Plan

- Listes chaînées
- Piles

Listes chaînées

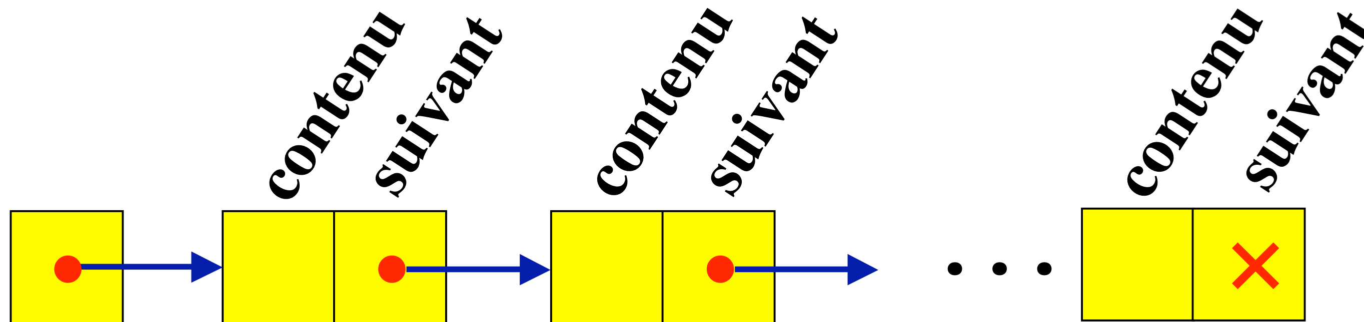
Une liste chaînée est une suite de couples formés d'un élément et de l'adresse (référence) vers l’élément suivant. C'est un jeu de piste (ou un lien dans une page).

Opérations usuelles sur les listes

- Créer une liste vide et tester si une liste est vide.
- Afficher une liste
- Ajouter un élément en tête de liste.
- Rechercher un élément.
- Supprimer un élément.
- Afficher, retourner, concaténer, dupliquer . . .

Listes chaînées en Java

```
class Liste {
  int contenu;
  Liste suivant;

  Liste(int x,Liste a) {
    contenu = x;
    suivant = a;
  }
}
```



Les opérations primitives sur les listes

```
static int tete(Liste a)
{
  return a.contenu;
}
```

non destructives !

```
static Liste queue(Liste a)
{
  return a.suivant;
}
```

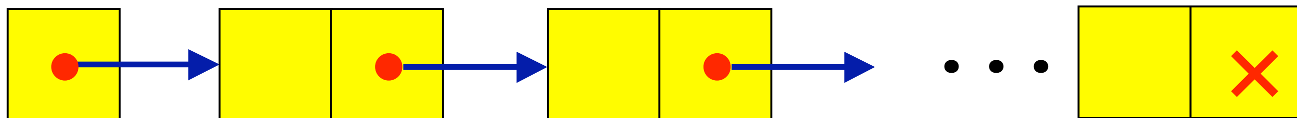
```
static Liste ajouter(int x, Liste a)
{
  return new Liste(x, a);
}
```

Identités sur les opérations de liste

- `tete(ajouter(x, a)) = x`
- `queue(ajouter(x, a)) = a`
- `ajouter(tete(a), queue(a)) = a`
  `(a ≠ `**`null`**`)`

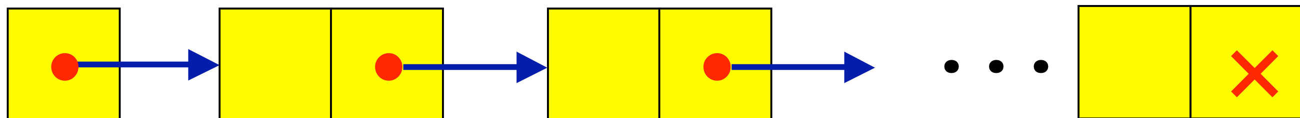
Recherche dans une liste (itérative)

```
static boolean estDansI(int x, Liste a)
{
  while (a != null)
  {
    if (a.contenu == x)
      return true;
    a = a.suivant;
  }
  return false;
}
```



Recherche itérative (variante)

```
static boolean estDansI(int x, Liste a)
{
  for ( ; a != null; a = a.suivant)
    if (a.contenu == x)
      return true;
  return false;
}
```



Recherche dans une liste (récursive)

```
static boolean estDans(int x, Liste a)
{
  if (a == null)
    return false;
  if (a.contenu == x)
    return true;
  return estDans(x, a.suivant);
}
```

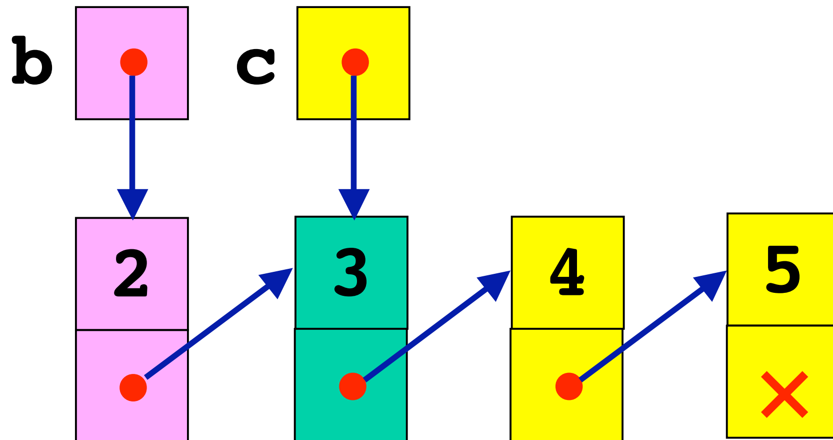
Recherche dans une liste (récursive abrégée)

```
static boolean estDans(int x, Liste a)
{
  return (a != null) && (tete(a) == x
    || estDans(x, queue(a)));
}
```

Suppression de la première occurrence de x

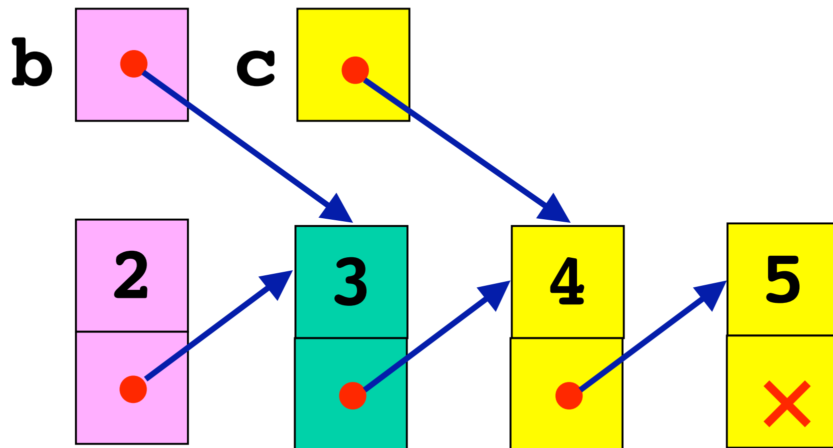
```
static Liste supprimerI(int x, Liste a){
  if (a == null)
    return a;
  if (a.contenu == x)
    return a.suivant;
  Liste b = a, c = b.suivant;
  for (; c != null; b = c, c =
c.suivant)
    if (c.contenu == x){
      b.suivant = c.suivant;
      break;
    }
  return a;
}
```

Suppression (itératif)

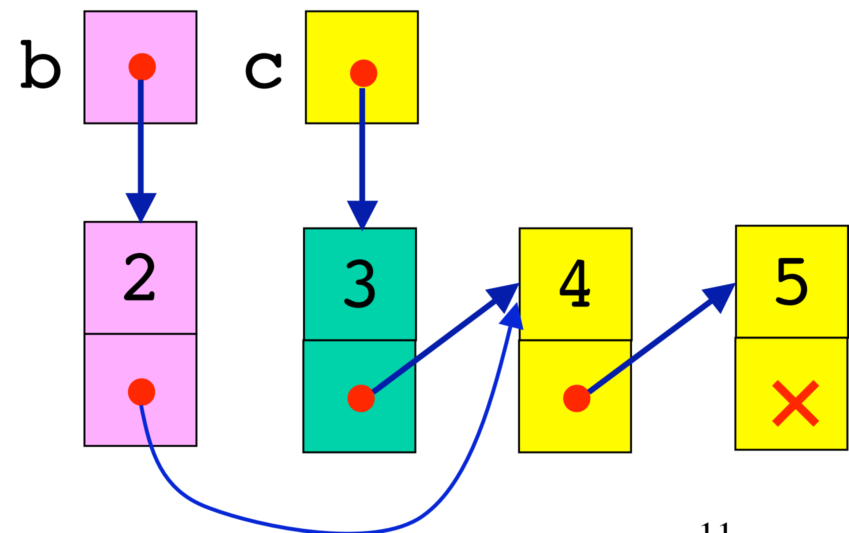



```
// invariant
b.suivant == c
```

(1) c.contenu ≠ x

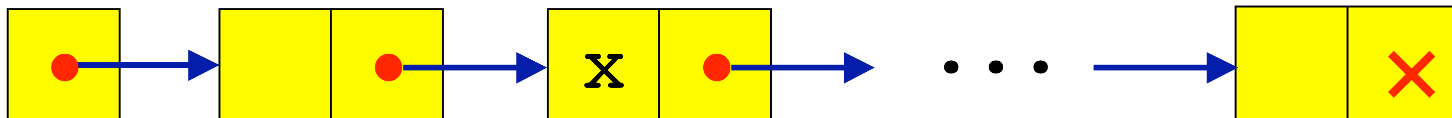


(2) c.contenu = x



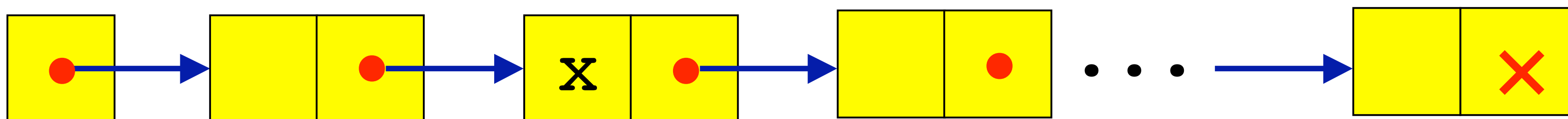
Suppression dans une liste (récursive)

```
static Liste supprimer(int x, Liste a)
{
  if (a == null)
    return a;
  if (a.contenu == x)
    return a.suivant;
  a.suivant = supprimer(x, a.suivant);
  return a; // variante dans le poly
}
```



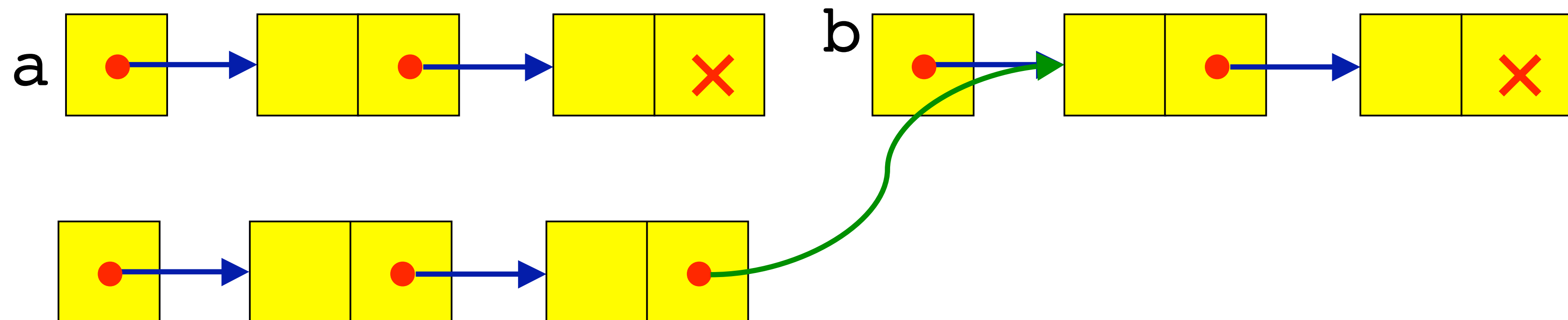
Suppression récursive en conservant la liste

```
static Liste supprimer(int x, Liste a)
{ // variante dans le poly
  if (a == null)
    return a;
  if (a.contenu == x)
    return a.suivant;
  return new Liste(a.contenu,
             supprimer(x, a.suivant));
}
```



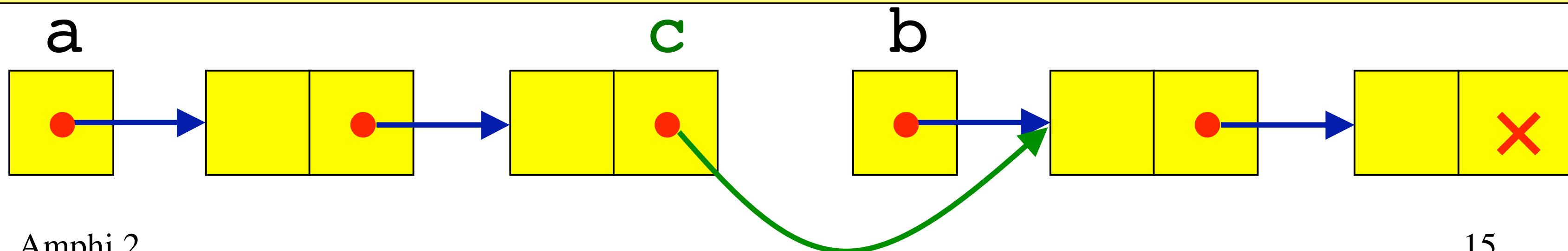
Concaténation de listes (avec copie de a)

```
static Liste concat(Liste a, Liste b)
{ // variante dans le poly
  if (a == null)
    return b;
  return ajouter(a.contenu,
                 concat(a.suivant, b);
}
```



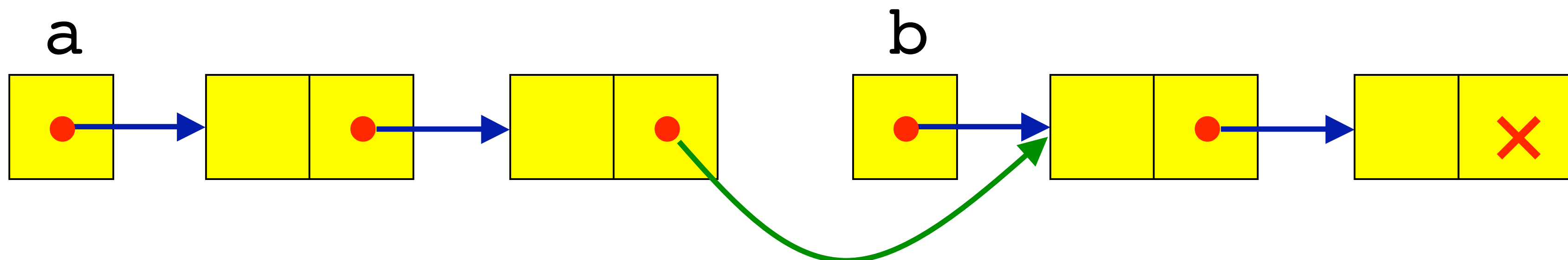
Fusion de deux listes (itératif)

```
static Liste fusionI(Liste a, Liste b)
{ // Suppose a et b distinctes
  if (a == null)
    return b;
  Liste c = a;
  while (c.suivant != null)
    c = c.suivant;
  c.suivant = b;
  return a;
}
```



Fusion de deux listes

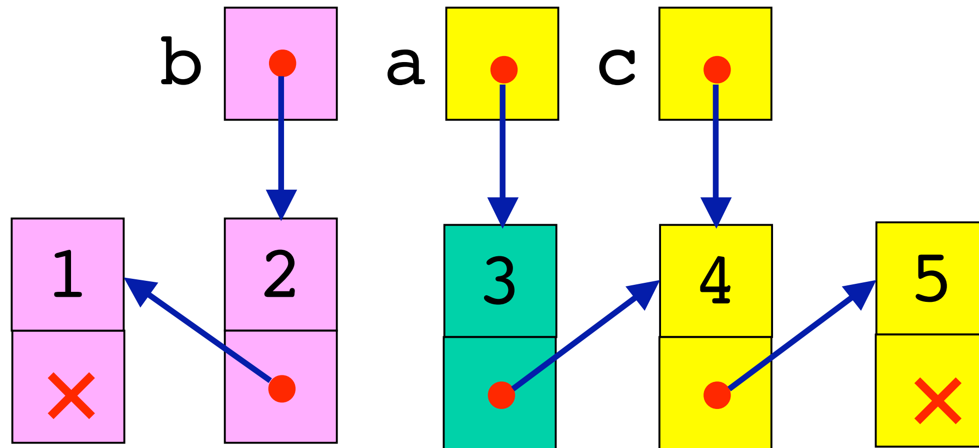
```
static Liste fusion(Liste a, Liste b)
{
  if (a == null)
    return b;
  a.suivant = fusion(a.suivant, b);
  return a;
}
```



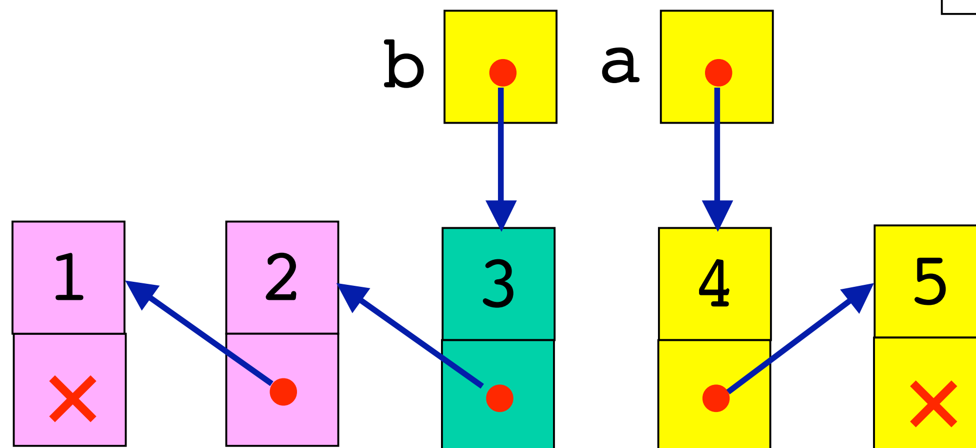
Inverser une liste (itératif, détruit la liste)

```
static Liste inverserID(Liste a)
{
  Liste b = null;
  while (a != null)
  {
    Liste c = a.suivant;
    a.suivant = b;
    b = a;
    a = c;
  }
  return b;
}
```

Inverser une liste (itératif)



On raccroche le wagon bleu au train mauve ...




```
(1) a.suivant = b
```

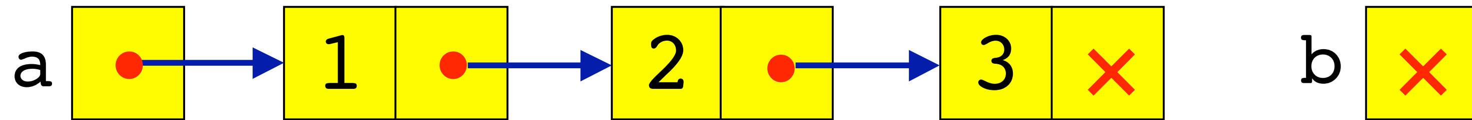
```
(2) b = a
```

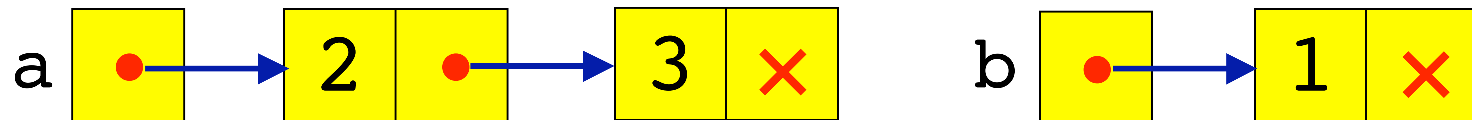
```
(3) a = c
```

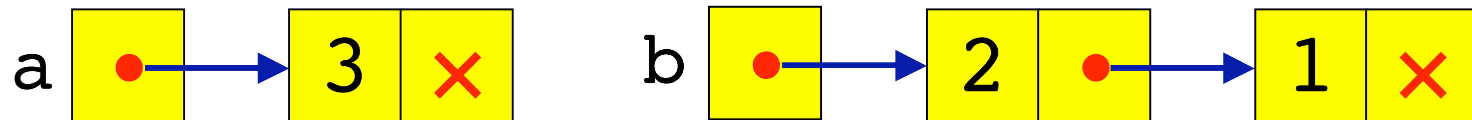
Inverser une liste (récursif, temps linéaire)

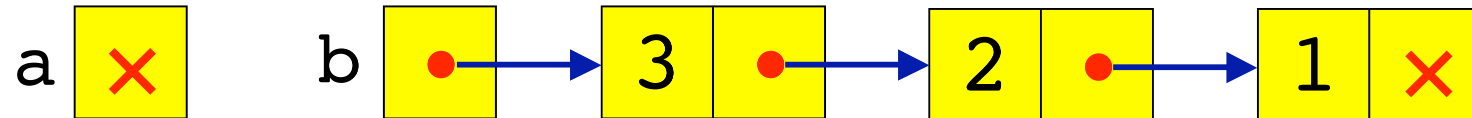
```
static Liste inverser(Liste a)
{
  return passer(a, null);
}

static Liste passer(Liste a, Liste b)
{
  if (a == null)
    return b;
  return passer(a.suivant,
                ajouter(a.contenu, b));
}
```

Inverser une liste (récursif, temps linéaire)









Copie d'une liste

```
static Liste copier(Liste a)
{
  if (a == null)
    return a;
  return ajouter(tete(a),
                 copier(queue(a));
}
```

Variation: listes gardées à la fin

```
class ListeG { //garde a la fin
  int contenu;
  ListeG suivant;

  ListeG(int x, ListeG a){
    contenu = x;
    suivant = a; // a != null !
  }

  ListeG(){
    contenu = 0;
    suivant = null;
  }
}
```

Variation: listes gardées à la fin (2)

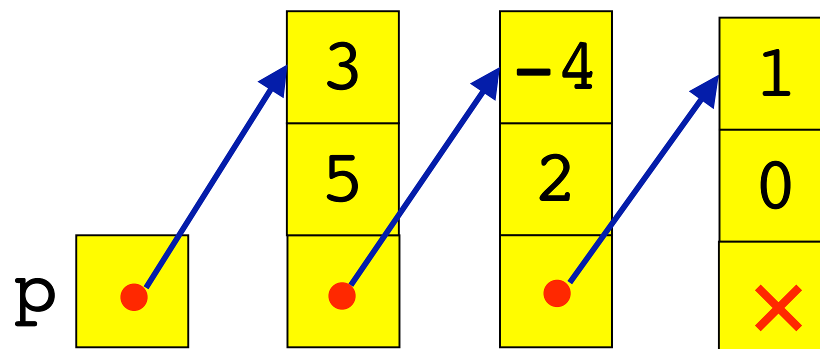
```
static boolean estVide(ListeG a)
{
  return (a.suivant == null);
}
```

Variations sur les listes

- Liste gardée à la fin : toute liste contient une dernière paire qui signale la fin.
- Liste gardée au début : toute liste contient une première paire (sans élément).
- Liste avec référence vers début et fin.
- Liste doublement chaînée : toute liste contient une référence vers la liste suivante et une référence vers la liste précédente.
- Liste circulaire : le « suivant » de la dernière cellule est la première cellule (listes gardées ou non).

Polynômes à coefficients entiers

- On écrit les polynômes par degré décroissant.
- On les représente par des listes chaînées où chaque cellule contient le coefficient et le degré d'un monôme.
- Le polynôme nul est représenté par la liste vide.




```
3X^5 - 4X^2 + 1
```

```
class Pol
{
  int coeff, degre;
  Pol suivant;

  Pol(int c, int d, Pol p)
  {
    coeff = c;
    degre = d;
    suivant = p;
  }
}
```

Dérivée d'un polynôme

```
static Pol deriver(Pol p)
{
  if (p == null || p.degre == 0)
    return null;
  Pol r = deriver(p.suivant);
  return new Pol(p.coeff * p.degre,
                 p.degre - 1, r);
  }
}
```

```
p  = 3X^5 - 4X^2 + 1
p' = 15X^4 - 8X
```

Addition de polynômes (1)

```
static Pol additionner(Pol p, Pol q)
{
  if (p == null) return q;
  if (q == null) return p;
  if (p.degre > q.degre)
  {
    r = additionner(p.suivant, q));
    return new Pol(p.coeff, p.degre, r);
  }
  if (q.degre > p.degre)
  {
    r = additionner(p, q.suivant));
    return new Pol(q.coeff, q.degre, r);
  }
// ... à suivre
```

Addition de polynômes (2)

```
static Pol additionner(Pol p, Pol q)
{
... // cas restant : p.degre == q.degre
  Pol r = additionner(p.suivant,
                               q.suivant);
  int coeff = p.coeff + q.coeff;
  if (coeff == 0)
    return r;
  return new Pol(coeff, p.degre, r);
}
```

Multiplication de polynômes (1)

```
static Pol multiMono(Pol p, Pol m)
{
  if (p == null)
    return null;
  Pol pm = multiMono(p.suivant, m);
  return new Pol(p.coeff * m.coeff,
                 p.degre + m.degre, pm);
}
```

```
(3X^5 - 4X^2 + 1)(5X^3) = 15X^8 +
(- 4X^2 + 1)(5X^3)
```

Multiplication de polynômes (2)

```
static Pol multiplier(Pol p, Pol q)
{
  if (q == null)
    return null;
  Pol r, s;
  r = multiMono(p, q);
  s = multiplier(p, q.suivant);
  return additionner(r, s);
}
```

Affichage d'un polynôme (1)

```
static void afficherSigne(int n)
{
  System.out.print(
    (n >= 0) ? " + " : " - ");
}
```

```
(booleen) ? expression1 : expression2
```

Si le booléen est **`true`** évalue
  `expression1;`

Si le booléen est **`false`** évalue
  `expression2;`

```java
static void afficherMonome(Pol p,
    boolean premierMonome){
  int a = p.coeff;
  if (premierMonome && (a < 0))
    System.out.print("-");// Evite + 5X
  else if (!premierMonome)
    afficherSigne(a);
  if ((a != 1 && a != -1) ||
    (p.degre == 0)) // Evite 1X^3
    System.out.print(Math.abs(a));
  if (p.degre > 0)  // Evite X^0
    System.out.print("X");
  if (p.degre > 1)  // Evite X^1
    System.out.print("^" + p.degre);
}
```

Affichage d'un polynôme (2)

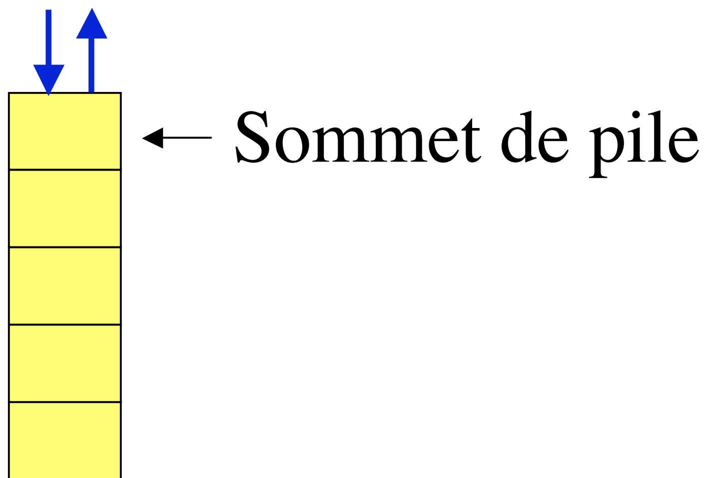
```
static void afficher(Pol p)
{
  if (p == null)
    System.out.print("0");
  else
  {
    afficherMonome(p, true);
    p = p.suivant;
    for (; p != null; p = p.suivant)
      afficherMonome(p, false);
  }
}
```

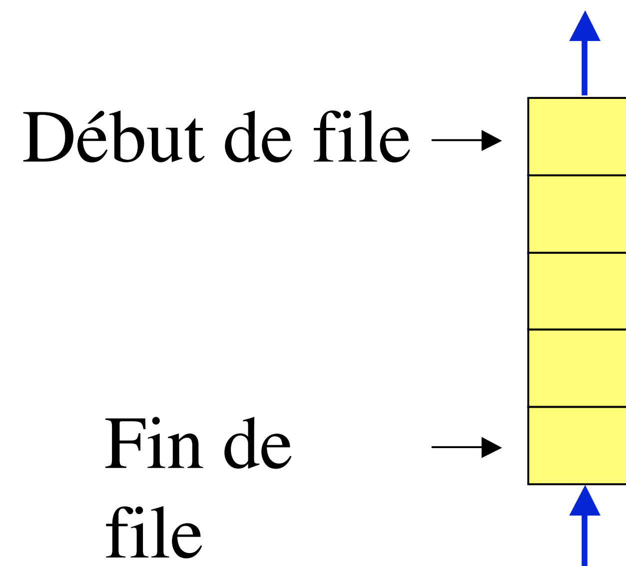
Plan

- Piles

Une pile est une liste où les insertions et les suppressions se font toutes du même côté. LIFO

Une file est une liste où les insertions se font d'un côté et les suppressions se font de l'autre côté. FIFO





Utilisation des piles et des files

- Les piles sont utilisées pour implanter les appels de procédures (cf. pile système).
- En particulier, les procédures récursives gèrent une pile de récursion.
- Evaluation d'expressions arithmétiques.
- Les files permettent de gérer des processus en attente d'une ressource du système.
- Servent de modèle pour réaliser systèmes de réservation, gestion de pistes d'aéroport, etc.

Le type abstrait de données "Pile"

Créer une pile vide.

Tester si une pile est vide.

Ajouter un élément en sommet de pile (empiler).

Valeur du sommet de pile.

Supprimer le sommet de pile (dépiler).

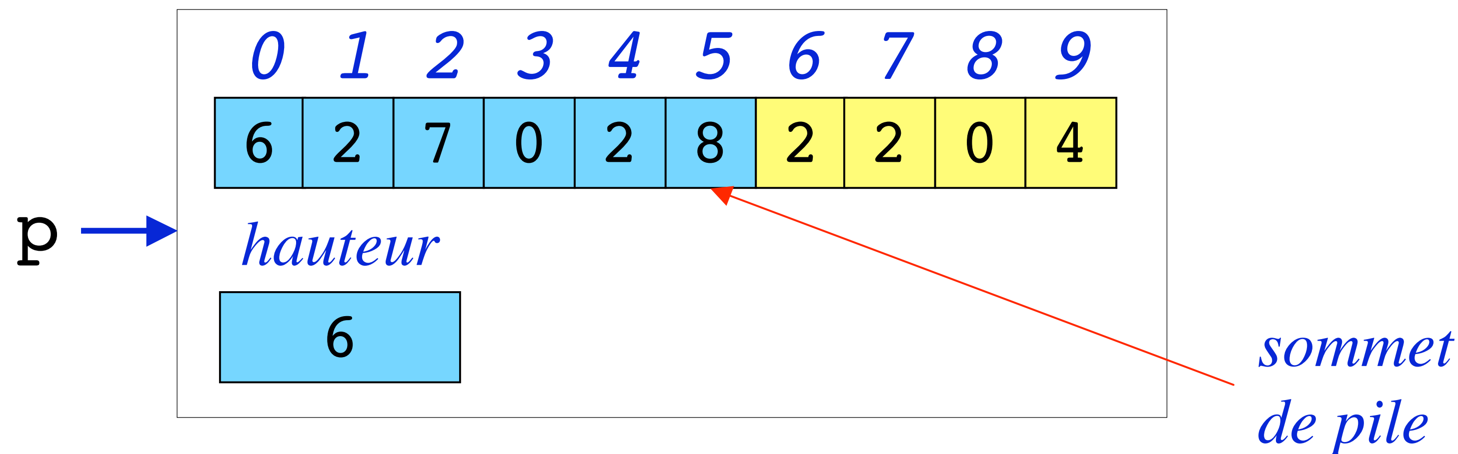
- `estVide(pileVide) = true`
- `estVide(ajouter(x, p)) = false`
- `valeur(ajouter(x, p)) = x`
- `supprimer(ajouter(x, p)) = p`

Intérêt des types abstraits

- Les types abstraits servent à utiliser les piles (resp. les files) uniquement à travers leurs fonctions de base.
- Peu importe la réalisation (qui peut varier en fonction des modules), seule compte l'interface.
- La notion d'interface et de module sera développée dans le cours « Fondements de l'informatique ».

Solution 1 : utiliser des listes.

Solution 2 : couple formé d’un tableau et de la hauteur de pile.

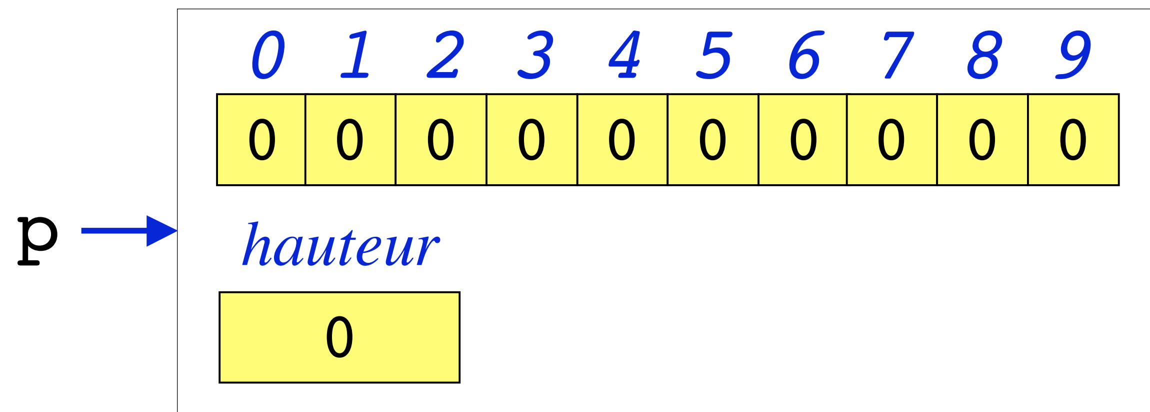


Les valeurs des cases jaunes sont inutilisées.

```
class Pile
{
  static final int maxP = 10;

  int hauteur;
  int[] contenu;

  Pile()
  {
    hauteur = 0;
    contenu = new int[maxP];
  }
}
```

Constructeur

```
Pile p = new Pile();
```



Les piles en Java (1)

```
static boolean estVide(Pile p)
{
  return p.hauteur == 0;
}

static boolean estPleine(Pile p)
{
  return p.hauteur == maxP;
}

static void vider(Pile p)
{
  p.hauteur = 0;
}
```

Vider une pile

```
static void vider(Pile p) {
  p.hauteur = 0;
}
```

Avant

0	1	2	3	4	5	6	7	8	9
6	2	7	0	2	8	0	2	3	4

hauteur : 5

Après

0	1	2	3	4	5	6	7	8	9
6	2	7	0	2	8	0	2	3	4

hauteur : 0

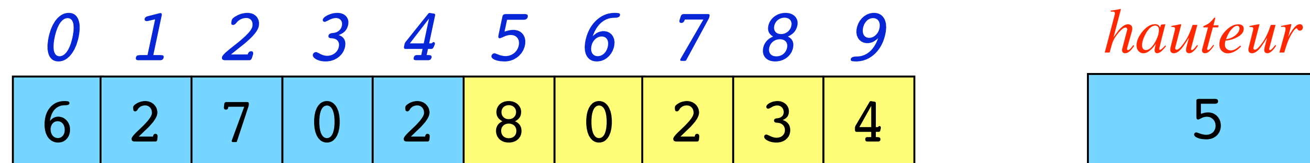
Rappel : les valeurs des cases jaunes sont inutilisées.

Les piles en Java (2)

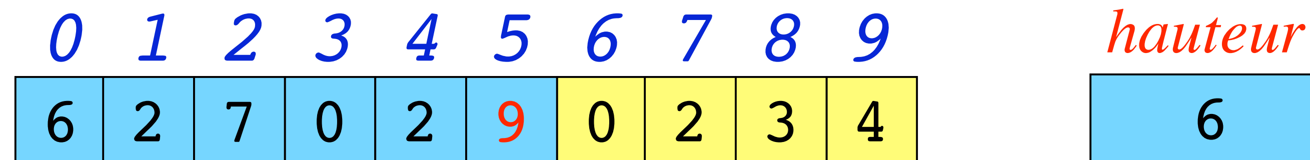
```
static int valeur(Pile p)
{
  return p.contenu[p.hauteur-1];
}

static void ajouter(int x, Pile p)
{
  p.contenu[p.hauteur++] = x;
}

static void supprimer(Pile p)
{
  --p.hauteur;
}
```

Ajouter au sommet de la pile

```
static void ajouter(int x, Pile p){
  p.contenu[p.hauteur++] = x;
}
```



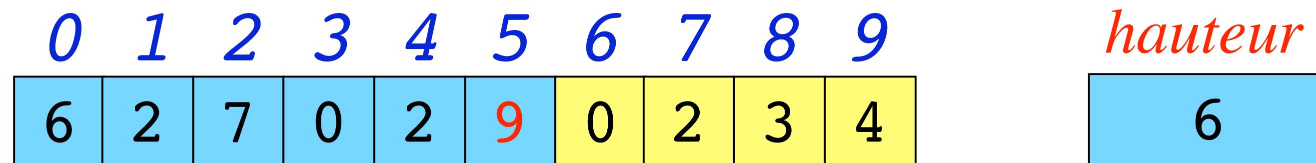
Ajouter 9

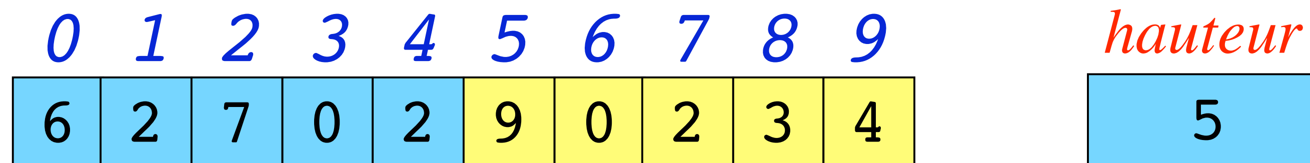



```
(1) p.contenu[p.hauteur] = x;
(2) p.hauteur++;
```

Suppression du sommet de pile

```java
static void supprimer(Pile p){
  --p.hauteur;
}
```





Supprimer 9 de la pile. Les valeurs des cases jaunes sont inutilisées.