TP2 Algorithmique des images, textes et données

Arbres binaires et codage de Huffman.

Le but de ce TP est de créer un programme permettant de compresser un texte passé en paramètre à l'aide du codage de Huffman. La première partie du TP concerne la création et la manipulation d'arbres binaires qui seront nécessaires pour implémenter l'algorithme de Huffman dans la deuxième partie.

Attention : tout autre include que `stdlib.h` et `stdio.h` est interdit !

1 Arbres binaires

Un arbre binaire est une structure de données qui peut se représenter sous la forme d'une hiérarchie dont chaque élément est appelé nœud, le nœud initial étant appelé *racine*. Chaque nœud contient une *clé* qui peut être formée d'une où plusieurs valeurs de type différents ou non. Dans un arbre binaire, chaque élément possède au plus deux éléments *fils* au niveau inférieur, habituellement appelés gauche et droit. Du point de vue de ces éléments fils, l'élément dont ils sont issus au niveau supérieur est appelé *père*.

Au niveau le plus élevé il y a donc un nœud racine. Au niveau directement inférieur, il y a au plus deux nœuds fils. En continuant à descendre aux niveaux inférieurs, on peut en avoir quatre, puis huit, seize, etc. c'est-à-dire la suite des puissances de deux. Un nœud n'ayant aucun fils est appelé *feuille*. La plus grande distance entre une feuille et la racine est appelée *profondeur* de l'arbre.

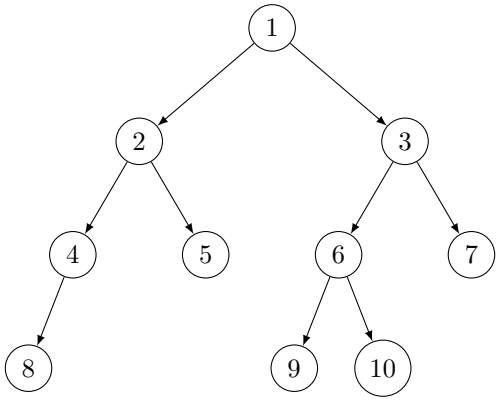


FIGURE 1 – Exemple d'arbre binaire à clé entière, de profondeur 3 avec 5 feuilles.

Dans un premier temps nos arbres binaires auront pour clé des entiers. En `C`, la structure de nœud est définie de la manière suivante :

```
typedef int Tval;
struct node_t
{
  Tval key; // La clef
  struct node_t *left, *right; // Pointeurs vers les fils gauche et droit
};
typedef struct node_t node;
```

1. Implémentez une fonction `node *createNode(Tval key, node *left, node *right)` qui crée un noeud ayant pour clé `key` et ayant pour fils gauche `left` et fils droit `right`.
2. Implémentez une fonction récursive `void freeTree(node *root)` qui libère la mémoire allouée pour l'arbre dont la racine est `root`. Attention votre fonction devra libérer la mémoire de tous les nœuds de l'arbre ! Vous pouvez utiliser l'outil `valgrind` qui vous indiquera si vous avez bien libérée toute la mémoire allouée. Pour l'utiliser taper dans votre terminal :

```
valgrind --leak-check=full ./nom_du_programme
```

3. Créez une fonction `void printTreePre(node *root)` qui affiche les nœuds de l'arbre dont la racine est `root` avec un parcours préfixe. Chaque nœud sera affiché sous le format {`clé nœud, clé fils gauche, clé fils droit`}. Si un des deux fils n'existe pas, on affichera `NULL` au lieu de la clé en correspondante. Par exemple, l'arbre de la Figure 1 sera affiché comme :

```
Affichage prefixe :
{1, 2, 3}
{2, 4, 5}
{4, 8, NULL}
{8, NULL, NULL}
{5, NULL, NULL}
{3, 6, 7}
{6, 9, 10}
{9, NULL, NULL}
{10, NULL, NULL}
{7, NULL, NULL}
```

   Pour rappel un parcours préfixe commence par visiter un nœud, puis son fils gauche puis son fils droit.

4. De manière similaire, créez une fonction `void printTreePost(node *root)` qui affiche les nœuds de l'arbre dont la racine est `root` avec un parcours postfixe. Pour rappel un parcours préfixe commence par visiter le fils gauche d'un nœud, puis son fils droit et enfin le nœud lui même. Par exemple, l'arbre de la Figure 1 sera affiché comme :

```
Affichage postfixe :
{8, NULL, NULL}
{4, 8, NULL}
{5, NULL, NULL}
{2, 4, 5}
{9, NULL, NULL}
{10, NULL, NULL}
{6, 9, 10}
{7, NULL, NULL}
{3, 6, 7}
{1, 2, 3}
```

5. Implémentez une fonction `unsigned int nbLeafs(node *root)` qui renvoit le nombre de feuilles de l'arbre dont la racine est `root`.
6. Implémentez une fonction `unsigned int depth(node *root)` qui renvoit la profondeur de l'arbre dont la racine est `root`.

Chaque nœud peut être atteint depuis la racine par un chemin unique. Ce chemin peut être encodé par une suite de 0 (fils gauche) et de 1 (fils droit) en partant de la racine. Ainsi sur l'arbre de la Figure 1, le nœud de clé 5 peut être identifé par le code : 01, tandis que le nœud de clé 9 aura pour code : 100. La racine quand à elle aura un code vide, c'est à dire de longeur nulle.

7. Créer une fonction `int getCode(Tval key, node *root, char code[], unsigned int *n);` permettant d'obtenir le code du chemin menant au nœud d'une clé `key` dans l'arbre dont la racine est `root`. La fonction renverra 1, si la clé `key` a été trouvée et 0 sinon. Lorsque la clé sera trouvée son code sera stocké dans le tableau `code` et `n` pointera vers la longueur du code.

2 Codage de Huffman

Le principe du codage de Huffman repose sur la création d'un arbre binaire. Il se déroule en 4 étapes :

— On calcule le nombre d'occurence `occ` de chaque caractère `ch` dans le texte. On obtient donc un tableau de couple (`ch`, `occ`) que l'on peut trier par ordre croissant sur les occurences. Les caractères non-alphabétiques sont aussi comptabilisés : espace, retour à la ligne, etc...

— On construit l'arbre de Huffman associé au tableau. Il s'agit d'un arbre binaire dont les feuilles sont les caractères `ch` pondérés par `occ`. La construction de l'arbre est détaillée ci-dessous.

   Tant que le tableau contient plus d'un élément :

   — On prend comme noeud les deux éléments d'occurences les plus faibles.

   — On crée un noeud-père de ces deux noeuds dont la valeur `occ` est la somme des occurences de ses deux fils et on attribue une valeur arbitraire à `ch`, par exemple `'\0'`.

— On insère ce couple dans le tableau sans modifier son tri.

Le dernier couple (`'\0',occ`) du tableau correspond à la racine de l'arbre.

— On calcule le symbole de chaque caractère à partir de l'arbre de Huffman. En partant de la racine on attribue un 0 et un 1 respectivement aux fils gauche et droit, et ce en descendant jusqu'aux feuilles de l'arbre. Le code de chaque caractère est ensuite donné par la suite de '0' et '1' obtenue en partant de la racine de l'arbre.

— On compresse le texte en appliquant le symbole correspondant à chaque caractère le composant.

La Figure 2 illustre à l'aide d'un exemple le fonctionnement de cet algorithme.

Dans cette partie, les arbres binaires que nous considérons ont pour clé une structure `couple`, telle qu'elle a été définie dans le TP1.

1. À l'aide de la fonction `printCouple` du TP1, modifier la fonction `printTreePre` de la première partie de sorte à ce qu'elle puisse afficher des nœuds dont les clés sont de type `couple`. Adapter également les fonctions `createNode` et `getCode`.
2. Créer une fonction `void sortOccurencesList(couple *list, unsigned int n)` qui va trier le tableau `list` de `n couple` dans l'ordre croissant sur les occurences.
3. Créer une fonction `node *buildHuffmanTree(couple *list, unsigned int n)` qui renvoit la racine de l'arbre de Huffman correspondant au tableau `list` de `n couple`. On suppose que le tableau `list` est déjà trié par ordre croissant sur les occurrences.

On va considérer le type `code` suivant :

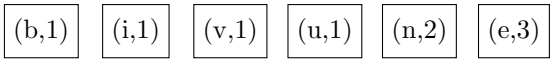
```
struct code_t
{
  char c; // le caractere
  char *code; // le mot de code correspondant
  unsigned int n; // la longueur du mot de code
};
typedef struct code_t code;
```

4. Créer une fonction `code *builCodeTable(couple *list, unsigned int n, node *root)` qui va renvoyer la table des symboles des caractères du tableau `list`, de taille `n`, associéé à l'arbre dont la racine `root` est passée en paramètre.
5. Créer une fonction `char *compressString(char *string, code *codeList, unsigned int n)` qui va substituer les caractères de la chaîne `string` à l'aide de la table des symboles `codeList`, de taille `n`, passée en paramètre et renvoyer le résultat dans une nouvelle chaîne de caractères.
6. Créer un programme qui code le contenu d'un fichier texte passé en paramètre à l'aide du codage de Huffman correspondant. Le programme respectera le format d'affichage donné ci-dessous et ne causera pas de fuite mémoire : vérifiez avec `valgrind`.

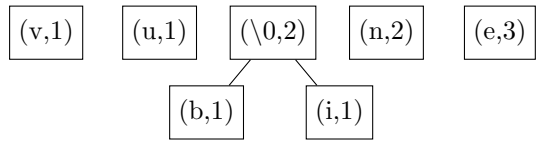
```
./huffman texte.txt

Message a compresser :

bienvenue

Table des symboles :
'v' : 000
'u' : 001
'b' : 010
'i' : 011
'n' : 10
'e' : 11

Message compresse :

0100111110000111000111

Longueur de la compression : 22 bits
Longueur du message initial : 9x7 = 63 bits

Taux de compression : 0.349
```

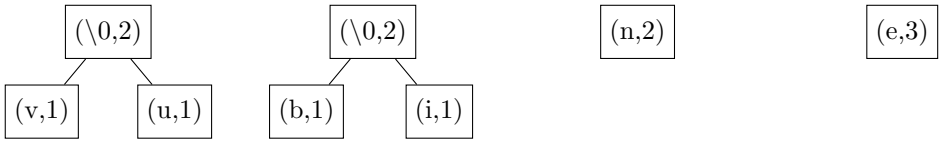
Étape 1 : Tri de la liste des caractères en fonction de leur nombre d'occurences.

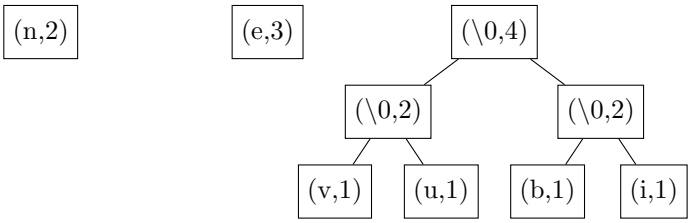
bienvenue

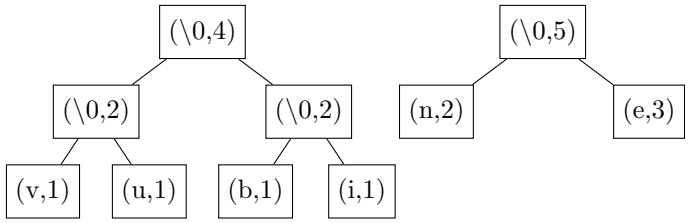


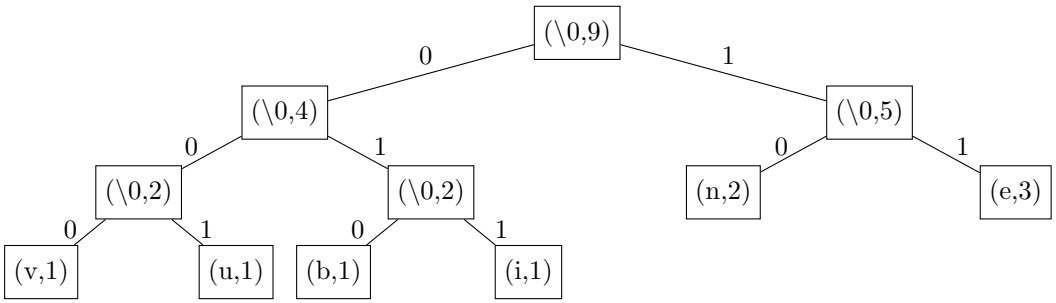
Étape 2 : Construction de l'arbre de Huffman.



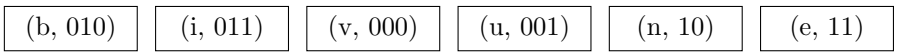








Étape 3 : Construction de la table des symboles.



Étape 4 : Remplacement des caractères par leur symbole.

$$010011111000011100011 \rightarrow 22 \text{ bits au lieu de } 9 \times 7 = 63 \text{ en ASCII simple.}$$

Cela correspond à un taux de compression d'environ 0,349. Évidemment, en pratique, il faudrait aussi prendre en compte la taille de la table des symboles.

FIGURE 2 – Illustration de la compression de Huffman pour le mot "bienvenue"