Les listes linéaires chaînées

① Accès Par valeur :

```
Fonction Acces_Val_LLC (tête : Ptr(maillon), Val : entier) : Ptr (maillon)
Var : P : Ptr (maillon)

Debut
    P ← tête
    tant que ((P<>Nil) et (Val <> Valeur(P)))
        P ← Suivant (P)
    FTQ
    Acces_Val_LLC ← P
Fin
```

② Accès Par Position :

```
Fonction Acces_Pos_LLC (tête : Ptr(maillon), Pos : entier) : Ptr (maillon)

Var : cpt : entier
      P : Ptr (maillon)
Debut
    P ← tête   cpt ← 1
    tant que (P<>Nil et cpt<>Pos)
        P ← Suivant (P)
        cpt ← cpt + 1
    FTQ
    Acces_Pos_LLC ← P
Fin
```

③ Nb_Occurrence d'une valeur :

```
Fonction Nb_OCC (tête : Ptr(maillon), Val : entier) : entier
Var : cpt : entier
      P : Ptr(maillon)
Debut
    cpt ← 0
    P ← tête
    Tant que (P <> Nil)
        Si (Valeur(P) = Val) alors cpt ← cpt+1
        P ← Suivant (P)
    FTQ
    Nb_Occurrence ← cpt
Fin
```

④ Maximum nb_occurrence :

```
Procedure Max_nb_OCC (tête : ptr(maillon), Var max_val : entier
    max_occ : entier)
Var : P : ptr(maillon), fréquence : entier
Debut
    P ← tête
    max_val ← Valeur(tête)
    max_occ ← nb_Occurrence (tête, max_val)
    P ← suivant(P)
    Tant que (P <> Nil)
        Si (valeur(P) <> max_val)
            fréquence ← nb_occurrence (tête, Valeur(P))
            Si (fréquence > max_occ)
                max_val ← Valeur(P)
                max_occ ← fréquence
            FSi
        FSi
        P ← suivant (P)
    FTQ
Fin
```

⑤ Accès Par valeur (2ème méthode) :

```
Procedure Acces_val_LLC2 (tête : ptr(maillon), Val : entier
Var p : ptr(maillon), Var Prec : ptr(maillon) )
Debut
    P ← tête
    Prec ← nil
    tant que (P<>Nil et Valeur(P) <>Val)
        Prec ← P
        P ← suivant(P)
    FTQ
Fin
```

Représentation d'une liste ...

⑥ Suppression dans LLC

```
Procedure (val: entier, Var tête: ptr(maillon)
Var : P, Prec, adr = ptr(maillon)
Debut
  Si (Valeur(tête) = val) alors
    P ← tête
    tête ← suivant (tête)
    liberer (P).
  Sinon
    P ← suivant (tête)
    Acces_Val_LLCE (P, Val, Prec)
    Aff-adr (Prec, suivant(adr))
    Liberer (adr)
  Fsi
Fin.
```

⑦ Fusion de deux LLC ordonnées

```
Procedure fusionLLC (tête1 : ptr(maillon), tête2 = ptr(maillon),
  Var tête : ptr(maillon)).
Var = P1, P2, P2, Q : Ptr(maillon)
    cpt = entier.
Debut
  P1 ← tête 1      P2 ← tête 2
  cpt ← 0.
  Tant que (P1 ≠ Nil ou P2 ≠ Nil)
    cpt ← cpt + 1
    Allouer (Q)
    Si (Valeur(P1) < Valeur(P2) ou P2 = Nil)
      Si (cpt = 1)
      Aff-val (Q, Valeur(P1))
      Si (cpt = 1) // Création de la tête
        tête ← Q
        P ← tête
      Sinon
        Aff-adr (P, Q)
        P ← Q
      Fsi
      P1 ← suivant (P1)
    Sinon
      Si (Valeur(P1) > Valeur(P2) ou P1 = Nil)
        Aff-val (Q, Valeur(P2))
        P2 ← suivant(P2)
      Sinon
        Aff-Val (Q, Valeur(P1))
        P1 ← suivant (P1)
        P2 ← suivant (P2)
      Fsi
    Fsi
  FTQ
Fin
```

⑧ Eclatement en 2 LLC

```
Procedure Eclatement (tête: ptr(maillon),
Var tête1 = ptr(maillon), Var tête2 = ptr(maillon))
Var P1 P2, P = ptr(maillon)
    cpt1, cpt2 = entier
Debut
  P1 ← tête1, P2 ← tête2, P ← tête
  cpt1 ← 0, cpt2 ← 0.
  Tant (P <> Nil)
    allouer (Q).
    Aff-Val (Q, Valeur(P))
    Aff-adr (Q, Nil).
    Si (Criteres (Valeur(P)))
      cpt1 ← cpt1 + 1.
      Si (cpt1 = 1)
        tête1 ← Q
        P1 ← tête1.
      Sinon.
        Aff-adr (P1, Q)
        P1 ← Q.
      Fsi
    Fsinon.
      cpt2 ← cpt2 + 1
      Si (cpt2 = 1)
        tête2 ← Q
        P2 ← tete2
      sinon.
        aff adr (P2, Q)
        P2 ← Q.
      Fsi
    Fsi
    P ← suivant (P).
  FTQ
Fin.
```

Liste bidirectionnelle :

① Création :

```
Procedure Creation_LCB (var T : ptr(maillonB) ; n : Entier)
Var : i, val : entier
      P, Q : ptr(maillonB)
Debut
    T ← NIL
    Pour i allant de 1 à n
        Allouer (Q)
        Aff_adrd (Q, Nil)
        Lire (val)
        Aff_val (Q, Val)
        Si (T = Nil) alors
            T ← Q
            Aff_adrg (Q, Nil)
            P ← T
        Sinon
            aff_adrd (P, Q)
            aff_adrg (Q, P)
        Fsi
        P ← Q
    FPour
Fin
```

② Insertion :

```
Procedure Insert_Pos_LCB (Pos : entier ; var tête : ptr(maillonB))
Var  m, Q, Prec : ptr(maillonB)
     Val : Entier
Debut
    Allouer (Q)
    Lire (val)
    Aff_val (Q, Val)
    Si (Pos = 1)
        Aff_adrd (Q, tête)
        Aff_adrg (Q, Nil)
        Aff_adrg (tête, Q)
        tête ← Q
    Sinon
        Rech_Pos (Pos, tête, m)
        Prec ← Precedent (m)
        Si (m ≠ Nil)
            Aff_adrd (Q, m)
            Aff_adrg (Q, Prec)
            Aff_adrd (Prec, Q)
            Aff_adrg (m, Q)
        Sinon
        Fsi
    Fsi
Fin
```

③ Suppression :

```
Procedure Suppr_LCB (var tête : ptr(maillonB) ;
                     Pos : entier)
Var
Debut
    Rech_Pos (Pos, tête, P)
    Si (P = tête)
        tête ← Suivant (tête)
        aff_adrg (tête, Nil)
        Liberer (P)
    Sinon
        Aff_adrd (Precedent (P), Suivant (P))
        Si (Suivant (P) <> Nil)
            Aff_adrg (Suivant (P), Precedent (P))
        Fsi
        Liberer (P)
    Fsi
Fin
```

Représentation d'une liste en contigu :

Type Element = Structure
    Val = type qq
    Vide = booléen
    Svt = entier (indice dans le tableau)
    Fin.

T = Tableau [1 .. max] de Element

Tête : entier (l'indice on commence la liste).

Allouer (var I = entier)
Liberer (I)
Aff adr (I, δ)
Valeur (I)
Suivant (I)
aff-val (I, val)

Procedure Allouer (T=tab, var I = entier, tai = entier)
Var
Debut
    i ← 1
    trouv ← faux
    Tant que (i <= tai) et (Ftrouv)
        Si T[i].vide = vrai alors
            I ← i
            trouv ← vrai
        Fsi
        i ← i+1
    FTQ
    Si (trouv = faux) alors I ← -1
Fin

Solution pour eviter le Sacs p... séquentiel :

⇒ Liste des Cases vide
⇒ Libre = entier ⇒ indice de la tete de la liste des cases vides.

Procedure Allouer (T = tab, Var I = entier)
Var
Debut
    I ← libre
    Si (libre ≠ -1) Alors
        libre ← T[libre].svt
    Fsi
Fin.

Procedure Liberer (T = tab, var I = entier)
// insertion dans la tête de la liste des Cases vides
Debut
    T[I].svt ← libre
    libre ← I
Fin

Pile en Représentation Contigue :

```
Type Pile Structure
    | T : tableau [1 ... max] de type g/q
    | Sommet : entier
Fin
```

CréerPile (P : Pile) ≡ (P.Sommet ← 0)
PileVide (P : Pile) ≡ (P.Sommet = 0)
PilePleine (P : Pile) ≡ (P.Sommet = max)
Empiler (var P : Pile, x : type g/q)
Dépiler (var P : Pile, var x : type g/q)

```
Procedure Empiler (P : Pile, x : entier)
Debut
    Si (¬PilePleine (P))
        | P.Sommet ← P.Sommet + 1
        | P.T[Sommet] ← x
    Sinon
        | Ecrire ("Pile Pleine")
    Fsi
Fin.
```

```
Procedure Depiler (P : Pile, var x : entier)
Debut
    Si (¬PileVide (P))
        | x ← P.T[Sommet]
        | P.Sommet ← P.Sommet - 1
    Sinon
        | Ecrire ("Pile Vide")
    Fsi
Fin
```

Pile en Représentation Dynamique :

Empiler : Revient à faire une insertion en tête de liste.
Dépiler : Retirer la tête donc suppression de la tête

```
Type Pile = Structure
    | Val : type g/q
    | adr : ptr (pile)
Fin.
```

CréerPile (P : Pile) ≡ P ← nil
PileVide (P : Pile) ≡ P = nil

```
Procedure Empiler (P : Pile, x : type g/q)
Debut
    Allouer (Q)
    aff-val (Q, x)
    aff-adr (Q, P)
    P ← Q
Fin
```

```
Procedure Depiler (P : Pile, x : type g/q)
Var Sauv : ptr (liste P)
Debut
    Si ¬Pile Vide (P)
        | x ← Valeur (P)
        | Sauv ← P
        | P ← Suivant (P)
        | Liberer (Sauv)
    Sinon
        | Ecrire ("Pile Vide")
    Fsi
Fin
```

Evaluation d'une operation Arithmétique :

On suppose qu'on dispose les procedures suivantes :

- operande (x) : retourne vrai si x est un operande
- binaire (x) : retourne vrai si x est un operateur binaire
- unaire (x) : retourne vrai si x est un operateur unaire
- operationb (x, y, op) : retourne x op y
- operationU (x, op) : retourne op x

```
Procedure Eval_Exp_Post (Postf = tableau[1...max] de chaine;
                         Var Resultat : réel
                         Var Erreur : Booléen)

Var P : Pile(Réel), i : int ;
Debut
  CreerPile(P)
  i ← 1
  Erreur ← faux
  Tant que (Postf[i] ≠ "." et ¬Erreur)
    si (operande Postf[i]) alors
      Empiler(P, Postf[i])
    sinon
      si (unaire Postf[i])
        si ¬PileVide(P)
          Depiler(P, x)
          x ← operationU(x, Postf[i])
          Empiler(P, x)
        sinon
          Fsi Erreur ← vrai
      sinon
        si binaire (Postf[i])
          si ¬PileVide(P)
            Depiler(P, x)
            si (¬PileVide(P))
              Depiler(P, y)
              Empiler(P, operationb(x, y, Postf[i]))
            sinon
              Erreur ← vrai
            Fsi
          sinon
            Erreur ← vrai
          Fsi
        sinon
          Erreur ← vrai
        Fsi
      Fsi
    Fsi
    i ← i + 1
  FTQ
  si (¬Erreur)
    Depiler(P, Resultat)
  sinon
    Resultat ← 0
  Fsi
Fin
```

infixée → (a+b) * c

Prefixée → * + a b c

Postfixée → a b + c *

Par

Files d'attente :

Représentation Par Flot :

```
Type File = Structure
    | T = tableau [1 .. max] d'entier
    | queue, tête = entier
Fin
```

```
Procedure CreerFile (F : File)
    | F[0] ← 0    // queue ← 0
    | F[-1] ← 1   // tete ← 1
Fin.
```

```
Fonction FileVide (F : File) : Booléen
    | si (F[-1] > F[0]) alors File_Vide ← Vrai
    | Sinon File Vide ← faux
Fin.
```

- Fonction FilePleine (F : File) : Booléen
```
    | si (F[0] = max) alors File_Pleine ← Vrai
    | sinon File Pleine ← faux
Fin
```

- Procedure Enfiler (Var F : File, x : entier)
```
    | si (¬FilePleine (F)) alors
    |     F[0] ← F[0] + 1
    |     F[F[0]] ← x
    | Sinon
    |     Ecrire ("la file est Pleine")
    | Fsi
Fin
```

```
Procedure Defiler (Var F : File, Var x : entier)   ↖ par decalage
    | si ¬FileVide (F) alors
    |     x ← F[F[-1]]
    |     // Decalage d'une Pos vers la gauche
    |     Pour i allant de 1 à F[0] - 1 /
    |         | F[i] ← F[i+1]
    |     FPR
    |     F[0] ← F[0] - 1 // avancer la queue
    | Sinon
    |     Ecrire ("File vide")
    | Fsi
Fin
```

* Representation avec un tableau Circulaire :

```
Procedure CreerFile (F : File)
    | F[0] ← max
    | F[-1] ← max
Fin
```

```
Fonction FilePleine (F : File) : Booléen
    | si (F[-1] = (F[0] mod max) + 1) alors
    |     FilePleine ← vrai
    | sinon
    |     FilePleine ← faux
    | Fsi
Fin
```

```
Procedure Enfiler (Var F : File, x = entier)
    | si (¬FilePleine (F)) alors
    |     F[0] ← (F[0] MOD max) + 1
    |     F[F[0]] ← x
    | sinon
    |     Ecrire ("la File est Pleine")
    | Fsi
Fin.
```

```
Procedure Defiler (Var F : File, Var V : entier)
Debut
    | si (¬FileVide) alors
    |     si (F[-1] = max) alors
    |         F[-1] ← 1
    |     sinon
    |         F[-1] ← F[-1] + 1
    |     Fsi
    |     V ← F[F[-1]]
    | sinon Ecrire ("File Vide")
Fin.
```

```
Fonction FileVide (F : Tab) : Booléen
    | si (F(-1) = F[0]) alors FileVide ← vrai
    | sinon FileVide ← faux
Fin.
```

* File avec Representation Dynamique LLC :

```
type File = Structure
    | queue = ptr(tmaillon)
    | tête = ptr(maillon)
    Fin.
```

Enfilement → Insertion en fin de la liste (à partir de queue)

Defilement → Suppression de la tête

```
Procedure CreerFile (var F: File)
    F.tête ← nil
    F.queue ← nil
Fin
Fonction FileVide (F: File)
    si (F.tête = nil) alors File_Vide ← vrai
    sinon File vide ← faux
Fin
Procedure Enfiler (var F: File, x: entier)
Debut
    Allouer(P)
    Aff-val(P, x)
    Aff-adr(P, Nil)
    si (1 FileVide(F)) alors
        Aff-adr(F.Queue, P)
        F.Queue ← P
    sinon
        F.tête ← P
        F.Queue ← P
Fin Fsi
```

```
Procedure Defiler (var F: File, var x: entier)
Var : P : ptr(maillon)
Debut
    si (1 FileVide(F))
        x ← Valeur(F.tête)
        P ← F.tête
        F.tête ← suivant(F.tête)
        Liberer(P)
    sinon
        Ecrire("File vide")
    Fsi
Fin.
```

* Comment implementer une File d'attente à chaque element de la File Contient un nombre variable d'entier ?

Declaration de la Structure :

```
Type ListeF = Structure
    | valF : ptr(typemaillon)
    | adrF : ptr(ListeF)
    Fin.
Type File = Structure
    | tête, queue : ptr(ListeF)
    Fin.
Type Typemaillon = Structure
    | val = entier
    | adr = ptr(typemaillon)
    Fin.
```

* File d'attente [illegible] priorité

Ajouter au modele :

Priorité(P) = recuperer la valeur du champs priorité de P

Aff-prio(P, Prio) : affectation

Declaration de la Structure :

```
Type ListeFP = Structure
    | Val = type 99
    | Prio = entier
    | adr = ptr(ListeFP)
    Fin.
Type ListeFPrio = Structure
    | tête, queue = ptr(ListeFP)
    Fin.
```

```
Procedure DefilerFlot (var F: File, var V: entier)
Debut
    si (1 File vide (F))
        V ← F[F[-1]]
        F[-1] ← F[-1] + 1
    sinon Ecrire("File vide")
Fin.
```

Files d'attente :

Représentation Statique :

① Par flot :

files avec Priorité :

```
Procedure Enfiler (varf: File, x=T_val)
Var P,q= pointeur (liste_File)
    trouv : booléen

Debut
  P← f.tête
  trouv ← faux
  Tant que( P≠nil ) et (¬trouv)
    si ( Priorité(P) > x.prio) alors
      trouv ← Vrai
    sinon
      Prec ← P
      P← suivant(P)
    Fsi
  FTQ
  Allouer(Q)
  Aff-val(q, x.val)
  Aff-prio (q,x.prio)
  Aff-adr (q,P)
  si (f.tete=nil) alors f.tête← q
  sinon aff adr(Prec, q)
  si (P=nil) alors f.queue← q
Fin
```

```
Procedure Défiler (var f :File, varx=T_val)
Var P. ptr(listeFile)
Debut
  si (¬ File vide(f)) alors
    x.val ← Valeur (f.tête)
    x.prio← priorité (f.tête)
    P← f.tête
    f.tête ← suivant (f.tête)
    Liberer(P)
  sinon
    Ecrire(" File vide")
  Fsi
Fin.
```

Les arbres :

```
Type Nbeud = Structure
    | info : type q/q
    | FG : ptr(Nbeud)
    | FD : ptr(Nbeud)
Fin.
```

```
Fonction Nbr_Nbeud (R : nbeud) : Entier
Debut
    | Si (R = nil) alors Nbr_Nbeud ← 0
    | Sinon
    |   Nbr_Nbeud ← Nbr_Nbeud(FG(R)) +
    |     Nb_Nbeud(FD(R)) + 1
    | Fsi
Fin.
```

```
Fonction Nb_feuille (R : ptr(Nbeud)) : entier
Debut
    | Si (R ≠ NIL et FD(R) = NIL et FG(R) = NIL)
    |   Nb_feuille ← Nb_feuille + 1
    | Sinon
    |   Si (R = NIL) alors Nb_feuille ← 0
    |   Sinon
    |     Nb_feuille ← Nb_feuille(FG(R)) + Nb_feuille
    |       (FD(R))
    |   Fsi
    | Fsi
Fin.
```

```
Fonction Som_Nbeud (R : ptr(Nbeud)) : entier
Debut
    | Si (R = NIL) alors Som_Nbeud ← 0.
    | Sinon
    |   Som_Nbeud ← SomNbeud(FG(R)) + SomNbeud(FD(R)) +
    |       info(R)
    | Fsi
Fin.
```

```
Fonction Nb_Nbeud_ite (R : ptr(Nbeud)) : entier
Var P : Pile (ptr(Nbeud))   Nb_N : entier
Debut
    | CreerPile (P) , Nb_N ← 0
    | Tant que (R ≠ nil ou TPileVide (P)) faire
    |   Tant que (R ≠ nil)
    |     | Nb_N ← Nb_N + 1
    |     | Empiler (P, R)
    |     | R ← FG(R)
    |   FTQ
    |   Depiler (P, R)
    |   R ← FD(R)
    | FTQ
    | Nb_Nbeud ← Nb_N
Fin
```

```
Fonction Nb_feuille_ite (R : ptr(Nbeud)) : entier
Var P : Pile (ptr(Nbeud))
    Nb_F : Entier
Debut
    | CreerPile (P)
    | Nb_F ← 0
    | Tant que (R ≠ NIL ou TPile vide (P))
    |   Tant que (R ≠ NIL)
    |     | Empiler (P, R)
    |     | Si (FD(R) = NIL et FG(R) = NIL)
    |     |   alors Nb_F ← Nb_F + 1
    |     | R ← FG(R)
    |   FTQ
    |   Depiler (P, R)
    |   R ← FD(R)
    | FTQ
    | Nb_feuille ← Nb_F
Fin
```

```
Fonction Profondeur (R : ptr(Nbeud)) : entier
Debut
    | Si (R = NIL) alors Prof ← -1
    | Sinon Prof ← 1 + max (Prof(FG(R)), Prof(FD(R)))
Fin
```

```
Type Niv_Nbeud = Structure
    | Nd : ptr(Nbeud)
    | Niv : Entier
Fin.
```

(6) maximum nb. d'occurrence :

```
Fonction Prof(R=ptr(Noeud)) : Entier
Var  Niveau, Pf = Entier
     A : Niv_Noeud
     P : pile (Niv_Noeud)
Debut
   Creer Pile(P)
   Pf ← 0
   Niveau ← 0
   Tant que (R ≠ Nil ou ¬Pile vide(P))
      Tant que (R ≠ nil)
         Niveau ← Niveau + 1
         A.Nd ← R
         A.Niv ← niveau
         Empiler(A,P)
         R ← FG(R)
         Niveau ← Niveau + 1
      FTQ
      Depiler(P,A)
      R ← A.Nd.
      Niveau ← A.Niv.
      Si (FD(R) = nil) ∧ (FG(R) = Nil) alors
         Si (Niveau > Pf) alors Pf ← Niveau
      FSi
      R ← FD(R)
      Niveau ← Niveau.
   FTQ
Fin
```

```
Fonction Strict_binaire R (R=ptr(Noeud)) : Booléen
Debut
   Si (R ≠ Nil)
      Si (FG(R) = Nil et FD(R) ≠ Nil) ou
         (FG(R) ≠ Nil et FD(R) = Nil) alors
         Strict_binaireR ← Faux
      Sinon
         Strict_binaire ← Strict_binaire(FD(R)) et
                          strict_binaire(FG(R))
      FSi
   Sinon
      Strict_binaire ← Vrai
   FSi
Fin
Fonction Strict_bin_itr (R:ptr(Noeud)) : Booléen
Debut
   Creer Pile(P) , Bin ← Vrai
   Tant que (R ≠ nil ou ¬Pile vide(P) et Bin)
      Tant que (R ≠ nil)
         Empiler(R,P)
         R ← FG(R)
      FTQ
      Depiler(R,P)
      Si ((FG(R) = Nil et FD(R) ≠ Nil) ou
         (FG(R) ≠ Nil et FD(R) = Nil)
         Bin ← Faux
      FSi
      R ← FD(R)
   FTQ
Fin Strict_Bin ← Bin
```

arbre Complet ⇔ Nb_Noeud = 2

```
Fonction Complet (R=ptr(Noeud))
Debut
   Pf ← Profondeur(R)
   Nb_N ← Nb_Noeud(R)
   Si Nb_N = puiss(2, Pf+1) - 1 alors
      Complet ← Vrai
   Sinon
      Complet ← Faux
   FSi
Fin
```

~~Fonction CompletR (R=p~~

```
Procedure RComplet (R:ptr(noeud), Comp:Booléen,
                    L=entier)
Debut
   Si R = Nil
      Comp ← Vrai
      L ← -1
   Sinon
      RComplet(FG(R), Comp1, L1)
      RComplet(FD(R), Comp2, L2)
      L ← max(L1,L2) + 1
      Comp ← (Comp1 et Comp2) et (L1=L2)
   FSi
Fin.
```

```
Inordre (N=ptr(Noeud))
Debut
  si (n ≠ nil)
    Inordre (FG(R))
    Ecrire (Info(R))
    Inordre (FD(R))
  Fsi
Fin
```

```
Procedure Preordre (N=ptr(Noeud))
Debut
  si (R ≠ Nil)
    Ecrire (Info(R))
    Preordre (FG(R))
    Preordre (FD(R))
  Fsi
Fin.
```

```
Procedure Postordre (R=ptr(Noeud))
Debut
  si (R ≠ Nil)
    Postordre (FG(R))
    Postordre (FD(R))
    Ecrire (Info(R))
  Fsi
Fin.
```

```
Procedure Inordre_itr (R=ptr(Noeud))
Var: P = Pile (Noeud)
Debut
  Creer Pile (P)
  Tant que (R ≠ Nil ou ¬Pile vide (P))
    Tant que (R ≠ Nil)
      Empiler (P, R)
      R ← FG(R)
    FTQ
    Depiler (P, R)
    Ecrire (Info(R))
    R ← FD(R)
  FTQ
Fin.
```

```
Type Sauv_Noeud = structure
    noeud : ptr(Noeud)
    Visité : Booléen
  fin
```

```
Procedure-postOrdre (R = ptr(noeud))
Var = P : Pile (Noeud)   A = Sauv Noeud
Debut
  Creer Pile (P)
  Tant que (R ≠ nil ou ¬Pile vide (P))
    Tant que (R ≠ nil)
      A.noeud ← R
      A.visité ← Vrai
      Empiler (P, A)
      R ← FG(R)
    FTQ
    Depiler (P, A)
    R ← A.Noeud
    si (A.visité = Vrai)
      A.visité ← faux
      Empiler (P, A)
      R ← FD(R)
    sinon
      Ecrire (info(R))
      R ← nil
    Fsi
  FTQ
Fin.
```

```
Fonction Rech_elt (R: ptr(Noeud), Val: entier)
Procedure                          ptr(Noeud)
```

```
Procedure Rech_elt (R: ptr(Noeud), Val: Entier,
  var Père = ptr(Noeud), var P : ptr(Noeud))

Debut
  trouv ← faux, Père ← Nil.
  tant que (R ≠ Nil et ¬trouv)
    si (info(R) = val alors trouv ← vrai
    )
    sinon
      Père ← R
      si (val > inf(R) alors R ← FD(R)
      sinon R ← FG(R)
    Fsi
  FTQ
  P ← R
Fin.
```

```
Procedure Insertion (R:ptr(Noeud), val=entier)
Debut
  si (R=Nil) alors R← CreerNoeud (val)
  Sinon
    Recherche elt(R, val, Père, P)
    si (P=Nil) alors
      CreerNoeud (Q)
      Aff-info (Q, val)
      Aff-FG (Q, Nil)
      Aff-FD (Q, Nil)
      si (val > Info(Père) alors AffFD(Père, Q)
      Sinon AffFG(Père, Q)
    FSi
  Fsi
Fin.
```

```
Procedure Rech_P_Petit_elt (R : ptr (Noeud))
  P                                  Var Q : "
                                     Var PereQ : "
  Debut
    PereQ ← Nil
    Tant que (FG(R) ≠ Nil)
      PereQ ← R
      R ← FG(R)
    FTQ
    Q ← R
  Fin.
```

```
Procedure Suppression (Var R=ptr(Noeud), Val=ent)
Var
Debut
  Rechercher (R, val, P, PereP)
  si (P≠nil)
    si (FD(P) = nil)
      si (FG(P) = nil)
        si (PereP = nil) alors R ← nil.
        Sinon
          si P = FD(PereP)
            Aff-FD(PereP, nil)
          sinon
            Aff-FG(PereP, nil)
          Fsi
          liberer(P)
      Fsi
    Sinon
      si (PereP = nil) alors R ← FG(R)
      sinon
        si P=FG(PereP) alors AffFD(PereP, FG(P))
        sinon Aff_FG(PereP, FG(P))
        liberer(P)
      Fsi
    Fsi
  sinon
    si (FG(P) = nil)
      si P = FD(PereP) alors AffFD(PereP, FD(P))
      sinon AffFG(PereP, FD(P))
    Sinon
      Q ← FD(P)
      si (FG(Q) = nil)
        Aff-info (P, info(Q))
        Aff-FD(P, FD(Q))
        Liberer(Q)
      sinon
        RechPPetitElt (Q, elt, P
        Aff info (P, info (elt))
        Aff FG (Pere elt, FD (elt))
        Liberer (elt)
      Fsi
    Fsi
  Fsi
Fin.
```

```
Procedure Preodre_itr(R=ptr(Noeud))
Debut
  CreerPile (P)
  TQ (R ≠ Nil ou ¬ Pile vide(P))
    TQ (R ≠ nil)
      Ecrire (info (R))
      si (FD(R) ≠ nil)
        Empiler (P, FD(P))
      Fsi
      R ← FG(R)
    FTQ
    Depiler(P, R)
  FTQ
Fin
```

Vérifier que
et calculer les coefficients d
En cla

```
NoeudP = Structure
    info : type qq
    FD, FG : ptr (NoeudP)
    Père : Ptr (NoeudP)
Fin.
```

Père (P) : Pointe le père du NoeudP

Aff_Père (P, Q) : Q est Père de P.

```
Fonction Svt_inordre (R : ptr(NoeudP)) :
                ptr (NoeudP).

Var
Debut
    si (FD(R) ≠ nil)
        R ← FD(R)
        Tant que (FG(R) ≠ nil)
            R ← FG(R)
        FTQ
        Svt_inordre ← R
    sinon
        Tant que (FG(Pere(R)) ≠ R et R ≠ nil)
            R ← Pere(R)
        FTQ
        Svt_inordre ← Pere(R)
    Fsi
Fin
```

```
Procedure Parcours_inordre (R : ptr(noeud))
Var
Debut
    Tant que (FG(R) ≠ nil) faire
        R ← FG(R)
    FTQ
    Tant que (R ≠ nil)
        Ecrire (info(R))
        R ← Svt_inordre (R)
    FTQ
Fin.
```

```
Procedure Suivant_Postordre (P : ptr(noeud)) ptr(noeud)
Fonction

Debut
    Q ← Pere(P)
    si Q = nil
        suivant_Postordre ← nil
    sinon
        si P = FG(Q)
            si FD(Q) ≠ nil
                P ← FG(Q)
                Tant que (7 feuille (P))
                    si (FG(P) ≠ nil)
                        P ← FG(P)
                    sinon
                        P ← FD(P)
                    Fsi
                FTQ
                suivant_Postordre ← P
            sinon suivant_Postordre ← Q
        Fsi sinon suivant Postordre ← Q
```

arbres AVL :

```
procedure equilibrer (a: ptr(Nœud))
si |h(Fg) - h(Fd)| <= 1 alors ne rien faire
sinon
    si h(Fg) - h(Fd) = 2
        si h(Fg(Fg)) > h(Fg(Fd)) alors rd(N)
        sinon rg(Fg) puis rd(N)
    sinon
        si h(Fd(Fd)) > h(Fd(Fg)) alors rg(N)
        sinon rd(Ndt) puis rg(N)
    Fsi
Fsi
Fin
```

```
Fonction ROTG (A: ptr(Nœud)) : ptr(Nœud)
Var B : ptr(Nœud)
x, y = entier
Debut
    B ← FD(A)
    x ← bal(A)
    y ← bal(B)
    FD(A) ← FG(B) // aff-FD(A, FG(B))
    FG(B) ← A // aff FG(B, A)
    FD(Père(A)) ← B /* si Père(A) ≠ Nil */
    A.bal ← x - min(y, 0) + 1
    B.bal ← max(x+2, x+y+2, y+1)
    ROTG ← B
Fin
```

```
Fonction ROTD (A: ptr(Nœud)) : ptr(Nœud)
Var B : ptr(Nœud)
    x, y = entier
Debut
    B ← FG(A)
    x ← bal(A)
    y ← bal(B)
    FG(A) ← FD(B) // aff-FG(A, FD(B))
    FD(B) ← A // aff-FD(B, A)
    FG(Père(A)) ← B
    A.bal ← x - max(y, 0) - 1
    B.bal ← min(x-2, x+y-2, y-1)
Fin
```

```
Fonction DROTG (A: ptr(Nœud)) : ptr(Nœud)
Debut
    FD(A) ← ROTD(FD(A))
    DROTG ← ROTG(A)
Fin
```

```
Procedure Equilibrer (var A: ptr(Nœud AVL), var Pere A: ptr(Nœud))
Var
    B, Q : ptr(noeud_AVL)
Debut
si (bal(A) = -2) alors
    si (bal(FD(A)) <= 0) alors
        B ← ROTG(A)
    sinon // DROTG(A)
        Q ← ROTD(FD(A))
        Aff-FD(A, Q)
        B ← RotG(A)
    Fsi
sinon
    si (bal(A) = 2) alors
        si (bal(FG(A)) >= 0) alors
            B ← ROTD(A)
        sinon
            Q ← ROTG(FG(A))
            Aff-FG(A, Q)
            B ← ROTD(A)
        Fsi
    Fsi
Fsi
si (Pere A ≠ nil) alors
    si A = FD(Pere A) alors Aff-FD(Pere A, B)
    sinon Aff-FG(Pere A, B)
sinon
    A ← B
Fsi
Fin
```

```
Procedure Insertion_AVL (x=entier, VAR A : ptr(NoeudAVL))
Var : T,P:ptr(Noeud_AVL)
      Pile_Parent = Pile

Debut
  CreerPile (pile_Parent)
  Si (A ≠ nil) alors
    Trouv ← faux
    Insert ← faux
    Tant que (¬trouv et ¬insert) faire
      si ( x=info(A) alors trouv ← vrai
      sinon
        si (x > info(A)) alors
          si (FD(A) = nil) alors
            P ← CreerNoeud (x)
            Aff-bal (P,0)
            Aff-FD (A, P)
            Aff-bal(A , bal(A)-1) // car insertion à droite
            Insert ← vrai
          sinon
            Empiler(PileParent, A)
            A ← FD(A)
          Fsi
        sinon
          si (FG(A) = nil) alors
            P ← CreerNoeud (x)
            Aff-bal(P,0)
            Aff-FG (A,P)
            Aff-bal (A, bal(A)+1)
            Insert ← vrai
          sinon
            Empiler (PileParent, A)
            A ← FG (A)
          Fsi
        Fsi
      Fsi
    FTQ

  sinon // A = nil
    A ← Creer-noeud (x)
    Aff-bal(A,0)
  Fsi
  // mise à jour des balances des Parents
  si (Insert) alors
    si (balance(A)=0) alors Fin ← vrai
                  // A aurait un fils à l'autre Coté
    sinon fin ← faux
    si (¬PileVide (Pile_Parent)) alors
      Depiler ( PileParent, T)
    sinon Fin ← vrai
    Tant que (¬PileVide (PileParent) et ¬Fin)
      si A = FD(T) alors aff-Bal (T, bal(T)-1)
      sinon aff-bal (T, bal (T)+1)
      // Equilibrer (T)
      si (bal(T) <= -2 ou bal(T) >= 2 ou bal(T)=0)   (si une est vérifiée)
        Fin ← vrai
        si bal(T) ≠ 0 alors
          si (¬PileVide (PileParent)) alors
            Depiler (PileParent, Père)
          sinon Père ← nil
          Equilibrer (T, Pere)
        Fsi
      sinon A ← T
            T ← Pere
    Fsi Fsi Fsi
```

Suppression

hachage

hachage : Essai linéaire :

```
Type Typeelt = structure
     | Clé : typeclé
     | Vide : booléen
     Fin.

Type Hashtab : tableau [1..M] de
     typeelt

Procedure Rech_Ess_Lin (k : typeclé,
T : Hashtab, Var pos : entier, Var trouv :
Booléen)
Var : I : entier.

Debut
    I ← Hacher(k)
    trouv ← faux
    Tant que (T[I].Vide = faux et ¬trouv)
    | si (T[I].clé = k) alors trouv ← vrai
    | sinon
    | | I ← I - 1
    | | si (I = 0) alors I ← M
    | Fsi
    FTQ
    Pos ← I
Fin
```

```
Procedure Insert_Esso_lin(k : typeclé,
T : Hashtab, Var Nb_elem : entier)
Var Pos : entier
    trouv : Booléen
Debut
| si (Nb_element < M - 1)
| | Rech_Ess_lin(k, T, Pos, Trouv)
| | si (¬trouv)
| | | T[Pos].clé ← k
| | | T[Pos].vide ← faux
| | | Nb_elem ← Nb_elem + 1
| | sinon
| | | Ecrire ("Element existant")
| | Fsi
| sinon
| | Ecrire ("Tableau Plein")
| Fsi
Fin.
```

```
Procedure Supp_Ess_lin(k : typeclé, T : HashTab
Var Nb : entier)
Var : trouv : Booléen ; Pos : entier ; I, R : entier
Debut
| Rech_Ess_lin(k, T, trouv, Pos)
| si (¬trouv) alors Ecrire ("Clé inexistant")
| sinon
| | T[Pos].Vide ← Vrai
| | i ← Pos - 1
| | si i = 0 alors i ← M
| | j ← pos
| | Nb ← Nb - 1
| | Tant que (T[i].Vide = faux) faire
| | | R ← Hacher (T[i].clé)
| | | si (i < j)
| | | | si (R ≤ i) ou (R ≥ j)
| | | | | T[j].clé ← T[i].clé
| | | | | T[j].vide ← faux
| | | | | T[i].vide ← Vrai
| | | | | j ← i
| | | | Fsi
| | | sinon // i > j
| | | | si (R ≥ j) et (R < i) alors
| | | | | T[j].clé ← T[i].clé
| | | | | T[j].videc faux
| | | | | T[i].vide ← Vrai
| | | | | j ← i
| | | | Fsi
| | | Fsi
| | | i ← i - 1
| | | si i = 0 alors i ← M
| | FTQ
| Fsi
Fin
```

② Double Hachage :

```
Procedure Rech_DoubleH(k:typeClé, T:HashTab,
var trouv: Booléen, var Pos: entier)
Var
Debut
    P ← R'(k)   // le pas
    i ← R(k)
    Pos ← 0
    trouv ← faux
    Tantque (T[i].vide = faux et ¬trouv)
        si (T[i].clé = K et T[i].eff = faux)
            trouv ← vrai
            Pos ← i
        sinon
            si (Pos = 0 et T[i].eff = vrai)
                Pos ← i
            Fsi
            i ← i - P
            si i ≤ 0 alors i ← i - M
        Fsi
    FTQ
    si pos = 0 alors pos ← i
Fin.
```

```
Procedure Insert_Double_Hach(k:typeclé,
T: HashTab, Var Nb: entier)
Var: trouv : Booléen
     Pos : Entier
Debut
    si (Nb < M-1) alors
        Rech_Double_Hach(k, T, trouv, pos)
        si (trouv) alors Ecrire ("Clé existante")
        sinon
            T[Pos].clé ← clé
            T[Pos]. eff ← faux
            T[Pos]. vide ← faux
            Nb ← Nb + 1
        Fsi
    Fsi
Fin.
```

```
Procedure Supp_Double_Hach (k: type clé,
T: HashTab, Var Nb: entier)
Var: trouv : Booléen
     Pos : entier
Debut
    Rech_DoubleH (k, T, trouv, pos)
    SI (trouv)
        T[Pos]. eff ← vrai
        Nb ← Nb - 1
    Fsi
Fin.
```

avec chaînage interne

```
Procedure recherche_chain_int(k:typecle; T, tab, var trouv: Booléen, var i: entier,
var prec: booléen)
Debut
  Trouv ← faux
  i ← h(k)
  Prec ← 0
  Tant que (i ≠ 0 et T[i].vide = faux)
  Faire
    Prec ← i
    i ← T[i].lien
  FTQ
  si (T[i].vide = vrai)
  | Pos ← i
  sinon
  | Tant que (Ftrouv et i ≠ 0)
  | | si (T[i].clé = k) alors trouv ← vrai
  | | sinon
  | | | Prec ← i
  | | | i ← T[i].lien
  | | Fsi
  | FTQ
  | si (Ftrouv) alors pos ← prec
  Fsi
Fin
```

```
Procedure Insertion (tab: HachTab, taille: entier,
  Val: typecle)
  Var Pos: entier
      trouv: Booléen
Debut
  Rech_ch (Val, Tab, Pos, trouv)
  si (trouv) alors
  | Ecrire ("la clé existe")
  sinon
  | si (Tab[Pos].vide = vrai) alors
  | | T[Pos].clé ← k
  | | T[Pos].lien ← 0
  | | T[Pos].vide ← vrai
  | sinon
  | | r ← M - 1
  | | Tant que (r ≠ 0 et Tab[r].vide = faux)
  | | | r ← r - 1
  | | FTQ
  | | si (r = 0) alors
  | | | Ecrire ("Tab Plein")
  | | sinon
  | | | Tab[r].clé ← Val
  | | | Tab[r].vide ← faux
  | | | Tab[Pos].lien ← r
  | | | Tab[r].lien ← 0
  | | Fsi
  | Fsi
  Fsi
Fin.
```

```
Procedure Suppchainint(k: typecle, T: tabhash
  Var R: entier)
Debut
  Rech_chain_int (k, T, trouv, Pos, Prec)
  si (Ftrouv)
  | Ecrire ("clé inexistante")
  sinon
  | fin ← faux
  | tant que (Ffin) faire
  | | J ← Pos
  | | Tant que (Pos ≠ 0 et h(T[Pos]) ≠ J)
  | | | P ← Pos
  | | | Pos ← T[Pos].lien
  | | FTQ
  | | si (Pos ≠ 0)
  | | | T[J].clé ← T[Pos].clé
  | | sinon
  | | | fin ← Vrai
  | | Fsi
  | FTQ
  | si (R = T[J].clé)
  | | Precedent ← Prec
  | sinon
  | | Precedent ← P
  | Fsi
  | si (Precedent ≠ 0) alors
  | | T[Precedent].lien ← T[J].lien
  | sinon T[J].lien ← 0
  | | T[J].vide ← Vrai
  Fsi
Fin.
```

Hachage avec chainage externe (séparé)

```
Procedure Rech_chain_Sep (T:tab,
D:entier, Var i:entier, Var trouv:Booléen,
Var j: ptr(maillon))

Debut
    I ← R(D)
    trouv ← faux
    si (T[I].clé = d)
        trouv ← vrai
    Sinon
        si (T[I].lien ≠ nil)
            j ← T[I].lien
            Tant que (j≠nil et ¬trouv)
                si (valeur(j)=d)
                    trouv ← vrai
                sinon
                    j ← suivant(j)
                Fsi
            FTQ
        Fsi
    Fsi
Fin.
```

```
Procedure Insert_chain_Sep (V:entier, T:tab)
Var : i:entier, trouv:booléen,
      j : ptr(maillon)
Debut
    i ← R(V)
    Rech_chain_Sep(T,V,I,trouv,j)
    si (trouv) alors Ecrire("La valeur existe")
    sinon
        si (T[I].vide) alors
            T[i].clé ← v
            T[i].vide ← faux
        sinon
            si (T[i].lien = Nil) alors
                Allouer(P)
                aff-val(P,V)
                aff-adr(P, nil)
                T[i].lien ← P
            sinon
                Allouer(P)
                Aff-val(P,V)
                aff adr(P, T[i].lien)
                T[i].lien ← P
            Fsi
        Fsi
    Fsi
Fin.
```

```
Procedure Supp_chain_Sep(T:tab, d:entier)
Debut
    i ← R(d)
    Rech_chain_sep (T, V, i, trouv, j)
    si (¬trouv)
        Ecrire (" Clé inexistante")
    sinon si (T[i].clé = V
        si (T[i].lien = nil)
            T[i].vide ← Vrai
        sinon
            P ← T[i].lien
            T[i].clé ← valeur(T[i].lien)
            T[i].lien ← suivant(T[i].lien)
            Liberer(P)
        Fsi
    sinon si (T[i].clé ≠ V) // dans la liste
        Q ← T[i].lien
        Tant que (suivant(Q) ≠ j)
            Q ← suivant(Q)
        FTQ
        Aff-adr(Q, suivant(j))
        Liberer(j)
    Fsi
Fsi
Fin.
```