

Développement efficace (R3.02)

Récurtivité III : listes

Marin Bougeret
LIRMM, IUT/Université de Montpellier



- 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".

1 Liste

- Définition
- Algorithmes sur les listes : échauffement
- Ecriture des méthodes `void`
- Notions d'égalité, d'indépendance

1 Liste

- Définition
- Algorithmes sur les listes : échauffement
- Ecriture des méthodes `void`
- Notions d'égalité, d'indépendance

Notations

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

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

    public Liste(int x){//construit la liste avec
        un entier x
        this.val = x;
        this.suiv = null;
    }

    public Liste(int x, Liste l){
        this.val = x;
        this.suiv = l;
    }

    public boolean estVide(){
        return suiv==null
    }
}
```

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
```

Exemple

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

- Liste $L_1 = \text{new Liste}(1);$
- Liste $L_2 = \text{new Liste}(2);$
- Liste $L_3 = \text{new Liste}(3);$

Exemple

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

- Liste $L_1 = \text{new Liste}(1);$
- Liste $L_2 = \text{new Liste}(2);$
- Liste $L_3 = \text{new Liste}(3);$
- $L_1.\text{suiv} = L_2;$
- $L_2.\text{suiv} = L_3;$ // on a ici $L_1 = (1, 2, 3)$

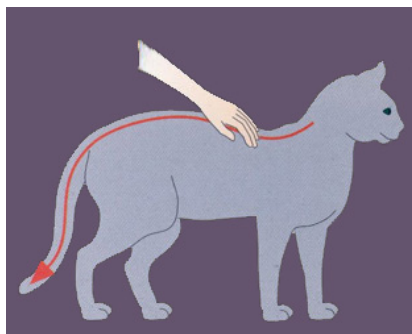
Les 2 structures de données les plus usuelles sont :

- les tableaux
- les listes

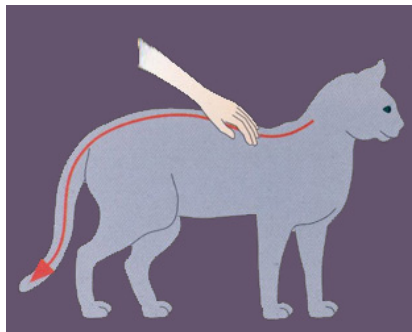
Est ce qu'une des deux structures est mieux que l'autre ? NON!

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



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



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

1 Liste

- Définition
- Algorithmes sur les listes : échauffement
- Ecriture des méthodes `void`
- Notions d'égalité, d'indépendance

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

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

```
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));  
    }  
}
```

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);  
    }  
}
```

1 Liste

- Définition
- Algorithmes sur les listes : échauffement
- **Ecriture des méthodes void**
- Notions d'égalité, d'indépendance

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.

Un version fausse :

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

```
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()`.

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  
}
```

```
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();  
        }  
    }  
}
```

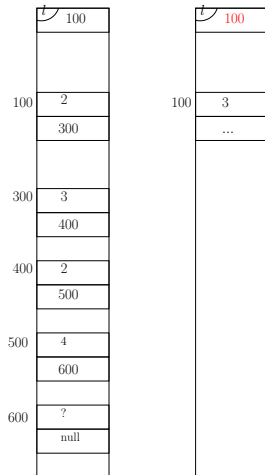
Un petit coup de stress..

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

1 Liste

- Définition
- Algorithmes sur les listes : échauffement
- Ecriture des méthodes `void`
- Notions d'égalité, d'indépendance

```
boolean estEgale(Liste l){  
    //action : retourne vrai ssi this et l  
    contiennent les mêmes entiers, dans le même 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==l.val) && suiv.estEgale(l.  
            suiv));  
    }  
}
```

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)`

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

```
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
```

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)

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

On impose $l'_1! = null$ et $l'_2! = null$ dans la définition .. car autrement on aurait jamais l_1 et l_2 indépendante, puisqu'on pourrait choisir $l'_1 = null$, et $l'_2 = null$ (qui sont bien accessibles depuis l_1 et l_2), et on aurait $l'_1! = l'_2!$.

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 ne 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  
}
```

Indépendante vs Non indépendante

- pour ajoutTete comme pour beaucoup d'autres fonctions, les deux versions sont possibles.
- y a t-il une version mieux que l'autre ?

Non:

- la version sans imposer de retourner une liste indépendante peut être plus rapide, ou permettre d'éviter de créer des maillons
 - .. mais n'a pas l'avantage de retourner une liste indépendante
- toutes les versions peuvent être demandées dans les specs

Spec V1 : "insérer x dans this (qui est triée)" (on ne demande pas de retourner une liste indépendante, donc on se permet effectivement de ne pas en retourner une)

```
Liste insertDansTrieV1(int x){
    if(estVide())
        return new Liste(x);
    else{
        if(x <= val)
            return new Liste(x, this); //ici O(1) :
            on est content de ne pas devoir
            recopier toute la fin de this
        else{
            this.suiv = suiv.insertDansTrieV1(x);
            return l;
        }
    }
}
```

Spec V2 : "insérer x dans this (qui est triée), ne pas modifier le paramètre, et retourner une liste indépendante"

```
Liste insertDansTrieV2(int x){  
    if(estVide())  
        return new Liste(x);  
    else{  
        if(x <= val)  
            return new Liste(x,new Liste(l));  
        else  
            return new Liste(val,suiv.insertDansTrieV2(x));  
    }  
}
```

Attention, pas besoin de faire :

```
return new  
    Liste(l.val,copie(suiv.insertDansTrieV2(x)));
```

Un autre point qui peut également apparaître dans les spécifications : la contrainte de ne pas modifier this

Si l'on revient sur nos deux versions de insertDansTrie,

- la V1 modifiait this
- la V2 modifiait ne modifiait pas this

Un autre point qui peut également apparaître dans les spécifications : la contrainte de ne pas modifier this

Sommes-nous obligés de tout copier pour ne pas modifier this

- NON! une V3 est possible

Spec V3 : "insérer x dans this (qui est triée), ne pas modifier le paramètre"

Il suffit d'éviter de faire `val = ...` , ou `suiv = ...`

```
Liste insertDansTrieV3(int x){  
    if(estVide())  
        return new Liste(x);  
    else{  
        if(x <= val)  
            return new Liste(x,this); //ici on perd  
                l'indépendance, mais ok  
        else  
            return new  
                Liste(val,suiv.insertDansTrieV3(x));  
    }  
}
```

Bilan des différentes spécifications

	ret. liste indep	modifie param	rmq
v1	non	oui	ne crée qu'un maillon
v2	oui	non	recopie tous les maillons
v3	non	non	ne recopie que les maillons avant l'insertion

Conclusion

- En général, on spécifiera seulement si oui/non :
 - on veut retourner une liste indépendante
 - on s'autorise à modifier this
- Une telle spécification est donc volontairement incomplète.
- Exemple : on spécifie uniquement "ne pas modifier le paramètre". Cette spécification est incomplète, et vous pouvez coder v2 ou v3 (ou la version void)

Les deux spécifications qui paraissent les plus naturelles sont les "extrêmes" :

- Liste $m(\dots)$: "ne modifie pas, et retourne liste indépendante qui .."
- void $m(\dots)$: "modifie this pour que .."

Les spécifications du type suivant peuvent sembler étrange :

- Liste $m(\dots)$: "modifie pas, et retourne.."

mais elles permettent l'écriture agréable suivante :

- `l1.m(xx).m(xx).m(xx)`