Université Mohamed Khider-Biskra
Faculté des Sciences Exactes et des Sciences de la Nature et de la Vie
Département d'informatique

$2^{ème}$ année LMD — TD ALGO
Série d'exercices N°5 — Janvier 2022

# Arbres (Corrigé)

**Exercice 1** *Arbres m-aires*

1. Le maximum dans l'arbre.

```
Fonction Max( nœud : Pointeur(Tnœud)) : entier;
Var  i,M1,M : entier ;
Début
    Si (nœud = Nil) Alors
        Ecrire('Arbre vide')
    Sinon
        M ← Valeur(nœud) ;
        Pour i de 1 à NbFils faire
            Si (Fils_i(nœud) ≠ Nil) Alors
                M1 ← Max(Fils_i(nœud)) ;
                Si (M1 > M) Alors
                    M ← M1 ;
                Fin Si;
            Fin Si;
        Fin Pour;
        Max ← M ;
    Fin Si;
Fin;
```

2. La taille de l'arbre.

```
Fonction Taille( nœud : Pointeur(Tnœud)) : entier;
Var  i,N : entier ;
Début
    Si (nœud = Nil) Alors
        Taille ← 0 ;
    Sinon
        N ← 0 ;
        Pour i de 1 à NbFils faire
            N ← N + Taille(Fils_i(nœud)) ;
        Fin Pour;
        Taille ← N ;
    Fin Si;
Fin;
```

3. Le nombre de feuilles de l'arbre

```
Fonction NbFeilles( nœud : Pointeur(Tnœud)) : entier;
Var  i, j,N : entier ;
Début
    Si (nœud = Nil) Alors
        NbFeuilles ← 0 ;
    Sinon
        i ← 1 ;
        Tant que (i<= NbFils et Fils_i(nœud) = Nil) faire
            i ← i+1 ;
        Fin TQ;
        Si (i > NbFils) Alors
            NbFeuilles ← 1 ;
        Sinon
            N ← 0 ;
            Pour j de i à NbFils faire
                N ← N + NbFeuilles(Fils_j(nœud)) ;
            Fin Pour;
            NbFeuilles ← N ;
        Fin Si;
    Fin Si;
Fin;
```

4. La hauteur de l'arbre

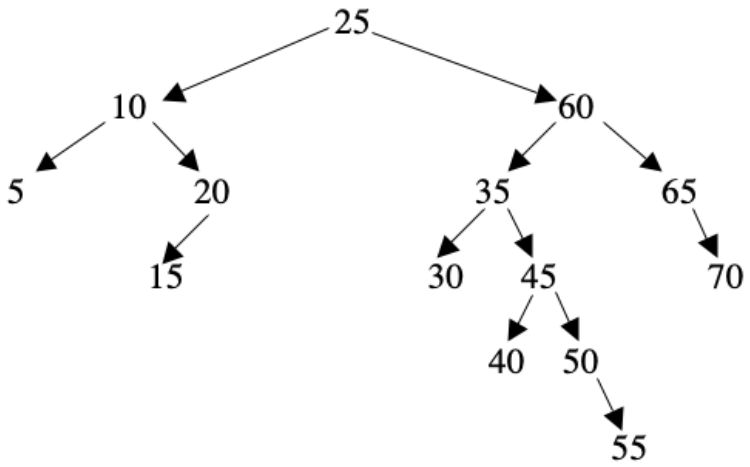
```
Fonction Hauteur( nœud : Pointeur(Tnœud)) : entier;
Var  i,H1,H : entier ;
Début
    Si (nœud = Nil) Alors
        Hauteur ← 0 ;
    Sinon
        H ← 0 ;
        Pour i de 1 à NbFils faire
            H1 ← Hauteur(Fils_j(nœud)) ;
            Si (H1 > H) Alors
                H ← H1 ;
            Fin Si;
        Fin Pour;
        Hauteur ← 1 + H ;
    Fin Si;

Fin;
```

**Exercice 2** *Arbres de recherche binaires*

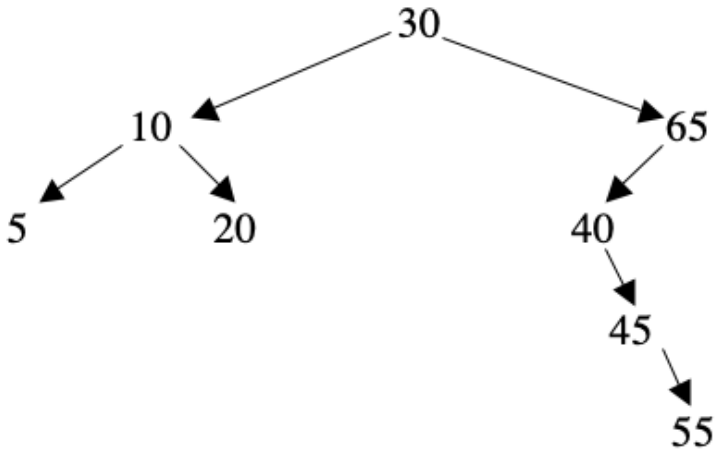
1. Construire un arbre de recherche binaire d'entiers à partir des clés ordonnées suivantes :

**25 60 35 10 5 20 65 45 70 40 50 55 30 15**



2. Supprimer de l'arbre obtenu, et dans l'ordre, les éléments suivants :

**15 70 50 35 60 25**




3. Min(R) : permettant de retourner le minimum dans l'ARB de racine R.

```
Fonction Min( R : Pointeur(Tnœud)) : entier;
Var  N : Pointeur(Tnœud) ;
Début
    N ← R ;
    Si (N = Nil) Alors
        Ecrire('Arbre vide') ;
    Sinon
        Tant que (FG(N) ≠ Nil) faire
            N ← FG(N) ;
        Fin TQ;
        Min ← Valeur(N) ;
    Fin Si;
Fin;
```

4. Père(R, x) : permettant de retourner le père du nœud contenant un entier x dans l'ARB de racine R.

```
Fonction Pere( R : Pointeur(Tnœud), x : entier) : Pointeur(Tnœud);
Var  N, PN : Pointeur(Tnœud) ;
Début
    N ← R ;
    PN ← Nil ;
    Tant que (N ≠ Nil et Valeur(N) ≠ x) faire
        PN ← N ;
        Si (Valeur(N) > x) Alors
            N ← FG(N) ;
        Sinon
            N ← FD(N) ;
        Fin Si;
    Fin TQ;
    Si (N ≠ Nil) Alors
        Pere ← PN ;
    Sinon
        Pere ← Nil ;
    Fin Si;
Fin;
```

5. Successeur(R,x) : permettant de retourner l'élément immédiatement supérieur à x dans l'ARB de racine R.

```
Fonction Successeur( R : Pointeur(Tnœud), x : entier) : Pointeur(Tnœud);
Var  N, PN, DNG : Pointeur(Tnœud) ;
Début
    N ← R ;
    PN ← Nil ;
    DNG ← Nil ; // Dernier noeud à gauche
    Tant que (N ≠ Nil et Valeur(N) ≠ x) faire
        PN ← N ;
        Si (Valeur(N) > x) Alors
            DNG ← N ;
            N ← FG(N) ;
        Sinon
            N ← FD(N) ;
        Fin Si;
    Fin TQ;
    Si (N = Nil) Alors
        Successeur ← Nil ;
    Sinon
        Si (FD(N) = Nil) Alors
            Si (PN=Nil ou N = FG(PN)) Alors
                Successeur ← PN ;
            Sinon
                Successeur ← DNG ;
            Fin Si;
        Sinon
            PN ← FD(N) ;
            Tant que (FG(PN) ≠ Nil) faire
                PN ← FG(N) ;
            Fin TQ;
            Successeur ← PN ;
        Fin Si;
    Fin Si;
Fin;
```

6. InsérerARB(R,x) : permettant d'insertion un entier x dans l'ARB de racine R.

```
Procédure InsérerARB(var R : Pointeur(Tnœud), x : entier);
Var N, PN, N1 : Pointeur(Tnœud) ;
Début
    N ← R ;
    PN ← Nil ;
    Tant que (N ≠ Nil et Valeur(N) ≠ x) faire
        PN ← N ;
        Si (Valeur(N) > x) Alors
            N ← FG(N) ;
        Sinon
            N ← FD(N) ;
        Fin Si;
    Fin TQ;
    Si (N ≠ Nil) Alors
        Ecrire(x,'existe déjà !!!!!')
    Sinon
        Allouer(N1) ;
        Aff_val(N1,x) ;
        Aff_FG(N1,Nil) ;
        Aff_FD(N1,Nil) ;
        Si (PN = Nil) Alors
            R ← N1 ;
        Sinon
            Si (Valeur(PN) > x) Alors
                Aff_FG(PN,N1) ;
            Sinon
                Aff_FD(PN,N1) ;
            Fin Si;
        Fin Si;
    Fin Si;
Fin;
```

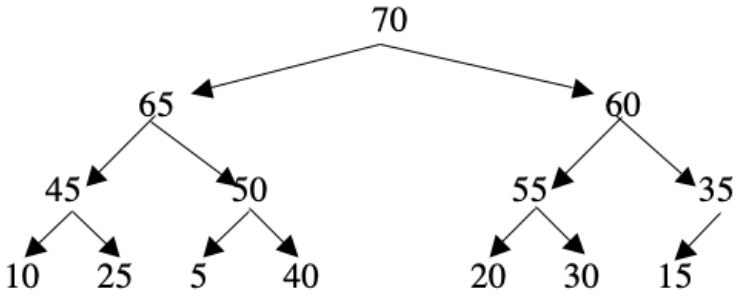
7. SupprimerARB(R,x) : permettant de supprimer un entier x de l'ARB de racine R.

```
Procédure SupprimerARB(var R : Pointeur(Tnœud), x : entier);
Var N, PN, N1 : Pointeur(Tnœud) ;
Début
    N ← R ; PN ← Nil ;
    Tant que (N ≠ Nil et Valeur(N) ≠ x) faire
        PN ← N ;
        Si (Valeur(N) > x) Alors
            N ← FG(N) ;
        Sinon
            N ← FD(N) ;
        Fin Si;
    Fin TQ;
    Si (N = Nil) Alors
        Ecrire(x,'n'existe pas!!!!!')
    Sinon
        Si (FG(N)=Nil ou FD(N) = Nil) Alors
            Si (FG(N) ≠ Nil) Alors
                N1 ← FG(N) ;
            Sinon
                N1 ← FD(N) ;
            Fin Si;
            Si (PN=Nil) Alors
                R ← N1 ;
            Sinon
                Si (Valeur(PN)>x) Alors
                    Aff_FG(PN,N1) ;
                Sinon
                    Aff_FD(PN,N1) ;
                Fin Si;
            Fin Si;
        Sinon
            N1 ← N ; PN ← N ; N ← FD(N) ;
            Tant que (FG(N) ≠ Nil) faire
                PN ← N ;
                N ← FG(N) ;
            Fin TQ;
            Aff_val(N1,Valeur(N)) ;
            Si (PN = N1) Alors
                Aff_FD(PN,FD(N)) ;
            Sinon
                Aff_FG(PN,Nil) ;
            Fin Si;
        Fin Si;
        Libérer(N) ;
    Fin Si;
Fin;
```

**Exercice 3** *Tas*

1. Construire un tas à partir des clés ordonnées suivantes :

   **25 60 35 10 5 20 65 45 70 40 50 55 30 15**




2. Supprimer du tas et dans l'ordre les éléments suivants :

   **15 70 50 35 60 25**

   Impossible, on ne peut pas supprimer directement un élément d ?un tas, on ne peut que retirer celui dans la racine

3. Écrire la procédure InsérerTasStatique(T,x) permettant d'insérer un entier x dans un tas statique d'entiers rangé ans le tableau d'entiers T.

```
Procédure InsérerTasStatique(var Tas : Tableau[1..n] de entier,var NbClés : entier, x : entier);
Var i,Temp : entier ;
Début
    Si (NbClés=N) Alors
        Ecrire(Insertion impossible, le tas est plein !!!!)
    Sinon
        NbClés ← NbClés + 1 ;
        Tas[NbClés] ← x ;
        I ← NbClés / 2 ;
        Tant que (I> 0 et Tas[I]<Tas[I*2]) faire
            Temp ← Tas[I] ;
            Tas[I] ← Tas[I × 2] ;
            Tas[I×2] ← Temp ;
            I ← I / 2
        Fin TQ;

    Fin Si;
Fin;
```

4. Écrire la procédure RetirerTasStatique(T,x) permettant de supprimer un entier x du tas statique d'entiers rangé ans le tableau d'entiers T.

```
Procédure RetirerTasStatique(var Tas : Tableau[1..n] de entier,var NbClés,x : entier);
Var i,Temp : entier ;
Début
    Si (NbClés=0) Alors
        Ecrire(Retrait impossible, le tas est vide !!!!)
    Sinon
        x ← Tas[1] ;
        Tas[1] ← Tas[NbClés] ;
        NbClés ← NbClés - 1 ;
        I ← 1 ;
        Tant que (I< NbClés/2 et (Tas[I]<Tas[I×2] ou Tas[I]<Tas[I×2 + 1] )
        faire
            Si (Tas[I×2]<Tas[I×2 + 1]) Alors
                Temp ← Tas[I] ;
                Tas[I] ← Tas[I × 2 +1] ;
                Tas[I×2 + 1] ← Temp ;
            Fin Si;
            Temp ← Tas[I] ;
            Tas[I] ← Tas[I × 2] ;
            Tas[I×2] ← Temp ;
            I ← I × 2 ;
        Fin TQ;

    Fin Si;
Fin;
```

5. Écrire la procédure InsérerTasDynamique(R,x) permettant d'insérer un entier x dans un tas dynamique d'entiers de racine R.
6. Écrire un algorithme qui permet de trier un tableau en utilisant un tas. Calculer la complexité de cet algorithme.

```
Procédure TrierTableau(var  T : Tableau[1..n] de entier);
Var  i,Temp,NbClés : entier ;
        Tas : Tableau[1..n] de entier ;
Début
    NbClés ← 0 ;
    Pour i de 1 à n faire
      | InsérerTasStatique(Tas,NbClés,T[i]) ;
    Fin Pour;
    Pour i de 1 à n faire
      | RetirerTasStatique(Tas,NbClés,T[i]) ;
    Fin Pour;
Fin;
```

La complexité de cet algorithme est : $O(nLog_2(n))$