

Cours 3: Listes. Piles. Files. Tris. Complexité

Olivier Bournez

bournez@lix.polytechnique.fr
LIX, Ecole Polytechnique

INF421-a

Bases de la programmation et de l'algorithmique

1

Aujourd'hui

Programmer avec des listes

Complexité

Trier

Piles. Files.

Utilisation des piles et des files

Programmer des piles et des files

Piles et files dans la bibliothèque JAVA

2

Rappel: Liste

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

```
Liste lst = new Liste(4, new Liste (7, new Liste (3, null)));
```



3

Calculer le maximum

■ Le problème MAX.

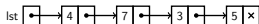
- ▶ On se donne une liste d'entiers naturels e_1, e_2, \dots, e_n .
- ▶ On veut calculer $\max\{e_1, e_2, \dots, e_n\}$.

-1 si la liste est vide.

4

Calculer le maximum (itératif)

```
static int max(Liste a) {
    if (a == null) return -1;
    int max = a.contenu;
    for (a = a.suivant; a != null; a = a.suivant) {
        if (a.contenu > max)
            max = a.contenu;
    }
    return max;
}
```



Nombre de comparaisons $C(n)$: $C(n) = n - 1$.

Nombre d'affectations $B(n)$: $n + 1 \leq B(n) \leq 2n$.

n = longueur de la liste.

comparaison = comparaison entre entiers.

Calculer le maximum (récursif)

- Une liste d'entiers est solution de l'équation

$$L = \text{null} \cup (\text{int} \times L)$$

- Equations récursives, en notant \emptyset la liste vide, et en notant (x, L) une liste non-vide.

$$\text{max}_{\text{liste}}(\emptyset) = -1$$

$$\text{max}_{\text{liste}}((x, L)) = \max(x, \text{max}_{\text{liste}}(L))$$

Calculer le max (récursif)

```
static int max(Liste a) {
    if (a == null) return -1;
    return max(a.contenu, max(a.suivant));
}

static int max(int a, int b) {
    if (a > b) return a;
    return b;
}
```

Remarque de JAVA: on utilise une surcharge de méthode.

Nombre de comparaisons $C(n)$: $C(n) = n$.

Solution de $C(0) = 0$,

$C(n) = 1 + C(n-1)$, pour $n > 0$.

Calculer le max (récursif inefficace)

Equations récursives:

$$\text{max}_{\text{liste}}(\emptyset) = -1$$

$$\text{max}_{\text{liste}}((x, L)) = \begin{cases} x & \text{si } x > \text{max}_{\text{liste}}(L) \\ \text{max}_{\text{liste}}(L) & \text{sinon.} \end{cases}$$

```
static int max(Liste a) {
    if (a == null) return -1;
    if (a.contenu > max(a.suivant))
        return a.contenu;
    else
        return max(a.suivant);
}
```

Nombre de comparaisons: $n \leq C(n) \leq 2^n - 1$.

Meilleur cas $C(n) = 1 + C(n-1)$, pour $n > 0$, $C(0) = 0$

Pire cas $C(n) = 1 + 2 * C(n-1)$, pour $n > 0$, $C(0) = 0$.

Quel est le meilleur programme?

- Styles
 - ▶ Récurusif / Itératif
Souvent:
 - Itératif plus efficace (en Java).
 - Récurusif plus simple et élégant. **Récurusif plus simple et élégant.**
 - ▶ Persistant / Non persistant
Souvent:
 - Persistant moins problématique.
- Structures de données:
 - ▶ Tableaux / Listes / Piles, Files, Arbres, Graphes, ...
Souvent:
 - Fonction du contexte

9

Aujourd'hui

Programmer avec des listes

Complexité

Trier

Piles, Files.

Utilisation des piles et des files

Programmer des piles et des files

Piles et files dans la bibliothèque JAVA

10

Complexité d'un problème, d'un algorithme

- On fixe une mesure élémentaire:
 - ▶ nombre de comparaisons,
 - ▶ nombre d'affectations,
 - ▶ mémoire utilisée,
 - ▶ temps utilisé,
 - ▶ ...
- Cette mesure associe à un algorithme \mathcal{A} et une entrée d une valeur
$$\text{Complexité}(\mathcal{A}, d).$$
- On cherche souvent à évaluer cette complexité en fonction d'un paramètre naturel $n = \text{taille}(d)$, représentatif des entrées.
- Exemple:
 - ▶ \mathcal{A} : l'algorithme itératif pour MAX précédent.
 - ▶ d : une liste donnée en entrée.
 - ▶ $n = \text{taille}(d)$, le nombre d'éléments dans la liste.
 - ▶ $\text{Complexité}(\mathcal{A}, d)$, le nombre de comparaisons de \mathcal{A} sur d .

11

- On peut alors parler de la complexité (au pire cas)
 - ▶ d'un algorithme (on fait varier seulement les entrées)

$$\text{Complexité}_{\mathcal{A}}(n) = \max_{d/\text{taille}(d)=n} \text{Complexité}(\mathcal{A}, d).$$

- ▶ d'un problème (on fait varier l'algorithme, et les entrées)

$$\text{Complexité}(n) = \inf_{\mathcal{A} \text{ correct}} \max_{d/\text{taille}(d)=n} \text{Complexité}(\mathcal{A}, d)$$

- On arrive parfois à parler de la complexité exacte.
- On doit souvent se limiter à une étude asymptotique, ou à des bornes asymptotiques.

12

Exemple d'étude exacte

- Etude du problème *MAX* en nombre de comparaisons:
 - ▶ Le problème *MAX* admet une solution avec $n - 1$ comparaisons.

```
static int max(Liste a) {
    if (a== null) return -1;
    int max = a.contenu;
    for (a = a.suivant; a ≠ null; a = a.suivant) {
        if (a.contenu > max)
            max = a.contenu;}
    return max;}
```

- ▶ Cet algorithme est optimal en nombre de comparaisons.

13

Preuve

- Proposition: dans tout algorithme, tout élément autre que le maximum doit être comparé au moins une fois avec un élément qui lui est plus grand.
- Preuve:
 - ▶ soit i_0 le rang du maximum M retourné par l'algorithme sur une liste $L = e_1.e_2.\dots.e_n$.
 - ▶ Par l'absurde: soit $j_0 \neq i_0$ tel que e_{j_0} n'est pas comparé avec un élément plus grand que lui.
 - ▶ e_{j_0} n'a donc pas été comparé avec e_{i_0} le maximum
 - ▶ Considérer la liste $L' = e_1.e_2.\dots.e_{j_0-1}.M + 1.e_{j_0+1}.\dots.e_n$ obtenue à partir de L en remplaçant l'élément d'indice j_0 par $M+1$.
 - ▶ L'algorithme effectue exactement les mêmes comparaisons sur L et L' , sans comparer $L'[j_0]$ avec $L'[i_0]$ et donc retournera $L'[i_0]$, ce qui est incorrect.
- Conséquence: il faut au moins $n - 1$ comparaisons pour résoudre *MAX*.

14

Asymptotiques

- Pour un processeur capable d'effectuer un million d'instructions élémentaires par seconde.

n	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	4 s
$n = 30$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	18 m	10^{25} a
$n = 50$	< 1 s	< 1 s	< 1 s	< 1 s	11 m	36 a	∞
$n = 10^2$	< 1 s	< 1 s	< 1 s	1 s	12.9 a	10^{17} a	∞
$n = 10^3$	< 1 s	< 1 s	1 s	18 m	∞	∞	∞
$n = 10^4$	< 1 s	< 1 s	2 m	12 h	∞	∞	∞
$n = 10^5$	< 1 s	2 s	3 h	32 a	∞	∞	∞
$n = 10^6$	1 s	20 s	12 j	31710 a	∞	∞	∞

- Notations: $\infty =$ le temps dépasse 10^{25} années, s= seconde, m= minute, h = heure, a = an.

15

Notation de Landau (Pire Cas)

- Notation de Landau:
 - ▶ $f(n) = O(g(n))$ si et seulement si il existe deux constantes positives n_0 et B telles que

$$\forall n \geq n_0, f(n) \leq Bg(n)$$

Ce qui signifie que f ne croît pas plus vite que g .

- Exemple:
 - ▶ Le problème *MAX* admet une solution itérative en $O(n)$ affectations.
 - ▶ L'algorithme itératif précédent fonctionne en $O(n)$ affectations.
- Un algorithme
 - ▶ en temps $O(1)$ effectue un nombre constant d'opérations.
 - ▶ en temps $O(n)$ s'appelle un algorithme linéaire.
 - ▶ en temps $O(n^k)$ s'appelle un algorithme polynomial.

16

Notation de Landau

■ Notation de Landau (suite)

- ▶ $f(n) = \Omega(g(n))$ si et seulement si il existe deux constantes positives n_0 et B telles que

$$\forall n \geq n_0, f(n) \geq Bg(n)$$

- ▶ $f(n) = \Theta(g(n))$ si et seulement si il existe trois constantes positives n_0 , B et C telles que

$$\forall n \geq n_0, Bg(n) \leq f(n) \leq Cg(n)$$

■ Exemple:

- ▶ Le problème *MAX* nécessite $\Theta(n)$ comparaisons.
- ▶ (à venir) Trier nécessite $\Omega(n \log n)$ comparaisons.

17

Complexité d'un algorithme en moyenne

- Si on fixe une distribution π sur les entrées d , il est parfois possible d'évaluer la complexité en moyenne d'un algorithme:

$$\text{Complexité-Moyenne}_{\mathcal{A}}(n) = \sum_{d/\text{taille}(d)=n} \pi(d) \text{Complexité}(\mathcal{A}, d)$$

- Exemple: pour l'algorithme \mathcal{A} suivant.

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

Nombre de comparaisons (entre entiers): k , où k est le rang de l'élément lorsqu'il est dans la liste, n sinon.

18

- Si on suppose que les entrées sont les listes permutations de $\{1, 2, \dots, n\}$, et qu'elles sont équiprobables,

$$\begin{aligned} \text{Complexité-Moyenne}_{\mathcal{A}}(n) &= \text{Esperance}_{d/\text{taille}(d)=n}[k] \\ &= \sum_{i=1}^n i \times P(k=i) \end{aligned}$$

Puisque les permutations sont équiprobables,
 $\text{Proba}(k=i) = 1/n$.

$$\begin{aligned} \text{Complexité-Moyenne}_{\mathcal{A}}(n) &= \sum_{i=1}^n i \frac{1}{n} \\ &= \frac{n(n+1)}{2n} \\ &= \frac{n+1}{2} \end{aligned}$$

- (poly) D'autres exemples plus compliqués.
- (joli exercice) Pour \mathcal{A} algorithme itératif précédent pour *MAX*, $\text{Complexité-Moyenne}_{\mathcal{A}}(n)$ en affectations entre variables entières est en $\Theta(\log n)$ (de l'ordre de $H_n = 1 + \frac{1}{2} + \dots + \frac{1}{n}$).

19

Aujourd'hui

Programmer avec des listes

Complexité

Trier

Piles. Files.

Utilisation des piles et des files

Programmer des piles et des files

Piles et files dans la bibliothèque JAVA

20

Pourquoi trier?

- Trier est une opération naturelle.
- Travailler sur des données triées est parfois plus efficace.
 - ▶ Exemple:
 - Rechercher un élément dans un tableau à n éléments: $O(n)$ (utiliser le principe de la fonction *Dans* précédente sur les listes).
 - Rechercher un élément dans un tableau trié à n éléments: $O(\log n)$ (par dichotomie).
 - Trier devient intéressant dès que le nombre de recherches est en $\Omega(\text{Complexité(tri)}/n)$.

21

Recherche par dichotomie

```
static boolean trouve(int[] T, int v, int min, int max){
    if(min >= max) // vide
        return false;
    int mid = (min + max) / 2;
    if (T[mid] == v) return true;
    else if (T[mid] > v) return trouve(T, v, min, mid);
    else return trouve(T, v, mid + 1, max);
}
```

Nombre de comparaisons

- $C(1) = 2$,
- $C(n) \leq 2 + C(n/2)$, pour n pair,
- $C(n) \leq 2 \log n + 2$, pour n puissance de 2.

$$C(n) = O(\log n),$$

dans le cas général.

22

Trier par insertion

- Méthode souvent utilisée pour trier un jeu de cartes.
- Etape préliminaire: savoir insérer un élément x dans une liste triée $e_1.e_2.\dots.e_n$, avec $e_i \leq e_{i+1}$.
 - ▶ Equations récurrentes:

$$\begin{aligned} \text{Insere}(x, \emptyset) &= (x, \emptyset) \\ \text{Insere}(x, (a, L)) &= (x, (a, L)) && \text{si } x \leq a \\ \text{Insere}(x, (a, L)) &= (a, \text{Insere}(x, L)) && \text{sinon} \end{aligned}$$

- ▶ Correction: par induction, si on suppose (a, L) triée, le résultat est bien trié.

```
static Liste Insere(int x, Liste a) {
    if (a==null) return new Liste(x, null);
    if (x <= a.contenu) return new Liste(x,a);
    return new Liste(a.contenu, Insere(x,a.suivant));
}
```

23

Tri par insertion

```
static Liste Trie(Liste p) {
    Liste r = null;
    for (; p != null; p = p.suivant)
        r = Insere(p.contenu,r);
    return r;}
}
```

- Complexité $I(n)$ de *Insere* en comparaisons : $I(n) = O(n)$.
- Complexité $C(n)$ de *Trie* en comparaisons: $C(n) \leq nI(n) = O(n^2)$.
- Le pire cas est atteint lorsqu'on trie une liste déjà triée, et mène à $\Omega(n^2)$ comparaisons. La complexité du tri par insertion est donc bien en $\Theta(n^2)$.

24

Tri par insertion en moyenne

- Le meilleur cas est atteint lorsqu'on trie une liste dans l'ordre décroissant, et mène à $O(n)$ comparaisons.
- $O(n) \leq C(n) \leq O(n^2)$.
- En moyenne? Si on se restreint au tri des listes d'entiers distincts deux à deux, et en supposant les permutations équiprobables,

$$\begin{aligned}C(n) &= \frac{1}{2}n^2 + \frac{3}{4}n - \ln n + O(1) \\ &= \Theta(n^2)\end{aligned}$$

26

(Supplément) Preuve

Principe:

- Coût moyen de Insere avec déjà $k - 1$ éléments:

$$I(k) = \frac{1}{k}(1 + 2 + \dots + k - 1 + k - 1) = \frac{1}{k}\left(\frac{k(k+1)}{2} - 1\right)$$

(position de l'élément inséré équiprobable)

- Coût moyen de Trie:

$$C(n) = 0 + \frac{1}{2}\left(\frac{2(2+1)}{2} - 1\right) + \dots + \frac{1}{n}\left(\frac{n(n+1)}{2} - 1\right)$$

(k premiers éléments répartis uniformément)

26

Tri par fusion

- Fusion:
 - Construire une liste triée qui contienne l'union des éléments de deux listes triées.
- Exemple: $Fusion(1.4.5.7, 2.4.9) = 1.2.4.4.5.7.9$
- Equations récursives de $Fusion(X, Y)$:

$$\begin{aligned}Fusion(X, \emptyset) &= X \\ Fusion(\emptyset, Y) &= Y \\ Fusion((a, L), (b, M)) &= (a, Fusion(L, (b, M))) \text{ si } a \leq b \\ Fusion((a, L), (b, M)) &= (b, Fusion((a, L), M)) \text{ sinon}\end{aligned}$$

- Correction: par induction, si on suppose (a, L) et (b, M) triées, le résultat est correct.

27

Fusion

```
static Liste Fusion(Liste a, Liste b) {
    if (a == null)
        return b;
    if (b == null)
        return a;
    if (a.contenu < b.contenu)
        return new Liste(a.contenu, Fusion(a.suivant, b));
    else
        return new Liste(b.contenu, Fusion(a, b.suivant));
}
```

- Complexité en nombre de comparaisons:
 $O(\text{longueur}(a) + \text{longueur}(b))$.

28

Tri fusion

- Une liste de 0 ou 1 élément est triée.
- Toute autre liste L
 - ▶ peut se découper en deux sous-listes L_1 et L_2 de même taille (à 1 près).
 - ▶ Les sous-listes L_1 et L_2 sont triées récursivement,
 - ▶ puis fusionnées.

$$\text{TriFusion}(L) = \text{Fusion}(\text{TriFusion}(L_1), \text{TriFusion}(L_2)).$$

29

Paradigme "Diviser pour règner"

- Complexité $C(n)$ en nombre de comparaisons en $O(\text{Fusion}) + 2C(n/2)$, soit

$$C(n) = O(n + 2C(n/2)),$$

ce qui mène à

$$C(n) = O(n \log n).$$

- Note:
 - ▶ comment résoudre une telle récurrence: poser $n = 2^p$ et faire le changement de variable $C'(p) = C(2^p)$.
 - ▶ $C'(p) = O(2^p + 2C'(p-1))$,
 - ▶ donc $C'(p) = O(2^p p)$.

30

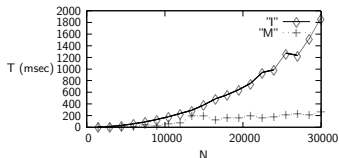
Tri fusion

```
static Liste MergeSort(Liste l) {
    Liste l1 = null, l2 = null;
    boolean even = true;
    for (; l != null; l = l.suivant) {
        if (even) {
            l1 = new Liste (l.contenu, l1);
        } else {
            l2 = new Liste (l.contenu, l2);
        }
        even = !even;
    }
    if (l2==null) return l1;
    return Fusion(MergeSort(l1),MergeSort(l2));
}
```

- Note: L_1 et L_2 sont en fait ici dans l'ordre inverse des éléments pairs et impairs, par commodité.
- Note: le tri fusion fait *toujours* de l'ordre de $n \log n$ comparaisons, et pas seulement au pire cas.

31

Insertion vs. fusion



I est le tri par insertion.
M est le tri par fusion.

32

Complexité du problème du tri

■ Tout algorithme de tri

- ▶ qui fonctionne par comparaisons,
- ▶ qui n'a pas d'autres informations sur les données,

effectue $\Omega(n \log n)$ comparaisons dans le pire des cas.

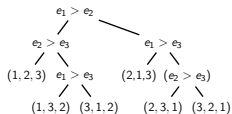
■ Autrement dit, la complexité du problème du tri (avec ces hypothèses) est en $\Theta(n \log n)$.

■ Le tri par fusion est donc parmi les tris optimaux en $O(n \log n)$.

33

Preuve de la borne inférieure $\Omega(n \log n)$: 1/2

■ A un algorithme on peut associer un arbre de décision:



- ▶ En chaque noeud qui n'est pas une feuille: une comparaison effectuée par l'algorithme.
 - La racine est la première comparaison.
 - Fils gauche: récursivement ce qui se passe alors pour un résultat positif.
 - Fils droit: récursivement ce qui se passe alors pour un résultat négatif.
- ▶ En chaque feuille, le résultat produit.

34

Preuve de la borne inférieure $\Omega(n \log n)$: 2/2

■ Un arbre binaire de hauteur h a au plus 2^h feuilles.

- ▶ par récurrence.

■ Les feuilles doivent contenir au moins les $n!$ permutations, et donc $h \geq \log(n!) = \Omega(n \log n)$ par la formule de Stirling.

35

Aujourd'hui

Programmer avec des listes

Complexité

Trier

Piles. Files.

Utilisation des piles et des files

Programmer des piles et des files

Piles et files dans la bibliothèque JAVA

36

Le type abstrait "sac"

- On souhaite considérer une structure de données, que l'on va appeler *sac*, permettant de stocker des objets.
- On veut pouvoir:
 - ▶ tester si un sac est vide?
 - ▶ ajouter un élément dans un sac.
 - ▶ retirer un élément d'un sac non vide.

```
class Sac {
...
Sac() { ... } //Construire un sac vide
boolean EstVide() { ... } //Tester si un sac est vide
void Ajouter(Objet e) { ... } //Ajouter un élément à un sac
Objet Enlever() { ... } //Enlever un élément d'un sac
}
```

37

Les types abstraits

permettent

- de décrire un objet uniquement par ses fonctions de base.
- de faire abstraction de la réalisation.
 - ▶ Un sac peut être implémenté par une liste, par un tableau, ...
 - ▶ Modifier son implémentation ne modifie pas le reste du programme.

La notion d'interface, d'héritage, et de module sera vue plus avant dans le cours INF431.

38

Files. Piles

- Les piles et les files sont des sacs.
- Une pile est un sac LIFO (Last-In First-Out)
 - ▶ lorsqu'on retire un élément, on récupère toujours le dernier ajouté.
 - ▶ ajouter se dit *empiler* (push), retirer se dit *dépiler* (pop).
- Une file est un sac FIFO (First-in First-Out)
 - ▶ on retire les éléments dans l'ordre de leur insertion.
 - ▶ ajouter se dit *enfiler*, retirer se dit *défiler*.



39

Piles, Files d'entiers

```
class Pile {
...
Pile() { ... }
boolean EstVide() { ... }
int Tete() { ... }

void Empiler(int e) { ... }
int Dépiler() { ... }
}

Pile s= new Pile();
```

```
class File {
...
File() { ... }
boolean EstVide() { ... }
int Tete() { ... }

void Enfiler(int e) { ... }
int Défiler() { ... }
}

File s= new File();
```

40

Aujourd'hui

Programmer avec des listes

Complexité

Trier

Piles, Files.

Utilisation des piles et des files

Programmer des piles et des files

Piles et files dans la bibliothèque JAVA

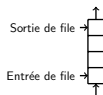
42

Les files

- Elles sont présentes dès qu'on a besoin d'utiliser ou de modéliser une file d'attente.

par exemple:

- les impressions dans un système d'exploitation sont gérées par une file;
- dans une simulation d'un guichet, on modélisera généralement l'arrivée des clients par une file.



- Il existe une théorie des files d'attente en évaluation de performances/recherche opérationnelle.

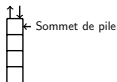
43

Les piles

- Les piles sont des objets informatiques omniprésents:

par exemple:

- les annulations d'édition dans un traitement de texte sont gérées par une pile;
- reconnaître une expression bien parenthésée nécessite une pile;
- les appels récursifs sont gérés par une pile de récursion;
- évaluer une expression arithmétique nécessite une pile.



44

Analyse syntaxique

- Reconnaître une expression bien parenthésée avec deux types de parenthèses [et (.
 - [([])] est bien parenthésé.
 - [(]) est mal parenthésé.

```
static boolean BienParenthésé(String s) {
    Pile p = new Pile ();
    for (int i=0; i<s.length(); i++) {
        char c= s.charAt(i);
        if (c=='[' || c=='(') p.Empiler(c);
        else if (c==']') {
            if (!p.EstVide() && p.Dépiler() != '[')
                return false;
        }
        else if (c==')') {
            if (!p.EstVide() && p.Dépiler() != '(')
                return false;
        }
        return p.EstVide();
    }
}
```

- Les méthodes `s.length()` et `s.charAt(i)` de la classe `String` retournent la longueur et le caractère numéro `i` de `s`.

45

Les appels récursifs sont gérés par une pile

Une preuve.

```
static void f() {  
    int a = 1;  
    f();  
}  
  
public static void main (String [] args) {  
    f();  
}
```

Affiche: *Exception in thread "main" java.lang.StackOverflowError*

46

Comment sont gérés les appels récursifs?

En simplifiant:

- Le système d'exécution dispose d'une pile.
- Lorsqu'on appelle une fonction (méthode):
 - ▶ on empile l'adresse de retour (pc = program counter)
 - ▶ on empile les arguments de la fonction
 - ▶ on se rend à l'adresse de la fonction, pour exécuter son code.
- Le code de la méthode:
 - ▶ Alloue éventuellement des variables locales dans la pile
 - ▶ Lorsqu'on exécute l'instruction **return** (ou termine)
 - on dépile variables locales + arguments
 - on dépile l'adresse de retour pc
 - on empile éventuellement la valeur de retour
 - on se rend à l'adresse pc

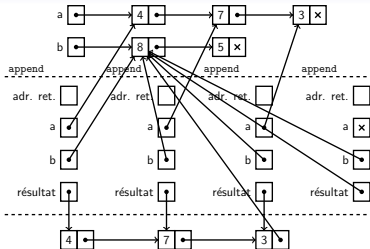
47

Un exemple

```
static Liste append(Liste a,Liste b) {  
    if (a== null)  
        return b;  
    else  
        return new Liste(a.contenu, append(a.suivant,b));  
}
```

48

La pile système s'obtient par une lecture dans l'ordre 1- haut vers bas, 2- gauche vers droite des cases entre les lignes en pointillés.



49

Un autre exemple

```
static Liste Fusion(Liste a, Liste b) {
    if (a == null)
        return b;
    if (b == null)
        return a;
    if (a.contenu < b.contenu)
        return new Liste(a.contenu, Fusion(a.suivant, b));
    else
        return new Liste(b.contenu, Fusion(a, b.suivant));
}
```

- A quoi ressemble le code compilé de cette fonction?

50

Code compilé

- Un modèle de processeur en JAVA.
 - ▶ Toutes les variables sont globales (registres).
 - ▶ Le contrôle est explicite:
 - il y a des étiquettes, spécifiant des instructions particulières.
 - il y a des instructions **goto** (transfert direct) et **goto*** (transfert indirect).
- Le code de Fusion:

```
Fusion: // A ce moment, la pile contient @adr. ret., a, b
    b = S.Dépiler();
    a = S.Dépiler();
    S.Empiler(a);
    S.Empiler(b);
    if (a== null) {
        S.Dépiler();
        S.Dépiler();
        lab= S. Dépiler();
        S.Empiler(b); //A ce moment, la pile contient b
        goto* lab;
    }
    ...
```

51

```
...
if (a.contenu < b.contenu) {
    S.Empiler(lab1); // Préparation appel Fusion(a.suivant,b)
    S.Empiler(a.suivant);
    S.Empiler(b);
    goto Fusion;
lab1:
//A ce moment, la pile contient @adr. ret.,a,b,résultat
    r = S.Dépiler(); // résultat de l'appel récursif
    b = S.Dépiler();
    a = S.Dépiler();
    // On construit new Liste(a.contenu, Fusion(a.suivant,b));
    r = new Liste(a.contenu,r);
    lab = S.Dépiler(); // on récupère l'adresse de retour
    S.Empiler(r); // on empile le résultat
    goto* lab;
}
...
```

52

Aujourd'hui

Programmer avec des listes

Complexité

Trier

Piles. Files.

Utilisation des piles et des files

Programmer des piles et des files

Piles et files dans la bibliothèque JAVA

53

Programmer des piles et des files

- Solution 1: avec des listes.
- Solution 2: avec des tableaux.

54

Aujourd'hui

Programmer avec des listes

Complexité

Trier

Piles, Files.

Utilisation des piles et des files

Programmer des piles et des files

Piles et files dans la bibliothèque JAVA

55

Utiliser les bibliothèques JAVA

- Piles et files sont tellement utilisées qu'elles sont proposées dans les classes JAVA standards.
- Exemple: la classe `Stack` du package `java.util` (qui correspond à l'implémentation par tableaux dynamiques précédente) pour les piles.
- Des piles de quoi?

56

Classes génériques

Les codes

- d'une pile/file d'entiers
- d'une pile/file de réels
- d'une pile/file de chaînes de caractères
- d'une pile/file de piles
- d'une pile/file de files
- d'une pile/file de ...

se ressemblent beaucoup.

57

Classes génériques

- Les classes de la bibliothèque JAVA sont en fait génériques.
 - ▶ Stack est une classe générique qui prend une classe en argument.
 - ▶ Exemple: Stack<String> désigne une pile de chaînes de caractères.
 - ▶ Plus généralement, Stack<E> est une pile d'objets de la classe E.

58

Exemple (attention ordre opérations incorrect)

```
import java.util.*;

class Traduit {
    public static void main (String [] args) {
        Stack<String> p = new Stack<String> () ;
        for (int i = 0 ; i < args.length ; i++) {
            String cmd = args[i] ;
            if (cmd.equals("+") || cmd.equals("-")
                || cmd.equals("*") || cmd.equals("/"))
                p.push("(" + p.pop() + cmd + p.pop() + ")");
            else
                p.push(cmd) ;
        }
        System.out.println(p.pop()) ;
    }
}
```

- Dans les librairies standards, Empiler se dit push, Dépiler se dit pop, EstVide() se dit empty, Tete se dit peek (voir documentation JAVA).

59

Comment créer une pile d'entiers

- On ne peut pas utiliser le type Stack<int>, car int n'est pas une classe.
- Cependant, la bibliothèque JAVA standard fournit une classe appropriée par type scalaire.
- Par exemple, la classe Integer pour int:
 - ▶ La classe Integer est une classe avec un champ private de type int.
 - ▶ On convertit un scalaire int en un objet Integer et réciproquement par:
 - public static Integer.valueOf(int i): du scalaire vers un Integer.
 - public int intValue(): réciproquement.
- Il y a des classes semblables (Char, Short, etc.) pour tous les types scalaires (char, short, etc.).

60

Exemple

```
import java.util.*;

class Calc {
    public static void main (String [] arg) {
        Stack<Integer> stack = new Stack<Integer> () ;
        ...
        int i1 = stack.pop().intValue() ;
        int i2 = stack.pop().intValue() ;
        stack.push(Integer.valueOf(i2+i1)) ;
        ...
    }
}
```

- En fait, le compilateur accepte stack.push(i2+i1), car il fait les conversions int vers Integer tout seul.

61

Files de la bibliothèque

- La classe `LinkedList<C>` (package `java.util`) fait l'affaire.
- Il s'agit en fait d'une *double-ended queue*.
- On peut utiliser
 - ▶ les méthodes `addFirst` et `addLast` (`put`) pour ajouter un élément au début ou à la fin.
 - ▶ les méthodes `removeFirst` (`get`) et `removeLast` pour récupérer le premier ou dernier élément.
 - ▶ Le tout en temps constant, par des listes doublement chaînées (voir le poly, section 2.3.3).
- Il existe d'autres classes de la bibliothèque standard avec des noms plus séduisants (comme `Queue`), mais `LinkedList<C>` est peut-être la plus pratique.