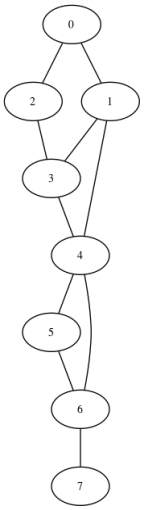
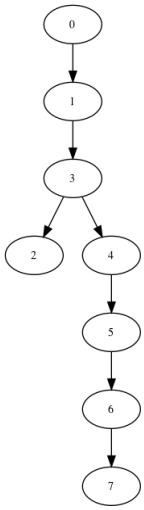
1 Parcours et numérotations préfixe et suffixe



graphe G



un parcours de G

M : (0, 1, 3, 2, 4, 5, 6, 7)
N : (2, 7, 6, 5, 4, 3, 1, 0)

ordres associés

sommet	numéro préfixe	numéro suffixe
0	1	8
1	2	7
2	4	1
3	3	6
4	5	5
5	6	4
6	7	3
7	8	2

numérotations correspondantes

FIGURE 1 – Un graphe non orienté G et les données d'un parcours en profondeur d'abord de ce graphe partant du sommet 0

Deux ordres sur les sommets nous intéressent dans le cadre des parcours :
— l'ordre dans lequel les sommets sont visités, qui correspond à l'ordre dans lequel débute l'exploration des sommets (représenté par la liste M dans le cours) ;
— l'ordre dans lequel s'achève l'exploration des sommets (représenté par la liste N dans le cours).

On appelle numérotations *préfixe* et *suffixe* le rang des sommets dans ces deux ordres : le premier sommet visité est de numéro préfixe 1, le deuxième sommet visité est de numéro préfixe 2, et ainsi de suite ; le premier sommet totalement exploré est de numéro suffixe 1, le deuxième sommet totalement exploré est de numéro suffixe 2, et ainsi de suite. Un exemple est fourni dans la figure 1.

Question 1 il s'agit d'implémenter en C une version itérative (semblable à celle du cours) de l'algorithme de parcours en profondeur d'abord. Pour cela :

1. vous devez vous munir d'une bibliothèque de manipulation des graphes ;
2. vous devez doter cette bibliothèque d'une fonction qui, étant donné un graphe, en fait le parcours en profondeur d'abord, et fournit la numérotation préfixe, la numérotation suffixe, et la forêt orientée associées à ce parcours. Idéalement, votre fonction doit se dérouler en temps $O(n^2)$ si votre graphe est représenté par matrice d'adjacence, $O(n+m)$ s'il est représenté par listes d'adjacence.
3. Vous devez écrire un programme dont l'objet est de tester votre fonction sur le graphe de la figure 1, et dont le déroulement devra produire une trace similaire à celle de la figure 2. Dans le dernier tableau de cette trace, la colonne `pere` indique pour chaque sommet son père dans l'arborescence du parcours.

Pour les structures de données manipulées :

```
 **** graphe de test:
n = 8, m = 0
 0  1  1  0  0  0  0  0
 1  0  0  1  1  0  0  0
 1  0  0  1  0  0  0  0
 0  1  1  0  1  0  0  0
 0  1  0  1  0  1  1  0
 0  0  0  0  1  0  1  0
 0  0  0  0  1  1  0  1
 0  0  0  0  0  0  1  0
*** donnees obtenues pour le parcours (version iterative):
 sommet  prefixe  suffixe  pere
 0       1        8        -
 1       2        7        0
 2       4        1        3
 3       3        6        1
 4       5        5        3
 5       6        4        4
 6       7        3        5
 7       8        2        6
```

FIGURE 2 – Traces attendues du déroulement du programme test sur le graphe G de la figure 1

— pour les graphes, vous pouvez utiliser l'une des bibliothèques fournies en TP, ou votre propre bibliothèque ;
— pour la liste d'exploration, vous devez utiliser la bibliothèque liste fournie avec le sujet ;
— pour représenter les données du parcours (numérotation préfixe, numérotation suffixe et forêt du parcours), vous pouvez manipuler trois tableaux d'entiers séparés, comme vous munir d'une structure de données plus élaborée.

2 Algorithme récursif

Une écriture récursive des parcours en profondeur d'abord est assez naturelle : parcourir (en profondeur d'abord) un graphe G depuis un sommet r, c'est visiter r, puis parcourir G depuis les voisins (non encore visités) de v. Autrement dit, on remplace la gestion explicite d'une pile d'exploration par l'empilement des appels récursifs.

Question 2 il s'agit d'implémenter en C une version récursive de l'algorithme de parcours en profondeur d'abord. Pour cela :

1. vous devez enrichir la bibliothèque graphe utilisée d'une fonction qui, étant donné un graphe, en fait un parcours en profondeur d'abord, et fournit la numérotation préfixe, la numérotation suffixe, et la forêt orientée associées à ce parcours. Cette fonction doit idéalement se dérouler en temps $O(n^2)$ si votre graphe est représenté par matrice d'adjacence, $O(n+m)$ s'il est représenté par listes d'adjacence.
2. Vous devez dans votre programme tester aussi cette nouvelle fonction sur le graphe de la figure 1, et en afficher le résultat (voir figure 3).

3 Complexité

Question 3 il s'agit de mesurer l'efficacité de vos fonctions. Précisément :

1. en combien (en ordre de grandeur, au pire des cas) d'opérations élémentaires vos fonctions de parcours se terminent-elles sur un graphe d'ordre n ayant m arêtes ?
2. Combien (en ordre de grandeur, au pire des cas) d'octets vos fonctions occupent-elles en mémoire sur un graphe d'ordre n ayant m arêtes ?

```
**** donnees obtenues pour le parcours (version récursive):
sommet  prefixe  suffixe  pere
0       1        8        -
1       2        7        0
2       4        1        3
3       3        6        1
4       5        5        3
5       6        4        4
6       7        3        5
7       8        2        6
```

FIGURE 3 – Trace du programme associée au test de la fonction récursive de parcours

3. S'il faut choisir, recommanderiez-vous une écriture itérative, ou récursive de l'algorithme de parcours en profondeur d'abord ?

4 Consignes

Le travail est à réaliser seul ou en binôme, à remettre sous forme d'une archive sur l'ent, **au plus tard le mardi 15 décembre midi**. L'archive doit contenir vos fichiers source (.h, .c) pour les questions 1 et 2, un fichier texte ou pdf pour l'analyse de complexité pour la question 3, un fichier texte README dans lequel vous précisez les nom, prénom, numéro étudiant et groupe TD du ou des contributeurs, le contenu de l'archive, les instructions à suivre pour compiler et exécuter votre programme.

Un makefile sera apprécié, mais n'est pas requis. Concernant le code, il va sans dire qu'il doit être utilement commenté, et compiler sans avertissement. Il ne doit pas non plus produire de fuite mémoire.

Barème indicatif : 2/3 sur le code, 1/3 sur la complexité.

Contrat de base (permettant d'atteindre ~ 15 si l'analyse de complexité est correcte) :

— respect des formats et contraintes demandés pour le rendu ;
— le code fourni doit compiler et s'exécuter sans avertissement, produire le résultat escompté, dans le temps imparti, et ce sans perte mémoire ;
— le code (alignement, organisation, commentaires) et les analyses de complexité (argumentation) doivent être proprement présentés.

Au-delà :

— optimisation du code (notamment, en utilisant une structure graphe, possiblement personnelle, la plus appropriée) ;
— organisation du code (notamment, définition d'une structure pour les parcours).