

Chapitre 5

Structures de Données arborescentes :

Les Arbres

PART I I



Primitives des arbres binaires

```
/*****************************************************************
 *  Fichier : ARBPRIM.H
 *  Contenu : Déclaration des primitives du TDA ARBIN
*****************************************************************/
#ifndef _ARBPRIM_H
#define _ARBPRIM_H
#include <stdio.h>
#include <stdlib.h>
#include "ARBSDD.H"

/*permet de créer un arbre binaire. Elle renvoie NULLen cas d'insuffisance de mémoire*/
ARBIN arbinCreer(void);

/*permet de détruire un arbre binaire et libérer l'espace mémoire alloué */
void arbinDetruire(ARBIN);

int arbinVide(ARBIN);

int arbinTaille(ARBIN);

int arbinHauteur(ARBIN);

int construire(ARBIN, ELEMENT, ARBIN, ARBIN);

/*permet de renvoyer le sous arbre binaire gauche */
ARBIN SAGauche(ARBIN);

/*permet de renvoyer le sous arbre binaire droit*/
ARBIN SADroit(ARBIN);

/* permet de renvoyer le contenu de racine d'un arbre binaire*/
ELEMENT racine(ARBIN);

/* permet d'affiche le contenu de l'arbre en profondeur: 1=préfixé, 2=infixé, 3=postfixé*/
void arbinAfficher(ARBIN, int);

ARBIN arbinCopier(ARBIN);

int arbinComparer(ARBIN, ARBIN);

#endif
```

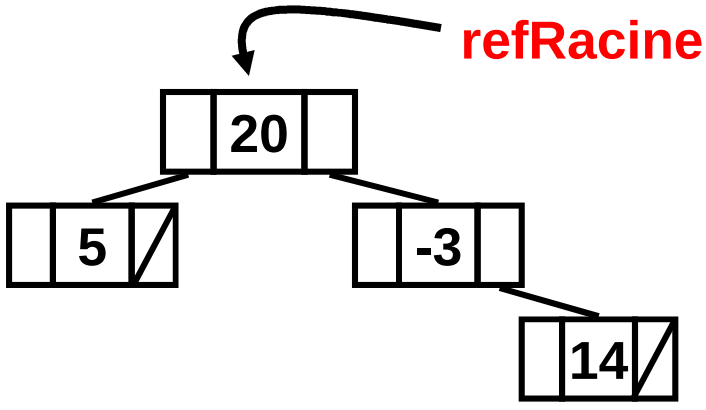
Représentation chaînée des AB

Dans l'implémentation chainée chaque nœud est subdivisé en 3 champs :

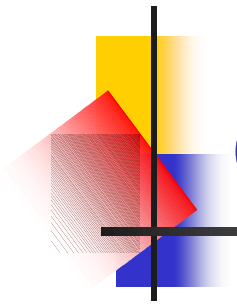
- L'information qu'il contient: **info**
- L'adresse de son fils gauche: **fg**
- L'adresse de son fils droit: **fd**

fg	info	fd

L'arbre n'est accessible que par l' **adresse** de sa racine (**refRacine**)



Représentation chaînée des AB



FG	Info	FD

```
/*************************************************************
 *  Fichier : ARBIN.H
 *  Contenu : Déclaration de la strucure de donnée choisie du TDA ARBIN.
*************************************************************/
#ifndef _ARBIN_H
#define _ARBIN_H
#include "ELTPRIM.H"

/* déclaration de structure structNoeud */
typedef struct structNoeud {
  ELEMENT info;                 /* information stockée dans le noeud */
  struct structNoeud * fg;      /* adresse de fils gauche */
  struct structNoeud * fd;      /* adresse de fils droit */
}  structNoeud, * NOEUD;

/* déclaration de structure laStruct */
typedef struct {
  NOEUD refRacine;  /* racine de l'arbre bianire */
} laStruct, * ARBIN;
#endif
```

Créer

```
/*************************************************************
 *  Fichier : ARBIN.C
 *  Contenu : Définition des primitives pour une réalisation chaînée du
TDA ARBIN.
*************************************************************/

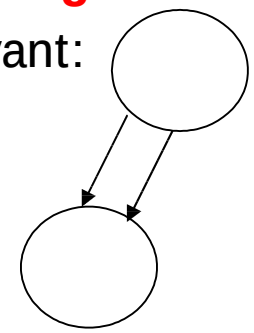
#include "ARBPRIM.H"  /* les primitives d'un arbre bianire     */

/*  fonction arbinCreer() permet de créer un arbre binaire vide. Elle
renvoie NULL en cas d'insuffisance de mémoire*/

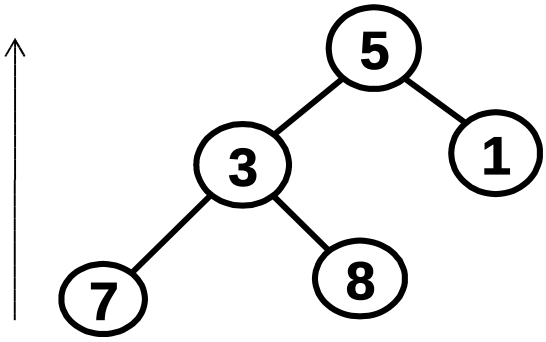
ARBIN arbinCreer() {
   ARBIN r;
   r = (ARBIN) malloc(sizeof(laStruct));
   if(!r) {                  /* pas d'espace mémoire */
      printf("Erreur de création d'un arbre binaire: pas assez de
            mémoire.\n");
   } else {  /* initialisation de racine d'arbre */
      r->refRacine = NULL;
   }
   return r;
}
```

Construire (1)

- **construire(a, e, g, d)** permet de construire un nouveau arbre binaire **a** dont l'information dans la racine est **e**, le sous arbre droit est **d** et le sous arbre gauche est **g**
- Avant la construction il faut vérifier que **d** et **g** ne soient pas identiques afin d'éviter le schéma suivant:



- Le seul cas où ils peuvent être identiques c'est le cas où ils sont NULL
- La construction d'un arbre se fait des feuilles vers la racine
- **Exemple** : si on veut construire l'arbre suivant on peut le faire dans la séquence : 7 – 8 – 3 – 1 – 5 cad
  - on construit 7 et 8
  - ensuite un arbre avec comme racine 3, SAD 8 et SAG 7
  - Puis on construit 1 puis un arbre avec comme racine 5, SAD 1 et SAG l'arbre dont la racine est 3



Construire (2)

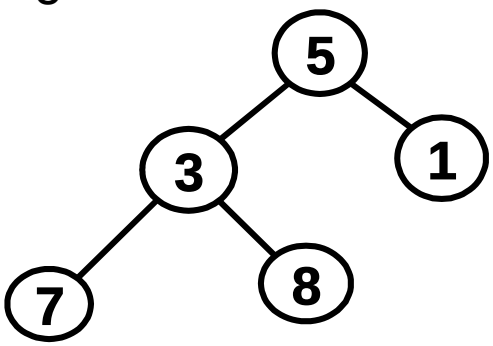
```
int construire(ARBIN a, ELEMENT e, ARBIN g, ARBIN d) {
   NOEUD n;
   if(g == d) && (g!=NULL) {
      printf("Erreur de construction: SAG et SAD identiques.\n");
      return 0;
   }
    n = noeudCreer(e, g->refRacine, d->refRacine);
   if(n) {   a->refRacine = n;
             return 1;
             }
   else
      return 0;
 }
```

```
/*  fonction noeudCreer() permet de créer un noeud */

NOEUD noeudCreer(ELEMENT e, NOEUD g, NOEUD d) {
   NOEUD r;
   r = (NOEUD) malloc(sizeof(structNoeud));
   if(!r) {
      printf("mémoire insuffisante.\n");
   } else {
      r->info = e;
      r->fg = g;
      r->fd = d;
            }
   return r;
}
```

Construire (3)

- **Exemple** : si on veut construire l'arbre suivant dans la séquence : 7 – 8 – 3 – 1 - 5



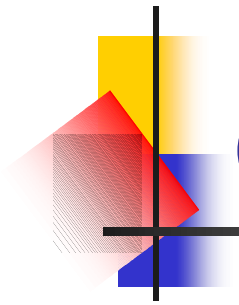

```
/*************************************************************
*   Fichier : main.c
*   Contenu : Test du TDA ARBIN
*************************************************************/

#include <stdio.h>
#include "ELTPRIM.H"
#include "ARBPRIM.H"

int main()
{

   ELEMENT elt;

   ARBIN a1 = arbinCreer();
   ARBIN a2 = arbinCreer();
   ARBIN a3 = arbinCreer();
   ARBIN a4 = arbinCreer();
   ARBIN a5 = arbinCreer();
```



Construire (4)

```
printf ("\nEntrez l'info du premier noeud : \n") ;
elt = elementCreer();
elementLire(&elt) ;
construire(a1, elt, arbinCreer(),arbinCreer());

printf ("\nEntrez l'info du deuxième noeud : \n") ;
elt = elementCreer();
elementLire(&elt) ;
construire(a2, elt, arbinCreer(),arbinCreer());

printf ("\nEntrez l'info du troisième noeud : \n") ;

elt = elementCreer();
elementLire(&elt) ;
construire(a3, elt, a1,a2);

printf ("\nEntrez l'info du quatrième noeud : \n") ;
elt = elementCreer();
elementLire(&elt) ;
construire(a4, elt, arbinCreer(),arbinCreer());

printf ("\nEntrez l'info du cinquième noeud : \n") ;
elt = elementCreer();
elementLire(&elt) ;
construire(a5, elt, a3,a4);
}
```

On introduit 7

On introduit 8

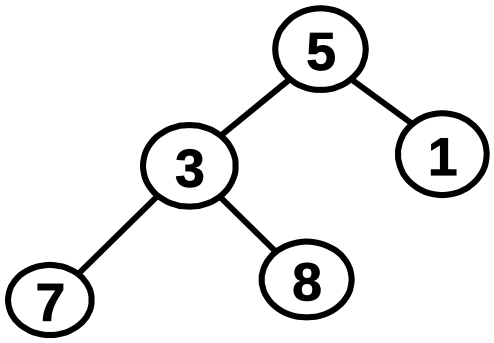
On introduit 3

On introduit 1

On introduit 5

Détruire (1)

- la fonction **arbinDetruire** appelle la fonction recursive **noeudDetruire** qui va détruire les noeuds en commencant par les feuilles vers la racine
- Dans l'exemple suivant la destruction se fera dans l'ordre suivant: 7 – 8 – 3 – 1 – 5
- Autrement dit on détruit les noeuds du SAG ensuite ceux du SAD



Détruire (2)

```
/* fonction arbinDetruire() permet de détruire un arbre binaire et libérer
l'espace mémoire alloué */

void arbinDetruire(ARBIN a) {
      noeudDetruire(a->refRacine);
}
```

```
/* fonction noeudDetruire() permet de détruire un noeud */


void noeudDetruire(NOEUD n){
      if(n) {
      if(!n->fg && !n->fd) {    /* le noeud courant est une feuille */
         elementDetruire(n->info);
         free(n);
      } else {
               noeudDetruire(n->fg); /* déstruction à partir du FG */
               noeudDetruire(n->fd); /* déstruction à partir du FD */
               n->fg=NULL;
               n->fd=NULL;
               noeudDetruire(n);
      }
     }    }
```

Vide – Racine

```
int arbinVide(ARBIN a) {



}
```

```
/* fonction racine() permet de renvoyer le contenu de racine
d'un arbre binaire a*/

 ELEMENT racine(ARBIN a) {

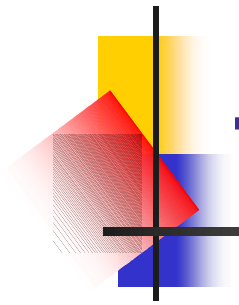








  }
```



Taille

```
/*  arbinTaille() permet de renvoyer la taille d'un arbre binaire a*/

int arbinTaille(ARBIN a) {

}

----------------------------------------------------------------------------

/* noeudTaille est une fonction récursive pour déterminer la taille de
l'arbre dont la référence est n*/

int noeudTaille(NOEUD n) {



}
```

- **Remarque**: la fonction arbinTaille a un arbre en paramètre et elle appelle la fonction récursive noeudTaille qui a un nœud en paramètre
- Ce choix est justifié par le fait que si la fonction arbinTaille était programmée d'une façon récursive alors on aura

  `return 1 + arbinTaille(SADroit(a)) + arbinTaille(SAGauche(a))` qui va créer des copies de tous les SAG et SAD dans a

Hauteur

```
/*  fonction arbinHauteur() permet de renvoyer l'hauteur d'un arbre binaire a*/

int arbinHauteur(ARBIN a) {


}
```

```
/* fonction noeudTaille est une fonction récursive pour déterminer l'hauteur de l'arbre dont la référence est n*/

int noeudHauteur (NOEUD n) {



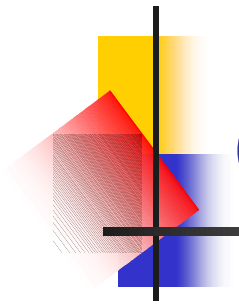






}
```

- **Remarque**: la fonction arbinHauteur a un arbre en paramètre et elle appelle la fonction récursive noeudHauteur qui a un nœud en paramètre (même explication que pour arbinTaille)



Copier

```
ARBIN arbinCopier(ARBIN a) {



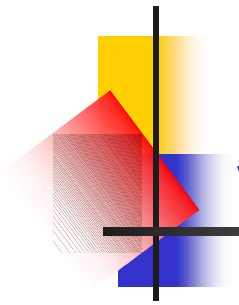

}
```

```
NOEUD noeudCopier(NOEUD n) {




}
```

- **Remarque**: la fonction arbinCopier a un arbre en paramètre et elle appelle la fonction récursive noeudCopier qui a un nœud en paramètre (même explication que pour arbinTaille)



SAGauche - SADroit

```
ARBIN SAGauche(ARBIN a) {

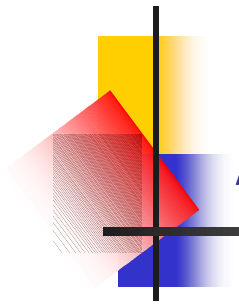



}

ARBIN SADroit(ARBIN a) {




}
```



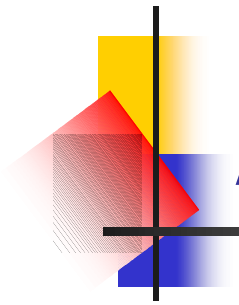
Affichage (1)

```
void arbinAfficher(ARBIN a, int ordre) {




}
```

- Cette procédure appelle la fonction récursive **afficherNoeud qui** permet un parcours en profondeur:
  - ordre=1: parcours préfixé
  - ordre=2: parcours infixé
  - ordre=3: parcours postfixé

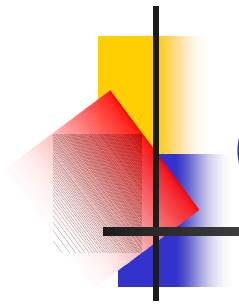


Affichage (2)

```
void afficherNoeud(NOEUD n, int ordre) {




}
```



Comparaison

```
/*fonction récursive arbinComparer() permet de comparer
deux arbres binaires  a1 et a2*/

int arbinComparer(ARBIN a1, ARBIN a2) {




}
```