Polytech'Lille — GIS 3 — Structures de Données — Feuille de TP numéro 5

Ce TP se déroule sur deux séances (quatre heures). Les étudiants qui se sentent en difficulté devraient réaliser la section 1, uniquement. Les autres peuvent réaliser la section 3, après s'être assurés qu'ils savent effectivement faire la première partie.

1 Implantation d'un module d'ABR

On veut réaliser un module dédié aux ABR, qui permette de faire fonctionner le programme principal `testABR.c` suivant. Les valeurs sont des entiers. Vous pouvez vous inspirer de l'extrait de code disponible sur le site du cours mais il serait préférable que vous refassiez tout vous-même.

```
#include "ABR.h"
#include <stdio.h>

int main ()
{   struct ABR* racine;
    int x;
    racine = NIL;
    scanf ("%d", &x);
    while (x != -1)
    {   racine = ajouter_ABR (x, racine);
        afficher_ABR (racine);
        scanf ("%d", &x);
    }
    printf ("la hauteur de l'ABR est %d\n", hauteur_ABR (racine));
    printf ("le nombre de noeuds de l'ABR est %d\n",
                                        nombre_noeuds_ABR (racine));
    clear_ABR (racine);
    return 0;
}
```

Question 1. Combien de fichiers doit-on écrire ?

Question 2. On souhaite compiler séparément ce qui peut l'être. Quelles seront les commandes de compilation nécessaires ?

Question 3. Écrire le fichier d'entête. Spécifier la structure de données, dans un commentaire, placé dans le fichier.

Question 4. Écrire le fichier source. Lors des essais, commentez, dans le programme principal, les appels aux fonctions que vous n'avez pas encore réalisées.

Question 5. Écrire une fonction qui permette de visualiser un ABR avec `dot` (voir feuille de TD). Gérer le cas de l'arbre vide et celui de l'arbre réduit à une feuille. Pour les tests, sortir l'affichage de la boucle du programme principal.

2 Expérimentations

Question 6. Modifier `testABR.c` pour qu'il affiche, à chaque nouvel entier enregistré dans l'ABR, le nombre $f(n)$ cumulé de comparaisons effectuées, où n désigne le nombre de nœuds de l'arbre. Afin de faire des analyses, afficher les résultats sur deux colonnes : n et $f(n)$.

On souhaite écrire un programme `gentest.c` qui imprime sur sa sortie standard les entiers de 1 à 16384, terminés par -1, suivant différents ordres. L'idée consiste ensuite à rediriger la sortie de `gentest.c` sur l'entrée de `testABR.c`, afin de générer des ABR de différentes formes. Des exemples sont disponibles sur le site du cours, dans l'archive `exemples-arbres.tar`.

Question 7. Programmer `gentest.c` pour qu'il fabrique un ABR filiforme. Rediriger sortie de `gentest.c` sur l'entrée de `testABR.c`. En utilisant vos connaissances et la fonction `fit` de GNUPLOT, déterminer une approximation de $f(n)$.

Question 8. Programmer `gentest.c` pour qu'il fabrique un ABR équilibré en nombre de nœuds. Chercher une solution récursive.

Question 9. En utilisant vos connaissances et la fonction `fit` de GNUPLOT, déterminer une approximation de $f(n)$. Attention : `gentest.c` doit énumérer les entiers de façon à ce que l'ABR soit en permanence équilibré en nombre de nœuds. Si vous ne trouvez pas de solution récursive, utilisez le code suivant :

```
static void equilibre (int n)  /* n = le nombre d'entiers à imprimer */
{   int w, b, i, k;
    b = n;
    w = 1;
    for (k = n; k >= 1; k /= 2)
    {   for (i = 0; i < w; i++)
            printf ("%d\n", b*(2*i+1));
        b /= 2;
        w *= 2;
    }
}
```

Question 10. Programmer `gentest.c` pour qu'il imprime des entiers aléatoires distincts deux-à-deux. Utiliser `srand48` et `drand48` pour la génération de nombres aléatoires. Voici une suggestion pour initialiser le générateur aléatoire :

```
#include <time.h>
...
    srand48 (time (0));
```

Un résultat théorique affirme que la hauteur moyenne d'un ABR après n insertions d'éléments aléatoires, se comporte asymptotiquement comme $c \ln(n)$, où c vaut approximativement 4.311 [2, Theorem 5.10, page 261]. Ce résultat se vérifie-t-il sur vos expérimentations ?

3 Les codages de Huffman

Le TP porte sur les codages de Huffman, qui constituent une technique de compression de données très utilisée. Dans ce TP, la donnée à compresser est un texte. Du point de vue des structures de données, ce TP nous amène à utiliser des arbres binaires et des files de priorité.

L'idée consiste à coder chaque caractère c d'un texte par une suite de bits, qui dépend du nombre d'occurences de c dans le texte. Plus le caractère est fréquent, plus la suite de bits est courte. Prenons pour exemple, le texte : « exemple de codage de Huffman », formé de 29 caractères. Le codage de Huffman qui lui est associé est présenté sous la forme d'un arbre binaire, Figure 1.

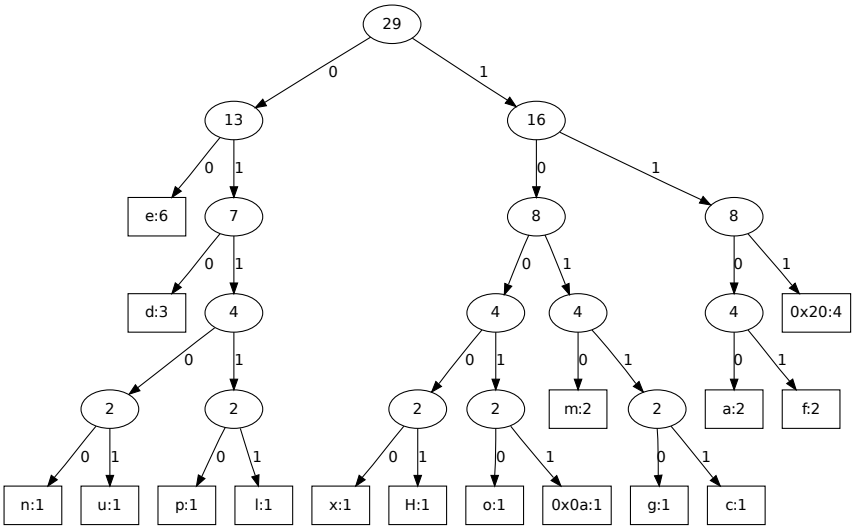

FIGURE 1 – Arbre binaire représentant un codage de Huffman. Chaque feuille est étiquetée par un caractère et son nombre d'occurences. L'espace et le retour chariot sont représentés par leur code ASCII en hexadécimal. Chaque nœud intérieur est étiqueté par le nombre d'occurences total des feuilles du sous-arbre dont il est la racine. Les arcs vers les fils gauches sont étiquetés 0 ; les arcs vers les fils droits sont étiquetés 1.

Pour obtenir la suite de bits qui code un caractère c, il suffit de suivre le chemin qui part de la racine vers la feuille c et d'écrire les étiquettes des arcs suivis. Par exemple, le

caractère « `e` », qui apparaît 6 fois dans le texte, est codé par la suite de deux bits « `00` ». Le caractère « `H` » qui n'apparaît qu'une fois, est codé par la suite de cinq bits « `10001` ». Le texte complet est codé par la concaténation des codages des caractères. Il commence donc par « `001000000` », c'est-à-dire « `exe` ». Au total, la chaîne se code sur 107 bits (14 octets). C'est deux fois plus court que les $7 \times 29 = 203$ bits utilisés par le codage ASCII. Ce codage a de nombreuses propriétés, très intéressantes. Voir [1, chapitre 16.3].

Construction de l'arbre

On se donne une file de priorité F d'arbres de Huffman. Un arbre de Huffman H_1 est plus prioritaire qu'un arbre de Huffman H_2 si le nombre d'occurences qui étiquette la racine de H_1 est *inférieur* à celui de H_2. On lit le texte, caractère par caractère. Pour chaque caractère c, deux cas de figure se présentent : si c est lu pour la première fois, on crée un nouvel arbre de Huffman (une feuille) étiquetée par c et le nombre d'occurences 1, qu'on enfile dans F ; si c a déjà été lu, on incrémente le nombre d'occurences de la feuille qui lui correspond (le caractère est forcément présent dans la file F, sous la forme d'une feuille) et on restructure la file, puisque la priorité de c a baissé.

À la fin de la lecture du texte, on a donc une file de priorité, ne comportant que des feuilles. Pour former l'arbre H, il suffit alors d'appliquer l'algorithme de la Figure 2. Une trace d'exécution est donnée Figure 3.

```
while la file F contient deux arbres ou plus do
  G := défiler (F)
  D := défiler (F)
  N := un nouveau nœud avec fils gauche G, fils droit D (opération de fusion)
  le nombre d'occurences qui étiquette N doit être égal à la somme des
       nombres d'occurences qui étiquettent G et D
  enfiler N dans F
end do
H := défiler (F)
```

FIGURE 2 – Algorithme de construction de l'arbre de Huffman. Initialement, la file de priorité contient les feuilles correspondant aux caractères lus. À la fin, la file ne contient qu'un arbre : l'arbre de Huffman.

Travail à faire

Question 11. Écrire un algorithme qui construise l'arbre de Huffman d'un texte et qui imprime le nombre de bits nécessaire au codage de Huffman de ce texte. Pour cela, mettre au point deux structures de données : une pour les arbres de Huffman et une pour les files de priorité.

Le programme devrait se répartir sur cinq fichiers : deux fichiers d'entête et deux fichiers source correspondant aux deux structures, ainsi qu'un programme principal.

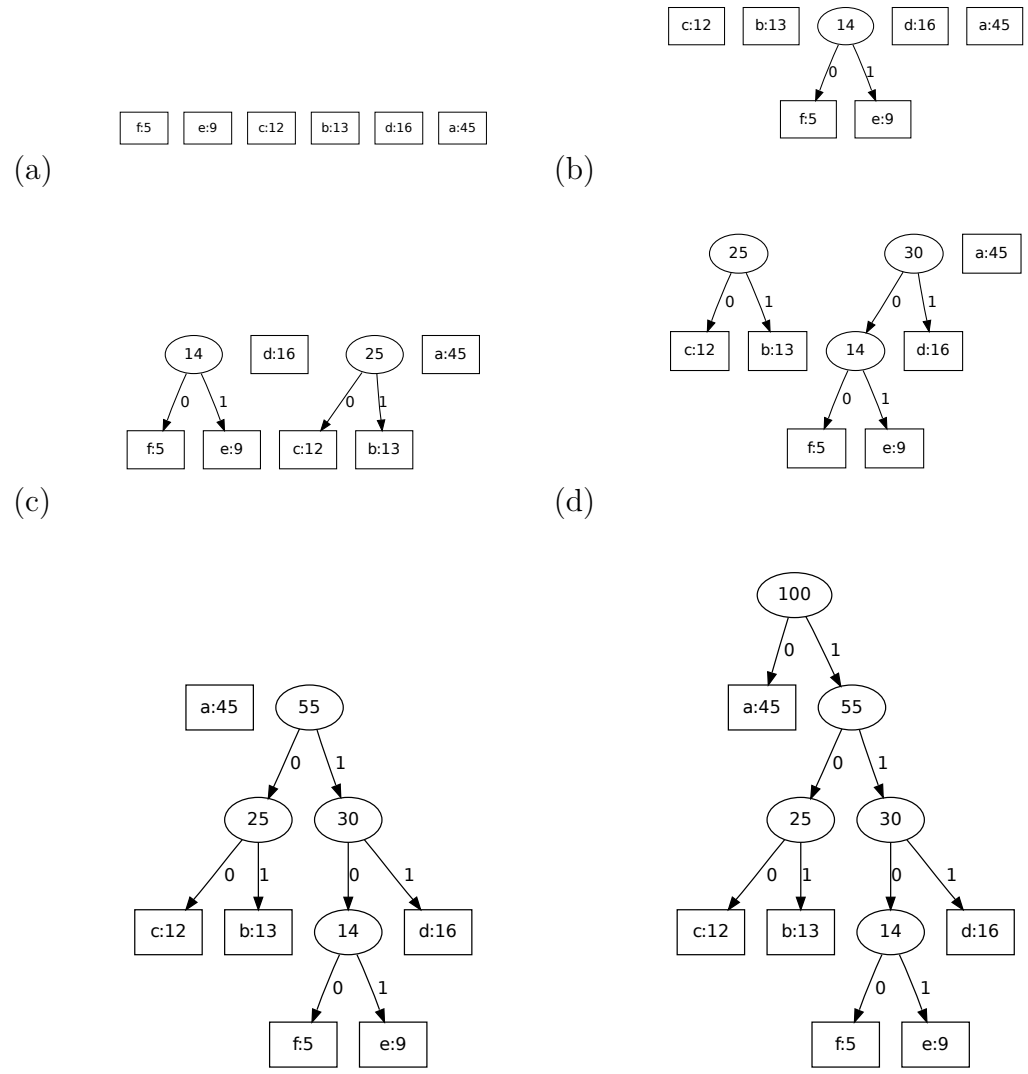

FIGURE 3 – États successifs de la file de priorité, lors de l'exécution de l'algorithme de la Figure 2, sur un exemple. Les éléments de la file sont des arbres. Initialement, la file contient 6 feuilles correspondant aux caractères lus. À la fin, la file ne contient qu'un arbre : l'arbre de Huffman de l'exemple.

Bien spécifier la structure d'arbre de Huffman.

Pour vérifier que l'arbre est correct, il peut être utile de fabriquer un fichier ".dot" en s'inspirant de la Figure 4.

Question 12. Votre programme fini, comparer le taux de compression que vous obtiendriez avec celui de l'utilitaire `gzip`. Que constatez-vous ? Expliquez rapidement pourquoi en effectuant une recherche sur internet.

Question 13. On souhaite maintenant mettre au point une structure de données permettant d'imprimer les suites de bits, correspondant au codage d'un texte, sur la sortie standard. On ne peut imprimer ces séquences de bits que par paquets de huit, sous la forme d'un caractère. Quelle structure de données vous paraît la plus appropriée ? Spécifiez-la.

Question 14. Déterminer le codage d'un caractère dans un arbre de Huffman n'est pas complètement immédiat. Quelle solution proposez-vous ?

```
digraph G {
    label_b965b0 [label="29"];
    label_b96550 [label="13"];
    label_b965b0 -> label_b96550 [label="0"];
    label_b96580 [label="16"];
    label_b965b0 -> label_b96580 [label="1"];
    label_65 [label="e:6"];
    label_b96550 -> label_65 [label="0"];
    label_b964c0 [label="7"];
    label_b96550 -> label_b964c0 [label="1"];
    label_65 [shape=box];
    label_64 [label="d:3"];
    label_b964c0 -> label_64 [label="0"];
    label_b96490 [label="4"];
    label_b964c0 -> label_b96490 [label="1"];
    label_64 [shape=box];
    label_b96340 [label="2"];
    label_b96490 -> label_b96340 [label="0"];
    label_b96370 [label="2"];
    label_b96490 -> label_b96370 [label="1"];
    label_6e [label="n:1"];
    label_b96340 -> label_6e [label="0"];
    label_75 [label="u:1"];
    label_b96340 -> label_75 [label="1"];
    label_6e [shape=box];
    label_75 [shape=box];
    label_70 [label="p:1"];
    label_b96370 -> label_70 [label="0"];
    label_6c [label="l:1"];
    label_b96370 -> label_6c [label="1"];
    label_70 [shape=box];
    label_6c [shape=box];
    label_b964f0 [label="8"];
    label_b96580 -> label_b964f0 [label="0"];
    label_b96520 [label="8"];
    label_b96580 -> label_b96520 [label="1"];
    label_b96400 [label="4"];
    label_b964f0 -> label_b96400 [label="0"];
    label_b96460 [label="4"];
    label_b964f0 -> label_b96460 [label="1"];
    label_b96310 [label="2"];
    label_b96400 -> label_b96310 [label="0"];
    label_b963d0 [label="2"];
    label_b96400 -> label_b963d0 [label="1"];
    label_78 [label="x:1"];
    label_b96310 -> label_78 [label="0"];
    label_48 [label="H:1"];
    label_b96310 -> label_48 [label="1"];
    label_78 [shape=box];
    label_48 [shape=box];
    label_6f [label="o:1"];
    label_b963d0 -> label_6f [label="0"];
    label_0a [label="0x0a:1"];
    label_b963d0 -> label_0a [label="1"];
    label_6f [shape=box];
    label_0a [shape=box];
    label_6d [label="m:2"];
    label_b96460 -> label_6d [label="0"];
    label_b963a0 [label="2"];
    label_b96460 -> label_b963a0 [label="1"];
    label_6d [shape=box];
    label_67 [label="g:1"];
    label_b963a0 -> label_67 [label="0"];
    label_63 [label="c:1"];
    label_b963a0 -> label_63 [label="1"];
    label_67 [shape=box];
    label_63 [shape=box];
    label_b96430 [label="4"];
    label_b96520 -> label_b96430 [label="0"];
    label_20 [label="0x20:4"];
    label_b96520 -> label_20 [label="1"];
    label_61 [label="a:2"];
    label_b96430 -> label_61 [label="0"];
    label_66 [label="f:2"];
    label_b96430 -> label_66 [label="1"];
    label_61 [shape=box];
    label_66 [shape=box];
    label_20 [shape=box];
}
```

FIGURE 4 – Le fichier ".dot" qui a servi à produire la Figure 1. Les identificateurs des feuilles ont été fabriqués à partir des codes ASCII des caractères. Ceux des nœuds intermédiaires à partir des adresses des structures, écrites en hexadécimal.

Compression à la volée

A priori, compresser un texte se fait en deux temps : dans un premier temps, on compte les occurences des caractères ; dans un deuxième temps, on construit le codage et on compresse le texte.

Peut-on compresser un texte à la volée (sans le lire deux fois de suite) avec le codage de Huffman ? Oui. On lit le texte caractère par caractère. Pour chaque caractère c, deux cas de figure se présentent : si c est lu pour la première fois, on l'imprime « en clair » sur la sortie standard, puis on l'enfile dans F ; si c a déjà été lu, on construit l'arbre de Huffman

correspondant à l'état courant de la file F, on imprime la séquence de bits correspondant à c dans cet arbre, puis on met à jour F en incrémentant le nombre d'occurences de c et en abaissant sa priorité. À chaque caractère lu, un nouvel arbre de Huffman est créé.

Comment l'algorithme de décodage fait-il pour distinguer les séquences de bits correspondant à un caractère écrit « en clair » des séquences de bits du codage de Huffman ? Il suffit de se donner un caractère spécial (appelons-le `NYT`) avec un nombre d'occurences 0, qui n'appartient pas au texte[1], de l'enfiler dans F en début d'algorithme, et d'imprimer le codage de Huffman de `NYT` juste avant les huit bits de c.

Pour pouvoir décoder le texte, l'algorithme de décodage est obligé de mimer le comportement de l'algorithme de codage et de faire évoluer, lui aussi, l'arbre de Huffman du texte, à chaque fois qu'un caractère est lu. Il peut ainsi repérer la séquence de bits codant `NYT` et lire les huit bits suivants, qui donnent un nouveau caractère « en clair ».

Une variante de cette méthode est connue sous le nom d'algorithme de Vitter [3].

Question 15. Implantez cet algorithme.

Références

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction à l'algorithmique*. Dunod, Paris, 2ème edition, 2002.

[2] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison Wesley, 1996.

[3] Jeffrey Scott Vitter. Design and Analysis of Dynamic Huffman Codes. *Journal of the ACM*, 34(4) :825–845, 1987.

1. C'est facile en UTF-8.