Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mohamed Khider - Biskra
Faculté des Sciences Exactes et des Sciences de la Nature et de la Vie
Département d'Informatique

2ème année LMD

Cours d'Algorithmique et structures de données 1

Chargé du cours : Dr. Abdelhamid DJEFFAL

Année Universitaire 2012/2013

Sommaire

1 Introduction **3**

1.1 Résolution d'un problème en informatique 3

1.2 Notion d'algorithme 4

1.3 Langage algorithmique utilisé 5

2 Complexité des algorithmes **7**

2.1 Introduction 7

2.2 O-notation 7

2.3 Règles de calcul de la complexité d'un algorithme 8

2.4 Complexité des algorithmes récursifs 13

2.5 Types de complexité algorithmique 14

2.6 Exercices 15

3 Structures séquentielles **20**

3.1 Les listes linéaires chaînées (LLC) 20

3.2 Les piles (stacks) 26

3.3 Les Files d'attente (Queues) 34

3.4 Exercices 39

4 Structures Hiérarchiques **45**

4.1 Les arbres 45

4.2 Les arbres binaires de recherche 56

4.3 Les tas (Heaps) 63

4.4 Exercices 66

5 Structures en Tables **69**

5.1 Introduction 69

5.2 Accès séquentiel . . . 70
5.3 Table triée . . . 70
5.4 Hachage (HashCoding) . . . 70
5.5 Exercices . . . 73

6 Les graphes **75**
6.1 Introduction . . . 75
6.2 Définitions . . . 75
6.3 Représentation des graphes . . . 78
6.4 Parcours de graphes . . . 80
6.5 Plus court chemin (algorithme de Dijkstra) . . . 83
6.6 Exercices . . . 84

7 Preuve d'algorithmes **86**
7.1 Introduction . . . 86
7.2 Méthode de preuve d'algorithme . . . 86
7.3 Outils de preuve d'algorithme (Logique de Hoare) . . . 88
7.4 Exemple . . . 91
7.5 Conclusion . . . 92

8 Sujets d'examens **93**

Références **137**

Chapitre 1

Introduction

L'utilisation d'un ordinateur pour la résolution d'un problème nécessite tout un travail de préparation. N'ayant aucune capacité d'invention, l'ordinateur ne peut en effet qu'exécuter les ordres qui lui sont fournis par l'intermédiaire d'un programme. Ce dernier doit donc être établi de manière à envisager toutes les éventualités d'un traitement.
Exemple : le problème Div(a,b), n'oubliez pas le cas b=0 !

1.1 Résolution d'un problème en informatique

Plusieurs étapes sont nécessaires pour résoudre un problème en informatique :

- Etape 1 : Définition du problème

  Il s'agit de déterminer toutes les informations disponibles et la forme des résultats désirés.

- Etape 2 : Analyse du problème

  Elle consiste à trouver le moyen de passer des données aux résultats. Dans certains cas on peut être amené à faire une étude théorique. Le résultat de l'étape d'analyse est un algorithme. Une première définition d'un algorithme peut être la suivante :
  "On appelle algorithme une suite finie d'instructions indiquant de façon unique l'ordre dans lequel doit être effectué un ensemble d'opérations pour résoudre tous les problèmes d'un type donné."
  Sachez aussi qu'il existe des problèmes pour lesquels on ne peut trouver une solution et par conséquent il est impossible de donner l'algorithme correspondant.

- Etape 3 : Ecriture d'un algorithme avec un langage de description algorithmique

  Une fois qu'on trouve le moyen de passer des données aux résultats, il faut être capable de rédiger une solution claire et non ambiguë. Comme il est impossible de le

faire en langage naturel, l'existence d'un langage algorithmique s'impose.

- Etape 4 : Traduction de l'algorithme dans un langage de programmation

  Les étapes 1, 2 et 3 se font sans le recours à la machine. Si on veut rendre l'algorithme concret ou pratique, il faudrait le traduire dans un langage de programmation. Nous dirons alors qu'un programme est un algorithme exprimé dans un langage de programmation.

- Etape 5 : Mise au point du programme

  Quand on soumet le programme à la machine, cette dernière le traite en deux étapes :

  1. La machine corrige l'orthographe, c'est ce qu'on appelle syntaxe dans le jargon de la programmation.
  2. La machine traduit le sens exprimé par le programme.

  Si les résultats obtenus sont ceux attendus, la mise au point du programme se termine. Si nous n'obtenons pas de résultats, on dira qu'il y a existence des erreurs de logique. Le programme soit ne donne aucun résultat, soit des résultats inattendus soit des résultats partiels. Dans ce cas, il faut revoir en priorité si l'algorithme a été bien traduit, ou encore est-ce qu'il y a eu une bonne analyse.

1.2 Notion d'algorithme

1.2.1 Définition

On peut définir un algorithme comme suit :

Résultat d'une démarche logique de résolution d'un problème. C'est le résultat de l'analyse.

Ou encore :

Une séquence de pas de calcul qui prend un ensemble de valeurs comme entrée (input) et produit un ensemble de valeurs comme sortie (output).

1.2.2 Propriétés

On peut énoncer les cinq propriétés suivantes que doit satisfaire un algorithme :

1. Généralité : un algorithme doit toujours être conçu de manière à envisager toutes les éventualités d'un traitement.
2. Finitude : Un algorithme doit s'arrêter au bout d'un temps fini.
3. Définitude : toutes les opérations d'un algorithme doivent être définies sans ambiguïté

4. Répétitivité : généralement, un algorithme contient plusieurs itérations, c'est à dire des actions qui se répètent plusieurs fois.
5. Efficacité : Idéalement, un algorithme doit être conçu de telle sorte qu'il se déroule en un temps minimal et qu'il consomme un minimum de ressources.

1.2.3 Exemples

- PGCD (Plus Grand Commun Diviseur) de deux nombres u et v.
  - Algorithme naïf : on teste successivement si chaque nombre entier est diviseur commun.
  - Décomposition en nombres premiers.
- Algorithmes de tri
- Algorithmes de recherche
  - Recherche d'une chaîne de caractère dans un texte (Logiciels de traitement de texte).
  - Recherche dans un dictionnaire.
  - ... etc.

1.2.4 Remarque

Attention, certains problèmes n'admettent pas de solution algorithmique exacte et utilisable. On utilise dans ce cas des algorithmes heuristiques qui fournissent des solutions approchées.

1.3 Langage algorithmique utilisé

Durant ce cours, on va utiliser un langage algorithmique pour la description des différentes solutions apportées aux problèmes abordés. L'algorithme suivant résume la forme générale d'un algorithme et la plupart des déclarations et instructions utilisées.

```
Algorithme PremierExemple;
Type  TTab = tableau[1..10] de reel ;
Const Pi = 3.14 ;
Procédure Double( x : reel);
Début
    | x ← x * 2 ;
Fin;
Fonction Inverse( x : reel) : reel;
Début
    | Inverse ← 1/x ;
Fin;

Var  i, j, k : entier ;
     T : TTab ;
     S : chaine ;
     R : reel ;
Début
| Ecrire (' Bonjour, donner un nombre entier ≤ 10 :') ;
| Lire (i) ;
| Si (i>10) Alors
|     | Ecrire ('Erreur : i doit être ≤ 10')
| Sinon
|     | Pour j de 1 à i faire
|     |     | Lire(R) ;
|     |     | Double(R) ;
|     |     | T [j] ← R ;
|     | Fin Pour;
|     | k ← 1 ;
|     | Tant que (k ≤ i) faire
|     |     | Ecrire (T [k] * Inverse(Pi)) ;
|     |     | k ← k + 1 ;
|     | Fin TQ;
|     | S ← 'Programme terminé' ;
|     | Ecrire(S) ;
| Fin Si;
Fin.
```


Algorithme 1: Algorithme type

Chapitre 2

Complexité des algorithmes

2.1 Introduction

Le temps d'exécution d'un algorithme dépend des facteurs suivants :

- Les données utilisées par le programme,
- La qualité du compilateur (langage utilisé),
- La machine utilisée (vitesse, mémoire,...),
- La complexité de l'algorithme lui-même,

On cherche à mesurer la complexité d'un algorithme indépendamment de la machine et du langage utilisés, c-à-d uniquement en fonction de la taille des données n que l'algorithme doit traiter. Par exemple, dans le cas de tri d'un tableau, n est le nombre d'éléments du tableau, et dans le cas de calcul d'un terme d'une suite n est l'indice du terme, ...etc.

2.2 O-notation

Soit la fonction $T(n)$ qui représente l'évolution du temps d'exécution d'un programme P en fonction du nombre de données n.
Par exemple :

$$T(n) = cn^2$$

Où c est une constante non spécifiée qui représente la vitesse de la machine, les performances du langage, ...etc. On dit dans l'exemple précédent que la complexité de P est $O(n^2)$. Cela veut dire qu'il existe une constante c positive tel que pour n suffisamment grand on a :

$$T(n) \leq cn^2$$

Cette notation donne une majoration du nombre d'opérations exécutées (temps d'exécution) par le programme P. Un programme dont la complexité est $O(f(n))$ est un programme qui a $f(n)$ comme fonction de croissance du temps d'exécution.

2.3 Règles de calcul de la complexité d'un algorithme

2.3.1 La complexité d'une instruction élémentaire

Une opération élémentaire est une opération dont le temps d'exécution est indépendant de la taille n des données tel que l'affectation, la lecture, l'écriture, la comparaison ...etc.

La complexité d'une instruction élémentaire est $O(1)$

2.3.2 La multiplication par une constante

$$O(c * f(n)) = O(f(n))$$

Exemple :

$$O(\frac{n^3}{4}) = O(n^3)$$

2.3.3 La complexité d'une séquence de deux modules

La complexité d'une séquence de deux modules M_1 de complexité $O(f(n))$ et M_2 de complexité $O(g(n))$ est égale à la plus grande des complexité des deux modules : $O(max(f(n), g(n)))$.

$$O(f(n)) + O(g(n)) = O(max(f(n), (g(n))$$

2.3.4 La complexité d'une conditionnelle

La complexité d'une conditionnelle

Si (Cond) **Alors**
  M_1 ;
Sinon
  M_2 ;
Fin Si;

est le max entre les complexités de l'évaluation de <Cond>, M_1 et M_2.

Si (<Cond> $[O(h(n))]$) **Alors**
  | M_1 $[O(f(n))]$
Sinon
  | M_2 $[O(g(n))]$
Fin Si;

La complexité de la conditionnelle est : $Max\{O(h(n)), O(f(n)), O(g(n))\}$

2.3.5 La complexité d'une boucle

La complexité d'une boucle est égale à la somme sur toutes les itérations de la complexité du corps de la boucle.

Tant que (<Condition> $[O(h(n))]$) **faire**
  |
  | *[m fois]*
  | P ; $[O(f(n))]$
Fin TQ;

La complexité de la boucle est : $\sum^m Max(h(n), f(n))$ où m est le nombre d'itérations exécutées par la boucle.

Exemple 1 :

Algorithme Recherche;
Var T : **tableau**[1..n] **de entier ;**
x,i : **entier ;**
trouv : **booleen ;**
Début
Pour i **de** 1 **à** n **faire**
Lire (T[i]) ; $O(1)$ $O(\sum_1^n 1) = O(n)$
Fin Pour;
Lire(x) ; O(1)
Trouv ← faux ; $O(1)$ $O(1)$
i ← 1 ; $O(1)$ $O(n)$
Tant que (trouv=faux et i<=n [$O(1)$]) **faire**
Si (T[i]=x [$O(1)$]) **Alors**
Trouv ← vrai ; [$O(1)$] O(1) O(n)
Fin Si;
i ← i + 1 ; [$O(1)$]
Fin TQ;
Si (trouv=vrai [$O(1)$]) **Alors**
Ecrire (x,'existe') [$O(1)$] O(1)
Sinon
Ecrire (x, 'n"existe pas')[$O(1)$]
Fin Si;
Fin.

La complexité de l'algorithme est de $O(n) + O(1) + O(n) + O(1) = O(n)$.

Exemple 2 : Soit le module suivant :

```
Pour i de 1 à n faire                                   [n fois          O(∑_{i=1}^{n} n − i))]
    Pour j de i+1 à n faire                             [ n − i fois     O(∑_{1}^{n−i} 1) = O(n − i)]
        Si (T[i] > T[j] [O(1)] ) Alors
            tmp ← T[i] ; [O(1)]
            T[i] ← T[j] ; [O(1)]                                  O(1)
            T[j] ← tmp ;[O(1)]
        Fin Si;
    Fin Pour;
Fin Pour;
```

La complexité $= O(\sum^{n}(n-i) = O((n-1)+(n-2)\ldots+1)$
$= O(\frac{n(n-1)}{2}) = O(\frac{n^2}{2} - \frac{n}{2}) = O(n^2)$

Exemple 3 : Recherche dichotomique

```
Algorithme RechercheDecho;
Var  T : tableau[1..n] de entier ;
     x,sup,inf,m : entier ;
     trouv : booleen ;
Début
  Lire(x) ;            [O(1)]
  Trouv ← faux ;       [O(1)]
  inf ← 1 ;            [O(1)                    O(1)]
  sup ← n ;            [O(1)]
  Tant que (trouv=faux et inf<sup) faire
                                                [log2(n) fois]
      m ← (inf + Sup ) div 2 ;  [O(1)]
      Si (T[m]=x [O(1)]) Alors
          Trouv ← vrai [O(1)]
      Sinon
          Si (T[m]<x [O(1)]) Alors
              inf ← m +1 ;  [ O(1)          O(1)          O(log2(n))]
          Sinon
              Sup ← m - 1 ; [O(1)]
          Fin Si;
      Fin Si;
  Fin TQ;

  Si (trouv=vrai [O(1)]) Alors
      Ecrire (x,'existe') ; [O(1)]          O(1)
  Sinon
      Ecrire (x, 'n'existe pas') ; [O(1)]
  Fin Si;
Fin.
```

La complexité de l'algorithme est de :

$$O(1) + O(log_2(n)) + O(1) = O(log_2(n)) = O(log(n)).$$

2.4 Complexité des algorithmes récursifs

La complexité d'un algorithme récursif se fait par la résolution d'une équation de récurrence en éliminant la récurrence par substitution de proche en proche.

Exemple

Soit la fonction récursive suivante :

```
Fonction Fact( n : entier) : entier;
Début
    Si (n <= 1 [O(1)] ) Alors
        Fact ← 1                          [O(1)]
    Sinon
        Fact ← n * Fact(n - 1)            [O(1)]
    Fin Si;
Fin;
```

Posons $T(n)$ le temps d'exécution nécessaire pour un appel à $Fact(n)$, $T(n)$ est le maximum entre :

- la complexité du test $n \leq 1$ qui est en $O(1)$,
- la complexité de : Fact ← 1 qui est aussi en O(1),
- la complexité de : Fact ← n*Fact(n - 1) dont le temps d'exécution dépend de $T(n-1)$ (pour le calcul de Fact(n - 1))

On peut donc écrire que :

$$T(n) = \begin{cases} a & si \quad n <= 1 \\ b + T(n-1) & sinon \quad (sin > 1) \end{cases}$$

- La constante a englobe le temps du test ($n <= 1$) et le temps de l'affectation (Fact ← 1)
- La constante b englobe :
  * le temps du test ($n <= 1$),
  * le temps de l'opération * entre n et le résultat de $Fact(n-1)$,

* et le temps de l'affectation finale (Fact ← ...)

$T(n-1)$ (le temps nécessaire pour le calcul de $Fact(n-1)$ sera calculé (récursivement) avec la même décomposition. Pour calculer la solution générale de cette équation, on peut procéder par substitution :

$$\begin{aligned}
T(n) &= b + T(n-1) \\
&= b + [b + T(n-2)] \\
&= 2b + T(n-2) \\
&= 2b + [b + T(n-3)] \\
&= \dots \\
&= ib + T(n-i) \\
&= ib + [b + T(n-i+1)] \\
&= \dots \\
&= (n-1)b + T(n-n+1) = nb - b + T(1) = nb - b + a \\
T(n) &= nb - b + a
\end{aligned}$$

Donc Fact est en $O(n)$ car b est une constante positive.

2.5 Types de complexité algorithmique

1. $T(n) = O(1)$, temps constant : temps d'exécution indépendant de la taille des données à traiter.
2. $T(n) = O(log(n))$, temps logarithmique : on rencontre généralement une telle complexité lorsque l'algorithme casse un gros problème en plusieurs petits, de sorte que la résolution d'un seul de ces problèmes conduit à la solution du problème initial.
3. $T(n) = O(n)$, temps linéaire : cette complexité est généralement obtenue lorsqu'un travail en temps constant est effectué sur chaque donnée en entrée.
4. $T(n) = O(n.log(n))$: l'algorithme scinde le problème en plusieurs sous-problèmes plus petits qui sont résolus de manière indépendante. La résolution de l'ensemble de ces problèmes plus petits apporte la solution du problème initial.
5. $T(n) = O(n^2)$, temps quadratique : apparaît notamment lorsque l'algorithme envisage toutes les paires de données parmi les n entrées (ex. deux boucles imbriquées) Remarque : $O(n^3)$ temps cubique
6. $T(n) = O(2^n)$, temps exponentiel : souvent le résultat de recherche brutale d'une solution.

2.6 Exercices

1. Calculer la complexité de l'algorithme suivant :

```
Pour i de 2 à n faire
    k ← i-1 ;
    x ← T[i] ;
    Tant que (T[k] > x et k > 0) faire
        T[k+1] ← T[k] ;
        k ← k-1 ;
    Fin TQ;
    T[k+1] ← x ;
Fin Pour;
```

2. Calculer la complexité de l'algorithme suivant :

```
i ← n ;
S ← 0 ;
Tant que (i > 0) faire
    j ← 2*i ;
    Tant que (j > 1) faire
        S ← S+(j-i)* (S+1) ;
        j ← j-1 ;
    Fin TQ;
    i ← i div 2 ;
Fin TQ;
```

3. Calculer la complexité de l'algorithme suivant :

```
i ← 1 ;
j ← 0 ;
Pour k de 1 à n faire
    j ← i+j ;
    i ← j-i ;
Fin Pour;
```

Que fait cet algorithme sachant que le résultat est dans j ?

4. Calculer la complexité de la fonction récursive suivante :

```
Fonction Fib( n : entier) : entier;
Début
    Si (n < 2) Alors
        Fib ← 1 ;
    Sinon
        Fib ← Fib(n - 1)+Fib(n - 2) ;
    Fin Si;
Fin;
```

5. Calculer la complexité de l'algorithme suivant :

```
P ← 1 ;
Pour I de 1 à n faire
    J ← 1 ;
    K ← 1 ;
    Tant que (K ≤ n) faire
        P ← P * (K + J) ;
        K ← K + 1 ;
        Si (K > n) Alors
            J ← J + 1 ;
            Si (J > n) Alors
                K ← 1
            Fin Si;
        Fin Si;
    Fin TQ;
Fin Pour;
```

6. Calculer la complexité de l'algorithme suivant :

```
i ← 1 ;
Tant que (i < n) faire
    j ← 1 ;
    Tant que (j < 2*n) faire
        j ← j*2 ;
    Fin TQ;
    i ← i+1 ;
Fin TQ;
```

7. Trouver la complexité de la fonction suivante :

```
Var  x : entier ;
Fonction f( i, j, k : entier) : entier;
Début
    Si (k+j = i) Alors
        f ← ((i-j) div k) + 1 ;
    Sinon
        x ← f(i, j+1, k-2) ;
        f ← f(i+1, j+x, k-2) ;
    Fin Si;
Fin;
```

8. Trouver la complexité de l'algorithme suivant :

```
i ← 1 ;
j ← 1 ;
Tant que (i < n) faire
    Si (j < n) Alors
        j ← j * 2 ;
    Sinon
        j ← 1 ;
    Fin Si;
    i ← i + 1 ;
Fin TQ;
```

9. Calculer la complexité de la procédure récursive F(x, y, z) suivante :

```
Procédure F( x, y, z : reel);
Début
    y ← 2*z ;
    Si (x > x/(y - z)) Alors
        x ← x - 2 ;
        y ← y / 4 ;
        z ← z / 5 ;
        F(x, y, z) ;
    Fin Si;
Fin;
```

Chapitre 3

Structures séquentielles

3.1 Les listes linéaires chaînées (LLC)

3.1.1 Notion d'allocation dynamique

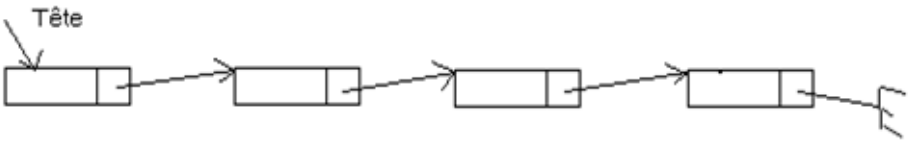
L'utilisation des tableaux statiques implique que l'allocation de l'espace se fait tout à fait au début d'un traitement, c'est à dire que l'espace est connu à la compilation. Pour certains problèmes, on ne connaît pas à l'avance l'espace utilisé par le programme. On est donc contraint à utiliser une autre forme d'allocation. L'allocation de l'espace se fait alors au fur et à mesure de l'exécution du programme. Afin de pratiquer ce type d'allocation, deux opérations sont nécessaires : allocation et libération de l'espace.

Exemple

Supposons que l'on veuille résoudre le problème suivant : "Trouver tous les nombres premiers de 1 à n et les stocker en mémoire". Le problème réside dans le choix de la structure d'accueil. Si on utilise un tableau, il n'est pas possible de définir la taille de ce tableau avec précision même si nous connaissons la valeur de n (par exemple 10000). On est donc, ici, en face d'un problème où la réservation de l'espace doit être dynamique.

3.1.2 Définition d'une liste linéaire chaînée

Une liste linéaire chaînée (LLC) est un ensemble de maillons, alloués dynamiquement, chaînés entre eux. Schématiquement, on peut la représenter comme suit :

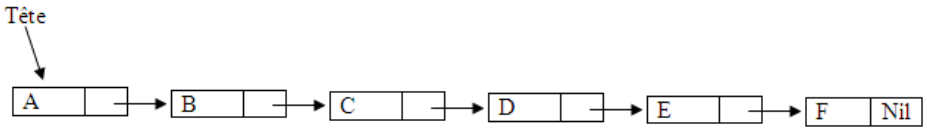


Un élément ou maillon d'une LLC est toujours une structure (Objet composé) avec deux champs : un champ *Valeur* contenant l'information et un champ *Adresse* donnant l'adresse du prochain élément. A chaque maillon est associée une adresse. On introduit ainsi une nouvelle classe d'objet : le type *Pointeur* qui est une variable contenant l'adresse d'un emplacement mémoire.

Une liste linéaire chainée est caractérisée par l'adresse de son premier élément souvent appelé *tête*. *Nil* constitue l'adresse qui ne pointe aucun maillon, utilisé pour indiquer la fin de la liste dans le dernier maillon.

3.1.3 Représentation réelle en mémoire

Soit la liste linéaire chainée représentée par la figure suivante :



La représentation réelle en mémoire de cette liste ressemble à la représentation suivante :

Tête= 4302

@	Valeur	Pointeur
:		
10	C	4300
12	F	NIL
:		
106	B	10
108	E	12
110	4302	
:		
4300	D	108
4302	A	106
4304		
:		

3.1.4 Modèle sur les listes linéaires chaînées

Dans le langage algorithmique, on définira le type d'un maillon comme suit :

```
Type TMaillon = Structure
    Valeur : Typeqq ; // désigne un type quelconque
    Suivant : Pointeur(TMaillon) ;
Fin ;
```

Afin de développer des algorithmes sur les LLCs, on construit une machine abstraite avec les opérations suivantes : Allouer, Libérer, Aff_Adr, Aff_Val, Suivant et Valeur définies comme suit :

- **Allouer(P) :** allocation d'un espace de taille spécifiée par le type de P. L'adresse de cet espace est rendue dans la variable de type Pointeur P.
- **Libérer(P) :** libération de l'espace pointé par P.
- **Valeur(P) :** consultation du champ Valeur du maillon pointé par P.
- **Suivant(P) :** consultation du champ Suivant du maillon pointé par P.
- **Aff_Adr(P, Q) :** dans le champ Suivant du maillon pointé par P, on range l'adresse Q.
- **Aff_Val(P,Val) :** dans le champ Valeur du maillon pointé par P, on range la valeur Val.

Cet ensemble est appelé modèle.

3.1.5 Algorithmes sur les listes linéaires chaînées

De même que sur les tableaux, on peut classer les algorithmes sur les LLCs comme suit :

- Algorithmes de parcours : accès par valeur, accès par position,...
- Algorithmes de mise à jour : insertion, suppression,...
- Algorithmes sur plusieurs LLCs : fusion, interclassement, éclatement,...
- Algorithmes de tri sur les LLCs : trie par bulle, tri par fusion,...

Création d'une liste et listage de ses éléments

```
Algorithme CréerListe;
Type  TMaillon = Structure
    Valeur : Typeqq ;
    Suivant : Pointeur(TMaillon) ;
Fin ;
Var  P, Q, Tete : Pointeur(TMaillon ;
    i, Nombre : entier ;
    Val : Typeqq ;
Début
  Tete ← Nil ;
  P ← Nil ;
  Lire(Nombre) ;
  Pour i de 1 à Nombre faire
      Lire(Val) ;
      Allouer(Q) ;
      Aff_val(Q, val) ;
      Aff_adr(Q, NIL) ;
      Si (Tete ≠ Nil) Alors
          Aff_adr(P, Q)
      Sinon
          Tete ← Q
      Fin Si;
      P ← Q ;
  Fin Pour;
  P ← Tete ;
  Tant que ( P ≠ Nil) faire
      Ecrire(Valeur(P)) ;
      P ← Suivant(P) ;
  Fin TQ;
Fin.
```

cet algorithme crée une liste linéaire chaînée à partir d'une suite de valeurs données, puis imprime la liste ainsi créée.

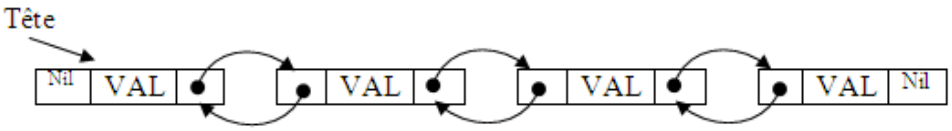
Recherche d'un élément dans une liste

Il s'agit bien entendu de la recherche séquentielle d'une valeur donnée.

```
Algorithme Recherche;
Type  TMaillon = Structure
    Valeur : Typeqq ;
    Suivant : Pointeur(TMaillon) ;
Fin ;
Var  P, Tete : Pointeur(TMaillon ;
    Trouv : booleen ;
    Val : Typeqq ;
Début
| Lire(Val) ;
| P ← Nil ;
| Trouv ← Faux ;
| Tant que ( P ≠ Nil et non Trouv) faire
|     Si (Valeur(P)=Val) Alors
|         | Trouv ← Vrai
|     Sinon
|         | P ← Suivant(P)
|     Fin Si;
| Fin TQ;
| Si (Trouv=Vrai) Alors
|     | Ecrire(Val,'Existe dans la liste')
| Sinon
|     | Ecrire(Val,'n"existe pas dans la liste'
| Fin Si;
Fin.
```

3.1.6 Listes linéaires chaînées bidirectionnelles

C'est une LLC où chaque maillon contient à la fois l'adresse de l'élément précédent et l'adresse de l'élément suivant ce qui permet de parcourir la liste dans les deux sens.



Déclaration

Type TMaillon = **Structure**
Valeur : **Typeqq ;** // **désigne un type quelconque**
Précédent, Suivant : **Pointeur(TMaillon) ;**
Fin ;
Var Tete : **TMaillon ;**

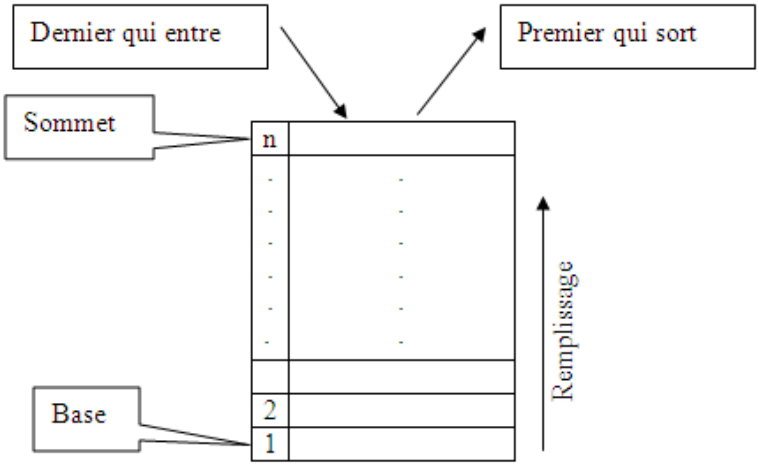
Modèle sur les listes linéaires chaînées bidirectionnelles

- **Allouer :** Création d'un maillon.
- **Libérer :** Libérer un maillon.
- **Valeur(P) :** retourne la valeur du champ val du maillon pointé par P .
- **Suivant(P) :** retourne le pointeur se trouvant dans le champs suivant du maillon pointé par P
- **Précédent(P) :** retourne le pointeur se trouvant dans le champs suivant du maillon pointé par P
- **Aff_Val(P,x) :** Ranger la valeur de x dans le champs val du maillon pointé par P
- **Aff_Adr_Précedent(P,Q) :** Ranger Q dans le champs précédent du maillon pointé par P
- **Aff_Adr_Suivant(P,Q) :** Ranger Q dans le champs suivant du maillon pointé par P

3.2 Les piles (stacks)

3.2.1 Définition

Une pile est une liste ordonnée d'éléments où les insertions et les suppressions d'éléments se font à une seule et même extrémité de la liste appelée *le sommet de la pile.*
Le principe d'ajout et de retrait dans la pile s'appelle LIFO (*Last In First Out*) : "le dernier qui entre est le premier qui sort"



3.2.2 Utilisation des piles

3.2.2.1 Dans un navigateur web

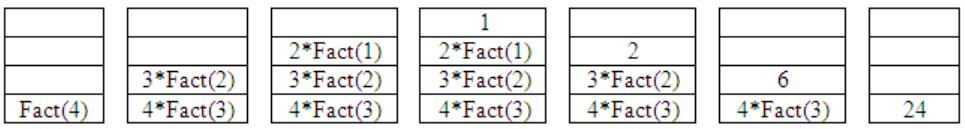
Une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton "Afficher la page précédente".

3.2.2.2 Annulation des opérations

La fonction "Annuler la frappe" d'un traitement de texte mémorise les modifications apportées au texte dans une pile.

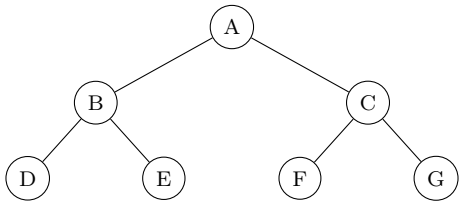
3.2.2.3 Gestion des appels dans l'exécution des programmes

Fact(4)=4*Fact(3)=4*3*Fact(2)=4*3*2*Fact(1)=4*3*2*1=4*3*2=4*6=24



3.2.2.4 Parcours en profondeur des arbres

Soit l'arbre suivant :



L'algorithme de parcours en profondeur est le suivant :

```
Mettre la Racine dans la pile ;
Tant que (La pile n'est pas vide) faire
    Retirer un noeud de la pile ;
    Afficher sa valeur ;
    Si (Le noeud a des fils) Alors
        Ajouter ces fils à la pile
    Fin Si;
Fin TQ;
```

Le résultat de parcours en profondeur affiche : A, B, D, E, C, F, G.

3.2.2.5 Evaluation des expressions postfixées

Pour l'évaluation des expressions arithmétiques ou logiques, les langages de programmation utilisent généralement les représentation préfixée et postfixée. Dans la représentation postfixée, on représente l'expression par une nouvelle, où les opérations viennent toujours après les opérandes.

Exemple

L'expression $((a + (b * c))/(c - d)$ est exprimée, en postfixé, comme suit : $bc * a + cd - /$
Pour l'évaluer, on utilise une pile. On parcourt l'expression de gauche à droite, en exécutant l'algorithme suivant :

$i \leftarrow 1$;

Tant que ($i < Longeur(Expression)$) **faire**

  Si ($Expression[i]$ est un Opérateur) **Alors**

    Retirer deux éléments de la pile ;

    Calculer le résultat selon l'opérateur ;

    Mettre le résultat dans la pile ;

  Sinon

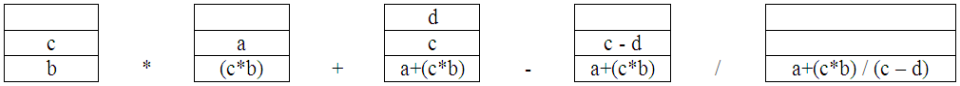
    Mettre l'opérande dans la pile ;

  Fin Si;

  $i \leftarrow i + 1$;

Fin TQ;

Le schéma suivant montre l'évolution du contenu de la pile, en exécutant cet algorithme sur l'expression précédente.



3.2.3 Opérations sur les piles

Les opérations habituelles sur les piles sont :

- Initialisation de la pile (généralement à vide)
- Vérification du contenu de la pile (pile pleine ou vide)
- Dépilement (POP) : retirer un élément du sommet de la pile si elle n'est pas vide
- Empilement (PUSH) : ajout d'un élément au sommet de la pile si elle n'est pas saturée.

Exercice : Donner l'état de la pile après l'exécution des opérations suivantes sur une pile vide :

Empiler(a), Empiler(b), Dépiler, Empiler(c), Empiler(d), Dépiler, Empiler(e), Dépiler, Dépiler.

3.2.4 Implémentation des piles

Les piles peuvent être représentés en deux manières : par des tableaux ou par des LLCs :

3.2.4.1 Implémentation par des tableaux

L'implémentation statique des piles utilise les tableaux. Dans ce cas, la capacité de la pile est limitée par la taille du tableau. L'ajout à la pile se fait dans le sens croissant des indices, tandis que le retrait se fait dans le ses inverse.

3.2.4.2 Implémentation par des LLCs

L'implémentation dynamique utilise les listes linéaires chinées. Dans ce cas, la pile peut être vide, mais ne peut être jamais pleine, sauf bien sur en cas d'insuffisance de l'espace mémoire. L'empilement et le dépilement dans les piles dynamiques se font à la tête de la liste.

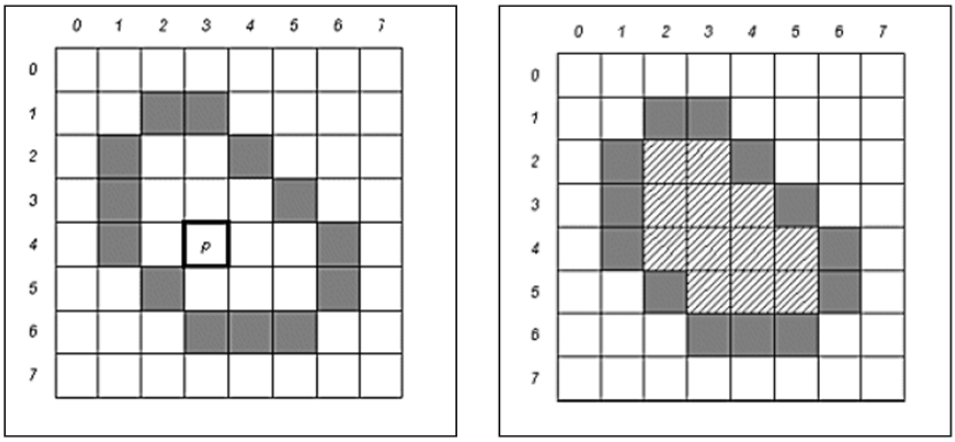
Les deux algorithmes "PileParTableaux" et "PileParLLCs" suivant présentent deux exemples d'implémentation statique et dynamique des piles.

```
Algorithme PileParTableaux;
Var  Pile : Tableau[1..n] de entier ;      Sommet : Entier ;
Procédure InitPile();
Début
    | Sommet ← 0 ;
Fin;
Fonction PileVide() : Booleen;
Début
    | PileVide ← (Sommet = 0) ;
Fin;
Fonction PilePleine() : Booleen;
Début
    | PilePleine ← (Sommet = n) ;
Fin;
Procédure Empiler( x : entier);
Début
    Si (PilePleine) Alors
        | Ecrire('Impossible d'empiler, la pile est pleine !!')
    Sinon
        | Sommet ← Sommet + 1 ;
        | Pile[Sommet] ← x ;
    Fin Si;
Fin;
Procédure Dépiler( x : entier);
Début
    Si (PileVide) Alors
        | Ecrire('Impossible de dépiler, la pile est vide !!')
    Sinon
        | x ← Pile[Sommet] ;
        | Sommet ← Sommet − 1 ;
    Fin Si;
Fin;

Début
| ... Utilisation de la pile ...
Fin.
```

```
Algorithme PileParLLCs;
Type  TMaillon = Structure
    Valeur : entier ;
    Suivant : Pointeur(TMaillon) ;
Fin ;
Var  P, Sommet : Pointeur(TMaillon) ;
Procédure InitPile();
Début
    | Sommet ← Nil ;
Fin;
Fonction PileVide() : Booleen;
Début
    | PileVide ← (Sommet = Nil) ;
Fin;
Procédure Empiler( x : entier);
Début
    | Allouer(P) ;          Aff_Val(P,x) ;
    | Aff_Adr(P,Sommet) ;   Sommet ← P ;
Fin;
Procédure Dépiler( x : entier);
Début
    | Si (PileVide) Alors
    |     | Ecrire('Impossible de dépiler, la pile est vide !!')
    | Sinon
    |     | x ← Valeur(Sommet) ;          P ← Sommet ;
    |     | Sommet ← Suivant(Sommet) ;    Libérer(P) ;
    | Fin Si;
Fin;

Début
| ... Utilisation de la pile ...
Fin.
```

3.2.5 Exemple d'application : Remplissage d'une zone d'une image

Une image en informatique peut être représentée par une matrice de points $'Image'$ ayant M colonnes et N lignes. Un élément $Image[x, y]$ de la matrice représente la couleur du point p de coordonnées (x, y). On propose d'écrire ici une fonction qui, à partir d'un point p, *étale* une couleur c autour de ce point. La progression de la couleur étalée s'arrête quand elle rencontre une couleur autre que celle du point p. La figure suivante illustre cet exemple, en considérant $p = (3, 4)$.



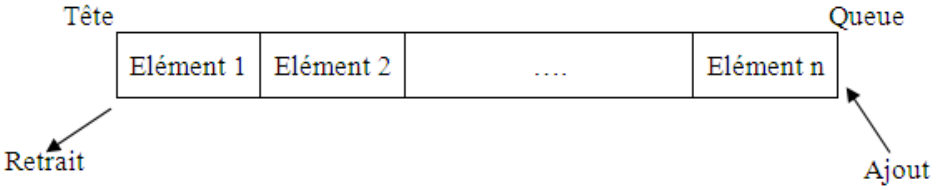
Pour effectuer le remplissage, on doit aller dans toutes les directions à partir du point p. Ceci ressemble au parcours d'un arbre avec les noeuds de quatre fils. La procédure suivante permet de résoudre le problème en utilisant une pile.

```
Procédure Remplir( Image : Tableaux[0..M−1, 0..N−1] de couleur ; x, y : entier ;
c : couleur);
Var c1 : couleur ;
Début
    c1 ← Image[x, y] ;
    InitPile ;
    Empiler((x, y)) ;
    Tant que (¬ PileVide) faire
        Depiler((x,y)) ;
        Si (Image[x, y] = c1) Alors
            Image[x, y] ← c ;
            Si (x > 0) Alors
                Empiler((x − 1, y))
            Fin Si;
            Si (x < M − 1) Alors
                Empiler((x + 1, y))
            Fin Si;
            Si (y > 0) Alors
                Empiler((x, y − 1))
            Fin Si;
            Si (y < N − 1) Alors
                Empiler((x, y + 1))
            Fin Si;
        Fin Si;
    Fin TQ;
Fin;
```

3.3 Les Files d'attente (Queues)

3.3.1 Définition

La file d'attente est une structure qui permet de stocker des objets dans un ordre donné et de les retirer dans le même ordre, c'est à dire selon le protocole FIFO *'first in first out'*. On ajoute toujours un élément en queue de liste et on retire celui qui est en tête.



3.3.2 Utilisation des files d'attente

Les files d'attente sont utilisées, en programmation, pour gérer des objets qui sont en attente d'un traitement ultérieur, tel que la gestion des documents à imprimer, des programmes à exécuter, des messages reçus,...etc. Elles sont utilisées également dans le parcours des arbres.

Exercice

Reprendre le parcours de l'arbre de la section 3.2.2.4 (parcours en profondeur) en utilisant une file d'attente au lieu de la pile.

3.3.3 Opérations sur les files d'attente

Les opérations habituelles sur les files sont :

- Initialisation de la file
- Vérification du contenu de la file (vide ou pleine)
- Enfilement : ajout d'un élément à la queue de la file
- Défilement : retrait d'un élément de la tête de la file

3.3.4 Implémentation des files d'attente

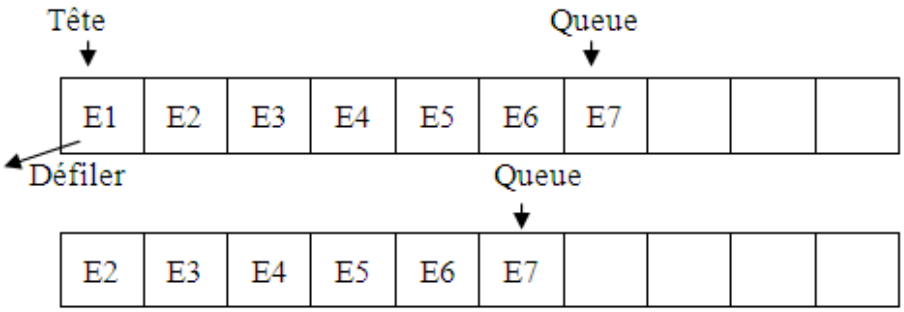
De même que pour les piles, les files d'attente peuvent être représentées en deux manières :

- par représentation statique en utilisant les tableaux,
- par représentation dynamique en utilisant les listes linéaires chaînées.

3.3.4.1 Implémentation statique

L'implémentation statique peut être réalisée par décalage en utilisant un tableau avec une tête fixe, toujours à 1, et une queue variable. Elle peut être aussi réalisée par flot utilisant un tableau circulaire où la tête et la queue sont toutes les deux variables.

1. Par décalage

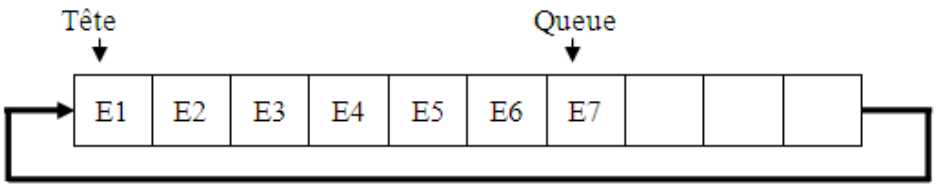


→ La file est vide si $Queue = 0$

→ La file est pleine si $Queue = n$

↓ Problème de décalage à chaque défilement

2. Par flot : La file est représentée par un tableau circulaire



→ La file est vide si $Tête = Queue$

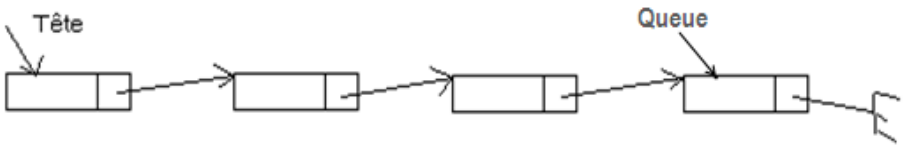
→ La file est pleine si $(Queue + 1)\ mod\ n = Tête$

↓ On sacrifie une case pour distinguer le cas d'une file vide de celui d'une file pleine.

Exercice Pensez à une solution qui évite le sacrifice d'une case.

3.3.4.2 Implémentation dynamique

La représentation dynamique utilise une liste linéaire chaînée. L'enfilement se fait à la tête de la liste et de défilement se fait de la queue. La file d'attente, dans ce cas, peut devenir vide, mais ne sera jamais pleine.



Les deux algorithmes FileParFlot et FileParLLCs, à la fin de cette section, présentent des exemples d'implémentation, respectivement, statique et dynamique.

3.3.5 File d'attente particulière (File d'attente avec priorité)

Une file d'attente avec priorité est une collection d'éléments dans laquelle l'insertion ne se fait pas toujours à la queue. Tout nouvel élément est inséré, dans la file, selon sa priorité. Le retrait se fait toujours du début.
Dans une file avec priorité, un élément prioritaire prendra la tête de la file même s'il arrive le dernier. Un élément est toujours accompagné d'une information indiquant sa priorité dans la file.
L'implémentation de ces files d'attente peut être par tableau ou listes, mais l'implémentation la plus efficace et la plus utilisée utilise des arbres particuliers qui s'appellent 'les tas'.

Algorithme FileParFlot;

Var File : **Tableau[1..n] de entier ;** Tête, Queue : **Entier ;**

Procédure InitFile();

Début

$Tête \leftarrow 1$; $Queue \leftarrow 1$;

Fin;
Fonction FileVide() : **Booleen**;

Début

$FileVide \leftarrow (Tête = Queue)$;

Fin;
Fonction FilePleine() : **Booleen**;

Début

$FilePleine \leftarrow (((Queue + 1)\ mod\ n) = Tête)$;

Fin;
Procédure Enfiler(x : **entier**);

Début

Si (FilePleine) **Alors**

Ecrire('Impossible d"enfiler, la file est pleine ! !')

Sinon

$File[Queue] \leftarrow x$;

$Queue \leftarrow (Queue + 1)\ mod\ n$;

Fin Si;

Fin;
Procédure Défiler(x : **entier**);

Début

Si (FileVide) **Alors**

Ecrire('Impossible de défiler, la file est vide ! !')

Sinon

$x \leftarrow File[Tete]$;

$Tête \leftarrow (Tête + 1)\ mod\ n$;

Fin Si;

Fin;

Début

... Utilisation de la File ...

Fin.

```
Algorithme FileParLLCs;
Type  TMaillon = Structure
    Valeur : entier ;
    Suivant : Pointeur(TMaillon) ;
Fin ;
Var  P, Tête, Queue : Pointeur(TMaillon) ;
Procédure InitFile();
Début
    | Tête ← Nil ;        Queue ← Nil ;
Fin;
Fonction FileVide() : Booleen;
Début
    | FileVide ← (Tête = Nil) ;
Fin;
Procédure Enfiler( x : entier);
Début
    | Allouer(P) ;          Aff_Val(P,x) ;
    | Aff_adr(P,Nil) ;
    | Si (Queue = Nil) Alors
    |     | Tête ← P ;
    | Sinon
    |     | Aff_Adr(Queue,P) ;
    | Fin Si;
    | Queue ← P ;
Fin;
Procédure Défiler( x : entier);
Début
    | Si (FileVide) Alors
    |     | Ecrire('Impossible de défiler, la file est vide !!')
    | Sinon
    |     | x ← Valeur(Tete) ;        P ← Tête ;
    |     | Tête ← Suivant(Tête) ;        Libérer(P) ;
    | Fin Si;
Fin;

Début
| ... Utilisation de la File ...
Fin.
```

3.4 Exercices

1. Listes linéaires chaînées

1. **Algorithmes de base**
   - Ecrire les algorithmes de base suivants sur les listes linéaires chaînées unidirectionnelles :
     (a) Suppression par valeur et par position.
     (b) Inversement d'une liste
         - En créant une nouvelle liste
         - En Inversant le chainage dans la même liste
     (c) Tri par la méthode "sélection et permutation"
     (d) Tri par la méthode des bulles
     (e) Recherche de l'élément qui a le plus grand nombre d'occurrences
   - Ecrire les algorithmes de base suivants sue les listes linéaires chaînées bidirectionnelles :
     (a) Recherche d'un élément
     (b) Suppression d'un élément
     (c) Inversement d'une liste
2. **Fonctions LISP**

   Soit une liste linéaire chaînée contenant des nombres entiers et dont la tête est L :
   (a) Ecrire la fonction CAR(L) qui retourne la valeur du premier élément de la liste.
   (b) Ecrire la fonction CDR(L) qui retourne la liste sans le premier élément.
   (c) Ecrire la fonction CONS(x, L) qui retourne une liste dont le premier élément est x et le reste est la liste L.
   (d) Ecrire la fonction Triée(L) qui retourne vrai si la liste L est triée dans l'ordre croissant et faux sinon.
   (e) Ecrire la fonction Fusion(L1, L2) qui prend deux listes triées dans l'ordre croissant L1 et L2 et retourne une liste triée, dans le même ordre, contenant les deux listes et cela en utilisant les fonctions précédentes.
3. **Différence de deux listes linéaires chaînées**

   Soient L_1 et L_2 deux listes linéaires chaînées unidirectionnelle. Ecrire la procédure

qui permet de construire la liste $L = L_1 - L_2$ contenant tous les éléments appartenant à L_1 et n'appartenant pas à L_2.

4. **Liste de nombre premiers**

   Nous considérons la construction d'une liste des nombres premiers inférieurs ou égaux à un entier n donnée. Pour construire cette liste, on commence, dans une première phase, par y ajouter tous les entiers de 2 à n en commençant par le plus grand et en terminant par le plus petit qui se trouvera à la tête de la liste. On considère ensuite successivement les éléments de la liste dans l'ordre croissant en on supprime tous leurs multiples stricts. Ecrire cet algorithme.

5. **Listes de caractères**

   Nous considérons la représentation d'une chaîne de caractères sous forme de LLC où chaque maillon contient un caractère.

   Ecrire la fonction qui vérifie qu'une liste est un palindrome ou non dans le cas de :

   (a) Liste doublement chaînée.

   (b) Liste dans un seul sens.

   Une liste est dite palindrome si elle est de la forme $a_1 a_2 \cdots a_n b a_n \cdots a_2 a_1$

6. **Matrices creuses**

   Une matrice est dite creuse lorsque le nombre d'éléments nuls y figurant est très supérieur à celui des éléments non nuls. On peut représenter une matrice creuse en ne tenant compte que des éléments non nuls. Chaque ligne de la matrice est une liste linéaire chaînée ordonnée (selon le rang de la colonne) des éléments non nuls. Une table de N éléments (N étant le nombre de lignes de la matrice) donne les adresses de tête de chacune des listes. Un élément de la liste contient l'indice de la colonne et la valeur de l'élément.

   (a) Donner la déclaration de la structure de données nécessaire.

   (b) Ecrire la procédure permettant de Remplir une telle structure à partir d'une matrice A(M, N) donnée.

   (c) Ecrire la procédure permettant de calculer la somme de deux matrices ainsi représentées.

   Sachant qu'un élément de la matrice occupe 4 octets, qu'un pointeur occupe 2 octets et qu'un indice occupe 2 octets,

   (a) quel est le gain d'espace quand le nombre de zéros est de μ

(b) quelle valeur minimale attribuée à μ pour que l'on puisse opter pour une telle représentation

7. **Représentation d'un nombre binaire par une LLC**

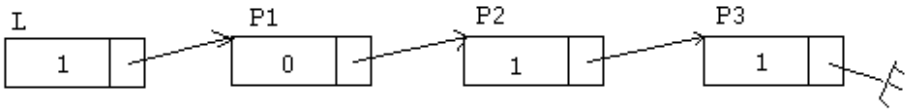
On convient de représenter un nombre binaire $b_1 b_2..b_n$, où chaque b_i est un 0 ou un 1, par une liste linéaire chaînée où chaque élément contient un b_i.

Considérons la procédure récursive suivante :

- L est une liste linéaire chaînée représentant un nombre binaire,
- R est une variable globale initialisée à 1,
- A mod B désigne le reste de la division de A par B.

```
Procédure P( L : Pointeur(TMaillon));
Var  x : entier ;
Début
    Si ( L ≠ Nil) Alors
        P(Suivant(L)) ;
        x ← Valeur(L) + R ;
        Aff_val(L, x mod 2) ;
        Si (x=2) Alors
            R := 1 ;
        Sinon
            R := 0 ;
        Fin Si;
    Fin Si;
Fin;
```

Donner les différentes valeurs de L et x, dans l'ordre, après l'appel récursif pour la liste suivante représentant le nombre 1011 :



- Que fait cette procédure ?

– Ecrire un algorithme qui utilise cette procédure pour incrémenter un nombre binaire ainsi représenté.

8. **Interclassement de n listes linéaires chaînées**

   Soit T un tableau de n listes linéaires chaînées ordonnées.

   (a) Transformer T en une Liste L de listes sans création de nouvelles listes

   (b) Donner la procédure qui affiche tous les éléments de L.

   (c) Créer une liste ordonnée LD contenant tous les éléments de T en parcourant parallèlement les n listes de T (Interclassement)

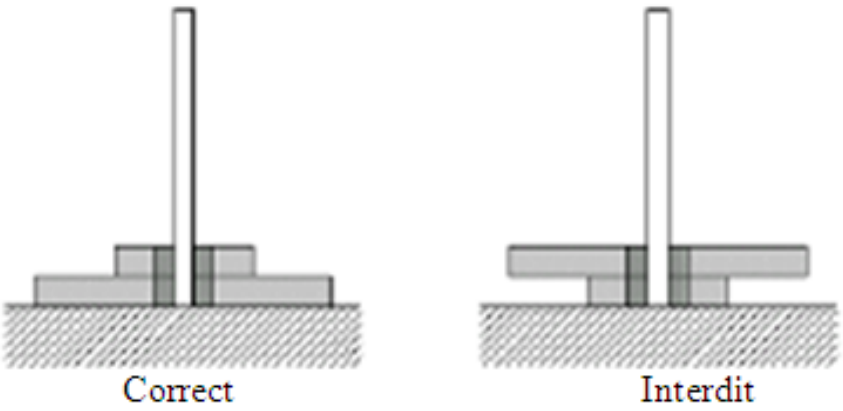
   (d) Eliminer les doublons dans LD.

2. Piles & Files

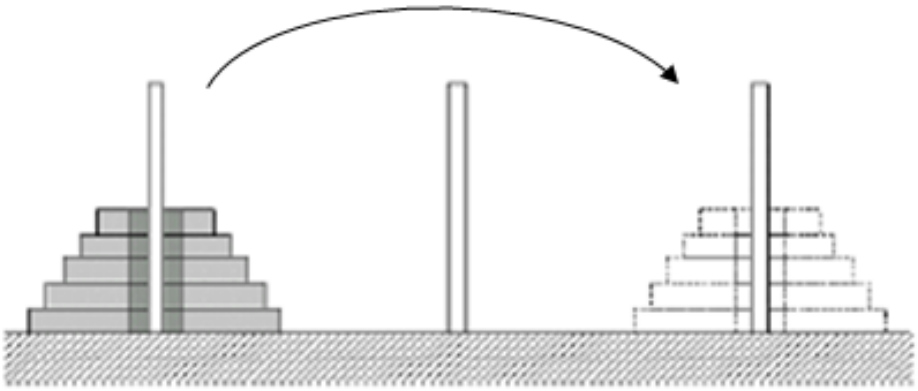
1. Ecrire un algorithme qui permette d'afficher les éléments d'une liste linéaire chaînée dans l'ordre inverse en utilisant une pile.

2. **La tour de Hanoi**

   Il s'agit d'un jeu de réflexion dont voici le principe. Des anneaux de diamètres différents sont empilés sur un poteau. Un anneau peut être empilé sur un autre seulement si il a un diamètre inférieur à celui de l'anneau sur lequel il repose.



   Le but du jeu est de déplacer n anneaux initialement empilés sur un seul poteau vers un autre en respectant les règles du jeu et en n'utilisant qu'un seul poteau intermédiaire.



Ecrire un algorithme qui permette d'afficher les étapes nécessaires pour le déplacement de n anneaux d'un poteau A vers un poteau C en passant par le poteau intermédiaire B en utilisant trois piles.

3. **Evaluation d'une expression arithmétique post-fixée**

   Une expression arithmétique post-fixée est une expression où les opérandes sont placés avant les opérateurs.

   Exemple : l'expression ((a + b + c) * (a * b / c)) / c est exprimée comme suit :

   ab+c+ab*c/*c/

   (a) Représenter les expressions suivantes sous forme post-fixée :
   - a + b, (a + b) / d
   - ((c + d) + (d - e)) + 5
   - - (a + b) + (5 + b) c
   - ((a + b) + (c - d)) / 5

   (b) Donner l'algorithme qui évalue une expression arithmétique post-fixée. On suppose que cette dernière se trouve dans un tableau dont les élément sont de type : (Valeur, Type (opérateur ou opérande)).

4. Une file d'attente avec priorité est une file où les éléments sont caractérisés par une priorité de service : un élément de priorité supérieure est servi même s'il n'est pas arrivé le premier.

   (a) Décrire les structures nécessaires à l'implémentation de ce modèle en utilisant les listes.

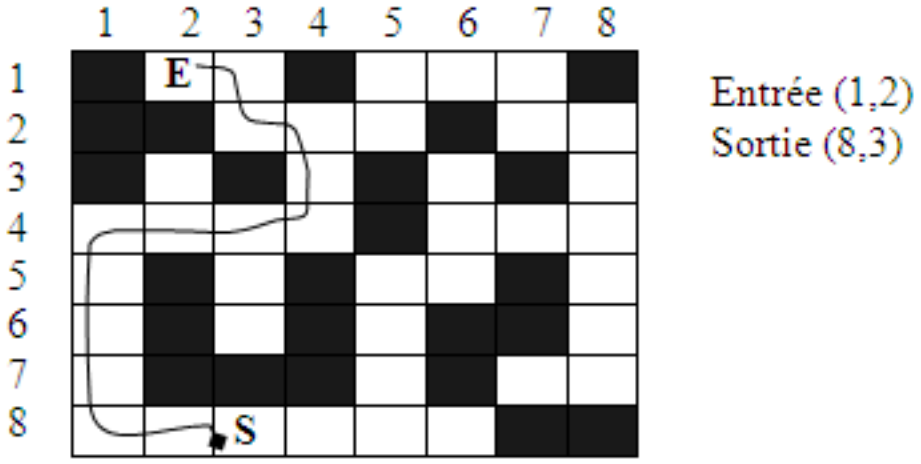
   (b) Ecrire les procédures Enfiler et Défiler de ce modèle.

   (c) Expliquer comment peut-on implémenter une pile à l'aide d'une file d'attente avec priorité.

(d) Expliquer comment peut-on implémenter une file d'attente ordinaire à l'aide d'une file d'attente avec priorité.

5. Ecrire un algorithme qui permette d'implémenter le modèle des files d'attente utilisant deux piles.

6. Soit le labyrinthe suivant :



(a) Décrire les structures de données nécessaires à la représentation d'un labyrinthe $N \times N$ en mémoire.

(b) Ecrire la procédure qui vérifie s'il existe un chemin allant d'une entrée donnée vers une sortie donnée en utilisant une pile.

Chapitre 4

Structures Hiérarchiques

4.1 Les arbres

4.1.1 Introduction

Dans les tableaux nous avons :

+ Un accès direct par indice (rapide)

- L'insertion et la suppression nécessitent des décalages

Dans les listes linéaires chaînées nous avons :

+ L'insertion et la suppression se font uniquement par modification de chaînage

- Accès séquentiel lent

Les arbres représentent un compromis entre les deux :

+ Un accès relativement rapide à un élément à partir de sa clé

- Ajout et suppression non coûteuses

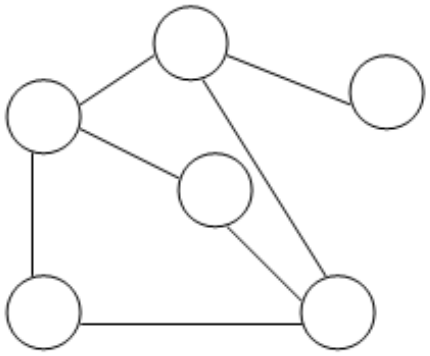
En plus plusieurs traitements en informatique sont de nature arborescente tel que les arbres généalogiques, hiérarchie des fonctions dans une entreprise, représentation des expressions arithmétiques,.. etc.

4.1.2 Définitions

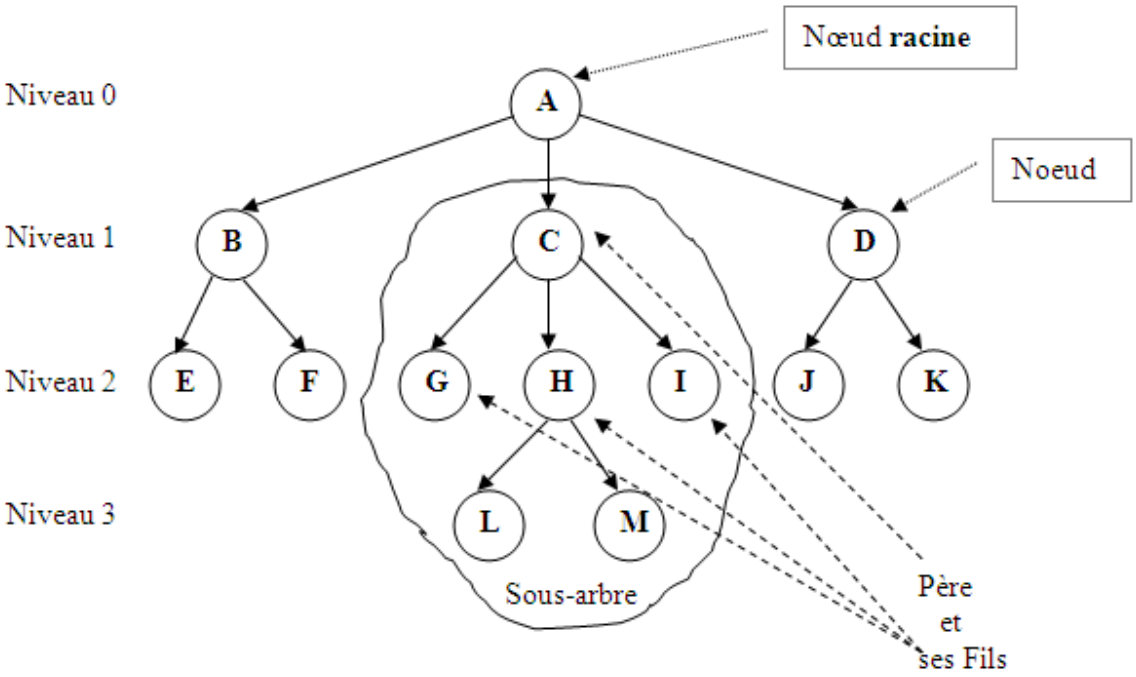
4.1.2.1 Définition d'un arbre

Un arbre est une structure non linéaire, c'est un graphe sans cycle où chaque nœud a au plus un prédécesseur.

Graphe



Arbre



- Le prédécesseur s'il existe s'appelle père (père de C = A, père de L = H)
- Le successeur s'il existe s'appelle fils (fils de A = { B,C,D }, fils de H= {L,M })
- Le nœud qui n'a pas de prédécesseur s'appelle racine (A)
- Le nœud qui n'a pas de successeur s'appelle feuille (E,F,G,L,J,...)
- Descendants de C={G,H,I,L,M}, de B={E,F},...
- Ascendants de L={H,C,A t}, E={B,A},...

4.1.2.2 Taille d'un arbre

C'est le nombre de nœuds qu'il possède.

- Taille de l'arbre précédent = 13
- Un arbre vide est de taille égale à 0.

4.1.2.3 Niveau d'un nœud

- Le niveau de la racine = 0
- Le niveau de chaque nœud est égale au niveau de son père plus 1
- Niveau de E,F,G,H,I,J,K = 2

4.1.2.4 Profondeur (Hauteur) d'un arbre

- C'est le niveau maximum dans cet arbre.
- Profondeur de l'arbre précédent = 3

4.1.2.5 Degré d'un nœud

- Le degré d'un nœud est égal au nombre de ses fils.
- Degré de (A = 3, B =2, C = 3, E= 0, H=2,...)

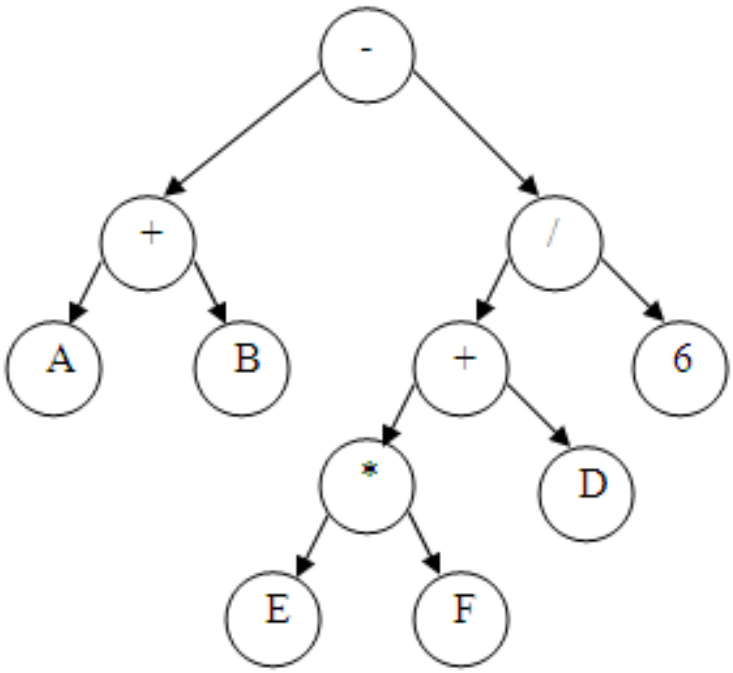
4.1.2.6 Degré d'un arbre

- C'est le degré maximum de ses nœuds.
- Degré de l'arbre précédent = 3.

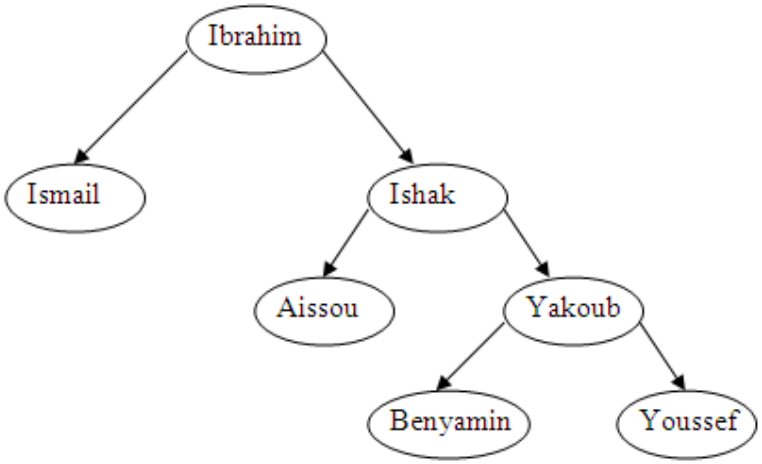
4.1.3 Utilisation des arbres

- Représentation des expressions arithmétiques

(A+B)*c - (d+E*f)/6



– Représentation d'un arbre généalogique



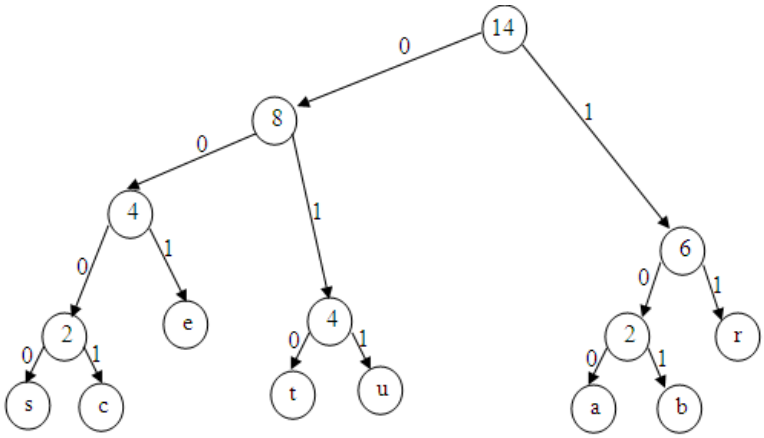
– Codage

Exemple : coder la chaine "structure arbre"

1. Construite la table des fréquences des caractères

Caractère	Fréquence	Caractère	Fréquence
s	1	c	1
t	2	e	2
r	4	a	1
u	2	b	1

2. Construite l'arbre des codes



3. Construire la table des codes

Caractère	Code	Caractère	Code
s	0000	c	0001
t	010	e	001
r	11	a	100
u	011	b	101

4. Coder la chaine :

   "structure arbre" ⇒ 0000 010 11 011 0001 010 011 11 001 100 11 101 11 001

4.1.4 Implémentation des arbre

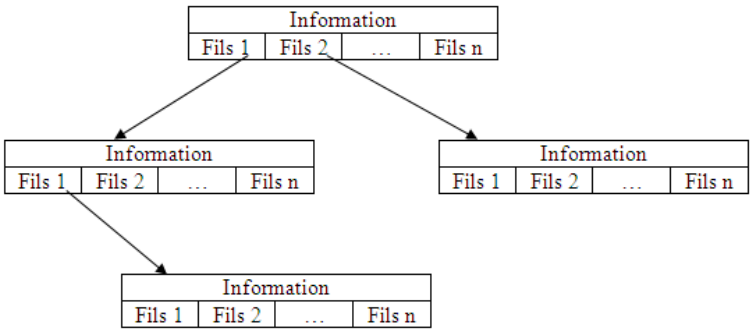
Les arbres peuvent être représentés par des tableaux, des listes non linéaires ou tous les deux :

4.1.4.1 Représentation statique

L'arbre du premier exemple peut être représenté par un tableau comme suit :

Num	Information	Fils 1	Fils 2	Fils 3
1	A	2	3	4
2	B	5	6	0
3	C	7	8	9
4	D	10	11	0
5	E	0	0	0
6	F	0	0	0
7	G	0	0	0
8	H	12	13	0
9	I	0	0	0
10	J	0	0	0
11	K	0	0	0
12	L	0	0	0
13	M	0	0	0

4.1.4.2 Représentation dynamique




```
Type  Tnœud = Structure
        Info : typeqq ;
        Fils : Tableau[1..NbFils] de Pointeur(Tnœud) ;
Fin ;
Var  Racine : Pointeur(Tnœud) ;
```

4.1.5 Modèle sur les arbres

Pour manipuler les structures de type arbre, on aura besoin des primitives suivantes :

- **Allouer (N) :** créer une structure de type Tnœud et rendre son adresse dans N.
- **Liberer(N) :** libérer la zone pointée par N.
- **Aff_Val(N, Info) :** Ranger la valeur de Info dans le champs info du nœud pointé par N.
- Aff_Fils_i**(N1,N2) :** Rendre N2 le fils numéro i de N1.
- $Fils_i$**(N) :** donne le fils numéro i de N.
- **Valeur(N) :** donne le contenu du champs info du nœud pointé par N.

4.1.6 Traitements sur les arbres

4.1.6.1 Parcours des arbres

Le parcours d'un arbre consiste à passer par tous ses nœuds. Les parcours permettent d'effectuer tout un ensemble de traitement sur les arbres. On distingue deux types de parcours :

1. Parcours en profondeur

   Dans un parcours en profondeur, on descend le plus profondément possible dans l'arbre puis, une fois qu'une feuille a été atteinte, on remonte pour explorer les autres branches en commençant par la branche "la plus basse" parmi celles non encore parcourues. L'algorithme est le suivant :

```
Procédure PP( nœud : Pointeur(Tnœud));
Début
    Si (nœud ≠ Nil) Alors
        Pour i de 1 à NbFils faire
            PP(Fils_i(nœud))
        Fin Pour;
    Fin Si;
Fin;
```

   Le parcours en profondeur peut se faire en deux manière :

   – Parcours en profondeur Prefixe : où on affiche le père avant ses fils

   – Parcours en profondeur Postfixe : où on affiche les fils avant leur père.

   Les algorithmes récursifs correspondant sont les suivants :

```
Procédure PPPrefixe( nœud : Pointeur(Tnœud));
Début
    Si (nœud ≠ Nil) Alors
        Afficher(Valeur(nœud)) ;
        Pour i de 1 à NbFils faire
            PPPrefixe(Fils_i(nœud))
        Fin Pour;
    Fin Si;
Fin;
```

```
Procédure PPPostfixe( nœud : Pointeur(Tnœud));
Début
    Si (nœud ≠ Nil) Alors
        Pour i de 1 à NbFils faire
            PPPostfixe(Fils_i(nœud))
        Fin Pour;
        Afficher(Valeur(nœud)) ;
    Fin Si;
Fin;
```

Le parcours en profondeur préfixe de l'arbre du premier exemple donne :

A,B,E,F,C,G,H,L,M,I,D,J,K

Tandis que le parcours en profondeur postfixe donne :

E,F,B,G,L,M,H,I,C,J,K,D,A

2. Parcours en largeur

   Dans un parcours en largeur, tous les nœuds à une profondeur i doivent avoir été visités avant que le premier nœud à la profondeur i + 1 ne soit visité. Un tel parcours nécessite que l'on se souvienne de l'ensemble des branches qu'il reste à visiter. Pour ce faire, on utilise une file d'attente.

```
Procédure PL( nœud : Pointeur(Tnœud));
Var  N : Pointeur(Tnœud) ;
Début
    Si (nœud ≠ Nil) Alors
        InitFile ; Enfiler(Neuud) ;
        Tant que (Non(FileVide)) faire
            Défiler(N) ;
            Afficher(Valeur(N)) ;
            Pour i de 1 à NbFils faire
                Si (Fils_i(N) ≠ Nil) Alors
                    Enfiler(Fils_i(N)) ;
                Fin Si;
            Fin Pour;
        Fin TQ;
    Fin Si;
Fin;
```

L'application de cet algorithme sur l'arbre du premier exemple donne

A,B,C,D,E,F,G,H,I,J,K,L,M

4.1.6.2 Recherche d'un élément

```
Fonction Rechercher( nœud : Pointeur(Tnœud) ; Val : Typeqq) : Booleen;
Var  i : entier ;
     Trouv : Booleen ;
Début
    Si (nœud = Nil) Alors
        Rechercher ← Faux ;
    Sinon
        Si (Valeur(nœud)=Val) Alors
            Rechercher ← Vrai ;
        Sinon
            i ← 1 ;
            Trouv ← Faux ;
            Tant que ((i ≤ NbFils) et non Trouv) faire                    ;
                Trouv ← Rechercher(Fils_i(Neoud), Val) ;
                i ← i + 1 ;
            Fin TQ;
            Rechercher ← Trouv ;
        Fin Si;
    Fin Si;
Fin;
```

4.1.6.3 Calcul de la taille d'un arbre

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

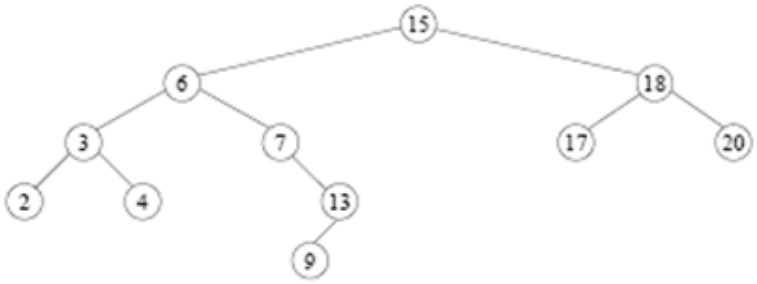
4.1.7 Typologie des arbres

- **Arbre m-aire :** un arbre m-aire d'ordre n est un arbre ou le degré maximum d'un nœud est égal à n.
- **B-Arbre :** Un arbre B d'ordre n est un arbre où :
  - la racine a au moins 2 fils
  - chaque nœud, autre que la racine, a entre n/2 et n fils
  - tous les nœuds feuilles sont au même niveau
- **Arbre binaire :** c'est un arbre ou le degré maximum d'un nœud est égal à 2.
- **Arbre binaire de recherche :** c'est un arbre binaire où la clé de chaque nœud est supérieure à celles de ses descendants gauche, et inférieure à celles de ses descendants droits.

4.2 Les arbres binaires de recherche

4.2.1 Définition

Les arbres binaires de recherche sont utilisés pour accélérer la recherche dans les arbres m-aires. Un arbre binaire de recherche est un arbre binaire vérifiant la propriété suivante : soient x et y deux nœuds de l'arbre, si y est un nœud du sous-arbre gauche de x, alors $clé(y) \leq clé(x)$, si y est un nœud du sous-arbre droit de x, alors $clé(y) \geq clé(x)$.



Un nœud a, donc, au maximum un fils gauche et un fils droit.

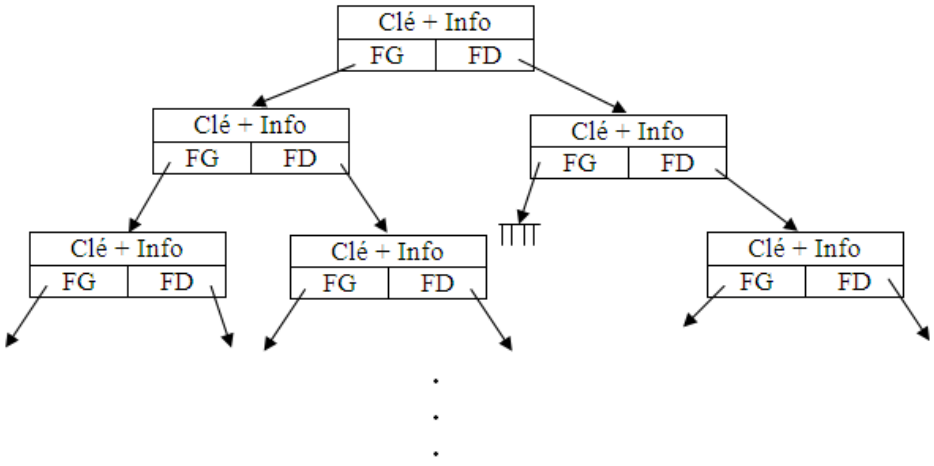
4.2.2 Implémentation des ABR

Les arbres de recherche binaires sont implémentés de la même manière que celles m-aires (statique ou dynamique)

4.2.2.1 Représentation Statique

Num	Information	Fils gauche	Fils droit
1	15	2	3
2	6	4	5
3	18	6	7
4	3	8	9
5	7	0	10
6	17	0	0
7	20	0	0
8	2	0	0
9	4	0	0
10	13	11	0
11	9	0	0

4.2.2.2 Représentation dynamique




```
Type  TNoeud = Structure
        Clé : entier ;
        Info : typeqq ;
        FG,FD : Pointeur(TNoeud) ;
Fin ;
Var  Racine : Pointeur(TNoeud) ;
```

4.2.3 Modèle sur les ABR

- **Allouer (N) :** créer une structure de type TNoeud et rendre son adresse dans N.
- **Liberer(N) :** libérer la zone pointée par N.
- **Aff_Val(N, Info) :** Ranger la valeur de Info dans le champs info du nœud pointé par N.
- **Aff_Clé(N, Clé) :** Ranger la valeur de Clé dans le champs Clé du nœud pointé par N.
- **Aff_FG(N1,N2) :** Rendre N2 le fils gauche de N1.
- **Aff_FD(N1,N2) :** Rendre N2 le fils droit de N1.
- **FG(N) :** donne le fils gauche de N.
- **FD(N) :** donne le fils droit de N.
- **Valeur(N) :** donne le contenu du champs info du nœud pointé par N.
- **Clé(N) :** donne le contenu du champs Clé du nœud pointé par N.

4.2.4 Traitements sur les ABR

4.2.4.1 Parcours

De même que pour les arbres m-aire le parcours des ARB peut se faire en profondeur ou en largeur :

– En profondeur

```
Procédure PP( noeud : Pointeur(TNoeud));
Début
    Si (nœud ≠ Nil) Alors
        PP(FG(noeud)) ;
        PP(FD(noeud)) ;
    Fin Si;
Fin;
```

Le listage des éléments de l'arbre en profondeur peut se faire en :

– préfixe (préordre) : Pére FG FD,

– infixe (inordre) : FG Père FD,

– postfixe (postordre) : FG FD Père.

```
Procédure PPréfixe( noeud : Pointeur(TNoeud));
Début
    Si (nœud ≠ Nil) Alors
        Ecrire(Valeur(noeud)) ;
        PPréfixe(FG(noeud)) ;
        PPréfixe(FD(noeud)) ;
    Fin Si;
Fin;
```

Trace : 15 6 3 2 4 7 13 9 18 17 20

```
Procédure Infixe( noeud : Pointeur(TNoeud));
Début
    Si (nœud ≠ Nil) Alors
        Infixe(FG(noeud)) ;
        Ecrire(Valeur(noeud)) ;
        Infixe(FD(noeud)) ;
    Fin Si;
Fin;
```

Trace : 2 3 4 6 7 9 13 15 17 18 20

```
Procédure Postfixe( noeud : Pointeur(TNoeud));
Début
    Si (nœud ≠ Nil) Alors
        Postfixe(FG(noeud)) ;
        Postfixe(FD(noeud)) ;
        Ecrire(Valeur(noeud)) ;
    Fin Si;
Fin;
```

Trace : 2 4 3 9 13 7 6 17 20 18 15

– En largeur

```
Procédure PL( noeud : Pointeur(TNoeud));
Var  N : Pointeur(TNoeud) ;
Début
    Si (noeud ≠ Nil) Alors
        InitFile ;
        Enfiler(noeud) ;
        Tant que (Non(FileVide)) faire
            Défiler(N) ;
            Afficher(Valeur(N)) ;
            Si (FG(N) ≠ Nil) Alors
                Enfiler(FG(N)) ;
            Fin Si;
            Si (FD(N) ≠ Nil) Alors
                Enfiler(FD(N)) ;
            Fin Si;
        Fin TQ;
    Fin Si;
Fin;
```

4.2.4.2 Recherche

```
Fonction Rechercher( noeud : Pointeur(TNoeud) ; xClé : entier) : Pointeur(TNoeud);
Var  i : entier ;          Trouv : Booleen ;
Début
    Si ((noeud = Nil) ou Clé(noeud)=xClé)) Alors
        Rechercher ← nœud ;
    Sinon
        Si (Clé(noeud) > xClé ) Alors
            Rechercher ← Rechercher(FG(noeud)) ;
        Sinon
            Rechercher ← Rechercher(FD(noeud)) ;
        Fin Si;
    Fin Si;
Fin;
```

4.2.4.3 Insertion

L'élément à ajouter est inséré là où on l'aurait trouvé s'il avait été présent dans l'arbre. L'algorithme d'insertion recherche donc l'élément dans l'arbre et, quand il aboutit à la conclusion que l'élément n'appartient pas à l'arbre (l'algorithme aboutit sur NIL), il insère l'élément comme fils du dernier nœud visité.

4.2.4.4 Suppression

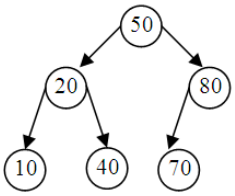
Plusieurs cas de figure peuvent être trouvés : soit à supprimer le nœud N

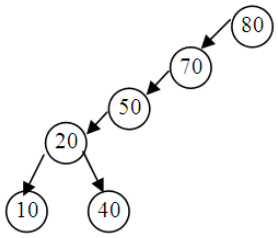
Cas			Action
FG(N)	**FD(N)**	**Exemple**	
Nil	Nil	Feuille (2,4,17)	Remplacer N par Nil
Nil	$\neq$ Nil	7	Remplacer N par FD(N)
$\neq$ Nil	Nil	13	Remplacer N par FG(N)
$\neq$ Nil	$\neq$ Nil	6	1- Rechercher le plus petit descendant à droite de N soit P (7) 2- Remplacer Valeur(N) par Valeur(P) (6 $\leftarrow$7) 3- Remplacer P par FD (P) (7 $\leftarrow$13)

Exercice : Donner l'arbre après la suppression de 3 puis de 15.

4.2.5 Equilibrage

Soit les deux ARB suivants :





Ces deux ARB contiennent les mêmes éléments, mais sont organisés différemment. La profondeur du premier est inférieure à celle du deuxième. Si on cherche l'élément 10 on devra parcourir 3 éléments (50, 20, 10) dans le premier arbre, par contre dans le deuxième, on devra parcourir 5 élément (80, 70, 50, 20,10). On dit que le premier arbre est plus équilibré.

Si on calcule la complexité de l'algorithme de recherche dans un arbre de recherche binaire, on va trouver O(h) ou h est la hauteur de l'arbre. Donc plus l'arbre est équilibré, moins élevée est la hauteur et plus rapide est la recherche.

On dit qu'un ARB est équilibré si pour tout nœud de l'arbre la différence entre la hauteur du sous-arbre gauche et du sous-arbre droit est d'au plus égalé à 1. Il est conseillé toujours de travailler sur un arbre équilibré pour garantir une recherche la plus rapide possible. L'opération d'équilibrage peut être faite à chaque fois qu'on insère un nouveau nœud où à chaque fois que le déséquilibre atteint un certain seuil pour éviter le coût de l'opération d'équilibrage qui nécessite une réorganisation de l'arbre.

4.3 Les tas (Heaps)

4.3.1 Introduction

Pour implémenter une file d'attente avec priorité, souvent utilisée dans les systèmes d'exploitation, on peut utiliser :

- Une file d'attente ordinaire (sans priorité), l'insertion sera alors simple (à la fin) en O(1), mais le retrait nécessitera la recherche de l'élément le plus prioritaire, en O(n).
- Un tableau (ou une liste) trié où le retrait sera en O(1) (le premier élément), mais l'insertions nécessitera O(n).

Les tas apportent la solution à ce problème.

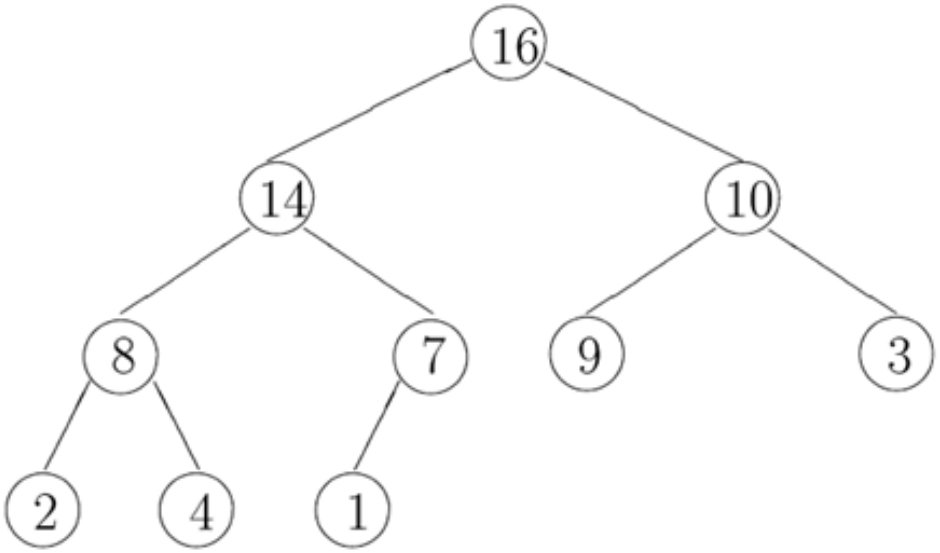
4.3.2 Définition

Un tas (*heap* en anglais) est un arbre qui vérifie les deux propriétés suivantes :

1. C'est un arbre binaire complet c'est-à-dire un arbre binaire dont tous les niveaux sont remplis sauf éventuellement le dernier où les éléments sont rangés le plus à gauche possible.
2. La clé de tout nœud est supérieure à celle de ses descendants.

L'élément le plus prioritaire se trouve donc toujours à la racine.

Exemple d'un tas :



4.3.3 Opérations sur les tas

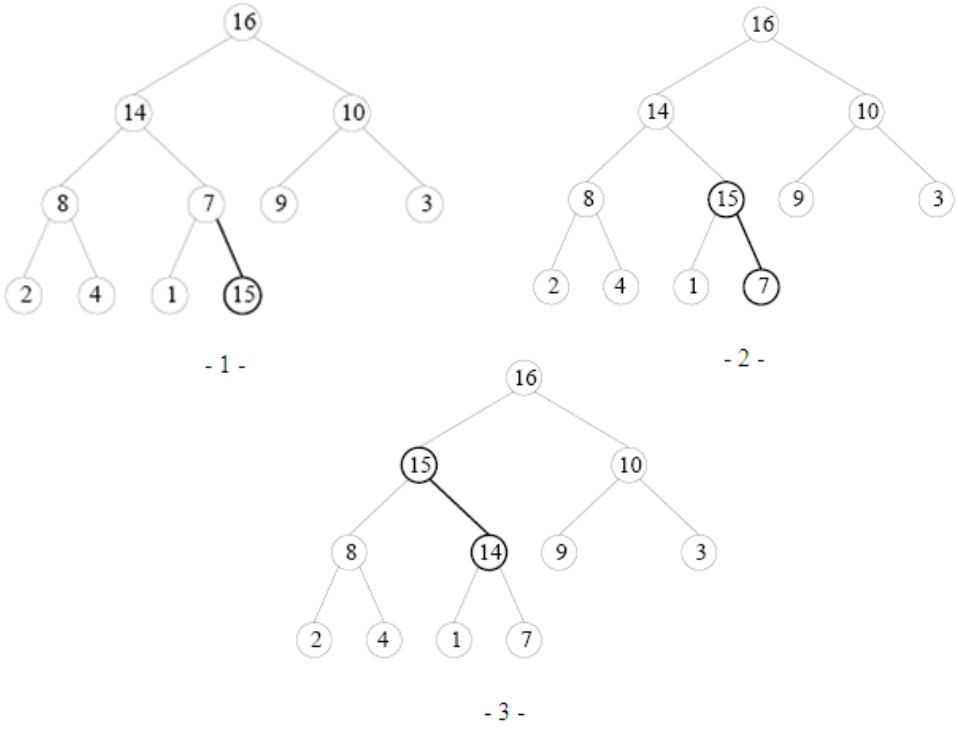
Pour coder une file d'attente avec priorité par un tas, on doit définir les opérations d'ajout et de retrait.

4.3.3.1 Ajout

Pour ajouter un nouvel élément dans la file avec priorité c-à-d dans le tas on doit :

1. Créer un nœud contenant la valeur de cet élément,
2. Attacher ce nœud dans le dernier niveau dans la première place vide le plus à gauche possible (créer un nouveau niveau si nécessaire). On obtient toujours un arbre binaire complet mais pas nécessairement un tas.
3. Comparer la clé du nouveau nœud avec celle de son père et les permuter si nécessaire, puis recommencer le processus jusqu'il n'y ait plus d'éléments à permuter.

Exemple : soit à ajouter l'élément de priorité 15 :



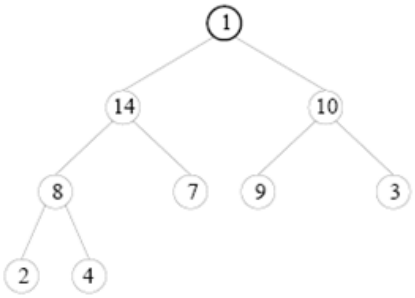
La complexité de cette opération est de $O(h = Log_2(n))$ si n est le nombre d'éléments.

4.3.3.2 Retrait

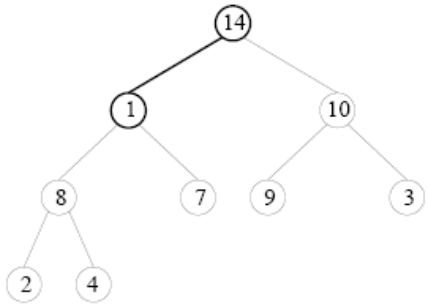
L'élément le plus prioritaire se trouve toujours à la racine, donc le retrait consiste à lire la racine puis la supprimer. Pour ce faire on doit :

1. Remplacer la valeur de la racine par la valeur de l'élément le plus à droite dans le dernier niveau.
2. Supprimer de l'arbre cet élément (le plus à droite dans le dernier niveau), on obtient un arbre binaire mais pas nécessairement un tas.
3. On compare la valeur de la racine avec les valeurs de ses deux fils et on la permute avec la plus grande. On recommence le processus jusqu'il n'y ait plus d'éléments à permuter.

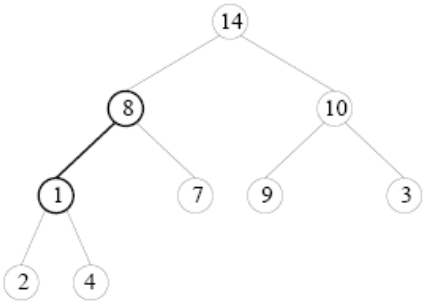
Exemple :



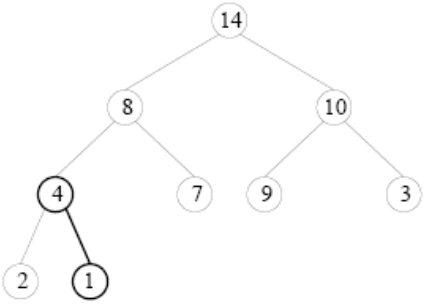
- 1 -



- 2 -



- 3 -



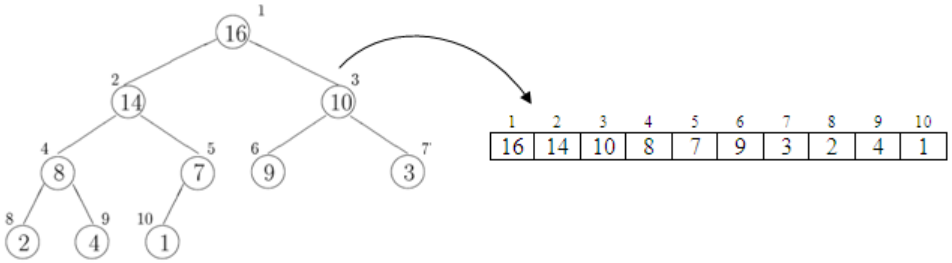
- 4 -

La suppression est aussi en $O(h = Log_2(n))$.

4.3.4 Implémentation des tas

Les tas peuvent être implémentés dynamiquement exactement comme les ARB, et sont utilisés par le même modèle.

Une représentation statique très efficace utilisant des tableaux est très utilisée en pratique, elle consiste à ranger les éléments du tas dans un tableau selon un parcours en largeur :



On remarque sur le tableau obtenu que le fils gauche d'un élément d'indice i se trouve toujours s'il existe à la position $2i$, et son fils droit se trouve à la position $(2i + 1)$ et son père se trouve à la position $i/2$. Les opérations d'ajout et de retrait sur le tas statique se font de la même façon que dans le cas du tas dynamique. Avec ce principe les opérations d'ajout et de retrait se font d'une manière très simple et extrêmement efficace.
Les tas sont utilisés même pour le tri des tableaux : on ajoute les éléments d'un tableau à un tas, puis on les retire dans l'ordre décroissant.

4.4 Exercices

1. **Arbres m-aires**

   Ecrire sur les arbres m-aires les fonctions récursives qui retournent :
   - Le père d'un nœud donné,
   - Le maximum dans un arbre donné,
   - Le minimum dans un arbre donné.

2. **Arbres de recherche binaires**

   (a) Compréhension
   - Construire un arbre de recherche binaire à partir des clés ordonnées suivantes :

     25 60 35 10 5 20 65 45 70 40 50 55 30 15
   - Ajouter à l'arbre obtenu et dans l'ordre, les éléments suivants :

     22 62 64 4 8
   - Supprimer de l'arbre obtenu et dans l'ordre les éléments suivants :

     15 70 50 35 60 25

   (b) Ecrire sur les arbres de recherche binaires les fonctions récursives qui retournent :
   - Vrai si un nœud donné est une feuille et Faux sinon.
   - Le nombre de feuilles de l'arbre.
   - La taille de l'arbre.

- Le père d'un nœud donné.
- Le successeur d'un nœud donné (l'élément immédiatement supérieur).
- Le prédécesseur d'un nœud donné (l'élément immédiatement inférieur).
- Le maximum dans un arbre donné.
- Le minimum dans un arbre donné.
- La hauteur d'un arbre.

(c) Ecrire les algorithmes d'insertion et de suppression dans un ARB.

(d) Ecrire la fonction qui crée un arbre binaire de recherche équilibré à partir d'un tableau trié.

3. Ecrire un algorithme itératif de parcours en profondeur dans un arbre de recherche binaire avec utilisation de piles dans le cas d'un parcours :
   - Préordre (Père FG FD)
   - Inordre (FG Père FD)
   - PostOrdre (FG FD Père)

4. **Algorithme de Huffman**

   Un village d'Anglesey (petite île au sud-ouest de l'Angleterre) porte l'un des plus long nom au monde : "Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch" Les habitants du village sont fières de ce merveilleux nom mais réalisent, avec tristesse, que (certainement par manque de bois !) toutes les affiches annonçant leur village ne contiennent qu'une toute petit partie du nom originale : "Llanfairpwll". Pour être certain de préserver leur héritage culturel, ils décident d'encoder le nom original de leur village en une séquence de bits. Bien entendu, ils veulent que cette séquence soit la plus courte possible.

   (a) Faire un tableau de la fréquence d'apparitions de chacune des lettres apparaissant dans le nom du village. ("L" et "l" considérées comme étant la même lettre, (fréquence=nombre d'occurrences de la lettre divisé par longueur du mot).

   (b) Utiliser ce tableau pour construire, avec l'algorithme de Huffman, l'arbre binaire donnant le meilleur codage. (Lorsque plusieurs arbres on le même poids - ici poids = fréquence- choisir les arbres selon l'ordre lexicographique)

   (c) Donner un tableau associant à chacune des lettres son code binaire.

   (d) Quel est le nombre de bits minimal nécessaire pour encoder le nom de ce charmant village.

Pour les curieux : Ce nom de village n'est pas une invention ! ! Le nom date du 19ème siècle et se traduit, plus ou moins, comme : "St Mary's church in the hollow of the white hazel near a rapid whirlpool and the church of St Tysilio of the red cave" (voir http ://www.bbc.co.uk/h2g2/guide/A403642). Le nom complet du village n'est maintenant utilisé que pour impressionner les touristes. Le nom "Llanfairpwll" est utilisé par les habitants.

5. **Tas**

   (a) Compréhension

   - Construire un tas à partir des clés ordonnées suivantes :

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

   - Ajouter au tas et dans l'ordre les éléments suivants :

     22 62 64 4 8

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

     15 70 50 35 60 25

   (b) Ecrire les algorithmes d'insertion et de suppression dans un tas statique.

   (c) Ecrire les algorithmes d'insertion et de suppression dans un tas dynamique.

   (d) Ecrire un algorithme qui permet de trier un tableau en utilisant un tas. Calculer la complexité de cet algorithme.

Chapitre 5

Structures en Tables

5.1 Introduction

On utilise souvent en informatique une structure appelée "Table" ou dictionnaire pour ranger des informations en mémoire. Une table est un ensemble de couples <*clé, information*> où chaque clé n'apparaît qu'une seule fois dans la table.

Exemples :

1. Annuaire téléphonique

Clé (Prénom)	Information(Tél + Adresse)	
SAMIR	033701520	Sidi Okba
FARID	072171304	Eloued
.	.	.
.	.	.

2. Dictionnaire

Clé (Mot)	Information(Signification)
Arbre	Graphe connexe sans cycle
Table	Ensemble de couples <clé, info>
.	.
.	.

3. Codage

Clé (Mot)	Information(Code)
a	001
b	010
.	.
.	.

Le problème posé est : comment organiser la table pour que les accès (recherche, insertion, suppression) soient les plus rapides possibles ?

5.2 Accès séquentiel

Il consiste à ranger les clés dans la table dans l'ordre de leur arrivée, les une à la suite des autres, c'est-à-dire que l'ajout se fait toujours à la fin de la table. La recherche d'une clé consiste à tester les éléments l'un après l'autre jusqu'à la trouver c'est-à-dire passer par tous les éléments placés avant. La recherche est alors en moyenne de $n/2$ ($O(n)$).

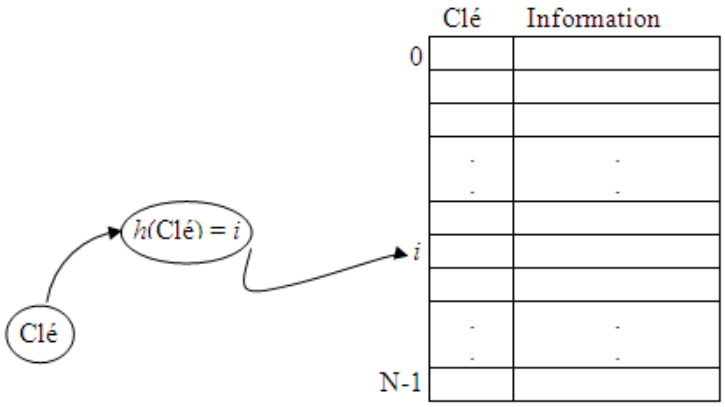
5.3 Table triée

La table est triée selon les valeurs des clé : la recherche est en $log_2(n)$ c-à-d dichotomique.

5.4 Hachage (HashCoding)

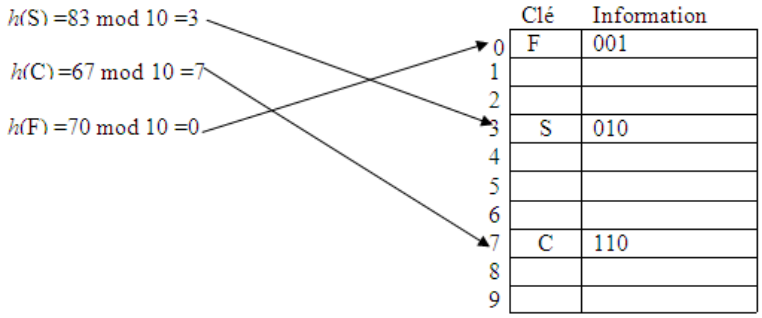
5.4.1 Principe

C'est une technique très utilisée en informatique, elle se base sur une fonction h appelée fonction de hachage ou de hashcoding, qui appliquée à la clé fournit l'indice correspondant dans la table.



Exemple : Codage

- Clé = Caractère (A, B, $\cdots$)
- $N = 10$
- h : Code ASCII mod N



5.4.2 Fonctions de Hachage

On trouve plusieurs types de fonctions utilisées en Hashcoding :

1. h(clé) = CodeASCII (1^{er} car) + Code ASCII($2^{ème}$ car) mod N

   Si la clé est une chaine de caractères (nom)

2. h(clé) = clé mod N

   Si clé est une valeur numérique.

3. Méthode du milieu du carré

   - Clé = 453
   - $(Cl)^2 = (453)^2 = 20\mathbf{52}09$
   - h(453) = 52

   Si n $>$ 100, on prend 3 chiffres.

4. Hachage de Fibonacci

   C'est une fonction de hachage fréquemment utilisée :

   $h(clé) = |N \times (clé \times r - |clé \times r|)|$

   où || donne la partie entière, avec $r = \frac{\sqrt{5}-1}{2}$

La fonction de hachage doit donner des valeurs entières dans l'intervalle des indices de la tables : $0 \leq h(clé) \leq N$.

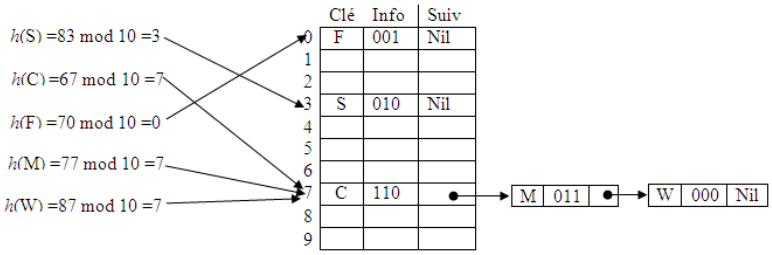
En plus, cette fonction doit être la plus distribuée possible sur cet intervalle, pour que les informations ne se concentrent pas dans une partie de la table.

5.4.3 Problème de collisions

Un problème sérieux se pose avec les fonctions de hachage si deux clés différentes donnent lieu à une même adresse lorsqu'on leur applique la fonction de hachage, c-à-d $h(clé_1) = h(clé_2)$.
Une telle situation est appelée « collision » et plusieurs solutions existent pour sa résolution.

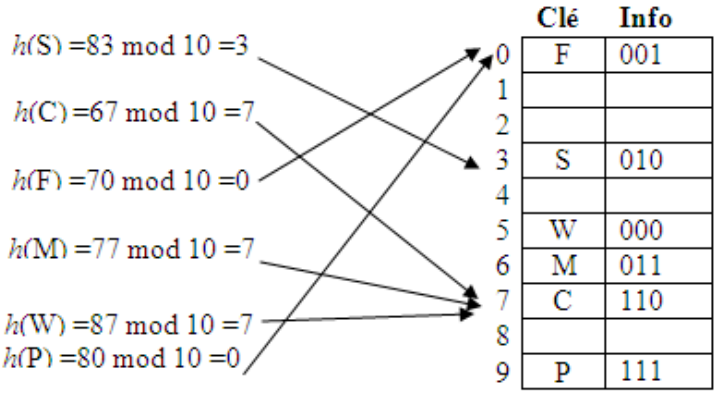
5.4.3.1 Les listes linéaires chaînées

Elle consiste à placer toutes les clés qui donnent le même indice qu'une clé existante dans une liste linéaire chainée, appelée liste de débordement :



5.4.3.2 Essai linéaire (adressage ouvert)

Cette méthode est généralement utilisée si le nombre d'informations est peu par rapport à la taille de la table. On range la clé qui a causé la collision (k par exemple) dans la première position vide dans la séquence cyclique :
$h(k-1), h(k-2), \cdots, 0, N-1, N-2, \ldots, h(k+1)$
Exemple :



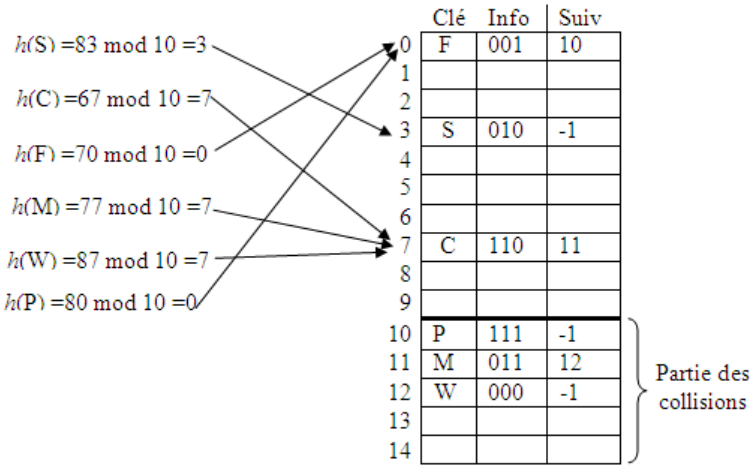
On doit garder, dans ce cas, une case toujours vide pour indiquer la fin de la recherche, c-à-d une valeur interdite dans les clés.

5.4.3.3 Chaînage interne séparé

On ajoute à la table une partie réservée aux collisions de taille M. La taille de la table sera donc $N + M$.

On insère l'élément en collision dans la première place vide dans la partie des collisions et on le relie par un chaînage.

Exemple :



5.5 Exercices

On veut enregistrer un annuaire téléphonique sur micro, en utilisant la méthode de Hash-Coding. Chaque entrée de la table représente les informations concernant une personne (Nom, prénom, Numéro de téléphone). La clé considérée est le nom, et la fonction de hachage est donnée par :

$$H(cle) = (CodeASCII(1^{er}\ car) + CodeASCII(2^{ème}\ car))\ Mod\ N \qquad (5.1)$$

Où N est la taille de la table est égale à 100.

Remarque : les caractères peuvent être en majuscule ou en minuscule et il ne doit pas y avoir de distinction.

Questions :

- Décrire les structures de données nécessaires à la représentation des données en cas d'utilisation de la méthode des LLCs pour résoudre les collisions.
- Ecrire les procédures de recherche, d'insertion et de suppression.
- Répéter les mêmes questions en utilisant la méthode d'essai linéaire.

– Répéter les mêmes questions en utilisant la méthode de chaînage interne séparé.

Chapitre 6

Les graphes

6.1 Introduction

La notion de graphe est une structure qui permet de représenter plusieurs situations réelles, en but de leur apporter des solutions mathématiques et informatique, tel que :

- Les réseaux de transport (routiers, ferrés, aériens $\cdots$),
- Les réseaux téléphoniques, électriques, de gaz,$\cdots$ etc,
- Les réseaux d'ordinateurs,
- Ordonnancement des tâches,
- Circuits électroniques,
- $\ldots$

Les arbres et les listes linéaires chaînées ne sont que des cas particuliers des graphes.

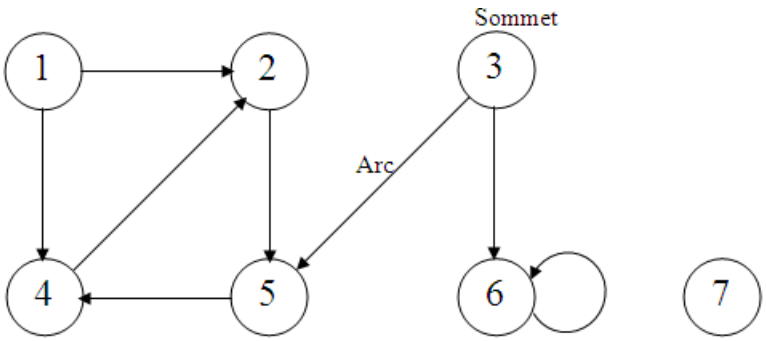
6.2 Définitions

6.2.1 Graphe

Un graphe est défini par un couple (S, A) où S est un ensemble de sommets (nœuds ou points) et A est un sous ensemble du produit cartésien $(S \times S)$ représentant les relations existant entre les sommets.

Exemple : $S = 1, 2, 3, 4, 5, 6, 7$

$A = (1, 2), (1, 4), (2, 5)(3, 5), (3, 6), (4, 2), (5, 4), (6, 6)$

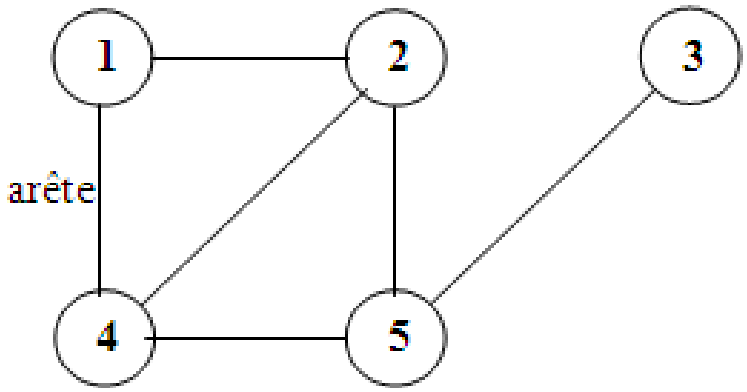


6.2.2 Graphe orienté

C'est un graphe où les relations entre les sommets sont définies dans un seul sens (exemple précédent). Dans ce cas les relations sont appelées "arcs".

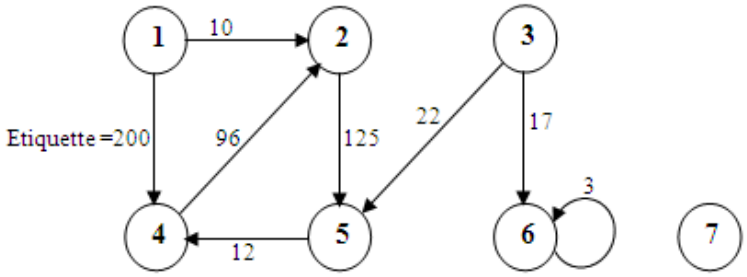
6.2.3 Graphe non orienté

C'est un graphe où les relations sont définies dans les deux sens. Dans ce cas, les relations sont appelées "arêtes".



6.2.4 Graphe étiqueté ou pondéré

C'est un graphe orienté ou non où à chaque arc ou arête correspond une valeur ou une étiquette représentant son coût (ou distance).



6.2.5 Origine et extrémité

Si $a = (x, y)$ est un arc de x vers y alors :

– x est l'origine de a et y est son extrémité

– x est un prédécesseur de y et y est un successeur de x

Si l'origine et l'extrémité d'un arc se coïncident on l'appelle une boucle (6,6).

6.2.6 Chemin

Un chemin est un ensemble d'arcs $a_1, a_2, \cdots, a_p$ où Origine(a_{i+1}) = Extrémité(a_i), $1 \leq i \leq p$

On dit que le chemin est de longueur $p - 1$

Exemple : $\{(1, 2), (2, 5), (5, 4)\}$

6.2.7 Circuit

Un circuit est un chemin $a_1, a_2, \cdots, a_p$ où Origine(a_1) = Extrémité(a_p)

Exemple : $\{(2, 5), (5, 4), (4, 2)\}$

6.2.8 Chaine

Une chaine est un chemin non orienté.

Exemple : $\{(1, 4), (5, 4), (3, 5)\}$

6.2.9 Cycle

Un cycle est une chaîne fermée.

Exemple : $\{(1, 2), (2, 5), (5, 4), (1, 4)\}$

6.2.10 Graphe connexe

Un graphe connexe est un graphe où pour tout couple de sommets (x, y), il existe une chaîne d'arcs les joignant.

Exemple : Pour le couple $(1, 6)$, il existe une chaîne d'arcs, et il n'en existe pas pour (1,7).

6.2.11 graphe fortement connexe

Un graphe fortement connexe est un graphe où pour tout couple de sommets (x, y), il existe un chemin d'arcs les joignant.

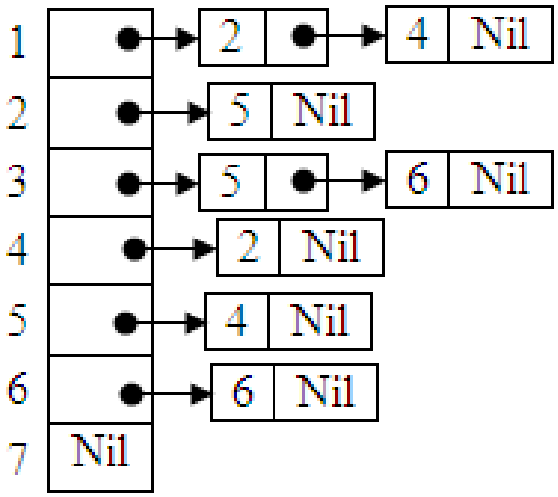
Exemple : Pour $(3, 2)$ il existe un chemin $\{(3, 5), (5, 4), (4, 2)\}$ mais il n'en existe pas pour $(2, 3)$.

6.3 Représentation des graphes

Les graphes peuvent être représentés en deux manières : en listes d'adjacence (dynamique) ou en matrice d'adjacence (statique)

6.3.1 Listes d'adjacence

Dans cette représentation, les successeurs d'un nœud sont rangés dans une liste linéaire chainée. Le graphe est représenté par un tableau T où $T[i]$ contient la tête de la liste des successeurs du sommet numéro i. Le graphe (S, A) présenté au début de cette section peut être représenté comme suit :



Les listes d'adjacence sont triées par numéro de sommet, mais les successeurs peuvent apparaître dans n'importe quel ordre.

```
Type  TMaillon = Structure
        Successeur : entier ;
        Suivant : Pointeur(TMaillon) ;
Fin ;
Var  Graphe : Tableau[1..n] de Pointeur(TMaillon) ;
```

6.3.2 Matrice d'adjacence

Dans cette représentation le graphe est stocké dans un tableau à deux dimensions de valeurs booléennes ou binaires. Chaque case (x, y) du tableau est égale à *vrai* (1) s'il existe un arc de x vers y, et *faux* (0) sinon. Dans la matrice suivante, on représente le graphe précédent (1 pour *vrai* et 0 pour *faux*) :

	1	2	3	4	5	6	7
1	0	1	0	1	0	0	0
2	0	0	0	0	1	0	0
3	0	0	0	0	1	1	0
4	0	1	0	0	0	0	0
5	0	0	0	1	0	0	0
6	0	0	0	0	0	1	0
7	0	0	0	0	0	0	0

Il est clair que la matrice d'adjacence d'un graphe non orienté est symétrique puisque chaque arête existe dans les deux sens.

Var Graphe : **Tableau[1..n,1..n] de booléen ;**

Pour un graphe étiqueté les valeurs de la matrice peuvent être les étiquettes elles-mêmes, avec une valeur particulière pour les arcs inexistant. Par exemple le graphe étiqueté de l'exemple peut être représenté comme suit :

	1	2	3	4	5	6	7
1	-1	10	-1	200	-1	-1	-1
2	-1	-1	-1	-1	125	-1	-1
3	-1	-1	-1	-1	22	17	-1
4	-1	96	-1	-1	-1	-1	-1
5	-1	-1	-1	12	-1	-1	-1
6	-1	-1	-1	-1	-1	3	-1
7	-1	-1	-1	-1	-1	-1	-1

La représentation matricielle permet de tirer des conclusions intéressantes sur les graphes en utilisant les calculs matriciels.

6.4 Parcours de graphes

De même que pour les arbres, il est important de pouvoir parcourir un graphe selon certaines règles, cependant, le parcours des graphe est un peut différent de celui des arbres. Dans un arbre, si on commence à partir de la racine on peut atteindre tous les nœuds, malheureusement, ce n'est pas le cas pour un graphe où on est obligé de reprendre le parcours tant qu'il y a des sommets non visités. En plus un graphe peut contenir des cycles, ce qui conduit à des boucles infinies de parcours.
Il existe deux types de parcours de graphes : le parcours en profondeur d'abord (Depth First Search) et le parcours en largeur d'abord (Breadth First Search).

6.4.1 En profondeur d'abord (DFS)

Le principe du DFS est de visiter tous les sommets en allant le plus profondément possible dans le graphe.

```
Var  Graphe : Tableau[1..n,1..n] de booleen ;
     Visité : Tableau[1..nn] de booleen ;
Procédure DFS( Sommet : entier ;);
var  i : entier ;
Début
    Visité[Sommet] ← Vrai ;
    Afficher(sommet) ;
    Pour i de 1 à n faire
        Si (Graphe[sommet,i] et non Visité[i]) Alors
            DFS(i)
        Fin Si;
    Fin Pour;
Fin;

Appel :
Pour s de 1 à n faire
    Si (Non Visité[s]) Alors
        DFS(s)
    Fin Si;
Fin Pour;

Trace : 1 2 5 4 3 6 7
```

La procédure DFS peut être utilisée pour divers objectifs :

- Pour vérifier s'il existe une chemin d'un sommet s_1 vers un autre s_2 en initialisant le tableaux Visité à faux et en appelant DFS(s_1) pour ce sommet. Si Visité[s_2] est à vrai à la fin de l'appel de DFS, alors un chemin existe entre s_1 et s_2.
- Pour vérifier si un circuit contentent deux sommets s_1 et s_2 existe, en appelant DFS(s_1) pour un tableaux Visité1 et DFS(s_2) pour un tableaux Visité2, si Visité1[s_2] et Visité2[s_1] sont à Vrai après l'appel alors un tel circuit existe.
- De la même manière pour vérifier si un graphe est cyclique (contient des circuits) ou non.
- Pour trouver les chemins minimums d'un sommet s vers tous les autres.

6.4.2 En largeur d'abord (BFS)

Dans ce parcours, un sommet s est fixé comme origine et l'on visite tous les sommet situés à une distance k de s avant de passer à ceux situés à $k+1$. On utilise pour cela une file d'attente.

```
Var  Graphe : Tableau[1..n,1..n] de booleen ;
     Visité : Tableau[1..n] de booleen ;
     F : File d'attente ;
Procédure BFS( Sommet : entier ;);
Var  i,s : entier ;
Début
    Enfiler(F,Sommet) ;
    Tant que (non File_Vide(F)) faire
        Defiler(F,s) ;
        Afficher(s) ;
        Pour i de 1 à n faire
            Si (Graphe[s,i] et non Visité[i]) Alors
                Enfiler(F,i) ;
            Fin Si;
        Fin Pour;
    Fin TQ;

Fin;

Appel :

Pour s de 1 à n faire
    Si (non Visité[s]) Alors
        BFS(s)
    Fin Si;
Fin Pour;

Trace : 1 2 4 5 3 6 7
```

6.5 Plus court chemin (algorithme de Dijkstra)

Soit un graphe $G(S,U)$ orienté sans boucles avec n sommets,

$(x_i, x_j) \in U \Rightarrow l_{ij}$ = longueur de l'arc (x_i, x_j)

La longueur d'un chemin $\mu = \sum_{(x_i,x_j)\in\mu} l_{ij}$

On peut trouver plusieurs chemins reliant un sommet à un autre. Le problème du chemin minimum ou de plus court chemin consiste à trouver le chemin de moindre coût. On trouve plusieurs algorithmes pour la résolution de ce problème : Dijkstra, Ford, Belman-Kalaba,$\cdots$ etc.

6.5.1 Algorithme de Dijkstra

L'algorithme de Dijkstra résout le problème de la recherche d'un plus court chemin à origine unique pour un graphe orienté pondéré $G = (S,U)$ dans le cas où tous les arcs ont un poids positif ou nul.

L'algorithme de Dijkstra maintient à jour un ensemble D des sommets de S dont le plus court chemin à partir de l'origine s est connu et calculé. A chaque itération, l'algorithme choisit parmi les sommets de (S - D) c'est-à-dire parmi les sommets dont le plus court chemin à partir de l'origine n'est pas connu, le sommet x dont l'estimation de plus court chemin est minimale. Une fois un sommet u choisi, l'algorithme met à jour, si besoin est, les estimations des plus courts chemins de ses successeurs (les sommets qui peuvent être atteint directement à partir de u).

1- $D = x_i, \lambda_i = 0$ // x_i sommet de départ

$\lambda_j = l_{ij}$, $j \neq i$ (si un arc (x_i, x_j) n'existe pas $l_{ij} = +\infty$)

//λ_j coût du chemin minimum entre x_i et x_j

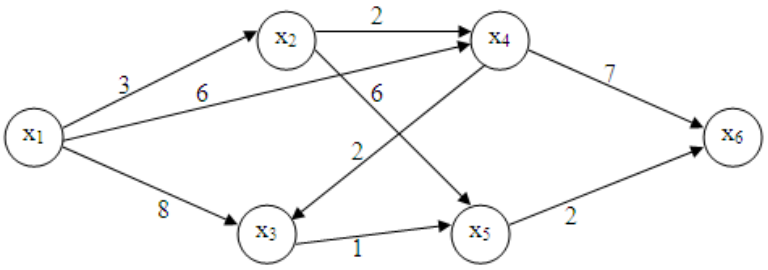
2- $\lambda_k = min\ \lambda_j$ / $x_j \notin D$

3- $D = D \cup \{x_k\}$

$\lambda_j = min\{\lambda_j, \lambda_k + l_{kj}\}$

4- Aller à 2 si $D \neq S$

Exemple :



D	λ_1	λ_2	λ_3	λ_4	λ_5	λ_6
x_1	0	3	8	6	∞	∞
x_1, x_2	0	3	8	5	9	∞
x_1, x_2, x_4	0	3	7	5	9	12
x_1, x_2, x_4, x_3	0	3	7	5	8	12
x_1, x_2, x_4, x_3, x_5	0	3	7	5	8	10
$x_1, x_2, x_4, x_3, x_5, x_6$	0	3	7	5	8	10

6.6 Exercices

1. Ecrire une procédure permettant de créer un graphe en listes d'adjacence à partir d'un graphe représenté sous forme d'une matrice d'adjacence.
2. Ecrire une procédure permettant de créer un graphe en une matrice d'adjacence à partir d'un graphe représenté sous forme de listes d'adjacence.

En utilisant le parcours DFS sur une matrice d'adjacence :

1. Ecrire une fonction permettant de vérifier s'il existe un chemin entre deux sommets donnés.
2. Ecrire une fonction permettant de vérifier s'il existe une chaîne entre deux sommets donnés.
3. Ecrire une fonction permettant de vérifier s'il existe un circuit dans le graphe.
4. Ecrire une fonction permettant de vérifier s'il existe un cycle dans le graphe.
5. Ecrire une fonction permettant de vérifier si un graphe est cyclique ou non.
6. Ecrire une fonction permettant de détecter les composantes connexes d'un graphe sachant qu'une composante connexe est un ensemble de sommets dont chaque paire est reliée par au moins une chaîne.

7. Ecrire une fonction permettant de détecter les composantes fortement connexes d'un graphe sachant qu'une composante fortement connexe est un ensemble de sommets dont chaque paire est reliée par un chemin.
8. Reprendre les mêmes exercices avec un graphe représenté par des listes d'adjacence.
9. Ecrire une procédure permettant de données les chemins minimums à partir d'un sommet donné vers tous les autres en utilisant l'algorithme de Dijkstra.

Chapitre 7

Preuve d'algorithmes

7.1 Introduction

La preuve d'un algorithme consiste à montrer que cet algorithme transforme bien ses entrées en les sorties attendues, autrement dit, tout état initial de l'algorithme conduit en sortie à un état final qui vérifie une certaine propriété (ex : factorielle = $n!$).
La méthode usuelle pour vérifier la correction du'un algorithme P, devant calculer la fonction f, est la méthode des tests : on choisit un échantillon fini de données $d_1, ..., d_n$ on fait exécuter l'algorithme P pour chacune d'entre elles et on vérifie que :

$$P(d_1) = f(d_1), \cdots, P(d_n) = f(d_n)$$

C'est-à-dire démontrer un théorème équivalent à :

$$\forall \quad d \in D, \quad P(d) = f(d)$$

L'insuffisance de cette méthode est évidente dès que l'échantillon testé ne recouvre pas l'ensemble D des données.

7.2 Méthode de preuve d'algorithme

La méthode des preuves d'algorithme est bien plus satisfaisante : elle consiste à prouver mathématiquement que l'algorithme P est correct en démontrant qu'il :

1. termine (terminaison), et
2. calcule bien sa fonction f (correction partielle)

On distingue donc trois notions : terminaison, correction partielle et correction totale :

Correction totale = Correction partielle + Terminaison

7.2.1 Terminaison

On dit qu'un algorithme P termine, si et seulement si, tout état initial E donne une exécution terminante de P.

7.2.2 Correction partielle

Soit l'algorithme P dont la correction est exprimée par une condition C, On dit que P est partiellement correct si et seulement si, pour tout état initial E qui donne une exécution terminante $F = P(E)$ on a $F(C) = Vrai$.

7.2.3 Correction totale

On dit que P est totalement correct si et seulement si tout état initial E donne une exécution terminante $F = P(E)$ qui vérifie $F(C) = Vrai$.

7.2.4 Exemples :

1. L'algorithme suivant ne termine pas :

Tant que (V) **faire**
  |
Fin TQ;

2. Etant donné la propriété $C = (r = max(x, y))$. L'algorithme suivant n'est pas partiellement correct parce qu'il ne vérifie pas C pour des cas d'exécutions terminantes $(x > y, r =?)$.

Si $(x < y)$ **Alors**
  | $r \leftarrow y$
Fin Si;

3. L'algorithme suivant est partiellement correct et termine ; il est donc totalement correct.

Si $(x < y)$ **Alors**
  $r \leftarrow y$
Sinon
  $r \leftarrow x$
Fin Si;

4. L'algorithme suivant est partiellement correct mais ne termine pas (toujours) ; il n'est donc pas totalement correct

Si $(x < y)$ **Alors**
  $r \leftarrow y$
Sinon
  Tant que $(x \neq y)$ **faire**
    $r \leftarrow x$
  Fin TQ;
Fin Si;

7.3 Outils de preuve d'algorithme (Logique de Hoare)

Pour vérifier la correction totale d'un algorithme, on doit vérifier qu'il est partiellement correct et qu'il se termine. Pour cela, on utilise la logique de Hoare qui représente un système déductif proposé par *Tony Hoare*. Elle permette de vérifier si un algorithme vérifie nécessairement une certaine propriété à la fin de toutes ses exécutions.

Dans la logique de Hoare, la preuve d'un algorithme P est représentée par un triplet appelé triplet de Hoare :

$$\{c\} \quad P \quad \{c^{'}\}$$

tel que

$\{c\}$: est une expression booléenne appelée la précondition,

P : l'algorithme (le programme),

$\{c^{'}\}$: est une expression booléenne appelée la postcondition.

Ce triplet est interprété comme suit : "Si les variables de P vérifient initialement la condition c alors, si P se termine, ces variables vérifieront la condition $c^{'}$ après l'exécution de P".

Exemple

Soit l'algorithme suivant :

```
Algorithme Div;
Var  a,b,r,q : entier ;
Début
| Lire(a,b) ;
| r ← a ;
| q ← 0 ;
| Tant que (r ≥ b) faire
|     | r ← r - b ;
|     | q ← q + 1 ;
| Fin TQ;
| Ecrire(q, r) ;
Fin.
```

Le triplet de Hoare utilisé dans la preuve de cet algorithme est le suivant :

$$\{a \geq 0 \quad \& \quad b > 0\} \quad Div \quad \{a = b * q + r \quad \& \quad r < b\}$$

La preuve consiste à démonter que ce triplet soit valide en utilisant un axiome et six règles :

1. Axiome de l'affectation :

$$\{c^{'}\} \quad x \leftarrow v \quad \{c\}$$

Tel que $c^{'}$ est la condition c où l'on remplace toute occurrence de x par v.

Exemple : $\{x = y\} \quad x \leftarrow 2 * x \quad \{x = 2 * y\}$

2. Règle de la séquence :

$$Si \quad [\{c\} \quad P_1 \quad \{c_1\} \quad \wedge \quad \{c_1\} \quad P_2 \quad \{c_2\}] \quad alors \quad [\{c\} \quad P_1; P_2 \quad \{c_2\}]$$

Exemple : $Si\,[\quad \{c\}x \leftarrow 2\{x = 2\} \wedge \{x = 2\}y \leftarrow x\{y = 2\}] \quad alors\,[\quad \{c\}x \leftarrow 2; y \leftarrow x\{y = 2\}]$

3. Règle de la conditionnelle :

$$\begin{array}{l} Si \quad \left[\{c \quad \& \quad b\} \quad P_1 \quad \{c^{'}\} \quad \wedge \quad \{c \quad \& \quad \neg b\} \quad P2 \quad \{c^{'}\}\right] \\ alors \quad \left[\{c\} \quad si \quad b \quad alors \quad P_1 \quad sinon \quad P_2 \quad Fsi \quad \{c^{'}\}\right] \end{array}$$

Exemple :

$Si \quad [\{x < y\} \quad r \leftarrow y \quad \{r = max(x, y)\} \wedge \{x \geq y\} \quad r \leftarrow x \quad \{r = max(x, y)\}] \quad Alors$
$[\{V\} \quad si \quad x < y \quad alors \quad r \leftarrow y \quad sinon \quad r \leftarrow x \quad finsi \quad \{r = max(x, y)\}]$

4. Règle de la répétitive

$$Si \quad [\{c \quad \& \quad b\} \quad P \quad \{c\}] \quad alors \quad [\{c\} \quad Tanque \quad b \quad Faire \quad P \quad FTQ \quad \{c \quad \& \quad \neg b\}]$$

Note : la condition c s'appelle invariant de boucle

Exemple :

$Si \quad \{(r \geq 0) \wedge (x > y)\} \quad r \leftarrow r + 1; x \leftarrow x - y \quad \{r \geq 0\} \quad alors \quad \{r \geq 0\} \quad Tantque \quad (x > y) \quad faire \quad r \leftarrow r+1; \quad x \leftarrow x-y \quad fintq \quad \{(r \geq 0) \wedge (y \geq x)\}$

5. Règle de conséquence

$$Si \quad [c_1 \Rightarrow c_2 \quad \wedge \quad \{c_2\} \quad P \quad \{c_3\} \quad \wedge \quad c_3 \Rightarrow c_4] \quad alors \quad [\{c_1\} \quad P \quad \{c_4\}]$$

6. Règle de la conjonction

$$Si \quad [\{c\} \quad P \quad \{c_1\} \quad \wedge \quad \{c\} \quad P \quad \{c_2\}] \quad alors \quad [\{c\} \quad P \quad \{c_1 \quad \wedge \quad c_2\}]$$

7. Règle de la disjonction

$$Si \quad [\{c_1\} \quad P \quad \{c\} \quad \wedge \quad \{c_2\} \quad P \quad \{c\}] \quad alors \quad [\{c_1 \quad \vee \quad c_2\} \quad P \quad \{c\}]$$

On utilise ces règles pour construire l'algorithme proposé en remontant à partir de la postcondition pour arriver à la précondition. Si l'algorithme a pu être construit, on dit que l'algorithme est partiellement correct.

Pour démonter la terminaison de l'algorithme, il faut démontrer que toutes ses boucles se terminent, pour cela, il faut démonter que l'expression de la condition de chaque boucle

forme une suite convergente. Cette démonstration peut être faite par la logique de Hoare comme suit :

$$\{c\} \quad Boucle \quad \{\neg c\}$$

tel que c est la condition de la boucle.

7.4 Exemple

Soit l'algorithme P suivant :

```
i ← 1 ;
r ← 1 ;
Tant que (i ≤ n) faire
    r ← r*i ;
    i ← i+1 ;
Fin TQ;
```

Ce programme semble calculer factorielle(n). On prend donc la précondition $c = (n \geq 0)$ et la postcondition $c^{'} = (r = n!)$. Montrons que le triplet suivant est valide :

$$\{n \geq 0\} \quad i \leftarrow 1; \quad r \leftarrow 1; \quad Tantque \quad i \leq n \quad faire \quad r \leftarrow r * i; \quad i \leftarrow i + 1 \quad fintq \quad \{r = n!\}$$

La solution est représentée comme suit :

$\{n \geq 0\}$

$\{(n \geq 0) \wedge (1 = 0!) \wedge (1 \leq n + 1)\}$

$i \leftarrow 1;$

$\{(n \geq 0) \wedge (1 = (i-1)!) \wedge (i \leq n + 1)\}$

$r \leftarrow 1;$

$\{(n \geq 0) \wedge (r = (i-1)!) \wedge (i \leq n + 1)\}$

Tant que (i $\leq$ n) **faire**

$\{(n \geq 0) \wedge (r = (i-1)!) \wedge (i \leq n)\}$

$r \leftarrow r * i;$

$\{(n \geq 0) \wedge (r = i!) \wedge (i \leq n)\}$

$i \leftarrow i + 1;$

$\{(n \geq 0) \wedge (r = (i-1)!) \wedge (i \leq n + 1)\}$

Fin TQ;

$\{(n \geq 0) \wedge (r = (i-1)!) \wedge (i \leq n + 1) \wedge (i > n)\}$

$\{r = n!\}$

7.5 Conclusion

La preuve d'un algorithme, même petit, est une opération très longue et fastidieuse, même si l'application des règles est mécanique, l'invention de la précondition et la post-condition n'est pas évidente et difficilement automatisable.

Malgré tout, la preuve formelle de hoare reste le seul moyen de preuve des algorithmes.

Chapitre 8

Sujets d'examens

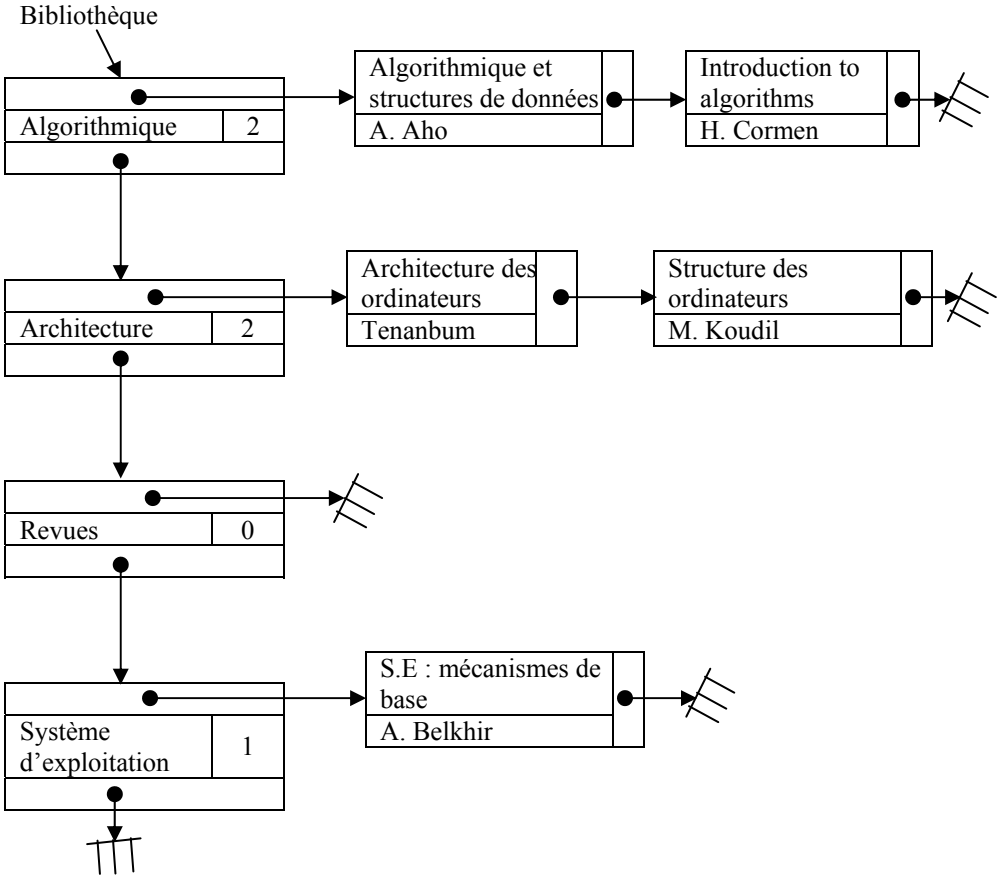
2LMD **12 Fev 2008**

Examen d'algorithmique 1

14h-15h30 **A1**

Exercice 1 (10 pts)

Dans cet exercice, un étudiant en informatique souhaite représenter sa bibliothèque personnelle en utilisant une structure dynamique. La structure proposée est représentée dans la figure suivante :



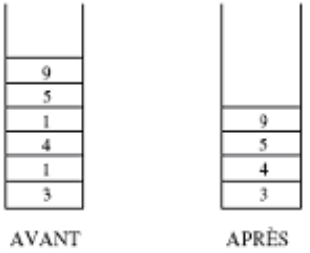
La liste verticale contient les catégories des livres avec le nombre de livres dans chacune, tandis que les listes horizontales contiennent les titres des livres avec leurs auteurs dans chaque catégorie.

1. Donner la déclaration des structures de données nécessaires à l'implémentation de cette bibliothèque ainsi que la procédure d'initialisation de ces structures. 1.5
2. Ecrire la procédure d'ajout d'une nouvelle catégorie (l'ajout se fait toujours à la fin de la liste). 1.5
3. Ecrire la procédure d'insertion d'un nouveau livre (l'ajout se fait au début de la liste) 2
4. Ecrite la procédure qui permette d'afficher les livres d'une catégorie donnée. 1.5
5. Ecrire la fonction qui retourne le nombre total de livre dans la bibliothèque. 1
6. Ecrire la procédure qui permette de supprimer une catégorie avec tous ses livres 2.5

Exercice 2 (5 pts)

1. Les listes (1, 6, 1, 6, 7, 9, 1, 10, 6) et (1, 3, 1, 4, 3, 2, 1, 4, 2) peuvent-elles représenter un tas statique ? Si non, quels échanges faut–il effectuer pour obtenir un tas ?
2. Un tableau trié en ordre décroissant représente-t-il un tas ?
3. On veut réaliser une procédure qui purge une pile de toutes les occurrences d'un entier donné (1 dans l'exemple):

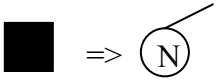
   a. Réaliser cette procédure en utilisant une deuxième pile.
   b. Réaliser cette procédure d'une manière récursive sans utiliser aucune autre structure (ni pile ni file)



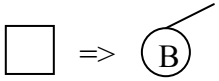
Tournez la page ../..

Exercice 3 (6 Pts)

On convient de représenter une image en noire et blanc à l'aide un arbre m-aire de degré 4 comme suit :
Une image toute noire est représentée par une feuille d'une valeur 'N'



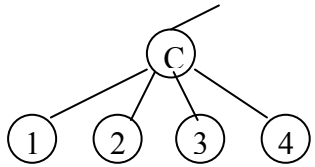
Une image toute blanche est représentée par une feuille d'une valeur 'B'



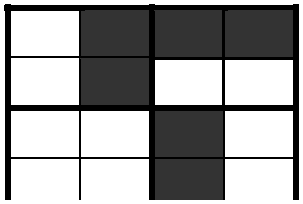
Une image contenant les deux couleurs est représentée par un sous arbre dont la racine est de valeur 'C' et chaque fils représente un quart de l'image.

1	2
3	4

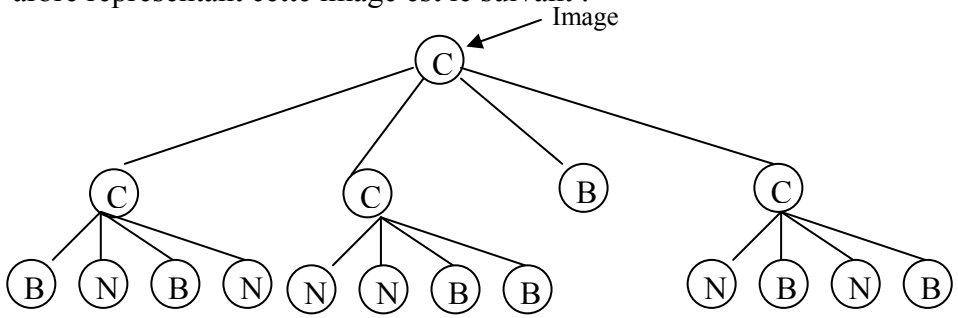
=>



Exemple :
Soit l'image suivante :



L'arbre représentant cette image est le suivant :

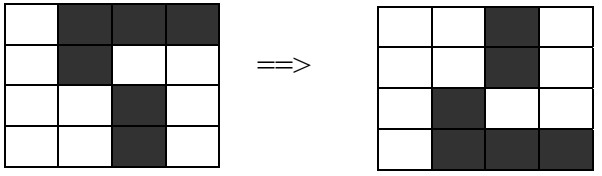


1. Donner les déclarations des structures de données nécessaires à l'implémentation de ce modèle.
2. Soit la procédure suivante :

```
Procedure Traiter (Racine : Pointeur(TNoeud)) ;
Var P : Pointeur(TNoeud) ;
     I : entier ;
Debut
   Si Racine ≠ Nil etValeur(Racine) = 'C' alors
        P ← Fils1 ( Racine ) ;
        Aff_Fils1 ( Racine , Fils3 ( Racine ) ) ;
        Aff_Fils3 ( Racine , Fils4 (Racine ) ) ;
        Aff_Fils4 ( Racine, Fils2 ( Racine ) ) ;
        Aff_Fils2 (Racine,P) ;

        Pour I=1 à 4 faire
          Traiter ( Filsi (Racine) ) ;
        FPour ;
    FSi
Fin ;
```

Donner l'image correspondant à l'arbre précédent après l'appel
Traiter (Image) ;

3. Ecrire la procédure permettant de tourner verticalement une image représentée par un arbre.



Bon courage

A.Djeffal

Corrigé type

Exercice 1

1. (1.5 pt)

```
Type TLivre = Structure
            Titre, Auteur : chaine ;
            Suivant : Pointeur ( TLivre )
Fin ;

Type  TCategorie = Structure
            NomCateg : chaine ;
            NbLivre : entier ;
            TeteListeLivres : Pointeur ( TLivre )
            Suivant : Pointeur ( TCategorie ) ;
Fin ;
Var
    Bibliotheque : Pointeur ( TCategorie ) ;

Procedure Initialisation ;
Debut
    Bibliotheque ← Nil ;
Fin ;
```

2. (1.5 pt)

```
Procedure AjouterCategorie ( NCateg : chaine) ;
Var
     P, PP, Q : Pointeur (TCategorie) ;
Debut

    P ← Bibliotheque ;
    PP ← Nil ;
    TQ P ≠ Nil et NomCteg(P) ≠ NCateg Faire
            PP ← P ;
            P ← Suivant ( P ) ;
    FTQ ;
    Si P ≠ Nil alors Ecrire ( 'Cette categorie existe déjà' ) ;
        Sinon
    Allouer ( Q ) ;
    Aff_Val (Q, NCateg, 0, Nil) ;
    Aff_Adr (Q,Nil) ;
    Si PP = Nil alors Bibliotheque ← P
    Sinon
        Aff_Adr (PP, Q) ;
    FSi ;
Fin;
```

3. (2 pts)

```
Procedure AjouterLivre(NCateg, NTitre,NAuteur:chaine) ;
Var
    P : Pointeur (TCategorie) ;
    Q : Pointeur (TLivre) ;
Debut
    P ← Bibliotheque ;
    TQ P ≠ Nil et NomCateg(P) ≠ NCateg Faire
        P ← Suivant ( P ) ;
    FTQ ;
    Si P = Nil alors Ecrire( 'Cette catégorie n'existe pas')
    Sinon
        Allouer (Q) ;
        Aff_Adr(Q, TeteListeLivre(P)) ;
        Aff _Val(Q, NTitre, NAuteur) ;
        Aff_Val(P, NomCateg (P), NbLivre(P) + 1, Q) ;
    FSi
Fin ;
```

4. (1.5 pt)

```
Procedure Afficher  ( NCateg : chaine) ;
Var
    P : Pointeur (TCategorie) ;
    Q : Pointeur (TLivre) ;
Debut
    P ← Bibliotheque ;
    TQ P ≠ Nil et NomCateg(P) ≠ NCateg Faire
        P ← Suivant ( P ) ;
    FTQ ;
    Si P = Nil alors Ecrire( 'Cette catégorie n'existe pas')
    Sinon
        Q ← TeteListeLibre(P) ;
        TQ Q ≠ Nil Faire
                Ecrite (Titre(Q), Auteur(Q)) ;
                Q ← Suivant (Q) ;
        FTQ ;
    FSi
Fin ;
```

5. (1 pt)

```
Fonction NbreLivre :Entier ;
Var
    P : Pointeur (TCategorie) ;
    Nb : entier ;
Debut
    Nb ← 0 ;
    P ← Bibliotheque ;
    TQ P ≠ Nil Faire
        Nb ← Nb + NbLivres(P) ;
        P ← Suivant (P) ;
    Fin ;
    NbreLivre ← Nb ;
Fin ;
```

6. (2.5 pts)

```
Procedure SupprimerCategorie ( NCateg : chaine) ;
Var

Debut
    P ← Bibliotheque ;
    PP ← Nil ;
    TQ P ≠ Nil et NomCateg(P) ≠ NCateg Faire
        Pp ← Pil.
        P ← Suivant ( P ) ;
    FTQ ;
    Si P = Nil alors Ecrire( 'Cette catégorie n'existe pas')
    Sinon
        Q ← TeteListeLivre(P) ;
        TQ Q ≠ Nil Faire
            Q1 ← Q ;
            Q ← Suivant(Q) ;
             Liberer (Q1) ;
        FTQ ;
        Aff_Adr(PP, Suivant(P)) ;
        Liberer (P) ;
    FSi
Fin ;
```

<u>Exercice 2</u>

1. (1 pt)

Le tableau (1,6,1,6,7,9,1,10,6) n'est pas un tas ; pour obtenir un tas, il faut le mettre dans l'ordre suivant : (10,7,9,6,6,6,1,1,1)

Le tableau (1,3,1,4,3,2,1,4,2) n'est pas un tas ; pour obtenir un tas, il faut le mettre dans l'ordre suivant : (4, 4,2,3,3,1,1,2)

2. (0.5 pt)

Oui un tableau trié dans l'ordre décroissant représente un tas.

3.a. (1.5 pt)

```
Procedure  Purger ( P : Pile, n : entier) ;
Var
    P1 : Pile ;
    x : entier ;
Debut
    InitPile(P1) ;
    TQ non Pile_Vide (P) Faire
       Depiler ( P, x) ;
      Si x ≠ n alors Empiler (P1 , x ) ;
    FTQ ;
    TQ non Pile_Vide(P1) Faire
       Depiler ( P1 , x ) ;
       Empiler ( P, x ) ;
    FTQ ;
Fin ;
```

b. (2 pts)

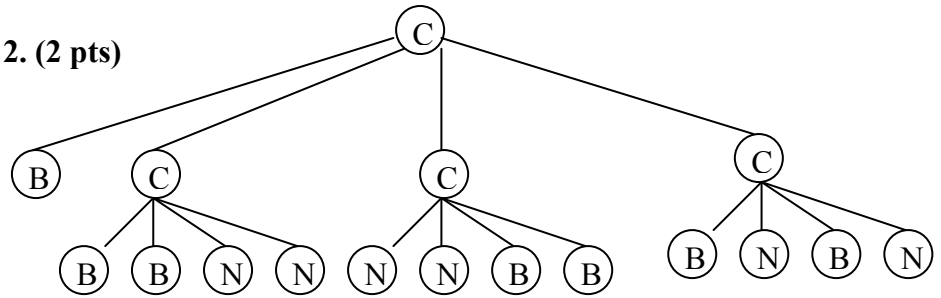
```
Procedure Purger( P : Pile, n : entier ) ;
Var
    x : entier ;
Debut
    Si non Pile_Vide(P) alors
            Depiler ( P, x ) ;
            Purger (P, n) ;
            Si x ≠ n alors Empiler ( P, x) ;
    FSi
Fin ;
```

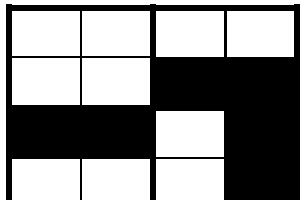
Exercice 3

1. (1 pt)

```
    Type TNoeud = structure
            Couleur : caractère ;
            Fils : Tableau [1..4] de Pointeur (TNoeud) ;
     Fin ;
    Var  Image : Pointeur (TNoeud) ;
```

2. (2 pts)





L'image résultante est une rotation de 90° au sens des aiguilles d'une montre.

3. (3 pts)

```
Procedure Traiter (Racine : Pointeur(TNoeud)) ;
    Var P : Pointeur(TNoeud) ;
        I : entier ;
    Debut
       Si Racine ≠ Nil etValeur(Racine) = 'C' alors
            P ← Fils1 ( Racine ) ;
            Aff_Fils1 ( Racine , Fils3 ( Racine ) ) ;
            Aff_Fils3 ( Racine ,P) ;
            P ← Fils2 ( Racine ) ;
            Aff_Fils2 ( Racine, Fils4 ( Racine ) ) ;
            Aff_Fils4 (Racine,P) ;

            Pour I=1 à 4 faire
              Traiter ( Filsi (Racine) ) ;
            FPour ;
       FSi
    Fin ;
```

2LMD **10 Sep 2008**

Examen de Rattrapage ALGO1

8h-9h30 **A2 + S10**

Exercice 1 (8 Pts : 1+1+1+2+3)

Soit une liste linéaire chaînée contenant des nombres entiers et dont la tête est L :

1. Ecrire la fonction **CAR(L)** qui retourne la valeur du premier élément de la liste.
2. Ecrire la fonction **CDR(L)** qui retourne la liste sans le premier élément.
3. Ecrire la fonction **CONS(x, L)** qui retourne une liste dont le premier élément est x et le reste est la liste L.
4. Ecrire la fonction **Triée(L)** qui retourne vrai si la liste L est triée dans l'ordre croissant et faux sinon.
5. Ecrire la fonction **Fusion(L1, L2)** qui prend deux listes triées dans l'ordre croissant L1 et L2 et retourne une liste triée, dans le même ordre, contenant les deux listes et cela en utilisant les fonctions précédentes.

Exercice 2 (12 Pts : 1 + 1.5 + 1.5 + 1 + 2 + 2 + 3)

1. Dessiner l'arbre binaire de recherche obtenu en insérant les éléments de la liste suivante dans leur ordre d'arrivée: 30, 40, 23, 58, 48, 26, 11, 13, 20
2. Donner le résultat des parcours préfixe, infixe et postfixe de l'arbre précédent
3. Donner l'arbre précédent après la suppression des clés suivantes : 48 puis 23.
4. Donner l'arbre précédent (avant la suppression) après l'ajout des clés suivantes : 60 puis 17.
5. Ecrire la fonction **NB_Sup (R :Nœud, x :clé)** qui retourne le nombre de clés supérieures à x dans l'arbre binaire de recherche de racine R.
6. Ecrire la fonction **Val_Sup (R :Nœud, x :clé)** qui retourne la valeur de la plus petite clé supérieure à x dans l'arbre binaire de recherche de racine R.
7. Ecrire la fonction **Fusion(R1,R2 : Nœud)** qui prend deux arbres binaires de recherches de racines R1, R2 et retourne la racine d'un arbre binaire de recherche contenant les deux. Utilisez les deux procédures Suppr(R, clé) et Inser(R, clé) qui permettent respectivement la suppression et l'insertion de clé dans l'arbre de racine R.

Bon courage

A.Djeffal

Corrigé type

Exercice 1

1.

```
Fonction  CAR(L :Pointeur(TMaillon)) :entier ;
Debut
    Si L<>Nil Alors CAR ← Valeur(L) ;
Fin ;
```

2.

```
Fonction  CDR(L :Pointeur(TMaillon)) : Pointeur(TMaillon) ;
Debut
    Si L<>Nil Alors
          CDR ← Suivant(L) ;
           Libérer(L) ;
     Sinon
       CDR← Nil ;
      FSi ;
Fin ;
```

3.

```
Fonction  CONS(x :entier ; L : Pointeur(TMaillon)) : Pointeur(TMaillon);
  Var P : Pointeur(TMaillon) ;
Debut
    Allouer(P) ;
    Aff_Val(P,x) ;
    Aff_Adr(P,L) ;
    CONS← P;
Fin ;
```

4.

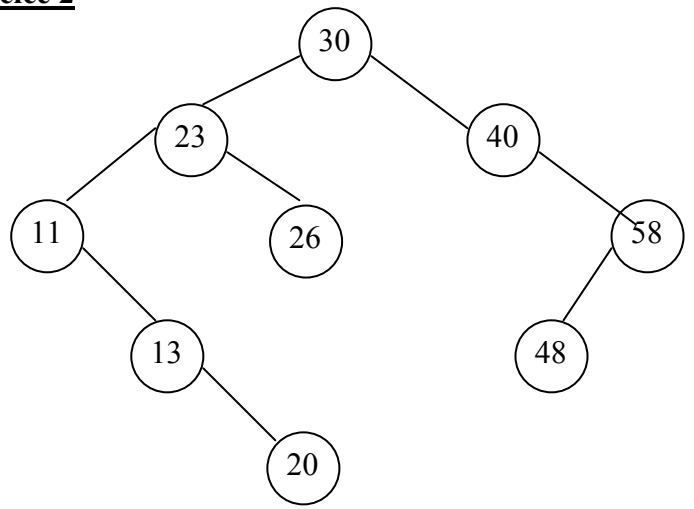
```
Fonction  Triée(L : Pointeur(TMaillon) : Booleen;
Var P : Pointeur(TMaillon) ;
Debut
   P ← L ;
Tantque P<>Nil et Suivant(P)<>Nil et Valeur(P)<Valeur(Suivant(P) Faire
      P ← Suivant(P) ;
 FTQ ;
Triée ← ((P=Nil) ou (Suivant(P)=Nil)) :
Fin ;
```

5.

```
Fonction  Fusion(L1,L2 :Pointeur(TMaillon)) : Pointeur(TMaillon) ;
Debut
    Si L1=Nil Alors  Fusion ← L2
    Sinon
      Si L2=Nil Alors  Fusion ← L1
      Sinon
        Si CAR(L1)<CAR(L2) alors
          Fusion ← CONS(CAR(L1), Fusion(CDR(L1),L2))
        Sinon
          Fusion ← CONS(CAR(L2), Fusion(L1,CDR(L2)))
        FSi ;
      Fin ;
    FSi
Fin;
```

Exercice 2

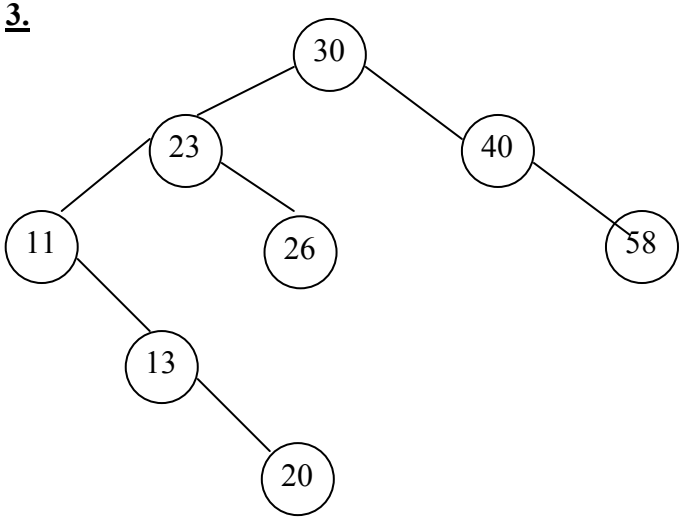
1.

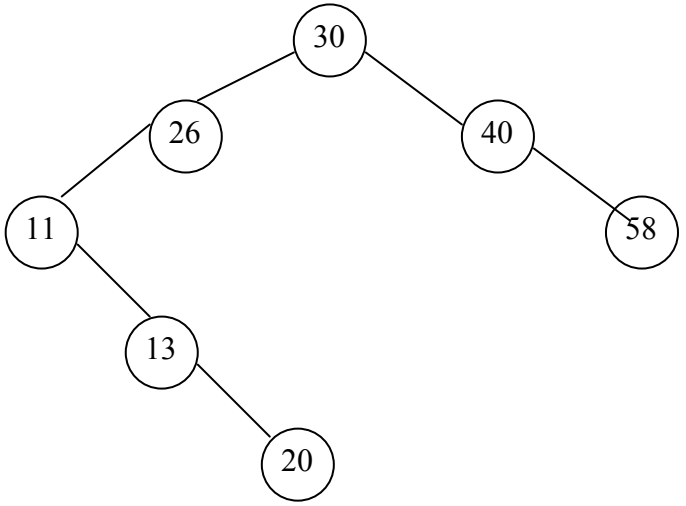


2.

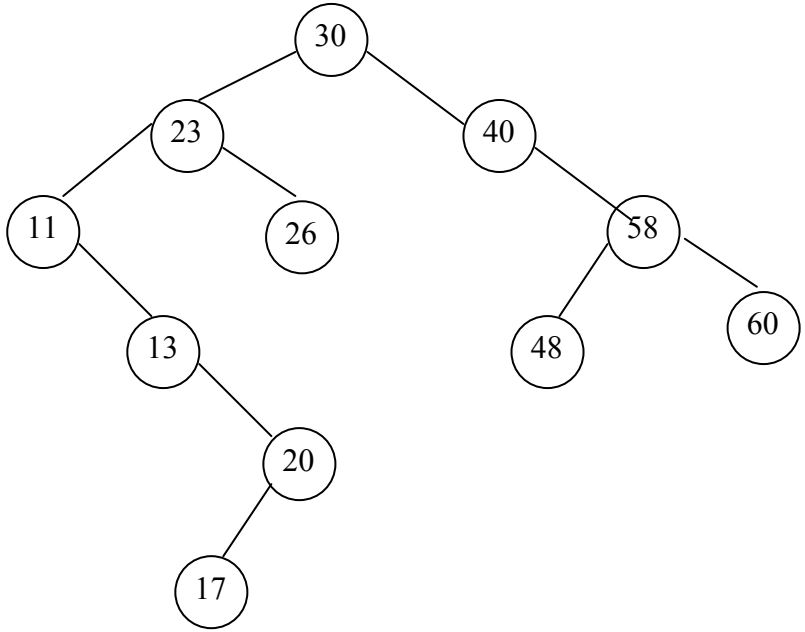
Préfixe : 30 23 11 13 20 26 40 58 48
Infixe : 11 13 20 23 26 30 40 48 58
Postfixe : 11 13 20 26 23 40 48 58 30

3.





4.



5.

```
Fonction   NB_Supp(R :Nœud, x :clé) :entier ;
Debut
   Si R=nil  alors  NB_Supp ← 0
   Sinon
     Si  Clé(R)>x alors
          NB_Supp ← 1 + NB_Supp(FG(R)) + NB_Supp(FD(R))
     Sinon
        NB_Supp ← NB_Supp(FG(R)) + NB_Supp(FD(R))
     FSi
   FSi
Fin;
```

6.

```
Fonction   Val_Supp(R :Nœud, x :clé) :clé ;
Var DG,N : Nœud ;
Debut
   N ← R ;
   TQ N<> Nil et clé(N) <> x Faire
       Si Clé(N)>x  alors  DG ← N ;  N← FG(N)
       Sinon    N← FD(N)
       FSi
   FTQ ;
   Si N=Nil  alors erreur
   Sinon
      N←FD(N);
      Si N=nil alors  Val_Supp ← Clé(DG)
      Sinon
         TQ FG(N) <> nil faire
           N← FG(N) ;
          FTQ
         Val_Supp ← Clé(N) ;
      FSi
    FSi
Fin ;
```

7.

```
Fonction   Fusion(R1,R2 :Nœud) :Noeud ;
Debut
   TQ R2<> Nil Faire
       Inser (R1, Clé(R2))
       Suppr(R2, Clé(R2))
   FTQ ;
  Fusion ← R1 ;
Fin;
```

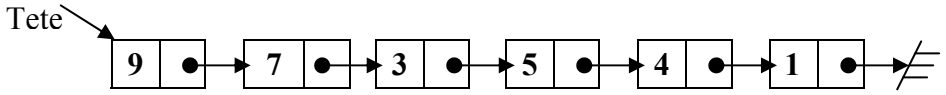
2LMD **10 Fev 2009**

Examen d'algorithmique 1

14h-15h30 **A2, A3**

Exercice 1 (10 pts: 4 + 3 + 3)

On souhaite représenter les nombres naturels par des listes linéaires chaînées où chaque chiffre d'un nombre est rangé dans un maillon de la liste :
Exemple :
La nombre "145379" est représentée par la liste suivante :



On suppose que les fonctions suivantes sont disponibles et peuvent être utilisées :

- Fonction Nbre(c :car) :entier retourne la valeur entière du caractère numérique c : Nbre('1')=1

- Fonction Car(N :entier) : car : retourne le caractère du naturel N<10 : Car(1) ='1'

- Fonction **Comparer**(N1, N2 : pointeur(TMaillon)) : entier
  Compare les deux entiers représentés par les deux listes N1 et N2 et retourne : -1 si N1<N2, 0 si N1=N2 et 1 si N1>N2.

- Fonction **Somme**(N1,N2 : pointeur(TMaillon)): pointeur(TMaillon) qui retourne une liste contenant la somme des deux naturels N1 et N2.

Il est demandé d'écrire les fonctions suivantes :

1. Fonction **Moins**(N1, N2 : pointeur(TMaillon)) : pointeur(TMaillon) ;
Retourne NIL si N1<=N2 et la liste représentant N1 – N2 sinon.

2. Fonction **Mult**(N1, N2 : pointeur(TMaillon)) : pointeur(TMaillon) ;
Retourne une liste représentant N1*N2.

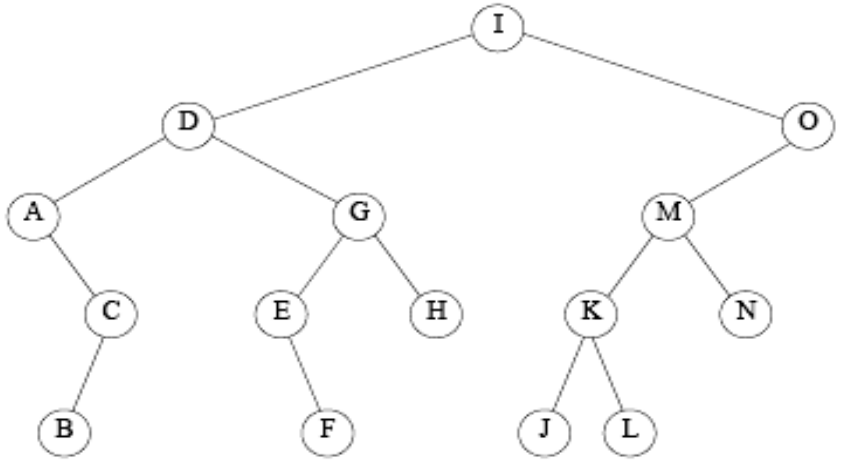
3. Fonction **Div**(N1, N2 : pointeur(TMaillon)) : pointeur(TMaillon) ;
Retourne NIL si N1<N2 et le résultat de la division sinon.

Exercice 2 (2.5 pts)

Ecrire la procédure **TrierTas**(T :Tableau[1..N]), permettant de trier le tableau T en *utilisant* un tas (utiliser sans implémentation les procedures InsererTas et RetirerTas)

Exercice 3 (7.5 pts : 1.5 + 1 + 2.5 + 2.5)

Soit l'arbre binaire de recherche suivant composé de caractères et basé sur l'ordre alphabétique :

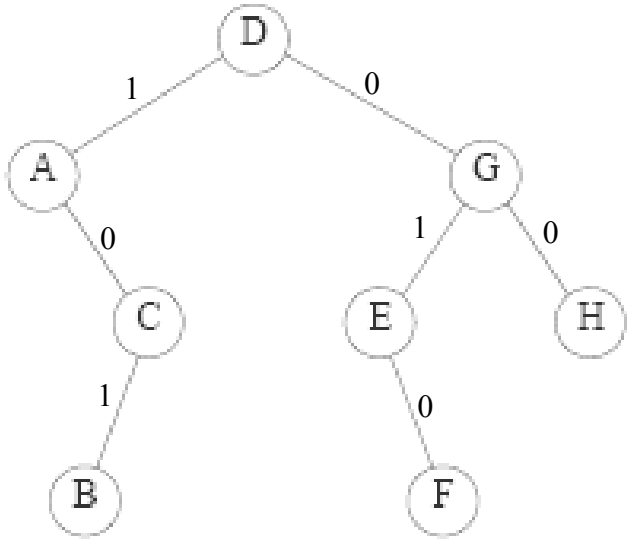


Tournez la page ../..

1. Donner les chaînes de caractères obtenus en parcourant l'arbre en profondeur en l'ordre préfixe, infixe puis postfixe.

2. Donner l'arbre après la suppression de 'D' puis de 'I'.

On associe à chaque caractère un code binaire représentant sa position dans l'arbre binaire comme suit :



A chaque fois qu'on passe à un fils gauche, on considère un 1 et chaque fois qu'on passe à un fils droit, on considère un 0.

Il est demandé d'écrire les deux fonctions suivantes :

3. Fonction **Code**(Racine: Pointeur(TNoeud),C: caractère): chaîne ;

   qui retourne le code binaire du caractère C dans une chaîne de caractères contenant des '0' et des '1'.
   Exemple : Code('C') = '10', Code('F') = '010'

4. Fonction **Caract**(Racine:Pointeur(TNoeud), B:chaine):caractère;

   qui retourne le caractère correspondant au code binaire donné dans la chaine B.
   Exemple : Caract('00') = 'H' ; Caract('101') = 'B'

N.B : On utilise la relation d'ordre entre les caractères alphabétiques, si C1 et C2 sont deux caractères, C1>C2 si C2 se trouve après C1 dans l'ordre alphabétique.

Bon courage

A. Djeffal

Corrigé type

Exercice 1

1. 4 Pts

```
Fonction Moins(N1,N2: pointeur(TMaillon)): pointeur(TMaillon);
Var  P, tete,Q : pointeur(TMaillon) ;
      V1, V2, Reste : entier ;
Debut
 Si Comparer(N1,N2)<1 alors  Moins ← NIL ;
 Sinon
     Reste ← 0 ;
     Tete ← NIL ;
     TQ N1≠NIL Faire
         Allouer(P) ;  Aff_Adr(P,NIL) ;
         Si Tete=NIL alors Tete ←P
                      Sinon Aff_Adre(Q,P) ;
         FSi
         V1← Nbre(Valeur(N1)) ;
         Si N2≠NIL alors V2 ← Nbre(Valeur(N2))
                     Sinon V2←0 ;
         FSi
         Si V1≥(V2+ Reste) alors
            Aff_Val(P, Car(V1- V2 – Reste))) ;
            Reste ← 0 ;
         Sinon
            Aff_Val(P, Car(V1 + 10 - V2 - Reste))) ;
            Reste ← 1 ;
         Fsi
         Q ← P ;
         N1 ← Suivant(N1) ;
         Si N2≠NIL alors N2 ← Suivant (N2) ;
     FTQ ;
     Mois ← tete ;
 FSi
Fin ;
```

2. 3 Pts

```
Fonction Mult(N1, N2 : pointeur(TMaillon)) : pointeur(TMaillon) ;
Var  Un, Resultat : pointeur(TMaillon) ;
Debut
 Si N1 = NIL ou N2 = NIL alors Mult ← NIL ;
 Sinon
     Allouer(Un) ; Aff_Adr(Un,Nil) ;
     Aff_Val(Un,'1') ;
     Resultat ← NIL ;
     TQ Comparer(N2,NIL) =1 Faire
            Resultat ← Somme (Resultats, N1) ;
            N2 ← Moins(N2, Un) ;
     FTQ ;
     Mult ← Resultat ;
  FSi
Fin ;
```

3. 3 Pts

```
Fonction Div(N1, N2 : pointeur(TMaillon)) : pointeur(TMaillon) ;
Var Un, Resultat, Reste : pointeur(TMaillon) ;
Debut
   Si Comparer(N1,N2)=-1 ou N1 = NIL ou N2 = NIL alors Div ← Nil ;
   Sinon
        Resultat ← Nil ;
        Reste ← N1 ;
        Allouer(Un) ; Aff_Adr(Un,Nil) ;
        Aff_Val(Un,'1') ;
        TQ Comparer(Reste,N2) ≥ 0 Faire
             Resultat ← Somme(Resultat, Un) ;
             Reste ← Moins (Reste, N2) ;
        FTQ
        Div ← Resultat ;
     FSi
Fin
```

Exercice 2 (2.5 pts)

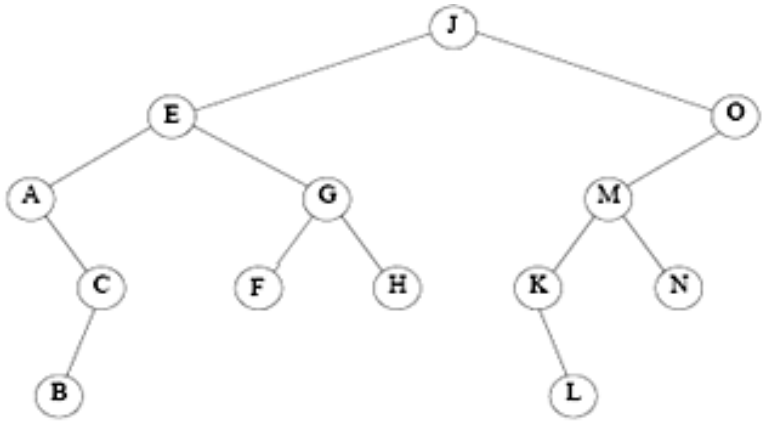
```
Procedure TrierTas(T :Tableau[1..N]) ;
Var  i : entier ;
Debut
     Init_Tas ;

     Pour i =1 à N faire
       Inserer_Tas(T[i]) ;
     FPour ;

     Pour i = 1 à N faire
        Retirer_Tas(T[i]) ;
     FPour ;

Fin ;
```

Exercice 3 (7.5 pts : + 1 + 2.5 + 2.5)

1. **1.5 Pt**

- Parcours en profondeur préfixe : IDACBGEFHOMKJLN
- Parcours en profondeur infixe : ABCDEFGHIJKLMNO
- Parcours en profondeur postfixe : BCAFEHGDJLKNMOI

2. **1 Pt**



3. **2.5 Pts**

```
Fonction Code(Racine: Pointeur(TNoeud),C: caractère): chaîne ;
Var N : pointeur(TNoeud) ;
     S : Chaine ;
Debut
    N ← Racine ;  S ← '' ;
    TQ N≠ NIL et Valeur(N) ≠ C Faire ;
        Si Valeur(N)>C et FG(N) ≠NIL alors
              S ← S + '1' ; N ← FG(N) ;
        FSi
        Si Valeur(N)<C et FD(N) ≠NIL alors
              S ← S + '0' ; N ← FD(N) ;
        FSi
    FTQ
    Si N =NIL alors Code ← '#'          // Erreur : caractère inexistant
                Sinon  Code ← S ;
    FSi
Fin ;
```

4. **2.5 Pts**

```
Fonction Caract(Racine:Pointeur(TNoeud), B:chaine):caractère;
Var N : pointeur(TNoeud) ;
      I : entier ;
 Debut
    N ← Racine ; I ← 1;
    TQ N≠ NIL et I <= Longeur(B) Faire ;
        Si B[I]='1' alors  N ← FG(N) ;
                     Sinon N ← FD(N) ;
        FSi
        I←I+1;
    FTQ
    Si N =NIL alors Caract ← '#'          // Erreur : code inexistant
                Sinon  Caract ← Valeur(N) ;
    FSi
Fin ;
```

2LMD **18 Mars 2009**

Examen de Rattrapage ALGO1

14h50-16h20 **Amphi 2**

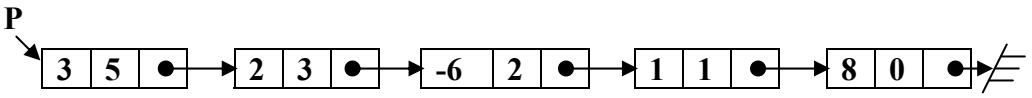
Exercice 1 (12 Pts: 1 + 2 + 3 + 3 + 3)

On convient de représenter les polynômes par des listes linéaires chaînées triées dans l'ordre décroissant des puissances où chaque maillon contient un exposant et le cœfficient correspondant.

Exemple: Le polynôme

$$P(x) = 3x^5 + 2x^3 - 6x^2 + x + 8$$

est représenté par la liste P suivante :



Ecrire les fonctions suivantes :

1. fonction **Degré(P :Pointeur(TMaillon)) :entier** qui retourne le degré du polynôme P donné.
2. fonction **PValeur(P :Pointeur(TMaillon), x : reel) :reel** qui retourne la valeur du polynôme P pour x donné. (utiliser x ^ y = x^y)
3. fonction **Somme(P1,P2 :Pointeur(TMaillon)): Pointeur(TMaillon))** qui retourne la somme des deux polynômes P1 et P2.
4. fonction **Dérivé(P:Pointeur(TMaillon)): Pointeur(TMaillon))** qui retourne le polynôme égal à la dérivée du polynôme P.
5. fonction **Intégrale(P:Pointeur(TMaillon)): Pointeur(TMaillon))** qui retourne le polynôme égal à l'intégrale du polynôme P.

N.B : Utiliser les primitives suivantes:

Aff_ValC(P,c) pour changer le coefficient d'un maillon,

Aff_ValE(P,e) pour changer l'exposant d'un maillon,

ValC(P) pour lire le coefficient d'un maillon,

ValE(P) pour lire l'exposant d'un maillon.

Exercice 2 (3 Pts: 1.5 + 1 + 0.5)

1. Dessiner le tas obtenu en insérant les éléments de la liste suivante dans l'ordre d'arrivée: 10, 5, 12, 8, 7, 1, 2, 4, 6.
2. Donner le tableau représentant le tas statique correspondant.
3. Dessiner le tas après un retrait.

Exercice 3 (5 Pts : 2 + 3)

On dit qu'un arbre de recherche binaire est équilibré si pour tout nœud de l'arbre, la différence entre la hauteur du sous-arbre gauche et du sous-arbre droit est d'au plus égale à 1.

1. Ecrite la fonction **Hauteur(R: Pointeur(TNoeud)): Entier** qui retourne la hauteur de l'arbre de recherche binaire de racine R.
2. Ecrire la fonction **Equilibré(R : Pointeur(TNoeud)) : Booleen** qui retourne vrai si l'arbre de recherche binaire de racine R est équilibré et faux sinon.

Bon courage

A.Djeffal

Corrigé type

Exercice 1 (12 Pts: 1 + 2 + 3 + 3 + 3)

1. (1 Pt)

```
fonction Degré(P :Pointeur(TMaillon)) :entier ;
Debut
    Si P=Nil alors Degré ← 0
              Sinon Degré ← ValE(P)
    FSi
Fin ;
```

2. (2 Pts)

```
fonction PValeur(P :Pointeur(TMaillon), x : reel) :reel
Var S : reel ;
Debut
    S←0 ;
    TQ P ≠ Nil Faire
        S ← S + ValC(P)*(x ^ ValE(P));
        P ← Suivant(P);
    FTQ
    PValeur ← S ;
Fin ;
```

3. (3 Pts)

```
fonction Somme(P1,P2 :Pointeur(TMaillon)): Pointeur(TMaillon)) ;
Var Tete_Sm, Q, P, P12 : Pointeur(TMaillon);
Debut

  Tete_Sm ← Nil ;   // tête de la liste somme
  Q ← Nil ;         // dernier élément de la liste somme

TQ P1 ≠ Nil et P2 ≠ Nil Faire
  Allouer (P) ; Aff_Adr(P,Nil) ;
  Si Tete_Sm=Nil alors Tete_Sm ← P ;
                  Sinon Aff_Adr(Q,P) ;
  FSi
  Q ← P;
  Si ValE(P1) = ValE(P2) alors
      Aff_ValE(P, ValE(P1)) ; Aff_ValC(P, ValC(P1) + ValC(P2)) ;
      P1 ← Suivant(P1);
      P2 ← Suivant(P2);
  Sinon
      Si ValE(P1) > ValE(P2) alors
          Aff_ValE(P, ValE(P1)) ; Aff_ValC(P, ValC(P1)) ;
          P1 ← Suivant(P1);
      Sinon
          Aff_ValE(P, ValE(P2)) ; Aff_ValC(P, ValC(P2)) ;
          P2 ← Suivant(P2);
      FSi
  FSi
FTQ ;

Si P1  ≠ alors P12 ← P1          // P12 pointe l'éventuel reste de l'une des
          Sinon P12 ← P2 ;       // deux listes
FSi

TQ P12 ≠ Nil  Faire                  // recopier le reste de la liste
  Allouer (P) ; Aff_Adr(P,Nil) ;
  Si Tete_Sm=Nil alors Tete_Sm ← P ;
                  Sinon Aff_Adr(S,P) ;
  FSi
  Q ← P;
  Aff_ValE(P, ValE(P12)) ; Aff_ValC(P, ValC(P12)) ;
  P12 ← Suivant(P12);
FTQ
Somme ← Tete_Sm ;
Fin ;
```

4. (3 Pts)

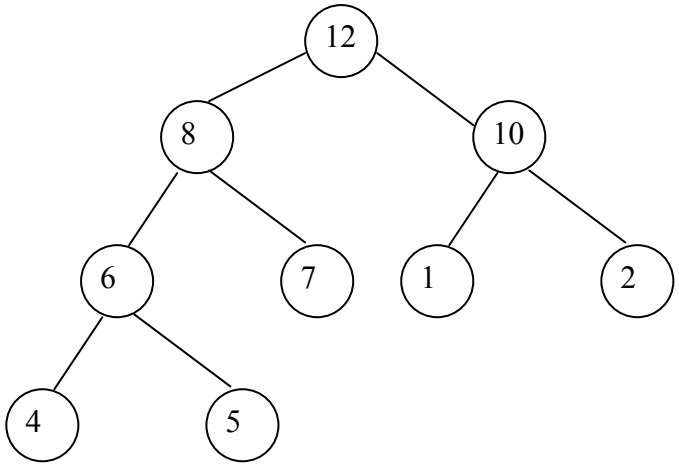
```
fonction Dérivé(P:Pointeur(TMaillon)): Pointeur(TMaillon))
Var Tete_D, Q, P1: Pointeur(TMaillon);
Debut
  Tete_D ← Nil ;   // tête de la liste dérivée
  Q ← Nil ;            // dernier élément de la liste dérivé
  TQ P ≠ Nil et ValE(P) ≠ 0  Faire
    Allouer (P1) ; Aff_Adr(P1,Nil) ;
    Si Tete_D = Nil alors Tete_D ← P1 ;
                          Sinon Aff_Adr(Q,P1) ;
    FSi
    Q ← P1;
    Aff_ValC(P1, ValC(P) * ValE(P)) ;
    Aff_ValE(P1, ValE(P) - 1) ;
    P ← Suivant(P);
  FTQ ;
  Dérivé ← Tete_D ;
Fin ;
```

5. (3 Pts)

```
fonction Intégrale(P:Pointeur(TMaillon)): Pointeur(TMaillon))
Var Tete_Int, Q, P1: Pointeur(TMaillon);
Debut
  Tete_Int ← Nil ;   // tête de la liste intégrale
  Q ← Nil ;            // dernier élément de la liste intégrale

  TQ P ≠ Nil   Faire
    Allouer (P1) ; Aff_Adr(P1,Nil) ;
    Si Tete_D = Nil alors Tete_D ← P1 ;
                          Sinon Aff_Adr(Q,P1) ;
    FSi
    Q ← P1;
    Aff_ValC(P1, ValC(P) / (ValE(P) + 1)) ;
    Aff_ValE(P1, ValE(P) + 1) ;
    P ← Suivant(P);
  FTQ ;
    // Ajouter une constante C (supposons 1)
  Allouer (P1) ; Aff_Adr(P1,Nil) ;
  Si Tete_D = Nil alors Tete_D ← P1 ;
                        Sinon Aff_Adr(Q,P1) ;
  FSi
  Aff_ValC(P1, 1) ;
  Aff_ValE(P1, 0) ;
  Intégrale ← Tete_Int;
Fin ;
```

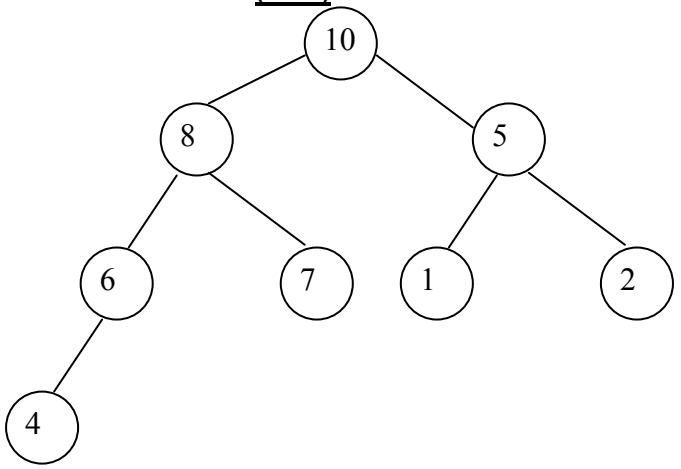
Exercice 2 (3 Pts: 1.5 + 1 + 0.5)

1. (1.5 Pt)



2. Le tas statique : 12 8 10 6 7 1 2 4 5 **(1 Pt)**

3. (0.5 Pt)



Exercice 3 (5 Pts : 2 + 3)

1. (2 Pts)

```
fonction Hauteur(R: Pointeur(TNoeud)): Entier
Debut
 Si R = Nil alors Hauteur ← 0
 Sinon Hauteur ← 1 + Max ( Hauteur (FG(R)) , Hauteur(GD(R)) )
FSi
Fin ;
```

2. (3 Pts)

```
fonction Equilibré(R : Pointeur(TNoeud)) : Booleen
Debut
 Si R = Nil alors Equilibré ← Vrai
 Sinon
    Equilibré ← (Abs(Hauteur (FG(R)) - Hauteur(GD(R)) ) ≤ 1) et
                        Equilibré(FG(R)) et Equilibré(FD(R)) ;
 FSi
Fin ;
```

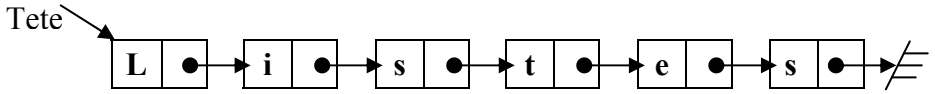
2LMD **11 Fev 2010**

Examen d'algorithmique 1

08h-09h30 **A1, A2**

Exercice 1 (10 pts: 1.5 + 1.5 + 3 + 1.5 + 2 .5)

On souhaite représenter les chaînes de caractères par des listes linéaires chaînées où chaque caractère d'une chaîne est rangé dans un maillon de la liste :
Exemple :
La chaîne de caractères "Listes" est représentée par la liste suivante :



Il est demandé d'écrire en langage algorithmique vu dans le cours les fonctions suivantes :

1. Fonction **NbreOccur**(L :Pointeur(TMaillon), C : caractère) : entier
Qui retourne le nombre d'occurrences du caractère C dans la chaîne de caractères représentée par la liste de tête L.

2. Fonction **Concat**(L1, L2 : Pointeur(TMaillon)) : Pointeur(TMaillon) ;
Qui retourne la liste représentant le résultat de concaténation des deux chaines représentées par L1 et L2.

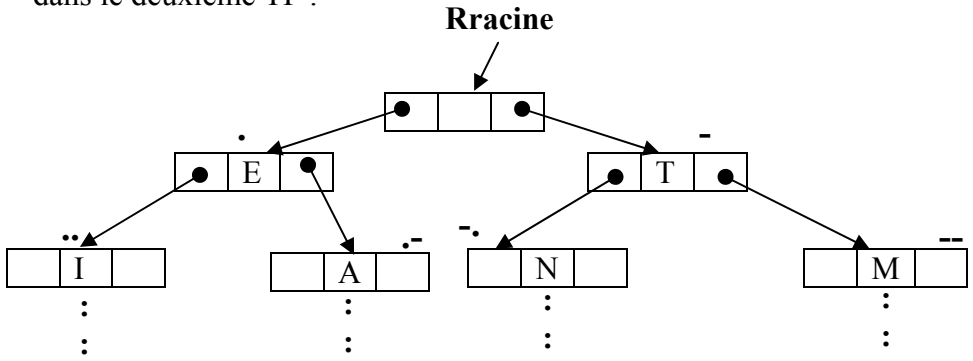
3. Fonction **Retirer**(L:Pointeur(TMaillon), C:caractère):Pointeur(TMaillon);
Qui retourne une liste égale à la liste L sans le caractère C.

4. Fonction **Position**(L:Pointeur(TMaillon), C:caractère):Pointeur(TMaillon);
Qui retourne un pointeur sur la première occurrence du caractère C dans la liste L.

5. Fonction **Remplacer**(L:Pointeur(TMaillon), C1,C2:caractère): Pointeur(TMaillon);
Qui retourne une liste égale à L avec toutes les occurrences de C1 remplacées par C2.

Exercice 2 (10 pts : 3 + 1.5 + 1.5 + 4)

Soit l'arbre binaire de recherche représentant les codes morse vu dans le deuxième TP :



Ecrire les fonctions suivantes :

1. Fonction **NbrePréfixes** (Racine: Pointeur(TNoeud),Ch: chaîne):entier ;
Qui retourne le nombre de caractères dont le code commence par la chaîne Ch.

2. Fonction **NbrePoints** (Racine: Pointeur(TNoeud)):entier ;
Qui retourne le nombre de points dans tous les codes de l'arbre.

3. Fonction **NbreTirets** (Racine: Pointeur(TNoeud)):entier ;
Qui retourne le nombre de tirets dans tous les codes de l'arbre.

4. Procedure **Ajouter** (Racine: Pointeur(TNoeud),C:Caractère, Code:chaîne) ;
Qui ajoute le caractère C de code Code à l'arbre.

Bon courage

Corrigé type

Exercice 1 (10 pts: 1.5 + 1.5 + 3 + 1.5 + 2 .5)

1. Fonction **NbreOccur**(L :Pointeur(TMaillon), C : caractère) : entier

```
Var P : Pointeur(TMaillon) ;
    Nb : entier ;
Debut
  P ← L ;
  Nb ← 0 ;
  TQ P≠NIL faire
    Si Valeur(P) = C alors
       Nb ← Nb + 1
    Fsi ;
    P ← Suivant(P) ;
  FTQ ;
  NbreOccur ← Nb ;
Fin ;
```

(1.5 Pt)

2. Fonction **Concat**(L1, L2 : Pointeur(TMaillon)) : Pointeur(TMaillon) ;

```
Var P, P1, NTete, Q : Pointeur(TMaillon) ;
Debut
  P ← L1 ;
  NTete ← NIL ;
  TQ P≠NIL  faire
    Allouer(P1) ;
    Aff_Val(P1,Valeur(P));
    Aff_Adr(P1,NIL) ;
    Si NTete=NIL alors  NTete ← P1
                  Sinon Aff_Adr(Q,P1)
    Fsi
    Q ← P1 ;
    P ← Suivant(P) ;
  FTQ ;
  P ← L2 ;
  TQ P≠NIL  faire
    Allouer(P1) ;
    Aff_Val(P1,Valeur(P));
    Aff_Adr(P1,NIL) ;
    Si NTete=NIL alors  NTete ← P1
                  Sinon Aff_Adr(Q,P1)
    Fsi
    Q ← P1 ;
    P ← Suivant(P) ;
  FTQ ;
  Concat ← NTete ;
Fin ;
```

(1.5 Pt)

3. Fonction **Retirer**(L:Pointeur(TMaillon), C:caractère):Pointeur(TMaillon);

```
Var P, P1, NTete, Q : Pointeur(TMaillon) ;
Debut
  P ← L1 ;
  NTete ← NIL ;
  TQ P≠NIL  faire
    Si Valeur(P) ≠ C alors
      Allouer(P1) ;
      Aff_Val(P1,Valeur(P));
      Aff_Adr(P1,NIL) ;
      Si NTete=NIL alors  NTete ← P1
                    Sinon Aff_Adr(Q,P1)
      Fsi
      Q ← P1 ;
    Fsi
    P ← Suivant(P) ;
  FTQ ;
  Retirer ← NTete ;
Fin ;
```

(3 Pts)

4.Fonction **Position**(L:Pointeur(TMaillon), C:caractère):
Pointeur(TMaillon);

```
Var P : Pointeur(TMaillon) ;
Debut
   P ← L ;
   TQ P≠NIL et Valeur(P)≠C faire
      P ← Suivant(P) ;
   FTQ ;
   Position ← P;
Fin ;
```

(1.5 Pt)

5. Fonction **Remplacer**(L:Pointeur(TMaillon), C1,C2:caractère):
Pointeur(TMaillon);

```
Var P, P1, NTete, Q : Pointeur(TMaillon) ;
Debut
   P ← L;
   NTete ← NIL ;
   TQ P≠NIL  faire
      Allouer(P1) ;
      Si Valeur(P) = C1 alors Aff_Val(P1,C2);
                        Sinon Aff_Val(P1,Valeur(P));
      Fsi ;
      Aff_Adr(P1,NIL) ;
      Si NTete=NIL alors  NTete ← P1
                    Sinon Aff_Adr(Q,P1)
      Fsi
      Q ← P1 ;
      P ← Suivant(P) ;
   FTQ ;
   Remplacer ← NTete ;
Fin ;
```

(2.5 Pt)

Exercice 2 (10 pts : 3 + 1.5 + 1.5 + 4)

1. Fonction **NbrePréfixes** (Racine: Pointeur(TNoeud),Ch: chaîne):entier ;

```
Var  R : Pointeur(TNoeud) ;
     Ch1 : Chaine ;
     i : entier ;
Debut
   Si Racine = NIL alors NbrePréfixes ← 0
   Sinon
      Si Ch='' alors NbrePrefixes ← Taille(Racine)
      Sinon
         Si Ch[1] = '.' alors  R ← FG(Racine)
                       Sinon R ← FD(Racine)
         Fsi
         Ch1 ← ''
         Pout i ←2 à Longueur(Ch) faire
            Ch1 ← Ch1 + Ch[i];
         FPour ;
         NbrePréfixes ← NbrePréfixes(R, Ch1) ;
      Fsi;
   Fsi
Fin ;

Fonction Taille (R : Pointeur(TNoeud)) :entier ;
Debut
   Si R = NIL alors Taille ← 0
   Sinon
      Taille ← 1 + Taille(FG(R)) + Taille(FD(R))
   Fsi
Fin ;
```

(3 Pts)

2. Fonction **NbrePoints** (Racine: Pointeur(TNoeud)):entier ;

```
Var Nb :entier ;
Debut
   Nb ← 0 ;
   NbrePoints ← NbPts(Racine, Nb) ;
Fin ;
```

```
Fonction NbPts(R : Pointeur(TNoeud), N :entier) : entier ;
Var NG, ND :entier ;
Debut
  Si R = NIL alors NbPts ← N
  Sinon
    Si FG(R) = Nil  alors   NG ← 0 ;
                    Sinon   NG ← NbPts(FG(R), N+1)
    Fsi ;
    Si FD(R) = Nil  alors   ND ← 0 ;
                    Sinon   ND ← NbPts(FD(R), N)
    Fsi ;
     NbPts ← N + NG + ND ;
  Fsi
Fin ;                                          (1.5 Pt)
```

3. Fonction **NbreTirets** (Racine: Pointeur(TNoeud)):entier ;

```
Var Nb :entier ;
Debut
  Nb ← 0 ;
  NbreTirets ← NbTrs(Racine, Nb) ;
Fin ;

Fonction NbTrs(R : Pointeur(TNoeud), N :entier) : entier ;
Var NG, ND :entier ;
Debut
  Si R = NIL alors NbPts ← N
  Sinon
    Si FG(R) = Nil  alors   NG ← 0 ;
                    Sinon   NG ← NbTrs(FG(R), N)
    Fsi ;
    Si FD(R) = Nil  alors   ND ← 0 ;
                    Sinon   ND ← NbTrs(FD(R), N+1)
    Fsi ;
    NbTrs ← N + NG + ND ;
  Fsi
Fin ;                                          (1.5 Pt)
```

4. Procedure **Ajouter** (Racine: Pointeur(TNoeud),C:Caractère, Code:chaîne) ;

```
Var
Debut
  Si Racine = NIL alors
     Allouer (Racine ) ;
     AFF_FG(Racine, NIL) ;
     AFF_FD(Racine,NIL) ;
  Fsi ;
  N ← Racine ;
  PN ← NIL ;
  i ← 1 ;
  TQ N ≠ NIL et i ≤ Longueur(Code) Faire
     PN ← N ;
     Si Code[i] = '.' Alors  N ← FG(N)
                     Sinon  N ← FD(N)
     Fsi ;
     i ← i + 1
  FTQ ;
  Si N = NIL alors
        Pour j = i -1 à Longueur(Code) Faire
           Allouer(N1) ;
           AFF_FG(N1, NIL) ;
           AFF_FD(N1, NIL) ;
           Si Code[j] = '.' Alors AFF_FG(PN, N1) ;
                          Sinon AFF_FD(PN, N1) ;
           Fsi
          PN ← N1 ;
        FPour ;
        AFF_Info(Q, C);
  Sinon
          AFF_Info(N, C) ;
  Fsi
Fin ;                                          (4 Pts)
```

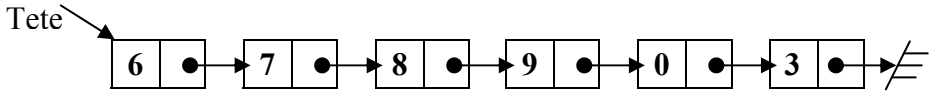
2LMD **24 Avr 2010**

Rattrapage d'algorithmique 1

08h-09h30 **A1, A2**

Exercice 1 (7 pts: 1 + 2 + 2 + 2)

On souhaite représenter les nombres entiers par des listes linéaires chaînées de caractères où chaque chiffre d'un nombre est rangé dans un maillon de la liste sous forme d'un caractère:
Exemple :
Le nombre "678903" est représentée par la liste suivante :



Il est demandé d'écrire en langage algorithmique vu dans le cours les fonctions suivantes :

1. Fonction **Pair** (L :Pointeur(TMaillon)) : Booleen
Qui retourne Vrai si le nombre représenté par L est pair, et Faux sinon.

2. Fonction **Mult100**(L: Pointeur(TMaillon)) : Pointeur(TMaillon) ;
Qui retourne une nouvelle liste représentant le nombre représenté par L multiplié par 100.

3. Fonction **Div100**(L:Pointeur(TMaillon)):Pointeur(TMaillon);
Qui retourne une nouvelle liste représentant le nombre représenté par L divisé par 100.

4. Fonction **Mod100**(L: Pointeur(TMaillon)):Pointeur(TMaillon);
Qui retourne une nouvelle liste représentant le reste de division du nombre représenté par L par 100.

On donne les fonctions:
CarVersEnt(C :caractère) : donne la valeur entière du caractère C
CarVersEnt('8') = 8
EntVersCar(I :Entier) : donne le caractère correspondant au chiffre I
EntVersCar(8) = '8'

Exercice 2 (8 pts : 1.5 + 2 + 2 + 2.5)

Soit les valeurs des clés triés selon l'ordre d'arrivé :
14, 23, 4, 9, 17, 11, 28, 16, 3, 7

- Dessiner l'arbre binaire de recherche obtenu en insérant ces clés dans l'ordre d'arrivé

- Dessiner l'arbre après la après la suppression des clés : 11, 17, 23, 14

Ecrire les fonctions suivantes :

1. Fonction **NbrePairs** (Racine: Pointeur(TNoeud)):entier ;
Qui retourne le nombre de clés paires dans l'arbre.

2. Fonction **SommeInterne** (Racine: Pointeur(TNoeud)):entier ;
Qui retourne la somme des clés non feuilles dans l'arbre.

Exercice 3 (5 pts : 3 + 1 + 1)

- Dessiner l'arbre représentant le tas construit après l'insertion dans l'ordre des clés suivants : 14, 23, 4, 9, 17, 11, 28, 16, 3, 7

- Donner le tas statique correspondant

- Donner le tas (arbre) après un retrait

Bon courage

Corrigé type

Exercice 1 (7 pts: 1 + 2 + 2 + 2)

1. Fonction **Pair** (L :Pointeur(TMaillon)) : Booleen

```
Var  P : Pointeur(TMaillon) ;
Debut
    P ← L ;
    Si P=Nil alors Pair ← Vrai
    Sinon
      TQ Suivant(P) ≠ Nil Faire
        P ← Suivant(P) ;
      FTQ ;
      Si CarVersEnt(Valeur(P)) mod 2 = 0 alors Pair ← Vrai
      Sinon
          Pair ← Faux ;
      FSi
    FSi
Fin ;
```

2. Fonction **Mult100**(L: Pointeur(TMaillon)) : Pointeur(TMaillon) ;

```
 Var  P,P1,Q,Tete : Pointeur(TMaillon) ;
Debut
      P ← L ; Tete ← Nil ;
      TQ P ≠ Nil Faire
        Allouer (P1) ; Aff_Val(P1, Valeur(P)) ; Aff_Adr(P1,Nil) ;
        Si Tete=Nil alors Tete ← P1
                    Sinon  Aff_Adr(Q,P1)
        FSi;
        Q← P1 ;  P ←Suivant(P) ;
      FTQ ;
      Si Tete ≠ Nil alors
          Allouer(P) ; Aff_Val(P,'0') ; Aff_Adr(Q,P) ; Q ← P;
          Allouer(P) ; Aff_Val(P,'0') ; Aff_Adr(Q,P) ; Aff_Adr(P, Nil);
      FSi
    Mult100 ← Tete ;
Fin ;
```

3. Fonction **Div100**(L:Pointeur(TMaillon)):Pointeur(TMaillon);

```
 Var  P,P1, Q,Tete : Pointeur(TMaillon) ;
Debut
      P ← L ; Tete ← Nil ;
      TQ (P ≠ Nil et Suivant(P) ≠ Nil et Suiavnt(Suivant(P)) ≠ Nil) Faire
        Allouer (P1) ; Aff_Val(P1, Valeur(P)) ; Aff_Adr(P1,Nil) ;
        Si Tete=Nil alors Tete ← P1
                    Sinon  Aff_Adr(Q,P1)
        FSi;
        Q← P1 ;  P ←Suivant(P) ;
      FTQ ;
    Div100 ← Tete ;
Fin ;
```

4. Fonction **Mod100**(L: Pointeur(TMaillon)):Pointeur(TMaillon);

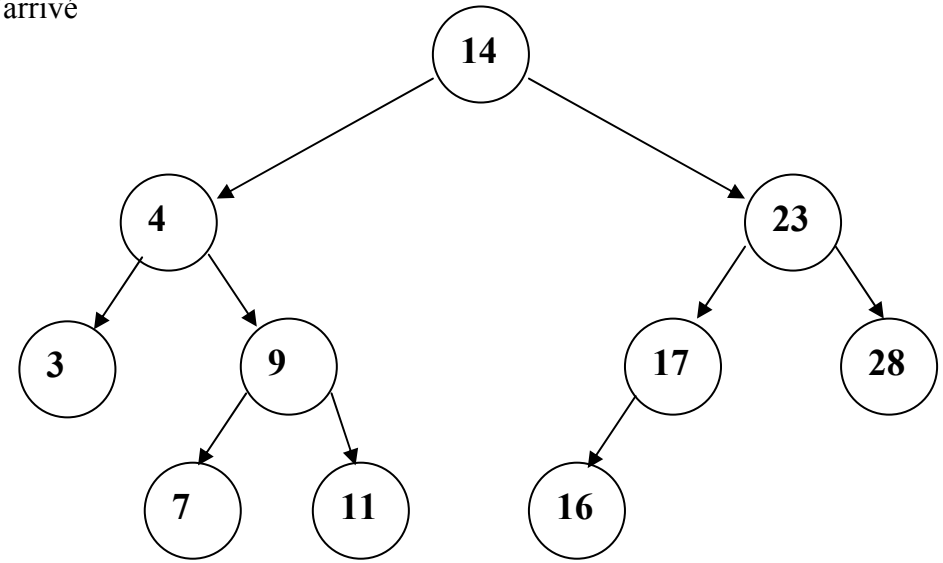
```
 Var  P,P1, Q,Tete : Pointeur(TMaillon) ;
Debut
      P ← L ; Tete ← Nil ;
      TQ (P ≠ Nil et Suivant(P) ≠ Nil et Suiavnt(Suivant(P)) ≠ Nil) Faire
        P ←Suivant(P) ;
       FTQ ;
      Tete ← Nil ;
      TQ P ≠ Nil Faire
        Si Val(P) ≠ '0' ou Q ≠Nil alors
            Allouer (P1) ; Aff_Val(P1, Valeur(P)) ; Aff_Adr(P1,Nil) ;
            Si Tete=Nil alors Tete ← P1
                        Sinon  Aff_Adr(Q,P1)
            FSi;
            Q← P1 ;
         FSi
         P ←Suivant(P) ;
      FTQ ;
    Mod100 ← Tete ;
Fin ;
```

Exercice 2 (8 pts : 1.5 + 2 + 2 + 2.5)

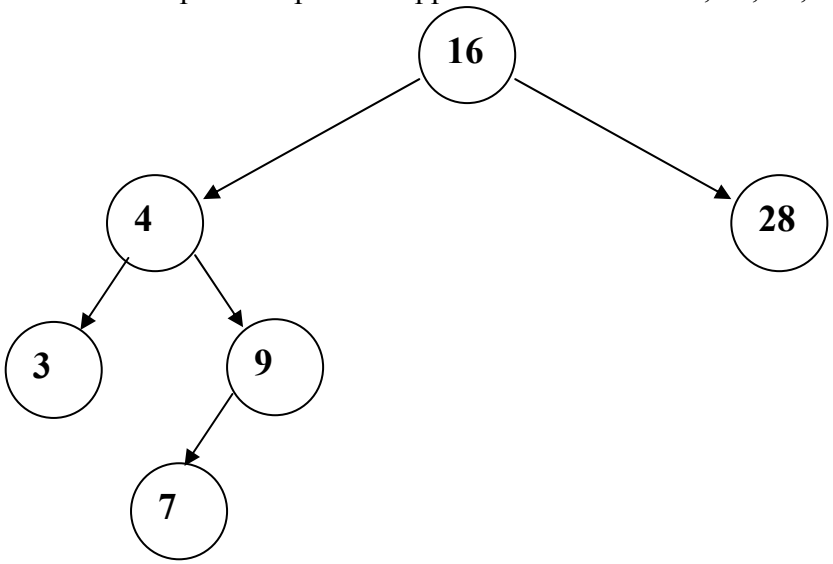
Soit les valeurs des clés triés selon l'ordre d'arrivé :

14, 23, 4, 9, 17, 11, 28, 16, 3, 7

- Dessiner l'arbre binaire de recherche obtenu en insérant ces clés dans l'ordre d'arrivé



- Dessiner l'arbre après la après la suppression des clés : 11, 17, 23, 14



Ecrire les fonctions suivantes :

1. Fonction **NbrePairs** (Racine: Pointeur(TNoeud)):entier ;

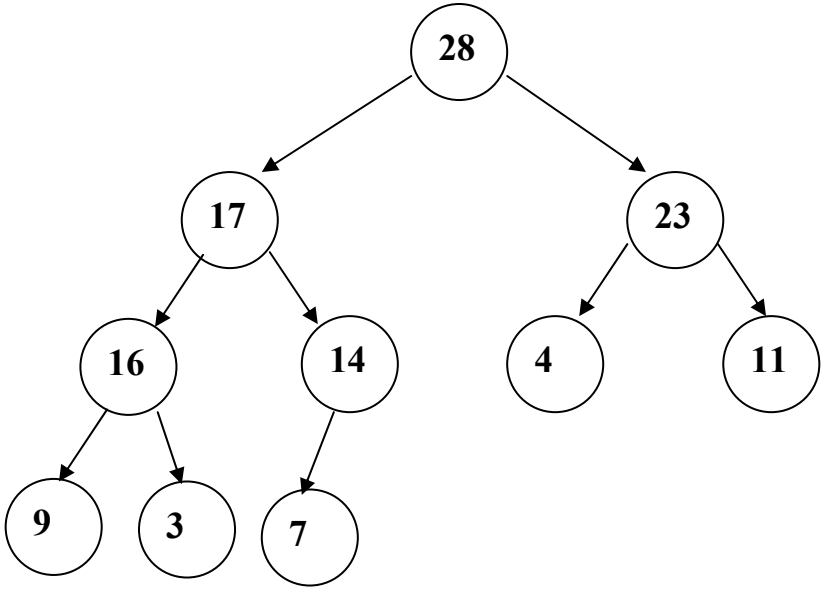
```
Debut
   Si Racine = Nil alors NbrePairs ← 0
   Sinon
      NbrePairs ← ((Clé(Racine) +1) mod 2) + NbrePairs(FG(Racine)) +
                  NbrePairs(FD(Racine)) ;
    FSi ;
Fin ;
```

2. Fonction **SommeInterne** (Racine: Pointeur(TNoeud)):entier ;

```
Debut
   Si Racine = Nil ou (FG(Racine)=Nil et FD(Racine)=Nil) alors
               SommeInterne ← 0
   Sinon
    SommeInterne ← (Clé(Racine) + SommeInterne(FG(Racine)) +
                    SommeInterne(FD(Racine)) ;
    FSi ;
Fin ;
```

Exercice 3 (5 pts : 3 + 1 + 1)

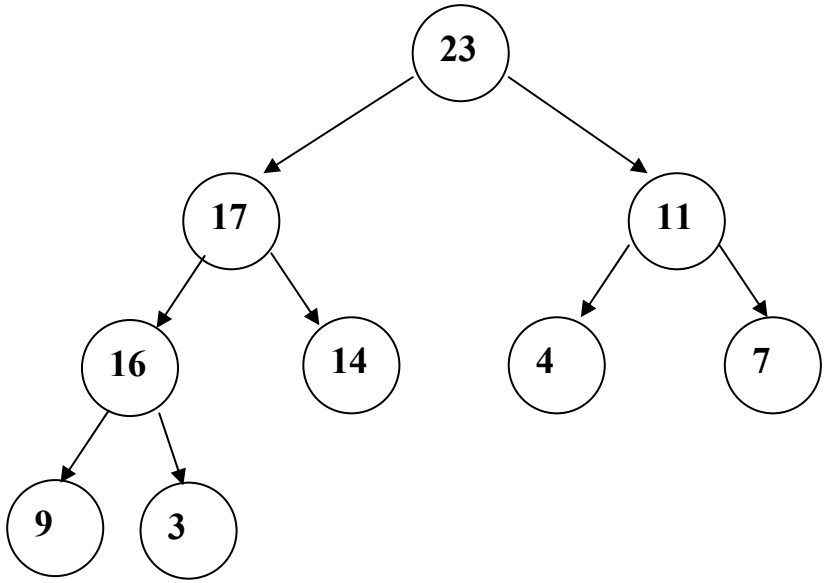
- Dessiner l'arbre représentant le tas construit après l'insertion dans l'ordre des clés suivants : 14, 23, 4, 9, 17, 11, 28, 16, 3, 7



- Donner le tas statique correspondant

28	17	23	16	14	4	11	9	3	7

- Donner le tas (arbre) après un retrait



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

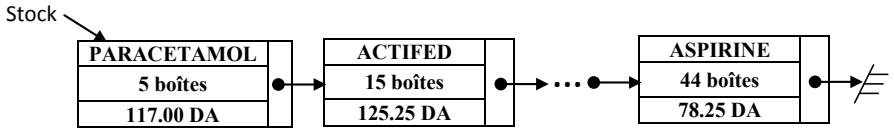
2ème année LMD — ALGO1
29 Jan 2012 — 8:00-9:30, Amphi H

Examen

Exercice 1 LLCs (8 pts : 1 + 3 + 2.5 + 1.5)

Un pharmacien souhaite traiter les informations concernant son stock de médicaments par ordinateur. On vous propose de représenter ces informations sous forme de liste linéaire chaînée où chaque maillon contient le libellé d'un médicament, la quantité disponible (nombre de boîtes) et le prix unitaire.

Exemple :



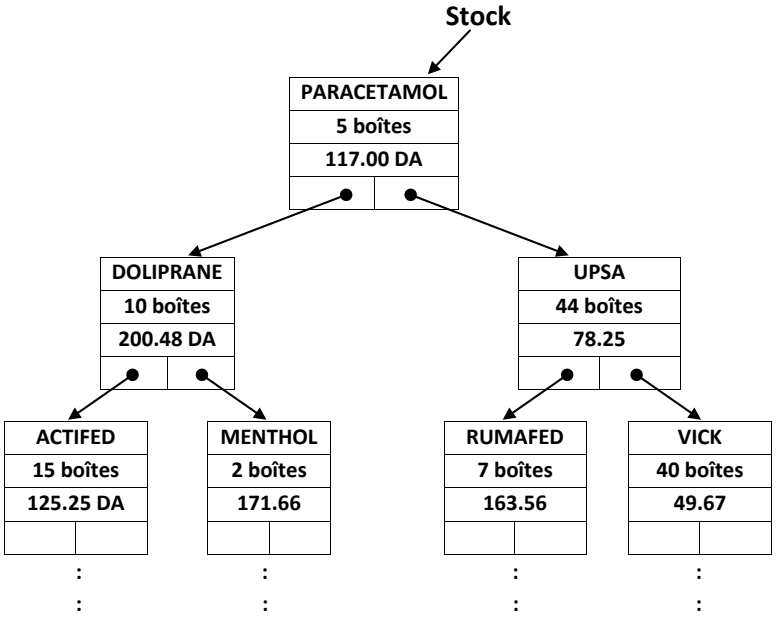
On vous demande de :

1. Donner les structures de données nécessaires à la représentation de ce stock.
2. Ecrire la procédure **Vendre(Med, NbBoites)** permettant de retirer, si possible, 'NbBoites' du médicament 'Med' du stock.
   Attention : Il faut supprimer du stock le médicament dont la quantité atteint 0.
3. Ecrire la procédure **Acheter(Med, NbBoites, Prix)** permettant au pharmacien d'alimenter son stock par 'NbBoites' du médicament 'Med' ayant le prix unitaire 'Prix' DA. On considère qu'un médicament prenne toujours le nouveau prix.
4. Ecrire la fonction **PrixStock** permettant de calculer le prix total des médicaments dans le stock.

Exercice 2 ARB(9 pts : 1 + 3 + 3 + 2)

Le pharmacien, appréciant la méthode des LLCs, a remarqué rapidement la lenteur des procédures Vendre et Acheter et il vous demande une solution plus rapide. On vous propose de réécrire la solution précédente en utilisant un arbre de recherche binaire basé sur l'ordre alphabétique des libellés des médicaments.

Exemple :

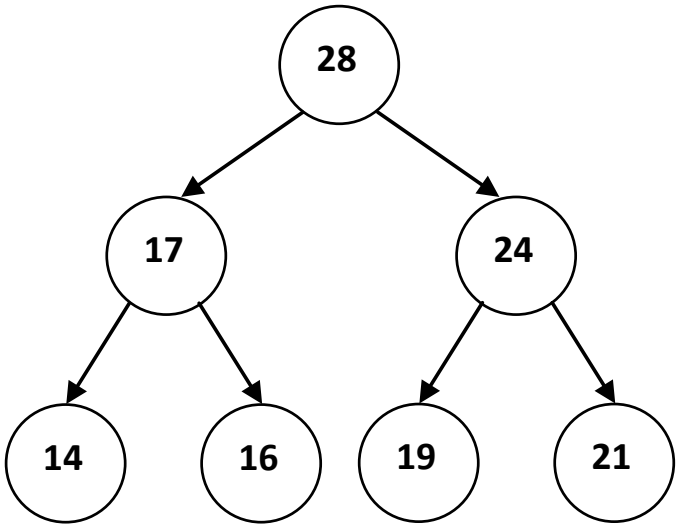


On vous demande de :

1. Donner les structures de données nécessaires à la représentation de ce stock.
2. Réécrire la procédure **Vendre** décrite dans l'exercice précédent sur la nouvelle structure.
   NB : On donne la procédure **Nettoyer (Stock)** permettant de supprimer du stock tous les médicaments ayant une quantité nulle.
3. Réécrire la procédure **Acheter** décrite dans l'exercice précédent sur la nouvelle structure.
4. Réécrire la fonction **PrixStock** décrite dans l'exercice précédent sur la nouvelle structure.

Exercice 3 (3 pts : 1 + 1 + 1)

Soit l'arbre binaire suivant :



1. Que représente cette structure : un ARB ou un tas ?
2. Donner cette structure après l'ajout de 40 puis 22.
3. Donner cette structure après la suppression de 19 puis 28.

★★★ Bonne chance ★★★

A.Djeffal

Corrigé type

Exercice 1 : Listes linéaires chainées

1. Structures de données (1 pt)

```
Type  TMaillon = Structure
        Libellé : chaine ;
        Quantité : entier ;
        PrixUnit : réel ;
        Suivant : Pointeur(TMaillon) ;
Fin ;
Var  Stock : Pointeur(TMaillon) ;
```

2. Procédure Vendre(Med, NbBoîtes) (3 pts)

```
Procédure Vendre( Med : chaine, NbBoites : entier);
Var  P,PP : Pointeur(TMaillon) ;
Début
    P ← Stock ;
    PP ← Nil ;
    Tant que ((P ≠ Nil et Libellé(P) ≠ Med) faire
        PP ← P ;
        P ← Suivant(P) ;
    Fin TQ;
    Si (P ≠ Nil) Alors
        Si (Quantité(P)< NbBoites) Alors
            Ecrire('Quantité insuffisantes')
        Sinon
            Aff_Quantité(P,Quantité(P)-NbBoites) ;
            Si (Quantité(P) = 0) Alors
                Si (PP=Nil) Alors
                    Stock ← Suivant(Stock)
                Sinon
                    Aff_Adr(PP,Suivant(P))
                Fin Si;
                Libérer(P) ;
            Fin Si;
        Fin Si;
    Sinon
        Ecrire('Ce médicament n'existe pas') ;
    Fin Si;
Fin;
```

3. Procédure Acheter(Med, NbBoites, PrixUnit) (2.5 pts)

```
Procédure Acheter( Med : chaine, NbBoites : entier, PrixUnit : réel);
Var  P : Pointeur(TMaillon) ;
Début
    P ← Stock ;
    Tant que ((P ≠ Nil et Libellé(P) ≠ Med) faire
        P ← Suivant(P) ;
    Fin TQ;
    Si (P ≠ Nil) Alors
        Aff_Quantité(P,Quantité(P)+NbBoites) ;
        Aff_PrixUnit(P,PrixUnit) ;
    Sinon
        Allouer(P) ;
        Aff_Libellé(P,Med) ;
        Aff_Quantité(P,NbBoites) ;
        Aff_PrixUnit(P,PrixUnit) ;
        Aff_Adr(P,Stock) ;
        Stock ← P ;
    Fin Si;
Fin;
```

4. La fonction PrixStock (1.5 pt)

```
Fonction PrixStock() : réel;
Var  P : Pointeur(TMaillon) ;
     S : réel
Début
    P ← Stock ;
    S ← 0 ;
    Tant que ((P ≠ Nil) faire
        S ← S + Quantité(P) × PrixUnit(P) ;
        P ← Suivant(P) ;
    Fin TQ;
    PrixStock ← S ;
Fin;
```

Exercice 2 : Arbres de recherche binaires

1. Structures de données (1 pt)

```
Type  TNoeud = Structure
        Libellé : chaine ;
        Quantité : entier ;
        PrixUnit : réel ;
        FG,FD : Pointeur(TNoeud) ;
Fin ;
Var  Stock : Pointeur(TNoeud) ;
```

2. Procédure Vendre(Med, NbBoîtes) (3 pts)

```
Procédure Vendre( Med : chaine, NbBoites : entier);
Var  P,PP : Pointeur(TNoeud) ;
Début
    P ← Stock ;
    PP ← Nil ;
    Tant que ((P ≠ Nil et Libellé(P) ≠ Med) faire
        PP ← P ;
        Si (Libellé(P)> Med) Alors
            P ← FG(P)
        Sinon
            P ← FD(P)
        Fin Si;
    Fin TQ;
    Si (P ≠ Nil) Alors
        Si (Quantité(P)< NbBoites) Alors
            Ecrire('Quantité insuffisante')
        Sinon
            Aff_Quantité(P,Quantité(P)-NbBoites) ;
            Si (Quantité(P) = 0) Alors
                Nettoyer(P)
            Fin Si;
        Fin Si;
    Sinon
        Ecrire('Ce médicament n'existe pas') ;
    Fin Si;
Fin;
```

3. Procédure Acheter(Med, NbBoites, PrixUnit) (3 pts)

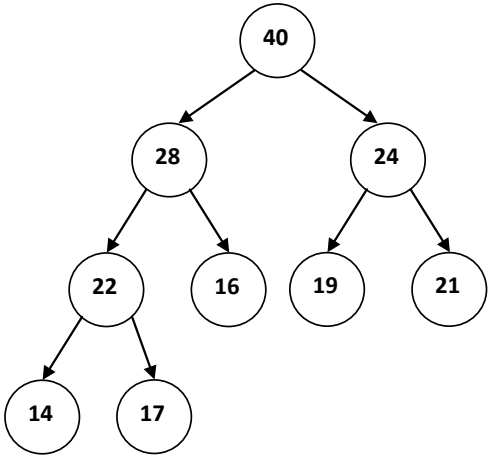
```
Procédure Acheter( Med : chaine, NbBoites : entier, PrixU-
nit : réel);
Var  P,PP : Pointeur(TNoeud) ;
Début
    P ← Stock ; PP ← Nil ;
    Tant que ((P ≠ Nil et Libellé(P) ≠ Med) faire
        PP ← P ;
        Si (Libellé(P)> Med) Alors
            P ← FG(P)
        Sinon
            P ← FD(P)
        Fin Si;
    Fin TQ;
    Si (P ≠ Nil) Alors
        Aff_Quantité(P,Quantité(P)+NbBoites) ;
        Aff_PrixUnit(P,PrixUnit) ;
    Sinon
        Allouer(P) ;
        Aff_Libellé(P,Med) ;
        Aff_Quantité(P,NbBoites) ;
        Aff_PrixUnit(P,PrixUnit) ;
        Aff_FG(P,Nil) ; Aff_FD(P,Nil) ;
        Si (PP=Nil) Alors
            Stock ← P
        Sinon
            Si (Libellé(P)>Med) Alors
                Aff_FG(PP,P)
            Sinon
                Aff_FD(PP,P)
            Fin Si;
        Fin Si;
    Fin Si;
Fin;
```

4. La fonction PrixStock (2 pt)

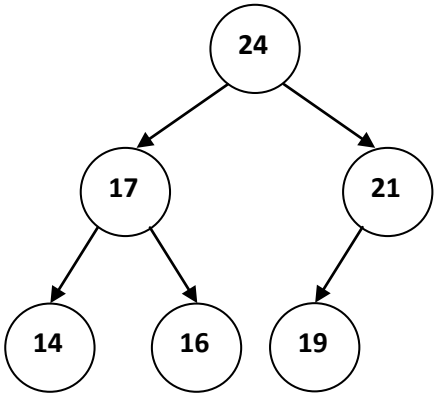
```
Fonction PrixStock( P : Pointeur(TNoeud)) : réel;
Début
    Si (P=Nil) Alors
        PrixStock ← 0
    Sinon
        PrixStock ← Quantité(P) × PrixUnit(P) + Prix-
        Stock(FG(P)) + PrixStock(FD(P)) ;
    Fin Si;
Fin;
```

Exercice 3

1. Cette structure représente un tas. (1 pt)
2. Le tas après l'ajout de 40 puis 22. (1 pt)



3. Impossible de supprimer 19 du tas (0.5 pt). Après le retrait de 28, le tas devient (0.5 pt) :



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

$2^{ème}$ année LMD — ALGO1
13 Mars 2012 — 8:00-9:30, Amphi 1

Examen de rattrapage

Exercice 1 Tas (10 pts : 1.5 + 1.5 + 2.5 + 2.5 + 2)

Soit les valeurs des clés suivantes :

38, 5, 2, 17, 10, 3, 16, 22, 11, 33

On vous demande de :

1. Donner le tas dynamique obtenu après l'insertion de ces clés dans l'ordre.
2. Donner le tas statique correspondant.
3. Donner la déclaration des structures de données permettant de représenter en mémoire le tas statique précédent.
4. Ecrire la procédure **Enfiler(x)** permettant d'ajouter au tas statique la valeur x.
5. Ecrire la fonction **Défiler** permettant de défiler une clé du tas statique.
6. Donner le tas statique de la question 2, après les appels suivants : Enfiler(4) ; Enfiler(14) ; Défiler ; Enfiler(1) ; Défiler ;

Exercice 2 LLCs (10 pts : 1 + 1.5 + 2.5 + 2.5 + 2.5)

On souhaite représenter les polynômes par des listes linéaires chaînées, où chaque terme d'un polynôme est rangé dans un maillon de la liste contenant le degré du terme et le coefficient correspondant.

Exemple : Le polynôme $2x^5+3x^4+\frac{1}{2}x^2+1$ est représentée par la liste suivante :

On vous demande de :

1. Donner les structures de données nécessaires à la représentation des polynômes.
2. Ecrire la fonction **Valeur(P, x)** permettant de donner la valeur du polynôme P pour x.
   Exemple : Valeur(tete, 1) = 6.5
3. Ecrire la procédure **Dériver(P)** permettant de calculer la dérivée du polynôme P et la retourner dans P.
4. Ecrire la procédure **Intégrer(P)** permettant de calculer l'intégrale du polynôme P et le retourner dans P.
5. Ecrire la procédure **Déviation(P, x)** permettant de retourner *Vrai* si x est un point de déviation du polynôme P et *Faux* sinon.

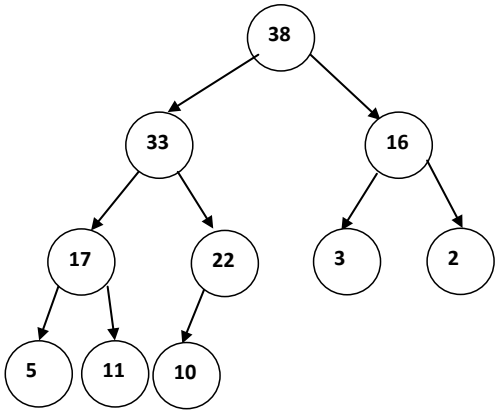
★★★ Bonne chance ★★★

A.Djeffal

Corrigé type

Exercice 1 : Tas

1. Tas dynamique (1.5 pt)



2. Tas statique (1.5 pt)

1	2	3	4	5	6	7	8	9	10
38	33	16	17	22	3	2	5	11	10

3. Structures de données (1 pt)

```
Var  Tas : Tableau[1..NMax] de entier
     N : entier ; // nombre d'éléments dans le tas, initialisé à 0
```

4. Procédure Enfiler(x) (2.5 pts)

```
Procédure Enfiler( x : entier ;);
Var  P,Pere : entier ;
Début
    Si (N=NMax) Alors
        Ecrire('le tas est plein')
    Sinon
        N ← N+1 ;
        Tas[N] ← x ;
        P ← N ;
        Pere ← N div 2 ;
        Tant que ((P > 1) et Tas[Pere] < Tas[P])) faire
            x ← Tas[Pere] ;
            Tas[Pere] ← Tas[P] ;
            Tas[P] ← x ;
            P ← Pere ;
            Pere ← P div 2 ;
        Fin TQ;
    Fin Si;
Fin;
```

5. Fonction Défiler (2.5 pts)

```
Fonction Defiler() : entier;
Var  P,Pere : entier ;
Début
    Si (N=0) Alors
        Ecrire('le tas est vide')
    Sinon
        Defiler ← Tas[1] ;
        Tas[1] ← Tas[N] ;
        N ← N-1 ;
        P ← 1 ;
        Fils1 ← 2 ;
        Fils2 ← 3 ;
        Tant que (((Fils1 ≤ N) et Tas[P] < Tas[Fils1]))
        ou ((Fils2 ≤ N) et Tas[P] < Tas[Fils2])) ) faire
            Si (Fils2 > N) Alors
                PMax ← Fils1
            Sinon
                Si (Tas[Fils1] > Tas[Fils2]) Alors
                    PMax ← Fils1
                Sinon
                    PMax ← Fils2
                Fin Si;
            Fin Si;

            x ← Tas[P] ;
            Tas[P] ← Tas[PMax] ;
            Tas[PMax] ← x ;
            P ← PMax ;
            Fils1 ← P * 2 ;
            Fils2 ← P * 2 + 1 ;
        Fin TQ;
    Fin Si;
Fin;
```

6. Tas statique (1 pt)

1	2	3	4	5	6	7	8	9	10	11
22	17	16	11	10	14	2	5	1	3	4

Exercice 2 : LLCs

1. Structures de données (1 pt)

```
Type  TTerme = Structure
        Coef : réel ;
        Degre : entier ;
        Suivant : Pointeur(TTerme) ;
Fin ;
```

2. Fonction Valeur (1.5 pt)

```
Fonction Valeur( P : Pointeur(TTerme), x : réel) : réel;
Var  P1 : Pointeur(TTerme) ;
       S,C : réel ;
Début
    S ← 0 ;
    P1 ← P ;
    Tant que (P1 ≠ Nil) faire
        C ← 1 ;
        Pour i de 1 à Degre(P1) faire
            C ← C*x ;
        Fin Pour;
        S ← S + C * Coef(P1) ;
        P1 ← Suivant(P1) ;

    Fin TQ;
    Valeur ← S ;

Fin;
```

3. Procédure Dériver (2.5 pt)

```
Procédure Dériver( P : Pointeur(TTerme));
Var  P1,PP1 : Pointeur(TTerme) ;
Début
    PP1 ← Nil ;
    P1 ← P ;
    Tant que (P1 ≠ Nil) faire
        Si (Degré(P1) ≠ 0) Alors
            Aff_ Coef(P1, Coef(P1)*Degré(P1)) ;
            Aff_ Degré(P1,Degré(P1)-1) ;
            PP1 ← P1 ;
            P1 ← Suivant(P1) ;
        Sinon
            Si (PP1 = Nil) Alors
                P ← Nil ;
            Sinon
                Aff_ Adr(PP1,Nil)
            Fin Si;
            Libérer(P1) ;
            P1 ← Nil ;

        Fin Si;
    Fin TQ;
Fin;
```

4. Procédure Intégrer (2.5 pt)

```
Procédure Intégrer( P : Pointeur(TTerme));
Var  P1,PP1 : Pointeur(TTerme) ;
Début
    PP1 ← Nil ;
    P1 ← P ;
    Tant que (P1 ≠ Nil) faire
        Aff_Coef(P1, Coef(P1)/(Degré(P1)+1)) ;
        Aff_Degré(P1,Degré(P1)+1) ;
        PP1 ← P1 ;
        P1 ← Suivant(P1) ;
    Fin TQ;
    Allouer(P1) ;
    Aff_Coef(P1,1) ;
    Aff_Degré(P1,0) ;
    Aff_Adr(P1,Nil) ;
    Si (PP1 = Nil) Alors
        P ← P1 ;
    Sinon
        Aff_Adr(PP1,P1) ;
    Fin Si;
Fin;
```

5. Fonction Déviation (2.5 pt)

```
Fonction Déviation( P : Pointeur(TTerme),  x : réel) :
booléen;
Début
    Si (P=Nil ou Degré(P)< 2) Alors
        Déviation ← Faux
    Sinon
        Dériver(P) ;
        Dériver(P) ;
        Déviation ← (Valeur(P,x) = 0) ;
    Fin Si;
Fin;
```

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

2ème année LMD — ALGO1
27 Jan 2013 — 10:00-11:30, Amphis 1,2,3,4

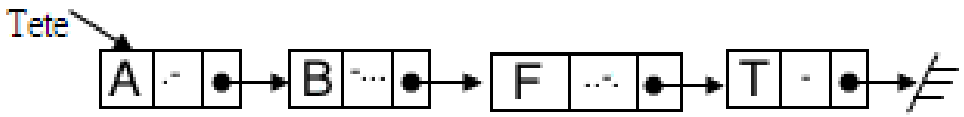
Examen

Exercice 1 LLCs (8.5 pts : 0.5 + 1.5 + 1.5 + 2 + 2 + 1)

Le code morse a été inventé par Samuel Morse, peintre et physicien Américain. C'est un code permettant de transmettre un texte à l'aide de séries d'impulsions courtes et longues. Il a été longtemps utilisé pour effectuer des liaisons longue distance.
En informatique, et plus particulièrement dans les textes, une impulsion longue est représentée par un tiret '-' et une impulsion courte par un point '.'. Chaque caractère de l'alphabet est présené par un code morse.

Lettre	Code	Lettre	Code	Lettre	Code	Lettre	Code
A	.-	H	....	O	—	V	...-
B	-...	I	..	P	.–.	W	.–
C	-.-.	J	.—	Q	–.-	X	-..-
D	-..	K	-.-	R	.-.	Y	-.–
E	.	L	.-..	S	...	Z	–..
F	..-.	M	–	T	-		
G	–.	N	-.	U	..-		

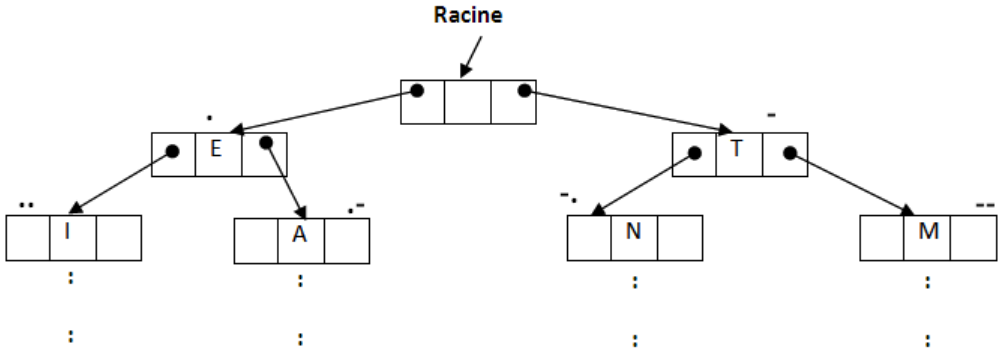
Exemple :



Questions :

1. Donner les structures de données nécessaires à la représentation de la liste des codes morse.
2. Ecrire la procédure AjouterCode(Tete, car, cde) permettant d'ajouter la lettre car avec son code cde à la liste des codes.
3. Ecrire la procédure RetirerCar(Tete, car) permettant de supprimer le caractère car de la liste des codes.
4. Ecrire la fonction Coder(Tete, Ch) permettant de retourner le code morse de la chaine de caractères Ch. Les codes morse de deux caractères sont séparés par un espace, tandis que les codes morses de deux mots sont séparés par deux espaces (blancs). Les codes des caractèrés non trouvés dans la liste sont replacés par des " ?".
5. Ecrire la fonction Decoder(Tete, ChMorse) permettant de retourner la chaine de caractères codée dans la chaine morse ChMorse. Les caractères correspondant aux codes non trouvés dans la liste sont replacés par des " ?".
6. Calculer la complexité des fonctions Coder et Décoder.

Exercice 2 ARB (9.5 pts : 0.5 + 3 + 2 + 3 + 1)
On souhaite représenter la liste des codes morse présentée dans l'exercice précédent par un arbre de recherche binaire, où l'orientation vers un fils gauche signifie un '.' et l'orientation vers un fils droit signifie un '-'.



Questions :

1. Donner les structures de données nécessaires à la représentation de l'arbre des codes morses.
2. Ecrire la procédure AjouterCode(Racine, car, cde) permettant d'ajouter la lettre car avec son code cde à l'arbre des codes.
3. Ecrire la fonction Coder(Racine, Ch) permettant de retourner le code morse de la chaine de caractères Ch. Les codes morse de deux caractères sont séparés par un espace, tandis que les codes morses de deux mots sont séparés par deux espaces (blancs). Les codes des caractèrés non trouvés dans la liste sont replacés par des " ?".
4. Ecrire la fonction Décoder(Racine, ChMorse) permettant de retourner la chaine de caractères codées par la chaine morse ChMorse. Les caractères correspondant aux codes non trouvés dans la liste sont replacés par des " ?".
5. Calculer la complexité des fonctions Coder et Décoder. Que peut-on conclure par rapport à l'exercice précédent ?

Exercice 3 Tas (2 pts : 1.5 + 0.5)

1. Donner le tas statique obtenu après l'insertion dans l'ordre des clés suivantes :

   12, 3, 5, 14, 15, 8, 1, 6

2. Donner le tas après deux retraits

★★★ Bonne chance ★★★

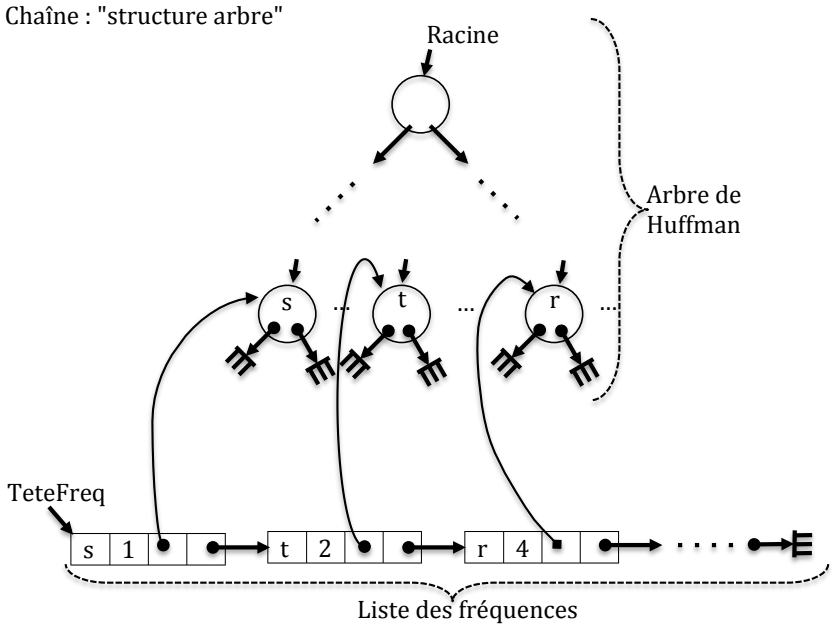
Dr A.Djeffal

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

$2^{ème}$ année LMD — ALGO1
12 Mars 2013 — 16:30-18:00, Amphis 1,2

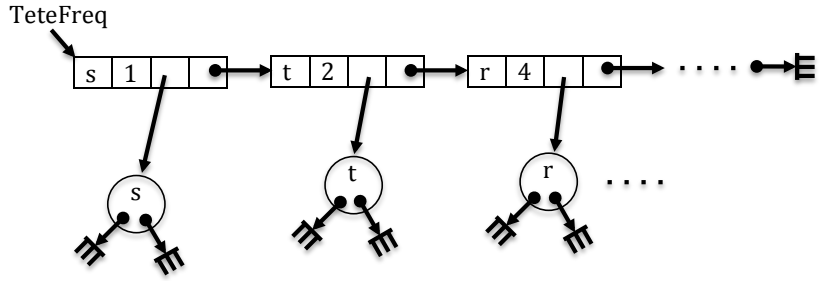
Examen de rattrapage

On souhaite écrire les algorithmes permettant de créer l'arbre de Huffman correspondant à une chaîne de caractères donnée, tel que vu dans le cours. Pour celà, on vous propose de commencer par créer une liste linéaire chainée contenat les caractères de la chaîne à coder avec leurs fréquences d'apparition et des pointeurs vers leurs noeuds correspondants dans l'arbre de Huffman, tel que représenté dans la figure suivante :



Initialement, l'arbre et la liste des fréquneces sont vides. Pour réaliser ce travail, on vous demande de :

1. Donner les structures de données nécessaires à la représentation en mémoire de la liste des fréquences et de l'arbre de Hauffman avec les initialisations nécessaires.
2. Ecrire la procédure **AjouterCar(C :caractère)** permettant d'ajouter le caractère **C** à la liste des fréquences, avec un pointeur nil au nœud de l'arbre. (si **C** existe déjà, on incrémente uniquement sa fréquence)
3. Ecrire la procédure **CreerListeFreq(Ch :chaîne)** permettant de créer la liste de tête TeteFreq, contenant les caractères de la chaîne **Ch** avec leurs fréquences et des pointeurs **nil** vers l'arbre, en utilisant la procédure précédente.
4. Ecrire la procédure **CreerFeuilles** permettant de créer, pour chaque caractère de la liste des fréquences, un nœud feuille de l'arbre contenant ce caractère, tel que présenté dans la figure suivante :

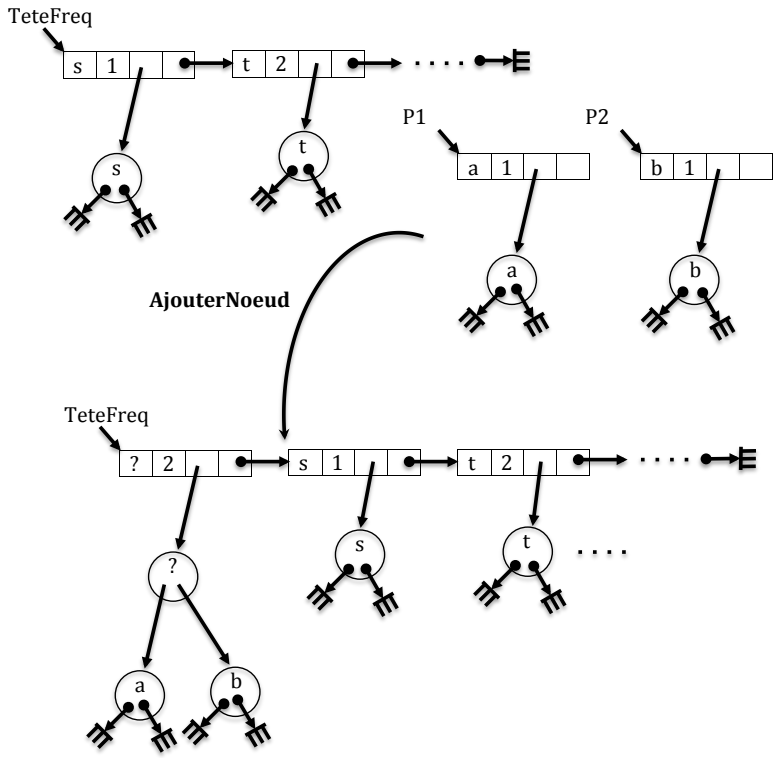


5. Ecrire la fonction **GetMinFreq** permettant de retourner le pointeurs vers le caractère de fréquence minimale dans la liste des fréquences tout en le supprimant de la liste **sans le libérer**.
6. Ecrire la procédure **AjouterNoeud(P1,P2 :Pointeur(TMaillon))** permettant de :

(a) Créer un nouveau nœud de l'arbre ayant deux fils, les nœuds pointés par P1 et P2,

(b) Ajouter un nouveau maillon à la liste des fréquences avec une fréquence égale à la somme des fréquences de P1 et P2, et avac un pointeur vers le nouveau nœud crée de l'arbre.

Exemple :



7. Ecrire la procédure **CreerArbreHuffman(Ch :chaîne)** permettant de créer l'arbre de Huffman correspondant à la chaîne de caractères **Ch** en utilisant les procédures et la fonction précédentes.

Barème : $2 + 3 + 3 + 3 + 3 + 3 + 3$

★★★ Bonne chance ★★★

A.Djeffal

Corrigé type

Exercice 1 :

1. Structures de données (2 pts)

```
Type  TNoeud = Structure
        Car : caractère ;
        FG,FG : Pointeur(TNoeud) ;
Fin ;

Type  TMaillon = Structure
        Car : caractère ;
        Freq : entier ;
        Noeud : Pointeur(TNoeud) ;
        Suivant : Pointeur(TMaillon) ;
Fin ;

Var  Racine : Pointeur(TNoeud) ; Initialisée Nil
     TeteFreq : Pointeur(TMaillon) ; Initialisée Nil
```

2. Procédure AjouterCar (3 pts)

```
Procédure AjouterCar( C : caractère);
Var  P : Pointeur(TMaillon) ;
Début
    P ← TeteFreq ;
    Tant que ((P ≠ Nil) et (Car(P) ≠ C)) faire
        P ← Suivant(P) ;
    Fin TQ;
    Si (P ≠ Nil) Alors
        Aff_Freq(P, Freq(P)+1)
    Sinon
        Allouer(P) ;
        Aff_Car(P, C) ;
        Aff_Freq(P, 1) ;
        Aff_Noeud(P, Nil) ;
        Aff_Adr(P, TeteFreq) ;
        TeteFreq ← P ;
    Fin Si;
Fin;
```

3. Procédure CreerListeFreq (3 pts)

```
Procédure CreerListeFreq( Ch : chaîne);
Var  P : Pointeur(TMaillon) ;
Début
    TeteFreq ← Nil ;
    Pour i de 1 à Longueur(Ch) faire
        AjouterCar(Ch[i])
    Fin Pour;
Fin;
```

4. Procédure CeerFeuilles (3 pts)

```
Procédure CeerFeuilles();
Var  P : Pointeur(TMaillon) ;
       N : Pointeur(TNoeud) ;
Début
    P ← TeteFreq ;
    Tant que ((P ≠ Nil)) faire
        Allouer(N) ;
        Aff_Car(N, Car(P)) ;
        Aff_FG(N, Nil) ;
        Aff_FD(N, Nil) ;
        Aff_Noeud(P, N) ;
    Fin TQ;
Fin;
```

5. Fonction GetMinFreq (3 pts)

```
Fonction GetMinFreq() : Pointeur(TMaillon);
Var  P, Prec, PrecMin, PMin : Pointeur(TMaillon) ;
Début
    P ← TeteFreq ;
    PMin ← P ;
    PrecMin ← Nil ;
    Prec ← Nil ;
    Tant que ((P ≠ Nil)) faire
        Si (Freq(P) < Freq(PMin)) Alors
            PMin ← P ;
            PrecMin ← Prec ;
        Fin Si;
        Prec ← P ;
        P ← Suivant(P) ;
    Fin TQ;
    Si (PrecMin ≠ Nil) Alors
        Aff_Adr(PrecMin, Suivant(P)) ;
    Sinon
        Si (TeteFreq≠ Nil) ) Alors
            TeteFreq ← Suivant(TeteFreq) ;
        Fin Si;
    Fin Si;
    GetMinFreq ← PMin ;
Fin;
```

6. Procédure AjouterNoeud (3 pts)

```
Procédure AjouterNoeud( P1, P2 : Pointeur(TMaillon));
Var  P : Pointeur(TMaillon) ;
     N : Pointeur(TNoeud) ;
Début
    Allouer(N) ;
    Aff_Car(N, '?') ;
    Aff_FG(N, Noeud(P1)) ;
    Aff_FD(N, Noeud(P2)) ;
    Allouer(P) ;
    Aff_Car(P, '?') ;
    Aff_Freq(P, Freq(P1) + Freq(P2)) ;
    Aff_Noeud(P, N) ;
    Aff_Adr(P, TeteFreq) ;
    TeteFreq ← P ;
Fin;
```

7. Procédure CreerArbreHuffman (3 pts)

```
Procédure CreerArbreHuffman( Ch : chaîne);
Var  P1, P2 : Pointeur(TMaillon) ;
Début
    CreerListeFreq(Ch) ;
    CreerFeuilles ;
    Tant que ((TeteFreq ≠ Nil) et (Suivant(TeteFreq) ≠
    Nil)) faire
        P1 ← GetMinFreq ;
        P2 ← GetMinFreq ;
        AjouterNoeud(P1, P2) ;
        Libérer(P1) ;
        Libérer(P2) ;
    Fin TQ;
    Si (TeteFreq ≠ Nil) Alors
        Racine ← Noeud(TeteFreq) ;
        Libérer(TeteFreq) ;
    Sinon
        Racine ← Nil ;
    Fin Si;
Fin;
```

Références

[1] D. Beauquier, J. Berstel, P. Chrétienne, et al. *Éléments d'algorithmique*, volume 8. Masson, 1992.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, T.H. Cormen, and T.H. Cormen. *Introduction à l'algorithmique*. Dunod, 1996.

[3] J. Courtin and I. Kowarski. *Initiation à l'algorithmique et aux structures de données*. Dunod, 1990.

[4] M.C. Gaudel, M. Soria, and C. Froidevaux. Types de données et algorithmes 2 : Recherche, tri, algorithmes sur les graphes. 1987.

[5] Guillaume Poupard. *Algorithmique*. Ecole Nationale Supérieure des Techniques Avancées, 2000.