**Structures et Algorithmes : Entrainement DE**

**La structure d’un arbre :**

Typedef struct NODE

{

NODE\* right; *// un nœud représente un arbre*

NODE\* left;

Int data; *// un nœud contient une valeur*

} Node;

**Créer un arbre :**

New\_tree(e : valeur) tree

-----------------------------------------

Debut

| tree <- allouer NODE // on alloue la mémoire d’un noeud

| tree->data <- e;

| tree->left <- NULL; // on initialise les valeurs

| tree-right <- NULL;

| retourner tree;

Fin

**Nombre d’éléments dans un arbre (tree) :**

Count\_elem(tree : Node) entier

------------------------------------------------

Debut

| Si tree <- NULL

| retourne NULL ;

| Sinon

| retourner 1 + count\_elem(tree.left) + count\_elem(tree.right)

| Fin Si

Fin

**Rechercher un élément dans un tree : (en Booléen (VRAI OU FAUX))**

Search\_elem(tree : Node, element : entier) booléen

------------------------------------------------

Debut

| Si tree <- NULL

| retourne FAUX;

| Fin Si

| Si tree.data <- element

| retourne VRAI ;

| Sinon

| rechercher tree.left ;

| rechercher tree.right ;

| Fin Si

Fin

**Modifier les éléments d’un tree : (ici par exemple nous les multiplierons par 2)**

Multiple\_tree\_fois\_deux (tree : NODE) entier

----------------------------------------

Debut

| Si tree <- NULL

| retourne NULL;

| Sinon

| tree.data <- tree.data \* 2 ;

| Multiple\_tree\_fois\_deux(tree.left) ;

| Multiple\_tree\_fois\_deux(tree.right) ;

| Fin Si

Fin

**Parcourir récursivement un tree :**

Browse\_tree (tree : NODE)

------------------------------------------

Debut

| Si tree <- NULL

| retourne NULL ;

| Fin Si

| Parcours\_tree(tree.left) ;

| Parcours\_tree(tree.right) ;

Fin

**Supprimer un tree :**

Delete\_tree(tree : NODE)

-------------------------------------------

Debut

| Si tree <- NULL

| retourne NULL ;

| Sinon

| Delete\_tree(tree.left);

| Delete\_tree(tree.right);

| Liberer(&tree);

| Fin si

Fin

**Connaitre la profondeur d’un tree :**

* **C’est le chemin, allant de la racine jusqu’au nœud ou feuille, le plus long**

Depth\_tree(tree : NODE) entier

Variable locale : deep\_left : entier, deep\_right : entier|

---------------------------------------

Debut

| Si tree = NULL

| retournre NULL;

| Sinon

| deep\_left <- Depth\_tree(tree.gauche):

| deep\_right <- Depth\_right(tree\_right);

| Si droite <= gauche

| retourner deep\_left + 1 ;

| Sinon

| retourner deep\_right + 1 ;

|

|

|

|

|

| Fin Si

| Fin Si

| Fin

**Rotation d’un arbre binaire :**  E

B F

A D

C

**Equilibrer cet arbre :**

Balance\_tree(tree : NODE)

----------------------------------------------

Debut

| Rotation\_tree\_left(B);

| Rotation\_tree\_right(E);

Fin

*Ce qui donne :*

E E D

B F Rotation gauche -> D F Rotation droite -> B E

sur B sur E

A D B A C F

C A

C

**Créer un arbre parfait :**

Create\_perfect\_tree(n : entier, e : entier) entier

Variable locale : tree : NODE

--------------------------------------------

Debut

| tree = create\_tree(e);

| Si n > 0

| tree -> tree.left <- Create\_perfect\_tree(n - 1, 2 \* e);

| tree -> tree.right <- Create\_perfect\_tree(n – 1, 2 \* e + 1);

| Fin si

| retourner tree;

Fin