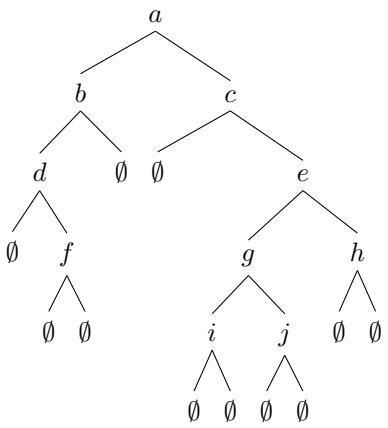
Corrigé des TD 5 et 6

Exercice 1.



Question 1. *Donner l'ordre de visite des nœuds de l'arbre binaire ci-dessus, suivant les différents types de parcours :*

- parcours préfixe : a, b, d, f, c, e, g, i, j, h
- parcours infixe : d, f, b, a, c, i, g, j, e, h
- parcours suffixe : f, d, b, i, j, g, h, e, c, a
- parcours en largeur : a, b, c, d, e, f, g, h, i, j

Question 3. *Pour chacun des quatre ordres de parcours, écrire une procédure qui prend en argument un arbre binaire* `A` *et affiche ses nœuds dans cet ordre.*

Algorithmes récursifs vus en cours :

```
affichagePrefixe(A : arbreBinaire)
    si A <> None alors
        afficher A->valeur
        affichagePrefixe(A->gauche)
        affichagePrefixe(A->droit)

affichageInfixe(A : arbreBinaire)
    si A <> None alors
        affichageInfixe(A->gauche)
        afficher A->valeur
        affichageInfixe(A->droit)

affichageSuffixe(A : arbreBinaire)
    si A <> None alors
        affichageSuffixe(A->gauche)
        afficher A->valeur
        affichageSuffixe(A->droit)
```

Voici une procédure d'affichage en largeur légèrement différente de celle du cours. L'argument est une file (éventuellement vide) d'arbres non vides. On affiche les valeurs des racines tout en enregistrant les enfants dans une nouvelle file (noeuds du niveau suivant) puis on fait un appel récursif sur cette nouvelle file. Pour afficher en largeur un unique arbre `A` on exécute `F : File ; F = initFile() ; F = enfiler(F, A) ; affichageLargeur2(F, 0)`. Chaque appel de la procédure affiche un niveau de l'arbre.

```
affichageLargeur2(F : File, N : entier)
    si fileVide(F) alors retourner
    afficher 'Niveau : ' + chaîne(N)
    G : file ; G = initFile()
```

```
    tant que non fileVide(F) faire
        A : pointeur sur noeud ; A = tete(F)
        F = defiler(F)
        afficher A->valeur
        si A->gauche <> None alors
            G = enfiler(G, A->gauche)
        si A->droit <> None alors
            G = enfiler(G, A->droit)
    affichageLargeur2(G, N+1)
```

Remarquons qu'en fait il n'y a pas besoin de séparer les files par niveau : on peut prendre `G = F` et se passer de l'appel récursif, les nœuds seront enfilés niveau après niveau de la même manière. C'est ce que fait la procédure vue en cours.

Question 4. *Donner une procédure récursive qui renvoie le nombre de nœuds d'un arbre binaire. Modifier cette procédure pour retourner le nombre de nœuds internes.*

```
nombreNoeuds(A : arbreBinaire) : entier
    si A == None alors retourner 0
    sinon
        retourner 1 + nombreNoeuds(A->gauche) + nombreNoeuds(A->gauche)

nombreInternes(A : arbreBinaire)
    si A == None ou (A->gauche == None et A->droit == None) alors
        retourner 0
    sinon
        retourner 1 + nombreInternes(A->gauche) + nombreInternes(A->gauche)
```

Question 5. *On considère la procédure récursive suivante qui calcule le nombre de feuilles d'un arbre binaire. Calculer le nombre d'appels récursifs de cette procédure en fonction du nombre de nœuds internes.*

```
nombreFeuilles1(A : arbreBinaire) : entier
    si A = None alors
        retourner 0
    sinon
        si A->gauche = None et A->droit = None alors
            retourner 1
        sinon
            retourner nombreFeuilles1(A->gauche) + nombreFeuilles1(A->droit)
```

On arrive aux appels récursifs exactement quand la racine de A est interne. Le nombre d'appels récursifs est donc le double du nombre de nœuds internes.

Question 6. *Écrire une procédure récursive qui calcule le nombre de feuilles d'un arbre binaire* supposé non vide*, sans faire d'appel récursif avec l'arbre vide.*

```
nombreFeuilles2(A : arbreBinaire) : entier
    N : entier ; N = 0
    si A->gauche == None et A->droit == None alors
        retourner 1
    si A->gauche != None alors
        N = N + nombreFeuilles2(A->gauche)
    si A->droit != None alors
        N = N + nombreFeuilles2(A->droit)
    retourner N
```

Question 7. *Calculez le nombre d'appels récursifs de la procédure* `nombreFeuilles2` *en fonction du nombre de nœuds.*

La procédure est appelée une fois pour chaque nœud interne. Pour un arbre dégénéré (filiforme) c'est deux fois mieux que la première procédure, pour un arbre localement complet c'est (presque) pareil.

Question 8. *Écrire une procédure qui prend en argument un arbre binaire* `A` *et retourne une copie de celui-ci, c'est-à-dire, un arbre identique à* `A` *mais qui n'a aucun nœud commun avec* `A`.

```
copieArbre(A : arbreBinaire) : arbreBinaire
    si A == None alors retourner None
    B : arbreBinaire ; B = Nouveau(noeud)
    B->valeur = A->valeur
    B->gauche = copieArbre(A->gauche)
    B->droit = copieArbre(A->droit)
    retourner B
```

Question 9. *Écrire une procédure* itérative *qui affiche les nœuds d'un arbre binaire dans l'ordre préfixe, en utilisant une pile.*

```
affichagePrefixeIt(A : arbre binaire)
    P : pile ; P = initPile()
    P = empiler(P, A)
    tant que non pileVide(P):
        B = sommet(P)
        afficher B->valeur
        P = depiler(P)
        si B->droit <> None alors
            P = empiler(P, B->droit)
        si B->gauche <> None alors
            P = empiler(P, B->gauche)
```

Exercice 2.

Question 1. *Classer les nœuds de l'arbre binaire de l'exercice 1 par niveau.*

- niv. 0 : a
- niv. 1 : b, c
- niv. 2 : d, e
- niv. 3 : f, g, h
- niv. 4 : i, j

Question 2. *Écrire une procédure récursive qui affiche tous les nœuds de niveau* `k` *d'un arbre binaire* `A`*.*

Si `k > 0`, les nœuds de niveau `k` dans `A` sont exactement les nœuds de longueur `k-1` dans les sous-arbres gauche et droit de `A`.

```
afficheNiveau(A : arbreBinaire, k : entier)
    si A == None alors retourner
    si k == 0 alors
        afficher A->valeur
    sinon
        afficheNiveau(A->gauche, k-1)
        afficheNiveau(A->droit, k-1)
```

Question 3. *Calculer la longueur de cheminement de l'arbre binaire de l'exercice 1, c'est-à-dire la somme des profondeurs de ses nœuds.*

La longueur de cheminement de cet arbre vaut 23.

Question 4. *Écrire une procédure* `LC` *qui calcule la longueur de cheminement d'un arbre. On utilisera une procédure auxiliaire récursive* `LCaux(B,p)` *qui calcule la somme des profondeurs dans* `A` *des nœuds d'un sous-arbre* `B` *dont la racine est de niveau* `p`*.*

```
LCaux(B : arbreBinaire, p : entier) : entier
    si B == None alors retourner 0
    retourner p + LCaux(B->gauche, p+1) + LCaux(B->droit, p+1)

LC(A : arbreBinaire) : entier
    retourner LCaux(A, 0)
```

Question 5. *Calculer la longueur de cheminement externe de l'arbre binaire de l'exercice 1, c'est-à-dire la sommes des profondeurs de ses feuilles.*

La longueur de cheminement externe de cet arbre vaut 14.

Question 6. *Écrire une procédure récursive qui calcule la longueur de cheminement externe d'un arbre binaire. Comme précédemment on pourra utiliser une procédure auxiliaire.*

```
LCEaux(B : arbreBinaire, p : entier) : entier
    si B == None alors retourner 0
    si B.gauche == None et B.droit == None: retourner p
    retourner LCEaux(B->gauche, p+1) + LCEaux(B->droit, p+1)

LCE(A : arbreBinaire) : entier
    retourner LCEaux(A, 0)
```

Question 7. *Écrire une procédure* itérative *qui calcule la longueur de cheminement externe d'un arbre binaire, en effectuant un parcours en largeur de l'arbre à l'aide d'une file. On enfilera chaque nœud avec sa profondeur.*

Rappelons la procédure d'affichage en largeur d'un arbre binaire :

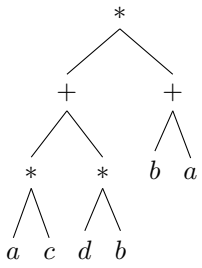
```
affichageLargeur(A : arbreBinaire)
    F : file ; F = initFile() ; F = enfiler(F, A)
    tant que non fileVide(F) faire
        B : pointeur sur noeud ; B = tete(F)
        F = defiler(F)
        afficher B->valeur
        si B->gauche <> None alors
            F = enfiler(F, B->gauche)
        si B->droit <> None alors
            F = enfiler(F, B->droit)
```

On adapte cette procédure pour calculer la profondeur de cheminement externe de manière itérative :

```
LCEIt(A : arbreBinaire) : entier
    total : entier ; total = 0
    p : entier ; p = 0
    F : file ; F = initFile()
    F = enfiler(F, A) ; F = enfiler(F, p)
    tant que non fileVide(F) faire
        B = tete(F) ; F = defiler(F)
        p = tete(F) ; F = defiler(F)
        si B->gauche == None et B->droit == None alors
            total = total + p
        si B->gauche <> None alors
            F = enfiler(F, B->gauche)
            F = enfiler(F, p+1)
        si B->droit <> None alors
            F = enfiler(F, B->droit)
            F = enfiler(F, p+1)
    retourner total
```

Exercice 3. *On représente une expression arithmétique par un arbre binaire localement complet, où les nœuds internes portent la valeur* `+` *ou* `*` *et où les feuilles portent une lettre d'un alphabet* `A`*.*

Question 1. Représenter l'arbre de l'expression $(((a * c) + (d * b)) * (b + a))$.



Question 2. *Écrire une procédure récursive qui prend en argument un arbre binaire* `E` *représentant une expression arithmétique et une fonction de valuation* `V(a : A) : entier` *et qui retourne la valeur de l'expression arithmétique pour* `V`*.*

```
valeurExp(E : arbreBinaire, V : fonction) : entier
    si E->gauche == None et E->droit == None alors
        retourner V(E->valeur)
    si E->valeur == '+' alors
        retourner valeurExp(E->gauche, V) + valeurExp(E->droit, V)
    si E->valeur == '*' alors
        retourner valeurExp(E->gauche, V) * valeurExp(E->droit, V)
```

Question 3. *Écrire une procédure récursive, à partir d'une expression arithmétique lue au clavier, caractère par caractère, retourne l'arbre de l'expression.*

On indique juste en commentaire quel devrait être le contrôle d'erreur :

```
arbreExp() : arbreBinaire
    c : caractère ; c = lire(1) # lit un caractère
    A : arbreBinaire ; A = Nouveau(noeud)
    si c == '(' alors
        A->gauche = arbreExp()
        A->valeur = lire() # doit être '+' ou '*'
        A->droit = arbreExp()
        lire() # doit être ')'
    sinon
        A->valeur = c # ne doit pas être '+', '*', ')'
        A->gauche = None
        A->droit = None
    retourner A
```