Université du Luxembourg 2004–2005

D.U.T. Informatique 2ème année mardi 27 avril

Programmation avancée en langage C++
Examen n° 2

Sebastien.Varrette@imag.fr (Bureau BR.4.07)

Durée: 3 heures

Calculatrice interdite - Documents interdits

Il sera tenu compte dans la notation de la clarté des explications et du soin apporté à la rédaction (propreté, organisation, respect de la langue française). Le barème est fournit à titre indicatif.

1 Questions de cours (Une fois n'est pas coutume...) (8 pts)

Répondre aux questions suivantes de manière succinte :

1. (2 pts) On définit principalement des classes en C++ de 2 façons différentes. Lesquelles ? Quelle(s) différence(s) existe(nt) entre ces deux constructions, en particulier au niveau de la visibilité par défaut des membres ?
2. (1 pt) Pourquoi est il déconseillé de passer par valeur à une fonction un paramètre dont le type est une classe (ou une structure) ?
3. (1 pt) Dans une classe, comment fait-on pour initialiser un attribut dont le type est une classe (par opposition à un type fondamentatal)
4. (1 pt) Dans quel cadre utilise-t-on le mot-clé *virtual* ?
5. (1 pt) On souhaite écrire une fonction qui calcule le carré de deux entiers, de deux réels, de deux fractions, de deux matrices etc... Comment fait-on de manière simple ? Donner le prototype d'une telle fonction.
6. (2 pts)Qu'est ce que la *STL* ?

2 Le grand classique....(3 pts)

Quel sera l'affichage obtenu après l'exécution du programme suivant ?

```
#include <iostream.h>
/*** Définition de la classe A ***/
class A {
    char _a;
  public:
    A(char a) : _a(a) {}
    virtual void print() const;
};
void A::print() const { cout << _a; }

/*** Définition de la classe B ***/
class B: public A {
    char _b;
```

```
 public:
    B(char ch1, char ch2);
    void print() const;
};
B::B(char ch1, char ch2) : A(ch1), _b(ch2) {}
void B::print() const {
    cout << "B : ";
    A::print();
    cout << _b;
}
/*****************************/
void print(const A & objA) {
    cout << "Global print : ";
    objA.print();
    cout << endl;
}

int main() {
    A objA('x');
    B objB('y','z');
    print(objA);
    print(objB);
    return 0;
}
```

3 Un arbre peut en cacher un autre...

L'objectif de cet exercice est d'implémenter un modèle de classe `Arbre` permettant de regrouper une collections 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.

Remarque : la structure d'arbre est l'une des plus importantes et des plus spécifiques de l'informatique. Elle est utilisée par exemple dans l'organisation des fichiers dans un OS.

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 infixé, préfixé, postfixé.
- vidage d'un arbre.

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

3.1 Définitions

Un arbre est constitué d'un ensemble de *noeuds* auquels sont associées 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.

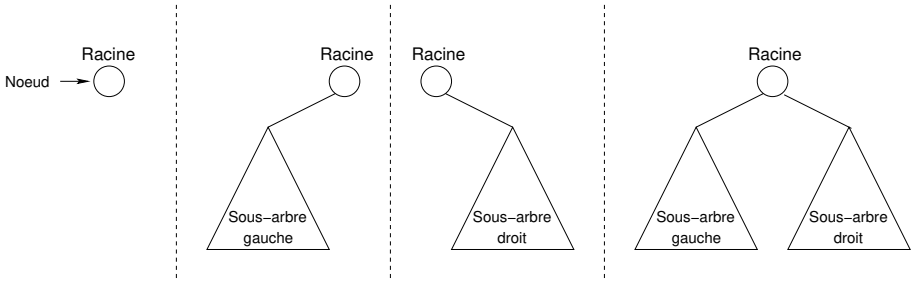

FIGURE 1 – Définition récursive d'un arbre

Soit A un arbre. $A = \emptyset$ désignera un arbre vide et $|A|$ le nombre de noeuds de l'arbre [1]. Soit x un noeud. 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 noeud 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 père et c'est le seul sommet dans ce cas. La figure 2 illustre ces définitions sur un arbre binaire hébergeant des noeuds de type `int` (`T=int`).

En particulier sur cet exemple :
— le père du noeud 2 est le noeud 5 ;
— le frère du noeud 7 est le noeud 2 ; le noeud 14 n'a pas de frère ;
— le noeud 12 n'a qu'un fils : le noeud 14.

1. En particulier, si $A = \emptyset$, alors |A|=0.

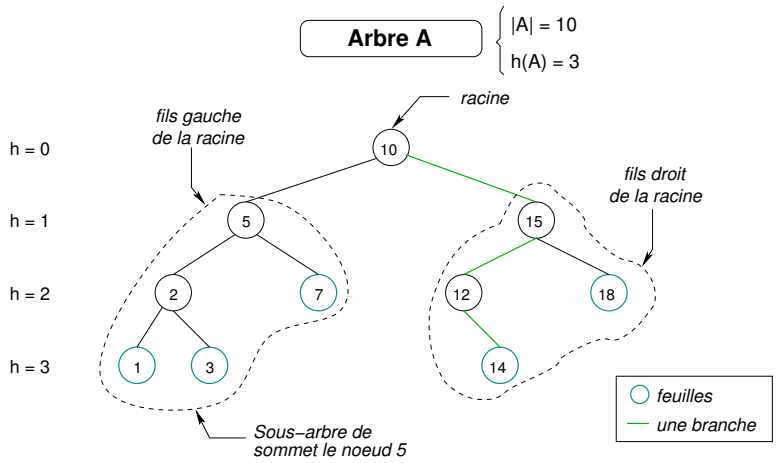

FIGURE 2 – Illustration des définitions sur un arbre binaire d'entiers

3.2 Quelques conseils avant de poursuivre...

— Prenez le temps de bien comprendre la façon dont sont implémentées chaques classes et les liens qui unissent chaques membres (faites des dessins).
— Les arbres ont naturellement une structure récursive. Souvenez-vous en !

3.3 Classe `Noeud` (2,5 pts)

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

— un champ `val` contenant une valeur (de type `T`) ;
— un champ `fg` correspondant à un pointeur sur un noeud (le fils gauche)
— un champ `fd` correspondant à un pointeur sur un noeud (le fils droit)
— un champ `p` correspondant à un pointeur sur un noeud (le père).

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

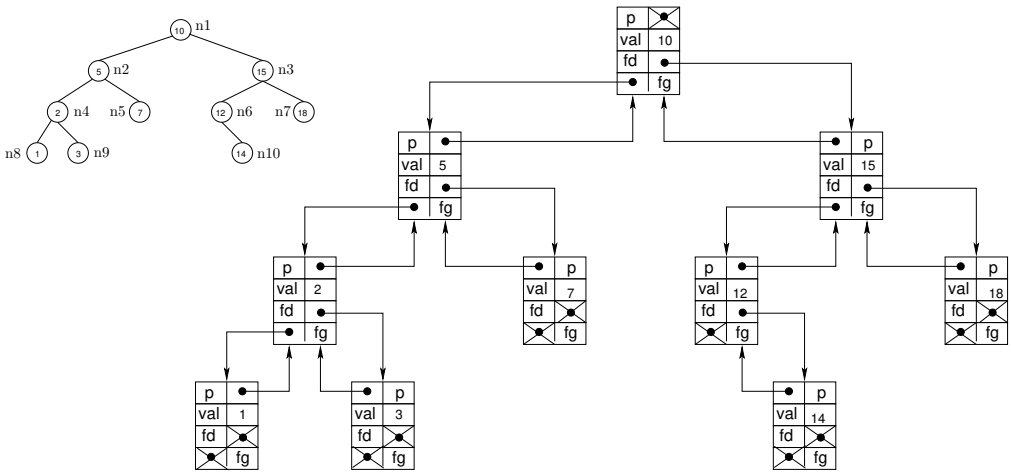

FIGURE 3 – Illustration de la classe `Noeud` sur l'arbre de la figure 2

```
template <class T>
class Noeud {
public:
    /*** Une fois n'est pas coutume: les attributs sont publics ***/
    T val;                   // La valeur associée au noeud
    Noeud<T> *fg, *fd, *p;   // Pointeur sur fils gauche, fils droit et pere

    /*** Constructeur / Destructeur ***/
    Noeud();// Constructeur par défaut
    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
    // Pas de destructeur ici, il sera géré dans la classe Arbre!
    /*** Operateur ***/
    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) {}
```

1. (0,5 pt) Définir le corps du constructeur principal. On veillera a ne pas oublier la déclaration `template` dans la définition de cette fonction (cette remarque est valable pour toute la suite)
2. (1 pt) Définir le corps du constructeur par recopie. On veillera a détecter les erreurs d'allocation via l'exception `bad_alloc` (On utilisera un bloc `try...catch`).

   Remarque : on se contentera de ne recopier que le noeud n, sans faire de copie récursive des fils de n. Cette dernière sera faite dans la méthode `copieTree` de la classe `Arbre`
3. (1 pt) Définir le corps de la fonction `operator<<` qui devra afficher les éléments suivants :
   — la valeur du champs `val` ;
   — l'adresse en hexadécimal du Noeud `fg` (ou `NULL` sinon)
   — l'adresse en hexadécimal du Noeud `fd` (ou `NULL` sinon)
   — l'adresse en hexadécimal du Noeud `p` (ou `NULL` sinon)
   — l'adresse en hexadécimal du Noeud courant

   Exemple :

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

3.4 Gestion des arbres binaires : la classe `Arbre` (18 pts)

On propose de caractériser un arbre par les éléments suivants :
— un champ `_nbElem` qui stocke le nombre de noeuds dans l'arbre ;
— un champ `_racine` qui pointe sur la racine de l'arbre ;
— un champ `_courant` qui pointe sur le noeud courant (ce qui permettra de simplifier l'ajout et la consultation)

Cette représentation est illustrée dans la figure 4.
L'interface de la classe `Arbre` est spécifiée en annexe A.

3.4.1 Constructeurs de la classe `Arbre` (3,5 pts)

1. (0,5 pt) Définir le corps du constructeur C1

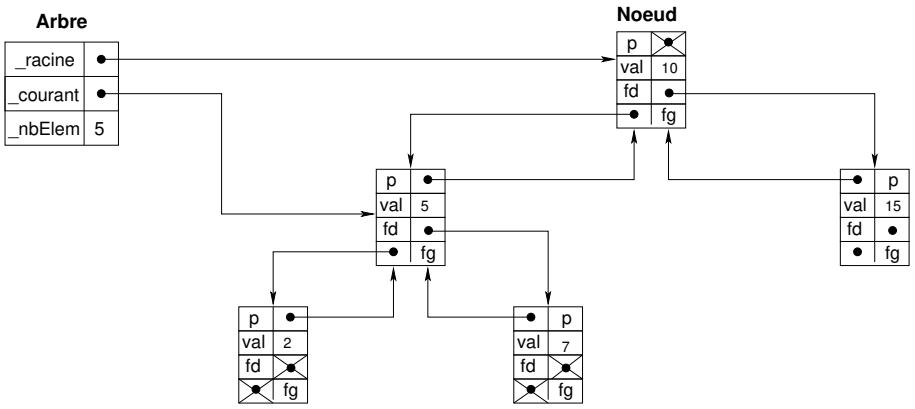

FIGURE 4 – Illustration du chaînage de la classe `Arbre` sur un exemple

2. (0,5 pt) Définir le corps du constructeur C2
3. (2 pts) La fonction `copyTree` 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 (initialement, lors de la copie de l'arbre complet de racine `_racine`, ce paramètre vaut donc `NULL`, le père d'une racine valant, par définition, `NULL`). Il est utilisé pour mettre a jour le champ `p` des copies des fils de r lors des appels récursifs.

   Définir le corps de la fonction `copyTree`.
4. (0,5 pt) En utilisant la fonction `copieTree`, définir le corps du constructeur par recopie.

3.4.2 Destructeur de la classe `Arbre` et fonctions associées (3 pts)

1. (2 pts) La fonction (récursive) `remove` permet de supprimer "proprement" le sous-arbre de sommet r, en mettant éventuellement 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.

   Ecrire l'implémentation de la méthode `remove`.
2. (1 pt) utiliser cette méthode dans le corps de la méthode `clear` et du destructeur de la classe `Arbre`.

3.4.3 Ajout de nouveaux noeuds dans la classe `Arbre` (4 pts)

1. (2 pts) Ecrire l'implémentation de la méthode `addLeftSon` qui permet d'ajouter un fils gauche au noeud courant. Si un fils gauche existe déjà, sa valeur est modifiée. Dans le cas contraire, un nouveau noeud 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.
2. (2 pts) Dans les mêmes conditions, écrire l'implémentation de la méthode `addRightSon` qui permet d'ajouter un fils droit au noeud courant.

3.4.4 Navigation dans l'arbre sans deplacement du pointeur courant (1,5 pts)

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 légitime (i.e que le noeud correspondant existe et que le pointeur courant est valide).

3.4.5 Récupérations d'informations sur un noeud r de l'arbre (3 pts)

1. Ecrire le corps de la méthode `h` calculant la hauteur du noeud r.
2. Spécifier le corps de la méthode `nbLeaf` calculant le nombre de feuilles dans le sous-arbre de sommet r.
3. Spécifier le corps de la méthode `nbNode` calculant le nombre de noeuds dans le sous-arbre de sommet r.

3.4.6 Parcours de l'arbre (3 pts)

Un parcours a pour but de passer en revue (pour un traitement quelconque, ici l'affichage) chaque sommet une et une seule fois. Ce parcours se fait de façon récursive.
Les arbres binaires offrent trois possibilités de parcours :

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

Ainsi, sur l'arbre de la figure 2, on otiendrait 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

1. (1 pt) Spécifier le corps de la méthode `printPrefixe` effectuant l'affichage des valeurs des noeuds du sous-arbre de sommet r selon un parcours préfixe.
2. (1 pt) Spécifier le corps de la méthode `printInfixe` effectuant l'affichage des valeurs des noeuds du sous-arbre de sommet r selon un parcours infixe.
3. (1 pt) Spécifier le corps de la méthode `printPostfixe` effectuant l'affichage des valeurs des noeuds du sous-arbre de sommet r selon un parcours postfixe.

3.5 Les arbres binaires de recherche : la classe `ABR` (12,25 pts)

Pour l'instant, la recherche d'un élément dans un arbre de n noeuds nécessite de parcourir l'ensemble des noeuds de cet arbre (selon l'un des parcours évoqués au §3.4.6). Cette recherche requiert donc $\mathcal{O}(n)$ comparaisons.
En supposant que les éléments du type `T` peuvent être comparés deux à deux (donc que les opérateurs `<`, `==` et `>` existent pour le type `T`), il est possible de construire l'arbre de façon à ce que les recherches dans cet arbre soient quasi-linéaires et ne requièrent que $\mathcal{O}(\log n)$ comparaisons. Un tel arbre est alors appelé *arbre binaire de recherche*.

Définition 1 (Arbre Binaire de Recherche) *Un arbre binaire de recherche est un arbre binaire tel que la valeur stockée dans chaque sommet est superieure à celles stockées dans son sous-arbre gauche et inferieure 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 noeud de valeur v dans l'arbre en respectant les conditions de la définition 1
- `void supprime(const T & v)` qui supprime le noeud de valeur v (s'il existe) tout en garantissant les conditions de la définition 1
- `Noeud<T> * find(const T & v)` qui recherche le noeud de valeur v et renvoit un pointeur sur ce noeud (ou `NULL` sinon).
- `Noeud<T> * min()` qui renvoit le noeud de valeur minimale dans l'arbre
- `Noeud<T> * max()` qui renvoit le noeud 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 noeud n du sous-arbre de sommet n. On définit également la méthode publique
`Noeud<T> * successor(Noeud<T> * r)` qui renvoit un pointeur sur le successeur d'un noeud (voir question 4 du §3.5.1).
En plus des méthodes publiques, on trouve bien entendu les constructeurs et destructeurs de la classe `ABR`, construits sur des prototypes similaires à ceux de la classe `Arbre`.

(2,25 pts) La spécification de la classe ABR est partiellement fournie en annexe B. Compléter cette spécification en remplaçant les zones marquées par la séquence `#########`

3.5.1 Recherches dans un arbre binaire de recherche (4,5 pts)

1. (1 pt) L'élément minimal d'un arbre binaire de recherche correspond au noeud le plus à gauche. Spécifier le corps de la méthode privée `min`.
2. (1 pt) De manière similaire, L'élément maximal d'un arbre bianire de recherche correspond au noeud le plus à droite. Spécifier le corps de la méthode privée `max`.
3. (1 pt) La recherche d'un élément v dans un arbre binaire de recherche de sommet r est très 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 renvoit un pointeur sur le noeud trouvé.) ou si on a atteind une feuille sans trouver v (et on renvoit alors `NULL`). En s'aidant de cette description, spécifier le corps de la méthode privée `find`.
4. (1,5 pts) Le successeur d'un noeud r correspond au noeud ayant une valeur directement supérieure à r dans l'arbre. Deux cas sont possibles :
   — le noeud 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.

   Ainsi, sur l'exemple de la figure 2, le successeur du noeud 7 correspond au noeud 10. A partir de cette description, donner le corps de la méthode `successor`.

3.5.2 Insertion d'un nouvel élément (2 pts)

L'insertion d'un **nouveau** noeud v dans un arbre binaire de recherche de sommet r se fait au niveau des feuilles. Le principe, illustré dans la figure 5, est le suivant :

1. si l'arbre est vide, on créer une racine comme on le ferait pour un `Arbre` classique
2. sinon, si $v < r$, on continue dans le fils gauche, sinon dans le fils droit ; On procède ainsi tant que le fils ou on souhaite aller existe. On arrive ainsi soit à une feuille, soit à un noeud qui ne possède qu'un fils. Soit p ce noeud (p=12 dans la figure 5).
3. on créer un nouveau noeud (une feuille) dont le pere sera initialisé à p.

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

3.5.3 Suppression d'un élément (3,5 pts)

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 noeud qui n'a qu'un fils, on le remplace par ce fils ;
— si c'est un noeud qui a 2 fils, on peut
  — soit le remplacer par son prédecesseur ;

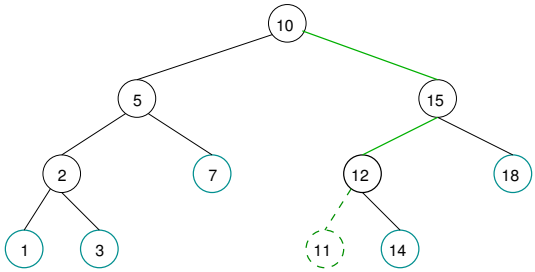

FIGURE 5 – Illustration de l'insertion de 11 dans un arbre binaire de recherche

— soit le remplacer par son successeur.

Le deuxième cas est illustré dans la figure 6 pour la suppression du noeud 10. L'idée consiste donc à remplacer le noeud par son successeur et à supprimer ce successeur. Plus précisément, on procède comme suit :

— on recherche le noeud à détacher s (sur l'exemple de la figure 6, il s'agit du noeud 12). On remarquera que le successeur admet au plus un fils puisqu'on se trouve dans le cas ou le fils droit du noeud à supprimer existe.
— on détache le noeud s (en mettant à jour les champs des noeud environnants)
— on remplace e par s.

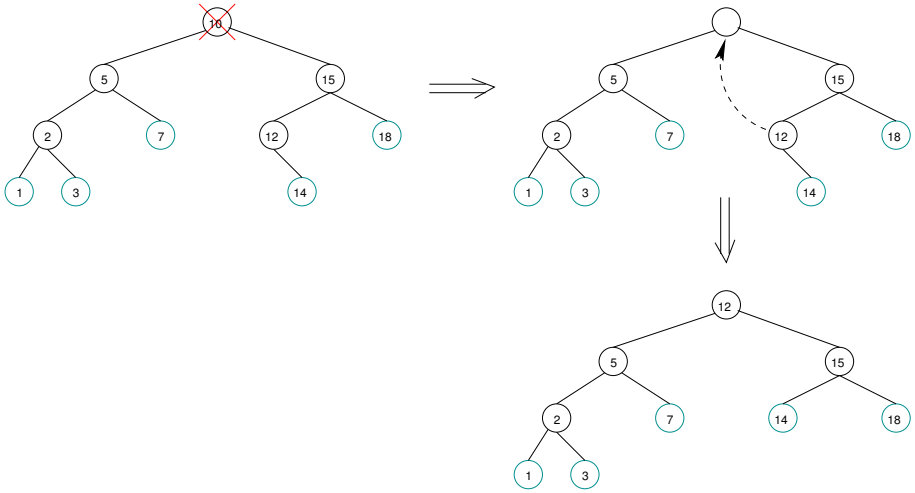

FIGURE 6 – Suppression de 10 dans un arbre binaire de recherche

(0,5 + 3 pts) En appliquant l'algorithme décrit précédemment, spécifier le corps des méthodes privées `supprime` et `deleteNode` .

FIN bravo !

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 récursive 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 à 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; }
    // Renvoit 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 à 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 ds 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() { ############ } // renvoit le noeud min
    Noeud<T> * max() { ############ } // renvoit le noeud min
    Noeud<T> * successor(Noeud<T> * r); //renvoit le successeur de r
};
```