Chapitre 1

Arbres

Un arbre est un graphe particulier, sans boucle, chaque sommet n'a qu'un prédecesseur. L'implantation peut être la même que celle d'un graphe, on n'a pas besoin de l'ensemble des sommets, la racine de l'arbre suffit.

1.1 Les opérations de bases

Il nous faut

- la racine
- la liste des branches qui en dépendent
- l'adjonction d'une nouvelle branche à la racine
- la suppression d'une branche à la racine
- le test d'arbre vide
- le test pas de fils
- la création d'un arbre réduit à sa racine

Remarquons que la liste des branches peut être ordonnée ou non.

1.1.1 Nommons

Soit a un arbre, on définit

- r =racine(a)
- eb =branches(a)
  - b =nimporte(eb)
  - b =tete(eb)
  - eb' =corps(eb)
  - pour toute branche b de branches(a)
  - etc.
- a' =greffe(b, a)
- a' =taille(b, a)
- x =vide(a)
- y =vide(branches(a))
- a =plant(r)

1.1.2 Convenons que

- une **branche** est un **arbre**
- une **feuille** est un **arbre**

1.2 Une arborescence de fichiers

Ecrire les programmes permettant de gerer une arborescence de fichiers style Unix ou VMS.

1.2.1 Liste des opérations

- **mkdir** création d'un directoire
- **ls** liste du directoire
- **cd** changement de directoire
- **rm** suppression d'un fichier
- **mv** recopie d'un fichier
- **pwd** où suis-je?

1.3 Expression arithmétique

Comment se calcule l'expression

$$1 + 2 * 3 + (4 + 5 * 6) * 7$$

Construire l'arbre syntaxique associé.

1.3.1 Grammaire et arbre syntaxique

La grammaire canonique des expressions arithmétiques s'ecrit

$$\begin{aligned}
E &\rightarrow E + T && (1.1)\\
E &\rightarrow T && (1.2)\\
T &\rightarrow T * F && (1.3)\\
T &\rightarrow F && (1.4)\\
F &\rightarrow < chiffre > && (1.5)\\
F &\rightarrow (E) && (1.6)
\end{aligned}$$

L'expression peut être obtenue par la suite de dérivations:

$$\begin{aligned}
E_0 &\rightarrow E_1 + T_1\\
E_1 &\rightarrow E_2 + T_2
\end{aligned}$$

$$\begin{aligned}
E_2 &\rightarrow E_3 + T_3 \\
E_3 &\rightarrow T_4 \\
T_4 &\rightarrow F_1 \\
F_1 &\rightarrow < chiffre > \\
T_3 &\rightarrow T_5 * F_2 \\
etc.
\end{aligned}$$

1.3.2 Construction de l'arbre et évaluation

En supposant que l'arbre est construit écrire un programme qui évalue l'expression. Ecrire les fonctions EEpT, ET, TTfF, TF, FpEp, et Fc qui permettent de construire l'arbre.

EEpT(E, T)
arbre E, T;

{
return(grefq(grefq(plant('+'),E),T));
}

la fonction grefq() **gref**fe en **q**ueue de liste.

Chapitre 2

Arbres binaires

Les arbres binaires sont trés utilisés

2.1 Opérations de base

- noeud rac(a)
- arbin fg(a)
- arbin fd(a)
- arbin enrac(g, r, d)
- arbin abvide()

2.2 Arbres binaires triés

2.2.1 Insertion

On ne permet plus l'enracinement mais uniquement l'insertion qui sera faite à la bonne feuille.

arbin insere(x, a)
    if vide(a) **return**(planb(x))
    if (x ≤rac(a)) insere(x,fg(a))

else insere(x,fd(a))

2.2.2 Suppression

Supprimer une feuille est facile, supprimer un noeud quelconque se fera de la manière suivante:

arbin supprimme(x, a)
if vide(a) **return**(a)
if (x <rac(a)) supprimme(x,fg(a))
if (x >rac(a)) supprimme(x,fd(a))
if (x =rac(a)) enrac(fg(a),mini(fd(a)),otermini(fd(a))

2.3 Arbres binaires triés équilibrés

Nous dirons qu'un arbre est équilibré si

- ses deux branches sont équilibrées
- les hauteurs gauches et droites ne diffèrent pas de plus de 1.

L'arbre vide est évidemment équilibré.

2.3.1 Rééquilibrage

Un arbre quelconque ne se rééquilibre que par reconstruction à l'aide de l'opération d'*insertion équilibrée*. Il faudra veiller àce que toute opération garde l'équilibre.
Un arbre **déséquilibré par l'insertion ou la suppression** d'une feuille se rééquilibre facilement de la manière suivante:

Quatre cas sont à envisager, le déséquilibre est dû à

1. la branche **gauche** de la branche **gauche**

2. la branche **droite** de la branche **gauche**

3. la branche **gauche** de la branche **droite**

4. la branche **droite** de la branche **droite**

2.3.2 Opérations licites

Nous définirons *insereq*, *supprimmeq*.