Leçon 3 : Les arbres binaires

Université Virtuelle de Côte d'Ivoire

Table des matières

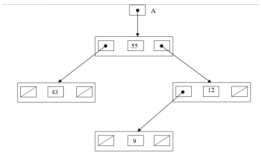
1- Généralité sur les arbres binaires

I

1.1- Définition

Un *arbre binaire* est un arbre 2-aire, chaque nœud possède 0, 1 ou 2 suivants.

Chaque nœud d'un arbre binaire peut posséder un *sous-arbre droit* et un *sous-arbre gauche* dont la racine est respectivement son fils droit et son fils gauche.



1.2- Création d'un arbre binaire

Type Nœud = Structure

filsG: Arbre

Info : Entier

filsD: Arbre

Fin Structure

- *Définir le type du pointeur :*

Type Arbre = ^Nœud

Déclarer une variable pointeur :

Var

A : Arbre

Si A est un pointeur sur la racine de l'arbre, alors :

A = NULL correspond à un arbre vide ;

A^.filsG pointe sur le fils gauche de A;

A^.filsD pointe sur le fils droit de A;

A^.info permet d'accéder au contenu du nœud

Application 1 :

II

Exercice

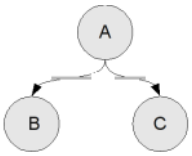
[solution n°1 p.12]

Un arbre binaire est composé :

- de trois sous arbres
- d'une racine
- de deux sous-arbres

Exercice

[solution n°2 p.12]



Veiller compléter la déclaration de la structure ci-dessus en respectant l'ordre d'apparition des nœuds :

_____ _____ = _____

filsG : _____

Info : chaîne

filsD : _____

Fin Structure

_____ _____ = _____ Nœud

Var

_____, _____, _____ : _____

2- Procédures et fonctions

III

2.1- Test de vacuité

```
Fonction vide({E}A: Arbre) : Entier
var
rep : entier
Début
// Vérifie sur l'arbre A est vide
Si A = Nil alors
rep ← 1 //si l'arbre est vide ramène 1
sinon
rep ← 0 //si l'arbre n'est vide ramène 0
finsi
Renvoyer rep
Fin
```

2.2- Accès au fils gauches

```
Fonction filsGauche ({E}A : Arbre) : Arbre
Var
res : Arbre
Début
res ← A
Si (vide(res)=0) Alors
res  ← res^.filsG // vérifier la fonction vide ramène 0 alors res reçoit le fils gauche
Finsi
Renvoyer res // renvoie la valeur du fils gauche
Fin
```

2.3- Accès au fils droit

Fonction filsDroit({E}A : Arbre) : Arbre

Var

res : Arbre

Début

res ← A

Si (vide(res)=0) Alors

res ← res^.filsD *// vérifier la fonction vide ramène 0 alors res reçoit le fils droit*

Finsi

Renvoyer res *// renvoie la valeur du fils droit*

Fin

2.4- Créer un nouveau nœud

Fonction nouvNoeud ({E}val : entier) : Arbre

var

A : Arbre

Début

Allouer (A)

A^.info ← val

A^.filsG ← Nil

A^.filsD ← Nil

Renvoyer A

Fin

2.5- Parcours préfixé

Parcours d'un arbre et affiche les informations des nœuds dans un ordre préfixé

Procédure parcours_prefixe({E}A : Arbre)

var

tmp : Arbre

Début

tmp ← A

Si (vide(tmp) = 0) alors // *si tmp n'est pas vide*

Afficher tmp^.info // *Afficher le contenu du nœud*

parcours_prefixe(filsGauche(tmp)) *// envoi la valeur du fils gauche*

parcours_prefixe(filsDroit(tmp)) *// envoi la valeur du fils droit*

Finsi

Fin

2.6- Parcours infixe

Un parcours infixe visite chaque nœud entre les nœuds de son sous-arbre de gauche et les nœuds de son sous-arbre de droite. C'est une manière assez commune de parcourir un arbre binaire de recherche, car il donne les valeurs dans l'ordre croissant.

Procédure parcours_infixe({E}A : Arbre)

var

tmp : Arbre

Début

tmp ← A

SI (vide(tmp) = 0) alors // *si tmp n'est pas vide*

parcours_infixe(filsGauche(tmp)) *// envoi la valeur du fils gauche*

Afficher tmp^.info // *Afficher le contenu du nœud*

parcours_infixe (filsDroit(tmp)) *//envoi la valeur du fils droit*

Finsi

Fin

2.7- Parcours postfixe ou suffixe

Dans un parcours postfixe ou suffixe, on affiche chaque nœud après avoir affiché chacun de ses fils

Procédure parcours_postfixe({E}A : Arbre)

var

tmp : Arbre

Début

tmp ← A

Si (vide(tmp) = 0) alors // *si tmp n'est pas vide*

parcours_postfixe(filsGauche(tmp)) *// envoi la valeur du fils gauche*

parcours_postfixe(filsDroit(tmp)) *//envoi la valeur du fils droit*

Afficher tmp^.info // *Afficher le contenu du nœud*

FINSI

FIN

2.8- Détruire un arbre

Fonction detruireArbre({E}A : Arbre) : Arbre

var

tmp : Arbre

Début

tmp ← A

Si (vide(tmp) = 1) alors *// vérifie si l'arbre est vide*

Afficher "Arbre Vide"

renvoyer tmp

Sinon

SI (vide(tmp^.filsG) = 1 ET vide(tmp^.filsD) = 1) alors *// vérifie si les deux fils sont vides*

Désallouer (tmp) *//Détruire le nœud*

Tmp ← NULL *//vide le nœud*

Afficher "nœud détruit"

renvoyer tmp

Sinon *// si les fils ne sont pas vides les détruire avantla suppression*

detruireArbre (tmp^.filsG)

Afficher "arbre gauche detruit"

detruireArbre(tmp^.filsD)

Afficher "arbre droit detruit"

Finsi

Finsi

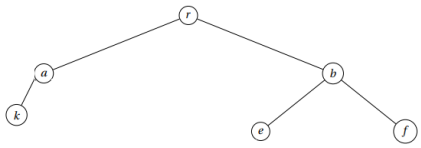
FIN

Application 2 :

IV

Exercice

[solution n°3 p.12]



Soit le graphique ci-dessus, Complétez l'algorithme suivant pour la création du nœuds r afficher le fils droit de r.

Fonction vide({E}val: Arbre) :

var

r : entier

Début

Si alors

r ← 1

sinon

r ← 0

finsi

Fin

Fonction filsDroit(: Arbre) :

Var

res : Arbre

Début

res ← val

Si ( ) Alors

res ←

Finsi

Fin

Algorithme CréationNoeud

Fonction ({E}val: Arbre) :

Fonction filsDroit(: Arbre) : Arbre

=

filsG :

Info : chaîne

filsD :

Fin Structure

TYPE =

Var

un_arbre_b : Arbre

un_arbre_f : Arbre

un_arbre_e : Arbre

Début

. ← 'r'

.filsG ←

.filsD ←

arb_f ←

Afficher

Fin

Solutions des exercices

> **Solution** n°1

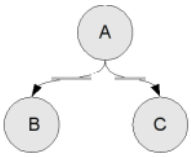
Exercice p. 4

Un arbre binaire est composé :

- ○ de trois sous arbres
- ○ d'une racine
- ◉ de deux sous-arbres

> **Solution** n°2

Exercice p. 4

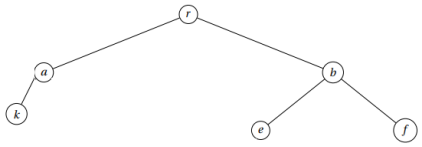


Veiller compléter la déclaration de la structure ci-dessus en respectant l'ordre d'apparition des nœuds :

```
Type Nœud = Structure
filsG : Arbre
Info : chaîne
filsD : Arbre
Fin Structure
Type Arbre = ^Nœud
Var
A,B,C : Arbre
```

> **Solution** n°3

Exercice p. 10



Soit le graphique ci-dessus, Complétez l'algorithme suivant pour la création du nœuds r afficher le fils droit de r.

Fonction vide({E}val: Arbre) : `Entier`

var

r : entier

Début

Si `val=Nil` alors

r ← 1

sinon

r ← 0

finsi

`Renvoyer r`

Fin

Fonction filsDroit(`{E}Val` : Arbre) : `Arbre`

Var

res : Arbre

Début

res ← val

Si (`vide(res)=0`) Alors

res ← `res^.filsD`

Finsi

`Renvoyer res`

Fin

Algorithme CréationNoeud

Fonction `vide`({E}val: Arbre) : `Entier`

Fonction filsDroit(`{E}Val` : Arbre) : Arbre

`TYPE Nœud = Structure`

filsG : `Arbre`

Info : chaîne

filsD : `Arbre`

Fin Structure

TYPE `Arbre` = `^Nœud`

Var

un_arbre_b : Arbre

un_arbre_f : Arbre

un_arbre_e : Arbre

Début

`Allouer(un_arbre_b)`

`un_arbre_b^.info` ← 'r'

`un_arbre_b^`.filsG ← `@un_arbre_e`

`un_arbre_b^`.filsD ← `@un_arbre_f`

arb_f ← `filsDroit(un_arbre_b)`

Afficher `arb_f^.info`

Fin