Dans les arbres binaires : exercice de synthèse en C++

Travaux dirigés et pratiques - PO2 (UE ULIN606)

1 Présentation

Nous allons étudier deux sortes d'arbres binaires étiquetés aux nœuds, les arbres binaires de recherche et les peignes pleins© (qui ont une structure particulière).
Ces arbres partagent un certain nombre de caractéristiques :
- d'un point de vue "spécification" : ils doivent répondre à des messages du même genre (sous-arbre gauche, droit, impression, insertion d'une valeur, recherche d'une valeur, ...)
- d'un point de vue "implémentation" : ils doivent pouvoir s'appuyer sur un même type de représentation interne, qui devrait même être utilisable par toute espèce d'arbre binaire.

Un *peigne plein* est :
- soit réduit à un nœud unique,
- soit un peigne *toutPlein*,
- soit un arbre de sous-arbre gauche vide et dont le sous-arbre droit est un peigne *toutPlein*.

Un peigne *toutPlein* est un arbre binaire tel que :
- le sous-arbre gauche est une feuille
- le sous-arbre droit est vide ou *toutPlein*.

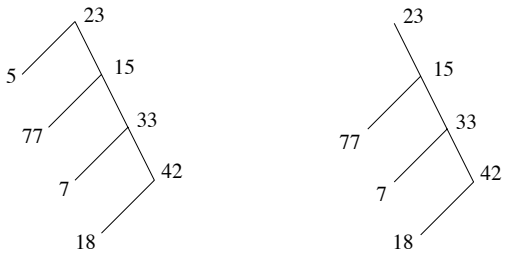

Fig. 1 – Deux exemples de peignes pleins

Le peigne de gauche a été construit à partir des données `42 18 33 7 15 77 23 5`, celui de droite à partir de `42 18 33 7 15 77 23`.

Comme vous le savez déjà, un *arbre binaire de recherche* est un arbre binaire dont les clés sont ordonnées de la façon suivante : si x est un sommet quelconque de clé c, et X l'arbre enraciné en x, toute clé c' du sous-arbre gauche de X est telle que $c' < c$ et toute clé c'' du sous-arbre droit de X est telle que $c < c''$.
On suppose qu'on ne trouve pas deux fois la même valeur dans l'arbre (les clefs sont uniques).

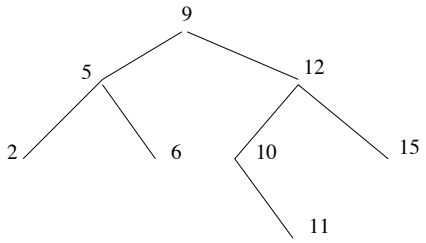

Fig. 2 – Un arbre de recherche

Cet arbre a été construit à partir des données `9 5 2 6 12 10 11 15`.

Une fois n'est pas coutume, nous n'allons pas nous lancer dans une spécification complète des classes avant tout travail sur l'implémentation, *mais* cela ne nous empêchera pas de bien prêter attention à ne pas mélanger les deux. Nous allons avancer selon des étapes qui vous permettront, lors du TP, de tester au fur et à mesure les méthodes.

2 Un type de données récursif

Les arbres binaires ont une (jolie) définition récursive naturelle sur laquelle nous allons nous appuyer pour la représentation interne de la classe et l'écriture des méthodes.
Dit rapidement, un arbre binaire est soit vide, soit composé de trois éléments : une clé, un arbre (le sous-arbre gauche), un autre arbre (le sous-arbre droit).
On souhaite utiliser les déclarations partielles C++ suivantes :

```
class Noeud{
  private:
  TypeCle cle; Arbre * filsGauche, * filsDroit;...}
```

La classe *Arbre* contiendra un attribut *racine*, de type pointeur sur un *nœud*, et déclaré `protected`.
Un arbre vide sera représenté par un arbre dont l'attribut *racine* a pour valeur NULL.

1. Dessinez la représentation en mémoire de l'arbre binaire de recherche et des deux peignes donnés en exemple dans le paragraphe précédent.
2. Proposez un diagramme de classes qui contient ce qu'il faut pour représenter les types d'arbres binaires décrits dans la présentation. Pour le moment, les classes ne doivent contenir que les attributs (en accord avec les déclarations C++ ci-dessus) et les constructeurs.

3 Début d'implémentation

3.1 Information préliminaire

Vu les types des attributs des classes *Nœud* et *Arbre*, il faudrait inclure le fichier `Arbre.h` dans le fichier `Noeud.h`, et le fichier `Noeud.h` dans le fichier `Arbre.h`. Évidemment, ce n'est pas possible, et ce n'est pas accepté à la compilation. On va donc utiliser la syntaxe suivante :

fichier *Noeud.h*

```
....
class Arbre;
class Noeud{...};
....
```

fichier *Arbre.h*

```
....
#include "Noeud.h"
class Arbre{...};
....
```

fichier *Noeud.cc*

```
....
#include "Noeud.h"
#include "Arbre.h"
....
```

fichier *Arbre.cc*

```
....
#include "Arbre.h"
....
```

La déclaration `class Arbre ;` annonce qu'une classe *Arbre* sera définie. C'est suffisant pour que la déclaration de la classe *Nœud* soit compréhensible, donc il n'y a pas besoin d'autre chose dans `Noeud.h`. Par contre, on ajoute `#include "Arbre.h"` dans `Noeud.cc` : si on fait appel à une méthode de *Arbre*, il faut avoir la déclaration complète de la classe, et pas seulement son nom[1].

3.2 Classe Nœud

Écrivez en C++ la classe complète, qui contient uniquement constructeurs, destructeur, et accesseurs aux attributs.
Pour la clé, les accesseurs sont ceux habituellement utilisés.
Pour les fils, on met uniquement les accesseurs suivants :

Dans le `.h` :

```
virtual Arbre*& refFilsG();
virtual Arbre*& refFilsD();
```

Dans le `.cc` :

```
Arbre*& Noeud::refFilsG() {return FilsG;}
Arbre*& Noeud::refFilsD() {return FilsD;}
```

Ce type d'accesseur renvoie l'adresse de l'attribut au lieu de renvoyer une copie de sa valeur. Voir la raison d'utilisation dans la sous-section suivante.

[1] En fait, dans le cas particulier de cet exercice, vous n'aurez certainement pas besoin des méthodes de *Arbre* dans la classe *Nœud*, et vous pourriez ne pas mettre cet *include*. Mais ainsi vous avez la syntaxe complète pour un cas général d'inclusions croisées ou circulaires.

3.3 Classe Arbre : opérations de base

On veut pouvoir appliquer aux arbres représentés les opérations suivantes :
- *sag* : renvoie l'adresse du sous-arbre gauche ;
- *sad* : renvoie l'adresse du sous-arbre droit ;
- *clef* : renvoie la valeur contenue dans la racine ;
- *estVide* : renvoie *vrai* si l'arbre est vide, *faux* sinon ;
- *feuille* : renvoie *vrai* si l'arbre est réduit à un nœud unique, *faux* sinon.

Écrivez en C++ la classe Arbre munie de ces opérations.
Les méthodes *sag* et *sad* renvoient l'adresse de la racine du sous arbre, et non une copie de cette racine, pour deux raisons :

- éviter de faire continuellement des copies de pointeurs inutiles quand on fait des parcours dans les arbres (ce cas se présente beaucoup dans ce tp) ;
- donner un accès fugitif à une case de l'arbre (ce cas ne se présente pas dans ce tp, mais est logique si on veut que la représentation soit suffisamment générale).

4 Insertion

On veut pouvoir insérer des clefs dans les arbres représentés.
L'insertion dans le peigne doit préserver sa structure (voir les deux exemples). L'insertion dans l'arbre de recherche doit préserver l'ordre des éléments (vous l'avez vue en algorithmique).
Ajoutez les méthodes nécessaires.

5 Affichage

Ecrivez les méthodes nécessaires pour que l'affichage d'un peigne soit sous le format suivant (avec l'exemple de droite de la Figure 1) :

```
Je suis un peigne plein
J'affiche mes clefs dent par dent
----------------------
Gauche = .   Cle = 23
Gauche = 77  Cle = 15
Gauche =  7  Cle = 33
Gauche = 18  Cle = 42
```

... et l'affichage d'un arbre de recherche sous celui-ci (avec l'exemple de la Figure 2) :

```
Je suis un arbre de recherche
J'affiche mes clefs par ordre croissant
----------------------
2 5 6 9 10 11 12 15
```

Prévoyez également un affichage indenté des arbres (où le placez-vous dans la hiérarchie ?), qui donnerait sur l'arbre de recherche :

```
 9
  5
   2
    .
    .
   6
    .
    .
  12
   10
    .
    11
     .
     .
   15
    .
    .
```

Quand ces méthodes seront au point ajoutez ce qu'il faut pour pouvoir écrire :
`cout << B` où B est une instance de n'importe laquelle de vos classes arbres.

6 Recherche

Ecrivez les méthodes qui renvoient *vrai* si une clef donnée est présente dans un arbre, *faux* sinon.

7 Destruction, Copie

Ecrivez les destructeurs, vous pouvez vous lancer dans la destruction récursive ...
Écrivez les constructeurs par copie et les opérateurs =

8 Exceptions

Etudiez les différentes erreurs pouvant se produire dans les méthodes des arbres binaires, représentez-les par des exceptions que vous signalerez et tâcherez de récupérer dans un programme.

9 Généricité paramétrique

Dans les exemples qui vous ont été proposés, les clefs sont des entiers. Proposez des arbres binaires paramétrés par le type des étiquettes. Quelle contrainte doit vérifier le type des étiquettes dans le cas des arbres binaires de recherche ? Pouvez-vous l'exprimer en C++ ?