Algorithmique et Structures de données II

Chargé du cours : Dr. Ilhem Abdelhedi Abdelmoula

Ilhem.abdelhedi@cristal.rnu.tn

1 INF ING

ESTI - Ecole Supérieure des Technologies et de l'Informatique

Semestre : II **Année universitaire :2012-2013**

Chapitre 2 les pointeurs et les listes chaînées

Partie 1 : Les Pointeurs

Pourquoi les pointeurs ?

Pour utiliser les adresses mémoires des variables à la place des variables elles-mêmes

I – Pointeurs

- **Définition**
  - Une variable pointeur est une variable qui pointe sur une variable
  - Sa valeur est l'adresse de cette variable (variable pointée)

10000	2000	p pointeur contient l'adresse d'une autre case
20000	12	case correspondant à la valeur pointée

- **Utilisation**

  1 – Création de la variable

  2 – Initialisation de la variable pointeur et de la variable pointée

  3 – Utilisation de la variable pointée

Déclaration d’une variable Pointeur

Var

nom_pointeur : ↑ type_valeur_pointée
nom_pointeur : pointeur sur type_valeur_pointée

Ex. Var pentier : ↑entier
=>Crée une variable pointeur pentier de type pointeur sur entier qui pointera sur une variable de *type entier*

Var :

cpt : entier
pInt : ↑ *entier*
Txt : chaîne
Ptxt: ↑ *chaîne*

Accès au contenu de la variable pointée

- Utilisation de l'objet pointé
  - Nom_pointeur ↑ (de type_valeur_pointée)
  - Nom_pointeur ^
  - *Nom_pointeur

```
Var :
    var_i : entier
    pi : ↑entier
Début
    var_i ← 38
    Nouveau(pi)
    pi ← #var_i
    Écrire  pi ↑          'afficher 38
    pi ↑ ← pi ↑ +2
    Écrire pi ↑       'afficher 40
    Écrire  var_i     'afficher 40
```

Opérations sur les pointeurs

- Pas de lecture, écriture ou opérations sur les pointeurs
- Le type pointeur supporte
  - Les initialisations
    - Appel de nouveau
    - Affectation de NIL à un pointeur p ←NIL
    - Affectation de la valeur d'un pointeur à un autre
  - Les comparaisons
    - Test = et ≠ entre pointeurs de même type et entre un pointeur et nil

Pointeurs et allocation dynamique

- Dans certains cas, la taille de l'espace à allouer (taille d'un tableau) diffère selon le type de la variable
- On connait pas l'adresse
- Il est parfois nécessaire d'allouer de la mémoire dynamiquement
- Solution : Réserver un emplacement mémoire pour une donnée pointée directement

Pointeur et allocation dynamique

- Créer un pointeur sur un type de donnée (ex. entier)

  nouveau(Pointeur) ou allouer(Pointeur)

  Pointeur ← nouveau (type)

- Libérer la mémoire utilisée !!!

Libérer (Pointeur) ou disposer(Pointeur)

Pointeur ← NIL

Var :

pEntier : pointeur sur entier

x: entier

Début

pEntier ← nouveau (Entier) ou nouveau (pentier)

pEntier↑ ←12345

Écrire pEntier↑

……..

pEntier ← NIL

……

pEntier ←#x

x←5

Écrire pEntier↑ affiche 5

…….

Libérer (pEntier)

réserver un espace mémoire qui contiendra cet entier, sur lequel la variable pointeur pointera

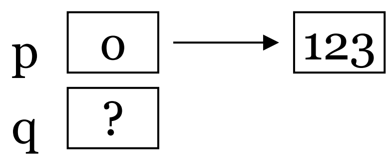
Exemple

Algorithme avec_des_pointeurs

var p,q:↑entier

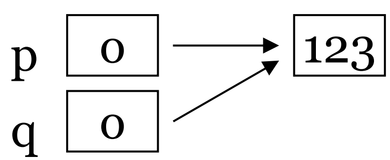
début

nouveau(p)



p↑ ← 123

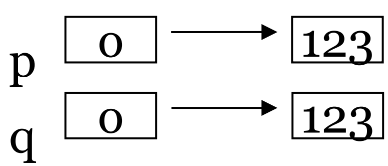
q ← p



l'adresse de p est recopiée dans q

nouveau(q)

q↑ ← p↑



affectation entre variables pointées

Pointeur sur enregistrement

Type tarticle = Enregistrement

ref : chaine

libellé: chaine

prix : réel

finEnregistrement

Var :

art : tarticle

part : ↑tarticle

Déclaration d'un pointeur de variable structurée

Début

art.ref ← « ref0122435 »

nouveau (part)

part← #art pointe sur l'adresse de la variable art

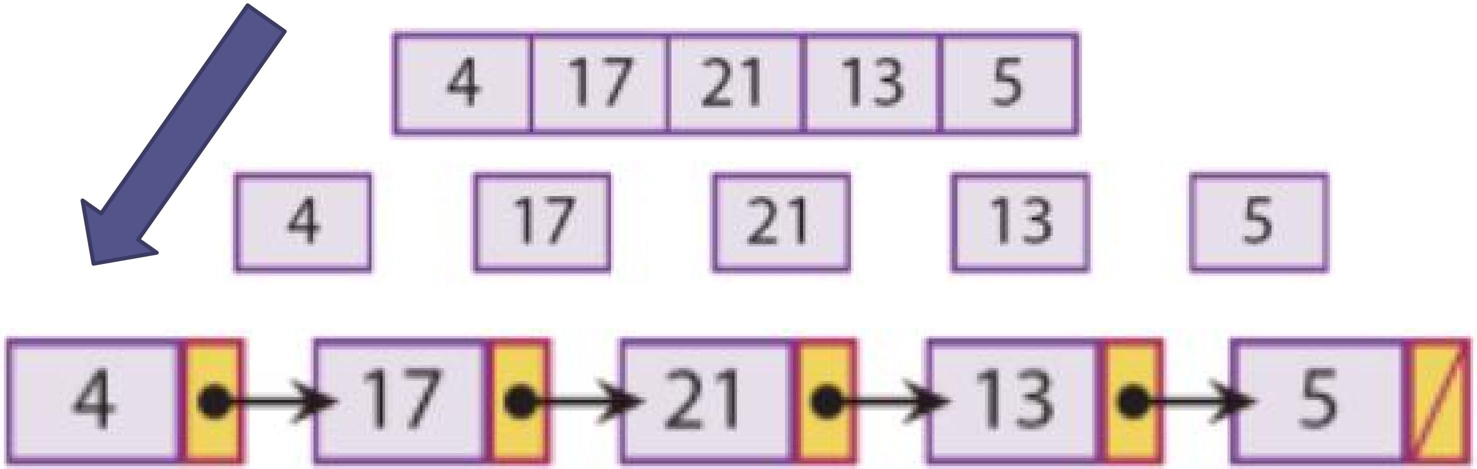
Écrire part →ref Accès aux champs de la structure

part ↑.ref ← « ref01235 »
ou
part →ref ← « ref01235 »

Partie 2 : les listes chaînées

Tableau vers liste chaînée

- Une liste chaînée est une structure de données dans laquelle les objets sont arrangés linéairement. L'ordre linéaire est déterminé par des pointeurs sur les éléments.
  - A la différence du tableau, les éléments n'ont aucune raison d'être contigus ni ordonnés en mémoire.

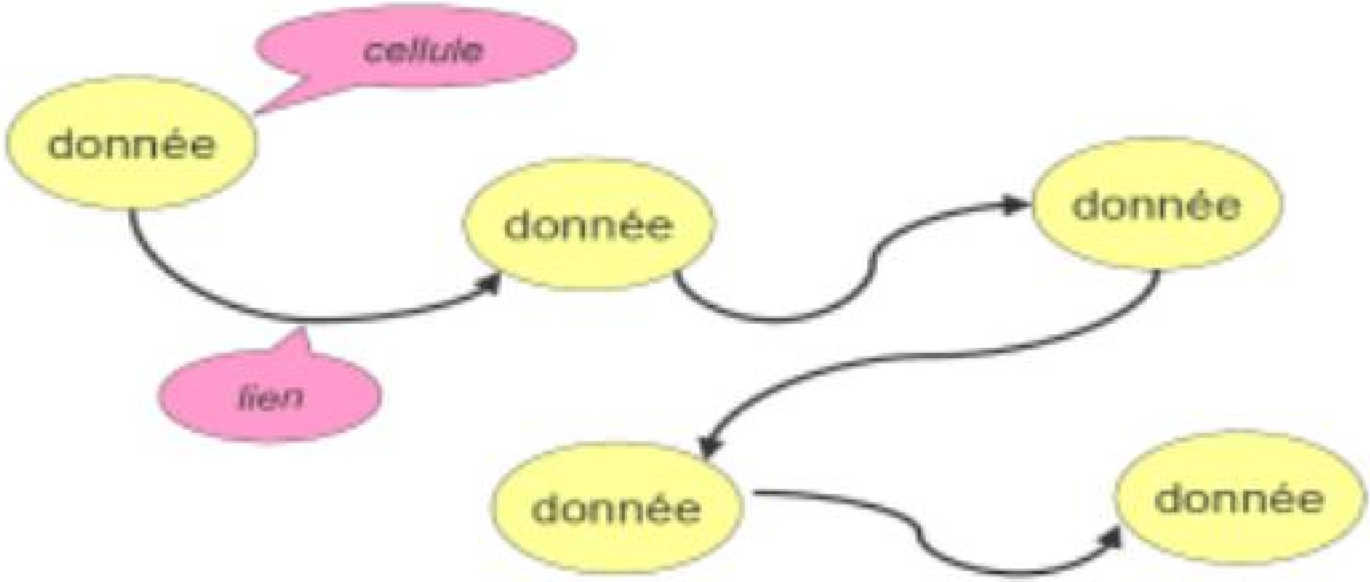


Les listes chaînées

- Une liste chainée est une structure de données, similaire aux tableaux, qui contient des éléments d’un même type.
- L’ajout et la suppression d’un élément se font de manière très rapide,
- En revanche, l’accès à un élément est un peu plus long que sur un tableau.
- Elle est dynamique : sa taille n’est pas figée et n’est pas limitée (contrairement aux tableaux).
- Elle repose essentiellement sur les pointeurs.

Description d’une liste chaînée

- C’est un ensemble de cellules ou maillons joint les uns aux autres par le biais de pointeurs



Représentations d'une liste linéaire

- Plusieurs représentations des listes linéaires ont été proposées.
- La plupart consistent à enregistrer chaque valeur dans une cellule de la mémoire et à chaîner ces cellules entre elles.
- Elles se différencient principalement par :
  - le mode de mémorisation des cellules : dans un tableau ou bien dans une zone mémoire allouée dynamiquement,
  - le mode de de marquage du début ou de la fin de la liste,
  - le mode de chaînage des cellules : unidirectionnel ou bidirectionnel.

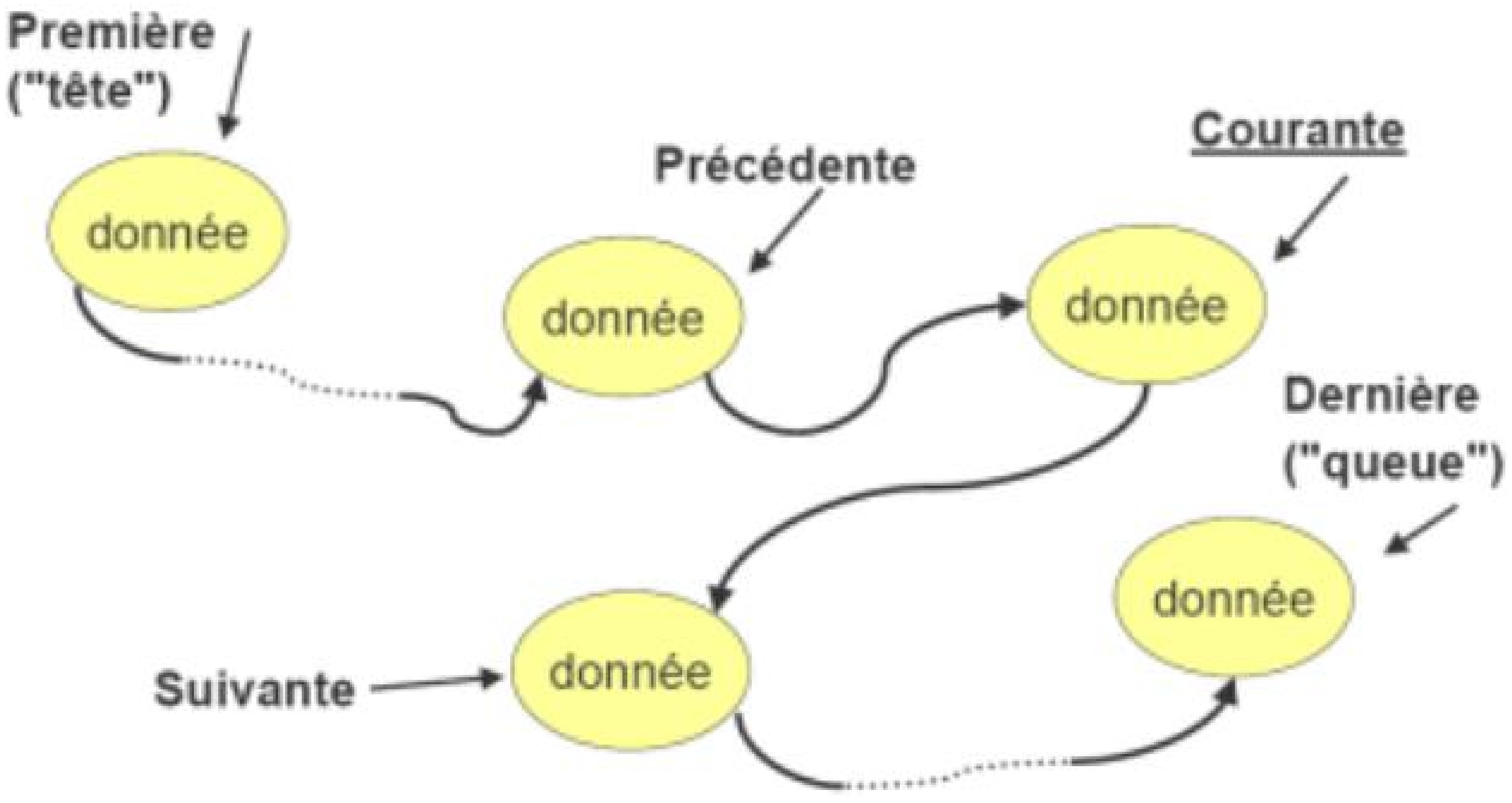
Définition d’une liste linéaire

- Une liste linéaire est une chaîne de **maillons composée :**
  - d’un **maillon de début,**
  - d’une suite éventuellement vide de **maillons internes,**
  - d’un **maillon de fin.**
- Chaque maillon a un **identifiant.**
- Le maillon de début contient l’identifiant du 1er maillon interne.
- Le *ième maillon interne contient la ième valeur de la liste linéaire et l’identifiant du maillon contenant la (i + 1)ème valeur*.
- Le maillon de fin a un identifiant nul.
- Le maillon suivant du maillon de début d’une **liste linéaire vide est le maillon de fin.**
- Une liste est identifiée par l'identifiant de son maillon de début

Définition de la classe Liste

- Les attributs de la classe Liste doivent permettre:
  - le positionnement sur les différents maillons de la liste
  - la définition du type d'information enregistrée dans un maillon
- Il faut rajouter trois pointeurs pour faciliter le repérage dans une liste chaînée:
  1. un pointeur premier qui pointe vers le premier maillon de la liste,
  2. un pointeur dernier qui pointe vers le dernier maillon
  3. un pointeur courant qui pointe sur un maillon quelconque de la liste

Repérage d’un maillon



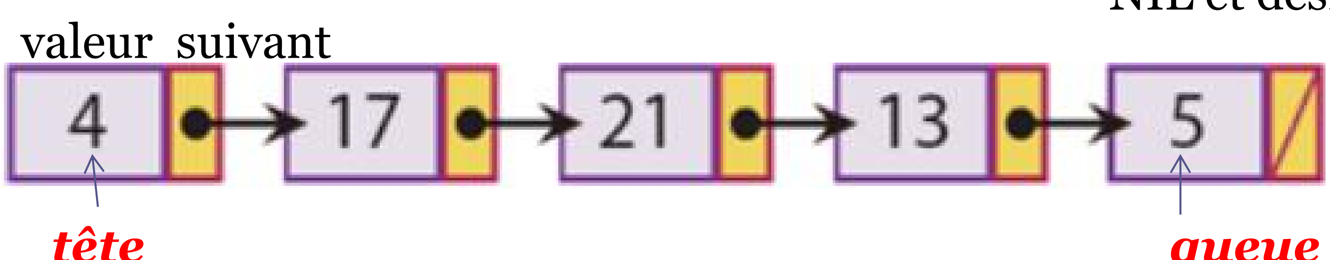
La structure d’un maillon

- Un maillon est constitué de :

1. La valeur à stocker dans le maillon
2. Un pointeur *suivant qui pointe vers le maillon suivant*.
3. Un pointeur *précédent* qui pointe vers le maillon précédent

- Pour le cas d’une *chaîne simple*, le pointeur *précédent est omis de la structure du maillon*

Le pointeur suivant pointe sur NIL et désigne la fin de la liste



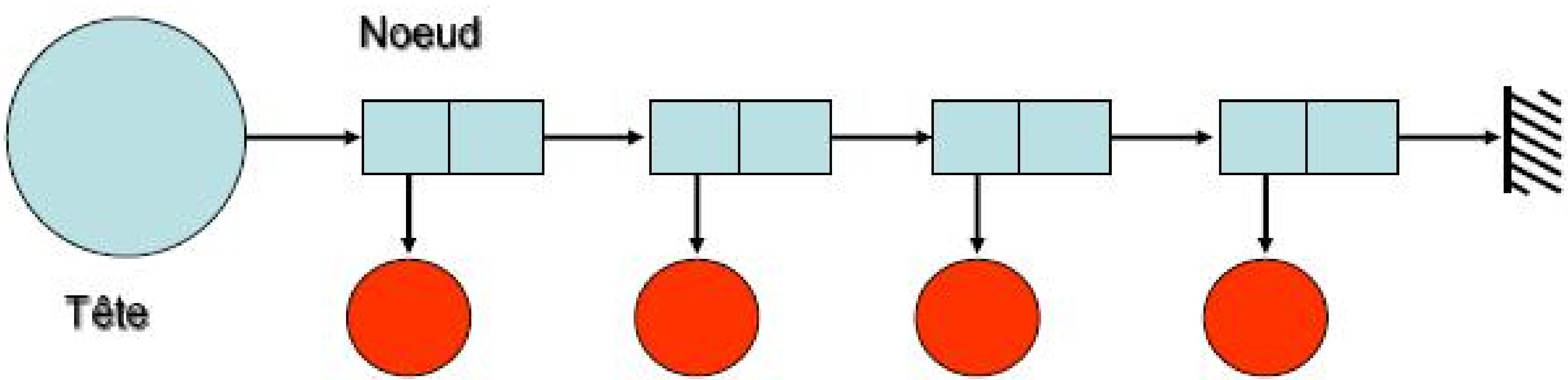
Définition d'un maillon

```
Type
Structure Maillon
    valeur: info
    suivant: ↑Maillon
    Précédent : ↑Maillon
Fin structure
```

- Info: le type de l'information stockée; peut-être un type de base (ex: entier) ou bien un type complexe (un agrégat)

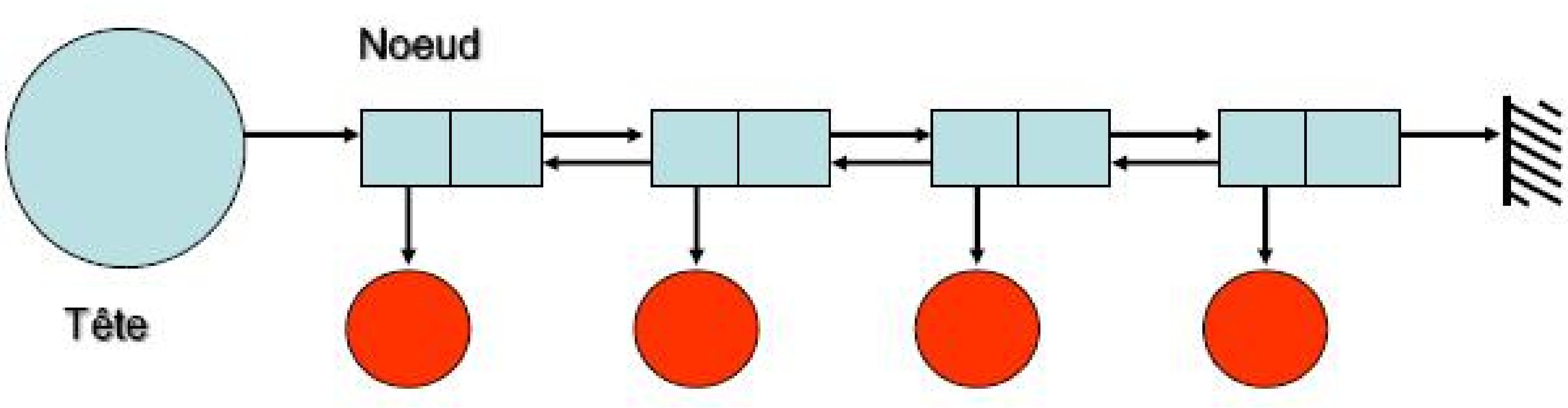
Liste chaînée simple

- Si une liste chaînée est ***simple***, on omet le pointeur précédent de chaque maillon
- Structure :

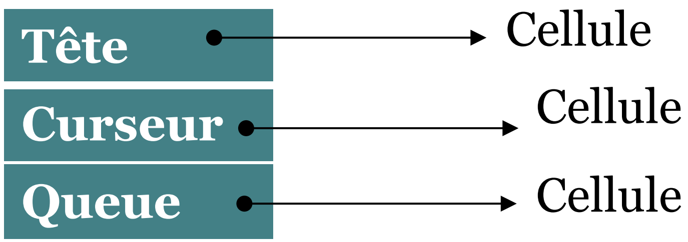


liste doublement chaînée

- Sa structure:



Définition de la classe Liste



- Attributs:
  - premier ou tête: pointeur sur *la cellule tête de liste*
  - Courant : pointeur sur *la cellule courante*
  - *dernier ou queue: pointeur sur la cellule queue de liste*

En Algorithmique

Type

Structure Maillon

valeur: entier

suivant: ↑Maillon

Fin structure

Var: cell1,cell2, cell3, cell4, cell5: Maillon

DEBUT

cell1.valeur ← 4

cell1.suivant←#cell2

cell2.valeur←17

cell2.suivant←#cell3

cell3.valeur←21

cell3.suivant←#cell4

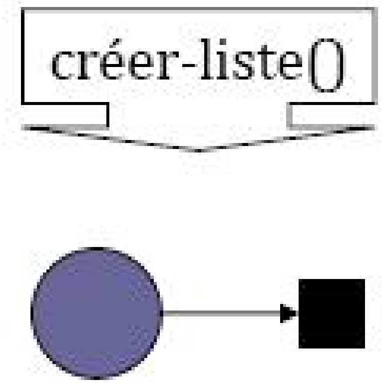
……..

cell5.valeur ←5

cell5.suivant ← NIL

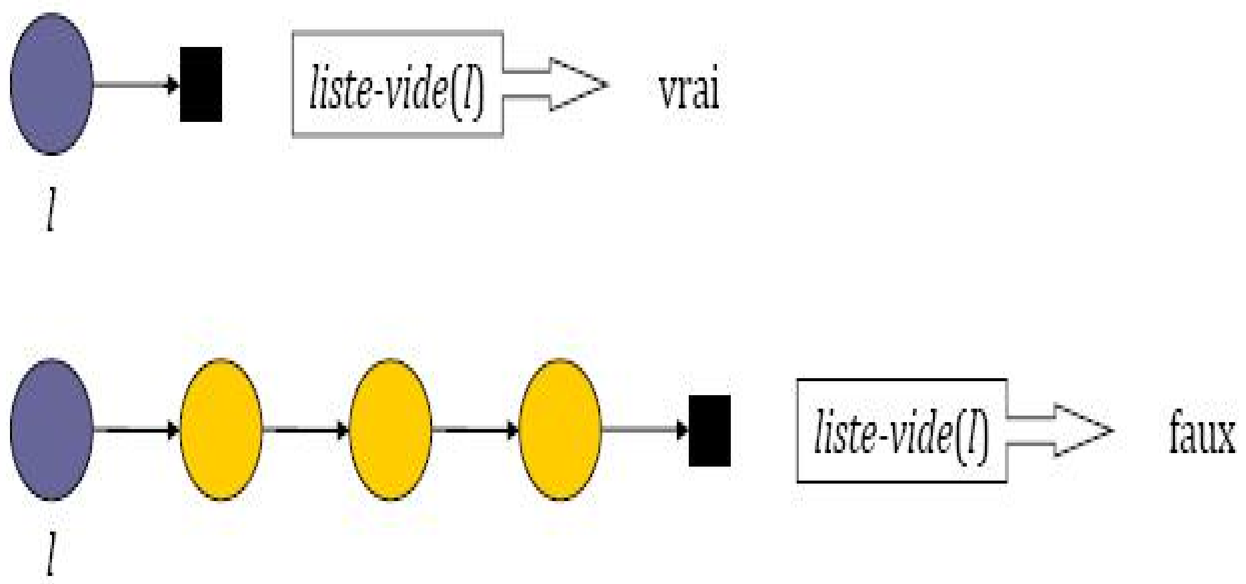
FIN

Créer_liste

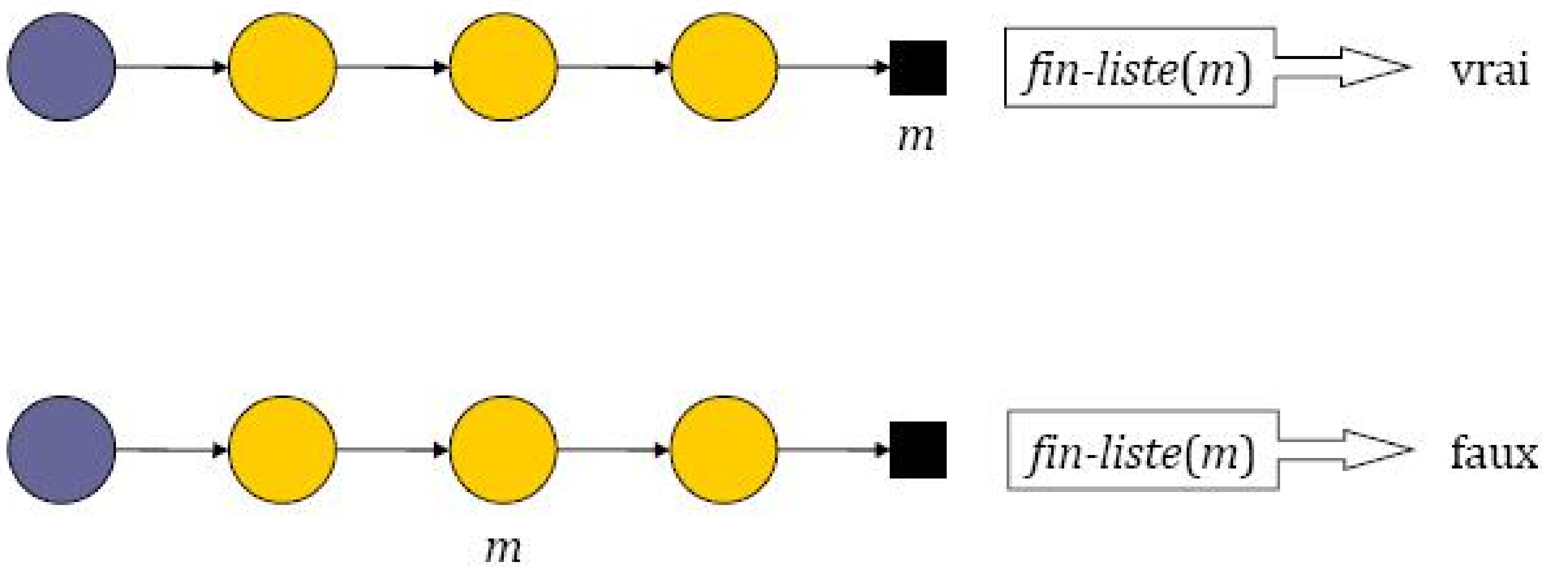


Fonction créer_liste( ): ↑Maillon
Var tête : ↑Maillon
Début
tête ← nouveau (Maillon)
Ecrire (donner une valeur entière:)
Lire (tête ↑.val)
tête →suivant ←NIL
Renvoyer (tête)
Fin

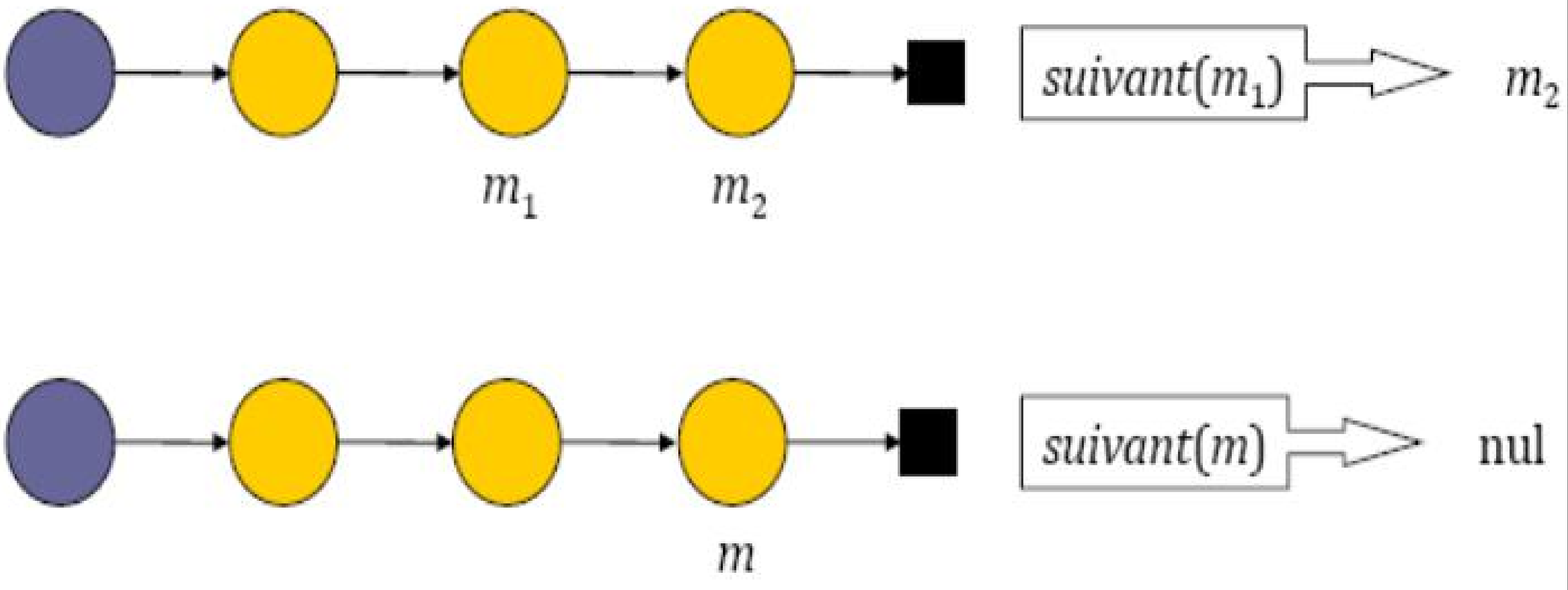
Liste_vide



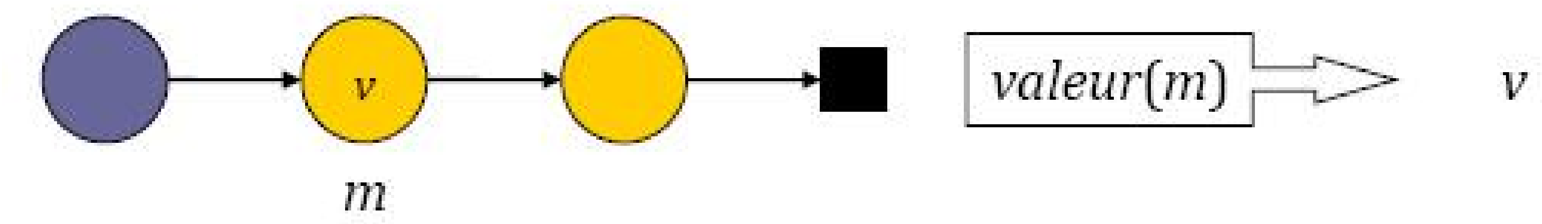
Fin_liste



Suivant (m)



Valeur (m)



D’autres types de listes chaînées

- Une liste peut prendre différentes formes: *chaînée, ou doublement chainée, triée ou non, circulaire ou non.*
  - Si une liste est ***triée***, l’ordre linéaire de la liste correspond à l’ordre linéaire des valeurs stockées dans les éléments de la liste: la tête est le minimum et la queue est le maximum.
  - Si une liste est ***circulaire***, le pointeur précédent de la tête de liste pointe sur la queue et le pointeur suivant de la queue de la liste pointe sur la tête

Traitements sur les listes

Relatifs à la composition structurelle de la liste :

–Positionnement sur la première cellule de la structure

–Positionnement sur la dernière cellule de la structure

–Calcul de la longueur d'une liste (nombre de cellules)

–Reconnaissance d'une liste vide

–Déplacement du positionnement courant sur la cellule suivante

Traitements sur les listes

Relatifs à l'information enregistrée dans une liste:

–Enregistrement de données jusqu'à épuisement du flot de données

–Visualisation de l'information enregistrée dans une cellule, quelle que soit sa place dans la liste

–Visualisation de l'ensemble des informations enregistrées dans la liste

–Suppression d'une cellule; ajout d’une cellule

Parcourir les éléments d’une liste

```
Procedure Parcours (E/S   l: liste)
Var courant : ↑Maillon
Debut
courrant ← L →premier
tant que courant < > NIL
courant ← courant→suivant
fintantque
fin
```

Rechercher un élément dans une liste

Fonction Rechercher1 (Liste L, x: entier) : booleen
Var courant: ↑Maillon
B:booleen
Début
courant ← L→premier
B ←faux
Tant que (courant < >NIL et B = faux)faire
Si courant↑.val = x alors B←vrai
Sinon
B←faux
courant ← courant→suivant
Finsi
Fin tantque
Renvoyer B
Fin

Recherche dans une liste chainée

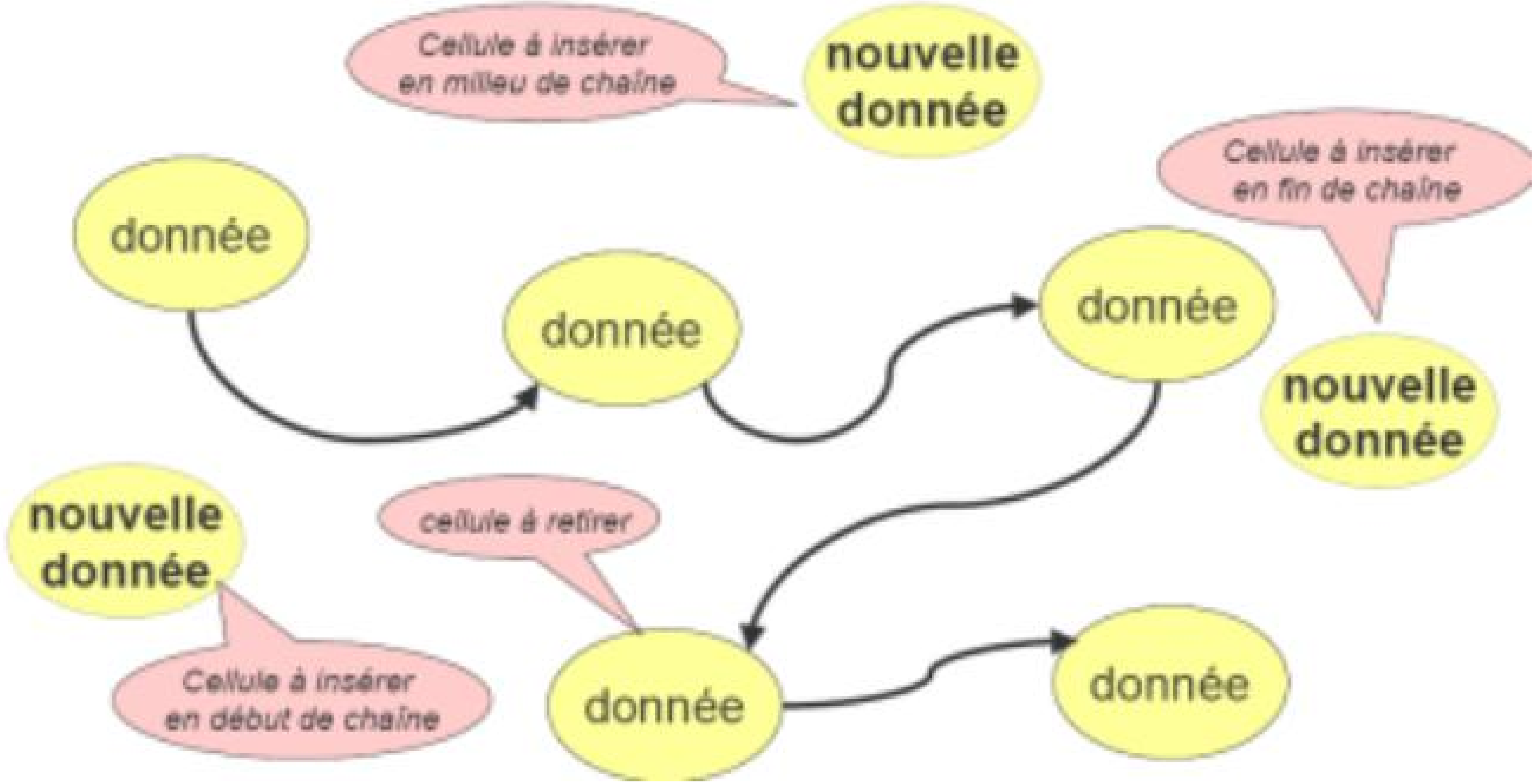
```
Fonction Recherche2 (L: liste, x : entier):↑maillon
Var trouve : booleen
    courant : ↑maillon
Début
courant ← L→premier
Trouve ← faux
Tant que courant <>NIL et trouve = faux faire
    Si courant ↑.valeur = x   Alors trouve ←vrai
    Sinon  courant ← courant→suivant
fin tantque
Si trouve =vrai alors Renvoyer courant
Sinon Renvoyer NIL
Finsi
Fin
```

L'insertion dans une liste chaînée

- Le cas de listes chaînées simples et non ordonnées

1. Créer un élément de type maillon et ensuite changer les pointeurs.
2. L'insertion peut se faire
   – en tête de liste,
   – en fin de liste,
   – après l'élément courant
   – avant l'élément courant.

Insertion / Suppression d’une cellule



En Algorithmique

Type

Structure Maillon

valeur: entier

suivant: ↑Maillon

Finstructure

Structure liste ou var liste : ↑Maillon

Premier: ↑Maillon

Dernier : ↑Maillon

Finstructure

Insertion en tête d'une liste L

```
Procedure Insertion_tete (E/S L:Liste)
Var element : ↑Maillon
Début
Nouveau(element)
Lire (element↑.val)
Si L→premier = NIL alors L→premier← element
                                  L→dernier← element
                                   element →suivant ← NIL
 Sinon  element→suivant ← L→premier
         L→premier ← element
Finsi
Fin
```

Insertion en fin de liste

Procédure Insert_fin (E/S L:Liste)

Var element : ↑Maillon

Début

Nouveau(element)

Lire (element↑.val)

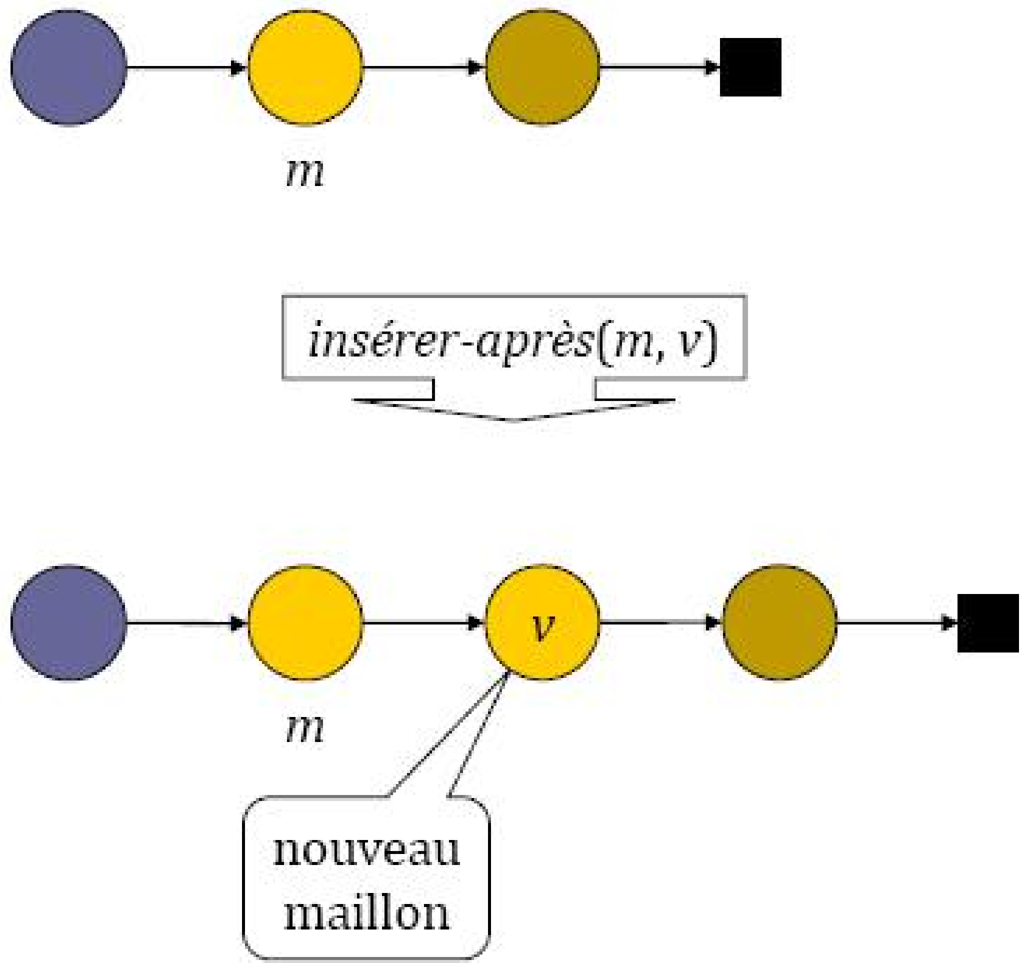
L→dernier→suivant ← element

Element→Suivant ← NIL

L→dernier ← element

Fin

Exercice : Insérer après ?

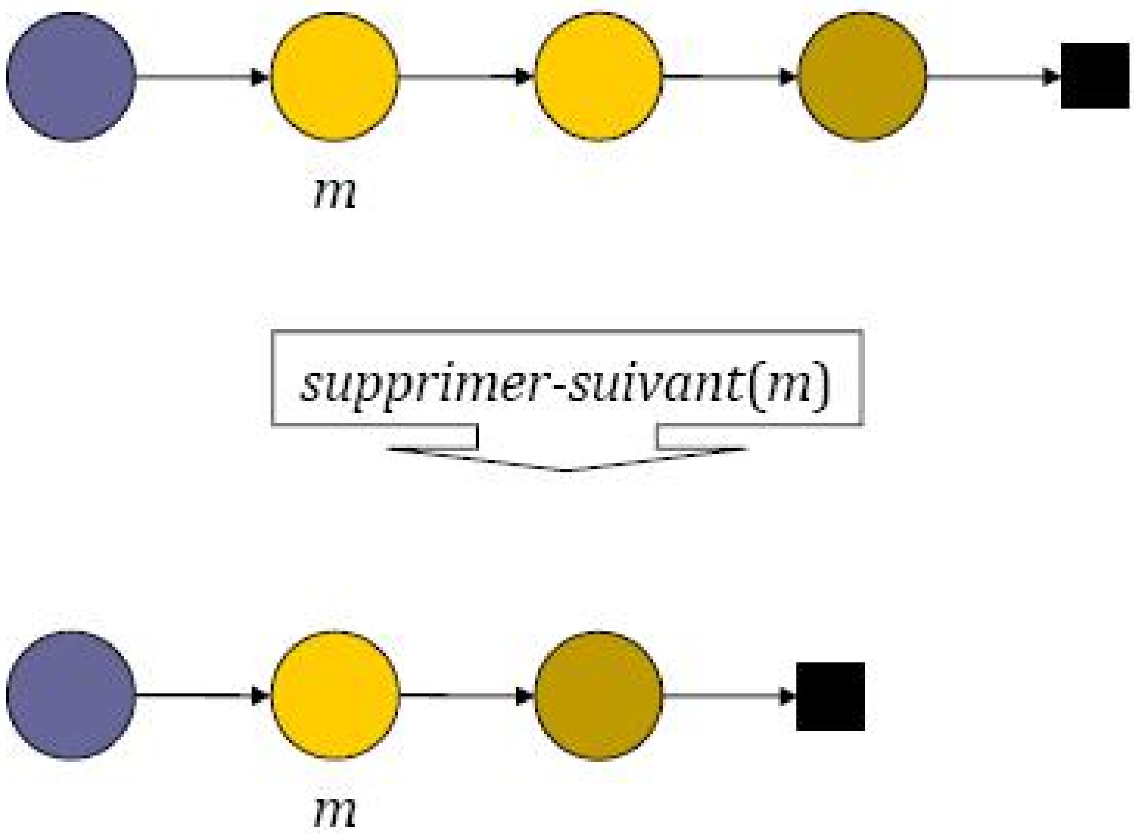


Suppression de l'élément courant

- Trois cas sont possibles :

1. soit le pointeur courant est égal au pointeur premier: on supprime l'élément de tête avec l'algorithme associé.
2. soit le pointeur courant est le pointeur de queue : supprimer le dernier élément avec l'algorithme précédent.
3. Sinon : on procède comme suit :

Supprimer_suivant



Intérêts des listes chaînées ?

- Elle fournit une représentation simple et souple pour les ensembles dynamiques, supportant toutes les opérations (recherche, insertion, suppression, min, max., successeur, prédécesseur)
- Permettre l'allocation de mémoire en fonction des besoins, de façon dynamique
- Faciliter la gestion de la mémoire occupée en cas d'insertion ou de suppression de nouvelles données
- Simulation de phénomènes du monde physique mal représentés par la structure en tableaux

Ex. File d'attente à un guichet; Urgences d'un hôpital ;Gestion des dossiers empilés sur un bureau

Avantages et Inconvénients

- ☺On peut avoir autant d'éléments que la mémoire le permet.
- ☺Pour déclarer une liste il suffit de créer le pointeur qui va pointer sur le premier élément de la liste. Aucune taille n'est à spécifier.
- ☺Il est possible d'ajouter, de supprimer, d'intervertir des éléments d'une liste chaînée sans recréer la liste en entier, mais en manipulant simplement leurs pointeurs.
- ☹Il est impossible d'accéder directement à l'élément i de la liste. Il faut traverser les i – 1 éléments précédents de la liste.

Les listes vs. Les tableaux

- Avantages sur les tableaux
  - Taille variable
  - Réarrangement efficace des éléments
- Inconvénients sur les tableaux
  - Impossibilité d'accéder directement à un élément quelconque

Cycle de vie de la variable pointée

- Lors du lancement de l'algorithme: seule la variable pointeur est créée. Aucun espace mémoire n'est affecté à la variable pointée qui n'existe pas encore.
- Au cours de l'exécution de l'algorithme:
  - création de variable pointée (nouveau)
  - Libérer cet espace mémoire
  - L'associer à une nouvelle variable pointée …

=> Créer un grand nombre de variables pointées et les associées toutes à une seule variable pointeur.

Implémentation d’une liste à l’aide d’une représentation contigue

Type position = entier

Liste = tableau [1..N]d’éléments

Chapitre 4 : les piles et les files

Définition

- Les piles et les files sont des structures de données.
- Chacune de ces structures offre trois opérations élémentaires :

1. Tester si la structure est vide.
2. Ajouter un élément dans la structure.
3. Retirer un élément de la structure.

- Une pile, ou une file, est une structure qui se modifie au cours du temps !

Les piles et les files

- Elles se distinguent par la relation entre éléments ajoutés et éléments retirés.

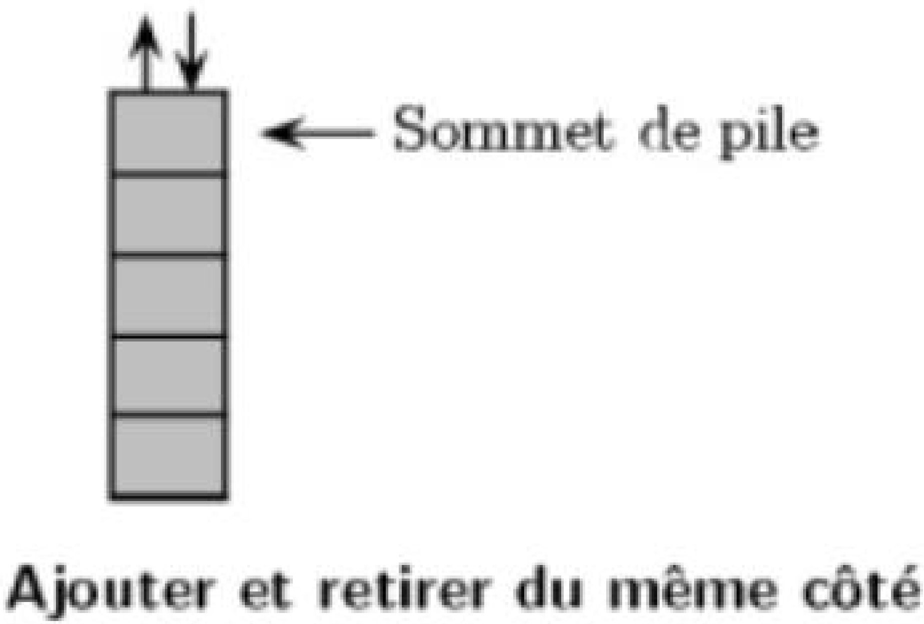
1. Dans le cas des piles :
   - C'est le dernier élément ajouté qui est retiré.
   - La pile est une structure LIFO.
2. Dans le cas des files :
   - C'est le premier élément ajouté qui est retiré.
   - La file est une structure FIFO.

Une pile

- Une structure de données mettant en œuvre le principe (LIFO), appelées aussi liste LIFO (Last In First Out)
  - Ex : pile de documents, pile d'assiettes ...
- Une liste particulière dont les insertions et les suppressions ne se font qu'à une seule extrémité appelée sommet de la pile.

Les piles

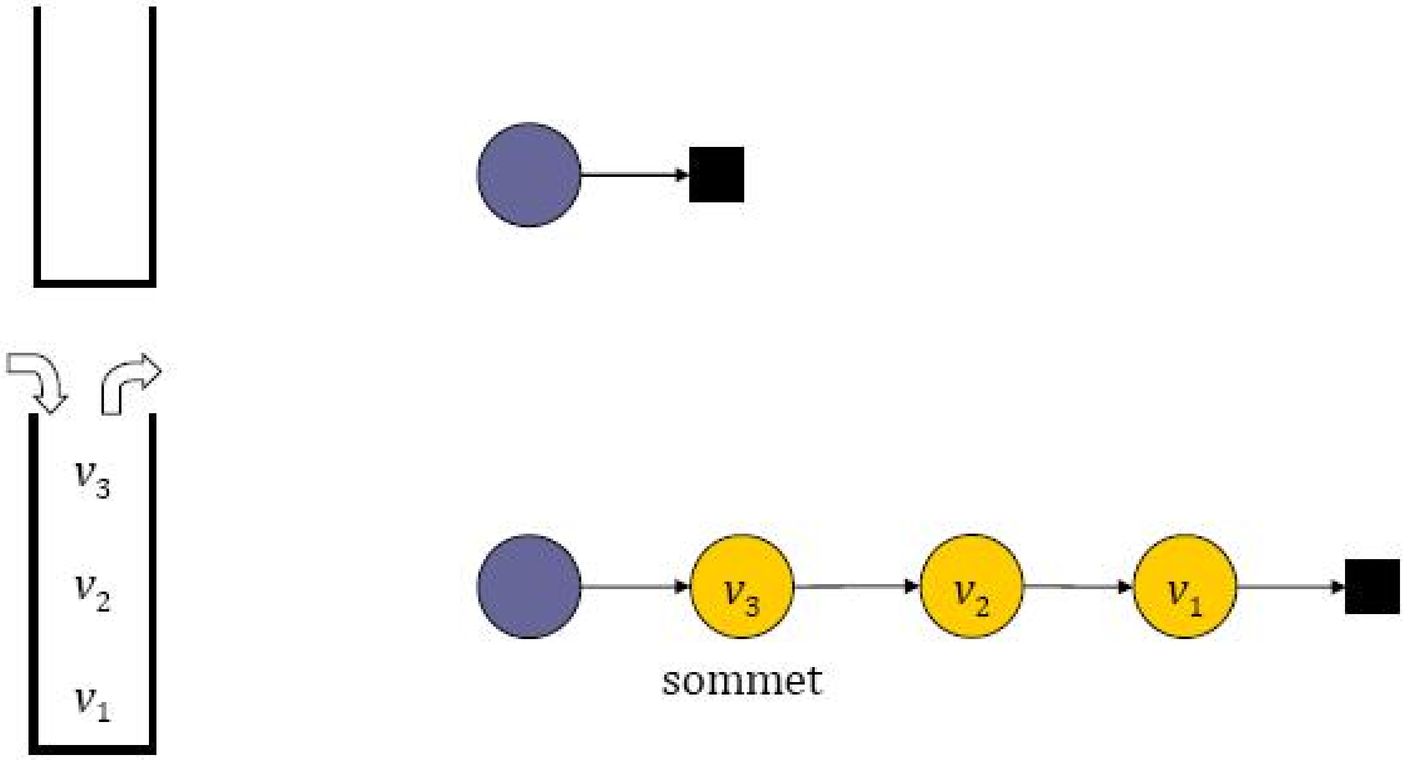
- Les piles sont souvent nécessaires pour rendre itératif un algorithme récursif.



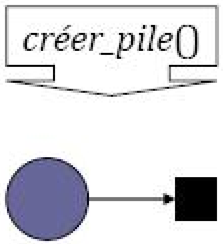
Pile

e5
e4
e3
e2
e1

Représentation de la pile



Opération sur les piles



Structure nœud

val:entier

Suivant: ↑nœud

Finstructure

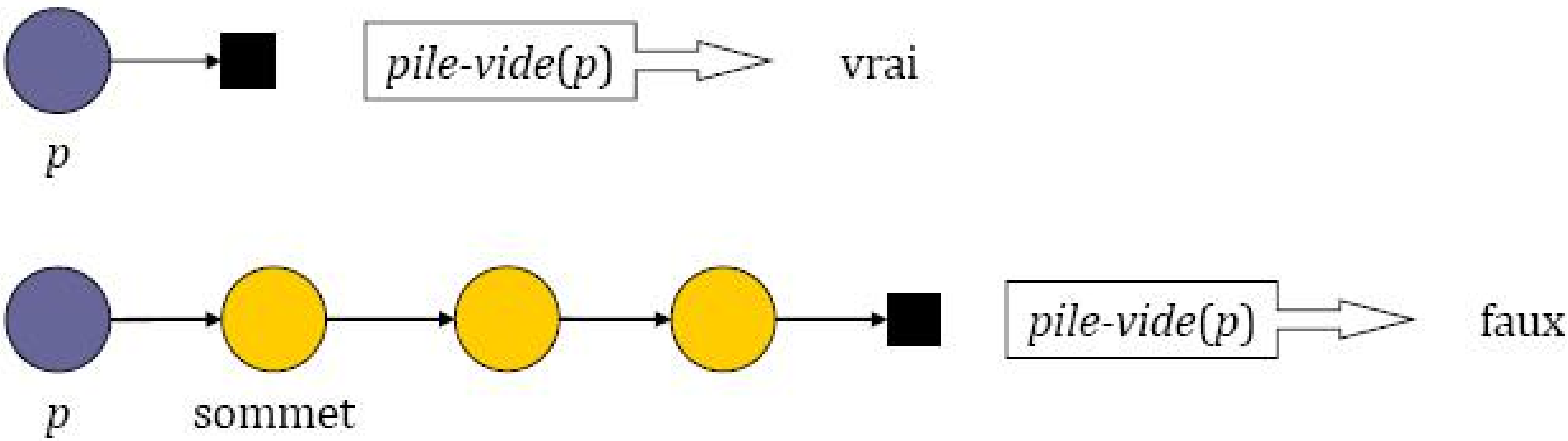
Procédure Creér_Pile(E/S sommet: ↑nœud)

  Début

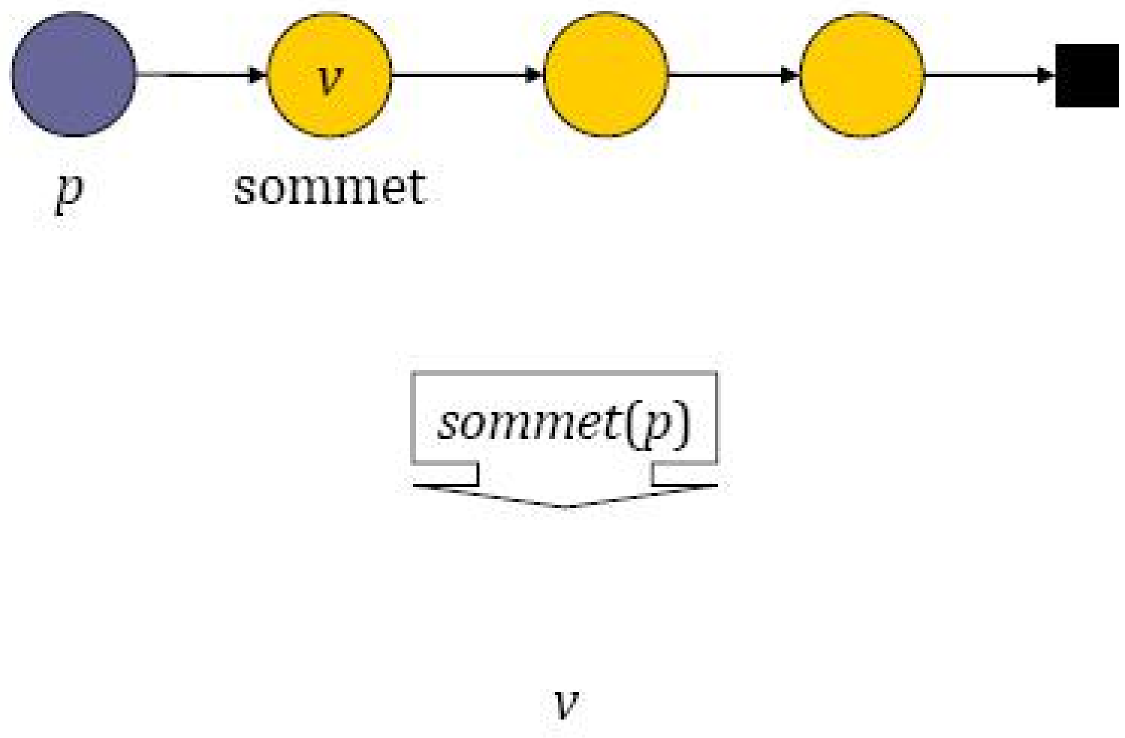
  Sommet ←NIL

Fin

Pile_vide(p)



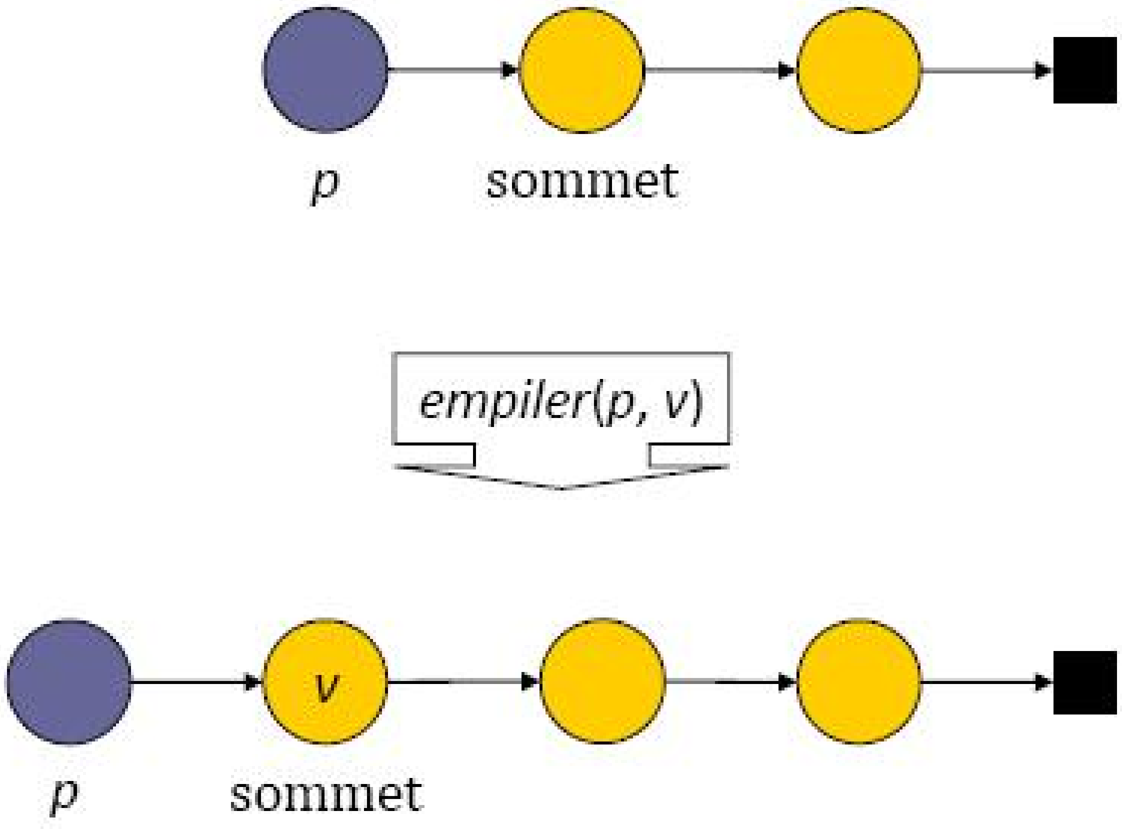
Sommet(p)



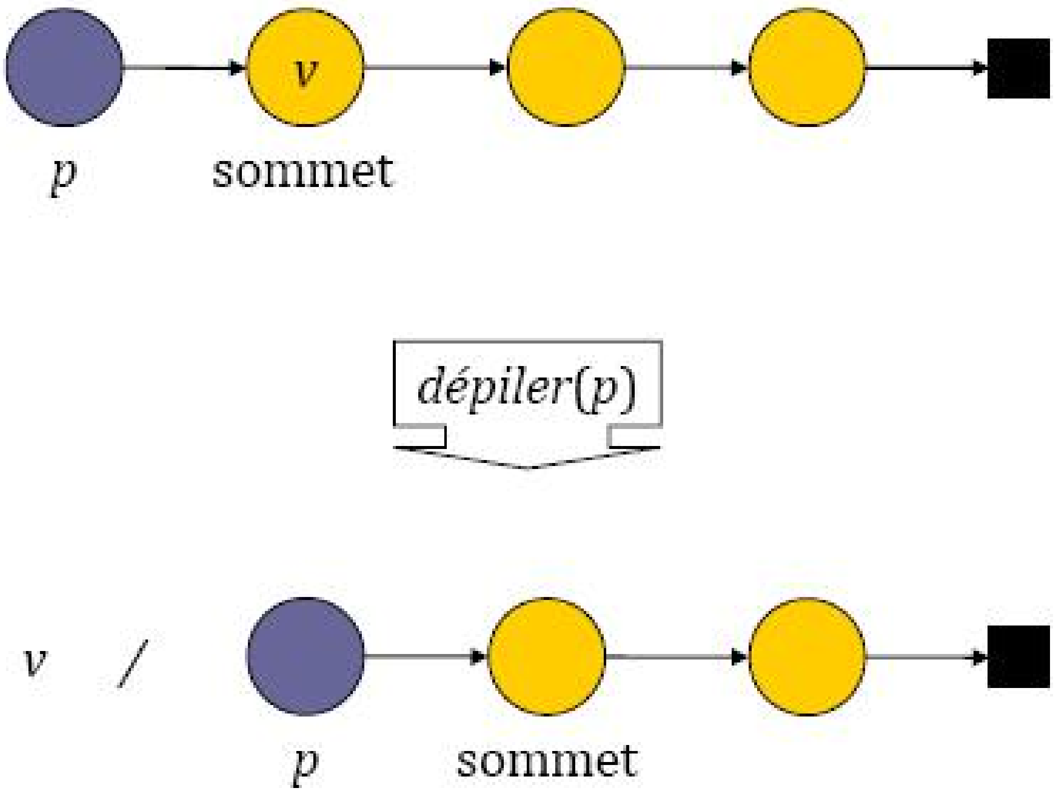
Opérations de base sur les piles

1. Insérer un élément dans une pile : Empiler(élément)
2. Supprimer un élément d'une pile : Dépiler()
   - L'élément supprimé est celui le plus récemment inséré.
3. Créer une pile (toujours vide) : creerPile()
4. Récupérer l'index du haut de la pile: sommet()
5. Connaître la taille courante de la pile : longueur()

Empiler(p,v)

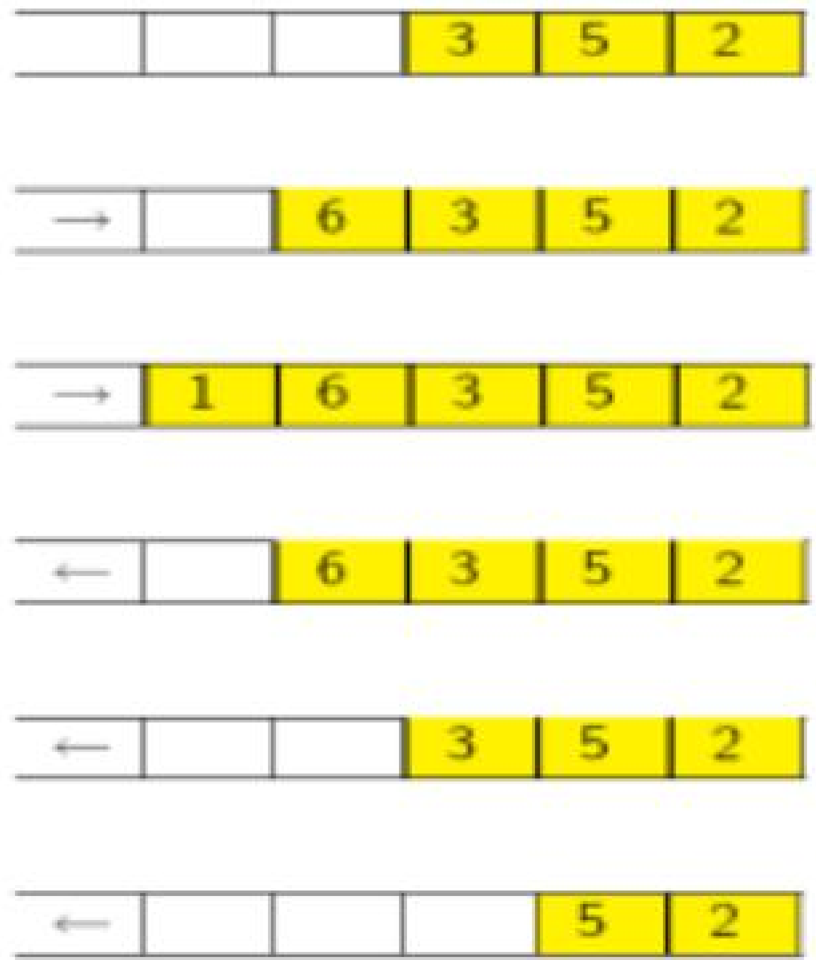


$V \leftarrow$ *Dépiler(p)*



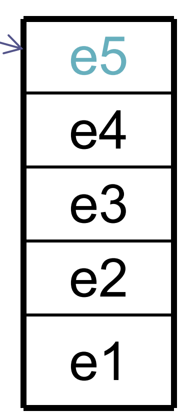
Manipulation des piles

- **Empiler(6)**
- **Empiler(1)**
- **Dépiler()** → La pile renvoie 1
- **Dépiler()** → La pile renvoie 6
- **Dépiler()** → La pile renvoie 3



Implémentation à l’aide d’une représentation contigue

```
Type pile = structure
        element : tableau [1..N] d’entiers
        sommet : entier
        nb_elt : entier
Fin
Var p : pile
```



Opérations avec la représentation contiguë

```
Procedure Pile_vide(S p:pile)
p.Sommet ←null
p.nb_elt  ←0
Fin

Fonction est_vide(p: pile):booleen
Debut
Renvoyer (nb_elt = 0)
Fin
```

Opérations avec la représentation contiguë

Procedure empiler(E x: entier; E/S p: pile)

Var courant : ↑entier

Debut

Allouer(courant)

Courant.val ←x

Courant →suivant ←p.sommet

p.sommet ←courant

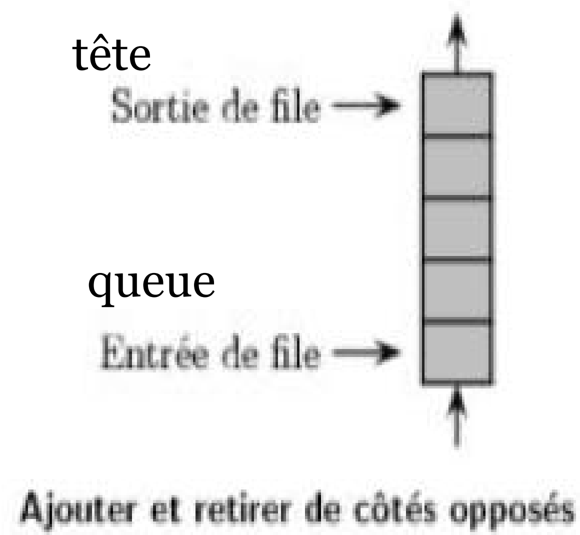
p.nb_elt ←p.nb_elt +1

Fin

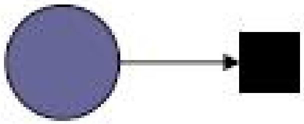
Les files

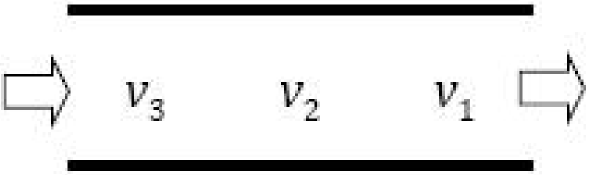
e5	e4	e3	e2	e1

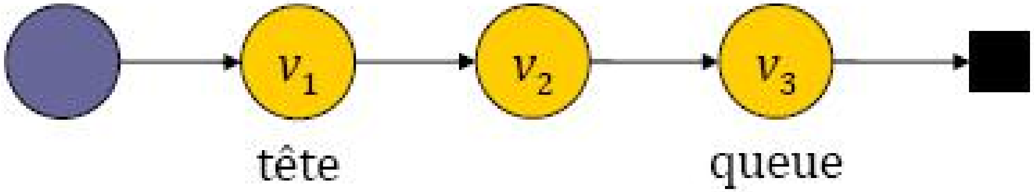
- Une file est une liste particulière dans laquelle les éléments sont insérés à une extrémité, la queue, et retirés à une autre extrémité, la tête.
- Une file est appelée liste FIFO (First In First Out)
- Ex : file d'attente



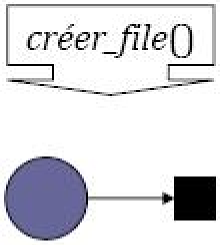
Représentation d’une file





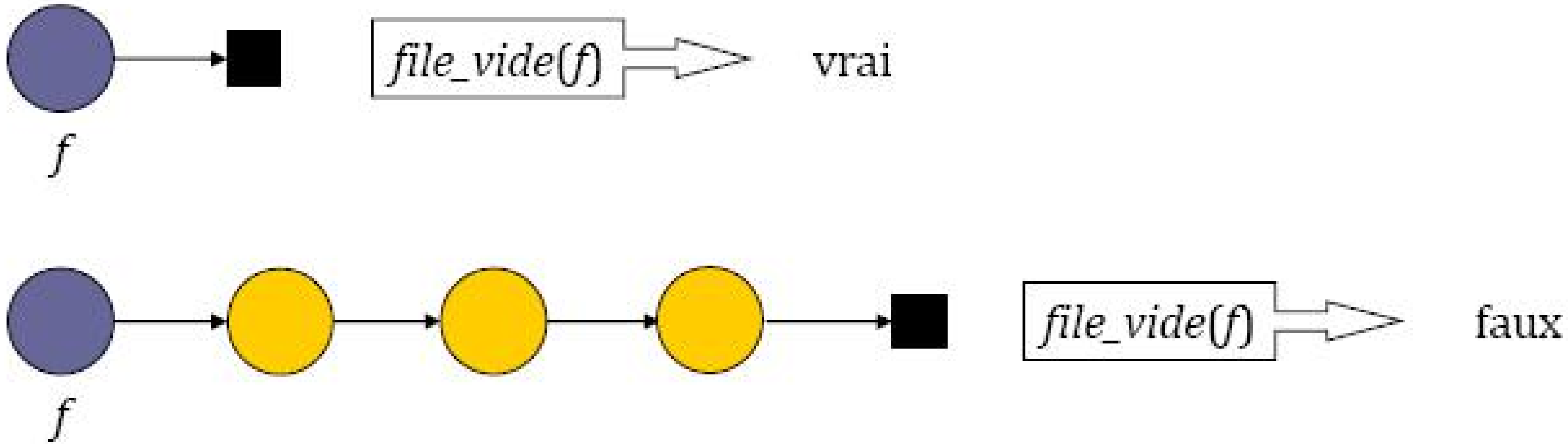


Opération sur les files

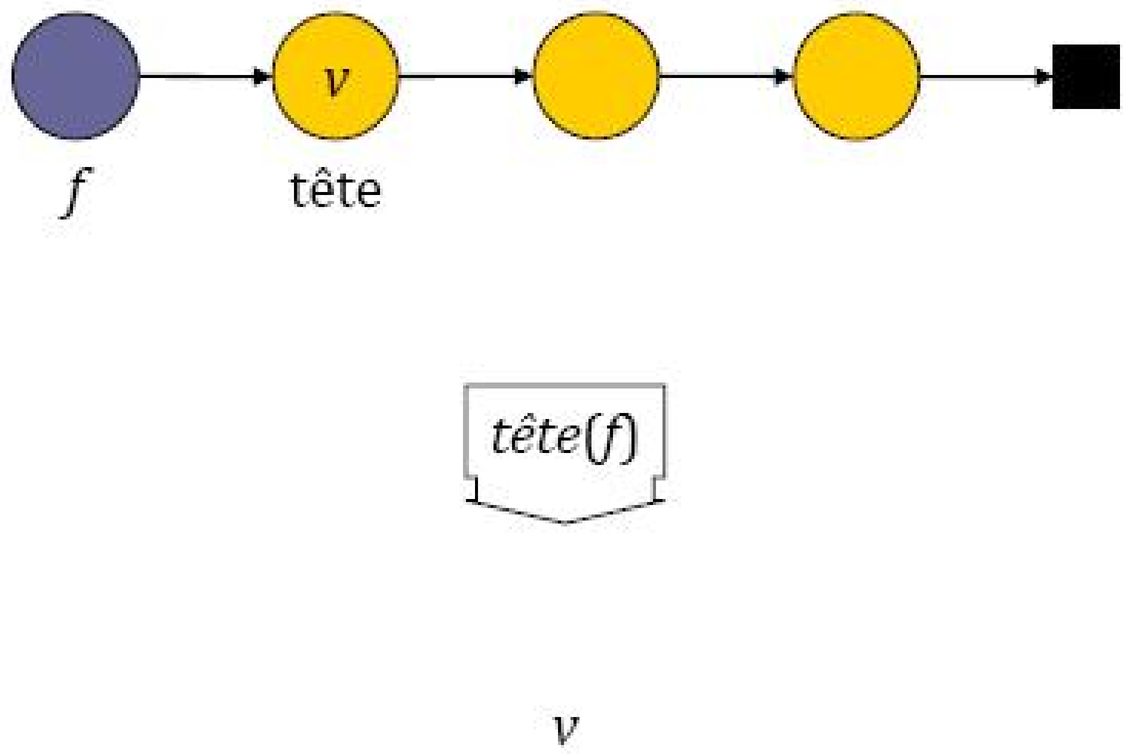


Structure nœud
val:entier
Suivant: ↑nœud
Finstructure
Procédure CreérFile(E/S sommet, queue: ↑nœud)
  Début
  Sommet ←NIL
  Queue ←NIL
Fin

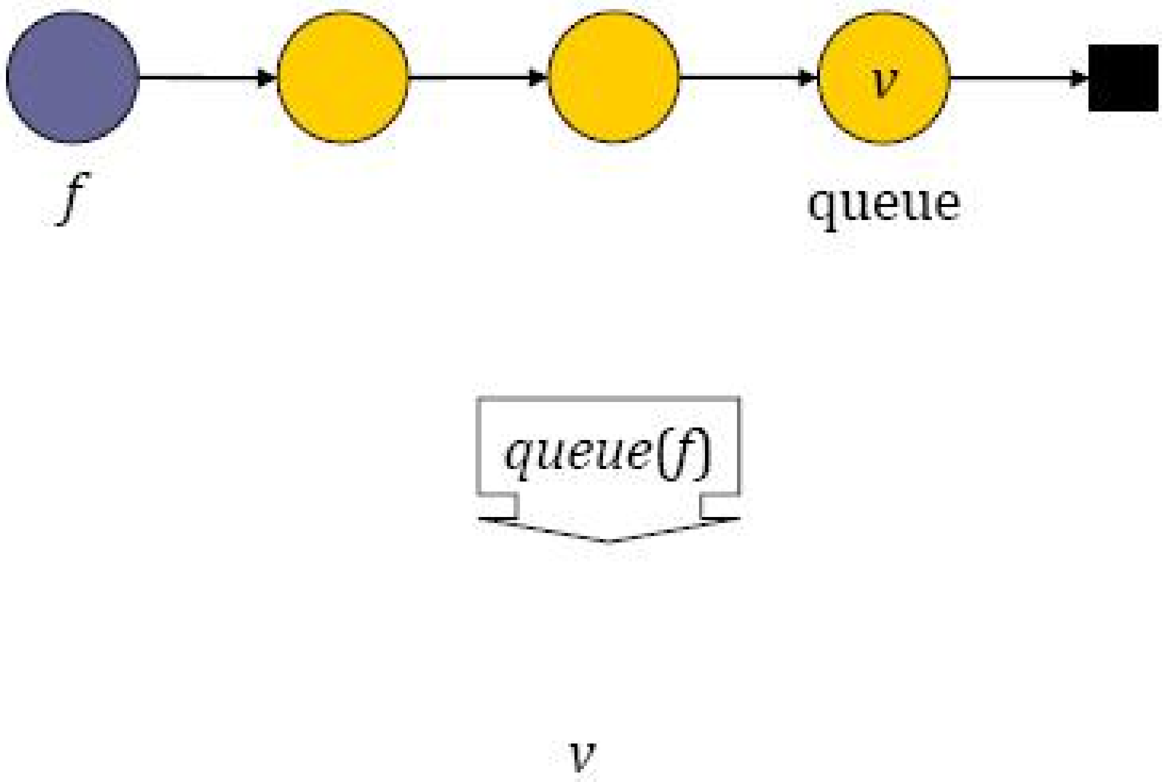
File_vide



Tête(f)



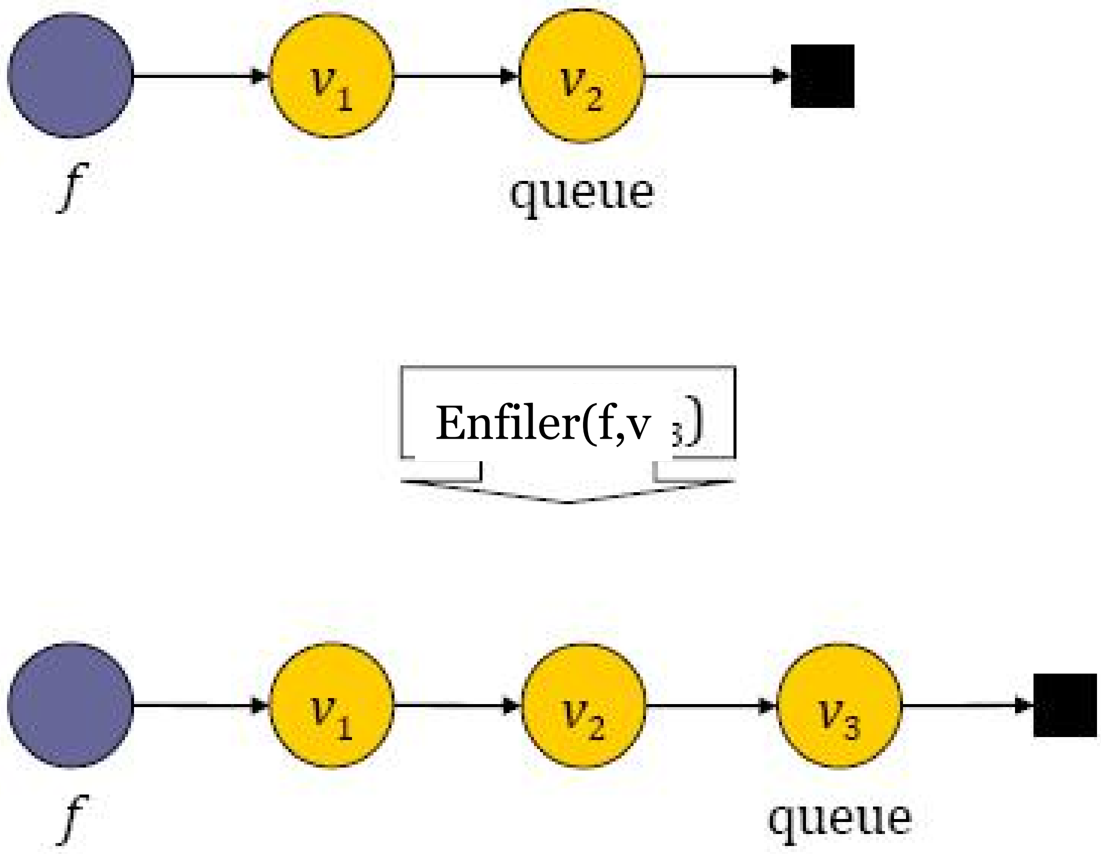
Queue(f)



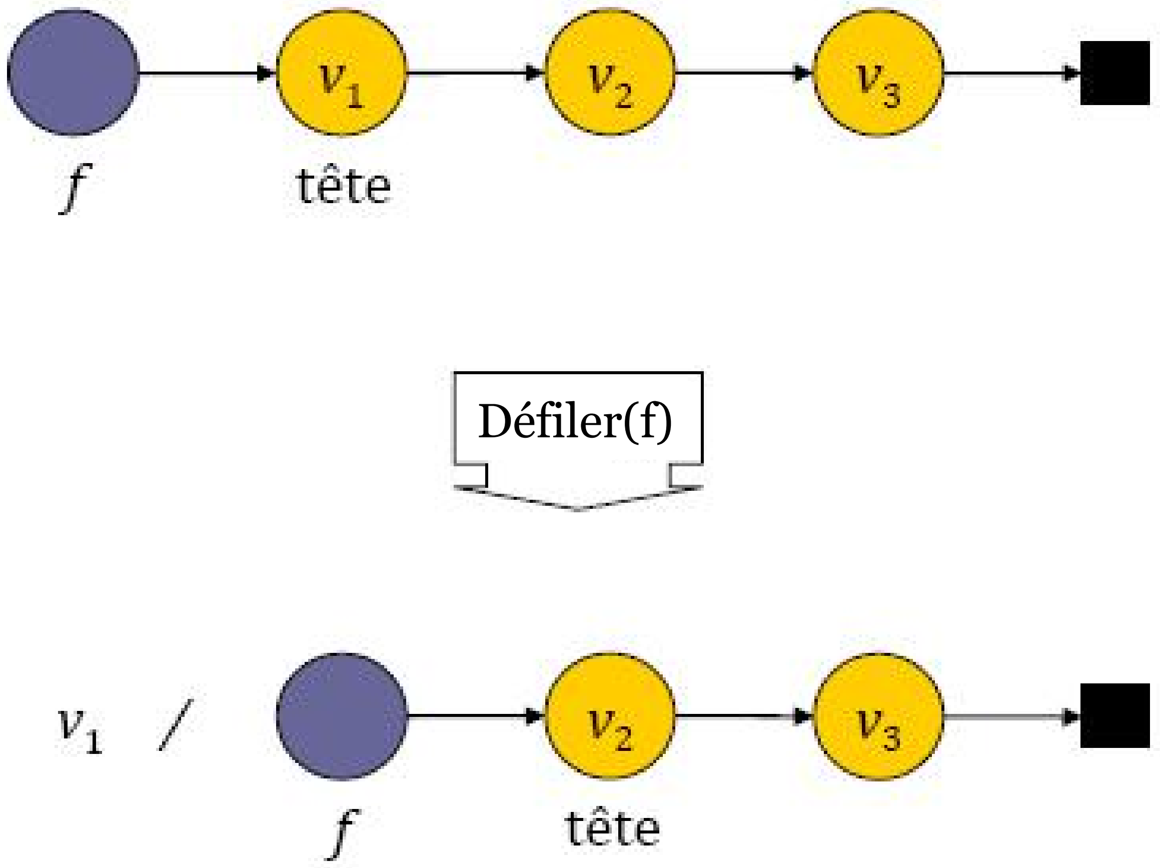
Opérations de base sur les files

1. Créer une file (toujours vide) : creerFile()
2. Insérer un élément dans une file : Enfiler(élément)
3. Supprimer un élément d'une file : Défiler()
    - L'élément supprimé est celui le plus ancien dans la file.
4. Récupérer l'index du premier élément de la file: tete()
5. Connaître la taille courante de la pile : longueur()
6. Récupérer l'index du dernier élément de la file: queue()

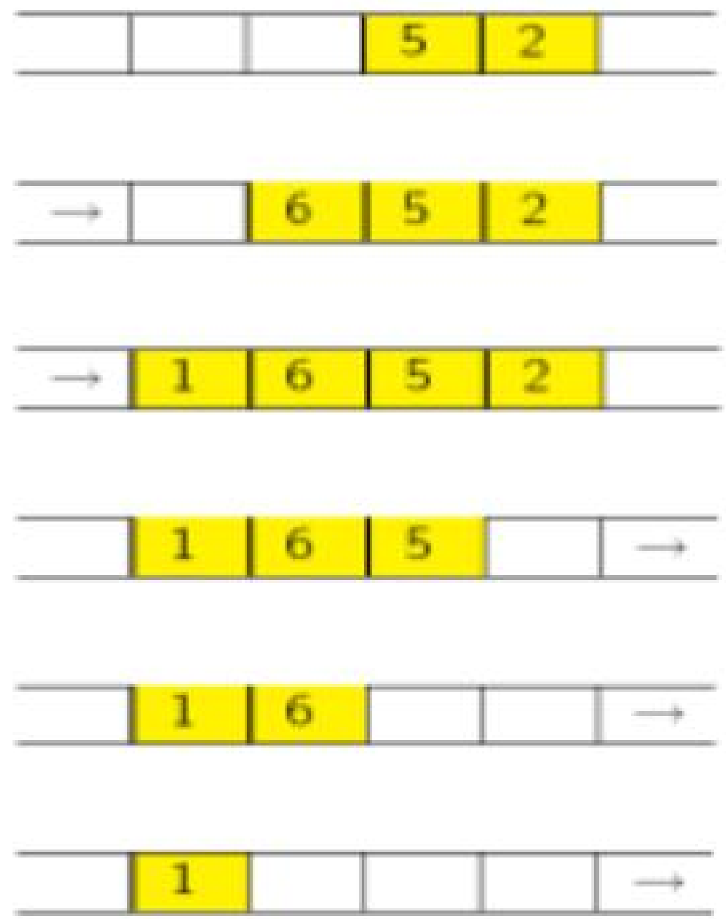
Enfiler(f,v)



Défiler(f)



Manipulation des files



- **Enfiler(6)**
- **Enfiler(1)**
- **Défiler()** → La file renvoie 2
- **Défiler()** → La file renvoie 5
- **Défiler()** → La file renvoie 6

Implémentation d’une file à l’aide d’une représentation contiguë

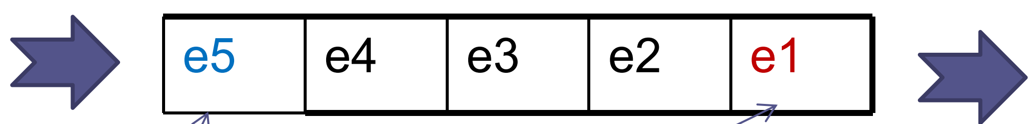
Type file = structure

val : tableau [1..N] d’entiers

nb_elt : entier

tete : entier

fin : entier



Fin

Var f: file

Opérations avec la représentation contiguë

```
Procedure file_vide (S f: file)
Debut
f.Tete ←null
f.Fin ←null
f.nb_elt ←0
Fin

Fonction  est_vide (E/S f: file): booleen
Debut
Renvoyer (f.nb_elt = 0)
Fin
```

Opérations avec la représentation contiguë

Procedure enfiler (E x: entier; E/S f: file)
Var courant : ↑entier
Debut
Nouveau (courant)
Courant →val ←x
Courant →suivant ←NULL
Si f.tete = NULL alors f.tete ← courant
f.fin ←courant
Sinon (f.fin) →suivant ←courant
f.fin ← courant
Finsi
f.nb_elt ← f.nb_elt +1
Fin

Calcul arithmétique

- Une application courante des piles se fait dans le calcul arithmétique:
  - l'ordre dans la pile permet d'éviter l'usage des parenthèses.
- La notation postfixée consiste à placer les opérandes devant l'opérateur.
- La notation infixée (parenthèsée) consiste à entourer les opérateurs par leurs opérandes.

Exemple

- La notation usuelle, comme (3 + 5) * 2, est dite infixée. Elle s'écrira en notation postfixée : 3 5 + 2 *
- La notation infixée 3 + (5 * 2) s'écrira: 3 5 2 * +
- Notation infixe: A * B/C, qui s’écrira en notation postfixe est: AB * C/.
- Forme infixe: A/B ** C + D * E - A * C
- Forme postfixe: ABC ** /DE * + AC * -
- Ecrire un algorithme qui transforme une expression infixe en une notation postfixe.

Principe

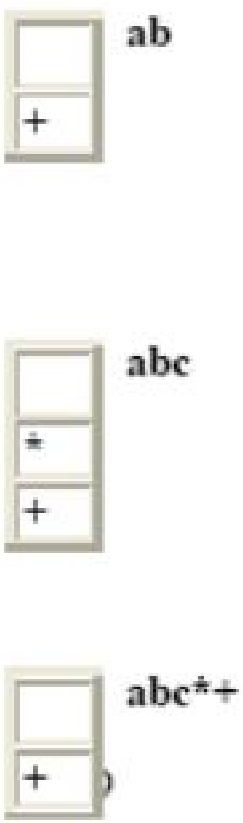
```
initialise la pile et l'output postfixe à vide;
Tant que( pas la fin de l'expression infixe)
    {
prendre le prochain item infixe
Si (item est une valeur)  alors
        concaténer item à postfixe
Sinon si (item == '(')   alors
            empiler item
             tant que (x !=')')
              {empiler item
               }
Sinon si  (item == ')') {
              dépiler sur x
              tant que (x != '(')
              concaténer x à postfixe
              dépiler sur x
                }
Sinon  {
Tant que(precedence(sommet) >=
    precedence(item))
     dépiler sur x
     concaténer x à postfixe;
      empiler item;
      }
}
Tant que (pile non vide)
dépiler sur x
concaténer x à postfixe;
```

- Précédence des opérateurs :
  - 4 : '(' – dépiler seulement si une ')' est trouvée
  - 3 : tous les opérateurs unaires
  - 2 : / *
  - 1 : + -

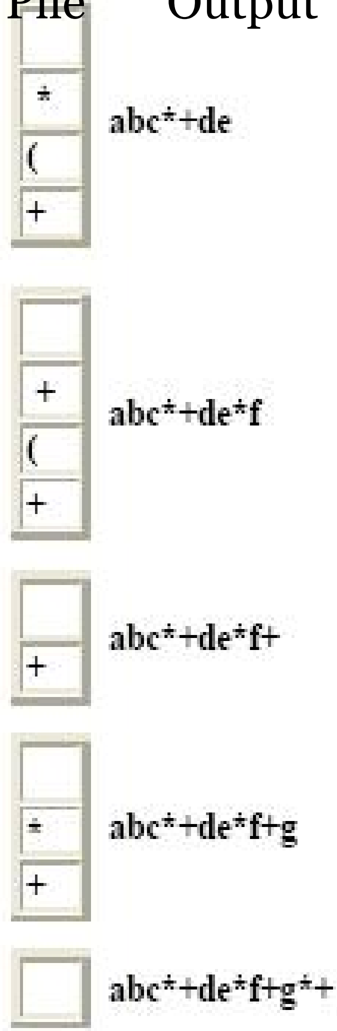
Application

- considérons la forme infixe de l’expression a+b*c+(d*e+f)*g

Pile Output



Pile Output



Algorithme du postfixe au calcul

Initialiser la pile à vide;

Tant que (ce n'est pas la fin de l'expression postfixée)

{

prendre l'item prochain de postfixe;

Si (item est une valeur) alors empiler;

Sinon si (item operateur binaire)

{

dépiler dans x;

dépiler dans y;

effectuer y operateur x;

empiler le résultat obtenu;

}

Sinon si (item opérateur unaire)

{

dépiler dans x;

effectuer opérateur(x);

empiler le résultat obtenu;

}

}

La seule valeur qui reste dans la pile est le résultat recherché

Opérateur binaire: +, -, *, /, etc.,
Opérateur unaire: moins unaire, racine carrée, sin, cos, exp, … etc.

Application

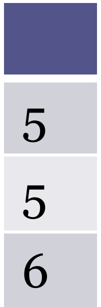
- Considérons l’expression en postfixe suivante:

6 5 2 3 + 8 * + 3 + *

- Le premier item est une valeur (6); elle est empilée.
- Le deuxième item est une valeur (5); elle est empilée.
- Le prochain item est une valeur (2); elle est empilée.
- Le prochain item est une valeur (3); elle est empilée.
- La pile devient

3
2
5
6

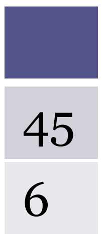
- Les items restants à cette étape sont: + 8 * + 3 + *
- Le prochain item lu est '+' (opérateur binaire): 3 et 2 sont dépilés et leur somme '5' est ensuite empilée:



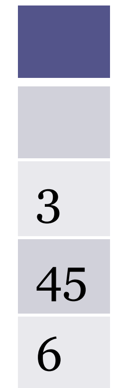
- Ensuite 8 est empilé et le prochain opérateur *:

8, 5 sont dépilés, 40 est empilé

- Ensuite l’opérateur + suivi de 3:

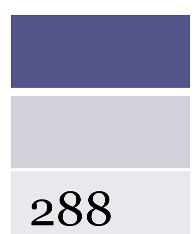


40, 5 sont dépilés ; 45 pushed, 3 est empilé



Ensuite l’opérateur +: 3 et 45 sont dépilés et 45+3=48 est empilé

Ensuite c’est l’opérateur *: 48 et 6 sont dépilés et 6*48=288 est empilé



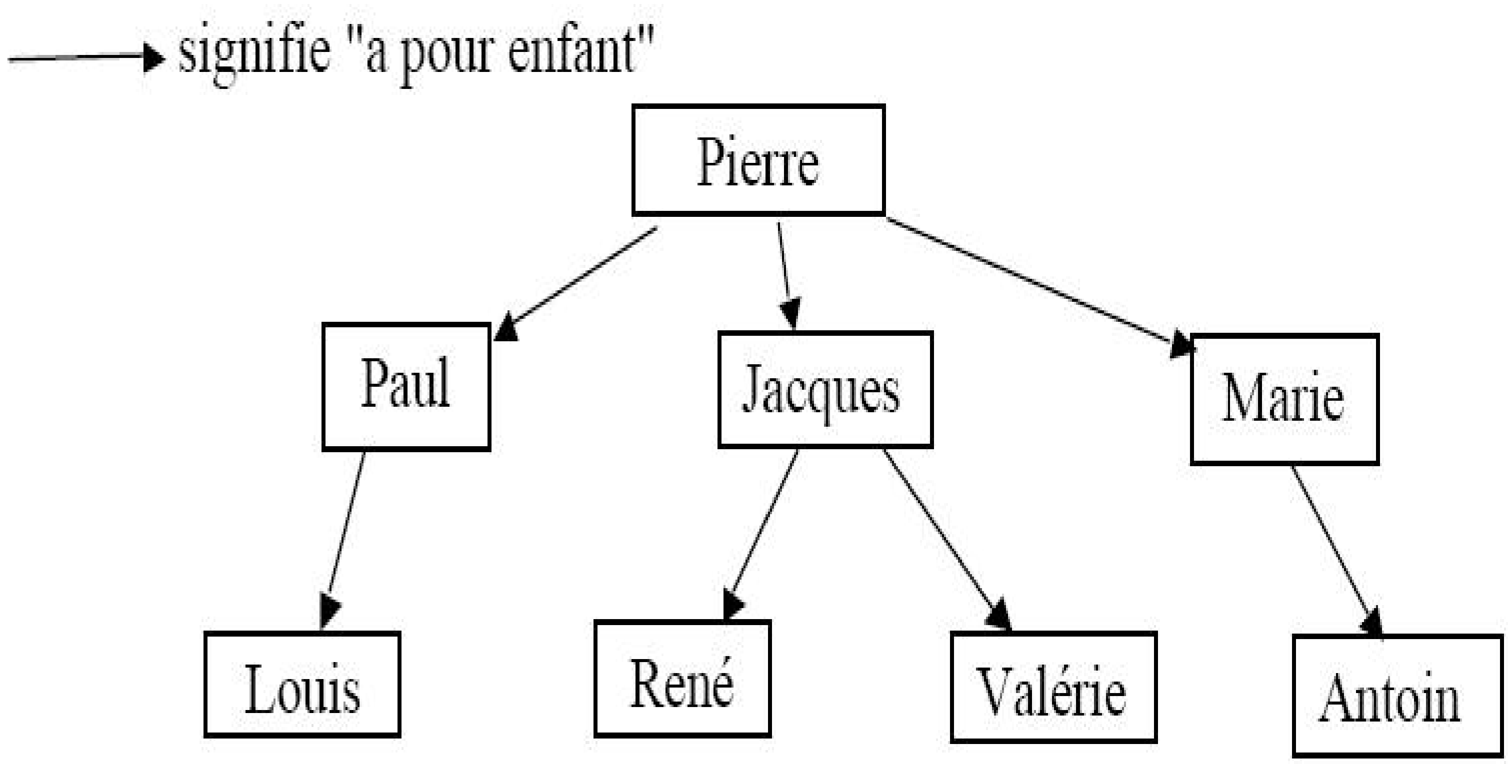
Il n’y plus d’items à lire dans l’expression postfixée et il n’y a qu’une seule valeur dans la pile représentant la réponse finale: 288.

Chapitre 5 : les arbres

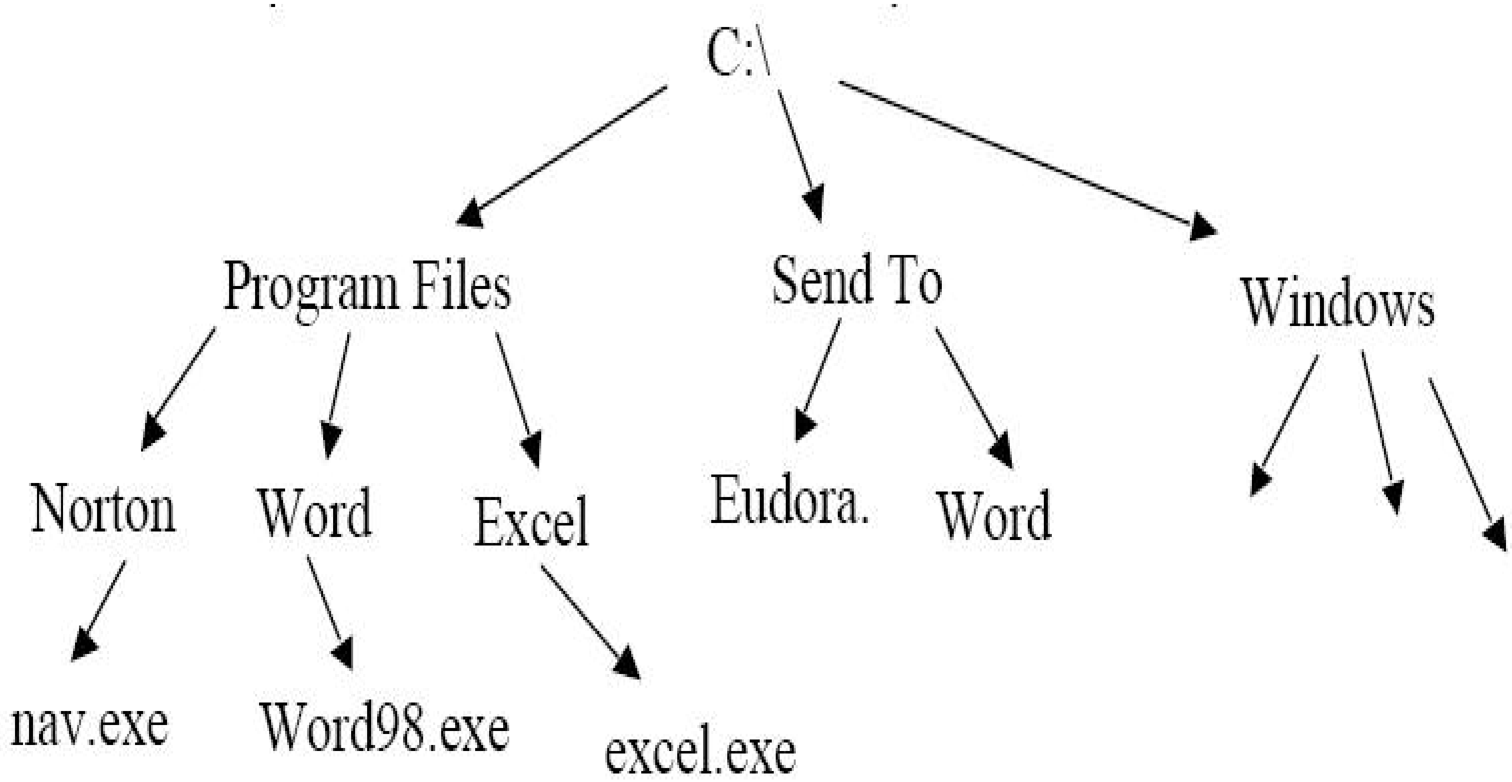
Introduction

- La structure d’arbre est l’une des plus importantes et des plus spécifiques en informatique.
- Elle est utilisée pour l’organisation des fichiers dans le SE, la représentation d’une table des matières, d’un arbre généalogique, etc…
- Dans ce cours nous intéressons aux arbres binaires

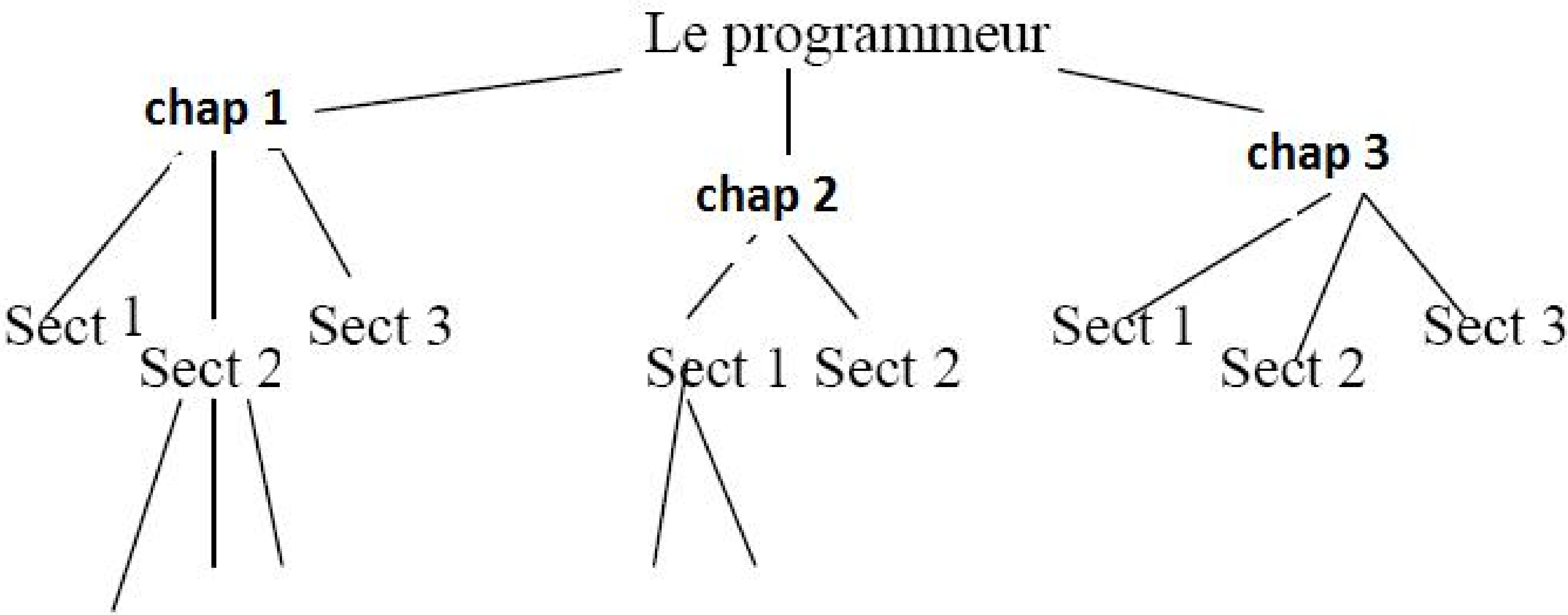
Exemple 1 : arbre généalogique



Exemple 2 : arborescence de fichiers



Exemple 3 : mise en page d'un texte



Définitions

- **Père** de x = prédécesseur du nœud x
- **Fils** de x = le ou les nœuds accrochés sous x; son successeur
  - **FG Fils Gauche de x** = un nœud accroché à sa gauche
  - **FD Fils Droit de x** = un nœud accroché à sa droite
- **Frère** de x = le fils du même père que x
- Un nœud x est **descendant** du nœud N s'il existe une succession de fils de N à x où N est l'ancêtre de x
- **sous-arbre** est un nœud avec tous ses descendants
- Une **arête** est un segment entre un nœud et son succ/préd

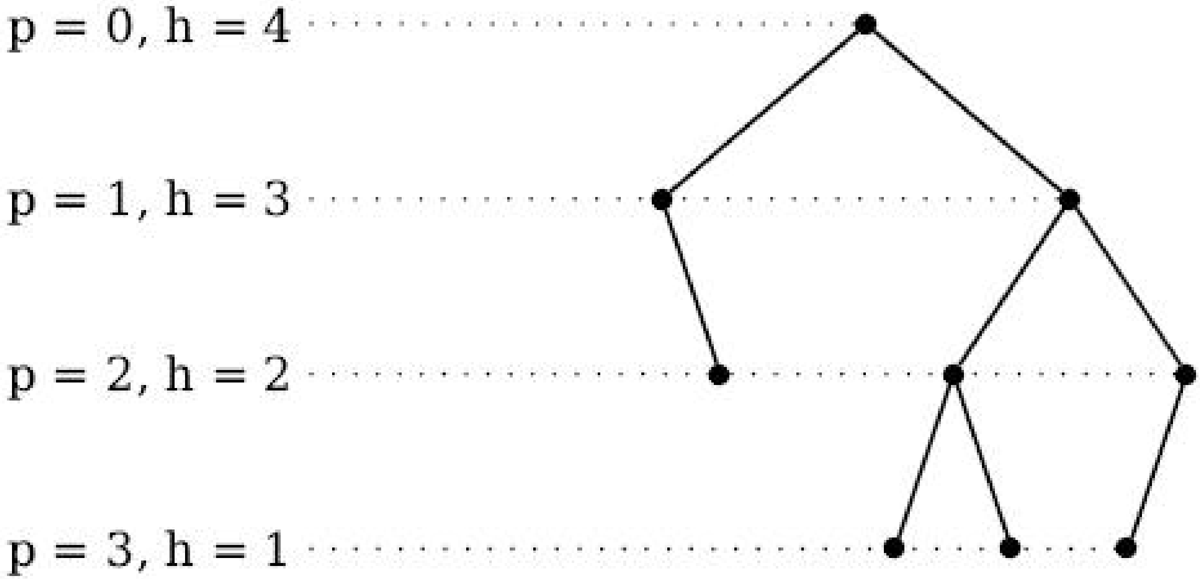
Définitions

- **racine de l'arbre =** ancêtre commun de tous les nœuds de l’arbre (le seul qui n'a pas de père).
- **Nœud interne** = un nœud qui a au moins un fils
- **Feuille =** nœud qui n’a pas de fils
- **Chemin** = une séquence d’arêtes successives
- **Branche** = chemin qui se termine par une feuille
  - **Branche gauche** = la branche de fils gauche en fils gauche
  - **Branche droite** = la branche de fils droit en fils droit

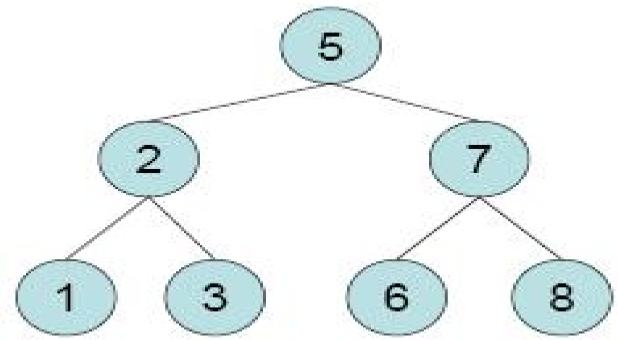
Définitions …

- **Niveau du fils = niveau du père +1**
  - La racine est au niveau 0
  - Les fils de la racine sont au niveau 1
- **Profondeur d'un nœud N** = longueur du chemin de la racine au nœud N = nombre max de nœuds dans une branche = nombre d'arêtes = nbre nœuds -1
  - **Profondeur de la racine** est 0
- **Hauteur d'un nœud N** = longueur (nombre d'arêtes) du chemin le plus long de N à une feuille
- **Hauteur d'un arbre** = hauteur de la racine = la profondeur maximale de ses nœuds **= dernier niveau (niveau feuille) +1**
- Un arbre complet de hauteur h a $2^{h+1}-1$ nœuds

Exemple



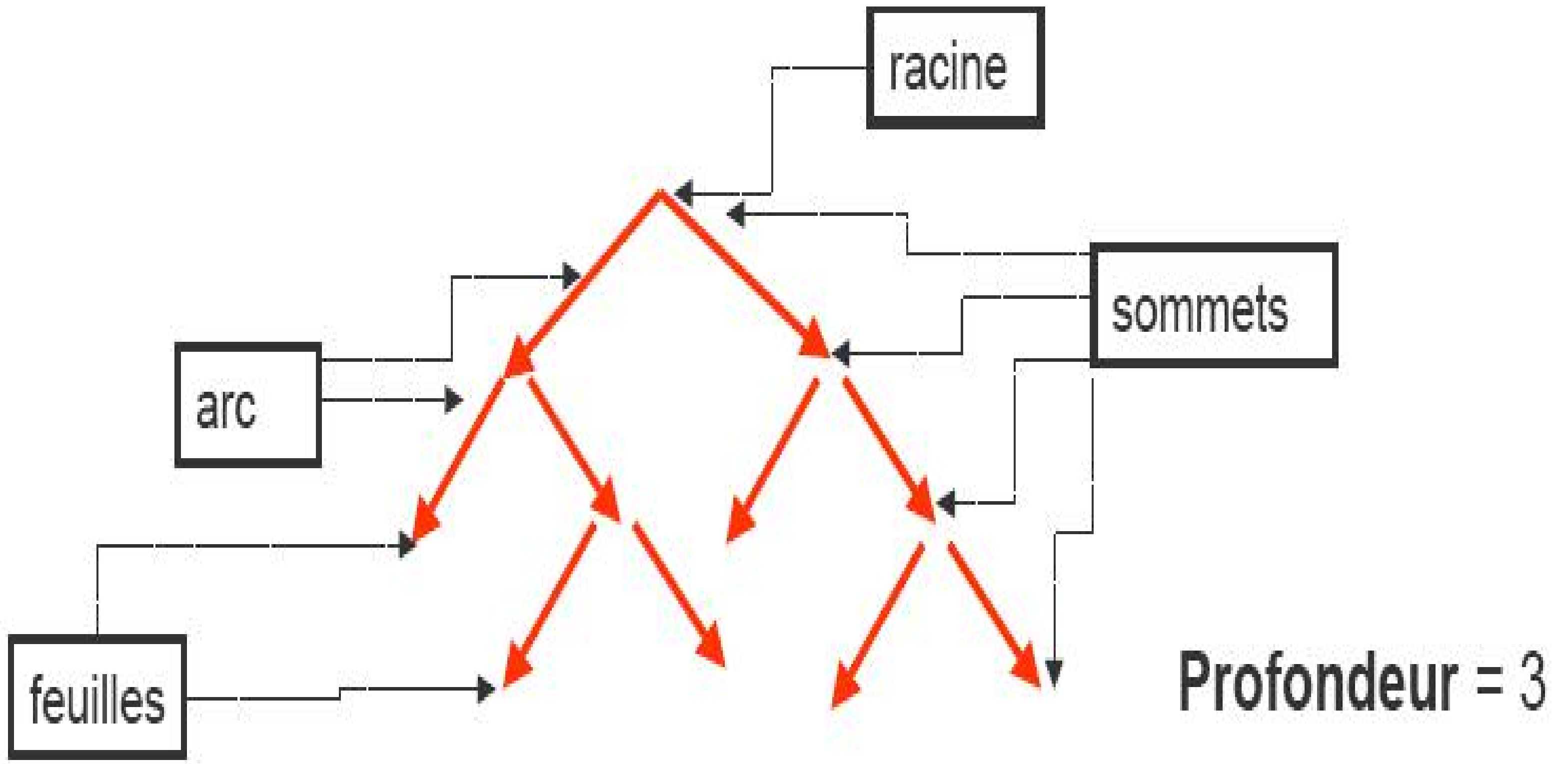
- Exemple d'arbre binaire avec hauteur et profondeur des nœuds



Arbres binaires AB

- Est un ensemble de nœuds qui est :
  - soit **vide**,
  - soit **composé d'une racine** et d'au plus deux sous –arbres binaires disjoints (un **sous-arbre droit** et un **sous-arbre gauche**)
- *Où chaque nœud a au plus 2 fils*
  - *0 fils*
  - *1 fils FG/FD*
  - *2 fils FG et FD*

Exemple



Arbres binaires complets

- C’est un arbre binaire où tous ses niveaux (à l’exception du dernier niveau) comportent le nombre maximum de nœuds
- Au niveau N ➔ 2^N nœuds

Représentation d'un AB

- Un arbre binaire non vide de racine *v est représenté par un* ***nœud qui contient :***
  - la valeur *v*,
  - l'identifiant du sous-arbre gauche,
  - l'identifiant du sous-arbre droit.
- Un arbre binaire a un identifiant qui est :
  - l'identifiant nul, si cet arbre est vide
  - l'identifiant du nœud qui contient sa racine, si cet arbre n'est pas vide,

Déclaration d’un arbre

- On indique ici une représentation par pointeur.
- Il en existe d'autres, par pointeurs, ou des représentations par tableaux.

```
Type
      Structure Nœud
    Val: info
    FG: ↑Nœud
    FD: ↑ Nœud
Fin structure
```

```
Structure  Arbre
         Racine : ↑Nœud
Finstructure
```

Opérations sur les arbres binaires

Fonction est_vide (B : arbre) :booleen

Fonction Racine (B : arbre) : ↑ nœud

Fonction est_feuille (B : arbre; f :noeud) :booleen

Fonction fils_gauche (B : arbre; x: noeud) : ↑ nœud

Fonction valeur(B : arbre; x: noeud) :entier

Fonction fils_droit(B : arbre; x: noeud) : ↑ nœud

Opérations sur les arbres binaires

Fonction est_vide (B : arbre) :booleen

Debut

Renvoyer (B.racine = NIL)

Fin

Fonction Racine (B : arbre) : ↑ noeud

Debut

Si B.racine <> NIL alors Renvoyer B.racine

Sinon renvoyer NIL

Finsi

Finsi

Fin

Fonction est_feuille (B : arbre; f :noeud) :booleen

Debut

Renvoyer (f.FG=NIL ET f.FD = NIL)

Fin

Opérations sur les arbres binaires

Fonction fils_gauche (B : arbre; x: noeud) : ↑ noeud
Debut

………………………….
Fin

Fonction valeur(B : arbre; x: noeud) :entier
Debut

…………………………………..
Fin

Fonction fils_droit(B : arbre; x: noeud) : ↑ noeud
Debut

…………………………………………
Fin

Opérations sur les arbres binaires

Fonction fils_gauche (B : arbre; x: noeud) : ↑ noeud
Debut
Renvoyer x →FG
Fin
Fonction valeur(B : arbre; x: noeud) :entier
Debut
Renvoyer x→ val
Fin
Fonction fils_droit(B : arbre; x: noeud) : ↑ noeud
Debut
Renvoyer x →FD
Fin

Nombre de sommets d'un arbre binaire

- Cas particulier : arbre vide : résultat = 0
- Cas général : 1 (sommet de l'arbre courant)
  + nb sommets dans FG
  + nb sommets dans FD

```
Fonction compteSommets(B: Arbre) : entier
début
si B→sommet = NIL alors Renvoyer 0
sinon Renvoyer (1 + compteSommets (B→sommet→gauche)
   + compteSommets (B→sommet→droit))
finsi
fin
```

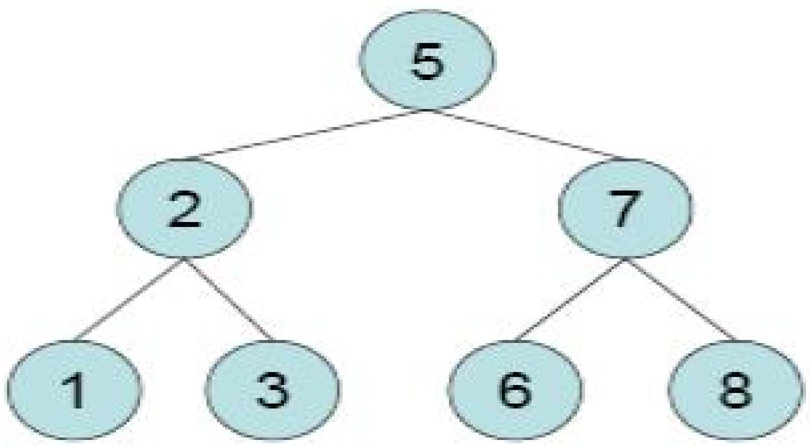
```
Fonction Père(B: arbre; x: ↑ nœud): ↑ nœud
Var tmp, p : ↑ nœud
Debut
Si est_vide(B) alors renvoyer NIL
   Sinon   p ←B.racine
      Si p = x alors renvoyer NIL
         Sinon
            Si (p→FG = x ou p→FD = x ) alors renvoyer p
             Sinon
                  tmp ←père(p→FG, x)
                  Si tmp = NIL    alors     tmp ←père(p→FD,x)
                  finsi
             Renvoyer tmp
         Finsi
      Finsi
 Finsi
Fin
```

```
Fonction Père(B: arbre; x: entier): entier
Var tmp, p : ↑ nœud
Debut
Si est_vide(B) alors renvoyer NULL
   Sinon   p ←B.racine
      Si valeur(racine(B)) = x alors renvoyer NULL
         Sinon
         Si (p→FG→valeur = x ou p→FD→valeur = x ) alors
                  Renvoyer p →valeur
             Sinon  tmp→valeur ←père(p→FG, x)
                  Si tmp →valeur = NULL alors
                            tmp→valeur ←père(p→FD,x)
                  Finsi
         Renvoyer tmp→valeur
         Finsi
     Finsi
 Finsi
Fin
```

Arbre binaire de recherche

- Sert à mémoriser de l'information qui a la propriété d'être ordonnée et à réaliser ainsi des recherches rapides => structure de données plus performante que les listes
- C'est un arbre binaire tels que pour tout nœud v de cet arbre:
  - Les éléments associés à tout nœud du sous-arbre gauche sont inférieurs ou égaux à l'élément associé à v
  - Les éléments associés à tout nœud du sous-arbre droit sont supérieurs à l'élément associé à v

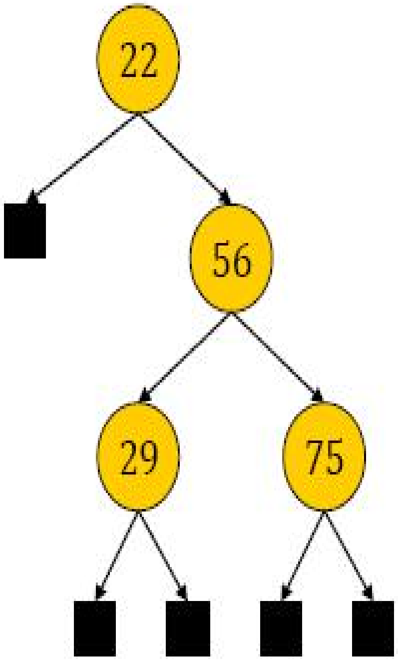
Exemple d’arbre binaire de recherche



Exercice : Dessiner les arbres binaires de recherche possibles avec les valeurs suivantes : {1,2, 3}

Arbres binaires de recherche

- Un **arbre binaire de recherche est un arbre binaire dans lequel les valeurs sont placées relativement à une relation d'ordre ≼ , de la façon suivante.**
- Pour tout sous-arbre de racine *r* :
  - les valeurs contenues dans le sous-arbre gauche, sont les valeurs *v telles que v* ≼ *r*,
  - les valeurs contenues dans le sous-arbre droit sont les valeurs *v telles que v* ≻ *r*.
- La recherche d'une valeur ne nécessite que le parcours de la branche à laquelle appartient cette valeur.



≼ est la relation ≤ sur les entiers

Création d'un arbre de recherche binaire

```
Procédure Construire(E/S  B: arbre, E: entier)
Var element : ↑noeud
Début
Si (B.racine=NIL) alors nouveau(element)
                        element↑.valeur ←E
                         element→gauche ←NIL
                         elment→droit ←NIL
Sinon si (E =< B→sommet↑.valeur) alors Construire (E,
   B→sommet→gauche)
Sinon Construire (E, B→sommet→droit)
Finsi
Finsi
Fin
```

Parcours d’un arbre binaire

- Un arbre est une structure non linéaire : une fois entrée par la racine, il est possible de visiter ses nœuds de plusieurs façons puisqu’il y a plusieurs parcours possibles de visite.
- Deux tu types de parcours:
  1. En largeur
  2. En profondeur

Parcours en profondeur

- On examine complètement un chemin et passer au chemin suivant tant qu'il en reste
- Pour traiter un nœud n, on traite d'abord tous ses descendants et on remonte ensuite pour traiter le père de n et son autre fils.

1. Parcours pré-ordre ou préfixe
2. Parcours ordre ou infixe
3. Parcours post-ordre ou postfixe

Affichage : Préordre préfixe

- On commence par la racine, puis son 1er fils, puis le 1er fils du 1er fils,

… Quand on arrive à une feuille, il faut revenir en arrière jusqu'à trouver un fils non encore parcouru.

- **Si arbre non vide alors**
  Traiter la racine
  Parcourir en préordre le sous-arbre gauche de la racine
  Parcourir en préordre le sous-arbre droit de la racine

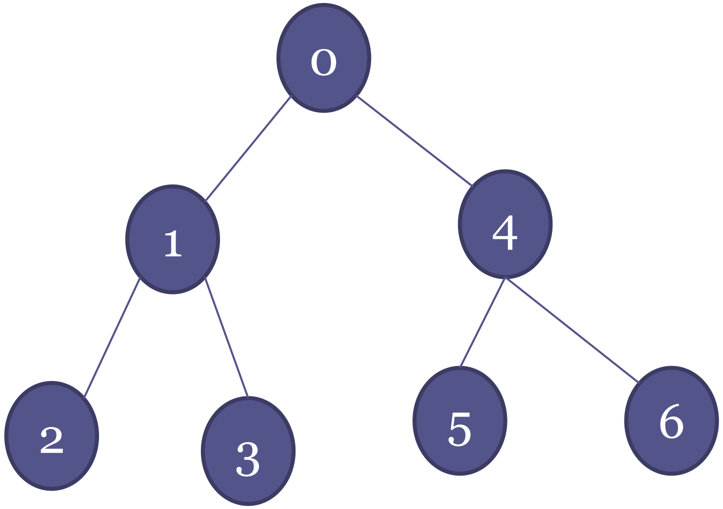
*Affiche les valeurs portées par les sommets de l'arbre binaire, en affichant la valeur portée par la racine **avant les valeurs portées par les sous-arbres** gauche et droit*

Procédure   Préordre préfixe

Procédure ParcoursPréfixe(B: arbre)
début
si B.racine <>NIL alors
    écrire (B.racine↑.valeur)
    affichePréfixe (B.racine↑.gauche)
    affichePréfixe (B.racine↑.droite)
Finsi
fin

Exercice

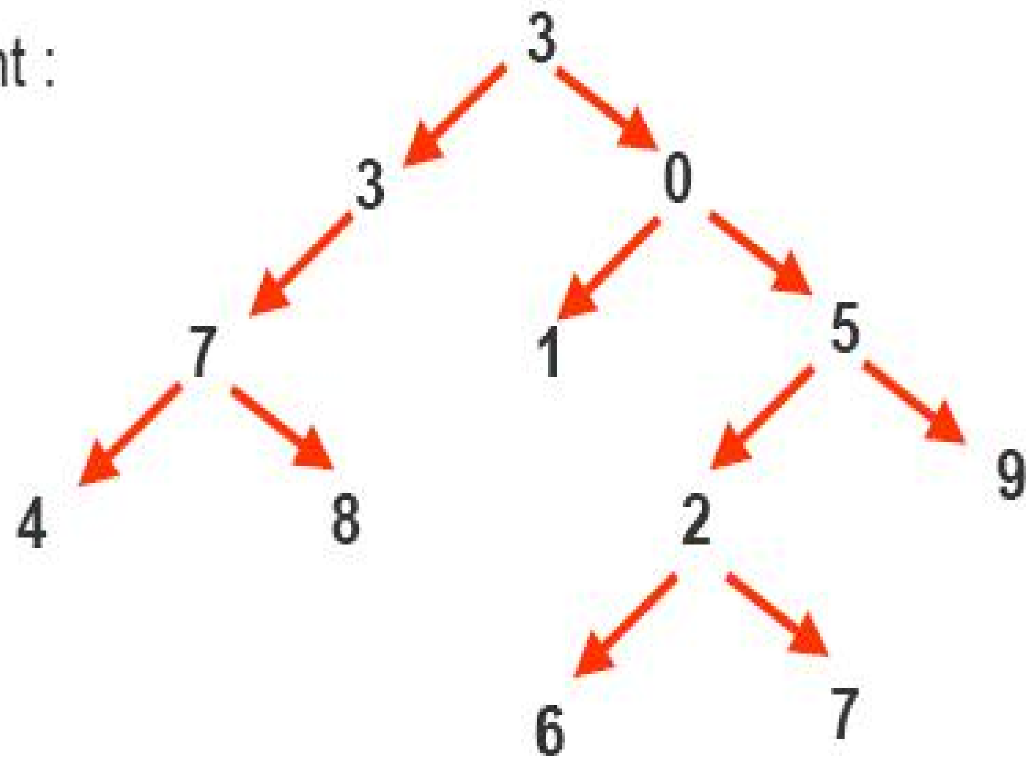
- Donner la trace d’exécution de cet algorithme pour l’affichage de l’arbre binaire suivant :



0 1 2 3 4 5 6

Exemple d’affichage : Préordre Préfixe

Soit l’arbre binaire suivant :



Affichage

- ordre préfixe : 3 3 7 4 8 0 1 5 2 6 7 9 (racine d’abord)

Affichage : Ordre Infixe

- Traite d'abord le sous-arbre gauche, puis le nœud racine puis son sous-arbre droit
- **Si arbre non vide alors**
  - **Parcourir en ordre le sous-arbre gauche de la racine**
  - **Traiter la racine**
  - **Parcourir en ordre le sous-arbre droit de la racine**

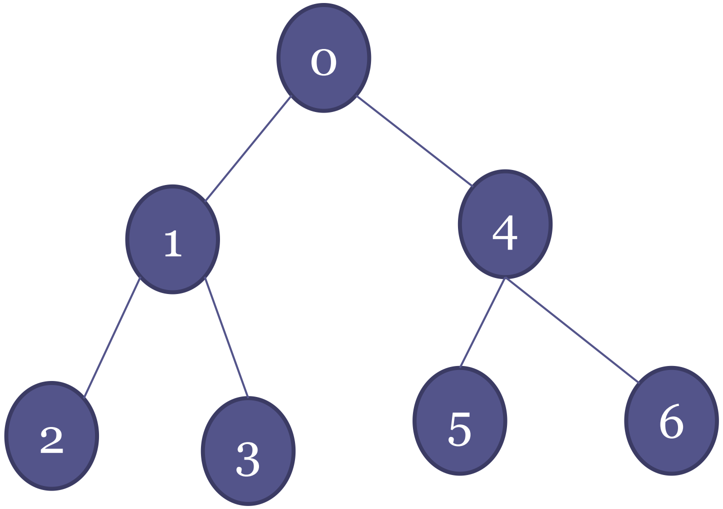
- *Affiche les valeurs portées par les sommets de l'arbre binaire, en affichant la valeur portée par la racine* ***entre les valeurs portées par les sous-arbres*** *gauche et droit*

Affichage : Ordre Infixe

```
Procédure afficheInfixe (B: arbre)
début
si B.racine <>NIL alors
                    afficheInfixe (B.racine↑.gauche)
                    ecrire (B.racine↑.valeur)
                    afficheInfixe (B.racine↑.droite)
Finsi
fin
```

Exercice

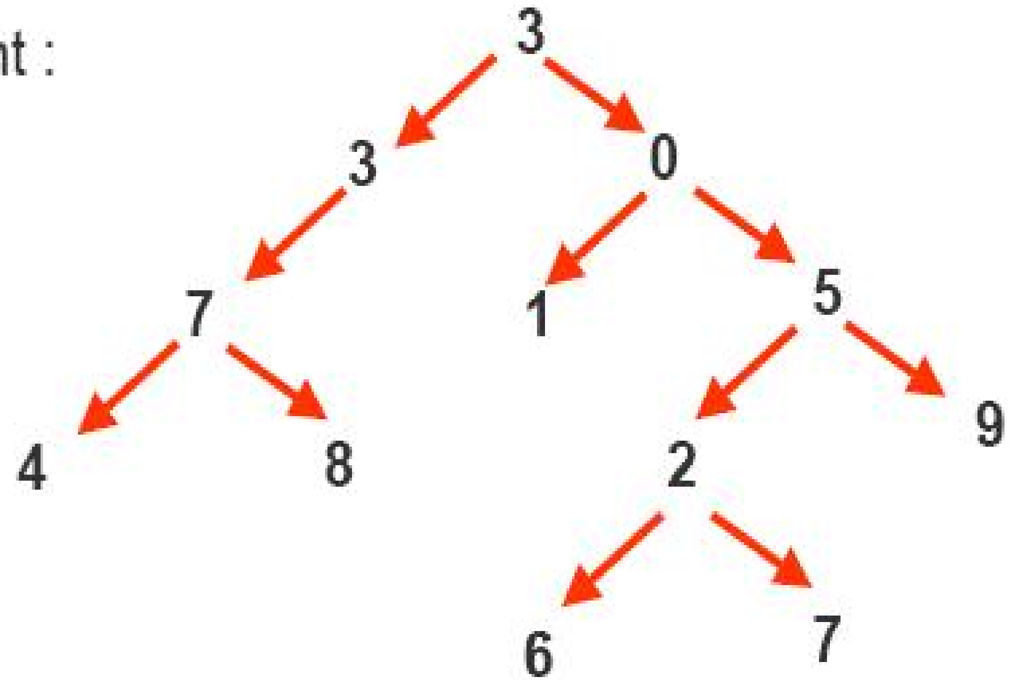
- Donner la trace d'exécution de cet algorithme pour l'affichage de l'arbre binaire suivant :



2 1 3 0 5 4 6

Exemple d'Affichage : Ordre Infixe

Soit l'arbre binaire suivant :



Affichage

- ordre infixe : 4 7 8 3 3 1 0 6 2 7 5 9 (racine au milieu)

Affichage : Postordre ou Postfixe

- *Affiche les valeurs portées par les sommets de l'arbre binaire, en affichant la valeur portée par la racine* ***après les valeurs portées par les sous-arbres*** *gauche et droit*
- **Si arbre non vide alors**
  - **Parcourir en post-fixe du sous-arbre gauche**
  - **Parcourir en postfixe du sous-arbre droit**
  - **Traiter la racine**

- traite d'abord le sous-arbre gauche, puis le sous-arbre droit, puis le noud courant.

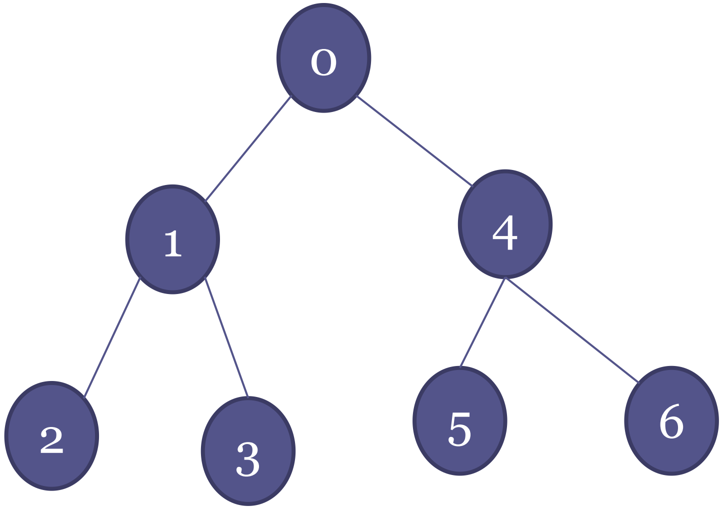
Procédure Postfixe ou post-ordre

Procédure affichePostfixe (B: arbre)
début
si B.racine <>NIL alors
    affichePostfixe (B.racine↑.gauche)
    affichePostfixe (B.racine↑.droite)
    Ecrire (B.racine↑.valeur)

Finsi
fin

Exercice

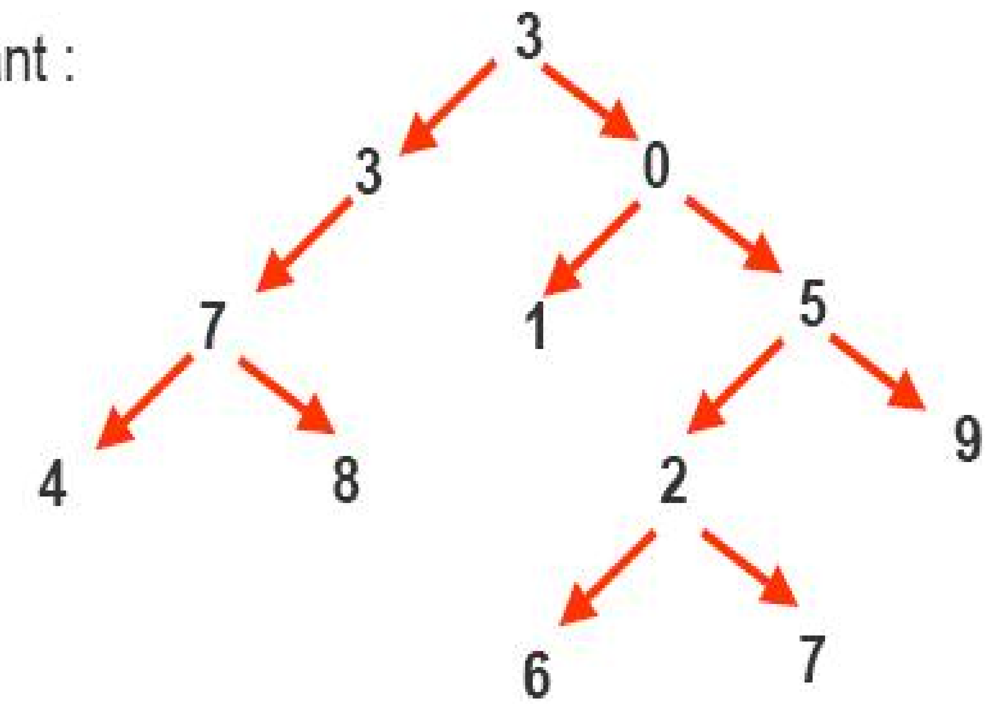
- Donner la trace d'exécution de cet algorithme pour l'affichage de l'arbre binaire suivant :



2 3 1 5 6 4 0

Affichage: Post-ordre ou postfixe

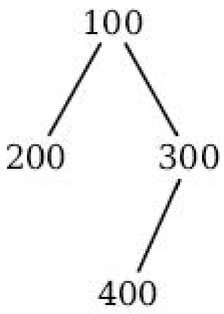
Soit l'arbre binaire suivant :



Affichage

- ordre postfixe 4 8 7 3 1 6 7 2 9 5 0 3 (racine en dernier)

Une autre représentation



Type

structure Nœud

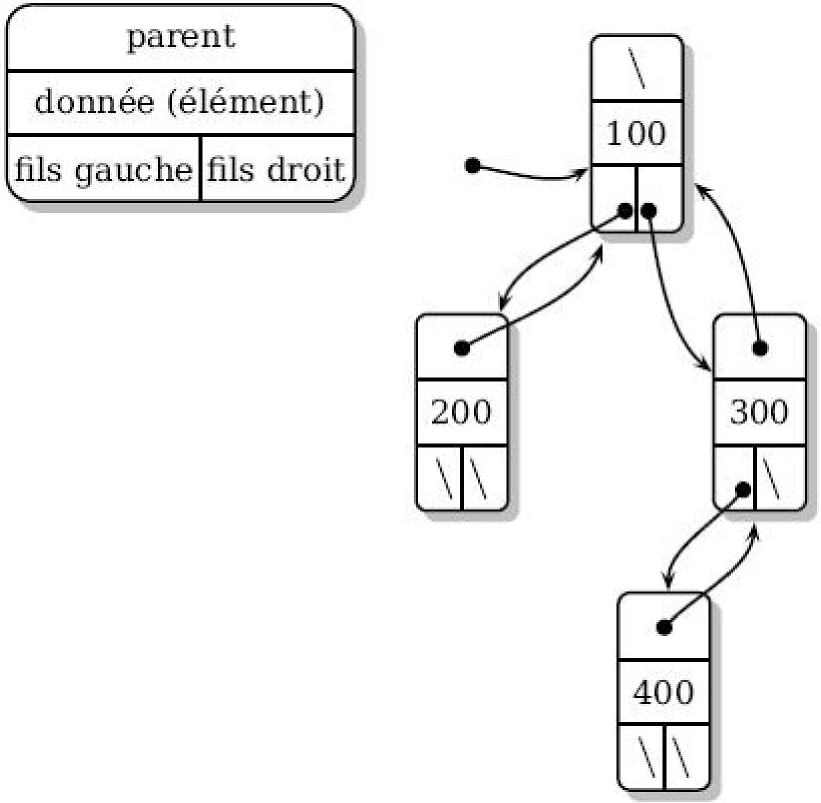
Parent:↑ Nœud

FG: ↑ Nœud

FD : ↑ Nœud

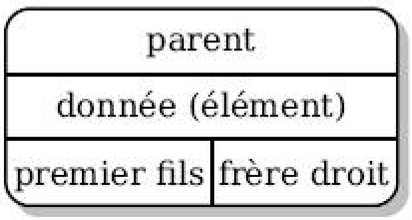
Valeur : entier

Fin structure



Une autre représentation

Représentation d'un nœud



Type

structure Nœud

Parent:↑ Nœud

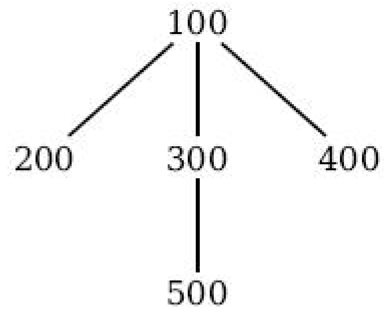
Fils : ↑ Nœud

frère : ↑ Nœud

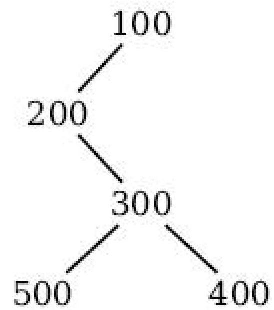
Valeur : entier

Fin structure

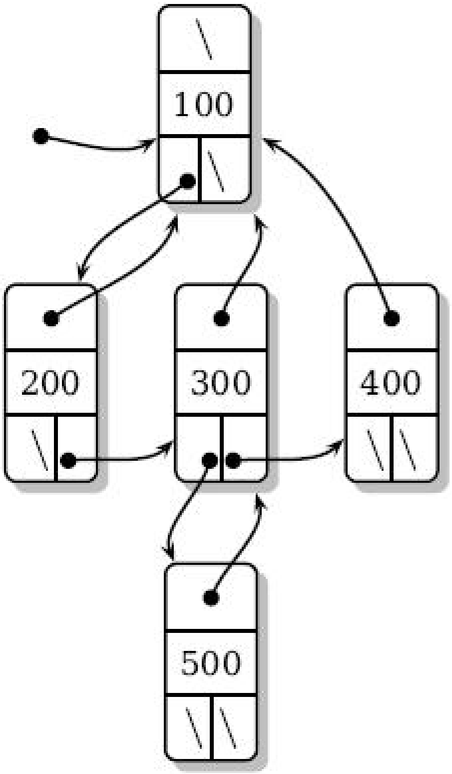
Un exemple d'arbre



Arbre binaire correspondant



Sa représentation

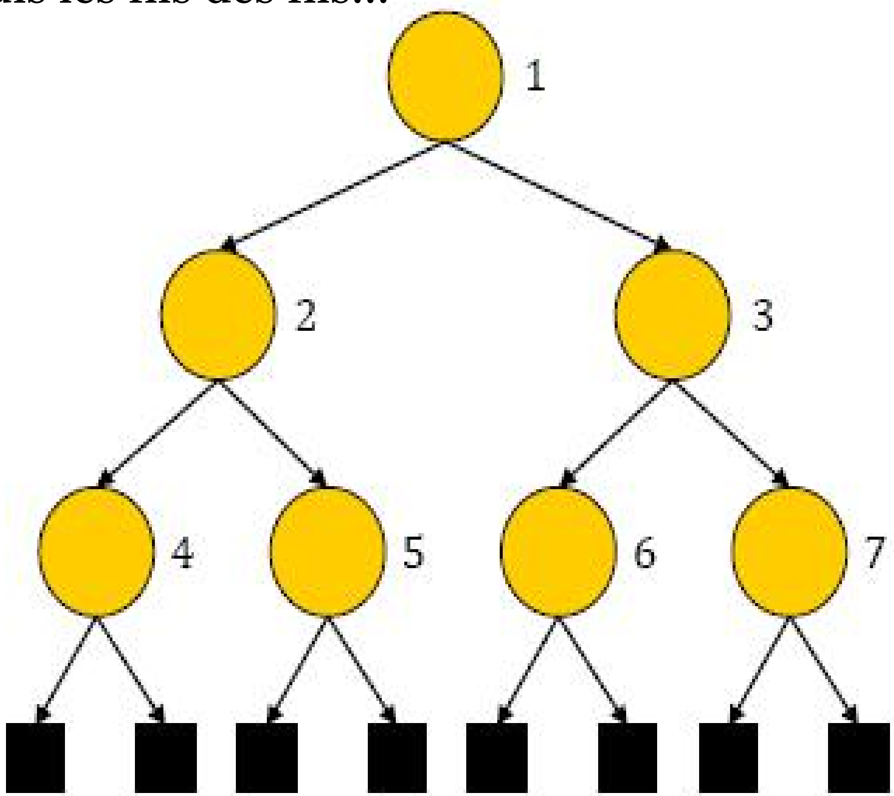


Parcours en largeur

- On examine les valeurs des nœuds niveau par niveau,
- On commence par la racine, puis on énumère les valeurs des nœuds qui sont a la distance 1 de la racine,
- puis les valeurs des nœuds qui sont a la distance 2 de la racine, etc.
- La distance entre deux nœuds est le nombre d'arcs entre ces deux nœuds.

Parcours en largeur d'un arbre binaire

On commence par la racine, puis tous ses fils, puis les fils des fils…



Parcours en largeur
niveau par niveau

parcours-en-largeur(a) =

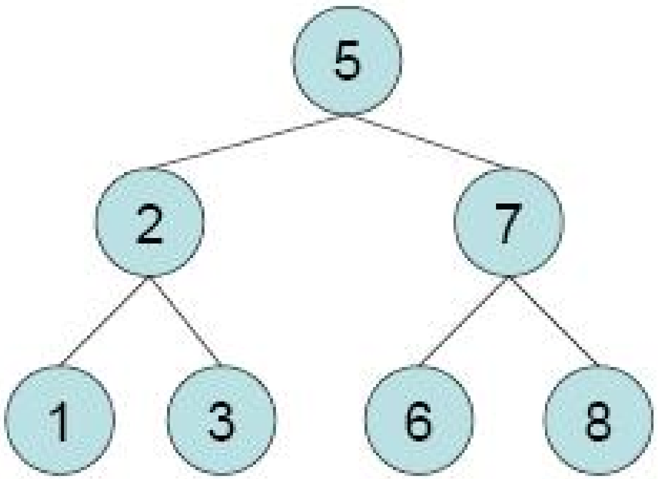
- **si** ¬*arbre-vide*(a) **alors**
- f = *file-vide*
- *entrer*(f, a)
- **faire**
  - a = *sortir*(f)
  - **traiter** *racine*(a)
  - g = *gauche*(a)
  - d = *droit*(a)
  - **si** ¬*arbre-vide*(g) **alors**
  - *entrer*(f, g)
  - **si** ¬*arbre-vide*(d) **alors**
  - *entrer*(f, d)

  jusqu'à *file-vide*(f)

Parcours par niveaux ou en largeur

```
Procedure pracours_largeur   (B: arbre)
Var f: file, Pt_noeud: ↑noeud
Début
Si B.racine <>Nil alors
Pt_noeud ←B.racine
file_vide(f)
Enfiler(f, Pt_noeud)
Tant que non file_vide(f) faire
      Pt_noeud ←Défiler(f)
      Écrire Pt_noeud ↑.valeur
      Si Pt_noeud →FG <>NIL alors enfiler (Pt_noeud →FG )
      Finsi
       Si Pt_noeud →FD <>NIL alors enfiler (Pt_noeud →FD )
       Finsi
Fintantque
Fin
```

Recherche d’une valeur



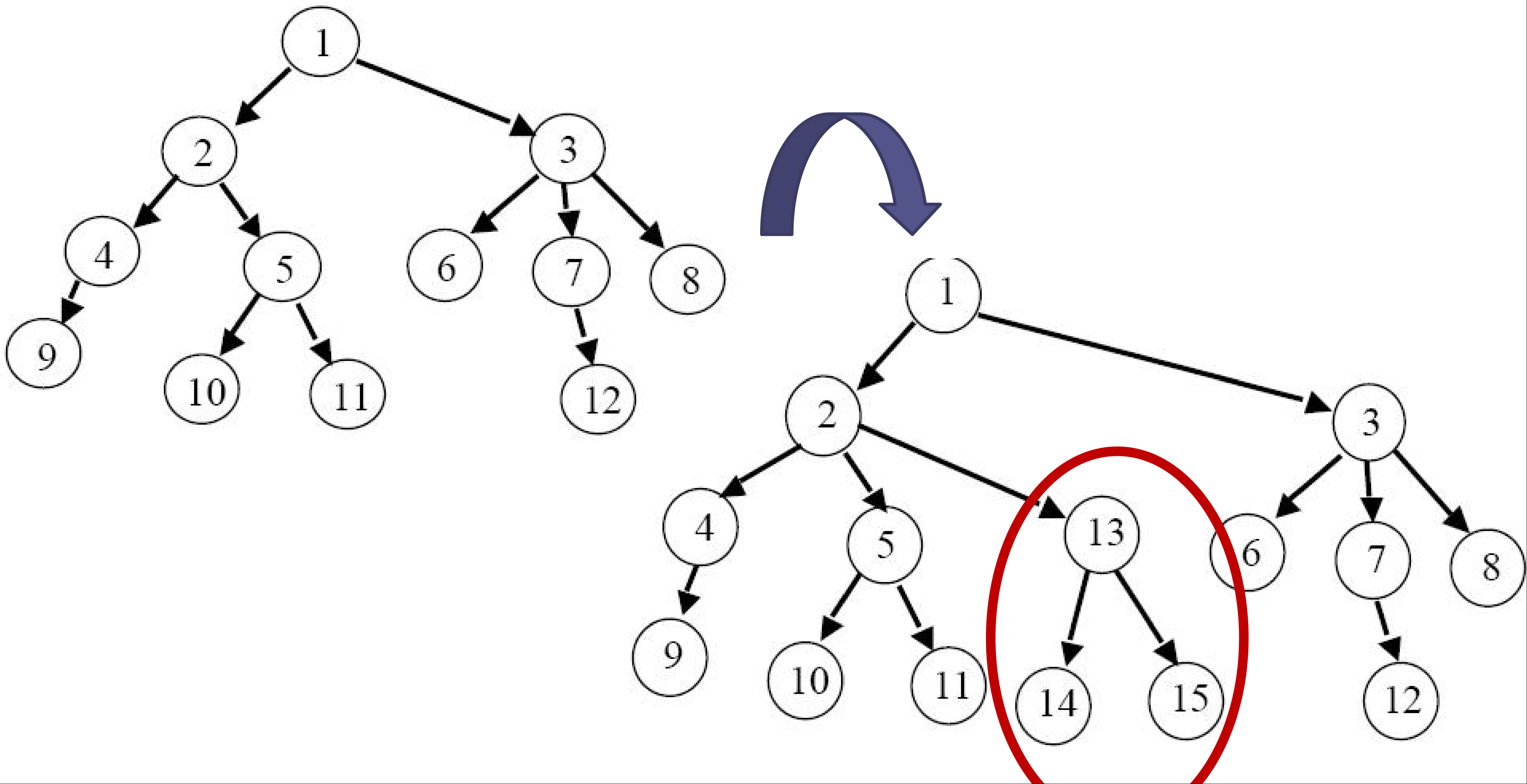
Rechercher 3: 5 → 2 → 3

Rechercher 6: 5 → 7 → 6

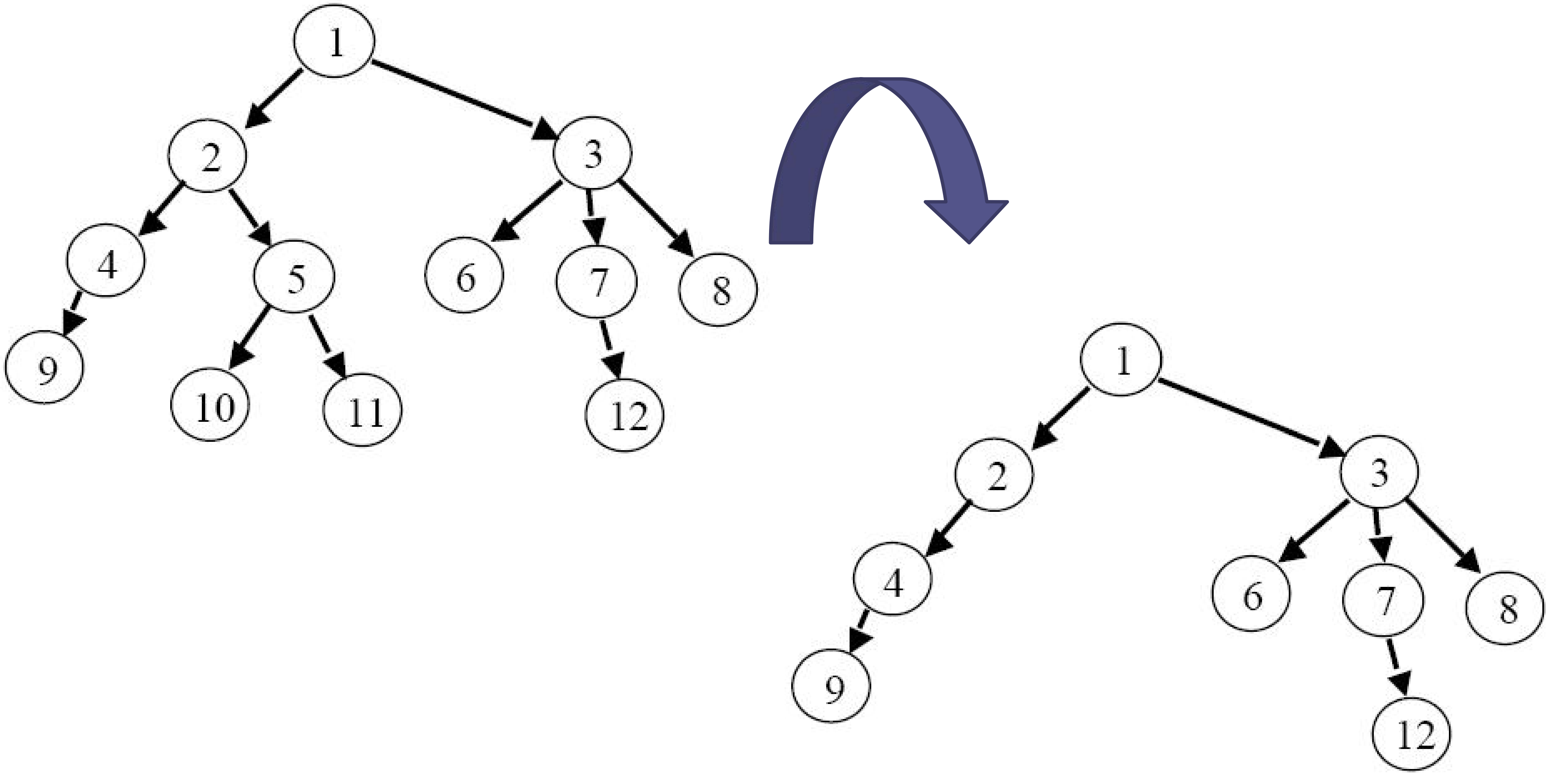
Recherche dans un AB ordonné

```
Fonction Rechercher(x:entier; B:arbre):booléen
Début
Si B.racine= NIL Alors Renvoyer Faux
   Sinon  Si x = B.racine↑.valeur Alors Renvoyer Vrai
                  Sinon Si x < B.racine↑.valeur  Alors
                            Rechercher(x, B.racine→gauche)
                       Sinon Rechercher(x, B.racine→droite)
                       Finsi
            Finsi
  Finsi
Fin
```

Ajout d'un sous arbre



Suppression d'un sous-arbre



Modification d’un arbre binaire

- On notera qu’il n’y a pas de primitives d’insertion ou de suppression de sous-arbres, ou bien de remplacement de la valeur de la racine, comme c’est le cas pour les maillons d’une liste linéaire :
- Il n’est donc pas possible de modifier une arbre « en place » .
- La solution est de construire, à partir de l’arbre à modifier, un nouvel arbre comportant qui peut partager des sous-arbres avec l'arbre initial.

Modification par reconstruction

$cons\_arbre(racine(a), gauche(a), cons\_arbre(racine(droit(a)), droit(droit(a)), arbre\_vide()));$