École Polytechnique

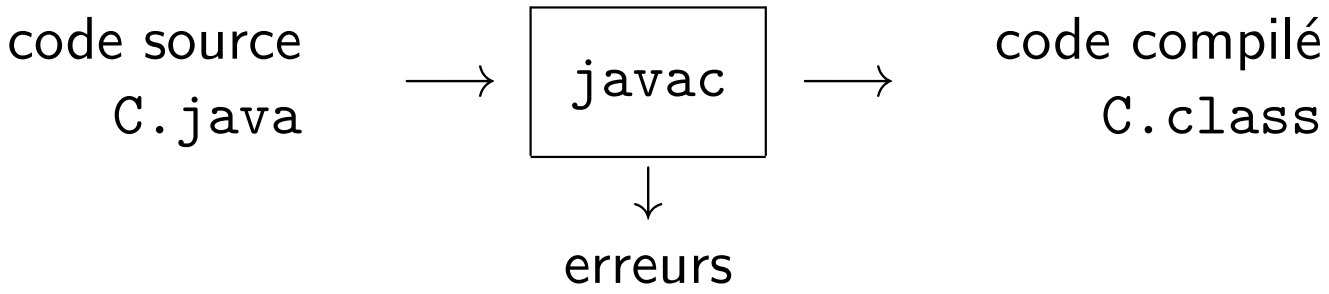
INF411

Les bases de la programmation et de l'algorithmique

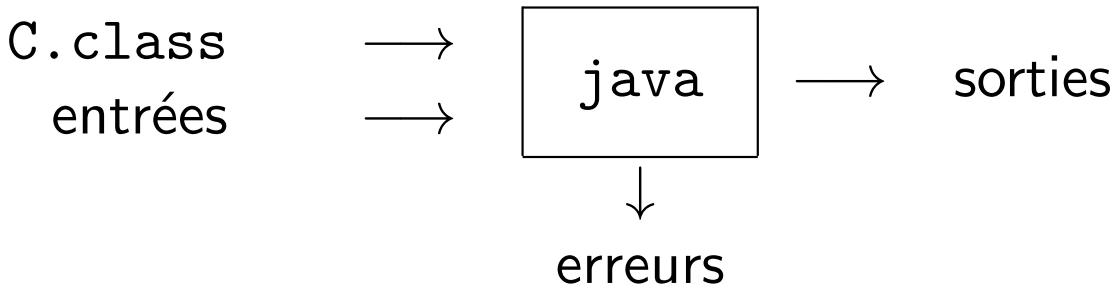
Jean-Christophe Filliâtre

tableaux redimensionnables, listes chaînées, piles

- **compilation**



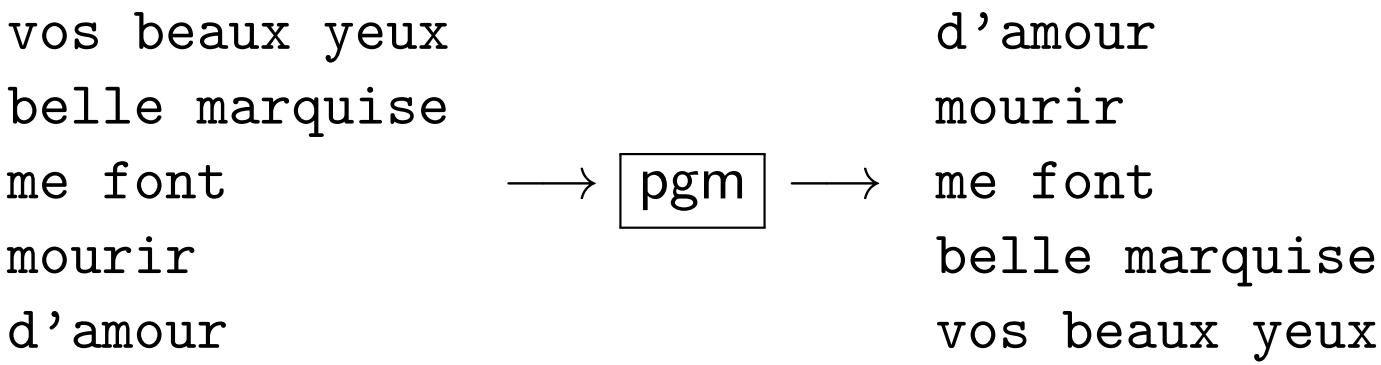
- **exécution**



- **entrées**
  - la ligne de commande (`main(String[] args)`)
  - l'entrée standard (`System.in`)
  - des fichiers
  - le générateur aléatoire
  - etc.
- **sorties**
  - la sortie standard (`System.out`)
  - la sortie d'erreur (`System.err`)
  - des fichiers
  - etc.

écrivons un programme qui lit des chaînes sur son entrée,
une par ligne,
et les ré-affiche ensuite dans l'ordre inverse

exemple


vos beaux yeux
belle marquise
me font
mourir
d'amour
pgm
d'amour
mourir
me font
belle marquise
vos beaux yeux

lire les lignes de l'entrée

```
public static void main(String[] args) throws IOException {

  BufferedReader r =
    new BufferedReader(new InputStreamReader(System.in));

  while (true) {
    String s = r.readLine();
    if (s == null) break;
    ...
  }
}
```

difficulté

on ne connaît pas le nombre total de lignes qui seront lues

(sans quoi, un simple tableau suffirait)

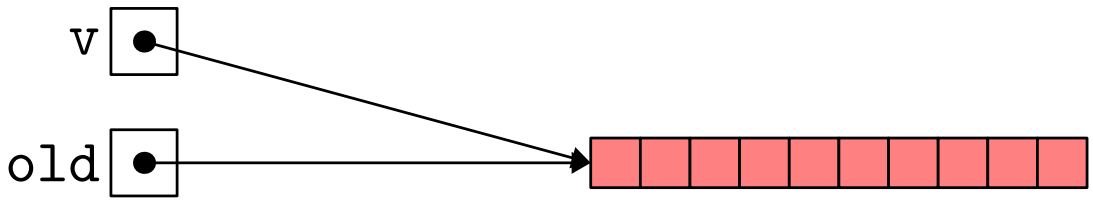
une première solution

utilisons un tableau néanmoins

```
String[] v = new String[10];
int size = 0;
while (true) {
  String s = r.readLine();
  if (s == null) break;
  v[size++] = s;
}
for (int i = size - 1; i >= 0; i--)
  System.out.println(v[i]);
```

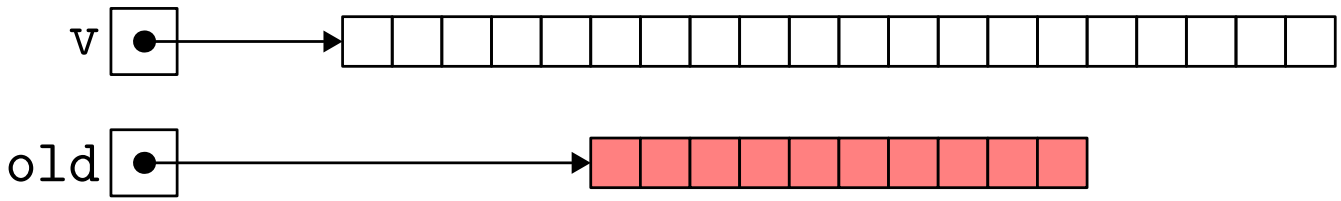
et agrandissons-le à la volée

```
...
while (true) {
  String s = r.readLine();
  if (s == null) break;
  if (size == v.length) {
    String[] old = v;
    v = new String[size + 10];
    System.arraycopy(old, 0, v, 0, size);
  }
  v[size++] = s;
}
...
```



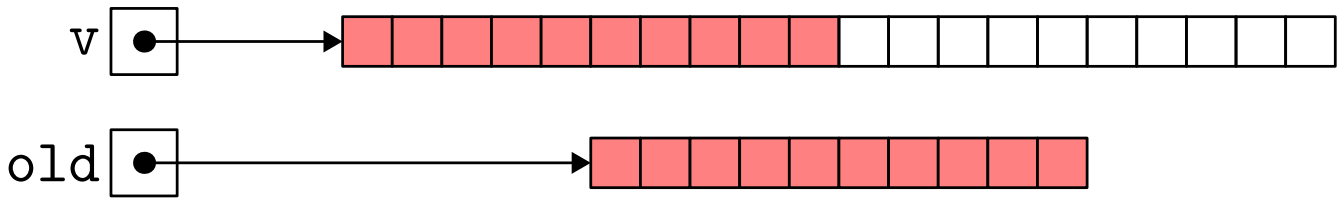
et agrandissons-le à la volée

```
...
while (true) {
  String s = r.readLine();
  if (s == null) break;
  if (size == v.length) {
    String[] old = v;
    v = new String[size + 10];
    System.arraycopy(old, 0, v, 0, size);
  }
  v[size++] = s;
}
...
```



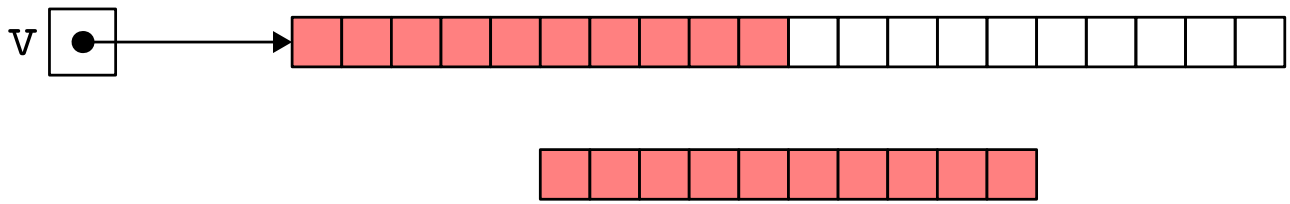
et agrandissons-le à la volée

```
...
while (true) {
  String s = r.readLine();
  if (s == null) break;
  if (size == v.length) {
    String[] old = v;
    v = new String[size + 10];
    System.arraycopy(old, 0, v, 0, size);
  }
  v[size++] = s;
}
...
```

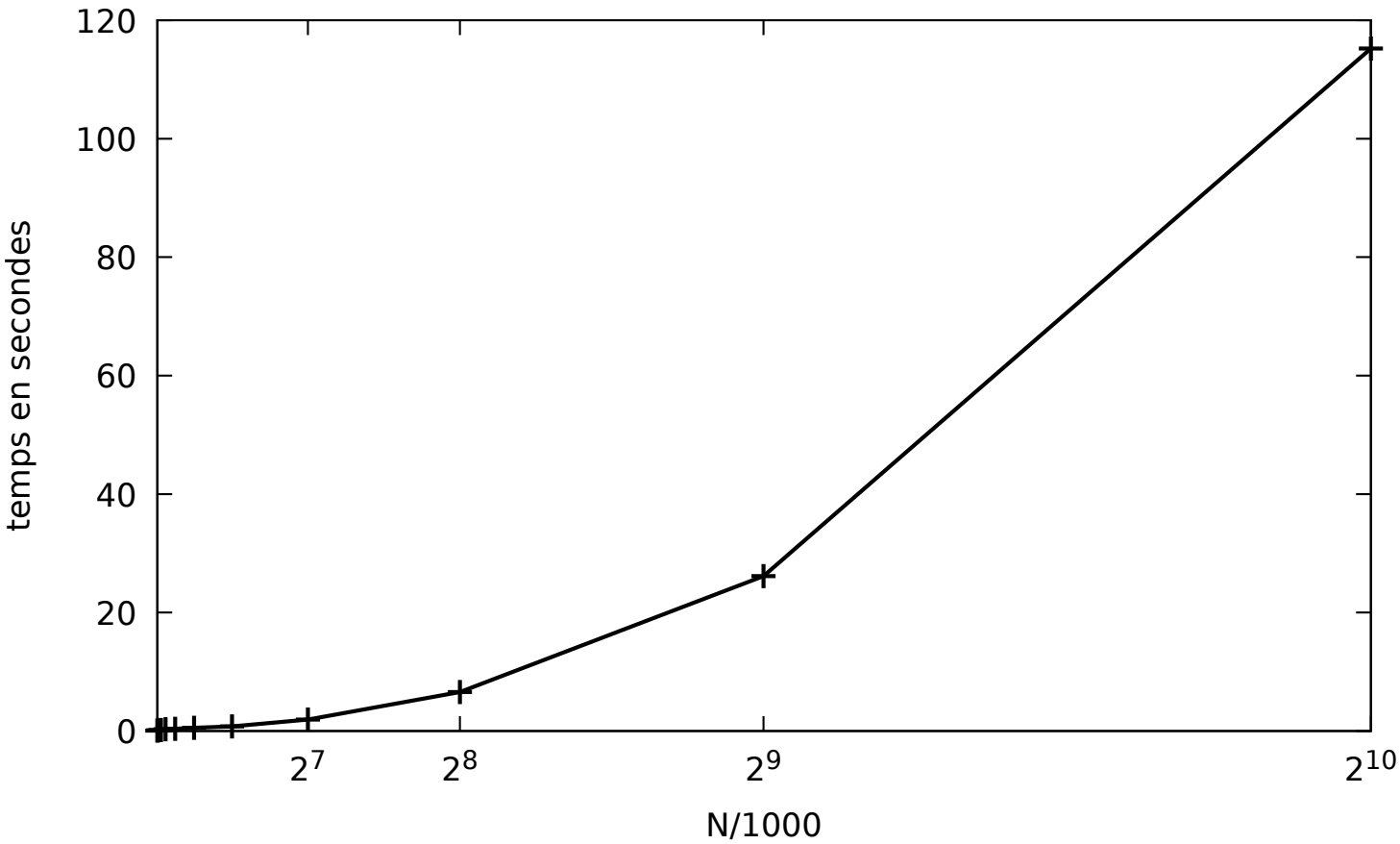


et agrandissons-le à la volée

```
...
while (true) {
  String s = r.readLine();
  if (s == null) break;
  if (size == v.length) {
    String[] old = v;
    v = new String[size + 10];
    System.arraycopy(old, 0, v, 0, size);
  }
  v[size++] = s;
}
...
```



expérimentalement



la première fois que le tableau est agrandi, on copie 10 éléments
la deuxième fois que le tableau est agrandi, on copie 20 éléments
la troisième fois que le tableau est agrandi, on copie 30 éléments
etc.

au total, le coût pour N lignes est

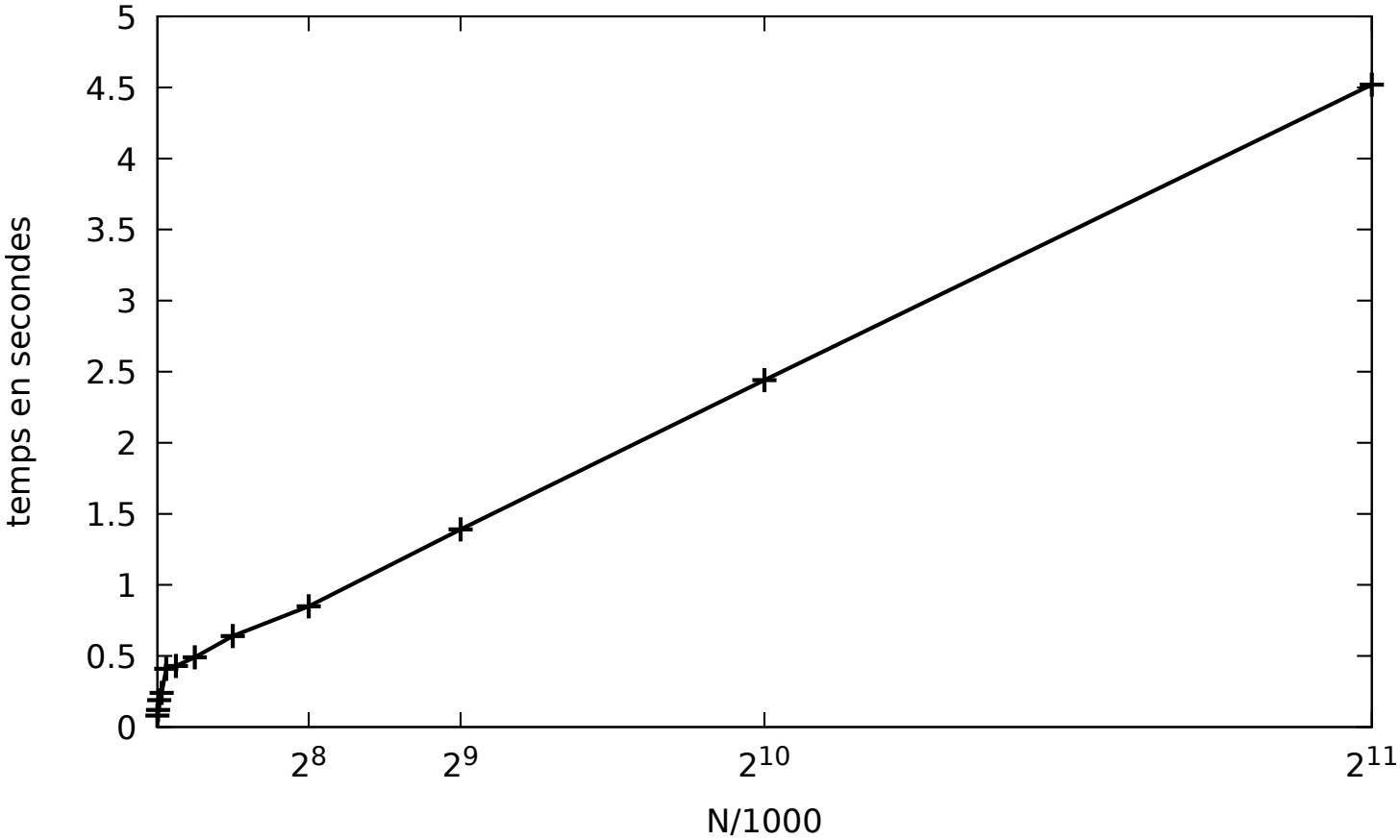
$$10 \sum_{i<N/10} i = O(N^2)$$

c'est **quadratique**

plutôt que d'ajouter 10 éléments, **doublons** la taille du tableau

```
...
if (size == v.length) {
  String[] old = v;
  v = new String[2 * size];
  System.arraycopy(old, 0, v, 0, size);
}
...
```

expérimentalement bien meilleur



la première fois que le tableau est agrandi, on copie $2^0 \times 10$ éléments
la deuxième fois que le tableau est agrandi, on copie $2^1 \times 10$ éléments
la troisième fois que le tableau est agrandi, on copie $2^2 \times 10$ éléments
etc.

en posant $k = \lfloor \log_2(N/10) \rfloor$, on a au total

$$10 \times \sum_{i=0}^{k} 2^i = 10 \times (2^{k+1} - 1) = O(N)$$

c'est **linéaire**

ce n'est pas très joli cependant

```
String[] v = new String[10];
int size = 0;
while (true) {
  String s = r.readLine();
  if (s == null) break;
  if (size == v.length) {
    String[] old = v;
    v = new String[2 * size];
    System.arraycopy(old, 0, v, 0, size);
  }
  v[size++] = s;
}
for (int i = size - 1; i >= 0; i--)
  System.out.println(v[i]);
```

nous venons d'identifier la notion de **tableau redimensionnable**

isolons-la dans une classe à part, dont l'interface est

```
    int size();
 String get(int i);
   void add(String s);
```

```
class ResizableArray {
  private String[] elts;
  private int size;
```

les champs sont privés ; c'est le principe d'**encapsulation**

cela permet (entre autres) de maintenir l'invariant

$$0 \leq \texttt{this.size} \leq \texttt{this.elts.length}$$

```
ResizableArray() {
  this.elts = new String[10];
  this.size = 0;
}
```

la valeur 10 est bien sûr arbitraire

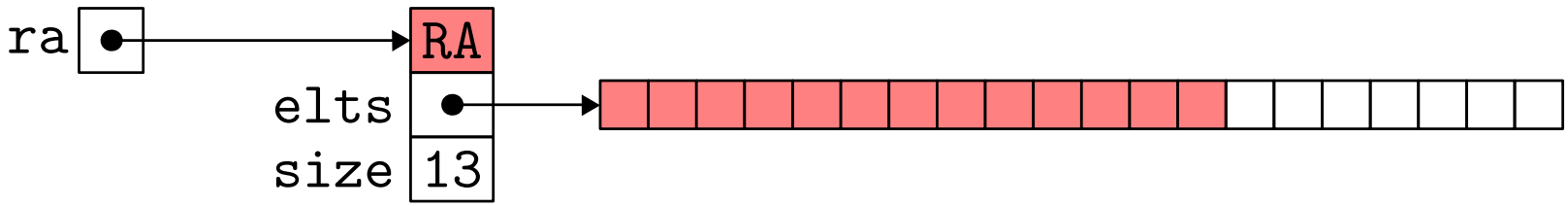
le constructeur pourrait prendre une valeur alternative en argument

```
  int size() { return this.size; }

  String get(int i) { return this.elts[i]; }

  void add(String s) {
    if (this.size == this.elts.length) {
      String[] old = this.elts;
      this.elts = new String[2 * this.size];
      System.arraycopy(old, 0, this.elts, 0, this.size);
    }
    this.elts[this.size++] = s;
  }
}
```

pour bien faire, la méthode `get` devrait vérifier la validité de `i` (cf poly)

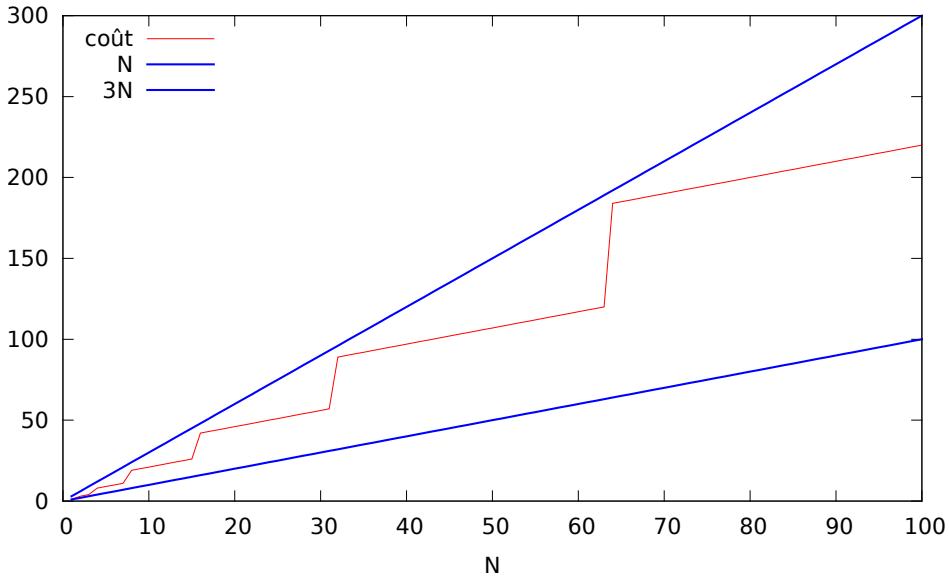
le programme redevient lisible

```
ResizableArray ra = new ResizableArray();
while (true) {
  String s = r.readLine();
  if (s == null) break;
  ra.add(s);
}
for (int i = ra.size() - 1; i >= 0; i--)
  System.out.println(ra.get(i));
```


ra
RA
elts
size
13

la complexité reste la même : $O(N)$ au total

certains appels à add ont un coût constant (un test + une affectation)
d'autres un coût plus élevé (copie de tous les éléments)



tout se passe comme si chaque appel à add avait un coût constant $O(1)$
on dit que add a une **complexité amortie** $O(1)$

en pratique, on préférera écrire une version **générique**,
paramétrée par le type `E` des éléments (voir le poly, section 3.4.5)

```
class ResizableArray<E> {
  private E[] elts;
  private int size;
  ResizableArray() { ... }
  int size() { ... }
  void add(E x) { ... }
  E get(int i) { ... }
}
```

cette classe existe dans la bibliothèque : c'est `java.util.Vector<E>`
(ou encore `java.util.ArrayList<E>`)

la classe Vector<E> de Java

```
class Vector<E> {
       Vector()           // un tableau de taille 0
   int size()             // c'est un tableau
     E get(int i)
  void set(int i, E x)
  void setSize(int n)     // dont on peut changer la taille
  ...
}
```

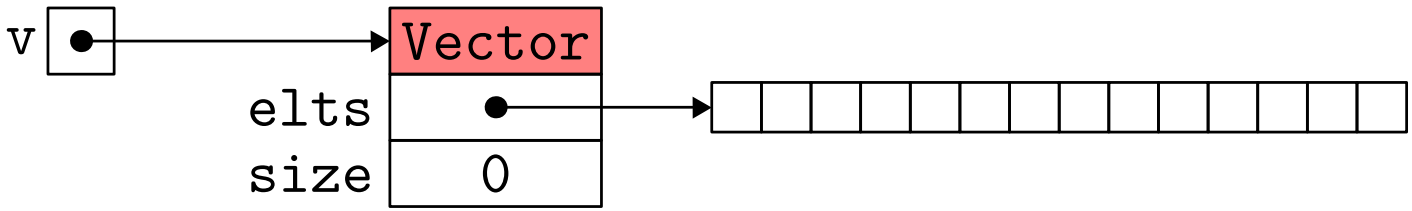
fournit également add, qui augmente la taille de 1 et ajoute l'élément dans la dernière case

attention, il y a un autre constructeur, prenant un entier en argument,

```
Vector<E> v = new Vector<E>(n);
```

mais il construit aussi un tableau redimensionnable **de taille 0**

la valeur `n` passée en argument est la **capacité**
i.e. la taille du tableau interne



si on veut un tableau redimensionnable de taille `n`,

- soit on lui donne la taille `n` avec `setSize`

```
Vector<E> v = new Vector<E>(n);
v.setSize(n);
```

- soit on lui ajoute successivement `n` éléments

```
Vector<E> v = new Vector<E>(n);
for (int i = 0; i < n; i++)
  v.add(...);
```

les “listes” de Python

ne sont rien d'autres que des tableaux redimensionnables

```
v = []
v.append(1)
v.append(2)
v[0] = v[1] + 1
print(v.pop())
```

`v` contient `[1, 2]`
`v` contient `[3, 2]`
`v` contient `[3]`

en particulier, `append` et `pop` ont une complexité amortie $O(1)$

mais d'autres opérations, comme par exemple une extraction `v[i:j]`, coûtent beaucoup plus cher !

une deuxième solution

accumuler les lignes lues dans une **liste simplement chaînée**

la dernière ligne lue étant ajoutée en tête

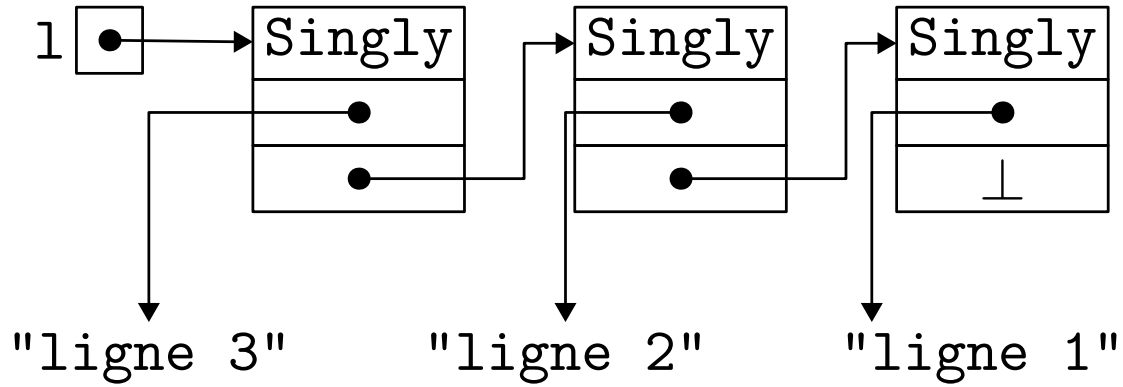
puis la parcourir pour les afficher toutes

```
class Singly {
  String element;
  Singly next;

  public Singly(String element, Singly next) {
    this.element = element;
    this.next = next;
  }
}
```

d'où le code

```
Singly l = null;
while (true) {
  String s = r.readLine();
  if (s == null) break;
  l = new Singly(s, l);
}
```

après la lecture de trois lignes, on a



affichage de la liste

```
while (l != null) {
  System.out.println(l.element);
  l = l.next;
}
```

remarque : on peut aussi écrire

```
for (; l != null; l = l.next)
  System.out.println(l.element);
```

car la construction `for (e1; e2; e3) e4`
correspond à `e1; while(e2) { e4; e3; }`

affichage de la liste

```
while (l != null) {
  System.out.println(l.element);
  l = l.next;
}
```

remarque : on peut aussi écrire

```
for (; l != null; l = l.next)
  System.out.println(l.element);
```

car la construction `for (e1; e2; e3) e4`
correspond à `e1; while(e2) { e4; e3; }`

là encore, on peut encapsuler la liste chaînée dans une classe

on peut choisir ici l'interface d'une **pile**

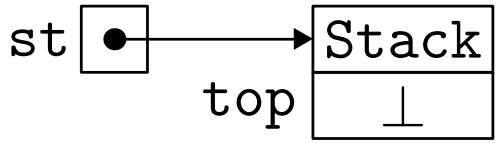
```
boolean isEmpty();        // pile vide ?
   void push(String s);   // empile
 String pop();            // dépile et renvoie
```

```
class Stack {
  private Singly top;

  Stack() { this.top = null; }

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

  void push(String s) { this.top = new Singly(s, this.top); }

  String pop() {
    if (this.isEmpty()) throw new NoSuchElementException();
    String r = this.top.element;
    this.top = this.top.next;
    return r;
  }
}
```

(en pratique, on écrit une version générique ; cf le poly, section 4.6)

la pile vide, ce n'est pas `null`

```
Stack st = new Stack();
```



donc on n'écrit pas

```
if (st == null) ...
```

mais

```
if (st.isEmpty()) ...
```

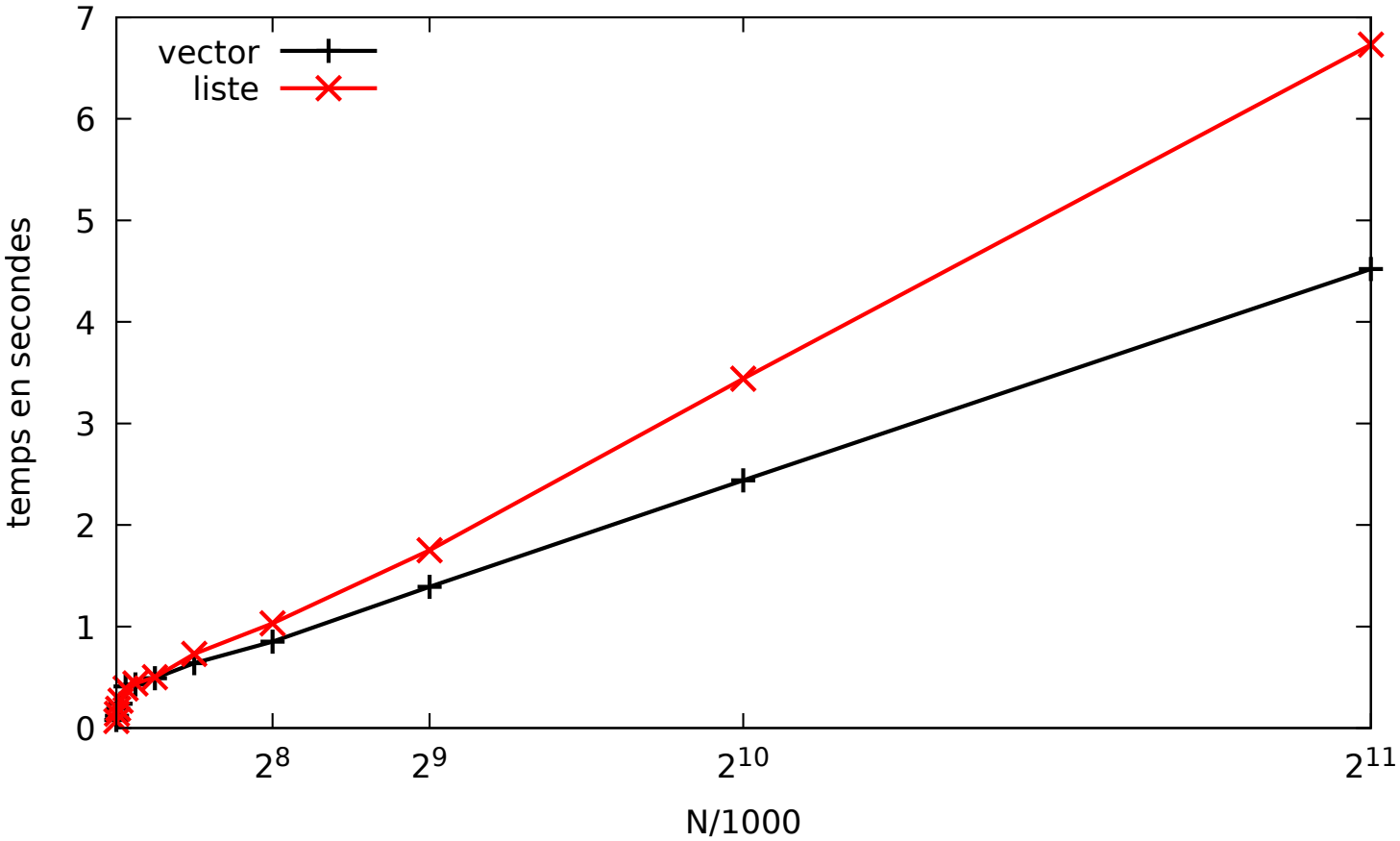
les opérations `isEmpty`, `push` et `pop` sont clairement en temps constant

d'où un code linéaire

```
Stack st = new Stack();
while (true) {
  String s = r.readLine();
  if (s == null) break;
  st.push(s);
}
while (!st.isEmpty())
  System.out.println(st.pop());
```

chaque boucle ayant une complexité $O(N)$

on le vérifie expérimentalement



deux solutions à notre problème

- tableau redimensionnable
- liste chaînée

en particulier, ce sont deux façons d'obtenir une **pile**

- tableau redimensionnable : on ajoute/retire en fin de tableau
- liste chaînée : on ajoute/retire en tête de liste

comparaison (sur notre problème)

en **temps**

- même complexité totale $O(N)$
- léger avantage au tableau redimensionnable (facteur constant)
- seule la liste offre `push` en $O(1)$

en **espace**

- le tableau redimensionnable est plus compact
- la liste chaînée sollicite plus le gestionnaire mémoire (GC)

la bibliothèque fournit des piles dans `java.util.Stack<E>`

utilise un tableau redimensionnable, pas une liste

une troisième solution

on peut écrire une méthode **récursive**

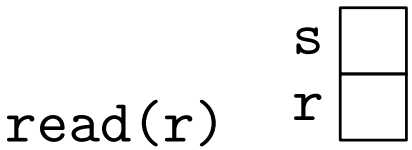
```
static void read(BufferedReader r) throws IOException {
  String s = r.readLine();
  if (s == null) return;
  read(r);
  System.out.println(s);
}
```

il suffit de l'appeler depuis `main`

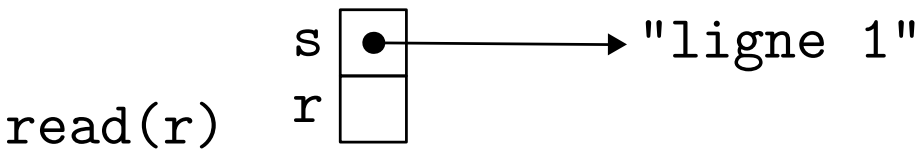
```
public static void main(String[] args) throws IOException {
  BufferedReader r = ...;
  read(r);
}
```

il n'y a plus aucune structure de données explicite !

il y a bien une structure de données, c'est la **pile d'appels**

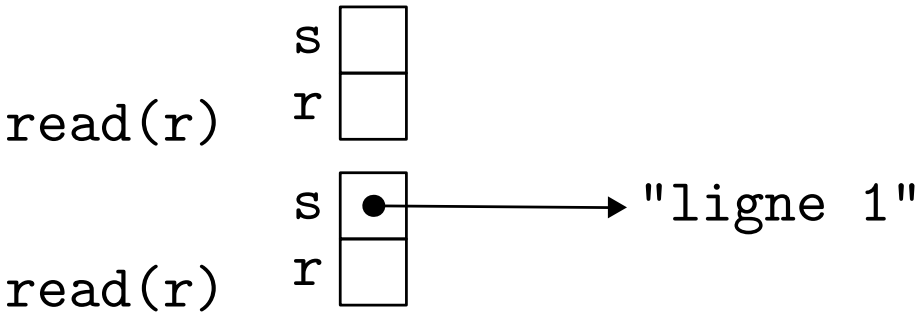



```
static void read(BufferedReader r) throws IOException {
  String s = r.readLine();
  if (s == null) return;
  read(r);
  System.out.println(s);
}
```



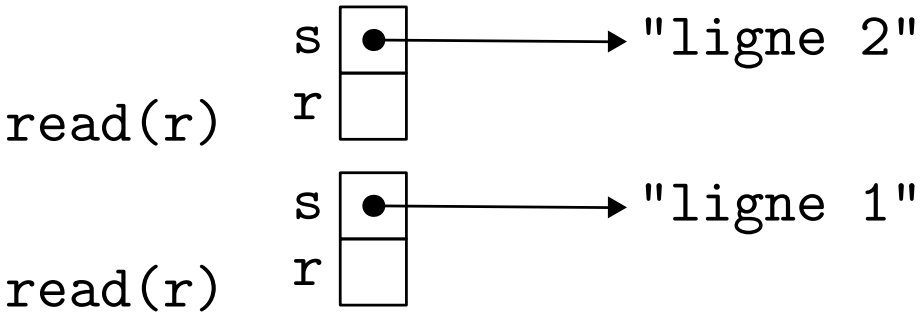

```
static void read(BufferedReader r) throws IOException {
  String s = r.readLine();
  if (s == null) return;
  read(r);
  System.out.println(s);
}
```

pile d'appels



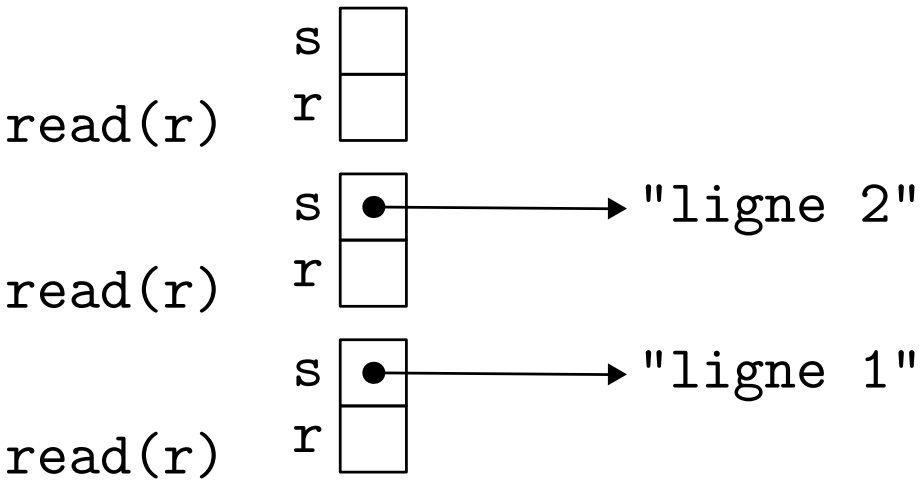

```
static void read(BufferedReader r) throws IOException {
  String s = r.readLine();
  if (s == null) return;
  read(r);
  System.out.println(s);
}
```

pile d'appels



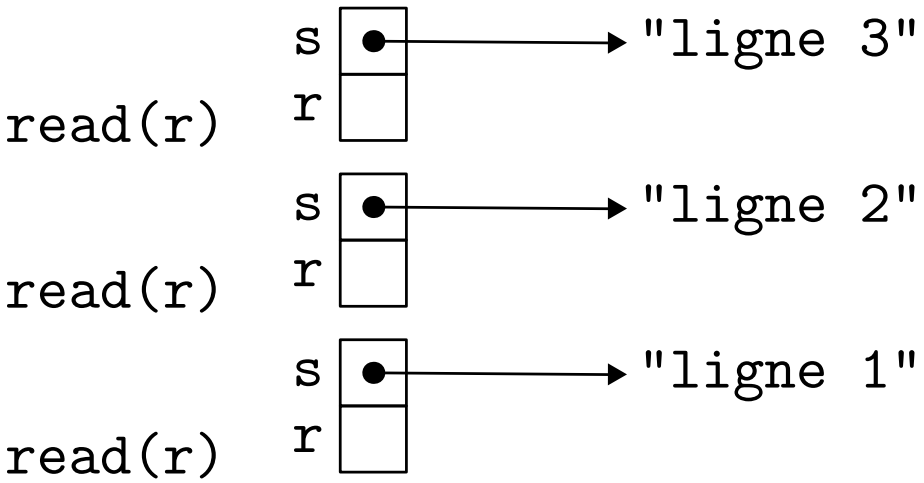

```
static void read(BufferedReader r) throws IOException {
  String s = r.readLine();
  if (s == null) return;
  read(r);
  System.out.println(s);
}
```

pile d'appels



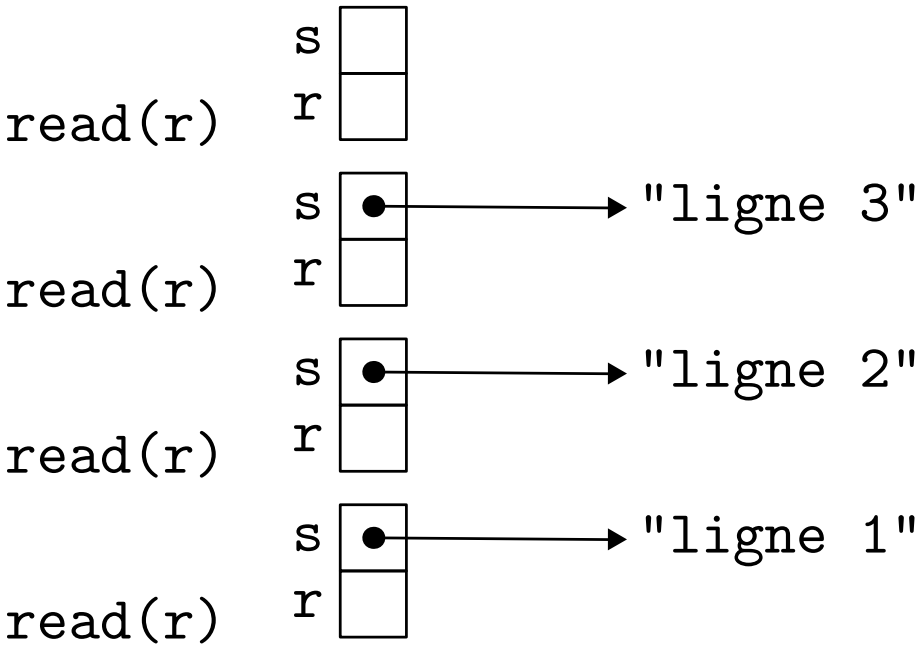

```
static void read(BufferedReader r) throws IOException {
  String s = r.readLine();
  if (s == null) return;
  read(r);
  System.out.println(s);
}
```

pile d'appels



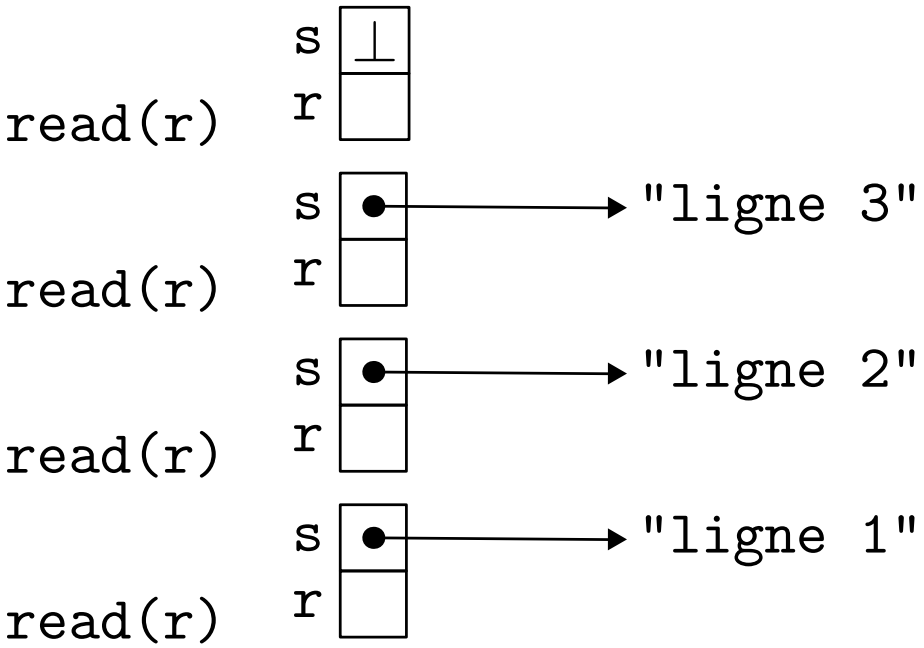

```
static void read(BufferedReader r) throws IOException {
  String s = r.readLine();
  if (s == null) return;
  read(r);
  System.out.println(s);
}
```

pile d'appels



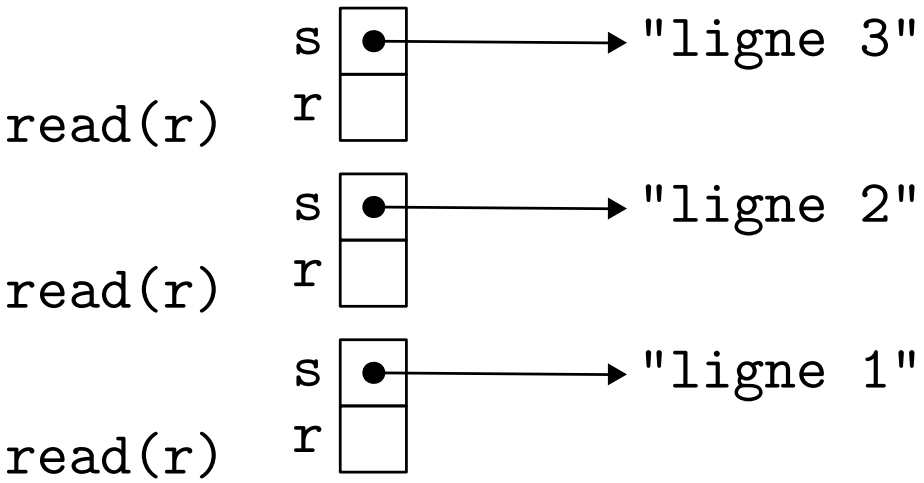

```
static void read(BufferedReader r) throws IOException {
  String s = r.readLine();
  if (s == null) return;
  read(r);
  System.out.println(s);
}
```

pile d'appels




```
static void read(BufferedReader r) throws IOException {
  String s = r.readLine();
  if (s == null) return;
  read(r);
  System.out.println(s);
}
```

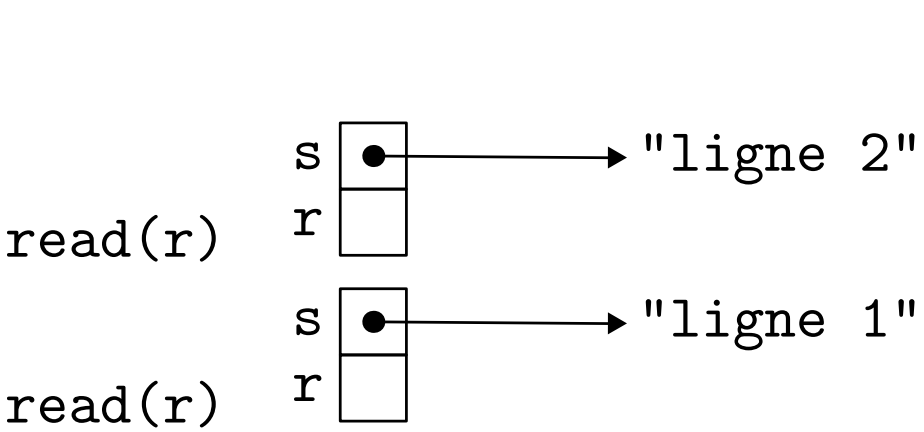
pile d'appels




```
ligne 3
```

```
static void read(BufferedReader r) throws IOException {
  String s = r.readLine();
  if (s == null) return;
  read(r);
  System.out.println(s);
}
```

pile d'appels

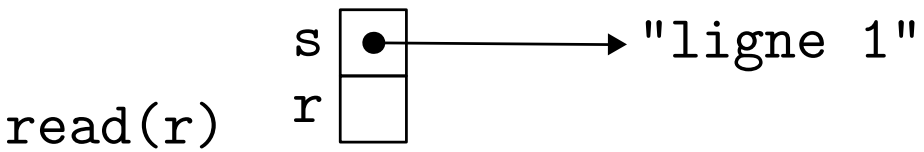



```
ligne 3
ligne 2
```

```
static void read(BufferedReader r) throws IOException {
  String s = r.readLine();
  if (s == null) return;
  read(r);
  System.out.println(s);
}
```

pile d'appels

```
ligne 3
ligne 2
ligne 1
```




```
static void read(BufferedReader r) throws IOException {
  String s = r.readLine();
  if (s == null) return;
  read(r);
  System.out.println(s);
}
```

pile d'appels

```
ligne 3
ligne 2
ligne 1
```

```
static void read(BufferedReader r) throws IOException {
  String s = r.readLine();
  if (s == null) return;
  read(r);
  System.out.println(s);
}
```

la pile est limitée (de l'ordre de 1 Mo)

pour $N = 10\,500$ on obtient une erreur

```
Exception in thread "main" java.lang.StackOverflowError
        at amphis.Amphi1rec.read(Amphi1.java:240)
        at amphis.Amphi1rec.read(Amphi1.java:239)
        at amphis.Amphi1rec.read(Amphi1.java:239)
        at amphis.Amphi1rec.read(Amphi1.java:239)
        ...
```

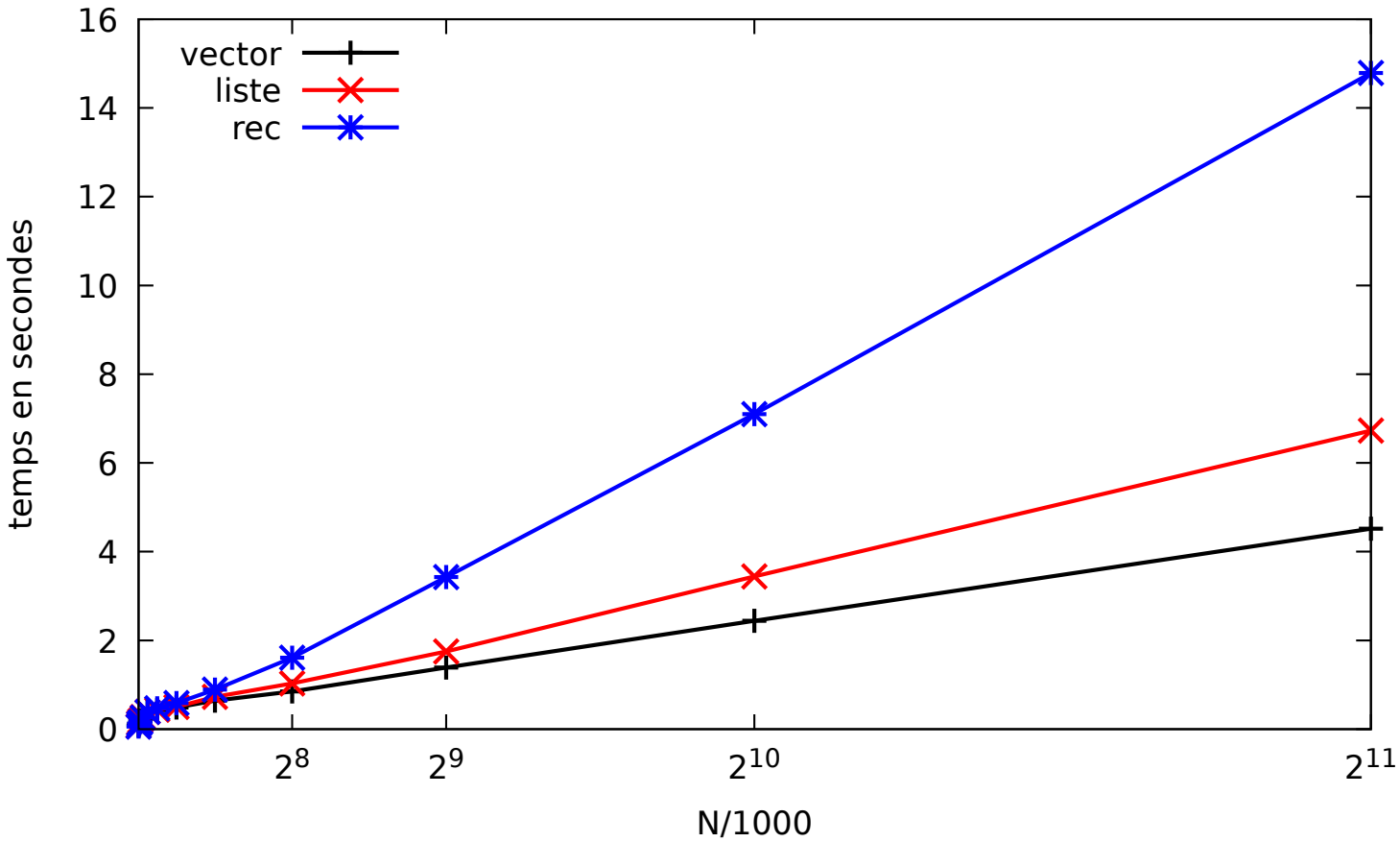
on peut augmenter la taille de pile

pour une pile de 100 Mo

```
java -Xss100M Pgm ...
```

(dans Eclipse : `Run Configurations` → `Arguments` → `VM arguments`)

toujours linéaire, mais un peu plus cher



modification d'une liste

```
class Singly {
  int element;
  Singly next;
  ...
}
```

on peut modifier une liste *a posteriori*

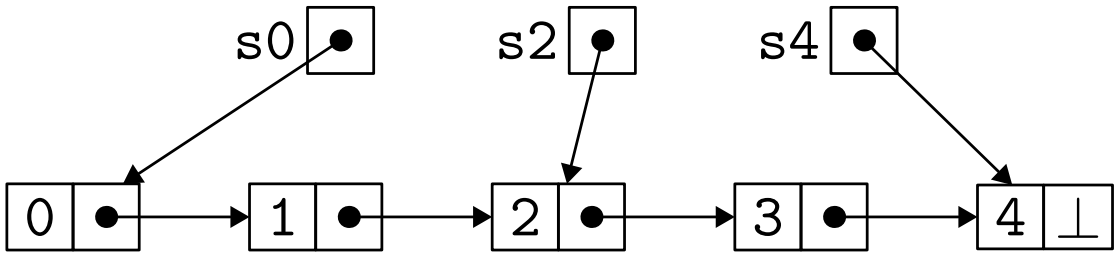
- soit son contenu

```
x.element = ...;
```

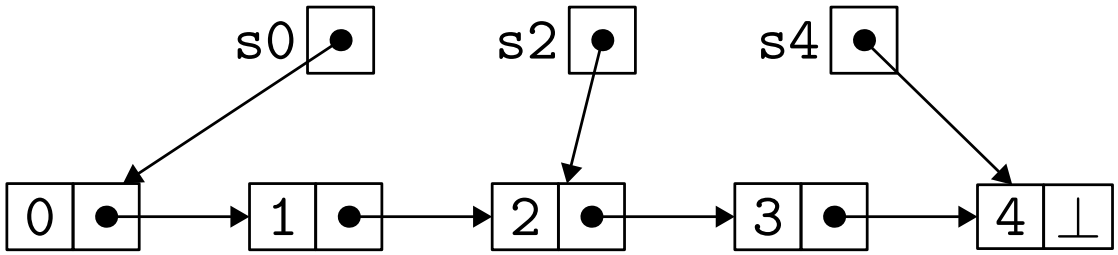
- soit sa structure

```
x.next = ...;
```

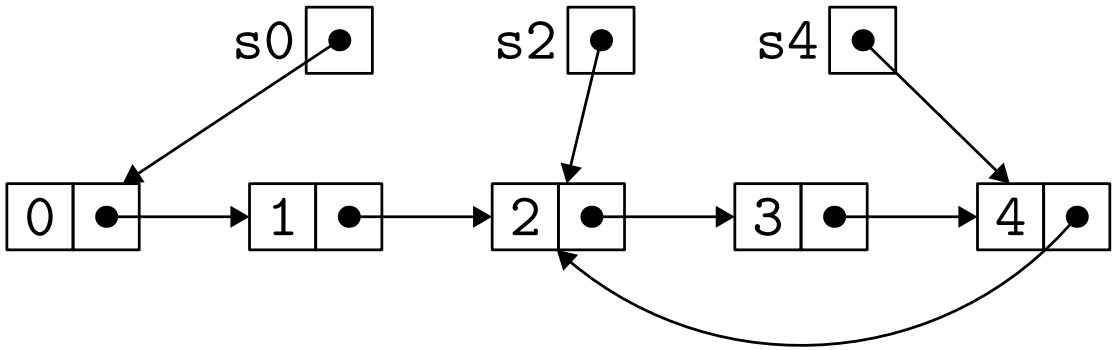
```
Singly s4 = new Singly(4, null);
Singly s2 = new Singly(2, new Singly (3, s4));
Singly s0 = new Singly(0, new Singly (1, s2));
```



```
Singly s4 = new Singly(4, null);
Singly s2 = new Singly(2, new Singly (3, s4));
Singly s0 = new Singly(0, new Singly (1, s2));
```



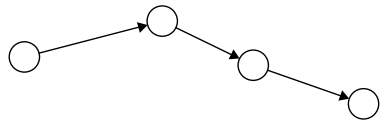

```
s4.next = s2;
```



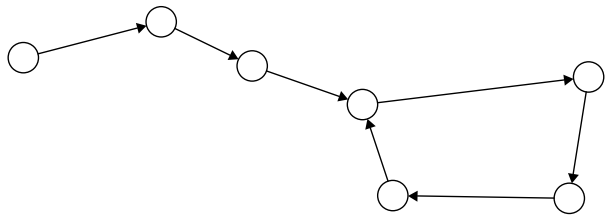
on vient de construire une **liste cyclique**

d'une manière générale, une liste de la classe `Singly` est

- soit linéaire *i.e.* terminée par `null`



- soit cyclique à partir d'un certain rang



un parcours de liste du type

```
while (l != null) { ...; l = l.next; }
```

ne termine plus sur une liste cyclique

question intéressante : comment détecter qu'une liste est cyclique ?

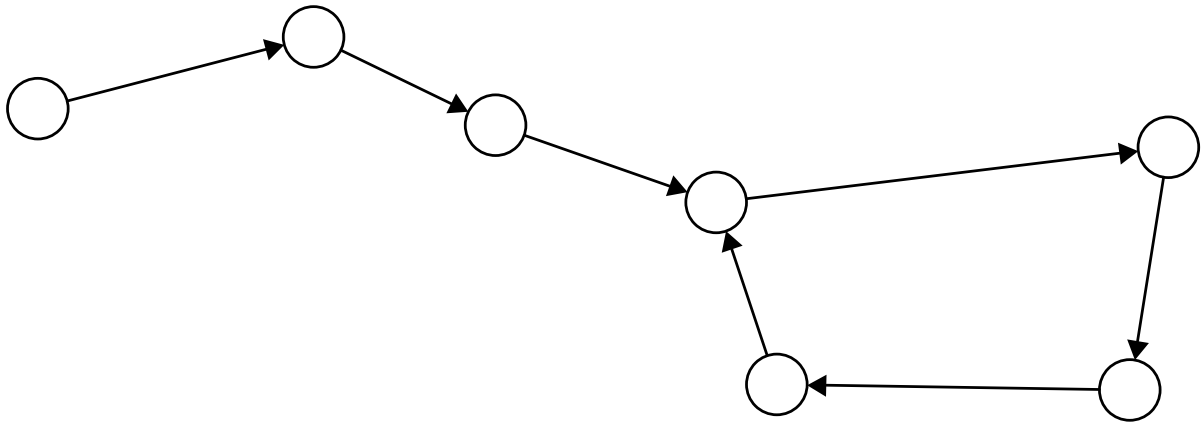
```
static boolean hasCycle(Singly s) {
  ...
}
```

un algorithme de détection de cycle dû à Floyd
connu sous le nom d'**algorithme du lièvre et de la tortue**

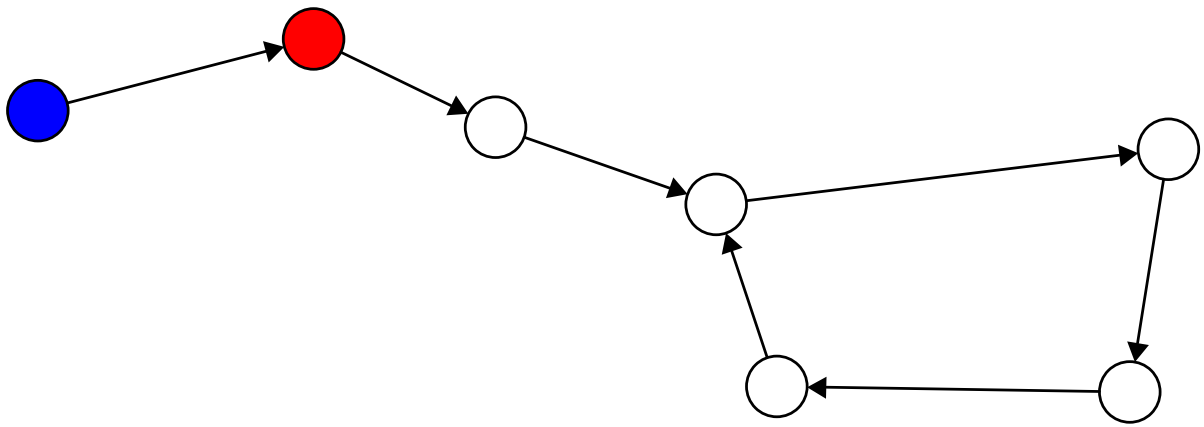
idée :

- on parcourt la liste simultanément à vitesses 1 (tortue) et 2 (lièvre)
- si la liste n'est pas cyclique, le lièvre atteint `null`
- si la liste est cyclique, la tortue et le lièvre se rencontrent

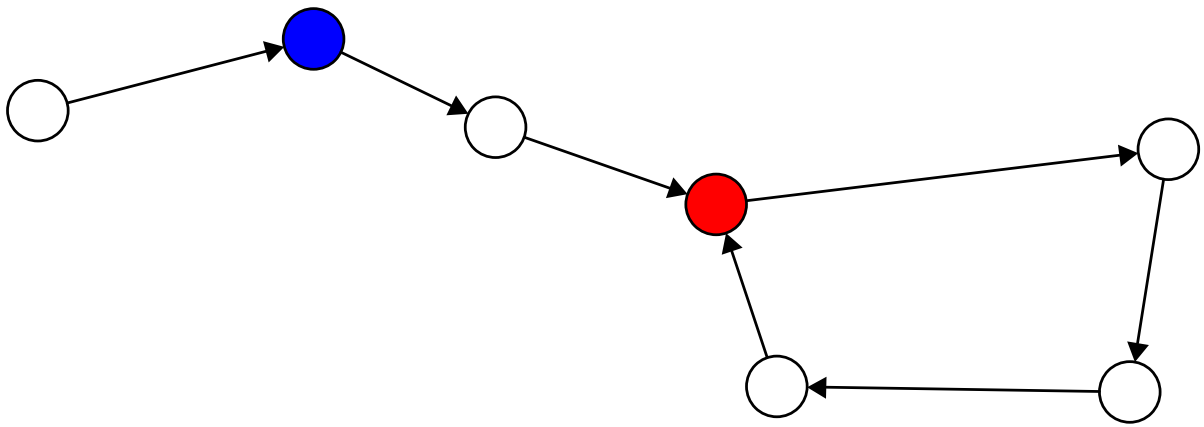
le lièvre et la tortue



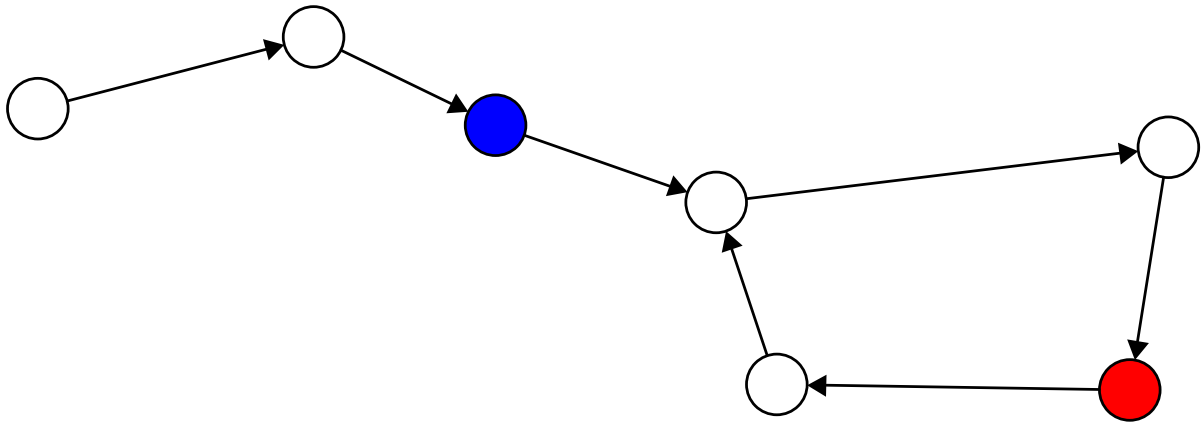
le lièvre et la tortue



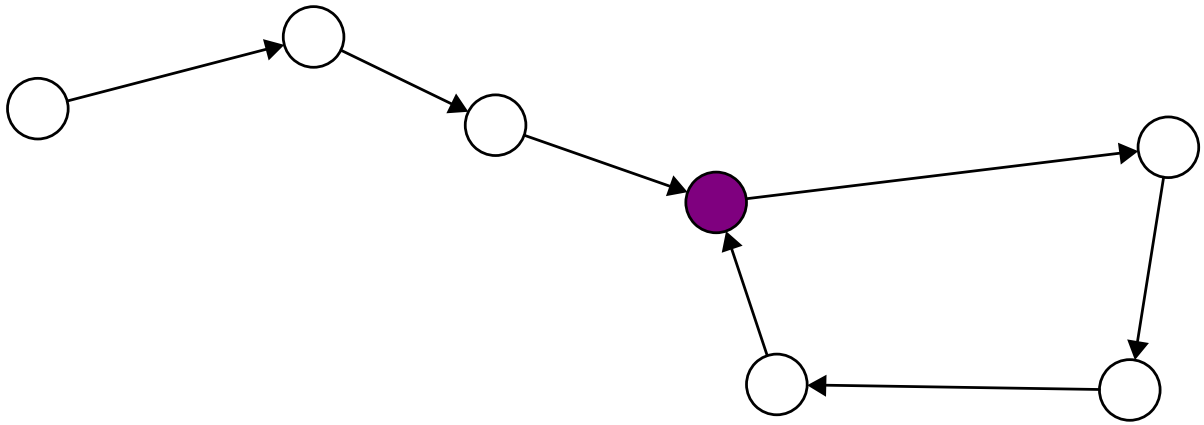
le lièvre et la tortue



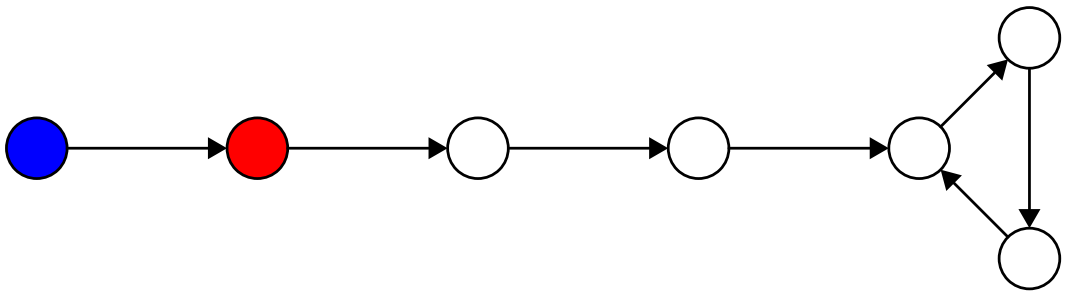
le lièvre et la tortue



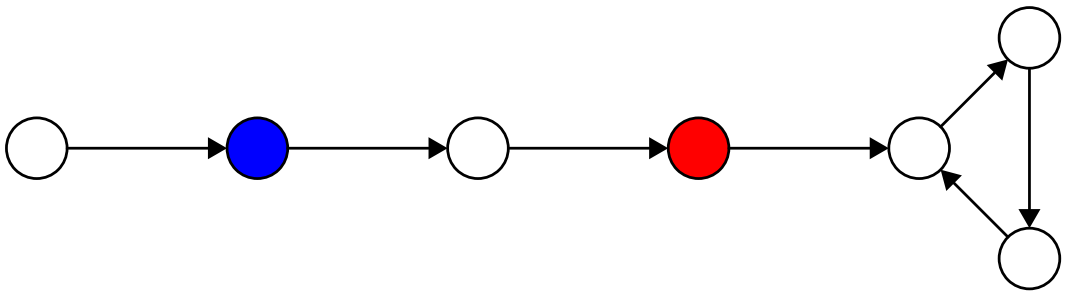
le lièvre et la tortue



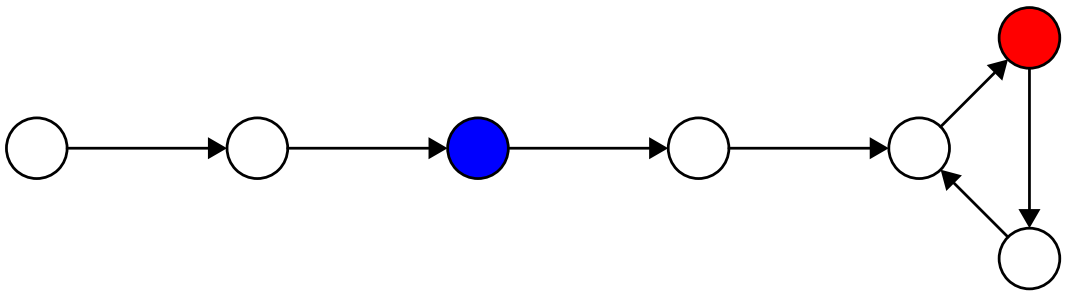
le lièvre et la tortue



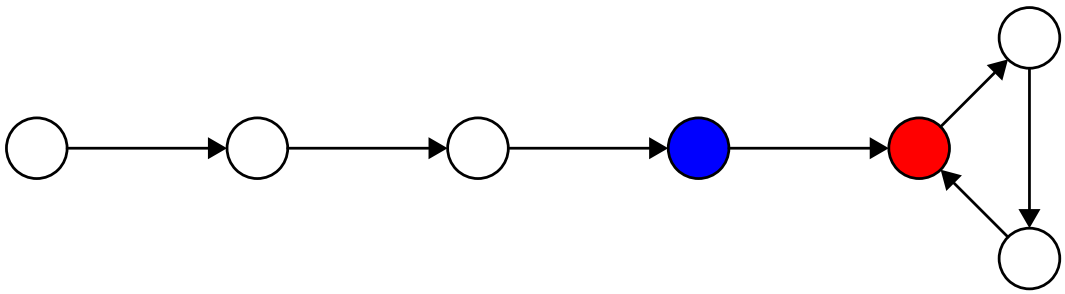
le lièvre et la tortue



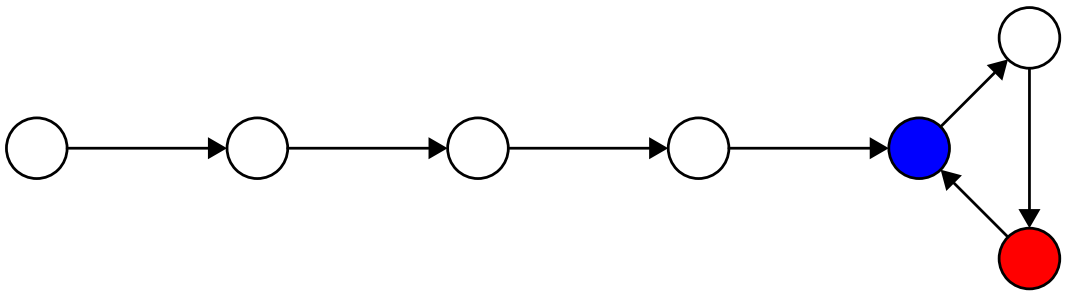
le lièvre et la tortue



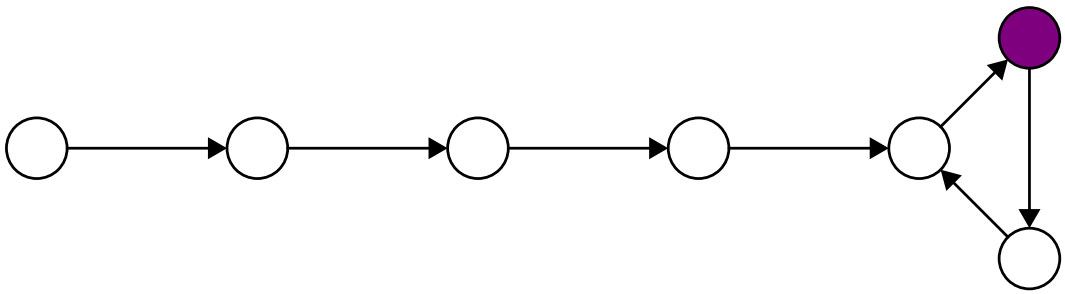
le lièvre et la tortue



le lièvre et la tortue

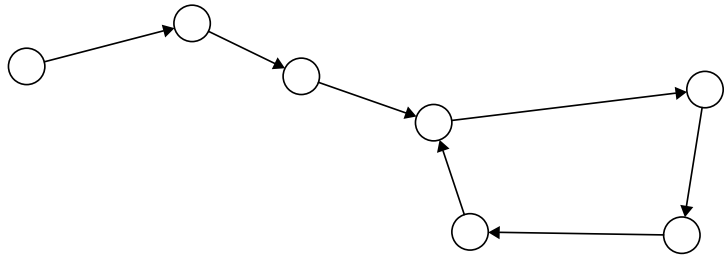


le lièvre et la tortue



```
static boolean hasCycle(Singly s) {
  if (s == null) return false;
  Singly tortoise = s, hare = s.next;
  while (tortoise != hare) {
    if (hare == null) return false;
    hare = hare.next;
    if (hare == null) return false;
    hare = hare.next;
    tortoise = tortoise.next;
  }
  return true;
}
```

- si la liste se termine par `null`, c'est clair
- si la liste est cyclique
  - tant que la tortue est hors du cycle, elle s'en approche à chaque pas
  - une fois la tortue dans le cycle, le lièvre s'en approche à chaque pas



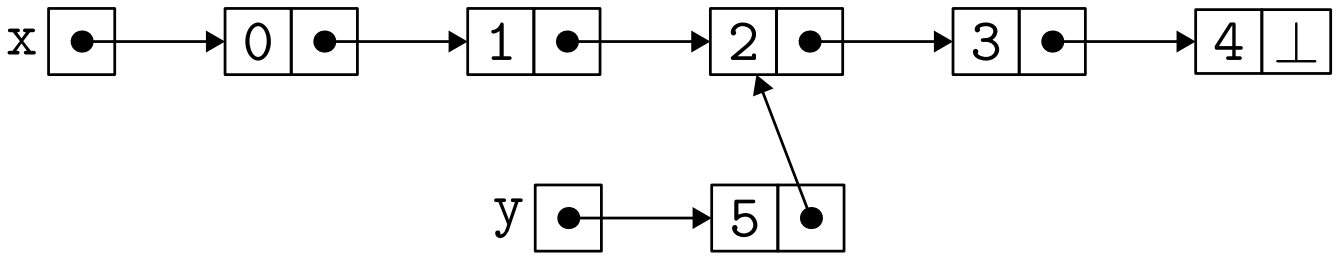
étonnamment efficace

- linéaire en temps
- constant en espace (deux variables)

nombreuses applications (cf le TD de cette semaine)

et si on ne **veut pas** modifier les listes ?

il y a effectivement un **risque** à modifier une liste *a posteriori*



modifier `x` (resp. `y`) pourra affecter `y` (resp. `x`)

on parle d'**aliasing** (plusieurs noms pour la même donnée,
par exemple `x.next.next` et `y.next` ici)

```
class Singly {
  final int element;
  final Singly next;
  ...
}
```

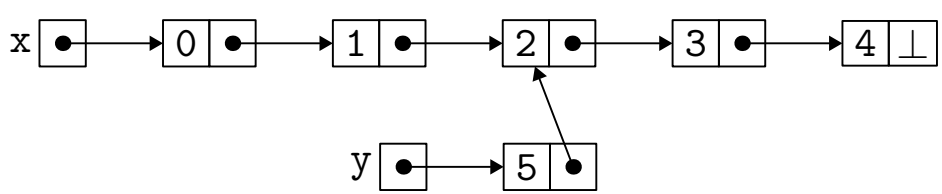
garantit que les champs ne sont pas affectés en dehors du constructeur

une fois construite, la liste ne peut plus être modifiée

on parle de structure **immuable**

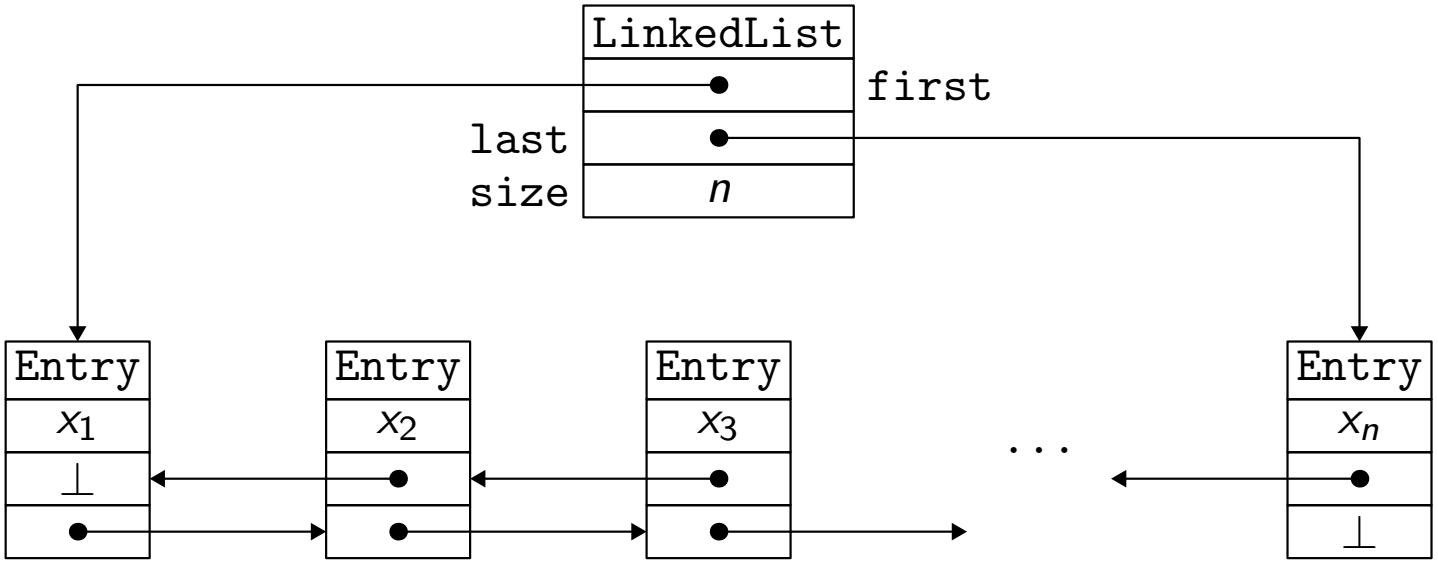
intérêts

- diminue les risques d'erreurs
- raisonnement facile sur le code (pas d'état)
- partage possible



la bibliothèque `LinkedList<E>`

liste **doublement chaînée** encapsulée dans un objet conservant des pointeurs sur le premier et le dernier élément



(le poly décrit les listes doublement chaînées en détail, section 4.5)

on peut opérer aux deux extrémités
(méthodes `addFirst`, `removeFirst`, `addLast`, `removeLast`, etc.)

parcours des éléments $x_1, \ldots, x_n$ avec la construction

```
for (E x : l) ...
```

on peut s'en servir comme

- une pile
- une file
- un sac
- etc.

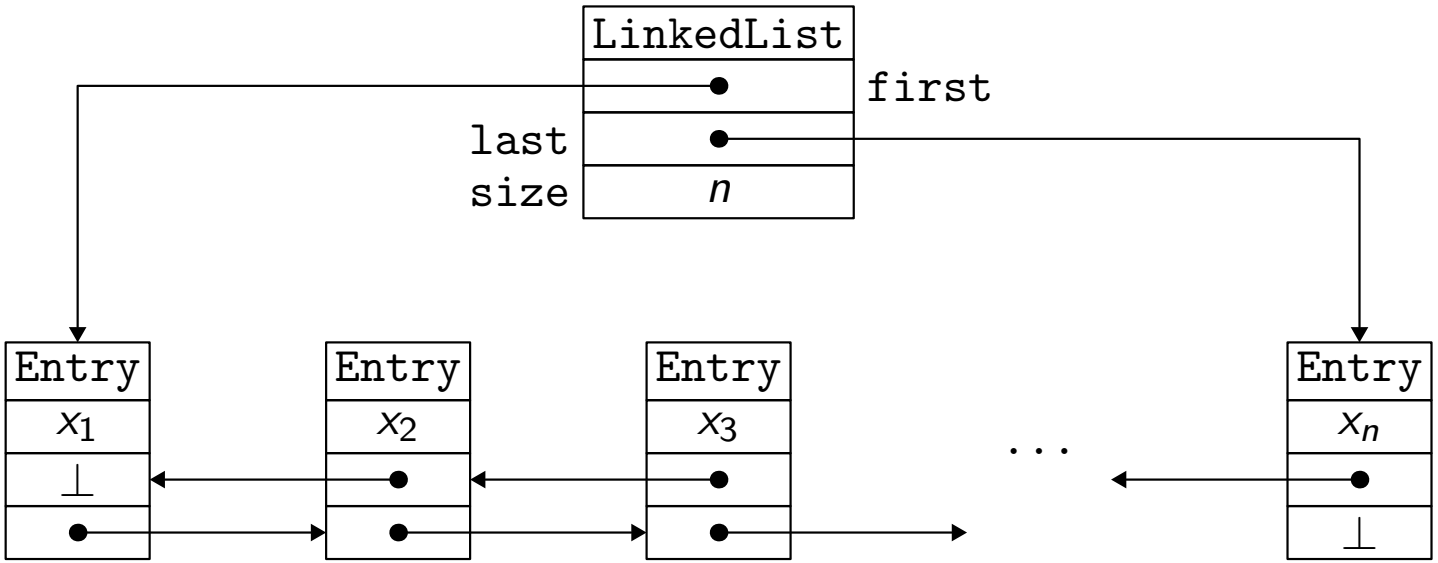
solution au problème initial avec `LinkedList`

```
LinkedList<String> l = new LinkedList<String>();
while (true) {
  String s = r.readLine();
  if (s == null) break;
  l.addFirst(s);
}
for (String s : l)
  System.out.println(s);
```

mais **surtout pas**

```
for (int i = 0; i < l.size(); i++)
  System.out.println(l.get(i));
```

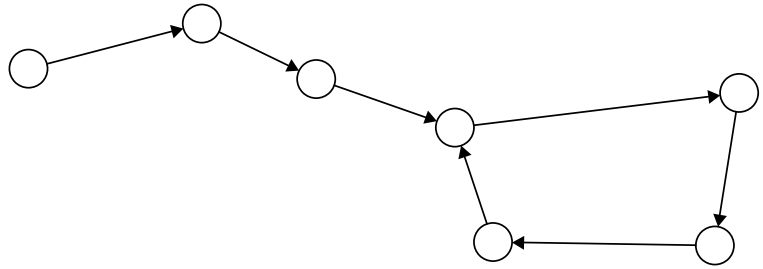
car `get(i)` coûte $O(i)$, d'où un coût total $O(N^2)$



le TD de cette semaine

mélanger un paquet de cartes
(en utilisant `LinkedList`)

déterminer si une partie de bataille est infinie



- **lire le poly**, chapitres
  1. Le langage Java
  2. Notions de complexité
  3. Tableaux
  4. Listes chaînées

  il y a des **exercices** dans le poly (avec solutions)
  suggestions : ex 20 p 54, ex 24 p 66

- **bloc 2** : tables de hachage, mémoïsation