Développement efficace (R3.02)
Récursivité III : listes

Marin Bougeret
LIRMM, IUT/Université de Montpellier

Introduction

- précédemment : algorithmes récursifs
- maintenant : structures récursives

- précédemment : définir un algorithme à partir de lui même
- maintenant :
  - définir une structure à partir d'elle même
  - manipuler ces structures avec des algorithmes récursifs

On parle également de type "autoréférents".

Outline

Outline

Définition informelle

Notations

- la liste vide sera notée ()
- les listes non vides seront notées, par exemple, $(7, 8, 9)$

Définition en Java

```
class Liste{
  private int val;
  private Liste suiv;

  public Liste(){
    this.suiv = null;
  }


}
```

Remarque

Différence par rapport à l'année dernière il y avait deux classes : Maillon et Liste.

Comment représenter la liste vide ?

- `Liste l = null` ?
- `Liste l = new Liste()`?

Pensons à une classe familière : ArrayList.

```
.. main(..) {
    ArrayList<Integer> l = new ArrayList(); //on
        démarre d'une liste vide
    l.add(3);

    //pareil pour nous
    Liste l2 = new Liste();
    l2.ajout(5);

    //si on faisait
    Liste l3 = null;
    l3.uneMethode(); //nullpointerexception
```

Définition en Java

```
class Liste{
  private int val;
  private Liste suiv;

  public Liste(){ //construit la liste vide
    this.suiv = null;
  }

  public boolean estVide(){
    //une liste (val,suiv) est vide ssi suiv==
        null (peu importe val)
    return suiv==null
  }
}
```

Remarque

On a donc un "maillon bidon" à la fin de toutes les listes.

Définition en Java

Exemple

Comment construire la liste L représentant $(1, 2, 3)$?

- Liste L_1 = new Liste();
- Liste L_2 = new Liste();
- Liste L_3 = new Liste();

Définition en Java

Exemple

Comment construire la liste L représentant $(1, 2, 3)$?

- Liste L_1 = new Liste();
- Liste L_2 = new Liste();
- Liste L_3 = new Liste();
- L_1.val = 1;
- L_2.val = 2;
- L_3.val = 3;

Définition en Java

Exemple

Comment construire la liste L représentant $(1, 2, 3)$?

- Liste L_1 = new Liste();
- Liste L_2 = new Liste();
- Liste L_3 = new Liste();
- L_1.val = 1;
- L_2.val = 2;
- L_3.val = 3;
- L_1.suiv = L_2;
- L_2.suiv = L_3;//attention, on a ici seulement $L_1 = (1, 2)$

Définition en Java

Exemple

Comment construire la liste L représentant $(1, 2, 3)$?

- Liste L_1 = new Liste();
- Liste L_2 = new Liste();
- Liste L_3 = new Liste();
- L_1.val = 1;
- L_2.val = 2;
- L_3.val = 3;
- L_1.suiv = L_2;
- L_2.suiv = L_3;//attention, on a ici seulement $L_1 = (1, 2)$
- L_3.suiv = new Liste();//ici $L_1 = (1, 2, 3)$

Liste Vs Tableau

Les 2 structures de données les plus usuelles sont :

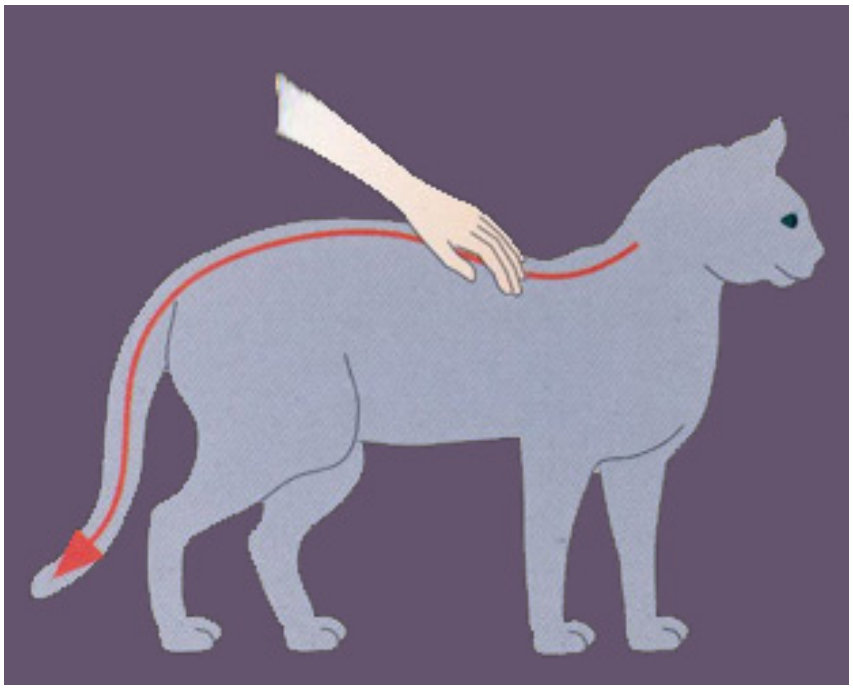
- les tableaux
- les listes

Est ce qu'une des deux structures est mieux que l'autre ? NON!

Propriétés des listes & tableaux

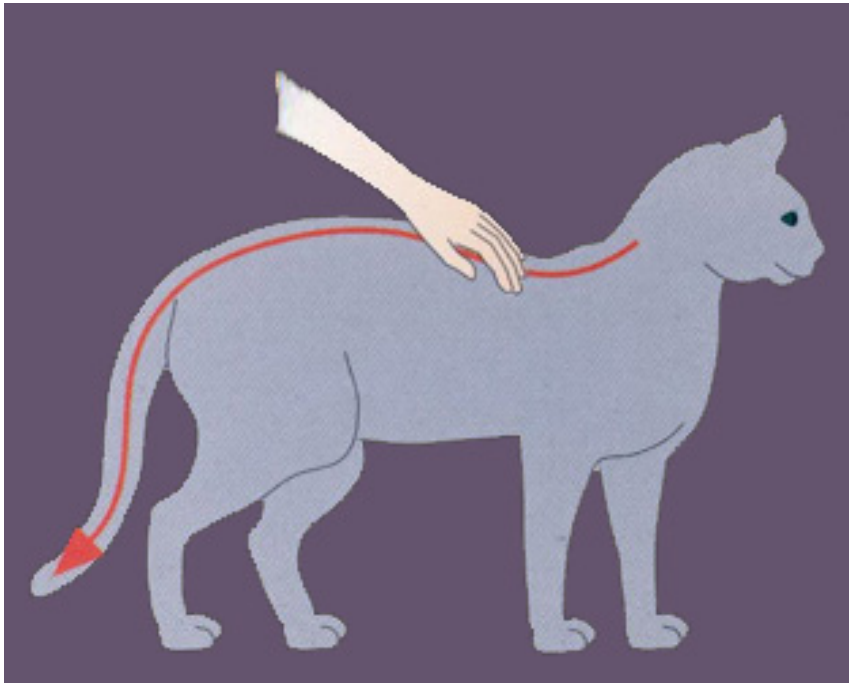
- tableaux :
  - AVANTAGE : accès immédiats à tous les éléments
  - INCONVENIENT : taille fixée (tableaux redims. n'existent pas)
- listes
  - AVANTAGE : taille non fixée (ajout d'élément toujours ok)
  - INCONVENIENT : accès aux éléments en fin de liste coûteux

Introduction



- Avant : algo. itératifs sur les listes
- Maintenant : algo. récursifs sur les listes

Introduction



- Avant : algo. itératifs sur les listes
- Maintenant : algo. récursifs sur les listes

Outline

toString

```
String toString(){

   if(estVide()){
      return "";
   }
   else{
      String aux = suiv.toString();
      return val +"␣"+aux;
   }
}
```

toStringEnvers

```
String toStringEnvers(){

   if(estVide()){
      return "";
   }
   else{
      String aux = suiv.toStringEnvers();
      return aux+"␣"+val;
   }
}
```

recherche

```
boolean recherche(int x){
  //action : retourne vrai ssi x dans this

  if(estVide()){
    return false;
  }
  else{
    if (val == x)
      return true;
    else
      return suiv.recherche(x);
  }
}
```

Une version équivalente, mais mieux écrite :

```
boolean recherche(int x){
  //action : retourne vrai ssi x dans this

  if(estVide()){
    return false;
  }
  else{
    return ((val==x)|| suiv.recherche(x));
  }
}
```

Liste(Liste l)

Un constructeur utile pour la suite :

```
Liste Liste(Liste l){
  //action : constructeur par copie en
      profondeur (recopie tous les maillons)

  if(l.estVide()){
    suiv =  null;
  }
  else{
    val = l.val;
    suiv = new Liste(l.suiv);
  }
}
```

Outline

Briques de base

Pour cette partie

On s'intéresse aux méthodes avec des spécifications du type
`void m(..)` : modifie this afin que ...

Les méthodes de base suivantes vont être utiles pour écrire ces algorithmes:

- `void ajoutTete(int x)`
- `void supprimeTete()`

Regardons donc comment écrire ces deux méthodes de base.

void supprimeTete()

Un version fausse :

```
public void supprimeTete(){
  //this non vide
  this = this.suiv; //"this =" est INTERDIT en
      JAVA
}
```

void supprimeTete()

```
public void supprimeTete(){
  //this non vide
  this.val = suiv.val;
  this.suiv = this.suiv.suiv;
}
```

Pénible à écrire, on sera donc content d'invoquer `supprimeTete()`.

void ajoutTete(int x)

Une version fausse :

```
public void ajoutTete(int x){
  Liste res = new Liste();
  res.val = x;
  res.suiv = this;
  this = res; //"this =" est INTERDIT en JAVA
}
```

void ajoutTete(int x)

```
public void ajoutTete(int x){
  Liste copieTete = new Liste();
  copieTete.val = this.val;
  copieTete.suiv = this.suiv;
  this.val = x;
  this.suiv = copieTete;
}
```

Pénible à écrire, on sera donc content d'invoquer `ajoutTete()`.

Ecrivons maintenant une méthode `void` avec nos méthodes de base.

void supprOccs(int x)

```
void supprOccs(int x){
  //modifie this pour supprimer toutes les occs
      de x
  if(!estVide()){
    suiv.supprOccs(x);
    if(val==x){
      supprimeTete();
    }
  }
}
```

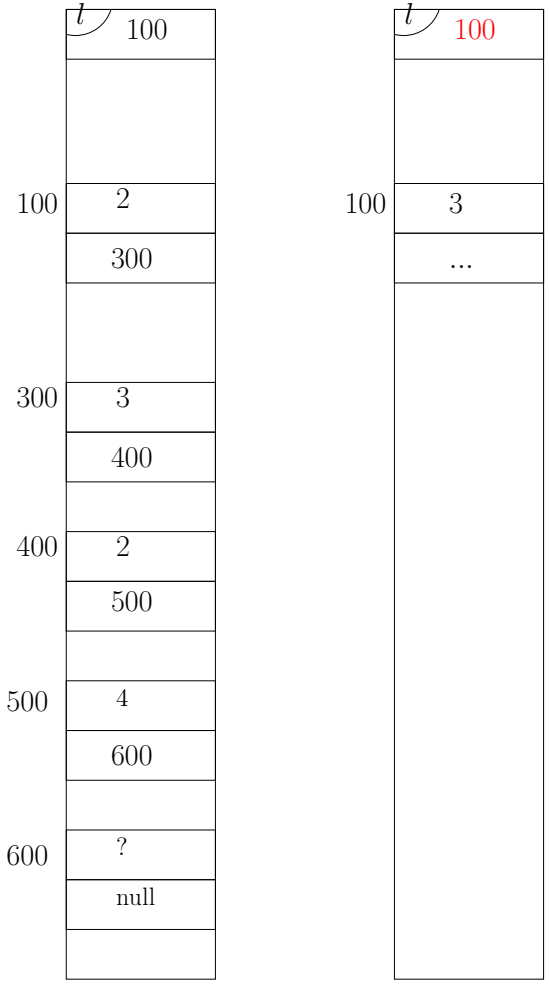
Un petit coup de stress..

- avant l'appel rec, this.suiv vallait 500 (par ex)
- après l'appel rec (qui a "tout changé" à partir du deuxième maillon), this.suiv .. vaut toujours 500 !
- sommes nous sûr que le premier "nouveau maillon" (après l'appel rec.) est bien à l'adresse 500 ?
- oui, cf slide suivant

Remarque sur la spécification "modifie this"

Avec une spécification du type "void m(..) : modifie this afin que ..", si l'on exécute "l.m(..)", le contenu de la liste va être modifié, mais pas le pointeur l.

```
Liste l = .. ;//on suppose l=
     (2,3,2,4)
Liste sauv = l;
l.supprOccs(2);
println(l==sauv); //true
```



Un deuxième exemple

Spec : "modifie this (qui est triée) pour y insérer x"

```
void insertDansTriee(int x){
  if(estVide())
    ajoutTete(x);
  else{
    if(x <= val)
      ajoutTete(x);
    else
      suiv.insertDansTriee(x);
  }
}
```

Outline

estEgale

```
boolean estEgale(Liste l){
  //action : retourne vrai ssi this et l
      contiennent les mêmes entiers, dans le mê
      me ordre

  if(estVide()){
    return l.estVide();
  }
  else{
    if(l.estVide())
      return false;
    //les deux listes sont vides, on peut donc
        regarder les valeurs en tête
    return ((val==lval) && suiv.estEgale(l.
        suiv));
  }
}
```

Notion d'égalité

Attention

estEgale n'est pas la même chose que == :

- l1.estEgale(l2) vrai ssi l1 et l2 contiennent les mêmes entiers, dans le même ordre
- l1==l2 vrai ssi l1 et l2 pointent sur la même adresse

- l1==l2 $\Rightarrow$ l1.estEgale(l2)
- la réciproque est fausse:
  - Liste l1 = new Liste(10);
  - Liste l2 = new Liste(10);
  - //on a l1!=l2 mais l1.estEgale(l2)

Notion d'indépendance

Définition

On dit que deux listes l_1 et l_2 sont indépendantes ssi elles n'ont aucun "maillon" en commun, plus formellement :

- $\forall$ Liste $l'_1 != null$ accessible depuis l_1 (ex $l'_1 = l_1.suiv.suiv$),
- $\forall$ Liste $l'_2 != null$ accessible depuis l_2
- on doit avoir $l'_1 != l_2$

```
Liste L1 = new Liste(1);
Liste L1b = new Liste(2);
Liste L1c = new Liste(3);
Liste L1d = new Liste(4);
L1.suiv = L1b;
L1b.suiv = L1c;
L1c.suiv = L1d; //ici L1=(1,2,3,4)
Liste L2 = new Liste(5);
L2.suiv = L1c; //ici L2=(5,3,4)
//L1 et L2 ne sont donc pas indépendantes
    car L1.suiv.suiv == L2.suiv
```

Notion d'indépendance

Définition

On dit que deux listes l_1 et l_2 sont indépendantes ssi elles n'ont aucun "maillon" en commun, plus formellement :

- $\forall$ Liste $l_1' != null$ accessible depuis l_1 (ex $l_1' = l_1.suiv.suiv.suiv$),
- $\forall$ Liste $l_2' != null$ accessible depuis l_2 (ex $l_2' = l_2.suiv.suiv$),
- on doit avoir $l_1' != l_2$

Remarque

- attention, si l_1 et l_2 ne sont pas indépendantes alors une modification d'une liste peut changer l'autre!
- dans l'exemple précédent:
  - si l'on fait L1c.=10 (remplacer le 3 de L1 par 10), L2 devient (5,10,4)

Notion d'indépendance

Définition

On dit que deux listes l_1 et l_2 sont indépendantes ssi elles n'ont aucun "maillon" en commun, plus formellement :

- $\forall$ Liste $l_1'! = \mathit{null}$ accessible depuis l_1 (ex $l_1' = l_1.\mathit{suiv}.\mathit{suiv}.\mathit{suiv}$),
- $\forall$ Liste $l_2'! = \mathit{null}$ accessible depuis l_2 (ex $l_2' = l_2.\mathit{suiv}.\mathit{suiv}$),
- on doit avoir $l_1'! = l_2$

Remarque

On impose $l_1'! = \mathit{null}$ et $l_2'! = \mathit{null}$ dans la définition .. car autrement on aurait jamais l_1 et l_2 indépendante, puisqu'on pourrait choisr $l_1' = \mathit{null}$, et $l_2' = \mathit{null}$ (qui sont bien accessibles depuis l_1 et l_2), et on aurait $l_1' == l_2'$.

ajoutTete

Une V1 où l'on ne demande pas de retourner une liste indépendante :

```
Liste ajoutTeteV1(int x){
  //action : ajoute x en tete de this

  return new Liste(x,this);
}
main(...){
  Liste l =; // on construit l=(1,2,3);
  Liste l2 = l.ajoutTeteV1(0); //l2 n'est pas
     indep de l car l2.suiv == l
  //au passage, il est courant de faire
  l = l.ajoutTeteV1(0);
  //on évite ainsi de garder 2 listes non indé
     pendantes
}
```

Une V2 où l'on demande de retourner une liste indépendante :

```
Liste ajoutTeteV2(int x){
  //action : ajoute x en tete de this
  //et retourne une liste indépendante

  return new Liste(x,new Liste(this));
}
main(...){
  Liste l =; // on construit l=(1,2,3);
  Liste l2 = ajoutTeteV2(l,0); //ici l2 est
     indépendante de l

}
```

Différentes variantes des spécifications

Pour ajoutTete comme pour beaucoup d'autres fonctions, plusieurs versions sont possibles selon que

- l'on demande de retourner une `Liste` ou qu'on soit de type `void`
- et même lorsque l'on retourne une `Liste`, on peut :
  - demander de retourner une liste indépendante ou non
  - autoriser ou non à modifier this (c'est à dire à modifier une des valeurs, ou un des chaînages) ou un paramètre
  - ou d'autres contraintes (par ex n'allouer qu'un seul nouveau maillon)

Il n'y a pas vraiment de version mieux qu'une autre, il faut être capable de toute écrire.

Illustrons ces variantes pour la méthode `insertDansTriee`.

insertDansTriee

Spec V0 : modifie this pour y insérer x (qui est triée).

```
void insertDansTrieeV0(int x){
  //méthode vue précédemment
```

insertDansTriee

Spec V1 : retourne la liste obtenue en insérerant x dans this (qui est triée), et ne crée qu'un seul nouveau maillon. Remarque : la liste retournée ne sera pas indépendante.

```
Liste insertDansTrieeV1(int x){
  if(estVide())
    return new Liste(x);
  else{
    if(x <= val)
      return new Liste(x,this); //c'est ici
          qu'on créé le maillon, et qu'on
          perd l'indépendance
    else{
      this.suiv = suiv.insertDansTrieeV1(x);
           //ici on modifie le paramètre
      return this;
    }
  }
}
```

insertDansTriee

Spec V1 : retourne la liste obtenue en insérerant x dans this (qui est triée), et ne crée qu'un seul nouveau maillon. Remarque : la liste retournée ne sera pas indépendante.

Intérêt de la V1 par rapport à la `void` V0

- retourner une liste non indépendante peut paraître dangereux : `L2 = L1.insertDansTrieeV1(5);`
- mais ce type de retour est pratique pour le "method chaining": on peut écrire des expressions du type `L1.insertDansTrieeV1(4).insertDansTrieeV1(6)`
- et de plus, si l'on récupère le résultat dans `L1`, on évite d'avoir 2 variables dépendantes: `L1=L1.insertDansTrieeV1(4).insertDansTrieeV1(6)`

insertDansTriee

Spec V2 : retourne la liste obtenue en insérant x dans this (qui est triée), sans modifier this, et en retournant une liste indépendante.

```
Liste insertDansTrieeV2(int x){
  if(estVide())
    return new Liste(x);
  else{
    if(x <= val)
      return new Liste(x,new Liste(this));
     else
   return new Liste(val,suiv.insertDansTrieeV2(x));
  }
}
```

Attention, pas besoin de faire :

```
return new Liste(val,new
    Liste(suiv.insertDansTrieeV2(x)));
```