Slide 1

Programmer avec des listes

Slide 2

0. Résumé des épisodes précédents

Slide 3

Enregistrements

Type de données dynamiques = enregistrements + récursivité

un champ récursif : type de listes

plusieurs champs récursifs : type d'arbres

Slide 4

I. Les ensembles finis

Slide 5

Les ensembles finis

```
static boolean mem (final int s, final List b) {
 if (b == null) return false;
 if (s == b.hd) return true;
 return mem(s, b.tl);}
```

Slide 6

Le nombre d'opérations

Rechercher : linéaire en le nombre d'éléments de l'ensemble

Ajouter : constant

Supprimer : linéaire

Les fonctions de domaine fini

Ensemble fini de couples : liste d'associations

Slide 7

```
class AList {
 int cle;
 int val;
 AList tl;

 AList (int x, int y, AList l) {
  this.cle = x; this.val = y; this.tl = l;}}
```

Slide 8

```
static int assoc (final int x, final AList l) {
 if (l == null) return 0;
 if (x == l.cle) return l.val;
 return assoc (x, l.tl);}
```

Slide 9

II. Concaténer : modifier ou copier

Slide 10

```
static List append (final List x, final List y) {
 if (x == null) return y;
 List p = x;
 while (p.tl != null) p = p.tl;
 p.tl = y;
 return x;}
```

```
a = new List(3,new List(1,null));
b = new List(4,null);
c = append(a,b);
```

Slide 11

```
printList(c);

printList(a);

printList(b);
```

Slide 12

```
printList(c);

printList(a);

printList(b);

d = append(a,a);
printList(d);
```

Slide 13

```
static List append (final List x, final List y) {
 if (x == null) return y;
 else {
  List p = x;
  List q = new List(x.hd,null);
  List r = q;
  while (p.tl != null) {
   q.tl = new List (p.tl.hd,null);
   q = q.tl;
   p = p.tl;}
  q.tl = y;
  return r;}}
```

Slide 14

```
a = new List(3,new List(1, null));
b = new List(4,null);
c = append(a,b);
```

Slide 15

```
printList(c);

printList(a);

printList(b);
```

Slide 16

```
printList(c);

printList(a);

printList(b);


d = append(a,a);
printList(d);
```

Slide 17

La même récursivement

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

Slide 18

La première fonction `append`

- modifie ses arguments,
- demande que ses arguments ne partagent pas de cellules
- n'alloue pas

La seconde

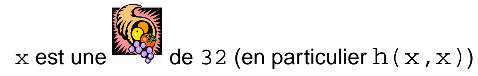
- ne modifie pas ses arguments,
- permet à ses arguments de partager des cellules
- alloue

Slide 19

Les réactions chimiques et les fonctions mathématiques

En mathématiques : $f = y \mapsto y + 1, g = y \mapsto 2\ y$

$x = 32,\ f(x),\ g(x)$

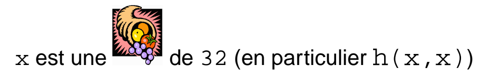
x est une  de 32 (en particulier $h(x,x)$)

Slide 20

Les réactions chimiques et les fonctions mathématiques

En mathématiques : $f = y \mapsto y + 1, g = y \mapsto 2\ y$

$x = 32,\ f(x),\ g(x)$

x est une  de 32 (en particulier $h(x,x)$)

En chimie : $O_2 + 2\ H_2 \longrightarrow 2\ H_2O$

Les molécules d'oxygène et d'hydrogène sont consommés

pas de cornes d'abondance

une molécule d'hydrogène ne peut pas réagir avec elle-même

Slide 21

Quand utiliser quelle fonction ?

Modifier : impératif sur de grosses données

E.g. on ajoute un nom à l'annuaire : on veut modifier l'annuaire

Copier : à chaque fois que l'on peut

les données ne disaparaissent pas

pas besoin de se soucier du partage

simple à programmer

Slide 22

III. Inverser une liste : un argument de plus

Slide 23

Inverser une liste

1 2 3 4 5 $\longrightarrow$ 5 4 3 2 1

Slide 24

Inverser une liste

1 2 3 4 5 $\longrightarrow$ 5 4 3 2 1

Slide 25

```
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);}
```

Slide 26

```
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));}
```

Nombre d’opérations proportionnel à la longueur de `x`

Slide 27

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

Nombre d'opérations proportionnel à `1 + 2 + ... + n`

Équivalent à n^2 à une constante près

Fonction `reverse` en n^2

Slide 28

Complexité en temps d'un programme

Fonction `f(n)` : nombre d'opérations effectuées par un programme quand on l'éxécute sur une entrée de taille `n`

– ne dépend que de `n` (`reverse`)
– en moyenne (sur toutes les données de taille `n`)
– dans le pire des cas (sur toutes les données de taille `n`)

Slide 29

Complexité en temps d'un programme

Fonction `f(n)` : nombre d'opérations effectuées par un programme quand on l'éxécute sur une entrée de taille `n`

– ne dépend que de `n` (`reverse`)

– en moyenne (sur toutes les données de taille `n`)

– dans le pire des cas (sur toutes les données de taille `n`)

Si $f(n) = 3\ n^2 + 18\ n + 5$

Slide 30

Complexité en temps d'un programme

Fonction `f(n)` : nombre d'opérations effectuées par un programme quand on l'éxécute sur une entrée de taille `n`

– ne dépend que de `n` (`reverse`)

– en moyenne (sur toutes les données de taille `n`)

– dans le pire des cas (sur toutes les données de taille `n`)

Si $f(n) = 3\ n^2 + 18\ n + 5$

Slide 31

Complexité en temps d'un programme

Fonction `f(n)` : nombre d'opérations effectuées par un programme quand on l'éxécute sur une entrée de taille `n`

- ne dépend que de `n` (`reverse`)
- en moyenne (sur toutes les données de taille `n`)
- dans le pire des cas (sur toutes les données de taille `n`)

Si $f(n) = 3\ n^2 + 18\ n + 5$

Une fonction `g` équivalente à `f` à une constante près

`lim f/g` finie non nulle

Slide 32

Peut-on inverser une liste en temps linéaire ?

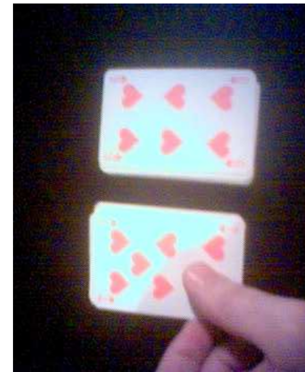
Slide 33

Au début ...

Slide 34

Et après un temps linéaire ...

Slide 35



Slide 36

Slide 37

Slide 38

Slide 39

```
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);}
```

Slide 40

IV. Les piles et les files

Des structures différentes

Des opérations différentes sur le même ensemble

Les listes :

– fabriquer la liste vide (`null`)
– tester si une liste est vide (`l == null`)
– fabriquer une liste non vide (`new List`)
– accéder à la tête d'une liste (`l.hd`)
– accéder à la queue d'une liste (`l.tl`)
– modifier la tête d'une liste (`l.hd = t`)
– modifier la queue d'une liste (`l.tl = t`)

Les piles

Des opérations différentes sur le même ensemble

Les piles :

– fabriquer la pile vide
– tester si une pile est vide
– ajouter un premier élément à une pile
– accéder au premier élément d'une pile
– retirer le premier élément d'une pile

Slide 43

Slide 44

Ajouter un élément modifie la pile

`l2 = new List(3,l1) ;` ne modifie pas la liste `l1`

`push(3,p) ;` modifie la pile `p`

`p` ne peut pas être de type `List`

Le type des piles : le type enveloppé des listes

```
class Pile  {
 List c;

 Pile(List a) {this.c = a;}}
```

Slide 45

```
static Pile empty () {
 return new Pile (null);}

static void push (final int a, final Pile l) {
 l.c = new List(a,l.c);}
```

Slide 46

Un exemple d'utilisation des piles

Évaluer une expression post-fixée (notation polonaise)

```
2 4 5 + 3 * +
```

Un exemple d'utilisation des piles

Évaluer une expression post-fixée (notation polonaise)

`2 4 5 + 3 * +`

On lit les éléments un par un

- un nombre : on le met au sommet de la pile
- un `+` : on pope deux éléments et on met leur somme au sommet de la pile
- un `*` : on pope deux éléments et on met leur produit au sommet de la pile

Slide 47

Un exemple d'utilisation des piles

2

2

Slide 48

Un exemple d'utilisation des piles

2 4

2

2 4

Slide 49

Un exemple d'utilisation des piles

2 4 5

2

2 4

2 4 5

Slide 50

Slide 51

Un exemple d'utilisation des piles

```
2 4 5 +
```

```
2
2 4
2 4 5
2 9
```

Slide 52

Un exemple d'utilisation des piles

```
2 4 5 + 3
```

```
2
2 4
2 4 5
2 9
2 9 3
```

Un exemple d'utilisation des piles

```
2 4 5 + 3 *
```

```
2
2 4
2 4 5
2 9
2 9 3
2 27
```


Un exemple d'utilisation des piles

```
2 4 5 + 3 * +
```

```
2
2 4
2 4 5
2 9
2 9 3
2 27
29
```

Slide 55

Les files

Des opérations différentes sur le même ensemble

Les files :

- fabriquer la file vide
- tester si une file est vide
- ajouter un dernier élément à une file
- accéder au premier élément d'une file
- retirer le premier élément d'une file

Slide 56

premier arrivé premier servi

Les files

Slide 57

Type des file : type enveloppé des listes

Ajouter un élément à la fin de la liste prend un temps linéaire (et non constant) en la longueur de la liste

Les files de priorité

Slide 58

Chaque élément a un niveau de priorité

On accède à un élément de priorité maximale, et on le supprime (Urgences)

Les files de priorité

Slide 59

Liste : accès et suppression linéaires

Liste ordonnée par priorité décroissante : ajout linéaire

Slide 60

V. Le GC

Slide 61

```
e = [x = r1,y = r4]
m = [r1 = r2, r2 = {hd = 1, tl = r3},
r3 = {hd = 2, tl = null}, r4 = r5,
r5 = {hd = 3, tl = r6}, r6 = {hd = 4, tl =
null}]
```

`y = null;`

`r5` peut être recyclée

puis `r6`

Slide 62

Nouvelle mémoire `m'`

Différence entre `m` et `m'` non observable

Pour tout `p`, Σ`(p,e,m,G,C) = `Σ`(p,e,m',G,C)`

Mais devient observable si `Ref` est remplacé par un ensemble fini : on épuise la mémoire

Deux générations de langages

Slide 63

C : on gère la mémoire à la main

on met `r5` et `r6` dans une liste

prochaine fois qu'on a besoin d'une cellule : on utilise `r6` au lieu d'allouer

Java, Caml : le GC (*Garbage collector*, Glaneur de cellules)

Recyclage automatique

Le marquage

Slide 64

On marque en partant des références déclarées

On ramasse tout ce qui n'est pas marqué

Slide 65

Exercices 6.10 et 6.16

Slide 66

La prochaine fois : les objets