Dissection de glib : les arbres binaires balancés

Nous avions vu, deux mois auparavant, une façon de stocker des ensembles de clés et valeurs sous la forme de tables de hachage. Nous revenons à la charge avec les arbres binaires balancés.

Introduction

Un arbre binaire balancé ? Un arbre qui se balance du 0 au 1 ? Non, rien à voir. Un arbre, en informatique, est une structure de données arborescente (ça tombe bien !), c'est-à-dire que chacune de ses cellules est reliée à d'autres par des liens que l'on appelle branches et qui sont orientés. Ce qui diffère par rapport à un graphe orienté est que chaque nœud a un nœud père et un seul, avec une exception : le nœud dit " racine " qui n'a pas de père. Il peut par contre avoir autant de fils qu'il veut. Un arbre binaire impose une contrainte supplémentaire : pour chaque nœud, le nombre de fils est compris entre 0 et 2. Enfin, un arbre binaire balancé est un arbre binaire dans lequel un nœud ne peut avoir de fils tant que tous les nœuds du même niveau n'ont pas exactement deux fils.
De ces contraintes apparaissent des règles mathématiques intéressantes que les généalogistes connaissent bien lorsqu'ils dressent un arbre de leurs ancêtres. Ainsi, dans un niveau, le nombre de nœuds est 2niveau (sauf pour le dernier s'il n'est pas rempli). Ou alors, pour tout nœud, le sous-arbre dont il est la racine est également binaire et balancé (cela nous servira plus tard).
Si l'arbre respecte une relation d'ordre (ce qui est le cas des arbres binaires balancés de glib), une recherche se fait par une dichotomie implicite : pour chaque nœud, on sait si le nœud recherché se trouve dans les fils à droite ou à gauche. Cela nous amène à une complexité de l'algorithme de recherche en n.log(n).
Avec les arbres binaires balancés, la difficulté ne se trouve pas dans la recherche, mais dans la gestion de l'arbre. En effet, toute insertion ou retrait de donnée implique a priori une révision complète de l'arbre afin qu'il continue à respecter les contraintes de binarité et de balancement, sans oublier la notion d'ordre dans le cadre d'un ajout. Nous allons voir la façon relativement élégante dont glib implémente cela.

Remarque :
Cet article est basé sur la version 2.12.4 de glib.

Les structures de données

Un arbre binaire balancé est défini par deux structures dans glib : **GTree** et **GTreeNode**. La première permet de travailler avec l'arbre en entier et la seconde avec des nœuds seulement. Voici leur définition (issue de **glib/gtree.c** et **glib/gtree.h**) :

Gtree

```
01 typedef struct _GTree  GTree;
01 struct _GTree
02 {
03   GTreeNode        *root;
04   GCompareDataFunc  key_compare;
05   GDestroyNotify    key_destroy_func;
06   GDestroyNotify    value_destroy_func;
07   gpointer          key_compare_data;
08   guint             nnodes;
```

```
09 };
```

Vous remarquez dans cette définition un premier lien (ligne 3) vers le nœud racine de l'arbre. C'est également ici que doivent être indiquées la fonction de comparaison nécessaire à la relation d'ordre du **GTree** (ligne 4) et de façon facultative une donnée à passer à cette fonction (ligne 7). Deux fonctions utiles lors du retrait d'un nœud servent à détruire la clé et la valeur du nœud et peuvent être indiquées lignes 5 et 6. Enfin, le nombre de nœuds est en permanence tenu à jour grâce à l'entrée ligne 8.

GTreeNode

```
01 typedef struct _GTreeNode  GTreeNode;
```

```
01 struct _GTreeNode
02 {
03   gpointer   key;         /* key for this node */
04   gpointer   value;       /* value stored at this node */
05   GTreeNode *left;        /* left subtree */
06   GTreeNode *right;       /* right subtree */
07   gint8      balance;     /* height (left) - height (right) */
08   guint8     left_child;
09   guint8     right_child;
10 };
```

Un nœud est caractérisé par sa clé et sa valeur (lignes 3 et 4). Comme dans tout graphe, des liens vers les voisins sont nécessaires. Comme nous avons un arbre binaire balancé, les voisins sont le père et les deux fils. Ces deux derniers sont indiqués lignes 5 et 6. Quant au père, un pointeur est inutile, car le seul moyen logique et utile d'atteindre un nœud est par son père. Il est donc implicitement connu. Les champs **left_child** et **right_child** indiquent si le nœud a un fils à gauche et un à droite. Enfin, la balance, comme indiquée en commentaire ligne 7, indique pour le sous-arbre dont ce nœud est la racine, la différence entre la hauteur du sous-arbre des descendants à gauche et celle de ceux de droite. Par définition d'un arbre binaire balancé, cette différence ne doit prendre que les valeurs -1, 0 ou 1. C'est ici qu'apparaît donc la difficulté d'ajouter ou supprimer un nœud, opérations qui modifient la balance de l'arbre.

Attention :
Le commentaire ligne 7 est faux : droite et gauche ont été inversées. Nous allons voir plus loin comment s'en rendre compte.

Création d'un arbre

Utilisez la fonction **g_tree_new()** dont voici le code :

```
01 GTree*
02 g_tree_new (GCompareFunc key_compare_func)
03 {
04   g_return_val_if_fail (key_compare_func != NULL, NULL);
05
06   return g_tree_new_full ((GCompareDataFunc) key_compare_func, NULL,
07                           NULL, NULL);
08 }
```

Cette fonction n'est autre qu'une interface simplifiée à g_tree_new_full() dont nous allons voir le code. Auparavant, touchons un mot sur l'argument de notre fonction, qui est un pointeur sur une fonction de prototype suivant (extrait de **glib/gtypes.h**) :

```
typedef gint          (*GCompareFunc)         (gconstpointer  a,
                                               gconstpointer  b);
```

La fonction dont vous fournissez le pointeur doit donc accepter deux pointeurs et retourner un entier, négatif si a est inférieur à b, nul s'ils sont égaux, et positif sinon. Un exemple d'une telle fonction est **strcmp()** pour les chaînes de caractères.
Revenons à **g_tree_new_full()** dont voici le code :

```
01 GTree*
02 g_tree_new_full (GCompareDataFunc key_compare_func,
03                   gpointer         key_compare_data,
04                  GDestroyNotify   key_destroy_func,
05                   GDestroyNotify   value_destroy_func)
06 {
07   GTree *tree;
08
09   g_return_val_if_fail (key_compare_func != NULL, NULL);
10
11   tree = g_new (GTree, 1);
12   tree->root               = NULL;
13   tree->key_compare        = key_compare_func;
14   tree->key_destroy_func   = key_destroy_func;
15   tree->value_destroy_func = value_destroy_func;
16   tree->key_compare_data   = key_compare_data;
17   tree->nnodes             = 0;
18
19   return tree;
20 }
```

Cette fonction ne fait finalement rien d'autre que d'allouer un espace disque pour y loger une structure **GTree** (ligne 11) et initialiser celle-ci (lignes 12 à 17). Vous noterez néanmoins qu'un arbre vide a ici une racine absente (ligne 12) et par conséquent un nombre de nœuds nul (ligne 17). Quant à la fonction de comparaison que nous avons discutée plus haut, elle est intrinsèque à l'arbre, puisqu'elle y est indiquée (ligne 16).

Modifier l'arbre

Ajouter un nœud

Pour ajouter un nœud, nous utilisons **g_tree_insert()** dont voici le code :

```
01 void
02 g_tree_insert (GTree    *tree,
03                gpointer  key,
04                gpointer  value)
05 {
06   g_return_if_fail (tree != NULL);
07
```

```
08   g_tree_insert_internal (tree, key, value, FALSE);
09
10 #ifdef G_TREE_DEBUG
11   g_tree_node_check (tree->root);
12 #endif
13 }
```

Nous avons comme d'habitude affaire à une interface à la véritable fonction qui est **g_tree_insert_internal**(). Son code est relativement long, mais vous allez voir pourquoi :

```
01 static void
02 g_tree_insert_internal (GTree    *tree,
03                          gpointer  key,
04                          gpointer  value,
05                          gboolean  replace)
06 {
07   GTreeNode *node;
08   GTreeNode *path[MAX_GTREE_HEIGHT];
09   int idx;
```

Dans les déclarations ci-dessus, notez le tableau **path** qui va stocker le chemin permettant d'atteindre le nœud à insérer. Remarquez également que ce chemin est limité de façon statique à **MAX_GTREE_HEIGHT** dont la valeur est 40 (comme défini au début du fichier **glib/gtree.c**). Une valeur de 40 peut paraître surprenante, mais l'arbre étant binaire et balancé, le nombre de nœuds est ainsi de l'ordre de 239 (39 car la hauteur peut augmenter lors d'un ajout). Avez-vous assez de mémoire pour stocker 239 pointeurs vers des **GtreeNode** ? Donc 40 est tout à fait correct. Le code suivant, lignes 13 à 18, permet d'insérer un nœud dans un arbre vide, ce qui est entre autres le cas avec un arbre qui vient d'être créé.

Si l'arbre contenait déjà des nœuds, nous allons partir à la recherche de l'emplacement adéquat pour notre nouveau nœud. Nous initialisons pour cela notre chemin lignes 20 et 21, ainsi que notre pointeur de nœud courant, **node**, ligne 22.

```
10
11   g_return_if_fail (tree != NULL);
12
13   if (!tree->root)
14     {
15       tree->root = g_tree_node_new (key, value);
16       tree->nnodes++;
17       return;
18     }
19
20   idx = 0;
21   path[idx++] = NULL;
22   node = tree->root;
23
```

Les développeurs ont fait le choix d'une boucle sans fin pour rechercher le bon emplacement pour notre nœud à insérer. Pour en sortir, nous avons deux cas de figure. Soit nous trouvons un nœud dont la clé est égale à celle du nœud à insérer (condition ligne 28, action lignes 29 à 50) et nous effectuons un remplacement avant de quitter ligne 48 avec return, le travail étant achevé. Soit le nœud ne se trouve pas encore dans l'arbre et nous cherchons l'endroit le plus approprié. Cela

consiste à chaque itération à comparer (avec la variable ligne 26) notre clé avec celle du nœud courant. Si elle est inférieure, nous cherchons dans le sous-arbre de gauche. Si elle est supérieure, nous continuons dans celui de droite. Ces tests se trouvent lignes 51 et 73, et le changement de nœud courant lignes 53 à 57 (gauche) ou 75 à 79 (droite). Le changement de nœud courant est à chaque fois précédé d'un test pour savoir s'il existe bien un nœud à droite ou à gauche en fonction de là où nous voulons aller. Lorsque le test de présence du nœud fils est négatif, nous créons un nouveau nœud (lignes 60/82) que nous accrochons à l'arbre dans les lignes qui suivent.
Lorsque le nœud est accroché à l'arbre, nous rencontrons l'instruction break qui nous sort de la boucle infinie et nous amène ligne 100.

```
24   while (1)
25     {
26       int cmp = tree->key_compare (key, node->key, tree->key_compare_data);
27
28       if (cmp == 0)
29         {
30           if (tree->value_destroy_func)
31             tree->value_destroy_func (node->value);
32
33           node->value = value;
34
35           if (replace)
36             {
37               if (tree->key_destroy_func)
38                 tree->key_destroy_func (node->key);
39
40               node->key = key;
41             }
42           else
43             {
44               /* free the passed key */
45               if (tree->key_destroy_func)
46                 tree->key_destroy_func (key);
47             }
48
49           return;
50         }
51       else if (cmp < 0)
52         {
53           if (node->left_child)
54             {
55               path[idx++] = node;
56               node = node->left;
57             }
58           else
59             {
60               GTreeNode *child = g_tree_node_new (key, value);
61
62               child->left = node->left;
63               child->right = node;
64               node->left = child;
65               node->left_child = TRUE;
66               node->balance -= 1;
67
68               tree->nnodes++;
```

L'accrochage d'un nœud à l'arbre peut paraître surprenant. En effet, nous initialisons les fils à d'autres valeurs que **NULL**. En réalité, ces pointeurs vers des fils de gauche et droite permettent

d'accéder au nœud directement inférieur (fils gauche) et directement supérieur (fils droit) de notre nœud. Ce sont **left_child** et **right_child**, initialisés à **NULL** dans **g_tree_node_new()** qui permettent de s'assurer que notre nœud n'a pas de fils.
Par ailleurs, ligne 66, nous décrémentons la balance. Cela signifie bien que lorsque l'arbre est plus chargé à gauche, la balance décrémente. La balance, comme nous l'avons vu plus haut, est donc définie ainsi : **height(right) - height(left)**.
Vous retrouvez le pendant de ce code pour une insertion à droite lignes 82 à 90.

```
69
70                break;
71              }
72          }
73        else
74          {
75            if (node->right_child)
76              {
77                path[idx++] = node;
78                node = node->right;
79              }
80            else
81              {
82                GTreeNode *child = g_tree_node_new (key, value);
83
84                child->right = node->right;
85                child->left = node;
86                node->right = child;
87                node->right_child = TRUE;
88                node->balance += 1;
89
90                tree->nnodes++;
91
92                break;
93              }
94          }
95      }
96
97    /* restore balance. This is the goodness of a non-recursive
98       implementation, when we are done with balancing we 'break'
99       the loop and we are done. */
```

Lorsqu'un nœud est inséré, il faut rétablir la balance. L'algorithme suivant consiste à revoir l'équilibre de chaque sous-arbre dont les nœuds sont sur le chemin d'accès au nœud que nous venons d'insérer, et ce, en partant du nouveau nœud. C'est ce que vous retrouvez ligne 102 avec la variable **bparent** qui indique le nœud parent du nœud courant.
Ainsi, pour chaque nœud, nous allons :

- rétablir la balance si nécessaire (lignes 106 à 115) ;
- tester si l'arbre est à nouveau équilibré (test ligne 117) et sortir de la boucle infinie si tel est le cas ;
- réajuster la balance du nœud courant après l'équilibrage opéré ligne 108 (lignes 120 à 123) ;
- itérer sur le nœud courant, **node**, en le faisant remonter le long du chemin vers la racine.

```
100   while (1)
101     {
102       GTreeNode *bparent = path[--idx];
103       gboolean left_node = (bparent && node == bparent->left);
104       g_assert (!bparent || bparent->left == node || bparent->right ==
```

```
node);
105
106       if (node->balance < -1 || node->balance > 1)
107         {
108           node = g_tree_node_balance (node);
109           if (bparent == NULL)
110             tree->root = node;
111           else if (left_node)
112             bparent->left = node;
113           else
114             bparent->right = node;
115         }
116
117       if (node->balance == 0 || bparent == NULL)
118         break;
119
120       if (left_node)
121         bparent->balance -= 1;
122       else
123         bparent->balance += 1;
124
125       node = bparent;
126     }
127 }
```

Remarque :
Il n'y a pas de rééquilibrage à effectuer lorsque la balance prend une valeur -1, 0 ou 1 (ligne 106). Ensuite, qu'il y ait eu rééquilibrage ou non, soit nous retrouvons un équilibre parfait avec une balance à zéro, ce qui signifie que l'équilibre, déjà bon par ailleurs, est bon partout, soit nous arrivons à la racine à force d'itérer. Dans ces deux cas, il n'y a plus besoin d'itérer : la boucle prend fin (lignes 117 et 118). Ou alors, nous ne sommes dans aucun de ces deux cas, cela reste le chaos dans l'arbre, et il faut continuer à itérer. La balance est revue lignes 120 à 123 et l'itération a lieu ligne 125.

Nous abordons le sujet de la rotation d'arbres après avoir vu comment ôter un nœud d'un arbre.

Enlever un nœud

Il existe deux fonctions pour retirer un nœud de l'arbre : **g_tree_remove()** et **g_tree_steal()**. La seule différence entre ces deux interfaces à **g_tree_remove_internal()** est le troisième argument pour cette fonction, ligne 9.

```
01 gboolean
02 g_tree_remove (GTree         *tree,
03                gconstpointer  key)
04 {
05   gboolean removed;
06
07   g_return_val_if_fail (tree != NULL, FALSE);
08
09   removed = g_tree_remove_internal (tree, key, FALSE);
10
11 #ifdef G_TREE_DEBUG
12   g_tree_node_check (tree->root);
13 #endif
14
15   return removed;
```

```
16 }
```

Voyons le code de cette fonction, étrangement proche de **g_tree_insert_internal(),** mais pas tant que cela :

```
01 static gboolean
02 g_tree_remove_internal (GTree         *tree,
03                         gconstpointer  key,
04                         gboolean       steal)
05 {
06   GTreeNode *node, *parent, *balance;
07   GTreeNode *path[MAX_GTREE_HEIGHT];
08   int idx;
09   gboolean left_node;
```

Le début est le même comme vous pouvez le constater. La suite consiste à rechercher le nœud (appelons " AS " pour la suite ce nœud " À Supprimer "), ce que nous avons déjà vu plus haut. Aussi, nous coupons cette partie du code. La suite consiste à supprimer le nœud AS. Cela est facile quand il s'agit d'une feuille, c'est-à-dire un nœud sans descendance (lignes 53 à 58). Mais dès qu'il y a un fils, cela se complique.

```
44   /* the following code is almost equal to g_tree_remove_node,
45      except that we do not have to call g_tree_node_parent. */
46   balance = parent = path[--idx];
47   g_assert (!parent || parent->left == node || parent->right == node);
48   left_node = (parent && node == parent->left);
49
50   if (!node->left_child)
51     {
52       if (!node->right_child)
53         {
54           if (!parent)
55             tree->root = NULL;
56           else if (left_node)
57             {
58               parent->left_child = FALSE;
59               parent->left = node->left;
60               parent->balance += 1;
61             }
62           else
63             {
64               parent->right_child = FALSE;
65               parent->right = node->right;
66               parent->balance -= 1;
67             }
68         }
```

Lorsque le nœud AS a un fils et un seul, ici le droit (le gauche pour les lignes 91 à 107 dont le code est symétrique à celui-ci et que nous ne présentons pas pour cette raison), il suffit que le fils s'attache à son grand-père du côté où était attaché son père le nœud AS, libérant ainsi celui-ci dans l'opération.

```
69       else /* node has a right child */
70         {
71           GTreeNode *tmp = g_tree_node_next (node);
```

```
72            tmp->left = node->left;
73
74            if (!parent)
75              tree->root = node->right;
76            else if (left_node)
77              {
78                parent->left = node->right;
79                parent->balance += 1;
80              }
81            else
82              {
83                parent->right = node->right;
84                parent->balance -= 1;
85              }
86          }
87      }
88    else /* node has a left child */
89      {
90        if (!node->right_child)
[...]
108        else /* node has a both children (pant, pant!) */
109          {
```

Nous voici dans le cas le plus difficile à traiter. En effet, en supprimant notre nœud AS, nous avons deux sous-arbres à rattacher à notre arbre et seulement un emplacement de libre sur le nœud père de notre nœud.
Le principe consiste à rechercher le nœud qui suit AS (lignes 118 à 122). Il s'agit du nœud le plus à gauche du sous-arbre de droite du nœud AS. Ce nœud a la propriété de ne pas avoir de fils à gauche. En effet, s'il en avait un, ce serait AS puisqu'il le suit directement. Par effet de bord, ce nœud a également la particularité que tous les nœuds qui le précèdent précèdent également AS et ceux qui le suivent suivent aussi AS. Si nous supprimons AS, ce nœud peut le remplacer. Nous allons voir comment...

```
110            GTreeNode *prev = node->left;
111            GTreeNode *next = node->right;
112            GTreeNode *nextp = node;
113            int old_idx = idx + 1;
114            idx++;
115
116            /* path[idx] == parent */
117            /* find the immediately next node (and its parent) */
118            while (next->left_child)
119              {
120                path[++idx] = nextp = next;
121                next = next->left;
122              }
123
124            path[old_idx] = next;
125            balance = path[idx];
126
```

Nous avons ici **next** qui pointe sur le nœud suivant directement AS. Son père est **nextp**. Et nous avons déjà modifié le chemin d'accès à AS ligne 124. Quant à **balance**, il pointe sur le père de **next**. Cette manière de faire ligne 125 peut paraître surprenante au premier abord, mais elle évite l'écueil de la boucle jamais parcourue ligne 118.

L'étape suivante consiste à détacher le nœud **next** de l'endroit où il se trouvait. Ceci est plus facile, car il n'a pas de fils gauche. Il suffit donc de donner son fils droit à son père au même emplacement où il était. Et comme nous avons trouvé **next** en itérant sur les fils gauches, il s'agit bien du fils gauche du père qui va maintenant pointer sur le fils droit de **next** (lignes 131 ou 133). Le fils droit de **next** peut maintenant pointer sur le fils droit du nœud AS (lignes 136 et 137).
L'étape qui suit, lignes 143 à 145 consiste simplement à ce que le nœud précédent directement le nœud AS pointe non plus vers AS au niveau du fils droit, mais vers celui qui va le remplacer, **next**.

```
127           /* remove 'next' from the tree */
128           if (nextp != node)
129             {
130               if (next->right_child)
131                 nextp->left = next->right;
132               else
133                 nextp->left_child = FALSE;
134               nextp->balance += 1;
135
136               next->right_child = TRUE;
137               next->right = node->right;
138             }
139           else
140             node->balance -= 1;
141
142           /* set the prev to point to the right place */
143           while (prev->right_child)
144             prev = prev->right;
145           prev->right = next;
146
```

Lignes 148 à 150, nous nous occupons du fils gauche de **next** qui va pointer vers le fils gauche du nœud AS **node**. Ainsi, **next** a presque fini de remplacer **node**. Il ne reste plus qu'à lui donner pour père celui du nœud AS (lignes 152 à 157) et l'arbre sera à nouveau un arbre.

```
147           /* prepare 'next' to replace 'node' */
148           next->left_child = TRUE;
149           next->left = node->left;
150           next->balance = node->balance;
151
152           if (!parent)
153             tree->root = next;
154           else if (left_node)
155             parent->left = next;
156           else
157             parent->right = next;
158         }
159     }
160
161   /* restore balance */
```

Le code suivant sert à rétablir l'équilibre de l'arbre qui n'est plus balancé. Nous appliquons exactement le même algorithme que pour l'insertion.

```
[...]
191   if (!steal)
192     {
193       if (tree->key_destroy_func)
```

```
194         tree->key_destroy_func (node->key);
195       if (tree->value_destroy_func)
196         tree->value_destroy_func (node->value);
197     }
198
199   g_slice_free (GTreeNode, node);
200
201   tree->nnodes--;
202
203   return TRUE;
204 }
```

Nous voilà à la fin, avec la destruction facultative des valeurs contenues dans le nœud AS (lignes 191 à 197), puis à la destruction du nœud lui-même (ligne 199). L'arbre perd un nœud, ce qui est reporté ligne 201.

Rotation d'un arbre

La rotation d'un arbre, surtout appliquée à des sous-arbres dans notre cas, permet de rétablir leur équilibre et garantir que l'arbre est bien balancé. La rotation d'un arbre consiste à changer la racine de celui-ci, en en prenant une à gauche (rotation à gauche) ou à droite (rotation à droite). Voici le code de **g_tree_node_balance()** qui détermine le côté des rotations à effectuer en fonction du déséquilibre de l'arbre :

```
01 static GTreeNode*
02 g_tree_node_balance (GTreeNode *node)
03 {
04   if (node->balance < -1)
05     {
06       if (node->left->balance > 0)
07         node->left = g_tree_node_rotate_left (node->left);
08       node = g_tree_node_rotate_right (node);
09     }
10   else if (node->balance > 1)
11     {
12       if (node->right->balance < 0)
13         node->right = g_tree_node_rotate_right (node->right);
14       node = g_tree_node_rotate_left (node);
15     }
16
17   return node;
18 }
```

Si l'arbre penche trop à gauche (ligne 4), nous regardons si par hasard le sous-arbre de gauche a le moindre déséquilibre à droite (ligne 6, remarquez la comparaison stricte). Si tel était le cas, nous effectuerions une rotation à gauche du sous-arbre de gauche. Puis, dans tous les cas, comme notre arbre penche trop à gauche, nous nous lançons dans une rotation à droite (ligne 8). Si l'arbre penche trop à droite (ligne 10), il s'agit du même algorithme, mais avec gauche et droite inversées (lignes 12 à 14).
Comme des illustrations ne font jamais de mal, voyons ce qui se passe sur l'arbre (figure 1) sur lequel nous venons d'ajouter le nœud [D]. Les nœuds sont symbolisés par des cadres. Nous avons également fait figurer quelques pointeurs vers des nœuds qui ne sont pas encadrés. Ce sont ce sur quoi pointent les fils gauches et droits des nœuds sans enfants (lorsque le champ **left_child** ou

right_child est **FALSE**).

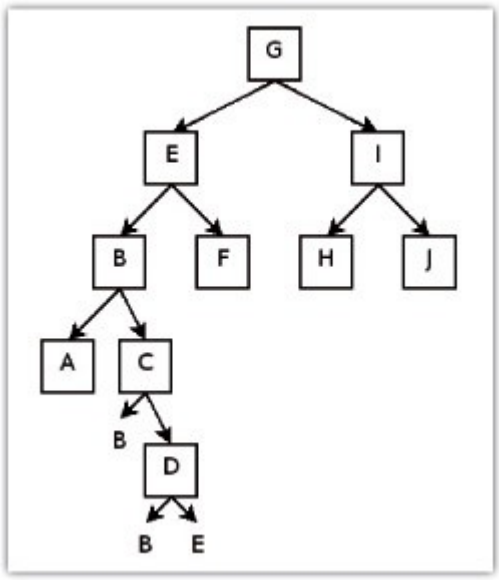

Fig. 1

Nous travaillons sur le nœud E et appliquons **g_tree_node_balance()** sur celui-ci. La première étape consiste à effectuer une rotation à gauche, car pour le nœud [C], le sous-arbre est trop lourd. Voyez la figure 2 dans laquelle le nœud [C], initialement sans fils gauche, prend aisément le nœud [B] en tant que fils gauche. La figure 3 correspond à la figure 2, mais est plus présentable.

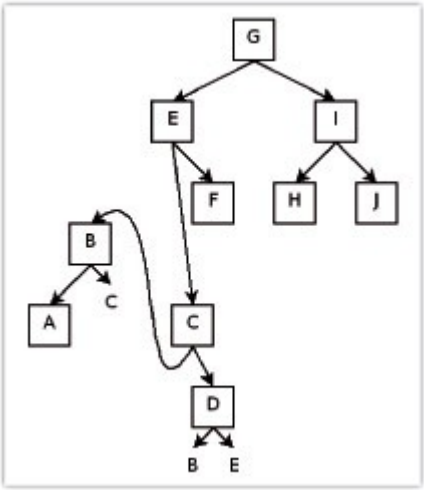

Fig. 2

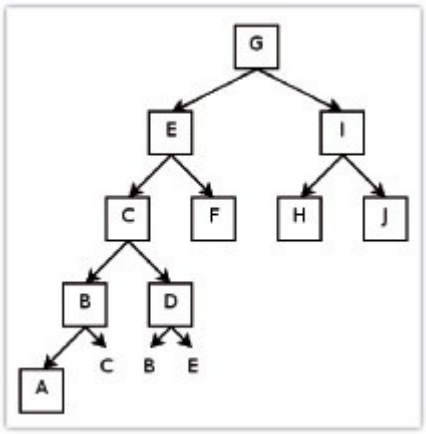

Fig. 3

Ensuite, nous pouvons effectuer la rotation à droite sur le nœud [E]. La figure 4 présente cette rotation alors que le nœud [G] ne sait pas encore que son nouveau fils sera [C] et non [E]. La figure 5 remet tout cela en forme avec pour seul changement le nœud [G] pointant sur [C].

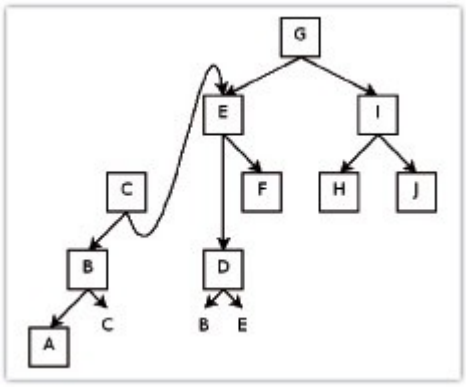

Fig. 4

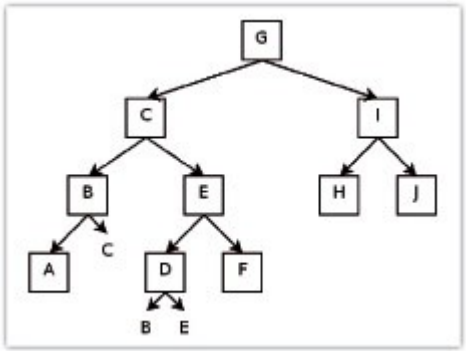

Fig. 5

Nous utilisons la fonction **g_tree_node_rotate_left()** pour la rotation à gauche. Ce code est en deux parties. La première, jusqu'à la ligne 19, effectue la rotation proprement dite. La suite modifie le champ **balance** des nœuds touchés par la rotation. Voici le code :

```
01 static GTreeNode*
02 g_tree_node_rotate_left (GTreeNode *node)
03 {
04   GTreeNode *right;
05   gint a_bal;
06   gint b_bal;
```

```
07
08   right = node->right;
09
10   if (right->left_child)
11     node->right = right->left;
12   else
13     {
14       node->right_child = FALSE;
15       node->right = right;
16       right->left_child = TRUE;
17     }
18   right->left = node;
19
```

Nous nous intéressons au nœud fils de droite **right** de la racine **node**, car c'est lui, **right**, qui va devenir la nouvelle racine du sous-arbre (ligne 40). Vous voyez donc bien ici qu'il s'agit d'une rotation à gauche, car **right**, en devenant racine, se déplace vers la gauche. Une autre façon de voir les choses, plus proche de l'algorithme, est de considérer que nous allons enlever la racine **node>** au profit de son fils droit **right**, pour la réinsérer ailleurs, un ailleurs qui est forcément le fils gauche de **right** pour conserver l'ordre et le balancement.
Ceci pose un problème : qu'allons nous faire des nœuds du côté gauche de **right** ? Comme ils sont à droite de **node** et à gauche de **right**, ils sont tous classés entre ces deux nœuds. De plus, comme **right** va changer de nœud père, l'emplacement droit de **node** sera rendu disponible. Pour respecter l'ordre et le balancement de l'arbre, c'est l'endroit idéal pour placer les nœuds du côté gauche de **right**.
C'est pourquoi, ligne 10, nous regardons si **right** a des fils à gauche. Si oui, ligne 11, nous les déplaçons du côté droit de **node**. Sinon, nous désactivons la branche droite de **node** ligne 14 tout en indiquant que son nœud suivant est **right** (ligne 15). De plus, dans ce cas, il faut activer la branche gauche de right. Enfin, dans tous les cas, nous finissons la rotation à gauche en attachant **node** à **right** en tant que son fils gauche (ligne 18).
La suite consiste à revoir la balance des nœuds dont la descendance a été modifiée. Vous pouvez observer qu'il ne s'agit que de **node** et **right**. Nous vous laissons vérifier en exercice que les cas rencontrés ci-dessous et les tests associés correspondent bien à ce que nous venons de voir.

```
20   a_bal = node->balance;
21   b_bal = right->balance;
22
23   if (b_bal <= 0)
24     {
25       if (a_bal >= 1)
26         right->balance = b_bal - 1;
27       else
28         right->balance = a_bal + b_bal - 2;
29       node->balance = a_bal - 1;
30     }
31   else
32     {
33       if (a_bal <= b_bal)
34         right->balance = a_bal - 2;
35       else
36         right->balance = b_bal - 1;
37       node->balance = a_bal - b_bal - 1;
38     }
39
```

```
40   return right;
41 }
```

Enfin, le nœud **right** correspondant à la nouvelle racine de l'arbre, est renvoyé à la fonction appelante qui va s'empresser de corriger le nœud qui pointait auparavant sur **node** et qui va maintenant pointer sur **right**.
Le code de **g_tree_node_rotate_right()** est le même que celui que nous venons de voir, avec gauche et droite inversées.

Parcourir un arbre

Il existe trois façons logiques de parcourir un arbre. Elles dépendent du moment où vous allez travailler sur chaque nœud : avant de visiter les nœuds fils (parcours préfixe), après avoir visité le nœud gauche et avant le droit (parcours infixe) ou après avoir visité les deux fils (parcours postfixe).
La fonction **g_tree_traverse()** traduit bien ces trois possibilités et se passe de commentaire :

```
01 void
02 g_tree_traverse (GTree         *tree,
03                  GTraverseFunc  traverse_func,
04                  GTraverseType  traverse_type,
05                  gpointer       user_data)
06 {
07   g_return_if_fail (tree != NULL);
08
09   if (!tree->root)
10     return;
11
12   switch (traverse_type)
13     {
14     case G_PRE_ORDER:
15       g_tree_node_pre_order (tree->root, traverse_func, user_data);
16       break;
17
18     case G_IN_ORDER:
19       g_tree_node_in_order (tree->root, traverse_func, user_data);
20       break;
21
22     case G_POST_ORDER:
23       g_tree_node_post_order (tree->root, traverse_func, user_data);
24       break;
25
26     case G_LEVEL_ORDER:
27       g_warning ("g_tree_traverse(): traverse type G_LEVEL_ORDER isn't
implemented.");
28       break;
29     }
30 }
```

Attention :
Cette fonction est dépréciée au profit de **g_tree_foreach()**. Nous l'avons indiquée ici dans le but d'illustrer les différents parcours possibles d'un arbre. Dans une utilisation effective de glib, utilisez **g_tree_foreach()**.

Nous vous présentons également une de ces trois fonctions, par exemple celle du parcours infixe, **g_tree_node_in_order()** :

```
01 static gint
02 g_tree_node_in_order (GTreeNode     *node,
03                       GTraverseFunc  traverse_func,
04                       gpointer       data)
05 {
06   if (node->left_child)
07     {
08       if (g_tree_node_in_order (node->left, traverse_func, data))
09         return TRUE;
10     }
11
12   if ((*traverse_func) (node->key, node->value, data))
13     return TRUE;
14
15   if (node->right_child)
16     {
17       if (g_tree_node_in_order (node->right, traverse_func, data))
18         return TRUE;
19     }
20
21   return FALSE;
22 }
```

Cette fonction est récursive à souhait, avec comme condition d'arrêt l'absence de nœud fils, à gauche ligne 6 comme à droite ligne 15. Il s'agit bien d'un parcours infixe, puisque vous voyez le traitement du nœud ligne 12, après le parcours de la descendance gauche (ligne 8) et avant celui de droite (ligne 17).
Par ailleurs, vous pouvez remarquer que la fonction renvoie toujours FALSE, sauf si la fonction de traitement du nœud (celle de type ***traverse_func**) renvoie **TRUE** (ligne 13). Dans ce cas, le traitement de l'arbre prend fin comme s'il s'agissait du premier domino qui fait tomber tous les autres. En effet, les fonctions **g_tree_node_in_order()** appelantes repèrent alors le retour de la valeur **TRUE** et renvoient elles-mêmes **TRUE** (lignes 9 et 18).
Les deux autres fonctions **g_tree_node_pre_order()** et **g_tree_node_post_order()** ne présentent pas un intérêt supplémentaire, et nous ne vous les montrons pas d'autant plus que **g_tree_traverse()** est dépréciée. La fonction à utiliser pour parcourir l'arbre est **g_tree_foreach()**. Cette fonction travaille comme s'il s'agissait d'une liste chaînée :

```
01 void
02 g_tree_foreach (GTree         *tree,
03                 GTraverseFunc  func,
04                 gpointer       user_data)
05 {
06   GTreeNode *node;
07
08   g_return_if_fail (tree != NULL);
09
10   if (!tree->root)
11     return;
12
13   node = g_tree_first_node (tree);
14
15   while (node)
16     {
17       if ((*func) (node->key, node->value, user_data))
18         break;
19
20       node = g_tree_node_next (node);
```

```
21     }
22 }
```

Ce code, proche de celui du parcours d'une liste chaînée simple, se passe de commentaire. Une question cependant (réponse à la fin) : est-ce un parcours préfixe, infixe ou postfixe ? Nous allons par ailleurs voir comment s'implémentent les deux fonctions **g_tree_first_node()** utilisée ligne 13 et **g_tree_node_next()** ligne 20.

```
01 static inline GTreeNode *
02 g_tree_first_node (GTree *tree)
03 {
04   GTreeNode *tmp;
05
06   if (!tree->root)
07     return NULL;
08
09   tmp = tree->root;
10
11   while (tmp->left_child)
12     tmp = tmp->left;
13
14   return tmp;
15 }
```

Le premier nœud est évidemment celui le plus à gauche, car s'il y en avait un encore plus à gauche, il lui serait inférieur et donc mieux placé pour être le premier. Nous avons donc affaire à une boucle (ligne 11) qui itère sur les fils gauches (ligne 12) de l'arbre.

```
01 static inline GTreeNode *
02 g_tree_node_next (GTreeNode *node)
03 {
04   GTreeNode *tmp;
05
06   tmp = node->right;
07
08   if (node->right_child)
09     while (tmp->left_child)
10       tmp = tmp->left;
11
12   return tmp;
13 }
```

Cette fonction utilise la propriété de l'implémentation de glib que nous avons vue plusieurs fois auparavant : le pointeur de droite d'un nœud, lorsque celui-ci n'a pas de fils droit, pointe vers le nœud suivant. Cette propriété nous facilite énormément le travail, car en l'absence d'un fils droit (test ligne 8), nous avons immédiatement ce que nous cherchons (ligne 6). Par contre, si ce fils droit existe, il est certes après notre nœud, mais il faut chercher le nœud suivant comme le nœud le plus à gauche de ce fils droit. Nous retrouvons la même boucle (lignes 9 et 10) que dans **g_tree_first_node()** pour rechercher le premier nœud du sous-arbre dont le nœud droit est la racine.

Rechercher une clé dans un arbre

A quoi bon construire un arbre si ce n'est pas pour chercher dedans ? Voici la fonction la plus attendue, que nous avons gardée pour la fin étant donné sa simplicité : **g_tree_lookup()** :

```
01 gpointer
02 g_tree_lookup (GTree         *tree,
03                gconstpointer  key)
04 {
05   GTreeNode *node;
06
07   g_return_val_if_fail (tree != NULL, NULL);
08
09   node = g_tree_find_node (tree, key);
10
11   return node ? node->value : NULL;
12 }
```

Comme vous vous en doutiez, il s'agit encore d'une interface. La véritable fonction de recherche est, comme le porte si bien son nom, **g_tree_find_node()** :

```
01 static GTreeNode *
02 g_tree_find_node (GTree        *tree,
03                   gconstpointer key)
04 {
05   GTreeNode *node;
06   gint cmp;
07
08   node = tree->root;
09   if (!node)
10     return NULL;
11
12   while (1)
13     {
14       cmp = tree->key_compare (key, node->key, tree->key_compare_data);
15       if (cmp == 0)
16         return node;
17       else if (cmp < 0)
18         {
19           if (!node->left_child)
20             return NULL;
21
22           node = node->left;
23         }
24       else
25         {
26           if (!node->right_child)
27             return NULL;
28
29           node = node->right;
30         }
31     }

32 }
```

Cette fonction ne vous rappelle rien ? Il ne s'agit rien d'autre que ce que nous avons déjà vu dans le cas de la suppression d'un nœud de l'arbre, éventuellement même de l'ajout d'un nœud : il

s'agissait déjà de rechercher un nœud ! Aussi, nous n'allons pas détailler cette fonction.
Il existe une autre fonction, **g_tree_lookup_extended()**, dont le code, similaire à **g_tree_lookup()**, ne se distingue que par deux arguments supplémentaires susceptibles d'accueillir le pointeur vers la clé et celui vers la valeur du nœud lorsque celui-ci a été trouvé. Mais il est tellement plus facile d'utiliser **node->key** et **node->value** que de disposer de pointeurs pour accueillir ceux-ci que cette fonction a peu de chances d'être plébiscitée. En voici néanmoins son prototype pour vous éclaircir les idées :

```
gboolean g_tree_lookup_extended (GTree           *tree,
                                 gconstpointer    lookup_key,
                                 gpointer        *orig_key,
                                 gpointer        *value);
```

Une autre fonction, **g_tree_search()**, interface à **g_tree_node_search()**, permet d'effectuer une recherche à l'aide d'une fonction de recherche. Celle-ci ressemble comme deux gouttes d'eau à **g_tree_find_node()**. La différence porte sur la ligne 14. Dans **g_tree_node_search()**, voici ce que vous pouvez lire :

```
01 static gpointer
02 g_tree_node_search (GTreeNode     *node,
03                     GCompareFunc   search_func,
04                     gconstpointer  data)
05 {
06   gint dir;
07
08   if (!node)
09     return NULL;
10
11   while (1)
12     {
13       dir = (* search_func) (node->key, data);
14       if (dir == 0)
15         return node->value;
16       else if (dir < 0)
[...]
```

A partir de la ligne 14 déjà le code est le même entre les deux fonctions.

Conclusion

Cet article vous a présenté les arbres binaires balancés et leur implémentation dans glib. Vous avez pu constater qu'une modification dans l'arbre n'est pas triviale, mais qu'une recherche est, par contre, très rapide. Si vous avez encore en tête l'article qui portait sur les tables de hachage, vous pouvez vous interroger sur le meilleur choix à effectuer entre un arbre binaire balancé et une table de hachage. La différence n'est pas significative sur de petits jeux de données avec les machines puissantes dont nous disposons aujourd'hui. Si la table de hachage est suffisamment grande, et les données assez homogènes, le hachage sera probablement la meilleure solution avec un accès quasi direct aux données grâce à la fonction de hachage. Dans les autres cas, en particulier si vous devez chercher des données selon plusieurs critères, l'arbre se révèle plus adapté.
Le mois prochain : nous en avons fini avec cette série d'article sur la dissection de la glib. En effet, il est délicat de traiter d'autres thèmes sans avoir plusieurs milliers de lignes à disséquer, alors que tout ce que vous avez lu portait sur environ 1000 à 2000 lignes de code. Voici quelques fichiers du

répertoire **glib** des sources de glib, ainsi que leur nombre de lignes pour vous faire une idée :

```
fichier          | lignes | octets | commentaire
-----------------|--------|--------|-------------------------------------
glib/garray.c    |   719  | 15962  | Tableaux dynamiques
glib/garray.h    |   167  |  6442  |     (cf Linux Magazine 85)
glib/ghash.c     |   791  | 22688  | Tables de hachage
glib/ghash.h     |   117  |  4396  |     (cf Linux Magazine 88)
glib/glib.h      |    82  |  2501  | En-têtes génériques de Glib
glib/glist.c     |   652  | 11167  | Listes chaînées doubles
glib/glist.h     |   116  |  4961  |     (cf Linux Magazine 86)
glib/gmacros.h   |   261  |  8959  | Macros pour tous les en-têtes de Glib
glib/gmain.c     |  4025  | 98658  | Boucle principale de Glib
glib/gmain.h     |   324  | 10990  | Boucle principale de Glib
glib/goption.c   |  2066  | 50665  | Analyseur de la ligne de commande
glib/goption.h   |   158  |  5679  |     (cf Linux Magazine 89)
glib/gslice.c    |  1132  | 39959  | Gestion de la mémoire
glib/gslice.h    |    77  |  2909  | Gestion de la mémoire
glib/gslist.c    |   621  | 10482  | Listes chaînées simples
glib/gslist.h    |   110  |  4759  |     (cf Linux Magazine 86)
glib/gstrfuncs.c |  2818  | 66679  | Fonctions autour des gchar*
glib/gstrfuncs.h |   248  |  9376  | Fonctions autour des gchar*
glib/gstring.c   |   930  | 19213  | Chaînes de caractères
glib/gstring.h   |   157  |  5572  |     (cf Linux Magazine 87)
glib/gtree.c     |  1290  | 30381  | Arbres binaires balancés
glib/gtree.h     |    88  |  3726  |     (cf ce présent numéro)
glib/gtypes.h    |   422  | 15252  | Définitions de types pour Glib
```

Comme vous pouvez le constater, aborder d'autres sujets comme les fonctions autour de **gchar*** ou la boucle principale pose un problème de concision. Aussi, nous vous invitons à regarder vous-même dans le code si celui-ci vous intéresse.
La réponse à la question : le parcours était infixe ! En effet, nous pouvons éliminer d'entrée de jeu le parcours postfixe, car le traitement d'un nœud est suivi d'un appel à **g_tree_node_next()**. Pour déterminer s'il est préfixe ou infixe, il faut réfléchir un peu. Le principe est d'aller du premier nœud aux suivants, donc en commençant chaque branche par son extrémité gauche. Cela revient à visiter tous les fils gauches avant de les traiter et non le contraire. Nous avons donc bien un parcours infixe.