Ministère de l'Education Nationale,
de l'Enseignement Supérieur, de la Formation
des Cadres et de la Recherche Scientifique

Fondation ACADEMIA

Les Journées Informatiques des Classes Préparatoires aux Grandes Ecoles
Rabat, 29 Octobre-2 Novembre 2009

Cours d'informatique

en classes préparatoires aux grandes écoles marocaines

Version provisoire, octobre 2009

Denis Monasse
Ancien élève de l'Ecole normale supérieure
Agrégé de Mathématiques
Professeur en classe de MP*
au lycée Louis Le Grand, Paris

Remarque: Il s'agit là d'une version provisoire de ce cours. Faute de temps, beaucoup de programmes n'ont pu être sérieusement vérifiés.

L'auteur s'excuse par avance de toutes les fautes qui peuvent y figurer.

Table des matières

Chapitre 1

Récursion et itération

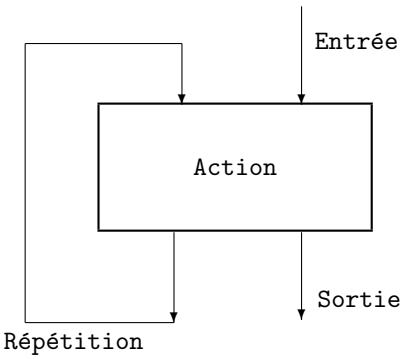
1.1 Itération

1.1.1 Boucles

L'informatique : l'exécution de tâches répétitives, voilà une opinion couramment admise ; même si cette vision est un peu simpliste, elle contient certainement une part de vérité. Il est certain que les tâches répétitives sont celles que les machines sont les plus aptes à accomplir, tout en étant les moins gratifiantes pour l'homme.

Il arrive donc souvent qu'on écrive un programme juste pour répéter un certain nombre de fois une même tâche : calculer 500 valeurs d'une fonction pour tracer un graphe, parcourir un dictionnaire de 100 000 mots non triés pour rechercher le mot *anachorète*, etc.

La structure fondamentale de répétition des langages de programmation est la boucle. Nous pouvons symboliser une boucle sous la forme suivante



Une boucle possède donc une entrée, un corps qui accomplit une action, un circuit de répétition qui répète l'exécution du corps de la boucle, et une sortie.

Nous discuterons par la suite deux types de boucles

- les boucles indexées où le corps de la boucle doit être répété un nombre (fini) de fois, ce nombre étant connu dès l'entrée dans la boucle
- les boucles conditionnelles où le corps de la boucle doit être exécuté tant qu'une condition est (ou n'est pas) vérifiée

1.1.2 Boucles indexées

Il s'agit d'actions répétitives qui doivent être effectuées un nombre fini de fois, variable, mais fixé dès l'entrée dans la boucle. En général, le corps de la boucle dépend d'un entier `i` qui varie entre une valeur initiale `iDeb` et une valeur finale `iFin`, en augmentant de 1 à chaque nouvelle exécution de la boucle ou au contraire en diminuant de 1 à chaque exécution de la boucle.

Pour cela, la plupart des langages de programmation disposent de deux instructions du type

`for` *i* `from` *iDeb* `to` *iFin* `do` *corps de la boucle* `done`

lorsque l'indice i augmente de 1, et

`for` *i* `from` *iDeb* `downto` *iFin* `do` *corps de la boucle* `done`

lorsque l'indice i diminue de 1. Dans certain langages (comme Java ou Pascal), le mot clé `from` est remplacé par une affectation (ou une liaison) du type

`for` *i*=*iDeb* `(down)to` *iFin* `do` *corps de la boucle* `done`

Dans une boucle croissante, le corps de la boucle n'est jamais exécuté lorsque *iFin*<*iDeb* ; par contre dans une boucle décroissante, le corps de la boucle n'est jamais exécuté lorsque *iDeb*<*iFin*

Les boucles indexées sont particulièrement adaptées aux objets dont la longueur est connue à l'avance comme les tableaux. Nous allons illustrer leur utilisation à travers trois exemples très simples en Java, sans aucune exigence d'efficacité maximale : la recherche d'un élément dans un tableau, la somme des éléments d'un tableau d'entiers et la multiplication des polynômes.

Recherche dans un tableau

Commençons par la recherche d'un élément dans un tableau donné. Nous allons tout simplement parcourir toute la longueur du tableau en maintenant un indicateur qui nous dit si l'élément en question a été trouvé. L'indicateur en question sera une référence sur un booléen. Avant le début de l'exécution de la boucle, l'élément n'a pas encore été trouvé, donc la référence est initialisée à `false`. Ensuite, cette référence est actualisée au fur et à mesure du parcours du tableau ; cette actualisation n'a pas lieu d'être si l'élément a déjà été trouvé précédemment (d'où le test `if not !trouve then`). A la fin, la valeur de la référence est renvoyée comme valeur de la fonction [1].

```
1 static boolean cherche(int[] tableau, int x) {
2     boolean trouve = false;
3     int n = tableau.length;
4       for(int i=0; i<n; i++)
5           if(!trouve) trouve = (x == tableau[i]);
6      return trouve;
7      }
```

Programme 1.1 –

L'erreur à ne pas faire est d'actualiser systématiquement la valeur de la référence sous la forme

```
1 static boolean mauvais_cherche(int[] tableau, int x) {
2     boolean trouve = false;
3     int n = tableau.length;
4       for(int i=0; i<n; i++)
5           trouve = (x == tableau[i]);
6      return trouve;
7 }
```

Programme 1.2 –

Cette procédure ne teste en effet réellement que le dernier élément du tableau, puisque tous les résultats précédents sont perdus au fur et à mesure. Si l'on cherche à démontrer la validité de la boucle, on détecte facilement ce type d'erreur. Les deux boucles (la bonne et la mauvaise) ont la même condition d'entrée (la *précondition*) : l'indicateur est à `false`. Elles ont la même condition de sortie (la *postcondition*) : tout le tableau a été parcouru et (théoriquement) l'indicateur indique si l'élément figure dans le tableau. Recherchons donc un *invariant* de la boucle, c'est à dire un prédicat qui ne change pas au cours de la boucle.

Dans la boucle correcte, nous pouvons associer à chaque $i \in \{0, \ldots, n-1\}$ le prédicat $P(i)$ défini par

> $P(i)$: à l'entrée dans la i-ième exécution du corps de la boucle (nous parlerons plutôt de la i-ième *itération*) la valeur pointée par `trouve` indique si l'élément `x` figure dans `tableau`$[0], \ldots,$ `tableau`$[i-1]$

1. On remarque que Java interprète notre fonction comme une fonction qui recherche la présence d'un élément de n'importe quel type dans un tableau d'éléments du même type, il s'agit donc d'une fonction *polymorphe* qui serait presque impossible à construire dans un langage comme Pascal

Il est clair que $P(0)$ est vrai. Supposons maintenant que $P(i)$ est vrai ; deux cas sont alors possibles :
- soit l'élément x a déjà été trouvé précédemment et à ce moment là, le corps de la boucle ne fait rien
- soit l'élément x n'a pas encore été trouvé précédemment et à ce moment là, le corps de la boucle teste si l'élément de numéro i du tableau est égal à x

Dans tous les cas, à la fin de l'itération, la valeur pointée par la référence `trouve` indique si l'élément `x` figure dans `tableau`$[0], \ldots,$ `tableau`$[i]$. A ce moment là, i est augmenté de 1 et à l'exécution suivante (ou après la sortie) $P(i+1)$ est vrai. Par une récurrence évidente, $P(i)$ est vrai pour tout i. A la sortie, i vaut n (car il a été augmenté de 1) et donc $P(n)$ est vrai ; autrement dit la valeur pointée par la référence `trouve` indique si l'élément `x` figure dans `tableau`$[0], \ldots,$ `tableau`$[n-1]$, ce que l'on voulait.

Par contre, dans la boucle incorrecte, $P(i)$ n'est plus invariant dans l'exécution de la boucle puisque toute l'information précédente est perdue.

Somme des éléments d'un tableau

Suivant le même principe que pour la recherche d'un élément dans un tableau, pour faire la somme des éléments d'un tableau d'entiers, il suffit d'initialiser une référence à 0 puis d'y ajouter au fur et à mesure les éléments du tableau. On obtient

```
1 static int somme(int[] tableau){
2       int s=0;
3       for(int i=0; i<tableau.length; i++)
4         s+=tableau[i];
5        return  s;
6      }
```

Programme 1.3 –

La preuve de la correction est claire : la précondition est que `somme` vaut 0 ; l'invariant de boucle est

$P(i)$: à l'entrée dans l'itération d'indice i, `somme` contient $\sum_{k=0}^{i-1}$ `tableau`$[k]$

la postcondition est : `somme` contient $\sum_{k=0}^{n-1}$ `tableau`$[k]$, c'est à dire la somme de tous les éléments du tableau.

Multiplication des polynômes

Pour simplifier, nous travaillerons avec des polynômes à coefficients entiers. Si $P(X) = \sum_{i=0}^{n} a_i X^i \in \mathbb{Z}[X]$, nous stockerons les coefficients dans un tableau de taille $n+1$, la valeur de a_i étant stockée dans l'élément de numéro i du tableau. Pour des raisons de commodité, nous définirons une fonction qui renvoie le degré d'un polynôme (égal à la longueur du tableau diminué de 1) et une autre fonction qui initialise un polynôme $P(X) = \sum_{i=0}^{n} a_i X^i$ avec tous les a_i égaux à 0. Pour cela nous utiliserons les deux fonctions prédéfinies de Java : la fonction `vect_length` qui renvoie la longueur d'un tableau et la fonction `make_vect` qui renvoie un tableau de longueur donné avec tous les éléments initialisés à une valeur donnée

```
1 static int deg(float[] pol){
2      return pol.length-1;
3 }
4 static float[] init_poly(int n){
5       return new float[n+1];
6 }
```

Programme 1.4 –

En ce qui concerne le produit de deux polynômes nous utiliserons le schéma de calcul suivant

$$\left(\sum_{i=0}^{m} a_i X^i\right)\left(\sum_{j=0}^{n} b_j X^j\right) = \sum_{i=0}^{m}\left(\sum_{j=0}^{n} a_i b_j X^{i+j}\right)$$

qui est le plus facile à programmer. Il suffit donc de faire parcourir à i l'intervalle $[0, m]$, puis de faire parcourir à j l'intervalle $[0, n]$ en ajoutant le produit $a_i b_j$ au terme de degré $i+j$ du polynôme produit. Ce polynôme produit aura été préalablement initialisé à 0.

```
static float[] mult_poly(float[] p,float[] q){
      int m=deg(p), n=deg(q);
      float[] prod=init_poly(m+n);
      for(int i=0;i <= m;i++)
          for(int j=0; j <= n; j++)
               prod[i+j] += p[i]*q[j];
      return prod;
}
```

Programme 1.5 –

A titre d'application, on peut construire ainsi, en élevant le polynôme $1 + X$ à la puissance n, la famille des coefficients du binôme C_n^k pour $0 \leq k \leq n$; pour cela il suffit de stocker dans une référence sur un polynôme les puissances successives de $1 + X$ en effectuant au fur et à mesure les multiplications par $1 + X$.

```
static float[] binome(int n){
      float[] p={1,1}, puiss={1,1};
      for(int i=2; i <= n; i++)
          puiss=mult_poly(p,puiss);
      return puiss;
}
```

Programme 1.6 –

1.1.3 Boucles conditionnelles

La technique que nous avons utilisée pour rechercher la présence d'un élément dans un tableau est très inefficace puisqu'elle parcourt systématiquement tout le tableau, même si l'élément a été trouvé dès la première exécution de la boucle. Il semble beaucoup plus judicieux de parcourir le tableau, mais de s'arrêter dès que l'élément recherché a été trouvé (à condition qu'il y figure). On tombe sur un cas typique de boucle conditionnelle où la sortie (éventuelle) de la boucle s'effectue après un nombre d'exécutions non prédéterminé et dépend en fait de l'exécution même du corps de la boucle.

On distingue en général quatre types de boucles conditionnelles que l'on peut résumer de la façon suivante

- faire *corps-de-la-boucle* tant que *condition* est vraie
- faire *corps-de-la-boucle* jusqu'à ce que *condition* soit vraie
- tant que *condition* est vraie, faire *corps-de-la-boucle*
- jusqu'à ce que *condition* soit vraie, faire *corps-de-la-boucle*

Les boucles *tant que* et les boucles *jusqu'à ce que* sont équivalentes ; en effet l'instruction *tant que condition est vraie* signifie

- si *condition* est vraie, alors continuer la boucle
- si *condition* est fausse, alors sortir de la boucle

alors que l'instruction *jusqu'à ce que condition soit vraie* signifie

- si *condition* est vraie, alors sortir de la boucle
- si *condition* est fausse, alors continuer la boucle

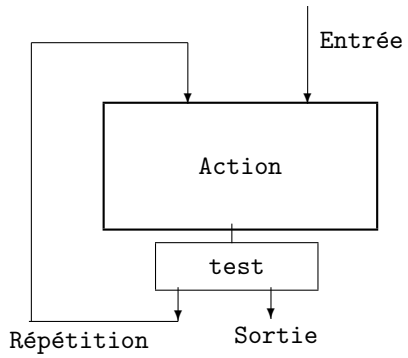
On voit donc que l'instruction *tant que condition est vraie* est équivalente à *jusqu'à ce que (non condition) soit vraie*, où *non condition* désigne la négation de la valeur de *condition*. Comme tous les langages de programmation disposent d'un opérateur de négation, il leur suffit d'implémenter les boucles *tant que* ou les boucles *jusqu'à ce que*.

Il ne nous reste donc qu'à examiner par exemple les deux types de boucles

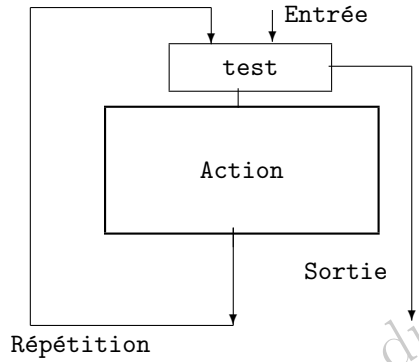
- faire *corps-de-la-boucle* tant que *condition* est vraie
- tant que *condition* est vraie, faire *corps-de-la-boucle*

La différence entre ces deux boucles concerne le moment où le test de *condition* est effectué. Dans une boucle *faire* ... *tant que* ... le test est effectué après l'exécution du corps de la boucle, alors que dans une boucle *tant que* ... *faire* ... le test est effectué avant l'exécution du corps de la boucle. La différence peut sembler minime puisqu'en général, la fin de l'exécution du corps de la boucle n'est autre que le début de l'exécution suivante du corps de la boucle ; c'est presque vrai, sauf qu'échappent à cette règle la première itération ainsi que la dernière. Dans une boucle *faire* ... *tant que* ..., le test suit l'exécution du corps et donc celui-ci est exécuté au moins une fois, la première. Par contre, dans une boucle *tant que* ... *faire* ..., le test précédant l'exécution de la boucle, celle-ci n'est jamais exécutée si au départ la condition n'est pas vérifiée.

Pour résumer sur un graphique, on a les deux types de boucles



faire ... tant que ...



tant que ... faire ...

La plupart des langages possèdent une structure de boucle avec test initial (en général indispensable pour éviter des incohérences graves comme des vecteurs à 0 éléments). Par contre tous ne disposent pas d'une structure de boucle avec test final (en particulier Java). Celle-ci est fort heureusement facile à simuler à l'aide d'une boucle avec test initial en introduisant une variable booléenne provisoire `test` (dont on suppose qu'elle n'intervient ni dans le corps de la boucle, ni dans la condition). La variable `test` est mise à la valeur `true` avant l'entrée dans la boucle et c'est cette variable qui sert ensuite de test de boucle. A la fin du corps de la boucle on écrit simplement `test = condition`. Ceci conduit en pseudo-langage à une structure du type :

```
1
2     test  =  true;
3     tant_que test faire
4         corps_de_la_boucle;
5         test  =  condition
6     fait
```

Programme 1.7 –

et en Java à :

```
1     test = true;
2     while(test){
3         corps_de_la_boucle;
4         test = condition
5   }
```

Programme 1.8 –

Recherche dans un tableau

Revenons à notre problème de recherche d'un élément dans un tableau en parcourant les éléments du tableau à partir du premier. En utilisant une boucle conditionnelle avec test final, il est possible de stopper la recherche dès que l'élément recherché a été trouvé. Encore faut-il ne pas sortir du tableau et savoir si la sortie de la boucle a été provoquée par le fait que l'élément a été trouvé ou par le fait que le tableau a été épuisé (auquel cas l'élément ne figurait pas dans la tableau). Le corps de la boucle doit tester si l'élément a été trouvé, mais aussi actualiser la valeur du compteur. Quant au test, il doit à la fois vérifier que l'élément n'a pas été trouvé lors de l'exécution précédente du corps de la boucle, mais aussi que l'on ne sort pas des limites du tableau (dans ce dernier cas c'est que l'élément n'a pas été trouvé). Ces considérations conduisent à un *premier jet*

```
1  static boolean cherche(int[] tableau, int x){
2       boolean trouve=false;
3       int i=0;
4       while(!trouve && (i<tableau.length)){
5            trouve=(tableau[i] == x);
6            i++;
7        }
8        return trouve;
```

```
9 }
```

Programme 1.9 –

En fait on s'aperçoit qu'à chaque fois que le test est effectué, on recalcule la longueur du tableau et qu'il vaut donc mieux la stocker une bonne fois pour toutes grâce à un nouvel identificateur n et on aboutit à une forme raisonnable :

```
1 static boolean cherche(int[] tableau, int x){
2       boolean trouve=false;
3       int i=0, n=tableau.length;
4       while(!trouve && (i<n)){
5             trouve=(tableau[i] == x);
6             i++;
7        }
8        return trouve;
9 }
```

Programme 1.10 –

Pour démontrer que ce programme fournit bien le résultat voulu, il nous faut donner les préconditions, postconditions et invariant de la boucle. Les préconditions sont claires : `non_trouve` vaut `true` (l'élément est "non trouvé"), $i = 0$ et n est la longueur du tableau. La postcondition est que la variable `non_trouve` vaut `true` si l'élément n'a pas été trouvé, `false` sinon. Quant à l'invariant de boucle, on peut par exemple prendre le prédicat

`non_trouve` indique si l'élément x ne figure pas dans `tableau`$[0], \ldots,$ `tableau`$[i-1]$

Avant l'exécution de la boucle, i a la valeur 0 et x n'appartenant pas à l'ensemble vide, ce prédicat est vrai. Supposons maintenant que ce prédicat est vrai à l'entrée dans le corps de la boucle ; comme ce corps est exécuté, c'est que `non_trouve` vaut false et $i < n$; le corps de la boucle teste alors si `tableau`$[i]$ est égal à x puis augmente i de 1 ; donc à la fin de l'itération `non_trouve` indique de nouveau si l'élément x ne figure pas dans `tableau`$[0], \ldots,$ `tableau`$[i-1]$. Lorsque se produit une sortie de la boucle, ce peut être pour deux raisons

- soit `non_trouve` vaut `false` et l'élément a été trouvé
- soit `non_trouve` vaut `true` et $i = n$, mais alors c'est que l'élément x ne figure pas dans la tableau

Dans ces deux cas, `non_trouve` contient la valeur du prédicat ***x*** *ne figure pas dans* ***tableau*** et il faut donc renvoyer la valeur contraire.

1.2 Le principe de récurrence

1.2.1 Les axiomes de Peano

On montre en théorie des ensembles qu'il existe un ensemble ordonné $(\mathbb{N}, \leq)$ vérifiant les propriétés suivantes

- toute partie non vide de $\mathbb{N}$ a un plus petit élément
- toute partie non vide majorée de $\mathbb{N}$ a un plus grand élément
- $\mathbb{N}$ n'a pas de plus grand élément

et que cet ensemble ordonné est unique à une bijection croissante près (isomorphisme d'ensemble ordonné). Considérons un tel ensemble dont nous allons voir un certain nombre de propriétés

Proposition 1.2.1 *L'ensemble* $(\mathbb{N}, \leq)$ *est totalement ordonné.*

Démonstration: Si $a, b \in \mathbb{N}$, la partie $\{a, b\}$ doit avoir un plus petit élément, et donc on a soit $a \leq b$, soit $b \leq a$.

Proposition 1.2.2 *L'ensemble* $(\mathbb{N}, \leq)$ *a un plus petit élément noté 0.*

Démonstration: Comme toute partie de $\mathbb{N}$, $\mathbb{N}$ lui même a un plus petit élément.

Proposition 1.2.3 *Tout élément a de $\mathbb{N}$ a un unique successeur, c'est à dire un élément b de $\mathbb{N}$ tel que $b > a$ et l'intervalle ouvert $]a, b[$ est vide.*

Démonstration: Comme a n'est pas plus grand élément de $\mathbb{N}$, il existe des éléments x de $\mathbb{N}$ tels que $a < x$. Soit donc $X = \{x \in \mathbb{N} \mid x > a\}$; X est une partie non vide de $\mathbb{N}$, donc elle a un plus petit élément b. Si $]a, b[$ n'était pas vide, il existerait un élément x dans $]a, b[$ qui vérifierait à la fois $x > a$, donc $x \in X$ et $x < b = \min X$ ce qui est absurde. L'unicité résulte bien évidemment de l'unicité du plus petit élément.

Proposition 1.2.4 *Tout élément a de $\mathbb{N}$ différent de 0 a un unique prédécesseur, c'est à dire un élément b de $\mathbb{N}$ tel que $b < a$ et l'intervalle ouvert $]b, a[$ est vide.*

Démonstration: Comme a n'est pas le plus petit élément de $\mathbb{N}$, il existe des éléments x de $\mathbb{N}$ tels que $x < a$. Soit donc $X = \{x \in \mathbb{N} \mid x < a\}$; X est une partie non vide de $\mathbb{N}$, majorée par a, donc elle a un plus grand élément b. Si $]b, a[$ n'était pas vide, il existerait un élément x dans $]a, b[$ qui vérifierait à la fois $x < a$, donc $x \in X$ et $x > b = \max X$ ce qui est absurde. L'unicité résulte bien évidemment de l'unicité du plus grand élément.

Remarque Nous noterons $n+1$ le successeur de n et $n-1$ le prédécesseur de n (si $n > 0$). La définition même du prédécesseur et du successeur montre que

$$\forall n \in \mathbb{N},\ (n+1) - 1 = n \tag{1.1}$$
$$\forall n \in \mathbb{N}^*,\ (n-1) + 1 = n \tag{1.2}$$

1.2.2 Principe de récurrence simple

Définition 1.2.1 *Soit E un ensemble ; on appelle prédicat sur E toute propriété $P(x)$ dépendant de $x \in E$ et dont on peut dire pour tout $x \in E$ si elle est vraie ou fausse, autrement dit une application P de E dans l'ensemble à deux éléments $\{vrai, faux\}$.*

Dans la pratique, si P est un tel prédicat, on écrira souvent $P(x)$ à la place de $P(x) = vrai$.

Théorème 1.2.1 (Principe de récurrence simple) *Soit P un prédicat sur $\mathbb{N}$. On suppose que*
- *$P(0)$ est vrai*
- *$\forall n \in \mathbb{N},\ P(n) \Rightarrow P(n+1)$*

Alors, pour tout $n \in \mathbb{N}$, $P(n)$ est vrai.

Démonstration: Soit X l'ensemble des $n \in \mathbb{N}$ tel que $P(n)$ soit fausse. Si X est non vide, il a un plus petit élément $a \in X$. Comme $P(0)$ est vraie, on a $a \neq 0$. Donc a a un prédécesseur $a-1$. Comme $a - 1 < a = \min X$, on a $a - 1 \notin X$, donc $P(a-1)$ est vraie. Mais $a = (a-1) + 1$ et comme $P(a-1)$ est vraie, $P(a) = P((a-1)+1)$ est vraie, ce qui contredit $a \in X$. Donc X est vide, et donc pour tout $n \in \mathbb{N}$, $P(n)$ est vrai.

Remarque De la même façon on montre que, pour $n_0 \in \mathbb{N}$, si
- $P(n_0)$ est vrai
- $\forall n \geq n_0,\ P(n) \Rightarrow P(n+1)$

alors, pour tout $n \geq n_0$, $P(n)$ est vrai.

Exemple On montre ainsi que $\forall n \in \mathbb{N},\ 0^2 + 1^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$. Soit $S_n = 0^2 + 1^2 + \ldots + n^2$ et soit $P(n)$ le prédicat

$$S_n = \frac{n(n+1)(2n+1)}{6}$$

Alors $P(0)$ est vrai et **si** $P(n)$ est vrai, **alors**

$$\begin{aligned} S_{n+1} &= S_n + (n+1)^2 = \frac{n(n+1)(2n+1)}{6} + (n+1)^2 \\ &= \frac{n+1}{6}\left(n(2n+1) + 6(n+1)\right) = \frac{n+1}{6}(2n^2 + 7n + 6) \\ &= \frac{(n+1)(n+2)(2n+3)}{6} \end{aligned}$$

donc $P(n+1)$ est vrai. Ceci montre que $P(n)$ est vrai pour tout $n \in \mathbb{N}$.

1.2.3 Principe de récurrence étendu

Théorème 1.2.2 (Principe de récurrence étendu) *Soit P un prédicat sur $\mathbb{N}$. On suppose que*
- *$P(0)$ est vrai*
- *$\forall n \in \mathbb{N},\ (\forall k < n,\ P(k)$ est vrai$) \Rightarrow P(n)$ est vrai*

Alors, pour tout $n \in \mathbb{N}$, $P(n)$ est vrai.

Démonstration: Soit X l'ensemble des $n \in \mathbb{N}$ tel que $P(n)$ soit faux. Si X est non vide, il a un plus petit élément $a \in X$. Comme $P(0)$ est vrai, on a $a \neq 0$. Pour tout $k < a$, on a $k \notin X$, donc $P(k)$ est vrai. On en déduit que $P(a)$ est vrai, ce qui contredit $a \in X$. Donc X est vide, et donc pour tout $n \in \mathbb{N}$, $P(n)$ est vrai [2].

Exemple On peut ainsi montrer que tout nombre entier supérieur ou égal à 2 est produit de nombres premiers. C'est clair si $n = 2$. Si maintenant la propriété est vraie pour tous les nombres $2, \ldots, n-1$ alors deux cas sont possibles. Soit n est premier et alors il est produit d'un nombre premier. Soit n n'est pas premier et il est produit de deux nombres entiers p et q avec $2 \leq p \leq n-1$ et $2 \leq q \leq n-1$; d'après l'hypothèse de récurrence, p et q sont produits de nombres premiers et donc n aussi, ce qui achève la récurrence.

1.2.4 Ensembles bien ordonnés, bien fondés

Le principe de récurrence simple est très intimement lié à la structure de l'ensemble des entiers naturels et à l'existence des fonctions réciproques *prédécesseur* et *successeur*. Par contre, le principe de récurrence étendu se généralise facilement à d'autres ensembles ordonnés. La propriété essentielle de l'ensemble $\mathbb{N}$ que nous avons utilisée pour la démonstration de ce principe est le fait qu'une partie non vide a un plus petit élément. Cette propriété s'étend à d'autres ensembles ordonnés.

Définition 1.2.2 *On dit qu'un ensemble ordonné $(E, \preceq)$ est bien ordonné si toute partie non vide a un plus petit élément.*

Remarque Dans un ensemble bien ordonné, une partie à deux éléments doit nécessairement avoir un plus petit élément, ce qui montre que l'ordre est total. Par contre, il existe des ensembles totalement ordonnés qui ne sont pas bien ordonnés, comme par exemple l'ensemble $\mathbb{Z}$ des entiers relatifs et l'ensemble $\mathbb{R}$ des nombres réels (munis de leurs ordres naturels).

Exemple Sur $\mathbb{N}^2 = \mathbb{N} \times \mathbb{N}$, on pose

$$(n,p) \preceq (n',p') \text{ si } \Big(n < n' \text{ ou } (n = n' \text{ et } p \leq p')\Big)$$

Soit A une partie non vide de $\mathbb{N}^2$. Alors A a un plus petit élément (n_0, p_0), où $n_0 = \min\{n \mid \exists p \in \mathbb{N},\ (n,p) \in A\}$ et $p_0 = \min\{p \mid (n_0,p) \in A\}$. L'ordre en question sur $\mathbb{N}^2$ est un bon ordre appelé l'ordre *lexicographique* ; il sera largement utilisé par la suite.

Théorème 1.2.3 (Principe d'induction) *Soit P un prédicat sur l'ensemble bien ordonné $(E, \preceq)$. On suppose que*
- *$P(x_0)$ est vrai, où x_0 est le plus petit élément de E*
- *$\forall x \in E,\ (\forall y \prec x,\ P(y)$ est vrai$) \Rightarrow P(x)$ est vrai*

Alors, pour tout $x \in E$, $P(x)$ est vrai.

Démonstration: Exactement la même que pour le principe de récurrence étendu.

2. Les logiciens purs et durs nous objecteront que la deuxième hypothèse implique la première. Les pédagogues raisonnables leur répondront qu'il vaut mieux mettre les deux.

En fait, il est possible de généraliser encore ce principe à l'aide de la notion d'ordre bien fondé qui généralise celle de bon ordre.

Définition 1.2.3 *On dit qu'un ordre $\preceq$ sur E est bien fondé si toute partie non vide a un élément minimal.*

Exemple Sur l'ensemble $\mathbb{N}^*$ des entiers naturels non nuls la relation $x \preceq y$ si x divise y est un ordre bien fondé : tout plus petit élément d'une partie au sens de l'ordre ordinaire est un élément minimal pour la relation de divisibilité. Les éléments minimaux de $\mathbb{N} \setminus \{0, 1\}$ sont les nombres premiers.

La proposition suivante est intéressante en programmation car elle constitue en général un argument décisif pour montrer qu'une boucle se termine.

Proposition 1.2.5 *L'ordre $\preceq$ sur E est bien fondé si et seulement si il n'existe pas de suite strictement décroissante d'éléments de E.*

Démonstration: Supposons tout d'abord qu'il existe dans E une suite $(a_n)_{n\in\mathbb{N}}$ strictement décroissante et soit $A = \{a_n \mid n \in \mathbb{N}\}$; tout élément de A est de la forme a_p avec $a_{p+1} \prec a_p$, donc A n'admet pas d'élément minimal. Inversement, supposons qu'il existe une partie non vide A n'admettant pas d'élément minimal et soit $a_0 \in A$. L'élément a_0 n'est pas minimal dans A et donc il existe $a_1 \in A$ tel que $a_1 \prec a_0$. Supposons $a_0, \ldots, a_n$ construits tels que $a_n \prec a_{n-1} \prec \ldots \prec a_0$. Puisque a_n n'est pas élément minimal de A, il existe $a_{n+1} \in A$ tel que $a_{n+1} \prec a_n$. On construit ainsi par récurrence une suite strictement décroissante d'éléments de A. On a donc montré l'équivalence entre *l'ordre n'est pas bien fondé* et *il existe une suite strictement décroissante d'éléments de E*, ce qui est le résultat cherché.

On a alors

Théorème 1.2.4 (Principe d'induction généralisé) *Soit P un prédicat sur l'ensemble ordonné $(E, \preceq)$, où $\preceq$ est un ordre bien fondé. On suppose que*
- *$P(x)$ est vrai pour tous les éléments minimaux de E*
- *$\forall x \in E$, $(\forall y \prec x,\ P(y)$ est vrai$) \Rightarrow P(x)$ est vrai*

Alors, pour tout $x \in E$, $P(x)$ est vrai.

Démonstration: Soit X l'ensemble des $x \in E$ tel que $P(x)$ soit faux. Si X est non vide, il a un élément minimal $a \in X$. Comme $P(x)$ est vrai pour les éléments minimaux de E, a n'est pas un élément minimal de E. Pour tout $x \prec a$, on a $x \notin X$, donc $P(x)$ est vrai. On en déduit que $P(a)$ est vrai, ce qui contredit $a \in X$. Donc X est vide, et pour tout $x \in E$, $P(x)$ est vrai.

1.3 Récursivité

1.3.1 Procédures et fonctions récursives simples

Nous allons tout d'abord étudier le cas de la récursivité simple indexée par $\mathbb{N}$. Supposons que nous ayons un problème à résoudre dépendant d'un entier n et considérons le prédicat (un peu informel)

$$P(n) : \text{je sais résoudre le problème pour } n$$

Le principe de récurrence simple nous dit que si nous savons résoudre le problème pour $n = 0$ et si nous savons comment passer d'une solution pour n à une solution pour $n + 1$, autrement dit si
- $P(0)$ est vrai
- ($P(n)$ est vrai)$\Rightarrow$ ($P(n + 1)$ est vrai)

alors pour tout $n \in \mathbb{N}$, $P(n)$ est vrai ; nous savons donc résoudre notre problème pour tout $n \in \mathbb{N}$. Nous allons traiter quelques exemples de cette démarche dans la suite de ce paragraphe.

Calcul des puissances La définition même de la puissance n-ième en mathématiques est récursive, puisque l'on pose $a^0 = 1$ (on sait calculer la puissance n-ième pour $n = 0$) et $a^{n+1} = aa^n$ (si on sait calculer la puissance n-ième, on sait calculer la puissance $n+1$-ième). Nous pouvons donc définir une fonction puissance en Java de la façon suivante :

```
1 static int puiss(int x,int n){
2         if(n==0) return 1;
3         else return x*puiss(x,n-1);
4 }
```

Programme 1.11 –

Calcul de la factorielle La définition même de la factorielle de n en mathématiques est récursive, puisque l'on pose $0! = 1$ (on sait calculer la factorielle de 0) et $(n+1)! = (n+1)n!$ (si on sait calculer la factorielle de n, on sait calculer la factorielle de $n+1$). On posera donc :

```
1 int fact(int n){
2     if(n==0) return 1;
3     else return n*fact(n-1);
4 }
```

Programme 1.12 –

Un bon moyen d'observer ce qui se passe, est de demander à Java de *tracer* (au sens de suivre à la trace) la fonction `fact`. Cela se fait très simplement en modifiant quelque peu notre fonction pour qu'elle affiche la valeur du paramètre d'entrée et al valeur retournée. A partir de là, chaque fois que la fonction Java `trace_fact` est appelée, Java affiche les valeurs des paramètres qu'elle reçoit en entrée sous la forme `fact  = -...`, et chaque fois qu'elle retourne un résultat, est affichée la valeur de ce résultat sous la forme `fact -->...`. Voici un exemple

```
1  static int trace_fact(int n){
2     int res;
3     System.out.println("fact  =  "+ n);
4      if(n==0) return 1;
5      else{
6           res=n*trace_fact(n-1);
7           System.out.println("fact  -->  "+res);
8           return res;
9        }
10 }
11 #trace_fact(3);
12 fact  = - 3
13 fact  = - 2
14 fact  = - 1
15 fact  = - 0
16 fact --> 1
17 fact --> 2
18 fact --> 6
```

Programme 1.13 –

On voit que `fact` reçoit en entrée le paramètre 3, puis qu'elle est de nouveau appelée avec en entrée le paramètre 2, puis qu'elle est de nouveau appelée avec en entrée le paramètre 1, puis qu'elle est de nouveau appelée avec en entrée le paramètre 0 ; à ce moment là, la fonction `fact` retourne comme valeur $0! = 1$, puis l'appel précédent renvoie $1 * 0! = 1$, puis l'appel précédent renvoie $2 * 1! = 2$ et enfin le premier appel effectué renvoie $3 * 2! = 6$, ce qui achève le calcul.

Miroir d'un mot Prenons maintenant le problème de l'image miroir d'un mot, l'image du mot *abcdef* devant être *fedcba*. Le retournement d'un mot d'un seul caractère est trivial : il suffit de ne rien faire. Pour retourner un mot à n caractères, il suffit de retourner le mot formé par les $n-1$ premiers caractères et de le concaténer avec le dernier caractère mis en première position. En utilisant les fonctions suivantes de la bibliothèque Java de base

- `s.length()` qui retourne la longueur de la chaîne `s`
- `s.subtring(debut, fin)` qui retourne une chaîne de caractères de s en commençant au caractère numéro *debut* (le premier caractère portant le numéro 0) et en terminant au caractère *fin-1*
- `s.concat(s1)` qui concatène (*colle ensemble*) deux chaînes de caractères s et *s1*

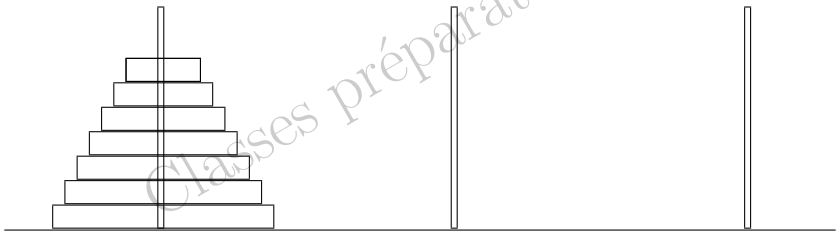
on obtient une version possible (dont l'efficacité sera discutée plus loin)

```
1  static String retourne(String  s){
2        int n = s.length();
3        if(n == 1) return s;
4        else return (s.substring(n-1,n)).concat(retourne (s.substring(0,n-1)));
5  }
```

Programme 1.14 –

Les tours de Hanoi Il s'agit d'un exemple classique qui montre la puissance de la récursivité (même simple) pour résoudre un problème. Une ancienne légende raconterait (d'après certains informaticiens) que des moines de Hanoi se livrent depuis la nuit des temps à une curieuse occupation. Dans la cour de leur temple se trouvent trois piquets, les fameuses tours. Ces moines disposent de disques en or de diamètres décroissants percés en leurs centres de manière à pouvoir les enfiler sur les piquets. La règle à respecter est que le diamètre d'un disque placé au dessus doit toujours être plus petit que le diamètre du disque placé en dessous.

A l'origine du jeu, tous les disques étaient empilés sur le piquet de gauche (le plus à l'ouest). Les moines ne peuvent déplacer qu'un disque à la fois de l'un des piquets vers l'un des deux autres piquets, à condition de toujours respecter la règle de décroissance des diamètres. Le but que poursuivent les moines est de transférer tous les disques sur le piquet de droite (le plus à l'est) : la légende assure que cela sonnera la fin de notre monde.



position initiale

Le problème semble difficile à résoudre. Pourtant une solution récursive est immédiate. Appelons n le nombre de disques à transférer. Si $n = 1$, la solution est triviale, il suffit de transférer le seul et unique disque. Supposons que nous sachions transférer $n-1$ disques et que nous voulions en transférer n d'un piquet a vers le piquet c en utilisant comme piquet intermédiaire b. Il nous suffit de transférer les $n-1$ disques supérieurs du piquet a vers le piquet b (le disque inférieur étant le plus grand de tous et ne bougeant pas, il n'impose aucune contrainte aux mouvements des autres et tout se déroule comme s'il n'existait pas), puis de transférer le dernier disque de a vers c (qui est libre) et enfin de transférer les $n-1$ disques de b vers c en utilisant comme piquet intermédiaire a : de nouveau le plus grand des disques ne bouge pas et donc n'impose aucune contrainte aux mouvements des autres. Autrement dit, si on appelle *transfert* n a b c le transfert de n disques du piquet a vers le piquet c en utilisant l'intermédiaire b, on peut le réaliser par

- `transfert` $n-1$ `a c b`
- `transfert 1 a b c` (ce transfert en fait n'utilise pas `b`)
- `transfert` $n-1$ `b a c`

En Java, nous appellerons les piquets *gauche*, *milieu* et *droite* et le transfert du disque supérieur du piquet a vers le piquet c sera noté a`-->`c. On obtient alors la procédure suivante :

```
1  static void Hanoi(int n, String a, String b, String c){
2        if(n==1) System.out.println(a+" --> "+c);
3        else{
4            Hanoi(n-1,a,c,b);
5            Hanoi(1,a,b,c);
6            Hanoi(n-1,b,a,c);
7          }
8  }
9  Hanoi(3, "gauche", "milieu", "droite");
10 gauche --> droite
11 gauche --> milieu
```

```
12 droite --> milieu
13 gauche --> droite
14 milieu --> gauche
15 milieu --> droite
16 gauche --> droite
```

Programme 1.15 –

La méthodologie récursive nous a permis de résoudre très simplement un problème d'apparence bien complexe. Faisons une petite étude du temps nécessaire à l'exécution de cet algorithme récursif. Si nous notons a_n le nombre d'opérations de transfert d'un disque d'un piquet à un autre, la suite a_n est définie par récurrence par $a_1 = 1$ et $a_n = 2a_{n-1} + 1$. Le lecteur vérifiera immédiatement que l'unique solution de cette récurrence est $a_n = 2^n - 1$. Dans la légende originale, il y avait 64 disques à transférer. Un petit calcul rapide montrera que même en déplaçant un disque par seconde (ce qui est quand même un bon rythme pour un moine), la fin du monde ne devrait pas intervenir avant quelques centaines de milliards d'années : on obtient comme temps nécessaire quelques 18446744073709551615 secondes, soit environ $6 \cdot 10^{11}$ années. Ceci laisse du temps devant nous. On peut en effet montrer qu'il n'existe pas de solution plus efficace (en nombre d'opérations) que celle que nous avons donnée.

1.3.2 Procédures et fonctions récursives multiples

Pour le moment nous ne nous sommes intéressés qu'à des récursivités simples, où l'on passe d'une solution pour n à une solution pour $n+1$. On rencontre fréquemment des types de problèmes dépendant d'un entier n pour lesquels la solution pour n dépend de la solution du même problème pour plusieurs entiers strictement inférieurs à n, ce que nous appellerons une récursivité multiple.

Un exemple classique est celui de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ définie par $F_0 = F_1 = 1$ et $\forall n \geq 2,\ F_n = F_{n-1} + F_{n-2}$. Ces récurrences multiples peuvent se résoudre de la même façon que les récurrences simples. C'est ainsi que l'on peut définir en Java :

```
1 static int fibo(int n){
2       if(n==0 || n==1) return 1;
3       else return fibo(n-1)+fibo(n-2);
4 }
```

Programme 1.16 –

Faisons une étude du temps de calcul de F_n pour un n donné ; ce temps de calcul est en gros proportionnel au nombre d'appels T_n de la fonction `fibo`. Or il est clair que $T_0 = 1$, $T_1 = 1$ et que $T_n = T_{n-1} + T_{n-2}$ pour $n \geq 2$ si bien que $T_n = F_n$. Mais il est facile de montrer par récurrence que

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{n+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{n+1} \right)$$

si bien que F_n est équivalent, quand n tend vers $+\infty$ à $\frac{1}{\sqrt{5}}\omega^{n+1}$ avec $\omega = \frac{1+\sqrt{5}}{2} \approx 1.6$. A titre d'exemples, à supposer que notre ordinateur soit capable d'effectuer 10^6 appels de la fonction `fibo` par seconde, le calcul de F_{64} par cette méthode demanderait environ 200 jours, quant au calcul de F_{100} il demanderait environ 18 millions d'années [3] !

Il faut dire que notre méthode est plutôt inefficace. Pour le constater, il suffit de *tracer* notre fonction `fibo`.

```
1
2 #trace "fibo";
3 >> fibo is now traced.
4 - : unit = ()
5 #fibo 4;
6 fibo  = - 4
7 fibo  = - 2
8 fibo  = - 0
9 fibo --> 1
10 fibo  = - 1
11 fibo --> 1
12 fibo --> 2
```

3. En fait, indépendamment du temps nécessaire, les deux calculs ne sont pas faisables directement à cause des limitations de Java sur la taille des entiers, le débordement se produisant aux alentours de $n = 35$. Nous verrons plus tard comment remédier à cet état de fait.

```
fibo  = - 3
fibo  = - 1
fibo --> 1
fibo  = - 2
fibo  = - 0
fibo --> 1
fibo  = - 1
fibo --> 1
fibo --> 2
fibo --> 3
fibo --> 5
- : int = 5
```

Programme 1.17 –

On voit que F_3 a été calculé 1 fois, F_2 a été calculé 2 fois, F_1 a été calculé 3 fois et F_0 a été calculé 2 fois. Il est facile de modifier notre procédure pour qu'elle affiche en plus le nombre de fois où elle a été appelée avec un certain paramètre. Pour cela nous utiliserons un tableau initialisé à 0 et nous incrémenterons le i-ième élément de ce tableau chaque fois que la procédure sera appelée avec le paramètre i. On obtient une nouvelle fonction `essai_fibo` qui commence par initialiser le tableau dans lequel est stocké le nombre des appels puis définit la fonction récursive `fibo` qui calcule effectivement F_n en mettant à jour au fur et à mesure le tableau `nb_appels` ; ensuite `essai_fibo` appelle effectivement la fonction `fibo` avec le paramètre n et retourne le tableau des nombres d'appels.

```
static int fibo(int n, int[] nbAppels){
       nbAppels[n]++;
       if(n==0 || n==1) return 1;
       else return fibo(n-1,nbAppels)+fibo(n-2,nbAppels);
}

static int[] essaiFibo(int n){
        int[] nbAppels=new int[n+1];
        int res=fibo(n,nbAppels);
        return nbAppels;
}
essaiFibo(14);
[233; 377; 233; 144; 89; 55; 34; 21; 13; 8; 5; 3; 2; 1; 1]
```

Programme 1.18 –

On constate ainsi que F_2 a été calculé 233 fois lors du calcul de F_{14}.

> **Note de programmation** On aurait pu bien entendu utiliser un tableau global pour stocker le nombre d'appels (car il n'est pas question d'en faire une variable locale à la fonction `fibo` qui serait recréée à chaque appel de la fonction). Cela aurait pourtant de nombreux inconvénients :
> – mobiliser en permanence de la mémoire alors qu'on n'a besoin de ce tableau que lors du calcul du nombre de Fibonacci
> – lors de plusieurs utilisations successives de notre fonction, on risque d'oublier de réinitialiser à 0 le tableau, si bien que les nombres obtenus n'auraient plus aucune signification

La principale source d'inefficacité de notre fonction réside dans le recalcul incessant des mêmes valeurs. Pour éviter ce recalcul, il suffit de stocker au fur et à mesure les valeurs déjà calculées ; lorsque la fonction `fibo` est appelée avec un paramètre i, on commence par tester si cette valeur a déjà été calculée ; si c'est le cas, la fonction retourne directement le résultat déjà calculé, sinon, elle s'appelle récursivement. Un moyen simple d'effectuer ce test est de tenir compte de ce que les nombres de Fibonacci sont strictement positifs : on initialise un tableau à 0 et lors d'un appel à `fibo n`,

– soit l'élément d'indice n du tableau vaut 0, auquel cas F_n n'a pas encore été calculé, il faut donc le calculer et stocker le résultat dans le tableau avant de le retourner,
– soit l'élément n du tableau ne vaut pas 0, auquel cas F_n a déjà été calculé et il suffit de retourner cet élément.

On obtient

```
static int fibo_aux(int n, int[] valeursFibo){
       int res;
       if(valeursFibo[n]!=0) return valeursFibo[n];
       else{
           res= fibo_aux(n-1,valeursFibo)+fibo_aux(n-2,valeursFibo);
           valeursFibo[n]=res;
           return res;
         }
}

static int fibo(int n){
        int[] valeursFibo=new int[n+1];
        valeursFibo[0]=1;
        valeursFibo[1]=1;
        return fibo_aux(n,valeursFibo);
}
```

```
17
18 fibo(35);
19 - : int = 14930352
```

Programme 1.19 –

On constate alors que la fonction `fibo` n'est appelée avec le paramètre i qu'au plus deux fois, lors du premier calcul éventuel de F_{i+1} et F_{i+2}. Le nombre total d'appels de la fonction `fibo` est cette fois majoré par $2(n+1)$. Dans le cas particulier [4] de F_{100}, le calcul demanderait seulement dans les 200 appels de la fonction, au lieu des 10^{22} appels dans la méthode purement récursive. Pour le calcul de F_{100}, la méthode avec stockage est en gros cent milliards de milliards de fois plus rapide !

Nous voyons sur cet exemple que la méthode consistant à stocker les résultats intermédiaires nous a permis de diminuer la *complexité* de l'algorithme de calcul de F_n

- dans la méthode sans stockage, le temps d'exécution est proportionnel à $\left(\frac{1+\sqrt{5}}{2}\right)^n$
- dans la méthode avec stockage, le temps d'exécution est proportionnel à n

Bien entendu le gain de rapidité, qui est gigantesque dès que n atteint quelques dizaines, a un coût : il a fallu utiliser de la mémoire pour stocker les résultats intermédiaires. En fait le programmeur est souvent confronté à un dilemme

- soit utiliser un algorithme plus rapide au prix d'une occupation plus importante en mémoire
- soit gagner de la place en mémoire (quand celle-ci est limitée) au prix d'un ralentissement de l'exécution

Dans notre cas particulier, l'occupation en mémoire supplémentaire, qui est l'ordre de quelques centaines d'octets, est tout à fait négligeable en face du gain de rapidité considérable obtenu, et il n'y pas d'hésitation possible : il faut utiliser le stockage des valeurs intermédiaires.

Généralisation de la récursivité multiple Soit E un ensemble muni d'un ordre bien fondé $\preceq$, supposons donnée une partie A de E et soit $\varphi_1, \ldots, \varphi_k$ des applications de $E \setminus A$ dans E vérifiant

$$\forall i \in \{1, \ldots, k\},\ \forall x \in E \setminus A,\ \varphi_i(x) \prec x$$

Nous supposons que nous avons un problème à résoudre pour $x \in E$ et que

- nous savons résoudre le problème pour $x \in A$
- nous savons trouver une solution de notre problème pour $x \in E$ si nous connaissons une solution pour $\varphi_1(x), \ldots, \varphi_k(x)$

Alors, d'après le principe d'induction généralisé appliqué au prédicat

$$P(x) : \text{je sais résoudre le problème pour } x$$

on sait résoudre le problème pour tout $x \in E$.

Essayons par exemple de construire une fonction qui calcule le pgcd de deux entiers sans division. Nous pouvons appliquer les règles suivantes

- si $x > y$, alors $\text{pgcd}(x, y) = \text{pgcd}(y, x)$
- si $x \leq y$ et $x \geq 1$, alors $\text{pgcd}(x, y) = \text{pgcd}(x, y - x)$
- si $x = 0$ alors $\text{pgcd}(x, y) = y$

Nous avons ici $A = \{(0, y) \mid y \in \mathbb{N}\}$ et $\varphi_1 : \mathbb{N}^2 \setminus A \to \mathbb{N}^2$ définie par

$$\varphi_1(x, y) = \begin{cases} (y, x) & \text{si } x > y \\ (x, y - x) & \text{si } 1 \leq x \leq y \end{cases}$$

Munissons $\mathbb{N}^2$ de l'ordre lexicographique

$$(n, m) \preceq (n', m') \text{ si } \Big(n < n' \text{ ou } (n = n' \text{ et } m \leq m')\Big)$$

On sait qu'il s'agit là d'un bon ordre. De plus il est clair que

$$\forall (x, y) \in \mathbb{N}^2 \setminus A,\ \varphi_1(x, y) \prec (x, y)$$

Le principe d'induction généralisé nous montre que nous pouvons ainsi calculer le pgcd de deux entiers. Voici la fonction Java correspondante :

4. En fait de nouveau le calcul n'est pas faisable directement à cause des limitations de Java sur la taille des entiers

```
1 static long pgcd(long x,long y){
2         if(x>y) return pgcd(y,x);
3         if(x>0) return pgcd(x,y-x);
4         else return x;
5 }
6
7 #pgcd( 69825,53865);
8 1995
```

Programme 1.20 –

L'algorithme d'Euclide L'exemple du calcul du pgcd par soustraction est un exemple d'école qui ne prétend à aucune efficacité ; par exemple la fonction est appelée 2000 fois pour calculer le pgcd de 2000 et de 1 ! On gagne un temps considérable en remplaçant $y - x$ par le reste de la division euclidienne de y par x que l'on peut écrire en mathématiques sous la forme $y \bmod x$ et en Java sous la forme `y mod x`. On obtient ainsi l'algorithme d'Euclide dont la justesse se démontre de la même façon que dans le cas précédent en posant $A = \{(0, y) \mid y \in \mathbb{N}\}$, $\varphi_1 : \mathbb{N}^2 \setminus A \to \mathbb{N}^2$ définie par

$$\varphi_1(x, y) = \begin{cases} (y, x) & \text{si } x > y \\ (x, y \bmod x) & \text{si } 1 \leq x \leq y \end{cases}$$

et en utilisant l'ordre lexicographique sur $\mathbb{N}^2$. Voici la fonction Java correspondante :

```
1 static long pgcd(long x,long y){
2         if(x>y) return pgcd(y,x);
3         if(x>0) return pgcd(x,y % x);
4         else return y;
5 }
6 #pgcd(710220,3644865);
7 1995
```

Programme 1.21 –

La fonction d'Ackermann La récursivité multiple peut prendre des formes encore plus complexes où la fonction elle même que l'on cherche à calculer intervient dans la définition des fonctions φ_i.

La fonction d'Ackermann est la fonction $\mathrm{Ack} : \mathbb{N}^2 \to \mathbb{N}$ *définie* par

- $\mathrm{Ack}(0, n) = n + 1$
- $\mathrm{Ack}(m, 0) = \mathrm{Ack}(m - 1, 1)$ si $n \geq 1$
- $\mathrm{Ack}(m, n) = \mathrm{Ack}(m - 1, \mathrm{Ack}(m, n - 1))$ si $n \geq 1$ et $m \geq 1$.

Pour voir que la fonction est effectivement définie, nous allons appliquer le principe d'induction. Pour cela, nous munissons $\mathbb{N}^2$ de l'ordre lexicographique

$$(n, m) \preceq (n', m') \text{ si } \Big(n < n' \text{ ou } (n = n' \text{ et } m \leq m')\Big)$$

On sait qu'il s'agit là d'un bon ordre. Considérons le prédicat suivant

$$P(n, m) : \text{les formules de définition calculent } \mathrm{Ack}(n, m) \text{ en un nombre fini d'opérations}$$

Nous allons montrer que $P(n, m)$ est vrai pour tout $(n, m) \in \mathbb{N}^2$. Raisonnons par l'absurde et supposons $A = \{(n, m) \in \mathbb{N}^2 \mid P(n, m) \text{ est faux }\}$ non vide. Alors A a un plus petit élément (n_0, m_0). Comme les formules de définition calculent en une seule fois $\mathrm{Ack}(n, m)$ si $n = 0$, on a $n_0 > 0$.

- si $m_0 = 0$, comme $(n_0 - 1, 1) \prec (n_0, m_0)$, les formules calculent $\mathrm{Ack}(n_0 - 1, 1) = \mathrm{Ack}(n_0, 0)$ en un nombre fini d'opérations soit $(n_0, m_0) \notin A$, ce qui est absurde
- si $m_0 > 0$, comme $(n_0, m_0 - 1) \prec (n_0, m_0)$, les formules calculent $\mathrm{Ack}(n_0, m_0 - 1)$ en un nombre fini d'opérations et comme $(n_0 - 1, \mathrm{Ack}(n_0, m_0 - 1)) \prec (n_0, m_0)$ les formules de définition calculent $\mathrm{Ack}(n_0 - 1, \mathrm{Ack}(n_0, m_0 - 1)) = \mathrm{Ack}(n_0, m_0)$ un un nombre fini d'opérations, soit $(n_0, m_0) \notin A$, ce qui est absurde.

On a donc finalement $A = \emptyset$, ce qui montre que la fonction est bien définie.

En Java, on peut donc écrire :

```
static long ack(long m, long n){
        if(m==0) return n+1;
        if(n==0) return ack(m-1,1);
        return ack(m-1,ack(m,n-1));
}

ack (3,4);
- : int = 125
```

Programme 1.22 –

1.3.3 Procédures et fonctions récursives croisées

Pour terminer notre étude de différents modes de récursivité, nous évoquerons les fonctions récursives croisées à l'aide d'un exemple tiré des mathématiques. Nous en rencontrerons des exemples plus fondamentaux dans le chapitre sur les expressions algébriques.

Pour illustrer cette récursivité croisée, nous utiliserons l'exemple de l'étude de suites récurrentes du type $x_n = f(x_{n-1}, y_{n-1}), y_n = g(x_{n-1}, y_{n-1})$, les valeurs initiales x_0 et y_0 étant données. Le calcul de x_n et y_n nécessite donc la connaissance simultanée de x_{n-1} et de y_{n-1}. Il est clair que la déclaration des fonctions x et y doit être à la fois récursive et simultanée.

Définissons par exemple deux suites récurrentes par $x_n = x_{n-1} + y_{n-1}$ et $y_n = x_{n-1}y_{n-1}$ avec $x_0 = 1$, $y_0 = 2$. Pour calculer x_n et y_n on peut utiliser les deux fonctions Java suivantes

```
static long x(int n){
        if(n==0) return 1;
        else return x(n-1)+y(n-1);
}

static long y(int n){
        if(n==0) return 2;
        else return x(n-1)*y(n-1);
}
#x(8);
1063433425
y(6);
5019630
```

Programme 1.23 –

Remarque Cet exemple de récursivité croisée est assez artificiel puisque le même calcul peut être effectué en une seule fois sur le couple $z = (x, y)$ en écrivant

```
static long[] z(int n){
       long[] z0={1,2},z1=new long[2];
       if(n==0) return z0;
       z0=z(n-1);
       z1[0]=z0[0]+z0[1];
       z1[1]=z0[0]*z0[1];
       return z1;
}
#z(6);
[13901, 5019630]
#z(8);
[1063433425, 505469074]
```

Programme 1.24 –

Qui plus est, au passage on a les valeurs des deux suites à la fois pour le même prix. Nous verrons plus tard des exemples moins artificiels de récursivité croisée, l'important est seulement ici de remarquer que cette récursivité croisée existe et est possible.

1.3.4 Ensembles définis récursivement

La méthodologie récursive ne s'applique pas seulement aux procédures ou fonctions, mais aussi à la définition même de certains objets. Intuitivement, la définition récursive d'un ensemble X consiste en la donnée explicite de certains

éléments de X et d'un moyen de construire de nouveaux éléments de X à partir d'éléments déjà connus. On peut ainsi construire des objets complexes à partir d'objets simples et d'opérations d'assemblages.

De manière plus formelle, donnons nous un ensemble E, une partie B de E et un ensemble K d'applications $\phi : E^{n(\phi)} \to E$.

Définition 1.3.1 *On dit qu'une partie A de E est K stable si*

$$\forall \phi \in K,\ \forall a_1, \ldots, a_{n(\phi)} \in A,\ \phi(a_1, \ldots, a_{n(\phi)}) \in A$$

Il est clair que toute intersection de parties K stables est encore K stable et que E lui-même est K stable. On en déduit que l'intersection de toutes les parties K stables contenant B est encore une partie K stable contenant B, et qu'elle est contenue dans toute partie K stable contenant B ; c'est donc le plus petit élément de l'ensemble des parties K stables contenant B.

Définition 1.3.2 *Soit E un ensemble, B une partie de E, K un ensemble d'opérations $\phi : E^{n(\phi)} \to E$. On appelle ensemble X défini récursivement par les deux relations*

- $B \subset X$
- $\phi \in K,\ x_1, \ldots, x_{n(\phi)} \in X \quad \Rightarrow \quad \phi(x_1, \ldots, x_{n(\phi)}) \in X$

la plus petite partie X de E qui est K stable et contient B (c'est aussi l'intersection de toutes les parties K stables contenant B).

Exemple Voici quelques exemples de définitions récursives (ou inductives)

- l'ensemble $\mathbb{N}$ des entiers naturels est défini récursivement par

$$0 \in \mathbb{N} \text{ et } \forall n \in \mathbb{N},\ n+1 \in \mathbb{N}$$

- l'ensemble $2\mathbb{N}$ des entiers pairs est défini récursivement par

$$0 \in \mathbb{N} \text{ et } \forall n \in \mathbb{N},\ n+2 \in \mathbb{N}$$

- Considérons l'ensemble E des mots formés avec les caractères accessibles sur votre clavier (un mot étant n'importe quel assemblage de caractères ne comportant pas d'espaces, que ce mot ait ou non un sens) ; on munit cet ensemble de l'opération de concaténation qui consiste simplement à coller bout à bout deux mots, opération que nous noterons $(x, y) \mapsto xy$; pour la commodité, nous considérerons qu'il existe également un mot vide (ne comportant aucune lettre) que nous noterons ϵ et qui est bien évidemment un élément neutre pour l'opération de concaténation. On peut alors considérer l'ensemble X des parenthésages bien formés défini inductivement par $\epsilon \in X$ et

$$x, y \in X \Rightarrow (x) \in X \text{ et } xy \in X$$

  Ainsi X est l'ensemble des mots comportant uniquement des parenthèses ouvrantes ou fermantes disposées de façon équilibrée ; par exemple

$$(()()(()()))() \in X$$

  alors que $())()(\notin X$.

La proposition suivante montre que, dans les définitions récursives d'ensembles, les objets construits sont obtenus à partir d'un nombre fini (mais variable) d'opérations à partir des objets de base.

Proposition 1.3.1 *Soit E, B et K comme ci dessus et soit X l'ensemble défini récursivement par la donnée de B et K. Alors tout élément de X peut s'obtenir à partir de la base B par application d'un nombre fini d'opérations $\phi \in K$; de manière plus formelle, si l'on pose $X_0 = B$ et pour $p \in \mathbb{N}$,*

$$X_{p+1} = X_p \bigcup \{\phi(x_1, \ldots, x_{n(\phi)}) \mid \phi \in K \text{ et } x_1, \ldots, x_{n(\phi)} \in X_p\}$$

alors $X = \bigcup_{p \in \mathbb{N}} X_p$.

Démonstration: soit $Y = \bigcup_{p\in\mathbb{N}} X_p$. Il est clair que $B = X_0 \subset Y$; d'autre part, Y est K stable car si $\phi \in K$ et $x_1, \ldots, x_{n(\phi)}$ sont dans Y, on a $x_1 \in X_{p_1}, \ldots, x_{n(\phi)} \in X_{p_{n(\phi)}}$ et comme la suite (X_p) est croissante, on a $x_1, \ldots, x_{n(\phi)} \in X_m$ où $m = \max\{p_1, \ldots, p_{n(\phi)}\}$; mais alors $\phi(x_1, \ldots, x_{n(\phi)}) \in X_{m+1} \subset Y$. Comme Y est K stable contenant B, on a nécessairement $X \subset Y$ (puisque X est l'intersection de toutes les parties K stables contenant B). Inversement, on a bien évidemment que si A est une partie K stable et si $X_p \subset A$, alors $X_{p+1} \subset A$. C'est vrai en particulier pour X et comme $B = X_0 \subset X$, on a par récurrence que $\forall p,\ X_p \subset X$, et donc $Y \subset X$. On en déduit que $X = Y$.

Beaucoup de démonstrations sur les ensembles définis récursivement utiliseront le théorème suivant

Théorème 1.3.1 (Principe d'induction structurelle) *Soit E un ensemble, B une partie de E, K un ensemble d'opérations $\phi : E^{n(\phi)} \to E$. On considère l'ensemble X défini récursivement par les deux relations*

- $B \subset X$
- $\phi \in K,\ x_1, \ldots, x_{n(\phi)} \in X \quad \Rightarrow \quad \phi(x_1, \ldots, x_{n(\phi)}) \in X$

Soit P un prédicat sur E. On suppose que $P(x)$ est vrai pour tout $x \in B$ et que pour tout $\phi \in K$ et pour tout $x_1, \ldots, x_{n(\phi)} \in E$,

$$P(x_1), \ldots, P(x_{n(\phi)}) \text{ sont vrais } \Rightarrow P(\phi(x_1, \ldots, x_{n(\phi)})) \text{ est vrai}$$

Alors $P(x)$ est vrai pour tout $x \in E$.

Démonstration: soit $Y = \{x \in E \mid P(x) \text{ est vrai}\}$. On sait que $B \subset Y$ et que Y est K stable puisque

$$\phi \in K,\ x_1, \ldots, x_{n(\phi)} \in Y \quad \Rightarrow \phi(x_1, \ldots, x_{n(\phi)})$$

On en déduit que Y contient l'ensemble X qui est l'intersection de toutes les parties de E qui sont K stables et contiennent B.

Exemple Revenons sur l'ensemble X des parenthésages bien formés constitué du sous ensemble de l'ensemble des mots défini récursivement par

- le mot vide ϵ est dans X
- si x et y sont des parenthésages bien formés, il en est de même de (x) et de xy

Nous allons montrer par le principe d'induction structurelle qu'un élément de X comporte autant de parenthèses ouvrantes que de parenthèses fermantes. C'est vrai pour le mot vide qui comporte 0 parenthèse ouvrante et 0 parenthèse fermante. De plus si x et y comportent autant de parenthèses ouvrantes que de parenthèses fermantes, il en est de même de (x) (qui a une parenthèse ouvrante et une parenthèse fermante de plus que x) et de xy (dont le nombre de parenthèses ouvrantes ou fermantes est la somme du nombre correspondant pour x et du nombre correspondant pour y). D'après le principe d'induction structurelle tout élément x de X vérifie la propriété.

Remarque La réciproque du résultat précédent est bien évidemment fausse. Le parenthésage ())()(a le même nombre de parenthèses ouvrantes et fermantes et cependant il n'est pas bien formé ; d'ailleurs le même principe d'induction structurelle montre qu'un parenthésage bien formé doit commencer par une parenthèse ouvrante et se terminer sur une parenthèse fermante. Ces conditions ne suffisent toujours pas à caractériser les parenthésages bien formés comme le montre l'exemple du parenthésage ())()(() qui n'est pas bien formé, bien qu'ayant le même nombre de parenthèses ouvrantes et fermantes, commençant par une parenthèse ouvrante et se terminant par une parenthèse fermante.

1.4 Récursivité et itération

1.4.1 Aperçus sur les appels de fonctions et de procédures

Imaginons le déroulement d'un programme comme le parcours d'un chemin ayant une origine et une extrémité. Un appel de sous programme (fonction ou procédure) est l'analogue d'un circuit greffé sur le chemin parcouru, circuit dont l'origine est confondu avec l'extrémité. Pour gérer ce circuit, le micro-processeur a besoin de retenir le point de départ du circuit ; ceci lui permet de retourner exactement au même point en fin de circuit. De plus l'exécution du sous

programme implique en général de créer de nouvelles variables tout en sauvegardant les valeurs des variables créées par la procédure appelante.

A cet effet, le micro-processeur gère une ou plusieurs piles (au sens d'une pile d'assiettes) dans lesquelles il stocke par empilement les adresses de retour des sous-programmes (l'endroit d'où le sous programme a été appelé et où il doit retourner en fin d'exécution) et les valeurs des différentes variables. A l'entrée dans le sous programme, le microprocesseur empile l'adresse de retour. Lors de l'exécution du sous programme il empile au fur et à mesure les nouvelles variables créées. Lors de la sortie du sous programme, il dépile les différentes variables créées par le sous programme et qui n'ont plus de signification en dehors de celui ci, puis il dépile l'adresse de retour qu'il utilise ensuite pour retourner à l'endroit voulu (c'est à dire le point d'appel).

Tout appel de sous programme (procédure ou fonction) consomme à la fois de la mémoire (pour stocker au moins l'adresse de retour et souvent des variables auxiliaires) et du temps (pour empiler et dépiler les adresses et les variables) [5].

1.4.2 Récursivité contre itération

Nous avons vu combien la programmation récursive était naturelle, élégante et puissante pour trouver des solutions à des problèmes. Mais cette programmation fait un usage intensif d'appels de sous programmes et comme nous l'avons vu par ailleurs tout appel de sous programme consomme à la fois du temps et de la mémoire. D'autre part, l'exemple du calcul de la suite de Fibonacci nous a démontré qu'un usage non réfléchi de la récursivité pouvait conduire à une très grande inefficacité : un temps de calcul de F_n de l'ordre de 2^n à la place du temps de calcul proportionnel à n que l'on peut obtenir avec un peu d'intelligence. Bien entendu ce n'est pas la récursivité elle-même qui est en cause, mais une utilisation non réfléchie de celle-ci.

Il peut parfois sembler intéressant pour des raisons d'efficacité de remplacer des appels récursifs de sous programmes par une simple itération obtenue par une boucle indexée ou une boucle avec test booléen. Nous allons d'abord étudier quelques exemples très simples avant d'essayer de faire une théorie de cette transformation.

En ce qui concerne le calcul des puissances, on peut remarquer que $a^n = \prod_{i=1}^{n} a$ ce qui conduit immédiatement à introduire une référence y sur $\prod_{i=1}^{j} a$ et donc à définir

```
1  static long puiss(long x,int n){
2        long y = 1;
3        for(int i=1; i <= n; i++)
4              y *= x;
5         return y;
6  }
7  #puiss(2,8);
8  256
```

Programme 1.25 –

Revenons sur le cas du calcul de la factorielle. Bien que la définition de la factorielle soit le plus souvent récursive, il est très facile d'en donner une définition itérative en écrivant $n! = \prod_{i=1}^{n} i$. Ceci nécessite le stockage dans une variable provisoire `prov` des valeurs successives du produit des i premiers nombres entiers. Les valeurs de cette variable doivent être actualisées au fur et à mesure. On obtient une nouvelle fonction factorielle que nous pouvons écrire en Java sous la forme :

```
1  static long fact(int n){
2        long prod=1;
3        for(int i=1; i <= n; i++)
4              prod = prod * i;
5         return prod;
6  }
7  #fact(8);
8  -40320
```

Programme 1.26 –

La même méthode s'applique à tout suite définie par récurrence par $x_0 = a$ et pour $n \geq 0$, $x_n = f(n, x_{n-1})$. Essayons d'appliquer la même méthode à la suite de Fibonacci définie par $F_0 = F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$. Pour cela posons $x_n = F_n$ et $y_n = F_{n+1}$. On a $x_0 = 1$, $y_0 = 1$; d'autre part $x_n = F_n = y_{n-1}$ et $y_n = F_{n+1} = F_n + F_{n-1} = y_{n-1} + x_{n-1}$.

5. Cependant certains compilateurs peuvent dans certains cas optimiser des appels de sous programmes aussi bien du point de vue de la mémoire que du temps d'exécution.

En définitive, si on pose $z_n = (x_n, y_n)$, la suite z_n est définie par récurrence par $z_n = f(z_{n-1})$ avec $f(x, y) = (y, x+y)$. Ceci permet d'obtenir une fonction itérative de calcul des nombres de Fibonacci en écrivant en Java :

```
1  static long fib2(int n){
2       long x=1, y=1, xPrec;
3          for(int i = 1;i <= n;i++){
4               xPrec= x;
5               x = y;
6               y = xPrec + y;
7             }
8          return x;
9  }
10 #fib2(35);
11 14930352, 24157817
```

Programme 1.27 –

Remarquons que cette fonction calcule à la fois F_n et F_{n+1}, dans un temps manifestement proportionnel à n, comme la fonction récursive avec stockage des résultats intermédiaires ; mais cette fonction itérative n'utilise pas de tableau de stockage des résultats intermédiaires (elle utilise uniquement une référence au lieu d'un tableau à n éléments). Ceci montre l'amélioration obtenue en transformant la fonction récursive en fonction itérative. Il faut prendre garde à ne pas perdre la valeur pointée par la référence x tant que l'on en a besoin, d'où l'utilisation d'une variable provisoire `xPrec` qui permet de stocker cette valeur.

Remarque Le lecteur attentif remarquera que nous avons calculé un terme de trop dans la suite et que l'on peut tenter d'écrire

```
1  static long fib2(int n){
2       long x=1, y=1, xPrec;
3          for(int i = 1;i < n;i++){
4               xPrec= x;
5               x = y;
6               y = xPrec + y;
7             }
8          return y;
9  }
10 #fib(35);
11 14930352
```

Programme 1.28 –

avec une petite économie de temps d'exécution. Il faut simplement remarquer que ce gain de temps a été obtenu au prix d'un programme fondamentalement incorrect puisque, lorsqu'il est appelé avec $n = 0$, il renvoie F_1 au lieu de F_0. Autrement dit l'utilisation du même schéma de programme avec une autre suite récurrente serait susceptible de renvoyer un résultat erroné. La correction de ce programme nécessite alors un test de la valeur de n pour renvoyer le terme d'indice 0 si $n = 0$, mais à ce moment, le gain en temps d'exécution disparaît. Comme quoi, à vouloir mieux faire ...

1.4.3 Récursivité terminale

Nous avons vu dans le paragraphe précédent la manière de procéder pour remplacer un appel récursif de fonction par une boucle indexée dans le calcul des termes d'une suite récurrente $x_n = f(n, x_{n-1})$. Le fait que l'ensemble sur lequel on travaille est l'ensemble $\mathbb{N}$ des entiers naturels peut masquer la méthode générale sous-jacente.

Donnons nous un ensemble E muni d'un ordre bien fondé $\preceq$ (pour lequel toute partie admet donc un élément minimal) ; soit B une partie de E et $\phi : E \setminus B \to E$ telle que $\forall x \in E,\ \phi(x) \prec x$. Soit f une fonction de B dans un ensemble E'. Il existe alors une unique application $F : E \to E'$ définie récursivement par

- $F(x) = f(x)$ si $x \in B$
- $F(x) = F(\phi(x))$ si $x \in E \setminus B$

Supposons que nous avons défini une procédure `action` qui pour chaque $x \in E$ effectue une certaine action sur l'environnement et soit `prog1` le sous programme récursif défini par

```
1  prog1(x){
2        action(x);
3        if( dansB(x) ) f(x)
4                    else prog1(phi(x));
```

```
5 }
```

Programme 1.29 –

Considérons d'autre part le sous programme `prog2` défini par :

```
1 prog2(x){
2     xRef  = x;
3         while( ! dansB(x)){b
4             action !xRef;
5             xRef = phi(xRef)
6         }
7         action(xRef);
8         f(xRef);
9 }
```

Programme 1.30 –

Proposition 1.4.1 *Les deux sous programmes* `prog1` *et* `prog2` *effectuent la même action et retournent le même résultat pour tout* $x \in E$.

> **Démonstration**: C'est clair pour $x \in B$, puisqu'alors aussi bien `prog1` que `prog2` effectuent l'action `action(x)` puis renvoient $f(x)$. Soit X l'ensemble des éléments vérifiant cette propriété et supposons $X \neq E$. Soit x_0 un élément minimal de $E \setminus X$. Le sous programme `prog1` appliqué à x_0 effectue l'action `action`(x_0), puis appelle le sous programme `prog1` sur $\phi(x_0)$. Le sous programme `prog2` appliqué à x_0 effectue l'action `action`(x_0) puis remplace x_0 par $\phi(x_0)$ avant d'effectuer la même action et de renvoyer le même résultat que le sous programme `prog2` sur $\phi(x_0)$. Mais, comme $\phi(x_0) \prec x_0$ et $x_0 = \min(E \setminus X)$, on a $\phi(x_0) \notin E \setminus X$, et donc $\phi(x_0) \in X$. On en déduit que `prog1` et `prog2` effectuent la même action et retournent le même résultat avec le paramètre x_0, donc $x_0 \in X$ contrairement à sa définition.

Le fait essentiel qui permet le remplacement des appels récursifs par une simple boucle booléenne est que ces appels récursifs sont la dernière instruction du sous programme ; de ce fait, toute l'action du sous programme étant déjà effectuée au moment de l'appel récursif, aucune sauvegarde de l'environnement n'est nécessaire. On dit dans ce cas que la récursivité est terminale [6].

1.4.4 Un exemple de dérécursivation non terminale

Nous venons de voir comment un appel récursif terminal pouvait être remplacé par une boucle conditionnelle. Il n'en va pas de même si l'appel récursif se situe avant que l'action du sous programme ne soit terminée, cas de la récursivité non terminale. Dans ce cas il est nécessaire de sauvegarder au minimum les valeurs successives des variables qui seront utilisées par le suite de l'action du sous programme. La dérécursivation est encore possible mais beaucoup plus complexe. Nous allons en voir un exemple dans le cas du calcul de fonctions définies par induction.

Soit f une fonction de B dans un ensemble E', g une application de $E \times E'$ dans E'. Il existe alors une unique application $F : E \to E'$ définie récursivement par

- $F(x) = f(x)$ si $x \in B$
- $F(x) = g(x, F(\phi(x)))$ si $x \in E \setminus B$

La première méthode de calcul repose sur un calcul récursif suivant le schéma de définition même de la fonction F que nous pouvons écrire en pseudo-Java sous la forme

```
1 F(x){
2       if(dansB(x))   f(x)
3       else g(x,F(phi(x)));
4 }
```

Programme 1.31 –

où la fonction `dansB` est un test booléen qui renvoie `true` si et seulement si x est dans B.

La deuxième méthode de calcul repose sur une boucle à test booléen. Elle suppose la gestion de deux références, une pile pour conserver les valeurs successives de x et un élément de E' pour conserver le résultat de l'appel de fonction.

6. Rien n'empêche en contre-partie un compilateur de remplacer une boucle par un appel récursif qu'il sera particulièrement apte à optimiser. La transformation marche dans les deux sens !

Dans un souci de lisibilité et d'efficacité, nous introduirons une troisième référence sur la variable x ; celle ci n'est pas indispensable puisqu'elle sert uniquement de relais pour conserver la valeur de tête de la liste. Nous commencerons par calculer les valeurs successives de $x, \phi(x), \ldots, \phi^k(x)$ jusqu'à ce que nous arrivions dans B, ce qui ne peut manquer de se produire du fait du caractère bien fondé de notre ordre ; ces valeurs sont stockées au fur et à mesure dans la pile p. Il suffit ensuite de remonter le calcul de F tout au long de la pile en utilisant et supprimant au fur et à mesure l'élément de tête de la pile. Nous pouvons donc écrire notre calcul itératif sous la forme (en pseudo-Java) :

```
1  static Y F(X x){
2         lapile=new Pile<X>();
3         Y y;
4          while(!dansB(x)){
5                x = phi(x);
6                lapile.push(x);
7           }
8           y = f(x); /* x est dans B */
9           pile.pop();
10          while(!lapile.empty()){
11               y = g(lapile.pop(),y);
12          }
13          return y;
14 }
```

Programme 1.32 –

Faut-il procéder à un tel remplacement d'appels récursifs par une (ou plusieurs) boucles booléennes ? Notons tout d'abord la différence de simplicité et de lisibilité entre les deux formes. La version récursive de la fonction tient en trois lignes et colle au plus près à la définition mathématique, la version itérative demande une dizaine de lignes, et pas mal de réflexion, pour se convaincre qu'elle effectue le même calcul.

En dehors de l'exercice académique que représente la dérécursivation d'un sous programme, exercice qui oblige à se plonger plus profondément dans la logique de la machine, le seul intérêt que l'on peut voir à ce type de transformation est d'améliorer un sous programme qui est critique soit du point de vue de la vitesse d'exécution (notre exemple n'étant certainement pas grandement amélioré avec ses deux boucles booléennes et une gestion des listes qui échappe en grande partie à notre contrôle), soit du point de vue de la mémoire, en gérant avec intelligence ce qui doit être sauvegardé (là encore notre exemple n'est guère significatif puisqu'il oblige à créer une référence sur une liste qui croît au fur et à mesure de nos appels à ϕ).

De ces deux points de vue (vitesse d'exécution et gestion de la mémoire), peut-être vaut-il mieux utiliser de manière plus intelligente et moins brutale la récursivité, en introduisant par exemple des fonctions auxiliaires, plutôt que de se lancer dans une dérécursivation coûteuse en temps de travail et aux résultats plus ou moins incertains.

1.5 Exemples de tris

1.5.1 Introduction aux tris

Le tri est une opération fondamentale de traitement des données. Il constitue le plus souvent une opération préalable à un traitement efficace de ces données. Pour se convaincre de l'utilité d'un tri, imaginez que les mots dans votre dictionnaire préféré soient mis en vrac au lieu d'être triés par ordre alphabétique et que vous ayiez à y chercher le mot *existentiel* ; la seule solution serait alors de parcourir tout le dictionnaire jusqu'à trouver le mot en question. De même, la recherche d'un nom sur une liste d'admis à un concours est grandement facilitée par un tri alphabétique de la liste des reçus. C'est ainsi que de nombreux algorithmes informatiques supposent que des données ont été préalablement triées.

Dans le cas de gros volumes de données comme on en trouve très fréquemment dans l'informatique courante, le choix d'une méthode de tri est cruciale. Il se trouve cependant qu'il n'existe pas de méthode de tri optimale universelle. Toute méthode de tri utilise quelques opérations fondamentales comme la comparaison entre deux éléments et l'échange de deux éléments. Suivant le type de données considéré le coût de la comparaison et celui de l'échange peuvent être très différents et, dans un souci d'efficacité, on peut être amené à privilégier un petit nombre de comparaisons ou un petit nombre d'échanges.

De plus la façon dont on accède aux données influe grandement sur le choix d'une méthode de tri :

– directement s'il s'agit de données stockées dans un tableau en mémoire centrale

– séquentiellement s'il s'agit de données stockées dans une liste en mémoire centrale ou sur une mémoire externe (disquette ou disque dur)

Dans la suite de ce chapitre, nous supposerons que nous disposons de N données stockées dans un tableau $a[0], \ldots, a[N+1]$ dans les éléments d'indice 1 à N (les éléments d'indice 0 et $N+1$ seront souvent réservés à un usage particulier). Pour simplifier , nous supposerons que les données stockées sont des entiers : il est immédiat d'adapter les méthodes au cas d'un type quelconque muni d'une relation d'ordre total. Nous utiliserons systématiquement une procédure `echange` qui reçoit en paramètres un tableau et deux indices et procède à l'échange des deux éléments d'indices correspondants du tableau. En Java, on peut écrire une telle procédure sous la forme :

```
static void echange(int[] tableau, int i, int  j){
       int temp = tableau[i];
       tableau[i] =  tableau[j];
       tableau[j] =  temp;
}
```

Programme 1.33 –

la variable `temp` servant à stocker temporairement la valeur de `tableau[i]`.

A titre d'application, nous utiliserons un tableau de taille $n+2$ initialisé avec des nombres entiers choisis au hasard parmi $1 \ldots$ `lim`. Pour cela nous construisons une fonction Java `hasard_vect` ; cette fonction utilise une fonction de la bibliothèque mathématique `Math.random()` qui retourne un nombre réel au hasard compris entre 0 et 1 que l'on va transformer en un entier compris entre 1 et `lim`.

L'intérêt du paramètre `lim` est de forcer des répétitions à l'intérieur du tableau. Nous obtenons la fonction :

```
static int[] hasard_vect(int  n, int lim){
    int[] v = new int[n+2];
    for(int i = 1; i <= n; i++){
       v[i] =  (int) (Math.random()*lim)+1;
    }
    return v;
}
```

Programme 1.34 –

Pour la commodité, nous utiliserons également une procédure d'affichage des tableaux d'entiers qui enlève le premier et le dernier élément du tableau :

```
static void affiche_tableau(int[] a){
    int n = a.length;
       System.out.print( "[" );
       for(int i = 1; i <= n-3; i++)
          System.out.print(a[i]+";");
       System.out.println(a[n-2]+"]");
}
#affiche_tableau( v );
[0;0;0;0;0;4;4;4;4;6;6;6;6;8;8;8;0]
```

Programme 1.35 –

1.5.2 Le tri par sélection

C'est l'une des méthodes de tri les plus simples : on cherche d'abord l'élément le plus petit dans le tableau et on l'échange avec le premier élément du tableau ; ensuite on cherche l'élément immédiatement supérieur et on l'échange avec le deuxième élément du tableau et ainsi de suite jusqu'à épuisement du tableau ; ce tri procède donc par sélection du plus petit élément parmi $a[i], \ldots, a[N]$ puis échange ce plus petit élément avec $a[i]$.

Une telle procédure peut s'écrire en Java.

```
static void tri_selection(int[] a){
     int N = (a.length)-2, min = 0;
        for(int i = 1; i <= N-1; i++){
            min  =  i;
            for(int j = i+1; j <= N; j++){
              if( a[j]<a[min] ) min  =  j;
            }
            echange(a, i, min);
          };
 }
```

Programme 1.36 –

et voici son application à un exemple avec les modifications successives du tableau

```
[|11;9;14;9;5;0;13;0;13;13;1;14;10;11;11|]
[|0;9;14;9;5;11;13;0;13;13;1;14;10;11;11|]
[|0;0;14;9;5;11;13;9;13;13;1;14;10;11;11|]
[|0;0;1;9;5;11;13;9;13;13;14;14;10;11;11|]
[|0;0;1;5;9;11;13;9;13;13;14;14;10;11;11|]
[|0;0;1;5;9;11;13;9;13;13;14;14;10;11;11|]
[|0;0;1;5;9;9;13;11;13;13;14;14;10;11;11|]
[|0;0;1;5;9;9;10;11;13;13;14;14;13;11;11|]
[|0;0;1;5;9;9;10;11;13;13;14;14;13;11;11|]
[|0;0;1;5;9;9;10;11;11;13;14;14;13;13;11|]
[|0;0;1;5;9;9;10;11;11;11;14;14;13;13;13|]
[|0;0;1;5;9;9;10;11;11;11;13;14;14;13;13|]
[|0;0;1;5;9;9;10;11;11;11;13;13;14;14;13|]
[|0;0;1;5;9;9;10;11;11;11;13;13;13;14;14|]
[|0;0;1;5;9;9;10;11;11;11;13;13;13;14;14|]
- : unit = ()
```

Programme 1.37 –

Théorème 1.5.1 *Le tri par sélection trie un tableau de N éléments avec un nombre de comparaisons équivalent à $\frac{N^2}{2}$ et un nombre d'échanges égal à $N-1$.*

Démonstration: Il faut tout d'abord démontrer que le tri par sélection trie effectivement le tableau. La terminaison de l'algorithme est garantie par le fait qu'il s'agit de boucles indexées. Il nous reste à exhiber deux invariants de boucles, l'un pour la boucle interne indexée par j, l'autre pour la boucle externe indexée par i. En ce qui concerne la boucle interne, un invariant est

$P(j)$: après exécution de l'itération d'indice $j \in [i+1, N]$, $a[\texttt{min}]$ est le minimum de $a[i], a[i+1], \ldots, a[j]$

C'est évidemment vrai après la première itération, et si la propriété est vraie à la sortie du corps pour $j-1$, on a deux possibilités

- soit $a[j] \geq a[\texttt{min}]$, alors la boucle ne fait rien et on a après exécution

$$\min(a[i], a[i+1], \ldots, a[j]) = \min(a[\texttt{min}], a[j]) = a[\texttt{min}]$$

- soit $a[j] < a[\texttt{min}]$, alors la boucle affecte j à `min` et on a après exécution

$$\begin{aligned} \min(a[i], a[i+1], \ldots, a[j]) &= \min\left(\min(a[i], a[i+1], \ldots, a[j-1]), a[j]\right) \\ &= a[j] = a[\texttt{min}] \end{aligned}$$

ce qui montre que $P(j)$ est vrai.

Après exécution de la boucle interne, on en déduit que $a[\texttt{min}]$ est le minimum de $a[i], a[i+1], \ldots, a[N]$. L'invariant de la boucle externe indexée par i sera donc

$P'(i)$: après l'itération d'indice i, on a

$$a[1] \leq a[2] \leq \ldots \leq a[i] = \min(a[i], a[i+1], \ldots, a[N])$$

Après exécution de la boucle externe, on a donc

$$a[1] \leq a[2] \leq \ldots \leq a[N-1] = \min(a[N-1], a[N])$$

ce qui montre que le tableau est trié.

Dans le tri par sélection de N éléments, on effectue une comparaison par exécution du corps de la boucle interne, soit $(N-1)+(N-2)+\ldots+1 = \frac{N(N-1)}{2}$ comparaisons et un échange par exécution du corps de la boucle externe, soit $N-1$ échanges. Le tri par sélection de N éléments demande donc un nombre de comparaisons qui est équivalent à $\frac{N^2}{2}$ et un nombre d'échanges qui est équivalent à N.

1.5.3 Le tri par insertion

Le tri par insertion est similaire au tri du joueur de cartes qui insère les cartes une à une dans son jeu de manière à ce qu'à tout instant le jeu soit trié. Pour cela il parcourt les cartes déjà insérées en partant de la plus grande et insère la carte considérée juste après la première carte immédiatement inférieure qu'il rencontre au cours de son parcours (en début de jeu s'il n'en a rencontré aucune). On peut formaliser cet algorithme sous la forme

pour chaque i, insérer l'élément $a[i]$ parmi $a[1], \dots a[i-1]$

où la procédure d'insertion peut être définie par

prendre le premier j, en descendant à partir de i, tel que $a[j-1] \leq a[i]$, en décalant au fur et à mesure les éléments rencontrés vers la droite, puis insérer $a[i]$ à la j-ième place.

ce qui peut s'écrire en Java :

```
1  static void tri_insertion(int[] a){
2       int N = (a.length)-2,v,j;
3          for(int i = 2; i <= N; i++){
4               v = a[i] ;
5               j = i;
6                 while( (a[j-1]>v) ){
7                    a[j]  =  a[j-1];
8                    j  =  j -1;
9                 }
10                a[j] =  v;
11         }
12 }
```

Programme 1.38 –

En fait ce sous programme contient une erreur évidente : si $a[i]$ est le plus petit élément de $a[1], \dots, a[i-1]$, la boucle interne va se poursuivre jusqu'à $j = 1$, puis va provoquer un débordement vers la gauche du tableau avec la comparaison suivante de $a[0]$ à v. La première manière de corriger cette erreur est d'éviter un débordement vers la gauche en testant si $j > 1$. Cela peut se faire de la manière suivante

```
1  static void tri_insertion(int[] a){
2       int N = (a.length)-2,v,j;
3          for(int i = 2; i <= N; i++){
4               v = a[i] ;
5               j = i;
6                 while( j>1 && (a[j-1]>v) ){
7                    a[j]  =  a[j-1];
8                    j  =  j -1;
9                 }
10                a[j] =  v;
11         }
12 }
```

Programme 1.39 –

Cette précaution double le nombre de tests à effectuer alors que la situation décrite est relativement exceptionnelle [7]. Une autre solution est d'utiliser la place réservée $a[0]$ et de mettre à cet endroit une sentinelle, c'est à dire un élément dont on est certain qu'il va arrêter la boucle. Il suffit pour cela qu'il soit inférieur à tout élément rencontré. Tout naturellement, si j devient égal à 1, on aura $a[j-1] \leq v$ et la boucle s'arrêtera d'elle même. Si nous travaillons par exemple avec des entiers positifs, la valeur 0 que nous avons choisie pour $a[0]$ convient parfaitement, si bien que la première version du programme de tri par insertion, qui était erronée si l'on ne tenait pas compte de $a[0]$ devient maintenant correcte si $a[0] = 0$ et si tous les $a[i]$ sont des entiers positifs. Voici un exemple d'exécution avec affichage des tableaux successifs

```
1  [|9;11;14;9;5;0;13;0;13;13;1;14;10;11;11|]
2  [|9;11;14;9;5;0;13;0;13;13;1;14;10;11;11|]
3  [|9;9;11;14;5;0;13;0;13;13;1;14;10;11;11|]
4  [|5;9;9;11;14;0;13;0;13;13;1;14;10;11;11|]
```

7. Il faut quand même remarquer que les tests supplémentaires ne sont pas de la même nature que les tests portant sur les éléments du tableau, puisqu'ils consistent simplement à comparer un entier à 1, ce qui est extrêmement rapide

```
[|0;5;9;9;11;14;13;0;13;13;1;14;10;11;11|]
[|0;5;9;9;11;13;14;0;13;13;1;14;10;11;11|]
[|0;0;5;9;9;11;13;14;13;13;1;14;10;11;11|]
[|0;0;5;9;9;11;13;13;14;13;1;14;10;11;11|]
[|0;0;5;9;9;11;13;13;13;14;1;14;10;11;11|]
[|0;0;1;5;9;9;11;13;13;13;14;14;10;11;11|]
[|0;0;1;5;9;9;11;13;13;13;14;14;10;11;11|]
[|0;0;1;5;9;9;10;11;13;13;13;14;14;11;11|]
[|0;0;1;5;9;9;10;11;11;13;13;13;14;14;11|]
[|0;0;1;5;9;9;10;11;11;11;13;13;13;14;14|]
```

Programme 1.40 –

Remarquons cependant que la méthode de la sentinelle est rarement praticable, puisqu'elle exige que l'on connaisse a priori un élément plus petit que tous les autres ; de plus elle oblige à travailler sur un tableau ayant un élément de plus que le tableau initial, d'où la plupart du temps une recopie du tableau dans un tableau plus grand. Le gain obtenu n'est donc pas franchement évident.

Théorème 1.5.2 *Le tri par insertion trie un tableau de N éléments avec un nombre de comparaisons et un nombre d'affectations équivalents à*

- $\frac{N^2}{2}$ *dans le cas le plus défavorable*
- $\frac{N^2}{4}$ *dans le cas moyen*

Démonstration: il faut tout d'abord démontrer que le tri par insertion trie effectivement le tableau. La terminaison de la boucle extérieure est garantie par le fait qu'il s'agit d'une boucle indexée et la terminaison de la boucle intérieure par la décroissance stricte de l'indice j qui doit de toute façon rester positif puisque $a_0 \leq v$.

Il nous reste à exhiber des invariants des deux boucles. En ce qui concerne la boucle booléenne intérieure, il est clair qu'à l'entrée dans le corps de la boucle, on a $v < a_j \leq a_{j+1} \leq \ldots \leq a_i$ et que lors de la sortie de la boucle on a en plus $v \geq a_{j-1}$. Après la sortie de la boucle booléenne, on a donc $a_0 \leq a_1 \leq \ldots \leq a_{j-1} \leq v < a_{j+1} \leq \ldots \leq a_i$ et après l'affectation de v à a_j, on a donc $a_0 \leq a_1 \leq \ldots \leq a_i$ ce qui fournit un invariant pour la boucle extérieure (indexée). Lorsque i atteint la valeur n à la sortie de la boucle extérieure, on a donc $a_0 \leq a_1 \leq \ldots \leq a_n$ ce qui montre que le tableau est trié.

Dans le cas le plus défavorable (quand le tableau est à l'origine trié en ordre décroissant), chaque élément a_i remonte toute la chaîne jusqu'à arriver en première position et on effectue donc $N+(N-1)+\ldots+2+1 = \frac{N(N+1)}{2} \sim \frac{N^2}{2}$ comparaisons et autant d'affectations.

Dans le cas moyen (cas d'un tableau aléatoire), on peut s'attendre à ce que a_i vienne s'insérer environ à la moitié de la liste, ce qui divise par deux le nombre de comparaisons et d'affectations.

1.5.4 Le tri à bulles

Il s'agit d'une méthode de tri dont l'intérêt est plutôt anecdotique, ce tri n'étant intéressant ni du point de vue du nombre de comparaisons, ni du point de vue du nombre d'échanges. L'idée est de parcourir le tableau et d'échanger deux éléments consécutifs dès qu'ils sont dans l'ordre contraire de l'ordre souhaité. On itère ce parcours tant qu'un échange a été effectué au cours du balayage. Lorsque plus aucun échange n'est effectué, c'est que le tableau est trié. On peut écrire une procédure Java sous la forme d'une fonction `parcours` qui retourne `true` si un échange a été effectué au cours du parcours et d'une fonction de tri qui se contente d'appeler la fonction de parcours tant que celle ci retourne qu'un échange a été effectué.

```
static void tri_bulles(int[] a){
    boolean y_a_echange = true;
    int N = (a.length)-2;
     while(y_a_echange){
             for(int i = 1; i <= N-1; i++){
               if( a[i]>a[i+1] )
                  {
                    echange(a, i, i+1);
                    y_a_echange  =  true;
                  }
              }
         }
 }
```

Programme 1.41 –

Remarque Le nom de tri à bulles provient de ce que les éléments les plus petits remontent au fur et à mesure des échanges vers l'origine du tableau, de la même façon que les bulles d'un élément plus léger remontent vers la surface d'un élément plus lourd.

La présence de la boucle booléenne montre que le nombre de comparaisons et d'échanges dépend de manière essentielle de la distribution des éléments. Dans le cas le plus favorable d'un tableau déjà trié, il y aura un seul parcours et donc $N-1$ comparaisons et aucun échange. Au contraire, dans le cas d'un tableau ordonné par ordre décroissant, il y aura $(N-1)^2$ comparaisons et autant d'échanges.

En fait, on peut simplifier la procédure de tri à bulles en faisant la remarque suivante : lors du premier parcours du tableau, le plus grand élément du tableau va s'échanger avec tous les éléments consécutifs jusqu'à parvenir à la dernière position à la fin du parcours. A partir de là, cet élément ne va plus intervenir (et donc n'a plus à être considéré dans les comparaisons) ; au cours du second parcours, c'est l'élément suivant en taille qui va s'enfoncer dans les profondeurs du tableau, jusqu'à parvenir à l'avant dernière position, et ainsi de suite. Ceci permet d'écrire le tri à bulles sous la forme

```
1 static void tri_bulles(int[] a){
2     int N = (a.length)-2;
3         for(int i = N; i >=  1; i--)
4             for(int j = 2; j <= i; j++){
5                 if( a[j-1]>a[j] )
6                     echange(a, j-1, j);
7         }
8 }
```

Programme 1.42 –

Sous cette forme on constate que le tri à bulles procède de la même façon que le tri par sélection, mais en commençant par les éléments les plus grands. Il opère le même nombre de comparaisons $\frac{N(N-1)}{2}$ mais un nombre beaucoup plus grands d'échanges (encore $\frac{N(N-1)}{2}$ dans le cas le plus défavorable d'un tableau trié *à l'envers*, au lieu des N échanges du tri par sélection). En conclusion, le tri à bulles est de peu d'intérêt et on peut toujours lui préférer un tri par sélection.

1.5.5 Méthodes performantes de tri

Nous verrons dans les autres chapitres des méthodes de tri de performances supérieures (tout au moins pour les gros tableaux) :

- tri rapide, en anglais *quicksort* (chapitre *Diviser pour régner*)
- tri fusion, en anglais *mergesort* (chapitre *Diviser pour régner*)
- tri en tas, en anglais *heapsort* (chapitre *Arbres*)

Ceci ne veut pas dire que le tri par sélection ou le tri par insertion soient à négliger. D'une part ils sont faciles à programmer et, pour des tableaux de petite taille, sont parfois plus rapides que les méthodes plus sophistiquées. Signalons également une variante plus performante du tri par insertion appelée le *Shell sort* qui procède en triant les éléments de p en p en utilisant une suite décroissante bien choisie d'entiers p : sa complexité est en $N^{3/2}$ dans le cas le plus défavorable et n'est pas connue dans le cas moyen.

Chapitre 2

Diviser pour régner

2.1 Quelques idées sur la complexité

2.1.1 Notion de taille des données

Rappelons que les différentes mémoires sont des éléments essentiels des ordinateurs. On distingue en général deux sortes de mémoires

- les mémoires internes à accès extrêmement rapide, elles mêmes subdivisées en
  - mémoire morte (ou ROM pour Read Only Memory) : c'est une mémoire figée, non modifiable et permanente qui contient l'ensemble des programmes de base de la machine
  - mémoire permanente (parfois appelée PRAM pour Permanent Random Access Memory) : c'est une mémoire à la fois modifiable et permanente, de petite taille et qui contient quelques éléments de configuration de la machine
  - mémoire vive (ou RAM pour Random Access Memory) : c'est une mémoire modifiable mais volatile (elle est effacée à l'extinction de la machine) qui est à la disposition des logiciels et de l'utilisateur
- les mémoires externes, elles mêmes subdivisées en
  - disquettes de faible capacité, servant à stocker certains logiciels ou de petits volumes de données sous une forme permanente et modifiable, facilement transportables mais d'accès lent
  - disques durs de plus grandes capacités (l'équivalent de 15 à 2000 disquettes), servant à stocker l'ensemble des logiciels et de gros volumes de données sous une forme permanente et modifiable, d'accès rapide mais peu ou pas transportables
  - CD-ROM de grandes capacités (l'équivalent de 400 disquettes sur un support semblable au disque compact CD), servant à stocker de très gros volumes de données sous une forme non modifiable, d'accès lent et très facilement transportables

Toutes les données (nombres, caractères, noms, adresses, etc.) manipulées doivent être stockées dans l'ordinateur. A cet effet, toutes ces mémoires sont divisées en petites cases appelées *octets* ; ces octets peuvent stocker une suite de 8 chiffres égaux à 0 ou à 1 (des *bits*), soit 256 éléments différents. Ces octets sont numérotés dans chaque mémoire de 0 jusqu'à la capacité de la mémoire (moins un). L'ordinateur se charge d'effectuer une traduction entre les données entrées et un ensemble d'octets, c'est l'opération de *codage*. Typiquement

- les caractères typographiques sont codés sur un octet (puisqu'il y en a moins de 256)
- un mot est stocké sous forme d'une suite d'octets, chacun d'entre eux correspondant à un caractère
- les entiers courts, compris entre -32768 et 32767, sont codés sur deux octets (car $2 \times 32768 = 256^2$)
- les entiers longs compris entre -2 147 483 648 et 2 147 483 647 sont codés sur quatre octets (car $2 \times 2147483648 = 256^4$)
- les réels en virgule flottante sont codés sous forme mantisse m et exposant e (soit $x = m10^e$ avec $m \in [0, 1[$), la mantisse étant elle même codée à partir d'un entier

A partir de là, on voit que toute donnée (la liste des élèves d'une classe et leurs notes au cours de l'année, l'annuaire du téléphone) a une taille que l'on peut exprimer en octets. Cette taille dépend de la méthode de codage utilisée (par exemple MS-DOS code les caractères sur un octet alors que Windows les code sur deux octets) mais obéit à quelques

règles simples

- le rapport des tailles des données entre deux systèmes différents est un nombre assez proche de 1 (typiquement entre 1/8 et 8)
- cette taille est proportionnelle au nombre N d'objets élémentaires dans ces données

Nous allons dans la suite de cette section étudier le rapport entre la taille des données et le temps d'exécution d'un traitement concernant ces données. Il nous suffit donc d'étudier le rapport entre le nombre N d'objets élémentaires à traiter et le temps d'exécution.

2.1.2 Types de complexité

Rappelons que si l'on dispose de deux fonctions f et g définies sur $\mathbb{N}$ et à valeurs réelles positives, on dit que $f(n) = O(g(n))$ s'il existe une constante $K \geq 0$ telle que $\forall n \in \mathbb{N},\ f(n) \leq Kg(n)$.

Considérons une donnée constituée d'un certain nombre d' objets et un certain traitement concernant ces données. Soit $T(N)$ le temps d'exécution de ce traitement lorsqu'il concerne N objets. Ce temps d'exécution dépend évidemment de la machine utilisée et du codage des données, mais il possède certains invariants : en particulier, si l'on utilise deux machines de performances différentes, les temps $T_1(N)$ et $T_2(N)$ sont en gros proportionnels (le facteur de proportionnalité mesurant justement la différence de rapidité des deux machines).

Ceci va nous permettre de différencier certains types d'algorithmes, du point de vue de leur complexité, c'est à dire de la rapidité de croissance de $T(N)$ quand N devient grand. On dit qu'un algorithme de traitement de données est à complexité

- logarithmique si $T(N) = O(\log_2 N)$ (ou par extension $T(N) = O((\log_2 N)^k)$)
- linéaire si $T(N) = O(N)$
- polynomiale si $T(N) = O(N^k)$ pour un certain k
- exponentielle si $T(N) = O(\rho^n)$ pour un certain $\rho > 1$
- hyperexponentielle si pour tout $\rho > 1$, $\lim \frac{T(N)}{\rho^n} = +\infty$ (par exemple $T(N) = N!$)

Ce temps d'exécution dépend bien entendu du nombre d'opérations élémentaires effectuées sur ces données et de la plus ou moins grande rapidité de ces opérations élémentaires. Parmi ces opérations élémentaires on peut citer

- les opérations algébriques sur les nombres entiers (addition, soustraction, multiplication, division) sachant qu'une multiplication ou une division est souvent 100 fois plus lente qu'une addition ou qu'une soustraction
- les opérations de comparaison (égalités ou inégalités) entre nombres ou entre caractères
- les opérations de déplacement en mémoire d'un objet ; la rapidité dépend beaucoup de la taille de l'objet, mais aussi du type de mémoire utilisée : accès rapide ou accès lent.

Dans l'évaluation de ce temps d'exécution, on sera amené à préciser quelles sont les opérations élémentaires à prendre en considération : si l'on effectue à peu près autant d'additions que de multiplications, on peut négliger les additions qui sont beaucoup plus rapides et ne compter que les multiplications, les échanges en mémoire interne peuvent souvent être négligés vis à vis d'échanges entre la mémoire interne et la mémoire externe, etc.

2.1.3 Comparaison de temps d'exécution

Prenons une machine (très rapide) capable d'accomplir 10^8 opérations par seconde, et imaginons des algorithmes qui effectuent un traitement donné dans un temps $T(N)$. Nous donnons ci dessous un tableau des temps de traitement que l'on peut espérer pour N éléments suivant la complexité de l'algorithme (où s, m, h, j, a désignent respectivement la seconde, la minute, l'heure, le jour, l'année et u l'âge estimé de l'univers, et où les nombres totalement inimaginables n'ont pas été affichés)

	$\log_2 N$	N	$N \log N$	N^2	N^4	2^N	4^N	$N!$
$N = 5$	$2.10^{-8}s$	$5.10^{-8}s$	$8.10^{-8}s$	$3.10^{-7}s$	$6.10^{-6}s$	$3.10^{-7}s$	$1.10^{-5}s$	$1.10^{-6}s$
$N = 10$	$2.10^{-8}s$	$1.10^{-7}s$	$2.10^{-7}s$	$1.10^{-6}s$	$1.10^{-4}s$	$1.10^{-5}s$	$1.10^{-2}s$	$4.10^{-2}s$
$N = 15$	$3.10^{-8}s$	$2.10^{-7}s$	$4.10^{-7}s$	$2.10^{-6}s$	$5.10^{-4}s$	$3.10^{-4}s$	$1.10^{1}s$	$2h$
$N = 20$	$3.10^{-8}s$	$2.10^{-7}s$	$6.10^{-7}s$	$4.10^{-6}s$	$2.10^{-3}s$	$1.10^{-2}s$	$2h$	$6.10^{2}a$
$N = 30$	$3.10^{-8}s$	$3.10^{-7}s$	$1.10^{-6}s$	$9.10^{-6}s$	$8.10^{-3}s$	$1.10^{1}s$	$3.10^{2}a$	$6.10^{6}u$
$N = 50$	$4.10^{-8}s$	$5.10^{-7}s$	$2.10^{-6}s$	$3.10^{-5}s$	$6.10^{-2}s$	$1.10^{2}j$	$2.10^{4}u$	$6.10^{38}u$
$N = 100$	$5.10^{-8}s$	$1.10^{-6}s$	$5.10^{-6}s$	$1.10^{-4}s$	$1s$	$2.10^{4}u$	$4.10^{34}u$	
$N = 500$	$6.10^{-8}s$	$5.10^{-6}s$	$3.10^{-5}s$	$3.10^{-3}s$	$1.10^{1}m$			
$N = 1000$	$7.10^{-8}s$	$1.10^{-5}s$	$7.10^{-5}s$	$1.10^{-2}s$	$2.10^{0}h$			
$N = 5000$	$9.10^{-8}s$	$5.10^{-5}s$	$4.10^{-4}s$	$3.10^{-1}s$	$7.10^{1}j$			
$N = 10000$	$9.10^{-8}s$	$1.10^{-4}s$	$9.10^{-4}s$	$1s$	$3a$			
$N = 50000$	$1.10^{-7}s$	$5.10^{-4}s$	$5.10^{-3}s$	$3.10^{1}s$	$2.10^{3}a$			

On constate que les algorithmes hyperexponentiels deviennent impraticables dès que N dépasse une dizaine, que les algorithmes exponentiels dépassent les limites acceptables dès que N atteint la trentaine. Ces deux types d'algorithmes sont donc incapables de traiter raisonnablement même de petits volumes de données.

On constate également, que pour rechercher un nom parmi 50000 personnes, entre le temps d'exécution d'une recherche linéaire en $O(N)$ et celui d'une recherche dichotomique en $O(\log_2 N)$, il existe un facteur 1000. De même, pour trier 10000 nombres réels (ce qui est courant en synthèse d'images avec une grille 100×100), on gagne un facteur 1000 en utilisant un tri en $O(N \log_2 N)$ comme le tri fusion, par rapport à un tri en $O(N^2)$ comme le tri par insertion : sur la même machine, l'un des tris pourra prendre 4 secondes alors que l'autre tri demandera une heure !

Remarque Il convient de nuancer un peu la dernière argumentation. Les algorithmes performants sont souvent complexes à écrire et cette complexité peut se ressentir sur les temps d'exécution pour de petits volumes de données. Autrement dit, pour un volume de données de l'ordre de quelques unités, il arrive souvent que les algorithmes *bêtes* soient plus rapides que les algorithmes *intelligents*. Entre un algorithme *bête* en $N^2/2$ et un algorithme *intelligent* en $3N \log_2 N$, la différence ne se fera sentir en faveur du second qu'à partir de $N = 30$. Il peut parfois être nécessaire d'adapter l'algorithme utilisé au volume des données à traiter. N'oublions pas également que les algorithmes sophistiqués utilisent souvent plus de mémoire, ce qui peut être un inconvénient même si celle-ci n'est plus une denrée rare ; il faut savoir établir un compromis entre rapidité et occupation mémoire.

2.2 Principes généraux de "diviser pour régner"

2.2.1 Aperçus philosophiques

La méthode de programmation généralement désignée par *diviser pour régner* procède d'une philosophie dont nous allons tenter de donner une idée. Supposons que nous ayions à traiter un problème algorithmique concernant un objet de *taille* N (cet objet pouvant être lui même un ensemble d'objets). On peut procéder en trois étapes

- Partitionner l'objet de taille N en p *sous objets* de taille approximative N/p d'une manière adéquate
- Résoudre notre problème pour chacun des sous objets
- Fusionner les résultats obtenus pour obtenir une solution du problème pour l'objet initial.

Cette méthode est bien connue des adeptes de Machiavel : pour détruire un ensemble d'ennemis, les diviser en sous ensembles que l'on écrase un à un, la fusion des résultats ne posant alors guère de difficultés.

Beaucoup d'algorithmes performants se fondent sur ces principes, très souvent avec $p = 2$. C'est le seul cas que nous examinerons par la suite, mais le lecteur n'aura pas de mal à adapter raisonnements et méthodes à des cas plus complexes. Dans ce cas la méthode devient donc

- Partitionner l'objet de taille N en 2 *sous objets* de taille approximative $N/2$ d'une manière adéquate
- Résoudre notre problème pour les deux sous objets
- Fusionner les résultats obtenus pour obtenir une solution du problème pour l'objet initial

2.2.2 Evaluations de temps de calcul

Remarque En classe de Mathématiques Supérieures, on pourra sauter les démonstrations de cette section et se contenter de retenir les énoncés des deux théorèmes ci-dessous. Ces résultats seront repris dans un cadre un peu plus général en classe de Mathématiques Spéciales.

Appelons $T(N)$ le temps nécessaire pour traiter un objet de taille N, $P(N)$ le temps nécessaire pour partitionner l'objet de taille N en 2 *sous-objets* de taille approximative $N/2$ d'une manière adéquate et $F(N)$ le temps nécessaire pour fusionner deux solutions de taille $N/2$ en une solution de taille N. On voit que la fonction $T(N)$ va vérifier

$$T(N) = P(N) + 2T(\frac{N}{2}) + F(N)$$

Supposons en particulier que N est de la forme 2^k et posons $t(k) = T(2^k)$. On aura alors

$$t(k) = P(2^k) + 2t(k-1) + F(2^k)$$

Nous nous intéresserons tout particulièrement au cas où P et F sont des fonctions polynomiales de N. Quitte à les majorer, on peut alors supposer que $P(N) = \alpha N^p$ et $F(N) = \beta N^p$, auquel cas on obtient une suite $t(k)$ vérifiant

$$t(k) = 2t(k-1) + \gamma 2^{kp}$$

avec $\gamma = \alpha + \beta$. Posons $x_k = 2^{-k}t(k)$. La suite x_k vérifie la relation de récurrence

$$x_k = x_{k-1} + \gamma 2^{k(p-1)}$$

Si $p = 1$ (c'est à dire si le temps nécessaire à la partition et à la fusion est proportionnel à la taille N des données ce qui est un cas courant), on obtient $x_k = x_1 + \gamma(k-1) \sim \gamma k$, donc $t(k) = 2^k x_k \sim \gamma k 2^k$. Autrement dit $T(N) \sim \gamma N \log_2 N$ lorsque N est de la forme 2^k. Si N n'est pas une puissance de 2, choisissons k de telle sorte que $2^{k-1} < N < 2^k$. On a alors $T(2^{k-1}) \leq T(N) \leq T(2^k)$ où k est égal à la partie entière de $\log_2 N$ équivalente à $\log_2 N$ et $2^k = 2 \cdot 2^{k-1} < 2N$, ce qui nous donnera une majoration du type $T(N) \leq \delta N \log_2 N$.

Si $p > 1$, on a

$$x_k = x_1 + \gamma \sum_{i=2}^{k} 2^{i(p-1)} = x_1 + \gamma \frac{2^{(k+1)(p-1)} - 2^{2(p-1)}}{2^{p-1} - 1} \sim \delta 2^{k(p-1)}$$

avec $\delta = \gamma \frac{2^{p-1}}{2^{p-1}-1}$. On a alors $t(k) = 2^k x_k \sim \delta 2^{kp}$. On obtient donc $T(N) \sim \delta N^p$ lorsque N est de la forme 2^k. Lorsque N n'est pas de la forme 2^k, on choisit k de telle sorte que $2^{k-1} < N < 2^k$. On a alors

$$T(N) \leq T(2^k) \leq \delta' 2^{kp} \leq \delta'' N^p$$

puisque $2^k = 2 2^{k-1} < 2N$.

Théorème 2.2.1 *On suppose que les temps de partition et de fusion sont des $O(N^p)$ pour un problème de taille N. Alors le temps de calcul dans une méthode "diviser pour régner" est*

- *un $O(N \log_2 N)$ si $p = 1$*
- *un $O(N^p)$ si $p > 1$*

En fait les méthodes *diviser pour régner* sont parfois un peu plus complexes et ne permettent pas tout à fait de traiter les deux sous problèmes de taille $N/2$ de manière indépendante. Comme nous le verrons par la suite, il peut être nécessaire en fait de traiter q sous problèmes de taille $N/2$ ce qui conduit, pour le temps de calcul $T_0(N)$ à des relations de récurrence du type $T_0(N) = qT_0(N/2) + P(N) + F(N)$. Supposons de nouveau que $P(N) + F(N) \leq \gamma N^p$. Alors $T_0(N)$ est clairement majoré par la solution de la récurrence $T(N) = \gamma T(N/2) + \gamma N^p$. Posons alors $t(k) = T(2^k)$. Nous avons donc

$$t(k) = qt(k-1) + \gamma 2^{kp}$$

Soit $\omega = \log_2 q$ et $x_k = 2^{-\omega k} t_k$. On a alors

$$x_k = 2^{-\omega k} \left(2^{\omega} 2^{\omega(k-1)} x_{k-1} + \gamma 2^{kp} \right)$$

soit encore

$$x_k = x_{k-1} + \gamma 2^{k(p-\omega)}$$

On en déduit que $x_k = x_0 + \gamma \sum_{i=1}^{k} 2^{i(p-\omega)}$.

Si $p > \omega$, soit $2^p > q$, ceci nous donne

$$x_k = x_0 + \gamma \frac{2^{(k+1)(p-\omega)} - 2^{p-\omega}}{2^{p-\omega} - 1} = O(2^{k(p-\omega)})$$

soit encore

$$T(N) = T(2^k) = t(k) = 2^{\omega k} O(2^{k(p-\omega)}) = O(2^{kp}) = O(N^p)$$

Par contre, si $p < \omega$, soit $2^p < q$, la suite x_k admet une limite. On a donc $x_k = O(1)$, ce qui montre que

$$T(N) = T(2^k) = t(k) = 2^{\omega k} O(1) = O(2^{\omega k}) = O(N^\omega)$$

Enfin, si $p = \omega$, soit $2^p = q$, on a $x_k = x_0 + \gamma(k-1) = O(k)$. On en déduit que

$$T(N) = T(2^k) = t(k) = 2^{\omega k} O(k) = O(k 2^{\omega k}) = O(N^\omega \log_2 N)$$

Les mêmes méthodes par encadrement que dans le cas $q = 2$ montre que ces estimations restent valables lorsque N n'est pas de la forme 2^k. On a donc

Théorème 2.2.2 *On suppose que dans une méthode "diviser pour régner", la méthode conduise à résoudre q problèmes de taille $N/2$ avec des processus de partition et de fusion en $O(N^p)$, si bien que le temps de calcul $T(N)$ vérifie une relation*

$$T(N) = qT(N/2) + O(N^p)$$

Alors, en posant $\omega = \log_2 q$

- *si $2^p < q$, on a $T(N) = O(N^\omega)$*
- *si $2^p = q$, on a $T(N) = O(N^\omega \log_2 N)$*
- *si $2^p > q$, on a $T(N) = O(N^p)$*

2.2.3 Calcul des puissances d'un entier

Nous allons évaluer le temps de calcul $T(N)$ de la puissance x^N d'un entier x en fonction de l'exposant N. Nous avons déjà vu deux méthodes pour calculer cette puissance

- récursive en remarquant que $x^0 = 1$ et $x^N = xx^{N-1}$
- itérative en remarquant que $x^N = \prod_{i=1}^{N} x$

Dans les deux méthodes, le temps de calcul est proportionnel à N. Nous allons donc utiliser une méthode *diviser pour régner* pour diminuer le temps de calcul. Pour cela nous remarquons que $x^N = (x^2)^p$ si $N = 2p$ est pair et $x^N = x \times (x^2)^p$ si $N = 2p+1$ est impair. Nous obtenons ainsi $T(N) = T(N/2) + \alpha$ si N est pair et $T(N) = T(\frac{N-1}{2}) + 2\alpha$ si N est impair, où α est le temps nécessaire à une multiplication. Dans tous les cas, on a donc $T(N) = T(N/2) + O(1)$, ce qui d'après le théorème précédent nous donne $T(N) = O(\log_2 N)$.

La fonction puissance ainsi définie est naturellement récursive et on obtient une fonction Caml

```
1 static long puiss2(long x, int n){
2       if( n == 0 ) return 1;
3       else if( n %2 == 0 ) return puiss2(x*x, n/2);
4           else return x*puiss2(x*x, n/2);
5 }
```

Programme 2.1 –

Une version non récursive est un peu plus difficile à construire. Définissons une suite x_n par $x_0 = x$ et $x_{n+1} = (x_n)^2$. On a facilement par récurrence $\forall n \geq 0,\ x_n = x^{2^n}$. Décomposons alors n en base 2 sous la forme $n = m_0 2^0 + m_1 2^1 + \cdots + m_k 2^k$ avec $m_i \in \{0, 1\}$. On a alors $x^n = x_0^{m_0} x_1^{m_1} \ldots x_k^{m_k}$. Or on sait calculer les m_i : il suffit de définir les n_i et m_i par récurrence par

$$n_0 = n, \quad m_i = n_i \bmod 2, \quad n_{i+1} = n_i \operatorname{div} 2$$

où l'on désigne par p `div` q le quotient de la division entière de p par q. On stocke alors dans trois références d'une part n_i, d'autre part x_i et enfin $x_0^{m_0} x_1^{m_1} \ldots x_i^{m_i}$. Ceci conduit à la fonction suivante :

```
1 static long puiss2it(long x, int n){
2       long y = x, res = 1;
3        while(n != 0 ){
4             if(n % 2 == 1) res = res * y;
5             y = y * y;
6             n = (n/2);
7        }
8        return res;
9 }
```

Programme 2.2 –

Proposition 2.2.1 *La fonction* `puiss2it` *calcule* x^n *en un temps* $O(\log_2 N)$

Démonstration: L'assertion sur le temps est évidente puisque p étant à itération divisé par 2 et étant initialisé à n au départ, le corps de la boucle ne peut être effectué qu'au plus $\log_2 N$ fois. Ceci montre en même temps que la boucle s'achève bien.

Il nous suffit donc d'exhiber un invariant de la boucle. Or on constate que l'on a l'invariant suivant à la fin de la i-ième itération

$$y = x_i,\ p = n_i \text{ et } \texttt{res} = x_0^{m_0} x_1^{m_1} \dots x_{i-1}^{m_{i-1}}$$

C'est vrai après la 0-ième itération, c'est à dire à l'initialisation de la boucle puisque $y = x = x_0$, $p = n = n_0$ et $\texttt{res} = 1$. De plus, si cette condition est vérifiée après la i-ième itération, on a alors $p \texttt{ mod } 2 = n_i \texttt{ mod2} = m_i$ et en distinguant suivant que cet entier vaut 1 ou 0, on obtient après l'affectation de `res`,

$$\texttt{res} = x_0^{m_0} x_1^{m_1} \dots x_{i-1}^{m_{i-1}} y^{m_i} = x_0^{m_0} x_1^{m_1} \dots x_{i-1}^{m_{i-1}} x_i^{m_i}$$

Après les affectations de y et p, on a $y = x_i^2 = x_{i+1}$ et $p = n_i \texttt{ mod2} = n_{i+1}$, donc la propriété est encore vérifiée après la $(i+1)$-ième itération.

A la sortie de la boucle, on a alors $n_i = 0$, donc $i = k+1$ et alors $\texttt{res} = x_0^{m_0} x_1^{m_1} \dots x_k^{m_k} = x^n$, ce que l'on voulait montrer.

Les méthodes (récursive ou itérative) qui permettent de calculer la puissance n-ième d'un entier s'adaptent sans aucune difficulté au calcul de la puissance n-ième de tout objet pour lequel cela peut avoir un sens (nombre réel, polynôme, matrice, etc). Revenons par exemple au calcul des nombres de Fibonacci F_n. On vérifie immédiatement que

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

d'où bien entendu

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

Le calcul de la puissance n-ième de la matrice pouvant être effectué en un temps $O(\log_2 N)$, il en est donc de même du calcul de F_n. De la méthode récursive brutale d'ordre 2 à la méthode récursive dichotomique (diviser pour régner), nous sommes donc passés d'un temps de calcul en $O(2^n)$ à un temps de calcul en $O(\log_2 n)$; de performances intermédiaires, la méthode itérative ou la méthode récursive d'ordre 1 ont un temps de calcul en $O(n)$.

2.2.4 La multiplication des polynômes

Pour simplifier, nous travaillerons avec des polynômes à coefficients réels. Si $P(X) = \sum_{i=0}^n a_i X^i \in \mathbb{Z}[X]$, nous stockerons les coefficients dans un tableau de taille $n+1$, la valeur de a_i étant stockée dans l'élément de numéro i du tableau. Pour des raisons de commodité, nous définirons une fonction qui renvoie le degré d'un polynôme (égal à la longueur du tableau diminuée de 1) et une autre fonction qui initialise un polynôme $P(X) = \sum_{i=0}^n a_i X^i$ avec tous les a_i égaux à 0.

```
1 static int deg(double[] p){ return (p.length)-1;
2 }
3
4 static double[] init_poly(int n){ return new double[n+1];
5 }
```

Programme 2.3 –

La somme de deux polynômes est facile à définir à l'aide de la fonction suivante :

```
static double[] add_poly(double[] p, double[] q){
       int m = deg(p), n = deg(q);
       double[] somme;
         if( m<n ) return add_poly(q, p);
         else
           {
             somme = init_poly(m);
             for(int i = 0; i <= n; i++){
                  somme[i]= p[i]+q[i];
               }
              for(int i = n+1; i <= m; i++){
                  somme[i]= p[i];
               }
              return somme;
           }
}
```

Programme 2.4 –

Note de programmation On a ici utilisé une petite astuce de programmation en rendant la fonction récursive. Ceci nous permet, dans le cas où le degré de p est strictement plus petit que le degré de q, d'intervertir facilement les paramètres en se contentant de rappeler la fonction `add_poly` avec les paramètres q et p, grâce à la commutativité de l'addition. Ceci améliore sensiblement la lisibilité du programme sans ajouter de complexité supplémentaire. En fait ici, la récursivité n'est que de façade.

La structure même des deux boucles indexées successives montre la proposition suivante :

Proposition 2.2.2 *La fonction* `add_poly` *calcule la somme de deux polynômes p et q en un temps $O(\max(\deg p, \deg q))$.*

En ce qui concerne le produit de deux polynômes, dans une première méthode, nous utiliserons le schéma de calcul suivant

$$\left(\sum_{i=0}^{m} a_i X^i\right)\left(\sum_{j=0}^{n} b_j X^j\right) = \sum_{i=0}^{m}\left(\sum_{j=0}^{n} a_i b_j X^{i+j}\right)$$

qui est le plus facile à programmer. Il suffit donc de faire parcourir à i l'intervalle $[0, m]$, puis de faire parcourir à j l'intervalle $[0, n]$ en ajoutant le produit $a_i b_j$ au terme de degré $i + j$ du polynôme produit. Ce polynôme produit aura été préalablement initialisé à 0.

```
static double[] mult_poly(double[] p, double[] q){
     int m = deg(p), n = deg(q);
     double[] prod = init_poly(m+n);
     for(int i = 0; i <= m; i++){
           for(int j = 0; j <= n; j++){
                prod[i+j]=prod[i+j]+p[i]*q[j];
            }
      }
      return prod;
}
```

Programme 2.5 –

L'imbrication des deux boucles indexées montre que le temps de calcul du produit de deux polynômes de degrés respectifs m et n est proportionnel à $(m+1)(n+1)$. Pour deux polynômes de degrés inférieurs ou égaux à N, le temps de calcul par cette méthode est donc un $O(N^2)$ (d'où la qualification d'algorithme quadratique).

Prenons maintenant deux polynômes $P(X)$ et $Q(X)$ de degrés strictement inférieurs à $N = 2^k = 2M$ et posons $P(X) = P_1(X) + X^M P_2(X)$, $Q(X) = Q_1(X) + X^M Q_2(X)$ (partition). On a donc

$$P(X)Q(X) = P_1(X)Q_1(X) + X^M\big(P_1(X)Q_2(X) + P_2(X)Q_1(X)\big) + X^{2M}Q_1(X)Q_2(X)$$

Il semble qu'il faille calculer 4 produits de polynômes de degré strictement inférieurs à $M = N/2$ pour calculer le produit $P(X)Q(X)$. Comme le temps nécessaire à l'addition de polynômes de degré N est un $O(N)$, on trouve un temps de calcul $T(N) = 4T(N/2) + O(N)$. Avec les notations du paragraphe précédent, on a donc $p = 1$ et $q = 4$; on a $2^p < q$ et donc $T(N) = O(N^{\log_2 4}) = O(N^2)$. Nous n'avons donc rien gagné par rapport à la méthode brutale.

En fait un peu d'astuce va nous éviter de calculer un des produits. Il suffit de remarquer que

$$P_1Q_2 + P_2Q_1 = (P_1 + P_2)(Q_1 + Q_2) - P_1Q_1 - Q_1Q_2$$

autrement dit, si nous posons $R_1 = P_1Q_1$, $R_2 = (P_1 + P_2)(Q_1 + Q_2)$, $R_3 = Q_1Q_2$, nous avons (fusion)

$$PQ = R_1 + (R_2 - R_1 - R_3)X^M + R_3X^{2M}$$

Nous n'avons plus que trois produits de polynômes de degrés strictement inférieurs à $M = N/2$ à effectuer pour calculer le produit $P(X)Q(X)$. Le temps de calcul nécessaire pour effectuer la partition $P(X) = P_1(X) + X^M P_2(X)$, $Q(X) = Q_1(X) + X^M Q_2(X)$ et celui nécessaire pour opérer la fusion

$$PQ = R_1 + (R_2 - R_1 - R_3)X^M + R_3X^{2M}$$

étant clairement un $O(N)$, le temps de calcul nécessaire pour calculer par cette méthode le produit de deux polynômes de degrés strictement inférieurs à $N = 2^k$ vérifiera

$$T(N) = 3T(N/2) + O(N)$$

ce qui nous donne grâce au théorème du paragraphe précédent

$$T(N) = O(N^{\log_2 3})$$

avec $\log_2 3 \sim 1.58$, ce qui est un gain non négligeable.

La méthode de calcul ci-dessus, encore appelée méthode de Karatsuba, est clairement récursive, la récursivité portant sur l'entier $N = 2^k$. Pour $k = 0$, on a $N = 1$, auquel cas les deux polynômes sont constants, leur produit est alors évident ce qui initialise la récursion.

Les polynômes de degré strictement inférieur à $N = 2^k$ seront stockés dans des tableaux de longueur N. Nous utiliserons quelques fonctions utilitaires :

- `scinde_poly` reçoit comme paramètre un polynôme p et envoie comme résultat les deux polynômes p_1 et p_2 de degrés strictement inférieurs à $N/2$ tels que $p(X) = p_1(X) + X^{N/2}p_2(X)$
- `add_poly` et `sub_poly` effectuent la somme et la différence de deux polynômes.

```
1  static double[][] scinde_poly(double[] p){
2     int N = p.length;
3     double[][] couple = new double[2][1];
4     double[] p1=init_poly(N/2); p2=init_poly(N/2);
5     for(int i=0; i<N/2; i++) p1[i] = p[i];
6     for(int i=N/2; i<N; i++) p2[i-N/2] = p[i];
7     couple[1] = p1;
8     couple[2] = p2;
9  }
10
11 static double[] add_poly(double[] p, double[] q){
12         int N = p.length;
13         double[] somme = new double[N];
14         for(int i = 0; i <= N-1; i++){
15                    somme[i]= p[i]+q[i];
16                 }
17         return somme;
18 }
19
20 static double[] sub_poly(double[] p, double[] q){
21         int N = p.length;
22         double[] diff = new double[N];
23         for(int i = 0; i <= N-1; i++){
24                    diff[i]= p[i]-q[i];
25                 }
26         return diff;
27 }
```

Programme 2.6 –

La fonction `fusion` reçoit en paramètre trois polynômes `r1`, `r2` et `r3` de degrés strictement inférieurs à N et l'entier $N = 2M$, et renvoie le polynôme $R_1(X) + R_2(X)X^M + R_3(X)X^{2M}$ grâce à un calcul évident.

```
1  static double[] fusion(double[] r1, double[] r2, double[] r3, int  N){
2           double[] p = new double[2*N];
3                for(int i = 0; i <= N-1; i++){
4                     p[i]=r1[i];
5                }
6               for(int i = 0; i <= N-1; i++){
```

```
                    p[N+i]= r3[i];
                }
                for(int i = 0; i <= N-1; i++){
                    p[N/2+i]=p[N/2+i]+r2[i];
                }
            return p;
}
```

Programme 2.7 –

A partir de là, il suffit de suivre la formulation mathématique pour obtenir la procédure de calcul du produit de deux polynômes :

```
static double[] mult_poly(double[] p, double[] q){
      int N = p.length;
          if( N ==1 )
              {
                 double[] res = new double[1];
                 res[0]= p[0]*q[0];
                 return res;
              }
          else
             {
                double[][] couple_p =  scinde_poly(p);
                double[] p1 = couple_p[0];
                double[] p2 = couple_p[1];
                double[][] couple_q =  scinde_poly(q);
                double[] q1 = couple_q[0];
                double[] q2 = couple_q[1];

                double[] r1 = mult_poly(p1,q1);
                double[] r3 = mult_poly(p2,q2);
                double[] r22 = mult_poly (add_poly( p1, p2),  add_poly( q1, q2));
                double[] r2 = sub_poly (sub_poly( r22, r1), r3);
                 return fusion( r1, r2, r3, N);
              }
}
```


Programme 2.8 –

Un petit programme nous a permis de calculer le nombre de multiplications de nombres entiers effectuées au cours du calcul de $(1+X)^{2^n}$ par la méthode quadratique et par la méthode diviser pour régner (sachant que la multiplication est de loin l'opération la plus lente parmi toutes celles qui sont effectuées sur les coefficients)

n	méthode quadratique	diviser pour régner
3	38	39
4	119	120
5	408	363
6	1497	1092
7	5722	3279

Le gain est peu sensible (voire inexistant) pour des polynômes de petit degré et ne commence à devenir sensible que pour $n = 6$ (résultat de degré $2^6 = 64$). Pour un résultat de degré 128, on voit que la deuxième fonction est environ deux fois plus rapide que la fonction initiale, ce qui n'est pas négligeable [1].

Note de programmation Notre deuxième fonction n'est évidemment pas universelle. Elle suppose *a priori* que les polynômes sont tous deux entrés dans des tableaux de taille $N = 2^k$. Si l'on veut en faire une fonction d'usage général, pour multiplier deux polynômes p et q de degrés respectifs m et n, il faut commencer par déterminer le plus petit entier k tel que $m < 2^k$ et $n < 2^k$, ensuite recopier nos polynômes dans deux tableaux de taille 2^k, appliquer la fonction `mult_poly` et enfin recopier le tableau obtenu (de taille 2^{k+1}) dans un polynôme de degré $m + n$ en supprimant les $2^{k+1} - (m + n + 1)$ derniers coefficients qui sont nuls. Le lecteur vérifiera que ceci ne change pas la complexité de la fonction dont le temps d'exécution est donc un $O(\max(m, n)^{\log_2 3})$.

1. Le lecteur pourra sembler peu convaincu de l'intérêt de manipuler des polynômes de très grands degrés. Le paragraphe suivant sur les grands nombres entiers lui montrera cependant que l'on peut très facilement être amené à manipuler des polynômes de degré 100 ou plus

2.2.5 Les opérations sur les grands nombres entiers

Soit b un entier strictement positif appelé base de numération. On sait que tout nombre entier n peut s'écrire de manière unique sous la forme $n = a_k b^k + a_{k-1} b^{k-1} + \ldots + a_0$ avec $0 \leq a_i \leq b-1$. Introduisons alors le polynôme $P_n(X) = a_k X^k + a_{k-1} X^{k-1} + \ldots + a_0$. On a $n = P_n(b)$. On en déduit que si m et n sont deux entiers, on peut écrire $mn = P_m(b)P_n(b) = Q(b)$ où $Q(X)$ est le produit $P_m(X)P_n(X)$. Le polynôme à coefficients entiers $Q(X)$ s'écrit sous la forme $Q(X) = \alpha_l X^l + \ldots + \alpha_0$, seulement il ne vérifie plus la condition $0 \leq \alpha_i \leq b-1$. Pour calculer les *chiffres* du produit mn dans la base b, il nous faut donc renormaliser le polynôme $Q(X)$. Pour cela, il nous suffit de remarquer que si l'on prend $i \in [0, k]$ et $q_i \in \mathbb{Z}$, le polynôme $Q(X)$ et le polynôme $Q_i(X) = \alpha_l X^l + \ldots + (\alpha_{i+1} + q_i)X^{i+1} + (\alpha_i - bq_i)X^i + \ldots + \alpha_0$, on a $Q_i(b) = Q(b)$; en effet

$$\begin{aligned} Q_i(X) - Q(X) &= (\alpha_{i+1} + q_i)X^{i+1} + (\alpha_i - bq_i)X^i - \alpha_{i+1}X^{i+1} - \alpha_i X^i \\ &= q_i X^{i+1} - bq_i X^i = q_i(X-b)X^{i+1} \end{aligned}$$

s'annule au point $X = b$. En prenant en particulier pour q_i le quotient de la division euclidienne de α_i par b, on a alors $0 \leq \alpha_i - bq_i \leq b-1$, sans perturber les coefficients $\alpha_0, \ldots, \alpha_{i-1}$. Il nous suffit donc pour renormaliser le polynôme Q de parcourir tous les coefficients de $Q(X)$ à partir du bas en remplaçant successivement α_i par $r_i = \alpha_i - bq_i$ et α_{i+1} par $\alpha_{i+1} + q_i$. On finit ainsi par aboutir à un polynôme $R(X) = \beta_{l+\lambda} X^{l+\lambda} + \ldots + \beta_0$ (de degré supérieur à celui de Q) qui vérifie à la fois $0 \leq \beta_i \leq b-1$ et $R(b) = Q(b) = P_m(b)P_n(b) = mn$.

L'usage d'une base de numération permet de dépasser les limitations du langage de programmation en termes de taille des entiers manipulés. Il suffit en effet de remplacer l'entier $n = a_k b^k + a_{k-1} b^{k-1} + \ldots + a_0$ par un tableau noté à la Caml $[|a_0; \ldots; a_k|]$, qui coïncide justement avec la notation utilisée pour le polynôme $P_n(X) = a_k X^k + a_{k-1} X^{k-1} + \ldots + a_0$ vérifiant $n = P_n(b)$. Il nous suffit donc de définir une opération de renormalisation d'un polynôme relativement à la base b pour adapter les procédures de calcul sur les polynômes aux calculs sur les grands nombres entiers stockés sous formes de tableaux en base b.

> **Note de programmation** La procédure de renormalisation d'un polynôme P par rapport à la base b suit l'algorithme décrit ci-dessus en ce qui concerne tous les coefficients, sauf celui de plus haut degré. Il n'est en effet pas possible en Caml de faire croître le tableau au fur et à mesure des besoins pour tenir compte de l'augmentation du degré. Mais il suffit de remarquer qu'une fois le polynôme $P(x)$ écrit sous la forme $P(X) = \beta_l X^l + \ldots + \beta_0$ avec $\forall i \in [0, l-1],\ 0 \leq \beta_i \leq b-1$, si l'écriture de β_l en base b est $\beta_l = \gamma_\lambda b^\lambda + \ldots + \gamma_0$, alors le polynôme normalisé de $P(X)$ n'est autre que
>
> $$(\gamma_\lambda X^\lambda + \ldots + \gamma_0)X^l + \ldots + \beta_0 = \gamma_\lambda X^{\lambda+l} + \ldots + \gamma_0 X^l + \beta_{l-1} X^{l-1} + \ldots + \beta_0$$
>
> Nous utiliserons donc une fonction `table_decimale` qui renvoie dans un tableau les *chiffres* de l'écriture en base b d'un entier n. Pour cela nous stockons au fur et à mesure les *chiffres* dans une liste qui croît dynamiquement suivant l'algorithme évident par divisions euclidiennes successives : le coefficient de b^0 est le reste de la division euclidienne de n par b, puis on remplace n par son quotient par b et on recommence jusqu'à obtenir $n = 0$. Une fois cette liste construite, on la retourne à l'aide de la fonction prédéfinie `rev` et on la transforme en tableau à l'aide de la fonction prédéfinie `vect_of_list`.

```
1  static int[] table_decimales(long n, int b){
2       long x = n,y = b;
3       int l = 1;
4       while(y <= x){
5            y = y*b;
6            l++;
7       }
8       int[] tab=new int[l];
9        for(int i=0; i<l; i++){
10            tab[i] = (int) x % b;
11            x = x/b;
12          }
13      return tab;
14 }
```

Programme 2.9 –

Il nous suffit maintenant de renormaliser le polynôme suivant la base b par l'algorithme suivant :

- on normalise les coefficients de 0 à $N-2$ (si P est de degré $N-1$, donc de longueur N) en remplaçant successivement α_i par $r_i = \alpha_i - bq_i$ et α_{i+1} par $\alpha_{i+1} + q_i$
- on calcule les *chiffres* en base b du coefficient de plus haut degré à l'aide de la fonction `table_decimales` définie ci-dessus
- on concatène les deux tableaux obtenus à l'aide de la fonction `concat`

```
static int[] concat(int[] tab1, int[] tab2){
     int N1= tab1.length, N2 = tab2.length;
     int[] tab = new int[N1+N2];
     for(int i=0; i<N1; i++) tab[i] = tab1[i];
     for(int i=0; i<N2; i++) tab[N1+i] = tab2[i];
     return tab;
}

static int[] renormalise(int[] p, int b){
    int N = p.length,r,q;
    int[] pp = p.clone();
       for(int i = 0; i <= N-2; i++){
          r = pp[i] % b;
          q = pp[i]/b;
          pp[i]=r;
          pp[i+1]=pp[i+1]+q;
       }
       return concat(pp, table_decimales(pp[N-1], b));
}
```

Programme 2.10 –

Il est alors facile de définir l'addition et la multiplication de deux nombres entiers écrits en base b dans deux tableaux :

```
static int[] add_entier(int m, int n, int b){
     return(renormalise (add_poly(m, n), b);
}
static int[] mult_entier(int[] m,int[] n, int b){
     return renormalise(mult_poly(m,n), b);
}
```

Programme 2.11 –

A titre d'exemples, nous allons calculer 100 ! en base 10 ; nous utiliserons une petite fonction d'affichage qui se contente de juxtaposer dans le bon ordre (c'est-à-dire l'ordre inverse) les éléments du tableau

```
static long facto(int n, int b){  (* calcule n! en base b *)
     int[] m = {1};
       for(int i = 2; i <= n; i++){
          m = mult_entier(m, table_decimales( i, b), b);
        }
     return m;
}

static affiche_entier(t){
     for(int i = (t.length)-1; i>= 0;i--)
        System.out.print(t[i]);
     }
     System.out.println("");
}
affiche_entier : int vect -> unit = <fun>
#affiche_entier (facto 100 10)
}
933262154439441526816992388562667004907 1\
596826438162146859296389521759999322991 5\
608941463976156518286253697920827223758 2\
511852109168640000000000000000000000000
```

Programme 2.12 –

On constate que les polynômes manipulés par cette méthode peuvent être de très grands degrés (environ 160 pour celui décrivant l'écriture de 100! en base 10) et que la méthode diviser pour régner prend tout son intérêt pour de tels polynômes.

2.3 Recherches et tris

2.3.1 Recherche binaire ou dichotomique

La recherche d'éléments dans un ensemble est une opération essentielle en algorithmique. Dans le cas où l'ensemble n'est muni d'aucune structure, la seule possibilité est une recherche exhaustive : parcourir tout les éléments de

l'ensemble jusqu'à trouver l'élément choisi. Clairement, si l'ensemble a N éléments, le temps de recherche est un $O(N)$. Le placage d'une structure sur l'ensemble permet souvent de réduire de manière drastique les temps de recherche.

Nous allons montrer que si l'ensemble est un tableau ordonné, le temps de recherche peut être réduit en un $O(\log_2 N)$ par une méthode de type *diviser pour régner*. Considérons en effet un ensemble $X = \{a_0, \ldots, a_{N-1}\}$ ordonné de telle sorte que $a_0 \leq a_1 \leq \ldots \leq a_{N-1}$ et soit b un élément de même type que les a_i. Nous voulons savoir si b appartient à X et éventuellement calculer un indice i tel que $a_i = b$. Donnons nous pour cela $m \in \{0, \ldots, N-1\}$. Si $b \leq a_m$, alors $b \in X$ si et seulement si $b \in \{a_0, \ldots, a_m\}$; si par contre $b > a_m$, alors $b \in X$ si et seulement si $b \in \{a_{m+1}, \ldots, a_{N-1}\}$.

En choisissant un m de l'ordre de $N/2$, on voit que le temps de recherche $T(N)$ d'un élément b dans un ensemble à N éléments vérifiera une récurrence du type $T(N) = T(N/2) + O(1)$. Il s'agit ici d'une méthode diviser pour régner où la partition s'effectue en temps constant (la recherche du milieu m et la comparaison à a_m) et où la fusion est triviale (s'effectue en temps nul). On sait d'après un paragraphe précédent que dans ce cas $T(N) = O(\log_2 N)$.

La recherche dans un tableau trié peut donc se faire de manière récursive (nous avons choisi de renvoyer l'indice i de l'élément a_i tel que $a_i = b$ si $b \in X$, et -1 si $b \notin X$) ; pour cela nous définissons une procédure récursive `cherche` qui effectue le gros du travail : elle reçoit en paramètre les indices i et j qui sont les bornes du sous-tableau $a_i, a_{i+1}, \ldots, a_j$ dans lequel doit s'effectuer la recherche.

- si ce tableau n'a qu'un seul élément, c'est à dire si $i = j$, il suffit de comparer b à a_i pour savoir si $b \in X$ et renvoyer éventuellement l'indice i
- si ce tableau a au moins deux éléments, c'est à dire si $j > i$, on coupe le tableau en 2 à l'indice m quotient entier de i et de j :
  - si $b \leq a_m$, on effectue la recherche dans $a_i, \ldots, a_m$ par un appel récursif à `cherche i m`
  - si $b > a_m$, on effectue la recherche dans $a_{m+1}, \ldots, a_j$ par un appel récursif à `cherche i m`

La procédure de recherche dichotomique se contente de recevoir les paramètres b et X, de créer l'environnement nécessaire à le recherche et d'appeler la procédure `cherche` avec comme paramètres les limites du tableau initial.

```
1  static int recherche_aux(int b, int i, int j, int[] tab){
2           if( i == j )
3              if( tab[i] == b ) return  i; else return -1;
4           else{
5              int m = (i+j)/2;
6              if( b <= tab[m] ) return recherche_aux(b, i, m, tab);
7                    else return recherche_aux(b, m+1, j, tab);
8           }
9  }
10
11 static int recherche_bin(int b, int[] tab){
12        return rec herche_aux(b, 0, N-1, tab);
13 }
```

Programme 2.13 –

Proposition 2.3.1 *La fonction de recherche binaire récursive effectue bien la recherche d'un élément b dans un tableau trié de N éléments en un temps $O(\log_2 N)$.*

Démonstration: l'assertion sur le temps de calcul a déjà été démontrée. Il nous suffit donc de démontrer que la fonction `cherche` renvoie bien le résultat de la recherche de b dans le sous-tableau $a_i \leq \ldots \leq a_j$. Nous allons le faire par récurrence sur $j - i$. C'est clair, si $j - i = 0$. Si $j - i \geq 1$, m est la partie entière de $\frac{i+j}{2}$ si bien que $i \leq m \leq \frac{i+j}{2} < j$.

- Si $b \leq a_m$, alors $b \in \{a_i, \ldots, a_j\} \iff b \in \{a_i, \ldots, a_m\}$ et donc le résultat de la recherche de b dans $\{a_i, \ldots, a_j\}$ est le même que celui de la recherche de b dans $\{a_i, \ldots, a_m\}$. Comme $m - i < j - i$, c'est aussi par l'hypothèse de récurrence le résultat renvoyé par l'appel `cherche` i m
- Si $b > a_m$, alors $b \in \{a_i, \ldots, a_j\} \iff b \in \{a_{m+1}, \ldots, a_j\}$ et donc le résultat de la recherche de b dans $\{a_i, \ldots, a_j\}$ est le même que celui de la recherche de b dans $\{a_i, \ldots, a_m\}$. Comme $j - m - 1 < j - i$, c'est aussi par l'hypothèse de récurrence le résultat renvoyé par l'appel `cherche` $m + 1$ $N - 1$

En conclusion, l'appel `cherche 0 (N-1)` renvoie bien le résultat de la recherche de b dans $\{a_0, \ldots, a_{N-1}\}$ donc dans X.

Note de programmation La vérification de la relation $i \leq m \leq \frac{i+j}{2} < j$ est une étape essentielle de la démonstration. Il serait en effet catastrophique d'avoir un *croisement des indices* c'est à dire qu'à un moment donné on puisse appeler la fonction `cherche i j` avec $i > j$; c'est une des erreurs fréquentes dans les méthodes de dichotomie.

Il est clair que la récursivité ci-dessus est terminale (la fonction se termine par l'appel récursif). Ceci nous permet de construire facilement une version itérative de la recherche en introduisant

- deux références i et j sur les bornes du tableau à tester qui seront actualisées au fur et à mesure
- une boucle booléenne qui procède aux divisions successives du tableau

```
1  static int recherche_bin(int b, int[] tab){
2       int N = tab.length;
3       int i = 0, j =N-1,m;
4            while( (j-i)>0 ){
5                 m = (i+j)/2;
6                 if( b <= tab[m] ) j =m;
7                    else i = m+1;
8            }
9            if( tab[i] == b ) return i; else return -1;
10 }
```

Programme 2.14 –

Proposition 2.3.2 *La fonction de recherche binaire itérative effectue bien la recherche d'un élément b dans un tableau trié de N éléments en un temps $O(\log_2 N)$.*

Démonstration: l'assertion sur le temps de calcul a déjà été démontrée. Il nous faut donc démontrer que la boucle se termine et que la fonction renvoie bien le résultat de la recherche. Pour cela nous allons exhiber un invariant de la boucle booléenne. Posons $f(i,j) = -1$ si $b \notin \{a_i, \ldots, a_j\}$ et $f(i,j) = k$ si k est le plus petit indice tel que $b = a_k$ avec $i \leq k \leq j$. Soit i_n et j_n les valeurs des références i et j à l'entrée dans la n-ième itération. Montrons alors que la condition $i_n < j_n$ et la valeur de $f(i_n, j_n)$ est un invariant de la boucle booléenne à l'entrée dans l'itération. A la première exécution du corps de la boucle, on a $i_1 = 0$, $j_1 = N-1$ et puisque ce corps est exécuté c'est que $0 < N-1$; quant à $f(i_1, j_1)$ il vaut tout simplement $f(0, N-1)$ (c'est à dire la valeur recherchée). Au début de la n-ième itération, on sait que $i_n < j_n$ (puisque le corps de la boucle est exécuté) ; on montre comme dans la démonstration de la fonction récursive que $i_n \leq m \leq \frac{i_n+j_n}{2} < j_n$. On a donc $i_n \leq m < m+1 \leq j_n$. Si $b \leq a_m$, on a alors $f(i_n, j_n) = f(i_n, m) = f(i_{n+1}, j_{n+1})$ et à l'itération suivante, soit $i_{n+1} = j_{n+1}$ auquel cas on sortira de la boucle, soit $i_{n+1} < j_{n+1}$. Si par contre, $b > a_m$, on a alors $f(i_n, j_n) = f(m+1, j_n) = f(i_{n+1}, j_{n+1})$ et à l'itération suivante, soit $i_{n+1} = j_{n+1}$ auquel cas on sortira de la boucle, soit $i_{n+1} < j_{n+1}$.

La stricte décroissance de la suite d'entiers naturels $j_n - i_n$ montre que la boucle s'achève au bout d'un nombre fini d'itérations. A ce moment on a $i_n = j_n$ et $f(i_n, j_n) = f(i_0, j_0) = f(0, N-1)$ puisque cette valeur est un invariant de la boucle. Or $f(i_n, j_n) = f(i_n, i_n)$ vaut i_n si $b = a_{i_n}$ et -1 sinon. On a donc bien calculé $f(0, N-1)$, qui était l'entier recherché.

Remarque Le fait que le tableau soit trié est bien évidemment essentiel. Mais il est aussi essentiel que la structure soit un tableau, ce qui permet d'accéder à l'élément médian a_m en un temps constant. Si au lieu d'un tableau on travaillait avec une liste, cette recherche ne pourrait se faire qu'en un temps $O(N)$ (puisqu'on ne peut accéder aux éléments que séquentiellement) et la recherche dichotomique perdrait tout son intérêt. C'est toute la différence entre une structure à accès direct (où l'on peut accéder directement à tous les éléments) et une structure à accès séquentiel. Remarquons également qu'il est stupide de commencer par trier un tableau avant d'effectuer *une* recherche dans ce tableau puisque le tri lui-même demande un temps en $O(N \log_2 N)$ au minimum ; dans ce cas une banale recherche exhaustive en $O(N)$ est plus performante. Le tri, puis la recherche dichotomique dans un tableau ne peuvent se justifier que si l'on effectue de nombreuses recherches dans ce tableau. Si l'on effectue p recherches dans un tableau à N éléments,

- une recherche exhaustive conduit à un temps d'exécution de l'ordre de αpN
- un tri suivi de recherches dichotomiques peut conduire à un temps d'exécution de l'ordre de $\beta N \log_2 N + \gamma p \log_2 N$ si la méthode de tri est bien choisie.

En gros, on peut dire que dès que p devient de l'ordre de $\log_2 N$, il peut être intéressant d'effectuer un tri préalable.

2.3.2 Le tri fusion

Supposons qu'un enseignant se trouve devant deux paquets de copies, chacun des paquets étant trié par ordre croissant de notes. Il cherche à réunir les deux paquets en un seul trié par ordre croissant. La solution est évidente : il prend à chaque étape la copie ayant la plus petite note parmi les sommets des deux paquets jusqu'à avoir épuisé l'un des deux.

Autrement dit la fusion de deux tableaux triés par ordre croissant peut se faire grâce à la fonction suivante, où il faut simplement prendre garde à l'épuisement possible d'un des deux tableaux :

```
1  static int[] fusion(int[] a, int[] b){
2      int m = a.length, n = b.length, i=0 , j=0;
3      int[] c = new int[ m+n];
4      for(int k = 0; k <= m+n-1; k++){
5              if( (i<m) )
6                {
7                 if( (j<n) )
8                      {
9                       if( a[i] <= b[j] ) { c[k]= a[i]; i = i+1; }
10                        else { c[k]= b[j]; j = j+1; }
11                     }
12                   else
13                     {
14                       c[k]=a[i]; i = i+1;
15                     }
16                }
17             else
18                {
19                  c[k] = b[j]; j = j+1;
20                }
21         }
22       return c;
23 }
```

Programme 2.15 –

Dans le cas où l'on dispose d'une place supplémentaire à la fin des deux tableaux a et b, on peut y disposer une sentinelle dont on est certain qu'elle est supérieure à tous les éléments des deux tableaux. Ceci permet alors de se passer des deux tests de débordement puisque lorsque $i = m$, on est certain que $a_i = a_m \geq b_j$ et lorsque $j = n$, on est certain que $b_j = b_n \geq a_i$. Cependant l'usage d'une telle sentinelle n'est pas évident dans le cas général : ceci oblige à trouver un élément plus grand que tous les éléments des deux tableaux et ensuite à réserver une place supplémentaire dans chacun d'eux pour y stocker la sentinelle.

Dans tous les cas, on constate que le temps d'exécution de la fusion de deux tableaux triés de longueurs m et n est un $O(m+n)$ puisque le corps de la boucle indexée est exécuté $m+n$ fois.

Une fois l'idée de la fusion de deux tableaux triés acquise, la procédure de tri par fusion d'un tableau ne pose plus de difficulté. Etant donné un tableau de taille N, on le partage en deux tableaux de taille (approximative) $N/2$ que l'on trie récursivement et que l'on fusionne ensuite. Bien entendu les appels récursifs se terminent lorsque le tableau est de taille 1 auquel cas il est tout trié.

La partition du tableau se fait en temps constant et la fusion des deux tableaux en un temps $O(\frac{N}{2} + \frac{N}{2}) = O(N)$ si bien que le temps $T(N)$ d'exécution du tri par fusion sur un tableau de N éléments vérifie $T(N) = 2T(N/2) + O(N) + O(1) = 2T(N/2) + O(N)$. On en déduit que $T(N) = O(N \log_2 N)$.

Il nous faut utiliser une procédure de fusion de deux sous-tableaux consécutifs d'un même tableau, plutôt que de fusion de deux tableaux indépendants. La procédure de fusion des deux tableaux triés $\{a_l, a_{l+1}, \ldots, a_m\}$ et $\{a_{m+1}, a_{m+2}, \ldots, a_u\}$ peut s'écrire de la même manière que ci-dessus :

```
1  static void fusion(int[] a, int  l, int m, int u){
2       int[] b = new int[u-l+1];
3       int i = l, j =m+1;
4          for(int k = 0; k <= u-l; k++){
5              if( (i <= m) )
6                  {
7                    if( (j <= u) )
8                      {
9                        if( a[i] <= a[j]) { b[k]= a[i]; i = i+1; }
10                         else { b[k]= a[j]; j = j+1; }
11                     }
12                   else
13                     { b[k]= a[i]; i = i+1; }
14                 }
```

```
15              else { b[k]= a[j]; j = j+1; }
16            }
17          for(int k = 0; k <= u-l; k++){
18            a[l+k]=b[k];
19          }
20      }
```

Programme 2.16 –

En fait, plutôt que de réinitialiser à chaque fusion un nouveau tableau b, il est plus simple de définir un tableau auxiliaire ce qui conduit simplement à supprimer le `b=b = new int[u-l+1];` à condition de disposer dans l'environnement de la fonction de fusion d'un tableau auxiliaire b. Ceci conduit à une procédure de tri d'un tableau par fusion sous la forme :

```
1  static void fusion(int[] a, int[] b, int  l, int m, int u){
2       int i = l, j =m+1;
3          for(int k = 0; k <= u-l; k++){
4             if( (i <= m) )
5                 {
6                   if( (j <= u) )
7                     {
8                       if( a[i] <= a[j]) { b[k]= a[i]; i = i+1; }
9                         else { b[k]= a[j]; j = j+1; }
10                    }
11                  else
12                    { b[k]= a[i]; i = i+1; }
13                }
14              else { b[k]= a[j]; j = j+1; }
15            }
16          for(int k = 0; k <= u-l; k++){
17            a[l+k]=b[k];
18          }
19      }
20
21 static void tri_fusion_aux(int[] a, int[] b, int l, int u){
22       int m;
23        if( u>l )
24              {
25                  m = (l+u)/2;
26                    tri_fusion_aux(a, b, l, m);
27                    tri_fusion_aux(a, b, m+1, u);
28                    fusion(a, b, l, m, u);
29              }
30
31 }
32
33 static void tri_fusion(int[] a){
34      int N = a.length, i=0 , j=0;
35      int[] b = new int[N];
36      tri_fusion_aux(a, b, 0, N-1);
37 }
```

Programme 2.17 –

Note de programmation L'auteur dans une première version avait fait une erreur difficilement décelable en remplaçant l'instruction `b=new int[N]` par `b=a`; il oubliait que dans une telle liaison,dans la mesure où un tableau est une structure modifiable en place, le tableau b devenait physiquement égal au tableau a; toute modification du tableau b (et il y en a) se répercutait sur le tableau a avec toutes les conséquences que l'on peut imaginer. C'est un point à ne pas oublier dans la programmation en Java.

Proposition 2.3.3 *Le tri par fusion trie un tableau de N éléments en un temps $O(N \log_2 N)$.*

Démonstration: Nous allons commencer par montrer que la procédure de fusion fusionne bien deux sous-tableaux triés $a_l \leq a_{l+1} \leq \ldots \leq a_m$ et $a_{m+1} \leq a_{m+2} \leq \ldots \leq a_u$ en un tableau $b_0 \leq b_1 \leq \ldots \leq b_{l-u}$ qui est ensuite recopié en place dans le tableau a.

L'achèvement de la procédure de fusion est garanti par le fait qu'il s'agit d'une boucle indexée. Il nous suffit donc d'exhiber un invariant de la boucle. Pour cela nous allons utiliser un invariant quadruple à l'entrée dans le corps de la boucle

1. $l \leq i \leq m+1$
2. $m+1 \leq j \leq u+1$

3. $i + j = l + m + 1 + k$
4. on a
$$b_0 \leq \ldots \leq b_{k-1} \leq \begin{cases} a_i \leq \ldots \leq a_m & \text{si } i \leq m \\ a_j \leq \ldots \leq a_u & \text{si } j \leq u \end{cases}$$
et la famille $b_0, \ldots, b_{k-1}$ est à une permutation près la famille
$$a_l, \ldots, a_{i-1}, a_{m+1}, \ldots, a_{j-1}$$

C'est clair avant la première itération puisqu'alors $i = l$, $j = m + 1$ et $k = 0$. Supposons que ce soit vrai avant la n-ième itération. Les trois premières conditions sont clairement conservées par le corps de la boucle : les deux premières puisque i n'est pas modifié si $i = m + 1$ et j n'est pas modifié si $j = u + 1$, la troisième puisque lors de l'exécution du corps de la boucle, soit i, soit j est augmenté de 1 et ce de manière exclusive, et qu'ensuite k est augmenté de 1 pour l'exécution suivante du corps de la boucle, ce qui conserve l'égalité $i+j = l+m+1+k$ pour la $(n+1)$-ième itération. Si maintenant $a_i \leq b_j$ alors le corps de la boucle affecte a_i à b_k et augmente i de 1, si bien que l'on a encore $b_0 \leq \ldots \leq b_{k-1} \leq b_k = a_{i-1} \leq \min(a_i, b_j)$ et la quatrième condition est bien vérifiée. De manière similaire, la quatrième condition est bien conservée si $b_j \leq a_i$. Avant l'itération d'indice $k = u - l$, on a donc

1. $l \leq i \leq m + 1$
2. $m + 1 \leq j \leq u + 1$
3. $i + j = l + m + 1 + u - l = u + m + 1$
4. on a
$$b_0 \leq \ldots \leq b_{u-l-1} \leq \begin{cases} a_i \leq \ldots \leq a_m & \text{si } i \leq m \\ a_j \leq \ldots \leq a_u & \text{si } j \leq u \end{cases}$$
et la famille $b_0, \ldots, b_{u-l-1}$ est à une permutation près la famille
$$a_l, \ldots, a_{i-1}, a_{m+1}, \ldots, a_{j-1}$$

Comme $i+j = u+m+1$ avec $i \leq m+1$ et $j \leq u+1$, on a soit $i = m+1$ et $j = u$, soit $j = u+1$ et $i = m$. Dans le premier cas, le corps de la boucle se contente d'affecter a_u à b_k et d'augmenter i, donc après la sortie de la boucle on a $b_0 \leq \ldots \leq b_k = a_u$ et cette famille est à une permutation près la famille
$$a_l, \ldots, a_m, a_{m+1}, \ldots a_u$$
ce qui est ce que l'on souhaitait. Le raisonnement est similaire dans le second cas.

Une fois que l'on sait que la procédure de fusion effectue bien son travail, la démonstration de la procédure `tri` est une banale récurrence sur $u - l$. Si $u = l$ la procédure ne fait rien et le tableau est tout trié, donc elle le trie bien. Si $u - l \geq 1$, la procédure trie deux sous tableaux de taille strictement inférieure, ce qu'elle fait bien par hypothèse de récurrence, puis fusionne les deux tableaux, ce qui est correct.

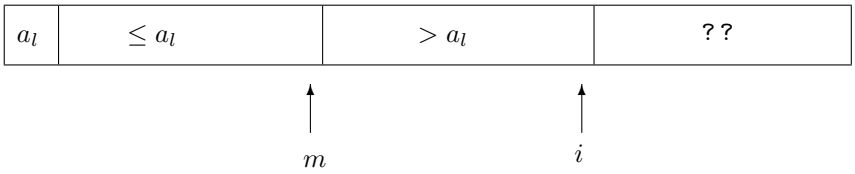
L'assertion sur le temps d'exécution a déjà été justifiée. Ceci achève la démonstration.

2.3.3 Le tri rapide

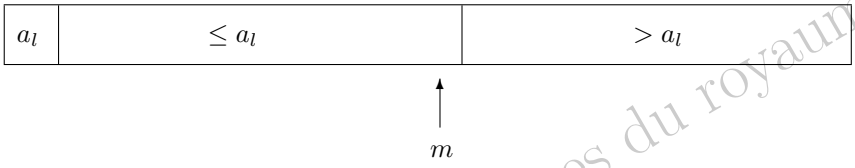
Il procède des mêmes idées que le tri fusion, en partant d'une autre méthode de partition. Considérons un tableau $a_0, \ldots, a_{N-1}$ de N éléments. Effectuons alors une permutation du tableau pour obtenir un nouveau tableau $b_0, \ldots, b_{N-1}$ tel que $a_0 = b_m$ avec
$$\forall i < m,\ b_i \leq b_m \text{ et } \forall i > m,\ b_i > b_m$$
Il est clair que si maintenant on trie les sous-tableaux $b_0, \ldots, b_{m-1}$ et $b_{m+1}, \ldots, b_{N-1}$, le tableau obtenu sera trié.

Contrairement au tri fusion dans lequel la partition est triviale et la fusion plus délicate, dans le tri rapide, la fusion est inexistante (les sous-tableaux étant triés en place) et la partition cruciale. De plus, contrairement au tri fusion, le tri rapide n'a pas besoin de tableau auxiliaire pour effectuer des recopies, il utilise donc moins de mémoire que le tri fusion. Par contre, on verra que dans certains cas très particuliers, il n'a de rapide que le nom !

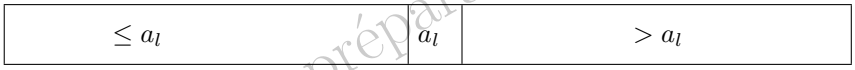
Pour effectuer la partition d'un sous-tableau $a_l, \ldots, a_u$ nous allons effectuer une boucle indexée par $i \in [l+1, u]$ avec un deuxième indice m de manière à maintenir l'invariant suivant



On initialise la boucle avec $i = m = l$. Lorsqu'on augmente i de 1, deux cas sont à considérer
– soit $a_i > a_l$ et dans ce cas il n'y a rien à faire pour conserver l'invariant
– soit $a_i \leq a_l$; il suffit alors d'échanger $a_{m+1} > a_l$ et $a_i \leq a_l$ et d'augmenter m de 1, et l'invariant sera conservé
Après exécution de la totalité de la boucle, on aura donc la situation



Il suffit alors d'échanger a_l et a_m pour avoir la situation



qui est exactement ce que l'on souhaitait.

L'implémentation en Caml de cette procédure de partition peut se faire sous la forme suivante qui partitionne le tableau et retourne l'entier m où est situé le pivot :

```
static void echange(int[] a, int i, int j){
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
}

static int partition(int[] a, int  l, int  u){
    int m =  l;
    int pivot = a[l];
    for(int i = l+1; i <= u; i++){
        if( a[i] <= pivot )
           {
             echange(a, i, m+1);
             m = m+1;
           }
      }
      echange(a, l, m);
      return m;
}
```

Programme 2.18 –

Il suffit ensuite d'insérer cette procédure de partition à l'intérieur d'une procédure de tri récursive en tenant compte de ce qu'un sous-tableau qui a au plus un élément est tout trié, et d'appeler cette procédure récursive pour le tableau entier.

```
static void tri_rapide_aux(int[] a, int l, int u){
   int m = 0;
   if( u>l )
             {
               m = partition(a, l, u);
               tri_rapide_aux(a, l, m-1);
               tri_rapide_aux(a, m+1, u);
             }
}

```

```
11 static void tri_rapide(int[] a){
12     int N = a.length;
13     tri_rapide_aux(a, 0, N-1);
14 }
```

Programme 2.19 –

Proposition 2.3.4 *Le tri rapide trie un tableau de N éléments en un temps $O(N \log_2 N)$ dans le cas moyen et $O(N^2)$ dans le cas le plus défavorable (tableau déjà trié).*

Démonstration: La construction même de la procédure de partition à l'aide d'un invariant de boucle montre qu'elle effectue bien son travail, c'est à dire que lors du retour `partition l u`, l'indice m a été modifié de telle sorte que $l \leq m \leq u$, $\forall i \in [l, m-1],\ a_i \leq a_m$ et $\forall i \in [m+1, u],\ a_i > a_m$. Montrons maintenant par récurrence sur $u - l$ que la procédure `tri l u` trie bien le sous-tableau de l à u. C'est clair si $u - l \leq 0$, puisque la procédure ne fait rien et que le sous-tableau est tout trié. Si $u - l \geq 1$, alors la procédure `tri l u` commence par appeler la procédure de partition. A la sortie de la procédure, on a donc $l \leq m \leq u$, $\forall i \in [l, m-1],\ a_i \leq a_m$ et $\forall i \in [m+1, u],\ a_i > a_m$; ensuite la procédure s'appelle récursivement sur les deux sous-tableaux de tailles strictement inférieures $a_l, \ldots, a_{m-1}$ et $a_{m+1}, \ldots, a_u$; par hypothèse de récurrence la procédure trie bien ces sous-tableaux et à la fin de la procédure on a donc

$$a_l \leq \ldots \leq a_{m-1}\,(a_m)\,a_{m+1} \leq \ldots \leq a_u$$

ce qui achève la démonstration par récurrence.

Le temps d'exécution de la procédure `tri l u` dépend essentiellement de la taille des sous-tableaux $a_l, \ldots, a_{m-1}$ et $a_{m+1}, \ldots, a_u$. Si on note $t(l, u)$ le temps d'exécution de la procédure `tri l u`, comme le temps d'exécution de la procédure de partition est clairement proportionnel à $u - l$, on aura $t(l, u) = t(l, m-1) + t(m+1, l) + \alpha(l - u)$. Le cas le plus défavorable est le cas $m = l$ ou $m = u$ c'est à dire où l'élément a_l est le plus petit ou le plus grand élément du sous-tableau. Dans ce cas, l'un des sous-tableaux est vide, et l'autre est de taille $u - l$. Si ceci se produit tout au long du tri, c'est à dire si le tableau initial est déjà trié par ordre croissant ou décroissant, la formule de récurrence concernant le temps $T(N)$ de tri du tableau sera $T(N) = T(N-1) + O(N)$ et on sait alors que $T(N) = 0(N^2)$.

Si par contre, le tableau n'est *en aucune façon ordonné* et donc parfaitement aléatoire, on peut espérer [2] que m est de l'ordre de $\frac{u+l}{2}$, auquel cas les deux sous-tableaux sont de taille moitié. La formule de récurrence devient alors $T(N) = 2T(N/2) + O(N)$ et on obtient alors $T(N) = O(N \log_2 N)$.

2. La démonstration précise de ce résultat moyen est difficile et dépasse le cadre envisageable ici.

Chapitre 3

Listes et piles

3.1 Listes

Nous oublierons volontairement dans cette section que les listes sont une des structures préexistantes de Java pour mieux comprendre leur structure, leur puissance et leurs limitations.

3.1.1 Listes mathématiques

Soit E un ensemble. Définissons une suite d'ensembles $(E_n)_{n\in\mathbb{N}}$ par $E_0 = \{\emptyset\}$ et $E_{n+1} = E \times E_n$.

Définition 3.1.1 *On appelle ensemble des listes d'éléments de E l'ensemble $\mathcal{L}(E) = \bigcup_{n\in\mathbb{N}} E_n$.*

Autrement dit une liste d'éléments de E est soit rien, soit un élément de E_n du type

$$(a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))$$

où les a_i sont dans E.

Remarque Ce qui est important à noter, c'est qu'on s'interdit d'utiliser l'associativité du produit cartésien et donc d'identifier E_n à $E^n = E \times \ldots \times E$. En effet cette identification a bien un sens pour chaque n fixé : il suffit d'identifier $(a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots))$ à $(a_1, \ldots, a_n)$. Par contre, elle est beaucoup moins significative sur la réunion des E_n. En particulier la projection $p_i : E^n \to E$, $(a_1, \ldots, a_n) \mapsto a_i$ n'est plus définie sur $\bigcup_{n\in\mathbb{N}} E^n$ tout entier, mais seulement sur $\bigcup_{n\geq i} E^n$.

Définition 3.1.2 *On appelle première projection, ou fonction* Tête*, l'application* tete : $\mathcal{L}(E) \setminus \{\emptyset\} \to E$ *qui à toute liste non vide d'éléments de E associe son premier élément :*

$$\text{tete} : (a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots)) \mapsto a_1$$

On appelle deuxième projection, ou fonction Queue*, l'application* queue : $\mathcal{L}(E) \setminus \{\emptyset\} \to \mathcal{L}(E)$ *qui à toute liste non vide d'éléments de E associe son deuxième élément :*

$$\text{queue} : (a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots)) \mapsto (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots)$$

3.1.2 Listes informatiques

On peut reprendre de manière inductive la définition des listes d'éléments de E. Elle se fonde sur les deux règles suivantes

- il existe une suite n'ayant aucun élément, que nous noterons *nil* ou ()
- si a est un élément de E et ℓ une liste, alors le couple (a, ℓ) est encore une liste

ce que l'on peut encore symboliser sous la forme

$$liste = nil \text{ ou } (element, liste)$$

ou si l'on préfère par

$$liste = nil \ + \ element \times liste$$

Ceci correspond exactement à la définition d'un objet en Java. Voilà l'exemple du début de la définition d'une liste d'entiers, avec deux constructeurs, l'un pour construire une liste réduite à un élément, l'autre pour construire une liste ayant une tête et une queue données. La liste vide sera dénotée par `null`.

```
1  public class Liste
2  {
3      private int contenu;
4      private Liste suivant;
5
6      Liste(int x){
7          contenu=x;
8          suivant=null;
9      }
10
11     Liste(int x,Liste q){
12         contenu=x;
13         suivant=q;
14     }
```

Programme 3.1 –

Le test de vacuité d'une liste et la construction des fonctions Tête et Queue se font alors par :

```
1  static boolean estVide(Liste a){
2          return a == null;
3      }
4
5  static int tete(Liste a){
6         if (estVide (a)) throw new Error("VIDE");
7         return a;
8      }
9
10 static Liste queue(Liste a){
11         if (estVide (a)) throw new Error("VIDE");
12         return a.suivant;
13     }
```

Programme 3.2 –

Quant à l'ajout d'un élément en tête de liste, il peut se faire par la fonction `cons`

```
1  static Liste cons(x, l){ return new Liste(x,l);
2  }
```

Programme 3.3 –

Remarque On constate que dans une liste construite par une application répétée de cette fonction `cons`, le seul élément auquel on peut accéder directement est le dernier élément qui a été ajouté. On dit encore que la liste est une structure de type LIFO : *last in first out* ou encore *dernier entré premier sorti*

3.1.3 Opérations sur les listes

Nous allons montrer comment l'on peut définir une opération sur les listes. Toutes les démonstrations de ce paragraphe seront faites par induction structurelle en partant de la définition inductive d'une liste, suivant le schéma décrit par la proposition *informelle* suivante.

Proposition 3.1.1 *Soit f une fonction sur les listes d'éléments de E. On suppose que*
- *f fournit le résultat attendu sur la liste vide*
- *si f fournit le résultat attendu sur la liste ℓ, alors f fournit le résultat attendu sur la liste (a, ℓ) pour tout élément a de E*

Alors f fournit le résultat attendu sur toute liste d'éléments de E.

La définition d'une liste étant récursive, les procédures naturelles sur les listes sont elles mêmes récursives. Nous en donnerons néanmoins presque toujours une version itérative, nous réservant un paragraphe final pour confronter facilité d'écriture et efficacité.

Dans les premières opérations, nous privilégierons les fonctions `tete` et `queue` sur la reconnaissance de motifs, dans un souci de généralisation à d'autres langages que Java. Mais au fur et à mesure, dans un souci de lisibilité accrue, nous privilégierons la reconnaissance de motifs ; nous conseillons au lecteur de faire l'effort de traduction dans les deux sens.

Affichage des éléments d'une liste

A l'aide des fonctions Tête et Queue, on peut parcourir toute une liste

$$(a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots))$$

et en afficher les divers éléments dans l'ordre.

```
1 static void print_liste(Liste l){
2             if(estVide(l)) return;
3             else {
4                  System.out.print(tete(l));
5                  print_liste(queue(l));
6              }
7 }
```

Programme 3.4 –

Proposition 3.1.2 *La procédure* `print_liste` *affiche tous les éléments d'une liste dans l'ordre.*

Démonstration: par induction structurelle. Si la liste est vide, la procédure ne fait rien ce qui est le résultat attendu. Si la liste est de type (a, l) et si la procédure affiche correctement l, alors la procédure commence par afficher a, puis affiche la liste l, ce qui est bien le résultat attendu.

Itération d'une procédure sur une liste

La procédure précédente est le type même d'un sous-programme plus général qui applique une fonction f à tous les éléments d'une liste, en ignorant tous les résultats. Utilisant le caractère fonctionnel de Java, nous pouvons en faire une procédure à deux paramètres, la fonction f et la liste l que nous nommerons `liste_iteration`. Appliquée à une fonction f et à une liste $(a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots))$ elle est équivalente à

$$\texttt{for } i = 1 \texttt{ to } n \texttt{ do } f(a_i) \texttt{ done}$$

```
1 #let rec liste_iteration f l =
2           match l with
3               [] -> ()
4           |    _ -> f (tete l); liste_iteration f (queue l)
5 }
6 liste_iteration : ('a -> 'b) -> 'a Liste -> unit = <fun>
```

Programme 3.5 –

Proposition 3.1.3 *La procédure* `liste_iteration` *applique bien une fonction f à tous les éléments d'une liste, dans l'ordre, en ignorant tous les résultats obtenus.*

Démonstration: par induction structurelle comme dans la démonstration précédente.

Remarque Notre procédure d'affichage peut se définir par

```
let print_liste l = liste_iteration print l
```

ou encore plus simplement (en utilisant la programmation fonctionnelle) par

```
let print_liste = liste_iteration print
```

Projection d'une fonction sur une liste

Dans le même ordre d'idée, étant donnée une fonction $f : E \to F$, il existe une unique application $m_f : \mathcal{L}(E) \to \mathcal{L}(F)$ qui envoie la liste vide sur la liste vide, et qui envoie la liste $(a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots))$ sur la liste $(f(a_1), (f(a_2), (\ldots, (f(a_{n-1}), (f(a_n), \emptyset))) \ldots))$ On peut la définir en Java de manière fonctionnelle sous le nom `map_liste` par

```
1 static Liste map_liste (Liste l){
2           if(estVide(l)) return null;
3           return new Liste(f(tete(l)), map_liste(queue(l)));
4 }
```

Programme 3.6 –

Longueur d'une liste

La longueur d'une liste se calcule de manière récursive
- la longueur de la liste vide est nulle
- la longueur d'une liste non vide est égale à la longueur de sa queue augmentée de 1

```
1 static int longueur(Liste l){
2          if(estVide(l)) return 0;
3          else return longueur(queue(l))+1;
4 }
```

Programme 3.7 –

ou itérative : on introduit une référence sur ℓ, puis on remplace au fur et à mesure ℓ par sa queue en incrémentant un compteur i, jusqu'à obtenir la liste vide ; un invariant évident de la boucle est $i + \text{longueur}(\ell)$ après l'exécution du corps de la boucle ; on voit que la fonction est un peu plus compliqué

```
1 static int longueur(Liste l){
2        Liste x = l;
3        int i = 0;
4        while( !estVide(x) ){
5              i = i+1; x = queue(x);
6        }
7        return i;
8 }
```

Programme 3.8 –

Maximum d'une liste d'entiers

On peut par un simple parcours rechercher le maximum d'une liste d'entiers positifs par
- le maximum de la liste vide est 0 (convention)
- le maximum d'une liste non vide est égal au plus grand des deux nombres suivants : sa tête, le maximum de sa queue.

D'où une procédure récursive

```
1 static int maximum(Liste l){
2          if(estVide(l)) return 0;
3          else return max(tete(l), maximum(queue(l)));
4 }
```

Programme 3.9 –

On peut également en donner une version itérative en parcourant toute la liste et en actualisant au fur et à mesure une référence sur le maximum trouvé (un invariant de la boucle est que $m = \max(a_1, \ldots, a_i)$ après la i-ième itération)

```
1 static int maximum(Liste l){
2        Liste x =  l;
3        int m = 0;
4        while( !estVide(x) ){
5              if(tete(x) > m) m = tete(x);
6              x = queue(x);
```

```
7            }
8           return m;
9   }
```

Programme 3.10 –

Element d'indice n d'une liste

On peut par un simple parcours rechercher le n-ième élément ($n \geq 1$) d'une liste par
- le n-ième élément de la liste vide n'existe pas (on provoque une erreur `Not_found`)
- le premier élément d'une liste est sa Tête
- le n-ième élément d'une liste non vide est égal au $(n-1)$-ième élément de sa Queue

D'où une fonction récursive :

```
1   static int nieme_elt(Liste l, int n){
2         if (estVide (a)) throw new Error("VIDE");
3         if(n==1) return tete(l);
4         else return nieme_elt(queue(l), n-1);
5   }
```

Programme 3.11 –

Une procédure itérative peut être écrite suivant la même méthode que précédemment en remplaçant au fur et à mesure des itérations la liste par sa Queue, et en interrompant brutalement la boucle indexée si la liste devient vide ; un invariant de boucle est qu'après la i-ième exécution du corps de la boucle, la liste contient $a_{i+1}, \ldots, a_m$ où m est la longueur de la liste.

```
1   static int nieme_elt(Liste l, int n){
2          Liste x = l;
3            for(int i = 1; i <= n-1; i++){
4               if( estVide(x)) throw new Error("NON_TROUVE");
5               x = queue(x);
6            }
7            return tete(x);
8   }
```

Programme 3.12 –

Concaténation récursive de deux listes

On veut ici une fonction qui reçoit en paramètres deux listes l_1 d'éléments $a_1, \ldots, a_m$ et l_2 d'éléments $b_1, \ldots, b_n$ et qui retourne la liste concaténée $l_1 * l_2$ d'éléments $a_1, \ldots, a_m, b_1, \ldots, b_n$.

La fonction récursive travaille naturellement sur la liste l_1
- si l_1 est vide, $l_1 * l_2$ est la liste l_2
- sinon, on commence par concaténer la Queue de l_1 avec l_2 et ensuite on ajoute en première position la Tête de l_1

ce qui s'écrit :

```
1   static Liste concat(Liste l1, Liste l2){
2         if(estVide(l1)) return l2;
3         return new Liste(tete(l1),concat(queue(l1),l2));
4   }
```

Programme 3.13 –

La validité de cette fonction se démontre trivialement par induction structurelle sur la liste l_1.

Image miroir d'une liste

Etant donné une liste $(a_1, (a_2, (\ldots, (a_{n-1}, (a_n, \emptyset))) \ldots))$, on cherche à construire la liste *miroir* (c'est à dire inversée) $(a_n, (a_{n-1}, (\ldots, (a_2, (a_1, \emptyset))) \ldots))$. Une première solution récursive se présente à nous si nous disposons d'une fonction de concaténation de deux listes comme nous allons la définir dans le paragraphe suivant. Elle se fonde sur le raisonnement inductif suivant

– l'image miroir de la liste vide est la liste vide
– l'image miroir d'une liste non vide l est obtenue en concaténant l'image miroir de la Queue de la liste l avec la liste dont le seul élément est la Tête de la liste l

Ce qui conduit à la fonction suivante :

```
1 static Liste miroir_quad(Liste l){
2       if(estVide(l)) return null;
3       return concat(miroir_quad(r), newListe(tete(l)));
4 }
```

Programme 3.14 –

La validité de la fonction est tout entière contenue dans le raisonnement inductif. Faisons par contre une étude de complexité. Si n désigne la longueur de la liste l, on voit que la fonction `miroir_quad` est appelée n fois, c'est à dire qu'il s'opère n concaténations d'une liste à $n-1$, puis $n-2$,... , puis 1 éléments avec une liste à 1 élément. Comme on le verra dans le paragraphe suivant, la concaténation d'une liste à p éléments a un temps de calcul proportionnel à p. On en déduit que le temps de calcul d'une image miroir par cet algorithme est proportionnel à $(n-1)+(n-2)+\ldots+2+1 = \frac{n(n-1)}{2} = O(n^2)$, d'où le nom d'algorithme quadratique. Ceci ne semble guère raisonnable, et effectivement ne l'est pas.

Pour obtenir un algorithme récursif qui travaille en un temps $O(n)$ nous allons généraliser notre problème d'image miroir en envisageant une fonction de concaténation à l'envers. On veut ici une fonction qui reçoit en paramètres deux listes ℓ_1 d'éléments $a_1, \ldots, a_m$ et ℓ_2 d'éléments $b_1, \ldots, b_n$ et qui retourne la liste concaténée $\ell_1 * \ell_2$ d'éléments $a_m, \ldots, a_1, b_1, \ldots, b_n$. La fonction récursive travaille naturellement sur la liste ℓ_1

– si ℓ_1 est vide, $\ell_1 * \ell_2$ est la liste ℓ_2
– sinon, on commence par ajouter en première position de ℓ_2 la Tête de ℓ_1 puis on concatène l'inverse de la Queue de ℓ_1

Ceci se traduit en Java par

```
1 static Liste concatene_envers(Liste l1,Liste l2){
2       if(estVide(l1)) return l2;
3       return concatene_envers(queue(l1), new Liste(tete(l1),l2));
4 }
```

Programme 3.15 –

Proposition 3.1.4 *La fonction `concatene_envers` renvoie le résultat de la concaténation de l'image miroir de ℓ_1 avec ℓ_2.*

> **Démonstration**: par induction structurelle. C'est clair si ℓ_1 est la liste vide puisqu'elle renvoie ℓ_2. Supposons que la tête de ℓ_1 soit h et la queue r, et que la proposition soit vraie pour r. Soit ℓ_1 d'éléments $a_1, \ldots, a_m$ et ℓ_2 d'éléments $b_1, \ldots, b_n$. Alors h : : ℓ_2 est la liste d'éléments $a_1, b_1, \ldots, b_n$, h est la liste d'éléments $a_2, \ldots, a_m$ et la concaténation de l'image miroir de h avec h : : ℓ_2 est la liste d'éléments $a_m, \ldots, a_1, b_1, \ldots, b_n$, ce que l'on voulait.

Une fois cette fonction de concaténation à l'envers construite, il suffit simplement pour construire l'image miroir d'une liste ℓ de la concaténer à l'envers avec la liste vide. La fonction `concatene_envers` travaille manifestement dans un temps proportionnel à la longueur de la liste ℓ_1, et donc cette fonction `miroir` calcule l'image miroir d'une liste de longueur n en un temps $O(n)$ (linéaire).

```
1 static Liste miroir(Liste l){
2       return concatene_envers(l, null);
3 }
```

Programme 3.16 –

La même démarche conduit à une procédure itérative de concaténation à l'envers en utilisant deux références, l'une sur la liste ℓ_1 qui décroît, l'autre sur la liste ℓ_2 qui s'adjoint au fur et à mesure les éléments de têtes de ℓ_1, avec l'invariant de boucle : après la i-ième itération, x_1 est la liste d'éléments $a_{i+1}, \ldots, a_n$ et x_2 est la liste d'éléments $a_i, \ldots, a_1, b_1, \ldots, b_n$; la terminaison est garantie par la décroissance stricte de la longueur de la liste x_1.

```
static Liste concatene_envers(Liste l1,Liste l2){
     Liste x2 = l2, x1 = l1;
     while(!estVide(x1)){
            x2 = new Liste(tete(x1), x2);
            x1 = queue(x1);
        }
      return x2;
}
```

Programme 3.17 –

L'image miroir peut alors se construire comme précédemment par concaténation inverse avec la liste vide, ce qui conduit à :

```
static Liste miroir(Liste l){
     Liste x = l,  accu = null;
         while( !estVide(x) ){
            accu = new Liste(tete(x), accu);
            x = queue(x);
         }
      return accu;
}
```

Programme 3.18 –

Concaténation itérative de deux listes

Une version itérative de la concaténation de deux listes $a_1, \ldots, a_m$ et $b_1, \ldots, b_n$ est *a priori* plus difficile à construire car de toute évidence l'appel récursif n'est pas terminal. On constate que l'on est un peu bloqué pour construire itérativement la liste d'éléments $a_1, \ldots, a_m, b_1, \ldots, b_n$, sachant qu'à la première étape on n'accède aisément qu'à a_1 et b_1 :

- on ne peut pas construire facilement $a_m, b_1, \ldots, b_n$ car on accède difficilement à a_m
- on ne peut pas construire facilement $a_1, \ldots, a_m, b_1$ car on ne sait pas ajouter facilement un élément en bout de liste

La solution consiste simplement à utiliser les fonctions itératives d'image miroir du paragraphe précédent et à concaténer à l'envers l'image miroir de l_1 avec l_2, ce qui nous donne

```
static Liste concat(Liste l1, Liste l2){
         return concat_rev(miroir(l1), l2);
}
```

Programme 3.19 –

Insertion d'un élément dans une liste

Nous recherchons une fonction qui étant donnée une liste d'éléments $a_1, \ldots, a_m$, un entier $n \in [1, m+1]$ et un élément x de E, nous construise la liste $a_1, \ldots, a_{n-1}, x, a_n, \ldots, a_m$, où x a été inséré à la n-ième place L'induction structurelle se formule ainsi

- si $n = 1$, il suffit d'ajouter x en tête de la liste l
- si $n > 1$ et si la liste est vide, il y a erreur
- si $n > 1$ et la liste est non vide, le résultat est la liste dont la tête est la tête de l et donc la queue est obtenue en insérant l'élément x à la $n-1$-ième place de la queue de l.

Nous obtenons la fonction suivante (démonstration évidente par récurrence sur n)

```
static Liste insere(int  x, Liste  l, int n){
        if( n == 1) return new Liste(x, l);
         if(estVide(l)) throw new Error("Vide");
         return new Liste( tete(l), insere(x, queue(l), n-1));
}
```

Programme 3.20 –

De nouveau ici la récursivité n'est pas terminale, et il n'est pas facile de donner une version itérative de cette fonction d'insertion. En effet le parcours obligatoire de la liste à partir du premier élément nous oblige a priori à ajouter les éléments successifs en queue de l'accumulateur qui va recevoir le résultat, puis à ajouter x en queue de cet accumulateur et enfin à ajouter les éléments successifs de la liste en queue de l'accumulateur, ce que nous ne pouvons pas faire facilement puisqu'on ne sait ajouter efficacement des éléments qu'en tête d'une liste.

Nous contournerons la difficulté en ajoutant tous ces éléments en tête de l'accumulateur. Le résultat sera bien entendu non pas la liste avec x inséré, mais son image miroir. Il suffit ensuite de prendre l'image miroir de ce résultat stocké dans l'accumulateur. Un invariant de la première boucle est : après la i-ième itération, `accu` contient $a_i, \ldots, a_1$ et $n \leq m$; un invariant de la deuxième boucle est : après la i-ième itération, `accu` contient $a_{n+i}, a_{n+i-1}, \ldots, a_{n+1}, x, a_n, \ldots, a_1$.

```
1  static Liste insere(int x, Liste l, int  n){
2        Liste l1 =  l, accu = null;
3          for(int i = 1; i <= n-1; i++){
4              if( estVide(l1)) throw new Error("Liste vide");
5              accu = new Liste(tete( l1), accu);
6              l1 = queue (l1);
7          }
8          accu = new Liste(x, accu);
9          while( !estVide(l1)){
10             accu = new Liste(tete( l1), accu);
11             l1 = queue (l1);
12         }
13         return miroir(accu);
14 }
```

Programme 3.21 –

Suppression d'un élément dans une liste

Nous cherchons une fonction qui à une liste ℓ et un élément x associe la liste obtenue en supprimant *tous* les éléments de la liste égaux à x. L'induction structurelle est

- si la liste est vide on ne fait rien
- si la liste est non vide, on supprime x de la queue de ℓ et on met en tête la tête de ℓ si celle ci n'est pas égale à x ce qui conduit à

```
1  static Liste suppr(Liste l, int  x){
2          if(estVide(l)) return l;
3          int t = tete(l);
4          if(t==x) return queue(l);
5          return new Liste(tete(l), supprime(x,queue(l)));
6  }
```

Programme 3.22 –

Une version itérative est aisément obtenue en cherchant l'image miroir de la liste ℓ et en *oubliant* d'ajouter les éléments égaux à x. Il suffit ensuite de renvoyer l'image miroir de l'accumulateur. Un invariant de la boucle est : après la i-ième exécution du corps de la boucle, `accu` contient ceux des élément $a_i, \ldots, a_1$ non égaux à x, dans cet ordre.

```
1  static Liste suppr(Liste l, int  x){
2       Liste l1 =l, accu = null;
3       while(!estVide(l1){
4           int t = tete(l1);
5           if(t != x) accu = new Liste(t, accu);
6           l1 = queue(l1);
7        }
8        return miroir(accu);
9  }
```

Programme 3.23 –

3.1.4 Comparaison des opérations récursives et itératives sur les listes

Nous avons construit dans le paragraphe précédent les opérations de base sur les listes avec chaque fois une version récursive et une version itérative. Nous cherchons donc à comparer ces deux modes de programmation sur les listes selon trois points de vue

– clarté et démonstration de la méthode
– efficacité en temps de calcul
– efficacité en occupation mémoire

En ce qui concerne la clarté et la concision, la balance penche de façon évidente en faveur des versions récursives ; ceci favorise également leur démonstration puisque la formulation même des fonctions suit l'induction structurelle qui en constitue par là-même la démonstration.

En ce qui concerne l'efficacité en temps de calcul, il est clair que tous les algorithmes récursifs que nous avons construits ont des temps de calcul en $O(n)$, si n désigne la longueur de la liste traitée. Il peut en sembler de même pour les algorithmes itératifs, qui tous sont constitués d'une boucle qui est effectuée au plus n fois. En fait une partie du travail est masquée par les affectations des références et en particulier les affectations `x:=queue !x` qui font décroître la liste sur laquelle on travaille : il est clair que le processeur doit commencer par procéder à une recopie de la liste avant de l'affecter et que le temps de cette recopie ne peut être que proportionnel à la longueur de la liste. Nos versions itératives ne sont peut-être pas si linéaires que cela, et un temps de calcul en $O(n^2)$ est fort probable.

En ce qui concerne l'efficacité en occupation mémoire, les versions récursives obligent la machine à une sauvegarde du contexte à chaque appel récursif. Par contre, dans les versions itératives, l'encombrement mémoire est uniquement celui des références et peut sembler moindre ; ce n'est vrai que dans la mesure où la machine parvient à récupérer efficacement les emplacements mémoires occupés par les listes qui ne sont plus pointées par les références après leurs affectations répétées, ce qui n'est peut être pas garanti (cela l'est en Java). De toute façon, cette libération de mémoire prend du temps.

En conclusion, en ce qui concerne les listes, comme en général pour toute structure définie inductivement, les sous-programmes récursifs sont souvent les plus concis, les plus clairs, les moins susceptibles de contenir des erreurs, et les plus efficaces. Il faut donc les privilégier.

3.1.5 Tris de listes

Nous allons ici étudier les méthodes de tri de listes, en traitant le cas de listes d'entiers ; le lecteur généralisera facilement au cas du tri d'éléments d'un ensemble ordonné quelconque. Nous n'allons étudier ici que deux tris, particulièrement bien adaptés aux listes : le tri par insertion en $O(n^2)$ qui est simple et facile à programmer, le tri par fusion en $O(n \log_2 n)$ qui est très efficace en temps de calcul. Nous n'étudierons que des versions récursives de ces tris, les versions itératives ne présentant, comme nous l'avons vu dans le paragraphe précédent, aucun intérêt.

Tri d'une liste par insertion

Recherchons tout d'abord comment faire l'insertion (à la bonne place) d'un élément x dans une liste déjà triée ℓ. L'induction structurelle est la suivante

– si la liste est vide, le résultat est la liste dont le seul élément est x
– si la liste est non vide et si x est inférieur ou égal à la tête de ℓ, le résultat est la liste dont la tête est x et la queue est ℓ
– si la liste est non vide et si x est supérieur à la tête de ℓ, le résultat est la liste dont la tête est la tête de ℓ et la queue est le résultat de l'insertion de x dans la queue de ℓ.

On obtient une fonction Java (dont le temps de calcul est en $O(n)$ si n est la longueur de ℓ)

```
1 static Liste insere_tri(int x, Liste  l){
2         if(estVide(l)) return new Liste(x);
3         int t = tete(l);
4         if( x <= t) return new Liste(x , l);
5         else return new Liste(t, insere(x,l));
6 }
```

Programme 3.24 –

Le tri par insertion d'une liste est alors très simple

– si la liste est vide, on ne fait rien
– si la liste est non vide, on trie la queue de ℓ et on y insère la tête de ℓ

Ceci conduit à une fonction de tri dont le temps de calcul vérifie $T(n) = T(n-1) + O(n)$ (le $O(n)$ étant dû à l'insertion), soit $T(n) = O(n^2)$.

```
static Liste tri_insertion(Liste  l){
        if(estVide(l)) return l;
        return insere(tete(l), tri_insertion(queue(l)));
}
```

Programme 3.25 –

3.1.6 Tri par fusion

Nous renvoyons le lecteur au paragraphe sur les tris pour ce qui concerne le principe du tri par fusion. Rappelons simplement que nous avons besoin de deux fonctions : une fonction de partition qui partage une liste en deux listes de tailles similaires, une fonction de fusion qui fusionne deux listes déjà triées en une nouvelle liste triée. Les démonstrations de ce paragraphe par induction structurelle sont laissées au soin du lecteur : elles ne présentent aucune difficulté.

La fonction de partition opère de manière inductive

- si la liste est vide, le résultat est le couple formé de deux listes vides
- si la liste a un seul élément, le résultat est le couple formé de la liste ℓ et de la liste vide
- si la liste a au moins deux éléments, on partitionne la queue de la queue de ℓ (c'est à dire ℓ privé de ses deux premiers éléments) et on ajoute le premier élément de ℓ à la première liste et le deuxième élément de ℓ à la deuxième liste

On obtient la fonction Java `partage` :

```
static void partage_aux(Liste[] couple, Liste l){
          if(estVide(l)) return;
          if(estVide(queue(l))){ couple[0] = new Liste(tete(l), couple[0]); return; }
          int t1 = tete(l);
          int t2 = tete(queue(l));
          Liste r = queue(queue(l));
          couple[0] = new Liste(t1, couple[0]);
          couple[1] = new Liste(t2, couple[1]);
          partage_aux(couple, r);
}
static Liste[] partage(Liste l){
           Liste[] couple = new Liste[2];
           partage_aux(couple, l);
           return couple;
}
```

Programme 3.26 –

La fusion de deux listes triées l_1 et l_2 opère également de manière inductive

- si l'une des listes est vide, le résultat est l'autre liste
- si la tête de ℓ_1 est inférieure ou égale à la tête de ℓ_2, on fusionne la queue de ℓ_1 avec ℓ_2 et on ajoute la tête de ℓ_1 en tête du résultat obtenu
- si la tête de ℓ_1 est supérieure à la tête de ℓ_2, on fusionne la queue de ℓ_2 avec ℓ_1 et on ajoute la tête de ℓ_2 en tête du résultat obtenu

On obtient la fonction Java suivante :

```
static Liste fusion(Liste l1, Liste l2){
          if(estVide(l1)) return l2;
          if(estVide(l2)) return l1;
          int t1 = tete(l1);
          int t2 = tete(l2);
          Liste q1 = queue(l1);
          Liste q2 = queue(l2);
          if(t1 <= t2) return new Liste(t1, fusion(q1, l2));
          else return new Liste(t2, fusion (l1, q2);
}
fusion : int list -> int list -> int list = <fun>
```

Programme 3.27 –

Le tri par fusion procède alors simplement par partage de la liste, tri récursif des deux listes obtenues et fusion de ces deux listes. Le partage et la fusion ayant visiblement des temps de calcul en $O(n)$, le temps de calcul $T(n)$ du tri par fusion vérifiera $T(n) = 2T(n/2) + O(n)$, ce qui conduit à $T(n) = O(n \log_2 n)$.

```
static Liste tri_fusion(Liste l){
          Liste[] couple = partage(l);
          return fusion(tri_fusion(couple[0]),tri_fusion(couple[1]));
}
```

Programme 3.28 –

3.1.7 Listes et structures mathématiques

Les listes sont particulièrement adaptées à des structures mathématiques creuses, c'est à dire où beaucoup d'éléments ont des valeurs par défauts, la plupart du temps 0. C'est ainsi que si l'on travaille avec le polynôme $1 + 3X^2 + X^{100}$, c'est un épouvantable gâchis de le stocker dans un tableau de taille 101 dont 98 éléments vaudront 0. Nous allons étudier ces objets *creux* sur deux exemples : les polynômes et les matrices.

3.1.8 Tri rapide

Polynômes creux

L'idée est de stocker un polynôme comme une liste ordonnée de monômes non nuls et de stocker un monôme a_iX^i sous la forme du couple (i, a_i). Un polynôme creux à coefficients réels sera donc stocké comme une liste de couples formés d'un entier positif et d'un nombre réel.

- le polynôme nul est stocké comme la liste vide
- le polynôme $1 + 3X^2 + X^{100}$ est stocké comme $[(0, 1.), (2, 3.), (100, 1.)]$.

Ceci nous conduit simplement à la classe des polynômes :

```
public class Polynome{
          int deg;
          double coeff;
          Polynome suivant;

          Polynome(int deg, float coeff){
                this.deg = deg;
                this.coeff = coeff;
                suivant = null;
          }

           Polynome(int deg, float coeff, Polynome suivant){
                this.deg = deg;
                this.coeff = coeff;
                this.suivant = suivant;
          }
}
```

Programme 3.29 –

avec quelques fonctions utilitaires

```
static boolean estNul(Polynome p){
         return p==null;
}
static int degre(Polynome p){
        return p.deg;
}
static double coefficient(Polynome p){
       return p.coeff;
}
```

Programme 3.30 –

L'addition se fait par une variante très simple de la procédure utilisée pour la fusion de deux listes triées : au lieu de simplement fusionner les deux listes, on procède par addition des coefficients lorsque les degrés des monômes de tête sont les mêmes (et la somme des coefficients non nulle), par simple fusion dans le cas contraire, et ceci récursivement.

```
static Polynome add_pol(Polynome p1, Polynome p2){
         if(estNul(p1)) return p2;
         if(estNul(p2)) return p1;
         int d1=degre(p1); int d2=degre(p2);
```

```
5           double c1=coefficient(p1); double c2=coefficient(p2);
6           if( d1<d2 ) return new Polynome(d1, c1, add_pol(p1.suivant,p2));
7            else if( d2<d1 ) return new Polynome(d2,c2,  add_pol(p1,p2.suivant));
8            else if( c1+ c2 != 0 ) return new Polynome(d1, c1+c2,  add_pol(p1.suivant,p2.suivant));
9            else return add_pol(p1.suivant, p2.suivant);
10  }
```

Programme 3.31 –

Pour définir le produit de deux polynômes, on commence par définir le produit d'un polynôme par un monôme de la manière évidente : $aX^d \sum_{i=0}^{n} b_i X^i = \sum_{i=0}^{n} (ab_i) X^{d+i}$

```
1   static Polynome prod_mon_pol(int d, double c, Polynome p){
2           if( a == 0. ) return null;
3           if(estNul(p)) return null;
4           int dp = degre(p);
5           double cp = coefficient(p);
6           return new Polynome( d+dp, c*cp, prod_mon_pol(d, c, p.suivant));
7   }
```

Programme 3.32 –

Puis on définit le produit de deux polynômes par

$$(\sum_{i=0}^{m} a_i X^i) P_2(X) = \sum_{i=0}^{m} (a_i X^i P_2(X))$$

soit en Java :

```
1   static Polynome prod_pol(Polynome p1, Polynome p2){
2           if(estNul(p1)) return null;
3           if(estNul(p2)) return null;
4            int d1=degre(p1);
5           double c1=coefficient(p1);
6           return add_pol (prod_mon_pol(d1, c1, p2), prod_pol(p1.suivant, p2));
7   }
```

Programme 3.33 –

Nous laissons le soin au lecteur d'écrire quelques procédures utiles sur les polynômes creux : multiplication par un scalaire, soustraction, puissance, dérivée, dérivée n-ième.

Matrices creuses

Le principe est le même que pour les polynômes creux, sauf que l'on remplacera le degré du monôme par le couple (i, j) qui indexe l'élément $a_{i,j}$, ces éléments étant ordonnés par exemple selon l'ordre lexicographique.

```
1
2
3   public class Matrice{
4              int row, col;
5              double coeff;
6              Matrice suivant;
7
8              Matrice(int ligne, int colonne, float coeff){
9                     this.row = ligne;
10                    this.col= colonne;
11                    this.coeff = coeff;
12                    suivant = null;
13             }
14
15              Polynome(int deg, float coeff, Polynome suivant){
16                    this.row = ligne;
17                    this.col = colonne;
18                    this.coeff = coeff;
19                    this.suivant = suivant;
20             }
21  }
```

Programme 3.34 –

```
static boolean lex(int i1, int j1, int i2, int j2){  (* ordre lexico strict *)
       return (i1<i2) || (i1 == i2 && j1<j2);
}

static boolean estNulle(Matrice m){
         return m==null;
}
static int ligne(Matrice m){
        return m.row;
}
static int colonne(Matrice m){
        return m.col;
}
static double coefficient(Matrice m){
       return m.coeff;
}
```

Programme 3.35 –

L'addition des matrices creuses est tout à fait similaire à l'addition des polynômes creux, à la seule différence près que l'on utilise l'ordre lexicographique :

```
static Matrice add_mat(Matrice m1, Matrice m2){
         if(estNulle(m1)) return m2;
         if(estNul(m2)) return m1;
         int i1=ligne(m1); int j1=col(m1);
         int i2=ligne(m2); int j2=col(m2);
         double c1=coefficient(p1); double c2=coefficient(p2);
         if( lex(i1 , j1, i2, j2) ) return new Matrice(i1, j1, c1, add_mat(m1.suivant, m2));
          else if( lex(i2, j2, i1, j1) ) return new Matrice(i2, j2, c2,  add_mat(m1.suivant, m2));
          else if( c1+ c2 != 0 ) return new Matrice(i1, j1, c1+c2,  add_mat(m1.suivant, m2.suivant));
          else return add_mat(m1.suivant, m2.suivant);
}
```

Programme 3.36 –

En ce qui concerne le produit des matrices creuses, on commence par définir le produit d'une matrice B par une matrice élémentaire du type $aE_{i,j}$ où $E_{i,j}$ est la matrice qui a des zéros partout, sauf à l'intersection de la i-ième ligne et de la j-ième colonne où figure un 1. On rappelle que $E_{i,j}E_{k,l} = \delta_j^k E_{i,l}$ avec $\delta_j^k = 1$ si $j = k$ et $\delta_j^k = 0$ sinon. On en déduit que si $B = bE_{k,l} + B_1$ alors

$$aE_{i,j}B = ab\delta_j^k E_{i,l} + aE_{i,j}B_1$$

Le lemme suivant nous garantit que l'ordre lexicographique sur les indices correspondant à des termes non nuls est conservé.

Lemme 3.1.1 *L'ordre lexicographique sur les matrices $E_{i,j}$ est compatible avec la multiplication à gauche et à droite.*

Démonstration: Bien entendu notre affirmation est un peu abusive, à moins de prendre par convention que les deux affirmations $A \prec 0$ et $0 \prec A$ sont vraies. Ceci signifie simplement que si $(i,j) \prec (i',j')$ et si les produits sont tous deux non nuls, $E_{i,j}E_{k,l} \prec E_{i',j'}E_{k,l}$ et de même $E_{k,l}E_{i,j} \prec E_{k,l}E_{i',j'}$.

Examinons le premier cas. Le fait que les deux produits soient non nuls nécessite $j = j' = k$. Comme $(i,j) \prec (i',j')$, c'est donc que $i < i'$ et donc $(i,l) \prec (i',l)$ soit $E_{i,j}E_{k,l} = E_{i,l} \prec E_{i',l} = E_{i',j'}E_{k,l}$.

Dans le second cas, le fait que les deux produits soient non nuls nécessite $i = i' = l$. Comme $(i,j) \prec (i',j')$, c'est donc que $j < j'$ et donc $(k,j) \prec (k,j')$ soit $E_{k,l}E_{i,j} = E_{k,j} \prec E_{k,j'} = E_{k,l}E_{i',j'}$.

Ceci conduit à la fonction Java

```
static Matrice prod_mat_elem(int i, int j , double a, Matrice m){
        if( a==0) return null;
        if(estNulle(m)) return null;
        int k = ligne(m); int l = colonne(m);
        double c = coefficient(m);
        if(j==k) return new Matrice(i, l, a*c, prod_mat_elem(i, j, a, m.suivant));
        else return prod_mat_elem(i, j, a, m.suivant);
```

Programme 3.37 –

On peut alors écrire une fonction de produit de matrices creuses comme pour les polynômes creux. Le lemme précédent nous garantit encore une fois que l'ordre lexicographique sur les indices des termes non nuls est conservé.

```
1 static Matrice prod_mat(Matrice m1, Matrice m2){
2         if( estNulle(m1)) return null;
3         if( estNulle(m2)) return null;
4         int i1 = ligne(m1); int j1 = colonne(m1);
5         double c1 = coefficient(m1);
6         return add_mat(prod_mat_elem(i1, j1, c1, m2), prod_mat(m1.suivant, m2));
7 }
```

Programme 3.38 –

3.2 Piles

3.2.1 Types de piles informatiques

Il arrive souvent qu'en algorithmique on ait besoin de sauvegarder des valeurs. On utilise alors généralement une pile. Qu'est ce qu'une pile ? C'est tout simplement un objet informatique modifiable qui dispose de deux méthodes (sous-programmes), une procédure permettant de ranger une valeur dans cette pile (en général notée `push`), une fonction retournant une valeur rangée dans cette pile et la supprimant au passage de la pile (en général notée `pop`). On voit donc qu'une pile est particulièrement adaptée à ranger provisoirement des valeurs dont on n'aura besoin qu'une seule fois par la suite.

Bien entendu, les opérations de rangement dans la pile (on dit plutôt *empilage*) et les opérations de récupération/suppression dans la pile (on dit plutôt *dépilage*) ont tendance à être très imbriquées. On peut avoir deux empilages, suivis d'un dépilage, suivi de trois empilages, suivis de quatre dépilages, etc. On distingue deux grandes méthodes de dépilages.

Le premier type de pile est similaire à une pile d'assiettes rangées dans un placard. Lorsqu'on y empile une assiette, elle se trouve sur le dessus. Lorsque l'on dépile une assiette, on la prend sur le dessus. Autrement dit, l'assiette que l'on dépile est toujours la dernière qui a été empilée. On parle alors de pile LIFO pour *Last In First Out*, soit *dernier entré premier sorti*. Ces piles portent encore en anglais le nom de *stack* ; ce sont celles que l'on rencontre le plus fréquemment en informatique, en particulier dans tous les problèmes liés à la récursivité : en effet lorsqu'on revient d'un sous programme appelé, c'est toujours dans le dernier sous programme appelant. C'est ce type de piles que nous allons considérer.

Le deuxième type est similaire à certains distributeurs de gobelets, où l'on introduit les gobelets par le haut et où on les récupère par le bas. Dans ce cas, le gobelet que l'on récupère est toujours le premier qui a été empilé. On parle alors de pile FIFO pour *First In First Out*, soit *premier entré premier sorti*. Ces piles portent encore le nom de *queue* ou *files d'attente* (analogue à une queue pour aller au cinéma ou pour franchir un péage d'autoroute). Elles sont moins fondamentales pour nous et interviennent principalement dans les gestions de files d'attente et tous les phénomènes où la chronologie a de l'importance (gestion du clavier, des clics de la souris, etc.).

3.2.2 Piles informatiques (LIFO)

Nous connaissons déjà des structures analogues aux piles LIFO, ce sont les listes. L'élément d'une liste auquel on peut accéder directement est le dernier élément qui y a été entré. Tout ce qu'il nous faut c'est une pile qui est modifiable tout au long du travail et qui dispose de deux sous-programmes `push` et `pop`. Le premier sous-programme est une procédure qui ajoute un élément x au sommet de la pile. Le second est une fonction qui renvoie l'élément du sommet de la pile tout en le supprimant de la pile.

Une référence sur une liste est tout à fait adaptée à cet effet et on peut simuler une pile en Java avec les deux sous programmes. Bien entendu, il est prudent de prévoir un message d'erreur pour le cas où l'on essaye de dépiler dans une pile vide ce qui arrive plus fréquemment qu'on ne le voudrait.

```
1 public class Pile{
2             Liste lapile;
3
4             Pile(){
5                  lapile=null;
6             }
7
8            public boolean estVide(){
9                  return lapile==null;
```

```
10            }
11
12            static boolean estVide(Pile s){
13                  return s.lapile==null;
14            }
15
16          public void push(int x){
17                lapile = new Liste(x , lapile);
18            }
19
20            static void push(Pile s, int x){
21                  s.lapile = new Liste(x, s.lapile);
22            }
23
24            public int pop(){
25                  if(lapile == null) throw new Error("Pile vide");
26                  int x = lapile.contenu;
27                  lapile = lapile.suivant;
28                  return x;
29            }
30
31           static int pop(Pile s){
32                   if(s.lapile == null) throw new Error("Pile vide");
33                   int x = s.lapile.contenu;
34                   s.lapile = s.lapile.suivant;
35                   return x;
36           }
```

Programme 3.39 –

Remarque Une autre méthode pour construire des piles est d'utiliser un tableau et un compteur indiquant le sommet de la pile. La procédure `push` est alors simplement `i=i+1; tab[i] = x` (augmenter le compteur et stocker l'élément à la nouvelle place) et la fonction `pop` est alors `i=i+1; return tab[i+1]` (diminuer le compteur et renvoyer l'élément suivant). L'inconvénient de cette technique est bien entendu d'obliger dès le départ à estimer la taille nécessaire sous peine de débordement de la pile et à réserver toute cette place, même si on n'en a pas vraiment besoin tout au long du déroulement du programme. On peut également réallouer le vecteur lors d'un débordement.

3.2.3 Introduction aux piles FIFO

Elles sont plus difficiles à construire car on ne dispose pas directement de structure dynamique de type FIFO. Nous allons décrire un moyen utilisé par exemple dans les *buffers* ou *tampons* de claviers (emplacements mémoires destinés à recevoir les codes des touches sur lesquelles vous avez appuyé, en attendant que le logiciel puisse les traiter).

Si l'on sait que l'on n'aura pas plus de n objets à stocker on utilise un tableau $a_0, \ldots, a_{n-1}$ et deux indices indiquant le début et la fin de la pile. On stocke les éléments de manière circulaire dans la pile, c'est à dire que lorsque l'on déborde vers la droite, on revient à gauche de la pile ; autrement dit, on considère ces indices modulo n. On obtient alors les déclarations suivantes :

```
1  public class FIFO{
2            int debut, fin;
3            int[] contenu;
4            int taille;
5
6            FIFO(int taille){
7                  contenu = new int[taille+1];
8                  debut = 0;
9                  fin = 0;
10                 this.taille = taille;
11           }
12
13 }
```

Programme 3.40 –

Dans cette déclaration, `pf_contenu` désigne le tableau de stockage de la pile, `taille` la taille de ce tableau stockée une fois pour toute pour éviter de la recalculer à chaque empilage ou dépilage, `debut` pointe sur le premier élément entré (donc celui qui sera le premier à sortir) et `fin` l'indice de la case qui suit le dernier entré. Nous avons utilisé une petite astuce qui consiste à ajouter 1 à la taille souhaitée et à travailler modulo $n + 1$. Ceci permet de distinguer aisément

une pile vide (quand `debut=fin`) d'une pile pleine (quand `debut` désigne la case d'après la case d'indice `fin`), mais en sacrifiant une place dans la pile.

```
1  static void push_FIFO(FIFO s, int x){
2          int nouvelle_fin = (s.fin +1) % s.taille;
3              if( nouvelle_fin = s.debut ) throw new Error( "debordement de pile");
4              s.contenu[s.fin] = x;
5              s.fin = nouvelle_fin;
6  }
7
8  static int pop_FIFO(FIFO s){
9          int nouveau_deb = s.debut +1 % s.taille
10         int x = s.contenu.(s.debut);
11             if( s.in = s.debut ) throw new Error("pile vide");
12             s.debut = nouveau_deb ;
13             return x;
14 }
```

Programme 3.41 –

3.3 Expressions algébriques postfixées

La notation habituelle des expressions algébriques, sous forme dite *infixe*, où les opérateurs (addition, multiplication, soustraction, division) figurent entre leurs deux opérandes, souffre *a priori* d'une grande ambiguïté si l'on n'introduit pas de priorités entres les opérateurs. C'est ainsi que la notation $2 + 3 * 4$ peut aussi bien désigner $2 + (3 * 4) = 14$ que $(2 + 3) * 4 = 20$. Des parenthèses ou des règles de priorité sont donc nécessaires pour lever cette ambiguïté. Nous allons étudier ici une autre notation, appelée notation algébrique postfixée ou encore notation polonaise inversée qui ne souffre pas de ces inconvénients. Cette notation est utilisée par certains langages de programmation comme Forth ou certaines calculatrices.

3.3.1 Syntaxe

Soit E un ensemble (les *nombres*), $\mathcal{O}$ un ensemble d'applications de $E \times E$ dans E (les *opérateurs*) et $\mathcal{F}$ un ensemble d'applications de E dans E (les *fonctions*). On considère l'ensemble $\mathcal{E}$ des suites $a_1 \ \ldots \ a_n$ où les a_i sont dans $E \cup \mathcal{O} \cup \mathcal{F}$, qu'on appellera l'ensemble des expressions algébriques.

Exemple On pourra prendre $E = \mathbb{R}$, $\mathcal{O} = \{+, -, *, /\}$ et $\mathcal{F} = \{\sin, \cos, \exp, \log\}$. Les suites ci-dessous sont des expressions algébriques

- 2 + 3 * 4
- 2 3 + sin 4 * 5 6 + -
- + 3 5 * sin cos

Définition 3.3.1 *On appelle ensemble des expressions algébriques postfixées le sous ensemble $\mathcal{EP}$ de $\mathcal{E}$ construit inductivement par les règles suivantes*

- *pour tout élément $a \in E$, la suite à un élément a est une expression algébrique postfixée*
- *si $A = a_1 \ \ldots \ a_m$ et $B = b_1 \ \ldots \ b_n$ sont deux expressions algébriques postfixées et c un opérateur, alors $A\ B\ c = a_1 \ \ldots \ a_m\ b_1 \ \ldots \ b_n\ c$ est encore une expression algébrique postfixée*
- *si $A = a_1 \ \ldots \ a_m$ est une expression algébrique postfixée et f une fonction, alors $A\ f = a_1 \ \ldots \ a_m\ f$ est encore une expression algébrique postfixée.*

Proposition 3.3.1 *Toute expression algébrique postfixée commence par un nombre et toute expression algébrique postfixée de longueur strictement supérieure à 1 se termine par un opérateur ou par une fonction.*

Démonstration: par induction structurelle en appliquant les règles de construction précédentes. Les expressions algébriques de longueur 1 sont les suites n'ayant qu'un seul élément, donc leur premier élément est un nombre ; de plus si A commence par un nombre, aussi bien $A\ B\ c$ que $A\ f$ commencent par un nombre. Ceci montre que toute expression algébrique postfixée commence par un nombre. D'autre part, le seul moyen d'obtenir une expression algébrique postfixée de longueur strictement supérieure à 1

est d'appliquer une des deux dernières règles de construction, et donc l'expression algébrique postfixée se termine forcément par un opérateur ou une fonction.

Exemple On reprend les expressions de l'exemple précédent
- 2 + 3 * 4 n'est pas une expression algébrique postfixée, car elle est de longueur plus grande que 1 et se termine par un nombre
- 2 3 + sin 4 * 5 6 + - est une expression algébrique postfixée comme le montre le parenthésage suivant qui donne les différentes étapes de la construction ((((2 3 +) sin) 4 *) (5 6 +) -)
- + 3 5 * sin cos n'est pas une expression algébrique postfixée car elle commence par un opérateur
- 2 + n'est pas une expression algébrique postfixée bien qu'elle commence par un nombre et se termine par un opérateur

Ce dernier exemple montre qu'il n'est pas facile de caractériser immédiatement les expressions algébriques postfixées. Nous allons travailler en deux temps et introduire les deux définitions suivantes

Définition 3.3.2 *Soit* $p : E \cup \mathcal{O} \cup \mathcal{F} \to \mathbb{N}$ *définie par* $p(x) = 1$ *si* $x \in E$, $p(c) = -1$ *si* $c \in \mathcal{C}$ *et* $p(f) = 0$ *si* $f \in \mathcal{F}$. *Si* $A = a_1 \ \ldots \ a_m$ *est une expression algébrique, on appelle poids de* A *le nombre entier noté* $p(A)$ *défini par* $p(A) = \sum_{i=1}^{m} p(a_i)$.

Définition 3.3.3 *Soit* $A = a_1 \ \ldots \ a_m$ *une expression algébrique. On appelle préfixes stricts de* A *les expressions algébriques* $A_i = a_1 \ \ldots \ a_i$ *pour* $1 \leq i \leq m-1$.

Proposition 3.3.2 *Soit* A *une expression algébrique postfixée. Alors* A *est de poids égal à 1 et tout préfixe de* A *a un poids supérieur ou égal à 1.*

Démonstration: par récurrence sur la longueur de A. C'est clair si A est de longueur 1, puisqu'alors $A = a$ avec $a \in E$ et que $p(A) = p(a) = 1$; de plus A n'a dans ce cas aucun préfixe strict.

Supposons que A est construit par application de la règle 2 ; on a alors $A = B\ C\ c$ où B et C sont des expressions algébriques postfixées (de longueur strictement inférieure) et c un opérateur ; l'hypothèse de récurrence donne alors $p(A) = p(B) + p(C) + p(c) = 1 + 1 - 1 = 1$. Soit A' un préfixe strict de A. Alors soit A' est un préfixe de B ou même B, auquel cas par l'hypothèse de récurrence $p(A') \geq 1$, soit $A' = B\ C'$ où C' est ou bien un préfixe strict de C ou bien C ; alors par l'hypothèse de récurrence $p(C') \geq 1$, soit $p(A') = p(B) + p(C') = 1 + p(C') \geq 2$.

Supposons A construit par application de la règle 3 ; on a alors $A = B\ f$ où B est une expression algébrique postfixée (de longueur strictement inférieure) et f une fonction. Alors $p(A) = p(B) + p(f) = 1 + 0 = 1$ par l'hypothèse de récurrence. De plus si A' est un préfixe strict de A, c'est soit B de poids 1, soit un préfixe de B qui par hypothèse de récurrence est de poids supérieur ou égal à 1.

Théorème 3.3.1 *Soit* A *une expression algébrique postfixée de longueur strictement supérieure à 1.*
- *Si* $A = B\ C\ c$ *où* B *et* C *sont des expressions algébriques postfixées et* c *un opérateur, alors* B *est le plus long préfixe strict de* A *de poids 1*
- *Si* $A = B\ f$ *où* B *est une expression algébrique postfixée et* f *une fonction, alors* B *est le plus long préfixe strict de* A *de poids 1*

Démonstration: la deuxième assertion est triviale puisque B est le plus long préfixe strict de A et qu'il est de poids 1 d'après la proposition précédente. Montrons maintenant la première. On sait que B est un préfixe strict de A et qu'il est de poids 1 (d'après la proposition précédente). Soit A' un préfixe strict de A, strictement plus long que B. Alors $A' = B\ C'$ où C' est un préfixe strict de C ou bien C lui même ; on sait alors que $p(C') \geq 1$ et donc $p(A') = p(B) + p(C') = 1 + p(C') \geq 2$. Donc B est bien le plus long préfixe strict de poids 1.

La proposition précédente montre que, contrairement à la notation infixe, la notation postfixée ne présente aucune ambiguïté, ce que nous allons traduire sous forme de corollaire.

Corollaire 3.3.1 *L'écriture d'une expression algébrique postfixée de longueur strictement supérieure à 1 sous l'une des deux formes*
- $A = B\ C\ c$ *où* B *et* C *sont des expressions algébriques postfixées et* c *un opérateur*

– $A = B\ f$ *où* B *est une expression algébrique postfixée et* f *une fonction*

est unique.

Démonstration: en effet dans les deux cas, B est parfaitement caractérisée par le fait d'être le plus long préfixe strict de poids 1 et dans le premier cas, C n'est autre que A privé de B et du dernier élément de A.

Nous allons revenir à la caractérisation des expressions algébriques postfixées.

Théorème 3.3.2 *Une expression algébrique A est une expression algébrique postfixée si et seulement si elle est de poids 1 et tout préfixe strict a un poids supérieur ou égal à 1.*

Démonstration: nous avons déjà vu que la condition est nécessaire. Montrons qu'elle est suffisante. Soit $A = a_1\ \dots\ a_m$ une expression algébrique de poids 1 telle que tout préfixe strict soit de poids supérieur ou égal à 1. Nous allons montrer par récurrence sur m que A est une expression algébrique postfixée. Si $m = 1$, on a $1 = p(A) = p(a_1)$ donc a_1 est un nombre et A est bien une expression algébrique postfixée. Si $m > 1$, remarquons que si $A' = a_1\ \dots\ a_{m-1}$, alors A' est un préfixe strict de A donc $p(A') \geq 1$. Comme $p(A) = p(A') + p(a_m) = 1$, on a nécessairement $p(a_m) \leq 0$, donc a_m est soit une fonction, soit un opérateur.

Si a_m est une fonction f, on a $p(A') = 1$; comme tout préfixe strict de A' est un préfixe strict de A, il a un poids supérieur ou égal à 1, donc par l'hypothèse de récurrence, A' est une expression algébrique postfixée, soit $A = A'\ f$ est aussi un expression algébrique postfixée.

Si a_m est un opérateur c, on a $p(A') = 2$. A admet au moins un préfixe strict de poids 1, à savoir a_1. Soit donc B un préfixe strict de A de poids 1 de longueur maximale ; comme tout préfixe strict de B est un préfixe strict de A, par l'hypothèse de récurrence B est une expression algébrique postfixée. Comme de plus $p(A') = 2$, on a $B \neq A'$ soit $A' = B\ C$. On a $2 = p(A') = p(B) + p(C) = 1 + p(C)$, donc $p(C) = 1$. Soit C' un préfixe strict de C. Alors $B\ C'$ est un préfixe strict de A strictement plus long que B, donc $p(B\ C') \geq 1$ et $p(B\ C') \neq 1$, soit $p(B\ C') \geq 2$. Mais $p(C') = p(B\ C') - p(B) = p(B\ C') - 1 \geq 1$; par l'hypothèse de récurrence, C est une expression algébrique postfixée. Mais alors $A = B\ C\ c$ est encore une expression algébrique postfixée.

3.3.2 Sémantique

Pour le moment, nos expressions algébriques postfixées sont uniquement des objets formels, des suites de symboles, sans signification. Nous avons seulement appris à les reconnaître et à les manipuler, c'est-à-dire leur syntaxe. Nous allons maintenant donner un sens à ces objets formels, c'est à dire leur définir une sémantique, à travers le théorème suivant.

Théorème 3.3.3 *Il existe une unique application* Ev *de l'ensemble des expressions algébriques postfixées dans l'ensemble E définie inductivement par*

– $\mathrm{Ev}(A) = a$ *si* $A = a$ *avec* $a \in E$
– $\mathrm{Ev}(A) = c\big(\mathrm{Ev}(B), \mathrm{Ev}(C)\big)$ *si* $A = B\ C\ c$ *où* B *et* C *sont des expressions algébriques postfixées et* c *un opérateur*
– $\mathrm{Ev}(A) = f\big(\mathrm{Ev}(B)\big)$ *si* $A = B\ f$ *où* B *est une expression algébrique postfixée et* f *une fonction*

Démonstration: ceci découle immédiatement de l'unicité de l'écriture d'une expression algébrique postfixée sous l'une des trois formes, et du chapitre sur la récursion et les définitions inductives de fonctions.

Définition 3.3.4 *L'application* Ev *est appelée l'évaluation. L'élément de E,* $\mathrm{Ev}(A)$ *est appelé l'évaluation de l'expression algébrique postfixée A.*

3.3.3 Technique d'évaluation

La méthode précédente pour l'évaluation d'une expression algébrique postfixée se heurte à la difficulté de déterminer B et C pour le cas où $A = B\ C\ c$ avec c un opérateur. La définition de B comme le plus long préfixe de poids 1 se prête mal à un algorithme. Nous allons donc donner une méthode plus pratique utilisant une pile.

Nous allons définir une suite d'applications de l'ensemble des expressions algébriques postfixées dans l'ensemble des suites d'éléments de E par récurrence de la manière suivante. Soit $A = a_1\ \dots\ a_m$ une expression algébrique postfixée.

- $f_1(A) = a_1$
- Posons $f_i(A) = b_1 \ \dots \ b_k$; alors $f_{i+1}(A)$ est défini par
  - $f_{i+1}(A) = f_i(A) = b_1 \ \dots \ b_k$ si $i \geq m$
  - $f_{i+1}(A) = f_i(A)\ a_{i+1} = b_1 \ \dots \ b_k \ a_{i+1}$ si $a_{i+1} \in E$
  - $f_{i+1}(A) = b_1 \ \dots \ b_{k-2}\ c(b_{k-1}, b_{k-2})$ si $a_{i+1} = c$ est un opérateur (défini si $k \geq 2$)
  - $f_{i+1}(A) = b_1 \ \dots \ b_{k-1}\ f(b_k)$ si $a_{i+1} = f$ est une fonction

Lemme 3.3.1 *Pour tout $i \leq m$, $f_i(A)$ est défini et la longueur de $f_i(A)$ est égale au poids de $a_1 \ \dots \ a_i$.*

Démonstration: nous allons le montrer par récurrence sur i. C'est clair pour $i = 1$. Supposons que ce soit vrai pour $i \leq m-1$ et montrons le pour $i+1$.

Si $a_{i+1} \in E$, alors il est clair que $f_{i+1}(A)$ est définie et sa longueur est égale à la longueur de $f_i(A)$ plus 1, soit, par l'hypothèse de récurrence, au poids de $a_1 \ \dots \ a_i$ augmenté de $1 = p(a_{i+1})$, donc au poids de $a_1 \ \dots \ a_{i+1}$.

Si a_{i+1} est un opérateur, comme on sait que $p(a_1 \ \dots \ a_{i+1}) \geq 1$ et que $p(a_{i+1}) = -1$, on a $p(a_1 \ \dots \ a_i) \geq 2$; par l'hypothèse de récurrence ce nombre est aussi égal à la longueur de $f_i(A)$ ce qui montre que $k \geq 2$, donc que $f_{i+1}(A)$ est bien définie. Mais alors la longueur de $f_{i+1}(A)$ est, d'après sa définition, égale à la longueur de $f_i(A)$ diminuée de 1, donc au poids de $a_1 \ \dots \ a_i$ diminué de 1, donc au poids de $a_1 \ \dots \ a_{i+1}$.

Enfin, si a_{i+1} est une fonction, alors il est clair que $f_{i+1}(A)$ est définie et sa longueur est égale à la longueur de $f_i(A)$, soit, par l'hypothèse de récurrence, au poids de $a_1 \ \dots \ a_i$, donc au poids de $a_1 \ \dots \ a_{i+1}$ (puisque $p(a_{i+1} = 0)$. Ceci achève la démonstration du lemme.

Théorème 3.3.4 *Soit A une expression algébrique postfixée de longueur m. Alors $f_m(A) = \mathrm{Ev}(A)$.*

Démonstration: par récurrence sur m. Posons $A = a_1 \ \dots \ a_m$. C'est clair si $m = 1$ puisqu'alors $f_1(A) = a_1 = \mathrm{Ev}(A)$. Supposons le résultat vrai pour toute expression algébrique postfixée de longueur strictement inférieure à m. Si $a_m = f$ est une fonction, alors $A = B\ f$. Par hypothèse de récurrence $f_{m-1}(A) = f_{m-1}(B) = \mathrm{Ev}(B)$ et donc $f_m(A) = f\big(\mathrm{Ev}(B)\big) = \mathrm{Ev}(A)$.

Si $a_m = c$ est un opérateur, alors $A = B\ C\ c$ où B et C sont des expressions algébriques postfixées. Appelons p la longueur de B, q la longueur de C si bien que $m = p + q + 1$. Pour tout $i \leq p$, on a $f_i(A) = f_i(B)$ et en particulier pour $i = p$, par l'hypothèse de récurrence, $f_p(A) = f_p(B) = \mathrm{Ev}(B)$.

On montre maintenant par récurrence sur $i \in [1, q]$ que $f_{p+i}(A) = \mathrm{Ev}(B)\ f_i(C)$. C'est clair si $i = 1$; supposons le donc pour i et distinguons suivant le type de a_{p+i+1}. Si $a_{p+i+1} \in E$, alors

$$f_{p+i+1}(A) = f_{p+i}(A)\ a_{p+i+1} = \mathrm{Ev}(B)\ f_i(C)\ a_{p+i+1} = \mathrm{Ev}(B)\ f_{i+1}(C)$$

Si a_{p+i+1} est une fonction f, posons $f_i(C) = b_1 \ \dots \ b_k$ avec $k \geq 1$; on a $f_{p+i}(A) = \mathrm{Ev}(B) \ = b_1 \ \dots \ b_k$ et donc

$$f_{p+i+1}(A) = \mathrm{Ev}(B)\ b_1 \ \dots \ f(b_k) = \mathrm{Ev}(B)\ f_{i+1}(C)$$

Si maintenant a_{p+i+1} est un opérateur d, posons $f_i(C) = b_1 \ \dots \ b_k$ avec $k \geq 1$; comme B est le plus grand préfixe de poids 1, le poids de $a_1 \ \dots \ a_p\ a_{p+1} \ \dots a_{p+i+1}$ est au mois égal à 2 et comme $p(a_{p+i+1}) = -1$, le poids de $a_1 \ \dots \ a_p\ a_{p+1} \ \dots a_{p+i}$ est au mois égal à 3 ; mais ce nombre est aussi la longueur de $f_{p+i}(A) = \mathrm{Ev}(B)\ f_i(C)$ (par l'hypothèse de récurrence). On a donc $k \geq 2$; on en déduit que

$$f_{p+i+1}(A) = \mathrm{Ev}(B)\ b_1 \ \dots \ b_{k-2}\ d(b_{k-1}, b_k) = \mathrm{Ev}(B)\ f_{i+1}(C)$$

Pour $i = q$, on a donc $f_{m-1}(A) = \mathrm{Ev}(B)\ f_q(C) = \mathrm{Ev}(B)\ \mathrm{Ev}(C)$ par l'hypothèse de récurrence. On a alors $f_m(A) = c\big(\mathrm{Ev}(B), \mathrm{Ev}(C)\big) = \mathrm{Ev}(A)$.

Dans tous les cas, on a bien $f_m(A) = \mathrm{Ev}(A)$.

Remarque Cette méthode fournit en même temps un vérificateur de la syntaxe de l'expression algébrique A. D'après la caractérisation des expressions algébriques postfixées, A est une expression algébrique postfixée si et seulement si $f_m(A)$ est définie (ce qui correspond à tout préfixe strict est de poids supérieur ou égal à 1) et $f_m(A)$ est de longueur 1 (ce qui correspond à $p(A) = 1$).

Pour écrire une procédure Java, nous allons utiliser une structure de liste pour l'expression algébrique : l'expression algébrique $a_1 \ \ldots \ a_m$ sera entrée dans une liste Java $[a_1, \ldots, a_m]$ ce qui permettra d'en extraire les éléments a_i dans l'ordre de leur apparition. Pour permettre d'entrer dans cette liste aussi bien des nombres (ici réels) que des opérateurs (ici $+, -, *, /$) et des fonctions (ici $\sin, \cos, \exp, \log$), nous définirons un type construit `eap_elt` (pour *élément* d'une *e*xpression *a*lgébrique *p*ostfixée) qui pourra être de l'un des ces trois types, les opérateurs étant symbolisés par des caractères et les fonctions par leur nom :

```
static public class EapElt{
               enum type_eap_elt {Nb, Op, Fct};
               type_eap_elt type;

               public class EapNb extends EapElt{
                    double x;

                    EapNb(double x){
                        super(type_eap_elt.Nb);
                         this.x = x;
                     }
                 }

                 class EapOp extends EapElt{
                     String op;

                     EapOp(String op){
                           super(type_eap_elt.Op);
                           this.op = op;

                       }
                  }

                 class EapFct extends EapElt{
                    String fct;

                     EapFct(String fct){
                           super(type_eap_elt.Fct);;
                           this.fct = fct;
                       }

                  }
}
```

Programme 3.42 –

On peut alors définir une EAP comme une liste d'éléments

```
class Eap{
       Node  premier;
       class Node{
           EapElt eapelt;
           Node suivant;
           Node(EapElt eapelt, Node suivant){
               this.eapelt = eapelt;
               this.suivant = suivant;
            }
        }

        Eap(Node n){premier = n; }

        boolean estVide(){ return premier==null;}
        EapElt tete(){ return premier.eapelt;}
        Eap queue(){ return new Eap(premier.suivant); }
}
```

Programme 3.43 –

Les manipulations à faire pour calculer les $f_i(A)$ sont clairement d'adjoindre un élément a_{i+1}, de retirer deux éléments b_{k-1} et b_k puis d'adjoindre $c(b_{k-1}, b_k)$, de retirer b_k et d'adjoindre $f(b_k)$. Une structure de pile est donc idéale pour cela, puisque ce sont toujours les derniers éléments entrés que l'on doit retirer. Ceci conduit directement à la fonction suivante où nous avons défini une pile de nombres réels , initialement vide, et deux fonctions adaptées `push` (qui adjoint un élément à la pile) et `pop` (qui retire un élément de la pile et renvoie la valeur de cet élément).

```
class Pile{
           Node  premier;
           class Node{
```

```
4                       double x;
5                       Node suivant;
6                       Node(double x,Node suivant){
7                               this.x = x;
8                               this.suivant = suivant;
9                           }
10                  }
11                  Pile(){premier = null;}
12
13                  boolean estVide(){ return premier==null;}
14                  void push(double x){
15                          premier = new Node(x,premier);
16                  }
17                  double pop(){
18                          double x=premier.x;
19                          premier=premier.suivant;
20                          return x;
21                  }
22  }
```

Programme 3.44 –

La procédure `evalue_aux` effectue tout le travail de manière récursive sur l'expression algébrique. L'invariant évident de ces appels récursifs est qu'après le i-ième appel de la fonction `eval`, B contient $[a_{i+1}; \ldots; a_m]$ et `accu` contient $[f_i(A)] = [b_k; \ldots; b_1]$ (dans l'ordre inverse puisque les listes Java ont pour premier élément le dernier entré) ; la terminaison est garantie par la stricte décroissance de la liste B ; tout ceci démontre la validité de la fonction.


```
1   private static void evalue_aux(Eap A, Pile lapile){
2             if(A.estVide()) return;
3             EapElt premier=A.tete();;
4             switch(premier.type){
5                   case Nb : lapile.push(((EapElt.EapNb)premier).x); evalue_aux(A.queue(),lapile); break;
6                   case Fct :
7                         double x = lapile.pop();
8                         String nom =((EapElt.EapFct)premier).fct;
9                         if(nom=="sin") lapile.push(Math.sin(x));
10                        else if(nom=="cos") lapile.push(Math.cos(x));
11                        else if(nom=="exp") lapile.push(Math.exp(x));
12                        else if(nom=="log") lapile.push(Math.log(x));
13                        else throw new Error("fonction inconnue");
14                        break;
15                  case Op:
16                        double x1=lapile.pop();
17                        double x2=lapile.pop();
18                        nom =((EapElt.EapOp)premier).op;
19                        if(nom=="+") lapile.push(x1+x2);
20                        else if(nom=="-") lapile.push(x2-x1);
21                        else if(nom=="*") lapile.push(x2*x1);
22                        else if(nom=="/") lapile.push(x2/x1);
23                        else throw new Error("operateur inconnu");
24            }
25      }
26  public double evalue_eap(Eap A){
27            Pile lapile = new Pile();
28            evalue_aux(A,lapile);
29            if(lapile.estVide()) throw new Error("erreur");
30            double x = lapile.pop();
31            if(!lapile.estVide()) throw new Error("erreur");
32            return x;
33  }
```

Programme 3.45 –

Avec quelques essais :

```
1   evalue_eap : eap_elt list -> float = <fun>
2   #evalue_eap [Nb 2. ; Nb 3. ; Nb 4. ; Op '+'; Op '*'];
3   ;;
4   - : float = 14
5   #evalue_eap [Nb 2. ; Nb 3. ; Fct "sin" ; Op '+'; Op '*'];
6   ;;
7   Uncaught exception: Failure "erreur de syntaxe: trop d'opérateurs"
8   #evalue_eap [Nb 2. ; Nb 3. ; Fct "sin" ; Op '+'];
9   ;;
10  - : float = 2.14112000806
```

Programme 3.46 –

Cette fonction d'évaluation peut être facilement enrichie avec toutes les fonctions et tous les opérateurs usuels.

Remarque les puristes objecteront que la fonction précédente a un double rôle : l'un de vérification de la syntaxe de l'expression algébrique, l'autre d'évaluation de cette même expression ; de ce point de vue, il y a confusion entre la syntaxe et la sémantique. Mais ceci est inhérent aux expressions algébriques postfixées : l'évaluation de l'expression en vérifie en même temps la correction. Une deuxième approche plus générale sera vue à propos des arbres et des expressions algébriques *infixes* : dans ce cas il y a séparation claire entre l'analyse syntaxique (qui consiste à construire l'arbre de l'expression) et l'évaluation sémantique.

Chapitre 4

Arbres

4.1 Généralités sur les arbres

4.1.1 Graphes et arbres

Définition 4.1.1 *On appelle graphe un couple $G = (V, E)$ d'un ensemble fini V (les sommets ou noeuds) et d'une partie E de $V \times V$ (les arêtes). Si $x, y \in V$, on note $x \to y$ pour $(x, y) \in E$. On dit que le graphe est non orienté si*

$$\forall x, y \in V, \quad x \to y \iff y \to x$$

Définition 4.1.2 *Soit G un graphe non orienté. Une chaîne dans un graphe est une suite de sommets reliés par des arêtes. La longueur d'une chaîne est le nombre d'arêtes utilisées, ou, ce qui revient au même, le nombre de sommets utilisés moins un. Une chaîne est dite simple si elle ne visite pas deux fois le même sommet (sauf éventuellement en ses extrémités).*

Définition 4.1.3 *Soit G un graphe non orienté. On appelle cycle une chaîne simple dont les extrémités coïncident, de longueur différente de 2. On ne rencontre pas deux fois le même sommet, sauf celui choisi comme sommet de départ et d'arrivée.*

Définition 4.1.4 *Soit G un graphe non orienté. La relation $\mathcal{R}$ définie par $a\mathcal{R}b$ signifie "il existe une chaîne d'origine a et d'extrémité b" est une relation d'équivalence sur l'ensemble V. Les classes d'équivalence sont appelées les composantes connexes du graphe. on dit que le graphe est connexe s'il possède une seule classe d'équivalence.*

Théorème 4.1.1 *Pour tout graphe ayant m arêtes, n sommets et p composantes connexes, on a $m - n + p \geq 0$. De plus, on a $n = m + p$ si et seulement si G est sans cycle.*

Démonstration:

Par récurrence sur le nombre n d'arêtes. Si $n = 0$, G comporte exactement n composantes connexes, donc $m = p$ et le résultat est vérifié. Soit G un graphe ayant m arêtes, n sommets et p composantes connexes et supprimons une arête $a \leftrightarrow b$. Le nouveau graphe G_1 comporte $m_1 = m - 1$ arêtes, $n_1 = n$ sommets et p ou $p + 1$ composantes connexes (la composante connexe commune de a et b a pu exploser en 2). Par hypothèse de récurrence, on a $m_1 + p_1 \geq n_1$ soit encore $m - 1 + p_1 \geq n$, soit encore $m + (p_1 - 1) \geq n$. Mais p_1 est soit égal à p, auquel cas $m + p - 1 \geq n$ et a fortiori $m + p \geq n$, soit à $p + 1$ auquel cas $m + p \geq n$, ce qui achève la récurrence.

Supposons maintenant que l'on a égalité. On se trouve forcément dans le second cas, ce qui montre que, dès que l'on retire une arête $a \leftrightarrow b$, le nombre de composantes connexes augmente de 1, autrement dit aucune arête ne peut faire partie d'un cycle, donc le graphe est sans cycle.

Inversement, supposons le graphe sans cycle. Alors le graphe G_1 est sans cycle et par récurrence vérifie $m_1 + p_1 = n_1$. Mais le graphe, G ne comportant pas de cycle, les composantes connexes de a et b dans G_1 sont disctinctes, donc $p_1 = p + 1$. On a donc $m + p = n$.

Le théorème précédent justifie l'équivalence des propriétés suivantes qui caractérisent un arbre :

Définition 4.1.5 *On dit qu'un graphe non orienté $G = (V, E)$ est un arbre s'il vérifie les propriétés équivalentes*
(i) G est sans cycle et connexe.
(ii) G est sans cycle et $|V| = |E| + 1$
(iii) G est connexe et $|V| = |E| + 1$

Définition 4.1.6 *On appelle forêt un graphe non orienté sans cycle. Dans ce cas, ses composantes connexes sont des arbres.*

Remarque Dans un arbre, il existe un unique chemin reliant un sommet a à un sommet b.

4.1.2 Arbres enracinés

Définition 4.1.7 *On appelle arbre enraciné tout couple (r, G) d'un arbre $G = (V, E)$ et d'un sommet $r \in V$ (appelé la racine de l'arbre).*

Définition 4.1.8 *Soit (r, V, E) un arbre enraciné, $x, y \in V$ distincts. On dit que y est un descendant de x (ou que x est un ascendant de y) s'il existe un chemin (nécessairement unique) de r à y passant par x.*

Remarque La relation "x est un ascendant de y" est visiblement une relation d'ordre strict partielle sur V.

Définition 4.1.9 *On dit que x est père de y (ou que y est fils de x) si x est un ascendant de y tel que $x \leftrightarrow y$.*

Remarque Tout élément x de V distinct de r possède un unique père. C'est le dernier sommet visité dans l'unique chemin reliant la racine à x.

Définition 4.1.10 *On appelle feuille de l'arbre (ou encore noeud externe), un noeud sans descendant. On appelle noeud interne de l'arbre, un noeud qui possède des descendants.*

Remarque Considérons $G = (r, V, E)$ un arbre enraciné. Soit $V_1 = V \setminus \{r\}$ et $E_1 = E \setminus \{r \leftrightarrow x \mid r \leftrightarrow x \in E\}$. Alors (V_1, E_1) est une forêt, chacun des arbres qui la compose comportant un et un seul fils de r. On peut donc considérer que cette forêt est une forêt d'arbres enracinés par les fils de r. Ceci nous conduit à la définition inductive suivante :

Définition 4.1.11 *On appelle ensemble des arbres enracinés l'ensemble défini inductivement par*
(i) Tout singleton est un arbre enraciné (réduit à une feuille)
(ii) Toute réunion disjointe d'arbres enracinés est une forêt enracinée
(iii) Tout couple (r, F) d'un élément r et d'une forêt enracinée F est un arbre enraciné.

Définition 4.1.12 *Soit $A = (r, V, E)$ un arbre enraciné, V_i l'ensemble de ses noeuds internes et V_e l'ensemble de ses feuilles ou noeuds externes. On appelle étiquetage de A tout couple (f, g) d'une application f de V_i dans un ensemble N et d'une application g de V_e dans un ensemble F. On dit que l'étiquetage est homogène si $N = F$, hétérogène si $N \neq F$.*

Définition 4.1.13 *On dit qu'un arbre enraciné est ordonné si, pour tout noeud interne, l'ensemble de ses fils est totalement ordonné (de l'aîné au cadet).*

Définition 4.1.14 *Dans toute la suite de ce chapitre, on désignera par arbre hétérogène (resp. homogène) un arbre enraciné ordonné muni d'un étiquetage hétérogène (resp. homogène).*

4.1.3 Représentation graphique d'un arbre

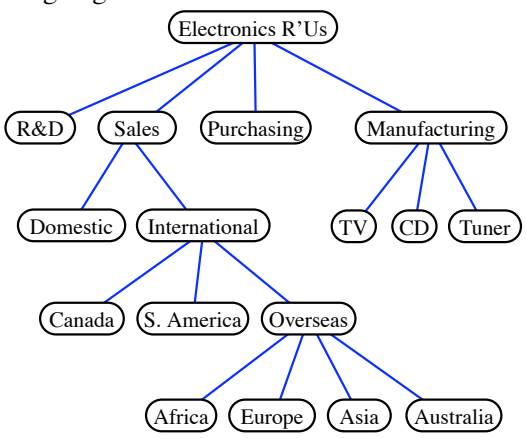
Il est d'usage de ne pas faire figurer dans les arbres les noms des éléments, mais au contraire d'y faire figurer leur étiquette (qui peut être de nature différente suivant que les noeuds de l'arbre sont externes ou internes).

On adopte une représentation *généalogique* de l'arbre avec la racine en haut, ses fils en dessous ordonnés de gauche à droite, les fils des fils en dessous, et ainsi de suite.

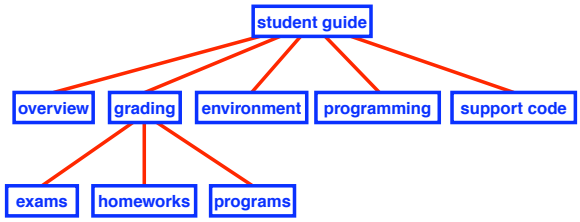
4.1.4 Exemples d'arbres

L'organigramme d'une société :

- organigramme d'une société



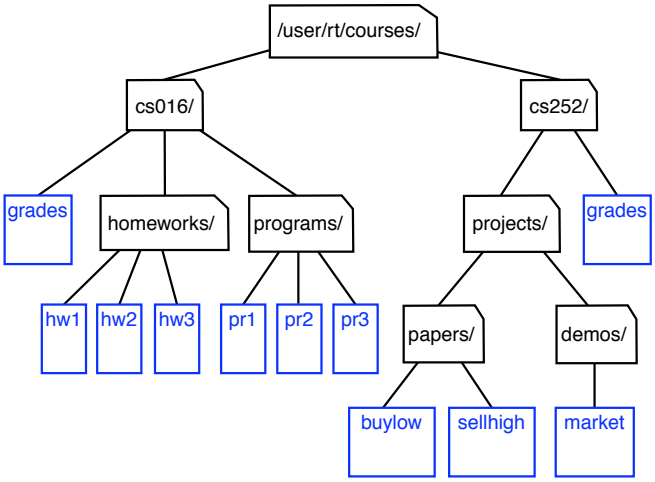
- table des matières d'un livre



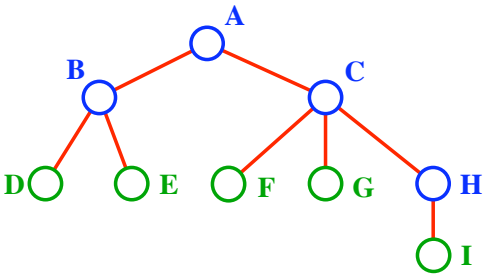
La table des matières d'un livre :

Le contenu d'un disque dur

• Système de fichiers Unix ou DOS/Windows



4.1.5 Terminologie



- A est la racine de l'arbre
- B est le père de D et E.
- C est le frère de B
- D et E sont les fils de B.
- D, E, F, G, I sont des noeuds externes ou des feuilles
- A, B, C, H sont des noeuds internes.
- La profondeur du noeud E est2
- La hauteur de l'arbre est 3.
- Le degré du noeud B est2.

4.1.6 Typages des arbres hétérogènes en Java

Les différents typages diffèrent essentiellement en la manière de typer les forêts comme ensembles d'arbres :
- totalement dynamique : sous forme de liste d'arbres
- semi-dynamique : sous forme de tableaux d'arbres
- statique : sous forme de n-uplet (sous-entend que le degré des noeuds internes est fixe ou au moins majoré)

```
1 /* définition d'un arbre hétérogène à l'aide de listes */
2 abstract class Arbres<F,N>
3 {
4     boolean interne;
5
6     boolean estNoeud(){ return interne; }
7
```

```
8       boolean estFeuille(){ return !interne;}
9   }
10
11  class Feuille<F,N> extends Arbres<F,N>
12  {
13       F contenu;
14
15       Feuille(F contenu){
16            interne = false;
17            this.contenu=contenu;
18           }
19  }
20
21  class Noeud<F,N> extends Arbres<F,N>
22  {
23      N contenu;
24      Liste<Arbres<F,N>> branches;
25
26      Noeud(N contenu, Liste<Arbres<F,N>> branches){
27           interne=true;
28           this.contenu=contenu;
29           this.branches=branches;
30      }
31  }
32  /*définition d'un arbre hétérogène à l'aide de tableaux */
33  class Noeud<F,N> extends Arbres<F,N>
34  {
35      N contenu;
36     Arbres<F,N>[] branches;
37
38      Noeud(N contenu, Arbres<F,N>[] branches){
39           interne=true;
40           this.contenu=contenu;
41           this.branches=branches;
42      }
43  }
```

Programme 4.1 –

Enfin, si p est un entier fixé et si tous les noeuds de l'arbre ont le même degré p, il peut être intéressant de représenter les ensembles d'arbres à p éléments sous formes de p-uplets.

Définition 4.1.15 *On appelle arbre binaire (resp. ternaire) un arbre dont tous les noeuds ont pour degré 2 (resp. 3).*

On pourra implémenter les arbres binaires hétérogènes en Java, en choisissant de mettre un fils à gauche et un fils à droite de l'étiquette, de la façon suivante :

```
1   class Noeud<F,N> extends Arbre<F,N>
2   {
3       N contenu;
4      Arbre<F,N> gauche;
5      Arbre<F,N> droite;
6
7       Noeud(N contenu, Arbre<F,N> gauche, Arbre<F,N> droite){
8            interne=true;
9            this.contenu=contenu;
10           this.gauche=gauche;
11           this.droite=droite;
12      }
13
14  }
```

Programme 4.2 –

4.1.7 Noeuds, feuilles, arêtes

Théorème 4.1.2 *Tout arbre à n noeuds (internes ou externes) possède $n-1$ arêtes.*

Démonstration: par induction structurelle. Si l'arbre A est réduit à une feuille, il a 1 sommet et 0 arêtes et le résultat est vérifié. Sinon, soit n le nombre de sommets de A, a_0 la racine de A, $A_1, \ldots, A_p$ les branches issues de A ayant pour nombre de sommets respectifs $n_1, \ldots, n_p$. On a $n = n_1 + \ldots + n_p + 1$.

Par hypothèse d'induction, chaque arbre A_i possède $n_i - 1$ arêtes. Or les arêtes de A sont d'une part les arêtes des A_i, d'autre part les p arêtes reliant a_0 aux racines des A_i. En conséquence, A possède $p + (n_1 - 1) + \ldots + (n_p - 1) = n_1 + \ldots + n_p = n - 1$ arêtes, ce qui achève la démonstration.

Théorème 4.1.3 *Soit A un arbre binaire hétérogène. Si A possède p noeuds internes, il possède $p + 1$ feuilles.*

Démonstration: par induction structurelle. Si l'arbre est réduit à une feuille, il possède 0 noeuds et 1 feuille et la propriété est vérifiée. Sinon, soit a_0 la racine de A, n le nombre de sommets de A, A_1 et A_2 les deux branches partant de a_0, n_1 et n_2 le nombre de leurs noeuds, si bien que $n = n_1 + n_2 + 1$. Par hypothèse d'induction, A_1 possède $n_1 + 1$ feuilles et A_2 possède $n_2 + 1$ feuilles, donc A possède $n_1 + n_2 + 2 = n + 1$ feuilles, ce que l'on voulait démontrer.

4.1.8 Implémentation des opérations élémentaires sur les arbres hétérogènes

Une première opération souvent nécessaire est le calcul du nombre de noeuds d'un arbre. Prenons tout d'abord le cas d'un arbre hétérogène dont les noeuds sont étiquetés par N et les feuilles étiquetées par F, défini par :

- si x est un élément de F, alors x est un arbre (réduit à une feuille)
- toute famille d'arbres est une forêt
- tout couple formé d'un élément y de N et d'une forêt est un arbre

Le calcul du nombre de noeuds peut se faire par induction structurelle :

- si l'arbre est réduit à une feuille, il possède 0 noeuds
- le nombre de noeuds d'une forêt est la somme du nombre de noeuds des arbres qui la composent
- le nombre de noeuds d'un arbre non réduit à une feuille est le nombre de noeuds de la forêt des branches de sa racine augmenté de 1 (le noeud racine)

Dans le cas où l'on implémente les forêts comme des listes d'arbres, ceci conduit à une fonction Java doublement récursive, une première récursivité étant due à la structure d'arbre, la deuxième à la structure de liste de la forêt :

```
1  abstract class Arbre<F,N>
2  {
3     abstract int nombreFeuilles();
4  }
5
6  class Feuille<F,N> extends Arbre<F,N>
7  {
8       int nombreFeuilles(){ return 1;}
9
10 }
11
12 class Noeud<F,N> extends Arbre<F,N>
13 {
14     int nombreFeuillesForet(Liste<Arbre<F,N>> foret){
15         if(foret.isEmpty()) return 0;
16         else return foret.tete().nombreFeuilles()+nombreFeuillesForet(foret.reste());
17     }
18
19     int nombreFeuilles(){ return nombreFeuillesForet(branches); }
20
21 }
```

Programme 4.3 –

Dans le cas où l'on implémente les forêts comme des tableaux d'arbres, il est plus naturel d'introduire une boucle pour calculer le nombre de noeuds d'une forêt :

```
1  int nombreFeuillesForet(Arbre<F,N>[] foret){
2        int nb=0;
3        for(int i=0;i<foret.length;i++)
4           nb+=foret[i].nombreFeuilles();
5  }
```

Programme 4.4 –

Enfin dans le cas d'un arbre binaire hétérogène, on peut se passer de la fonction calculant le nombre de noeuds d'une forêt et écrire directement

```
int nombreFeuilles(){
        return gauche.nombreFeuilles()+droite.nombreFeuilles();
}
```

Programme 4.5 –

Le calcul du nombre de feuilles se fait suivant une méthode tout à fait similaire
- si l'arbre est réduit à une feuille, il possède 1 feuille
- le nombre de feuilles d'une forêt est la somme du nombre de feuilles des arbres qui la composent
- le nombre de feuilles d'un arbre non réduit à une feuille est le nombre de feuilles de la forêt des branches de sa racine

Si les forêts sont implémentées comme des listes :

```
abstract class Arbre<F,N>
{
    abstract int nombreNoeuds();
}

class Feuille<F,N> extends Arbre<F,N>
{
     int nombreNoeuds(){ return 0; }

}

class Noeud<F,N> extends Arbre<F,N>
{
    int nombreNoeudsForet(Liste<Arbre<F,N>> foret){
        if(foret.isEmpty()) return 0;
        else return foret.tete().nombreNoeuds()+nombreNoeudsForet(foret.reste());
    }

    int nombreNoeuds(){return 1+ nombreNoeudsForet(branches);}
}
```

Programme 4.6 –

Si les forêts sont implémentées comme des tableaux :

```
int nombreFeuillesForet(Arbre<F,N>[] foret){
       int nb=0;
       for(int i=0;i<foret.length;i++)
          nb+=foret[i].nombreNoeuds();
 }
```

Programme 4.7 –

Dans le cas d'un arbre binaire :

```
int nombreNoeuds(){
        return gauche.nombreNoeuds()+droite.nombreNoeuds();
}
```

Programme 4.8 –

Quant à la hauteur d'un arbre hétérogène, par induction structurelle on obtient
- si l'arbre est réduit à une feuille, il est de hauteur 0
- la hauteur d'une forêt est le maximum des hauteurs des arbres qui la composent
- la hauteur d'un arbre non réduit à une feuille est la hauteur de la forêt des branches de sa racine augmentée de 1

Si les forêts sont implémentées comme des listes :

```
abstract class Arbre<F,N>
{
    abstract int hauteur();
}

class Feuille<F,N> extends Arbre<F,N>
{
     int hauteur(){ return 0;  }
}

```

```
11 class Noeud<F,N> extends Arbre<F,N>
12 {
13     private int max(int x, int y){ return x<y ? y : x;  }
14
15     int hauteurForet(Liste<Arbre<F,N>> foret){
16         if(foret.isEmpty()) return 0;
17         else return max(foret.tete().hauteur(),hauteurForet(foret.reste()));
18     }
19
20     int hauteur(){ return 1+hauteurForet(branches); }
21 }
```

Programme 4.9 –

Si les forêts sont implémentées comme des tableaux, on utilise une boucle :

```
1 int hauteurForet(Arbre<F,N>[] foret){
2         int nb=0;
3         for(int i=0;i<foret.length;i++)
4             nb=max(nb,foret[i].hauteur());
5  }
```

Programme 4.10 –

Dans le cas d'un arbre binaire, on peut écrire :

```
1 int hauteur(){ return 1+max(gauche.hauteur(),droite.hauteur()); }
```

Programme 4.11 –

4.2 Arbres binaires

4.2.1 Arbres binaires hétérogènes

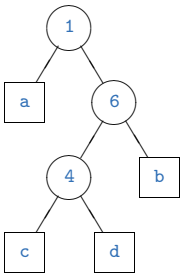
Un arbre binaire possède des noeuds et des feuilles. Certains disent plutôt noeuds internes et noeuds externes. On peut définir la structure d'arbre binaire de façon récursive. Si N (resp. F) désigne l'ensemble des valeurs des noeuds (resp. des valeurs des feuilles), l'ensemble A(N,F) des arbres binaires est défini par : - toute feuille est un arbre : $F \subset A(N,F)$; - si α et β sont deux arbres, et si $n \in N$, alors (n, α, β) est un arbre.

Le typageJava correspondant est le suivant :

```
1  abstract class Arbres<F,N>
2  {
3      boolean interne;
4
5      boolean estNoeud(){ return interne; }
6
7      boolean estFeuille(){  return !interne;  }
8  }
9
10 class Feuille<F,N> extends Arbres<F,N>
11 {
12      F contenu;
13 }
14
15 class Noeud<F,N> extends Arbre<F,N>
16 {
17     N contenu;
18    Arbre<F,N> gauche;
19    Arbre<F,N> droite;
20 }
```

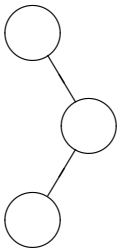
Programme 4.12 –

Un exemple d'arbre binaire du type `arbre_binaire<String, int>` est (on a choisi de représenter les noeuds par des ronds et les feuilles par des carrés)



4.2.2 Squelette d'un arbre

Le squelette d'un arbre définit sa géométrie : on l'obtient en supprimant toute information aux noeuds et feuilles, et en supprimant les feuilles. Voici le squelette de l'arbre précédent :



On type aisément les squelettes d'arbres binaires ainsi :

```
class SqueletteBinaire
{
        SqueletteBinaire gauche, droite;

        SqueletteBinaire(SqueletteBinaire gauche, SqueletteBinaire droite){
                this.gauche = gauche; this.droite = droite;
        }
}
```

Programme 4.13 –

en autorisant un squelette à être vide.

Quelques fonctions sur les squelettes d'arbres binaires : une méthode qui construit le squelette d'un arbre binaire

```
// pour un arbre
abstract Squelette decharne();
// pour une feuille
Squelette decharne(){ return null;}
// pour un noeud
Squelette decharne(){
      return new Squelette(decharne(gauche),decharne(droite));
}
```

Programme 4.14 –

et deux fonctions, l'une qui symétrise un arbre, l'autre qui teste l'égalité de deux squelettes

```
static SqueletteBinaire symetrie(SqueletteBinaire s){
        if(s==null) return null;
        return new SqueletteBinaire(symétrie(s.droite), symetrie(s.gauche));
}

static egalite_squelettes( SqueletteBinaire a, SqueletteBinaire b){
        if(a==null) return b==null;
        if(b==null) return false;
        return egalite_squelettes(a.gauche,a.droite) && egalite_squelettes(b.gauche,b.droite);
}
```

Programme 4.15 –

4.2.3 Propriétés combinatoires

Vocabulaire Nous appellerons taille d'un arbre binaire le nombre de ses noeuds internes. C'est aussi la taille de son squelette. On peut donc écrire :

```
1 // pour un arbre
2 abstract int taille();
3 // pour une feuille
4 int taille(){ return 0;}
5 // pour un noeud
6 int taille(){ return 1+gauche.taille()+droite.taille(); }
```

Programme 4.16 –

Sa hauteur (on parle aussi de profondeur) est définie inductivement par : toute feuille est de hauteur nulle, la hauteur d'un arbre (n, g, d) est égale au maximum des hauteurs de ses fils g et d augmenté d'une unité : la hauteur mesure donc l'imbrication de la structure.

```
1 // pour un arbre
2 abstract int hauteur();
3 // pour une feuille
4 int hauteur(){ return 0;}
5 // pour un noeud
6 int hauteur(){ return 1+max(gauche.hauteur(), droite.hauteur()); }
```

Programme 4.17 –

Le nombre de feuilles se déduit facilement du nombre de noeuds, c'est-à-dire de la taille :

Théorème 4.2.1 (Feuilles et noeuds d'un arbre binaire) *Le nombre de feuilles d'un arbre binaire de taille n est égal à n + 1.*

> **Démonstration**: D'aucuns peuvent utiliser une preuve inductive. On peut aussi dire que si f est le nombre de feuilles, $n + f$ compte le nombre de noeuds et feuilles qui sont, sauf la racine, fils d'un noeud interne : $n + f - 1 = 2n$. D'où le résultat : $f = n + 1$.

Taille et profondeur d'un arbre binaire sont étroitement liés.

Théorème 4.2.2 (Hauteur d'un arbre binaire) *Soit h la hauteur d'un arbre binaire de taille n. On dispose de :* $1 + \lceil \lg n \rceil \leq h \leq n$.

> **Démonstration**: La hauteur est la longueur du plus long chemin de la racine à une feuille. Au lieu de compter les arêtes, on peut compter les noeuds (internes)de départ, et on a bien $h \leq n$. Un arbre binaire de hauteur h est dit complet si toutes ses feuilles sont à la profondeur h ou $h - 1$. On vérifie facilement pour un tel arbre la relation $h = 1 + \lceil \lg n \rceil$.
>
> A un arbre binaire non complet on peut ajouter suffisament de noeuds (et de feuilles) pour qu'il soit complet : la taille passe alors de n à $n' = 2^h - 1 \geq n$, donc $\lceil \lg n \rceil < h$ et ainsi $\lceil \lg n \rceil + 1 \leq h$.

On en déduit aussitôt le

Corollaire 4.2.1 *Soit h la hauteur d'un squelette binaire de taille n. On dispose de :* $\lceil \lg n \rceil \leq h \leq n - 1$.

Ici encore les bornes sont optimales (c'est-à-dire qu'il existe des squelettes pour lesquels elles sont atteintes).

Une fonction Java qui renvoie un squelette binaire complet de taille donnée.

```
1 SqueletteBinaire squeletteDeTaille(int n){
2         if(n==0) return null;
3         return new SqueletteBinaire(squeletteDeTaille((n-1)/2),squeletteDeTaille(n - 1 - (n - 1)/2));
4 }
```

Programme 4.18 –

Une fonction Java qui décide si un squelette binaire est ou non complet.

```
1   static boolean est_complet(SqueletteBinaire s){
2             if(s==null) return true;
3             int hg= s.gauche.hauteur(), hd=s.droite.hauteur();
4             if(hg==hd) return est_complet(s.gauche) && est_complet(s.droite);
5             else if(hg==1+hd) return est_complet(s.gauche) && est_vraiment_complet(s.droite);
6             else if(hg+1==hd) return est_vraiment_complet(s.gauche) && est_complet(s.droite);
7             else return false;
8   }
9    static boolean est_vraiment_complet(SqueletteBinaire s){
10                if(s==null) return true;
11                return s.gauche.hauteur() == s.droite.hauteur()
12                   && est_vraiment_complet(s.gauche)
13                   && est_vraiment_complet(s.droite);
14   }
```

Programme 4.19 –

En fait il est beaucoup plus facile (et efficace) d'écrire d'abord une fonction qui renvoie une liste des profondeurs des feuilles.

```
1  static Liste<int> aux1(int n, Liste<int> l, SqueletteBinaire s){
2       if(s==null) return l.cons(n);
3       return aux1(n+1,aux1(n+1,l,s.droite),s.gauche)
4  }
5  static boolean estComplet(SqueletteBinaire s){
6           Liste<int> l = aux1(0,null,s);
7           int m=0, M=0,t;
8           while(!s.isEmpty()){
9                 t=l.tete();
10                m= t<m ? t : m;
11                M=t>M ? t : M;
12                l = l.reste();
13          }
14          return (M-m)<=1;
15 }
```

Programme 4.20 –

4.2.4 Dénombrement

Soit C_n le nombre de squelettes binaires de taille n : $C_0 = 1$, $C_1 = 1$,$C_2 = 2/$

On dispose de la récurrence : $\forall n \geq 1,\ C_n = \sum_{k=0}^{n-1} = C_k C n - 1 - k$, d'où pour la série génératrice $C(z) = \sum_{n=0}^{+\infty} C_n z^n$, la relation :

$$C(z) = C_0 + z \sum_{n=1}^{+\infty} \sum_{k=1}^{n-1} C_k C_{n-1-k} z^{n-1} = 1 + zC(z)^2$$

On résout cette équation (en tenant compte que $C(0) = C_0 = 1$) et on trouve

$$C(z) = \frac{1 - \sqrt{1-4z}}{2z}$$

et on en déduit le

Théorème 4.2.3 (Nombres de Catalan) *Le nombre de squelettes binaires de taille n est égal à $\frac{1}{n+1} C_{2n}^n$. Il est équivalent à $\frac{4^n}{n\sqrt{\pi n}}$.*

4.2.5 Complexité moyenne d'accès à un noeud dans un arbre binaire

Soit A un arbre binaire pris au hasard possédant n noeuds. Nous allons estimer la longueur moyenne $L(n)$ d'un chemin allant de la racine à un noeud x pris au hasard. Soit a_0 la racine de l'arbre, A_1 et A_2 les deux branches issues de a_0. Supposons que A_1 possède i noeuds ; en conséquence A_2 possède $n-i-1$ noeuds. Il y a i chances que x soit un noeud de A_1 avec une longueur moyenne de chemin égale à $L(i)+1$, $n-i-1$ chances que x soit un noeud de A_2 avec une longueur moyenne de chemin égale à $L(n-i-1)+1$ et enfin 1 chance que x soit égale à a_0 avec une longueur de

chemin égale à 0. On en déduit que la longueur moyenne du chemin dans le cas où A_1 possède i noeuds est égale à $\frac{1}{n}\Big(i\big(L(i)+1\big)+(n-i-1)\big(L(n-i-1)+1\big)\Big)$.

Maintenant, pour un arbre pris au hasard, toutes les valeurs de i de 0 à $n-1$ sont équiprobables et donc la longueur moyenne du chemin d'accès est égale à

$$\begin{aligned} L(n) &= \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{n}\Big(i\big(L(i)+1\big)+(n-i-1)\big(L(n-i-1)+1\big)\Big) \\ &= \frac{1}{n^2}\left(\sum_{i=0}^{n-1} iL(i)+\sum_{i=0}^{n-1}(n-i-1)L(n-i-1)+n(n-1)\right) \\ &= \frac{2}{n^2}\sum_{i=0}^{n-1} iL(i)+\frac{n-1}{n} \leq \frac{2}{n^2}\sum_{i=1}^{n-1} iL(i)+1 \end{aligned}$$

puisque $\sum_{i=1}^{n-1} i = \sum_{i=1}^{n-1}(n-i-1) = \frac{n(n-1)}{2}$ et

$$\sum_{i=0}^{n-1} iL(i) = \sum_{i=0}^{n-1}(n-i-1)L(n-i-1)$$

(changer i en $n-1-i$).

Nous allons montrer par récurrence sur n que $L(n) \leq 4\log n$. C'est vrai pour $n=1$ et pour $n=2$. Supposons donc l'inégalité vérifiée pour $i = 1, \ldots, n-1$. On a alors

$$L(n) \leq \frac{8}{n^2}\sum_{i=1}^{n-1} i\log(i)+1$$

Comme la fonction $x \mapsto x\log x$ est croissante sur $[1,+\infty[$, on a

$$\forall i \in \mathbb{N},\ i\log i \leq \int_i^{i+1} t\log t\ dt$$

d'où

$$L(n) \leq \frac{8}{n^2}\int_1^n t\log t\ dt+1 = \frac{8}{n^2}\left(\frac{n^2}{2}\log n-\frac{n^2}{4}+\frac{1}{4}\right)+1 \leq 4\log n$$

dès que $n \geq 3$, ce qui achève la récurrence. On obtient donc

Théorème 4.2.4 *La longueur moyenne du chemin allant de la racine à un noeud dans un arbre binaire pris au hasard est un $O(\log n)$.*

4.2.6 Arbres binaires homogènes

Nous désignerons encore par N l'ensemble des étiquettes des noeuds et F l'ensemble des étiquettes des feuilles, mais nous supposerons ici que $N = F$ (ce que nous appelerons un arbre homogène). Nous pouvons alors définir de manière récursive l'ensemble des arbres dont les noeuds et les feuilles sont étiquetés par F de la manière suivante :

- si x est un élément de F, alors x est un arbre (réduit à une feuille)
- tout famille finie d'arbres est une forêt
- tout couple formé d'un élément y de F et d'une forêt est un arbre

Dans les arbres homogènes, les feuilles ne se distinguent des noeuds que par le fait qu'elles n'ont pas de fils. Autrement dit, les feuilles sont simplement des noeuds de degré 0.

Il y a deux manières de traduire cela. La première consiste à autoriser une forêt à être vide : dans ce cas les feuilles sont les noeuds dont la forêt des branches est vide. La deuxième est d'autoriser un arbre lui même à être vide : dans ce cas, les feuilles sont les noeuds dont toutes les branches sont vides.

L'implémentation des branches partant d'un noeud sous forme de liste est tout particulièrement adaptée à la première traduction puisqu'une liste peut être vide, mais il est plus simple d'autoriser un arbre à être vide (`null` en Java), ce qui conduit à l'implémentation suivante pour les arbres binaires :

```java
public class ArbreBin<T>
{
   T contenu;
   ArbreBin<T> gauche, droite;

   ArbreBin(T contenu, ArbreBin<T> gauche,ArbreBin<T> droite){
       this.contenu=contenu;
       this.gauche=gauche;
       this.droite=droite;
    }
}
```

Programme 4.21 –

Autrement dit, nous identifions un arbre binaire homogène à un arbre binaire hétérogène dont les feuilles sont `null`. Par la suite, c'est à cette implémentation des arbres binaires homogènes que nous intéresserons plus particulièrement. Remarquons que dans ce cas, de tout noeud de l'arbre partent deux branches (une branche gauche et une branche droite), mais que, dans la mesure où certaines de ces branches peuvent être vides, un noeud peut avoir 2, 1 ou même 0 fils, ce dernier cas caractérisant les noeuds dits *terminaux* (pour éviter de parler de feuilles, ce qui pourrait prêter à confusion entre l'arbre homogène et l'arbre hétérogène associé). Les noeuds non terminaux seront dits *internes*.

Remarque Une présentation plus élégante des arbres binaires homogènes aurait utilisé l'implémentation suivante

```java
public class ArbreBin<T>
{   Node<T> racine;

        class Node<T>{
                T contenu;
                ArbreBin<T> gauche, droite;
                Node(T r){ this.contenu=r; gauche=null; droite = null;}
                Node(T r, ArbreBin<T> gauche,ArbreBin<T> droite){
                    this.contenu=r; this.gauche=gauche; this.droite=droite;
                }
        }
    ArbreBin(T,r){
          racine = new Node(r);
    }

     ArbreBin(T r, ArbreBin<T> gauche,ArbreBin<T> droite){
         racine = new Node(r,gauche,droite);
      }
}
```

Programme 4.22 –

Par souci de simplicité, nous avons privilégié la première approche.

4.3 Parcours d'arbres

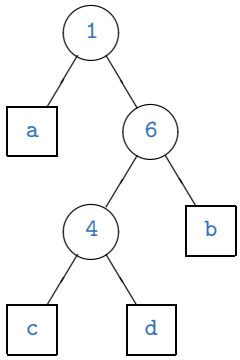
4.3.1 Principes

Prenons un arbre de type (F, N). Nous allons étudier les méthodes de parcours de l'arbre, le but étant de passer par chaque noeud et chaque feuille, tout en effectuant au passage certaines actions.

La programmation d'un tel parcours, en profondeur d'abord, de manière récursive est évidente : pour parcourir un arbre il suffit d'explorer la racine et (si l'arbre n'est pas réduit à une feuille) de parcourir chaque branche. Par contre, l'ordre de ces deux types d'opérations (exploration de la racine et parcours de chaque branche) n'est pas fixé. On dira que l'on parcourt l'arbre de manière préfixée si l'on explore la racine avant d'explorer les branches qui en sont issues, de manière postfixée si l'on explore la racine après avoir exploré les branches qui en sont issues, de manière infixée si l'on explore la racine entre le parcours de deux branches qui en sont issues.

Pour être complet, nous envisagerons également un parcours en largeur d'abord, encore appelé le parcours militaire : il consiste à lire profondeur par profondeur, de gauche à droite

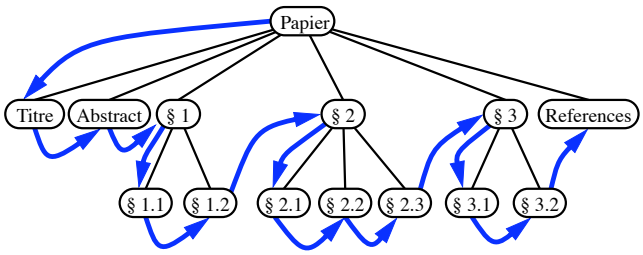
4.3.2 Exemples



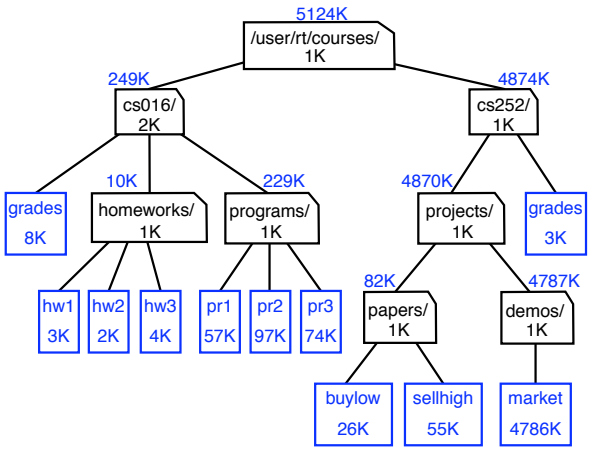
Voici ses parcours :

militaire : 1a64bcd ; préfixe : 1a64cdb ; infixe : a1c4d6b ; suffixe : acd4b61.

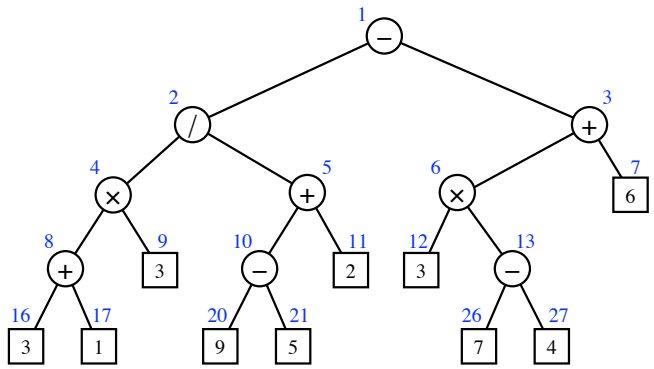
Parcours infixe : lecture d'un livre à partir de sa table des matières.



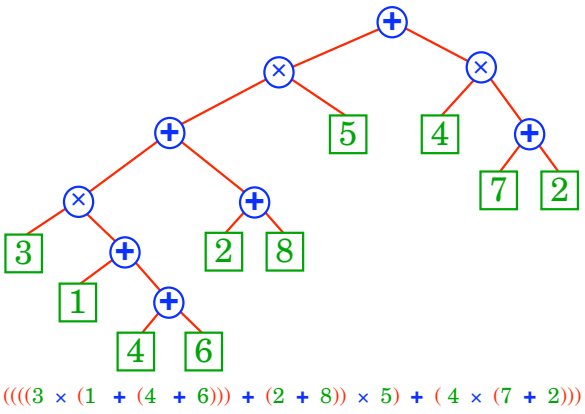
Parcours postfixe : calcul de la taille occupée par un répertoire sur un disque dur (commande `du` d'unix)



Parcours postfixe : évaluation d'une expression arithmétique



Parcours (presque) infixe : impression d'une expression arithmétique



La programmation de ces parcours est évidente. Nous utiliserons pour le parcours deux procédures qui agissent sur les objets de type F et de type N, l'une appelée `explore_feuille` et l'autre `explore_noeud` de types respectifs `F-> void` et `N -> void`. On a alors une procédure de parcours préfixé dans le cas où les forêts sont implémentées comme des listes

```
abstract class Arbre<F,N>
{       ...
         abstract void parcours_prefixe();
}
class Feuille<F,N> extends Arbre<F,N>
{       ...
        void parcours_prefixe(){ explore_feuille(contenu); }
}
class Noeud<F,N> extends Arbre<F,N>
{      ...
        private void prefixe_foret(Liste<Arbre<F,N>> foret){
                 if(foret.isEmpty()) return ;
                 foret.tete().parcours_prefixe();
                 prefixe_foret(foret.reste()));
        }
        void parcours_prefixe(){
                explore_noeud(contenu);
                prefixe_foret(branches);
        }
}
```

Programme 4.23 –

Pour le parcours postfixé, il suffit d'échanger l'ordre de l'exploration de la racine et de l'exploration des branches

```
abstract class Arbre<F,N>
{       ...
          abstract void parcours_postfixe();
}
class Feuille<F,N> extends Arbre<F,N>
{       ...
```

```
7         void parcours_postfixe(){ explore_feuille(contenu); }
8   }
9   class Noeud<F,N> extends Arbre<F,N>
10  {     ...
11        private void postfixe_foret(Liste<Arbre<F,N>> foret){
12                  if(foret.isEmpty()) return ;
13                  foret.tete().parcours_postfixe();
14                  postfixe_foret(foret.reste()));
15        }
16        void parcours_postfixe(){
17              postfixe_foret(branches);
18               explore_noeud(contenu);
19           }
20  }
```

Programme 4.24 –

Pour le parcours infixé, il faut se prémunir contre l'exploration de la forêt vide et passer la valeur de l'étiquette de la racine à la fonction de parcours de la forêt

```
1   abstract class Arbre<F,N>
2   {     ...
3           abstract void parcours_infixe();
4   }
5   class Feuille<F,N> extends Arbre<F,N>
6   {     ...
7         void parcours_infixe(){ explore_feuille(contenu); }
8   }
9   class Noeud<F,N> extends Arbre<F,N>
10  {     ...
11        private void infixe_foret(Liste<Arbre<F,N>> foret, N n){
12                  foret.tete().parcours_infixe();
13                  if(foret.reste().isEmpty()) return;
14                  explore_noeud(n);
15                  infixe_foret(foret.reste(),n);
16        }
17
18      void parcours_infixe(){
19                  infixe_foret(branches, contenu);
20      }
21  }
```

Programme 4.25 –

4.3.3 Exemples d'impression

Voici un exemple d'impression d'arbre sous forme préfixée, postfixée et infixée dans le cas de l'arbre déjà rencontré

```
1   #abd2415c3
2   24d1b53ca
3   2d4b1a5a3- : unit = ()
```

Programme 4.26 –

4.3.4 Parcours d'Euler

On peut aussi écrire une procédure générale de parcours d'arbre où le noeud est exploré une première fois avant les branches, puis entre chaque branche, puis une dernière fois après la dernière branche, avec trois procédures différentes d'exploration des noeuds :

```
1   abstract class Arbre<F,N>
2   {     ...
3           abstract void parcours();
4   }
5   class Feuille<F,N> extends Arbre<F,N>
6   {     ...
7         void parcours(){ explore_feuille(contenu); }
8   }
9   class Noeud<F,N> extends Arbre<F,N>
10  {     ...
11        private void parcours_foret(Liste<Arbre<F,N>> foret, N n){
```

```
12                  foret.tete().parcours();
13                  if(foret.reste().isEmpty()) return;
14                  in_explore_noeud(n);
15                  parcours_foret(foret.reste(),n);
16          }
17
18      void parcours(){
19                  pre_explore(contenu);
20                  parcours_foret(branches, contenu);
21                  post_explore(contenu);
22      }
23  }
```

Programme 4.27 –

Avec un exemple intéressant qui imprime la syntaxe concrète d'un arbre

```
1  void explore_feuille(F f){ System.out.print(f); }
2  void pre_explore(N n){ System.out.print("(");
3  void post_explore(N n){ System.out.print(")");
4  void in_explore(N n){ System.out.print(n);
5
6  a(b(d(2,4),1),5,c(3))
```

Programme 4.28 –

4.3.5 Parcours d'un arbre binaire

Dans le cas d'un arbre binaire, on peut écrire de manière plus simple l'exploration des noeuds :

```
1  class Noeud<F,N> extends Arbre<F,N>
2  {     ...
3        void parcours_prefixe(){
4                   explore_noeud(contenu);
5                   gauche.parcours_prefixe();
6                   droite.parcours_prefixe();
7      }
8        void parcours_postfixe(){
9                   gauche.parcours_prefixe();
10                  droite.parcours_prefixe();
11                  explore_noeud(contenu);
12
13     }
14       void parcours_infixe(){
15                  gauche.parcours_prefixe();
16                  explore_noeud(contenu);
17                  droite.parcours_prefixe();
18     }
19 }
```

Programme 4.29 –

4.3.6 Reconstitution d'un arbre à l'aide de son parcours préfixe

On suppose qu'on se donne un parcours d'un arbre par une liste d'objets du type `Parcours` qui correspond soit à une feuille soit à un noeud.

```
1  abstract class Parcours<F,N>
2  {      private int type;
3          boolean estFeuille(){ return type==0; }
4          boolean estNoeud(){ return type==1;}
5  }
6  classe ParcoursF<F,N> extends Parcours<F,N>{
7         F contenu;
8         ParcoursF(F x){ type=0; contenu = x; }
9  }
10 classe ParcoursN<F,N> extends Parcours<F,N>{
11        N contenu;
12        ParcoursN(N x){ type=1; contenu = x; }
13 }
```

Programme 4.30 –

On remarque que le parcours préfixe de la branche gauche est l'unique préfixe du parcours total qui soit un parcours préfixe syntaxiquement correct d'un arbre. On en déduit comment récupérer l'arbre depuis son parcours préfixe :

```
1  private static Arbre<F,N> aux_prefixe(Liste<Parcours<F,N>> parcours){
2              if(parcours.isEmpty()) throw new Error("Description préfixe incorrecte");
3              Parcours<F,N> tete = parcours.pop();
4              if(tete.estFeuille()) return new Feuille(tete.contenu);
5              else {
6                    Arbre<F,N> gauche = aux_prefixe(parcours);
7                    Arbre<F,N> droite = aux_prefixe(parcours);
8                    return new Noeud(tete.contenu,gauche,droite);
9              }
10 }
11
12 static Arbre<F,N> recompose_prefixe(Liste<Parcours<F,N>> parcours){
13             Liste<Parcours<F,N>> reste = parcours.clone();
14             Arbre<F,N> resultat=aux_prefixe(reste);
15             if(reste.isEmpty()) return resultat;
16             else  throw new Error("Description préfixe incorrecte");
17 }
```

Programme 4.31 –

4.3.7 Reconstitution d'un arbre à l'aide de son parcours postfixe

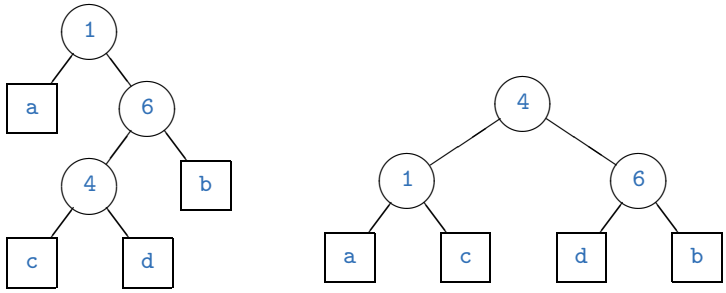
On utilise une pile d'arbres. Lorsqu'on rencontre une feuille, on l'empile. Lorsqu'on rencontre un noeud, on assemble à l'aide de ce noeud les deux arbres au sommet de la pile et on réempile le résultat. Voici ce que cela donne :

```
1  private void Arbre<F,N> aux_postfixe(Liste<Parcours<F,N>> parcours, Pile<Arbre<F,N>> pile){
2              if(parcours.isEmpty()) return;
3              Parcours<F,N> tete = parcours.pop();
4              if(tete.estFeuille()) pile.push(Feuille(tete.contenu));
5              else {
6                    Arbre<F,N> gauche = pile.pop();
7                    Arbre<F,N> droite = pile.pop();
8                    pile.push(Noeud(tete.contenu,gauche,droite));
9              }
10 }
11
12 static Arbre<F,N> recompose_postfixe(Liste<Parcours<F,N>> parcours){
13             Liste<Parcours<F,N>> reste = parcours.clone();
14             Pile<Arbre<F,N>> pile = new Pile();
15             aux_postfixe(parcours, pile);
16             if(pile.isEmpty()) throw new Error("Description postfixe incorrecte");
17             Arbre<F,N> resultat = pile.pop();
18             if(!pile.isEmpty()) throw new Error("Description postfixe incorrecte");
19             return resultat;
20 }
```

Programme 4.32 –

4.3.8 Reconstitution d'un arbre à l'aide de son parcours infixe

La reconstitution d'un arbre binaire à l'aide de son parcours infixe est malheureusement impossible, deux arbres distincts pouvant avoir le même parcours infixe :



4.4 Arbres binaires de recherche

4.4.1 Objet des arbres binaires de recherche

Le gestionnaire d'un hôtel dans une ville touristique est amené à gérer un fichier des clients. Bien entendu, chaque jour, certains clients quittent leur chambre, d'autres prolongent leur séjour, enfin de nouveaux clients arrivent. Nous recherchons une structure de données qui permette de gérer un tel fichier en évolution constante tout en autorisant une recherche aussi rapide que possible d'un client donné.

Une première solution est de gérer un tableau des clients. Ceci nécessite soit que nous choisissions dès le départ un tableau assez grand pour contenir le nombre maximum de clients que peut recevoir l'hôtel (attention au surbooking) en perdant une place considérable dans notre fichier en basse-saison, soit que nous procédions à des recopies incessantes de tableau dans des tableaux plus grands ou plus petits dès que l'évolution du nombre de clients le nécessite. Dans un tel tableau, si nous voulons avoir une procédure efficace de recherche d'un client donné, nous avons vu dans le cours de Math Sup qu'il fallait trier le tableau des clients, par exemple par ordre alphabétique, et procéder ensuite à une recherche dichotomique. La complexité de la recherche est alors un $O(\log_2 n)$ si n est le nombre de clients.

Pour supprimer la fiche d'un client, il faut d'abord la rechercher dans le tableau (complexité en $O(\log_2 n)$), puis décaler tous les clients suivants d'un cran vers la gauche (complexité moyenne en $O(n)$: si l'élément figure à la p-ième place, il faut décaler $n-p$ éléments, soit $n-p$ opérations ; comme toutes les places p de 1 à n sont équiprobables, la complexité moyenne est $\frac{1}{n}\sum_{p=1}^{n}(n-p) = \frac{n-1}{2} = O(n)$, ce qui maintient le tableau trié.

Pour insérer un client, il faut d'abord trouver par une recherche dichotomique à quel endroit il faut l'insérer (complexité en $O(\log_2 n)$) puis décaler tous les clients suivants d'un cran vers la droite (complexité moyenne en $O(n)$) avant d'insérer l'élément (complexité en $O(1)$), ce qui maintient le tableau trié. La structure de tableau trié est donc une structure de donnée statique (figée dans le temps hors recopie) pour laquelle la recherche se fait avec une complexité en $O(\log_2 n)$ alors que l'insertion et la suppression se font avec une complexité moyenne en $O(n)$.

Une deuxième solution est de gérer une liste des clients. La structure peut alors croître et décroître suivant les besoins. Le fait que de toute manière on ne puisse accéder directement qu'à la tête d'une liste rend sans intérêt tout tri de la liste : la recherche dans une liste ne peut se faire que de manière linéaire avec une complexité moyenne en $O(n)$. L'insertion d'un élément dans une liste se fait en temps constant $O(1)$ puisqu'il suffit de le mettre en tête de la liste. Par contre la suppression d'un élément nécessite sa recherche (complexité en $O(n)$), puis la concaténation des deux listes obtenues en supprimant cet élément (complexité moyenne en $O(n)$). La structure de liste est donc une structure de données dynamique (évolutive avec le temps) pour laquelle l'insertion se fait avec une complexité en $O(1)$ alors que la recherche ou la suppression se font avec une complexité moyenne en $O(n)$.

Nous allons voir que les arbres de recherche permettent de gérer un tel fichier de manière dynamique, la recherche, l'insertion et la suppression se faisant avec une complexité moyenne en $O(\log_2 n)$ et une complexité dans le pire des cas en $O(n)$.

4.4.2 Arbres binaires de recherche

Définition 4.4.1 *Soit $(X, \leq)$ un ensemble ordonné. On appelle arbre binaire de recherche dans X tout arbre binaire homogène A étiqueté par X tel que*

- *pour tout noeud r de A étiqueté par $\varepsilon(r) \in X$, pour tout noeud g de la branche gauche de r, étiqueté par $\varepsilon(g) \in X$, on a $\varepsilon(g) \leq \varepsilon(r)$*
- *pour tout noeud r de A étiqueté par $\varepsilon(r) \in X$, pour tout noeud d de la branche droite de r, étiqueté par $\varepsilon(d) \in X$, on a $\varepsilon(r) < \varepsilon(d)$*

Autrement dit dans un arbre binaire de recherche, les éléments à gauche d'un noeud sont inférieurs ou égaux à l'étiquette de ce noeud et les éléments à droite sont strictement supérieurs à cette étiquette.

Par la suite, et uniquement pour la facilité d'écriture, nous supposerons que l'ensemble ordonné X est celui des entiers naturels, implémenté par le type Java `int arbre_bin`.

Voici un exemple d'arbre binaire de recherche dans les entiers :

Par contre, si on remplace le 3 par un 5, on n'a plus un arbre binaire de recherche puisque le noeud étiqueté par 5 se trouve dans la branche gauche du noeud étiqueté par 4 alors que $5 > 4$.

Nous utiliserons le typage suivant des arbres binaires de recherche

```
1  public class ArbreBinInt
2  {
3      long contenu;
4      ArbreBinInt gauche, droite;
5      static long tresGrand=100000000;
6
7
8      ArbreBinInt(long contenu){
9          this.contenu=contenu;
10         gauche=null;
11         droite=null;
12     }
13
14     ArbreBinInt(long contenu, ArbreBinInt gauche, ArbreBinInt droite){
15         this.contenu=contenu;
16         this.gauche=gauche;
17         this.droite=droite;
18     }
19 }
```

Programme 4.33 –

4.4.3 Test d'un arbre binaire de recherche

Soit A un arbre binaire, par exemple étiqueté par $\mathbb{N}$; nous définirons $\min(A)$ comme la plus petite des étiquettes des noeuds de A et $\max(A)$ comme la plus grande de ces mêmes étiquettes. Pour un arbre vide, nous poserons $\min(A) = +\infty$ et $\max(A) = -\infty$.

Soit alors A un arbre binaire de racine r (étiquetée par $\varepsilon(r)$) ayant pour branches les deux arbres binaires A_1 et A_2. Alors A est un arbre binaire de recherche si et seulement si il vérifie

- A_1 et A_2 sont des arbres binaires de recherche
- $\max(A_1) \leq \varepsilon(r) < \min(A_2)$

Pour que cette définition récursive soit complète, il faut préciser qu'un arbre vide est un arbre binaire de recherche.

Nous pouvons alors définir une fonction qui à tout arbre A associe un triplet formé d'un booléen indiquant si A est un arbre binaire de recherche et de deux entiers $\min(A)$ et $\max(A)$ de la façon suivante :

```
1  private static void testeMinMax(ArbreBinInt a, long[] res){
2          if(a==null) { res[0]=1; res[1]=tresGrand; res[2]=-tresGrand;}
3          long[] resG=new long[3];
4          long[] resD=new long[3];
5          testeMinMax(a.gauche,resG);
6          testeMinMax(a.droite, resD);
7          res[1]= max(res[1],max(resG[1],resD[1]));
8          res[2]= min(res[2],min(resG[2],resD[2]));
9          if(resG[0]==1 && resG[1]==1 && resG[1]<=a.contenu && resD[2]>a.contenu) res[0]=1;
10         else res[0]=0;
11     }
12
13 static boolean testeArbreRecherche(ArbreBinInt a){
14         long[] res=new long[3];
15         testeMinMax(a,res);
16         return res[0]==1;
17     }
```

Programme 4.34 –

La complexité de ce calcul est en $O(n)$ si n est le nombre total de noeuds de l'arbre, puisque chaque noeud est examiné une et une seule fois. Il est alors facile de construire une fonction de test d'un arbre binaire en éliminant le minimum et le maximum qui sont sans intérêt pour l'arbre complet (alors qu'ils sont essentiels pour les sous-arbres) :

```
1  static boolean testeArbreRecherche(ArbreBinInt a){
2          long[] res=new long[3];
3          testeMinMax(a,res);
4          return res[0]==1;
5      }
```

Programme 4.35 –

4.4.4 Recherche dans un arbre binaire

Nous allons nous intéresser à quatre opérations sur les arbres binaires de recherche : le test pour savoir si un élément figure comme étiquette dans un arbre donné, l'insertion d'un nouvel élément, la suppression d'un élément, et enfin le parcours des éléments dans l'ordre.

Le test de présence d'un élément x comme étiquette de l'arbre binaire de recherche A se programme par induction structurelle :

- si l'arbre est vide, l'élément x ne figure pas dans l'arbre
- si l'arbre possède r comme racine étiquetée par $\varepsilon(r)$, de branche gauche A_1 et de branche droite A_2 alors
  - si $x = \varepsilon(r)$, alors x figure dans l'arbre
  - si $x < \varepsilon(r)$ alors x figure dans A si et seulement si il figure dans A_1 (puisque A_1 contient tous les éléments de l'arbre inférieurs ou égaux à $\varepsilon(r)$)
  - si $x > \varepsilon(r)$ alors x figure dans A si et seulement si il figure dans A_2 (puisque A_2 contient tous les éléments de l'arbre supérieurs à $\varepsilon(r)$)

Ceci conduit à la fonction Java suivante :

```
1 static boolean figureDansArbre(long x, ArbreBinInt a){
2         if(a==null) return false;
3         if(x==a.contenu) return true;
4         else if(x<=a.contenu) return figureDansArbre(x,gauche);
5         else return figureDansArbre(x,droite);
6     }
```

Programme 4.36 –

Remarque Nous n'avons pas mis de surveillant dans le dernier cas du filtrage pour éviter un avertissement de Java disant que notre filtrage n'est pas exhaustif ; le compilateur émet ce message de manière à peu près systématique dans un filtrage avec surveillants car il n'a aucun moyen de savoir que les différents surveillants `when ...` couvrent tous les cas possibles. Nous avons simplement rappelé en commentaire quelle est la situation lorsque ce dernier cas du filtrage est examiné.

Dans tous les cas, on constate que le nombre de comparaisons effectuées avant de trouver (ou de ne pas trouver) l'élément est égal au nombre de noeuds explorés, et comme on suit un chemin descendant de la racine au dernier noeud exploré, ce nombre est majoré par la hauteur de l'arbre augmentée de 1. Si l'arbre possède n noeuds, ce nombre est donc majoré par n. De plus comme la hauteur moyenne d'un arbre à n noeuds est un $O(\log_2 n)$, le nombre moyen de comparaisons effectuées pour rechercher un élément dans un arbre binaire pris au hasard est aussi un $O(\log_2 n)$.

Théorème 4.4.1 *La recherche d'un élément dans un arbre binaire de recherche à n noeuds se fait en un temps moyen $O(\log_2 n)$ et en un temps $O(n)$ dans le pire des cas.*

4.4.5 Insertion aux feuilles dans un arbre binaire de recherche

L'insertion d'un élément x dans un arbre binaire de recherche A (tout en maintenant évidemment la structure ordonnée d'arbre binaire de recherche) peut se faire par induction structurelle en plaçant le nouvel élément comme noeud terminal de l'arbre :

- si l'arbre est vide, le résultat est l'arbre dont l'unique noeud (terminal) est x
- si l'arbre possède r comme racine étiquetée par $\varepsilon(r)$, de branche gauche A_1 et de branche droite A_2 alors
  - si $x \leq \varepsilon(r)$ alors le résultat est l'arbre de racine r de branche droite A_2 et de branche gauche le résultat de l'insertion de x dans A_1
  - si $x > \varepsilon(r)$ alors le résultat est l'arbre de racine r de branche gauche A_1 et de branche droite le résultat de l'insertion de x dans A_2.

Il est clair par induction structurelle que l'on maintient ainsi la structure d'arbre de recherche : c'est évident pour un arbre vide et si c'est vrai pour les branches A_1 et A_2, cela reste vrai pour A puisque toutes les étiquettes des noeuds de la branche gauche restent inférieures à $\varepsilon(r)$ et toutes les étiquettes de la branche droite restent strictement supérieures à $\varepsilon(r)$.

On construit ainsi une fonction Java, en utilisant des surveillants :

```
1  static ArbreBinInt insere(long x,ArbreBinInt a){
2          if(a==null) return new ArbreBinInt(x);
3          if(x==a.contenu) return a;
4          else if(x<a.contenu) return new ArbreBin(a.contenu,insere(x,gauche),droite);
5          else return new ArbreBin(a.contenu,gauche,insere(x,droite));
6      }
```

Programme 4.37 –

On peut alors insérer tous les éléments d'une liste dans un arbre binaire de recherche :

```
1  static ArbreBinInt insereListe(Liste<Long> l,ArbreBinInt a){
2          if(l.isEmpty()) return a;
3          else return insereListe(l.reste(),insere(l.tete(),a));
4      }
```

Programme 4.38 –

Nous donnons ci dessous quelques exemples d'exécutions, tout d'abord avec une liste triée par ordre croissant :

```
1  #liste_a_arbre [1;2;3;4;5;6];;
2  - : int arbre_bin =
3   Noeud
4    (Noeud
5      (Noeud (Noeud (Noeud (Noeud (Vide, 1, Vide), 2, Vide), 3, Vide), 4, Vide),
6       5, Vide),
7     6, Vide)
```

Programme 4.39 –

puis avec une liste triée par ordre décroissant :

```
1  #liste_a_arbre [6;5;4;3;2;1];;
2  - : int arbre_bin =
3   Noeud
4    (Vide, 1,
5     Noeud
6      (Vide, 2,
7       Noeud (Vide, 3, Noeud (Vide, 4, Noeud (Vide, 5, Noeud (Vide, 6, Vide))))))
```

Programme 4.40 –

puis avec une liste non triée :

```
1  #liste_a_arbre [4;2;3;6;5;1];;
2  - : int arbre_bin =
3   Noeud
4    (Vide, 1,
5     Noeud
6      (Noeud (Noeud (Vide, 2, Vide), 3, Noeud (Vide, 4, Vide)), 5,
7       Noeud (Vide, 6, Vide)))
```

Programme 4.41 –

Testons les hauteurs des arbres de recherche obtenus :

```
1  #hauteur (liste_a_arbre [1;2;3;4;5;6]);;
2  - : int = 5
3  #hauteur (liste_a_arbre [6;5;4;3;2;1]);;
4  - : int = 5
5  #hauteur (liste_a_arbre [4;2;3;6;5;1]);;
6  - : int = 3
```

Programme 4.42 –

Nous constatons que dans les deux premiers cas, l'introduction de données préalablement ordonnées a donné lieu à des constructions d'arbres binaires complètement déséquilibrés (ils ont même une structure linéaire) et qui sont de hauteur maximum. Dans le troisième cas, l'introduction de données dans un ordre aléatoire a donné lieu à la construction d'un arbre beaucoup plus équilibré.

Les mêmes arguments que pour la recherche dans un arbre binaire conduisent à une conclusion similaire :

Théorème 4.4.2 *L'insertion d'un élément dans un arbre binaire de recherche à n noeuds se fait en un temps moyen $O(\log_2 n)$ et en un temps $O(n)$ dans le pire des cas.*

4.4.6 Insertion à la racine dans un arbre binaire de recherche

Nous allons maintenant insérer le nouvel élément comme racine de l'arbre binaire de recherche. Pour cela nous allons définir une fonction qui découpe un arbre binaire de recherche suivant une valeur pivot x. Cette fonction renverra deux arbres binaires de recherche, l'un contenant les éléments inférieurs ou égaux à x, l'autre contenant les éléments strictement supérieurs à x.

```
1  static ArbreBinInt[] decoupe(long x,ArbreBinInt a){
2          ArbreBinInt[] res = new ArbreBinInt[2];
3          if(x==a.contenu) { res[0]=a.gauche; res[1]=a.droite; return res;}
4          else if(x<a.contenu){
5              ArbreBinInt[] resG = decoupe(x,a.gauche);
6              res[0]=resG[0];
7              res[1]=new ArbreBinInt(a.contenu,resG[1], a.droite);
8              return res;
9          }
10         else {
11             ArbreBinInt[] resD = decoupe(x,a.droite);
12             res[0]=new ArbreBinInt(a.contenu, a.gauche, resD[0]);
13             res[1]= resD[1];
14             return res;
15     }
16 }
```

Programme 4.43 –

Pour insérer un élément x à la racine d'un arbre, il suffit maintenant de découper l'arbre suivant la valeur x, puis de reconstituer l'arbre en prenant x comme racine.

```
1  static ArbreBinInt insereRacine(long x,ArbreBinInt a){
2         ArbreBinInt[] res = decoupe(x,a);
3         return new ArbreBinInt(x,res[0],res[1]);
4      }
```

Programme 4.44 –

4.4.7 Suppression dans un arbre binaire de recherche

Soit A un arbre binaire de recherche sur X et $x \in X$. Nous cherchons à supprimer la première occurrence d'un noeud ayant l'étiquette x dans l'arbre A, l'arbre étant parcouru de haut en bas.

Pour savoir supprimer un élément d'un arbre binaire de recherche, il suffit, par induction structurelle, de savoir supprimer la racine d'un tel arbre avec le code évident :

```
1  static ArbreBinInt supprime(long x, ArbreBinInt a){
2          if(a==null) throw new Error("Non trouvé");
3          if(x==a.contenu) return supprimeRacine(a);
4          else if(x<a.contenu) return new ArbreBinInt(a.contenu,supprime(x,a.gauche),a.droite);
5          else return new ArbreBinInt(a.contenu,a.gauche,supprime(x,a.droite));
6      }
```

Programme 4.45 –

Il nous reste donc à savoir supprimer la racine d'un tel arbre. Soit donc A un arbre de racine r ayant une branche gauche A_1 et une branche droite A_2. Si A_1 est vide, le résultat de la suppression de la racine est bien évidemment A_2 ; de même si A_2 est vide, le résultat de la suppression de la racine est bien évidemment A_1. Nous supposerons donc que A_1 et A_2 sont non vides. Soit alors n le noeud de A_1 qui a l'étiquette la plus grande. On a donc $b = \varepsilon(n) \leq \varepsilon(r) = a$ et pour tout noeud n' de A_1 on a $\varepsilon(n') \leq \varepsilon(n)$. Le noeud n ne peut pas avoir de fils droit d, sinon on devrait avoir à la fois $\varepsilon(d) > \varepsilon(n)$ (car d est un fils droit de n) et $\varepsilon(d) \leq \varepsilon(r)$ (car d est un noeud de A_1), ce qui contredirait la définition de n. Autrement dit le noeud n est facile à supprimer de l'arbre. Il suffit alors de reporter son étiquette sur la racine r : on obtient un nouvel arbre binaire A', dont nous allons montrer qu'il s'agit bien d'un arbre binaire de recherche.

La branche droite de la racine r' n'a pas changé, donc c'est un arbre binaire de recherche. Dans la branche gauche de la racine, nous avons supprimé un noeud n'ayant pas de fils droit, et par conséquent nous avons encore un arbre binaire de recherche A'_1. Il nous suffit de vérifier que pour tout noeud p de A'_1 et pour tout noeud q et A_2, on a $\varepsilon(p) \leq b = \varepsilon(r') < \varepsilon(q)$. La première inégalité est claire par la définition de b comme l'étiquette la plus grande de A_1 ; la deuxième inégalité provient de ce que

$$\varepsilon(r') = b = \varepsilon(n) \leq \varepsilon(r) < \varepsilon(q)$$

puisque A était un arbre binaire de recherche.

Il nous reste donc à trouver le noeud dont l'étiquette est la plus grande dans l'arbre gauche A_1. Il nous suffit pour cela de rechercher le noeud le plus à droite de cet arbre c'est-à-dire de parcourir l'arbre en restant le plus à droite possible. Nous commencerons donc par définir une fonction qui parcourt un arbre vers la droite tout en éliminant le noeud le plus à droite ; une autre fonction retournera l'étiquette de ce noeud :

```
1  private static ArbreBinInt supprimeDroite(ArbreBinInt a){
2          if(a==null) throw new Error("Non trouvé");
3          if(a.droite==null) return a.gauche;
4          ArbreBinInt nouveauD = supprimeDroite(a.droite);
5          return new ArbreBinInt(a.contenu,a.gauche,nouveauD);
6      }
7
8      private static long plusADroite(ArbreBinInt a){
9          if(a==null) throw new Error("Non trouvé");
10         if(a.droite==null) return a.contenu;
11         return plusADroite(a.droite);
12     }
```

Programme 4.46 –

On peut alors écrire notre procédure de suppression de la racine d'un arbre binaire de recherche :

```
1  private static ArbreBinInt supprimeRacine(ArbreBinInt a){
2          if(a==null) throw new Error("Non trouvé");
3         if(a.droite==null) return a.gauche;
4         if(a.droite==null) return a.droite;
5         long x = plusADroite(a.gauche);
6         ArbreBinInt nouveauG=supprimeDroite(a.gauche);
7         return new ArbreBinInt(x,nouveauG,a.droite);
8  }
```

Programme 4.47 –

4.4.8 Fusion de deux arbres binaires de recherche

La fusion de deux arbres binaires de recherche qui ont la même racine est évidente : on fusionne les deux branches gauche et on fusionne les deux branches droites. On reconstitue ensuite un arbre dont la racine a pour étiquette la valeur commune des étiquettes des racines des deux arbres. Or cette situation est facile à obtenir : il suffit d'insérer à la racine de l'un des arbres l'étiquette de la racine de l'autre. On obtient ainsi le code

```
1  static ArbreBinInt fusionne(ArbreBinInt a, ArbreBinInt b){
2          if(a==null) return b;
3          if(b==null) return a;
4          ArbreBinInt b1 = insere(a.contenu,b);
5          return new ArbreBinInt(a.contenu, fusionne(a.gauche,b1.gauche),fusionne(a.droite,b1.droite));
6  }
```

Programme 4.48 –

4.5 Files de priorité

4.5.1 files d'attente

Types courants de files d'attente :

1. file LIFO ou pile : dernier entré, premier sorti
2. file FIFO ou queue : premier entré, premier sorti
3. file de priorité : celui qui a la plus grande priorité est le premier sorti

Analogue à la file d'attente aux urgence d'un hopital : on commence par s'occuper du plus gravement atteint.
Le type de donnée *File de priorité* doit disposer des méthodes :
– test pour savoir si la liste est vide
– insérer un élément dans la file

– trouver l'élément de plus grande priorité
– retirer l'élément de plus grande priorité

Pour simplifier, nous supposerons par la suite que les objets stockés sont des entiers et que le plus grand a la plus grande priorité.

4.5.2 Files de priorité et tris

Toute file de priorité permet de trier des éléments : on insère au fur et à mesure les éléments à trier dans la file de priorité, puis on retire les éléments un par un du plus grand au plus petit.

4.5.3 Implémentations élémentaires

Dans une liste non triée

Dans une liste non triée
– test pour savoir si la liste est vide : évident, $O(1)$
– insérer un élément dans la file : en tête de la liste, $O(1)$
– trouver l'élément de plus grande priorité : on parcourt la liste, $O(n)$
– retirer l'élément de plus grande priorité : on parcourt la liste, $O(n)$

Tri associé : tri par sélection.

```
1  public class FilePriorite
2  {
3      private Liste<Long> file;
4
5      void insere(long x){
6          file=file.cons(x);
7      }
8
9      private long plusGrand(Liste<Long> liste){
10         if(liste.isEmpty()) throw new Error("file vide");
11         return max(liste.tete(),plusGrand(liste.reste()));
12     }
13
14     private Liste<Long> supprimeListe(long x,Liste<Long> liste){
15         if(liste.isEmpty()) throw new Error("file vide");
16         long y=liste.tete();
17         if(x==y) return liste.reste();
18         return supprimeListe(x,liste.reste()).cons(x);
19     }
20
21     long depile(){
22         long x = plusGrand(file);
23         file=supprimeListe(x,file);
24         return x;
25     }
26 }
```

Programme 4.49 –

Dans une liste triée

Dans une liste triée
– test pour savoir si la liste est vide : évident, $O(1)$
– insérer un élément dans la file : parcourir la liste, $O(n)$
– trouver l'élément de plus grande priorité : en tête de la liste, $O(1)$
– retirer l'élément de plus grande priorité : on supprime la tête de la liste, $O(1)$

Tri associé : tri par insertion.

```
1  public class FilePriorite
2  {
3      private Liste<Long> file;
4
5      static Liste<Long> insereaux(long x, Liste<Long> l){
6          long y = l.tete();
7          if(x>y) return l.cons(x);
```

```
8           else return insereaux(x, l.reste()).cons(x);
9       }
10
11      void insere(long x){
12          file = insereaux(x,file);
13      }
14
15      long depile(){
16          long x = file.tete();
17          file=file.reste();
18          return x;
19      }
20  }
```

Programme 4.50 –

4.5.4 Arbre maximier

C'est un arbre binaire homogène dans lequel l'étiquette d'un noeud interne est toujours supérieure ou égale aux étiquettes de chacun de ses fils.

Test d'un arbre maximier :

```
1   public class ArbreBinInt
2   {
3       long contenu;
4       ArbreBinInt gauche, droite;
5
6           static boolean estMaximier(ArbreBinInt a){
7                   if(a==null) return true;
8                   if(a.gauche==null && a.droite==null) return true;
9                   if(a.droite==null) return estMaximier(a.gauche) && a.contenu>=a.gauche.contenu;
10                  if(a.gauche==null) return estMaximier(a.droite) && a.contenu>=a.droite.contenu;
11                  return estMaximier(a.gauche) && a.contenu>=a.gauche.contenu && estMaximier(a.droite) && a.
                        contenu>=a.droite.contenu;
12  }
13  }
```

Programme 4.51 –

Fonctions évidentes :

```
1           static boolean estVide(ArbreBinInt a){ return a==null;}
2           static boolean plusGrand(ArbreBinInt a){ return a.contenu;}
```

Programme 4.52 –

4.5.5 Insertion dans un arbre maximier

Par induction structurelle.

Insérer un élément x dans un arbre maximier A.

- Si l'arbre est vide, retourner l'arbre à un seul noeud étiqueté par x.
- Si l'arbre possède une racine r étiquetée par $\varepsilon(r)$
  - si $\varepsilon(r) \geq x$, insérer l'élément x dans l'une des branches g issue de r ; cette branche deviendra un arbre maximier g' dont la racine sera soit étiquetée par $x \leq \varepsilon(r)$, soit étiquetée par l'une des étiquettes d'un noeud de g, donc inférieure ou égale à $\varepsilon(r)$; comme l'autre branche n'aura pas été modifiée, l'arbre obtenu sera bien un arbre maximier
  - si $\varepsilon(r) < x$, remplacer l'étiquette de la racine par x puis insérer l'élément $\varepsilon(r)$ dans l'une des branches d issue de r ; cette branche deviendra un arbre maximier d' dont la racine sera soit étiquetée par $\varepsilon(r) < x$, soit par l'étiquette d'un noeud de d donc inférieure à $\varepsilon(r) < x$ (en fait, il est clair que ce dernier cas ne peut pas se produire) ; quant à l'autre branche g issue de r, elle n'a pas été modifiée, c'est donc encore une arbre maximier et l'étiquette de sa racine est inférieure ou égale à $\varepsilon(r)$ et a fortiori à x.

Dans tous les cas, l'arbre obtenu sera un arbre maximier.

Ceci peut se traduire en Java par la fonction :

```
static ArbreBinInt insere(long x, ArbreBinInt a){
        if(a==null) return new ArbreBinInt(x,null,null);
        if(x<a.contenu) return new ArbreBinInt(a.contenu,insere(x,a.gauche),a.droite);
        return new ArbreBinInt(x,a.gauche,insere(a.contenu,a.droite);
}
```

Programme 4.53 –

Pourquoi gauche-droite ? ?

4.5.6 Depilement récursif dans un arbre maximier

La suppression de la racine d'un arbre maximier peut se faire de manière récursive : on choisit la branche ayant la racine de plus grande priorité, on extrait cette racine et on en fait la nouvelle racine de l'arbre.

On aboutit à l'algorithme suivant :

```
static ArbreBinInt supprime_racine(ArbreBinInt a){
        if(a==null) throw new Error("Arbre_vide");
        if(a.gauche==null && a.droite==null) return null;
        if(a.gauche==null) return a.droite;
        if(a.droite==null) return a.gauche;
        if(a.gauche>=a.droite) return newArbreBinInt(a.gauche.contenu, supprime_racine(a.gauche),a.droite);
        else return newArbreBinInt(a.droite.contenu, a.gauche, supprime_racine(a.droite));
}
```

Programme 4.54 –

Inconvénient : on ne maîtrise absolument pas la géométrie de l'arbre.

4.5.7 Dépilement dans un arbre maximier par percolation

Il est facile de supprimer sont les noeuds terminaux d'un arbre maximier.

La suppression de la racine peut alors se faire en trois étapes.

- échanger la racine avec un noeud terminal de l'arbre ;
- supprimer ce noeud terminal ;
- rétablir ensuite la structure de file de priorité : la percolation.

La percolation va consister à faire redescendre l'étiquette de la racine tout au long de l'arbre en l'échangeant récursivement avec la plus grande des étiquettes de ses fils, jusqu'à ce que son étiquette soit plus grande que toutes les étiquettes de ses fils.

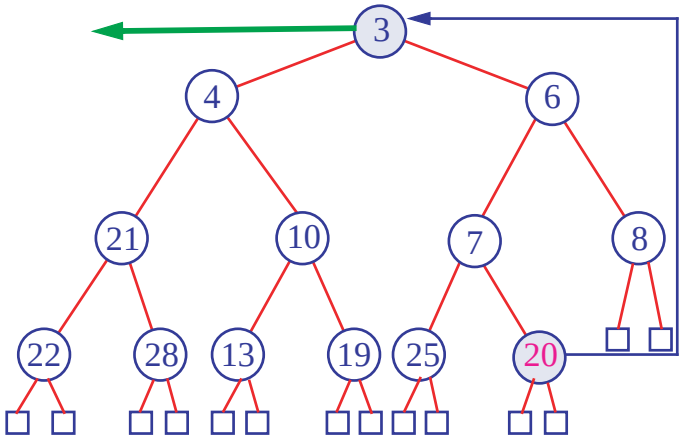


FIGURE 4.1 – Retrait de 3, à remplacer par 20

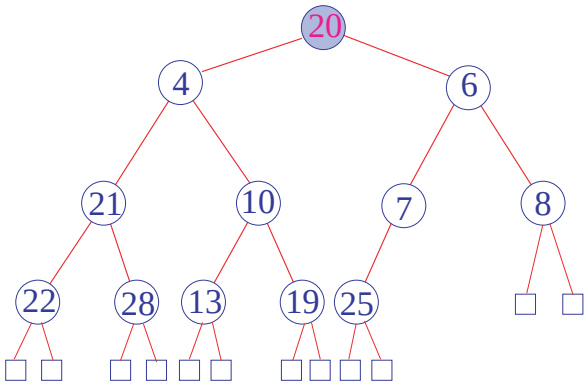


FIGURE 4.2 – Arbre à corriger

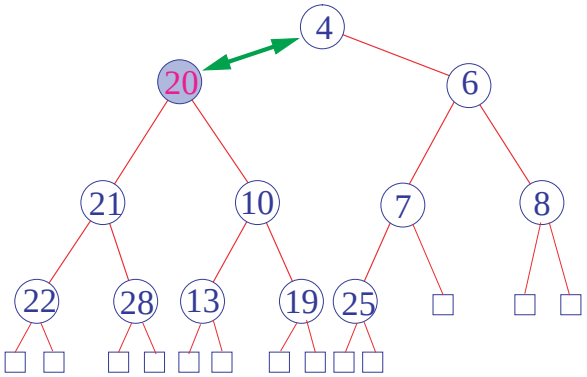


FIGURE 4.3 – Echanger 20 avec le plus petit de ses fils

4.5.8 Percolation de la racine à travers un arbre maximier

```
1  static ArbreBinInt percole(ArbreBinInt a){
2          if(a==null) return a;
3          if(a.gauche==null && a.droite==null) return a;
4          if(a.droite==null && a.gauche.contenu > a.contenu)
5                      return new ArbreBinInt(a.gauche.contenu, percole(new ArbreBinInt(a.contenu, a.gauche.
                           gauche,a.gauche.droite)),null);
6          if(a.gauche==null && a.droite.contenu > a.contenu)
7                      return new ArbreBinInt(a.droite.contenu, null, percole(new ArbreBinInt(a.contenu, a.
                           droite.gauche,a.droite.droite)));
8           if(a.gauche.contenu>a.contenu && a.gauche.contenu>a.droite.contenu)
9                     return new ArbreBinInt(a.gauche.contenu, percole(new ArbreBinInt(a.contenu, a.gauche.
                          gauche,a.gauche.droite)),a.droite);
10           if(a.droite.contenu>a.contenu && a.gauche.contenu<a.droite.contenu)
11                    return new ArbreBinInt(a.gauche.contenu, a.gauche,percole(new ArbreBinInt(a.contenu, a.
                          droite.gauche,a.droite.droite)));
12            return a;
13    }
```

Programme 4.55 –

4.5.9 Extraction de la racine d'un arbre maximier

```
1  let supprime_racine =
2      let rec supprime_terminal = function
3        | Vide -> invalid_arg "arbre_vide"
4        | Noeud(Vide,n,Vide) -> n,Vide
5        | Noeud(Vide,n,droite) -> let (x,d) = supprime_terminal droite
6                                  in (x,Noeud(Vide,n,d))
```

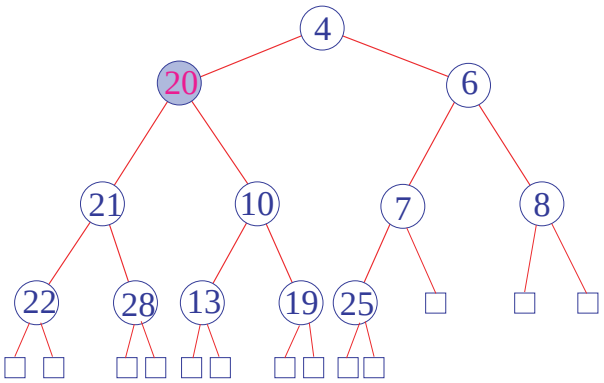


FIGURE 4.4 – Arbre à corriger

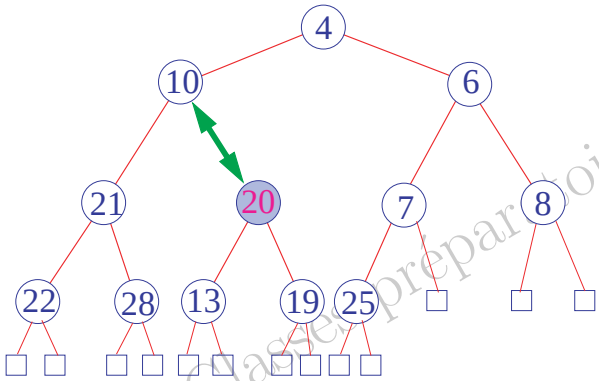


FIGURE 4.5 – Echanger 20 avec le plus petit de ses fils

```
7         | Noeud(gauche,n,droite) -> let (x,g) = supprime_terminal gauche
8                                     in (x,Noeud(g,n,droite))
9       in function
10        | Vide -> invalid_arg "arbre_vide"
11        | Noeud(Vide,n,Vide) -> Vide
12        | a -> begin
13             match supprime_terminal a with
14              | (term , Noeud (gauche,r,droite)) ->
15                    (r , percole (Noeud(gauche,term,droite)))
16              | _ -> invalid_arg "erreur_inconnue"
17           end;;
```

Programme 4.56 –

4.5.10 La structure de tas

Procédures de suppression de la racine ou d'insertion d'un élément dans un arbre maximier : complexité en $O(h)$ où h est la hauteur de l'arbre.

On peut espérer une complexité moyenne en $O(\log_2 n)$, si n est le nombre de noeuds.

Mais ces opérations répétées ont tendance à déséquilibrer les arbres du fait que l'on a à tout moment la liberté de choix entre la *droite* et la *gauche*, et que la méthode la plus naturelle consiste à faire ce choix de manière systématique.

Or pour des arbres déséquilibrés, la complexité passe en $O(n)$.

Solution : forcer les arbres à être équilibrés. Intervention sur la géométrie de l'arbre.

Exemple : dans la suppression par percolation, supprimer un des noeuds terminaux de profondeur maximale.

Considérons un arbre binaire homogène de hauteur h. Si $0 \leq d \leq h$, on appellera *niveau* d de l'arbre l'ensemble des noeuds situés à une distance d de la racine.

Le niveau d de l'arbre a au plus 2^d éléments.

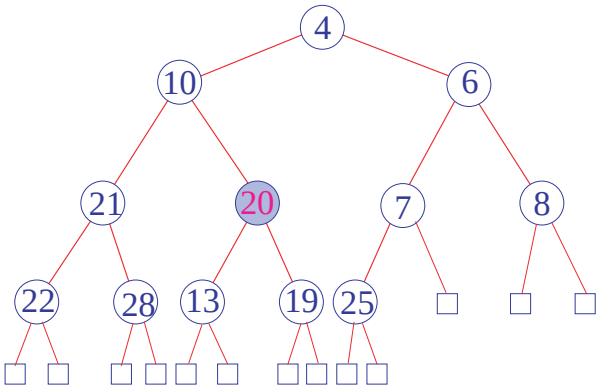


FIGURE 4.6 – Arbre à corriger

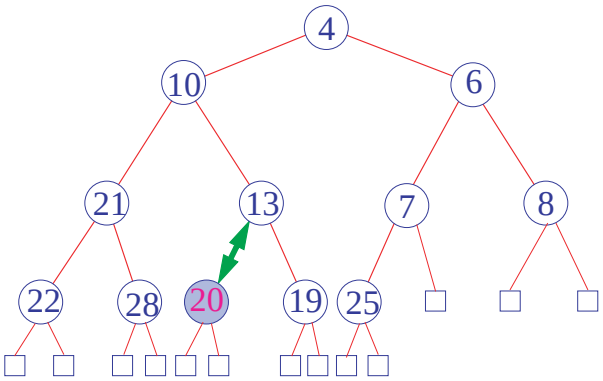


FIGURE 4.7 – Echanger 20 avec le plus petit de ses fils

Définition 4.5.1 *On dit qu'un arbre binaire homogène de hauteur h est complet si pour tout $d \in [0, h-1]$, le niveau d de l'arbre possède 2^d éléments et si les noeuds du niveau h sont les plus à gauche possible.*

Définition 4.5.2 *On appelle tas (en anglais heap) un arbre binaire homogène qui est un arbre maximier.*

On numérote alors les noeuds de haut en bas et de gauche à droite, depuis 0 jusqu'à $n-1$, suivant le schéma suivant :

Cette numérotation permet de stocker les éléments de l'arbre dans un tableau de longueur n (dont les indices varieront de 0 à $n-1$, attention au décalage).

Les fils de l'élément numéroté i sont l'élément numéroté $2i+1$ pour le fils gauche et $2i+2$ pour le fils droit.

On utilisera un tableau de grande taille dont seuls les n premiers éléments seront utilisés.

```
1  public class Tas
2  {
3      long[] contenu;
4      int nb;
5      static int taille=100;
6
7      Tas(){
8          contenu = new long[taille];
9          nb = 0;
10      }
11 }
```

Programme 4.57 –

Transformer un tas en un arbre :

```
1  private ArbreBinInt traite_noeud(int i, Tas t){
```

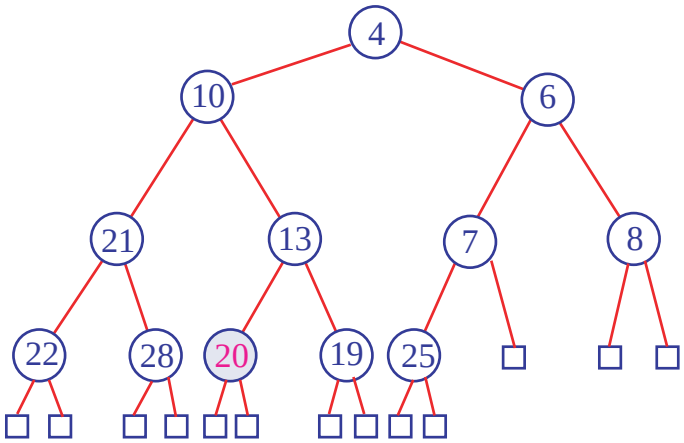


FIGURE 4.8 – Arbre corrigé

```
        if(i>=t.nb) return null;
        return new ArbreBinInt(t.contenu[i], traite_noeud(2*i+1,t), traite_noeud(2*i+2,t));
    }

    ArbreBinInt tas_en_arbre(Tas t){
        return traite_noeud(0,t);
    }
```

Programme 4.58 –

Transformer un arbre (supposé complet) en tas :

```
private static void traite_noeud(int i, ArbreBinInt a, Tas t){
        if(a==null) return;
        if(i>=t.nb) t.nb = i+1;
        t.contenu[i] = a.contenu;
        traite_noeud(2*i+1, a.gauche, t);
        traite_noeud(2*i+2, a.droite, t);
}

static Tas arbre_en_tas(ArbreBinInt a){
        Tas t = new Tas();
        traite_noeud(0, a, t);
        return t;
}
```

Programme 4.59 –

4.5.11 Suppression de la racine dans un tas

On cherche à adapter l'algorithme de suppression de la racine d'un arbre maximier : échanger l'étiquette de la racine avec l'étiquette d'un noeud terminal, puis supprimer ce noeud terminal, et enfin effectuer une percolation.

Supprimer le noeud le plus à droite du niveau maximal, c'est-à-dire dans la structure de tas, le dernier élément numéroté.

```
static long supprime_racine(Tas t){
      int n = t.nb;
      long racine = t.contenu[0];
      t.contenu[0] = t.contenu[n-1];
      t.nb = n-1;
      percole(t);
      return racine;
}
```

Programme 4.60 –

Percolation : on fait redescendre l'étiquette de la racine tout au long du tas en l'échangeant récursivement avec la plus grande des étiquettes de ses fils, jusqu'à ce que son étiquette soit plus grande que toutes les étiquettes de ses fils.

```
private static void echange(int i, int j, long[] v){
       long temp=v[i]; v[i]=v[j]; v[j]=temp;
}

private static void percole_aux(int i, Tas t){
      int n=t.nb; long[] v = t.contenu;
      if(2*i+1 >= n) return; /* le noeud n'a pas de fils */
       int j = 2*i+1; /* le fils gauche */
       int fils = (j+1 < n && v[j]<v[j+1]) ? j+1 : j ; /* le plus grand fils */
       if(v[fils] > v[i]){
              echange(i, fils, v);
              percole_aux(fils,t);
         }
}
static void percole(Tas t){
      percole_aux(0,t);
}
```

Programme 4.61 –

Amélioration : les échanges concernent toujours un même élément, la racine.

Pas nécessaire d'effectuer complètement chaque échange :

– affecter au père l'étiquette de son fils lorsqu'un échange est nécessaire
– lorsque le processus s'arrête, d'affecter au noeud l'étiquette de la racine.

```
private static void percole_aux(int i, Tas t, long racine){
      int n=t.nb;    long[] v = t.contenu;
      if(2*i+1 <  n){ /* le noeud a des fils */
       int j = 2*i+1; /* le fils gauche */
       int fils = (j+1 < n && v[j]<v[j+1]) ? j+1 : j ; /* le plus grand fils */
       if(v[fils] > racine){
              v[i] = v[fils]; /* on remonte le fils */
              percole_aux(fils, t, racine);
         } else {
             v[i] = racine; /* on stocke l'élément */
          }
        else v[i] = racine;  /* le noeud est terminal */
}

 static void percole(Tas t){
      percole_aux(0, t, t.contenu[0]);
}
```

Programme 4.62 –

Complexité de cet algorithme est dominée par la hauteur de l'arbre, c'est donc un $O(\log_2 n)$ si n est le nombre de noeuds.

4.5.12 Insertion dans un tas

La technique d'insertion dans un arbre maximier par induction structurelle est totalement inadaptée à la structure de tas.

Cette méthode consistait

– dans un cas à insérer l'élément dans la branche gauche ou la branche droite de l'arbre
– dans l'autre cas à remplacer l'étiquette de la racine par le nouvel élément puis à insérer l'ancienne racine dans la branche gauche ou la branche droite de l'arbre

Détruit la structure d'arbre complet : un seul chemin d'insertion dans l'arbre qui permette de maintenir cette structure, mais il faudrait pratiquement être devin pour le trouver.

Privilégier la structure d'arbre complet en insérant le nouvel élément à la première place libre. On détruit la structure d'arbre maximier.

Rétablir la structure de tas par une opération inverse de la percolation : remonter l'élément de proche en proche en l'échangeant avec l'étiquette de son père tant que cette étiquette est inférieure.

```
private static retablis(Tas h){
        int n = h.nb -1;
        int p = (n-1)/2;
        long[] v = h.contenu;
        while( n>=1 && v[p]<v[n]){
```

```
6                 echange(n,p,v);
7                 n=p; p=(n-1)/2;
8           }
9   }
10
11  static insere(long x, Tas h){
12                if(h.nb>=h.contenu.length) throw new Error("débordement du tas");
13                h.contenu[h.nb] = x;
14                h.nb += 1;
15                retablis(h);
16  }
```

Programme 4.63 –

Validité de cet algorithme : la procédure `retablis` appliquée à un tableau qui est un tas, sauf peut-être en ce qui concerne son dernier élément, transforme ce tableau en un tas.

Cette procédure ne modifiant pas la structure de l'arbre conserve la propriété d'être un arbre complet. Il suffit donc de montrer qu'elle transforme l'arbre étiqueté en un arbre maximier.

Terminaison de la boucle : stricte décroissance du contenu de la référence n (qui est divisée par 2 à chaque itération) et le contenu de n est supérieure ou égale à 1 (ce qui garantit que n n'est pas la racine de l'arbre).

Invariant de la boucle :

– au début de l'itération, p est père de n, les deux branches issues de p sont des files de priorité

– l'étiquette de p est supérieure ou égale à celle de son autre fils éventuel n'

La complexité de cette insertion est manifestement majorée par la hauteur de l'arbre, elle est donc un $O(\log_2 n)$ où n est le nombre de noeuds.

4.5.13 Tri en tas

Le principe général : on insère les éléments du tableau successivement dans un tas initialement vide puis on extrait itérativement la racine de ce tas (qui en est toujours le plus petit élément) jusqu'à ce que ce tas soit vide.

```
1   static void tri_en_tas(long [] v){
2         int n = v.length;
3         Tas h = new Tas();
4         for(int i =0; i<n ; i++) insere(v[i], h);
5         for(int i =0; i<n ; i++){ v[i] = h.contenu[0]; supprime_racine(h);
6   }
```

Programme 4.64 –

Comme l'insertion et la suppression de la racine ont une complexité en $O(\log_2 n)$, le tri en tas a une complexité en $O(n \log_2 n)$ dans le pire des cas.

Eviter le recours à un tas auxiliaire : économie de temps et de mémoire.

Propager la structure de tas à partir des sous-arbres de l'arbre complet obtenu à partir du tableau, en remontant la structure de tas depuis les noeuds terminaux jusqu'à la racine de l'arbre.

Les sous-arbres réduits aux noeuds terminaux sont évidemment des tas.

Soit n un noeud non terminal et supposons que la ou les branches issues de ce noeud soient des tas. Si le sous-arbre de racine n n'est pas un tas, il le deviendra en effectuant une percolation de l'étiquette de n à travers ce sous arbre. Après l'exécution complète de cette procédure, le tableau aura été transformé en un tas sans avoir utilisé d'insertions dans un autre tas.

```
1   private static void percole_aux(int i, Tas t, long racine){
2         int n=t.nb;     long[] v = t.contenu;
3         if(2*i+1 <  n){ /* le noeud a des fils */
4          int j = 2*i+1; /* le fils gauche */
5          int fils = (j+1 < n && v[j]<v[j+1]) ? j+1 : j ; /* le plus grand fils */
6          if(v[fils] > racine){
7                   v[i] = v[fils]; /* on remonte le fils */
8                   percole_aux(fils, t, racine);
9            } else {
10                 v[i] = racine; /* on stocke l'élément */
11           }
12         else v[i] = racine;  /* le noeud est terminal */
13  }
14   static Tas vec_en_tas{long[] v){
15          Tas t = new Tas();
```

```
16          for(int i = (n+1)/2; i>=0, i--) percole_aux(i, t, v[i]);
17          return t;
18  }
```

Programme 4.65 –

Invariant de la dernière boucle indexée par i :

– après exécution de l'itération d'indice i, tous les sous-arbres ayant pour racines $v[i], v[i+1] \ldots, v[n]$ sont des tas

La terminaison de la boucle est évidente puisqu'il s'agit d'une boucle indexée. Après l'exécution de l'itération d'indice 0, le tableau tout entier a été transformé en tas.

L'extraction successive des racines peut alors se faire directement en place : pour extraire la racine d'un tas (c'est à dire son plus grand élément), nous l'avons échangée avec le dernier élément du tas (ce qui la place donc en dernière position), puis nous l'avons supprimée (ce qui revient à diminuer de 1 la longueur de travail) et enfin nous avons effectué une percolation de l'étiquette du premier élément du tableau. En itérant cette méthode, nous allons donc obtenir un tableau dans lequel le plus grand élément sera en dernière position, puis l'élément juste un peu plus petit sera en avant-dernière position, et ainsi de suite. Autrement dit, notre tableau sera trié en ordre croissant.

Ceci nous conduit à la procédure suivant de tri par ordre croissant d'un tas :

```
1   private static void percole_aux(int i, long[] v, long racine, int n){
2         if(2*i+1 <  n){ /* le noeud a des fils */
3          int j = 2*i+1; /* le fils gauche */
4          int fils = (j+1 < n && v[j]<v[j+1]) ? j+1 : j ; /* le plus grand fils */
5          if(v[fils] > racine){
6                  v[i] = v[fils]; /* on remonte le fils */
7                  percole_aux(fils, t, racine);
8            } else {
9                v[i] = racine; /* on stocke l'élément */
10             }
11         else v[i] = racine;  /* le noeud est terminal */
12  }
13   static void tri_tas_prov{long[] v){
14          for(int n = v.length -1; n>=1 ; n--){
15                echange(0, n, v);
16                percole_aux(0, v, v[0], n-1);
17          }
18   }
```

Programme 4.66 –

Invariant de la boucle finale :

après l'itération d'indice n les éléments $v[0], \ldots, v[n-1]$ forment un tas et on a

$$v[0] \geq v[n] \geq v[n+1] \geq \cdots \geq v[N-1]$$

si N désigne la longueur du tableau.

Tri en tas d'un tableau :

```
1   static void tri_tas_prov{long[] v){
2          for(int i = (n+1)/2; i>=0, i--) percole_aux(i, v, v[i], v.length);
3          for(int n = v.length -1; n>=1 ; n--){
4                echange(0, n, v);
5                percole_aux(0, v, v[0], n-1);
6          }
7   }
```

Programme 4.67 –

```
1   #let v= [|8;1;2;3;4;3;8;5;1;8;2;4;12;14;1;3;5|];;
2   v : int vect = [|8; 1; 2; 3; 4; 3; 8; 5; 1; 8; 2; 4; 12; 14; 1; 3; 5|]
3   #tri_en_tas v; v;;
4   - : int vect = [|1; 1; 1; 2; 2; 3; 3; 3; 4; 4; 5; 5; 8; 8; 8; 12; 14|]
```

Programme 4.68 –

Comme la procédure `traite_noeud` a une complexité en $O(\log_2 n)$ et que les deux boucles sont au plus de longueur n, la complexité de la procédure de tri en tas est en $O(n \log_2 n)$ dans le pire des cas.

Chapitre 5

Eléments du calcul propositionnel

5.1 Les propositions

Définition 5.1.1 *On appelle proposition (ou assertion) toute phrase dont il est possible de dire si elle est vraie ou fausse.*

Exemple
- *Deux plus deux égale quatre* est une proposition qui est vraie.
- *Deux plus deux égale cinq* est une proposition qui est fausse.
- *Aucun homme n'a été sur Mars* est une proposition qui est vraie à l'heure actuelle mais qui peut devenir fausse dans l'avenir
- $x + y > 0$ n'est pas une proposition car le fait qu'elle soit vraie ou fausse dépend des valeurs prises par x et y, donc on ne peut pas dire si cette assertion est vraie ou fausse
- *Cette phrase est fausse* n'est pas une proposition ; en effet, si elle est vraie, c'est qu'elle est fausse et si elle est fausse, c'est qu'elle est vraie ; on ne peut donc lui attribuer aucune valeur logique.

5.2 Propositions composées

A partir d'une proposition p, on peut construire sa *négation* que nous noterons (très provisoirement) NON p. La règle à appliquer est que si p est vraie, alors NON p est fausse et que si p est fausse alors NON p est vraie.

A partir de deux propositions p et q, on peut construire leur *conjonction* que nous noterons provisoirement p ET q. La proposition p ET q est vraie si et seulement si les deux propositions p et q sont vraies, dans tous les autres cas elle est fausse.

A partir de deux propositions p et q, on peut construire leur *disjonction* que nous noterons provisoirement p OU q. La proposition p OU q est vraie si et seulement si l'une au moins des deux propositions p et q est vraie ; sinon elle est fausse.

Définition 5.2.1 *Si p et q désignent deux propositions, alors*
- *la proposition p IMPL q est vraie si et seulement si*
  - *soit q est vraie*
  - *soit p est fausse*
- *la proposition p EQUIV q est vraie si et seulement p et q sont*
  - *soit simultanément vraies*
  - *soit simultanément fausses*

5.3 Syntaxe des formules logiques

Des variables (en général p, q, r éventuellement indexées en $p_1, \ldots, p_n$, $q_1, \ldots$, $r_1, \ldots$) auxquelles nous pourrons substituer n'importe quelles propositions : *variables propositionnelles.*

Nous remplacerons
- l'opérateur de négation NON par le symbole $\neg$
- l'opérateur de conjonction ET par le symbole $\wedge$
- l'opérateur de disjonction OU par le symbole $\vee$
- l'opérateur d'exclusion OUX par le symbole $\otimes$
- l'opérateur d'implication IMPL par le symbole $\Rightarrow$
- l'opérateur d'équivalence EQUIV par le symbole $\Longleftrightarrow$

Ensemble A formé
- des variables propositionnelles,
- des symboles d'opérateurs logiques des parenthèses ouvrante et fermante,

$$p, q, r, p_1, \ldots, q_1, \ldots, r_1, \ldots, \neg, \wedge, \vee, \otimes, \Rightarrow, \Longleftrightarrow, (\ ,\ )$$

et l'ensemble A^* des suites finies d'éléments de A (écrites par juxtaposition de gauche à droite).

Définition 5.3.1 *On appelle ensemble des formules logiques le sous ensemble de A^* défini inductivement par*
- *toute variable propositionnelle est une formule logique*
- *si F est une formule logique $(\neg F)$ est une formule logique*
- *si F_1 et F_2 sont deux formules logiques, alors $(F_1 \wedge F_2)$, $(F_1 \vee F_2)$, $(F_1 \otimes F_2)$, $(F_1 \Rightarrow F_2)$ et $(F_1 \Longleftrightarrow F_2)$ sont des formules logiques*

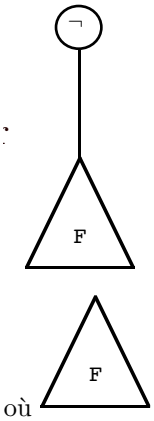
Autorisé : oublier une éventuelle parenthèse ouvrante initiale et la parenthèse fermante finale correspondante.

Provisoirement : pas de règle de priorité sur les opérateurs logiques et en conséquence nous ne nous autoriserons pas à supprimer de quelconques parenthèses intermédiaires.

Exemple
- $((\neg p_1) \vee p_2) \Longleftrightarrow (q_1 \wedge q_2)$ est une formule logique
- $\neg p_1 \vee p_2$ n'est pas une formule logique (sans ordre de priorité sur les opérateurs, il y ambiguïté sur la manière de placer des parenthèses : $(\neg p_1) \vee p_2$ ou $\neg (p_1 \vee p_2)$?)
- $p \vee (q \wedge r$ n'est pas une formule logique (manque une parenthèse fermante)

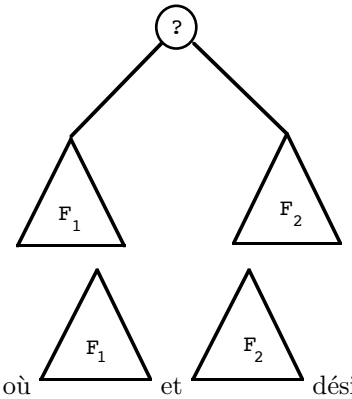
5.4 Représentation arborescente d'une formule logique

- une variable propositionnelle est représentée par son symbole dans un cercle
- si F est une formule logique, $(\neg F)$ est représentée par



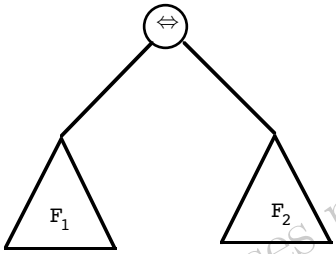
où

représente la formule F.
- $(F_1 \wedge F_2)$, $(F_1 \vee F_2)$, $(F_1 \otimes F_2)$, $(F_1 \Rightarrow F_2)$ et $(F_1 \Longleftrightarrow F_2)$ sont représentées par



où et désignent les représentations arborescentes respectives des formules F_1 et F_2 et où le point d'interrogation désigne l'un des opérateurs $\vee, \wedge, \otimes, \Rightarrow, \iff$.

Exemple La formule logique $((\neg\, p_1) \vee p_2) \iff (q_1 \wedge q_2)$ admet la représentation arborescente

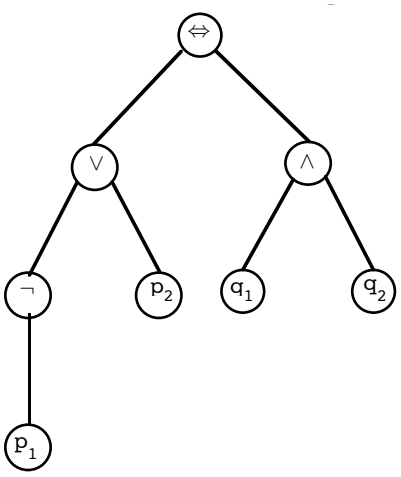


Exemple où et désignent les représentations arborescentes respectives de $(\neg\, p_1) \vee p_2$ et $q_1 \wedge q_2$,

Exemple soit encore et .

Exemple

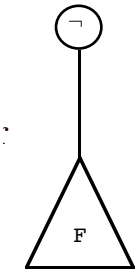
Représentation arborescente pour l'expression totale.



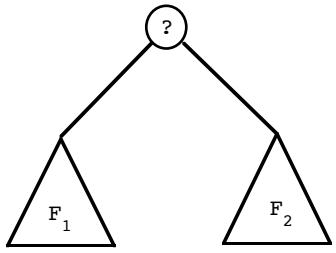
A partir d'un arbre dont les *feuilles* sont des variables propositionnelles, les *noeuds* d'ordre 1 sont *étiquetés* par le symbole $\neg$ et les noeuds d'ordre 2 sont étiquetés par l'un des symboles $\vee, \wedge, \otimes, \Rightarrow, \Longleftrightarrow$, il est facile de reconstruire la formule logique de départ en appliquant récursivement les règles

– 

donne la variable propositionnelle p

– 

donne la formule $(\neg F)$

– 

donne la formule $(F_1 \ ? \ F_2)$

5.5 Sémantique : évaluation des formules logiques

A chaque formule logique, nous allons maintenant assigner une signification logique lorsque les variables propositionnelles représentent des propositions susceptibles d'être vraies ou fausses. Pour faciliter le calcul algébrique sur les formules logiques, il est habituel de remplacer les mots *vrai* et *faux* qui sont les deux valeurs prises par les propositions par V et F ou encore mieux par 1 et 0. C'est cette dernière convention que nous suivrons par la suite.

Nous allons commencer par évaluer les opérateurs logiques $\neg, \wedge, \vee, \otimes, \Rightarrow, \Leftrightarrow$ en les rapprochant des opérateurs correspondants introduit précédemment sur les propositions, à savoir NON, ET, OU, OUX, IMPL, EQUIV. A cet effet nous introduirons les fonctions $f_\neg, f_\wedge, f_\vee, f_\otimes, f_\Rightarrow, f_\Leftrightarrow$ définies

– la première sur $\{0,1\}$
– les suivantes sur $\{0,1\}^2$

et à valeurs dans $\{0,1\}$, grâce aux formules suivantes (qui suivent exactement les valeurs de vérité de NON, ET, OU, OUX, IMPL, EQUIV).

$$f_\neg(0)=1 \quad f_\neg(1)=0 \quad ' \quad f_\wedge(1,1)=1 \quad f_\wedge(1,0)=f_\wedge(0,1)=f_\wedge(0,0)=0 \quad '$$
$$f_\vee(0,0)=0 \quad f_\vee(1,0)=f_\vee(0,1)=f_\vee(1,1)=1$$
$$f_\otimes(1,1)=f_\otimes(0,0)=0 \quad f_\otimes(1,0)=f_\otimes(0,1)=1 \quad ' \quad f_\Rightarrow(1,0)=0 \quad f_\Rightarrow(1,1)=f_\Rightarrow(0,1)=f_\Rightarrow(0,0)=1$$
$$f_\Leftrightarrow(1,1)=f_\Leftrightarrow(0,0)=1 \quad f_\Leftrightarrow(1,0)=f_\Leftrightarrow(0,1)=0$$

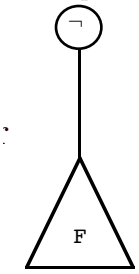
Soit σ une application de l'ensemble des valeurs propositionnelles dans $\{0,1\}$ (c'est à dire que l'on assigne à chaque variable propositionnelle l'une des valeurs *faux* ou *vrai*).

On définit alors facilement la valeur d'une formule logique élémentaire, par exemple par $\mathrm{Val}(p\vee q,\sigma)=f_\vee(\sigma(p),\sigma(q))$.

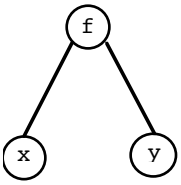
Nous allons étendre cette évaluation à toute formule logique de manière inductive grâce aux règles suivantes :
- si $F=p$ est une variable propositionnelle, $\mathrm{Val}(F,\sigma)=\sigma(p)$
- si $F=(\neg F_1)$, alors $\mathrm{Val}(F,\sigma)=f_\neg(\mathrm{Val}(F_1,\sigma))$ (autrement dit la valeur de F est l'opposée de la valeur de F_1
- si $F=(F_1\wedge F_2)$ alors $\mathrm{Val}(F,\sigma)=f_\wedge(\mathrm{Val}(F_1,\sigma),\mathrm{Val}(F_2,\sigma))$
- si $F=(F_1\vee F_2)$ alors $\mathrm{Val}(F,\sigma)=f_\vee(\mathrm{Val}(F_1,\sigma),\mathrm{Val}(F_2,\sigma))$
- si $F=(F_1\otimes F_2)$ alors $\mathrm{Val}(F,\sigma)=f_\otimes(\mathrm{Val}(F_1,\sigma),\mathrm{Val}(F_2,\sigma))$
- si $F=(F_1\Rightarrow F_2)$ alors $\mathrm{Val}(F,\sigma)=f_\Rightarrow(\mathrm{Val}(F_1,\sigma),\mathrm{Val}(F_2,\sigma))$
- si $F=(F_1\Leftrightarrow F_2)$ alors $\mathrm{Val}(F,\sigma)=f_\Leftrightarrow(\mathrm{Val}(F_1,\sigma),\mathrm{Val}(F_2,\sigma))$

Remarque Cette évaluation des formules logiques est encore équivalente à l'opération suivante (très facile à exécuter à l'aide d'un logiciel de calcul formel) :
- on prend la représentation arborescente de la formule logique ;
- dans cette représentation arborescente, on commence par remplacer dans toutes les feuilles les variables propositionnelles p par leurs valeurs $\sigma(p)\in\{0,1\}$
- on remplace ensuite dans chaque noeud les symboles logiques $\neg,\wedge,\vee,\otimes,\Rightarrow$ et $\Leftrightarrow$ par la fonction associée

Remarque
- on simplifie alors l'arbre obtenu avec les conventions que si $x,y\in\{0,1\}$,



se simplifie en $f(x)$ et que



se simplifie en $f(x,y)$

5.6 Tables de vérité

Soit F une formule logique dépendant de n variables propositionnelles distinctes (chacune d'entre elles pouvant apparaître plusieurs fois dans la formule).

A chacune des 2^n applications σ de l'ensemble des variables propositionnelles dans $\{0,1\}$, on sait donc associer la valeur de F dans l'environnement σ, notée $\mathrm{Val}(F,\sigma) \in \{0,1\}$.

On regroupe dans un même tableau les différentes valeurs σ attribuées aux variables propositionnelles et l'évaluation de F correspondante.

On construit un tableau à $n+1$ colonnes et à 2^n lignes. Dans les n premières colonnes on met les valeurs 0 ou 1 attribuées aux valeurs propositionnelles (avec par exemple un ordre lexicographique consistant à faire varier d'abord la n-ième variable propositionnelle, puis la $n-1$-ième et ainsi de suite jusqu'à la première) et dans la dernière colonne on met la valeur correspondante de $\mathrm{Val}(F,\sigma)$.

Un tel tableau est appelé une table de vérité.

Commençons par construire les tables de vérité des opérations élémentaires sur les variables propositionnelles données par les opérateurs $\neg, \wedge, \vee, \otimes, \Rightarrow$ et $\Leftrightarrow$.

p	$\neg p$
1	0
0	1

p	q	$p \wedge q$
1	1	1
1	0	0
0	1	0
0	0	0

p	q	$p \vee q$
1	1	1
1	0	1
0	1	1
0	0	0

p	q	$p \otimes q$
1	1	0
1	0	1
0	1	1
0	0	0

p	q	$p \Rightarrow q$
1	1	1
1	0	0
0	1	1
0	0	1

p	q	$p \Leftrightarrow q$
1	1	1
1	0	0
0	1	0
0	0	1

La table de vérité d'une formule logique plus complexe peut alors se construire en introduisant des colonnes intermédiaires pour les sous formules. C'est ainsi que l'on pourra construire en plusieurs étapes la table de vérité de la formule logique

$$(((\neg p) \vee q) \wedge r) \iff (p \otimes r)$$

de la manière suivante

p	q	r	$\neg p$	$(\neg p) \vee q$	$((\neg p) \vee q) \wedge r$	$p \otimes r$	$(((\neg p) \vee q) \wedge r) \iff (p \otimes r)$
1	1	1	0	1	1	0	0
1	1	0	0	1	0	1	0
1	0	1	0	0	0	0	1
1	0	0	0	0	0	1	0
0	1	1	1	1	1	1	1
0	1	0	1	1	0	0	1
0	0	1	1	1	1	1	1
0	0	0	1	1	0	0	1

5.7 Tautologies, satisfiabilité

On appelle tautologie toute formule logique qui est vraie quelles que soient les valeurs que l'on attribue aux variables propositionnelles. C'est donc une formule dont la table de vérité ne comprend que des 1 dans la dernière colonne, ou encore qui satisfait à la définition suivante.

Définition 5.7.1 *On dit qu'une formule logique F est une tautologie si pour toute application σ de l'ensemble des variables propositionnelles dans $\{0,1\}$, l'évaluation de F dans l'environnement σ vérifie* $\mathrm{Val}(F,\sigma) = 1$.

Exemple La formule logique $(p \Rightarrow q) \iff ((\neg p) \vee q)$ est une tautologie. En effet la table de vérité de cette formule est

p	q	$p \Rightarrow q$	$\neg p$	$(\neg p) \vee q$	$(p \Rightarrow q) \iff ((\neg p) \vee q)$
1	1	1	0	1	1
1	0	0	0	0	1
0	1	1	1	1	1
0	0	1	1	1	1

Une formule est satisfiable si l'on peut attribuer aux variables propositionnelles des valeurs pour lesquelles la formule est vraie. De manière précise, on a la définition suivante

Définition 5.7.2 *On dit qu'une formule logique F est satisfiable s'il existe une application σ de l'ensemble des variables propositionnelles dans $\{0,1\}$ telle que* $\mathrm{Val}(F,\sigma) = 1$.

Une formule qui n'est pas satisfiable est dite contradictoire ; on dit encore que c'est une contradiction. Une formule F est contradictoire si et seulement si la formule $(\neg F)$ est une tautologie. On constate ainsi que le problème de tester si une formule est une tautologie ou de tester si une formule est satisfiable sont intimement reliés.

Moyen évident de tester si une formule logique est satisfiable : construire la table de vérité de la formule et de tester l'existence d'un 1 dans la dernière colonne. Cela revient à donner les 2^n valeurs possibles à la famille des n variables propositionnelles qui interviennent dans la formule et à évaluer la formule dans chacun de ces environnements. Le temps de calcul de cette méthode est clairement proportionnel à 2^n.

On ne connait à l'heure actuelle aucun algorithme réellement plus performant que cet essai systématique et on ne sait même pas s'il peut en exister.

Ce problème de la satisfiabilité d'une formule logique entre dans une catégorie de problèmes dont on peut montrer qu'ils sont en un certain sens équivalents, les problèmes NP-complets, problèmes dont à l'heure actuelle on ne connait que des solutions dont le temps de calcul est exponentiel en la taille des données. Parmi ceux ci on peut citer le problème du déménageur (optimiser le chargement d'un camion avec des colis de différentes tailles) et le problème du voyageur de commerce (trouver le plus court chemin reliant certaines villes données sans repasser deux fois dans la même ville).

5.8 Fonctions booléennes

5.9 Fonctions booléennes

Définition 5.9.1 *On appelle fonction booléenne toute application $f : \{0,1\}^n \to \{0,1\}$.*

Définition 5.9.2 *Soit F une formule logique. Soit $p_1, \ldots, p_n$ un ensemble ordonné de variables propositionnelles contenant*[1] *toutes les variables propositionnelles intervenant dans F. On associe à F la fonction booléenne f_F de $\{0,1\}^n$ dans $\{0,1\}$ définie par $f_F(x_1, \ldots, x_n) = \mathrm{Val}(F,\sigma)$ où σ est définie par $\forall i \in [1,n],\ \sigma(p_i) = x_i$.*

La fonction booléenne associée à une formule logique et à la famille de ses variables propositionnelles se lit directement sur la table de vérité de la formule logique. C'est la fonction qui aux éléments des n premières colonnes associe l'élément correspondant de la dernière colonne.

Exemple En reprenant une table de vérité précédente, si $F = (((\neg p) \vee q) \wedge r) \iff (p \otimes r)$, on a la table de vérité

p	q	r	$\neg p$	$(\neg p) \vee q$	$((\neg p) \vee q) \wedge r$	$p \otimes r$	$(((\neg p) \vee q) \wedge r) \iff (p \otimes r)$
1	1	1	0	1	1	0	0
1	1	0	0	1	0	1	0
1	0	1	0	0	0	0	1
1	0	0	0	0	0	1	0
0	1	1	1	1	1	1	1
0	1	0	1	1	0	0	1
0	0	1	1	1	1	1	1
0	0	0	1	1	0	0	1

1. Pour des raisons que l'on verra lorsqu'interviennent plusieurs formules logiques, il vaut mieux ne pas se limiter strictement aux variables intervenant effectivement dans la formule F.

ce qui donne $f_F(1,0,1) = f_F(0,1,1) = f_F(0,1,0) = f_F(0,0,1) = f_F(0,0,0) = 1$ et $f_F(1,1,1) = f_F(1,1,0) = f_F(1,0,0) = 0$.

Remarque Une formule F est une tautologie si et seulement si $f_F = 1$. Une formule est satisfiable si et seulement si $f_F \neq 0$.

5.10 Equivalence des formules logiques

Définition 5.10.1 *On dit que deux formules logiques F_1 et F_2 sont équivalentes si pour toute application σ de l'ensemble des variables propositionnelles dans $\{0,1\}$, on a $\mathrm{Val}(F_1,\sigma) = \mathrm{Val}(F_2,\sigma)$. On notera alors $F_1 \equiv F_2$.*

Théorème 5.10.1 *Soit F_1 et F_2 deux formules logiques, $p_1,\ldots,p_n$ un ensemble ordonné de variables propositionnelles contenant toutes les variables propositionnelles intervenant soit dans F_1 soit dans F_2, f_{F_1} et f_{F_2} les fonctions booléennes associées à F_1 et F_2. On a équivalence de*

1. $F_1 \equiv F_2$
2. $f_{F_1} = f_{F_2}$
3. $(F_1 \Leftrightarrow F_2)$ *est une tautologie*

Démonstration: L'équivalence des deux premières assertions est évidente puisque seules les valeurs attribuées à $p_1,\ldots,p_n$ interviennent dans le calcul de $\mathrm{Val}(F_1,\sigma)$ et de $\mathrm{Val}(F_2,\sigma)$. L'équivalence entre la première assertion et la troisième résulte de ce que $\mathrm{Val}(F_1 \Leftrightarrow F_2) = f_{\Leftrightarrow}(\mathrm{Val}(F_1,\sigma),\mathrm{Val}(F_2,\sigma))$ qui d'après la définition de $f_{\Leftrightarrow}$ vaut 1 si et seulement si $\mathrm{Val}(F_1,\sigma) = \mathrm{Val}(F_2,\sigma)$.

Proposition 5.10.1 *L'équivalence des formules logiques est une relation d'équivalence sur l'ensemble des formules logiques.*

Démonstration: La définition même montre que cette relation est à la fois réfléxive, symétrique et transitive.

Théorème 5.10.2 (invariance de l'équivalence par substitution) *Soit F_1 et F_2 deux formules logiques équivalentes dépendant des variables propositionnelles $p_1,\ldots,p_n$ et soit $G_1,\ldots,G_n$ une famille de n formules logiques. Soit F_1' et F_2' les formules logiques obtenues en remplaçant dans F_1 et F_2 les variables $p_1,\ldots,p_n$ par $G_1,\ldots,G_n$. Alors F_1' et F_2' sont encore équivalentes.*

Démonstration: Soit $q_1,\ldots,q_p$ les variables propositionnelles intervenant dans $G_1,\ldots,G_n$. Soit σ une application de $\{q_1,\ldots,q_p\}$ dans $\{0,1\}$. On démontre immédiatement par induction structurelle sur F_1 que $\mathrm{Val}(F_1',\sigma) = f_{F_1}(Val(G_1,\sigma),\ldots,Val(G_n,\sigma))$. Mais comme F_1 et F_2 sont équivalentes, on a $f_{F_1} = f_{F_2}$, ce qui montre que $\forall\sigma,\ \mathrm{Val}(F_1',\sigma) = \mathrm{Val}(F_2',\sigma)$, donc F_1' et F_2' sont équivalentes.

Remarque On s'autorisera par la suite à écrire 1 pour une tautologie générique et 0 pour une contradiction générique, si bien que
- F *est une tautologie* s'écrira $F \equiv 1$
- F *est une contradiction* s'écrira $F \equiv 0$

5.11 Equivalences fondamentales

Tant qu'on ne s'intéresse qu'à la validité d'un raisonnement ou d'une assertion, on peut remplacer toute formule logique portant sur des variables propositionnelles $p_1,\ldots,p_n$ par une formule logique équivalente. Ceci nous conduit à examiner un certain nombre d'équivalences fondamentales.

Proposition 5.11.1 *Soit F_1, F_2 et F_3 trois formules logiques. Alors*
- $F_1 \vee F_2 \equiv F_2 \vee F_1$ *et* $F_1 \wedge F_2 \equiv F_2 \wedge F_1$ *(*commutativité *de la disjonction et de la conjonction)*

- $(F_1 \vee F_2) \vee F_3 \equiv F_1 \vee (F_2 \vee F_3)$ *et* $(F_1 \wedge F_2) \wedge F_3 \equiv F_1 \wedge (F_2 \wedge F_3)$ *(*associativité *de la disjonction et de la conjonction)*
- $(F_1 \vee F_2) \wedge F_3 \equiv (F_1 \wedge F_3) \vee (F_2 \wedge F_3)$ *et* $(F_1 \wedge F_2) \vee F_3 \equiv (F_1 \vee F_3) \wedge (F_2 \vee F_3)$ *(*distributivité *de la conjonction par rapport à la disjonction et de la disjonction par rapport à la conjonction)*

Démonstration: D'après l'invariance de l'équivalence par substitution, il suffit de montrer ces équivalences lorsque F_1, F_2 et F_3 sont trois variables propositionnelles p_1, p_2 et p_3. La vérification est évidente sur les tables de vérités pour la première et la deuxième assertion.

Démonstration: Pour la troisième, on peut construire les tables de vérités et on obtient par exemple pour la distributivité de la conjonction par rapport à la disjonction

p_1	p_2	p_3	$p_1 \vee p_2$	$(p_1 \vee p_2) \wedge p_3$	$p_1 \wedge p_3$	$p_2 \wedge p_3$	$(p_1 \wedge p_3) \vee (p_2 \wedge p_3)$
1	1	1	1	1	1	1	1
1	1	0	1	0	0	0	0
1	0	1	1	1	1	0	1
1	0	0	1	0	0	0	0
0	1	1	1	1	0	1	1
0	1	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0

Remarque A partir de maintenant, nous jouerons sur l'associativité de la disjonction pour noter $F_1 \vee \ldots \vee F_n$ à la place de $(\ldots((F_1 \vee F_2) \vee F_3)\ldots) \vee F_{n-1}) \vee F_n)$. Nous procéderons de même pour la conjonction.

Proposition 5.11.2 *Soit F_1 et F_2 deux formules logiques. Alors*
- $\neg(\neg F_1) \equiv F_1$ *(loi du* tiers exclu*)*
- $\neg(F_1 \vee F_2) \equiv (\neg F_1) \wedge (\neg F_2)$ *et* $\neg(F_1 \wedge F_2) \equiv (\neg F_1) \vee (\neg F_2)$ *(lois de Morgan)*

Démonstration: Comme précédemment, on utilise le théorème d'invariance par substitution pour se limiter au cas où les deux formules sont des variables propositionnelles. Il suffit alors d'établir les tables de vérité.

Proposition 5.11.3 *Deux tautologies sont équivalentes. Deux contradictions sont équivalentes. Soit F une formule logique.*
- *si T est une tautologie, alors $F \vee T \equiv T$ et $F \wedge T \equiv F$*
- *si C est une contradiction, alors $F \vee C \equiv F$ et $F \wedge C \equiv C$*
- *$F \vee (\neg F)$ est une tautologie, $F \wedge (\neg F)$ est une contradiction.*

Démonstration: Même méthode en remplaçant F par une variable propositionnelle et la tautologie (resp. la contradiction) par une variable propositionnelle assujettie à ne prendre que la valeur 1 (resp. 0).

Proposition 5.11.4 *Soit F_1 et F_2 deux formules logiques. Alors*
- $F_1 \Rightarrow F_2 \equiv (\neg F_1) \vee F_2$ *(expression de l'implication en fonction de la négation et de la disjonction)*
- $F_1 \Rightarrow F_2 \equiv (\neg F_2) \Rightarrow (\neg F_1)$ *(loi de* contraposition*)*
- $F_1 \Leftrightarrow F_2 \equiv (F_1 \Rightarrow F_2) \wedge (F_2 \Rightarrow F_1)$

Démonstration: Même méthode. La deuxième assertion peut également se montrer à l'aide de la première et de la loi du tiers exclu en écrivant

$$F_1 \Rightarrow F_2 \equiv (\neg F_1) \vee F_2 \equiv (\neg(\neg F_2)) \vee (\neg F_1) \equiv (\neg F_2) \Rightarrow (\neg F_1)$$

Proposition 5.11.5 *Soit F_1 et F_2 deux formules logiques. Alors* $F_1 \otimes F_2 \equiv (F_1 \wedge (\neg F_2)) \vee ((\neg F_1) \wedge F_2)$

Démonstration: Idem

5.12 Formes normales des formules logiques

Définition 5.12.1 *On appelle disjonction toute formule logique F s'écrivant $F = F_1 \vee \ldots \vee F_n$ où chaque F_i est un* littéral*, c'est à dire soit une variable propositionnelle p_i, soit une négation de variable propositionnelle $\neg p_i$.*

Définition 5.12.2 *On appelle conjonction toute formule logique F s'écrivant $F = F_1 \wedge \ldots \wedge F_n$ où chaque F_i est soit une variable propositionnelle p_i soit une négation de variable propositionnelle $\neg p_i$.*

Si une variable p apparaît plusieurs fois dans une conjonction, on peut jouer sur la commutativité et les différentes formules $p \wedge p \equiv p$, $(\neg p) \wedge (\neg p) \equiv \neg p$, $(p \wedge (\neg p)) \equiv 0$ pour transformer la conjonction en une conjonction équivalente où ne figure qu'une seule fois la variable p ou en une contradiction 0.

De même, si la variable p apparaît plusieurs fois dans une disjonction, on peut jouer sur la commutativité et les formules $p \vee p \equiv p$, $(\neg p) \vee (\neg p) \equiv \neg p$, $(p \vee (\neg p)) \equiv 1$ pour transformer la conjonction en une conjonction équivalente où ne figure qu'une seule fois la variable p ou en une tautologie 1.

A équivalence près, on pourra toujours supposer que les variables propositionnelles apparaissent au plus une fois dans les conjonctions ou disjonctions.

Lemme 5.12.1 *Si F est une conjonction, alors $\neg F$ est équivalente à une disjonction. Si F est une disjonction, alors $\neg F$ est équivalente à une conjonction.*

> **Démonstration**: En effet, si F s'écrit$F = F_1 \vee \ldots \vee F_n$ où chaque F_i est soit une variable propositionnelle p_i, soit une négation de variable propositionnelle $\neg p_i$, d'après les lois de Morgan, $\neg F \equiv (\neg F_1) \wedge \ldots \wedge (\neg F_p)$ et chacune des $\neg F_i$ est soit une négation de variable propositionnelle (si F_i est une variable propositionnelle), soit équivalente à une variable propositionnelle (si F_i est une négation de variable propositionnelle). Donc $\neg F$ est équivalente à une conjonction. La démonstration est similaire pour une conjonction F.

Théorème 5.12.1 *Toute formule logique F est équivalente à une formule logique $F' = C_1 \vee \ldots \vee C_p$ où chacune des C_i est une conjonction ; autrement dit toute formule logique est équivalente à une disjonction de conjonctions. De même, toute formule logique est équivalente à une formule logique $F'' = D_1 \wedge \ldots \wedge D_q$ où chacune des D_j est une disjonction ; autrement dit toute formule logique est équivalente à une conjonction de disjonctions.*

> **Démonstration**: On montre les deux résultats à la fois par induction structurelle. Si F est réduite à une variable propositionnelle, alors F est à la fois une conjonction et une disjonction, donc le résultat est clair.
>
> Si $F = \neg F_1$ où F_1 est équivalente à une disjonction de conjonctions, on a $F_1 \equiv C_1 \vee \ldots \vee C_p$ et alors, d'après la loi de Morgan, $F = \neg F_1 \equiv (\neg C_1) \wedge \ldots \wedge (\neg C_p) \equiv D_1 \wedge \ldots \wedge D_p$ si $\neg C_i \equiv D_i$ où D_i est une disjonction (d'après le lemme précédent). De même, si F_1 est équivalente à une conjonction de disjonctions, alors F est équivalente à une disjonction de conjonctions.

> **Démonstration**:
>
> Si $F = F_1 \vee F_2$ où F_1 et F_2 sont équivalentes à des disjonctions de conjonctions, on a $F_1 \equiv C_1 \vee \ldots \vee C_p$ et $F_2 \equiv C'_1 \vee \ldots \vee C'_q$ d'où $F_1 \vee F_2 \equiv C_1 \vee \ldots \vee C_p \vee C'_1 \vee \ldots \vee C'_q$ qui est encore une disjonction de conjonctions. Si $F = F_1 \vee F_2$ où F_1 et F_2 sont équivalentes à des conjonctions de disjonctions, on a $F_1 \equiv D_1 \wedge \ldots \wedge D_p$ et $F_2 \equiv D'_1 \wedge \ldots \wedge D'_q$, d'où, par distributivité de la disjonction par rapport à la conjonction
>
> $$F_1 \vee F_2 \equiv (D_1 \wedge \ldots \wedge D_p) \vee (D'_1 \wedge \ldots \wedge D'_q) = \bigwedge_{i,j} (D_i \vee D'_j)$$
>
> où chacune des $D_i \vee D'_j$ est de façon évidente une disjonction.
>
> On montre de même que si $F = F_1 \wedge F_2$ où F_1 et F_2 sont à la fois des disjonctions de conjonctions et des conjonctions de disjonctions, il en est de même de F. (On peut aussi utiliser la loi de Morgan qui dit que $F \equiv \neg ((\neg F_1) \vee (\neg F_2))$ pour se ramener aux deux opérations précédentes.)

Démonstration:
En ce qui concerne $F = F_1 \otimes F_2$, cela résulte alors de $F \equiv (F_1 \wedge (\neg F_2)) \vee ((\neg F_1) \wedge F_2)$ et des résultats déjà démontrés. Il en est de même pour $F = F_1 \Rightarrow F_2 \equiv (\neg F_1) \vee F_2$. L'équivalence $F_1 \Leftrightarrow F_2 \equiv (F_1 \Rightarrow F_2) \wedge (F_2 \Rightarrow F_1)$ garantit alors la véracité du résultat pour $F = F_1 \Leftrightarrow F_2$.

Exemple Soit $F = (((\neg p) \vee q) \wedge r) \Rightarrow (p \otimes r)$. On a alors

$$\begin{aligned} F &\equiv (\neg (((\neg p) \vee q) \wedge r)) \vee (p \otimes r) \equiv (\neg ((\neg p) \vee q) \vee (\neg r)) \vee ((p \wedge (\neg r)) \vee ((\neg p) \wedge r) \\ &\equiv (p \wedge (\neg q)) \vee (\neg r) \vee (p \wedge (\neg r)) \vee ((\neg p) \wedge r) \end{aligned}$$

qui est une disjonction de conjonctions.

Pour obtenir une conjonction de disjonctions, il suffit d'utiliser la distributivité de $\vee$ par rapport à $\wedge$ et d'utiliser les règles de simplification pour ne faire apparaître les variables qu'une seule fois dans chaque disjonction. On a alors

Exemple

$$\begin{aligned} F &\equiv (p \wedge (\neg q)) \vee (\neg r) \vee (p \wedge (\neg r)) \vee ((\neg p) \wedge r) \\ &\equiv \Big((p \vee (\neg r)) \wedge ((\neg q) \vee (\neg r))\Big) \vee \Big((p \vee (\neg p) \wedge (p \vee r) \\ &\qquad \wedge((\neg r) \vee (\neg p)) \wedge ((\neg r) \vee r)\Big) \\ &\equiv \Big((p \vee (\neg r)) \wedge ((\neg q) \vee (\neg r))\Big) \vee \Big(1 \wedge (p \vee r) \wedge ((\neg r) \vee (\neg p)) \wedge 1\Big) \\ &\equiv \Big((p \vee (\neg r)) \wedge ((\neg q) \vee (\neg r))\Big) \vee \Big((p \vee r) \wedge ((\neg r) \vee (\neg p))\Big) \\ &\equiv \Big(p \vee (\neg r) \vee p \vee r\Big) \wedge \Big(p \vee (\neg r) \vee (\neg r) \vee (\neg p)\Big) \\ &\quad \wedge\Big((\neg q) \vee (\neg r) \vee p \vee r\Big) \wedge \Big((\neg q) \vee (\neg r) \vee (\neg r) \vee (\neg p)\Big) \\ &\equiv \Big(p \vee 1\Big) \wedge \Big(1 \vee (\neg r)\Big) \wedge \Big(p \vee (\neg q) \vee 1\Big) \wedge \Big((\neg p) \vee (\neg q) \vee (\neg r)\Big) \\ &\equiv 1 \wedge 1 \wedge 1 \wedge (\neg p) \vee (\neg q) \vee (\neg r) \equiv (\neg p) \vee (\neg q) \vee (\neg r) \end{aligned}$$

qui est une conjonction d'une seule disjonction.

5.13 Utilisation de tables de vérité

Les méthodes que nous venons de voir pour transformer une formule logique en une formule équivalente qui est soit une conjonction de disjonctions, soit une disjonction de conjonctions sont en fait basées sur des règles formelles de simplification de formules logiques à équivalence près (commutativité, distributivité, etc.).

Nous allons maintenant voir d'autres méthodes qui utilisent la fonction booléenne associée à la formule logique.

Lemme 5.13.1 *Soit $p_1, \ldots, p_n$ des variables propositionnelles et $\sigma : \{p_1, \ldots, p_n\} \to \{0,1\}$. Posons $F_i = p_i$ si $\sigma(p_i) = 1$ et $F_i = (\neg p_i)$ si $\sigma(p_i) = 0$. Soit C_σ la conjonction $C_\sigma = F_1 \wedge \ldots \wedge F_n$. Alors pour toute application $\tau : \{p_1, \ldots, p_n\} \to \{0,1\}$, on a* $\mathrm{Val}(C_\sigma, \tau) = \begin{array}{ll} 1 & \text{si } \sigma = \tau \\ 0 & \text{si } \sigma \neq \tau \end{array}$.

Démonstration: En effet $\mathrm{Val}(F_\sigma, \tau)$ vaut 1 si et seulement si chacun des $\mathrm{Val}(F_i, \tau)$ vaut 1, c'est à dire si $\tau(p_i) = \begin{array}{ll} 1 & \text{si } \sigma(p_i) = 1 \\ 0 & \text{si } \sigma(p_i) = 0 \end{array}$ ce qui équivaut à dire que $\tau = \sigma$.

Théorème 5.13.1 *Soit F une formule logique, $p_1, \ldots, p_n$ les variables propositionnelles intervenant dans F et pour toute application $\sigma : \{p_1, \ldots, p_n\} \to \{0,1\}$, soit C_σ la conjonction définie ci dessus. Soit Σ l'ensemble des applications $\sigma : \{p_1, \ldots, p_n\} \to \{0,1\}$ telles que* $\mathrm{Val}(F, \sigma) = 1$. *Alors F est équivalente à la disjonction de conjonctions $F \equiv \bigvee_{\sigma \in \Sigma} C_\sigma$.*

Démonstration: Posons $G = \bigvee_{\sigma\in\Sigma} C_\sigma$ et soit τ une application de $\{p_1,\dots,p_n\}$ dans $\{0,1\}$. Alors $\mathrm{Val}(G,\tau) = 1$ si et seulement si au moins l'un des $\mathrm{Val}(C_\sigma,\tau)$ vaut 1 pour un $\sigma \in \Sigma$, autrement dit si et seulement si τ est l'un des $\sigma \in \Sigma$, c'est à dire si et seulement si $\tau \in \Sigma$. Donc $\mathrm{Val}(G,\tau) = 1$ si et seulement si $\mathrm{Val}(F,\tau) = 1$. Ceci montre bien que $\forall\tau,\ \mathrm{Val}(G,\tau) = \mathrm{Val}(F,\tau)$. Donc F et G sont équivalentes.

Définition 5.13.1 *La formule logique $\bigvee_{\sigma\in\Sigma} C_\sigma$ du théorème précédent s'appelle la forme normale disjonctive de la formule logique F. Les C_σ sont appelés des* minterms *(prononcer mine-termes).*

Remarque Toutes les conjonctions intervenant dans la forme normale disjonctive de F ont la particularité de contenir toutes les variables propositionnelles de F une et une seule fois, soit par elle mêmes, soit par leur négation.

On peut construire la forme normale disjonctive à partir de la table de vérité de la formule F. On commence par supprimer toutes les lignes pour lesquelles l'évaluation de F donne 0 (correspondant aux σ telles que $\mathrm{Val}(F,\sigma) \neq 1$). Pour chaque ligne restante on construit la conjonction obtenue en prenant toutes les variables propositionnelles, celles ayant pour valeur 1 dans la ligne étant prises directement, celles ayant pour valeur 0 étant niées (la conjonction obtenue n'est autre que C_σ). On prend alors la disjonction de toutes les conjonctions ainsi obtenues.

Exemple Repartons de $F = (((\neg p) \vee q) \wedge r) \Leftrightarrow (p \otimes r)$. On a précédemment construit la table de vérité

p	q	r	$\neg p$	$(\neg p) \vee q$	$((\neg p) \vee q) \wedge r$	$p \otimes r$	$(((\neg p) \vee q) \wedge r) \iff (p \otimes r)$
1	1	1	0	1	1	0	0
1	1	0	0	1	0	1	0
1	0	1	0	0	0	0	1
1	0	0	0	0	0	1	0
0	1	1	1	1	1	1	1
0	1	0	1	1	0	0	1
0	0	1	1	1	1	1	1
0	0	0	1	1	0	0	1

Exemple

Si l'on ne garde que les lignes donnant la valeur 1 on obtient une table réduite

p	q	r	$(((\neg p) \vee q) \wedge r) \iff (p \otimes r)$	conjonction correspondante
1	0	1	1	$p \wedge (\neg q) \wedge r$
0	1	1	1	$(\neg p) \wedge q \wedge r$
0	1	0	1	$(\neg p) \wedge q \wedge (\neg r)$
0	0	1	1	$(\neg p) \wedge (\neg q) \wedge r$
0	0	0	1	$(\neg p) \wedge (\neg q) \wedge (\neg r)$

d'où

$$\begin{aligned} F \quad \equiv \quad & \Big(p \wedge (\neg q) \wedge r\Big) \vee \Big((\neg p) \wedge q \wedge r\Big) \vee \Big((\neg p) \wedge q \wedge (\neg r)\Big) \\ & \vee \Big((\neg p) \wedge (\neg q) \wedge r\Big) \vee \Big((\neg p) \wedge (\neg q) \wedge (\neg r)\Big) \end{aligned}$$

Pour trouver maintenant une conjonction de disjonctions équivalente à une formule F, on peut appliquer la méthode précédente à la formule $\neg F$ puis appliquer les lois de Morgan, c'est à dire remplacer les conjonctions par des disjonctions, les disjonctions par des conjonctions, les p_i par des $\neg p_i$ et les $\neg p_i$ par des p_i. Sur la table de vérité, cela revient à ne garder que les lignes pour lesquelles la valeur de F vaut 0 ; pour chacune de ces lignes on construit la disjonction obtenue en prenant toutes les variables logiques, celles ayant pour valeur 0 étant prises directement, celles ayant pour valeur 1 étant niées. On prend alors la conjonction de toutes les disjonctions ainsi obtenues.

Exemple Repartons de $F = (((\neg p) \vee q) \wedge r) \Leftrightarrow (p \otimes r)$. On a précédemment construit la table de vérité

p	q	r	$\neg p$	$(\neg p) \vee q$	$((\neg p) \vee q) \wedge r$	$p \otimes r$	$(((\neg p) \vee q) \wedge r) \iff (p \otimes r)$
1	1	1	0	1	1	0	0
1	1	0	0	1	0	1	0
1	0	1	0	0	0	0	1
1	0	0	0	0	0	1	0
0	1	1	1	1	1	1	1
0	1	0	1	1	0	0	1
0	0	1	1	1	1	1	1
0	0	0	1	1	0	0	1

Exemple

Si l'on ne garde que les lignes donnant la valeur 0 on obtient une table réduite

p	q	r	$(((\neg p) \vee q) \wedge r) \iff (p \otimes r)$	disjonction correspondante
1	1	1	0	$(\neg p) \vee (\neg q) \vee (\neg r)$
1	1	0	0	$(\neg p) \vee (\neg q) \vee r$
1	0	0	0	$(\neg p) \vee q \vee r$

si bien que $F \equiv \Big((\neg p) \vee (\neg q) \vee (\neg r)\Big) \wedge \Big((\neg p) \vee (\neg q) \vee r\Big) \wedge \Big((\neg p) \vee q \vee r\Big)$

Définition 5.13.2 *La formule logique $\bigwedge_{\sigma \in \Sigma} D_\sigma$ ainsi obtenue s'appelle la forme normale conjonctive de la formule logique F.*

Remarque Toutes les disjonctions intervenant dans la forme normale conjonctive de F ont la particularité de contenir toutes les variables propositionnelles de F une et une seule fois, soit par elle mêmes, soit par leur négation.

Bien entendu les deux méthodes précédentes peuvent s'appliquer à toute fonction booléenne, ce qui montre que toute fonction booléenne est la fonction associée à une conjonction de disjonctions et aussi la fonction associée à une disjonction de conjonctions.

Une discussion s'impose alors sur l'utilité de ce que nous venons de faire. La transformation d'une formule logique en une conjonction de disjonctions (ou une disjonction de conjonctions) équivalente est une opération fondamentale dans la programmation logique (dont un exemple est le langage Prolog qui sert en particulier dans de nombreux domaines de l'intelligence artificielle) ; elle est à la base des algorithmes de raisonnement. Deux méthodes sont possibles pour obtenir une telle formule équivalente. La première est celle du paragraphe précédent ; elle utilise certaines règles mécaniques (associativité, commutativité, distributivité, lois de Morgan, tiers exclu) ; elle est d'un emploi pénible à la main, mais relativement facile à implémenter avec un outil de calcul formel rompu à ce genre de manipulations. La deuxième, celle des tables de vérité, est relativement facile à effectuer à la main dans le cas d'un petit nombre de variables ; par contre on sait qu'elle est coûteuse en temps de calcul puisque l'établissement d'une table de vérité pour une formule logique à n variables demande un temps proportionnel à 2^n.

5.14 Circuits logiques élémentaires

5.15 Notion de circuit logique

Il s'agit ici, sans rentrer dans les détails techniques, de décrire l'implémentation physique possible de l'évaluation d'une formule logique.

Donnons nous une formule logique F dépendant des variables propositionnelles $p_1, \ldots, p_n$. Nous allons chercher à construire un circuit électronique $\mathcal{C}$ disposant de n entrées $P_1, \ldots, P_n$ et d'une sortie Q. Chacune de ces entrées P_i peut être portée soit à la tension 0 Volts, soit à la tension +5 Volts (les chiffres réels n'ont pas d'importance) suivant que la valeur attribuée à la variable p_i vaut 0 ou 1. A chaque application $\sigma = \{p_1, \ldots, p_n\} \to \{0, 1\}$ correspond donc

une distribution de tensions sur les entrées $P_1, \ldots, P_n$ du circuit. On cherche à faire en sorte que la tension sur la sortie Q soit alors 0 Volts si $\mathrm{Val}(F, \sigma) = 0$ et de 5 Volts si $\mathrm{Val}(F, \sigma) = 1$.

Il est clair que si un circuit logique effectue l'évaluation d'une formule logique F, il effectue aussi l'évaluation de toute formule logique équivalente à F.

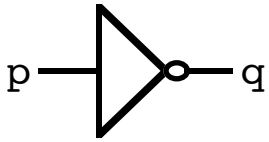
5.16 Portes logiques et circuits élémentaires

Nous supposerons par la suite que nous disposons d'un certain nombre de circuits élémentaires que nous pourrons assembler en reliant leurs entrées soit à une des entrée P_i soit à la sortie d'un autre circuit élémentaire. Ces circuits élémentaires sont appelés des portes logiques.

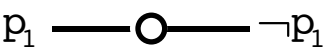
Un assemblage de portes logiques réalisant l'évaluation d'une formule logique particulière pourra ensuite être considérée comme un nouveau circuit élémentaire susceptible de réutilisation en tant que porte logique.

Nous allons décrire un système d'assemblage de portes logiques utilisant deux portes élémentaires [2] : une porte "non" et une porte "ou" ; ces deux portes logiques sont facilement réalisables par un assemblage d'un ou deux transistors.

La porte "non" est une porte logique à une entrée P et une sortie Q : Q est à la tension 5V si et seulement si P est à le tension 0V. La porte "non" réalise donc l'évaluation de la formule logique $q = \neg\, p$. Nous la symboliserons sous la forme

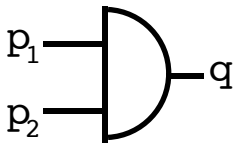


ou même sous forme simplifiée par un simple cercle

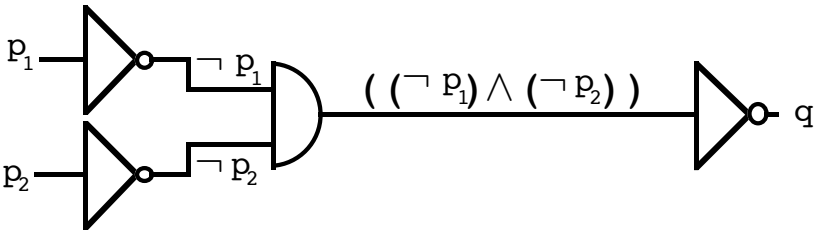


ce simple cercle pouvant être éventuellement accolé à un autre symbole en entrée ou en sortie.

La porte "et" est une porte logique à deux entrées P_1 et P_2 et une sortie Q. La sortie Q est à le tension 5V si et seulement si chacune des deux entrées est à la tension 5V. La porte "et" réalise donc l'évaluation de la formule logique $q = p_1 \wedge p_2$. Nous la symboliserons sous la forme

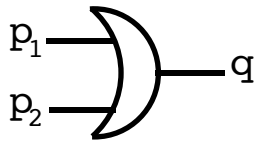


On peut alors réaliser un circuit logique qui effectue l'évaluation de la formule logique $q = p_1 \vee p_2$. Il suffit d'utiliser la loi de Morgan qui nous dit que $p_1 \vee p_2 \equiv \neg\,((\neg\, p_1) \wedge (\neg\, p_2))$ ce qui nous donne un circuit logique

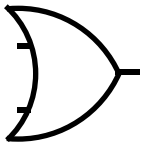


Etant donné l'utilité de ce circuit logique, nous en ferons une nouvelle porte logique, que nous appellerons porte "ou" et que nous symboliserons sous la forme

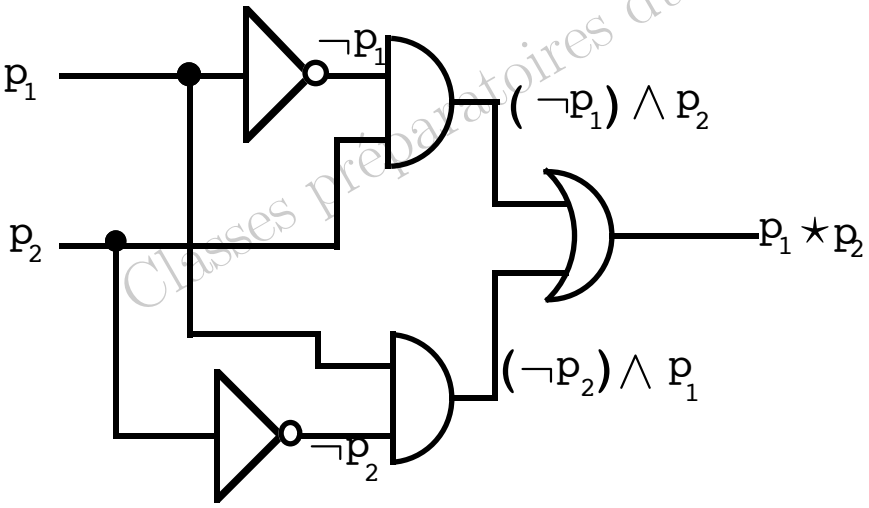
2. En fait on peut montrer qu'il est possible de construire tout circuit logique en utilisant une seule porte logique bien choisie



ou encore avec le symbole

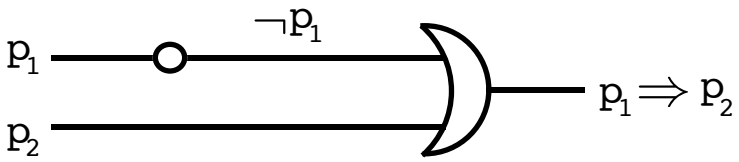


On peut alors assembler ces trois portes logiques pour évaluer les autres opérateurs logiques élémentaires. En ce qui concerne l'opérateur xor que nous avons noté $\otimes$, on peut utiliser $p_1 \otimes p_2 \equiv ((\neg\, p_1) \wedge p_2) \vee (p_1 \wedge (\neg\, p_2))$ d'où le circuit logique

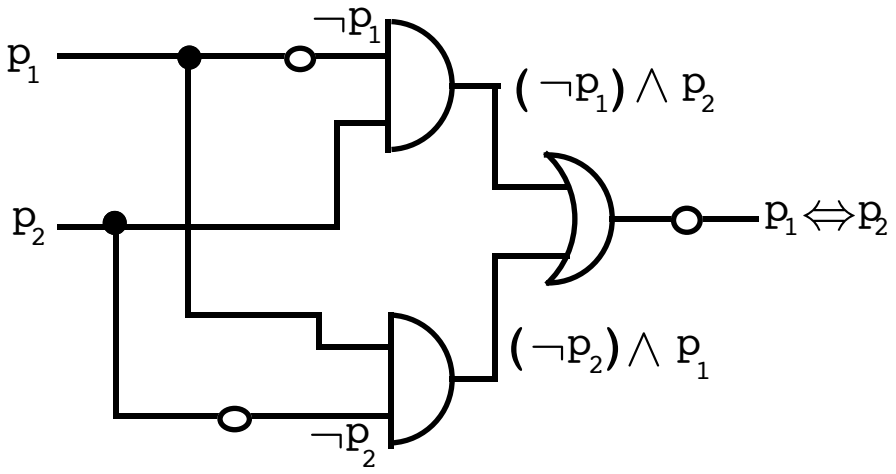


Remarque Par convention, dans un circuit logique, lorsque deux fils se croisent, ils sont considérés comme passant sur deux niveaux différents et donc être isolés l'un de l'autre. Lorsque l'on veut connecter deux fils, on doit expliciter cette connexion par un rond noir qui la symbolise. Dans la réalisation pratique de circuit logique, ces croisements de fils sur plusieurs niveaux sont difficilement réalisables et la conception de circuits intégrés comportant des centaines de milliers ou des millions de portes logiques interconnectées pose de redoutables problèmes de géométrie combinatoire dans lesquels la théorie des graphes joue un rôle essentiel.

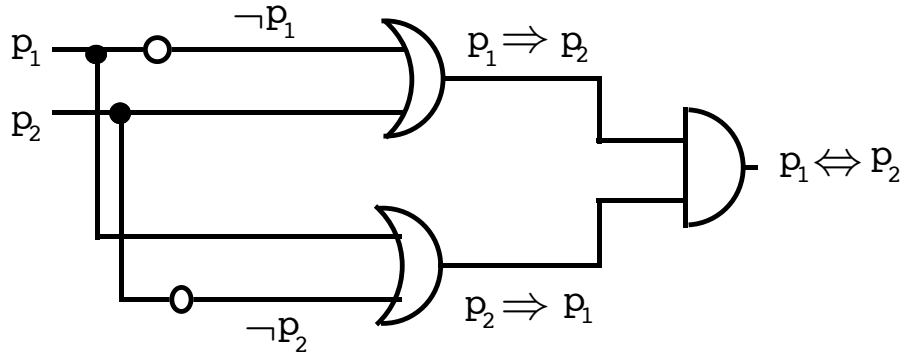
Pour ce qui est de l'opérateur d'implication, on peut utiliser $p_1 \Rightarrow p_2 \equiv p_2 \vee (\neg\, p_1)$ ce qui donne le circuit



Quant à l'opérateur d'équivalence, on peut remarquer que $p_1 \Leftrightarrow p_2 \equiv \neg\,(p_1 \otimes p_2)$ ce qui nous fournit le circuit



ou encore que $p_1 \Leftrightarrow p_2 \equiv (p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_1)$ ce qui nous donne le circuit

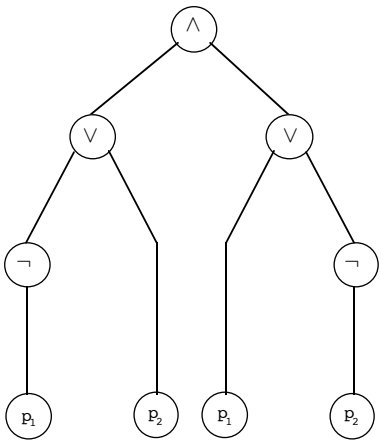


5.17 Circuits logiques et représentations arborescentes

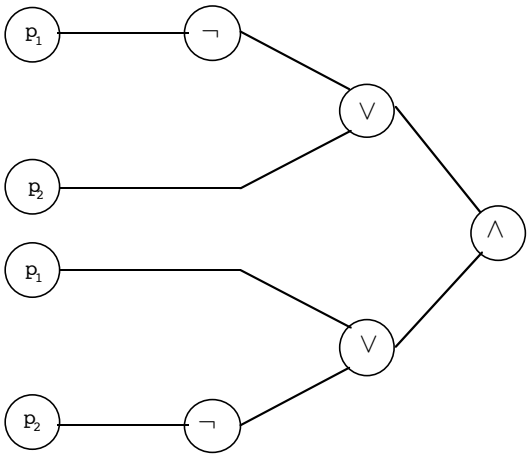
Reprenons le dernier circuit logique que nous avons construits, où nous avons écrit que

$$p_1 \Leftrightarrow p_2 \equiv (p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_1) \equiv (p_2 \vee (\neg\, p_1)) \wedge ((\neg\, p_2) \vee p_1)$$

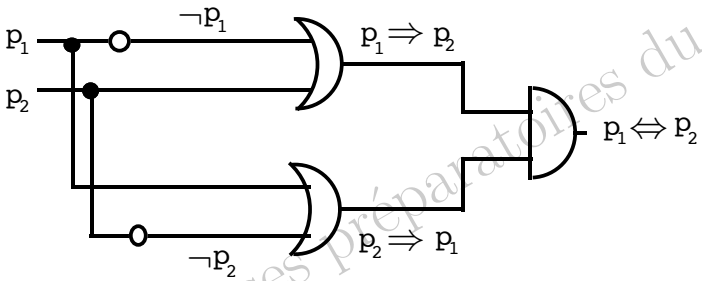
La représentation arborescente de cette dernière formule est



Effectuons une rotation de cette représentation arborescente. Nous obtenons alors un arbre "couché"



que nous pouvons comparer avec le circuit logique qui évalue la formule



La structure du circuit logique et la structure de l'arbre sont tout à fait identiques :
– chaque noeud étiqueté par $\wedge, \vee$ ou $\neg$ correspond à une porte logique "et", "ou" ou "non"
– les feuilles étiquetées par les variables logiques p_i correspondent aux entrées de même nom du circuit logique
– la racine de l'arbre correspond à la sortie du circuit

Ceci se généralise de manière évidente à toute formule logique ne faisant intervenir que les opérateurs $\wedge, \vee$ et $\neg$; à partir de la représentation arborescente de la formule logique, on peut construire un circuit logique évaluant la formule en remplaçant tous les noeuds de l'arbre par les portes logiques correspondantes, en reliant les feuilles de l'arbre aux entrées du circuit portant la même étiquette et en reliant la racine de l'arbre à la sortie du circuit.

Pour les formules logiques faisant intervenir les opérateurs $\otimes, \Rightarrow$ et $\Leftrightarrow$, on remplace les noeuds correspondant par les circuits logiques construits dans le paragraphe précédent, considérés comme de nouvelles portes logiques.

5.18 Additionneurs

Prenons deux nombres écrits en base 2 avec p chiffres, $m = x_p \dots x_0$ et $n = y_p \dots y_0$, chacun des x_i et y_i étant égal soit à 0 soit à 1. On peut alors effectuer l'addition de ces deux nombres suivant le même algorithme qu'en base 10. On additionne les chiffres de droite à gauche en base 2 en tenant compte des retenues éventuelles. Autrement dit on effectue l'algorithme suivant (où c_i désigne la retenue)
– affecter 0 à c_0
– pour i allant de 0 à p faire
– additionner x_i, y_i et c_i, en déduire le chiffre $z_i \in \{0, 1\}$ et une retenue $c_{i+1} \in \{0, 1\}$

L'écriture en base 2 de $m + n$ est alors $c_{p+1} z_p \dots z_0$.

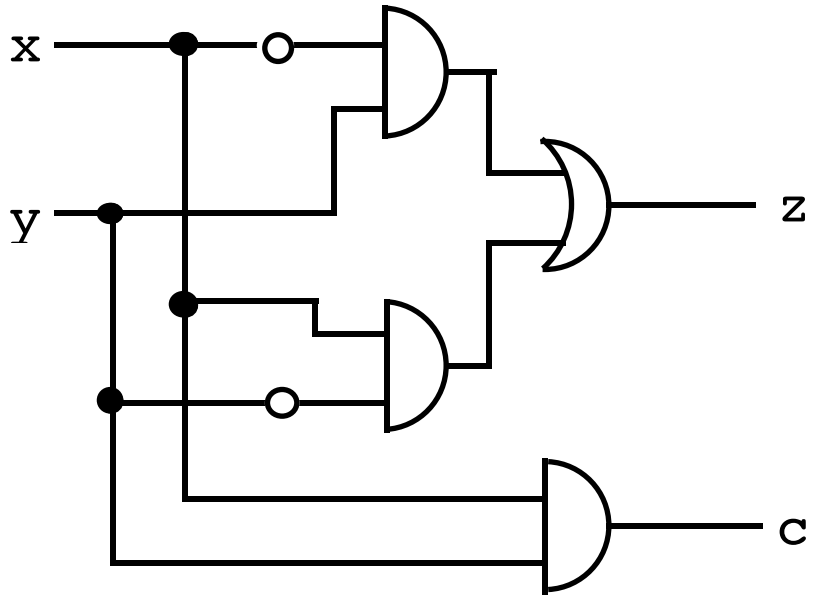
Remarque Dans la pratique des ordinateurs, $p + 1$ est fixe égal à 8, 16, 32 ou même 64 suivant les processeurs. En règle générale, le résultat retourné par une addition est non pas $c_{p+1} z_p \dots z_0$ mais simplement $z_p \dots z_0$. La dernière retenue c_{p+1} est donc négligée dans le résultat final. Elle sert cependant à positionner un *drapeau* qui signalera un débordement éventuel dans l'addition, autrement dit que le résultat ne tient pas vraiment sur $p + 1$ bits.

Circuit logique qui étant donné deux entrées x, y calcule z et une retenue c : demi-additionneur parce qu'il ne tient pas compte d'une retenue précédente qu'il faut ajouter ensuite au résultat obtenu : ceci nécessitera un deuxième

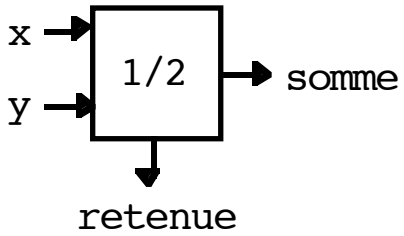
demi-additionneur pour confectionner un additionneur complet. Pour cela nous allons simplement calculer une table de vérité à deux entrées et deux sorties.

x	y	c	z
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

Nous constatons immédiatement que $z = f_{\otimes}(x, y)$ et que $c = f_{\wedge}(x, y)$. Ceci nous permet de construire notre circuit demi-additionneur.



Nous symboliserons par la suite un tel circuit demi-additionneur sous la forme

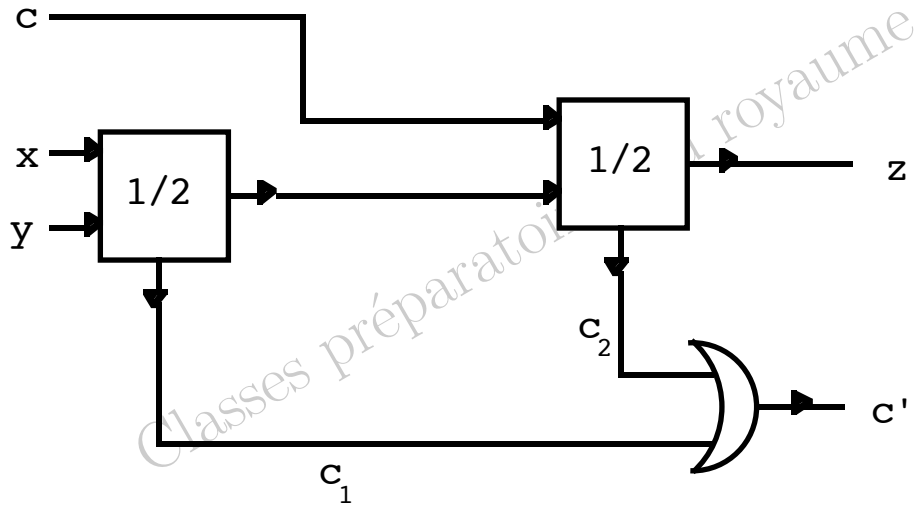


Construisons alors un circuit additionneur complet qui prend trois entrées x, y et c (la retenue précédente) et qui retourne un chiffre z et une nouvelle retenue c'.

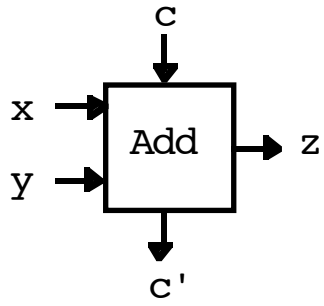
Pour cela on commence par additionner x et y obtenant un chiffre z_1 et une retenue c_1. Il faut ensuite que nous additionnions z_1 et c pour obtenir un chiffre qui n'est autre que z et une retenue c_2. Mais remarquons que si $c_1 = 1$, alors nécessairement $z_1 = 0$ (voir la table de vérité ci dessus) et donc nécessairement $c_2 = 0$, et alors $c' = c_1$. Si par contre $c_1 = 0$, alors $c' = c_2$. Autrement dit, on a toujours $c' = f_{\vee}(c_1, c_2)$. On peut aussi constater de visu le résultat sur la table de vérité à trois entrées

x	y	c	z_1	c_1	z	c_2	c'
1	1	1	0	1	1	0	1
1	1	0	0	1	0	0	1
1	0	1	1	0	0	1	1
1	0	0	1	0	1	0	0
0	1	1	1	0	0	1	1
0	1	0	1	0	1	0	0
0	0	1	0	0	1	0	0
0	0	0	0	0	0	0	0

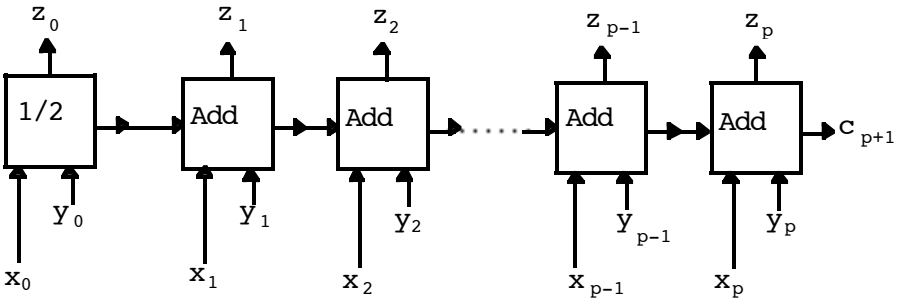
Ceci nous conduit à un circuit additionneur, complet de la forme



Nous symboliserons un tel circuit additionneur sous la forme



Par assemblage, on peut alors construire un circuit additionneur p bits de la manière suivante



Remarque En fait la réalisation d'un tel circuit pose de délicats problèmes de temporisation. En effet l'établissement des tensions dans les sorties n'est pas instantanée et il est essentiel qu'un additionneur ne prenne en compte la retenue provenant de l'additionneur de gauche qu'après qu'elle ait été clairement établie. Le cadencement est donc une étape importante de la conception des circuits intégrés, d'où le rôle des quartz qui jouent le rôle de chefs d'orchestre de tous les circuits logiques.