Algorithmique avancée : introduction à la récursivité & complexité (partie 4)

Marin Bougeret

IUT Montpellier

Plan général

Partie I : algorithmes récursifs

- conception d'algorithmes récursifs (I) : exemples introductifs
- conception d'algorithmes récursifs (II) : tests, idée de preuve, exemples
- conception d'algorithmes récursifs (III) : diviser pour régner

Partie II : structures récursives

- listes et arbres

Partie III : complexité

- ..

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

  boolean estVide(){return this.suiv==null};
}
```

Liste vide

- Par convention, la liste vide est représentée par n'importe quel objet *l* de type Liste avec *l.suiv* == *null* (peut importe *l.val*).
- Liste l = new Liste() représente donc la liste vide
- Liste l = null ne représente pas la liste vide (*l.methode*() ..)

Définition en Java

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

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

  boolean estVide(){return this.suiv==null};
}
```

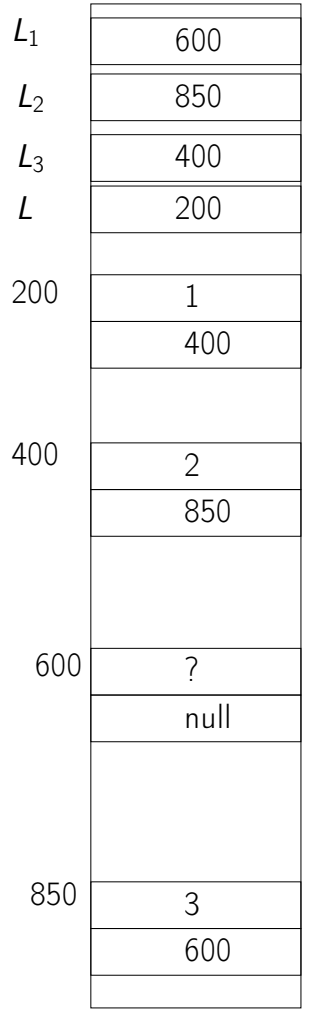
Remarque

Différence par rapport à l'année dernière où vous avez vu une modélisation des listes avec deux classes (Maillon et Liste).

Définition en Java

Exemple

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



Remarque

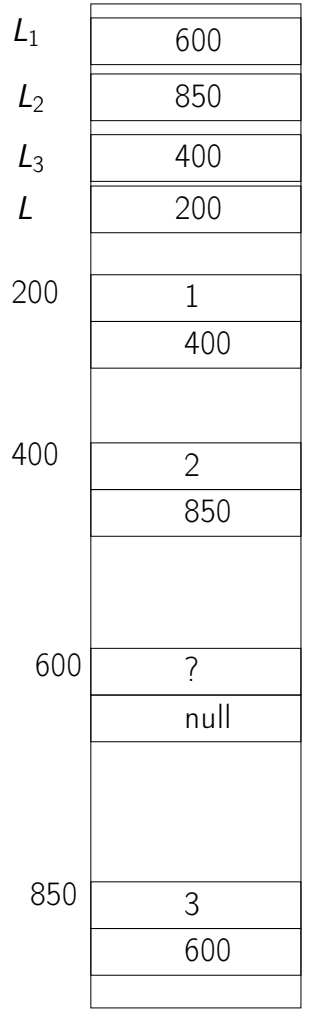
Attention, la notation $(1, 2, 3)$ est traître car elle ne montre pas le maillon vide final.

Définition en Java

Exemple

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

- Liste L_1 = new Liste();



Remarque

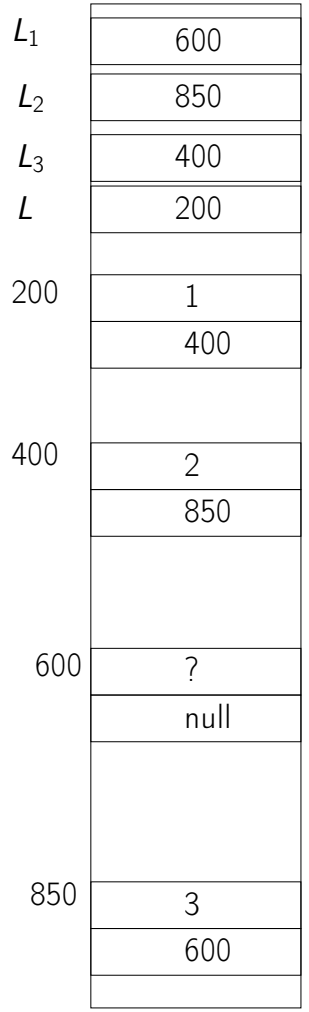
Attention, la notation $(1, 2, 3)$ est traître car elle ne montre pas le maillon vide final.

Définition en Java

Exemple

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

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



Remarque

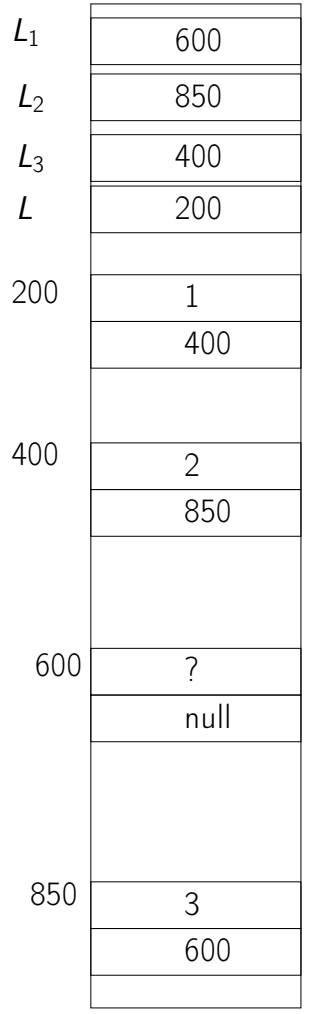
Attention, la notation $(1, 2, 3)$ est traître car elle ne montre pas le maillon vide final.

Définition en Java

Exemple

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

- Liste L_1 = new Liste();
- Liste L_2 = new Liste();
- L_2.val = 3; L_2.suiv = L_1;



Remarque

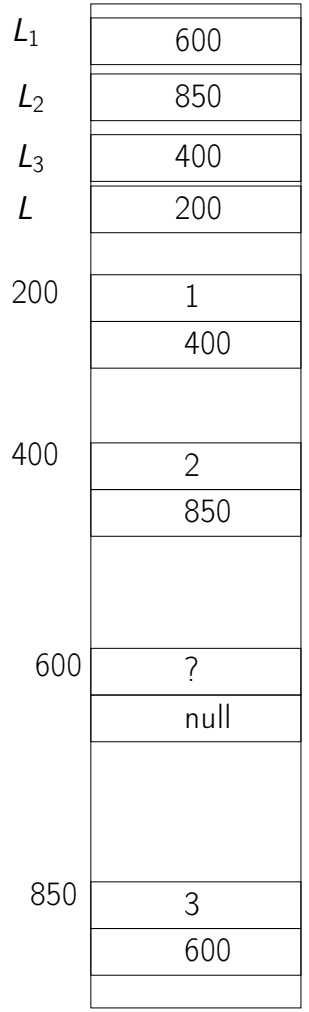
Attention, la notation $(1, 2, 3)$ est traître car elle ne montre pas le maillon vide final.

Définition en Java

Exemple

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

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



Remarque

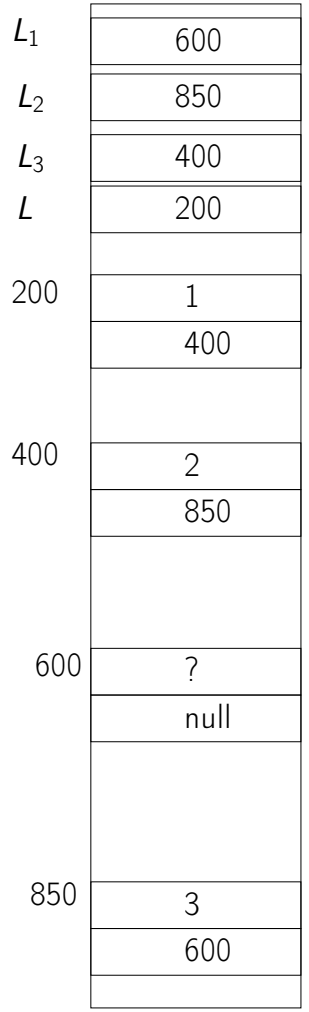
Attention, la notation $(1, 2, 3)$ est traître car elle ne montre pas le maillon vide final.

Définition en Java

Exemple

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

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



Remarque

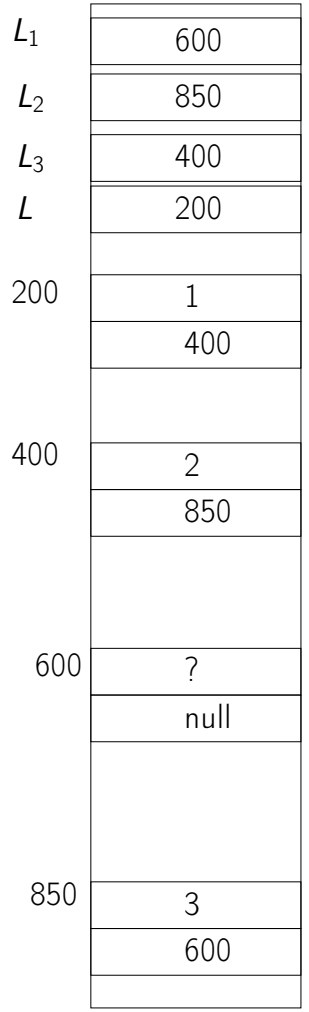
Attention, la notation $(1, 2, 3)$ est traître car elle ne montre pas le maillon vide final.

Définition en Java

Exemple

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

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



Remarque

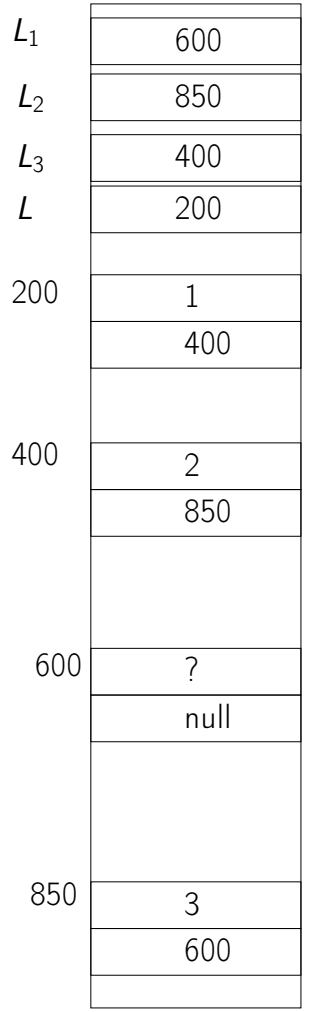
Attention, la notation $(1, 2, 3)$ est traître car elle ne montre pas le maillon vide final.

Définition en Java

Exemple

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

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



Remarque

Attention, la notation $(1, 2, 3)$ est traître car elle ne montre pas le maillon vide final.

Liste Vs Tableau

Les 2 structures de données les plus usuelles sont :

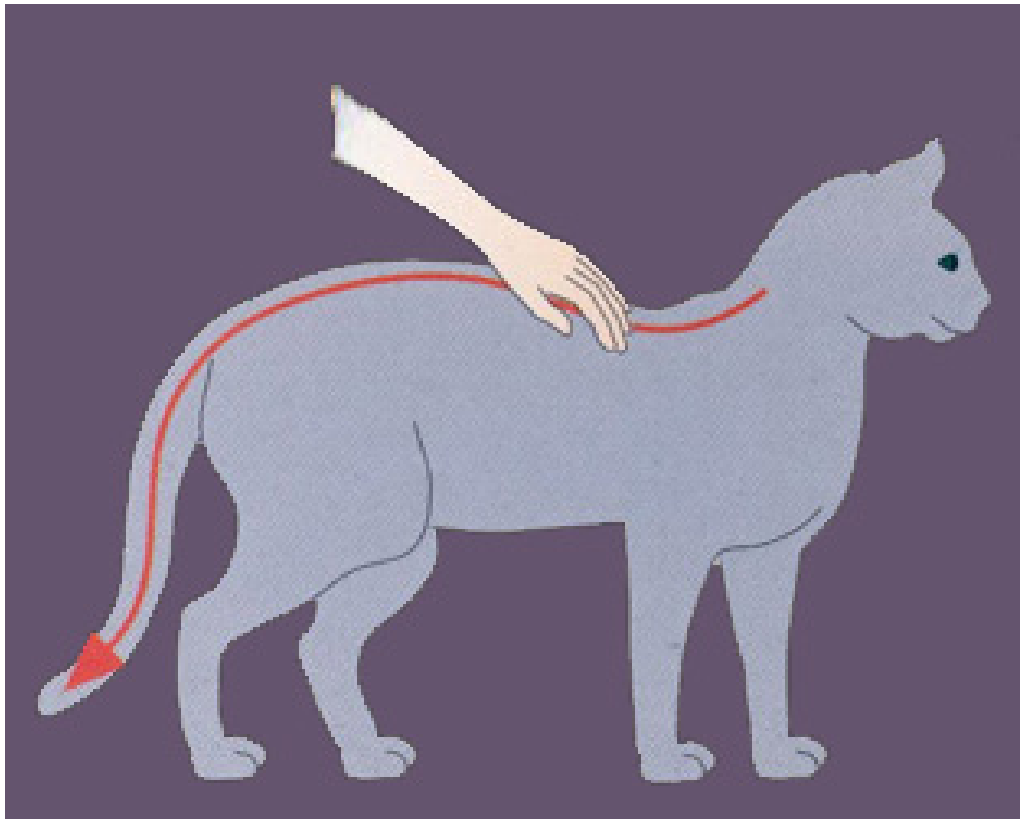
- les tableaux
- les listes

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

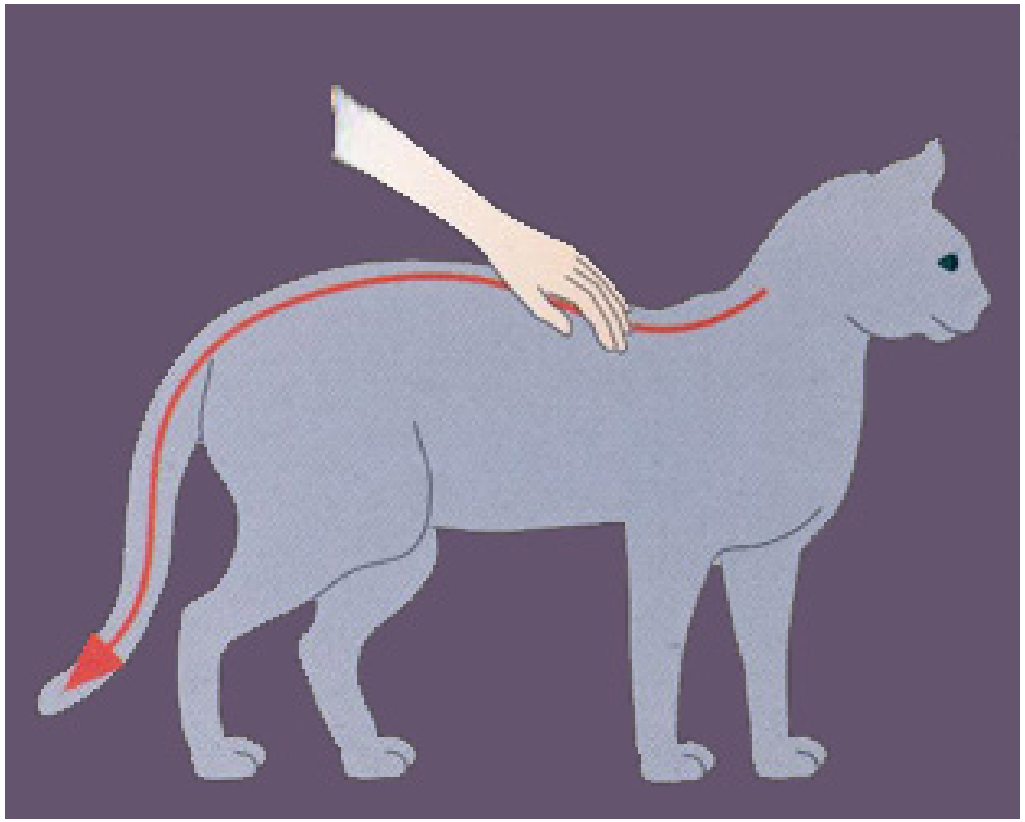
Introduction



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

La méthode de conception habituelle (étape I] et II]) s'applique toujours!

Introduction



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

La méthode de conception habituelle (étape I] et II]) s'applique toujours!

N'oublions pas les principes de l'objet :

Version procédurale :

```
//ds une classe quelconque

  int A(Liste l){
    //prerequis : l
        verifie ...

    if(l.estVide()){
      //cas de base
    }
    else{
       ..
      A(l.suiv);
       ..
    }
  }
```

Version objet :

```
//ds la classe Liste

  int A(){
    //prerequis : this
        verifie ...

    if(this.estVide()){
      //cas de base
    }
    else{
       ..
      this.suiv.A();
       ..
    }
  }
```

Outline

Ex1 : toString()

```
String toString(){
  //prerequis : aucun

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

Ex1 : toStringEnvers()

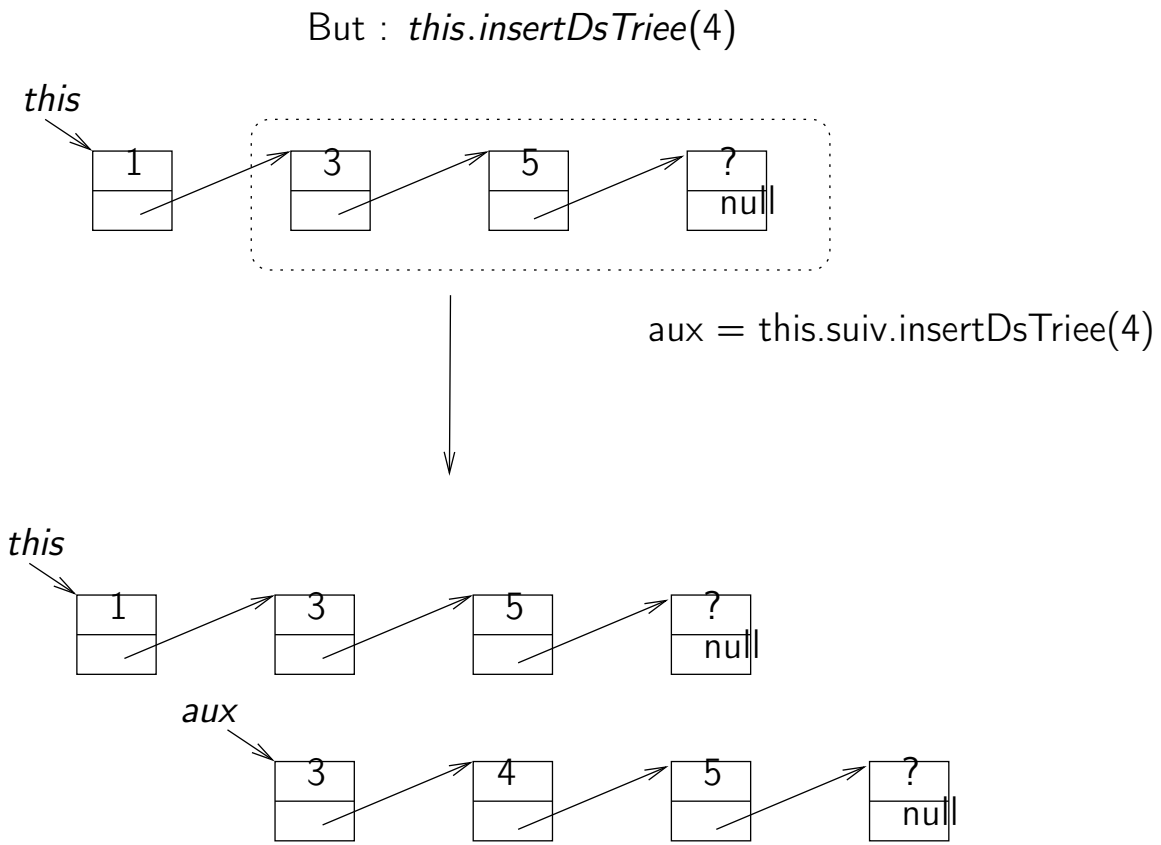
```
String toStringEnvers(){
  //prerequis : aucun

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

Ex2A : Liste supprOccs(int x)

```
public Liste supprOccs(int x){
  //action : suppr. toutes les occurrences de x,
       garde les autres elements dans meme ordre,
       et retourne cela dans une nouvelle liste
  if(estVide())
    return new Liste();
  else{
    Liste aux = suiv.supprOcc(x);
    if(this.val == x){
      return aux;
    }
    else{
      Liste res = new Liste();
      res.val = this.val;
      res.suiv = aux;
      return res;
    }
  }
}
```

Ex2B : Liste insertDsTriee(int x)

But : insérer x dans une liste triée par ordre croissant en créant une nouvelle liste indep. On s'autorise le constructeur par recopie en profondeur Liste(Liste l)



Ex2B : Liste insertDsTriee(int x)

```
public Liste insertDansTriee(int x){
  //prerequis : this triee par odre croissant
  //action : retourne une nouvelle liste triee l
      contenant x
  if(estVide()){
    Liste res = new Liste();
    res.val = x;
    res.suiv = new Liste();
    return res;
  }
  else{
    if(x <= val)
      .. ATTENTION ! user constructeur par
         recopie (mais on peut eviter 0_0)
    else
      Liste aux = suiv.insertDansTriee(x);
      Liste res = new Liste();
      res.val = val;
      res.suiv = aux;
      return res;
```

Outline

Comparaison

Jusqu'à maintenant

- Spéficifications du type "Liste m(..) : retourne une nouvelle liste indépendante de this et ne modifie pas this"

Pour cette partie

- Spéficifications du type "void m(..) : modifie this afin que .."

Ex1 : void supprimeTete()

Un version fausse :

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

Ex1 : void supprimeTete()

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

Pénible à écrire!

Ex2 : 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
}
```

Ex2 : 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!

Bilan

Jusqu'à maintenant

- Spécifications du type "Liste m(..) : retourne une nouvelle liste independante de this et ne modifie pas this"
- Briques de base utilisées pour écrire les algorithmes:
  - Méthode estVide()
  - Constructeur new Liste()

Pour cette partie

- Spécifications du type "void m(..) : modifie this afin que .."
- Briques de base utilisées pour écrire les algorithmes:
  - Méthode estVide()
  - Constructeur new Liste()
  - Méthode void ajoutTete(int x), void supprimeTete() (car pénibles à écrire!)

Ex3 : 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();
    }
  }
}
```

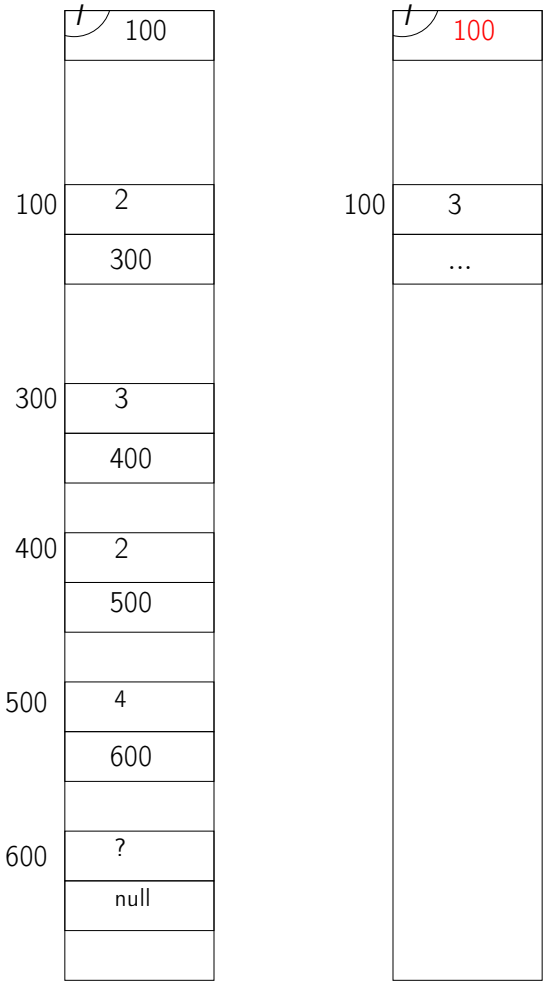
Un petit coup de stress..

- 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
```



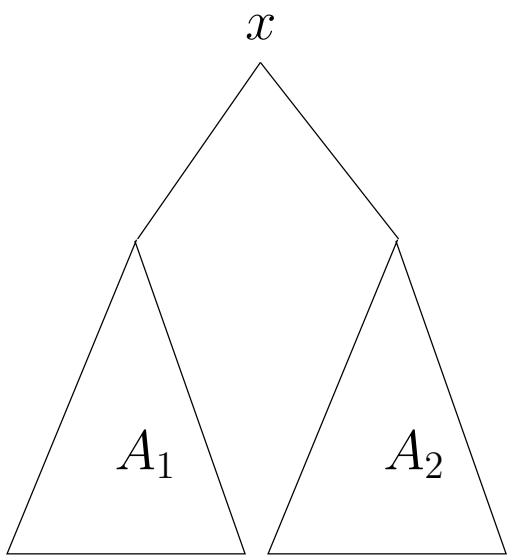
Outline

Outline

Définition informelle

Un arbre est

- soit vide (noté ())
- soit constitué d'une racine $x \in \mathbb{Z}$, et de 2 sous arbres A_1 et A_2



Définition informelle

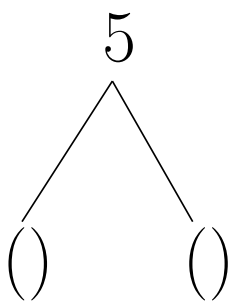
Un arbre est

- soit vide (noté ())
- soit constitué d'une racine $x \in \mathbb{Z}$, et de 2 sous arbres A_1 et A_2

Exemple 1

Comment dessiner l'arbre ayant seulement 5 ?
Plus précisément, cet arbre aura

- 5 pour racine
- un sous arbre gauche vide et un sous arbre droit vide



Définition informelle

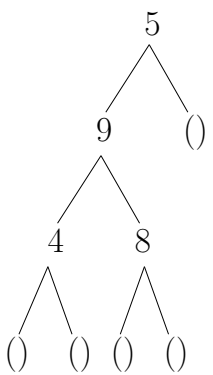
Un arbre est

- soit vide (noté ())
- soit constitué d'une racine $x \in \mathbb{Z}$, et de 2 sous arbres A_1 et A_2

Exemple 2

Comment dessiner l'arbre ayant 5 pour racine, puis

- un sous arbre droit vide
- un sous arbre gauche constitué de 9, ayant lui même 4 à gauche et 8 à droite



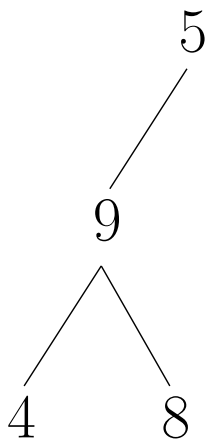
Définition informelle

Un arbre est

- soit vide (noté ())
- soit constitué d'une racine $x \in \mathbb{Z}$, et de 2 sous arbres A_1 et A_2

Remarque

- par abus de notation, nous ne dessinerons plus les arbres vides
- le dessin précédent devient ainsi :



Définition en Java

```
class Arbre{
  private int val;
  private Arbre filsG;
  private Arbre filsD;
  //invariant : filsG==null <=> filsD==null

  public Arbre(){//construit l'arbre vide
    this.filsG = null;
    this.filsD = null;
  }
  boolean estVide()
     return (this.filsG==null);
     //vu l'invariant, pas besoin d'ajouter "&&(
        this.filsD==null)";
}
```

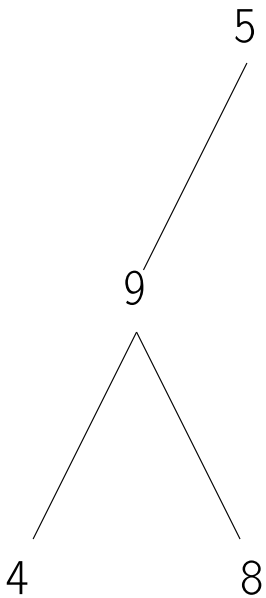
Définition en Java

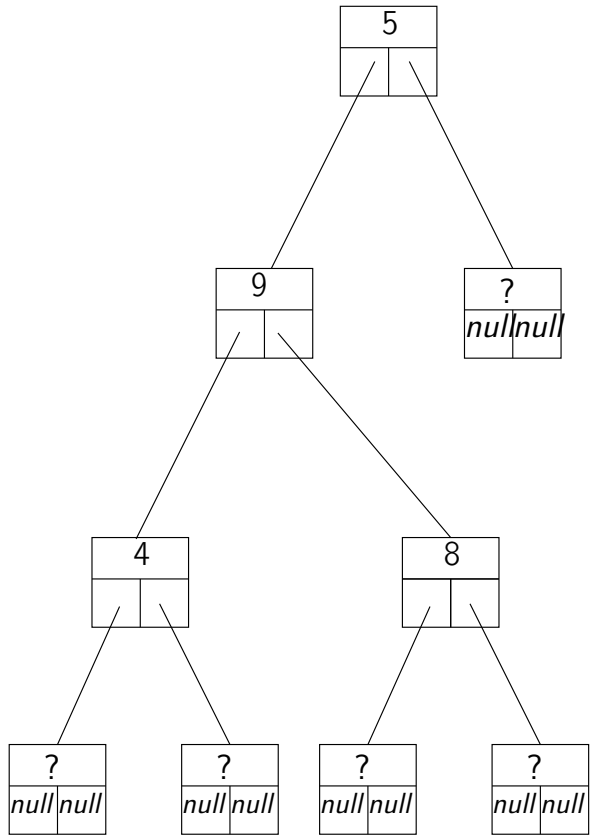
Remarque sur l'arbre vide

D'après le code précédent, notre convention est la suivante :

- l'arbre binaire vide est représentée par n'importe quel objet a de type Arbre avec $a.filsG == a.filsD == null$ (peut importe $a.val$).
- Arbre a = new Arbre() représente donc l'arbre vide
- Arbre a = null ne représente pas l'arbre vide ($a.methode()$ ..)

Exemple






```
Arbre Avide = new Arbre();
Arbre A9 = ...;
Arbre A5 = new Arbre();
A5.val=5;
A5.filsG=A9;
A5.filsD=Avide;//et pas null
```

Outline

Ex 0 : nombre de sommets (= d'entiers) dans l'arbre

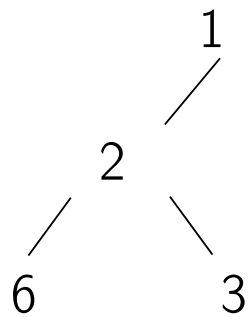
Rappel : les méthodes suivantes sont dans la classe Arbre

```
int nbSommets(){
  if(estVide()){
    return 0;
  }
  else{
   return 1+filsG.nbSommets()+filsD.nbSommets();
  }
}
```

Ex 1 : hauteur de l'arbre

Rappel : les méthodes suivantes sont dans la classe Arbre

```
int hauteur(){
 if(estVide()){
    return 0;
 }
 else{
    return
 }
}
```

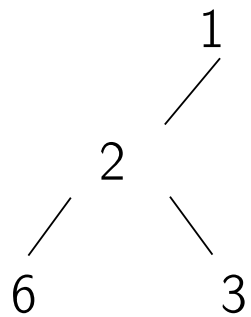


arbre de hauteur 3

Ex 1 : hauteur de l'arbre

Rappel : les méthodes suivantes sont dans la classe Arbre

```
int hauteur(){
 if(estVide()){
   return 0;
 }
 else{
  return 1+max(filsG.hauteur(),filsD.hauteur());
 }
}
```

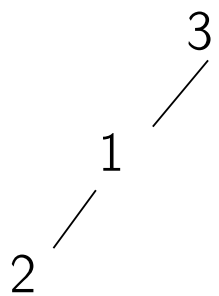


arbre de hauteur 3

Ex 2 : est filiforme

Un arbre est filiforme ssi il a au plus une feuille

```
boolean estFiliforme(){
 if(estVide())
  return true;
 else{
  bool aux1 = filsG.estFiliforme();
  bool aux2 = filsD.estFiliforme();
  return

 }
}
```

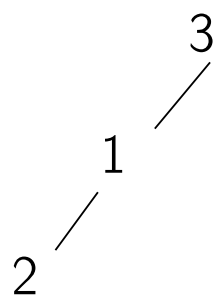


Un arbre filiforme

Ex 2 : est filiforme

Un arbre est filiforme ssi il a au plus une feuille

```
boolean estFiliforme(){
 if(estVide())
  return true;
 else{
  bool aux1 = filsG.estFiliforme();
  bool aux2 = filsD.estFiliforme();
  return ((aux1 && filsD.estVide()) || (aux2 &&
      filsG.estVide()));
 }
}
```



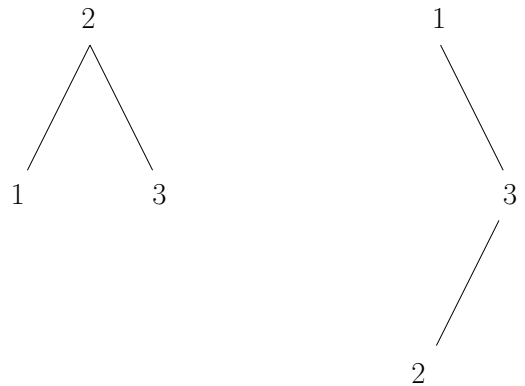
Un arbre filiforme

Ex 3 : toString()

```
String toStringNaif(){
 if(estVide())
  return "";
 else
   return filsG.toStringNaif()+"␣"+this.val+"␣"+
       filsD.toStringNaif();
}
```

Problème

- plusieurs arbres différents peuvent donner la même chaîne!
- ex : la chaîne 1 2 3 peut provenir des deux arbres suivants :

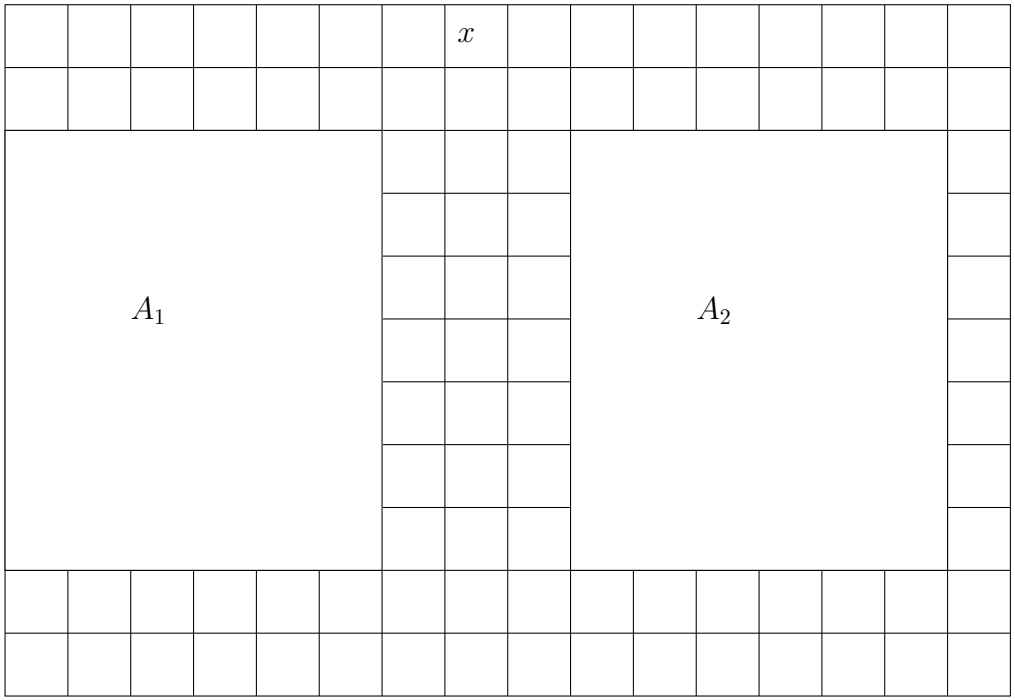


Ex 3 : toString()

Une solution

On va donc "dessiner" l'arbre dans le terminal

- pb : on ne peut pas le dessiner facilement récursivement avec la racine en haut
- solution : on tourne la tête de 90° vers la gauche!

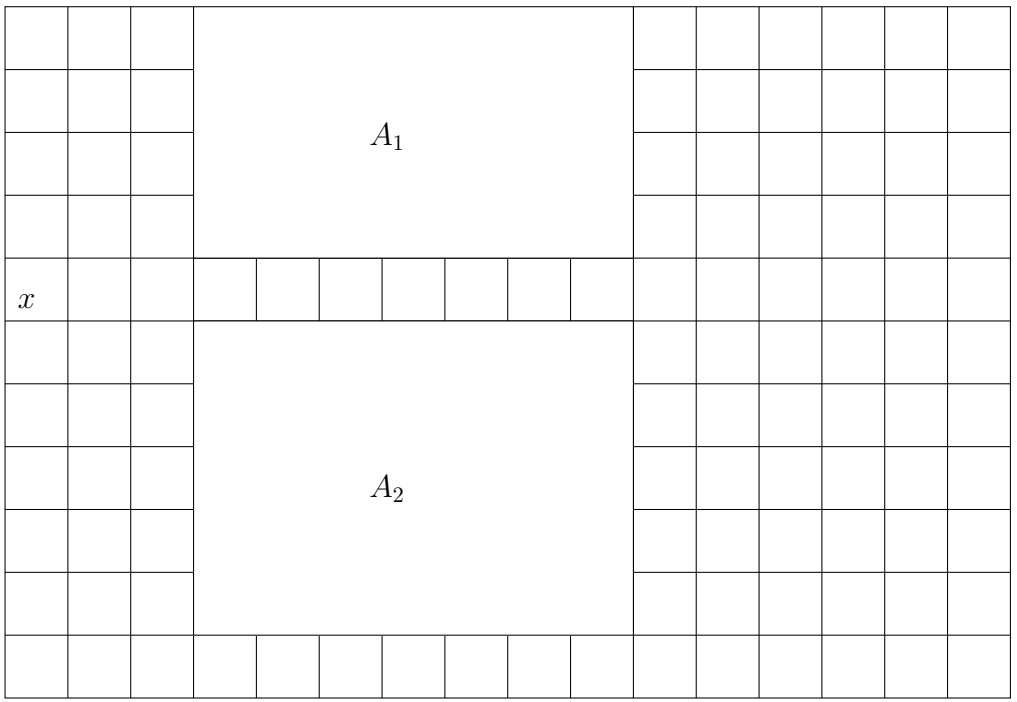


Ex 3 : toString()

Une solution

On va donc "dessiner" l'arbre dans le terminal

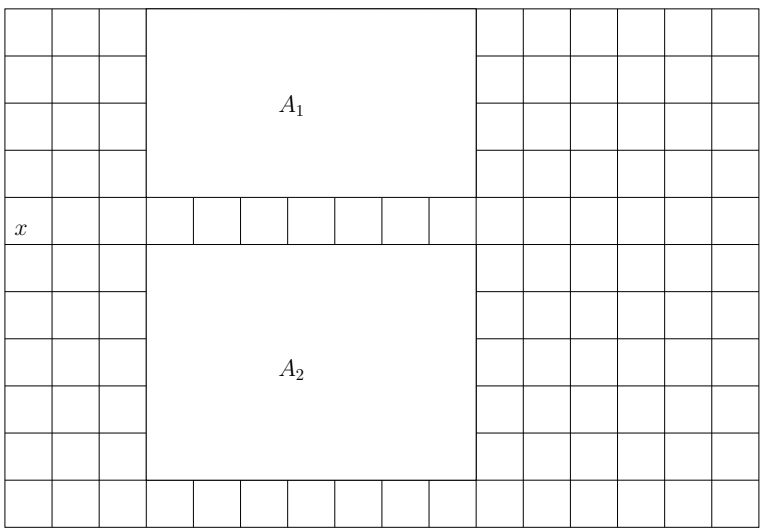
- pb : on ne peut pas le dessiner facilement récursivement avec la racine en haut
- solution : on tourne la tête de 90° vers la gauche!



Ex 3 : toString()

Deux précisions importantes

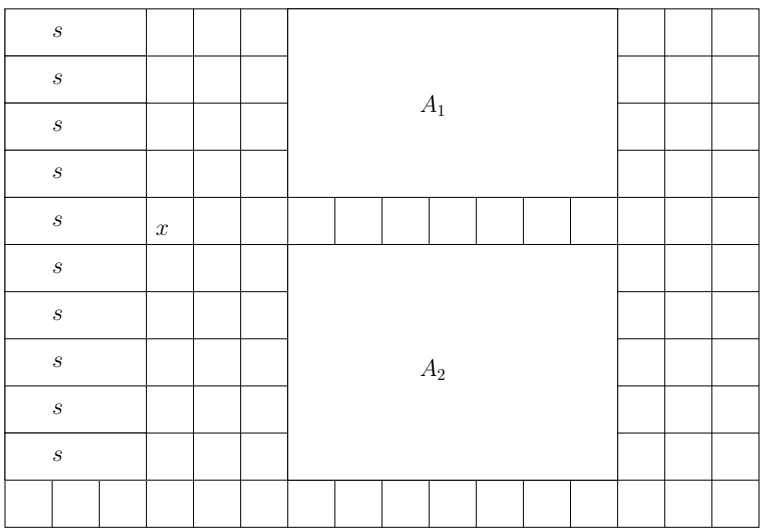
- il ne faut pas simplement demander récursivement "affiche toi", mais "affiche toi avec 3 espaces à chaque début de ligne"
  - on prend donc en paramètre une chaîne s contenant les espaces à ajouter au début de chaque ligne
  - la phrase précédente devient "affiche toi avec 3 espaces de plus dans s à chaque début de ligne"



Ex 3 : toString()

Deux précisions importantes

- il ne faut pas simplement demander récursivement "affiche toi", mais "affiche toi avec 3 espaces à chaque début de ligne"
  - on prend donc en paramètre une chaîne s contenant les espaces à ajouter au début de chaque ligne
  - la phrase précédente devient "affiche toi avec 3 espaces de plus dans s à chaque début de ligne"



Ex 3 : toString()

```
public String toStringV2aux(String s){
  //pre : aucun
  //resultat : retourne une chaine de caracteres
      permettant d'afficher this dans un
     terminal (avec l'indentation du dessin
     precedent, et en ajoutant s au debut de
     chaque ligne ecrite) et passe a la ligne
     apres chaque entier ecrit

  if( estVide ())
    return "";
  else
   return filsD.toStringV2aux (s + "␣␣␣") + s +
      val + "\n"+ filsG.toStringV2aux (s + "␣␣␣"
      );
```

Ex 3 : toString()

Conclusion

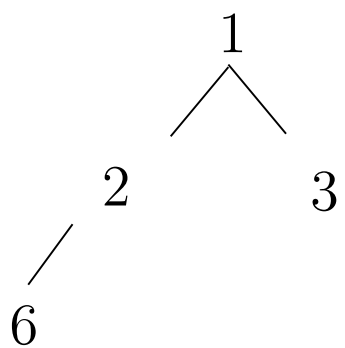
Le toString recherché est donc le suivant.

```
public String toStringV2(){
  //pre : aucun

  return toStringV2aux("");
}
```

Outline

Vide vs null

- notre classe Arbre vérifie le prérequis suivant
  - soit les 2 sous arbres sont $\neq$ null (this est alors un arbre non vide)
  - soit les 2 sous arbres sont null (this est alors un arbre vide)
- sur l'exemple ci dessous, on peut avoir l'impression que le sous arbre enraciné en 2 ne vérifie pas le prérequis..
- .. mais il le verifie bien (penser au dessin avec les () en plus) :
  - cet arbre a certes un sous arbre vide et un sous arbre non vide, MAIS
  - ses 2 sous arbres sont bien $\neq$ null

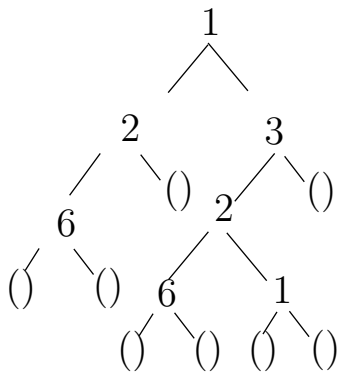


Vide vs null

Exercice

Sur l'arbre ci-dessous, repérer les racines des arbres

- ayant leur 2 sous arbres non vides
- ayant 1 sous arbre vide et 1 sous arbre non vide
- ayant leur 2 sous arbres vides
- ayant leur 2 sous arbres non null
- ~~ayant 1 sous arbre null et 1 sous arbre non null~~ (impossible)
- ayant leur 2 sous arbres null



Vide vs null

Exercice

Le code ci-dessous est il correct ?

```
public boolean rechArbre(int x){
  //pre : aucun
  //resultat : retourne vrai ssi x est dans l'
      arbre

  if( estVide ())
    return this.val == x;
  else
   return (this.val ==x) || filsD.rechArbre(x)
       || filsG.rechArbre(x);
```

Vide vs null

Exercice

Il est incorrect! Correction :

```
public boolean rechArbre(int x){
  //pre : aucun
  //resultat : retourne vrai ssi x est dans l'
      arbre

  if( estVide ())
    return false;
  else
   return (this.val ==x) || filsD.rechArbre(x)
       || filsG.rechArbre(x);
```

Vide vs null

Remarque

Le main suivant affiche donc "false", et ce n'est pas choquant (puisque a est en fait toujours l'arbre vide, même si l'on a écrit 5 dans a.val).

```
public static void main(String[] args){
   Arbre a = new Arbre();
   a.val = 5;
   print(a.rechArbre(5));
}
```

Conseil pour les algorithmes sur les arbres

- pour écrire le bloc I), la méthode dit "penser à une grande entrée x", ...
- puis, on trouve les cas qui provoquent une erreur dans I) pour compléter le code
- ces cas sont toujours un peu les mêmes : tableau (ou liste) de longueur 0, ou 1

Conseil pour les algorithmes sur les arbres

Pour le cas des arbres :

- pour écrire le bloc I), la méthode dit "penser à une grande entrée x", ... ici, un "gros arbre" est un arbre dont les 2 sous arbres sont non vides
- puis, on trouve les cas qui provoquent une erreur dans I) pour compléter le code
- ici, les cas typiques sont les suivants :
  - l'arbre est vide (ce qui entraîne en général l'ajout d'un cas de base)
  - l'arbre est non vide, mais l'un (ou les deux) sous arbres sont vides (ce qui peut entraîner soit des ajouts de cas de base, soit d'autres branches qui feront des appels récursifs différemment)

Ex sur nbFeuilles et pereFilsEgaux.