Rappel sur les variables statiques

- Ce qui a été vu?
  - variables de type simple (entier, réels, caractère, etc.)
  - variables de type tableau(array), enregistrement (record)
- Exemple :

```
var c : char;
var tab : array[1..20] of integer;
var étudiant : record
                    nom, prénom : string[20];
                    dateNaiss : record
                                    jour, mois, année : integer;
                                end;
                    end;
```

Rappel sur les variables statiques

- Que se passe t'il lorsqu'on déclare ?

var N: integer

  - un certain espace, désigné par son adresse dans la mémoire est associé à la variable *N*
  - association faite par le compilateur (avant l'exécution du programme)
  - les occurrences de *N* dans le programme sont remplacées par l'adresse de la variable
  - à l'exécution toutes ces adresses sont fixées

↳ allocation statique

Variables dynamiques

- C'est quoi?

« C'est une variable pouvant être allouée pendant l'exécution, au moment où il y en a besoin! sous le contrôle du programme au moyen d'instructions spéciales. »

↳On parle alors d'allocation dynamique (contrôlée),

… ⇨***pointeurs***

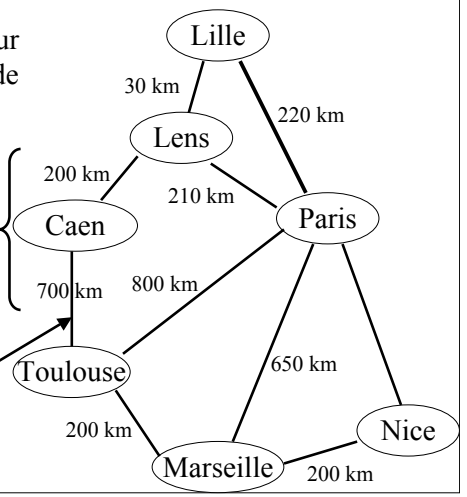
Les pointeurs

Introduction

La notion de pointeur permet la construction de structures de données :

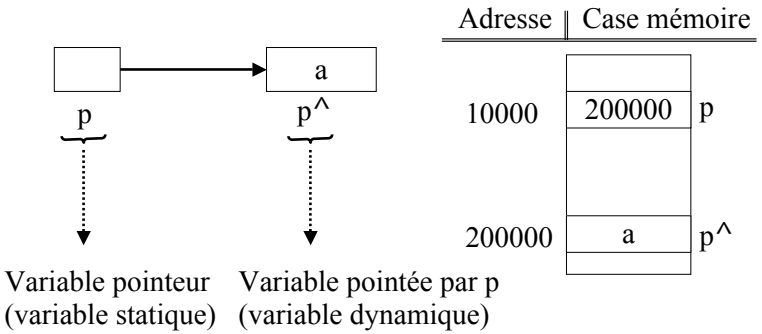
dynamique (dont la forme et la taille peuvent évoluer en cours d'exécution)

chaînées (les objets sont chaînées entre eux)



Variables de type pointeur

- Définition : une variable de type pointeur est une variable dont la valeur est l'adresse d'une autre variable.



Variables de type pointeurs

- La mise en place d'une variable dynamique se déroule en trois étapes :

1. Déclaration de la variable pointeur
2. Initialisation de la variable pointeur et/ou allocation de la variable dynamique (pointée)
3. Utilisation de la variable dynamique

Variables de type pointeur

1. Déclaration :

Pseudo (Pasc al): var p : ^ typeObjet;

- *p* est une variable statique de type pointeur vers un objet de type *typeObjet*
- *p^* désigne la variable dynamique de type « *typeObjet* » dont l'adresse est rangée dans *p*

Pseudo	*Pascal*	*Schéma*
var pi : ^entier	*var pi: ^integer;*	pi ?

« *pi* est une variable de type pointeur vers un entier »

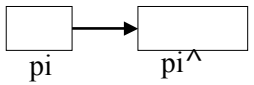
Variables de type pointeur

2. Initialisation et/ou allocation :

a) initialisation de la variable pointeur et allocation de la variable dynamique (pointée) :

- *Pseudo : allouer (p)*
- *Pascal : new(p);*

«réservation d'une zone mémoire correspondant à une variable de type 'typeObjet' et enregistre dans p l'adresse de cette zone.»

Pseudo	*Pascal*	*Schéma*
var pi : ^entier allouer (pi)	var pi : ^integer; new(pi);	

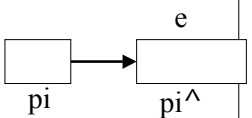
Variables de type pointeur

b) initialisation de la variable pointeur par l 'adresse d'une autre variable :

- Pseudo : p := adresse (variable)

- Pascal : p := @variable;

Pseudo	*Pascal*	*Schéma*
var pi : ^entier e : entier	var pi : ^integer; e : integer;	
pi := adresse(e)	pi := @ e;	

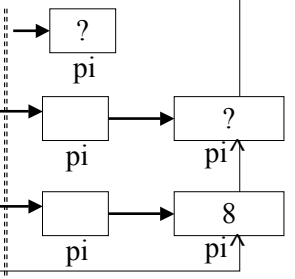


Variables de type pointeur

3. Utilisation de la variable dynamique

Exemple:

Pseudo	*Pascal*	*Schéma*
var pi : ^entier	var pi: ^integer;	
pi = allouer(entier)	new(pi);	
pi^ := 8	pi ^ := 8;	



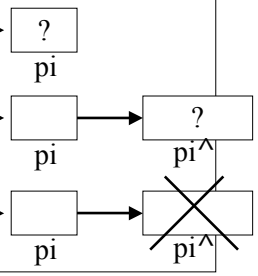
Variables de type pointeur

- Libérer la variable dynamique (pointée)

« rendre disponible l'espace mémoire occupé par la variable dynamique créé par new »

Exemple:

Pseudo	*Pascal*	*Schéma*
var pi : ^entier	var pi: ^integer;	
allouer(pi)	new(pi);	
libérer(pi)	dispose(pi);	



Variables de type pointeur

- Opérations courantes sur les pointeurs

- affectation, comparaison

- addition et soustraction (dépend du langage)!!

Constante

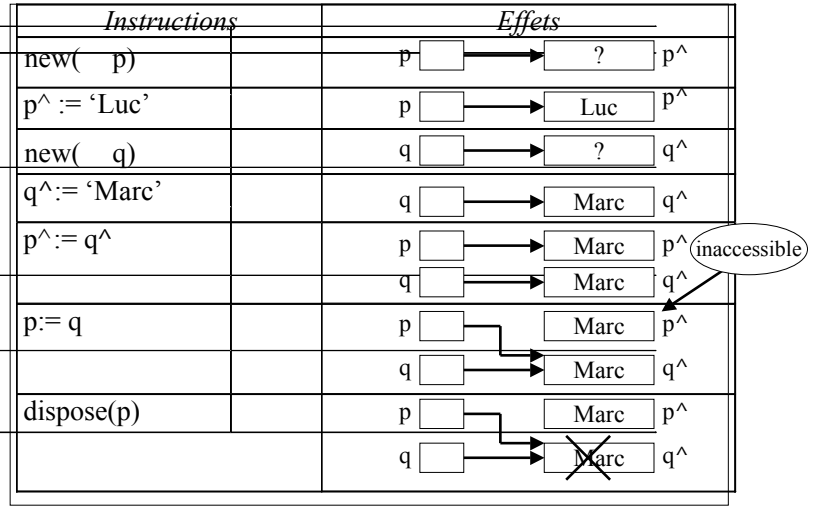
Le type pointeur admet une seule constante prédéfinie nil en Pascal et NULL en C et C++

« p:=nil; signifie que ***p pointe vers rien »***

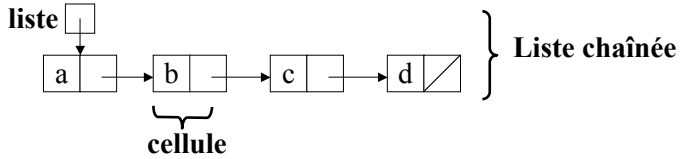
Pseudo	*Pascal*	*Schéma*
var p: ^entier p:=nil	var p: ^integer; p := nil;	

Résumé

type pchaine = ^ string ***var p, q : pchaine***



Listes chaînées



- une cellule est composée :
  - d'un *élément* de la suite
  - d'un *lien* vers la cellule suivante (pointeur)
- une liste chaînée est composé d'un ensemble de cellule

liste contient l'adresse de la première cellule; qui à son tour contient l'adresse de la cellule suivante, ...

Listes chaînées : représentation

Définition récursive d'une liste

Pascal

```
Type
Liste = ^cellule;
cellule = record
            info : typeElement;
            suivant : Liste;
          end;
var a : Liste
```

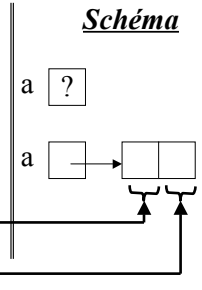
- où ***typeElement*** est un type simple (entier, caractère, ...), ou composé (tableau, structure, …)

Listes chaînées : manipulation

Accès aux champs d'une cellule

Pascal ***Schéma***

```
var a : Liste
…
new(a);
a^.info
a^.suivant
```



Listes chaînées : manipulation

Création d'une liste

Exemple: création d'une liste ***a*** représentant ***(x,y)***

var a, p: Liste; ..... **a** ? **p** ?

1. **Création d'une liste vide**

a :=nil; ..... **a** ⊘

2. **Création de la liste a contenant *(y)***

new(p); ..... p

p^.info := 'y'; ..... p — *y*

p^.suivant := a; ..... p — *y* ⊘

a := p; ..... a

p — *y* ⊘

Listes chaînées : manipulation

3. Création de la liste a contenant *(x, y)* :

new(p); ..... p

p^.info := x; ..... p — *x*

p^.suivant := a; ..... p — *x*; a — *y* ⊘

a := p; ..... p — *x*; a — *y* ⊘

on obtient : a — *x* — *y* ⊘

Listes chaînées : opérations courantes

1. Initialiser une liste à vide

```
procedure initListe( var a:Liste)
begin
   a:=nil;
end;
```

2. Tester si une liste est vide

```
fonction listeVide( a:Liste ):boolean
begin
 listeVide:= (a=nil);
end;
```

Listes chaînées : opérations courantes

3**. Insertion en tête de la liste**

```
procedure insertête(elem :typeElement, var a:Liste)
var p:Liste;
begin new(p);
   p^.info := elem;
   p^.suivant := a;
   a :=p;
end;
```

4. **Suppression en tête de la liste**

```
procedure supptête( var a:Liste)
var p:Liste;
   if (a<>nil)
   then begin
     p := a; a := a^.suivant; dispose(p);
   end;
end;
```

insertête* et *supptête

- **sont très important (utilisées souvents)!!!**
- **cas particuliers (modification de la tête de la liste)**

Listes chaînées : manipulation

- **Application : passage d 'une représentation vecteur à une représentation liste chaînée.**

```
Const Max = 100;
Type vecteur = array[1..Max] of typeElement;

procedure vecteurListe( v:vecteur, nb:integer,  var a:Liste)
begin
   a:=nil;
   for i:=1 to nb do
        insertete(v[i], a);
end;
```

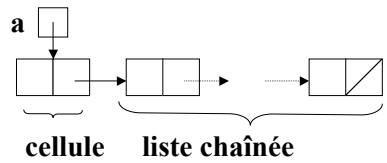
Listes chaînées : manipulation

Parcours d'une liste chaînée

Définition récursive d'une liste

Une liste chaînée est :

- **Soit une liste vide** a
- **Soit composé d'une cellule (la première) chaînée à une liste**



Remarque :

Définition très importante, à la base de tous les algorithmes récursifs sur les listes

Listes chaînées : manipulation

Premier parcours

1. **On traite la cellule courante (première)**
2. **On effectue le parcours de la sous-liste**

```
procedure parcours1(a:Liste)
begin
    if (a<>nil)
    then begin
       traiter(a^.info);
       parcours1(a^.suivant);
    end;
end;
```

Exemple

Soit la liste *a = (1, 2, 3, 4),* en remplaçant traiter par afficher, parcours1 affiche les éléments de *a* dans l'ordre

Listes chaînées : manipulation

Second parcours

1. **On effectue le parcours de la sous-liste**
2. **On traite la cellule courante (première)**

```
procedure parcours2(a:Liste)
begin
    if (a<>nil)
    then begin
       parcours2(a^.suivant);
       traiter(a^.info);
    end;
end;
```

Exemple

Soit la liste *a = (1, 2, 3, 4),* en remplaçant traiter par afficher, parcours2 affiche les éléments de *a* dans l'ordre inverse

Listes chaînées : manipulation

Version itérative

```
procedure  parcours(a:Liste)
    var p:Liste;
begin
    p:= a;
    while(p<>nil)
    do begin
          traiter(p^.info);
          p:= p^.suivant;
        end;
end;
```

Le paramètre a est passé par valeur

```
procedure  parcours(a:Liste)
    begin
      while(a<>nil) do begin
          traiter(a^.info);
          a:= a^.suivant;
      end;
    end;
```

Listes chaînées : manipulation

Exercice

Soit a = (1, 2, 3), qu'affiche la procédure suivante?

```
procedure P(a:Liste)
begin
   if (a<>nil)
   then begin
                write(a^.info);
                P(a^.suivant);
                write(a^.info);
        end;
end;
```

Résultats : 1 2 3 3 2 1

Listes chaînées : manipulation

Exercice Soit a = (1, 2, 3), qu'affiche la procédure suivante?

```
procedure P(a: Liste)
begin
    if (a<>nil) then begin
       P(a^.suivant);
       write(a^.info);
       P(a^.suivant);
    end;
end;
```

Résultats : 3 2 3 1 3 2 3

Listes chaînées particulières

Les piles

Exemple : pile d'assiettes, pile d'exécution, …

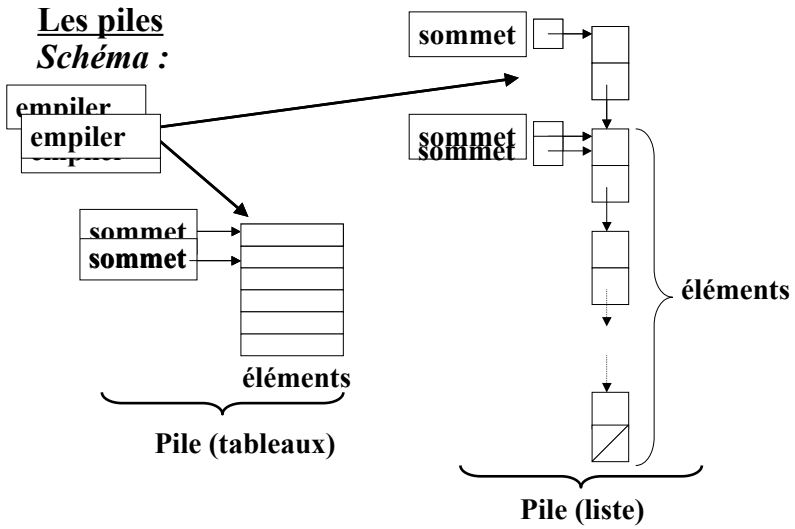
Utilisation : sauvegarder temporairement des informations en respectant leur ordre d'arrivée, et les réutiliser en ordre inverse

Principe de fonctionnement :

- Dernier arrivé, premier servi
(en Anglais « Last In First Out »
- ajout et retrait au sommet de la pile

Listes chaînées particulières

Les piles
Schéma :



Listes chaînées particulières

Les files

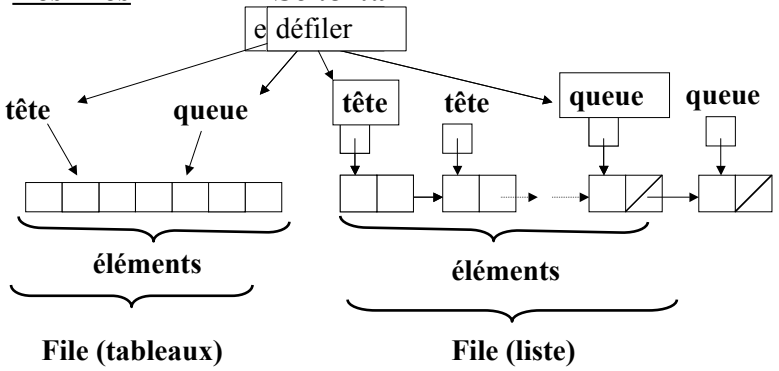
Exemple : file d'attente, file d'impression, …

Utilisation : sauvegarder temporairement des informations en respectant leur ordre d'arrivée, et les réutiliser en ordre d'arrivée

Principe de fonctionnement :
- Premier arrivé, premier servi
(en Anglais « First In First Out »)
- ajout en queue et retrait en tête de la file

Listes chaînées particulières

Les files ***Schéma***



Piles et files ⇨ TD

Listes chaînées : quelques algorithmes

Calcul du nombre d'occurrences d'un élément dans une liste

Version itérative

```
function nbOcc(val :typeElement, a:Liste):integer
    var nb :integer;
begin
    nb:=0;
    while (a<>nil) do begin
       if (a^.info =val) then nb:=nb+1;
       a := a^.suivant;
    end;
    nbOcc:=nb;
end;
```

Listes chaînées : quelques algorithmes

Calcul du nombre d'occurrences d'un élément dans une liste

Version récursive

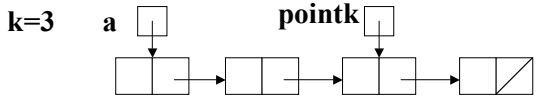
```
function nbOcc(val typeElement, a:Liste):integer
begin
     if (a=nil)
     then nbOcc:= 0
     else if (a^.info =val)
          then nbOcc:=1 + nbOcc(val, a^.suivant);
          else nbOcc:=nbOcc(val, a^.suivant);
end;
```

Listes chaînées : quelques algorithmes

Algorithmes d'accès

a) accès par position

algorithme d'accès au kème élément :



<u>données</u> :
k : entier
a : Liste
<u>résultats</u> :
pointk : Liste
<u>Spécifications</u> : retourne l'adresse de la kème cellule si elle existe; nil sinon.

Listes chaînées : Accès par position

- ***Version itérative***

```
function accesK(k:integer, a:Liste ):Liste
var i:integer:
begin
     i := 1;
     while(i<k and a<> nil) begin
        a := a^.suivant;
        i:=i+1;
     end;
     acceK:=a;
end;
```

- ***Version récursive***

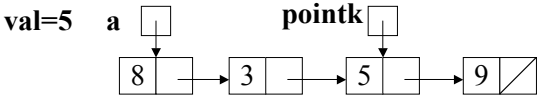
```
function accesK(k:integer, a:Liste ): Liste
begin
     if (a=nil) then accesK:= nil
     else if (k=1) then accesL:= a
          else accesK:= accesK(k-1, a^.suivant);
end;
```

Listes chaînées : quelques algorithmes

Algorithmes d'accès

b) accès par valeur

algorithme d'accès à la première occurrence d'une valeur donnée :



<u>données</u> :
val : typeElement
a : Liste
<u>résultats</u> :
pointk : Liste
<u>Spécifications</u> : retourne l'adresse de la cellule contenant la première occurrence de val; nil si val∉a

Listes chaînées : Accès par valeur

- *Version itérative*

```
function accesV(val:typeElement, a: Liste):Liste
var trouve :boolean;
begin
     trouve := false;
     while(not trouve and a<>nil) begin
        trouve := (val =a^.info);
        if (not trouve) then a := a^.suivant;
     end;
     accesV:=a;
end;
```

- *Version récursive*

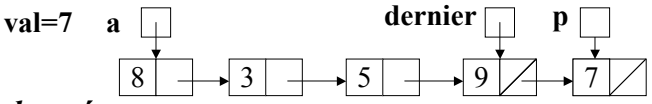
```
function accesV(val:typeElement, a:Liste):Liste
begin
      if (a=nil) then acceV:= nil;
      else if (val=a^.info) then acceV:=a
           else accesV:= accesV(val, a^.suivant);
end;
```

Listes chaînées : quelques algorithmes

Algorithmes d'insertion

a) insertion en fin de liste

algorithme d'insertion d'une valeur donnée en fin de liste



<u>données</u> :
val : typeElement
a : Liste
<u>résultats</u> :
a : Liste
<u>Spécifications</u> : insertion d'une valeur donnée val en fin de liste a

Listes chaînées : insertion en fin de liste

Calcul de l 'adresse du dernier

- *Version itérative*

```
function dernier(a:Liste):Liste;
begin
     if(a<>nil) then
        while(a^.suivant<>nil) do
           a := a^.suivant;
     dernier:=a;
end;
```

- *Version récursive*

```
function dernier(a:Liste):Liste;
begin
      if (a=nil) then dernier:=nil
      else if (a^.suivant=nil) then dernier:=a
           else dernier:= dernier(a^.suivant);
end;
```

Listes chaînées : insertion en fin de liste

- *Ajout en fin de liste*

```
procedure inserFin(val :typeElement , var a:Liste);
var der:Liste;
begin
     der := dernier(a);
     if(der=nil) then insertete(val, a)
     else insertete(val, der^.suivant);
end;
```

- *Version récursive (sans utilisation de dernier)*

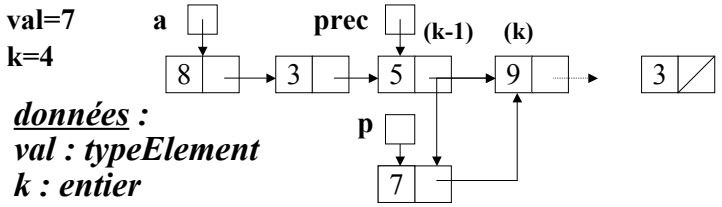
```
procedure inserFin(val:typeElement, var a:Liste)
begin
      if (a=nil) then insertete(val, a)
      else inserFin(val, a^.suivant);
end;
```

Listes chaînées : quelques algorithmes

Algorithmes d'insertion

b) insertion à la kème place

algorithme d'insertion d'une valeur à la kème place



<u>données</u> :
val : typeElement
k : entier
a : Liste
<u>résultats</u> :
a : Liste
<u>Spécifications</u> : insertion d'une valeur donnée val à la kème place de la liste a (si possible)

Listes chaînées : insertion à la kème place

- ***Version itérative***

```
procedure inserK(val:typeElement , k:integer, var a:Liste);
var  prec:Liste;
     if (k=1) then  insertete(val, a)
     else begin
            prec := accesK(k-1,a);
            if(prec<>nil) then
                insertete(val, prec^.suivant)
            else writeln(' insertion impossible ');
          end;
end;
```

- ***Version récursive***

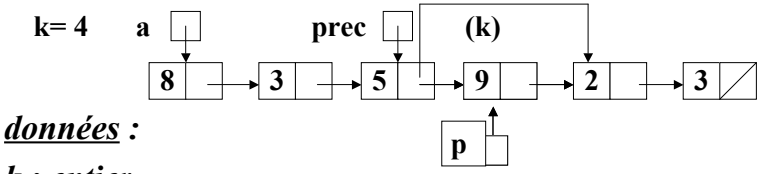
```
procedure inserK(val:typeElement, k:integer, var a:Liste);
begin
     if (k=1) then insertete(val, a)
     else if (a=nil) then writeln(' insertion impossible ')
     else inserK(val, k-1, a^.suivant);
end;
```

Listes chaînées : quelques algorithmes

Algorithmes de suppression

b) Suppression du kème élément

algorithme de suppression du kème éléments



<u>données</u> :
k : entier
a : Liste
<u>résultats</u> :
a : Liste
<u>Spécifications</u> : suppression du kème élément de la liste; si possible.

Listes chaînées : suppression du kème élément

- ***Version itérative***

```
procedure suppK(k:integer, var a:Liste)
var  prec:Liste;
     if (k=1) then supptete(a);
     else begin
            prec := accesK(k-1,a);
            if(prec<>nil) then
                supptete(prec^.suivant)
            else writeln(' suppression impossible ');
          end;
end;
```

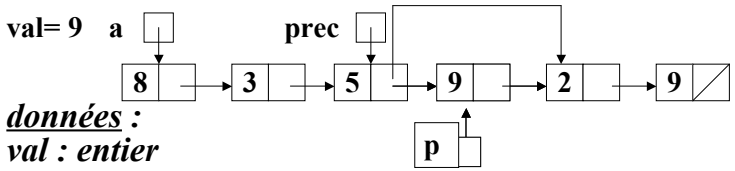
- ***Version récursive***

```
procedure  suppK(k:integer, var a:Liste)
begin  if (k=1) then supptete(a)
   else if (a=nil) then writeln(' suppression impossible ')
   else suppK(k-1, a^.suivant);
end;
```

Listes chaînées : quelques algorithmes

Algorithmes de suppression

b) Suppression de la première occurrence d 'une valeur donnée



données :
val : entier
a : Liste
résultats :
a : Liste
Spécifications : suppression de la première occurrence d'une valeur donnée.

Listes chaînées : suppression 1ère occurrence

- ***Version itérative***

```
procedure suppV(val : typeElement, var a:Liste);
var  prec,p : Liste;
     trouve: boolean;
 begin
     trouve := false; p:= a; prec:=nil;
     while (not trouve and p<>nil) do
     begin
        trouve := (val=p^.info);
        if (not trouve) then begin
           prec := p; p:=p^.suivant;
        end;
     end
     if (trouve) then
        if (prec=nil) then supptete(a)
        else supptete(prec^.suivant)
     else writeln(' suppression impossible ');
end;
```

Listes chaînées : suppression 1ère occurrence

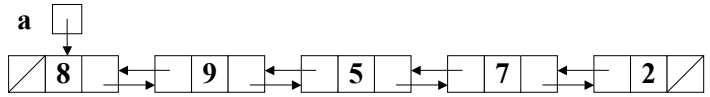
- ***Version récursive***

```
procedure suppV(val:typeElement, var a:Liste);
begin
      if (a=nil) then
            writeln(' suppression impossible ')
      else if (val=a^.info) then supptete(val, a)
            else suppV(val, a^.suivant);
end;
```

Listes chaînées particulières

Listes chaînées bidirectionnelles

Schéma



Représentation

```
type ListeBi = ^cellule
      cellule = record
                     info : typeElement;
                     precedent, suivant : ListeBi
               end;
```

Listes chaînées particulières

Listes chaînées bidirectionnelles

Utilisation

Les listes chaînées bidirectionnelles sont utilisées uniquement dans le cas où on a besoin d 'effectuer souvent des retour arrière.

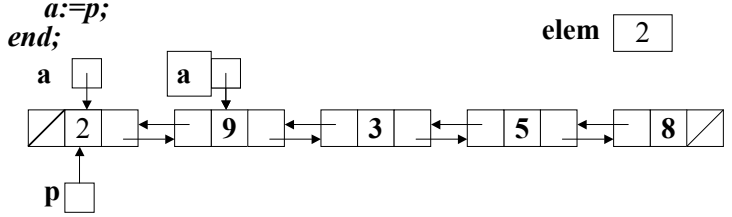
Exercices

En considérant une liste chaînée bidirectionnelle, réécrire les différents algorithmes vu en Cours

Listes chaînées bidirectionnelles

Insertion en tête d'une liste bidirectionnelle

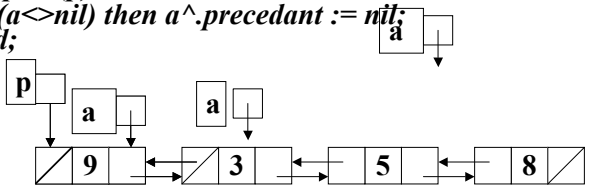
```
procedure insertête(elem :typeElement, var a: ListeBi)
var   p:ListeBi;
begin
    new(p);
    p^.info := elem;
    p^.precedant := nil;
    p^.suivant := a;
    if (a<>nil) then a^.precedant := p;
    a:=p;
end;
```



Listes chaînées bidirectionnelles

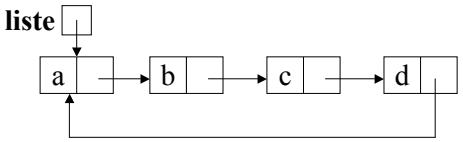
Suppression en tête d'une liste bidirectionnelle

```
procedure supptête( var
a:ListeBi)
var p:Liste;
begin
  if (a<>nil)
  then begin
          p := a;
          a := a^.suivant;
          dispose(p);
          if (a<>nil) then a^.precedant := nil;
         end;
end;
```



Listes chaînées circulaires

Schéma



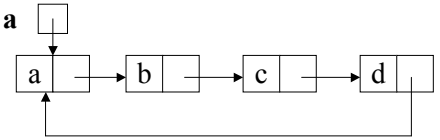
Représentation

même type que pour les listes chaînées

Exercice : écrire un procédure affichant les élément d'une liste chaînée circulaire

Listes chaînées circulaires

Version récursive




```
procedure afficher(a:Liste   , p: Liste )  // à l 'appel p = a
   begin
        if (  A<>nil       ) then
        begin   write( a^.info ) ;
                  if (a^.suivant<>p) then
                        afficher(a^.suivant   , p   );
        end;
   end;
```

Listes chaînées / tableaux

- **Liste chaînée** :
  - **avantages :**
    - **la taille peut évoluer (limite : espace mémoire)**
    - **mise à jour plus efficace : ne nécessite aucune recopie**
  - **inconvénients :**
    - **plus encombrante**
    - **accès à un élément : plus long**
- **Tableau** :
  - **avantages** :
    - **accès plus rapide**
    - **moins encombrantes**
  - **inconvénients** :
    - **plus rigide (limité par la taille spécifiée au départ)**
    - **mise à jour plus coûteuse**

Listes chaînée / tableaux

1) représentation chaînée (listes):
   mises à jour > consultations
2) représentation contiguë (tableaux) :
   consultations > mises à jour

En pratique :

On utilise très souvent des structures mixtes composées de listes chaînées et de tableaux

Polynôme?

```
Type monome = record
                    exp:integer;
                    coeff:real;
                end;
   ListeMono = ^cellule
   cellule     = record
                    m: monome;
                    monoSuivant:ListeMono;
                end;
   polynome = record
                    nbMono:integer;
                    poly : ListeMono;
                end;
```

poly

```
Procedure lireMono(var m:monome);
begin
 repeat
    write(' exp? ');readln(m.exp);
    write(' coeff? '); readln(m.coeff);
 until m.coeff<>0;
end;

Procedure inserTete(var l:listeMono;
                                m;monome);
var p:listeMono;
begin
  new(p);
  p^.m:=m;
  p^.monoSuivant:=l;
 l:=p;
end;

Procedure lirePoly(var p:polynome);
var m:monome;
    i :integer;
begin
    write(' nb Mnome? '); readln(p.nbMono);
    p.poly:=nil;
    writeln(' entrez les monomes <ordre croissant>? ');
    for i:=1 to p.nbMono do begin
          lireMono(m);
          inserTete(p.poly, m);
    end;
end;
```

poly

```
Procedure ecrireMono(var m:monome);
begin
    if(m.coeff>0) then
          write(' + ', m.coeff, ' ^ ', m.exp )
    else write(' - ', abs(m.coeff), ' ^ ', m.exp )
end;

Procedure afficherMonomes(monos:ListeMono);
begin
 if  (monos<>nil) then begin
    ecrireMonome (monos^.m);
    afficherMonomes(monos^.monoSuivant);
 end;
 writeln;
end;

Procedure afficherPolynome(p:polynome);
begin
    afficherMonomes(p.poly);
end;
```

poly

```
Procedure addPolynomes(p1:polynome; p2:polynome; var p3:polynome);
var m:monome;
begin
    p3.poly:=nil;
    while(p1.poly<>nil and p2.poly<>nil) do begin
        if(p1.poly^.m.exp=p2.poly^.m.exp) then begin
              m.coeff:= p1.poly^.m.coeff + p2.poly^.m.coeff;
              if(m.coeff<>0)  then begin
                    m.exp:=p1.poly^.m.exp;  inserTete(p3.poly, m);
              end;
        end else begin
                    if (p1.poly^.m.exp<p2.poly^.m.exp) begin
                      m.coeff:=p1.poly^.m.coeff; m.exp:=p1.poly^.m.exp;
                      inserTete(p.poly, m)
                    end else begin
                              m.coeff:=p2.poly^.m.coeff; m.exp:=p2.poly^.m.exp;
                              inserTete(p.poly, m)
                              end;
                end;
    end;
```

poly

```
    while(p1.poly<>nil) do begin
              m.coeff:=p1.poly^.m.coeff; m.exp:=p1.poly^.m.exp;
              inserTete(p3.poly, m);
    end;
    while(p2.poly<>nil) do begin
              m.coeff:=p2.poly^.m.coeff; m.exp:=p2.poly^.m.exp;
              inserTete(p3.poly, m);
    end;
end;
```

Arbres

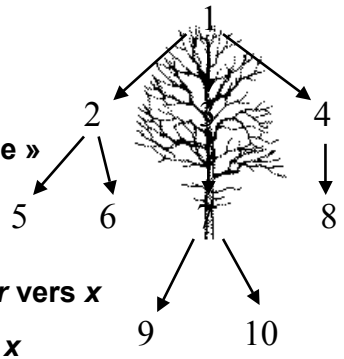
- Arbre ?

Arbre ordinaire : $A = (N,P)$

- N ensemble des nœuds

- P relation binaire « parent de »

- $r \in N$ la racine



$\forall x \in N \ \exists$ un seul chemin de r vers x

$r = y_0 \ P \ y_1 \ P \ y_2 \ \ldots \ P \ y_n = x$

↳ **r n'a pas de parent**

$\forall x \in N - \{ r \}$ x a exactement un parent

Arbres

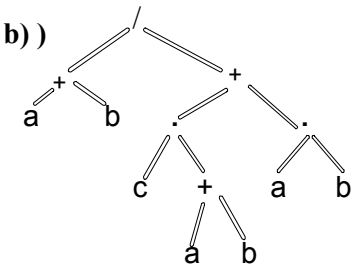
Exemples

- ***tables des matières***
  - ***1. Introduction***
  - ***2. Pointeurs***
  - ***3. Listes***
    - ***3.1. Pile***
    - ***3.2. Files***
  - ***4. Arbres***
    - ***4.1. Arbres binaires***
    - ***4.2 Arbres n-aires***
    - ***4.3. Forêts***
  - ***5. Tables***

Arbres (Exemples suite)

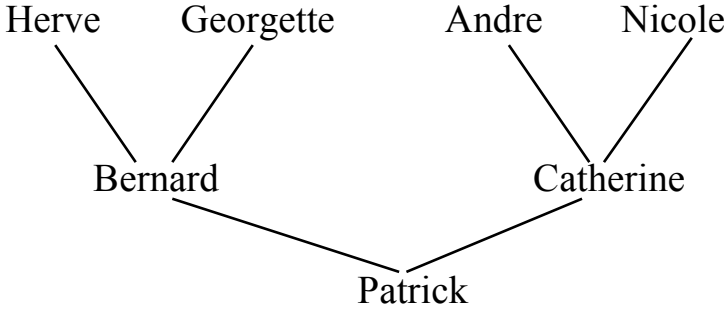
- organisation des fichiers dans des systèmes d'exploitation tels que Unix
- expression arithmétique

(a + b) / (c . (a + b) + (a . b))



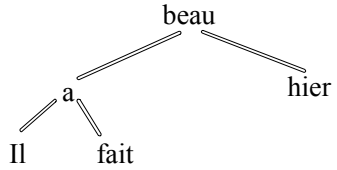
Arbres (Exemples suite)

- Arbre généalogique

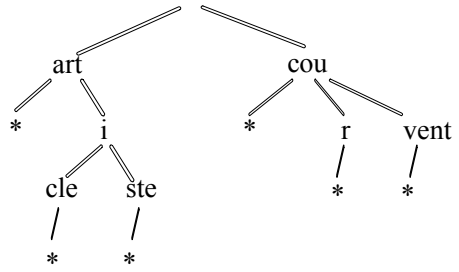


Arbres (Exemples suite)

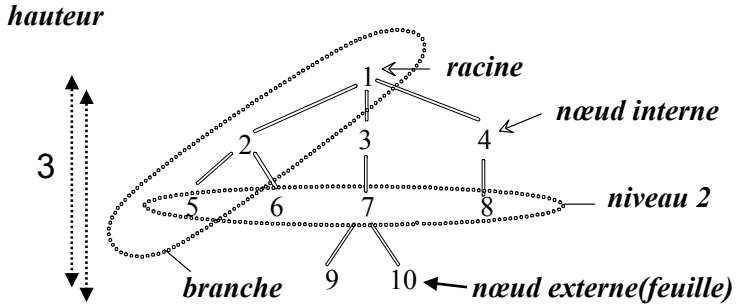
- Phrases d'une langue naturelle



- Dictionnaire



Terminologie



2, 3, 4	*enfants de 1*
3, 4	*frères de 2*
2	*fils de 1*
1	*père de 2, 3 et 4*
1, 3	*ancêtres de 7*
9, 10	*descendants de 7*

Quelques définitions

Définitions récursives

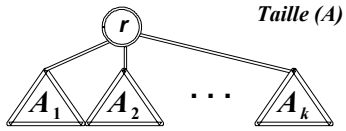
Arbre A = Λ *arbre vide ou* $(r, \{A_1, ..., A_k\})$

r élément, $A_1, ..., A_k$ *arbres*

Nœuds (A) = $\{r\} \cup (\cup$ *Nœuds* $(A_i))$

Taille (A) = $|$ *Nœuds(A)*$|$

$A = \Lambda$ **ou**



Une autre définition récursive Condition analogue

un arbre est :

- ***soit vide***
- ***soit constitué d'un nœud auquel sont chaînées un ou plusieurs sous arbres***

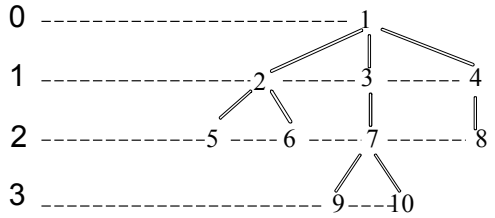
Quelques définitions

Niveaux

A arbre ***x nœud de A***

$niveau_A(x)$ = *distance de x à la racine*

$$niveau_A(x) = \begin{cases} 0 & \text{si } x = racine(A) \\ 1 + niveau\ (parent\ (x)) & \text{sinon} \end{cases}$$



Quelques définitions

Hauteurs

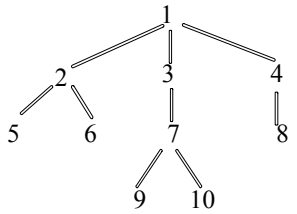
A arbre x nœud de A

$h_A(x)$ = distance de x à son plus lointain descendant qui est un nœud externe

$$h_A(x) = \begin{cases} 0 & \text{si A est vide} \\ 1 + \max \{ h_A(e) \mid e \text{ enfant de } x \} & \text{sinon} \end{cases}$$

$h(A) = h_A(\text{racine}(A))$

$h_A(8) = 1$
$h_A(7) = 2$
$h_A(3) = 3$
$h(A) = h_A(1) = 4$

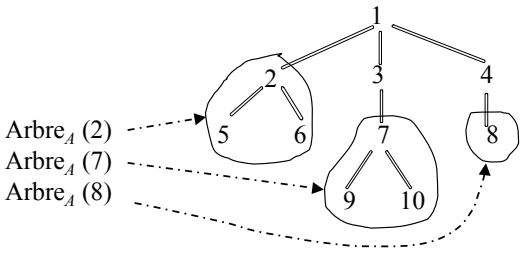


Quelques définitions

Sous-arbres

A arbre x nœud de A

$Arbre_A(x)$ = sous-arbre de A qui a racine x



Quelques définitions

Arbre binaire et arbre n-aire

- ***lorsqu 'un arbre admet, pour chaque nœud, au plus n fils, l'arbre est dit n-aire***
- ***si n est égale 2, l'arbre est dit binaire***

Remarque : un arbre n-aire peut être représenté par un arbre binaire équivalent

Quelques définitions

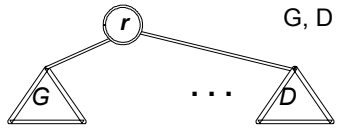
Arbre binaire : définitions récursives

Arbre binaire A = $\begin{cases} \Lambda & \textit{arbre vide} \\ \textit{ou} & \\ (r, G, D) & \end{cases}$

r élément,

G, D sous-arbres gauche et droite

***A* = Λ ou**



Une autre définition récursive un arbre binaire est :

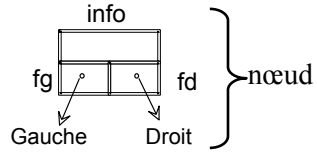
Condition analogue

- ***soit vide***
- ***soit constitué d'un nœud auquel sont chaînées un sous arbre gauche et un sous arbre droit***

Arbre binaire

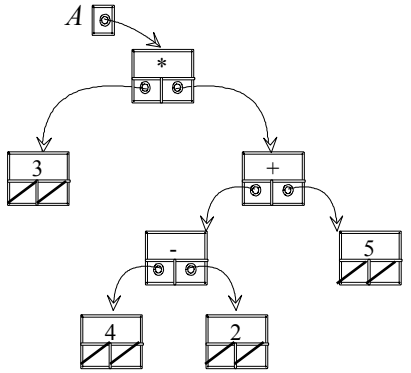
Représentation interne

```
type Arbre = ^nœud;
     nœud = record
                info : typeElement;
                fg, fd : Arbre;
            end;
```



Arbre binaire (exemple)

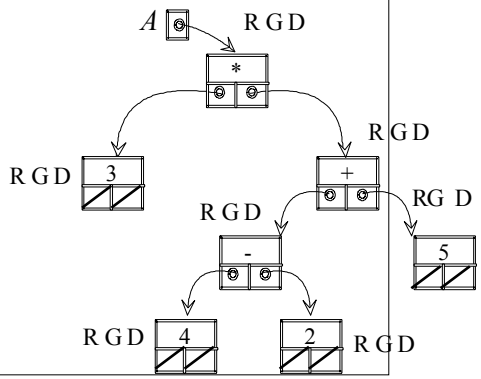
3 * ((4 - 2) + 5)



Parcours d'un arbre binaire

- Pré-ordre (préfixé, RGD)
  - racine
  - sous-arbre gauche
  - sous-arbre droit
- Sur l'exemple :

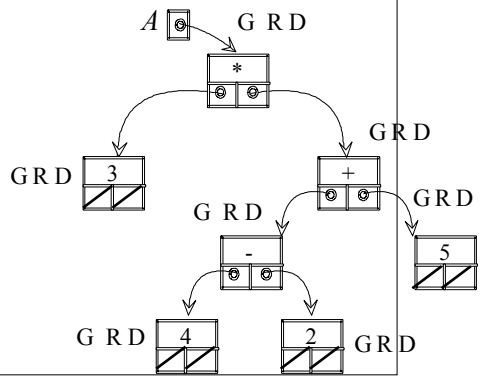
*** 3 + - 4 2 5**



Parcours d'un arbre binaire

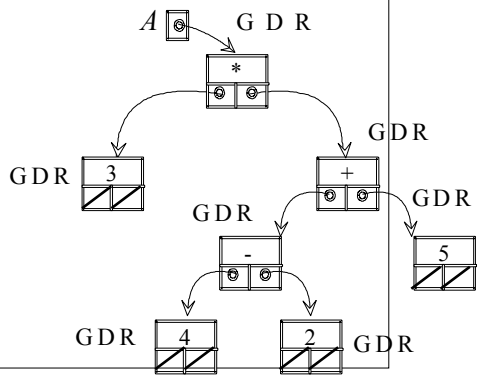
- In-ordre (infixé, GRD)
  - sous-arbre gauche
  - racine
  - sous-arbre droit
- Sur l'exemple :

3 * 4 - 2 + 5



Parcours d'un arbre binaire

- Post-ordre (postfixé, GDR)
  - sous-arbre gauche
  - sous-arbre droit
  - racine
- Sur l'exemple :



3 4 2 - 5 + *

Algorithmes

```
procedure préfixe(a: Arbre );
begin
    if (a<>nil) then begin
          traiter(a) ;
          préfixé(a^.fg);
          préfixé(a^.fd);
    end;
end;
```

```
procedure infixe(a: Arbre );
begin
    if (a<>nil) then begin
          infixé(a^.fg);
          traiter(a) ;
          infixé(a^.fd);
    end;
end;
```

```
procedure postfixe(a: Arbre );
begin
    if (a<>nil) then begin
          postfixé(a^.fg);
          postfixé(a^.fd);
          traiter(a) ;
    end;
end;
```

Algorithmes

- **Calculer la taille d'un arbre binaire**

$$\text{Taille}(A) = \begin{cases} 0 & \text{si } A = \Lambda \quad \text{arbre vide} \\ 1 + \text{Taille}(G) + \text{Taille}(D) & \text{si } A = (r, G, D) \end{cases}$$

```
function taille(a: Arbre):integer;
begin
        if (a=nil) then taille:= 0
        else taille:= 1+ taille(a^.fg) + taille(a^.fd);
end;
```

Algorithmes

- **Calculer le nombre de feuilles d'arbre binaire**

$$\text{nbFeuilles}(A) = \begin{cases} 0 & \text{si } A = \Lambda \quad \text{arbre vide} \\ 1 & \text{si } A \text{ est une feuille} \\ \text{nbFeuille}(G) + \text{nbFeuille}(D) & \text{si } A = (r, G, D) \end{cases}$$

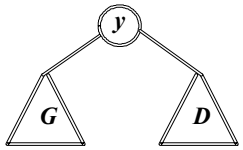
```
function nbFeuilles(a: Arbre): integer;
begin
        if (a=nil) then nbFeuilles := 0
        else if (a^.fg= nil and a^.fd=nil) then nbFeuilles:= 1
            else nbFeuilles:= nbFeuilles(a^.fg) + nbFeuilles(a^.fd);
end;
```

Algorithmes

- **Rechercher un élément**

Appartient ?

Appartient (A, x)* = vrai ssi *x* est étiquette d'un noeud de *A



Appartient (A, x) =

faux	**si *A* vide**
vrai	**si *x = y***
Appartient (G(A), x)* ou *Appartient (D(A), x);	**sinon**

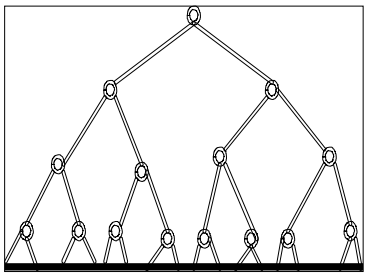
Algorithmes

```
function appartient(a:Arbre, val: typeElement)
begin
        if (a=nil)
        then appartient:=faux
        else if (a^.info =val)
             then appartient:= true
            else  appartient:= appartient(a->fg, val) or
                               appartient(a->fd, val);
end;
```

Quelques définitions

Arbre binaire complet

- **chaque nœud autre qu'une feuille admet deux descendants**
- **toutes les feuilles se trouve au même niveau**



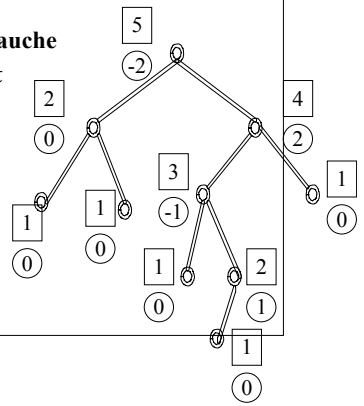
Quelques définitions

Facteur d'équilibre d 'un arbre binaire

- **le facteur d'équilibre de chaque sous arbre est associé à sa racine**
- **le facteur d'équilibre d 'un nœud est égal à la hauteur du sous-arbre gauche moins la hauteur du sous-arbre droit**

1 **La hauteur**

Facteur d'équilibre

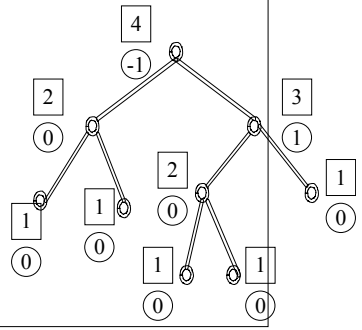


Quelques définitions

arbre binaire équilibré

- **pour tout nœud de l'arbre la valeur absolue du facteur d 'équilibre est inférieur ou égale à 1**

Arbre binaire équilibré →



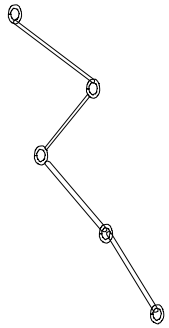
1 **La hauteur**

Facteur d'équilibre

Quelques définitions

Arbre binaire dégénéré

- **un arbre binaire est dit dégénéré, si tous les nœuds de cet arbre ont au plus 1 fils.**



Quelques définitions

Arbre binaire de ordonnée (de recherche)

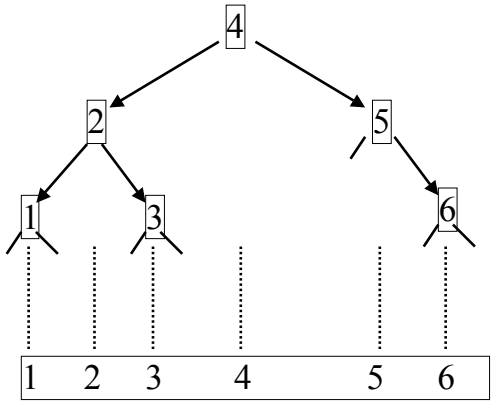
Soit A un arbre binaire
nœuds étiquetés par des éléments
A est un arbre binaire ordonnée (de recherche) :

ssi en tout noeud p de A
$Elt(G(p)) \leq Elt(p) < Elt(D(p))$

ssi A = Arbre_vide ou
A = (r, G, D) avec
. G, D arbres binaires de recherche et
$. Elt(G) \leq Elt(r) < Elt(D)$

Arbre binaire de recherche

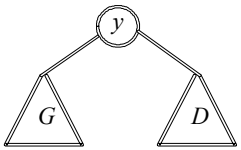
Exemple



Arbre binaire de recherche

Appartient ?

Appartient (A, x) = vrai ssi x est étiquette d'un noeud de A



Appartient (A, x) =

faux	*si A vide*
vrai	*si x = y*
Appartient (G(A), x)	*si x < y*
Appartient (D(A), x)	*si x > y*

Arbre binaire de recherche

- *Version récursive*

```
function appartient(a:Arbre, val:typeElement);
begin
   if (a=nil)
   then appartient:= faux
   else if (a^.info = val)
        then appartient:= true
        else  if (val < a^.info)
              then appartient:= appartient(a^.fg, val)
              else appartient:= appartient(a^.fd, val);
end;
```

Arbre binaire de recherche

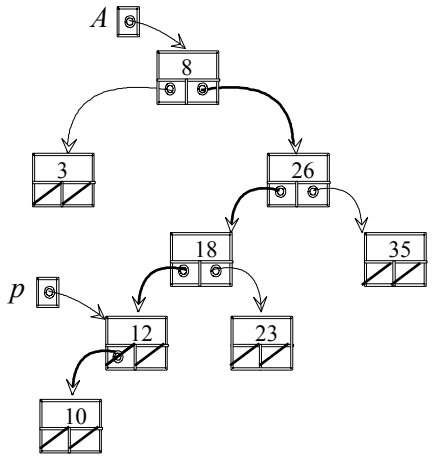
- *Version itérative*

```
function  appartient(a:Arbre, val:typeElement);
   var  trouve:boolean;
begin
   trouve:= false;
   while (a<>nil and not(trouve)) do
   begin
        trouve := (a^.info = val);
        if (val < a^.info)
        then a := a^.fg
        else a := a^.fd;
   end;
   appartient:= trouve;
end;
```

Arbre binaire de recherche

- *Ajout d'une valeur donnée*

Val = 10



Arbre binaire de recherche

- *Version itérative*

```
procedure ajout (var a:ARbre, val:integer);
var p , pere:Arbre;
begin
   p:=a; pere:=nil;
   while (p<>nil) do begin
         pere := p;
         if (val <= p^.info)
         then p := p^.fg
         else p := p^.fd;
   end;
   new(p); p^.info:= val; p^.fg :=nil; p^.fd :=nil;
   if (pere =nil) then a := p
   else if (val <=pere^.info)then  pere^.fg := p
         else pere^.fd := p;
end;
```

Arbre binaire de recherche

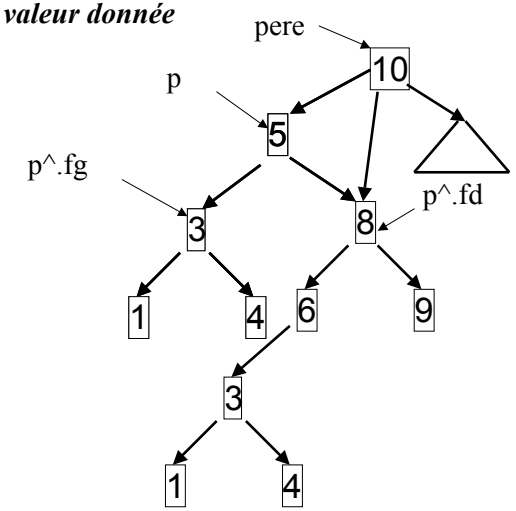
- *Version récursive*

```
procedure ajout (var a:Arbre, val:integer);
   if (a =nil)  then begin
                new(a);
                a^.info := val;
                a^.fg := nil;a^.fd := nil;
   end
 else if (val <= a^.info) then
                ajout (a^.fg, val)
        else ajout (a^.fd, val);
end;
```

Arbre binaire de recherche

- *suppression d'une valeur donnée*

  Val = 5



Arbre binaire de recherche

- *Version récursive*

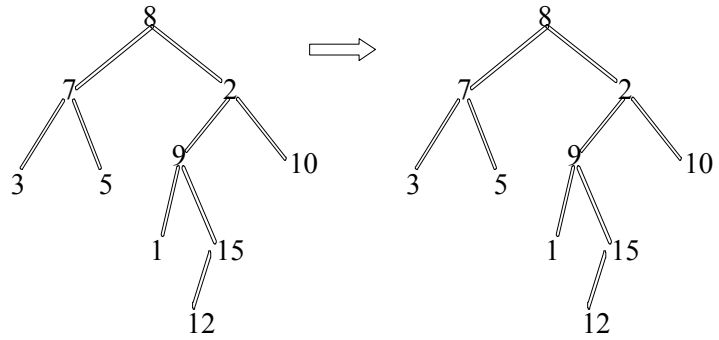
```
procedure suppression (var a:Arbre, val:integer);
begin
if (a <>nil)
       if( a^.info=val) then
          suppNoeud (a)
        else if (val < a^.info) then
                suppression (a^.fg, val)
             else suppression(a^.fd, val);
end;
```

Arbre binaire de recherche

```
procedure suppNoeud (var a:Arbre)
var p,q:Arbre;
begin
   p := a;
   if (a^.fd <>nil)  then
   begin
        q := p^.fd;
        while(q^.fg<>nil) do q := q^.fg;

        q^.fg := p^.fg;
        a := a^.fd;
   end
   else a := a^.fg;
   dispose(p);
end;
```

Arbre binaire

- *Dupliquer un arbre*



Arbre binaire

- *Dupliquer un arbre*

```
procedure dupliquer (a1:Arbre, var a2:Arbre);
begin
   if (a1 <>nil)
   then begin
                new(a2);
                a2^.info := a1^.info;
                dupliquer(a1^.fg, a2^.fg);
                dupliquer(a1^.fd, a2^.fd) );
        end
   else a2 := nil;
end;
```

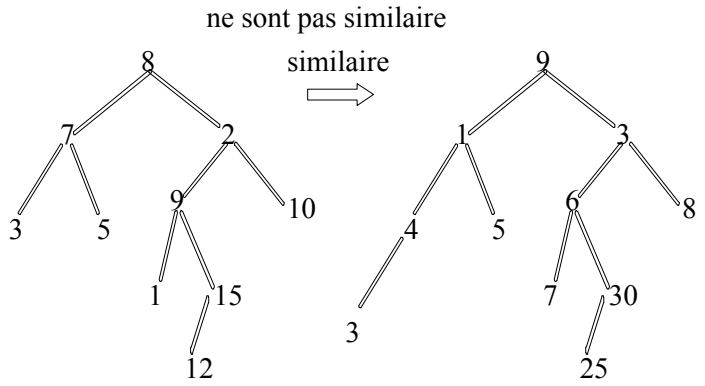
Arbre binaire

- *Une autre version*

```
function dupliquer (a1:Arbre ): Arbre
var a2: Arbre;
begin
   if (a1 <>nil)
   then begin
                new(a2);
                a2^.info := val;
                a2^.fg := dupliquer(a1^.fg);
                a2^.fd := dupliquer(a1^.fd);
                dupliquer:=a2;
        end
   else dupliquer:=nil;
end;
```

Arbre binaire

- ***Tester la similarité de deux arbres binaires***



Arbre binaire

- ***Arbres similaire***

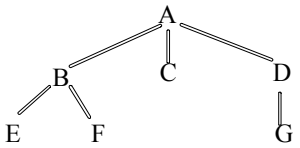
```
function similaire (a1, a2 : Arbre ):boolean;
begin
    if (a1 <>nil and  a2 <>nil)
     then similaire:= similaire (a1^.fg, a2^.fg) and similaire (a1^.fd, a2^.fd)
    else  if ( (a1=nil and a2<>nil) or  (a1<>nil and a2=nil)  )
          then similaire:= false
          else similaire:= true;
end;
```

Exercices

- Calculer la hauteur d ’un arbre binaire
- Tester si un arbre binaire est dégénéré
- Calculer le nombre d ’occurrences d ’une valeur donnée
- …

Arbre n-aire

- ***lorsqu ’un arbre admet, pour chaque nœud, au plus n fils, l’arbre est dit n-aire***



Arbre 3-aire (arbre ternaire)

Arbre n-aire

- ***Représentation***
  - ***première solution :***

    on définit un nœud comme une structures à plusieurs champs :

    info, fils1, fils2, …, filsn

    ☞ ***irréaliste (cas où il y a de nombreux fils absents***
  - ***Deuxième solution :***

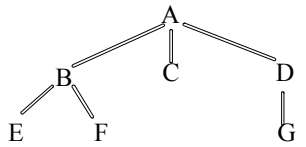
    Se ramener aux arbres binaires avec trois informations : info, successeur (ou fils ainé) et alternant (frère)

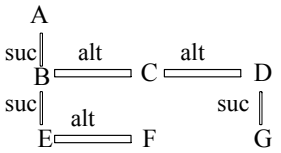
    ☞ ***Solution préférée en général***

Arbre n-aire

De l'exemple suivant :

On obtient :





Arbre n-aire

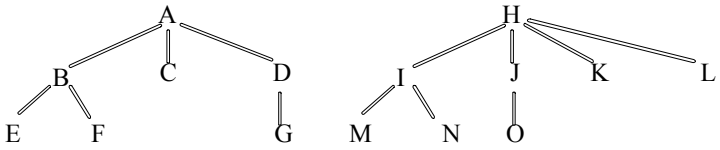
Représentation interne

```
type Arbre = ^ nœud;
    nœud = record
             info : typeElement;
             suc, alt : Arbre;
             end;

                    OU
  nœud = record
             info : typeElement;
             fils, frere : Arbre;
         end;
```

Forêt

Définition : on appelle une forêt un ensemble d'arbres n-aires

Exemple :

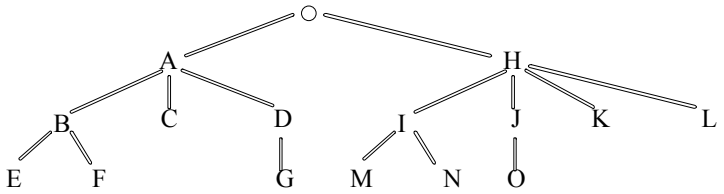


Forêt

Représentation

on se ramène aux arbres n-aires en créant une racine virtuelle permettant « d' enraciner » tous les arbres n-aires de la forêt.

Pour l'exemple précédent, on obtient :



Forêt

Algorithmes traitant des forêts

Comme pour les arbres binaires, on peut définir des parcours infixés, préfixés ou postfixés sur la représentation équivalentes de la forêt, etc.

Application :

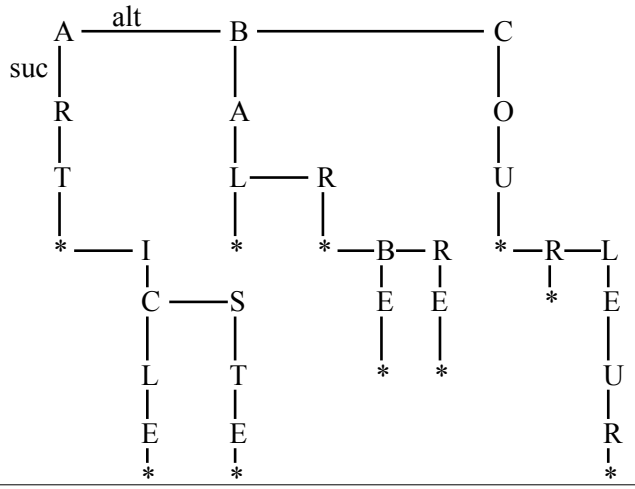
Gestion d'un dictionnaire arborescent

Très utilisée pour manipuler de gros dico en minimisant la place occupée.

Exemple :

ART	**BAL**	**COU**
ARTICLE	**BAR**	**COUR**
ARTISTE	**BARBE**	**COULEUR**

Forêt



Informations

- Cours : dernière séance - mardi 6 janvier
- TD :
  - G 2 : Vendredi 9 , 15h - 16h15 (Salle E06), Rattrapage
  - pas TD la semaine du 12 janvier
- TP : semaine du 12 janvier :
  - 30 mn pour rendre votre TP
  - 1H30 Contrôle TP

Consignes pour la remise du projet Laby

- **Listing du programme:**
- Pour chaque procédure ou fonction, on veut trouver au moins les informations suivantes sous l'entête :
  - *Entrées* : quelles sont-elles, quelles sont leurs propriétés ...
  - *Sorties* : quelles sont-elles, quelles sont leurs propriétés ...
  - *Spécification* : Une description (en utilisant par exemple le langage naturel) de ce que la fonction est sensé réaliser.
  - etc.

Consignes pour la remise du projet Laby

- Il faut éviter la programmation par effet de bord.
- Pour l'écriture du code source, il faut également tenir compte des conseils suivants :
  - *Commentaires (pertinents) dans le listing*
    - ***dans les déclarations (à quoi sert une variable)***
    - ***dans le code : présentation d'un sous-traitement ...***
  - *Sur l'écriture du code (quelques règles)*
    - ***garder une même unité de présentation***
    - ***noms évocateurs pour les identificateurs***
    - ***une procédure ou une fonction ne doit pas excéder une page***
    - ***ne pas oublier l'indentation***

Exemple de barème : TP1

- Programme source :
  - commentaires
  - types et variables (structures de données)
  - indentation
  - procédure et fonctions
  - algorithmique
- Exécution
  - Interface
  - correctes?

Le TP 1 est annulé

Correction du contrôle

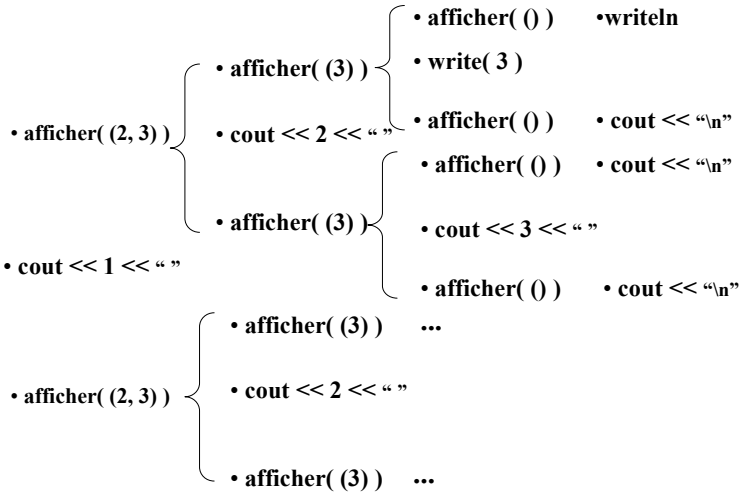
- Exercice 1:

Soit le type liste simplement chaînée codant une suite d'entiers .
Qu'affiche la procédure suivante pour une liste codant la suite (1,2,3)?

```
procedure afficher ( a: Liste )
begin
    if (a <>nil)
    then begin
                    afficher (a^.suivant) ;
                    write(a^.info);
                    afficher (a^.suivant) ;
         end
    else writeln;
end;
```

Correction du contrôle

afficher ((1, 2, 3)) ?



Correction du contrôle

- **Affichage :**

3
2
3
1
3
2
3

Correction du contrôle

- **Exercice 2 :**
  - **Définir une liste chaînée codant une suite d'entiers**

```
Type  Liste = ^cellule
      cellule = record
                    info:integer;
                    suivant: Liste;
                end;
```

Correction du contrôle

- **Ecrire les procédures ou fonctions suivantes :**
  - **Vérification qu'une liste est triée par ordre croissant.**

Version récursive

```
function estTrié(a:Liste):boolean;
begin
 if (a=nil)                          if (a=nil or a^.suivant =nil)
 then estTrié:= true                 then estTrié:= true;
 else if (a^.suivant =nil)
    then estTrié:= true
    else if(a^.info <= a^.suivant^.info)
          then estTrié:= estTrié(a^.suivant)
          else estTrié:= false;
end;
```

Correction du contrôle

- **Version itérative**

```
function estTrié(a:Liste):boolean;
var trié:boolean;
begin
 if (a=nil or a^.suivant =nil )
 then estTrié:= true
 else begin
          trié := true;
          while (a^.suivant<>nil and trié) do
          begin
             trié := (a^.info<=a^.suivant^.info);
            a := a^.suivant;
          end;
          estTrié:= trié;
       end;
end;
```

Correction du contrôle

- **Version itérative**

```
procédure insererListeTriee(x: integer; var a:Liste);
var p:Liste;
     trouve : boolean;
begin
 p:=a; prec:=nil;
 while (a<>nil and x>p^.info) do
 begin
   prec:=p; p := p^.suivant;
 end;
 if (prec=nil)
 then insertete(x, a)
 else insertete(x,prec^.suivant);
end;
```

Correction du contrôle

- **Compter le nombre d'entiers compris dans un intervalle donné [binf, bsup] (version récursive et itérative)**

Version récursive

```
function compter(a:Liste, binf, bsup:integer):integer;
begin
 if (a=nil)
 then compter:= 0
 else if (a^.info >= binf and a^.info <= bsup)
       then compter:= 1 + compter (a^.suivant, binf,bsup)
       else compter:= compter (a^.suivant, binf,bsup) ;
end;
```

Correction du contrôle

- **Compter le nombre d'entiers compris dans un intervalle donné [binf, bsup] (version récursive et itérative)**

Version itérative

```
function compter(a :Liste, binf, bsup:integer):integer;
var nb:integer;
  nb =0;
  while(a<>nil) do
  begin
        if (a^.info >= binf and a^.info <= bsup)
        then nb:=nb+1;
        a := a^.suivant;
  end;
 compter:= nb;
end;
```

Correction du contrôle

- **Suppression de la dernière occurrence d'une valeur donnée dans une liste.**
- **Version itérative**

```
procedure suppDerOcc(var a: Liste, val:integer)
var p, prec, precDer: Liste
begin
  p :=a, prec :=nil, precDer :=nil;
  while(p<>nil) do
  begin
        if (p^.info =val) then precDer := prec;
        prec := p;
        p := p^.suivant;
  end;
  if (precDer<>nil) then supptete(precDer^.suivant)
  else if (a<>nil and a^.info =val) then supptete(a);
end;
```

Correction du contrôle

- **Suppression de la dernière occurrence d'une valeur donnée dans une liste.**
- **Version récursive**

```
procedure suppDerOcc(var a:Liste, val:integer, var fait:boolean)
begin
  if (a<>nil)
  then begin
        suppDerOcc(a^.suivant, val, fait);
        if (not(fait) and a^.info =val)
        then begin
              supptete(a);
              fait := true;
            end;
  end;
end;
```

Appel : fait = false

Correction du contrôle

- **Ajouter un élément avant la première occurrence d'une valeur donnée**
- **version récursive**

```
procedure ajoutAvOcc(var a:Liste, val, occ:integer)
begin
  if (a<>nil)
  then if (a^.info =occ)
        then insertete(a, val)
        else ajoutAvOcc(a^.suivant, val,occ);
end;
```

Correction du contrôle

- **Ajouter un élément avant la première occurrence d'une valeur donnée**
- **version itérative**

```
procedure ajoutAvOcc(var a:Liste, val, occ:integer)
var  p,prec: Liste;
     trouve:boolean;
begin
  p := a; prec := nil;
  trouve := false;
  while(p<>nil and not trouve)do
  begin
        trouve := (p^.info =occ);
        if (not trouve) then begin  prec := p; p:=p^.suivant;end;
  end;
  if (trouve)
  then if (prec =nil) then insertete(a, val)
       else insertete(prec^.suivant, val);
end;
```

TP de synthèse

Objectifs

- Synthèse des différentes parties du cours de structures de données : chaînes de caractères, fichiers, listes et arbres.
- Réalisation et présentation d'un dossier d'analyse.
- Présentation d'un dossier de programmation

TP de synthèse

Sujet du TP

- Il s'agit d'écrire un programme qui étant donnée un fichier texte (par exemple un programme C++) l'affiche en numérotant les lignes. Puis affiche la liste de ses identificateurs, classée par ordre alphabétique, chaque identificateur étant suivi de la liste des numéros de lignes où il apparaît.
- La syntaxe d'un identificateur est la suivante :

  <identificateur> := <lettre>{<lettre> | <chiffre> | '_' }
  <lettre>:= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'
  <chiffre>:= '0' | '1' | ... | '9'

TP de synthèse

Sujet du TP

- Les identificateurs seront rangés dès leur rencontre dans un arbre binaire. Cet arbre devra être construit de telle sorte que, pour tout nœud portant un identificateur <id>, tous les nœuds se trouvant dans le sous-arbre gauche portent des identificateurs situés avant <id> dans l'ordre alphabétique; de même tous les nœuds se trouvant dans le sous-arbre droit portent des identificateurs situés après <id>. A chacun des identificateurs sera associé la liste des numéros de lignes où ils sont apparus.
- La liste des numéros de lignes où chaque identificateur apparaît sera donc traité aussi au moyen de pointeurs. Pour gagner du temps, chaque nouvelle apparition pourra être insérée en tête de liste; il ne faudra pas oublier alors à l'impression que les références se trouvent en ordre inverse.
- Le langage de programmation à utiliser est le C++ (sous Linux). Le fichier source est à mettre dans le répertoire **PROJETSD** avant le vendredi 25 mai 2001.

Exemple

Soit un fichier texte contenant les lignes suivantes :

```
Definition recursive d'une liste:

une liste est:
- soit une liste vide
- soit compose d'un element, suivi d'une liste

Definition recursive d'un arbre binaire:

un arbre binaire est :
- soit vide
- soit compose d'un noeud, du sous_arbre gauche et du
  sous_arbre droit
```

Exemple

```
Première partie : Numerotation du fichier
1 : Definition recursive d'une liste:
2 :
3 : une liste est:
4 : - soit une liste vide
5 : - soit compose d'un element, suivi d'une liste
6 :
7 : Definition recursive d'un arbre binaire:
8 :
9 : un arbre binaire est :
10 : - soit vide
11 : - soit compose d'un noeud, du sous_arbre gauche
       et du sous_arbre droit
12 :
```

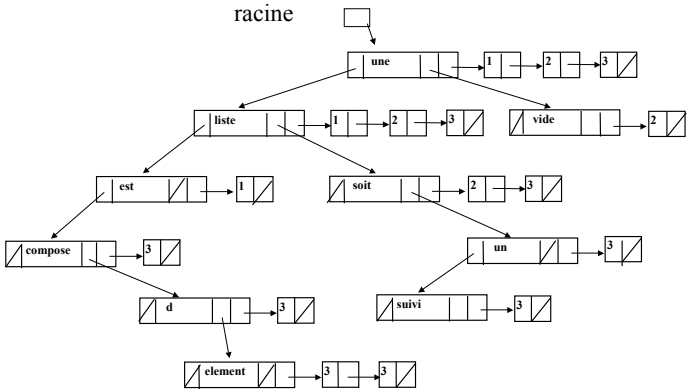
Exemple

Seconde partie : Traitement des identificateurs

Definition : 1 7	**arbre** : 7 9
binaire : 7 9	**compose** : 5 11
d : 1 5 5 7 11	**droit** : 11
du : 11 11	**element** : 5
est : 3 9	**et** : 11
gauche : 11	**liste** : 1 3 4 5
noeud : 11	**recursive** : 1 7
soit : 4 5 10 11	**sous_arbre** : 11 11
suivi : 5	**un** : 5 7 9 11
une : 1 3 4 5	**vide** : 4 10

Représentation

“ une liste est :
- soit une liste vide,
- soit composé d’un élément, suivi d’une liste ”



Tables

Méthodes de hachage

1. **Introduction**
2. **Hachage ouvert**
3. **Hachage fermé**
4. **Implémentation des fonctions**

Introduction

Recherche dichotomique

table	3	4	8	9	10	20	40	50	70	75	80	83	85	90

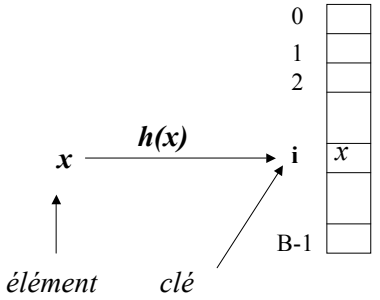
4 ?

```
bool  ELEMENT (int x, Table T, int d, int f )
{   int m;
    if  (d > f) {
        return false;
     } else {
        m  = (d+f ) / 2 ;
        if (x==T[m]) return true;
        else if  (x > T [i])
                return  ELEMENT (x, T, m+1, f );
             else  return  ELEMENT (x, T, d, m) ;
       }
}
```

Introduction

Idée :

Établir une relation entre un élément et la case à laquelle il est rangé dans le tableau



Hachage

Type abstrait « dictionnaire »
- Ensembles avec les opérations principales
  - ELEMENT (*x, A*)
  - AJOUTER (*x, A*)
  - ENLEVER (*x, A*)

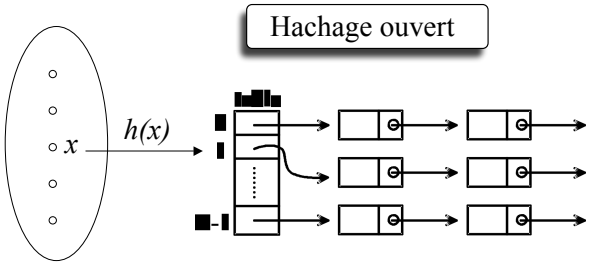
Table de hachage
- table dont les indices sont dans [0 .. G-1]

Fonction de hachage
- *h* : éléments → [0 .. B-1] non injective en général

Résolution des collisions
- Hachage ouvert : avec listes, triées ou non.
- Hachage fermé : linéaire, quadratique, aléatoire, uniforme, double,

...

Tables



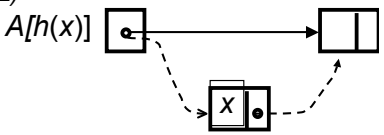

```
int h (string x)
{ int somme = 0 ;
  for (int i=0; i<x.length(); i++)
       somme = somme + x.charAt(i) ;
  return (somme % B) ;
fin
```

Hachage ouvert : implémentation

```
const int B = ... ;
typedef struct cel  {
                        string mot ;
                        struct cel * suivant;
} cellule;
typedef cellule * Liste;
typedef Liste DICO[B];

void  init (Table T)
{  for (int i=0; i<B-1;i++)  T[i] = NULL; }

bool  appartient (string x,  DICO A)
{  Liste p = A [ h(x) ] ;
   while ( p != NULL)
      if (p->elt == x)  return true;
      else p = p->suivant ;
   return  false ;
}
```

Hachage ouvert : implémentation

```
void ajouter ( string x,  DICO A)
{
    ajouterListe ( x, A[h(x)] ) ;
}


void ajouterListe (string x,  Liste & L)
{
        if  (!appartient(x, L)) {
                p = L ;
                L = new cellule  ;
                L->mot =  x ;
                L->suivant  =  p ;
        }
}
```



Introduction

<u>De l'analyse du problème à l'exécution du programme</u>

1. Bien lire l 'énoncé du problème, être certain de bien le comprendre.
2. Réfléchir au problème, déterminer les points principaux à traiter.
3. Trouver un bon algorithme de résolution, l'écrire sur papier en français.
4. Coder l'algorithme en un programme écrit sur papier
5. Introduire le programme dans l'ordinateur au moyen d'un éditeur.
6. Vérifier la syntaxe du programme au moyen d 'un vérificateur de syntaxe.
7. Compiler le programme
8. Exécuter le programme, vérifier son bon fonctionnement par des tests.

Revision (TD)

Problème :

- On désire créer un programme permettant la manipulation d'ensemble d'entiers.

  a) Après avoir préalablement défini le type ensemble d'entiers(utilisation de tableau). Écrire les procédures ou fonctions de manipulations (opérateurs) d'ensembles décrites ci-dessous:

TD (Révision)

- |E| ? retourne la cardinalité (le nombre d'éléments) d'un ensemble E.
- E = Ø? retourne un booléen qui indique si E est l'ensemble vide ou non.
- x ∈ E? retourne un booléen indiquant si un élément x appartient à un ensemble E.
- E1 ∩ E2 retourne l'intersection des ensembles E1 et E2. E3 = { x | x∈ E1 ∧ x ∈ E2}.
- E1 ∪ E2 retourne l'union des ensembles E1 et E2, E3 = { x | x∈ E1 ∨ x ∈ E2}.
- E1 \ E2 retourne les éléments de E1 qui ne sont pas dans E2, appelé la différence de E1 et E2. Si E3 = E1 \ E2 alors E3 = { x | x∈ E1 ∧ x ∉ E2}.
- E2 ← E1 + x ≡ E2 = E1 ∪ {x}
- E2 ← E1 - x ≡ E2 = E1 \ {x}
- E1 = E2? retourne un booléen qui indique si les 2 ensembles contiennent exactement les mêmes éléments
- E1 ⊂ E2? indique si les éléments de E1 sont inclus dans E2, sans que E1 = E2. E1 ⊂ E2 si ∀x ∈ E1 x ∈ E2 et E1 ≠ E2.

TD (Révision)

```
Const Max = 100;
type ensemble = record
                  nbElement : integer;
                  element: array[1..Max] of integer;
                end;
```

- *|E| ? retourne la cardinalité (le nombre d'éléments) d'un ensemble E.*

```
Function cardinal(e:ensemble) : integer;
begin
    cardinal := e.nbElement;
end;
```

- *E = Ø? retourne un booléen qui indique si E est l'ensemble vide ou non.*

```
Function estVide(e:ensemble) : boolean;
begin
    estVide := (e.nbElement = 0);
end;
```

TD (Révision)

- *x ∈ E? retourne un booléen indiquant si un élément x appartient à un ensemble E.*

```
Function appartient(x:integer; e:ensemble) : boolean;
var trouve:boolean;
    i:integer;
begin
   trouve:=false; i:=1;
   while (i<=cardinal(e) and (not trouve) )
   do begin
            trouve:=(e.element[i]=x);
            i:=i+1;
      end;
   appartient:=trouve;
end;
```

TD (Révision)

- *E1 ∩ E2 retourne l'intersection des ensembles E1 et E2.*
  E3 = { x | x∈E1 ∧ x ∈E2}.

```
procedure intersection(e1:ensemble; e2:ensemble;
                       var e3:ensemble);
var i:integer;
begin
   e3.nbElement:=0;
   for i:=1 to cardinal(e1)
   do if(appartient(e1.element[i], e2) )
      then ajout(e1.element[i], e3);
end;
```

- *E1 ∪ E2 retourne l'union des ensembles E1 et E2, E3 = { x | x∈E1 ∨ x ∈E2}.*

```
procedure union(e1:ensemble; e2:ensemble;
                var e3:ensemble);
var i:integer;
begin
   copier(e1,e3);
   for i:=1 to cardinal(e2)
   do if(not appartient(e2.element[i], e1) )
      then ajout(e2.element[i], e3);
end;
```

TD (Révision)

E1 \ E2 retourne les éléments de E1 qui ne sont pas dans E2, appelé la différence de E1 et E2. Si E3 = E1 \ E2 alors E3 = { x | x∈E1 ∧ x ∉E2}.

```
procedure différence(e1:ensemble; e2:ensemble;
                        var e3:ensemble);
var i:integer;
begin
   e3.nbElement:=0;
   for i:=1 to cardinal(e1)
   do if(not appartient(e1.element[i], e2) )
      then ajout(e1.element[i], e3);
end;
```

E1 Δ E2 retourne la différence symétrique de E1 et E2.
E3 = {E1 \ E2} ∪ {E2 \ E1}.

```
procedure différenceSym(e1:ensemble; e2:ensemble;
                        var e3:ensemble);
var e4,e5:ensemble;
begin
   difference(e1,e2,e4); difference(e2,e1,e5);
   union(e4,e5,e3);
end;
```

TD (Révision)

E2 ← E1 + x ≡ E2 = E1 ∪ {x}

```
procedure ajout(x:integer; var e:ensemble);
var i:integer;
begin
   if(cardinal(e)<max)
   then begin
           e.nbElement:=e.nbElement+1;
           e.element[e.nbElement]:=x;
        end;
end;
```

TD (Révision)

E2 ← E1 - x ≡ E2 = E1 \ {x}

```
procedure supprimer(x:integer; var e:ensemble);
var i:integer;
begin
      i:=1;
      while(i<=cardinal(e)and x<>e.element[i])
      do i:=i+1;
      if(i<=cardinal(e))
      then begin
              e.element[i]:=e.element[cardinal(e)];
              e.nbElement:=e.nbelement-1;
           end;
end;
```

TD (Révision)

E1 ⊂ E2? indique si les éléments de E1 sont inclus dans E2, sans que E1 = E2. E1 ⊂ E2 si ∀x ∈E1 x ∈E2 et E1 ≠ E2.

```
function inclus(e1,e2:ensemble):boolean;
var i:integer;
    dans:boolean;
begin
   dans:=true; i:=1;
   while(i<=cardinal(e1)and dans )
   do begin
           dans := appartient(e1.element[i],e2);
           i:=i+1;
      end;
   inclus:=dans;
end;
function inclus(e1,e2:ensemble):boolean;
var e:ensemble;
begin
   difference(e1,e2,e);
   inclus:=vide(e);
end;
```

TD (Révision)

- *E1 = E2? retourne un booléen qui indique si les 2 ensembles contiennent exactement les mêmes éléments*

```
function egal(e1,e2:ensemble):boolean;
begin
    egal := inclus(e1,e2) and inclus(e2,e1);
end;
function egal(e1,e2:ensemble):boolean;
var e:ensemble;
begin
    differenceSym(e1,e2,e);
    egal := vide(e);
end;
```

- *E2 := E1 ; copier E1 dans E2.*

```
procedure copier(e1:ensemble; var e2:ensemble);
var i:integer;
begin
    e2.nbElement:=0;
    for i:=1 to cardinal(e1)
    do ajout(e1.element[i], e2);
end;
```

TD (Révision)

b) En supposant un ordre (croissant) sur les éléments de l'ensemble, apportez les modifications nécessaires aux opérations précédentes.

- *x ∈ E? retourne un booléen indiquant si un élément x appartient à un ensemble E.*

```
Function appartient(x:integer; e:ensemble) : boolean;
var trouve:boolean;
     i:integer;
begin
    trouve:=false;i:=1;
    while (i<=cardinal(e) and x>e.element[i] )
    do i:=i+1;

    if i<=cardinal(e)
    then trouve:=(x=e.element[i]);
    appartient:=trouve;
end;
```

TD (Révision)

b) En supposant un ordre (croissant) sur les éléments de l'ensemble, apportez les modifications nécessaires aux opérations précédentes.

- *E2 ← E1 + x ≡ E2 = E1 ∪ {x}*

```
procedure ajout(x:integer; var e:ensemble);
var i:integer;
begin
    if(cardinal(e)<max)
    then begin
            pos:=positionInsertion(x,e);
            decalerDroite(pos,e);
            e.element[pos]:=x;
            e.nbElement:=e.nbElement+1;
         end;
end;
```

TD (Révision)

b) En supposant un ordre (croissant) sur les éléments de l'ensemble, apportez les modifications nécessaires aux opérations précédentes.

```
function positionInsertion(x:integer; e:ensemble):integer;
  var i:integer;
begin
     i:=1;
     while (i<=cardinal(e) and x >e.element[i])
     do i:=i+1;
     positionInsertion:=i;
end;
procedure decalerDroite(pos:integer; var e:ensemble);
  var i:integer;
begin
     for i:=cardinal(e) downto pos
     do e.element[i+1]:=e.element[i];
end;
```

TD (Révision)

b) En supposant un ordre (croissant) sur les éléments de l'ensemble, apportez les modifications nécessaires aux opérations précédentes.

- *E2 ← E1 - x ≡ E2 = E1 \ {x}*

```
procedure supprimer(x:integer; var e:ensemble);
var i:integer;
begin
    while(i<=cardinal(e)and x>e.element[i])
    do i:=i+1;
    if(i<=cardinal(e) and e.element[i]=x)
    then begin
            decalerGauche(i,e);
            e.nbElement:=e.nbElement-1;
         end;
end;
procedure decalerGauche(pos:integer; var e:ensemble);
  var i:integer;
begin
     for i:= pos to cardinal(e)-1
     do e.element[i]:=e.element[i+1];
end;
```

TD (Révision)

c) donnez les versions récursives pour les fonctions ou procédures 3, 4, 5, 9 et 10

- *x ∈ E? retourne un booléen indiquant si un élément x appartient à un ensemble E.*

```
Function appartient(x:integer; e:ensemble;
                    i:integer):boolean;
begin
    if(i>cardinal(e)
    then appartient:=false
    else appartient:=(x=e.element[i])
         or appartient(x,e,i+1);
end;

Appel : appartient(x,e,1);
```

TD (Révision)

```
type  Liste = ^cellule
     cellule = record
                 val: integer;
                 suivant : Liste;
               end;
     ensemble = record
                 nbElement : integer;
                 element: Liste;
               end;
```

TD (révision)

- *|E| ? retourne la cardinalité (le nombre d'éléments) d'un ensemble E.*

```
Function cardinal(e:ensemble) : integer;
begin
    cardinal := e.nbElement;
end;
```

- *E = Ø? retourne un booléen qui indique si E est l'ensemble vide ou non.*

```
Function estVide(e:ensemble) : boolean;
begin
    estVide := (e.nbElement = 0);
end;
```

TD (Révision)

x ∈ E? retourne un booléen indiquant si un élément x appartient à un ensemble E.

```
Function appartient(x:integer; e:ensemble) : boolean;
var trouve:boolean;
    p:Liste;
begin
   trouve:=false; p:=e.element;
   while (p<>nil and (not trouve) )
   do begin
           trouve:=(p^.val=x);
           p:=p^.suivant;
      end;
   appartient:=trouve;
end;
```

TD (Révision)

E1 ∩ E2 retourne l'intersection des ensembles E1 et E2.
E3 = { x | x∈E1 ∧ x ∈E2}.

```
procedure intersection(e1:ensemble; e2:ensemble;
                       var e3:ensemble);
var p:Liste;
begin
   e3.nbElement:=0; e3.element:=nil;p:=e1.element;
   while( p<>nil)
   do if(appartient(p.val, e2) )
      then ajout(p.val e3);
end;
```

E1 ∪ E2 retourne l'union des ensembles E1 et E2, E3 = { x | x∈E1 ∨ x ∈E2}.

```
procedure union(e1:ensemble; e2:ensemble;
                var e3:ensemble);
var p:Liste;
begin
   copier(e1,e3); p:=e2.element;
   while(p<>nil)
   do if(not appartient(p^.val, e1) )
      then ajout(p^.val, e3);
end;
```

TD (Révision)

E1 \ E2 retourne les éléments de E1 qui ne sont pas dans E2, appelé la différence de E1 et E2. Si E3 = E1 \ E2 alors E3 = { x | x∈E1 ∧ x ∉E2}.

```
procedure différence(e1:ensemble; e2:ensemble;
                     var e3:ensemble);
var p:Liste;
begin
   e3.nbElement:=0;e3.element:=nil;p:=e1.element;
   while(p<>nil)
   do if(not appartient(p^.val e2) )
      then ajout(p^.val e3);
end;
```

E1 Δ E2 retourne la différence symétrique de E1 et E2.
E3 = {E1 \ E2} ∪{E2 \ E1}.

```
procedure différenceSym(e1:ensemble; e2:ensemble;
                        var e3:ensemble);
var e4,e5:ensemble;
begin
   difference(e1,e2,e4); difference(e2,e1,e5);
   union(e4,e5,e3);
end;
```

TD (Révision)

E2 ← E1 + x ≡ E2 = E1 ∪ {x}

```
procedure ajout(x:integer; var e:ensemble);
var p:Liste;
begin
      if(not appartient(x,e))
      then begin
              e.nbElement:=e.nbElement+1;
              insertete(x, e.element);
           end;

end;
```

TD (Révision)

- *E2 ← E1 - x ■ E2 = E1 \ {x}*

```
procedure supprimer(x:integer; var e:ensemble);
var p, prec:Liste
begin
       p:=e.element; prec:=nil;
       while(p<>nil and x<>p.val)
       do begin pres:=p; p:=p^.suivant; end;
       if(p<>nil)
       then begin
              e.nbElement:=e.nbElement-1;
              if ( prec=nil)
              then supptete(e.element)
              else supptete(prec^.suivant);
            end;
end;
```

TD (Révision)

- *E1 ⊂ E2? indique si les éléments de E1 sont inclus dans E2, sans que E1 = E2. E1 ⊂ E2 si ∀x ∈ E1 x ∈ E2 et E1 ≠ E2.*

```
procedure inclus(e1,e2:ensemble):boolean;
var p:Liste;
    dans:boolean;
begin
   dans:=true; p:=e1.element;
   while(p<>niland dans )
   do begin
           dans := appartient(p^.val,e2);
           p:=p^.suivant;
      end;
   inclus:=dans;
end;
procedure inclus(e1,e2:ensemble):boolean;
var e:ensemble;
begin
   difference(e1,e2,e);
   inclus:=vide(e);
end;
```

TD (Révision)

- *E1 = E2? retourne un booléen qui indique si les 2 ensembles contiennent exactement les mêmes éléments*

```
procedure egal(e1,e2:ensemble):boolean;
begin
    egal := inclus(e1,e2) and inclus(e2,e1);
end;
procedure egal(e1,e2:ensemble):boolean;
var e:ensemble;
begin
    differenceSym(e1,e2,e);
    egal := vide(e);
end;
```

- *E2 := E1 ; copier E1 dans E2.*

```
procedure copier(e1:ensemble; var e2:ensemble);
var p:Liste;
begin
    e2.nbElement:=0; e2.element:=nil;p:=e1.element;
    while (p<>nil)
    do begin insertete(p^.val, e2);p:=p^.suivant; end;
end;
```

TD (Révision)

b) En supposant un ordre (croissant) sur les éléments de l'ensemble, apportez les modifications nécessaires aux opérations précédentes.

- *x ∈ E? retourne un booléen indiquant si un élément x appartient à un ensemble E.*

```
Function appartient(x:integer; e:ensemble) : boolean;
var trouve:boolean;
    p:Liste;
begin
    trouve:=false;p:=e.element;
    while (p<>nil and x>p^.val )
    do p:=p^.suivant;

    if p<>nil
    then trouve:=(x=p^.val)
    appartient:=trouve;
end;
```

TD (Révision)

b) En supposant un ordre (croissant) sur les éléments de l'ensemble, apportez les modifications nécessaires aux opérations précédentes.

- *E2 ← E1 + x ≡ E2 = E1 ∪ {x}*

```
procedure ajout(x:integer; var e:ensemble);
var p,pos:Liste;
begin
    if(not appartient(x,e))
    then begin
            pos:=positionInsertion(x,e);
            if (pos=nil)
            then insertete(x,e.element)
            else insertete(x,pos^.suivant);
            e.nbElement:=e.nbElement+1;
         end;
end;
```

TD (Révision)

b) En supposant un ordre (croissant) sur les éléments de l'ensemble, apportez les modifications nécessaires aux opérations précédentes.

```
function positionInsertion(x:integer; e:ensemble):integer;
  var p, prec:Liste;
begin
      p:=e.element;prec:=nil;
      while (p<>nil and x >p^.val
      do begin
              prec:=p; p:=p^.suivant;
         end;
      positionInsertion:=prec;
end;
```

TD (Révision)

b) En supposant un ordre (croissant) sur les éléments de l'ensemble, apportez les modifications nécessaires aux opérations précédentes.

- *E2 ← E1 - x ≡ E2 = E1 \ {x}*

```
procedure supprimer(x:integer; var e:ensemble);
var p,prec:Liste;
begin
    p:=e.element;prec:=nil;
    while(p<>nil and x>p^.val)
    do begin
            prec:=p;
            p:=p^.suivant;
       end;
    if(p<>nil and p^.val=x)
    then if (prec=nil)
         then supptete(e.element)
         else supptete(prec^.suivant);
end;
```

TD (Révision)

c) donnez les versions récursives pour les fonctions ou procédures 3, 4, 5, 9 et 10

- *x ∈ E? retourne un booléen indiquant si un élément x appartient à un ensemble E.*

```
Function appartient(x:integer; e:ensemble):boolean;


begin
    if(e.element=nil)
    then appartient:=false
    else if(x=e.element^.val)
         then appartient:=true
         else begin
              e.element:=e.element^.suivant;
              appartient:=appartient(x,e);
              end;
end;
```

Projet de TP

- **Jeu du Labyrinthe**

On se propose de **programmer un jeu** (interactif) qui doit permettre tout d'abord de créer un **labyrinthe** graphiquement, puis à un utilisateur (le joueur) de se déplacer dans ce labyrinthe avec comme objectif d'en trouver la sortie.

Le programme se décompose en **deux parties** distinctes que nous détaillons :

- **Création du labyrinthe**
- **Jeu avec le labyrinthe**

Projet de TP

- **Remarques Importantes :**

- Evitez de passer du temps dans les aspects graphiques, limitez vous aux stricte nécessaire. Nous vous rappelons que ce qui sera primordial pour la notation, c'est bon fonctionnement du jeu, et non sa beauté de la grille etc.
- Il est vivement conseillé, d'adopter la démarche de résolution de problèmes (voir cours et TD). Dans une première étape, on analyse et on ne programme pas !
- Ce projet est à réaliser en binôme, le programme doit être rendu la semaine du **05/01/2003.** La séance de cette dernière semaine sera décomposée en deux parties, la finalisation et la remise du projet ainsi que le contrôle de TP.

Projet de TP (Jeu Puissance 4)

- On se propose de programmer un jeu interactif; voici les règles que nous utiliserons :
- **Règles du jeu :**
- Ce jeu se déroule sur une grille LxC avec des pions blancs et des pions rouge. Ce jeu oppose 2 joueurs, l'un posant sur la grille les pions blancs, l'autre les pions rouge. Chacun des joueurs posera à son tour un pion sur la grille. Le premier pion déposé est blanc, et il est placé en bas de la grille. Tout nouveau pion déposé sur la grille doit l'être :
  - ❑ En bas de la grille(ligne 1) ou juste au-dessus d'un pion (blanc ou rouge) déjà déposé (s'il reste de la place) ; a priori il y a autant de possibilités qu'il y a de colonnes.
  - ❑ Sur une case vide (ceci restreint les possibilités précédentes)
- Le jeu s'arrête dès qu'un joueur a placé 3 pions à la file et en ligne horizontale, verticale ou diagonale. Ce joueur est déclaré vainqueur. Si la grille est remplie et qu'aucun joueur n'a gagné, la partie sera déclaré nulle (match nul).

Projet de TP (Jeu Puissance 4)

But du projet :

- Dans le programme à concevoir, il faut permettre à deux joueurs de jouer de manière interactive. Pour placer un pion, le joueur désignera sa case (x,y) en utilisant les flèches du clavier, en considérant "← " pour le déplacement vers la gauche, "↑" pour le déplacement vers le haut, etc. Un contrôle sera opéré sur la validité des coups du joueur.
- Ci-dessous les codes ASCII :
  - ❑"← " : aller à gauche (code = 75)
  - ❑" → " : aller à droite (code = 77)
  - ❑" ↑ " : monter (code = 72)
  - ❑" ↓ " : descendre (code= 80)
  - ❑" ↲ " : valider (code=13)

Projet de TP (Jeu Puissance 4)

- On utilisera la représentation de la grille (e.g. 7x6) et des pions proposée ci-dessous :

6						
5						
4						
3				B		
2	B		B	N		
1	N	B	N	N		
	1	**2**	**3**	**4**	**5**	**6**	**7**

- La taille de la grille sera définie en fonction des limites de l'écran de l'ordinateur que vous utiliserez .
- Les primitives graphiques nécessaires et autorisées, vous serons explicité en cours ou en TP.

Projet de TP

- **Un algorithme général :**

  Nous proposons un schéma d'algorithme qui permet de programmer ce problème. Vous n'êtes pas obligés de le respecter ; il vous est donnée pour vous montrer la démarche générale à adopter, et vous exprimer la faisabilité du projet.
- Début
  - {initialisation}
  - <initialisation des variables >
  - <initialisation de l'écran>
  - {jeu – les blancs commencent}
  - répéter
    - si (joueur=blanc)
    - alors début
      - » <lire action>
      - » <exécuter action>
      - » si(non <fin de partie>) alors joueur := noir
      - » fin
    - sinon début
      - » <lire action>
      - » <exécuter action>
      - » si(non <fin de partie>) alors joueur := blanc
      - » fin
  - jusqu'à (<fin de partie>)
  - <afficher le résultat>
- Fin.

Projet de TP

- Le programme doit permettre deux possibilités de jeux
  - ❑entre deux joueurs humains
  - ❑entre un joueur humain et la machine. Dans ce dernier cas, l'ordinateur utilisera la stratégie à courts termes que nous décrivons ci-dessous (Blanc désigne l'utilisateur et Rouge l'ordinateur) :
    - ❖Si Rouge peut réaliser une file (horizontale, verticale ou en diagonale) de 3 pions ; il la réalise
    - ❖Sinon si Blanc peut réaliser au coup suivant une file de 3 pions ; Rouge s'y oppose (en plaçant un pion à un bout). Ceci est parfois insuffisant pour éviter la défaite.
    - ❖Sinon si Rouge peut réaliser peut réaliser une file de 2 pions , il la réalise.
    - ❖Sinon mettre un pion rouge au dessus d'un pion noir ou en bas de la grille.

Projet de TP

- **Remarques importantes :**

  Il est vivement conseille, d'adopter la démarche de résolution de problèmes (vu en cours : voir exemple de tri).
- **Dans une première étape, on analyse et on ne programme pas !**