IN 103 - TD 04

Solution

1 Compression et codage de Huffman

Le codage de Huffman s'appuie sur le fait que dans les fichiers, certains caractères apparaissent plus souvent que d'autres. Par exemple en Français, le 'e' est plus fréquent que le 'w'.

De cette observation, l'idée est d'attribuer aux caractères des codes de longueur variable en fonction de leur fréquence d'apparition. Les caractères les plus fréquents recevront des codes courts ; les caractères plus rares des codes plus longs.

La compression consiste alors à remplacer les caractères du fichier d'origine par les codes leur correspondant, puis d'assembler les séquences de bits en octets qui seront finalement écrits sur disque.

Il est possible de se baser sur des tables de fréquences en fonction de la langue du fichier encoder, mais ceci est peu satisfaisant du fait que l'on n'encode pas que du texte et qu'il n'est pas forcément facile de déterminer ladite langue.

Dans le cas de fichiers sur disque, donc dont le contenu est entièrement disponible (pas comme dans le cas d'un flux continu) une meilleure solution existe : lire le fichier et construire cette table de fréquences. À la place d'une réelle fréquence, on préférera une mesure ≪inverse≫ : le nombre d'occurrences (ça évite des divisions et permet de n'utiliser que des entiers).

Q1 On souhaite écrire une fonction qui prenne en entrée un nom de fichier, construise puis retourne une table d'occurrences. Les octets étant sur 8 bits, la table comportera forcément ... 256 entrées.

— Comment allez-vous allouer cette table ?

— Écrivez cette fonction.

Pour lire le fichier caractère par caractère, vous pourrez au choix utiliser `scanf ()` ou la fonction `fgetc ()`. Cette dernière prend en argument le descripteur de fichier où lire et retourne un **int** contenant le code du caractère lu ou la valeur `EOF` pour signifier l'atteinte de fin de fichier.

Solution

Vu que la table doit être retournée par la fonction, il faut qu'elle survive à son appel et devra donc être allouée dynamiquement.

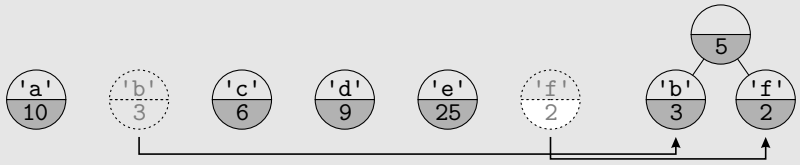
freq.c

```
#include <stdio.h>
#include <stdlib.h>


/** \brief Calcul du nombre d'occurrences par caractères dans le contenu d'un
    fichier. La table obtenue est de taille 256, avec une entrée par caractère,
    et indexée sur le code ASCII des caractères.
    \param fname : Nom du fichier à analyser.
```

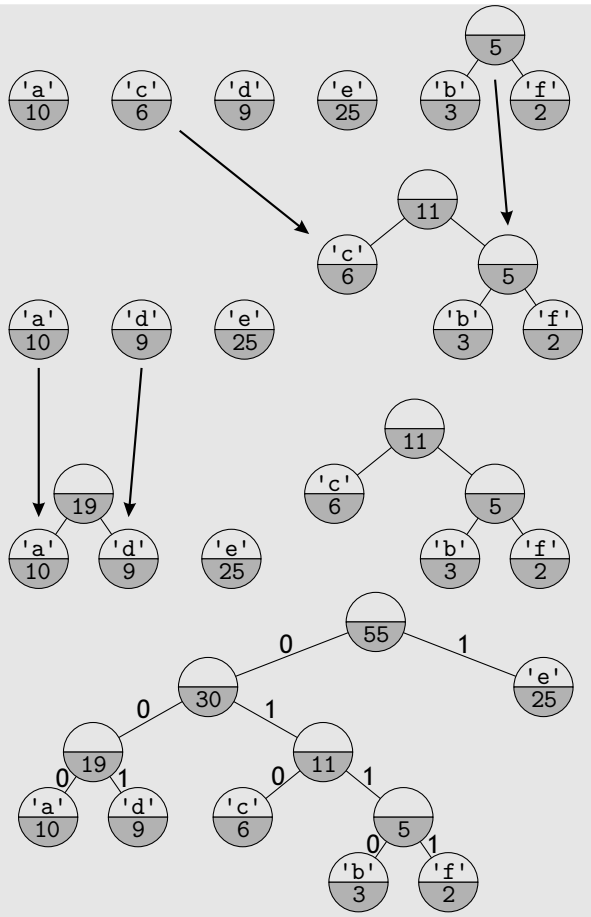
```
    \return Table du nombre d'occurrence ce chaque caractère dans le fichier
    analysé ou NULL si une erreur s'est produite. La libération de cette table
    est à faire par appel à un simple free de la stdlib C. */
unsigned int* get_occs_table (char *fname)
{
  unsigned int *tbl ;
  int c ;

  FILE *in = fopen (fname, "rb") ;
  if (in == NULL) {
    printf ("Error. Unable to open file '%s'.\n", fname) ;
    return (NULL) ;
  }

  /* Allocation avec mise à zéro de chaque octet directement. Ca évite d'écrire
     soit même la boucle qui s'en charge. */
  tbl = calloc (256, sizeof (unsigned int)) ;
  if (tbl == NULL) {
    fclose (in) ;
    return (NULL) ;
  }

  c = fgetc (in) ;
  while (c != EOF) {
    tbl[c]++ ;
    c = fgetc (in) ;
  }

  fclose (in) ;
  return (tbl) ;
}
```

Le calcul du code de Huffman de chaque lettre est réalisé en construisant un arbre. On commence par construire toutes les feuilles, c'est-à-dire une feuille par caractère de la table.

Puis, de manière itérative on choisit les deux arbres les plus rares (donc ayant une occurrence minimale) et on les fusionne en un nouvel arbre dont l'occurrence est la somme de celles de ses 2 fils.



On fusionne ainsi tous les arbres jusqu'à n'avoir plus qu'une racine. C'est l'arbre de Huffman.



Le code correspondant à un caractère correspond au parcours qu'il faut effectuer dans l'arbre pour atteindre la feuille contenant ce caractère : descendre à gauche correspond à un 0, descendre à droite à un 1. Dans l'arbre ci-dessus on obtient donc : `a=000`, `d=001`, `c=010`, `b=0110`, `f=0111`, `e=1`.

Q2 Écrivez la structure de données représentant un arbre de Huffman.

Solution

```
    ------------- hufftree.h -------------

#ifndef __HUFFTREE_H__
#define __HUFFTREE_H__

struct huff_node_t {
  unsigned char letter ;
  unsigned int nb_occs ;
  struct huff_node_t *left ;
  struct huff_node_t *right ;
};

void free_tree (struct huff_node_t *tree) ;
struct huff_node_t* build_tree (char *fname) ;
#endif
```

Q3 Écrivez une fonction `build_tree` qui construit l'arbre de Huffman pour un fichier dont le nom est donné en argument. Cette fonction se chargera de faire appel au calcul de fréquences

que vous avez écrit en question **Q1**. Vous pouvez stocker tous les arbres que vous créez dans un tableau de 256 cases qu'il suffit de parcourir pour trouver l'arbre le plus rare.

Sous-questions pour vous guider :

1. Comment saurez-vous que vous avez terminé la fusion de tous les arbres ?
2. Comment, à chaque itération de fusion, allez-vous trouver les 2 arbres de poids minimaux ?
3. Comment allez-vous ≪boucher≫ le trou laissé par l'un des arbres fusionnés ?

Solution

1. Il suffit de garder un compteur `n_trees` initialisé à 256 (le nombre d'arbres initial) que l'on décrémente à chaque fusion. Lorsque ce compteur arrivera à 1 alors on aura fini (il ne restera plus qu'un seul arbre, la racine).
2. Il suffit de considérer que l'arbre d'indice 0 est celui de poids le plus faible et de parcourir la tableau de l'indice 1 à l'indice `n_trees - 1` en regardant si l'on en trouve un de poids plus faible. Si oui on mémorise son indice.
3. Il suffit de remettre l'arbre résultat de la fusion à l'indice d'un des deux arbres fusionnés et de mettre le dernier arbre du tableau à la place l'indice du second. Ainsi, on ≪tasse≫ les arbres dans le tableau.

---------- hufftree.c ----------

```c
#include <stdio.h>
#include <stdlib.h>
#include "freq.h"
#include "hufftree.h"


/** \brief Libération de la mémoire utiliser par un arbre. La fonction procède
    en libérant récursivement les fils gauche et droite avant de libérer le
    noeud courant.
    \param tree : Pointeur sur l'arbre à libérer. */
void free_tree (struct huff_node_t *tree)
{
  if (tree != NULL) {
    free_tree (tree->left) ;
    free_tree (tree->right) ;
    free (tree) ;
  }
}



/** \brief Construction de l'arbre de Huffman associé au fichier dont le nom
    est passé en argument.
    \param fname : Nom du fichier à analyser et pour lequel créer l'arbre.
    \return Le pointeur sur l'arbre créé ou NULL si une erreur s'est produite
    durant le traitement. L'arbre obtenu devra être libéré ultérieurement en
    fin d'utilité. */
struct huff_node_t* build_tree (char *fname)
{
  struct huff_node_t *trees[256] ;
  struct huff_node_t *min_tree1, *min_tree2 ;
  int i, n_trees ;
  unsigned int min ;

  /* On commence par récupérer la table des occurrences pour ce fichier. */
  unsigned int *occs_tbl = get_occs_table (fname) ;
  if (occs_tbl == NULL) return (NULL) ;

  /* Création de toutes les feuilles. On les stocke dans un tableau. */
```

```
  for (i = 0; i < 256; i++) {
    trees[i] = malloc (sizeof (struct huff_node_t)) ;
    if (trees[i] == NULL) {
      /* Si plus de mémoire pour les feuilles, on libère celles déjà allouées
         et on libère la table d'occurrences. */
      for (i--;i >= 0; i--) free (trees[i]) ;
      free (occs_tbl) ;
      return (NULL) ;
    }   /* Fin de if (trees[i] == NULL). */
    trees[i]->letter = (char) i ;
    trees[i]->left = NULL ;
    trees[i]->right = NULL ;
    trees[i]->nb_occs = occs_tbl[i] ;
  }     /* Fin de for (i = 0; i < 256; i++). */

  /* Maintenant début le processus itératif d'aggrégation des 2 sous arbres
     d'occurrences minimales. */
  n_trees = 256 ;
  /* On s'arrêtera lorsqu'il ne restera plus qu'un seul sous-arbre. */
  while (n_trees > 1) {
    /* On extrait un premier "min". */
    min = 0 ;
    for (i = 1; i < n_trees; i++) {
      if (trees[i]->nb_occs < trees[min]->nb_occs) min = i ;
    }
    min_tree1 = trees[min] ;
    /* On bouche le trou que va laisser l'aggrégation de ce "min" avec le
       prochain. En effet, on prend 2 arbres pour n'en faire qu'un seul. Il va
       donc en disaraître un. */
    trees[min] = trees[n_trees - 1] ;
    /* On a (enfin, va bientôt effectivement avoir) un arbre de moins. */
    n_trees-- ;

    /* On extrait un second "min". */
    min = 0 ;
    for (i = 1; i < n_trees; i++) {
      if (trees[i]->nb_occs < trees[min]->nb_occs) min = i ;
    }
    min_tree2 = trees[min] ;

    /* On fusionne les 2 "mins" et on replace l'arbre résultat dans le
       tableau. On replace le nouveau sous-arbre directement à la place de
       min. */
    trees[min] = malloc (sizeof (struct huff_node_t)) ;
    if (trees[min] == NULL) {
      /* Plus de mémoire: il faudrait libère ce qu'il reste d'arbres. Pour
         ne pas complexifier l'exercice, on fait l'impasse :$ */
      return (NULL) ;
    }
    /* Le champ lettre n'a pas de signification: on l'ignore. */
    trees[min]->left = min_tree1 ;
    trees[min]->right = min_tree2 ;
    trees[min]->nb_occs = min_tree1->nb_occs + min_tree2->nb_occs ;
  }   /* Fin de while (n_trees > 1). */

  /* Il ne nous reste plus qu'un noeud, c'est la racine de l'arbre. */
  return (trees[0]) ;
}
```

Dans le fichier `code.(c|h)` vous est donnée la fonction :

char** get_codes (**struct** huff_node_t *tree)

qui prend en argument un arbre et retourne la table contenant, pour chaque caractère, sont code sous forme d'une chaîne de caractères '0' ou '1'. Cette table va servir à encoder le fichier initial.

Q4 Écrivez une fonction qui permet d'encoder un texte à l'aide du codage de Huffman. Cette fonction prendra en arguments le nom du fichier à encoder, un nom de fichier dans lequel écrire le résultat et la table des codes issue de l'analyse du premier fichier. Le codage se fait en lisant le fichier d'entrée caractère par caractère et en écrivant la séquence ≪binaire≫ associée dans le fichier de sortie (le fichier de sortie doit contenir des caractères `'0'` et `'1'`).

Dans la réalité, on n'écrit pas le code binaire sous forme de chaînes de caractères de `'0'` et de `'1'` : on travaille directement bit-à-bit sur des octets. Nous faisons donc une ≪simulation≫ de compression pour simplifier le programme en omettant les manipulations de bits.

Solution

Le principe est très simple : pour chaque caractère lu dans le fichier, on va rechercher dans la tableau des codes quelle est la séquence (chaîne de caractères) qui lui correspond et on l'affiche dans le fichier destination. Encore une fois, un caractère étant un simple entier sur 8 bits, on s'en sert directement comme indice dans le tableau.

encode.c

```c
#include <stdio.h>
#include <stdbool.h>


/** \details Encodage du fichier \a in_fname selon la table de codes de Huffman
    \a codes et écriture du résultat dans le fichier \a out_fname.
    \param in_fname : Fichier source à lire et à encoder.
    \param out_fname : Fichier destination dans lequel écrire le résultat de
    l'encodage.
    \param codes : Table des codes de Huffman construite à partir de l'analyse
    du fichier source \a in_fname à encoder.
    \return Booléen disant si l'encodage s'est passé sans erreur. */
bool encode (char *in_fname, char *out_fname, char **codes)
{
  FILE *in, *out ;
  int c ;

  in = fopen (in_fname, "rb") ;
  if (in == NULL) {
    printf ("Error. Unable to open input file '%s'.\n", in_fname) ;
    return (false) ;
  }

  out = fopen (out_fname, "wb") ;
  if (out == NULL) {
    printf ("Error. Unable to open output file '%s'.\n", out_fname) ;
    fclose (in) ;
    return (false) ;
  }

  /* Lecture caractère par caractère du fichier d'entrée. */
  c = fgetc (in) ;
  while (c != EOF) {
    fprintf (out, "%s", codes[c]) ;
    c = fgetc (in) ;
  }

  fclose (in) ;
  fclose (out) ;
  return (true) ;
}
```

Q5 Écrivez un `main` vous permettant de tester votre fonction. Pour vérifier que votre algorithme ≪compresse≫ effectivement, vous pourrez vérifier que le fichier codé a bien une taille

plus petite que le fichier d'origine. Attention, dans le fichier codé chaque bit est écrit sur un caractère (donc un octet). En l'écrivant en binaire directement il serait 8 fois plus petit. Il faut donc vérifier que le fichier codé est moins de 8 fois plus grand que le fichier d'origine.

Q5-si-on-a-le-temps Expliquez en pseudo-code ou oralement l'algorithme de décodage.

1. Quel doit être le prototype de cette fonction ?
2. Quelles sont ses grandes étapes ?

Solution

1. bool decode (**char** ∗in_fname, **char** ∗out_fname, **struct** huff_node_t ∗tree)
   Il lui faut un nom de fichier d'entrée contenant des données codées, un nom de fichier de sortie où écrire le résultat du décodage et l'arbre pour analyser la suite de bits contenue dans le fichier d'entrée.
2. Il faut commencer par ouvrir les fichiers. Tant qu'il y a des ≪bits ≫ dans le fichier d'entrée on en lit un pour avancer dans l'arbre. À la fin on ferme les fichiers.

Dans le fichier `decode.(c|h)` vous est donnée la fonction :
bool decode (**char** ∗in_fname, **char** ∗out_fname, **struct** huff_node_t ∗tree)
qui prend en argument un nom de fichier codé, un nom de fichier de sortie et un arbre de Huffman qui décode le contenu du premier fichier et écrit le résultat dans le second fichier.

Q6 Pour vérifier que votre algorithme de codage fonctionne correctement, décodez ce qu'il a codé. Si tout fonctionne, le fichier avant codage et celui issu du décodage doivent être exactement identiques.

Il vous suffit donc d'appeler la fonction de décodage `decode ()` donnée en lui transmettant l'arbre que vous avez construit pour encoder le fichier initial, ainsi que le nom du fichier codé.

Q7 Écrivez la fonction qui libère la mémoire utilisée par un arbre. Dans combien de cas est-elle nécessaire ? Utilisez-la où elle nécessaire.

Solution

Pour le code source, regardez le code fourni en réponse à la question **Q3**. La fonction `free_tree ()` se trouve au début du fichier.

Il y a besoin d'appeler cette fonction dans 2 cas : à la fin du programme et dans le cas où la construction de la table des codes a échoué. En effet, pour créer cette table, il est nécessaire d'avoir construit l'arbre. Et si cette table n'est pas disponible, alors il est impossible de continuer.