

UE ALGO5 — TD2 — Séance 7 : Arbres binaires

Objectifs

À la fin de cette séance, vous devriez être capable de :

- manipuler et concevoir des arbres binaires comme des structures abstraites ;
- réfléchir aux propriétés des arbres binaires ;
- proposer des implémentations d'arbres binaires cohérentes avec les spécifications choisies en utilisant des structures sous-jacentes adaptées.

On donne un exemple de spécification d'un type abstrait **Arbre** construit sur un type **Élément**.

```

1  ArbreVide {
    Données : aucun
3   Résultat : un Arbre vide
    Post-condition : renvoie un Arbre vide
5   Effet de bord : aucun
    }
7  NouveauNœud(G,x,D) {
9   Données : un Arbre G, un Élément x, un Arbre D
    Résultat : un Arbre constitué du nœud (G, x, D)
11  Effet de bord : un nouveau nœud a été créé
    }
13 EstArbreVide(A) {
15  Donnée : un Arbre A
    Résultat : un booléen vrai ssi A est un Arbre vide
17  }

19 Racine(A) {
    Donnée : un Arbre A
21  Résultat : l'Élément associé à la racine de A
    Pré-condition : A non vide
23  }

25 FilsDroit(A) {
    Donnée : un Arbre A
27  Résultat : un Arbre, renvoie le fils droit associé à la
    racine de A
    Pré-condition : A non vide
29  }

31 FilsGauche(A) {
    Donnée : un Arbre A
33  Résultat : un Arbre, renvoie le fils gauche associé à
    la racine de A
    Pré-condition : A non vide
35  }

37 Libérer(A) {
    Donnée : un Arbre A
39  Pré-condition : A non vide
    Post-condition : la mémoire associée au nœud racine
    de A est libérée
41  Effet de bord : La racine de A est détruite, les fils
    gauche et droit sont inchangés
    }

43 InsérerGauche(A,G) {
45  Donnée-résultat : un Arbre A
    Donnée : un Arbre G
47  Pré-condition : A non vide
    Post-condition : le fils gauche de A est remplacé par
    l'Arbre G.
49  Effet de bord : l'Arbre A est modifié, son ancien fils
    gauche est «détruit»
    }

51 InsérerDroit(A,D) {
53  Donnée-résultat : un Arbre A
    Donnée : un Arbre D
55  Pré-condition : A non vide
    Post-condition : le fils droit de A est remplacé par
    l'Arbre D.
57  Effet de bord : l'Arbre A est modifié, son ancien fils
    droit est «détruit»
    }

```

Exercice 1. Utilisation du type abstrait **Arbre**

Q1. Dessiner un arbre binaire de hauteur 3 comportant 5 feuilles

En n'utilisant que des primitives du type abstrait :

Q2. Écrivez une séquence d'instructions permettant de construire l'arbre binaire proposé à la question 1.

Corrigé —

Par exemple :

A0, A1, A3, A4 : Arbre

```
2 A0 := NouveauNoeud (ArbreVide, 0, ArbreVide)
4 A1 := NouveauNoeud (ArbreVide, 1, ArbreVide)
  InsérerGauche(A0, A1)
6 InsérerDroit (A0, NouveauNoeud(ArbreVide, 2, ArbreVide))
  A3 := NouveauNoeud (ArbreVide, 3, ArbreVide)
8 InsérerGauche(A1, A3)
  A4 := NouveauNoeud (ArbreVide, 4, ArbreVide)
10 InsérerDroit(A1, A4)
  InsérerGauche (A3, NouveauNoeud(ArbreVide, 5, ArbreVide))
12 InsérerDroit (A3, NouveauNoeud(ArbreVide, 6, ArbreVide))
  InsérerGauche (A4, NouveauNoeud(ArbreVide, 7, ArbreVide))
14 InsérerDroit (A4, NouveauNoeud(ArbreVide, 8, ArbreVide))
```

Q 3. Écrivez une fonction qui prend en paramètre un arbre binaire et renvoie le nombre de feuilles de cet arbre.

Corrigé —

```
NbFeuilles(A : Arbre)
2  si EstVide(A) alors
  renvoyer 0
4  sinon
  si EstVide(FGauche(A)) et EstVide(FDroit(A)) alors
6    renvoyer 1
  sinon
8    renvoyer NbFeuilles(FGauche(A)) + NbFeuilles(FDroit(A))
```

Q 4. Écrivez une fonction qui prend en paramètre un arbre binaire et renvoie la hauteur de cet arbre.

Corrigé —

Rappel de la définition de la «hauteur» vue en cours : longueur en nombre d'arcs du plus long chemin de la racine vers les feuilles.

```
Hauteur(A : Arbre)
2  si EstVide(A) alors
  renvoyer -1 { par convention }
4  sinon
  si EstVide(FGauche(A)) et EstVide(FDroit(A)) alors
6    renvoyer 0
  sinon
8    renvoyer 1 + max(Hauteur(FGauche(A)), Hauteur(FDroit(A)))
```

Q 5. Écrivez une procédure permettant de «supprimer» un arbre entier, en libérant la mémoire de chacun de ses nœuds.

Comment procéderiez-vous sans récursivité ?

Corrigé —

Version récursive

Attention à l'ordre des appels récursifs !

```
Suppression(A : Arbre)
2  si non EstVide(A) alors
    Suppression(FGauche(A))
4  Suppression(FDroit(A))
    Libérer(A)
```

Version itérative

Et en itératif ? Il faut une structure intermédiaire pour stocker les arbres «à traiter» (pile ou file). (NB : dans le cas d'une procédure récursive, cette structure est justement la pile d'appels.)

Or on peut implémenter une pile (ou liste) d'arbres binaires avec un arbre binaire :

—la pile/liste vide est représentée par l'arbre vide

—le fils gauche d'un arbre représente le contenu de la pile/liste, le fils droit la suite de la pile/liste.

```
1 Pile : le type Arbre

3 PileVide: ArbreVide

5 Empiler(A, P): NouveauNoeud(A, 0, P) // l'élément n'est pas utilisé
Dépiler(P) :
7  A := FGauche(P)
  P := FDroit(P)
9  renvoyer A
```

La suppression de l'arbre entier devient :

```
1 Suppression(A):
  P : une pile
3  P := PileVide
  Empiler(A,P)
5  tant que non EstVide(P)
    A1 := Depiler(P)
7    si non EstVide(A1) alors
        Empiler(FGauche(A1), P)
9        Empiler(FDroit(A1), P)
        Libérer(A1)
11  fin si
  fin tant que
```

Exercice 2. Implémentation du type abstrait Arbre

Q1. Proposez une implémentation du type Arbre à l'aide d'un tableau de taille N alloué statiquement (on suppose donc ici que le nombre de nœuds de tout arbre binaire sera limité à N). Écrivez le code de chaque primitive pour cette implémentation.

Corrigé —

On peut explorer différentes solutions :

—«recopier» les nœuds de l'arbre dans le tableau selon un mode de parcours fixé (infixe, postfixe, préfixe).

Problème : l'arbre associé à un tel tableau ne sera pas unique !

En codant également les arbres vides, on obtient une correspondance exacte entre un arbre et son parcours préfixe (ou postfixe), mais le tableau résultant reste peu adapté pour écrire les primitives souhaitées.

—chaînage implicite : utiliser une fonction associant à chaque nœud d'indice i les indices de ses fils gauche et droit.

Par exemple : fils gauche à l'indice $2i$, fils droit à l'indice $2i + 1$. Si l'arbre est binaire complet, tous les nœuds sont contigus (et stockés dans la partie gauche du tableau). Si l'arbre n'est pas complet il faut en plus conserver l'information (booléenne) de l'existence ou non du nœud i (ensemble de nœuds implémenté par exemple par un tableau de booléens, ou une valeur spéciale de l'élément).

—chaînage explicite : tableau de triplets \langle élément, indice fils gauche, indice fils droit \rangle . On «singe» ici le chaînage par pointeurs.

Un «arbre» est l'indice dans le tableau de sa racine.

On peut utiliser une valeur arbitraire (hors de $0..N-1$) pour représenter l'arbre vide (ne comportant aucun nœud), par exemple -1 . Un nœud feuille a alors nécessairement -1 comme valeurs de fils gauche et droit.

Il y a ensuite plusieurs politiques possibles pour choisir un emplacement libre dans le tableau lorsque l'on construit un nouveau nœud. On pourra par exemple faire en sorte que les cases occupées soient toujours comprises entre 1 et $NbNoeud$ (représentation contiguë), et les cases libres entre $NbNoeud+1$ et N . Cela devient cependant assez impraticable lorsqu'on souhaite «libérer» des nœuds.

Une autre solution est de conserver un chaînage explicite des nœuds libres (ci-dessous en utilisant le champ `fd`).

Remarque : cette solution marche encore si on utilise le même tableau pour mémoriser une «forêt» de plusieurs arbres disjoints.

En C, sur l'exemple, cela s'écrirait :

```
#define N ...
2 #define ABV -1

4 typedef int Arbre; /* indice de la racine d'un arbre */

6 typedef struct {
    int elem
8   Arbre fd ;
    Arbre fg ;
10 } noeud ;

12 noeud T[N]; /* Tableau statique global */

14 // Premier element libre
   Arbre Libre;
```

Initialisation de la structure (chaînage des éléments libres) :

```
1 void InitArbres() {
    int i;
3   for (i=0; i<N-1; i++) {
        T[i].fd = i + 1;
5   }
    T[N-1].fd = ABV;
7   Libre = 0;
}
```

On peut alors écrire les algos des opérateurs :

```
Arbre ArbreVide () {
2   return ABV ; /* valeur arbitraire hors de 0..TAILLE_MAX */
}

4
int EstVide(Arbre a) {
6   return (a == ABV) ;
}
```

```

    }
8   Arbre FGauche(Arbre a) {
10  return (T[a].fg) ;
    }
12  Arbre FDroit(Arbre a) {
14  return (T[a].fd)
    }
16  char Racine(Arbre a) {
18  return (T[a].elem) ;
    }
20  Arbre NouveauNoeud(Arbre g, int x, Arbre d) {
22  Arbre N;
    N = Libre;
24  if (N <> ABV) {
    Libre = T[Libre].fd;
26  T[N].fg = g ;
    T[N].fd = d ;
28  T[N].elem = x ;
    return N ;
30  }
    }
32  void InsérerGauche(Arbre a, arbre g) {
34  T[a].fg = g ;
    }
36  void InsérerDroit(Arbre a, arbre d) {
38  T[a].fd = d;
    }
40  void Libérer(Arbre a) {
42  // Insertion en tête dans la liste d'arbres Libre
    T[a].fd = Libre;
44  Libre = a;
    }

```

Q2. Proposez maintenant une implémentation du type Arbre à l'aide de cellules mémoires allouées dynamiquement et chaînées par pointeurs.

Corrigé —

On déclare une structure contenant un champ «élément» `elem`, un champ «fils gauche» (`fg`) et un champ «fils droit» (`fd`) :

```

typedef struct noeud {
2  int elem ;
    struct noeud *fg;
4  struct noeud *fd;
    } Noeud;

```

Le type Arbre est alors un pointeur sur cette structure :

```

1 typedef Noeud *Arbre ;

```

Les primitives en C (attention, ne pas oublier de libérer la mémoire qui devient inaccessible) :

```
1  Arbre ArbreVide() {
    return NULL ;
3  }

5  Arbre NouveauNoeud (Arbre fg, int elem, Arbre fd) {
    Arbre A ;
7
    A = (Arbre) malloc(sizeof(struct noeud)) ;
9    A->elem = elem ;
    A->fg = fg ;
11   A->fd = fd ;
    return A ;
13  }

15  Arbre FGauche(Arbre A) {
    return (A->fg) ;
17  }

19  Arbre FDroit(Arbre A) {
    return (A->fd) ;
21  }

23  int Racine (Arbre A) {
    return (A->elem) ;
25  }

27  int EstVide (Arbre A) {
    return (A == NULL) ;
29  }

31  void InsérerGauche(Arbre A, Arbre G) {
    Libérer(A->fg) ;
33   A->fg = G ;
    }
35

37  void InsérerDroit(Arbre A, Arbre D) {
    Libérer(A->fd) ;
    A->fd = D ;
39  }

41  void Libérer (Arbre A) {
    free (A) ;
43  }
```
