FACULTÉ DES SCIENCES ET TECHNIQUES
DE MOHAMMEDIA
UNIVERSITÉ HASSAN II DE CASABLANCA

Ingénierie Logicielle et Intégration des Systèmes Informatiques

Filière d'ingénieur:

Ingénierie Logicielle et Intégration des Systèmes Informatiques

Module : Structure des données

Fonction de manipulation des arbres binaires

Réalisé par :

Jalal Eddine OUTGOUGA

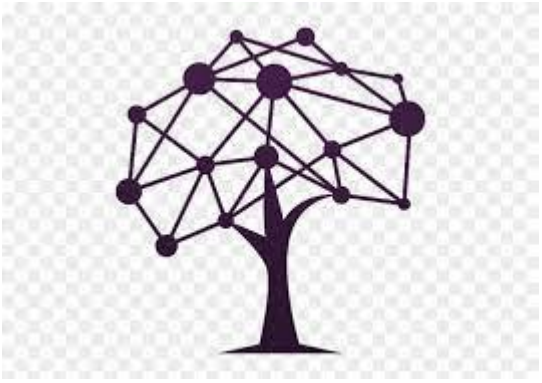
Abdelmajid EZADDI

Encadré par :

Pr Abdelkarim BEKKHOUCHA

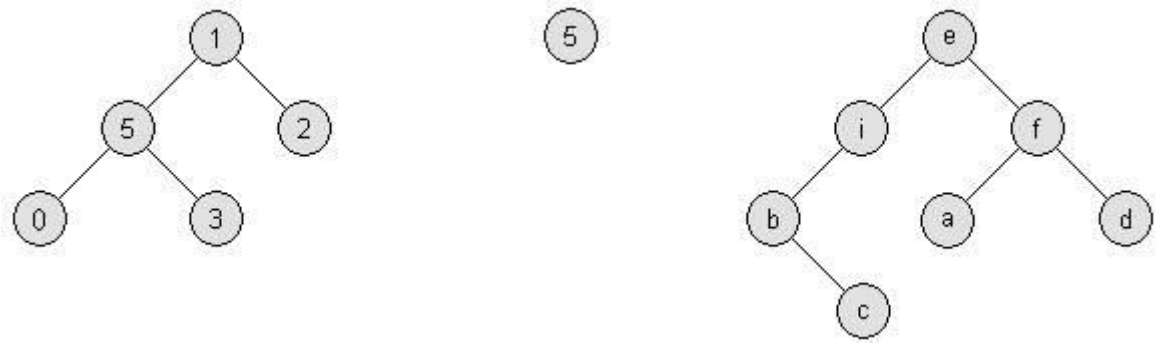
Année universitaire : 2022-2023

Table de Matière



I-Introduction :

Dans ce compte rendu nous allons présenter quelques fonctions de manipulation des arbres à l'aide des tableaux, et nous allons focaliser sur les arbres binaires. Mais d'abord c'est quoi un arbre en informatique ? En informatique, un arbre est une structure de données non linaire qui peut se représenter sous la forme d'une hiérarchie dont chaque élément est appelé nœud, le nœud initial étant appelé racine. Dans un arbre, chaque élément possède 0 à n éléments fils au niveau inférieur, les nœuds qui possèdent 0 fils sont appelés des feuilles. Un arbre binaire est un arbre dont chaque élément possède aux plus deux fils, habituellement appelés gauche et droit. Un arbre binaire est ordonné horizontalement si la clé de tout nœud non feuille est supérieur à toutes celles de son sous arbre gauche et inferieur a toutes celles de son sous arbre droit.



Les cercles portent le nom de **noeud** et celui qui se situe au haut de l'arbre se nomme **racine**. De plus, chaque nœud peut posséder un **fils gauche** et un **fils droit.** Un nœud ne possédant aucun fils se nomme **feuille**. Finalement, tous les nœuds possèdent **un parent** (sauf la racine).

On nomme **profondeur d'un arbre** le nombre maximal de « descentes » pouvant être effectuées à partir de la racine. Par exemple, le troisième arbre binaire de la figure 1 possède une profondeur de 3. Suivant ce raisonnement, un arbre ne possédant qu'un seul ou aucun nœud est de profondeur 0.

II-Analyse :

Comment conserver un arbre binaire de façon statique ?

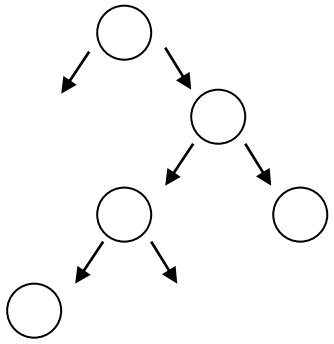
Posons que la première case du tableau porte l'indice 0. Alors, voici comment conserver les données :

- La première case (indice 0) du tableau constitue la racine de l'arbre,
- Pour toute case portant l'indice i, son fils gauche sera situé à l'indice (2i+1) et son fils droit à l'indice (2i+2).
-Une case vide signifie un arbre vide (l'absence de nœud).

Évidemment, la notion de case vide n'existe pas vraiment au tableau alors il faudra utiliser une stratégie pour savoir si une case contient une donnée. Une façon simple de résoudre ce problème consiste à définir une valeur représentant l'absence de donnée (par exemple, 0) et de remplir le tableau de cette valeur au départ.

Exemple :

1	2	3	4	5	6	5	3	3	8
1	0	1	0	0	1	1	-0	0	1



Nous allons adopter la structure suivante :

Alors notre structure va contenir deux tableaux de la même taille, un contient les valeurs et l'autre pour savoir si la case est vide ou non comme on a vu dans l'exemple précèdent.

```c
#include <stdio.h>
#include <stdlib.h>
#define Max 50
// Définition de la structure
typedef struct
{
    int Infos[Max]; // table des info
    int Exist[Max]; // table pour savoir si la case est vide ou non
} TArbre;

```

III- Fonctions de manipulation :

```c
/*
Nom Fonction :Init_Arbre
Entrée : Rien(Procedure)
Sortie : structure d'arbre static initialiser
Description : la fonction initialise notre arbre
*/
TArbre *Init_Arbre()
{
    int i;//indice pour parcourir le tableau
    TArbre *TA;//La structure a retourner
    TA = (TArbre *)malloc(sizeof(TArbre));//Allocation de la mémoire
    if (!TA)
    {
        //Verifier l'allocation est ce qu'il a été bien effectuer
        printf("\nErreur d'allocation ");
        exit(0);
    }
    //Initialiser tous les cellule de  tableaux à 0
    for (i = 0; i < Max; i++)
        TA->Exist[i] = 0;

    return ((TArbre *)TA);//Retourne l'arbre
}
```

```c
/*
Nom Fonction :arbre_est_vide
Entrée : Structure d'Arbre
Sortie : 1 si l'arbre est vide,sinon 0
Description : verifier si l'arbre est vide
*/
int arbre_est_vide(TArbre TA)
{
    return ((int)TA.Exist[0] == 0);
}
```

```c
/*
Nom Fonction :Insert_Arbre
Entrée : L'arbre et la valzue à inserer
Sortie : 0 en cas d'echec d'insertion, sinon 1
Description : la fonction d'insertion d'un élément dans l'arbre
*/
int Insert_Arbre(TArbre *TA, int value)
{
    int indice, i = 0;

    // Si l'arbre est vide on insere dans la premiere case
    if (arbre_est_vide(*TA))
    {
        TA->Infos[0] = value;
        TA->Exist[0] = 1;
        return ((int)1);
    }

    // L'arbre n'est pas vide
    // tanque l'indice inferieur à la taille du tableau
    while ((i < Max))
    {
        // si la case du fils droit est vide on fait l'insertion
        if (TA->Exist[2 * i + 1] == 0)
        {
            TA->Infos[2 * i + 1] = value;
            TA->Exist[2 * i + 1] = 1;
            return ((int)1);
        }

        // si la case du fils gauche est vide on fait l'insertion
        if (TA->Exist[2 * i + 2] == 0)
        {
            TA->Infos[2 * i + 2] = value;
            TA->Exist[2 * i + 2] = 1;
            return ((int)1);
        }

        // charche la sous arbre convenable pour l'insertion

        if (TA->Infos[2 * i + 1] > value)
            i = (2 * i) + 1;
        else
            i = (2 * i) + 2;
    }

    return ((int)0); // L'insertion n'est pas effectuer
}
```

```c
/*
Nom Fonction :Recherhce_Arbre
Entrée : L'arbre et la valeur à rechercher
Sortie : 0 si la valeur n'existe pas, sinon 1
Description : la fonction verifier l'existance d'une valeur dans l'arbre
*/
int Recherhce_Arbre(TArbre *TA, int value)
{
    int i;
    for (i = 0; i < Max; i++)
    {
        if (TA->Infos[i] = value)
            return ((int)1);//variable exixte
    }
    return ((int)0);//varibale n'existe pas
}
```

```c
/*
Nom Fonction :Vider_Arbre
Entrée : L'arbre et la valeur à vider
Sortie : 0 si l'arbre est déja vide, sinon 1
Description : la fonction pour détruire l'arbre
*/
int Vider_Arbre(TArbre *TA)
{
    int i; // variable pour parcourir le tableau
    // cas d'un arbre vide
    if (arbre_est_vide(*TA))
        return ((int)0);
    // mettre tous les case à 0
    for (i = 0; i < Max; i++)
        TA->Exist[i] = 0;
    // retourne 1
    return ((int)1)
}

```

```c
/*
Nom Fonction :Supp_Val_Arbre
Entrée : L'arbre et la valeur à supprimer
Sortie : retourne 1
Description : la fonction de suppression d'un élément dans l'arbre
*/
int Supp_Val_Arbre(TArbre *TA, int value)
{
    int i, j;
    for (i = 0; i < Max; i++)
    {
        if (TA->Infos[i] == value && TA->Exist[i] != 0)
        {
            //Si le noeud n'a pas des fils
            if ((TA->Exist[2 * i + 1] == 0) && (TA->Exist[2 * i + 1] == 0))
            {
                TA->Exist[i] = 0;
            }
            else
            {
                //Si le noeud a un fils gauche
                if (TA->Exist[2 * i + 1] == 1)
                {
                    j = i;
                    while (TA->Exist[2 * j + 1])
                    {
                        j = 2 * j + 1;
                    }
                }
                else
                {
                    //Si le noeud a un fils  droit
                    j = i;
                    while (TA->Exist[2 * j + 2] == 1)
                    {
                        j = 2 * j + 2;
                    }
                }

                TA->Infos[i] = TA->Infos[j];
                TA->Exist[j] = 0;
            }
        }
    }

    return ((int)1);
}

```

```c
void Prefixe(TArbre *TA,int i)
{
    if(TA->Exist[i]!=0)//Condition d'arret
    {
        printf("%d\t",TA->Infos[i]);//Affichge
        Prefixe(TA,2*i+1);//fils gauche
        Prefixe(TA,2*i+2);//fils droit
    }
}

void Postfixe(TArbre *TA,int i)
{
    if(TA->Exist[i]!=0)//Condition d'arret
    {
        Postfixe(TA,2*i+1);//fils gauche
        Postfixe(TA,2*i+2);//fils droit
        printf("%d\t",TA->Infos[i]);//Affichge

    }
}

void Infixe(TArbre *TA,int i)
{
    if(TA->Exist[i]!=0)//Condition d'arret
    {
        Infixe(TA,2*i+1);//fils gauche
        printf("%d\t",TA->Infos[i]);//Affichge
        Infixe(TA,2*i+2);//fils droit
}
```

IV- Conclusion :

Nous avons vu des fonctions de manipulation des arbres statique afin de comprendre leurs fonctionnements, mais cette utilisation est limite il vaut l'utiliser d'une manière dynamique.

I. Introduction :

Dans ce compte rendu nous allons présenter quelques fonctions de manipulation des arbres binaires, et nous allons focaliser sur les arbres binaires ordonnés horizontalement et les arbres binaires quelconques. Mais d'abord c'est quoi un arbre en informatique ?

En informatique, **un arbre** est une structure de données non linaire qui peut se représenter sous la forme d'une hiérarchie dont chaque élément est appelé nœud, le nœud initial étant appelé racine. Dans un arbre, chaque élément possède 0 à n éléments fils au niveau inférieur, les nœuds qui possèdent 0 fils sont appelés des feuilles. (Figure 1)
Un arbre binaire est un arbre dont chaque élément possède aux plus deux fils, habituellement appelés gauche et droit. (Figure 2)
Un arbre binaire est **ordonné horizontalement** si la clé de tout nœud non feuille est supérieur à toutes celles de son sous arbre gauche et inferieur a toutes celles de son sous arbre droit. (Figure 3)

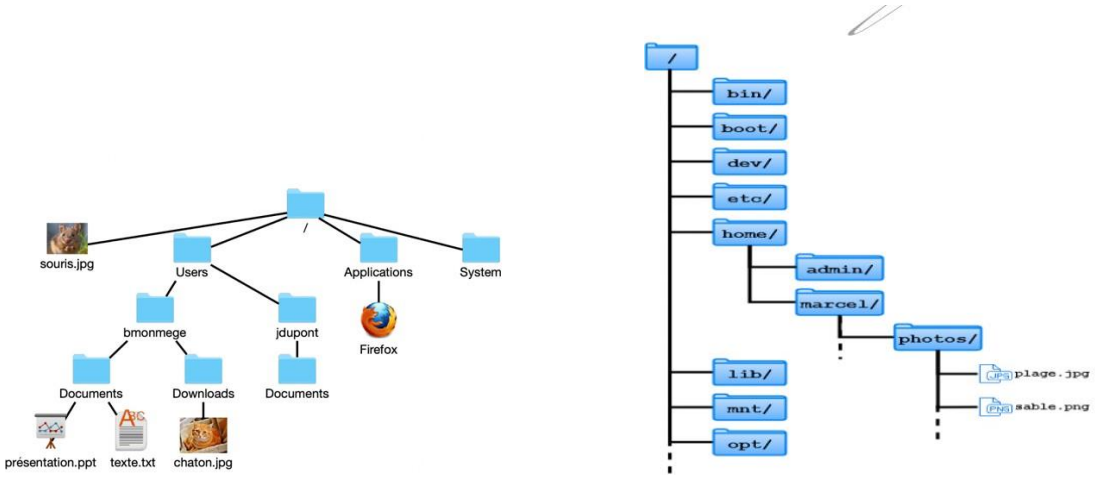


Figure 1 – Arborescence de fichiers

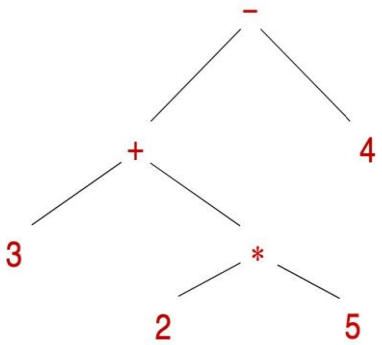


Figure 2 – Expressions mathématiques : 3 + 2*5 - 4

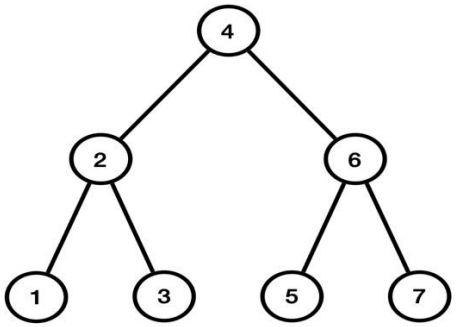


Figure 3 – arbre binaire ordonné horizontal des entiers

II. Analyse :

1- Définition de la structure d'un arbre binaire :

```
#include  <stdio.h>
#include <stdlib.h>
//structure  d'un  arbre  à  l'aide  des  pointeurs
typedef struct Ned
{
    int info;//étiquette
    struct  Ned  *  left;//pointeur  sur  fils  gauche
    struct Ned * right;//pointeur sur fils droit
}Noeud;//structure noeud
```

2- Traitement sur les arbres binaires :

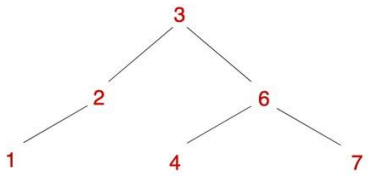
Le traitement sera effectué sur des arbres binaires dont leurs étiquettes sont des entiers pour simplifier le traitement :

- Création et initialisation d'un nœud

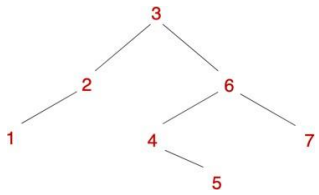
  La création se fait à l'aide d'une fonction similaire a celle des listes, où on fait une allocation de la mémoire d'un pointeur de taille Nœud, on vérifie si l'allocation n'a pas échoué puis on initialise les champs : info par une valeur passée en argument, les pointeurs du fils gauche et droite par NULL et à la fin on retourne le nœud créé.

- Insertion d'un nœud dans un arbre

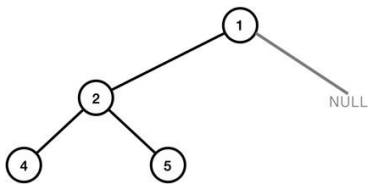
  Pour **un arbre binaire ordonné horizontalement** l'insertion se fait d'une manière simple. Si l'arbre ou le sous arbre est vide le nœud devient la racine, si non si l'étiquette du nœud est plus petite que celle de la racine l'insertion se fait dans le sous arbre gauche si non dans le sous arbre droit.

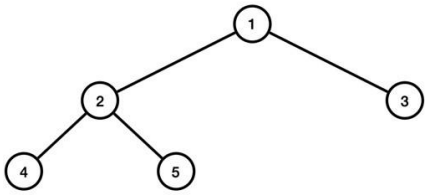


On insère la valeur 5



Pour **un arbre binaire quelconque** et comme le nom l'indique, y'a une infinité de méthodes d'insertion. Dont nous allons introduire une, elle consiste à insérer dans le champ du fils NULL et si le nœud a deux fils au préalable on refait la procédure dans le sous arbre gauche ou le sous arbre droit, selon un caractère passé en argument 'D' : droit ou 'G' : gauche. On insère la valeur 3



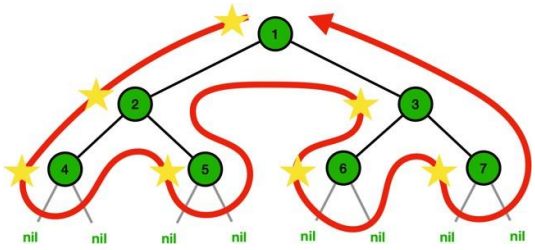


- Affichage de l'arbre

  Il y'a 3 façons générales d'affichage d'un arbre binaire à l'aide des parcours suivant :

  Préfixé :

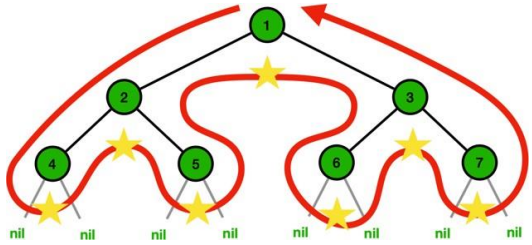
  On affiche la racine, puis le sous arbre gauche, puis le sous arbre droit.



  Affichage : 1 2 4 5 3 6 7.

  Infixé :

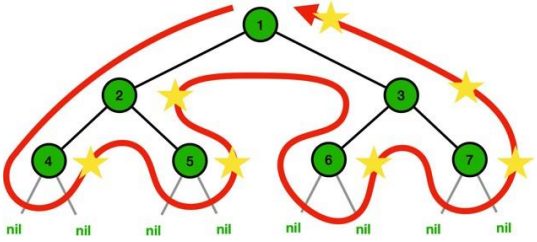
  On affiche le sous arbre gauche, puis la racine, puis le sous arbre droit.



  Affichage : 4 2 5 1 6 3 7.

  Postfixé :

  On affiche le sous arbre gauche, puis le sous arbre droit, puis la racine.



  Affichage : 4 5 2 6 7 3 1.

  On a introduit aussi **l'affichage des feuilles** en écrivant une fonction qui n'affiche que les nœuds qui n'ont pas des fils (*fils_droit=fils_gauche=NULL*). *Ex : 4 5 6 7.*

  Affichage schématique horizontale : on utilise le parcours infixe pour réaliser ce type d'affichage pour avoir le sous arbre droit en haut et affiche un nombre de tabulation égale à la hauteur du nœud suivi par la valeur du nœud puis un retour chariot.

```
                18
        17
                        15
                14
                        13
10
                9
        8
                        7
                6
```

  Note : malheureusement on n'a pas pu fait l'affichage vertical.

- Suppression d'un nœud

  Pour **les arbres binaires ordonnés horizontalement** on a appliqué l'algorithme du cours mais en ajoutant notre propre touche.

  Pour **les arbres binaires quelconques** la suppression se fait toujours au niveau des feuilles c.à.d. si le nœud à supprimer est une feuille un le supprime et on retourne la racine de l'arbre sinon on le remplace par une feuille de ses fils puis on supprime la feuille.

- Chercher un nœud

  La recherche se fait de la manière suivante :
  Si le nœud est la racine on retourne 1 sinon on parcourt le sous arbre gauche puis droit récursivement, si le nœud n'existe pas on retourne 0.

- Déterminer le nœud minimal

  Initialiser une variable **min** par la racine puis parcourir le sous arbre gauche et comparer les nœuds avec le min et remplacer au cas où on trouve une valeur **inférieure**, ensuite refaire le même traitement dans le sous arbre droit d'une manière récursive dans les deux cas, en fin retourner le min.

- Déterminer le nœud maximal

  Initialiser une variable **max** par la racine puis parcourir le sous arbre gauche et comparer les nœuds avec le max et remplacer au cas où on trouve une valeur **supérieure**, ensuite refaire le même traitement dans le sous arbre droit d'une manière récursive dans les deux cas, en fin retourner le max. *même principe du min.*

- Calcule de la hauteur d'un arbre

  Pour calculer **la hauteur** d'un arbre, on commence par le calcul de la hauteur du sous arbre gauche en incrémentant une variable récursivement ensuite on refait le traitement pour le sous arbre droit puis on compare les deux hauteurs et à la fin on retourne la plus grande.

- Vider un arbre

  Pour **vider un arbre binaire**, on parcourt l'arbre en supprimant la racine tant que la taille de l'arbre est différente de 0, autrement dit tant que l'arbre est non vide.

III. Fonctions de manipulation :

1. Les fonctions communes des arbres binaires :

Voici quelques fonctions de manipulation des arbres binaires (Ordonnés/quelconques).

```
/* 1
Nom   Fonction   :   creer_noeud
Entree        : Un entier (val)
Sortie        : Un nœud (Nd)
Description   : la fonction crée un nouveau nœud
*/
Noeud*creer_noeud(int val)
{
    Noeud   *Nd;//declaration   du   nouveua   noeud
    Nd=(Noeud*)malloc(sizeof(Noeud));
    if(!Nd)//echec d'allocation
    {
        printf("erreur d'allocation");
        exit(-1);//sortir du programme
    }//fin if
    //initialisation du nœud crée
    Nd->info=val;
    Nd->left=Nd->right=NULL;
    return ((Noeud*)Nd);
}//fin fonction
/* 3
Nom Fonction  : Prefix_aff
Entree        : racine d'un arbre (Arbre)
Sortie        :
Description   : la fonction permet l'affichage d'un arbre
                a l'aide du parcours préfixe
*/
void Prefix_aff(Noeud *Arbre)
{
    if(Arbre)
    {
        printf("%d\n",Arbre->info);
        Prefix_aff(Arbre->left);
        Prefix_aff(Arbre->right);
    }}//fin fonction
```

```c
/* 4
Nom Fonction  : infix_aff
Entrée        : racine d'un arbre (Arbre)
Sortie        :
Description   : la fonction permet l'affichage d'un arbre
                a l'aide du parcours infixe
*/
void infix_aff(Noeud *Arbre)
{
    if(Arbre)
    {
        infix_aff(Arbre->left);
        printf("%d\t",Arbre->info);
        infix_aff(Arbre->right);
    }
}//fin fonction
/* 5
Nom Fonction  : Postfix_aff
Entrée        : racine d'un arbre (Arbre)
Sortie        :
Description   : la fonction permet l'affichage d'un arbre
                a l'aide du parcours Postfixe
*/
void postfix_aff(Noeud*Arbre )
{
    if(Arbre)
    {
        postfix_aff(Arbre->left);
        postfix_aff(Arbre->right);
        printf("%d\t",Arbre->info);
    }
}//fin fonction
/* 6
Nom Fonction  : taille_Arbre
Entrée        :  racine  d'un  arbre  (Arbre)
Sortie        : nombre des nœuds de l'arbre
Description   : la fonction retourne le nombre des nœuds
                d'un arbre
```

```
*/
int taille_Arbre(Noeud*Arbre)
{
    if(!Arbre) return((int)0);//arbre vide
    //retourner le nombre des fils gauche et droit et la racine
    return ((int)(1+taille_Arbre(Arbre->left)+taille_Arbre(Arbre-
>right)));
}//fin fonction

/* 7
Nom Fonction  : hauteur_Arbre
Entree        : racine d'un arbre (Arbre)
Sortie        : hauteur d'un arbre
Description   : la fonction retourne la hauteur d'un arbre
                en calculant la hauteur max gauche et max droit et
                retourne la plus grande entre eux
*/
int hauteur_Arbre(Noeud*Arbre)
{
    int  hg,//pour  stocker  la  hauteur  max  gauche
        hd;//pour stocker la hauteur max droit
    if(!Arbre) return((int)0);//arbre vide
    //retourner le nombre des fils gauche et droit et la racine
    else{
    hg=hauteur_Arbre(Arbre->left);//hauteur gauche max
    hd=hauteur_Arbre(Arbre->right);//hauteur droite max
    //retourner  la  plus  grande  entre  les  deux
    return((hd<hg)?((int)hg+1):((int)hd+1));
    }
}//fin fonction
```

```
/* 8
Nom Fonction  : max_arbre
Entree        : racine d'un arbre (Arbre)
Sortie        : noued le plus grand
Description   : la fonction permet de retourner le
                nœud le plus grand
*/
 int max_arbre(Noeud* Arbre)
{
    int max = Arbre->info;// max prend la valeur du racine
    if (Arbre->left)
    {   //determiner le max du sous arbre gauche
        int gauche = max_arbre (Arbre->left);
        //comparer et affecter le max avec le max gauche
        max = (max > gauche ) ? max : gauche;
    }
    //refait la même procédure avec le sous arbre droit
    if (Arbre->right)
    {
        int droit = max_arbre (Arbre->right);
        max = (max > droit ) ? max : droit;
    }
    return max;
}
 /* 9
Nom Fonction  : min_arbre
Entree        : racine d'un arbre (Arbre)
Sortie        : noued le plus petit
Description   : la fonction permet de retouner le
                noued le plus petit
*/
 int min_arbre(Noeud* Arbre)
{
    int min = Arbre->info;// min prend la valeur du racine
    if (Arbre->left)
    {   //determiner le min du sous arbre gauche
        int gauche = min_arbre (Arbre->left);
        //comparer et affecter le min avec le min gauche
```

```
        min = (min< gauche ) ? min : gauche;
    }
    //refait la meme procedure avec le sous arbre droit
    if (Arbre->right)
    {
        int droit = min_arbre (Arbre->right);
        min = (min < droit ) ? min : droit;
    }
    return min;
}
/* 12
Nom Fonction  : aff_feuilles
Entrée        : racine d'un arbre (Arbre)
Sortie        :
Description   : la fonction permet l'affichage de tous les
                feuilles d'un arbre
*/
 void aff_feuilles(Noeud*Arbre )
{
    if(Arbre)
    {
        aff_feuilles(Arbre->left);
        //si le noeud n'a pas des fils c'est une feuille
        if((!Arbre->left)&&(!Arbre->right))
            //afficher la feuille
            printf(" \t\t\t%d\n",Arbre->info);
        aff_feuilles(Arbre->right);

    }
}//fin fonction
 /* 13
Nom Fonction  : Aff_Arbre_horiz
Entree        : racine d'un arbre (Arbre)
                Niveau de la racine (0)
Sortie        :
Description   : la fonction permet l'affichage de l'arbre
                d'une manière horizontale
*/
```

```c
void Aff_Arbre_horiz(Noeud *Arbre,int Niv)
{
    int esp; //Pour l'affichage des espaces
     if (Arbre) //Condition d'arret
     {
        //Affichage    des    fils    droits
         Aff_Arbre_horiz(Arbre->right,Niv+1);
         printf("\n");
         for (esp = 0; esp < Niv; esp++)
            printf("\t");
         printf("%d",Arbre->info);
         //Affichage   des   fils   gauches
         Aff_Arbre_horiz(Arbre->left,Niv+1);
     }
}//fin fonction
/* 14
Nom Fonction  : rechercher
Entrée        : racine d'un arbre (Arbre)
                entier valeur cherche (val)
Sortie        : un entier test=(0 ou 1)
Description   : la fonction retourne 1 si une valeur passe
                en argument existe dans l'arbre et 0 sinon
*/
int rechercher(Noeud*Arbre,int val)
{
    int test=0;//test initialiser par 0
    if(Arbre)
    {   //si la valeur est trouve test=0+1 sinon reste égal 0
        if(Arbre->info==val) return 1;
        //chercher dans le sous arbre gauche
        if (Arbre->left) test+= rechercher(Arbre->left,val);
        // chercher dans le sous arbre droit
        if(Arbre->right) test+= rechercher(Arbre->right,val);
    }
    //retourner test
    return test;
}
```

```
/* 16
Nom Fonction  : affichage
Entree        : racine d'un arbre (Arbre)
Sortie        :
Description   : la fonction permet l'affichage de l'arbre
                de plusieurs manières
*/
void affichage(Noeud*arbre)
{
    int aff;
            printf("\nQuelle type d'affichage voulez vous ?\n");
            printf("\t\t1 affichage prefixe \n");
            printf("\t\t2 affichage infixe \n");
            printf("\t\t3 affichage postfixe \n");
            printf("\t\t4 affichage schematise\n");
            printf("\t\t5 affichage des feuilles de l'arbre\n");
            scanf("%d",&aff);
            if(!arbre) printf("l'arbre est vide\n");
            else
        {//cas où l'arbre est no vide
            switch (aff)
            {
            case 1:
                printf(" ------\n");
                Prefix_aff(arbre);
                break;
            case 2:
                printf(" ------\n");
                infix_aff(arbre);
                break;
            case 3:
                printf(" ------\n");
                postfix_aff(arbre);
                break;
            case 4: printf(" ------\n");
                Aff_Arbre_horiz(arbre,0);
                break;
            case 5:
```

```
                printf("\t\tvoici tous les feuilles de l'arbre\n");
                aff_feuilles(arbre);
                break;
            default:
                printf("choix invalide\n");
                break;
            }
        }//fin sinon
}
```

2. Les fonctions des arbres binaires ordonnés horizontalement :

```
/* 2 **
Nom Fonction  :  Inserer_Arbre_horiz
Entree        : Un entier (val)
                racine d'un arbre (Arbre)
Sortie        :  racine du nouveau arbre après l'insertion
Description   : la fonction permet l'insertion d'un noeud dans
                dans un arbre donne
*/
Noeud*Inserer_Arbre_horiz(Noeud*Arbre,int val)
{
    Noeud*Nd;
    if(!Arbre)//arbre vide ou (sous-arbre vide)
    {
        Nd=creer_noeud(val);
        return((Noeud*)Nd);
    }
    if(Arbre->info>val)//val inferieur a l'étiquette du noeud
        //insertion a gauche
        Arbre->left=Inserer_Arbre_horiz(Arbre->left,val);
    else// sinon
        //insertion a droit
        Arbre->right=Inserer_Arbre_horiz(Arbre->right,val);
    return((Noeud*)Arbre);
}//fin fonction
```

```
/* 10 **
Nom Fonction  : eliminer
Entree        : Racine d'un arbre (ou sous arbre)(Arbre)
                entier ver pour le fils que arbre doit
                prendre
Sortie        : racine de l'arbre (ou sous arbre) après le changement
Description   : la fonction retourne le racine après le chainage et
                l'élimination de du noeud d'un arbre (ou sous arbre)
*/
Noeud* eliminer(Noeud* Arbre,int ver)
{
    Noeud* temp=Arbre;//pointeur de suppression
    //ver=1 pas de fils droit, 0 pas de fils gauche
    ver==1?(Arbre=Arbre->left):(Arbre=Arbre->right);
    free(temp);//supprimer le nœud voulu
    return ((Noeud*)Arbre);
}
/* 11 **
Nom Fonction  : supp_Arbre
Entree        : racine d'un arbre (Arbre)
                valeur a supprimer (val)
Sortie        : arbre après suppression
Description   : la fonction reçoit un élément et le supprime
                de l'arbre dont son racine est passe en argument
                et retourne l'arbre après la suppression
*/
Noeud* supp_Arbre(Noeud*Arbre,int val)
{
    if (!Arbre) return NULL;
    //si l'element est dans l'arbre gauche
    else if(val<(Arbre->info)) Arbre->left=supp_Arbre(Arbre->left,val);
    //si l'element est dans l'arbre droit
    else    if(val>(Arbre->info))Arbre->right=supp_Arbre(Arbre->right,val);
    else //val==Arbre->info
    {
        // s'il n y a pas de fils droit
        if(!(Arbre->right)) Arbre=eliminer(Arbre,1);
        // s'il n y a pas de fils gauche
```

```
        else if(!(Arbre->left)) Arbre=eliminer(Arbre,0);
        //si il y a les deux (droit et gauche)
        else
            Arbre->right=supp_Arbre(Arbre->right,((Arbre-
>info)=min_arbre(Arbre->right)));
    }
    return ((Noeud*)Arbre);
}//fin fonction

     /* 15 **
Nom Fonction  : vider_arbre_ordon
Entree        : racine d'un arbre (Arbre)
Sortie        : arbre vide
Description   : la fonction supprime tous les noeuds
                d'un arbre binaire ordonne horizontalement
*/
Noeud* vider_arbre_ordon(Noeud* Arbre)
{
    //tanque la taille de l'arbre est diffèrent de 0
    while(taille_Arbre(Arbre))
    {   //supprimer        la        racine
       Arbre=supp_Arbre(Arbre,Arbre->info);
    }
    //retourner l'arbre vide
    return Arbre;
}
```

Conclusion :

Dans ce TP nous avons traite des fonctions de manipulations des arbres a l'aide des pointeurs, malgré que nous ayons des problèmes dans l'affichage verticale, mais nous avons fait des tentatives qui sont malheureusement échoué.