Cours: ASD 2

Chapitre 14: Les Arbres

Réalisé par:
Dr. Sakka Rouis Taoufik

1

Ch 14: Les Arbres

I. Concepts de base et définition

arbre, nœud, arête, branche et racine.

Un arbre est une collection de nœuds et d'arêtes soumise à certaines collections conditions.

Un nœud est une entité ou objet au sens général qui peut porter un nom et éventuellement des informations pertinentes liées à l'application.

Une arête est un lien entre deux nœuds.

Une branche est une séquence de nœuds distincts dans laquelle deux nœuds consécutifs sont reliés par une arête.

Une racine est un nœud particulier désigné d'une façon explicite parmi les nœuds formant l'arbre.

Propriété fondamentale : la propriété fondamentale qui caractérise la notion d'arbre est il existe exactement une seule branche entre la racine et chacun d'autres nœuds de l'arbre. S'il existe plus qu'une branche ou pas de branche alors il s'agit d'un **graphe** et non pas d'un arbre.

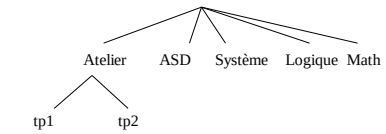
2

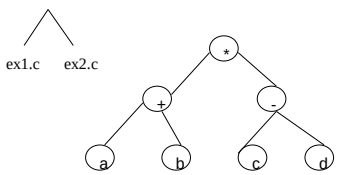
Ch 14: Les Arbres

II. Représentation de la structure d'arbre

2.1 Exemples :

1- Un support de mémorisation à long terme (disquette ,disque dur ,cd), sous MS-DOS ou sous Windows peut être modélisé par arbre.

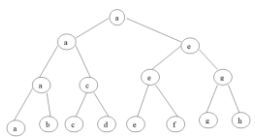




2 - Les expressions peuvent être modélisées par des arbres. Par exemple, l'expression numérique (a+b)*(c-d) est traduite par l'arbre suivante. Les nœuds traduisent soient des opérandes soient des opérateurs

3- Tournois sportifs ou par exemple un tournoi de Boxe ou deping-pong.

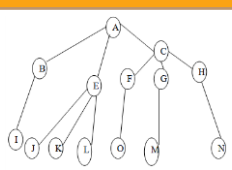
➔ La racine modélise le groupement du tournoi



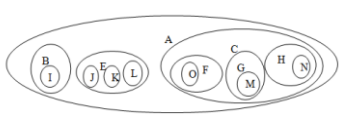
3

Ch 14: Les Arbres

II. Représentation de la structure d'arbre



Sous forme d'un graphe.



Sous forme des ensembles imbriqués

Sous forme d'indentation

(A(B(I)),(E(J,K,L)), (C(F(O),G(M),H(N)))

Sous forme de parenthèses imbriquées

➔Les quatre représentations sont équivalentes. Dans la suite de ce cours, on va utiliser la représentation sous forme graphique. La représentation sous forme d'indentation peut être obtenu par un sous programme.

4

Ch 14: Les Arbres

II. Représentation de la structure d'arbre

->Direction sur les arêtes, père, fils, nœuds terminaux et nœuds non terminaux.
Une arête peut être dirigée :
Soit de haut en bas : de la racine vers un nœud : arc
Soit de bas vers haut : du nœud vers la racine
Soit dans les deux sens.

Tous les nœuds de l'arbre (excepté la racine) possèdent un père situé au dessus de nœud considéré les fils (ou les enfants) d'un nœud sont situés au dessous de celui-ci.
Les nœuds terminaux (ou encore nœuds externes ou encore feuilles) sont des nœuds sans descendances.
Les nœuds non terminaux(ou encore nœuds internes) sont des nœuds avec descendances.

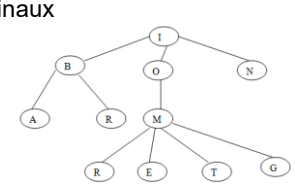
->Sous arbres, forêt, arbre ordonné, niveaux, hauteur, degré d'un nœud et arbre binaire.
Tout nœud est la racine d'un sous arbre formé de lui-même et sa descendance.

5

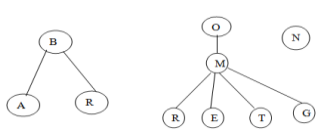
Ch 14: Les Arbres

II. Représentation de la structure d'arbre

L'arbre suivante comporte :
7 sous arbres formés d'un seul nœud : nœuds terminaux
un sous arbre de trois nœuds B, A, R
un sous arbre de cinq nœuds dont la racine est M
un sous arbre de six nœuds dont la racine est o



une forêt est un ensemble d'arbres.
➔ Si on supprime la racine I et les arcs partant de cette racine, on obtient les trois arbres suivants



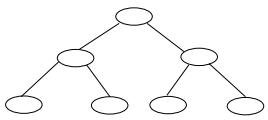
Remarque : A partir d'une forêt on peut construire un arbre. Les enfants d'un nœud peuvent être ordonnés. Ceci caractérise **un arbre ordonné**

6

Ch 14: Les Arbres

II. Représentation de la structure d'arbre

➔Règle pour ordonner d'une manière systématique les enfants de chaque nœud formant l'arbre.



- Le niveau d'un nœud est le nombre de nœud qui le sépare de la racine (excepté lui-même).
- La hauteur d'un arbre est son niveau maximum. Le degré d'un nœud est le nombre de ses enfants.
- Un arbre binaire comporte deux types de nœuds :
  - Les nœuds terminaux ou externes.
  - Les nœuds non terminaux ou internes ayant au maximum 2 enfants

Illustration :

- un tournoi de boxe peut être modélisé par un arbre binaire dont les nœuds terminaux modélisent les participants à ce tournoi.
- Les expressions (numériques ou booléennes) peuvent être modélisées par des arbres binaires dont les nœuds internes sont des opérateurs et les nœuds externes sont des variables ou constantes.

Les arbres binaires sont largement utilisés, dans la suite de ce chapitre, on va s'intéresser uniquement aux arbres binaires.

7

Ch 14: Les Arbres

III. Opérations de base sur les arbres binaires

creer_arbre : permet de créer un arbre binaire vide ->création

arbre_vide : permet de tester la vacuité d'un arbre binaire->consultation

construire : permet de fabriquer un arbre binaire à partir d'une racine et de deux sous arbres de gauche et de droit ->création

gauche : permet de fournir le sous arbre de gauche d'un arbre binaire non vide ->consultation

droite : permet de fournir le sous arbre de droite d'un arbre binaire non vide ->consultation

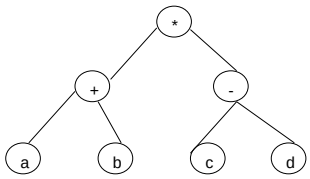
8

Ch 14: Les Arbres

IV. Représentation physique des arbres binaires

1. Représentation chaînée

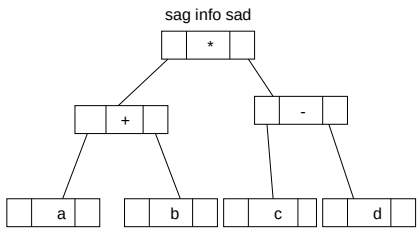
L'expression (a+b)*(c-d) peut être représenté par l'arbre binaire suivant :



Les nœuds internes modélisent les opérations. Tandis que les nœuds externe modélisent les variables

info : c'est l'information portée par les nœuds.
sag : sous arbre de gauche
sag : sous arbre de gauche

Traduction en C :




```
struct noeud  {
  char info ;
  struct noeud * sag ;
  struct noeud * sad ;
};
```

➔ Pour repérer un arbre il suffit de repérer sa racine.

9

Ch 14: Les Arbres

IV. Représentation physique des arbres binaires

1. Représentation chaînée

TDA d'un arbre binaire matérialisé par une représentation chainée.

Partie interface :arbre.h

```
struct noeud {
  char info;
  struct noeud *sag;
  struct noeud *sad;
};

struct noeud *creer_arbre (void);
unsigned arbre_vide (struct noeud*);
struct noeud *construire (char, struct noeud*, struct noeud*);
struct noeud *gauche (struct noeud*);
struct noeud *droite (struct noeud*);
char lire_racine (struct noeud*);
```

10

Ch 14: Les Arbres

IV. Représentation physique des arbres binaires

1. Représentation chaînée

Partie implémentation :arbre.c

```
#include <stdlib.h>
#include <assert.h>
#include "Arbre.h"
struct noeud * creer_arbre (void) {
   return(NULL);
}
unsigned arbre_vide (struct noeud *ab) {
  return(ab==NULL);
}
struct noeud* construire (char info, struct noeud*g, struct noeud*d) {
  struct noeud* ab;
  ab= (struct noeud*) malloc (sizeof(struct noeud));
  ab->info=info;
  ab->sag=g;
  ab->sad=d;
  return(ab);
}
```

11

Ch 14: Les Arbres

IV. Représentation physique des arbres binaires

1. Représentation chaînée

```
struct noeud* gauche (struct noeud* ab) {
  assert(!arbre_vide(ab));
  return(ab->sag);
}

struct noeud* droite (struct noeud* ab)  {
  assert (! arbre_vide(ab));
  return(ab->sad);
}

char lire_racine (struct noeud *ab) {
  assert (! arbre_vide(ab));
  return ab->info;
}
```

12

Ch 14: Les Arbres

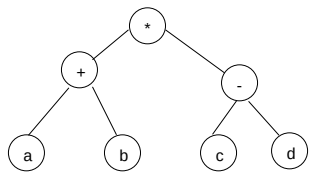
IV. Représentation physique des arbres binaires

1. Représentation chaînée

Partie utilisation

Ecrire un programmeC permettant de construire un arbre binaire traduisant l'expression (a+b)*(c+d) en utilisant les services offerts par TDA arbre.

```
#include "arbre.h"
void main (void) {
  struct  noeud* expr;
  expr=construire('*',construire('+',construire('a',NULL,NULL),
  construire('b',NULL,NULL)),
  construire ('-',construire('c',NULL,NULL),construire('d',NULL,NULL))) ;
}
```



➔ L'activation imbriquée de la fonction construire permet d'éviter le passage par des variables intermédiaires et par conséquent l'utilisation de l'affectation. Ce style de programmation est connu sous le nom de programmation fonctionnelle encouragée notamment par le langage LISP

13

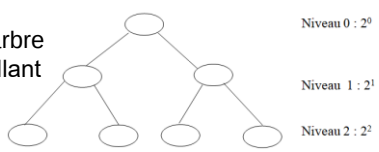
Ch 14: Les Arbres

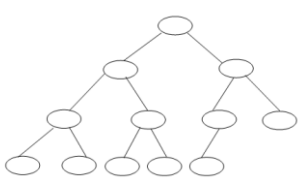
IV. Représentation physique des arbres binaires

2. Représentation contiguë

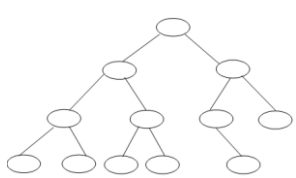
La représentation contiguë peut être envisagée pour les arbres binaires complets (full) et les arbres binaires quasi-complet.

Un arbre binaire complet ou encore plein est un arbre qui comporte à chaque niveau 2^i nœuds avec i allant de 0 à la hauteur de l'arbre.





Arbre binaire quasi-complet



Arbre binaire quasi-complet

14

Ch 14: Les Arbres

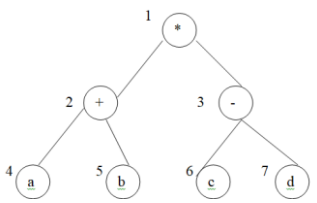
IV. Représentation physique des arbres binaires

2. Représentation contiguë

Pour mémoriser un arbre binaire complet dans un tableau, il suffit de numéroter les nœuds formant l'arbre en largeur.

Pour tout nœud de position i (au sein du tableau), son fils de gauche est donné par 2i et son fils de droite est donnée par 2i+1. Ce ci est valable uniquement pour les nœuds internes.

-Illustration : soit l'expression (a+b)*(c-d) celle-ci peut être traduit par l'arbre



*	+	-	a	b	c	d

15

Ch 14: Les Arbres

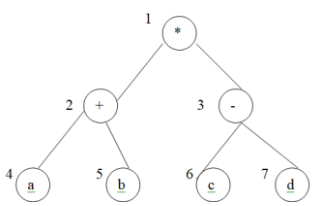
IV. Représentation physique des arbres binaires

2. Représentation contiguë

Dans le cas ou l'arbre binaire est plein ou quasi complet, la représentation contiguë apporte les bénéfices suivants :

- économie de l'espace mémoire par rapport à l'espace exigé par la représentation chainée.
- accès rapide (ou direct) à un nœud via son indice.
possibilité d'accéder au père d'un fils donné si le fils à pour indice i alors son père a pour indice i/2(division entière) : l'accès est gratuit.

Remarque : Dans une représentation chaînée, la possibilité de remonter au père à partir d'un fils donné nécessite un pointeur supplémentaire appelé père.



*	+	-	a	b	c	d

16

Ch 14: Les Arbres

V. Algorithme de parcours des arbres binaires quelconque

Un algorithme de parcours d'un arbre binaire est un procédé permettant de visiter chaque nœud de l'arbre une et une seul fois.

Lors de la visite d'un nœud, un traitement(ou action) sera effectué.

L'algorithme de parcours est indépendant de l'action à effectuer sur chaque nœud visité. On distingue :

Les algorithmes de parcours par niveau.
Les algorithmes de parcours par récursifs

17

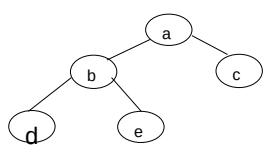
5.1 Parcours par niveau

Ch 14: Les Arbres

V. Algorithme de parcours des arbres binaires quelconque

1. Parcours par niveau

Le parcours par niveau appliqué à l'arbre donné ci-dessus visite les nœuds de cet arbre de l'ordre suivant : a,b,c,d et e.



Le parcours par niveau peut avoir un sens, par exemple un arbre modélisant un tournoi de tennis, le parcours par niveau sur cet arbre consiste à visiter le vainqueur, finalistes, demi finalistes.

Dans la suite, on va proposer une réalisation de l'algorithme de parcours par niveau sous forme d'une procédure.

18

Ch 14: Les Arbres

V. Algorithme de parcours des arbres binaires quelconque

1. Parcours par niveau

Hypothèse:
L'arbre à parcourir est concrétisé par une représentation chaînée.

Rappel :

```
struct noeud {
  char info ;
  struct noeud*sag ;
  struct noeud*sad ;
} ;
```

L'action à effectuer lors de la visite à un nœud consiste tout simplement à afficher l'information portée par ce nœud.

```
void visiter (struct noeud*t) {
   printf(" %c " ,t->info) ;
}
```

19

Ch 14: Les Arbres

V. Algorithme de parcours des arbres binaires quelconque

1. Parcours par niveau

La réalisation de l'algorithme de parcours par niveau nécessite une file d'attente. Celle-ci va mémoriser des nœuds ou des arbres. Chaque nœud est désigné par un pointeur.

Besoin : une file d'attente dont chaque élément est un sous arbre.

Rappel : file d'entiers

```
struct element {
  int cle ; /*mémoire des entiers*/
  struct element*suivant ;
} ;
struct file {
  struct element*tete ;
  struct element*queue ;
} ;
```

Adaptation par rapport aux arbres :

```
struct element {
  struct noeud * cle;
  struct element*suivant ;
} ;
struct file {
  struct element*tete ;
  struct element*queue ;
} ;
```

20

Ch 14: Les Arbres

V. Algorithme de parcours des arbres binaires quelconque

1. Parcours par niveau

On suppose qu'on a notre disposition un objet abstrait file d'attente permettant de mémoriser des sous arbres. C'est-à-dire chaque élément est de type (struct noeud *).

```
void parcourir_niveau(struct noeud*t) {
  /*t est la racine de l'arbre à visiter*/
  cree_file() ;
  enfiler(t) ;
  while( ! file_vide()) {
    t=premier();
    visiter(t);
   defiler();
   if(t->sag) ; /*t->sag !=NULL*/
      enfiler(t->sag);
   if(t->sad) /*t->sad !=NULL*/
     enfiler(t->sad);
   }/*fin while*/
}/*fin parcours niveau*/
```

21

Ch 14: Les Arbres

V. Algorithme de parcours des arbres binaires quelconque

2. Parcours récursifs

Un parcours récursif consiste à ordonner les trois opérations suivantes

- Visiter la racine R
- Parcourir le sous arbre de gauche G
- Parcourir le sous arbre de droite D

Application :
(R,G,D) on a 6 algorithmes possibles 3 !=6

Les parcours couramment utilisés sont au nombre de trois :
Parcours **infixé** ou en ordre GRD.
Parcours **préfixé** ou en pré ordre RGD.
Parcours **post fixé** ou en post- ordre GDR.

Ces trois parcours infixé, préfixé et post fixé sont appelés des parcours récursifs (contrairement au parcours par niveau).

22

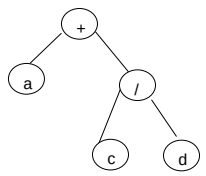
Ch 14: Les Arbres

V. Algorithme de parcours des arbres binaires quelconque

2. Parcours récursifs

Exemple :soit l'expression a+(c/d) une telle expression peut être modélisée par l'arbre binaire suivant

Parcours infixé :GRD
a+c/d
ce type de parcours correspond à l'écriture infixé de la même expression sans parenthèses.



Parcours préfixé :RGD
+a/cd

Parcours post fixé :GDR

acd/+

23

Ch 14: Les Arbres

V. Algorithme de parcours des arbres binaires quelconque

2. Parcours récursifs

6.1 Parcours infixé
- Loi GRD

```
void parcours_infixe(struct noeud*t) {
    /*t est la racine de l'arbre à visiter*/
   if(t==NULL)
      ; /*instruction vide, cas trivial : rien  à faire*/
  else  {/*cas générale*/
       parcours_infixe(t->sag) ;
       visiter(t) ;
       parcours_infixe(t->sad) ;
   }
}
```

➔simplification

```
void parcours_infixe (struct noeud*t) {
   if(t)   {    /*if (t!=NULL)  */
       parcours_infixe(t->sag) ;
       visiter(t)
       parcours_infixe(t->sad) ;
   }
}
```

24

Ch 14: Les Arbres

V. Algorithme de parcours des arbres binaires quelconque

2. Parcours récursifs

6.2 Parcours préfixé
loi :RGD

```
void parcours_prefixe(struct noeud*t) {
   if(t) {
        visiter(t) ;
        parcours_prefixe(t->sag) ;
        parcours_prefixe(t->sad) ;

   }
}
```

25

Ch 14: Les Arbres

V. Algorithme de parcours des arbres binaires quelconque

2. Parcours récursifs

6.3 Parcours postfixé
loi :GDR

```
void parcours_posfixe (struct noeud*t) {
   if(t) {
        parcours_posfixe(t->sag) ;
        parcours_posfixe(t->sad) ;
        visiter(t) ;
    }
}
```

26

Ch 14: Les Arbres

V. Algorithme de parcours des arbres binaires quelconque

2. Parcours récursifs

Remarques importantes:
Les trois solutions récursives précédentes traduisent naturellement les lois (ou règles) Elles sont **élégantes**.
Pour une solution récursive, il faut comprendre son principe et non pas ses détails.
Les trois solutions précédentes engendrent des processus récursifs modélisés par des arbres binaires car chaque appel est susceptible d'introduire des appels récursifs.

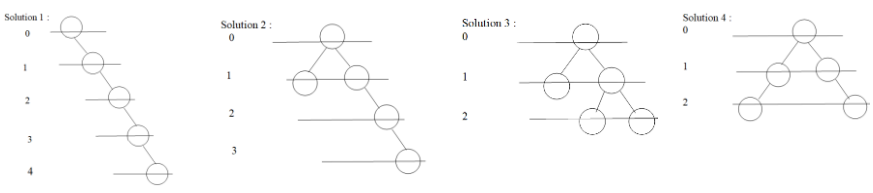
La structure de données Arbre est une structure récursive et par conséquent la plupart des opérations applicables sur cette structure admettent des solutions récursives naturelles.
On peut paramétrer d'avantage les algorithmes de parcours en proposant un paramètre fonctionnel.

27

Ch 14: Les Arbres

VII. Construction d'arbres équilibrés

Problème : on a n nœuds à répartir plus au moins équitablement sur un arbre binaire. L'arbre obtenu est un **arbre binaire équilibré** c'est-à-dire sa hauteur est minimale.
Illustration: n=5, donc on a 5 nœuds à répartir sur un arbre binaire.



- Les solution 3 et 4 forment deux arbres équilibrés.
- La solution 4 est un arbre parfaitement équilibré

Un arbre est **parfaitement équilibré** est un arbre obéissent à la propriété suivante :
- **Pour tout nœud de l'arbre**, le nombre de nœuds de son sous arbre de gauche et de son sous arbre de droite **différé au plus d'un nœud**. À ne pas confondre avec les arbres complets ou quasi complet.

28

Ch 14: Les Arbres

VII. Construction d'arbres équilibrés

Dans la suite, on va proposer un algorithme permettant de construire un arbre équiibré formé de n nœuds. Dont **n** est un paramètre d'entrée.

Règles de répartition :
Prendre un nœud comme racine
Générer un sous arbre de gauche de taille ng=n div 2 neuds
Générer un sous arbre de droite de taille nd=n-ng-1

29

Ch 14: Les Arbres

VII. Construction d'arbres équilibrés

```
struct noeud*arbre(unsigned n) {
struct noeud  * nouveau_noeud;
unsigned ng; /* nombre de noeuds à répartir sur le sous-arbre de gauche*/
unsigned nd ; /* nombre de noeuds à répartir sur le sous-arbre de droite*/
char x ;/*information portée par  un neud, elle est fournie au clavier*/
if(n==0)
nouveau_noeud=NULL ;
else
{ /*cas genera*/
   nouveau_noeud=(struct-noeud)malloc(sizeof(struct noeud ));
printf("x=");
scanf("%c",&x);
nouveau_noeud-> info=x ;
ng=n/2 ;
nd=n-ng-1 ;
nouveau_nœud->sag=arbre(ng) ;
nouveau_nœud->sad=arbre(nd) ;
}
return(nouveau_nœud) ;
}
```

30

Ch 14: Les Arbres

VIII. Principe diviser pour résoudre

Un tel principe est connu encore sous le nom diviser pour régner. Il permet de décomposer un problème non trivial en plusieurs sous problèmes, **souvent disjoints** et plus facile à résoudre.

L'application de ce principe en algorithmique, permet souvent d'obtenir des algorithmes **efficaces** et **élégants**.

Pour résoudre un problème non-trivial, il faut le décomposer en plusieurs sous-problèmes plus faciles à résoudre : **règle récursive**.
En algorithmique le problème est souvent décomposé en deux sous-problème.

Exemples : Problèmes décomposés en deux sous problèmes **disjoints**.

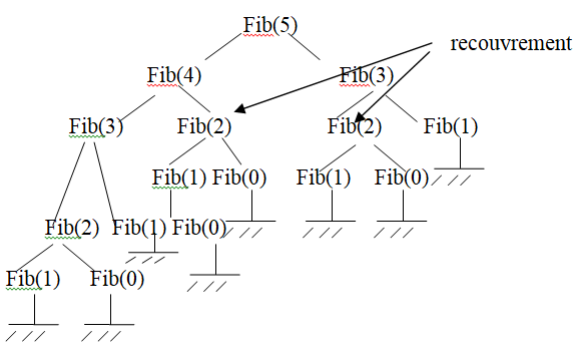
- Recherche dichotomique d'une information dans une table ➔solution itérative.
- Parcours récursifs des arbres binaires : GRD, RGD, GDR
- Taille d'un arbre binaire,
- Construction d'un arbre binaire équilibré

31

Ch 14: Les Arbres

VIII. Principe diviser pour résoudre

Contre exemple : Problème décomposé en deux sous problèmes non disjoints.
Suite Fibonaccci



Il y a recouvrement ceci explique l'inefficacité de cette solution.
➔ Solution : programmation dynamique. Éviter les calculs déjà faits.

32

Ch 14: Les Arbres

IX. Exercices d'application

Exercice 1.

A. Établir la structure de données nœud_Int permettant de matérialiser un arbre d'entiers.

B. Écrire une fonction qui calcule le maximum dans un arbre de recherche d'entier.

C. Écrire un sous-programme permettant de calculer la taille d'un arbre binaire matérialisé par une représentation chaînée.

Règle de calcul:

Si l'arbre binaire est vide alors le nombre de neouds est 0

Si non 1+taille de sous arbre de gauche + taille de sous arbre de droite.

D. Écrire un sous-programme permettant de calculer le nombre des terminaux d'un arbre binaire matérialisé par une représentation chaînée.

E. Écrire une fonction prenant un arbre binaire et rendant sa hauteur, c'est à dire le nombre d'éléments contenus dans la plus longue branche.

Exercice 2. Écrire une fonction définie par récurrence qui calcule le résultat du terme décrit par un arbre binaire qui matérialise une expression algébrique dans R (l'expression est supposée valide: 6+5*4-8). **NB.** Ne considérer pas la priorité des operateurs.

Démarche : Quelle est la condition d'arrêt ? Comment faut-il placer les appels récurrents ?

33

Ch 14: Les Arbres

IX. Exercices d'application

```
int evaluer_expression(struct noeud* ab) {
   int droite, gauche;
   /* si le nœud est un terminal, renvoyer sa valeur*/
   if (ab->sag == NULL && ab->sad == NULL)
     return atol(& ab->info);
   /* return 48- ab->info ;  sol 2 */
else {
      gauche = evaluer_expression(ab->sag);
      droite = evaluer_expression(ab->sad);
      switch(ab->info) {
         case '+': return gauche + droite;
         case '-': return gauche - droite;
         case '*': return gauche * droite;
         case '/': return gauche / droite;
      }
   }
}
```

```
float evaluer_Relle(struct noeud* ab) {
   float droite, gauche;
   /* si le nœud est un terminal, renvoyer sa valeur */
   if (ab->sag == NULL && ab->sad == NULL)
     return atof(& ab->info);
   /* sinon, évaluer les sous-expressions et appliquer
l'opérateur */
   else {
      gauche = evaluer_Relle(ab->sag);
      droite = evaluer_Relle(ab->sad);
      switch(ab->info) {
         case '+': return gauche + droite;
         case '-': return gauche - droite;
         case '*': return gauche * droite;
         case '/': return gauche / droite;
      }
   }
}
```

34