TP2 : Mini-Projet Bibliothèque de Graphes Codage par listes d'adjacences

à rendre avant la 7ème séance de TP
travail à réaliser **en binôme**

Avant propos

Pour tous les travaux pratiques à rendre, il est indispensable de se soumettre à certaines règles :
- **l'énoncé doit être suivi à la lettre**. Il constitue le cahier des charges sur lequel vous êtes évalués. Ainsi, il est inutile d'en faire trop si ce n'est lorsqu'une partie est laissée libre. Dans le cas contraire, les parties en plus ne seront pas notées, sauf justification de votre part dans un dossier ;
- si des **noms de variables** sont spécifiés dans l'énoncé, ils doivent être employés obligatoirement et littéralement ;
- la date de livraison du TP devra être respectée scrupuleusement. Les travaux pratiques seront à rendre par **dépôt Moodle** avant la séance de TP de la semaine indiquée sur le sujet. La date exacte vous sera communiquée en temps utile. Si un dossier doit être rendu, vous le rendrez soit par dépôt Moodle avec votre projet, ou alors vous le placerez en version papier dans le casier du correcteur avec une date apposée par le secrétariat ;
- si votre TP ne comprend qu'un fichier, il devra porter comme nom les deux noms juxtaposés du binôme, suivi de l'extension qui convient. Si votre TP est fait de plusieurs fichiers, il devra être rendu sous la forme d'une archive ZIP ou TGZ composée d'un répertoire nommé selon les mêmes conventions, et dans lequel figureront uniquement les fichiers sources et un `Makefile`. Cette archive doit se déployer dans un répertoire portant les noms du binôme ;
- les codes que vous rendrez devront être conformes au **standard de programmation**.

Sujet : Manipulation d'un graphe par listes d'adjacences

Le but de ce TP est de développer en C une structure de données de type graphe, adaptée à une gestion dynamique pour laquelle :
- l'occupation mémoire n'est réservée qu'au coup par coup,
- la continuité mémoire n'est pas assurée,
- l'insertion et la suppression d'un sommet ou d'une arête est possible,
- l'accès aux sommets suivant n'est pas direct.

N.B. Il est doublement important de réaliser correctement ce sujet qui sera évalué en tant que tel, mais qui servira également à programmer le projet qui sera proposé en deuxième partie de semestre.

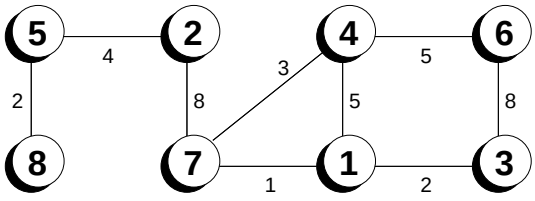

Fig. 1 – Exemple d'un graphe G non orienté

Soit $G = (S, A)$ un graphe pondéré non orienté avec S l'ensemble des sommets de G et A l'ensemble des arêtes. Un exemple de graphe non orienté est dessiné à la figure 1. Le stockage

en mémoire de ce graphe est fait par le tableau de listes d'adjacences illustré par la figure 2. Vous aurez noté que ce codage permet de traiter à la fois le cas des graphes orientés et non orientés, car un graphe non orienté peut se coder sous la forme d'un graphe orienté symétrique. Par convention, on désigne un sommet isolé comme étant un sommet ayant pour seul *voisin* le sommet fictif "`-1`". Les sommets pour lesquels la valeur dans le tableau est celle du symbole de pointeur `NULL` n'appartiennent pas au graphe (c'est le cas dans l'exemple des sommets 9 et 10).

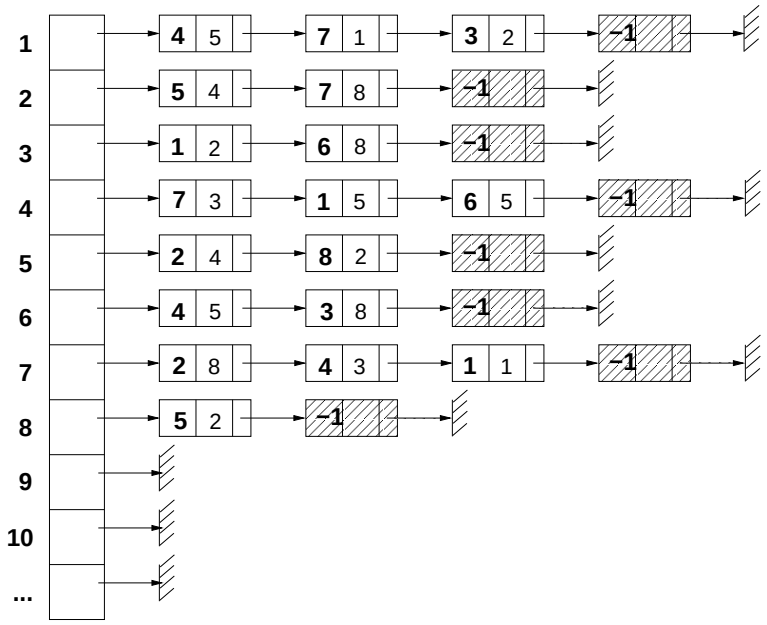

FIG. 2 – Représentation du graphe G par un tableau de listes d'adjacences. Attention, un graphe non orienté est codé comme un graphe orienté pour lequel chaque arête (x, y) de G implique que l'arête (y, x) existe aussi.

La liste des voisins d'un sommet peut également être gérée par une liste doublement chaînée circulaire avec sentinelle. Ici, un sommet appartenant au graphe est un sommet dont la liste des voisins désigne au minimum la liste vide (c'est à dire la sentinelle). Un exemple de liste chaînée avec sentinelle est donné par la figure 3. Cette représentation a l'avantage de simplifier les opérations de gestion des listes.

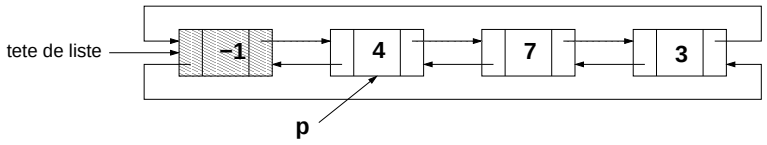

FIG. 3 – Exemple de liste doublement chaînée circulaire avec sentinelle

Le graphe G est stocké sous la forme d'un tableau de listes d'adjacences. Les indices du tableau correspondent aux numéros de sommets, éventuellement (à vous de gérer à votre convenance) avec un décalage de une unité car les indices des tableaux commencent le plus souvent par 0 alors que les numéros des sommets du graphe commencent à 1. Le contenu de chaque case est l'adresse de la liste des successeurs du sommet désigné par cette case. La liste des successeurs est l'une des listes chaînées décrites précédemment, construite dans l'ordre des insertions des

sommets voisins et non dans l'ordre croissant de leur numéro. Que l'insertion se fasse en début ou fin de liste, la complexité de cette opération est la même dans le cas où la structure de la liste est du type liste doublement chaînée (avec sentinelle ou non). Par contre, avec des listes simplement chaînées, pour des raison d'efficacité, les insertions doivent se faire en début de liste. Afin de faciliter la manipulation du graphe, la structure de données définie par le tableau précédemment décrit est complétée par le nombre maximum de sommets de ce graphe. Ainsi on définit une structure de type `TypGraphe` avec comme champs :

1. `estOriente`, qui indique si le graphe est orienté ou pas,
2. `nbMaxSommets`, le nombre maximum de sommets pouvant appartenir au graphe,
3. `listesAdjacences`, un tableau dont les cases sont de type `TypVoisins`. Le type `TypVoisins` est composé des champs suivants :
   - `voisin`, un entier qui stocke le numéro du voisin du sommet courant ;
   - `voisinSuivant`, un pointeur sur la cellule suivante dans l'ordre de la liste, c'est à dire sur le voisin suivant du sommet courant ;
   - `voisinPrecedent`, un pointeur sur la cellule précédente dans l'ordre de la liste, c'est à dire sur le voisin précédent du sommet courant (uniquement pour la représentation d'une liste doublement chaînée circulaire avec sentinelle) ;

Travail à réaliser :

Afin de réaliser le travail demandé ci après, vous construirez deux bibliothèques en C les plus générales possible, l'une utile à la manipulation des listes chaînées d'un des deux types décrits ci-dessus et l'autre utile à la gestion des graphes stockés sous la forme d'un tableau de listes d'adjacences. De cette manière, toute application sur les listes ou les graphes pourra utiliser ces bibliothèques pour effectuer les manipulations élémentaires telles que les ajouts, les suppressions, les parcours, etc. La construction de ces bibliothèques doit être faite dans un esprit de réutilisation.
Le programme C que vous écrirez au final pour ce TP utilisera ces bibliothèques. Il devra permettre l'affichage d'un menu donnant accès aux fonctionnalités suivantes :

- `creation` : création d'un graphe en fonction d'un nombre *maximum* de sommets, saisi au clavier, si celui-ci n'a pas encore été créé ;
- `lecture` : lecture d'un graphe dans un fichier texte suivant le format donné par la figure 4 ;
- `insertionSommet` : insertion d'un sommet dans le graphe (numéro inférieur au nombre maximum de sommets dans le graphe) ;
- `insertionArete` : insertion d'une arête du graphe après vérification que les sommets extrémités existent et que cette arête n'est pas déjà présente dans le graphe. Il sera demandé alors s'il s'agit d'une arête symétrique "`o`" ou non "`n`" (sauf dans le cas d'un graphe totalement non orienté où chaque arête sera vue comme symétrique) ;
- `suppressionSommet / suppressionArete` : suppression d'un sommet ou d'une arête entre deux sommets de G ;
- `affichage` : affichage du graphe dans le même format que celui du fichier texte en entrée ;
- `sauvegarde` : sauvegarde du graphe dans le format texte donné figure 4 ;
- `quitter`.

Pour le codage de toutes ces opérations de base, vous devrez tenir compte du fait que le graphe est déjà créé ou non. En effet, si le graphe n'est pas encore créé, nous avons seulement le choix de le créer à la main, de saisir le nom d'un fichier contenant la description du graphe à charger ou bien de quitter le programme. Toutes les cellules allouées dynamiquement lors de l'exécution du programme doivent être libérées soit lors d'une suppression ou soit à la fin du programme

(ce point sera vérifié à l'aide du programme `valgrind` lors de la correction). Aucune opération ne doit conduire à une sortie du programme à part la fonction `quitter` qui libère toute la mémoire avant d'autoriser la fin du programme. À chaque opération, un message devra être délivré à l'utilisateur afin qu'il ait connaissance de ses erreurs ou des manipulations réussies. Pour cela il est recommandé de coder toutes vos fonctions avec des retours d'erreurs analysables par la fonction appelante afin qu'elle puisse retourner elle même un code d'erreur à sa fonction appelante ou délivrer le bon message à l'intention de l'utilisateur. Ces messages peuvent être par exemple, "*saisie d'une arête déjà existante*", "*suppression d'un sommet inconnu*", "*arête (x,y) ajoutée au graphe*", etc).

```
# nombre maximum de sommets
11
# oriente
n
# sommets : voisins
1 : (4/5), (7/1), (3/2)
2 : (5/4), (7/8)
3 : (1/2), (6/8)
4 : (7/3), (1/5), (6/5)
5 : (2/4), (8/2)
6 : (4/5), (3/8)
7 : (2/8), (4/3), (1/1)
8 : (5/2)
```

FIG. 4 – Structure du fichier texte de définition et de sauvegarde d'un graphe

NB 1 : La mise en œuvre de ce sujet nécessite évidemment l'écriture de nombreuses fonctions. L'écriture de fonctions claires, judicieusement nommées, de petites taille (max un écran) et modulaires sera appréciée lors de la notation. La facilité et la simplicité d'utilisation seront également prises en compte. Le programme principal ne devra pas gérer le menu. Une fonction d'affichage et une fonction de gestion de ce menu devront être écrites.

Le code produit doit impérativement tenir compte des recommandations données par le standard de programmation. Il sera tenu compte de son respect dans la notation du TP.

NB 2 : l'utilisation des bibliothèques se concrétise par l'utilisation de leur API dans le programme principal ou tout autre code source codant les fonctions utiles. Dans la compilation de la partie applicative de ce TP (partie programme d'exploitation des graphes avec le menu permettant sa manipulation), la bibliothèque apparaît bien comme une bibliothèque (`-lgraphe -lliste` dans la ligne de compilation respectivement pour les bibliothèques dans l'ordre inverse des dépendances pour les fichiers `libgraphe.a` et `libliste.a`) et non comme des fichiers `.o`. Pour cela votre `makefile` doit fabriquer les bibliothèques si cela est nécessaire en fonction des dépendances avec la ligne de compilation idoine et doit ensuite compiler tout fichier source permettant de tenir compte des seules modifications apportées depuis la dernière compilation. Une attention particulière doit donc être portée aux dépendances entre les différentes règles, à l'utilisation des variables et à la généricité de la compilation. Les codes et autres résultats de compilation sont rangés respectivement dans les dossiers `SRC`, `OBJ`, `BIN`, `LIB`. Le dossier `INCLUDE` est optionnel.