Université Mohammed V
Faculté des Sciences, Rabat
Département d'Informatique

STRUCTURES DE DONNEES : DEVOIRS CORRIGES

Licence Fondamentale : Sciences Mathématiques et Informatiques

Préparer par :

Pr. Fouzia OMARY

Année universitaire 2014 / 2015

Table des matières

I. TAD général 4
1. TAD Polynôme 4
a. Enoncée 4
b. Corrigé 4
2. TAD Vecteur 6
c. Enoncée 6
d. Corrigé : 7
II. Listes 9
1. Listes doublement triées 9
e. Enoncée 9
f. Corrigé 10
2. Liste chaînée de chaîne de caractères 11
a. Enoncée 11
b. Corrigé : 12
III. Piles 13
1. Tri avec deux piles 13
a. Enoncée 13
b. Corrigé 13
2. PileMin 15
a. Enoncée 15
b. Corrigé 15
IV. Arbres 17
1. Arbre binaire de Recherche 17
a. Enoncée 17
b. Corrigé 17
2. Tri par tas 18
a. Enoncée 18
b. Corrigé 18
V. Graphes 20
1. Puits universel de graphe orienté 20
a. Enoncé 20
b. Corrigé 20

2. Représentations de graphe orienté 21
a. Enoncé 21
b. Corrigé : 22
VI. Références 25

I. TAD général

1. TAD Polynôme

a. Enoncée

Soit le T.A.D. défini pour représenter les polynômes à coefficients entiers.
Donner le nom poly à **Structure(deg : entier, coef : Tableau(entier) :pointeur)**.

1. Ecrire les algorithmes pour les fonctions de manipulation des polynômes suivantes :
   a. **lirepoly() : poly** qui permet de saisir le degré du polynôme ainsi que ses coefficients.
   b. **afficherpoly(p : poly)** qui permet d'afficher le polynôme.
   c. **coef(p : poly, j : entier) : entier** qui retourne le coefficient correspondant à l'exposant j spécifié comme argument.
   d. **insere(p : poly, c : entier, j : entier) : poly** qui permet d'insérer un terme d'exposant j et de coefficient c spécifiés comme arguments.
   e. **supprime(p : poly, j : entier) : poly** qui permet de supprimer du polynôme p le terme d'exposant j.
   f. **somme (p : poly, q : poly) : poly** qui retourne le polynôme somme des deux polynômes spécifiés comme arguments.
2. Evaluer un polynôme consiste à calculer la valeur du polynôme pour une valeur de x donné. Par exemple, si P(x) = x 3+3x 2+1 et x = 2 alors P(2) = 23+3∗2 2+1 = 21. La méthode de Horner permet d'évaluer un polynôme sans calculer de puissances : l'idée consiste à réécrire le polynôme en utilisant des parenthèses emboîtées, ainsi P(x) = x 3 + 3x 2 + 1 devient P(x) = (((1)x + 3)x + 0)x + 1. Ecrire les algorithmes des fonctions :
   a. **evaluerpoly** qui utilise la fonction de puissance
   b. **evaluerpolyrec** qui utilise la méthode de Horner.

b. Corrigé

```
typedef struct {
      int dg ;
      int *coef ;
}Poly ;
Poly   lirepoly(){
      Poly  P ;
      int i ;
      printf("\n donner le degré du polynôme ") ;
      scanf("%d",&P.dg) ;
      P.coef=(int *)malloc((P.dg+1)*sizeof(int)) ;
      for(i=0 ;i<=P.dg ;i++){
            printf("\n donner le coefecient d'ordre %d du
polynôme ",i) ;
            scanf("%d",&(P.coef)[i]) ;
      }
      return P ;
}
void  afficherpoly(Poly P){
      int i ;
      printf("\n P = ") ;
      for(i=P.dg ;i>=0 ;i--)
            if((P.coef)[i] !=0)
                  if(i==0)
```

```
                    printf("%d\n",(P.coef)[i]) ;
               else
                    printf("%dx%d + ",(P.coef)[i],i) ;
}
int coef(Poly  P,int  j){
     return((P.coef)[j]) ;
}
Poly insere(Poly P,int c, int   j) {
     if(j>P.dg){
          printf("\n erreur") ;
          exit(-1) ;
     }
     else
          (P.coef)[j]=c ;
     return P ;
}
Poly supprime(Poly P, int   j) {
     if(j>P.dg){
          printf("\n erreur") ;
          exit(-1) ;
     }
     else
          (P.coef)[j]=0 ;
     return P ;
}
Poly  somme (Poly P, Poly Q){
     Poly S ;
     int i ;
     S.dg=(P.dg>Q.dg) ? P.dg , Q.dg ;
     S.coef=(int *)malloc(S.dg*sizeof(int)) ;
     if(P.dg>Q.dg) {
          for(i=0 ;i<=Q.dg ;i++)
               (S.coef)[i]=(P.coef)[i]+(Q.coef)[i] ;
          for(i=Q.dg+1 ;i<=P.dg ;i++)
               (S.coef)[i]=(P.coef)[i] ;
     }
     else
          if(P.dg<Q.dg) {
               for(i=0 ;i<=P.dg ;i++)
                    (S.coef)[i]=(P.coef)[i]+(Q.coef)[i] ;
               for(i=P.dg+1 ;i<=Q.dg ;i++)
                    (S.coef)[i]=(Q.coef)[i] ;
          }
          else{
               for(i=0 ;i<=P.dg ;i++)
                    (S.coef)[i]=(P.coef)[i]+(Q.coef)[i] ;
               i=P.dg ;
               while((S.coef)[i]==0){
                    i-- ;
                    S.dg-- ;
               }

          }
}
```

3. pow(x,n) est une fonction de math.h qui permet de calculer x à la puissance n :

```
int evaluerPoly(Poly P, int x){
      int i,p ;
      p=coef(P,i);
      for(i=1 ;i<=P.dg ;i++)
            p+=coef(P,i)*pow(x,i) ;
      return p ;
}
int evaluerPolyHor(Poly P, int x){
      int i,p ;
      if(P.dg==0)
            p=coef(P,0) ;
      else{
            p=0 ;
            for(i=P.dg ;i>0 ;i++)
                  p=(p+coef(P,i))*x ;
            p+=coef(P,0) ;
      }
      return p ;
}
```

2. TAD Vecteur

c. Enoncée

On désire implanter, en langage C, **le type abstrait de données Vecteur**, défini dans le cours. On rappelle, ci-dessous, une spécification minimale du **TAD Vecteur :**

```
Type Vecteur

Utilise Entier, Elément

Opérations

    vect       : Entier → Vecteur

    changer_ième: Vecteur x Entier x Elément → Vecteur

    ième       : Vecteur x Entier → Elément

    taille     : Vecteur → Entier

Préconditions

    vect(i) est_défini_ssi i ≥ 0

    ième(v,i) est_défini_ssi 0 ≤ i < taille(v)

    changer_ième(v,i,e)  est_défini_ssi  0  ≤  i  <
taille(v)

Axiomes

    Soit, i, j : Entier, e : Elément, v : Vecteur

    si 0 ≤ i < taille(v) alors ième(changer_ième(v,i,e),i)=e
```

```
si 0 ≤ i < taille(v) et 0 ≤ j < taille(v) et i ≠ j
        alors ième(changer_ième(v,i,e),j) = ième(v,j)
taille(vect(i)) = i
taille(changer_ième(v,i,e)) = taille(v)
```

Il existe plusieurs méthodes pour représenter les vecteurs. Parmi celles-ci une méthode qui consiste à représenter un vecteur sous la forme d'un tableau. On considère ici, un vecteur comme est une structure regroupant un entier, pour la taille, et un pointeur sur les éléments du vecteur.

1. Donner la déclaration complète, en C, de la structure de données représentant le TAD Vecteur d'entiers, c-à-d :

- Les déclarations nécessaires pour définir **le type Vecteur d'entiers** ;
- Les déclarations, en C, **des primitives** du TAD Vecteur d'entiers.

2. Définir les primitives (*ou opérations de base*) du TAD Vecteur d'entiers, c'est-à-dire pour chaque primitive, écrire la fonction C correspondante.
3. Au TAD Vecteur d'entiers on ajoute les deux opérations auxiliaires suivantes :

- **produit_scalaire** qui calcule le produit scalaire de deux vecteurs d'entiers.
- **decalage_circulaire** qui prend un vecteur et retourne le vecteur dans lequel tous les éléments sont décalés d'une place vers la droite (*le dernier est inséré à la 1ère place*).

Ecrire en langage C les deux opérations ci-dessus.

d. Corrigé :

1. Déclarations dans le fichier **`Vecteur.h`**

```
typedef  Element int ;
typedf struct{
        Int taille ;
        Element *table ;
}Vecteur ;
Vecteur Vect(int) ;
Vecteur  changer_i_eme( Vecteur,int,Element) ;
Element i_eme(Vecteur,int) ;
Int taille(Vecteur) ;
```

2. Implémentation dans **`Vecteur.c`**

```
#include "Vecteur.h"
#include "stdlib.h"
#include "conio.h"
Vecteur Vect(int n) {
     Vecteur V ;
     V.taille=n ;
     V.table=(Element *)malloc(n*sizeof(Element)) ;
     return V ;
}
Vecteur  changer_i_eme( Vecteur V,int i,Element e) {
     (V.table)[i]=e ;
     return V ;
}
Element i_eme(Vecteur V,int i) {
     return((V.table)[i]) ;
}
```

```
int taille(Vecteur) {
     return(V.taille) ;
}
```

3. Opérations auxiliaires

```
int produit_scalaire(Vecteur V1, Vecteur V2){
   int  i, ps=0 ;
   if(taille(V1) !=taille(V2)){
        printf("\n Erreur ") ;
        exit(-1) ;
   }
   else{
        for(i=0 ;i<taille(V1) ;i++)
             ps+=ieme(V1,i)* ieme(V2,i) ;
        return (ps) ;
   }
}
Vecteur dec_circulaire(Vecteur V){
   int i,x ;
   x=ieme(V,taille(V)-1) ;
   for(i=taille(V)-1 ; i>0 ;i++)
        changer_ieme(V,i,ieme(V.i-1)) ;
   changer_ieme(V,0,x) ;
   return V ;
}
```

II. Listes

1. Listes doublement triées

e. Enoncée

Dans une liste multi-chaînée, chaque noeud contient plusieurs liens vers d'autres noeuds de la liste (la liste doublement chaînée vue en cours en est un cas particulier).
Dans cet exercice, nous considérons la structure suivante d'une liste doublement chaînée :

```
typedef struct st_maillon {
      char nom [100];
      unsigned int age;
      st_maillon * suivant_nom ;
      st_maillon * suivant_age ;
} Maillon ;

typedef struct st_liste {
      Maillon * par_nom ;
      Maillon * par_age
 } ListeTriee ;
```

Cette structure peut être utilisée pour organiser les informations (nom, age) d'une collection de personnes dans l'ordre croissant, selon deux critères simultanément : le nom et l'age. En partant de par_nom et en suivant les pointeurs suivant_nom, nous parcourons la liste dans l'ordre alphabétique des noms ; ainsi en partant de par_age et en suivant les pointeurs suivant_age, nous parcourons la liste dans l'ordre croissant des ages.

1. Dessinez la liste triée (par nom et par age) correspondante aux personnes suivantes `<("Anne", 29), ("Cyrille", 41), ("Guillaume", 39)>`.
2. Écrivez la fonction d'insertion des informations d'une nouvelle personne dans une liste triée. La liste reste triée par nom et par age après l'insertion.

   void inserer (ListeTriee * l, char * nom , unsigned int age);

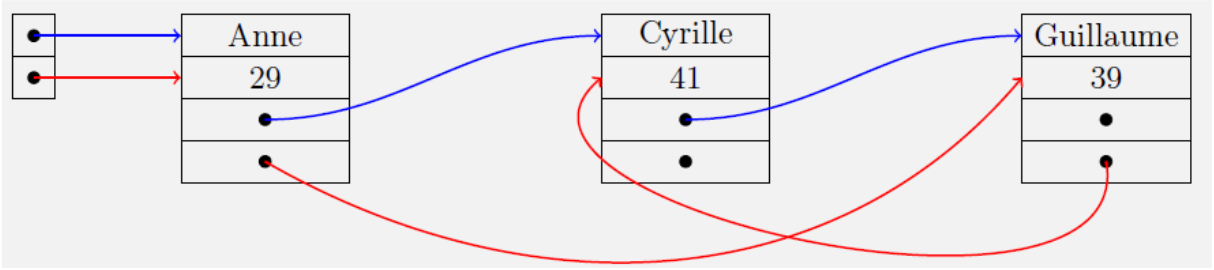
   Quelle est la complexité asymptotique de cette fonction en nombre de comparaison en fonction de la longueur de la liste n ?
3. Écrivez une fonction qui prend une liste multi-chaînée et renvoie une copie de cette liste :

```
 /* retourne un pointeur vers la tête de la nouvelle liste */
ListeTriee * dupliquerListe ( ListeTriee l );
```

   Cette fonction doit avoir une complexité en temps linéaire O(n) ou n est la longueur de la liste. Vous disposez de la fonction `Maillon* copierMaillon( Maillon* p );` qui copie le maillon p et renvoie un pointeur vers la copie (vous n'avez pas à l'implémenter)

f. Corrigé

1.



1.

```
/* on suppose que l est non null */
 void inserer ( ListeTriee * l, char * nom , unsigned int age) {
 /* créer un maillon pour le nouvel élé ment */
       Maillon * nouv = ( Maillon *) malloc ( sizeof ( Maillon ) );
       strcpy (nouv ->nom , nom );
       nouv ->age = age ;
       nouv -> suivant_nom = NULL ;
       nouv -> suivant_age = NULL ;

       /* insérer au bon endroit selon l'ordre
           alphab é tique des nom */
       Maillon * prev = NULL , cour = l-> par_nom ;
       while ( cour != NULL && strcmp (cour ->nom , nouv -> nom) > 0)
      {
            prev = cour ;
            cour = cour -> suivant_nom ;
       }
       if( prev == NULL )
            l-> par_nom = nouv ;
       else
            prev -> suivant_nom = nouv ;
       nouv -> suivant_nom = cour ;

       /* ajuster les pointeurs d'age */
       Maillon * prev = NULL ;
       Maillon * cour = l-> par_age ;
       while ( cour != NULL && cour ->age < nouv -> age) {
            prev = cour ;
            cour = cour -> suivant_age ;
       }
       if( prev == NULL )
            l-> par_age = nouv ;
       else
            prev -> suivant_age = nouv ;
       nouv -> suivant_age = cour ;
}
— Complexité en temps : O(n)
   2.
/* on suppose que l est non null */
 ListeTriee * dupliquerListe ( ListeTriee l ) {
       /* é tape 1: dupliquer chaque noeud de l et insé rer le
      nouveau noeud
       apr ès le noeud originale dans la liste l */
       Maillon * cour = l-> par_nom ;
```

```
        while ( cour != NULL ) {
              Maillon * copie = copierMaillon ( cour );
              copie -> suivant_nom = cour -> suivant_nom ;
              cour -> suivant_nom = copie ;
        }
        /* é tape 2: ajuster les pointeurs d'age */
        Maillon * cour = l-> par_nom ;
        while ( cour != NULL ) {
              cour -> suivant_nom -> suivant_age = cour -> suivant_age
             -> suivant_nom ;
              cour -> suivant_nom -> suivant_nom ;
        }
        /* é tape 3: séparer la liste copi ée de la liste originale */
        ListeTriee * dup_liste = ( ListeTriee *) malloc ( sizeof (
       ListeTriee ) );
        dup_liste -> par_nom = NULL ;
        dup_liste -> par_age = NULL ;
        if(l-> par_nom == NULL )
              return dup_liste ;
        else {
              dup_liste -> par_nom = l-> par_nom -> suivant_nom ;
              sup_liste -> par_age = l-> par_age -> suivant_nom ;
        }
        Maillon * orig = l-> par_nom ;
        Maillon * copie = NULL ;
        while ( orig != NULL ) {
              copie = orig -> suivant_nom ;
              orig -> suivant_nom = copie -> suivant_nom ;
              if(copie -> suivant_nom != NULL )
                    copie -> suivant_nom = copie -> suivant_nom ->
                   suivant_nom ;
              orig = orig -> suivant_nom ;
        }
        return dup_liste ;
  }
```

— Complexité en temps : O(n)
— Espace auxiliaire utilisée : O(1)

2. Liste chaînée de chaîne de caractères

a. Enoncée

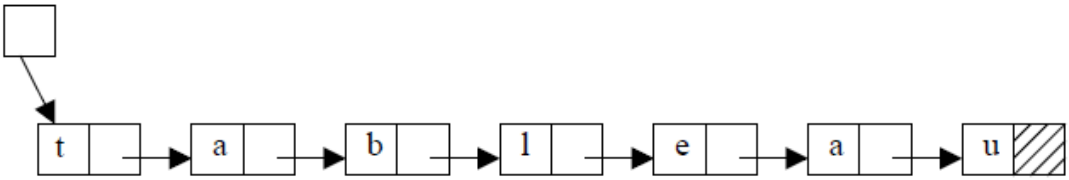
On représente une chaîne de caractères sous la forme d'une liste chaînée. L'information contenue dans une cellule est réduite à un seul caractère.

1. Définir la structure de données représentant une liste chaînée de caractères.

On veut définir une fonction égalité qui teste si deux chaînes de caractères représentées par des listes chaînées sont identiques. Pour deux listes chaînées représentant des chaînes de caractères quelconques, la fonction égalité retourne vrai si les deux chaînes sont identiques et faux dans le cas contraire.

2. Ecrire en C la fonction égalité sous forme itérative et sous forme récursive.

b. Corrigé :



1. Liste chaînée de caractères

```
Typedef struct cellule{
      Char c ;
      Cellule suivant ;
}cellule ;
Typedef cellule *Liste ;
```

2. Fonction égalité

```
/* Forme itérative */
Boolean egale(Liste *L1,Liste L2){
      While(L1 !=NULL && L2 !=NULL && L1->c==L2->c){
            L1=L1->suivant ;
            L2=L2->suivant ;
      }
      If(L1 ==NULL && L2 ==NULL)
            Return 1;
      Else
            Return 0 ;
}
/* Forme récursive */
Boolean egale(Liste *L1,Liste L2){
      if(L1 ==NULL && L2 ==NULL)
            Return 1 ;
      Else if(L1 ==NULL || L2 ==NULL)
            Return 0;
               Else if  (L1->c !=L2->c)
                        Return 0 ;
            Else
                  Return(L1->suivant,L2=L2->suivant ) ;
}
```

III. Piles

1. Tri avec deux piles

a. Enoncée

Dans cet exercice, nous implémentons deux piles d'entiers dans un seul tableau tab de taille fixe n. Chacune des deux piles peut contenir au plus n entiers, sans qu'à aucun moment le nombre d'entiers contenus dans les deux piles ne soit supérieur à n. Nous utilisons cette structure en suite pour trier une suite d'entiers. La structure utilisée est la suivante :

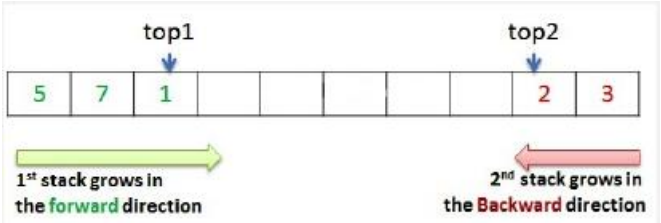
```
typedef struct deux_piles_tab {
      int * tab;
      int n;
} Deux_piles ;

 Deux_piles * p; /* variable globale partag ée par les fonctions */
```

1. Implémenter les fonctions estVide, empiler, sommet, depiler, qui prendront un premier argument (0 ou 1) désignant la pile à utiliser. Les fonctions partageront la variable p, il n'est donc pas nécessaire de la passer en paramètre.
2. Avec n = 4, et en supposant que les deux piles sont initialement vides, écrivez le contenu du tableau tab après exécution de la suite d'instructions suivante :
   ```
   empiler (0 ,1); empiler (0 ,2); empiler (1 ,5); empiler (0 ,3);
   depiler (0); depiler (0); empiler (1 ,4); empiler (0 ,1);
   ```
3. En utilisant les fonctions de la question 1, écrivez une fonction qui utilise la structure précédente de deux piles pour trier une suite d'entiers dans l'ordre croissant. Nous supposons que la suite d'entiers est déjà contenue dans la pile 0. A la fin de l'exécution, la suite triée est contenue dans la pile 1.
4. Donnez la complexité asymptotique de votre fonction de tri en nombre d'opérations nécessaires sur les piles.

b. Corrigé

1. Il faut stocker l'indice du sommet de chaque pile (ou bien sa hauteur/taille). Pour cela on peut modifier la structure proposée, en créer une nouvelle ou utiliser des variables globales, etc.




```
typedef struct deux_piles_tab {
      int * tab;
      int n; /* taille du tableau */
      int top1 ; /* sommet de la
                  premi ère pile , initialisé à 0 */
      int top2 ; /* sommet de la deuxième
                        pile , initialisé à n -1 */
 } Deux_piles ;

 Deux_piles * p; /* variable globale partagée
          par les fonctions */
```

```
int estVide (int pile ) {
      if ( pile == 0)
            return p-> top1 == 0;
      else
      return p-> top2 == n - 1;
}

void empiler (int pile , int e) {
      if ( top1 == top2 + 1)
            printf (" piles pleines \n");
      else if ( pile == 0) {
            p-> tab[ top1 ] = e;
            p-> top1 ++;
              }
              else {
            p-> tab[ top2 ] = e;
            p->top2 --;
                  }
}
int sommet (int pile ) {
      if ( estVide ( pile ))
            printf (" pile vide \n");
      else if ( pile == 0)
           return p->tab[p-> top1 - 1];
                 else
           return p->tab[p-> top2 + 1];
}
int depiler (int pile ) {
      int som;
      if ( estVide ( pile ))
            printf (" pile vide \n");
      else if ( pile == 0) {
            som = p->tab[p-> top1 - 1];
            p->top1 --;}
                  else {
            som = p->tab[p-> top2 + 1];
            p-> top2 ++;
                    }
      return som;
}
```

2. Voir la figure suivante :

empiler(0,1)	1			
empiler(0,2)	1	2		
empiler(1,5)	1	2		5
empiler(0,3)	1	2	3	5
depiler(0)	1	2		5
depiler(0)	1			5
empiler(1,4)	1		4	5
empiler(0,1)	1	1	4	5

3. L'algorithme suivant est inspiré de l'algorithme de tri par insertion :

```
void tri () {
```

```
        while (! estVide (0) ) {
              int e = depiler (0);
              while ( ! estVide (1) && e < sommet (1) )
              empiler (0, depiler (1));
              empiler (1, e);
        }
}
```

4. La complexité de l'algorithme O(n2) dans le pire cas et en moyenne, et linéaire O(n) dans le meilleur cas

2. PileMin

a. Enoncée

Le but de cet exercice est de concevoir une structure de données appelée PileMin qui implémente le type abstrait Pile et permet d'obtenir son élément minimum efficacement.
Nous supposons que l'on dispose d'un type Pile, ainsi que les opérations suivante :

```
void empiler ( Pile p, int valeur );
int depiler ( Pile p); /*retourne l'élément au sommet et le supprime
*/
int sommet ( Pile p); /* retourne l'élé ment au sommet sans le
supprimer */
int estVide ( Pile p);
```

1. Complétez la structure de données PileMin pour implémenter une pile d'entiers qui permet d'effectuer les opérations empiler, depiler, estVide, minimum en temps O(1).

```
typedef struct pile_min_st {
Pile * elems ;
 ... /* à compl éter */
 } PileMin ;
```

2. Écrivez les opérations empilerMin, depilerMin, minimum et vérifier que leur complexité en temps de calcul est de O(1) :

```
void empilerMin ( PileMin p, int valeur );
int depilerMin ( PileMin p);
int minimum ( PileMin p); /* retourne la valeur minimale */
```

b. Corrigé

1. L'idée est d'utiliser une pile auxiliaire pour stocker les valeurs minimums. Les opérations empiler et depiler seront implémenter de façon à garantir que le sommet de la pile auxiliaire correspond toujours à la valeur minimum de la pile d'éléments.

```
typedef struct pile_min_st {
    Pile * elems ;
    Pile * min;
 } PileMin ;
```

2. Fonctions de PileMin

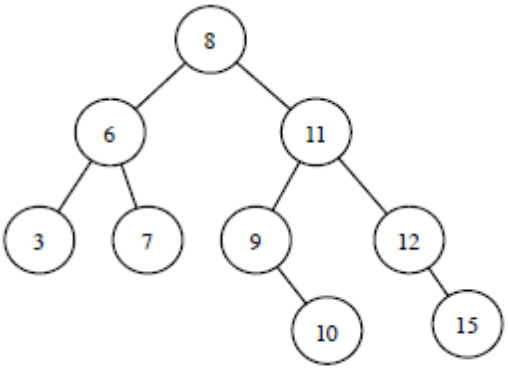
```
void empilerMin ( PileMin p, int valeur ) {
      if( estVide (p. elems ) ) {
            empiler (p.elems , valeur );
            empiler (p.min , valeur );
      }
     else {
            if ( valeur < sommet (p.min) )
                  empiler (p.min , valeur );
            else
                  empiler (p.min , valeur );
      }
}
int depilerMin ( PileMin p) {
      depiler (p.min );
      return depiler (p. elems );
 }
int minimum ( PileMin p) {
      return sommet (p.min );
}
```

IV. Arbres

1. Arbre binaire de Recherche

a. Enoncée

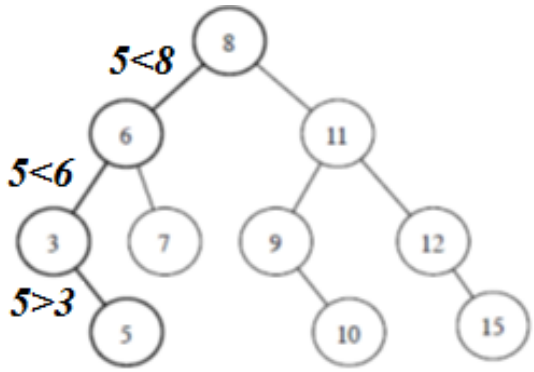
Considérons l'arbre binaire de Recherche suivant.



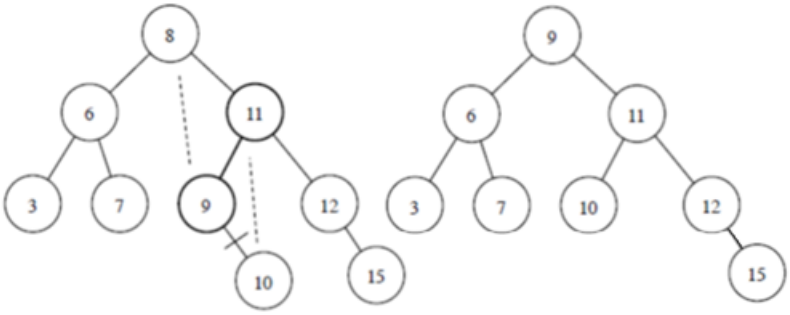
1. Donner les opérations successives correspondant à l'ajout de la clé 5.
2. Donner les opérations successives correspondant à la suppression de la clé 8, par les 2 méthodes données dans le cours.

b. Corrigé

1.

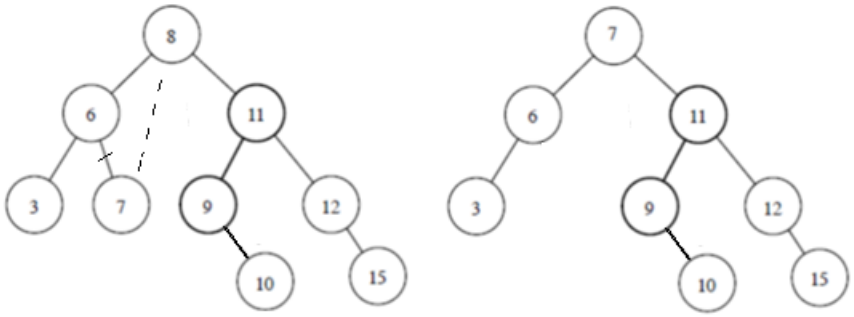


2. Le nœud de clé 8 étant la racine, en le supprimant, il peut être remplacé soit par le nœud de clé minimum du sous arbre droit, ou le nœud de clé maximum du sous arbre gauche.
   Cas 1 :



La clé 9 est le minimum, le noeud correspondant est supprimé pour remplacer le noeud de clé 8. Ensuite, il est remplacer par le noeud de clé 10 qui est son fils unique.

Cas 2 :



La clé 7 est le minimum des clés du sous arbre gauche, le noeud correspondant est une feuille, il est donc supprimé pour remplacer le noeud de la racine

2. Tri par tas

a. Enoncée

Considérons le tableau ci-dessous :

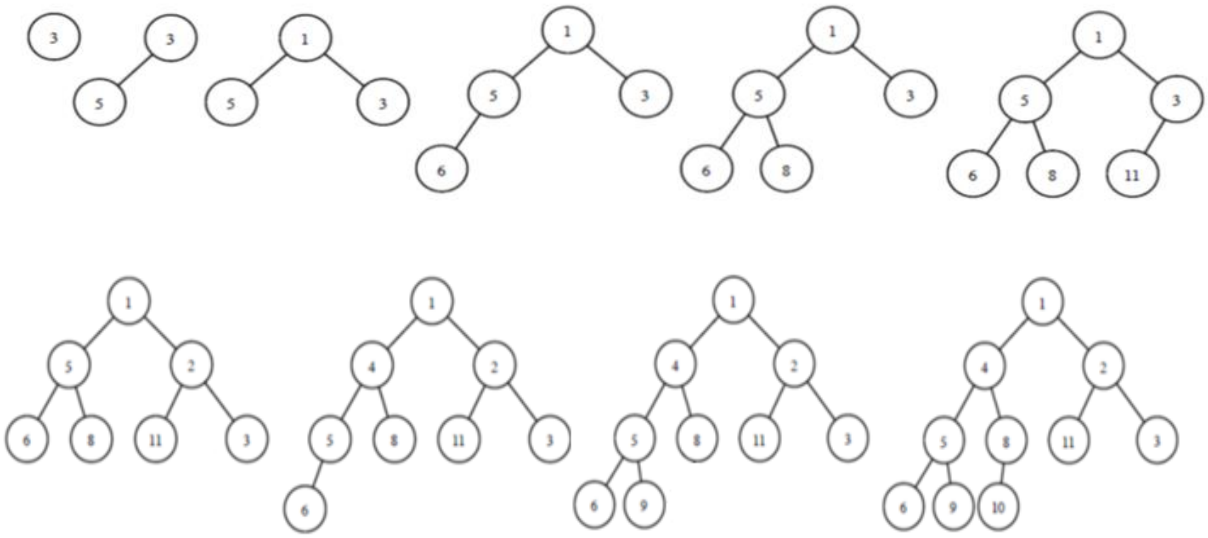
3 5 1 6 8 11 2 4 9 10

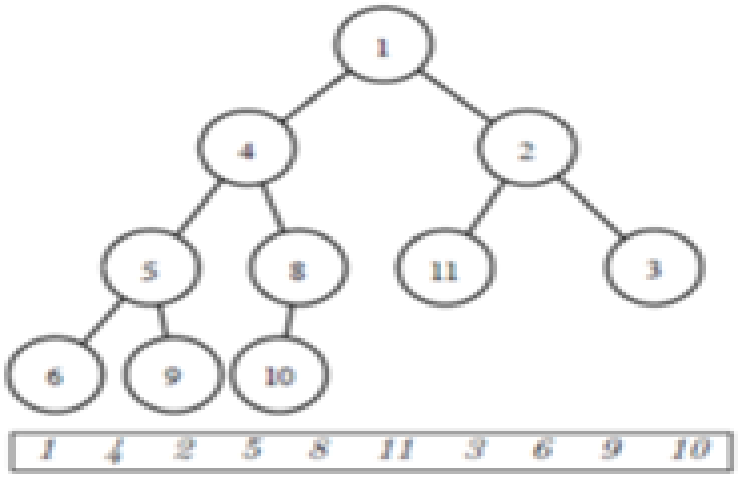
1. Construire un ***Tas Minimier*** correspondant à l'ensemble de ces éléments. Représenter le tout sous la forme d'un tableau et sous la forme d'un arbre. Préciser quelle méthode vous aller utiliser.
2. Effectuer un tri par tas à partir du tas obtenu à la question précédente. Pour chaque itération représenter le tableau résultant ainsi que le tas correspondant aux éléments non encore triés sous la forme d'un arbre.

b. Corrigé

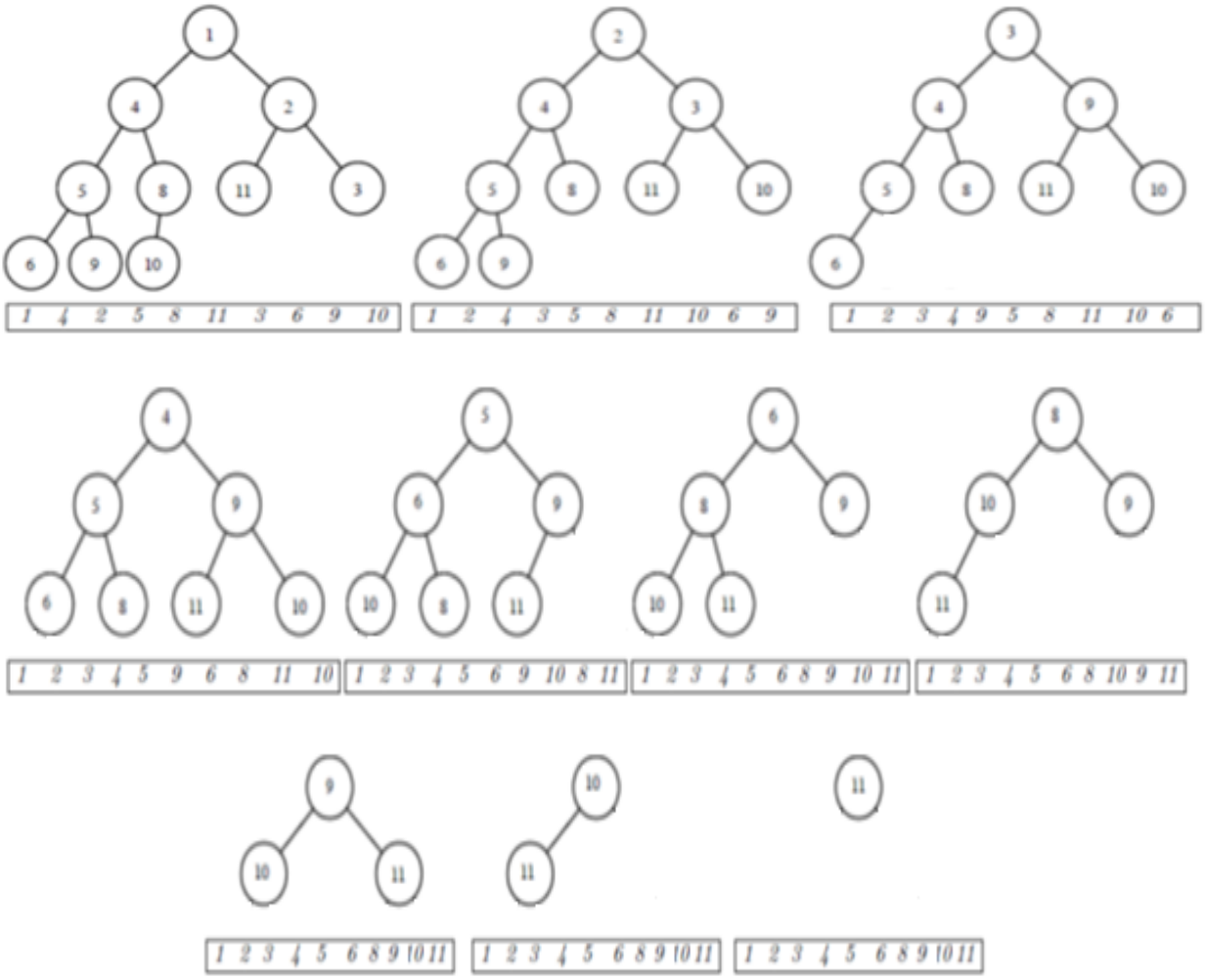
Un tas minimier est un arbre binaire parfait dont la clé de chaque nœud père est inférieure aux clés des nœuds descendant (la racine possède la clé minimum et la différence entre les profondeurs des sous arbres est 0 ou 1, de plus seulement le dernier niveau peut ne pas être rempli)

1. Tas minimier





2. Tri avec tas :



V. Graphes

1. Puits universel de graphe orienté

a. Enoncé

1. Considérons un graphe orienté **G = (S, A).** Un sommet X est un puits universel s'il est de degré entrant : /S/-1 (le nombre d'éléments de S moins 1) et de degré sortant 0. Etant donné une représentation d'un graphe G = (S, A) par une matrice d'adjacence, proposer un algorithme permettant de déterminer s'il existe un puits universel.

Note : Il faut rappeler l'implémentation du TAD Graphe par une matrice d'adjacence.

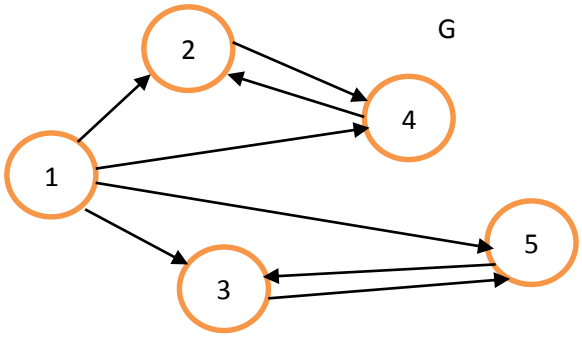
2. Donner un exemple simple d'un graphe orienté contenant un puits.
3. Un graphe orienté contenant un puits universel peut-il être fortement connexe ? Justifie

b. Corrigé

1. Considérons que les sommets du graphe sont numérotés de 1 à n=/S/.
   Le sommet ayant le numéro i est un puits universel si et seulement si dans la matrice la ligne i ne contient que des 0, et la colonne i ne contient que des 1 sauf à l'intersection de la ligne i. On peut donc tester de façon exhaustive, ce qui prend un temps quadratique en n nombre de sommets.

```
typedef    N_max   100 ;
typedef  struct graphe{
      int  n ;
      int M[N_max][N_max] :
}GrapheM ;
int  puit_Universel(GrapheM G){
      int i,j,c1,c2;
      for(i=0 ;i<G.n ;i++){
            c1=0 ;c2=0 ;
            for(j=0 ;j<G.n ;j++)
                  c1=c1+G.M[i][j] ;
            if(c1==0)
                  for(j=0 ;j<G.n && j !=i ;j++)
                        c2=c2+G.M[j][i] ;
            if(c2==(G.n-1))
                  return 1 ;
      }
      return 0 ;
}
```

2. Graphe orienté avec le sommet de numéro 1 un puits universel

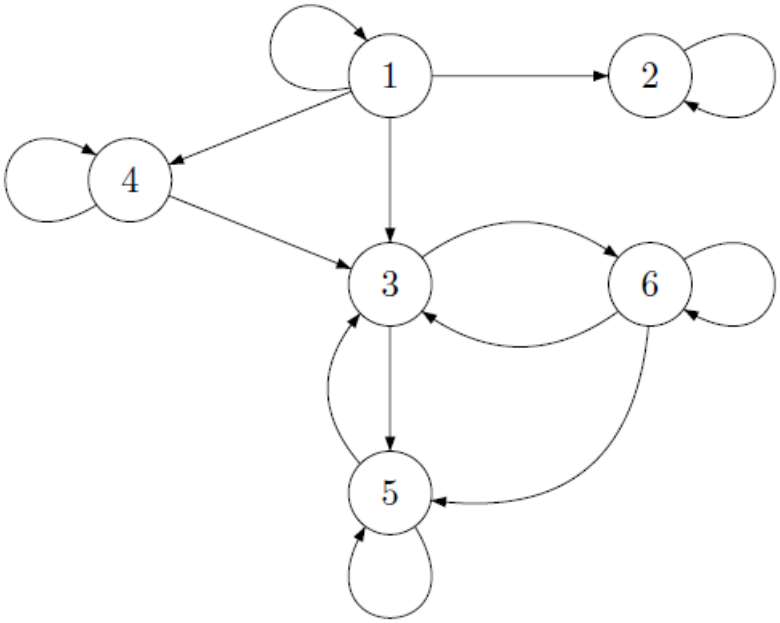


3. Un graphe orienté est fortement connexe si chacun de ses sommets est atteignable depuis les autres sommets. Cependant, si un graphe orienté contient un sommet qui est un puits universel, ce dernier n'est pas atteignable depuis les autres sommets. Ce qui vaut dire qu'un graphe orienté avec puits universel n'est pas fortement connexe.

2. Représentations de graphe orienté

a. Enoncé

Soit le graphe orienté suivant :

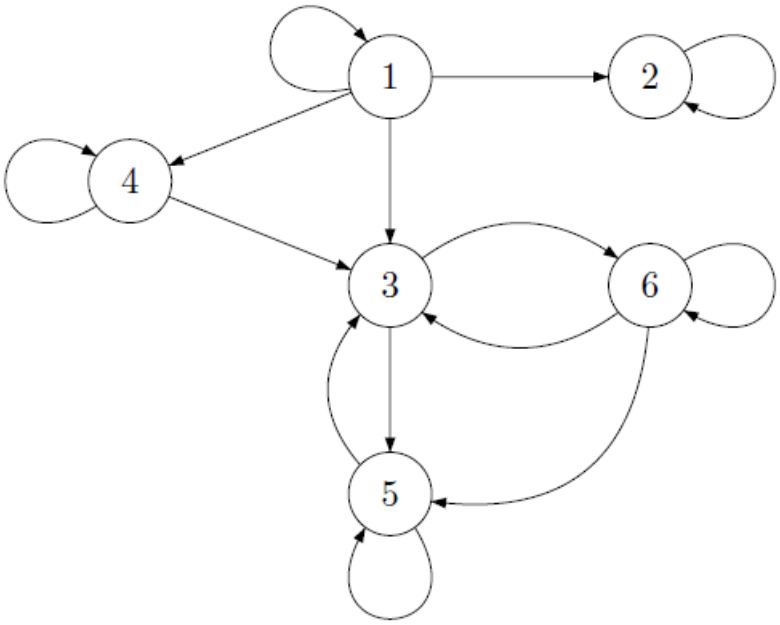


1. Donner une représentation du graphe ci-dessus au moyen d'une liste d'adjacence, puis au moyen d'une matrice d'adjacence. Considérons un graphe G = (V,E) sans boucle triviale. On appelle matrice d'incidence du graphe G, la matrice à |V| lignes et |E| colonnes, B = (bi,j) définie par :

$$b_{i,j} = \begin{cases} -1 & \text{si l'arête j part du sommet i} \\ 1 & \text{si l'arête j arrive dans sommet i} \\ 0 & \text{sinon} \end{cases}$$

2. Donner une représentation du même graphe (ci-dessus) en matrice d'incidence. Quels problèmes rencontre-t-on?
3. Proposer un algorithme de construction de la matrice d'incidence à partir de la liste d'adjacence d'un graphe, puis à partir de sa matrice d'adjacence.
4. Si B^T désigne la transposée de B, que représente la matrice BB^T ?
5. Dans chacune des trois représentations, quel temps faut-il pour déterminer si deux sommet u et v donnés sont voisins ?
6. Dans chacune des trois représentations, quel temps faut-il pour calculer le degré sortant d'un sommet ? Même question pour le degré entrant.
7. Réécrivez l'algorithme générique de parcours d'un graphe vu en cours de façon adaptée à chaque représentation. Quel est le temps d'exécution, à chaque fois ?

b. Corrigé :



1. En listes d'adjacences : 1 : (1,2,3,4); 2 : (2) ; 3 : (5,6) ; 4 : (4,3); 5 : (5,3) ; 6 : (6,5,3) ; il s'agit en fait de listes chaînées
   En matrice d'adjacences :

	1	2	3	4	5	6
1	1	1	1	1	0	0
2	0	1	0	0	0	0
3	0	0	0	0	1	1
4	0	0	1	1	0	0
5	0	0	1	0	1	0
6	0	0	1	0	1	1

2. On rencontre deux difficultés : d'abord, les arêtes n'ont pas de nom ; il faut en donner – Les boucles! On décide de mettre pour cet exercice seulement un 2 en $b_{u,e}$ dans la matrice d'incidence, si e est une boucle autour du sommet u. Notons qu'on étend ainsi la définition standard.

	a	b	c	d	e	f	g	h	i	j	k	l	m	n
1	2	-1	-1	-1	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	2	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	-1	-1	0	1	0	1	0	0	1
4	0	0	0	1	0	0	0	2	-1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	2	-1	0	1	0
6	0	0	0	0	0	0	1	0	0	0	0	2	-1	-1

3. Pour construire la matrice d'incidence : tout d'abord on détermine sa taille et pour cela on compte le nombre d'arêtes du graphe. Ensuite, pour chaque arête on remplit la matrice (qui est au départ initialisée à 0).

Cas des listes d'adjacence :

```
Entrée: tableau de liste d'adjacence E de taille n
Variables: entier i initialisé à 0
           sommet x,y
Début:
      Pour x de 1 à n faire
            Pour tout voisin de y de x faire
                  i++
            fin pour
      fin pour
      B = matrice de taille n*i
      i=0
      Pour x de 1 à n faire
            Pour tout voisin de y de x faire
                  i++
                  B(x,i)=-1
                  B(y,x)=1
            Fin pour
      Fin pour
Fin.
```

Cas des matrices d'adjacence :

```
Entrée: matrice d'adjacence M de taille n
Variables: entier i initialisé à 0
           sommet x,y
Début:
      Pour x de 1 à n faire
            Pour tout y de 1 à n faire
                  si M(x,y)=1 faire
                  i++
                  finsi
            fin pour
      fin pour
      B = matrice de taille n*i
      i=0
      Pour x de 1 à n faire
            Pour tout y de 1 à n faire
                  si M(x,y)=1 faire
                  i++
                  B(x,i)=-1
                  B(y,x)=1
                  Fin si
            Fin pour
      Fin pour
```

Fin.

4. La matrice $M = BB^T = (m_{i,j})$ est carrée de taille n. De plus :

$$m_{i,j} = \sum_{k=1}^{nbarêtes} b_{i,k}b_{j,k} = \begin{cases} si\ i \neq j & le\ nombre\ d'arêtes\ entre\ i\ et\ j \\ si\ i = j & le\ degré\ du\ sommet\ i \end{cases}$$

5. – **Listes** : $O(d^+(x))$, car on parcours la liste de u à la recherche de v
   – **Matrice d'adjacence** : O(1), c'est un simple accès à $E_{u,v}$
   – **Matrice d'incidence** : O(m), car pour toute arête i on teste $b_{u,i}$ et $b_{v,i}$ qui doivent être non-nuls tous les deux

6. **Degré sortant :**
   – **Listes** : $O(d^+(x))$, c'est la taille taille d'une liste d'adjacence d'un sommet.
   – **Matrice d'adjacence** : O(n), comptant les 1 dans la *u*ème ligne de la matrice
   – **Matrice d'incidence** : O(m), comptant pour toute arête i les $b_{u,i}$ valant 1
   Degré entrant :
   – **Listes** : O(m), car il faut parcourir toutes les listes et compter les listes où *u* apparaît)
   – **Matrice d'adjacence** : O(n), comptant les 1 dans la *u*ème colonne de la matrice
   – **Matrice d'incidence** : O(m), comptant pour toute arête i les $b_{u,i}$ valant -1

7. Seule la ligne "pour tout voisin v de u" est à réécrire
   – **Listes** :

```
C = u.tete
Tant que C != null
      v = C.val
      ... // calcul sur v
Fin tant que
```

   Se fait en $O(d^+(u))$ pour chaque sommet u. Chaque sommet est vu une fois. On est donc en O(n+m) EN TOUT.
   – **Matrice d'adjacence** :

```
Pour v de 1 a n
      Si E(u,v) = 1
            ... // calcul sur v
      Fin si
Fin pour
```

   Se fait en $O(n)$ pour chaque sommet u donc $O(n^2)$ en tout.
   – **Matrice d'incidence** :

```
Pour e de 1 a m
      Si b(u,e) = 1 // u origine de e
            Pour v de 1 a n
                  Si b(v,e) = -1 //y destination de e
                   ... // calcul sur v
                  Fin si
            Fin pour
      Fin si
Fin pour
```

   La complexité est grande : O(nm) pour chaque sommet u donc $O(n^2m^2)$ en tout.

VI. Références

[1] M. Divay. Algorithmes et structures de données génériques. Dunod, 2004.
[2] J.P. Braquelaire. Méthodologie de la programmation en C. Dunod, 2000.
[3] T. Cormen, C. Leiserson et R. Rivest. Introduction à l'algorithmique. Dunod, 1994