IFT339 Automne 2014 Laboratoire 4

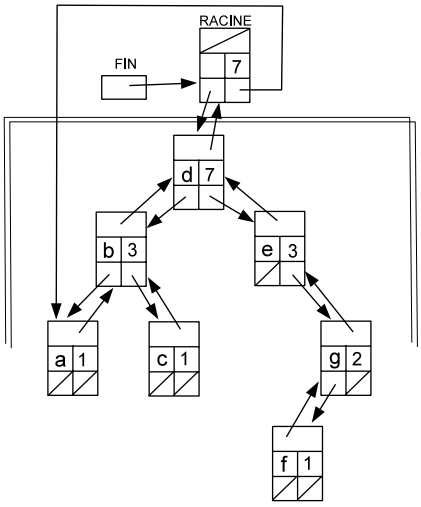
L'équilibre des arbres binaires

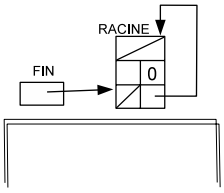
Objectifs de ce laboratoire

Dans ce laboratoire, nous allons gérer l'équilibre d'un arbre binaire de recherche. À partir du code de base pour un arbre binaire de recherche, nous allons ajouter les fonctions d'équilibre en poids.

Description de la tâche à réaliser

On vous fournit le code de la classe générique WBT (Weight-balanced Tree). Tout fonctionne déjà pour l'insertion, l'élimination, l'itération, etc. Il ne manque que l'équilibre. À chaque fois qu'on modifie le nombre de nœuds d'un des deux sous-arbres d'un nœud donné, on appelle la fonction `equilibrer` en lui passant par référence le pointeur qui pointe vers le noeud à vérifier. Cette fonction est appelée dans le cadre de l'insertion et de l'élimination. Vous n'avez que trois fonctions à coder: `equilibrer`, `rotation_droite_gauche` et `rotation_gauche_droite`. Tout le reste fonctionne. Vous disposez en particulier d'une fonction afficher qui vous donne tout le détail de la représentation d'un arbre, pour vérifier que votre code fait le bon travail. Voici une illustration d'un arbre quelconque (de 7 éléments) et d'un arbre vide.





Dans cette représentation, chaque nœud comprend un contenu, le poids de ce nœud, un pointeur vers son sous-arbre de gauche, un pointeur vers son sous-arbre de droite, et un pointeur vers le parent du nœud. la partie automatique comprend un nœud (RACINE) qui contient le nombre total de nœuds de l'arbre. Ce nœud particulier pointe sur la vraie racine de l'arbre et sur son premier élément. Son parent est 0. L'intérêt de cette représentation est d'unifier le traitement de la fin dans l'itération. Un pointeur vers cette cellule est en effet la représentation de la fin. Reculer de cette position est le même code que reculer de n'importe quelle autre position. Et cette position est naturellement la suivante de celle du dernier élément.

Vous pouvez choisir la règle d'équilibre de votre choix. Voici une suggestion possible : tant qu'un nœud contient moins de trois éléments, on ne se préoccupe pas de son équilibre. On procède à une rotation si un sous-arbre en vient à contenir plus de trois fois plus d'éléments que l'autre. On fait une première rotation pour transférer du poids du centre vers le côté si la rotation simple ne fait que transférer le déséquilibre de l'autre côté.

Il n'y a pas tant de code à faire ici. Il faut faire très attention aux détails bien sûr, et bien traiter les cas limites. Une idée possible: ne vous occupez dans un premier temps que du déséquilibre d'un des deux côtés. Quand cela fonctionnera parfaitement, vous pourrez faire l'équivalent de l'autre côté. Testez continuellement votre code à chaque modification que vous codez.

Mardi, le 18 novembre, la période du cours sera consacrée à la préparation de votre code. Assurez-vous alors de bien comprendre le code existant, et de bien planifier SUR PAPIER les modifications à faire, AVANT de sauter sur le clavier!

Remise du travail

Ce travail doit être complété à 23 h 59 jeudi le 20 novembre au plus tard. Soumettez-le à `turnin`, après vous être assurés qu'il fonctionne bien sur tarin. Ne soumettez pas un répertoire, et attention à la syntaxe! Une seule soumission par équipe! Ne soumettez que votre fichier WBT.h. J'utiliserai mon propre programme principal. Ne changez pas le nom!

```
turnin -c ift339 -p labo4 WBT.h
```

Jean Goulet

Voici une partie de la classe WBT générique

```
template <typename TYPE>
class WBT{
public:
    class iterator;
    friend class iterator;
private:
    struct noeud{
       //un noeud de l'arbre
};
    noeud RACINE;
    noeud* FIN;
    //fonctions privees que vous devez coder
    void reequilibrer(noeud*&);
    void rotation_gauche_droite(noeud*&);
    void rotation_droite_gauche(noeud*&);
public:
    WBT();
    ~WBT(){clear();}
    WBT(const WBT&);
    WBT& operator=(const WBT&);
    void swap(WBT&);

    size_t size()const{return RACINE.POIDS;}
    bool empty()const{return RACINE.POIDS==0;}
    void clear();

    iterator find(const TYPE&)const;
    std::pair<iterator,bool> insert(const TYPE&);
    size_t erase(const TYPE&);
    iterator erase(iterator);

    //fonction d'iteration
    iterator begin()const{return iterator(RACINE.DROITE);}
    iterator end()const{return iterator(FIN);}

    //fonction de mise au point
    void afficher()const;
};
```

La fonction insert publique reçoit un objet à insérer. Elle retourne un objet composite: une paire comprenant un itérateur (la position de l'objet dans l'arbre) et un bool qui est vrai si l'élément n'était pas déjà présent dans l'arbre et a dû être ajouté. Cette fonction s'occupe du cas limite de l'insertion dans un arbre vide, et laisse à une fonction récursive privée la gestion du cas général.

La fonction erase peut prendre un objet à enlever ou un itérateur vers cet objet. La première version retourne 1 si on a effectivement enlevé l'objet, 0 sinon. Elle appelle la seconde, après avoir localisé l'objet en question. La seconde version retourne la position qui suit celle où on a effectué l'élimination.

Ces fonctions imitent les fonctions équivalentes du set et du map de la SL.

Une classe iterator complète est disponible aussi.

Vous disposez d'une fonction "afficher" qui vous permet de voir exactement comment se comporte votre arbre quand vous lui faites des modifications. Pour chaque nœud, on affiche le contenu, le poids, l'adresse du nœud ainsi que les pointeurs vers son parent, son enfant de gauche et son enfant de droite. La fonction essaie aussi de dessiner grossièrement la structure de l'arbre, sa partie droite étant en haut du dessin.