1. Structures linéaires

1.1. Tableaux

La caractéristique fondamentale d'une structure linéaire est l'existence d'une fonction *successeur* qui à chaque élément de la structure – sauf éventuellement un élément particulier, le *dernier* – fait correspondre un autre élément de la structure.

Par exemple, un tableau est une structure de données basée sur la disposition contiguë d'un certain nombre d'éléments ayant le même type. Homogénéité et contiguïté rendent possible l'*accès indexé* aux éléments ; il en découle que, dans un tableau T, le successeur naturel de l'élément T_i est l'élément T_{i+1}. Un tableau est donc naturellement une structure linéaire.

Figure 1 - Tableau

Les opérations les plus fréquentes sur les structures linéaires sont :

- l'*accès* à un élément particulier (le premier élément, le dernier, le $i^{ème}$),
- le *parcours* de la structure dans l'ordre déterminé par la fonction *successeur*.

Par exemple, vous avez déjà écrit d'innombrables fois le parcours d'un tableau T de n éléments :

```
for (i = 0; i < n; i++)
```
« effectuer une certaine opération avec `T[i]` *»*

1.2. Listes chaînées

1.2.1. Notion

Dans un tableau, la relation successeur est implicite, elle découle de la contiguïté des composantes. Dans une liste chaînée, au contraire, la relation successeur est explicite : à chaque élément est accolée une information supplémentaire qui renvoie à son successeur, lequel n'a donc pas besoin d'être contigu, ni même voisin.

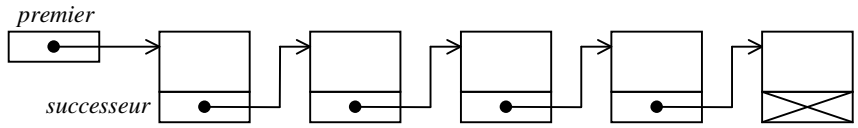


Figure 2 - Liste chaînée

La notion de liste chaînée n'est pas liée à une manière concrète de matérialiser la relation successeur. Cependant, si le langage utilisé le permet, on choisit généralement d'associer deux principes :

- les éléments sont référencés par leurs adresses dans la mémoire,
- chaque élément est alloué dynamiquement, au moment où il commence à être utile.

Ce qui, en C, donne les déclarations :

```
typedef ... OBJET;  /* dépend du problème particulier considéré */

typedef struct maillon {
    OBJET info;
    struct maillon *suiv;
} MAILLON, *POINTEUR;

POINTEUR premier;
```

A retenir, trois points clés des listes chaînées :

- Chaque élément de la liste est formé de n+1 champs :
  - n champs constituant l'information portée par le maillon, dépendante du problème particulier considéré,
  - un champ supplémentaire qui est la matérialisation de la relation *successeur*.
- Le dernier élément est signalé par une valeur conventionnelle du champ successeur (ex. : *NULL*).
- On accède à la liste par l'adresse de son premier élément.

1.2.2. Insertion à la suite d'un élément donné

A titre de modèle, examinons l'opération la plus fondamentale et fréquente dans le traitement des listes chaînées : la création d'un maillon portant une certaine valeur *X* et son insertion dans une liste, à la suite d'un maillon donné. Supposons que *p* ait pour valeur l'adresse de ce maillon (ne cherchons pas ici à savoir comment *p* a été déterminé). La séquence à exécuter est :

```
                                /* état initial:   fig. 3 (a)  */
q = malloc(sizeof(MAILLON));    /*    résultat:    fig. 3 (b)  */
q->info = X;                    /*    résultat:    fig. 3 (c)  */
q->suiv = p->suiv;              /*    résultat:    fig. 3 (d)  */
p->suiv = q;                    /*    résultat:    fig. 3 (e)  */
```

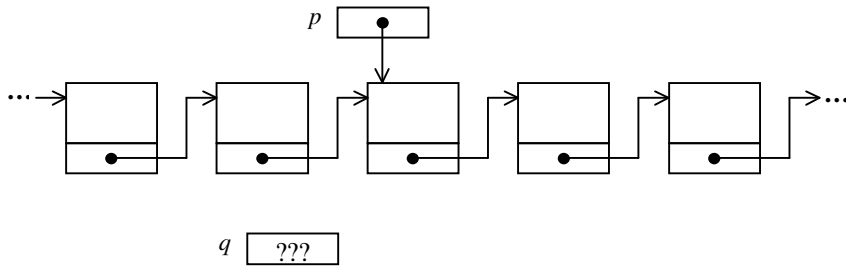


Figure 3 (a)

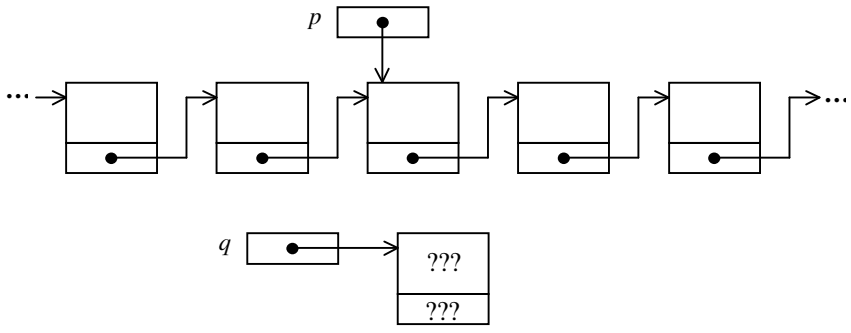


Figure 3 (b)

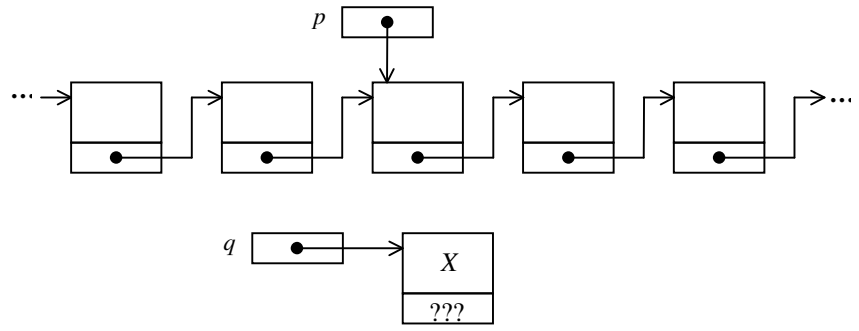


Figure 3 (c)

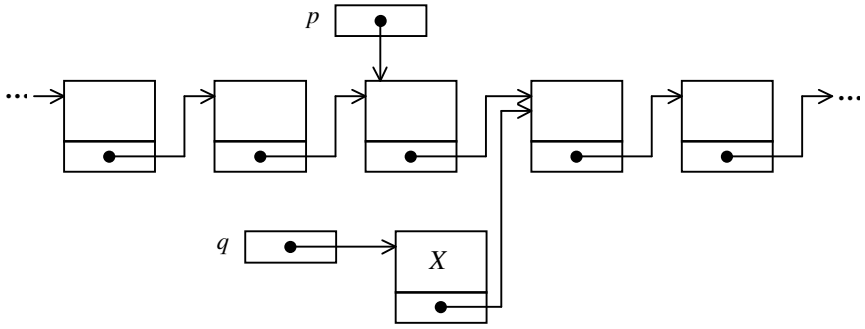


Figure 3 (d)

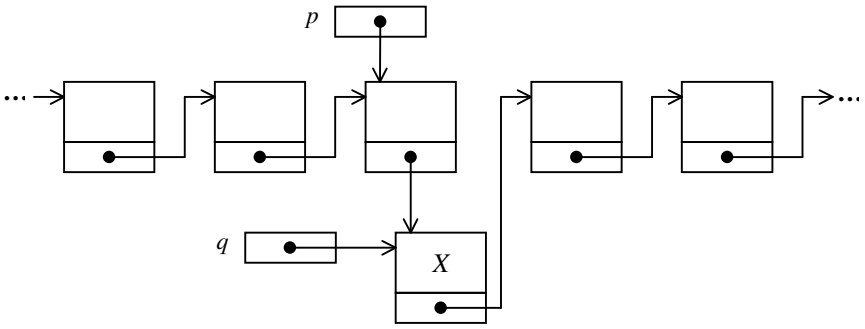


Figure 3 (e)

1.2.3. Utilisation d'un allocateur particularisé

Deux opérations se présentent très souvent ensemble : l'allocation dynamique d'une structure pointée (l'appel de *malloc*) et le remplissage des champs de la structure ainsi créée. Il est clair et pratique de les réunir en une fonction spécifique, une sorte de version personnalisée de *malloc* :

```
POINTEUR maillon(OBJET i, POINTEUR s) {
    POINTEUR p = malloc(sizeof(MAILLON));
    assert(p != NULL);
    p->info = i;
    p->suiv = s;
    return p;
}
```

Cette fonction dépend étroitement de la structure utilisée (ses arguments formels en reproduisent exactement les champs), mais elle est indépendante de l'usage qu'on en fait dans une application. Dans la suite, chaque fois qu'une structure sera déclarée, nous supposerons que nous avons écrit l'allocateur correspondant.

Avec un tel allocateur, les quatre lignes de la page précédente s'écrivent bien plus simplement :

```
p->suiv = maillon(X, p->suiv);
```

1.2.4. Faut-il utiliser un tableau ou une liste chaînée ?

Cette question se pose au programmeur dans de nombreuses applications. Lorsque le choix est possible, on doit considérer les points suivants.

Avantages des tableaux par rapport aux listes :

- *Accès indexé « direct »*. C'est la définition même des tableaux : l'accès au $i^{ème}$ élément se fait en un temps indépendant de i, alors que dans une liste chaînée ce temps est de la forme $k \times i$ (car il faut exécuter i fois l'opération $p = p \rightarrow suiv$).
- *Pas de surencombrement*. La relation *successeur* étant implicite (définie par la contiguïté des composantes), il n'y a pas besoin d'espace supplémentaire pour son codage. Alors que dans une liste chaînée l'encombrement de chaque maillon est augmenté de la taille du pointeur *suivant*[1].

Avantages des listes par rapport aux tableaux :

- *Efficacité et souplesse dans la définition de la relation successeur*. Cette relation étant explicite, on peut la modifier aisément (les maillons des listes chaînées peuvent être réarrangés sans avoir à déplacer les informations portées). Autre aspect de la même propriété : un même maillon peut appartenir à plusieurs listes (cf. § 1.5).
- *Encombrement total selon le besoin*, si on utilise, comme dans les exemples précédents, l'allocation dynamique des maillons. Le nombre de maillons d'une liste correspond au nombre d'éléments effectivement présents, alors que l'encombrement d'un tableau est fixé d'avance et constant.

NOTE. Une conséquence de l'encombrement constant d'un tableau est le risque de saturation : l'insertion d'un élément échoue parce que toutes les cases du tableau sont déjà occupées. Bien sûr, ce risque existe aussi dans le cas des listes chaînées (il se manifeste par l'échec de la fonction *malloc*), mais il correspond alors à la saturation de la mémoire de l'ordinateur ; c'est un événement grave mais rare. Alors que le débordement des tableaux est beaucoup plus fréquent, car il ne traduit qu'une erreur d'appréciation de la part du programmeur (si le tableau est statique) ou de l'utilisateur (si le tableau est dynamique).

1.2.5. Le parcours des listes, NULL et le coup de la sentinelle

Le parcours des listes est une opération qui revient fréquemment dans les applications. Par exemple, le parcours d'une liste à la recherche d'un maillon portant une information x donnée alors qu'on n'est pas sûr qu'un tel maillon existe, peut se programmer ainsi :

```
POINTEUR trouver(OBJET x, POINTEUR p) { /* rend p tel que p->info == x ou NULL */
    while (p != NULL && p->info != x)
        p = p->suiv;
    return p;
}
```

Supposons avoir à écrire un programme où cette opération est critique (autrement les considérations suivantes sont sans intérêt). Nous devons essayer de rendre la boucle *while* ci-dessus, qui est déjà très simple, encore moins onéreuse.

C'est le moment de nous rappeler que la seule caractéristique de *NULL* que nous utilisons est le fait d'être une adresse conventionnelle distincte de toutes les autres adresses manipulées. Par conséquent, n'importe quelle adresse pourra jouer le rôle de *NULL*, à la condition qu'elle ait fait l'objet d'une convention préalable, connue et suivie par la partie de notre programme qui a construit la liste. Par exemple, l'adresse *NULLBIS* d'un maillon créé par nos soins :

```
MAILLON laFinDeToutesLesListes;
#define NULLBIS (&laFinDeToutesLesListes)
```

Il est clair qu'un programme correct (ne contenant pas initialement les deux noms ci-dessus) dans lequel on a ensuite ajouté les deux lignes précédentes, puis remplacé par *NULLBIS* chaque occurrence de *NULL*, est encore correct. Par exemple (ceci suppose que, comme dit, en construisant la liste on a utilisé *NULLBIS* à la place de *NULL*) :

---

1 Néanmoins, ce surencombrement est bien moins important qu'il n'y parait car, contrairement à ce que suggèrent nos dessins, le champ *info* est souvent assez volumineux, ce qui rend la taille du champ *suiv* négligeable devant celle du maillon tout entier.

```
POINTEUR trouver(OBJET x, POINTEUR p) { /* rend p t.q. p->info == x ou NULLBIS */
    while (p != NULLBIS && p->info != x)
        p = p->suiv;
    return p;
}
```

Pour le moment nous n'avons rien gagné. Mais cela devient intéressant quand on s'aperçoit qu'on peut améliorer la fonction **trouver** en logeant dans le maillon pointé par *NULLBIS* une valeur « sentinelle » :

```
POINTEUR trouver(OBJET x, POINTEUR p) { /* rend p t.q. p->info == x ou NULLBIS */
    NULLBIS->info = x;
    while (p->info != x)    /* maintenant on est sûr de "trouver" x */
        p = p->suiv;
    return p;
}
```

Dans la boucle (la seule partie qui doit nous intéresser, du point de vue de l'efficacité) nous avons remplacé deux comparaisons par une seule. On peut estimer que la boucle a été réduite de 2/3.

1.2.6. La structure de liste vue de haut

Des sections précédentes se dégage une notion de liste assez terre à terre, faisant penser plus à une collection de trucs et astuces pour économiser l'espace ou gagner du temps qu'à une structure de données tant soit peu abstraite. Pour rehausser le débat, voici comment on peut présenter les listes d'une manière plus formelle, comme cela se fait dans les langages (Lisp, Prolog, etc.) où elles sont les structures de données fondamentales.

DEFINITION. Soit X un ensemble. Une liste d'éléments de X est définie récursivement comme

- un symbole conventionnel, appelé la *liste vide*, ou bien
- un couple (i , L) où $i \in X$ et L est une liste d'éléments de X.

Le rapport avec ce que nous appelions liste précédemment devient évident si on décide qu'une liste sera représentée dans les programmes par une adresse : soit *NULL*, représentant la liste vide, soit l'adresse d'un maillon à deux champs représentant une paire (i , L) :

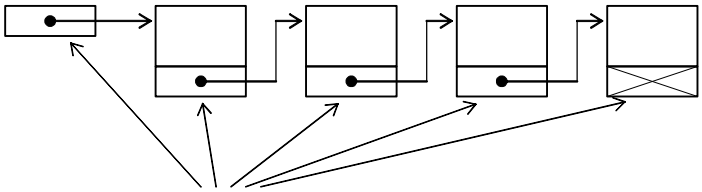


Figure 4. Autour de la notion de liste

De cette définition récursive des listes dérive immédiatement une définition récursive de l'algorithme de leur parcours : pour parcourir une liste L il faut :

- si L est la liste vide, ne rien faire ;
- si $L = (i , L')$, exploiter i puis parcourir L'.

Ici, « exploiter » représente une opération qui dépend de l'ensemble X et du problème particulier considéré. Ce qui donne le programme :

```
void parcours(POINTEUR liste) {
    if (liste != NULL) {
        EXPLOITER(liste->info);
        parcours(liste->suiv);
    }
}
```

1.2.7. Les listes ordonnées

Une liste ordonnée est une liste dont les maillons ont un champ *clé* (éventuellement le champ *info* tout entier), dont les valeurs appartiennent à un ensemble totalement ordonné, et la liste vérifie

si $p \rightarrow suiv$ existe alors $p \rightarrow cle \leq p \rightarrow suiv \rightarrow cle$

Les listes ordonnées se rencontrent très fréquemment, nous en verrons des exemples dans les pages suivantes. L'ajout d'un maillon nouveau dans une telle structure met en évidence une qualité des listes : l'insertion d'un élément au milieu de la liste ne requiert pas le déplacement « physique » des éléments qui seront derrière le nouveau venu.

L'insertion d'un élément dans une liste ordonnée se fait en trois temps (les deux premiers sont interchangeables) :

- rechercher la place dans la liste du nouvel élément,
- allouer le nouveau maillon et remplir ses champs,
- raccrocher le nouvel élément à la liste, ce qui se fait de deux manières, selon que
  - la liste est vide ou le nouvel élément se place à la tête de la liste (i.e. : le nouvel élément ne sera le successeur d'aucun maillon),
  - la liste n'est pas vide et le nouvel élément ne se place pas en tête.

Ce qui donne, en supposant que la liste est repérée par un pointeur *teteDeListe* :

```
POINTEUR pr, pc, nouveau;

pc = teteDeListe;       /* pc parcourt la liste,  pr court après pc */
while (pc != NULL && pc->cle < x) {
    pr = pc;
    pc = pc->suiv;
}
nouveau = maillon(x, autres informations, pc);
if (pc == teteDeListe)
    teteDeListe = nouveau;
else
    pr->suiv = nouveau;
```

1.2.8. Le célèbre maillon « bidon »

On peut abolir la différence de traitement qui existe entre l'insertion d'un maillon en tête d'une liste et l'insertion à une autre place (voir le programme précédent), en convenant que toute liste commencera par un maillon « technique », sans signification pour l'application. Ainsi, d'un point de vue pratique, une liste ne sera jamais vide et, lors de l'insertion d'un maillon, il n'y aura qu'un seul cas à considérer.

Si on procède ainsi, une liste ne sera pas représentée par l'adresse de son premier maillon utile mais par l'adresse d'un maillon sans signification, voire – si on programme en C – par ce maillon sans signification lui-même. A la place de :

```
POINTEUR teteDeListe;
```

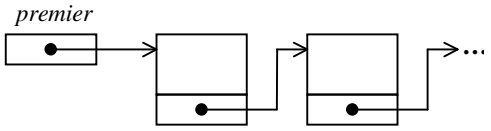


Figure 5 (a). Liste habituelle

on aura

```
MAILLON maillonBidon;
```

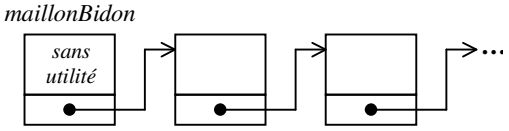


Figure 5 (b). Liste commençant par un maillon sans signification

Voici le programme de recherche-insertion qui correspond à une telle liste. Comme prévu, la deuxième partie de l'insertion se simplifie bien :

```
POINTEUR pr, pc;
…
pr = &maillonBidon;
pc = maillonBidon.suiv;
while (pc != NULL && pc->cle < x) {
    pr = pc;
    pc = pc->suiv;
}
pr->suiv = maillon(x, autres informations, pc);
```

On peut trouver cette manière de gérer les listes préférable, mais ce n'est pas une évidence. Elle ne simplifie que le texte du programme, dont l'exécution n'est ni plus ni moins rapide que celle de la version précédente, car nous n'avons pas modifié la boucle *while* qui, du point de vue de l'efficacité, est la seule partie à considérer. D'autre part, cette simplification se paye par un encombrement supplémentaire de chaque liste, puisque nous avons remplacé un pointeur par un maillon tout entier. Ce sur-encombrement ne peut être tenu pour négligeable que lorsqu'on doit gérer un petit nombre de listes ou bien des listes faites de maillons dont la partie *info* est de petite taille.

VARIANTE. Ayant introduit le maillon sans signification, qui garantit qu'une liste a toujours au moins un élément, nous pouvons récrire le programme précédent à l'aide d'un seul pointeur :

```
POINTEUR pr;
…
pr = &maillonBidon;
while (pr->suiv != NULL && pr->suiv->cle < x)
    pr = pr->suiv;

pr->suiv = maillon(x, autres informations, pr->suiv);
```

Il est difficile de dire si cette version du programme est plus ou moins rapide que la précédente car, dans la boucle, nous avons supprimé une affectation mais nous avons ajouté deux indirections.

1.3. Piles

Nous avons défini les tableaux et les listes chaînée par leur constitution (en répondant à la question « comment cela est fait ? »). Au contraire, les piles et les queues se définissent par leur comportement (en répondant à la question « qu'est-ce que cela fait ») ; on dit que ce sont des structures de données abstraites.

Une pile est une structure de données dynamique (des éléments y sont introduits, puis extraits) ayant la propriété que, lors d'une extraction, l'élément extrait est celui qui a y été introduit le plus récemment. On dit « dernier entré, premier sorti » ou, en bon français, LIFO (« Last Int First Out »).

C'est une propriété abstraite, qui n'impose pas une constitution particulière. En fait, on réalise aussi bien des piles à partir de tableaux qu'à partir de listes chaînées.

1.3.1. Réalisation d'une pile à l'aide d'un tableau

```
typedef ... OBJET;      /* dépend du problème */

#define MAXPILE ...     /* selon estimation du besoin */

OBJET pile[MAXPILE];
int niveau;

void initialiser(void)
    {
    niveau = 0;
    }

void empiler(OBJET x)
    {
    assert(niveau < MAXPILE);
    pile[ niveau++ ] = x;
    }

OBJET depiler(void)
    {
    assert(niveau > 0);
    return pile[ --niveau ];
    }
```

1.3.2. Réalisation d'une pile à l'aide d'une liste chaînée

```
typedef struct maillon
    {
    OBJET info;
    struct maillon *suiv;
    } MAILLON, *POINTEUR;

POINTEUR maillon(OBJET i, POINTEUR s)
    {
    POINTEUR p;
    p = malloc(sizeof(MAILLON));
    assert(p != NULL);
    p->info = i;
    p->suiv = s;
    return p;
    }

POINTEUR pile;

void initialiser(void)
    {
    pile = NULL;
    }

void empiler(OBJET x)
    {
    pile = maillon(x, pile);
    }

OBJET depiler(void)
    {
    POINTEUR p;
    OBJET r;
    assert(pile != NULL);
    p = pile;
    pile = pile->suiv;
    r = p->info;
    free(p);
    return r;
    }
```

1.4. Queues

Une *queue*, ou *file d'attente*, est une structure de données dynamique telle que, lors d'une extraction, l'élément extrait est celui qui s'y trouve depuis le plus longtemps. On dit « premier entré, premier sorti » ou encore FIFO (« First Int First Out »).

1.4.1. Réalisation d'une queue à l'aide d'une liste chaînée

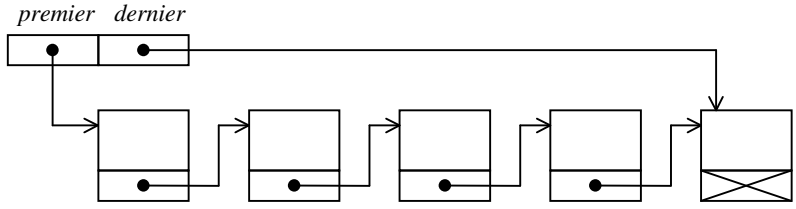


Figure 6. Queue réalisée par une liste chaînée

```
POINTEUR premier, dernier;

void initialiser(void)
    {
    premier = NULL;
    }
```

```
void entrer(OBJET x)
    {
    POINTEUR p;
    p = maillon(x, NULL);
    if (premier == NULL)
        premier = p;
    else
        dernier->suiv = p;
    dernier = p;
    }

OBJET sortir(void)
    {
    POINTEUR p;
    OBJET r;
    assert(premier != NULL);
    p = premier;
    premier = premier->suiv;
    r = p->info;
    free(p);
    return r;
    }
```

1.4.2. Réalisation d'une queue à l'aide d'un tableau

Pour implanter une queue nous nous donnerons deux indices, appelés *premier* et *dernier*, et nous utiliserons un *tableau circulaire*, obtenu à partir d'un tableau ordinaire en décidant que la première case du tableau fait suite à la dernière. D'un point de vue technique, il suffira de modifier légèrement l'opération « passage au successeur ». Pour un tableau ordinaire, cette opération se traduit par l'incrémentation d'un indice :

```
i = i + 1;
```

Pour avoir un tableau circulaire, nous faisons une incrémentation modulo la taille du tableau :

```
i = (i + 1) % TAILLE_TABLEAU;
```

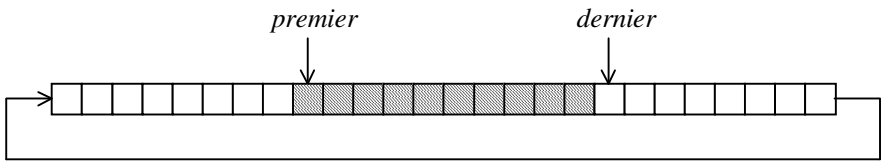


Figure 7 (a) Queue réalisée par un tableau circulaire

Attention, *premier* n'est pas nécessairement devant *dernier* :

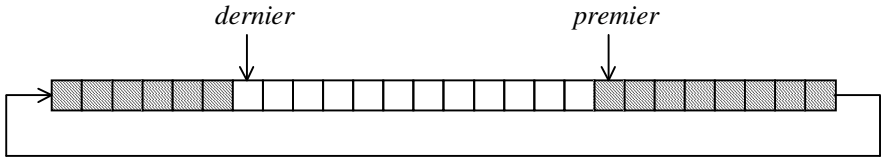


Figure 7 (b) Queue réalisée par un tableau circulaire

```
#define MAXQUEUE 10

OBJET queue[MAXQUEUE];
int premier, dernier, nombre;

#define SUCCESS(i) (((i) + 1) % MAXQUEUE)

void initialiser(void)
    {
    nombre = 0;
    premier = dernier = 0;
    }
```

```
void entrer(OBJET x)
    {
    assert(nombre < MAXQUEUE);
    nombre++;
    queue[dernier] = x;
    dernier = SUCCESS(dernier);
    }

OBJET sortir(void)
    {
    OBJET r;
    assert(nombre > 0);
    nombre--;
    r = queue[premier];
    premier = SUCCESS(premier);
    return r;
    }
```

Remarque. On peut être surpris par la présence du compteur *nombre*, croyant que la position relative de *premier* et *dernier* suffit pour déceler qu'une queue est vide ou pleine. Or, il y a un petit piège : la figure 8 montre une queue avec deux éléments, que l'on fait sortir successivement, et une autre avec deux cases libres, que l'on remplit.

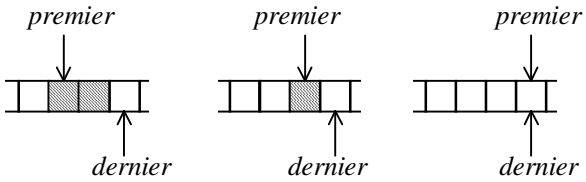


Figure 8 (a) Comment une queue se vide

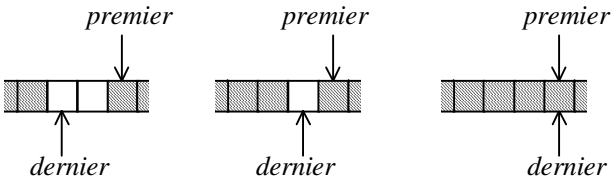


Figure 8 (b) Comment une queue se remplit

On constate que la position relative de *premier* et *dernier* lorsque la queue est vide est la même que celle qui correspond à une queue pleine. D'où la nécessité de prendre des mesures supplémentaires, comme veiller à ce qu'il y ait toujours une case libre ou bien utiliser un compteur du nombre d'éléments.

1.5. Application aux matrices creuses (listes orthogonales)

Un avantage important des listes chaînées est de permettre de donner à une même collection d'informations plusieurs structures de liste, selon plusieurs critères. Par exemple, une collection de fiches décrivant les abonnés à un certain service peut être organisée de telle manière que chaque élément appartienne à plusieurs listes : la liste des abonnés qui sont dans la même tranche d'âge, la liste des abonnés qui ont le même type de profession, la liste des abonnés qui habitent dans la même région, etc.

Nous allons considérer un exemple concret d'une telle organisation : les listes orthogonales, utilisées pour la représentation économique des matrices creuses, ou matrices contenant « beaucoup » de zéros[2]. L'idée est de maintenir de telles matrices en ne représentant que les coefficients non nuls. Plus précisément (voyez la figure 9) chaque valeur $a_{i,j}$ telle que $a_{i,j} \neq 0$ occupe un maillon appartenant à *deux* listes chaînées : la liste des coefficients non nuls de la ligne i (ordonnée par rapport aux indices de colonne, nous verrons l'intérêt d'un tel arrangement) et la liste des coefficients non nuls de la colonne j (ordonnée par rapport aux indices de ligne). Il y a donc une liste, éventuellement vide, pour chaque ligne et une liste pour chaque colonne.

Ces maillons pourront être déclarés de la manière suivante :

[2] De telles matrices ne sont pas aussi marginales qu'on pourrait le croire. Par exemple, une méthode classique pour représenter un graphe, comme un réseau routier, dont les sommets sont numérotés de 1 à n, consiste à introduire la matrice $(a_{i,j})$ définie par $a_{i,j}$ = la longueur de l'arc joignant le sommet i au sommet j, si cet arc existe, 0 sinon. Cette matrice possède n^2 coefficients. Or, voyez une carte routière, le nombre d'arêtes du graphe, c'est-à-dire le nombre de $a_{i,j}$ non nuls, est « de l'ordre de n » (i.e. majoré par $k \times n$ avec, souvent, k ridiculement petit).

```
typedef struct maillon
    {
    double val;
    short il;               /* indice de la ligne           */
    short ic;               /* indice de la colonne         */
    struct maillon *sl;     /* suivant dans la même ligne   */
    struct maillon *sc;     /* suivant dans la même colonne */
    } MAILLON, *POINTEUR;
```

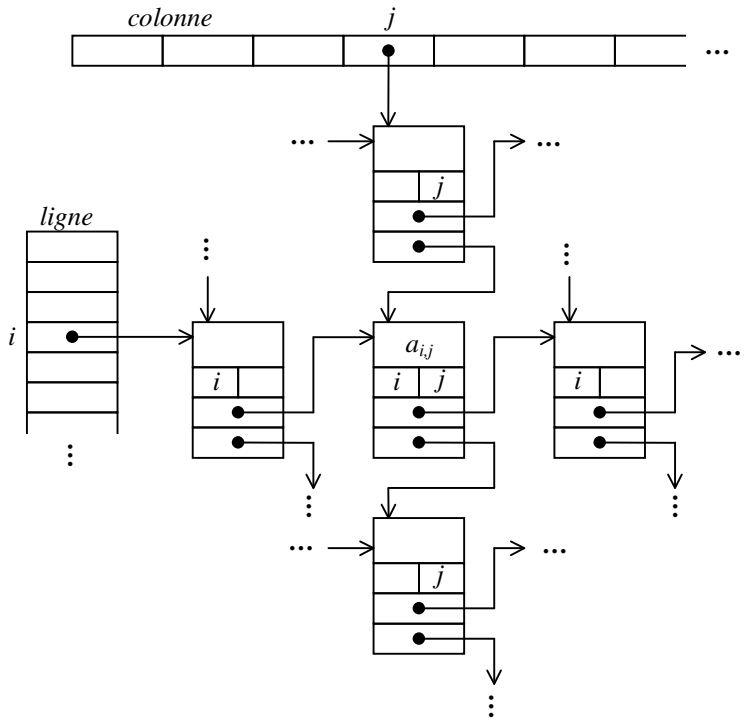


Figure 9. Représentation d'une matrice creuse

Supposons que les matrices qui nous intéressent aient *n* ligne et *n* colonnes. Une matrice sera alors déterminée par la donnée de deux tableaux de n pointeurs chacun : le tableau des têtes des lignes et celui des têtes des colonnes :

```
#define N … /* dépend du problème considéré */

typedef struct matrice
    {
    POINTEUR ligne[N], colonne[N];
    } MATRICE;
```

Supposons que, comme le suggère la figure 9, les champs *val*, *sl* et *sc* ont la même taille, qui est le double de celle des champs *il* et ic (si *val* avait une taille supérieure, ce dispositif serait encore plus avantageux). Soit *p* le nombre de coefficients non nuls. Exprimé en nombre de fois la taille d'un pointeur, l'encombrement de notre structure est $2n + 4p$. La place occupée par la matrice ordinaire correspondante aurait été n^2. L'affaire est donc rentable dès que

$$p < \frac{n^2 - 2n}{4} \approx \frac{n^2}{4}$$

Examinons à présent la manipulation de telles structures. Comme on pouvait s'y attendre, la création d'un élément est un peu compliquée, mais les autres opérations sont effectuées avec une efficacité égale ou supérieure à celle des matrices habituelles ; à titre d'exemple nous présentons le produit.

Ajout d'un élément. L'ajout d'un coefficient revient à faire deux fois l'insertion d'un maillon dans une liste ordonnée. Notez que l'utilisation d'une sentinelle (*NULL* a été remplacé par *NULLBIS*, l'adresse d'un maillon ayant le plus grand entier comme numéro de ligne et de colonne) simplifie les deux boucles *while* qui constituent l'essentiel du travail :

```
MAILLON finDesListes = { 0, INT_MAX, INT_MAX, NULL, NULL };

#define NULLBIS (&finDesListes)

void insertion(double a, int i, int j, MATRICE *m)
    {
    POINTEUR pr, pc, q = maillon(a, i, j, NULLBIS, NULLBIS);

    pc = m->ligne[i];
    while (pc->ic < j)          /* rappel: NULLBIS->ic == INT_MAX */
        pr = pc, pc = pc->sl;
    q->sl = pc;
    if (pc == m->ligne[i])
        m->ligne[i] = q;
    else
        pr->sl = q;

    pc = m->colonne[j];
    while (pc->il < i)          /* rappel: NULLBIS->il == INT_MAX */
        pr = pc, pc = pc->sc;
    q->sc = pc;
    if (pc == m->colonne[j])
        m->colonne[j] = q;
    else
        pr->sc = q;
    }
```

Multiplication de deux matrices. Le programme obtenu, malgré la complexité des structures mises en œuvre, est plus rapide que celui qui utilise la structure usuelle des matrices car, dans la boucle la plus profonde, il n'y a pas d'accès complexe du type *t[i][j]* :

```
void produit(MATRICE *a, MATRICE *b, MATRICE *c)
    {
    int i, j;
    double v;
    POINTEUR pa, pb;

    for (i = 0; i < N; i++)
        c->ligne[i] = c->colonne[i] = NULLBIS;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            {
            v = 0;
            pb = b->colonne[j];
            for (pa = a->ligne[i]; pa != NULLBIS; pa = pa->sl)
                {
                while (pb->il < pa->ic)
                    pb = pb->sc;
                if (pb->il == pa->ic)
                    v += pa->val * pb->val;
                }
            if (v != 0)
                insertion(v, i, j, c);
            }
    }
```

2. Arbres et arborescences

Il existe deux approches classiques de la notion d'arbre : le point de vue de la théorie des graphes, où un arbre est une variété d'espace topologique, et le point de vue des structures de données, où un arbre – ou plutôt une arborescence – est une structure que l'on donne à un ensemble d'informations.

2.1. Arbres

Un *arbre* est un graphe[3] non orienté $G = (S , A)$ qui vérifie les propriétés équivalentes suivantes :

- G est connexe minimal (si on lui enlève une arête il n'est plus connexe) ;
- G est acyclique maximal (si on lui ajoute une arête on crée un cycle) ;
- $| A | = | S | - 1$ et G est connexe ou acyclique ;
- deux sommets quelconques de G sont reliés par un unique chemin élémentaire.

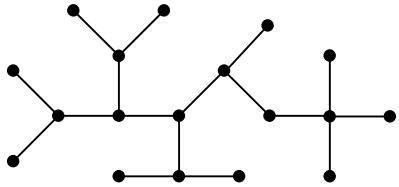


Figure 1. Arbre

Un *arbre enraciné* est un arbre dans lequel un des sommets, appelé la *racine*, est distingué.

Soit G un arbre enraciné de racine r et x un sommet de G. Un sommet y appartenant au chemin unique allant de r à x est appelé *ancêtre* de x, et x est alors appelé *descendant* de y. Si (y, x) est le dernier arc du chemin joignant r à x, on dit que y est le *père* de x et que x est le *fils* de y. La racine n'a pas de père. Si deux sommets ont le même père, on dit qu'ils sont *frères*. On appelle *sous-arbre de racine x* le sous-arbre de G composé de x et ses descendants.

2.2. Arborescences

Une *arborescence* est un ensemble fini non vide A muni de la structure suivante :

- il existe un élément distingué, appelé la *racine* de A ;
- l'ensemble des autres éléments de A, s'il n'est pas vide, est *partitionné*[4] en une famille de sous-ensembles qui sont des arborescences.

---

[3] L'étude des graphes fait l'objet d'un autre chapitre de ce cours.

[4] Rappelons qu'une partition d'un ensemble E est une famille de sous-ensembles de E deux à deux disjoints dont la réunion est l'ensemble E.

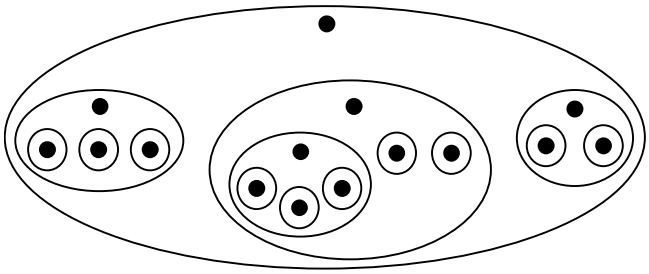


Figure 2. Arborescence

2.2.1. Jargon sur les arbres

ABUS DE LANGAGE. Il arrive rarement que l'on ait à traiter, dans le même discours, des arbres et des arborescences. On se permet donc d'employer le même vocabulaire, celui des arbres, pour travailler sur les arbres et sur les arborescences ; en principe, le contexte indique toujours de quoi il s'agit.

Nous pratiquerons ici cet abus de langage, et nous dirons désormais arbre pour arborescence.

Sous-arbre immédiat. Soit A un arbre de racine r. Par définition, l'ensemble $A - \{ r \}$ est partitionné en une famille d'arbres ; ces derniers sont appelés les sous-arbres immédiats de A.

Sous-arbre. B est un sous-arbre de A si

- B est un sous-arbre immédiat de A, ou
- il existe C, sous-arbre immédiat de A, tel que B soit un sous-arbre de C

Nœud. Les éléments de A sont appelés nœuds. Chaque nœud est racine d'un sous-arbre de A (pour s'en convaincre il suffit d'appliquer récursivement la définition d'arbre). Inversement, chaque sous-arbre possède une racine. Il y a donc une bijection entre les nœuds et les sous-arbres d'un arbre.

Ancêtre, descendant. Soit X un sous-arbre de A de racine x et soit y un élément d'un sous-arbre de X. On dit que x est un ancêtre de y et que y est un descendant de x.

Père, *fils*, *frère*. Soit X un sous-arbre de A de racine x et soit y la racine d'un sous-arbre immédiat de X. On dit que x est le père de y, et que y est un fils de x. Parler de « un arbre et ses sous-arbres immédiats » revient donc à parler de « un nœud et ses fils ». Entre eux, les fils d'un nœud sont dits frères.

Représentation par un graphe orienté. La représentation habituelle des arbres consiste à représenter les nœuds, puis à relier par des flèches, implicitement orientées du haut vers le bas, chaque nœud à ses fils :

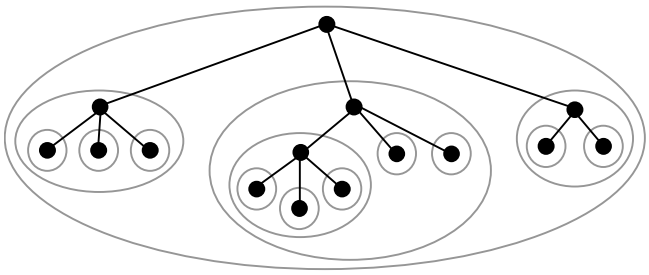


Figure 3. Arbre, représentation usuelle

Feuilles. Un nœud qui n'a pas de fils est appelé nœud externe ou feuille. Un nœud qui n'est pas une feuille est un nœud interne.

Degré. Le nombre de fils d'un nœud est le degré de ce nœud. Un nœud de degré 0 est donc une feuille.

Profondeur. Soit x un nœud d'un arbre A. La profondeur de x dans A est la longueur (i.e. le nombre d'arcs) du chemin allant de la racine de A à x.

Hauteur. La hauteur d'un arbre A est le maximum des profondeurs de ses nœuds.

Arbre ordonné. S'il existe une relation d'ordre définie sur les fils de chaque nœud, alors on dit que l'arbre est ordonné. Les arbres sont souvent ordonnés, parfois involontairement : comment nommer, dessiner, etc. un arbre sans l'ordonner ?

2.2.2. Implantation des arbres

En vue de les implanter dans les programmes, on peut formaliser la définition des arbres de la manière suivante :

> Soit X un ensemble donné. Un arbre dont les nœuds portent des éléments de X est un couple (i , E) constitué d'un élément $i \in X$ et d'un ensemble fini E, éventuellement vide, d'arbres dont les nœuds portent des éléments de X.

Comme on l'a dit, on a beaucoup de mal à empêcher les arbres d'être ordonnés ; en pratique un ordre existe toujours entre les fils d'un nœud. Pour ce que nous avons à en faire ici, nous ne perdons pas grand-chose en remplaçant la définition précédente par la définition plus restrictive suivante :

> Soit X un ensemble donné. Un arbre dont les nœuds portent des éléments de X est un couple (i , ($A_1, A_2, \ldots A_n$)) constitué d'un élément $i \in X$ et d'une suite finie ($A_1, A_2, \ldots A_n$), éventuellement vide, d'arbres dont les nœuds portent des éléments de X.

L'implantation des arbres découle de cette définition :

- la suite ($A_1, A_2, \ldots A_n$) est représentée par une liste chaînée d'arbres ;
- un arbre est représenté comme un couple (*info* , *liste-de-fils*)

Si on était tout à fait rigoureux, on devrait donc déclarer : d'un côté, un arbre comme un couple :

```
typedef struct arbre
    {
    OBJET info;
    LISTE_D_ARBRES fils;
    } ARBRE;
```

et, d'un autre côté, une liste d'arbres, comme nous savons déjà le faire :

```
typedef struct maillon
    {
    ARBRE sousArbre;
    struct maillon *frere;
    } *LISTE_D_ARBRES;
```

En réalité, on se contente d'un peu moins de rigueur et on fait l'unique définition :

```
typedef struct nœud
    {
    OBJET info;
    struct noeud *fils;
    struct noeud *frere;
    } NOEUD, *POINTEUR;
```

qui correspond à :

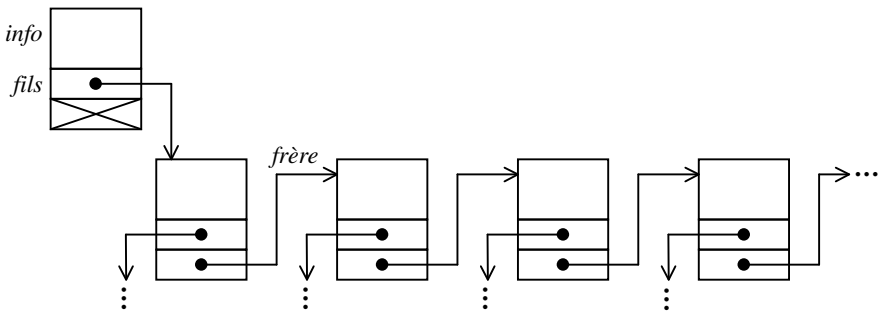


Figure 4. Implantation des arbres

C'est moins rigoureux parce qu'on met sur le même plan les fils et les frères, ce qui n'a pas de justification dans la notion d'arbre que nous avons donnée.

A RETENIR. Dans le cas général (la situation est différente dans le cas des arbres binaires), l'arbre dont la structure « logique » est celle de la figure 5 (a) s'implante comme le montre la figure 5 (b) :

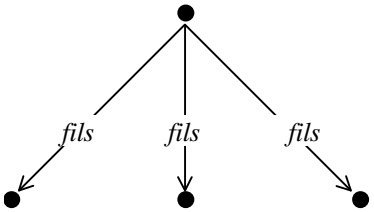


Figure 5 (a)

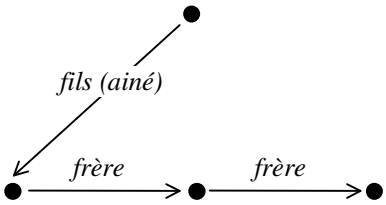


Figure 5 (b)

2.3. Arbres binaires

Dans les arbres binaires, chaque nœud a au plus deux fils. Bien sûr, on peut les voir comme un cas particulier des arbres quelconques, mais dans beaucoup de domaines on préfère la définition spécifique suivante (qui est à rapprocher de la définition des listes du chapitre 1, section 1.2.4) :

DEFINITION. Soit X un ensemble donné. Un arbre binaire portant des éléments de X est

- un symbole conventionnel, appelé arbre vide, ou bien
- un triplet (i , G, D) où $i \in X$ et où G et D sont des arbres binaires portant des éléments de X.

L'implantation usuelle des arbres binaires découle de cette définition :

```
typedef struct nœud
    {
    OBJET info;
    struct noeud *gauche, *droite;
    } NOEUD, *POINTEUR;
```

Par exemple, l'expression arithmétique $3 \times (a + 5)$ pourrait être représentée par l'arbre binaire :

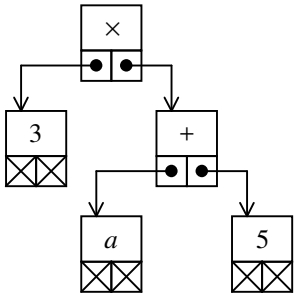


Figure 6. Représentation d'une expression arithmétique par un arbre binaire

2.4. Parcours des arbres

On appelle parcours d'un arbre A le travail qui consiste à effectuer un certain traitement, dépendant de l'application particulière considérée, sur l'information portée par chaque nœud de A.

C'est sans doute le meilleur exemple d'algorithme récursif que l'on puisse trouver. Partant du problème « parcourir l'arbre $A = (i , (A_1 , \ldots A_n))$ », on obtient :

- un problème simple : « traiter i »
- n problèmes analogues au problème initial : « parcourir A_1 », … « parcourir A_n »

Un point reste à régler, qui dépend de l'application particulière considérée : dans quel ordre doit-on faire les $n + 1$ opérations mentionnées ci-dessus ?

2.4.1. Parcours des arbres binaires

Dans le cas des arbres binaires les trois manières possibles d'arranger ce parcours sont fréquemment utilisées et ont fait l'objet de dénominations spécifiques : le parcours en pré-ordre, le parcours en in-ordre et le parcours en post-ordre, définis respectivement par le fait que chaque nœud est traité avant, entre ou après les parcours de ses deux fils.

Ainsi, le parcours en pré-ordre peut se programmer :

```
void parcours(POINTEUR arbre)
    {
    if (arbre != NULL)
        {
        EXPLOITER(arbre->info);
        parcours(arbre->gauche);
        parcours(arbre->droite);
        }
    }
```

Les fonctions qui implantent les deux autres parcours ne diffèrent de celle-ci que par la position de l'appel de *EXPLOITER* relativement aux deux appels de parcours.

A titre d'exemple, considérons encore le codage par un arbre binaire de l'expression $3 \times (a + 5)$, et supposons que la fonction *EXPLOITER* se réduise à l'affichage du champ *info* de chaque nœud. L'effet du parcours de l'arbre sera donc un certain affichage de l'expression. On obtient :

- dans le parcours en in-ordre : 3 × a + 5 (notation infixée – et ambiguë !)
- dans le parcours en pré-ordre : × 3 + a 5 (notation « polonaise »)
- dans le parcours en post-ordre : 3 a 5 + × (notation « polonaise inversée »)

2.4.2. Parcours d'un arbre quelconque

On utilise souvent le parcours qui correspondrait au pré-ordre :

```
void parcours(POINTEUR arbre)
    {
    POINTEUR p;

    EXPLOITER(arbre->info);
    for (p = arbre->fils; p != NULL; p = p->frere)
        parcours(p);
    }
```

Notez que ce programme est juste parce que, avec notre définition, un arbre (non binaire) ne peut pas être vide.

La figure 7 montre le travail de la fonction *parcours*, comme l'itinéraire d'une bestiole se promenant sur l'arbre. Les flèches vers le bas représentent les appels que la fonction *parcours* fait d'elle même ; on appelle cela la « descente aux fils ». Chaque appel *parcours(p)* provoque le parcours du sous-arbre ayant p pour racine. Lorsque ce parcours est terminé, le contrôle revient à la fonction qui a fait l'appel de parcours. Ce retour est symbolisé par la partie remontante des flèches ; vu du côté du fils, on appelle cela la « remontée au père ».

Les nombres écrits à gauche de chaque nœud expriment la chronologie de leur exploitation dans le cas du parcours en pré-ordre. Les nombres écrits entre parenthèses, à droite de chaque nœud, correspondent au cas du post-ordre.

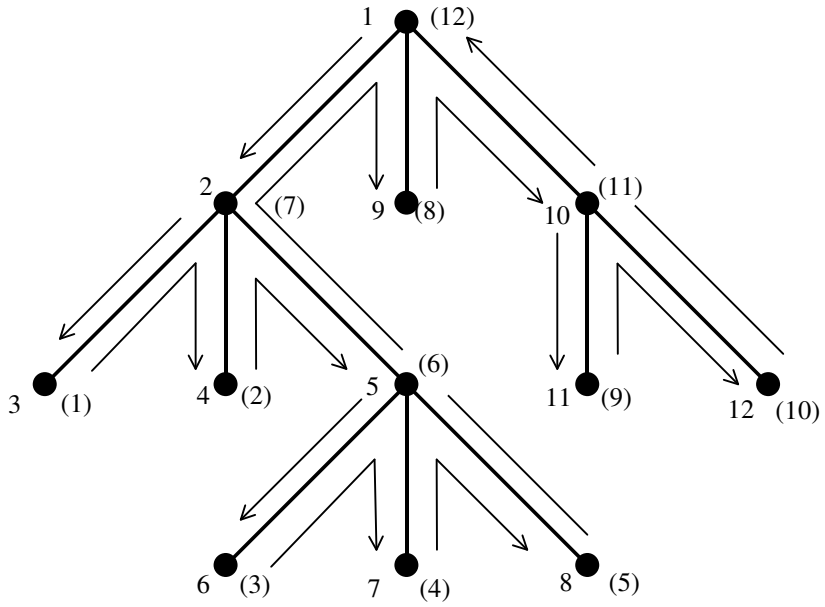


Figure 7. Parcours d'un arbre en pré-ordre (post-ordre)

La figure 8 montre un moment du parcours de l'arbre précédent : celui où le sommet marqué 7 - (4) va être exploité. Quatre instances de la fonction *parcours* sont actives (quatre générations distinctes des variables *arbre* et *p* existent donc en même temps, empilées les unes au-dessus des autres) : dans la première, la boucle *for* en est à son début ; i.e. la variable *p* correspondante pointe le premier fils de arbre ; dans la deuxième instance, la boucle *for* en est à son dernier tour, dans la troisième, elle en est au milieu. Dans la quatrième instance, la boucle for ne démarrera même pas.

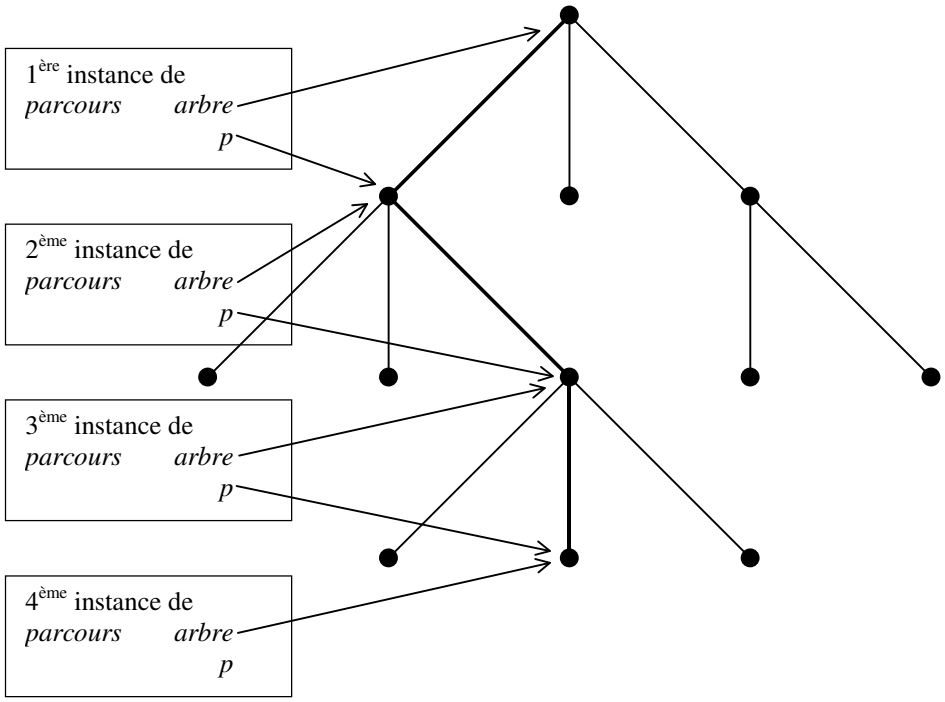


Figure 8. Un moment du parcours de l'arbre de la figure 7

VARIANTE. Dans l'algorithme de parcours d'un arbre il y a un aspect naturellement récursif, la descente au fils, et un aspect naturellement itératif, le parcours de la liste de frères. Or, nous l'avons vu, une liste chaînée peut aussi être parcourue selon une fonction récursive. On peut en déduire une fonction de parcours d'arbre doublement récursive, qui utilise la récursivité pour la descente aux fils et aussi pour le parcours de la liste des frères. Ce qui donne le programme correct et apparemment très élégant suivant :

```
void parcours(POINTEUR arbre)
    {
    if (arbre != NULL)
        {
        EXPLOITER(arbre->info);
        parcours(arbre->fils);
        parcours(arbre->frere);
        }
    }
```

A notre avis, ce programme est moins satisfaisant pour l'esprit que la version précédemment donnée, aussi bien du point de vue stylistique, car on met sur le même plan les fils et les frères, que de point de vue de l'efficacité, car on programme récursivement un traitement qui aurait pu être programmé aussi simplement de manière itérative.

2.5. Applications

2.5.1. Arbre lexicographique (codage en parties communes)

Proposons-nous d'implanter une sorte de dictionnaire, c'est-à-dire une structure de données servant à mémoriser un ensemble de mots[5]. Le dispositif réalisé doit permettre de rechercher et au besoin d'insérer des mots au fur et à mesure des besoins. On supposera qu'aucun mot ne se répète dans le dictionnaire, et que l'échec de la recherche d'un mot doit provoquer automatiquement son insertion dans le dictionnaire. Enfin, nous souhaitons disposer d'une fonction qui affiche la liste par ordre alphabétique de tous les mots contenus dans le dictionnaire.

Ce qui va suivre sera fait avec des chaînes (c'est-à-dire des tableaux) de caractères, mais s'adapte facilement à des tableaux de n'importe quelle sorte d'objets appartenant à un ensemble totalement ordonné. Le principe du dispositif consiste à maintenir un arbre, dont les nœuds ont pour informations spécifiques des caractères, défini de la manière suivante

- la racine est conventionnelle et ne porte aucune information significative ;
- les fils de la racine sont des nœuds qui portent les premières lettres des mots, rangées par ordre alphabétique ;
- soit r un des fils de la racine, c la lettre qu'il porte. Les fils de r sont des nœuds qui portent les deuxièmes lettres, rangées par ordre alphabétique, des mots commençant par c ;
- plus généralement, soit $r_0r_1r_2\ldots r_n$ un chemin depuis la racine (r_0 est la racine), $c_1c_2\ldots c_n$ les lettres respectivement rangées dans ces nœuds. Les fils de r_n sont des nœuds qui portent les $(n+1)^{\text{èmes}}$ lettres, rangées par ordre alphabétique, des mots commençant par $c_1c_2\ldots c_n$.

Par exemple, la figure 9 montre l'arbre correspondant aux mots : *main*, *mais*, *mal*, *male*, *mon*, *son*, *sono*, *sons*, *sont* :

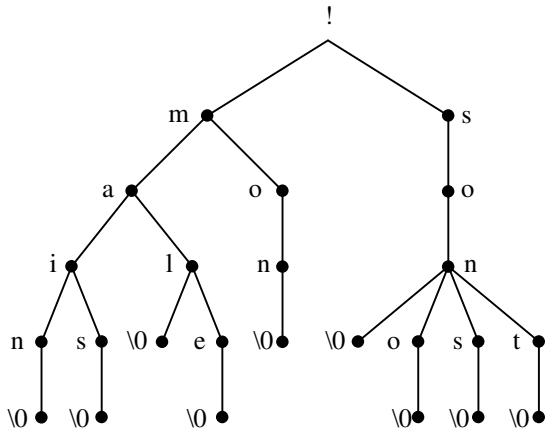


Figure 9. Arbre lexicographique

[5] En principe, un dictionnaire sert à mémoriser des mots *avec des informations associées*, mais nous négligerons ici cet aspect de la question.

On notera que les mots ne sont pas représentés par les nœuds, mais par les chemins depuis la racine. L'appellation « arbre lexicographique » est tout à fait justifiée : c'est l'arbre qui incarne l'ordre lexicographique, un nœud pris isolément n'a aucune signification.

Chaque mot se termine par un caractère spécial (en C, nous utiliserons le '\0' qui se trouve à la fin de chaque chaîne) qui

- caractérise le mot : c'est le seul caractère dont on est sûr qu'il y en a un par mot[6],
- doit être inférieur aux autres lettres (afin que "abc" soit inférieur à "abcd").

Voici le programme. Une fois de plus, le problème se ramène à celui de l'insertion dans une liste triée. Déclarations :

```
typedef struct nœud {
    char info;
    struct noeud *fils, *frere;
} NOEUD, *PTR;

PTR noeud(char info, PTR fils, PTR frere) {
    PTR p = malloc(sizeof(NOEUD));
    assert(p != NULL);
    p->info = info;
    p->fils = fils;
    p->frere = frere;
    return p;
}
```

La fonction de recherche-insertion rend 0 lorsque le mot recherché n'existait pas dans l'arbre (il y a donc eu création d'au moins un nœud pour un caractère), 1 lorsque le mot était déjà dans l'arbre (aucune création) :

```
int rechinsertion(char mot[], PTR ancetre) {
    PTR pr, pc;
    int i;

        /* à chaque tour de cette boucle on recherche le caractère */
        /* mot[i] parmi les fils du nœud pointé par ancetre  */
    for(i = 0; ; i++) {
        pr = NULL;
        pc = ancetre->fils;
        while (pc != NULL && pc->info < mot[i]) {
            pr = pc;
            pc = pc->frere;
        }

        if (pc != NULL && pc->info == mot[i]) {
            if (mot[i] == '\0')
                return 1;       /* le mot existait */
            ancetre = pc;

        }
        else {
            pc = noeud(mot[i], NULL, pc);
            if (pr != NULL)
                pr->frere = pc;
            else
                ancetre->fils = pc;

            while (mot[i] != '\0') {
                pc->fils = noeud(mot[++i], NULL, NULL);
                pc = pc->fils;
            }
            return 0;           /* le mot est nouveau */
        }
    }
}
```

[6] Si on voulait associer une information à chaque mot mémorisé dans le dictionnaire, c'est au maillon portant ce `'\0'` qu'il faudrait la raccrocher.

Affichage, dans l'ordre lexicographique, des mots de l'arbre :

```
static struct {
    char t[80];
    int n;
} pile;

void parcours(PTR arbre) {
    PTR p;

    pile.t[pile.n++] = arbre->info;
    if (arbre->info == '\0')
        printf("%s\n", pile.t + 1);
    else
        for (p = arbre->fils; p != NULL; p = p->frere)
            parcours(p);
    pile.n--;
}
```

Petit programme pour essayer tout cela :

```
main() {
    char mot[80];
    int i;

    PTR racine = noeud('!', NULL, NULL);
    while (gets(mot) != NULL)
        printf("le mot %s est %s\n", mot,
            rechinsertion(mot, racine) ? "ancien" : "nouveau");

    pile.n = 0;
    parcours(racine);
}
```

2.5.2. Sauvegarde et restauration d'arbres

Donnons-nous maintenant le problème suivant : un certain programme ayant construit un arbre, on souhaite l'enregistrer dans un fichier en vue de son exploitation ultérieure par un autre programme. Deux difficultés font que ce travail n'est pas aussi simple que la sauvegarde d'un tableau :

- les nœuds de l'arbre n'occupent pas un espace contigu, ils sont en principe dispersés dans la mémoire et il faut parcourir l'arbre pour les retrouver ;
- même si on fait en sorte que les nœuds soient contigus, une difficulté demeure : la structure d'arbre est réalisée par des pointeurs, c'est-à-dire par des adresses dans la mémoire de l'ordinateur. Copiés dans un fichier, ces pointeurs perdent toute signification, car une recharge en mémoire ultérieure de l'arbre n'a aucune raison de « tomber » dans les mêmes adresses.

Il nous faudra donc inventer une sorte de syntaxe rudimentaire pour enregistrer l'arbre sans les pointeurs, de manière qu'il soit ensuite possible de le reconstituer par le programme devant l'exploiter. Nous supposerons que nous ne pouvons écrire sur le fichier de sauvegarde que des articles qui ont une même structure, à savoir celle du champ *info* des nœuds de l'arbre. Nous postulerons l'existence d'une valeur « impossible » du champ info, c'est-à-dire une valeur Ω qui ne peut être confondue avec aucune des valeurs réellement présentes dans l'arbre.

L'idée est la suivante : pour sauvegarder l'arbre $(I , (A_1 , \ldots A_n))$:

- écrire la valeur I
- sauvegarder l'arbre A_1

  …
- sauvegarder l'arbre A_n
- écrire la valeur Ω.

Comme exemple, examinons les fonctions de sauvegarde et de restauration de l'arbre lexicographique défini à la section précédente. La partie info de chaque nœud étant un caractère, choisissons pour Ω la valeur ';' qui n'est pas une lettre possible. Voici ce qu'il faut ajouter au programme précédemment écrit :

```
#define OMEGA ';'
static FILE *fichier;
```

Sauvegarde :

```
static void sauvegardeBis(POINTEUR p)
    {
    fputc(p->info, fichier);
    for (p = p->fils; p != NULL; p = p->frere)
        sauvegardeBis(p);
    fputc(OMEGA, fichier);
    }

void sauvegarde(POINTEUR arbre, char *nomfic)
    {
    if ((fichier = fopen(nomfic, "w")) == NULL)
        erreurFatale(IMPOSSIBLE_CREER_FICHIER);
    sauvegardeBis(arbre);
    fclose(fichier);
    }
```

Restauration :

```
static char calu;
static POINTEUR restaurationBis(void)
    {
    POINTEUR p, f;
    p = noeud(calu, NULL, NULL);
    calu = fgetc(fichier);
    if (calu != OMEGA)
        {
        f = p->fils = restaurationBis();
        while (calu != OMEGA)
            {
            f->frere = restaurationBis();
            f = f->frere;
            }
        }
    calu = fgetc(fichier);
    return p;
    }

void restauration(POINTEUR *arbre, char *nomfic)
    {
    if ((fichier = fopen(nomfic, "r")) == NULL)
        erreurFatale(IMPOSSIBLE_OUVRIR_FICHIER);
    calu = fgetc(fichier);
    *arbre = restaurationBis();
    fclose(fichier);
    }
```

Essai de la sauvegarde :

```
sauvegarde(racine, "ArbreSauve");
```

Essai de la restauration :

```
restauration(&racine, "ArbreSauve");
```

Pour finir, imaginons qu'après avoir sauvegardé l'arbre donné en exemple, composé des mots *main*, *mais*, *mal*, *male*, *mon*, *son*, *sono*, *sons* et *sont* (voyez la figure 9), nous ayons bricolé une sorte de programme espion pour voir le contenu du fichier de sauvegarde, en affichant le caractère '\0' sous la forme @ . Voici ce que nous y trouverons :

```
!main@;;s@;;;l@;e@;;;;on@;;;;son@;o@;;s@;;t@;;;;;;
```

☺☹

3. Quelques arbres particuliers

3.1. Arbre maximier (tas), file de priorité

3.1.1. Arbres binaires complets et presque complets

De la même manière que les structures linéaires peuvent être implantées, avec différents avantages et inconvénients, soit par des structures chaînées, soit par des tableaux, une certaine catégorie d'arbres s'accommode très bien d'une représentation par tableaux.

Soit A un arbre binaire de hauteur k. Il est facile de prouver que

- le nombre maximum de nœuds de niveau i est 2^{i-1} ;
- le nombre maximum de nœuds de A est $2^k - 1$.

Un *arbre binaire complet* est un arbre binaire de hauteur k possédant $2^k - 1$ nœuds. Puisqu'il n'a pas de trous, un tel arbre peut être représenté par un tableau de $2^k - 1$ éléments, tout simplement en rangeant les valeurs des nœuds niveau par niveau[7] :

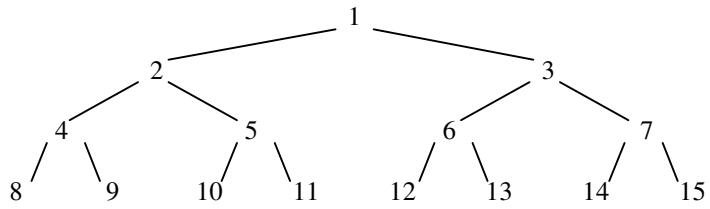


Figure 1. Arbre binaire complet

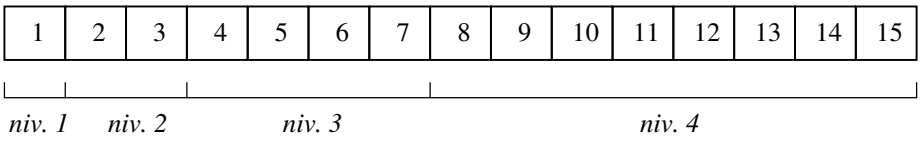


Figure 2. Arbre binaire complet représenté par un tableau

Soit N le nombre d'éléments du tableau (donc $N = 2k - 1$, k étant la hauteur de l'arbre). On a les propriétés suivantes :

- si $i \leq Ent\left(\frac{N}{2}\right)$, le fils gauche de T_i est T_{2i}, le fils droit est T_{2i+1}

[7] Exceptionnellement, nous utiliserons ici des tableaux commençant à l'indice 1 (c'est-à-dire des tableaux dont le premier élément, d'indice 0, est ignoré) car cela rend leur manipulation bien plus simple.

- si $i > Ent\left(\frac{N}{2}\right)$, T_i est une feuille
- si $j > 1$, le père de T_j est $T_{Ent\left(\frac{j}{2}\right)}$

Un *arbre binaire presque complet* est un arbre binaire dont tous les niveaux sont pleins sauf le dernier, qui ne comporte que les p nœuds de gauche (un tel arbre n'est pas complet, mais au moins il n'a pas de trous). Dans une représentation par tableau définie comme pour les arbres binaires complets, il ne remplit donc que le début du tableau :

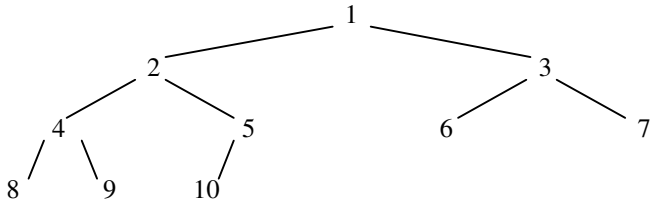


Figure 3. Arbre binaire presque complet

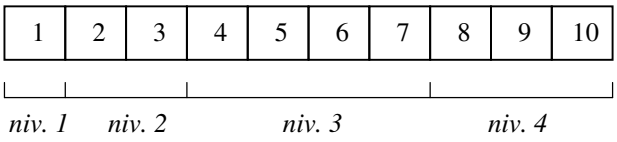


Figure 4. Arbre binaire presque complet représenté par un tableau

3.1.2. Arbres maximiers

Un *arbre maximier*, ou *tas*, est un arbre binaire presque complet, dont les nœuds ont un champ *clé* appartenant à un type ordonné, et tel que pour tout nœud p autre que la racine

$$clé(père(p)) \geq clé(p)$$

Par conséquent, la clé de la racine est le maximum des clés, d'où le nom de ces arbres. S'agissant d'arbres binaires presque complets, on les représente volontiers par des tableaux. Ainsi, un tableau $T_1, \ldots T_N$ est la représentation d'un arbre maximier si, pour tout $i = 1, \ldots N$:

- $2i \leq N \Rightarrow clé(T_{2i}) \leq clé(T_i)$;
- $2i + 1 \leq N \Rightarrow clé(T_{2i+2}) \leq clé(T_i)$.

3.1.3. Files de priorité

Les files de priorité sont une des principales applications des arbres maximiers. Nous avons déjà étudié la notion de file d'attente, caractérisée par le fait que lorsqu'un élément est extrait de la file, c'est celui qui s'y trouve depuis le plus longtemps qui est extrait. Dans une file de priorité, les éléments de la file ont un champ, nommé *priorité*, dont les valeurs appartiennent à un type ordonné. Le comportement d'une file de priorité est défini par la propriété suivante : lors d'une extraction, c'est l'élément ayant la plus grande priorité qui est extrait.

Le rapport avec les arbres maximiers (prenant pour clé la priorité) est facile à voir : la racine d'un maximier est toujours l'élément ayant la plus grande priorité, donc le prochain qui devra être extrait. L'intérêt de cette manière d'implanter les files de priorité réside dans son efficacité :

- l'effort nécessaire pour extraire la racine d'un maximier (ou, plus exactement, pour refaire un maximier à partir d'un maximier décapité) est de l'ordre de $\log_2 N$;
- l'effort nécessaire pour introduire un élément dans un maximier, en le plaçant à l'endroit qui lui revient selon sa priorité, est de l'ordre de $\log_2 N$.

Les algorithmes correspondants sont assez naturels : pour introduire un élément dans un arbre maximier, on le copie à la fin du tableau (dont le nombre d'éléments passe de *N* à *N*+1), ce qui en fait le fils de quelqu'un, puis on le compare à son père. Si la relation qui doit exister entre un nœud et ses fils est satisfaite, l'insertion est terminée ; sinon, on permute le nouveau nœud et son père, ce qui fait du nouvel élément le fils d'un autre nœud, et on recommence.

Pour faire sortir la racine d'un maximier on fait à peu près le même travail, dans le sens contraire : on bouche le trou laissé par le nœud extrait avec le dernier élément du tableau (le nombre d'éléments de celui-ci passe de N à N-1), puis on vérifie si le nœud en question est bien placé par rapport à ses fils. Si c'est le cas on s'arrête, sinon on le permute avec le plus grand des deux fils, et on recommence. Voici les programmes correspondants.

Nous supposons avoir défini une macro PRIOR(x) donnant accès au champ priorité d'un élément (peut-être l'élément tout entier), et nous supposons que ce champ est d'un type supportant les opérateurs <, <= .

Ajout d'un élément à la file de priorité :

```
void entrer(OBJET x)
    {
    int i, j;

    assert(N < TAILLE);
    N++;

    j = N;
    while (j > 1)
        {
        i = j / 2;
        if (PRIOR(t[i]) >= PRIOR(x))
            break;
        t[j] = t[i];
        j = i;
        }
    t[j] = x;
    }
```

Extraction de l'élément dont c'est le tour :

```
OBJET sortir(void)
    {
    OBJET x, y;
    int i, j;

    assert(N > 0);
    x = t[1];
    y = t[N];
    N--;

    i = 1;
    while ((j = 2 * i) <= N)
        {
        if (j + 1 <= N && PRIOR(t[j + 1]) > PRIOR(t[j]))
            j++;
        if (PRIOR(t[j]) < PRIOR(y))
            break;
        t[i] = t[j];
        i = j;
        }
    t[i] = y;

    return x;
    }
```

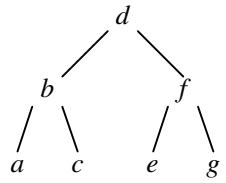
3.2. Arbres binaires équilibrés

3.2.1. Arbre binaire de recherche

Un arbre binaire de recherche (*ABR*) est un arbre binaire *A* dont les nœuds ont un champ *clé*, éventuellement le champ *info* tout entier, dont les valeurs appartiennent à un ensemble totalement ordonné, et qui vérifie, pour tout nœud *r* de *A* :

- pour tout nœud *r'* appartenant au sous-arbre gauche de *r* : $r'.cle \leq r.cle$
- pour tout nœud *r"* appartenant au sous-arbre droit de *r* : $r".cle > r.cle$

L'arbre binaire de recherche correspondant à un ensemble de clés n'est pas unique : les deux ABR représentés dans la figure 5 correspondent aux mêmes clés. La structure précise de l'ABR est déterminée par l'algorithme d'insertion utilisé et par l'ordre d'arrivée des éléments.



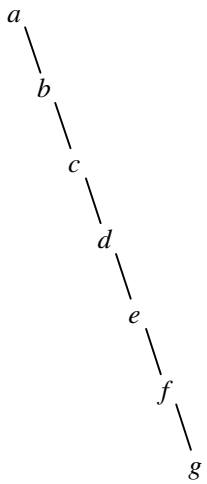


Figure 5 (ci-dessus et ci-contre) . Deux ABR avec les mêmes clés

Déclarations :

```
typedef char *CLE;

#define COMPAR(a, b) strcmp((a), (b))

typedef struct nœud
    {
    CLE cle;
    struct noeud *fg, *fd;
    } NOEUD, *POINTEUR;
```

La fonction *rechInsertion* recherche la clé *x* dans l'ABR pointé par *rac*. Si cette recherche est infructueuse, elle crée un maillon nouveau portant la clé *x* et l'insère dans l'arbre à la place voulue.

La version itérative d'une telle fonction (cf. TD) est claire et efficace. Pour fixer les idées, nous en donnons ici une version récursive dont l'intérêt se manifestera ultérieurement.

```
void rechInsertion(CLE x, POINTEUR *rac)
    {
    int c;

    if (*rac == NULL)
        *rac = noeud(x, NULL, NULL);
    else
        if ((c = COMPAR(x, (*rac)->cle)) == 0)
            EXPLOITER(*rac)
        else
            if (c < 0)
                rechInsertion(x, &(*rac)->fg);
            else
                rechInsertion(x, &(*rac)->fd);
    }
```

3.2.2. Arbres équilibrés au sens des hauteurs (AVL)

L'intérêt des ABR réside dans le fait que, s'ils sont équilibrés, c'est-à-dire si chaque nœud a à peu près autant de descendants gauches que de descendants droits, alors

- la recherche d'un élément est dichotomique (i.e. : à chaque étape la moitié des clés restant à examiner sont éliminées), et demande donc $\log_2 n$ itérations ;
- l'insertion consécutive à une recherche a un coût nul.

Cela à la condition que l'arbre soit équilibré, car si on le laisse se déséquilibrer, les performances peuvent se dégrader significativement, jusqu'au cas tout à fait dégénéré de la figure 5, où l'arbre devient une liste chaînée, et la recherche devient séquentielle.

Nous allons examiner une méthode classique pour maintenir équilibré un arbre binaire de recherche. Auparavant, il faut préciser de quel équilibre on parle :

- un arbre *parfaitement équilibré* est tel que pour chacun de ses nœuds *r* on a, à l'unité près :

$$cardinal\ (\ sousArbreGauche(\ r\ )\ ) = cardinal\ (\ sousArbreDroit(\ r\ )\ )$$

- un arbre *équilibré au sens des hauteurs*, ou *AVL*, est tel que pour chaque nœud *r* on a, à l'unité près :

$$hauteur\ (\ sousArbreGauche(\ r\ )\ ) = hauteur\ (\ sousArbreDroit(\ r\ )\ )$$

L'effort à fournir pour garder un arbre parfaitement équilibré est très important. On se contente donc de l'équilibre des hauteurs, après avoir constaté que ce n'est pas un compromis trop mauvais : Adelson-Velskii et Landis (à l'origine de l'appellation *AVL*) ont montré que la hauteur d'un arbre binaire équilibré ne dépasse jamais de plus de 45% la hauteur d'un arbre parfaitement équilibré ayant les mêmes clés[8].

Nous allons donc écrire une fonction de recherche et insertion qui maintiendra l'arbre équilibré. Pour cela, nous allons augmenter la fonction *rechInsertion* récursive déjà écrite.

Pour faire une insertion, la première version de la fonction *rechInsertion* fait une descente le long des branches de l'arbre, à la recherche de l'emplacement où le nouveau nœud se trouve ou doit être accroché. Nous lui ajouterons un travail à faire pendant la remontée vers la racine : examiner si, du fait de la création qui vient d'être faite, un nœud n'est pas devenu déséquilibré. Voici comment cela se présentera :

- après la création d'un nouveau nœud, durant la remontée vers la racine, on réagira sans tarder dès la rencontre d'un nœud déséquilibré ;
- la rencontre d'un nœud déséquilibré déclenchera une réparation locale (bricolage de quelques pointeurs appartenant à quelques nœuds voisins) avec deux conséquences :
  - dans le sous-arbre dont le nœud déséquilibré était la racine, il n'y aura plus de déséquilibre,
  - ce sous-arbre, dont la hauteur avait augmenté (puisqu'il était devenu déséquilibré) sera revenu à la hauteur qu'il avait avant l'insertion, ce qui garantira l'équilibre de tous ses ancêtres.

L'équilibre d'un nœud est l'égalité des hauteurs de ses deux sous-arbres à l'unité près. Si un nœud devient déséquilibré (la hauteur d'un côté dépasse d'*au moins deux* unités la hauteur de l'autre côté) à la suite d'une insertion, c'est que avant cette insertion le nœud « penchait » déjà de ce côté. Savoir si un nœud penche d'un côté ou de l'autre est donc une information capitale. Pour en disposer, nous ajouterons un champ *bal* à chaque nœud, ainsi défini[9] :

$$r.bal = \begin{cases} -1, si\ hauteur(sousArbreGauche(r)) = hauteur(sousArbreDroit(r)) + 1 \\ 0, si\ hauteur(sousArbreGauche(r)) = hauteur(sousArbreDroit(r)) \\ +1, si\ hauteur(sousArbreGauche(r)) = hauteur(sousArbreDroit(r)) - 1 \end{cases}$$

Commençons par déterminer les cas possibles. La situation est la suivante : après avoir descendu le long d'une branche de l'arbre, nous avons créé un nœud nouveau, nous l'avons accroché à une feuille, et nous avons entrepris de remonter vers la racine en revisitant chaque nœud par lequel nous étions passés lors de la descente, à la recherche du premier nœud que cette création a rendu déséquilibré.

Soit *p* ce nœud, supposons que l'ajout du nœud nouveau s'est faite dans son sous-arbre gauche ; le déséquilibre se traduit donc par $hauteur(sousArbreGauche(p)) = hauteur(sousArbreDroit(p)) + 2$, comme le montre la figure 6.

---

8 C'est la hauteur plus ou moins grande d'un *ABR* qui rend la recherche moyenne plus ou moins longue.

9 On pourrait aussi ne pas mémoriser ce renseignement et le calculer chaque fois que cela est nécessaire. Mais le programme deviendrait ainsi beaucoup plus lent.

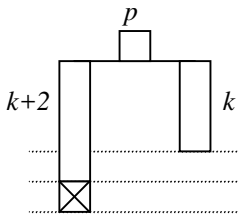


Figure 6. « p », le nœud déséquilibré le plus bas

Bien entendu, il y a un autre cas possible, rigoureusement symétrique de celui-ci, nous n'en parlerons pas (mais nous le rencontrerons dans le programme final).

Examinons de plus près le sous-arbre gauche de *p*. Compte tenu du fait qu'on vient de faire la création d'un seul nœud, avant laquelle l'arbre était équilibré, il n'y a que deux configurations possibles:

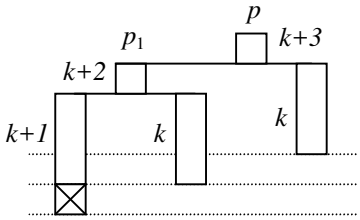


Figure 7 (a) Type I

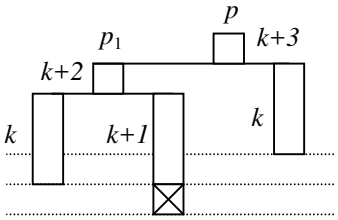


Figure 7 (b) Type II

La réparation d'un nœud de type I s'appelle *rotation simple* :

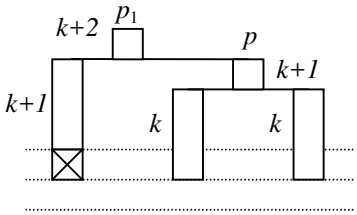


Figure 8. Rotation simple

Pour la réparation d'un nœud de type II il faut examiner le fils droit de p_1 (cf. figure 9). Notez que, des deux carrés barrés de cette figure, un seul existe (sinon l'arbre aurait été déséquilibré lors de l'insertion précédente), mais peu importe lequel est-ce.

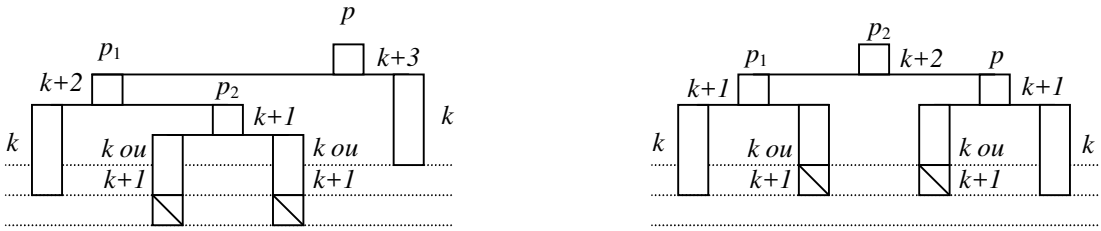


Figure 9. Double rotation

On vérifie bien que, comme annoncé, la hauteur du sous-arbre déséquilibré après la correction du déséquilibre est la même que celle qu'avait le sous-arbre avant l'insertion.

Déclarations :

```
typedef char *CLE;

typedef struct nœud
    {
    CLE cle;
    int bal;
    struct noeud *fg, *fd;
    } NOEUD, *POINTEUR;
```

La nouvelle fonction *rechInsertion* possède un argument supplémentaire. Un appel comme

```
rechInsertion(cle, arbre, &attention);
```

recherche et au besoin crée un nœud portant la *cle* indiquée, dans l'*arbre* indiqué. Au retour, *attention* est vrai si cette opération a augmenté la hauteur de l'arbre (il y a donc lieu d'examiner l'état d'équilibre de sa racine).

```
void rechInsertion(CLE x, POINTEUR *p, int *attention)
    {
    int c;
    POINTEUR p1, p2;

    if (*p == NULL)
        {
        *p = noeud(x, 0, NULL, NULL);
        *attention = 1;
        }
    else
        if ((c = COMPAR(x, (*p)->cle)) == 0)
            {
            EXPLOITER(*p)
            *attention = 0;
            }
        else
            if (c < 0)
                {
                rechInsertion(x, &(*p)->fg, attention);

                if (*attention)
                    if ((*p)->bal == 0)
                        (*p)->bal = -1;
                    else
                        if ((*p)->bal > 0)
                            {
                            (*p)->bal = 0;
                            *attention = 0;
                            }
                        else

                            {
                            p1 = (*p)->fg;
                            if (p1->bal < 0)          /* type I */
                                {
                                (*p)->fg = p1->fd;
                                p1->fd = *p;
                                (*p)->bal = 0;
                                *p = p1;
                                }
                            else                      /* type II */
                                {
                                p2 = p1->fd;
                                p1->fd = p2->fg;
                                (*p)->fg = p2->fd;
                                p2->fg = p1;
                                p2->fd = *p;
                                if (p2->bal < 0)
                                    p1->bal = 0, (*p)->bal = 1;
                                else
                                    p1->bal = -1, (*p)->bal = 0;
                                *p = p2;
                                }
                            (*p)->bal = 0;
                            *attention = 0;            /* le problème est éliminé */
                            }
                }
            else
                {
                Cas symétrique du précédent : échangez fg et fd, < et >, -1 et +1.
                }
    }
```

Pour finir, voici deux tranches de la vie des AVL. La figure 10 montre une suite de rotations simples, provoquées par l'insertion successive, dans un arbre vide, de la suite de clés (*a*, *b*, *c*, *d*, *e*, *f*, *g*) (il s'agit justement la suite qui produit l'arbre dégénéré de la figure 5) :

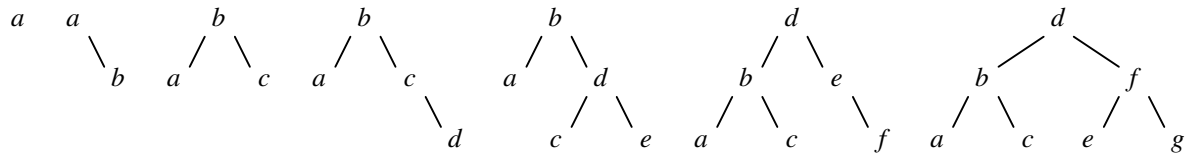


Figure 10. Suite de rotations simples

La figure 11 montre une double rotation provoquée par l'insertion de la clé *g* dans l'arbre de la figure 11 (*a*) :

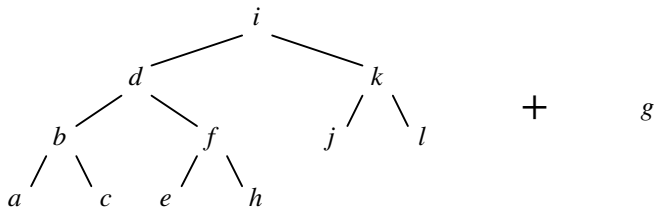


Figure 11 (a) Brrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr...

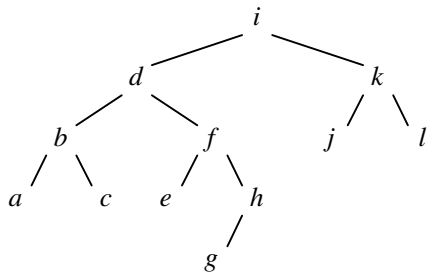


Figure 11 (b) ... et ...

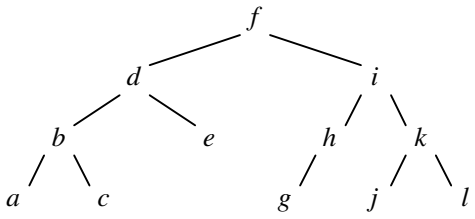


Figure 11 (c) ... hop !

3.3. B-arbres

3.3.1. Position du problème

Comme dans les sections précédentes, nous nous intéressons à la recherche rapide d'un élément d'un ensemble de clés, chacune associée à une information, qui dépend de l'application particulière considérée.

Nous supposons maintenant que la taille totale de cet ensemble est si importante qu'il est hors de question de le conserver dans la mémoire primaire de l'ordinateur. Il se trouve donc sur une mémoire secondaire, en général un fichier sur disque magnétique. Ce fichier est constitué d'enregistrements, chacun pouvant être lu (chargé dans la mémoire) ou écrit (enregistré dans le fichier) indépendamment des autres, à partir de son rang dans le fichier.

Il y a ici une différence importante avec les problèmes étudiés précédemment : jusqu'à présent, pour faire vite on cherchait à optimiser l'ensemble des opérations effectuées. Or le transfert d'un enregistrement entre la mémoire secondaire et la mémoire primaire (ce qu'on appelle un « accès disque ») est tellement plus lent que les opérations internes, qu'il suffit de minimiser le nombre de ces transferts : tout gain de temps sur des opérations internes est négligeable devant le temps requis pour un accès disque.

Un premier cas simple est celui où le problème est posé par les informations associées, non par les clés. L'ensemble des clés est assez peu volumineux pour tenir dans la mémoire primaire. Il suffit alors de maintenir une table, appelée *table d'index*, dans laquelle chaque clé est associée au numéro de l'enregistrement la concernant. Cette table est gérée par une des méthodes expliquées dans les sections précédentes. L'accès à l'information correspondant à une clé donnée se fait en deux temps :

- ayant la clé, rechercher dans la table le rang de l'article correspondant,
- ayant son rang, transférer l'article en question depuis ou vers le fichier.

Ici nous allons nous intéresser plutôt au cas où l'ensemble des clés est lui-même beaucoup trop volumineux pour pouvoir être conservé dans la mémoire principale.

3.3.2. B-arbres

Nous avons remarqué l'intérêt des arbres binaires équilibrés. Pour trouver une clé dans un arbre binaire d'un million de nœuds, il faut seulement une vingtaine d'accès aux nœuds de l'arbre, puisque $2^{20} \sim 10^6$. Mais si chaque accès à un nœud implique le chargement d'un enregistrement d'un fichier sur disque magnétique, cela fait encore beaucoup trop.

Les B-arbres sont une manière efficace, élégante et assez simple de conserver les avantages des arbres équilibrés tout en les adaptant aux caractéristiques particulières des fichiers. Bien sûr, l'idée de départ est de gonfler les nœuds et de leur faire porter plus d'une clé. Par exemple, avec des nœuds d'une capacité maximale de 100 clés, un B-arbre contenant un million de clés aura entre 10.000 et 20.000 nœuds et pour retrouver une clé il faudra 3,2 accès à un nœud en moyenne, et 3,5 dans le pire des cas.

La structure d'un B-arbre est la généralisation de celle d'un *ABR* : à la place d'une clé et deux pointeurs, chaque nœud porte k clés $c_0, c_1, \ldots c_{k-1}$ et k+1 pointeurs $p_{-1}, p_0, \ldots p_{k-1}$. Les clés sont triées par ordre croissant. Le pointeur p_i renvoie à un nœud qui est la racine d'un sous-arbre A_i contenant les clés c qui vérifient $c_i < c < c_{i+1}$.

Cas particuliers, p_{-1} pointe le sous-arbre des clés vérifiant $c < c_0$ et p_{k-1} pointe le sous-arbre des clés vérifiant $c_{k-1} < c$ (en réalité, ces clés extrêmes ont elles aussi des bornes à droite et à gauche, héritées du père du nœud en question).

Les nœuds des B-arbres sont appelés *pages* ; chacune correspond à un enregistrement du fichier sous-jacent. Lorsque la capacité d'une page (la valeur maximum de k) est $2N$, on dit que l'on a affaire à un B-arbre de degré N. L'algorithme de manipulation que nous allons présenter garantit que

- chaque page, sauf la racine, porte au moins N clés ;
- une page qui porte k clés et qui n'est pas une feuille possède k+1 filles ;
- toutes les feuilles sont au même niveau ; du point de vue de la hauteur, l'arbre est donc équilibré.

INSERTION D'UNE CLE. La procédure d'insertion est responsable du maintien des propriétés ci-dessus. Comme pour les arbres binaires, une insertion se produit à la suite d'une recherche infructueuse qui a consisté à descendre le long d'une branche de l'arbre, jusqu'à une feuille qui a montré que la clé cherchée n'existait pas. Deux cas sont alors possibles :

- La feuille en question n'est pas saturée ($k < 2N$) : il s'agit d'une insertion ordinaire dans une table triée.
- La feuille est pleine ($k = 2N$). On a représenté par J l'éventuel pointeur associé à la clé j :

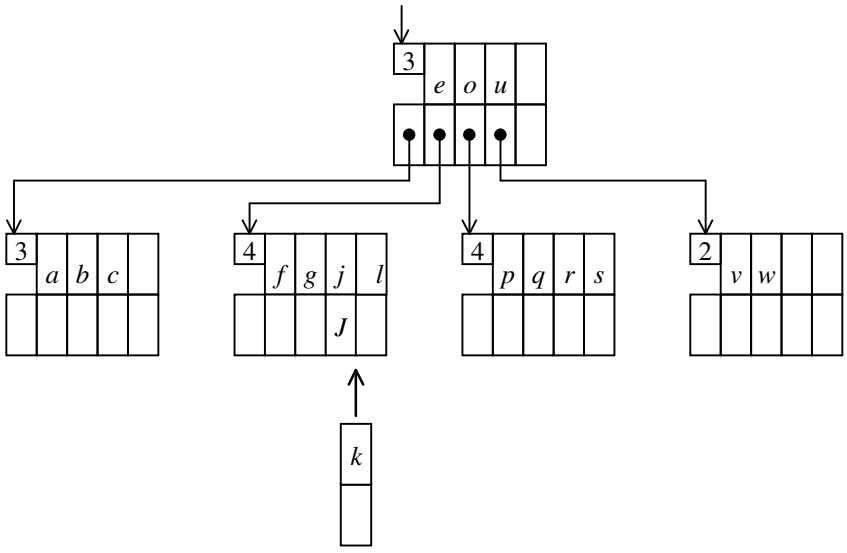


Figure 12 (a) Insertion dans une page pleine

On effectue la scission de la page, qui produit deux pages à moitié pleines et un élément à insérer dans la page mère :

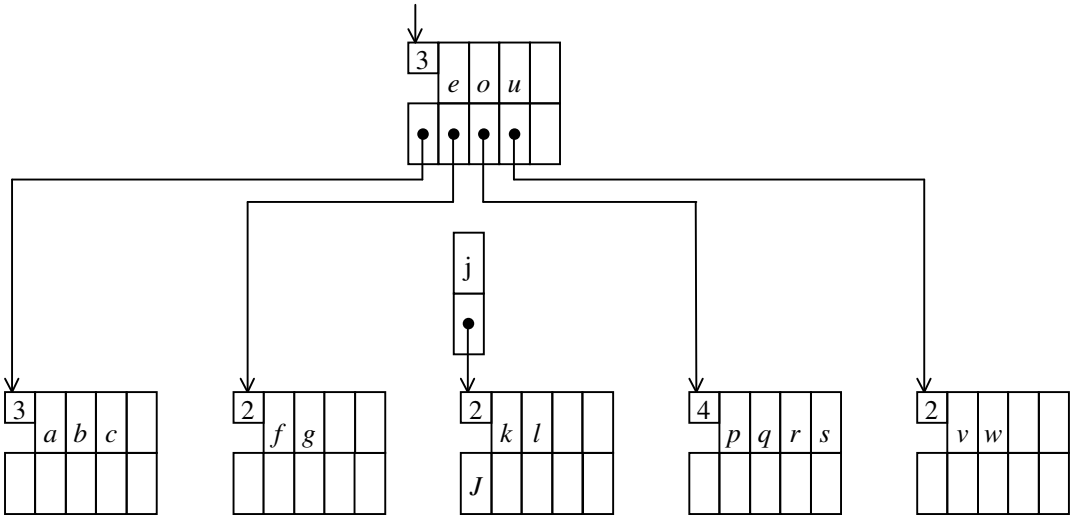


Figure 12 (b) Scission de page, début.

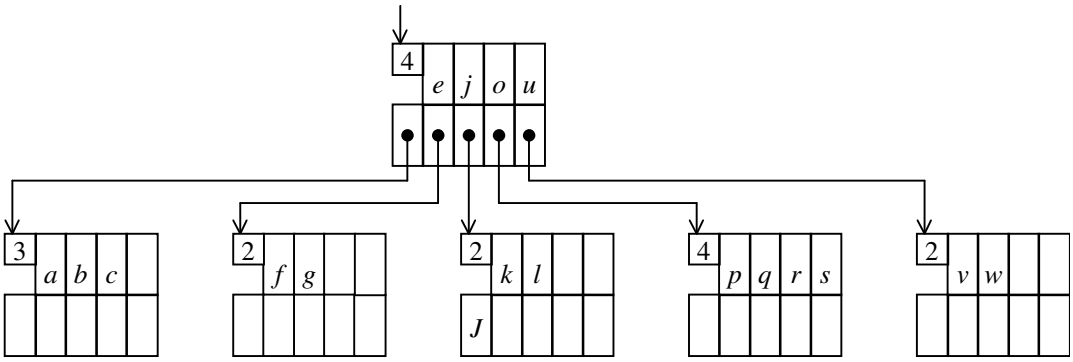


Figure 12 (c) Scission de page, fin.

Dans le partage d'une page à $2N$ clés en deux pages à N clés, chaque clé voyage avec son information associée. Un petit trafic de pointeurs a lieu, mais on vérifie aisément que c'est celui qu'il faut pour que les propriétés du B-arbre soient vérifiées : la clé qui « s'en va » cède sa descendance éventuelle à la nouvelle page, à titre de pointeur p_{-1}, et elle prend pour descendance la nouvelle page. Bien entendu, si la page mère est pleine elle aussi, alors elle se scindera à son tour en deux pages à demi pleines et fera remonter un élément vers sa propre page mère, etc.

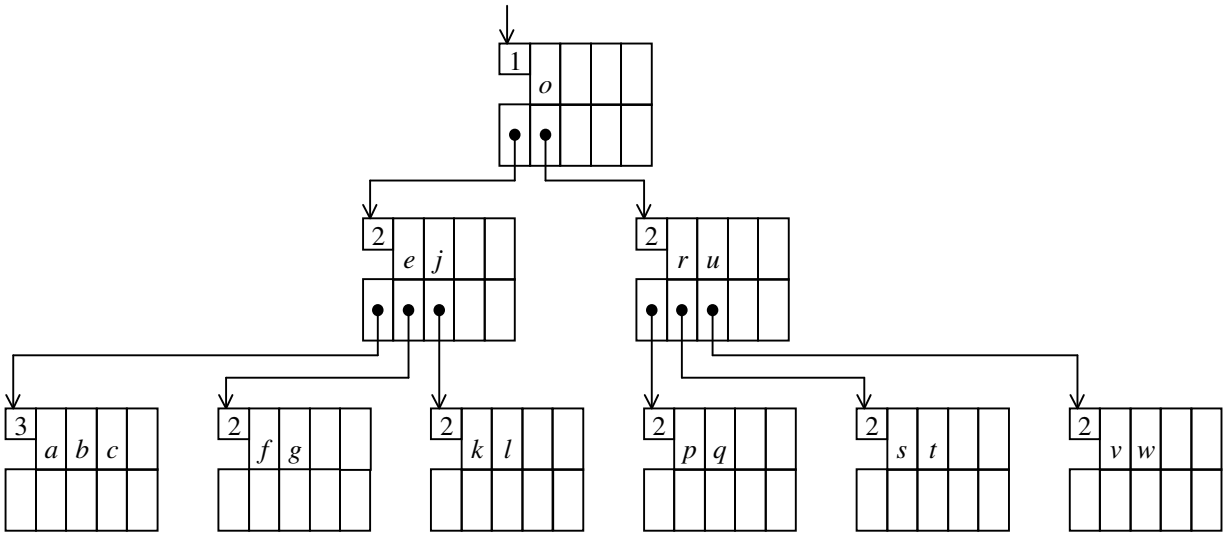


Figure 13. Scission de la racine

Cas particulier : si la page qui est pleine n'a pas de mère, c'est-à-dire s'il s'agit d'une insertion dans la racine (saturée), alors cette dernière se divise en deux et il y a création d'une nouvelle racine avec une unique clé et deux descendants. Ainsi, par exemple, la figure 13 montre le résultat d'une scission de la racine, provoquée par l'insertion de la clé *t* dans le B-arbre de la figure 12.

Les B-arbres ont une curieuse façon de croître : ce sont les feuilles qui font pousser l'arbre du côté de la racine !

3.3.3. Implantation des B-arbres

Ce que nous avons appelé *pointeurs* dans les explications précédentes sont en fait des renvois à des enregistrements d'un fichier permettant l'accès relatif, c'est-à-dire des nombres exprimant les rangs de ces enregistrements. L'opération fondamentale d'un pointeur ordinaire, l'accès à la variable pointée, se traduira ici par un travail beaucoup plus important, le chargement en mémoire de l'enregistrement en question. Pour ne pas entrer dans les détails de l'accès aux fichiers, nous supposerons disposer des trois primitives suivantes :

```
void charger(PAGE *adrPage, long rang);
```

cette fonction copie dans la page (en mémoire) pointée par *adrPage* la composante du fichier ayant le rang indiqué ;

```
void ranger(PAGE *adrPage, long rang);
```

cette fonction copie la page (en mémoire) pointée par *adrPage* sur la composante du fichier ayant le rang indiqué.

```
long allouerPage(void);
```

cette fonction détermine et renvoie le *rang* d'un emplacement libre dans le fichier. Nous supposerons que cette opération réussit toujours.

Nous postulerons enfin qu'une composante d'un fichier ne peut pas avoir le rang -1. Cette valeur tiendra donc ici le rôle du pointeur impossible (NULL).

Déclarations globales.

```
typedef … CLE;              /* dépend de l'application */
#define N …                 /* le degré du B-arbre */
typedef struct item
    {
    CLE cle;
    long p;
    ...                     /* autres informations associées aux clés */
    } ITEM;
typedef struct page
    {
    int k;
    long p_1;               /* le pointeur p[-1] */
    ITEM t[2 * N];
    } PAGE;
#define NIL (-1)
```

Nous supposons posséder par ailleurs la fonction

```
int existe(CLE cle, PAGE *page, int *pos)
```

qui cherche la *cle* indiquée dans la *page* indiquée. La fonction rend 1 si la clé s'y trouve (**pos* renvoie alors à son indice), 0 si elle ne s'y trouve pas (**pos* renvoie alors à l'indice où il faudrait l'insérer pour respecter l'ordre) ; il s'agit de la recherche dichotomique dans une table triée.

RECHERCHE. La fonction suivante rend 1 ou 0 pour traduire le succès ou l'échec de la recherche de la cle indiquée dans l'arbre dont la racine est la page ayant l'adresse indiquée. Cette fonction peut s'écrire aussi simplement de manière itérative ; nous en donnons une version récursive en vue de la fonction de recherche-insertion qui va suivre et qui, elle, gagnera à être récursive.

```
int recherche(CLE cle, long adresse)
    {
    PAGE A;
    int i;
    if (adresse == NIL)
        return 0;
```

```
    else
        {
        charger(&A, adresse);
        if (existe(cle, &A, &i))
            {
            EXPLOITER(A.t[i])
            return 1;
            }
        else
            return recherche(cle, i == 0 ? A.p_1 : A.t[i - 1].p);
        }
}
```

Remarque sur la variable A. Déclarée comme ci-dessus, c'est-à-dire locale à la fonction de recherche, il existe à un moment donné autant d'exemplaires de la variable *A* que d'activations de la fonction. Il ne faut pas que cela dépasse la capacité de la mémoire principale.

Nous obtenons ainsi une limitation pour la taille *N* des pages, et un mode d'estimation de la valeur adaptée aux besoins et possibilités d'une application particulière. Soit *T* la taille globale de l'ensemble des clés, *N* le degré du B-arbre. Dans le cas le plus défavorable (les pages ne sont remplies qu'à 50%) la hauteur de l'arbre est

$$pr_N = \frac{\log T}{\log N} - 1$$

Dans la recherche-insertion d'un élément on peut avoir jusqu'à pr_N activations de la fonction, chacune ayant alloué sa propre page; une page auxiliaire est parfois nécessaire dans le cas de l'insertion. L'encombrement total sera donc de

$$e_N \cong 2N \times sizeof(ITEM) \times \frac{\log T}{\log N}$$

Nous devrons donc prendre la plus grande valeur de *N* telle que e_N soit inférieur ou égal à l'espace disponible. Nous aurons ainsi la meilleure valeur de pr_N.

RECHERCHE-INSERTION. Cette opération est assurée par deux fonctions complémentaires :

```
void rechInser(CLE cle, long adresse, int *att, ITEM *v);
```

cette fonction recherche et au besoin insère la *cle* donnée dans l'arbre dont la racine est la page ayant l'*adresse* indiquée. Au retour, **att* est vrai lorsque cette insertion a projeté « vers le haut » un item, qu'il fait insérer dans la page mère ; **v* est alors cet item ;

```
void inserItem(PAGE *p, ITEM u, int pos, int *att, ITEM *v);
```

cette fonction insère l'item *u*, à la position indiquée par *pos*, dans la page (en mémoire) pointée par *p*. Au retour, **att* est vrai si cette opération a projeté vers le haut un item, qu'il faut insérer dans la page mère ; **v* est alors cet item.

Voici les textes de ces fonctions :

```
void inserItem(PAGE *, ITEM, int, int *, ITEM *);

void rechInser(CLE cle, long adresse, int *att, ITEM *v)
    {
    PAGE A;
    int i;

    if (adresse != NIL)
        {
        charger(&A, adresse);
        if (existe(cle, &A, &i))
            {
            EXPLOITER(A.t[i])
            *att = 0;
            }
        else
```

```
            {
            rechInser(cle, i == 0 ? A.p_1 : A.t[i - 1].p, att, v);
            if (*att)
                {
                inserItem(&A, *v, i, att, v);
                ranger(&A, adresse);
                }
            }
        }

    else
        {                                /* un item nouveau est traité */
        v->cle = COPIE(cle);             /* comme un item produit par  */
        v->p = NIL;                      /* la scission d'une page     */
        *att = 1;                        /* (fictive)                  */
        }
    }

void inserItem(PAGE *p, ITEM u, int pos, int *att, ITEM *v)
    {
    int i;

    if (p->k < 2 * N)                    /* insertion sans problème */
        {
        for (i = p->k; i > pos; i--)
            p->t[i] = p->t[i - 1];
        p->t[pos] = u;
        p->k++;
        *att = 0;
        }

    else                                 /* scission de page (aïe...!)   */
        {
        PAGE B;
        long adresseB = allouerPage();
                                         /* le fatras qui suit fait la   */
        if (pos <= N)                    /* distribution sur deux pages  */
            {                            /* des clés en jeu              */
            if (pos == N)
                *v = u;
            else
                {
                *v = p->t[N - 1];
                for (i = N - 1; i > pos; i--)
                    p->t[i] = p->t[i - 1];
                p->t[pos] = u;
                }
            for (i = 0; i < N; i++)
                B.t[i] = p->t[i + N];
            }
        else
            {
            *v = p->t[N];
            for (i = N + 1; i < pos; i++)
                B.t[i - (N + 1)] = p->t[i];
            B.t[pos - (N + 1)] = u;
            for (i = pos; i < 2 * N; i++)
                B.t[i - N] = p->t[i];
            }
        p->k = B.k = N;
        B.p_1 = v->p;
        v->p = adresseB;
                                         /* on sauve la page B (A le sera */
        ranger(&B, adresseB);            /* par la fonction appelante)    */
        *att = 1;
        }
    }
```

L'appel initial de *rechInser* se fait de la manière suivante (appel, éventuellement suivi de la scission de la racine) :

```
long adRacine;                          /* rang de la page qui est */
                                        /* la racine du B-arbre)   */
void inserer(CLE cle)
    {
    PAGE a;
    ITEM u;
    int scission;
    long r;

    rechInser(cle, adRacine, &scission, &u);
    if (scission)
        {
        r = allouerPage();
        a.p_1 = adRacine;
        a.k = 1;
        a.t[0] = u;
        ranger(&a, r);
        adRacine = r;
        }
    }
```

☺☺

4. Recherche rapide

4.1. Recherche rapide

Nous nous donnons le problème suivant : organiser une collection d'informations, chacune associée à une clé qui l'identifie de manière unique, de façon à pouvoir retrouver (sous-entendu : le plus vite possible) une clé et l'information qui lui est associée. Nous supposerons que les clés sont des chaînes de caractères.

Dans les applications, ce problème apparaît sous deux variantes principales :

- les dispositifs de type *fichier indexé*, où la clé est l'accès biunivoque à une information beaucoup plus volumineuse, qui se trouve généralement dans une mémoire secondaire ;
- les dispositifs de type *dictionnaire*, où l'ensemble de la structure se trouve en mémoire centrale, car l'information associée à chaque clé est peu volumineuse. Exemple : le dictionnaire que gère un compilateur, contenant les identificateurs du programme en cours de compilation.

Le mécanisme réalisé doit permettre les opérations de recherche, insertion et suppression d'un couple (*clé* , *info*). On cherche à optimiser la recherche sans pénaliser l'insertion. La suppression, parfois sans objet, est rarement optimisée.

4.1.1. Méthodes connues

RECHERCHE SEQUENTIELLE

Recherche. La fonction suivante renvoie 1 ou 0 selon que la valeur *x* figure ou non dans le tableau *t* de *n* éléments. En cas de succès, la fonction range dans la variable pointée par *rang* la valeur de l'indice de l'élément dans le tableau.

```
int existe(CLE x, CLE t[], int n, int *rang)
    {
    int i;
    for (i = 0; i < n; i++)
        if (COMPAR(t[i], x) == 0)
            {
            *rang = i;
            return 1;
            }
    return 0;
    }
```

Recherche avec insertion :

```
    if (existe(x, t, n, &position))
        EXPLOITER(table[position])
    else
        t[n++] = COPIE(x);
```

N.B. Le type *CLE* et la fonction *COMPAR* sont jumelés. La fonction *COPIER* dépend en plus de la nature du tableau *t*. Exemple (très critiquable, car pour bien faire il faudrait tester la réussite de l'appel de *malloc*) :

```
typedef char *CLE;
#define COMPAR(a, b) strcmp(a, b)

#define COPIE(x) strcpy(malloc(strlen(x) + 1), x)
```

Coût moyen

- d'une recherche : $\frac{n}{2}$
- d'une insertion : 1 (autant dire : rien)

RECHERCHE DICHOTOMIQUE

Principe : le tableau est trié ; à chaque étape, on le découpe en deux moitiés et, en comparant la valeur cherchée avec la valeur de l'élément médian, on élimine celle des deux moitiés qui ne peut pas contenir la valeur cherchée.

La fonction suivante renvoie 1 ou 0 selon que la valeur *x* figure ou non dans le tableau trié *t* de *n* éléments. La variable pointée par *rang* est affectée par le rang de l'élément cherché, si ce dernier existe dans le tableau, sinon elle est affectée par le rang où il faudra ranger *x* si on veut l'insérer en conservant le tableau trié :

```
int existe(CLE x, CLE t[], int n, int *rang)
    {
    int i, j, k;

    i = 0;
    j = n - 1;              /* assertion: que x soit présent ou absent */
    while (i <= j)          /* son rang r vérifie: i <= r <= j + 1     */
        {
        k = (i + j) / 2;
        if (COMPAR(x, t[k]) <= 0)
            j = k - 1;
        else
            i = k + 1;
        }
                            /* ici, en outre:  i = j + 1 */
    *rang = i;
    return i < n && COMPAR(t[i], x) == 0;
    }
```

Recherche avec insertion :

```
if (existe(x, t, n, &position))
    EXPLOITER(table[position])
else
    {
    for (i = n; i > position; i--)
        t[i] = t[i - 1];
    t[position] = COPIE(x);
    n++;
    }
```

Coût moyen

- d'une recherche : $\log_2 n$
- d'une insertion : $\frac{n}{2}$

En conclusion, la recherche dichotomique est une excellente méthode de recherche, mais les insertions coûtent cher.

ARBRE BINAIRE DE RECHERCHE, ARBRE AVL

Les arbres binaires de recherche offrent l'efficacité de la recherche dichotomique (du moins si l'on prend soin de les garder équilibrés, comme les *AVL*) sans le coût de l'insertion dans un tableau ordonné. Les AVL font l'objet du § 3.2.

ARBRE LEXICOGRAPHIQUE

Citons, pour mémoire, l'arbre lexicographique développé au § 2.5.1. Son principal avantage réside dans le fait qu'une recherche prend un temps qui ne dépend que de la longueur de la clé, non du nombre de clés.

Exemple (théorique) : avec des clés longues de 8 caractères, si on suppose que pour chaque caractère il y a 6 choix en moyenne, on pourra gérer 6^8 (~ 1,7 millions) clés, alors qu'une recherche ou une insertion moyenne ne nécessitera que $3 \times 8 = 24$ opérations.

L'inconvénient majeur de l'arbre lexicographique est le sur-encombrement créé par l'information de service (les pointeurs *fils* et *frere*). Ainsi, la partie « solitaire » d'un mot, c'est-à-dire les caractères qui ne sont pas partagés avec d'autres mots, occupe au moins neuf octets par caractère. Selon la nature de l'ensemble de clés manipulées, cette dépense peut se révéler prohibitive.

4.1.2. Adressage dispersé (hash-code)

Les méthodes de la famille « *hash-code* », ou *adressage dispersé*, sont les plus employées, car elles dépassent en efficacité toutes les précédentes.

L'idée de base est la suivante : pour trouver une clé donnée dans un tableau on ne fait pas une *recherche*, mais un *calcul* à partir de la clé, beaucoup plus rapide.

Si C est l'ensemble des clés possibles, et si $T_0 \ldots T_{N-1}$ est le tableau contenant les clés effectives, on se donne donc une fonction, à évaluation peu onéreuse :

$$h : C \rightarrow \{ 0, 1, \ldots N\text{-}1 \}$$

et on espère (c'est une utopie) que:

pour toute clé effective c, l'emplacement de c est $T_{h(c)}$

Bien entendu, l'idéal serait d'avoir une fonction h bijective (i.e. h atteint tous les indices du tableau, et à deux clés différentes h associe des indices différents). Cela est impossible, parce que l'ensemble des clés effectives est un sous-ensemble peu connu, voire tout à fait inconnu, de l'ensemble des clés possibles, qui est souvent gigantesque. On se contente donc de fonctions qu'on pourrait appeler « bijectives avec des ratés », et on met en place un mécanisme pour prendre en charge les « ratés », qu'on appelle des *collisions*. Il y a collision lorsque la fonction h donne la même valeur pour deux clés distinctes c_1 et c_2:

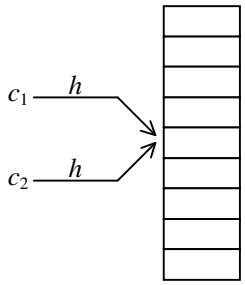


Figure 1. Une collision

Le premier travail, dans la mise en place d'un dispositif d'adressage dispersé, est donc le choix d'une fonction d'adressage $h : C \rightarrow \{ 0, 1, \ldots N\text{-}1 \}$. A cette fonction on demande essentiellement de posséder deux qualités :

- être facile à calculer, car nous voulons une méthode rapide ;
- être uniforme : la probabilité pour que $h(c)$ soit égale à v doit être la même pour toutes les valeurs $0 \leq v \leq N\text{-}1$, c'est-à-dire il ne faut pas qu'à certains endroits de la table se produisent de nombreuses collisions, tandis qu'à d'autres endroits des cases restent libres.

Trouver une bonne fonction d'adressage est un problème difficile, qu'on résout souvent empiriquement : on choisit une fonction, et on « espionne » le fonctionnement global du système durant quelque temps. On améliore la fonction, on espionne de nouveau, et ainsi de suite. En outre, on fait confiance aux recettes transmises par les anciens, comme :

- c'est mieux si N, la taille de la table, est un nombre premier ; en aucun cas N ne doit être une puissance de 2 ;
- les fonctions de la forme $h(c) \equiv h_0(c) \bmod N$ (c'est le cas de la plupart des fonctions) sont d'autant meilleures que h_0 fournit des valeurs très grandes.

Exemples de fonctions d'adressage : en supposant qu'une clé est une chaîne de caractères, notée $c_0 c_1 \ldots c_{n-1}$:

$$h_1(c) = (c_0 - \text{'A'}) \bmod N$$

Attention, h_1 est une très mauvaise fonction, car elle n'est pas uniforme. En voici une meilleure : elle consiste à prendre un mot comme l'écriture d'un nombre en base 26 (puisqu'il y a 26 lettres)

$$h_2(c) = (c_0 + c_1 \times 26 + c_2 \times 26^2 + \ldots + c_k \times 26^k) \bmod N$$

Remarque. Les fonctions du style de h_2 sont utilisées très fréquemment. Elles justifient le conseil « la taille de la table doit être un nombre premier ». Imaginez que N soit égal à 26 : h_2 devient analogue à h_1 ! Si N n'est pas 26 mais possède des diviseurs communs avec 26, certains termes du polynôme ci-dessus seront systématiquement nuls, et la répartition des valeurs fournies par la fonction ne sera pas être uniforme. Prendre N premier est un moyen d'écarter ces défauts.

CHAINAGE EXTERNE

Intéressons-nous maintenant à la résolution des collisions. Il existe principalement deux manières de s'y prendre : le *chaînage externe* et le *chaînage interne* (ou *chaînage ouvert*).

Dans le chaînage externe la fonction h d'adressage ne sert pas à adresser une table de clés, mais une table de listes chaînées, chacune constituée des clés pour lesquelles h fournit la même valeur.

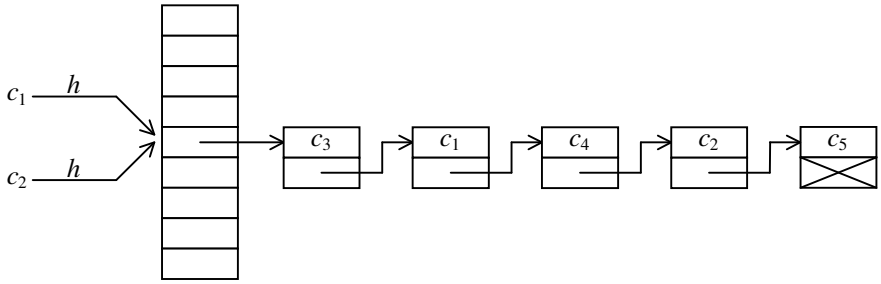


Figure 2. Chaînage externe

La recherche ou l'insertion d'un élément se font en deux temps :

- utilisation de la fonction d'adressage pour obtenir la tête de la liste susceptible de contenir la clé recherchée ;
- parcours séquentiel de cette liste en recherchant la clé en question.

Les programmes correspondants sont laissés au lecteur à titre d'exercice.

Cette méthode convient particulièrement aux ensembles très dynamiques, avec de nombreuses insertions et suppressions, notamment lorsqu'il est difficile ou impossible d'estimer à l'avance le nombre de clés qu'il faudra mémoriser. Dans cette optique, le chaînage externe, que nous avons présenté en parlant de *collisions* (le mot collision suggère l'idée d'un événement exceptionnel, un « raté » de la fonction d'adressage) apparaît plutôt comme une méthode de *partitionnement* des clés : étant donnée une clé, un moyen rapide, la fonction h, détermine le compartiment susceptible de la contenir, dans lequel on fait une recherche séquentielle. Ici, la collision de deux clés signifie simplement qu'elles habitent dans le même compartiment et n'a rien d'exceptionnel.

CHAINAGE INTERNE

Le chaînage interne, au contraire, suppose qu'on a d'avance une bonne connaissance du nombre des clés qui seront à gérer, et qu'on a décidé de les ranger dans un tableau possédant une taille suffisante pour les loger toutes. On ne veut donc pas payer le sur-encombrement dû aux pointeurs des listes chaînées. Ici, l'adressage dispersé est vu comme un substitut de l'indexation, et les collisions comme des accidents regrettables qu'une meilleure fonction d'adressage et une meilleure connaissance de l'ensemble des clés auraient pu éviter.

Le principe du chaînage interne est le suivant :

- les cases du tableau adressé par la fonction d'adressage sont occupées par les clés elles-mêmes ; il existe une valeur de clé conventionnelle pour indiquer qu'une case est libre (nous la noterons *LIBRE*) ;
- recherche : lorsque la fonction h, appliquée à une clé c_1, donne l'indice $h(c_1)$ d'une case occupée par une autre clé, on parcourt le tableau à partir de $h(c_1)$. Cette recherche s'arrête dès la rencontre
  - d'une case contenant la clé c_1, qui signifie le succès de la recherche,
  - d'une case libre, qui signifie l'échec de la recherche ;
- insertion : comme la recherche, en s'arrêtant dès la rencontre
  - d'une case contenant c_1 (l'insertion est refusée, puisqu'on ne veut pas de doublon),
  - d'une case libre: la clé c_1 est insérée à cet endroit

Bien entendu, le tableau est parcouru circulairement : la première case suit logiquement la dernière. De plus, on fait en sorte qu'il y ait constamment au moins une case libre dans le tableau.

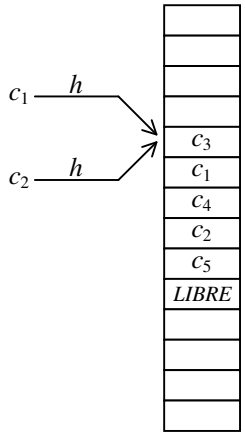


Figure 3. Chaînage interne

Recherche :

```
typedef char *CLE;

#define N 101

CLE t[N];

#define LIBRE       NULL
#define SUIVANT(i)  (((i) + 1) % N)

static int hash(CLE s)
    {
    int r = 0;
    while (*s != 0)
        r = (26 * r + (*s++ - 'A')) % N;
    return r;
    }

int recherche(CLE cle)
    {
    int h;
    h = hash(cle);
    while (t[h] != LIBRE && COMPAR(t[h], cle) != 0)
        h = SUIVANT(h);
    return t[h] != LIBRE ? h : -1;
    }
```

L'insertion n'est pas très différente de la recherche :

```
void insertion(CLE cle)
    {
    int h;
    if (nbre_elements >= N - 1)
        erreurFatale(TABLE_PLEINE);
    h = hash(cle);
    while (t[h] != LIBRE && COMPAR(t[h], cle) != 0)
        h = SUIVANT(h);
    if (t[h] != LIBRE)
        erreurFatale(CLE_DEJA_INSEREE);
    t[h] = COPIE(cle);
    nbre_elements++;
    }
```

La suppression requiert quelques précautions. Pour nous en convaincre, imaginons que la clé d'indice h (voyez la figure 4) doive être supprimée, et considérons la clé d'indice p qui se trouve après h alors que son hash-code h_p correspond à une case qui se trouve devant h. Si, pour supprimer la clé d'indice h, nous nous contentions de la remplacer par la valeur *LIBRE*, nous mettrions la table dans un état incohérent dans lequel la clé d'indice p serait devenue inaccessible : sa recherche commencerait à la case d'indice h_p et serait arrêtée par la case libre d'indice h.

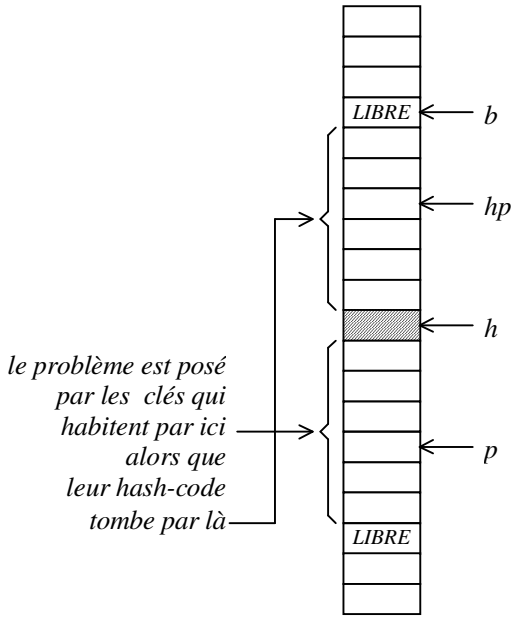


Figure 4. Chaînage interne: suppression d'une clé

Pour éviter cette incohérence, le programme de suppression passe en revue toutes les clés qui sont ainsi susceptibles de devenir inaccessibles. Ce sont les clés (cf. figure 3) :

- qui se trouvent entre la case d'indice *h* et la première case vide après celle-là
- dont le hash-code renvoie à une case qui se trouve entre la dernière case vide devant la case *h* et la case *h*.

Nous utilisons la pseudo-fonction *COMPRIS_ENTRE(a, b, c)* pour caractériser le fait que l'indice *a* est compris entre les indices *b* et *c* en tenant compte du fait que le tableau est circulaire :

```
#define SUIVANT(i)    (((i) + 1) % N)

#define PRECEDENT(i) (((i) + (N - 1)) % N)

#define COMPRIS_ENTRE(a, b, c)                  \
            ((b < c && (b <= a && a <= c)) ||   \
             (c < b && (a <= c || b <= a)))
```

Nous avons le programme :

```
void suppression(CLE cle)
    {
    int h, b, p, hp;
    h = hash(cle);
    while (t[h] != LIBRE && COMPAR(t[h], cle) != 0)
        h = SUIVANT(h);
    if (t[h] == LIBRE)
        erreurFatale(CLE_ABSENTE);

    b = h;
    while (t[b] != LIBRE)
        b = PRECEDENT(b);
```

```
for (;;)
    {
    t[h] = LIBRE;
    for (p = SUIVANT(h);; p = SUIVANT(p))
        {
        if (t[p] == LIBRE)
            return;
        hp = hash(t[p]);
        if (COMPRIS_ENTRE(hp, b, h))
            {
            t[h] = t[p];
            h = p;
            break;
            }
        }
    }
}
```

REHACHAGE.

Un risque majeur, dans les dispositifs de chaînage interne, est le danger d'entrelacement des listes de collisions. Avec le programme précédent si, par un défaut de la fonction d'adressage, les clés ont tendance à se regrouper en une région du tableau, notre manière de résoudre les collisions augmentera encore cet effet d'entassement. Le système deviendra alors inefficace, car les accès à la table se traduiront par des recherches de plus en plus séquentielles.

Une manière de résoudre ce problème consiste à améliorer la méthode de « sondage », c'est-à-dire la manière de chercher la case voulue à partir de la case indiquée par la fonction h.

Ce que nous avons programmé plus haut est le *sondage séquentiel*. Pour en montrer l'inconvénient, imaginons que deux clés c_1 et c_2 aient des hash-codes consécutifs : $h_1 = h(c_1)$, $h_2 = h(c_2)$ et $h_2 = h_1+1$. Les cases qui seront sondées sont

- dans la recherche de c_1 : h_1, h_1+1, h_1+2, h_1+3, …
- dans la recherche de c_2 : $h_2 = h_1+1$, h_1+2, h_1+3, …

Les listes de collision correspondant à h_1 et à h_2 se sont entremêlées : des clés qui n'auraient dû intervenir que dans la recherche de c_1 ou que dans celle de c_2 sont examinées lors de la recherche de c_1 et lors de celle de c_2.

Une meilleure méthode est le *sondage quadratique*, consistant à examiner, pour la recherche ou l'insertion d'une clé c, les cases d'indices

$$h_0 = h(c),\ h_1 = (h_0 + 1^2) \bmod N,\ h_2 = (h_0 + 2^2) \bmod N,\ \ldots\ h_i = (h_0 + i^2) \bmod N$$

Une méthode encore meilleure est l'adressage double, ou *réhachage*, consistant à examiner

$$h_0 = h(c),\ h_1 = (h_0 + u_c) \bmod N,\ h_2 = (h_0 + 2 \times u_c) \bmod N,\ \ldots\ h_i = (h_0 + i \times u_c) \bmod N$$

où u_c est une constante donnée par une deuxième fonction d'adressage :

$$u_c = h'(c)$$

La fonction de réhachage h' doit respecter quelques principes simples :

- elle doit éviter la valeur 0,
- N (la taille du tableau) et u_c doivent être premiers entre eux (sinon les h_i ne décrivent pas tout le tableau),
- plus généralement, h' ne doit pas être « parente »[10] avec h, pour éviter des consanguinités susceptibles de créer des accumulations assez vicieuses.

[10] Ce saisissant raccourci remplace de longues et difficiles explications mathématiques.

4.2. Recherche de motifs dans des chaînes

Le problème que nous étudions ici n'est pas de même nature que le précédent. Il s'agit toujours de rechercher rapidement une clé, mais maintenant nous ne la cherchons plus dans un ensemble de clés habilement structuré en vue d'optimiser les recherches, mais au contraire dans une collection d'informations élémentaires « en vrac ». Cas typique : la recherche d'un mot dans un texte.

On se donne un ensemble E, deux suites d'éléments de E, notées $M = (m_0\ m_1\ \ldots\ m_{lm-1})$, le *motif*, et $C = (c_0\ c_1\ \ldots\ c_{lc-1})$, la *chaîne*, vérifiant $lc \geq lm$, et on recherche la première occurrence de M dans C, c'est-à-dire le plus petit indice i vérifiant

$$i + lm \leq lc$$

et

$$m_j = c_{i+j} \text{ pour } j = 0, 1, \ldots lm\text{-}1.$$

Ce problème se présente souvent, et sous des formes qui font aisément comprendre l'importance d'avoir un algorithme efficace. Des exemples :

- Recherche d'information : E est l'ensemble des caractères, et on cherche un mot ou une expression dans le texte d'une encyclopédie, dans tous les fichiers d'un ordinateur, dans toutes les pages du web, etc. ;
- Recherches en génétique : E = { 'A', 'T', 'G', 'C' } et on recherche une configuration donnée dans une chaîne d'ADN décrite par une suite de plusieurs millions d'occurrences de ces lettres ;
- Traitement de formes (images, sons…) : E = { 0, 1 } et on recherche une configuration binaire donnée dans une suite de plusieurs milliards de 0 et 1.

Pour fixer les idées, nous supposerons ici que E est l'ensemble des caractères du langage de programmation utilisé.

4.2.1. Algorithme naïf

Un algorithme de recherche découle immédiatement de la définition donnée :

```
int recherche(char m[], int lm, char c[], int lc)
    {
    int i, j, imax = lc - lm;

    for (i = 0; i <= imax; i++)
        {
        for (j = 0; j < lm; j++)
            if (m[j] != c[i + j])
                break;
        if (j >= lm)
            return i;           /* succès */
        }
    return -1;                  /* échec */
    }
```

La complexité en moyenne de cet algorithme est $\Theta(lm \times lc)$.

Plusieurs méthodes permettent d'avoir $\Theta\ (lm + lc)$. Elles obtiennent un tel gain en effectuant un travail préalable, une étude du motif, avant de commencer le parcours de la chaîne.

4.2.2. Automate d'états finis

La méthode consiste à construire un automate d'états finis qui reconnaît le motif donné. Le nombre d'opérations est $k \times lm + lc$. Si lc est important devant lm, cette méthode est certainement la meilleure.

Elle sera étudiée en TD.

4.2.3. Algorithme de Knuth, Morris et Pratt

On peut présenter cette méthode en critiquant la méthode naïve. Cette dernière consiste à « aligner » les caractères du motif avec ceux de la chaîne à partir d'une position *i*, puis à avancer de concert, sur le motif et sur la chaîne, tant que les caractères correspondants coïncident :

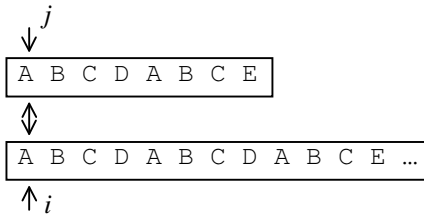


Figure 5. Le début de la recherche

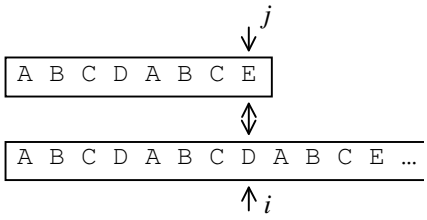


Figure 6. Echec d'un appariement

Si la méthode naïve est peu efficace, c'est que lorsque les caractères correspondants ne coïncident pas, on se contente de recommencer l'appariement à partir du début du motif, après avoir décalé d'un cran ce dernier (figure 7) :

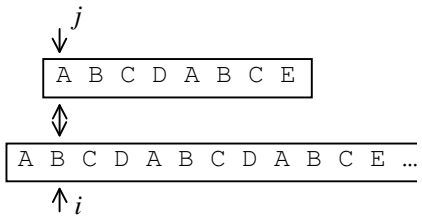


Figure 7. Etape suivante dans la méthode naïve

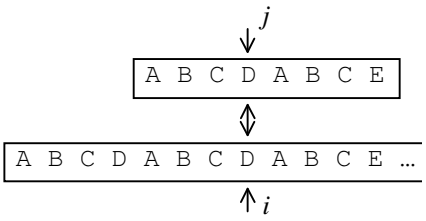


Figure 8. Etape suivante dans la méthode KMP

La méthode de Knuth, Morris et Pratt part de la remarque suivante : si la discordance se produit sur le caractère c_{i+p} c'est que les caractères (c_i c_{i+1} ... c_{i+p-1}) de la chaîne coïncident avec les caractères (m_0 m_1 ... m_{p-1}) du motif. Il devrait donc être possible, par une étude *préalable* du motif, de « précalculer » une fois pour toutes les conduites efficaces à adopter lorsque un appariement échoue.

En fait, dans l'algorithme de KMP on arrive à ne jamais reculer l'indice *i* sur la chaîne, comme le montre la figure 8, en calculant au préalable dans une table *suivant*, pour chaque caractère du motif, la valeur à donner à l'indice *j* lorsqu'une discordance se produit sur ce caractère.

Pour le motif « ABCDABCE », le tableau *suivant* est :

A	B	C	D	A	B	C	E
−1	0	0	0	0	1	2	3

Dans l'exemple de la figure 6, la discordance s'est produite pour *j* = 7, et *suivant*[7] = 3 ; d'où l'état de la figure 8.

La fonction de recherche peut s'écrire ainsi :

```
int recherche_KMP(char m[], int lm, char c[], int lc, int suivant[])
    {
    int i, j;

    for (i = j = 0; j < lm && i < lc; i++, j++)
        while (j >= 0 && c[i] != m[j])
            j = suivant[j];

    if (j == lm)
        return i - lm;              /* succès */
    else
        return -1;                  /* échec */
    }
```

La fonction qui calcule le tableau *suivant* à partir du motif ressemble à une recherche du motif dans lui-même :

```
void initSuivant(char *m, int lm, int suivant[])
    {
    int i, j;

    suivant[0] = -1;
    j = -1;
    i = 0;
    while (i < lm - 1)
        {
        while (j >= 0 && m[i] != m[j])
            j = suivant[j];
        suivant[++i] = ++j;
        }
    }
```

Références

Sur les structures de données et les algorithmes

Robert SEDGEWICK
Algorithmes en langage C
InterEditions, 1991

Christine FROIDEVAUX, Marie-Claude GAUDEL et Michèle SORIA
Types de données et algorithmes
Ediscience International, 1993

Thomas CORMEN, Charles LEISERSON et Ronald RIVEST
Introduction à l'algorithmique
Dunod, 1994

Alfred AHO, John HOPCROFT et Jeffrey ULLMAN
Structures de données et algorithmes
InterEditions, 1987

Sur la programmation en langage C

Brian KERNIGHAN, Dennis RITCHIE
Le langage C (2ème édition)
Masson, 1990

Henri GARRETA
C : Langage, bibliothèques, applications
InterEditions, 1992

Jean-Pierre BRAQUELAIRE
Méthodologie de la programmation en C, 3ème édition
Masson, 1998