Algorithmique et Programmation C++ 2 TD ❷

1. Structure de base d'un Arbre

```
#include <iostream.h>


class Noeud
{
    friend class Arbre;
    int info;
    Noeud * fg, * fd;
public:
    Noeud();
    Noeud(const int &, Noeud *, Noeud *);
    ~Noeud();
    void affprof(const int &);
};

class Arbre
{
    Noeud * racine;
public:
    Arbre();
    Arbre(Noeud *);
    ~Arbre();
    void suppnode(Noeud *);
    void affprof();
    int hauteur();
    int maximum();
    int maximumord();
    bool operator==(const Arbre &)const;
    Arbre miroir();
    Noeud * miroir(Noeud *);
    bool egalitemiroir(Arbre &);
    bool sousarbre(Arbre &);
    bool sousarbre(Noeud *, Noeud *);
    bool estinclus(const Arbre &);
};
```

Définition des constructeurs

```
Noeud::Noeud()
{
    // Valeurs par defaut pour les membres donnees :
    fg=fd=NULL;
    info=0;
}

Noeud::Noeud(int i, Noeud * filsg=NULL, Noeud * filsd=NULL)
{
    // Constructeur plus avance permettant de configurer
    // chaque membre donnee :
    info=i;
    fg=filsg;
    fd=filsd;
}

Arbre::Arbre()
{
    // Constructeur par defaut : arbre vide :
    racine=NULL;
}

Arbre::Arbre(Noeud * n)
{
    // Construit un arbre a partir d'un Noeud :
    racine=n;
}
```

Destructeurs avec travail partagé

```
Arbre::~Arbre()
{
    if(racine) delete racine;
}

Noeud::~Noeud()
{
    if(fg) delete fg;
    if(fd) delete fd;
}
```

Destructeurs avec travail par l'Arbre

```
Arbre::~Arbre()
{
    if(racine) suppnode(racine);
}

void Arbre::suppnode(Noeud * n)
{
    if(n->fg) suppnode(n->fg);
    if(n->fd) suppnode(n->fd);
    delete n;
}

Noeud::~Noeud()
{}
```

Destructeur optimisé

```
Arbre::~Arbre()
{
    if(racine)
    {
        // Construit un sous-arbre a partir de chacun des deux fils,
        // ainsi, a la sortie de la fonction, ces deux arbres seront
        // detruits automatiquement puisqu'ils sont ici crees en
        // statique, ce qui appelera le destructeur d'arbre,
        // c'est-a-dire reappellera cette fonction, qui recursivement
        // detruira chaque Noeud de cette facon. Le cas de base etant
        // celui ou racine est NULL.
        Arbre fg(racine->fg);
        Arbre fd(racine->fd);
        // Puis detruit le premier Noeud de l'arbre.
        delete racine;
    }
}

Noeud::~Noeud()
{}
```

2. Traitements sur les Arbres

1°) Profondeur d'un Nœud

```
void Arbre::affprof()
{
    // Par convention, le Noeud racine a une profondeur de 1 :
    if(racine) racine->affprof(1);
}

void Noeud::affprof(const int & actuel)
{
    // Affiche simplement la profondeur actuelle puis celle de ses
    // deux sous fils :
    cout << actuel << ';';
    if(fg) fg->affprof(actuel+1);
    if(fd) fd->affprof(actuel+1);
}
```

2°) Hauteur d'un Arbre

```
int max(const int & a, const int & b)
{
    // Fonction generique renvoyant le max de deux entiers :
    if(a>b) return a; else return b;
}

int Arbre::hauteur()
{
    // Cas d'un arbre (ou d'un sous-arbre) vide :
    if(racine==NULL) return 0;
    Arbre tmp;
    // Cree deux sous-arbres a partir des deux branches puis calcule
    // leur hauteur par appel recursif :
    tmp.racine=racine->fg;
    int hg=tmp.hauteur();
    tmp.racine=racine->fd;
    int hd=tmp.hauteur();
    // Reinitialise la racine de tmp pour ne pas que les autres Noeud
    // (qui appartiennent a l'Arbre this) ne soient detruits :
    tmp.racine=NULL;
    // Renvoie la hauteur maximale trouvee, plus 1 pour l'etage
    // supplementaire ou l'on est :
    return max(hg,hd)+1;
}
```

3°) Maximum

```
int Arbre::maximum()
{
    // Cas ou l'arbre est vide : au sens strict du terme, il faudrait
    // renvoyer l'equivalent de moins l'infini, ici on prend -1 en
    // supposant que notre arbre ne contient que des valeurs
    // positives :
    if(racine==NULL) return -1;
    if(racine && racine->fg==NULL && racine->fd==NULL)
        return racine->info;
    // On ne travaille que sur des arbres, jamais sur les noeuds :
    Arbre tmp;
    // Recherche le maximum dans chacune des deux branches de l'arbre
    // (meme technique que precedemment) :
    tmp.racine=racine->fg;
    int maxg=tmp.maximum();
    tmp.racine=racine->fd;
    int maxd=tmp.maximum();
    // Pour ne pas detruire tout l'arbre a la fin de la fonction :
    tmp.racine=NULL;
    // Renvoie le maximum de tout ca :
    return max(racine->info,max(maxd,maxg));
}
```

4°) Maximum d'un Arbre ordonné

```
int Arbre::maximumord()
{
    // Si l'arbre est ordonne, alors il est clair que la valeur
    // maximale est celle qui se situe le plus a droite :
    if(racine==NULL) return -1;
    Noeud * tmp=racine;
    while(tmp->fd!=NULL) tmp=tmp->fd;
    return tmp->info;
}
```

5°) Egalité de deux Arbres

```
bool Arbre::operator==(const Arbre & a)const
{
    // Traitements de quelques cas particuliers :
    if(racine && !a.racine || !racine && a.racine)
        return false;
    if(racine==NULL) return true;
    // Verifions tout de suite si les infos des deux arbres est
    // la meme, sinon, c'est termine :
    if(racine->info != a.racine->info) return false;

    // Bien, nos deux arbres ne sont pas vides et ont l'air egaux :
    Arbre tmp1,tmp2;
    bool res;
    // Compare tout d'abord les branches gauches deux a deux, pour
    // cela, au lieu de faire appel a une fonction membre de Noeud,
    // on rappelle l'operateur == de Arbre en creant deux sous-arbres
    // a partir des deux branches gauches a comparer :
    tmp1.racine=racine->fg;
    tmp2.racine=a.racine->fg;
    if( !(tmp1==tmp2) ) res=false;
    else
    {
        // Premier test reussi avec succes, comparons maintenant
        // les branches droites :
        tmp1.racine=racine->fd;
        tmp2.racine=a.racine->fd;
        res=tmp1==tmp2;
    }
    // Comme toujours, pour ne pas detuire le contenu de this et a en
    // detruisant tmp1 et tmp2 :
    tmp1.racine=NULL;
    tmp2.racine=NULL;
    // Puis retour de la conclusion :
    return res;
}
```

6°) Fabrication de l'arbre miroir

```
Arbre Arbre::miroir()
{
    // Ici on utilise le constructeur d'Arbre a partir
    // d'un Noeud deja existant :
    return Arbre(miroir(racine));
}

Noeud * Arbre::miroir(Noeud * n)
{
    // Cette fonction fournit le miroir du noeud n :
    if(n=NULL) return NULL;
    Noeud * res=new Noeud();
    res->info=n->info;
    res->fg=miroir(n->fd);
    res->fd=miroir(n->fg);
    return res;
}
```

7°) Es-tu un arbre miroir ?

```
bool Arbre::egalitemiroir(Arbre & a)
{
    // Le code est similaire a l'operateur ==, sauf qu'au lieu de
    // comparer les branches gauches et droites entre elles, on
    // les inverse, c'est tout.
    if(racine==NULL && a.racine) return true;
    if((racine && !a.racine) || (!racine && a.racine)) return false;
    if(racine->info != a.racine->info) return false;
    Arbre tmp1(racine->fd);
    Arbre tmp2(a.racine->fg);
    bool res=tmp1.egalitemiroir(tmp2);
    if(res)
    {
        tmp1=Arbre(racine->fg);
        tmp2=Arbre(a.racine->fd);
        res=tmp1.egalitemiroir(tmp2);
    }
    tmp1.racine=tmp2.racine=NULL;
    return res;
}
```

8°) Sous-arbre

```
bool Arbre::sousarbre(Arbre & a)
{
    // Appelle une autre fonction membre qui agira sur les Noeud :
    return sousarbre(racine,a.racine);
}

bool Arbre::sousarbre(Noeud * a, Noeud * b)
{
    // Quelques cas particuliers :
    if(b==NULL) return true;
    if(a==NULL) return false;
    // Puis cas general en faisant appel a une fonction egalite(.,.)
    // prenant en argument deux Noeud * (le code ne figure pas ici) :
    if(a->info==b->info)
        return Arbre::egalite(a,b) || sousarbre(a->fg,b) ||
sousarbre(a->fd,b);
    else
        return sousarbre(a->fg,b) || sousarbre(a->fd,b);
}

bool Arbre::estinclus(const Arbre & a)
{
    // Ceci est une deuxieme methode qui utilise l'egalite
    // deja definie plus haut :
    bool res=(*this==a);
    // Se sert de l'arbre this comme tampon en modifiant sa racine :
    Noeud * tmp=racine;
    // Teste si l'arbre a n'est pas inclus dans la branche gauche :
    if(res==false)
    {
        if(tmp->fg) racine=tmp->fg;
        res=estinclus(a);
    }
    // Puis si l'arbre a n'est pas inclus dans la branche droite :
    if(res==false)
    {
        if(tmp->fd) racine=tmp->fd;
        res=estinclus(a);
    }
    // Remet les choses en l'etat et renvoie le resultat :
    racine=tmp;
    return res;
}
```