École Polytechnique

CSC_41011

# Les bases de la programmation et de l'algorithmique

Jean-Christophe Filliâtre

arbres (2/2)

un tutorat a été mis en place

le **mardi de 18h45 à 20h45** en salle info 31

s'inscrire par mail (`tutorat.1a-2a@polytechnique.fr`)

# aujourd'hui

deux nouvelles structures de données utilisant des arbres

1. arbres de préfixes
2. cordes

# arbres de préfixes

# Boggle

- une grille $4 \times 4$ de lettres est donnée
- quels mots peut-on former en se déplaçant dans cette grille, chaque lettre étant utilisée au plus une fois ?

organiser **des mots** dans une structure de données

avec en particulier les opérations

- ajouter un nouveau mot
- trouver tous les mots dont $w$ est un préfixe

un **tableau trié**

- recherche en $O(\log N)$ — *binary search*, voir poly
- ajout en $O(N)$ — **inacceptable**

une **table de hachage**

- ajout en $O(1)$
- recherche en $O(N)$ — **inacceptable**
  (car aucune façon simple de trouver les mots de préfixe $w$)

un **AVL**

- ajout en $O(\log N)$
- recherche du préfixe $w$ en $O(\log N)$

  ensuite, les mots ayant $w$ comme préfixe sont “à côté”

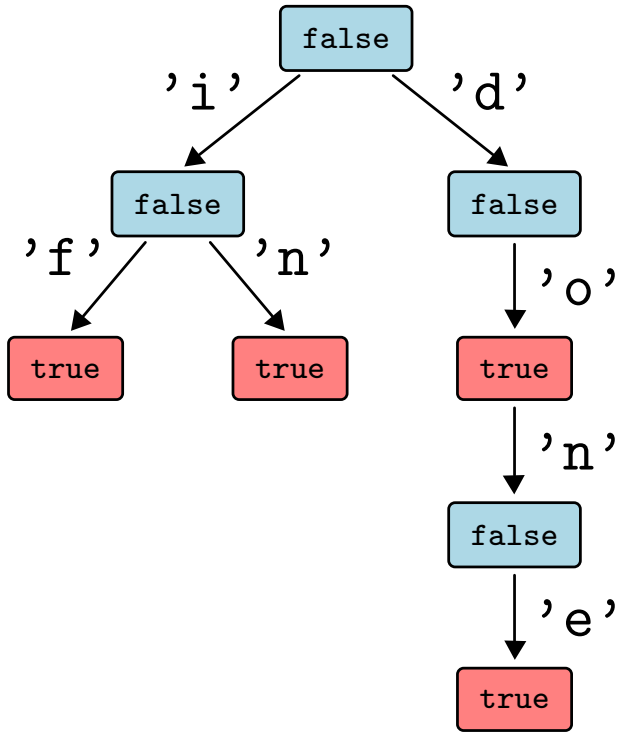
déjà très bien

oui,

en exploitant la structure **séquentielle** des chaînes de caractères

# arbre de préfixes : principe

$\{\texttt{"if"}, \texttt{"in"}, \texttt{"do"}, \texttt{"done"}\}$




(**trie** en anglais, de *retrieval*)

l'arbre `false` représente l'ensemble vide $\emptyset$

l'arbre `true` représente le singleton $\{\texttt{""}\}$

```
class Trie {
          Trie()                  // crée un ensemble vide
  boolean isEmpty()               // vide ?
  boolean contains(String s)      // test d'appartenance
     void add(String s)           // ajoute un élément
     void remove(String s)        // supprime un élément

Collection<String> below(String s)
                                  // tous les mots de préfixe s
}
```

le branchement est matérialisé par une table de hachage

```
class Trie {
  private boolean word;
  private HashMap<Character, Trie> branches;
```

(si l'alphabet est petit, un tableau est encore plus simple)

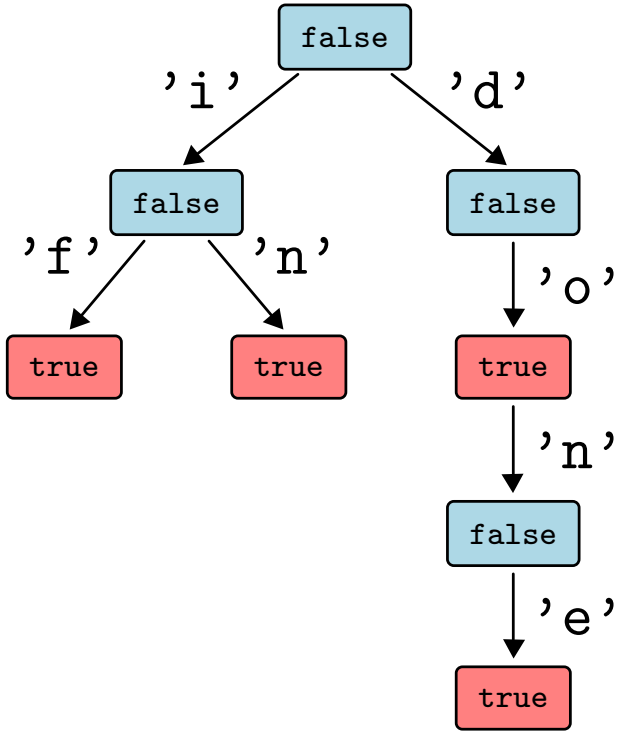
```
  Trie() {
    this.word     = false;
    this.branches = new HashMap<Character, Trie>();
  }
```

pour chercher un mot

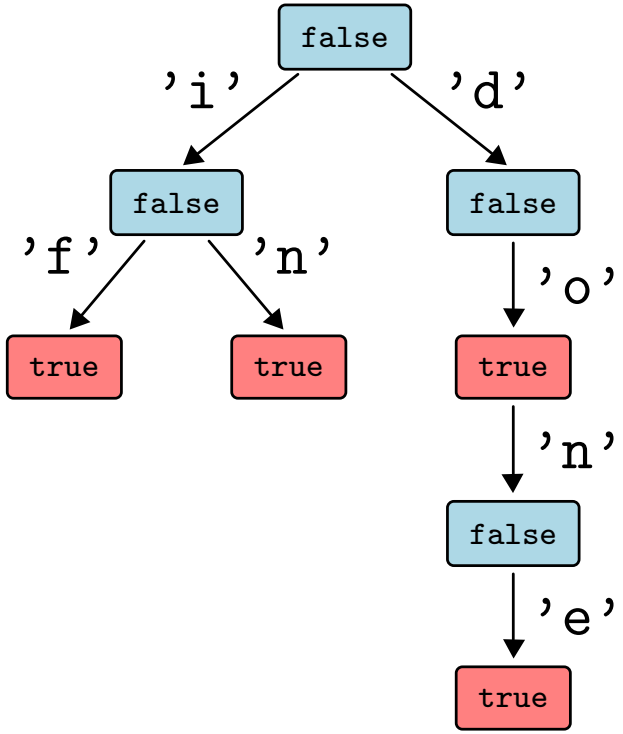
$$\mathtt{s} = s_0 s_1 \dots s_{n-1}$$

il faut descendre dans l'arbre
en suivant les caractères $s_i$

la branche peut ne pas exister

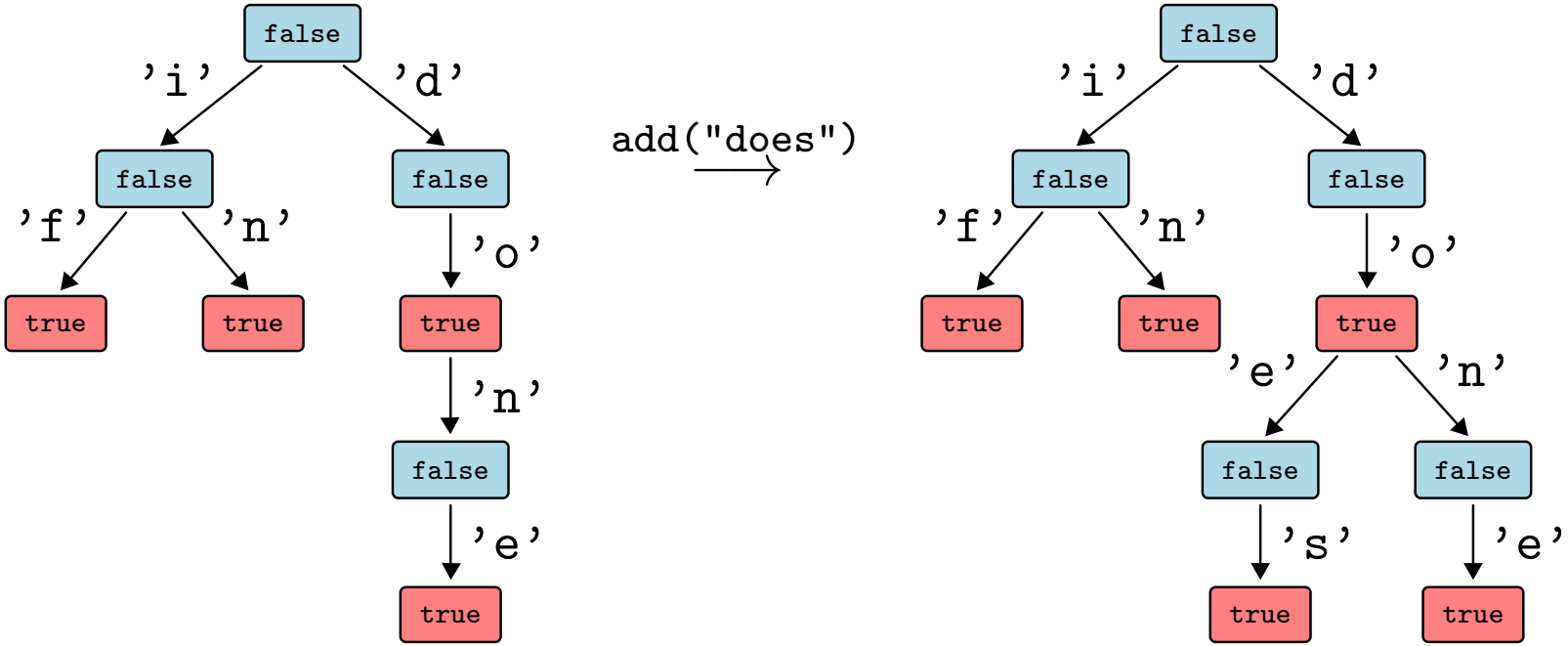


```
boolean contains(String s) {
  Trie t = this;
  for (int i = 0; i < s.length(); i++) {
    // invariant t != null
    t = t.branches.get(s.charAt(i));
    if (t == null) return false;
  }
  return t.word;
}
```

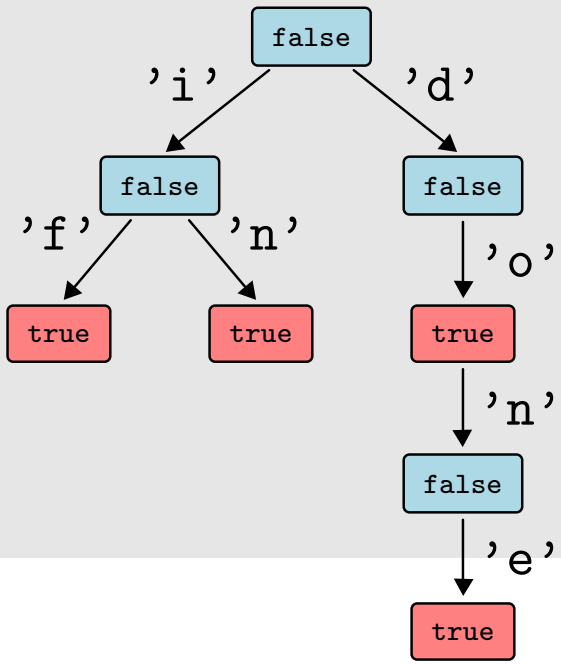



complexité $O(|s|)$

pour ajouter un mot, on procède de la même façon
mais il faut **créer** les branches qui n'existent pas

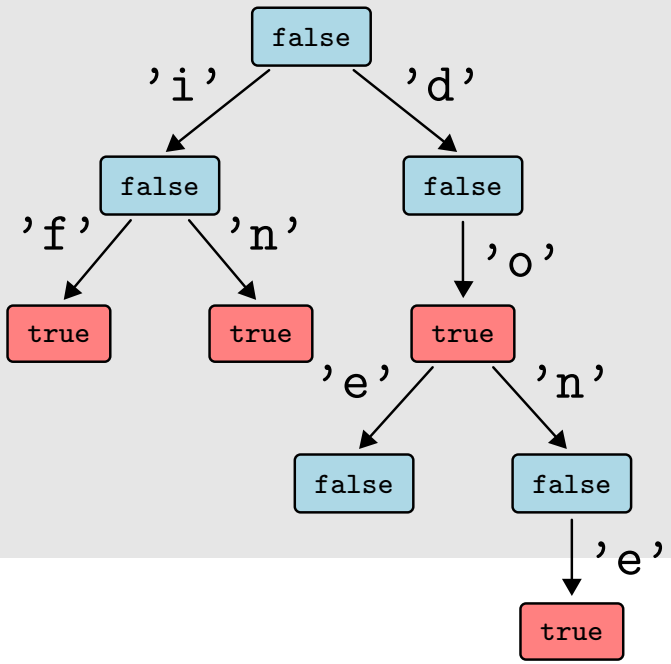


```
void add(String s) {
  Trie t = this;
  for (int i = 0; i < s.length(); i++) {
    // invariant t != null
    char c = s.charAt(i);
    HashMap<Character, Trie> b = t.branches;
    t = b.get(c);
    if (t == null) {
      t = new Trie();
      b.put(c, t);
    }
  }
  t.word = true;
}
```



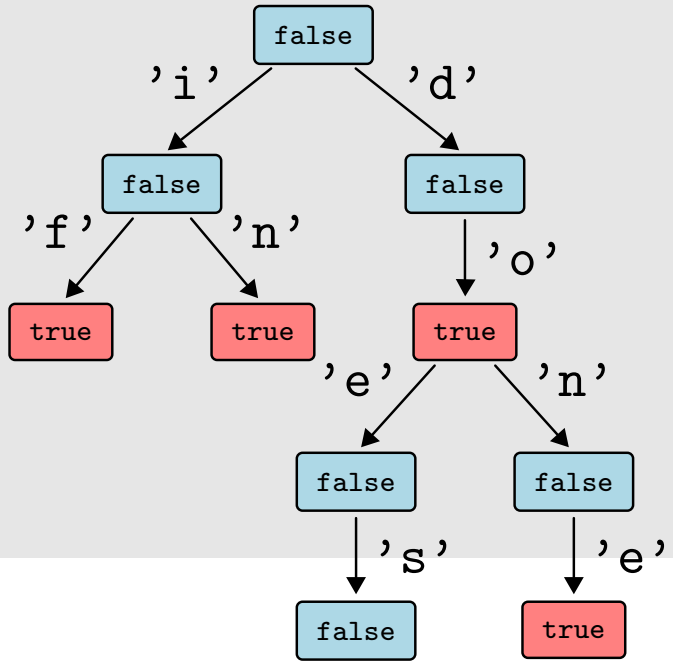

complexité $O(|s|)$

```
void add(String s) {
  Trie t = this;
  for (int i = 0; i < s.length(); i++) {
    // invariant t != null
    char c = s.charAt(i);
    HashMap<Character, Trie> b = t.branches;
    t = b.get(c);
    if (t == null) {
      t = new Trie();
      b.put(c, t);
    }
  }
  t.word = true;
}
```



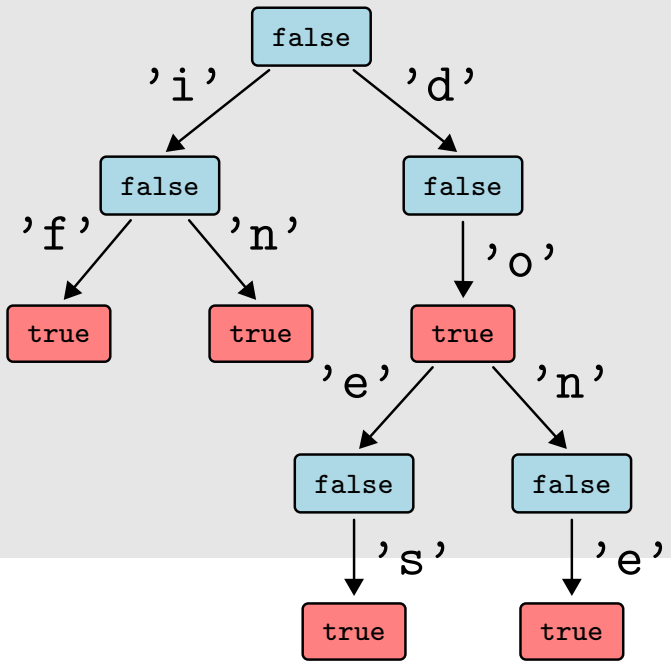

complexité $O(|s|)$

```
void add(String s) {
  Trie t = this;
  for (int i = 0; i < s.length(); i++) {
    // invariant t != null
    char c = s.charAt(i);
    HashMap<Character, Trie> b = t.branches;
    t = b.get(c);
    if (t == null) {
      t = new Trie();
      b.put(c, t);
    }
  }
  t.word = true;
}
```




complexité $O(|s|)$

```
void add(String s) {
  Trie t = this;
  for (int i = 0; i < s.length(); i++) {
    // invariant t != null
    char c = s.charAt(i);
    HashMap<Character, Trie> b = t.branches;
    t = b.get(c);
    if (t == null) {
      t = new Trie();
      b.put(c, t);
    }
  }
  t.word = true;
}
```

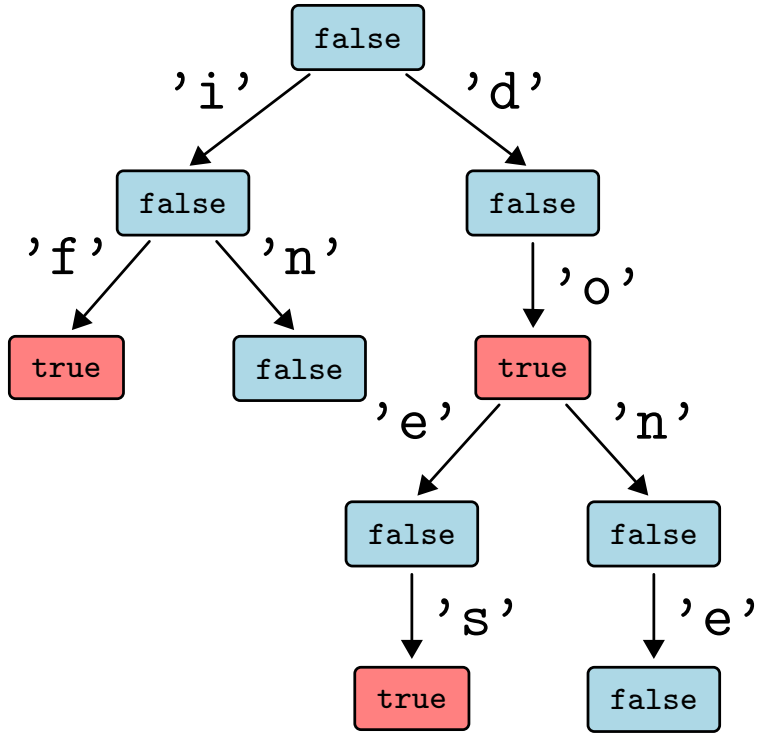



complexité $O(|s|)$

la suppression peut se faire de la même façon

mais cela peut conduire à des **branches vides**

```
s.remove("done");
s.remove("in");
```




on peut les supprimer en remontant (cf exercices 44–46 du poly)

# retour sur le problème initial (Boggle)

1. on met tous les mots du dictionnaire dans un arbre de préfixes
2. on applique la technique du rebroussement (amphi 4)

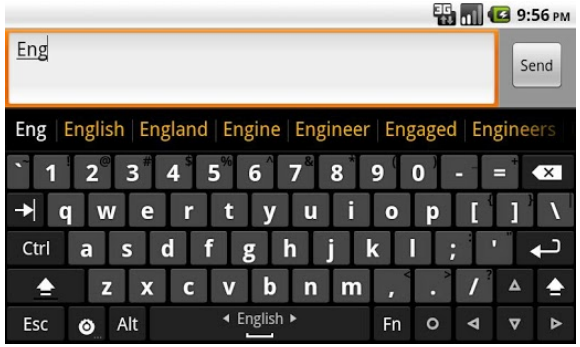
```
void find(Trie t, int i, int j, int mask, String prefix) {
  ...
}
```

- on se déplace dans la grille tout en descendant dans l'arbre
- on rebrousse chemin quand on ne peut plus descendre dans l'arbre

(code sur la page du cours)

# autre application : autocomplétion

- saisie prédictive sur les téléphones




- complétion des noms de fichiers dans le terminal

```
$ ba
badblocks   baobab      base64      bash
banner      base32      basename    bashbug
```

(en appuyant sur TAB)

pour trouver tous les mots dont $w$ est un préfixe

1. on cherche le nœud correspondant à $w$, comme dans `contains`

   complexité $O(|\texttt{w}|)$

2. on récolte tous les mots dans le sous-arbre
   c'est un parcours (cf `inorder` la semaine dernière)

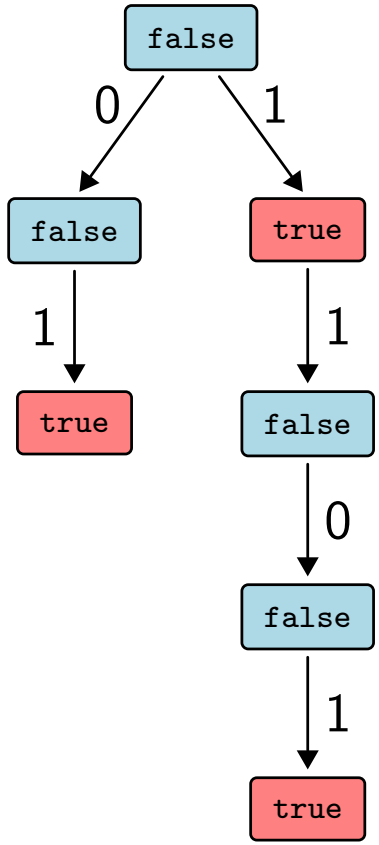
   complexité $O(M)$ s'il y a $M$ mots dans le sous-arbre

(code sur la page du cours)

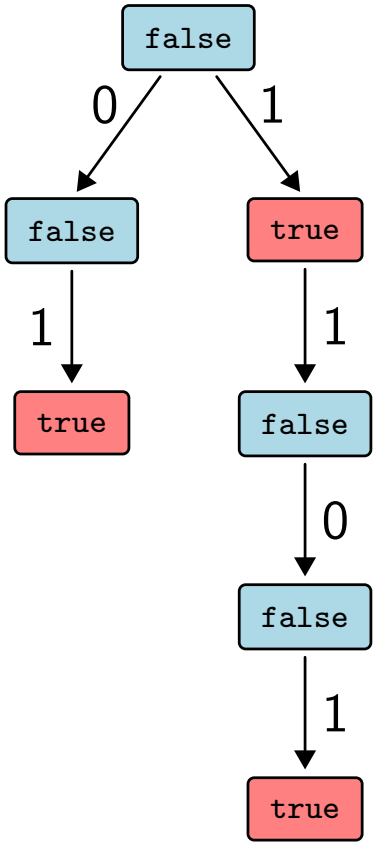
l'idée s'applique dès que les éléments ont une structure de **séquence**

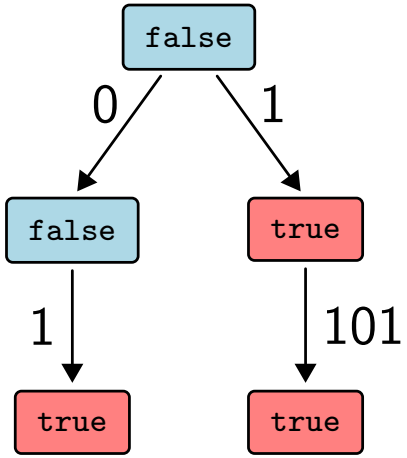
exemple :
les chiffres d'un entier en base 2

$$\{1, 2, 11\} = \{1_2, 10_2, 1011_2\}$$



on peut compresser les chemins ne contenant que `false`




cette structure s'appelle
**un arbre de Patricia**
(Morrison, 1968)

# cordes

concours de programmation
ICFP 2007

`http://save-endo.cs.uu.nl/`

une longue chaîne de 7 523 060 caractères à manipuler
(chercher, copier, coller, remplacer, etc.)

# le type `String` de Java

une chaîne de caractères est une structure **immuable**

une fois construite, une chaîne ne peut pas être modifiée

en particulier, il y a une méthode `charAt`
mais pas de méthode `setCharAt`

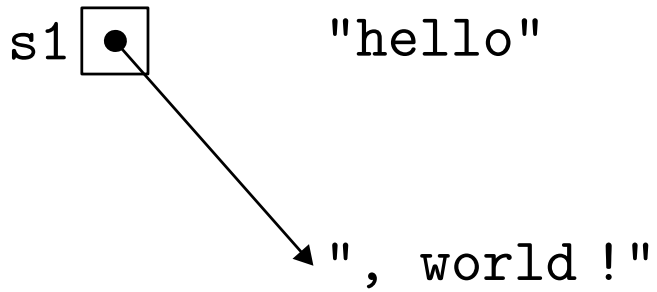
bien entendu, on peut toujours modifier le contenu d'une **variable** de type `String`

```
String s1 = "hello";
s1 = ", world!";
```


", world !"

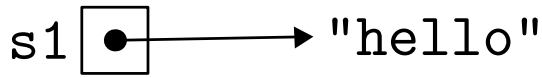
bien entendu, on peut toujours modifier le contenu d'une **variable** de type `String`

```
String s1 = "hello";
s1 = ", world!";
```



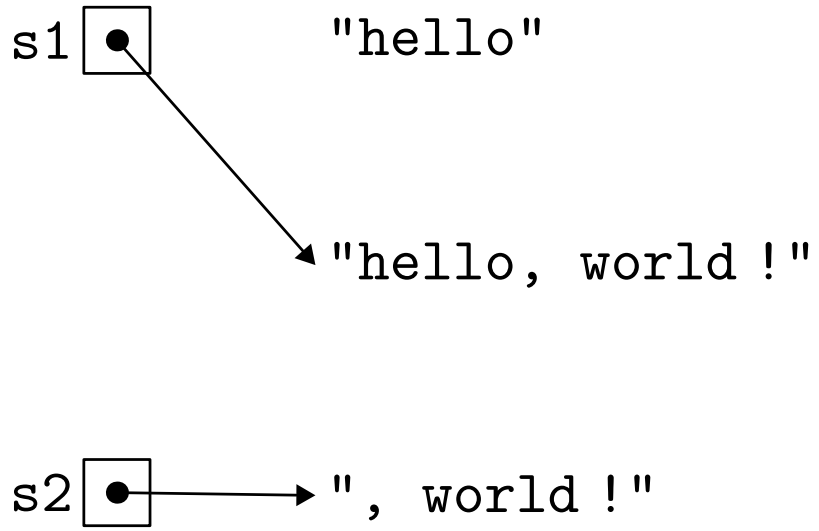
quand on concatène deux chaînes, une **nouvelle** chaîne est construite

```
String s1 = "hello";
String s2 = ", world!";
s1 = s1 + s2;
```



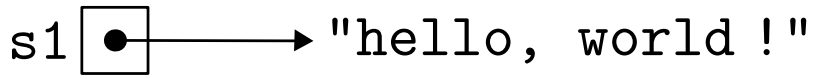

quand on concatène deux chaînes, une **nouvelle** chaîne est construite

```
String s1 = "hello";
String s2 = ", world!";
s1 = s1 + s2;
```



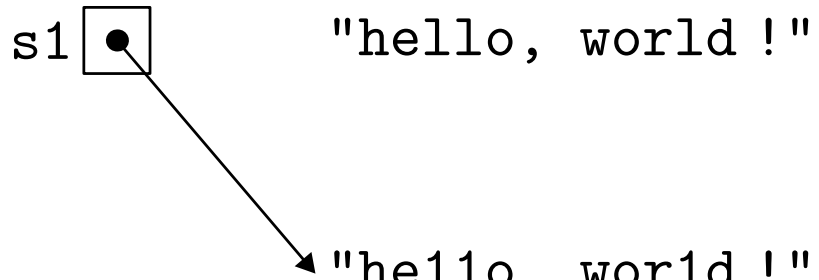
de même si on utilise `replace`

```
String s1 = "hello, world!";
s1 = s1.replace("l", "1");
```



de même si on utilise `replace`

```
String s1 = "hello, world!";
s1 = s1.replace("l", "1");
```



quel intérêt y a-t-il à avoir des chaînes immuables ?

$\Rightarrow$ pas de risque de **modification a posteriori** (accidentelle ou malicieuse)

en particulier lorsque les chaînes

- entrent dans la composition d'objets

```
class Person { final String name; ... }
```

(`final` empêche la modification du champ `name`, pas de l'objet)

- jouent un rôle clé dans des structures de données
(modifier une chaîne modifierait `hashCode`/`compareTo`)

# cordes

une structure de **chaîne de caractères** où

- la concaténation
- l'extraction de sous-chaîne

sont efficaces

`String` ne convient pas

- `s1.substring(begin, end)` coûte

$$O(\texttt{end} - \texttt{begin})$$

- `s1.concat(s2)` coûte

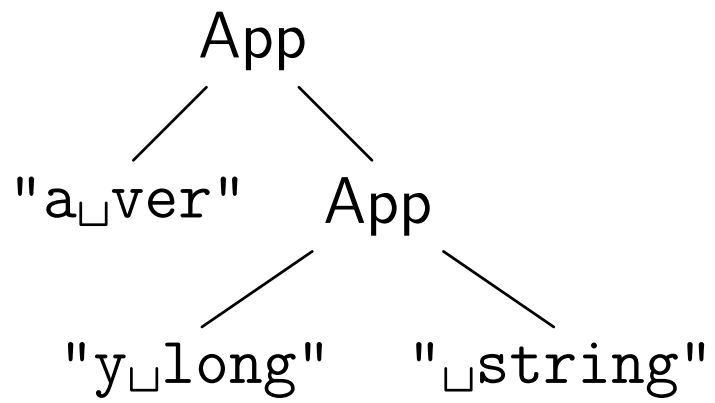
$$O(\texttt{s1.length()} + \texttt{s2.length()})$$

**démo**

principe = représenter la chaîne

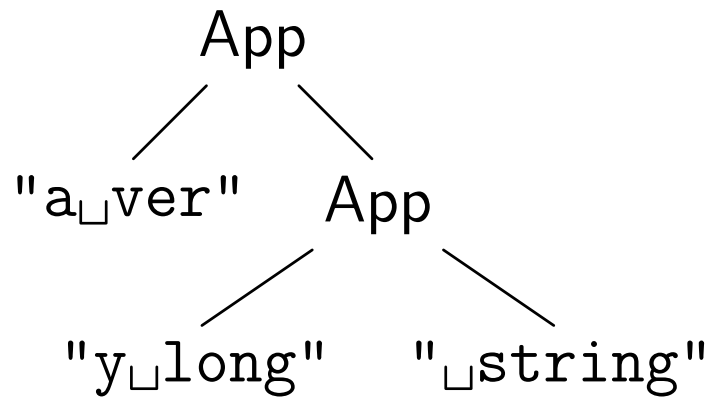
```
"a ver" + "y long" + " string"
```

par l'arbre



```
class Rope {
        Rope(String s)              // une chaîne égale à s
     int length()                   // longueur
    char get(int i)                 // i-ième caractère
    Rope append(Rope r)             // concaténation
    Rope sub(int begin, int end); // sous-chaîne
}
```

on fait ici le choix d'une structure **immuable**
(i.e. append et sub renvoient de nouvelles cordes)

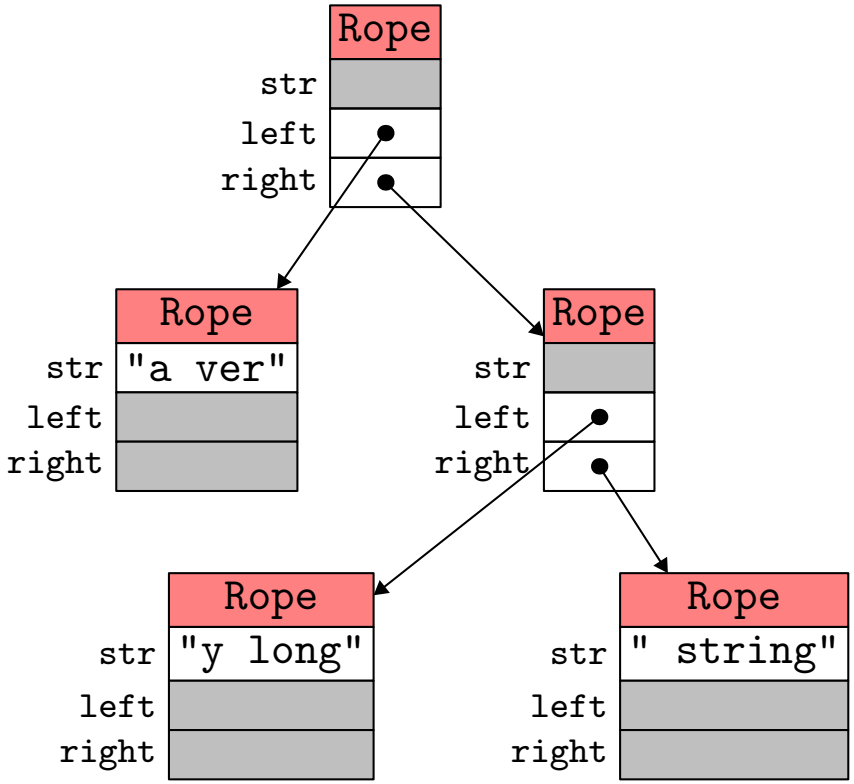



on pourrait écrire

```
class Rope {
  String str;           // si c'est une feuille
  Rope left, right;     // si c'est un noeud interne
  ...
}
```

mais ce serait **gâcher** de la mémoire

et on risque d'utiliser
un champ non significatif
**par accident**



une autre représentation utilisant l'**héritage** de classes

- une classe `Rope`
- deux sous-classes `Str` et `App`

# la classe Rope

représente une corde quelconque, avec sa longueur

```
abstract class Rope {
  final int length;
}
```

(conserver la longueur sera important pour la suite)

classe abstraite $\Rightarrow$ on ne peut pas construire d'objet de type Rope

```
Rope r = new Rope();
```

```
Cannot instantiate the type Rope
```

représente une feuille

```
class Str extends Rope {
  final String str;
```

hérite du champ `length` de `Rope`

```
  Str(String str) {
    this.length = str.length();
    this.str    = str;
  }
}
```

(un peu plus subtile en pratique ; cf poly page 100)

# la sous-classe App

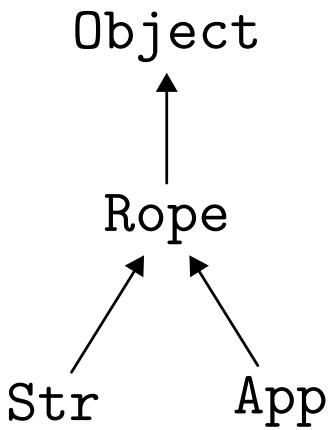
représente un nœud interne

```
class App extends Rope {
  final Rope left, right;
}
```

hérite également du champ `length` de `Rope`

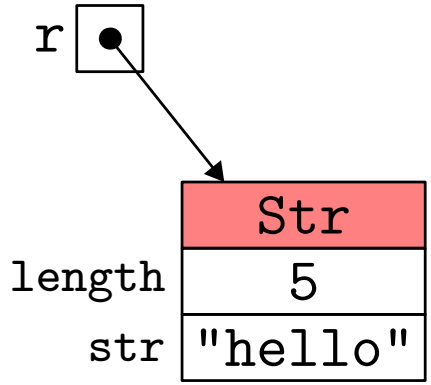
```
  App(Rope left, Rope right) {
    this.length = left.length + right.length;
    this.left   = left;
    this.right  = right;
  }
}
```

la relation d'héritage introduit une relation de **sous-typage**




se lit « toute valeur de type `Str` (resp. `App`) peut être considérée comme une valeur de type `Rope`, ou encore de type `Object` »

```
Rope r = new Str("hello");
```

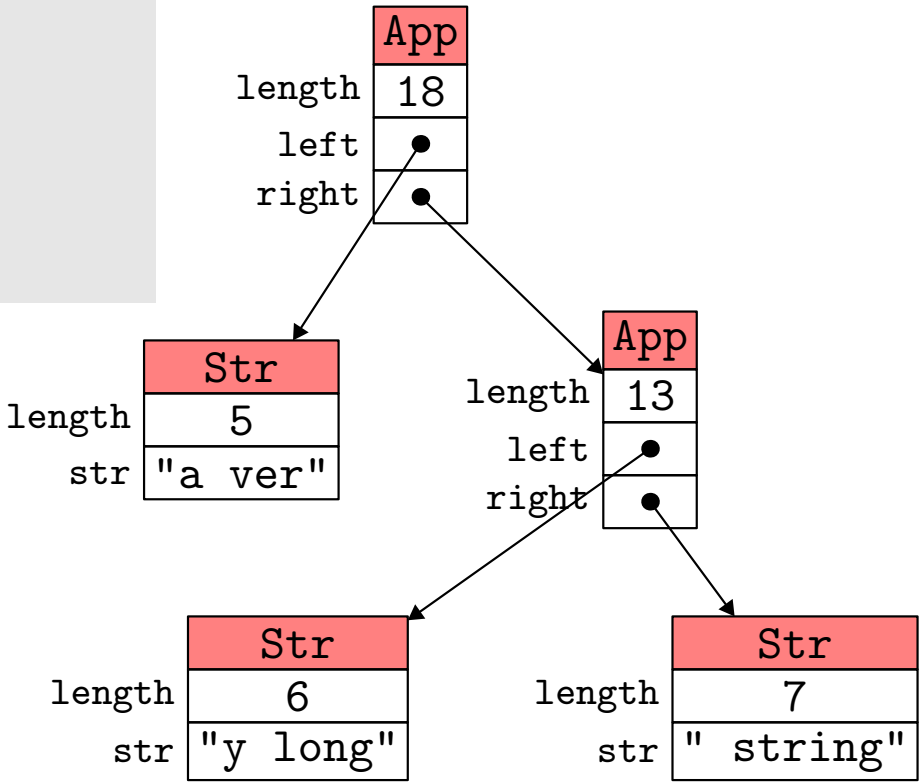



l'**objet** créé a le type `Str`
(rappel : la classe d'un objet ne peut être modifiée)

la **variable** `r` a le type `Rope`

# exemple

```
Rope r =
  new App(
    new Str("a ver"),
    new App(
      new Str("y long"),
      new Str(" string")));
```



comment définir une **opération** sur les cordes ?

par le mécanisme de **redéfinition**

accès au `i`-ième caractère d'une corde

```
abstract class Rope {
  ...
  abstract char get(int i);
}
```

une méthode abstraite n'est pas définie, seulement **déclarée**

en revanche, il **faut** la définir dans les classes `Str` et `App`

c'est immédiat

```
class Str extends Rope {
  ...

  @Override
  char get(int i) {
    return this.str.charAt(i);
  }
}
```

(rappel : `@Override` indique une redéfinition)

le caractère `i` se trouve à gauche ou à droite

```
class App extends Rope {
  ...

  @Override
  char get(int i) {
    return (i < this.left.length) ?
           this.left.get(i) :
           this.right.get(i - this.left.length);
  }
}
```

on ne connaît pas la nature de `this.left` (à savoir `Str` ou `App`)
et pourtant on peut écrire

```
    this.left.get(i)
```

le bon code sera appelé ; c'est l'**appel dynamique** de méthode

(cf amphi 2 et poly section 1.1.4)

extraire une sous-corde

```
abstract class Rope {
  ...
  // les caractères de begin (inclus) à end (exclu)
  abstract Rope sub(int begin, int end);
}
```

c'est immédiat

```
class Str extends Rope {
  ...

  @Override
  Rope sub(int begin, int end) {
    return new Str(this.str.substring(begin, end));
  }
}
```

```
class App extends Rope {
  ...
  @Override
  Rope sub(int begin, int end) {
    int endr = end - this.left.length;
    // entièrement à gauche
    if (endr <= 0)
      return this.left.sub(begin, end);
    int beginr = begin - this.left.length;
    // entièrement à droite
    if (beginr >= 0)
      return this.right.sub(beginr, endr);
    // à cheval
    return new App(
      this.left.sub(begin, this.left.length),
      this.right.sub(0, endr));
  }
```

conserver la longueur dans chaque corde est crucial

si on avait seulement une méthode récursive `length()`
alors `get` et `sub` seraient quadratiques dans le pire des cas

(exercice : le démontrer)

**démo**

on ajoute les contrôles d'accès appropriés

`private` = visible seulement dans la classe

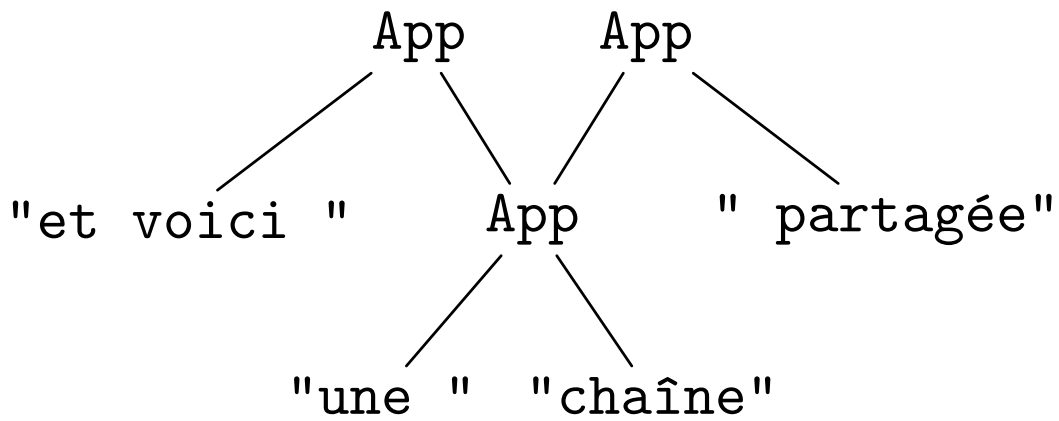
`protected` = visible seulement dans les sous-classes

exercice 50 du poly

les champs `length`, `str`, `left` et `right` sont tous déclarés `final`

les cordes sont donc un exemple de structure immuable :
une fois construite, une corde ne peut être modifiée

le caractère immuable des cordes permet le **partage**



une autre application des cordes : un éditeur de texte

- une ligne est une corde de caractères
- un texte est une corde de lignes

# récapitulation

- une autre façon de représenter des arbres, utilisant l'héritage
- une alternative au type `String`
- un exemple de structure immuable

# des ABR immuables

il est très facile de rendre les ABR de la semaine dernière immuables en ajoutant `final` devant les champs

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

(de même si c'était `AVL`)

certaines méthodes restent inchangées

```
static boolean contains(BST b, String x)
static String  getMin(BST b)
```

car elles ne cherchent pas à modifier les champs

les autres, en revanche, ne peuvent plus modifier les arbres en place

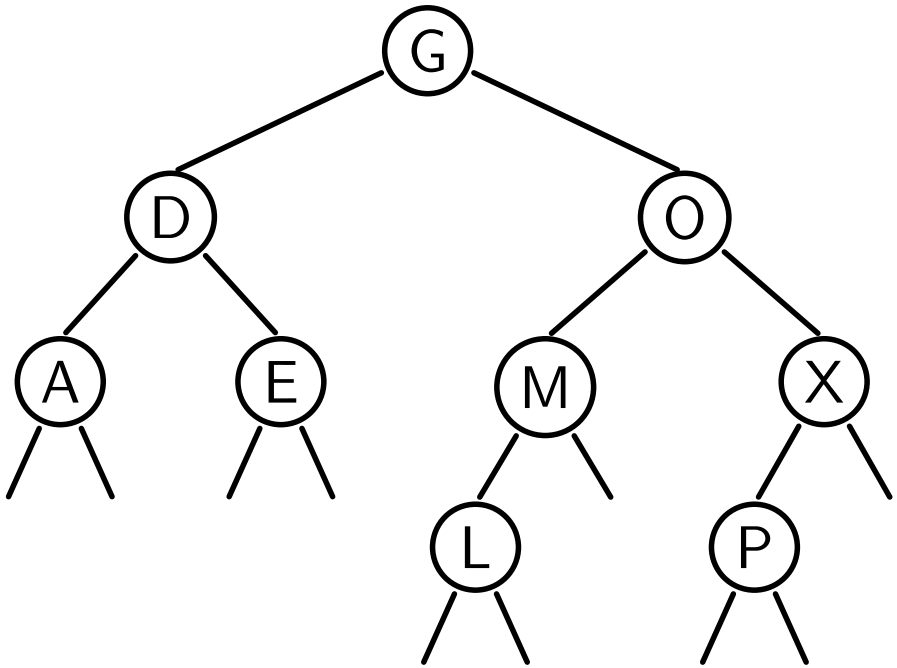
```
static BST add(BST b, String x) {
  ...
  if (c < 0)
    b.left = add(b.left, x);
  ...
}
```

```
  the final field BST.left cannot be assigned
```

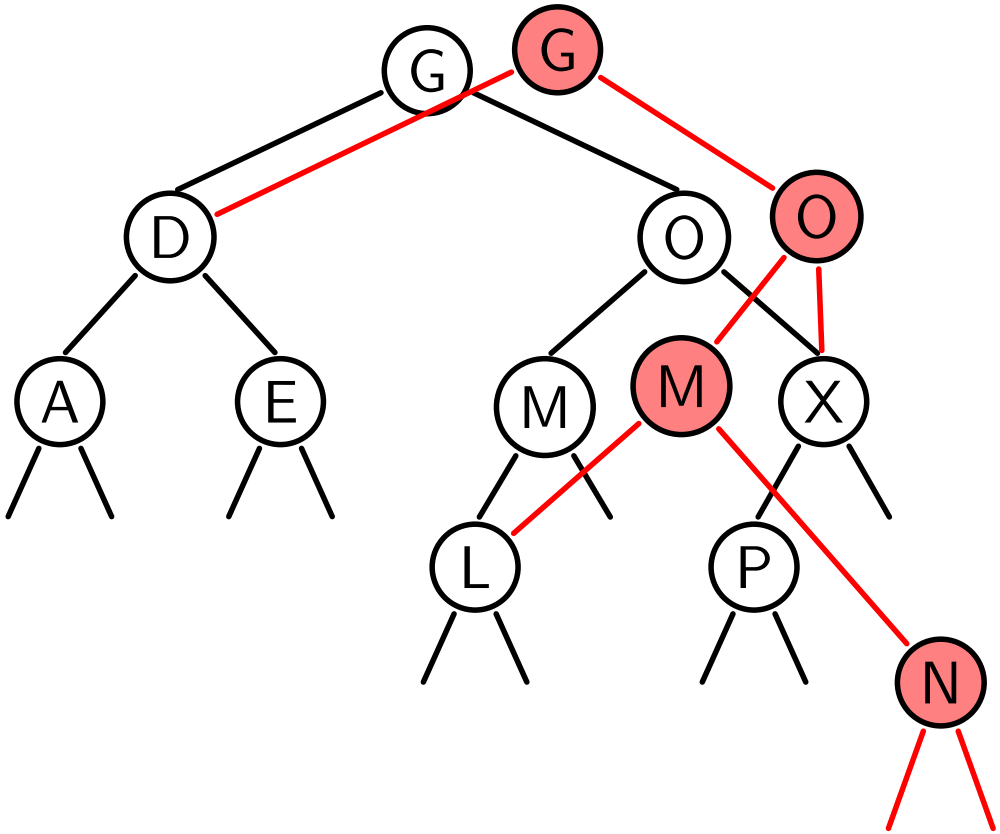
la méthode add doit maintenant renvoyer **un nouvel arbre**,
sans modifier son argument

```
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)
    return new BST(add(b.left, x), b.value, b.right);
  if (c > 0)
    return new BST(b.left, b.value, add(b.right, x));
  return b; // x déjà dans b
}
```

```
BST t1 = ...;
BST t2 = BST.add(t1, "N");
```



```
BST t1 = ...;
BST t2 = BST.add(t1, "N");
```



on a

- **dupliqué** les nœuds rencontrés pendant la descente
- **partagé** les sous-arbres ignorés pendant la descente

si l'arbre contient $N$ nœuds

- dans le pire des cas, on peut construire $N$ nouveaux nœuds (insertion tout en bas d'un peigne)
- si l'arbre est équilibré, ce sera au plus $O(\log N)$ nouveaux nœuds

# expérience (ici avec des entiers)

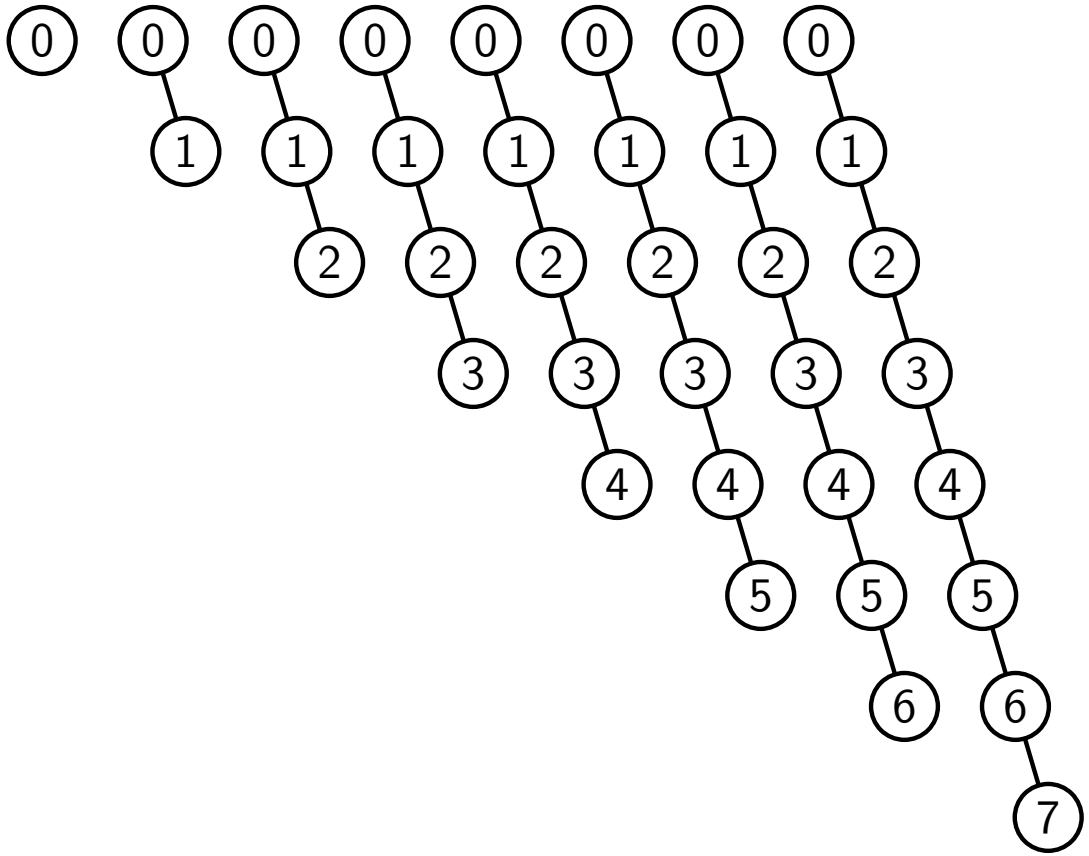
on insère successivement les entiers 0, 1, 2, ..., 7

```
s0 = add(null, 0);
s1 = add(s0,   1);
...
s7 = add(s6,   7);
```

et on observe l'ensemble des 8 arbres obtenus

ce sont 8 peignes

on a construit
$1+2+\cdots+8 = 36$
nœuds au total



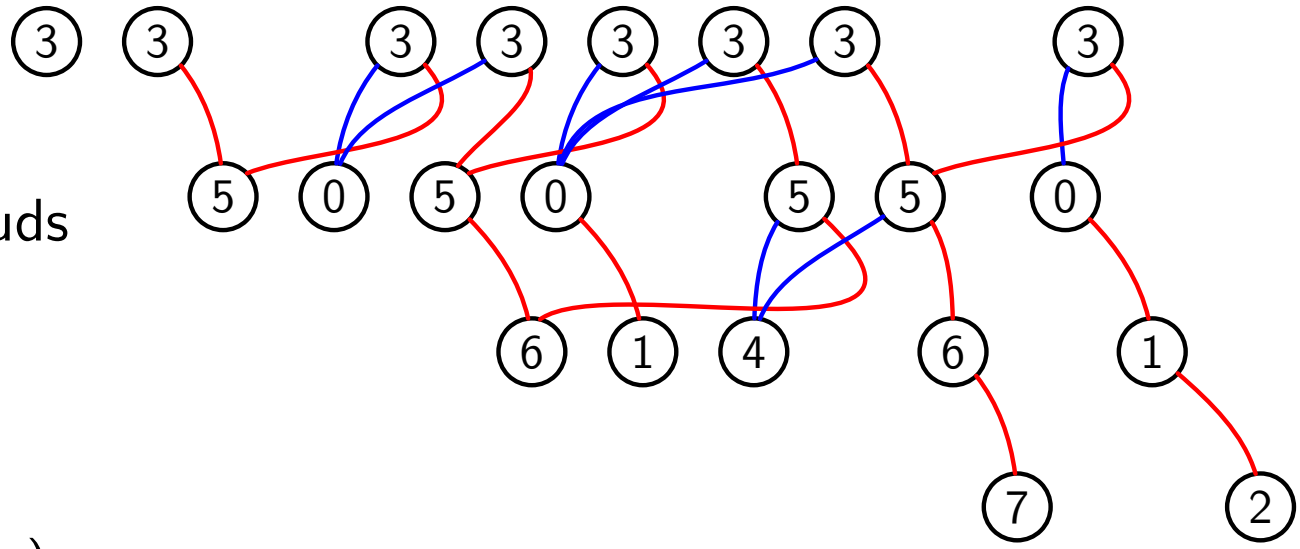
on insère maintenant les entiers dans un ordre différent
à savoir 3, 5, 0, 6, 1, 4, 7, 2

```
s0 = add(null, 3);
s1 = add(s0,   5);
...
s7 = add(s6,   2);
```

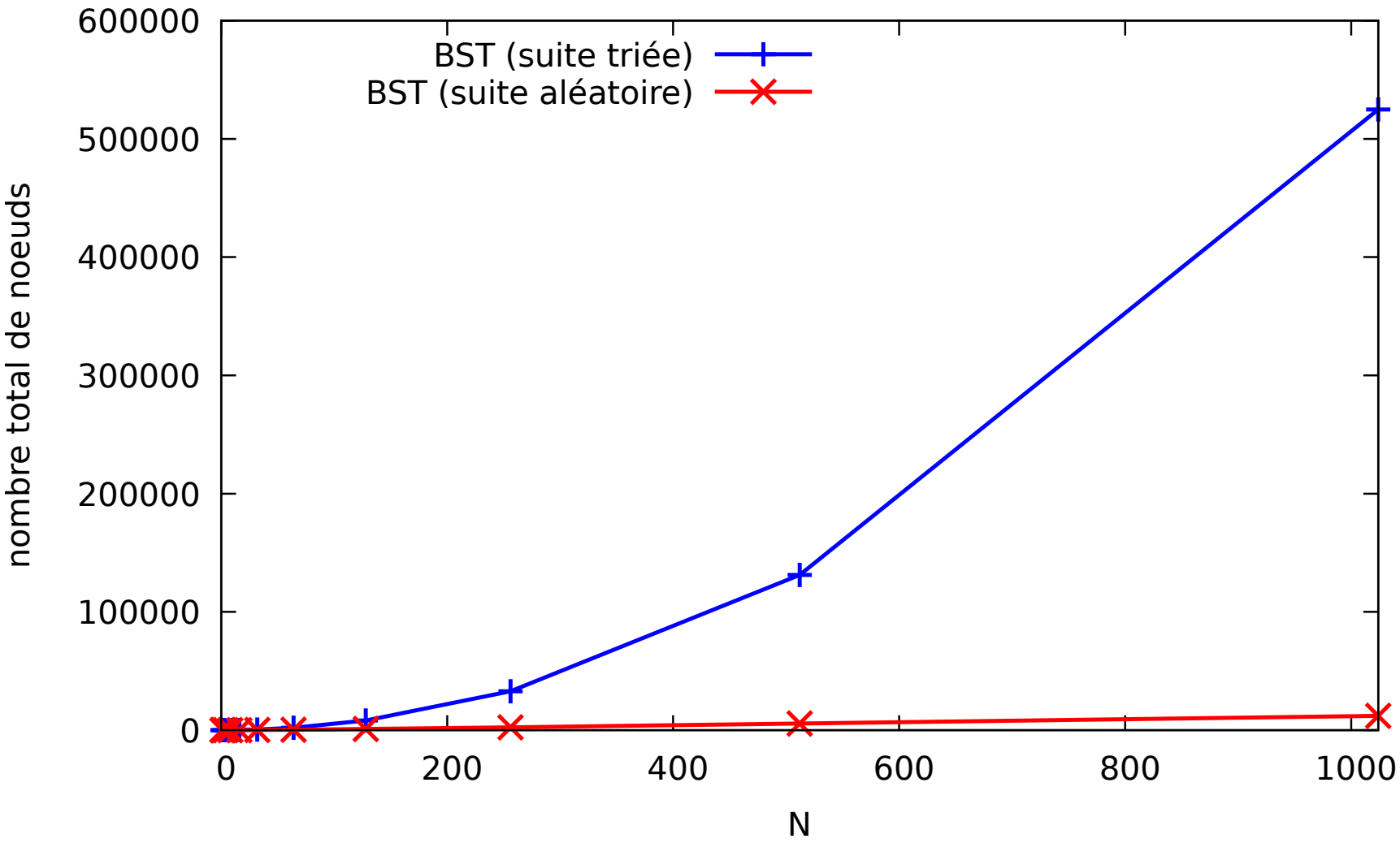
on a construit
seulement 22 nœuds

grâce au **partage**

(rouge = fils droit
bleu = fils gauche)



répétons l'expérience avec $N$ entiers

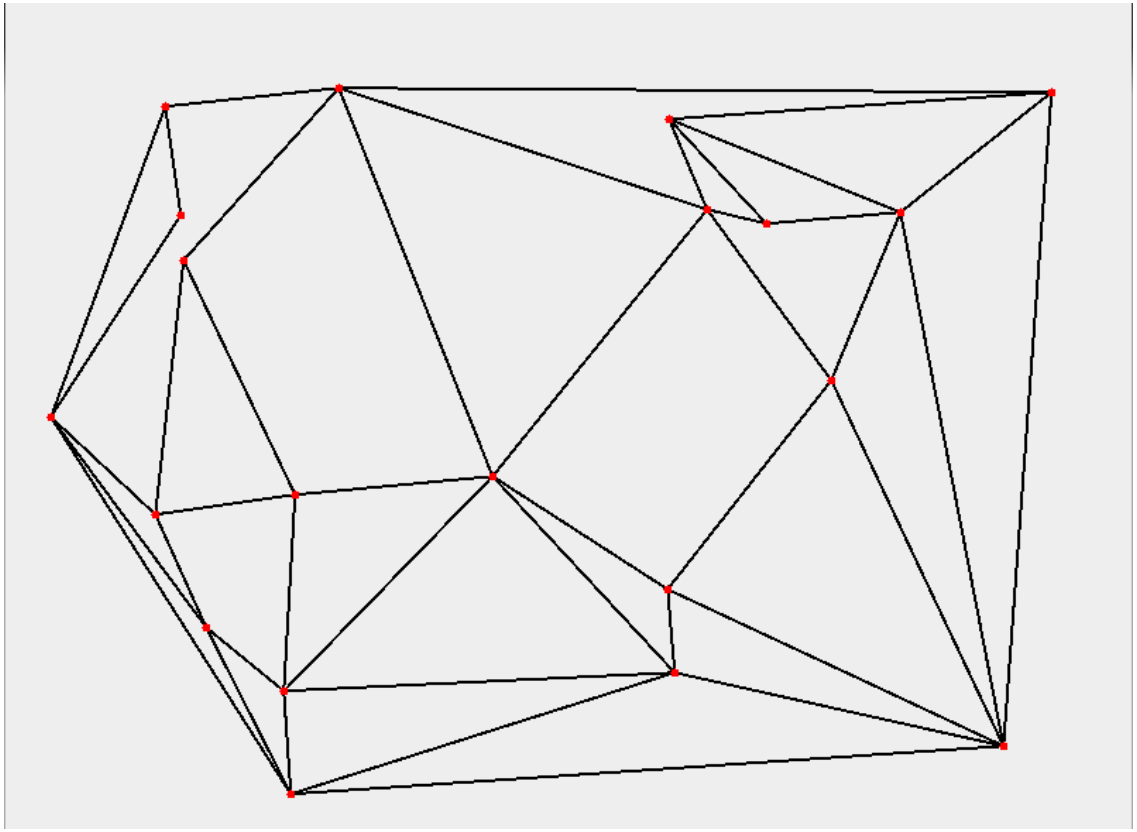


de même, on modifie les méthodes

```
static BST removeMin(BST b)
static BST remove(BST b, String x)
```

pour qu'elles renvoient de nouveaux arbres

le plan est partagé en polygones disjoints, qui partagent des arêtes



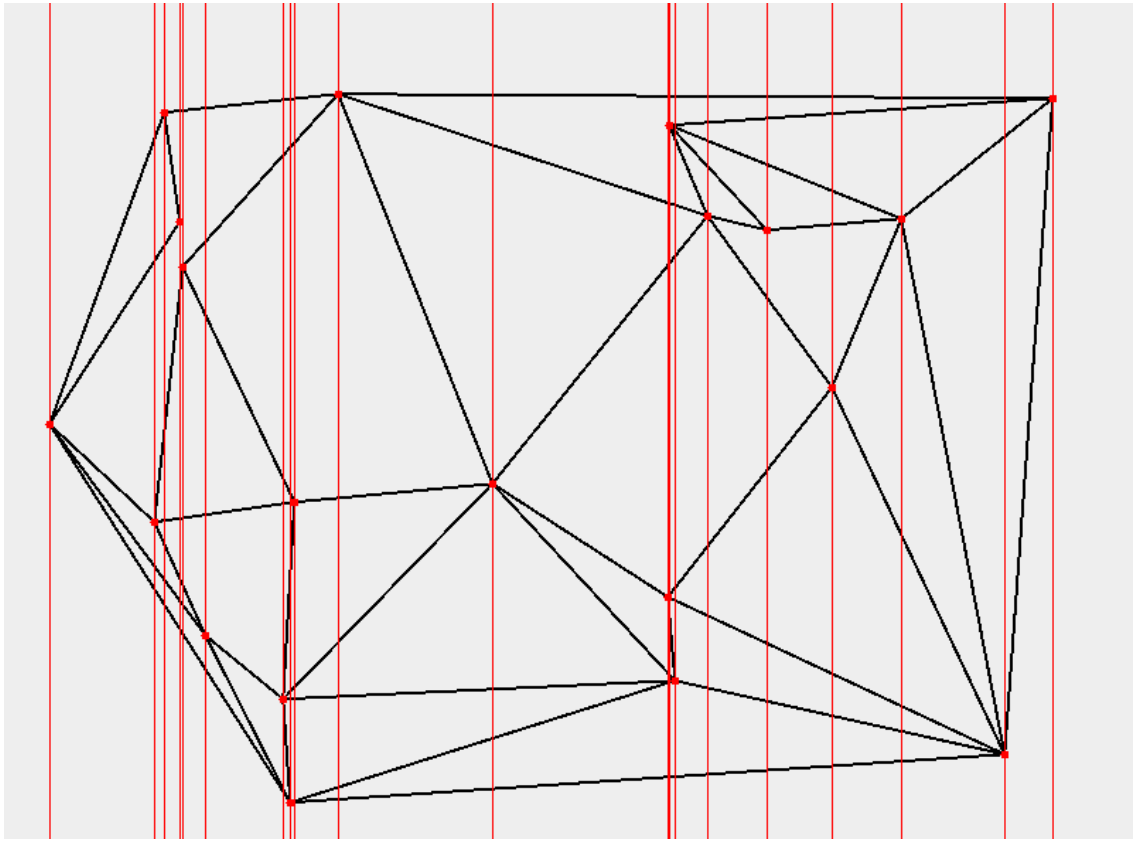
étant donné un point, dans quel polygone se trouve-t-il ?

# plus précisément

- les polygones sont fixés
- ils contiennent $N$ arêtes au total

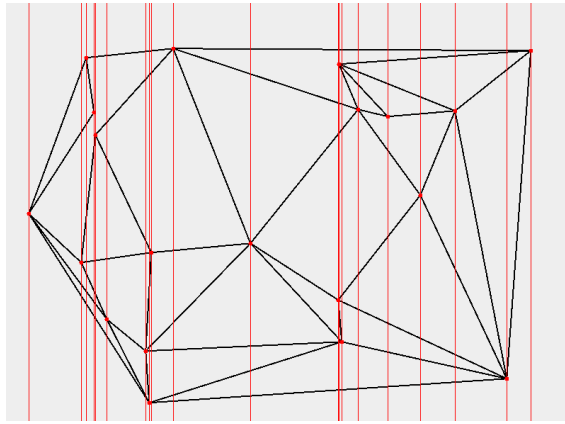
- on veut stocker cette information dans une structure une seule fois, pour répondre ensuite au problème plusieurs fois

on découpe le plan en bandes verticales, triées selon $X$



dans chaque bande, les arêtes sont triées selon $Y$

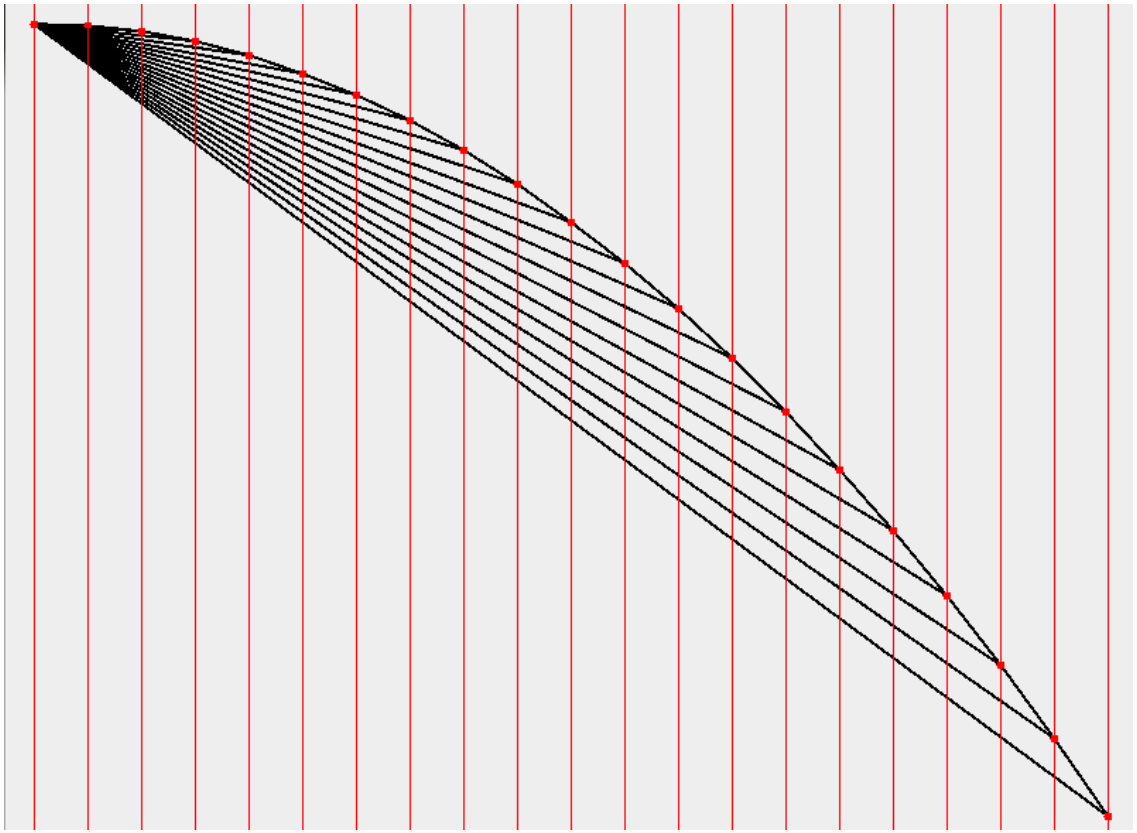
étant donné un point $(X, Y)$



1. on cherche dans quelle bande il se situe
   - recherche dichotomique en temps $O(\log N)$ (au plus $N$ bandes)

2. dans cette bande, on cherche entre quelles arêtes il se situe
   - recherche dichotomique en temps $O(\log N)$ (au plus $N$ arêtes dans la bande)

il peut y avoir $O(N)$ bandes
chaque bande peut contenir $O(N)$ arêtes



d'où un espace total $O(N^2)$ potentiellement

si les bandes sont représentées par des AVL immuables

on peut les construire de la gauche vers la droite

- les arêtes se terminant sont supprimées
- les arêtes commençant sont ajoutées

chaque arête est ajoutée une fois et retirée une fois
d'où un coût total $O(N \log N)$, en temps **et en espace**

la clé est le **partage** (de sous-arbres)

avec les polygones du transparent 78

| $N$ | nœuds |
|---|---|
| 10 | 28 |
| 20 | 58 |
| 100 | 194 |
| 200 | 344 |

- beaucoup de types d'arbres différents
  - `BST`/`AVL`, `Trie`, `Rope`, etc.
- plusieurs façons de représenter un arbre
  - `null` / sous-classes
  - binaire / $n$-aire
  - modifiable / immuable

# le TD de cette semaine

comment maintenir toutes les sommes $\sum$ `a[lo..hi[` d'un tableau `a`, tout en permettant de le modifier ?

avec des **arbres de Fenwick**

application : compter les inversions dans un tableau

- **lire le poly**

  6.4 Arbres de préfixes

  7 Structures de données immuables (cordes)

  il y a des **exercices** dans le poly
  suggestions : ex 44 p 94, ex 51 p 102

- **amphi 7** : files de priorité

# TD 7 = pale machine

**mardi 8 octobre**

- conditions habituelles d'un TD
  **mais pas d'accès à Internet**
- programme = blocs 1 à 6
- dictionnaire électronique autorisé

- les sujets des années précédentes sont sur Moodle

**penser à amener une feuille et un crayon**