Chapitre 3

Structures de données

Dans ce chapitre, nous passons en revue les structures de données élémentaires, à savoir les piles, les files, les listes, ainsi que les arbres binaires et les arbres binaires de recherche. Nous décrivons ensuite les files de priorité et leur implémentation au moyen de tas. La dernière section traite du problème de la gestion efficace d'une partition.

3.1 Types de données et structures de données

Dans un langage de programmation comme Pascal, les objets sont déclarés avec leur *type*. Les types constituent une description du format de représentation interne des données en machine, et ils sont par là-même un outil d'abstraction, puisqu'ils libèrent le programmeur du souci de la représentation physique des données. Ainsi, il n'est pas nécessaire de savoir comment est représentée une variable de type `char` ou `boolean`, voire `string`, du moment que l'on sait la manipuler de la manière convenue.

Les types prédéfinis sont peu nombreux. C'est pourquoi des *constructeurs de types* permettent de définir des types plus complexes, et donc des structures de données moins élémentaires. Là encore, l'implémentation précise d'un `array of char` par exemple importe peu au programmeur (est-il rangé par ligne, par colonne ?), aussi longtemps que sa manipulation satisfait à des règles précises. En revanche, lorsque l'on veut réaliser une pile — disons de caractères — , les problèmes se posent autrement, puisqu'il n'existe pas, en Pascal par exemple, de constructeur permettant de définir des `stack of char` : il est nécessaire de recourir à une structure de données.

Une *structure de données* est l'implémentation explicite d'un ensemble organisé d'objets, avec la réalisation des opérations d'accès, de construction et de modification afférentes.

Un *type de données abstrait* est la description d'un ensemble organisé d'objets et des opérations de manipulation sur cet ensemble. Ces opérations comprennent les moyens d'accéder aux éléments de l'ensemble, et aussi, lorsque l'objet est dynamique, les possibilités de le modifier. Plus formellement, on peut donner une définition mathématique d'un type abstrait : c'est une structure algébrique, formée d'un ou de plusieurs ensembles, munis d'opérations vérifiant un ensemble d'axiomes.

Avec ces définitions, une structure de données est la réalisation, l'implémentation explicite d'un type de données. Décrire un type de données, le *spécifier* comme on dit, c'est décrire les opérations possibles et licites, et leur effet. Décrire une structure de données, c'est expliciter comment les objets sont représentés et comment les opérations sont implémentées. Pour l'évaluation d'algorithmes, il importe de plus de connaître leur coût en temps et en place.

Du point de vue des opérations, un type abstrait est à un type de base d'un langage de programmation ce que l'en-tête de procédure est à un opérateur. La procédure réalise une opération qui n'est pas élémentaire au sens du langage de programmation; de la même manière, un type de données abstrait propose une organisation qui n'est pas offerte par le langage. Une réalisation du type abstrait, c'est-à-dire une implémentation par une structure de données, correspond alors à l'écriture du corps d'une procédure.

Le concept de type abstrait ne dépend pas du langage de programmation considéré, et on peut se livrer à l'exercice (un peu stérile) de définir abstraitement les booléens, les réels, etc. En revanche, l'éventail des types concrets varie d'un langage de programmation à un autre. Ainsi, les piles existent dans certains langages de programmations, les tableaux ou les chaînes de caractères dans d'autres.

On peut également considérer un type abstrait comme une boîte noire qui réalise certaines opérations selon des conventions explicites. Cette vision correspond à un style de programmation que l'on pourrait appeler la *programmation disciplinée*. Le programmeur convient d'accéder à la réalisation d'un type de données uniquement à travers un ensemble limité de procédures et de fonctions, et s'interdit notamment tout accès direct à la structure qui serait rendue possible par une connaissance précise de l'implémentation particulière. De nombreux langages de programmation facilitent ce comportement par la possibilité de créer des modules ou unités séparés.

Lorsqu'un type de données est implémenté, on peut se servir de ses opérations comme opérations élémentaires, et en particulier les utiliser dans l'implémentation de types de données plus complexes. Ceci conduit à une *hiérarchie* de types, où l'on trouve, tout en bas de l'échelle, des types élémentaires comme les booléens, entiers, réels, tableaux, à un deuxième niveau les piles, files, listes, arbres, puis les files de priorités, dictionnaires, graphes.

La *réalisation efficace* de types de données par des structures de données qui implémentent les opérations de manière optimale en temps et en place constitue

l'une des préoccupations de l'algorithmique. Le temps requis par une opération de manipulation d'un type s'exprime en fonction du temps requis par les opérations intervenant dans sa réalisation. Ces opérations elles-mêmes sont peut-être complexes, et implémentées dans une autre structure de données. En descendant la hiérarchie, on arrive à des opérations élémentaires (opérations arithmétiques ou tests sur des types élémentaires, affectation de scalaires) dont le temps d'exécution est considéré, par convention, comme constant, et ceci quel que soit le langage de programmation utilisé.

Pour les types de données abstraits les plus simples, plusieurs réalisations efficaces sont faciles à obtenir et à décrire. Il en est ainsi des piles, des files, des listes, et dans une moindre mesure des dictionnaires et des files de priorités, qui sont des ensembles munis de fonctions d'accès spécifiques. La réalisation d'arbres ayant de bons comportements (arbres 2–4, bicolores, persistants, voir chapitre 6) requiert en revanche un soin certain.

3.2 Les structures linéaires

Une *liste linéaire*sur un ensemble E est une suite finie $(e_1, \ldots, e_n)$ d'éléments de E. La liste est *vide* si $n = 0$. Les éléments de la suite sont soit accessibles directement, par leur indice, soit indirectement, quand la liste est représentée par une suite d'objets chaînés. Dans les implémentations, un indice est représenté par une *place*, et on accède à l'élément de E qui figure à cette place par une fonction particulière appelée *contenu* (ou parfois *clé*) (voir figure 2.1). Cette représentation

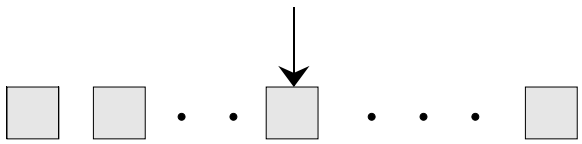

Figure 2.1: *Une liste et une place.*

sépare proprement la position d'un élément dans la liste de l'élément lui-même. De plus, elle permet de définir, sur une place, d'autres fonctions, comme la fonction successeur qui donne la place suivante.

On considère dans cette section plusieurs variantes des listes linéaires et quelques implémentations usuelles. Dans une liste, on distingue deux côtés, le début et la fin, et les opérations de manipulation peuvent se faire de chacun des deux côtés. Selon la nature des opérations autorisées, on définit des listes plus ou moins contraintes. Les *piles* sont des listes où l'insertion et la suppression ne se font que d'un seul et même côté. Les *files* permettent l'insertion d'un côté, et la suppression de l'autre. Les *files bilatères* permettent les insertions et suppressions des deux côtés; enfin, les *listes* ou *listes pointées* autorisent des insertions et suppressions

aussi à l'« intérieur » de la liste et pas seulement aux extrémités. De manière très parlante, une pile est appelée dans la terminologie anglo-saxonne une structure LIFO (« last-in first-out ») et une file une structure FIFO (« first-in first-out »).

3.2.1 Piles

Une *pile* est une liste linéaire où les insertions et suppressions se font toutes du même côté. Les noms spécifiques pour ces opérations sont *empiler* pour insérer et *dépiler* pour supprimer. Si la pile n'est pas vide, l'élément accessible est le *sommet* de la pile. La propriété fondamentale d'une pile est que la suppression annule l'effet d'une insertion : si on empile un élément, puis on dépile, on retrouve la pile dans l'état de départ. Les opérations sur une pile dont les éléments sont de type `élément` sont les suivantes :

PILEVIDE(p : pile) ;
: Crée une pile vide p.

SOMMET(p : pile) : élément ;
: Renvoie l'élément au sommet de la pile p ; bien sûr, p doit être non vide.

EMPILER(x : élément ; p : pile) ;
: Insère x au sommet de la pile p.

DÉPILER(p : pile) ;
: Supprime l'élément au sommet de la pile p ; la pile doit être non vide.

EST-PILEVIDE(p : pile) : booléen ;
: Teste si la pile p est vide.

Avant de discuter les implémentations, donnons un exemple d'utilisation de ce type.

Exemple. Le *tri par insertion* d'une suite $(x_1, \ldots, x_n)$ fonctionne comme suit : si $n = 0$, la suite est triée ; sinon, on trie $(x_1, \ldots, x_{n-1})$ en une suite croissante, puis on insère x_n dans cette suite en le comparant successivement aux éléments de la suite. Plus précisément, l'insertion de x dans une suite $L = (y_1, \ldots, y_m)$ se fait par comparaison :

- si L est vide, le résultat est (x) ; sinon
- si $x < y_1$, le résultat est $(x, y_1, \ldots, y_m)$; sinon
- le résultat est composé de y_1 et du résultat de l'insertion de x dans $(y_2, \ldots, y_m)$.

En représentant une suite triée par une pile, voici comment réaliser le tri par insertion :

```
procédure TRI-PAR-INSERTION(n, L);
  PILEVIDE(L);
  pour i de 1 à n faire INSÉRER(x_i, L);
  pour i de 1 à n faire
    x_i :=SOMMET(L); DÉPILER(L)
  finpour.
```

avec

```
procédure INSÉRER(x, L);
  si EST-PILEVIDE(L) oualors x <SOMMET(L) alors
    EMPILER(x, L)
  sinon
    y :=SOMMET(L); DÉPILER(L);
    INSÉRER(x, L); EMPILER(y, L)
  finsi;
  retourner(L).
```

(L'opérateur *oualors* est un *ou* séquentiel : la deuxième partie du test n'est évaluée que si la première rend la valeur faux. Voir aussi chapitre 1.) La procédure d'insertion demande un nombre d'opérations proportionnel à la longueur de L, et la réalisation du tri par insertion est donc en $O(n^2)$ opérations. L'*implémentation* la plus répandue d'une pile utilise un tableau p et un indice sp (indice de sommet de pile). L'indice indique l'emplacement du sommet de pile (une variante consiste à désigner le premier emplacement vide). Avec ces structures, l'implémentation des opérations est la suivante :

- PILEVIDE(p) se réalise par $sp := 0$;
- SOMMET(p) est : $p[sp]$;
- EMPILER(x, p) se réalise par $sp := sp + 1$; $p[sp] := x$;
- DÉPILER(p) se réalise par $sp := sp - 1$;
- EST-PILEVIDE(p) est : $sp = 0$?.

En toute rigueur, un test de débordement est nécessaire dans la procédure EMPILER, pour éviter de dépasser les bornes du tableau p.

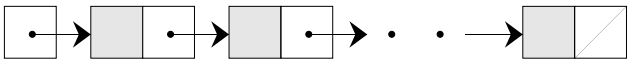

Figure 2.2: *Un pile représentée par une liste chaînée.*

L'implémentation par *liste simplement chaînée* est séduisante parce qu'elle permet une grande souplesse. Une pile est alors réalisée par un pointeur vers un couple formé de l'élément au sommet de pile, et d'un pointeur vers le couple suivant (voir figure 2.2). En Lisp par exemple, les opérations s'implémentent comme suit :

- PILEVIDE(p) se réalise par `(setq p nil)`;
- SOMMET(p) est `(car p)`;
- EMPILER(x, p) se réalise par `(setq p (cons x p))`;
- DÉPILER(p) se réalise par `(setq p (cdr p))`;
- EST-PILEVIDE(p) est `(equal p nil)`.

Chacune de ces opérations demande un temps constant.

3.2.2 Files

Une *file* est une liste linéaire où les insertions se font toutes d'un même côté et les suppressions toutes de l'autre côté. Les noms spécifiques pour ces opérations sont *enfiler* pour insérer et *défiler* pour supprimer. Les opérations sur les files sont syntaxiquement les mêmes que sur les piles; c'est par leur effet qu'elles diffèrent : l'élément supprimé est le premier arrivé dans la file. Ainsi, une file se comporte donc comme une file d'attente; on dit aussi *queue*, en anglais «first-in first-out»(FIFO). Les opérations sur une file dont les éléments sont de type `élément` sont les suivantes :

FILEVIDE(f : file);
: Crée une file vide f.

TÊTE(f : file) : élément;
: Renvoie l'élément en tête de la file f; bien sûr, f doit être non vide.

ENFILER(x : élément; f : file);
: Insère x à la fin de la file f.

DÉFILER(f : file);
: Supprime l'élément en tête de la file f; la file doit être non vide.

EST-FILEVIDE(f : file) : booléen;
: Teste si la file f est vide.

Deux *implémentations* des files sont courantes : l'une par un tableau avec deux variables d'indices, représentant respectivement le début et la fin de la file, l'autre par une liste circulaire. L'implémentation par tableau est simple, et utile quand on sait borner *a priori* le nombre d'insertions, comme cela se présente souvent dans des algorithmes sur les graphes. L'implémentation par *liste circulaire* est plus souple, mais un peu plus complexe (voir figure 2.3).

Dans cette implémentation, une file est repérée par un pointeur sur sa *dernière* place (si la file est vide, le pointeur vaut `nil`). La tête de la file est donc l'élément suivant, et peut être facilement supprimée; de même, l'insertion en fin de file se fait facilement.

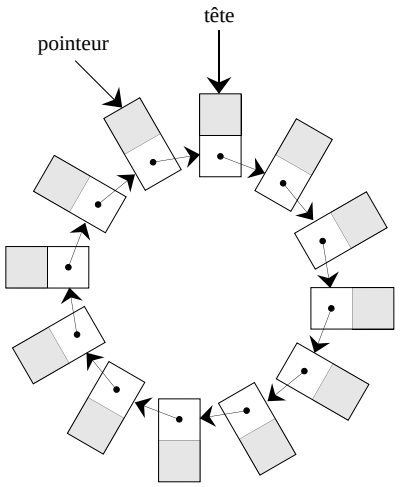

Figure 2.3: *La file* $(c, a, b, c, c, a, b, d, a, c, c, d)$.

Dans la réalisation ci-dessous, on utilise les types suivants :

```
TYPE
   file = ^bloc;
   bloc = RECORD
              cont: element;
              suiv: file
          END;
```

Le type `élément` est supposé connu. Les en-têtes de procédure sont :

```
PROCEDURE filevide  (VAR f : file);
FUNCTION  est_filevide (f : file): boolean;
PROCEDURE enfiler (x: element; VAR f : file);
PROCEDURE defiler (VAR f : file);
FUNCTION  tete (f : file): element;
```

La réalisation fait en plus appel à une fonction `faireplace` qui crée un bloc supplémentaire lorsque cela est nécessaire.

```
PROCEDURE filevide (VAR f: file);
   BEGIN
      f := NIL
   END;

FUNCTION est_filevide (f : file): boolean;
   BEGIN
      est_filevide := f  = NIL
   END;

FUNCTION faireplace (x: element): file;
   VAR
```

```
      q: file;
   BEGIN
      new(q); q^.cont := x; q^.suiv := NIL ;
      faireplace := q
   END;

PROCEDURE enfiler (x: element; VAR f : file);
   VAR
      q: file;
   BEGIN
      IF est_filevide(f) THEN BEGIN
         f  := faireplace(x); f ^.suiv := f
      END
      ELSE BEGIN
         q := faireplace(x);
         q ^.suiv := f ^.suiv; f ^.suiv := q; f  := q
      END
   END;

PROCEDURE defiler (VAR f : file);
   VAR
      q: file;
   BEGIN
      q := f ^.suiv;
      IF f  = f ^.suiv THEN
         f  := NIL
      ELSE
         f ^.suiv := f ^.suiv^.suiv;
      dispose(q)
   END;

FUNCTION tete (f : file): element;
   BEGIN
      tete := f ^.suiv^.cont
   END;
```

3.2.3 Listes

Une *liste* est une liste linéaire où les insertions et suppressions se font non seulement aux extrémités, mais aussi à l'intérieur de la liste. Les listes constituent un type de données très souple et efficace. Ainsi, on peut parcourir des listes (on ne parcourt ni les files ni les piles) sans les détruire, on peut rechercher un élément donné, on peut concaténer des listes, les scinder, etc. Dans la manipulation des listes, la distinction entre les places et leur contenu est importante. Une place est propre à une liste, c'est-à-dire ne peut pas être partagée entre plusieurs listes; en revanche, un élément peut figurer dans plusieurs listes, ou plusieurs fois dans une même liste.

Nous commençons par décrire les opérations de base sur les listes, et exprimons ensuite d'autres opérations à l'aide des opérations de base. Les 5 opérations d'accès sont les suivantes :

EST-LISTEVIDE(L : liste) : booléen ;
Teste si la liste L est vide.

CONTENU(p : place ; L : liste) : élément ;
Donne l'élément contenu dans la place p ; bien sûr, p doit être une place de L.

PREMIER(L : liste) : place ;
Donne la première place dans L ; indéfini si L est la liste vide.

SUIVANT(p : place ; L : liste) : place ;
Donne la place suivante ; indéfini si p est la dernière place de la liste.

EST-DERNIER(p : place ; L : liste) : booléen ;
Teste si la place p est la dernière de la liste L.

Les 4 opérations de construction et de modification sont :

LISTEVIDE(L : liste) ;
Crée une liste vide L, de longueur 0.

INSÉRERAPRÈS(x : élément ; p : place ; L : liste) ;
Crée une place contenant x et l'insère à la place suivant p dans L.

INSÉRERENTÊTE(x : élément ; L : liste) ;
Crée une place contenant x et l'insère comme premier élément de L.

SUPPRIMER(p : place ; L : liste) ;
Supprime l'élément qui se trouve à la place p dans la liste L : si avant la suppression la liste est $(e_1, \ldots, e_p, \ldots, e_n)$, alors après suppression, la liste est $(e_1, \ldots, e_{p-1}, e'_p, \ldots, e'_{n-1})$, avec $e'_j = e_{j+1}$ pour $j = p, \ldots, n-1$.

Au moyen de ces 9 opérations primitives, on peut en réaliser d'autres ; ainsi, TÊTE(L) =CONTENU(PREMIER(L), L) donne le premier élément d'une liste L. En voici d'autres :

CHERCHER(x : élément ; L : liste) : booléen ;
Teste si x figure dans la liste L.

TROUVER(x : élément ; p : place ; L : liste) : booléen ;
Teste si x figure dans la liste L. Dans l'affirmative, p contient la première place dont le contenu est x.

Cette fonction s'écrit comme suit :

```
fonction TROUVER(x : élément; p : place; L : liste) : booléen;
  si EST-LISTEVIDE(L) alors retourner (faux)
  sinon p :=PREMIER(L);
    tantque x ≠CONTENU(p) faire
      si EST-DERNIER(p) alors retourner (faux) sinon p :=SUIVANT(p)
    fintantque
  finsi;
  retourner (vrai).
```

On programme la fonction CHERCHER de la même manière.

Plusieurs *implémentations* des listes sont possibles; le choix de l'implémentation dépend des opérations effectivement utilisées. Si seules les opérations de base sont utilisées, on peut employer une implémentation simple; pour plus de souplesse, on utilisera une implémentation par liste doublement chaînée. C'est elle qui permet notamment de concaténer deux listes, ou de scinder une liste en temps constant.

L'implémentation par un *tableau* est simple : une place correspond à un indice, et la liste est conservée dans les premiers emplacements du tableau. Une variable supplémentaire tient à jour la longueur de la liste. Certaines des fonctions sont particulièrement simples à réaliser, comme *suivant* ou *contenu*. L'insertion et la suppression prennent un temps linéaire en fonction de la taille de la liste, puisqu'il faut déplacer toute la partie de la liste à droite de la position considérée. Dans les arbres par exemple, la liste des fils d'un sommet peut être rangée dans un tableau, si l'on connaît une majoration de leur nombre, comme c'est le cas pour les arbres *a*–*b* (voir chapitre 6).

L'implémentation par une *liste chaînée* est plus souple. Une place est un *pointeur* vers un couple formé de l'élément et d'un pointeur vers le couple suivant. Une liste est un pointeur vers un premier couple. Cette structure permet la réalisation de chacune des 9 opérations de base en un temps constant. En revanche, d'autres opérations, comme la concaténation de deux listes, prennent un temps non constant.

L'implémentation par une *liste doublement chaînée circulaire* est la plus souple, et permet aussi la réalisation efficace d'opérations supplémentaires sur les listes. Détaillons cette structure. Une place est un *pointeur* vers un triplet formé de l'élément et de deux pointeurs, l'un vers la place précédente, l'autre vers la place suivante. La liste est circulaire, de sorte que la première place est la place qui suit la dernière. Une liste est un pointeur vers le premier élément de la liste. Il a la valeur `nil` quand la liste est vide. En Pascal, les déclarations de type sont :

```
TYPE
   place = ^bloc;
   bloc = RECORD
             cont: element;
```

```
            suiv, prec: place
          END;
   liste = place;
```

où `élément` est un type défini par ailleurs. Les en-têtes de procédures et fonctions se déclarent comme suit :

```
(* fonctions d'acces *)
FUNCTION est_listevide (L: liste): boolean;
FUNCTION contenu (p: place; L: liste): element;
FUNCTION premier (L: liste): place;
FUNCTION suivant (p: place; L: liste): place;
FUNCTION est_dernier (p: place; L: liste): boolean;
(* constructeurs *)
PROCEDURE listevide (VAR L: liste);
PROCEDURE insererapres (x: element; p: place; L: liste);
PROCEDURE insererentete (x: element; VAR L: liste);
PROCEDURE supprimer (VAR p: place; VAR L: liste);
```

Voici l'implémentation des cinq fonctions d'accès :

```
IMPLEMENTATION (* des fonctions d'acces *)
FUNCTION est_listevide (L: liste): boolean;
        BEGIN
      est_listevide := L = NIL
        END;
FUNCTION contenu (p: place; L: liste): element;
        BEGIN
      contenu := p^.cont
        END;
FUNCTION premier (L: liste): place;
        BEGIN
      premier := L
        END;
FUNCTION suivant (p: place; L: liste): place;
        BEGIN
      suivant := p^.suiv
        END;
FUNCTION est_dernier (p: place; L: liste): boolean;
        BEGIN
      est_dernier := p^.suiv = L
        END;
```

Pour l'implémentation des fonctions de construction, nous utilisons quelques procédures auxiliaires.

```
PROCEDURE chainer (p, q: place);
        BEGIN
```

```
      q^.prec := p; p^.suiv := q
   END;
```

Cette procédure lie deux places (figure 2.4). Elle est utilisée pour créer, insérer et supprimer une place.

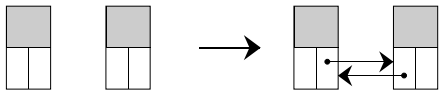

Figure 2.4: *Effet de* `chainer(p,q)`.

La procédure `dechainer` (figure 2.5) délie une place.

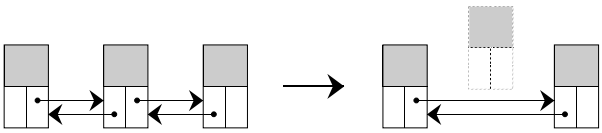

Figure 2.5: *Effet de* `dechainer(q)`.

```
FUNCTION faireplace (x: element): place;
   VAR
        q: place;
        BEGIN
      new(q); faireplace := q;
      q^.cont := x; chainer(q, q)
        END;
PROCEDURE enchainer (p, q: place);
        BEGIN
      chainer(q, p^.suiv); chainer(p, q)
        END;
PROCEDURE dechainer (q: place);
        BEGIN
      chainer(q^.prec, q^.suiv); dispose(q)
        END;
```

Les constructeurs s'écrivent :

```
IMPLEMENTATION (* des constructeurs *)
PROCEDURE listevide (VAR L: liste);
   BEGIN
      L := NIL
        END;
```

```
PROCEDURE insererapres (x: element; p: place; L: liste);
   VAR
      q: place;
   BEGIN
      q := faireplace(x); enchainer(p, q)
   END;
PROCEDURE insererentete (x: element; VAR L: liste);
   VAR
        q: place;
      BEGIN
   IF est_listevide(L) THEN
      L := faireplace(x)
   ELSE BEGIN
      q := faireplace(x);
      enchainer(L^.prec, q);
      L := q
      END
   END;
PROCEDURE supprimer (VAR p: place; VAR L: liste);
   VAR
        q: place;
        BEGIN
      q := p; p := p^.suiv;
      IF p = q THEN
         L := NIL
      ELSE BEGIN
         IF q = L THEN  L := p; dechainer(q)
        END;
      dispose(q)
   END;
```

Voici quelques fonctions ou opérations sur les listes qui sont faciles à réaliser avec les listes doublement chaînées :

DERNIER(L : liste) : place;
: Donne la dernière place dans L; indéfini si L est la liste vide.

SUIVANT-CYCLIQUE(L : liste) : place;
: Donne la place suivante dans L, et la première place de L si p est la dernière; indéfini si L est la liste vide.

PRÉCÉDENT(p : place; L : liste) : place;
: Donne la place précédente; indéfini si p est la première place de L.

INSÉRER-DIRIGÉ(x : élément; p : place; L : liste; d : direction);
: Insère x après la place p dans L si d =après, et avant p si d =avant.

INSÉRER-EN-QUEUE(x : élément; L : liste);
: Insère x en fin de liste.

CONCATÉNER(L_1, L_2 : liste);
: Accroche la liste L_2 à la fin de la liste L_1 : Si $L_1 = (e_1, \ldots, e_n)$ et $L_2 = (e'_1, \ldots, e'_m)$, alors la procédure retourne $L_1 = (e_1, \ldots, e_n, e'_1, \ldots, e'_m)$.

SCINDER(L : liste; p : place; L_1, L_2 : liste);
: Coupe la liste L en deux listes L_1 et L_2 telles que p soit la dernière place de L_1. La liste L est supposée non vide.

Voici une implémentation, en temps constant, des deux dernières opérations :

```
PROCEDURE concatener (VAR L1: liste; L2: liste);
   VAR
        q1, q2: place;
        BEGIN
      IF L1 = NIL THEN  L1 := L2
      ELSE IF L2 = NIL THEN
      ELSE BEGIN
         q1 := dernier(L1); q2 := dernier(L2);
         chainer(q1, L2); chainer(q2, L1)
          END
   END;
PROCEDURE scinder (L: liste; p: place; VAR L1, L2: liste);
   VAR
        q: place;
        BEGIN
      L1 := L;
      IF est_dernier(p, L) THEN L2 := NIL
      ELSE BEGIN
         L2 := p^.suiv; q := dernier(L);
         chainer(p, L1);        chainer(q, L2)
        END
        END;
```

Le *parcours* d'une liste doublement chaînée, avec l'exécution d'une instruction notée I pour chaque place, se fait en temps linéaire, par :

si non EST-LISTEVIDE(L) alors
  p :=PREMIER(L);
  répéter
    I;
    p :=SUIVANT-CYCLIQUE(p)
  jusqu'à p =PREMIER(L)
finsi.

Remarquons que, dans l'implémentation par une liste simplement chaînée, chaque place pointe en fait sur une liste, à savoir sur le reste de la liste commençant à cette

place. Naturellement, la place suivant la dernière est alors la liste vide, en général représentée par `nil`. On peut donc parcourir une telle liste par l'instruction

tantque $p \neq$ `nil` faire I ; p :=SUIVANT(p) fintantque.

Proposition 2.1. *Les opérations de liste* PREMIER, DERNIER, SUIVANT, CONCATÉNER, SCINDER *se réalisent en temps* $O(1)$. *Les opérations* CHERCHER, TROUVER, SUPPRIMER *et le parcours d'une liste se réalisent en temps* $O(n)$, *où* n *est la longueur de la liste.*

3.3 Arbres

On considère d'abord l'implémentation des arbres binaires, puis des arbres plus généraux. Au chapitre suivant, nous étudions plus en détail ces familles d'arbres, et nous nous bornons ici aux structures de données.

3.3.1 Arbres binaires

La représentation la plus naturelle des arbres binaires s'appuie sur leur définition récursive : un arbre binaire est soit l'arbre vide, soit formé d'une *racine* et de deux arbres binaires disjoints, appelés sous-arbres *gauche* et *droit* (voir section 4.3.4). Lorsque l'arbre est étiqueté, ce qui est le cas notamment pour les arbres binaires de recherche, un sommet comporte un champ supplémentaire, qui est son *contenu* ou sa *clé*. Nous commençons par décrire les opérations de base sur les arbres binaires étiquetés, et exprimons ensuite d'autres opérations à l'aide de ces opérations de base. Les 5 opérations d'accès sont les suivantes :

EST-ARBREVIDE(a : arbre) : booléen ;
: Teste si l'arbre a est vide.

CONTENU(s : sommet) : élément ;
: Donne l'élément contenu dans le sommet s.

RACINE(a : arbre) : sommet ;
: Donne le sommet qui est la racine de a ; indéfini si a est l'arbre vide.

SOUS-ARBRE-GAUCHE(a : arbre) : arbre ;
: Donne le sous-arbre gauche ; indéfini si a est l'arbre vide.

SOUS-ARBRE-DROIT(a : arbre) : arbre ;
: Donne le sous-arbre droit ; indéfini si a est l'arbre vide.

Les opérations de construction et de modification sont :

ARBREVIDE(a : arbre) ;
: Crée un arbre vide a.

FAIRE-ARBRE(x : élément ; g, d : arbre) : arbre ;

Crée un sommet qui contient x, et retourne l'arbre ayant ce sommet pour racine, et g et d comme sous-arbres gauche et droit.

FIXER-CLÉ(x : élément ; a : arbre) ;
Le contenu de la racine de a devient x. Indéfini si a est l'arbre vide.

FIXER-GAUCHE(g : arbre ; a : arbre) ;
Remplace le sous-arbre gauche de a par g. Indéfini si a est l'arbre vide.

FIXER-DROIT(d : arbre ; a : arbre) ;
Remplace le sous-arbre droit de a par d. Indéfini si a est l'arbre vide.

Les trois dernières opérations peuvent être réalisées à l'aide de FAIRE-ARBRE : par exemple, FIXER-CLÉ(x, a) est équivalent à FAIRE-ARBRE($x, G(a), D(a)$). Pour faciliter l'écriture et la lecture, nous abrégeons SOUS-ARBRE-GAUCHE en G et SOUS-ARBRE-DROIT en D. Si nous les introduisons explicitement, c'est par souci d'efficacité dans les implémentations : il n'y a pas lieu de créer un nouveau sommet si l'on veut simplement changer son contenu. A l'aide de ces opérations de base, on peut définir de nombreuses autres opérations. Il est par exemple commode de disposer de l'abréviation :

CLÉ(a : arbre) : élément ;
Donne le contenu de la racine de a, indéfini si a est vide.

ainsi que d'opérations sur les feuilles, comme :

EST-FEUILLE(a : arbre) : booléen ;
Teste si la racine de l'arbre a est une feuille ; indéfini si a est vide.

FAIRE-FEUILLE(x : élément) : arbre ;
Crée un arbre réduit à un seul sommet qui contient x.

Bien entendu, FAIRE-FEUILLE(x) s'obtient par ARBREVIDE(g), ARBREVIDE(d), suivi de FAIRE-ARBRE(x, g, d), et EST-FEUILLE(a) est égal à la conjonction de
EST-ARBREVIDE($G(a)$) et EST-ARBREVIDE($D(a)$)

Dans certains cas, il est commode de disposer de FILS-GAUCHE et de FILS-DROIT, procédures qui rendent la racine du sous-arbre gauche, respectivement droit.

3.3.2 Dictionnaires et arbres binaires de recherche

Un *dictionnaire* est un type de données opérant sur les éléments d'un ensemble totalement ordonné, appelés les *clés* et doté des opérations suivantes :

DICOVIDE(d : dictionnaire) ;
Crée un dictionnaire d vide.

EST-DICOVIDE(d : dictionnaire) : booléen;
Teste si le dictionnaire d est vide.

CHERCHER(x : clé; d : dictionnaire) : booléen;
Teste si la clé x figure dans le dictionnaire d.

INSÉRER(x : clé; d : dictionnaire);
Insère x dans le dictionnaire d.

SUPPRIMER(x : clé; d : dictionnaire);
Supprime x dans le dictionnaire d.

Les arbres binaires de recherche se prêtent particulièrement bien à l'implémentation des dictionnaires. On les verra en détail au chapitre 6. Il suffit ici de savoir qu'un arbre binaire de recherche est un arbre binaire dont chaque sommet est muni d'une clé prise dans un ensemble totalement ordonné. De plus, pour chaque sommet de l'arbre, les clés contenues dans son sous-arbre gauche sont strictement inférieures à la clé du sommet considéré et cette clé est elle-même strictement inférieure aux clés contenues dans le sous-arbre droit. En d'autres termes, un parcours symétrique de l'arbre, comme décrit au chapitre 4, donne une liste des clés en ordre croissant. Donnons d'abord les en-têtes des opérations, qui sont les mêmes que dans un arbre :

CHERCHER(x : clé; a : arbre) : booléen;
Teste si x figure dans l'arbre a.

INSÉRER(x : clé; a : arbre);
Insère x dans l'arbre a selon l'algorithme exposé ci-dessous; on suppose que x ne figure pas dans a, et on crée donc un sommet dont la clé est x.

SUPPRIMER(x : clé; a : arbre);
Supprime la clé x et un sommet de l'arbre a selon l'algorithme exposé ci-dessous; on suppose que x figure dans a.

Voici la réalisation de la recherche dans un arbre :

```
fonction CHERCHER(x : clé; a : arbre) : booléen;
  si EST-ARBREVIDE(a) alors retourner (faux)
  sinon si x=CLÉ(a) alors retourner (vrai)
  sinon si x <CLÉ(a) alors
    retourner CHERCHER(x, G(a))
  sinon
    retourner CHERCHER(x, D(a)).
```

Pour l'insertion, on procède de manière similaire :

```
procédure INSÉRER(x : clé; a : arbre);
  si EST-ARBREVIDE(a) alors a :=FAIRE-FEUILLE(x)
  sinon si x <CLÉ(a) alors
    INSÉRER(x, G(a))
  sinon
    INSÉRER(x, D(a)).
```

(La réalisation, en Pascal par exemple, de cette procédure pose quelques problèmes techniques car on ne peut transmettre en référence le résultat de l'évaluation d'une fonction...) La suppression est plus difficile à réaliser parce qu'il faut conserver l'ordre sur l'arbre résultat. Soit x la clé à supprimer, et soit s le sommet dont x est le contenu. Si s est une feuille, il suffit de la supprimer. Si s n'a qu'un seul fils, on le remplace par son fils unique. Si s a deux fils, on cherche son descendant t dont le contenu le précède dans l'ordre infixe : c'est le sommet le plus à droite dans son sous-arbre gauche. On supprime ce sommet (qui n'a pas de fils droit), après avoir remplacé le contenu de s par le contenu de t. (On pourrait aussi bien, symétriquement, remplacer le contenu de s par celui du sommet qui lui succède dans l'ordre infixe.)

Voici une réalisation de cet algorithme :

```
procédure SUPPRIMER(x : clé; a : arbre );
  si x <CLÉ(a) alors SUPPRIMER(x, G(a))
  sinon si x >CLÉ(a) alors SUPPRIMER(x, D(a))
  sinon si EST-ARBREVIDE(G(a)) alors a := D(a)
  sinon si EST-ARBREVIDE(D(a)) alors a := G(a)
  sinon FIXER-CLÉ(SUPPRIMER-MAX(G(a)), a).
```

L'appel à la fonction SUPPRIMER-MAX se fait donc dans le cas où l'arbre a deux sous-arbres non vides, et où le contenu de la racine de a est à supprimer. La fonction SUPPRIMER-MAX réalise à la fois la suppression du sommet de plus grande clé parmi les descendants, et retourne le contenu de ce sommet.

```
fonction SUPPRIMER-MAX(a : arbre ) : clé;
  si EST-ARBREVIDE(D(a)) alors
    SUPPRIMER-MAX :=CLE(a); ARBREVIDE(a)
  sinon SUPPRIMER-MAX :=SUPPRIMER-MAX(D(a)).
```

Plusieurs *implémentations* des arbres sont possibles; la plus souple utilise des pointeurs. A chaque sommet on associe deux pointeurs, l'un vers le sous-arbre

gauche, l'autre vers le sous-arbre droit. Un arbre est donné par un pointeur vers sa racine. Un sommet comporte en plus un champ qui contient la clé associée. On est donc conduit à définir les types suivants :

```
TYPE
   arbre = ^sommet;
   sommet = RECORD
               val: element;
               g, d: arbre
            END;
```

3.3.3 Arborescences

Une *arborescence ordonnée* s'implémente en associant, à chaque sommet, la liste linéaire de ses fils. Lorsque le nombre de fils est borné *a priori*, comme c'est le cas par exemple pour les arbres a–b (voir chapitre 6), on utilisera un tableau pour représenter cette liste, sinon une liste chaînée.

Ainsi, à chaque sommet est associé un couple (premier fils, frère droit). Cette façon d'implémenter les arborescences ordonnées revient en fait à se ramener aux arbres binaires. Plus formellement, on associe à chaque sommet s d'une arborescence ordonnée un sommet $\beta(s)$ d'un arbre binaire dont le fils gauche est l'image, par β, du premier fils de s, et le fils droit l'image du frère suivant de s si s a un frère, l'arbre vide sinon. La figure 3.1 explique la construction.

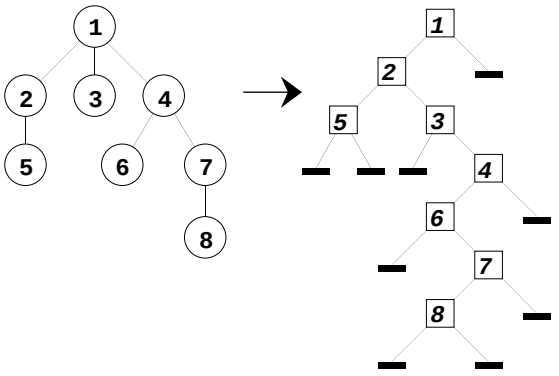

Figure 3.1: *Une arborescence représentée comme arbre binaire.*

Les autres types d'arbres ou d'arborescences s'implémentent comme des arborescences ordonnées.

3.4 Files de priorité

Les files de priorité constituent un type de données intéressant, et qui illustre bien comment une hiérarchie de structures peut permettre d'implémenter un type de données.

Une *file de priorité* est un type de données opérant sur des clés, donc sur les éléments d'un ensemble totalement ordonné, et muni des opérations suivantes :

FILEPVIDE(f : fileP);
: Crée une file de priorité f vide.

EST-FILEPVIDE(f : fileP) : booléen;
: Teste si la file de priorité f est vide.

MINIMUM(f : fileP) : clé;
: Donne la plus petite clé contenue dans f.

SUPPRIMER-MIN(f : fileP);
: Supprime la plus petite clé de f ; indéfini si f est vide.

INSÉRER(x : clé; f : fileP);
: Insère x dans la file de priorité f.

Il est commode de disposer de la combinaison de MINIMUM et de SUPPRIMER-MIN sous la forme :

EXTRAIRE-MIN(f : fileP) : clé;
: Donne la plus petite clé de f, et la supprime de f ; indéfini si f est vide.

Fréquemment, et notamment dans les algorithmes de tri, on convient qu'une même clé peut figurer plusieurs fois dans une file.

Si les clés sont toutes distinctes, on peut réaliser une file de priorité à l'aide d'un arbre binaire de recherche. L'efficacité des opérations dépend alors de l'efficacité de l'insertion et de la suppression de la plus petite clé dans un arbre binaire de recherche. On verra au chapitre 6 comment ces opérations peuvent être réalisées en temps logarithmique, moyennant un rééquilibrage de l'arbre binaire. Nous présentons ici une autre implémentation, au moyen d'un tas.

Un *tas* est un arbre tournoi parfait, défini ci-dessous. On verra comment implémenter un tas très simplement à l'aide d'un tableau.

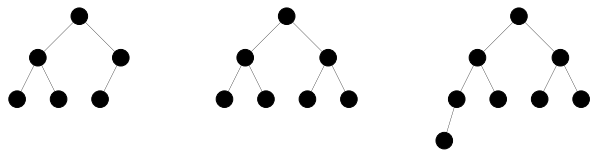

Figure 4.1: *Les arbres parfaits à* $6, 7$*, et* 8 *sommets.*

Un *arbre parfait* est un arbre binaire dont toutes les feuilles sont situées sur deux niveaux au plus, l'avant-dernier niveau est complet, et les feuilles du dernier

niveau sont regroupées le plus à gauche possible. La figure 4.1 donne des exemples d'arbres parfaits. Notons qu'il n'y a qu'un seul arbre parfait à n sommets, pour chaque entier n.

Un *arbre tournoi* est un arbre binaire dont les sommets sont munis d'une clé, et tel que pour tout sommet autre que la racine, la clé du sommet est plus grande que celle de son père. En d'autres termes, les clés sur un chemin croissent de la racine vers une feuille.

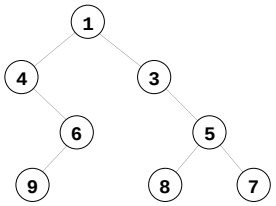

Figure 4.2: *Un arbre tournoi.*

La figure 4.2 montre un arbre tournoi, et la figure 4.3 un tas (arbre tournoi parfait).

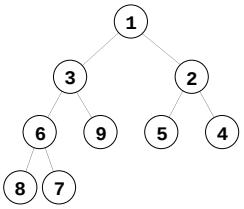

Figure 4.3: *Un tas (tournoi parfait).*

Sur les arbres parfaits, on utilise certaines des opérations sur les arbres binaires : EST-ARBREVIDE, ARBREVIDE, RACINE, PÈRE, FILS-GAUCHE, FILS-DROIT. Par ailleurs, on définit les opérations spécifiques suivantes :

EST-FEUILLE, qui teste si le sommet en argument est une feuille ;

DERNIÈRE-FEUILLE, qui donne la feuille la plus à droite du dernier niveau de l'arbre ;

CRÉER-FEUILLE, qui crée une feuille au dernier niveau de l'arbre, ou entame un niveau supplémentaire si le dernier niveau est plein.

SUPPRIMER-FEUILLE, qui supprime cette feuille.

Ces opérations conservent donc la perfection d'un arbre. Pour les arbres tournoi, la fonction CONTENU donne le contenu (la clé) d'un sommet, et on fait appel à une procédure supplémentaire, ÉCHANGER-CONTENU(p, q) qui échange les clés des sommets p et q.

Etudions comment on implémente les opérations des files de priorités au moyen des opérations des tas. La détermination du minimum est évidente : c'est le contenu de la racine du tas. Elle se fait donc en temps constant.

Pour l'*insertion* d'une clé x, on crée d'abord une feuille contenant cet élément (et qui est donc ajoutée au dernier niveau du tas) ; ensuite on compare son contenu à celui de son père ; s'il est plus petit, on l'échange, et on continue en progressant vers la racine du tas. Voici comment cette procédure se réalise :

```
procédure INSÉRERTAS(x : clé; a : tas);
  q :=CRÉER-FEUILLE(x, a);
  tantque q ≠RACINE(a) etalors CONTENU(PÈRE(q)) >CONTENU(q) faire
    ÉCHANGER-CONTENU(q,PÈRE(q)); q :=PERE(q)
  fintantque.
```

L'*extraction* de la plus petite clé se fait sur un canevas similaire :

```
fonction EXTRAIREMIN(a : tas);
  EXTRAIREMIN :=CONTENU(RACINE(a));
  ÉCHANGER-CONTENU(RACINE(a),DERNIÈRE-FEUILLE(a));
  SUPPRIMER-FEUILLE(a);
  si EST-ARBREVIDE(a) est faux alors
    p :=RACINE(a);
    tantque non EST-FEUILLE(p) faire
      f :=FILS-MIN(p, a);
      si CONTENU(f) <CONTENU(p)
      alors ÉCHANGER-CONTENU(f, p) sinon exit;
      p := f
    fintantque
  finsi.
```

On remplace donc le contenu de la racine par le contenu de la dernière feuille. Cette feuille est supprimée. Puis, on descend de la racine vers les feuilles pour mettre la clé à une place convenable si elle ne l'est pas, c'est à dire si elle est plus grande que l'une des clés des fils. Si le sommet en question a deux fils, on échange la clé du père avec la plus petite des clés de ses fils. On obtient ainsi une correction locale ; on poursuit l'opération tant que nécessaire. Revenons à l'exemple de la figure 4.3.

La suppression de la clé 1 amène la clé 7 à la racine. La descente qui s'ensuit est illustrée sur la figure 4.4. La fonction FILS-MIN(p, a) retourne le fils de p dont le contenu est le plus petit. Elle s'écrit :

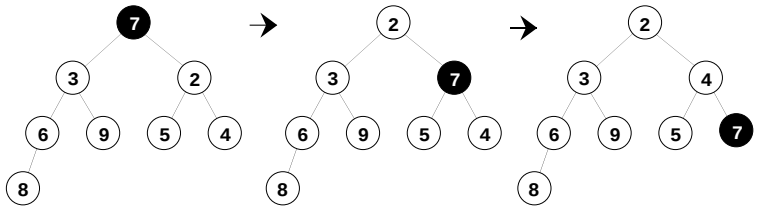

Figure 4.4: *Mise en place de la clé* 7 *par comparaison aux fils.*

```
fonction FILS-MIN(p : sommet; a : tas) : sommet;
  g :=FILS-GAUCHE(p);
  si g =DERNIÈRE-FEUILLE(a) alors FILS-MIN:= g
  sinon d :=FILS-DROIT(p);
    si CONTENU(g) <CONTENU(d) alors FILS-MIN:= g sinon FILS-MIN:= d
  finsi.
```

Il est clair que la hauteur d'un tas est logarithmique en fonction du nombre de ses sommets, donc que l'insertion et l'extraction d'une clé prennent un temps logarithmique, sous réserve que les opérations élémentaires s'implémentent en temps constant.

Comme nous l'avons annoncé plus haut, il existe une implémentation très efficace des tas à l'aide d'un couple formé d'un tableau a et d'un entier n donnant le nombre de clés présents dans le tas. Les sommets du tas étant numérotés niveau par niveau de la gauche vers la droite, chaque sommet est représenté par l'indice d'une clé d'un tableau a. Le contenu d'un sommet p est rangé dans $a[p]$, et est donc accessible en temps constant. Passons en revue les opérations; en face de leur nom, nous indiquons leur réalisation :

EST-ARBREVIDE	$n = 0$?
ARBREVIDE	$n := 0$
RACINE	1
PÈRE(p)	$\lfloor p/2 \rfloor$
FILS-GAUCHE(p)	$2p$
FILS-DROIT(p)	$2p + 1$

Les autres opérations se réalisent aussi simplement :

EST-FEUILLE(p)	$p > \lfloor n/2 \rfloor$?
DERNIÈRE-FEUILLE	n
CRÉER-FEUILLE(x, a)	$n := n + 1; a[n] := x$
SUPPRIMER-FEUILLE	$n := n - 1$
CONTENU(p)	$a[p]$

Seule la procédure ÉCHANGER-CONTENU(p, q) qui échange les clés des sommets p et q, demande trois instructions. L'efficacité pratique de la réalisation des files de priorités est évidemment améliorée si l'on travaille directement sur le tableau, et si l'on remplace les appels de procédures par les instructions correspondantes. Voici une implémentation concrète. On définit les types :

```
TYPE
   sommet = 1..taillemax;
   tas = RECORD
            n: integer;
            a: ARRAY[sommet] OF cle
         END;
```

où `taillemax` et `cle` sont prédéfinis. La traduction des procédures ci-dessus se fait littéralement, comme suit :

```
PROCEDURE inserertas (x: cle; VAR t: tas);
   VAR
      q: sommet;
   BEGIN
      WITH t DO BEGIN
         n := n + 1; a[n] := x; q := n;
         WHILE (q <> 1) AND (a[q DIV 2] > a[q]) DO BEGIN
            echanger(a[q], a[q DIV 2]); q := q DIV 2
         END
      END;
   END;

FUNCTION extrairemin (VAR t: tas): cle;
   VAR
      p, f: sommet;
   FUNCTION fils_min (q: sommet): sommet;
      VAR
         g: sommet;
      BEGIN
         WITH t DO BEGIN
            g := 2 * q;
            fils_min := g;
            IF (g < n) AND (a[g] > a[g + 1]) THEN
               fils_min := g + 1
         END
      END; (* de "fils_min" *)
   BEGIN
      WITH t DO BEGIN
         extrairemin := a[1];
         echanger(a[1], a[n]);
         n := n - 1;
```

```
        IF n > 0 THEN BEGIN
           p := 1;
           WHILE 2 * p <= n DO BEGIN
              f := fils_min(p);
              IF a[f] < a[p] THEN
                 echanger(a[f], a[p])
              ELSE exit(extrairemin);
              p := f
           END;
        END
     END
  END; (* de "extrairemin" *)
```

Bien entendu, `echanger` est une procédure d'échange de deux clés.

3.5 Gestion des partitions

3.5.1 Le problème « union-find »

Dans cette section, nous considérons le problème de la gestion efficace d'une partition d'un ensemble $U = \{1, \ldots, N\}$. Les opérations considérées sont

TROUVER(x)
: Donne le nom de la classe à laquelle appartient l'élément x de U.

UNIR(A, B, C)
: Donne une nouvelle classe, de nom C, qui est la réunion des classes de noms A et B.

Ce problème est appelé le problème de l'union et recherche (« union-find problem » en anglais) ; il apparaît dans de nombreuses situations, et notamment dans la recherche d'un arbre couvrant minimum considéré plus loin.

Dans la suite, nous supposerons que la partition initiale est la partition la plus fine, dont les classes sont composées d'un seul élément. Les noms des classes sont des entiers dans $\{1, \ldots, N\}$, et au début, le nom de la classe $\{i\}$ est i.

Une solution simple du problème est de représenter la partition par un tableau *classe* de taille N qui, pour un élément $x \in U$ contient le nom *classe*$[x]$ de la classe contenant x. L'opération TROUVER se réduit au calcul de *classe*$[x]$. L'opération UNIR(A, B, C) est simple mais plus longue : on parcourt le tableau *classe* et on remplace tous les A et B par C.

La solution que nous considérons maintenant en détail utilise une structure plus élaborée. Chaque classe est représentée par une arborescence, et la partition est représentée par une forêt. Le nom d'une classe est la racine de son arborescence (voir figure 5.1).

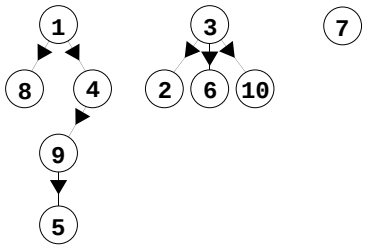

Figure 5.1: *Partition en trois classes, de noms* 1, 3, *et* 7.

La forêt est rangée dans un tableau qui, pour chaque élément x autre qu'une racine, donne le père de x. Le père d'une racine est par exemple le nombre 0. Avec cette représentation, on a :

TROUVER(x)
: Donne le nom de la racine de l'arborescence contenant x.

UNIR(x, y, z)
: Prend les arborescences de racines x et y, et en fait une arborescence de racine z; z est soit égal à x, soit égal à y.

3.5.2 Union pondérée et compression des chemins

Clairement, UNIR(x, y, z) se réalise en temps constant. Le temps d'exécution d'un TROUVER(x) est proportionnel à la profondeur de x. Si au total n unions sont exécutées, ce temps est au plus $O(n)$. On peut réduire considérablement ce temps si l'on parvient à éviter la formation d'arborescences trop «filiformes». Pour cela, il convient de réaliser l'union de manière plus soignée. On utilise à cet effet la règle dite de l'*union pondérée*. Lors de l'exécution de UNIR(x, y, z), la racine de la plus petite des deux arborescences est accrochée comme fils à la racine de la plus grande. Appelons *taille* de x le nombre de sommets de l'arborescence de racine x. Alors l'union pondérée se réalise en temps constant par la procédure que voici :

```
procédure UNION-PONDÉRÉE(x, y, z);
   si taille[x] <taille[y] alors
      z := y; père[x] := y
   sinon
      z := x; père[y] := x
   finsi;
   taille[z] :=taille[x]+taille[y].
```

Nous allons voir que cette règle est très efficace. Notons $t_i(x)$ et $h_i(x)$ la taille et la hauteur du sommet x après la i-ième opération d'union pondérée. On a $t_0(x) = 1$,

$h_0(x) = 0$. Si la i-ième union pondérée est UNIR(x, y, z), et si $t_{i-1}(x) < t_{i-1}(y)$, alors $z = y$, et on a

$$\begin{array}{ll} t_i(x) = t_{i-1}(x), & t_i(y) = t_{i-1}(x) + t_{i-1}(y) \\ h_i(x) = h_{i-1}(x), & h_i(y) = \max(1 + h_{i-1}(x), h_{i-1}(y)) \end{array}$$

Lemme 5.1. *Si l'on utilise la règle de l'union pondérée, alors*

$$t_i(x) \geq 2^{h_i(x)}$$

pour tout $x \in U$. En particulier, après n unions, on a $h_n(x) \leq \log(n+1)$.

Preuve. Si $h_i(x) = 0$, alors $t_i(x) = 1$. Supposons donc $h_i(x) \geq 1$. Alors x a un fils y avec $h_i(y) = h_i(x) - 1$. Considérons l'opération d'union pondérée qui a fait de y un fils de x. Si cette opération est la j-ième, avec $j \leq i$, alors $t_{j-1}(x) \geq t_{j-1}(y)$, et donc $t_j(y) = t_{j-1}(y)$ et $t_j(x) \geq 2 \cdot t_j(y)$. Ensuite, comme y n'est plus racine d'une arborescence, sa taille ne varie plus. En revanche, la taille de x peut augmenter. De même, la hauteur de y ne varie plus, donc $h_i(y) = h_{j-1}(y)$. Par récurrence, on a $t_{j-1}(y) \geq 2^{h_{j-1}(y)}$, donc

$$t_i(x) \geq t_j(x) \geq 2 \cdot t_j(y) \geq 2 \cdot 2^{h_i(y)} = 2^{h_i(x)}.$$

Comme n unions ne créent que des arborescences d'au plus $n+1$ sommets, on a $t_n(x) \leq n+1$ pour tout x, d'où la seconde inégalité. ∎

Proposition 5.2. *Avec la représentation par forêt et la règle de l'union pondérée, une suite de $n-1$ opérations* UNION-PONDÉRÉE *et de m opérations* TROUVER *se réalise en temps $O(n + m \log n)$.*

Preuve. Chaque union pondérée prend un temps $O(1)$. De plus, chaque arborescence obtenue durant les $n-1$ unions a une hauteur au plus $\log n$, donc une opération TROUVER prend un temps $O(\log n)$. ∎

Une autre façon de diminuer le coût des TROUVER est la *compression des chemins*. Si l'on exécute un TROUVER(x), on parcourt un chemin $x_0, x_1, \ldots, x_\ell$ de $x = x_0$ vers la racine $r = x_\ell$ de l'arborescence contenant x. On peut comprimer ce chemin en faisant un deuxième parcours du chemin pendant lequel on fait, de chaque sommet x_i, $0 \leq i < \ell$, un fils de la racine r (voir figure 5.2). Même si cette compression ne réduit pas le coût de ce TROUVER (en fait, elle en double le coût), les coûts des TROUVER subséquents qui s'appliquent aux sommets dans les sous-arbres des x_i sont fortement réduits. Voici la réalisation du TROUVER avec compression :

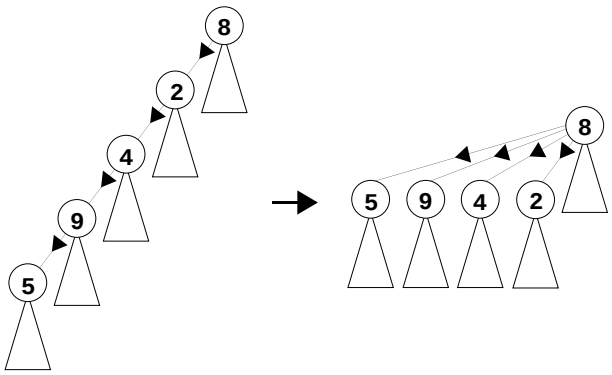

Figure 5.2: *Compression d'un chemin.*

```
procédure TROUVER-AVEC-COMPRESSION(x);
  r := x;
  tantque r n'est pas racine faire r :=père[r] fintantque;
  tantque x ≠ r faire
    y :=père[x]; père[x] := r; x := y
  fintantque;
  retourner(r).
```

La compression des chemins seule conduit déjà à un algorithme efficace. La combinaison de la compression des chemins et de l'union pondérée fournit un algorithme très efficace, dont le temps d'exécution est presque linéaire. Pour les valeurs des paramètres que l'on rencontre en pratique, l'algorithme est linéaire.

Pour pouvoir présenter l'analyse de l'algorithme, nous avons besoin d'une notation. Soit $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ la fonction définie par

$$\begin{array}{ll} A(i,0) = 1 & \text{pour } i \geq 1; \\ A(0,j) = 2j & \text{pour } j \geq 0; \\ A(i+1,j+1) = A(i,A(i+1,j)) & \text{pour } i,j \geq 0. \end{array}$$

Cette fonction est une variante de la fonction d'Ackermann. Elle croît très rapidement, et n'est pas primitive récursive. Voici une table des premières valeurs de A :

$i\backslash j$	0	1	2	3	4	5
0	0	2	4	6	8	10
1	1	2	4	8	16	32
2	1	2	4	16	65536	2^{65536}
3	1	2	4	65536	$2^{\cdot^{\cdot^{\cdot^{2}}}}\Big\}65536$ fois	

Soit α la fonction définie pour $m \geq n$ par

$$\alpha(m,n) = \min\{z \geq 1 \mid A(z, 4\lceil m/n \rceil) > \log n\}$$

Cette fonction est une sorte d'inverse de la fonction A, et croît très lentement. Pour $m \geq n$, on a $\alpha(m,n) \leq \alpha(n,n)$. On a $\alpha(n,n) = 1$ pour $1 \leq n < 2^{16}$, et $\alpha(n,n) = 2$ pour $2^{16} \leq n < 2^{65536}$. Pour des valeurs de m et n apparaissant en pratique, on a donc $\alpha(m,n) \leq 2$.

Nous allons prouver le résultat suivant, dont la démonstration n'est pas facile, et peut être omise en première lecture :

Théorème 5.3. *Avec les règles de l'union pondérée et de la compression des chemins, une suite de $n-1$ opérations* UNIR *et de m opérations* TROUVER *($m \geq n$) se réalise en temps $O(m \cdot \alpha(m,n))$.*

3.5.3 Preuve du théorème

Note 3.5.1

Pour démontrer le théorème 5.3, nous reformulons le problème. Soit T une forêt obtenue par $n-1$ opérations UNIR avec la règle de l'union pondérée. Nous autorisons que la forêt soit ensuite modifiée par ce que nous appelons des « TROUVER partiels ». Un TROUVER partiel dans une arborescence parcourt un chemin $x_0, x_1, \ldots, x_k$ d'un sommet x_0 à un sommet x_k qui n'est pas nécessairement la racine. Ensuite, chacun des sommets $x_0, x_1, \ldots, x_{k-1}$ est fait fils du sommet x_k. Le coût du TROUVER partiel est k.

Il est facile de voir qu'une suite arbitraire de $n-1$ opérations UNIR et de m TROUVER peut être simulée par une suite de $n-1$ UNIR *suivie* d'une suite de m TROUVER partiels, et que le coût de la simulation est le coût de la suite originelle. En effet, les UNIR de la simulation sont juste les UNIR de la suite de départ, et chaque TROUVER de la suite de départ est remplacé par un TROUVER partiel avec les mêmes extrémités. Il suffit donc de démontrer la majoration pour une suite de $n-1$ opérations UNIR suivies de m TROUVER partiels.

Soit T la forêt obtenue après les $n-1$ opérations UNIR, et soit $h(x)$ la hauteur d'un sommet x dans T. Pour chaque arc (x,y) de T (les arcs sont orientés en direction de la racine), on a $h(x) < h(y)$.

Le coût total des TROUVER partiels est proportionnel au nombre d'arcs parcourus lors de tous les TROUVER partiels. Soit F l'ensemble de ces arcs. Nous allons montrer que $|F| = O(m\alpha(m,n))$ en répartissant les arcs en groupes, et en évaluant le nombre d'arcs dans chaque groupe. La répartition des arcs de F se fait en fonction de la hauteur de leurs extrémités. C'est pourquoi on définit, pour $i, j \geq 0$, les ensembles de sommets

$$G_{i,j} = \{x \mid A(i,j) \leq h(x) < A(i,j+1)\}.$$

Ainsi, on a par exemple

$$\begin{array}{ll} G_{0,0} = \{x \mid 0 \leq h(x) \leq 1\} & \\ G_{k,0} = \{x \mid h(x) = 1\} & \text{pour } k \geq 1 \\ G_{k,1} = \{x \mid 2 \leq h(x) \leq 3\} & \text{pour } k \geq 0 \end{array}$$

Lemme 5.4. *Pour $k \geq 0$ et $j \geq 1$, on a $|G_{k,j}| \leq 2n/2^{A(k,j)}$.*

Preuve. Soit $\ell \geq 1$ un entier avec $A(k,j) \leq \ell < A(k,j+1)$, et comptons le nombre de sommets x tels que $h(x) = \ell$. Deux sommets distincts de hauteur ℓ sont racines de sous-arbres disjoints, et la taille du sous-arbre est, d'après le lemme 5.1, au moins 2^ℓ. Il y a donc au plus $n/2^\ell$ sommets de hauteur ℓ. Par conséquent,

$$|G_{k,j}| \leq \sum_{\ell=A(k,j)}^{A(k,j+1)-1} \frac{n}{2^\ell} \leq \frac{2n}{2^{A(k,j)}}$$ ∎

L'ensemble F des arcs des TROUVER partiels est réparti en classes N_k pour $k = 0, \ldots, z+1$, où z est un paramètre fixé ultérieurement et une classe N' par :

$$N_k = \{(x,y) \in F \mid k = \min\{i \mid \exists j : x, y \in G_{i,j}\}\}$$

pour $0 \leq k \leq z$, et

$$N' = \{(x,y) \in F \mid h(x) = 0,\ h(y) \geq 2\}$$

enfin

$$N_{z+1} = F - N' - \bigcup_{0 \leq k \leq z} N_k$$

Pour un $i \geq 1$ fixé, les nombres $A(i,j)$, $(j \geq 0)$ partitionnent les entiers positifs en intervalles $[A(i,j), A(i,j+1)[$, et $(x,y) \in N_k$ si k est le plus petit entier i tel que les hauteurs de x et y appartiennent au *même* intervalle relativement aux nombres $A(i,j)$, $j \geq 0$. Dans N_{z+1} on trouve tous les arcs (x,y) pour lesquels ce nombre est plus grand que k, à l'exception des arcs (x,y) pour lesquels x est une feuille. Ceux-ci se répartissent dans N_0 si $h(y) = 1$, et dans N', si $h(y) \geq 2$.

Posons, pour $0 \leq k \leq z+1$,

$$L_k = \{(x,y) \in N_k \mid \text{sur le chemin du TROUVER partiel contenant } (x,y), \text{ l'arc } (x,y) \text{ est le dernier arc dans } N_k\}$$

Notons en effet que si (x,y) figure sur le chemin d'un TROUVER partiel, les sommets x et y deviennent ensuite frères, donc ne peuvent plus se retrouver sur un tel chemin.

Lemme 5.5. *On a les inégalités suivantes :*

(1) $|L_k| \leq m$ *pour* $0 \geq k \geq z+1$; $|N'| \leq m$;
(2) $|N_0 - L_0| \leq n$;
(3) $|N_k - L_k| \leq n$ *pour* $1 \leq k \leq z$;
(4) $|N_{z+1} - L_{z+1}| \leq b(z,n)$, *avec* $b(z,n) = \min\{i \mid A(i,z) \geq \log n\}$.

Preuve. (1) Chaque TROUVER partiel a au plus un arc dans L_k, donc $|L_k| \leq m$. De même, chaque TROUVER partiel a au plus un arc (x, y) dans N', parce que x est une feuille, donc $|N'| \leq m$.

(2) Soit $(x, y) \in N_0 - L_0$. Il existe j tel que

$$2j = A(0, j) \leq h(x) < h(y) < A(0, j + 1) = 2(j + 1)$$

donc tel que $h(x) = 2j$ et $h(y) = 2j + 1$. Par ailleurs, il existe un arc $(s, t) \in N_0$ qui vient après l'arc (x, y) sur le chemin du même TROUVER partiel. On a donc

$$h(y) \leq h(s) < h(t)$$

A nouveau, il existe j' tel que $h(s) = 2j'$, $h(t) = 2j' + 1$, donc en fait $j < j'$ et $h(y) < h(s)$. Après compression du chemin, le père de x est un ancêtre de t, soit u, et en particulier $h(u) \geq h(t) > A(0, j + 1)$. Par conséquent, aucun TROUVER partiel ultérieur ne peut contenir un arc dont x est l'extrémité initiale, et qui appartienne à N_0. Il en résulte que $|N_0 - L_0| \leq n$.

Une partie de la preuve est commune pour les cas (3) et (4). Soit x un sommet, et supposons que $x \in G_{k,j}$ pour un k avec $1 \leq k \leq z + 1$, c'est-à-dire

$$A(k, j) \leq h(x) < A(k, j + 1).$$

Soit q le nombre d'arcs dans $N_k - L_k$ débutant par x. Notons-les

$$(x, y_1), \ldots, (x, y_q)$$

avec $h(y_1) \leq \cdots \leq h(y_q)$. Pour chaque i, $1 \leq i \leq q$, il existe un arc $(s_i, t_i) \in N_k$ qui figure ultérieurement dans le chemin du TROUVER partiel de (x, y_i). Par conséquent,

$$h(y_i) \leq h(s_i) < h(t_i)$$

De plus, après compression du chemin, le père de x est un ancètre de t_i (éventuellement t_i lui-même). Ceci prouve que $h(t_i) \leq h(y_{i+1})$. Maintenant, on a $(x, y_i) \notin N_{k-1}$ et $(s_i, t_i) \notin N_{k-1}$. Par définition, il existe $j' < j''$ avec

$$h(x) < A(k - 1, j') \leq h(y_i) \leq h(s_i) < A(k - 1, j'') \leq h(t_i)$$

et en particulier

$$h(y_i) < A(k - 1, j'') \leq h(y_{i+1}).$$

Il existe donc un entier ℓ tel que

$$h(y_1) < A(k - 1, \ell) \leq A(k - 1, \ell + q - 2) \leq h(y_q) \tag{5.1}$$

Les preuves de (3) et (4) se séparent maintenant.

Fin de la preuve de (3). Nous montrons que pour $x \in G_{k,j}$, $1 \leq k \leq z$, $j \geq 0$, il y a au plus $A(k, j)$ arcs dans $N_k - L_k$. Ceci est évident si $q = 1$. C'est également

clair si $j = 0$ ou $j = 1$, car pour ces valeurs de j, il n'y a pas d'arc dans N_k. On peut donc supposer $q \leq 2$, $j \leq 2$. Comme $x \in G_{k,j}$ et $y_q \in N_k$, on a $y_q \in G_{k,j}$ et par conséquent

$$h(y_q) < A(k, j+1) = A(k-1, A(k, j))$$

Ceci prouve, en vue de (5.1), que

$$A(k-1, \ell+q-2) < A(k-1, A(k, j))$$

donc

$$\ell + q - 2 < A(k, j)$$

Or $\ell \geq 1$ parce que $h(y_1) \leq 2$ et $A(k-1, \ell) > h(y_1)$, donc $A(k-1, \ell) > 2$. Par conséquent

$$q \leq A(k, j)$$

De cette inégalité, on obtient

$$|N_k - L_k| \leq \sum_{j \geq 2} |G_{k,j}| \cdot A(k, j)$$

et en majorant $|G_{k,j}|$ par la valeur donnée dans le lemme 5.4

$$|N_k - L_k| \leq \sum_{j \geq 2} 2n \cdot A(k, j)/2^{A(k,j)}$$

et comme $A(k, j) \geq 2^j$,

$$|N_k - L_k| \leq \sum_{j \geq 2} 2n \cdot \frac{2^j}{2^{2^j}} \leq 5n/8$$

Fin de la preuve de (4). Par le lemme 5.1, on a $h(y_q) \leq \log n$, et de l'équation (5.1) on obtient

$$A(z, \ell + q - 2) \leq \log n$$

La définition même de $b(z, n)$ implique que

$$\ell + q - 2 \leq b(z, n)$$

et comme ci-dessus, $q \leq b(z, n)$. Il y a donc au plus $b(z, n)$ arcs dans $N_{z+1} - L_{z+1}$ pour chaque sommet x qui n'est pas une feuille, d'où l'inégalité cherchée. ∎

Preuve du théorème. On a

$$\begin{aligned} |F| &= \sum_{k=0}^{z+1} |L_k| + |N'| + \sum_{k=0}^{z+1} |N_k - L_k| \\ &\leq m(z+3) + n + \frac{5}{8} nz + b(z, n) \end{aligned} \tag{5.2}$$

Fixons le paramètre z à

$$z = \min\{i \mid A(i, 4\left\lceil \frac{m}{n} \right\rceil) > \log n\} = \alpha(m, n).$$

Alors

$$b(z, n) = b(\alpha(m, n), n) = \min\{j \mid A(\alpha(m, n), j) > \log n\} \leq 4\left\lceil \frac{m}{n} \right\rceil \leq 8\frac{m}{n}$$

donc par (5.2)

$$|F| \leq (m + \frac{5}{8}n)\alpha(m, n) + 11m + n = O(m, \alpha(m, n))$$

parce que $m \leq n$. ∎

Notes

Un *type abstrait* est un ensemble organisé d'objets, muni d'un ensemble d'opérations permettant de les manipuler. Le langage des *types abstraits algébriques* permet de donner une description formelle des types abstraits, comme éléments d'une algèbre hétérogène particulière. Chaque opération sur un type est considérée comme une application (éventuellement partielle) qui, à une ou plusieurs instances du type, associe une nouvelle instance du type. Par exemple, empiler est une fonction (*pile, objet*) $\mapsto$ *pile*, et dépiler est une fonction *pile* $\mapsto$ *pile* qui n'est définie que si la pile de départ n'est pas vide. Ces fonctions doivent en outre vérifier certaines propriétés qui s'expriment souvent par des équations. Ainsi, pour toute pile p, et tout objet x, on doit avoir

$$\textit{dépiler}(\textit{empiler}(p, x)) = p.$$

Un type abstrait algébrique est défini alors comme l'ensemble de toutes les algèbres satisfaisant ces conditions. Cette approche algébrique de la description d'un type abstrait est présentée systématiquement dans

C. Froidevaux, M. C. Gaudel, M. Soria, *Types de données et algorithmes*, McGraw-Hill, 1990.

Les structures de données décrites dans ce chapitre sont des plus classiques, et sont traitées dans tous les livres d'algorithmique. Une présentation voisine est donnée dans le livre de Froidevaux, Gaudel, Soria, et dans

T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, MIT Press, McGraw-Hill, 1990.

L'analyse de l'algorithme de gestion des partitions est due à Tarjan. Nous suivons l'exposition de Mehlhorn. Tarjan a montré que l'algorithme n'est pas linéaire en général. L'existence d'un algorithme linéaire reste ouvert.

Les pagodes sont dues à Françon, Viennot et Vuillemin.

Exercices

3.1. Montrer que l'on peut implémenter une pile avec deux files. Quelle est la complexité en temps des opérations ?

3.2. Montrer que l'on peut implémenter une file avec deux piles. Quelle est la complexité en temps des opérations ?

3.3. Implémenter une liste avec un tableau.

3.4. Ecrire une procédure PURGER qui supprime toutes les répétitions dans une liste. Par exemple, le résultat de cette procédure sur la liste (a, b, a, a, b, c, a) est (a, b, c).

3.5. Ecrire une procédure RENVERSER qui retourne une liste. Par exemple, le résultat de cette procédure sur la liste (a, b, c, a, b, d) est (d, b, a, c, b, a).

3.6. Ecrire une procédure FUSION qui fusionne deux listes. Les listes sont triées au départ, le résultat est également trié, et un élément qui figurerait dans les deux listes de départ ne figure qu'une fois dans la liste résultat. Par exemple, la fusion des listes $(3, 5, 8, 11)$ et $(2, 3, 5, 14)$ est la liste $(2, 3, 5, 8, 11, 14)$.

3.7. Les arborescences ordonnées dont les sommets ont un nombre borné de fils peuvent s'implémenter en associant à chaque sommet un tableau de pointeurs vers ses fils. Réaliser les opérations de manipulation d'arborescences ordonnées dans cette représentation.

3.8. Un *arbre fileté* («threaded tree») est une structure de données pour représenter les arbres binaires, déclarée par

```
TYPE
   arbre = ^sommet;
   sommet = RECORD
               val: element;
               g, d: arbre;
               gvide, dvide: boolean
            END;
```

Pour un sommet s, `gvide` est vrai si et seulement si le sous-arbre gauche de s est vide, et dans ce cas, `g` pointe sur le sommet qui précède s en ordre symétrique. Dans le cas contraire, `g` pointe comme usuellement vers la racine du sous-arbre gauche de s. Bien entendu, la même convention vaut pour le sous-arbre droit.

a) Décrire des algorithmes pour parcourir un arbre binaire fileté en ordre préfixe, suffixe, symétrique.

b) Décrire l'insertion et la suppression dans un tel arbre, et décrire les implications des rotations (voir chapitre 6) sur cette représentation.

3.9. Une *pagode* est une représentation des tournois. Chaque sommet s d'une pagode contient deux pointeurs $g(s)$ et $d(s)$ définis comme suit.

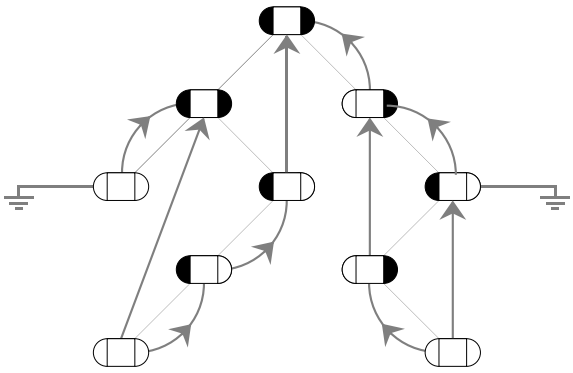

Figure 5.3: *Un arbre fileté.*

(1) si s est la racine ou est un fils droit, $g(s)$ pointe vers le premier sommet en ordre symétrique du sous-arbre de racine s ;
(2) si s est un fils gauche, $g(s)$ pointe vers le père de s.

Le pointeur $d(s)$ est défini de manière symétrique.

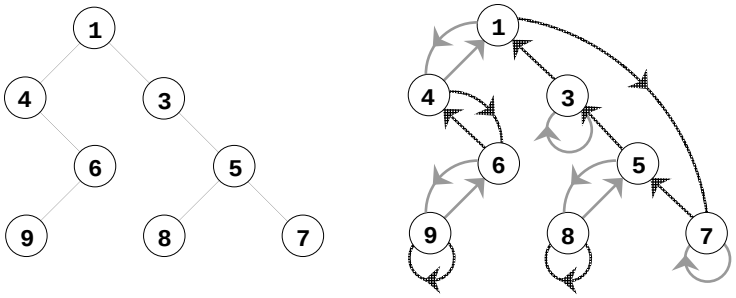

Figure 5.4: *Un tournoi et sa pagode.*

a) Ecrire une procédure d'insertion d'une clé dans une pagode, en l'insérant le plus à droite possible. En déduire qu'un parcours symétrique donne les clés dans leur ordre d'insertion.

b) Ecrire une procédure d'extraction de la plus petite clé présente.

c) Décrire un algorithme de fusion de deux pagodes.

3.10. On peut utiliser, pour la représentation d'une partition, en plus du tableau *classe*, une liste circulaire pour chaque classe. Réaliser les opérations UNIR et TROUVER et montrer qu'avec la règle de l'union pondérée, on peut réaliser une suite de $n-1$ UNIR et de m TROUVER en temps $O(m + n \log n)$.

3.11. Analyser l'algorithme de gestion de partitions qui utilise l'UNIR naïf et un TROUVER avec compression des chemins.