Université Paris 13 L2
Institut Galilée
Licence 2 Info - 1er semestre
Structure de Données et Algorithmes

Mini-Projet de Structure de Données et Algorithmes 2018

1 Consignes à respecter IMPÉRATIVEMENT

— Le projet est à faire en binômes et appartenant au même groupe de TP. S'il y a plus que deux étudiants dans le groupe, le projet ne sera pas considéré. S'il y a des étudiants que à cause de force majeur, ils désirent faire le projet tout seuls ou le faire avec un(e) camarade d'un autre groupe de TP, ils devront avoir l'aval de leur(s) chargé(e).s de TP. Sans cette aval, le projet ne sera pas considéré.
— Les binômes devront envoyer, par émail, à leur respectif responsable de TP, UN SEUL fichier .c contenant la source du projet et un mini-rapport avec les explications de comme doit faire l'utilisateur pour exécuter le programme. Le fichier .c devra contenir, entre commentaires, le nom et prénom du binôme. Les adresses émail des responsables de TP sont :
  — TP Groupe 1. Resp. : Tito Nguyen. Émail : nltd@nguyentito.eu
  — TP Groupe 2. Resp. : Sophie Toulouse. Émail : sophie.toulouse@lipn.univ-paris13.fr
  — TP Groupe 3. Resp. : Mourad Kmimech. Émail : mkmimech@gmail.com
  — TP Groupe 4. Resp. : Massinissa Hamidi. Émail : hamidi@lipn.univ-paris13.fr
— La date limite d'envoi du projet sera le **vendredi 14 décembre à 23h :59**
— La date de soutenance sera : **date à confirmer ultérieurement**
— Tout projet qui s'avère copié d'un autre et/ou de matériel trouvé sur le web sera sévèrement puni.

2 Problématique

Considérons le problème consistant à déterminer un `codage binaire des caractères` (ou en abrégé `codage`) dans lequel chaque caractère est représenté par une chaîne binaire unique. Par exemple, considérons un fichier de données de 100000 caractères ne contenant que les caractères `a-f`, avec les fréquences indiquées ci-dessous.

	a	b	c	d	e	f
Fréquence (en milliers)	45	13	12	16	9	5
Mot de code de longueur fixe	000	001	010	011	100	101
Mot de code de longueur variable	0	101	100	111	1101	1100

Si l'on utilise un `codage de longueur fixe`, on a besoin de 3 bits pour représenter six caractères : $a = 000, b = 001, \cdots, f = 101$. Cette méthode demande 300000 bits pour coder entièrement le fichier. Si l'on utilise un `codage de longueur variable` dans lequel on attribue aux caractères fréquents les mots de code courts et aux caractères moins fréquents les mots de code longs, on peut faire nettement mieux qu'un codage de longueur fixe. Dans l'exemple ci-dessus, la chaîne 0 sur 1 bit représente ici `a`, et la chaîne 1100 sur 4 bits représente `f`. Ce codage demande $(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000 = 224000$ bits pour représenter le fichier, soit une économie d'environ 25%.

Remarquez que dans le codage à longueur variable dans l'exemple précédent, aucun mot du code n'est aussi préfixe d'un autre mot du code. Ce type de codages sont dits `préfixés`. Les codages préfixes sont souhaitables car ils simplifient l'encodage (la compression) et le décodage. Pour l'encodage, il suffit de concaténer les mots de code représentant chaque caractère du fichier. Dans l'exemple antérieur, avec le codage à longueur variable, on code le fichier de trois caractères `abc` par $0 \cdot 101 \cdot 100 = 0101100$, où "$\cdot$" représente l'opération de concaténation.

Le décodage est aussi très simple avec un codage préfixe. Comme aucun mot de code n'est préfixe d'un autre, le mot de code qui commence un fichier encodé n'est pas ambigu. Il suffit d'identifier le premier mot du code, de le traduire par le caractère initial, de le supprimer du fichier encodé, et de répéter le processus de décodage sur le reste du fichier. Dans notre exemple, la chaîne 001011101 ne peut être interprétée que comme $0 \cdot 0 \cdot 101 \cdot 1101$, ce qui donne `aabe`.

Le processus de décodage demande que le codage préfixe ait une représentation commode, de manière qu'on puisse facilement repérer le mot de code initial. Un arbre binaire dont les feuilles sont les caractères donnés est bien adapté. On interprète le mot de code binaire pour un caractère comme le chemin allant de la racine à ce caractère, où 0 signifie "bifurquer vers le fils gauche" et 1 signifie "bifurquer vers le fils droit". La figure 2 montre l'arbre pour le codage à longueur variable de notre exemple.

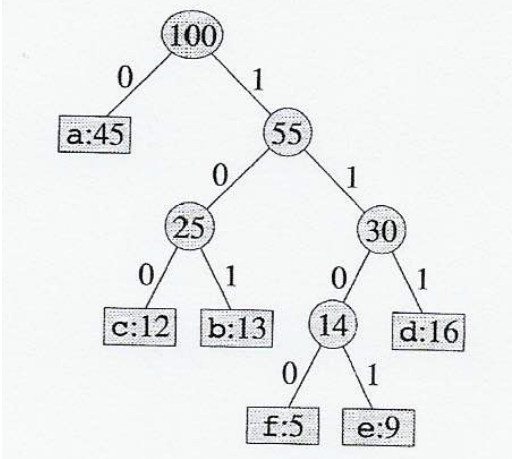

FIGURE 1 – Exemple d'arbre de codage

On remarque que ce n'est pas un arbre binaire de recherche, puisque les feuilles n'ont pas besoin d'être triées (par leur fréquence). Cependant, il s'agit d'un arbre binaire localement complet, dans lequel chaque nœud interne a deux fils. L'algorithme de codage préfixe construit du bas vers le haut un arbre T correspondant au dit codage. Il commence avec un ensemble de $|C|$ feuilles et effectue une série de $|C| - 1$ "fusions" pour créer l'arbre final.

Dans le pseudo-code ci suivant, on suppose que C est un ensemble de n caractères et que chaque caractère $c \in C$ est un objet possédant une fréquence $f[c]$. Une *file de priorité* F (modelée par un **Tas minimum**), dont les clés sont prises dans f, permet d'identifier les deux objets les moins fréquentes à fusionner. Le résultat de la fusion de deux objets est un nouvel objet dont la fréquence est la somme des fréquences de deux objets fusionnés.

```
Construction-Arbre-Codage (C)
{
  n = |C|;
  pour (i=1; i <= n; i++) faire Inserer_tas(F,C[i]);
  pour (i=1; i <= n; i++) faire
  {
    z = creer_node();
    x = Extraire_min(F);
    y = Extraire_min(F);
    z->gauche = x;
    z->droit  = y;
    f[z] = f[x] + f[y];
    Inserer_tas(F,z);
  }
  Retourner Extraire_min(F);
}
```

Pour notre exemple, l'algorithme de construction de l'arbre de codage se déroule comme illustré à la fgure 2. Comme l'alphabet comprend 6 lettres, la taille initiale du tas est $n = 6$, et 5 étapes de fusion sont nécessaires pour construire l'arbre. L'arbre final représente le codage final. Le mot de code pour une lettre est la séquence d'étiquettes de fusion sur le chemin reliant la racine à la lettre.

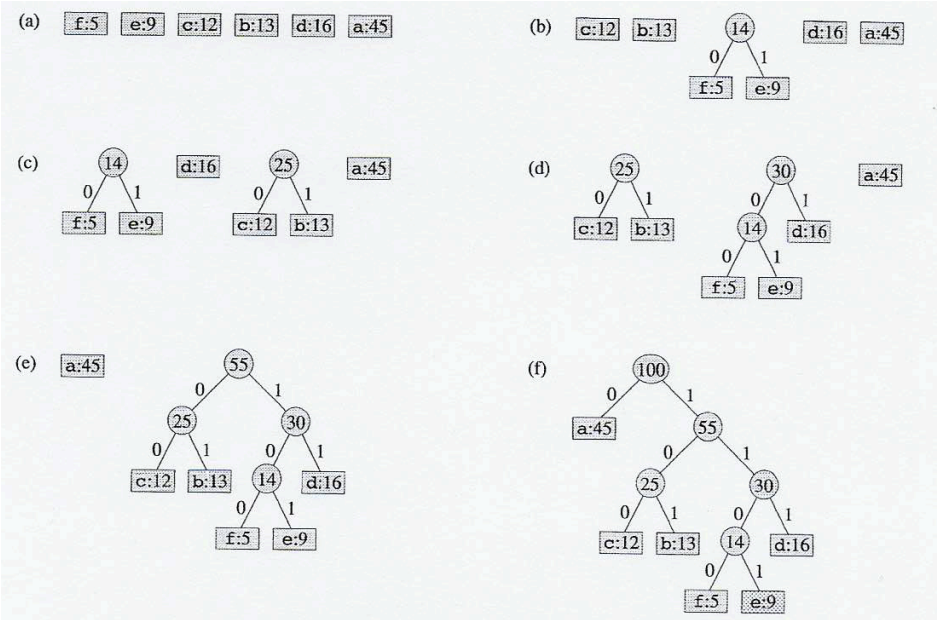

FIGURE 2 – Déroulement de l'algorithme de construction de l'arbre de codage

3 Travail à effectuer

L'objectif de ce mini-projet est d'implémenter en langage C le codage binaire préfixe d'un ensemble de n caractères avec leur respective fréquence d'apparition qui sera saisie par l'utilisateur de votre programme. Pour ceci, nous allons définir les structures suivantes :

3.1 Structure de l'arbre binaire

```
typedef struct node_s {
  char symbol;
  int freq;
  struct node_s * fg;
  struct node_s * fd;
}node;
```

Où le champ `symbol` contiendra un caractère de l'alphabet donné par l'utilisateur (s'il s'agit d'une feuille) ou rien s'il s'agit d'un nœud interne. Le champ `freq` contiendra la fréquence d'apparition du caractère (un entier positif). Les pointeurs `fg` et `fd` contiendront les adresses des sous-arbres gauche et droit respectivement. S'il s'agit d'une feuille, `fg` et `fd` seront égaux à `NULL`.

3.2 Structure du Tas

```
typedef struct tas_s {
  int m;  /* nombre maximum d'éléménts */
  int n;  /* nombre d'élements dans le tas */
  node ** tab;
}tas;
```

Il faut remarquer que le champ `tab` est un tableau dynamique de type `node *`, c'est-à-dire, une fois qu'on a alloue de la mémoire pour `tab`, chaque casse de `tab` pourra allouer une donnée de type `node *`, c'est-à-dire, un arbre binaire.

ATTENTION : Les structures `node` et `tas` doivent impérativement être respectées. Par aucun motif, elles pourront être changées ! ! !

3.3 Fonctions à implémenter

1. `node * creer-node (char, int)`. Cette fonction reçoit une caractère et une fréquence et renvoie une location de mémoire allouée de type node * contenant le caractère, la fréquence, et deux pointeurs de type `node *` initialisés à `NULL`.
2. `tas * inic-tas (int)`. Cette fonction reçoit un entier et alloue la mémoire pour une structure de type `tab *` qui contiendra, entre autres champs, un tableau de type `node *` de taille l'entier reçu qui sera aussi alloué par cette fonction.
3. `int est-vide-tas (tas *)`. Cette fonction renvoie 1 si le tas passé en parametre est vide, 0 sinon.
4. `void inserer-tas (tas *, node*)`. Cette fonction insère un élément de type `node *` dans le tas. Remarquez qu'il s'agit d'un tas *minimum*, c-à-d, le premier élément du tableau `tab` doit contenir un sous-arbre où sa racine a la fréquence la plus petite ou égale à toutes les autres fréquences des racines des sous-arbres dans les autres positions du tableau.
5. `tas * saisie-alphabet ()`. Cette fonction demande à l'utilisateur le nombre de caractères de l'alphabet, crée un tas avec un tableau de la taille de l'alphabet et demande à l'utilisateur de saisir chaque caractère et sa fréquence que seront ensuit, insérés dans le tas.
6. `node * supprimer-tas (tas *)`. Cette fonction renvoie le premier élément du tas et actualise le tas.
7. `node * creer-arbre (tas *)`. Cette fonction reçoit un tas où initialement tous ses éléments sont des feuilles de type `node *`, et en utilisant l'idée de l'algorithme `Construction-Arbre-Codage` décrit dans la Section 2, renvoie l'arbre binaire préfixe de codage correspondant à l'entrée faite par l'utilisateur.
8. `int est-feuille (node *)`. Fonction qui renvoie 1 si l'arbre de type `node *` est une feuille, 0 sinon.
9. `void imprimer-arbre (node *)`. Cette fonction affiche à l'écran l'arbre de type `node *` en le parcourant dans l'ordre infixe.
10. `void imprimer-codes (node *, char *, int i)`. Cette fonction reçoit un arbre binaire de codage préfixe, une chaîne de caractères de taille égale au moins le nombre de feuilles dans l'arbre +1 et l'index de la chaîne où écrire soit le caractère 0 soit le caractère 1. Dans l'appel de cette fonction, cet index i est égal à 0. On fait un parcours infixe de l'arbre et si on visite le fils gauche, on ajoute avant à la chaîne un 0 en position i et on appel récursivement la fonction avec le fils gauche, la chaîne actualisée et l'index $i + 1$. On fait pareil si l'on visite le fils droit, mais au lieu d'un 0, on ajoute à la chaîne un 1 dans la position i. Notez que si l'on arrive à une feuille, on peut déjà écrire le caractère et son code (la chaîne passée comme paramètre) sans oublier que avant d'afficher la chaîne, il faut ajouter à la chaîne dans l'index i le caractère de fin de la chaîne.

11. `void supprimer-arbre(node *)`. Désalloue la mémoire allouée à l'arbre binaire de codage. Rappelez vous qu'il faudra parcourir l'arbre de manière récursive et libérer la mémoire d'un nœud uniquement lorsqu'on est sûrs qu'il s'agit d'une feuille.
12. `void liberer-memoire-tas (tas *)`. Désalloue la mémoire allouée au tableau du tas et au tas en lui même.