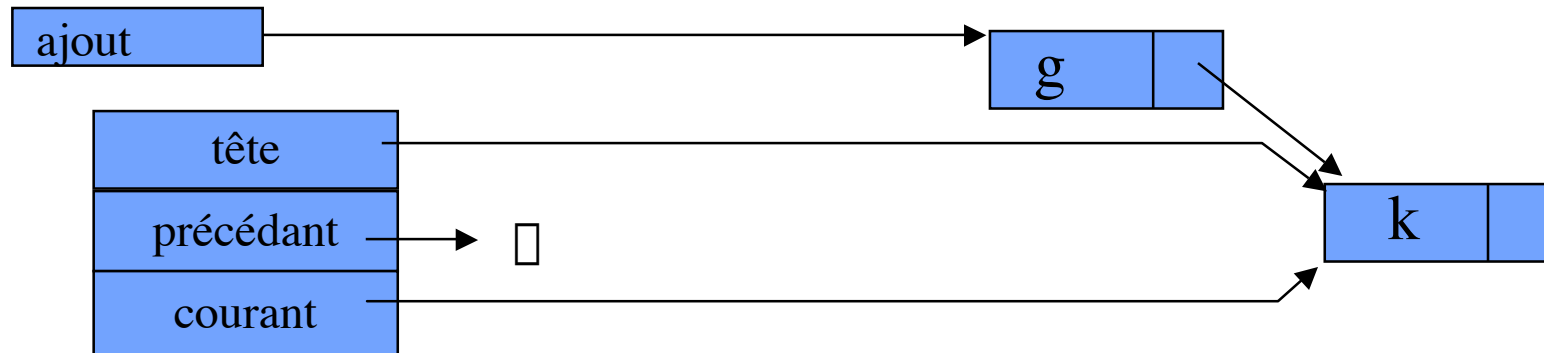
- visibilité: limitée

# liste chaînée pointée (2)



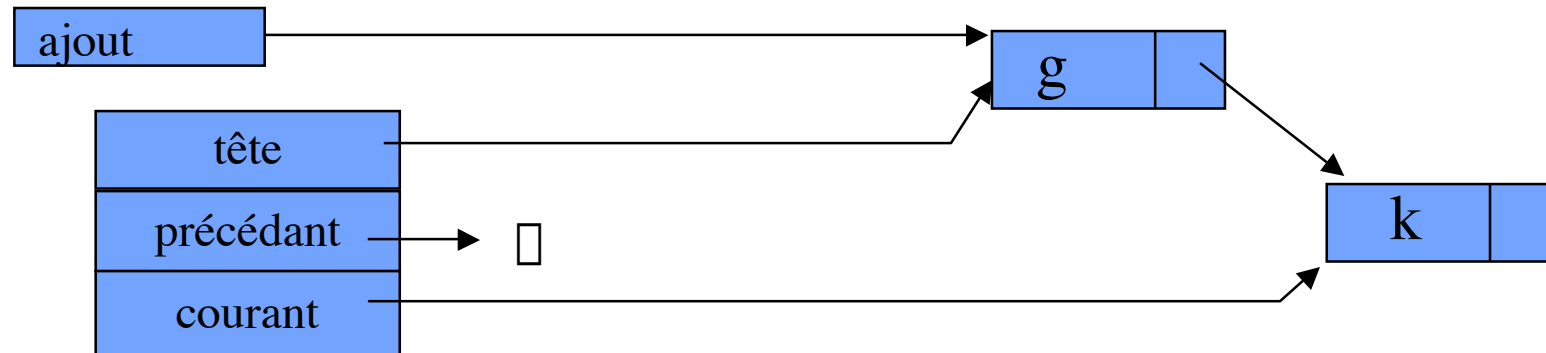
- Soit à ajouter "g" devant la position courante, qui est en tête
- Création de la place qui contiendra g

# liste chaînée pointée (2)



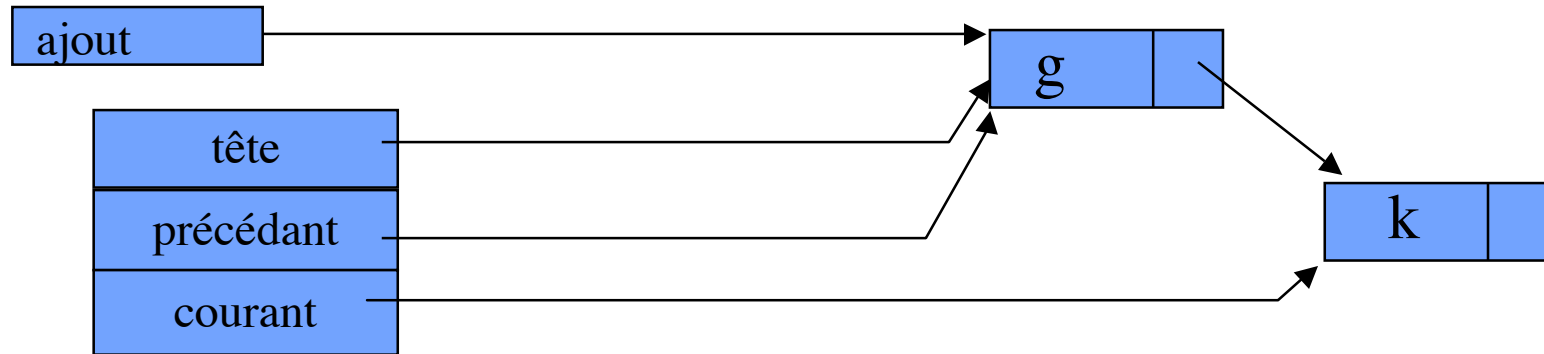
- Le suivant de cette place ajoutée est la place courante

# liste chaînée pointée (2)



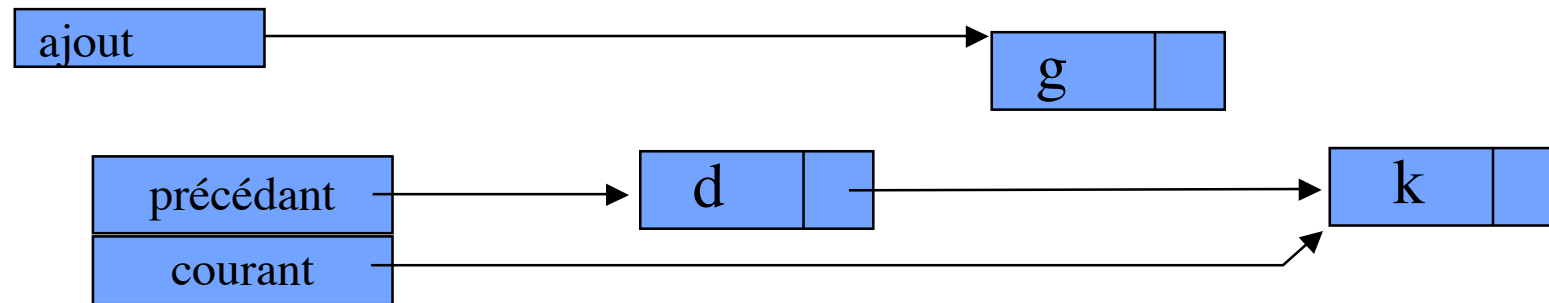
- La tête est la place ajoutée

# liste chaînée pointée (2)



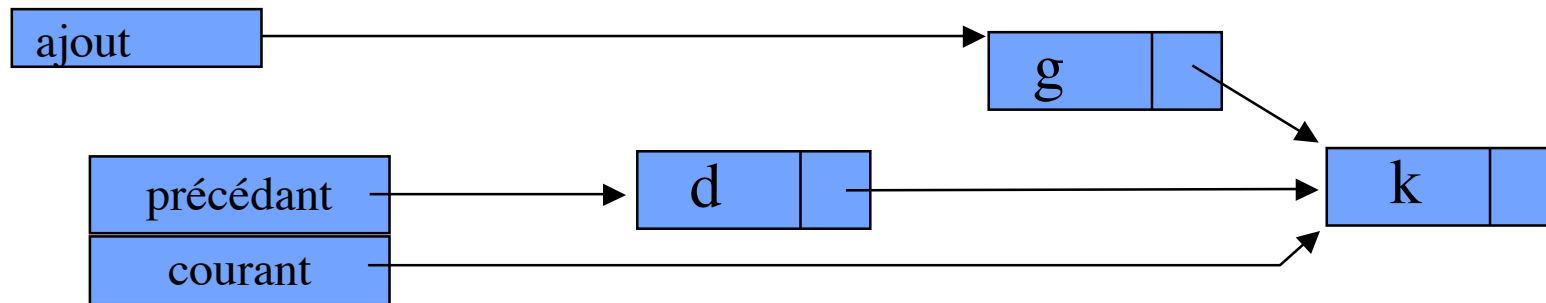
- Le précédant devient la place ajoutée

# liste chaînée pointée (2)



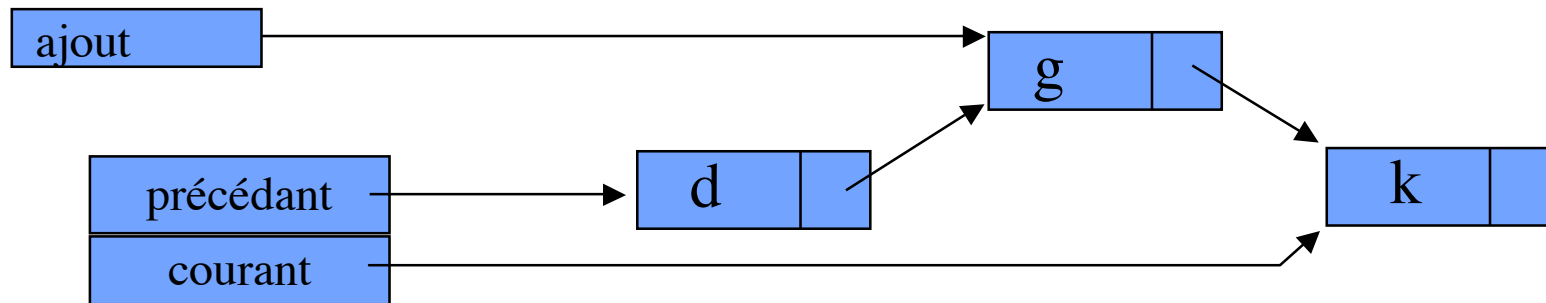
- Soit à ajouter "g" devant la position courante
- Création de la place qui contiendra g

# liste chaînée pointée (2)



- Le suivant de cette place ajoutée est la place courante

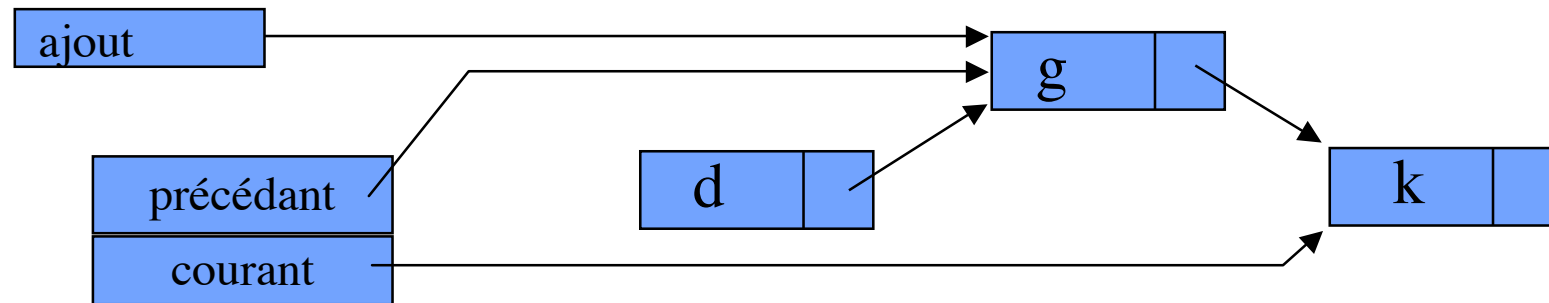
# liste chaînée pointée (2)



- Le suivant du précédent est la place ajoutée

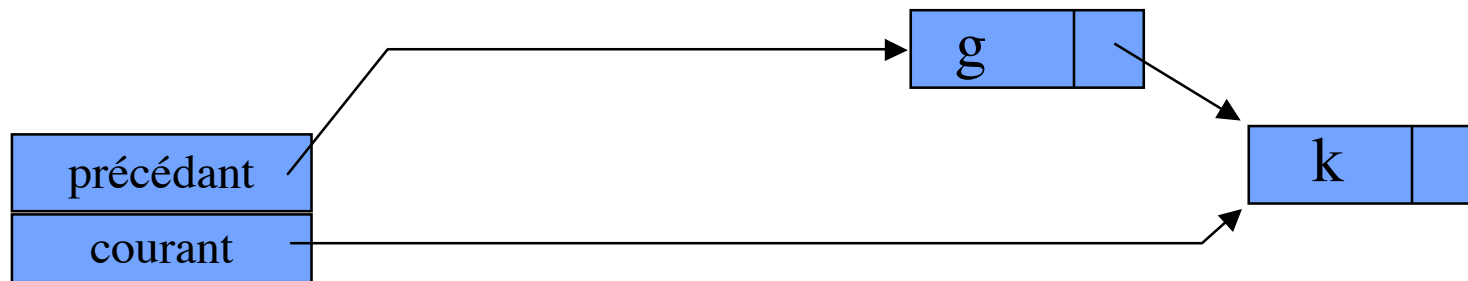


# liste chaînée pointée (2)



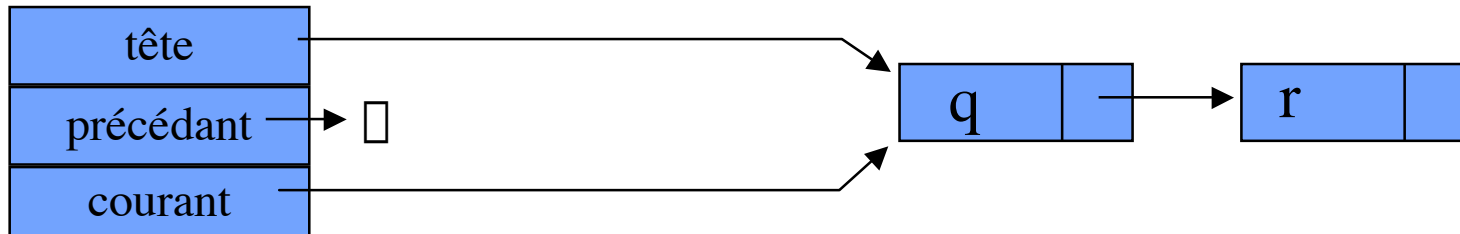
- Le précédent de la place courante est la place ajoutée

# liste chaînée pointée (2)



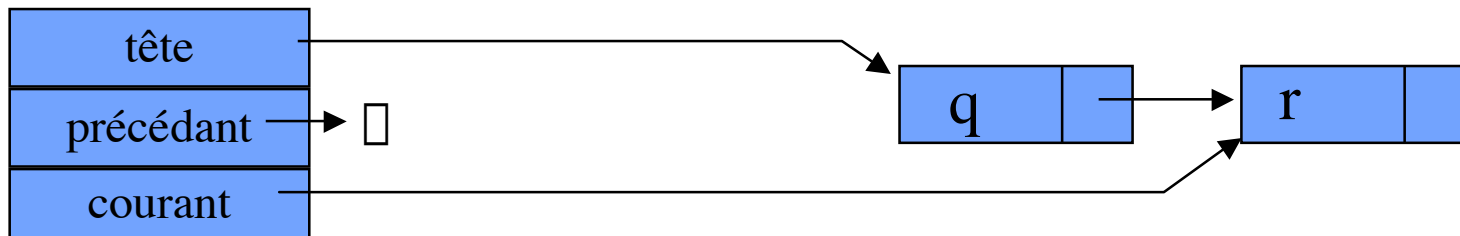
```
procedure Insérer_Avant_Courant (L: in out Liste; Val : in Element) is  
    Ajout : Pt_Place := new Place'(Contenu => Val, Suivant => L.Courant);  
begin  
    if L.Precedant = null then L.Tete := Ajout; -- debut de la liste  
    else L.Precedant.Suivant := Ajout; end if;  
    L.Precedant := Ajout;  
    L.Longueur := L.Longueur + 1;  
end Insérer_Avant_Courant;
```

# liste chaînée pointée (3)



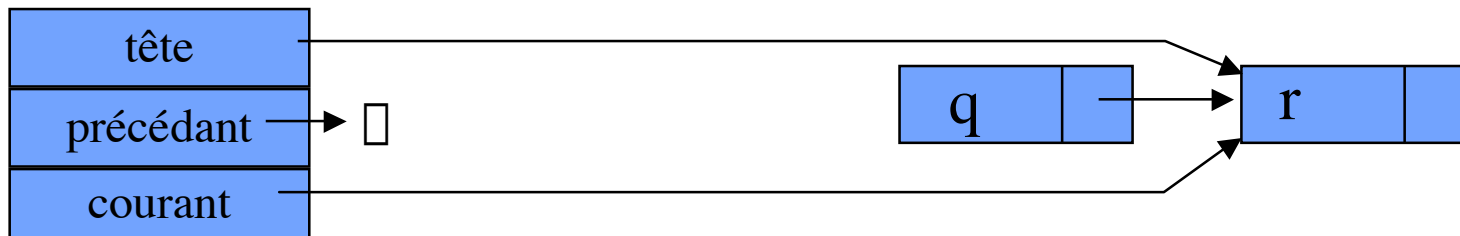
- Soit à supprimer l'élément courant, supposé en tête.

# liste chaînée pointée (3)



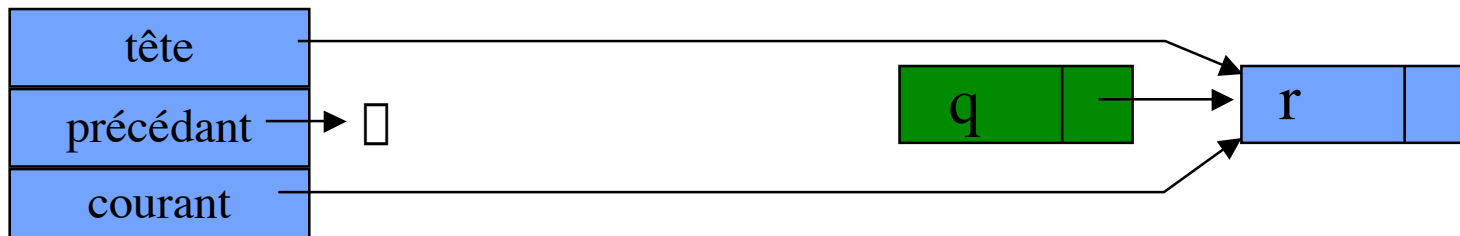
- Le courant est le suivant de l'ancien courant

# liste chaînée pointée (3)



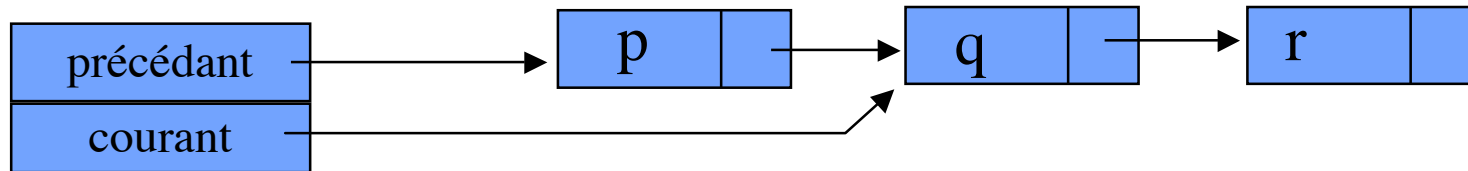
- La tête est le courant

# liste chaînée pointée (3)



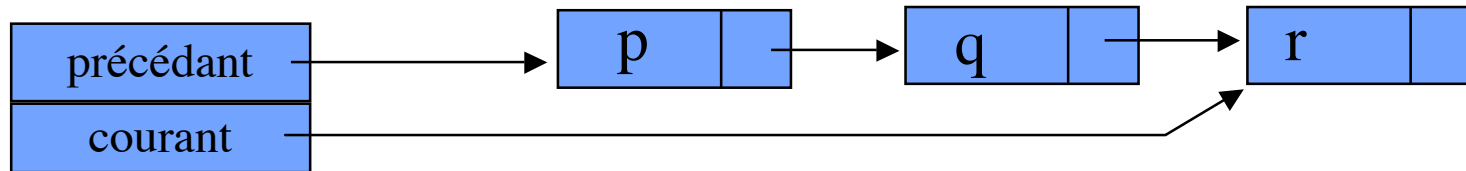
- La place devient libre

# liste chaînée pointée (3)



- Soit à supprimer l'élément courant, qui n'est pas en tête.

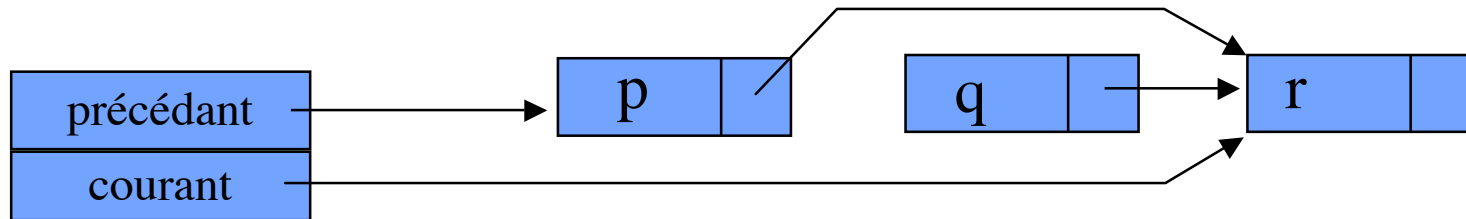
# liste chaînée pointée (3)



- Le courant est le suivant de l'ancien courant

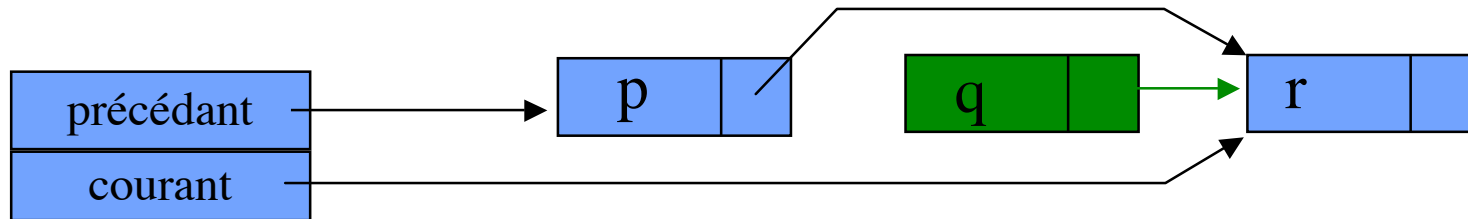


# liste chaînée pointée (3)



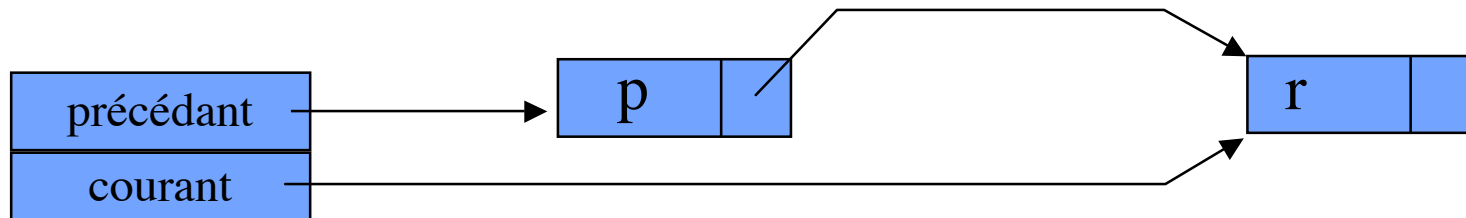
- Le suivant du précédant est le nouveau courant

# liste chaînée pointée (3)



- La place devient libre

# liste chaînée pointée (3)



```
procedure Supprimer_Courant (La_Liste : in out Liste) is  
  P : Pt_place := La_Liste.Courant;  
begin  
  if La_Liste.Courant = null then raise Erreur_Specification; end if;  
  La_Liste.Courant := La_Liste.Courant.Suivant;  
  if La_Liste.Precedant = null then La_Liste.Tete := La_Liste.Courant;  
  else La_Liste.Precedant.Suivant := La_Liste.Courant; end if;  
  La_Liste.Longueur := La_Liste.Longueur - 1;  
  Liberer (P);  
end Supprimer_Courant;
```

# Sur les listes chaînées d'objet

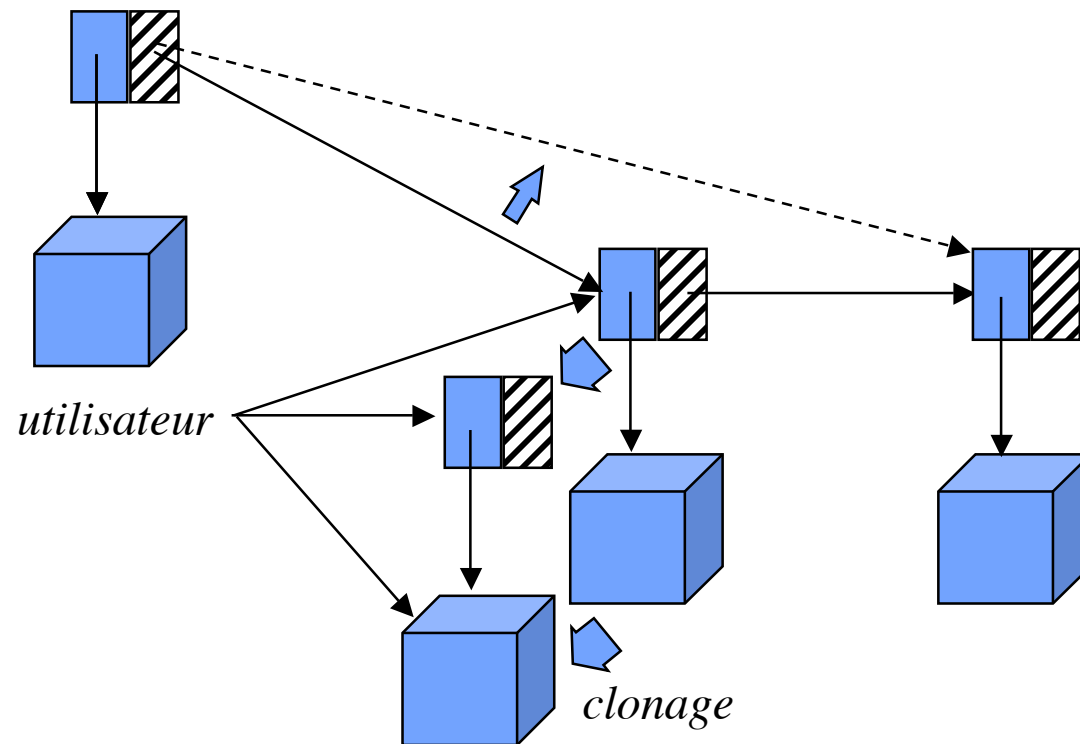
- Soit une liste chaînée d'objets en Java

vision de l'utilisateur:

- une place,
- un clone de place,
- un clone d'objet

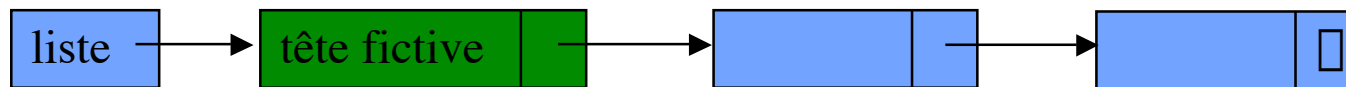


L'utilisateur peut manipuler un objet de la structure qui n'est dans aucune liste!

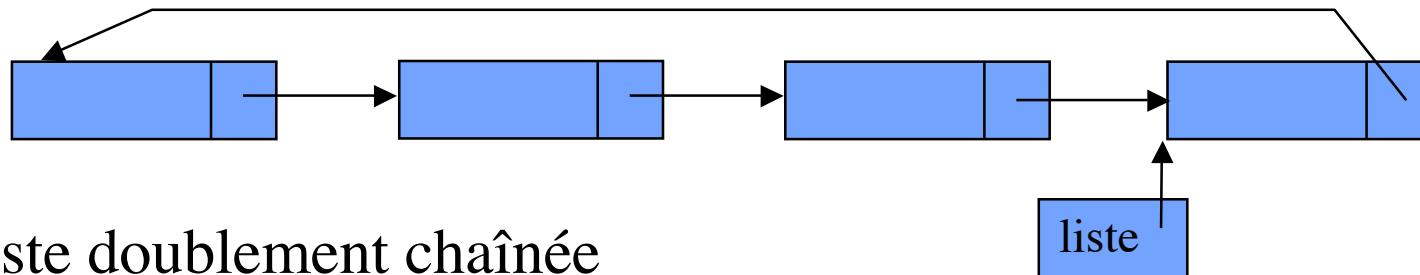


# variantes des listes chaînées

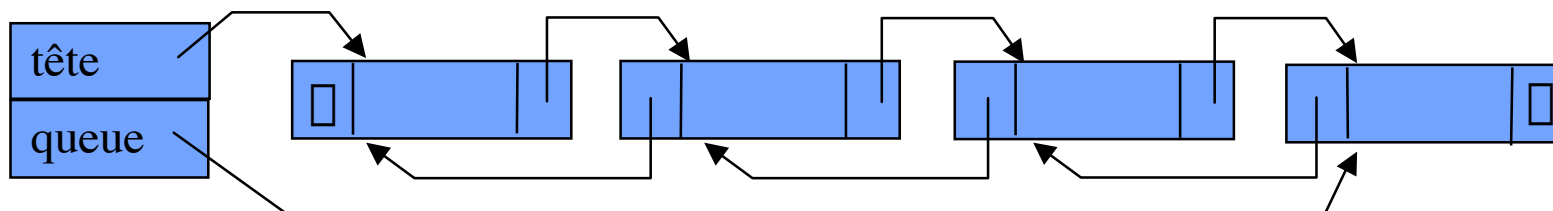
- liste avec tête fictive



- liste circulaire



- liste doublement chaînée



# les piles

## type pile

**paramètre** élément **utilise** booléen

## opérations

pilevide :  $\square$  pile  
estvide : pile  $\square$  booléen  
sommet : pile /  $\square$  élément  
empiler : pile  $\square$  élément  $\square$  pile  
dépiler : pile /  $\square$  pile

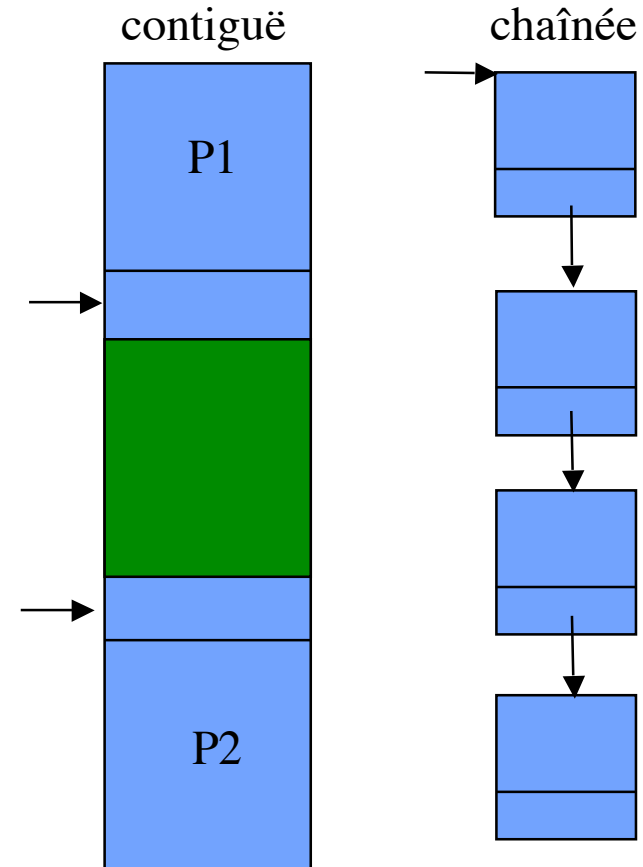
## préconditions

sommet(p):  $\neg$  estvide(p)  
dépiler(p):  $\neg$  estvide(p)

**sémantique**  $\square$  p: pile, e: élément

estvide(pilevide)  
 $\neg$  estvide(empiler(p, e))  
dépiler(empiler(p, e)) = p  
sommet(empiler(p, e)) = e

## implantation



# les files (1)

**type** file

**paramètre** élément **utilise** booléen

**opérations**

filevide :  $\square$  file

estvide : file  $\square$  booléen

premier : file /  $\square$  élément

ajouter : file  $\square$  élément  $\square$  file

retirer : file /  $\square$  file

**préconditions**

premier(f):  $\neg$  estvide(f)

retirer(f):  $\neg$  estvide(f)

**sémantique**  $\square$  f: file, e: élément

estvide(filevide)

$\neg$  estvide(ajouter(f, e))

premier(ajouter(f, e)) = **si** estvide(f) **alors** e **sinon** premier(f) **fsi**

retirer (ajouter (f, e)) = **si** estvide(f) **alors** filevide **sinon** ajouter(retirer(f),e) **fsi**

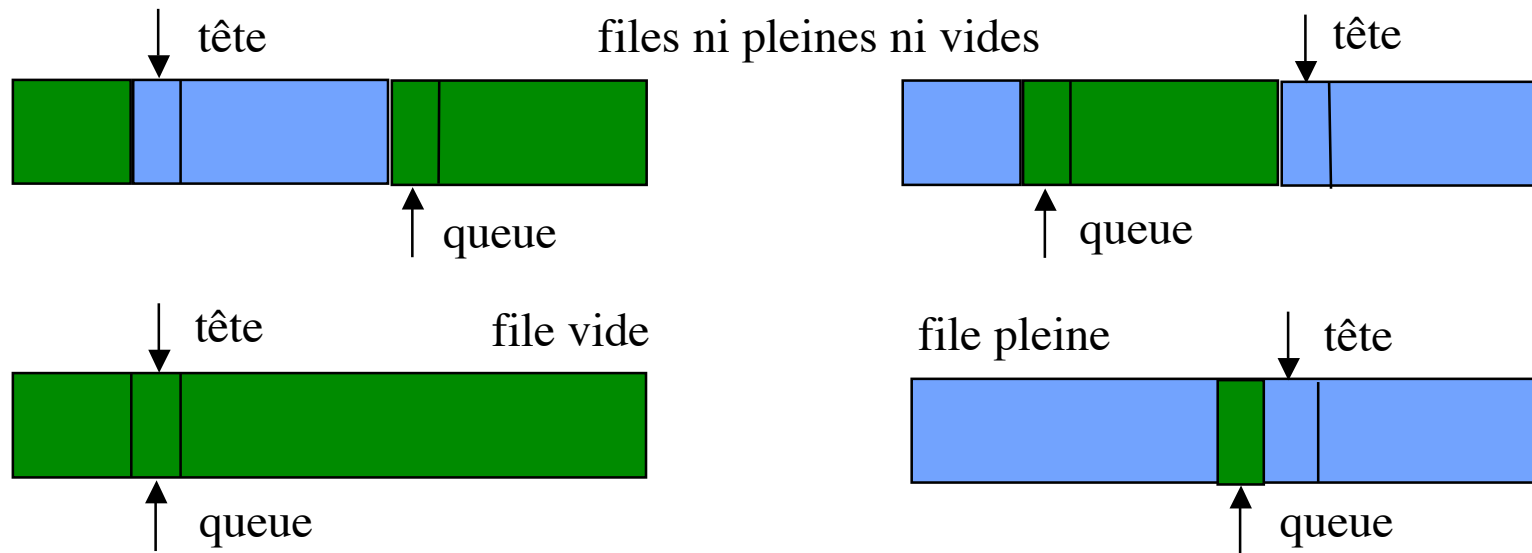
# les files (2)

- implantation contiguë

"tableau roulant" => premier repéré par *tête*, prochain ajout par *queue*

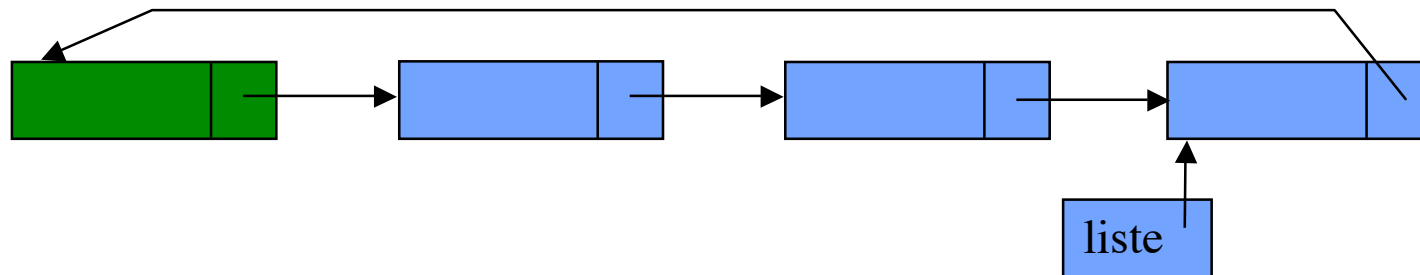
ambiguïté plein/vide => toujours un emplacement vide au moins

exemples de configurations:





# les files (3)



- représentation par liste circulaire avec tête fictive

Premier(L) => **return** L.Suivant.Suivant.Contenu;

Ajouter(L,E) =>

P := new Place'(E, L.Suivant);

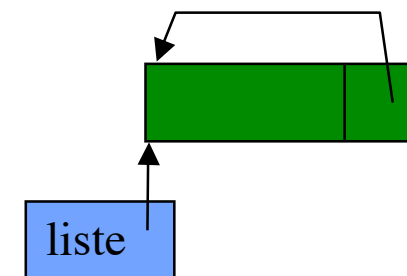
L.Suivant := P;

L := P;

Retirer(L) =>

L.Suivant.Suivant := L.Suivant.Suivant.Suivant;

ou encore => L.Suivant := L.Suivant.Suivant;



cas de la file vide

# fichiers séquentiels (1)

**type** fichier

**paramètre** article

**utilise** entier, booléen, liste [article]

**opérations**

fichiervide : □ fichier

effacer : fichier □ fichier

écrire : fichier □ article □ fichier

courant : fichier /□ article

avancer : fichier /□ fichier

finfichier : fichier □ booléen

rebobiner : fichier □ fichier

**locales**

contenu : fichier □ liste

position : fichier □ entier

# fichiers séquentiels (2)

## préconditions

écrire(f, a): finfichier(f)

courant(f):  $\neg$  finfichier(f)

avancer(f):  $\neg$  finfichier(f)

## sémantique

effacer(f) = fichiervide

contenu (f) = **si** f = fichiervide **alors** listevide

**sin**si f = avancer(f') **alors** contenu(f')

**sin**si f = rebobiner(f') **alors** contenu(f')

**sinon soit** f = écrire(f',a); insérer(contenu(f'),position(f'),a) **fsi**

position(f) = **si** f = fichiervide **alors** 1

**sin**si f = avancer(f') **alors** position (f') + 1

**sin**si f = rebobiner(f') **alors** 1

**sinon soit** f = écrire(f',a); position(f') + 1 **fsi**

courant(f) = ième(contenu(f), position(f))

finfichier(f)  $\equiv$  position(f) = longueur(contenu(f)) + 1