Licence Informatique **TD Algorithmique 3**
Université Bordeaux 1

Dictionnaire / Arbres AVL

Fichiers sources : `http://dept-info.labri.fr/ENSEIGNEMENT/algo3`

Dans ce TD, nous allons reprendre le type abstrait "dictionnaire", puis étudier et utiliser le type concret arbres AVL pour en donner une implémentation.

Au TD précédent, nous avons utilisé les arbres binaires de recherche (ABR) comme type concret pour le type abstrait dictionnaire. Nous avons mentionné et constaté de manière expérimentale que le temps *moyen* d'insertion dans un ABR est de l'ordre de $O(\log N)$ si l'arbre contient déjà N sommets. Cela dit, l'arbre peut parfois produire une forme dégénérée et produire des temps d'insertion de l'ordre de $O(N)$. C'est le cas lorsque les séquences d'éléments à insérer sont totalement ordonnées ; les séquences croissantes produisent des segments de branches droites, et les séquences décroissantes produisent des segments de branches gauches. Incidemment, même si une séquence n'est pas totalement ordonnées, les sous-séquence qui le sont introduisent dans l'arbre des segments de branches et pénalisent ainsi les performances.

Les arbres AVL apportent un remède à cette situation. Ces arbres sont définis ainsi :

Un arbre AVL est un arbre binaire de recherche $T = (T', a, T'')$ tel que les hauteurs des sous-abres T' et T'' diffèrent au plus de 1, et tel que les sous-arbres T' et T'' sont aussi des arbres AVL.

L'objectif visé est donc d'avoir des arbres équilibrés, aussi proches que possible d'un arbre binaire parfait. Avec ces arbres, l'opération d'insertion se fait *dans le cas le pire* en temps $O(\log N)$.

Exercice 1. - Rotations

Les opérations d'insertion et de deletion dans les rabres AVL se font en deux temps :

- on insère ou retire l'élément comme on le fait pour les arbres binaires de recherche, sans tenir compte des hauteurs relatives des sous-arbres ;
- au besoin, on rétablit l'équilibre des sous-arbres.

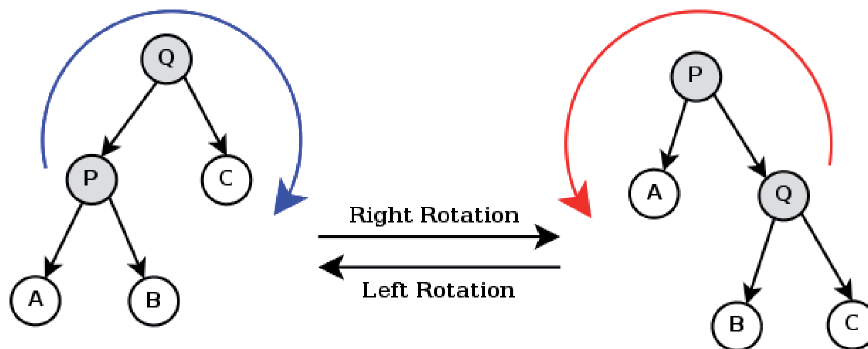

FIG. 1 – Opération de rotation sur un arbre.

Par conséquent, la question qui se pose lorsqu'on utilise les arbres AVL et que l'on considère les opérations d'insertion et de deletion est de maintenir l'équilibre des sous-arbres gauches. Le rétablissement de l'équilibre se fait en utilisant (et en composant) des *opérations de rotations*.

1. La figure décrit l'effet sur un arbre de l'opération de rotation. Montrer que lorsque l'arbre est un arbre binaire de recherche (ABR), alors l'arbre obtenu par rotation gauche ou droite est aussi un ABR.

2. IL faut maintenant étudier les situations où il nous faudra effectuer un ré-équilibrage des arbres. Notez que parce que l'insertion/deletion d'effectuera dans un arbre AVL, les seules situations de déséquilibres sont celles où les sous-arbres sont de hauteur respectives h et $h+2$ (ou vice-versa). Ces cas sont résumés par la figure suivante :

FIG. 2 – Situations possibles de déséquilibre (après insertion, par exemple).

A chaque fois ; l'arbre déséquilibré est en première ligne et l'arbre équilibré résultant apparaît au bas. Exprimer chacun de ces cas sous la forme d'opérations de rotation appliqué à l'arbre de départ.

Exercice 2. - Implémentation avec les arbres AVL

Nous effectuons quelques hypothèses concernant notre dictionnaire.

- Nous interdisons de stocker l'objet NULL dans le dictionnaire, qui sert de valeur de retour en cas d'erreur, notamment lorsque la recherche d'un objet échoue.
- Nous interdisons les doublons dans le dictionnaire.
- Les clés sont de type non connus.

– L'utilisateur nous fourni une fonction de type `compare` qui permet de comparer deux clés pour savoir si la première est strictement inférieure, égale ou supérieure à la seconde. Cette fonction encapsule en quelque sorte l'ordre total sur les clés. On conviendra que la fonction retourne une valeur négative, zéro ou positive selon le cas recontré.

Voilà l'interface C que nous proposons pour le dictionnaire, ou table associative (*map* en anglais).

```
struct map;

typedef struct map *map;

/* map an integer key to an object pointer */
typedef int (*compare)(void *, void *);

/* create an empty map */
extern map map_create(compare f);

/* find an object in the map and return it or NULL if not found */
extern void *  map_find(map m, void * key);

/* insert an object in a map and return it or NULL in case of
   failure (NULL object or object already inside the map) */
extern void * map_insert(map m, void * object);

/* delete an object from a map and return it or NULL in not found */
extern void * map_delete(map m, void * key);
```

L'objectif de cet exercice est d'implanter l'interface proposée en utilisant les arbres AVL. Vous souhaiterez implémenter en interne les fonctions réalisant le ré-équilibrage des arbres, et notamment les opérations de rotations gauche et droite pour faciliter la lecture et la modularité du code.

Afin de faciliter l'analyse et le debuggage des algorithmes, on pourra implanter la fonction suivante qui représente l'arbre binaire de recherche sous-jacent.

```
/* dump the underlying binary research tree */
extern void map_dump(map m);
```

Exercice 3. - Efficacité des arbres AVL Quelle est le nombre d'éléments et la profondeur maximale d'un arbre AVL lorsque l'espace disponible est de 2 Go ? En déduire le nombre d'étapes nécessaires pour la recherche, l'insertion, et la deletion d'un élément d'un arbre de cette taille occupant cette espace mémoire.