Prof : Elie Hamouche année : 2019

**Structures de données**

**Ex1 supplément : Arbres binaires de recherches ABR**

Soit (**a**) un arbre binaire défini comme suit :

parent

Node

fg

fd

* 1. **Définir une structure** Node permettant de coder un nœud de l’arbre biaire de recherche contenant une valeur entière et de pointer vers ses 2 fils fg et fd ?

// 1 structures arbre binaire de recherche ABR

 typedef struct node{

 int val;

 struct node \*parent;

 struct node \*fg, \*fd;

 }ARB;

* 1. Ecrire une fonction ARB **\* CreateArbre()** qui crée un arbre vide ?

// 2.fonction qui crée un arbre vide

ARB \*createArbre()

{

 ARB \*a = new ARB;

 a->val = NULL;

 a->parent = NULL;

 a->fg = NULL;

 a->fd = NULL;

 return a;

}

* 1. Ecrire une fonction récursive **Node\* DetruireArbre(Node\* a)** qui libère la mémoire occupée de tous les nœuds de l’arbre binaire de recherche ?

//3. Fonction récursive qui libère la mémoire occupée de tous les nœuds de l’arbre binaire de recherche ABR cad ou dtruire l’arbre ABR

void detruireArbreR(ARB \*a)

{

 ARB \*tmp = a; if (tmp != NULL) {

 detruireArbreR(a->fg);

 detruireArbreR(a->fd);

 delete(tmp);

 }

 tmp = NULL;

 cout << "\nL'arbre est supprimee\n";

}

* 1. Ecrire une fonction **récursive** ARB \* insert(ARB \*a, int ov)

 qui ajoute une valeur entière correctement placée comme feuille dans

 l’arbre binaire de Recherche ?

//4.fonction récursive qui ajoute une valeur entière correctement placée comme feuille dans l’arbre binaire de Recherche ABR

ARB \* insert(ARB \*a, int ov)

{

 ARB \*tmp = new ARB;

 tmp->val = ov;

 tmp->fg = NULL;

 tmp->fd = NULL;

 if (a == NULL)

 {

 a = tmp; return a;

 }

 if (ov < a->val)

 {

 a->fg = insert(a->fg, ov);

 }

 if (ov > a->val)

 {

 a->fd = insert(a->fd, ov);

 }

 return a;

}

* 1. Donner l’affichage **infixe** de l’arbre binaire de recherche obtenu en justifiant le raisonnement ?

//fonction qui donne l’affichage infixe de l’arbre binaire de recherche obtenu en justifiant le raisonnement

void parcoursInfixe(ARB \*a)

{

 if (a != NULL)

 {

 parcoursInfixe(a->fg);

 cout << a->val << " ";

 parcoursInfixe(a->fd);

 }

}

* 1. Ecrire une fonction itérative **bool compare(Node \*a , Node \* b)** qui compare 2 arbres binaires de recherche . la fonction retourne true ssi les 2 arbres binaires ont la même structure et qui contiennent les mêmes valeurs aux nœuds correspondants ?

//6. fonction récursive qui compare 2 arbres binaires de recherche (structure et valeurs)

bool compare(ARB \*a, ARB \*b)

{

 if (a == NULL && b == NULL) return true;

 if (a == NULL || b == NULL) return false;

 if (a->val != b->val) return false;

 bool tmp = compare(a->fg, b->fg);

 if (!tmp) return false;

 return (compare(a->fd, b->fd));

}

* 1. Construire un arbre binaire de recherche par insertion des valeurs entières l’une après l’autre dans l’ordre suivant : **{25,60,35,10,5,20,65,45,70,40,50,55,30,15}** ?

déduire le nombre de niveau et la hauteur h ?

 25 niv0

 < >

 20 60 niv1

 10 35 65 niv2

 La hauteur h = h(55) = 6

5 20 30 45 70 niv3

 15 50 niv4

 55 niv5

* 1. Donner l’arbre binaire obtenu après insertion des valeurs entières à la racine de l’arbre dans l’ordre suivant :{ **18,12}** ?

 18

 < >

 12 25 niv0

 20 60 niv1

 10 35 65 niv2

 La hauteur h = h(55) = 6

5 20 30 45 70 niv3

 15 50 niv4

 55 niv5

* 1. Est-ce que l’arbre obtenu est toujours un arbre binaire de recherche. justifier votre réponse ? **oui**
1. .Supprimer le nœud **45** ? puis **35**? construis le nouveau ABR ?

Le nœud **45** ayant 1 seul fils de descendance, supprimer le père revient a le remplacer par son fils unique c.a.d **50** et l’arbre est tjrs de recherche ABR , l’arbre ABR devient :

 18

 < >

 12 25 niv0

 20 60 niv1

 10 35 65 niv2

 La hauteur h = h(55) = 5

5 20 30 50 70 niv3

 15 55 niv4

Le nœud **35** ayant 2 fils de descendance, supprimer le père revient a le remplacer par son **min(SAG)** et l’arbre est tjrs recherche ABR .

Donc l’arbre ABR devient :

 18

 < >

 12 25 niv0

 20 60 niv1

 10 30 65 niv2

 La hauteur h = h(55) = 5

5 20 50 70 niv3

 15 55 niv4

**Ex2 supplément : Arbres binaires de recherches ABR**

Soit un T tableau qui contient un les éléments suivants :

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **10** | **6** | **8** | **4** | **9** | **7** | **2** | **17** | **20** | **25** | **30** | **35** |

1. Écrire une fonction qui créeun arbre de recherche ABR ? dessiner cet arbre ?

 10 niv0

 < >

 6 17 niv1

 4 8 20 niv2

 La hauteur h = h(55) = 6

2 7 9 25 niv3

 30 niv4

 35 niv5

typedef struct node {

 int \*val;

 node \* fg;

 node \* fd;

 }ABR ;

1. Écrire une fonction **itérative** qui affiche l’arbre ABR **?**
2. Écrire une fonction qui créeun arbre ABR à partir des éléments du tableau

 ABR \* createArbreFromTable(int \* t)?

ABR \* createArbreFromTable(int \* t) {

 int i = 0;

 ABR \* c = new ABR ; // null

 for (i = 0; i<sizeof(t) ; i++) {

 c = insertnodeR (c, t[i]);

 }

 return c;

}

 ABR \* insertnodeR (ABR \* a, int ov) {

 if (a == NULL || a->val == NULL) {

 ABR \* c = new ABR ; // null

 x->fg = NULL;

 x->fd = NULL;

 x->val = ov;

 return x;

 }

 else {

 if (a->val > ov) {

 if (a->fd != NULL) {

 insertnodeR (a->fd, ov);

 }

 }

 if (a->val < s) {

 if (a->left != NULL) {

 insertnodeR (a->fg, s);

 }

 }

 }

 return a;

}

1. Écrire une fonction qui donne la hauteur de l’arbre ABR

int height(ABR \*a)

int maxi(int a, int b) {

 if (a>b) {

 return a;

 }

 else {

 return b;

 }

}

int height(arbre \*a) {

 if (a == NULL) {

 return 1;

 }

 else {

 return (maxi(height(a->fg), height(a->fd)) + 1);

 }

}

1. Ecrire une fonction qui vérifie si ABR est totalement équilibré c.a.d **: | hg-hd| =0** bool arbreEquilibre(ABR \* a)

 bool arbreEquilibre(ABR \* a) {

 int l = height (a->fg);

 int r = height (a->fd);

 if (l == r) {

 return true;

 }

 else {

 return false;

 }

 }

1. Compter par **récursivité** le nombre des nœuds par niveau int **nombreNodesparNiveau(ABR\* a)** ?évaluer la somme des valeurs contenues dans chacune des branches gauches(fg) et droites(fd) à partir de la racine ? déduire le total des valeurs?

int **nombreNodesparNiveau** (ABR \* a) {

 int count = 0;

 if (a == NULL) {

 return 0;

 }

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

 count = 2;

 }

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

 count = 1;

 }

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

 count = 1;

 }

 count = count + nombreNodesparNiveau (a->fg) + nombreNodesparNiveau (a->fd);

 return count;

}

Void **sommeparNiveau**(ABR \* a) {

 int count = 0;

 int s1,s2,s3,s4 ;

 if (a == NULL) {

 cout<< "somme des valeurs est null " <<" \n";

 }

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

 count = 2;

 s1+ = (a->val)+sommeparNiveau(a->fg)+ sommeparNiveau(a->fd) ;

cout<< "somme des valeurs est :" <<s1<< " \n";

 }elseif (a->fg != NULL && a->fd == NULL) {

 count = 1;

 s2+ = (a->val)+ sommeparNiveau(a->fg) ;

cout<< "somme des valeurs est : " <<s2<< " \n";

 }elseif (a->fd != NULL && a->fg == NULL) {

 count = 1;

 s3+ = (a->val)+ sommeparNiveau(a->fd ) ;

cout<< "somme des valeurs est : " <<s3<< " \n";

 }else{

 s4 + = (a->val) + sommeparNiveau (a->fg) + sommeparNiveau (a->fd);

 cout<< "somme des valeurs est : " <<s4<< " \n";

 }

 cout<< "total des valeurs est : " <<s1+s2+s3+s4 << " \n";

}

1. Ecrire une fonction **récursive** qui donne les valeurs des nœuds comprises entre **inf et sup** d’après void **Between** (ABR \* a, int inf, int sup) **?**

void BetweenR(ABR \* a, int inf, int sup) {

 if (a != NULL) {

 if (a->val>inf && a->val<sup) {

 cout << a->val << " ";

 }

 if (a->left != NULL) {

 BetweenR (a->fg, inf, sup);

 }

 if (a->right != NULL) {

 } BetweenR(a->fd, inf, sup);

 }

}

1. Ecrire une fonction qui effectue la recherche d’un nœud dans un ABR ABR **\* recherchenoeud ( ABR\* a,** int ov**) ?**

ABR\* recherchenoeud (ABR \* a, int ov)

{

 if (x == a->val) return a;

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

 ABR \* tmp = new ABR;

 tmp = recherchenoeud(a->fg, x);

 if (tmp != NULL) return tmp;

 tmp = recherchenoeud(a->fd, x);

 return z;

}

1. Ecrire une fonction **récursive** qui vérifie si l’arbre **ABR** est binaire de recherche bool **VerifBinaireABR (ABR \*a**)?

bool verifierBinaireABR(ABR\* a)

{

 if (a == NULL) return true;

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

 if (a->fd != NULL)

 {

 if (a->fd->val < a->val) return false;

 }

 if (a->fg != NULL)

 {

 if (a->fg->val > a->val) {

 return false;

 }

 }

 bool x = verifierBinaireABR (a->fg);

 if (!x) return false;

 return verifierBinaireABR (a->fd);

}

1. Ecrire une fonction qui retrouve **la valeur minimale** dans ABR**int min(ABR \*a)?** supprimer la  ABR **\* supprimerMin(ABR \*a)?**

int minimumdansABR(ABR\* a){

 if (a == NULL) return 0;

 if (a->fg == NULL) return a->val;

 else

 return minimumdansABR(a->fg);

}

ABR\* supressionMin(ABR\* a){

 ABR \* tmp = new ABR ; //NULL

 if (a == NULL) return NULL;

 else {

 tmp = supressionMin (a->fg);

 delete(tmp) ;

 }

 return a;

 }

1. Ecrire une fonction **qui fusionne 2** arbres binaires de recherches ABR void **fusionR (ABR \*a, ABR \*b)?**

ABR fusionR(ABR \*a, ABR \*b)

{

 ABR\* tmp = new ABR;

 if(a== NULL) return b ;

 if(a== NULL) return a ;

 if(a!=NULL && b!=NULL){

 tmp = insertnodeR (a,b->val );

 // récursivité gauche et droite

 fusionR(a->fg,tmp) ;

 fusionR(a->fd,tmp) ;

 }

**}**

1. Ecrire une f onction récursive qui coupe un arbre binaire de recherche ABR à partir d’une valeur x entre comme argument : **void couperR(ABR\*a , int x)**  ?

Void couperR(ABR \*a,int x)

{

ABR \*GA = new ABR ; // SAG

ABR \*DA = new ABR ; // SAD

 if (a == NULL) {

 ABR \* c = new ABR ; // null

 GA = NULL;

 GD = NULL;

 }

 else if (X < a->val ) {

 DA = a;

 couperR (a->fg, x); //récursivité gauche

 } else if (X > a->val ) {

 GA = a;

 couperR (a->fd, x); //récursivité droite

 }else{

 afficheArbreinfixe(GA) ;

 afficheArbreinfixe(DA) ;

 }

1. Ecrire une fonction itérative qui supprime un nœud dans ABR

ABR **\* supprimeRnodeI( ABR\* a, int ov)?**

ABR \* SupprimerNoeud(ABR \*a, int ov){

 ABR \* tmp = new ABR;

 if (a->val == ov){

 tmp = a;

 if (tmp->fg == NULL) return tmp->fd;

 else{

 a = tmp->fg;

 while (a->fd != NULL){

 a = a->fd;

 }

 a->fd = tmp->fd;

 return tmp->fg;

 }

 delete tmp;

 }

 else{

 if (a->val > ov){

 a->fg = SupprimerNoeud(a->fg, ov);

 }

 else{

 a->fd = SupprimerNoeud(a->fd, ov); //sinon on va a droite

 }

 }

 return a;

 }

1. Ecrire une fonction qui effectue une rotation gauche de l’arbre de recherche ABR **\* rotationGauche( ABR\* a) ?**

ABR\* rotationGauche(ABR \*a)

{

 ABR\* b = a->fd;

 a->fd = b->fg;

 b->fg = a;

 return b;

}

**Ex3 supplément : Arbres binaires à base Math et piles**

On désire remplacer utiliser **l’arbre binaire** à based’une expression **mathémati,ques E** , illustré comme suit :

 +

  **/ /**

 **\* 5 30 -**

 **2 - 10 4**

1. **14**
2. Citer les noeuds au format préfix, infixe? postfixe ? déduire E   ?

Résultat préfixe = **{ +, / , \* , 2 , - , 4 , 14 , 5 , / , 30 , - , 10, 4**}

Résultat infixe(parenthèse) = **{ 2, , 4 ,- 14 , / , 5 , + , 30 , / , 10 , -, 4}**

* E= ***[ 2 \*(4-14))/ 5 + 30/(10--4)]***

Résultat postfixe= {**2, 4 , 14 , - , \* , 5 , / , 30 , 10 , 4 , - , /, +}**

1. Ecrire l’expression Mathématique E de cet arbre au **format infixe ? préfixe ? postfixe** ?

//pile par list chainee

Typedef struct cell

{

 char \* x;

 struct cell \*next ;

}pile;

Typedef struct node {

 Char val;

 node \*fg;

 node \*fd;

}arbre;

void afficheArbreinfixe(arbre \* a) {

 if (a != NULL) {

 afficheArbrePostfixe(a->left);

 cout << a->val << " ";

 afficheArbrePostfixe(a->right);

 }

}

void afficheArbrePrefixe(arbre \* a) {

 if (a != NULL) {

 cout << a->val << " ";

 afficheArbrePostfixe(a->left);

 afficheArbrePostfixe(a->right);

 }

}

void afficheArbrePostfixe(arbre \* a) {

 if (a != NULL) {

 afficheArbrePostfixe(a->fg);

 afficheArbrePostfixe(a->fd);

 cout << a->val << " ";

 }

}

1. Ecrire une fonction **récursive** qui évalue cette expression E en utilisant l’arbre binaire ?

En sachant que P est une pile et E une expression mathématique , tel que

int getArbreMathValue(arbre \* a) {

 if (a != NULL) {

 if (a->val == "/" || a->val == "\*" || a->val == "+" || a->val == "-") {

 if (a->val == "/") {

 return getArbreMathValue (a->left) / getArbreMathValue (a->right);

 }

 if (a->val == "\*") {

 return getArbreMathValue e(a->left) \* getArbreMathValue (a->right);

 }

 if (a->val == "-") {

 return getArbreMathValue (a->left) - getArbreMathValue (a->right);

 }

 if (a->val == "+") {

 return getArbreMathValue (a->left) + getArbreMathValue (a->right);

 }

 }

 else {

 return atoi(a->val);

 }

 }

}

***E = [ 2 \*(4-14))/ 5 + 30/(10--4)] a=’+’ ; b=’\*’ ; c=’-’ ; d=’/’ et e=’-’ ;***

 ***=( (2 b (4 e 14) d 5 ) a (30 d (10 e 4)) )***

3.Evaluer l’expression arithmétique E au format postfixée(par pile) codée sur un tableau de caractères, en supposant pour simplifier que tous les opérateurs sont binaires et limités à (+,−,∗,/) et on utilisant uniquement des nombres sur un caractère ?

void EvalArbreparPile(ABR \*a,pile \*p) { // parcours profondeur

 if (a == NULL ) cout << "error , arbre vide" << "\n";

 else

 {

 int x,y,z;

 //recursivite gauche et droite

 EvalArbreparPile(a->fg) ;

 EvalArbreparPile(a->fd) ;

 while (a != NULL) {

 if(a->val == " + "){

 x= peek(p) ;

 depiler (p);

 y= peek(p) ;

 depiler (p);

 z= x+y ;

 p = empiler(p,z);

 }else if(a->val == " - "){

 x= peek(p) ;

 depiler (p);

 y= peek(p) ;

 depiler (p);

 z= x-y ;

 p = empiler(p,z);

 }else if(a->val == " \* "){

 x= peek(p) ;

 depiler (p);

 y= peek(p) ;

 depiler (p);

 z= x\*y ;

 p = empiler(p,z);

 }

 }else {

 x= peek(p) ;

 depiler (p);

 y= peek(p) ;

 depiler (p);

 z= x/y ;

 p = empiler(p,z);

 }

 Empiler(p,a->val) ;

 }

 }

1. Ecrire une fonction qui remplit 2 piles p1 et p2 par les branches gauche et droite de l’arbre Math **a** ?

// parcours DFS :Depth-First Search en profondeur qui s’appuie sur la gestion des piles par itération

void ParocursPrefixeI(ABR \*a) { // parcours profondeur

 pile \*p1 = new pile; // pile vide

pile \*p2 = new pile; // pile vide

 if (a == NULL ) cout << "error , arbre vide" << "\n";

 else

 {

 P1 = empiler(p1, a->val);

 P2 = empiler(p2, a->val);

 while (a != NULL) {

 if(a->fg !=NULL) {

 p1 = empiler(p1, a->fg->val); // ajout de la partie SG

 }

 if(a->fd !=NULL){

 p2 =empiler(p2, a->fd->val); // ajout de la partie SD

 }

 } //fin while

 cout << "les piles p1 et p2 sont remplis:" << "\n";

 displayPile(P1) ;

 displayPile(P2) ;

 }

 }

1. Ecrire une Procédure qui construit un arbre binaire Math à partir d’une chaine écrite au format d’une expression postfixe*?*

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

 afficheArbreinfixe(a) ;

 }