Exercice 1

Appliquer l'algorithme de *tri topologique* basé sur l'algorithme `PP` au graphe dessiné dans la Figure 1 en suivant l'ordre lexicographique:

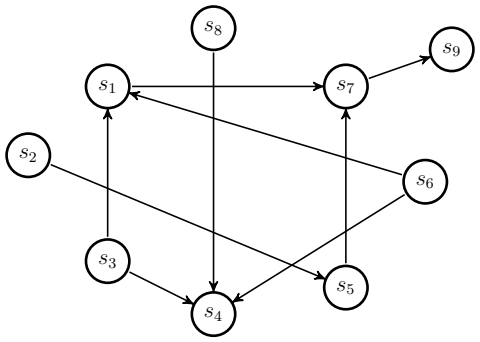

Figure 1: Le graphe G_1

Rappel:

```
0   PP(G)
1     pour chaque sommet u de V faire
2         couleur[u] <- BLANC
3         pere[u] <- nil
4     temps <- 0
5     pour chaque sommet u de V faire
6         si couleur[u] = BLANC
7             alors Visiter_PP(u)
```

```
0   Visiter_PP(u)
1     couleur[u] <- GRIS
2     d[u] <- temps <- temps + 1
3     pour chaque v de Adj[u] faire
4         si couleur[v] = BLANC
5             alors pere[v] <- u
6                   Visiter_PP(v)
7     couleur[u] <- NOIR
8     f[u] <- temps <- temps + 1
```

Exercice 2

On considère l'algorithme suivant:

```
1 pour chaque sommet u de V[G] faire:
2     deg_entrant[u] <- calculer_degre_entrant(u)
3 tant qu'il existe un sommet u tel que deg_entrant[u] == 0 faire:
4     afficher(u)
5     deg_entrant[u] <- deg_entrant[u] - 1
6     pour chaque v de Adj[u] faire:
7         deg_entrant[v] <- deg_entrant[v] - 1
```

i. Démontrer que tout graphe orienté fini, non vide, sans circuit possède un sommet sans prédécesseur.

ii. Que fait cet algorithme lorsqu'on l'applique à un graphe orienté sans circuit?

iii. Comment l'utiliser pour détecter un circuit?

iv. Quelle est sa complexité suivant que l'on utilise, pour représenter le graphe, une matrice d'adjacence ou des listes de successeurs?

Exercice 3

Modifier l'algorithme `PP` pour calculer la longueur du plus long chemin dans un graphe orienté sans circuit.

Exercice 4

(*) On définit un arbre `A` par une structure de donnée munie de deux fonctions `racine(A)`, et `fils(A)` qui renvoie, pour l'un, un sommet apellée la racine de A et pour l'autre la liste des fils de A qui sont des arbres.

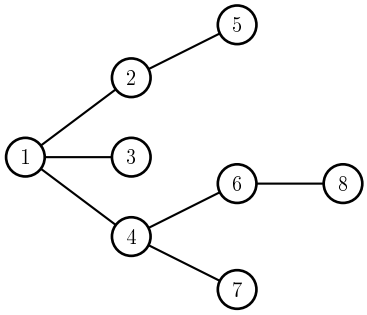

Figure 2: Arbre A_1

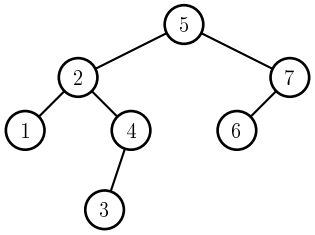

Figure 3: Arbre B_1

i. Modifier les algorithmes `PP` et `PL` de façon à les adapter aux arbres. Les algorithmes devront remplir un tableau `p[s]` donnant l'ordre de visite du sommet `s` dans le parcours. Pour `PL`, ils devront aussi remplir un tableau `d[s]` donnant la distance à la racine de l'arbre A.

Lors du parcours en profondeur d'un arbre, chaque sommet est rencontré une première fois, avant tous ses fils, et une dernière fois, après tous ses fils, ce qui induit deux ordres classiques sur les sommets: l'ordre dit *préfixe* (ou préordre) et l'ordre dit *postfixe* (ou postordre).

ii. Donner les ordres préfixe et postfixe de l'arbre A_1.

iii. Donnez deux versions du parcours en profondeur. Dans la première version, le tableau `p[s]` contiendra un ordre préfixe, dans la seconde, il contiendra un ordre postfixe.

Un arbre binaire `B` est un arbre dont chaque sommet possède au plus deux successeurs. Un arbre binaire est donc une structure de donnée munie de trois fonctions : `racine(B)` qui renvoie un sommet appellée la racine de B, `fils_gauche(B)` et `fils_droit(B)` qui renvoient chacun un arbre.

Dans le parcours `PP` d'un arbre binaire, chaque sommet est rencontré trois fois (avant son fils gauche, entre les deux fils et après son fils droit). On obtient ainsi un troisième ordre classique sur les sommets: l'ordre dit *infixe* (ou symétrique).

iii. Donner l'ordre préfixe, postfixe et infixe de l'arbre binaire B_1.

iv. Décrire trois versions de l'algorithme `PP` pour un arbre binaire de façon à remplir le tableau `p[s]` suivant les trois ordres.

Solutions des exercices optionnels

Correction de l'exercice 4

i) Voici l'algorithme de parcours en profondeur modifié :

```
PP_ARBRE(A)                      Visiter_PP_ARBRE(A)
1 temps <- 0                     1  p[racine(A)] <- temps <- temps + 1
2 si A <> nil alors              2  pour chaque B de fils(A) faire:
3     Visiter_PP_ARBRE(A)        3      Visiter_PP_ARBRE(B)
```

Voici l'algorithme de parcours en largeur modifié :

```
PL_ARBRE(A)
1 si A <> nil alors:
2     p[racine(A)] <- temps <- 1
3     d[racine(A)] <- 0
4     Enfiler(F, A)
5     tant que non vide(F) faire:
6          B <- tete(F)
7          pour chaque C de fils(B) faire:
8               p[racine(C)] <- temps <- temps + 1
9               d[racine(C)] <- d[racine(B)] + 1
10              Enfiler(F, C)
11         Defiler(F)
```

ii) L'ordre préfixe de A_1 est $[1, 2, 5, 3, 4, 6, 8, 7]$.
L'ordre postfixe de A_1 est $[5, 2, 3, 8, 6, 7, 4, 1]$.

iii) La première version du Parcours en Profondeur, celle de la question *i*), calcule déjà l'ordre prefixe.

Voici donc la version du Parcours en Profondeur qui calcul l'ordre postfixe :

```
PP_ARBRE_POST(A)                 Visiter_PP_ARBRE_POST(A)
1 temps <- 0                     1  pour chaque B de fils(A) faire:
2 si A <> nil alors              2      Visiter_PP_ARBRE_POST(B)
3     Visiter_PP_ARBRE_POST(A)   3  p[racine(A)] <- temps <- temps + 1
```

iv) L'ordre préfixe de B_1 est $[5, 2, 1, 4, 3, 7, 6]$.
L'ordre infixe de B_1 est $[1, 2, 3, 4, 5, 6, 7]$.
L'ordre postfixe de B_1 est $[1, 3, 4, 2, 6, 7, 5]$.

v) Voici maintenant les trois versions de l'algorithme PP qui calcule les trois ordres demandés :

```
0 temps <- 0

PREFIXE(A)
1 si A <> nil alors:
2    p[racine(A)] <- temps <- temps + 1
3    PREFIXE(fils_gauche(A))
4    PREFIXE(fils_droit(A))

POSTFIXE(A)
1 si A <> nil alors:
2    POSTFIXE(fils_gauche(A))
3    POSTFIXE(fils_droit(A))
4    p[racine(A)] <- temps <- temps + 1

INFIXE(A)
1 si A  <> nil alors:
2    INFIXE(fils_gauche(A))
3    p[racine(A)] <- (temps <- temps + 1)
4    INFIXE(fils_droit(A))
```