

Cours 3: Piles. Files.

Olivier Bournez

bournez@lix.polytechnique.fr
LIX, Ecole Polytechnique

INF421-b

Bases de la programmation et de l'algorithmique

Aujourd'hui

Piles. Files.

Utilisation des piles et des files

Programmer des piles et des files

Piles et files dans la bibliothèque JAVA

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  
}
```

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.

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 *empiler*, retirer se dit *défiler*.



5

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();
```

6

Aujourd'hui

Piles. Files.

Utilisation des piles et des files

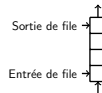
Programmer des piles et des files

Piles et files dans la bibliothèque JAVA

8

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.

9

Simulation d'une file d'attente

- Phénomène souvent stochastique:
 - ▶ Arrivées et départs aléatoires des clients.
 - ▶ Temps de service aléatoire.
- On cherche souvent à comprendre certaines caractéristiques du système.
 - ▶ Exemple: nombre moyen de clients en attente, temps de service moyen, ...
- C'est l'objet de la théorie des files d'attente
 - ▶ qui vise à produire des résultats analytiques.
- Le seul outil est parfois la simulation pour les cas complexes.

10

Un exemple simple

- Le modèle:
 - ▶ Un unique guichet ouvert 8h00 ($8 \times 3600 = 28800$ 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 .
 - on suppose que la probabilité d'arrivée de plus que deux clients sur un intervalle $[t, t + 1]$ est négligeable (loi de Poisson).
 - ▶ Le temps de service suit une loi uniforme sur $[30, 300]$.
 - ▶ Chaque client est plus ou moins patient: le temps après lequel il part sans être servi est donné par une loi uniforme sur $[120, 1800]$.
- Problème à résoudre:
 - ▶ Évaluer le rapport clients non-servis sur nombre total de clients en fonction de p .

11

Les clients

- Pour faire un tirage aléatoire:
 - ▶ il faut créer une variable partagée de la classe Random.

```
import java.util.*;
...
static Random rand = new Random();
```

- ▶ `rand.nextInt(n)` retourne alors un entier entre 0 et $n - 1$ selon une loi uniforme.
- ▶ `rand.nextFloat()` retourne alors un réel entre 0 (inclus) et 1 (exclu) selon une loi uniforme.

```
class Client {
    int arrivee;
    int seuil;
    Client(int arrivee) {
        this.arrivee = arrivee;
        this.seuil = arrivee+ 120 + rand.nextInt(1800-120);
    }
}
```

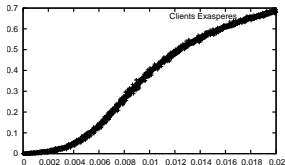
12

La file d'attente

```
static float Simule(float p) {
    File f = new File (); int libre = 0, clientsArrivés = 0;
    int clientsExaspérés = 0;
    for (int t = 0; t < 28800; t++) {
        if (rand.nextFloat() <= p) { // Une arrivée?
            clientsArrivés++; f.Enqueue(new Client(t));
        }
        if (libre <= t) { // Le guichet est il libre?
            Client c = null;
            while (! f.EstVide()) {
                c = f.Défiler();
                if (t <= c.seuil) break;
                clientsExaspérés ++; c = null;
            }
            if (c != null) // c est le prochain client à servir
                libre = t + 30 + rand.nextInt(300-30);
        }
    }
    return ((float) clientsExaspérés)/clientsArrivés;
}
```

13

Tracé avec gnuplot (de moyennes sur plusieurs simulations)



- Abscisse: p , la probabilité d'arrivée d'un client par seconde.
- Ordonnée: rapport clients non-servis sur nombre total de clients.

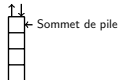
16

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.



15

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`.

16

Complément: comment fonctionne une calculatrice?

(sur les entiers, qui gère les priorités entre + et *)

- On utilise des accumulateurs S , P et C .
 - ▶ C : nombre courant.
 - ▶ P : produit dont fait partie ce nombre (ou 1 par défaut).
 - ▶ S : somme des autres produits rencontrés jusque-là.
- Initialement, $S = 0$, $P = 1$, $C = 0$.
- Action:
 - ▶ Appui sur $x \in \{0, \dots, 9\}$: $C = 10*C+x$; `Afficher(C)`;
 - ▶ Appui sur $*$: $P=P*C$; `Afficher(P)`; $C=0$;
 - ▶ Appui sur $+$: $S = S + P*C$; `Afficher(S)`; $P=1$; $C=0$;
 - ▶ Appui sur $=$: $S = S + P*C$; `Afficher(S)`; $S=0$; $P=1$; $C=0$;
 - ▶ Appui sur $($: `Empiler (S)`; `Empiler(P)`; $S=0$; $P=1$; $C=0$;
 - ▶ Appui sur $)$: `P=Dépiler()* (S+P*C)`; `S=Dépiler()`; $C=0$
- Exemple: $14 + 2 * 3 = 20$

17

Plus simple à comprendre:

Calculatrices en notation polonaise inverse (RPN)

- La calculatrice manipule et affiche explicitement une pile
- Principe:
 - ▶ Rentrer un entier C correspond à Empiler(C);
 - ▶ Appuyer sur un symbole $\oplus \in \{+, *, /, -\}$, correspond à $x = \text{Dépiler}(); y = \text{Dépiler}(); \text{Empiler}(y \oplus x)$;
- Exemple: 14 2 3 * + donne 20

- Exemple: 14 2 3 + * donne 70

18

Programmer une calculatrice RPN

```
class Calculatrice {
    public static void main (String [] args) {
        Pile p = new Pile ();
        for (int i = 0 ; i < args.length ; i++) {
            String cmd = args[i];
            if (cmd.equals("+")) {
                int x=p.Dépiler(), y=p.Dépiler();
                p.Empiler(y+x);
            }
            else
                ... // Pareil pour "*", "/", "-"
            else
                p.Empiler(Integer.parseInt(cmd));
        }
        System.out.println(p.Dépiler());
    }
}
```

(Integer.parseInt() transforme une chaîne de caractères codant un entier en cet entier).

19

Exemple

```
% java Calculatrice 14 2 3 * +
donne 20.
```

```
% java Calculatrice 14 2 3 + *
donne 70.
```

```
% java Calculatrice 14 +
produit une erreur.
```

20

Traduire une notation post-fixe (RPN) en infixe

```
class Traduit {
    public static void main (String [] args) {
        Pile p = new Pile ();
        for (int i = 0 ; i < args.length ; i++) {
            String cmd = args[i];
            if (cmd.equals("+") || cmd.equals("-")
                || cmd.equals("*") || cmd.equals("/")) {
                int x=p.Dépiler(), y=p.Dépiler();
                p.Empiler("(" + y + cmd + x + ")");
            }
            else
                p.Empiler(cmd);
        }
        System.out.println(p.Dépiler());
    }
}
```

- On utilise cette fois une pile de chaînes de caractères.
- Le résultat retourné est complètement parenthésé.
% java Traduit 14 2 3 * +
donne (14+(2*3)).

21

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*

22

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

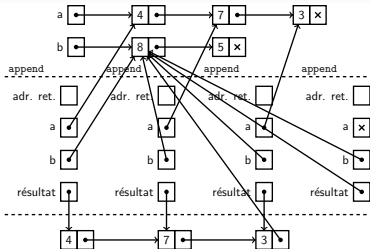
23

Un exemple

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

24

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.



25

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?

26

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;}
    ...
```

27

```
...
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;
}
...
```

28

Aujourd'hui

Piles. Files.

Utilisation des piles et des files

Programmer des piles et des files

Piles et files dans la bibliothèque JAVA

29

Programmer des piles et des files

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

30

Une pile avec une liste

```
class Pile {  
    private Liste l; // sommet de la pile  
    Pile() { l=null; }  
    boolean EstVide() { return (l==null); }  
    int Tete() { return l.contenu;}  
  
    void Empiler(int e) { l=new Liste(e,l); }  
    int Dépiler() { int c=l.contenu; l=l.suivant; return(c); }  
}
```



- Le mot clé **private** permet de garantir qu'aucune autre classe ne puisse modifier ou accéder au champ l.
- Cela permet de garantir qu'il n'est pas modifié autrement que par les méthodes de la classe Pile.

31

Gérer les erreurs

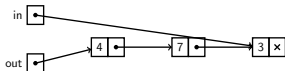
```
class Pile {  
    private Liste l; // sommet de la pile  
    Pile() { l=null; }  
    boolean EstVide() { return (l==null); }  
    int Tete() { if (l==null) throw new Error("Pile Vide");  
                return l.contenu;}  
  
    void Empiler(int e) { l=new Liste(e,l); }  
    int Dépiler() { if (l==null) throw new Error("Pile Vide");  
                  int c=l.contenu; l=l.suivant; return(c); }  
}
```

- **throw new Error("Pile Vide")** arrête l'exécution du programme en lançant (instruction **throw**) une exception (ici un objet de la classe Error).
- Affiche: *Exception in thread "main" java.lang.Error: Pile Vide* en cas d'erreur.

32

Une file avec une liste

```
class File {  
    private Liste in; // entrée de la liste  
    private Liste out; // sortie de la liste  
    File() { in=out=null; }  
    boolean EstVide() { return (in==null && out==null); }  
    int Tete() { return out.contenu;}  
    ...  
}
```



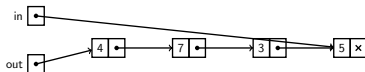
Le champ

- out pointe sur la première cellule de liste (pour enlever)
- in pointe sur la dernière cellule de liste (pour ajouter)

33

Enfiler

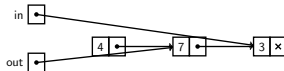
```
void Enfiler(int i) {  
    if (EstVide())  
        in = out = new List(i, null);  
    else {  
        in.suivant = new List(i, null);  
        in = in.suivant;  
    }  
}
```



36

Défiler

```
int Défiler() {  
    if (EstVide())  
        throw new Error("Pile Vide");  
    int r = out.contenu;  
    out = out.suivant;  
    if (out == null)  
        in = null;  
    return r;  
}
```

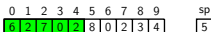


- Ne pas oublier de remettre in à **null** lorsque la pile est vide.

37

Une pile avec un tableau

- On crée un tableau suffisamment grand.
- On utilise un entier *sp*, le *pointeur de pile*, qui compte le nombre d'éléments dans la pile.
- Les éléments d'indice inférieur à *sp* contiennent les éléments de la pile.
- Les autres, ici en blanc, sont disponibles.



- Principe:
 - ▶ Tete(): t[sp-1].
 - ▶ Empiler(e): t[sp++] = e.
 - ▶ Dépiler(): t[--sp].

38

Une pile avec un tableau

```
class Pile {  
    final static int TAILLE = 100;  
    private int[] t;  
    private int sp;  
  
    Pile() { t = new int[TAILLE]; sp = 0; }  
    boolean EstVide() { return (sp <= 0); }  
  
    int Tete() {  
        if (EstVide()) throw new Error("Pile Vide");  
        return t[sp-1];  
    }  
  
    void Empiler(int e) {  
        if (sp >= t.length) throw new Error("Pile Pleine");  
        t[sp++] = e; }  
  
    int Dépiler() {  
        if (EstVide()) throw new Error("Pile Vide");  
        return t[--sp]; } }  
}
```

39

Une pile avec un tableau dynamique

```
private void resize() {
    int[] newT = new int [2*t.length];
    for (int i=0; i<t.length; i++)
        newT[i] = t[i];
    t = newT;
}
```

```
void Empiler(int e) {
    if (sp >= t.length) resize();
    t[sp++] = e;
}
```

38

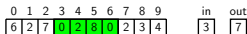
Analyse de complexité

- Les versions précédentes de Empiler() et Dépiler() fonctionnaient en temps $O(1)$ (temps constant).
- Maintenant, Empiler() peut provoquer la copie d'un tableau arbitrairement grand, et donc prendre un temps arbitrairement grand.
- Cependant, Empiler() reste en coût $O(1)$ amorti (coût global ramené au nombre d'opérations).
 - Supposons que l'on fasse N fois Empiler() ou Dépiler(), et que la taille initiale du tableau soit 2^{p_0} .
 - Dans le pire cas, le tableau est redimensionné $p + 1 - p_0$ fois, avec $2^p < N \leq 2^{p+1}$.
 - Coût total dans ce cas:
 $O(2^{p_0} + 2^{p_0+1} + \dots + 2^{p+1}) = O(2^{p+2}) < O(4N) = O(N)$.
 - Ce qui fait un coût amorti $O(1)$ par élément. En effet, le coût global/ le nombre d'éléments est $\leq O(N)/N = O(1)$.

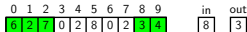
39

Une file avec un tableau

- On crée toujours un tableau de taille TAILLE suffisamment grande.
- On utilise cette fois deux entiers in, et out.
- Les éléments entre in et out-1 contiennent les éléments de la file.
- Les autres, ici en blanc, sont disponibles.



- Variante: on peut supposer le tableau circulaire: les éléments entre in et out-1 modulo TAILLE contiennent les éléments de la file.



40

Une file avec un tableau

```
class File {
    final static int TAILLE = 100;
    private int[] t;
    private int in,out;

    File() { t=new int[TAILLE]; in=out=0; }
    boolean EstVide() { return (in == out); }

    boolean EstPleine() { return ((out+1) % TAILLE == in);}

    void Enfiler(int e) {
        if (EstPleine()) throw new Error("Pile Pleine");
        t[out]=e;
        out = (out + 1) % TAILLE;
    }

    int Défiler() {
        if (EstVide()) throw new Error("Pile Vide");
        int c=t[in];
        in = (in+1) % TAILLE;
        return (c); }
}
```

41

Aujourd'hui

Piles, Files.

Utilisation des piles et des files

Programmer des piles et des files

Piles et files dans la bibliothèque JAVA

42

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?

43

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.

44

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.

45

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).

46

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.).

47

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.

48

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.

49