

INF 321
Piles, files et complexité algorithmique

Eric Goubault

Cours 6

7 juin 2011

DANS LE COURS PRÉCÉDENT

- Types disjonctifs
- Début des structures de données dynamiques (listes)

DANS CE COURS

- Quelques considérations de *complexité*
- Piles et files
- Le GC

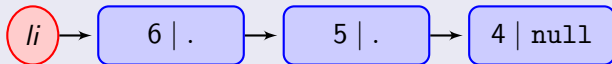
UN CAS D'ÉTUDE DANS LA COMPLEXITÉ...L'INVERSION DE LISTE

CE QUE L'ON VEUT FAIRE:

Partir d'une liste:



Et rendre la liste contenant les mêmes éléments, mais énumérée dans l'ordre inverse:



UNE IMPLÉMENTATION POSSIBLE

SI LA LISTE EST VIDE, RETOURNER VIDE:

```
static List reverse(final List x) {  
    if (x == null) return null;  
    ...  
}
```

UNE IMPLÉMENTATION POSSIBLE

SINON, INVERSER LA QUEUE DE `x` ET RAJOUTER `x.hd` EN FIN DE LISTE:

```
static List reverse(final List x) {  
    ...  
    return add(reverse(x.tl), x.hd);  
}
```

UNE IMPLÉMENTATION POSSIBLE DE ADD

SI x EST VIDE, RAJOUTER UNE NOUVELLE CELLULE
CONTENANT y:

```
static List add(final List x, final int y) {  
    if (x == null) return new List(y, null);  
    ...  
}
```

UNE IMPLÉMENTATION POSSIBLE

SINON, RAJOUTER Y EN FIN DE LA QUEUE DE LA LISTE X ET RAJOUTER x.hd EN TÊTE:

```
static List add(final List x, final int y) {  
    ...  
    return new List(x.hd, add(x.tl, y));  
}
```

NAIVEMENT...:

```
static List add(final List x, final int y) {  
    if (x == null) return new List(y, null);  
    return new List(x.hd, add(x.tl, y));  
}  
  
static List reverse(final List x) {  
    if (x == null) return null;  
    return add(reverse(x.tl), x.hd);  
}
```

EXEMPLE D'EXÉCUTION

ETAPE 1

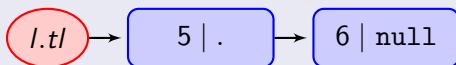
On appelle reverse de:



```
static List reverse(final List x) {  
    ...  
    return add(reverse(x.tl), x.hd);  
}
```

ETAPE 2

Qui appelle reverse de:



Et reprendra en faisant `add(.,4)`.

```
static List reverse(final List x) {  
    ...  
    return add(reverse(x.tl), x.hd);  
}
```

ETAPE 3

Qui appelle reverse de:



Et reprendra en faisant `add(.,5)`.

```
static List reverse(final List x) {  
    ...  
    return add(reverse(x.tl), x.hd);  
}
```

ETAPE 3

Qui appelle reverse sur:

rev(l.tl.tl.tl)

```
static List reverse(final List x) {  
    if (x == null) return null;  
    ...  
}
```

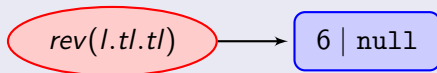
qui renvoie null

ETAPE 4

Qui appelle `add(null,6);`:

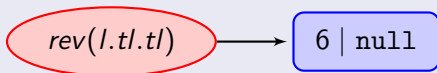
```
static List add(final List x, final int y) {  
    if (x == null) return new List(y, null);  
    ...  
}
```

et renvoie:



ETAPE 5

Puis `add(., 5)` sur le résultat précédent:



```
static List add(final List x, final int y) {  
    ...  
    return new List(x.hd, add(x.tl, y));  
}
```

ETAPE 6

Qui appelle `add(null,5)` et reconstruit une cellule contenant 6 en tête:



```
static List add(final List x, final int y) {  
    if (x == null) return new List(y, null);  
    ...  
}
```

ETAPE 7

En faisant `new List(6,.)` sur le résultat précédent (on obtient ainsi `rev(l.tl)`):



```
static List add(final List x, final int y) {  
    ...  
    return new List(x.hd, add(x.tl, y));  
}  
  
static List reverse(final List x) {  
    ...  
    return add(reverse(x.tl), x.hd);  
}
```

ETAPE 8

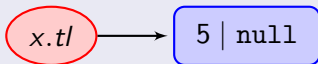
Puis `add(., 4)` sur le résultat précédent:



```
static List add(final List x, final int y) {  
    ...  
    return new List(x.hd, add(x.tl, y));  
}
```

ETAPE 9

Qui appelle `new List(6, add(x.tl, 4))` donc appelle `add(., 4)` sur:

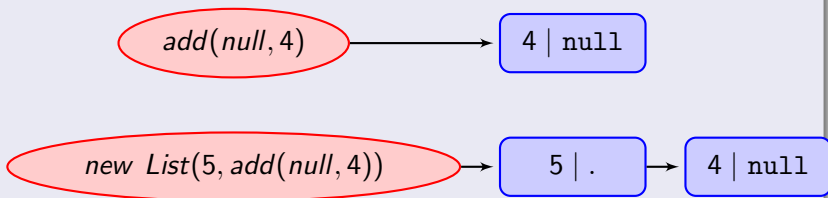


```
static List add(final List x, final int y) {  
    ...  
    return new List(x.hd, add(x.tl, y));  
}
```

EXEMPLE D'EXÉCUTION

ETAPES 10, 11

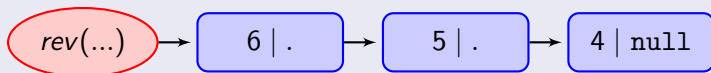
Qui appelle `new List(5, add(null, 4))` qui renvoie donc successivement:



```
static List add(final List x, final int y) {  
    if (x == null) return new List(y, null);  
    ...  
}
```

ET ENFIN

Qui appelle `new List(6, .)` sur la liste précédente:



```
static List add(final List x, final int y) {  
    ...  
    return new List(x.hd, add(x.tl, y));  
}  
  
static List reverse(final List x) {  
    ...  
    return add(reverse(x.tl), x.hd);  
}
```

Ouf!

COMPLEXITÉ?

On va compter le nombre d'opérations élémentaires effectuées par `add` (dans un premier temps), en fonction de la *taille* de ses arguments

- on suppose un coût unitaire pour chaque opération arithmétique, pour chaque test, pour chaque allocation mémoire etc. (pas tout à fait vrai sur une machine moderne, mais pas complètement faux quand on raisonne à *proportions* près)
- la taille d'une liste l (de scalaires) est comptée comme étant égale à $length(l)$.

PAR RÉCURRENCE:

`add` a un coût proportionnel à la longueur de x (complexité *linéaire*).



SUPPOSONS QUE LA TAILLE DE x SOIT DE n :

- Le premier appel (dans `reverse`) de `add` se fait sur une taille $n - 1$: coût proportionnel à $n - 1$
- Le deuxième appel de `add`, se fait sur une taille $n - 2$: coût proportionnel à $n - 2$
- etc.
- Le dernier appel de `add` se fait sur une taille 1: coût proportionnel à 1

DONC COÛT TOTAL PROPORTIONNEL À $1 + 2 + \dots + n - 1 \sim n^2$

QUE VEUT-ON DIRE PAR “EST DE L’ORDRE DE” (\sim)?

NOTATIONS DE LANDAU ET DE HARDY

Comme en analyse (ici, pour f, g deux fonctions de \mathbb{N} vers \mathbb{N}):

- $f = O(g)$ ssi $\exists \alpha, N, \forall n \in \mathbb{N}, n \geq N \implies f(n) \leq \alpha g(n)$
- $f = o(g)$ ssi $\forall \epsilon, \exists N, \forall n \in \mathbb{N}, n \geq N \implies f(n) \leq \epsilon g(n)$
- $f \sim g$ ssi $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$

EN FAIT, PLUS UTILE POUR NOUS:

L’ordre de grandeur...:

- $f = \Omega(g)$ ssi $\exists \alpha, N, \forall n \in \mathbb{N}, n \geq N \implies f(n) \geq \alpha g(n)$
- $f = \Theta(g)$ ssi $\exists \alpha, \beta, N, \forall n \in \mathbb{N}, n \geq N \implies \beta g(n) \leq f(n) \leq \alpha g(n)$

ICI POUR REVERSE NAIF

Complexité en $\Theta(n^2)$



EN MOYENNE, DANS LE CAS LE PIRE...

Etant donné un problème de *taille* n , la complexité d'un algorithme est $f(n)$, le nombre d'opérations *élémentaires* impliquées dans son calcul:

- Complexité en moyenne: moyenne arithmétique des nombres d'opérations pour une taille n
- Complexité dans le cas le pire: max des nombres d'opérations pour une taille n

(sur une machine *séquentielle*)

EXEMPLE

- La complexité (dans le cas le pire, et en moyenne) de reverse (naif) est en $\Theta(n^2)$
- La complexité d'un problème est la meilleure complexité d'un algorithme résolvant ce problème:
- En fait, la complexité de l'inversion de liste est de $\Theta(n)$, cf. algorithme suivant...

ON UTILISE UNE LISTE INTERMÉDIAIRE (ACCUMULATEUR)

Si x est vide, on retourne l'accumulateur:

```
static List revappend(final List x, final List y) {  
    if (x == null) return y;  
    ...  
}
```

INVERSION NON NAIVE DE LISTE

SINON ON MET `x.hd` EN TÊTE DE L'ACCUMULATEUR ET ON INVERSE LA QUEUE DE `x` AVEC CE NOUVEL ACCUMULATEUR:

```
static List revappend(final List x, final List y) {  
    ...  
    return revappend(x.tl, new List(x.hd, y)); }  
}
```

FONCTION FINALE

On appelle cette inversion avec accumulateur, avec un accumulateur vide au début:

```
static List reverse(final List x) {  
    return revappend(x, null); }  

```

CODE FINAL:

```
static List revappend(final List x, final List y) {  
    if (x == null) return y;  
    return revappend(x.tl, new List(x.hd, y)); }  
  
static List reverse(final List x) {  
    return revappend(x, null); }
```

REMARQUE: VERSION ITÉRATIVE:

CODE FINAL:

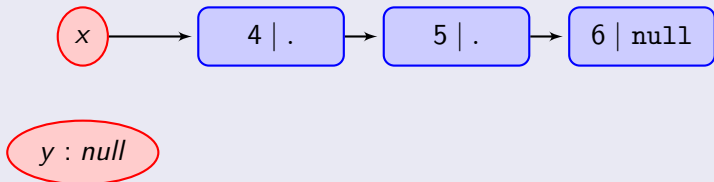
On parcourt la liste dans un sens, pour reconstruire une autre dans l'autre!

```
static List reverse(List x) {  
    List res = null;  
    while (x != null) {  
        res = new List(x.hd, res);  
    }  
    return res;  
}
```

EXEMPLE D'EXÉCUTION

ETAPE 1

On appelle `reverse` (donc `revappend(x,y)`) avec:

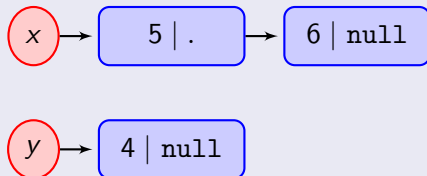


```
static List reverse(final List x) {  
    return revappend(x, null);  
}
```

EXEMPLE D'EXÉCUTION

ETAPE 2

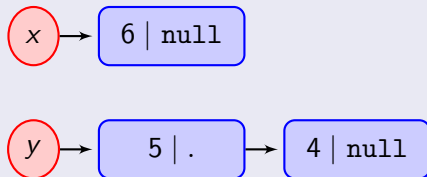
Qui appelle `revappend(x.tl, new List(4,null))`; donc les nouveaux arguments `x` et `y` sont:



```
static List revappend(final List x, final List y) {  
    ...  
    return revappend(x.tl, new List(x.hd, y)); }  
}
```

ETAPE 3

Qui appelle `revappend(x.tl, new List(5,y))`; donc les nouveaux arguments `x` et `y` sont:



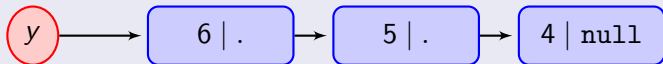
```
static List revappend(final List x, final List y) {  
    ...  
    return revappend(x.tl, new List(x.hd, y)); }  
}
```

EXEMPLE D'EXÉCUTION

ETAPE FINALE

Qui appelle `revappend(null, new List(6,y))`; donc les nouveaux arguments `x` et `y` sont:

x : null



```
static List revappend(final List x, final List y) {  
    if (x == null) return y;  
    ... }  
}
```

Bien plus court!

- Etant donné une liste de taille n
- On fait un appel à `revappend` sur une liste de taille n plus une opération élémentaire (création d'une cellule de liste)
- qui fait un appel à `revappend` sur une liste de taille $n - 1$ plus...
- ...
- qui fait `return y`

$\Theta(n)$ opérations! Et on ne peut faire mieux sur une machine séquentielle...

COMPLEXITÉ *en temps*

Dans l'ordre (strictement) croissant, les plus classiques:

Temps	Classe	Exemple
$\Theta(1)$	temps constant	addition d'entiers (machine)
$\Theta(\log(n))$	temps logarithmique	
$\Theta(n)$	temps linéaire	inversion de liste
$\Theta(n \log(n))$	temps quasi-linéaire	tri (cf. cours 10)
$\Theta(n^2)$	temps quadratique	produit matrice vecteur
$\Theta(n^p)$	temps polynomial	multiplication de matrices
$\Theta(e^n)$	temps exponentiel	plus tard...

CLASSES DE COMPLEXITÉ

ON NOTE CLASSIQUEMENT:

- L pour le temps logarithmique (*fonctions très peu coûteuses*)
- P (ou PTIME) pour le temps polynomial (*pas trop coûteuses*)
- EXPTIME pour le temps exponentiel (*très coûteuses*)

EXEMPLE D'ORDRES DE GRANDEUR (EN SECONDES)

Classe/ n	1	2	5	10	...	20	50
L	0...	0.3	0.7	1		1.3	1.7
$\Theta(n)$	1	2	5	10		20	50
$\Theta(n \log(n))$	0...	0.6	3.5	10		26	85
$\Theta(n^2)$	1	4	25	100		400	2500
$\Theta(n^3)$	1	8	125	1000		8000	125000
$\Theta(e^n)$	2.7	7.4	148	22026		$4.9 \cdot 10^8$	$5.2 \cdot 10^{21}$

Ce dernier nombre fait environ $1.6 \cdot 10^{14}$ années alors que l'âge de l'univers estimé est d'environ $15 \cdot 10^9$ années;

Notre système solaire aura disparu depuis bien longtemps...

POUR ALLER PLUS LOIN: CLASSE NP; NP-COMPLÉTUDE

SANS RENTRER DANS LES DÉTAILS...

- Problèmes les “plus durs” dont on peut vérifier la solution en temps polynomial
- Hélas beaucoup de problèmes classiques sont NP-complet:
 - exemple SAT, déterminer si une formule logique est toujours vraie ou pas
 - aussi compliqué qu'énumérer les valeurs booléennes pour toutes les variables (exponentiel) et tester si la formule est vraie avec ces affectations (polynomial)
 - ex.: $a \vee \neg a$, essayer $a = true$ puis $a = false$...
- On ne connaît à l'heure actuelle que des algorithmes EXPTIME au mieux pour résoudre les problèmes NP-complets
- On ne sait pas à l'heure actuelle si $P \neq NP$ (mais bien sûr $P \subseteq NP$); problème à **1 million de dollars!**



- On compte non pas le temps mais le nombre d'octets nécessaire à résoudre un problème algorithmique...

RÉSUMÉ...

$$L \subseteq P \subseteq PSPACE \subseteq EXPTIME$$

Remarquez le rapport complexité en temps et en espace...

Pour aller plus loin...INF 423!

PILES

Listes, avec 2 opérations principales:

- push pour ajouter un nouvel élément en tête
- pop pour retirer l'élément en tête

(plus création de pile vide, test vide, et récupérer valeur en tête)

FILES

Listes, avec 2 opérations principales:

- add pour ajouter un nouvel élément en queue
- pop pour retirer l'élément en tête

(plus création de file vide, test vide, et récupérer valeur en tête)

LISTES?

Les deux ont comme ensemble sous-jacent les listes, mais les opérations que l'on fait sont différentes (ex. en mathématiques, le corps $(\mathbb{R}, +, \times)$ et le semi-anneau idempotent $(\mathbb{R}, \max, +)$)

```
class Pile {  
    List c;  
    Pile(List x) { this.c = x; } }  
  
class Prog {  
    static Pile empty() {  
        return new Pile(null); }  
  
    static boolean testempty(Pile l) {  
        return l.c == null; }  
}
```

IMPLÉMENTATION DES PILES (2)

```
static void push(int a, Pile l) {  
    l.c = new List(a, l.c); }  
  
static void pop(Pile l) {  
    l.c = l.c.tl; }  
  
static int top(Pile l) {  
    return l.c.hd; }  
}
```

EXEMPLE ET REPRÉSENTATION D'UNE PILE

TEST SIMPLE:

```
Pile p = empty();  
push(2, p);  
push(3, p);  
System.out.println(top(p));  
pop();  
System.out.println(top(p));  
pop();  
System.out.println(testempty(p));
```

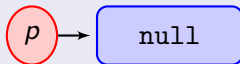
Donne:

```
3  
2  
true
```

EXEMPLE ET REPRÉSENTATION D'UNE PILE

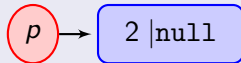
EXÉCUTION:

```
Pile p = empty();
```



EXEMPLE ET REPRÉSENTATION D'UNE PILE

EXÉCUTION:



```
push(2, p);
```

Que l'on représente aussi

par: $\boxed{\rightarrow 2}$

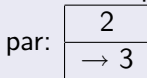
EXEMPLE ET REPRÉSENTATION D'UNE PILE

EXÉCUTION:



```
push(3, p);
```

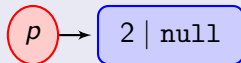
Que l'on représente aussi



EXEMPLE ET REPRÉSENTATION D'UNE PILE

EXÉCUTION:

```
System.out.println(top(p));  
pop();
```

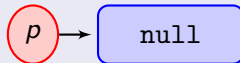


Que l'on représente aussi
par: $\boxed{\rightarrow 2}$

EXEMPLE ET REPRÉSENTATION D'UNE PILE

EXÉCUTION:

```
System.out.println(top(p));  
pop();  
System.out.println(testempty(p));
```



Que l'on représente aussi
par:

RET==0: CAS DES APPELS *descendants*

```

class Factpile {
  static int factpile(Pile p) {
    int current, temp, res, ret;
    ret = 0; res = 1;
    while (!Prog.testempty(p)) {
      current = Prog.top(p);
      Prog.pop(p);
      switch (ret) {
        case 0:
          System.out.println("Appel_avec_x:_"+current);
          if (current == 0) { ret = 1;
            Prog.push(1,p); }
          else { Prog.push(current,p);
            Prog.push(current-1,p);
          } break;
      }
    }
  }
}

```



REMONTÉE DANS LES APPELS RÉCURSIFS

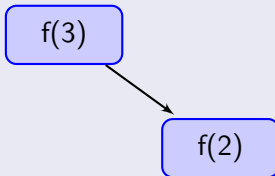
```
default:  
System.out.println("Retour_:_"+current);  
if (Prog.testempty(p)) res=current;  
else { temp = Prog.top(p);  
    System.out.println("Valeur_de_x:_"+temp);  
    Prog.pop(p);  
    Prog.push(current*temp,p); } } }  
return res; }
```

PROGRAMME PRINCIPAL

```
static public void main(String [] args) {  
    Pile p = Prog.empty();  
    Prog.push(Integer.parseInt(args[0]),p);  
    System.out.println(factpile(p)); } }
```

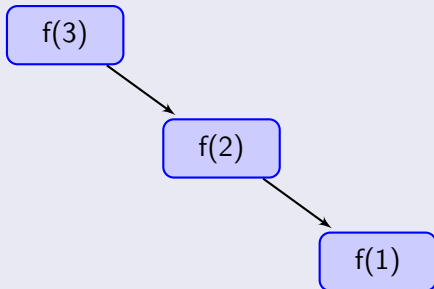
EXÉCUTION (1)

```
> java Factpile 3  
Appel avec x: 3  
Appel avec x: 2
```



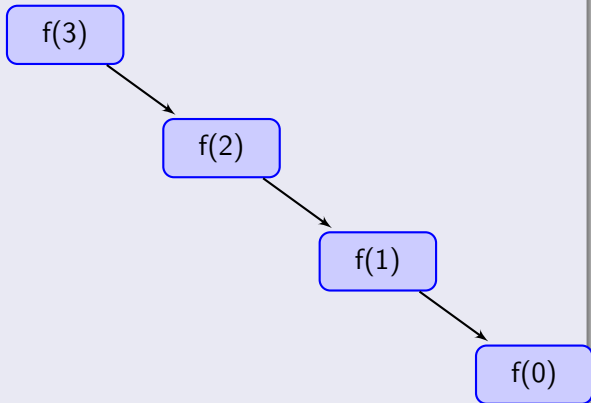
EXÉCUTION (2)

```
> java Factpile 3  
Appel avec x: 3  
Appel avec x: 2  
Appel avec x: 1
```



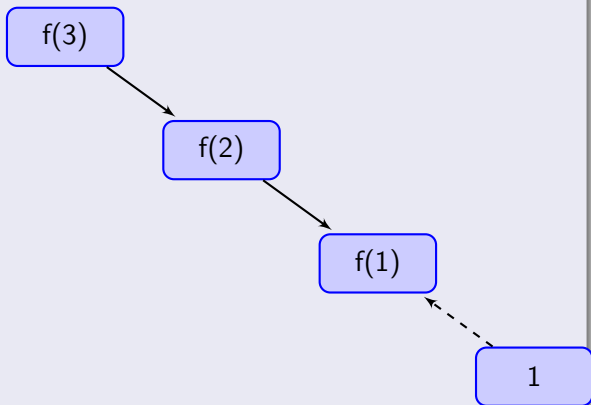
EXÉCUTION (3)

```
> java Factpile 3  
Appel avec x: 3  
Appel avec x: 2  
Appel avec x: 1  
Appel avec x: 0
```



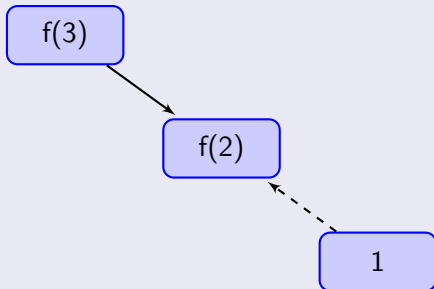
EXÉCUTION (4)

```
> java Factpile 3  
Appel avec x: 3  
Appel avec x: 2  
Appel avec x: 1  
Appel avec x: 0  
Retour dernier : 1  
Valeur locale de x: 1
```



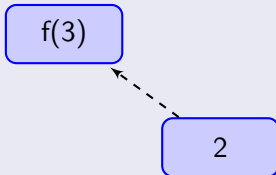
EXÉCUTION (5)

```
> java Factpile 3
Appel avec x: 3
Appel avec x: 2
Appel avec x: 1
Appel avec x: 0
Retour dernier: 1
Valeur locale de x: 1
Retour dernier: 1
Valeur locale de x: 2
```



EXÉCUTION (6)

```
> java Factpile 3
Appel avec x: 3
Appel avec x: 2
Appel avec x: 1
Appel avec x: 0
Retour dernier: 1
Valeur locale de x: 1
Retour dernier: 1
Valeur locale de x: 2
Retour dernier: 2
Valeur locale de x: 3
Retour dernier: 6
6
```



```
class File {  
    List c;  
    File(List x) { this.c = x; } }  
  
class Prog {  
    static File empty() {  
        return new File(null); }  
  
    static boolean testempty(File l) {  
        return l.c == null; }  
}
```

IMPLÉMENTATION DES FILES (2)

```
static void add(int a, File l) {  
    if (l == null) l.c = new List(a, null);  
    else l.c = new List(l.c.hd,  
                        add(a, new File(l.c.tl))); }  
}
```

```
static void pop(File l) {  
    l.c = l.c.tl; }  
}
```

```
static int top(File l) {  
    return l.c.hd; }  
}
```

TEST SIMPLE:

```
File f = empty();  
add(2, f);  
add(3, f);  
System.out.println(top(f));  
pop();  
System.out.println(top(f));  
pop();  
System.out.println(testempty(f));
```

Donne:

```
2  
3  
true
```

EXEMPLE ET REPRÉSENTATION D'UNE FILE

EXÉCUTION:

```
File f = empty();
```



`null`

EXEMPLE ET REPRÉSENTATION D'UNE FILE

EXÉCUTION:



```
add(2, f);
```

Que l'on représente aussi
par: $\boxed{\rightarrow 2}$

EXEMPLE ET REPRÉSENTATION D'UNE FILE

EXÉCUTION:



```
add(3, f);
```

Que l'on représente aussi

par:

→ 2	3
-----	---

EXEMPLE ET REPRÉSENTATION D'UNE FILE

EXÉCUTION:

```
System.out.println(top(f));  
pop();
```

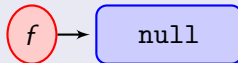


Que l'on représente aussi
par: $\rightarrow 3$

EXEMPLE ET REPRÉSENTATION D'UNE FILE

EXÉCUTION:

```
System.out.println(top(f));  
pop();  
System.out.println(testempty(f));
```



Que l'on représente aussi
par:

FILES SIMPLES

- Permettent de modéliser des files d'attentes “physiques”
- Ou de traiter des requêtes/événements dans un programme, “équitablement”, sur le principe *premier arrivé, premier servi*

FILES DE PRIORITÉ

- Généralisation où chaque entrée a une *priorité*.
- C'est la priorité la plus forte qui est servie en premier, et à priorité égale, c'est premier arrivé, premier servi
- Permet par exemple de coder des ordonnanceurs de tâches parallèles...

GARBAGE COLLECTOR/GLANEUR DE CELLULES

- Inventé par John MacCarthy pour le LISP (prix Turing 1971)
- Permet de récupérer la mémoire non utilisée, c'est un processus qui s'exécute en parallèle et automatiquement:
 - détermine quels objets ne peuvent plus être utilisés par un programme
 - récupère cet espace mémoire (pour être utilisé lors d'allocations futures)
- Nombreux algorithmes...implémentés par exemple dans Java, Caml, mais pas dans C

ALGORITHMES “MARK AND SWEEP”

- Le GC commence à parcourir les locations mémoires *vivantes* (accessibles depuis les *racines*, i.e. les noms de variables du programme Java) - l'exécution du programme Java est suspendue; 2 phases alors:
 - (*mark*): Les objets alloués et visitables par le GC depuis les racines sont *taggués*: visité et pas visité
 - (*sweep*): Le GC parcourt adresse par adresse le *tas* (l'endroit en mémoire où sont alloués les objets) et “efface” les objets non taggués “visité”

COMPTEURS DE RÉFÉRENCES

- Le GC maintient avec chaque objet, un nombre de références pointant sur chaque objet
- Si ce compteur arrive à zéro, l'objet est libéré...

ÉGALEMENT...

- "Stop and copy"
- GC conservatifs, incrémentaux, générationnels (cas de Java) etc.

ALLOCATION/DÉALLOCATION MANUELLE

Exemple (listes):

```
List cons(int car, List cdr) {  
    /* allocation */  
    List res = (List) malloc(sizeof(struct stList));  
    res->hd = car; res->tl = cdr; return res; }  
  
void freelist(List l) {  
    if (l == null) return;  
    freelist(l->tl);  
    /* deallocation */  
    free(l); }
```

CONSÉQUENCES

- C'est au programmeur de prévoir quand utiliser free, freelist etc.
- Parfois très compliqué quand on fait du partage!

REGARDEZ SUR LA PAGE WEB:

<http://www.enseignement.polytechnique.fr/informatique/INF321/> L'année dernière:

...

Une solution simple serait d'utiliser des tableaux de

Neanmoins, cette représentation est peu compacte
une valeur entière (l'indice d'un cycle) ;

une valeur booléenne (la valeur du signal pendant ce

...

CORRIGÉ...(1)

```
class Signal {
    int t;
    boolean b;
    Signal next;
    Signal( int t0, boolean b0, Signal next0 ){
        t = t0;
        b = b0;
        next = next0;
    }
    // Q4
    Signal( boolean b0 ){
        b = b0; // default init: t=0; next=null
    }
}
```

CORRIGÉ...(2)

```
class Simu {
    // Q3
    static void ppb( boolean b ){
        System.out.print( b ? "1" : "0" );
    }
    static void displayDebug( Signal f ){
        while( f != null ){
            System.out.println( f.t + "|=>" + f.b );
            f = f.next;
        }
        System.out.println();
    }

    // Q5
    static void display( int n, Signal f ){
        Signal ff = f;
        boolean bb = f.b;
        for( int i = 0; i <= n; i ++ ){
            if( ff != null && i == ff.t ){
                bb = ff.b;
                ff = ff.next;
            }
            ppb( bb );
        }
        System.out.println();
    }
}
```

C'EST TOUT POUR AUJOURD'HUI..

LA PROCHAINE FOIS

- Classes et objets
- Gestion des erreurs et exceptions

BON TD!

C'EST TOUT POUR AUJOURD'HUI..

LA PROCHAINE FOIS

- Classes et objets
- Gestion des erreurs et exceptions

BON TD!