IUT2 Université Grenoble Alpes
Département INFO

R3.02 – Développement efficace

TP 2 *(associé au Cours 2)*

Arbre binaire ordonné (ou arbre binaire de recherche
(1 séance encadrée et travail personnel)

Pour ce TP

- Télécharger et compléter le projet CLion « R302-TP2 »
- On fournit une configuration d'exécution par exercice.

Arbre binaire de recherche (ABR)

On propose la classe abstraite modèle **ArbreBinaireInterface** (fichier `ArbreBinaireInterface.h`) qui réalise le Type Abstrait de Données **ArbreBinaire**.

On propose une implantation partielle de la classe modèle **ArbreNoeudBinaireRecherche** (fichier ArbreNoeudBinaireRecherche`.h`) qui réalise le TAD Liste sous forme d'arbre de **NoeudBinaire**.

Note importante :

Veiller toujours à faire apparaître l'algorithme sous forme de commentaire après le prototype de la méthode à compléter.

Exercice 1 : Problèmes simples

I. Insertion associative dans un ABR (récursif)

L'insertion d'une nouvelle information (`nouvelleInfo`) dans `abr` un ABR de telle sorte qu'il demeure un ABR se définit comme suit :

- Si `abr == nullptr` (arbre vide), il suffit de créer un nœud d'adresse `abr`, contenant l'information `nouvelleInfo`.
- Sinon, on parcourt l'arbre, afin de rechercher un nœud qui sera le père du nœud contenant `nouvelleInfo` à insérer. Ce nœud père est tel que :
  - `(pere->getInfo() ≤ nouvelleInfo et pere->getPtrFilsDroit() == nullptr)` ou
  - `(pere->getInfo() > nouvelleInfo et pere->getPtrFilsGauche() == nullptr)`

Père est toujours au bout de la filiation : c'est soit une feuille soit un nœud qui n'a qu'un seul sous-arbre.

On fournit la méthode publique suivante de la classe **ArbreNoeudBinaireRecherche** :

```
void ArbreNoeudBinaireRecherche<TypeInfo>::insere(const TypeInfo& nouvelleInfo) {
    insertWorker(ptrRacine, nouvelleInfo);
}
```

Implanter le worker privé récursif qui fait le travail.

```
void ArbreNoeudBinaireRecherche<TypeInfo>::insertRecWorker(
                        NoeudBinaire<TypeInfo>*& ptrRac, const TypeInfo& nouvelleInfo);
```

Tester avec la procédure `testeInsereAffiche()` de **exercice 1**.

II. Nombre de nœuds d'un arbre binaire (récursif)

On fournit la méthode publique suivante de la classe **ArbreNoeudBinaireRecherche** :

```
/**
  * @return nombre de noeuds de cet arbre
  */
int ArbreNoeudBinaireRecherche<TypeInfo>::getNombreDeNoeuds() const {
    return getNombreDeNoeudsWorker(ptrRacine);
}
```

Implanter le worker privé récursif qui fait le travail.

```
int ArbreNoeudBinaireRecherche<TypeInfo>::getNombreDeNoeudsRecWorker(
                                                NoeudBinaire<TypeInfo>* ptrRac) const;
```

Tester avec la procédure `testeNbNoeuds()` de **exercice 1**.

III. Hauteur d'un arbre binaire (récursif)

On définit la hauteur d'un arbre binaire comme suit :

- La hauteur d'un arbre est égale à la hauteur de sa racine
- La hauteur d'un arbre vide est nulle
- La hauteur d'un nœud est égale au maximum des hauteurs du sous-arbre gauche et du sous-arbre droit plus un

On fournit la méthode publique suivante de la classe **ArbreNoeudBinaireRecherche** :

```
int ArbreNoeudBinaireRecherche<TypeInfo>::getHauteur() const {
    return getHauteurWorker(ptrRacine);
}
```

Implanter le worker privé récursif qui fait le travail.

```
int ArbreNoeudBinaireRecherche<TypeInfo>::getHauteurRecWorker(
                                                NoeudBinaire<TypeInfo>* ptrRac) const;
```

Tester avec la procédure `testeHauteur()` de **exercice 1**.

IV. Plus grande valeur dans un ABR (récursif)

On fournit la méthode publique suivante de la classe **ArbreNoeudBinaireRecherche** qui lève une exception `PrecondViolatedExcep` si l'arbre est vide :

```
TypeInfo ArbreNoeudBinaireRecherche<TypeInfo>::getMax() const;
```

Implanter l'algorithme du worker récursif qui fait le travail.

```
TypeInfo ArbreNoeudBinaireRecherche<TypeInfo>::getMaxRecWorker(
                                    const NoeudBinaire<TypeInfo>* ptrRac) const;
```

Tester avec la procédure `testeGetMax()` de **exercice 1**.

Exercice 2 : Problèmes plus originaux

I. Deux arbres ont-ils la même géométrie (récursif)

On fournit la méthode publique suivante de la classe **ArbreNoeudBinaireRecherche** :

```
/**
  * @return  true si cet arbre et autreArbre ont le même dessin (sans les informations)
  */
bool ArbreNoeudBinaireRecherche<TypeInfo>::aMemeGeometrieQue(
                        const ArbreNoeudBinaireRecherche<TypeInfo>& autreArbre) const ;
```

Implanter l'algorithme du worker récursif qui fait le travail.

```
bool ArbreNoeudBinaireRecherche<TypeInfo>::aMemeGeometrieQueRecWorker(
                const NoeudBinaire<TypeInfo>* monPtrRac,
                const NoeudBinaire<TypeInfo>* sonPtrRac) const;
```

Tester ce worker avec la procédure `testeAMemeGeometrieQue()` de **exercice 2**.

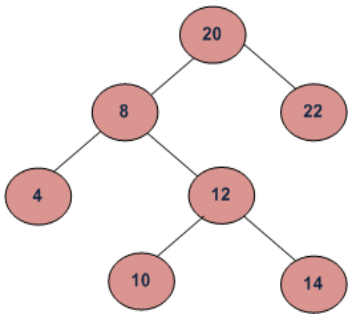
II. Ancêtre commun le plus bas (récursif)

Selon Wikipedia, l'ancêtre commun le plus bas entre deux nœuds n1 et n2 d'un arbre binaire de recherche A est défini comme le nœud le plus bas dans A qui a à la fois n1 et n2 comme descendants (où on permet à un nœud d'être un descendant de lui-même).

L'ancêtre commun le plus bas de n1 et n2 dans A est l'ancêtre commun de n1 et n2 qui est situé le plus loin de la racine. Le calcul des ancêtres communs les plus bas peut être utile, par exemple, dans le cadre d'une procédure visant à déterminer la distance entre des paires de nœuds dans un arbre : la distance de n1 à n2 peut être calculée comme la distance de la racine à n1, plus la distance de la racine à n2, moins deux fois la distance de la racine à leur ancêtre commun le plus bas.

Implanter une méthode publique (`TypeInfo getAncetreCommunLePlusBas(TypeInfo Val1, TypeInfo val2)`) et son worker récursif dans la classe **ArbreNoeudBinaireRecherche** qui retourne la valeur de l'ancêtre commun le plus bas de `val1` et `val2`.

Exemple : Pour l'arbre



On doit obtenir

- Ancêtre commun le plus bas de 10 et 14 → 12
- Ancêtre commun le plus bas de 4 et 14 → 8
- Ancêtre commun le plus bas de 8 et 14 → 8
- Ancêtre commun le plus bas de 4 et 22 → 20

On fournit la méthode publique suivante de la classe **ArbreNoeudBinaireRecherche** :

```
/**
  * @return  valeur de l'ancêtre commun le plus bas de val1 et val2
  */
TypeInfo ArbreNoeudBinaireRecherche<TypeInfo>::getAncetreCommunLePlusBas(
                        const TypeInfo& val1, const TypeInfo& val2) const ;
```

Implanter l'algorithme du worker récursif qui fait le travail.

```
TypeInfo ArbreNoeudBinaireRecherche<TypeInfo>:: getAncetreCommunLePlusBasRecWorker(
                const NoeudBinaire<TypeInfo>* ptrRac,
                const TypeInfo& val1, const TypeInfo& val2) const;
```

Tester ce worker avec la procédure `testeAncetreCommunLePlusBas()` de **exercice 2**.

III. Un arbre est-il sous-arbre d'un autre arbre

Soit R_s la racine d'un arbre S. L'arbre S est un sous-arbre d'un arbre A si A possède un sous-arbre de racine R_s avec la même géométrie et le même contenu que S.

On fournit la méthode publique suivante de la classe **ArbreNoeudBinaireRecherche** :

```
/**
  * @return  true si unArbre est un sous-arbre de cet Arbre
  */
bool ArbreNoeudBinaireRecherche<TypeInfo>::aPourSousArbre(
                        const ArbreNoeudBinaireRecherche<TypeInfo>& unArbre) const ;
```

Implanter l'algorithme du worker récursif qui fait le travail.

```
bool ArbreNoeudBinaireRecherche<TypeInfo>:: aPourSousArbreRecWorker(
                const NoeudBinaire<TypeInfo>* ptrRacCetArbre,
                const NoeudBinaire<TypeInfo>* ptrRacUnArbre) const;
```

Note en vous inspirant du worker `bool aMemeGeometrieQueRecWorker(…)` vous pouvez écrire un worker privé récursif `bool estEgalARecWorker(…)` qui vérifie que deux arbres sont égaux en géométrie et en contenu.

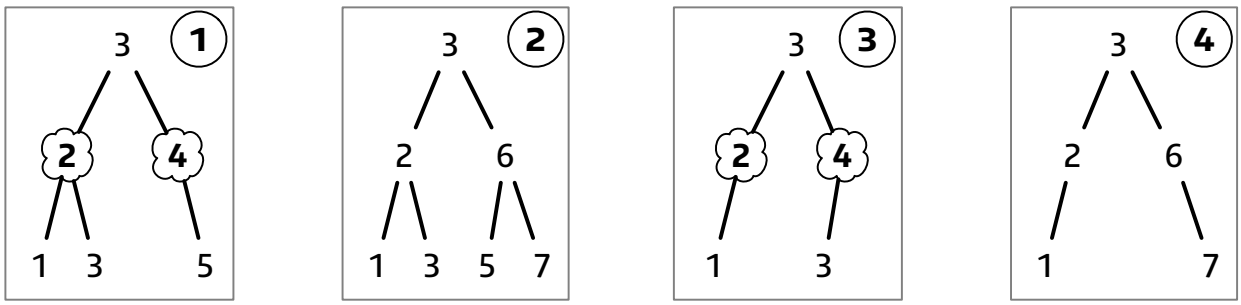
Tester ce worker avec la procédure `testeSousArbre()` de **exercice 2**.

IV. Arbre binaire pliable (récursif) *(exercice d'un contrôle machine de 2018)*

Un arbre binaire est pliable si ses deux sous-arbres gauche et droit sont des images en miroir l'un de l'autre sur le plan de la structure.

Un arbre vide est considéré comme pliable.

Exemples :



Les arbres 2 et 4 sont pliables.

Les arbres 1 et 3 ne sont pas pliables, les violations sont marquées en gras avec nuage.

Compléter le code de la méthode publique `bool estPliable() const`; pour accomplir cette tâche.

Cette méthode publique fera appel à un worker récursif qui sera nommé `estPliableRecWorker(…)` pour lequel il faut renseigner l'entête dans la définition de la classe `ArbreNoeudBinaireRecherche` et le code dans le fichier `ArbreNoeudBinaireRecherche.h`.

Note il faut bien réfléchir aux paramètres du worker `estPliableRecWorker(…)`.

Indice il en faut deux 😉

Tester `estPliable()` avec la procédure `testEstPliable()` de **exercice 2**..

Annexes

Annexe 1 : arbre nœud binaire de recherche

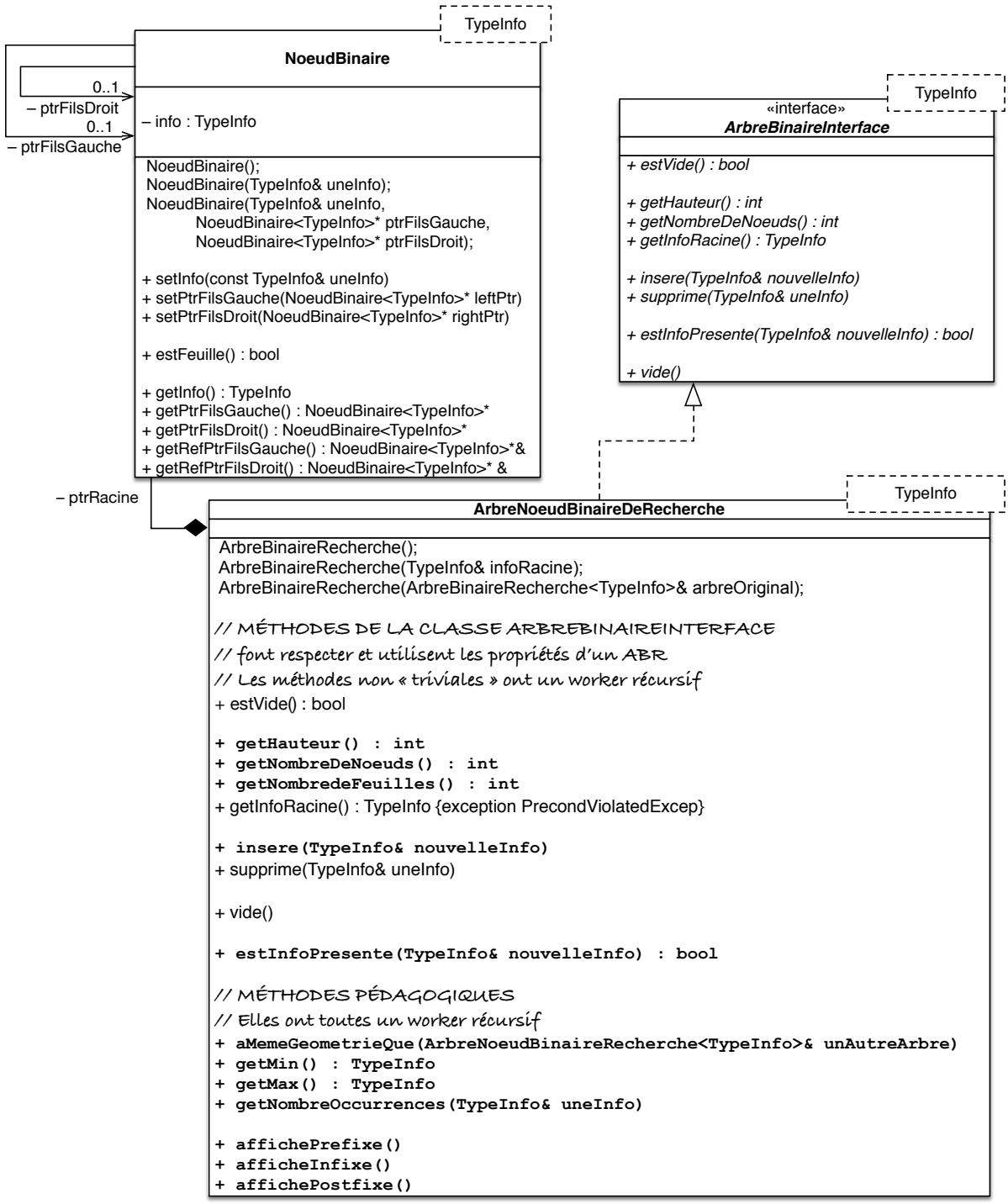


Figure 1 : Spécification UML partielle de la classe ArbreNoeudBinaireDeRecherche