TP Graphes 3

Licence Informatique - 2nde année

Année 2025-2026

L'objectif de ce TP est double : (i) appliquer les principes de la compilation séparée, qui faciliteront les développements que vous ferez dans ce TP et les suivants et (ii) implanter l'algorithme de parcours en largeur de graphes, tout en expérimentant sur la manière d'utiliser les données qui sont générées. Afin d'éviter tout souci de codage lié aux exercices qui étaient à réaliser dans le TP précédent, vous récupérerez l'archive jointe à cet énoncé, qui fournit certaines des fonctions qui étaient à développer dans ce tp dans le fichier `tp3.cpp`.

Exercice 1

A partir de ce qui a été vu en cours sur le principe de la compilation séparée, répartissez les sources disponibles dans le fichier `tp3.cpp` en deux modules :

— `tp3.cpp` qui ne contiendra que la fonction `main` ;

— `matrice.cpp` qui contiendra toutes les fonctions en lien avec la gestion des matrices d'adjacence.

Vous y ajouterez le fichier `matrice.hpp` qui contiendra la définition des types utilisés pour la représentation des matrices d'adjacence, ainsi que la définition des fonctions exportées par la module `matrice.cpp`.

Vous finaliserez l'exercice en créant le fichier `Makefile` permettant la compilation et testerez que tout fonctionne correctement à partir des fichier fournis dans le dossier `Data`.

Exercice 2

Il a été vu en cours que l'implantation de la méthode de parcours en largeur d'un graphe nécessite l'utilisation d'une file de type FIFO (*First In - Fisrt Out*). L'objectif de cet exercice est donc d'écrire les fonctions qui permettront de gérer cette structure, afin de pouvoir l'utiliser dans l'algorithme de parcours en largeur. **Les développments liés à cette file devront être effectués dans un nouveau module nommé `fifo.cpp`, associé à son module d'export `fifo.hpp`.**

Structure de données

La structure que vous utiliserez pour représenter cette file sera celle de **liste doublement chaînée**, c'est à dire que chaque maillon de la liste chaînée connaîtra l'élément qui le suit dans la liste, ainsi que celui qui le précède. La structure de données correspondante, illustrée sur la figure 1, sera la suivante :

```
// structure d'un maillon d'une fifo de valeurs entières

struct MaillonEntier {
  int valeur; // valeur stockée dans le maillon
  MaillonEntier *suiv;// élément suivant dans la file
  MaillonEntier *prec;// élément précédent dans la file
} ;

// structure d'une file de type FIFO pouvant contenir des entiers

struct Fifo {
  MaillonEntier *in;// pointeur vers le début de la liste
  MaillonEntier *out;// pointeur vers la fin de la liste
};
```

Une file sera ainsi représentée par les deux pointeurs `in` et `out`, qui représenteront respectivement le début et la fin de la liste dans laquelle sont stockées les valeurs entrées dans la file. La liste en elle-même ne stockera que des nombres entiers, qui représenteront par la suite des numéros de sommet.

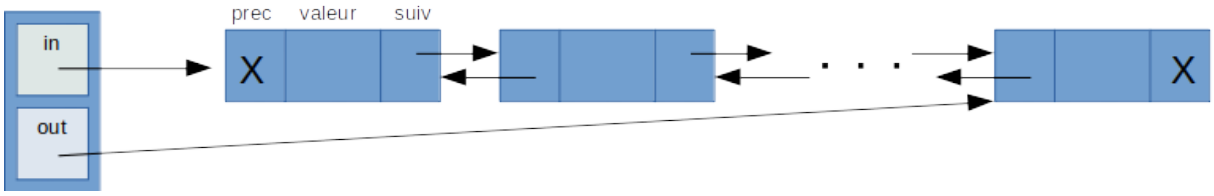

FIGURE 1 – Représentation d'une file de type FIFO. Elle est constituée de deux pointeurs, l'un permettant d'accèder à la tête de la liste chaînée des éléments qui y sont stockés, l'autre au dernier élément de cette liste.

Question 1

Ecrire le code de la fonction suivante :

```
void initialiserFifo(Fifo *file);
```

Cette fonction a juste pour objectif d'initialiser à `nullptr` ses deux membres.

Exemple d'utilisation :

```
Fifo  mafile;

initialiserFifo(&mafile);
```

Question 2

Ecrire le code de la fonction suivante :

```
bool estVide(Fifo file);
```

Cette fonction retourne `true` si la file passée en paramètre est vide, `false` sinon.

Question 3

Afin de pouvoir tester ces fonctions et celles qui suivent, vous allez écrire le code de la fonction suivante :

```
bool afficherFifo(Fifo file);
```

Cette fonction devra afficher la liste des éléments présents dans la file, en partant du dernier entré vers le premier (donc de gauche à droite par rapport aux schémas qui vous sont fournis).

Vous modifierez le fonction `main` afin de permettre de tester ces 3 premières fonctions.

Question 4

Ecrire le code des deux fonctions suivantes :

```
void ajouter(Fifo *file, int v);
int retirer(Fifo *file);
```

La première permet d'ajouter la valeur `v` dans la file. Conformément au mode de fonctionnement d'une FIFO, la valeur est insérée en tête de la liste chaînée représentant les éléments que cette file stocke. La seconde permet de renvoyer la valeur entière stockée en fin de liste chaînée et d'effacer le maillon correspondant. La figure 2 suivante illustre l'état de la file après 4 opérations d'ajout et une de retrait.

Vous considérerez que la file n'est pas vide, lors d'un appel à la fonction `retirer`. Vous testerez ces deux nouvelles fonctions dans votre application, afin d'être certains de leur validité avant de passer à l'exercice suivant.

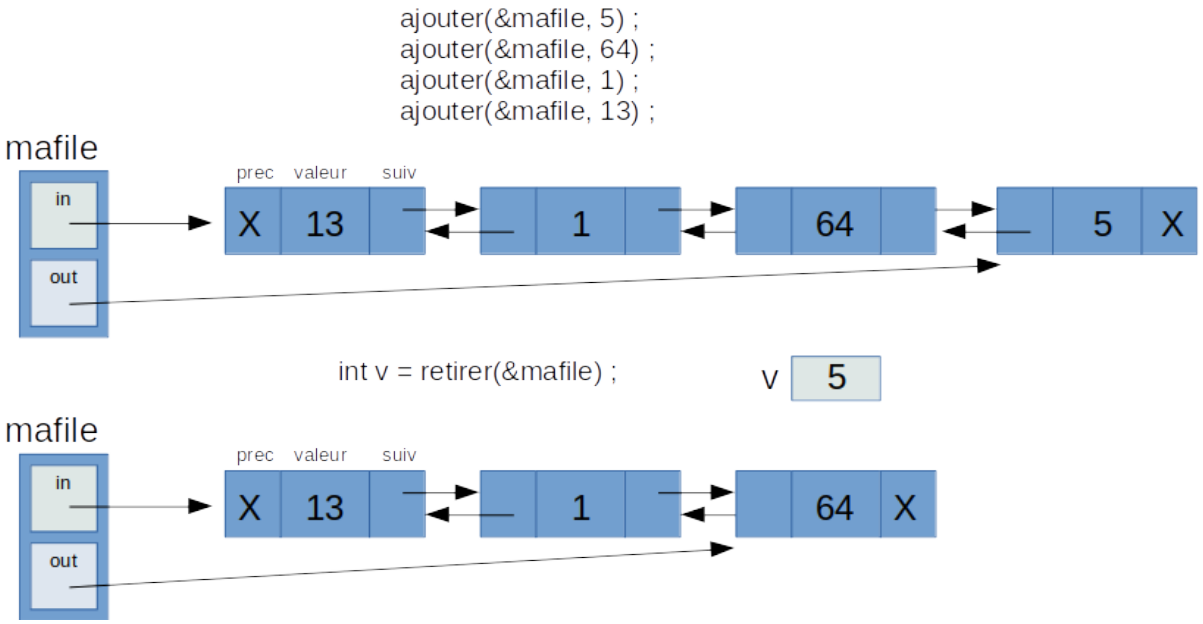

FIGURE 2 – Etat de la file après ajout de 4 valeurs entières (en haut), puis retrait d'une valeur (en bas).

Exercice 3

L'objectif de cet exercice est d'implémenter le parcours d'un graphe en largeur, en suivant l'algorithme qui a été détaillé en cours. Les fonctions nécessaires à cet exercice devront être implémentées dans un module nommé `parcours.cpp` et son module d'export associé.

Question 5

Ecrire le code de la fonction suivante :

```
void parcoursEnLargeur(MatriceAdjacence mat, int sommetDepart,
                       Couleur *coul, int *dist, int *parent)
```

Cette méthode doit réaliser le parcours en largeur du graphe à partir du sommet dont l'indice est donné dans le paramètre `sommetDepart`. Ses paramètres sont :

- `mat` : la matrice d'adjacence à parcourir ;
- `sommetDepart` : l'indice du sommet à partir duquel effectuer le parcours
- `coul` : un tableau de couleurs, de même taille que l'ordre de la matrice. Le tableau aura été alloué avant l'appel de la la fonction, mais pas initialisé. Vous définirez, à l'endroit qui vous semble adéquat, le type énuméré suivant :

```
/*
 * type énuméré permettant de représenter les couleurs
 */
    enum Couleur {BLANC, GRIS, NOIR};
```

- `dist` : un tableau de distances par rapport au sommet de départ, de même taille que l'ordre de la matrice. En sortie de la fonction, chaque case contiendra le nombre d'étapes nécessaires pour aller du sommet de départ au sommet dont l'indice correspond à l'indice de la case. Le tableau aura été alloué avant l'appel de la fonction, mais pas initialisé. Vous définirez une constante/macro nommée `INFINI`, qui prendra ici la valeur -1, à l'endroit qui vous semble adéquat. Elle sera utilisée pour initialiser le tableau de distances ;
- `parent` : un tableau mémorisant l'indice du sommet parent à un sommet, sur le chemin permettant de joindre ce dernier au sommet de départ. Le tableau aura été alloué avant l'appel de la la fonction, mais pas initialisé.Vous définirez une constante/macro nommée `INDEFINI`, qui prendra ici la valeur -1, à l'endroit qui vous semble adéquat. Elle sera utilisée pour initialiser le tableau de parents.

Vous testerez votre fonction sur les matrices fournies dans le dossier Data, à partir du noeud d'indice 0. Vous ferez afficher les résultats obtenus dans les trois tableaux sous la forme suivante :

- Pour les couleurs : la valeur de chaque case, avec `B` pour Blanc, `G` pour Gris et `N` pour Noir ;

— Pour les distances : la valeur de chaque distance, remplacée par `X` lorsque celle-ci est infinie ;
— Pour les parents : l'indice de chaque parent, remplacé par `X` lorsque celui-ci est indéfinie.

Exemple

```
./tp3 Data/matrice04.txt

couleurs  :  N  N  B  N  N  N  B
distances :  0  1  X  1  2  2  X
parents   :  X  0  X  0  1  1  X
```

Question 6

On souhaite à présent pouvoir afficher le chemin à emprunter depuis le sommet de départ vers chacun des sommets atteignables. La seule information dont on dispose à l'issue du parcours en largeur est l'indice du sommet parent pour chaque sommet du graphe, c'est à dire la dernière partie de chaque chemin. Il est donc nécessaire de remonter ces chemins pour retrouver le sommet de départ et chacun des sommets qui constitue chaque chemin et, enfin afficher le chemin dans le bon ordre ...

Vous allez implanter une fonction **récursive** qui permet d'effectuer cette opération. Son proptype sera le suivant :

```
void afficherCheminVers(int sf, int *parent)
```

avec `sf` l'indice du sommet vers lequel on souhaite se diriger et `parent` le tableau des parents de chaque sommet. Cette fonction sera appelée dans le programme principal pour chaque sommet (sauf le sommet de départ ...), et son utilisation générera un affichage tel que celui figurant ci-dessous, où on suppose que le sommet de départ est le sommet d'indice 0 :

```
./tp3 Data/matrice04.txt

couleurs  :  N  N  B  N  N  N  B
distances :  0  1  X  1  2  2  X
parents   :  X  0  X  0  1  1  X

chemin vers 1 = 0 1
pas de chemin de 0 vers 2
chemin vers 3 = 0 3
chemin vers 4 = 0 1 4
chemin vers 5 = 0 1 5
pas de chemin de 0 vers 6
```

Question 7

Modifiez votre fonction `main` de telle sorte qu'elle puisse récupérer le numéro du sommet de départ comme second paramètre sur la ligne de commande. La syntaxe de lancement de votre application deviendra alors :

```
./tp3  fichierMatrice.txt  numeroDuSommetDeDépart
```

La fonction devra vérifier que ce paramètre supplémentaire est bien présent et que sa valeur correspond bien à un indice autorisé dans le graphe qui a été chargé. On précise que pour convertir ce second paramètre, qui est considéré comme une chaîne de caractères, en sa valeur entière, vous pourrez utiliser la fonction :

```
int atoi(char *str)
```

qui convertit la chaîne de caractères passée en paramètre et supposée ne contenir que des caractères numériques, en sa valeur entière.

Exemples

```

./tp3 Data/matrice04.txt
Erreur - Syntaxe = ./tp3 <file>  <indice_sommet_depart>

./tp3 Data/matrice04.txt 0

```

```
couleurs  : N N B N N N B
distances : 0 1 X 1 2 2 X
parents   : X 0 X 0 1 1 X

chemin vers 1 = 0 1
pas de chemin de 0 vers 2
chemin vers 3 = 0 3
chemin vers 4 = 0 1 4
chemin vers 5 = 0 1 5
pas de chemin de 0 vers 6

./tp3 Data/matrice04.txt 7
indice de sommet 7 incorrect ! valeurs autorisées dans [0,6]

./tp3 Data/matrice04.txt 5

couleurs  : N N B N N N B
distances : 2 1 X 2 1 0 X
parents   : 1 5 X 1 5 X X

chemin vers 0 = 5 1 0
chemin vers 1 = 5 1
pas de chemin de 5 vers 2
chemin vers 3 = 5 1 3
chemin vers 4 = 5 4
pas de chemin de 5 vers 6
```