Table de hachage

Cette partie peut être vue à la fois comme un TD sur feuille (en pseudo-code), ou un TP sur machine (en C). Il s'agit d'implémenter "à la main" le type abstrait "ensemble" (`set` en Python) via une table de hachage. Si vous souhaitez le faire sur machine, il est recommandé de l'écrire en C.

EXERCICE 1 – IMPLÉMENTATION DU TYPE "ENSEMBLE D'ENTIERS"

On définira deux structures :

```
Structure Noeud {
        valeur : entier
        suivant : pointeur vers Noeud
}

ListeChainee = pointeur vers Noeud
```

et

```
Structure Ensemble {
        tableHachage : tableau de ListeChainee
        taille : entier
}
```

En C, cela s'écrit :

```
struct Noeud{
        int valeur;
        struct Noeud* suivant;
};

typedef struct Noeud* ListeChainee;


struct Ensemble{
        ListeChainee* table;
        int taille;
};
```

Petite liste des opérations qu'il sera possible de faire en pseudo-code:

- Allouer de la mémoire pour un ensemble
- Allouer de la mémoire pour un noeud de liste chaînée
- Allouer un tableau de taille donnée
- Accéder et modifier à un membre d'une structure. Par exemple :
  - `Pour i allant de 0 à E.taille - 1`
  - `E.table[3] = pointeur nul`
  - `L = L.suivant`
  - ...

Q1. Ecrire une fonction `creeEnsemble` qui prend en paramètre un entier[1] `n` et qui renvoie un élément de structure Ensemble de taille `n`.

Correction. En pseudo-code :

```
Fonction creeEnsemble(n : entier){
        Allouer un nouvel ensemble E
        E.table = tableau de taille n remplis de vecteurs nuls
        E.taille = n
        Renvoyer E
}
```

En C :

```
struct Ensemble creeEnsemble(int n){

        struct Ensemble E;

        E.table = malloc(sizeof(ListeChainee)*n);
        for (int i = 0; i < n; i++) E.table[i] = NULL;
        /* Pour les deux dernières lignes, un "calloc"
        aurait pu très bien marcher aussi */

        E.taille = n;

        return E;
}
```

Q2. Ecrire une fonction `affiche` qui prend en paramètre un ensemble E et qui affiche tous les éléments dans E.

[1] Pourquoi impose-t-on ici une taille pour notre table de hachage, alors qu'en python ou en Java, quand on crée un ensemble, on ne précise jamais la taille ? Et bien, `python` ou `java` calcule automatiquement la taille de la table de hachage de sorte que chaque opération élémentaire associée à la notion d'ensemble se fasse en temps constant. Plus précisément, si le nombre d'éléments dans la table de hachage devient trop grand ou trop petit, on alloue une nouvelle table de hachage de taille plus adéquate, puis on recopie les valeurs de l'ancienne table de hachage vers la nouvelle. J'arrête les détails : cela reste un peu trop technique pour que ce soit réellement demandé dans le cadre de cet exercice.

Correction. En pseudo-code :

```
Fonction affiche(E : ensemble){
        Pour i allant de 0 jusqu'à E.taille - 1
        Faire
                L = E.table[i]
                Tant que (L n'est pas le pointeur nul)
                Faire   Afficher( L -> valeur )
                        L = L -> suivant
}
```

En C :

```
void affiche(struct Ensemble E){

        for(int i = 0; i < E.taille ; i++){

                ListeChainee L = E.table[i];
                while( L != NULL ){
                        printf("%d ", L->valeur );
                        L = L->suivant;
                }

        }
}
```

Q3. On va considérer la fonction de hachage :

$$h : x \mapsto (15073 \times x) \text{ modulo (taille de la table de hachage)}$$

Autrement dit, quand on va insérer un entier x dans l'ensemble, on va allouer un nouveau noeud contenant x et mettre l'adresse mémoire de ce noeud à la position $h(x)$ dans la table de hachage associée.

Écrire une fonction `ajout` qui prend en paramètre un ensemble `E` et un entier `x`, et qui rajoute dans `E` l'entier x. (On rajoutera une occurrence de x même si x est dans E)

Correction. En pseudo-code :

```
Fonction ajout(E : ensemble, x : entier){
        hx = (15073*x) modulo E.taille
        L = pointeur vers un nouveau noeud
        L.valeur = x
        L.suivant = E.table[hx]
        E.table[hx] = L
}
```

En C :

```
void ajout(struct Ensemble E, int x){

        int hx = (15073*x) % E.taille;
        ListeChainee L = malloc(sizeof(struct Noeud));
        L -> valeur = x;
        L -> suivant = E.table[hx];
        E.table[hx] = L;

}
```

Note : Pour être plus propre, il aurait mieux valu écrire la fonction de hachage en dehors de la fonction `ajout`, voire même de l'incorporer dans la structure `Ensemble`.

Q4. Écrire une fonction `appartient` qui prend en paramètre un ensemble `E` et un entier `x`, et qui renvoie `VRAI` si `x` appartient dans `E` ; `FAUX` sinon.

Correction. En pseudo-code :

```
Fonction appartient(E : ensemble, x : entier){
        hx = (15073*x) modulo E.taille

        L = E.table[hx]
        Tant que (L n'est pas nul)
        Faire Si L.valeur == x
                Alors Renvoyer VRAI
                SINON L = L->suivant

        Renvoyer L
}
```

En C :

```
bool appartient(struct Ensemble E, int x){
        int hx = (15073*x) % E.taille;
        ListeChainee L = E.table[hx];

        while(L != NULL){
                if (L->valeur == x) return true;
                L = L->suivant;
        }

        return false;
}
```

Q5. Écrire une fonction `supprime` qui prend en paramètre un ensemble `E` et un entier `x` (on supposera que E contient bien x), et qui supprime de E une occurrence de x.

Correction. En pseudo-code :

```
Fonction enleveOccurrence(L : ListeChainee, x : entier) {
        Si (L est nul) Alors Renvoyer Pointeur Nul
        /* Pas nécessaire si on suppose que L contient x */

        Si (L.valeur == x)
        Alors Renvoyer L.suivant

        L.suivant = enleveOccurrence(L.suivant,x)
        Renvoyer L
}

Fonction appartient(E : ensemble, x : entier){
        hx = (15073*x) modulo E.taille
        E.table[hx]     = enleveOccurrence(E.table[hx],x)
}
```

En C :

```
ListeChainee enleveOccurrence(ListeChainee L, int x){
        if (L == NULL) return NULL; //pas nécessaire si on suppose que x est dans L
        if (L->valeur == x) {
                ListeChainee tmp = L->suivant;
                free(L);
                return tmp;
        }
        L->suivant = enleveOccurrence(L->suivant,x);
        return L;
}

void supprime(struct Ensemble E, int x){
        int hx = (15073*x) % E.taille;
        E.table[hx] = enleveOccurrence(E.table[hx],x);
}
```

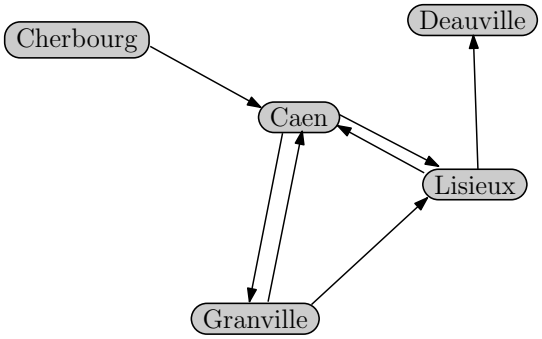
Note : Il est possible de faire une version itérative, mais elle est plus délicate car il faut a priori séparer le cas où E.table[$h(x)$] pointe directement vers un noeud avec x comme valeur et le cas contraire. Voir cette correction en C pour la version itérative :

```
void supprime2(struct Ensemble E, int x){

        int hx = (15073*x) % E.taille;

        ListeChainee l = E.table[hx];

        if (l->valeur == x){
                E.table[hx] = l->suivant;
                free(l);
        }
        else{

                while(l->suivant->valeur != x){
                        l = l->suivant;
                }

                ListeChainee aSupprimer = l->suivant;
                l->suivant = l->suivant->suivant;
                free(aSupprimer);

        }
}
```

Structures des graphes

EXERCICE 2 – STRUCTURES DE DONNÉES À TRAVERS UN EXEMPLE

Soit G le graphe suivant :



Q1. Écrivez le tableau listant les nom des sommets triés selon l'ordre alphabétique. Puis représentez le graphe G sous forme de matrice d'adjacence. Pourquoi faut-il garder le tableau contenant les noms des sommets ? (Ne cherchez pas très loin pour la dernière question.)

Correction. Tableau : `["Caen","Cherbourg","Deauville","Granville","Lisieux"]`
Matrice d'adjacence :

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

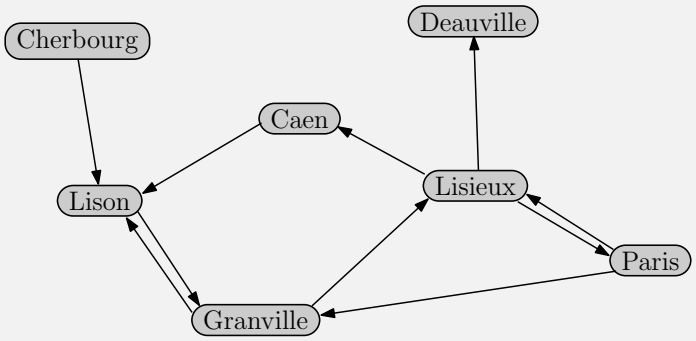
On garde le tableau des noms des sommets pour pouvoir retrouver à quelles villes correspondent les lignes et les colonnes de la matrice.

Q2. En prenant comme fonction de hachage :

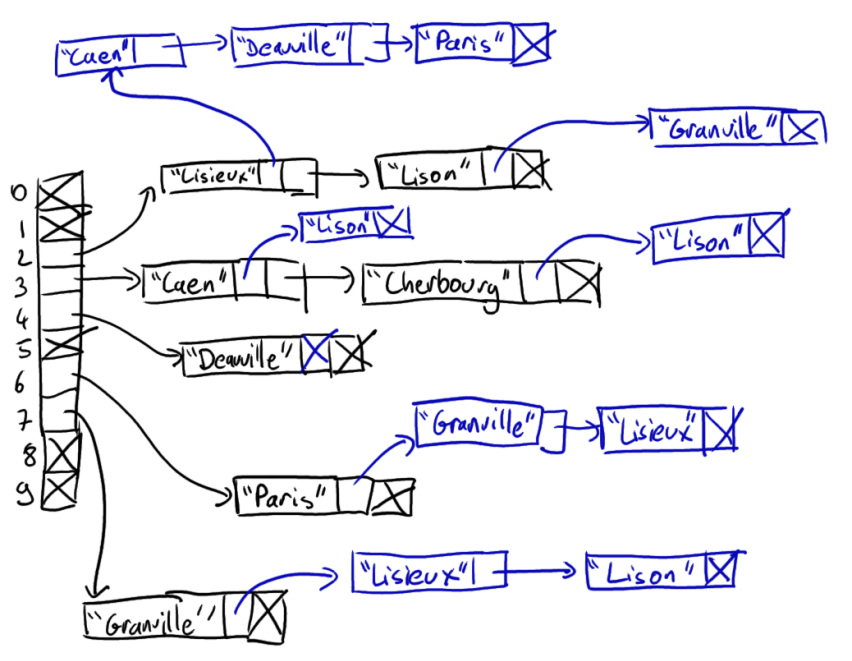
$$\text{chaine de caracteres} \mapsto \text{position de la première lettre dans l'alphabet modulo } 10$$

représentez le graphe comme un dictionnaire (table de hachage) où les clés sont le nom des villes et les valeurs les listes des successeurs des sommets sous forme de listes (simplement) chaînées.

Correction.
A l'origine, je l'avais fait pour ce graphe :



Voici la correction pour le graphe ci-dessus :



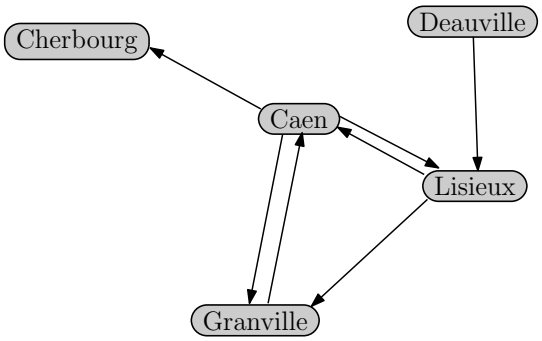
J'ai un peu la flemme de refaire la correction pour le graphe que j'ai rapetissé pour que ça prenne moins de temps en TD. Normalement, si vous avez compris la transformation du graphe ci-dessus en la table de hachage ci-dessus, il ne devrait pas y avoir de problème pour que vous trouviez la correction tout seul !

A noter qu'en pratique, une structure de graphes plus courante serait de stocker les listes des successeurs non pas sous forme de listes chaînées, mais sous forme de tableaux dynamiques. Mais on n'a toujours pas vu ce que c'était ! (programme du dernier CM ?)

EXERCICE 3 – RETOUR À L'ENVOYEUR

On veut écrire une fonction qui étant donné un graphe orienté G retourne le graphe miroir ; c'est-à-dire le graphe obtenu à partir de G en retournant le sens de chacune des arêtes.

Par exemple, pour le graphe de l'exercice précédent, on souhaite renvoyer



Q1. Écrire un algorithme qui prend un graphe M sous forme de matrice d'adjacence et qui renvoie le graphe miroir de M. Quelle est la complexité de cet algorithme ?

Correction.

```
Fonction Graphe_Miroir(M : matrice d'adjacence)
        n = taille(M)
        Miroir = matrice de taille n fois n remplie de 0
        Pour i allant de 1 à n
        Faire   Pour j allant de 1 à n
                Faire   Si (M[i][j] == 1)
                        Alors Miroir[j][i] = 1
        Renvoyer Miroir
```

La complexité de l'algorithme est en $O(|S|^2)$.

Q2. Maintenant on suppose qu'un graphe est un dictionnaire où les clefs sont les noms des sommets et les valeurs sont les listes des successeurs sous formes de listes.

Écrire un algorithme qui prend un tel graphe G et qui renvoie le graphe miroir de G. Quelle est la complexité de cet algorithme ?

Évidemment, dans le pseudo-code, on peut parcourir les clefs d'un dictionnaire, ainsi que les valeurs dans une liste, avec une boucle "Pour".

Correction.

```
Fonction Graphe_Miroir(G : dictionnaire)

        Miroir = dictionnaire vide

        Pour toute clef s dans G
        Faire   Miroir[s] = liste vide

        Pour toute clef s dans G
        Faire   Pour chaque successeur t de s
                Faire   Ajouter s à la liste Miroir[t]

        Renvoyer Miroir
```

La complexité de l'algorithme est en $O(|S| + |A|)$. Pourquoi ? Clairement la première boucle est en $O(|S|)$; pour la seconde, elle est de complexité

$$\sum_{sommets} O(\text{nombre de successeurs de } s)$$

ce qui est bien égal grâce à la formule des poignées de mains à $O(|A|)$.
A noter que la première boucle est nécessaire car il peut y avoir des sommets isolés (par exemple).

Complexité chez les graphes

EXERCICE 4 – UNE FONCTION MYSTÈRE

Voici une fonction mystère sur un graphe non orienté G et un sommet s appartenant à G :

```
Fonction Mystère_mystérieux(G : graphe, s : sommet){
        Pour tout sommet u de G
        Faire Colorier u en blanc

        Colorier s en noir

        P = pile contenant uniquement s
        nb = 1

        Tant que P n'est pas vide
        Faire   t = sommet de P
                Dépiler P
                Pour tout voisin v de t
                Faire   Si v est colorié en blanc
                        Alors   Empiler v dans P
                                Colorier v en noir
                                nb = nb + 1

        Renvoyer nb
```

Q1. Que fait la fonction mystère ? (On ne demande pas de justifier.)

Correction. Elle compte le nombre de sommets se situant dans la composante connexe de G.

Q2. Quelle est la complexité de la fonction mysère ?

Correction. Un sommet v ne peut être ajouté qu'une seule fois dans la pile P. En effet, pour être ajouté dans P, il faut être colorié en blanc ; mais une fois ajouté dans P, on est colorié en noir. On ne peut donc pas être ajouté deux fois dans P.
On en déduit que le nombre d'itérations de la boucle "Tant que" est au plus égal au nombre de sommets.
De plus, chaque passage de la boucle est proportionnel au degré de t (plus un grand $O(1)$ pour les opérations sur les piles).
Donc la complexité de la fonction est borné par

$$O\left(\sum_{\text{sommet } t}(deg(t)+O(1))\right) = O\left(\sum_{\text{sommet } t} deg(t)\right) + O(|S|) = O(|A|+|S|)$$

où la dernière égalité provient de la formule des poignées de mains. (On rappelle que $\sum_{sommets} deg(s)$ est égal à deux fois le nombre d'arêtes.)

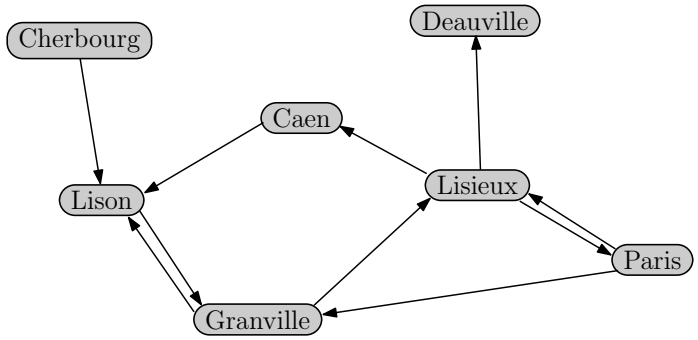
Q3. A-t-on procédé à un parcours en largeur ? en profondeur ?

Correction. Non, il s'agit d'un parcours hybride (ni largeur, ni profondeur).

EXERCICE 5 – CHEMIN LE PLUS COURT ENTRE DEUX POINTS

Écrire une fonction qui prend en paramètres un graphe orienté G, deux sommets s et t de ce graphe. Cette fonction doit renvoyer la longueur du chemin le plus court de s vers t.

Ici la longueur d'un chemin est le nombre d'arêtes compris dans ce chemin. Par exemple, pour le graphe



La longueur du chemin le plus court de Paris vers Deauville est 2.

La complexité de votre algorithme doit être en $O(|S| + |A|)$. Prouvez que votre algorithme a cette complexité.

Indication : Parcours en largeur.

Correction.

```
Fonction DistanceEntre(G : graphe, s : sommet, t : sommet)
        F = file contenant le couple (s,0)
        Tant que F n'est pas vide
        Faire   (u,nb) = tête de F
                Défiler F
                Si (u == t)
                Alors   Renvoyer nb
                Sinon   Pour tout voisin u de v
                        Faire   Enfiler dans F le couple (v,nb+1)
        Renvoyer +infini /* sommet t non accessible depuis s */
```

Il y a autant de passages de boucle "Tant que" que de sommets étant été enfilés dans la pile. De plus, la complexité d'un passage de cette boucle est proportionnel au degré sortant de u (plus un grand $O(1)$ pour les opérations sur les files).
Donc la complexité de la fonction est borné par

$$O\left(\sum_{\text{sommet } u}(deg(u) + O(1))\right) = O\left(\sum_{\text{sommet } u} deg(u)\right) + O(|S|) = O(|A| + |S|)$$

où la dernière égalité provient de la formule des poignées de mains dans un graphe orienté.