**Prof : elie Hamouche année : 2019**

**Structures de données**

**TD9- arbres binaires Math et piles**

Le but de cet exercice est de gérer et simplifier les **expressions mathématiques complexes** d’une manière automatique .Afin de comprendre comment coder une telle expression à l’aide de l’arbre, voir le cas des exemples suivants:

E1 = (x+2) sin(x\*x) –cos(x) E2: a-(b+c\*(d/e))\*f

 - - niv0

 \* cos a \* niv1

+ sin x + f niv2

x 2 \* b \* niv3

 x x c / niv4

 d e niv5

**arbre a**  **arbre b**

\* \*

 + + + -

 5 \* \* - 1 2 3 4

 2 3 10 4 9 8

 **Arbre C (incomplet) Arbre d (parfait)**

Une telle expressions arithmétique sur les entiers utilise seulement les opérateurs + , -, \*, / Ces expressions peuvent être représentées par des arbres binaires dont les nœuds internes sont étiquetés par l’un des 4 opérateurs(+, -, \*, / ) tandis que les feuilles sont étiquetées par des entiers.

par exemple, les expressions E1et E2 , E3et E4 sont représentés par les figures ci-haut sont écrites comme suit:

 **par exemple pour l’ arbre a : E1 = (x+2)\*sin(x^2) –cos(x)**

**par exemple pour l’ arbre b : E2 = a-(b+c\*(d/e))\*f 🡺 donc une**

**écriture préfixe de E2 {-, a, \*, +, b, \*, c, /, d, e,\*, f}**

**écriture infixe de E2 = {a, -, b, +, c, \*, d, /, e, \*, f}**

**écriture postfixe de E2 = E2 { b, c, d, e, /, \*, +, a, f,\*, -}**

**autres exemples :**

**Ei= 3+(5\*((-2)\*6\*8/4-4/2\*6/3/2)\*7)**

**Ej= sqrt(y) -(cos(x) +y\*(e^x/sin(x)))\*ln(y)**

**Questions**

**a) Ecrire l’arbre binaire C au format Préfixe, Infixe et Postfixé ?**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \* | + | 5 | \* | 2 | 3 | + | \* | 10 | 4 | - | 9 | 8 |

**Ecriture Préfixe 🡺**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 2 | 3 | \* | + | 10 | 4 | \* | 9 | 8 | - | + | \* |

**Ecriture Postfixe 🡺**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | + | 2 | \* | 3 | \* | 10 | \* | 4 | + | 9 | - | 8 |

**Ecriture Infixe 🡺**

**Ecriture parenthèse E = (5+ (2\*3)) \* ((10\*4) + (9-8))**

1. Construire une **structure arbre** définie par une donnée \*char, et 2 pointeurs gauche(fg) et droite (fd) ?
2. Construire une **structure pile** par liste qui contient une info x et un pointeur \*suiv ?
3. pour simplifier la démarche de création d’un arbre binaire arithmétique , il est conseillé de créer une classe TD\_Arbre muni d’un **constructeur plein** ?
4. Construire les arbres binaires a base Mathématique?
* 3.x^2-5x+2 ?
* 3.cosx.sinx-5lnx+2sinx.e^x+5

***on suppose que les fonctions init(pile \*p) , empiler(pile \*p. char pelem), depiler(pile \*p) , peek(pile \*p), estvide(pile \*) , afficherPile(pile \*p) existent***

 **Ecrire des fonctions et procédures permettant** :

1. **de créer un Arbre binaire à partir d’une expression mathématique E** en utilisant la gestion des piles par liste chainée : arbre \* createArbre(char x) ;

. ajouter un element x(char) a l’arbre Math : arbre \* insertR(arbre \* a, char x);

1. d’ afficher l'arbre au **format préfixe,infix ete postfixe par récursivité et itération ?**

void prefixeR(arbre \* a);

void infixeR(arbre \* a);

void postfixeR(arbre \* a);

1. **Ecrire une fonction récursive qui réalise l'insertion d'un nœud dans un arbre Math :**

 arbre \* insertR(arbre \* a, char x)

1. **de compter le nombre de feuilles , des noeuds et le nombre des nœuds internes dans** l’arbre :

 int nbfeuilles(arbre \*a);

int nbNoeuds(arbre \* a);

int nbNoeudsInternes(arbre \* a);

1. **de compter le nombre de niveaux**  dans l’arbre : int nbNiveaux (arbre \*a);
2. **de rechercher un noeud** donné dans l’arbre : void recherche(arbre \* a, char x);

***NB :Utiliser les primitives ou les fonctions des piles et écrire des fonction itératives***

1. **qui réalise l’affichage au format d’un parcours préfix en utilisant la pile**

void ParocoursPrefixI(arbre \*a) ;

1. **de créer une pile à partir d’un tableau et évaluer une expression arithmétique a partir de cette dernière**

 pile\* createPilefromTab(char \*tab); void Evaluer\_pile(Pile \*p) ;

1. **de créer un arbre binaire mathématique à partir d’un tableau** puis évaluer la valeur de l’expression dans l’arbre a :

arbre\* createPilefromTab(char \*tab); double eval(arbre \* a);

1. **d’écrire une fonction qui évalue l’expression mathématique sous forme arbre binaire**  : void evaluer\_arbre(arbre \*a, Pile \*p)
2. **d’écrire une fonction récursive de création d’un arbre représentant une expression mathématique donnée au format préfixe**

void CreateArbrePrefixeR(arbre \*a, Pile \*prefixe);

1. **d’écrire une expression parenthésé à partir de l’arbre binaire**

 void EcritureExpParentheseR(arbre \*a, Pile \*p);

1. **Ecrire une Fonction qui donne la dernière occurrence de la pile P ?**

int Derniereoccurence(Pile \*p, int k)

1. **consrtuire un arbre binaire a base mathematique a partir d’une pile P ?**

 Void construireArbre(String epf,Pile \*P)

1. **d’écrire une fonction qui renvoie la dérivée d’une expression** Math sous forme d’arbre en appliquant l’algorithme récursif d’après : arbre \* dérivée(arbre \*a) qui dérive une expression arithmétique par rapport à la variable (x par exemple).

On admet les cas suivants :

* 1. *la derivée d’un entier = 0*
	2. *la derivée de x est 1*
	3. *la derivée d’une somme (f+g)’ = f’+g’*
	4. *la derivée d’un produit f.g = f’.g+f.g’*

La dérivée de l’expression f+g est f’+g’, celle de f\*g est f’\*g+f\*g’, etc. L’arbre dérivé est un arbre indépendant de l’expression d’origine :

•Si le nœud "r" est un nombre constant, on retourne un nœud contenant la valeur ‘0’.

•Si le nœud "r" est la variable x, on retourne un nœud contenant la valeur ‘1’.

•Si le nœud "r" est un +, on a alors une expression du type f+g : (fg) + (fd)

 Sa dérivé est donc f’+g’. Il faut donc créer un nœud "a" de type opérateur contenant ‘+’.

 Il faut mettre f’ (la dérivé du fg du noeud "r") dans le fils gauche de "a"

De même, le fd contient la dérivé du fils droit du noeud "r" .

*

• Si c’est un \*, la dérivé est un peu plus complexe et l’arbre dérivé est:

Le code de cette fonction deriv pour l’opérateur plus ressemblera donc à cela

*

**Ecrire une fonction récursive renvoyant la dérivée d’un arbre** : arbre \* derivee(arbre \* a) ;

**exemples: r1 = 2+x la dérivée r1’ = 0+1 =1 r2 = x^2 + 2x + 1 la dérivée r2’ = 2\*x+2+0 = 2x+2 r3= : (((x+2)\*sin(x\*x)-cos(x)) la dérivée (r3)’ = (((x+0)sin(x\*x) + ((x+2)\*(((1\*x)+(x\*1)\*cos(x\*x)))))-(-1\*sin(1\*sin(x)))).**

1. **.** Developper le **programme procipal main pour simuler le problème de l’arbre binaire** a expression arithmétique ?

**corrigé**

#include "stdafx.h"

#include <iostream>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

using namespace std;

#define NB\_MAX 30

Les arbres représentés par l’expression math, les suivants :

E = ((1+2)\*(3-4))

E1 = (5 - 4) / (3 + 2 \* 1).

E2= 3+(5\*((-2)\*6\*8/4-4/2\*6/3/2)\*7)

E3= 3 / 8 - 4 - 4

/\* ARBRE binaire sur une expression math

E4 = a-(b+c\*(d/e))\*f

E5= sqrt(y) -(cos(x) +y\*(e^x/sin(x)))\*ln(y)

infixe = {a,-,b,+,c,\*,d,/,e,\*,f}

préfixe = {-,a,\*,+,b,\*,c,/,d.e,f}

\*/

// structure pile par list chainee

typedef struct elem {

 char x; // donnée

 struct elem \*suiv;

} Pile;

typedef struct arbre

{

 arbre \* fg;

 arbre \* fd;

 char data;

}arbre;

class TD\_ARBRE

{

 // Variables membres

private:

public:

 TD\_ARBRE \*fg;

 char c;

 TD\_ARBRE \*fd;

 TD\_ARBRE() {} // constructeur vide

 // constructeur plein

 TD\_ARBRE(TD\_ARBRE \*g, char nc, TD\_ARBRE \*d){

 this->fg = g;

 this->c = nc;

 this->fd = d;

 }

 /\* arbre binaire mtah au format préfixe

 \*

 + -

 1 2 3 4

 \*/

*// 1. de créer un Arbre binaire à partir d’une expression mathématique E en nutilisant un constructeur plein*

TD\_ARBRE \* creerARBRE\_Prefix() {

 new TD\_ARBRE(new TD\_ARBRE(new TD\_ARBRE(NULL, '1', NULL),

 '+',

 new TD\_ARBRE(NULL, '2',NULL),

 '\*',

 new TD\_ARBRE(new TD\_ARBRE(NULL, '3', NULL),

 '-',

 new TD\_ARBRE(NULL, '4', NULL)));

 }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Gestion pile

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

 // Initialiser une pile

 Pile \*InitPile(Pile \*p) {

 p = NULL;

 return p;

 }

 // empiler un élément au sommet de la pile

 void Empiler(Pile \*p, char pelem)

 {

 Pile \*nouv;

 nouv = (Pile \*)malloc(sizeof (Pile));

 if (nouv == NULL) cout << "Erreur, pile pleine" << "\n";

 else

 {

 nouv->x = pelem;

 nouv->suiv = p;

 p = nouv;

 cout << " un element est ajoute au sommet de la pile " << nouv->x << "\n";

 }

 }

 // Dépiler un élément du sommet de la pile

 char Depiler(Pile \*p ) {

 TD\_ARBRE ob;

 char pelem;

 Pile \*q = NULL;

 if (ob.EstVide(p))

 return NULL; // echec

 else

 pelem = p->x;

 q = p;

 p = p->suiv;

 delete(q);

 return pelem;

 }

 // Vérifier si une pile est vide

 int EstVide(Pile \*p)

 {

 return (p == NULL) ? 1 : 0;

 }

 // Accès au sommet de la pile

 char peek(Pile \*p)

 {

 TD\_ARBRE ob;

 char pelem;

3 if (ob.EstVide(p)) return NULL;

 else

 {

 pelem = p->x;

 }

 return pelem;

 }

 // Fonction qui affiche les données d’une pile

 void AffichePile(Pile \*p){

 Pile \*q;

 q = p;

 while (q != NULL){

 cout << p->x << "\n";

 q = q->suiv;

 }

 }

 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

 // Gestion arbre binaire math et pile par récursivité

 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

 //1. fonction création arbre binaire a base math

 arbre \* createArbre(char x)

 {

 arbre \* a = (arbre\*)malloc(sizeof(arbre));

 a->data = x;

 a->fd = NULL;

 a->fg = NULL;

 return a;

 }

 // 2. Fonction qui affiche une expression préfixe, Prostfixé et infixe

 void prefixeR(arbre \* a)

 {

 TD\_ARBRE ob;

 if (a == NULL)

 return;

 cout << a->data << " ";

 ob.prefixeR(a->fg);

 ob.prefixeR(a->fd);

 }

 void postfixeR(arbre \* a)

 {

 TD\_ARBRE ob;

 if (a == NULL)

 return;

 ob.postfixeR(a->fg);

 ob.postfixeR(a->fd);

 cout << a->data << " ";

 }

 void infixeR(arbre \* a)

 {

 TD\_ARBRE ob;

 if (a == NULL)

 return;

 ob.infixeR(a->fg);

 cout << a->data << " ";

 ob.infixeR(a->fd);

 }

 //3. Fonction récursive qui réalise l'insertion d'un nœud dans un arbre Math

 arbre \* insertR(arbre \* a, char x)

 {

 TD\_ARBRE ob;

 arbre \*tmp = (arbre\*)malloc(sizeof(arbre));

 if (x != NULL) {

 if (a->fg != NULL)

 tmp = ob.insertR(a->fg, x);

 else

 tmp->fg = ob.createArbre(x);

 }

 else{

 if (a->fd != NULL)

 tmp=ob.insertR(a->fd, x);

 else

 tmp->fd = ob.createArbre(x);

 }

 return tmp;

 }

 // Fonction qui vérifie s’il y a feuilles dans l’arbre

 bool isfeuille(arbre \*a)

 {

 if (a->fg == NULL && a->fd == NULL)

 return true;

 else

 return false;

 }

 //4.a Fonction qui évalue le nombre des feuilles dans l’arbre

 int nbfeuilles(arbre \*a)

 {

 TD\_ARBRE ob;

 if (a == NULL) return 0;

 if (a->fg == NULL && a->fd == NULL) return 1;

 else

 return (ob.nbfeuilles(a->fg) + ob.nbfeuilles(a->fd));

 }

 //4.b fonction qui compte nb de nœuds dans arbre

 int nbNoeuds(arbre \* a)

 {

 TD\_ARBRE ob;

 if (a == NULL)

 {

 return 0;

 }

 else

 {

 return(1 + ob.nbNoeuds(a->fg) + ob.nbNoeuds(a->fd));

 }

 }

 //4.c fonction qui compte nb de nœuds internes dans arbre

 int nbNoeudsInternes(arbre \* a)

 {

 int res = nbNoeuds(a)- nbfeuilles(a) ;

 return res ;

 }

 // *5. Fonction qui compte le nombre de niveaux dans l’arbre*

 int NbNiveaux(arbre \*a)

 {

 TD\_ARBRE ob;

 int res = (a == NULL) ? 0 : ob.NbNiveaux(a->fg) > ob.NbNiveaux(a->fd) ? 1 + ob.NbNiveaux(a->fg) : 1 + ob.NbNiveaux(a->fd);

 return res;

 }

 //6. Fonction qui recherche un nœud s’il appartient ou non à l'arbre

 void rechercheR(arbre \* a, char x)

 {

 TD\_ARBRE ob;

 if (a == NULL)

 {

 return;

 }

 if (a->data == x)

 {

 cout << "\n\nL'element est trouve\n\n";

 }

 ob.rechercheR(a->fg, x);

 ob.rechercheR(a->fd, x);

 }

*//7. Fonction qui réalise l’affichage au format d’un parcours préfix en utilisant la pile*

 void ParocoursPrefixI(arbre \*a) {

 TD\_ARBRE ob;

 Pile \*p = new Pile;

 p = NULL;

 ob.Empiler(p, a->data);

 cout << "node" << a->data<< "\n";

 while (a != NULL){

 if (a->fg != NULL)

 {

 a = a->fg;

 ob.Empiler(p, a->data);

 // cout << "node" << a->data << "\n";

 }

 if (a->fd != NULL)

 {

 a = a->fd;

 ob.Empiler(p, a->data);

 // cout << "node" << a->data << "\n";

 }

 }

 if (!ob.EstVide(p)){ // pile pleine

 cout << "arreter " << "\n";

 }

 }

*//8.a Fonction qui crée une pile à partir d’un tableau et évaluer une expression arithmétique a partir de cette dernière*

 Pile\* createPiledeTab(char \*tab)

 {

 TD\_ARBRE ob;

 Pile \* tmp = new Pile;

 for (int i = 0; i <sizeof(tab); i++)

 {

 ob.Empiler(tmp,tab[i]);

 }

 return tmp;

 }

//8.b Fonction qui évalue d'une expression Math en utilisant la pile

 void Evaluer\_pile(Pile \*p) {

 TD\_ARBRE ob;

 char op = NULL;

 int x = ob.Depiler(p); // operande

 int y = ob.Depiler(p);

 if (p != NULL) {

 switch (op) {

 case '+': ob.Empiler(p, x + y); break;

 case '-': ob.Empiler(p, x - y); break;

 case '\*': ob.Empiler(p, x \* y); break;

 case '/': ob.Empiler(p, x / y); break;

 case '%': ob.Empiler(p, x % y); break;

 default: cout << "error " << "\n";

 ob.Empiler(p, (char)x);

 ob.Empiler(p, (char)y);

 }

 }

 }

*// 9.a fonction qui crée un arbre binaire mathématique à partir d’un tableau puis évaluer la valeur*

 arbre\* createArbrefromTab(char \*tab){

 TD\_ARBRE ob;

 arbre \*a = (arbre\*)malloc(sizeof(arbre\*));

 a->data = tab[0];

 a->fg = NULL;

 a->fd = NULL;

 for (int i = 1; i <sizeof(tab); i++){

 ob.insertR(a, tab[i]);

 }

 return a;

 }

//9.b Fonction récursive qui évalue l'arbre binaire math

 double eval(arbre \* a)

 {

 TD\_ARBRE ob;

 double res;

 if (a == NULL)

 res = 0;

 if (a->data == '/')

 res = ((ob.eval(a->fg)) / (ob.eval(a->fd)));

 else

 {

 if (a->data == '-')

 res = ((ob.eval(a->fg)) - (ob.eval(a->fd)));

 else

 {

 if (a->data == '+')

 res = ((ob.eval(a->fg)) + (ob.eval(a->fd)));

 else

 {

 if (a->data == '\*')

 res = ((ob.eval(a->fg))\*(ob.eval(a->fd)));

 else

 res = 0;

 }

 }

 }

 return res;

 }

 //10.fonction qui évalue une expression math sous forme d’un arbre binaire

 void evaluer\_arbre(arbre \*a, Pile \*p)

 {

 TD\_ARBRE ob;

 char x,y;

 int res;

 if (a != NULL) {

 ob.evaluer\_arbre(a->fd, p);

 ob.evaluer\_arbre(a->fg, p);

 if (a->data == '+') {

 x= ob.Depiler(p);

 y= ob.Depiler(p);

 res = (x - '0') + (y - '0');;

 cout << res;

 ob.Empiler(p,(char) res);

 }

 else if (a->data == '-'){

 x = ob.Depiler(p);

 y = ob.Depiler(p);

 res = (x - '0') - (y - '0');

 cout <<res;

 ob.Empiler(p, (char)res);

 }

 else if (a->data == '\*'){

 x = ob.Depiler(p);

 y = ob.Depiler(p);

 res = (x - '0') \* (y - '0');

 cout << res;

 ob.Empiler(p, (char)res);

 }

 else if (a->data == '/'){

 x = ob.Depiler(p);

 y = ob.Depiler(p);

 res = (x - '0') / (y - '0');

 cout << res;

 ob.Empiler(p, (char)res);

 }

 else

 ob.Empiler(p, a->data);

 }

 }

*// 11. fonction qui réalise l’affichage au format d’un parcours préfix en utilisant la pile*

 void CreateArbrePrefixeR(arbre \*a, Pile \*prefixe) {

 TD\_ARBRE ob;

 char pelem;

 if (prefixe != NULL)

 {

 ob.Depiler(prefixe);

 pelem= a->data ;

 if (pelem == '+' || pelem == '-' || pelem == '\*' || pelem == '/')

 {

 a->fg = (arbre\*)malloc(sizeof(arbre));

 a->fd = (arbre\*)malloc(sizeof(arbre));

 ob.CreateArbrePrefixeR(a->fg, prefixe); // appel rec Gauche

 ob.CreateArbrePrefixeR(a->fd, prefixe); // aappel rec droite

 }

 else {

 a->fg = NULL;

 a->fd = NULL;

 }

 }

 }

*//12. fonction qui crée une expression parenthésé à partir de l’arbre binaire*

 void EcritureExpParentheseR(arbre \*a, Pile \*p) {

 TD\_ARBRE ob;

 char x, \*y, \*z;

 Pile \*q;

 char \*res;

 if (a != NULL)

 {

 ob.EcritureExpParentheseR(a->fd, p);

 ob.EcritureExpParentheseR(a->fg, p);

 x = a->data;

 if (x == '+' || x == '-' || x == '\*' || x == '/')

 {

 \*y = ob.Depiler(p);

 \*z = ob.Depiler(p);

 strcpy(res, "(");

 strcat(res, y);

 strcat(res, a->data);

 strcat(res, z);

 strcat(res, ")");

 ob.Empiler(p, res);

 }

 else

 ob.Empiler(p, a->data);

 }

 }

 //13. Fonction qui donne la dernière occurrence de la pile

 int Derniereoccurence(Pile \*p, int k)

 {

 TD\_ARBRE ob;

 Pile \*q;

 char res;

 q = p;

 int i = 1;

 while (!ob.EstVide(q) && i < k)

 {

 i= i + 1;

 q = q->suiv;

 }

 res = ob.peek(q);

 return(res);

 }

*//14. Procédure qui construit un arbre binaire Math à partir d’une d’une chaine comme expression post fixe*

void construireArbre(String epf,Pile \*P) {

 TD\_ARBRE ob;

 Char car = ‘’;

 while (epf.length() != 0) {

 cin>>car ;

 if (epf.length() !=0 )

 {

Arbre \* a = new Arbre;

 a->data= car // Creation Racine de l’arbre

 a->fg = NULL;

 a->fd = NULL;

 }

 if (car == "+" || car=="-" || car=="\*" || car=="/" )

 {

 a->fg = ob.depiler(P);

 a->fd = ob.depiler(P);

 }

 if (car == "")

 ob.empiler(a); ;

 }

 //affichage arbre

 ob.prefixeR(a);

 }

//15. Fonction dérivée par récursivité en tenant compte des opérateurs + et \* seulement

 arbre \* derivee(arbre \* r) {

 TD\_ARBRE ob;

 arbre \*c = NULL;

 arbre \*c1, \*c2;

 if (r != NULL) {

 if (r->data == "+") // C’est un +

 {

 c = ob.createted("+"); // Creation d’un noeud +

 c->fg = ob.derivee(r->fg); // Copie de la dérivée du fg

 c->fd = ob.derivee(r->fd); // Copie de la dérivée du fd

 }

 else if (r->data == "\*") // C’est un \*

 {

 c1 = ob.createted("\*"); // Creation d’un noeud \*

 c1->fg = ob.derivee(r->fg); // Copie de la dérivée du fg

 c1->fd = r->fd; // Copie de fd

 c2 = ob.createted("\*"); // Creation d’un noeud \*

 c2->fg = r->fg; // Copie de fg

 c2->fd = ob.derivee(r->fd); // Copie de la dérivée du fd

 c = ob.createted("+"); // Creation d’un noeud +

 c->fg = c1->fg; // Copie le fg de c1

 c->fd = c2->fd; // Copie le fd de c2

 }

 else {

 return NULL;

 }

 }

 return c; // On retourne l’arbre dérivé créé

 }

 //15.b Procédure qui teste la fonction dérivée

 void testerDerivee() {

 TD\_ARBRE ob;

 arbre \*a, \*b;

 arbre \*tmp1,\*tmp2;

 tmp1 = tmp2 =NULL;

 arbre \*res1 = NULL;

 arbre \*res2 = NULL;

 //creation arbre a

 a = ob.create();

 // Il sera plus simple de créer un tableau qui correspond au

 Caractères de l'expression e1 et e2

 // E1 = 2\*x

 char \*ta[] = { "\*", "2", "x"};

 b = ob.createArbrefromTab(ta);

 cout << "arbre E1 " << "\n";

 ob.prefixe(a);

 cout << "\n\n";

 b = ob.create();

 // E2 = x\*x+2\*x+1

 char \*tb[] = { "+", "+", "1", "\*", "\*", "x", "x", "2", "x" };

 b = ob.createArbrefromTab(tb);

 cout << "arbre E2 " << "\n";

 ob.prefixe(b);

 cout <<"\n\n";

 // derivees

 res1= ob.derivee(a);

 res2 = ob.derivee(b);

 // affichage au format préfixe

 ob.prefixe(res1);

 ob.prefixe(res2);

 }

}; // fin class

// Programme principal

int main()

{

 TD\_ARBRE ob;

 int n;

 arbre \* a,\*tmp;

 Pile \*p = NULL;

 char tab[] = { '\*', '+', '1', '2', '-', '3', '4' };

 // creation des noeuds arbre

 a = (arbre\*)malloc(sizeof(arbre));

 tmp= (arbre\*)malloc(sizeof(arbre));

 // creation arbre from tab

 a = ob.createArbrefromTab(tab);

 cout << "Parcours Prefixe de l'arbre : \n\n";

 ob.prefixeR(a);

 cout << "Parcours Postfixe de l'arbre : \n\n";

 ob.postfixeR(a);

 cout << "Parcour Infixe de l'arbre : \n\n";

 ob.infixeR(a);

 cout << "Le nbre de noeud est : \n\n" << ob.nbNoeuds(a)<<"\n";

 cout << "le nbre de feuilles est : " << ob.nbfeuilles(a) << "\n";

 cout << "le nbre de niveaux : " << ob.NbNiveaux(a) << "\n";

 cout << "evaluation de l'arbre a donne :" << ob.eval(a) << "\n";

 // createe pile from tab et evaluation

 p = ob.createPiledeTab(tab);

 ob.AffichePile(p);

 ob.Evaluer\_pile(p);

 // derniere occurrence dans la pile

 int res = ob.Derniereoccurence(p, 10);

 cout << "dernier occ dan la pile" << res << "\n";

 // Fonction itérative de l’arbre a

 ob.ParocoursPrefixI(a);

//b-évaluation expression math sous forme arbre

 ob.evaluer\_arbre(a, p);

 cout << "------------------------------------" << "\n";

//d- fonction récursive de création d'un arbre représentant une expression arithm. donnée au format préfixe

 ob.CreateArbrePrefixeR(a, p);

 cout << "------------------------------------" << "\n";

//e fonction récursive d'écriture d'une expression arithm. sous forme parenthèse a partir d'un arbre

 ob.EcritureExpParentheseR(a, p);

//e fonction recursive dérivée

 ob.testerDerivee()

 cout << "programme terminé ...." << "\n";

 system("PAUSE");

 return 0;

}