Examen C++ Avancée

Documents non autorisés

Exercice :

1. Définir en utilisant la STL, un ensemble (*set*) de caractères contenant trois éléments (par exemple : 'a', 'b' et 'c').

2. Parcourir cet ensemble à l'aide d'un itérateur pour l'afficher élément par élément.

3. Définir une chaîne de caractères (*string*) et utiliser l'algorithme copy pour copier le contenu de l'ensemble dans la chaîne de caractères.

4. Terminer en affichant l'ensemble directement à l'écran en utilisant l'algorithme *copy* et un *ostream_iterator*.

Problème :

L'objectif de ce problème est d'implémenter un modèle de classe ***Arbre*** permettant de regrouper une collection d'objets du même type ***T***, sous forme d'une structure d'arbre binaire. Ce type ***T*** dispose par hypothèse un constructeur, un destructeur, ainsi que des opérateurs d'affectation et de comparaison.

On souhaite implémenter les opérations suivantes sur un objet de la classe arbre :

- Création et destruction.
- Ajout/retrait d'un élément.
- Nombre d'éléments de l'arbre.
- Calcul de la hauteur de l'arbre.
- Calcul du nombre de feuilles.
- Affichage de l'arbre en parcours infixe, préfixe, postfixe
- Vidage d'un arbre.

On verra ensuite une extension aux cas des arbres binaires de recherche.

1. Définitions

Un arbre est constitué d'un ensemble de nœuds auxquels sont associés des valeurs (de type ***T***). Un arbre binaire est défini récursivement de la manière suivante : un arbre binaire est composé

- soit d'un seul sommet appelé racine,
- soit d'un sommet racine à la gauche duquel est accroché un sous-arbre binaire gauche,
- soit d'un sommet racine à la droite duquel est accroché un sous-arbre binaire droit,
- soit d'un sommet racine auquel sont accrochés un sous-arbre binaire droit et un sous-arbre binaire gauche.

Ces différentes configurations sont illustrées dans la figure 1.

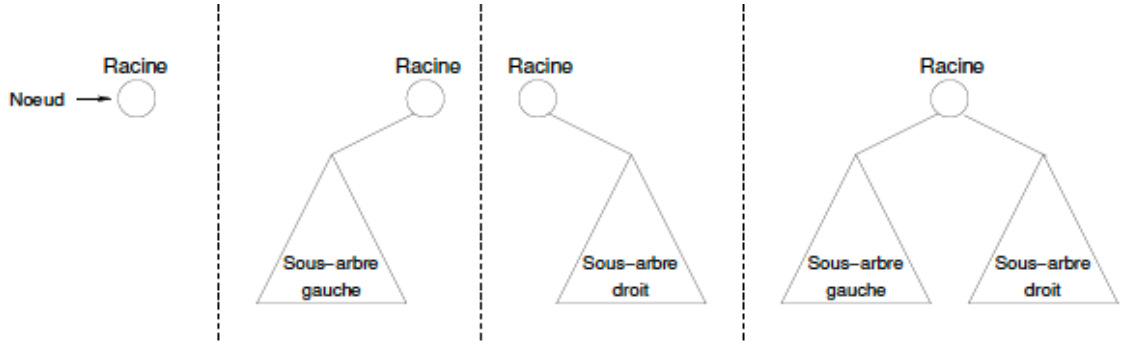

Figure 1 : Définition récursive d'un arbre.

Soit A un arbre. A = ∅ désignera un arbre vide et |A| le nombre de nœuds de l'arbre. Soit x un nœud. On définit alors les termes suivants :

Fils gauche de x : le sommet (s'il existe) accroché à la gauche de x
Fils droit de x : le sommet (s'il existe) accroché à la droite de x
Fils de x : le ou les deux sommets accrochés sous x
Sous-Arbre de sommet x : le sous-arbre de A de racine x
Père de x : le sommet (s'il existe) p tel que x est fils de p
Frère de x : un sommet (s'il existe) qui a le même père
Feuille : un sommet qui n'a pas de fils
Branche : un chemin de fils en fils de la racine vers une feuille
Hauteur de x : la longueur (en nombre d'arcs) du chemin allant de de x à la racine. En particulier, la hauteur du nœud racine est 0.
Hauteur d'un arbre : la hauteur maximal (notée h(A)) des feuilles de l'arbre.

On remarquera que la racine de l'arbre n'a pas de p`ere et c'est le seul sommet dans ce cas. La figure 2 illustre ces définitions sur un arbre binaire hébergeant des nœuds de type ***int*** (***T=int***). En particulier sur cet exemple, le père du nœud 2 est le nœud 5 ; le frère du nœud 7 est le nœud 2 ; le nœud 14 n'a pas de frère ; le nœud 12 n'a qu'un fils qui est le nœud 14.

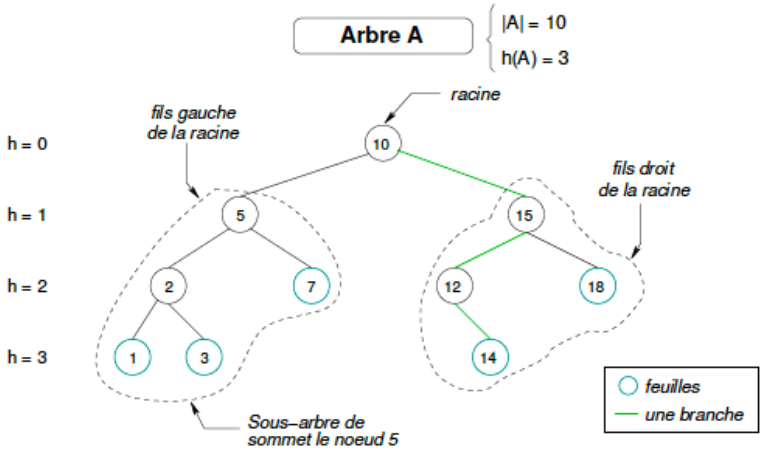

Figure 2 : Illustration des définitions sur un arbre binaire d'entiers.

2. Classe Nœud

Chacun des éléments de l'arbre est représenté à l'aide d'un objet de la classe Nœud. On adoptera une représentation sous forme de liste chaînée. Aussi, un nœud sera composé des éléments suivants :

- un champ ***val*** contenant une valeur (de type ***T***),
- un champ ***fg*** correspondant à un pointeur sur un nœud (le fils gauche)
- un champ ***fd*** correspondant à un pointeur sur un nœud (le fils droit)
- un champ **p** correspondant à un pointeur sur un nœud (le père).

La figure 3 illustre ce chaînage sur l'exemple de l'arbre décrit dans la figure 2. L'interface de la classe Nœud est la suivante :

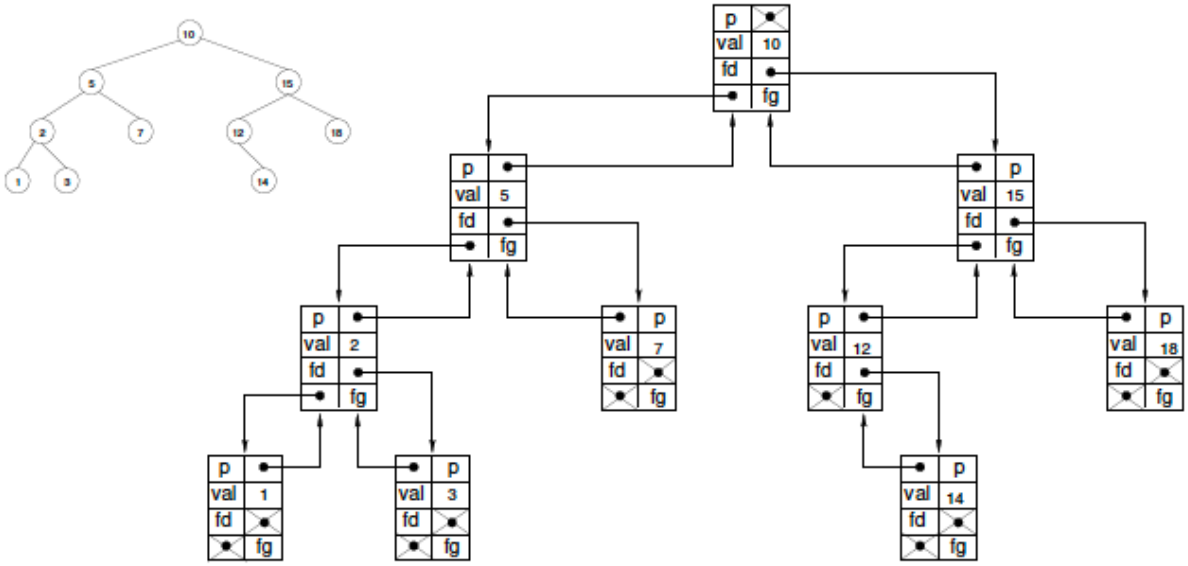

Figure 3 : Illustration de la classe Nœud sur l'arbre de la figure 2.

```
template <class T>
class Noeud {
public:
        T val;                          // La valeur associ´ee au noeud
        Noeud<T> *fg, *fd, *p;          // Pointeur sur fils gauche, fils droit et pere

        Noeud();                        // Constructeur par d´efaut
        Noeud(const T & i, Noeud<T> *leftSon = NULL, Noeud<T> *rightSon = NULL,
        Noeud<T> *father = NULL); // Constructeur principal; leve bad_alloc
        Noeud(Noeud<T> & node); // Constructeur par recopie

        template<class U> friend ostream& operator<<(ostream& o, const Noeud<U>& n);
};
template <class T> inline Noeud<T>::Noeud(): fg(NULL), fd(NULL), p(NULL) {}
```

- Définir le corps du constructeur principal. On veillera à ne pas oublier la déclaration template dans la définition de cette fonction (cette remarque est valable pour toute la suite)

- Définir le corps du constructeur par recopie. On veillera à détecter les erreurs d'allocation via l'exception bad_alloc (on se contentera de ne recopier que le nœud n, sans faire de copie récursive des fils de n. Cette dernière sera faite dans la méthode ***copieTree*** de la classe ***Arbre***).

- Définir le corps de la fonction ***operator<<*** qui devra afficher les éléments suivants :
    i. la valeur du champs val ;

ii. l'adresse en hexadécimal du ***Nœud fg*** (ou ***NULL*** sinon)
iii. l'adresse en hexadécimal du ***Nœud fd*** (ou ***NULL*** sinon)
iv. l'adresse en hexadécimal du ***Nœud p*** (ou ***NULL*** sinon)
v. l'adresse en hexadécimal du ***Nœud*** courant

Exemple :

```
Noeud<int> n1(14);
cout << n1 << endl;     // Affiche : [14 (p=NULL, fg=NULL, fd=NULL) - @ : 0x804ad38]
```

3. Gestion des arbres binaire : Classe Arbre

On propose de caractériser un arbre par les éléments suivants :

- un champ ***_nbElem*** qui stocke le nombre de nœuds dans l'arbre,
- un champ ***_racine*** qui pointe sur la racine de l'arbre,
- un champ **_courant** qui pointe sur le nœud courant

L'interface de la classe Arbre est spécifiée en Annexe.

3.1. Constructeur de la classe Arbre

i. Définir les corps des constructeurs ***C1*** et ***C2***.
ii. Définir le corps de la fonction ***copyTree*** qui assure une copie récursive de l'arbre de racine ***r***. Le second paramètre contient un pointeur sur le père de la copie du noeud ***r***. Il est utilisé pour mettre à jour le champ ***p*** des copies des fils de ***r*** lors des appels récursifs.
iii. En utilisant la fonction ***copieTree***, définir le corps du constructeur par recopie.

3.2. Destructeur de la classe Arbre et fonctions associés

i. Écrire l'implémentation de la méthode ***remove*** permettant de supprimer le sous-arbre de sommet ***r***, en mettant à jour les pointeurs du père de ***r*** (s'il existe) et/ou de l'arbre. Après appel de cette fonction, le pointeur **_courant** pointe sur le père de ***r***.
ii. Utiliser cette méthode dans le corps de la méthode ***clear*** et du destructeur de la classe ***Arbre***.

3.3. Ajout d'un nouveau nœud à la classe Arbre

i. Écrire l'implémentation de la méthode ***addLeftSon*** qui permet d'ajouter un fils gauche au nœud courant. Si un fils gauche existe déjà, sa valeur est modifiée. Dans le cas contraire, un nouveau nœud sera créé. On veillera à traiter le cas de l'arbre vide et on utilisera au besoin la fonction ***assert*** pour assurer que le pointeur ***_courant*** est valide.
ii. Dans les mêmes conditions, écrire l'implémentation de la méthode ***addRightSon*** qui permet d'ajouter un fils droit au nœud courant.

3.4. Navigation dans la classe Arbre

Spécifier le corps des méthodes ***peekLeft***, ***peekRight*** et ***peekFather*** qui permettent de renvoyer respectivement la valeur associée au fils gauche, au fils

droit et au père du pointeur courant sans modifier ce pointeur. On utilisera la fonction ***assert*** pour assurer que l'accès est valide.

3.5. Récupération d'information sur un nœud

i. Ecrire le corps de la méthode ***h*** calculant la hauteur du nœud ***r***.
ii. Spécifier le corps de la méthode ***nbLeaf*** calculant le nombre de feuilles dans le sous-arbre de sommet **r**.
iii. Spécifier le corps de la méthode ***nbNode*** calculant le nombre de nœuds dans le sous-arbre de sommet ***r***.

3.6. Parcours de l'arbre

Les arbres binaires offrent trois possibilités de parcours :

- affichage de la valeur du nœud puis récursivement de ses deux fils (on parle de parcours préfixe).
- affichage du fils gauche, de la valeur du nœud puis du fils droit (il s'agit alors du parcours infixe).
- affichage récursif des deux fils puis de la valeur du nœud. (c'est alors le parcours postfixe).

Ainsi, sur l'arbre de la figure 2, on obtiendrait pour chacun des parcours les affichages suivants :

- Parcours infixe : 1 2 3 5 7 10 12 14 15 18
- Parcours prefixe : 10 5 2 1 3 7 15 12 14 18
- Parcours postfixe : 1 3 2 7 5 14 12 18 15 10

i. Spécifier le corps de la méthode ***printPrefixe*** effectuant l'affichage des valeurs des nœuds du sous-arbre de sommet ***r*** selon un parcours préfixe.
ii. Spécifier le corps de la méthode ***printInfixe*** effectuant l'affichage des valeurs des nœuds du sous-arbre de sommet ***r*** selon un parcours infixe.
iii. Spécifier le corps de la méthode ***printPostfixe*** effectuant l'affichage des valeurs des nœuds du sous-arbre de sommet ***r*** selon un parcours postfixe.

4. ABR : les arbres binaires de recherche

Un arbre binaire de recherche est un arbre binaire tel que la valeur stockée dans chaque sommet est supérieure à celles stockées dans son sous-arbre gauche et inférieure à celles de son sous- arbre droit. L'arbre de la figure 2 est un exemple d'arbre binaire de recherche. On y trouve donc que des valeurs distinctes.

L'objectif de cette section est de définir la classe ***ABR***. Cette classe hérite de la classe ***Arbre*** et ajoute les méthodes publiques suivantes :

- ***void insert(const T & v)*** qui permet d'ajouter un nouveau nœud de valeur ***v*** dans l'arbre en respectant la définition d'un ABR.
- ***void supprime(const T & v)*** qui supprime le nœud de valeur ***v*** (s'il existe) tout en garantissant les conditions de la définition d'un ABR.
- ***Noeud<T> * find(const T & v)*** qui recherche le nœud de valeur ***v*** et renvoie un pointeur sur ce nœud (ou NULL sinon).
- ***Noeud<T> * min()*** qui renvoie le nœud de valeur minimale dans l'arbre.
- ***Noeud<T> * max()*** qui renvoie le nœud de valeur maximale dans l'arbre.

Ces méthodes utilisent leurs redéfinitions privées qui effectuent les mêmes opérations sur les sous-arbres de sommet ***r*** :

- ***void insert(Noeud<T> * r, const T & v)***
- ***void supprime(Noeud<T> * r, const T & v)***
- ***noeud<T> * find(Noeud<T> * r, const T & v)***
- ***Noeud<T> * min(Noeud<T> * r)***
- ***Noeud<T> * max(Noeud<T> * r)***

Enfin, la méthode privée ***supprime*** appelle la méthode privée ***void deleteNode(Noeud<T> * n)*** qui supprime le nœud ***n*** du sous-arbre de sommet ***n***. On définit également la méthode publique ***Noeud<T> * successor(Noeud<T> * r)*** qui renvoie un pointeur sur le successeur d'un nœud. En plus des méthodes publiques, on trouve les constructeurs et destructeurs de la classe ***ABR***, construits sur des prototypes similaires à ceux de la classe ***Arbre***.

La spécification de la classe ABR est partiellement fournie en annexe. Compléter cette spécification.

4.1. Recherche dans un ABR

i. L'élément minimal d'un arbre binaire de recherche correspond au nœud le plus à gauche. Spécifier le corps de la méthode privée ***min***.

ii. De manière similaire, L'élément maximal d'un ABR correspond au nœud le plus à droite. Spécifier le corps de la méthode privée ***max***.

iii. La recherche d'un élément ***v*** dans un ABR de sommet ***r*** est simple : si ***v*** n'est pas au sommet, on effectue la recherche dans le fils gauche ***si v < r*** et dans le fils droit sinon. On s'arrête lorsqu'on a trouvé ***v*** (et dans ce cas, on renvoie un pointeur sur le nœud trouvé.) ou si on a atteint une feuille sans trouver ***v*** (et on renvoie ***NULL***). Spécifier le corps de la méthode privée ***find***.

iv. Le successeur d'un nœud ***r*** correspond au nœud ayant une valeur directement supérieure à **r** dans l'arbre. Deux cas sont possibles :

- le nœud ***r*** admet un fils droit auquel cas le successeur de ***r*** est le plus petit élément de ce fils,
- dans le cas contraire, on peut montrer que le successeur de ***r*** est le premier ancêtre de ***r*** tel que le fils gauche soit un ancêtre de ***r***.

Donner le corps de la méthode ***successor***.

4.2. Insertion d'un nouvel élément

L'insertion d'un nouveau nœud ***v*** dans un arbre binaire de recherche de sommet ***r*** se fait au niveau des feuilles. Le principe est le suivant :

- si l'arbre est vide, on créer une racine comme on le ferait pour un Arbre classique,
- sinon, si ***v < r***, on continue dans le fils gauche, sinon dans le fils droit. On procède ainsi tant que le fils où on souhaite aller existe. On arrive ainsi soit `a une feuille, soit à un nœud qui ne possède qu'un fils.
- on créer un nouveau nœud (une feuille) dont le père sera initialisé à ***p***.

En appliquant l'algorithme décrit précédemment, spécifier le corps de la méthode privée ***insert***.

4.3. Suppression d'un élément

La suppression commence par la recherche de l'élément ***e*** à supprimer. Puis :
- si c'est une feuille, la suppression s'effectue sans problèmes ;
- si c'est un nœud qui n'a qu'un fils, on le remplace par ce fils ;
- si c'est un nœud qui a 2 fils, on peut :
    - soit le remplacer par son prédécesseur,
    - soit le remplacer par son successeur.

Le deuxième cas est illustré dans la figure 4 pour la suppression du nœud 10. L'idée consiste donc à remplacer le nœud par son successeur et à supprimer ce successeur. Plus précisément, on procède comme suit :
- on recherche le nœud à détacher s (sur l'exemple de la figure 4, il s'agit du nœud 12). On remarquera que le successeur admet au plus un fils puisqu'on se trouve dans le cas où le fils droit du nœud à supprimer existe.
- on détache le nœud s (en mettant `a jour les champs des nœuds environnants).
- on remplace e par s.

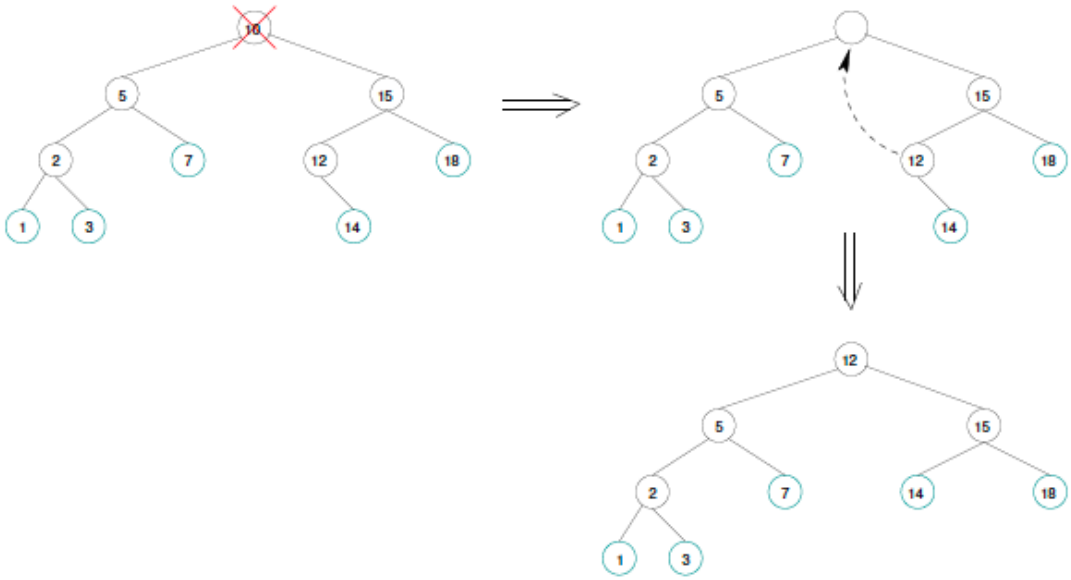

Figure 4 : Suppression du nœud 10 dans un ABR.

En appliquant l'algorithme décrit précédemment, spécifier le corps des méthodes privées ***supprime*** et ***deleteNode***.

Annexes

A Interface de la classe Arbre

```
template <class T>
class Arbre {
protected :
Noeud<T>* _racine; // Pointeur sur la racine
Noeud<T>* _courant; // Pointeur sur l'élément courant
int _nbElem; // Nombre de noeuds
// Copie r´ecursive d'un arbre de racine r;
Noeud<T>* copyTree(Noeud<T> * r, Noeud<T> * father);
public:
/*** Constructeurs/Desctructeur ***/
Arbre() : _racine(NULL), _courant(NULL), _nbElem(0) {}
Arbre(Noeud<T>* r); // Constructeur C1: creer l'arbre de racine r
Arbre(const T & v); // Constructeur C2: creer l'arbre de racine `a valeur v
Arbre(const Arbre<T>& a); // Constructeur par recopie
~Arbre();
/*** Modification de l'arbre ***/
//Permet d'ajouter un élément par rapport à l'élément courant
void addLeftSon(const T & valg); // leve l'exception bad_alloc en cas de pb
void addRightSon(const T & vald); // idem
void remove(Noeud<T>* r); // Suppression du noeud r et du sous-arbre de sommet r
void clear(); // Supprime tous les noeuds de l'arbre; maj des pointeurs
/*** Accesseurs ***/
Noeud<T>* rootNode() const { return _racine; }
int size() const { return _nbElem; }
Noeud<T>* currentNode() const { return _courant; }
Noeud<T>* leftSon() const { if (_courant != NULL) return _courant->fg; }
Noeud<T>* rightSon() const { if (_courant != NULL) return _courant->fd; }
Noeud<T>* father() const { if (_courant != NULL) return _courant->p; }
// Renvoit la valeur du noeud courant
T value() const { assert(_courant != NULL); return _courant->val; }
T & value() { assert(_courant != NULL); return _courant->val; }
// Renvoie les valeurs des fils et du pere sans changer le pointeur _courant
T peekLeft() const;
T peekRight() const;
T peekFather() const;
/*** Navigation dans l'arbre - Deplacement du pointeur _courant ***/
void left() { if (_courant != NULL) _courant=_courant->fg; }
void right() { if (_courant != NULL) _courant=_courant->fd; }
void father(){ if (_courant != NULL) _courant=_courant->p; }
void reset() { _courant=_racine; } //retour `a la racine
void setCurrent(Noeud<T>* r) { _courant = r; } // déplace _courant en r
/*** Accès au fils gauche/droit et au pere du noeud r, NULL sinon ***/
Noeud<T>* leftSon(Noeud<T>* r) { return ((r != NULL)?(r->fg):NULL); }
Noeud<T>* rightSon(Noeud<T>* r) { return ((r != NULL)?(r->fd):NULL); }
Noeud<T>* father(Noeud<T>* r) { return ((r != NULL)?(r->p):NULL); }

/*** Récupérations d'infos sur l'arbre ***/
int h() const { return h(_racine); } // hauteur de l'arbre
int nbLeaf() const { return nbLeaf(_racine); } //nombre de feuilles de l'arbre
bool isEmpty() const { return isEmpty(_racine); }// l'arbre est il vide?
// Mêmes fonctions mais pour l'arbre de racine r
int h(Noeud<T>* r) const;
int nbLeaf(Noeud<T>* r) const;
bool isEmpty(Noeud<T>* r) const { return (r == NULL); }
// Accès au caractéristiques d'un noeud
bool isLeaf(Noeud<T>* r) const // r est-t-il une feuille?
{ return (r!= NULL)&&(r->fg==NULL)&& (r->fd==NULL); }
unsigned long long nbNode(Noeud<T>* r); // nombre de noeud dans le sous-arbre
// Affichage des informations contenant l'arbre
void printInfo();
/*** Affichage selon diff. parcours - suppose que T supporte l'opérateur << ***/
```

```
void printInfixe() { printInfixe(_racine); } // Parcours infixe
void printPrefixe() { printPrefixe(_racine); } // Parcours prefixe
void printPostfixe(){ printPostfixe(_racine); } // Parcours postfixe
// idem, mais a partir d'un noeud r
void printInfixe(Noeud<T>* r); // Parcours infixe
void printPrefixe(Noeud<T>* r); // Parcours prefixe
void printPostfixe(Noeud<T>* r); // Parcours postfixe

};
```

B Interface de la classe ABR

```
template <class T>
class ABR : public Arbre<T> {
private:
void insert(Noeud<T> * r, const T &v);// insert v dans le sous-arbre de sommet r
void supprime(Noeud<T> * r, const T &v); // supprime v "
void deleteNode(Noeud<T> * n); // supprime le noeud n dans l'arbre de
Noeud<T> * find(Noeud<T> * r, const T &v);// cherche v "
Noeud<T> * min(Noeud<T> * r); // recherche du noeud min dans le S-A de sommet r
Noeud<T> * max(Noeud<T> * r); // recherche du noeud max dans le S-A de sommet r
public:
/*** Constructeur/Destructeur ***/
ABR() : {}
ABR(Noeud<T>* r) : {}
ABR(const T & v) : {}
ABR(const ABR<T> & a) : {}
~ABR() {}
/*** Méthodes d'ajout/retrait/recherche ***/
void insert(const T & v) { ########## } // insert v
void supprime(const T & v) { ########## } // supprime v
Noeud<T> * find(const T & v) { ########## } // recherche v
Noeud<T> * min() { ########## } // renvoie le noeud min
Noeud<T> * max() { ########## } // renvoie le noeud min
Noeud<T> * successor(Noeud<T> * r); //renvoie le successeur de r

};
```