STRUCTURES DE DONNEES

INTRODUCTION

Ce document est un résumé concernant les structures les plus classiques rencontrées en informatique pour organiser des données. On suppose que le lecteur connait déjà les **tableaux** et les **enregistrements** (exemple: `record` en Pascal, `struct` en C). Pour aborder les différentes structures de données présentées ici, le lecteur devra également bien maîtriser la notion de **pointeurs** et de **gestion dynamique de la mémoire**.

Les structures de données présentées ici sont:

- les **tableaux** (***arrays*** en anglais),
- les **listes chaînées** (***linked lists*** en anglais),
- les **piles** (***stacks*** en anglais),
- les **files** (***queues*** en anglais),
- les **arbres binaires** (***binary trees*** en anglais).

Pour chacune de ces structures de données, nous présentons avant tout différentes manières de les modéliser. Ensuite, nous détaillons en langage algorithmique les principales opérations qui peuvent être appliquées sur ces structures. Enfin, pour certaines d'entre elles, nous développons quelques exemples d'utilisation.

NOTATIONS

Avant d'entrer dans les détails de chaque structure, nous introduisons ici quelques notations qui seront utilisées tout au long de ce document. Elles permettront de formaliser les modélisations proposées pour les différentes structures de données ainsi que les opérations applicables sur ces structures.

Opérateurs

- `*p` est le contenu pointé par p;
- `T *` est le type pointeur sur un élément de type T;
- `&x` est l'adresse de l'élément x;
- `x ← y` affecte la valeur y à la variable x;

- `/* x */` signifie que x est un commentaire;
- `=`, `<=`, `<`, `!=`, `>`, `>=` sont les opérateurs de test d'égalité, d'infériorité ou d'égalité, d'infériorité, de différence, de supériorité et de supériorité ou d'égalité;
- `rendre x` termine la fonction en cours et renvoie la valeur x à la fonction appelante;
- `x.y` est le champ y dans la structure x;
- `x→y` est le champ y dans la structure pointée par x.

Déclarations

Fonction

On définit une fonction de la manière suivante.

```
fonction TR ← f(TX x, TY y):
 ...
fin fonction;
```

Dans cet exemple, f a deux paramètres, x de type TX et y de type TY, et renvoie un élément de type TR.

Type

On déclare un nouveau type de donnée de la manière suivante.

```
type TX: TY *;
```

Dans cet exemple, le type TX est défini comme étant un pointeur sur un élément de type TY.

Enregistrement / Structure

On définit un enregistrement, appelé aussi une structure ici, de la manière suivante.

```
structure S:
 TX x;
 TY y;
fin structure;
```

Dans cet exemple, la structure s est composée de deux champs: x de type TX et y de type TY.

Types et constantes

- **BOOLEEN** est le type booléen, il prend uniquement les valeurs **VRAI** ou **FAUX**;

- **ENTIER** est le type nombre entier;
- **ELEMENT** est le type des éléments stockés dans une structure de données;
- **NIL** est une constante symbolique, un pointeur qui a cette valeur est un pointeur qui pointe sur rien du tout.

Instructions

- **T * ← ALLOUER(T, ENTIER n)** est une instruction qui alloue un espace mémoire pouvant contenir n éléments de type T. Si l'allocation est possible, la fonction retourne l'adresse de l'espace alloué. Dans le cas contraire, la valeur NIL est retournée, indiquant que l'allocation a échouée.
- **LIBERER(T * p)** est une instruction qui libére l'espace mémoire pointé par p. Cet espace doit avoir été alloué auparavant avec l'instruction ALLOUER.

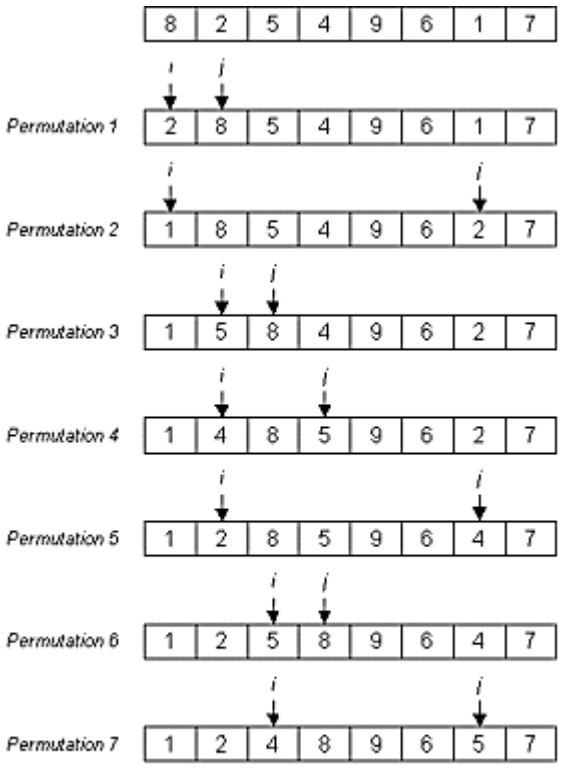
1. TABLEAUX

INTRODUCTION

Dans ce chapitre, nous allons présenter deux méthodes pour trier les éléments d'un tableau. Nous ne présenterons pas les algorithmes les plus efficaces. Nous avons choisi de présenter tout d'abord la méthode de tri dite "par sélection". Il s'agit d'une méthode qui n'est pas très rapide. Ensuite, nous présenterons la méthode dite "par fusion" qui est beaucoup plus efficace. Dans ce chapitre, nous utiliserons la fonction PLUS_PETIT(a,b) pour trier. Cette fonction renvoie VRAI si l'élément a est plus petit que l'élément b.

TRI PAR SELECTION

Cette méthode est très simple. Supposons que l'on veuille trier les n éléments du tableau t. On commence par parcourir le tableau pour trouver la plus petite valeur. On la place à l'indice 0. Ensuite, on recommence à parcourir le tableau à partir de l'indice 1 pour trouver la plus petite valeur que l'on stocke à l'indice 1. Et ainsi de suite pour l'indice 2, 3 jusqu'à n - 2. La figure suivante montre comment l'algorithme fonctionne sur un tableau de 8 éléments. Seulement quelques étapes sont représentées.



...

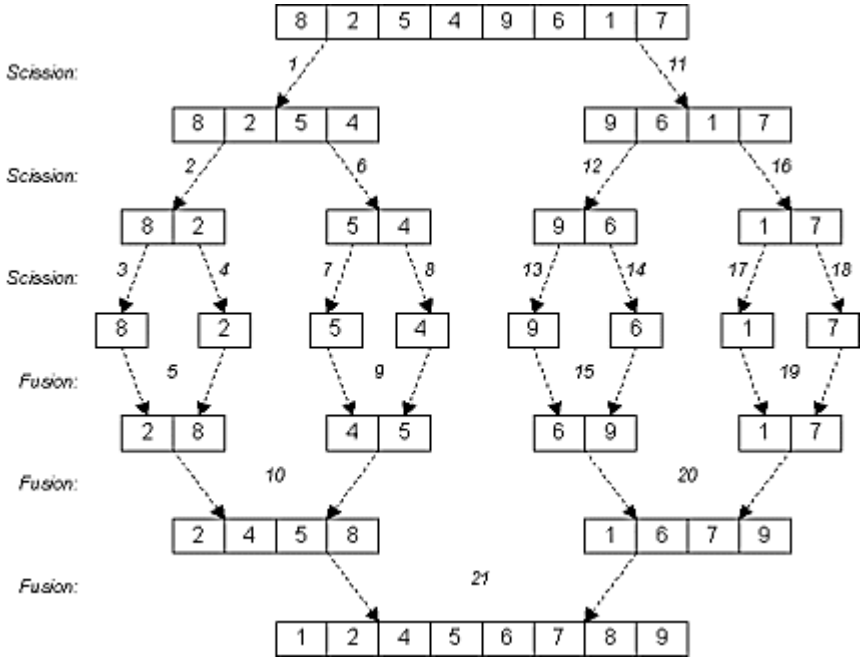
La fonction se déroule de la manière suivante. Le tableau est parcouru du premier élément

(indice 0) à l'avant dernier (indice n - 2). On note i l'indice de l'élément visité à une itération donnée. On compare l'élément i avec chaque élément j qui suit dans le tableau, c'est-à-dire de l'indice i + 1 jusqu'à l'indice n - 1. Si l'élément d'indice j est plus petit que l'élément d'indice i alors on permute i et j dans le tableau. Voici le détail de la fonction de tri.

```
fonction trierSelection(ELEMENT * t, ENTIER n):
 i ← 0;

 tant que (i < n - 1) faire
  j ← i + 1;

  tant que (j < n) faire
   si (PLUS_PETIT(t[j],t[i])) alors
    tmp ← t[j];
    t[j] ← t[i];
    t[i] ← tmp;
   fin si;

   j ← j + 1;
  fin tant que;

  i ← i + 1;
 fin tant que;
fin fonction;
```

TRI PAR FUSION

L'idée de cette méthode est la suivante. Pour trier un tableau t de n éléments, on le scinde en deux tableaux de même taille (à un élément près). On les note t1 de taille n1 et t2 de taille n - n1. Ces deux tableaux sont ensuite triés (appel récursif) et enfin fusionnés de manière à reformer le tableau t trié. La figure suivante reprend l'exemple du tri par sélection et montre comment le tri par fusion fonctionne au travers d'étapes numérotées de 1 à 21.



Pour réaliser ce tri, on a besoin de plusieurs fonctions dont voici la liste.

- `scinder(ELEMENT * t, ENTIER n, ELEMENT * t1,`
  `ENTIER n1, ELEMENT * t2)`
  Copie les n1 premiers éléments du tableau t dans un tableau t1 et le reste dans un tableau t2.

- `ENTIER ← concatener(ELEMENT * t1, ENTIER n1, ELEMENT * t2,`
  `ENTIER n2, ENTIER i2)`
  Copie le tableau t2 de taille n2 à la fin du tableau t1 de taille initiale n1. La copie débute à l'indice i2 dans t2. Après la copie, la nouvelle taille de t1 est retournée par la fonction.

- `fusionner(ELEMENT * t, ELEMENT * t1, ENTIER n1,`
  `ELEMENT * t2, ENTIER n2)`
  Recopie les éléments des tableaux t1 et t2 dans le tableau t de façon à ce qu'ils soient triés. Les éléments de t1 et de t2 sont supposés triés.

- `trierFusion(ELEMENT * t, ENTIER n)`
  Trie les n éléments du tableau t par la méthode de tri par fusion.

Scinder un tableau

La fonction scinder copie les n1 premiers éléments du tableau t dans t1 et le reste dans t2.

```
fonction scinder(ELEMENT * t, ENTIER n, ELEMENT * t1,
                 ENTIER n1, ELEMENT * t2):
 i ← 0;
 j ← 0;

 tant que (i < n1) faire
  t1[i] ← t[i];
  i ← i + 1:
 fin tant que;

 tant que (i < n) faire
  t2[j] ← t[i];
  j ← j + 1;
  i ← i + 1;
 fin tant que;
fin fonction;
```

Concaténer deux tableaux

Cette fonction copie le tableau t2 à la fin du tableau t1 de taille initiale n1. On suppose que t1 a la capacité suffisante pour recevoir tous les éléments de t2. Le tableau t2 est parcouru, en commençant à partir de l'indice i2. Chaque case de t2 visitée est copiée à l'indice n1 qui est augmenté d'une unité. A la fin de l'exécution, n1 est retourné puisqu'il exprime la nouvelle taille de t1.

```
fonction ENTIER ← concatener(ELEMENT * t1, ENTIER n1,
                             ELEMENT * t2, ENTIER n2,
                             ENTIER i2):
 i ← 0;

 tant que (i < n2) faire
  t1[n1] ← t2[i2 + i];
  n1 ← n1 + 1;
```

```
  i ← i + 1;
 fin tant que;

 rendre n1;
fin fonction;
```

Fusionner deux tableaux

Cette fonction fusionne les deux tableaux t1 de taille n1 et t2 de taille n2 supposés triés dans le tableau t. La fusion se fait de façon à ce que t soit trié. Pour cela, on parcours t1 et t2 parallèlement. Quand l'élément visité dans t1 est plus petit que celui visité dans t2, on copie l'élément de t1 dans t et on passe à l'élément suivant de t1, sinon on copie celui de t2 et on avance dans t2. On progresse comme cela jusqu'à ce que l'un des deux tableaux ait été complètement visité. Dans ce cas, on copie la partie non visitée de l'autre tableau directement dans t.

```
fonction fusionner(ELEMENT * t, ELEMENT * t1, ENTIER n1,
                   ELEMENT * t2, ENTIER n2):
 i1 ← 0;
 i2 ← 0;
 i ← 0;

 tant que (i1 < n1 et i2 < n2) faire
  si (PLUS_PETIT(t1[i1],t2[i2])) alors
   t[i] ← t1[i1];
   i1 ← i1 + 1;
  sinon
   t[i] ← t2[i2];
   i2 ← i2 + 1;
  fin si;

  i ← i + 1;
 fin tant que;

 i ← concatener(t,i,t1,n1 - i1,i1);
 concatener(t,i,t2,n2 - i2,i2);
fin fonction;
```

Trier un tableau par fusion

Cette fonction effectue le tri du tableau t de n éléments. Elle alloue d'abord la mémoire nécessaire pour t1 et t2. Ensuite, elle copie chaque moitié de t dans t1 et t2. Ensuite, par appel récursif, elle trie les tableaux t1 et t2. Enfin, elle fusionne ces deux tableaux dans t et libère la mémoire occupée par t1 et t2. On utilise la fonction ENT qui retourne la partie entière d'un nombre.

```
fonction trierFusion(ELEMENT * t, ENTIER n):
 si (n > 1) alors
  n1 ← ENT(n / 2);
  t1 ← ALLOUER(ELEMENT,n1);
  t2 ← ALLOUER(ELEMENT,n - n1);

  si (t1 ≠ nil et t2 ≠ nil) alors
   scinder(t,n,t1,n1,t2);
   trierFusion(t1,n1);
   trierFusion(t2,n - n1);
   fusionner(t,t1,n1,t2,n - n1);
   LIBERER(t1);
```

```
    LIBERER(t2);
   sinon
    /* Erreur: Pas assez de mémoire. */
    si (t1 ≠ nil) LIBERER(t1);
    si (t2 ≠ nil) LIBERER(t2);
   fin si;
  fin si;
fin fonction;
```

CONCLUSION

Dans ce chapitre, nous avons vu deux méthodes pour trier les éléments d'un tableau. La méthode par sélection est très simple à mettre en oeuvre et nécessite peu de mémoire. Par contre, elle est très lente. A l'opposé, la méthode par fusion est un peu plus compliquée à écrire et nécessite beaucoup plus de mémoire. En contrepartie, elle est plus rapide.

En effet, la méthode par sélection effectue un nombre d'opérations de l'ordre de n^2 opérations pour un tableau de n éléments. La méthode par fusion effectue quant à elle $n \log(n)$ opérations pour un tableau de même taille. Pour simplifier, $\log(n)$ peut être vu comme le nombre de fois que l'on peut diviser le nombre n par 2 avant d'arriver à 1. Par exemple, $245 / 2 = 122$, $122 / 2 = 61$, $61 / 2 = 30$, $30 / 2 = 15$, $15 / 2 = 7$, $7 / 2 = 3$, $3 / 2 = 1$. Donc, on considérera que $\log(245)$ vaut 7.

2. LISTES CHAINEES

MODELISATION DE LA STRUCTURE

Introduction

Une liste est une structure qui permet de stocker de manière ordonnée des éléments. Une liste est composée de maillons, un maillon étant une structure qui contient un élément à stocker et un pointeur (au sens large) sur le prochain maillon de la liste.

```
structure MAILLON:
 ELEMENT  elt;
 POINTEUR suiv;
fin structure;
```

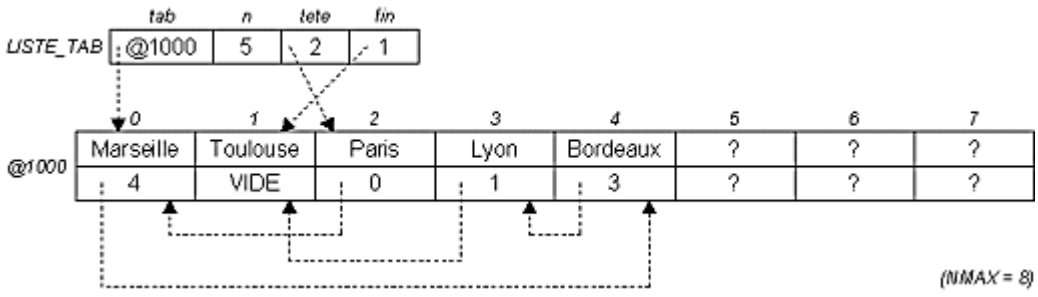
On parle de pointeur au sens large car il y a principalement deux manières de représenter une liste chaînée, ce qui entraîne deux types de pointeur qui sont précisés par la suite.

Modélisation par tableau

Une première façon de modéliser une liste chaînée consiste à allouer à l'avance un certain nombre de maillons. Autrement dit, un tableau de maillons, de taille fixée, est alloué à la création de la liste. Par la suite, la liste est formée en utilisant comme maillons des cases de ce tableau. Avec cette modélisation, un pointeur sur un prochain maillon dans la liste sera tout simplement un indice dans le tableau.

```
type POINTEUR: ENTIER;
```

Par convention, un pointeur sur rien aura la valeur VIDE = -1. La figure suivante représente une liste chaînée par cette modélisation. La liste contient les chaînes de caractères suivantes classées dans cet ordre: "Paris", "Marseille", "Bordeaux", "Lyon", "Toulouse".



Voici la structure de données correspondant à une liste chaînée représentée par tableau.

```
structure LISTE_TAB:
 MAILLON  tab[NMAX];
 ENTIER   n;
```

```
 POINTEUR tete;
 POINTEUR fin;
fin structure;
```

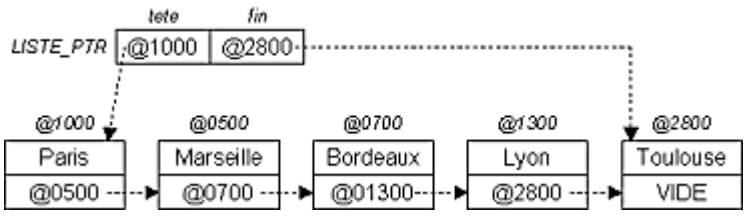
NMAX est une constante représentant la taille du tableau alloué. n est le nombre d'éléments dans la liste. tete et fin pointent respectivement sur la tête (premier élément) et la queue (dernier élément) de la liste.

Modélisation par pointeurs

Une seconde façon de modéliser une liste chaînée consiste à allouer dynamiquement les maillons chaque fois que cela est nécessaire. Dans ce cas, un pointeur sur un maillon sera bien ce que l'on appelle couramment un pointeur, c'est-à-dire un pointeur sur la mémoire centrale de l'ordinateur.

```
type POINTEUR: MAILLON *;
```

Un pointeur sur rien aura donc la valeur VIDE = NIL. La figure suivante représente une liste chaînée par cette modélisation. Elle reprend la même liste que pour l'exemple de modélisation par tableau.



Voici la structure de données correspondant à une liste chaînée représentée par pointeurs.

```
structure LISTE_PTR:
 POINTEUR tete;
 POINTEUR fin;
fin structure;
```

Comme pour la représentation par tableau, tete et fin pointent respectivement sur la tête et la queue de la liste.

OPERATIONS SUR LA STRUCTURE

Introduction

A partir de maintenant, nous allons employer le type LISTE qui représente une liste chaînée au sens général, c'est-à-dire sans se soucier de sa modélisation. LISTE représente aussi bien une liste par tableau (LISTE_TAB) qu'une liste par pointeurs (LISTE_PTR). La présentation des opérations appliquées sur une liste chaînée est divisée en deux parties. Tout d'abord, on présente quelques opérations de base dont l'implémentation (le contenu) est propre à chacune des modélisations.

- `initialiserListe(LISTE * l)`

Initialise la liste pointée par l, i.e. fait en sorte que la liste soit vide.

- `MAILLON ← CONTENU(LISTE l, POINTEUR p)`
  Retourne le maillon pointé par le pointeur (au sens large) p dans la liste l. Attention, ceci n'est pas une fonction, CONTENU va seulement remplacer une série d'instructions. Il n'y a donc pas de mécanisme d'appel de fonction mis en jeu. C'est important, car le maillon retourné par CONTENU peut alors être modifié par affectation. On peut comparer cela à une macrocommande en C (e.g. `#define`).

- `POINTEUR ← allouerMaillon(LISTE * l)`
  Alloue un nouveau maillon pour la liste pointée par l et retourne son adresse. En cas d'échec de cette allocation, la valeur VIDE est retournée.

- `libererMaillon(LISTE * l, POINTEUR p)`
  Libère l'espace occupé par le maillon pointé par p appartenant à la liste pointée par l.

Néanmoins, quelle que soit la modélisation choisie pour la liste chaînée, ces opérations ont les mêmes prototypes (mêmes paramètres et type de retour), ce qui permet par la suite d'écrire des opérations générales indépendantes de la modélisation choisie.

- `BOOLEEN ← listeVide(LISTE l)`
  Indique si la liste chaînée l est vide.

- `POINTEUR ← preparerMaillon(LISTE * l, ELEMENT e)`
  Alloue un maillon pour la liste pointée par l et y place l'élément e. Si aucun maillon n'a pu être alloué, la valeur VIDE est retournée.

- `BOOLEEN ← rechercherMaillon(LISTE l, POINTEUR * p)`
  Recherche un maillon dans la liste l selon un critère donné. L'adresse du maillon qui précède le maillon trouvé est alors stockée à l'adresse p. Si le maillon trouvé est le premier de la liste, VIDE est stockée à l'adresse p. La fonction retourne VRAI uniquement si un maillon a été trouvé correspondant au critère de recherche.

- `insererMaillon(LISTE * l, POINTEUR p1, POINTEUR p2)`
  Insère le maillon pointé par p2 dans la liste pointée par l, juste après le maillon pointé par p1. Si p1 = VIDE, alors l'insertion s'effectue en tête de la liste.

- `supprimerMaillon(LISTE * l, POINTEUR p)`
  Supprime de la liste pointée par l le maillon suivant celui pointé par p. Si p = VIDE, alors la suppression s'effectue en tête de la liste.

- `BOOLEEN ← ajouterTete(LISTE * l, ELEMENT e)`
  Ajoute l'élément e en tête de la liste pointée par l. VRAI est retournée si l'opération a réussi.

- `BOOLEEN ← ajouterQueue(LISTE * l, ELEMENT e)`
  Ajoute l'élément e en queue de la liste pointée par l. VRAI est retournée si l'opération a réussi.

- `BOOLEEN ← retirerTete(LISTE * l)`
  Retire l'élément en tête de la liste pointée par l. VRAI est retournée si l'opération a réussi.

- `ELEMENT ← teteListe(LISTE l)`
  Retourne l'élément en tête de la liste l.

- `ELEMENT ← queueListe(LISTE l)`
  Retourne l'élément en queue de la liste l.

Opérations pour la modélisation par tableau

Initialiser une liste

Cette fonction initialise les valeurs de la structure représentant la liste pointée par l pour que celle-ci soit vide. Dans le cas d'une représentation par tableau, une liste est vide lorsque tete et fin pointent sur VIDE. Dans ce cas, il faut également que n soit nul.

```
fonction initialiserListe(LISTE * l):
 l→n ← 0;
 l→tete ← VIDE;
 l→fin ← VIDE;
fin fonction;
```

Contenu d'un pointeur

Comme nous l'avons précisé précédemment, CONTENU n'est pas une fonction mais un mot-clé qui remplace littéralement un jeu d'instructions. Cette opération retourne donc le contenu d'un pointeur p sur un maillon d'une liste l. Dans le cas d'une représentation par tableau, il s'agit de retourner le maillon d'indice p du tableau représentant la liste l.

```
macro MAILLON ← CONTENU(LISTE l, POINTEUR p):
 rendre (l.tab[p]);
fin macro;
```

Allouer un maillon

Cette fonction réserve l'espace mémoire nécessaire pour un nouveau maillon dans la liste pointée par l et retourne un pointeur sur ce maillon. Dans le cas d'une représentation par tableau, on prendra comme nouveau maillon un maillon non utilisé dans le tableau l→tab. Le plus simple ici est de prendre le maillon pointé par n qui représente le nombre de maillons utilisés par la liste. n est alors augmenté d'une unité. Si n vaut déjà NMAX alors il ne reste plus de maillon disponible dans le tableau. La valeur VIDE est alors retournée.

```
fonction POINTEUR ← allouerMaillon(LISTE * l):
 si (l→n < NMAX)
  l→n ← l→n + 1;
  rendre (l→n - 1);
 sinon
  rendre VIDE;
 fin si;
fin fonction;
```

Libérer un maillon

Cette fonction libère l'espace occupé par un maillon à l'adresse p dans une liste pointée par l. Dans le cas d'une représentation par tableau, on supprimera un maillon en décalant d'une case vers la gauche tous les maillons dont l'indice est supérieur à p dans l→tab. Bien entendu, n est diminué d'une unité. Ensuite, il faut parcourir tous les maillons pour diminuer d'une unité tous les pointeurs qui sont supérieurs à p. De même si le pointeur sur la tête ou sur la queue est supérieur à p, il faut le diminuer d'une unité.

```
fonction libererMaillon(LISTE * l, POINTEUR p):
 decalerMaillons(l,p);
 majPointeurs(l,p);

 si (l→tete != VIDE et l→tete > p) alors
  l→tete ← l→tete - 1;
 fin si;

 si (l→fin != VIDE et l→fin > p) alors
  l→fin ← l→fin - 1;
 fin si;
fin fonction;

fonction decalerMaillons(LISTE * l, POINTEUR p):
 tant que (p + 1 < l→n) faire
  l→tab[p] ← l→tab[p + 1];
  p ← p + 1;
 fin tant que;

 l→n ← l→n - 1;
fin fonction;

fonction majPointeurs(LISTE * l, POINTEUR p):
 i ← 0;

 tant que (i < l→n) faire
  si (l→tab[i].suiv > p) alors
   l→tab[i].suiv ← l→tab[i].suiv - 1;
  fin si;

  i ← i + 1;
 fin tant que;
fin fonction;
```

Opérations pour la modélisation par pointeurs

Initialiser une liste

Cette fonction initialise les valeurs de la structure représentant la liste pointée par l pour que celle-ci soit vide. Dans le cas d'une représentation par pointeurs, une liste est vide lorsque tete et fin pointent sur VIDE.

```
fonction initialiserListe(LISTE * l):
 l→tete ← VIDE;
 l→fin ← VIDE;
fin fonction;
```

Contenu d'un pointeur

Comme nous l'avons précisé précédemment, CONTENU n'est pas une fonction mais un mot-

clé qui remplace littéralement un jeu d'instructions. Cette opération retourne donc le contenu d'un pointeur p sur un maillon d'une liste l. Dans le cas d'une représentation par pointeurs, il s'agit de retourner le maillon pointé directement par p.

```
macro MAILLON ← CONTENU(LISTE l, POINTEUR p):
 rendre (*p);
fin macro;
```

Allouer un maillon

Cette fonction réserve l'espace mémoire nécessaire pour un nouveau maillon dans la liste pointée par l et retourne un pointeur sur ce maillon. Dans le cas d'une représentation par pointeurs, l'espace nécessaire est alloué en mémoire centrale.

```
fonction POINTEUR ← allouerMaillon(LISTE * l):
 rendre (ALLOUER(MAILLON,1));
fin fonction;
```

Libérer un maillon

Cette fonction libère l'espace occupé par un maillon à l'adresse p dans une liste pointée par l. Dans le cas d'une représentation par pointeurs, il suffit de libérer l'espace alloué dans la mémoire centrale.

```
fonction libererMaillon(LISTE * l, POINTEUR p):
 LIBERER(p);
fin fonction;
```

Opérations générales

Liste vide ?

Cette fonction indique si la liste l est vide. Une liste est vide si la tête pointe sur VIDE.

```
fonction BOOLEEN ← listeVide(LISTE l):
 rendre (l.tete = VIDE);
fin fonction;
```

Préparer un maillon

Cette fonction alloue un nouveau maillon pour la liste pointée par l et place un élément e à l'intérieur. Si un maillon a pu être alloué, son adresse est retournée. Sinon, la valeur VIDE est renvoyée.

```
fonction POINTEUR ← preparerMaillon(LISTE * l, ELEMENT e):
 p ← allouerMaillon(l);

 si (p != VIDE) alors
  CONTENU(*l,p).elt ← e;
 fin si;

 rendre p;
```

```
fin fonction;
```

Rechercher un maillon

Cette fonction recherche un maillon dans la liste l. Le critère de recherche est défini par le mot-clé TROUVE(e) qui retourne VRAI si l'élément e correspond au critère de recherche. Une fois l'élément trouvé, l'adresse du maillon qui le précède est stockée à l'adresse p. Dans le cas où le maillon trouvé est le premier de la liste, alors VIDE est stockée à l'adresse p. La fonction renvoie VRAI si elle a trouvé un élément correspondant au critère de recherche.

```
fonction BOOLEEN ← rechercherMaillon(LISTE l, POINTEUR * p):
 *p ← VIDE;
 p2 ← l.tete;

 tant que (p2 != VIDE et non TROUVE(CONTENU(l,p2).elt)) faire
  *p ← p2;
  p2 ← CONTENU(l,p2).suiv;
 fin tant que;

 rendre (p2 != VIDE);
fin fonction;
```

Insérer un maillon

Cette fonction insère, dans la liste pointée par l, le maillon pointé par p2 juste après le maillon pointé par p1. Si p1 = VIDE, alors l'insertion se fait en tête de la liste. Les liens de chaînage sont modifiés pour intégrer le nouveau maillon. On prend garde de mettre à jour également le pointeur sur la queue de la liste.

```
fonction insererMaillon(LISTE * l, POINTEUR p1, POINTEUR p2):
 si (p1 = VIDE) alors
  /* Insertion en tête */
  CONTENU(*l,p2).suiv ← l→tete;
  l→tete ← p2;
 sinon
  /* Insertion au milieu */
  CONTENU(*l,p2).suiv ← CONTENU(*l,p1).suiv;
  CONTENU(*l,p1).suiv ← p2;
 fin si;

 si (CONTENU(*l,p2).suiv = VIDE) alors l→fin ← p2;
fin fonction;
```

Supprimer un maillon

Cette fonction supprime, dans la liste pointée par l, le maillon suivant celui pointé par p. Si p = VIDE, alors c'est l'élément de tête qui est supprimé. Les liens de chaînage sont modifiés et l'espace mémoire occupé par le maillon est libéré. On prend garde de mettre à jour le pointeur sur la queue de la liste.

```
fonction supprimerMaillon(LISTE * l, POINTEUR p):
 si (p = VIDE) alors
  /* Suppression en tête */
  m ← l→tete;
  l→tete ← CONTENU(*l,m).suiv;
  libererMaillon(l,m);
```

```
  si (l→tete = VIDE) alors l→fin ← VIDE;
 sinon
  /* Suppression au milieu */
  m ← CONTENU(*l,p).suiv;
  CONTENU(*l,p).suiv ← CONTENU(*l,m).suiv;
  libererMaillon(l,m);
  si (CONTENU(*l,p).suiv = VIDE) alors l→fin ← p;
 fin si;
fin fonction;
```

Ajouter en tête de liste

Cette fonction insère l'élément e en tête de la liste pointée par l. Si l'opération réussi (i.e. il reste de la mémoire pour créer un maillon) alors la valeur VRAI est renvoyée.

```
fonction BOOLEEN ← ajouterTete(LISTE * l, ELEMENT e):
 p ← preparerMaillon(l,e);

 si (p != VIDE) alors
  insererMaillon(l,VIDE,p);
  rendre VRAI;
 fin si;

 rendre FAUX;
fin fonction;
```

Ajouter en queue de liste

Cette fonction insère l'élément e en queue de la liste pointée par l. Si l'opération réussi (i.e. il reste de la mémoire pour créer un maillon) alors la valeur VRAI est renvoyée.

```
fonction BOOLEEN ← ajouterQueue(LISTE * l, ELEMENT e):
 p ← preparerMaillon(l,e);

 si (p != VIDE) alors
  insererMaillon(l,l→fin,p);
  rendre VRAI;
 fin si;

 rendre FAUX;
fin fonction;
```

Retirer la tête de liste

Cette fonction supprime le maillon en tête de la liste pointée par l. Si la liste n'est pas déjà vide, la valeur VRAI est renvoyée.

```
fonction BOOLEEN ← retirerTete(LISTE * l):
 si (non listeVide(*l)) alors
  supprimerMaillon(l,VIDE);
  si (l→tete = VIDE) alors l→fin ← VIDE;
  rendre VRAI;
 sinon
  rendre FAUX;
 fin si;
fin fonction;
```

Elément en tête de liste

Cette fonction retourne le maillon en tête de la liste l. A n'utiliser que si la liste l n'est pas vide.

```
fonction ELEMENT ← teteListe(LISTE l):
 si (non listeVide(l)) alors
  rendre (CONTENU(l,l.tete).elt);
 sinon
  /* Erreur */
 fin si;
fin fonction;
```

Elément en queue de liste

Cette fonction retourne le maillon en queue de la liste l. A n'utiliser que si la liste l n'est pas vide.

```
fonction ELEMENT ← queueListe(LISTE l):
 si (non listeVide(l)) alors
  rendre (CONTENU(l,l.fin).elt);
 sinon
  /* Erreur */
 fin si;
fin fonction;
```

CONCLUSION

La représentation par tableau d'une liste chaînée n'a finalement que peu d'intérêt par rapport à un tableau, si ce n'est au niveau de l'insertion. En effet, pour insérer un élément à n'importe quelle position dans une liste chaînée par tableau, il suffit d'ajouter un élément à la fin du tableau et de faire les liens de chaînage. Alors que dans un tableau, l'insertion d'un élément implique le décalage vers la droite d'un certain nombre d'éléments. Cependant au niveau de la suppression, la liste chaînée par tableau a le même défaut qu'un tableau: il faut décaler un certain nombres d'éléments vers la gauche.

Dans le cas d'une liste chaînée par pointeurs, le défaut constaté au niveau de la suppression d'un élément disparait. En résumé, une liste chaînée par pointeurs permet une insertion et une suppression rapide des éléments. Cependant, contrairement au tableau, une liste chaînée interdit un accès direct aux éléments (mis à part la tête et la queue).

3. PILES

MODELISATION DE LA STRUCTURE

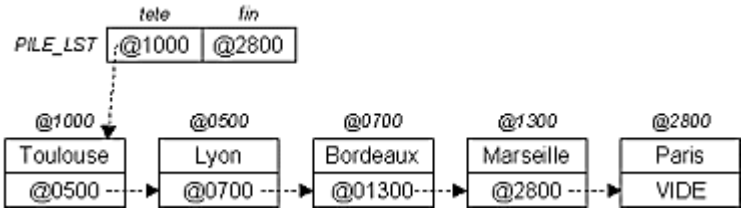
Introduction

Une pile est une structure qui stocke de manière ordonnée des éléments, mais rend accessible uniquement un seul d'entre eux, appelé le sommet de la pile. Quant on ajoute un élément, celui-ci devient le sommet de la pile, c'est-à-dire le seul élément accessible. Quant on retire un élément de la pile, on retire toujours le sommet, et le dernier élément ajouté avant lui devient alors le sommet de la pile. Pour résumer, le dernier élément ajouté dans la pile est le premier élément à en être retiré. Cette structure est également appelée une liste LIFO (*Last In, First Out*).

Généralement, il y a deux façons de représenter une pile. La première s'appuie sur la structure de liste chaînée vue précédemment. La seconde manière utilise un tableau.

Modélisation par liste chaînée

La première façon de modéliser une pile consiste à utiliser une liste chaînée en n'utilisant que les opérations ajouterTete et retirerTete. Dans ce cas, on s'aperçoit que le dernier élément entré est toujours le premier élément sorti. La figure suivante représente une pile par cette modélisation. La pile contient les chaînes de caractères suivantes qui ont été ajoutées dans cet ordre: "Paris", "Marseille", "Bordeaux", "Lyon" et "Toulouse".

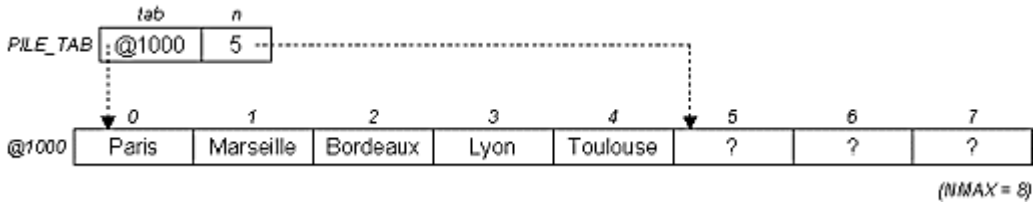


Pour cette modélisation, la structure d'une pile est celle d'une liste chaînée.

```
type PILE_LST: LISTE;
```

Modélisation par tableau

La deuxième manière de modéliser une pile consiste à utiliser un tableau. L'ajout d'un élément se fera à la suite du dernier élément du tableau. Le retrait d'un élément de la pile se fera en enlevant le dernier élément du tableau. La figure suivante représente une pile par cette modélisation. Elle reprend la même pile que pour l'exemple de modélisation par liste chaînée.



Voici la structure de données correspondant à une pile représentée par un tableau.

```
structure PILE_TAB:
 ELEMENT tab[NMAX];
 ENTIER  n;
fin structure;
```

NMAX est une constante représentant la taille du tableau alloué. n est le nombre d'éléments dans la pile.

OPERATIONS SUR LA STRUCTURE

Introduction

A partir de maintenant, nous allons employer le type PILE qui représente une pile au sens général, c'est-à-dire sans se soucier de sa modélisation. PILE représente aussi bien une pile par liste chaînée (PILE_LST) qu'une liste par pointeurs (PILE_PTR). Voici les opérations que nous allons détailler pour ces deux modélisations.

- `initialiserPile(PILE * p)`
  Initialise la pile pointée par p, i.e. fait en sorte que la pile soit vide.

- `BOOLEEN ← pileVide(PILE p)`
  Indique si la pile p est vide.

- `ELEMENT ← sommet(PILE p)`
  Retourne l'élément au sommet de la pile p.

- `BOOLEEN ← empilerElement(PILE * p, ELEMENT e)`
  Empile l'élément e au sommet de la pile pointée par p. VRAI est retournée si l'opération a réussi.

- `BOOLEEN ← depilerElement(PILE * p, ELEMENT * e)`
  Dépile et copie à l'adresse e l'élément au sommet de la pile pointée par p. VRAI est retournée si l'opération a réussi.

Les prototypes de ces opérations (paramètres et type de retour) sont les mêmes quelque soit la modélisation choisie.

Opérations pour la modélisation par liste chaînée

Initialiser la pile

Cette fonction initialise les valeurs de la structure représentant la pile pointée par p, afin que celle-ci soit vide. Dans le cas d'une représentation par liste chaînée, il suffit d'initialiser la liste chaînée qui représente la pile.

```
fonction initialiserPile(PILE * p):
 initialiserListe(p);
fin fonction;
```

Pile vide ?

Cette fonction indique si la pile p est vide. Dans le cas d'une représentation par liste chaînée, la pile est vide si la liste qui la représente est vide.

```
fonction BOOLEEN ← pileVide(PILE p):
 rendre listeVide(p);
fin fonction;
```

Sommet d'une pile

Cette fonction retourne l'élément au sommet de la pile p. Dans le cas d'une représentation par liste chaînée, cela revient à retourner la valeur de l'élément en tête de la liste. A n'utiliser que si la pile p n'est pas vide.

```
fonction ELEMENT ← sommet(PILE p):
 si (non pileVide(p)) alors
  rendre teteListe(p);
 sinon
  /* Erreur */
 fin si;
fin fonction;
```

Empiler un élément sur une pile

Cette fonction empile l'élément e au sommet de la pile pointée par p. Pour la représentation par liste chaînée, cela revient à ajouter l'élément e en tête de la liste. VRAI est retournée si l'élément a bien été ajouté.

```
fonction BOOLEEN ← empilerElement(PILE * p, ELEMENT e):
 rendre ajouterTete(p,e);
fin fonction;
```

Dépiler un élément d'une pile

Cette fonction dépile l'élément au sommet de la pile pointée par p et stocke sa valeur à l'adresse e. Pour la représentation par liste chaînée, cela revient à récupérer la valeur de l'élément en tête de liste avant de le supprimer de cette dernière. VRAI est retournée si la pile n'est pas déjà vide.

```
fonction BOOLEEN ← depilerElement(PILE * p, ELEMENT * e):
 si (non pileVide(*p)) alors
  *e ← sommet(*p);
  rendre retirerTete(p);
 sinon
```

```
  rendre FAUX;
 fin si;
fin fonction;
```

Opérations pour la modélisation par tableau

Initialiser la pile

Cette fonction initialise les valeurs de la structure représentant la pile pointée par p, afin que celle-ci soit vide. Dans le cas d'une représentation par tableau, il suffit de rendre n nul.

```
fonction initialiserPile(PILE * p):
 p→n ← 0;
fin fonction;
```

Pile vide ?

Cette fonction indique si la pile p est vide. Dans le cas d'une représentation par tableau, la pile est vide si le champ n est nul.

```
fonction BOOLEEN ← pileVide(PILE p):
 rendre (p.n = 0);
fin fonction;
```

Sommet d'une pile

Cette fonction retourne l'élément au sommet de la pile p. Dans le cas d'une représentation par tableau, cela revient à retourner la valeur du $n^{ième}$ élément du tableau (i.e. l'élément d'indice n - 1). A n'utiliser que si la pile p n'est pas vide.

```
fonction ELEMENT ← sommet(PILE p):
 si (non pileVide(p)) alors
  rendre (p.tab[p.n - 1]);
 sinon
  /* Erreur */
 fin si;
fin fonction;
```

Empiler un élément sur une pile

Cette fonction empile l'élément e au sommet de la pile pointée par p. Pour la représentation par tableau, cela revient à ajouter l'élément e à la fin du tableau. S'il reste de la place dans l'espace réservé au tableau, la fonction retourne VRAI.

```
fonction BOOLEEN ← empilerElement(PILE * p, ELEMENT e):
 si (p→n = NMAX) alors
  rendre FAUX;
 sinon
  p→tab[p→n] ← e;
  p→n ← p→n + 1;
  rendre VRAI;
 fin si;
```

```
fin fonction;
```

Dépiler un élément d'une pile

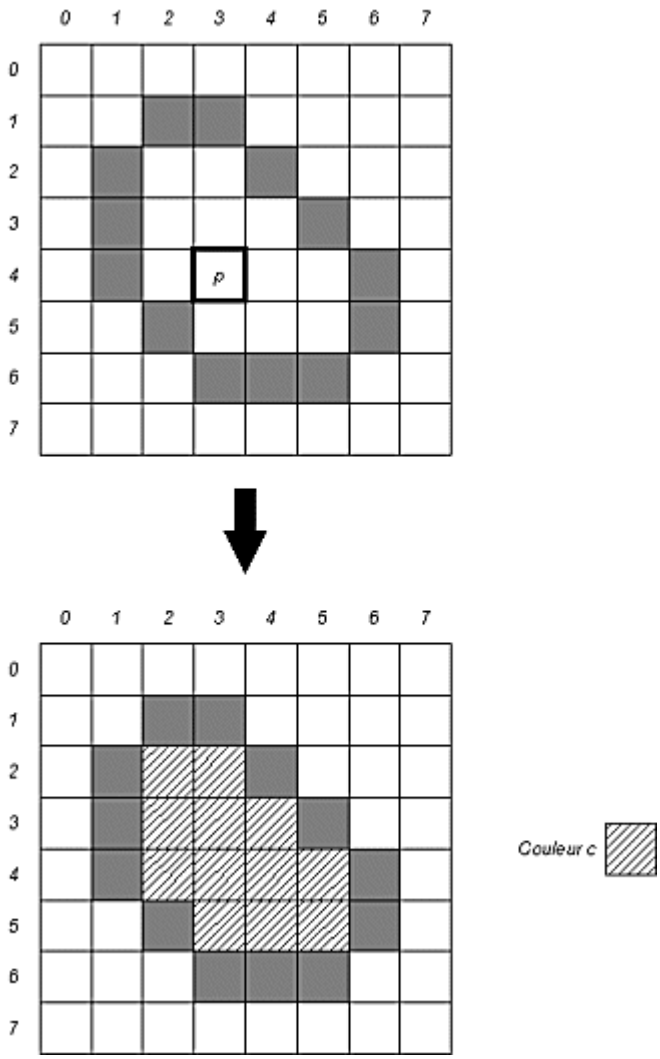
Cette fonction dépile l'élément au sommet de la pile pointée par p et stocke sa valeur à l'adresse e. Pour la représentation par tableau, cela revient à diminuer d'une unité le champ n, et à renvoyer l'élément d'indice n du tableau. Si la pile n'est pas déjà vide, VRAI est renvoyée.

```
fonction BOOLEEN ← depilerElement(PILE * p, ELEMENT * e)
 si (non pileVide(*p)) alors
  *e ← sommet(*p);
  p→n ← p→n - 1;
  rendre VRAI;
 sinon
  rendre FAUX;
 fin si;
fin fonction;
```

EXEMPLE: SUPPRESSION DE LA RECURSIVITE

Présentation

Une image en informatique peut être représentée par une matrice de points m ayant XMAX colonnes et YMAX lignes. Un élément m[x][y] de la matrice représente la couleur du point p de coordonnées (x;y). On propose d'écrire ici une fonction qui, à partir d'un point p, "étale" une couleur c autour de ce point. La progression de la couleur étalée s'arrête quand elle rencontre une couleur autre que celle du point p. La figure suivante illustre cet exemple, en considérent p = (3;4).



On propose ici deux façon d'écrire cette fonction. La première solution est récursive. La seconde reprend la première en éliminant la récursivité à l'aide d'une pile. Par la suite, COULEUR sera le type qui représente une couleur, et POINT sera la structure qui représente un point.

```
structure POINT:
 ENTIER x;
 ENTIER y;
fin fonction;
```

Implémentation récursive

La manière récursive consiste à écrire une fonction qui change la couleur d'un point p en une couleur c2 si sa couleur originale vaut c1. Dans le même temps, cette fonction s'appelle elle-même pour les quatres points adjacents à p. Bien entendu, il faut prendre garde de ne pas sortir des limites de la matrice.

```
fonction remplirR(IMAGE i, POINT p, COULEUR c1, COULEUR c2):
 si (i[p.x][p.y] = c1) alors
  i[p.x][p.y] ← c2;
  si (p.x > 0) alors remplirR(i,(p.x - 1;p.y),c1,c2);
  si (p.x < XMAX) alors remplirR(i,(p.x + 1;p.y),c1,c2);
  si (p.y > 0) alors remplirR(i,(p.x;p.y - 1),c1,c2);
```

```
  si (p.y < YMAX) alors remplirR(i,(p.x;p.y + 1),c1,c2);
 fin si;
fin fonction;
```

Pour réaliser l'exemple présenté en introduction, la fonction récursive s'utilisera de la manière suivante.

```
remplirR(m,(3;4),m[3][4],c);
```

Implémentation itérative (non récursive)

Quant on appelle une fonction récursivement, il y a "empilement" des appels à cette fonction. En effet, chaque fois qu'une fonction est lancée, on empile le contexte de la fonction appelante de manière à pouvoir y revenir ensuite. Lorsque la fonction se termine, on dépile le dernier contexte empilé et on reprend l'exécution de la fonction appelante.

L'idée ici est d'identifier les données qui définissent le contexte de la fonction et de les empiler de manière explicite. Ensuite, itérativement, tant que la pile n'est pas vide, on applique le corps de la fonction récursive sur les données dépilées du sommet de la pile. Un appel initialement récursif à la fonction se traduira alors par l'empilement de nouvelles données sur la pile.

Ici, la donnée cruciale, c'est le point de la matrice que l'on est en train de traiter. On va donc empiler les coordonnées des différents points traités pour reproduire un comportement similaire à l'implémentation récursive qu'est la fonction remplirR.

```
fonction remplirI(IMAGE i, POINT p, COULEUR c2):
 c1 ← i[p.x][p.y];
 initialiserPile(&pl);
 empilerElement(&pl,p);

 tant que (non pileVide(pl)) faire
  depilerElement(&pl,&p);

  si (i[p.x][p.y] = c1) alors
   i[p.x][p.y] ← c2;
   si (p.x > 0) alors empilerElement(&pl,(p.x - 1;p.y))
   si (p.x < XMAX) alors empilerElement(&pl,(p.x + 1;p.y));
   si (p.y > 0) alors empilerElement(&pl,(p.x;p.y - 1));
   si (p.y < YMAX) alors empilerElement(&pl,(p.x;p.y + 1));
  fin si;
 fin tant que;
fin fonction;
```

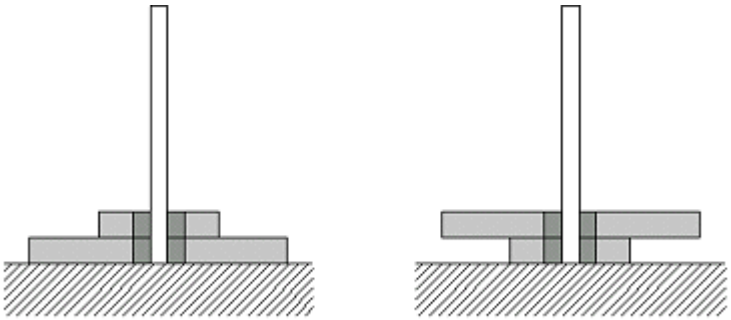
Pour réaliser l'exemple présenté en introduction, la fonction itérative s'utilisera de la manière suivante.

```
remplirI(m,(3;4),c);
```

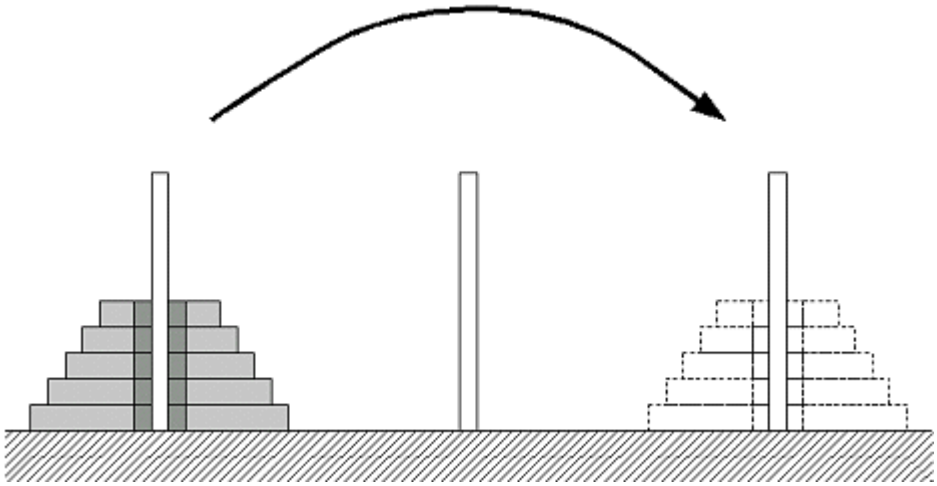
UN AUTRE EXEMPLE: LA TOUR DE HANOI

Présentation

Il s'agit d'un jeu de réflexion dont voici la règle. Des anneaux de diamètres différents sont empilés sur un poteau. Un anneau peut être empilé sur un autre seulement si il a un diamètre inférieur à celui de l'anneau sur lequel il repose.

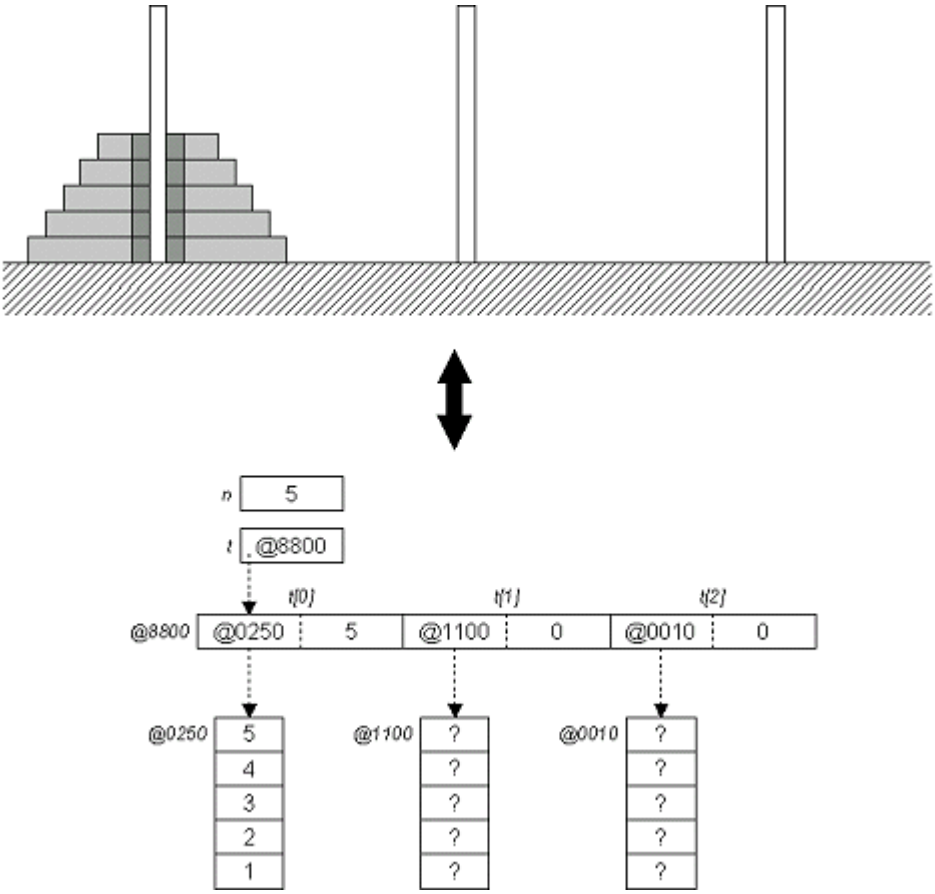


Le but du jeu est de déplacer les anneaux initialement empilés sur un seul poteau vers un autre en respectant les règles et en n'utilisant qu'un seul poteau intermédiaire.

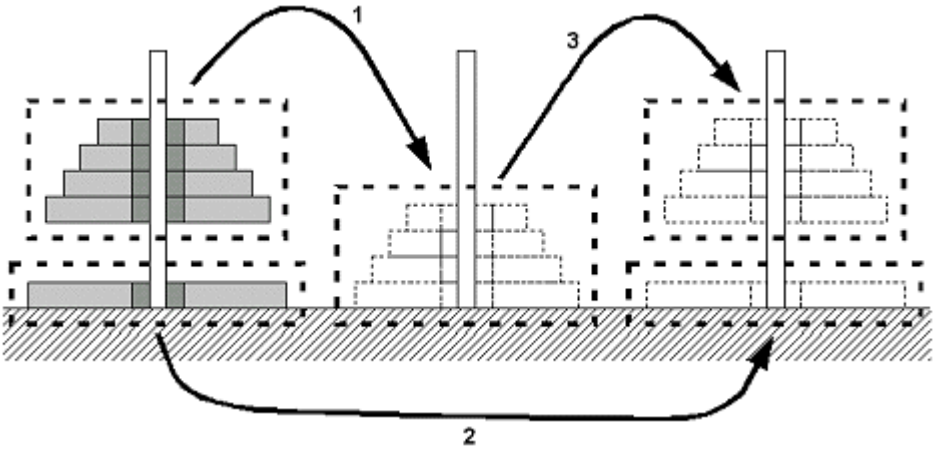


Solution

Les poteaux sont représentés par des piles d'entiers numérotées 0, 1 et 2. Le jeu entier sera alors représenté par un tableau t de trois piles. On suppose qu'il y a n anneaux dans le jeu. On attribue à chaque anneau un numéro de manière à ce que si l'anneau a est plus petit que l'anneau b, alors son numéro est plus petit. Au départ, on aura donc:



Voici maintenant la fonction qui réalise le déplacement des anneaux du poteau numéro a au poteau numéro b (on déduit le numéro du poteau intermédiaire c par la formule c = 3 - a - b). Cette fonction est récursive. S'il y a un seul anneau, on le déplace directement, sinon on déplace les n - 1 premiers anneaux sur le poteau intermédiaire c (appel récursif à la fonction) et le dernier anneau sur le poteau final b (appel récursif à la fonction). Ensuite, on déplace les anneaux du poteau intermédiaire c sur le poteau final a (appel récursif à la fonction). Voici une illustration.



Et voici maintenant le détail de la fonction qui déplace les anneaux.

```
fonction deplacerAnneaux(PILE * t, ENTIER n,
                         ENTIER a, ENTIER b):
 si (n = 1) alors
  depilerElement(&t[a],&e);
  empilerElement(&t[b],e);
 sinon
```

```
  c ← 3 - a - b; /* Un moyen de trouver le numero du
                    poteau intermediaire. */
  deplacerAnneaux(t,n - 1,a,c);
  deplacerAnneaux(t,1,a,b);
  deplacerAnneaux(t,n - 1,c,b);
 fin si;
fin fonction;
```

CONCLUSION

Comme la pile ne permet l'accès qu'à un seul de ses éléments, son usage est limité. Cependant, elle peut être très utile pour supprimer la récursivité d'une fonction comme nous l'avons vu précédemment. La différence entre la modélisation par liste chaînée et la modélisation par tableau est très faible. L'inconvénient du tableau est que sa taille est fixée à l'avance, contrairement à la liste chaînée qui n'est limitée que par la taille de la mémoire centrale de l'ordinateur. En contrepartie, la liste chaînée effectue une allocation dynamique de mémoire à chaque ajout d'élément et une libération de mémoire à chaque retrait du sommet de la pile. En résumé, la modélisation par liste chaînée sera un peu plus lente, mais plus souple quant à sa taille.

4. FILES D'ATTENTE

MODELISATION DE LA STRUCTURE

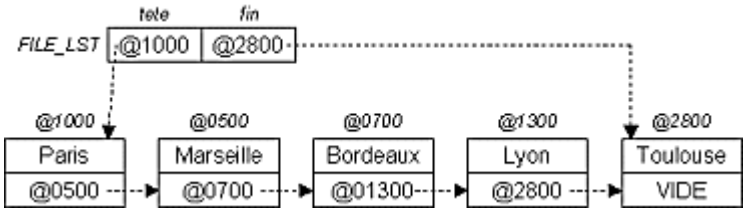
Introduction

Une file d'attente est une structure qui stocke de manière ordonnée des éléments, mais rend accessible uniquement un seul d'entre eux, appelé la tête de la file. Quant on ajoute un élément, celui-ci devient le dernier élément qui sera accessible. Quant on retire un élément de la file, on retire toujours la tête, celle-ci étant le premier élément qui a été placé dans la file. Pour résumer, le premier élément ajouté dans la pile est le premier élément à en être retiré. Cette structure est également appelée une liste FIFO (*First In, First Out*).

Généralement, il y a deux façons de représenter une file d'attente. La première s'appuie sur la structure de liste chaînée vue précédemment. La seconde manière utilise un tableau d'une façon assez particulière que l'on appelle modélisation par "tableau circulaire".

Modélisation par liste chaînée

La première façon de modéliser une file d'attente consiste à utiliser une liste chaînée en n'utilisant que les opérations ajouterQueue et retirerTete. Dans ce cas, on s'aperçoit que le premier élément entré est toujours le premier élément sorti. La figure suivante représente une file d'attente par cette modélisation. La file contient les chaînes de caractères suivantes qui ont été ajoutées dans cet ordre: "Paris", "Marseille", "Bordeaux", "Lyon" et "Toulouse".

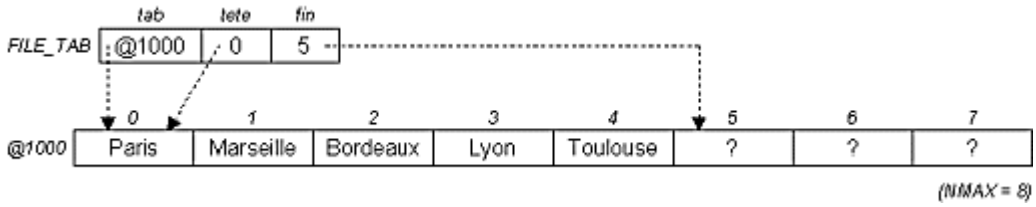


Pour cette modélisation, la structure d'une file d'attente est celle d'une liste chaînée.

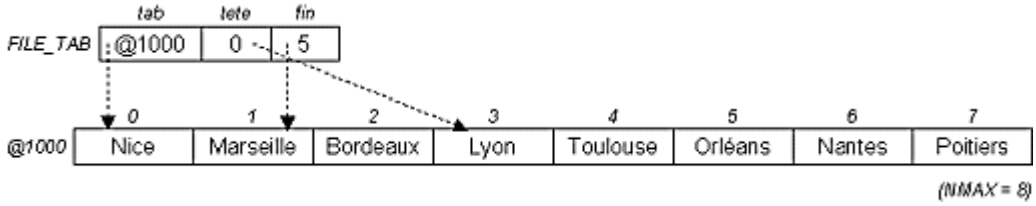
```
type FILE_LST: LISTE;
```

Modélisation par tableau circulaire

La deuxième manière de modéliser une file d'attente consiste à utiliser un tableau. L'ajout d'un élément se fera à la suite du dernier élément du tableau. Le retrait d'un élément de la file se fera en enlevant le premier élément du tableau. Il faudra donc deux indices pour ce tableau, le premier qui indique le premier élément de la file et le deuxième qui indique la fin de la file. La figure suivante représente une file d'attente par cette modélisation. Elle reprend la même pile que pour l'exemple de modélisation par liste chaînée.



On peut noter que progressivement, au cours des opérations d'ajout et de retrait, le tableau se déplace sur la droite dans son espace mémoire. A un moment, il va en atteindre le bout de l'espace. Dans ce cas, le tableau continuera au début de l'espace mémoire comme si la première et la dernière case étaient adjacentes, d'où le terme "tableau circulaire". Ce mécanisme fonctionnera tant que le tableau n'est effectivement pas plein. Pour illustrer, reprenons l'exemple précédent auquel on applique trois opérations de retrait de l'élément de tête. On ajoute également en queue les éléments suivants et dans cet ordre: "Orléans", "Nantes", "Poitiers", "Nice".



La file d'attente contient alors, et dans cet ordre: "Lyon", "Toulouse", "Orléans", "Nantes", "Poitiers" et "Nice". On s'aperçoit que, ayant atteint la fin du tableau, la file redémarre à l'indice 0.

Voici la structure de données correspondant à une file d'attente représentée par un tableau circulaire.

```
structure FILE_TAB:
 ELEMENT tab[NMAX];
 ENTIER  tete;
 ENTIER  fin;
fin structure;
```

NMAX est une constante représentant la taille du tableau alloué. tete est l'indice de tableau qui pointe sur la tête de la file d'attente. fin est l'indice de tableau qui pointe sur la case suivant le dernier élément du tableau, c'est-à-dire la prochaine case libre du tableau.

OPERATIONS SUR LA STRUCTURE

Introduction

A partir de maintenant, nous allons employer le type FILE qui représente une file d'attente au sens général, c'est-à-dire sans se soucier de sa modélisation. FILE représente aussi bien une file d'attente par liste chaînée (FILE_LST) qu'une file d'attente par tableau circulaire (FILE_TAB). Voici les opérations que nous allons détailler pour ces deux modélisations.

- `initialiserFile(FILE * f)`
  Initialise la file pointée par f, i.e. fait en sorte que la file soit vide.

- `BOOLEEN ← fileVide(FILE f)`
  Indique si la file f est vide.

- `ELEMENT ← teteFile(FILE f)`
  Retourne l'élément en tête de la file f.

- `BOOLEEN ← entrerElement(FILE * f, ELEMENT e)`
  Entre l'élément e dans la file pointée par f.

- `BOOLEEN ← sortirElement(FILE * f, ELEMENT * e)`
  Sort et copie à l'adresse e l'élément en tête de la file pointée par f.

Les prototypes de ces opérations (paramètres et type de retour) sont les mêmes quelque soit la modélisation choisie.

Opérations pour la modélisation par liste chaînée

Initialiser une file

Cette fonction initialise les valeurs de la structure représentant la file pointée par f pour que celle-ci soit vide. Dans le cas d'une représentation par liste chaînée, il suffit d'initialiser la liste chaînée qui représente la file d'attente.

```
fonction initialiserFile(FILE * f):
 initialiserListe(f);
fin fonction;
```

File vide ?

Cette fonction indique si la file f est vide. Dans le cas d'une représentation par liste chaînée, la file est vide si la liste qui la représente est vide.

```
fonction BOOLEEN ← fileVide(FILE f):
 rendre listeVide(f);
fin fonction;
```

Tête d'une file

Cette fonction retourne l'élément en tête de la file f. Dans le cas d'une représentation par liste chaînée, cela revient à retourner la valeur de l'élément en tête de la liste. A n'utiliser que si la file f n'est pas vide.

```
ELEMENT ← teteFile(FILE f):
 si (non fileVide(f)) alors
  rendre teteListe(f);
 sinon
  /* Erreur */
 fin si;
fin fonction;
```

Entrer un élément dans une file

Cette fonction place un élément e en queue de la file pointée par f. Pour la représentation par liste chaînée, cela revient à ajouter l'élément e en queue de la liste. VRAI est retournée si l'ajout a bien pu se faire.

```
fonction BOOLEEN ← entrerElement(FILE * f, ELEMENT e):
 rendre ajouterQueue(f,e);
fin fonction;
```

Sortir un élément d'une file

Cette fonction retire l'élément en tête de la file pointée par f et stocke sa valeur à l'adresse e. Pour la représentation par liste chaînée, cela revient à récupérer la valeur de l'élément en tête de liste avant de le supprimer de cette dernière. Si la file n'est pas déjà vide, VRAI est retournée.

```
fonction BOOLEEN ← sortirElement(FILE * f, ELEMENT * e):
 si (non fileVide(*f)) alors
  *e ← teteFile(*f);
  rendre retirerTete(f);
 sinon
  rendre FAUX;
 fin si;
fin fonction;
```

Opérations pour la modélisation par tableau circulaire

Initialiser une file

Cette fonction initialise les valeurs de la structure représentant la file pointée par f pour que celle-ci soit vide. Dans le cas d'une représentation par tableau circulaire, on choisira de considérer la file vide lorsque f→tete = f→fin. Arbitrairement, on choisit ici de mettre ces deux indices à 0 pour initialiser une file d'attente vide.

```
fonction initialiserFile(FILE * f):
 f→tete ← 0;
 f→fin ← 0;
fin fonction;
```

File vide ?

Cette fonction indique si la file f est vide. Dans le cas d'une représentation par tableau circulaire, la file est vide lorsque f.tete = f.fin.

```
fonction BOOLEEN ← fileVide(FILE f):
 rendre (f.tete = f.fin);
fin fonction;
```

Tête d'une file

Cette fonction retourne l'élément en tête de la file f. Dans le cas d'une représentation par tableau circulaire, il suffit de retourner l'élément pointé par l'indice de tête dans le tableau. A n'utiliser que si la file f n'est pas vide.

```
ELEMENT ← teteFile(FILE f):
 si (non fileVide(f)) alors
  rendre (f.tab[f.tete]);
 sinon
  /* Erreur */
 fin si;
fin fonction;
```

Entrer un élément dans une file

Cette fonction place un élément e en queue de la file f. Pour la représentation par tableau circulaire, l'élément est placé dans la case pointée par l'indice fin. Ce dernier est ensuite augmenté d'une unité, en tenant compte du fait qu'il faut revenir à la première case du tableau si il a atteint la fin de celui-ci. D'où l'utilisation de l'opérateur mod qui retourne le reste de la division entre ses deux membres. Si le tableau n'est pas plein au moment de l'insertion, VRAI est retournée.

```
fonction BOOLEEN ← entrerElement(FILE * f, ELEMENT e):
 si ((f→fin + 1) mod NMAX = f→tete) alors rendre FAUX;
 f→tab[f→fin] ← e;
 f→fin ← (f→fin + 1) mod NMAX;
 rendre VRAI;
fin fonction;
```

On détecte que le tableau est plein si l'indice fin est juste une case avant l'indice tete. En effet, si on ajoutait une case, fin deviendrait égal à tete, ce qui est notre configuration d'une file vide. La taille maximale de la file est donc NMAX - 1, une case du tableau restant toujours inutilisée.

Sortir un élément d'une file

Cette fonction retire l'élément en tête de la file f et stocke sa valeur à l'adresse e. Pour la représentation par tableau circulaire, cela revient à récupérer la valeur de l'élément en tête de file avant de le supprimer en augmentant l'indice tete d'une unité. Il ne faut pas oublier de ramener tete à la première case du tableau au cas où il a atteint la fin de ce dernier. VRAI est retournée s'il restait au moins un élément dans la file.

```
fonction BOOLEEN ← sortirElement(FILE * f, ELEMENT * e):
 si (fileVide(*f)) alors rendre FAUX;
 *e ← teteFile(*f);
 f→tete ← (f→tete + 1) mod NMAX;
 rendre VRAI;
fin fonction;
```

CONCLUSION

Ce genre de structure de données est très utilisée par les mécanismes d'attente. C'est le cas notamment d'une imprimante en réseau, où les tâches d'impressions arrivent aléatoirement de

n'importe quel ordinateur connecté. Les tâches sont placées dans une file d'attente, ce qui permet de les traiter selon leur ordre d'arrivée. Les remarques concernant les différences la modélisation par liste chaînée et la modélisation par tableau circulaire sont les mêmes que pour la structure de file. Très peu de différences sont constatées. En résumé, la modélisation par liste chaînée sera un peu plus lente, mais plus souple quant à sa taille.

---

5. ARBRES BINAIRES

MODELISATION DE LA STRUCTURE

Introduction

Les tableaux sont des structures qui permettent un accès direct à un élément à partir de son indice. Par contre, l'insertion ou la suppression dans de telles structures d'un élément à une position donnée sont des opérations coûteuses. D'un autre côté, les listes chaînées facilitent les actions d'insertion et de suppression d'un élément, mais ne permettent pas l'accès direct à un élément.

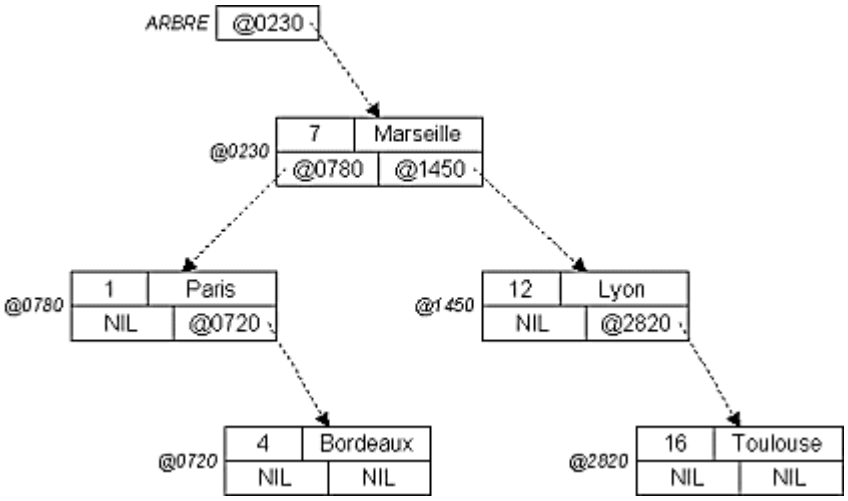
Les arbres binaires présentés ici sont en fait un compromis entre ces deux structures. Ils permettent un accès "relativement" rapide à un élément à partir de son identifiant, appelé ici une "clé". Dans les arbres binaires, l'ajout et la suppression sont également des opérations "relativement" rapides.

Par "relativement" rapide, on entend que le temps d'exécution d'une opération n'est pas proportionnel au nombre d'éléments dans la structure. En effet, dans un tableau non ordonné de 256 éléments par exemple, il faut parcourir au plus les 256 éléments pour trouver celui que l'on cherche. Comme vous le constaterez par la suite, dans un arbre binaire de 256 éléments, il suffit de parcourir au plus 8 éléments pour trouver celui que l'on cherche.

De la même manière qu'une liste chaînée, un arbre est constitué de maillons, appelés ici "noeuds", la différence étant qu'un noeud n'a pas un successeur mais deux. Ceux-ci sont désignés comme étant le "fils gauche" et le "fils droit" du noeud. Un noeud est donc une structure qui contient un élément à stocker et deux pointeurs sur les noeuds fils.

```
structure NOEUD:
 ELEMENT elt;
 NOEUD * fg;
 NOEUD * fd;
fin structure;
```

Dans la suite de ce chapitre, on supposera que l'identifiant d'un élément, que l'on a appelé la clé, est un objet de type CLE. On disposera aussi de la fonction cle, cle(e) retournant la clé de l'élément e. On supposera également qu'un arbre ne peut pas contenir deux éléments ayant la même clé. La clé est donc un moyen unique d'identifier et de localiser un élément dans un arbre. La figure qui suit est l'exemple d'un arbre qui contient les éléments suivants: (1;"Paris"), (4;"Bordeaux"), (7,"Marseille"), (12,"Lyon"), (16,"Toulouse").



Ici, la clé des éléments c'est le chiffre avant la chaîne de caractères. Ainsi, cle ((1;"Paris")) = 1. On peut remarquer que les éléments ne sont pas placés n'importe comment dans l'arbre. En effet, un arbre sera toujours fait de telle sorte que les éléments dans le sous-arbre gauche d'un noeud ont une clé inférieure à celle du noeud. De même, les éléments dans le sous-arbre droit d'un noeud ont une clé supérieure à celle du noeud. Ce qui facilite par la suite la recherche d'un élément par rapport à sa clé.

Le noeud au sommet de l'arbre est appelé "racine" et un arbre sera référencé simplement par un pointeur sur cette racine, de la même manière qu'une liste chaînée est référencée par un pointeur sur la tête de la liste.

```
type ARBRE: NOEUD *;
```

OPERATIONS SUR LA STRUCTURE

Introduction

Nous allons présenter ici quelques opérations classiques que l'on peut effectuer sur un arbre. On a choisi de les écrire de manière récursive car c'est la manière la plus simple d'aborder une telle structure de données. Voici maintenant une brève description des opérations détaillées dans cette section.

- `initialiserArbre(ARBRE * a)`
  Initialise l'arbre pointé par a, i.e. fait en sorte que l'arbre soit vide.

- `NOEUD * ← preparerNoeud(ELEMENT e)`
  Alloue un noeud et y place l'élément e. Si aucun noeud n'a pu être alloué, la valeur NIL est retournée.

- `BOOLEEN ← ajouterNoeud(ARBRE * a, NOEUD * n)`
  Insère le noeud pointé par n dans l'arbre pointé par a. Si un noeud avec la même clé existe déjà, l'insertion est annulée et la fonction retourne FAUX.

- `BOOLEEN ← ajouterElement(ARBRE * a, ELEMENT e)`
  Insère un élément e dans l'arbre pointé par a. La fonction renvoie VRAI si l'opération a réussi.

- `BOOLEEN ← existeCle(ARBRE a, CLE c)`
  Indique si un élément avec la clé c est présent dans l'arbre a.

- `NOEUD * ← extraireMaximum(ARBRE * a)`
  Extrait le noeud de l'arbre pointé par a ayant la plus grande clé, i.e. enlève ce noeud de l'arbre et retourne son adresse. NIL est retournée s'il n'y a aucun noeud à extraire.

- `BOOLEEN ← supprimerRacine(ARBRE * a)`
  Supprime le noeud à la racine de l'arbre pointé par a. FAUX est retournée s'il n'y a aucun noeud à supprimer.

- `BOOLEEN ← extraireElement(ARBRE * a, CLE c, ELEMENT * e)`
  Extrait l'élément ayant la clé c de l'arbre pointé par a, i.e. le noeud contenant l'élément est supprimé de l'arbre et l'élément est copié à l'adresse pointée par e. VRAI est retournée si l'élément de clé c a été trouvé.

Préparation

Initialiser un arbre

Cette fonction initialise les valeurs de la structure représentant l'arbre pointé par a, afin que celui-ci soit vide. Il suffit de mettre le pointeur sur la racine égal à NIL.

```
fonction initialiser(ARBRE * a):
 *a ← NIL;
fin fonction;
```

Préparer un noeud

Cette fonction alloue un nouveau noeud et place l'élément e à l'intérieur. Ses deux fils sont initialisés à la valeur NIL. La fonction retourne l'adresse de ce nouveau noeud. Au cas où l'allocation de mémoire échoue, NIL est renvoyée.

```
fonction NOEUD * ← preparerNoeud(ELEMENT e):
 n ← ALLOUER(NOEUD,1);

 si (n != NIL) alors
  n→elt ← e;
  n→fg ← NIL;
  n→fd ← NIL;
 fin si;

 rendre n;
fin fonction;
```

Ajout

Ajouter un noeud

Cette fonction ajoute le noeud pointé par n dans l'arbre pointé par a. Pour cela, l'arbre est parcouru à partir de la racine pour descendre jusqu'à l'endroit où sera inséré le noeud. A

chaque noeud, deux possibilités sont offertes pour descendre. On prendra à gauche si la clé du noeud visité est supérieure à la clé du noeud à insérer. A l'opposé, on prendra à droite si la clé du noeud visité est inférieure. En cas d'égalité, l'insertion ne peut pas se faire et la fonction retourne FAUX. On arrête la descente quand le fils gauche ou droit choisi pour descendre vaut NIL. Le noeud pointé par n est alors inséré à ce niveau.

```
fonction BOOLEEN ← ajouterNoeud(ARBRE * a, NOEUD * n):
 si (*a = NIL) alors
  *a ← n;
  rendre VRAI;
 fin si;

 si (cle(n→elt) = cle((*a)→elt)) alors rendre FAUX;

 si (cle(n→elt) < cle((*a)→elt)) alors
  rendre ajouterNoeud(&((*a)→fg),n);
 fin si;

 rendre ajouterNoeud(&((*a)→fd),n);
fin fonction;
```

Ajouter un élément

Cette fonction ajoute l'élément e dans l'arbre pointé par a. Pour cela, un noeud est préparé puis ajouté dans l'arbre. Si le noeud ne peut pas être alloué ou si la clé de l'élément est déjà présente dans l'arbre, alors la valeur FAUX est retournée.

```
fonction BOOLEEN ← ajouterElement(ARBRE * a, ELEMENT e):
 n ← preparerNoeud(e);
 si (n = NIL) alors rendre FAUX;

 si (non ajouterNoeud(a,n)) alors
  LIBERER(n);
  rendre FAUX;
 fin si;

 rendre VRAI;
fin fonction;
```

Parcours / Recherche

Clé existe ?

Cette fonction cherche si un élément possède la clé c dans l'arbre a. Pour cela, l'arbre est parcouru à partir de la racine. Comme pour l'ajout d'un noeud, deux possibilités sont offertes quand on visite un noeud. Si on trouve l'élément qui a la clé c, alors VRAI est retournée. Sinon, on aboutit sur un fils (gauche ou droit) qui vaut NIL, ce qui signifie que la recherche est terminée et qu'aucun élément n'a la clé recherchée. FAUX est alors retournée.

```
BOOLEEN ← existeCle(ARBRE a, CLE c):
 si (a = NIL) alors rendre FAUX;
 si (c = cle(a→elt)) alors rendre VRAI;
 si (c < cle(a→elt)) alors rendre existeCle(a→fg,c);
 rendre existeCle(a→fd,c);
fin fonction;
```

Suppression

Extraire la clé maximum

Cette fonction extrait le noeud qui a la plus grande clé dans l'arbre pointé par a. Pour cela, l'arbre est parcouru en descendant sur la droite tant que cela est possible. Le dernier noeud visité est celui recherché. Il est supprimé de l'arbre et son adresse est retournée. Au cas où l'arbre est vide, la valeur NIL est retournée.

```
fonction NOEUD * ← extraireMaximum(ARBRE * a):
 si (*a = NIL) alors rendre NIL;

 si ((*a)→fd = NIL) alors
  n ← *a;
  *a ← (*a)→fg;
  rendre n;
 fin si;

 rendre extraireMaximum(&((*a)→fd));
fin fonction;
```

Supprimer la racine

Cette fonction supprime le noeud à la racine de l'arbre pointé par a. Quatre cas se présentent. Si l'arbre est vide, FAUX est retournée. Si la racine n'a pas de fils gauche, alors la racine est supprimée et le fils droit prend sa place. Si la racine n'a pas de fils droit, alors la racine est supprimée et le fils gauche prend sa place. Enfin, si la racine a deux fils, alors la racine est supprimée et le noeud ayant la plus grande clé dans le fils gauche prend sa place. Dans les trois derniers cas, VRAI est retournée.

```
fonction BOOLEEN ← supprimerRacine(ARBRE * a):
 si (*a = NIL) alors rendre FAUX;
 n ← *a;

 si (n→fg = NIL) alors *a ← n→fd;
 sinon si (n→fd = NIL) alors *a ← n→fg;
 sinon
  *a ← extraireMaximum(&(n→fg));
  (*a)→fg ← n→fg;
  (*a)→fd ← n→fd;
 fin si;

 LIBERER(n);
 rendre VRAI;
fin fonction;
```

Extraire un élément

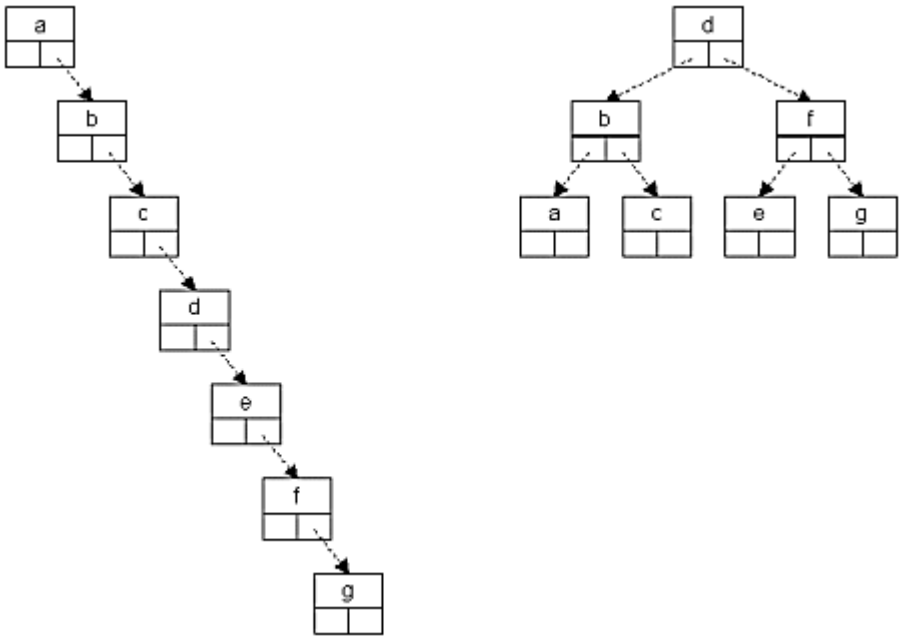
Cette fonction extrait l'élément ayant la clé c de l'arbre pointé par a. L'élément extrait est stocké à l'adresse e. A partir de la racine, on parcours l'arbre jusqu'à trouver la clé c. A ce moment, l'élément du noeud trouvé est stocké à l'adresse e et le noeud est supprimé en appliquant la fonction supprimerRacine sur le sous-arbre dont la racine est le noeud en question.

```
fonction BOOLEEN ← extraireElement(ARBRE * a, CLE c,
                                   ELEMENT * e):
 si (*a = NIL) alors rendre FAUX;

 si (c < cle((*a)→elt)) alors
  rendre extraireElement(&((*a)→fg),c,e);
 fin si;

 si (c > cle((*a)→elt)) alors
  rendre extraireElement(&((*a)→fd),c,e);
 fin si;

 *e ← (*a)→elt;
 rendre supprimerRacine(a);
fin fonction;
```

EQUILIBRAGE DES ARBRES

Introduction

Très facilement, on peut se rendre compte que les arbres binaires tels qu'ils ont été présentés jusqu'à présent ont un inconvénient majeur. En effet, les opérations d'ajout et de suppression ne garantissent pas un certain équilibre de l'arbre, c'est-à-dire qu'il se peut qu'il y ait beaucoup plus de noeuds d'un côté que de l'autre. Cela signifie que la recherche d'une clé d'un côté sera plus lente qu'une recherche de l'autre côté. Pour illustrer cela, penchons-nous sur la figure suivante. Elle représente deux arbres. Celui de gauche résulte de l'ajout successif des clés a,b,c,d,e,f,g dans cet ordre, alors que celui de droite résulte de l'ajout des mêmes éléments dans l'ordre d,b,f,a,c,e,g.



On considérera par la suite qu'un arbre est équilibré si, pour chacun de ses noeuds, la différence entre la hauteur du sous-arbre gauche et la hauteur du sous-arbre droit est d'au plus une unité. Dans un premier temps, nous allons modifier la structure d'un noeud afin d'y intégrer des informations concernant l'équilibrage de l'arbre. Ensuite, nous présenterons des opérations appelées "rotations" qui permettent de rééquilibrer un arbre le cas échéant. Enfin, nous verrons à quelle occasion employer ces rotations et comment modifier les opérations

présentées en première partie pour qu'elles guarantissent l'équilibre d'un arbre à tout moment.

Modification sur la structure

Introduction

La structure de données présentée au début du chapitre est légèrement modifiée ici pour intégrer quelques informations concernant l'équilibrage de l'arbre. En fait, pour chaque noeud n, on rajoute un champ h qui indique la hauteur du sous-arbre de racine n, un arbre sans noeud ayant une hauteur égale à 0. Voici la nouvelle structure.

```
structure NOEUD:
 ELEMENT elt;
 NOEUD * fg;
 NOEUD * fd;
 ENTIER  h;
fin structure;
```

Concernant cette nouvelle information, nous présentons maintenant les deux fonctions suivantes.

- `majHauteur(NOEUD * n)`
  Met à jour le champ h du noeud pointé par n en se basant sur les hauteurs (supposées correctes) des deux fils.

- `ENTIER ← desequilibre(ARBRE a)`
  Mesure le déséquilibre de l'arbre a, i.e. retourne la différence entre la hauteur du sous-arbre gauche et celle du sous-arbre droit.

Mise à jour de la hauteur d'un noeud

Cette fonction met à jour le champ h du noeud pointé par n. L'actualisation est faite en se basant sur les champs h des fils du noeud. Si un fils vaut NIL, alors on considérera sa hauteur comme étant nulle. La hauteur du noeud pointé par n est donc la plus grande des deux hauteurs des fils augmentée d'une unité. On utilise l'instruction MAX qui retourne la plus grande des deux valeurs passées en paramètre.

```
fonction majHauteur(NOEUD * n):
 si (n→fg = NIL) alors hg ← 0;
 sinon hg ← n→fg→h;

 si (n→fd = NIL) alors hd ← 0;
 sinon hd ← n→fd→h;

 n→h ← MAX(hd,hg) + 1;
fin fonction;
```

Mesure du déséquilibre d'un arbre

Cette fonction retourne une valeur qui mesure le déséquilibre de l'arbre a. En fait, il s'agit de la différence entre la hauteur du sous-arbre gauche et celle du sous-arbre droit. Si un fils vaut NIL, alors on considérera sa hauteur comme étant nulle.

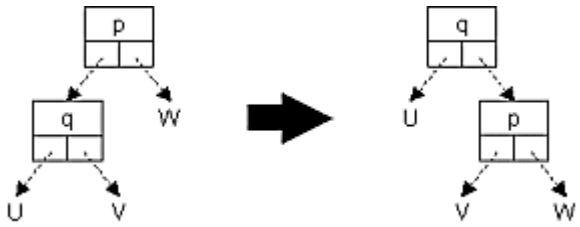
```
fonction ENTIER ← desequilibre(ARBRE a):
 si (a = NIL) alors rendre 0;

 si (a→fg = NIL) alors hg ← 0;
 sinon hg ← a→fg→h;

 si (a→fd = NIL) alors hd ← 0;
 sinon hd ← a→fd→h;

 rendre (hg - hd);
fin fonction;
```

Les rotations

Pour rééquilibrer un arbre, nous allons utiliser deux types de rotation. La première est la rotation simple dont on présente deux alternatives symétriques: la rotation simple à droite (notée RD) et la rotation simple à gauche (notée RG). La seconde est la rotation double qui est une succession de deux rotations simples. On en présente également deux alternatives symétriques: la rotation double gauche-droite (notée RGD) et la rotation double droite-gauche (RDG).

Rotation RD

La figure suivante illustre l'opération de rotation simple à droite.



En supposant que tous les noeuds impliqués dans la rotation existent, voici la fonction qui effectue cette rotation sans oublier de mettre à jour la hauteur des noeuds déplacés.

```
fonction rotationRD(ARBRE * a):
 n ← *a;
 si (n→fg = NIL) alors /* Erreur */
 *a ← n→fg;
 n→fg ← (*a)→fd;
 (*a)→fd ← n;
 majHauteur(n);
 majHauteur(*a);
fin fonction;
```

Rotation RG

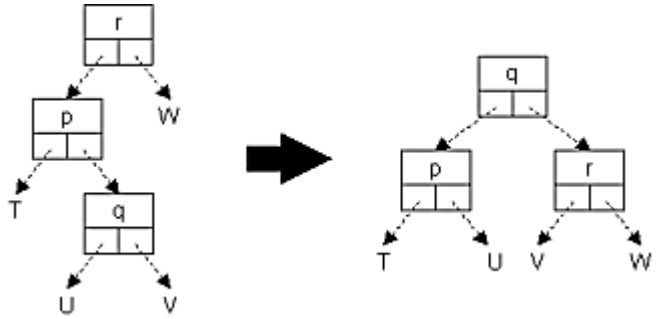
La rotation simple à gauche est symétrique à la rotation simple à droite. Les notions de gauche et de droite sont simplement inversées.

```
fonction rotationRG(ARBRE * a):
 n ← *a;
 si (n→fd = NIL) alors /* Erreur */
```

```
 *a ← n→fd;
 n→fd ← (*a)→fg;
 (*a)→fg ← n;
 majHauteur(n);
 majHauteur(*a);
fin fonction;
```

Rotation RGD

La figure suivante illustre l'opération de rotation double gauche-droite. Il s'agit d'appliquer une rotation RG sur le fils gauche de la racine, puis d'appliquer une rotation RD sur la racine elle-même.



En supposant que tous les noeuds impliqués dans la rotation existent, voici la fonction qui effectue cette rotation.

```
fonction rotationRGD(ARBRE * a):
 si ((*a)→fg = NIL) alors /* Erreur */
 rotationRG(&((*a)→fg));
 rotationRD(a);
fin fonction;
```

Rotation RDG

La rotation double droite-gauche est symétrique à la rotation double gauche-droite. Les notions de gauche et de droite sont simplement inversées.

```
fonction rotationRDG(ARBRE * a):
 si ((*a)→fd = NIL) alors /* Erreur */
 rotationRD(&((*a)→fd));
 rotationRG(a);
fin fonction;
```

Opérations d'ajout et de suppression avec rééquilibrage

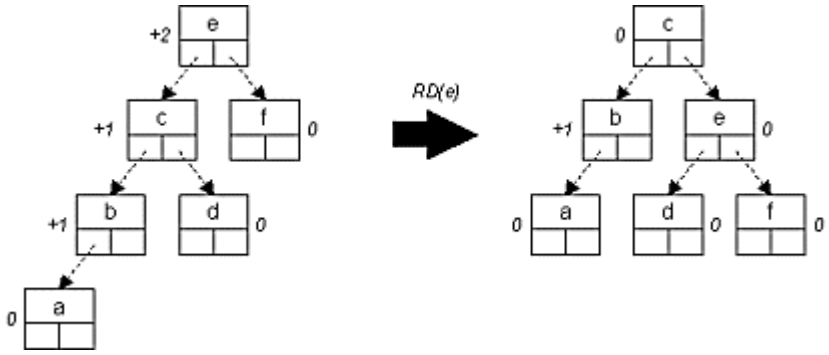
Les opérations d'ajout et de suppression d'un élément présentées ici guarantissent qu'un arbre reste toujours équilibré. L'idée principale est la suivante. On effectue tout d'abord l'opération d'ajout ou de suppression comme elle a été présentée précédemment. Seulement, on mémorise le chemin qui nous a permis de trouver la position de l'élément inséré ou supprimé. Ensuite, on remonte ce chemin jusqu'à la racine. A chaque noeud visité, on regarde s'il y a un déséquilibre de plus d'une unité. Si c'est le cas, un rééquilibrage s'impose en effectuant une des quatre rotations présentées précédemment.

Dans un premier temps, on présente la fonction de rééquilibrage qui effectue l'une des quatres rotations sur la racine d'un arbre dans le but de le rééquilibrer. Ensuite, on détaille les modifications apportées aux opérations suivantes pour que l'ajout et la suppression guarantissent l'équilibre de l'arbre à tout moment.

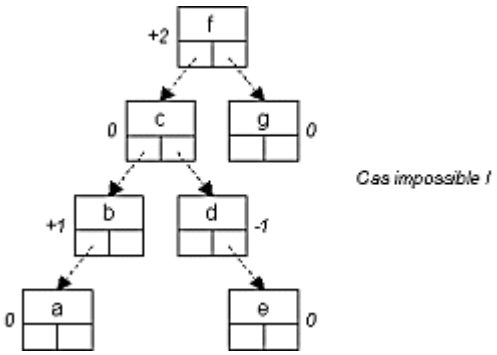
- `ajouterNoeud`,
- `extraireMaximum`,
- `extraireElement`.

Rééquilibrer un arbre

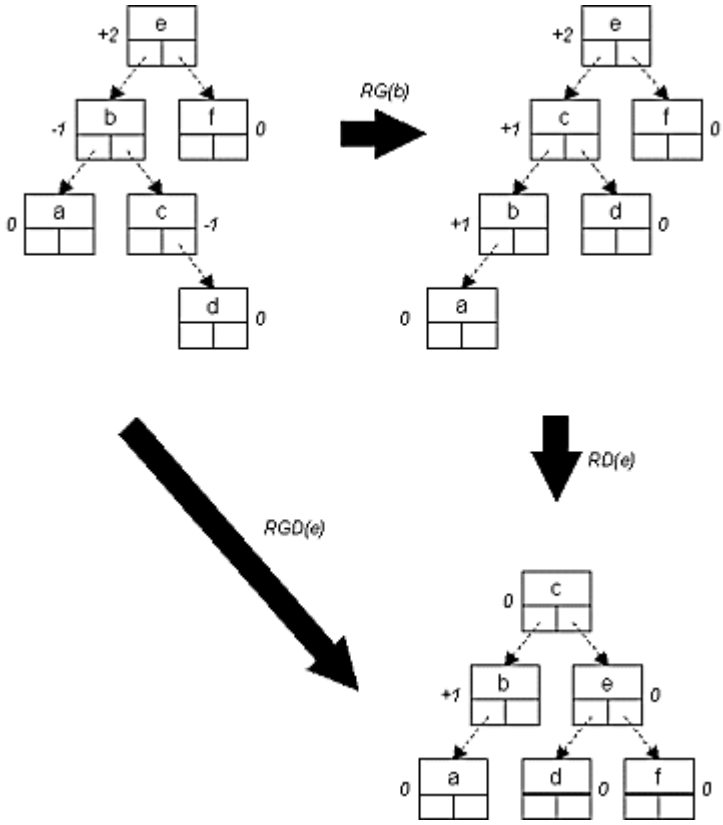
Cette fonction rééquilibre l'arbre pointé par a en effectuant l'une des quatres rotations présentées. Cette opération suppose que les sous-arbres de la racine sont équilibrés. Elle est exécutée après l'ajout ou la suppression d'un élément sur un arbre équilibré. Un rééquilibrage sera effectué si le déséquilibre vaut +2 ou -2. On détaille ici le cas où le déséquilibre vaut +2, le cas -2 étant symétrique. Dans ce cas, il y a un déséquilibre à gauche. Trois possibilités se présentent alors: le déséquilibre du fils gauche vaut +1, 0 ou -1. Dans le premier cas, une rotation simple à droite rééquilibre l'arbre.



Le deuxième cas ne peut pas se produire, car cela signifierait que l'arbre était déjà déséquilibré avant la suppression ou l'ajout d'un élément.



Dans le dernier cas, une rotation double gauche-droite rééquilibre l'arbre.



Voici donc le code de la fonction de rééquilibrage.

```
fonction reequilibrer(ARBRE * a):
 d ← desequilibre(*a);

 si (d = +2) alors
  si (desequilibre((*a)→fg) = -1) alors
   rotationRGD(a);
  sinon
   rotationRD(a);
  fin si;
 sinon si (d = -2) alors
  si (desequilibre((*a)→fd) = +1) alors
   rotationRDG(a);
  sinon
   rotationRG(a);
  fin si;
 fin si;
fin fonction;
```

Ajouter un noeud (avec rééquilibrage)

Les modifications apportées à cette fonction sont les suivantes. Tout d'abord, le noeud inséré se voit attribué une hauteur de 1. Ensuite, après chaque appel récursif à la fonction, la hauteur du noeud racine est mise à jour et une action de rééquilibrage est engagée sur ce même noeud. En effet, après chaque appel récursif, l'arbre est susceptible d'être déséquilibré puisqu'un élément y a été ajouté.

```
fonction BOOLEEN ← ajouterNoeud(ARBRE * a, NOEUD * n):
 si (*a = NIL) alors
  *a ← n;
  n→h ← 1;
  rendre VRAI;
```

```
 fin si;

 si (cle(n→elt) = cle((*a)→elt)) alors rendre FAUX;

 si (cle(n→elt) < cle((*a)→elt)) alors
  ok ← ajouterNoeud(&((*a)→fg),n);
 sinon
  ok ← ajouterNoeud(&((*a)→fd),n);
 fin si;

 si (ok) alors
  majHauteur(*a);
  reequilibrer(a);
 fin si;

 rendre ok;
fin fonction;
```

Extraire la clé maximum (avec rééquilibrage)

Les modifications apportées à cette fonction sont les suivantes. Après chaque appel récursif à la fonction, la hauteur du noeud racine est mise à jour et une action de rééquilibrage est engagée sur ce même noeud. En effet, après chaque appel récursif, l'arbre est susceptible d'être déséquilibré puisqu'un élément y a été supprimé.

```
fonction NOEUD * ← extraireMaximum(ARBRE * a):
 si (*a = NIL) alors rendre NIL;

 si ((*a)→fd = NIL) alors
  n ← *a;
  *a ← (*a)→fg;
  rendre n;
 fin si;

 n ← extraireMaximum(&((*a)→fd));

 si (n != NIL) alors
  majHauteur(*a);
  reequilibrer(a);
 fin si;

 rendre n;
fin fonction;
```

Extraire un élément (avec rééquilibrage)

Les modifications apportées à cette fonction sont les suivantes. Après chaque appel récursif à la fonction, la hauteur du noeud racine est mise à jour et une action de rééquilibrage est engagée sur ce même noeud. En effet, après chaque appel récursif, l'arbre est susceptible d'être déséquilibré puisqu'un élément y a été supprimé.

```
fonction BOOLEEN ← extraireElement(ARBRE * a, CLE c,
                                   ELEMENT * e):
 si (*a = NIL) alors rendre FAUX;

 si (c < cle((*a)→elt)) alors
  ok ← extraireElement(&((*a)→fg),c,e);
 sinon si (c > cle((*a)→elt)) alors
  ok ← extraireElement(&((*a)→fd),c,e);
```

```
 sinon
  *e ← (*a)→elt;
  ok ← supprimerRacine(a);
 fin si;

 si (ok et *a != NIL) alors
  majHauteur(*a);
  reequilibrer(a);
 fin si;

 rendre ok;
fin fonction;
```

CONCLUSION

Cette structure de données est un compromis entre le tableau et la liste chaînée puisqu'elle permet l'accès, l'insertion et la suppression d'un élément "relativement" rapidement. Cependant, pour que cela soit possible, il faut s'assurer de l'équilibre de l'arbre, chose assez compliquée à mettre en place. Les opérations ont été présentées ici de manière récursive, car leur compréhension en est plus simple. Mais le lecteur est invité à réécrire ces opérations de manière itérative. Elles seront plus compliquées mais nettement plus efficaces.