Fiche n°19 : Structures autoréférentes en C++

Le langage C++ permet la réalisation de structures de données autoréférentes, telles que des listes chaînées pu des arbres, de manière plus lisible qu'en langage C. Nous allons illustrer certaines des possibilités offertes par un exemple d'implantation d'arbres binaires

Un arbre binaire est soit vide, soit non vide. Un arbre non vide comporte un fils gauche et un fils droit, qui sont des arbres (éventuellement vides). Dans notre exemple, une chaîne de caractères est associée à la racine de tout arbre non vide. On maintiendra l'arbre trié dans le sens où toutes les chaînes contenues dans le fils gauche d'un arbre sont situées, au regard de l'ordre lexicographique, avant la chaîne associée à sa racine. Les variables d'instances privée de la classe **ArbreBin** sont les suivantes.

```
class ArbreBin
{
    private:
    bool vide;
    string st;
    ArbreBin* fg;
    ArbreBin* fd;
    …
};
```

Indique si l'arbre est vide.

Chaîne rattachée à la racine.

Pointeurs sur les fils gauche et droits.

En C++, les Booléens peuvent toujours être représentés par des entiers avec la même convention qu'en C, mais il existe un type **bool** pouvant prendre les valeurs **true** et **false** représentant respectivement les entiers 1 et 0.

Pour l'instant, nous allons munir notre classe **ArbreBin** d'un constructeur qui crée un arbre vide, d'une méthode pour ajouter une chaîne dans un arbre et d'une méthode pour afficher les chaînes contenues dans un arbre en ordre infixe, qui correspond ici à l'ordre lexicographique.

```
ArbreBin::ArbreBin(){vide=true;}
```

Ce constructeur crée un arbre vide. Inutile d'assigner les autres variables d'instance.

```
void ArbreBin::ajoute(string s)
{
    if(vide)
    {
        vide=false;
        st=s;
        fd=new ArbreBin();
        fg=new ArbreBin();
    }
    else
    {
        if(s<=st) fg->ajoute(s);
        else fd->ajoute(s);
    }
}
```

Ajout d'une chaîne dans l'arbre.

Si l'arbre est initialement vide, il est transformé en arbre non vide ayant deux fils vides. La chaîne est placée à la racine, dans la variable d'instance **st**.

Si l'arbre est non vide, la chaîne est comparée à celle rattachée à la racine et ajoutée au sous arbre gauche ou droit selon le résultat de cette comparaison.

Fiche n°20 : Structures autoréférentes en C++

```
void ArbreBin::printInfixe()
{
    if(!vide)
    {
        fg->printInfixe();
        cout << st << " ";
        fd->printInfixe();
    }
}
```

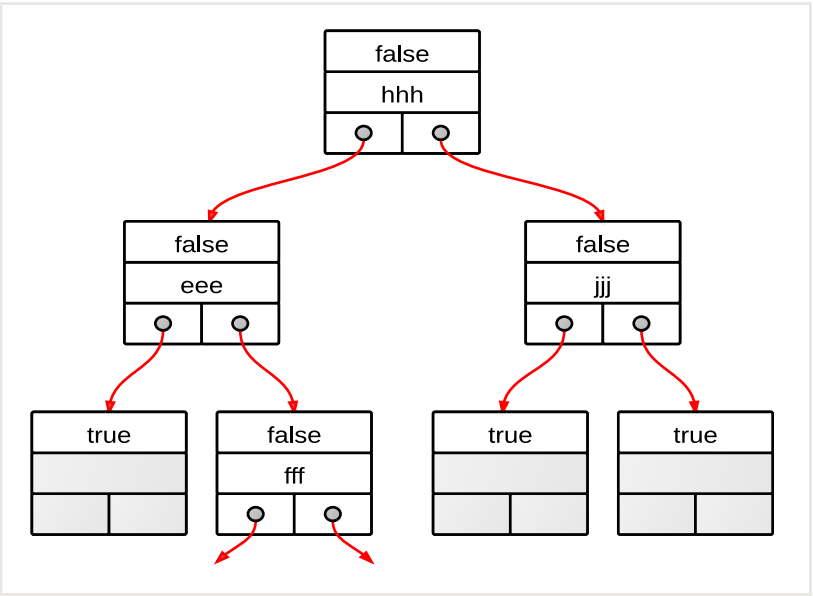
Pour afficher le contenu d'un arbre non vide, on affiche (par un appel récursif) le contenu du fils gauche, puis la chaîne située à la racine, puis (par un appel récursif) le contenu du fils droit.

A ce stade, nous disposons des méthodes permettant de construire un arbre de l'afficher.

Affiche **eee fff hhh jjj** à l'écran.

```
ArbreBin t;
t.ajoute("hhh"); t.ajoute("eee");
t.ajoute("jjj"); t.ajoute("fff");
t.printInfixe();
```

La représentation en mémoire de l'arbre obtenu est la suivante.



Il nous faut ajouter un destructeur à notre classe **ArbreBin**. Toute classe en possède un par défaut. Il est appelé automatiquement par exemple lorsqu'une variable locale est détruite à l'issue de l'exécution d'une méthode, ou lorsque l'opérateur **delete** est explicitement utilisé avec l'adresse d'une variable dynamique préalablement créée par **new**.

Or le destructeur par défaut *ne détruit pas les objet pointés par les des variables d'instance de l'objet à détruire.*

Destructeur de la classe ArbreBin

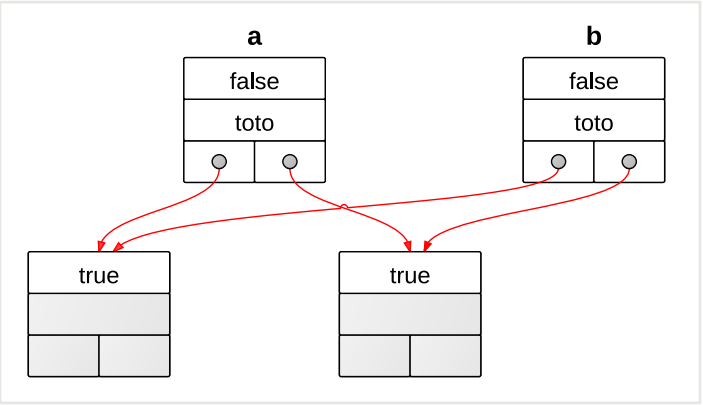
```
ArbreBin::~ArbreBin()
{
    if(!vide)
    {
        delete fd;
        delete fg;
    }
}
```

Si l'arbre n'est pas vide, on doit appeler les destructeurs de ses fils gauche et droits (qui, si applicable, appelleront récursivement les destructeurs de leurs fils etc.).

Fiche n°21 : Structures autoréférentes en C++

Mais il y a encore un problème. Examinez le code ci-contre et la représentation en mémoire des arbres créés.

```
ArbreBin a;
a.ajoute("toto");
ArbreBin b=a;
```



Il s'agit d'une *copie superficielle*, c'est à dire que les variables d'instances de type pointeurs pointent sur les mêmes adresses.

La modification d'un des deux arbres entraînera une modification de l'autre puisque tous les nœuds à l'exception des racines seront communs aux deux arbres.

Pire, la destruction d'un des deux arbres supprimera les nœuds de l'autre.

Pour éviter ce problème, il faut définir un **constructeur par copie.** Dans notre exemple, ce constructeur fera appel à une méthode de clonage.

```
ArbreBin* ArbreBin::clone()
{
    ArbreBin* r = new ArbreBin();
    r->vide=vide;
    if(!vide)
    {
        r->st=st;
        r->fd=fd->clone();
        r->fg=fg->clone();
    }
    return r;
}
```

Réalise une copie d'un arbre complètement indépendante de l'original et retourne l'adresse de cette copie.

Constructeur par copie de la classe ArbreBin

```
ArbreBin::ArbreBin(const ArbreBin& a)
{
    vide=a.vide;
    if(!vide)
    {
        st=a.st;
        fd = a.fd->clone();
        fg = a.fg->clone();
    }
}
```

Mais ***attention***, ce constructeur en copie n'est appelé que lors d'une initialisation d'une variable de type **ArbreBin**, telle que :

ArbreBin b=a;

Et pas lors d'une assignation telle que :

ArbreBin b;

b=a;

Il faut donc, en plus, surcharger l'opérateur d'affectation = pour qu'il aie un comportement similaire.

```
ArbreBin& ArbreBin::operator=(const ArbreBin& a)
{
    vide=a.vide;
    if(!vide)
    {
        st=a.st;
        fd = a.fd->clone();
        fg = a.fg->clone();
    }
    return *this;
}
```

Surcharge de l'opérateur = pour que l'affectation de variables de type **ArbreBin** réalise une copie en profondeur à la place du comportement par défaut qui est une copie superficielle.

Signalons au passage que tous les opérateurs du C++ (+,-,+=,*,&&, etc.) peut être surchargés de manière à adapter leur comportement à de nouvelles classes.

Fiche de préparation n°6

Réalisez une méthode

ArbreBin* ArbreBin::cherche(string s)

Qui recherche si une chaîne est dans l'arbre. Si la chaîne est trouvée, la méthode doit retourner l'adresse du sous-arbre à la racine duquel elle est stockée, sinon elle doit retourner la valeur NULL.

Quelle est, dans le pire des cas, la complexité algorithmique de la recherche d'une chaîne dans un arbre binaire trié ? Illustrez le "pire des cas" par un dessin.