III. Récursivité :

Le sous-programme est dit recursive lorsqu'il contient une ou plusieurs instructions d'appel du sous-programme lui même.

Exemple :

Factorielle d'un entier

$$fact(n) = \begin{cases} 1 & \text{si } n=0 \\ n * fact(n-1) & \text{sinon} \end{cases}$$

```
Function fact (n: integer): integer
Begin
  if n=0 then fact(n) = 1
  else fact (n) = n * fact (n-1);
End;
```

```
if <condition> then <appel>
else <instruction d'appel recursif>;
```

ou

```
while <condition> do <appel recursif>;
```

Remarque :

Soient les 2 programmes suivants :

```
program pr1;
var k, n : integer;
Function fact (n: integer) ;integer;
Begin
  if (n=0) then fact(n):= 1
  else fact(n):= n * fact(n-1);
End;
Begin
  read (a n);
```

```
program pr2;
Var i, n, fact : integer;
Begin
  read (a);
  fact := 1;
  for i=1 to n do
    fact := fact * i;
End;
```

```
K := fact (n);
write (K);
End;
```

- Le programme pr2 est mailleur que le program pr1 car dans pr2 La taille mémoire nécessaire est réduite (gain mémoire) Alors que dans pr1 (est) elle progrésse proportionnellement à La variable n
- Gain en tant d'escécution, l'activation et la fin d'activation d'une fonction nécessite un certain travail (allocation de place sur La pile ; mis à jours des pointeurs) se qui augmente bcp le temps d'escecution d'un programme
- Bien que trés esthètique l'utilisation de la récursivité peut être trés couteuse

Exemple : Calculer La puissance x^n ;

```
Function puis (X, n : integer) : real ; Var p : real;
                                            i : integer;
Begin
  P := 1 ;
  if n = 0 then p := 1
  else for i := 1 to n do
    P := P * X ;
  Puis := P;
End;
```

```
Function puis (n, x: integer): real;
  Begin
  if n=0 then puis := 1
  else puis := X * puis(x,n-1);
  End;
```

Structure de données :

. Introduction aux types abstraits :

La conception d'un algorithme se fait en plusieurs étapes qui correspondent à des raffinements successifs. La 1ère version de l'algorithme est autant que possible indépendant d'une implimentation particulière ; à ce niveau les données sont considérées de manière abstraite, on se donne une notation pour les décrire ainsi que l'ensemble des opération qu'on peut leur apliquer et les propriété de ses opération, on parle alors de type abstrait de données.

La conception d'un algorithme se fait en utilisant les opérations de type abstrait.

Remarque

Les différentes représentations permettent d'obtenir des différents versions de l'algorithme.

Exemple : . La conception de l'algorithme en utilisant le concept de type abstrait de données TAD applique une démarche décendente : On se donne la définition des types de données et on conçoi l'algorithme à ce niveau ; on donne ensuite une représentation concrete (réelle) des types et des opérations qui peuvent être abstraites jusqu'à obtenir un programme exécutable

- Les avantages :
- La conception est plus simple (pas de details de programmation.
- Elle est définitive : quelque soit la représentation (abs) de type abstrait.

II_ Signature :

Elle décrit la syntaxe de type outil (nom d'opérations et type des arguments)

La signature d'un type comporte :

. le nom pour (s) certains nombre d'ensemble de valeurs ; ce nom est appelé Sorte (la définition d'un type peut faire intervenir plusieurs sortes).

. le nom d'un certain nombres d'opérations (es) et de leurs profils ; le profil précise à quel ensemble de valeur appartiennent les arguments et le résultat de l'opération

\# Exemple de singnature :

sorte, element, vecteur, integer

opération :

i-éme : vecteur, x : entier → element.

changer-iéme : vecteur x entier x element → vecteur

Borne sup : vecteur → entier

Borne inf : vecteur → entier

Remarque

Le nom de l'opération est utilisé comme le nom d'une fonction ; Ex : Si f a le profil suivant :

$$f : T_1 \times T_2 \rightarrow T_3$$

et si $a_1 \in T_1$ et $a_2 \in T_2$

Alors : $f(a_1 + a_2) \in T_3$

Remarque :

- On précise la place des arguments pour ou des résultats par un tire ou une flèche "→" ou "-".
- On peut utiliser les parenthèses "( )"
- Une opération qui n'a pas d'arguments est une constante ; Ex : 0 : → Entier ;

Vrai : → Booleen ;

III - Réutilisation et hiérarchie dans les TAD :

Quand on défini un type abstrait des données il est possible d'utiliser des types déja défini

Ex : sorte vecteur

utilise entier ; élement

Remarque :

- La signature d'un type est l'union disjointe des signature des types utilisés
- Il y'a une hiérarchie entre les types

Chapitre 5 - Structure séquentielles :

I_ Les listes :

Une liste linéaire est une suite finie éventuellement vide d'éléments repérés par leur rang (place) dans la liste $A: < e_1, e_2, e_3, \ldots e_n >$.

Soit E l'ensemble des éléments et L l'ensemble des listes on peut écrire $L = \phi + E \underline{X} L$

l'ordre des elements dans une liste est fondamental ce n'est pas un ordre sur les contenus des elements mais un ordre (des) sur les places des eléments; les palaces sont ordonnés, c'est-à-dire il existe une fonction "succ" telque : a chaque place p d'une liste non viole on a:

$\exists K \geq 0$ tp: $p = succ^{K}(tête(d))$;

où :

tête (d) : indique la 1ère place de la liste d.

. Chapue place a un contenu de type élément.

. Le nombre de place de la liste d est appelé longueur de d (si n=0 Alors la liste est vide).

. la fonction succ n'est pas défini pour la dernière place de d.

. parmis les traitement de manipulation de liste, les plus frequent consiste a examinie séquentiellement dans l'ordre des place toutes les places d'une listes non vide pour appliqué le même traitement à leur contenues.

x := tête (d);

traiter (contenu (x))

I.2 : Type abstrait liste-recursive :

Les opérations de bases ne sont plus l'accés et l'insértion à la place numéro K mais plutôt l'opération tête qui range la première place d'une liste et l'opération fin.

Sorte liste, place
Utilise entier, élement
Opérations :
liste vide : → liste
tête : liste → place
fin : liste → liste
cons : élement x liste → liste
premier : liste → élement
contenu : place → élement
Succ : place → place

Remarque :

D'autres operations peuvent être rajouter telles que la concaténation de deux liste
Concatener : liste x liste → liste

II. Représentation de liste :

II.1 : " contigue :

On utilise un tableau dont la $i^{ème}$ case représente la $i^{ème}$ place de la liste

```
Const Lmax = ---;
type liste = record
        T : Array [1 ... Lmax] of élement;
        longueur : 0 ... Lmax;
    End;
```

La procedure supprimer (Var L : liste; K : 1 .. lmax);

```
Var   n = 0 .. lmax;
      i = 1 .. lmax;
  Begin
  n := L.longueur;
  if K <= n then  Begin
                  for i := k to n-1 do
                  L.T[i] := L.T[i+1];
                   L.longueur := n-1;
                  End;
  End;
```

La procedure Insérer (Var L : liste; K : 1 .. lmax; x : élément);

```
Var   n, i = 0 .. lmax;
  Begin
  n := L.longueur
  for i := n down to k do
     L.T[i+1] := L.T[i];
     L.T[k] := x;
     L.longueur := n+1;
  End;
```

Remarques :

. La longueur de la liste ne peut dépasser la taille de tableau

. Une supression demande au pire une n-1 affectation et une instruction où n est le nombre de la liste.

. Pour une insersion demande au pire n affectation + 2 instruction : une pour l'insertion de l'element et la 2ème

```
. Procedure Empiler (var P: Pile, e: element);
  Begin
  P.sommet := P.sommet + 1;
  P.Tab[P.sommet] := e;
  End;

. Procedure Dépiler (var p: pile);
  Begin
  P.sommet := P.sommet - 1;
  End;

. Function sommet (P: Pile): element;
  Begin
  sommet := P.Tab[P.sommet];
  End;

. Function test-vide (P: Pile): booleen;
  Begin
  test-vide := P.sommet = 0
  End;
```

2.2: les représentation chaînée:

```
type  Pile = ^element
      element = record
          Val: element;
          suiv: pile;
      End;
```

• procedure Empiler (var P: Pile; e: elemen);

```
Var q: pile;
Begin
New(q); q^.val := e;
q^.suiv := P;
P := q;
End;
```

• procedure Dépiler (var P: Pile);

```
Var q: pile;
Begin
q := P;
P := P^.suiv;
dispose (q);
End;
```

• Function sommet (P: Pile): élément;

```
Begin
sommet := P^.suiv;   { val }
End;
```

• Function test-vide (P: Pile): boolean;

```
Begin
test-vide := P = Nil;
End;
```

3- Les Files :

Dans les file les ajouts se font à une extrémité et les supressions à l'autre (ext raim) extrémité.

L'élément présent depuis le plus longtemps est l'élément premier. les files sont appellées FIFO (First in First out)

Les opérations sur les piles sont :

1- tester si une piles est vide.

2- Accéder au premier élément du file.

3- Ajouter un élément dans une file.

4- Retirer un élément de la file.

sorte File
utilise booleam, élément
Opérations :
File-vide : → File.
Ajouter : File x element → File.
Retirer : File → File.
Premier : File → élément.
test-vide : File → booleam.

3.1 : Représentation contigue :

Il faut procédé (conserve) l'indice " i " du premier élément et l'indice " j " de la première case vide (libre

Les indices i, j progressent au modulant Lmax

3.2 : Représentation chaînée :

a)- On utilise 2 pointeurs ; l'un vers la tête et l'autre vers le dernier élément de la (piles) file.

b)- On utilise une liste serculaire : le dernier élément pointe vers la tête.

Les ensembles :

La notion d'ordre des éléments est fondamentale dans certains cas l'ordre des " peut ne pas être important et la propriété est (pas) donnée à la présence ou l'absence d'un élément, c'est la notion des ensembles d'éléments.

- Les opérations sur les éléments sont :

1. L'appartenance d'un élément à un ensemble.
2. Ensemble vide
3. Ajouter un élément à un ensemble.
4. Suprimer un élément d'un ensemble.

Les ensembles qui tolèrent les répétitions de même élément sont appelés ensembles avec répétition ou multi-ensemble.

1. Spécification abstraite :

- Sorte ensemble
- Utilise élément, booleau

• Operations :

Ensemble_vide : → Ensemble

Ajouter : Element * ensemble → Ensemble

$\in$: Elément * Ensemble → booleau

2. Incrementation des ensembles :

Dans les ensembles l'ordre du traitement des éléments n'est pas déterminé on traite tous les éléments d'un ensemble non vide ou on choisi un élément et on les traite et on recommence sur l'ensemble obtenu en suprimant l'élément traité et tand que l'ensemble n'est pas vide

choisit (e)

tand que ¬(test_vide) Faire

Debut
  Traiter (e);
  Supprimer (e);
  choisi (e);
Fin

4- Cas des multi-Ensembles:

Il y'a une opération supplimentaire qui donne le nombre d'occurence (Nb_occurence) d'un élement dans un multi-ensembles.

Nb_occurences: Element x Ensemble → Entier;

4- Représentation des ensembles:

4.1: Rprst par les tableaux de booleans:

Lorsqu'on travail avec des tableaux dont les élements possible sont en ordre fini, il est possible de représenter L'ensemble par tableau de booléens.

Type Ensemble = Array [1.. N] of boolean;

Var E = Ensemble;

E[i] a la valeur vraie
si $i \in E$

E[i] a la valeur fausse
si $i \notin E$

opérations:

programme:

Ajouter (x,e)	E [x] := true;
Supprimer(x,e)	E [x] := False;
x ∈ E	E[x]
Ensemble_vide	For i = 1 to N Do E[i] := False
Card(E)	c := 0; For i = 1 to N Do if E[i] then c := c + 1

Remarques:

- Cette représentation peut être couteuse en place mémoire surtout si N est grand et les ensembles considérés n'ont pas beaucoup d'élement.
- Les opérations d'appartenance des lieux d'ajout et du suppressions representées d'une manière très efficace. puisqu'elle correspond à un accès ou une affectation.
- L'ensemble dit est représenté par un tableau où tout les élements sont faux.
- Dans le cas de multi-Ensembles le tableau booleau doit être remplacé par un tableau d'entier (chaque case représente le nb. d'occurences d'un élement).
- Dans le langage pascal le type Ensemble est défini par: set of < Type de base >

```
Type
   intervalle: 1..10;
   ensemble = Set of intervalle.
```

4.2- Représentation Chaînée :

Un ensemble est représenté par une liste de N éléments.
Il y'a plusieurs possibilités de la représentation du liste car l'ordre n'est pas important.
Les test d'appartenance d'un élément à un ensemble se fait par un parcours séquentiel de liste. En comparant chaque élément à x ; Le parcours s'arrête lorsqu'on trouve x (succès) ou bien lorsqu'on a la liste vide (échec).

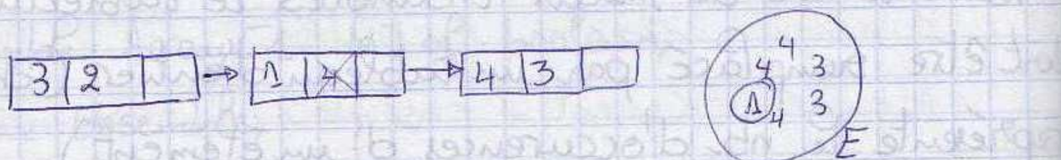
L'ajout d'un élément se fait à une place quelconque de la liste, on distingue L'ajout paresseux qui ne test pas si un élement à ajouter est déja présent dans la liste et L'ajout non paresseux qui présente un avantage d'économie de place mémoire

Dans le multi-Ensemble une autre représentation peut être envisager : une liste de couple

<x, nb_occ>

x : Element ;

nb_occ : Nb d'occurences de x dans L'ensemble nb_occ ≠ 0



VI- Structure Arborésentes :

Un arbre est un ensemble de noeud ordonné de façon hiérachique à partir d'un noeud distingué appeler racine.
La structure d'arbre est la une des plus importantes d'informatique ;
Exemple : Organization des dossiers et des fichiers dans le système d'explôitation.

2)_ Arbre binaire :

Un arbre binaire soit vide noté ϕ soit de la forme B: $<o, B_1, B_2>$ où : B_1, B_2 sont des arbres binaire disjoints et o est la racine.

. Une arbre binaire n'est pas similé B ^A C

Type abstrait arbre binaire :

Sorte arbre

Utilise c noeud, element

Opérations :

Arbre_vide : → arbre

<-,-,->: noeud x arbre x arbre → arbre

racine : Arbre → noeud

g : Arbre → arbre

d : Arbre → arbre

preconditions :

racine (o) est défini ssi o ≠ arbre_vide

g(a) est défini ssi g(a) ≠ arbre_vide

Rqs : Un arbre dont les noeuds contient des élements est dit arbre étiquetté.

* Soit L'arbre $B = <o, B_1, B_2>$ alors o est la racine de B_1 et B_2 sous arbre de droit B.

* On dit que c est un sous arbre de B ssi c = B où $c = B_1$ ou $c = B_2$.

. Définitions :

On appelle fils gauche (respectivement droite) d'un noeud, la racine de son sous arbre fils.

On dit qu'il y'a un lien gauche (respectivement droite) entre la racine et son fils gauche (" droite) si un noeud n_i un fils gauche (respectivement droite) un noeud n_j ; on dit que n_i le père de n_j.

Remarques:

- Chaque noeud n'a qu'un seul père.
- Deux noeuds qui ont le même père sont des frères.
- Le noeud n_i est un descendante ou cètre des noeuds n_j ssi n_i est le père n_j ou un adescendante du père n_j.

n_i est un adescendante de n_j ssi n_i est le fils de n_j ou n_i est le descendant est le fils n_j

- Tout les noeuds d'un arbre binaire ont aux plus 2 fils
- Un noeud qui a 2 fils est appelle noeud interne ou point double.
- Un noeud qui a seulement un fils gauche (respectivement fils à droite) est appellé point simple à gauche (resp à droite) ou noeud interne au sens large.
- Un noeud sans fils est appellé noeud externe ou feuille.
- On appelle branche de l'arbre R tout chemaine

Un chemaine est une suite de noeud consécutifs de la racine à une fiulle.

Il y'a autand de branches que des feuilles dans un arbre binaire.

*** Mesures sur les arbres:**

La taille d'un arbre binaire est le nombre des noeuds

$$Taille(\text{arbre vide}) = 0$$

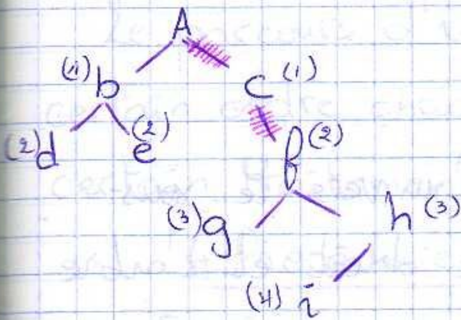
$$Taille(<o, B_1, B_2>) = 1 + taille(B_1) + taille(B_2)$$

La hauteur d'un noeud (profondeur) est le nombre de lien sur l'unique chemaine de la racine à se noeud.

La longueur de cheminement $Lc(B) = \sum R(x)$ pour tous les noeuds x de R

$Lc(B) = 18$

. Hauteur (e) $\begin{cases} 0 \text{ si } e = 1 \text{ racine} \\ 1 \text{ si père (e) = racine} \end{cases}$

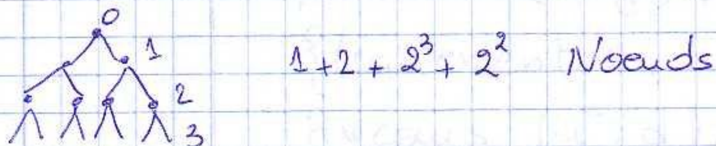


Entre A et f il y a 2 liens

Arbre binaire particulière :

a./ Filiforme ou dégenère :

Un arbre binaire est dit complé s'il contient un noeud au niveau 0, et deux noeud au niveau 1.



$1 + 2 + 2^3 + 2^2$ Noeuds

Un arbre binaire est dit parfait ; c'est un arbre dont tous les niveaux sont complétement remplitent sans erentu élément le dernier niveau et dans ce cas les feuilles sont regroupées le plus à gauche possible.

Un peigne gauche :

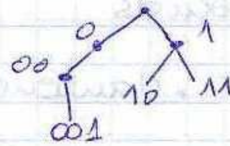
Est un arbre binaire borné (dont) dans lequel tous les fils sont des feuille (droit resp gauche).

Occurence et numérotation particulière :

On désigne un noeud de l'arbre lui associant un mot formé de 0 et 1 qui décrit le chemin de la racine à ce noeud ; ce mot est appellé occurence de noeud dans l'arbre.

La définition d'occurence de la racine est le mot vide noté ε <u>si</u> un noeud de l'arbre a un occurence x ; son fils gauche aura pour occurence x_0 et son fils droit x_1

$\{\varepsilon, 0, 1, 00, 10, 11, 001\}$

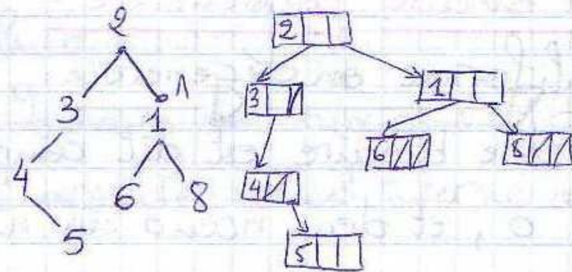


Représentation des arbres binaires :

a°/ Utilisation des pointeurs :

A chaque noeud on associe 2 pointeurs l'un vers le sous-arbre gauche et l'autre vers le sous-arbre droit et l'arbre est déterminée par l'adresse de sa racine

```
Type arbre = ^noeud
     noeud = record
        val : element;
        g, d : arbre;
     End;
```



Les (types) opérations de type abstrait ; l'arbre binaire se traduise :

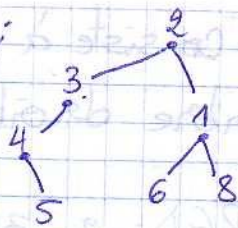
A = nil correspond à A = arbre_vide
A^.val " à continu (racine (A))
A^.g et A^.d : sous arbres gauche et droit.

b°/ Utilisation des tableaux :

```
Type element = record
        Val : element;
        g, d : 0..N;
     End;
  Arbre = record
     T : Array [1..N] of element;
```

rac : 0..N;

End;



	elt	ind G	ind D
1	3	7	-
2	-	-	-
3	2	1	4
4	1	5	6
5	6	-	-
6	8	-	-
7	4	-	8
8	5	-	-

Parcours:

Le parcours d'un arbre consiste à examiner dans un certain ordre chaqu'un des noeuds pour y effectuer un certain traitement ;

Procedure Parcours (a : arbre);

```
Begin
  if a = nil then {terminaison}
  else Begin
         {traitement 1};
         parcours (a^.g);
         {traitement 2;
         Parcours (a^.d);
         {traitement 3};
       End;
End;
```

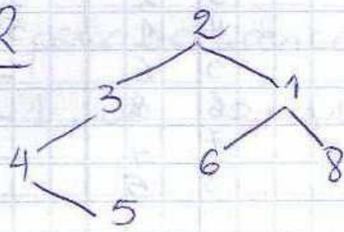
On examine d'abord la racine suivi de sous-arbre gauche suivi de sous-arbre droit.

Exercice :

Ecrire un sous-programme itlératif de parcours d'un arbre binaire selon le parcours est RGD (racine, gauche, droit).

- L'ordre infixe (symétrique) : consiste à examiner le sous_arbre gauche suivi de la racine suivi de sous arbre droit : GRD.

L'ordre suffixe (postordre) : consiste à examiner le sous arbre gauche suivi de sous-arbre droit suivi de la racine : GDR



RGD : 2 3 4 5 1 6 8

GRD : 4 5 3 2 6 1 8

GDR : 5 4 3 6 8 1 2

Exercice :

1°/ Sous-programme qui calcule la taille d'un arbre binaire.

2°/ Sous-programme qui calcule l'hauteur " ".

3°/ Sous-programme qui " " d'un noeud d'un arbre.

. Parcours RGD itératif d'un arbre binaire.

```
# procedure Parcours (a : arbre);
    Var P : Pile;
Begin
  P := Pile_vide;
  while (a<>nil) and (not (Pile_vide (P))) Do
    Begin
      traiter (a^.val);
      Empiler (P, a^.d);
      a := a^.g;
    End;
  if not (Pile-vide (P)) then
    Begin
      a := sommet(P);
      dépiler (P);
    End;
End;
```

Arbre planaires généraux:

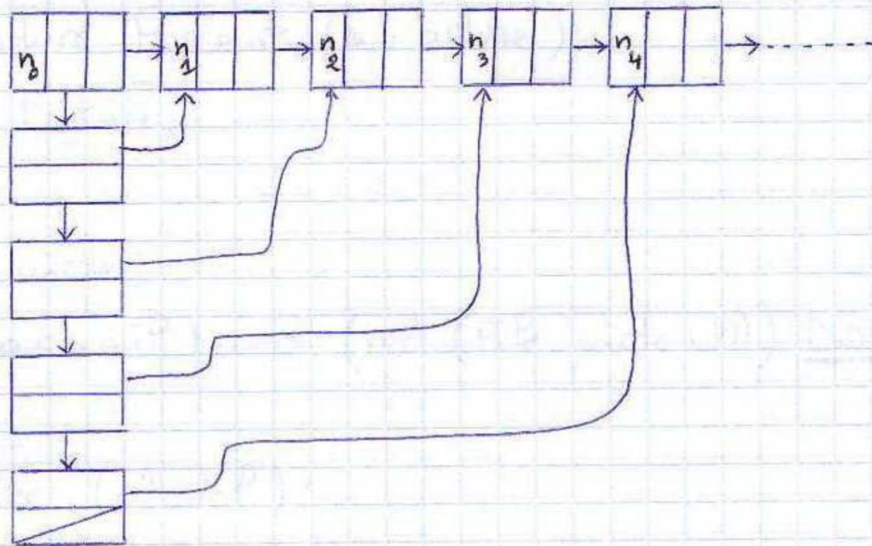
$A = \langle o, A_1, A_2, \ldots A_p \rangle$; o: racine;

$\langle A_1, \ldots A_p \rangle$: fôret : Ensemble d'arbres disjoints.

2.1: Représentation des arbres planaires:

a/. Représentation simple:

```
Type  arbre = ^noeud ; ptr = ^elt ;
      elt = record            noeud = record
        Val: ptr                Val: elt;
        suiv: arbre;            fils: ptr;
      End;                      suiv: arbre;
                              End;
```



Exercice.

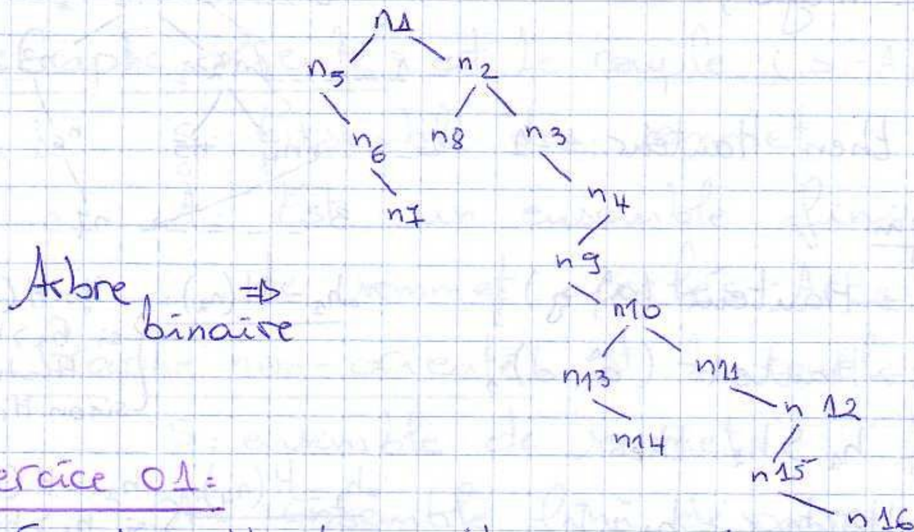
. Ecrire un sous-programme qui calcule La hauteur d'un arbre planaire

. Ecrire un sous-programme qui calcule L'hauteur d'un noeud dans une arbre planaire.

b/- Représentation sous-forme d'un arbre binaire :

On peut établir une correspondance entre les arbres planaires généraux et les arbres binaires par la méthode suivante : Bijection fils aîné frère droit.

Pour obtenir l'arbre binaire associé à une forêt on construit pour chaque noeud un lien gauche vers son premier fils (fils aîné) et un lien droit vers son (1er) frère situé immédiatement à son sa droite dans la forêt. (on considère que les racines des différents arbres de la forêt son des noeud frère

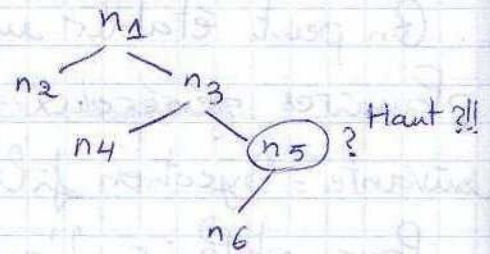


Exercice 01 :

Fonction Hauteur d'un noeud dans une arbre binaire.

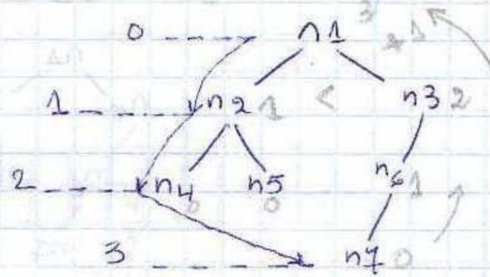
```
Function Hauteur ( a: arbre; Nd: elt); integer;
   Function Haut (a:arbre): integer;
      Var h: integer;
   Begin
   if a=Nil then Haut :=-1
   else if a^.val = Nd then Haut := 0
      else Begin
           h:= Haut (a^.g);
           if h=-1 then h:= Haut (a^.d)
```

```
    if h=-1 then Haut:=-1
    else Haut:= h+1
    End;
  End;
Begin
  Hauteur:= Haut(a);
End;
```



Fonction Hauteur d'un arbre binaire:

```
Function Hauteur (a:arbre): integer;
Var h1, h2: integer;
Begin
  if a=nil then Hauteur:=-1
  else Begin
    h1:= Hauteur (a^.g);
    h2:= Hauteur (a^.d);
    if h1 > h2 then
      Hauteur:= h1 + 1
    else Hauteur:= h2 + 1;
  End;
End;
```



- $h_1 = H(n_2)$: $H_1 := H(n_3)$
  - si $h_1 > h_2$ alors $H_1 := h_1 + 1$
  - sinon $H_1 := h_2 + 1$
- $h_1 = H(n_4)$: $h_1 := H(n5)$
  - si $h_1 > h_2$ alors $H := h_1 + 1$
  - sinon $H := h_2 + 1$;

Les Graphes :

Définition des sommets :

Un graphe est un ensemble d'objet appellés sommet et relations entre ces sommets.

Le domaine d'application des graphe sont les gestions de reseau de communication ordonancement des tâches.

_Graphe orienté = est le couple (S - A)

S: Ensemble des sommet

A: Est un ensemble fini de paires ordonné de sommet appelés -Arcs.

_Graphe non-orienté : est le couple (S-A)

S: ensemble de sommet.

A: Ensemble fini de paires de sommets appellés arrêts.

_Graphe valué orienté : (respectivement non orienté) est un triple (S-A-c)

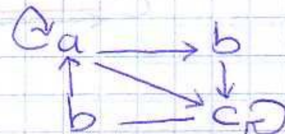
S: Ensemble de sommet

A: Ensemble arcs (resp arrêts)

c: fonction $A \to R$ Appellées fonctions de coût.

Cette valuation permet la recherche de plus court chemin entre 2 sommets.

Rq: certain graphe admet des boocles c-à-d des arcs (ou arêtes) qui connect un sommet à lui même



Les Graphes :

1. Définition des sommets :

Un graphe est un ensemble d'objet appellés sommet et de relations entre ces sommets.

Les domaine d'application des graphe sont les gestions des reseau de communication ordonancement des tâches.

- Graphe orienté : est le couple (S - A)

  S : Ensemble des sommet

  A : Est un ensemble fini de paires ordonné de sommet appelés : Arcs.

- Graphe non-orienté : est le couple (S-A)

  S : ensemble de sommet.

  A : Ensemble fini de paires de sommets appellés arrêts.

- Graphe valué orienté : (respectivement non orienté) est un triple (S-A-c)

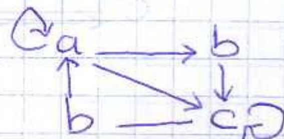
  S : Ensemble de sommet

  A : Ensemble arcs (resp arrêts)

  c : fonction $A \to R$ Appellées fonctions de coût.

Cette valuation permet la recherche de plus court chemin entre 2 sommets.

Rq : certain graphe admet des boucles c-à-d des arcs (ou arêtes) qui connect un sommet à lui même.



On peut trouver aussi deux graphe où 2 sommets sont reliées par plusieurs arcs (   ) correspondants à des relation différents ; On parle de multigraphe.

2. Terminologie :

On note $x \to y$ l'arc (x, y)

x : extremité initiale.

y : extrémité terminale.

y : successeur de x

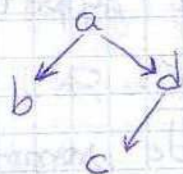
x : Prédecesseur de y.

$x \to y$ (arête (x, y)) x et y extrimités de l'arête.

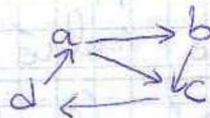
Soit le graphe $G = \langle S, A \rangle$ le sous-graphe de G engendré par $S' \subset S$ et le graphe G dont les sommets sont les élements de S' et les arcs (arêtes) sont les arcs de G ayant les 2 extremité de S'

$S = \{a, b, c, d\}$

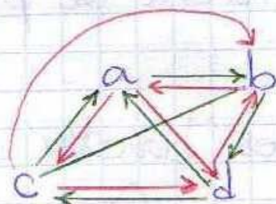
$S' = \{b, c, d\}$



Deux arcs (respectivement arêtes) d'un graphe orienté (resp non orienté) sont dite adjacentes s'il sont au moins une extremité comme :

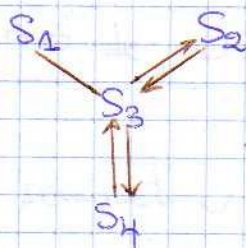


Deux sommets d'un graphe non orientés sont dit adjacente s'il existe une arête qui jougne un graphe orienté (resp non orienté) est dit complet si pour tous couple (x, y) il existe un arc $x \to y$ (resp une arête $x \to y$)



Dans un graphe orienté si un sommet x est l'extrémité initiale de l'arc : $u = x \rightarrow y$ est l'extrémité de l'arc x.y ; On dit que l'arc U est un incident à x vers l'extérieur.

Le nombre d'arcs ayant l'extrémité x initiale se note d(x) demi degré exterieur de x de la même façon on défini l'arc indécent y vers l'interieur et le demi interieur d'un sommet.



$\Leftrightarrow$ $d^\circ(x) = d^{\circ +}(x) + d^{\circ -}(x)$

$d^\circ(S_3) = d^{\circ +}(S_3) + S^{\circ -}(S_3)$
$= 2 + 3 = 5$

Dans un graphe orienté G (resp non orienté) on appelle chemin (resp chaîne) de longueur l une suite de $l+1$ sommet $(S_0, S_1, \ldots, S_l)$ tq :

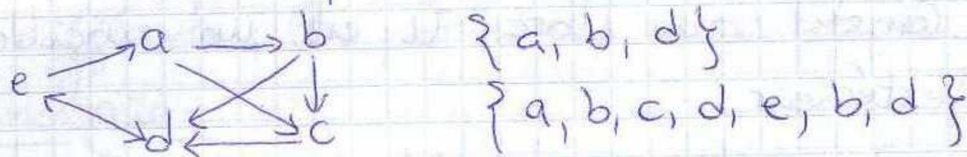
$\forall \; 0 \leq i < l-1 : S_i \rightarrow S_{i+1}$ est un arc (resp arête) de G par convention la longueur d'un chemin vers lui même est nieux.

Définition récursive d'un chemin de longueur ayant d'un sommet x vers un sommet y :

Si $l = 1$ ayant de x vers y

sinon le suite composé d'un arc de x vers un certain sommet z est un chemin de z vers y de longueur $(l-1)$.

Exemple : Un chemin (resp chaîne) est dit élémentaire s'il ne contient pas plusieurs fois le même sommet



Un graphe non orienté est dit connexe si tous paire de sommets disting {u, v} il existe une chaîne relie u → v.

3. Le type abstrait graphe :

Soit sommet
utilise Entier
Opérations :
Som : entier → sommet
N° : sommet → entier

3.1 - Spécification des graphe orientés :

On prend les conventions suivantes :
- Qu'on ajoute un sommet il est isolé (aucun arc incident)
- Quand on ajoute à cette arc n'appartient pas au graphe on doit les ajoutés.
- Quand on supprime, tous les arc incidents sont supprimés un sommet.
- Quand on supprime un arc les sommet adjacents ne sont pas supprimés.

* Type abstrait Graphe :
Soit graphe (orienté)
utilise : sommet, entier, booleen
Operations :
Graphe-vide : → Graphe.

Ajouter_sommet v a v : sommet x somet x graphe
→ Graphe.

v : est sommet de : Sommet x graphe → booleean
<-,-> est arc de : Sommet x sommet x graphe
→ booleean.

d^{o+} de v dans ◡ : Sommet graphe → entier
ième succ de v dans ◡ : entier x sommet graphe
→ sommet.

ième pred de v dans ◡ : entier x sommet x graphe
→ sommet.

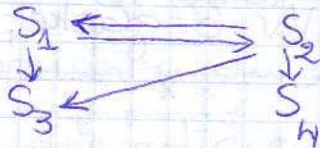
Retrie sommet v de ◡ : sommet x graphe → graphe
Retrie x arc v de ◡ : sommet x sommet x graphe
→ Graphe

4 Représentation :

4.1 :

Cette représentation est utilisée Lorque la ensemble de sommet ne volue pas.

Le graphe alors est représenté par une matrice de booleean appellée matrice adjacente.



Type Graphe = Array [1..n, 1..n] of booleean;

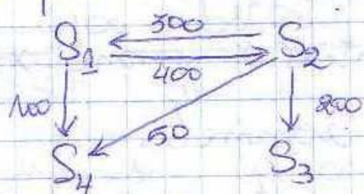
Rq : . l'élément d'indice i, j est vrai s'il existe un arc entre les sommet i, j

. Si la graphe est non orientée la matrice adjacente.

	1	2	3	4
1	F	T	T	F
2	T	F	T	T
3	F	F	F	F
4	F	F	F	F

	1	2	3	4
1	F	T	F	T
2	F	F	T	T
3	F	T	F	F
4	T	T	F	F

• Si le graphe est valu a utilisé une matrice où l'élément l'indice i, j a pour valeur le poid de l'arc i, j ;
- Si l'arc (respc l'arête i → j) existe sinon une valeur particulière est utilisée comme poids.



-1	400	-1	100
300	-1	50	20
-1	-1	-1	-1
-1	-1	-1	-1

. Le test de l'existance, l'ajout; le retrait d'un arc sont des opérations facile.

```
# Procedure Ajout (Var G: graphe; i,j: integer);
Begin
  G[i,j] := True;
End;
```

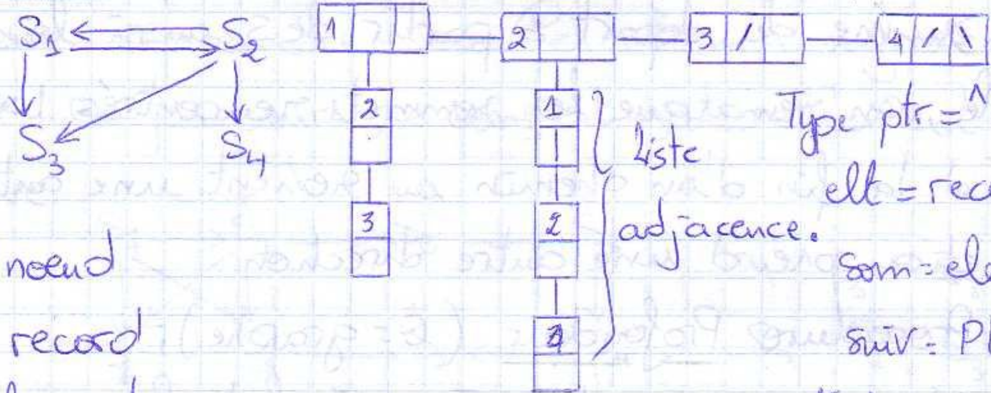
```
# Procedure Retrait (Var G: graphe; i,j: integer);
Begin
  G[i,j] := False;
End;
```

4.2: Utilisation des listes d'adjacence-

Une représentation du graphe consiste a représenter l'ensemble de sommets des graphes et a associer à chaque sommet la liste des ses séquencesseurs rangés dans certain ordre, ces listes sont appellées liste

Dans le cas où les ensemble des sommets n'evalue pas; ces liste sont accessibles, à partir d'un tableau qui contient chaque sommet vers le début de la liste.

a°/- Cas où l'ensemble de sommets n'evalue pas:



Type ptr = ^elt;
elt = record
Som: element;
Suiv: Ptr;
End;

Graphe = ^noeud
noeud = record
Som: element;
Suc : ptr;
Suiv: Graphe;
End;

Remarque: Cette représentation économise l'espace mémoire, le teste d'existance, l'ajout et le retrait sont des opérations coûteuses.

• Le calcule du demi degré $d^{o-}(x)$ interieur d'un sommet n'est pas facile; il nécessite la recherche du sommet x dans tous les listes adjacences de graphe, l'ajout ou la suppression du sommet nécessite une manipulation de liste adjacence.

b°/ Cas où l'ensemble de sommet n'avalue pas:

Graphe: Array [1..N] of ptr;
Type = Ptr = ^elt;
elt = record
Som: element;
Suiv: ptr;
End;

5- Parcours dans un Graphe :

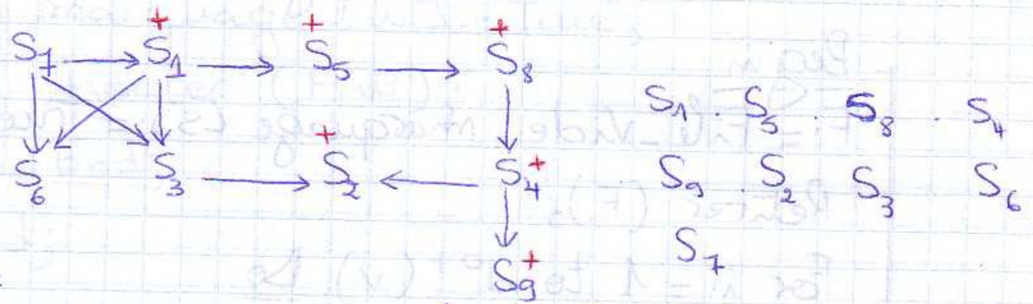
On considère un graphe orienté dont tous les sommets utilisés non marqués; le parcours consi) n profondeur consiste à choisir un sommet de départ S a remarquer et à suivre de départ à partir de S aussi loin que possible; on remarque les sommets rencontrés Lorsque on arrive à la fin d'un chemin on revient une autre direction fait et on prend une autre direction.

```
# Procedure Profondeur (G: graphe);
   Var marquage: Array [1..N] of boolean;
       i: integer;
 Begin
 For i:= 1 to n do
 marquage [i]:= False;
   For i:= 1 to n do
   if not(marquage[i]) then
   Prof (i);
 End;
  # Procedure Prof (s: integer);
     Var i: integer;
 Begin
 marquage [s]:= True;
 For i:= 1 to n do
 if (G[i])and(marquage [i]) then
    Prof (i);
 End;
```

5.1 - Parcours en profondeur :

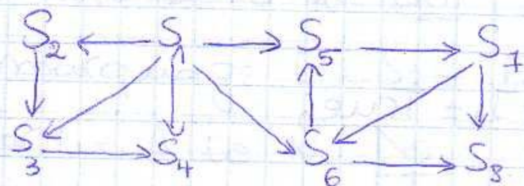
(Deph, First, search, DFS)

Consistre à partir d'un sommet donné à suivre un chemin le plus loin possible et à faire des retour pour reprendre tous les chemin ignorés précedent. ce parcour se conçoit naturellement récursive.



marquage :

F	F	F	F	F	F	F	...
T	T	T	T	T	T	T	...



exécution de la Procedure Profondeur

	1	2	3	4	5	6	7	8
1	F	T	T	T	T	T	F	F
2	F	F	T	F	F	F	F	F
3	F	F	F	T	F	F	F	F
4	F	F	F	F	F	F	F	F
5	F	F	F	F	F	F	T	F
6	F	F	F	F	T	F	F	T
7	F	F	F	F	F	T	F	T
8	F	F	F	F	F	F	F	F

5.2 - Parcours en Largeur :

(Breth, Frist, Search). BFS

à Partir d'un sommet on commence à visité à tous les successeurs de S de distance 2 ; ainsi de suite. On appelle distance de $s \to y$ la longueur de plus cours chemin allant de S vers y. ce parcours est iteratif.

\# Procedure Longueur (G: graphe);
Var i: integer;
marquage: Array [1..N] of boolean;

. Procedure Long (s: integer);
Var F: file;
v, w, i : integer;

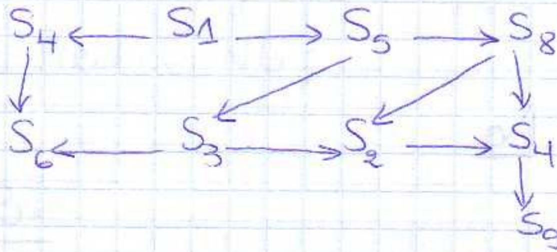
```
Begin
F:= File_Vide; marquage [s]:= True;
Retirer (F);
For i:= 1 to d°+(v) Do
   Begin
   w:= ième succ de v dans G;
   if not (marquage [w]) then
      Begin
      marquage [w]:= True;
      Ajouter (F,w);
      End;
```

Faux

. Procedure Long (s: integer);
Var F: file;
v, w, i: integer;

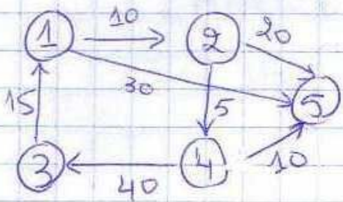
```
Begin
F:= File_vide;
marquage [s]:= True;
Ajouter (F,s);
while not (File_vide (F)) Do
   Begin
   V:= Premier (F);
```

```
        Retirer (F);
        For i:= 1 to d°+(v) Do
          Begin
          w:= ieme succ de v dans G;
          if not (marage [w]) then
            Begin
            marquage [w]:= True;
            Ajouter (F,w);
            End;
          End;
      End;
  End;
Begin
For i:= 1 to N Do
  marquage [i]:= False;
 For i:= 1 to N Do
   if not (marquage [i]) then
      larg (i);
End;
```

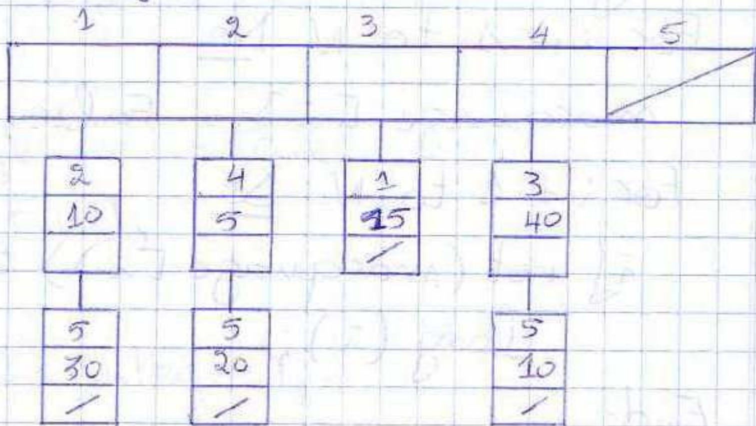


Exercice 01:

Écrire un sous-programme qui calcule la somme des poids des arcs dans un graphe orienté valué:




```
Const N = -- ;
Type Ptr = ^elt;
     elt = record
        som : 1..N;
        Poids: integer;
        suiv: Ptr;
     End;
   Graphe: Array [1..N] of Ptr;
Function Somme (G: graphe): integer;
  Var i, s: integer;
      P: Ptr;
  Begin
    S:= 0;
    For i:= 1 to N Do
      Begin
        P:= G[i];
        While P<> Nil Do
          Begin
            S:= S + P^. Poids;
            P:= P^. suiv;
          End;
      End;
    Somme:= S;
  End;
```



Exercice 02:

Ecrire le sous-programme qui calcule le nombre de prédecesseur d'un sommet dans un Graphe orienté.

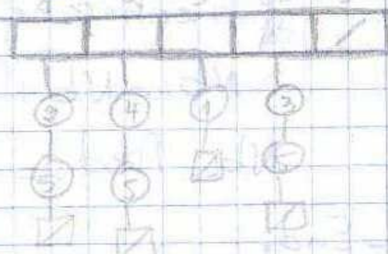
```
# Function Nbr_Pred (G : graphe ; s : integer) : integer;
    Var nb, i : integer;
    . Function recherche (P : Ptr ) : boolean;
  Begin
  While (P<> Nil) and (P^. som <> s) Do
      P := P^. suiv;
    Recherche := P <> Nil
  End;
  Begin
  Nb := 0;
  For i := 1 to s-1 Do
   if recherche (G[i]) then
      Nb := Nb + 1;
   For i := s+1 to N Do
    if recherche (G[i]) then
      Nb := Nb + 1;
     Nbr_Pred := Nb;
  End;
```

Exercice 03:

Soit Q = <S, A> un Graphe orienté r de S est dite racine de G s'il existe un chemin essu de r qui permet d'atteindre tous les sommets d'un graphe

Ecrire le sous-programme qui test l'existance d'une racine dans un graphe.

```
# Function Racine (G : graphe) : boolean ;
    Var marq : Array [1..N] of boolean;
        i, j : integer;
        existe = boolean;
    . Procedure Par (som : integer);
        Var P: Ptr;
    Begin
      marq [som] := True;
      P := G [som];
      while P<>Nil Do
        Begin
        if not (marq [P^.suiv]) then Par (P^.suiv);
          P := P^.suiv;
        End;
    End;
  Begin
  i := 1; Existe := False;
  while (i <= N) and (not (existe)) Do
     Begin
     For j := 1 to N Do
       marq [j] := False;
       par (i); j := 1;
     while (j <= 0) and (marq [j]) Do
       j := j + 1;
     if j > N then existe := True;
       c := i + 1;
     End;
   racine := Existe;
  End;
```



Exercice 04 :

Soit $G_1 = < S_1, A_1 >$ et $G_2 = < S_2, A_2 >$

Ecrire le sous-programme qui réalise les superposition de G_1 et G_2 :

- La superposition de 2 graphes est un graphe $G = < S, A >$ telque : $S := S_1 = S_2$ et $A = A_1 \cup A_2$

```
# Procedure superposition(G1, G2 : graphe; Var G : graphe);
    Var P, q : Ptr;
        i : integer;
    . Function cherche (P : Ptr; som : integer) : boolean;
  Begin
  While (P <> Nil) and (P^.suiv <> som) Do
      P := P^.suiv;
      cherche := P <> Nil;
  End;
  Begin
      For i := 1 to N Do  . Begin
  P := G1 [i];
  G [i] := Nil;
  While P <> Nil Do
      Begin
      q^.som := P^.som;
      q^.som := G [i];
        G [i] := q;
        P := P^.suiv;
      End;
  P := G2 [i];
  While P <> Nil Do
```

```
    Begin
    if not (cherche (G[i]; P^.som)) then
      Begin
      New (q); q^.suiv:= P^.som;
        q^.suiv:= G[i]; G[i]:= q;
      End;
    P:= P^.suiv;
    End;
  End;
End;
```

Partie II: Algorithme de Recherche:

Chap I: Methode Simple:

1. Introduction:

Une des utilisation les plus connues de (l'algo) L'informatique est le stockage et la collection des données qui présentes des caractéristiques communes et la recherche parmi ces données d'éléments qui satisfairont cette certain crétères.

Les opérations d'ajout, de suppression et de recherche doivent être réalisées de manière à ne pas demander un temps très long

Exemple: Annuaire téléphonique doit permettre La recherche, l'insertion et La suppression d'abonnés. On chosira une représentation qui prévéligie La performance de La recherche par rapport aux autres