Chapitre 1

Structures de données

Jean-Jacques Lévy, INRIA

Mots-clé: structure de donnée, enregistrement, tableau, liste, arbre, file, pile, graphe.

Résumé: dans ce chapitre, nous introduisons les structures de données élémentaires qui représentent les ensembles éventuellement ordonnés. Les exemples de programmes qui les manipulent sont exprimés ici dans un langage dont la syntaxe est proche de Java, de C ou de C++ avec une instruction nouvelle **foreach** pour itérer sur tous les éléments d'un ensemble ou d'une suite ordonnée. En outre, l'opérateur **new** construira des éléments de structures de données de manière intuitive, c'est-à-dire sans spécifier rigoureusement le fonctionnement de ce constructeur.

1.1 Produits cartésiens et enregistrements

Les valeurs manipulées par les programmes sont dans leur grande majorité des scalaires, c'est-à-dire des nombres entiers, des nombres réels, des booléens ou des caractères. Mais, on veut aussi souvent manipuler des paires de scalaires, comme les coordonnées (x, y) d'un point dans le plan, ou des triplets (x, y, z) dans l'espace à trois dimensions. La majorité des langages de programmation se servent de la notion d'enregistrement pour représenter des n-uplets de valeurs. Ainsi

```
struct Point { int x; int y; }
```

permet de définir des points `p = new Point(3,0)` ou `q = new Point(1,2)`. Pour obtenir l'abscisse de `p`, on utilise la notation qualifiée `p.x` ; de même l'ordonnée de `q` est `q.y`.

Les champs d'un enregistrement peuvent être de types différents. Ainsi

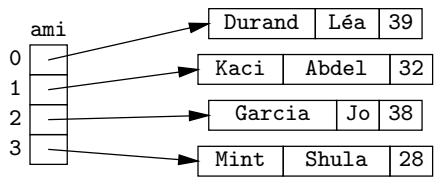
```
struct Ami {
  String nom; String prenom; int age;
}
```

est un enregistrement à trois champs donnant les nom, prénom et âge d'un ami. Le nom de l'ami `a` est `a.nom` ; de même pour le prénom `a.nom` et l'âge `a.age`. Certains langages de programmation autorisent la manipulation directe de n-uplets, que l'on peut affecter à des variables quelconques ou qui peuvent être retournés comme résultats de fonction. Nous ne traiterons pas de cet aspect dans ce chapitre.

1.2 Tableaux

Le tableau unidimensionnel est la structure de donnée de base en informatique. Une collection d'éléments $\langle a_0, a_1, \ldots a_{n-1}\rangle$ $(n \geq 0)$ est représentée par un tableau linéaire $a[\,]$ de n éléments rangés consécutivement en mémoire ; on accède directement au i-ième élément $a[i-1]$ en un coût constant. Par exemple, un ensemble de quatre amis est décrit par le tableau :

```
Ami[ ] ami = new Ami[4];
ami[0] = new Ami("Durand", "Léa", 39);
ami[1] = new Ami("Kaci", "Abdel", 32);
ami[2] = new Ami("Garcia", "Jo", 38);
ami[3] = new Ami("Mint", "Shula", 28);
```



Avec ce tableau `ami`, on vient de construire un minuscule répertoire, encore appelé une table en

informatique. Pour retrouver l'âge de Jo Garcia, il suffit de parcourir les champs nom du tableau `ami`, de vérifier son prénom, et de trouver l'information voulue dans le champ âge au même indice. Ainsi on compare successivement `ami[0].nom` à Garcia, puis `ami[1].nom`, puis `ami[2].nom`. On peut également faire des recherches croisées, et trouver les noms et prénoms des personnes d'âge compris entre 35 et 40 ans. La structure de tableau sert souvent à construire des tables, des index, ou de petites bases de données comme avec un logiciel de tableur.

Il y a de multiples manières de faire les recherches en table, toutes plus savantes les unes que les autres. La plus connue, la recherche dichotomique, consiste à ranger un des tableaux par ordre alphabétique, par exemple le tableau des noms, et à faire une recherche sur la table en regardant si le nom recherché est avant ou après l'indice du milieu de la table ; puis on itère sur le début ou la fin de la table, jusqu'à trouver le nom recherché. Le lecteur intéressé est invité à se reporter au chapitre XX ou au livre de **?** pour une étude extensive de la recherche en table.

On aurait aussi pu se passer des enregistrements et représenter notre répertoire par trois tableaux de même taille mis en parallèle, comme suit :

```
String[ ] nom = new String[4], prenom =
   new String[4], age = new String[4];
nom[0] = "Dupont"; prenom[0] = "Léa";
nom[1] = "Kaci"; prenom[1] = "Abdel";
nom[2] = "Garcia"; prenom[2] = "Jo";
nom[3] = "Mint"; prenom[3] = "Shula";
age[0] = 39; age[1] = 32;
age[2] = 38; age[3] = 28;
```

Mais cette représentation n'est pas modulaire, puisque les données relatives à un même ami sont dispersées sur trois tableaux. Le choix entre la première solution (la tableau d'enregistrements) et cette dernière solution (trois tableaux en parallèle) est très représentative des choix à faire dans la représentation informatique des données dans un programme.

Les tableaux à deux dimensions sont souvent représentés par des tableaux de tableaux. Ainsi une matrice $(m_{ij})_{i,j}$ devient un tableau $m[\,]$ dont chaque ligne est un tableau unidimensionnel. On écrit $m[i][j]$ pour accéder à l'élément m_{ij}. De même, les tableaux à N dimensions sont des tableaux unidimensionnels dont chaque élément est un tableau à $N-1$ dimensions.

La structure de tableau unidimensionnel est présente dans tous les langages de programmation, car elle colle à la structure de la mémoire des ordinateurs ; elle est donc très facile à implanter efficacement. Mais cette structure en a aussi le défaut : il est plus difficile d'implanter des tableaux à plusieurs dimensions, et impossible d'avoir des tableaux dont la taille varie après leur création.

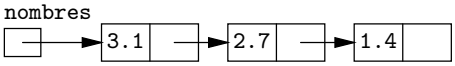
Dans les langages modernes, la taille d'un tableau est fixée à sa création ; et on ne peut plus la changer par la suite. La taille d'un tableau est souvent accessible par une primitive spéciale, par exemple `ami.length` donne la taille (4) du tableau des amis. Les tableaux sont homogènes alors que les enregistrements ne le sont pas. Tous les éléments d'un tableau ont le même type ; on considère des tableaux d'entiers, ou de réels, ou de triplets, etc. ; alors que les divers champs d'un enregistrement n'ont pas forcément le même type.

1.3 Listes

Quand le nombre d'éléments d'un ensemble varie dans le temps, on préfère utiliser la structure de liste. Une liste est une suite ordonnée d'éléments $\langle a_0, a_1, \ldots a_{n-1}\rangle$. Le premier élément a_0 est la *tête* de la liste ; la liste $\langle a_1, a_2, \ldots a_{n-1}\rangle$ des éléments suivants est la *queue* de la liste. Quand $n = 0$, la liste est *vide* ; elle correspond à l'ensemble vide.

On accède séquentiellement aux éléments d'une liste ; pour lire la valeur de a_k, on doit accéder à a_0, puis à a_1, ... puis à a_{k-1}, et enfin à a_k. Pour chaque élément, on stocke en mémoire une cellule de liste, c'est-à-dire une paire $\langle$valeur, suivant$\rangle$, qui permet d'accéder au champ valeur et de trouver l'emplacement mémoire de la cellule suivante dans la liste. Le premier champ correspond à la tête de liste ; le deuxième à sa queue. Ainsi on n'a pas à implanter consécutivement en mémoire les cellules contenant les éléments d'une liste. Comme pour les tableaux, les listes sont homogènes. Une liste de nombres réels se déclare comme suit :

```
struct Liste {float valeur; Liste suivant;}
...
Liste nombres = new Liste(3.1,
  new Liste(2.7, new Liste(1.4,
    null)));
```



où **null** représente la liste vide. On obtient le réel 3.1 par `nombres.valeur` ; 2.7 par `nombres.suivant.valeur` ; et 1.4 comme `nombres.suivant.suivant.valeur`. On peut

également définir des listes d'amis, dont le champ `valeur` a alors un type `Ami`.

```
struct Liste {Ami valeur; Liste suivant;}
...
Liste amis =
 new Liste(new Ami("Durand", "Léa", 39),
 new Liste(new Ami("Kaci", "Abdel", 32),
 new Liste(new Ami("Garcia", "Jo", 38),
 new Liste(new Ami("Mint", "Shula", 28),
 null))));
```

De nouveau, le prénom de l'ami `Garcia` est obtenu par `amis.suivant.suivant.valeur.prenom`.

Comme annoncé, l'intérêt des listes est de représenter des ensembles de taille dynamique. Pour ajouter un élément à une liste, il suffit de le rajouter en tête de liste. Ainsi, on ajoute 1.732 en créant une nouvelle cellule dont la valeur est 1.732 et le champ suivant pointe vers l'ancienne liste :

```
nombres = new Liste(1.732, nombres);
```

La nouvelle liste de nombres contient donc la suite $\langle 1.7, 3.1, 2.7, 1.4\rangle$. Mais pour garder la liste de nombres en ordre décroissant, on aurait dû ajouter 1.7 en troisième position en faisant :

```
nombres.suivant.suivant = new
  Liste(1.7, nombres.suivant.suivant);
```

Cette caractéristique de pouvoir insérer un élément de liste en n'importe quelle position a longtemps été un des gros avantages des listes, puisque la modification d'un ensemble ne coûtait que la taille de la cellule insérée en occupation mémoire.

Avec la taille mémoire des ordinateurs actuels, cet argument est moins vrai. Souvent on recopie le début de la liste jusqu'à la place de l'insertion, sans aboutir à un coût exorbitant, l'avantage des listes par rapport au tableau restant qu'on n'a pas à recopier toute la structure pour l'insertion d'un nouvel élément. Par ailleurs, ce style de programmation, parfois dénommé fonctionnel, considère les listes comme des données immuables ; il favorise le traitement concurrent des structures de données où plusieurs fils de calcul manipulent une même donnée ; il rend aussi la programmation plus sûre et donc plus facile à mettre au point.

Pour bien comprendre la programmation fonctionnelle sur les listes, il est commode de voir le type des listes comme décrit par l'équation suivante :

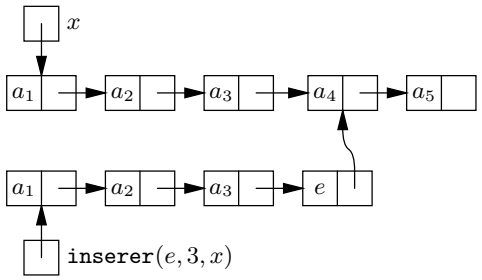
$$\text{Liste} = \text{Liste\_vide} + \text{Élément} \times \text{Liste}$$

puisqu'une liste est soit la liste vide, soit une cellule contenant la valeur d'un élément et un pointeur vers la deuxième cellule de la liste (donc assimilable elle-même à une liste). Cette équation peut se justifier mathématiquement ; ici nous n'en garderons que le coté intuitif. L'important est de comprendre qu'un principe d'induction structurelle existe sur les listes : toute propriété vraie sur une liste doit être vraie sur la liste vide, ou sur une paire $\langle$Élément, Liste$\rangle$. Toute fonction manipulant une liste peut être définie par récurrence à partir de la liste **null** et de la paire $\langle$valeur, suivant$\rangle$. Ainsi la longueur d'une liste se définit récursivement par

```
int longueur(Liste x) {
  if (x == null) return 0;
  else return 1 + longueur(x.suivant);
}
```

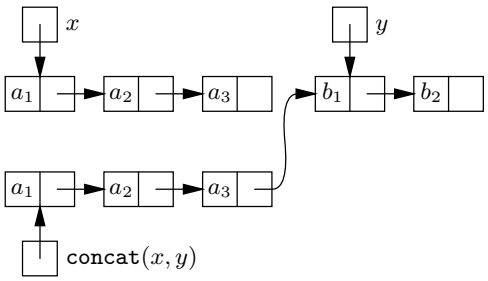
De même l'insertion de e en n-ième position se définit par

```
Liste inserer(float e, int n, Liste x) {
  if (x == null) return null;
  else if (n == 0) return new Liste(e, x);
  else return new Liste(x.valeur,
    inserer(e, n-1, x.suivant));
}
```



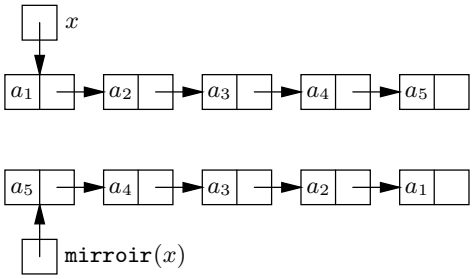
La concaténation de deux listes x et y s'écrit aussi

```
Liste concat(Liste x, Liste y) {
  if (x == null) return y;
  else return new Liste(x.valeur,
    concat(x.suivant, y));
}
```



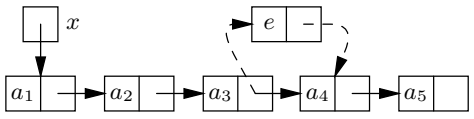
ou encore l'image miroir

```
Liste miroir(Liste x) {
  if (x == null) return null;
  else return concat(x.suivant,
    new Liste(x.valeur, null));
}
```

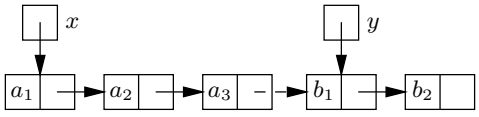


En utilisant des structures de données modifiables et, donc, un style de programmation plus dangereuse, on réécrit les fonctions précédentes comme suit :

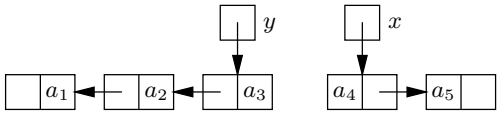
```
Liste insererM(float e, int n, Liste x) {
  if (x == null) return null;
  else if (n == 0) return new Liste(e, x);
  else { x.suivant =
      insererM(e, n-1, x.suivant));
    return x;
} }
```




```
Liste concatM(Liste x, Liste y) {
  if (x == null) return y;
  else { x.suivant =
    concatM(x.suivant, y);
    return x;
} }
```



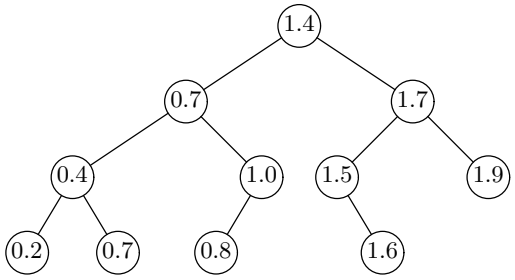

```
Liste miroirM(Liste x) {
  return miroir1M(null, x);
}

Liste miroir1M(Liste y, Liste x) {
  if (x == null) return y;
  else {
    Liste z = x.suivant;
    x.suivant = y;
    return miroir1M(x, z);
} }
```



La manipulation des listes demande donc un choix entre les représentations immuables ou modifiables. Dans le premier cas, on suit la définition inductive des listes ; les fonctions sur les listes sont donc des fonctions récursives suivant la définition récursive du type des listes.

1.4 Arbres

Les arbres de l'informatique sont représentés à l'envers par rapport à la botanique. La racine est en haut, les feuilles sont en bas.



Pour trouver 0.8 en partant de la racine de l'arbre, on part dans le sous-arbre de gauche dont la racine vaut 0.7, puis dans le sous-arbre de droite dont la racine vaut 1.0, puis dans son sous-arbre de gauche, qui n'ayant pas de sous-arbre est considéré comme une feuille.

Il y a de multiples raisons de considérer des structures de données arborescentes. D'abord, la recherche en table sur un ensemble de taille dynamique s'organise naturellement avec un arbre. Comme déjà vu, un tableau trié en ordre croissant (un dictionnaire de français par exemple) permet d'avoir une recherche en table plus rapide (en $\log n$ opérations) sur un ensemble de n éléments. Mais si le nombre n varie dans le temps, les listes sont inefficaces pour représenter cette table. On considère plutôt un arbre dont la racine est l'élément du milieu de la table et le sous-arbre de gauche décrit la table des éléments plus petits que la valeur du milieu et le sous-arbre de droite décrit la table des éléments plus grands que la valeur du milieu.

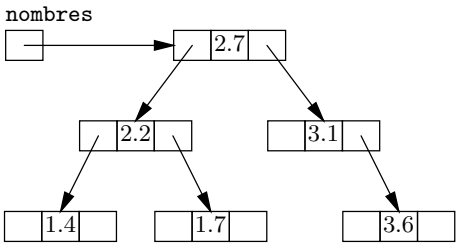
On peut montrer qu'alors la recherche en table prend un temps de l'ordre de $\log n$ opérations si l'arbre est bien équilibré. Nous laissons à nouveau les finesses de la recherche en table aux chapitres

concernés (ch. XX). On peut aussi lire le livre de Sedgewick.

Les arbres servent aussi à exprimer des ensembles de données ordonnées partiellement, comme par exemple la structure hiérarchique d'une entreprise, ou la généalogie d'une famille. Les arbres sont aussi utiles pour représenter la syntaxe des termes en calcul symbolique, dans les systèmes de démonstration automatique, dans les assistants de preuves mécaniques, dans les passes internes d'un compilateur. Les arbres décrivent aussi le contrôle récursif d'un programme ; ils sont souvent la représentation d'un ensemble de données sur lequel s'appuie un algorithme utilisant le principe « Diviser pour régner ».

Le traitement des arbres diffère peu de celui des listes. Nous ne considérons pour l'instant que les arbres binaires, où tout nœud de l'arbre a au plus deux sous-arbres. La représentation des arbres binaires suit celle des listes. À chaque nœud de l'arbre, on fait correspondre une cellule contenant un triplet $\langle$valeur, gauche, droite$\rangle$, où le premier champ donne la valeur du nœud, et les deuxième et troisième champs sont les sous-arbres de gauche et de droite. Les feuilles sont donc des nœuds sans sous-arbre gauche, ni sous-arbre droit. Comme pour les listes et tableaux, les arbres sont homogènes. Un arbre de réels se déclare comme suit :

```
struct Arbre { float valeur;
  Arbre gauche; Arbre droite;
}
...
Arbre nombres = new Arbre(2.7
  new Arbre(2.2
    new Arbre(1.4, null, null),
    new Arbre(1.7, null, null)),
  new Arbre(3.1,
    null,
    new Arbre(3.6, null, null)));
```



où **null** représente l'arbre vide. La constante 2.7 s'obtient par `nombres.valeur` ; la constante 3.1 comme `nombres.droite.valeur` ; et 1.7 par `nombres.gauche.droite.valeur`. On peut à nouveau considérer un arbre d'amis, dont le champ valeur a alors le type `Ami`, et les ranger par ordre alphabétique sur leur nom. Nous ne le faisons pas explicitement, mais il faudrait utiliser une représentation très similaire au cas des listes d'amis.

Comme pour les listes, le type des arbres peut être vu abstraitement comme décrit par l'équation suivante :

$$\text{Arbre} = \text{Arbre\_vide} + \text{Arbre} \times \text{Élément} \times \text{Arbre}$$

puisqu'un arbre est soit l'arbre vide, soit une cellule contenant la valeur d'un élément et deux pointeurs vers le sous-arbre de gauche et vers le sous-arbre de droite. Les programmes sur les arbres peuvent donc suivre un principe d'induction structurelle, associant leur contrôle récursif à la définition récursive du type des arbres. Ainsi la hauteur d'un arbre se définit récursivement par :

```
int hauteur(Arbre a) {
  if (a == null) return 0;
  else return 1 + max(
    hauteur(a.gauche), hauteur(a.droite));
}
```

C'est donc la longueur du plus long chemin menant de la racine aux feuilles. De même l'insertion de e dans un ensemble ordonné représenté par un arbre a s'écrit (en style fonctionnel) :

```
Arbre inserer(float e, Arbre a) {
  if (a == null)
    return new Arbre(e, null, null);
  else if (e < a.valeur)
    return inserer(e, a.gauche);
  else
    return inserer(e, a.droite);
}
```

Les arbres équilibrés sont des arbres où la différence entre les hauteurs des sous-arbres de gauche et de droite est bornée par un petit nombre pour tout nœud. L'application itérée de la fonction précédente peut fortement déséquilibrer un arbre. Si on veut garder un temps d'accès en $\log n$ pour les éléments d'un ensemble ordonné de n éléments, on doit donc régulièrement rééquilibrer l'arbre. Des méthodes sophistiquées le font très efficacement : arbres 2-3-4, arbres bicolores, etc.

La fonction `inserer` parcourt les arbres dans l'ordre préfixe. On distingue classiquement trois types de parcours sur les arbres pour visiter tous ses nœuds. Dans l'ordre préfixe, on considère d'abord la racine, puis le sous-arbre de gauche, et enfin le sous-arbre de droite. Dans l'ordre infixe, on rend visite d'abord au sous-arbre de gauche, puis à la racine, et enfin au sous-arbre de droite. Dans l'ordre postfixe, on considère d'abord le sous-arbre de gauche, puis le sous-arbre de droite et enfin la racine. Sur le premier exemple de cette

section, les suites des valeurs des nœuds correspondant aux parcours préfixe, infixe et postfixe sont respectivement :

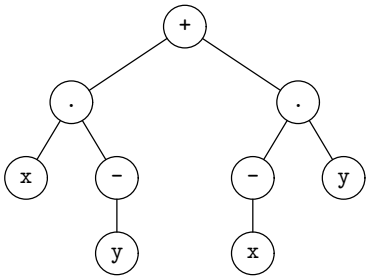
$\langle 1.4, 0.7, 0.4, 0.2, 0.7, 1.0, 0.8, 1.7, 1.5, 1.6, 1.9\rangle$
$\langle 0.2, 0.4, 0.7, 0.7, 0.8, 1.0, 1.4, 1.5, 1.6, 1.7, 1.9\rangle$
$\langle 0.2, 0.7, 0.4, 0.8, 1.0, 0.7, 1.6, 1.5, 1.9, 1.7, 1.4\rangle$

Les arbres de syntaxe abstraite sont à la base de la représentation des termes ou des formules dans le calcul symbolique. Considérons un petit exemple de formules propositionnelles comme $x\cdot\bar{y}+\bar{x}\cdot y$ où x et y ont des valeurs booléennes, où $+$ et $\cdot$ désignent le OU et le ET booléen, et où $\bar{x}$ désigne le complément de x. Nous voulons évaluer de telles propositions quand les valeurs de x et y sont fixées. Nous prenons 0 et 1 comme valeurs booléennes. Et donc $0 = 0+0$, $1 = 1+x = x+1$, $0 = 0\cdot x = x\cdot 0$, $1 = 1\cdot 1$, $\bar{x} = 1-x$. Nous supposons qu'il n'y a que deux variables booléennes x et y, dont les valeurs sont données par un enregistrement :

```
struct Environnement { int x; int y; }
```

Ainsi `e.x` et `e.y` seront les valeurs booléennes de x et y dans l'environnement e.

L'arbre de syntaxe abstraite d'une proposition booléenne est un arbre binaire dont les valeurs des nœuds sont les caractères `'+'`, `'.'`, `'-'`, `'x'`, `'y'`, `'0'`, `'1'` selon que l'opérateur associé au nœud est l'union, l'intersection, le complément, la valeur de la variable x, la valeur de la variable y, la constante 0 ou la constante 1. Ainsi l'arbre de syntaxe abstraite associée à la formule $x\cdot\bar{y}+\bar{x}\cdot y$ est :



L'évaluation d'une formule booléenne pour un certain environnement de ses variables s'écrit donc :

```
int evaluer(Arbre a, Environnement e) {
  if (a.valeur == '+')
    return min(1, min(
     evaluer(a.gauche, e),
     evaluer(a.droite, e)));
  else if (a.valeur == '.')
    return evaluer(a.gauche, e)
        * evaluer(a.droite, e);
  else if (a.valeur == '-')
    return 1 - evaluer(a.gauche, e);
  else if (a.valeur == '0') return 0;
  else if (a.valeur == '1') return 1;
  else if (a.valeur == 'x') return e.x;
  else if (a.valeur == 'y') return e.y;
}
```

où la fonction précédente est définie pour tout arbre non vide.

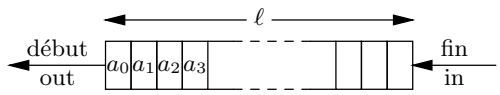
En notation polonaise postfixe, l'expression $x\cdot\bar{y}+\bar{x}\cdot y$ s'écrit `xy-.x-y.+`, où il n'y a plus de parenthèses et où tout opérateur figure sans ambiguïté après ses opérandes. (Ce fut pendant longtemps la seule notation acceptée pour calculer des expressions arithmétiques sur des calculettes). On génère cette représentation en faisant un parcours postfixe sur l'arbre de syntaxe abstraite de toute expression booléenne :

```
void postfixe(Arbre a) {
  if (a.valeur == '+') {
    postfixe(a.gauche); postfixe(a.droite);
    print('+');
  } else if (a.valeur == '.') {
    postfixe(a.gauche); postfixe(a.droite);
    print('.');
  } else if (a.valeur == '-') {
    postfixe(a.gauche); print('-');
  } else print(a.valeur);
}
```

Enfin, au-delà des arbres binaires, les arbres n-aires ont les nœuds qui peuvent avoir jusqu'à n sous-arbres. La représentation et les algorithmes sur ces arbres sont très similaires à celle et ceux développés pour les arbres binaires.

1.5 Files d'attente

Les files d'attente interviennent dès qu'il existe des événements concurrents. Très souvent les requêtes asynchrones à un serveur sont stockées dans une file, où le premier arrivé est le premier servi (FIFO, *first in, first out*). Par exemple, dans les circuits électroniques ou dans la transmission de données, beaucoup de fonctions doivent temporiser les requêtes avant de les servir. Ces circuits ou routeurs contiennent des FIFO. Également, certains parcours de structures de données utilisent des files d'attente.



Une file f est une suite ordonnée d'éléments $\langle a_0, a_1, \ldots a_{\ell-1}\rangle$ ($\ell \geq 0$), modifiable par deux

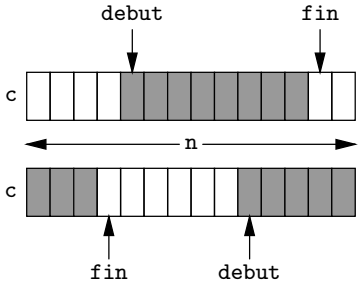
opérations : `ajouter`(x, f) qui ajoute x à la fin de la file f pour donner la file $\langle a_0, a_1, \ldots a_{\ell-1}, x\rangle$; `supprimer`(f) qui retourne l'élément a_0 en tête de la file f et le retire de la file qui devient $\langle a_1, \ldots a_{\ell-1}\rangle$

Traditionnellement, il y a deux représentations possibles pour une file d'attente. La première est statique et utilise un tampon circulaire ; la deuxième est dynamique et utilise une liste. Un tampon circulaire est un tableau c avec deux indices `debut` et `fin`. La taille n du tableau représente la taille maximale de la file ($\ell \leq n$). La file correspond à la partie du tableau dont l'indice est compris entre `debut` (inclus) et `fin` (exclu). Pour récupérer la place rendue par plusieurs suppressions dans c, on considère c comme un tableau circulaire, le suivant dans la file de $c[n-1]$ étant $c[0]$. Au bout de quelques ajouts et suppressions, on peut donc avoir `fin` < `debut`. Ceci donne une ambiguïté quand `fin` = `debut`, puisqu'on ne sait si la file est vide, ou pleine. On ajoute deux indicateurs `vide` et `pleine`. La file est donc représentée par l'enregistrement :

```
struct FIFO {
  float[ ] c;
  int debut, fin; boolean vide, pleine;
}
```

Le programme des deux opérations sur les files devient :

```
void ajouter(float x, FIFO f) {
  int n = f.c.length;
  if (f.pleine)
    error("File Pleine.");
  f.c[f.fin] = x;
  f.fin = (f.fin + 1) % n;
  f.vide = false;
  f.pleine = f.fin == f.debut;
}

float supprimer(FIFO f) {
  int n = f.c.length;
  if (f.vide)
    error("File Vide.");
  int res = f.c[debut];
  f.debut = (f.debut + 1) % n;
  f.vide = f.fin == f.debut;
  f.pleine = false;
  return res;
}
```



La deuxième représentation des files est dynamique, puisque ne faisant pas d'hypothèse sur la taille maximale de la file. La file est représentée par la liste de ses éléments $\langle a_0, a_1, \ldots a_{\ell-1}\rangle$, ainsi que par deux champs `debut` et `fin` pointant respectivement sur le premier et le dernier élément de la file. Ainsi on rajoute un élément derrière le dernier élément en modifiant le champ suivant de `fin` et en refaisant pointer `fin` vers le nouveau dernier élément ; et on supprime le premier de la file en donnant comme nouvelle valeur de `debut` la valeur de son ancien champ suivant. Ces opérations sont donc simples, mais il faut traiter à part le cas de la file vide. On peut la représenter par la liste vide, et chacune des opérations doit tester si le champ `debut` ou `fin` a la valeur **null**. Nous sautons les programmes correspondants qui ne sont que de simples manipulations de listes. Il faut néanmoins remarquer que cette représentation des files est moins compacte, puisque pour chaque élément il faut aussi garder un pointeur vers le suivant.

1.6 Piles

La structure de pile est duale de la structure de file d'attente. Une pile p est une suite d'éléments $\langle a_0, a_1, \ldots a_{h-1}\rangle$ ($h \geq 0$) avec deux opérations `ajouter`(x, p) et `supprimer`(p) comme pour les files, mais la politique de suppression est LIFO (*last in, first out*). C'est donc le dernier ajouté qui sort le premier de la pile. Les piles sont moins utilisées que les files, car très souvent elles ont un substitut : la récursivité. En effet, les piles servent principalement à compiler les programmes récursifs ; ce sont donc des opérations utilisées par le code machine, mais moins souvent dans un langage de programmation moderne. Toutefois, il existe des programmes où on a toujours besoin de piles, par exemple un programme récursif, qui contient une autre récursivité cachée ; mais ces cas sont peu fréquents.

La structure de pile a un intérêt pédagogique. En effet, les deux représentations des files sont facilement adaptables pour être utilisées pour les

piles. Il existe donc deux implantations des piles. Une première représentation statique range la pile dans un tableau c, et utilise un indice h donnant la hauteur de la pile, tel que $c[0] = a_0$, $c[1] = a_1$, $\ldots c[h-1] = a_{h-1}$. La hauteur de la pile est donc bornée par la taille n du tableau c ($h \leq n$). Une pile est donc représentée par l'enregistrement :

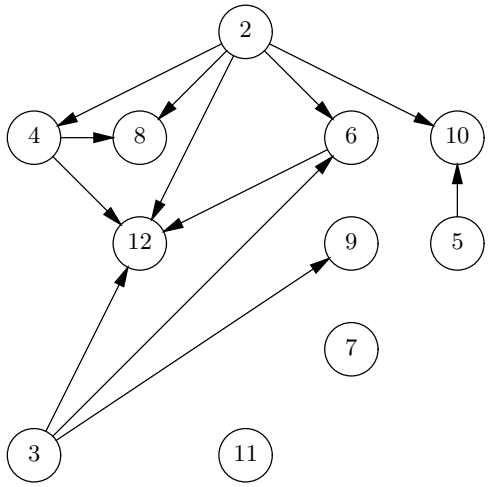
```
struct Pile {
  float[ ] c;
  int hauteur;
}
```

L'opération `ajouter`(x, p) consiste donc à incrémenter la valeur de h et à ranger x en $c[h-1]$. De même, l'opération `supprimer`(p) retourne $a[h-1]$ et décrémente h. Comme pour les files, il faut tester si la pile est vide ($h = 0$) avant une suppression, ou si la pile est pleine ($h = n$) avant un ajout. Dans le cas des piles, on dit qu'on empile x pour un ajout ; on dépile lors d'une suppression ; et l'élément à l'indice $h-1$ est aussi appelé le sommet de pile.

Une structure dynamique est également possible pour les piles, quoique très peu utilisée. Il suffit de représenter la file par la liste $\langle a_{h-1}, \ldots a_1, a_0 \rangle$ des éléments de la pile en ordre inversé. Alors pour `ajouter`(x, p), on insère x en début de liste ; et pour `supprimer`(p), on remplace la liste par sa queue. A nouveau, il faut tester le cas où la liste est vide avant toute suppression, mais il n'y a pas de borne supérieure (sauf la mémoire de l'ordinateur) lors d'un empilement.

1.7 Graphes

Un graphe $G = (S, A)$ a un ensemble S de sommets et un ensemble A d'arcs. Un arc a relie un sommet s à un sommet t ; le premier est l'origine de l'arc, le deuxième est son extrémité. Un exemple de graphe est le plan d'une ville dont les sommets sont les carrefours et les arcs sont les rues (avec leur sens de circulation). Un autre exemple est le plan d'un métro dont les sommets sont les stations et les arcs sont les liens entre stations directement reliées. Il y a bien d'autres exemples ; les graphes interviennent dès qu'une relation binaire (les arcs) existe dans un ensemble d'éléments (les sommets). Un bel exemple est le graphe des diviseurs illustré ci-dessous, où un arc relie un sommet divisant un autre. On dit qu'un graphe est non orienté si chaque fois qu'un arc existe entre s et t, il existe aussi l'arc inverse de t à s. Pour les graphes non orientés, on parle souvent d'arêtes au lieu d'arcs. Un parisien averti sait que le graphe du métro de Paris est un graphe orienté, mais très souvent le graphe d'un métro est non orienté.



Un chemin dans un graphe est une suite d'arcs a_1, a_2, $\ldots a_n$, tels que l'origine de a_{i+1} est l'extrémité de a_i ($1 \leq i < n$). Un chemin élémentaire ne passe pas deux fois par un même sommet ; toutefois l'origine de son premier arc peut être égal à l'extrémité de son dernier arc. Dans ce cas, on dit que le chemin forme un cycle. Un graphe non orienté est donc cyclique dès qu'il a une arête. Les graphes orientés sans cycles sont très utiles ; ce sont les *dags* (*directed acyclic graphs*). Les dags représentent par exemple les dépendances entre tâches dans la construction d'un projet ; dans le cas d'un projet informatique, ils correspondent aux dépendances entre modules, nécessaires pour compiler ou exécuter un gros programme. Un arbre est un autre exemple de dag, où il n'existe qu'un seul chemin entre la racine et tout sommet.

Il y a deux représentations classiques des graphes. La première, statique, représente un graphe de n sommets par une matrice $n \times n$ d'adjacence (m_{ij}), dont chaque élément m_{ij} est un booléen valant vrai s'il existe un arc du i-ème sommet vers le j-ème sommet, et faux sinon. Une deuxième représentation, toujours statique, utilise un tableau `succ` de n listes donnant pour le i-ème sommet x_i la liste `succ`$[i]$ des *successeurs* de x_i, c'est-à-dire l'ensemble des numéros des extrémités des arcs issus de x_i. Par exemple, dans le graphe des diviseurs, alors `succ`$[3]$ est la liste $\langle 12, 9, 6 \rangle$. On peut rendre cette représentation dynamique en considérant une liste de sommets au lieu du tableau `succ` et des listes de sommets directement reliés à chaque sommet par un arc. Mais en général le nombre des sommets d'un graphe ne change

pas ; et l'utilité d'une représentation dynamique d'un graphe est discutable. Un graphe est donc l'enregistrement suivant :

```
struct Graphe { Liste[ ] succ; }
```

où `Liste` est une simple liste d'entiers (désignant pour chaque sommet la liste de ses successeurs).

Il existe de multiples algorithmes sur les graphes, grande source d'inspiration pour les algorithmiciens. Nous ne décrivons ici que les parcours de graphes. Le premier de ces parcours est le parcours en profondeur (*depth-first-search*, ou plus simplement *dfs*. Il s'agit de parcourir tous les sommets d'un graphe sans boucler et en ne rendant visite qu'une seule fois à chaque sommet. La technique utilisée est celle du petit Poucet ; on dépose un caillou sur les sommets par lesquels on passe, et on évite de passer une deuxième fois par un sommet sur lequel il y a déjà un caillou. Informatiquement, on associe une couleur à chaque sommet : un sommet est blanc s'il n'a pas encore été parcouru, un sommet est gris s'il est en cours de visite, un sommet est noir si on a fini de lui rendre visite. Le programme de parcours en profondeur demande donc une initialisation pour mettre à blanc tous les sommets du graphe ; puis on effectue le parcours par la fonction récursive `dfs`(x) qui effectue un parcours *dfs* à partir du sommet de numéro x.

```
void visiter(Graphe g) {
  int n = g.succ.length;
  int[ ] couleur = new int[n];
  foreach x in 0..n-1 do
    couleur[x] = BLANC;
  foreach x in 0..n-1 do
    if (couleur[x] == BLANC)
      dfs(g, x, couleur);
}

void dfs(Graphe g, int x, int[ ] couleur){
  couleur[x] = GRIS;
  foreach y in g.succ[x] do
    if (couleur[y] == BLANC)
      dfs(g, y, couleur);
  couleur[x] = NOIR;
}
```

Le parcours en largeur (*breadth first search*, ou plus simplement *bfs*) explore le graphe selon la distance à un sommet de départ. On choisit un sommet x quelconque comme sommet de départ, puis on rend visite à tous les sommets, non déjà explorés, directement accessibles depuis le sommet de départ, puis aux sommets directement accessibles depuis les sommets directement accessibles depuis le sommet de départ, etc. Il faut donc maintenir une file d'attente pour enregistrer dans l'ordre les sommets à explorer. À chaque étape, on retire le sommet x en tête de la file en ajoutant les successeurs de x, non déjà explorés, au bout de la file. Ce qui donne le programme suivant :

```
void visiter(Graphe g) {
  int n = succ.length;
  int couleur = new int[n];
  FIFO f = new FIFO(n);
  foreach x in 0..n-1 do
    couleur[x] = BLANC;
  foreach x in 0..n-1 do
    if (couleur[x] == BLANC) {
      FIFO.ajouter(f, x);
      couleur[x] = GRIS;
      bfs(g, f, couleur);
} }

void bfs(Graphe g, FIFO f,
                   int[ ]couleur){
  while ( f.vide() == false ) {
    int x = FIFO.supprimer(f);
    foreach y in g.succ[x] do
      if (couleur[y] == BLANC) {
        FIFO.ajouter(f, y);
        couleur[y] = GRIS;
      couleur[x] = NOIR;
} } }
```

Dans le parcours *bfs*, la couleur d'un sommet devient grise dès qu'on l'ajoute dans la file pour éviter de retrouver un même sommet plusieurs fois dans la file ; cette technique est donc différente de celle utilisée pour *dfs* qui grise les sommets quand on leur rend visite, car, dans le parcours *dfs*, on traite immédiatement un sommet blanc. Cette différence est donc minime. Le principe de base consiste bien dans les deux cas à ne parcourir qu'une seule fois tous les sommets du graphe. (Remarquons que le parcours *dfs* peut aussi être réalisé comme *bfs* avec une pile au lieu d'une file.)

A tout parcours de graphe, on associe une structure arborescente correspondant à l'ordre dans lesquels les sommets sont ajoutés dans l'ensemble des sommets à visiter. Dans le parcours *dfs*, c'est l'arbre des appels à la procédure récursive `dfs`. Cette structure arborescente est appelée *forêt de recouvrement* ou souvent plus simplement *arbre de recouvrement*. Un arbre de recouvrement est donc un arbre dont les nœuds sont tous les sommets du graphe à recouvrir, et les branches sont des arcs reliant directement les sommets. Il y a donc plusieurs arbres de recouvrement possibles pour un même graphe, selon les sommets de départ pour parcourir le graphe, et selon la méthode de parcours. Beaucoup des algorithmes sur les graphes sont en fait des algorithmes sur les arbres de recouvrement associés.

Comme exemple d'algorithme sur les graphes, considérons la découverte d'un chemin vers la sortie d'un labyrinthe. Le labyrinthe est donné par un

graphe, avec un sommet de départ d et un sommet de sortie s. On veut donc écrire une fonction `chemin` qui retourne comme résultat un chemin de d à s, s'il existe. (On suppose au début que tous les sommets sont blancs.)

```
Liste chemin(int d, int s,
                    int[ ]couleur){
  couleur[d] = GRIS;
  if (d == s)
    return new Liste(d, null);
    foreach x in succ[d] do {
    if (num[x] == BLANC) {
      Liste r = chemin(x, s);
      if (r != null)
        return new Liste(d, r);
  } }
  return null;
}
```

Dans cette fonction, on peut admirer la puissance de la récursivité associée au parcours *dfs*. La fonction est donc naturellement écrite comme fournissant un chemin de d à s s'il existe un arc de d à x et un chemin de x à s. Remarquons que ce chemin n'est pas forcément le plus court menant à la sortie. Pour trouver un chemin plus court, il vaut mieux faire un parcours *bfs*, mais la fonction est alors moins esthétique.

1.8 Conclusion

Les tableaux, listes, arbres, files et piles ne sont que des représentations différentes d'ensembles éventuellement ordonnés ; chaque représentation a sa propre politique pour ajouter, insérer, ou supprimer un élément. L'art du programmeur consiste à bien choisir sa représentation en fonction de ses besoins. En fait, les paramètres ne sont pas si nombreux : structure statique pour les ensembles dont le nombre d'éléments est borné, ou structure dynamique, souvent moins compacte, où il n'y a pas de borne sur le nombre d'éléments ; structures immuables avec partage pour la programmation fonctionnelle, ou structures modifiables avec effets de bord. De manière générale, il faut toujours utiliser la structure la plus simple, car la simplicité est souvent synonyme d'écriture de programmes corrects. Il faut aussi tenir compte des progrès fulgurants dans la vitesse des ordinateurs qui amoindrissent les gains supposés avec des structures trop sophistiquées. Enfin, on doit bien évaluer le nombre d'éléments des ensembles manipulés. Si ce nombre est petit, il ne faut pas hésiter à utiliser les structures les plus simples avec des recopies si l'ensemble change de taille.

Il existe d'autres structures de données, telles que les graphes valués, les arbres équilibrés ou des structures plus spécifiques. Les algorithmes associés sont souvent époustouflants. Mais ces structures sont presque toujours construites à partir des structures de base vues dans ce chapitre ; et souvent on plaque ces structures plus élémentaires sur ces nouvelles structures (comme, par exemple, pour recouvrir les graphes par des arbres ou des forêts).

Il est donc important de choisir les structures de données en fonction de la taille des ensembles manipulés et des opérations que l'on veut mettre en œuvre sur ces ensembles de valeurs.

Index