

INF421-a

Bases de la programmation et de l'algorithmique

(Bloc 2 / 9)

Philippe Baptiste

CNRS LIX, École Polytechnique

2 septembre 2005

L'évaluation du module

- ▶ Un CC
- ▶ Un TP noté (le 5ème) avec une application du cours
- ▶ $\max(CC, \frac{3}{4}CC + \frac{1}{4}TP)$

Aujourd'hui

Les listes

Piles et Files

Le $k^{\text{ème}}$ élément : Une version itérative

```
static int nth(int i, Liste p) {  
    for (; p ≠ null; p = p.suivant) {  
        if (i == 0) return p.contenu;  
        i--;  
    }  
    throw new Error("Erreur d'index ");  
}
```

A propos de la dernière ligne `throw new Error()` ;

- ▶ On est à la fin de la liste et $i \neq 0$ et initialement $i \notin [0, length(p)]$
- ▶ Il s'agit donc de "générer" une erreur
 - ▶ Des détails sur la gestion des erreurs plus tard

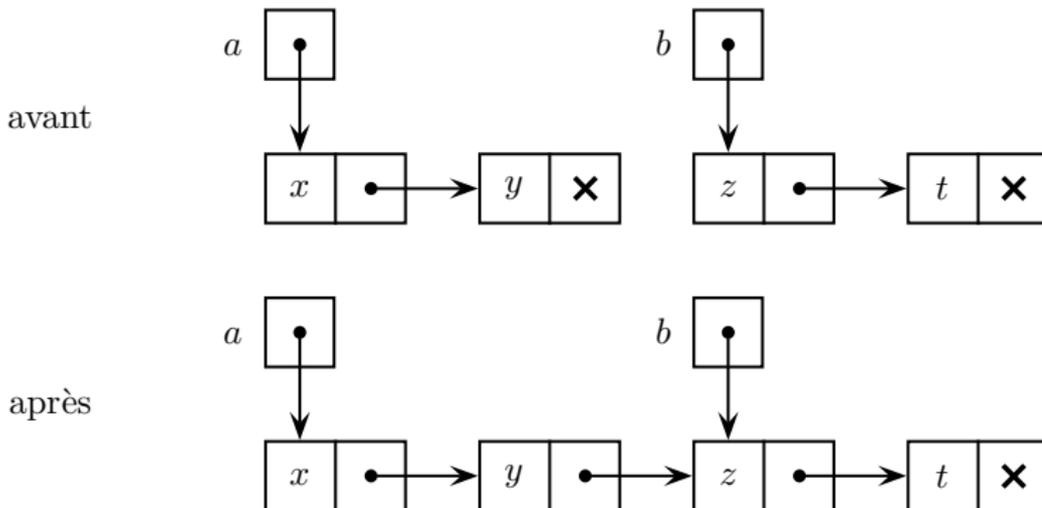
Le $k^{\text{ème}}$ élément : Une version itérative

```
static int nth(int i, Liste p) {
    for (; p ≠ null; p = p.suivant) {
        if (i == 0) return p.contenu;
        i--;
    }
    throw new Error("Erreur d'indice");
} ...
Liste l = new Liste(7, new Liste(9, new Liste(3, new Liste(1, null))));
System.out.println("l[" + 3 + "] = " + nth(3, l));
System.out.println("l[" + 4 + "] = " + nth(4, l));
```

```
~phb > java test
l[3]= 1
Exception in thread "main" java.lang.Error: Erreur d'indice
    at test.nth(test.java:222)
    at test.main(test.java:228)
```

Concaténation de deux listes

La concaténation de deux listes a et b produit une liste obtenue en ajoutant les éléments de la liste b à la fin de a



Concaténation destructive de deux listes

Attention destructif !

```
static Liste dernier(Liste a) {  
    if (a == null)  
        return null;  
    while (a.suivant  $\neq$  null)  
        a = a.suivant;  
    return a;  
}  
static Liste Concaténation(Liste a, Liste b) {  
    if (a == null)  
        return b;  
    dernier(a).suivant = b;  
    return a;  
}
```

Inversion (rappel)

Construire une liste qui contient les mêmes éléments à l'envers : $(1, 2, 3) \rightarrow (3, 2, 1)$

```
static Liste inverser(Liste a) {  
    Liste b = null;  
    while (a  $\neq$  null) {  
        b = new Liste(a.contenu, b);  
        a = a.suivant;  
    }  
    return b;  
}
```

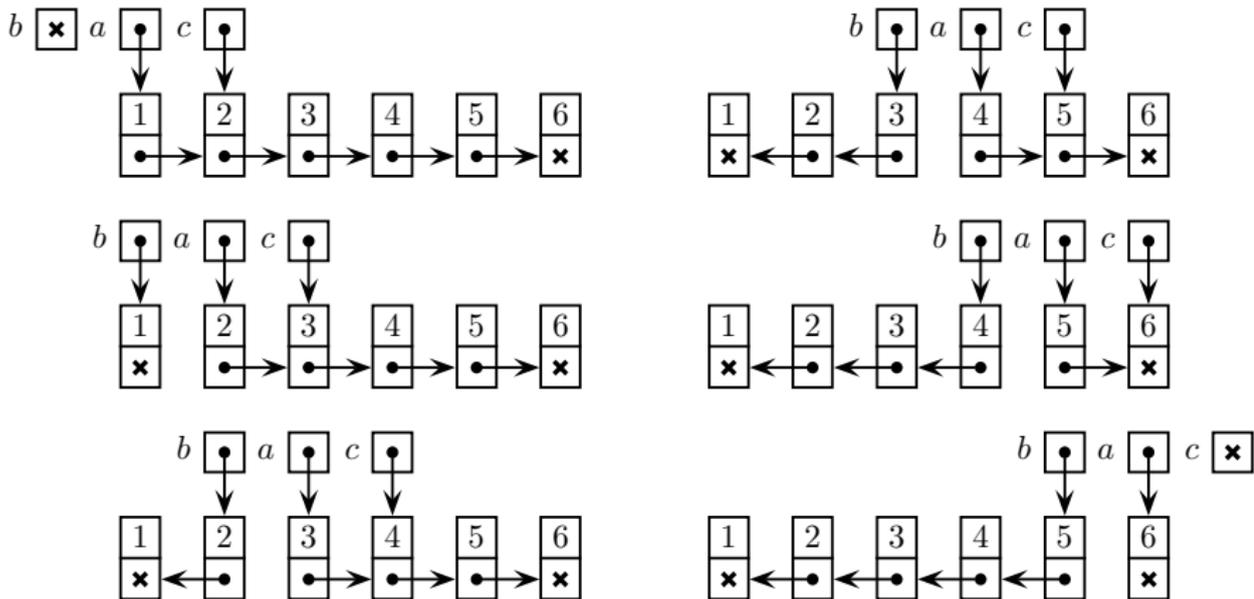
Inversion en place

On ne veut pas allouer la mémoire → Inverser les flèches.

Attention destructif!

```
static Liste inverserEnPlace (Liste a) {
    Liste b = null;
    while (a ≠ null) {
        Liste c = a.suivant;
        a.suivant = b; // inverse
        b = a;
        a = c;
    }
    return b;
}
```

Inversion en place



Fusionner deux listes triées

- ▶ Deux listes $\mathcal{L}_1, \mathcal{L}_2$ triées que l'on souhaite fusionner (en respectant le tri)
- ▶ $(1, 6, 12, 67, 98, 454), (5, 6, 11, 32, 123, 324, 444) \rightarrow (1, 5, 6, 6, 11, 12, 32, 67, 98, 123, 324, 444, 454)$
- ▶ Idée 1 : Concaténer puis trier.
 - ▶ Très mauvaise idée. Pourquoi ?
- ▶ Mieux :
 - ▶ Construire une liste vide \mathcal{L}
 - ▶ Tant que les deux listes \mathcal{L}_1 et \mathcal{L}_2 ne sont pas vides :
 - ▶ Prendre l'élément de tête le plus petit
 - ▶ L'enlever de la sous-liste et l'ajouter à \mathcal{L}

Complexité ?

Fusionner deux listes triées

```
static Liste fusion(Liste l1, Liste l2) {  
    if (l1 == null)  
        return l2;  
    if (l2 == null)  
        return l1;  
    if (l1.contenu < l2.contenu)  
        return new Liste(l1.contenu, fusion(l1.suivant, l2));  
    else  
        return new Liste(l2.contenu, fusion(l1, l2.suivant));  
}
```

Destructif ou pas ?

Tri par fusion

Quelques idées simples :

- ▶ Les listes de moins de 2 éléments sont triées
- ▶ Les autres peuvent être coupées en 2 sous-listes de longueurs égales
- ▶ Les 2 sous-listes sont triées récursivement
- ▶ puis fusionnées

Complexité :
 $O(n \log n)$.
Von Neuman 1945
(©USPS)



Le tri par fusion par l'exemple

```

tri 69 85 34 24 40 77 64 22
  tri 69 85 34 24
    tri 69 85
      tri 69
      tri 85
    fusion 69 et 85 -> 69 85
  tri 34 24
    tri 34
    tri 24
  fusion 34 et 24 -> 24 34
fusion 69 85 et 24 34 -> 24 34 69 85
tri 40 77 64 22
  tri 40 77
    tri 40
    tri 77
  fusion 40 et 77 -> 40 77
  tri 64 22
    tri 64
    tri 22
  fusion 64 et 22 -> 22 64
fusion 40 77 et 22 64 -> 22 40 64 77
fusion 24 34 69 85 et 22 40 64 77 -> 22 24 34 40 64 69 77 85

```

Tri par fusion

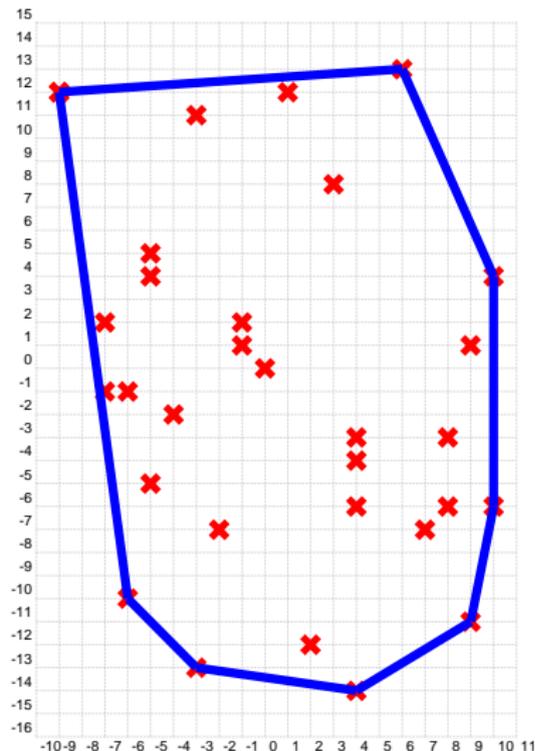
```
static Liste tri(Liste l) {  
    int n = longueur(l);  
    if (n ≤ 1)  
        return l;  
    Liste l1 = l;  
    for (int i = 0; i < n / 2 - 1; i++)  
        l = l.suivant;  
    Liste l2 = l.suivant;  
    l.suivant = null;  
    return fusion(tri(l1), tri(l2));  
}
```

Inconvénients :

- ▶ Espace mémoire
- ▶ Pour 3000 valeurs, 1.3 s vs. 0.2 s (pour `Arrays.sort`)
- ▶ **Complexité ?**

Calculer l'enveloppe convexe d'un nuage de points

- ▶ Définir et manipuler des points
- ▶ Définir et manipuler des polygones (insertion, suppression d'un sommet)
- ▶ Calculer l'enveloppe convexe d'un nuage
- ▶ **H**ypothèses
 - ▶ Les sommets sont tous distincts
 - ▶ Trois sommets quelconques ne sont pas alignés (\neq figure)

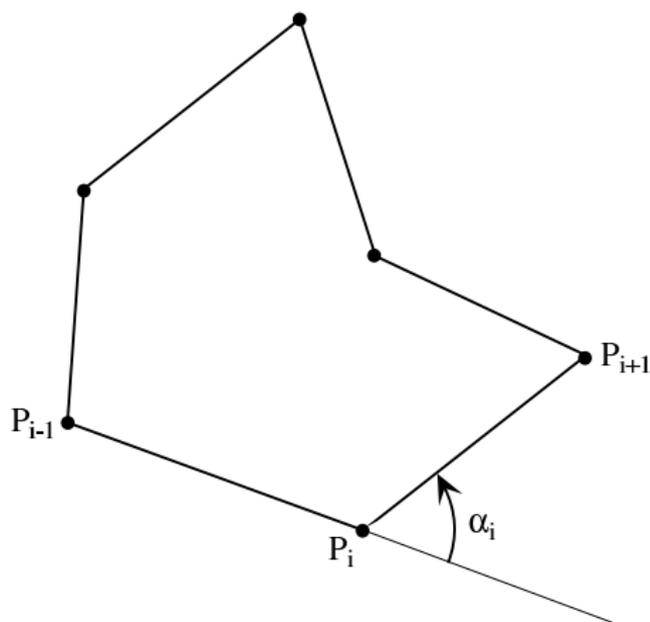


Polygones : Définitions

Un sommet P_i est dit

$$\alpha_i = \overrightarrow{(P_{i-1}P_i, P_iP_{i+1})} \geq 0.$$

Le polygone est dit *convexe* ssi tous ses sommets le sont.



Polygones, Points : modèle & objets

- ▶ Point du plan = couple (x, y) ; nuage = liste de sommets
- ▶ Polygone = liste ordonnée de sommets
- ▶ Sommet = 1 point

```
class Point {
    double x, y;
    Point (int x, int y) {
        this.x = ((double) x);
        this.y = ((double) y);
    }
    Point (double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

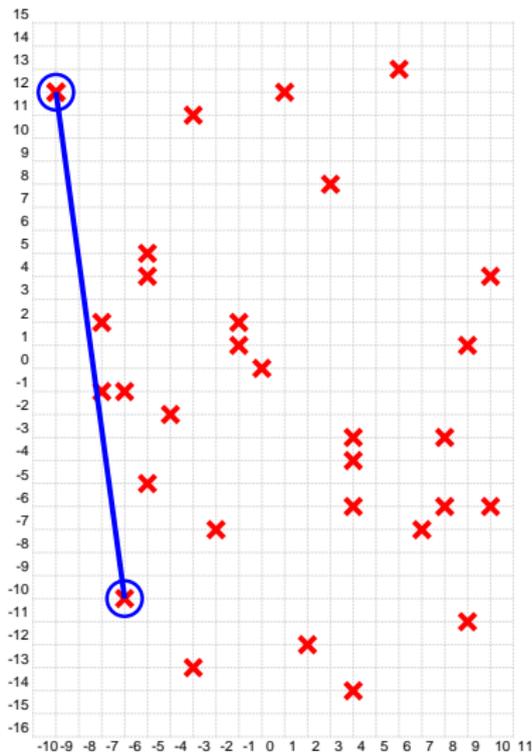
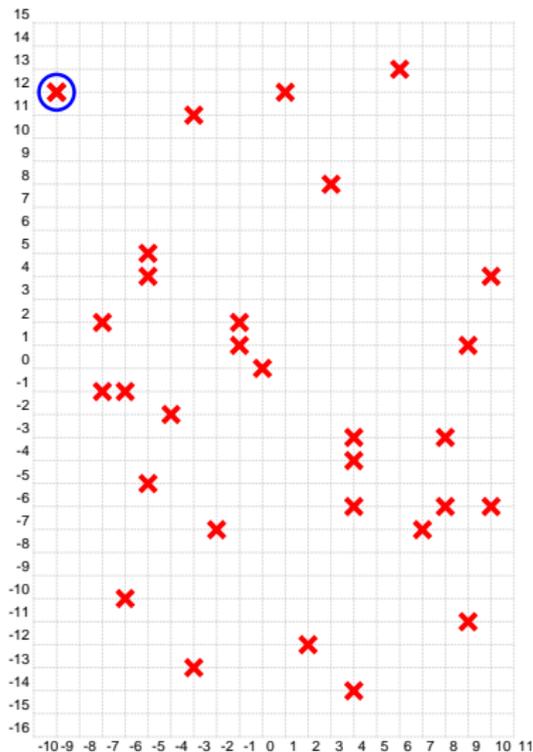
```
class Polygone {
    Polygone suivant; // LA SUITE
    Point s;
    // LE SOMMET
    Polygone(Point p, Polygone l) {
        s = p;
        suivant = l;
    }
    static Polygone
    envConvexe(Polygone p) {
        ???
    }
}
```

Algorithme de l'enveloppement

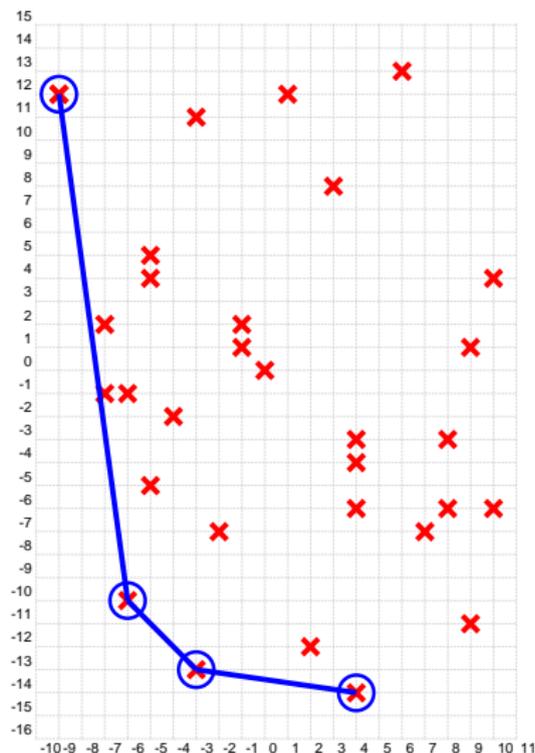
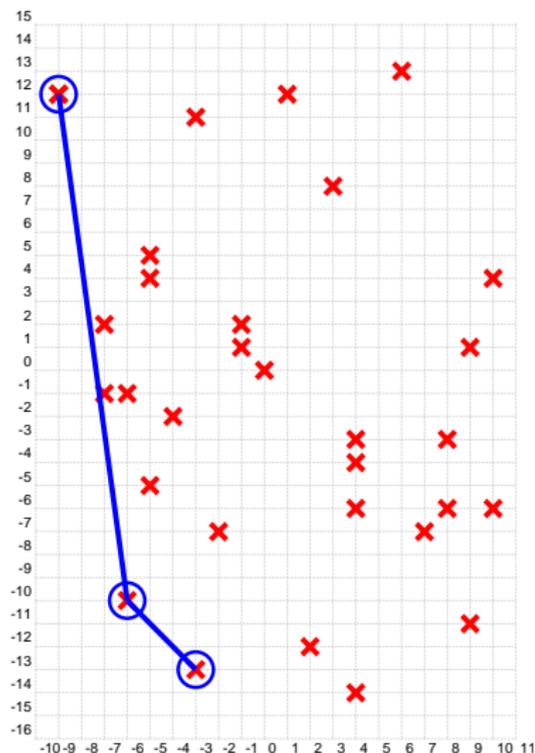
- ▶ Partir d'un point dont on sait qu'il est sur l'enveloppe convexe.
 - ▶ Ex : Un point d'abscisse minimale
- ▶ Envelopper l'ensemble de points en faisant tourner une demi-droite issue de ce point jusqu'à ce qu'elle s'appuie sur un point du nuage.
- ▶ Repartir de ce point jusqu'à retomber sur le point de départ.

Complexité ??

L'enveloppement par l'exemple



L'enveloppement par l'exemple



Algorithme de l'enveloppement

Les étapes

- ▶ Calculer un point P_0 d'abscisse minimale (**méthode premierPoint**)
 - ▶ C'est un point de l'enveloppe
- ▶ Le cas particulier du deuxième point P_1 (**méthode deuxièmePoint**)
 - ▶ Il minimise l'angle $(-\vec{j}, \overrightarrow{P_0P_1})$
- ▶ Tant qu'on a pas retrouvé le point P_0 (**méthode envConvexe**)
 - ▶ Chercher le point P_{i+1} qui minimise l'angle $(\overrightarrow{P_{i-1}P_i}, \overrightarrow{P_iP_{i+1}})$ (**méthode prochainPoint**)

Algorithme de l'enveloppement : géométrie (classe Point)

```
static double distance(Point p, Point q) {
    double dx = q.x - p.x; double dy = q.y - p.y;
    return Math.sqrt(dx * dx + dy * dy);
}
static double mes(Point p0, Point p1, Point p2) { // mesurer l'angle
    double d01 = distance(p0, p1); double d12 = distance(p1, p2);
    double pscal = (p1.x - p0.x) * (p2.x - p1.x)
        + (p1.y - p0.y) * (p2.y - p1.y);
    double alpha = Math.acos(pscal / (d01 * d12));
    double pvect = (p1.x - p0.x) * (p2.y - p1.y)
        - (p1.y - p0.y) * (p2.x - p1.x);
    if (pvect < 0) alpha = - alpha;
    return alpha;
}
static double angleVertical(Point p1, Point p2) {
    return mes(new Point(p1.x, p1.y - 1), p1, p2);
}
```

Algorithme de l'enveloppement : le 1er point

Calculer un point d'abscisse minimale

```
static Point premierPoint(Polygone nuage) {  
    Point le1er = null;  
    for (Polygone i = nuage; i  $\neq$  null; i = i.suivant)  
        if ((le1er == null) || (i.s.x < le1er.x)) le1er = i.s;  
    return le1er;  
}
```

Attention au test (le1er == null)

Algorithme de l'enveloppement : le 2ème point

Calculer un point qui minimise l'angle $(-\vec{j}, \overrightarrow{P_0P_1})$

```

static Point deuxièmePoint(Point le1er, Polygone nuage) {
    Point le2eme = null;
    for (Polygone i = nuage; i  $\neq$  null; i = i.suivant)
        if ((i.s  $\neq$  le1er) &&
            ((le2eme == null) ||
             Point.angleV(le1er, le2eme) > Point.angleV(le1er, i.s)))
            le2eme = i.s;
    return le2eme;
}

```

Algorithme de l'enveloppement : le i ème point

```
static Point prochainPoint(Point avDer, Point der, Polygone nuage) {  
    Point pchn = null;  
    for (Polygone i = nuage; i  $\neq$  null; i = i.suivant)  
        if ((i.s  $\neq$  avDer && i.s  $\neq$  der) &&  
            (pchn == null ||  
             Point.mes(avDer, der, i.s) < Point.mes(avDer, der, pchn)))  
            pchn = i.s;  
    return pchn;  
}
```

Algorithme de l'enveloppement : Itération principale

```
static Polygone envConvexe(Polygone nuage) {  
    Point le1er = premierPoint(nuage);  
    Point le2eme = deuxiemePoint(le1er, nuage);  
    Polygone env = new Polygone(le2eme, new Polygone(le1er, null));  
    for (Point pch = prochainPoint(env.suivant.s, env.s, nuage);  
        pch  $\neq$  le1er;  
        pch = prochainPoint(env.suivant.s, env.s, nuage))  
        env = new Polygone(pch, env);  
    return env;  
}
```

Les listes : pour conclure

Un extrait des 1001 variantes des listes

- ▶ Listes doublement chaînées
- ▶ Listes circulaires
- ▶ Listes gardées

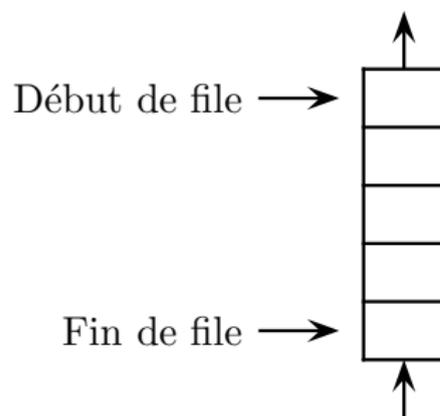
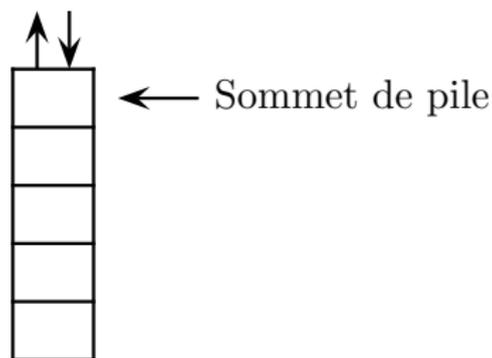
Aujourd'hui

Les listes

Piles et Files

Pile, file : définitions

- ▶ Une **pile** est une structure de données où les insertions et les suppressions se font toutes du même côté.
 - ▶ LIFO (last-in first-out).
- ▶ Une **file** est une structure où les insertions se font d'un côté et les suppressions de l'autre côté. Une telle structure est aussi appelée FIFO (first-in first-out).
 - ▶ FIFO (first-in first-out).



Pile, file : Utilité

Quelques exemple en informatique

- ▶ Pile d'appel dans un programme
- ▶ Pile d'annulation des actions dans un éditeur de texte
- ▶ File d'attente devant une imprimante
- ▶ Plus généralement la communication entre des systèmes est toujours assurée par des piles ou des files d'attente

Ce sont aussi des outils utilisés et étudiés en RO stochastique.

Pile, file : Ce qu'on veut en faire

- ▶ **créer** une pile/file vide.
- ▶ **tester** si une pile/file est vide.
- ▶ **ajouter** un élément à la pile/file.
 - ▶ au sommet de la pile
 - ▶ en fin de file
- ▶ **trouver** l'élément à dépiler/défiler
 - ▶ au sommet de la pile
 - ▶ en début de file
- ▶ **supprimer** un élément de la pile / file
 - ▶ au sommet de la pile
 - ▶ en début de file

Les files

Comment coder une file ?

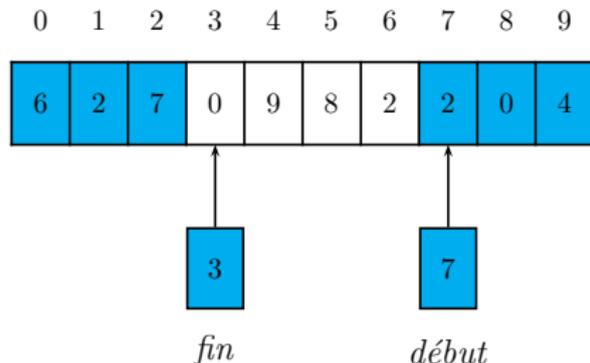
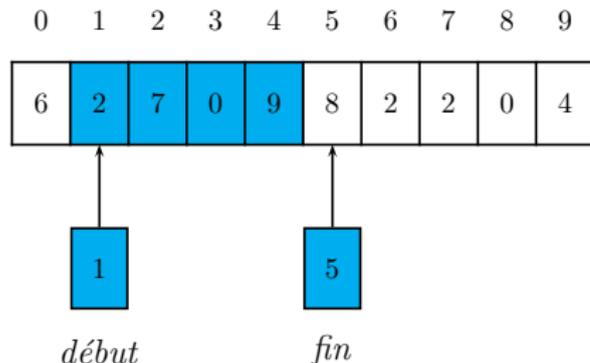
- ▶ Par un tableau
 - ▶ Limitation (taille du tableau)
 - ▶ Gestion des indices complexes
- ▶ Par une liste chaînée

Les deux implantations se présentent de la même manière a un utilisateur d'une classe `File`. On fait "*abstraction*" de l'implantation effective.

Les files avec un tableau

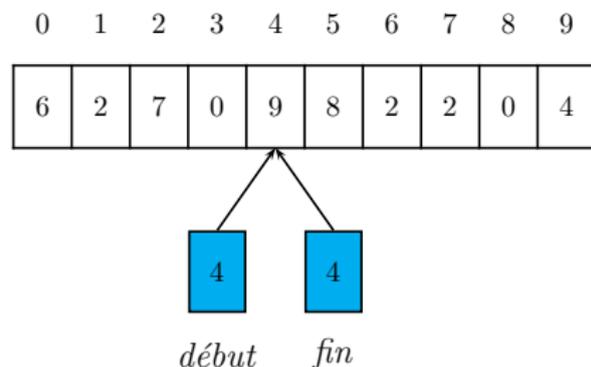
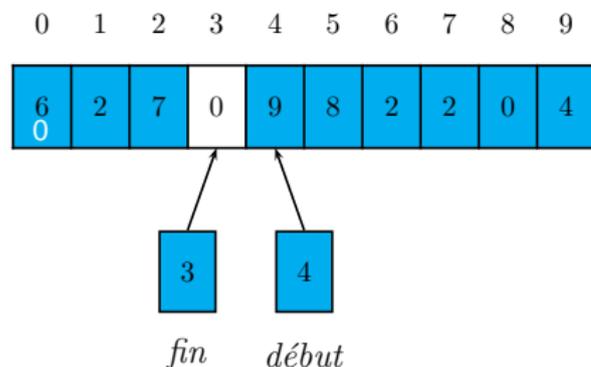
Idée simple : Utiliser un tableau pour enregistrer les éléments de la file.

- ▶ Attention : Le début **ET** la fin de la file changent
- ▶ Utilisation d'un tableau **contenu** "circulaire" de taille maxF
 - ▶ Les éléments de la file sont ceux compris entre début et fin modulo maxF



Les files avec un tableau : File vide, file pleine

- ▶ Une file vide : $\text{début} = \text{fin}$
- ▶ Une file pleine : $\text{fin} + 1 \equiv \text{début} \pmod{n}$
- ▶ On perd ainsi une place dans le tableau



Les files avec un tableau

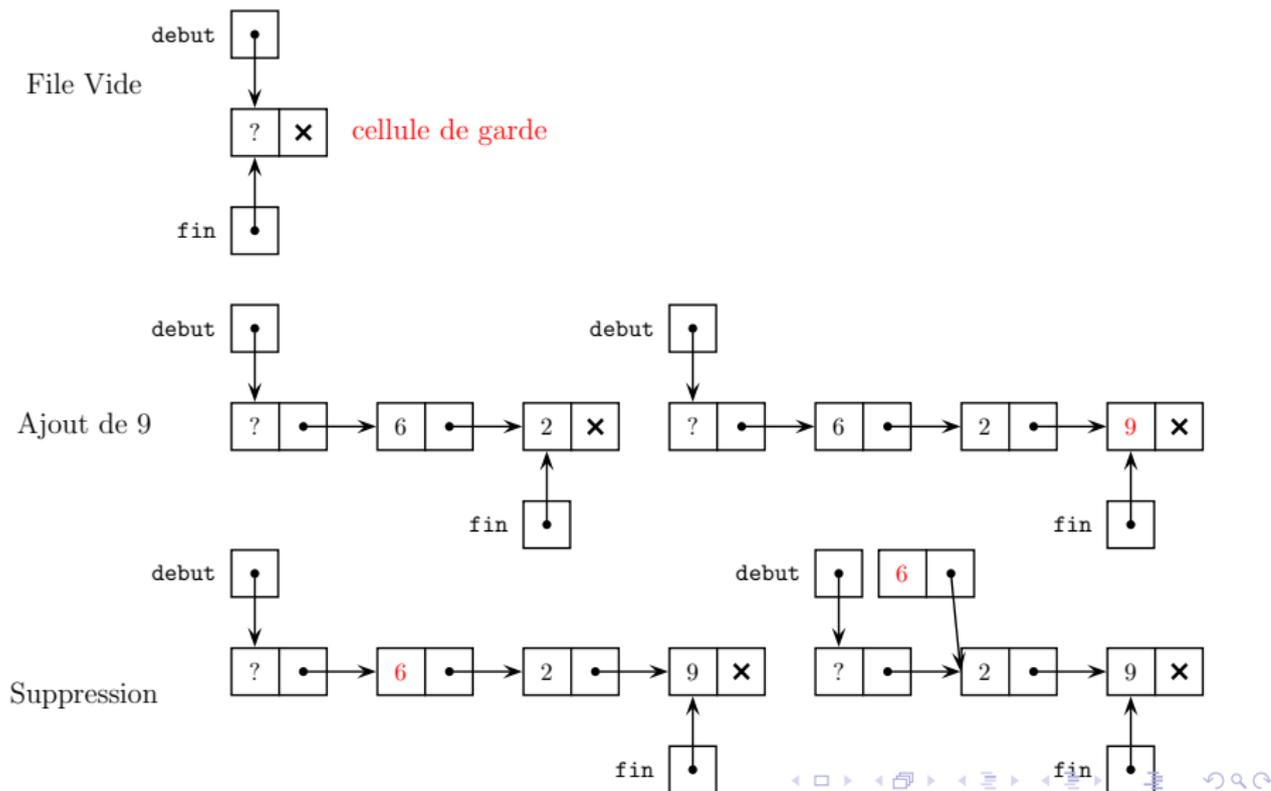
```
class File {
    static final int maxF = 10;
    int début, fin;
    int[] contenu;
    File() {
        début = 0;
        fin = 0;
        contenu = new int[maxF];
    }
    boolean estVide() {
        return début == fin;
    }
}
```

```
boolean estPleine() {
    return (fin + 1) % maxF == début;
}
void vider() {
    début = fin;
}
int valeur() {
    return contenu[début];
}
void supprimer() {
    début = (début + 1) % maxF;
}
void ajouter(int x) {
    contenu[fin] = x;
    fin = (fin + 1) % maxF;
}
}
```

Les files avec une liste chaînée gardée

- ▶ On conserve à la fois une référence sur le premier et sur le dernier élément de la liste
- ▶ **Les ajouts se font à la fin de la liste**
 - ▶ il faut donc modifier à chaque ajout la référence sur le dernier élément de la liste
- ▶ Les opérations de **suppression** concernent la **tête** de la liste
- ▶ Pour simplifier le codage, on utilise une cellule de garde
 - ▶ La dernière cellule de la liste existe toujours (→ ajout simplifié)

Les files avec une liste chaînée gardée



Les files avec une liste chaînée gardée

```
class File {
    Liste début;
    Liste fin;
    File () {
        Liste garde = new Liste();
        début = garde;
        fin = garde;
    }
    static boolean
    estVide (File f) {
        return f.début == f.fin;
    }
}
```

```
static int valeur (File f) {
    if (estVide (f))
        throw new Error("VIDE");
    Liste b = f.début.suivant;
    return b.contenu;
}
static void ajouter (int x, File f)
    Liste a = new Liste (x, null);
    f.fin.suivant = a;
    f.fin = a;
}
static void supprimer (File f) {
    if (estVide (f))
        throw new Error("VIDE");
    f.début = f.début.suivant;
}
}
```

Un exemple : Simuler une file d'attente

- ▶ Attente dans une file pour un service
- ▶ Phénomène stochastique :
 - ▶ Arrivée aléatoire des clients
 - ▶ Temps de service aléatoire
 - ▶ Patience limitée des clients (seuil de tolérance aléatoire)
 - ▶ ...

Les arrivées, les temps d'attente, etc., sont des variables aléatoires qui suivent des lois de probabilité données. \circ n cherche des caractéristiques du systèmes (temps d'attente moyen, ratio de clients exaspérés, etc).

- ▶ Étude théorique des files d'attente (RO stochastique, résultats analytiques)
- ▶ Simulation informatique (seule méthode possible pour des situations complexes)

Un exemple : Simuler une file d'attente

- ▶ Un unique guichet ouvert 8h00 ($8 * 3600 = 28800$ secondes) consécutives.
- ▶ Les clients arrivent et font la queue. La probabilité d'arrivée d'un nouveau client sur un intervalle $[t, t + 1)$ est p (elle ne dépend pas de ce qui s'est passé avant l'instant t et la probabilité d'arrivée de plusieurs clients dans $[t, t + 1)$ est "négligeable" → **Poisson**).
- ▶ Temps de service (loi uniforme sur $[30, 300)$).
- ▶ Chaque client est plus ou moins patient (le temps après lequel il part sans être servi → loi uniforme sur $[120, 1800)$).

OBJECTIF : Calculer le ratio de clients qui partent sans être servis en fonction de p .

Simuler une file d'attente : Le client & la file

```
class Client {
    int arrivée;
    int seuil;
    Client(int arrivée, int seuil) {
        this.arrivée = arrivée;
        this.seuil = seuil;
    }
}

class File {
    static final int maxF = 1000;
    int début, fin;
    Client[] contenu;
    File() {
        début = 0;
        fin = 0;
        contenu = new Client[maxF];
    } ...
}
```

Simuler une file d'attente : La simulation

Un algorithme naïf : simulation discrète (seconde par seconde)

- ▶ A chaque seconde t , faire un tirage aléatoire pour simuler l'arrivée d'un client
 - ▶ Si un client arrive, "l'enfiler" et tirer aléatoirement son seuil d'attente
- ▶ Conserver dans une variable `libre` la prochaine date à laquelle le guichet se libère
- ▶ Quand un client est disponible et que le guichet est libre,
 - ▶ "défiler" le client
 - ▶ Vérifier que le client n'est pas parti
 - ▶ tirer aléatoirement un temps de traitement,
 - ▶ mettre à jour `libre`

Simuler une file d'attente : La simulation

On dispose

- ▶ d'un générateur aléatoire
 - ▶ `RandGen.rnd()` renvoie un réel dans $[0, 1)$ tiré aléatoirement
 - ▶ `RandGen.rnd(a)` renvoie un entier dans $[0, a)$ tiré aléatoirement
 - ▶ `RandGen.rnd(a, b)` renvoie un entier dans $[a, b)$ tiré aléatoirement
- ▶ d'un module qui permet de lancer plusieurs simulations et de faire des moyennes
- ▶ de `gnuplot` qui permet d'afficher des tableaux de données

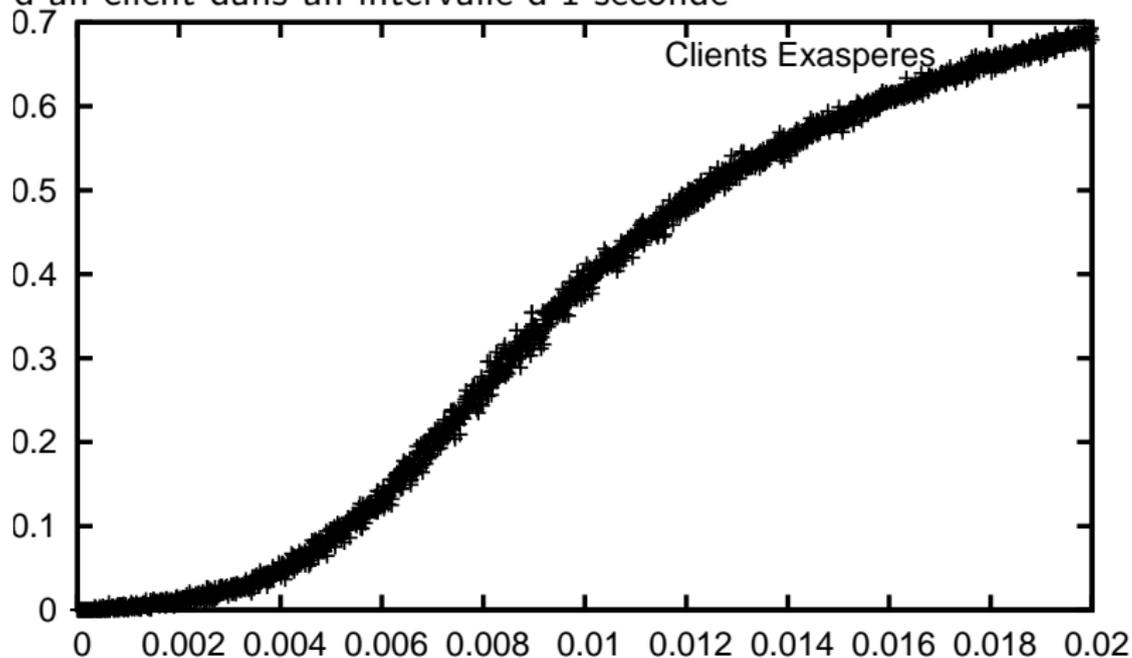
Simuler une file d'attente : La simulation

```
File f = new File (); int libre = 0;
int clientsArrivés = 0; int clientsExaspérés = 0;
for (int t = 0; t < tMax; t++) {
    if (RandGen.rnd() ≤ probArriveeT) {
        clientsArrivés ++;
        f.ajouter(new Client(t, RandGen.rnd(seuilMin,  seuilMax)));
    }
    if (libre ≤ t) {
        Client c = null;
        while (! f.estVide()) {
            c = f.défiler();
            if (t - c.arrivée ≤ c.seuil)
                break;
            clientsExaspérés ++;
            c = null;
        }
        if (c ≠ null) libre = t + RandGen.rnd(serviceMin, serviceMax); }}

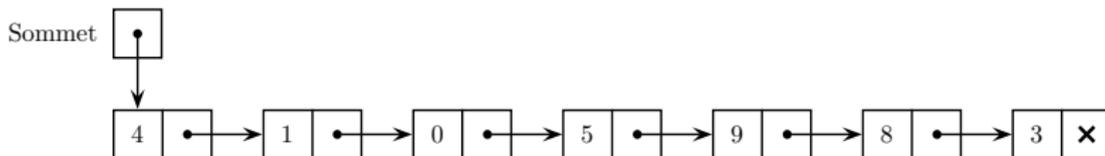
```

Simuler une file d'attente

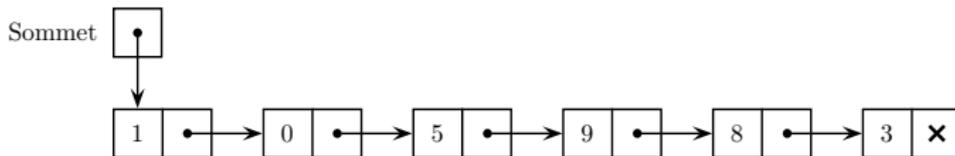
Le ratio de clients exaspérés en fonction de la probabilité d'arrivée d'un client dans un intervalle d'1 seconde



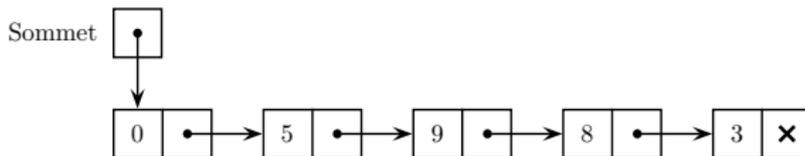
Comment coder une pile ? avec une liste



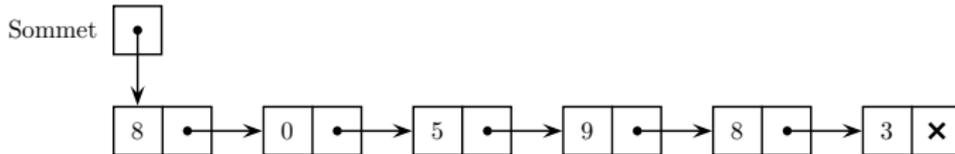
DÉPILER : 4



DÉPILER : 1



EMPIILER 8



Comment coder une pile ?

Très simplement avec une liste (ou un tableau).

```
class Liste {
    int valeur;
    Liste suivant;
    Liste(int v, Liste s) {
        valeur = v;
        suivant = s;
    }
}

class Pile {
    Liste l;
    public Pile() {
        l = null;
    }
}

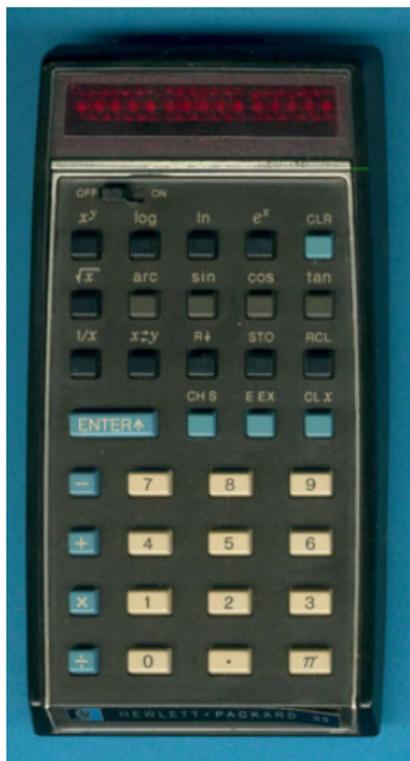
public static boolean estVide(Pile p) {
    return p.l == null;
}

void vider() {
    l = null;
}

static void ajouter(Pile p, int a) {
    p.l = new Liste(a, p.l);
}

static int retirer(Pile p) {
    int a = p.l.valeur;
    p.l = p.l.suivant;
    return a;
}
}}
```

Un investissement : la HP35 ou la fin de la règle à calcul



- ▶ janvier 1972 : la HP35, 1ere calculatrice de poche scientifique pour \$ 395.
- ▶ Très bon investissement (\$ 515 sur ebay 33 ans plus tard)
- ▶ RPN (Reverse Polish Notation)
 - ▶ développée en 1920 par Jan Lukasiewicz (formules sans parenthèses ni de crochets)
 - ▶ Exemple : $(3+5) / (7+6) = ?$.
 "Appuyez sur 3 puis sur la touche ENTER.
 Appuyez sur 5 puis sur la touche +. Appuyez sur 7, puis sur la touche ENTER. Appuyez sur 6 puis sur la touche +. Appuyez sur la touche de division et la calculatrice vous donne le résultat : 0,62."

La notation Polonaise inverse (RPN)

- ▶ Placer des nombres dans une pile
- ▶ effectuer des opérations sur les 2 nombres situés au sommet de la pile

Évaluation de "4 12 23 7 + + 3 * 121 - * "

LIT : 4	Pile : 4
LIT : 12	Pile : 12 4
LIT : 23	Pile : 23 12 4
LIT : 7	Pile : 7 23 12 4
LIT : +	Pile : 30 12 4
LIT : +	Pile : 42 4
LIT : 3	Pile : 3 42 4
LIT : *	Pile : 126 4
LIT : 121	Pile : 121 126 4
LIT : -	Pile : 5 4
LIT : *	Pile : 20

RPN

Algorithme : Utiliser une pile pour enregistrer les nombres et traiter les opérateurs “on the fly”.

Pour analyser la chaîne de caractère :

- ▶ La classe `String` permet les opérations usuelles sur les chaînes de caractères.
- ▶ Attention, les objets de la classe `String` sont **immuables** (variante destructrice des chaînes de caractères, la classe `StringBuffer`)
- ▶ Quelques méthodes utiles de la classe `String` : `int length()`, `char charAt(int index)`, `boolean equals(String s)`

RPN

Lire les caractères l'un après l'autre et appliquer les règles suivantes

- ▶ Si on rencontre un chiffre et que le caractère précédent en était déjà un, “poursuivre” la construction du nombre
- ▶ Si on rencontre un chiffre et que le caractère précédent n'en était pas un, “commencer” la construction du nombre
- ▶ Si on rencontre un caractère qui n'est pas un chiffre alors que le caractère précédent en était un, “empiler” le nombre construit
- ▶ Si on rencontre un caractère qui est un opérateur, dépiler deux fois, calculer et empiler

RPN

```
Pile p = new Pile(); int nbEnCours = 0; boolean lectNbEnCours = false;
for (int i = 0; i < args[0].length(); i++) {
    char c = args[0].charAt(i);
    if ((c ≤ '9') && (c ≥ '0')) {
        lectNbEnCours = true; nbEnCours = 10 * nbEnCours + c - 48;
    }
    else if (lectNbEnCours) {
        Pile.ajouter(p, nbEnCours);
        lectNbEnCours = false; nbEnCours = 0;
    }
    if (c == '+')
        Pile.ajouter(p, Pile.retirer(p) + Pile.retirer(p));
    if (c == '-')
        Pile.ajouter(p, - Pile.retirer(p) + Pile.retirer(p));
    if (c == '*')
        Pile.ajouter(p, Pile.retirer(p) * Pile.retirer(p));
}
```

TD 2 - Parcours de labyrinthe

