Rapport Projet N°2

Rapport portant sur le cours *Technique de Programmation*

Document rédigé par **David Taralla**
2e Bachelier en Sciences Informatiques

Dernière version : 1er mai 2011

Table des matières

Partie 1

Présentation de la fonction d'insertion dans un LLRBT

Cette partie explique par la méthode des décompositions successives l'algorithme d'insertion d'un nœud dans un arbre rouge-noir penchant à gauche (LLRBT [1]). J'y présenterai l'implémentation `C` de l'algorithme, avant d'expliquer l'insertion standard qui a lieu dans tout arbre binaire, pour terminer avec la complexification de cette insertion dans le cas des LLRBT.

1.1 Implémentation C de l'algorithme d'insertion d'un nœud dans un LLRBT

1.1.1 Fonction d'insertion proprement dite

Fonction ne faisant pas partie de l'interface. Appelée par la fonction `linsertNode()` de l'interface. Elle insère la structure `kv` dans le sous-arbre dont la racine est `n`.

```
static int cmpKeys(int a, int b) {
  return b - a;
}

static Node* putNode(Node* n, Keyval* kv) {
  /* -- First: insert the node at the right place --
     If the subtree where we put the node is empty, creating its root. */
  if (n == NULL) return create2Node(kv, RED);

  // Else record comparison result and increment comparisons counter
  int cmp = cmpKeys(kv->key, n->kv->key);
  t->nbCmp++;

  /* Then if key in node sup. to the key we want to add, put the node on
     its left child */
  if      (cmp > 0) n->lChild = putNode(n->lChild, kv);
```

1. Left Leaning Red-Black Tree

```
  /* Else if key in node inf. to the key we want to add, put the node on
     its right child */
  else if (cmp < 0) n->rChild = putNode(n->rChild, kv);

  // Else key in node equals key we want to add so replace existing value
  else strncpy(n->kv->value, kv->value, 20);


  /* -- Next: hold invariant true --
     #1: If red color not left-leaning for this node, rotate left */
  if (isRed(n->rChild) && !isRed(n->lChild)) n = rotateLeft(n);

  // #2: If red link succeeded by another one directly, rotate right
  if (isRed(n->lChild) && isRed(n->lChild->lChild)) n = rotateRight(n);

  // (case of two red links on the right never occures thx to #1)

  // #3: If both children are linked by red color to node, flip colors
  if (isRed(n->lChild) && isRed(n->rChild)) n = flipColors(n);

  // If a rotation occured, return the new root of the subtree
  return n;
}
```

1.1.2 Fonction d'insertion

```
int linsertNode(Tree* t, int key, char* value) {
  // -- First: creating key-value structure --
  Keyval* kv = (Keyval*)malloc(sizeof(Keyval));
  kv->key = key;
  strncpy(kv->value, value, 20);

  t->nbCmp = 0; // Reset comparisons counter

  // -- Next: calling insertion on the entire tree --
  t->root = putNode(t, t->root, kv);
  return t->nbCmp;
}
```

1.2 Étude de l'implémentation

1.1.2 Invariant conservé à chaque étape de l'insertion

Trois conditions très simples régissent l'invariant :

- La clé de chaque nœud est supérieure à celles du sous-arbre à gauche de celui-ci et est inférieure à celles du sous-arbre à droite de celui-ci.
- Aucun chemin reliant la racine au bas de l'arbre contient deux connexions rouges successives.
- Le nombre de connexions noires de chacun de ces chemins est identique.

1.2.2 Insertion standard

L'endroit où l'on doit insérer le nœud est déterminé de la même manière que dans un BST standard. Cet algorithme très simple est expliqué ci-dessous et constitue la première partie de l'insertion d'un nœud dans un LLRBT.

Tant que l'on n'a pas atteint une feuille, on descend dans l'arbre. Si la clé du nœud qu'on désire ajouter existe déjà, l'algorithme passera par le nœud la contenant et remplacera juste la valeur de ce nœud par la nouvelle sans en ajouter de nouveau. Si elle n'existait pas encore, une fois la bonne feuille atteinte, on détermine si le nouveau nœud sera son enfant à gauche ou à droite, puis on l'insère. Par défaut, la couleur de ce nœud est **rouge**.

On appelle l'insertion d'un nœud de clé k sur un arbre T.

1. Si l'arbre est vide, on insère le nœud dans l'arbre et on le considère *racine de* T. Aucune récursion nécessaire.
2. Sinon, on appelle l'insertion récursive sur la racine.

On appelle l'insertion d'un nœud de clé k sur un nœud n de l'arbre (racine d'un sous-arbre de T).

1. Si le nœud sur lequel l'insertion récursive est appelée est vide (`null`), c'est à cet endroit qu'on doit créer le nouveau nœud, ce qu'on fait. La récursion se termine.
2. Sinon,
   - Si k est strictement inférieur à la clé du nœud n, on appelle l'insertion récursive sur l'enfant gauche de celui-ci.
   - Sinon si k est strictement supérieur à la clé du nœud n, on appelle l'insertion récursive sur l'enfant droit de celui-ci.
   - Sinon on est dans le cas où k est égal à la clé du nœud n, il suffit donc de remplacer la valeur contenue dans celui-ci par celle du nouveau nœud. La récursion se termine.

1.2.3 Insertion tout en gardant une structure de LLRBT

Maintenant, il faut vérifier que l'équilibrage des connexions rouges et noires reste correct : en effet on a peut-être provoqué la création de deux connexions rouges successives dans le bas de l'arbre, ou un nœud dont les enfants sont tous les deux connectés par un lien rouge à leur parent. De plus, en remédiant à ces problèmes locaux, on en a peut-être créés de nouveaux...

On appelle l'insertion d'un nœud de clé k sur un arbre T.

1. Si l'arbre est vide, on insère le nœud dans l'arbre et on le considère *racine de* T. Aucune récursion nécessaire.
2. Sinon, on appelle l'insertion récursive sur la racine. La racine devient le nœud retourné par cette récursion.

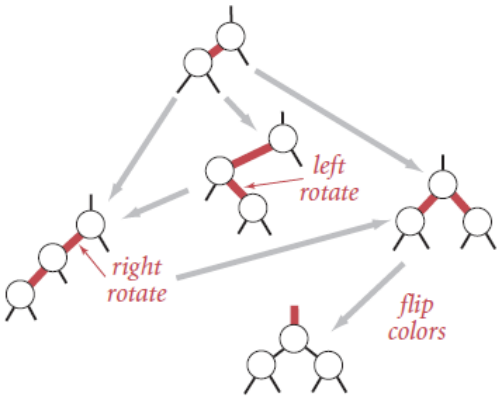
On appelle l'insertion d'un nœud de clé k sur un nœud n de l'arbre (racine d'un sous-arbre de T).

1. Insertion standard du nœud (expliqué en section 1.2.2). Soit m la feuille qui est devenue parent de n.

2. Si la connexion entre m et son enfant gauche est noire alors que celle vers son autre enfant est rouge, on effectue une rotation à gauche sur m (on fait « passer » la connexion rouge à gauche tout en respectant la condition 1 de l'invariant). m est alors permuté avec un de ses enfants, et ce dernier devient le nœud pointé par m.
3. Si la connexion entre m (qui peut être un nouveau nœud désormais à cause de l'étape précédente) et son enfant gauche est rouge alors que celle reliant son enfant de gauche à son petit-enfant l'est aussi, on effectue une rotation à droite sur m. m est alors permuté avec un de ses enfants, et ce dernier devient le nœud pointé par m.
4. Si après tout cela les connexions entre m (qui peut être un nouveau nœud désormais à cause des étapes précédentes) et ses deux enfants sont rouges, on complémente la couleur de chaque connexion (la connexion reliant m et son parent comprise !).

Les trois derniers points sont illustrés à la figure 1.1 qui résume bien ce qui se passe à chaque niveau de la récusion. La raison pour laquelle tout ceci fonctionne découle tout simplement du fait que cet algorithme est exécuté à chaque insertion et que chaque insertion conserve l'invariant.

Figure 1.1 – Les trois cas se présentant lors de l'insertion d'un nœud



Ceci termine mon étude par décompositions successives de l'algorithme d'insertion d'un nœud dans un LLRBT.

Partie 2

Comparaison des performances

Je présente dans cette partie un comparatif des performances de l'insertion dans un *Binary Search Tree* face à celles de l'insertion dans un *Left-Leaning Red-Black Tree*. J'y exprime la méthodologie adoptée avant de présenter les résultats que j'ai obtenus.

2.1 Méthodologie

L'étude des performances qui est présentée dans cette partie se fonde sur le **nombre de comparaisons** effectuées par l'un ou l'autre algorithme.

Pour avoir une idée du nombre moyen de comparaisons qui sont exécutées lors de l'insertion d'une valeur dans un arbre de N éléments aléatoires, il fallait créer ces arbres aléatoires avant toute autre chose. La fonction qui crée des arbres de taille N doit s'arranger pour que ces arbres aient bien N nœuds. Cela est fait en insérant l'un après l'autre les éléments d'un tableau contenant les entiers de l'intervalle $[0, N[$ mélangés aléatoirement. Le code de la fonction mélangeant le tableau $[0 : N-1]$ est fourni en Table 2.1.

Table 2.1 – Fonctions de mélange d'un tableau de N éléments

```
static void shuffle(int* theArr, int size) {
  srand((unsigned)time(NULL));
  int temporary, randomNum, last;
  for (last = size; last > 1; last--) {
    randomNum = rand( ) % last;
    temporary = theArr[randomNum];
    theArr[randomNum] = theArr[last - 1];
    theArr[last - 1] = temporary;
  }
}
```

Chaque moyenne du nombre de comparaison s'est faite sur 30 tableaux de taille identiques mais mélangés de manières différentes à chaque fois (grâce au changement de graine aléatoire

effectué dans la fonction `shuffle()`).

Les comparaisons observées se font à raison de une par appel (récursif) à la fonction d'insertion. Voir les implémentations en section 1.1 pour plus de détails.

L'élément inséré a toujours la clé de valeur $N + 1$.

2.2 Analyse et présentation des mesures

Première expérience Pour cette première expérience, on a considéré 8 tests. Chaque test consistait à l'insertion de l'élément de clé 100 001 dans un arbre de taille 100 000, et ce répété 30 fois avec des arbres de 100 000 éléments à chaque fois différents (bien que contenant les mêmes valeurs, celles de l'intervalle $[0, N[$). Les nombres présentés en Table 2.2 sont les moyennes du nombre de comparaisons effectuées pour chacune des trentaines de sous-tests.

Table 2.2 – Moyennes pour insertion dans des tableaux de 100 000 éléments

	Type								
Test		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	`BST`	13.70	10.47	11.90	12.97	11.30	12.27	11.90	11.60
	`LLRBT`	13.90	14.36	14.48	14.48	14.48	14.28	14.48	14.48

On remarque une très bonne stabilité moyenne des LLRBT pour ce qui est de l'insertion d'un élément dans un arbre aléatoire de taille 100 000. On la vérifie avec le même test, mais sur des arbres aléatoires de 10 000 éléments (voir Table 2.3).

Table 2.3 – Moyennes pour insertion dans des tableaux de 10 000 éléments

	Type								
Test		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	`BST`	6.07	8.00	13.00	13.33	13.00	7.93	12.53	11.67
	`LLRBT`	11.48	11.38	11.38	11.38	11.38	11.38	11.38	11.38

De plus, on observe bien (comme prévu par les études théoriques des LLRBT) que le nombre de comparaisons est proportionnel au logarithme en base 2 du nombre de nœuds dans l'arbre (et même inférieur à celui-ci). En effet,

$$\begin{aligned} \log_2 10000 &\approx 13.23 \\ \log_2 100000 &\approx 16.61 \end{aligned}$$

Seconde expérience Pour cette seconde expérience, on a considéré 4 tests pour des arbres de tailles différentes. Le but ici était de comparer le nombre de comparaisons effectuées par les deux algorithmes dans le pire des cas pour le BST : un arbre filiforme dans lequel on insère un élément plus grand que tous les autres. Les résultats de ce test apparaissent dans la Table 2.4.

Comme on pouvait s'y attendre grâce à l'étude théorique, le nombre de comparaisons nécessaires dans un arbre filiforme comprenant N nœuds est N pour un BST.

De plus, le choix des tailles n'est pas innocent : il a été fait pour qu'on remarque bien que le nombre de comparaisons effectuées par l'insertion dans un LLRBT était inférieur ou égal à $\log_2 N$.

Rien que pour ce cas de dégénérescence (dont les cas approchés sont plus nombreux qu'on ne le pense), le LLRBT apparaît ainsi comme une nécessité afin d'assurer de bonnes performances moyennes.

Table 2.4 – Nombre de comparaisons pour insertion dans le pire des cas

	Type				
Tailles		2^{10}	2^{13}	2^{14}	2^{15}
	`BST`	2^{10}	2^{13}	2^{14}	2^{15}
	`LLRBT`	10	13	14	15

2.3 Conclusion tirée des résultats

Tout d'abord, on déduira que si ce qui nous importe, c'est d'avoir une moyenne d'insertion stable (et non un temps d'insertion variable comme dans les BST), l'utilisation des LLRBT sera préconisée. Par contre, si on est sûrs de travailler avec des arbres qui ne font pas dégénérer l'algorithme de recherche des BST, ce dernier pourrait être intéressant. Mais comme ce cas de figure n'arrive presque jamais et que cet algorithme n'est pas stable, en pratique, on préférera toujours les LLRBT. On en conclut donc en fait les résultats théoriques annoncés.

Pour compléter ces derniers, le mieux aurait été de tester l'algorithme d'insertion dans un LLRBT sur le pire des cas de celui-ci, c'est- à dire dans le cas où tous les nœuds de l'arbre sont noirs sauf ceux d'un seul chemin, où les nœuds rouges et noirs s'alternent. Malheureusement, je n'ai pas trouvé le moyen de créer un tel arbre, donc je n'ai pu présenter le test.