École Polytechnique

INF411

Les bases de la programmation et de l'algorithmique

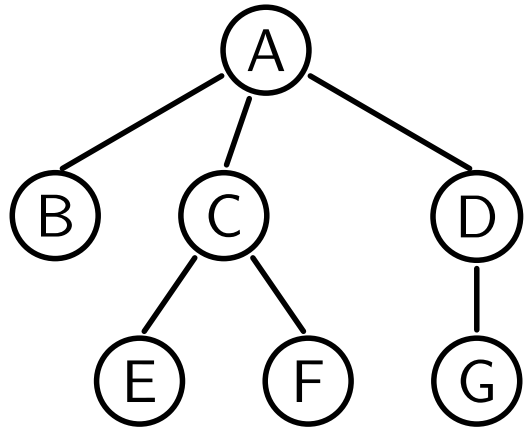
Jean-Christophe Filliâtre

arbres (1/2)

arbres

vocabulaire introduit en INF361

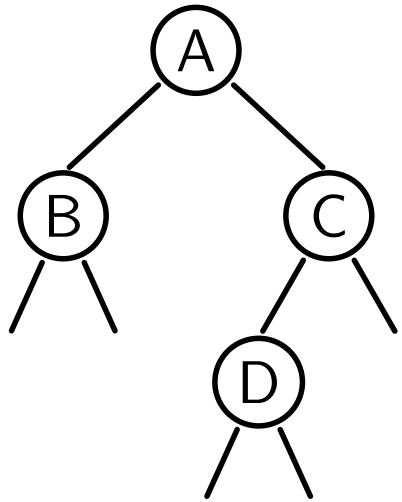
- racine (ici A)
- nœuds
- feuilles (ici B, E, F, G)
- hauteur (ici 3)



(si besoin voir le poly page 75)

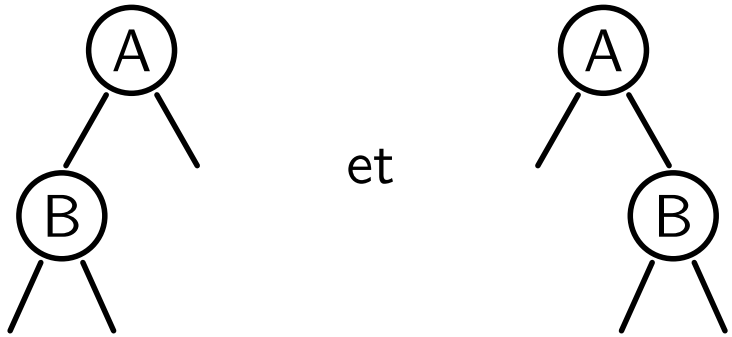
un **arbre binaire** est

- soit vide
- soit un nœud avec deux sous-arbres qui sont des arbres binaires



les deux sous-arbres sont appelés
sous-arbre gauche et **sous-arbre droit**

les deux arbres binaires



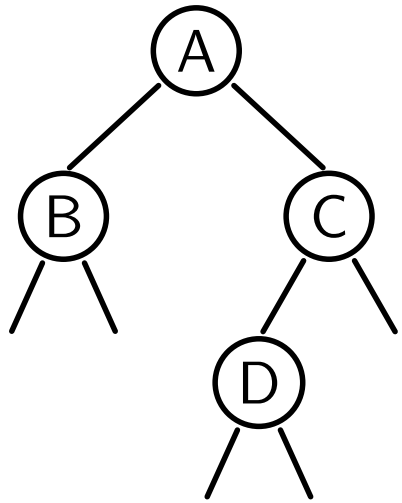
sont **différents**

(on parle d'arbre **positionnel**)

la **hauteur** d'un arbre binaire est le plus grand nombre de nœuds le long d'un chemin de la racine à une feuille

en particulier, l'arbre vide a la hauteur 0

exemple : l'arbre ci-contre a la hauteur 3



Propriété

Pour tout arbre binaire possédant N nœuds et de hauteur h, on a

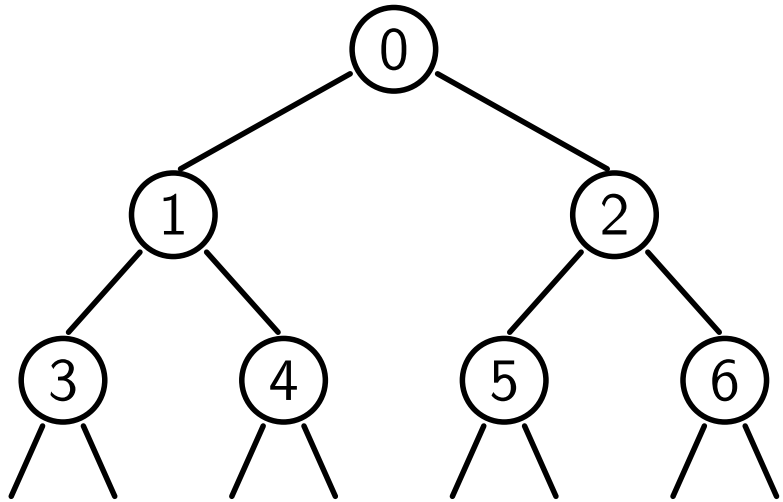
$$h \leq N \leq 2^h - 1.$$

par récurrence forte sur h

- pour $h = 0$ c'est clair
- pour $h > 0$, on a deux sous-arbres de hauteurs $h_1, h_2 < h$, l'un des deux étant de hauteur $h - 1$, donc au moins $h - 1 + 1$ nœuds et au plus

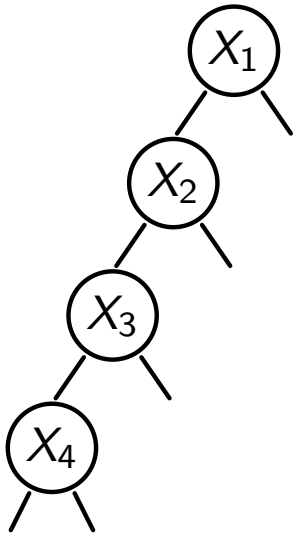
$$\begin{aligned} & 1 + 2^{h_1} - 1 + 2^{h_2} - 1 \\ \leq\ & 1 + 2^{h-1} - 1 + 2^{h-1} - 1 \\ =\ & 2^h - 1 \end{aligned}$$

atteint pour un arbre binaire **parfait**

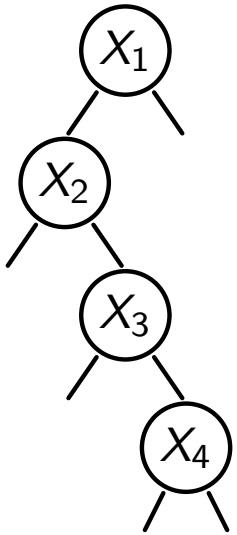


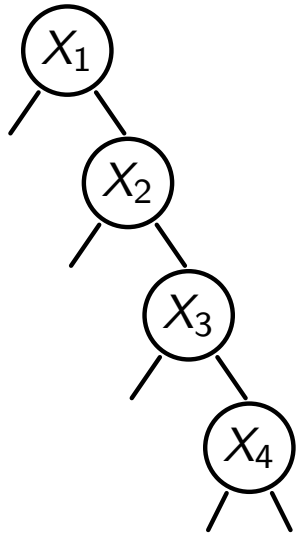
(toutes les feuilles sont à la même distance de la racine)

atteint pour un arbre **linéaire**



arbre linéaire gauche





arbre linéaire droit

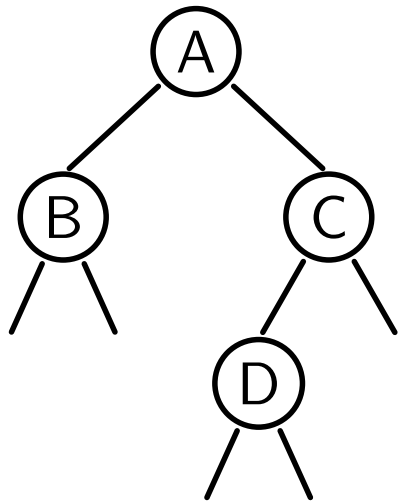
un arbre linéaire gauche ou droit est aussi appelé un **peigne**

```
class Tree {
  String value;       // noeud étiqueté ici par une chaîne
  Tree left, right;
```

et un constructeur

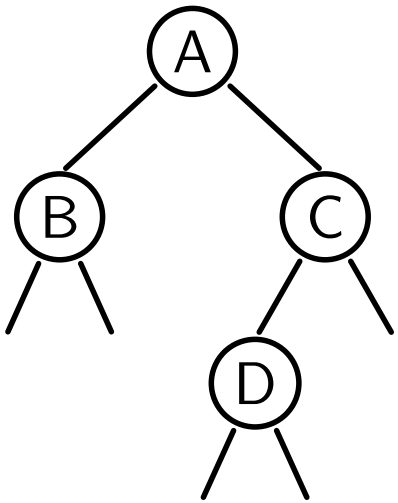
```
  Tree(Tree left, String value, Tree right) { ... }
```

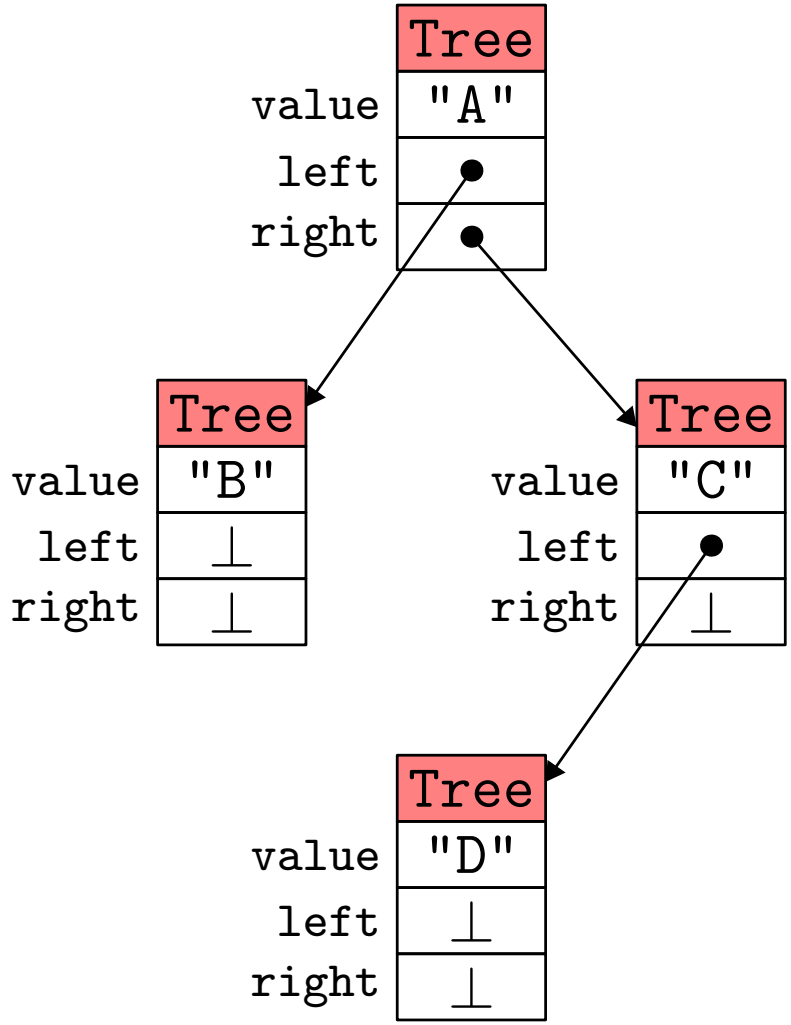
l'arbre vide est ici représenté par `null`




```
Tree t =
  new Tree(new Tree(null, "B", null),
           "A",
           new Tree (new Tree(null, "D", null),
                     "C",
                     null));
```

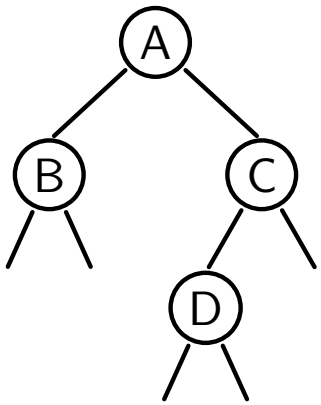
```
class Tree  {
  String value;
  Tree left, right;
}
```





parcours infixe

```
static void inorderTraversal(Tree t) {
  if (t == null) return;
  inorderTraversal(t.left);
  System.out.println(t.value); // infixe
  inorderTraversal(t.right);
}
```



B, A, D, C

(on écrit de même un parcours préfixe ou postfixe)

et si on ne veut pas afficher, mais renvoyer le résultat du parcours ?

par exemple

```
static LinkedList<String> inorder(Tree t) { ... }
```

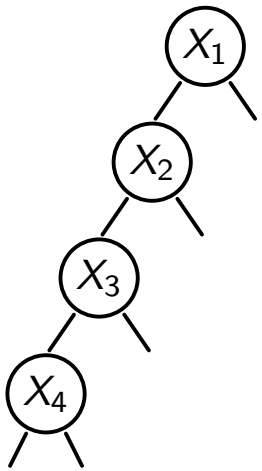
une écriture directe, récursive

```
static LinkedList<String> inorder(Tree t) {
  LinkedList<String> c = new LinkedList<String>();
  if (t == null) return c;
  c.addAll(inorder(t.left));
  c.add(t.value);
  c.addAll(inorder(t.right));
  return c;
}
```

on se sert ici de `c.addAll(l)`
qui ajoute tous les éléments de la liste `l` à la liste `c`

c'est naïf, car `addAll(l)` coûte $O($`l.size()`$)$

le coût total sur un peigne est $O(N^2)$



$$\begin{aligned} C_N &= C_{N-1} + \underbrace{N-1}_{\texttt{addAll}} + \underbrace{1}_{\texttt{add}} \\ &= 1 + 2 + \cdots + N \\ &= O(N^2) \end{aligned}$$

une méthode qui **accumule** dans une unique liste c

```
static void inorder(Tree t, LinkedList<String> c) {
  if (t == null) return;
  inorder(t.left, c);
  c.add(t.value);
  inorder(t.right, c);
}
```

il suffit de l'appeler initialement avec une liste vide

```
static LinkedList<String> inorder(Tree t) {
  LinkedList<String> c = new LinkedList<String>();
  inorder(t, c);
  return c;
}
```

note : une seule liste est manipulée

arbres binaires de recherche

pour obtenir une **structure d'ensemble** telle que

```
class Set<E> {
          Set()
  boolean contains(E x)
     void add(E x)
     void remove(E x)
}
```

on connaît déjà plusieurs solutions

	`add`	`contains`	`remove`
tableau non trié	$O(1)$	$O(N)$	$O(N)$
liste	$O(1)$	$O(N)$	$O(N)$
tableau trié	$O(N)$	$O(\log N)$	$O(N)$
table de hachage	$O(1)$	$O(1)$	$O(1)$

pourquoi chercher plus loin ?

on dispose parfois d'un **ordre total** sur les éléments

exemples

- ordre usuel sur $\mathbb{Z}$
- ordre lexicographique sur les chaînes de caractères
- événements triés par date
- etc.

dans ce cas, on peut attendre d'autres opérations sur les ensembles

```
class Set<E> {
                ...
              E getMin()
              E getMax()
              E floor(E x)
              E ceiling(E x)
            int rank(E x)
              E select(int i)
  Collection<E> range(E lo, E hi)
}
```

une structure de données où les opérations

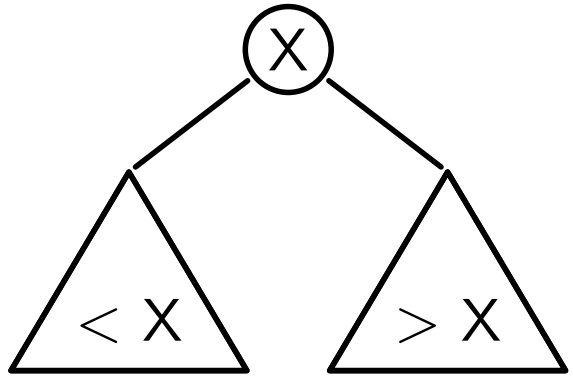
`add`, `contains`, `remove`, `getMin`, `getMax`,
`floor`, `ceiling`, `rank`, `select`

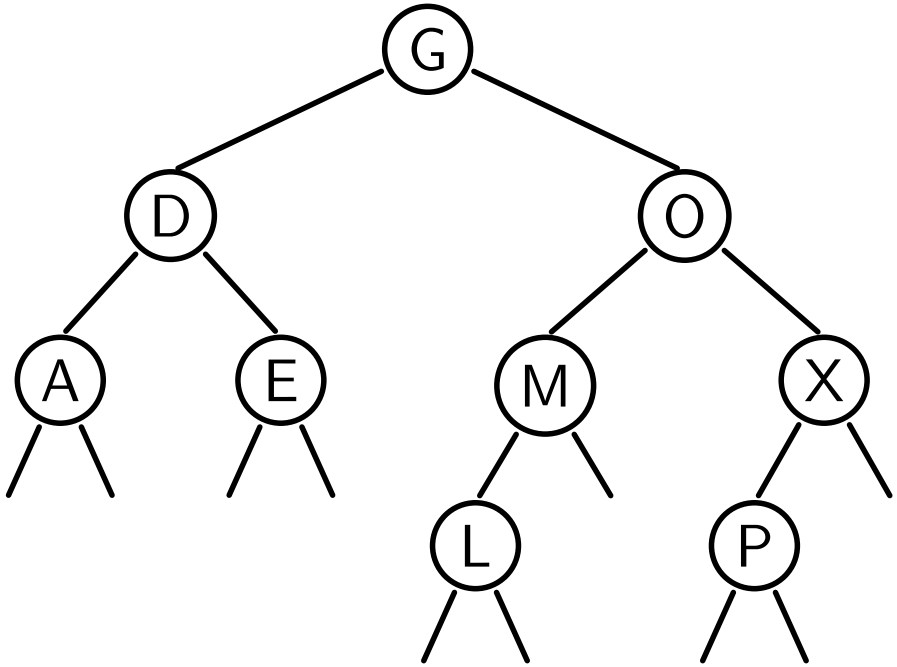
sont toutes de complexité au pire $O(\log N)$

un **arbre binaire de recherche** (en anglais *binary search tree*) est un arbre binaire où

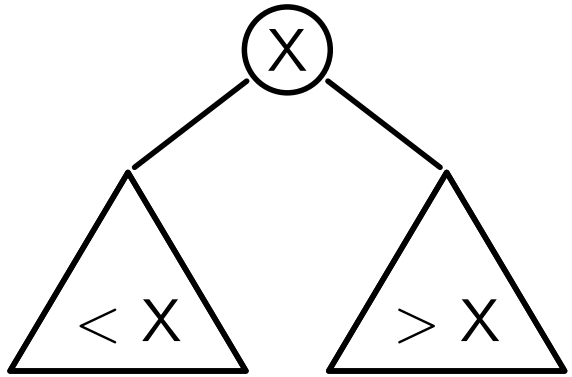
pour tout nœud X

- les éléments du sous-arbre gauche sont $< X$
- les éléments du sous-arbre droit sont $> X$



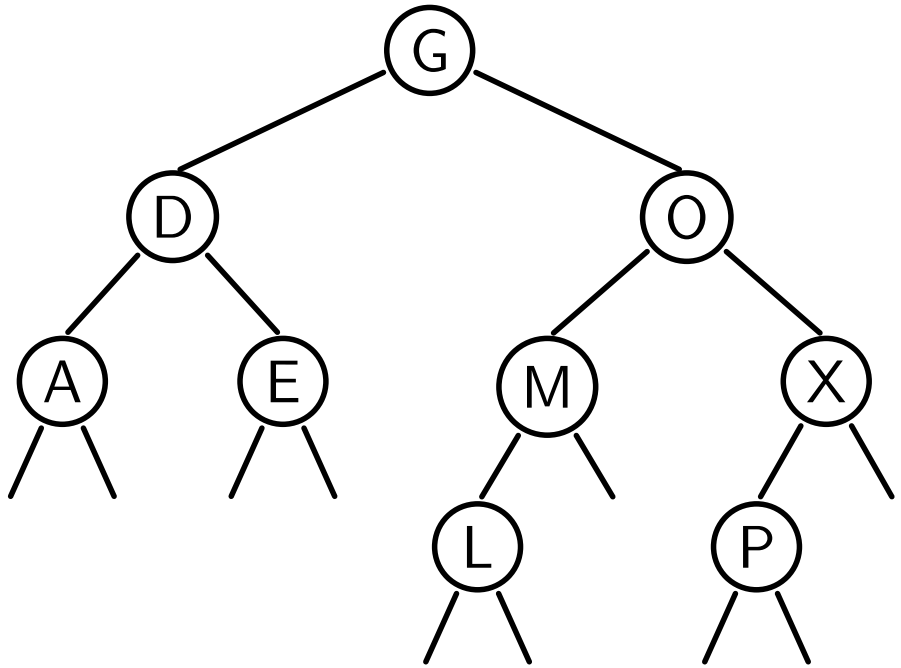

G
D
O
A
E
M
X
L
P

pour chercher / ajouter / supprimer / etc.
on ne considère qu'**un seul côté** de l'arbre



exemple : on cherche M

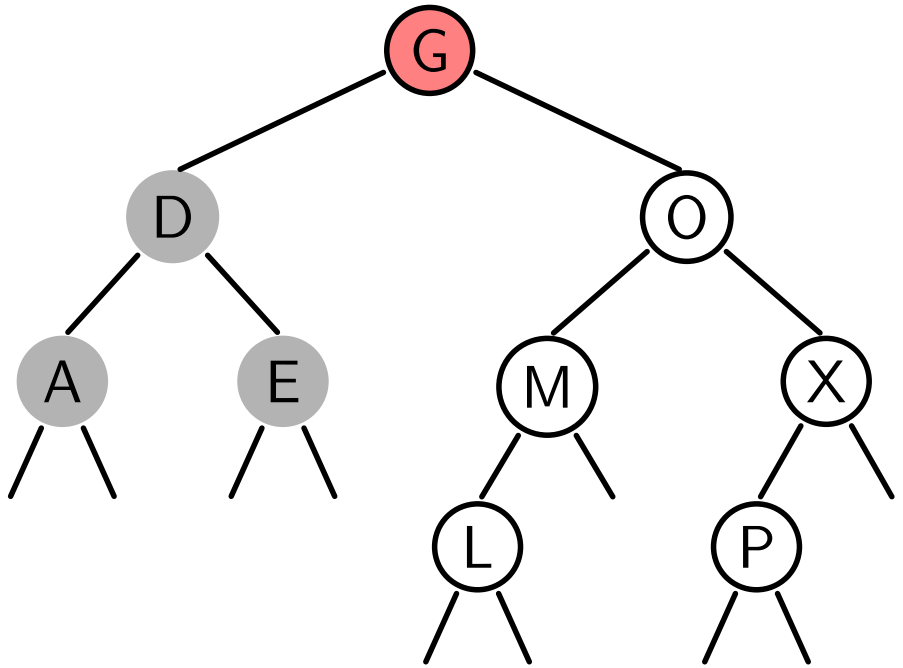
on compare M et G



exemple : on cherche M

$G < M \Rightarrow$ on va à droite

on compare M et O

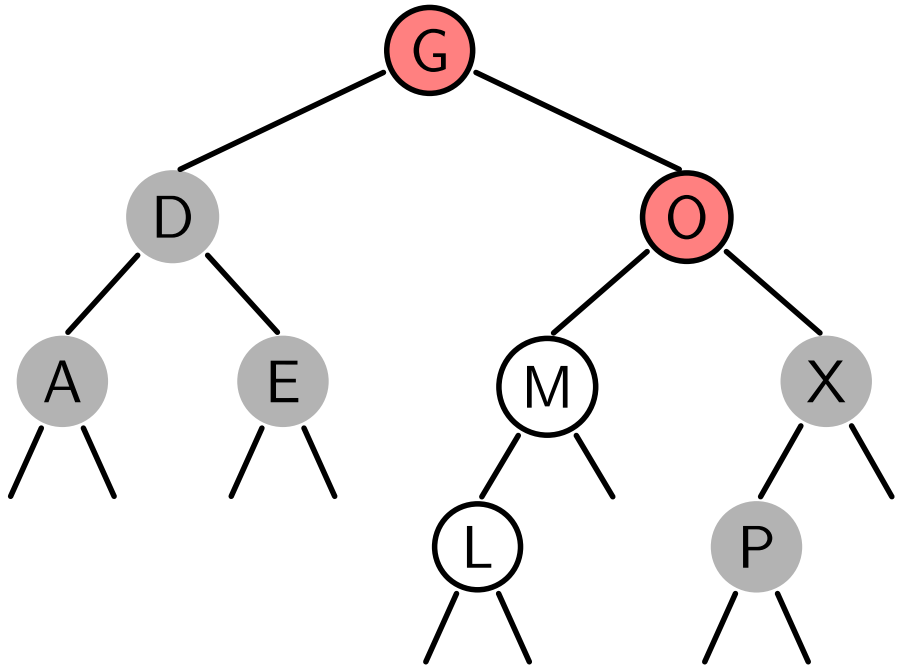


exemple : on cherche M

$G < M \Rightarrow$ on va à droite

$M < O \Rightarrow$ on va à gauche

on compare M et M

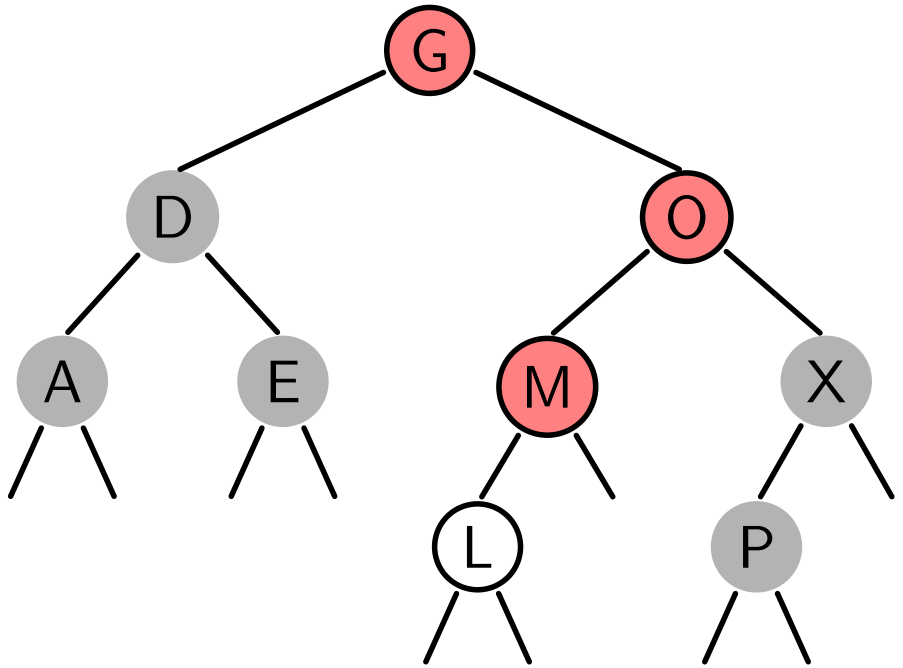


exemple : on cherche M

$G < M \Rightarrow$ on va à droite

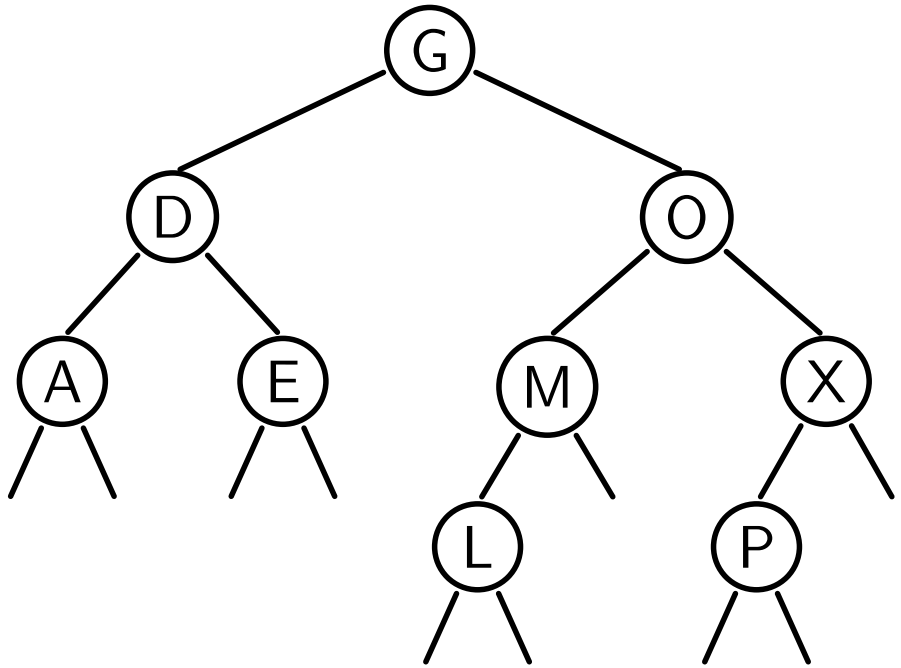
$M < O \Rightarrow$ on va à gauche

$M = M \Rightarrow$ succès



exemple : on cherche N

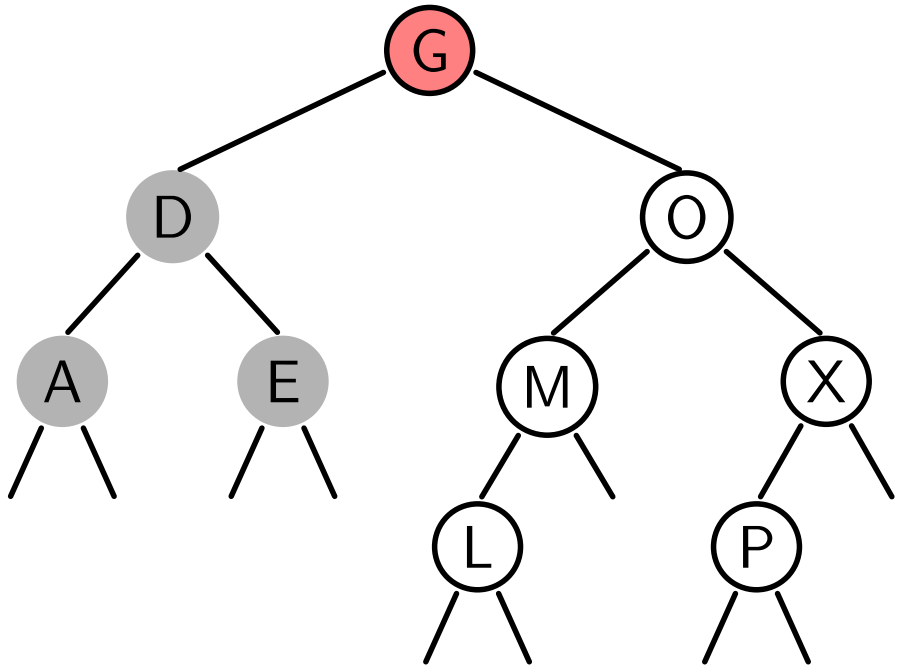
on compare N et G



exemple : on cherche N

$G < N \Rightarrow$ on va à droite

on compare N et O

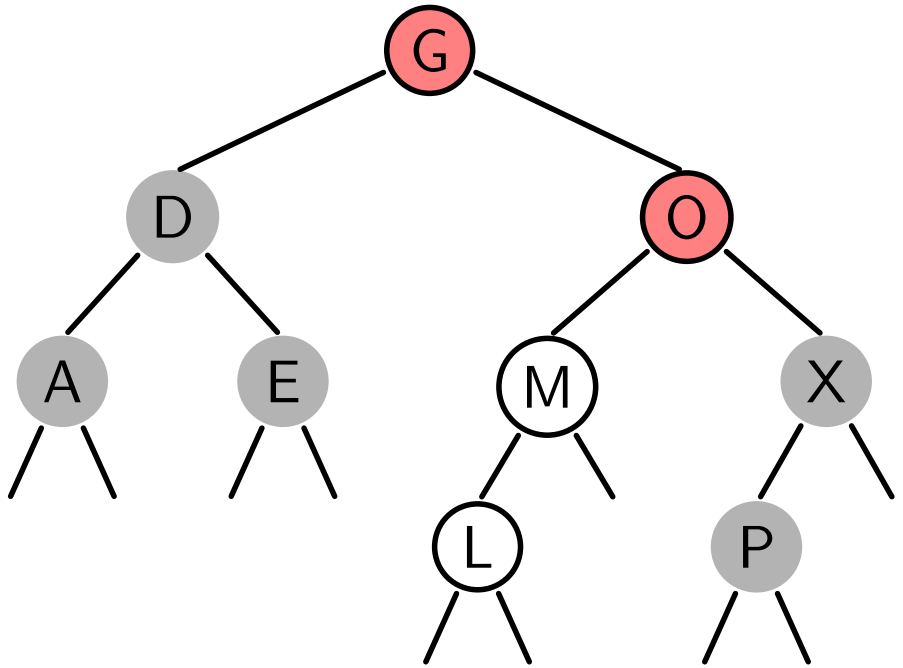


exemple : on cherche N

$G < N \Rightarrow$ on va à droite

$N < O \Rightarrow$ on va à gauche

on compare M et N

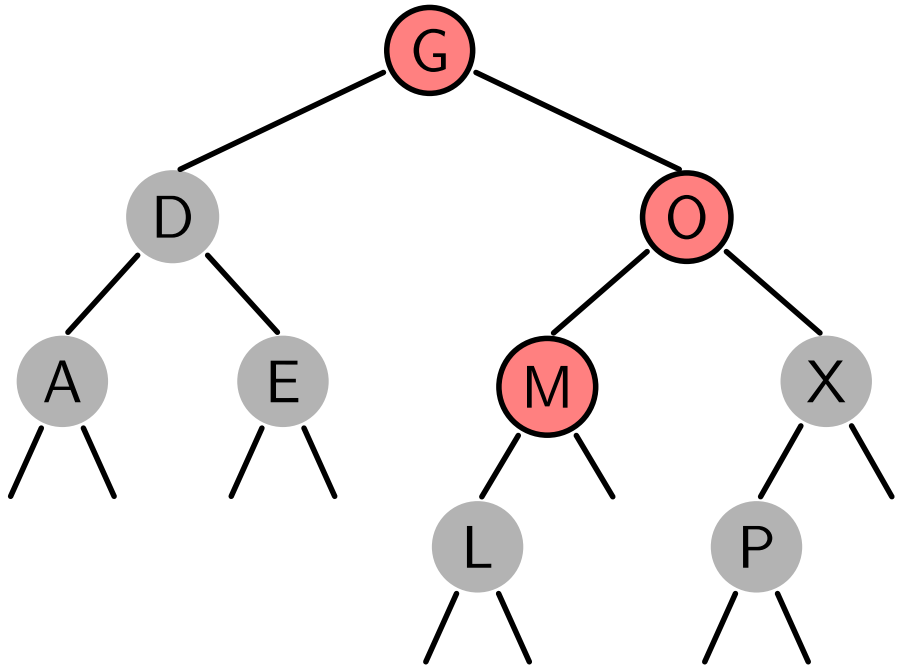


exemple : on cherche N

$G < N \Rightarrow$ on va à droite

$N < O \Rightarrow$ on va à gauche

$M < N \Rightarrow$ on va à droite



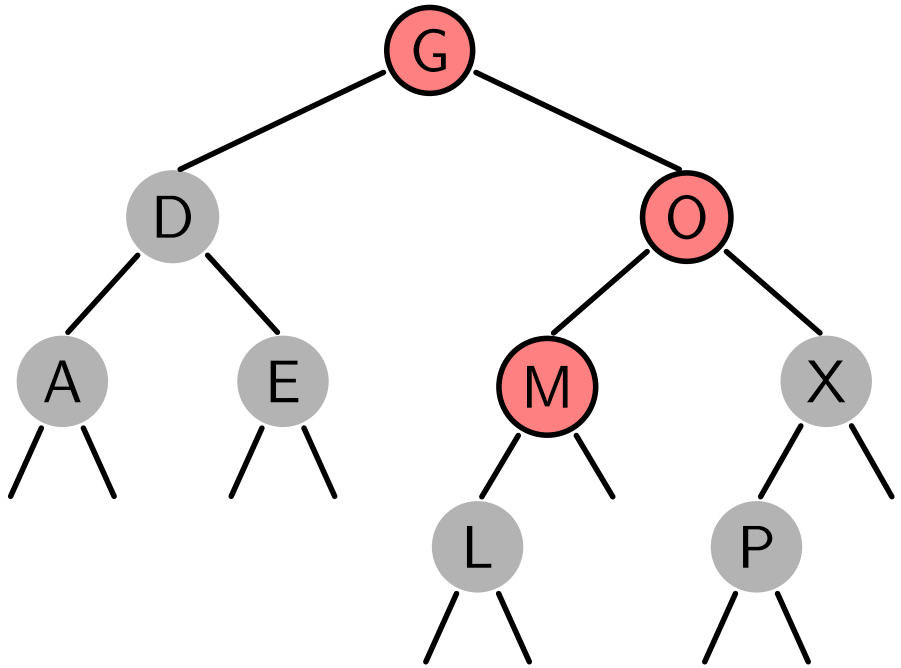
exemple : on cherche N

$G < N \Rightarrow$ on va à droite

$N < O \Rightarrow$ on va à gauche

$M < N \Rightarrow$ on va à droite

arbre vide $\Rightarrow$ échec



pour des éléments de type `String` pour l'instant

ordonnés lexicographiquement avec `compareTo`

`s1`	`s2`	`s1.compareTo(s2)`
`"bar"`	`"foo"`	< 0
`"foo"`	`"foo"`	$= 0$
`"foo"`	`"fool"`	< 0
`"small"`	`"big"`	> 0
etc.		

on va procéder en deux temps

1. on commence par une classe `BST`
   - représente un nœud de l'arbre et `null` est l'arbre vide
   - les méthodes sont **statiques**, pour faciliter la manipulation de `null`

2. on construit ensuite une classe `Set`
   - encapsule un arbre, pour cacher la représentation
   - les méthodes sont **dynamiques**, pour faciliter l'utilisation

```
class BST {
  String value;
  BST    left, right;
  ...
```

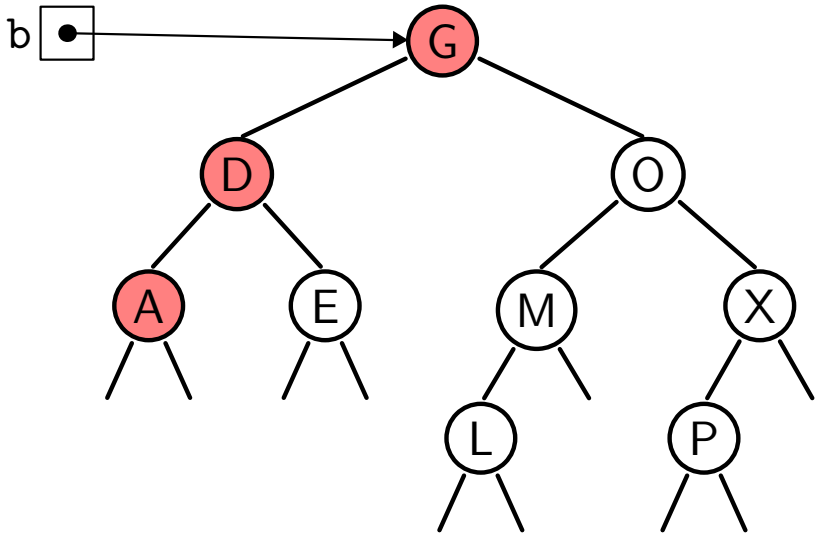
et un constructeur

```
  BST(BST left, String value, BST right) { ... }
```

plus petit élément

le plus petit élément est tout en bas à gauche

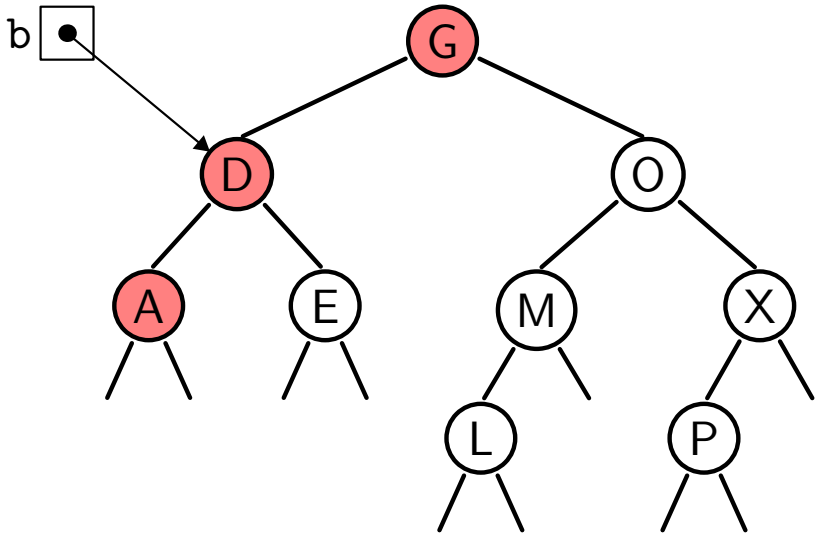
```
// suppose b != null
static String getMin(BST b) {
  while (b.left != null)
    b = b.left;
  return b.value;
}
```



note : on ne modifie que la **variable `b` locale** à `getMin`

le plus petit élément est tout en bas à gauche

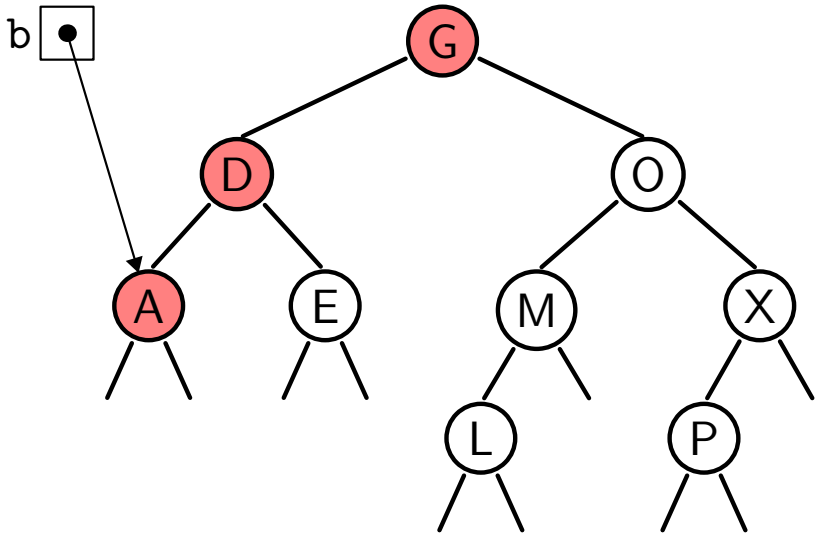
```
// suppose b != null
static String getMin(BST b) {
  while (b.left != null)
    b = b.left;
  return b.value;
}
```



note : on ne modifie que la **variable `b` locale** à `getMin`

le plus petit élément est tout en bas à gauche

```
// suppose b != null
static String getMin(BST b) {
  while (b.left != null)
    b = b.left;
  return b.value;
}
```

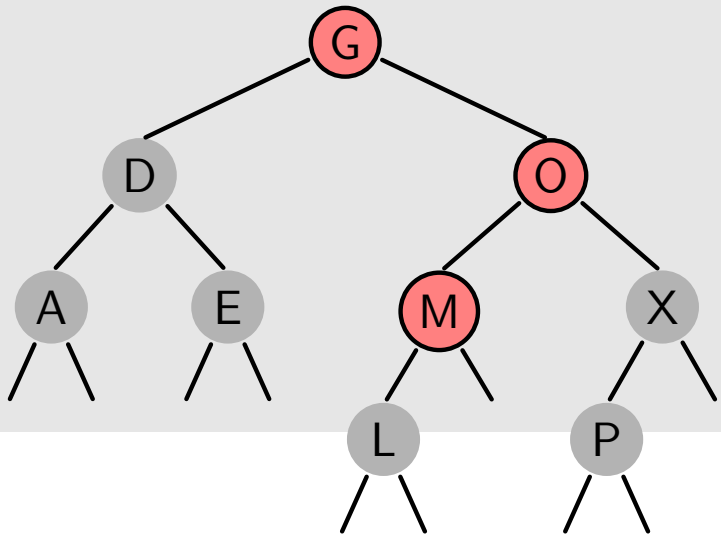


note : on ne modifie que la **variable `b` locale** à `getMin`

rechercher un élément

on descend dans l'arbre

```
static boolean contains(BST b, String x) {
  while (b != null) {
    int c = x.compareTo(b.value);
    if (c == 0) return true;
    b = (c < 0) ? b.left : b.right;
  }
  return false;
}
```

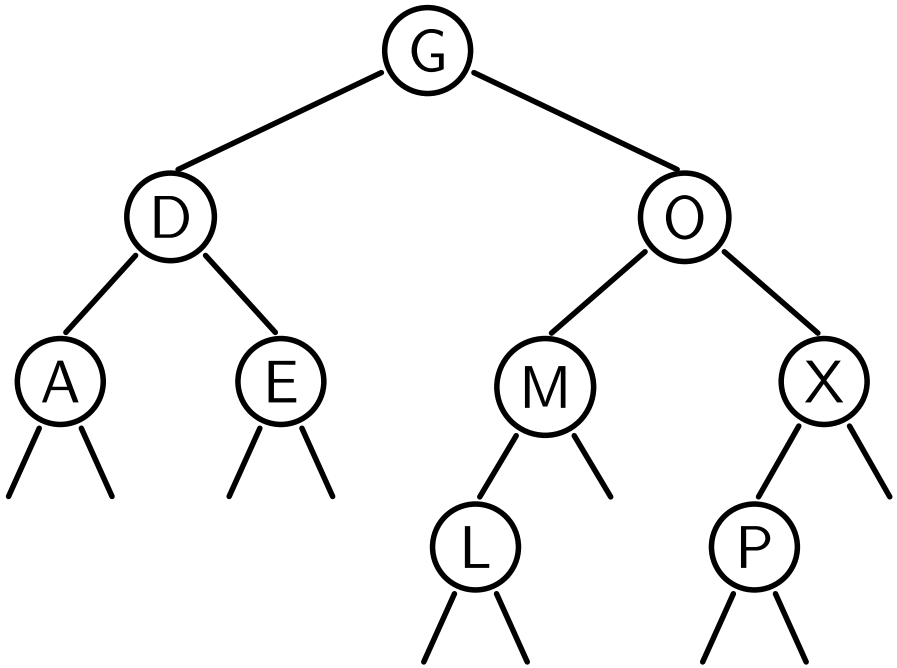


(exercice : l'écrire comme une méthode récursive)

ajouter un élément

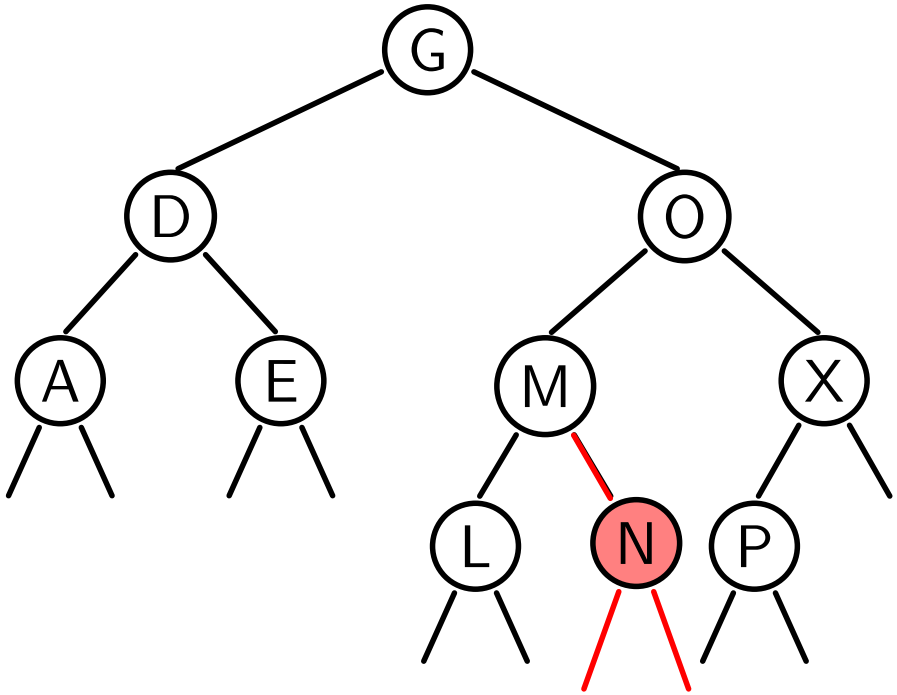
on descend dans l'arbre pour ajouter une nouvelle feuille au bon endroit

exemple : ajout de N



on descend dans l'arbre pour ajouter une nouvelle feuille au bon endroit

exemple : ajout de N



on crée **un** nouveau nœud N et on **modifie** le pointeur droit de M

comment écrire un tel code,
qui doit pouvoir insérer en particulier dans l'arbre vide `null` ?

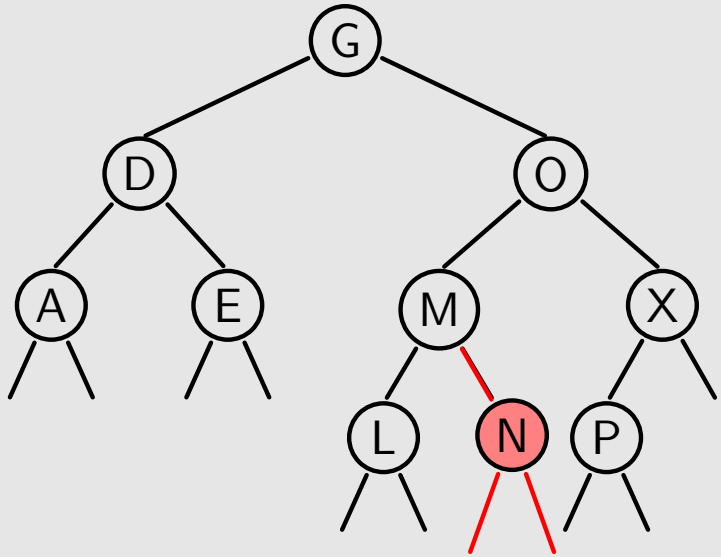
solution retenue ici : une méthode récursive

```
static BST add(BST b, String x)
```

qui renvoie la racine de l'arbre une fois l'insertion réalisée

(ce n'est pas la seule solution ; voir l'exercice 39 dans le poly)

```
static BST add(BST b, String x) {
  if (b == null)
    return new BST(null, x, null);
  int c = x.compareTo(b.value);
  if (c < 0)
    b.left = add(b.left, x);
  else if (c > 0)
    b.right = add(b.right, x);
  return b;
}
```

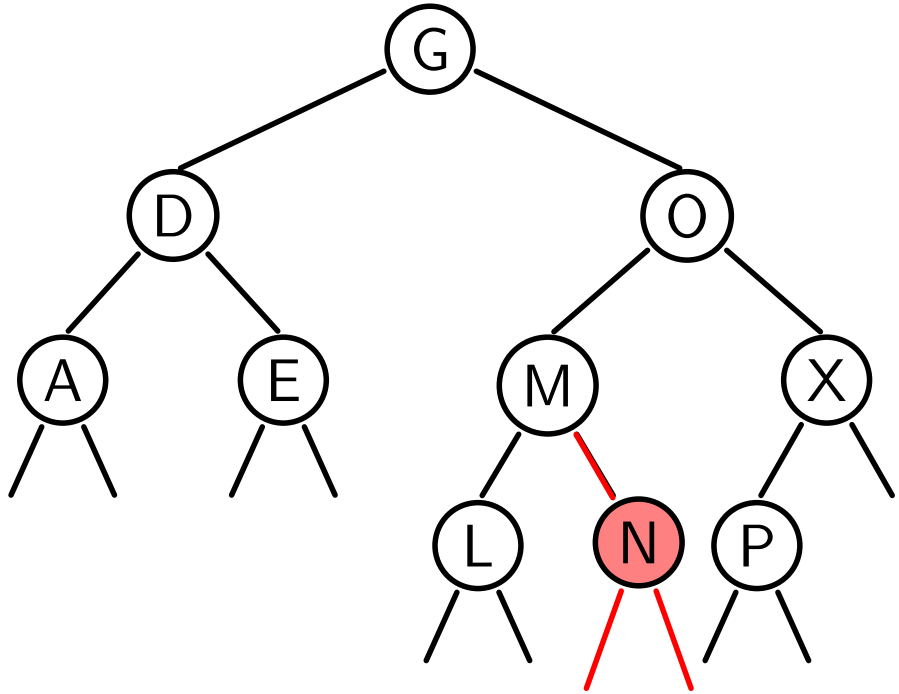


(pas de doublons : c'est un ensemble, pas un multi-ensemble)

on a fait trois affectations

- le sous-arbre droit de G
- le sous-arbre gauche de O
- le sous-arbre droit de M

seule la dernière a vraiment modifié l'arbre



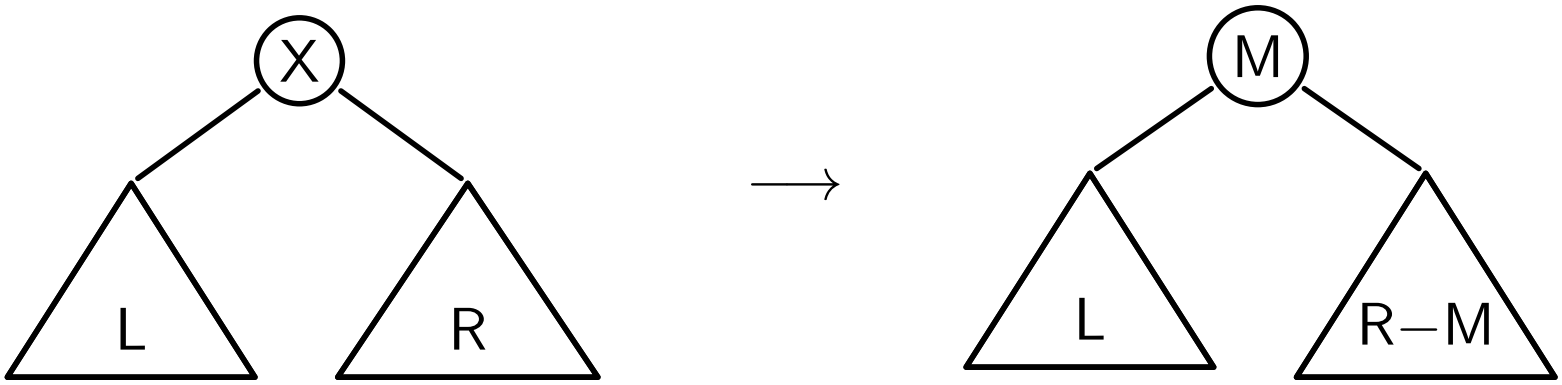
supprimer un élément

commence exactement comme l'insertion

```
static BST remove(BST b, String x) {
  if (b == null)
    return null; // rien à faire
  int c = x.compareTo(b.value);
  if (c < 0) // suppression à gauche
    b.left = remove(b.left, x);
  else if (c > 0) // suppression à droite
    b.right = remove(b.right, x);
  else { // suppression de la racine
    ...?...
```

que faire quand il faut supprimer la racine ?

remplacer X par M = min(R)



nous ramène au problème de la suppression du minimum,
plus simple

en effet

```
static BST removeMin(BST b) { // b != null
  if (b.left == null) // c'est la racine
    return b.right;
  b.left = removeMin(b.left);
  return b;
}
```

on peut maintenant terminer `remove`

```
static BST remove(BST b, String x) {
  ...
  else { // suppression de la racine
    if (b.right == null) // cas simple
      return b.left;
    b.value = getMin(b.right);
    b.right = removeMin(b.right);
  }
  return b;
}
```

encapsulation

comme pour les listes (cf amphi 1), on termine en **encapsulant** l'arbre binaire de recherche dans une classe

```
class Set {
  private BST root;

  Set() { this.root = null; }

  boolean isEmpty() {
    return this.root == null;
  }

  boolean contains(String s) {
    return BST.contains(this.root, s);
  }
  ...
```

en particulier, les méthodes add et remove ne renvoient plus rien

```
class Set {
  ...

  void add(String s) {
    this.root = BST.add(this.root, s);
  }

  void remove(String s) {
    this.root = BST.remove(this.root, s);
  }
}
```

l'utilisation n'est pas différente de celle de `HashSet`

```
Set s = new Set();
...
s.add("foo");
...
if (s.isEmpty()) // et non s == null
  ...
```

en particulier, l'utilisateur ignore la représentation interne
(*i.e.* l'existence de la classe `BST` et le rôle joué par `null`)

complexité

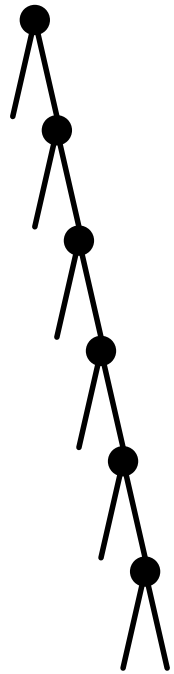
on peut décompter les comparaisons effectuées

cas de figure considéré

1. insertion successive de N éléments dans un arbre initialement vide
2. recherche dans cet arbre

malchance

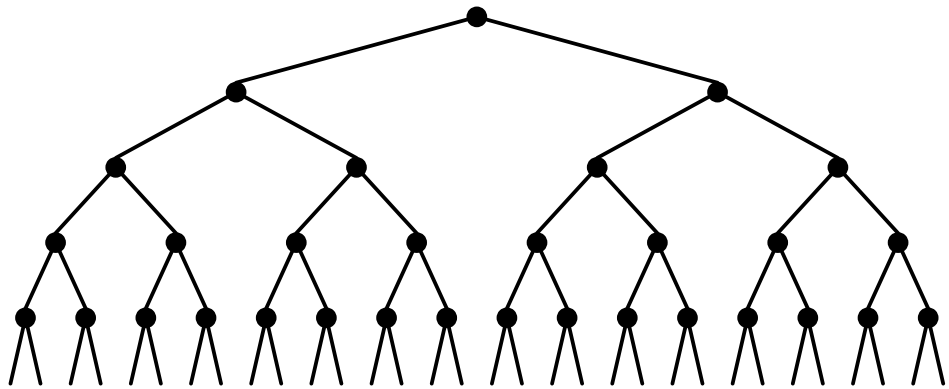
si on est malchanceux
l'arbre peut être un peigne
(éléments insérés dans l'ordre)



k-ième `add` en $O(k)$
construction totale en $O(N^2)$
`contains` en $O(N)$

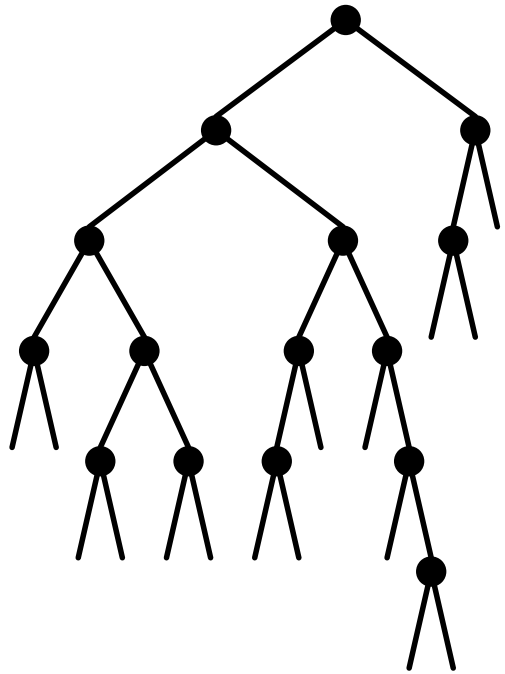
si on est (très) chanceux
l'arbre peut être parfait

chaque `add` en $O(\log N)$
construction totale en $O(N \log N)$
`contains` en $O(\log N)$



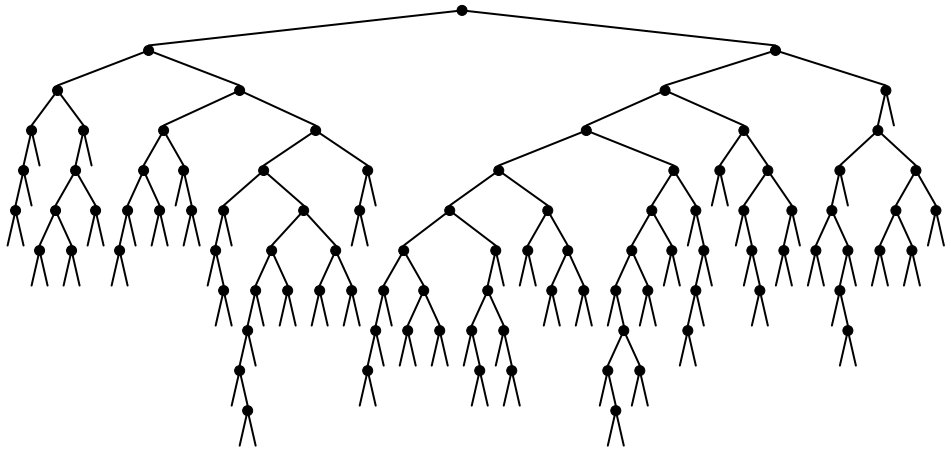
c'est plutôt cela

quel coût ?



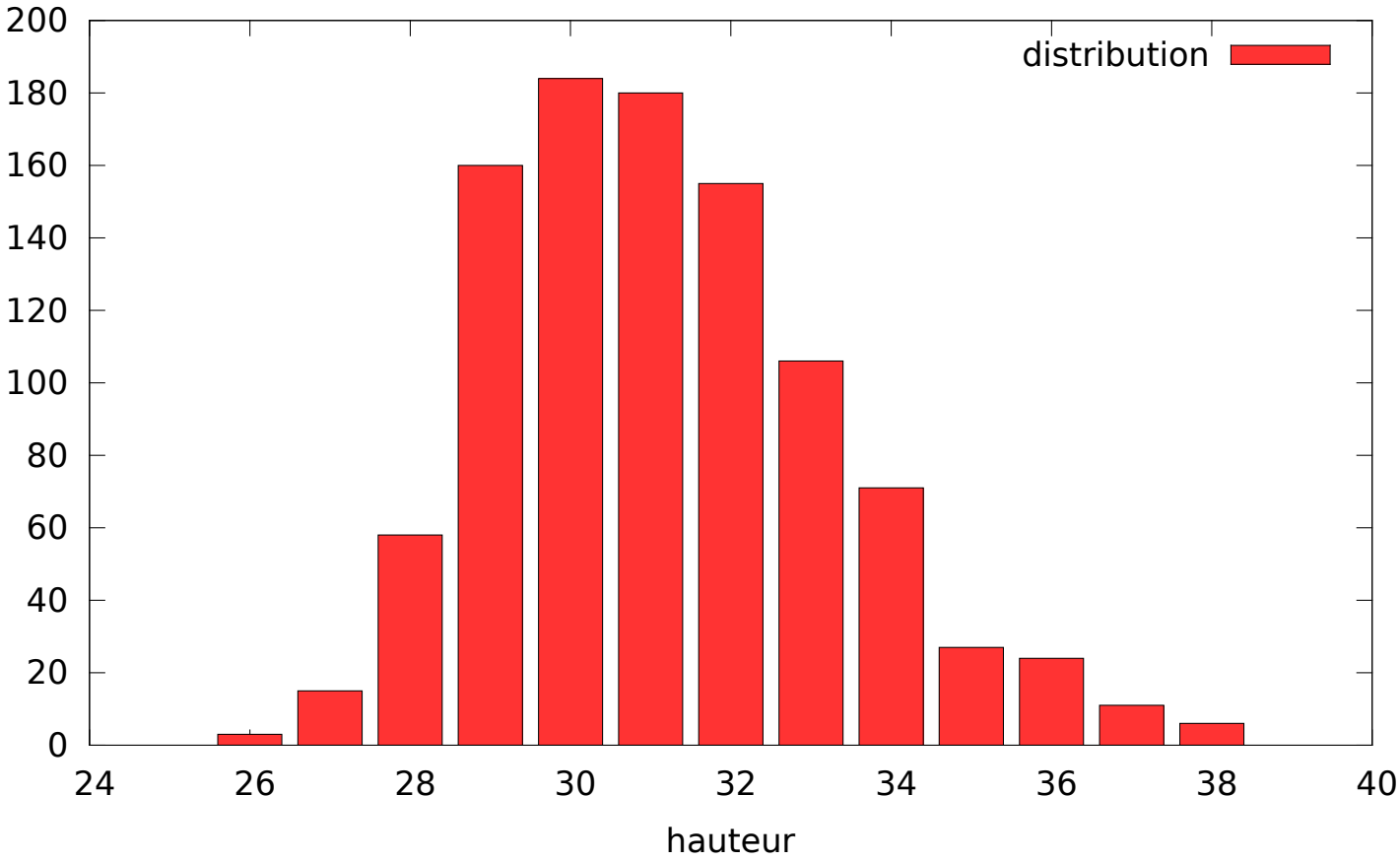
N valeurs aléatoires

on observe la hauteur obtenue



$N = 100$, hauteur 11

résultats pour $N = 10\,000$ (répété 1000 fois)



on peut montrer que

l'insertion et la recherche dans un ABR construit avec N valeurs aléatoires coûte **en moyenne** $2 \ln N$ comparaisons

éviter le pire cas

les valeurs ne suivent pas toujours une distribution aléatoire

comment éviter les cas pathologiques ?

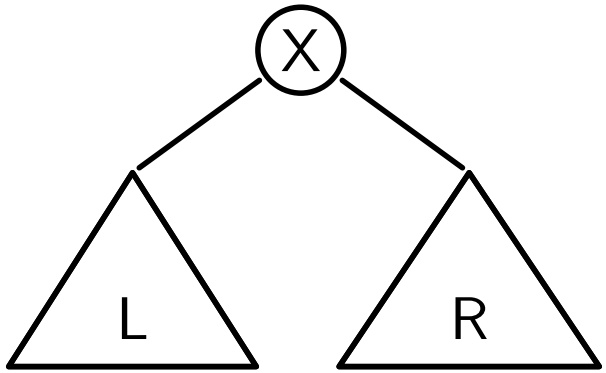
arbres binaires de recherche **équilibrés**

garantir une hauteur **logarithmique**

un arbre contenant N valeurs a une hauteur $\leq C \cdot \log N$

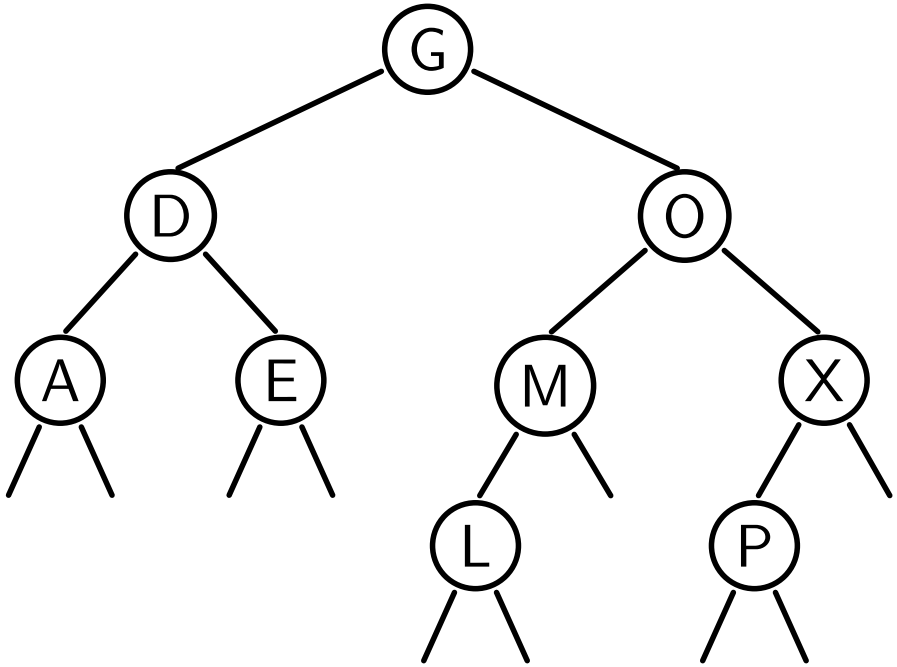
solution présentée ici : les arbres **AVL**
(de leurs auteurs **A**delson-**V**elsky et **L**andis)

un arbre binaire de recherche est un **AVL** si,
pour tout nœud

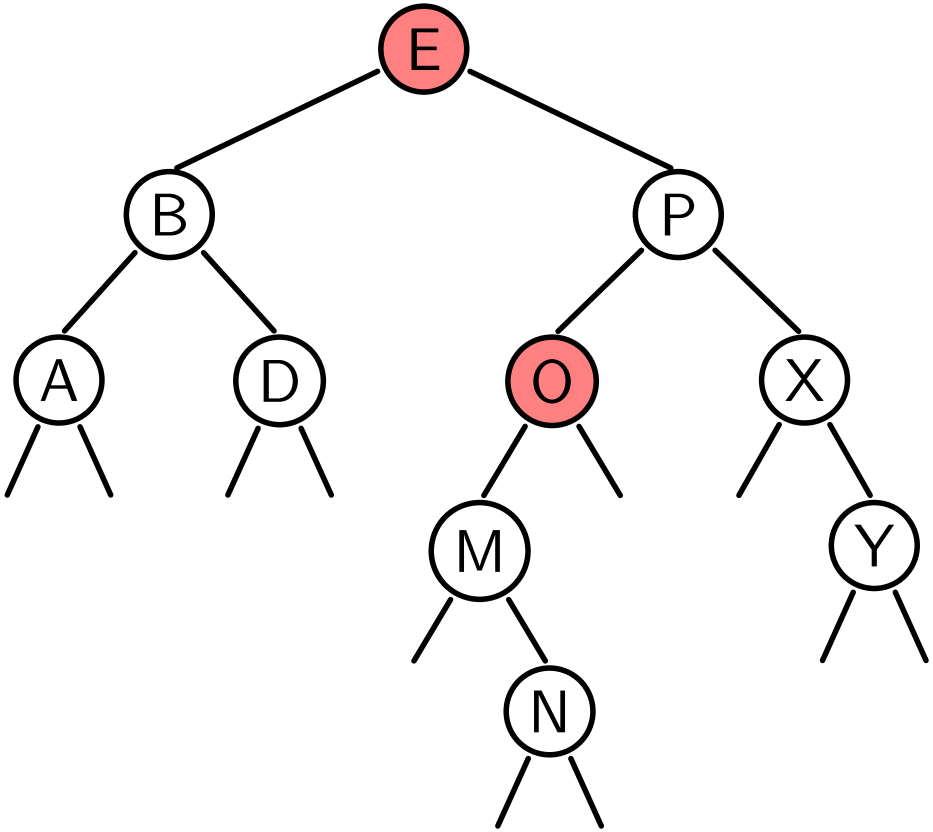


on a

$$|\text{hauteur}(L) - \text{hauteur}(R)| \leq 1$$


G
D
O
A
E
M
X
L
P

contre-exemple



quelle est la hauteur maximale d'un AVL contenant N nœuds ?

posons le problème inversement :
quel est le nombre minimal N_h de nœuds dans un AVL de hauteur h ?

N_h = nombre minimal de nœuds dans un AVL de hauteur h

$$\begin{cases} N_1 &= 1 \\ N_2 &= 2 \\ N_h &= 1 + N_{h-1} + N_{h-2} \end{cases}$$

en ajoutant 1, on reconnaît la suite de Fibonacci (0,1,1,2,3,5,8, . . .)

$$\begin{cases} N_1 + 1 &= 2 \\ N_2 + 1 &= 3 \\ N_h + 1 &= (N_{h-1} + 1) + (N_{h-2} + 1) \end{cases}$$

c'est-à-dire

$$N_h + 1 = F_{h+2}$$

or on a l'inégalité $F_i > \phi^i/\sqrt{5} - 1$ avec $\phi = \frac{1+\sqrt{5}}{2}$ (nombre d'or)

donc

$$N \geq N_h = F_{h+2} - 1 > \phi^{h+2}/\sqrt{5} - 2$$

d'où finalement

$$\begin{aligned} h &< \frac{1}{\log_2 \phi} \log_2(N+2) + \frac{\log_2 \sqrt{5}}{\log_2 \phi} - 2 \\ &\approx 1{,}44 \log_2(N+2) - 0{,}328 \end{aligned}$$

la hauteur d'un AVL est bien **logarithmique**

mise en œuvre en Java

la classe `BST` devient une classe `AVL`
avec un champ supplémentaire contenant la hauteur

```
class AVL {
  String value;
  AVL    left, right;
  int    height;
```

et une méthode pour la renvoyer

```
  static int height(AVL a) {
    return (a == null) ? 0 : a.height;
  }
```

le constructeur calcule la hauteur

```
AVL(AVL left, String value, AVL right) {
  this.left = left;
  this.value = value;
  this.right = right;
  this.height = 1 + Math.max(height(left), height(right));
}
```

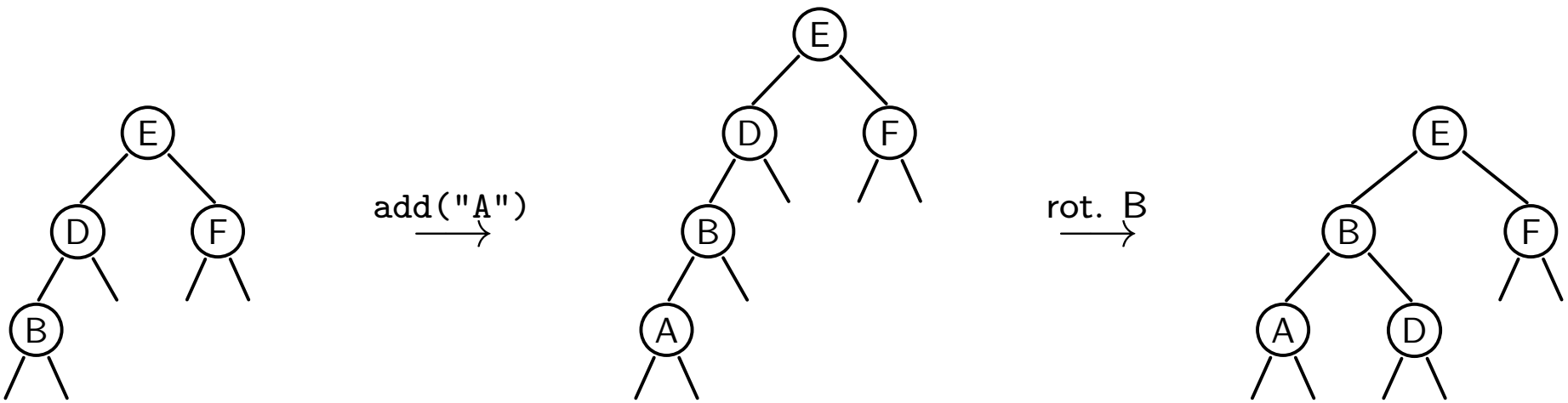
(pas de circularité : `left` et `right` sont déjà construits)

un AVL est un arbre binaire de recherche

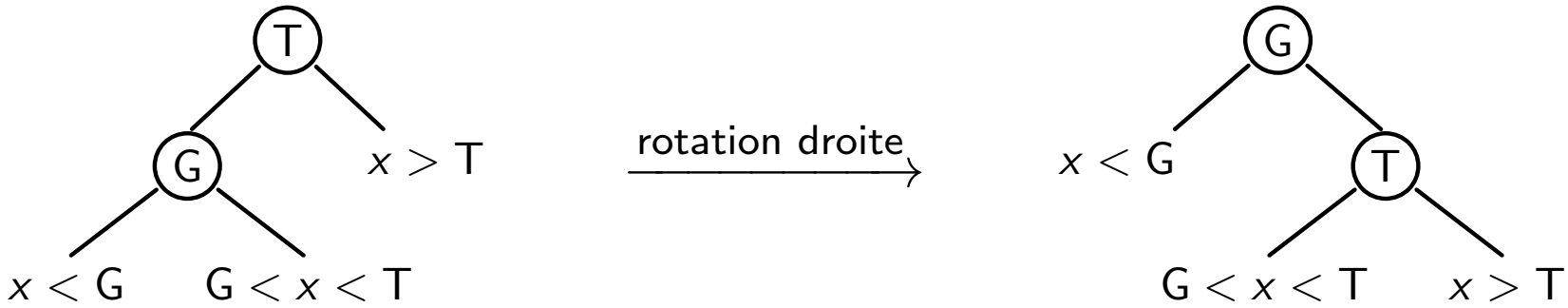
$\Rightarrow$ certaines méthodes sont inchangées

```
static boolean contains(AVL a, String s) { ... }
static String  getMin  (AVL a          ) { ... }
```

pour les méthodes qui construisent des arbres, en revanche,
il va falloir **équilibrer**

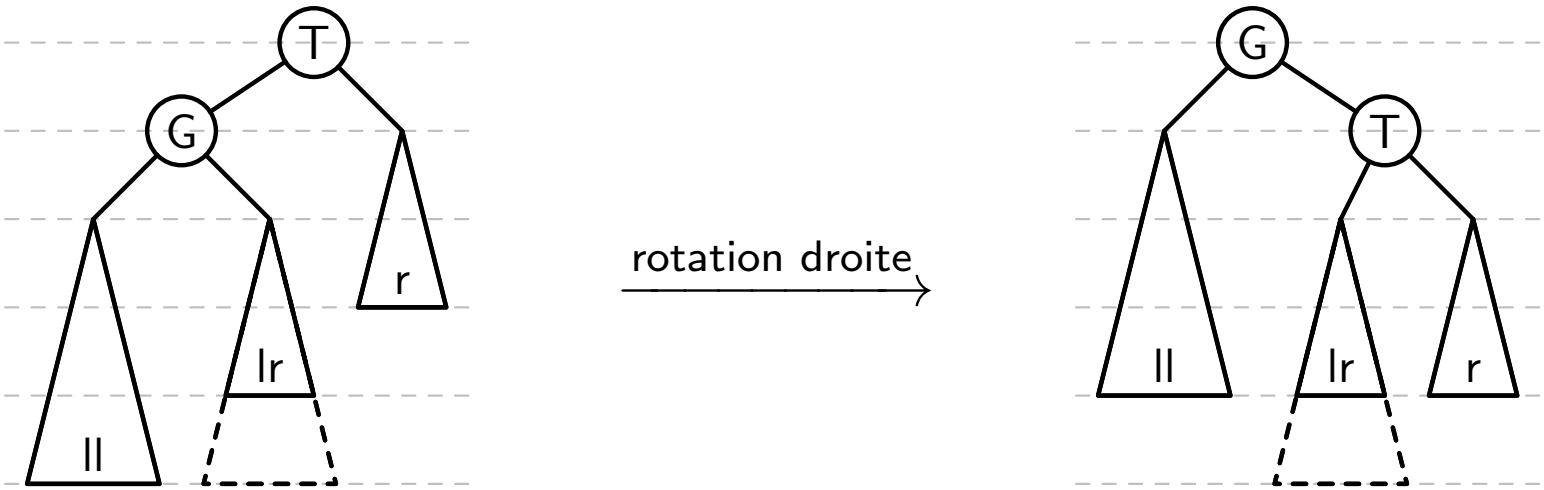

E
D
F
B
add("A")
E
D
F
B
A
rot. B
E
B
F
A
D

une **rotation à droite** conserve la propriété d'ABR

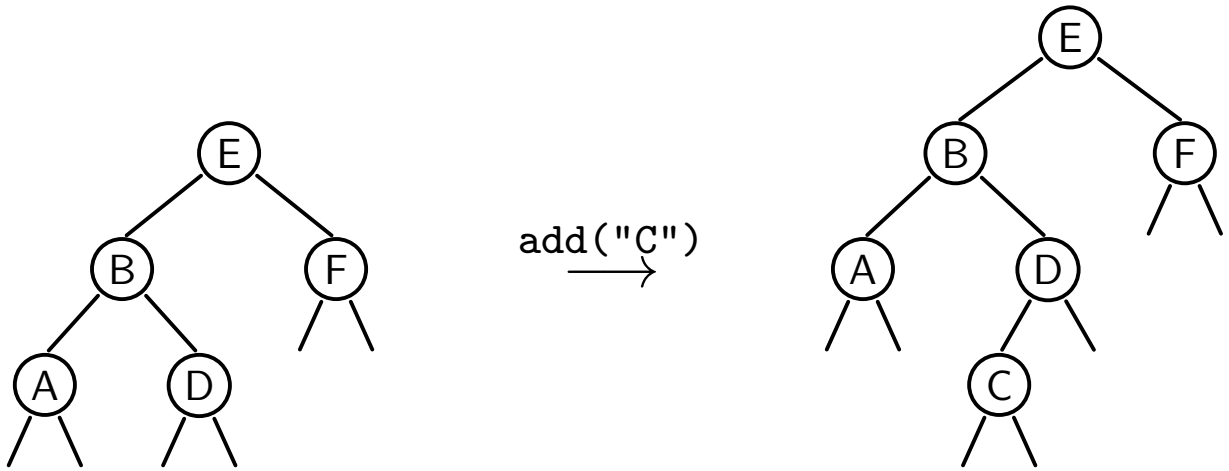


(de même pour une **rotation à gauche**)

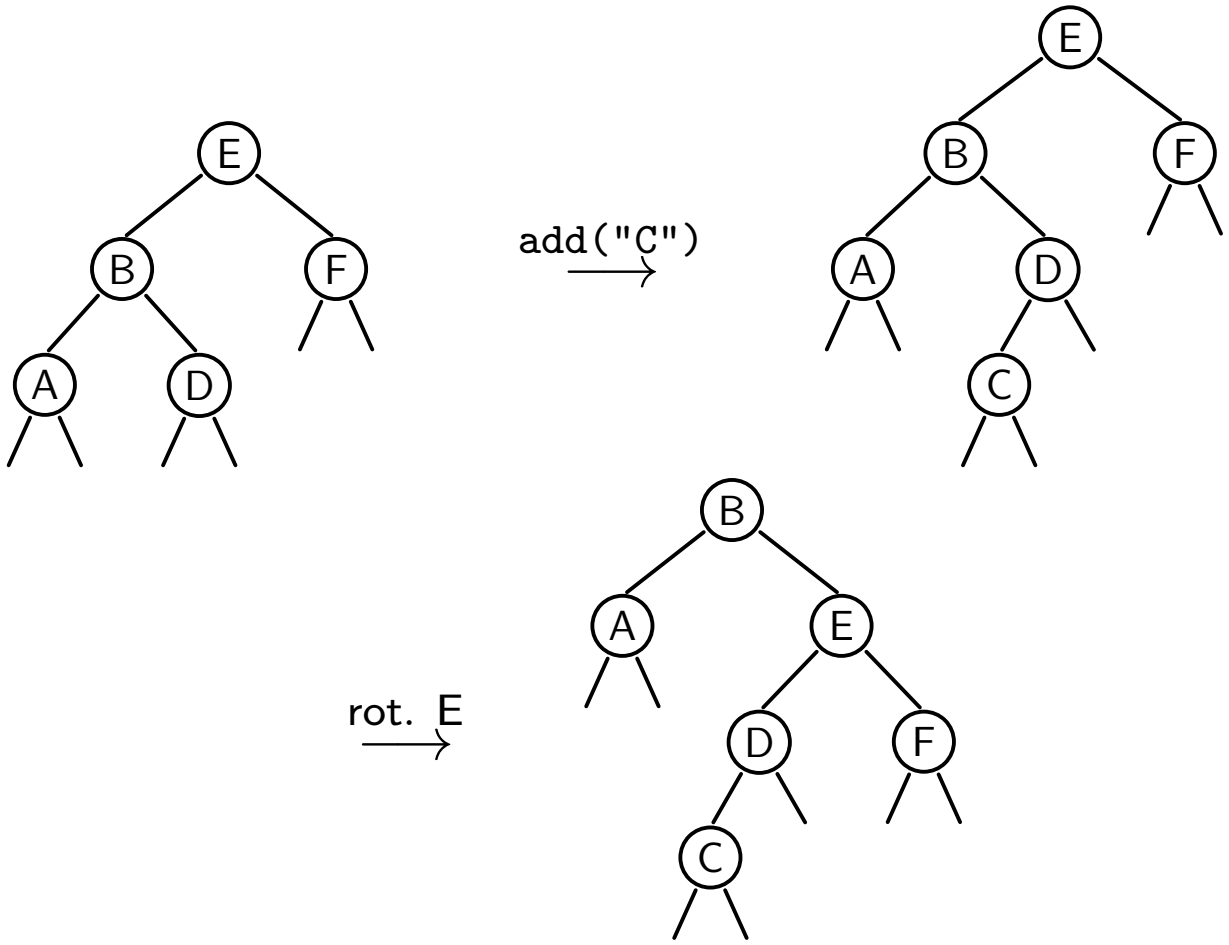
la rotation permet de rétablir la propriété d'AVL
quand le déséquilibre provient du sous-arbre gauche du sous-arbre gauche



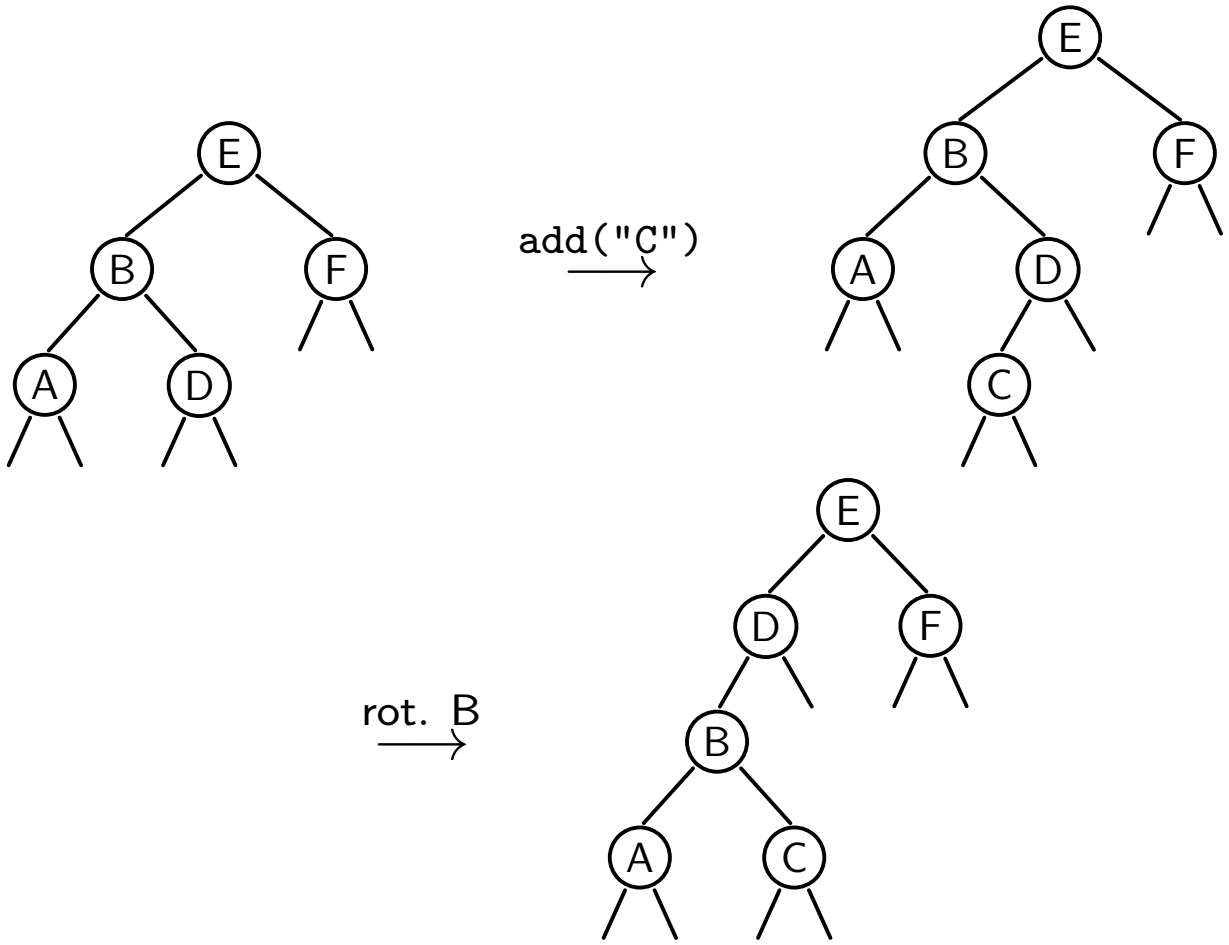
une seule rotation ne suffit pas toujours



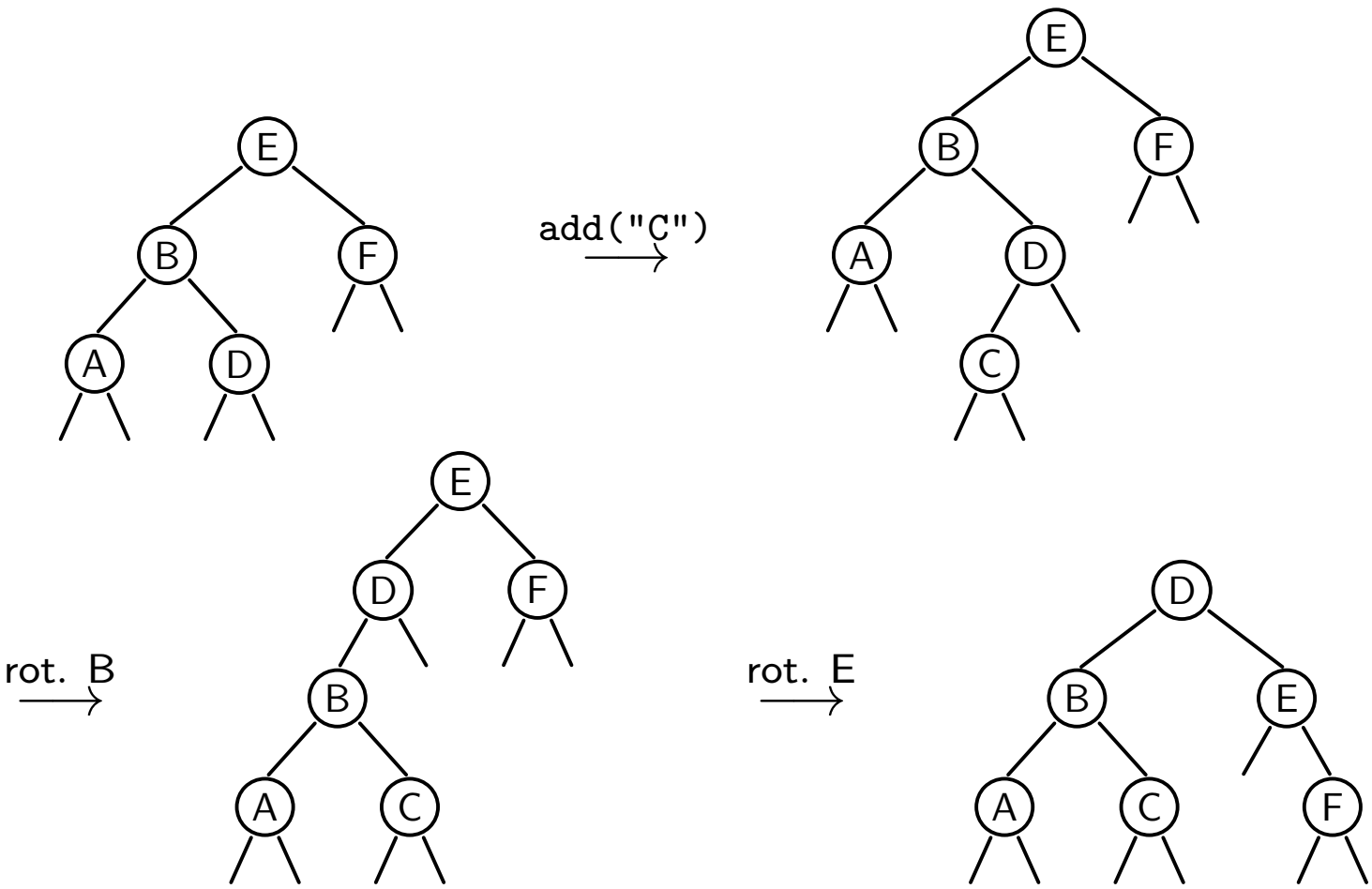
une seule rotation ne suffit pas toujours



une seule rotation ne suffit pas toujours



une seule rotation ne suffit pas toujours



la rotation gauche-droite permet de rétablir la propriété d'AVL
quand le déséquilibre provient du sous-arbre droit du sous-arbre gauche

de manière symétrique,
on peut rétablir l'équilibre en cas d'insertion à droite

(soit par une rotation gauche, soit par une rotation droite-gauche)

écrivons une méthode

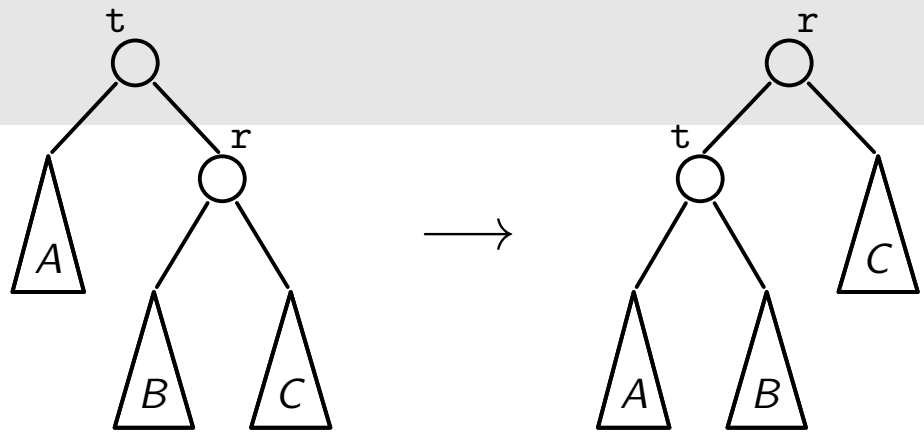
```
static AVL balance(AVL t) {
  ...
}
```

qui se comporte comme l'identité,
en rétablissant l'équilibrage à la racine si nécessaire

```
private static AVL balance(AVL t) { // t != null
  AVL l = t.left, r = t.right;
  int hl = height(l), hr = height(r);
  if (hl > hr + 1) { // déséquilibre à gauche
    AVL ll = l.left, lr = l.right;
    if (height(ll) >= height(lr))
      return rotateRight(t);
    else {
      t.left = rotateLeft(t.left);
      return rotateRight(t);
    }
  } else if (hr > hl + 1) { // déséquilibre à droite
    ... // symétrique
  } else {
    t.height = 1 + Math.max(hl, hr);
    return t;
  }
}
```

principe : une méthode de rotation renvoie la nouvelle racine

```
private static AVL rotateLeft(AVL t) {
  assert t != null && t.right != null;
  AVL r = t.right;
  t.right = r.left;
  r.left = t;
  t.height = 1 + Math.max(height(t.left), height(t.right));
  r.height = 1 + Math.max(height(r.left), height(r.right));
  return r;
}
```



de même pour la rotation droite (cf poly)

le **même code** qu'auparavant, avec balance ajouté dans la dernière instruction

```
static AVL add(AVL b, int x) {
  ...
  return balance(b);
}
static AVL removeMin(AVL b) {
  ...
  return balance(b);
}
static AVL remove(AVL b, int x) {
  ...
  return balance(b);
}
```

l'encapsulation dans une classe `Set` reste exactement la même
(en remplaçant partout `BST` par `AVL`)

on a vu que la hauteur est inférieure à $1{,}44 \log_2(N)$

$\Rightarrow$ chaque recherche, insertion ou suppression a un coût en $O(\log N)$

il existe d'autres techniques pour équilibrer les arbres binaires de recherche

notamment les arbres **rouges et noirs**

code générique

et si les éléments ne sont pas des chaînes ?

```
class BST<E> {
  E      value;
  BST<E> left, right;
  ...
}
```

on a besoin de pouvoir **comparer** les valeurs de type `E`

on veut pouvoir écrire `x.compareTo(y)` pour deux valeurs de type `E`

dans la bibliothèque Java, on trouve

```
interface Comparable<T> {
  int compareTo(T k);
}
```

on exige de E qu'elle implémente l'interface Comparable<E>

```
class BST<E extends Comparable<E>> {
  E       value;
  BST<E> left, right;
  ...
}
```

dans BST, on doit écrire

```
static<E extends Comparable<E>>
boolean contains(BST<E> b, E x) {
  ...
}
```

```
BST<Integer> bi = null;
bi = BST.add(bi, 42);
...
```

```
BST<String>  bs = null;
bs = BST.add(bs, "foo");
...
```

car `Integer` et `String` implémentent `Comparable`

note : l'argument `E` de `BST.add` est inféré

mais

```
class U { ... }
class Main { BST<U> b = ... }
```

est refusé

```
Bound mismatch: The type U is not a valid substitute for the
bounded parameter <E extends Comparable<E>> of the type BST<E>
```

`U` doit implémenter l'interface `Comparable`

```
class U implements Comparable<U> {
  ...
  public int compareTo(U o) { ... }
  ...
}
class Main { BST<U> b = ... }
```

bibliothèque Java

on trouve

- `java.util.TreeSet<E>` (ensemble)
- `java.util.TreeMap<K, V>` (dictionnaire)

- soit l'ordre naturel sur les éléments / clés si la classe implémente `Comparable` (comme on vient de le faire)

- soit un **comparateur** fourni au constructeur

```
TreeSet(Comparator<E> comparator)
TreeMap(Comparator<K> comparator)
```

avec

```
interface Comparator<T> {
  int compare(T o1, T o2);
}
```

les arbres binaires de recherche sont adaptés quand l'ordre sur les éléments importe

structure efficace

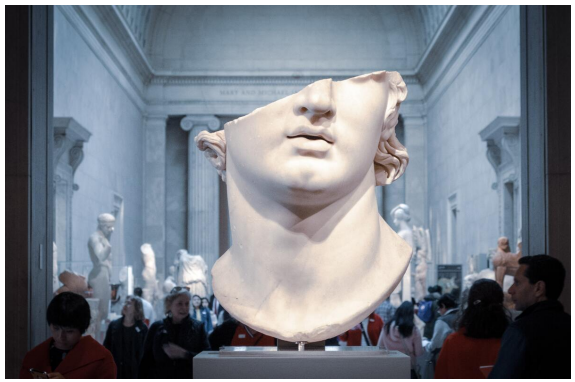
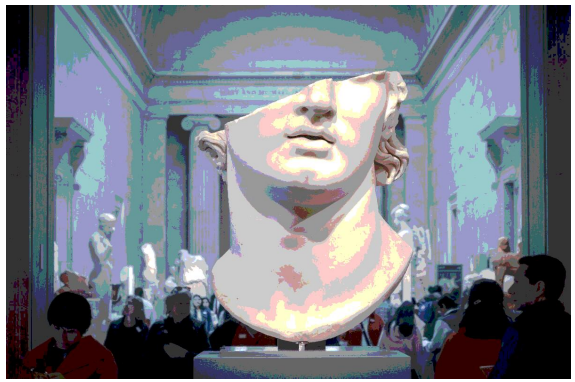
- équilibrage $\Rightarrow$ opérations en $O(\log N)$

le TD de cette semaine

une variante des ABR : les arbres k-dimensionnels

permet de trouver le plus proche voisin dans un ensemble de points de $\mathbb{R}^k$

application : sélection d'une palette de 256 couleurs parmi les 16777216 possibles pour compresser une image


la semaine prochaine

- **lire le poly**, chapitre 6 Arbres

  il y a des **exercices** dans le poly
  suggestions : ex 37 p 79, ex 40 p 91

- **bloc 6** : encore des arbres, mais différents