Feuille de TD N^o 5 : Arbres

- Remarques préliminaires, cours :

  Les arbres binaires sont définis récursivement : feuille est un AB ; si B et C sont des AB, alors N(B,C) est un AB. Il n'y a donc pas d'arbres vides.

  On trouve des variantes dans la littérature dans lesquelles un arbre peut être vide, ou bien on a un nœud racine, avec ou sans arbre à gauche, avec ou sans arbre à droite. Il y a bijection entre les deux structures si vous considérez que les nœuds des arbres de la variante sont les nœuds internes de la définition que j'utilise. Arbre vide donne arbre feuille. Arbre réduit a un nœud donne N(feuille,feuille), dit "queue de cerise". Arbre avec une racine et un fils droit donne N(feuille,N(feuille,feuille))

  Un arbre peut être valué, pas nécessairement avec le même type aux nœuds internes et aux feuilles. Un arbre d'expression arithmétique est valué par des nombres aux feuilles et par des opérateurs aux nœuds internes. Un ABR, un TAS sont des arbres valués par des entiers (ou autres objets issus d'un ensemble E totalement ordonné) aux nœuds internes et par rien du tout aux feuilles. Un ABR vide est donc en fait un arbre réduit à une feuille.

  Les arbres quelconques sont définis récursivement : si n est un entier de N et A1 ... An des arbres, alors N(A1,...,An) est un arbre. On a le cas de base en prenant n égal à zéro.

  Une forêt est une suite finie (éventuellement vide) d'arbres.

  Nous travaillerons sauf indication contraire avec des arbres binaires. Les fonctions de base sont similaires à celles sur les listes, on peut :

  Créer des arbres feuilles. Si elles ne sont pas valuées : "Feuille()" i.e. NULL. Si les feuilles sont valuées, "Feuille(vf)", dans ce cas, il y a un malloc derrière.
  On peut initialiser A à feuille : InitFeuille(out * B), ou avec valuation si les feuilles sont valuées InitFeuille(out * B, in vf). Ou encore "A = Feuille()" (i.e. "A=NULL;") ou, si les feuilles sont valuées, "A = Feuille(vf)".

  On pourra Tester si A est une feuille : TestFeuille(A), ou "rendre/si" "A est une feuille". En pratique, si les feuilles ne sont pas valuées, "A==NULL", si elles le sont, "A->Feuille"

  Rendre la valuation VF(A) d'un arbre feuille si elles sont valuées. En pratique A->VF.
  VF(A) plante si A n'est pas une feuille.

  Rendre la valuation de la racine nœud interne VR(A) d'un arbre A s'il est valué aux nœuds internes. En pratique A->VR.
  VR(A) plante si A est une feuille.

  Accéder aux sous abres gauches gauche et droit : A->SAG et A->SAD. S'il faut les modifier, nous serons amené à utiliser des "pointeurSA" : & (*B)->SAG, & (*B)->SAD. Ces opérations plantent si A est une feuille.

  Construire un arbre A à partir de A_G (qui sera le SAG de A), de A_D (qui sera le SAD), et, si les nœuds internes sont valués, d'une valuation v à mettre à la racine : $N(A_G, A_D)$ ou $N(A_G, A_D, v)$.
  Il y a un malloc derrière cette fonction.

1 Fonctions et procédures sur les arbres binaires

Écrire les fonctions et procédures suivantes sur les arbres binaires :

1. **Hauteur**

   - int hauteur(A)
     si A feuille

alors rendre 0
sinon rendre max(hauteur(A->SAG),hauteur(A->SAD)) +1

2. Le miroir d'un arbre est celui que vous obtiendriez en le regardant à travers un miroir vertical. Les côtés gauche et droite s'inversent.

   - **Fmiroir** qui donne en résultat le miroir de son argument.
   - **Pmiroir** qui transforme son argument en son miroir.
   - **SontMiroirs** qui rend vrai ssi ses deux arguments sont miroirs l'un de l'autre.

   - Code ci-dessous avec VF pour feuilles valuées, VR pour nœuds internes valués. L'énoncé ne précise pas si les arbres sont valués; les deux codes, avec ou sans valuation, seront acceptés.

```
arbre Fmiroir (A)
si A feuille
alors rendre Feuille(A->VF)
sinon rendre Construit(Fmiroir(A->SAD), Fmiroir(A->SAG), A->VR)

void Pmiroir(inout *B)
si *B n'est pas une feuille
alors {
      tmp = (*B)->SAG
      (*B)->SAG = (*B)->SAD
      (*B)->SAD = tmp
      miroir(& (*B)->SAG)
      miroir(& (*B)->SAD) }
```

Remarque : ici, A (le pointeur arbre) est supposé être passé par adresse (Pmiroir(& A) et B est un pointeur vers A) mais en réalité, s'il est passé par valeur, ça marche. Pourquoi ? parce que ce qui est modifié, ce n'est pas le pointeur mais le contenu du struct pointé par A. Donc si j'ai une copie de A, la copie pointe vers le même struct et (changer le pointeur sur la copie n'aurait pas d'incidence sur l'original) mais (changer le contenu de la valeur pointée par le pointeur copie a le même effet que changer le contenu de la valeur pointée par l'original).

```
void Pmiroir(inout A)
si A n'est pas une feuille
alors {
      tmp = A->SAG
      A->SAG = A->SAD
      A->SAD = tmp
      miroir(A->SAG)
      miroir(A->SAD) }

bool SontMiroirs(A1, A2)
      Si A1 est feuille
      alors rendre (A2 est feuille) et (A1->VF == A2->VF)
sinon
      si A2 est feuille // et A1 non feuille
      alors rendre faux
sinon rendre
         A1->VR == A2->VR
      et SontMiroirs(A1->SAG, A2->SAD)
```

et SontMiroirs(A1->SAD, A2->SAG)

Autres façons de gérer les cas de base :

si A1 feuille et A2 feuille alors ... sinon si A1 feuille et A2 non feuille alors ... sinon si A1 non feuille et A2 feuille alors ... sinon récursif
Correct mais LOURD, en écriture mais aussi en exécution (test A1 feuille fait trois fois)

Si A1 feuille et A2 feuille alors ... sinon si A1 feuille ou A2 feuille alors ... sinon récursif.
Ou bien:
Si A1 feuille ou A2 feuille alors si A1 feuille et A2 feuille alors ... sinon ... sinon récursif.
Moins lourd que le précédent mais lourd tout de même (test A1 feuille doublé)

Lourd, ce n'est pas top, mais mieux vaut être lourd (très nettement moins de points perdus) que faux en ne gérant pas un cas (plantage) ou en le gérant mal (mauvaise réponse)

3. (facultatif) Les **affichages préfixe, postfixe et infixe** en itératif.

- Dérécursifiaction : on simule le compilateur qui utilise des piles de numéros de lignes de code et des piles de variables (cf. poly sur la récursivité). Plus humainement, on conserve une pile de ce qu'il reste à finir, quoi (pile de lignes de code) sur qui (pile de variables), puis on simplifie si possible : pas la peine de mémoriser "quoi" si c'est toujours la même chose; s'il n'y a que deux possibilités, un drapeau booléen suffit; parfois, on arrive à se passer du "quoi" dans la pile parce que le "quoi" alterne (pile truc trac truc trac truc trac truc). De même, pas besoin de stocker un "qui" si c'est toujours le même (Le tableau dans la dérécursification de Quicksort)

Préfixe : pas besoin de stocker le "quoi", le "qui" suffit, d'où gestion par une pile d'arbres.

Postfixe : le qui ne suffira pas, il faudra aussi empiler le quoi, avc deux possibilités : Le quoi sera "imprimer tout l'arbre" ou "n'imprimer que VR de l'arbre". Un drapeau booléen suffira pour gérer l'alternative.

Infixe, on va empiler a priori le quoi, il y a des astuces pour n'empiler que le qui.

```
void AffichagePrefixe (A)
pile = vide
empiler A
tant que pile non vide {
      sortir X de la pile // ie X = premier(pile) puis dépile(pile)
      si X est une feuille
         alors afficher X->VF
      sinon {  afficher X->VR // cette ligne peut être placée après les empilements
               empiler X->SAD // ATTENTION, empiler le SAD d'abord,
               empiler X->SAG // car le premier empilé = le dernier sorti donc effectué
            }          }
```

Notez que les deux codes qui suivent ne différent l'un de l'autre que par la position de la ligne empiler(X, racine) par rapport aux deux lignes empile(SA(X),arbre). Si vous la mettez après, vous obtiendrez une variante lourde de AffichePrefixe (la version de AfficheInfixe ci-dessus s'obtient en traitant l'affichage de la racine directement sans passer par la pile puis en éliminant le z qui est devenu inutile)

```
void AffichageSuffixe (A)
pile = vide
empiler (A, arbre)
tant que pile non vide  {
      sortir (X,z) de la pile
         si X est une feuille
            alors afficher X->VF
      sinon // la racine est un nœud interne
         si z == racine
            alors afficher X->VR
      sinon { // z == arbre
            empiler ( X,       racine)
            empiler (X->SAD, arbre)
            empiler (X->SAG, arbre)
            }           }
```

```
void AffichageInfixe (A)
pile = vide
empiler (A, arbre)
tant que pile non vide  {
      sortir (X,z) de la pile
         si X est une feuille
            alors afficher X->VF
      sinon // la racine est un nœud interne
         si z == racine
            alors afficher X->VR
      sinon { // z==arbre
            empiler (X->SAD, arbre)
            empiler ( X,       racine)
            empiler (X->SAG, arbre)
            }           }
```

Pour l'infixe :

Astuce 1. On observe que dans la pile ci-dessus, "z==arbre" et "z==racine" alternent, avec un "z==arbre" en fond de pile. On observe aussi qu'un "z==racine" est traité immédiatement sans rien remettre sur la pile.

Du coup, on peut maintenir une pile avec "z==arbre" au sommet et au fond. Quand on dépile, c'est donc toujours (B,z) avec "z==arbre", puis : Soit B est traité sans rempiler et on va dépiler dans la foulée un "z==racine" que l'on traite aussi, et on retombe sur un "z==arbre" en sommet de pile (il faudra juste traiter le cas où B était le fond de pile). Soit on empilera un "z==arbre" puis un "z==racine" puis un "z==racine. Et du coup, il n'est plus nécessaire de stocker z

```
void AffichageInfixe (A)
pile = vide
empiler A
tant que pile non vide  {
      sortir X de la pile
      si X est une feuille
      alors {  afficher X->VF
               si pile non vide
```

```
            alors { sortir X2 de la pile
                    afficher X2->VR
        }           }
    sinon {   empiler (X->SAD)
              empiler (X)
              empiler (X->SAG)
        }             }
```

Astuce 2 : on maintient que quand un arbre B est dans la pile, B n'est pas feuille et il faut afficher sa racine puis l'intégralité de son sous arbre droit. Les feuilles sont gérées directement sans passer par la pile.

```
void AffichageInfixe (A)
pile = vide
P = A
tant que P n'est pas feuille
      {  empiler P
         P = P->SAG }
afficher P->VF
tant que pile non vide
      {  sortir X de la pile
         afficher X->VR
         P = X->SAD
         tant que P n'est pas feuille
               {  empiler P
                  P = P->SAG }
         afficher P->VF }
```

4. À partir d'un arbre binaire A, deux joueurs jouent au jeu J1 suivant: Un pion est initialement sur la racine. Alternativement, chaque joueur déplace le pion de là où il se trouve vers l'un de ses deux fils, au choix de celui qui joue. Celui qui doit jouer alors qu'il est sur une feuille a gagné (donc celui qui joue en amenant le pion sur une feuille perd). Un arbre est J1-gagnant si celui qui commence a une stratégie gagnante, il est J1-perdant s'il n'en a pas, i.e. si c'est celui qui ne commence pas qui a une stratégie gagnante.

   Dans le jeu J2, celui qui doit jouer alors qu'il est sur une feuille a perdu (donc celui qui joue en amenant le pion sur une feuille gagne). Est-ce que A est J1-gagnant ssi il est J2-perdant ?

   Écrire la fonction **J1gagnant** qui prend A en argument et rend vrai ssi A est J1-gagnant.

   - Non, N(N(F,F),F) est a la fois J1 et J2 gagnant

   Cela semble difficile mais c'est tout con ... J'ai la garantie de gagner si je peux jouer un coup qui amène sur une situation dans laquelle mon adversaire n'a pas la garantie de gagner, i.e. qu'il va perdre si je joue à la perfection, i.e. si je peux mettre l'adversaire dans une situation non J1-gagnante.

```
bool J1G(A)
Si A est une feuille
alors rendre vrai
sinon rendre (non J1G(A->SAG) ou (non J1G(A->SAD)

bool J2G(A)
Si A est une feuille
```

alors rendre faux
sinon rendre (non J2G(A->SAG) ou (non J2G(A->SAD)

Remarque on peut vouloir se ramener aux 4 appels sur SA(SA()), mais cela va être lourd car il faut traiter à part les cas où au moins un des fils est feuille. Si aucun des fils n'est une feuille, que fait-on des 4 appels ? Un ET global ? Un OU global ? Je vous laisse réfléchir. Cf le code ci-dessous.

```
bool J1G(A)
Si A est une feuille
alors rendre vrai
sinon
si A->SAG est une feuille
alors rendre NOT(A->SAD est une feuille)
            et (J1G(A->SAD->SAG) et J1G(A->SAD->SAD))
sinon
si A->SAD est une feuille
alors rendre (J1G(A->SAG->SAG) et J1G(A->SAG->SAD))
sinon rendre
              (J1G(A->SAD->SAG) et J1G(A->SAD->SAD))
            ou (J1G(A->SAG->SAG) et J1G(A->SAG->SAD))

bool J2G(A)
Si A est une feuille
alors rendre faux
sinon
si A->SAG est une feuille ou A->SAD est une feuille
alors rendre vrai
sinon rendre
              (J2G(A->SAD->SAG) et J2G(A->SAD->SAD))
            ou (J2G(A->SAG->SAG) et J2G(A->SAG->SAD))
```

5. **CompteSansAsc** qui prend A en argument et rend le nombre de nœuds internes valués par vrai mais dont aucun ascendant strict n'est valué par vrai.
   Exemple : 2 pour l'arbre ci-contre.

   - On utilise une sous-fonction avec un paramètre qui dit s'il y a un ancêtre valué à vrai. Cette information est descendante, elle est initialisée à la racine et se propage vers le BAS en direction des feuilles. Ce sera donc un paramètre "in". Notez qu'il n'y a pas besoin d'une variable pour gérer ce paramètre au niveau de la surfonction.

```
int CSA(A)
        rendre B(A,faux)

int B(A,b) int // b vrai ssi il y a un ancêtre vrai
si A est une feuille
alors rendre 0
sinon si VRA(A)
      alors si b
            alors rendre 0 + B(A->SAG, vrai) + B(A->SAD, vrai)
            sinon rendre 1 + B(A->SAG, vrai) + B(A->SAD, vrai)
      sinon rendre 0 + B(A->SAG, b) + B(A->SAD, b)
```

Mais on peut se rendre compte que quand le booléen est vrai, la fonction B rend toujours 0, ce qui est logique, si j'ai un ancêtre vrai, tous mes descendants aussi et personne ne sera compté. On peut le justifier par récurrence ascendante (B(feuille,vrai) rend 0, et si A n'est pas une feuille : si B(A->SAG, vrai) et B(A->SAD, vrai) rendent 0 alors B(A,vrai) aussi. Donc on peut enlever tous les appels avec vrai :

```
int CSA(A)
        rendre B(A,faux)

int B(A,b) int // b vrai ssi il y a un ancêtre vrai
si A est une feuille
alors rendre 0
sinon si VRA(A)
      alors si b
            alors rendre 0
            sinon rendre 1
      sinon rendre 0 + B(A->SAG, b) + B(A->SAD, b)
```

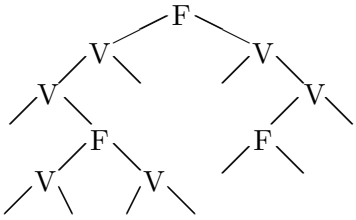
Mais du coup la fonction B est toujours appelée avec faux (récurrence descendante : c'est vrai à la racine, et si c'est vrai pour A non feuille, c'est aussi vrai pour ses fils), et le booléen ne sert plus à rien et la sous fonction non plus.

Ou plus directement, on descend dans l'arbre et quand on rencontre un nœud vrai, on le compte et on ne descend pas plus bas.

```
int CSA(A) : int
Si A est feuille alors rendre 0
sinon si val(A) alors rendre 1
      sinon rendre CSA(A->SAG)+CSA(A->SAD)
```

L'astuce qui donne le code final est une particularité de cette fonction. Dans beaucoup d'exemples, il faudra faire une sous-fonction avec des paramètres que l'on descend, et il n'y aura pas de simplification. Il est donc plus important dans cette exercice de comprendre la première solution que la seconde.

6. **CompteSansDesc** qui prend A en argument et rend le nombre de nœuds internes valués par vrai mais dont aucun descendant strict n'est valué par vrai. Exemple : 3 pour l'arbre ci-contre.



- On utilise une sous-fonction avec un paramètre qui dit s'il y a un descendant valué à vrai. Cette information est ascendante, elle est initialisée aux feuilles et se propage vers le HAUT en direction de la racine. Ce sera donc un paramètre "out".

  Au niveau de la surfonction, il faudra déclarer une variable et appeler la sous-fonction avec cette variable, même si, in fine, cette variable n'est pas utilisée au niveau de la surfonction. Cela reste inévitable car la sous-fonction aura un argument out et attend donc une variable soit donc plus qu'une simple expression.

  Cette sous-fonction va avoir un paramètre out,... m'ouais, ce n'est pas top propre d'avoir des fonctions avec des arguments autres que in. Du coup, nous allons basculer sur une sous-procédure, et le résultat de la fonction va être transformé en argument, et puisqu'il remonte, ce sera un deuxième argument out. Il faudra donc une deuxième déclaration de variable au niveau de la surfonction, qui, elle, sera utile puisque c'est elle qui sera renvoyée par la surfonction.

  Ces arguments sont "out", donc il va falloir les "remonter". Il est tentant de dire les "rendre", c'est cela et ce n'est pas cela. C'est celà moralement, remonter un résultat c'est

en quelque sorte le rendre à son appelant. Dans une fonction, la valeur rendue est un argument out implicite. De même que dans une fonction, votre mission est de rendre un résultat, dans une procédure, votre mission est de remonter ce résultat.

```
int CSD(A)
        bool b, int r ;
        B(A, & b, & r)
        rendre r

void B(A, out *B, out *R)
Si A est feuille
alors { *B = faux ; *R = 0 }
sinon { bool bg, bd, int rg, rd ;
        B(A->SAG, & bg, & rg )
        B(A->SAD, & bd, & rd )
        *B = bg ou bd
        *R = rg + rd
        si A->VR et non *B
        alors { (*R)++ ; *B = vrai }
        }
```

La mission de B est de remonter b et r, ce que l'on doit toujours faire.

Donc dans le cas feuille, on leur donne une valeur. Cela doit devenir un réflexe, var out, donc si feuille, la var reçoit une valeur initiale. De même que si vous avez une fonction, le réflexe est si feuille (vide pour des listes), alors que rends-je ?

Si ce n'est pas une feuille, il faut faire remonter la valeur. Pour une fonction, si A n'est pas feuille, alors que rends-je ?

D'une manière générale, il y a plusieurs manières de faire remonter la valeur :

(0) initialiser la valeur directement, v=42, tout comme on fera rendre 42 pour une fonction. Classique quand on est sur liste vide ou arbre feuille, mais il peut y avoir d'autres cas.

(1) Faire initialiser la valeur par un sous-appel, CallRec(..., out v), tout comme une fonction fera un appel récursif (potentiellement terminal).

(2) Récupérer la valeur renvoyée par l'appel récursif et la modifier : Rec(...,v); v++; tout comme une fonction fera rendre Rec(...)+1.

(3) Récupérer les valeurs renvoyées par plusieurs appels récursifs et en faire une synthèse que l'on remonte, Rec(SAG,vg); Rec(SAD,vd); vv=vg+vd+23;, tout comme une fonction pourra faire rendre Rec(SAG,...) + Rec(SAD,...) + 23.

On observe ici une différence clef entre ce que l'on fait souvent pour des listes et ce que l'on fait souvent pour les arbres (souvent ne veut pas dire toujours, il y a forcément des exceptions, mais vous ne comprendrez l'exception que quand vous aurez compris la règle)

Pour des listes, il est classique d'envoyer la variable out directement à l'appel récursif.

Situation 1 : L'appel initialisera la variable et on la renverra telle que. Pour une fonction : appel récursif terminal (rendre Rec(...)), pour une procédure, appel terminal avec v en argument (Rec(...,v)). Cf. cas (1) ci-dessus.

Situation 2 : L'appel initialisera la variable et on la modifiera. Pour une fonction, on fera par exemple rendre Rec(...)+1. Pour une procédure, on appelle la procédure récursive, puis on modifie : Rec(...,v); v++; Cas (2) ci-dessus

Pour des arbres, il y aura classiquement deux appels, donc deux valeurs qui remontent des fils. Nous ne pouvons plus demander aux deux appels d'initialiser notre variable, le deuxième appel ne pourrait qu'écraser ce qu'a fait le premier. Nous allons donc introduire, i.e. déclarer, deux variables, une pour chaque appel, faire les appels récursifs avec l'une

puis l'autre variable, et en déduire ce que vaudra notre variable out que nous allons faire remonter. Dans l'exemple ci-dessus, on déclare bg bd rg rd, on demande à l'appel gauche de remonter bg et rg, itou côté droit avec bd et rd, et on analyse les remontées des fils pour calculer le b et le r que l'on va remonter au père.

Bref, variable out donc ... pour une liste, possiblement appel récursif avec la même variable. Pour un arbre, c'est plutôt deux appels avec deux variables neuves et on effectue un calcul après pour savoir ce que l'on remonte soi-même.

Variante : Au lieu de gérer r avec un argument out, i.e. au lieu de récupèrer les infos des fils et d'agglomérer (d'additionner dans la plupart des cas) : On gère r avec un argument inout : on ne remonte pas r, on le promène le long de l'arbre et on l'incrémente quand il faut. Dans ce cas, r est inout, il n'y a qu'une variable pour tout l'arbre. Elle est classiquement initialisée à la racine dans la surfonction, puis est propagée dans l'arbre via une variable inout. Il n'y aura alors qu'une seule variable "globale" pour tout le monde, et donc l'appel récursif se fera en envoyant cette variable inout (par opposition aux deux variables nouvelles que l'on envoie pour une variable out)

```
int CSD(A)
        bool b, int r=0 ;
        B(A,& b,& r)
        rendre r

void B(A,out *B, inout *R)
Si A est feuille
alors *B = faux
sinon { bool bg, bd
        B(A->SAG, & bg, *R )
        B(A->SAD, & bd, *R )
        *B = bg ou bd
        si A->VR et non *B
        alors { (*R)++ ; *B = vrai }
        }
```

Puis (on démarre les acrobaties...) on peut se rendre compte (avec r en out) que b est inutile, en effet le boolén remonte à vrai ssi rg+rd est non nul, du coup, on peut se contenter de la variable r en out, et du coup, on peut tout faire en une fonction :

```
int CSD(A)
Si A est feuille
alors rendre 0
sinon
        si CSD(A->SAG)+CSD(A->SAD) == 0
        alors si A->VR
                alors rendre 1
                sinon rendre 0
        sinon rendre CSD(A->SAG)+CSD(A->SAD)
```

mais HORREUR !, on lance deux fois sur les fils, donc quatre sur les petits-fils, 2^p fois sur les p-descendants.

On ne fait pas ça, on appelle une seule fois et on sauve le résultat :

```
int CSD(A) : int
Si A est une feuille alors rendre 0
sinon
```

cs = CSD(A->SAG) + CSD(A->SAD)
si cs > 0 alors rendre cs
sinon
si A->VR alors rendre 1 sinon rendre 0

Cette dernière version est une jonglerie.

7. **CompteHEgalP** qui rend le nombre de nœuds internes de son argument dont la hauteur est égale à la profondeur.

   - Première solution un peu naïve, mais correcte :
     int CompteHEgalP(A)
     rendre CHP(A,0) // deuxième argument : profondeur courante

     int CHP(A,p)
     si A est une feuille
     alors rendre 0 // y compris si A-main feuille : l'énoncé dit comptez les nœuds internes.
     sinon
     si hauteur(A) == p
     alors rendre 1 // pas si naïf non plus : appels rec inutiles, ils rendraient 0
     sinon rendre CHP(A->SAG, p+1) + CHP(A->SAD, p+1)

     Non linéaire (quadratique sur un filiforme, $n \ln n$ sur un complet) parce que l'on appelle la fonction h profondeur(x) fois sur le nœud x, et pour calculer la même chose !

     Du coup, on va plutôt remonter la hauteur avec une variable out, en parallèle du comptage.

     Linéaire. Bien voir le jeu sur les informations ascendantes (h) et descendantes (p). Arguments in VS out, initialisation à la racine VS aux feuilles, calcul fait avant VS après l'appel récursif.

     Pour cpt, on peut le faire en inout ou bien en out. Nous le faisons en inout de façon à montrer comment on manipule chaque catégorie de variable : in, out, inout.

     Le protocole de pensée est "je vais devoir faire une sous-procédure transportant les informations nécessaires". Puis quels sont ces arguments (ici, p, h, cpt) et sont-ils descendants ou ascendants et dont de quelle catégorie (in, out, inout) sont-ils ?

     void Bis (in A, in p, out *H, inout * CPT)
     si A est feuille alors *H = 0
     sinon
         int hg, hd
         Bis(A->SAG, p+1, & hg, CPT)
         Bis(A->SAD, p+1, & hd, CPT)
         *H = 1 + max(hg,hd)
         si *H == p alors (*CPT)++

     int CompteHP (A)
     int cpt = 0
     Bis(A, 0, h, & cpt)
     rendre cpt

     Avec le comptage en out :

     int CompteHegalP(A) : int
     int h, n
     CHP(A, 0, & h, & n)

rendre n

void CHP(in A, in p, out *H, out *N)
Si A est une feuille
alors
    *H = 0
    *N = 0
sinon
    int hg, hd, ng, nd
    CHP(A->SAG, p+1, & hg, & ng)
    CHP(A->SAD, p+1, & hd, & nd)
    *H = 1+max(hg,hd)
    si *H == p
    alors *N = 1 // 1+ng+nd OK mais en fait ng==nd==0
    sinon *N = ng+nd

8. **EstComplet** qui prend A en argument et rend vrai ssi il est complet (ie ssi toutes les feuilles sont à la même profondeur)

- là encore, première solution non linéaire, puis pleins de solutions linéaires
  bool EC(A)
  si A feuille
  alors vrai
  sinon EC(A->SAG) et EC(A->SAD et h(A->SAG) == h(A->SAD)

  pas linéaire pour la raison habituelle
  Je recommence :

  void EC_TMP(A, out *B, out *H)
  si A feuille
  alors { *B = vrai, *H = 0 }
  sinon bool bg, bd, int hg, hd
      EC_TMP(A->SAG, & bg, & hg)
      EC_TMP(A->SAD, & bd, & hd)
      *B = bg et bd et (hg == hd)
      *H = max(hg,hd)+1

  bool EC(A)
  bool b, int h
  EC_TMP (A, & b, & h)
  rendre b

  variante possible : si pas complet, h ne sert a rien, donc je fais une fonction qui rend h si complet, -1 sinon
  int EC_TMP(A)
  si A feuille
  alors rendre 0
  sinon hg = EC_TMP(A->SAG)
      hd = EC_TMP(A->SAD)
      si hg == hd et hg $\neq -1$
      alors rendre hg+1
      sinon rendre -1

bool EC(A)
rendre EC_TMP (A) ≠ -1

AUTRE IDÉE :
en utilisant une fonction qui teste si A est complet de profondeur p
bool ECp (A,p)
si A feuille
alors rendre (p == 0)
sinon
si p == 0
alors rendre faux
sinon rendre ECp(A->SAG, p-1) et ECp(A->SAD, p-1)

bool EC (A)
rendre ECp (A, profondeur(A))

ou mieux, on remplace prof(A) par ProfGauche(A) (profondeur de la feuille la plus a gauche)
int ProfGauche (A)
si A feuille alors rendre 0
sinon rendre 1+ProfGauche(A->SAG)

Note : Si dans ECp, on ne fait pas le test "p==0", on risque d'explorer l'arbre au delà du nécessaire. Pas faux, on finira par un p négatif donc le test "p==0" rendra faux avec raison. Mais on peut court-circuiter et arrêter plus tôt.
Cela dit, si on utilise profondeur dans la surfonction, tester p==0 devient inutile car la profondeur ne dépassera jamais p. Par contre, ce test redevient utile si l'on utilise ProfGauche
VARIANTE
on parcourt une fois, on mémorise si on a déjà vu une feuille et si oui quelle est sa profondeur

void ECslave (in A, in p, inout * DEJAVU, inout * GOODP, inout * OK)
si A est une feuille
alors
    si *DEJAVU
    alors { si p ≠ *GOODP alors *OK = faux }
    sinon { *DEJAVU = vrai ; *GOODP = p }
sinon
    ECslave (A->SAG, p+1, DEJAVU, GOODP, OK)
    si *OK alors ECslave (A->SAD, p+1, DEJAVU, GOODP, OK)

bool EC (A)
bool dejavu, int goodp, bool ok
ECslave(A, 0, & dv, & gp, & ok)
rendre ok

AUTRES IDÉES

IDÉE 3

Solution linéaire non triviale : reprendre la version non linéaire et remplacer prof par profG. Si on fait profD(SAG) + profG(SAD), on s'aperçoit que tout nœud subit au plus un appel a profG ou profD, donc c'est lin. Si on fait profG(SAG) + profG(SAD) et qu'on le fait

APRÈS les tests récursifs, alors (”et” paresseux), on ne le fait que sur des complets, et profG(SAG) a donc le même coût que si faisait profGD(SAG) et donc c'est linéaire. Ce n'est pas linéaire si on fait profG(SAG) + profG(SAD) avant les appels récursifs (prendre un filiforme gauche)

IDÉE 4

bool EC(A)
si A feuille
alors vrai
sinon EC(A->SAG) et SontEgaux(A->SAG,A->SAD)

on obtient que c'est linéaire en disant que complexité de SontEgaux(A->SAG,A->SAD) est theta(taille(A->SAD))

IDÉE 5

EC(A)
rendre (NF(A) == $2^{profondeur(A)}$)

Cette dernier solution a un problème : si on a un filiforme de prof 1000, on se retrouve a faire 2^{1000} qui explose le maxint, on pourra faire la variante :
EC(A)
rendre Test (NF(A), profondeur(A))

Test (x,y) : bool
si y == 0 alors rendre x == 1
sinon
si x est impair alors rendre faux
sinon
rendre Test(x/2, y-1)

IDÉE 6

Faire un parcours en largeur, on doit ne voir que les nœuds internes puis que les feuilles. Il faut ajouter une astuce pour vérifier que les feuilles commencent au début d'un étage, par exemple, on compte les nœuds apparus, et la première feuille doit être la k^e avec k une puissance de 2, ou bien on injecte dans la file un symbole spécial qui marque la fin de ligne.

IDÉE 7

hmax(A)
si A feuille
alors 0
sinon max(hmax(A->SAG), hmax(A->SAD)) + 1

hmin(A)
si A feuille
alors 0
sinon min(hmin(A->SAG), hmin(A->SAD)) + 1

EC(A)
rendre (hmax(A) == hmin(A))

9. **EnumereArbres** qui prend en argument un entier n et rend la liste des arbres ayant n nœuds internes.

- EA(n)
T[0] = liste singleton contenant l'arbre feuille
pour i de 1 a n :
T[i] = liste vide
pour k de 0 i-1
    x = T[k]
    tant que x pas vide
        y = T[i-k-1]
        tant que y pas vide
            empiler N(premier(x), premier(y)) sur T[i]
            y = y->suite
        x = x->suite
rendre T[n]

Moins bien (répétition de calculs) :
EA(n)
Si n == 1 alors rendre ajoute(Feuille,Vide) // liste singleton contenant arbre feuille
sinon
    R = vide
    pour i de 0 a n-1, R = Concatene(ToutesPaires(EA(k),EA(n-k-1)),R)
    rendre R

Concatene (L1,L2) // fait en cours
si L1 est vide alors rendre L2
sinon rendre ajoute(premier(L1), concatene(suite(L1),L2))

ToutesPaires(L1,L2)
si L1 est vide alors rendre vide
sinon rendre concatene(ConsAetBdansListe(premier(L1),L2) ,ToutesPaires(suite(L1),L2)))

ConsAetBdansListe(A,L)
Si L est vide alors rendre vide
sinon rendre ajoute(Arbre(A,premier(L)), ConsAetBdansListe(A,suite(L))

2

Réécrire l'expression préfixe / ∗ + 1 2 3 − + ∗ 4 5 / − 6 7 8 9 en notation postfixe.

- Arbre
N(/ N(* N(+ F(1) F(2)) 3) N(- N(+ N(* F(4) F(5))) N(/ N(- F(6) F(7)) F(8)) 9))
1 2 + 3 * 4 5 * 6 7 - 8 / + 9 - /

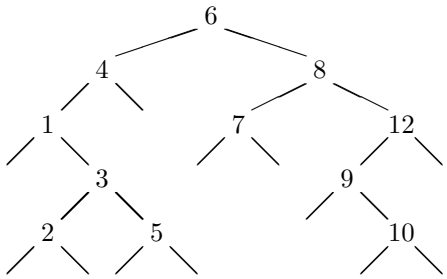
Réécrire l'expression postfixe 1 2 3 ∗ − 4 + 5 6 / 7 + 8 9 ∗ − / en notation préfixe.

- Arbre

N(/ N(+ N(- F(1) N(* F(2) F(3))) F(4)) N(- N(+ N(/ F(5) F(6)) F(7)) N(* F(8) F(9))))
/ + - 1 * 2 3 4 ’ + / 5 6 7 * 8 9

3 ABR

3.1 Généralités



Un ABR (Arbre Binaire de Recherche) est un arbre binaire dans lequel les nœuds internes sont valués par des entiers et tel que pour tous nœuds internes x et y, on a que si x est descendant gauche, respectivement droit de y alors on a $val(x) \leq val(y)$, resp. $val(y) \leq val(x)$.

1. L'arbre ci-contre est-il un ABR ?

   - Non, cf 4 et 5

2. Écrire la fonction `recherche` qui dit si un élément figure dans un ABR et la procédure `ajout` qui insère un nouvel élément dans un ABR au niveau d'une feuille.

   - bool EstDans (x, A)
     si A est une feuille
     alors rendre faux
     sinon
     si A->VR == x
     alors rendre vrai
     sinon
     si A->VR < x
     alors rendre EstDans(x,A->SAD)
     sinon rendre EstDans(x,A->SAG)

   - void ajout (x, inout * B)
     si *B est une feuille
     alors *B = AB(x,Feuille,Feuille)
     sinon
     si (*B)->VR < x
     alors ajout(x,& (*B)->SAD)
     sinon ajout(x,& (*B)->SAG)

     Note : pour EstDans, on pourrait faire "rendre ED(SAG) ou ED(SAD)", pas faux, mais ne profite pas de la structure en ABR et beaucoup plus cher en complexité. Faire cela ne vaut pas grand chose en examen.

3. Décrire la procédure `TriViaABR` qui prend en argument une liste non triée et la trie. Quelle est la complexité (en nombre de comparaisons) au pire de ce tri ? Quelle complexité obtient-on si les ABR restent de profondeur $\theta(profondeur\,minimale\,possible)$?

   - ABR reçoit vide, ajouter un a un tous les éléments de E, faire un parcours infixe. C'est en n^2 au pire (si l'ABR dégénère vers un filiforme) et en $n \ln n$ si l'arbre reste de profondeur $\theta \ln |E|$

4. Décrire la procédure `Supprime` qui cherche un élément et le supprime. Discuter selon la situation.

   - si le nœud contenant x a des feuilles pour fils alors le remplacer par une feuille.
     sinon
     si x a une feuille pour FG, alors mettre le SAD de x a la place de x
     sinon
     (on peut faire le symétrique s'il n'a pas de FD)
     mettre, à la place de x, le max du SAG de x, i.e. le y qui est tout au bout à droite du SAG de x.
     Le y laisse un trou mais son FD est une feuille, donc il suffit de mettre le SAG(y) à la place de y.
     On peut faire la même manip avec le min du SAD de x.

5. Donnez un ABR contenant tous les nombres de 1 à 11 tel que (7 est à la racine) et (1 et 5 sont frères) et (8 et 11 sont frères) et (6 est père de 4)

   - Le prof corrige les copies en faisant un parcours infixe de votre arbre, ce qui permet de repérer les bugs: ce n'est pas un ABR (un descendant est du mauvais côté), il manque un élément, un élément est en double. Suggestion: faites de même avant de rendre votre copie !

3.2 Rotations et recherche adaptative

On appelle rotation gauche autour du nœud n la transformation qui fait passer de l'arbre ci-dessous à gauche à celui à droite. Une rotation droite est l'opération symétrique.

1. Si A est un arbre binaire de recherche, et que l'on effectue une rotation autour de l'un de ses nœud, le résultat est-il toujours un arbre binaire de recherche ?

   - oui

2. Effectuez une rotation gauche autour de 6 dans l'arbre ci-dessus.

3. Écrire la procédure `rotation` qui prend en argument un arbre, un coté (`type cote = [gauche, droite]`) et effectue dans l'arbre la rotation de sens `cote` autour de la racine, quand cela a un sens.

   - void RotationDroite(inout *B)
     si *B est feuille ou (*B)->SAG est feuille
     alors ne rien faire (rotation impossible)
     sinon
     tmp = (*B)->SAG
     (*B)->SAG = tmp->SAD
     (*B)->SAD = *B
     *B = tmp

     variante :
     ...
     tmp = *B
     *B = (*B)->SAG
     tmp->SAG = (*B)->SAD
     (*B)->SAD = tmp

     variante :
     ...
     tmp = (*B)->SAG->SAD
     (*B)->SAG->SAD = *B
     *B = (*B)->SAG
     (*B)->SAD = tmp // (*B)->SAD était *B une ligne plus tôt

4. On part du principe que ce sont souvent les mêmes éléments qui sont recherchés, et qu'il est donc intéressant de rapprocher les éléments recherchés de la racine. Aussi, à chaque fois que l'on effectue la recherche d'un élément `e`, on effectue des rotations autour des ascendants de `e` pour remonter `e` à la racine. Écrire la procédure de recherche adaptative.

   - void Cherche(x, inout *B, out * trouve)
     si *B feuille
     alors *trouve = faux
     sinon
     si (*B)->VR == x
     alors *trouve = vrai
     sinon
     si (*B)->VR < x
     alors Cherche(x, & (*B)->SAD, trouve)
     si *trouve alors RotationGauche(B)
     sinon Cherche(x, & (*B)->SAD, trouve)
     si *trouve alors RotationDroite(B)

3.3 Arbres bicolores

- cf Cormen, Leiserson, Rivest, Stein, deuxième édition, section 13 page 273, red-black trees

Un arbre bicolore est un ABR dans lequel les nœuds internes sont coloriés en noir ou en rouge de sorte que:
• la racine est noire.
• si x est le père de y, ils ne sont pas rouges tous les deux.
• Le nombre de nœuds noirs rencontrés le long du chemin de la racine à une feuille est le même pour toutes les feuilles. On appelle hauteur noire de l'arbre ce nombre.

1. Majorez la profondeur d'un arbre bicolore en fonction du nombre de nœuds internes.
2. On insère aux feuilles un nouvel élément. Quels problèmes a-t-on si on décide de le colorier en rouge, resp. en noir ? Opter pour "un moindre mal".
3. On a un arbre bicolore, à ceci près qu'il y a un nœud x qui est rouge, de même que son père y. On veut éliminer le problème, quitte à ce qu'un nouveau problème apparaisse plus haut. Expliquez comment procéder (discuter en fonction de l'existence et de la couleur du frère de y)
4. Décrire un algorithme d'insertion dans un arbre bicolore.
5. Quelles sont les complexités de recherche, d'insertion dans un arbre bicolore, et quelle est la complexité au pire de `TriArbreBicolore` ?