# Examen - 1ère session

Durée : 1H30

*Seuls documents autorisés : cours et notes de cours*
*– Barême indicatif –*

## Exercice 1 (4 points) – AVL et table de hachage

Dans cet exercice, on s'intéresse à la manipulation d'entiers. Soit la séquence d'entiers suivante **3**, **10**, **2**, **16**, **8**, **6**, **1**.

**Q 1.1** En suivant l'ordre de la séquence, insérer les éléments un par un dans un AVL initialement vide. Donner l'arbre obtenu après chaque insertion, ainsi que les éventuelles rotations réalisées.

**Q 1.2** On souhaite maintenant stocker les entiers dans une table de hachage avec gestion des collisions par chaînage. On considère une table de hachage de 4 cases, initialement vide, dont la fonction de hachage est $h(x) = x \bmod 4$. Représenter graphiquement la structure obtenue en insérant dans la table les entiers de la séquence un par un, dans l'ordre donné par la séquence.

**Q 1.3** On s'intéresse au test d'existence d'un élément. Supposons que la séquence des nombres à stocker se limite aux entiers de 1 à 8. Quelle est la structure la mieux adaptée pour effectuer ce test (parmi l'AVL et la table de hachage) ? Justifiez votre réponse. Même question si les entiers considérés vont de 1 à 32.

**Q 1.4** Existe-t-il une structure de données plus efficace pour effectuer ce test, quand on sait que les entiers à stocker sont entre 1 et $m$, avec $m$ pas très grand ? Justifier votre réponse et proposer éventuellement une nouvelle structure.

## Exercice 2 (4 points) – Code de Huffman

Quand il s'agit de transmettre un message, il est intéressant de minimiser la taille de la représentation de ce message : c'est le problème de la compression de données. Le code de Huffman (1952) est un code permettant d'obtenir des messages dont la longueur moyenne est minimale.

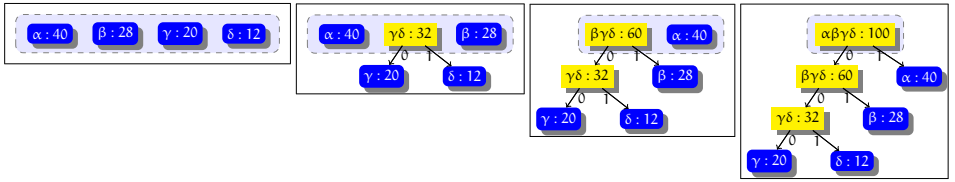
Le principe de base est de décider de coder chaque caractère en utilisant un code de longueur variable : plus un caractère est fréquemment utilisé dans un message, plus la longueur du code utilisé pour représenter ce caractère est compacte. Par exemple, en français, le 'e' est plus fréquent que le 'w' et devrait donc avoir un codage de plus petite taille. On observe ainsi des réductions de taille de l'ordre de 20 à 90%. Ce code est largement utilisé, souvent combiné avec d'autres méthodes de compression.

Connaissant la liste des caractères du message et la fréquence de chaque caractère, l'algorithme de construction du code consiste donc à déterminer quel code utiliser pour chaque caractère d'un message. Le code obtenu pour chaque caractère est un codage binaire. Par exemple, 0 pour le 'e', 1 pour le 'a', 01 pour le 't', 10 pour le 's', 100 pour le 'i',...
Cet algorithme repose sur la construction d'un arbre binaire où les feuilles correspondent chacune à un des caractères distincts du message. Un tel arbre est obtenu selon l'algorithme suivant :



- On part d'un ensemble d'arbres limités à une feuille : l'unique nœud de chaque arbre porte un des caractères du message et la fréquence du caractère.
- À chaque itération de l'algorithme :
  - choisir 2 arbres $A_1$ et $A_2$ dont la racine possède une fréquence minimale
  - on remplace ces 2 arbres par un nouvel arbre obtenu en créant une racine $r$ dont les fils sont $A_1$ et $A_2$. Par convention, un fils gauche est toujours de poids le plus fort que le fils droit. De plus, $r$ porte une fréquence qui est la somme des fréquences des racines de $A_1$ et $A_2$.

**Exemple :** En supposant un alphabet $(\alpha, \beta, \gamma, \delta)$ dont les fréquences sont $(40, 28, 20, 12)$, voici les 4 itérations de l'algorithme :




Une fois obtenu l'arbre de Huffman, on connait alors le codage de chaque caractère : pour chaque caractère, on détermine le chemin allant la racine à la feuille correspondante, si on a utilisé un fils gauche, on ajoute le code 0, si on a utilisé un fils droit, on ajoute le code 1. Dans l'exemple : $\alpha = [1]$, $\delta = [001]$, etc.

**Q 2.1** En utilisant la table de fréquence ci-contre, construire l'arbre du code de Huffman pour cette alphabet.

| a | b | c | d | e | f | g | h | i | j |
|---|---|---|---|---|---|---|---|---|---|
| 18 | 3 | 7 | 15 | 23 | 6 | 5 | 4 | 16 | 3 |

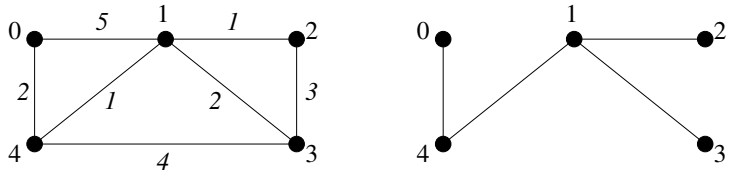
**Q 2.2** En se souvenant que la construction d'un arbre binaire à partir de ses feuilles est en $O(1)$, quelle est la partie la plus coûteuse dans l'algorithme de construction de l'arbre du code ? Quelle structure de données vous parait la plus pertinente (justifier sa réponse) ?

**Q 2.3** Dans l'algorithme de codage d'un message en utilisant un tel code, quelle structure de données faut-il utiliser pour effectuer cette opération rapidement ? La construire pour le code de la question 1.

## Exercice 3 (12 points) – Problème de graphe

Dans cet exercice, on considère un graphe non-orienté $G = (S, A)$ où $S$ est un ensemble de $n$ sommets numérotés de 0 à $n-1$ et $A$ est un ensemble d'arêtes. Chaque arête $(u, v)$ est ici valuée par une valeur réelle positive $val(u, v)$. L'objectif de cet exercice est d'implémenter un algorithme, dit de Prim, pour rechercher un arbre couvrant minimum, c'est-à-dire un arbre couvrant de $G$ de valeur minimale parmi tous les arbres couvrants de $G$ possibles. On rappelle qu'un arbre couvrant de $G$ est un sous-graphe de $G$ qui ne contient pas de cycle et qui relie toute paire de noeuds par une chaîne.

Pour illustrer le problème, considérons les graphes suivants :

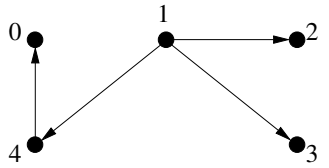



Le graphe de gauche correspond à un graphe non-orienté G à 5 sommets. Les valeurs $val$ des arêtes sont indiquées à côté de chaque arête. Le graphe de droite correspond à un arbre couvrant de G. On peut en effet voir qu'il s'agit bien d'un arbre couvrant car il ne contient pas de cycle et il relie tous les sommets de G (en utilisant des arêtes de G). De plus, on peut se rendre compte que c'est un arbre couvrant minimum, c'est-à-dire que la somme des ses arêtes est minimale parmi tous les arbres couvrants de G.
Pour implémenter le graphe G, nous allons utiliser la structure de données suivante, dite structure par liste d'adjacence.

```
1  typedef struct arete{
2     int vois;   /* voisin du sommet */
3     float val;  /* valeur de l'arete */
4     struct arete* suiv;
5  } Arete;
6
7  typedef Arete* Liste_arete;
8
9  typedef struct{
10    int nbsom;     /* Nombre de sommets dans le graphe */
11    Liste_arete* TabSom;  /* vecteur des listes d'adjacences des sommets */
12 } Graphe;
```

**Q 3.1** Donner la fonction `float val_arete(Graphe *G, int u, int v);` qui renvoie la valeur $val(u,v)$ de l'arête $(u,v)$ si l'arête existe et -1 si elle n'existe pas.

Pour implémenter un arbre couvrant, on va utiliser la remarque suivante : un arbre peut aussi être vu comme une arborescence avec un sommet de départ $r$ quelconque. On rappelle qu'une *arborescence* enracinée en $r$ est un arbre orienté de manière à ce que les arcs forment des chemins allant de $r$ à tous les autres sommets. Dans l'exemple de la figure précédente, si l'on choisit le sommet 1 comme sommet de départ, on peut lire l'arbre dessiné à droite comme l'arborescence donnée par le dessin suivant :




Grâce à la remarque précédente, une arborescence peut ainsi être décrite par le tableau d'entiers `ACM` tel que `ACM[u]` indique le père du sommet u dans l'arborescence et `ACM[r]=-1` pour le sommet $r$ de départ.

**Q 3.2** Donner la fonction `float valeurACM(Graphe *G, int *ACM);` qui calcule et renvoie la valeur d'un arbre couvrant stocké dans le tableau `ACM`.

L'algorithme de Prim peut être présenté de la façon suivante.



**Algorithme de Prim** pour le graphe G et un sommet de départ $r$ :

*(Initialisation)*
- $marque[u] = 0$ pour tout sommet $u$
- $marque[r] = 1$
- $valcour[u] = +\infty$ pour tout sommet $u$
- $ACM[r] = -1$
- Pour tout sommet $v$ voisin de $r$
  - $valcour[v] = val(r,v)$
  - $ACM[v] = r$

*(Fin Initialisation)* Fin Pour

*(Boucle Principale)* Pour $k$ de 1 à $n-1$ Faire
- Choisir un sommet $u$ non marqué de plus petite valeur $valcour$
- $marque[u] = 1$

*(Examiner_sommet)* Pour tout sommet $v$ voisin de $u$ non marqué Faire
- Si $valcour[v] > val(u,v)$ Alors
  - $valcour[v] = val(u,v)$
  - $ACM[v] = u$
- Fin Si

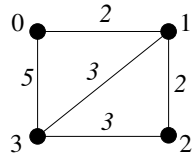
*(Fin Examiner_sommet)* Fin Pour

Fin Pour

Dans ce pseudo-code, on utilise :

- un vecteur *ACM* qui va peu à peu contenir un arbre couvrant et qui correspondra à la fin à l'arbre couvrant minimal.
- un vecteur *valcour* qui va stocker pour chaque sommet la valeur d'une arête incidente.
- un vecteur *marque* qui va contenir 1 si le sommet a été ajouté à l'arbre et 0 sinon.

**Q 3.3** Donner la trace d'exécution de l'algorithme de Prim à partir du graphe de la figure suivante et du sommet de départ 0. A chaque itération de la boucle principale, vous indiquerez les valeurs des tableaux `marque`, `valcour` et `ACM`. Dessiner également l'arbre couvrant minimal obtenu.




Pour implémenter l'algorithme de Prim, on va créer une fonction permettant de réaliser l'étape initialisation et une fonction qui effectue la tâche *examiner sommet*.

**Q 3.4** Donner la fonction `void initialisation(Graphe* G, int r, int** marque, float** valcour, int *ACM);` qui alloue les tableaux `marque` et `valcour` (le tableau `ACM` étant déjà alloué) et effectue les opérations *Initialisation* de l'algorithme de Prim.

**Q 3.5** Donner la fonction `void examiner_sommet(Graphe* G, int u, int* marque, float* valcour, int* ACM);` qui effectue les opérations correspondant à l'étape *Examiner_sommet* de l'algorithme de Prim.

**Q 3.6** Donner la fonction `void Prim(Graphe *G, int r, int *ACM);` qui prend en paramètre un tableau `ACM` alloué à $n$ cases et qui retourne dans `ACM` un arbre couvrant minimum de sommet de départ $r$.

**Q 3.7** Quelle est la complexité de la fonction `Prim` ainsi implémentée ?

On dispose à présent d'une structure de données de type `XX` qui permet de stocker des couples $(u, val)$ où $u$ est un entier entre 0 et $n$ et $val$ est un nombre réel. Cette structure peut en fait uniquement stocker au plus $n$ couples ayant chacun des valeurs $u$ distinctes. Cette structure possède pour fonction de manipulation :

- `void initXX(XX* SD, int n);` qui initialise la structure pour la valeur $n$.
- `void insertXX(XX* SD, int u, float val);` qui permet d'insérer d'un élément en $O(\log(n))$ dans la structure.
- `void extrait_min(XX* SD, int* u, float* val);` qui retourne les valeurs $(u, val)$ de l'élément qui minimise le champs $val$ de la structure, puis qui supprime cet élément : en $O(\log(n))$.
- `int u_dans_XX(XX* SD, int u);` qui retourne vrai si et seulement s'il existe un élément $(u, val)$ dans la structure : en $\Theta(1)$.
- `float champ_val(XX* SD, int u);` qui retourne la valeur $val$ associée à la valeur entière $u$ dans un élément de la structure : en $\Theta(1)$ (attention il doit exister un tel élément).
- `void suppr_u(XX* SD, int u);` qui supprime de la structure un élément $(u, val)$ en $O(\log(n))$ (attention, il doit exister un tel élément).

**Q 3.8** Donner le nom d'une structure de données permettant de réaliser les opérations précédentes avec la complexité indiquée pour une structure XX. Cette structure peut être composée de plusieurs structures de données connues.

**Q 3.9** On se propose d'utiliser la structure XX pour concevoir une meilleure implémentation de l'algorithme de Prim. Préciser quel rôle vous donneriez alors à cette structure. Indiquer sommairement les fonctions de manipulations de XX que vous utiliseriez ainsi que les emplacements correspondants. Est-ce que cette structure permet d'améliorer la complexité de l'algorithme ?