PROG L3-MI
Examen terminal, session 1

2024-12-11

Consignes. *La durée de cet examen est de deux heures. Seule une feuille A4 recto-verso manuscrite et non photocopiée est autorisée. Dans tous les exercices, il est possible d'admettre le résultat d'une question pour répondre aux questions suivantes. Pour l'ensemble des questions, on supposera que les entêtes nécessaires ont été incluses.* ***La mémoire allouée sera libérée lorsqu'elle n'est plus utile.*** *Le barème est donné à titre indicatif.*

Contexte : codage de Huffman

L'objectif de ce devoir est d'implémenter quelques fonctions pour réaliser un codage de Huffman, une méthode de compression sans perte. Le codage de Huffman est un code à longueur variable qui associe un code (binaire) plus court aux symboles les plus fréquents.

Le fonctionnement du codage repose sur la création d'un arbre binaire (voir l'exemple Figure 1). Une feuille de l'arbre représente un symbole (caractère). Chaque noeud de l'arbre est également associé à un poids qui est le nombre d'occurrences du caractère dans le cas des feuilles ou la somme des poids de ses deux fils dans les autres cas.

Pour construire un arbre, on commence avec un ensemble de noeuds constitué uniquement des feuilles. L'arbre est ensuite construit itérativement en retirant, à chaque étape, les deux noeuds de poids les plus faibles[1] et en les associant pour former un nouveau noeud binaire qui est ajouté à l'ensemble. Le processus s'arrête lorsqu'il n'y a plus qu'un seul noeud qui sera la racine de l'arbre.

Pour obtenir le code binaire d'un caractère, il suffit de parcourir l'arbre depuis la racine jusqu'au caractère et de concaténer `0` si on va à gauche, `1` si on va à droite. Sur l'arbre présenté Figure 1, le caractère `'p'` sera associé au code binaire `10011`.

Pour représenter indifféremment un noeud binaire ou une feuille, nous utiliserons la structure suivante :

```
typedef struct HNode {
        char c; // le caractère de la feuille, '\0' sinon
        uint32_t nb; // nb d'occurrences
        struct HNode *left,*right; // NULL si feuille
} HNode;
```

1. en cas d'ex-aequo, on choisira arbitrairement la paire à fusionner

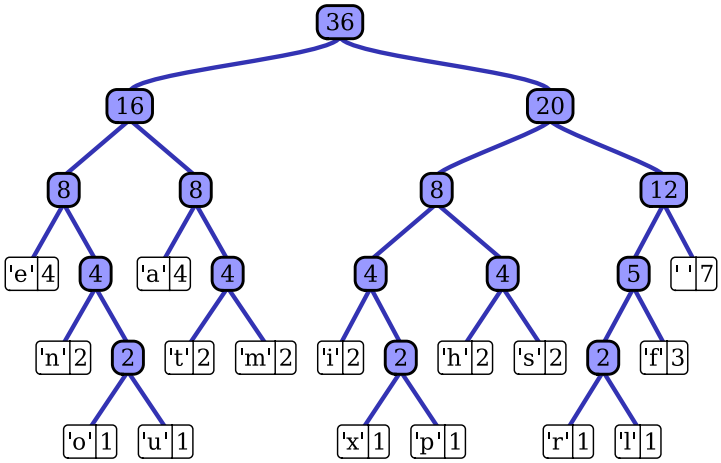


FIGURE 1 – Exemple d'arbre construit à partir du texte "this is an example of a huffman tree "

Initialisation des noeuds (2pts)

Q1 : Implémentez la fonction `HNode* hn_init(const char c, const uint32_t nb)` qui initialise et retourne une feuille à partir des valeurs passées en paramètre.

Q2 : Implémentez la fonction `HNode* hn_merge(HNode* l, HNode* r)` qui initialise et retourne un noeud binaire à partir des deux noeuds passés en paramètre. Les paramètres `l` et `r` pointeront respectivement vers les fils gauche et droit du noeud créé.

Un tas (8pts)

Afin de construire l'arbre, on va avoir besoin de récupérer à chaque itération les deux noeuds avec le plus petit poids (nombre d'occurrences). Pour cela, la structure de tas est particulièrement adaptée.

Pour rappel, un tas est un arbre binaire presque complet généralement représenté par un tableau. Le premier élément du tableau d'indice 0 est la racine de l'arbre. Les deux fils d'un élément à la position i sont les éléments d'indices $2i + 1$ et $2i + 2$. Dans notre cas, nous avons besoin d'un tas ordonné pour que la racine soit un élément minimal (i.e. un `HNode` avec le plus petit nombre d'occurrences). Cela signifie que l'élément à la position $(i - 1)/2$ soit plus petit (ou égal) à celui se trouvant à la position i.

Pour ajouter un élément à un tas, on part de la fin du tableau et on décale successivement les parents tant que l'élément à ajouter est plus petit que le parent considéré. Pour supprimer le plus petit élément (c.-à-d. la racine), il faut placer le dernier élément du tas à la place de racine puis le faire descendre tant qu'il est plus grand qu'au moins un de ses fils.

Pour représenter un tas, nous utiliserons la structure suivante :

```
typedef struct HMinHeap {
        size_t size; // nombre d'éléments
        size_t capacity; // taille de array
        HNode* array[];
} HMinHeap;
```

Q3 : Implémentez la fonction `HMinHeap* hnn_init(size_t capacity)` qui initialise et retourne un `HMinHeap` qui aura la capacité donnée en paramètre.

Q4 : Implémentez la fonction `bool hnn_push(HMinHeap* h, HNode* n)` qui permet d'ajouter le noeud `n` dans le tas `h`. La fonction retournera `true` si l'ajout a pu se faire (il restait de la place), `false` sinon.

Q5 : Quels changements faut-il effectuer sur la structure et la fonction précédente pour disposer d'un `HMinHeap` dont la capacité est dynamique, c.-à-d. que sa capacité augmente en fonction de sa taille ?

Q6 : Implémentez la fonction `HNode* hnn_pop(HMinHeap* h)` qui supprime et retourne le noeud du tas ayant le plus petit nombre d'occurrences.

Construction de l'arbre (6pts)

La construction de l'arbre de codage prend en entrée un tableau `uint32_t occ[NB_CAR]` contenant les nombres d'occurrences de chaque caractère. Les indices du tableau correspondent aux valeurs des caractères, ainsi `occ['a']` représente le nombre d'occurrences du caractère `'a'`.

Q7 : En sachant que la taille en bits d'un `char` est donnée par la directive `CHAR_BIT` du fichier d'entêtes `limits.h`, quelle taille doit faire le tableau `occ` pour pouvoir accueillir les nombres d'occurrences de n'importe quel caractère représentable par un `char` ?

Q8 : Implémentez la fonction `HNode* hn_buildTree(uint32_t occ[NB_CAR])` qui permet de construire l'arbre de Huffmann à partir du tableau des nombres d'occurrences `occ` et qui retourne un pointeur vers la racine de cet arbre. Dans ce tableau, le nombre d'occurrences d'un caractère `c` est à l'indice `occ[(unsigned char) c]`. Seuls les caractères ayant un nombre d'occurrences supérieur à 0 seront considérés.

Q10 : Implémentez la fonction `void hn_destroy(HNode* r)` qui libère la mémoire utilisée par l'arbre dont la racine est pointée par `r`.

Décodage d'un tableau d'octets (4pts)

Pour décoder un flux binaire, il suffit de lire bit à bit le flux et de parcourir l'arbre en fonction de la valeur du bit. Lorsqu'on l'arrive sur une feuille, on a décodé un caractère, et on peut recommencer à partir de la racine de l'arbre pour lire le caractère suivant.

Q10 : Quelle peut être la longueur maximale d'un code binaire issu d'un arbre de Huffman (c.-à-d. la hauteur de l'arbre) ?

Q11 : Implémentez la fonction suivante :
`char* decode(HNode* r, size_t len, uint8_t tab[static len/8])`

Cette fonction retourne la chaîne de caractères décodée à partir du tableau d'octets `tab` et de l'arbre dont la racine est pointée par `r`. Un élément d'indice i du tableau peut être écrit $tab[i] = \sum_{j=0}^{7} tab[i]_j.2^j$. Dans la convention choisie $tab[i]_j$ représente le bit d'indice

$(i * 8 + j)$ du flux. La chaîne de caractères retournée devra se terminer par le caractère `'\0'`.

Annexes

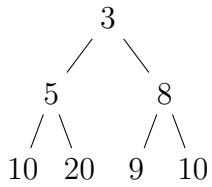
Quelques fonctions :

- `void *malloc(size_t size);` alloue `size` octets sur le tas et retourne un pointeur vers la mémoire allouée.
- `void free(void *ptr);` libère la mémoire pointée par `ptr`.
- `void *calloc(size_t nmemb, size_t size);` alloue `nmemb` × `size` octets sur le tas, l'initialise à 0 et retourne un pointeur vers la mémoire allouée.
- `void *realloc(void *ptr, size_t size);` change la taille du bloc de mémoire pointé par `ptr` pour qu'il soit de taille `size` octets. Le contenu de la mémoire entre le début jusqu'au minimum entre `size` et l'ancienne taille ne sera pas changé.

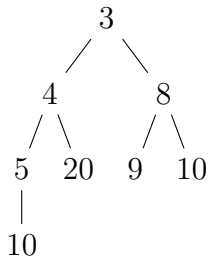
Quelques opérateurs :

- `<<` : informellement, `a << b` renvoie l'entier obtenu en décalant les bits de `a` de `b` positions « vers la gauche » (vers les bits de poids fort), en introduisant `b` zéros dans les bits de droite (de poids faible).
- `>>` : informellement, `a >> b` renvoie l'entier obtenu en décalant les bits de `a` de `b` positions « vers la droite » (vers les bits de poids faible), en introduisant `b` zéros dans les bits de gauche (de poids fort) (en faisant ici l'hypothèse d'une opérande `a` de type non signé).
- `~` : `~a` renvoie le complément binaire de `a`.
- `&` : `a & b` renvoie le ET bit à bit de ses deux opérandes.
- `^` : `a ^ b` renvoie le XOR bit à bit de ses deux opérandes.

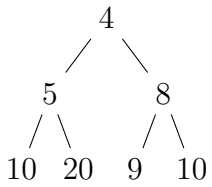
Tas : Soit le min tas $[3, 5, 8, 10, 20, 9, 10]$. Après, l'ajout de l'élément 4, le tas sera $[3, 4, 8, 5, 20, 9, 10, 10]$. Si par la suite on enlève 3 le tas deviendra $[4, 5, 8, 10, 20, 9, 10]$.



Au départ



Après l'ajout de 4



Après la suppression de 3