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

Plan du cours

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.

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 : entier);
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 lLeftarrow '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 du 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 $O(\sum_{i=1}^{n} n-i) = O(n-i)]$
  Pour j **de** i+1 **à** n **faire**
    [n − i fois $O(\sum_{1}^{n-i} 1) = O(n-i)]$
    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 $Fac(n-1)$ 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.

Chapitre 3

Structures séquentielles

3.1 Les listes linéaires chaînées (LLC)

3.1.1 Notion d'allocation dynamique

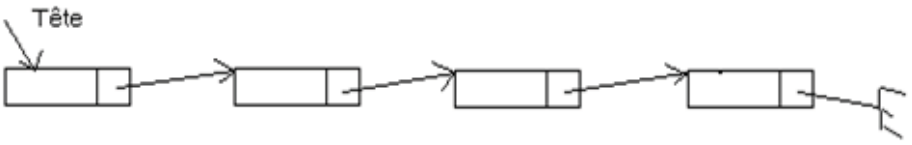
L'utilisation des tableaux statiques implique que l'allocation de l'espace se fait tout à fait au début d'un traitement, c'est à dire que l'espace est connu à la compilation. Pour certains problèmes, on ne connaît pas à l'avance l'espace utilisé par le programme. On est donc contraint à utiliser une 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 :

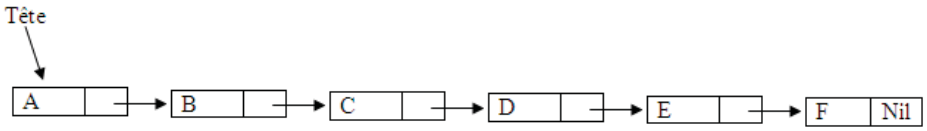


Un élément ou maillon d'une LLC est toujours une structure (Objet composé) avec deux champs : un champ *Valeur* contenant l'information et un champ *Adresse* donnant l'adresse du prochain élément. A chaque maillon est associée une adresse. On introduit ainsi une nouvelle classe d'objet : le type *Pointeur*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

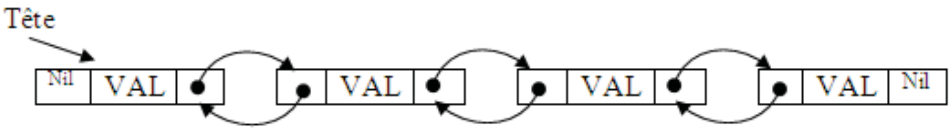
Recherche d'un élément dans une liste

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 ;**

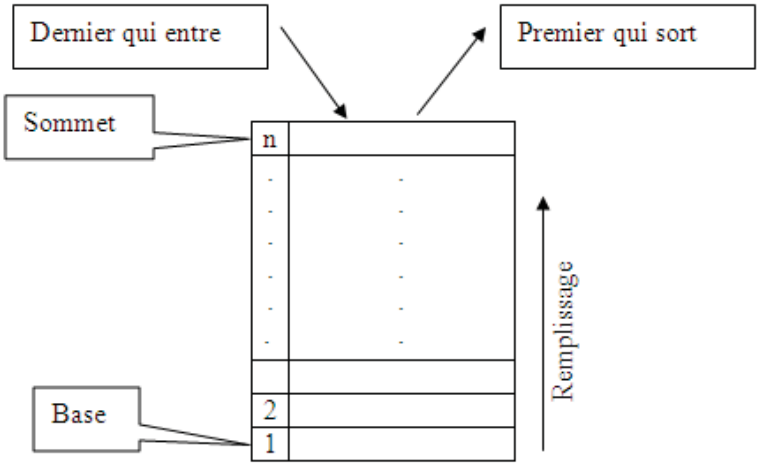
Modèle sur les listes linéaires chaînées bidirectionnelles

- **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

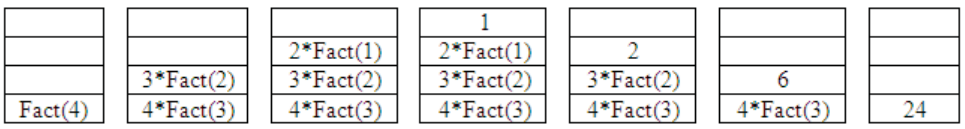
Une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton "Afficher la page précédente".

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.

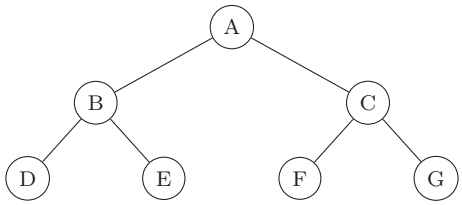
3.2.2.3 Gestion des appels dans l'exécution des programmes

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 $((a + (b * c))/(cd)$ 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

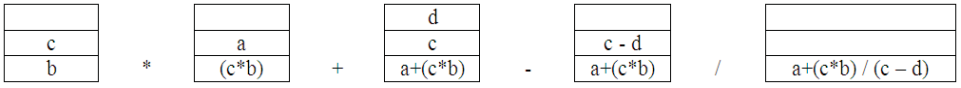
    Mettre l'opérande dans la pile ;

  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.

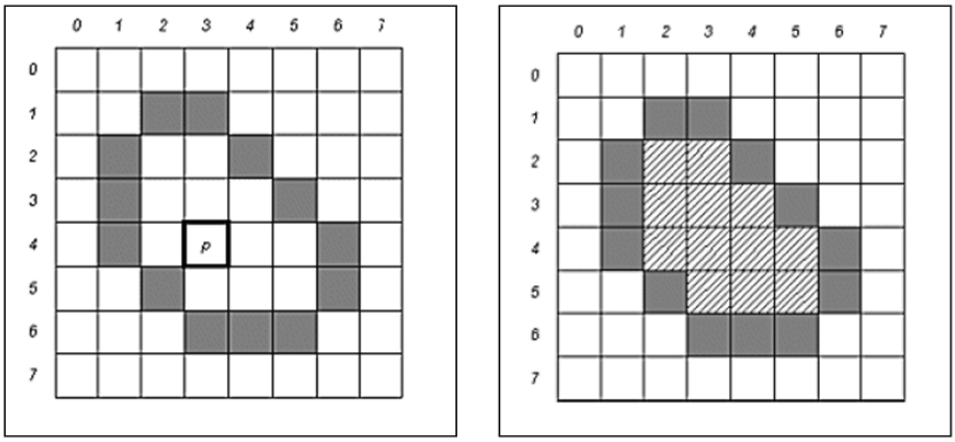
Les deux algorithmes "PileParTableaux" et "PileParLLCs" suivant présentent deux exemples d'implémentation statique et dynamique des piles.

```
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 $p = (3, 4)$.



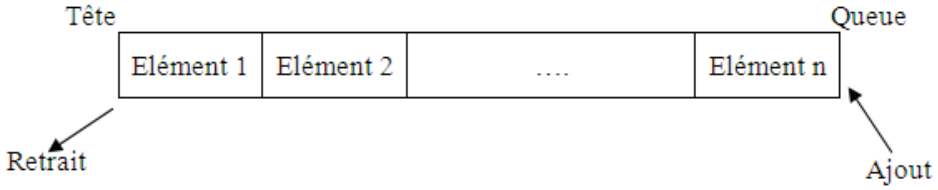
Pour effectuer le remplissage, on doit aller dans toutes les directions à partir du point p. Ceci ressemble au parcours d'un arbre avec les noeuds de quatre fils. La procédure suivante permet de résoudre le problème en utilisant une pile.

```
Procédure Remplir( Image : Tableaux[0..M−1, 0..N−1] de couleur ; x, y : entier ;
c : couleur);
Var  c_1 : couleur ;
Début
    c_1 ← Image[x, y] ;
    InitPile ;
    Empiler((x, y)) ;
    Tant que (¬ PileVide) faire
        Depiler((x,y)) ;
        Si (Image[x, y] = c_1) 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

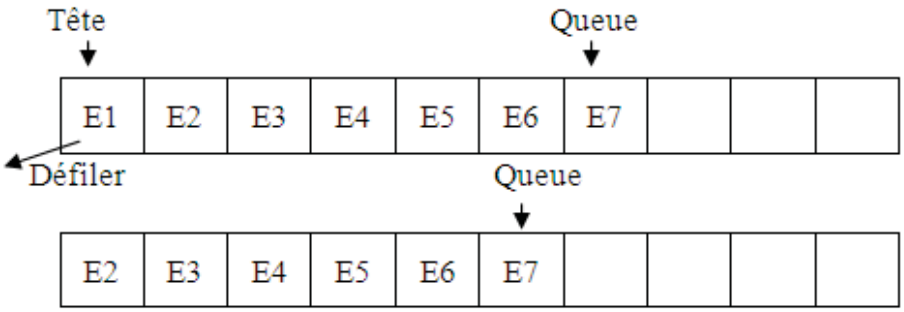
De même que pour les piles, les files d'attente peuvent être représentées en deux manières :

- 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

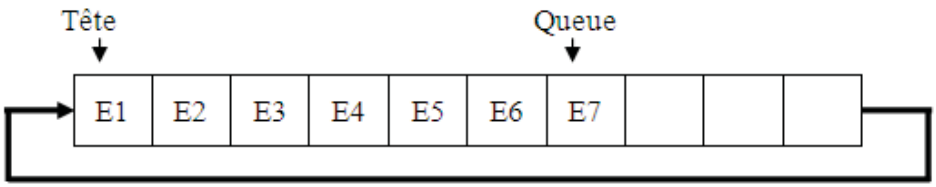


→ La file est vide si $Queue = 0$

→ 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 $Tête = Queue$

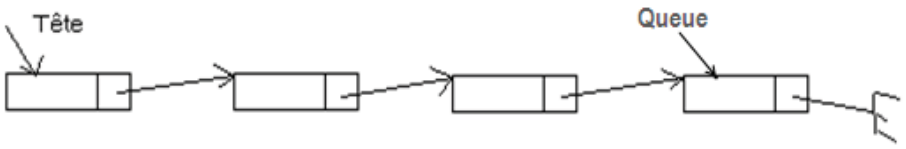
→ La file est pleine si $(Queue + 1)\ mod\ n = Tête$

↓ 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 ← 1 ; Queue ← 1 ;
Fin;
Fonction FileVide() : Booleen;
Début
    | FileVide ← (Tête = Queue) ;
Fin;
Fonction FilePleine() : Booleen;
Début
    | FilePleine ← (((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] ← x ;
        | Queue ← (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 ← File[Tete] ;
        | Tête ← (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.
```

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

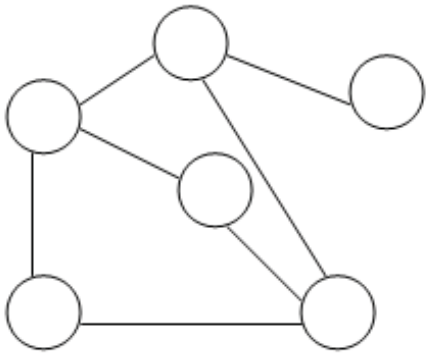
En plus plusieurs traitements en informatique sont de nature arborescente tel que les arbres généalogiques, hiérarchie des fonctions dans une entreprise, représentation des expressions arithmétiques,.. etc.

4.1.2 Définitions

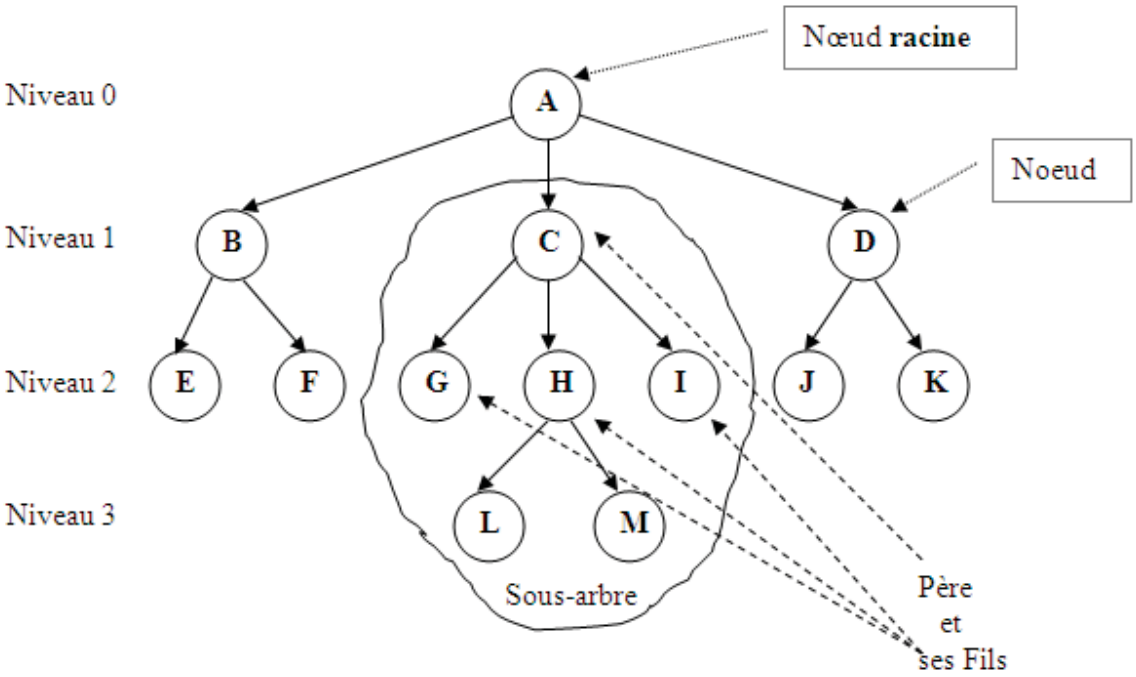
4.1.2.1 Définition d'un arbre

Un arbre est une structure non linéaire, c'est un graphe sans cycle où chaque noeud 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 noeud qui n'a pas de prédécesseur s'appelle racine (A)
- Le noeud 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 noeuds 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 noeud

- Le niveau de la racine = 0
- Le niveau de chaque noeud 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 noeud

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

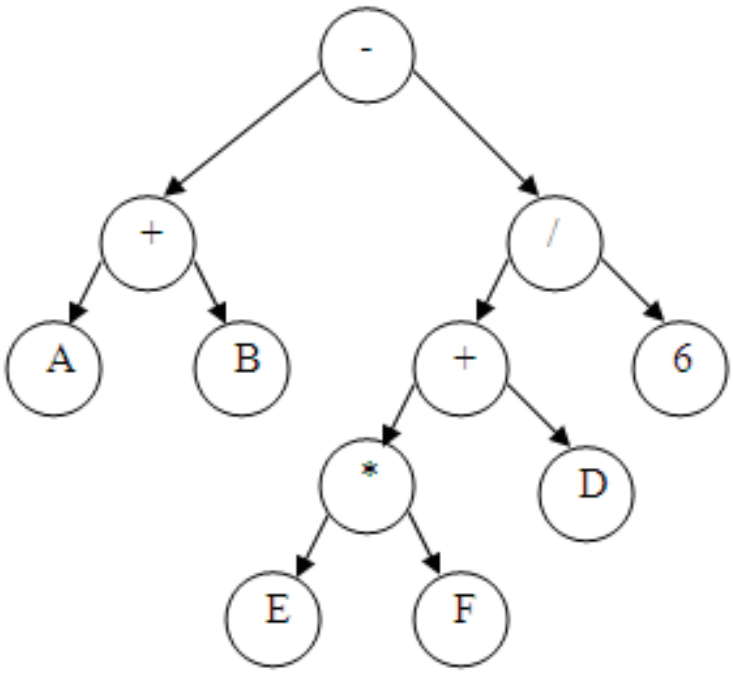
4.1.2.6 Degré d'un arbre

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

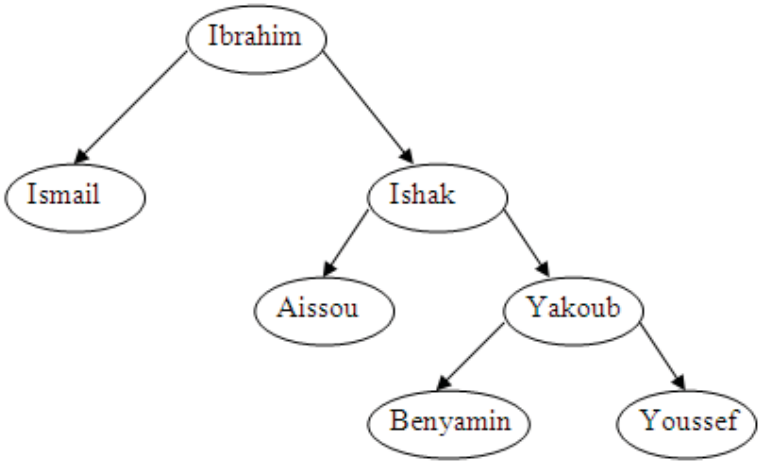
4.1.3 Utilisation des arbres

- Représentation des expressions arithmétiques

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



– Représentation d'un arbre généalogique



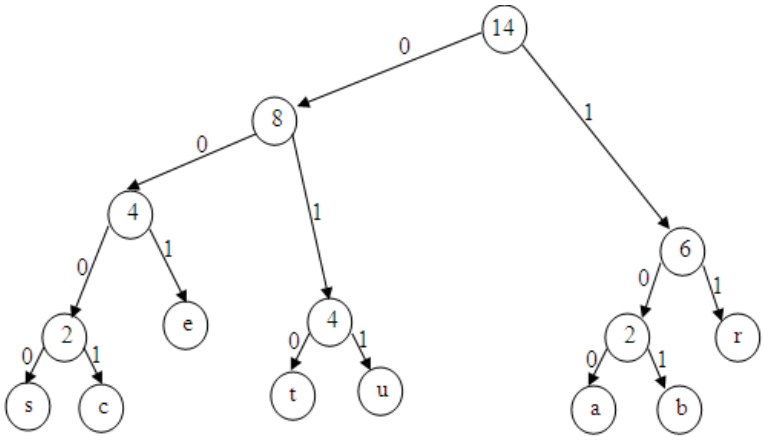
– Codage

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

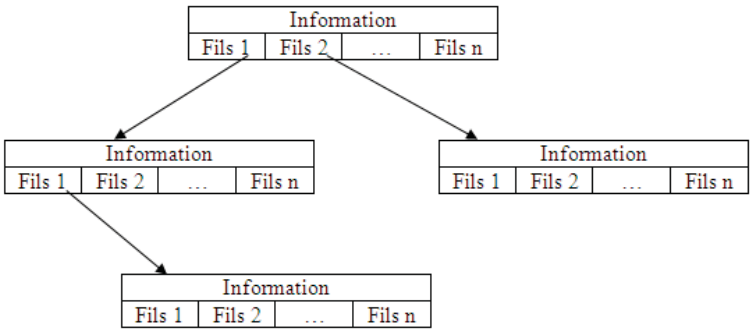
Les arbres peuvent être représentés par des tableaux, des listes non linéaires ou tous les deux :

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

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 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 noeud 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 noeud 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 noeuds. 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( Noeud : Pointeur(TNoeud));
Début
    Si (Noeud ≠ Nil) Alors
        Pour i de 1 à NbFils faire
            PP(Fils_i(Noeud))
        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( Noeud : Pointeur(TNoeud));
Début
    Si (Noeud ≠ Nil) Alors
        Afficher(Valeur(Noeud)) ;
        Pour i de 1 à NbFils faire
            PPPrefixe(Fils_i(Noeud))
        Fin Pour;
    Fin Si;
Fin;
```

```
Procédure PPPostfixe( Noeud : Pointeur(TNoeud));
Début
    Si (Noeud ≠ Nil) Alors
        Pour i de 1 à NbFils faire
            PPPostfixe(Fils_i(Noeud))
        Fin Pour;
        Afficher(Valeur(Noeud)) ;
    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 noeuds à une profondeur i doivent avoir été visités avant que le premier noeud 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( Noeud : Pointeur(TNoeud));
Var  N : Pointeur(TNoeud) ;
Début
    Si (Noeud ≠ 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( Noeud : Pointeur(TNoeud) ; Val : Typeqq) : Booleen;
Var  i : entier ;
     Trouv : Booleen ;
Début
    Si (Noeud = Nil) Alors
        Rechercher ← Faux ;
    Sinon
        Si (Valeur(Noeud)=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( Noeud : Pointeur(TNoeud)) : entier;
Var  i,S : entier ;
Début
    Si (Noeud = Nil) Alors
        Taille ← 0 ;
    Sinon
        S ← 1 ;
        Pour i de 1 à NbFils faire
            S ← S + Taille(Fils_i(Noeud)) ;
        Fin Pour;
        Taille ← S ;
    Fin Si;
Fin;
```

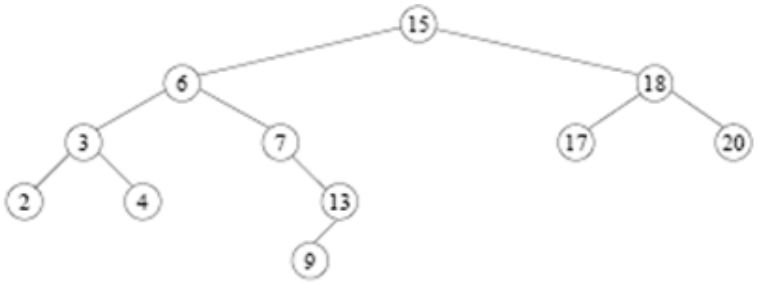
4.1.7 Typologie des arbres

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

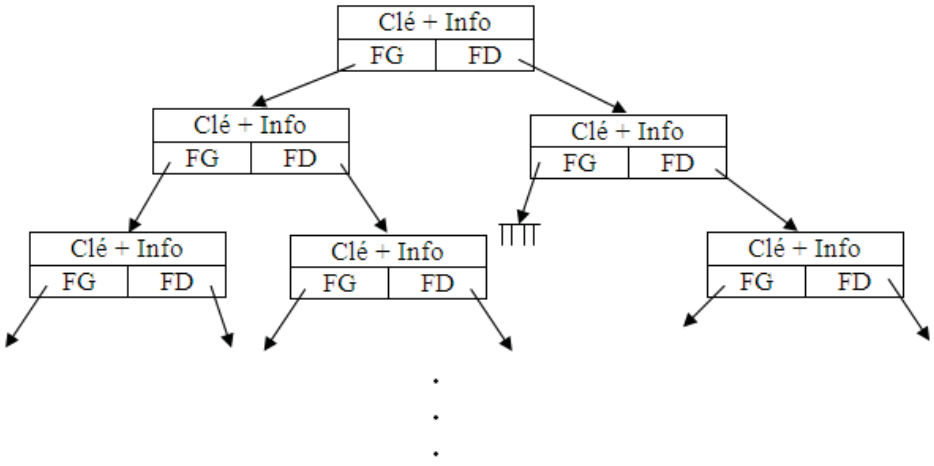
4.2.2 Implémentation des ABR

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

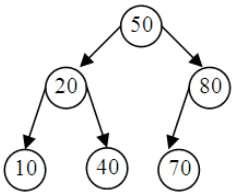
Plusieurs cas de figure peuvent être trouvés : soit à supprimer le nœud N

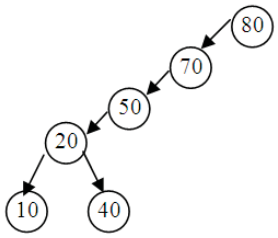
Cas			Action
FG(N)	**FD(N)**	**Exemple**	
Nil	Nil	Feuille (2,4,17)	Remplacer N par Nil
Nil	$\neq$ Nil	7	Remplacer N par FD(N)
$\neq$ Nil	Nil	13	Remplacer N par FG(N)
$\neq$ Nil	$\neq$ Nil	6	1- Rechercher le plus petit descendant à droite de N soit P (7) 2- Remplacer Valeur(N) par Valeur(P) (6 $\leftarrow$7)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

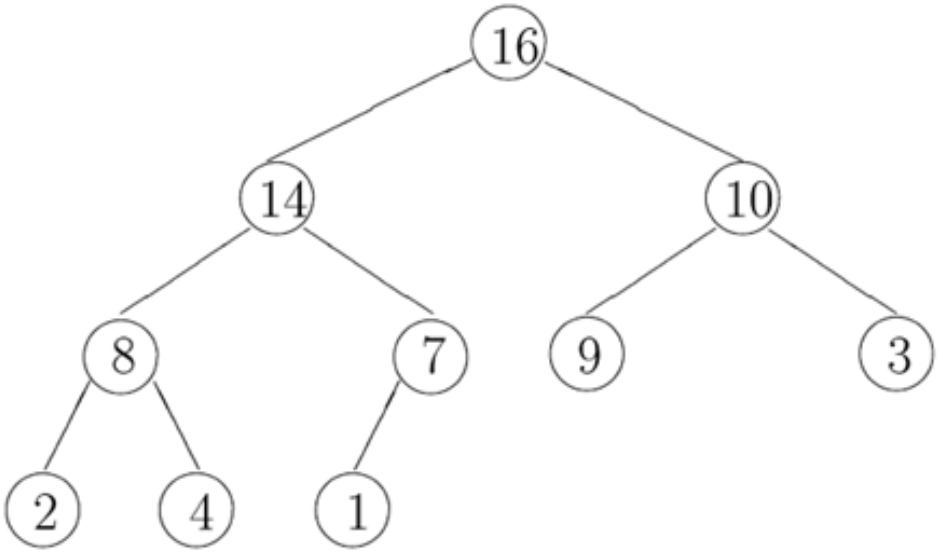
4.3.2 Définition

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

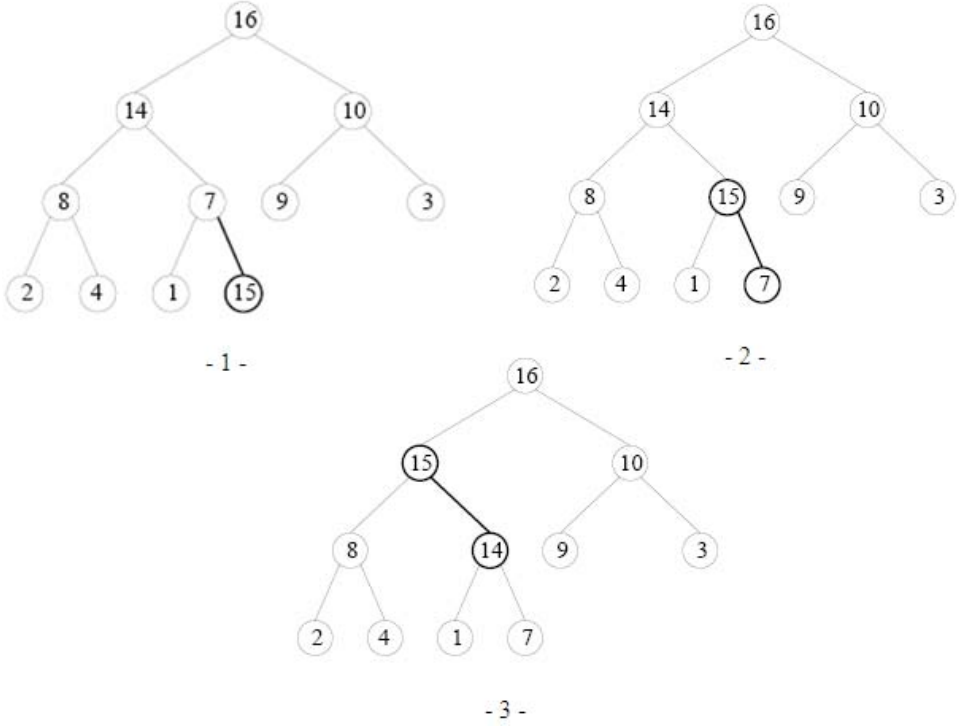
Pour coder une file d'attente avec priorité par un tas, on doit définir les opérations d'ajout et de retrait.

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 :



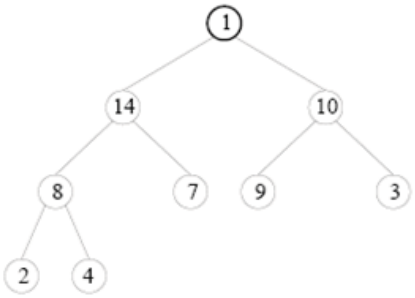
La complexité de cette opération est de $O(h = Log_2(n))$ si n est le nombre d'éléments.

4.3.3.2 Retrait

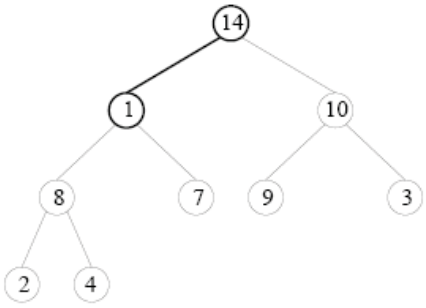
L'élément le plus prioritaire se trouve toujours à la racine, donc le retrait consiste à lire la racine puis la supprimer. Pour ce faire on doit :

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.

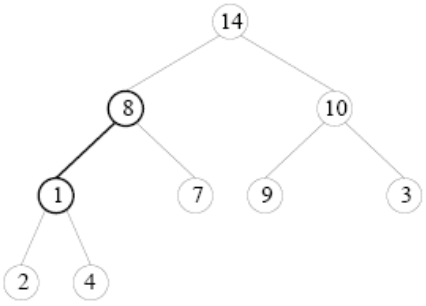
Exemple :



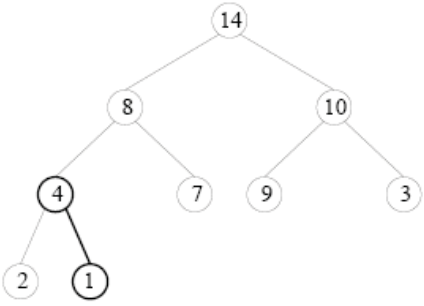
- 1 -



- 2 -



- 3 -

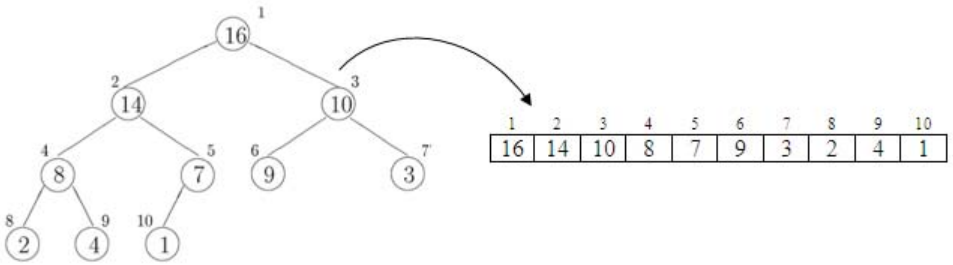


- 4 -

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.

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 le trouver c'est-à-dire passer par tous les éléments placés avant. La recherche est alors en moyenne de $n/2$ ($O(n)$).

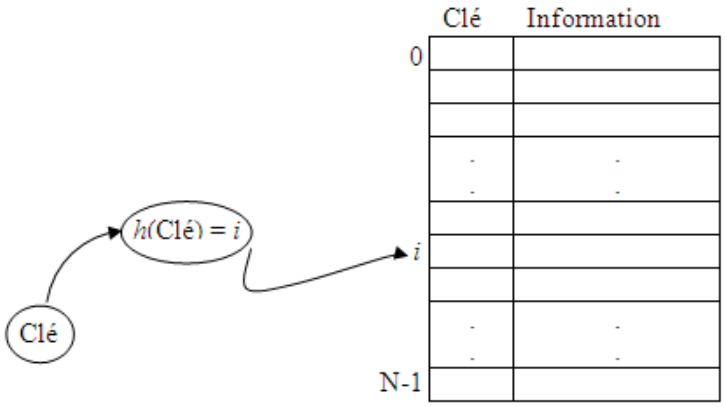
5.3 Table triée

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

5.4 Hachage (HashCoding)

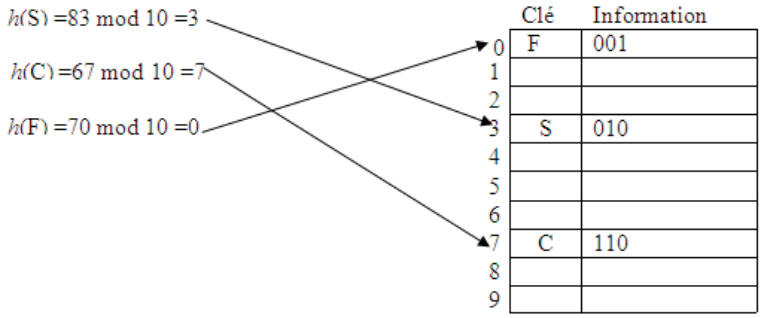
5.4.1 Principe

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
   - $(\text{Clé})^2 = (453)^2 = 20\mathbf{52}09$
   - 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 : $0 \leq h(clé) \leq N$.

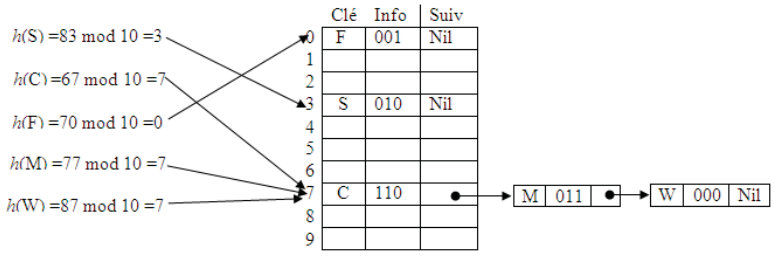
En plus, cette fonction doit être la plus distribuée possible sur cette intervalle, pour que les informations ne se concentrent pas dans une partie de la table.

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.

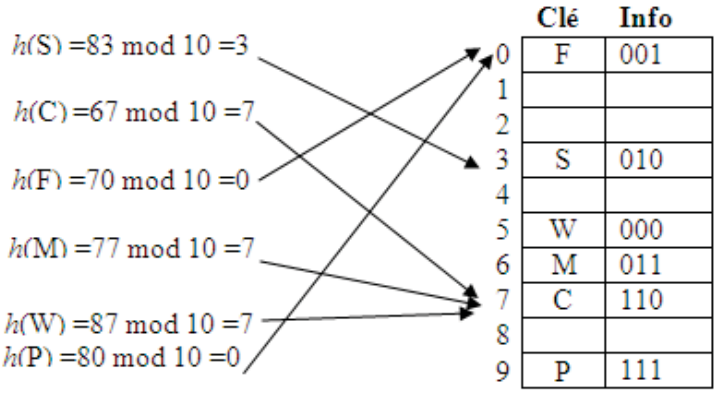
5.4.3.1 Les listes linéaires chaînées

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 :

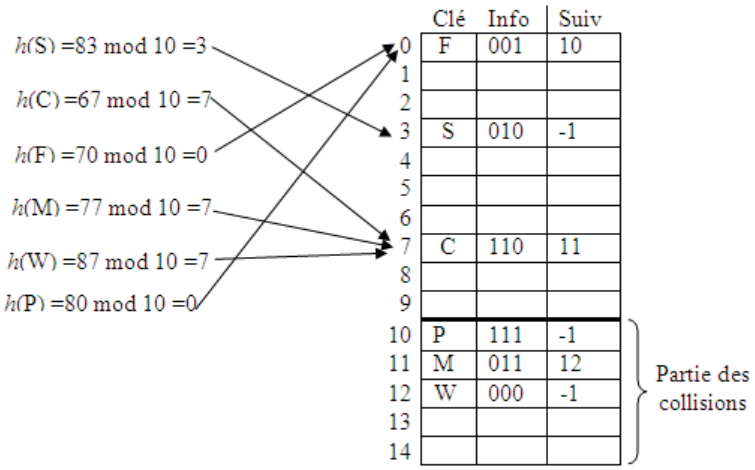


On doit garder, dans ce cas, une case toujours vide pour indiquer la fin de la recherche, c-à-d une valeur interdite dans les clés.

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 :



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.

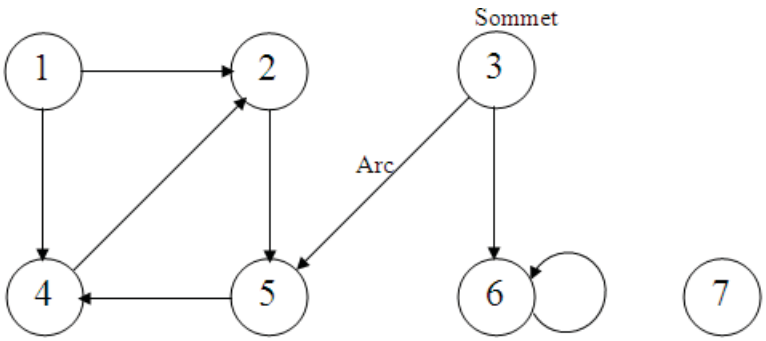
6.2 Définitions

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

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

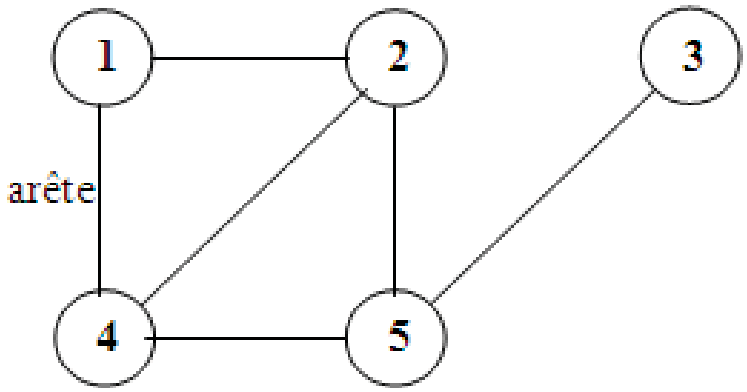


6.2.2 Graphe orienté

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".

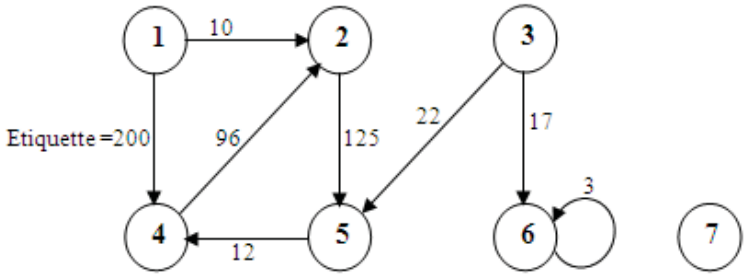
6.2.3 Graphe non orienté

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.

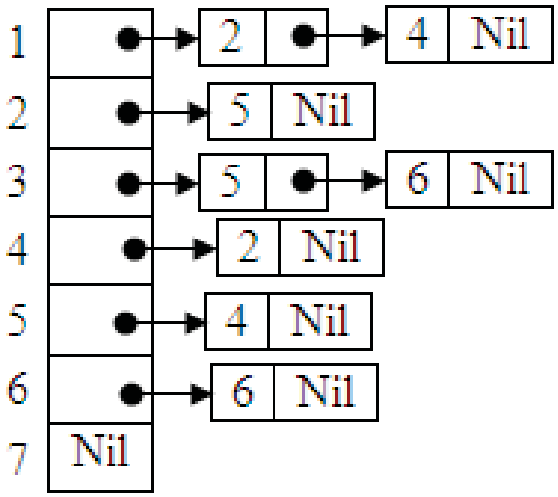
Exemple : Pour $(3, 2)$ il existe un chemin $\{(3, 5), (5, 4), (4, 2)\}$ mais il n'en existe pas pour $(2, 3)$.

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 $(S \quad D)$ 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

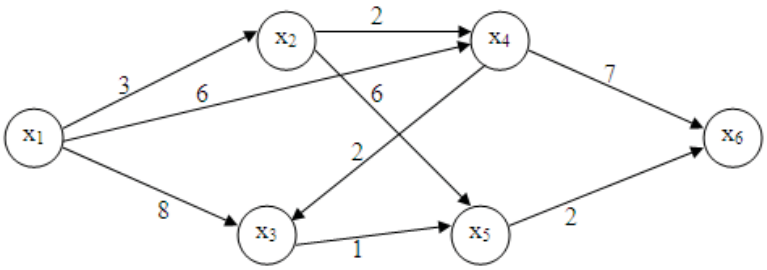
2- $\lambda_k = min\ \lambda_j \quad / \ x_j \notin D$

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

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 ← y
Sinon
    | r ← 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 ← y
Sinon
    | Tant que (x ≠ y)faire
    |     | r ← 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 :

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

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.