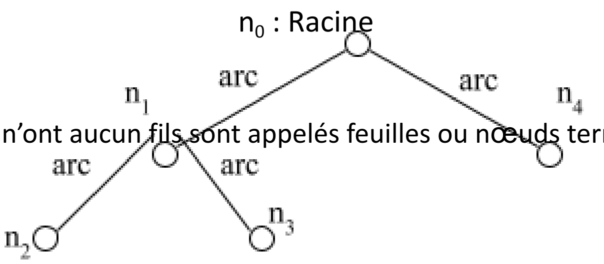
Les Arbres

Définitions :

a) Arbre :

L'arbre est une structure de donnée récursive constituée :

- d'un nœud particulier appelé *racine*,
- d'un ensemble de couples (n_i, n_j) reliant le nœud n_i au nœud n_j appelés *arcs* (ou arêtes). Le nœud n_i est appelé *père* de n_j. Le nœud n_j est appelé *fils* de n_i.



b) Feuilles :

Les nœuds qui n'ont aucun fils sont appelés feuilles ou nœuds terminaux (les nœuds n_2, n_3 et n_4 sont des feuilles).

c) Chemin :

On appelle chemin la suite de nœuds n_0 n_1 … n_k telle que (n_{i-1}, n_i) est un arc pour tout i ϵ {1,…, k}. L'entier k est appelé longueur du chemin n_0 n_1 … n_k.
k c'est aussi le nombre d'arcs.

Le nombre d'arcs d'un arbre = nombre de nœuds - 1.

d) Sous-arbre :

Les autres nœuds (sauf la racine n_0) sont constitués de nœuds fils, qui sont eux même des arbres. Ces arbres sont appelés sous-arbres de la racine.
Exemple : Les nœuds n_1, n_2 et n_3 constituent un sous-arbre.

e) Hauteur :

La hauteur d'un nœud est la longueur du plus long chemin allant de ce nœud jusqu'à une feuille. La hauteur d'un arbre est la hauteur de la racine (nombre de nœuds).

f) Niveau ou profondeur :

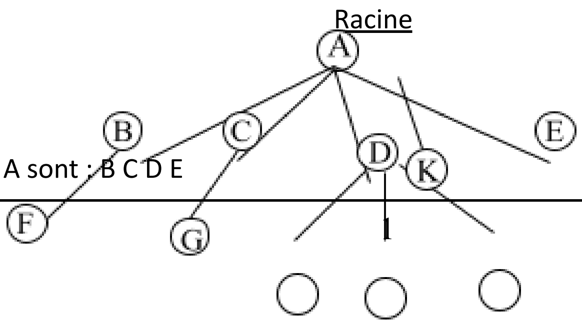
La profondeur d'un nœud est la longueur du chemin allant de la racine jusqu'à ce nœud. Tous les nœuds d'un arbre de même profondeur sont au même niveau.
Exemple : Les nœuds n_1 et n_4 ont la même profondeur et sont donc au même niveau.

g) Ascendance et descendance :

Soit deux nœuds ***a*** et ***b.*** S'il existe un chemin reliant le nœud ***a*** au nœud ***b,*** on dit que ***a*** est un ascendant de ***b*** ou ***b*** est un descendant de ***a***.

Exemple récapitulatif:

niveau 0



niveau 1

La racine c'est : A

niveau 1

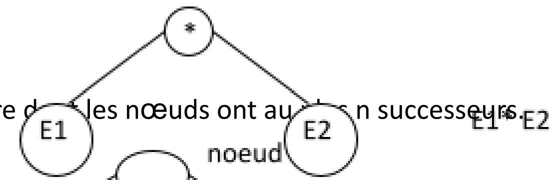
Les nœuds fils de A sont : B C D E

niveau 2

Le nombre de sous-arbres = 4
Le père de F c'est B
B est un ascendant de F
F est un descendant de B
Les feuilles de l'arbre sont : F G H K J E
La hauteur de l'arbre = 4 (nombre de nœuds)
La longueur du chemin A-F = 2 (nombre d'arcs)
La profondeur de l'arbre = 3
La profondeur du nœud G = 2

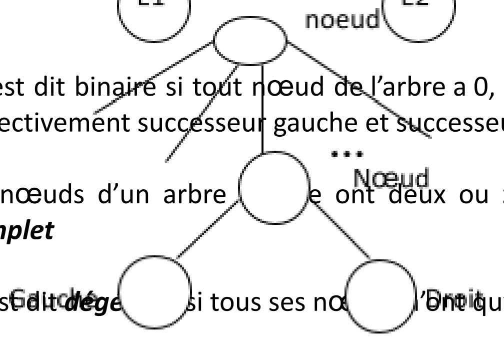
Un arbre peut aussi être représenté sous forme parenthésée :
(A (B(F), C(G), D(H, I(K), J), E))

h) Arbre étiqueté:
Un arbre étiqueté est un arbre dont chaque nœud possède une information ou étiquette. Cette étiquette peut être de nature très variée : entier, réel, caractère, chaîne, pointeur,… ou une structure complexe.
Exemple : on peut représenter une expression arithmétique par un arbre



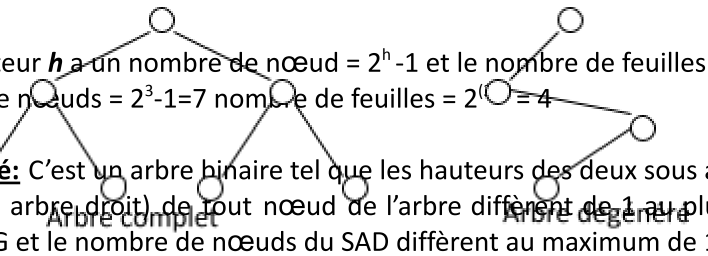
i) Arbre n-aire :
Un arbre n-aire est un arbre dont les nœuds ont au plus n successeurs.

ii) Arbre binaire :
Un arbre binaire est dit binaire si tout nœud de l'arbre a 0, 1, ou 2 successeurs. Ces successeurs sont alors appelés respectivement successeur gauche et successeur droit.



Lorsque tous les nœuds d'un arbre binaire ont deux ou zéro successeurs ont dit que l'arbre est ***homogène*** ou ***complet***

Un arbre binaire est dit ***dégénéré*** si tous ses nœuds n'ont qu'un seul descendant.



Un arbre complet de hauteur ***h*** a un nombre de nœud = 2^h -1 et le nombre de feuilles est $2^{(h-1)}$.
Exemple : *h*=3 nombre de nœuds = 2^3-1=7 nombre de feuilles = $2^{(3-1)}$ = 4

iii) Arbre binaire équilibré: C'est un arbre binaire tel que les hauteurs des deux sous arbres SAG, SAD (sous arbre gauche, sous arbre droit) de tout nœud de l'arbre diffèrent de 1 au plus. Ou encore le nombre de nœuds de SAG et le nombre de nœuds du SAD diffèrent au maximum de 1.

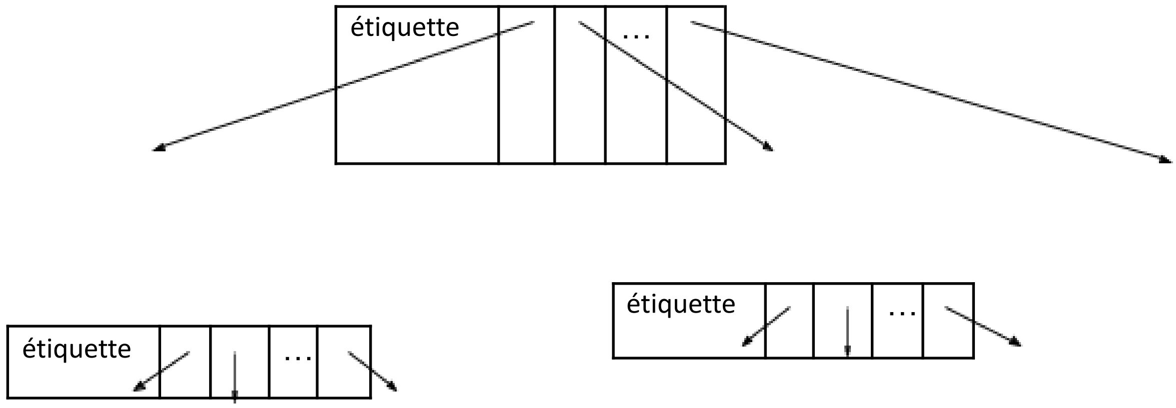
Représentation chaînée des arbres en mémoire
Arbre n-aire :

a) Une manière de représenter un arbre est d'associer à chaque nœud un enregistrement contenant un ou plusieurs champs pour coder l'étiquette et d'un tableau de pointeurs vers les nœuds fils. La taille du tableau est donnée par le nombre maximum de fils des nœuds de l'arbre.

étiquette	P1	P2	…	P3

Déclaration :

Type <u>arbre</u> : ^ objet ; Type objet : Enregistrement tab [max_fils] : <u>arbre</u> ; inf : <type_elt> ; Fing ;	typedef struct objet *arbre ; typedef struct objet { arbre tab [max_fils] ; <typel_elt> inf ; } nœud ;

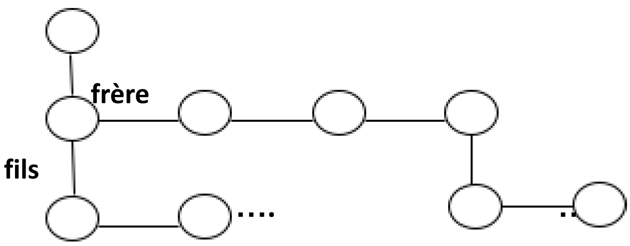


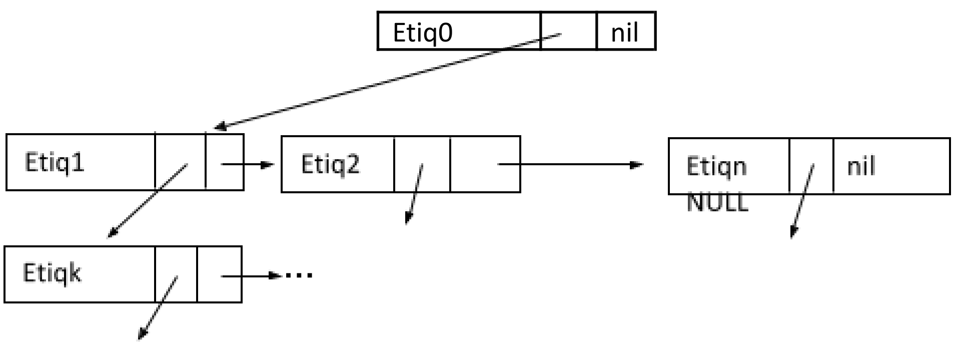
Inconvénients :

- l'arbre contient un petit nombre de nœuds ayant beaucoup de fils. (tableaux de grandes tailles)
- L'arbre contient beaucoup de nœuds ayant peu de fils. (plusieurs tableaux)

Ceci conduit à consommer beaucoup d'espace mémoire.

b) Avec deux pointeurs fils et frère.
Afin de contourner l'inconvénient du tableau, on utilise un pointeur vers fils ainée et chaque fils possède un lien vers son frère le plus proche.

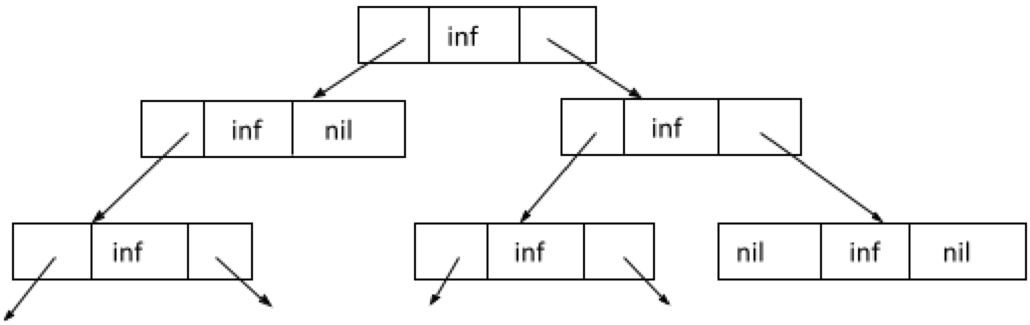




Arbre binaire :

Déclaration :

Type <u>arbre</u> : ^ objet ;
Type objet : Enregistrement
    fg, fd : <u>arbre</u> ;
    inf : <type_elt> ;
Fing ;
	typedef struct objet *arbre ;
	typedef struct objet {
	arbre fg, fd ;
	<type_elt> inf ;
	} nœud ;

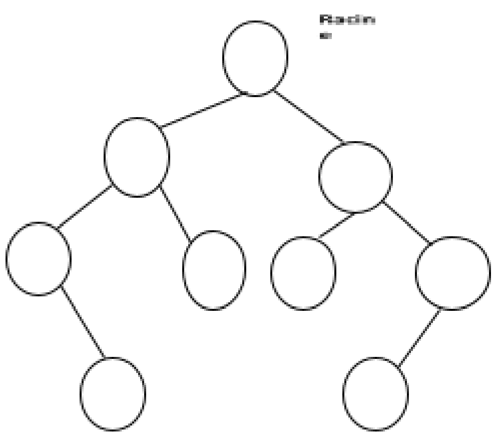


Représentation contigüe d'un arbre binaire en mémoire

- Un arbre binaire peut être représenté sous forme d'un vecteur où chaque élément sera composé de **trois** champs : *un champ info* représentant l'information contenue dans le nœud, *un champ succ_gauche* donnant la position du fils gauche dans le vecteur et *un champ succ_droit* donnant la position du fils droit dans le vecteur.

Type nœud = Enregistrement
  info : <type_elt> ;
  fg, fdt : <u>entier</u> ;
Fing ;
 	
V : tableau [taillemax] <u>nœud</u> ;
	typedef struct noeud
	  { <type_elt> info ;
	    int fg, fd ;
	  } noeud ;
	
	nœud V[taillemax] ;

<u>*Exemple d'u*n vecteur de caractères :</u>



Le vecteur correspondant à l'arbre sera comme suit :

A	2	3	**B**	4	5	**C**	6	7	**D**	0	8	**E**	0	0	**F**	0	0	**G**	9	0	**H**	0	0	**I**	**0**	**0**

La valeur 0 indique que le nœud n'a pas de successeur gauche et/ou droit. La racine étant l'élément d'indice 1 du tableau.

- ***Représentation séquentielle :***

Dans un vecteur ayant le type de l'information contenu dans un nœud de l'arbre. La racine est rangée dans t[0] (langage C).
Si un nœud se trouve à la position **k** du vecteur, son successeur gauche doit se trouver à la position **2*k+1** et son successeur droit à la position **2*k+2**.

Exemple :
Reprenons toujours le même exemple. Le vecteur sera déclaré comme suit :
t : tableau [taillemax] caractère ;
La taille est aussi limitée mais plus petite que dans la première représentation.

A	B	C	D	E	F	G		H					I						

Inconvénient: dans cette représentation, le vecteur contient des trous. Sauf s'il s'agit d'un arbre binaire complet.

Remarque : dans la suite de ce cours, nous considérons les arbres binaires avec la représentation chaînée.

Primitives de manipulation d'un arbre binaire

Vérifier si un nœud a est vide

Fonction EstVide (E/ a : Arbre) : booléen
Début
 si (a = nil) alors retourner (vrai) ;
 sinon retourner (faux) ;
 fsi ;
Fin ;

Retourner l'adresse du fils gauche d'un nœud a

Fonction Fils_Gauche (E/ a : Arbre) : Arbre
Début
 retourner (a^.fg) ; //a est ≠ de nil
Fin ;

Retourner l'adresse du fils droit d'un nœud a

Fonction Fils_Droit (E/ a : Arbre) : Arbre
Début
 retourner (a^.fd) ; //a est ≠ de nil
Fin ;

Vérifier si un nœud a est une feuille

Fonction EstFeuille (E/a : Arbre) : booléen
Début //a est ≠ de nil
si (EstVide(Fils_Gauche (a)) et EstVide(Fils_Droit(a)))
 alors retourner (vrai) ;

sinon retourner (faux) ;
fsi ;
Fin ;

Nombre de nœuds de l'arbre
Fonction nb_noeud(E/ a : Arbre) : entier
Début
si (EstVide(a))
  alors retourner (0);
  sinon retourner (1+nb_noeud(Fils_Gauche (a))+ nb_noeud(Fils_Droit(a)));
fsi ;
Fin ;

Parcours d'un arbre binaire

Il existe deux manières de parcourir un arbre en général et un arbre binaire en particulier :

- parcours en largeur : niveau par niveau
- parcours en profondeur ----> parcours préfixé (préordre)
  ----> parcours infixé (ordre)
  ----> parcours postfixé (post-ordre)

Parcours en profondeur

Il existe trois algorithmes de parcours d'un arbre binaire en profondeur : un parcours **préfixé** (ou préordre), un parcours **postfixé** (ou post-ordre) et un parcours **infixé** (ou ordre).

a- **Parcours préfxé d'un arbre binaire**

On commence par traiter (afficher, maj, ...) la **racine**. On effectue ensuite un parcours préfixé de tous les nœuds de A1 (**fils gauche**) jusqu'aux feuilles. On revient pour effectuer ensuite un parcours préfixé de tous les nœuds de A2 (**fils droit**).

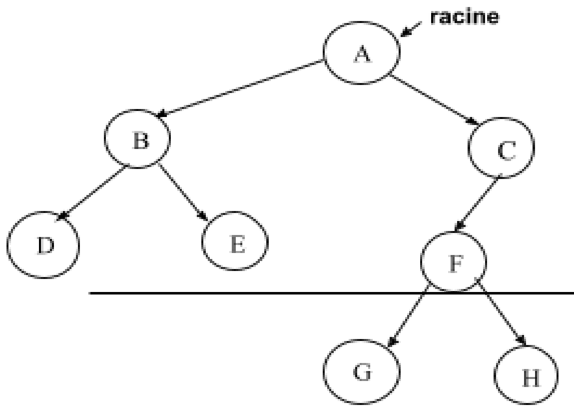
Remarque :

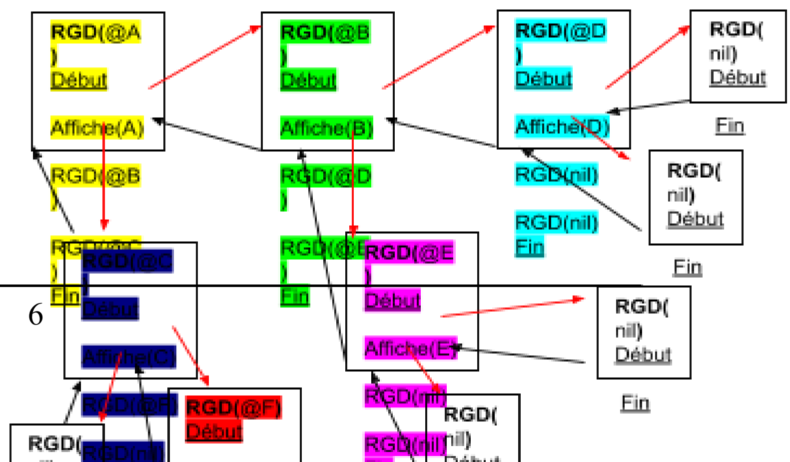
- On considère un arbre dont l'information est de type caractère.
- On suppose que le traitement à effectuer consiste simplement à afficher le contenu de chaque nœud.

La procédure de parcours **Préfixé** est **récursive** et s'écrit comme suit :

Procédure prefixe(E/ a : Arbre)// RGD
Début
Si (non EstVide(a))
  alors écrire (a^.inf) ;
    prefixe(Fils_Gauche(a)) ;
    prefixe(Fils_Droit(a)) ;
Fsi ;
Fin
	void prefixe (arbre a)
	{ if (a !=NULL) { afficher(a inf) ;
	    prefixe (a gauche) ;
	    prefixe (a droit);
	  }
	}

Exemple : En considérant l'exemple ci-dessous, effectuer un déroulement de la procédure Préfixe (RGD) en donnant les détails de chaque appel. @A, @B,…,@H représentent les adresses des nœuds.





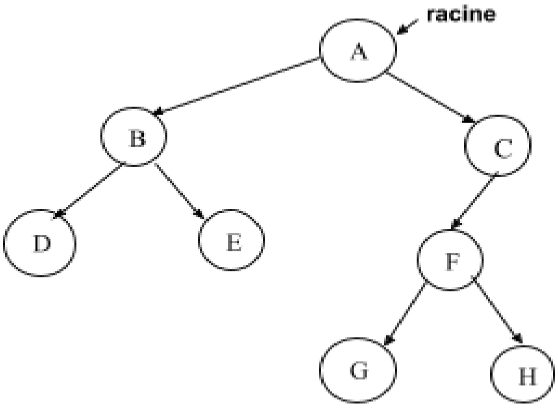
PP
RGD(@A) ;
Affichage : **A**
B
D
E
C
F
G
H

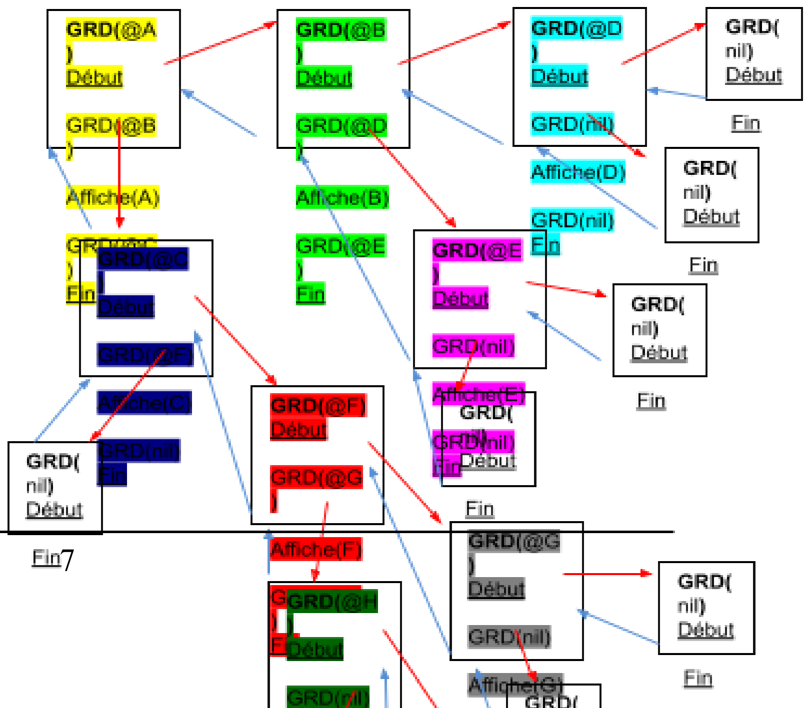
b- **Parcours infixé d'un arbre binaire**

Dans ce type de parcours, on effectue d'abord un parcours infixé de tous les nœuds de A1 (**fils gauche**) jusqu'aux feuilles, on traite la **racine**, on effectue ensuite un parcours infixé de tous les nœuds de A2 (**fils droit**).

La procédure de parcours **Infixé** est **récursive** et s'écrit comme suit :

Procédure infixe(E/ a : Arbre) // GRD
Début
si(non EstVide(a))
    alors
        infixe(Fils_Gauche(a)) ;
        écrire (a^.inf) ;
        infixe(Fils_Droit(a)) ;
Fsi ;
Fin
	void infixe (arbre a)
	{ if (a !=NULL) {
	        infixe (a fg) ;
	        printf("%c\t",a->inf);
	        infixe (a inf) ;
	    }
	}





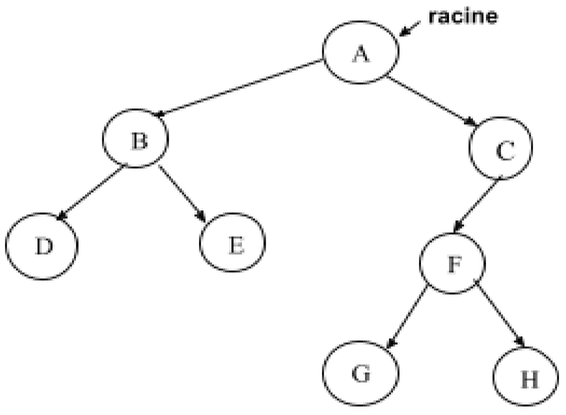
Affichage : **D**
B
E
A
G
F
H
C

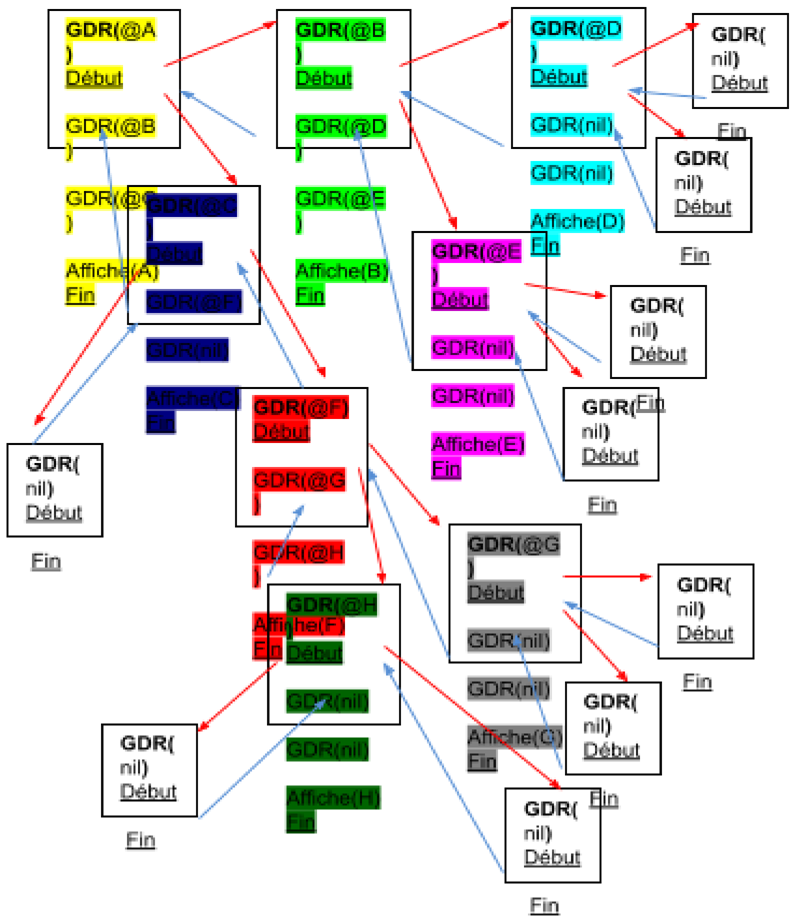
C. **Parcours postfixé d'un arbre binaire**

Dans ce type de parcours, on effectue d'abord un parcours postfixé de tous les nœuds de A1 (**fils gauche**) jusqu'aux feuilles, on effectue ensuite un parcours postfixé de tous les nœuds de A2 (**fils droit**). A la fin, on passe par la **racine**.

La procédure de parcours **Postfixé** est **récursive** et s'écrit comme suit :

Procédure postfixe(E/ a : Arbre) // GDR
Début
si(non EstVide(a))
alors
postfixe(Fils_Gauche(a)) ;
postfixe(Fils_Droit(a)) ;
écrire (a^.inf) ;
Fsi ;
Fin
	void postfixe (arbre a)
	{ if (a !=NULL) {
	prefixe (a fg) ;
	prefixe (a fdt);
	afficher(a inf) ;
	}
	}





Affichage : D
E
B
G
H
F
C
A

Parcours en largeur

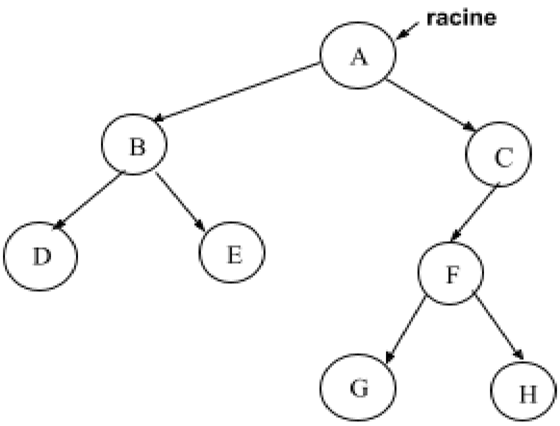
Le parcours en largeur consiste à parcourir tous les nœuds d'un niveau i (de gauche à droite) avant de parcourir les nœuds de niveau i+1. Le parcours commence par la racine (le nœud de niveau 0), ensuite ses descendants directs niveau 1, ensuite les descendants de niveau 2, jusqu'à arriver aux nœuds feuilles.

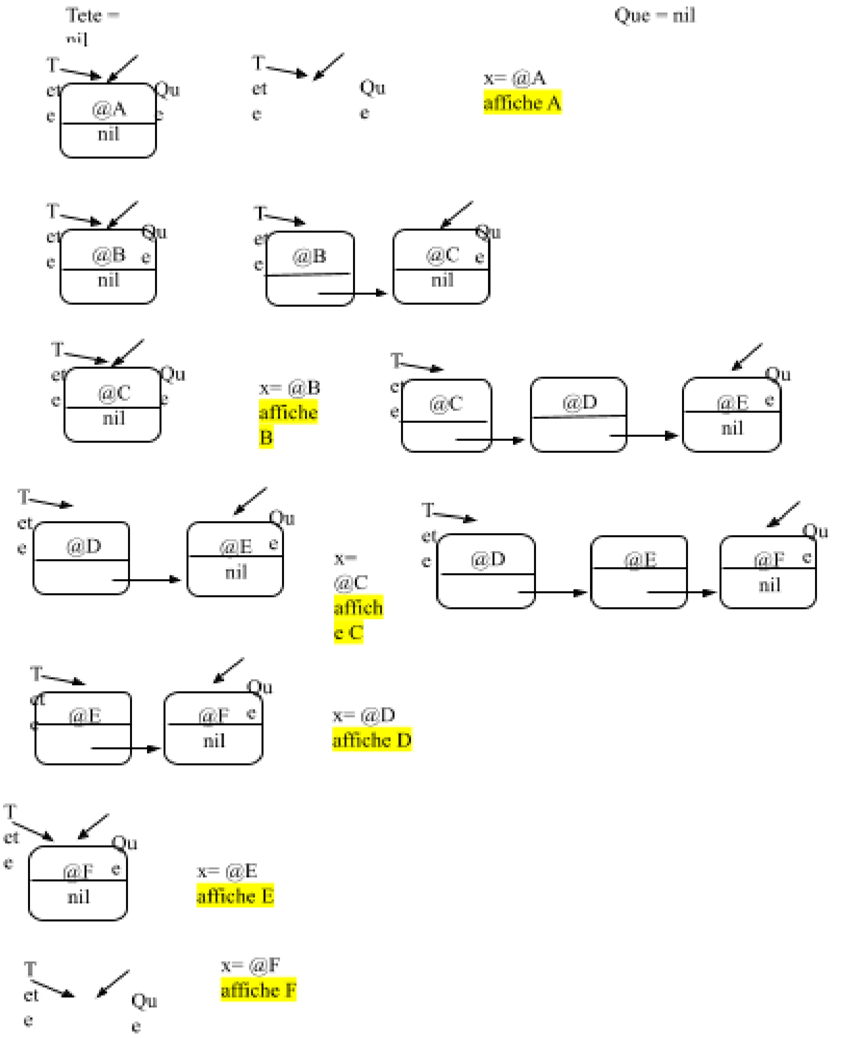
Dans ce type de parcours on aura besoin d'une file.

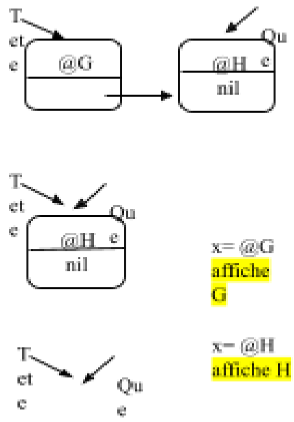
```
Procédure Parcour_en_largeur(E/a : arbre)
Début
  Tete, Que : File ;
  Tete    nil ;  Que    nil ;

  Enfiler (Tete, Que, a) ;
  Tantque (non fileVide(Tete))
    Faire
      Défiler (tete, Que,  x) ;
      Ecrire(x^.info) ;
      Si (non EstVide (Fils_Gauche(x))) alors Enfiler (Tete, Que, Fils_Gauche(x)) ; Fsi
      Si (non EstVide (Fils_Droit(x)))      alors Enfiler (Tete, Que, Fils_Droit(x)) ;    Fsi
    Fait
Fin
```

```
Type file : ^ objet ;
Type objet : Enregistrement
      inf : Arbre ;
      svt : file ;
  Fing ;
```







Algorithmes itératifs de parcours d'un arbre binaire en profondeur

a. Algorithme itératif de parcours préfixé

- On traite la racine
- Si le nœud possède un fils droit, on l'empile
- Si le nœud possède un fils gauche, on le traite sinon on dépile le fils droit et on le traite.

Procédure Préfixe_iter (E/a : **Arbre**) //RGD
S : pile;
Début
  S InitPile() ;
  Empiler (S, nil) ;
  Tant que (non ArbreVide (a))
    Faire
      Ecrire (^a.info) ;
      Si (non ArbreVide(Fils_droit(a)) **alors** Empiler (S, Fils_droit(a)) ; **fsi** ;
      Si (non ArbreVide(Fils_gauche(a)) **alors** a Fils_gauche(a) ;
                          sinon Désempiler (S, a) ;
      fsi ;
    Fait ;
Fin ;

b. Algorithme itératif de parcours infixé

(1) Empiler le chemin (branche) le plus à gauche du nœud **a.**
(2) Si la pile n'est pas vide
  - Désempiler un nœud et le traiter
  - Si le nœud possède un fils droit, empiler le chemin le plus à gauche de ce nœud et revenir à l'étape (2).

```
Procédure infixe_iter (E/a : Arbre) //GRD
S : pile;
Début
   S   InitPile() ;

   Tant que (non ArbreVide (a))
     Faire
       Empiler(s, a) ;
       a   Fils_gauche(a) ;
     Fait ;

   Tant que (non PileVide(S))
     Faire
       Désempiler (s, a) ;
       Ecrire (^a.info) ;
       Si (non ArbreVide(Fils_droit(a))) alors a   FilsDroit(a) ;
                                               Tant que (non ArbreVide (a))
                                                 Faire
                                                   Empiler(s, a) ;
                                                  a   Fils_gauche(a) ;
                                                Fait ;
       Fsi ;
     Fait ;
 Fin ;
```

c. Algorithme itératif de parcours postfixé

- Empiler le chemin le plus à gauche du nœud **a**
- Empiler le nœud **a**
- Si **a** possède un fils droit, empiler une adresse négative correspondant à (**-filsdroit(a)**)
- Tant que la pile n'est pas vide

  Désempiler un nœud

  Si adresse >0 alors traiter le nœud

  Sinon empiler le chemin le plus à gauche du nœud et s'il possède un fils droit empiler (**-filsdroit(a)**).

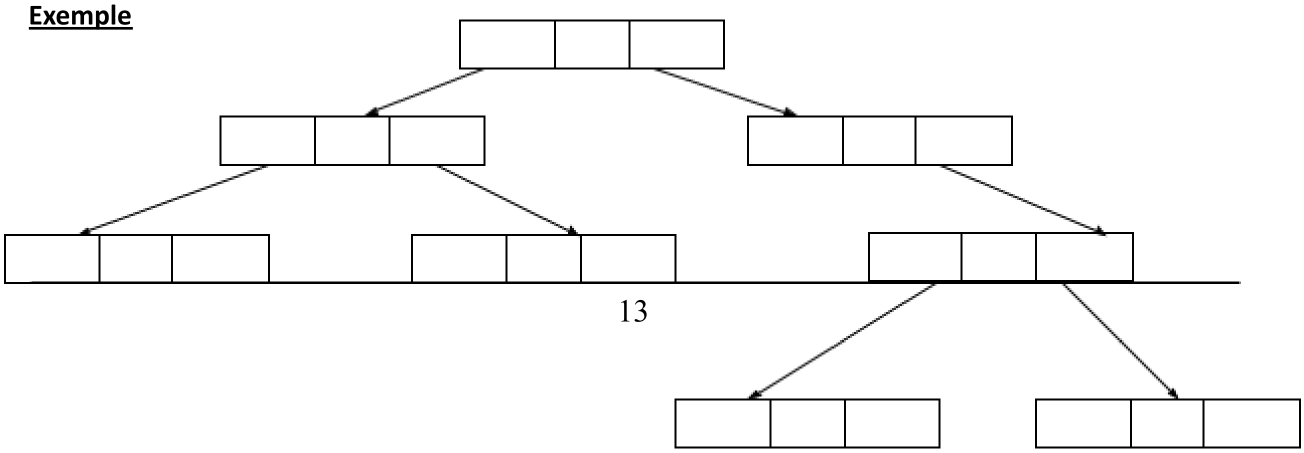
```
Procédure Postfixe_iter (E/a : Arbre)  //GDR
   S : pile ;
Début
  S   Initpile( ) ;
   Tant que (non ArbreVide (a))
      Faire
        Empiler (S, a) ;
        Si (non ArbreVide(Fils_droit(a))) alors Empiler (S, -fils_droit(a)) ; fsi ;
        a     Fils_gauche(a) ; /*Empilement du chemin le plus à gauche */
     Fait ;
  Tant que (non Pilevide(S))
   Faire
     Désempiler (S, a) ;
    Si (a>0)     /* il s'agit d'un fils gauche*/
         Alors Ecrire (^a.info) ;
         Sinon
          Si (Feuille(-a))
             alors Ecrire (-a^.info) ;
             sinon /*Empilement du chemin le plus à gauche */
                a     -a ;
                Tant que (non ArbreVide (a))
                Faire
                Empiler (S, a) ;
                Si (non ArbreVide(Fils_droit(a))alors Empiler(S, -fils_droit(a)) ; fsi ;
                  a     Fils_gauche(a) ;
                 Fait ;
          Fsi ;
      Fsi ;
     Fait ;
Fin ;
```

Arbre binaire de recherche (ou arbre binaire ordonné)

C'est un arbre binaire dont chaque nœud N possède la propriété suivante :

La valeur contenue dans N est supérieure à toute valeur contenue dans le sous-arbre gauche de N, et elle est inférieure à toute valeur contenue dans le sous-arbre droit de N.

Exemple



Question : soit la suite de valeurs suivantes {21, 9, 4, 25, 7, 12, 3, 10, 19, 29, 17, 6, 26, 18} données dans cet ordre. Construire l'arbre binaire ordonné qui les stocke puis afficher le résultat de son parcours **infixé**. Que constatez-vous ?

Recherche dans un arbre binaire ordonné

L'algorithme opère de manière dichotomique comme suit :

- Si la valeur recherchée est contenue dans le noeud courant alors arrêter la recherche et retourner l'adresse du nœud.
- Sinon, si la valeur recherchée est plus petite que celle contenue dans le nœud courant alors orienter la recherche vers le sous-arbre gauche, sinon orienter la recherche vers le sous-arbre droit.

On considère un arbre binaire ordonné d'entiers. Ecrire la fonction qui recherche une valeur val dans l'arbre et retourne son adresse.

Type arbre : ^ objet ;
Type objet : Enregistrement
    inf : <type_elt> ;
    fg, fd : arbre ;
Fing ;

Remarque :

- On utilise le paramètre prd de type arbre pour une éventuelle insertion.
- Les étiquettes dans un arbre binaire ordonné sont toujours uniques, donc l'élément à insérer est toujours une feuille.

Fonction Recherche_rec (E/ a : Arbre, ES/prd : Arbre, E/val : entier) : booleen

Debut

si (ArbreVide(a)) **alors** retouner (faux) ;

sinon **Si** (a^.info = val)

    alors retourner (vrai) ;

    Sinon

    Si (a^.info > val) **alors** retourner (Recherche_rec (Fils_gauche(a), a, val));

        sinon retourner (Recherche_rec (Fils_droit(a), a, val));

    Fsi ;

  Fsi ;

```
   Fsi ;
  Fin.



Fonction  Recherche_ite (E/ a : Arbre, ES/prd : Arbre, E/val : entier) : booleen
 Debut
  trouv : booleen ;
  trouv     faux ;
  Tant que (trouv = faux et a ≠ nil)
    Faire
       Si (a^.info = val)   alors   trouv    vrai ;
                             Sinon
                               prd     a ;
                               Si (a^.info > val) alors  a     Fils_Gauche(a);
                                                  sinon a     Fils_droit(a);
                               Fsi
       Fsi ;
   Fait ;
   Retourner trouv ;
  Fin.
```

Insertion dans un arbre binaire ordonné

L'insertion d'une valeur dans un arbre binaire ordonné doit maintenir l'ordre des éléments dans l'arbre. Si la valeur val existe dans l'arbre on arrête le traitement, sinon on vérifie la valeur val par rapport à la valeur contenue dans le nœud courant. Selon les cas, on s'oriente soit vers le fils gauche de a, soit vers le fils droit de a. L'insertion consiste à allouer un nouvel espace pour la valeur **val** et à mettre à jour les chaînages dans l'arbre, pour cela, il faut utiliser la fonction Recherche_rec/ Recherche_ite qui nous donne l'adresse du nœud **père.**

```
Procédure  Insere (ES/ racine : Arbre, E/val : entier)
 Debut
  prd, tmp : Arbre ;
  prd     nil ;
  si (Recherche_rec (racine, prd , val) = vrai )
             alors écrire('' La valeur existe déjà'') ;
             Sinon
               Allouer(tmp) ;
               tmp ^.inf     val ;
               tmp ^.fg     nil ;
               tmp ^.fg     nil ;
               Si (prd ≠ nil) alors
                        Si (val < prd^.inf) alors  prd^.fg      tmp ;
                                        Sinon prd^.fd      tmp ;
                        Fsi ;
```

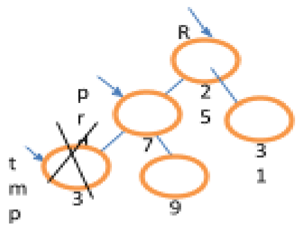
Sinon // l'arbre est vide
racine tmp ;
Fsi ;
Fsi ;
Fin.

Suppression dans un arbre binaire ordonné

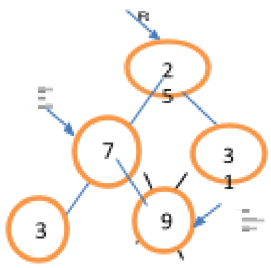
Soit à supprimer un nœud R de l'arbre. Trois cas de suppression sont possibles :

Cas 1 : Le nœud tmp est une feuille

Libérer le nœud tmp et mettre à nil le fils gauche/ fils droit du noeud prd.



prd^.fg nil ;
Libérer (tmp) ;



prd^.fd nil ;
Libérer (tmp) ;

Procédure sup_feuille(E/ prd : Arbre, E/ tmp : Arbre)

Début

Si (tmp^.inf < prd^. inf) **alors** prd^.fg nil ;

sinon prd^.fd nil.

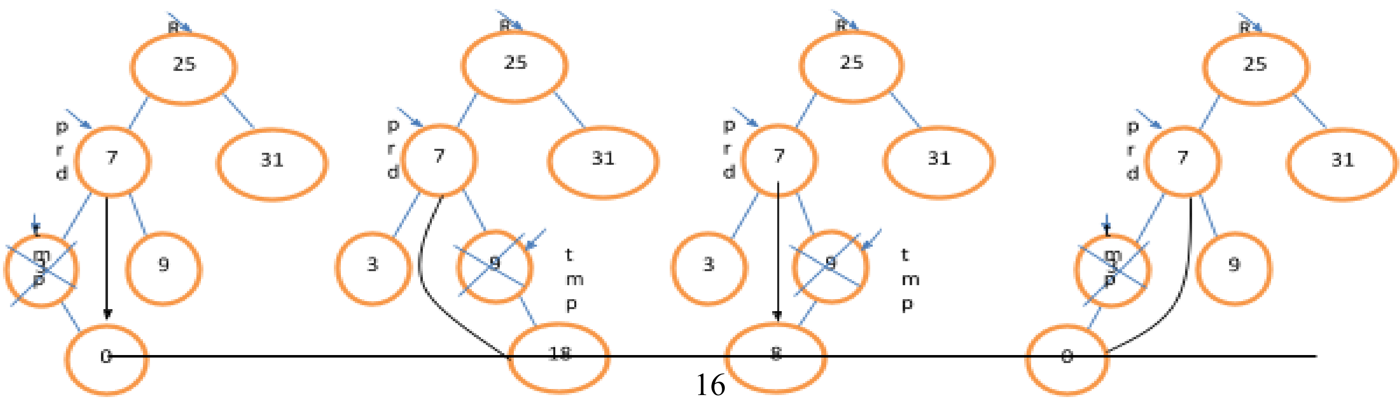
Fsi ;

Libérer (tmp) ;

Fin

Cas 2 : Le nœud tmp possède un seul fils

Libérer le nœud tmp et mettre à jour le fils gauche/ fils droit du noeud prd.

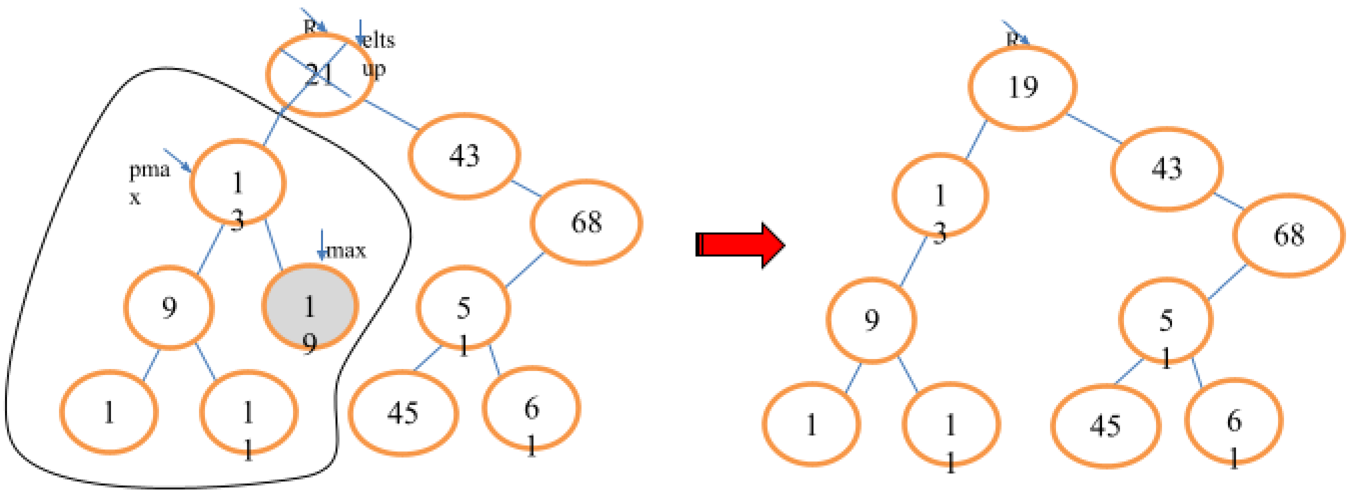


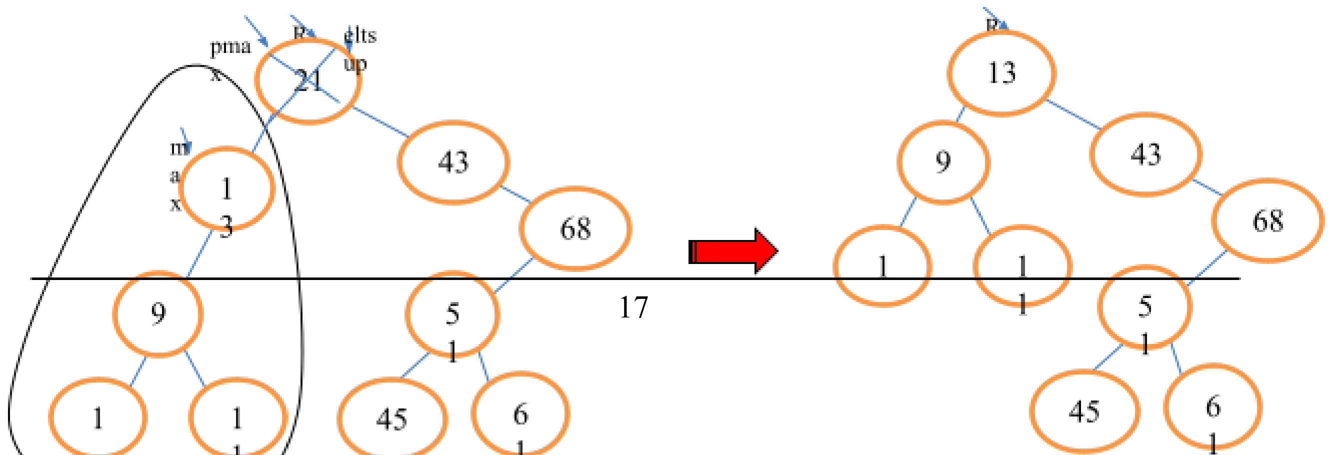
cas n°1 cas n° 2 cas n° 3 cas n°4

Procédure sup_1Fils(E/ prd : Arbre, ES/ tmp : Arbre)
Début
  Si (tmp^.fg = nil) **alors**
    Si (tmp^.inf < prd^.inf) **alors** prd^.fg   tmp^.fd ; // cas n°1
      sinon prd^.fd   tmp^.fd ; // cas n°2
    Fsi ;
  Sinon
    Si (tmp^.inf > prd^.inf) **alors** prd^.fd   tmp^.fg ; // cas n°3
      sinon prd^.fg   tmp^.fg ; // cas n° 4
    Fsi ;
  Fsi ;
Libérer (tmp) ;
Fin

Cas 3 : Le nœud tmp possède deux fils

Remplacer le nœud tmp par la valeur la plus grande du SAG (max) ou bien par la valeur la plus petite du SAD (min) puis supprimer max (ou min).

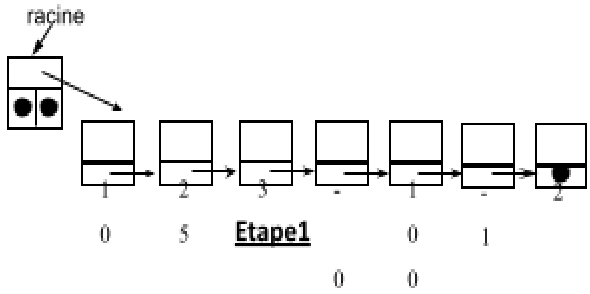


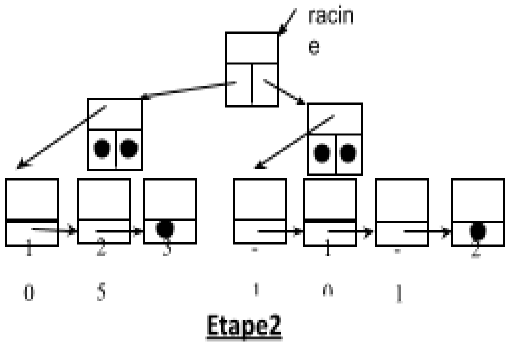


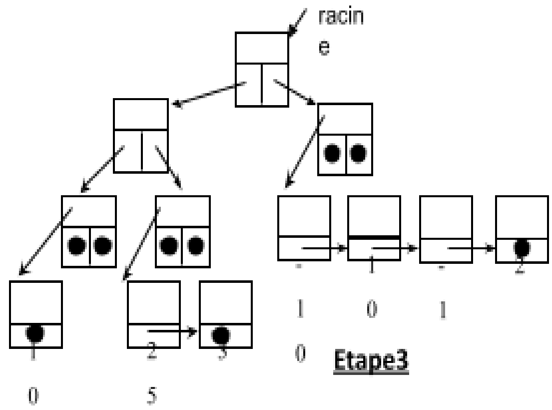
Procédure Remplace(ES/ max : Arbre, E/ eltsup : Arbre , E/ pmax : Arbre)
Début
Si (max^.fd ≠ nil) **alors**
pmax max ;
Remplace(max^.fd, eltsup, pmax) ;
Sinon
eltsup^.inf max^.inf ;
Si (pmax ≠ nil) **alors** pmax^.fd max^.fg ;
sinon eltsup^.fg max^.fg ;
Fsi ;
Liberer (max) ;
Fsi ;
Fin

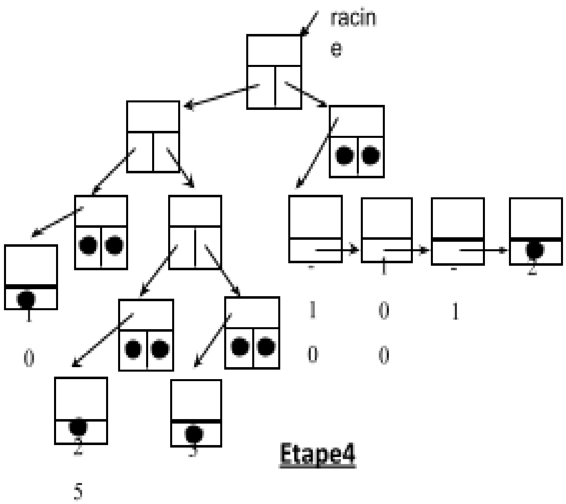
Exercice 1:
Le but de cette première partie est de créer un arbre binaire où le champ valeur de chaque élément contient l'adresse du premier élément d'une liste simplement chaînée (ou l'adresse particulière nil). Initialement, le champ valeur de la racine contient l'adresse du premier élément de la liste principale.
A chaque fois que la liste courante peut être fractionnée (divisée) en deux, on crée deux nœuds fils pour le nœud parent courant. On initialise l'étiquette du fils gauche par l'adresse (tête) de la première liste et l'étiquette du second fils par l'adresse de la deuxième liste. On retire par la suite l'étiquette du nœud parent. Ce principe est utilisé **récursivement** pour l'ensemble de nœud de l'arbre (voir les figures de la page 2).

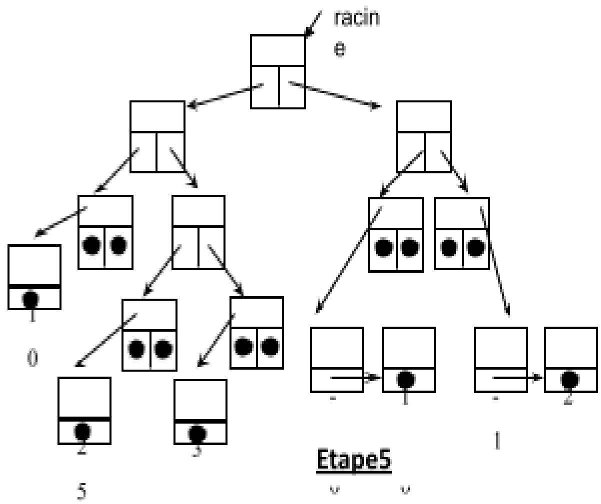
1. Ecrire une action paramétrée qui construit une liste T de N entier. Préciser son type.
2. Initialiser la racine de l'arbre binaire par T et préciser son type.
3. Ecrire l'action paramétrée FRACTIONNE_LISTE(??) qui fractionne (divise) une liste simplement chaînée en deux listes (le plus équitablement possible).
4. Ecrire l'action paramétrée **récursive** CREATION_ARBRE( ) en suivant le principe donné ci-dessus.
5. En utilisant le parcours postfixé (Gauche, Droite, Racine) pour parcourir un arbre, écrire une action paramétrée **récursive** AFFICHE_GDR( ) qui affiche les éléments de la liste principale.

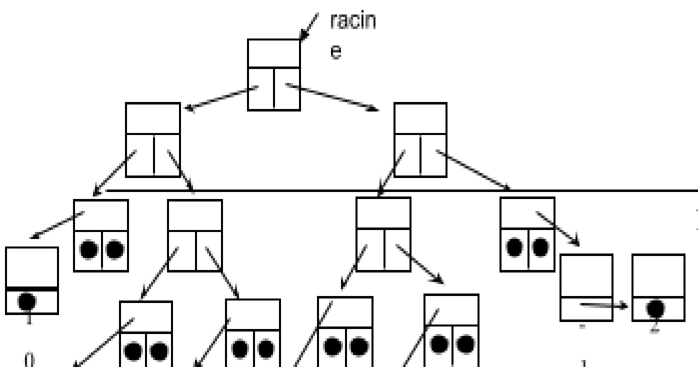

racine
Etape1

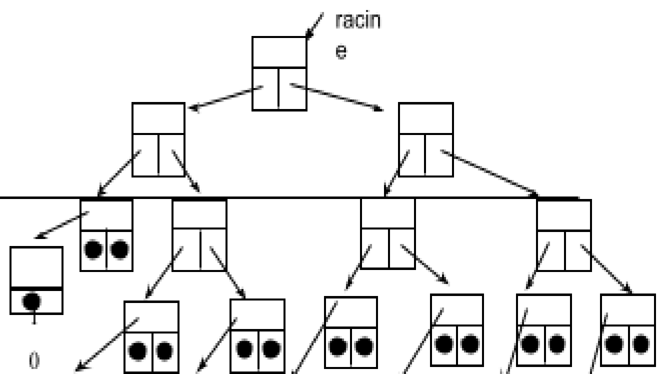

racine
Etape2


racine
Etape3


racine
Etape4


racine
Etape5


racine


racine

<u>Solution</u> :

(1) Essayer de fractionner la liste se trouvant dans le champ valeur du nœud A_i
(2) si on a pu fractionner (on a 2 listes)
<u>Alors</u>
(3) créer un fils gauche pour le nœud A_i et maj ses 3 champs
(4) créer un fils droit pour le nœud A_i et maj ses 3 champ
(5) maj par nil le champ valeur du nœud père : A_i
(6) faites le même traitement (étape : 1, 2, 3, 4, 5) avec le fils gauche de A_i : A_i fg(A_i)
(7) faites le même traitement (étape : 1, 2, 3, 4, 5) avec le fils droit de A_i : A_i fd(A_i)

1/

```
Type  liste : ^ cel ;
 Type cel = enregistrement
              val : entier ;
              svt : liste ;
 fin ;
```

```
fonction créer_liste(E/ N : entier) : liste
Début
  T, p : liste
  i : entier ;
  T    nil ;
Pour (i    1 à N)
   Faire
     Allouer(p) ;
     Lire(p^.val) ;
     P^.svt    T ;
     T    p ;
  Fait
 Retourner (T) ;
Fin ;
```

2/

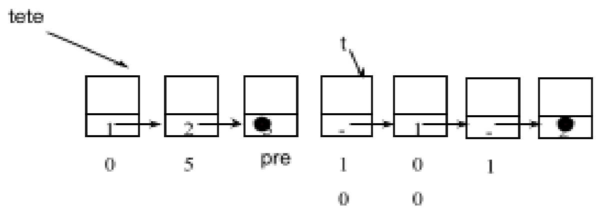
```
Type  arbre : ^ cel1 ;
Type cel1 = enregistrement
              val : liste ;
              fg : arbre ;
              fd : arbre ;
 fin ;
```

T : <u>liste</u> T créer_liste(10) ;	Fonction init_noeud(E/ L : <u>liste</u>) : <u>arbre</u> Début

racine : <u>arbre</u> ; Allouer(racine) ; Racine^.val T ; Racine^.fg nil ; Racine^.fd nil ;	a : <u>arbre</u> ; Allouer(a) ; a^.val L ; a^.fg nil ; a^.fd nil ; retourner (a) ; Fin ;

3/
Fonction fractionne_liste(E/ tete : <u>liste</u>) : <u>liste</u>
<u>Début</u>
 t, pre : liste ;
 nb, j : entier ;

 t <- tete ;
 nb 0 ;
 <u>TQ</u>(t ≠ nil)
  <u>Faire</u>
   nb nb + 1 ;
   t t^.svt ;
  <u>fait</u> ;



 <u>si</u>(nb > 1) **<u>alors</u>**
   t tete ;
   <u>Pour</u> (j 1 à nb/2) // division entière
    <u>Faire</u>
     pre t ;
     t t^.svt ;
    <u>Fait ;</u>
   pre ^.svt nil ;
<u>Fsi</u> ;
Retourner (t) ;
<u>Fin</u> ;

4/
<u>Procédure</u> creation_arbre(E/ r : <u>arbre</u>)
<u>Début</u>
 tn : <u>liste</u> ;

 <u>si</u> (r ≠ nil) **<u>alors</u>**
   tn FRACTIONNE_LISTE(r^.val) ;
   <u>si</u>(tn ≠ nil) **<u>alors</u>**
     r^.fg init_noeud(r^.val) ;
     r^.fd init_noeud(tn) ;
     r^.val nil ;
     creation_arbre(r^.fg) ;
     creation_arbre(r^.fd) ;
   <u>Fsi</u> ;
 <u>Fsi</u> ;
<u>Fin</u> ;

- Essayer de fractionner la liste se trouvant dans le champ valeur du nœud A_i
- si on a pu fractionner (on a 2 listes)
  <u>Alors</u>
  - créer un fils gauche pour le nœud A_i et maj ses 3 champs
  - créer un fils droit pour le nœud A_i et maj ses 3 champ
  - maj par nil le champ valeur du nœud père : A_i
  - faites le même traitement (étape : 1, 2, 3, 4, 5) avec le fils gauche de A_i : A_i fg(A_i)
  - faites le même traitement (étape : 1, 2, 3, 4, 5) avec le fils droit de A_i : A_i fd(A_i)

<u>Procédure</u> affiche_GDR(E/ r : <u>arbre</u>)

```
Début
  p : liste ;

  si (r ≠ nil ) alors
              affiche_GDR(r^.fg);
              affiche_GDR(r^.fd);
              si (r^.val ≠ nil ) alors ecrire((r^.val)^.val) ; fsi ;// est ce que c une feuille
  Fsi ;
Fin ;
```

Partie II

Le but de cette deuxième partie est de trier une liste simplement chaînée en utilisant l'arbre binaire de la question 4. Le principe est donné par les figures de la page 3. Pour chaque couple de nœuds feuilles de même nœud parent, on fusionne leurs étiquettes par ordre croissant, on met à jour l'étiquette de leur parent et on détruit les deux nœuds feuilles. Ce même principe est utilisé **récursivement** pour l'ensemble de nœud de l'arbre.

6. Ecrire l'action paramétrée FUSIONNE_LISTES( ) qui fusionne deux listes **triées** sans utiliser les actions allouer/libérer.
7. En utilisant l'action paramétrée FUSIONNE_LISTES( ) et l'arbre binaire généré dans la question 4, écrire l'action paramétrée **récursive** TRIER_LISTE ( ) qui tri la liste en suivant le principe donné ci-dessus.

Solution

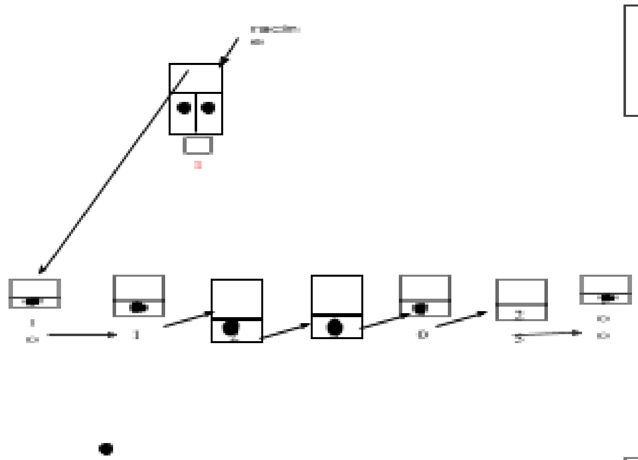
6/

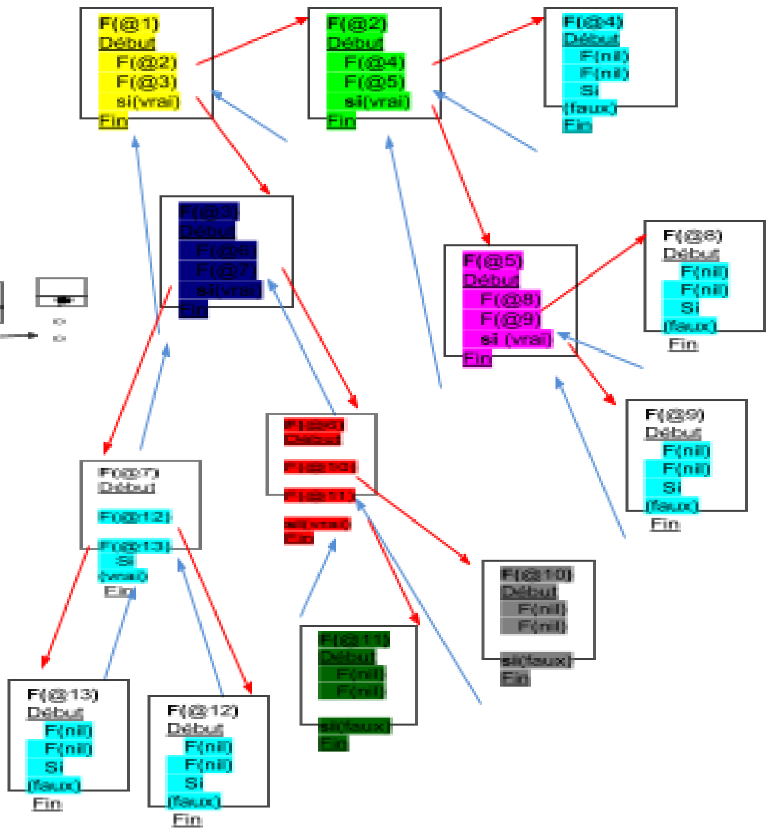
```
Fonction fusionne_listes (ES/ l1, l2 : liste) : liste
Début
  l, r : liste ;
  Si (l1 ≠ nil et l2 ≠ nil)
     Alors
       si (l1^.val < l2^.val) alors  l    l1 ;    l1    l1^.svt ;
                              Sinon  l    l2 ;    l2    l2^.svt ;
       Fsi ;
      r    l ;
       TQ (l1 ≠ nil et l2 ≠ nil)
         Faire
           si (l1^.val < l2^.val) alors  r^.svt    l1 ;   r    l1 ;    l1    l1^.svt ;
                                  sinon  r^.svt    l2 ;   r    l2;     l2    l2^.svt ;
           Fsi;
         Fait ;
         Si (l1 = nil) alors r^.svt    l2 ;  l2    nil ;  fsi ;
                      Sinon r^.svt    l1 ;  l1    nil ;  fsi ;
  Fsi;
  Retourner (l) ;
Fin ;
```

7/

```
Procédure Trier_liste(E/ r : arbre)
Début
```

si (r ≠ nil) **alors**
Trier_liste (r^.fg);
Trier_liste (r^.fd);
si (r^.fg ≠ nil **et** r^.fd ≠ nil) // nœud interne ≠ feuille
alors
r^.val fusionne_listes ((r^.fg)^.val, (r^.fd)^.val) ;
Libérer(r^.fg) ; r^.fg nil ;
Libérer(r^.fd) ; r^.fd nil ;
Fsi ;
Fsi ;
Fin ;





Procédure Trier_liste(E/ r : arbre)
Début
si (r ≠ nil) **alors**
Trier_liste (r^.fg);
Trier_liste (r^.fd);
si (r^.fg ≠ nil **et** r^.fd ≠ nil)
alors
r^.val fusionne_listes (r^.fg^.val, r^.fd^.val) ;
Libérer(r^.fg) ; r^.fg nil ;
Libérer(r^.fd) ; r^.fd nil ;
Fsi ;
Fsi ;
Fin ;

Fonction fusionne_listes (ES/ l1, l2 : liste) : liste
Début
l, r : liste ;
Si (l1 ≠ nil et l2 ≠ nil)
Alors
si (l1^.val < l2^.val) **alors** l l1 ; l1 l1^.svt ;
Sinon l l2 ; l2 l2^.svt ;
Fsi ;
r l ;
TQ (l1 ≠ nil **et** l2 ≠ nil)
Faire
si (l1^.val < l2^.val) **alors** r^.svt l1 ; r l1 ; l1 l1^.svt ;
sinon r^.svt l2 ; r l2; l2 l2^.svt ;
Fsi;
Fait ;
Si (l1 = nil) **alors** r^.svt l2 ; l2 nil ; **fsi** ;
Sinon r^.svt l1 ; l1 nil ; **fsi** ;
Fsi;
Retourner (l) ;
Fin ;