

Travaux Pratiques

séance nr. 3 et 4 - Graphe de voirie urbaine

Dans ce TP nous allons implanter une structure de "graphe orienté".

On souhaite modéliser un réseau de voirie urbaine recensant voies publiques et carrefours. Pour cela on construit un graphe valué dont les sommets sont les carrefours et les arêtes les rues. Chaque carrefour possède un nom qui est un code : 2 lettres suivies de 3 chiffres (un nombre). Chaque rue possède un nom (une chaîne de caractères de longueur arbitraire). Certaines rues sont à sens unique : c'est pourquoi le graphe est orienté.

Le schéma ci-dessous présente un exemple. Les points noirs sont les sommets qui marquent les carrefours. Chaque carrefour a un nom (ici, ZW231, ZW232, etc.). Chaque rue possède également un nom. Une même rue peut comprendre plusieurs carrefours, donc correspondre à plusieurs arêtes dans le graphe : voir la « rue Saint-Hippolyte ». Les flèches indiquent le sens unique, par exemple « rue de l'ancienne école ».



La modélisation de la voirie urbaine se fait en deux parties : la première concerne la table des rues et des carrefours. La seconde la modélisation du graphe à proprement parlé qui n'utilisera que des entiers naturels (les identifiants des sommets et des rues).

1. Table des carrefours et des rues

Les rues et les carrefours ont des noms (chaines de caractères). Ils seront stockés à l'aide de deux **tables de chaînes de caractères**. Ces deux tables sont de même type, mais séparées. Chaque carrefour et chaque rue est identifiée par un entier non signé. Les structures associées au type *table de chaîne de caractères* (appelée **StringTab**) sont les suivantes :

```
#define N_ELEM_INIT 100
typedef unsigned int Nat;
typedef char *String;

typedef struct s_couple { // adressage associatif
    String chaine;
    Nat ident;
} couple;

typedef struct s_stringtab {
    couple *tab;
    Nat n_elem; // nombre de chaines
    Nat max_elem; // taille du tableau tab
    Nat curr_id; // pour générer les identifiants uniques
} StringTab;
```

Dans **StringTab**, l'identifiant associé à chaque chaine est un entier positif non nul. Il est unique et reste inchangé. Il est généré automatiquement lors de l'ajout d'une chaine, à l'aide de *curr_id* initialisé à 1 puis incrémenté à chaque nouvel ajout. Les couples chaine/identifiant sont rangés dans un tableau contigu, alloué dynamiquement (adressage associatif). En utilisant une allocation dynamique du tableau, celui-ci peut être agrandi s'il devient trop petit. Les couples sont rangés de façon ordonnée, par identifiant croissant. Donc pour deux indices *i* et *j* dans *tab*, on a $i < j \Rightarrow \text{tab}[i] \rightarrow \text{ident} < \text{tab}[j] \rightarrow \text{ident}$

Implémenter les opérations suivantes :

```
StringTab StringTabNouv();
void StringTabFree(StringTab t);
Nat StringTabCardinal(StringTab t);
Nat StringTabAdd(StringTab t, String str);
void StringTabSup(StringTab t, String str);

// recherche de l'identifiant d'une chaine (recherche séquentielle)
Nat StringTabRech(StringTab t, String str);
// recuperation de la chaine a partir de l'identifiant (recherche dichotomique)
String StringTabGetChaine(StringTab t, Nat index);
```

1. Listes d'entiers

Certaines opérations sur le graphe nécessitent la gestion d'ensembles ou de listes. Par exemple, l'opération qui renvoie tous les successeurs d'un sommet renvoie une liste d'identifiants. Il faut donc implémenter une structure de liste d'entiers non signés :

```
typedef struct s_maillonui {
    Nat data; // the data
    struct s_maillonui *succ; // next link
} *listeui;
```

```
// Operations
listeui nouvlui();
listeui adjtetelui(listeui l, Nat x);
listeui rechlui(listeui l, Nat x);
int ranglui(listeui l, Nat x);
listeui supkiemelui(listeui l, Nat k);
Nat kiemelui(listeui l, Nat k);
bool videlui(listeui l);
Nat longlui(listeui l);
void destroylui(listeui l);
```

2. Modélisation du graphe

Nous modélisons la structure **GrapheUrbain** en utilisant une **liste d'adjacence**. La structure est la suivante :

```
typedef struct strarc { Nat icf, irue; struct strarc *suc; } Strarc, *ListeSom;
typedef struct { Nat icf; ListeSom lsucc, lpred; } Sommet;
typedef struct strsom { Sommet c; struct strsom *suiv; } Strsom, *grapheurbain;
```

Chaque carrefour a un identifiant **Nat icf**. Le nom correspondant est obtenu via une première StringTab. Chaque rue a également un identifiant **Nat irue**. Le nom correspondant est obtenu via une seconde StringTab. Dans la structure GrapheUrbain on ne s'intéresse pas aux noms de rues et de carrefour. On ne s'intéresse qu'à leurs identifiants.

Dans ce graphe, nous modélisons à la fois la liste des successeurs (*lsucc*) et la liste des prédécesseurs (*lpred*) pour chaque sommet. Par ailleurs le graphe est valué : chaque arc porte l'identifiant d'une rue *irue*.

Pour chacune des opérations suivantes, indiquer les préconditions.

3.1 Définir et implémenter les opérations de création et destruction d'un **GrapheUrbain**.

```
grapheurbain grapheurbainNouv();
void grapheurbainFree(grapheurbain g);
```

3.2 Définir et implémenter des opérations d'adjonction / suppressions de carrefour.

```
void addCarrefour(grapheurbain g, Nat idcar);
void supCarrefour(grapheurbain g, Nat idcar);
```

3.3 Implémenter les opérations usuelles sur le graphe et notamment une opération permettant d'ajouter une arête au graphe, donc de relier deux carrefours par une rue. Les autres opérations sont : l'existence d'une connexion entre deux carrefours (càd qu'un carrefour fait partie des successeurs/prédécesseurs d'un autre carrefour), le nombre d'arcs et le nombre de sommets (carrefours).

```
void addArc(grapheurbain g, Nat idcarfx, Nat idcarfy, Nat idrue);
void addArcDoubleSens(grapheurbain g, Nat idcarfx, Nat idcarfy, Nat idrue);
void supArc(grapheurbain g, Nat idcarfx, Nat idcarfy);
bool existeArc(grapheurbain g, Nat idcarfx, Nat idcarfy);
bool existeCarrefour(grapheurbain g, unsigned int icar);
Nat nArcs(grapheurbain g);
Nat nCarrefours(grapheurbain g);
```

Note : ne pas confondre *arc du graphe* et *rue*. Une même rue peut traverser de multiples carrefours, alors qu'un arc relie toujours deux carrefours seulement.

Ecrire un code « main » qui permet de construire le graphe donné à titre d'exemple.

3.4 Implémenter une opération permettant, pour un carrefour donné, d'obtenir la liste de toutes les rues incidentes. Ajouter dans le main un code qui teste cette opération. Par exemple, l'utilisateur saisit: « ZW233 » et on affiche : rue Haute, rue Saint Hippolyte. Utiliser la StringTab pour récupérer les identifiants correspondants.

On définira les deux opérations suivantes :

```
// les carrefours associes a une rue
listeui carrefoursDeRue(grapheurbain g, unsigned int irue) ;

// les rues incidentes en un carrefour
listeui ruesDeCarrefour(grapheurbain g, Nat icar);
```

3.5 Implanter une opération permettant, pour une rue donnée, de récupérer la liste de toutes les rues qui la croisent. Ecrire dans le main un test. Par exemple, pour la « rue Zellenberg » on affiche : rue Ziegelfeld, rue d'Altkirch.

On définira l'opération :

```
listeui ruesCroisement(grapheurbain g, Nat irue)
```