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 à l'aide des tableaux


**Réalisé par :**

**Jalal Eddine OUTGOUGA**

**Abdelmajid EZADDI**

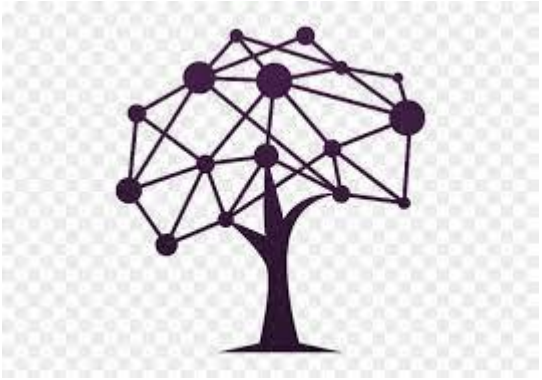
**Encadré par :**

**Pr Abdelkarim BEKKHOUCHA**


*Année universitaire : 2022-2023*

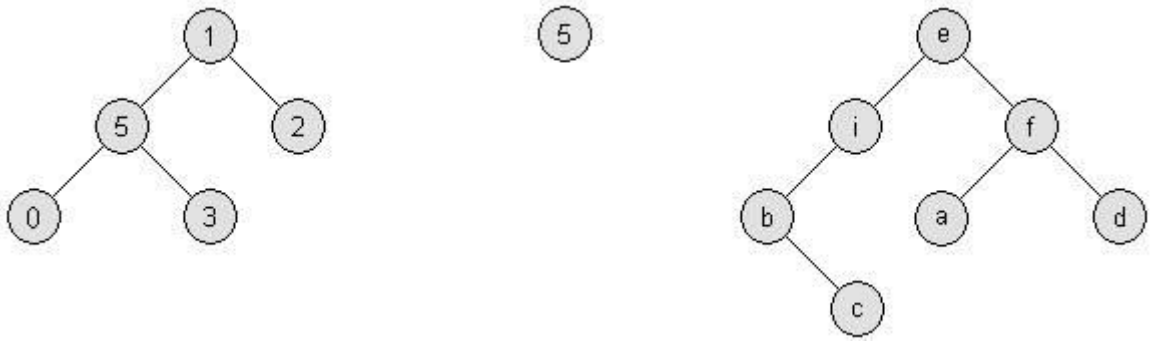
# Table de Matière





## 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 :

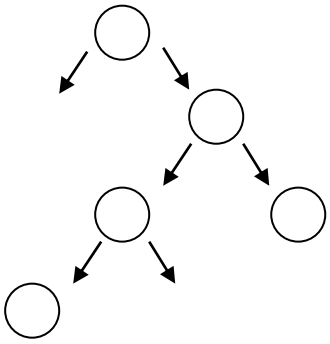
- La première case (indice 0) du tableau constitue la racine de l'arbre,
- Pour toute case portant l'indice i, son fils gauche sera situé à l'indice (2i+1) et son fils droit à l'indice (2i+2).

-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);
}

```

## 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.