Programmation et Algorithmique

INF 421, École Polytechnique

Philippe Baptiste et Luc Maranget

Avant-propos

Ce polycopié est utilisé pour le cours INF 421 intitulé *Les bases de la programmation et de l'algorithmique*. Ce cours fait suite au cours INF 311 *Introduction à l'informatique* et précède le cours INF 431 intitulé *Fondements de l'informatique*. L'objectif du cours est triple : (1) programmer en `java`, (2) maîtriser les bases de l'algorithmique sur des structures dynamiques (listes, arbres) et (3) introduire quelques notions fondamentales d'informatique comme les expressions régulières, les automates et l'analyse syntaxique.

Nous utilisions jusqu'à l'an passé le polycopié rédigé par Jean Berstel et Jean-Eric Pin. Pour prendre en compte les récents changements dans l'organisation des cours d'informatique de l'École, nous avons décidé de rédiger une nouvelle version de ce polycopié. Nous espérons qu'elle est aussi claire, aussi précise et aussi simple que la précédente. Nous avons d'ailleurs conservé de nombreux passages de J. Berstel et J.-E. Pin (en particulier pour les chapitres relatifs aux arbres).

Nous remercions nos collègues de l'École Polytechnique, et plus particulièrement ceux qui ont d'une manière ou d'une autre contribué au succès du cours INF 421 : Philippe Chassignet, Mathieu Cluzeau, Thomas Heide Clausen, Robert Cori, Xavier Dahan, Olivier Devillers, Thomas Houtmann, Philippe Jacquet, Fabien Laguillaumie, Fabrice Le Fessant, Laurent Mauborgne, David Monniaux, Sylvain Pradalier, Alejandro Ribes, Dominique Rossin, Éric Schost, Nicolas Sendrier, Jean-Jacques Lévy, François Morain, Laurent Viennot et Axelle Ziegler. Merci aussi à Christoph Dürr du LIX, qui a magnifiquement illustré la page de couverture[1] de ce polycopié.

Les auteurs peuvent être contactés par courrier électronique aux adresses suivantes :

`Philippe.Baptiste@polytechnique.fr`

`Luc.Maranget@inria.fr`

On peut aussi consulter la version `html` de ce polycopié ainsi que les pages des travaux dirigés sur le site `http ://www.enseignement.polytechnique.fr/informatique/inf421`

[1]Le 421 est un jeu de bar qui se joue au comptoir avec des dés.

Table des matières

Chapitre I

Listes

1 Structure dynamique, structure séquentielle

Les structures de données en algorithmique sont souvent complexes et de taille variable. Elles sont souvent aussi *dynamiques*, au sens où elles évoluent en forme et en taille en cours d'exécution d'un programme. Les listes sont l'exemple le plus simple d'une telle structure dynamique.

L'aspect dynamique renvoie aussi à la façon dont la mémoire nécessaire est allouée, la mémoire peut être allouée *dynamiquement* lors de l'exécution du programme, ou *statiquement* avant l'exécution du programme. L'allocation statique suppose que le compilateur arrive à connaître la taille de mémoire à allouer (c'est-a-dire à demander au système d'exploitation), cette taille est ensuite rangée dans le fichier exécutable produit et c'est le système d'exploitation qui alloue la mémoire demandée au moment de lancer le programme. En Pascal par exemple, la taille des tableaux globaux est donnée par des constantes, de sorte les tableaux globaux peuvent être et sont alloués statiquement.

```
program ;
var
  t : array [1..100] of integer ;
begin
  ⋮
end.
```

L'allocation dynamique de la mémoire existe aussi en Pascal (il existe une fonction **new**), mais elle est plus délicate qu'en Java, car en Pascal, comme dans beaucoup de langages, le programmeur doit rendre explicitement les cellules de mémoire dont il n'a plus besoin au système d'exploitation. En revanche, le système d'exécution de Java est assez malin pour réaliser cette opération tout seul, ce qui permet de privilégier l'allocation dynamique de la mémoire, qui est bien plus souple que l'allocation statique.

En Java toutes les structures de données sont allouées dynamiquement — généralement par **new**, mais les chaînes sont également allouées dynamiquement. Il n'en reste pas moins que les tableaux manquent un peu de souplesse, puisque leur taille est fixée au moment de la création. Ainsi, un tableau n'est pas très commode pour stocker une suite de points entrés à la souris et terminée par un double clic. Il faut d'abord allouer un tableau de points à une taille par défaut N supposée suffisante et ensuite gérer le cas où l'utilisateur entre plus de N points, par exemple en allouant un nouveau tableau plus grand dans lequel on recopie les points déjà stockés. Il est bien plus commode d'organiser ces points comme une suite de petits blocs de mémoire, un bloc contenant un point et un lien vers le bloc suivant. On alloue alors la mémoire proportionnellement au nombre de points stockés sans se poser de questions. Nous avons redécouvert la liste (simplement chaînée). Une liste $L\langle E\rangle$ de E est :

(1) la liste vide,

(2) ou une paire (une *cellule*) qui contient un élément E et la liste des éléments suivants.

Il s'agit d'une définition par induction, dans le cas 2 le mot « liste » apparaît à nouveau. On peut, sans vraiment tout expliquer, dire que l'ensemble $L\langle E\rangle$ est décrit par l'équation suivante.

$$L\langle E\rangle = \{\,\emptyset\,\} \cup (E \times L\langle E\rangle)$$

Où $\emptyset$ est la liste vide.

On notera que la définition théorique des listes ne fait plus usage du mot lien. Mais dès que l'on implémente les listes, le « lien » revient nécessairement. En Java, une liste est une référence (voir B.3.1.1), qui est :

(1) soit la référence **null** qui ne pointe nulle part et représente la liste vide,

(2) ou bien une référence qui pointe vers une cellule contenant un élément et une liste.

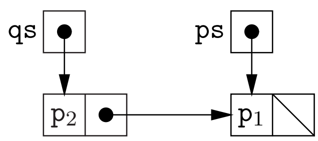
Autrement dit voici une définition possible des cellules de liste de points en Java.

```
class PointList {
  Point val ;          // L'élément
  PointList next ;     // La suite

  PointList (Point val, PointList next) {
    this.val = val ; this.next = next ;
  }
}
```

La cellule de liste est donc un objet de la classe PointList et une valeur de type PointList est bien une référence, puisque les valeurs des objets sont des références. On peut comprendre un aspect de l'organisation des listes à l'aide de dessins qui décrivent une abstraction de l'état de la mémoire, c'est-à-dire une simplification de cet état, sans les détails techniques. Par exemple, on suppose donnés deux points p_1 et p_2, et on produit les listes :

```
  PointList ps = new PointList (p₁, null) ;
  PointList qs = new PointList (p₂, ps) ;
```

Alors, l'état final de la mémoire est schématisé ainsi :



Étant entendu que, dans ce type de schéma, les flèches représentent les références, la barre diagonale **null**, les boîtes nommées les variables, et les boîtes anonymes les cellules mémoire allouées par **new** (voir B.3.1).

Il semble que ce qu'est une liste dans la vie de tous les jours est clair : une liste est une séquence d'éléments. Par exemple la liste des admis à un concours quelconque, ou la liste de courses que l'on emporte au supermarché pour ne rien oublier. L'intuition est juste, elle fait en particulier apparaître (surtout dans le premier exemple) que l'ordre des éléments d'une liste importe. Mais méfions nous un peu, car la liste de l'informatique est très contrainte. En effet, les opérations élémentaires possibles sur une liste ℓ sont peu nombreuses :

(1) Tester si une liste est la liste vide ou pas (ℓ == **null**),

(2) et si ℓ est (une référence vers) une cellule (e, ℓ'), on peut

   (a) extraire le premier élément e (ℓ.val),

(b) ou bien extraire la liste qui suit ℓ (ℓ`.next`).

Pour construite les listes, il n'y a que deux opérations,

(1) la liste vide existe (**null**),

(2) on peut ajouter un nouvel élément e en tête d'une liste ℓ (**new** PointList (e, ℓ)).

Et c'est tout, toutes les autres opérations doivent être programmées à partir des opérations élémentaires.

Une opération fréquemment programmée est le parcours de tous les éléments d'une liste. Cette opération se fait idéalement selon un *idiome*, c'est-à-dire selon une tournure fréquemment employée qui signale l'intention du programmeur, ici une boucle **for**.

```
// ps est une PointList
for (PointList qs = ps ; qs != null ; qs = qs.next ) {
  // Traiter l'élément qs.val
}
```

Les programmeurs Java emploient normalement une telle boucle pour signaler que leur code réalise un simple parcours de liste. Dès lors, la lecture de leur code en est facilitée. Notons que rien, à part le souci de la coutume, ne nous empêche d'employer un autre type de boucle, par exemple

```
PointList qs = ps ;
while (qs != null) {
  // Traiter l'élément qs.val
  qs = qs.next ;
}
```

La liste simplement chaînée est une *structure séquentielle*. Une structure séquentielle regroupe des éléments en séquence et l'accès à un élément donné ne peut se faire qu'en parcourant la structure, jusqu'à trouver l'élément cherché. Cette technique d'*accès séquentiel* s'oppose à l'*accès direct*. Un exemple simple de structure de données qui offre l'accès direct est le tableau. La distinction entre séquentiel et direct se retrouve dans des dispositifs matériels : une bande magnétique n'offre qu'un accès séquentiel puisqu'il faut dérouler toute la bande pour en lire la fin ; en revanche la mémoire de l'ordinateur offre l'accès direct. En effet, on peut voir la mémoire comme un grand tableau (d'octets) dont les indices sont appelés adresses. L'accès au contenu d'une case se fait simplement en donnant son adresse. En fait le tableau est une abstraction de la mémoire de l'ordinateur.

Exercice 1 Donner deux autres exemples de dispositifs matériels offrant accès séquentiel et accès direct.

Solution. L'utilisateur qui entre un texte au clavier n'offre qu'un accès séquentiel à un programme qui lit ce qu'il frappe. Un disque dur peut être considéré comme offrant l'accès direct, le dispositif d'adressage des données est juste un peu plus compliqué que celui de la mémoire. Les données sont réparties en cylindres qui correspondent à des anneaux sur le disque, l'accès à un cylindre donné se fait par un déplacement radial de la tête de lecture. Vu du programmeur il existe donc des fichiers à accès uniquement séquentiel (l'entrée standard si elle correspond au clavier) et des fichiers à accès direct (l'entrée standard si elle correspond à un fichier, par une redirection <*fichier*). □

En résumé la liste est une structure dynamique, parce que la mémoire est allouée petit à petit directement en fonction des besoins. La liste peut être définie inductivement, ce qui rend la programmation récursive assez naturelle (inductif et récursif sont pour nous informaticiens des synonymes). Enfin, par nature, la liste est une structure de données séquentielle, et le parcours de la structure est privilégié par rapport à l'accès à un élément arbitraire.

2 Listes chaînées, révisions

Les listes chaînées ont déjà été vues dans le cours précédent. Nous commençons donc par quelques révisions. Attention, nous profitons de ce que les listes ont « déjà été vues en cours » pour systématiser les techniques de programmation sur les listes. Il y a sans doute un vrai profit à lire cette section. Si vous en doutez, essayez tout de suite les exercices de la section 2.4.

2.1 Opérations élémentaires sur les listes

Comme révision nous montrons comment réaliser les *ensembles* (d'entiers) à partir des listes (d'entiers). Voici la définition des cellules de liste d'entiers.

```
class List {
  int val ;      // L'élément
  List next ;    // La suite

  List (int  val, List next) {
    this.val = val ; this.next = next ;
  }
}
```

Un ensemble est représenté par une liste dont tous les éléments sont deux à deux distincts. Nous ne spécifions aucune autre contrainte, en particulier aucun ordre des éléments d'une liste n'est imposé. Pour la programmation, un premier point très important à prendre en compte est que **null** est un ensemble valable. Il est alors naturel d'écrire des méthodes statiques, car **null** qui ne possède aucune méthode est une valeur légitime pour une variable de type List .

Commençons par calculer le cardinal d'en ensemble. Compte tenu de ce que les éléments des listes sont deux à deux distincts, il suffit de calculer la longueur d'une liste. Employons donc la boucle idiomatique

```
static int card(List xs) {
  int r = 0 ;
  for ( ; xs != null ; xs = xs.next)
    r++ ; // pour r = r+1 ;
  return r ;
}
```

Ce premier code fait apparaître qu'il n'est pas toujours utile de réserver une variable locale pour la boucle idiomatique. Écrire `xs = xs.next` ne fait que changer la valeur de la variable locale `xs` (qui appartient en propre à l'appel de méthode) et n'a pas d'impact sur la liste elle-même. Ainsi écrire le code suivant ne pose pas de problème particulier.

```
  List xs = new List (1, new List (2, new List (3, new List (4,null)))) 1
  int size = card(xs) ;
```

À la fin du code, la variable `xs` existe toujours et référence toujours la première cellule de la liste `{1, 2, 3, 4}`. En fait, cette variable `xs` n'a rien à voir avec le paramètre homonyme de la méthode `card` et heureusement.

Poursuivons en affichant les ensembles, ou, plus exactement, en fabriquant une représentation affichable des ensembles sous forme de chaîne. Voici un premier essai d'écriture d'une méthode statique `toString` dans la classe List avec la boucle idiomatique.

```
static String toString(List xs) {
  String r = "{" ;
  for ( ; xs != null ; xs = xs.next )
    r = r + xs.val + ", " ;
  r = r + "}" ;
```

```
  return r ;
}
```

Le code est critiquable :

- L'affichage est laid, il y a une virgule en trop à la fin. Par exemple, pour la liste des entiers 1 et 3, on obtient *"{1, 3, }"*.
- Le coût de production de la chaîne est quadratique en la longueur de la liste. En effet la concaténation de deux chaînes (par `+`) coûte en proportion de la somme des longueurs des chaînes concaténées, car il faut allouer une nouvelle chaîne pour y recopier les caractères des chaînes concaténées. Pour une liste `xs` de longueur n on procède donc à n concaténations de chaînes, de tailles supposées régulièrement croissantes ce qui mène au final à un coût quadratique.

$$k + 2 \times k + \cdots n \times k = \frac{n(n+1)}{2}k$$

Pour remédier au premier problème on peut distinguer le cas du premier élément de la liste. Pour remédier au second problème il faut utiliser un objet **StringBuilder** de la bibliothèque Java (voir B.6.1.4). Les objets **StringBuilder** sont des chaînes dynamiques, dont on peut changer la taille. En particulier, ils possèdent une méthode `append(String str)` qui permet de leur ajouter une chaîne `str` à la fin, pour un coût que l'on peut considérer proportionnel à la longueur de `str`. Bref on a :

```
static String toString(List xs) {
  StringBuilder r = new StringBuilder () ; // Un StringBuilder à nous

  r.append("{") ;
  if (xs != null) {
    r.append(xs.val) ; // ajouter l'écriture décimale du premier entier
    xs = xs.next ;
    // et celles des suivants préfixées par ", "
    for ( ; xs != null ; xs = xs.next )
      r.append(", " + xs.val) ;
  }
  r.append("}") ;

 // Renvoyer la chaîne contenue dans le StringBuilder
  return r.toString() ;
}
```

Après cet échauffement, écrivons ensuite la méthode `mem` qui teste l'appartenance d'un entier à un ensemble.

```
static boolean mem(int x, List xs) {
  for ( ; xs != null ; xs = xs.next) {
    if (x == xs.val) return true ;
  }
  return false ;
}
```

On emploie la boucle idiomatique, afin de parcourir la liste et de retourner de la méthode `mem` dès qu'un entier égal à `x` est trouvé. Comme on le sait certainement déjà une écriture récursive est également possible.

```
static boolean mem(int x, List xs) {
  if (xs == null) {
    return false ;
  } else {
```

```
    return (x == xs.val) || mem(x, xs.next) ;
  }
}
```

La version récursive provient directement de la définition inductive des listes, elle tient en deux équations :

$$\begin{array}{rcl} M(x, \emptyset) & = & \mathit{faux} \\ M(x, (x', X')) & = & (x = x') \vee M(x, X') \end{array}$$

Ces deux équations sont le support d'une preuve par induction structurelle évidente de la correction de `mem`.

Alors, que choisir ? De façon générale le code itératif (le premier, avec une boucle) est plus efficace que le code récursif (le second), ne serait-ce que parce que les appels de méthode sont assez coûteux. Mais le code récursif résulte d'une analyse systématique de la structure de données inductive, et il a bien plus de chances d'être juste du premier coup. Ici, dans un cas aussi simple, le code itératif est préférable (en Java qui favorise ce style).

2.2 Programmation sûre, style dit fonctionnel

Nous envisageons maintenant des méthodes qui fabriquent de nouveaux ensembles. Commençons par la méthode `add` qui ajoute un élément à un ensemble. Le point remarquable est qu'un ensemble contient au plus une fois un élément donné. Une fois disponible la méthode `mem`, écrire `add` est très facile.

```
static List add(int x, List xs) {
  if (mem(x, xs)) {
    return xs ;
  } else {
    return new List (x, xs) ;
  }
}
```

Attaquons ensuite la méthode `remove` qui enlève un élément x d'un ensemble X. Nous choisissons de suivre le principe des structures de données dites *fonctionnelles* ou *non-mutables* ou encore *persistantes*. Selon ce principe, on ne modifie jamais le contenu des cellules de liste. Cela revient à considérer les listes comme définies inductivement par $L\langle E\rangle = \{\emptyset\} \cup (E \times L\langle E\rangle)$, et il est important d'admettre dès maintenant que cette approche rend la programmation bien plus sûre.

Pour écrire `remove`, raisonnons inductivement : dans un ensemble vide, il n'y a rien à enlever ; tandis que dans un ensemble (une liste) $X = (x', X')$ on distingue deux cas :

- L'élément x à enlever est égal à x' et alors X moins x est X', car X est une liste d'éléments deux à deux distincts et donc X' ne contient pas x.
- Sinon, il faut enlever x de X' et ajouter x' à la liste obtenue, dont on sait qu'elle ne peut pas contenir x'.

Soit en équations :

$$\begin{array}{rcll} R(x, \emptyset) & = & \emptyset & \\ R(x, (x, X')) & = & X' & \\ R(x, (x', X')) & = & (x', R(x, X')) & \text{avec } x \neq x' \end{array}$$

Et en Java :

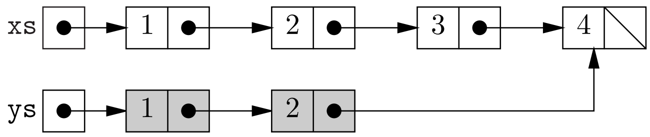
```
static List remove(int x, List xs) {
  if (xs == null) {
    return null ;
  } else if (x == xs.val) {
```

```
    return xs.next ;
  } else {
    return new List (xs.val, remove(x, xs.next)) ;
  }
}
```

En ce qui concerne l'état mémoire, `remove(x,xs)` renvoie une nouvelle liste (il y a des **new**), dont le début est une copie du début de `xs`, jusqu'à trouver l'élément `x`. Plus précisément, si nous écrivons

```
List xs = new List(1, new List(2, new List (3, new List (4, null)))) ;
List ys = remove(3, xs) ;
```

Alors on a les structures suivantes en mémoire

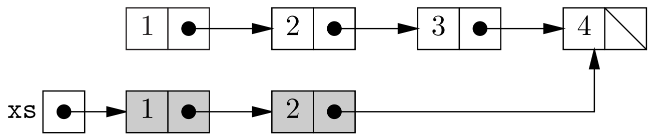


(Les cellules allouées par `remove` sont grisées.) On constate que la partie de la liste `xs` qui précède l'élément supprimé est recopiée, tandis que la partie qui suit l'élément supprimé est partagée entre les deux listes. Dans le style persistant on en vient toujours à copier les parties des structures qui sont changées.

Au passage, si on écrit plutôt

```
List xs = new List(1, new List(2, new List (3, new List (4, null)))) ;
xs = remove(3, xs) ;
```

Alors on obtient



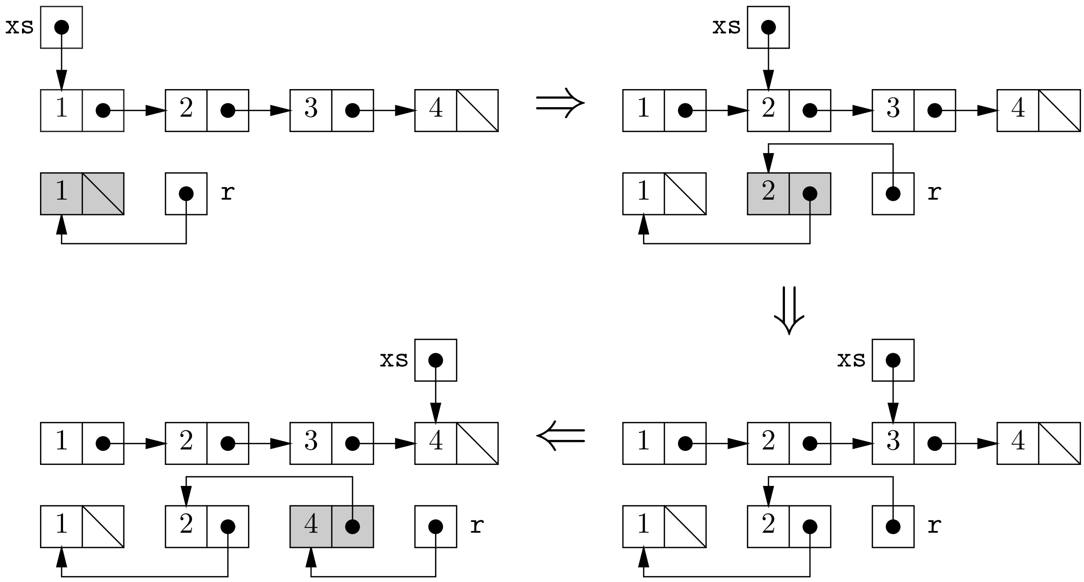
C'est-à-dire exactement la même chose, sauf que la variable `xs` pointe maintenant vers la nouvelle liste et qu'il n'y a plus de référence vers la première cellule de l'ancienne liste. Il en résulte que la mémoire occupée par les trois premières cellules de l'ancienne liste ne peut plus être accédée par le programme. Ces cellules sont devenues les *miettes* et le gestionnaire de mémoire de l'environnement d'exécution de Java peut les récupérer. Le ramassage des miettes (*garbage collection*) est automatique en Java, et c'est un des points forts du langage.

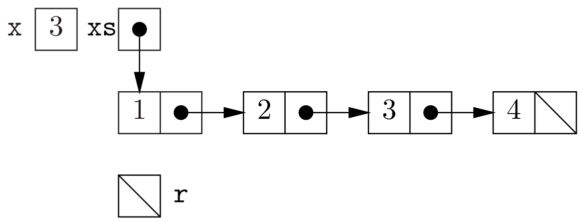
Écrivons maintenant `remove` avec une boucle. Voici un premier essai : parcourir la liste `xs` en la recopiant en évitant de copier tout élément égal à `x`.

```
static List remove(int x, List xs) {
  List r = null ;
  for ( ; xs != null ; xs = xs.next ) {
     if (x != xs.val)
       r = new List(xs.val, r) ;
  }
  return r ;
}
```

Et ici apparaît une différence, alors que la version récursive de `remove` préserve l'ordre des éléments de la liste, la version itérative inverse cet ordre. Pour s'en convaincre, on peut remarquer que lorsque qu'une nouvelle cellule `new List(xs.val, r)` est construite, l'élément `xs.val` est ajouté au début de la liste `r`, tandis qu'il est extrait de la fin de la liste des éléments déjà parcourus. On peut aussi schématiser l'état mémoire sur l'exemple déjà utilisé. Avant la boucle on a

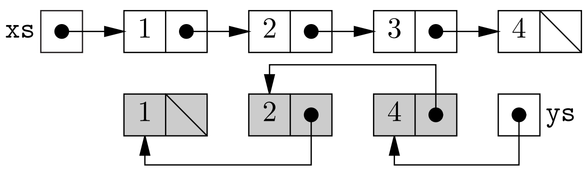
FIG. 1 – Les quatre itérations de `remove`





La figure 1 schématise l'état de la mémoire à la fin de chacune des quatre itérations, la cellule nouvelle étant grisée. Et enfin voici le bilan de l'état mémoire, pour le programme complet.

```
List xs = new List(1, new List(2, new List (3, new List (4, null)))) ;
List ys = remove(3, xs) ;
```



L'inversion n'a pas d'importance ici, puisque les ensembles sont des listes d'éléments deux à deux distincts sans autre contrainte, le code itératif convient. Il n'en y irait pas de même si les listes étaient ordonnées.

Il est possible de procéder itérativement et de conserver l'ordre des éléments. Mais c'est un peu plus difficile. On emploie une technique de programmation que faute de mieux nous appellerons *initialisation différée*. Jusqu'ici nous avons toujours appelé le constructeur des listes à deux arguments, ce qui nous garantit une bonne initialisation des champs `val` et `next` des cellules de liste. Nous nous donnons un nouveau constructeur `List (int val)` qui initialise seulement le champ `val`.[1] Le champ `next` sera affecté une seule fois plus tard, ce qui revient moralement à une initialisation.

[1]En fait, un champ de type objet non-initialisé explicitement contient la valeur par défaut **null**, voir B.3.3.

```
  List (int val) { this.val = val ; }
```

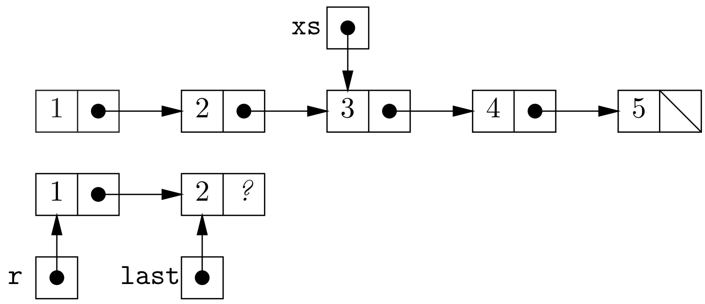
Pour illustrer l'initialisation différée, nous commençons par un exemple plus facile.

Exercice 2 Copier une liste en respectant l'ordre de ses éléments, sans écrire de méthode récursive.

Solution. L'idée revient à parcourir la liste `xs` en copiant ses cellules et en délégant l'initialisation du champ `next` à l'itération suivante. Il y a un petit décalage à gérer au début et à la fin du parcours.

```
static List copy(List xs) {
  if (xs == null) return null ;
  // Ici la copie possède au moins une cellule, que voici
  List r = new List (xs.val) ;
  // last pointe sur la dernière cellule de la liste r
  List last = r ;
  // Parcourir la suite de xs
  for ( xs = xs.next ; xs != null ; xs = xs.next ) {
    // « initialiser » last.next à une nouvelle cellule
    last.next = new List (xs.val) ;
    /** qui devient donc la dernière cellule du résultat **/
    last = last.next ;
  }
 // « initialiser » le next de la toute dernière cellule à une liste vide
  last.next = null ; // Inutile, mais c'est plus propre
  return r ;
}
```

On observe que le code construit une liste sur laquelle il maintient deux références, `r` pointe sur la première cellule, tandis que `last` pointe sur la dernière cellule — sauf très brièvement au niveau du commentaire `/**...**/`, où `last` pointe sur l'avant-dernière cellule. Plus précisément, la situation en « régime permanent » avant la ligne qui précède le commentaire `/**...**/` est la suivante :



Et après ajout d'une nouvelle cellule par `last.next = new List(xs.val)`.

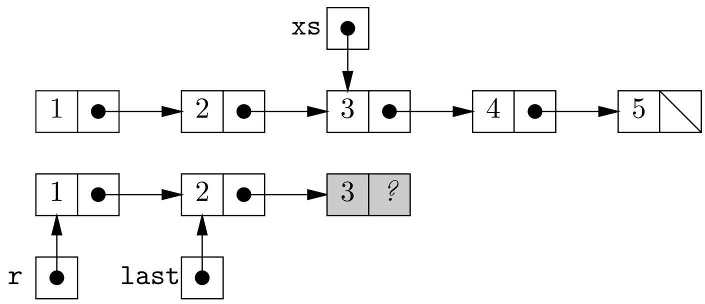
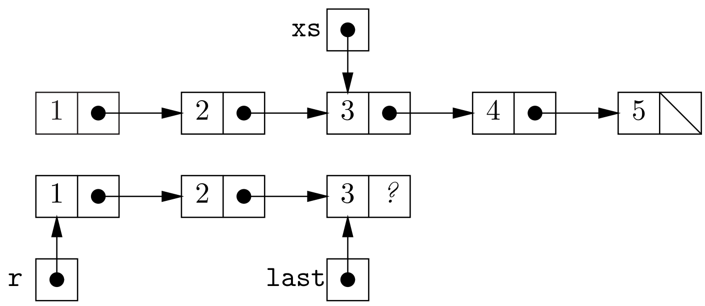


FIG. 2 – Enlever un élément selon la technique de l'initialisation différée

```
static List remove(int x, List xs) {
  /* Deux cas particuliers */
  if (xs == null) return null ;
  /* Ici xs != null, donc xs.val existe */
  if (x == xs.val) return xs.next ;
  /* Ici la première cellule du résultat existe et contient xs.val */
  List r = new List (xs.val) ;
  List last = r ;
  /* Cas général : copier xs (dans r), jusqu'à trouver x */
  for ( xs = xs.next ; xs != null ; xs = xs.next ) {
    if (x == xs.val) {
      // « initialiser » last.next à xs moins xs.val (c-a-d. xs.next)
      last.next = xs.next ;
      return r ;
    }
    // « initialiser » last.next à une nouvelle cellule
    last.next = new List (xs.val) ;
    last = last.next ;
  }
  // « initialiser » last.next à une liste vide
  last.next = null ; // Inutile mais c'est plus propre.
  return r ;
}
```

(La cellule qui vient d'être allouée est grisée). Et enfin après exécution de l'affectation `last == last.next`.



On conçoit qu'avec un tel dispositif il est facile d'ajouter de nouvelles cellules de liste à la fin du résultat et possible de rendre le résultat (avec `r` qui pointe sur la première cellule). On observe le cas particulier traité au début, il résulte de la nécessité d'identifier le cas où il ne faut pas construire la première cellule du résultat.

Enfin, la méthode itérative construit exactement la même liste que la méthode récursive, même si les cellules de listes sont allouées en ordre inverse. En effet la méthode récursive alloue la cellule après que la valeur du champ `next` est connue, tandis que la méthode itérative l'alloue avant. □

La figure 2 donne la nouvelle version itérative de `remove`, écrite selon le principe de l'initialisation différée. Par rapport à la copie de liste, le principal changement est la possibilité d'arrêt de la copie à l'intérieur de la boucle. En effet, lorsque l'élément à supprimer est identifié (`x == xs.val`), nous connaissons la valeur à ranger dans `last.next`, puisque par l'hypothèse « `xs` est un ensemble », `xs` moins l'élément `x` est exactement `xs.next`, on peut donc arrêter la copie. On observe également les deux cas particuliers traités au début, ils résultent, comme dans la copie de liste, de la nécessité d'identifier les cas où il ne faut pas construire la première cellule du résultat. Au final le code itératif est quand même plus complexe que le code récursif. Par conséquent, si l'efficacité importe peu, par exemple si les listes sont de petite taille, la programmation récursive est préférable.

Nous pouvons maintenant étendre les opérations élémentaires impliquant un élément et un ensemble et programmer les opérations ensemblistes du test d'inclusion, de l'union et de la différence. Pour programmer le test d'inclusion `xs` $\subseteq$ `ys`, il suffit de tester l'appartenance de tous les éléments de `xs` à `ys`.

```
static boolean included(List xs, List ys) {
  for ( ; xs != null ; xs = xs.next )
    if (!mem(xs.val, ys)) return false ;
  return true ;
}
```

Exercice 3 Programmer l'union et la différence ensembliste.

Solution. Pour l'union, il suffit d'ajouter tous les élément d'un ensemble à l'autre.

```
static List union(List xs, List ys) {
  List r = ys ;
  for ( ; xs != null ; xs = xs.next )
    r = add(xs.val, r) ;
  return r ;
}
```

On aurait même pu se passer de la variable `r` pour écrire `ys = add(xs.val, ys)` et puis `return ys`. Mais la clarté en aurait souffert.

Pour la différence des ensembles, légèrement plus délicate en raison de la non-commutativité, on enlève de `xs` tous les éléments de `ys`.

```
// Renvoie xs \ ys
static List diff(List xs, List ys) {
  List r = xs ;
  for ( ; ys != null ; ys = ys.next )
    r = remove(ys.val, r) ;
  return r ;
}
```

□

2.3 Programmation dangereuse, structures mutables

Les méthodes `remove` de la section précédente copient tout ou partie des cellules de la liste passée en argument. On peut, par exemple dans un souci (souvent mal venu, disons le tout de suite) d'économie de la mémoire, chercher à éviter ces copies. C'est parfaitement possible, à condition de s'autoriser les affectations des champs des cellules de liste. On parle alors de structure de données *mutable* ou *impérative*. Ainsi, dans le `remove` récursif on remplace l'allocation d'une nouvelle cellule par la modification de l'ancienne cellule, et on obtient la méthode `nremove` *destructive* ou *en place* suivante

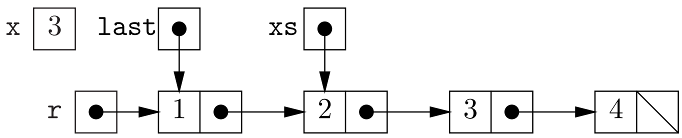
```
static List nremove(int x, List xs) {
  if (xs == null) {
    return null ;
```

```
  } else if (x == xs.val) {
    return xs.next ;
  } else {
    xs.next = nremove(x, xs.next) ;
    return xs ;
  }
}
```

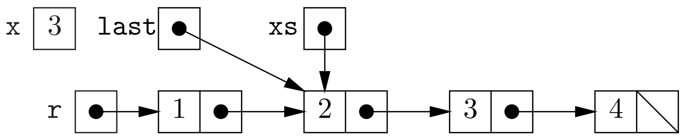
On peut adapter la seconde version itérative de `remove` (figure 2) selon le même esprit, les références `r` et `last` pointant maintenant, non plus sur une copie de la liste passée en argument, mais sur cette liste elle-même.

```
static List nremove(int x, List xs) {
  /* Deux cas particuliers */
  if (xs == null) return null ;
  if (x == xs.val) return xs.next ;

  // Ici la première cellule du résultat existe et contient xs.val,
  List r = xs ;
  List last = r ;
  /* Cas général : chercher x dans xs pour enlever sa cellule */
  for ( xs = xs.next ; xs != null ; xs = xs.next ) {
    /** Ici, on a last.next == xs **/
    if (x == xs.val) {
      // Enlever la cellule pointée par xs
      last.next = xs.next ;
      return r ;
    }
    last.next = xs ; // Affectation inutile
    last = last.next ;
    /** Exceptionnellement, on a xs == last **/
  }
  last.next = null ; // Affectation inutile
  return r ;
}
```

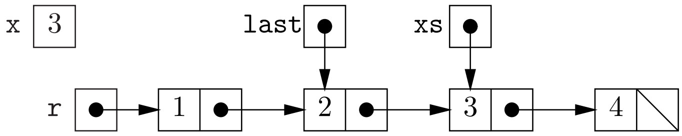
On note que la version itérative comporte des affectations de `last.next` inutiles. En effet, à l'intérieur de la boucle les références `xs` et `last.next` sont toujours égales, car `last` pointe toujours sur la cellule qui précède `xs` dans la liste passée en argument, sauf très brièvement au niveau du second commentaire `/**...**/`. En régime permanent, la situation au début d'une itération de boucle (avant le premier commentaire `/**...**/`) est la suivante



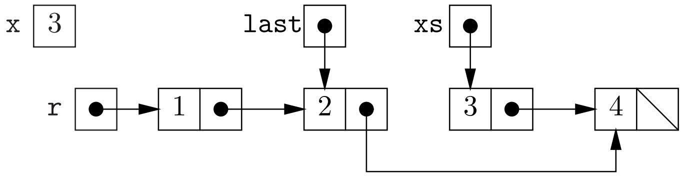
Et en fin d'itération, après le second commentaire `/**...**/`, on a



Au début de l'itération suivante on a



La recherche de `xs.val == x` est terminée, et le champ `next` de la cellule `last` qui précède la cellule `xs` est affecté.

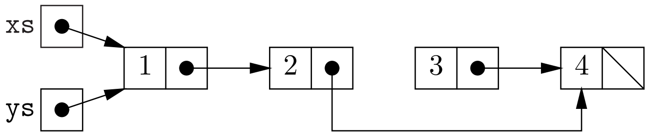


Puis la liste `r` est renvoyée.

Au final, l'effet de `nremove(int x, List xs)` s'apparente à un court-circuit de la cellule qui contient `x`. Ainsi exécuter

```
List xs = new List(1, new List(2, new List (3, new List (4, null)))) ;
List ys = nremove(3, xs) ;
```

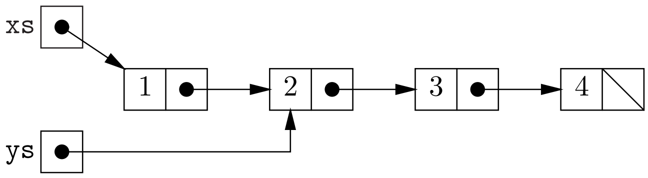
produit l'état mémoire



Un problème majeur survient donc : l'ensemble `xs` est modifié ! On comprend donc pourquoi `nremove` est dite destructive. On peut se dire que cela n'est pas si grave et que l'on saura en tenir compte. Mais c'est assez vain, car si on supprime le premier élément de l'ensemble `xs`, le comportement est différent

```
List xs = new List(1, new List(2, new List (3, new List (4, null)))) ;
List ys = nremove(1, xs) ;
```

Alors on obtient



Et on observe qu'alors l'ensemble `xs` n'a pas changé.

En conclusion, la programmation des structures mutables est tellement délicate qu'il est souvent plus raisonnable de s'abstenir de l'employer. Comme toujours en programmation, la maxime ci-dessus souffre des exceptions, dans les quelques cas où l'on possède une maîtrise complète des structures de données, et en particulier, quand on est absolument sûr que les structures modifiées ne sont connues d'aucune autre partie du programme.

Un bon exemple de ce type d'exception est la technique d'initialisation différée employée dans la seconde méthode `remove` itérative de la figure 2. En effet, le champ `next` est affecté une seule fois, et seulement sur des cellules fraîches, allouées par la méthode `remove` elle-même, et qui ne sont accessible qu'à partir des variables `r` et `last` elles-mêmes locales à `remove`. Par conséquent, les modifications apportées à certaines cellules sont invisibles de l'extérieur de `remove`.

2.4 Contrôle de la révision

Nous avons divisé les techniques de programmation sur les listes selon deux fois deux catégories. Les listes peuvent être fonctionnelles (persistantes) ou mutables (impératives), et la programmation peut être récursive ou itérative.

Afin d'être bien sûr d'avoir tout compris, programmons une opération classique sur les listes des quatre façons possibles. Concaténer deux listes signifie, comme pour les chaînes, ajouter les éléments d'une liste à la fin de l'autre. Nommons `append` la méthode qui concatène deux listes. La programmation récursive non-mutable résulte d'une analyse simple de la structure inductive de liste.

```
static List append(List xs, List ys) {
  if (xs == null) {
    return ys ; // A(∅,Y) = Y
  } else {
    return new List (xs.val, append(xs.next, ys)) ; // A((x,X),Y) = (x,A(X,Y))
  }
}
```

Nommons `nappend` la version mutable de la concaténation. L'écriture récursive de `nappend` à partir de la version fonctionnelle est mécanique. Il suffit de remplacer les allocations de cellules par des modifications.

```
static List nappend(List xs, List ys) {
  if (xs == null) {
    return ys ;
  } else {
    xs.next = nappend(xs.next, ys)) ;
    return xs ;
  }
}
```

Pour la programmation itérative non-mutable, on a recours à l'initialisation différée.

```
static List append(List xs, List ys) {
  if (xs == null) return ys ;
  List r = new List (x.val) ;
  List last = r ;
  for ( xs = xs.next ; xs != null ; xs = xs.next ) {
    last.next = new List(xs.val) ;
    last = last.next ;
  }
  last.next = ys ;
  return r ;
}
```

Pour la programmation mutable et itérative, on supprime les allocations de la version non-mutable et on remplace les initialisations différées par des affectations des champs `next` de la liste passée comme premier argument. On n'explicite que la dernière affectation de `last.next` qui est la seule qui change quelque chose.

```
static List nappend(List xs, List ys) {
  if (xs == null) return ys ;
  List r = xs ;
  List last = r ;
  for ( xs = xs.next ; xs != null ; xs = xs.next ) {
    last = last.next ;
  }
```

```
  last.next = ys ;
  return r ;
}
```

Voici un autre codage plus concis de la même méthode, qui met en valeur la recherche de la dernière cellule de la liste `xs`.

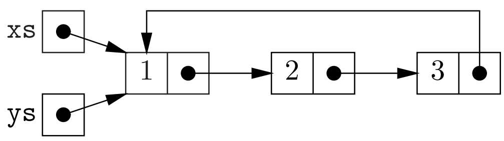
```
static List nappend(List xs, List ys) {
  if (xs == null) return ys ;
  List last = xs ;
  for (  ; last.next != null ; last = last.next )
    ; // idiome : boucle qui ne fait rien
  last.next = ys ;
  return xs ;
}
```

Enfin on constate que le coût de la concaténation est de l'ordre de n opérations élémentaires, où n est la longueur de la première liste.

Exercice 4 Que fait le programme suivant ?

```
  List xs = new List (1, new List (2, new List (3, null))) ;
  List ys = nappend(xs, xs) ;
```

Solution. Le programme affecte la référence `xs` au champ `next` de la dernière cellule de la liste `xs`. Il en résulte une liste « bouclée ».



□

Et pour être vraiment sûr d'avoir compris, nous conseillons encore un exercice, d'ailleurs un peu piégé.

Exercice 5 Programmer les deux versions fonctionnelles, itérative et récursive, de `reverse` la méthode qui inverse une liste.

Solution. Une fois n'est pas coutume, la version itérative est de loin la plus naturelle.

```
static List reverse(List xs) {
  List r = null ;
  for ( ; xs != null ; xs = xs.next )
    r = new List (xs.val, r) ;
  return r ;
}
```

En effet, en parcourant (lisant) une séquence quelconque d'entiers (par ex. en lisant des entiers entrés au clavier) pour construire une liste, la liste est naturellement construite dans l'ordre inverse de la lecture. De fait, les éléments sont ajoutés au début de la liste construite alors qu'il sont extraits de la fin de la séquence lue — voir aussi le premier `remove` itératif de la section 2.2. C'est souvent embêtant, mais ici c'est utile.

Écrire la version récursive est un peu gratuit en Java. Mais la démarche ne manque pas d'intérêt. Soit R la fonction qui inverse une liste et A la concaténation des listes, le premier élément de la liste que l'on souhaite inverser doit aller à la fin de la liste inversée. Soit l'équation :

$$R((x, X)) = A(R(X), (x, \emptyset))$$

On est donc tenté d'écrire :

```
// Mauvais code, montre ce qu'il faut éviter.
static List reverse(List xs) {
  if (xs == null) {
    return null ;
  } else {
    return append(reverse(xs.next), new List (xs.val, null)) ;
  }
}
```

Ce n'est pas une très bonne idée, car pour inverser une liste de longueur n, il en coûtera au moins de l'ordre de n^2 opérations élémentaires, correspondant aux n `append` effectués sur des listes de taille $n-1, \ldots 0$.

Un truc des plus classiques vient alors à notre secours. Le truc revient à considérer une méthode auxiliaire récursive qui prend deux arguments. Nous pourrions donner le truc directement, mais préférons faire semblant de le déduire de démarches plus générales, l'une théorique et l'autre pratique.

Commençons par la théorie. Généralisons l'équation précédente, pour toute liste Y on a

$$A(R((x, X)), Y) = A(A(R(X), (x, \emptyset)), Y)$$

Or, quelles que soient les listes X, Y et Z, on a $A(A(X, Y), Z) = A(X, A(Y, Z))$ (la concaténation est associative). Il vient :

$$A(R((x, X)), Y) = A(R(X), A((x, \emptyset), Y))$$

Autrement dit :

$$A(R((x, X)), Y) = A(R(X), (x, Y))$$

Posons $R'(X, Y) = A(R(X), Y)$, autrement dit R' renvoie la concaténation de X inversée et de Y. Avec cette nouvelle notation, l'équation précédente s'écrit :

$$R'((x, X)), Y) = R'(X, (x, Y))$$

Équation qui permet de calculer R' récursivement, puisque de toute évidence $R'(\emptyset, Y) = Y$.

```
private static reverseAux(List xs, List ys) {
  if (xs == null) {
    return ys ;
  } else {
    return reverseAux(xs.next, new List (xs.val, ys)) ;
  }
}
```

Par ailleurs on a $R'(X, \emptyset) = A(R(X), \emptyset) = R(X)$ et donc :

```
static List reverse(List xs) {
  return reverseAux(xs, null) ;
}
```

Pour l'approche pratique du truc, il faut savoir que l'on peut toujours et de façon assez simple remplacer une boucle par une récursion. Commençons par négliger les détails du corps de la boucle de `reverse` itératif. Le corps de la boucle lit les variables `xs` et `r` puis modifie `r`. On réécrit donc la boucle ainsi

```
for ( ; xs != null ; xs = xs.next )
  r = body(xs, r) ;
```

Où la méthode `body` est évidente.

Considérons maintenant une itération de la boucle comme une méthode `loop`. Cette méthode a besoin des paramètres `xs` et `r`. Si `xs` est **null**, alors il n'y a pas d'itération (pas d'appel de `body`) et il faut passer le résultat `r` courant à la suite du code. Sinon, il faut appeler l'itération suivante en lui passant `r` modifié par l'appel à `body`. Dans ce cas, le résultat à passer à la suite du code est celui de l'itération suivante. On opère ensuite un petit saut sémantique en comprenant « passer à la suite du code » comme « retourner en tant que résultat de la méthode `loop` ». Soit :

```
private static List loop(List xs, List r) {
  if (xs == null) {
    return r ;
  } else {
    return loop(xs.next, body(xs, r)) ;
  }
}
```

La méthode `loop` est dite *récursive terminale* car la dernière opération effectuée avant de retourner est un appel récursif. Ou plus exactement l'unique appel récursif est en position dite terminale.

Enfin, dans le corps de `reverse` itératif, on remplace la boucle par un appel à `loop`.

```
static List reverse(List xs) {
  List r = null ;
  r = loop(xs, r) ;
  return r ;
}
```

On retrouve, après nettoyage, le même code qu'en suivant la démarche théorique. De tous ces discours, il faut surtout retenir l'équivalence entre boucle et récursion terminale. □

3 Tri des listes

Nous disposons désormais de techniques de programmation élémentaires sur les listes. Il est temps d'aborder des programmes un peu plus audacieux, et notamment de considérer des algorithmes. Le tri est une opération fréquemment effectuée. En particulier, les tris interviennent souvent dans d'autres algorithmes, dans le cas fréquent où il plus simple ou plus efficace d'effectuer des traitements sur des données prises dans l'ordre plutôt que dans le désordre.

Exercice 6 Programmer la méthode `static List uniq(List xs)` qui élimine les doublons d'une liste.

Solution. Dans une liste triée les éléments égaux se suivent. Pour enlever les doublons d'une liste triée, il suffit donc de parcourir la liste en la copiant et d'éviter de copier les éléments égaux à leur successeur dans la liste.

```
private static List uniqSorted(List xs) {
  if (xs == null || xs.next == null) { // xs a zéro ou un élément
    return xs ;
  } // desormais xs != null && xs.next != null
    else if (xs.val == xs.next.val) ;
    return uniqSorted(xs.next) ;
  } else {
    return new List (xs.val, uniqSorted(xs.next)) ;
  }
}
```

La sémantique séquentielle du ou logique `||` permet la concision (voir B.7.3). En supposant donnée une méthode `sort` de tri des listes, on écrit la méthode `uniq` qui accepte une liste quelconque.

```
static List uniq(List xs) { return uniqSorted(sort(xs)) ; }
```

On peut discuter pour savoir si `uniq` ci-dessus est plus simple que la version sans tri ci-dessous ou pas.

```
static List uniq(List xs) {
  List r = null ;
  for ( ; xs != null ; xs = xs.next )
    if (!mem(x.val, xs))
      r = new List (xs.val, r) ;
  return r ;
}
```

En revanche, comme on peut trier une liste au prix de de l'ordre de $n \log n$ opérations élémentaires (n longueur de `xs`), et que le second `uniq` effectue au pire de l'ordre de n^2 opérations élémentaires, il est certain que le premier `uniq` est asymptotiquement plus efficace. □

Les algorithmes de tri sont nombreux et ils ont fait l'objet d'analyses très fouillées, en raison de leur intérêt et de leur relative simplicité. Sur ce point [6] est une référence fondamentale.

3.1 Tri par insertion

Ce tri est un des tris de listes les plus naturels. L'algorithme est simple : pour trier la liste `xs`, on la parcourt en insérant ses éléments à leur place dans une liste résultat qui est donc triée. C'est la technique généralement employée pour trier une main dans les jeux de cartes.

```
static List insertionSort(List xs) {
  List r = null ;
  for ( ; xs != null ; xs = xs.next )
    r = insert(xs.val, r) ;
  return r ;
}
```

Il reste à écrire l'insertion dans une liste triée. L'analyse inductive conduit à la question suivante : soit x un entier et Y une liste triée, quand la liste (x, Y) est elle triée ? De toute évidence cela est vrai, si

- Y est la liste vide $\emptyset$,
- ou bien, $Y = (y, Y')$ et $x \le y$.

En effet, si $x \le y$, alors tous les éléments de Y triée sont plus grands (au sens large) que x, par transitivité de $\le$. Notons $P(X)$ le prédicat défini par

$$P(x, \emptyset) = \mathit{vrai} \qquad\qquad P(x, (y, Y)) = x \le y$$

Soit I fonction d'insertion, selon l'analyse précédente on pose

$$I(x, Y) = (x, Y) \quad \text{si } P(x, Y)$$

Si $P(x, Y)$ est invalide, alors Y s'écrit nécessairement $Y = (y, Y')$ et on est tenté de poser

$$I(x, (y, Y') = (y, I(x, Y'))$$

Et on aura raison. On montre d'abord (par induction structurelle) le lemme évident que $I(x, Y)$ regroupe les éléments de Y plus x. Ensuite on montre par induction structurelle que $I(x, Y)$ est triée.

Base Si $P(x, Y)$ est vrai (ce qui comprend le cas $Y = \emptyset$), alors $I(x, Y)$ est triée.

Induction Sinon, on a $Y = (y, Y')$. Par hypothèse d'induction $I(x, Y')$ est triée. Par ailleurs, les éléments de $I(x, Y')$ sont ceux de Y' plus x, et donc y est plus petit (au sens large) que tous les éléments de $I(x, Y')$ (par hypothèse (y, Y') triée et $P(x, Y)$ invalide). Soit finalement $(y, I(x, Y'))$ est triée.

Il reste à programmer d'abord le prédicat P,

```
private static boolean here(int x, List ys) {
  return ys == null || x <= ys.val ;
}
```

puis la méthode d'insertion `insert`.

```
private static List insert(int x, List ys) {
  if (here(x, ys)) {
    return new List (x, ys) ;
  } else { // NB: !here(ys) => ys != null
    return new List (ys.val, insert(x, ys.next)) ;
  }
}
```

Penchons nous maintenant sur le coût de notre implémentation du tri par insertion. Par coût nous entendons d'abord temps d'exécution. Les constructions élémentaires de Java employées[2] s'exécutent en *temps constant* c'est à dire que leur coût est borné par une constante. Il en résulte que même si le coût d'un appel à la méthode `here` est variable, selon que le paramètre `ys` vaut **null** ou pas, ce coût reste inférieur à une constante correspondant à la somme des coûts d'un appel de méthode à deux paramètres, d'un test contre **null**, d'un accès au champs `val` etc.

Il n'en va pas de même de la méthode `insert` qui est récursive. De fait, le coût d'un appel à `insert` s'exprime comme une suite $I(n)$ d'une grandeur qui exprime la taille de l'entrée, ici la longueur de la liste `ys`.

$$I(0) \leq k_0 \qquad\qquad I(n+1) \leq I(n) + k_1$$

Notons que pour borner $I(n+1)$ nous avons supposé qu'un appel récursif était effectué. Il est donc immédiat que $I(n)$ est majorée par $k_1 \cdot n + k_0$. En première approximation, la valeur des constantes k_1 et k_0 importe peu et on en déduit l'information pertinente que $I(n)$ est en $O(n)$.

En regroupant le coût des diverses instructions de coût constant de la méthode `insertionSort` selon qu'elles sont exécutées une ou n fois, et en tenant compte des n appels à `insert`, le coût $S(n)$ d'un appel de cette méthode est borné ainsi :

$$S(n) \leq \sum_{k=0}^{n-1} I(k) + k_2 \cdot n + k_3 \leq k_1 \cdot \frac{n(n-1)}{2} + (k_0 + k_2) \cdot n + k_3$$

[2]Attention, ce n'est pas toujours le cas, une allocation de tableau par exemple prend un temps proportionnel à la taille du tableau. Par ailleurs nous négligeons le coût du *garbage collector*.

Et donc le coût de `insertionSort` est en $O(n^2)$. Il s'agit d'un coût *dans le cas le pire* en raison des majorations effectuées (ici sur $I(n)$ principalement). Notons que la notation $f(n)$ est en $O(g(n))$ (il existe une constante k telle que, pour n assez grand, on a $f(n) \leq k \cdot g(n)$) est particulièrement justifiée pour les coûts dans le cas le pire. Il faut noter que l'on peut généralement limiter les calculs aux termes dominants en n. Ici on dira, `insertionSort` appelle n fois `insert` qui est en $O(k)$ pour $k \leq n$, et que `insertionSort` est donc en $O(n^2)$.

On peut aussi compter précisément une opération en particulier. Pour un tri on a tendance à compter les comparaisons entre éléments. La démarche se justifie de deux façons : l'opération comptée est la plus coûteuse (ici que l'on songe au tri des lignes d'un fichier par exemple), ou bien on peut affecter un compte borné d'opérations élémentaires à chacune des opérations comptées. Nous y reviendrons, mais comptons d'abord les comparaisons effectuées par un appel à `insert`.

$$I(0) = 0 \qquad\qquad 1 \leq I(k+1) \leq k$$

On encadre ensuite facilement le nombre de comparaisons effectuées par `insertionSort` pour une liste de taille $n+1$.

$$n \leq S(n+1) \leq \frac{n(n+1)}{2}$$

Par conséquent $S(n)$ effectue au plus de l'ordre de n^2 comparaisons et au moins de l'ordre de n comparaisons. « De l'ordre de » a ici le sens précis que nous avons trouvé des ordres de grandeur asymptotiques pour n tendant vers $+\infty$. Plus exactement encore, $S(n)$ est bornée supérieurement par une fonction en $\Theta(n^2)$ et bornée inférieurement par une une fonction en $\Theta(n)$. Il est rappelé (cf. le cours précédent) que f est dans $\Theta(g)$ quand il existe deux constantes k_1 et k_2 telles que, pour n assez grand, on a

$$k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$$

Par ailleurs, nous disposons d'une estimation réaliste du coût global. En effet, d'une part, à une comparaison effectuée dans un appel donné à `insert` nous associons toutes les opérations du corps de cette méthode ; et d'autre part, le coût des opérations en temps constant effectuées lors d'une itération de la boucle de `insertionSort` peut être affecté aux comparaisons effectuées par l'appel à `insert` de cette itération.

Exercice 7 Donner des expériences particulières qui conduisent d'une part à un coût en n^2 et d'autre à un coût en n. Estimer le coût du tri de listes d'entiers tirés au hasard.

Solution. Trier des listes de taille croissantes et déjà triées en ordre croissant donne un coût en n^2, car les insertions se font toujours à la fin de la liste `r`. Mais si les listes sont triées en ordre décroissant, alors les insertion se font toujours au début de la liste `r` et le coût est en n. Enfin dans une liste quelconque, les insertions se feront plutôt vers le milieu de la liste `r` et le coût sera plutôt en n^2.

Plus rigoureusement, on peut estimer le nombre moyen de comparaisons, au moins dans le cas plus facile du tri des listes de n entiers deux à deux distincts. Considérons l'insertion du k-ième élément dans la liste `r` triée qui est de taille $k-1$. Clairement, parmi toutes les permutations de k éléments distincts, les sites d'insertion du dernier élément parmi les $k-1$ premiers triés se répartissent uniformément entre les k sites possibles. Dans ces conditions, si les permutations sont équiprobables, le nombre moyen de comparaisons pour une insertion est donc

$$\frac{1}{k}(1+2+\cdots+k-1+k-1) = \frac{1}{k}\left(\frac{k(k+1)}{2} - 1\right)$$

Par ailleurs, les permutations de n éléments distincts se repartissent uniformément selon les ensembles de leurs k premiers éléments. On a donc le coût en moyenne

$$\hat{S}(n) = 0 + \frac{1}{2}\left(\frac{2(2+1)}{2} - 1\right) + \cdots + \frac{1}{k}\left(\frac{k(k+1)}{2} - 1\right) + \cdots + \frac{1}{n}\left(\frac{n(n+1)}{2} - 1\right)$$

Autrement dit

$$\hat{S}(n) = \frac{(n+1)(n+2)}{4} - \ln n + O(1) = \frac{1}{4}n^2 + \frac{3}{4}n - \ln n + O(1)$$

Soit finalement le coût moyen est en $\Theta(n^2)$. On note au passage que le coût moyen est à peu près en rapport moitié du coût dans le cas le pire, ce qui confirme peut-être le résultat de notre premier raisonnement hasardeux. □

On peut pareillement estimer le coût en mémoire. Ici on peut se contenter de compter les cellules de liste allouées, et voir qu'un tri alloue au plus de l'ordre de n^2 cellules et au moins de l'ordre de n cellules.

3.2 Tri fusion

Le tri par insertion résulte essentiellement de l'opération d'insertion dans une liste triée. Un autre tri plus efficace résulte de l'opération de *fusion* de deux listes triés. Par définition, la fusion de deux listes triées est une liste triée qui regroupe les éléments des deux listes fusionnées. Cette opération peut être programmée efficacement, mais commençons par la programmer correctement. Sans que la programmation soit bien difficile, c'est notre premier exemple de fonction qui opère par induction sur deux listes. Soit $M(X, Y)$ la fusion (M pour *merge*). Il y a deux cas à considérer

Base De toute évidence, $M(\emptyset, Y) = Y$ et $M(X, \emptyset) = X$

Induction On a $X = (x, X')$ et $Y = (y, Y')$. On distingue deux cas à peu près symétriques.

- Si $x \leq y$, alors x minore non seulement tous les éléments de X' (X est triée) mais aussi tous les éléments de Y (transitivité de $\leq$ et Y est triée). Procédons à l'appel récursif $M(X', Y)$, qui par hypothèse d'induction renvoie une liste triée regroupant les éléments de X' et de Y. Donc, x minore tous les éléments de $M(X', Y)$ c'est à dire que la liste $(x, M(X', Y))$, qui regroupe bien tous les éléments de X et Y, est triée. On pose donc

$$M(X, Y) = (x, M(X', Y)), \quad \text{quand } X = (x, X'),\ Y = (y, Y') \text{ et } x \leq y$$

- Sinon, alors $y < x$ et donc $y \leq x$. On raisonne comme ci-dessus, soit on pose

$$M(X, Y) = (y, M(X, Y')), \quad \text{quand } X = (x, X'),\ Y = (y, Y') \text{ et } x > y$$

Et en Java, il vient

```
static List merge(List xs, List ys) {
  if (xs == null) {
    return ys ;
  } else if (ys == null) {
    return xs ;
  } // NB: désormais xs != null && ys != null
    else if (xs.val <= ys.val) {
    return new List (xs.val, merge(xs.next, ys)) ;
  } else { // NB: ys.val < xs.val
    return new List (ys.val, merge(xs, ys.next)) ;
  }
}
```

On observe que la méthode `merge` termine toujours, car les appels récursifs s'effectuent sur une structure plus petite que la structure passée en argument, ici une paire de listes.

Estimons le coût de la fusion de deux listes de tailles respectives n_1 et n_2, en comptant les comparaisons entre éléments. Au mieux, tous les éléments d'une des listes sont inférieurs à ceux de l'autre liste. Au pire, il y a une comparaison par élément de la liste produite.

$$\min(n_1, n_2) \leq M(n_1, n_2) \leq n_1 + n_2$$

Par ailleurs $M(n_1, n_2)$ est une estimation raisonnable du coût global, puisque compter les comparaisons revient quasiment à compter les appels et que, hors un appel récursif, un appel donné effectue un nombre borné d'opérations toutes en coût constant.

Trier une liste `xs` à l'aide de la fusion est un bon exemple du principe *diviser pour résoudre*. On sépare d'abord `xs` en deux listes, que l'on trie (inductivement) puis on fusionne les deux listes. Pour séparer `xs` en deux listes, on peut regrouper les éléments de rangs pair et impair dans respectivement deux listes, comme procède le début du code de la méthode `mergeSort`.

```
static List mergeSort(List xs) {
  List ys = null, zs = null ;
  boolean even = true ; // zéro est pair
  for (List p = xs ; p != null ; p = p.next ) {
    if (even) {
      ys = new List (p.val, ys) ;
    } else {
      zs = new List (p.val, zs) ;
    }
    even = !even ; // k+1 a la parité opposée à celle de k
  }
  ⋮
```

Il reste à procéder aux appels récursifs et à la fusion, en prenant bien garde d'appeler récursivement `mergeSort` exclusivement sur des listes de longueur strictement plus petite que la longueur de `xs`.

```
  ⋮
  if (zs == null) { // xs a zéro ou un élément
    return xs ; // et alors xs est triée
  } else { // Les longueurs de ys et sz sont < à la longueur de xs
    return merge(mergeSort(ys), mergeSort(zs)) ;
  }
}
```

Comptons les comparaisons qui sont toutes effectuées par la fusion :

$$S(0) = 0 \qquad S(1) = 0 \qquad S(n) = M(\lceil n/2 \rceil, \lfloor n/2 \rfloor) + S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor), \text{ pour } n > 1$$

Soit l'encadrement

$$\lfloor n/2 \rfloor + S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) \leq S(n) \leq n + S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor)$$

Si $n = 2^{p+1}$ on a la simplification très nette

$$2^p + 2 \cdot S(2^p) \leq S(2^{p+1}) \leq 2^{p+1} + 2 \cdot S(2^p)$$

Ou encore

$$1/2 + S(2^p)/2^p \leq S(2^{p+1})/2^{p+1} \leq 1 + S(2^p)/2^p$$

Les récurrences sont de la forme $T(0) = 0$, $T(p+1) = k + T(p)$ de solution $k \cdot p$. On a donc l'encadrement

$$1/2 \cdot p \cdot 2^p \leq S(2^p) \leq p \cdot 2^p$$

Soit un coût de l'ordre de $n \log n$ qui se généralise à n quelconque (voir l'exercice). Le point important est que le tri fusion effectue toujours de l'ordre de $n \log n$ comparaisons d'éléments. Par ailleurs, le compte des comparaisons estime le coût global de façon réaliste, puisque que l'on peut affecter le coût de entre une et deux itérations de la boucle de séparation à chaque comparaison de la fusion.

Exercice 8 (Franchement optionnel) Généraliser l'ordre de grandeur du compte des comparaisons à n quelconque.

Solution. Définissons une première suite $U(n)$ pour minorer $S(n)$

$$U(0) = 0 \qquad U(1) = 0 \qquad U(2q+2) = q+1+2 \cdot U(q+1) \qquad U(2q+3) = q+1+U(q+1)+U(q+2)$$

Notons que nous avons simplement détaillé

$$U(n) = \lfloor n/2 \rfloor + U(\lfloor n/2 \rfloor) + U(\lceil n/2 \rceil)$$

Si on y tient on peut montrer $U(n) \leq S(n)$ par une induction facile. Par ailleurs on sait déjà que $U(2^p)$ vaut $1/2 \cdot p \cdot 2^p$. Toute l'astuce est de montrer que pour p assez grand et n tel que $2^p \leq n < 2^{p+1}$, on a $U(2^p) \leq U(n) < U(2^{p+1})$ Posons $D(n) = U(n+1) - U(n)$. Il vient alors

$$D(0) = 0 \qquad D(1) = 1 \qquad D(2q+2) = D(q+1) \qquad D(2q+3) = 1 + D(q+1)$$

La suite $D(n)$ est donc de toute évidence à valeurs dans $\mathbb{N}$ (c'est-à-dire $D(n) \geq 0$) et on a certainement $D(n) > 0$ pour n impair. Ce qui suffit pour prouver l'encadrement pour tout p. En effet $U(2^p) \leq U(n)$ résulte de $D(n) \geq 0$, tandis $U(n) < U(2^{p+1})$ résulte de ce que $2^{p+1} - 1$ est impair.

Soit maintenant $n > 0$, il existe un unique p, avec $2^p \leq n < 2^{p+1}$ et on a donc

$$1/2 \cdot p \cdot 2^p \leq U(n)$$

Par ailleurs, la fonction $x \cdot \log_2(x)$ est croissante. Or, pour $n > 1$ on a $n/2 < 2^p$, il vient donc

$$1/2 \cdot \log_2(n/2) \cdot n/2 \leq U(n)$$

De même, on borne supérieurement $S(n)$ par la suite

$$V(0) = 0 \qquad V(1) = 0 \qquad V(n) = n + V(\lceil n/2 \rceil) + V(\lfloor n/2 \rfloor))$$

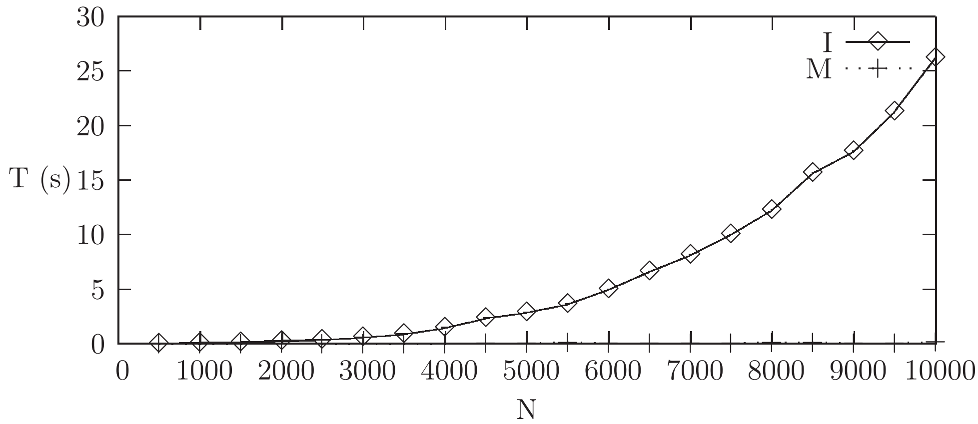
Suite dont on montre qu'elle est croissante, puis majorée par $\log_2(2n) \cdot 2n$. Soit enfin

$$1/2 \cdot \log_2(n/2) \cdot n/2 \leq S(n) \leq \log_2(2n) \cdot 2n$$

On peut donc finalement conclure que $S(n)$ est en $\Theta(n \cdot \log n)$, ce qui est l'essentiel. Si on souhaite estimer les constantes cachées dans le Θ et pas seulement l'ordre de grandeur asymptotique, notre encadrement est sans doute un peu large. □

3.3 Trier de grandes listes

Pour confirmer le réalisme des analyses de complexité des tris, nous trions des listes de N entiers et mesurons le temps d'exécution des deux tris par insertion I et fusion M. Les éléments des listes sont les N premiers entiers dans le désordre.



Cette expérience semble confirmer que le tri par insertion est quadratique. Elle montre surtout que le tri fusion est bien plus efficace que le tri par insertion.

Encouragés, nous essayons de mesurer le temps d'exécution du tri fusion pour des valeurs de N plus grandes. Nous obtenons :

```
N=10000 T=0.093999
N=20000 T=0.411111
N=30000 T=0.609787
Exception in thread "main" java.lang.StackOverflowError
        at List.merge(List.java:80)
        at List.merge(List.java:78)
  ⋮
```

Où de l'ordre de 1000 lignes `at List.merge`... sont effacées.

On voit que notre programme échoue en signalant que l'exception **StackOverflowError** (débordement de la pile) a été lancée. Nous expliquerons plus tard précisément ce qui se passe. Pour le moment, contentons nous d'admettre que le nombre d'appels de méthode en attente est limité. Or ici, `merge` appelle `merge` qui appelle `merge` qui etc. et `merge` doit attendre le retour de `merge` qui doit attendre le retour de `merge` qui etc. Et nous touchons là une limite de la programmation récursive. En effet, alors que nous avons programmé une méthode qui semble théoriquement capable de trier des centaines de milliers d'entiers, nous ne pouvons pas l'exécuter au delà de 30000 entiers.

Pour éviter les centaines de milliers d'appels imbriqués de `merge` nous pouvons reprogrammer la fusion itérativement. On emploie la technique de l'initialisation différée (voir l'exercice 2), puisqu'il est ici crucial de construire la liste résultat selon l'ordre des listes consommées.

```
static List merge(List xs, List ys) {
  if (xs == null) return ys ;
  if (ys == null) return xs ;
  /* Ici le resultat a une première cellule */
  /* reste à trouver ce qui va dedans */
  List r = null ;
  if (xs.val <= ys.val) {
    r = new List (xs.val) ; xs = xs.next ;
```

```
  } else {
    r = new List (ys.val) ; ys = ys.next ;
  }
  List last = r ; // Dernière cellule de la liste résultat
  /* Régime permanent */
  while (xs != null && ys != null) {
    if (xs.val <= ys.val) {
      last.next = new List (xs.val) ; xs = xs.next ;
    } else {
      last.next = new List (ys.val) ; ys = ys.next ;
    }
    last = last.next ; // Dernière cellule à nouveau
  }
  /* Ce n'est pas fini, une des deux listes peut ne pas être vide */
  if (xs == null) {
    last.next = ys ;
  } else {
    last.next = xs ;
  }
  return r ;
}
```

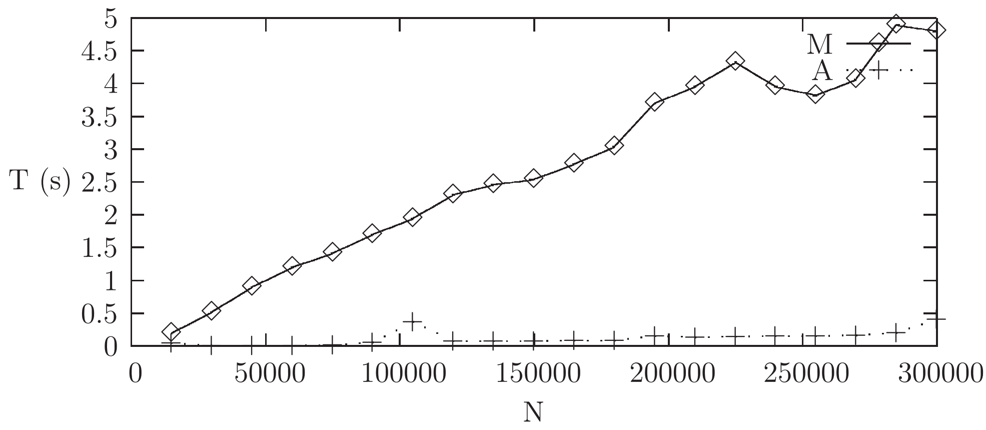
La nouvelle méthode `merge` n'a rien de difficile à comprendre, une fois assimilée l'initialisation différée. Il faut aussi ne pas oublier d'initialiser le champ `next` de la toute dernière cellule allouée.

Mais attendons ! Le nouveau tri peut il fonctionner en pratique, puisque `mergeSort` elle même est récursive ? Oui, car `mergeSort` s'appelle sur des listes de longueur divisée par deux, et donc le nombre d'appels imbriqués de `mergeSort` vaut au plus à peu près $log_2(N)$, certainement inférieur à 32 dans nos expériences où N est un **int**. Ceci illustre qu'il n'y a pas lieu de « dérécursiver » systématiquement.

Plutôt que de simplement mesurer le temps d'exécution du nouveau `mergeSort`, nous allons le comparer à un autre tri : celui de la bibliothèque de Java. La méthode statique `sort` (classe **Arrays**, package `java.util`) trie un tableau de **int** passé en argument. Puisque nous tenons à trier une liste, il suffit de changer cette liste en tableau, de trier le tableau, et de changer le tableau en liste.

```
static List arraySort(List xs) {
  // Copier les eléments de xs dans un tableau t
  int sz = card(xs) ; // Longueur de la liste
  int [] t = new int[sz] ;
  int k = 0 ;
  for (List p = xs ; xs != null ; xs = xs.next, k++) {
    t[k] = xs.val ;
  }
  // Trier t
  java.util.Arrays.sort(t) ;
  // Fabriquer une liste avec les éléments de t, attention à l'ordre !
  List r = null ;
  for (k = sz-1 ; k >= 0 ; k--) {
    r = new List(t[k], r) ;
  }
  return r ;
}
```

Employer une méthode de tri de la bibliothèque est probablement une bonne idée, car elle est écrite avec beaucoup de soin et est donc très efficace [1]. Observons qu'en supposant, et c'est plus que raisonnable, un coût du tri en $O(N \log N)$, qui domine asymtotiquement nos transferts de liste en tableau et réciproquement (en $O(N)$), on obtient un tri en $O(N \log N)$. Dans les mesures, M est notre nouveau tri fusion, tandis que A est le tri `arraySort`.



On voit que notre méthode `mergeSort` conduit à un temps d'exécution qui semble à peu près linéaire, mais qu'il reste des progrès à accomplir. De fait, en programmant un tri fusion destructif (voir [7, Chapitre 12] par ex.) qui n'alloue aucune cellule, on obtient des temps bien meilleurs. À vrai dire, les temps sont de l'ordre du double de ceux du tri de la bibliotèque.

4 Programmation objet

Dans tout ce chapitre nous avons exposé la structure des listes. C'est-à-dire que les programmes qui se servent de la classe des listes le font directement, en pleine conscience qu'ils manipulent des structures de listes. Cette technique est parfaitement correcte et féconde, mais elle ne correspond pas à l'esprit d'un langage objet. Supposons par exemple que nous voulions implémenter les ensembles d'entiers. Il est alors plus conforme à l'esprit de la programmation objet de définir un objet « ensemble d'entiers » **Set**. Cette mise en forme « objet » n'abolit pas la distinction fondamentale des deux sortes de structures de données, mutable et non mutable, bien au contraire.

4.1 Ensembles non mutables

Spécifions donc une classe **Set** des ensembles.

```
class Set {
  Set() { ... }
  Set add(int x) { ... }
  Set remove(int x) { ... }
}
```

Comparons la signature de méthode `add` des **Set** à celle de la méthode `add` de la classe **List**.

```
  static List add(List xs, int x)
```

La nouvelle méthode `add` n'est plus statique, pour produire un ensemble augmenté, on utilise maintenant la notation objet : par exemple s`.add(1)` renvoie le nouvel ensemble $s \cup \{1\}$ — alors que précédemment on écrivait **List** `.add(`s`, 1)` (voir section 2.2). On note que pour pouvoir appeler la méthode `add` sur tous les objets **Set**, il n'est plus possible d'encoder l'ensemble vide par **null**. L'ensemble vide est maintenant un objet construit par **new Set** `()`.

Bien entendu, il faut qu'une structure concrète regroupe les éléments des ensembles. Le plus simple est d'*encapsuler* une liste dans chaque objet **Set**.

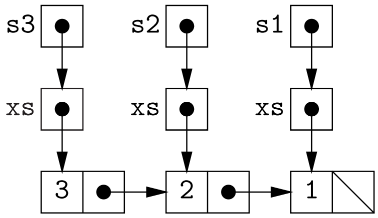
```
class Set {
  // Partie privée
  private List xs ;
  private Set (List xs) { this.xs = xs ; }
  // Partie visible
  Set () { xs = null ; }
  Set add(int x) { return new Set (List.add(xs, x)) ; }
  Set remove(int x) { return new Set (List.remove(xs, x)) ; }
}
```

La variable d'instance `xs` et le constructeur qui prend une liste en argument sont privés, de sorte que les clients de `Set` ne peuvent pas manipuler la liste cachée. Les « clients » sont les parties du programme qui appellent les méthodes de la classe `Set`.

Comme le montre l'exemple suivant, il importe que les listes soient traitées comme des listes non-mutables. En effet, soit le bout de programme

```
Set s0 = new Set () ;
Set s1 = s0.add(1) ;
Set s2 = s1.add(2) ;
Set s3 = s2.add(3) ;
```

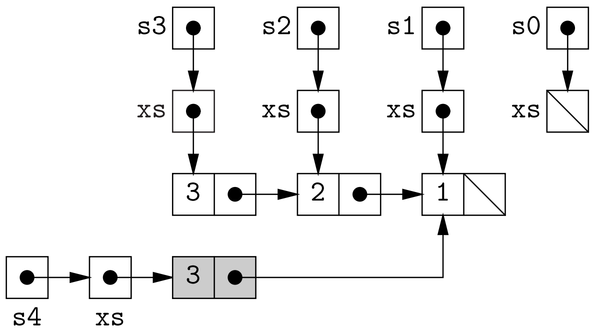
Nous obtenons l'état mémoire simplifié



La méthode `remove` (page 12) ne modifie pas les champs des cellules de listes, de sorte que que le programme

```
Set s4 = s3.remove(2) ;
```

n'entraîne aucune modification des ensembles `s2`, `s1` et `s0`.



Il semble bien que ce style d'encapsulage d'une structure non-mutable est peu utilisé en pratique. En tout cas, la bibliothèque de Java ne propose pas de tels ensembles « fonctionnels ». Les deux avantages de la technique, à savoir une meilleure structuration, et une représentation uniforme des ensembles, y compris l'ensemble vide, comme des objets, ne justifient sans doute pas la cellule de mémoire supplémentaire allouée systématiquement pour chaque ensemble. L'encapsulage se rencontre plus fréquemment quand il s'agit de protéger une structure mutable.

4.2 Ensembles mutables

On peut aussi vouloir modifier un ensemble et non pas en construire un nouveau. Il s'agit d'un point de vue impératif, étant donné un ensemble s, appliquer l'opération `add(int x)` à s, ajoute l'élément x à s. Cette idée s'exprime simplement en Java, en faisant de s un objet (d'une nouvelle classe `Set`) et de `add` une méthode de cet objet. On ajoute alors un élément x à s par un appel de méthode s`.add(`x`)`, et la signature de `add` la plus naturelle est

```
void add(int x)
```

La nouvelle signature de `add` révèle l'intention impérative : s connu est modifié et il n'y a donc pas lieu de le renvoyer après modification.

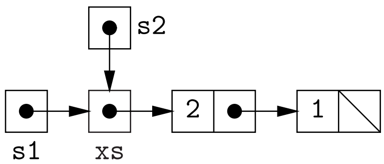
Pour programmer la nouvelle classe `Set` des ensembles impératifs, on a recours encore une fois à la technique d'encapsulage. Les objets de la classe `Set` possèdent une variable d'instance qui est une `List`. Mais cette fois, les méthodes des objets `Set` modifient la variable d'instance.

```
class Set {
  // Partie privée
  private List xs ;
  // Partie accessible
  Set () { this.xs = null ; }
  void add(int x) { xs = List.add (x, xs) ; }
  void remove(int x) { xs = List.nremove(x, xs) ; }
  public String toString() { return List.toString(xs) ; }
}
```

Le champ `xs` est privé, afin de garantir que la classe `Set` est la seule à manipuler cette liste. Cette précaution autorise, mais n'impose en aucun cas, l'emploi des listes mutables au prix d'un risque modéré, comme dans la méthode `remove` qui appelle `List.nremove`. Le caractère impératif des ensembles `Set` est clairement affirmé et on ne sera (peut-être) pas surpris par l'affichage du programme suivant :

```
Set s1 = new Set() ;
Set s2 = s1 ;
s1.add(1) ; s1.add(2) ;
System.out.println(s2) ;
```

Le programme affiche `{1, 2}` car `s1` et `s2` sont deux noms du même ensemble impératif.



Plus généralement, en utilisant une structure de donnée impérative, on souhaite faire profiter l'ensemble du programme de toutes les modifications réalisées sur cette structure. Cet effet ici recherché est facilement atteint en mutant le champ privé `xs` des objets `Set`, c'est la raison même de l'encapsulage.

Exercice 9 Enrichir les objets de la classe `Set` avec une méthode `diff` de signature :

```
void diff(Set s) ; // enlever les éléments de s
```

Aller d'abord au plus simple, puis utiliser les structures de données mutables. Ne pas oublier de considérer le cas s`.diff(`s`)`

Solution. On peut écrire d'abord

```
void diff(Set s) { this.xs = List.diff(this.xs, s.xs) ; }
```

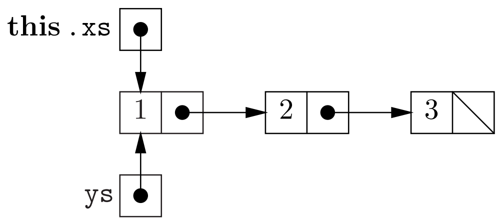
Où `List.diff` est non destructive (voir exercice 3), et où on a écrit `this.xs` au lieu de `xs` pour bien insister.

Utilsons maintenant les listes mutables. Compte tenu de la sémantique, il semble légitime de modifier les cellules accessibles à partir de `this.xs`, mais pas celles accessibles à partir de `s.xs`. On tente donc :

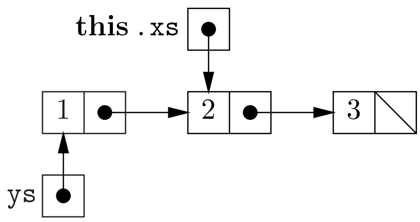
```
void diff(Set s) {
  for (List ys = s.xs ; ys != null ; ys = ys.next)
    this.xs = List.nremove(ys.val, this.xs) ;
}
```

L'emploi d'une nouvelle variable `ys` s'impose, car il ne faut certainement pas modifier `s.xs`. Plus précisément on ne doit *pas* écrire `for ( ; s.xs != null ; s.xs = s.xs.next )`. Notons au passage que la déclaration `private List xs` n'interdit pas la modification de `s.xs` : le code de `diff` se trouve dans la classe `Set` et il a plein accès aux champs privés de tous les objets de la classe `Set`.

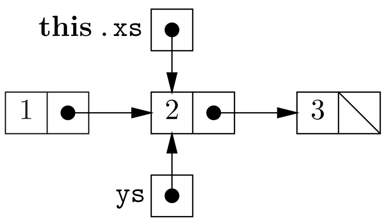
On vérifie que l'auto-appel s`.diff(`s`)` est correct, même si c'est de justesse. Supposons par exemple que `xs` pointe vers la liste $(1,(2,(3,\emptyset)))$. Au début de la première itération, on a

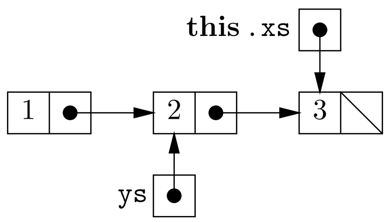


Dans ces conditions, enlever `ys.val` (1) de la liste `this.xs` revient simplement à renvoyer `this.xs.next`. Et à la fin de première itération, on a

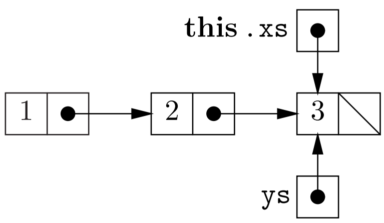


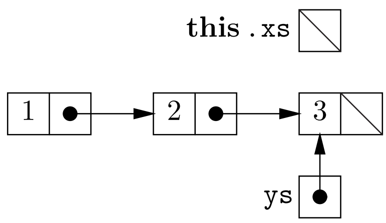
Les itérations suivantes sont similaires : on enlève toujours le première élément de la liste pointée par `this.xs`, de sorte qu'en début puis fin de seconde itération, on a





Puis au début et à la fin de la dernière itération, on a

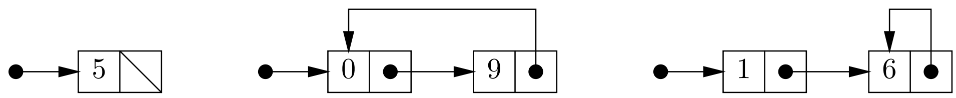




Et donc, **this**.xs vaut bien **null** en sortie de méthode `diff`, ouf. Tout cela est quand même assez acrobatique. □

5 Complément : listes bouclées

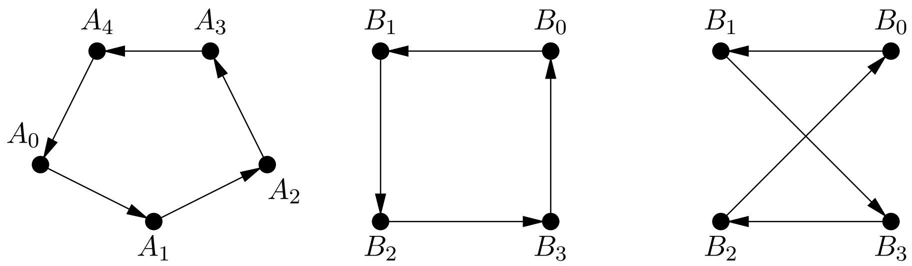
Des éléments peuvent être organisés en séquence, sans que nécessairement la séquence soit finie. Si la séquence est périodique, elle devient représentable par une liste qui au lieu de finir, repointe sur une de ses propres cellules. Par exemple, on peut être tenté de représenter les décimales de $1/2 = 0.5$, $1/11 = 0.090909\cdots$ et $1/6 = 0.16666\cdots$ par



Nous distinguons ici, d'abord une liste finie, ensuite une liste *circulaire* — le retour est sur la première cellule, et enfin une liste *bouclée* la plus générale.

5.1 Représentation des polygones

Plus couramment, il est pratique de représenter un polygone fermé par la liste circulaire de ses sommets. La séquence des points est celle qui correspond à l'organisation des sommets et des arêtes.



Les polygones étant fermés, les listes circulaires sont naturelles. L'ordre des sommets dans la liste est significatif, ainsi les polygones $(B_0B_1B_2B_3)$ et $(B_0B_1B_3B_2)$ sont distincts.

Pour représenter les polygones en machine, nous définissons une classe des cellules de liste de points.

```
import java.awt.* ;

class Poly {
  private Point val ; private Poly next ;

  private Poly (Point val, Poly next) { this.val = val ; this.next = next ; }
  ⋮
}
```

Tout est privé, compte tenu des manipulations dangereuses auxquelles nous allons nous livrer. La classe **Point** des points est celle de la bibliothèque (package `java.awt`), elle est sans surprise : p.x et p.y sont les coordonnées, p.distance(q) renvoie la distance entre les points p et q.

Nous allons concevoir et implémenter un algorithme de calcul de l'enveloppe convexe d'un nuage de points, algorithme qui s'appuie naturellement sur la représentation en machine des polygones (convexes) par les listes circulaires. Mais commençons d'abord par nous familiariser tant avec la liste circulaire, qu'avec les quelques notions de géométrie plane nécessaires. Pour nous simplifier un peu la vie nous allons poser quelques hypothèses.

- Les polygones ont toujours au moins un point. Par conséquent, **null** n'est pas un polygone.
- Les points sont en *position générale*. Ce qui veut dire que les points sont deux à deux distincts, et qu'il n'y a jamais trois points alignées.
- Le plan est celui d'une fenêtre graphique Java, c'est à dire que l'origine des coordonnées est en haut et à gauche. Ce qui entraîne que le repère est en sens inverse du repère usuel du plan (l'axe des y pointe vers le bas).

5.2 Opérations sur les polygones

L'intérêt de la liste circulaire apparaît pour, par exemple, calculer le périmètre d'un polygone. Pour ce faire, il faut sommer les distances (A_0A_1), (A_1A_2), $\ldots (A_{n-1}A_0)$. Si nous représentions le polygone par la liste ordinaire de ses points, le cas de la dernière arête $(A_{n-1}A_0)$ serait particulier. Avec une liste circulaire, il n'en est rien, puisque le champ `next` de la cellule du point A_{n-1} pointe vers la cellule du point A_0. La sommation se fait donc simplement en un parcours des sommets du polygone. Afin de ne pas sommer indéfiniment, il convient d'arrêter le parcours juste après cette « dernière » arête, c'est à dire quand on retrouve la « première » cellule une seconde fois.

```
static double perimeter(Poly p) {
  double r = 0.0 ;
  Poly q = p ;
  do {
    r += q.val.distance(q.next.val) ;
    q = q.next ;
  } while (q != p) ; // Attention : différence des références (voir B.3.1.1)
  return r ;
}
```

L'usage d'une boucle **do** {...} **while** (...) (section B.3.4) est naturel. En effet, on veut finir d'itérer quand on voit la première cellule une seconde fois, et non pas une première fois. On observe que le périmètre calculé est 0.0 pour un polygone à un sommet et deux fois la distance (A_0A_1) pour deux sommets.

Nous construisons un polygone à partir de la séquence de ses sommets, extraite d'un fichier ou bien donnée à la souris. La lecture des sommets se fait selon le modèle du **Reader** (voir section B.6.2.1). Soit donc une classe PointReader des lecteurs de points pour lire les sources de points (fichier ou interface graphique). Ses objets possèdent une méthode `read` qui consomme et renvoie le point suivant de la source, ou **null** quand la source est tarie (fin de fichier, double clic). Nous écrivons une méthode de construction du polygone formé par une source de points.

```
static Poly readPoly(PointReader in) {
   Point pt = in.read() ;
   if (pt == null) throw new Error ("Polygone vide interdit") ;

   Poly r = new Poly (pt, readPoints(in)) ;
   nappend(r,r) ; // Pour refermer la liste, voir exercice 4
   return r ;
}

private static Poly readPoints(PointReader in) {
  Point pt = in.read() ;
  if (pt == null) {
    return null ;
  } else {
    return new Poly(pt, readPoints(in)) ;
  }
}
```

La programmation est récursive pour facilement préserver l'ordre des points de la source. La liste est d'abord lue puis refermée. C'est de loin la technique la plus simple. Conformément à nos conventions, le programme refuse de construire un polygone vide. À la place, il lance l'exception **Error** avec pour argument un message explicatif. L'effet du lancer d'exception peut se résumer comme un échec du programme, avec affichage du message (voir B.4.2 pour les détails).

Nous nous posons maintenant la question d'identifier les polygones convexes. Soit maintenant P un polygone. La paire des points p.val et p.next.val est une arête (A_0A_1), c'est-à-dire un segment de droite orienté. Un point quelconque Q, peut se trouver à droite ou à gauche du segment (A_0A_1), selon le signe du déterminant $\det(A_0A_1, A_0P)$. La méthode suivante permet donc d'identifier la position d'un point relativement à la première arête du polygone P.

```
private static int getSide(Point a0, Point a1, Point q) {
  return
    (q.y - a0.y) * (a1.x - a0.x) - (q.x - a0.x) * (a1.y - a0.y) ;
}

static int getSide(Point q, Poly p) { return getSide (p.val, p.next.val, q) ; }
```

Par l'hypothèse de position générale, `getSide` ne renvoie jamais zéro. Une vérification rapide nous dit que, dans le système de coordonnées des fenêtres graphiques, si getSide(q, p) est négatif, alors le point q est *à gauche* de la première arête du polygone p.

Un polygone est convexe (et bien orienté) quand tous ses sommets sont à gauche de toutes ses arêtes. La vérification de convexité revient donc à parcourir les arêtes du polygone et, pour chaque arête à vérifier que tous les sommets autres que ceux de l'arête sont bien à gauche.

```
static boolean isConvex(Poly p) {
  Poly a = p ; // a est l'arête courante
  do {
    Poly s = a.next.next ; // s est le sommet courant
    while (s != a) {
      if (getSide(s.val, a) > 0) return false ;
      s = s.next ; // sommet suivant
    }
    a = a.next ; // arête suivante
  } while (a != p) ;
  return true ;
}
```

On note les deux boucles imbriquées, une boucle sur les sommets dans une boucle sur les arêtes. La boucle sur les sommets est une boucle **while** par souci esthétique de ne pas appeler `getSide` avec pour arguments des points qui ne seraient pas en position générale.

Exercice 10 Que se passe-t-il si p est un polygone à un, à deux sommets ?

Solution. Dans les deux cas, aucune itération de la boucle intérieure sur les sommets n'est exécutée. Par conséquent les polygones à un et deux sommets sont jugés convexes et bien orientés. □

Pour un polygone à n sommets, la boucle sur les arêtes est exécutée n fois, et à chaque fois, la boucle sur les sommets est exécutée $n-2$ fois. La vérification de convexité coûte donc de l'ordre de n^2 opérations élémentaires.

Notons que la méthode `getSide` nous permet aussi de déterminer si un point est à l'intérieur d'un polygone convexe.

```
static boolean isInside(Point q, Poly p) {
```

```
  Poly a = p ;
  do {
    if (getSide(q, a) > 0) return false
    a = a.next ;
  } while (a != p) ;
  return true ;
}
```

Notons que la convexité permet une définition et une détermination simplifiée de l'intérieur. La méthode `isInside` a un coût de l'ordre de n opération élémentaires, où n est la taille du polygone.

Exercice 11 Que se passe-t-il si p est un polygone à un, à deux sommets ?

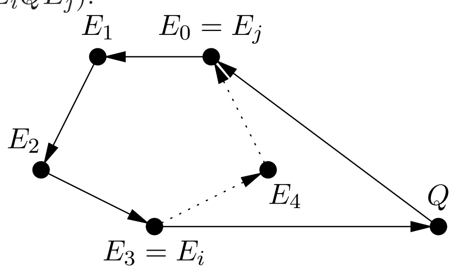
Solution. Le cas limite du polygone à deux sommets est bien traité : pour trois points en position générale, l'un des deux appels à `getSide` renvoie nécessairement un entier positif. Par conséquent `isInside` renvoie nécessairement **false**, ce qui est conforme à l'intuition que l'intérieur d'un polygone à deux sommets est vide.

Il n'en va pas de même du polygone à un seul sommet, dont l'intérieur est *a priori* vide. Or, dans ce cas, `getSide` est appelée une seule fois, les deux sommets de l'arête étant le même point. La méthode `getSide` renvoie donc zéro, et `isInside` renvoie toujours **true**. □

5.3 Calcul incrémental de l'enveloppe convexe

L'enveloppe convexe d'un nuage de points est le plus petit polygone convexe dont l'intérieur contient tous les points du nuage. Pour calculer l'enveloppe convexe, nous allons procéder incrémentalement, c'est-a-dire, en supposant que nous possédons déjà l'enveloppe convexe des points déjà vus, construire l'enveloppe convexe des points déjà vus plus un nouveau point. Soit donc la séquence de points $P_0, P_1, \ldots P_k$, dont nous connaissons l'enveloppe convexe $E = (E_0 E_1 \ldots E_c)$, Et soit $Q = P_{k+1}$ le nouveau point.

- Si Q est à l'intérieur de E, l'enveloppe convexe ne change pas.
- Sinon, il faut identifier deux sommets E_i et E_j, tels que tous les points de E sont à gauche de $(E_i Q)$ et $(Q E_j)$. La nouvelle enveloppe convexe est alors obtenue en remplaçant la séquence $(E_i \ldots E_j)$ par $(E_i Q E_j)$.



Sans réaliser une démonstration complète, il semble clair que Q se trouve à droite des arêtes supprimées et à gauche des des arêtes conservées. Mieux, lorsque l'on parcourt les sommets du polygone, le sommet E_i marque une première transition : Q est à gauche de $(E_{i-1} E_i)$ et à droite de $(E_i E_{i+1})$, tandis que le sommet E_j marque la transition inverse. On peut aussi imaginer que Q est une lampe, il faut alors supprimer les arêtes éclairées de l'enveloppe.

Il faut donc identifier les sommets de E où s'effectuent les transitions « Q à gauche puis à droite » et inversement. Le plus simple paraît de se donner d'abord une méthode `getRight` qui renvoie la « première » arête qui laisse Q à droite, à partir d'un sommet initial supposé quelconque.

```
static Poly getRight(Poly e, Point q) {
  Poly p = e ;
  do {
    if (getSide(q, p) > 0) return p ;
    p = p.next ;
  } while (p != e) ;
  return null ;
}
```

Notons que dans le cas où le point Q est à l'intérieur de E, c'est-à-dire à gauche de toutes ses arêtes, `getRight` renvoie **null**. On écrit aussi une méthode `getLeft` pour détecter la première arête de E qui laisse Q à gauche. Le code est identique, au test `getSide(q, p) > 0`, à changer en `getSide(q, p) < 0`, près.

La méthode `extend` d'extension de l'enveloppe convexe, trouve d'abord une arête qui laisse Q à droite, si une telle arête existe. Puis, le cas échéant, elle continue le parcours, en enchaînant un `getLeft` puis un `getRight` afin d'identifier les points E_j puis E_i qui délimitent la partie « éclairée » de l'enveloppe.

```
private static Poly extend(Poly e, Point q) {
  Poly oneRight = getRight(e, q) ;
  if (oneRight == null) return e ; // Le point q est intérieur
  // L'arête issue de oneRight est éclairée
  Poly ej = getLeft(oneRight.next, q) ; // lumière -> ombre
  Poly ei = getRight(ej.next, q) ;      // ombre -> lumière
  Poly r = new Poly(q, ej) ;
  ei.next = r ;
  return r ;
}
```

La structure de liste circulaire facilite l'écriture des méthodes `getRight` et `getLeft`. Elle permet aussi, une fois trouvés E_i et E_j, une modification facile et en temps constant de l'enveloppe convexe. Les deux lignes qui effectuent cette modification sont directement inspirées du dessin d'extension de polygone convexe ci-dessus. Il est clair que chacun des appels à `getRight` et `getLeft` effectue au plus un tour du polygone E. La méthode `extend` effectue donc au pire trois tours du polygone E (en pratique au plus deux tours, en raison de l'alternance des appels à `getRight` et `getLeft`). Elle est donc en $O(c)$, où c est le nombre de sommets de l'enveloppe convexe.

Il nous reste pour calculer l'enveloppe convexe d'un nuage de points lus dans un PointReader à initialiser le processus. On peut se convaincre assez facilement que `extend` fonctionne correctement dès que E possède deux points, car dans ce cas il existe une arête éclairée et une arête sombre. On écrit donc.

```
static Poly convexHull(PointReader in) {
  Point q1 = in.read() ;
  Point q2 = in.read() ;
  if (q1 == null || q2 == null) throw new (Error "Pas assez de points") ;

  Poly e = new Poly (q1, new Poly (q2, null)) ;
  nappend(e,e) ; // Refermer la liste
  for ( ; ; ) { // Idiome : boucle infinie
    Point q = in.read()
    if (q == null) return e ;
    e = extend(e, q) ;
  }
}
```

Pour un nuage de n points, on exécute de l'ordre de n fois `extend`, dont le coût est en $O(c)$, sur des enveloppes qui possèdent $c \leq n$ sommets. Le programme est donc en $O(n^2)$. Le temps d'exécution dépend parfois crucialement de l'ordre de présentation des points. Dans le cas où par exemple l'enveloppe convexe est formée par les trois premiers points lus, tous les points suivants étant intérieurs à ce triangle, alors on a un coût linéaire. Malheureusement, si tous les points de l'entrée se retrouvent au final dans l'enveloppe convexe, le coût est bien quadratique. La version web du cours pointe vers un programme de démonstration de cet algorithme.

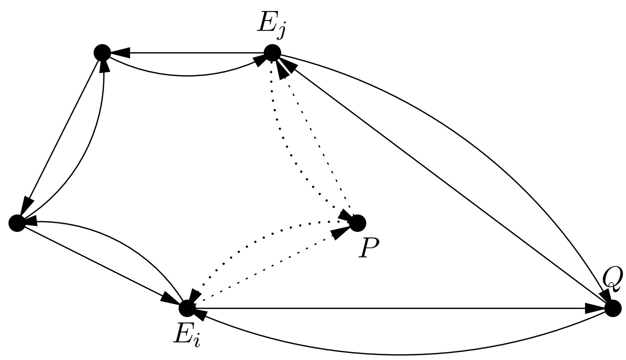
5.4 Calcul efficace de l'enveloppe convexe

L'enveloppe convexe d'un nuage de n points du plan peut se calculer en $O(n \log n)$ opérations, à l'aide de l'algorithme classique de la marche de Graham (*Graham scan*) Nous n'expliquerons pas la marche de Graham, voir par exemple [2, Chapitre 35] ou [7, Chapitre 25]. En effet, un raffinement de la technique incrémentale inspire un algorithme tout aussi efficace, dumoins asymptotiquement. De plus ce nouvel algorithme va nous permettre d'aborder une nouvelle sorte de liste.

Rappelons l'idée de l'algorithme incrémental, sous une forme un peu imagée. L'enveloppe convexe E est soumise à la lumière émise par le nouveau point Q. En raison de sa convexité même, E possède une face éclairée et une face sombre. La face éclairée va du sommet E_i au sommet E_j, et l'extension de l'enveloppe revient à remplacer la face éclairée par E_iQE_j. Supposons connaître un sommet P de l'enveloppe convexe qui est « éclairé » par le nouveau point Q (c'est-a-dire que l'arête issue de P laisse Q à droite). Le sens de rotation selon les `next` nous permet de trouver la fin de la face éclairée directement, c'est-à-dire sans passer par la face sombre. Pour pouvoir trouver le début de la face éclairée tout aussi directement, il faut pouvoir tourner dans l'autre sens ! Nous allons donc ajouter un nouveau champ `prev` à nos cellules de polygones, pour les chaîner selon l'autre sens de rotation.

```
class Poly {
  ⋮
  private Poly prev ;

  private Poly (Point val, Poly next, Poly prev) {
    this(val, next) ; // Appeler l'autre constructeur
    this.prev = prev ;
  }
  ⋮
}
```

Nos listes de sommets deviennent des listes doublement chaînées (et circulaires qui plus est), une structure de données assez sophistiquée, que nous pouvons illustrer simplement par le schéma suivant :



En suivant les champs `prev` (notés par des flèches courbes), nous trouvons facilement la première arête inverse sombre à partir du point P, c'est-à-dire le premier sommet E_i dont est issue une arête inverse qui laisse le point Q à droite.

```
static int getSidePrev(Poly a, Point q) {
  return getSide(a.val, a.prev.val, q) ;
}

static Poly getRightPrev(Poly e, Point q) {
  Poly p = e ;
  for ( ; ; ) {
    if (getSidePrev(q, p) > 0) return p ;
    p = p.prev ;
  }
}
```

Et nous pouvons écrire la nouvelle méthode d'extension de l'enveloppe convexe.

```
// Attention e.val doit être « eclairé » par q (ie getSide(e,q) > 0)
private static Poly extend2(Poly e, Point q) {
  Poly ej = getLeft(e.next, q) ;
  Poly ei = getRightPrev(e, q) ;
  Poly r = new Poly(q, ej, ei) ;
  ei.next = r ; ej.prev = r ;
  return r ;
}
```

La méthode `extend2` identifie la face éclairée pour un coût proportionnel à la taille de cette face. Puis, elle construit la nouvelle enveloppe en remplaçant la face éclairée par (E_iQE_j). Les manipulations des champs `next` et `prev` sont directement inspirées par le schéma ci-dessus.

La méthode n'est correcte que si Q éclaire l'arête issue de P. Ce sera le cas, quand

(1) P est le point d'abscisse maximale parmi les points déjà vus.

(2) Q est d'abscisse strictement supérieure à celle de de P.

En fait l'hypothèse de position générale n'interdit pas que deux points possèdent la même abscisse (mais c'est interdit pour trois points). En pareil cas, pour que Q éclaire l'arête issue de P, il suffit que l'ordonnée de Q soit *inférieure* à celle de P, en raison de la convention que l'axe des y pointe vers le bas. Autrement dit, P est strictement inférieur à Q selon l'ordre suivant.

```
// p1 < p2 <=> compare(p1,p2) < 0
static int compare(Point p1, Point p2) {
  if (p1.x < p2.x) return -1 ;
  else if (p1.x > p2.x) return 1 ;
  else if (p1.y < p2.y) return 1 ;
  else if (p1.y > p2.y) return -1 ;
  else return 0 ;
}
```

On note que le point ajouté Q fait toujours partie de la nouvelle enveloppe convexe. En fait, il sera le nouveau point distingué lors de la prochaine extension, comme exprimé par la méthode complète de calcul de l'enveloppe convexe.

```
static Poly convexHullSorted(PointReader in) {
  Point pt1 = in.read() ;
  Point pt2 = in.read() ;
  if (pt1 == null || pt2 == null || compare(pt1,pt2) >= 0)
    throw new Error ("Deux premiers point incorrects") ;
```

```
  Poly p1 = new Poly(pt1, null, null) ;
  Poly p2 = new Poly(pt2, p1, null) ;
  p1.next = p2 ;
  p1.prev = p2 ;
  p2.prev = p1 ;

  Poly e = p2 ;
  for ( ; ; ) {
    Point q = in.read() ;
    if (q == null) return e ;
    if (compare(e.val, q) >= 0) throw new Error ("Points non triés") ;
    e = extend2(e, q) ;
  }
}
```

On peut donc assurer la correction de l'algorithme complet en triant les points de l'entrée selon les abscisses croissantes, puis les ordonnées décroissantes en cas d'égalité. Évidemment pour pouvoir trier les points de l'entrée il faut la lire en entier, l'algorithme n'est donc plus incrémental.

Estimons maintenant le coût du nouvel algorithme de calcul de l'enveloppe convexe de n points. On peut trier les n points en $O(n \log n)$. Nous prétendons ensuite que le calcul proprement dit de l'enveloppe est en $O(n)$. En effet, le coût d'un appel à `extend2` est en $O(j - i)$ où $j - i - 1$ est le nombre de sommets supprimés de l'enveloppe à cette étape. Or un sommet est supprimé au plus une fois, ce qui conduit à une somme des coûts des appels à `extend2` en $O(n)$. Le tri domine et l'algorithme complet est donc en $O(n \log n)$. Cet exemple illustre que le choix d'une structure de données appropriée (ici la liste doublement chaînée) est parfois crucial pour atteindre la complexité théorique d'un algorithme. Ici, avec une liste chaînée seulement selon les champs `next`, il n'est pas possible de trouver le sommet E_i à partir d'un sommet de la face éclairée sans faire le tour par la face sombre.

Chapitre II

Piles et files

Les piles et les files sont des exemples de structures de données que faute de mieux, nous appellerons des *sacs*. Un sac offre trois opérations élémentaires :

(1) tester si le sac est vide,

(2) ajouter un élément dans le sac,

(3) retirer un élément du sac (tenter de retirer un élément d'un sac vide déclenche une erreur).

Le sac est une structure impérative : un sac se modifie au cours du temps. En Java, il est logique de représenter un sac (d'éléments E) par un objet (d'une classe **Bag**$\langle E\rangle$) qui possède trois méthodes.

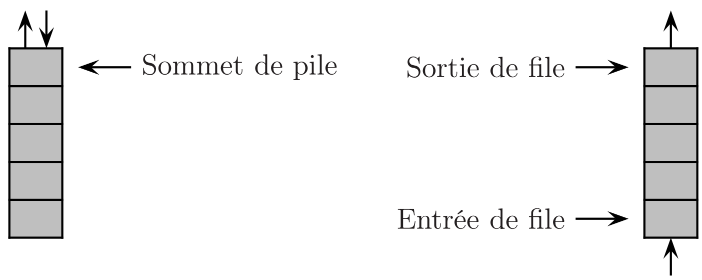
```
class Bag<E> {
  Bag<E> { ... } // Construire un sac vide
  boolean isEmpty() { ... }
  void add(E e) { ... }
  E remove() { ... }
}
```

Ainsi on ajoute par exemple un élément e dans le sac b par b`.add(`e`)`, ce qui modifie l'état interne du sac.

Piles et files se distinguent par la relation entre éléments ajoutés et éléments retirés. Dans le cas des piles, c'est le dernier élément ajouté qui est retiré. Dans le cas d'une file c'est le premier élément ajouté qui est retiré. On dit que la pile est une structure LIFO (*last-in first-out*), et que la file est une structure FIFO (*first-in first-out*).

Si on représente pile et file par des tas de cases, on voit que la pile possède un sommet, où sont ajoutés et d'où sont retirés les éléments, tandis que la file possède une entrée et une sortie. Tout ceci est parfaitement conforme à l'intuition d'une pile d'assiettes, ou de la queue devant un

FIG. 1 – Une pile (ajouter et retirer du même côté), une file (ajouter et retirer de côtés opposés).



guichet, et suggère fortement des techniques d'implémentations pour les piles et les files.

Exercice 1 Soit la méthode `build` suivante qui ajoute tous les éléments d'une liste dans un sac de **int** puis construit une liste en vidant le sac.

```
static List build(List p) {
  Bag bag = new Bag () ;
  for ( ; p != null ; p = p.next)
    bag.add(p.val) ;
  List r = null ;
  while (!bag.isEmpty())
    r = new List (bag.remove(), r) ;
  return r ;
}
```

Quelle est la liste renvoyée dans le cas où le sac est une pile, puis une file ?

Solution. Si le sac est une pile, alors `build` renvoie une copie de la liste `p`. Si le sac est une file, alors `build` renvoie une copie en ordre inverse de la liste `p`. □

Il y a un peu de vocabulaire spécifique aux piles et aux files. Traditionnellement, ajouter un élément dans une pile se dit *empiler* (*push*), et l'enlever *dépiler* (*pop*), tandis qu'ajouter un élément dans une file se dit *enfiler*, et l'enlever *défiler*.

1 À quoi ça sert ?

La file est peut-être la structure la plus immédiate, elle modélise directement les files d'attente gérées selon la politique premier-arrivé, premier-servi. La pile est plus informatique par nature. Dans la vie, la politique dernier-arrivé, premier-servi n'est pas très populaire. Elle correspond pourtant à une situation courante, la survenue de tâches de plus en plus urgentes. Les piles modélisent aussi tout système où l'entrée et la sortie se font par le même passage obligé : wagons sur une voie de garage, cabine de téléphérique, etc.

1.1 Premier arrivé, premier servi

Utiliser une file informatique est donc naturel lorsque l'on modélise une file d'attente au sens courant du terme. Considérons un exemple simple :

- Un unique guichet ouvert 8h00 consécutives (soit $8 \times 60 \times 60$ secondes).
- Les clients arrivent et font la queue. La probabilité d'arrivée d'un nouveau client dans l'intervalle $[t \ldots t+1[$ est p.
- Le temps que prend le service d'un client suit une loi uniforme sur $[30 \ldots 300[$.
- Les clients, s'il attendent trop longtemps, partent sans être servis. Leur patience est une loi uniforme sur $[120 \ldots 1800[$

Notre objectif est d'écrire une simulation informatique afin d'estimer le rapport clients repartis sans être servis sur nombre total de clients, en fonction de la probabilité p.

Nous implémentons la simulation par une classe **Simul**. Pour réaliser les tirages aléatoires nécessaires, nous employons une source pseudo-aléatoire, les objets de la classe **Random** du package `java.util` (voir B.1.4 pour la notion de package et B.6.3.1 pour une description de **Random**).

```
// Source pseudo-aléatoire
static private Random rand = new Random () ;

// Loi uniforme sur [min...max[
static private int uniform(int min, int max) {
  return min + rand.nextInt(max-min) ;
}
```

```java
  // Survenue d'un événement de probabilité prob ∈ [0...1]
  static private boolean occurs(double prob) {
    return rand.nextDouble() < prob ;
  }
```

Un client en attente est modélisé par un objet de la classe **Client** . Un objet **Client** est créé lors de l'arrivée du client au temps t, ce qui donne l'occasion de calculer l'heure à laquelle le client exaspéré repartira s'il il n'a pas été servi.

```java
class Client {
  // Temps d'attente
  final static int waitMin = 120,  waitMax = 1800 ;

  int arrival, departure ;

  Client (int t) {
    arrival = t ; departure = t+Simul.uniform(waitMin, waitMax) ;
  }
}
```

Noter que les constantes qui gouvernent le temps d'attente sont des variables **final** de la classe **Client** .

L'implémentation d'une file sera décrite plus tard. Pour l'instant une file (de clients) est un objet de la classe **Fifo** qui possède les méthodes `isEmpty`, `add` et `remove` décrites pour les **Bag** de l'introduction.

La simulation est discrète. Durant la simulation, une variable `tFree` indique la date où le guichet sera libre. À chaque seconde `t`, il faut :

- Faire un tirage aléatoire pour savoir si un nouveau client arrive.
  - Si oui, ajouter le nouveau client (avec sa limite de temps) dans la file d'attente.
- Quand un client est disponible et que le guichet est libre (si `t` est supérieur à `tFree`).
  - Extraire le client de la file.
  - Vérifier qu'il est toujours là (sinon, passer au « client » suivant).
  - Tirer aléatoirement un temps de traitement.
  - Et ajuster `tFree` en conséquence.

Le code qui réalise cette simulation naïve en donné par la figure 2. La méthode `simul` prend la probabilité d'arrivée d'un client en argument et renvoie le rapport clients non-servis sur nombre total de clients. Le code suit les grandes lignes de la description précédente. Il faut surtout noter comment on résout le problème d'une boucle **while** qui doit d'une part s'effectuer tant que la file n'est pas vide et d'autre part cesser dès qu'un client effectivement présent est extrait de la file (sortie par **break**). On note aussi la conversion de type explicite `((double)nbFedUp)/nbClients`, nécessaire pour forcer la division flottante. Sans cette conversion on aurait la division euclidienne. La figure 3 donne les résultats de la simulation pour diverses valeurs de la probabilité `prob`. Il s'agit de moyennes sur dix essais.

1.2 Utilisation d'une pile

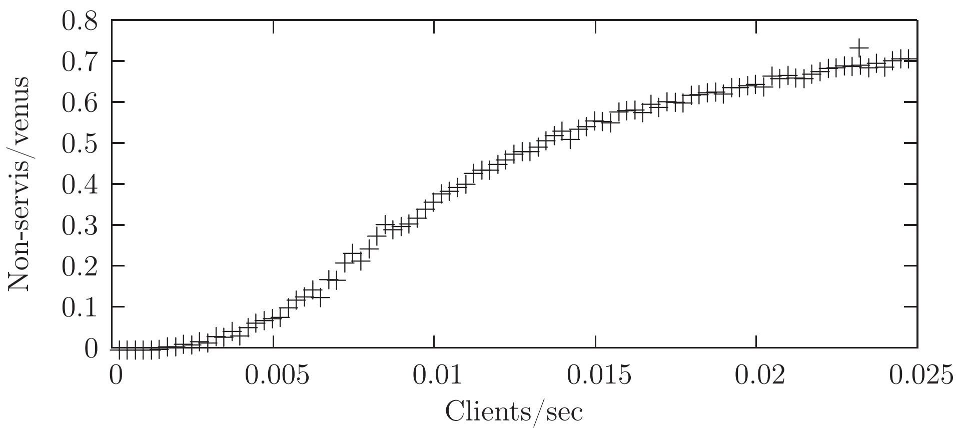
Nous allons prendre en compte la nature très informatique de la pile et proposer un exemple plutôt informatique. Les calculatrices d'une célèbre marque états-unienne ont popularisé une notation des calculs dite parfois « polonaise inverse »[1] ou plus généralement postfixe. Les calculs se font à l'aide d'une pile, les nombres sont simplement empilés, et l'exécution d'une opération

[1] En hommage au philosophe et mathématicien Jan Łukasiewicz qui a introduit cette notation dans les années 20.

Fig. 2 – Code de la simulation.

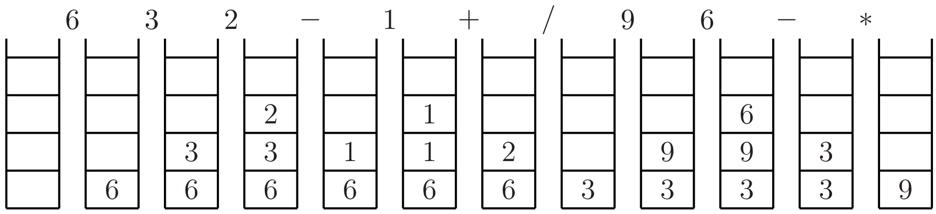
```
static double simul(double probNewClient) {
  Fifo f = new Fifo () ;
  int nbClients = 0, nbFedUp = 0 ;
  int tFree = 0 ; // Le guichet est libre au temps tFree

  for (int t = 0 ; t < tMax ; t++) {
    // Tirer l'arrivée d'un client dans [t..t+1[
    if (occurs(probNewClient)) {
      nbClients++ ;
      f.add(new Client(t)) ;
    }
    if (tFree <= t) {
      // Le guichet est libre, servir un client (si il y en a encore un)
      while (!f.isEmpty()) {
        Client c = f.remove() ;
        if (c.departure >= t) { // Le client est encore là
          tFree = t + uniform(serviceMin, serviceMax) ;
          break ; // Sortir de la boucle while
        } else { // Le client était parti
          nbFedUp++ ;
        }
      }
    }
  }
  return ((double)nbFedUp)/nbClients ;
}
```

Fig. 3 – Résultat de la simulation de la file d'attente



`op` revient à d'abord dépiler un premier opérateur e_1, puis un second e_2, et enfin à empiler le résultat e_2 `op` e_1. Par exemple, la figure 4 donne les étapes successives du calcul `6 3 2 - 1 + / 9 6 - *` exprimé en notation postfixe. Le fabricant états-unien de calculatrices affirme qu'avec

FIG. 4 – Calcul de l'expression postfixe `6 3 2 - 1 + / 9 6 - *`



un peu d'habitude la notation postfixe est plus commode que la notation usuelle (dite *infixe*). C'est peut-être vrai, mais on peut en tout cas être sûr que l'interprétation de la notation postfixe par une machine est bien plus facile à réaliser que celle de la notation infixe. En voici pour preuve le programme **Calc** qui réalise le calcul postfixe donné sur la ligne de commande. Dans ce code les objets de la classe **Stack** sont des piles d'entiers.

```
class Calc {
  public static void main (String [] arg) {
    Stack stack = new Stack () ;
    for (int k = 0 ; k < arg.length ; k++) {
      String x = arg[k] ;
      if (x.equals("+")) {
        int i1 = stack.pop(), i2 = stack.pop() ;
        stack.push(i2+i1) ;
      } else if (x.equals("-")) {
        int i1 = stack.pop(), i2 = stack.pop() ;
        stack.push(i2-i1) ;
      ...
      /* Idem pour "*" et "/" */
      ...
      } else {
        stack.push(Integer.parseInt(x)) ;
      }
      System.err.println(x + " -> " + stack) ;
    }
    System.out.println(stack.pop()) ;
  }
}
```

Le source contient des messages de diagnostic (sur `System.err` on n'insistera jamais assez), un essai[2] nous donne (le sommet de pile est à droite) :

```
% java Calc 6 3 2 - 1 + / 9 6 - '*'
6 -> [6]
3 -> [6, 3]
  ⋮
6 -> [3, 9, 6]
```

[2] Il faut citer le symbole `*` par exemple par `'*'`, afin d'éviter son interprétation par le shell, voir VI.3.1.

```
- -> [3, 3]
* -> [9]
9
```

La facilité d'interprétation de l'expression postfixe provient de ce que sa définition est très opérationnelle, elle dit exactement quoi faire. Dans le cas de la notation infixe, il faut régler le problème des priorités des opérateurs. Par exemple si on veut effectuer le calcul infixe `1 + 2 * 3` il faut procéder à la multiplication d'abord (seconde opération), puis à l'addition (première opération) ; tandis que pour calculer `1 * 2 + 3`, on effectue d'abord la première opération (multiplication) puis la seconde (addition). Par contraste, la notation postfixe oblige l'utilisateur de la calculatrice à donner l'ordre désiré des opérations (comme par exemple `1 2 3 * +` et `1 2 * 3 +`) et donc simplifie d'autant le travail de la calculatrice. Dans le même ordre d'idée, la notation postfixe rend les parenthèses inutiles (ou empêche de les utiliser selon le point de vue).

Exercice 2 Écrire un programme **Infix** qui lit une expression sous forme postfixe et affiche l'expression infixe (complètement parenthésée) qui correspond au calcul effectué par le programme **Calc**. On supposera donnée une classe **Stack** des piles de chaînes.

Solution. Il suffit tout simplement de construire la représentation infixe au lieu de calculer.

```
class Infix {
  public static void main (String [] arg) {
    Stack stack = new Stack () ;
    for (int k = 0 ; k < arg.length ; k++) {
      String x = arg[k] ;
      if (x.equals("+") || x.equals("-") || x.equals("*") || x.equals("/")) {
        String i1 = stack.pop(), i2 = stack.pop() ;
        stack.push("(" + i2 + x + i1 + ")") ;
      } else {
        stack.push(x) ;
      }
      System.err.println(x + " -> " + stack) ;
    }
    System.out.println(stack.pop()) ;
  }
}
```

L'emploi systématiques des parenthèses permet de produire des expressions infixes justes. Sans les parenthèses `1 2 3 - -` et `1 2 - 3 -` conduiraient au même résultat `1 - 2 - 3` qui serait incorrect pour `1 2 3 - -`. Sur l'expression déjà donnée, on obtient :

```
% java Infix 6 3 2 - 1 + / 9 6 - '*'
6 -> [6]
3 -> [6, 3]
  ⋮
6 -> [(6/((3-2)+1)), 9, 6]
- -> [(6/((3-2)+1)), (9-6)]
* -> [((6/((3-2)+1))*(9-6))]
((6/((3-2)+1))*(9-6))
```

Nous verrons plus tard que se passer des parenthèses inutiles ne se fait simplement qu'à l'aide d'une nouvelle notion. □

2 Implémentation des piles

Nous donnons d'abord deux techniques d'implémentation possibles, avec un tableau et avec une liste simplement chaînée. Ensuite, nous montrons comment utiliser la structure de pile toute faite de la bibliothèque.

2.1 Une pile dans un tableau

C'est la technique conceptuellement la plus simple, directement inspirée par le dessin de gauche de la figure 1.

Plus précisément, une pile (d'entiers) contient un tableau d'entiers, dont nous fixons la taille arbitrairement à une valeur constante. Seul le début du tableau (un segment initial) est utilisé, et un indice variable `sp` indique le *pointeur* de la pile. Le pointeur de pile indique la prochaine position libre dans la pile, mais aussi le nombre d'éléments présents dans la pile.

0	1	2	3	4	5	6	7	8	9		sp
6	2	7	0	2	8	0	2	3	4		5

Dans ce schéma, seules les 5 premières valeurs du tableau sont significatives.

```
class Stack {
  final static int SIZE = 10; // assez grand ?

  private int sp ;
  private int [] t ;

  Stack () { // Construire une pile vide
    t = new int[SIZE] ; sp = 0;
  }
  ⋮
}
```

Notons que tous les champs des objets `Stack` sont privés (**private**, voir B.1.4). La taille des piles est donnée par une constante (**final**), statique puisque qu'il n'y a aucune raison de créer de nombreux exemplaires de cette constante.

À la création, une pile contient donc un tableau de `SIZE` = 10 entiers. La valeur initiale contenue dans les cases du tableau est zéro (voir B.3.6.2), mais cela n'a pas d'importance ici, car aucune case n'est valide (`sp` est nul).

0	1	2	3	4	5	6	7	8	9		sp
0	0	0	0	0	0	0	0	0	0		0

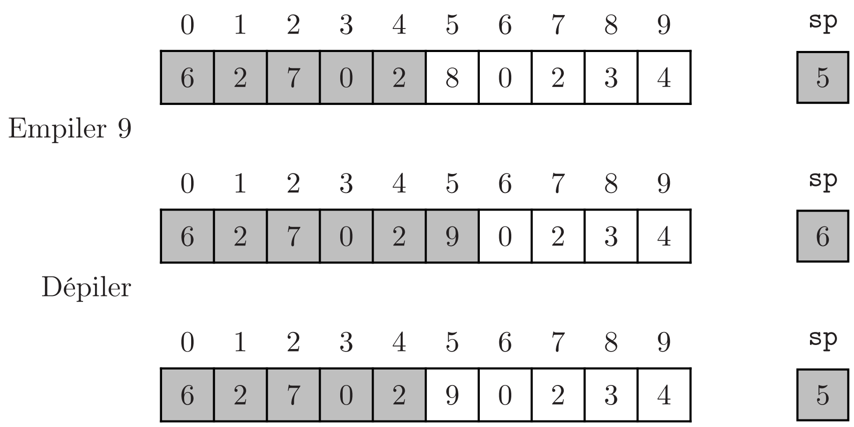
Il nous reste à coder les trois méthodes des objets de la classe `Stack` pour respectivement, tester si la pile est vide, empiler et dépiler.

```
  boolean isEmpty() { return sp == 0 ; }

  void push(int x) {
    if (sp >= SIZE) throw new Error ("Push : pile pleine") ;
    t[sp++] = x ; // Ou bien : t[sp] = x ; sp = sp+1 ;
  }
```

```
  int pop() {
    if (isEmpty()) throw new Error ("Pop : pile vide") ;
    return t[--sp] ; // Ou bien : sp = sp-1 ; return t[sp] ;
  }
```

La figure 5 donne un exemple d'évolution du tableau `t` et du pointeur de pile `sp`. Le code ci-

FIG. 5 – Empiler 9, puis dépiler.



dessus signale les erreurs en lançant l'exception **Error** (voir B.4.2), et emploie les opérateurs de post-incrément et pré-décrément (voir B.3.4). Notons surtout que toutes opérations sont en temps constant, indépendant du nombre d'éléments contenus dans la pile, ce qui semble attendu dans le cas d'opérations aussi simples que `push` et `pop`.

Les deux erreurs possibles *"Push : pile pleine"* et *"Pop : pile vide"* sont de natures bien différentes. En effet, la première découle d'une contrainte technique (la pile est insuffisamment dimensionnée), tandis que la seconde découle d'une erreur de programmation de l'utilisateur de la pile. Nous allons voir ce que nous pouvons faire au sujet de ces deux erreurs.

Commençons par l'erreur *"pile pleine"*. Une première technique simple est de passer le bébé à l'utilisateur, en lui permettant de dimensionner la pile lui-même.

```
class Stack {
  ...
  Stack (int sz) { t = new int[sz] ; sp = 0 ; }

  void push(int x) {
    if (sp >= t.length) throw new Error ("Push : pile pleine") ;
    t[sp++] = x ;
  }
  ...
}
```

Ainsi, en cas de pile pleine, on peut toujours compter sur l'utilisateur pour recommencer avec une pile plus grande. Cela paraît abusif, mais c'est à peu près ce qui se passe au sujet de la pile des appels de méthode.

Mais en fait, les tableaux de Java sont suffisamment flexibles pour autoriser le redimensionnement automatique de la pile. Il suffit d'allouer un nouveau tableau plus grand, de recopier les éléments de l'ancien tableau dans le nouveau, puis de remplacer l'ancien tableau par le nouveau (et dans cet ordre, c'est plus simple).

```
  private void resize() {
    int [] newT = new int [2 * this.t.length] ; // Allouer le nouveau tableau
    for (int k = 0 ; k < sp ; k++)            // Recopier
      newT[k] = this.t[k] ;
    this.t = newT ;                           // Remplacer
  }

  void push(int x) {
    if (sp >= t.length) resize() ;
    t[sp++] = x ;
  }
```

Le redimensionnement automatique offre une flexibilité maximale. En contrepartie, il a un coût : un appel à `push` ne s'exécute plus en temps garanti constant, puisqu'il peut arriver qu'un appel à `push` entraîne un redimensionnement dont le coût est manifestement proportionnel à la taille de la pile.

Mais vu globalement sur une exécution complète, le coût de N appels à `push` reste de l'ordre de N. On dira alors que le coût *amorti* de `push` est constant. En effet, considérons N opérations `push`. Au pire, il n'y a pas de `pop` et la pile contient finalement N éléments pour une taille 2^P du tableau interne, où 2^P est la plus petite puissance de 2 supérieure à N. En outre, supposons que le tableau est redimensionné P fois (taille initiale 2^0). L'ordre de grandeur du coût cumulé de tous les `push` est donc de l'ordre de N (coût constant de N `push`) plus $\sum_{k=1}^{k=P} 2^k = 2^{P+1} - 1$ (coût des P redimensionnements). Soit un coût final de l'ordre de N, puisque $2^P < 2 \cdot N$. Pour ce qui est de la mémoire, on alloue au total de l'ordre de 2^{P+1} cellules de mémoire, dont la moitié ont pu être récupérées par le garbage collector. Notons finalement que le coût amorti constant est assuré quand les tailles des tableaux successifs suivent une progression géométrique, dont la raison n'est pas forcément 2. En revanche, le coût amorti devient linéaire pour des tableaux les tailles suivent une progression arithmétique. Pour cette raison, écrire `int [] newT = new int [t.length + K]` est rarement une bonne idée.

Abordons maintenant l'erreur `"pile vide"`. Ici, il n'y a aucun moyen de supprimer l'erreur, car rien n'empêchera jamais un programmeur maladroit ou fatigué de dépiler une pile vide. Mais on peut signaler l'erreur plus proprement, ce qui donne la possibilité au programmeur fautif de la réparer. Pour ce faire, il convient de lancer, non plus l'exception **Error**, mais une exception spécifique. Les exceptions sont complètement décrites dans l'annexe Java en B.4. Ici, nous nous contentons d'une présentation minimale. La nouvelle exception **StackEmpty** se définit ainsi.

```
class StackEmpty extends Exception { }
```

C'est en fait une déclaration de classe ordinaire, car une exception est un objet d'une classe un peu spéciale (voir B.4.2). La nouvelle exception se lance ainsi.

```
  int pop () throws StackEmpty {
    if (isEmpty()) throw new StackEmpty () ;
    return t[--sp] ;
  }
```

On note surtout que la méthode `pop` déclare (par **throws**, noter le « s ») qu'elle peut lancer l'exception **StackEmpty**. Cette déclaration est ici obligatoire, parce que **StackEmpty** est une **Exception** et non plus une **Error**. La présence de la déclaration va obliger l'utilisateur des piles à s'occuper de l'erreur possible.

Pour illustrer ce point revenons sur l'exemple de la calculatrice **Calc** (voir 1.2). La calculatrice emploie une pile `stack`, elle dépile et empile directement par `stack.pop()` et `stack.push(...)`. Nous supposons que la classe **Stack** est telle que ci-dessus (avec une méthode `pop` qui déclare lancer **StackEmpty**) et que nous ne pouvons pas modifier son code. Pour organiser un peu la suite, nous ajoutons deux nouvelles méthodes statiques `push` et `pop` à la classe Calc, et transformons les appels de méthode dynamiques en appels statiques.

```
class Calc {
  static int pop(Stack stack) { return stack.pop() ; }

  static void push(Stack stack, int x) { stack.push(x) ; }

  public static void main (String [] arg) {
    ...
    if (x.equals("+")) {
      int i1 = pop(stack), i2 = pop(stack) ;
      push(stack, i2+i1) ;
    ...
  }
}
```

Le compilateur refuse le programme ainsi modifié. Il exige le traitement de StackEmpty qui peut être lancée par `stack.pop()`. Traiter l'exception peut se faire en *attrapant* l'exception, par l'instruction **try**.

```
  static int pop(Stack stack) {
    try {
      return stack.pop() ;
    } catch (StackEmpty e) {
      return 0 ;
    }
  }
```

L'effet est ici que si l'instruction **return** `stack.pop()` échoue parce que l'exception StackEmpty est lancée, c'est alors l'instruction **return** `0` qui s'exécute. Cela revient à remplacer la valeur « exceptionnelle » StackEmpty par la valeur plus normale `0` (voir B.4.1 pour les détails). Par conséquent, une pile vide se comporte comme si elle contenait une infinité de zéros.

On peut aussi décider de ne pas traiter l'exception, mais alors il faut signaler que les méthodes `Calc.pop` puis `main` peuvent lancer l'exception StackEmpty, même si c'est indirectement.

```
class Calc {
  static int pop(Stack stack) throws StackEmpty { return stack.pop() ; }

  public static void main (String [] arg) throws StackEmpty {
    ...
  }
}
```

Si l'exception survient, elle remontera de méthode en méthode et fera finalement échouer le programme. En fait, comme les déclarations **throws**, ici obligatoires, sont quand même pénibles, on a plutôt tendance à faire échouer le programme explicitement dès que l'exception atteint le code que l'on contrôle.

```
  static int pop(Stack stack) {
    try {
      return stack.pop() ;
    } catch (StackEmpty e) {
      System.err.println("Adieu : pop sur pile vide") ;
      System.exit(2) ; // Arrêter le programme immédiatement (voir B.6.1.5)
      return 0 ; // Le compilateur exige ce return jamais exécuté
    }
  }
```

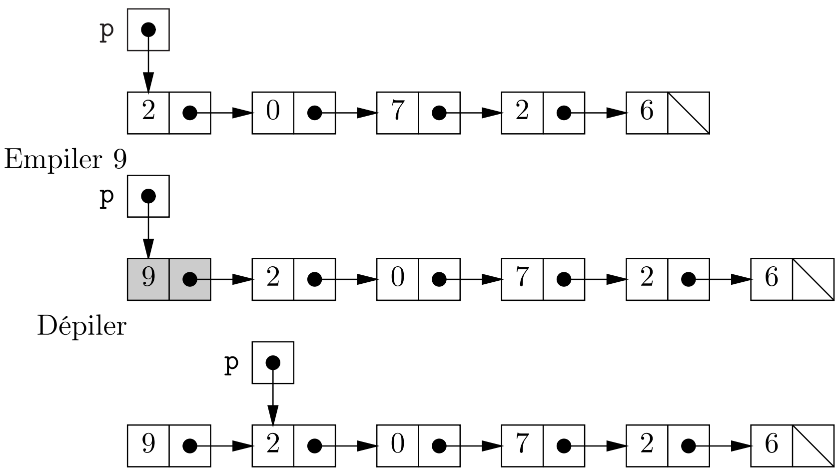
Bien sûr, si on veut que dépiler une pile vide provoque un arrêt du programme, il aurait été plus court de lancer **new Error** `("Pop : pile vide")` dans la méthode `pop` des objets **Stack**. Mais nous nous sommes interdit de modifier la classe **Stack**.

2.2 Une pile dans une liste

C'est particulièrement simple. En effet, les éléments sont empilés et dépilés du même côte de la pile, qui peut être le début d'une liste. On suppose donc donnée une classe des listes (d'entiers). Et on écrit :

```
class Stack {
  private List p ;

  Stack () { p = null ; }

  boolean isEmpty() { return p == null ; }

  void push(int x) { p = new List (x, p) ; }

  int pop() throws StackEmpty {
    if (p == null) throw new StackEmpty () ;
    int r = p.val ;
    p = p.next ;
    return p ;
  }
}
```

Voici un exemple d'évolution de la liste `p`.



2.3 Les piles de la bibliothèque

Il existe déjà une classe des piles dans la bibliothèque, la classe **Stack** du package `java.util`, implémentée selon la technique des tableaux redimensionnés. Mais une classe des piles de quoi ?

En effet, dans les deux sections précédentes, nous n'avons codé que des piles de **int** en nous disant que pour coder une pile, par exemple de **String**, il nous suffisait de remplacer **int** par **String** partout dans le code. Il est clair qu'une telle « solution » ne convient pas à une classe de bibliothèque qui est compilée par le fabricant. Après bien des hésitations, Java a résolu ce problème en proposant des classes *génériques*, ou *paramétrées*. La classe **Stack** est en fait une classe **Stack**<E>, où E est n'importe quelle classe, et les objets de la classe **Stack**<E> sont des piles d'objets de la classe E. Formellement, **Stack** n'est donc pas exactement une classe, mais plutôt une fonction des classes dans les classes. Informellement, on peut considérer que nous disposons d'une infinité de classes **Stack<String>**, **Stack< List >** etc. Par exemple on code facilement la pile de chaînes de l'exercice 2 comme un objet de la classe `java.util.Stack<String>`.

```
import java.util.* ;
class Infix {
  public static void main (String [] arg) {
    Stack<String> stack = new Stack<String> () ;
    ...
        String i1 = stack.pop(), i2 = stack.pop() ;
        stack.push("(" + i2 + x + i1 + ")") ;
    ...
  }
}
```

(Pour **import**, voir B.3.5.) Cela fonctionne parce que la classe **Stack<**E**>** possède bien des méthodes `pop` et `push`, où par exemple `pop()` renvoie un objet E, ici un **String**.

Comme le paramètre E dans **Stack<**E**>** est une classe, il n'est pas possible de fabriquer des piles de **int**. Plus généralement, il est impossible de fabriquer des piles de scalaires (voir B.3.1 pour la définition des scalaires). Mais la bibliothèque fournit une classe associée par type scalaire, par exemple **Integer** pour **int**. Un objet **Integer** n'est guère plus qu'un objet dont une variable d'instance (privée) contient un **int** (voir B.6.1.1). Il existe deux méthodes pour convertir un scalaire **int** en un objet **Integer** et réciproquement, `valueOf` (logiquement statique) et `intValue` (logiquement dynamique). De sorte qu'écrire la calculatrice **Calc** (voir 1.2) avec une pile de la bibliothèque semble assez lourd au premier abord.

```
import java.util.* ;
class Calc {
  public static void main (String [] arg) {
    Stack<Integer> stack = new Stack<Integer> () ;
    ...
        int i1 = stack.pop().intValue(), i2 = stack.pop().intValue() ;
        stack.push(Integer.valueOf(i2+i1)) ;
    ...
  }
}
```

Mais en fait, le compilateur Java sait insérer les appels aux méthodes de conversion automatiquement, c'est-à-dire que l'on peut écrire plus directement.

```
import java.util.* ;
class Calc {
  public static void main (String [] arg) {
    Stack<Integer> stack = new Stack<Integer> () ;
    ...
        int i1 = stack.pop(), i2 = stack.pop() ;
        stack.push(i2+i1) ;
    ...
  }
}
```

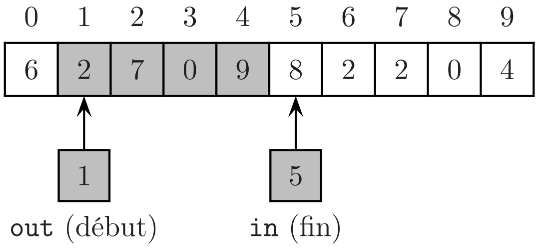
Tout semble donc se passer presque comme si il y avait une pile de **int** ; mais attention, il s'agit bien en fait d'une pile de **Integer**. Tout se passe plutôt comme si le compilateur traduisait le programme avec les **int** vers le programme avec les **Integer**, et c'est bien ce dernier qui s'exécute, avec les conséquences prévisibles sur la performance en général et la consommation mémoire en particulier.

3 Implémentation des files

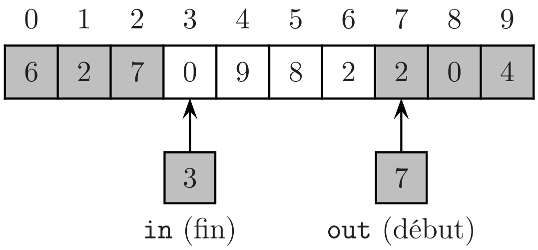
Comme pour les piles, nous présentons trois techniques, tableaux, listes et bibliothèque.

3.1 Une file dans un tableau

L'idée, conceptuellement simple mais un peu délicate à programmer, est de gérer deux indices : `in` qui marque la position où ajouter le prochain élément et `out` qui marque la position d'où proviendra le prochain élément enlevé. L'indice `out` marque le début de la file et `in` sa fin. Autrement dit, le contenu de la file va la case d'indice `out` à la case qui précède la case d'indice `in`.



Dans le schéma ci-dessus, les cases valides sont grisées, de sorte que la file ci-dessus contient les entiers $2, 7, 0, 9$ (dans l'ordre, 2 est le premier entré et 9 le dernier entré). Au cours de la vie de la file, les deux indices sont croissants. Plus précisément, on incrémente `in` après ajout et on incrémente `out` après suppression. Lorsqu'un indice atteint la fin du tableau, il fait tout simplement le tour du tableau et repart à zéro. Il en résulte que l'on peut avoir `out` < `in`. Par exemple, voici une autre file contenant cette fois $2, 0, 4, 6, 2, 7$.



Le tableau d'une file est un tableau *circulaire*, que l'on parcourt en incrémentant un indice modulo n, où n est la taille du tableau. Par exemple pour parcourir le contenu de la file, on parcourt les indice de `out` à $(\texttt{in} - 1) \mod n$. Soit un parcours de 1 à 4 dans le premier exemple et de 7 à 2 dans le second.

Une dernière difficulté est de distinguer entre file vide et file pleine. Comme le montre le schéma suivant, les deux indices `out` et `in` n'y suffisent pas.

Ici, `out` (début) et `in` (fin) valent tous deux 4. Pour connaître le contenu de la file il faut donc parcourir le tableau de l'indice 4 à l'indice 3. Une première interprétation du parcours donne une file vide, et une seconde une file pleine. On résout facilement la difficulté par une variable `nb` supplémentaire, qui contient le nombre d'éléments contenus dans la file. Une autre solution aurait été de décréter qu'une file qui contient $n - 1$ éléments est pleine. On teste en effet cette dernière condition sans ambiguïté par $\texttt{in} + 1$ congru à `out` modulo n. Nous choisissons la solution de la variable `nb`, car connaître simplement le nombre d'éléments de la file est pratique, notamment pour gérer le redimensionnement.

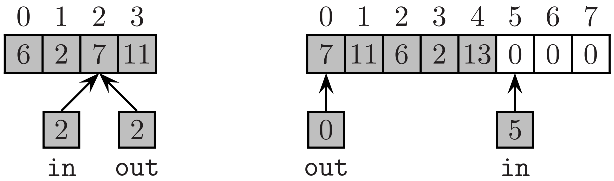
Fig. 6 – Implémentation d'une file dans un tableau

```
class FifoEmpty extends Exception { }

class Fifo {
  final static int SIZE=10 ;
  private int in, out, nb ;
  private int [] t ;

  Fifo () { t = new int[SIZE] ; in = out = nb = 0 ; }

  /* Increment modulo la taille du tableau t, utilisé partout */
  private int incr(int i) { return (i+1) % t.length ; }

  boolean isEmpty() { return nb == 0 ; }

  int remove() throws FifoEmpty {
    if (isEmpty()) throw new FifoEmpty () ;
    int r = t[out] ;
    out = incr(out) ; nb-- ; // Effectivement enlever
    return r ;
  }

  void add(int x) {
    if (nb+1 >= t.length) resize() ;
    t[in] = x ;
    in = incr(in) ; nb++ ; // Effectivement ajouter
  }

  private void resize() {
    int [] newT = new int[2*t.length] ; // Allouer
    int i = out ; // indice du parcours de t
    for (int k = 0 ; k < nb ; k++) {    // Copier
      newT[k] = t[i] ;
      i = incr(i) ;
    }
    t = newT ; out = 0 ; in = nb ;     // Remplacer
  }

/* Méthode toString, donne un exemple de parcours de la file */
  public String toString() {
    StringBuilder b = new StringBuilder () ;
    b.append("[") ;
    if (nb > 0) {
      int i = out ;
      b.append(t[i]) ; i = incr(i) ;
      for ( ; i != in ; i = incr(i))
        b.append(", " + t[i]) ;
    }
    b.append("]") ;
    return b.toString() ;
  }
}
```

La figure 6 donne la classe **Fifo** des files implémentées par un tableau, avec redimensionnement automatique. Un point clé de ce code est la gestion des indices `in` et `out`. Pour ajouter un élément dans la file, `in` est incrémenté (modulo n). Par exemple, voici l'ajout (méthode `add`) de l'entier 11 dans une file.

Pour supprimer un élément, on incrémente `out`. Par exemple, voici la suppression du premier élément d'une file (la méthode `remove` renvoie ici 2).

Le redimensionnement (méthode `resize`) a pour effet de tasser les éléments au début du nouveau tableau. Voici l'exemple de l'ajout de 13 dans une file pleine de taille 4.

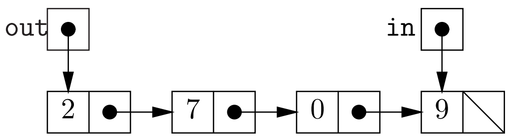


Avant ajout la file contient $7, 11, 6, 2$, après ajout elle contient $7, 11, 6, 2, 13$.

Le code du redimensionnement (méthode `resize`) est un peu simplifié par la présence de la variable `nb`. Pour copier les éléments de `t` dans `newT`, le contrôle est assuré par une simple boucle sur le compte des éléments transférés. Un contrôle sur l'indice `i` dans le tableau `t` serait plus délicat. La méthode `toString` donne un exemple de ce style de parcours du contenu de la file, entre les indices `out` (inclus) et `in` (exclu), qui doit particulariser le cas d'une file vide et effectuer la comparaison à `in` à partir du second élément parcouru (à cause du cas de la file pleine).

3.2 Une file dans une liste

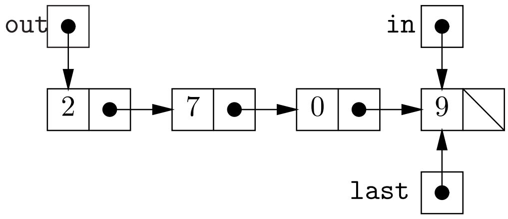
L'implémentation est conceptuellement simple. Les éléments de la file sont dans une liste, on enlève au début de la liste et on ajoute à la fin de la liste. Pour garantir des opérations en temps constant, on utilise une référence sur la dernière cellule de liste. Nous avons déjà largement exploité cette idée, par exemple pour copier une liste itérativement (voir l'exercice I.2). Nous nous donnons donc une référence `out` sur la première cellule de la liste, et une référence `in` sur la dernière cellule.



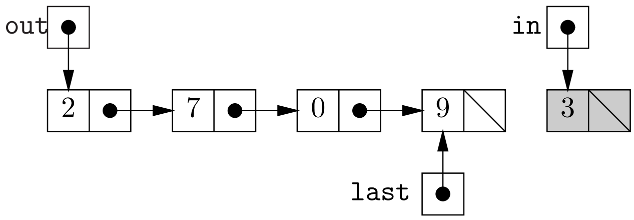
La file ci-dessus contient donc les entiers $2, 7, 0, 9$ dans cette ordre. Une file vide est logiquement identifiée par une variable `out` valant **null**. Par souci de cohérence `in` vaudra alors aussi **null**.

```
class Fifo {
  private List out, in ;

  Fifo () { out = in = null ; }

  boolean isEmpty() { return out == null ; }

  int remove() throws FifoEmpty {
    if (isEmpty()) throw new FifoEmpty () ;
    int r = out.val ;
    out = out.next ;
    if (out == null) in = null ; // La file est vide
    return r ;
  }

  void add(int x) {
    List last = in ;
    in = new List(x, null) ;
    if (last == null) { // La file était vide
      out = in ;
    } else {
      last.next = in ;
    }
  }
}
```

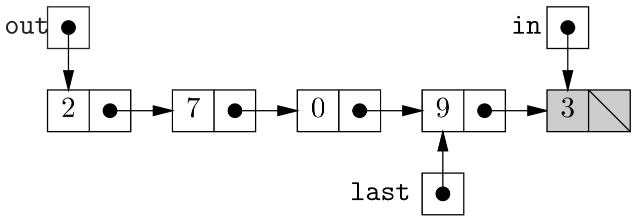
On constate que le code est bien plus simple que dans le cas des tableaux (figure 6). La méthode `remove` est essentiellement la même que la méthode `pop` des piles (voir 2.2), car les deux méthodes réalisent la même opération d'enlever le premier élément d'une liste. La méthode `add` est à peine plus technique, voici par exemple la suite des états mémoire quand on ajoute 3 à la file déjà dessinée. La première instruction `List last = in` garde une copie de la référence `in`.



L'instruction suivante `in = new List (x, null)` alloue la nouvelle cellule de liste.



Et enfin, l'instruction `last.next = in`, ajoute effectivement la nouvelle cellule pointée par `in` à la fin de la liste-file.



3.3 Les files de la bibliothèque

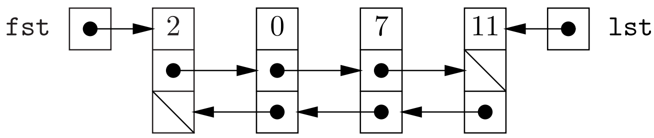
Le package `java.util` de la bibliothèque offre plusieurs classes qui peuvent servir comme une file, même si nous ne respectons pas trop l'esprit dans lequel ces classes sont organisées. (Les files de la bibliothèque **Queue** ne sont pas franchement compatibles avec notre modèle simple des files).

Nous nous focalisons sur la classe **LinkedList**. Comme le nom l'indique à peu près, cette classe est implémentée par une liste chaînée. Elle offre en fait bien plus que la simple fonctionnalité de file, mais elle possède les méthodes `add` et `remove` des files. La classe **LinkedList** est générique, comme la classe **Stack** de la section 2.3, ce qui la rend relativement simple d'emploi.

Exercice 3 Les objets **LinkedList** implémentent en fait les « queues à deux bouts » (*double ended queue*), qui offrent deux couples de méthodes `addFirst`/`removeFirst` et `addLast`/`removeLast` pour ajouter/enlever respectivement au début et à la fin de queue. Afin d'assurer des opérations en temps constant, la classe de bibliothèque repose sur les listes doublement chaînées. Voici une classe **DeList** des cellules de liste doublement chaînée.

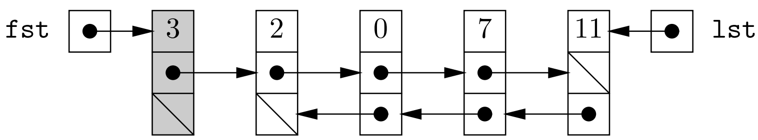
```
class DeList {
  int val ; DeList next, prev ;

  DeList (int val, DeList next, DeList prev) {
    this.val = val ; this.next = next ; this.prev = prev ;
  }
}
```

Le chaînage double permet de parcourir la liste de la première cellule à la dernière en suivant les champs `next`, et de la dernière à la première en suivant le champ `prev`.



Autrement dit, si `p` est une référence vers une cellule de liste doublement chaînée qui n'est pas la première, on a l'égalité `p.prev.next == p`; et si `p` pointe vers la première cellule, on a `p.prev == null`. De même, si `p` pointe vers une cellule qui n'est pas la dernière, on a `p.next.prev == p`; et si `p` pointe vers la dernière cellule, on a `p.next == null`.

Solution. Décomposons une opération, par exemple `addFirst`, en ajoutant 3 au début de la queue déjà dessinée. Dans un premier temps on peut allouer la nouvelle cellule, avec un champ `next` correct (voir `push` en 2.2). Le champ `prev` peut aussi être initialisé immédiatement à **null**.



Reste ensuite à ajuster le champ `prev` de la nouvelle seconde cellule de la liste, cellule accessible par `fst.next` (`(fst.next).prev = fst`).

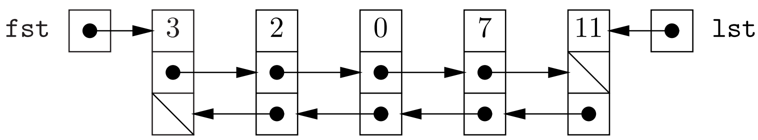
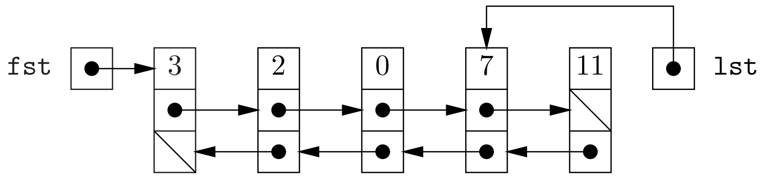


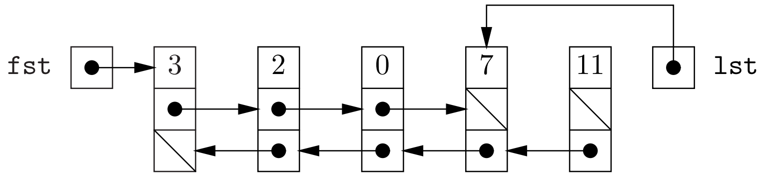
Fig. 7 – Classe des queues à deux bouts

```
class DeQueue {
  private DeList fst, lst ; // First and last cell

  DeQueue () { fst = lst = null ; }

  boolean isEmpty() { return fst == null ; }

  void addFirst(int x) {
    fst = new DeList(x, fst, null) ;
    if (lst == null) {
      lst = fst ;
    } else {
      fst.next.prev = fst ;
    }
  }

  void addLast(int x) {
    lst = new DeList(x, null, lst) ;
    if (fst == null) {
      fst = lst ;
    } else {
      lst.prev.next = lst ;
    }
  }

  int removeFirst() {
    if (fst == null) throw new Error ("removeFirst: empty queue") ;
    int r = fst.val ;
    fst = fst.next ;
    if (fst == null) {
      lst = null ;
    } else {
      fst.prev = null ;
    }
    return r ;
  }

  int removeLast() {
    if (lst == null) throw new Error ("removeLast: empty queue") ;
    int r = lst.val ;
    lst = lst.prev ;
    if (lst == null) {
      fst = null ;
    } else {
      lst.next = null ;
    }
    return r ;
  }
}
```

Pour supprimer par exemple le dernier élément de la queue, il suffit de changer le contenu de `lst` (`lst = lst.prev`).



Reste ensuite à ajuster le champ `next` de la nouvelle dernière cellule (`lst.next = ` **null**).



Le code, pas si difficile est donné par la figure 7. Une fois compris le mécanisme d'ajout et de retrait des cellules, le point délicat est de bien gérer le cas de la queue vide qui correspond à `fst` et `lst` tous deux égaux à **null**. □

4 Type abstrait, choix de l'implémentation

Nous avons présenté trois façons d'implémenter piles et files. Il est naturel de se demander quelle implémentation choisir. Mais nous tenons d'abord à faire remarquer que, du strict point de vue de la fonctionnalité, la question ne se pose pas. Tout ce qui compte est d'avoir une pile (ou une file). En effet, les trois implémentations des piles offrent exactement le même service, de sorte que le choix d'une implémentation particulière n'a aucun impact sur le résultat d'un programme qui utilise une pile. La pile qui est définie exclusivement par les services qu'elle offre est un exemple de type de données *abstrait*.

Java offre des traits qui permettent de fabriquer des types abstraits qui protègent leur implémentation des interventions intempestives. Dans nos deux classes **Stack**, le tableau (ainsi que le pointeur de pile `sp`) et la liste sont des champs privés. Un programme qui utilise nos piles doit donc le faire exclusivement par l'intermédiaire des méthodes `push` et `pop`. Champs privés et méthodes accessibles correspondent directement à la notion de type abstrait défini par les services offerts.

Il faut toutefois remarquer que, du point de vue du langage de programmation, l'existence des deux (trois en fait) méthodes reste pour le moment une convention : nous avons conventionnellement appelé **Stack** la classe des piles et conventionnellement admis qu'elle possède ces trois méthodes :

```
boolean isEmpty() { ... }
void push(int x) { ... }
int pop() throws StackEmpty { ... }
```

Il en résulte par exemple que nous ne pouvons pas encore mélanger les piles en tableau et les piles en liste dans le même programme, à moins de changer le nom des classes ce qui contredirait l'idée d'un type abstrait. Nous verrons au chapitre suivant comment procéder, à l'aide des interfaces.

Répondons maintenant sérieusement à la question du choix de l'implémentation. Dans disons 99 % des cas pour les files et 90 % des cas pour les piles, il faut choisir la classe de bibliothèque, parce que c'est la solution qui demande d'écrire le moins de code. La différence de pourcentage s'explique en comparant le temps de lecture de la documentation au temps d'écriture d'une classe des files ou des piles, et parce que la classe des piles est quand même simple à écrire.

La bibliothèque peut ne pas convenir pour des raisons d'efficacité : le code est trop lent ou trop gourmand en mémoire pour notre programme particulier et nous pouvons faire mieux que lui. Par exemple, le code de bibliothèque entraîne la fabrication d'objets **Integer** en pagaille, et une pile de scalaires se montre au final plus rapide. La classe de bibliothèque peut aussi ne pas convenir parce qu'elle n'offre pas la fonctionnalité inédite dont nous avons absolument besoin (par exemple une inversion complète de la pile), ou plus fréquemment parce que programmer cette fonctionnalité de la façon autorisée par la structuration de la bibliothèque serait trop compliqué, trop coûteux, ou tout simplement impossible avec notre connaissance limitée de Java.

Il reste alors à choisir entre tableaux et listes. Il n'y a pas de réponse toute faite à cette dernière question. En effet, d'une part, la difficulté de l'écriture d'un programme dépend aussi du goût et de l'expérience de chacun ; et d'autre part, l'efficacité respective de l'une ou de l'autre technique dépend de nombreux facteurs, (la pile croît-elle beaucoup, par exemple) et aussi de l'efficacité de la gestion de la mémoire par le système d'exécution de Java. Toutefois, dans le cas où le redimensionnement est inutile, le choix d'une pile-tableau s'impose probablement, puisque simplicité et efficacité concordent. Dans le cas général, on peut tout de même prévoir qu'utiliser les listes demande d'écrire un peu moins de code que d'utiliser les tableaux (redimensionnés). On peut aussi penser que les tableaux seront un plus efficaces que les listes, ou en tout cas utiliseront moins de mémoire au final. En effet, fabriquer une pile-tableau de N éléments demande d'allouer $\log_2 N$ objets (tableaux) contre N objets (cellules de listes) pour une pile-liste. Or, allouer un objet est une opération chère.

Supposons qu'un objet occupe deux cases de mémoire en plus de l'espace nécessaire pour ses données (c'est une hypothèse assez probable pour Java). Une cellule de liste occupe donc 4 cases et un tableau de taille n occupe $3+n$ cases (dont une case pour la longueur). Pour un programme effectuant au total P `push`, la pile-liste aura alloué au total $4 \cdot P$ cases de mémoire, tandis que la pile-tableau aura alloué entre une (si la taille initiale du tableau est 1, et que le programme alterne `push` et `pop`) et environ $4 \cdot P + 3 \cdot \log_2 P$ cases de mémoire (dans le cas où aucun `pop` ne sépare les `push` et où le tableau est redimensionné par le dernier `push`). La pile-tableau n'alloue donc jamais significativement plus de mémoire que la pile-liste, et généralement plutôt moins.

Un autre coût intéressant est l'*empreinte mémoire*, la quantité de mémoire mobilisée à un instant donné. Une pile-file de N éléments mobilise $4 \cdot N$ cases de mémoire. Une pile-tableau mobilise entre $3+N$ et un nombre arbitrairement grand de cases de mémoire (le nombre arbitraire correspond à la taille du tableau lorsque, dans le passé, la pile a atteint sa taille maximale). Un nouvel élément intervient donc dans le choix de l'implémentation : la profondeur maximale de pile au cours de la vie du programme. Si cette profondeur reste raisonnable le choix de la pile-tableau s'impose, autrement le choix est moins net, mais la flexibilité de l'allocation petit-à-petit des cellules de la pile-liste est un avantage. On peut aussi envisager de réduire la taille du tableau interne de la pile tableau, par exemple en réduisant le tableau de la moitié de sa taille quand il n'est plus qu'au quart plein. Mais il faut alors y regarder à deux fois, notre classe des piles commence à devenir compliquée est tout ce code ajouté entraîne un prix en temps d'exécution.

Chapitre III

Associations — Tables de hachage

Ce chapitre aborde un problème très fréquemment rencontré en informatique : la recherche d'information dans un ensemble géré de façon dynamique. Nous nous plaçons dans le cadre où une information complète se retrouve normalement à l'aide d'une clé qui l'identifie. La notion se rencontre dans la vie de tous les jours, un exemple typique est l'annuaire téléphonique : connaître le nom et les prénoms d'un individu suffit normalement pour retrouver son numéro de téléphone. En cas d'homonymie absolue (tout de même rare), on arrive toujours à se débrouiller. Dans les sociétés modernes qui refusent le flou (et dans les ordinateurs) la clé doit identifier un individu unique, d'où, par exemple, l'idée du numéro de sécurité sociale,

Nous voulons un ensemble dynamique d'informations, c'est-à-dire aussi pouvoir ajouter ou supprimer un élément d'information. On en vient naturellement à définir un type abstrait de données, appelé *table d'association* qui offre les opérations suivantes.

- Trouver l'information associé à une clé donnée.
- Ajouter une nouvelle association entre une clé et une information.
- Retirer une clé de la table (avec l'information associée).

La seconde opération mérite d'être détaillée. Lors de l'ajout d'une paire clé-information, nous précisons :

- S'il existe déjà une information associée à la clé dans la table, alors la nouvelle information remplace l'ancienne.
- Sinon, une nouvelle association est ajoutée à la table.

Il en résulte qu'il n'y a jamais dans la table deux informations distinctes associées à la même clé.

Il n'est pas trop difficile d'envisager la possibilité inverse, en ajoutant une nouvelle association à la table dans tous les cas, que la clé s'y trouve déjà ou pas. Il faut alors réviser un peu les deux autres opérations, afin d'identifier l'information concernée parmi les éventuellement multiples qui sont associées à une même clé. En général, on choisit l'information la plus récemment entrée, ce qui revient à un comportement de pile.

Enfin, il peut se faire que la clé fasse partie de l'information, comme c'est généralement le cas dans une base de données. Dans ce cas appelons enregistrement un élément d'information. Un enregistrement est composé de plusieurs champs (nom, prénom, numéro de sécurité sociale, sexe, date de naissance etc.) dont un certain nombre peuvent servir de clé. Il n'y a là aucune difficulté du moins en théorie. Il se peut aussi que l'information se réduise à la clé, dans ce cas la table d'association se réduit à l'ensemble (car il n'y a pas de doublons).

1 Statistique des mots

Nous illustrons l'intérêt de la table d'association par un exemple ludique : un programme **Freq** qui compte le nombre d'occurrences des mots d'un texte, dans le but de produire d'intéressantes statistiques. Soit un texte, par exemple notre hymne national, nous aurons

```
% java Freq marseillaise.txt
nous: 10
vous: 8
français: 7
leur: 7
dans: 6
liberté: 6
  ...
```

1.1 Résolution par une table d'association

Le problème à résoudre se décompose naturellement en trois :

(1) Lire un fichier texte mot à mot.

(2) Compter les occurrences des mots.

(3) Produire un bel affichage, par exemple présenter les mots dans l'ordre décroissant du nombre de leurs occurrences.

Supposons les premier et troisième points résolus. Le troisième par un tri et le premier par une classe WordReader des flux de mots. Les objets de cette classe possèdent une méthode `read` qui renvoie le mot suivant du texte (un **String**) ou **null** à la fin du texte.

Une table d'association permet de résoudre facilement le deuxième point : il suffit d'associer mots et entiers. Un mot qui n'est pas dans la table est conventionnellement associé à zéro. À chaque mot lu m on retrouve l'entier i associé à m, puis on change l'association en m associé à $i + 1$. Nous pourrions spécifier en français une version raffinée de table d'association (pas de suppression, information valant zéro par défaut). Nous préférons le faire directement en Java en définissant une *interface* Assoc.

```
interface Assoc {
  /* Créer/remplacer l'association key → val */
  void put(String key, int val) ;
  /* Trouver l'entier associé à key, ou zéro. */
  int get(String key) ;
}
```

La définition d'interface ressemble à une définition de classe, mais sans le code des méthodes. Cette absence n'empêche nullement d'employer une interface comme un type, nous pouvons donc parfaitement écrire une méthode `count`, qui compte les occurrences des mots du texte dans un Assoc `t` passé en argument, sans que la classe exacte de `t` soit connue[1].

```
  static void count(WordReader in, Assoc t) {
    for (String word = in.read() ; word != null ; word = in.read()) {
      if (word.length() >= 4) {      // Retenir les mots suffisamment longs
        word = word.toLowerCase() ; // Minusculer le mot
        t.put(word, t.get(word)+1) ;
      }
    }
  }
```

(Voir B.6.1.3 pour les deux méthodes des **String** employées). Une fois remplie, la table est simplement affichée sur la sortie standard.

```
  Assoc t = ... ;
  WordReader in = ... ;
```

[1]Vous noterez la différence avec les classes Fifo et Stack du chapitre précédent

```
  count(in, t) ;
  System.out.println(t.toString()) ;
```

Les points de la création du WordReader et du tri de la table dans sa méthode `toString` sont supposés résolus, nous estimerons donc le programme Freq écrit dès que nous aurons implémenté la table d'association nécessaire.

1.2 Implémentation simple de la table d'association

La technique d'implémentation la plus simple d'une table d'associations est d'envisager une liste des paires clé-information. Dans l'exemple de la classe Freq, les clés sont des chaînes et les informations des entiers, nous définissons donc une classe simple des cellules de listes.

```
class AList {
  String key ; int val ;
  AList next ;

  AList (String key, int val, List next) {
    this.key = key ; this.val = val ; this.next = next ;
  }
}
```

Nous aurions pu définir d'abord une classe des paires, puis une classe des listes de paires, mais nous préférons la solution plus économe en mémoire qui consiste à ranger les deux composantes de la paire directement dans la cellule de liste.

Nous dotons la classe AList d'une unique méthode (statique, **null** étant une liste valide) destinée à retrouver une paire clé-information à partir de sa clé.

```
  static AList getCell(String key, AList p) {
    for ( ; p != null ; p = p.next)
      if (key.equals(p.key)) return p ;
    return null ;
  }
```

La méthode `getCell` renvoie la première cellule de la liste `p` dont le champ vaut la clé passée en argument. Si cette cellule n'existe pas, **null** est renvoyé. Notons que les chaînes sont comparées par la méthode `equals` et non pas par l'opérateur `==`, c'est plus prudent (voir B.3.1.1). Cette méthode `getCell` suffit pour écrire la classe L des tables d'associations basée sur une liste d'association interne.

```
class L implements Assoc {

  private AList p ;

  L() { p = null ; }

  public int get(String key) {
    AList r = AList.getCell(key, p) ;
    if (r == null) { // Absent de la table
      return 0 ;
    } else {         // Présent dans la table
      return r.val ;
    }
  }

  public void put(String key, int val) {
    AList r = AList.getCell(key, p) ;
```

```
    if (r == null) {  // Absent de la table, créer une nouvelle association
      p = new AList (key, val, p) ;
    } else { // Présent dans la table, modifier l'ancienne association
      r.val = val ;
    }
  }
}
```

Les objets L encapsulent une liste d'association. Les méthodes `get` et `put` emploient toutes les deux la méthode AList`.getCell` pour retrouver l'information associée à la clé `key` passée en argument. La technique de l'encapsulage est désormais familière, nous l'avons déjà exploitée pour les ensembles, les piles et les files dans les chapitres précédents.

Mais il faut surtout noter une nouveauté : la classe L déclare implémenter l'interface Assoc (mot-clé **implements**). Cette déclaration entraîne deux conséquences importantes.

- Le compilateur Java vérifie que les objets de la classe L possèdent bien les deux méthodes spécifiées par l'interface Assoc, avec les signatures conformes. Il y a un détail étrange : les méthodes spécifiées par une interface sont obligatoirement **public**.
- Un objet L peut prendre le type Assoc, ce qui arrive par exemple dans l'appel suivant :

```
// in est un WordReader
   count(in, new L()) ; // Compter les mots de in.
```

Notez que `count` ne connaît pas la classe L, seulement l'interface Assoc.

Notre programme Freq fonctionne, mais il n'est pas très efficace. En effet si la liste d'association est de taille N, une recherche par `getCell` peut prendre de l'ordre de N opérations, en particulier dans le cas fréquent où la clé n'est pas dans la liste. Il en résulte que le programme Freq est en $O(n^2)$ où n est le nombre de mots de l'entrée. Pour atteindre une efficacité bien meilleure, nous allons introduire la nouvelle notion de table de hachage.

2 Table de hachage

La table de hachage est une implémentation efficace de la table d'association. Appelons *univers des clés* l'ensemble U de toutes les clés possibles. Nous allons d'abord observer qu'il existe un cas particulier simple quand l'univers des clés est un petit intervalle entier, puis ramener le cas général à ce cas simple.

2.1 Adressage direct

Cette technique très efficace ne peut malheureusement s'appliquer que dans des cas très particuliers. Il faut que l'univers des clés soit de la forme $\{0, \ldots, n-1\}$, où n est un entier pas trop grand, et d'autre part que deux éléments distincts aient des clés distinctes (ce que nous avons d'ailleurs déjà supposé). Il suffit alors d'utiliser un tableau de taille n pour représenter la table d'association.

Ce cas s'applique par exemple à la base de donnée des concurrents d'une épreuve sportive qui exclut les ex-æquos. Le rang à l'arrivée d'un concurrent peut servir de clé dans la base de données. Mais, ne nous leurrons pas un cas aussi simple est exceptionnel en pratique. Le fait pertinent est de remarquer que le problème de la recherche d'information se simplifie beaucoup quand les clés sont des entiers pris dans un petit intervalle.

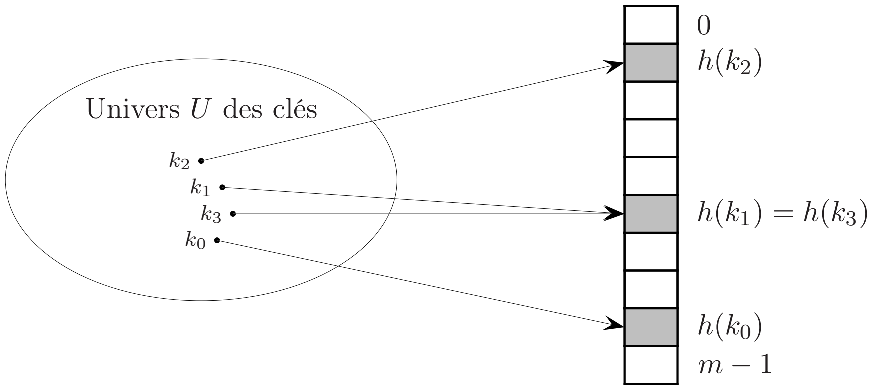
2.2 Table de hachage

L'idée fondamentale de la table de hachage est de se ramener au cas de l'adressage direct, c'est-à-dire de clés qui sont des indices dans un tableau. Soit m, entier pas trop grand, à vrai dire entier de l'ordre du nombre d'éléments d'informations que l'on compte gérer. On se donne une fonction

$$h : U \to \{0, \ldots, m-1\}$$

appelée *fonction de hachage*. L'idée est de ranger l'élément de clé k non pas dans une case de tableau `t[`k`]`, comme dans l'adressage direct (cela n'a d'ailleurs aucun sens si k n'est pas un entier) , mais dans `t[`$h(k)$`]`. Nous reviendrons en 3 sur le choix, relativement délicat, de la fonction de hachage. Mais nous devons affronter dès à présent une difficulté. En effet, il devient déraisonnable d'exclure le cas de clés (distinctes) k et k' telles que $h(k) = h(k')$. La figure 1 illustre la survenue d'une telle *collision* entre les clés k_1 et k_3 distinctes qui sont telles que $h(k_1) = h(k_3)$. Précisons un peu le problème, supposons que la collision survient lors de l'ajout

FIG. 1 – Une collision dans une table de hachage.



de l'élément d'information v_3 de clé k_3, alors qu'il existe déjà dans la table une clé k_1 avec $h(k_1) = h(k_3)$. La question est alors : où ranger l'information associée à la clé k_3 ?

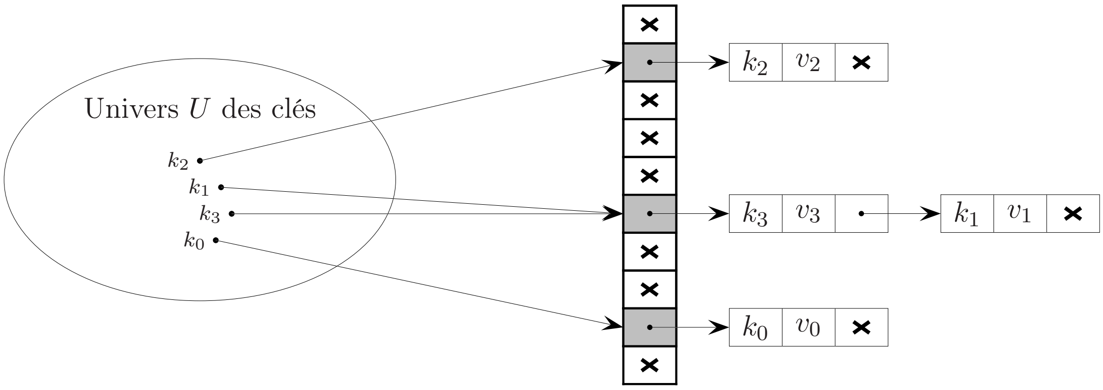
2.3 Résolution des collisions par chaînage

La solution la plus simple pour résoudre les collisions consiste à mettre tous les éléments d'information dont les clés ont même valeur de hachage dans une liste. On parle alors de résolution des collisions par chaînage. Dans le cas de l'exemple de collision de la figure 1, on obtient la situation de la figure 2. On remarque que les éléments de la table `t` sont tout bêtement des listes d'association, la liste `t[`i`]` regroupant tous les éléments d'information (k, v) de la table qui sont tels que $h(k)$ vaut l'indice i.

Nous proposons une nouvelle implémentation **H** des tables d'associations **Assoc**, en encapsulant cette fois une table de hachage.

```
class H implements Assoc {
  final static int SIZE=1024 ; // Assez grand ?
  private AList [] t ; // Tableau interne de listes d'associations.

  H() { t = new AList [SIZE] } ;

  private int hash(String key) { return Math.abs(key.hashCode()) % t.length ;}

  public int get(String key) {
    int h = hash(key) ;
    AList r = AList.getCell(key, t[h]) ;
```

FIG. 2 – Résolution des collisions par chaînage.




```
    if (r == null) {
      return 0 ;
    } else {
      return r.val ;
    }
  }

  public void put(String key, int val) {
    int h = hash(key) ;
    AList r = AList.getCell(key, t[h]) ;
    if (r == null) {
      t[h] = new AList(key, val, t[h]) ;
    } else {
      r.val = val ;
    }
  }
}
```

Le code de la fonction de hachage `hash` est en fait assez simple, parce qu'il utilise la méthode de hachage des chaînes fournie par Java (toute la complexité est cachée dans cette méthode) dont il réduit le résultat modulo la taille du tableau interne `t`, afin de produire un indice valide. La valeur absolue `Math.abs` est malheureusement nécessaire, car pour n négatif, l'opérateur « reste de la division euclidienne » `%` renvoie un résultat négatif (misère).

Il est surtout important de remarquer :

- Le code est en fait presque le même que celui de la classe **L** (page 67), en remplaçant `p` par `t[h]`.
- La classe **H** déclare implémenter l'interface **Assoc** et le fait effectivement ce que le compilateur vérifie. Un objet de la nouvelle classe **H** est donc un argument valide pour la méthode `count` de la classe **Freq**.

Estimons le coût de `put` et de `get` pour une table qui contient N éléments d'information. On suppose que le coût du calcul de la fonction hachage est en $O(1)$, et que hachage est uniforme, c'est-à-dire que la valeur de hachage d'une clé vaut $h \in [0 \ldots m[$ avec une probabilité $1/m$. Ces deux hypothèses sont réalistes. Pour la première, en supposant que le coût de calcul de la fonction de hachage est proportionnel à la longueur des mots, nous constatons que la longueur des mots d'un texte ordinaire de N mots est faible et indépendante de N.[2] La seconde hypothèse

[2]Un autre argument est de dire qu'il existe de l'ordre de $N = K^\ell$ mots de taille inférieure à ℓ, où K est le

traduit simplement que nous disposons d'une « bonne » fonction de hachage, faisons confiance à la méthode `hashCode` des **String**.

Sous ces deux hypothèses, la recherche d'un élément se fait en moyenne en temps $\Theta(1 + \alpha)$, où $\alpha = n/m$ est le *facteur de charge* (*load factor*) de la table (n est le nombre de clés à ranger et m est la taille du tableau). Plus précisément une recherche infructueuse dans la table parcourt en moyenne α cellules de listes, et une recherche fructueuse $1 + \alpha/2 - 1/(2m)$ cellules, coûts auxquels on ajoute le coût du calcul de la fonction de hachage. Ce résultat est démontré dans [2, section 12.2], contentons nous de remarquer que α est tout simplement la longueur moyenne des listes d'associations `t[`h`]`.

Peut-être faut il remarquer que le coût d'une recherche dans le cas le pire est $O(n)$, quand toutes les clés entrent en collision. Mais employer les tables de hachage suppose de faire confiance au hasard (hachage uniforme) et donc de considérer plus le cas moyen que le cas le pire. Une façon plus concrète de voir les choses est de considérer que, par exemple lors du comptage des mots d'un texte, on insère et recherche de nombreux mots uniformément hachés, et que donc le coût moyen donne une très bonne indication du coût rencontré en pratique.

Dans un premier temps, pour notre implémentation simple de **H** qui dimensionne le tableau `t` initialement, nous pouvons interpréter le résultat de complexité en moyenne d'une recherche en $\Theta(1 + \alpha)$, en constatant que si la taille du tableau interne est de l'ordre de n, alors nous avons atteint un coût (en moyenne) de `put` et `get` en temps constant. Il peut sembler que nous nous sommes livrés à une suite d'approximations et d'à-peu-près, et c'est un peu vrai. Il n'en reste pas moins, et c'est le principal, que les tables de hachage sont efficaces en pratique, essentiellement sous réserve que pour une exécution donnée, les valeurs de hachage des clés se répartissent uniformément parmi les indices du tableau interne correctement dimensionné, mais aussi que le coût du calcul de la fonction de hachage ne soit pas trop élevé. Dans cet esprit pragmatique, on peut voir la table de hachage comme un moyen simple de diviser le coût des listes d'association d'un facteur n, au prix de l'allocation d'un tableau de taille de l'ordre de n.

2.3.1 Complément : redimensionnement dynamique

Dans un deuxième temps, il est plus convenable, et ce sera aussi plus pratique, de redimensionner dynamiquement la table de hachage afin de maintenir le facteur de charge dans des limites raisonnables. Pour atteindre un coût amorti en temps constant pour `put`, il suffit de deux conditions (comme pour `push` dans le cas des piles, voir II.2.1)

- La taille des tableaux internes doit suivre une progression géométrique au cours du temps.
- Le coût du redimensionnement doit être proportionnel au nombre d'informations stockées dans la table au moment de ce redimensionnement.

Définissons d'abord une constante `alpha` qui est notre borne supérieure du facteur de charge, et une variable d'instance `nbKeys` qui compte le nombre d'associations effectivement présentes dans la table.

```
final static double alpha = 4.0 ;
private int nbKeys = 0 ;
final static int SIZE = 16 ;
```

Nous avons aussi changé la valeur de la taille par défaut de la table, afin de ne pas mobiliser une quantité conséquente de mémoire *a priori*. C'est aussi une bonne idée de procéder ainsi afin que le redimensionnement ait effectivement lieu et que le code correspondant soit testé.

La méthode de redimensionnement `resize`, à ajouter à la classe **H** double la taille du tableau interne `t`.

nombre de caractères possibles. Dans ce cas le coût du calcul de la fonction de hachage est en $O(\log N)$, réputé indépendant de n pour $n \ll N$.

```
private void resize() {
  int old_sz = t.length ;     // Ancienne taille
  int new_sz = 2*old_sz ;     // Nouvelle taille
  AList [] oldT = t ;         // garder une référence sur l'ancien tableau
  t = new AList [new_sz] ;    // Allouer le nouveau tableau
  /* Insérer toutes les paires clé-information de oldT
     dans le nouveau tableau t */
  for (int i = 0 ; i < old_sz ; i++) {
    for (AList p = oldT[i] ; p != null ; p = p.next) {
      int h = hash(p.key) ;
      t[h] = new AList (p.key, p.val, t[h]) ;
    }
  }
}
```

Il faut noter que la fonction de hachage `hash` qui transforme les clés en indices du tableau `t` dépend de la taille de `t` (de fait son code emploie **this**`.t.length`). Pour cette raison, le nouveau tableau est directement rangé dans le champ `t` de **this** et une référence sur l'ancien tableau est conservée dans la variable locale `oldT`, le temps de parcourir les paires clé-information contenues dans les listes de l'ancien tableau pour les ajouter dans le nouveau tableau `t`. Le redimensionnement n'est pas gratuit, il est même assez coûteux, mais il reste bien proportionnel au nombre d'informations stockées — sous réserve d'un calcul en temps constant de la fonction de hachage.

C'est la méthode `put` qui tient à jour le compte `nbKey` et appelle le méthode `resize` quand le facteur de charge `nbKeys/t.length` dépasse `alpha`.

```
public void put(String key, int val) {
  int h = hash(key) ;
  AList r = AList.getCell(key, t[h]) ;
  if (r == null) {
    t[h] = new AList(key, val, t[h]) ;
    nbKeys++ ;
    if (t.length * alpha < nbKeys) {
      resize() ;
    }
  } else {
    r.val = val ;
  }
}
```

Notez que le redimensionnement est, le cas échéant, effectué *après* ajout d'une nouvelle association. En effet, la valeur de hachage `h` n'est valide que relativement à la longueur de tableau `t`.

2.4 Adressage ouvert

Dans le hachage à adressage ouvert, les éléments d'informations sont stockés directement dans le tableau. Plus précisément, la table de hachage est un tableau de paires clé-information. Le facteur de charge α est donc nécessairement inférieur à un. Étant donnée une clé k on recherche l'information associée à k d'abord dans la case d'indice $h(k)$, puis, si cette case est occupée par une information de clé k' différente de k, on continue la recherche en suivant une séquence d'indices prédéfinie, jusqu'à trouver une case contenant une information dont la clé vaut k ou une une case libre. Dans le premier cas il existe un élément de clé k dans la table, dans le second il n'en existe pas. La séquence la plus simple consiste à examiner successivement les indices $h(k)$, $h(k)+1$, $h(k)+2$ etc. modulo m taille de la table. C'est le *sondage linéaire* (*linear probing*).

Pour ajouter une information (k, v), on procède exactement de la même manière, jusqu'à trouver une case libre ou une case contenant la paire (k, v'). Dans les deux cas, on dispose d'une case où ranger (k, v), au besoin en écrasant la valeur v' anciennement associée à k. Selon cette technique, une fois entrée dans la table, une clé reste à la même place dans le tableau et est accédée selon la même séquence, à condition de ne pas supprimer d'informations, ce que nous supposons.

Pour coder une nouvelle implémentation O de la table d'association Assoc, qui utilise le hachage avec adressage ouvert. Nous définissons d'abord une classe des paires clé-information.

```
class Pair {
  String key ; int val ;

  Pair(String key, int val) { this.key = key ; this.val = val ; }
}
```

Les objets O possèdent en propre un tableau d'objets Pair. Le code de la classe O est donné par la figure 3. Dans le constructeur, les cases du tableau **new** Pair [SIZE] sont implicitement initialisées à **null** (voir B.3.6.2), qui est justement la valeur qui permet à `getSlot` d'identifier les cases « vides ». La méthode `getSlot`, chargée de trouver la case où ranger une association en fonction de la clé, est appelée par les deux méthodes `put` et `get`. La relative complexité de `getSlot` justifie cette organisation. La méthode `getSlot` peut échouer, quand la table est pleine — notez l'emploi de la boucle **do**, voir B.3.4, ce qui rend la question du dimensionnement du tableau interne plus critique que dans le cas du chaînage.

Exercice 1 Modifier le code de la classe O afin de redimensionner automatiquement le tableau interne, dès que le facteur de charge dépasse une valeur critique `alpha`.

```
  final static double alpha = 0.5 ;
```

Solution. Comme dans le cas du chaînage, nous allons écrire une méthode privée `resize` chargée d'agrandir la table quand elle devient trop chargée. La méthode `put` est modifiée pour gérer le compte `nbKeys` des informations effectivement présentes dans la table, et appeler `resize` si besoin est.

```
  private int nbKeys = 0 ;

  public void put(String key, int val) {
    int h = getSlot(key) ;
    Pair p = t[h] ;
    if (p == null) {
      nbKeys++ ;
      t[h] = new Pair(key, val) ;
      if (t.length * alpha < nbKeys) resize() ;
    } else {
      p.val = val ;
    }
  }
```

La méthode `resize` fait appel à `getSlot` pour transférer les informations de l'ancienne à la nouvelle table. C'est le meilleur moyen de garantir des ajouts compatibles avec les méthodes `put` et `get`.

```
  private void resize() {
    int old_sz = t.length ;
    int new_sz = 2*old_sz ;
    Pair [] oldT = t ;
```

FIG. 3 – Implémentation d'une table de hachage à adressage ouvert

```
class O implements Assoc {
  private final static int SIZE = 1024 ; // Assez grand ?
  private Pair [] t ;      // Tableau interne de paires

  O() { t = new Pair[SIZE] ; }

  private int hash(String key) { return Math.abs(key.hashCode()) % t.length ; }

  /* Méthode de recherche de la case associée à key */
  private int getSlot(String key) {
    int h0 = hash(key) ;
    int h = h0 ;
    do {
      /* Si t[h] est « vide » ou contient la clé key, on a trouvé */
      if (t[h] == null || key.equals(t[h].key)) return h ;
      /* Sinon, passer à la case suivante */
      h++ ;
      if (h >= t.length) h = 0 ;
    } while (h != h0) ;
    throw new Error ("Table pleine") ; // On a fait le tour complet
  }

  public int get(String key) {
    Pair p = t[getSlot(key)] ;
    if (p == null) {
      return 0 ;
    } else {
      return p.val ;
    }
  }

  public void put(String key, int val) {
    int h = getSlot(key) ;
    Pair p = t[h] ;
    if (p == null) {
      t[h] = new Pair(key, val) ;
    } else {
      p.val = val ;
    }
  }
}
```

```
    t = new Pair[new_sz] ;
    for (int k = 0 ; k < old_sz ; k++) {
      Pair p = oldT[k] ;
      if (p != null) t[getSlot(p.key)] = p ;
      }
    }
  }
```

Il faut, comme dans le cas du chaînage, prendre la précaution de ranger le nouveau tableau dans la variable d'instance `t` avant de commencer à calculer les valeurs de hachage dans le nouveau tableau. On note aussi que `oldT[k]` peut valoir **null** et qu'il faut en tenir compte. □

On démontre [6, section 6.4] qu'une recherche infructueuse entraîne en moyenne l'examen d'environ $1/2 \cdot (1 + 1/(1 - \alpha)^2)$ cases et une recherche fructueuse d'environ $1/2 \cdot (1 + 1/(1 - \alpha))$, où α est le facteur de charge et sous réserve de hachage uniforme. Ces résultats ne sont *stricto sensu* plus valables pour α proche de un, mais les formules donnent toujours un majorant. Bref, pour un facteur de charge de 50 % on examine en moyenne pas plus de trois cases.

Le sondage linéaire provoque des phénomènes de regroupement (en plus des collisions). Considérons par exemple la table ci-dessous, où les cases encore libres sont en blanc :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

En supposant que $h(k)$ soit distribué uniformément, la probabilité pour que la case i soit choisie à la prochaine insertion est la suivante

$$\begin{array}{ccccc} P(0) = 1/19 & P(2) = 2/19 & P(3) = 1/19 & P(8) = 5/19 & P(9) = 1/19 \\ P(12) = 3/19 & P(13) = 1/19 & P(14) = 1/19 & P(16) = 2/19 & P(18) = 2/19 \end{array}$$

Comme on le voit, la case 8 a la plus grande probabilité d'être occupée, ce qui accentuera le le regroupement des cases 4–7. Ce phénomène se révèle rapidement quand des clés successives sont hachées sur des entiers successifs, un cas qui se présente en pratique avec le hachage modulo (voir 3), quand les clés sont par exemple les valeurs successives d'un compteur, ou, dans des applications plus techniques, des adresses d'objets qui se suivent dans la mémoire.

Plusieurs solutions ont été proposées pour éviter ce problème. La meilleure solution consiste à utiliser un *double hachage*. On se donne deux fonctions de hachage $h : U \to \{0, \ldots, m - 1\}$ et $h' : U \to \{1, \ldots, r - 1\}$ $(r < m)$. Ensuite le sondage est effectué selon la séquence $h(k) + h'(k)$, $h(k) + 2h'(k)$, $h(k) + 3h'(k)$, etc. Les regroupements ne sont plus à craindre essentiellement parce que l'incrément de la séquence est lui aussi devenu une fonction uniforme de la clé. En particulier en cas de collision selon h, il n'y a aucune raison que les sondages se fassent selon le même incrément. Pour que le sondage puisse parcourir toute la table on prend $h'(k) > 0$ et $h'(k)$ premier avec m taille de la table. Pour ce faire on peut prendre m égal à une puissance de deux et $h'(k)$ toujours impair, ou m premier et $h'(k)$ strictement inférieur à m (par exemple pour des clé entières $h(k) = k \bmod m$ et $h'(k) = 1 + (k \bmod (m - 2))$

Dans [6, section 6.4] D. Knuth montre que, sous des hypothèses de distribution uniforme et d'indépendance des deux fonctions de hachage, le nombre moyen de sondages pour un hachage double est environ $-\ln(1 - \alpha)/\alpha$ en cas de succès et à $1/(1 - \alpha)$ en cas d'échec.[3] Le tableau ci-dessous donne quelques valeurs numériques :

Facteur de charge	50 %	80 %	90 %	99 %
Succès	1.39	2.01	2.56	4.65
Echec	2.00	5.00	10.00	100.00

[3] Ces valeurs proviennent d'un modèle simplifié, et ont été vérifiées expérimentalement.

Comme on le voit $\alpha = 80$ % est un excellent compromis. Insistons sur le fait que les valeurs de ce tableau sont indépendantes de n. Par conséquent, avec un facteur charge de 80 %, il suffit en moyenne de deux essais pour retrouver un élément, même avec dix milliards de clés ! Notons que pour le sondage linéaire cet ordre de grandeur est atteint pour des tables à moitié pleines. Tandis que pour le chaînage on peut aller jusqu'à un facteur de charge d'environ 4.

En fixant ces valeurs de facteur de charge pour les trois techniques, nous égalisons plus ou moins les temps de recherche. Examinons alors la mémoire occupée pour n associations. Nous constatons que le chaînage consomme un tableau de taille $n/4$ plus n cellules de listes à trois champs, soit $3 + n/4 + 5 \cdot n$ cases de mémoire en Java, en tenant compte de deux cases supplémentaires par objet (voir II.4). Tandis que le sondage linéaire consomme $3 + 2 \cdot n + 4 \cdot n$ (tableau et paires), et le double hachage $3 + 5/4 \cdot n + 4 \cdot n$ (tableau et paires encore). Le gain des deux dernières techniques est donc minime, mais on peut coder avec moins de mémoire (par exemple en gérant deux tableaux, un pour les clés, un pour les valeurs). L'espace mémoire employé devient alors respectivement $6 + 4 \cdot n$ et $6 + 5/2 \cdot n$, soit finalement une économie de mémoire conséquente pour le double hachage.

2.5 Tables de hachage de la libraire

Il aurait été étonnant que la libraire de Java n'offre pas de table de hachage, tant cette structure est utile. La classe **HashMap** est une classe générique **HashMap**`<K,V>` paramétrée par les classes des clés `K` et des informations `V` (voir II.2.3). On écrit donc une dernière implémentation, très courte, de notre interface **Assoc**.

```
import java.util.* ;
class Lib implements Assoc {
  private HashMap<String,Integer> t ;

  H() { t = new HashMap<String,Integer> () ; }

  public int get(String key) {
    Integer val = t.get(key) ;
    if (val == null) {
      return 0 ;
    } else {
      return val ;
    }
  }

  public void put(String key, int val) { t.put(key,val) ; }
}
```

La table de hachage `t` est construite avec la taille initiale et le facteur de charge par défaut (respectivement 16 et 0.75). Les méthodes `t.put` et `t.get` sont celles des **HashMap**, qui se comportent comme les nôtres. À ceci près que clés et informations sont obligatoirement des objets et que l'appel `t.get(key)` renvoie **null** quand la clé `key` n'est pas dans la table `t`, fait que nous exploitons directement dans notre méthode `get`. On note la conversion automatique d'un **Integer** en **int** (**return** `val` dans le corps de `get`) et d'un **int** en **Integer** (appel `t.put(key,val)` dans `put`).

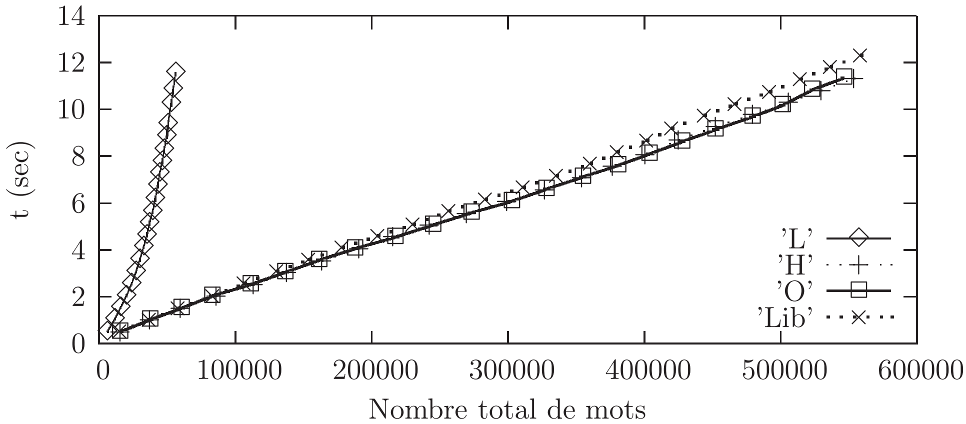
2.6 Performance

Nous nous livrons à une expérience consistant à appliquer le programme **Freq** à une série de fichiers contenant du source Java. Nous mesurons le temps cumulé d'exécution de la méthode `count` (celle qui lit les mots un par un et accumule les comptes dans la table d'association **Assoc**, page 66), pour les quatre implémentations des tables d'associations.

- La table **L** basée sur les listes d'associations.
- La table **H** basée sur le hachage avec chaînage (facteur de charge 4.0, taille initiale 16)
- La table **O** basée sur le hachage ouvert (sondage linéaire, facteur de charge 0.5, taille initiale 16)
- La table **Lib** basée sur les **HashMap** de la librairie (probablement chaînage, facteur de charge 0.75, taille initiale 16)

Toutes les tables de hachage sont redimensionnées automatiquement. Les résultats (figure 4) font

FIG. 4 – temps d'exécution de la méthode `count`



clairement apparaître que les listes d'associations sont hors-jeu et que les tables de hachage ont le comportement linéaire attendu et même que toutes les implémentations des tables de hachages se valent, au moins dans le cadre de cet expérience.

En dernière analyse, l'efficacité des tables de hachage repose sur l'uniformité de la fonction de hachage des chaînes de Java, c'est donc surtout cette dernière qui se montre performante ici. En voici pour preuve (figure 5) l'histogramme des effectifs des longueurs des listes de collisions à la fin de l'expérience **H** (tableau de taille $m = 4096$, nombre d'associations $n = 16092$). Cette figure montre par exemple qu'il y a un peu plus de 800 listes de collisions de longueur trois. En conclusion l'intérêt en pratique des tables de hachage est très important, compte tenu des performances atteintes pour une difficulté de réalisation particulièrement faible (surtout si on utilise la bibliothèque).

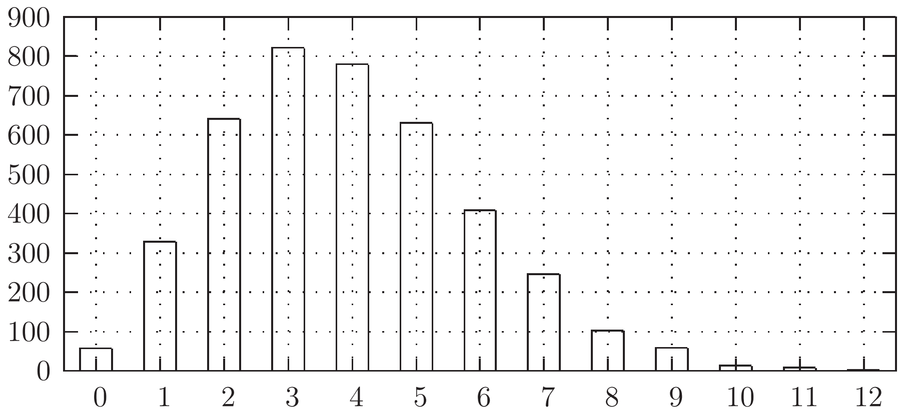
3 Choix des fonctions de hachage

3.1 En théorie

Rappelons qu'une fonction de hachage est une fonction $h : U \to \{0, \ldots m-1\}$. Une bonne fonction de hachage doit se rapprocher le plus possible d'une répartition uniforme. Formellement, cela signifie que si on se donne une probabilité P sur U et que l'on choisit aléatoirement chaque clé k dans U, on ait pour tout j avec $0 \leq j \leq m-1$,

$$\sum_{\{k \mid h(k)=j\}} P(k) = \frac{1}{m}$$

Fig. 5 – Répartition des collisions



En pratique, on connaît rarement la probabilité P et on se limite à des heuristiques. En particulier, on veut souvent que des clés voisines donnent des valeurs de hachages très distinctes.

Le cas que nous traitons ci-dessous est celui où l'univers U est un sous-ensemble des entiers naturels, car on peut toujours se ramener à ce cas. Par exemple, si les clés sont des chaînes de caractères, on peut interpréter chaque caractère comme un chiffre d'une base B et la chaîne comme un entier écrit dans cette base. C'est à dire que la chaîne $a_0 a_1 \ldots a_{n-1}$ est l'entier $a_0 \cdot B^{n-1} + a_1 \cdot B^{n-2} + \cdots + a_{n-1}$. Si les caractères sont quelconques on a $B = 2^{16}$ en Java et $B = 2^8$ en C ; si les caractères des clés sont limités à l'alphabet minuscule, on peut prendre $B = 26$; etc.

Nous pouvons donc revenir aux fonctions de hachage sur les entiers. Une technique courante consiste à prendre pour $h(k)$ le reste de la division de k par m :

$$h(k) = k \mod m$$

Mais dans ce cas, certaines valeurs de m sont à éviter. Par exemple, si on prend $m = 2^r$, $h(k)$ ne dépendra que des r derniers bits de k. Ce qui veut dire, par exemple, que le début des chaînes longues (vues comme des entiers en base $B = 2^p$) ne compte plus, et conduit à de nombreuses collisions si les clés sont par exemple des adverbes (se terminant toutes par `ment`). Dans le même contexte, si on prend $m = B - 1$, l'interversion de deux caractères passera inaperçue. En effet, comme $(a \cdot B + b) - (b \cdot B + a) = (a - b)(B - 1)$, on a

$$a \cdot B + b \equiv b \cdot B + a \pmod m$$

En pratique, une bonne valeur pour m est un nombre premier tel que $B^k \pm a$ n'est pas divisible par m, pour des entiers k et a petits.

Une autre technique consiste à prendre

$$h(k) = \lfloor m(Ck - \lfloor Ck \rfloor) \rfloor$$

où C est une constante réelle telle que $0 < C < 1$. Cette méthode a l'avantage de pouvoir s'appliquer à toutes les valeurs de m. Il est même conseillé dans ce cas de prendre $m = 2^r$ pour faciliter les calculs. Pour le choix de la constante C, Knuth recommande le nombre d'or, $C = (\sqrt{5} - 1)/2 \approx 0,618$.

3.2 En pratique

Continuons de considérer les chaînes comme des entiers en base 2^{16}. Ces entiers sont très vite grands, ils ne tiennent plus dans un **int** dès que la chaîne est de taille supérieure à 2.

On ne peut donc pas (pour un coût raisonnable, il existe des entiers en précision arbitraire) d'abord transformer la chaîne en entier, puis calculer h. Dans le cas du hachage modulo un nombre premier, il reste possible de calculer modulo m. Mais en fait ce n'est pas très pratique : la fonction de hachage (méthode `hashCode`) des chaînes de Java est indépendante de la tailles des tableaux internes des tables, puisqu'elle est définie dans la classe des chaînes et ne prend pas une taille de tableau en argument. À toute chaîne elle associe un **int** qui est ensuite réduit en indice. Selon la documentation, l'appel `hashCode()` de la chaîne $a_0 a_1 \ldots a_{n-1}$ renvoie $a_0 \cdot 31^{n-1} + a_1 \cdot 31^{n-2} + \cdots + a_{n-2} \cdot 31 + a_{n-1}$. Autrement dit le code de `hashCode` pourrait être celui-ci :

```
public int hashCode() {
  int h = 0 ;
  for (int k = 0 ; k < this.length ; k++)
    h = 31 * h + this.charAt(k) ;
  return h ;
}
```

Dans nos tables de hachage nous avons ensuite simplement réduit cet entier modulo la taille du tableau interne.

Le calcul `h = 31 * h + this.charAt(k)` effectue un mélange entre une valeur courante de `h` (qui est la valeur de hachage du préfixe de la chaîne) et la valeur de hachage d'un caractère de la chaîne qui est le caractère lui-même, c'est-à-dire son code en Unicode (voir B.3.2.3). Le multiplicateur 31 est un nombre premier, et ce n'est pas un hasard. L'expérience nous a montré que ce mélange simple fonctionne en pratique, sur un ensemble de clés qui sont les mots que l'on trouve dans des sources Java (voir en particulier la figure 5). Pour des clés plus générales cette façon de mélanger est critiquée [8], mais nous allons nous en tenir à elle.

Car il faut parfois construire nos propres fonction de hachage, ou plus exactement redéfinir nos propres méthodes `hashCode`. Notre table de hachage `H` appelle d'abord la méthode `hashCode` d'une clé (pour trouver un indice dans le tableau interne, page 69), puis la méthode `equals` de la même clé (pour par exemple trouver sa place dans une liste de collision, page 67). La table de hachage de la librairie procède similairement. Or, même si tous les objets possèdent bien des méthodes `hashCode` et `equals` (comme ils possèdent une méthode `toString`), les méthodes par défaut ne conviennent pas. En effet `equals` par défaut exprime l'égalité physique des clés (voir B.3.1.1), tandis que `hashCode` renvoie essentiellement l'adresse en mémoire de l'objet ! Il faut redéfinir ces deux méthodes, comme nous avons déjà parfois redéfini la méthode `toString` (voir B.2.3). La classe **String** ne procède pas autrement, son `hashCode` et son `equals` se basent non pas sur l'adresse en mémoire de la chaîne, mais sur le contenu des chaînes.

Supposons donc que nous voulions nous servir de paires d'entiers comme clés, c'est à dire d'objets d'une classe `Pair`.

```
class Pair {
  int x, y ;
  Pair (int x, int y) { this.x = x ; this.y = y ; }
}
```

Pour redéfinir `hashCode` nous utilisons tout simplement le mélangeur multiplicatif.

```
  public int hashCode() { return x * 31 + y ; }
```

La méthode redéfinie est déclarée **public** comme la méthode d'origine. Il importe en fait de respecter toute la signature de la méthode d'origine. Or, la signature de la méthode `equals` des **Object** (celle que nous voulons redéfinir) est :

```
  public boolean equals(Object o)
```

Nous écrivons donc (dans la classe `Pair`) :

```
public boolean equals(Object o) {
  Pair p = (Pair)o ; // Conversion de type, échoue si o n'est pas un Pair
  return this.x == p.x && this.y == p.y ;
}
```

Pour la conversion de type, voir B.3.2.2. Évidemment, pour que les tables de hachage fonctionnent correctement il faut que l'égalité selon `equals` (l'appel p_1`.equals(`p_2`)` renvoie **true**) entraîne l'égalité des code de hachage (p_1`.hashCode() ==` p_2`.hashCode()`), ce que la documentation de la bibliothèque appelle le contrat général de `hashCode`.

Chapitre IV

Arbres

Ce chapitre est consacré aux arbres, l'un des concepts algorithmiques les plus importants de l'informatique. Les arbres servent à représenter un ensemble de données structurées hiérarchiquement. Plusieurs notions distinctes se cachent en fait sous cette terminologie : arbres libres, arbres enracinés, arbres binaires, etc. Ces définitions sont précisées dans la section 1.

Nous présentons plusieurs applications des arbres : les arbres de décision, les files de priorité, le tri par tas et l'algorithme baptisé « union-find », qui s'applique dans une grande variété de situations. Les arbres binaires de recherche seront traités dans le chapitre suivant.

1 Définitions

Pour présenter les arbres de manière homogène, quelques termes empruntés aux graphes s'avèrent utiles. Nous présenterons donc les graphes, puis successivement, les arbres libres, les arbres enracinés et les arbres ordonnés.

1.1 Graphes

Un *graphe* $G = (S, A)$ est un couple formé d'un ensemble de *nœuds* S et d'un ensemble A d'*arcs*. L'ensemble A est une partie de $S \times S$. Les nœuds sont souvent représentés par des points dans le plan, et un arc $a = (s, t)$ par une ligne orientée joignant s à t. On dit que l'arc a part de s et va à t. Un chemin de s à t est une suite $(s = s_0, \ldots, s_n = t)$ de nœuds tels que, pour $1 \leqslant i \leqslant n$, (s_{i-1}, s_i) soit un arc. Le nœud s_0 est *l'origine* du chemin et le nœud s_n son *extrémité*. L'entier n est la *longueur* du chemin. C'est un entier positif ou nul. Un *circuit* est un chemin de longueur non nulle dont l'origine coïncide avec l'extrémité.

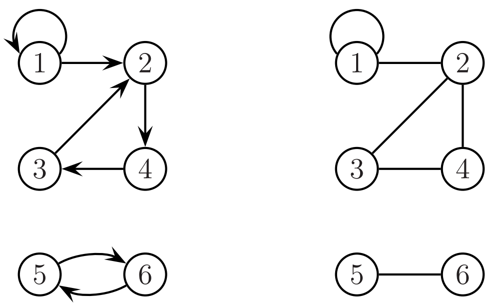


FIG. 1 – À gauche un graphe, à droite un graphe non orienté.

À côté de ces graphes, appelés aussi *graphes orientés* (« digraph » en anglais, pour « directed graph »), il existe la variante des graphes *non orientés*. Au lieu de couples de nœuds, on considère des *paires* $\{s, t\}$ de nœuds. Un graphe non orienté est donné par un ensemble de ces paires, appelées *arêtes*. Les concepts de chemin et circuit se transposent sans peine à ce contexte.

Un chemin est *simple* si tous ses nœuds sont distincts. Un graphe est *connexe* si deux quelconques de ses nœuds sont reliés par un chemin.

1.2 Arbres libres

Dans la suite de chapitre, nous présentons des familles d'arbres de plus en plus contraints. La famille la plus générale est formée des arbres libres. Un *arbre libre* est un graphe non orienté non vide, connexe et sans circuit. La proposition suivante est laissée en exercice.

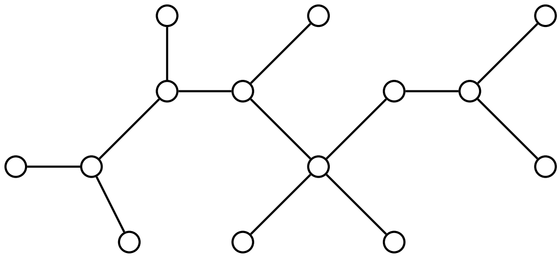


FIG. 2 – Un arbre « libre ».

Proposition 1 Soit $G = (S, A)$ un graphe non orienté non vide. Les conditions suivantes sont équivalentes :

(1) G est un arbre libre,

(2) Deux nœuds quelconques de S sont connectés par un chemin simple unique,

(3) G est connexe, mais ne l'est plus si l'on retire une arête quelconque,

(4) G est sans circuit, mais ne l'est plus si l'on ajoute une arête quelconque,

(5) G est connexe, et $\mathrm{Card}(A) = \mathrm{Card}(S) - 1$,

(6) G est sans circuit, et $\mathrm{Card}(A) = \mathrm{Card}(S) - 1$.

1.3 Arbres enracinés

Un *arbre enraciné* ou *arbre* (« rooted tree » en anglais) est un arbre libre muni d'un nœud distingué, appelé sa *racine*. Soit T un arbre de racine r. Pour tout nœud x, il existe un chemin simple unique de r à x. Tout nœud y sur ce chemin est un *ancêtre* de x, et x est un *descendant* de y. Le *sous-arbre* de racine x est l'arbre contenant tous les descendants de x. L'avant-dernier nœud y sur l'unique chemin reliant r à x est le *parent* (ou le *père* ou la *mère*) de x, et x est un *enfant* (ou un *fils* ou une *fille*) de y. L'*arité* d'un nœud est le nombre de ses enfants. Un nœud sans enfant est une *feuille*, un nœud d'arité strictement positive est appelé *nœud interne*. La *hauteur* d'un arbre T est la longueur maximale d'un chemin reliant sa racine à une feuille. Un arbre réduit à un seul nœud est de hauteur 0.

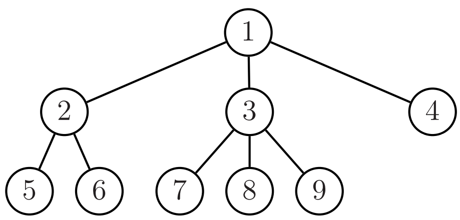


FIG. 3 – Un arbre « enraciné ».

Les arbres admettent aussi une définition récursive. Un arbre sur un ensemble fini de nœuds est un couple formé d'un nœud particulier, appelé sa racine, et d'une partition des nœuds restants en un ensemble d'arbres. Par exemple, l'arbre de la figure 3 correspond à la définition

$$T = (1, \{(2, \{(5), (6)\}), (3, \{(7), (8), (9)\}), (4)\})$$

Cette définition récursive est utile dans les preuves et dans la programmation. On montre ainsi facilement que si tout nœud interne d'un arbre est d'arité au moins 2, alors l'arbre a strictement plus de feuilles que de nœuds internes.

Une *forêt* est un ensemble d'arbres.

1.4 Arbres ordonnés

Un arbre *ordonné* est un arbre dans lequel l'ensemble des enfants de chaque nœud est totalement ordonné.

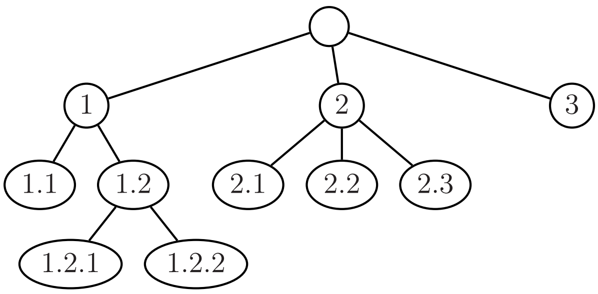


FIG. 4 – L'arbre ordonné de la table des matières d'un livre.

Par exemple, un livre, structuré en chapitres, sections, etc., se présente comme un arbre ordonné (voir figure 4). Les enfants d'un nœud d'un arbre ordonné sont souvent représentés, dans un programme, par une liste attachée au nœud. Un autre solution est d'associer, à chaque nœud, un tableau de fils. C'est une solution moins souple si le nombre de fils est destiné à changer. Enfin, on verra plus loin une autre représentation au moyen d'arbres binaires.

2 Union-Find, ou gestion des partitions

Comme premier exemple de l'emploi des arbres et des forêts, nous considérons un problème célèbre, et à ce jour pas encore entièrement résolu, appelé le problème *Union-Find*. Rappelons qu'une *partition* d'un ensemble E est un ensemble de parties non vides de E, deux à deux disjointes et dont la réunion est E. Étant donné une partition de l'ensemble $\{0, \ldots, n-1\}$, on veut résoudre les deux problèmes que voici :

- trouver la classe d'un élément (*find*)
- faire l'union de deux classes (*union*).

Nous donnons d'abord une solution du problème Union-Find, puis nous donnerons quelques exemples d'application.

2.1 Une solution du problème

En général, on part d'une partition où chaque classe est réduite à un singleton, puis on traite une suite de requêtes de l'un des deux types ci-dessus.

Avant de traiter ce problème, il faut imaginer la façon de représenter une partition. Une première solution consiste à représenter la partition par un tableau `classe`. Chaque classe est identifiée par un entier par exemple, et classe[x] contient le numéro de la classe de l'élément `x` (cf. figure 5).

x	0	1	2	3	4	5	6	7	8	9
classe[x]	2	3	1	4	4	1	2	4	1	4

FIG. 5 – Tableau associé à la partition $\{\{2,5,8\},\{0,6\},\{1\},\{3,4,7,9\}\}$.

Trouver la classe d'un élément se fait en temps constant, mais fusionner deux classes prend un temps $O(n)$, puisqu'il faut parcourir tout le tableau pour repérer les éléments dont il faut changer la classe. Une deuxième solution, que nous détaillons maintenant, consiste à choisir un représentant dans chaque classe. Fusionner deux classes revient alors à changer de représentant pour les éléments de la classe fusionnée. Il apparaît avantageux de représenter la partition par une forêt. Chaque classe de la partition constitue un arbre de cette forêt. La racine de l'arbre est le représentant de sa classe. La figure 6 montre la forêt associée à une partition.

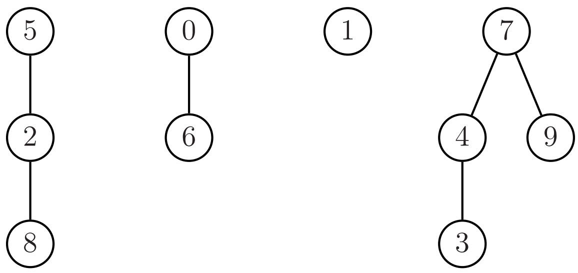


FIG. 6 – Forêt associée à la partition $\{\{2,5,8\},\{0,6\},\{1\},\{3,4,7,9\}\}$.

Une forêt est représentée par un tableau d'entiers pere (cf. Figure 7). Chaque nœud est représenté par un entier, et l'entier pere[x] est le père du nœud x. Une racine r n'a pas de parent. On convient que, dans ce cas, pere[r] = r.

x	0	1	2	3	4	5	6	7	8	9
pere[x]	0	1	5	4	7	5	0	7	2	7

FIG. 7 – Tableau associé à la forêt de la figure 6.

On suppose donc défini un tableau

```
int[] pere = new int[n];
```

Ce tableau est initialisé à l'identité par

```
static void initialisation()
{
  for (int i = 0; i < pere.length ; i++)
    pere[i] = i;
}
```

Chercher le représentant de la classe contenant un élément donné revient à trouver la racine de l'arbre contenant un nœud donné. Ceci se fait par la méthode suivante :

```
static int trouver(int x)
```

```
{
  while (x != pere[x])
    x = pere[x];
  return x;
}
```

L'union de deux arbres se réalise en ajoutant la racine de l'un des deux arbres comme nouveau fils à la racine de l'autre :

```
static void union(int x, int y)
{
  int r = trouver(x);
  int s = trouver(y);
  if (r != s)
    pere[r] = s;
}
```

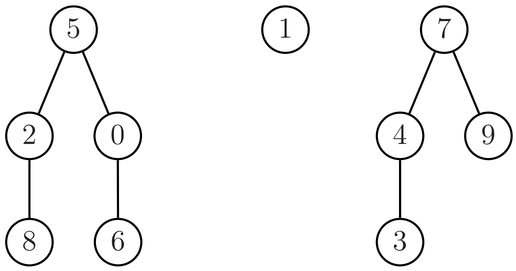


FIG. 8 – La forêt de la figure 6 après l'union pondérée de 5 et 0.

Il n'est pas difficile de voir que chacune de ces deux méthodes est de complexité $O(h)$, où h est la hauteur l'arbre (la plus grande des hauteurs des deux arbres). En fait, on peut améliorer l'efficacité de l'algorithme par la règle suivante (voir figure 8) :

Règle. *Lors de l'union de deux arbres, la racine de l'arbre de moindre taille devient fils de la racine de l'arbre de plus grande taille.*

Pour mettre en œuvre cette stratégie, on utilise un tableau supplémentaire qui mémorise la taille des arbres, qui doit être initialisé à 1 :

```
int[] taille = new int[n];
```

La nouvelle méthode d'union s'écrit alors :

```
static void unionPondérée(int x, int y)
{
  int r = trouver(x);
  int s = trouver(y);
  if (r == s)
    return;
  if (taille[r] > taille[s])
  {
    pere[s] = r;
    taille[r] += taille[s];
  }
  else
  {
```

```
      pere[r] = s;
      taille[s] += taille[r];
    }
  }
```

L'intérêt de cette méthode vient de l'observation suivante :

Lemme 2 La hauteur d'un arbre à n nœuds créé par union pondérée est au plus $1 + \lfloor \log_2 n \rfloor$.

Preuve. Par récurrence sur n. Pour $n = 1$, il n'y a rien à prouver. Si un arbre est obtenu par union pondérée d'un arbre à m nœuds et d'un arbre à $n - m$ nœuds, avec $1 \leqslant m \leqslant n/2$, sa hauteur est majorée par

$$\max(1 + \lfloor \log_2 (n - m) \rfloor, 2 + \lfloor \log_2 m \rfloor) .$$

Comme $\log_2 m \leqslant \log_2 (n/2) = \log_2 n - 1$, cette valeur est majorée par $1 + \lfloor \log_2 n \rfloor$. □

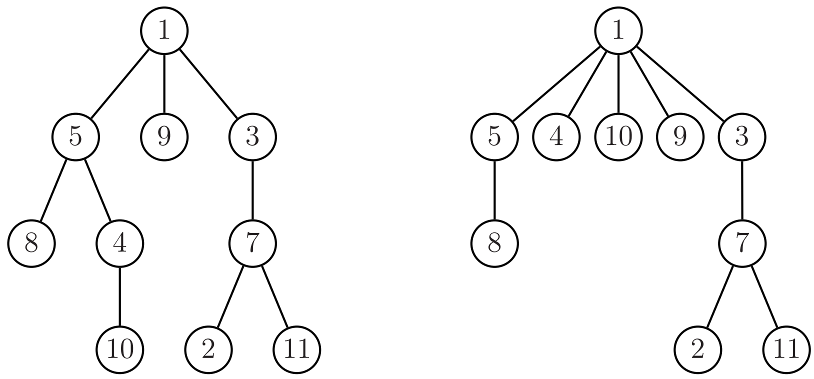


Fig. 9 – « Trouver » 10 avec compression du chemin.

Une deuxième stratégie, appliquée cette fois-ci lors de la méthode `trouver` permet une nouvelle amélioration considérable de la complexité. Elle est basée sur la règle de compression de chemins suivante :

Règle. *Après être remonté du nœud x à sa racine r, on refait le parcours en faisant de chaque nœud rencontré un fils de r.*

La figure 9 montre la transformation d'un arbre par une compression de chemin. Chacun des nœuds 10 et 4 devient fils de 1. L'implantation de cette règle se fait simplement.

```
static int trouverAvecCompression(int x)
{
  int r = trouver(x);
  while (x != r)
  {
    int y = pere[x];
    pere[x] = r;
    x = y;
  }
  return r;
}
```

L'ensemble des deux stratégies permet d'obtenir une complexité presque linéaire.

Théorème 3 (Tarjan) Avec l'union pondérée et la compression des chemins, une suite de $n-1$ « unions » et de m « trouver » $(m \geqslant n)$ se réalise en temps $O(n + m\alpha(n,m))$, où α est l'inverse d'une sorte de fonction d'Ackermann.

En fait, on a $\alpha(n,m) \leqslant 2$ pour $m \geqslant n$ et $n < 2^{65536}$ et par conséquent, l'algorithme précédent se comporte, d'un point de vue pratique, comme un algorithme linéaire en $n+m$. Pourtant, Tarjan a montré qu'il *n'est pas* linéaire et on ne connaît pas à ce jour d'algorithme linéaire.

2.2 Applications de l'algorithme Union-Find

Un premier exemple est la construction d'un *arbre couvrant* un graphe donné. Au départ, chaque nœud constitue à lui seul un arbre. On prend ensuite les arêtes, et on fusionne les arbres contenant les extrémités de l'arête si ces extrémités appartiennent à des arbres différents.

Un second exemple concerne les problèmes de connexion dans un réseau. Voici un exemple de tel problème. Huit ordinateurs sont connectés à travers un réseau. L'ordinateur 1 est connecté au 3, le 2 au 3, le 5 au 4, le 6 au 3, le 7 au 5, le 1 au 6 et le 7 au 8. Est-ce que les ordinateurs 4 et 6 peuvent communiquer à travers le réseau ? Certes, il n'est pas très difficile de résoudre ce problème à la main, mais imaginez la même question pour un réseau dont la taille serait de l'ordre de plusieurs millions. Comment résoudre ce problème efficacement ? C'est la même solution que précédemment ! On considère le réseau comme un graphe dont les nœuds sont les ordinateurs. Au départ, chaque nœud constitue à lui seul un arbre. On prend ensuite les arêtes (i.e. les connexions entre deux ordinateurs), et on fusionne les arbres contenant les extrémités de l'arête si ces extrémités appartiennent à des arbres différents.

3 Arbres binaires

La notion d'arbre binaire est assez différente des définitions précédentes. Un *arbre binaire* sur un ensemble fini de nœuds est soit vide, soit l'union disjointe d'un nœud appelé sa *racine*, d'un arbre binaire appelé *sous-arbre gauche*, et d'un arbre binaire appelé *sous-arbre droit*. Il est utile de représenter un arbre binaire non vide sous la forme d'un triplet $A = (A_g, r, A_d)$.

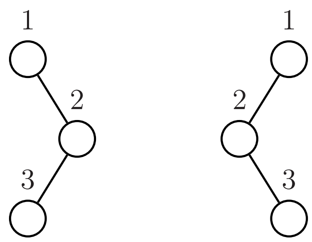


Fig. 10 – Deux arbres binaires différents.

Par exemple, l'arbre binaire sur la gauche de la figure 10 est $(\emptyset, 1, ((\emptyset, 3, \emptyset), 2, \emptyset))$, alors que l'arbre sur la droite de la figure 10 est $((\emptyset, 2, (\emptyset, 3, \emptyset)), 1, \emptyset)$. Cet exemple montre qu'un arbre binaire n'est pas simplement un arbre ordonné dont tous les nœuds sont d'arité au plus 2.

La *distance* d'un nœud x à la racine ou la *profondeur* de x est égale à la longueur du chemin de la racine à x. La *hauteur* d'un arbre binaire est égale à la plus grande des distances des feuilles à la racine.

Proposition 4 Soit A un arbre binaire à n nœuds, de hauteur h. Alors $h + 1 \geqslant \log_2(n+1)$.

Preuve. Il y a au plus 2^i nœuds à distance i, donc $n \leqslant 2^{h+1} - 1$. $\square$

Un arbre binaire est *complet* si tout nœud a 0 ou 2 fils.

Proposition 5 Dans un arbre binaire complet, le nombre de feuilles est égal au nombre de nœuds internes, plus 1.

Preuve. Notons, pour simplifier, $f(A)$ le nombre de feuilles et $n(A)$ le nombre de nœuds internes de l'arbre binaire complet A. Il s'agit de montrer que $f(A) = n(A) + 1$.

Le résultat est vrai pour l'arbre binaire de hauteur 0. Considérons un arbre binaire complet $A = (A_g, r, A_d)$. Les feuilles de A sont celles de A_g et de A_d et donc $f(A) = f(A_g) + f(A_d)$. Les nœuds internes de A sont ceux de A_g, ceux de A_d et la racine, et donc $n(A) = n(A_g) + n(A_d) + 1$. Comme A_g et A_d sont des arbres complets de hauteur inférieure à celle de A, la récurrence s'applique et on a $f(A_g) = n(A_g) + 1$ et $f(A_d) = n(A_d) + 1$. On obtient finalement $f(A) = f(A_g) + f(A_d) = (n(A_g) + 1) + (n(A_d) + 1) = n(A) + 1$. □

On montre aussi que, dans un arbre binaire complet, il y a un nombre pair de nœuds à chaque niveau, sauf au niveau de la racine.

3.1 Compter les arbres binaires

La figure 11 montre les arbres binaires ayant 1, 2, 3 et 4 nœuds.

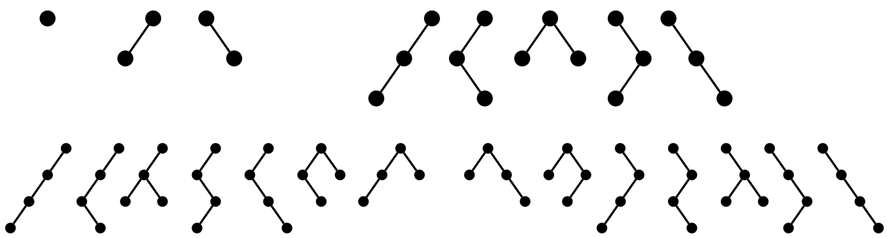


FIG. 11 – Les premiers arbres binaires.

Notons b_n le nombre d'arbres à n nœuds. On a donc $b_0 = b_1 = 1$, $b_2 = 2$, $b_3 = 5$, $b_4 = 14$. Comme tout arbre A non vide s'écrit de manière unique sous forme d'un triplet (A_g, r, A_d), on a pour $n \geqslant 1$ la formule

$$b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1}$$

La série génératrice $B(x) = \sum_{n \geqslant 0} b_n x^n$ vérifie donc l'équation

$$xB^2(x) - B(x) + 1 = 0\,.$$

Comme les b_n sont positifs, la résolution de cette équation donne

$$b_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{n!(n+1)!}$$

Les nombres b_n sont connus comme les *nombres de Catalan*. L'expression donne aussi $b_n \sim \pi^{-1/2} \cdot 4^n \cdot n^{-3/2} + O(4^n \cdot n^{-5/2})$.

3.2 Arbres binaires et mots

Nous présentons quelques concepts sur les mots qui servent à plusieurs reprises. D'abord, ils mettent en évidence des liens entre les parcours d'arbres binaires et certains ordres. Ensuite, ils seront employés dans des algorithmes de compression.

3.2.1 Mots

Un *alphabet* est un ensemble de *lettres*, comme $\{0, 1\}$ ou $\{a, b, c, d, r\}$. Un *mot* est une suite de lettres, comme 0110 ou *abracadabra*. La *longueur* d'un mot u, notée $|u|$, est le nombre de lettres de u : ainsi, $|0110| = 4$ et $|abracadabra| = 11$. Le mot vide, de longueur 0, est noté ε. Etant donnés deux mots, le mot obtenu par *concaténation* est le mot formé des deux mots juxtaposés. Le produit de concaténation est noté comme un produit. Si $u = abra$ et $v = cadabra$, alors $uv = abracadabra$. Un mot p est *préfixe* (propre) d'un mot u s'il existe un mot v (non vide) tel que $u = pv$. Ainsi, ε, abr, $abrac$ sont des préfixes propres de $abraca$. Un ensemble de mots P est *préfixiel* si tout préfixe d'un mot de P est lui-même dans P. Par exemple, les ensembles $\{\varepsilon, 1, 10, 11\}$ et $\{\varepsilon, 0, 00, 01, 000, 001\}$ sont préfixiels.

3.2.2 Ordres sur les mots

Un ordre total sur l'alphabet s'étend en un ordre total sur l'ensemble des mots de multiples manières. Nous considérons deux ordres, l'ordre lexicographique et l'ordre des mots croisés (« radix order » ou « shortlex » en anglais).

L'ordre lexicographique, ou ordre du dictionnaire, est défini par $u <_{\text{lex}} v$ si seulement si u est préfixe de v ou u et v peuvent s'écrire $u = pau'$, $v = pbv'$, où p est un mot, a et b sont des lettres, et $a < b$. L'ordre des mots croisés est défini par $u <_{\text{mc}} v$ si et seulement si $|u| < |v|$ ou $|u| = |v|$ et $u <_{\text{lex}} v$. Par exemple, on a

$$bar <_{\text{mc}} car <_{\text{mc}} barda <_{\text{mc}} radar <_{\text{mc}} abracadabra$$

3.2.3 Codage des arbres binaires

Chaque arête (p, f) d'un arbre A binaire est étiquetée par 0 si f est fils gauche de p, et par 1 si f est fils droit. L'étiquette du chemin qui mène de la racine à un nœud est le mot formé des étiquettes de ses arêtes. Le *code* de l'arbre A est l'ensemble des étiquettes des chemins issus de la racine. Cet ensemble est clairement préfixiel, et réciproquement, tout ensemble fini préfixiel de mots formés de 0 et 1 est le code d'un arbre binaire. La correspondance est, de plus, bijective.

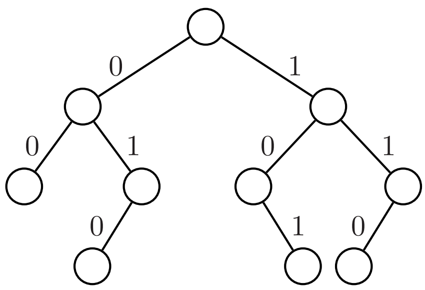


FIG. 12 – Le code de l'arbre est $\{\varepsilon, 0, 1, 00, 01, 10, 11, 010, 101, 110\}$.

Le code $c(A)$ d'un arbre binaire A se définit d'ailleurs simplement par récurrence : on a $c(\emptyset) = \{\varepsilon\}$ et si $A = (A_g, r, A_d)$, alors $c(A) = 0c(A_g) \cup \{\varepsilon\} \cup 1c(A_d)$.

3.3 Parcours d'arbre

Un *parcours d'arbre* est une énumération des nœuds de l'arbre. Chaque parcours définit un ordre sur les nœuds, déterminé par leur ordre d'apparition dans cette énumération.

On distingue les parcours de gauche à droite, et les parcours de droite à gauche. Dans un parcours de gauche à droite, le fils gauche d'un nœud précède le fils droit (et vice-versa pour un parcours de droite à gauche). Ensuite, on distingue les parcours en profondeur et en largeur.

Le *parcours en largeur* énumère les nœuds niveau par niveau. Ainsi, le parcours en largeur de l'arbre 13 donne la séquence $a, b, c, d, e, f, g, h, i, k$. On remarque que les codes des nœuds correspondants, ε, 0, 1, 00, 01, 10, 11, 010, 101, 110, sont en ordre croissant pour l'ordre des mots croisés. C'est en fait une règle générale.

Règle. *L'ordre du parcours en largeur correspond à l'ordre des mots croisés sur le code de l'arbre.*

On définit trois *parcours en profondeur* privilégiés qui sont

- le *parcours préfixe* : tout nœud est suivi des nœuds de son sous-arbre gauche puis des nœuds de son sous-arbre droit, en abrégé NGD (Nœud, Gauche, Droite).
- le *parcours infixe* : tout nœud est précédé des nœuds de son sous-arbre gauche et suivi des nœuds de son sous-arbre droit, en abrégé GND (Gauche, Nœud, Droite).
- le *parcours suffixe*, ou *postfixe* : tout nœud est précédé des nœuds de son sous-arbre gauche puis des nœuds de son sous-arbre droit, en abrégé GDN (Gauche, Droite, Nœud).

Les ordres correspondant sont appelés *ordres préfixe, infixe et suffixe*. Considérons l'arbre de la figure 13.

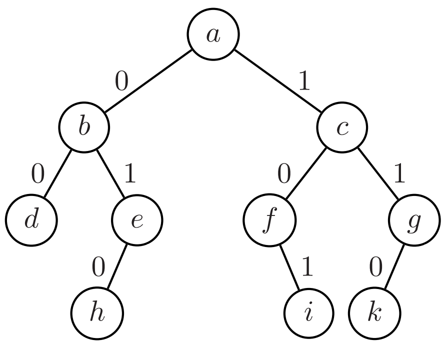


FIG. 13 – Un arbre binaire. Les nœuds sont nommés par des lettres.

Le parcours préfixe donne les nœuds dans l'ordre $a, b, d, e, h, c, f, i, g, k$, le parcours infixe donne la suite $d, b, h, e, a, f, i, c, k, g$ et le parcours suffixe donne $d, h, e, b, i, f, k, g, c, a$. De manière formelle, les parcours préfixe (infixe, suffixe) sont définis comme suit. Si A est l'arbre vide, alors $\mathrm{pref}(A) = \mathrm{inf}(A) = \mathrm{suff}(A) = \varepsilon$; si $A = (A_g, r, A_d)$, et $e(r)$ est le nom de r, alors

$$\begin{aligned}
\mathrm{pref}(A) &= e(r)\mathrm{pref}(\mathrm{A_g})\mathrm{pref}(\mathrm{A_d})\\
\mathrm{inf}(A) &= \mathrm{inf}(\mathrm{A_g})e(r)\mathrm{inf}(\mathrm{A_d})\\
\mathrm{suff}(A) &= \mathrm{suff}(\mathrm{A_g})\mathrm{suff}(\mathrm{A_d})e(r)
\end{aligned}$$

Règle. *Le parcours préfixe d'un arbre correspond à l'ordre lexicographique sur le code de l'arbre. Le parcours suffixe correspond à l'opposé de l'ordre lexicographique si l'on convient que* $1 < 0$.

Qu'on se rassure, il y a aussi une interprétation pour le parcours infixe, mais elle est un peu plus astucieuse ! À chaque nœud x de l'arbre, on associe un nombre formé du code du chemin menant à x, suivi de 1. Ce code complété est interprété comme la partie fractionnaire d'un nombre entre 0 et 1, écrit en binaire. Pour l'arbre de la figure 13, les nombres obtenus sont donnés dans la table suivante

a	.1	=	1/2
b	.01	=	1/4
c	.11	=	3/4
d	.001	=	1/8
e	.011	=	3/8
f	.101	=	5/8
g	.111	=	7/8
h	.0101	=	5/16
i	.1011	=	11/16
k	.1101	=	13/16

L'ordre induit sur les mots est appelé l'*ordre fractionnaire*.

Règle. *L'ordre infixe correspond à l'ordre fractionnaire sur le code de l'arbre.*

La programmation de ces parcours sera donnée au chapitre V.

3.4 Une borne inférieure pour les tris par comparaisons

Voici une application surprenante des arbres à l'analyse de complexité. Il existe de nombreux algorithmes de tri, certains dont la complexité dans le pire des cas est en $O(n^2)$ comme les tris par sélection, par insertion ou à bulles, d'autres en $O(n^{3/2})$ comme le tri Shell, et d'autres en $O(n \log n)$ comme le tri fusion ou le tri par tas, que nous verrons page 100. On peut se demander s'il est possible de trouver un algorithme de tri de complexité inférieure dans le pire des cas.

Avant de résoudre cette question, il faut bien préciser le modèle de calcul que l'on considère. Un *tri par comparaison* est un algorithme qui trie en n'utilisant que des comparaisons. On peut supposer que les éléments à trier sont deux-à-deux distincts. Le modèle utilisé pour représenter un calcul est un *arbre de décision*. Chaque comparaison entre éléments d'une séquence à trier est représentée par un nœud interne de l'arbre. Chaque nœud pose une question. Le fils gauche correspond à une réponse négative, le fils droit à une réponse positive (figure 14).

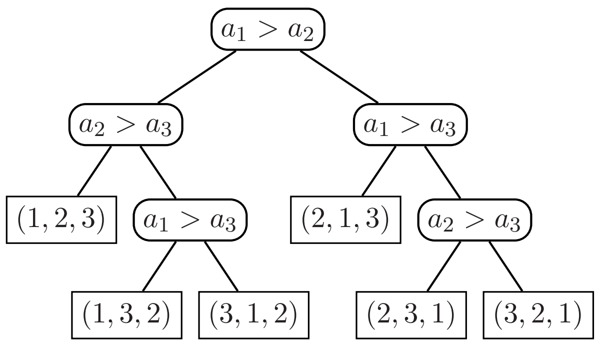


FIG. 14 – Exemple d'arbre de décision pour le tri.

Les feuilles représentent les permutations des éléments à effectuer pour obtenir la séquence triée. Le nombre de comparaisons à effectuer pour déterminer cette permutation est égale à la longueur du chemin de la racine à la feuille.

Nous prouvons ici :

Théorème 6 Tout algorithme de tri par comparaison effectue $\Omega(n \log n)$ comparaisons dans le pire des cas pour trier une suite de n éléments.

Preuve. Tout arbre de décision pour trier n éléments a $n!$ feuilles, représentant toutes les permutations possibles. La hauteur de l'arbre est donc minorée par $\log(n!)$. Or $\log(n!) = O(n \log n)$ par la formule de Stirling. □

Deux précisions pour clore cette parenthèse sur les tris. Tout d'abord, le résultat précédent n'est plus garanti si l'on change de modèle. Supposons par exemple que l'on veuille classer les notes (des entiers entre 0 et 20) provenant d'un paquet de 400 copies. La façon la plus simple et la plus efficace consiste à utiliser un tableau `T` de taille 21, dont chaque entrée `T[i]` sert à compter les notes égales à i. Il suffit alors de lire les notes une par une et d'incrémenter le compteur correspondant. Une fois ce travail accompli, le tri est terminé : il y a `T[0]` notes égales à 0, suivi de `T[1]` notes égales à 1, etc. Cet algorithme est manifestement linéaire et ne fait aucune comparaison ! Pourtant, il ne contredit pas notre résultat. Nous avons en effet utilisé implicitement une information supplémentaire : toutes les valeurs à trier appartiennent à l'intervalle $[0, 20]$. Cet exemple montre qu'il faut bien réfléchir aux conditions particulières avant de choisir un algorithme.

Seconde remarque, on constate expérimentalement que l'algorithme de tri rapide (QuickSort), dont la complexité dans le pire des cas est en $O(n^2)$, est le plus efficace en pratique. Comment est-ce possible ? Tout simplement parce que notre résultat ne concerne que la complexité dans le pire des cas. Or QuickSort est un algorithme en $O(n \log n)$ en moyenne.

4 Arbres de syntaxe abstraite

4.1 Les expressions sont des arbres

Considérons une définition des *expressions arithmétiques* avec un œil neuf. Une expression arithmétique e est :

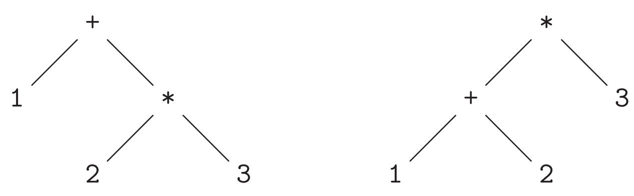
- un entier,
- ou bien une opération e_1 `op` e_2, où e_1 et e_2 sont des expressions arithmétiques et `op` est un opérateur (`+`, `-`, `*` et `/`).

L'œil neuf ne voit pas cette définition comme celle de l'écriture usuelle (notation infixe) des expressions, et d'ailleurs il manque les parenthèses. Il voit une définition inductive, l'ensemble des expressions est solution de cette équation récursive :

$$E = \mathbb{Z} \cup (E, \texttt{+}, E) \cup (E, \texttt{-}, E) \cup (E, \texttt{*}, E) \cup (E, \texttt{/}, E)$$

Cette définition inductive est une définition d'arbre, les expressions sont des feuilles qui contiennent un entier ou des nœuds internes à deux fils. Voir une expression comme un arbre évite toutes les ambiguïtés de la notation infixe. Par exemple, les deux arbres de la figure 15 disent clairement quels sont les arguments des opérations `+` et `*` dans les deux cas. Alors qu'en notation infixe,

Fig. 15 – Deux arbres de syntaxe abstraite



pour bien se faire comprendre, il faut écrire `1+(2*3)` et `(1+2)*3`.

Dès qu'un programme doit faire des choses un tant soit peu compliquées avec les expressions arithmétiques, il faut représenter ces expressions par des arbres de syntaxe abstraite. Le terme « abstraite » se justifie par opposition à la syntaxe « concrète » qui est l'écriture des expressions, c'est-à-dire ici la notation infixe. La production des arbres de syntaxes abstraite à partir de la syntaxe concrète est *l'analyse grammaticale* (*parsing*), une question cruciale qui est étudiée dans le cours suivant INF 431.

4.2 Implémentation des arbres de syntaxe abstraite

Écrivons une classe `Exp` des cellules d'arbre des expressions. Nous devons principalement distinguer cinq sortes de nœuds. Les entiers, qui sont des feuilles, et les quatre opérations, qui ont deux fils. La technique d'implémentation la plus simple est de réaliser tous ces nœuds par des objets d'une seule classe `Exp` qui ont tous les champs nécessaires, plus un champ `tag` qui indique la nature du nœud.[1] Le champ `tag` contient un entier censé être l'une de cinq constantes conventionnelles.

```
class Exp {
  final static int INT=0, ADD=1, SUB=2, MUL=3, DIV=4 ;
  int tag ;
// Utilisé si tag == INT
  int asInt ;
// Utilisés si tag ∈ {ADD, SUB, MUL, DIV}
  Exp e1, e2 ;

  Exp(int i) { tag = INT ; asInt = i ; }

  Exp(Exp e1, int op, Exp e2) {
    tag = op ; this.e1 = e1 ; this.e2 = e2 ;
  }
}
```

Ainsi pour construire l'arbre de gauche de la figure 15, on écrit :

```
new Exp
  (new Exp(1),
   ADD,
   new Exp (new Exp(2), MUL, new Exp(3)))
```

C'est non seulement assez lourd, mais aussi source d'erreurs. On atteint ici la limite de ce qu'autorise la surcharge des constructeurs. Il est plus commode de définir cinq méthodes statiques pour construire les divers nœuds.

```
static Exp mkInt(int i) { return new Exp (i) ; }

static Exp add(Exp e1, Exp e2) { return new Exp (e1, ADD, e2) ; }
  ⋮
static Exp div(Exp e1, Exp e2) { return new Exp (e1, DIV, e2) ; }
```

Et l'expression déjà vue, se construit par :

```
add(mkInt(1), mul(mkInt(2), mkInt(3)))
```

Ce qui est plus concis, sinon plus clair.

Un exemple d'opération « compliquée » sur les expressions arithmétiques est le calcul de leur valeur. L'opération n'est compliquée que si nous essayons de l'effectuer directement sur les notations infixes, car sur un arbre `Exp` c'est très facile.

[1] Une technique plus élégante à base d'héritage des objets est possible.

```
static int calc(Exp e) {
  switch (e.tag) {
    case INT: return e.asInt ;
    case ADD: return calc(e.e1) + calc(e.e2) ;
    case SUB: return calc(e.e1) - calc(e.e2) ;
    case MUL: return calc(e.e1) * calc(e.e2) ;
    case DIV: return calc(e.e1) / calc(e.e2) ;
  }
  throw new Error ("calc : arbre Exp incorrect") ;
}
```

L'instruction **throw** finale est nécessaire, car le compilateur n'a pas de moyen de savoir que le champ `tag` contient obligatoirement l'une des cinq constantes conventionnelles. En son absence, le programme est rejeté par le compilateur. Pour satisfaire le compilateur, on aurait aussi pu renvoyer une valeur « bidon » par **return** `0`, mais c'est nettement moins conseillé. Une erreur est une erreur, en cas d'arbre incorrect, mieux vaut tout arrêter que de faire semblant de rien.

Dans cet exemple typique, il faut surtout remarquer le lien très fort entre la définition inductive de l'arbre et la structure récursive de la méthode. La programmation sur les arbres de syntaxe abstraite est naturellement récursive.

4.3 Traduction de la notation postfixe vers la notation infixe

Nous avons déjà traité cette question de façon incomplète, en ne produisant que des notations infixes complètement parenthésées (exercice II.2). Nous pouvons maintenant faire mieux.

L'idée est d'abord d'interpréter la notation postfixe comme un arbre, puis d'afficher cet arbre, en tenant compte des règles usuelles qui permettent de ne pas mettre toutes les parenthèses. Pour la première opération il ne faut se poser aucune question, nous reprenons le calcul des expressions données en notation postfixe (voir II.1.2), en construisant un arbre au lieu de calculer une valeur. Nous avons donc besoin d'une pile d'arbres, ce qui est facile avec la classe des piles de la bibliothèque (voir II.2.3).

```
static Exp postfixToExp(String [] arg) {
  Stack<Exp> stack = new Stack<Exp> () ;
  for (int k = 0 ; k < arg.length ; k++) {
    Exp e1, e2 ;
    String cmd = arg[k] ;
    if (cmd.equals("+")) {
      e2 = stack.pop() ; e1 = stack.pop() ;
      stack.push(add(e1,e2)) ;
    } else if (cmd.equals("-")) {
      e2 = stack.pop() ; e1 = stack.pop() ;
      stack.push(sub(e1,e2)) ;
    } else if (cmd.equals("*")) {
      e2 = stack.pop() ; e1 = stack.pop() ;
      stack.push(mul(e1,e2)) ;
    } else if (cmd.equals("/")) {
      e2 = stack.pop() ; e1 = stack.pop() ;
      stack.push(div(e1,e2)) ;
    } else {
      stack.push(mkInt(Integer.parseInt(arg[k]))) ;
    }
  }
  return stack.pop() ;
}
```

Examinons la question d'afficher un arbre **Exp** sous forme infixe sans abuser des parenthèses. Tout d'abord, les parenthèses autour d'un entier ne sont jamais utiles. Ensuite, on distingue deux classes d'opérateurs, les additifs (`+` et `-`) et les multiplicatifs (`*` et `/`), les opérateurs d'une classe donnée ont le même comportement vis à vis du parenthésage. Il y a cinq positions possibles : au sommet de l'arbre, et à gauche ou à droite d'un opérateur additif ou multiplicatif. On examine ensuite l'éventuel parenthésage d'un opérateur.

- L'application des opérateurs additifs doit être parenthésée quand elle apparaît comme second argument d'un opérateur additif (`1-2+3` s'interprète comme `(1-2)+3`, il faut donc parenthéser `1-(2+3)`), ou comme argument d'un opérateur multiplicatif (considérer `(1+2)*3` et `1*(2+3)`).
- L'application des opérateurs multiplicatifs doit être parenthésée à droite des opérateurs multiplicatifs (même raisonnement que pour les additifs).

Ceci nous conduit à regrouper les positions possible en trois classes

(1) Sommet de l'arbre et à gauche des additifs : ne rien parenthéser.

(2) À droite des additifs et à gauche des multiplicatifs : ne parenthéser que les additifs.

(3) À droite des multiplicatifs : parenthéser tous les opérateurs.

On identifie les trois classes par 1, 2 et 3. On voit alors que les additifs sont à parenthéser pour les classes strictement supérieures à 1, et les multiplicatifs pour les classes strictement supérieures à 2. Ce qui conduit directement à la méthode suivante qui prend en dernier argument un entier `lvl` qui rend compte de la position de l'arbre `e` à afficher dans la sortie `out`.

```
static void expToInfix(PrintWriter out, Exp e, int lvl) {
  switch (e.tag) {
  case INT:
    out.print(e.asInt) ; return ;
  case ADD: case SUB:
    if (lvl > 1) out.print('(') ;
    expToInfix(out, e.e1, 1) ;
    out.print(e.tag == ADD ? '+' : '-') ;
    expToInfix(out, e.e2, 2) ;
    if (lvl > 1) out.print(')') ;
    return ;
  case MUL: case DIV:
    if (lvl > 2) out.print('(') ;
    expToInfix(out, e.e1, 2) ;
    out.print(e.tag == MUL ? '*' : '/') ;
    expToInfix(out, e.e2, 3) ;
    if (lvl > 2) out.print(')') ;
    return ;
  }
  throw new Error ("expToInfix : arbre Exp incorrect") ;
}
```

La méthode `expToInfix` mélange récursion et affichage. Cela ne pose pas de difficulté particulière : pour afficher une opération il faut d'abord afficher le premier argument (récursion) puis l'opérateur et enfin le second argument (récursion encore).

La sortie est un **PrintWriter** qui possède une méthode `print` exactement comme **System**`.out` mais est bien plus efficace (voir B.5.5.1). Le code utilise une particularité de l'instruction **switch** : on peut grouper les cas (ici des additifs et des multiplicatifs). Pour afficher l'opérateur, on a recours à l'expression conditionnelle (voir B.7.2). Par exemple `e.tag == ADD ? '+' : '-'` vaut `'+'` si `e.tag` est égal à `ADD` et `'-'` autrement — et ici « autrement » signifie nécessairement que `e.tag` est égal à `SUB` puisque nous sommes dans un cas regroupé du **switch** ne concernant que `ADD` et `SUB`.

Voici finalement la méthode `main` de la classe **Exp** qui appelle l'affichage infixe sur l'arbre construit en lisant la notation postfixe

```
public static void main (String [] arg) {
  PrintWriter out = new PrintWriter (System.out) ;
  Exp e = postfixToExp(arg) ;
  expToInfix(out, e, 1) ;
  out.println() ; out.flush() ;
}
```

Les **PrintWriter** sont bufferisés, il faut vider le tampon par `out.flush()` avant de finir, voir B.5.4. Reprenons l'exemple de la figure II.4.

```
% java Exp 6 3 2 - 1 + / 9 6 - '*'
6/(3-2+1)*(9-6)
```

Ce qui est meilleur que l'affichage `((6/((3-2)+1))*(9-6))` de l'exercice II.2.

5 Files de priorité

Nous avons déjà rencontré les files d'attente. Les files de priorité sont des files d'attente où les éléments ont un niveau de priorité. Le passage devant le guichet, ou le traitement de l'élément, se fait en fonction de son niveau de priorité. L'implantation d'une file de priorité est un exemple d'utilisation d'arbre, et c'est pourquoi elle trouve naturellement sa place ici.

De manière plus formelle, une *file de priorité* est un type abstrait de données opérant sur un ensemble ordonné, et muni des opérations suivantes :

- trouver le plus grand élément
- insérer un élément
- retirer le plus grand élément

Bien sûr, on peut remplacer « le plus grand élément » par « le plus petit élément » en prenant l'ordre opposé. Plusieurs implantations d'une file de priorité sont envisageables : par tableau ou par liste, ordonnés ou non. Nous allons utiliser des *tas*. La table 1 présente la complexité des opérations des files de priorités selon la structure de données choisie.

Implantation	Trouver max	Insérer	Retirer max
Tableau non ordonné	$O(n)$	$O(1)$	$O(n)$
Liste non ordonnée	$O(n)$	$O(1)$	$O(1)$[a]
Tableau ordonné	$O(1)$	$O(n)$	$O(1)$
Liste ordonnée	$O(1)$	$O(n)$	$O(1)$
Tas	$O(1)$	$O(\log n)$	$O(\log n)$

[a] Dans cette table, le coût de la suppression dans une liste non ordonnée est calculé en supposant l'élément déjà trouvé.

TAB. 1 – Complexité des implantations de files de priorité.

5.1 Tas

Un arbre binaire est *tassé* si son code est un segment initial pour l'ordre des mots croisés. En d'autres termes, dans un tel arbre, tous les niveaux sont entièrement remplis à l'exception peut-être du dernier niveau, et ce dernier niveau est rempli « à gauche ». La figure 16 montre un arbre tassé.

Proposition 7 La hauteur d'un arbre tassé à n nœuds est $\lfloor \log_2 n \rfloor$.

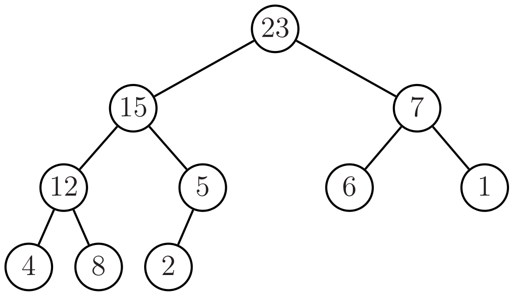


FIG. 16 – Un arbre tassé.

Un *tas* (en anglais « heap ») est un arbre binaire tassé tel que le contenu de chaque nœud soit supérieur ou égal à celui de ses fils. Ceci entraîne, par transitivité, que le contenu de chaque nœud est supérieur ou égal à celui de ses descendants.

L'arbre de la figure 16 est un tas.

5.2 Implantation d'un tas

Un tas s'implante facilement à l'aide d'un simple tableau. Les nœuds d'un arbre tassé sont numérotés en largeur, de gauche à droite. Ces numéros sont des indices dans un tableau (cf figure 17).

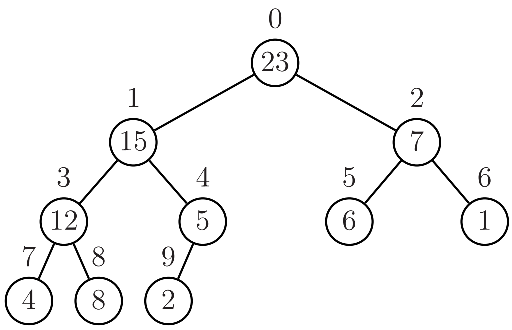


FIG. 17 – Un arbre tassé, avec la numérotation de ses nœuds.

L'élément d'indice i du tableau est le contenu du nœud de numéro i. Dans notre exemple, le tableau est :

i	0	1	2	3	4	5	6	7	8	9
a_i	23	15	7	12	5	6	1	4	8	2

Le fait que l'arbre soit tassé conduit à un calcul très simple des relations de filiation dans l'arbre (n est le nombre de ses nœuds) :

racine	:	nœud 0
parent du nœud i	:	nœud $\lfloor (i-1)/2 \rfloor$
fils gauche du nœud i	:	nœud $2i+1$
fils droit du nœud i	:	nœud $2i+2$
nœud i est une feuille	:	$2i+1 \geqslant n$
nœud i a un fils droit	:	$2i+2 < n$

L'*insertion* d'un nouvel élément v se fait en deux temps : d'abord, l'élément est ajouté comme contenu d'un nouveau nœud à la fin du dernier niveau de l'arbre, pour que l'arbre reste tassé. Ensuite, le contenu de ce nœud, soit v, est comparé au contenu de son père. Tant que le contenu du père est plus petit que v, le contenu du père est descendu vers le fils. À la fin, on remplace par v le dernier contenu abaissé (voir figure 18).

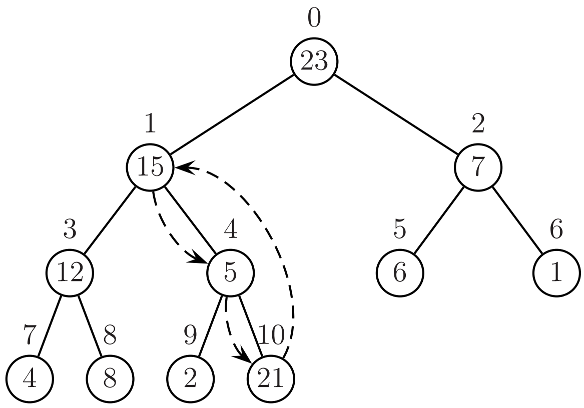


FIG. 18 – Un tas, avec remontée de la valeur 21 après insertion.

La *suppression* se fait de manière similaire. D'abord, le contenu du nœud le plus à droite du dernier niveau est transféré vers la racine, et ce nœud est supprimé. Ceci garantit que l'arbre reste tassé. Ensuite, le contenu v de la racine est comparé à la plus grande des valeurs de ses fils (s'il en a). Si cette valeur est supérieure à v, elle est remontée et remplace le contenu du père. On continue ensuite avec le fils. Par exemple, la suppression de 16 dans l'arbre de gauche de la figure 19 conduit d'abord à l'arbre de droite de cette figure, et enfin au tas de la figure 20

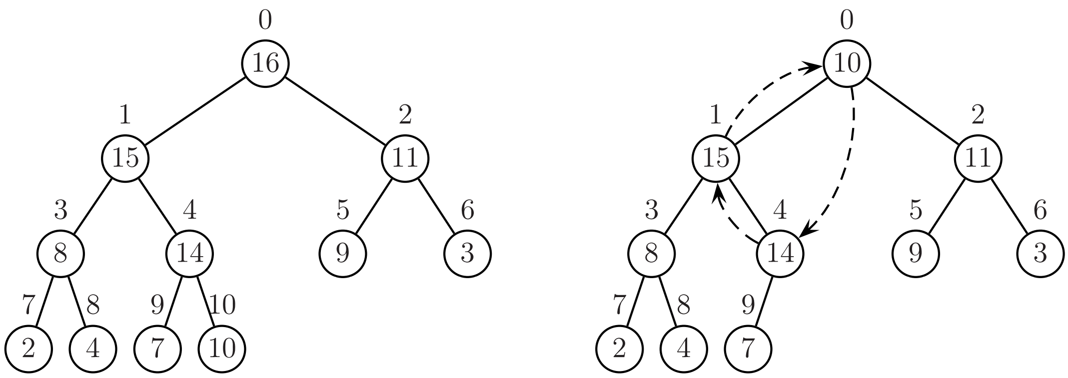


FIG. 19 – Un tas, et la circulation des valeurs pendant la suppression de 16.

La complexité de chacune de ces opérations est majorée par la hauteur de l'arbre qui est, elle, logarithmique en la taille.

Un tas est naturellement présenté comme une classe, fournissant les trois méthodes maximum(), inserer(), supprimer(). On range les données dans un tableau interne. Un constructeur permet de faire l'initialisation nécessaire. Voici le squelette :

```
class Tas
{
  int[] a;
  int nTas = 0;
```

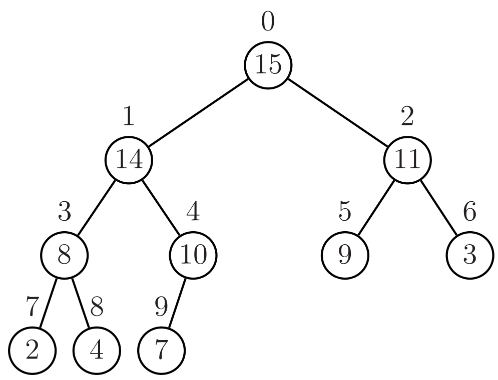


FIG. 20 – Le tas de la figure 19 après suppression de 16.

```
Tas(int n)
{
  nTas = 0;
  a = new int[n];
}

int maximum()
{
  return a[0];
}

void ajouter(int v) {...}
void supprimer() {...}
```

Avant de donner l'implantation finale, voici une première version à l'aide de méthodes qui reflètent les opérations de base.

```
void ajouter(int v)
{
  int i = nTas;
  ++nTas;
  while (!estRacine(i) && cle(parent(i)) < v)
  {
    cle(i) = cle(parent(i));
    i = parent(i);
  }
  cle(i) = v;
}
```

De même pour la suppression :

```
void supprimer()
{
  --nTas;
  cle(0) = cle(nTas);
  int v = cle(0);
  int i = 0;
  while (!estFeuille(i))
  {
    int j = filsG(i);
    if (existeFilsD(i) && cle(filsD(i)) > cle(filsG(i)))
```

```
      j = filsD(i);
    if (v >= cle(j)) break;
    cle(i) = cle(j);
    i = j;
  }
  cle(i) = v;
}
```

Il ne reste plus qu'à remplacer ce pseudo-code par les instructions opérant directement sur le tableau.

```
void ajouter(int v)
{
  int i = nTas;
  ++nTas;
  while (i > 0 && a[(i-1)/2] <= v)
  {
    a[i] = a[(i-1)/2];
    i = (i-1)/2;
  }
  a[i] = v;
}
```

On notera que, puisque la hauteur d'un tas à n nœuds est $\lfloor \log_2 n \rfloor$, le nombre de comparaisons utilisée par la méthode ajouter est en $O(\log n)$.

```
  void supprimer()
  {
    int v = a[0] = a[--nTas];
    int i = 0;
    while (2*i + 1 < nTas)
    {
      int j = 2*i + 1;
      if (j +1 < nTas && a[j+1] > a[j])
        ++j;
      if (v >= a[j])
        break;
      a[i] = a[j];
      i = j;
    }
    a[i] = v;
  }
}
```

Là encore, la complexité de la méthode supprimer est en $O(\log n)$.

On peut se servir d'un tas pour trier : on insère les éléments à trier dans le tas, puis on les extrait un par un. Ceci donne une méthode de tri appelée *tri par tas* (« heapsort » en anglais).

```
static int[] triParTas(int[] a)
{
  int n = a.length;
  Tas t = new Tas(n);
  for (int i = 0; i < n; i++)
    t.ajouter(a[i]);
  for (int i = n - 1; i >= 0; --i)
```

```
    {
      int v = t.maximum();
      t.supprimer();
      a[i] = v;
    }
    return a;
  }
```

La complexité de ce tri est, dans le pire des cas, en $O(n \log n)$. En effet, on appelle n fois chacune des méthodes ajouter et supprimer.

5.3 Arbres de sélection

Une variante des arbres tassés sert à la *fusion* de listes ordonnées. On est en présence de k suites de nombres ordonnées de façon décroissante (ou croissante), et on veut les fusionner en une seule suite croissante. Cette situation se présente par exemple lorsqu'on veut fusionner des données provenant de capteurs différents.

Un algorithme « naïf » opère de manière la suivante. À chaque étape, on considère le premier élément de chacune des k listes (c'est le plus grand dans chaque liste), puis on cherche le maximum de ces nombres. Ce nombre est inséré dans la liste résultat, et supprimé de la liste dont il provient. La recherche du maximum de k éléments coûte $k - 1$ comparaisons. Si la somme des longueurs des k listes de données est n, l'algorithme naïf est donc de complexité $O(nk)$.

Il semble plausible que l'on puisse gagner du temps en « mémorisant » les comparaisons que l'on a faites lors du calcul d'un maximum. En effet, lorsque l'on calcule le maximum suivant, seule une donnée sur les k données comparées a changé. L'ordre des $k-1$ autres éléments reste le même, et on peut économiser des comparaisons si l'on connaît cet ordre au moins partiellement. La solution que nous proposons est basée sur un tas qui mémorise partiellement l'ordre. On verra que l'on peut effectuer la fusion en temps $O(k + n \log k)$. Le premier terme correspond au « prétraitement », c'est-à-dire à la mise en place de la structure particulière.

L'*arbre de sélection* est un arbre tassé à k feuilles. Chaque feuille est en fait une liste, l'une des k listes à fusionner (voir figure 21). La hauteur de l'arbre est $\log k$. Chaque nœud de l'arbre

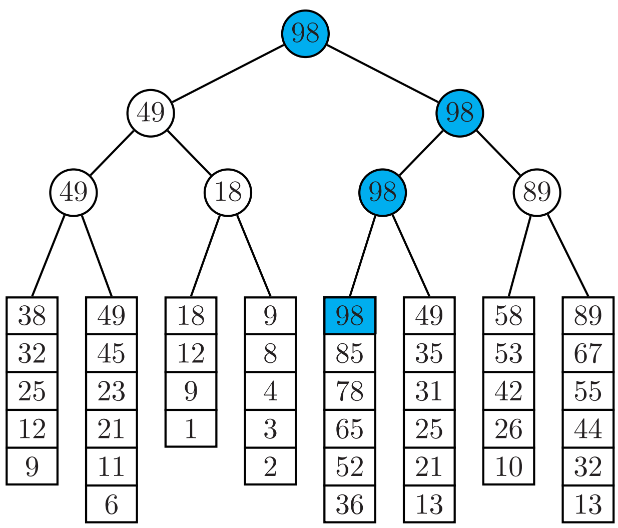


FIG. 21 – Un arbre de sélection pour 8 listes.

contient, comme clé, le maximum des clés de ses deux fils (ou le plus grand élément d'une liste).

En particulier, le maximum des éléments en tête des k listes (en grisé sur la figure 21) se trouve $\log k$ fois dans l'arbre, sur les nœuds d'un chemin menant à la racine. L'extraction d'un plus grand élément se fait donc en temps constant. Cette extraction est suivie d'un mise-à-jour : En descendant le chemin dont les clés portent le maximum, on aboutit à la liste dont la valeur de tête est ce maximum. L'élément est remplacé, dans la liste, par son suivant. Ensuite, on remonte vers la racine en recalculant, pour chaque nœud rencontré, le valeur de la clé, avec la nouvelle valeur du fils mis-à-jour. À la racine, on trouve le nouveau maximum (voir figure 22). La mise-à-jour se

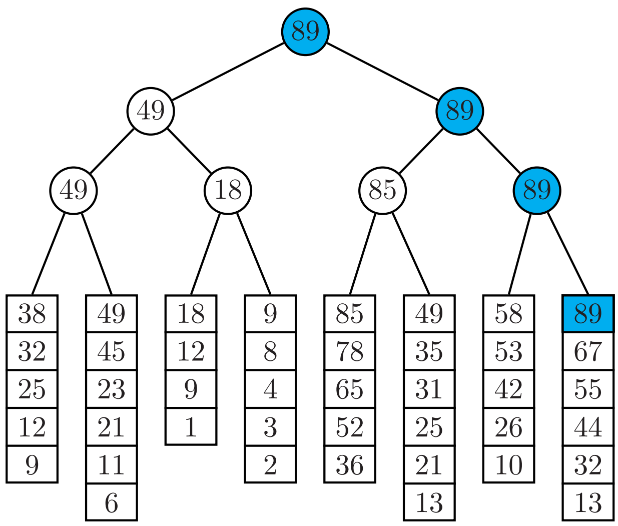


FIG. 22 – Après extraction du plus grand élément et recalcul.

fait donc en $O(\log k)$ opérations. La mise en place initiale de l'arbre est en temps k.

6 Codage de Huffman

6.1 Compression des données

La compression des données est un problème algorithmique aussi important que le tri. On distingue la compression sans perte, où les données décompressées sont identiques aux données de départ, et la compression avec perte. La compression sans perte est importante par exemple dans la compression de textes, de données scientifiques ou du code compilé de programmes. En revanche, pour la compression d'images et de sons, une perte peut être acceptée si elle n'est pas perceptible, ou si elle est automatiquement corrigée par le récepteur.

Parmi les algorithmes de compression sans perte, on distingue plusieurs catégories : les algorithmes statistiques codent les caractères fréquents par des codes courts, et les caractères rares par des codes longs ; les algorithmes basés sur les dictionnaires enregistrent toutes les chaînes de caractères trouvées dans une table, et si une chaîne apparaît une deuxième fois, elle est remplacée par son indice dans la table. L'algorithme de Huffman est un algorithme statistique. L'algorithme de Ziv-Lempel est basé sur un dictionnaire. Enfin, il existe des algorithmes « numériques ». Le codage arithmétique remplace un texte par un nombre réel entre 0 et 1 dont les chiffres en écriture binaire peuvent être calculés au fur et à mesure du codage. Le codage arithmétique repose aussi sur la fréquence des lettres. Ces algorithmes seront étudiés dans le cours 431 ou en majeure.

6.2 Algorithme de Huffman

L'algorithme de Huffman est un algorithme statistique. Les caractères du texte clair sont codés par des chaînes de bits. Le choix de ces codes se fait d'une part en fonction des fréquences d'apparition de chaque lettre, de sorte que les lettres fréquentes aient des codes courts, mais aussi de façon à rendre le décodage facile. Pour cela, on choisit un code préfixe, au sens décrit ci-dessous. La version du codage de Huffman que nous détaillons dans cette section est le codage dit « statique » : les fréquences n'évoluent pas au cours de la compression, et le code reste fixe. C'est ce qui se passe dans les modems, ou dans les fax.

Une version plus sophistiquée est le codage dit « adaptatif », présenté brièvement dans la section 6.3. Dans ce cas, les fréquences sont mises à jour après chaque compression de caractère, pour chaque fois optimiser le codage.

6.2.1 Codes préfixes et arbres

Un ensemble P de mots non vides est un *code préfixe* si aucun des mots de P n'est préfixe propre d'un autre mot de P.

Par exemple, l'ensemble $\{0, 100, 101, 111, 1100, 1101\}$ est un code préfixe. Un code préfixe P est *complet* si tout mot est préfixe d'un produit de mots de P. Le code de l'exemple ci-dessus est complet. Pour illustrer ce fait, prenons le mot 1011010101011110110011 : il est préfixe du produit

$$101 \cdot 101 \cdot 0 \cdot 101 \cdot 0 \cdot 111 \cdot 10 \cdot 1100 \cdot 11\mathit{1}$$

L'intérêt de ces définitions vient des propositions suivantes.

Proposition 8 Un ensemble fini de mots sur $\{0, 1\}$ est un code préfixe si et seulement s'il est le code des feuilles d'un arbre binaire.

Rappelons qu'un arbre binaire est dit *complet* si tous ses nœuds internes ont arité 2.

Proposition 9 Un ensemble fini de mots sur $\{0, 1\}$ est un code préfixe complet si et seulement s'il est le code des feuilles d'un arbre binaire complet.

La figure 23 reprend l'arbre de la figure 12. Le code des feuilles est préfixe, mais il n'est pas complet.

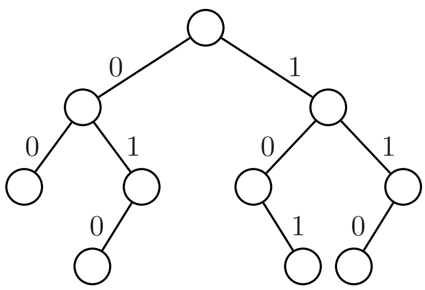


Fig. 23 – Le code des feuilles est $\{00, 010, 101, 110\}$.

En revanche, le code des feuilles de l'arbre de la figure 24 est complet.

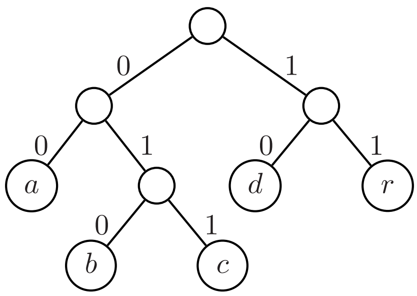


FIG. 24 – Le code des feuilles est $\{00, 010, 011, 10, 11\}$.

Le codage d'un texte par un code préfixe (complet) se fait de la manière suivante : à chaque lettre est associée une feuille de l'arbre. Le code de la lettre est le code de la feuille, c'est-à-dire l'étiquette du chemin de la racine à la feuille. Dans le cas de l'arbre de la figure 24, on associe les lettres aux feuilles de la gauche vers la droite, ce qui donne le tableau de codage suivant :

a	$\rightarrow$	00
b	$\rightarrow$	010
c	$\rightarrow$	011
d	$\rightarrow$	10
r	$\rightarrow$	11

Le codage consiste simplement à remplacer chaque lettre par son code. Ainsi, *abracadabra* devient 0001011000110010000010110. Le fait que le code soit préfixe permet un décodage instantané : il suffit d'entrer la chaîne caractère par caractère dans l'arbre, et de se laisser guider par les étiquettes. Lorsque l'on est dans une feuille, on y trouve le caractère correspondant, et on recommence à la racine. Quand le code est complet, on est sûr de pouvoir toujours décoder un message, éventuellement à un reste près qui est un préfixe d'un mot du code.

Le problème qui se pose est de minimiser la taille du texte codé. Avec le code donné dans le tableau ci-dessus, le texte *abracadabra*, une fois codé, est composé de 25 bits. Si l'on choisit un autre codage, comme par exemple

a	$\rightarrow$	0
b	$\rightarrow$	10
c	$\rightarrow$	1101
d	$\rightarrow$	1100
r	$\rightarrow$	111

on observe que le même mot se code par 01011101101011000101110 et donc avec 23 bits. Bien sûr, la taille du résultat ne dépend que de la fréquence d'apparition de chaque lettre dans le texte source. Ici, il y a 5 lettres a, 2 lettres b et r, et 1 fois la lettre c et d.

6.2.2 Construction de l'arbre

L'algorithme de Huffman construit un arbre binaire complet qui donne un code optimal, en ce sens que la taille du code produit est minimale parmi la taille de tous les codes produits à l'aide de codes préfixes complets.

Initialisation On construit une forêt d'arbres binaires formés chacun d'une seul feuille. Chaque feuille correspond à une lettre du texte, et a pour valeur la fréquence d'apparition de la lettre dans le texte.

Itération On fusionne deux des arbres dont la fréquence est minimale. Le nouvel arbre a pour fréquence la somme des fréquences de ses deux sous-arbres.

Arrêt On termine quand il ne reste plus qu'un seul arbre qui est l'arbre résultat.

La fréquence d'un arbre est, bien sûr, la somme des fréquences de ses feuilles. Voici une illustration de cet algorithme sur le mot *abracadabra*. La première étape conduit à la création des 5 arbres (feuilles) de la figure 25. Les feuilles des lettres c et d sont fusionnées (figure 26), puis cet arbre avec la feuille b (figure 27), etc. Le résultat est représenté dans la figure 29.

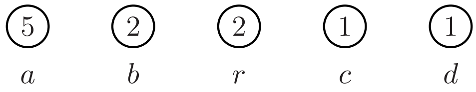


FIG. 25 – Les 5 arbres réduits chacun à une feuille.

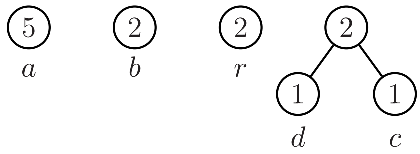


FIG. 26 – Les feuilles des lettres c et d sont fusionnées.

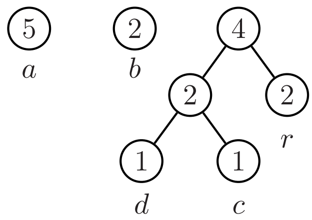


FIG. 27 – L'arbre est fusionné avec la feuille de r.

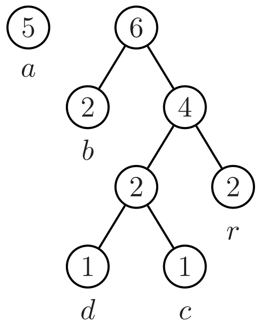


FIG. 28 – La feuille de b est fusionnée avec l'arbre.

Notons qu'il y a beaucoup de choix dans l'algorithme : d'une part, on peut choisir lequel des deux arbres devient sous-arbre gauche ou droit. Ensuite, les deux sous-arbres de fréquence minimale ne sont peut-être pas uniques, et là encore, il y a des choix pour la fusion. On peut prouver que cet algorithme donne un code optimal. Il en existe de nombreuses variantes. L'une d'elle consiste à grouper les lettres par deux (digrammes) ou même par bloc plus grands, notamment s'il s'agit de données binaires. Les techniques de compression avec dictionnaire (compression de Ziv-Lempel) ou le codage arithmétique donnent en général de meilleurs taux de compression.

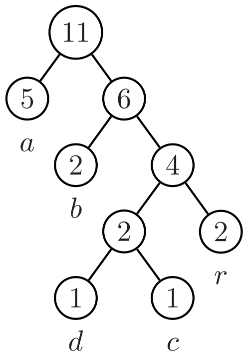


FIG. 29 – L'arbre après la dernière fusion.

6.2.3 Choix de la représentation des données

Si le nombre de lettres figurant dans le texte est n, l'arbre de Huffman est de taille $2n-1$. On le représente ici par un tableau pere, où, pour $x \geqslant 0$ et $y > 0$ pere[x] = y si x est fils droit de y et pere[x] = -y si x est fils gauche de y. Chaque caractère c a une fréquence d'apparition dans le texte, notée freq[c]. Seuls les caractères qui figurent dans le texte, i.e. de fréquence non nulle, vont être représentés dans l'arbre.

La création de l'arbre est contrôlée par un tas (mais un « tas-min », c'est-à-dire un tas avec extraction du minimum). La clé de la sélection est la fréquence des lettres et la fréquence cumulée dans les arbres. Une autre méthode rencontrée est l'emploi de deux files.

Cette représentation des données est bien adaptée au problème considéré, mais est assez éloignée des représentations d'arbres que nous verrons dans le chapitre suivant. Réaliser une implantation à l'aide de cette deuxième représentation est un bon exercice.

6.2.4 Implantation

La classe Huffman a deux données statiques : le tableau pere pour représenter l'arbre, et un tableau freq qui contient les fréquences des lettres dans le texte, et aussi les fréquences cumulées dans les arbres. On part du principe que les 26 lettres de l'alphabet peuvent figurer dans le texte (dans la pratique, on prendra plutôt un alphabet de 128 ou de 256 caractères).

```
class Huffman
{
  final static int N = 26, M = 2*N - 1; // nombre de caractères
  static int[] pere = new int[M];
  static int[] freq  = new int[M];

  public static void main(String[] args)
  {
    String s = args[0];
    calculFrequences(s);
    creerArbre();
    String[] table = faireTable();
    afficherCode(s,table);
    System.out.println();
  }
}
```

La méthode main décrit l'opération : on calcule les fréquences des lettres, et le nombre de lettres de fréquence non nulle. On crée ensuite un tas de taille appropriée, avec le tableau des fréquences comme clés. L'opération principale est creerArbre() qui crée l'arbre de Huffman. La table de codage est ensuite calculée et appliquée à la chaîne à compresser.

Voyons les diverses étapes. Le calcul des fréquences et du nombre de lettres de fréquence non nulle ne pose pas de problème. On notera toutefois l'utilisation de la méthode charAt de la classe String, qui donne l'unicode du caractère en position i. Comme on a supposé que l'alphabet était a–z et que les unicodes de ces caractères sont consécutifs, l'expression s.charAt(i) - 'a' donne bien le rang du i-ème caractère dans l'alphabet. Il faudrait bien sûr modifier cette méthode si on prenait un alphabet à 256 caractères.

```
static void calculFrequences(String s)
{
  for (int i = 0; i < s.length(); i++)
    freq[s.charAt(i) - 'a']++;
}

static int nombreLettres()
{
  int n = 0;
  for (int i = 0; i < N; i++)
    if (freq[i] > 0)
      n++;
  return n;
}
```

La méthode de création d'arbre utilise très exactement l'algorithme exposé plus haut : d'abord, les lettres sont insérées dans le tas ; ensuite, les deux éléments de fréquence minimale sont extraits, et un nouvel arbre, dont la fréquence est la somme des fréquences des deux arbres extraits, est inséré dans le tas. Comme il y a n feuilles dans l'arbre, il y a $n - 1$ créations de nœuds internes.

```
static void creerArbre()
{
  int n = nombreLettres();
  Tas tas = new Tas(2*n-1, freq);
  for (int i = 0; i < N; ++i)
    if (freq[i] >0)
      tas.ajouter(i);
  for (int i = N; i < N+n-1; ++i)
  {
    int x = tas.minimum();
    tas.supprimer();
    int y = tas.minimum();
    tas.supprimer();
    freq[i] = freq[x] + freq[y];
    pere[x] = -i;
    pere[y] = i;
    tas.ajouter(i);
  }
}
```

Le calcul de la table de codage se fait par un parcours d'arbre :

```
static String code(int i)
{
  if (pere[i] == 0) return "";
```

```
    return code(Math.abs(pere[i])) + ((pere[i] < 0) ? "0" : "1");
  }

  static String[] faireTable()
  {
    String[] table = new String[N];
    for (int i = 0; i < N; i++)
      if (freq[i] >0)
        table[i] = code(i);
    return table;
  }
```

Il reste à examiner la réalisation du « tas-min ». Il y a deux différences avec les tas déjà vus : d'une part, le maximum est remplacé par le minimum (ce qui est négligeable), et d'autre part, ce ne sont pas les valeurs des éléments eux-mêmes (les lettres du texte) qui sont utilisées comme comparaison, mais une valeur qui leur est associée (la fréquence de la lettre). Les méthodes des tas déjà vues se récrivent très facilement dans ce cadre.

```
class Tas
{
  int[] a;  // contient les caractères
  int nTas = 0;
  int[] freq; // contient les fréquences des caractères

  Tas(int taille, int[] freq)
  {
    this.freq = freq;
    nTas = 0;
    a = new int[taille];
  }

  int minimum()
  {
    return a[0];
  }

  void ajouter(int v)
  {
    int i = nTas;
    ++nTas;
    while (i > 0 && freq[a[(i-1)/2]] >= freq[v])
    {
      a[i] = a[(i-1)/2];
      i = (i-1)/2;
    }
    a[i] = v;
  }

  void supprimer()
  {
    int v = a[0] = a[--nTas];
    int i = 0;
    while (2*i + 1 < nTas)
    {
      int j = 2*i + 1;
```

```
        if (j +1 < nTas && freq[a[j+1]] < freq[a[j]])
          ++j;
        if (freq[v] <= freq[a[j]])
          break;
        a[i] = a[j];
        i = j;
      }
      a[i] = v;
    }
  }
```

Une fois la table du code calculée, encore faut-il la transmettre avec le texte comprimé, pour que le destinaire puisse décompresser le message. Transmettre la table telle quelle est redondant puisque l'on transmet tous les chemins de la racine vers les feuilles. Il est plus économique de faire un parcours préfixe de l'arbre, avec la valeur littérale des lettres représentées aux feuilles. Par exemple, pour l'arbre de la figure 29, on obtient la représentation `01[a]01[b]001[d]1[c]1[r]`. Une méthode plus simple est de transmettre la suite de fréquence, quitte au destinataire de reconstruire le code. Cette deuxième méthode admet un raffinement (qui montre jusqu'où peut aller la recherche de l'économie de place) qui consiste à non pas transmettre les valeurs exactes des fréquences, mais une séquence de fréquences fictives (à valeurs numériques plus petites, donc plus courtes à transmettre) qui donne le même code de Huffman. Par exemple, les deux arbres

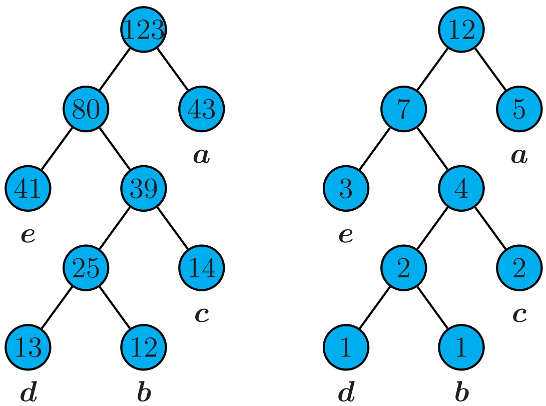


FIG. 30 – Deux suites de fréquences qui produisent le même arbre.

de la figure 30 ont des fréquences différentes, mais le même code. Il est bien plus économique de transmettre la suite de nombres $5, 1, 2, 1, 3$ que la suite initiale $43, 13, 12, 14, 41$.

6.3 Algorithme de Huffman adaptatif

Les inconvénients de la méthode de compression de Huffman sont connus :

- Il faut lire le texte entièrement avant de lancer la compression.
- Il faut aussi transmettre le code trouvé.

Une version adaptative de l'algorithme de Huffman corrige ces défauts. Le principe est le suivant :

- Au départ, toutes les lettres ont même fréquence (nulle) et le code est uniforme.
- Pour chaque lettre x, le code de x est envoyé, puis la fréquence de la lettre est augmentée et l'arbre de Huffman est recalculé.

Evidemment, le code d'une lettre change en cours de transmission : quand la fréquence (relative) d'une lettre augmente, la longueur de son code diminue. Le code d'une lettre s'adapte à sa fréquence. Le destinataire du message compressé mime le codage de l'expéditeur : il maintient

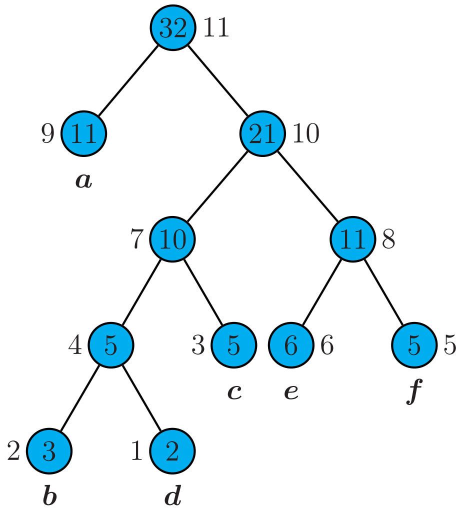


FIG. 31 – Un arbre de Huffman avec un parcours compatible.

un arbre de Huffman qu'il met à jour comme l'expéditeur, et il sait donc à tout moment quel est le code d'une lettre.

Les deux inconvénients du codage de Huffman (statique) sont corrigés par cette version. Il n'est pas nécessaire de calculer globalement les fréquences des lettres du texte puisqu'elles sont mises à jour à chaque pas. Il n'est pas non plus nécessaire de transmettre la table de codage puisqu'elle est recalculée par le destinataire.

Il existe par ailleurs une version « simplifiée » du codage de Huffman où les fréquences utilisées pour la construction de l'arbre ne sont pas les fréquences des lettres du texte à comprimer, mais les fréquences moyennes rencontrées dans les textes de la langue considérée. Bien entendu, le taux de compression s'en ressent si la distribution des lettres dans le texte à comprimer s'écarte de cette moyenne. C'est dans ce cas que l'algorithme adaptatif s'avère particulièrement utile.

Notons enfin que le principe de l'algorithme adaptatif s'applique à tous les algorithmes de compression statistiques. Il existe, par exemple, une version adaptative de l'algorithme de compression arithmétique.

L'algorithme adaptatif opère, comme l'algorithme statique, sur un arbre. Aux feuilles de l'arbre se trouvent les lettres, avec leurs fréquences. Aux nœuds de l'arbre sont stockées les fréquences cumulées, c'est-à-dire la somme des fréquences des feuilles du sous-arbre dont le nœud est la racine (voir figure 31). L'arbre de Huffman est de plus muni d'une liste de parcours (un ordre total) qui a les deux propriétés suivantes :

- le parcours est compatible avec les fréquences (les fréquences sont croissantes dans l'ordre du parcours),
- deux nœuds frères sont toujours consécutifs.

On démontre que, dans un arbre de Huffman, on peut toujours trouver un ordre de parcours possédant ces propriétés. Dans la figure 31, les nœuds sont numérotés dans un tel ordre. L'arbre de Huffman évolue avec chaque lettre codée, de la manière suivante. Soit x le caractère lu dans le texte clair. Il correspond à une feuille de l'arbre. Le code correspondant est envoyé, puis les deux opérations suivantes sont réalisées.

- Partant de la feuille du caractère, sa fréquence est incrémentée, et on remonte vers la racine en incrémentant les fréquences dans les nœuds rencontrés.
- Avant l'incrémentation, chaque nœud est permuté avec le *dernier nœud* (celui de plus grand numéro) de même fréquence dans l'ordre de parcours.

La deuxième opération garantit que l'ordre reste compatible avec les fréquences. Supposons que

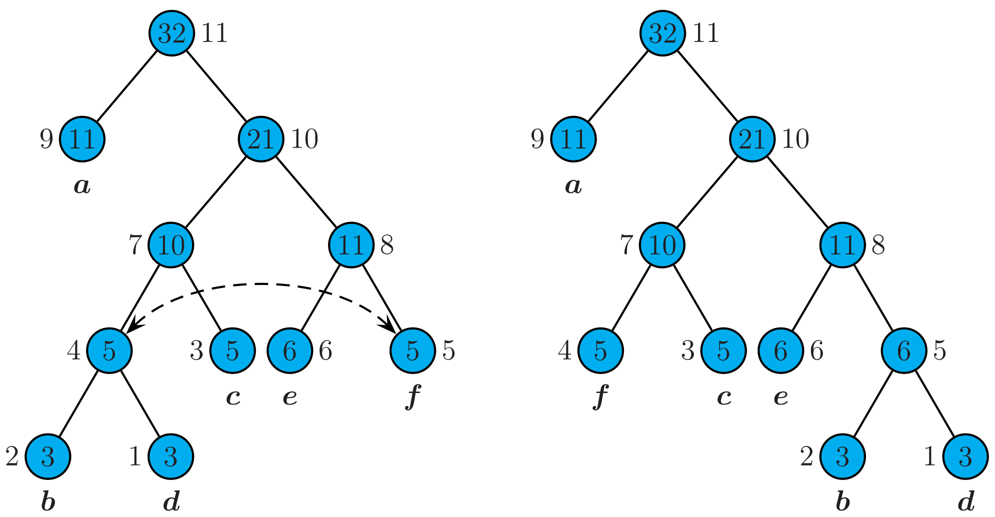


FIG. 32 – Après incrémentation de d (à gauche), et après permutation des nœuds (à droite).

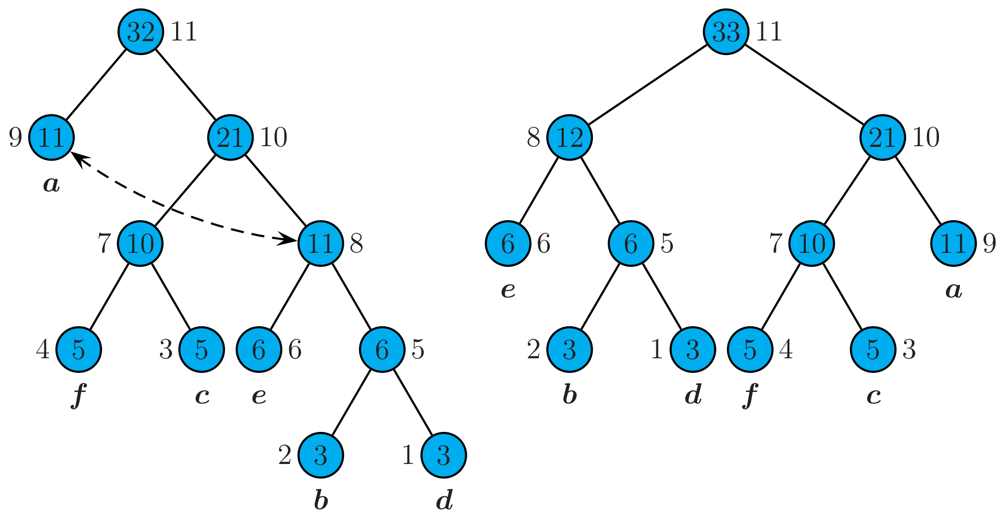


FIG. 33 – Avant permutation des nœuds 8 et 9 (à gauche) , et arbre final (à droite).

la lettre suivante du texte clair à coder, dans l'arbre de la figure 31, soit une lettre d. Le code émis est alors 1001. La fréquence de la feuille d est incrémentée. Le père de la feuille d a pour fréquence 5. Son numéro d'ordre est 4, et le plus grand nœud de fréquence 5 a numéro d'ordre 5. Les deux nœuds sont échangés (figure 32). De même, avant d'incrémenter le nœud de numéro 8, il est échangé avec le nœud de numéro 9, de même fréquence (figure 33). Ensuite, la racine est incrémentée.

Pour terminer, voici une table de quelques statistiques concernant l'algorithme de Huffman. Ces statistiques ont été obtenues à une époque où il y avait encore peu de données disponibles sous format électronique, et les contes de Grimm en faisaient partie... Comme on le voit, le taux de compression obtenu par les deux méthodes est sensiblement égal, car les données sont assez homogènes.

	Contes de Grimm	Texte technique
Taille (bits)	700 000	700 000
Huffman	439 613	518 361
Taille code	522	954
Total	440135	519315
Adaptatif	440164	519561

Si on utilise un codage par digrammes (groupe de deux lettres), les statistiques sont les suivantes :

	Contes de Grimm	Texte technique
Taille (bits)	700 000	700 000
Huffman	383 264	442 564
Taille code	10 880	31 488
Total	394 144	474 052
Adaptatif	393 969	472 534

On voit que le taux de compression est légèrement meilleur.

Chapitre V

Arbres binaires

Dans ce chapitre, nous traitons d'abord les arbres binaires de recherche, puis les arbres équilibrés.

1 Implantation des arbres binaires

Un arbre binaire qui n'est pas vide est formé d'un nœud, sa *racine*, et de deux sous-arbres binaires, l'un appelé le *fils gauche*, l'autre le *fils droit*. Nous nous intéressons aux arbres contenant des informations. Chaque nœud porte une information, appelée son *contenu*. Un arbre non vide est donc entièrement décrit par le triplet (fils gauche, contenu de la racine, fils droit). Cette définition récursive se traduit en une spécification de programmation. Il suffit de préciser la nature du contenu d'un nœud. Pour simplifier, nous supposons que le contenu est un entier. On obtient alors la définition.

```
class Arbre
{
  int contenu;
  Arbre filsG, filsD;

  Arbre(Arbre g, int c, Arbre d)
  {
    filsG = g;
    contenu = c;
    filsD = d;
  }
}
```

L'arbre vide est, comme d'habitude, représenté par `null`. Un arbre réduit à une feuille, de contenu `x`, est créé par

```
new Arbre(null, x, null)
```

L'arbre de la figure 1 est créé par

```
new Arbre(
  new Arbre(
    new Arbre(null, 3, null),
    5,
    new Arbre(
      new Arbre(
       new Arbre(null, 6, null),
       8,
```

```
          null)
        12,
        new Arbre(null, 13, null))),
    20,
    new Arbre(
      new Arbre(null, 21, null),
      25,
      new Arbre(null, 28, null)))
```

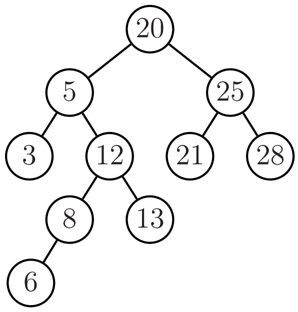


FIG. 1 – Un arbre binaire portant des informations aux nœuds.

Avant de poursuivre, reprenons le schéma déjà utilisé pour les listes. Quatre fonctions caractérisent les arbres : composer, cle, filsGauche, filsDroit. Elles s'implantent facilement :

```
static Arbre composer(Arbre g, int c, Arbre d)
{
  return new Arbre(g, c, d);
}

static int cle(Arbre a)
{
  return a.contenu;
}

static Arbre filsGauche(Arbre a)
{
  return a.filsG;
}

static Arbre filsDroit(Arbre a)
{
  return a.filsD;
}
```

Les quatre fonctions sont liées par les équations suivantes :

$$
\begin{aligned}
\mathsf{cle}(\mathsf{composer}(g, c, d)) &= c \\
\mathsf{filsGauche}(\mathsf{composer}(g, c, d)) &= g \\
\mathsf{filsDroit}(\mathsf{composer}(g, c, d)) &= d \\
\mathsf{composer}(\mathsf{filsGauche}(a), \mathsf{cle}(a), \mathsf{filsDroit}(a)) &= a; \quad (a \neq \mathsf{null})
\end{aligned}
$$

Comme pour les listes, ces quatre fonctions sont à la base d'opérations non destructives.

La définition récursive des arbres binaires conduit naturellement à une programmation récursive, comme pour les listes. Voici quelques exemples : la *taille* d'un arbre, c'est-à-dire le nombre $t(a)$ de ses nœuds, s'obtient par la formule

$$t(a) = \begin{cases} 0 & \text{si } a \text{ est vide} \\ 1 + t(a_g) + t(a_d) & \text{sinon.} \end{cases}$$

où sont notés a_g et a_d les sous-arbres gauche et droit de a. D'où la méthode

```
static int taille(Arbre a)
{
  if (a == null)
    return 0;
  return 1 + taille(a.filsG) + taille(a.filsD);
}
```

Des formules semblables donnent le nombre de feuilles ou la hauteur d'un arbre. Nous illustrons ce style de programmation par les *parcours* d'arbre définis page 89. Les trois parcours en profondeur s'écrivent :

```
static void parcoursPréfixe(Arbre a)
{
  if (a == null)
    return;
  System.out.print(a.contenu + " ");
  parcoursPréfixe(a.filsG);
  parcoursPréfixe(a.filsD);
}

static void parcoursInfixe(Arbre a)
{
  if (a == null)
    return;
  parcoursInfixe(a.filsG);
  System.out.print(a.contenu + " ");
  parcoursInfixe(a.filsD);
}

static void parcoursSuffixe(Arbre a)
{
  if (a == null)
    return;
  parcoursSuffixe(a.filsG);
  parcoursSuffixe(a.filsD);
  System.out.print(a.contenu + " ");
}
```

Le *parcours en largeur* d'un arbre binaire s'écrit simplement avec une *file*. Le *parcours préfixe* s'écrit lui aussi simplement de manière itérative, avec une *pile*. Nous reprenons les classes Pile et File du chapitre I, sauf que ce sont, cette fois-ci, des piles ou des files d'arbres. On écrit alors

```
static void parcoursPréfixeI(Arbre a)
{
  if (a == null)
    return;
```

```
    Pile p = new Pile();
    p.ajouter(a);
    while (!p.estVide())
    {
      a = p.valeur();
      p.supprimer();
      System.out.print(a.contenu + " ");
      if (a.filsD != null)
        p.ajouter(a.filsD);
      if (a.filsG != null)
        p.ajouter(a.filsG);
    }
  }
  static void parcoursLargeurI(Arbre a)
  {
    if (a == null)
      return;
    File f = new File();
    f.ajouter(a);
    while (!f.estVide())
    {
      a = f.valeur();
      f.supprimer();
      System.out.print(a.contenu + " ");
      if (a.filsG != null)
        f.ajouter(a.filsG);
      if (a.filsD != null)
        f.ajouter(a.filsD);
    }
  }
```

1.1 Implantation des arbres ordonnés par arbres binaires

Rappelons qu'un arbre est *ordonné* si la suite des fils de chaque nœud est ordonnée. Il est donc naturel de représenter les fils dans une liste chaînée. Un nœud contient aussi la référence à son fils aîné, c'est-à-dire à la tête de la liste de ses fils. Ainsi, chaque nœud contient deux références, celle à son fils aîné, et celle à son frère cadet. En d'autres termes, la structure des arbres binaires convient parfaitement, sous réserve de rebaptiser filsAine le champ filsG et frereCadet le champ filsD.

```
class ArbreOrdonne
{
  int contenu;
  Arbre filsAine, frereCadet;

  Arbre(Arbre g, int c, Arbre d)
  {
    filsAine = g;
    contenu = c;
    frereCadet = d;
  }
}
```

Cette représentation est aussi appelée « fils gauche — frère droit » (voir figure 2).

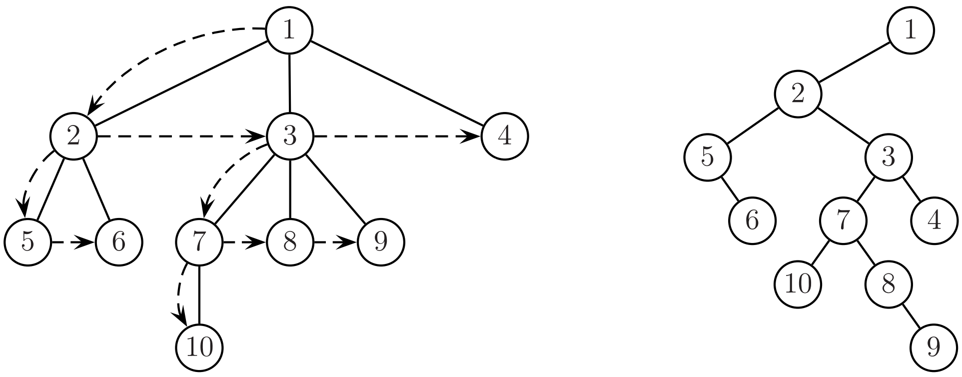


FIG. 2 – Représentation d'un arbre ordonné par un arbre binaire.

Noter que la racine de l'arbre binaire n'a pas de fils droit. En fait, cette représentation s'étend à la représentation, par un seul arbre binaire, d'une forêt ordonnée d'arbres ordonnés.

2 Arbres binaires de recherche

Les arbres binaires servent à gérer des informations. Chaque nœud contient une donnée prise dans un certain ensemble. Nous supposons dans cette section que cet ensemble est totalement ordonné. Ceci est le cas par exemple pour les entiers et pour les mots.

Un arbre binaire a est un *arbre binaire de recherche* si, pour tout nœud s de a, les contenus des nœuds du sous-arbre gauche de s sont strictement inférieurs au contenu de s, et que les contenus des nœuds du sous-arbre droit de s sont strictement supérieurs au contenu de s (cf. figure 3).

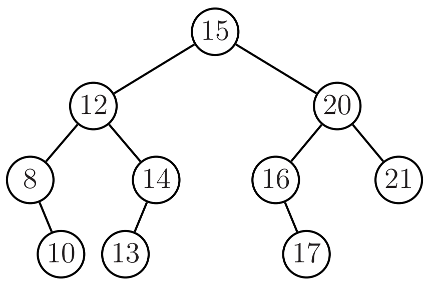


FIG. 3 – Un arbre binaire de recherche.

Une petite mise en garde : contrairement à ce que l'analogie avec les tas pourrait laisser croire, il ne suffit pas de supposer que, pour tout nœud s de l'arbre, le contenu du fils gauche de s soit strictement inférieur au contenu de s, et que le contenu du fils droit de s soit supérieur au contenu de s. Ainsi, dans l'arbre de la figure 3, si on change la valeur 13 en 11, on n'a plus un arbre de recherche...

Une conséquence directe de la définition est la règle suivante :

Règle. *Dans un arbre binaire de recherche, le parcours infixe fournit les contenus des nœuds en ordre croissant.*

Une seconde règle permet de déterminer dans certains cas le nœud qui précède un nœud donné dans le parcours infixe.

Règle. *Si un nœud possède un fils gauche, son prédécesseur dans le parcours infixe est le nœud le plus à droite dans son sous-arbre gauche. Ce nœud n'est pas nécessairement une feuille, mais il n'a pas de fils droit.*

Ainsi, dans l'arbre de la figure 3, la racine possède un fils gauche, et son prédécesseur est le nœud de contenu 14, qui n'a pas de fils droit...

Les arbres binaires de recherche fournissent, comme on le verra, une implantation souvent efficace d'un type abstrait de données, appelé *dictionnaire*, qui opère sur un ensemble totalement ordonné à l'aide des opérations suivantes :

- rechercher un élément
- insérer un élément
- supprimer un élément

Le dictionnaire est simplement une table d'associations (chapitre III) dont les informations sont inexistantes. Plusieurs implantations d'un dictionnaire sont envisageables : par tableau, par liste ordonnés ou non, par tas, et par arbre binaire de recherche. La table 1 rassemble la complexité des opérations de dictionnaire selon la structure de données choisie.

Implantation	Rechercher	Insérer	Supprimer
Tableau non ordonné	$O(n)$	$O(1)$[a]	$O(n)$
Liste non ordonnée	$O(n)$	$O(1)$[a]	$O(1)$[b]
Tableau ordonné	$O(\log n)$	$O(n)$	$O(n)$
Liste ordonnée	$O(n)$	$O(n)$	$O(1)$[b]
Tas	$O(n)$	$O(\log n)$	$O(n)$
Arbre de recherche	$O(h)$	$O(h)$	$O(h)$

[a] Ces coûts supposent que l'on sait que l'élément inséré n'est pas dans le dictionnaire.

[b] Ces coûts supposent l'élément supprimé déjà trouvé, ainsi que des opérations destructives.

Tab. 1 – Complexité des implantations de dictionnaires

L'entier h désigne la hauteur de l'arbre. On voit que lorsque l'arbre est bien équilibré, c'est-à-dire lorsque la hauteur est proche du logarithme de la taille, les opérations sont réalisables de manière particulièrement efficace.

2.1 Recherche d'une clé

Nous commençons l'implantation des opérations de dictionnaire sur les arbres binaires de recherche par l'opération la plus simple, la recherche. Plutôt que d'écrire une méthode booléenne qui teste la présence d'un élément dans l'arbre, nous écrivons une méthode qui retourne l'arbre dont la racine porte l'élément cherché s'il figure dans l'arbre, et `null` sinon. Comme toujours, il y a le choix entre une méthode récursive, calquée sur la définition récursive des arbres, et une méthode itérative, cheminant dans l'arbre. Nous présentons les deux, en commençant par la méthode récursive. Pour chercher si un élément x figure dans un arbre A, on commence par comparer x au contenu c de la racine de A. S'il y a égalité, on a trouvé la réponse ; sinon il y a deux cas selon que $x < c$ et $x > c$. Si $x < c$, alors x figure peut-être dans le sous-arbre gauche A_g de A, mais certainement pas dans le sous-arbre droit A_d. On élimine ainsi de la recherche tous les nœuds du sous-arbre droit. Cette méthode n'est pas sans rappeler la recherche dichotomique. La méthode s'écrit récursivement comme suit :

```
static Arbre chercher(int x, Arbre a)
{
  if (a == null || x == a.contenu)
    return a;
  if (x < a.contenu)
    return chercher(x, a.filsG);
  return chercher(x, a.filsD);
}
```

Cette méthode retourne null si l'arbre a ne contient pas x. Ceci inclut le cas où l'arbre est vide. Voici la méthode itérative.

```
static chercherI(int x, Arbre a)
{
  while(a != null && x != a.contenu)
    if (x < a.contenu)
      a = a.filsG;
    else
      a = a.filsD;
  return a;
}
```

On voit que la condition de *continuation* dans la méthode itérative chercherI est la négation de la condition d'*arrêt* de la méthode récursive, ce qui est logique.

2.2 Insertion d'une clé

L'adjonction d'un nouvel élément à un arbre modifie l'arbre. Nous sommes confrontés au même choix que pour les listes : soit on construit une nouvelle version de l'arbre (version non destructive), soit on modifie l'arbre existant (version destructive). Nous présentons une méthode récursive dans les deux versions. Dans les deux cas, si l'entier figure déjà dans l'arbre, on ne l'ajoute pas une deuxième fois. Voici la version destructive.

```
static Arbre inserer(int x, Arbre a)
{
  if (a == null)
    return new Arbre(null, x, null);
  if (x < a.contenu)
    a.filsG = inserer(x, a.filsG);
  else if (x > a.contenu)
    a.filsD = inserer(x, a.filsD);
  return a;
}
```

Voici la version non destructive.

```
static Arbre inserer(int x, Arbre a)
{
  if (a == null)
    return new Arbre(null, x, null);
  if (x < a.contenu)
  {
    Arbre b = inserer(x, a.filsG);
    return new Arbre(b, a.contenu, a.filsD);
  }
```

```
    else if (x > a.contenu)
    {
      Arbre b = inserer(x, a.filsD);
      return new Arbre(a.filsG, a.contenu, b);
    }
    return a;
  }
```

2.3 Suppression d'une clé

La *suppression* d'une clé dans un arbre est une opération plus complexe. Elle s'accompagne de la suppression d'un nœud. Comme on le verra, ce n'est pas toujours le nœud qui porte la clé à supprimer qui sera enlevé. Soit s le nœud qui porte la clé x à supprimer. Trois cas sont à considérer selon le nombre de fils du nœud x :

- si le nœud s est une feuille, alors on l'élimine ;
- si le nœud s possède un seul fils, on élimine s et on « remonte » ce fils.
- si le nœud s possède deux fils, on cherche le prédécesseur t de s. Celui-ci n'a pas de fils droit. On remplace le *contenu* de s par le contenu de t, et on élimine t.

La suppression de la feuille qui porte la clé 13 est illustrée dans la figure 4

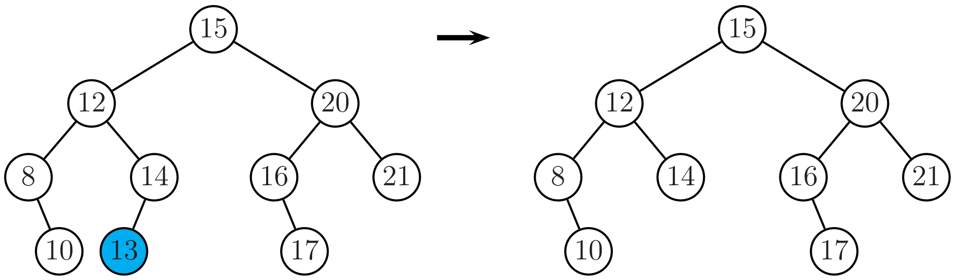


FIG. 4 – Suppression de la clé 13 par élimination d'une feuille.

La figure 5 illustre la « remontée » : le nœud s qui porte la clé 16 n'a qu'un seul enfant. Cet enfant devient l'enfant du père de s, le nœud de clé 20.

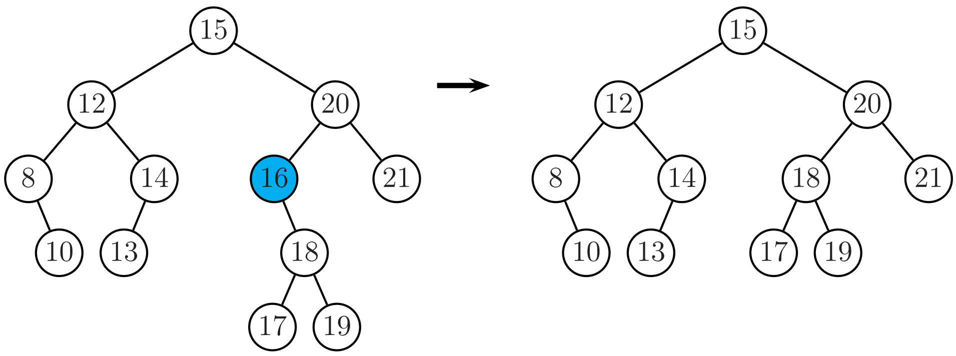


FIG. 5 – Suppression de la clé 16 par remontée du fils.

Le cas d'un nœud à deux fils est illustré dans la figure 6. La clé à supprimer se trouve à la racine de l'arbre. On ne supprime pas le nœud, mais seulement sa clé, en remplaçant la clé par une autre clé. Pour conserver l'ordre sur les clés, il n'y a que deux choix : la clé du prédécesseur dans l'ordre infixe, ou la clé du successeur. Nous choisissons la première solution. Ainsi, la clé 14 est mise à la racine de l'arbre. Nous sommes alors ramenés au problème de la suppression du nœud du prédécesseur et de sa clé. Comme le prédécesseur est le nœud le plus à droite du sous-arbre gauche, il n'a pas de fils droit, donc il a zéro ou un fils, et sa suppression est couverte par les deux premiers cas.

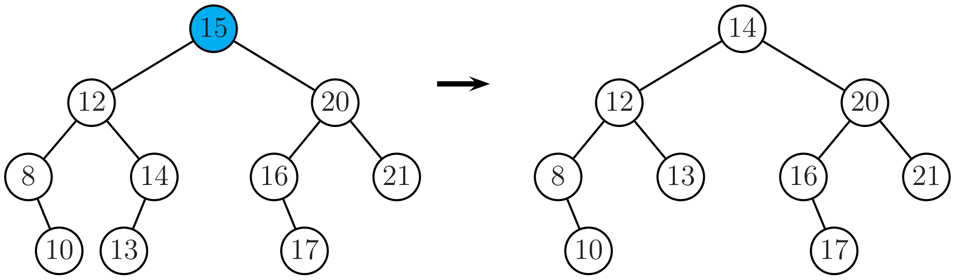


FIG. 6 – Suppression de la clé 15 par substitution de la clé 14 et suppression de ce nœud.

Trois méthodes coopèrent pour la suppression. La première, suppression, recherche le nœud portant la clé à supprimer ; la deuxième, suppressionRacine, effectue la suppression selon les cas énumérés ci-dessus. La troisième, dernierDescendant est une méthode auxiliaire ; elle calcule le précédesseur d'un nœud qui a un fils gauche.

```
static Arbre supprimer(int x, Arbre a)
{
  if (a == null)
    return a;
  if (x == a.contenu)
    return supprimerRacine(a);
  if (x < a.contenu)
    a.filsG = supprimer(x, a.filsG);
  else
    a.filsD = supprimer(x, a.filsD);
  return a;
}
```

La méthode suivante supprime la clé de la racine de l'arbre.

```
static Arbre supprimerRacine(Arbre a)
{
  if (a.filsG == null)
    return a.filsD;
  if (a.filsD == null)
    return a.filsG;
  Arbre f = dernierDescendant(a.filsG);
  a.contenu = f.contenu;
  a.filsG = supprimer(f.contenu, a.filsG);
}
```

La dernière méthode est toute simple :

```
static Arbre dernierDescendant(Arbre a)
{
  if (a.filsD == null)
```

```
    return a;
  return dernierDescendant(a.filsD);
}
```

La récursivité croisée entre les méthodes supprimer et supprimerRacine est déroutante au premier abord. En fait, l'appel à supprimer à la dernière ligne de supprimerRacine conduit au nœud prédécesseur de la racine de l'arbre, appelé f. Comme ce nœud n'a pas deux fils, il n'appelle pas une deuxième fois la méthode supprimerRacine...

Il est intéressant de voir une réalisation itérative de la suppression. Elle démonte entièrement la « mécanique » de l'algorithme. En fait, chacune des trois méthodes peut séparément être écrite de façon récursive.

```
static Arbre supprimer(int x, Arbre a)
{
  Arbre b = a;
  while (a != null && x != a.contenu)
    if (x < a.contenu)
      a = a.filsG;
    else
      a = a.filsD;
  if (a != null)
    a = supprimerRacine(a);
  return b;
}
```

Voici la deuxième.

```
static Arbre supprimerRacine(Arbre a)
{
  if (a.filsG == null)
    return a.filsD;
  if (a.filsD == null)
    return a.filsG;
  Arbre b = a.filsG;
  if (b.filsD == null)
  { // cas (i)
    a.contenu = b.contenu;
    a.filsG = b.filsG;
  }
  else
  { // cas (ii)
    Arbre p = avantDernierDescendant(b);
    Arbre f = p.filsD;
    a.contenu = f.contenu;
    p.filsD = f.filsG;
  }
  return a;
}
```

Et voici le calcul de l'avant-dernier descendant :

```
static Arbre avantDernierDescendant(Arbre a)
{
  while (a.filsD.filsD != null)
    a = a.filsD;
```

```
        return a;
    }
```

Décrivons plus précisément le fonctionnement de la méthode supprimerRacine. La première partie permet de se ramener au cas où la racine de l'arbre a a deux fils. On note b le fils gauche de a, et pour déterminer le prédécesseur de la racine de a, on cherche le nœud le plus à droite dans l'arbre b. Deux cas peuvent se produire :

(i) la racine de b n'a pas de fils droit, ou

(ii) la racine de b a un fils droit.

Les deux cas sont illustrés sur les figures 7 et 8.

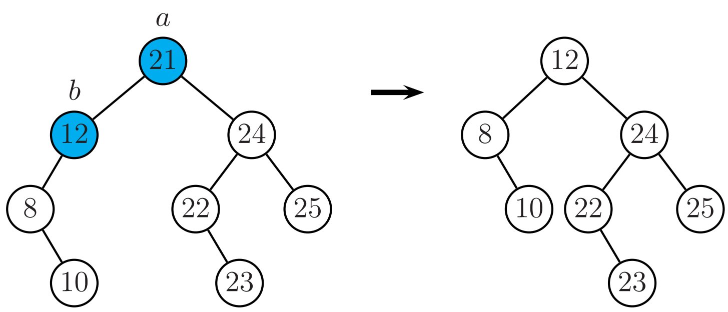


FIG. 7 – Suppression de la racine, version itérative, cas (i).

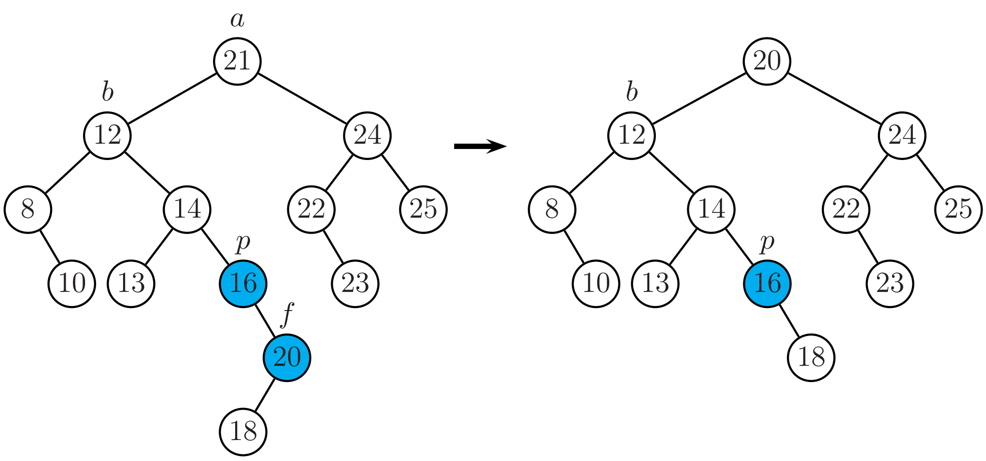


FIG. 8 – Suppression de la racine, version itérative, cas (ii).

Dans le premier cas, la clé de la racine de b est transférée à la racine de a, et b est remplacée par son sous-arbre gauche (qui peut d'ailleurs être vide). Dans le deuxième cas, on cherche l'avant-dernier descendant, noté p, de b sur la branche droite de b, au moyen de la méthode avantDernierDescendant. Cela peut être b lui-même, ou un de ses descendants (notons que dans le cas (i), l'avant-dernier descendant n'existe pas, ce qui explique le traitement séparé opéré dans ce cas). Le sous-arbre droit f de p n'est pas vide par définition. La clé de f est transférée à la racine de a, et f est remplacé par son sous-arbre gauche — ce qui fait disparaître la racine de f.

2.4 Hauteur moyenne

Il est facile de constater, sur nos implantations, que la recherche, l'insertion et la suppression dans un arbre binaire de recherche se font en complexité $O(h)$, où h est la hauteur de l'arbre.

Le cas le pire, pour un arbre à n nœuds, est $O(n)$. En ce qui concerne la *hauteur moyenne*, deux cas sont à considérer. La première des propositions s'applique aux arbres, la deuxième aux permutations.

Proposition 10 Lorsque tous les arbres binaires à n nœuds sont équiprobables, la hauteur moyenne d'un arbre binaire à n nœuds est en $O(\sqrt{n})$.

Proposition 11 Lorsque toutes les permutations de $\{1,\ldots,n\}$ sont équiprobables, la hauteur moyenne d'un arbre binaire de recherche obtenu par insertion des entiers d'une permutation dans un arbre initialement vide est $O(n \log n)$.

La différence provient du fait que plusieurs permutations peuvent donner le même arbre. Par exemple les permutations $2, 1, 3$ et $2, 3, 1$ produisent toutes les deux l'arbre de la figure 9.

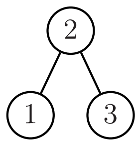


FIG. 9 – L'arbre produit par la suite d'insertions $2, 1, 3$, ou par la suite $2, 3, 1$.

3 Arbres équilibrés

Comme nous l'avons déjà constaté, les coûts de la recherche, de l'insertion et de la suppression dans un arbre binaire de recherche sont de complexité $O(h)$, où h est la hauteur de l'arbre. Le cas le pire, pour un arbre à n nœuds, est $O(n)$. Ce cas est atteint par des arbres très déséquilibrés, ou « filiformes ». Pour éviter que les arbres puissent prendre ces formes, on utilise des opérations plus ou moins simples, mais peu coûteuses en temps, de transformation d'arbres. À l'aide de ces transformations on tend à rendre l'arbre le plus régulier possible dans un sens qui est mesuré par un paramètre dépendant en général de la hauteur. Une famille d'arbres satisfaisant une condition de régularité est appelée une famille d'arbres *équilibrés*. Plusieurs espèces de tels arbres ont été développés, notamment les arbres AVL, les arbres 2-3, les arbres rouge et noir, ainsi qu'une myriade de variantes. Dans les langages comme Java ou C++, des modules de gestion d'ensembles sont préprogrammés. Lorsqu'un ordre total existe sur les éléments de ces ensembles, ils sont en général gérés, en interne, par des arbres rouge et noir.

3.1 Arbres AVL

La famille des arbres AVL est nommée ainsi d'après leurs inventeurs, Adel'son-Velskii et Landis, qui les ont présentés en 1962. Au risque de paraître vieillots, nous décrivons ces arbres plus en détail parce que leur programmation peut être menée jusqu'au bout, et parce que les principes utilisés dans leur gestion se retrouvent dans d'autres familles plus complexes.

Un arbre binaire est un *arbre AVL* si, pour tout nœud de l'arbre, les hauteurs des sous-arbres gauche et droit diffèrent d'au plus 1.

Rappelons qu'une feuille est un arbre de hauteur 0, et que l'arbre vide a la hauteur -1. L'arbre vide, et l'arbre réduit à une feuille, sont des arbres AVL.

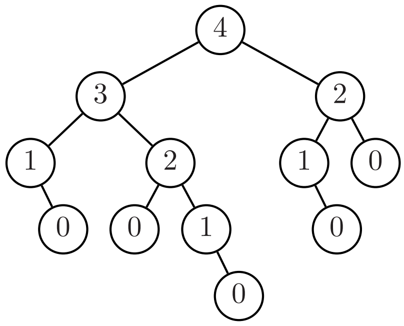


FIG. 10 – Un arbre AVL, avec les hauteurs aux nœuds.

L'arbre de la figure 10 porte, dans chaque nœud, la hauteur de son sous-arbre.

Un autre exemple est fourni par les *arbres de Fibonacci*, qui sont des arbres binaires A_n tels que les sous-arbres gauche et droit de A_n sont respectivement A_{n-1} et A_{n-2}. Les premiers arbres de Fibonacci sont donnés dans la figure 11.

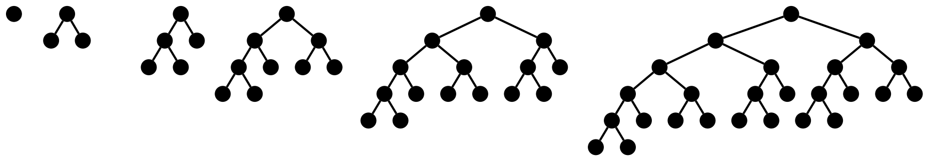


FIG. 11 – Les premiers arbres de Fibonacci.

L'intérêt des arbres AVL résulte du fait que leur hauteur est toujours logarithmique en fonction de la taille de l'arbre. En d'autres termes, la recherche, l'insertion et la suppression (sous réserve d'un éventuel rééquilibrage) se font en temps logarithmique. Plus précisément, on a la propriété que voici.

Proposition 12 Soit A un arbre AVL ayant n nœuds et de hauteur h. Alors

$$\log_2(1+n) \leqslant 1+h \leqslant \alpha \log_2(2+n)$$

avec $\alpha = 1/\log_2((1+\sqrt{5})/2) \leqslant 1.44$.

Preuve. On a toujours $n \leqslant 2^{h+1}-1$, donc $\log_2(1+n) \leqslant 1+h$. Soit $N(h)$ le nombre minimum de nœuds d'un arbre AVL de hauteur h. Alors

$$N(h) = 1 + N(h-1) + N(h-2)$$

car un arbre minimal aura un sous-arbre de hauteur $h-1$ et l'autre sous-arbre de hauteur seulement $h-2$. La suite $F(h) = 1+N(h)$ vérifie $F(0) = 2$, $F(1) = 3$, $F(h+2) = F(h+1)+F(h)$ pour $h \geqslant 0$, donc

$$F(h) = \frac{1}{\sqrt{5}}(\Phi^{h+3} - \Phi^{-(h+3)})$$

où $\Phi = (1+\sqrt{5})/2$. Il en résulte que $1+n \geqslant F(h) > \frac{1}{\sqrt{5}}(\Phi^{h+3}-1)$, soit en passant au logarithme en base Φ, $h+3 < \log_\Phi(\sqrt{5}(2+n)) < \log_2(2+n)/\log_2\Phi + 2$. □

Par exemple, pour un arbre AVL qui a 100000 nœuds, la hauteur est comprise entre 17 et 25. C'est le nombre d'opérations qu'il faut donc pour rechercher, insérer ou supprimer une donnée dans un tel arbre.

3.1.1 Rotations

Nous introduisons maintenant une opération sur les arbres appelée *rotation*. Les rotations sont illustrées sur la figure 12. Soit $A = (B, q, W)$ un arbre binaire tel que $B = (U, p, V)$. La *rotation gauche* est l'opération

$$((U, p, V), q, W) \to (U, p, (V, q, W))$$

et la rotation droite est l'opération inverse. Les rotations gauche (droite) ne sont donc définies que pour les arbres binaires non vides dont le sous-arbre gauche (resp. droit) n'est pas vide.

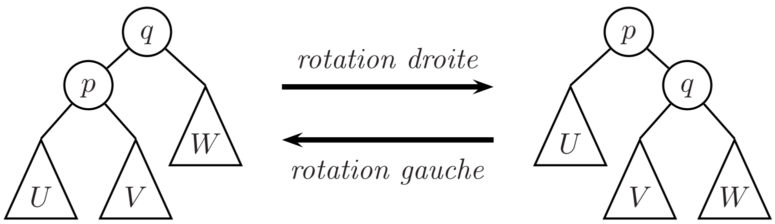


Fig. 12 – Rotation.

Remarquons en passant que pour l'arbre d'une expression arithmétique, si les symboles d'opération p et q sont les mêmes, les rotations expriment que l'opération est associative.

Les rotations ont la propriété de pouvoir être implantées en temps constant (voir ci-dessous), et de préserver l'ordre infixe. En d'autres termes, si A est un arbre binaire de recherche, tout arbre obtenu à partir de A par une suite de rotations gauche ou droite d'un sous-arbre de A reste un arbre binaire de recherche. En revanche, comme le montre la figure 13, la propriété AVL n'est pas conservée par rotation.

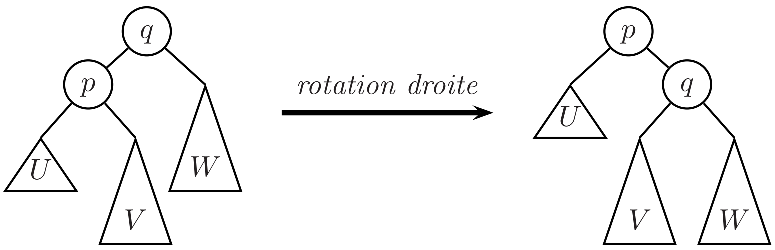


Fig. 13 – Les rotations ne préservent pas la propriété AVL.

Pour remédier à cela, on considère une *double rotation* qui est en fait composée de deux rotations. La figure 14 décrit une double rotation droite, et montre comment elle est composée d'une rotation gauche du sous-arbre gauche suivie d'un rotation droite. Plus précisément, soit $A = ((U, p, (V, q, W)), r, X)$ un arbre dont le sous-arbre gauche possède un sous-arbre droit. La *double rotation droite* est l'opération

$$A = ((U, p, (V, q, W)), r, X) \to A' = ((U, p, V), q, (W, r, X))$$

Vérifions qu'elle est bien égale à la composition de deux rotations. D'abord, une rotation gauche de $B = (U, p, (V, q, W))$ donne $B' = ((U, p, V), q, W)$, et l'arbre $A = (B, r, X)$ devient $A'' = (B', r, X)$; la rotation droite de A'' donne en effet A'. On voit qu'une double rotation droite diminue la hauteur relative des sous-arbres V et W, et augmente celle de X. La double rotation gauche est définie de manière symétrique.

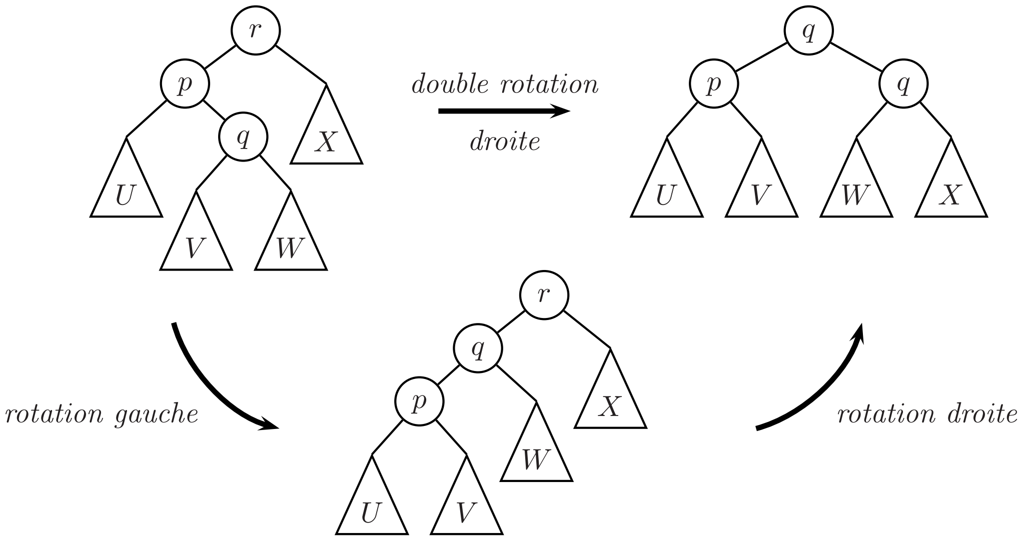


Fig. 14 – Rotations doubles.

3.1.2 Implantation des rotations

Voici une implantation non destructive d'une rotation gauche.

```
static Arbre rotationG(Arbre a)   // non destructive
{
  Arbre b = a.filsD;
  Arbre c = new Arbre(a.filsG, a.contenu, b.filsG);
  return new Arbre(c, b.contenu, b.filsD);
}
```

La fonction suppose que le sous-arbre gauche, noté `b`, n'est pas vide. La rotation gauche destructive est aussi simple à écrire.

```
static Arbre rotationG(Arbre a)   // destructive
{
  Arbre b = a.filsD;
  a.filsD = b.filsG;
  b.filsG = a;
  return b;
}
```

Les double rotations s'écrivent par composition.

3.1.3 Insertion et suppression dans un arbre AVL

L'insertion et la suppression dans un arbre AVL peuvent transformer l'arbre en un arbre qui ne satisfait plus la contrainte sur les hauteurs. Dans la figure 15, un nœud portant l'étiquette 50 est inséré dans l'arbre de gauche. Après insertion, on obtient l'arbre du milieu qui n'est plus AVL. Une double rotation autour de la racine suffit à rééquilibrer l'arbre.

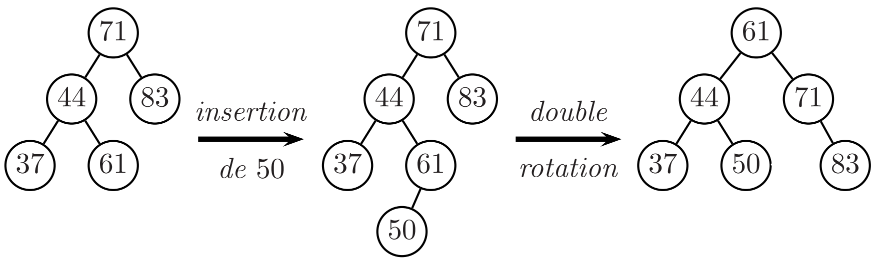


FIG. 15 – Insertion suivie d'une double rotation.

Cette propriété est générale. Après une insertion (respectivement une suppression), il suffit de rééquilibrer l'arbre par des rotations ou double rotations le long du chemin qui conduit à la feuille où l'insertion (respectivement la suppression) a eu lieu. L'algorithme est le suivant :

Algorithme. *Soit A un arbre, G et D ses sous-arbres gauche et droit. On suppose que $|h(G) - h(D)| = 2$. Si $h(G) - h(D) = 2$, on fait une rotation droite, mais précédée d'une rotation gauche de G si $h(g) < h(d)$ (on note g et d les sous-arbres gauche et droit de G). Si $h(G) - h(D) = -2$ on opère de façon symétrique.*

On peut montrer en exercice qu'il suffit d'une seule rotation ou double rotation pour rééquilibrer un arbre AVL après une insertion. Cette propriété n'est plus vraie pour une suppression.

3.1.4 Implantation : la classe Avl

Pour l'implantation, nous munissons chaque nœud d'un champ supplémentaire qui contient la hauteur de l'arbre dont il est racine. Pour une feuille par exemple, ce champ a la valeur 0. Pour l'arbre vide, qui est représenté par null et qui n'est donc pas un objet, la hauteur vaut -1. La méthode H sert à simplifier l'accès à la hauteur d'un arbre.

```
class Avl
{
  int contenu;
  int hauteur;
  Avl filsG, filsD;

  Avl(Avl g, int c, Avl d)
  {
    filsG = g;
    contenu = c;
    filsD = d;
    hauteur = 1 + Math.max(H(g),H(d));
  }

  static int H(Avl a)
  {
    return (a == null) ? -1 : a.hauteur;
  }

  static void calculerHauteur(Avl a)
  {
    a.hauteur = 1 + Math.max(H(a.filsG), H(a.filsD));
  }
  ...
}
```

La méthode calculerHauteur recalcule la hauteur d'un arbre à partir des hauteurs de ses sous-arbres. L'usage de H permet de traiter de manière unifiée le cas où l'un de ses sous-arbres serait l'arbre vide. Les rotations sont reprises de la section précédente. On utilise la version non destructive qui réévalue la hauteur. Ces méthodes et les suivantes font toutes partie de la classe Avl.

```
static Avl rotationG(Avl a)
{
  Avl b = a.filsD;
  Avl c = new Avl(a.filsG, a.contenu, b.filsG);
  return new Avl(c, b.contenu, b.filsD);
}
```

La méthode principale implante l'algorithme de rééquilibrage exposé plus haut.

```
static Avl equilibrer(Avl a)
{
  a.hauteur = 1 + Math.max(H(a.filsG), H(a.filsD));
  if(H(a.filsG) - H(a.filsD) == 2)
  {
    if (H(a.filsG.filsG) < H(a.filsG.filsD))
      a.filsG = rotationG(a.filsG);
    return rotationD(a);
  } //else version symétrique
  if (H(a.filsG) - H(a.filsD) == -2)
  {
    if (H(a.filsD.filsD) < H(a.filsD.filsG))
      a.filsD = rotationD(a.filsD);
    return rotationG(a);
  }
  return a;
}
```

Il reste à écrire les méthodes d'insertion et de suppression, en prenant soin de rééquilibrer l'arbre à chaque étape. On reprend simplement les méthodes déjà écrites pour un arbre binaire de recherche général. Pour l'insertion, on obtient

```
static Avl inserer(int x, Avl a)
{
  if (a == null)
    return new Avl(null, x, null);
  if (x < a.contenu)
    a.filsG = inserer(x, a.filsG);
  else if (x > a.contenu)
    a.filsD = inserer(x, a.filsD);
  return equilibrer(a); //seul changement
}
```

La suppression s'écrit comme suit

```
static Avl supprimer(int x, Avl a)
{
  if (a == null)
    return a;
  if (x == a.contenu)
```

```
      return supprimerRacine(a);
    if (x < a.contenu)
      a.filsG = supprimer(x, a.filsG);
    else
      a.filsD = supprimer(x, a.filsD);
    return equilibrer(a); // seul changement
  }

  static Avl supprimerRacine(Avl a)
  {
    if (a.filsG == null && a.filsD == null)
      return null;
    if (a.filsG == null)
      return equilibrer(a.filsD);
    if (a.filsD == null)
      return equilibrer(a.filsG);
    Avl b = dernierDescendant(a.filsG);
    a.contenu = b.contenu;
    a.filsG = supprimer(a.contenu, a.filsG);
    return equilibrer(a); // seul changement
    }

  static Avl dernierDescendant(Avl a) // inchangée
  {
    if (a.filsD == null)
      return a;
    return dernierDescendant(a.filsD);
  }
```

3.2 *B*-arbres et arbres *a*-*b*

Dans cette section, nous décrivons de manière succinte les arbres *a*-*b*. Il s'agit d'une des variantes d'arbres équilibrés qui ont la propriété que toutes leurs feuilles sont au même niveau, les nœuds internes pouvant avoir un nombre variable de fils (ici entre a et b). Dans cette catégorie d'arbres, on trouve aussi les *B*-arbres et en particulier les arbres 2-3-4. Les arbres rouge et noir (ou bicolores) sont semblables.

L'intérêt des arbres équilibrés est qu'ils permettent des modifications en temps logarithmique. Lorsque l'on manipule de très grands volumes de données, il survient un autre problème, à savoir l'accès proprement dit aux données. En effet, les données ne tiennent pas en mémoire vive, et les données sont donc accessibles seulement sur la mémoire de masse, un disque en général. Or, un seul accès disque peut prendre, en moyenne, environ autant de temps que 200 000 instructions. Les *B*-arbres ou les arbres *a*-*b* servent, dans ce contexte, à minimiser les accès au disque.

Un disque est divisé en pages (par exemple de taille 512, 2048, 4092 ou 8192 octets). La page est l'unité de transfert entre mémoire centrale et disque. Il est donc rentable de grouper les données par blocs, et de les manipuler de concert.

Les données sont en général repérées par des clés, qui sont rangées dans un arbre. Si chaque accès à un nœud requiert un accès disque, on a intérêt à avoir des nœuds dont le nombre de fils est voisin de la taille d'une page. De plus, la hauteur d'un tel arbre — qui mesure le nombre d'accès disques nécessaire — est alors très faible. En effet, si chaque nœud a de l'ordre de 1000 fils, il suffit d'un arbre de hauteur 3 pour stocker un milliard de clés.

Nous considérons ici des arbres de recherche qui ne sont plus binaires, mais d'arité plus grande. Chaque nœud interne d'un tel arbre contient, en plus des références vers ses sous-arbres, des *balises*, c'est-à-dire des valeurs de clé qui permettent de déterminer le sous-arbre où se trouve l'information cherchée. Plus précisément, si un nœud interne possède $d+1$ sous-arbres $A_0, \dots, A_d$, alors il est muni de d balises $k_1, \dots, k_d$ telles que

$$c_0 \leqslant k_1 < c_1 \leqslant \cdots \leqslant k_d < c_d$$

pour toute séquence de clés $(c_0, \dots, c_d)$, où chaque c_i est une clé du sous-arbre A_i (cf. figure 16).

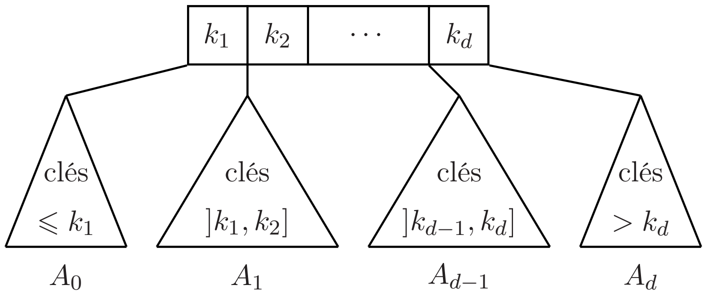


Fig. 16 – Un nœud interne muni de balises et ses sous-arbres.

Il en résulte que pour chercher une clé c, on détermine le sous-arbre A_i approprié en déterminant lequel des intervalles $]-\infty, k_1]$, $]k_1, k_2]$,. . .,$]k_{d-1}, k_d]$, $]k_d, \infty[$ contient la clé.

3.2.1 Arbres *a*-*b*

Soient $a \geqslant 2$ et $b \geqslant 2a - 1$ deux entiers. Un arbre *a*-*b* est un arbre de recherche tel que

- les feuilles ont toutes la même profondeur,
- la racine a au moins 2 fils (sauf si l'arbre est réduit à sa racine) et au plus b fils,
- les autres nœuds internes ont au moins a et au plus b fils.

Les arbres 2-3 sont les arbres obtenus quand a et b prennent leurs valeurs minimales : tout nœud interne a alors 2 ou 3 fils.

Les B-arbres sont comme les arbres *a*-*b* avec $b = 2a-1$ mais avec une interprétation différente : les informations sont aussi stockées aux nœuds internes, alors que, dans les arbres que nous considérons, les clés aux nœuds internes ne servent qu'à la navigation.

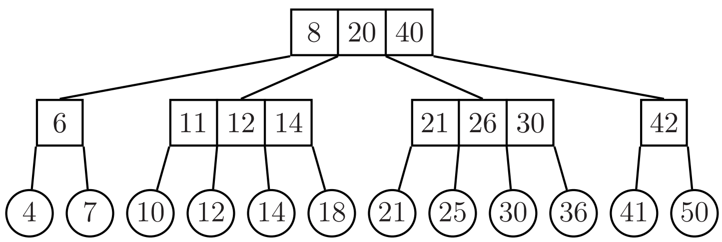


Fig. 17 – Un arbre 2-4. Un nœud a entre 2 et 4 fils.

L'arbre de la figure 17 représente un arbre 2-4. Les nœuds internes contiennent les balises, et les feuilles contiennent les clés. L'intérêt des arbres *a*-*b* est justifié par la proposition suivante.

Proposition 13 Si un arbre *a*-*b* de hauteur h contient n feuilles, alors

$$\log n / \log b \leqslant h < 1 + \log(n/2)/\log a\,.$$

Preuve. Tout nœud a au plus b fils. Il y a donc au plus b^h feuilles. Tout nœud autre que la racine a au moins a fils, et la racine en a au moins 2. Au total, il y a au moins $2a^{h-1}$ feuilles, donc $2a^{h-1} \leqslant n \leqslant b^h$. □

Il résulte de la proposition précédente que la hauteur d'un arbre a-b ayant n feuilles est en $O(\log n)$. La complexité des algorithmes décrit ci-dessous (insertion et suppression) sera donc elle aussi en $O(\log n)$.

3.2.2 Insertion dans un arbre a-b

La recherche dans un arbre a-b repose sur le même principe que celle utilisée pour les arbres binaires de recherche : on parcourt les balises du nœud courant pour déterminer le sous-arbre dans lequel il faut poursuivre la recherche. Pour l'insertion, on commence par déterminer, par une recherche, l'emplacement de la feuille où l'insertion doit avoir lieu. On insère alors une nouvelle feuille, et une balise appropriée : la balise est la plus petite valeur de la clé à insérer et de la clé à sa droite.

Reste le problème du rééquilibrage qui se pose lorsque le nombre de fils d'un nœud dépasse le nombre autorisé. Si un nœud a $b+1$ fils, alors il est éclaté en deux nœuds qui se partagent les fils de manière équitable : le premier nœud reçoit les $\lfloor (b+1)/2 \rfloor$ fils de gauche, le deuxième les fils de droite. Noter que $b+1 \geqslant 2a$, et donc chaque nouveau nœud aura au moins a fils. Les balises sont également partagées, et la balise centrale restante est transmise au nœud père, pour qu'il puisse à son tour procéder à l'insertion des deux fils à la place du nœud éclaté. L'insertion

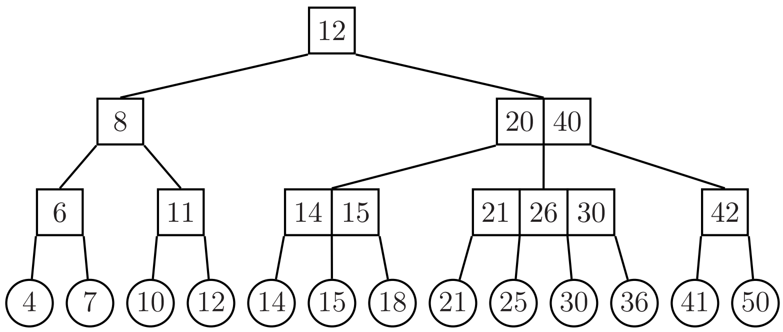


Fig. 18 – Un arbre 2-4. Un nœud a entre 2 et 4 fils.

de la clé 15 dans l'arbre 17 produit l'arbre de la figure 18. Cette insertion se fait par un double éclatement. D'abord, le nœud aux balises $11, 12, 14$ de l'arbre 17 est éclaté en deux. Mais alors, la racine de l'arbre 17 a un nœud de trop. La racine elle-même est éclatée, ce qui fait augmenter la hauteur de l'arbre.

Il est clair que l'insertion d'une nouvelle clé peut au pire faire éclater les nœuds sur le chemin de son lieu d'insertion à la racine — et la racine elle-même. Le coût est donc borné logarithmiquement en fonction du nombre de feuilles dans l'arbre.

3.2.3 Suppression dans un arbre a-b

Comme d'habitude, la suppression est plus complexe. Il s'agit de fusionner un nœud avec un nœud frère lorsqu'il n'a plus assez de fils, c'est-à-dire si son nombre de fils descend au dessous de a. Mais si son frère (gauche ou droit) a beaucoup de fils, la fusion des deux nœuds risque de conduire à un nœud qui a trop de fils et qu'il faut éclater. On groupe ces deux opérations sous la forme d'un *partage* (voir figure 19).

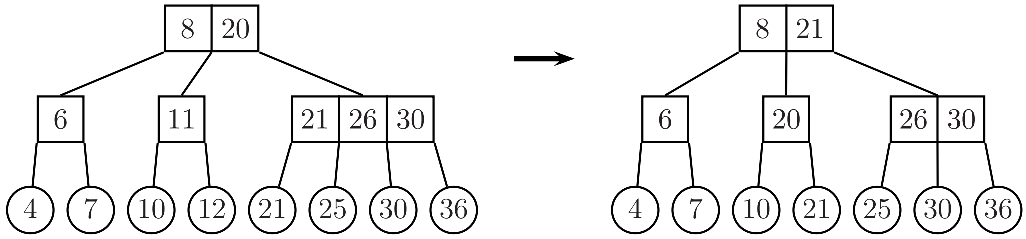


FIG. 19 – La suppression de 12 est absorbée par un partage.

Plus précisément, l'algorithme est le suivant :

- Supprimer la feuille, puis la balise figurant sur le chemin de la feuille à la racine.
- Si les nœuds ainsi modifiés ont toujours a fils, l'arbre est encore a-b. Si un nœud possède $a-1$ fils examiner les frères *adjacents*.
  - Si l'un des frères possède au moins $a+1$ fils, faire un partage avec ce frère.
  - Sinon, les frères adjacents ont a fils, et la fusion avec l'un des deux produit un nœud ayant $2a-1 \leqslant b$ fils.

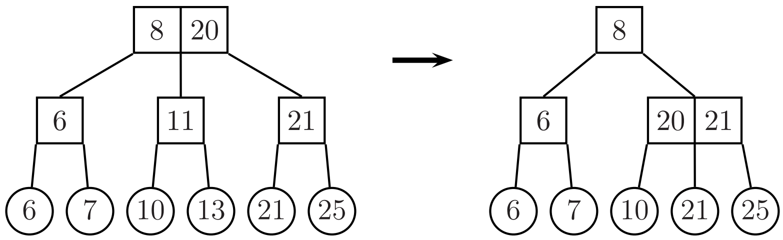


FIG. 20 – La suppression de 13 entraîne la fusion des deux nœuds de droite.

Là encore, le coût est majoré par un logarithme du nombre de feuilles. Il faut noter qu'il s'agit du comportement dans le cas le plus défavorable. On peut en effet démontrer

Proposition 14 On considère une suite quelconque de n insertions ou suppressions dans un arbre 2-4 initialement vide. Alors le nombre total d'éclatements, de partage et de fusions est au plus $3n/2$.

Ceci signifie donc qu'en moyenne, $1,5$ opérations suffisent pour rééquilibrer l'arbre, et ce, quel que soit sa taille ! Des résultats analogues valent pour des valeurs de a et b plus grandes.

Chapitre VI

Expressions régulières

1 Langages réguliers

1.1 Les mots

On se donne un ensemble Σ de caractères, parfois dénommé *alphabet*. L'ensemble des mots sur Σ, noté Σ^*, est l'ensemble des suites finies de caractères. Un *langage* est un sous-ensemble de Σ^*, c'est-à-dire un ensemble particulier de mots.

Par exemple si Σ est l'ensemble des lettres usuelles, on peut définir le langage L_1 des mots français. Ou encore si Σ est l'ensemble des chiffres de 0 à 9, on peut définir le langage L_2 des représentations en base 10 des entiers naturels. Dans le premier cas, on peut tout simplement tenter de définir L_1 comme l'ensemble de tous les mots présents dans le dictionnaire de l'Académie française, dans le second cas, on peut recourir à une définition du style « *un entier (décimal) est le chiffre zéro, ou une suite non vide de chiffres qui ne commence pas par zéro* ». Les expressions régulières (ou rationnelles) sont une notation très commode pour définir des langages de mots du style de L_1 et L_2 ci-dessus.

Définissons tout de suite quelques notations sur les mots et les langages. Parmi les mots de Σ^* on distingue le mot vide noté ϵ. Le mot vide est l'unique mot de longueur zéro. La concaténation de deux mots m_1 et m_2 est le mot obtenu en mettant m_1 à la fin de m_2. On note $\cdot$ l'opérateur de concaténation, mais on omet souvent cet opérateur et on note donc souvent la concaténation $m_1 \cdot m_2$ par une simple juxtaposition $m_1 m_2$. La concaténation est une opération associative qui possède un élément neutre : le mot vide ϵ. Dans l'écriture $m = m_1 m_2$, m_1 est le *préfixe* du mot m, tandis que m_2 est le *suffixe* du mot m.

La concaténation s'étend naturellement aux ensembles de mots, on note $L_1 \cdot L_2$ le langage obtenu en concaténant tous les mots de L_1 avec les mots de L_2.

$$L_1 \cdot L_2 = \{m_1 \cdot m_2 \mid m_1 \in L_1 \wedge m_2 \in L_2\}$$

Enfin on note $L^\star$ le langage obtenu en concaténant les mots de L.

$$L^0 = \{\,\epsilon\,\} \qquad\qquad L^{n+1} = L^n \cdot L \qquad\qquad L^\star = \bigcup_{i \in \mathbb{N}} L^i$$

C'est-à-dire qu'un mot m de L^* est la concaténation de n mots $(n \geq 0)$ $m_1, \ldots m_n$, où les m_i sont tous des mots de L.

1.2 Syntaxe des expressions régulières

Les expressions régulières ou motifs, notées p, sont définies ainsi :

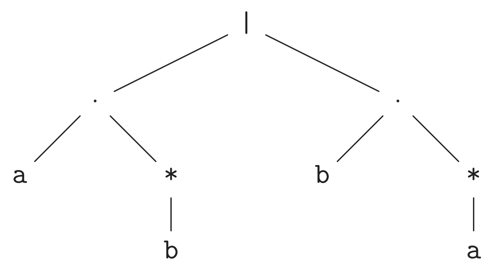
- Le mot vide ϵ est une expression régulière.

- Un caractère c (un élément de l'alphabet Σ) est une expression régulière.
- Si p_1 et p_2 sont des expressions régulières, alors l'alternative $p_1 \mid p_2$ est une expression régulière.
- Si p_1 et p_2 sont des expressions régulières, alors la concaténation $p_1 \cdot p_2$ est une expression régulière.
- Si p est une expression régulière, alors la répétition $p*$ est une expression régulière.

On reconnaît ici la définition d'un arbre, et même d'un arbre de syntaxe abstraite. Il y a deux sortes de feuilles, mot vide et caractères ; deux sortes de nœuds binaires, concaténation et alternative ; et une sorte de nœud unaire, la répétition. Comme d'habitude pour lever les ambiguïtés dans l'écriture linéaire d'un arbre, on a recours à des priorités (à savoir, la répétition est plus prioritaire que la concaténation, elle-même plus prioritaire que l'alternative) et à des parenthèses.

Ainsi, pour $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, le motif `ab*|ba*` est à comprendre comme `((a)(b*))|((b)(a*))`, ou plus informatiquement comme l'arbre de la figure 1. Pour se souvenir des priorités adoptées

Fig. 1 – L'arbre de syntaxe abstraite de l'expression `ab*|ba*`

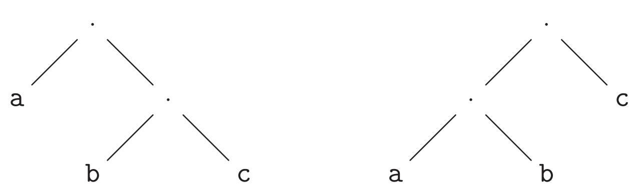


on peut noter l'analogie avec les expressions arithmétiques : les opérations de concaténation et d'alternative ont les même priorités respectives que la multiplication et l'addition, enfin la répétition se comporte comme la mise à la puissance.

Dans ce polycopié nous exprimons les expressions régulières sous la forme « programme » et non pas d'expression mathématiques Concrètement, nous avons écrit `ab*|ba*` et non pas $ab^\star \mid ba^\star$. Ce parti pris entraîne que le motif vide ne peut plus être donné par ϵ, si nécessaire nous le représenterons donc par exemple comme `()`.

Avec un peu d'habitude, on comprend nos écritures comme des arbres de syntaxe abstraites. Cela lève toutes les ambiguïtés d'entrée de jeu. On doit sans doute remarquer qu'en réalité, nos priorités ne lèvent pas toutes les ambiguïtés. En effet, on peut comprendre `abc` comme `a(bc)` ou comme `(ab)c`, c'est-à-dire comme l'arbre de gauche ou comme l'arbre de droite de la figure 2. Au fond cette ambiguïté n'a ici aucune importance, car l'opération de concaténation des mots est associative. Il en résulte que la concaténation des motifs est également associative. Comme

Fig. 2 – Deux arbres possibles pour `abc`



nous allons le voir immédiatement, il en va de même pour l'alternative.

1.3 Sémantique des expressions régulières

Une expression arithmétique a une valeur (plus généralement on dit aussi une sémantique) : c'est tout simplement l'entier résultat du calcul. De même, une expression régulière a une valeur qui est ici un ensemble de mots. Il est logique que la définition de la sémantique reprenne la structure de la définition des expressions régulières. C'est tout simplement une définition inductive, qui à chaque expression régulière p associe un sous-ensemble $[\![p]\!]$ de Σ^*.

$$\begin{array}{rcl} [\![\epsilon]\!] & = & \{\epsilon\} \\ [\![c]\!] & = & \{c\} \\ [\![p_1 \mid p_2]\!] & = & [\![p_1]\!] \cup [\![p_2]\!] \\ [\![p_1 \cdot p_2]\!] & = & [\![p_1]\!] \cdot [\![p_2]\!] \\ [\![p*]\!] & = & [\![p]\!]^* \end{array}$$

Cette définition a le parfum usuel des définitions de sémantique, on a presque rien écrit en fait ! Tout a déjà été dit ou presque dans la section 1.1 sur les mots. Lorsque m appartient au langage défini par p, on dit aussi que le motif p filtre le mot m. Un langage qui peut être défini comme la valeur d'une expression régulière est dit *langage* régulier.

Exemple 1 Nous savons donc que le langage des mots du français selon l'Académie est régulier, puisque qu'il est défini par une grosse alternative de tous les mots du dictionnaire publié par cette institution. Les représentations décimales des entiers sont également un langage régulier défini par :

```
0|(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*
```

Exercice 1 Montrer qu'un langage qui contient un nombre fini de mots est régulier.

Solution. Si $L = \{m_1, m_2, \ldots, m_n\}$ est un langage fini contenant n mots, alors L est exactement décrit par l'alternative de tous les m_i. □

1.4 Filtrage

La relation de filtrage $m \in [\![p]\!]$, également notée $p \preceq m$, peut être définie directement par les règles de la figure 3. La définition de $p \preceq m$ est faite selon le formalisme des règles d'inférence,

FIG. 3 – Définition de la relation p filtre m

$$\textsc{Empty}\quad \epsilon \preceq \epsilon \qquad \textsc{Char}\quad c \preceq c \qquad \textsc{Seq}\quad \frac{p_1 \preceq m_1 \qquad p_2 \preceq m_2}{p_1 p_2 \preceq m_1 m_2} \qquad \textsc{OrLeft}\quad \frac{p_1 \preceq m}{p_1 \mid p_2 \preceq m} \qquad \textsc{OrRight}\quad \frac{p_2 \preceq m}{p_1 \mid p_2 \preceq m}$$

$$\textsc{StarEmpty}\quad p* \preceq \epsilon \qquad \textsc{StarSeq}\quad \frac{p \preceq m_1 \qquad p* \preceq m_2}{p* \preceq m_1 m_2}$$

qui est très courant. Il permet la définition inductive d'un prédicat. Les règles se décomposent en axiomes (par ex. EMPTY) qui sont les cas de base de la définition, et en règles d'inférence proprement dites qui sont les cas inductifs de la définition. Les règles sont à comprendre comme des implications : quand toutes les prémisses (au dessus du trait) sont vraies, alors la conclusion (au dessous du trait) est vraie. En enchaînant les règles on obtient une preuve effective du prédicat, appelée *arbre de dérivation*. Voici par exemple la preuve de $\mathtt{ab*|ba*} \preceq \mathtt{baa}$.

$$\dfrac{\dfrac{\mathtt{b} \preceq \mathtt{b} \qquad \dfrac{\mathtt{a} \preceq \mathtt{a} \qquad \dfrac{\mathtt{a} \preceq \mathtt{a} \qquad \mathtt{a*} \preceq \epsilon}{\mathtt{a*} \preceq \mathtt{a}}}{\mathtt{a*} \prec \mathtt{aa}}}{\mathtt{ba*} \preceq \mathtt{baa}}}{\mathtt{ab*|ba*} \preceq \mathtt{baa}}$$

Nous disposons donc de deux moyens d'exprimer un langage régulier, $m \in [\![p]\!]$ ou $p \preceq m$. Selon les circonstances, il est plus facile d'employer l'un ou l'autre de des moyens. Par exemple, pour montrer qu'un motif filtre un mot, les règles d'inférences sont souvent commodes. En revanche, pour montrer des égalités de langages réguliers, il est souvent plus commode de se servir des opérations sur les langages.

Exercice 2 Prouver que pour tout motif p, $p\mathtt{*}$ et $p\mathtt{**}$ définissent le même langage.

Solution. Pour tout langage on a $(L^\star)^\star = L^\star$ par définition de l'opération de répétition sur les langages. Prouver l'équivalence $p\mathtt{*} \preceq m \iff p\mathtt{**} \preceq m$ est plus pénible. □

2 Notations supplémentaires

En pratique on rencontre diverses notations qui peuvent toutes être exprimées à l'aide des constructions de base des expressions régulières. Autrement dit, l'emploi de ces notations n'augmente pas la classe des langages réguliers, mais permet simplement d'écrire des expressions régulières plus compactes.

- Le motif optionnel $p\,\mathtt{?}$ défini comme $p\,\mathtt{|}\,\epsilon$.
- La répétition au moins une fois $p\mathtt{+}$ définie comme $pp\mathtt{*}$.
- Le joker, noté `.`, qui est l'alternative de tous les caractères de Σ. On a donc $[\![\mathtt{.}]\!] = \Sigma$.

Exemple 2 Ainsi une notation plus conforme à ce que les langages informatiques acceptent en fait d'entiers décimaux est :

```
(0|1|2|3|4|5|6|7|8|9)+
```

C'est-à-dire une suite non-vide de chiffres décimaux, sans chercher à éviter d'éventuels zéros initiaux. Une notation possible pour les entiers relatifs est

```
-?(0|1|2|3|4|5|6|7|8|9)+
```

C'est-à-dire un entier naturel précédé éventuellement d'un signe moins.

Il existe également toute une variété de notations pour des ensembles de caractères. Ces motifs sont des abréviations de l'alternative notées entre crochets `[...]`. Donc, au sein de ces crochets, on trouve une suite des *classes de caractères* suivantes :

- Un caractère c se représente lui-même.
- Un intervalle c_1 `-` c_2 représente les caractères dont les codes sont compris au sens large entre les codes de c_1 et c_2.

La notation `[^...]` se comprend comme le complémentaire des classes de caractères données. Il est clair que les notations de classes de caractères peuvent s'exprimer comme des alternatives de motifs caractères. Dans le cas du complémentaire, cela suppose un alphabet Σ fini, ce qui est bien évidemment le cas en pratique.

Exemple 3 Une notation compacte pour les entiers décimaux est `[0-9]+`. Cela fonctionne parce que dans tous les codages raisonnables les codes des chiffres sont consécutifs. De même, puisqu'en Java par exemple (en ASCII en fait) les codes des lettres sont consécutifs, l'expression régulière `[a-zA-Z]` représente toutes les lettres minuscules et majuscules (non-accentuées), Tandis que `[^a-zA-Z]` représente tous les caractères qui ne sont pas des lettres.

Exercice 3 Donner une expression régulière compacte qui décrit la notation adoptée pour les entiers par Java. À savoir, notation décimale (sans zéro initiaux), hexadécimale (introduite par `0x`, les chiffres de 10 à 15 étant représentes par les lettres minuscules ou majuscules de `a` à `f`), et notation octale (introduite par un zéro).

Solution. Il y a peu à dire à part :

```
(0|[1-9][0-9]*)|0x[0-9a-fA-F]+|0[0-7]+
```

On note que, selon l'expression donnée, `0` est décimal, `00` octal, et `09` interdit. □

Enfin certains caractères spéciaux sont exprimables à l'aide de notations assez usuelles, du style `\n` est le caractère fin de ligne et `\t` est la tabulation etc. Le caractère barre oblique inverse (*backslash*) introduit les notations de caractères spéciaux, mais permet aussi d'exprimer les caractères actifs des expressions régulières comme eux-mêmes, c'est à dire de les citer. Par exemple le caractère étoile se note `\*` et, bien évidemment, le caractère barre oblique inverse se note `\\`. Il n'est pas l'usage de citer les caractères quand c'est inutile, ainsi `[^*]` représente tous les caractères sauf l'étoile.

Exemple 4 Nous sommes désormais en possession d'un langage de description des ensembles de mots assez puissant. Par exemple voici le motif qui décrit une des deux sortes de commentaires Java.

```
//[^\n]*\n
```

C'est-à dire qu'un commentaire commence par deux `/` et se poursuit jusqu'à la fin de la ligne courante. Notons que le motif `//.*\n` ne convient pas, car il filtre par exemple :

```
//Commentaire avant
i++ ;
```

Exercice 4 Donner une expression régulière qui décrit l'autre sorte de commentaires de Java. À savoir, un commentaire commence par `/*` et s'étend jusqu'à `*/`.

Solution. On vise un motif de la forme `/\*`p`\*/`.Autrement dit, les commentaires commencent par le sous-mot `/*` et se terminent par le sous-mot `*/`, et, comme on doit identifier le premier sous-mot `*/` pour fermer le commentaire, le motif p doit correspondre au langage P de tous les mots dont `*/` n'est pas un sous-mot. Notons tout de suite qu'un mot de P peut se terminer par une suite d'étoiles, on a donc $p = q$`\**`. Soit Q le langage du motif q, en définissant Q comme l'ensemble des mots dont aucun sous-mot n'est `*/` et qui ne se terminent pas par une suite non-vide d'étoiles. Nous donnons ensuite à q la forme $q = r$`*`, c'est un moyen simple de prendre en compte que Q contient des mots de longueur arbitraire. Il faut maintenant decomposer un mot (non-vide) m comme un préfixe appartenant à R (langage de r) et un suffixe m', tels que m est dans Q, si et seulement si m' est dans Q.

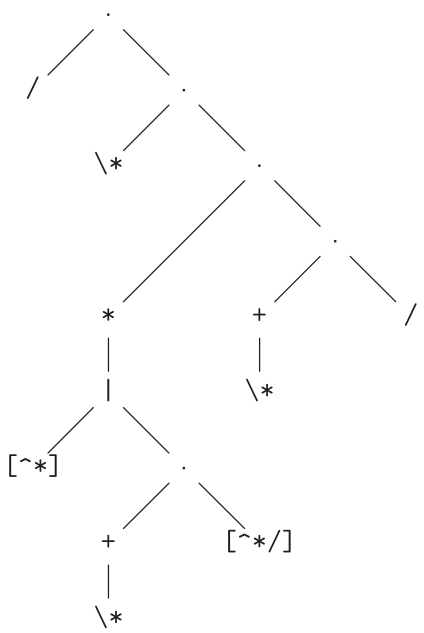
- Si m commence par un caractère c qui n'est pas l'étoile, alors m se décompose facilement comme $m = cm'$, et l'équivalence est évidente.
- Si m commence par une étoile. Commençons par supposer $m \in Q$, alors cette étoile peut être suivie d'autres, et obligatoirement d'un caractère qui n'est ni `*` ni `/`, ensuite on trouve le suffixe m'. Autrement dit, on pose un préfixe décrit par `\*+[^*/]`. Il est alors clair que $m' \in Q$. Réciproquement si, $m' \in Q$, alors le dernier caractère du préfixe n'est pas une étoile, et on ne peut pas introduire le sous-mot `*/` en y ajoutant m', même si m' commence par `/`.

Soit r = `[^*]|\*+[^*/]`. Au final, la solution est :

```
/\*([^*]|\*+[^*/])*\*+/
```

La syntaxe concrète est celle introduite dans les sections précédentes. L'arbre de syntaxe abstraite est donné à la figure 4. □

FIG. 4 – Syntaxe abstraite de `/\*([^*]|\*+[^*/])*\*+/`



3 Programmation avec les expressions régulières

Dans cette section nous nous livrons à une description rapide de quelques outils qui emploient les expressions régulières de façon cruciale. Le sujet est extrêmement vaste, car le formalisme des expressions régulières est suffisamment expressif pour permettre de nombreuses recherches dans les fichiers textes, voire de nombreuses transformations de ces fichiers. Le livre de Jeffrey E. F. Friedl [3] traite des expressions régulière du point de vue des outils.

3.1 Dans l'interprète de commandes Unix

Le *shell* c'est à dire l'interprète de commandes Unix reconnaît quelques expressions régulières qu'il applique aux noms des fichiers du répertoire courant. La syntaxe concrète des motifs est franchement particulière. Notons simplement que `?` représente un caractère quelconque (noté précédemment `.`), tandis que `*` représente un mot quelconque (noté précédemment `.*`), et que l'alternative se note à peu près `{`p_1`,`p_2`}` (pour $p_1 \mid p_2$)...

Ainsi, on pourra effacer tous les fichiers objets Java par la commande :

```
% /bin/rm *.class
```

Dans le même ordre d'idée, on peut compter toutes les lignes des fichiers source du répertoire courant :

```
% wc -l *.java
```

La commande `wc`[1] (pour *word count*) compte les lignes, mots et caractères dans les fichiers donnés en argument. L'option « `-l` » restreint l'affichage au seul compte des lignes. Enfin, on peut faire la liste de tous les fichiers du répertoire courant dont le nom comprend de un à trois caractères :

```
% ls {?,??,???}
```

3.2 Recherche de lignes dans un fichier, la commande `egrep`

La commande `egrep` *motif fichier* affiche toutes les lignes de *fichier* dont *motif* filtre un *sous-mot* (et non pas la ligne entière). La syntaxe des motifs est relativement conforme à ce que nous avons déjà décrit. Supposons, comme c'est souvent le cas, que le fichier `/usr/share/dict/french` de votre machine Unix est un dictionnaire français, donné sous forme d'un fichier texte, à raison d'un mot par ligne. On peut alors trouver les mots qui contiennent au moins six fois la lettre « i » de cette manière.

```
% egrep 'i.*i.*i.*i.*i.*i' /usr/share/dict/french
indivisibilité
```

On notera que l'argument *motif* est donné entre *simples quotes* « `'` », ceci afin d'éviter que le shell n'interprète les étoiles comme faisant partie d'un motif à appliquer aux noms de fichier. Ce n'est en fait pas toujours nécessaire, mais c'est toujours prudent.

3.3 En Java

Le support pour les expressions régulières est assuré par les classes **Pattern** et **Matcher** du package `java.util.regexp`, Les objets de la classe **Pattern** implémentent les motifs. et sont construits par la méthode statique **Pattern**`.compile` qui prend une chaîne représentant un motif en argument et renvoie un **Pattern**. On pourrait penser que la méthode `compile` interprète la chaîne donnée en argument et produit un arbre de syntaxe abstraite. En fait, par souci d'efficacité, elle procède à bien plus de travail, jusqu'à produire un *automate* (voir le chapitre suivant). Pour le moment, il suffit de considérer qu'un **Pattern** est la forme Java d'un motif.

Pour confronter un **Pattern** p à un mot m il faut fabriquer cette fois un **Matcher** en invoquant la méthode `matcher(`**String** `text)` de p, l'argument `text` étant le mot m. En simplifiant, le **Matcher** obtenu est donc la combinaison d'un motif p et d'un mot m, il offre (entre autres !) les méthodes `matches()` et `find()` qui renvoient des booléens. La première méthode `matches` teste si le motif p filtre le mot m, tandis que la seconde `find` teste si le motif p filtre un sous-mot du mot m. Ainsi, par exemple, un moyen assez compliqué de savoir si un mot `mot` contient le sous-mot `sous` au moins deux fois est d'écrire :

[1] On obtient le détail du fonctionnement d'une commande Unix *cmd* par `man` *cmd*.

```
static boolean estSousMotDeuxFois(String sous, String mot) {
  return Pattern.compile(sous + ".*" + sous).matcher.().find(mot) ;
}
```

Nous en savons maintenant assez pour pouvoir écrire la commande `egrep` en Java, la classe **Grep** dont le code est donné à la figure 5. Dans ce code, la méthode `main` se comporte principalement ainsi : elle ouvre le fichier dont le nom est donné comme second argument de la ligne de commande, par **new FileReader**`(arg[1])` ; puis enveloppe le fichier comme le **BufferedReader** (voir B.6.2.2) `in`, ceci afin de pouvoir le lire ligne à ligne ; enfin le code appelle la méthode `grep`. Cette dernière, après construction du **Pattern** `pat`, l'applique à toutes les lignes du fichier. En cas de succès de l'appel à `find`, la ligne est affichée sur la sortie standard. Le comportement global est donc bien celui de la commande `egrep`.

L'architecture du package `java.util.regex` peut paraître bien compliquée, et c'est tout à vrai. Mais...

- L'existence de la classe **Pattern** se justifie d'abord par le souci d'abstraction : les auteurs ne souhaitent pas exposer comment sont implémentés les motifs, afin de pouvoir changer cette implémentation dans les versions futures de Java. Il y a également un souci d'efficacité, la transformation des chaînes vers les motifs est coûteuse et on souhaite la rentabiliser. Par exemple dans notre **Grep** (figure 5), nous appelons **Pattern**`.compile` une seule fois et pratiquons de nombreux filtrages.
- L'existence de la classe **Matcher** s'explique autrement : les **Matcher** possèdent un état interne que les méthodes de filtrage modifient. Ceci permet d'abord d'obtenir des informations supplémentaires. Par exemple, si `find` réussit, alors on obtient le sous-mot filtré en appelant la méthode `group()`. Par ailleurs, l'appel suivant à `find` recherchera un sous-mot filtré, non plus à partir du début, mais au delà du dernier sous-mot identifié par `find`. Tout cela permet par exemple d'extraire tous les entiers présents dans une chaîne.

  ```
  static void allInts(String text) {
    Matcher m = Pattern.compile("[0-9]+").matcher(text) ;
    while (m.find()) {
      System.out.println(m.group()) ;
    }
  }
  ```

  On notera qu'il n'est pas immédiat que le code ci-dessus affiche bien tous les entiers de la chaîne `text`. En effet si `text` est par exemple `"12"` les deux affichages `12`, ou encore `1` puis `2` sont corrects : il n'affichent que des sous-mots filtrés par le motif `"[0-9]+"`.
  Plus généralement, spécifier complètement ce que fait le couple de méthode `find`/`group` est un peu délicat. La solution à mon avis la plus satisfaisante est spécifier que le sous-mot filtré est d'abord le plus à gauche et ensuite le le plus long. La spécification de Java n'est pas aussi générale : au lieu de dire globalement ce qui doit être filtré, elle décrit l'effet de chaque opérateur des expressions régulière individuellement. Ici elle dit que l'opérateur « `+` » est *avide* (*greedy*), c'est à dire qu'il filtre le plus de caractères possibles. Dans le cas de l'expression régulière simple `"[0-9]+"`, cela revient en effet à filtrer les sous-mots les plus longs.

Nous ne décrirons pas plus en détail les expressions régulières de Java. La documentation du langage offre de nombreuses précisions. En particulier, la documentation de la classe **Pattern** comprend la description complète de la syntaxe des motifs qui est plus étendue que ce que nous avons vu ici. Attention tout de mêmes les descriptions de motifs sont données dans l'absolu, alors que les motifs sont souvent en pratique dans des chaînes. If faut donc en plus tenir compte des règles de citation dans les chaînes. Ainsi le motif `\p{L}` filtre n'importe quelle lettre (y compris les lettres accentuées) mais on le donne sous la forme de la chaîne `"\\p{L}"`, car il faut citer un backslash avec un backslash !

Fig. 5 – La commande `egrep` en Java, source `Grep.java`

```
import java.io.* ;          // Pour  BufferedReader
import java.util.regex.* ; // Pour Pattern et Matcher

class Grep {

  // Affiche les lignes de in dont un sous-mot est filtré par le motif p
  static void grep(String p, BufferedReader in) throws IOException {
    Pattern pat = Pattern.compile(p) ;
    String line = in.readLine() ;
    while (line != null) {
      Matcher m = pat.matcher(line) ;
      if (m.find()) {
        System.out.println(line) ;
      }
      line = in.readLine() ;
    }
  }

  public static void main(String [] arg) {
    if (arg.length != 2) {
      System.err.println("Usage: java Grep motif fichier") ;
      System.exit(2) ;
    }
    try {
      BufferedReader in = new BufferedReader (new FileReader (arg[1])) ;
      grep(arg[0], in) ;
      in.close() ;
    } catch (IOException e) {
      System.err.println("Malaise: " + e.getMessage()) ;
      System.exit(2) ;
    }
  }
}
```

4 Implémentation des expressions régulières

Le but de cette section est de décrire une technique d'implémentation possible des expressions régulières en Java. Il s'agit d'une première approche, beaucoup moins sophistiquée que celle adoptée notamment par la bibliothèque Java. Toutefois, on pourra, même avec des techniques simples, déjà aborder les problèmes de programmation posés et comprendre « *comment ça marche* ». De fait nous allons imiter l'architecture du package `java.util.regexp` et écrire nous aussi un package que nous appelons `regex` tout court.

Nous en profitons donc pour écrire un package. Tous les fichiers source du package `regex` commencent par la ligne **package** `regex ;` qui identifie leur classe comme appartenant à ce package. En outre, il est pratique de regrouper ces fichiers dans un sous-répertoire nommé justement `regex`.

4.1 Arbres de syntaxe abstraite

Nous reprenons les techniques de la section IV.4 sur les arbres de syntaxe abstraite. À savoir nous définissons une classe Re des nœuds de l'arbre de syntaxe abstraite.

```
package regex ;

class Re {
  private final static int  EMPTY=0, CHAR=1, WILD=2, OR=3, SEQ=4, STAR=5 ;

  private int tag ;
  private char asChar ;
  private Re p1, p2 ;

  private Re() {}
  ⋮
}
```

Nous définissons cinq sortes de nœuds, la sorte d'un nœud étant identifiée par son champ `tag`. Des constantes nommées identifient les cinq sortes de nœuds. La correspondance entre constante et sorte de nœud est directe, on note la présence de nœuds « `WILD` » qui représentent les jokers. Ensuite nous définissons tous les champs nécessaires, un champ `asChar` utile quand le motif est un caractère (tag `CHAR`), et deux champs `p1` et `p2` utiles pour les nœuds internes qui ont au plus deux fils. Enfin, le constructeur par défaut est redéfini et déclaré privé.

On construira les divers nœuds en appelant des méthodes statiques bien nommées. Par exemple, pour créer un motif caractère, on appelle :

```
static Re charPat(char c) { // On ne peut pas nommer cette méthode « char »
  Re r = new Re() ;
  r.asChar = c ;
  return r ;
}
```

Pour créer un motif répétition, on appelle :

```
static Re star(Re p) {
  Re r = new Re() ;
  r.p1 = p ;
  return p ;
}
```

Les autres autres méthodes de construction sont évidentes.

Les méthodes statiques de construction ne se limitent évidemment pas à celles qui correspondent aux sortes de nœuds existantes. On peut par exemple écrire facilement une méthode `plus` qui construit un motif p+ comme pp*.

```
static Re plus(Re p) {
  return seq(p, star(p)) ;
}
```

Du point de vue de l'architecture, on peut remarquer que tous les champs et le constructeur sont privés. Rendre le constructeur privé oblige les utilisateurs de la classe Re appeler les méthodes statiques de construction, de sorte qu'il est garanti que tous les champs utiles dans un nœud sont correctement initialisés. Rendre les champs privés interdira leur accès de l'extérieur de la classe Re. Au final, la politique de visibilité des noms est très stricte. Elle renforce la sécurité de la programmation, puisque si nous ne modifions pas les champs dans la classe Re, nous pourrons être sûrs que personne ne le fait. En outre, la classe Re n'est pas publique, son accès est donc limité aux autres classes du package `regex`. La classe Re est donc complètement invisible pour les utilisateurs du package.

4.2 Fabrication des expressions régulières

Nous présentons maintenant notre classe Pattern, un modeste remplacement de la classe homonyme de la bibliothèque Java. Pour le moment nous évitons les automates et nous contentons de cacher un arbre Re dans un objet de la classe Pattern.

```
package regex ;

/* Une classe Pattern simple : encapsulage d'un arbre de syntaxe abstraite */

public class Pattern {
  private Re pat ;

  private Pattern(Re pat) { this.pat = pat ; }

  // Chaîne -> Pattern
  public static Pattern compile(String patString) {
    Re re = Re.parse(patString) ;
    return new Pattern(re) ;
  }

  // Fabriquer le Matcher
  public Matcher matcher(String text) { return new Matcher(pat, text) ; }
}
```

Comme dans la classe de la bibliothèque, c'est la méthode statique `compile` qui appelle le constructeur, ici privé. La partie la plus technique de la tâche de la méthode `compile` est le passage de la syntaxe concrète contenue dans la chaîne `patString` à la syntaxe abstraite représenté par un arbre Re, opération déléguée à la méthode Re`.parse`. Nous ne savons pas écrire cette méthode *d'analyse syntaxique* (*parsing*). (cours INF 431). Mais ne soyons pas déçus, nous pouvons déjà par exemple construire le motif qui reconnaît au moins `k` caractères `c`, en appelant la méthode `atLeast` suivante, à ajouter dans la classe Pattern.

```
public static Pattern atLeast(int k, char c) {
  return new Pattern(buildAtLeast(k, c)) ;
}

private static Re buildAtLeast(int k, char c) {
```

```
  if (k <= 0) {
    return Re.empty() ;
  } else if (k == 1) {
    return Re.charPat(c) ;
  } else {
    return
      Re.seq
        (Re.charPat(c),
         Re.seq(Re.star(Re.wild()), buildAtLeast(k-1, c)))
  }
}
```

Enfin, la méthode `matcher` de de la classe **Pattern** se contente d'appeler le constructeur de notre modeste classe **Matcher**, que nous allons décrire.

4.3 Filtrage

Le source de la classe **Matcher** (figure 6) indique que les objets contiennent deux champs `pat` et `text`, pour le motif et le texte à filtrer. Comme on pouvait s'y attendre, le constructeur `Matcher(Re pat, String text)` initialise ces deux champs. Mais les objets comportent trois champs supplémentaires, `mStart`, `mEnd` et `regStart`.

- La valeur du champ `regStart` indique l'indice dans `text` du début de la recherche suivante, c'est-à-dire où la méthode `find` doit commencer à chercher une sous-chaîne filtrée par `pat`. Ce champ permet donc aux appels successifs de `find` de communiquer entre eux.
- Les champs `mStart` et `mEnd` identifient la position de la dernière sous-chaîne de `text` dont un appel à `find` a déterminé que le motif `pat` la filtrait. La convention adoptée est celle de la méthode `substring` des objets **String** (voir la section B.6.1.3). Les deux champs servent à la communication entre un appel à `find` et un appel subséquent à `group` (voir la fin de la section 3.3).

La méthode `find` est la plus intéressante, elle cherche à identifier une sous-chaîne filtrée par `pat`, à partir de la position `regStart` et de la gauche vers la droite. La technique adoptée est franchement naïve, on essaie tout simplement de filtrer successivement toutes les sous-chaînes commençant à une positon donnée (`start`) des plus longues à la chaîne vide. On renvoie **true** (après mise à jour de l'état du **Matcher**), dès qu'une sous-chaîne filtrée est trouvée. Pour savoir si une sous-chaîne `text[start...end[` est filtrée, on fait appel à la méthode statique `Re.matches`. Notons que c'est notre parti-pris de rendre privés tous les champs de l'arbre de syntaxe des expressions régulière qui oblige à écrire toute méthode qui a besoin d'examiner cette structure comme une méthode de la classe **Re**.

Exercice 5 Écrire la méthode `matches` de la classe **Matcher**. On suivra la spécification de la classe **Matcher** de la bibliothèque. À savoir, l'appel `matches()` teste le filtrage de toute l'entrée par le motif et on peut utiliser `group()` pour retrouver la chaîne filtrée.

Solution. C'est simple : un appel à `Re.matches` et on affecte les champs `mStart` et `mEnd` selon le résultat.

```
public boolean matches() {
  if (Re.matches(text, pat, 0, text.length())) {
     mStart = 0 ;
     mEnd = text.length() ;
     return true ;
  } else {
     mStart = mEnd = -1 ;
```

Fig. 6 – Notre classe Matcher

```
package regex ;

public class Matcher {
  private Re pat ;
  private String text ;

  // Les recherches commencent à cette position dans text
  private int regStart ;
  // La dernière sous-chaîne filtrée est text[mStart...mEnd[
  private int mStart, mEnd ;

  Matcher(Re pat, String text) {
    this.pat = pat ; this.text = text ;
    regStart = 0 ;       // Commencer à filtrer à partir du début
    mStart = mEnd = -1 ; // Aucun motif encore reconnu
  }

  // Renvoie la dernière sous-chaîne filtrée, si il y a lieu
  public String group() {
    if (mStart == -1) throw new Error("Pas de sous-chaîne filtrée") ;
    return text.substring(mStart, mEnd) ;
  }

  // Méthode de recherche des sous-chaînes filtrées a peu près
  // conforme à celle des Matcher de java.util.regex
  public boolean find() {
    for (int start = regStart ; start <= text.length() ; start++)
      for (int end = text.length() ; end >= start ; end--) {
        if (Re.matches(text, pat, start, end)) {
          mStart = start ; mEnd = end ;
          regStart = mEnd ; // Le prochain find commencera après celui-ci
          return true ;
        }
      }
    mStart = mEnd = -1 ; // Pas de sous-chaîne reconnue
    regStart = 0 ;       // Recommencer au début, bizarre
    return false ;
  }
}
```

```
      return false ;
  }
}
```

□

Pour écrire la méthode `matches` de la classe **Re**, nous allons distinguer les divers motifs possibles et suivre la définition de $p \preceq m$ de la figure 3.

```
// Test  de pat ⪯ text[i...j[
static boolean matches(String text, Re pat, int i, int j) {
  switch (pat.tag) {
     ⋮
  }
  throw new Error ("Arbre Re incorrect") ;
}
```

Notons bien que `text[i...j[` est la chaîne dont nous cherchons à savoir si elle est filtrée par `pat`. La longueur de cette chaîne est `j-i`. Nous écrivons maintenant le source du traitement des cinq sortes de motifs possibles, c'est à dire la liste des cas du **switch** ci-dessus. Le cas des motifs vide, des caractères et du joker est rapidement réglé.

```
    case EMPTY:
      return i == j ;
    case CHAR:
      return i+1 == j && text.charAt(i) == pat.asChar ;
    case WILD:
      return i+1 == j ;
```

En effet, le motif vide filtre la chaîne vide et elle seule ($j - i = 0$), le motif caractère ne filtre que la chaîne composée de lui même une fois, et le joker filtre toutes les chaînes de longueur un.

Le cas de l'alternative est également assez simple, il suffit d'essayer les deux termes de l'alternative (regles OrLeft et OrRight).

```
    case OR:
      return matches(text, pat.p1, i, j) || matches(text, pat.p2, i, j) ;
```

La séquence (rule Seq) demande plus de travail. En effet il faut essayer *toutes* les décompositions en préfixe et suffixe de la chaîne testée, faute de quoi nous ne pourrions pas renvoyer **false** avec certitude.

```
    case SEQ:
      for (int k = i ; k <= j ; k++) {
        if (matches(text, pat.p1, i, k) && matches(text, pat.p2, k, j))
          return true ;
      }
      return false ;z
```

Et enfin, le cas de la répétition $q*$ est un peu plus subtil, il est d'abord clair (règle StarEmpty) qu'un motif $q*$ filtre toujours la chaîne vide. Si la chaîne `text[i...j[` est non-vide alors on cherche à la décomposer en préfixe et suffixe et à appliquer la règle StarSeq.

```
    case STAR:
      if (i == j) {
        return true ;
      } else {
        for (int k = i+1 ; k <= j ; k++) {
          if (matches(text, pat.p1, i, k) && matches(text, pat, k, j))
            return true ;
```

```
    }
    return false ;
  }
```

On note un point un peu subtil, dans le cas d'une chaîne non-vide, on évite le cas `k = j` qui correspond à une division de la chaîne testée en préfixe vide et suffixe complet. Si tel n'était pas le cas, la méthode `matches` pourrait ne pas terminer. En effet, le second appel récursif `matches(text, pat, k, j)` aurait alors les mêmes arguments que lors de l'appel. Un autre point de vue est de considérer que l'application de la règle StarSeq à ce cas est inutile, dans le sens qu'on ne risque pas de ne pas pouvoir prouver $q* \preceq m$ parce que l'on abstient de l'employer.

$$\frac{q \preceq \epsilon \qquad q* \preceq m}{q* \preceq m}$$

L'inutilité de cette règle est particulièrement flagrante, puisqu'une des prémisses et la conclusion sont identiques.

4.4 Emploi de notre package `regex`

Nos classes **Pattern** et **Matcher** sont suffisamment proches de celles de la bibliothèque pour que l'on puisse, dans le source `Grep.java` (figure 5), changer la ligne **import** `java.util.regex.*` en **import** `regex.*`, ce qui nous donne le nouveau source `ReGrep.java`. Dès lors, à condition que le source des classes du package `regex` se trouve dans un sous-répertoire `regex`, nous pouvons compiler par `javac ReGrep.java` et nous obtenons un nouveau programme **ReGrep** qui utilise notre implémentation des expressions régulières à la place de celle de la bibliothèque.

Nous nous livrons ensuite à des expériences en comparant les temps d'exécution (par les commandes `time java Grep` ... et `time java ReGrep` ...).

(1) Dans le dictionnaire français, nous recherchons les mots qui contiennent au moins n fois la même voyelle non accentuée. Par exemple, pour $n = 3$ nous exécutons la commande :

```
% java Grep '(a.*a.*a|e.*e.*e|i.*i.*i|o.*o.*o|u.*u.*u)' /usr/share/dict/french
```

	1	2	3	4	5	6
Grep	2.7	2.5	1.9	1.7	1.6	1.5
ReGrep	3.1	9.7	16.4	17.9	18.6	18.0

On voit que notre technique, sans être ridicule, est nettement moins efficace.

(2) Toujours dans le dictionnaire français, nous recherchons les mots qui contiennent n fois la lettre `e`, après effacement des accents. Par exemple, pour $n = 3$ nous exécutons la commande :

```
% java Grep '(e|é|è|ê).*(e|é|è|ê).*(e|é|è|ê)' /usr/share/dict/french
```

	1	2	3	4	5	6	7
Grep	2.9	2.2	1.9	1.7	1.6	1.5	1.5
ReGrep	3.1	9.0	12.8	15.2	15.8	16.0	15.9

Cette expérience donne des résultats similaires à la précédente. Plus précisément d'une part la bibliothèque est plus rapide en valeur absolue ; et d'autre part, nos temps d'exécution sont croissants, tandis que ceux de la bibliothèque sont décroissants. Mais et c'est important, il semble bien que les temps se stabilisent dans les deux cas.

(3) Nous recherchons une sous-chaîne filtrée par `X(.+)+X` dans la chaîne `XX=···=`, où `=···=` est le caractère `=` répété n fois. Cette reconnaissance doit échouer, mais nous savons [3, Chapitre 5] qu'elle risque de mettre en difficulté l'implémentation de la bibliothèque.

	16	18	20	22	24
Grep	0.3	0.5	1.3	4.8	18.6
ReGrep	0.2	0.2	0.2	0.2	0.2

Et effectivement l'emploi de la bibliothèque conduit à un temps d'exécution manifestement exponentiel. Il est fascinant de constater que notre implémentation ne conduit pas à cette explosion du temps de calcul.

5 Une autre approche du filtrage

5.1 Langage dérivé

Soit L un langage sur les mots et c un caractère. Nous notons $c^{-1} \cdot L$ le langage dérivé, défini comme les suffixes m des mots de L qui sont de la forme $c \cdot m$.

$$c^{-1} \cdot L = \{m \mid c \cdot m \in L\}$$

Dans le cas d'un langage L régulier engendré par le motif p, nous allons montrer que le langage $c^{-1} \cdot L$ est régulier en calculant le motif $c^{-1} \cdot p$ qui l'engendre.

On observera d'abord que, en toute rigueur, le langage $c^{-1} \cdot L$ peut être vide, par exemple si $L = \{ c' \}$ avec $c' \neq c$. Or, selon notre définition des langages réguliers, le langage vide $\emptyset$ n'est pas régulier. Qu'à cela ne tienne, nous inventons immédiatement un nouveau motif $\emptyset$, qui ne filtre aucun mot. En considérant les valeurs $[\![p]\!]$, on étend facilement les trois opérateurs des expressions régulières.

$$\emptyset \cdot p = \emptyset \qquad p \cdot \emptyset = \emptyset \qquad \emptyset \mid p = p \qquad p \mid \emptyset = p \qquad \emptyset{*} = \epsilon$$

Ces règles nous permettent d'éliminer les occurrences internes de $\emptyset$, de sorte qu'un motif est désormais $\emptyset$ ou un motif qui ne contient pas $\emptyset$.

Proposition 15 Si $L = [\![p]\!]$, alors le langage $c^{-1} \cdot L$ est régulier, engendré par le motif $c^{-1} \cdot p$ défini ainsi :

$$\begin{array}{rcll}
c^{-1} \cdot \emptyset & = & \emptyset & \\
c^{-1} \cdot \epsilon & = & \emptyset & \\
c^{-1} \cdot c & = & \epsilon & \\
c^{-1} \cdot c' & = & \emptyset & \text{pour } c \neq c' \\
c^{-1} \cdot (p_1 \mid p_2) & = & c^{-1} \cdot p_1 \mid c^{-1} \cdot p_2 & \\
c^{-1} \cdot (p_1 \cdot p_2) & = & (c^{-1} \cdot p_1) \cdot p_2 & \text{si } p_1 \not\preceq \epsilon \\
c^{-1} \cdot (p_1 \cdot p_2) & = & (c^{-1} \cdot p_1) \cdot p_2 \mid c^{-1} \cdot p_2 & \text{si } p_1 \preceq \epsilon \\
c^{-1} \cdot p{*} & = & (c^{-1} \cdot p) \cdot p{*} &
\end{array}$$

Preuve. On montre, par induction sur la structure de p, que pour tout mot m, on a l'équivalence :

$$c^{-1} \cdot p \preceq m \iff p \preceq c \cdot m$$

Seuls les trois derniers cas de la définition de $c^{-1} \cdot p$ présentent un intérêt.

- Posons $p = p_1 \cdot p_2$ et supposons en outre $p_1 \not\preceq \epsilon$. En revenant aux définitions, par induction, puis par définition, on a les équivalences successives :

$$p_1 \cdot p_2 \preceq c \cdot m \iff \bigwedge \left\{ \begin{array}{l} m = m_1 \cdot m_2 \\ p_1 \preceq c \cdot m_1 \\ p_2 \preceq m_2 \end{array} \right. \iff \bigwedge \left\{ \begin{array}{l} m = m_1 \cdot m_2 \\ c^{-1} \cdot p_1 \preceq m_1 \\ p_2 \preceq m_2 \end{array} \right. \iff (c^{-1} \cdot p_1) \cdot p_2 \preceq m$$

- Dans le cas où $p_1 \preceq \epsilon$, il faut tenir compte d'une autre décomposition possible du mot $c \cdot m$ en préfixe vide et suffixe complet. Dans ce cas on a par induction :

$$\bigwedge \left\{ \begin{array}{l} p_1 \preceq \epsilon \\ p_2 \preceq c \cdot m \end{array} \right. \iff \bigwedge \left\{ \begin{array}{l} p_1 \preceq \epsilon \\ c^{-1} \cdot p_2 \preceq m \end{array} \right.$$

  Ce qui conduit à l'équivalence :

$$p_1 \cdot p_2 \leq c \cdot m \iff \bigvee \left\{ \begin{array}{l} \bigwedge \left\{ \begin{array}{l} m = m_1 \cdot m_2 \\ c^{-1} \cdot p_1 \preceq m_1 \\ p_2 \preceq m_2 \end{array} \right. \\ \\ \bigwedge \left\{ \begin{array}{l} p_1 \preceq \epsilon \\ c^{-1} \cdot p_2 \preceq m \end{array} \right. \end{array} \right.$$

  Et on peut conclure.
- Posons $p = q*$. Alors on a :

$$q* \preceq c \cdot m \iff \bigwedge \left\{ \begin{array}{l} m = m_1 \cdot m_2 \\ q \preceq c \cdot m_1 \\ q* \preceq m_2 \end{array} \right.$$

  En toute rigueur, l'équivalence résulte d'un nombre arbitraire applications de la règle Star-Seq pour un préfixe vide et d'une application de la même règle pour la décomposition écrite ci-dessus. On peut ensuite conclure par induction.

□

Par ailleurs, on décide de la validité de $p \preceq \epsilon$ par une simple induction structurelle.

Lemme 16 On note $\mathcal{N}(p)$ le prédicat $p \preceq \epsilon$. Le prédicat $\mathcal{N}(p)$ se calcule inductivement ainsi :

$$\begin{array}{rcl} \mathcal{N}(\emptyset) & = & \mathit{faux} \\ \mathcal{N}(\epsilon) & = & \mathit{vrai} \\ \mathcal{N}(c) & = & \mathit{faux} \\ \mathcal{N}(p_1 \mid p_2) & = & \mathcal{N}(p_1) \vee \mathcal{N}(P_2) \\ \mathcal{N}(p_1 \cdot p_2) & = & \mathcal{N}(p_1) \wedge \mathcal{N}(P_2) \\ \mathcal{N}(p*) & = & \mathit{vrai} \end{array}$$

Preuve. Induction facile sur la structure des motifs. □

5.2 Filtrage par la dérivation des motifs

Pour savoir si un motif p filtre un mot m, on peut tout simplement itérer sur les caractères du mot, en calculant les dérivations successives de p par les caractères consommés. Un fois atteint la fin du mot, il ne reste qu'à vérifier si le motif dérivé filtre le mot vide.

Plus précisément, les deux résultats de la section précédente (proposition 15 et lemme 16) nous permettent d'écrire deux nouvelles méthodes statiques dans la classe Re.

```
// Renvoie le motif c^{-1} · p ou null si c^{-1} · p est ∅
static Re derivate(char c, Re p) ;

// Calculer N(p)
static boolean nullable(Re p) ;
```

Exercice 6 (Non corrigé, vu en TP) Programmer ces deux méthodes. On fera attention à la présence du motif $\emptyset$ qui sera représenté par **null** et ne se trouvera jamais à l'intérieur des arbres Re. C'est-à-dire que le résultat de `Re.derivate` est **null** ou un motif standard.

On peut alors écrire par exemple la méthode `matches` (simplifiée, sans tenir compte de `mStart` et `mEnd`) de la classe Matcher ainsi :

```
public boolean matches() {
  Re d = pat ;
  for (int k = 0 ; k < text.length() ; k++) {
    d = Re.derivate(text.charAt(k), d) ;
    if (d == null) return false ;
  }
  return Re.nullable(d) ;
}
```

Exercice 7 Récrire la méthode `find` de la classe Matcher (figure 6) en utilisant cette fois la dérivation de motifs. On s'efforcera de limiter le nombre des dérivations de motif effectuées, tout en identifiant la sous-chaîne filtrée la plus à gauche et la plus longue possible.

Solution. Une solution possible est d'introduire une méthode `findLongestPrefix` qui cherche le plus long préfixe filtré à partir de la position `start`. En cas de succès, la méthode renvoie la première position non-filtrée ; en cas d'échec, la méthode renvoie `-1`.

```
private int findLongestPrefix(int start) {
  Re d = pat ;
  int found = -1 ;
  if (Re.nullable(d)) found = start ; // Trouvé le préfixe vide
  for (int k = start ; k < text.length() ; k++) {
    d = Re.derivate(text.charAt(k), d) ;
    if (d == null) return found ;
    if (Re.nullable(d)) found=k+1 ; // Trouvé le préfixe de longueur k+1-start
  }
  return found ;
}
```

La méthode `findLongestPrefix` calcule simplement les motifs dérivés par tous les caractères de `text` à partir de la position `start`, en se souvenant (dans `found`) du filtrage le plus long. La recherche s'arrête quand la chaîne est lue en totalité ou quand le motif dérivé est $\emptyset$.

Écrire `find` est ensuite immédiat : chercher un préfixe filtré dans tous les suffixes possibles de `text`.

```
public boolean find() {
  for (int start = regStart ; start <= text.length() ; start++) {
    int end = findLongestPrefix(start) ;
    if (end >= 0) {
      mStart = start ; mEnd = end ;
      regStart = mEnd ;
      return true ;
    }
  }
  mStart = mEnd = -1 ;
  regStart = 0 ;
  return false ;
}
```

On observe que, pour une position `start` donnée, la nouvelle méthode `find` trouve bien la plus longue sous-chaîne filtrée. Une implantation qui donnerait la plus petite sous-chaîne filtrée est plus simple.

```
private int findShortestPrefix(int start) {
  Re d = pat ;
  if (Re.nullable(d)) return start ; // Trouvé le préfixe vide
  for (int k = start ; k < text.length() ; k++) {
    d = Re.derivate(text.charAt(k), d) ;
    if (d == null) return -1 ;
    if (Re.nullable(d)) return k+1 ;
  }
  return -1 ; // Aucun préfixe ne convient
}
```

□

La mesure des temps d'exécution de la nouvelle classe Matcher fait apparaître de meilleurs performances dans les deux premières expériences de la section 4.4 et une consommation très importante de la mémoire dans la dernière expérience.

Exercice 8 Calculer la succession des motifs dérivés pour le motif `X(.+)+X` et le mot `XX=`$\cdots$`=` et justifier le résultat de la dernière expérience.

Solution. Notons p_0, p_1 etc. la suite des motifs dérivés Il faut d'abord dériver deux fois par `X`.

$$p_0 = \texttt{X(.+)+X} \qquad p_1 = \texttt{(.+)+X} \qquad p_2 = \texttt{.*(..*)*X}$$

Pour calculer p_2 on a exprimé p`+` comme pp`*`. Si on suit la définition de la dérivation on a en toute rigueur des résultat différents — par exemple, $p_1 = \texttt{()(.+)+X}$. Mais on enlève les motifs `()` des séquences, les motifs sont déjà bien assez compliqués comme cela. Soit le motif $q = \texttt{.*(..*)*}$, on a $p_2 = q\texttt{X}$. Par ailleurs la dérivation $\texttt{=}^{-1} \cdot q$ vaut $q \mid q$. Le motif p filtre le mot vide, mais la dérivation de `X` par `=` vaut $\emptyset$. On a donc :

$$p_3 = (q \mid q) \cdot \texttt{X}$$

Et en posant $q_1 = q$ et $q_{n+1} = q_n \mid q_n$ il est clair que l'on a :

$$p_{n+2} = q_{n+1} \cdot \texttt{X}$$

Motif dont la taille est exponentielle en n. □

Chapitre VII

Les automates

1 Pourquoi étudier les automates

Ce chapitre est une très succincte introduction à la théorie des automates que vous aurez l'occasion de voir de façon détaillée si vous choisissez un cursus d'informatique. Ce chapitre est, par nature, un peu plus "théorique" et un peu moins algorithmique que les précédents.

Les automates sont des objets mathématiques, très utilisés en informatique, qui permettent de modéliser un grand nombre de systèmes (informatiques). L'étude des automates a commencé vers la fin des années cinquante. Elle se base sur de nombreuses techniques (topologie, théorie des graphes, logique, algèbre, *etc.*). De façon très informelle, un automate est un ensemble "d'états du système", reliés entre eux par des "transitions" qui sont marquées par des symboles. Étant donné un "mot" fourni en entrée, l'automate lit les symboles du mot un par un et va d'état en état selon les transitions. Le mot lu est soit accepté par l'automate soit rejeté.

Avant de donner une définition plus formelle des concepts décrits ci-dessus, citons quelques exemples classiques d'utilisation d'automates :

- Vérification d'un circuit électronique
- Recherche d'occurrence dans un texte (moteur de recherches sur le web, *etc.*)
- Vérification de protocoles de communication
- Compression de données
- Compilation
- Biologie (génomique)

En dehors de ces utilisations "pratiques" des automates, notons qu'ils sont aussi utilisés pour modéliser les ordinateurs et pour comprendre ce qu'un ordinateur peut faire (décidabilité) et ce qu'il sait faire efficacement (complexité). C'est donc une notion fondamentale de l'informatique.

2 Rappel : alphabets, mots, langages et problèmes

Nous reprenons ici les notations du chapitre VI. L'alphabet Σ est un ensemble de caractères (ou symboles). un mot est une suite finie de caractères. L'ensemble des mots sur Σ est noté Σ^*. Un *langage* est un sous-ensemble de Σ^*, c'est-à-dire un ensemble particulier de mots. Parmi les mots de Σ^* on distingue le mot vide noté ϵ. Le mot vide est l'unique mot de longueur zéro.

3 Automates finis déterministes

Un automate fini déterministe est un quintuplé $(Q, \Sigma, \delta, q_0, F)$ constitué des éléments suivants

- un alphabet fini (Σ)
- un ensemble fini d'états (Q)

- une fonction de transition ($\delta : Q * \Sigma \to Q$)
- un état de départ ($q_0 \in Q$)
- un ensemble d'états finaux (ou acceptant) $F \subseteq Q$

3.1 Fonctionnement d'un automate fini déterministe

L'automate prend en entrée un mot et l'accepte ou la rejette. On dit aussi qu'il le reconnaît ou ne le reconnaît pas. *Le langage associé à un automate est constitué de l'ensemble des mots qu'il reconnaît.* Voici comment l'automate procède pour décider si un mot appartient à son langage.

- Le processus commence à l'état de départ q_0
- Les symboles du mot sont lus les uns après les les autres.
- À la lecture de chaque symbole, on emploie la fonction de transition δ pour se déplacer vers le prochain état (en utilisant l'état actuel et le caractère qui vient d'être lu).
- le mot est reconnu si et seulement si le dernier état (*i.e.*, l'état correspondant à la lecture du dernier caractère du mot) est un état de F.

De façon plus formelle, pour définir exactement le langage reconnu par un automate, nous introduisons la *fonction de transition étendue aux mots*, $\hat{\delta}$. Elle se définit récursivement comme suit.

- A partir d'un état q en lisant le mot vide ϵ on reste dans l'état q, *i.e.*, $\forall q \in Q, \hat{\delta}(q, \epsilon) = q$
- Étant donné un mot c se terminant par $a \in \Sigma$ (*i.e.*, $c = c'a$ avec $c' \in \Sigma \cup \{\epsilon\}$), et un état q de Q, $\hat{\delta}(q, c) = \hat{\delta}(q, c'a) = \delta(\hat{\delta}(q, c'), a)$

Nous pouvons maintenant définir le langage $L(A)$ accepté par un automate fini déterministe $A = (Q, \Sigma, \delta, q_0, F)$.

$$L(A) = \{c | \hat{\delta}(q_0, c) \in F\}$$

3.2 Des représentation "compactes" des automates

On peut associer à un automate une *table de transition* qui décrit de manière extensive la fonction de transition δ :

- Une colonne correspond à un caractère de l'alphabet.
- Une ligne correspond à un état de l'automate (l'état initial est précédé d'une flèche "$\to$" ; l'état final d'une étoile "$*$")

La valeur $\delta(q, a)$ pour $q \in Q, a \in \Sigma$ correspond à l'état indiqué à l'intersection de la ligne q et de la colonne a. Notons qu'à partir de cette table il est aisé de retrouver l'ensemble des états ainsi que l'alphabet et donc d'identifier exactement l'automate.

Exemple 1 Considérons la table de transition ci-dessous.

	a	b
$\to 1$	1	2
$* 2$	1	2

Il correspond à l'automate $(Q, \Sigma, \delta, q_0, F)$ avec

- $Q = \{1, 2\}$
- $\Sigma = \{a, b\}$
- $\delta(1, a) = 1$, $\delta(1, b) = 2$, $\delta(2, a) = 1$, $\delta(2, b) = 2$
- $q_0 = 1$
- $F = \{2\}$

Il est facile de voir que le langage de cet automate est constitué exactement des mots composés de a et de b qui se terminent par un b.

Pour représenter de façon très intuitive un automate fini déterministe $(Q, \Sigma, \delta, q_0, F)$, on peut utiliser un graphe de transition constitué des éléments suivants :

- Un ensemble de sommets (chaque sommet représente un élément de Q).
- Un ensemble d'arcs entre les sommets valués par un symbole de σ (un arc entre les états q et q' valué par le symbole s signifie que $\delta(q, s) = q'$).
- L'état initial q_0 est marqué par une flèche entrante.
- Les états finaux F sont entourés d'une double ligne.

L'automate de l'exemple 1 est ainsi représenté sur la figure 1.

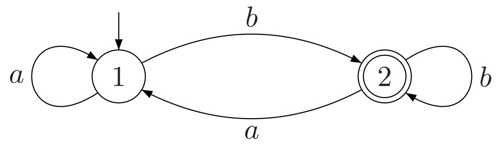


FIG. 1 – Un automate fini déterministe

Pour simplifier encore cette représentation, un arc entre deux sommets q, q' peut être valué par plusieurs symboles $s_1, ..., s_n$ séparés par des virgules. Cette dernière convention signifie simplement que $\forall i \leq n, \delta(q, s_i) = q'$ et elle permet d'éviter une multiplication d'arcs sur le graphe. La figure 2 illustre une telle simplification.

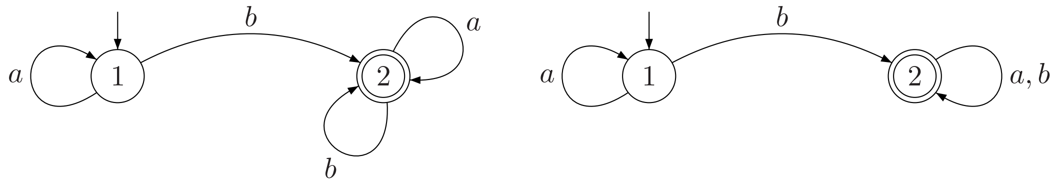
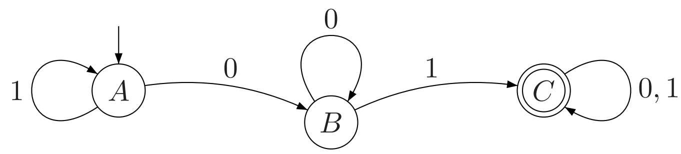


FIG. 2 – Deux représentations équivalentes du même automate fini

Exercice 1 Quel est le langage reconnu par l'automate de la figure 2 ?

Solution. Tous les mots qui contiennent un "b". □

Exercice 2 Écrire la table de transition de l'automate suivant. Quel est le langage reconnu ?



Solution. La table de transition de l'automate est

	0	1
$\rightarrow A$	B	A
B	B	C
$* C$	C	C

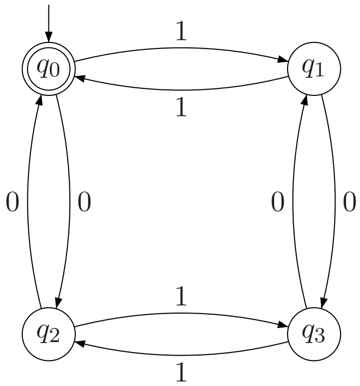
Cet automate reconnaît les mots qui contiennent "01". □

Exercice 3 Soit l'automate fini déterministe $(\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\})$ donné par la table

δ	0	1
$\rightarrow * \; q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Dessiner l'automate et montrer qu'il accepte "110101".

Solution.



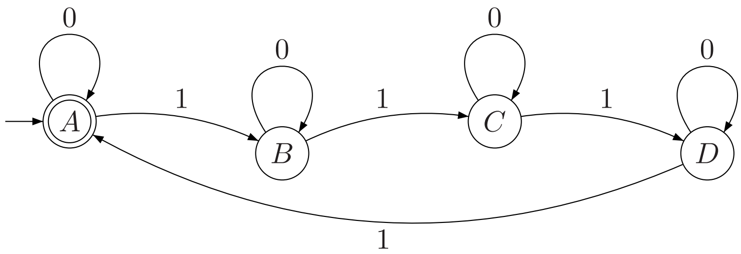
□

Exercice 4 Construire un automate fini déterministe qui reconnaît le langage

$$L = \{x \in \{0,1\}^* | n_1(x) \equiv 0 \bmod 4\}$$

où $n_1(x)$ est le nombre d'occurrence du symbole 1 dans le mot x.

Solution.



□

Exercice 5 Construire les automates finis déterministes qui reconnaissent les langages suivants

$L_1 = \{m \in (a+b)^* | \text{chaque } a \text{ de } m \text{ est immédiatement précédé et immédiatement suivi d'un } b\}$
$L_2 = \{m \in (a+b)^* | m \text{ contienne à la fois } ab \text{ et } ba\}$
$L_3 = \{m \in (a+b)^* | m \text{ contienne exactement une occurrence de } aaa\}$

4 Automates finis non-déterministes

Un automate fini non-déterministe est un automate tel que dans un état donné, il peut y avoir *plusieurs* transitions avec le même symbole. le fonctionnement d'un tel automate n'est donc pas totalement « déterminé », car on ne sait pas quel état l'automate va choisir.

Les automates non-déterministes permettent de modéliser facilement des problèmes complexes. Ils peuvent être convertis en des automates finis déterministe. Ces derniers peuvent être exponentiellement plus grand que les automates non déterministe dont ils sont issus.

Un automate fini non-déterministe est un quintuplé : $(Q, \Sigma, \delta, q_0, F)$

- un alphabet fini (Σ)
- un ensemble fini d'états (Q)

- une fonction de transition δ qui associe à tout état $q \in Q$ et tout symbole $s \in \Sigma$ un sous ensemble de Q noté $\delta(q, s)$.
- un état de départ (q_0)
- un ensemble d'états finaux (ou acceptant) F

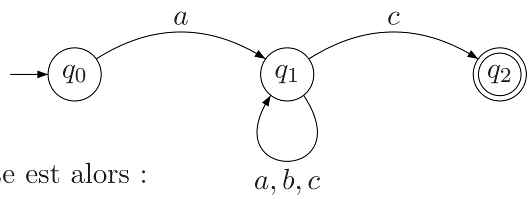
C'est la fonction de transition δ qui diffère ici de celle utilisée par les automates finis déterministes. Remarquons que tout automate fini déterministe est aussi un automate fini non-déterministe.

Les représentations compactes des automates finis déterministes s'étendent naturellement aux automates finis non-déterministes. Une cellule de la table de transition contient un sous-ensemble d'états (éventuellement vide).

4.1 Fonctionnement d'un automate fini non-déterministe

Comme pour un automate fini déterministe, l'automate prend en entrée un mot et l'accepte ou le rejette. Le langage associé est constitué de l'ensemble des mots qu'il reconnaît.

Exemple 2 Voici automate qui reconnaît les mots définis sur l'alphabet $\{a, b, c\}$ qui commencent par a et qui finissent par c.



La table associée à cet automate est alors :

	a	b	c
$\rightarrow q_0$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\{q_1\}$	$\{q_1\}$	$\{q_1, q_2\}$
$* q_2$	$\emptyset$	$\emptyset$	$\emptyset$

Comme pour les automates déterministes, nous nous introduisons la *fonction de transition étendue aux mots*, $\hat{\delta}$. Elle se définit récursivement comme suit.

- A partir d'un état q en lisant le mot vide ϵ (le mot vide ne contient aucun symbole et est toujours noté ϵ), on reste dans l'état q, *i.e.*, $\forall q \in Q, \hat{\delta}(q, \epsilon) = \{q\}$
- Étant donné un mot c se terminant par $a \in \Sigma$ (*i.e.*, $c = c'a$ avec $c' \in \Sigma \cup \{\epsilon\}$), et un état q de Q,

$$\hat{\delta}(q, c) = \hat{\delta}(q, c'a) = \bigcup_{p \in \hat{\delta}(q, c')} \delta(p, a)$$

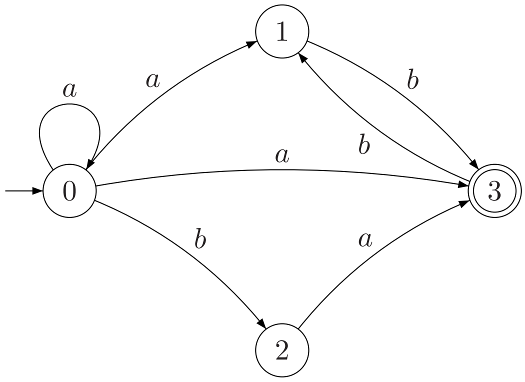
Nous pouvons maintenant définir le langage $L(A)$ accepté par un automate fini déterministe $A = (Q, \Sigma, \delta, q_0, F)$.

$$L(A) = \{c | \hat{\delta}(q_0, c) \cap F \neq \emptyset\}$$

Exercice 6 Construire l'automate fini non-déterministe associé à la table ci-dessous.

	a	b
$\rightarrow 0$	$\{0, 1, 3\}$	$\{2\}$
1	$\emptyset$	$\{3\}$
2	$\{3\}$	$\emptyset$
$* 3$	$\emptyset$	$\{1\}$

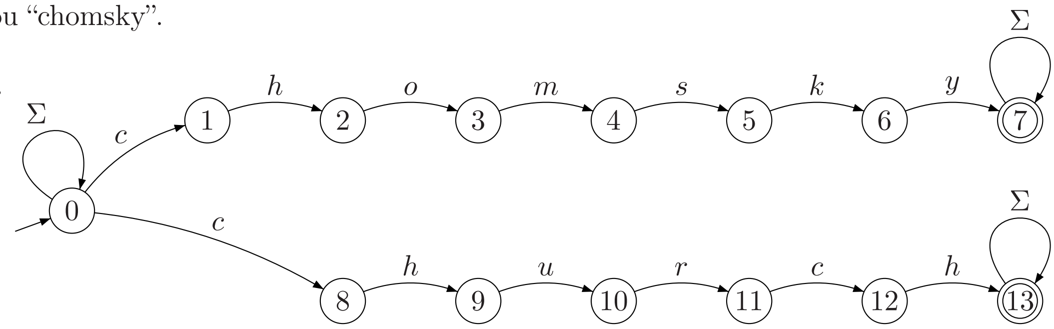
Solution.



□

Exercice 7 Construire un automate fini non-déterministe qui reconnaît les mots qui contiennent "church" ou "chomsky".

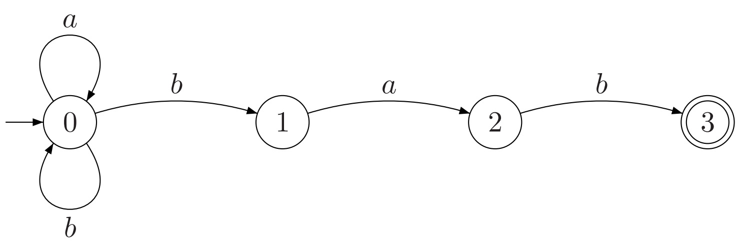
Solution.



□

Exercice 8 Construire un automate finis non-déterministe qui reconnaît les mots de l'alphabet $\{a, b\}$ qui terminent par *bab*.

Solution.



□

Exercice 9 Construire un automate fini non-déterministe et un automate fini déterministe qui reconnaît les mots sur l'alphabet $\{a, b, c\}$ décrits par l'expression régulière $(a+b+c)^*b(a+b+c)$.

Exercice 10 Construire un automate fini non-déterministe qui reconnaît les nombres dont le dernier chiffre n'apparaît qu'une fois.

Exercice 11 Modélisation d'un jeu (d'après la page de Jean-Eric Pin). Le joueur a les yeux bandés. Face à lui, un plateau sur lequel sont disposés en carré quatre jetons, blancs d'un côté et noirs de l'autre. Le but du jeu est d'avoir les quatre jetons du côté blanc. Pour cela, le joueur peut retourner autant de jetons qu'il le souhaite, mais sans les déplacer. A chaque tour, le maître de jeu annonce si la configuration obtenue est gagnante ou pas, puis effectue une rotation du plateau de zéro, un, deux ou trois quarts de tours. La configuration de départ est inconnue du joueur, mais le maître de jeu annonce avant le début du jeu qu'elle n'est pas gagnante. Chaque annonce prend une seconde, et il faut 3 secondes au joueur pour retourner les jetons. Pouvez-vous aider le joueur à gagner en moins d'une minute ?

4.2 Déterminisation d'un automate fini non-déterministe

Un automate fini déterministe est aussi non-déterministe. Donc tout langage reconnu par un automate fini déterministe est reconnu par un automate fini non-déterministe. Plus surprenant, la réciproque est aussi vraie (Théorème de Rabin-Scott).

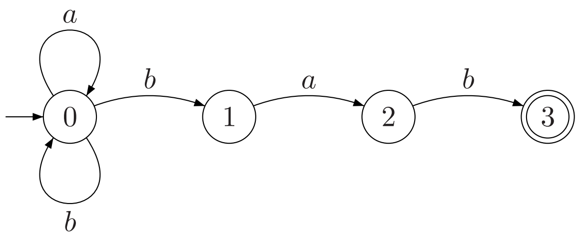
Considérons un automate fini non-déterministe $A_n = (Q_n, \Sigma, \delta_n, q_0, F_n)$ et construisons un automate fini déterministe $A_d = (Q_d, \Sigma, \delta_d, \{q_0\}, F_d)$ qui reconnaît exactement le même langage.

- Les alphabets de A_n et de A_d sont identiques.
- Les états de départ sont respectivement q_0 et le singleton $\{q_0\}$.
- Q_d est constitué de tous les sous-ensembles de Q_n.
- F_d est l'ensemble des sous-ensembles de Q_n qui contiennent au moins un élément de F_n.
- Étant donné un sous ensemble S de Q_n et un symbole $a \in \Sigma$, on définit la fonction de transition $\delta_d(S, a)$ de la manière suivante

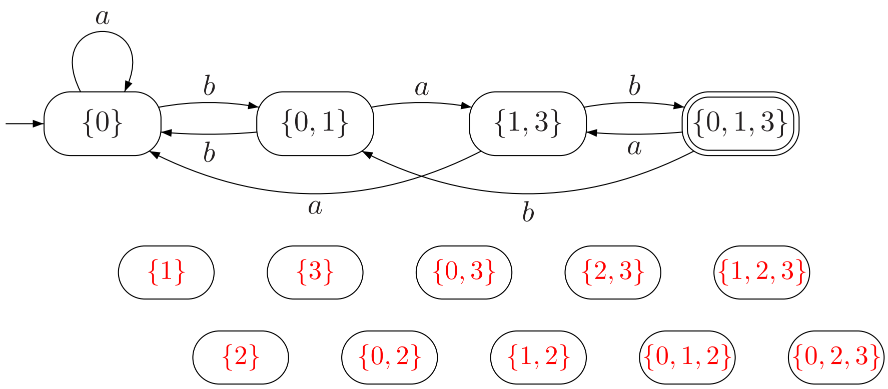
$$\delta_d(S, a) = \bigcup_{q \in S} \delta_n(q, a).$$

Nous illustrons le théorème de Rabin-Scott sur quelques exemples.

Exemple 3 reprenons l'exemple de l'exercice 8. Il s'agissait de construire un automate fini non-déterministe reconnaissant les mots de l'alphabet $\{a, b\}$ qui terminent par bab. L'automate suivant répond à la question.



Essayons maintenant de le déterminiser en construisant un nouvel état à partir de chaque sous ensemble d'état possible.



Remarquons que les états $\{1\}$, $\{2\}$, $\{3\}$, $\{0,2\}$, $\{0,3\}$, $\{1,2\}$, $\{2,3\}$, $\{0,1,2\}$, $\{1,2,3\}$, $\{0,2,3\}$ sont inatteignables et peuvent être "retirés" de l'automate.

En pratique, lors de la conversion, on ne crée pas immédiatement tous les états de l'automate fini déterministe. Les états "utiles" sont crées quand on en a besoin en suivant la méthode de construction ci-dessous :

- Q_d est initialisé à $\emptyset$ et soit E un ensemble d'états initialisé à $E = \{\{q_0\}\}$
- Tant que E est non vide,
  - choisir un élément S de E (S est donc un sous ensemble de Q_n),
  - ajouter S à Q_d,
  - pour tout symbole $a \in \Sigma$,
    + calculer l'état $S' = \bigcup_{q \in S} \delta_n(q, a)$
    + si S' n'est pas déjà dans Q_d, l'ajouter à E
    + ajouter un arc sur l'automate entre S et S' et la valuer par a

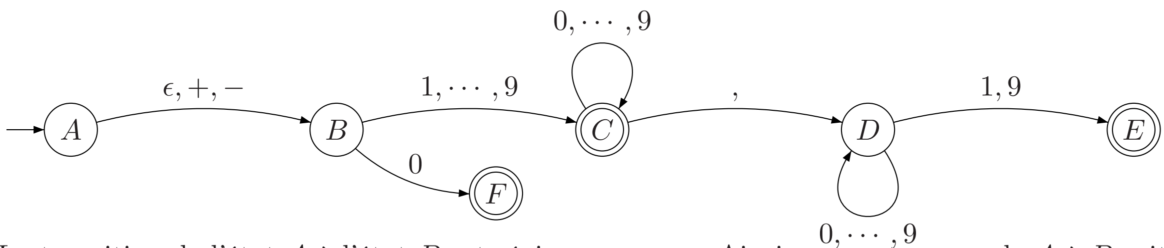
Exercice 12 Déterminiser l'automate de l'exercice 7 (long).

4.3 Les ϵ transitions

Rappelons qu'ϵ représente le mot vide. Une ϵ transition (notée ϵ sur l'arc d'un automate) permet de passer d'un état à l'autre d'un automate sans lire de symbole. Cette facilité permet de programmer facilement des automates complexes.

Une table de transition peut être associée à un automate contenant des ϵ transition. La table est identique à celle utilisée pour un automate fini non-déterministe à ceci près qu'on la complète d'une colonne associée au caractère vide ϵ.

Exemple 4 Pour illustrer les ϵ transitions, construisons un automate fini non déterministe qui reconnaît les nombres décimaux. Rappelons qu'un nombre décimal est un nombre réel qui est le quotient d'un entier relatif par une puissance de dix. Plus précisément, on souhaite pouvoir écrire le nombre décimal en commençant par un "+" ou un "-', suivi d'une suite de chiffres, d'une virgule et d'une suite de chiffres. Bien entendu, le "+" ou le "-" sont optionnels, la première chaîne de chiffres ne peut pas être vide et ne commence pas par "0" (sauf si le nombre décimal est 0). La seconde chaîne ne se termine pas par "0". Si seconde chaîne est vide, on omet la ",".



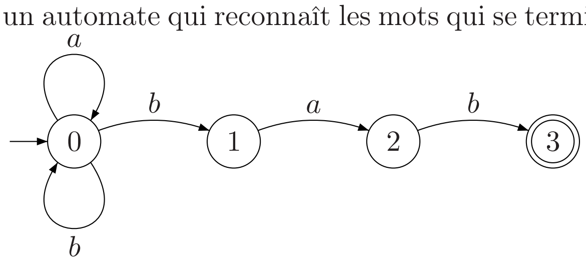
La transition de l'état A à l'état B est régie par $\epsilon, +, -$. Ainsi, on peut passer de A à B soit en lisant $+$, soit en lisant $-$ soit enfin en ne lisant rien.

La table de transition associée à cet automate est alors :

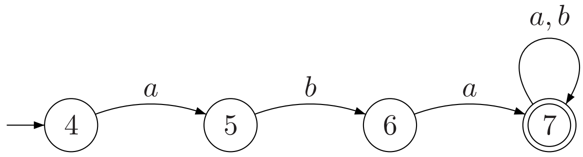
	ϵ	$+$	$-$	,	0	1	2	$\cdots$	9
$\rightarrow A$	$\{B\}$	$\{B\}$	$\{B\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\cdots$	$\emptyset$
B	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{F\}$	$\{C\}$	$\{C\}$	$\cdots$	$\{C\}$
C	$\emptyset$	$\emptyset$	$\emptyset$	$\{D\}$	$\{C\}$	$\emptyset$	$\emptyset$	$\cdots$	$\emptyset$
D	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{D\}$	$\{D,E\}$	$\{D,E\}$	$\cdots$	$\{D,E\}$
$*\ E$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\cdots$	$\emptyset$
$*\ F$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\cdots$	$\emptyset$

Exercice 13 On cherche à construire un automate qui reconnaît les mots qui se terminent par *bab* ou qui commencent par *aba*.

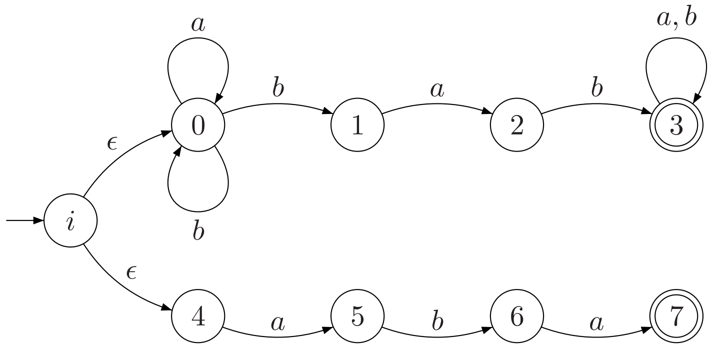
- On sait construire un automate qui reconnaît les mots qui se terminent par *bab* (exercice 8) :



- il est facile de construire un automate qui reconnaît les mots qui commencent par *aba*.



- Il suffit alors d'assembler ces automates avec une simple ϵ transition.



L'introduction des ϵ transition ne change pas la nature des langages reconnus par les automates. Comme pour les automates non-déterministes que l'on peut toujours déterminiser, il est toujours possible d'éliminer les ϵ transition et d'obtenir un automate fini déterministe équivalent. Nous n'aborderons pas ici cette élimination.

5 Automates finis et expressions régulières

Les automates finis et les expressions régulières ont la même expressivité. En effet, le théorème d'équivalence des expressions régulières et des automates finis (Kleene) établit que Le langage accepté par un automate fini correspond à la valeur d'une expression régulière et réciproquement.

6 Un peu de `Java`

Il est aisé de représenter les automates finis sous la forme d'un programme java. Notre objectif est de

- modéliser un automate fini non-déterministe (sans ϵ transition)
- et d'écrire un programme qui détermine si une chaîne de caractère est reconnue par l'automate.

Sans perte de généralité, nous allons supposer que l'alphabet est l'ensemble des caractères ASCII et que les états sont *numérotés* à partir de 0.

6.1 Modèle

Le modèle de données est alors très simple :

- L'état initial de l'automate est indiqué par un entier `q0`.
- La table de transition `delta` est un tableau bidimensionnel de listes d'entiers (la liste d'entier étant ici le moyen le plus simple de représenter un ensemble d'états). Ainsi `delta[q][c]` est la liste des états atteignables à partir de `q` en lisant le caractère `c`.
- L'ensemble des états `finaux` est une liste d'entiers.
- Enfin, le mot que l'on cherche à reconnaître est une chaîne de caractères `String mot`.

Soit donc en Java les deux classes `List` et `Automate`.

```
class Liste {
    int val;
    Liste suivant;
    Liste(int v, Liste x) {
        val = v;
        suivant = x;
    }
}

class Automate {
    int q0;             // état initial
    Liste[][] delta;    // fonction de transition
    Liste finaux;       // états finaux
    Automate(int q, Liste f, Liste[][]d) {
        q0 = q;
        delta = d;
        finaux = f;
    }
}
```

6.2 Algorithme de recherche

Nous aurons besoin de quelques fonctions classiques sur les listes

```
static int longueur(Liste x) {            // La longueur d'une liste
    if (x == null)
        return 0;
    else
        return 1 + longueur(x.suivant);
}
static int kieme(Liste x, int k) {        // Le k ème élément
    if (k == 1)
        return x.val;
    else
        return kieme(x.suivant, k-1);
}
static boolean estDans(Liste x, int v) { // Le test d'appartenance
    if (x == null)
        return false;
    else
        return x.val == v || estDans(x.suivant, v);
}
```

La fonction `accepter(String mot, Automate a)` qui permet de vérifier qu'un mot `mot` est accepté apr l'automate `a` appelle la fonction `static boolean accepter(String mot, Automate a, int i, int q)`.

```
static boolean accepter(String mot, Automate a) {
```

```
    return accepter(mot, a, 0, a.q0);
}
static boolean accepter(String mot, Automate a, int i, int q) {
    if (i == mot.length())
        return Liste.estDans(a.finaux, q);
    else {
        boolean resultat = false;
        int c = mot.charAt(i);   // le code ASCII du caractère courant
        for (Liste nv_q = a.delta[q][c]; nv_q != null; nv_q = nv_q.suivant)
            resultat = resultat || accepter(mot, a, i+1, nv_q.val);
        return resultat;
    }
}
```

La fonction `static boolean accepter(String mot, int i, Automate a, int q)` prend, en plus de l'automate et du mot que l'on étudie, deux autres paramètres :

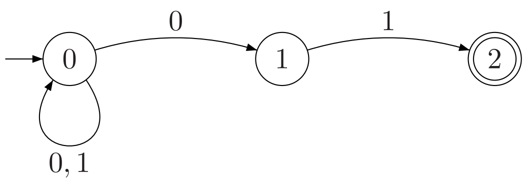
- La position du caractère courant `i` dans le mot.
- L'état courant `q`.

Elle renvoie `true` si et seulement si le sous-mot de `mot` dont on a retiré les `i` premiers caractères était reconnu par un automate semblable à `a` dont l'état initial serait `q`.

Remarquons que si `i` est le dernier caractère du mot (ce qui correspond au test `if (i == mot.length())`) alors il suffit de tester l'appartenance de `q` à `a.finaux`. Si ce n'est pas le cas, on va essayer d'emprunter toutes les transitions possibles. on explore ainsi tous les états `nv_q.val` atteignable à partir de `q` en lisant `c`. Une fois dans l'état `nv_q.val`, on appelle récursivement `accepter`.

6.3 Mise en œuvre sur un automate

Considérons l'automate suivant qui accepte toutes les chaînes qui se terminent par "01".



La table associée à cet automate est alors :

	0	1
$\rightarrow 0$	$\{0,1\}$	$\{0\}$
1	$\emptyset$	$\{2\}$
$*\,2$	$\emptyset$	$\emptyset$

Pour le construire, il nous suffit de construire la table de transition.

```
    public static void main(String [] arg) {
        Liste[][] delta = new Liste[3][128];
        delta[0][(int)'0'] = new Liste(0, new Liste(1, null));
        delta[0][(int)'1'] = new Liste(0, null);
        delta[1][(int)'0'] = null;
        delta[1][(int)'1'] = new Liste(2, null);
        delta[2][(int)'0'] = null;
        delta[2][(int)'1'] = null;
        Automate a = new Automate(0,
                                  new Liste(2, null),
                                  delta);
```

```
        System.out.println("accepter = " + accepter(arg[0], a));
    }
```

Remarquons que le code ASCII du caractère `'0'` est obtenu par `(int)'0'`.

Exercice 14 Comment procéder pour coder un automate avec des ϵ transitions ?

Annexe A

Le coût d'un algorithme

Ce chapitre rappelle les différents moyens d'évaluer le coût d'un algorithme.

1 Une définition très informelle des algorithmes

La formalisation de la notion d'algorithme est assez tardive. Les travaux fondamentaux de Post et de Turing datent de la fin des années 30. Nous ne disposons pas, pour ce cours, des outils qui permettent d'exposer très rigoureusement la notion d'algorithme et de complexité algorithmique. Nous nous contenterons de la définition suivante : *Un algorithme est un assemblage ordonné d'instruction élémentaires qui à partir d'un état initial bien spécifié permettent d'aboutir à un état final, lui aussi bien spécifié.* Notons que pour qu'un algorithme puisse s'exécuter, les données (d'entrées, de sortie et intermédiaires) sont structurées. Ceci permet de les conserver, de les utiliser et de les modifier.

Les étapes élémentaires sont éventuellement répétées (notion de boucle) et sont soumises à des test logiques (instruction de contrôle). Les algorithmes peuvent, la plupart du temps, être codés à l'aide de programmes informatique

2 Des algorithmes "efficaces" ?

Qu'est-ce qu'un bon algorithme ? C'est bien entendu un algorithme qui répond correctement au problème posé. On souhaite généralement qu'il soit aussi *rapide* et n'utilise pas trop de *mémoire.*

Quel que soit la mesure choisie, il est clair que la qualité de l'algorithme n'est pas absolue mais dépend des données d'entrée. Les mesures d'efficacité sont donc des fonctions des données d'entrée. Pour évaluer le temps d'exécution d'un algorithme (par exemple un tri) sur une donnée d (par exemple un ensemble d'entiers à trier), on calcule le nombre d'opérations élémentaires (opérations arithmétiques, affectation, instruction de contrôle, *etc.*) effectuées pour traiter la donnée d. On admet communément que toute opération élémentaire prend un temps constant. Attention, cette hypothèse n'est valable que si tous les nombres sont codés sur un même nombre de bits.

Plutôt que de calculer le nombre d'opérations associé à chaque donnée, on cherche souvent à évaluer le nombre d'opérations nécessaires pour traiter les données qui ont une certaine "taille". Il existe souvent un paramètre naturel qui est un estimateur raisonnable de la taille d'une donnée (par exemple, le nombre n d'éléments à trier pour un algorithme de tri, la taille n, m d'une matrice pour le calcul du déterminant, la longueur n d'une liste pour une inversion, *etc.*). Pour simplifier nous supposerons dans la suite que la "taille" d'une donnée est évaluée par un unique entier n.

Les informaticiens s'intéressent à l'ordre de grandeur des temps d'exécution (ou de taille mémoire) quand n devient très grand. Le coût exact en secondes est en effet très difficile à mesurer et dépend de très nombreux paramètres (*cf.* Section 4). On utilise donc les notations O, Θ et Ω pour exprimer des relations de domination : Soient f et g deux fonctions définies des entiers naturels vers les entiers naturels.

- $f(n) = O(g(n))$ si et seulement si il existe deux constantes positives n_0 et B telles que
$$\forall n \geq n_0, f(n) \leq Bg(n)$$
Ce qui signifie que f ne croît pas plus vite que g. Un algorithme en $O(1)$, c'est un algorithme dont le temps d'exécution ne dépend pas de la taille des données. C'est donc un ensemble constant d'opérations élémentaires (exemple : l'addition de deux entiers). On dit d'un algorithme qu'il est linéaire si il utilise $O(n)$ opérations élémentaires. Il est polynomial si il existe une constante a telle que le nombre total d'opérations élémentaires est $O(n^a)$.
- $f(n) = \Omega(g(n))$ si et seulement si il existe deux constantes positives n_0 et B telles que
$$\forall n \geq n_0, f(n) \geq Bg(n)$$
- $f(n) = \Theta(g(n))$ si et seulement si il existe trois constantes positives n_0, B et C telles que
$$\forall n \geq n_0, Bg(n) \leq f(n) \leq Cg(n)$$

Dans la pratique, le temps d'exécution d'un algorithme dépend non seulement de n mais aussi de la structure des données. Par exemple, le tri d'une liste d'entiers est, pour certains algorithmes, plus rapide si la liste est partiellement triée plutôt que dans un ordre parfaitement aléatoire. Pour analyser finement un algorithme, on a donc besoin de plusieurs mesures :

- La complexité dans le "pire cas". On mesure alors le nombre d'opérations avec les données qui mènent à un nombre maximal d'opérations. Soit donc
$$\text{complexité dans le pire cas} = \max_{d \text{ donnée de taille } n} C(d)$$
où $C(d)$ est le nombre d'opérations élémentaires pour exécuter l'algorithme sur la donnée d'entrée d.
- La complexité en moyenne. Cette notion n'a de sens que si l'on dispose d'une hypothèse sur la distribution des données. Soit donc
$$\text{complexité en moyenne} = \sum_{d \text{ donnée de taille } n} \pi(d)C(d)$$
où $\pi(d)$ est la probabilité d'avoir en entrée une instance d parmi toutes les données de taille n.

Ces notions s'étendent à la consommation mémoire d'un algorithme. On parle alors de complexité spatiale (maximale ou moyenne).

En pratique, le pire cas est rarement atteint et l'analyse en moyenne semble plus séduisante. Attention tout de même à deux écueils pour les calculs en moyenne :

- Pour que ce calcul garde un certain sens, il faut connaître la distribution des données, ce qui est délicat à estimer. On fait parfois l'hypothèse que les données sont équiprobables (ce qui est bien souvent totalement arbitraire).
- Comme nous allons le voir, les calculs de complexité en moyenne sont fort délicats à mettre en œuvre.

3 Quelques exemples

Nous donnons ici des exemples simples et variés d'analyse d'algorithmes. De nombreux autres exemples sont dans le polycopié.

3.1 Factorielle

La première version de notre programme permettant de calculer $n!$ est itérative.

```
static int factorielle(int n) {
    int f = 1;
    for (int i= 2; i <= n; i++)
        f = f * i;
    return f;
}
```

Nous avons $n-1$ itérations au sein desquelles le nombre d'opérations élémentaires est constant. La complexité est $O(n)$

La seconde version est récursive. Soit $C(n)$ le nombre d'opérations nécessaires pour calculer `factorielle(n)`. Nous avons alors $C(n) \leq \lambda + C(n-1)$ où λ est une constante qui majore le nombre d'opérations précédent l'appel récursif. De plus $C(1)$ se calcule en temps constant et donc $C(n) = O(n)$.

```
static int factorielle(int n) {
  if (n <= 0)
     return 1;
  else
     return n * factorielle(n-1);
}
```

3.2 Recherche Dichotomique

Considérons un tableau T de n entiers *triés*. On cherche à tester si un entier v donné se trouve dans le tableau. Pour ce faire, on utilise une recherche "dichotomique".

```
static boolean trouve(int[] T, int v, int min, int max){
    if(min >= max) // vide
        return false;
    int mid = (min + max) / 2;
    if (T[mid] == v)
        return true;
    else if (T[mid] > v)
        return trouve(T, v, min, mid);
    else
        return trouve(T, v, mid + 1, max);
}
```

La fonction `trouve` cherche l'entier `v` dans `T` entre les indices `min` et `max -1`. Pour effectuer une recherche sur tout le tableau, il suffit d'appeler `trouve(T, v, 0, T.length)`.

Le nombre total d'opérations est proportionnel au nombre de comparaisons $C(n)$ effectuées par l'algorithme récursif. Et donc, nous avons immédiatement : $C(n) = 1 + C(\frac{n}{2})$. Soit donc, $C(n) = O(\log n)$.

3.3 Tours de Hanoi

Le très classique problème des "tours de Hanoi" consiste à déplacer des disques de diamètres différents d'une tour de départ à une tour d'arrivée en passant par une tour intermédiaire. Les règles suivantes doivent être respectées : on ne peut déplacer qu'une disque à la fois, et on ne peut placer un disque que sur un disque plus grand ou sur un emplacement vide.

Identifions les tours par un entier. Pour résoudre ce problème, il suffit de remarquer que si l'on sait déplacer une tour de taille n de la tour `ini` vers `dest`, alors pour déplacer une tour de taille $n+1$ de `ini` vers `dest`, il suffit de déplacer une tour de taille n de `ini` vers `temp`, un disque de `ini` vers`dest` et finalement la tour de hauteur n de`temp` vers`dest`.

```
public static void hanoi(int n, int ini, int temp, int dest){
    if (n == 1){ // on sait le faire
        System.out.println("deplace" + ini + " " + dest);
        return;
    }           // sinon recursion
    hanoi(n - 1, ini, dest, temp);
    System.out.println("deplace" + ini + " " + dest);
    hanoi(n-1, temp, ini, dest);
}
```

Notons $C(n)$ le nombre d'instructions élémentaires pour calculer `hanoi(n, ini, temp, dest)`. Nous avons alors $C(n+1) \leq 2C(n)+\lambda$, où λ est une constante. Tour de Hanoi est exponentielle.

3.4 Recherche d'un nombre dans un tableau, complexité en moyenne

On suppose que les éléments d'un tableau `T` de taille n sont des nombres entiers distribués de façon équiprobable entre 1 et k (une constante). Considérons maintenant l'algorithme de recherche ci-dessous qui cherche une valeur `v` dans `T`.

```
static boolean trouve(int[] T, int v) {
    for (int i = 0; i < T.length; i++)
        if (T[i] == v)
            return true;
    return false;
}
```

La complexité dans le pire cas est clairement $O(n)$. Quelle est la complexité en moyenne ?

Remarquons que nous avons k^n tableaux. Parmi ceux-ci, $(k-1)^n$ ne contiennent pas `v` et dans ce cas, l'algorithme procède à exactement n itérations. Dans le cas contraire, l'entier est dans le tableau et sa première occurrence est alors i avec une probabilité de

$$\frac{(k-1)^{i-1}}{k^i}$$

et il faut alors procéder à i itérations. Au total, nous avons une complexité moyenne de

$$C = \frac{(k-1)^n}{k^n} \times n + \sum_{i=1}^{n} \frac{(k-1)^{i-1}}{k^i} \times i$$

Or

$$\forall x, \sum_{i=1}^{n} i x^{i-1} = \frac{1 + x^n(nx - n - 1)}{(1-x)^2}$$

(il suffit pour établir ce résultat de dériver l'identité $\sum_{i=1}^{n} x^i = \frac{1-x^{n+1}}{1-x}$) et donc

$$C = n\frac{(k-1)^n}{k^n} + k\left(1 - \frac{(k-1)^n}{k^n}(1+\frac{n}{k})\right) = k\left(1 - \left(1-\frac{1}{k}\right)^n\right)$$

4 Coût estimé *vs.* coût réel

Les mesures présentées dans ce chapitre ne sont que des estimations asymptotique des algorithmes. En pratique, il faut parfois atteindre de très grandes valeurs de n pour qu'un algorithme en $O(n \log n)$ se comporte mieux qu'un algorithme quadratique.

Les analyses de complexité peuvent servir à comparer des algorithmes mais le modèle de coût est relativement simple (par exemple, les opérations d'accès aux disques, ou le trafic réseau généré ne sont pas pris en compte alors que ces paramètres peuvent avoir une influence considérable sur un programme). Il est toujours nécessaire de procéder à des analyses expérimentales avant de choisir le "meilleur" algorithme, *i.e.*, l'algorithme spécifiquement adapté au problème que l'on cherche à résoudre.

Annexe B

Morceaux de Java

1 Un langage plutôt classe

Java est un langage objet avec des classes. Les classes ont une double fonction : structurer les programmes et dire comment on construit les objets. Pour ce qui est de la seconde fonction, il n'est pas si facile de définir ce qu'est exactement un objet dans le cas général. Disons qu'un objet possède un état et des *méthodes* qui sont des espèces de fonctions propres à chaque objet. Par exemple tous les objets possèdent une méthode `toString` sans argument et qui renvoie une chaîne représentant l'objet et normalement utilisable pour l'affichage. La section 2 décrit la construction des objets à partir des classes.

Mais décrivons d'abord la structuration des programmes réalisée par les classes. Les classes regroupent des *membres* qui sont le plus souvent des variables (plus précisément des *champs*) et des méthodes, mais peuvent aussi être d'autres classes.

1.1 Programme de classe

Un programme se construit à partir de une ou plusieurs classes, dont une au moins contient une méthode `main` qui est le point d'entrée du programme. Les variables des classes sont les variables globales du programme et leurs méthodes sont les fonctions du programme. Une variable ou une méthode qui existent dès que la classe existe sont dites *statiques*.

Commençons par un programme en une seule classe. Par exemple, la classe simple suivante est un programme qui affiche `Coucou !` sur la console :

```
class Simple {
  static String msg = "Coucou !" ;     // déclaration de variable

  public static void main (String [] arg) // déclaration de méthode
  {
    System.out.println(msg) ;
  }
}
```

Cette classe ne sert pas à fabriquer des objets. Elle se suffit à elle même. Par conséquent tout ce qu'elle définit (variable `msg` et méthode `main`) est statique. Par re-conséquent, toutes les déclarations sont précédées du mot-clé **static**, car si on ne met rien les membres ne sont pas statiques. Si le source est contenu dans un fichier `Simple.java`, il se compile par `javac Simple.java` et se lance par `java Simple`. C'est une bonne idée de mettre les classes dans des fichiers homonymes, ça permet de s'y retrouver.

En termes de programmation objet, la méthode `main` invoque la méthode `println` de l'objet **System**`.out`, avec l'argument `msg`. **System**`.out` désigne la variable `out` de la classe **System**, qui fait partie de la bibliothèque standard de Java. Notons que `msg` est en fait **Simple**`.msg`, mais dans la classe **Simple**, on peut se passer de rappeler que `msg` est une variable de la classe **Simple**, alors autant en profiter.

Reste à se demander quel est l'objet rangé dans **System**`.out`. Et bien, disons que c'est un objet d'une autre classe (la classe **PrintStream**) qui a été mis là par le système Java et ne nous en préoccupons plus pour le moment.

1.2 Complément : la méthode `main`

La déclaration de cette méthode doit obligatoirement être de la forme :

```
public static void main (String [] arg)
```

En plus d'être statique, la méthode `main` doit impérativement être *publique* (mot-clé **public**) et prendre un tableau de chaîne en argument. Le sens du mot-clé **public** est expliqué plus loin.

Le reste des obligations porte sur le type de l'argument de `main` (son nom est libre). Le tableau de chaîne est initialisé par le système pour contenir les *arguments de la ligne de commande*. De sorte que l'on peut facilement écrire une commande `echo` en Java.[1]

```
class Echo {
  public static void main (String [] arg) {
    for (int i = 0 ; i < arg.length ; i++) {
      System.out.println(arg[i]);
    }
  }
}
```

Ce qui nous donne après compilation :

```
% java Echo coucou foo bar
coucou
foo
bar
```

1.3 Collaboration de classe

La classe-programme **Simple** utilise déjà une autre classe, la classe **System** écrite par les auteurs du système Java. Pour structurer vos programmes, vous pouvez (devez) vous aussi écrire plusieurs classes. Par exemple, récrivons le programme simple à l'aide de deux classes. Le message est fourni par une classe **Message**

```
class Message {
  static String msg = "Coucou !" ;
}
```

Tandis que le programme est modifié ainsi :

```
class Simple {
  public static void main (String [] arg) { // déclaration de méthode
    System.out.println(Message.msg) ;
  }
}
```

Si on met la classe **Message** dans un fichier `Message.java`, elle sera compilée automatiquement lorsque l'on compile le fichier `Simple.java` (par `javac Simple.java`). Encore une bonne raison pour mettre les classes dans des fichiers homonymes.

[1] `echo` est une commande Unix qui affiche ses arguments

1.4 Méfiance de classe

Lorsque l'on fabrique un programme avec plusieurs classes, l'une d'entre elles contient la méthode `main`. Les autres fournissent des services, en général sous forme de méthodes accessibles à partir des autres classes.

Supposons que la classe Hello doit fournir un message de bienvenue, en anglais ou en français. On pourra écrire.

```
class Hello {
  private static String hello ;

  static void setEnglish() { hello = "Hello!" ; }
  static void setFrench()  { hello = "Coucou !" ; }
  static String getHello() { return hello ; }

  static { setEnglish() ; }
}
```

Classe utilisée par une nouvelle classe Simple.

```
class Simple {
  public static void main (String [] arg) {
    System.out.println(Hello.getHello()) ;
  }
}
```

La variable `hello` est *privée* (mot-clé **private**) ce qui interdit son accès à partir de code qui n'est pas dans la classe Hello. Une méthode `getHello` est donc fournie, pour pouvoir lire le message. Deux autres méthodes laissent la possibilité aux utilisateurs de la classe de sélectionner le message anglais ou le message français. Enfin, le bout de code `static { setEnglish() ; }` est exécuté lors de la création de la classe en machine, ce qui assure le choix initial de la langue du message. Finalement, la conception de la classe Hello garantit une propriété : à tout instant, `Hello.hello` contient nécessairement un message de bienvenue en français ou en anglais.

La pratique de restreindre autant que possible la visibilité des variables et méthodes améliore à la fois la sûreté et la structuration des programmes.

- Chaque classe propose un service, qui sera maintenu même si la réalisation de ce service change. Il est alors moins risqué de modifier la réalisation d'un service. En outre, puisque seul le code de la classe peut modifier les données privées, on peut garantir que ces données seront dans un certain état, puisqu'il suffit de contrôler le code d'une seule classe pour s'en convaincre.
- La structure des programmes est plus claire car, pour comprendre comment les diverses classes interagissent, il suffit de comprendre les méthodes (et variables) exportées (i.e., non-privées) des classes. En fait il suffit normalement de comprendre les déclarations des méthodes exportées assorties d'un commentaire judicieux.

On parle d'abstraction, la séparation en classe segmente le programme en unités plus petites, dont on n'a pas besoin de tout savoir.

À l'intérieur de la classe elle-même, la démarche d'abstraction revient à écrire plusieurs méthodes, chaque méthode réalisant une tâche spécifique. Les classes elle-mêmes peuvent être regroupées en *packages*, qui constituent une nouvelle barrière d'abstraction. La bibliothèque de Java, qui est énorme, est structurée en packages.

Le découpage en packages, puis en classes, puis en méthodes, qui interagissent selon des conventions claires qui disent le quoi et cachent les détails du comment, est un fondement de la bonne programmation, c'est-à-dire de la production de programmes compréhensibles et donc de programmes qui sont (plus) facilement mis au point, puis modifiés.

Il y a en Java quatre niveaux de visibilité, du plus restrictif au plus permissif.

- **private** : visible de la classe.
- Rien : visible du package.
- **protected** visible du package et des classes qui héritent de la classe (voir ci-dessous).
- **public** : visible de partout.

Nous connaissons déjà quelques emplois de **public**.

- Toutes les classes dont les sources sont dans le répertoire courant sont membres d'un même package implicite. La classe qui initialise le système d'exécution de Java, puis lance le programme de l'utilisateur, n'est pas membre de ce package. Il est donc logique que `main` soit déclarée **public**. Noter que déclarer **public** les autres méthodes de vos classes n'a aucun sens, à moins d'être train d'écrire un package.
- La déclaration complète de la méthode `toString` des objets est

  ```
  public String toString() ;
  ```

  Et c'est logique, puisqu'il est normal de pouvoir afficher un objet de n'importe où dans le programme.
- Quand vous lisez la documentation d'une classe de la bibliothèque (par exemple **String**) vous avez peut être déjà remarqué que tout est **public** (ou très rarement **protected**).

1.5 Reproduction de classe par héritage

La plus grande part de la puissance de la programmation objet réside dans le mécanisme de l'*héritage*. Comme première approche, nous examinons rapidement l'héritage et seulement du point de vue de la classe. Soit une classe Foo qui hérite d'une classe Bar.

```
class Foo extends Bar {
  ...
}
```

La classe Foo démarre dans la vie avec toutes les variables et toutes les méthode de la classe Bar. Mais la classe Foo ne va pas se contenter de démarrer dans la vie, elle peut effectivement étendre la classe dont elle hérite en se donnant de nouvelles méthodes et de nouvelles variables. Elle peut aussi redéfinir les méthodes et variables héritées. Par exemple, on peut construire une classe HelloGoodBye qui offre un message d'adieu en plus du message de bienvenue de Hello, et garantit que les deux messages sont dans la même langue.

```
class HelloGoodBye extends Hello {
  private static String goodbye ;

  static void setEnglish() { Hello.setEnglish() ; goodbye = "Goodbye!" ; }
  static void setFrench() { Hello.setFrench() ; goodbye = "Adieu !" ; }
  static String getGoodBye() { return goodbye ; }

  static { setEnglish() ; }
}
```

On note que deux méthodes sont redéfinies (`setEnglish` et `setFrench`) et qu'une méthode est ajoutée (`getGoodBye`). La méthode `setEnglish` ci-dessus doit, pour assurer la cohérence des deux messages, appeler la méthode `setEnglish` de la classe Hello, d'où l'emploi d'une notation complète Hello`.setEnglish`.

Comme démontré par la nouvelle et dernière classe Simple, la classe HelloGoodBye a bien reçu la méthode `getHello` en héritage.

```
class Simple {
  public static void main (String [] arg) {
```

```
    System.out.println(HelloGoodBye.getHello()) ;
    System.out.println(HelloGoodBye.getGoodBye()) ;
  }
}
```

En fait, l'héritage des classes n'a que peu d'intérêt en pratique, l'héritage des objets est bien plus utile.

2 Obscur objet

Dans cette section, nous examinons la dialectique question de la classe et de l'objet.

2.1 Utilisation des objets

Sans même construire des objets nous mêmes, nous en utilisons forcément, car l'environnement d'exécution de Java est principalement en style objet. Autrement dit on sait déjà que les objets ont des méthodes.

(1) Les tableaux sont presque des objets.

(2) Les chaînes **String** sont des objets.

(3) La bibliothèque construit des objets dont nous appelons les méthodes, voir `out.println()`.

Les objets possèdent aussi des *champs* également appelés *variables d'instance*, auxquels on accède par la notation en « . » comme pour les variables de classes. Par exemple, si t est un tableau t`.length` est la taille du tableau.

La bibliothèque de Java emploie énormément les objets. Par exemple, le point est un objet de la classe **Point** du package `java.awt`. On crée un nouveau point par un appel de constructeur dont la syntaxe est :

```
Point p = new Point () ; // Point origine
```

On peut aussi créer un point en donnant explicitement ses coordonnées (entières) au constructeur :

```
Point p = new Point (100,100) ;
```

On dit que le constructeur de la classe **Point** est surchargé (*overloaded*), c'est-à-dire qu'il y a en fait deux constructeurs qui se distinguent par le type de leurs arguments. Les méthodes, statiques ou non, peuvent également être surchargées.

On peut ensuite accéder aux champs d'un point à l'aide de la notation usuelle. Les points ont deux champs `x` et `y` qui sont leurs coordonnées horizontales et verticales :

```
if (p.x == p.y)
  System.out.println("Le point est sur la diagonale");
```

Les points possèdent aussi des méthodes, par exemple la méthode `distance`, qui calcule la distance euclidienne entre deux points, renvoyée comme un flottant double précision **double**. Un moyen assez compliqué d'afficher une approximation de $\sqrt{2}$ est donc

```
System.out.println(new Point ().distance(new Point (1, 1))) ;
```

Les objets sont reliés aux classes de deux façons :

- L'objet est créé par un constructeur défini dans une classe. Ce constructeur définit la classe d'origine de l'objet, et l'objet garde cette classe-là toute sa vie.
- Les classes sont aussi plus ou moins les types des objets, quand nous écrivons

  ```
  Point p = ...
  ```

> Nous déclarons une variable de type **Point**. Dans certaines conditions (voir les sections 2.3 et III.1.2), la classe-type peut changer au cours de la vie l'objet. Le plus souvent, cela signifie qu'un objet dont la classe reste immuable, peut, dans certaines conditions, être rangé dans une variable dont le type est une autre classe.

Un premier exemple (assez extrême) de cette distinction est donné par **null**. La valeur **null** n'a ni classe d'origine (elle n'est pas créé par **new**), ni champs, ni méthodes, ni rien, mais alors rien du tout. En revanche, **null** appartient à toutes les classes-types.

2.2 Fabrication des objets

Notre idée est de montrer comment créer des objets très similaires aux points de la section précédente. On crée très facilement une classe des paires (d'entiers) de la façon suivante :

```
class Pair {
  int x ; int y ;

  Pair () { this(0,0) ; }

  Pair (int x, int y) { this.x = x ; this.y = y ; }

  double distance(Pair p) {
    int dx = p.x - this.x ;
    int dy = p.y - this.y ;
    return Math.sqrt (dx*dx + dy*dy) ; // Math.sqrt est la racine carrée
  }
}
```

Nous avons partout explicité les accès aux variables d'instance et par `this.x` et `this.y`. Nous nous sommes aussi laissés aller à employer la notation `this(0,0)` dans le constructeur sans arguments, cela permet de partager le code entre constructeurs.

On remarque que les champs `x` et `y` ne sont pas introduits par le mot-clé **static**. Chaque objet possède ses propres champs, le programme

```
  Pair p1 = new Pair (0, 1) ;
  Pair p2 = new Pair (2, 3) ;

  System.out.println(p1.x + ", " + p2.y) ;
```

affiche « `0, 3` », ce qui est somme toute peu surprenant. De même, la méthode `distance` est propre à chaque objet, ce qui est logique. En effet, si `p` est une autre paire, les distances `p1.distance(p)` et `p2.distance(p)` n'ont pas de raison particulières d'être égales. Rien n'empêche de mettre des membres **static** dans une classe à créer des objets. Le concepteur de la classe `Pair` peut par exemple se dire qu'il est bien dommage de devoir créer deux objets si on veut simplement calculer une distance. Il pourrait donc inclure la méthode statique suivante dans sa classe `Pair`.

```
  static double distance(int x1, int y1, int x2, int y2) {
    int dx = x2-x1, dy = y2-y1 ;
    return Math.sqrt(dx*dx+dy*dy) ;
  }
```

Il serait alors logique d'écrire la méthode `distance` dynamique plutôt ainsi :

```
  double distance(Pair p) { return distance(this.x, this.y, p.x, p.y) ; }
```

Où bien sûr, `distance (this.x,`... se comprend comme `Pair.distance (this.x,`...

2.3 Héritage des objets, sous-typage

L'héritage dans toute sa puissance sera abordé au cours suivant. Mais nous pratiquons déjà l'héritage sans le savoir. En effet, les objets des classes que nous écrivons ne démarrent pas tout nus dans la vie. Toute classe hérite implicitement (pas besoin de **extends**, voir la section 1.5) de la classe **Object**. Les objets de la classe **Object** (et donc tous les objets) possèdent quelques méthodes, dont la fameuse méthode `toString`. Par conséquent, le code suivant est accepté par le compilateur, et s'exécute sans erreur. Tout se passe exactement comme si nous avions écrit une méthode `toString`, alors que cette méthode est en fait héritée.

```
Pair p = new Pair (1, 0) ;
String repr = p.toString() ;
System.out.print(repr) ;
```

Ce que renvoie la méthode `toString` des **Object** n'est pas bien beau.

```
Pair@10b62c9
```

On reconnaît le nom de la classe Pair suivi d'un nombre en hexadécimal qui est plus ou moins l'adresse dans la mémoire de l'objet dont on a appelé la méthode `toString`.

Mais nous pouvons redéfinir (*override*) la méthode `toString` dans la classe Pair.

```
// public car il faut respecter la déclaration initiale
 public String toString() {
   return "(" + x + ", " + y + ")" ;
 }
```

Et l'affichage de `p.toString()` produit cette fois ci le résultat bien plus satisfaisant `(1, 0)`.

Même si c'est un peu bizarre, il n'est au fond pas très surprenant que lorsque nous appelons la méthode `toString` de l'intérieur de la classe Pair comme nous le faisons ci-dessus, ce soit la nouvelle méthode `toString` qui est appelée. Mais écrivons maintenant plus directement :

```
System.out.print(p) ;
```

Et nous obtenons une fois encore l'affichage `(1, 0)`. Or, nous aurons beau parcourir la liste des neuf définitions de méthodes `print` surchargées de la classe **PrintStream**, de `print(boolean b)` à `print(Object obj)`, nous n'avons bien évidemment aucune chance d'y trouver une méthode `print(Pair p)`.

Mais un objet de la classe Pair peut aussi être vu comme un objet de la classe **Object**. C'est le *sous-typage* : le type des objets Pair est un sous-type du type des objets **Object** (penser sous-ensemble : Pair $\subset$ **Object**). En Java, l'héritage entraîne le sous-typage, on dit alors plus synthétiquement que Pair est une *sous-classe* de **Object**.

On note que le compilateur procède d'abord à un sous-typage (il se dit qu'un objet Pair est un objet **Object**), pour pouvoir résoudre la surcharge (il sélectionne la bonne méthode `print` parmi les neuf possibles). La conversion de type vers un sur-type est assez discrète, mais on peut l'expliciter.

```
System.out.print((Object)p) ; // Change le type explicitement
```

C'est donc finalement la méthode `print(Object obj)` qui est appelée. Mais à l'intérieur de cette méthode, on ne sait pas que `obj` est en fait un objet Pair. En simplifiant un peu, le code de cette méthode est équivalent à ceci

```
public void print (Object obj) {
  if (obj == null) {
    this.print("null") ;
  } else {
    this.print(obj.toString()) ;
```

```
  }
}
```

C'est-à-dire que, au cas de **null** près, la méthode `print(`**Object** `obj)` appelle `print(`**String** `s)` avec comme argument `obj.toString()`. Et là, il y a une petite surprise : c'est la méthode `toString()` de la classe d'origine (la « vraie » classe de `obj`) qui est appelée, et non pas la méthode `toString()` des **Object**. C'est ce que l'on appelle parfois la *liaison tardive*.

3 Constructions de base

Nous évoquons des concepts qui regardent l'ensemble des langages de programmation, et pas seulement les langages objet.

3.1 Valeurs, variables

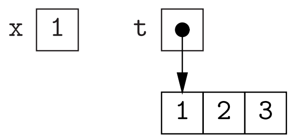
Par valeur on entend en général le résultat de l'évaluation d'une expression du langage de programmation. Si on prend un point de vue mathématique, une valeur peut être à peu près n'importe quoi, un entier (de $\mathbb{Z}$), un ensemble d'entiers etc. Si on prend un point de vue technique, une valeur est ce que l'ordinateur manipule facilement, ce qui entraîne des contraintes : par exemple, un entier n'a pas plus de 32 chiffres binaires (pas plus de 32 *bits*). Dans les descriptions qui suivent nous entendons valeur plutôt dans ce sens technique. Par variable on entend en général (selon le point de vue technique) une case *qui possède un nom* où peut être rangée une valeur. Une variable est donc une portion nommée de la mémoire de la machine.

3.1.1 Scalaires et objets

Il y a en Java deux grandes catégories de valeurs les *scalaires* et les *objets*. La distinction est en fait technique, elle tient à la façon dont ces valeurs sont traitées par la machine, ou plus exactement sont rangées dans la mémoire. Les valeurs scalaires se suffisent à elle-mêmes. Les valeurs des objets sont des *références*. Une référence « pointe » vers quelque chose (une zone de la mémoire) — référence est un nom civilisé pour *pointeur* ou *adresse en mémoire*. Écrivons par exemple

```
int x = 1 ;
int [] t = {1, 2, 3} ;
```

Les variables `x` et `t` sont deux cases, qui contiennent chacune une valeur, la première valeur étant scalaire et la seconde une référence. Un schéma résume la situation.

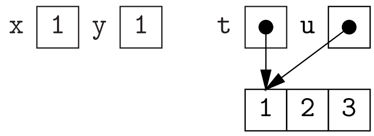


Le tableau `{1, 2, 3}` correspond à une zone de mémoire qui contient des trois entiers, mais la valeur qui est rangée dans la variable `t` est une référence pointant vers cette zone. Le schéma est une simplification de l'état de la mémoire, les zones mémoires apparaissent comme des cases (les variables portent un nom) et les références apparaissent comme des flèches qui pointent vers les cases.

Si x et y sont deux variables, la construction $y = x$ se traduit par une copie de la valeur contenue dans la variable x dans la variable y, que cette valeur soit une référence ou non. Ainsi, le code

```
int y = x ;
int [] u = t ;
```

produit l'état mémoire simplifié suivant.



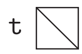
Le schéma permet par exemple de comprendre pourquoi (ou plutôt comment) le programme suivant affiche 4.

```
int [] t = {1, 2, 3} ;
int [] u = t ;
u[1] = 4 ;
System.out.println(t[1]) ;
```

Il existe une référence qui ne pointe nulle part **null**, nous pouvons l'employer partout où une référence est attendue.

```
int [] t = null ;
```

Dans les schémas nous représentons **null** ainsi :



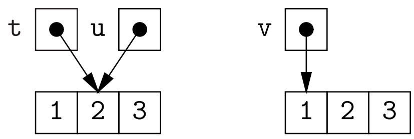
Puisque **null** ne pointe nulle part il n'est pas possible de le « *déréférencer* » c'est à dire d'aller voir où il pointe. Un essai par exemple de `t[0]` déclenche une erreur à l'exécution.

Égalité des valeurs

L'opérateur d'égalité == de Java s'applique aux valeurs — ainsi que l'opérateur différence `!=`. Si l'égalité de deux scalaires ne pose aucun problème, il faut comprendre que == entre deux objets traduit l'égalité des références et que deux références sont égales, si et seulement si elles pointent vers la même zone de mémoire. Autrement dit, le programme

```
int [] t = {1, 2, 3} ;
int [] u = t ;
int [] v = {1, 2, 3} ;
System.out.println("t==u : " + (t == u) + ", t==v : " + (t == v)) ;
```

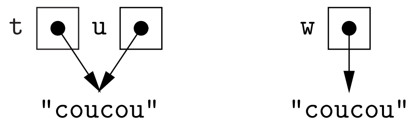
affiche `t==u : true, t==v : false`. Les références `t` et `u` sont égales parce qu'elles pointent vers le même objet. Les références `t` et `v` qui pointent vers des objets distincts sont distinctes. Cela peut se comprendre si on revient aux états mémoire simplifiés.



On dit parfois que == est l'égalité *physique*. L'égalité physique donne parfois des résultats surprenants. Soit le programme Test simple suivant

```
class Test {
  public static void main (String [] arg) {
    String t = "coucou" ;
    String u = "coucou" ;
    String v = "cou" ;
    String w = v + v ;
    System.out.println("t==u : " + (t == u) + ", t==w : " + (t == w)) ;
  }
}
```

Une fois compilé et lancé, ce programme affiche `t==u : true, t==w : false`. Ce qui révèle que les chaînes (objets) référencés par `t` et `u` sont exactement les mêmes, tandis que `w` est une autre chaîne.



La plupart du temps, un programme a besoin de savoir si deux chaînes ont exactement les mêmes caractères et non pas si elles occupent la même zone de mémoire. Il en résulte principalement qu'il ne faut pas tester l'égalité des chaînes (et à vrai dire des objets en général) par `==`. Dans le cas des chaînes, il existe une méthode `equals` spécialisée (voir 6.1.3) qui compare les chaînes caractère par caractère. La méthode `equals` est l'égalité *structurelle* des chaînes.

3.2 Types

Généralement un *type* est un ensemble de valeurs. Ce qui ne nous avance pas beaucoup ! Disons plutôt qu'un type regroupe des valeurs qui vont naturellement ensemble, parce qu'elles ont des représentations en machine identiques (**byte** occupe 8 bits en machine, **int** en occupe 32), ou surtout parce qu'elles vont naturellement ensemble (un objet **Point** est un point du plan, un objet **Object** est un objet).

3.2.1 Typage statique

Java est un langage typé statiquement, c'est-à-dire que si l'on écrit un programme incorrect du point de vue des types, alors le compilateur refuse de le compiler. Il s'agit non pas d'une contrainte irraisonnée imposée par des informaticiens fous, mais d'une aide à la mise au point des programmes : la majorité des erreurs stupides ne passe pas la compilation. Par exemple, le programme suivant contient deux erreurs de type :

```
class MalType {

  static int incr (int i) { return i+1 ; }

  static void mauvais() {
    System.out.println(incr(true)) ; // Mauvais type
    System.out.println(incr()) ;     // Oubli d'argument
  }
}
```

La compilation de la classe **MalType** échoue :

```
% javac MalType.java
MalType.java:9: Incompatible type for method. Can't convert boolean to int.
     System.out.println(incr(true)) ; // Mauvais type
                             ^
MalType.java:10: No method matching incr() found in class MalType.
     System.out.println(incr()) ;     // Oubli d'argument
                            ^
2 errors
```

Le système de types de Java assez puissant et les classes permettent certaines audaces. La plus courante se voit très bien dans l'utilisation de **System**`.out.println` (afficher une ligne sur la console), on peut passer n'importe quoi ou presque en argument, séparé par des « `+` » :

```
System.out.println ("booléen :" + (10 < 11) + "entier : " + (314*413)) ;
```

Cela peut se comprendre si on sait que + est l'opérateur de concaténation sur les chaînes, que `System.out.println` prend une chaîne en argument et que le compilateur insère des conversions là où il sent que c'est utile.

Il y a huit types scalaires, à savoir d'abord quatre types « entier », **byte**, **short**, **int** et **long**. Ces entiers sont en fait des entiers modulo 2^p (avec p respectivement égal à 8, 16, 32 et 64) les représentants des classes d'équivalence modulo 2^p sont centrés autour de zéro, c'est-à-dire compris entre -2^{p-1} (inclus) et 2^{p-1} (exclu). On dit aussi que les quatre types entiers correspondent aux entiers représentables sur 8, 16, 32 et 64 chiffres binaires, selon la technique dite du complément à la base (l'opposé d'un entier n est $2^p - n$).

Les autres types scalaires sont les booléens **boolean** (deux valeurs **true** et **false**), les caractères **char** et deux sortes de nombres flottants, simple précision **float** et double précision **double**. L'économie de mémoire réalisée en utilisant les **float** (sur 32 bits) à la place des **double** (64 bits) n'a d'intérêt que dans des cas spécialisés.

Les tableaux sont un cas à part (voir 3.6). Les classes sont des types pour les objets. Mais attention, les types sont en fait bien plus une propriété du source des programmes, que des valeurs lors de l'exécution. Nous essayons d'éviter de parler du « type » d'un objet. En revanche, il n'y a aucune difficulté à parler du type d'une variable qui peut contenir un objet, ou du type d'un argument objet.

3.2.2 Conversion de type

La syntaxe de la conversion de type est simple

(*type*) *expression*

La sémantique est un peu moins simple. Une conversion de type s'adresse d'abord au compilateur. Elle lui dit de changer son opinion sur *expression*. Par exemple, comme nous l'avons déjà vu en page 179

```
Pair p = ... ;
System.out.println((Object)p) ;
```

dit explicitement au compilateur que l'expression `p` (normalement de type **Pair**) est vue comme un **Object**. Comme **Pair** est une sous-classe de **Object** (ce que sait le compilateur), ce changement d'opinion est toujours possible et `(Object)p` ne correspond à aucun calcul au cours de l'exécution. En fait, dans ce cas d'une conversion vers un sur-type, on peut même omettre la conversion explicite, le compilateur saura changer son opinion tout seul si besoin est.

Il n'en va pas de même dans l'autre sens

```
Object o = ... ;
Pair p = (Pair)o ;
System.out.println(p.x) ;
```

Le compilateur accepte ce source, la conversion est nécessaire pour le faire changer d'opinion sur la valeur d'abord rangée dans `o`, puis dans `p` : cet objet est finalement une paire, et possède donc une variable d'instance `x`. Mais ici, rien ne le garantit, et l'expression `(Pair)o` correspond à une vérification lors de l'exécution. Si cette vérification échoue, alors le programme échoue aussi (en lançant l'exception **ClassCastException**). Il est malheureusement parfois nécessaire d'utiliser de telles conversions (vers un sous-type) voir III.3.2, même quand on programme proprement en ne mélangeant pas des objets de classes distinctes.

On peut aussi convertir le type des scalaires, cette fois les conversions entraînent des transformations des valeurs converties, car les représentations internes des scalaires ne sont pas toutes identiques. Par exemple, si on change un **int** (32 bits) en **long** (64 bits), la machine réalise un certain travail. Ce calcul n'est pas ici une vérification, mais un changement de représentation. La plupart des conversions de types entre scalaires restent implicites et sont effectuées à l'occasion des opérations. Si par exemple on écrit `1.5 + 2`, alors le compilateur arrive à comprendre `1.5 + (double)2`, afin d'effectuer une addition entre **double**. Il y a un cas où on insère soit-même ce type de conversions.

```
// a et b sont des int, on veut calculer le pourcentage a/b
double p = 100 * (((double)a)/b) ;
```

(Notez l'abondance de parenthèses pour bien spécifier les arguments de la conversion et des opérations.) Si on ne change pas explicitement un des arguments de la division en **double**, alors « `/` » est la division euclidienne, alors que l'on veut ici la division des flottants. On aurait d'ailleurs pu écrire :

```
double p = (100.0 * a) / b ;
```

En effet, le compilateur change alors `a` en **double**, pour avoir le même type que l'autre argument de la multiplication. Ensuite, la division est entre un flottant et un **int**, et ce dernier est converti afin d'effectuer la division des flottants.

Le compilateur n'effectue jamais tout seul une conversion qui risque de faire perdre de l'information. À la place, quand une telle conversion est nécessaire pour typer le programme, il échoue. Par exemple, prenons la partie entière d'un **double**.

```
// d est une variable de type double, on veut prendre sa partie entière
int e = d ;
```

Le compilateur, assez bavard, nous dit :

```
T.java:4: possible loss of precision
found   : double
required: int
    int e = d ;
            ^
```

Dans ce cas, on doit prendre ses responsabilités et écrire

```
int e = (int)d ;
```

Il faut noter que nous avons effectivement pris le risque de faire n'importe quoi. Si `d` est trop gros pour que sa partie entière tienne dans 32 bits (supérieure à $2^{31} - 1$), alors on a un résultat étrange. Le programme

```
double d = 1e100 ; // 10^100
System.out.println(d + ", " + (int)d) ;
```

conduit à afficher `1.0E100, 2147483647`.

3.2.3 Complément : caractères

Les caractères de Java sont définis par deux normes internationales synchronisées ISO/-CEI 10646 et Unicode. Le nom générique le plus approprié semblant être UTF-16. En simplifiant, une valeur du type **char** occupe 16 chiffres binaires et chaque valeur correspond à un caractère. C'est simplifié parce qu'en fait Unicode définit plus de 2^{16} caractères et qu'il faut parfois plusieurs **char** pour faire un caractère Unicode. Dans la suite nous ne tenons pas compte de cette complexité supplémentaire introduite notamment pour représenter tous les idéogrammes chinois.

Un **char** a une double personnalité, est d'une part un caractère (comme *'a'*, *'é'* etc.) et d'autre part un entier sur 16 bits (disons le « code » du caractère), qui contrairement à **short** est toujours positif. La première personnalité d'un caractère se révèle quand on l'affiche, la seconde quand on le compare à un autre caractère. Les 128 premiers caractères (c'est-à-dire ceux dont les codes sont compris entre 0 et 127) correspondent exactement à un autre standard international bien plus ancien, l'ASCII.

L'ASCII regroupe notamment les chiffres de *'0'* à *'9'* et les lettres (non-accentuées) minuscules et majuscules, mais aussi un certain nombre de caractères de contrôle, dont les plus fréquents expriment le « *retour à la ligne* ». Une petite digression va nous montrer que la standardisation du jeu de caractères n'est malheureusement pas suffisante pour tout normaliser. En Unix un retour à la ligne s'exprime par le caractère *line feed* noté *'\n'*, en Windows c'est la séquence d'un *carriage return* noté *'\r'* et d'un *line feed*, et en Mac OS, c'est un *carriage return* tout seul !

Le plus souvent ces détails restent cachés, par exemple **System**.out.println() effectue toujours un retour à la ligne sur l'affichage de la console, c'est le code de bibliothèque qui se charge de fournir les bons caractères à la console selon le système sur lequel le programme est en train de s'exécuter. Toutefois des problèmes peuvent surgir en cas de transfert de fichiers d'un système à l'autre...

3.3 Déclarations

De façon générale, une déclaration établit une correspondance entre un *nom* et une construction nommable (variable, méthode, mais aussi constante camouflée en variable). Les déclarations de variable sont de la forme suivante :

modifiers type name ;

Les *modifiers* sont des mots-clés (**static** , spécification de visibilité **private** etc., et **final** pour les constantes), le *type* est un type (genre **int**, **int** [] ou **String**) et *name* est un nom de variable. Une bonne pratique est d'initialiser les variables dès leur déclaration, ça évite bien des oublis. Pour ce faire :

modifiers type name = *expression* ;

Où *expression* est une expression du bon type. Par exemple, voici trois déclarations de variables, de types respectifs « entier », chaîne et tableau de chaîne :

```
int i = 0;
String message = "coucou" ;
String [] jours =
  {"dimanche", "lundi", "mardi", "mercredi",
  "jeudi", "vendredi", "samedi"} ;
```

Les déclarations de « variable » sont en fait de trois sortes :

(1) Dans une classe,
   (a) Une déclaration avec le modificateur **static** définit une variable (un champ) de la classe.
   (b) Une déclaration sans le modificateur **static** définit une variable d'instance des objets de la classe.
(2) Dans une méthode, une déclaration de variable définit une variable locale. Dans ce cas seul le modificateur **final** est autorisé.

Ces trois sortes de « variables » sont toutes des cases nommées mais leurs comportements sont différents, voire très différents : les variables locales sont autonomes, et leurs noms obéissent à la portée lexicale (voir "structure de bloc" dans 3.4), les deux autres « variables » existent comme sous-parties respectivement d'une classe et d'un objet ; elles sont normalement désignées comme C.x et o.x, où C et o sont respectivement une classe et un objet.

Sur une note plus mineure, les variables de classe et d'objet non-initialisées explicitement contiennent en fait une valeur par défaut qui est fonction de leur type — zéro pour les types d'entiers et de flottants (et pour **char**), **false** pour les **boolean**, et **null** pour les objets. En revanche, comme le compilateur Java refuse l'accès à une variable locale quand celle-ci n'a pas été initialisée explicitement, ou affectée avant lecture, il n'y a pas lieu de parler d'initialisation par défaut des variables locales.

Les déclarations de méthode ne peuvent se situer qu'au niveau des classes et suivent la forme générale suivante :

modifiers type name `(`*args*`)`

Les *modifiers* peuvent être, **static** (méthode de classe), une spécification de visibilité, **final** (redéfinition interdite), **synchronized** (accès en exclusion mutuelle, voir le cours suivant) et **native** (méthode écrite dans un autre langage que Java). La partie *type* est le type du résultat de la méthode (**void** si il n'y en a pas), *name* est le nom de la méthode et *args* sont les déclarations des arguments de la méthode qui sont de bêtes déclarations de variables (sans le « `;` » final) séparées par des virgules « `,` ». L'ensemble de ces informations constitue la *signature* de la méthode.

Suit ensuite le corps de la méthode, entre accolades « `{` » et « `}` ». Il est notable que les déclarations d'arguments sont des déclarations de variables ordinaires. En fait on peut voir les arguments d'une méthode comme des variables locales presque normales. À ceci près qu'il n'y a pas lieu d'initialiser les arguments ce qui d'ailleurs n'aurait aucun sens puisque les arguments sont initialisés à chaque appel de méthode.

3.4 Principales expressions et instructions

Nous décrivons maintenant ce qui se trouve dans le corps des méthodes, c'est-à-dire le code qui fait le vrai travail. Le corps d'une méthode est une séquence d'*instructions*). Les instructions sont *exécutées*. Une instruction (par ex. une affectation) peut inclure une *expression*. Les expressions sont *évaluées* en un résultat.

La distinction entre expressions (dont l'évaluation produit un résultat) et instructions (dont l'exécution ne produit rien) est assez hypocrite, car toute expression suivie de « `;` » devient une instruction (le résultat est jeté).
Les expressions les plus courantes sont :

Constantes Soit `1` (entier), **true** (booléen), *`"coucou !"`* (chaîne), etc. Une constante amusante est **null**, qui est un objet sans champs ni méthodes.

Usage de variable Soit « *x* », où *x* est le nom d'une variable (locale) déclarée par ailleurs.

*Mot-clé **this*** Dans une méthode dynamique , **this** désigne l'objet dont on a appelé la méthode. Dans un constructeur, **this** désigne l'objet en cours de construction. Il en résulte que **this** n'est jamais **null**, car si on en est arrivé à exécuter un corps de méthode, c'est bien que l'objet dont a appelé la méthode existait.

Accès à un champ Si *x* est un nom de champ et classe un nom de classe *C*, alors « *C*`.`*x* » désigne le contenu du champ *m* de *C*. De même « *objet*`.`*x* » désigne le champ de nom *x* de l'objet *objet*. Notez que, contrairement à *x*, *objet* n'est pas forcément un nom, c'est une expression dont la valeur est un objet. Heureusement ou malheureusement, il existe des notations abrégées qui allègent l'écriture (voir 3.5), mais font parfois passer l'accès à un champ pour un usage de variable.

Appel de méthode C'est un peu comme pour les champs :

statique Soit, *C*`.`*m*`(...)`,

dynamique ou bien, *objet*`.`*m*`(...)`.

Où *m* est un nom de méthode et `(...)` est une séquence d'expressions séparées par des virgules. Notez bien que les mêmes notations abrégées que pour les champs s'appliquent au nom de la méthode. Incidemment, si une méthode a un type de retour **void**, son appel n'est pas vraiment une expression, c'est une instruction.

Appel de constructeur Généralement de la forme **new** *C* `(...)`. La construction des tableaux est l'occasion de nombreuses variantes, voir 3.6

Usage des opérateurs Par exemple `i+1` (addition) ou `i == 1 || i == 2` (opérateur égalité et opérateur ou logique). Quelques opérateurs inattendus sont donnés en 7.2. Notons qu'un « *opérateur* » en informatique est simplement une fonction dont l'application se fait par une syntaxe spéciale. L'application des opérateurs est souvent *infixe*, c'est-à-dire que l'opérateur apparaît entre ses arguments. Mais elle peut être préfixe, c'est-à-dire que l'opérateur est avant son argument, comme dans le cas de la négation des booléens `!` ; ou encore postfixe, comme pour l'opérateur de post-incrément `i++`. En Java, comme souvent, les opérateurs eux-mêmes sont exclusivement composés de caractères non alphabétiques (`+`, `-`, `=`, etc.).

Accès dans les tableaux Par exemple `t[i]`, où `t` est un tableau défini par ailleurs. Il n'est pas surprenant que l'on puisse mettre une expression à la place de `i`. Il est un peu plus surprenant que cela soit également le cas pour `t`, comme par exemple dans `t[i][j]`, à comprendre comme `(t[i])[j]` (`t` est un tableau de tableaux).

Affectation Par exemple `i = 1`, l'expression à droite de `=` est calculée sa valeur est rangée dans la variable donnée à gauche de `=`. En fait, on peut trouver autre chose qu'une variable à gauche de `=`, on peut trouver tout ce qui désigne une case de mémoire, par exemple, un élément de tableau `t[i]` ou une désignation de champ *objet*`.`*x*.

L'affectation est une expression dont le résultat est la valeur affectée. Ce qui permet des trucs du genre `i = j = 0`, pour initialiser `i` et `j` à zéro. Cela se comprend si on lit cette expression comme `i = (j = 0)`.

Expression parenthésée Si *e* est une expression, alors `(`*e*`)` est aussi une expression. Cela permet essentiellement de contourner les priorités relatives des opérateurs, mais aussi de rendre un source plus clair. Par exemple, on peut écrire `(i == 1) || (i == 2)`, c'est peut-être plus lisible que `i == 1 || i == 2`.

Java a beaucoup emprunté au langage C, il reprend quelques expressions assez peu ordinaires.

Incrément, décrément Soit `i` variable de type entier (en fait, soit *e* désignation de case mémoire qui contient un entier). Alors l'expression `i++` range `i+1` dans `i` et renvoie l'ancienne valeur de `i`. L'expression `i--` fait la même chose avec `i-1`. Enfin l'expression `++i` (resp. `--i`) est similaire, mais elle renvoie la valeur incrémentée (resp. décrémentée) de `i` en résultat.

Affectations particulières La construction `i` *op*`=` *expression* est sensiblement équivalente à `i = i` *op expression*. Par exemple :

```
i *= 2
```

range deux fois le contenu de `i` dans `i` et renvoie donc ce contenu doublé. Les finauds noteront que `++i` est aussi `i += 1`.

Ces expressions avancées sont géniales, mais il est de bon goût de les employer avec parcimonie. Que l'on essaie de deviner ce que fait `t[++i] *= --t[i++]` et on comprendra.

Les instructions les plus courantes sont les suivantes :

Expression comme une instruction : « *e* `;` » Évidemment, cette construction n'est utile que si *e* fait des effets de bord (c'est-à-dire fait autre chose que rendre son résultat). C'est bien sûr le cas d'une affectation.

Séquence On peut grouper plusieurs instructions en une séquence d'instruction, les instructions s'exécutent dans l'ordre.

```
i = i + 1;
i = i + 1;
```

Déclarations de variables locales Une déclaration de variable (suivie de ;) est une instruction qui réserve de la place pour la variable déclarée et l'initialise éventuellement.

```
int i = 0;
```

Il ne faut pas confondre affectation et déclaration, dans le premier cas, on modifie le contenu d'une variable qui existe déjà, dans le second on crée une nouvelle variable.

Bloc On peut mettre une séquence d'instructions dans un bloc {...}. La portée des déclaration internes au bloc s'éteint à la sortie du bloc. Par exemple, le programme

```
int i = 0 ;
{
  int i = 1; // Déclaration d'un nouvel i
  System.out.println("i=" + i);
}
System.out.println("i=" + i);
```

affiche une première ligne `i=1` puis une seconde `i=0`. Il faut bien comprendre que d'un affichage à l'autre l'usage de variable « `i` » ne fait pas référence à la même variable. Lorsque l'on veut savoir à quelle déclaration correspond un usage, la règle est de remonter le source du regard vers le haut, jusqu'à trouver la bonne déclaration. C'est ce que l'on appelle parfois, la portée lexicale.

Attention, seule la portée des variables est limitée par les blocs, en revanche l'effet des instruction passe allègrement les frontières de blocs. Par exemple, le programme

```
int i = 0 ;
{
  i = 1; // Affectation de i
  System.out.println("i=" + i);
}
System.out.println("i=" + i);
```

affiche deux lignes `i=1`.

Retour de méthode On peut dans le corps d'une méthode retourner à tout moment par l'instruction « **return** *expression*; », où *expression* est une expression dont le type est celui des valeurs retournées par la méthode.

Par exemple, la méthode `twice` qui double son argument entier s'écrit

```
int twice (int i) {
  return i+i ;
}
```

Si la méthode ne renvoie rien, alors **return** n'a pas d'argument.

```
void rien () {
  return ;
}
```

À noter que, si la dernière instruction d'une méthode est **return** ; (sans argument), alors on peut l'omettre. De sorte que la méthode rien s'écrit aussi :

```
void rien () { }
```

Conditionnelle C'est la très classique instruction **if** :

```
    if (i % 2 == 0) { // opérateur modulo
      System.out.println("C'est pair") ;
    } else {
      System.out.println("C'est impair") ;
    }
```

Instruction `switch` C'est une généralisation de la conditionnelle aux types d'entiers. Elle permet de « brancher » selon la valeur d'un entier (de **byte** à **long**, mais aussi **char**). Par exemple :

```
switch (i) {
  case -1:
    System.out.println("moins un") ;
    break ;
  case 0:
    System.out.println("zéro") ;
    break ;
  case 1:
    System.out.println("un") ;
    break ;
  default:
    System.out.println("beaucoup") ;
  }
}
```

Selon la valeur de l'entier `i` on sélectionnera l'une des trois clauses **case**, ou la clause par défaut **default**. Il faut surtout noter le **break**, qui renvoie le contrôle à la fin du **switch**. En l'absence de **break** l'exécution se poursuit en séquence, donc la clause suivante est exécutée. Ainsi si on omet les **break** et que `i` vaut `-1` on a l'affichage

```
moins un
zéro
un
beaucoup
```

Cette abominable particularité permet de grouper un peu les cas. Par exemple,

```
switch (i) {
  case -1: case 0: case 1:
    System.out.println("peu") ; break ;
  default:
    System.out.println("beaucoup") ;
}
```

Noter que si la clause se termine par **return**, alors **break** n'est pas utile.

```
static String estimer(int i) {
  switch (i) {
    case -1: return "moins un" ;
    case 0: return "zéro" ;
    case 1: return "un" ;
    default: return "beaucoup" ;
  }
}
```

Boucle `while` C'est la boucle la plus classique, celle que possèdent tous les langages de programmation, ou presque. Voici les entiers de zéro à neuf.

```
  int i = 0 ;
```

```
while (i < 10) {
  System.out.println(i) ;
  i = i+1 ;
}
```

Soit **while** (*expression*) *instruction*, on exécute *expression*, qui est une expression booléenne, si le résultat est **false** c'est fini, sinon on exécute *instruction* et on recommence.

Boucle `do` C'est la même chose mais on teste la condition à la fin du tour de boucle, conclusion : on passe au moins une fois dans la boucle. Voici une autre façon d'afficher les entiers de 0 à 9.

```
int i = 0 ;
do {
  System.out.println(i) ;
  i = i+1 ;
} while (i < 10)
```

Boucle `for` C'est celle du langage C.

```
for (int i=0 ; i < 10 ; i = i+1)
  System.out.println(i);
```

La syntaxe générale est

```
for (e_init ; e_cond ; e_next)
  instruction
```

L'expression e_{init} est évaluée une fois initialement, exceptionnellement « l'expression » e_{init} peut être une déclaration de variable, auquel cas la portée de la variable est limitée à la boucle. L'expression e_{cond} est de type **boolean**, c'est la condition testée avant chaque itération (y compris la première), l'itération a lieu si e_{cond} vaut **true**. Enfin, e_{next} est évaluée à la fin de chaque itération. Autrement dit une boucle **for** est presque la même chose que la boucle **while** suivante.

```
{
   e_init ;
   while (e_cond) {
    instruction
    e_next ;
   }
}
```

Enfin, notez que e_{cond} peut être omise, en ce cas la condition est considérée vraie. Cette particularité est principalement utilisée dans l'idiome de la boucle infinie.

```
for ( ; ; ) { // Boucle infinie, on sort par break ou return
  ...
}
```

Gestion du contrôle Certaines instructions permettent de « *sauter* » quelque-part de l'intérieur des boucles. Ce sont des formes polies de `goto`. Il s'agit de **break**, qui fait sortir de la boucle, et de **continue** qui commence l'itération suivante en sautant par dessus ce qui reste de l'itération en cours.

Ainsi pour rechercher si l'entier `foo` est présent dans le tableau `t`, on peut écrire

```
boolean trouve = false ;
for (int i = 0 ; i < t.length ; i++) {
  if (t[i] == foo) {
```

```
      trouve = true ;
      break ;
    }
  }
  // trouve == true <=> foo est dans t
```

Ou encore, si on veut cette fois compter les occurrences de `foo` dans `t`, on peut écrire

```
int count = 0 ;
 for (int i = 0 ; i < t.length ; i++) {
   if (t[i] != foo) continue ;
   count++ ;
 }
```

Noter que dans les deux cas on fait des choses un peu compliquées. Dans le premier cas, on pourrait faire une méthode et utiliser **return**, ou une boucle **while** sur `trouve`. Dans le second cas, on pourrait écrire le test d'égalité. Dans certaines situations, ces instructions sont pratiques (**break** plus fréquemment que **continue**).

3.5 Notations complètes et abrégées

Pour les classes (et les constructeurs)

Quand une classe C est définie dans un package *pkg*, son nom est *pkg*.C. Il en va d'ailleurs de même pour les constructeurs de la classe C. Ainsi, le nom complet de la classe **Point** définie dans le package `java.awt` est `java.awt.`**Point**. On est donc censé écrire par exemple

```
java.awt.Point p = new java.awt.Point(100, 50) ;
```

C'est très lourd, pour l'éviter on peut introduire la déclaration suivante au début du fichier source

```
import java.awt.Point ;
```

Et si le nom de classe **Point** survient plus tard dans le fichier, le compilateur sait que l'on a voulu dire `java.awt.`**Point**.

Une autre déclaration possible en début de fichier est

```
import java.awt.* ;
```

Cela revient aux déclarations `java.awt.`C effectuées pour toutes les classes C du package `java.awt`.

Pour les champs et les méthodes

Une variable (une méthode) d'une classe C est normalement désignée en la préfixant par C. ; mais dans le code de la classe elle-même le nom de la variable suffit. De même, dans le code des méthodes des objets (et des constructeurs) une variable d'instance (une méthode) est normalement désignée en la préfixant par **this**. ; mais le nom de la variable (méthode) suffit.

Les notations complètes permettent parfois d'insister sur la nature d'une variable, et de surmonter le masquage des variables d'instance et de classe par des variables locales. Ces masquages ne sont pas toujours malvenus. Par exemple, nous avons tendance à donner aux constructeurs des arguments homonymes aux variables d'instance initialisées. Ce qui reste possible en explicitant les variables d'instance en tant que telles.

```
class List {
  int val ; List next ;

  List (int val, List next) { this.val = val ; this.next = next ; }
}
```

Dans le constructeur ci-dessus, **this**`.val` est la variable d'instance, `val` tout court est l'argument.

3.6 Les tableaux

3.6.1 Les types

Les tableaux sont presque des objets, presque car mais ils n'ont pas vraiment de classe, tout se passe plus ou moins comme si, pour chaque type *type*, un type des tableaux dont les éléments sont de type *type* tombait du ciel. Ce type des tableaux dont les éléments sont de type *type*, se note (comme dans le langage C) « *type*`[]` ». En particulier on a :

```
int [] ti ;         // tableau d'entiers
String [] ts ;      // tableau de chaînes
```

Les tableaux « à plusieurs dimensions » ou matrices, n'existent pas en tant que tels, ce sont en fait des tableaux de tableaux.

```
String [] [] tts ; // tableau de tableaux de chaînes
```

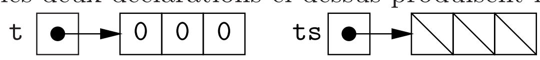
3.6.2 Valeurs tableaux

Contrairement à bien des langages, il est possible en Java, de fabriquer des tableaux directement. On distingue :

- La constante **null** avec laquelle on ne peut rien faire est aussi un tableau dont on ne peut rien faire.
- L'appel de constructeur : **new** *type* `[`*taille*`]`, où *type* est le type des éléments du tableau et *taille* la taille du tableau. Par exemple, pour déclarer et initialiser un tableau `t` de trois entiers, et un tableau `ts` de trois chaînes, on écrit :

  ```
  static int [] ti = new int [3] ;
  static String [] ts = new String [3] ;
  ```

  Les éléments du tableau sont initialisés « par défaut », à une valeur qui dépend de leur type — zéro pour les entiers et flottants, **false** pour les booléens, et **null** pour les objets. C'est exactement le même principe que pour les variables de classes et d'instance initialisées par défaut (voir 3.3). Ainsi les deux déclarations ci-dessus produisent l'état mémoire simplifié

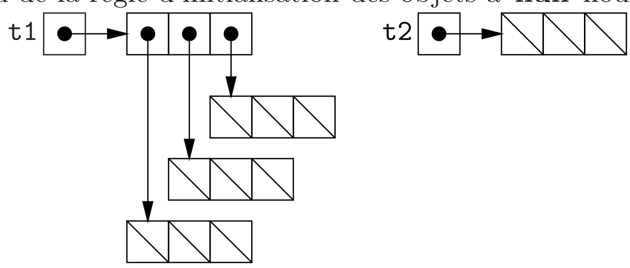
  La notation par constructeur s'applique aussi aux tableaux de tableaux.

  ```
  static String [] [] t1 = new String [3][3]  ; // 3 x 3 chaînes.
  ```

  L'aspect tableau de tableaux apparaît aussi. Il est possible d'omettre les dernières tailles.

  ```
  static String [] [] t2 = new String [3][]  ; // 3 tableaux de chaînes
  ```

  Ce qui, compte tenu de la règle d'initialisation des objets à **null** nous donne.

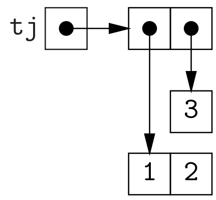
- Un tableau explicite, sous la forme « `{`e_1`,` e_2`,...` e_n`}` », où les e_i sont des expressions du type des éléments du tableau. Par exemple, pour déclarer et initialiser le tableau `ti` des seize premiers entiers, on écrira :

  ```
  int [] ti = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16} ;
  ```

  Dans le cas des tableaux de tableaux, on écrira :

```
int [] [] tj = {{1,2}, {3}};
```

Ce qui nous donne ceci :



Autrement dit `tj[0]` est le tableau `{1,2}` et `tj[1]` est le tableau `{3}`. Notons au passage l'existence du tableau vide (tableau à zéro case) `{}` à ne pas confondre avec **null**.

- La notation explicite ci-dessus est une espèce d'abréviation réservée à l'initialisation des variables. Dans un autre contexte, il faut indiquer le type du tableau en invoquant le constructeur. Ainsi **new int** `[] {1,2}` et **new int** `[] [] {{1,2}, {3}}` sont des valeurs tableaux.

3.6.3 Opérations sur les tableaux

Il y a deux opérations primitives :

- Accéder à un élément : t`[`e_i`]`, où t est le tableau et e_i une expression de type entier. *Attention, les indices commencent à zéro.*
- Trouver la longueur d'un tableau : c'est une syntaxe objet, tout se passe comme si la longueur était le champ `length` du tableau, soit t`.length`.

La classe de bibliothèque **Arrays** (package `java.util`) fournit quelques méthodes toutes statiques qui opèrent sur les tableaux, et notamment des tris.

3.7 Passage par valeur

La règle générale de Java est que les arguments sont passés par valeur. Cela signifie que l'appel de méthode se fait par copie des valeurs passées en argument et que chaque appel de méthode dispose de sa propre version des paramètres. On doit noter que Java est ici parfaitement cohérent : si une méthode `f` possède un argument **int** `x`, alors l'appel `f(2)` revient à créer, pour la durée de l'exécution de cet appel une nouvelle variable (de nom `x`) qui est initialisée à `2`.

Examinons un exemple.

```
static void dedans(int i) {
  i = i+2 ;
  System.out.println("i=" + i)
}

static void dehors(int i) {
   System.out.println("i=" + i) ;
   dedans(i) ;
   System.out.println("i=" + i) ;
 }
```

L'appel de `dehors(0)` se traduira par l'affichage de trois lignes `i=0`, `i=2`, `i=0`. C'est très semblable à l'exemple sur la structure de bloc. Ce qui compte c'est la portée des variables. Le résultat est sans doute moins inattendu si on considère cet autre programme rigoureusement équivalent :

```
static void dedans(int j) { //  change ici i -> j
  j = j+2 ;
  System.out.println("j=" + j)
}
```

```
  static void dehors(int i) {
     System.out.println("i=" + i) ;
     dedans(i) ;
     System.out.println("i=" + i) ;
   }
```

Mais ce que l'on doit bien comprendre c'est que le nom de l'argument de `dedans`, `i` ou `j` n'a pas d'importance, c'est une variable muette et heureusement.

La règle du passage des arguments par valeur s'applique aussi aux objets, mais alors c'est une référence vers l'objet qui est copiée. Ceci concerne en particulier les tableaux.

```
// Affichage d'un tableau (d'entiers) passé en argument
  static void affiche(int[] t) {
    for (int i = 0 ; i < t.length ; i++)
      System.out.print ("t[" + i + "] = " + t[i] + " ") ;
    System.out.println() ; // sauter une ligne
  }

// trois façons d'ajouter deux
  static void dedans (int[] t) {
    t[0] = t[0] + 2;
    t[1] += 2;
    t[2]++ ; t[2]++ ;
  }

  static void dehors () {
    int[] a = {1,2,3} ; // déclaration et initialisation d'un tableau
    affiche (a) ;
    dedans (a) ;
    affiche (a) ;
  }
```

L'affichage est le suivant :

```
1 2 3
3 4 5
```

Ce qui illustre bien qu'il n'y a, tout au cours de l'exécution du programme qu'un seul tableau, `a`, `t` étant juste des noms différents de ce même tableau. On peut aussi renommer l'argument `t` en `a` ça ne change rien.

4 Exceptions

Une fois un programme accepté par le compilateur, il n'est pas garanti qu'aucune erreur se produira. Bien au contraire, l'expérience nous apprend que des erreurs se produiront.

4.1 Exceptions lancées par le système d'exécution

Certaines erreurs empêchent l'exécution de se poursuivre dès qu'elles sont commises. Par exemple, si nous cherchons à accéder à la quatrième case d'un tableau qui ne comprend que trois cases, le système d'exécution ne peut plus rien faire de raisonnable et il fait échouer le programme.

```
class Test {
  public static void main(String args[])  {
    int [] a = {2, 3, 5};
```

```
    System.out.println(a[3]);
  }
}
```

Nous obtenons,

```
% java Test
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
        at Test.main(Test.java:4)
```

Le message affiché nous signale que notre programme a échoué à cause d'une *exception* dont le nom est **ArrayIndexOutOfBoundsException** (qui semble à peu près clair). L'affichage offre quelques bonus, l'indice fautif, et surtout la ligne du programme qui a lancé l'exception. D'autres exemples classiques d'erreur sanctionnée par une exception sont le déréférencement de **null** (sanctionné par **NullPointerException**) ou la division par zéro (sanctionnée par **ArithmeticException**).

Lancer une exception ne signifie pas du tout que le programme s'arrête immédiatement. On peut expliquer le mécanisme ainsi : l'exception remplace un résultat attendu (par exemple le résultat d'une division) et ce résultat « exceptionnel » est propagé à travers toutes les méthodes en attente d'un résultat, jusqu'à la méthode `main`. Le système d'exécution de Java affiche alors un message pour signaler qu'une exception a été lancée. Pour préciser cet effet de propagation considérons l'exemple suivant d'un programme **Square**.

```
class Square {
  static int read(String s) {
    return Integer.parseInt(s); // Lire l'entier en base 10, voir 6.1.1
  }

  public static void main(String[] args) {
    int i = read(args[0]);
    System.out.println(i*i);
  }
}
```

Le programme **Square** est censé afficher le carré de l'entier passé sur la ligne de commande.

```
% java Square 11
121
```

Mais si nous donnons un argument qui n'est pas la représentation d'un entier en base dix, nous obtenons ceci :

```
% java Square bonga
Exception in thread "main" java.lang.NumberFormatException: For input string: "bonga"
  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
  at java.lang.Integer.parseInt(Integer.java:447)
  at java.lang.Integer.parseInt(Integer.java:497)
  at Square.read(Square.java:3)
  at Square.main(Square.java:7)
```

Où on voit bien la suite des appels en attente que l'exception a du remonter avant d'atteindre `main`.

Le mécanisme des exceptions est intégré dans le langage, ce qui veut dire que nous pouvons manipuler les exceptions. Ici, la survenue de l'exception trahit une erreur de programmation : en réaction à une entrée incorrecte le programme devrait normalement avertir l'utilisateur et donner une explication. Pour ce faire on *attrape* l'exception à l'aide d'une instruction spécifique.

```
public static void main(String[] args) {
  try {
    int i = read(args[0]);
    System.out.println(i*i);
  } catch (NumberFormatException e) {
    System.err.println("Usage : java Square nombre") ;
  }
}
```

Et on obtient alors.

```
% java Square bonga
Usage : java Square nombre
```

Sans trop rentrer dans les détails, l'instruction **try** { $instr_{try}$ } **catch** ($E\ e$) { $instr_{fail}$ } s'exécute comme l'instruction $instr_{try}$, mais si une exception E est lancée lors de cette exécution, alors l'instruction $instr_{fail}$ est exécutée. En ce cas, l'exception attrapée disparaît. Dans l'instruction $instr_{fail}$ on peut accéder à l'exception attrapée par le truchement de la variable e. Cela permet par exemple d'afficher l'exception coupable par `e.toString()` comme ici, ou plus fréquemment le message contenu dans l'exception par `e.getMessage()`.

```
public static void main(String[] args) {
  try {
    int i = read(args[0]);
    System.out.println(i*i);
  } catch (NumberFormatException e) {
    System.err.println("Usage : java Square nombre") ;
    System.err.println("Échec sur : " + e) ;
  }
}
```

Et on obtient alors

```
% java Square bonga
Usage : java Square nombre
Échec sur : java.lang.NumberFormatException: For input string: "bonga"
```

Pour être tout à fait complet sur l'exemple de **Square**, il y a encore un problème. Nous écrivons `read(arg[0])` sans vérification que le tableau `arg` possède bien un élément. Nous devons aussi envisager d'attraper une exception **ArrayIndexOutOfBoundsException**. Une solution possible est d'écrire :

```
private static void usage() {
  System.err.println("Usage : java Square nombre") ;
  System.exit(2) ;
}

public static void main(String[] args) {
  try {
    int i = read(args[0]);
    System.out.println(i*i);
  } catch (NumberFormatException _) {
    usage() ;
  } catch (ArrayIndexOutOfBoundsException _) {
    usage() ;
  }
}
```

On note qu'il peut en fait y avoir plusieurs clauses **catch**. Les deux clauses **catch** appellent ici la méthode `usage`, qui affiche un bref message résumant l'usage correct du programme et arrête l'exécution.

4.2 Lancer une exception

Lorsque l'on écrit un composant quelconque, c'est-à-dire du code réutilisable par autrui, on se trouve parfois dans la situation de devoir signaler une erreur. Il faut alors procéder exactement comme le système d'exécution de Java et lancer une exception. Ainsi l'utilisateur de notre composant a l'opportunité de pouvoir réparer son erreur. Au pire, un message compréhensible sera affiché.

Supposons par exemple une classe des piles d'entiers. Une tentative de de dépiler sur une pile vide se solde par une erreur, on lance alors une exception par l'instruction **throw**.

```
int pop() {
  if (isEmpty()) throw new Error ("Pop: empty stack") ;
  ...
}
```

Il apparaît clairement qu'une exception est un objet d'une classe particulière, ici la classe **Error**. En raison de sa simplicité nous utilisons systématiquement **Error** dans nos exemples.

Si vous prenez la peine de lire la documentation vous verrez que dans l'esprit des auteurs de la bibliothèque, les exceptions **Error** ne sont pas censées être attrapées.

> An `Error` [...] indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions.

Pour signaler des erreurs réparables, la documentation encourage plutôt la classe **Exception**.

> The class `Exception` and its subclasses [...] indicates conditions that a reasonable application might want to catch.

En fait, il vaut mieux ne pas lancer **Exception**, mais plutôt une exception par nous définie (comme une sous-classe de **Exception**) ; ceci afin d'éviter les interférences avec les exceptions lancées par les méthodes de la bibliothèque.

Nous procédons donc en deux temps, d'abord déclaration de la classe de notre exception.

```
class Stack {
  ⋮
  static class Empty extends Exception { }
  ⋮
}
```

La classe **Empty** est définie comme un membre statique de la classe **Stack** (oui, c'est possible), de sorte que l'on la désigne normalement comme **Stack.Empty**. Ce détail mis à part, il s'agit d'une déclaration de classe normale, mais qui ne possède aucun membre en propre. Le code indique simplement que **Empty** est une sous-classe (voir 2.3 et 1.5) de **Exception**. Un constructeur par défaut **Empty ()** est implicitement fourni, qui se contente d'appeler le constructeur **Exception ()**.

Ensuite, notre exception est lancée comme d'habitude.

```
int pop() {
  if (isEmpty()) throw new Empty () ;
  ...
}
```

Mais alors, la compilation de la classe **Stack** échoue.

```
% javac Stack.java
Stack.java:13: unreported exception Stack.Empty;
              must be caught or declared to be thrown
    if (isEmpty()) throw new Empty () ;
                   ^
1 error
```

En effet, Java impose de déclarer qu'une méthode lance une exception **Exception** (mais pas une exception **Error**). On déclare que la méthode `pop` lance l'exception **Empty** par le mot clé **throws** (noter le « s »).

```
int pop() throws Empty {
    if (isEmpty()) throw new Empty () ;
    ...
}
```

On peut considérer que les exceptions lancées par une méthode font partie de sa signature (voir 3.3), c'est-à-dire font partie des informations à connaître pour pouvoir appeler la méthode.

Les déclarations obligatoires des exceptions lancées sont assez contraignantes, en effet il faut tenir compte non seulement des exceptions lancées directement, mais aussi de celles lancées indirectement par les méthodes appelées. Ainsi si on conçoit une méthode `remove(int n)` pour enlever n éléments d'une pile, on doit tenir compte de l'exception **Empty** éventuellement lancée par `pop`.

```
void remove (int n) throws Empty {
  for ( ; n > 0 ; n--) {
    pop() ;
  }
}
```

Noter qu'il est également possible pour `remove` de ne pas signaler que la pile comportait moins de n éléments. Cela revient à attraper l'exception **Empty**, et c'est d'autant plus facile que `remove` ne renvoie pas de résultat.

```
void remove (int n) {
  try {
    for ( ; n > 0 ; n--) {
      pop() ;
    }
  } catch (Empty e) { }
}
```

5 Entrées-sorties

Par entrées-sorties on entend au sens large la façon dont un programme lit et écrit des données, et au sens particulier l'interaction entre le programme et le système de fichiers de la machine.

Un fichier est un ensemble de données qui survit à la fin de l'exécution des programmes, et même, la plupart du temps, à l'arrêt de l'ordinateur. Les fichiers se trouvent le plus souvent sur le disque de la machine, mais ils peuvent aussi se trouver sur une disquette, ou sur le disque d'une autre machine et être accédés à travers le réseau. Un système d'exploitation tel qu'Unix, offre une abstraction des fichiers, généralement sous forme de *flux* (*stream*). Dans sa forme la plus simple le flux offre un accès séquentiel : on peut lire le prochain élément d'un flux (ce qui le consomme) ou écrire un élément dans un flux (généralement à la fin). Dans la majorité des cas (et surtout en Unix), les fichiers contiennent du texte, il est alors logique, vu de Java, de les considérer comme des flux de **char**. Par fichier texte, nous entendons d'abord fichier qu'un humain peut écrire, lire et comprendre, par exemple un source Java. Les fichiers qui ne s'interprètent pas ainsi sont dits *binaires*, par exemple une image ou un fichier de bytecode Java. Il est logique, vu de Java, de considérer les fichiers binaires comme des flux de **byte**. Par la suite nous ne considérons que les fichiers texte.

Quand vous lancez un programme dans une fenêtre de commandes (*shell*), il peut lire ce que vous tapez au clavier (flux d'entrée standard) et écrire dans la fenêtre (flux de sortie standard). Le shell permet de rediriger les flux standard vers des fichiers (par <*fichier* et >*fichier*), ce qui permet de lire et d'écrire des fichiers simplement. Enfin il existe un troisième flux standard, la sortie d'erreur standard normalement connectée à la fenêtre comme la sortie standard. Même si cela ne semble pas utile, il *faut* écrire les messages de diagnostic et d'erreur dedans, et nous allons voir pourquoi.

5.1 Le minimum à savoir : une entrée et une sortie standard

Nous connaissons déjà la sortie standard, c'est **System**.`out` — et **System**.`err` est la sortie d'erreur. Nous savons déjà écrire dans ces flux de classe **PrintStream**, par leurs méthodes `print` et `println`. Pour lire l'entrée standard (**System**.`in`, comme vous pouviez vous en douter), c'est un rien plus complexe, mais sans plus.

Il faut tout d'abord fabriquer un flux de caractères, objet de la classe, **Reader** (voir aussi 6.2.1) à partir de **System**.`in` :

```
Reader in = new InputStreamReader(System.in) ;
```

On note que l'objet **InputStreamReader** est rangé dans une variable de type **Reader**. Et en effet, **InputStreamReader** est une sous-classe de **Reader** (voir 2.3).

Ensuite, on peut lire le prochain caractère de l'entrée par l'appel `in.read()`. Cette méthode renvoie un **int**, avec le comportement un peu tordu suivant :

- Si un caractère est lu, il est renvoyé comme un entier.
- Si on est à la fin de l'entrée, `-1` est renvoyé. La fin de l'entrée est par vous signalée par « Control-d » au clavier et est la fin de fichier si il y a eu redirection.
- En cas d'erreur l'exception **IOException** est lancée. Ces erreurs sont du genre de celles détectées au niveau du système d'exploitation, comme la panne d'un disque (rare !), ou un délai abusif de transmission par le réseau (plus fréquent). Bref, l'exception est lancée quand, pour une raison ou une autre, la lecture est jugée impossible.

Voici **Cat** un filtre trivial qui copie l'entrée standard dans la sortie standard :

```
import java.io.*; // La classe Reader est dans le package java.io

class Cat {
  public static void main(String [] arg) {
    try {
      Reader in = new InputStreamReader(System.in) ;
      for ( ; ; ) {// Boucle infinie, on sort par break
        int c = in.read ();
        if (c == -1) break;
        System.out.print ((char)c);
      }
    } catch (IOException e) { // En cas de malaise
      System.err.print("Malaise : " + e.getMessage()) ;
      System.exit(2) ; // Terminer le programme sur erreur
    }
  }
}
```

Remarquons :

- Il y a un **try**... **catch** (voir 4.1), car la méthode `read` déclare lancer l'exception **IOException**, qui doit être attrapée dans `main` si elle n'est pas déclarée dans la signature de `main`. À la fin du traitement de l'exception, on utilise la méthode **System**.`exit` qui arrête le programme d'un coup. Le message d'erreur est écrit dans la sortie d'erreur.
- Pour afficher `c` comme un caractère, il faut le changer en caractère, par `(`**char**`)c`. Sinon, c'est un code de caractère qui s'affiche.

On peut tester le programme **Cat** par exemple ainsi :

```
% java Cat < Cat.java
import java.io.*; // La classe Reader est dans le package java.io
  ⋮
```

5.2 Un peu plus que le minimum : lire des fichiers

La commande `cat` de Unix est plus puissante que notre **Cat**. Si on donne des arguments sur la ligne de commande, ils sont interprétés comme des noms de fichiers. Le contenu des fichiers est alors affiché sur la sortie standard, un fichier après l'autre. La commande `cat` permet donc de concaténer des fichiers. Par exemple

```
% cat a a > b
```

range deux copies mises bout-à-bout du contenu du fichier `a` dans le fichier `b`. Comme souvent, si on ne donne pas d'argument à la commande `cat`, alors c'est l'entrée standard qui est lue. Ainsi `cat < a > b` copie toujours le fichier `a` dans le fichier `b`.

Pour lire un fichier à partir de son nom, on construit une nouvelle sorte de **Reader** : un **FileReader**. Cela revient à *ouvrir* un fichier en lecture, c'est-à-dire à demander au système d'exploitation de mobiliser les ressources nécessaires pour cette lecture. Si `name` est un nom de fichier, on procède ainsi.

```
Reader in = new FileReader (name) ;
```

Si le fichier de nom `name` n'existe pas, une exception est lancée (**FileNotFoundException** sous-classe de **IOException**).

Il est de bon ton de fermer les fichiers que l'on a ouvert, afin de rendre les ressources mobilisées lors de l'ouverture. En effet un programme donné ne peut pas posséder plus qu'un nombre fixé de fichiers ouvert en lecture. On ferme un **Reader** en appelant sa méthode `close()`. Il faut noter qu'une tentative de lecture dans une entrée fermée échoue et se solde par le lancement d'une exception spécifique. Voici la nouvelle classe **Cat2** .

```
import java.io.* ;

class Cat2 {
  static void cat(Reader in) throws IOException {
    for ( ; ; ) {
      int c = in.read ();
      if (c == -1) break;
      System.out.print ((char)c);
    }
  }

  public static void main(String [] arg) {
    if (arg.length == 0) {
      try {
```

```
        cat(new InputStreamReader(System.in)) ;
      } catch (IOException e) {
        System.err.println("Malaise : " + e.getMessage()) ;
        System.exit(2) ;
      }
    } else {
      for (int k = 0 ; k < arg.length ; k++) {
        try {
          Reader in = new FileReader(arg[k]) ;
          cat(in) ;
          in.close() ;
        } catch (IOException e) {
          System.err.println("Malaise : " + e.getMessage()) ;
        }
      }
    }
  }
}
```

Remarquons :

- La lecture d'un fichier est réalisée par une méthode `cat`, car cette lecture s'opère dans deux contextes différents. On note aussi l'intérêt du sous-typage : l'argument **Reader** `in` est ici un objet **InputStreamReader** ou **FileReader**, mais comme il n'est pas utile de le savoir dans la méthode `cat`, on en profite pour partager du code.
- Le second **try**. . . **catch** est dans la boucle **for** `(`**int** `k = 0 ;`. . . , et sa clause **catch** ne contient pas d'appel à **System**`.exit`. Il en résulte que le programme n'échoue pas en cas d'erreur de lecture d'un fichier donné par son nom — ou même de fichier inexistant, car **FileNotFoundException** est une sous-classe de **IOException**. Au lieu d'échouer, on passe au fichier suivant.
- Les messages d'erreur sont écrits dans la sortie d'erreur et non pas dans la sortie standard. Ainsi si le fichier `a` existe mais que le fichier `x` n'existe pas, la commande suivante a un comportement raisonnable.
  ```
  % java  Cat2 a x a > b
  Malaise : x (No such file or directory)
  ```
  Un message d'erreur est affiché dans la fenêtre, et `b` contient deux copies de `a`.

5.3 Encore un peu plus que le minimum : écrire dans un fichier

La sortie standard est souvent insuffisante, on peut vouloir écrire plus d'un fichier ou ne pas obliger l'utilisateur à faire une redirection. On se dit donc qu'il doit bien y avoir un moyen d'obtenir un flux de caractères en sortie connecté à un fichier dont on donne le nom. Le flux de sortie le plus simple est le **Writer**, qui possède une méthode `write(`**int** `c)` pour écrire un caractère. Comme pour les **Reader** on utilise en pratique des sous-classes de **Writer**. Il existe entre autres un **FileWriter** que nous utilisons pour écrire une version simple de la commande Unix `cp` , où `cp` $name_1$ $name_2$ copie le contenu du fichier $name_1$ dans le fichier $name_2$.

```
import java.io.* ;
class Cp {
  public static void main(String [] arg) {
    if (arg.length != 2) {
      System.err.println("Usage : java Cp name1 name2") ;
      System.exit(2) ;
    }
    String name1 = arg[0], name2 = arg[1] ;
```

```
    try {
      Reader in = new FileReader(name1) ;
      Writer out = new FileWriter(name2) ;
      for ( ; ; ) {
        int c = in.read() ;
        if (c == -1) break ;
        out.write(c) ;
      }
      out.close() ; in.close() ;
    } catch (IOException e) {
      System.err.println("Malaise : " + e.getMessage()) ;
      System.exit(2) ;
    }
  }
}
```

Il y a peu à dire, mais remarquons que nous prenons soin de fermer les flux que nous ouvrons. Cela semble une bonne habitude à prendre (voir aussi 5.6 pour tout savoir ou presque).

5.4 Toujours plus loin : entrées-sorties bufferisées

La documentation recommande l'emploi de flux *bufferisés* (*buffered* désolé pas de terme français adéquat) pour atteindre l'efficacité maximale « *top efficiency* ». L'idée des flux bufferisés, qui est très générale, est la suivante : l'écriture ou la lecture effective de caractères a un coût fixe important, qui est payé quelque soit le nombre de caractères effectivement transférés entre fichier et programme. Plus précisément, écrire ou lire n caractères coûte de l'ordre de $K_0 + K_1 \cdot n$, où K_0 est bien plus grand que K_1. Cela peut s'expliquer par divers phénomènes. Par exemple, un appel au système d'exploitation est relativement lent, et une opération d'entrée-sortie signifie un seul appel système, quelque soit le nombre de caractères impliqués. Ou encore, le coût des transferts entre disque et mémoire est largement indépendant du nombre de caractères transférés, jusqu'à une certaine taille, de par la nature même du dispositif physique « disque » qui lit et écrit des données par blocs de taille fixée.

Pour fixer les idées, prenons l'exemple de l'écriture. L'idée est alors de ne pas écrire effectivement chaque caractère en réaction à un appel `out.write(c)` mais à la place de le ranger dans une zone mémoire appelée *tampon* (*buffer*). Le tampon a une taille fixe, et les caractères du tampon sont effectivement transmis au système d'exploitation quand le tampon est plein. De cette façon le coût fixe K_0 est payé moins souvent et l'efficacité totale est améliorée. Les transferts vers les fichiers à travers un tampon mémoire présentent l'inconvénient que le caractère `c` peut ne pas se trouver dans le fichier[2] dès que l'on appelle `out.write(c)`. L'effet est particulièrement gênant à la fin du programme. Si le tampon n'est pas vidé (*flushed*), son contenu est perdu. En effet, le tampon est de la mémoire appartenant au programme et qui donc disparaît avec lui. Il en résulte généralement que la fin du flux ne se retrouve pas dans le fichier. Il faut donc vider le tampon avant de terminer le programme, ce que fait la méthode `close()` de fermeture des flux, avant de fermer effectivement le flux. On peut aussi vider le tampon plus directement en appelant méthode `flush()` des flux bufferisés. L'existence d'un retard entre ce qui est écrit par le programme dans le flux et ce qui est effectivement écrit dans le fichier donne donc une raison supplémentaire de fermer les fichiers.

Pour ce qui est d'un fichier ouvert en lecture, la technique du tampon s'applique également, avec les mêmes bénéfices en terme d'efficacité. Dans ce cas, les lectures (`read`) se font dans le tampon, qui est rempli à partir du fichier quand une demande de lecture trouve un tampon

[2] Ou plus exactement `c` n'est pas dans les tampons du système d'exploitation, en route vers le fichier.

vide. Il y a alors bien entendu une avance à la lecture mais ce décalage ne pose pas les mêmes problèmes que le retard à l'écriture.

Un flux bufferisé en écriture (resp. lecture) est un objet de la classe **BufferedWriter** (resp. **BufferedReader**), qui se construit simplement à partir d'un **Writer** (resp. **Reader**), et qui reste un **Writer** (resp. **Reader**). Voici une autre version **Cp2** de la commande cp écrite en Java, qui emploie les entrées-sorties bufferisées.

```
import java.io.* ;
class Cp2 {
  public static void main(String [] arg) {
    String name1 = arg[0], name2 = arg[1] ;
    try {
      Reader in = new BufferedReader (new FileReader(name1)) ;
      Writer out = new BufferedWriter (new FileWriter (name2)) ;
      for ( ; ; ) {
        int c = in.read() ;
        if (c == -1) break ;
        out.write(c) ;
      }
      out.close() ; in.close() ;
    } catch (IOException e) {
      System.err.println("Malaise : " + e.getMessage()) ;
      System.exit(2) ;
    }
  }
}
```

Une mesure rapide des temps d'exécution montre que le programme **Cp2** est à peu près trois fois plus rapide que **Cp**.

Les flux bufferisés **BufferedWriter** et **BufferedReader** offrent aussi une vue des fichier texte comme étant composés de lignes. Il existe une méthode `Newline` pour écrire une fin le ligne, et une méthode `readLine` pour lire une ligne. On utilise souvent **BufferedReader** pour cette fonctionnalité de lecture ligne à ligne. Il faut aussi noter que les lignes sont définies indépendamment de leur réalisation par les divers systèmes d'exploitation (voir 3.2.3).

5.5 Entrées-sorties formatées

5.5.1 Sorties formatées

Nous savons désormais écrire dans un fichier de nom `name` de façon efficace en construisant un **FileWriter**, puis un **BufferedWriter**.

```
Writer out = new BufferedWriter (new FileWriter name) ;
```

Toutefois, écrire caractère par caractère (ou même par chaîne avec `write(String str)`), n'est pas toujours très pratique. La méthode `print` de `System.out` est bien plus commode. Malheureusement, les **PrintStream** de la bibliothèque sont très inefficaces. Par exemple, la copie du fichier `a` dans le fichier `b` par `java Cat < a > b` est environ vingt-cinq fois plus lente que `java Cp2 a b`!

Mais la méthode `print` est bien pratique, voici par exemple une méthode simple qui affiche les nombres premiers jusqu'à n directement dans `System.out`.

```
static void sieve(int n) {
  boolean [] t = new boolean [n+1] ;
  int p = 2 ;
  for ( ; ; ) {
```

```
    System.out.println(p) ; // Afficher p premier
    /* Identifier les multiples de p */
    for (int k = p+p ; k <= n ; k += p) t[k] = true ;
    /* Chercher le prochain p premier */
    do {
      p++ ; if (p > n) return ;
    } while (t[p]) ;
  }
}
```

L'application du crible d'Eratosthène est naïve, mais le programme est inefficace d'abord à cause de ses affichages. On s'en rend compte en modifiant `sieve` pour utiliser un **PrintWriter**. Les objets de la classe **PrintWriter** sont des **Writer** (flux de caractères) qui offrent les sorties formatées, c'est-à-dire qu'ils possèdent une méthode `print` (et `prinln`) surchargée qui allège un peu la programmation. Par ailleurs, les **PrintWriter** sont bufferisés par défaut.

```
  static void sieve(int n, PrintWriter out) throws IOException {
    boolean [] t = new boolean [n+1] ;
    int p = 2 ;
    for ( ; ; ) {
      out.println(p) ;
      for (int k = p+p ; k <= n ; k += p) t[k] = true ;
      do {
        p++ ; if (p > n) return ;
      } while (t[p]) ;
    }
  }

  public static void main(String [] arg) {
      :
     try {
      PrintWriter out = new PrintWriter ("primes.txt") ;
      sieve(n, out) ;
      out.close() ;
    } catch (IOException e) {
      System.err.println("Malaise: " + e.getMessage()) ;
      System.exit(2) ;
    }
  }
```

La nouvelle méthode `sieve` écrit dans le **PrintWriter** `out` dont l'initialisation est rendue pénible par les exceptions possibles. Noter aussi que la sortie `out` est vidée (et même fermée) par la méthode `main` qui l'a créée. Il y a plusieurs constructeurs des **PrintWriter**, qui prennent divers arguments. Par exemple, si l'on avait voulu écrire dans la sortie standard, et non pas dans le fichier `primes.txt`, on aurait très bien pu « *emballer* » la sortie standard dans un **PrintWriter**, par **new PrintWriter(System**`.out`). Pour ce qui est du temps d'exécution des essais rapides montrent que le nouveau `sieve` (avec **PrintWriter**) peut être jusqu'à dix fois plus rapide que l'ancien (avec **PrintStream**). Nous insistons donc : **System**`.out` est vraiment très inefficace, il faut renoncer à son emploi dès qu'il y a beaucoup de sorties.

5.5.2 Entrées formatées

Les **Reader** permettent de lire caractère par caractère, les **BufferedReader** permettent de lire ligne par ligne. C'est déjà pas mal, mais c'est parfois insuffisant. Par exemple, on peut imaginer vouloir relire le fichier `primes.txt` de la section précédente. Ce que l'on veut alors c'est lire une séquence d'entiers **int**. Pour ce faire nous pourrions reconstituer nous même les **int** à partir de la séquence de leurs chiffres, mais ça ne serait pas très moderne.

Heureusement, il existe des objets qui savent faire ce style de lecture pour nous : ceux de la classe **Scanner** du package `java.util`. Les objets **Scanner** sont des flux de *lexèmes* (*tokens*). Les lexèmes sont simplement à un langage informatique ce que les mots sont à un langage dit naturel. Par défaut les lexèmes reconnus par un **Scanner** sont les mots de Java — un entier, un identificateur, une chaîne, etc.

Un **Scanner** possède une méthode `next()` qui renvoie le lexème suivant du flux d'entrée (comme un **String**) et une méthode `hasNext()` pour savoir si il existe un lexème suivant, ou si le flux est terminé. Il possède aussi des méthodes spécialisées pour lire des lexèmes particuliers ou savoir si le lexème suivant existe et est un lexème particulier (par exemple `nextInt()` et `hasNextInt()` pour un **int**). On peut donc utiliser `primes.txt` pour décomposer un **int** en facteurs premiers ainsi (même si ce n'est pas une très bonne idée algorithmiquement parlant).

```java
import java.util.* ;
import java.io.* ;

class Factor {
  private static PrintWriter out = new PrintWriter (System.out) ;

  private static void factor (int n, String filename) {
    try {
      Scanner scan = new Scanner (new FileReader (filename)) ;
      out.print(n + ":") ;
      while (n > 1) {
        if (!scan.hasNextInt())
          throw new IOException ("Format of file " + filename) ;
        int p = scan.nextInt() ;
        while (n % p == 0) {
          out.print(" " + p) ;
          n /= p ; // Pour n = n / p ;
        }
      }
      out.println() ; out.flush() ; scan.close() ;
    } catch (IOException e) {
      System.err.println("Malaise : " + e.getMessage()) ;
      System.exit(2) ;
    }

  }

  public static void main(String arg []) {
    for (int k = 0 ; k < arg.length ; k++)
      factor(Integer.parseInt(arg[k]), "primes.txt") ;
  }
}
```

On note que le **Scanner** est construit à partir d'un **Reader**. Il y a bien entendu d'autres constructeurs. On note aussi que l'entrée `scan` est fermée explicitement par `scan.close()`, dès qu'elle n'est plus utile ; tandis que la sortie n'est que vidée par `out.flush()`. En effet, il ne serait pas adéquat de fermer la sortie `out` qui sert plusieurs fois, mais il faut s'assurer que les sorties du programme sont bien envoyées au système d'exploitation.

5.6 Complément : tout ou presque sur les fichiers texte

Si Java et Unix sont bien d'accord sur ce qu'est en gros un fichier texte (un fichier qu'un humain peut lire au moins en principe), ils ne sont plus d'accord sur ce que sont les éléments d'un tel fichier. Pour Java, un fichier de texte est un flux de **char** (16 bits), tandis que pour Unix c'est un flux d'octets (8 bits, un **byte** pour Java). Le passage de l'un à l'autre demande d'appliquer un *encodage*.

Un exemple simple d'encodage est par exemple `ISO-8859-1`, l'encodage que java emploie par défaut sur les machines de l'école. C'est un encodage qui fonctionne pour presque toutes les langues européennes (mais pas pour le symbole €). C'est un encodage simple sur 8 bits, qui permet d'exprimer 256 caractères seulement parmi les 2^{16} d'Unicode. Pour voir les caractères définis en `ISO-8859-1`, vous pouvez essayer `man latin1` sur une machine de l'école. L'encodage `ISO-8859-1` est techniquement le plus simple possible : les codes des caractères sont les mêmes à la fois en `ISO-8859-1` et en Unicode. Il n'est donc tout simplement pas possible d'exprimer les **char** dont les codes sont supérieurs à 256 en `ISO-8859-1` (dont justement €, dont la valeur Unicode hexadécimale est `0x20AC`, notée `U+20AC`). Il existe d'autres encodages 8 bits, dont `ISO-8859-15` qui entre autres établit justement la correspondance entre le caractère Unicode `U+20AC` et le **byte** `0xA4`, au dépend du caractère Unicode `U+00A4` (¤) qui n'est plus exprimable. Il existe bien entendu des encodages qui permettent d'exprimer tout Unicode, mais alors un caractère Unicode peut s'exprimer comme plusieurs octets. Un encodage multi-octet répandu est `UTF-8`, où les caractères Unicode sont représentés par un nombre variable d'octets selon un système un peu compliqué que nous ne décrirons pas (voir `http ://fr.wikipedia.org/wiki/UTF-8` qui est raisonnablement clair).

Revenons aux flux de Java. Pour fixer les idées nous considérons d'abord les flux en lecture. Un flux d'octets est un **InputStream**. La classe **InputStream** fonctionne sur le même principe que la classe **Reader** : c'est une sur-classe des divers flux de **byte** qui peuvent être construits. Par exemple on ouvre un flux d'octets sur un fichier `name` en créant un objet de la classe **FileInputStream**.

```
InputStream inBytes = new FileInputStream (name) ;
```

Pour lire un flux d'octets comme un flux de **char**, on fabrique un **InputStreamReader**.

```
Reader inChars = new InputStreamReader (inBytes) ;
```

L'encodage est ici implicite, c'est l'encodage par défaut. On peut aussi expliciter l'encodage en donnant son nom comme second argument au constructeur.

```
Reader inChars = new InputStreamReader (inBytes, "UTF-8") ;
```

Et voilà nous disposons maintenant d'un **Reader** sur un fichier dont la suite d'octets définit une suite de caractères Unicode encodés en `UTF-8`. Ici, comme `UTF-8` est un encodage multi-octets, la lecture d'un **char** dans `inChars` implique de lire un ou plusieurs **byte** dans le flux `inBytes` sous-jacent. Les flux en écriture suivent un schéma similaire, il y a des flux de **byte** (**OutputStream**) et des flux de **char** (**Writer**), avec une classe pour faire le pont entre les deux (**OutputStreamWriter**). Par exemple, voici comment fabriquer un **Writer** connecté à la sortie standard et qui écrit de l'`UTF-8` sur la console :

```
// System.out (un PrintStream) est aussi un OutputStream
Writer outChar = new OutputStreamWriter (System.out, "UTF-8") ;
```

Notons qu'une fois obtenu un **Reader** ou un **Writer** nous pouvons fabriquer ce dont nous avons besoin, par exemple un **Scanner** ou un **PrintWriter** etc., à l'aide des constructeurs « naturels » de ces classes, qui prennent un **Reader** ou un **Writer** en argument. Les diverses classes de flux possèdent parfois des constructeurs qui semblent permettre de court-circuiter le passage par un **InputStreamReader** ou un **OutputStreamWriter** ; mais ce n'est qu'une apparence, il y aura toujours décodage et encodage. Par exemple, **new PrintWriter** (**String** `name`) ouvre directement le fichier `name`, mais à quelques optimisations internes toujours possibles près, employer ce constructeur synthétique revient à :

new PrintWriter (**new OutputStreamWriter** (**new FileOutputStream** (`name`))) ;

Finalement, en théorie nous savons lire et écrire tout Unicode. En pratique, il faut encore pouvoir entrer ces caractères au clavier et les visualiser dans une fenêtre, mais c'est une autre histoire qui ne regarde plus Java.

Toutes ces histoires d'encodage et de décodage de **char** en **byte** font qu'écrire un **char** dans un **Writer** ou lire un **char** dans un **Reader** ne sont jamais des opérations simples.[3] Comme on a tendance a tout simplifier on a parfois des surprises : par exemple, les **FileWriter** (voir 5.3) possèdent en fait déjà un tampon. On se rend vite compte de l'existence de ce tampon si on oublie de fermer un **FileWriter**. Si on en croit la documentation :

> Each invocation of a write() method causes the encoding converter to be invoked on the given character(s). The resulting bytes are accumulated in a buffer before being written to the underlying output stream.

Il s'agit donc d'un tampon de **byte** dans lequel le **FileWriter** stocke les octets résultants de l'encodage qu'il effectue. On peut alors se demander d'où provient le gain d'efficacité constaté en emballant un **FileWriter** dans un **BufferedWriter** (voir 5.4), puisque le coût irréductible de la véritable sortie est déjà amorti par un tampon. Et bien, il se trouve que le coût de l'application de l'encodage des **char** vers les **byte** suit lui-aussi le modèle d'un coût constant important relativement au coût proportionnel au nombre de caractères encodés. Le tampon (de **char**) introduit en amont du **FileWriter** par **new BufferedWriter** (**new FileWriter** (`name2`)) a alors pour fonction d'amortir ce coût irréductible de l'encodage.

6 Quelques classes de bibliothèque

En général une bibliothèque (*library*) est un regroupement de code suffisamment structuré et documenté pour que d'autres programmeurs puissent s'en servir. La bibliothèque associée à un langage de programmation est diffusée avec le compilateur, système d'exécution etc. La majeure partie de des programmes disponibles dans la bibliothèque d'un langage est normalement écrite dans ce langage lui-même.

La première source de documentation des classes de la bibliothèque de Java est bien entendu la documentation en ligne[4] de cette dernière, ou comme on dit la *Application Programmer Interface Specification*. Nous donnons donc des extraits de cette documentation assortis de quelques commentaires personnels. Cette section entend surtout vous aider à prendre la bonne habitude de consulter la documentation du langage que vous utilisez. En particulier, la version web du polycopié comporte des liens vers la documentation en ligne.

On objectera que la bibliothèque est énorme. Mais en pratique on s'y retrouve assez vite :

- La documentation en ligne est organisée de façon systématique, La page d'accueil est une liste de liens vers les pages des packages, la page d'un package est une liste de liens vers les pages des classes, qui contiennent (au début) une liste de liens vers les champs, méthodes etc.
- Les classes que nous utilisons appartiennent à trois packages seulement.
- Les descriptions sont parfois un peu cryptiques (car elles sont complètes), mais on arrive vite à en extraire l'essentiel.

[3] Encodage et décodage font aussi qu'il ne faut pas écrire **Cp** et **Cat** comme nous l'avons fait (avec des flux de **char**). C'est inutilement inefficace et même dangereux certains décodages pouvant parfois échouer, il aurait mieux valu employer les flux de **byte**.

[4] `http://java.sun.com/j2se/1.5.0/docs/api`

Il est vain d'objecter que la documentation est en anglais. C'est comme ça.

6.1 Package `java.lang`, bibliothèque « standard »

Ce package regroupe les fonctionnalités les plus basiques de la la bibliothèque. Il est importé par défaut (pas besoin de **import** `java.lang.*`).

6.1.1 Classes des scalaires

À chaque type scalaire (**int**, **char** etc.), correspond une classe (**Integer**, **Character** etc.) qui permet surtout de transformer les scalaires en objets. Une transformation que le compilateur sait automatiser (voir II.2.3).

Prenons l'exemple de la classe **Integer** le pendant objet du type scalaire **int**. Deux méthodes permettent le passage d'un scalaire à un objet et réciproquement.

- La méthode `valueOf` transforme le scalaire en objet.

```
public static Integer valueOf(int i)
```

- La méthode réciproque `intValue` extrait le **int** caché dans un objet **Integer**.

```
public int intValue()
```

La classe **Integer** a bien d'autres méthodes, comme par exemple

- La méthode parseInt permet de « lire » un entier.

```
public static int parseInt(String s) throws NumberFormatException
```

Si la chaîne `s` contient la représentation décimale d'un entier, alors `parseInt` renvoie cet entier (comme un **int**). Si la chaîne `s` ne représente pas un **int**, alors l'exception **NumberFormatException** est lancée.

6.1.2 Un peu de maths

Les fonctions mathématiques usuelles sont définies dans la classe **Math**. Il n'y a pas d'objets **Math**. Cette classe ne sert qu'à la structuration. Elle offre des méthodes (statiques).

- La valeur absolue abs, disponible en quatre versions.

```
public static int abs(int a)
public static long abs(long a)
public static float abs(float a)
public static double abs(double a)
```

- Les fonctions maximum et minimum, max et min, également disponibles en quatre versions.

```
public static int max(int a, int b)
public static int min(int a, int b)
⋮
```

- Les divers arrondis des **double**, par défaut floor, par excès ceil et au plus proche round.

```
public static double floor(double a)
public static double ceil(double a)
public static long round(double a)
```

On note que `floor` et `ceil` renvoient un **double**. Tandis que `round` renvoie un **long**. Les méthodes existent aussi pour les **float** et `round(`**float** `a)` renvoie un **int**. Cela peut sembler logique si on considère que les valeurs **double** et **long** d'une part, et **float** et **int** d'autre part occupent le même nombres de bits (64 et 32). Mais en fait ça ne règle pas vraiment le problème des flottants trop grands pour être arrondis vers un entier (voir la documentation).

- Et bien d'autres méthodes, comme la racine carrée sqrt, l'exponentielle exp, le logarithme naturel log, etc.

6.1.3 Les chaînes

Les chaînes sont des objets de la classe **String**. Une chaîne est tout simplement une séquence finie de caractères. En Java les chaînes sont non-mutables : on ne peut pas changer les caractères d'une chaîne. Les chaînes sont des objets normaux, à un peu de syntaxe spécialisée près. Les constantes chaînes s'écrivent entre doubles quotes, par exemple *"coucou"*. Si on veut mettre un double quote « `"` » dans une chaîne, il fait le citer en le précédant d'une barre oblique inverse (*backslash*). **System**`.err.print(`*"Je suis un double quote : \""*`)` affiche `Je suis un double quote : "` sur la console. En outre, la concaténation de deux chaînes s_1 et s_2, s'écrit avec l'opérateur `+` comme s_1+s_2.

Les autres opérations classiques sur les chaînes sont des méthodes normales.

- La taille ou longueur d'une chaîne s, c'est-à-dire le nombre de caractères qu'elle contient s'obtient par s.length()

  public int `length()`

- Le caractère d'indice `index` s'extrait par la méthode `charAt`.

  public char `charAt(`**int** `index)`

  Comme pour les tableaux, les indices commencent à zéro.
- On extrait une sous-chaîne par la méthode `substring`.

  public String `substring(`**int** `beginIndex,` **int** `endIndex)`

  `beginIndex` est l'indice du premier caractère de la sous-chaîne, et `endIndex` est l'indice du caractère qui suit immédiatement la sous-chaîne. La longueur de la sous-chaîne extraite est donc `endIndex-beginIndex`. Si le couple d'indices ne désigne pas une sous-chaîne l'exception **IndexOutOfBoundsException** est lancée. Les règles de définition du couple d'indices désignant une sous-chaîne valide (voir la documentation) conduisent à la particularité bien sympathique que s`.substring(`s`.length(),` s`.length())` renvoie la chaîne vide *""*.
- La méthode `equals` est l'égalité structurelle des chaînes, celle qui regarde le contenu des chaînes.

  public boolean `equals(`**Object** `anObject)`

  Il faut remarquer que l'argument de la méthode est un **Object**. Il faut juste le savoir, c'est tout. La classe **String** étant une sous-classe de **Object** (voir 2.3), la méthode s'applique en particulier au seul cas utile d'un argument de classe **String**.
- La méthode `compareTo` permet de comparer les chaînes.

  public int `compareTo(`**String** `anotherString)`

  Compare la chaîne avec une autre selon l'ordre lexicographique (l'ordre du dictionnaire). L'appel « s_1`.compareTo(`s_2`)` » renvoie un entier r avec :
  - $r < 0$, si s_1 est strictement inférieure à s_2.
  - $r = 0$, si les deux chaînes sont les mêmes.
  - $r > 0$, si s_1 est strictement plus grande t s_2.

  L'ordre sur les chaînes est induit par l'ordre des codes des caractères, ce qui fait que l'on obtient l'ordre du dictionnaire *stricto-sensu* seulement pour les mots sans accents.
- Il y a beaucoup d'autres méthodes, par exemple `toLowerCase` et `toUpperCase` pour minusculer et majusculer.

public String `toLowerCase()`
public String `toUpperCase()`

Cette fois-ci les caractères accentués sont respectés.

6.1.4 Les fabricants de chaînes

On ne peut pas changer une chaîne, par exemple changer un caractère individuellement. Or, on veut souvent construire une chaîne petit à petit, on peut le faire en utilisant la concaténation mais il y a alors un prix à payer, car une nouvelle chaîne est créée à chaque concaténation. En fait, lorsque l'on construit une chaîne de gauche vers la droite (pour affichage d'un tableau par ex.) on a besoin d'un autre type de structure de données : le **StringBuilder** (ou **StringBuffer** essentiellement équivalent). Un objet **StringBuilder** contient un tampon de caractère interne, qui peut changer de taille et dont les éléments peuvent être modifiés. Les membres les plus utiles de cette classe sont :

- Le constructeur sans argument.

  public StringBuilder `()`

  Crée un nouveau fabricant dont le tampon a la capacité initiale par défaut (16 caractères).
- Les méthodes `append` surchargées. Il y en a treize, six pour les scalaires, et les autres pour divers objets.

  public StringBuilder `append(`**int** `i)`
  public StringBuilder `append(`**String** `str)`
  public StringBuilder `append(`**Object** `obj)`

  Toutes ces méthodes ajoutent la représentation sous forme de chaîne de leur argument à la fin du tampon interne, dont la capacité est augmenté silencieusement si nécessaire. Dans le cas de l'argument objet quelconque (le dernier), c'est la chaîne obtenue par `obj.toString()` qui est ajoutée, ou *"null"* si `obj` est **null**. Les autres méthodes sont là, soit pour des raisons d'efficacité (cas de **String**), soit pour les tableaux dont le `toString()` est inutilisable, soit parce que l'argument est de type scalaire (cas de **int**). Toutes les méthodes `append` renvoient l'objet **StringBuilder** dont on appelle la méthode, ce qui est absurde, cet objet restant le même. Je crois qu'il en est ainsi afin d'autoriser par exemple l'écriture `sb.append(i).append(':')` ; au lieu de `sb.append(i) ; sb.append(':') ;` (où `i` est une variable par exemple de type **int**).
- Et bien évidemment, une méthode toString redéfinie.

  public String `toString()`

  Copie le contenu du tampon interne dans une nouvelle chaîne. Contrairement aux tampons de fichier, le tampon n'est pas vidé. Pour vider le tampon, on peut avoir recours à la méthode `setLength` avec l'argument zéro.

On trouvera un exemple d'utilisation classique d'un objet **StringBuilder** en I.2.1 : une création, des tas de `append(...)`, et un `toString()` final.

6.1.5 La classe System

Cette classe **System** est un peu fourre-tout. Elle posssède trois champs et quelques méthodes assez utiles.

- Les trois flux standards.

  public static final InputStream `in`
  public static final PrintStream `out`
  public static final PrintStream `err`

Ces trois flux sont des flux d'octets (voir la section sur les entrées-sorties 5). Notez que les champs sont déclarés **final** , on ne peut donc pas les changer (bizarrement il existe quand même des méthode pour changer les trois flux standard). Les deux flux de sortie offrent des sorties formatées par une méthode `print` bien pratique, car elle est surchargée et fonctionne pour toutes les valeurs.
La sortie `out` est la sortie normale du programme, tandis que la sortie `err` est la sortie d'erreur. En fonctionnement normal ces deux flux semblent confondus car aboutissant tous deux dans la même fenêtre. Mais on peut rediriger la sortie `out`, par exemple vers un fichier (>*fichier*). Les messages d'erreurs s'affichent alors au lieu de se mélanger avec la sortie normale du programme dans *fichier*.

- Une méthode pour arrêter imédiatement le programme.

  ```
  public static void exit(int status)
  ```

  L'entier `status` est le code de retour qui vaut conventionellement zéro dans le cas d'une exécution normale, et autre chose pour une exécution qui s'est mal passée, En Unix, le code est accessible juste après l'exécution comme `$status` ou `$?`, selon les shells.
- Deux méthodes donnent accès à un temps absolu. La seconce permet de mesurer le temps d'exécution par une soustractions.

  ```
  public static long currentTimeMillis()
  public static long nanoTime()
  ```

  La méthode `currentTimeMillis` renvoie les milisecondes écoulées depuis le 1^er janvier 1970. Il s'agit donc du temps ordinaire. La méthode `nanoTime()` renvoie les nanosecondes (10^{-9}) depuis une date arbitraire. Il s'agit du temps passé dans le code du programme, ce qui est différent du temps ordinaire — surtout dans un système d'exploitation multi-tâches comme Unix.

6.2 Package `java.io`, entrées-sorties

Ce package regroupe s'attaque principalement à la communication avec le système de fichiers de la machine. Nous ne passons en revue ici que les flux de caractères, qui correspondent aux fichiers texte (voir 5).

6.2.1 Classe des lecteurs

Les objets de la classe **Reader** sont des flux de caractères ouverts en lecture. Un tel flux est simplement une séquence de caractères que l'on lit les un après les autres, et qui finit par se terminer.

- La méthode la plus significative est donc celle qui permet de lire le caractère suivant du flux.

  ```
  public int read() throws IOException
  ```

  Cette méthode renvoie un **int** qui est le caractère en attente dans le flux d'entrée, ou -1, si le flux est terminé. On notera que `read` renvoie un **int** et non pas un **char** précisément pour pouvoir identifier la fin du flux facilement. En outre et comme annoncé par sa déclaration elle peut lever l'exception **IOException** en cas d'erreur (possible dès que l'on s'intéresse au fichiers).
- Quand la fin d'un flux est atteinte, il faut fermer le flux.

  ```
  public void close() throws IOException
  ```

  Cette opération libère les ressources mobilisées lors de la création d'un flux d'entrée (numéros divers liées au système de fichiers, tampons internes). Si on ne ferme pas les flux dont on ne sert plus, on risque de ne pas pouvoir en ouvrir d'autres, en raison de l'épuisement de ces ressources.

La classe **Reader** sert seulement à décrire une fonctionnalité et il n'existe pas à proprement parler d'objets de cette classe.

En revanche, il existe diverses sous-classes (voir 2.3) de **Reader** qui permettent de construire concrètement des objets, qui peuvent se faire passer pour des **Reader**. Il s'agit par entre autres des classes **StringReader** (lecteur sur une chaîne), **InputStreamReader** (lecteur sur un flux d'octets) et **FileReader** (lecteur sur un fichier). On peut donc par exemple obtenir un **Reader** ouvert sur un fichier `/usr/share/dict/french` par

```
Reader in = new FileReader ("/usr/share/dict/french") ;
```

et un **Reader** sur l'entrée standard par

```
Reader in = new InputStreamReader (System.in) ;
```

6.2.2 Lecteur avec tampon

Un objet **BufferedReader** est d'abord un **Reader** qui groupe les appels au flux sous-jacent et stocke les caractères lus dans un tampon, afin d'améliorer les performances (voir 5.4). Mais un **BufferedReader** offre quelques fonctionnalités en plus de celles des **Reader**.

- Le constructeur le plus simple prend un **Reader** en argument.

  ```
  public BufferedReader(Reader in)
  ```

  Construit un flux d'entrée bufferisé à partir d'un lecteur quelconque.
- La classe **BufferedReader** est une sous-classe de la classe **Reader** de sorte qu'un objet **BufferedReader** possède une méthode `read`.

  ```
  public int read() throws IOException
  ```

  Cette méthode n'est pas héritée mais redéfinie, afin de gérer le tampon interne.
- La lecture d'une ligne se fait par une nouvelle méthode, que les **Reader** ne possèdent pas.

  ```
  public String readLine() throws IOException
  ```

  Renvoie une ligne de texte ou **null** en fin de flux. La méthode fonctionne pour les trois conventions de fin de ligne (`'\n'`, `'\r'` ou `'\r'` suivi de `'\n'`) et le ou les caractères « fin de ligne » ne sont pas renvoyés. La méthode fonctionne encore pour la dernière ligne d'un flux qui ne serait pas terminée par une fin de ligne explicite.

6.3 Package `java.util` : trucs utiles

6.3.1 Les sources pseudo-aléatoires

Une source pseudo-aléatoire est une suite de nombres qui ont l'air d'être tirés au hasard. Ce qui veut dire que les nombres de la suite obéissent à des critères statistiques censés exprimer le hasard (et en particulier l'uniformité, mais aussi un aspect chaotique pas évident à spécifier précisément). Une fois la source construite (un objet de la classe **Random**), diverses méthodes permettent d'obtenir divers scalaires (**int**, **double**, etc.) que l'on peut considérer comme tirés au hasard. On a besoin de tels nombres « tirés au hasard » par exemple pour réaliser des simulations (voir II.1.1) ou produire des exemples destiner à tester des programmes.

- Les constructeurs.

  ```
  public Random()
  public Random(long seed)
  ```

La version avec argument permet de donner une semence de la source pseudo-aléatoire, deux sources qui ont la même semence se comportant à l'identique. La version sans argument laisse la bibliothèque choisir une semence comme elle l'entend. La semence choisie change à chaque appel du constructeur et on suppose qu'elle est choisie de façon la plus arbitraire et non-reproductible possible (par exemple à partir de l'heure qu'il est).

- Les nombres pseudo-aléatoires sont produits par diverses méthodes « `next` ». Distinguons particulièrement,
  - Un entier entre zéro (inclus) et `n` (exclu).

    ```
    public int nextInt(int n)
    ```

    Cette méthode est pratique pour tirer un entier au pseudo-hasard sur un intervalle `[a...b[`, par `a + rand.nextInt(b-a)`.
  - Un flottant entre `0.0` (inclus) et `1.0` (exclu).

    ```
    public float nextFloat()
    public double nextDouble()
    ```

    Ces méthodes sont pratiques pour décider selon une probabilité quelconque. Du style `rand.nextDouble() < 0.75` vaut **true** avec une probabilité de 75 %.

7 Pièges et astuces

Cette section regroupe des astuces de programmation ou corrige des erreurs fréquemment rencontrées.

7.1 Sur les caractères

Les codes des caractères de `'0'` à `'9'` se suivent, de sorte que les deux trucs suivants s'appliquent. Pour savoir si un caractère `c` est un chiffre (arabe) :

```
'0' <= c && c <= '9'
```

Il existe aussi une méthode **Character**`.isDigit(`**char** `c)` qui tient compte de tous les chiffres possibles (en arabe c'est-à-dire indiens, etc.). Pour récupérer la valeur d'un chiffre :

```
int i = c - '0' ;
```

Ici encore il y a des méthodes statiques qui donnent les valeurs de chiffres dans la classe **Character**.

7.2 Opérateurs

Java possède un opérateur ternaire (à trois arguments) *l'expression conditionnelle* e_c `?` e_t `:` e_f. L'expression e_c est de type **boolean**, tandis que le types de e_t et e_f sont identiques. L'expression conditionnelle vaut e_t si e_c vaut **true**, et e_f si e_c vaut **false** . De sorte que l'on peut afficher un booléen ainsi :

```
// b est un boolean, on veut afficher '+' si b est vrai et '-' autrement.
   System.out.println(b ? '+' : '-') ;
```

7.3 Connecteurs « paresseux »

Les opérateurs `||` et `&&` sont respectivement la disjonction (ou logique) et la conjonction (et logique), ils sont de la variété dite séquentielle (gauche-droite) ou parfois paresseuse : si évaluer la première condition suffit pour déterminer le résultat, alors la seconde condition n'est pas évaluée.

Plus précisément, si dans `e`$_1$ `||` `e`$_2$ (resp. `e`$_1$ `&&` `e`$_2$) `e`$_1$ vaut **true** (resp. **false**) , alors `e`$_2$ n'est pas évaluée, et la disjonction (resp. conjonction) vaut **true** (resp. false). C'est un idiome de profiter de ce comportement pour écrire des conditions concises. Par exemple, pour savoir si la liste `xs` contient au moins deux éléments, on peut écrire

```
... xs != null && xs.next != null ...
```

Cette expression n'échoue jamais, cas si `xs` vaut **null** alors l'accès `xs.next` n'est pas tenté.

On remarque que les connecteurs `&&` et `||` peuvent s'exprimer à l'aide d'un seul opérateur : l'opérateur ternaire e_1 `?` e_2 `:` e_3. La conjonction `e`$_1$ `&&` `e`$_2$ s'exprime comme e_1 `?` e_2 `:` **false** et la disjonction `e`$_1$ `||` `e`$_2$ comme e_1 `?` **true** `:` e_2

Bibliographie

[1] J. L. Bentley et M. D. McIlroy, Engineering a Sort Function, *Software - Practice and Experience* **23**,11 (1993), 1249–1265.

[2] T. H. Cormen, C. E. Leiserson et R. L. Rivest, *Introduction à l'algorithmique*, Dunod, 1994.

[3] J. E. F. Friedl, *Maîtrise des expressions régulières*, O'Reilly, 2001.

[4] D. E. Knuth, *Fundamental Algorithms. The Art of Computer Programming, vol. 1*, Addison Wesley, 1968.

[5] D. E. Knuth, *Seminumerical Algorithms. The Art of Computer Programming, vol. 2*, Addison Wesley, 1969.

[6] D. E. Knuth, *Sorting and Searching. The Art of Computer Programming, vol. 3*, Addison Wesley, 1973.

[7] R. Sedgewick, *Algorithms, (2nd edition)*, Addison-Wesley, 1988.

[8] R. Uzgalis, Hashing Concepts and the Java Programming Language, Rap. Tech., University of Auckland, New Zealand, 1996.

Index