Projet printemps 2003

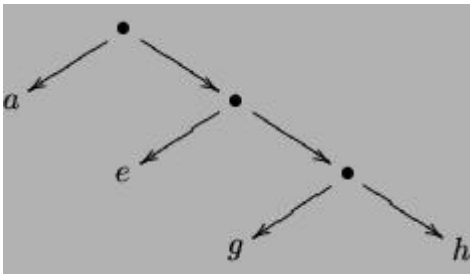
Codage de Huffman

Le codage de Huffman peut être utilisé pour comprimer une suite d'éléments (par exemple un texte, qui est une suite de caractères) en remplaçant chaque élément par son code, qui est une séquence de valeurs booléennes. On s'intéressera ici à coder et décoder des suites de char.

1. Présentation

Voyons d'abord le décodage, qui est le plus simple à réaliser. Un codage de Huffman d'une séquence de caractères est constitué d'un couple (arbre de décodage, séquence binaire codée).

L'arbre de décodage est un arbre binaire dont les feuilles sont étiquetées par des caractères. Chaque feuille (caractère) est codée par une séquence binaire de la façon suivante : le chemin qui va de la racine de l'arbre à une feuille peut être représenté par la liste des choix 0 (*sous-arbre de gauche*) ou 1 (*sous-arbre de droite*) effectués a chaque nœud interne. Ceci définit une injection de l'ensemble des feuilles de l'arbre dans l'ensemble des séquences binaires.
Appelons *ensemble des codes valides* l'image de cette injection. Par exemple, le dessin suivant représente un arbre avec des char sur les feuilles. Pour cet arbre, la liste 110 est un code valide qui représente la feuille g. Par contre, les listes 11 et 100 ne sont pas des codes valides.



Aucun code valide n'est préfixe d'un autre. Le décodage de Huffman se fait donc ainsi : étant donné un couple (arbre de décodage, séquence binaire codée) tel que la séquence binaire codée soit une suite de codes valides, on parcourt la séquence binaire codée pour en extraire cette suite de codes valides et en déduire la suite de caractères.

L'encodage consiste à construire un bon arbre de décodage en fonction des fréquences des caractères du fichier à compresser, puis à produire la séquence binaire codée correspondant à la suite de caractères et à l'arbre. Le choix d'un arbre optimal sera expliqué plus bas.

2. Manipulation d'arbres binaires

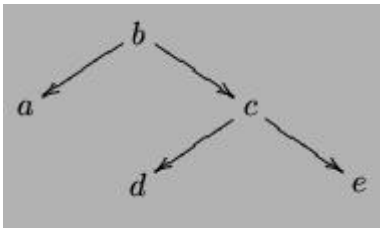
Le type de données `arbre` devra décrire un arbre binaire dont les nœuds sont étiquetés par un caractère. Le type de données est :

```
arbre = ^N_arbre_carac
N_arbre-carac = structure début
                    Donnée:car ;
                    gauche, droite:arbre ;
                Fin ;
```

Réalisez les fonctions suivantes :

1. Les constructeurs du type arbre :
   - `fonction feuille(c:car):arbre;`
   - `fonction noeud(c:car ; g, d:arbre):arbre;`
2. `fonction est_feuille(a:arbre):booléen;` teste si on est en bout d'arbre
3. `procédure libere_arbre(var a:arbre);` qui libère un arbre entier (récursivement). C'est une fonction à utiliser avec précaution. Une bonne habitude, est de mettre les valeurs des variables qui pointaient sur l'arbre à NULL pour la libération. Le passage par référence sert à modifier le pointeur désignant l'arbre.

4. `procédure imprime_arbre(a:arbre);` imprime un arbre. Comme on ne veut pas s'embêter avec une interface graphique, on propose d'utiliser une fonction auxiliaire procédure imprime_avec_blancs(a:arbre ; b:entier). Cette dernière affiche un arbre en mettant b espaces avant. Elle imprime d'abord le fils droit, puis la racine, puis le fils gauche, de façon à pouvoir le lire en penchant un peu la tête.

Par exemple, l'arbre 

sera affiché :

```
    e
  c
    d
b
  a
```

ou même

```
    /e
  /c
    \d
b
  \a
```

5. `procédure imprime_texte-arbre(a:arbre);` imprime un arbre sous forme textuelle. Cette procédure affiche la racine de l'arbre puis entre parenthèses les contenus des sous arbres gauches et droits séparés par une virgule. L'arbre plus haut sera affiché :

b (a (() , ()), c (d ((), ()), e ((), ())))

ou par sa forme simplifiée, où une absence de contenu se traduit simplement par aucune écriture :

b (a, c (d, e))

3. Décodage et encodage

On utilisera le type arbre défini en première partie et le type séquence ci-après pour coder les séquences binaires :

```
Binaire= entier;
Sequence = ^listeBinaire ;
ListeBinaire = structure début
                    donnée:binaire;
                    Suivant:listeBinaire;
               fin ;
```

Réalisez les fonctions suivantes :

1. Le constructeur du type sequence.
2. `fonction decode_un(h:arbre ; b:sequence):sequence;` qui affiche l'élément codé en tête de b et retourne la suite de b. Par exemple, decode_un appliqué à l'arbre du début et à la séquence 1100 affiche g et retourne la séquence 0.
3. `procédure decode(h:arbre ; b:sequence);` qui affiche la suite de caractères codés par b d'après l'arbre de codage h.
4. `fonction code_un(h:arbre ; c :car ; var b :sequence):booléen;` qui ajoute à b la séquence correspondant au caractère c dans h. La valeur de retour de la fonction est faux si h ne contient pas c.
5. `fonction code(h:arbre ; mot:chaîne):sequence;` qui retourne la séquence codée de la chaîne de caractères mot selon l'arbre de codage h.

4. Construction de l'arbre de codage

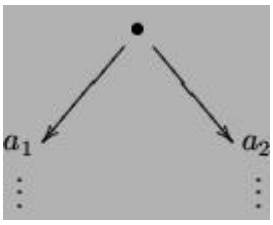
Le codage de Huffman est efficace si l'arbre de départ est bien choisi : il faut placer les éléments souvent rencontrés les plus proches possible de la racine. On va définir les types ci-dessous :

```
arbrepoids = structure début
                    element:arbre;
                    poids:entier;
             fin;
liste = elt_liste ;
elt_liste = structure début
                    donnée : arbrepoids;
                    suivant : liste;
             fin;
```

On commence par fabriquer une liste *L* de la forme [(<elem$_1$>, poids$_1$); ... ; (<elem$_n$>, poids$_n$)], où les elem$_i$ sont les éléments

qui doivent apparaître dans l'arbre de Huffman, et les poids$_i$ représentent leur fréquence probable dans les textes à comprimer. La notation <elem$_i$> représente l'arbre à un seul nœud étiqueté par elem$_i$.

Il est facile de construire un arbre de Huffman optimal contenant les elem$_i$: on construit l'arbre de bas en haut. Il faut donc toujours construire en premier les sous-arbres correspondant aux plus faibles fréquences. Pour construire un sous-arbre, on choisit dans la liste deux arbres a_1 et a_2 de poids p_1 et p_2 minimaux. Le sous-arbre construit est alors l'arbre A suivant :



On lui donne logiquement le poids $p_1 + p_2$ qui correspond à une fréquence la somme des fréquences de ses fils. Dans la liste, on enlève les couples (a_1, p_1) et (a_2, p_2) qui ont été utilisés, et on ajoute le sous-arbre (A, $p_1 + p_2$) qui peut être utilisé plus tard pour faire un nouveau sous-arbre. La taille de la liste a donc diminué. On applique à nouveau la transformation jusqu'à obtenir une liste réduite à un seul couple. Elle contient l'arbre optimal.

Pour trouver facilement les deux plus petits poids dans la liste, il faudra la construire de façon à ce qu'elle soit triée (du plus petit au plus grand poids), et à chaque nouvelle insertion, s'arranger pour que la liste reste triée.

Réalisez les fonctions suivantes :

1. Le constructeur du type liste, qui insère le nouvel élément *en gardant la liste triée*, et le destructeur, qui enlève un élément de poids minimal (en tête de liste, donc).
2. `fonction make_huff(l:liste):arbre;` prend une liste en argument, et retourne un arbre de Huffman optimal pour cette liste.
3. `fonction make_liste(ch1:chaine):liste;` construit la liste de fréquence des caractères de chaîne (celle qui sera utilisée pour faire l'arbre de Huffman). Il est recommandé d'utiliser un tableau de taille 256 pour calculer les fréquences.
4. Vous pouvez maintenant écrire la fonction de codage qui prend en argument une chaîne de caractères et calcule l'arbre de Huffman et la séquence correspondant à la chaîne avec cet arbre.

5. Questions supplémentaires

1. Le codage avec un arbre défini de bas en haut est très inefficace (il faut pratiquement parcourir tout l'arbre à chaque fois !). Faites le codage avec la structure suivante pour représenter un arbre binaire : un tableau de pères et un tableau qui indique si on est un fils gauche ou un fils droit (tous les deux de taille 2*256 -1).

2. Comment améliorer l'utilisation du type séquence pour gagner de la place ?

Écrivez le programme qui prend un fichier et produit dans un autre fichier son encodage de Huffman. Il faudra pour cela écrire l'entête du fichier qui permettra de connaître le code de chaque caractère. Vous utilisez la représentation vu en TP avec le codage de l'arbre par Lukasiewizc.

6. Codage de Huffman dynamique

On va maintenant examiner le codage de Huffman dynamique, ou adaptatif. L'algorithme fonctionne en transformant successivement un arbre correspondant à la partie du texte déjà traité. Les arbres construits sont d'un type un peu spécial et dits *arbre de Huffman évolutifs*. Ce sont, par définition, des arbres binaires pondérés par une fonction p dont les valeurs sont des entiers strictement positifs (sauf éventuellement une qui est nulle) et qui vérifie la condition suivante: les noeuds peuvent être ordonnés en une suite $(x_1, x_2, \ldots, x_{2n-1})$, où n est le nombre de feuilles et telle que:

1. la suite des poids $(p(x_1), p(x_2), \ldots, p(x_{2n-1}))$, est croissante.
2. pour tout i, $1 \leq i \leq n$, les noeuds x_{2i-1} et x_{2i} sont frères.

Le codage

Supposons déjà construit l'arbre de Huffman évolutif $H(T)$ correspondant au texte T et examinons le calcul de l'arbre $H(Ta)$ correspondant à l'ajout au texte T de la lettre suivante a. Deux cas se présentent:

1. la lettre a est déjà apparue dans le texte T.

   La traduction de a est dans ce cas le mot binaire représentant le chemin de la racine à la feuille représentant a dans $H(T)$. On augmente d'une unité le poids de la feuille x_i correspondant à la lettre a. Si la propriété 1 de la définition des arbres de Huffman adaptatifs n'est plus vérifiée, on échange x_i avec le noeud x_j où j est le plus grand indice tel que $p(x_i)>p(x_j)$, sauf si x_j est le père de x_i. On répète cette opération avec le père de x_i jusqu'à atteindre la racine. Lorsqu'on échange les noeuds, les sous-arbres enracinés en ces noeuds sont également échangés.

2. la lettre a n'a pas encore été rencontrée dans le texte T.

   On maintient dans l'arbre une feuille de poids nul destinée à accueillir les nouvelles lettres. On code a par le chemin correspondant à cette feuille suivi du code de départ de la lettre (son code ASCII). Elle est ensuite remplacée par une arbre pondéré à deux feuilles. La première est la nouvelle feuille de poids nul. La seconde, de poids 1, est associée à la lettre a. On répète les opérations de mise à jour de l'arbre à partir du père de ces feuilles comme ci-dessus en 1.

Le décodage

Il n'est pas nécessaire de transmettre l'arbre, comme dans le cas non évolutif, pour pouvoir décoder. Si une suite binaire w correspondant à un texte T a déjà été décodée, l'algorithme dispose d'un arbre $H(T)$. On regarde à quelle feuille de $H(T)$ correspondent les bits suivants à décoder. On déduit de cette feuille et de ces bits de quelle lettre $a$il s'agit. On a ainsi décodé la suite correspondant au texte Taet on construit $H(Ta)$ pour continuer le décodage.

Par exemple, si T=*abra*, le passage de l'arbre H(*abra*) à l'arbre H(*abrac*)est représenté sur la figure 2.

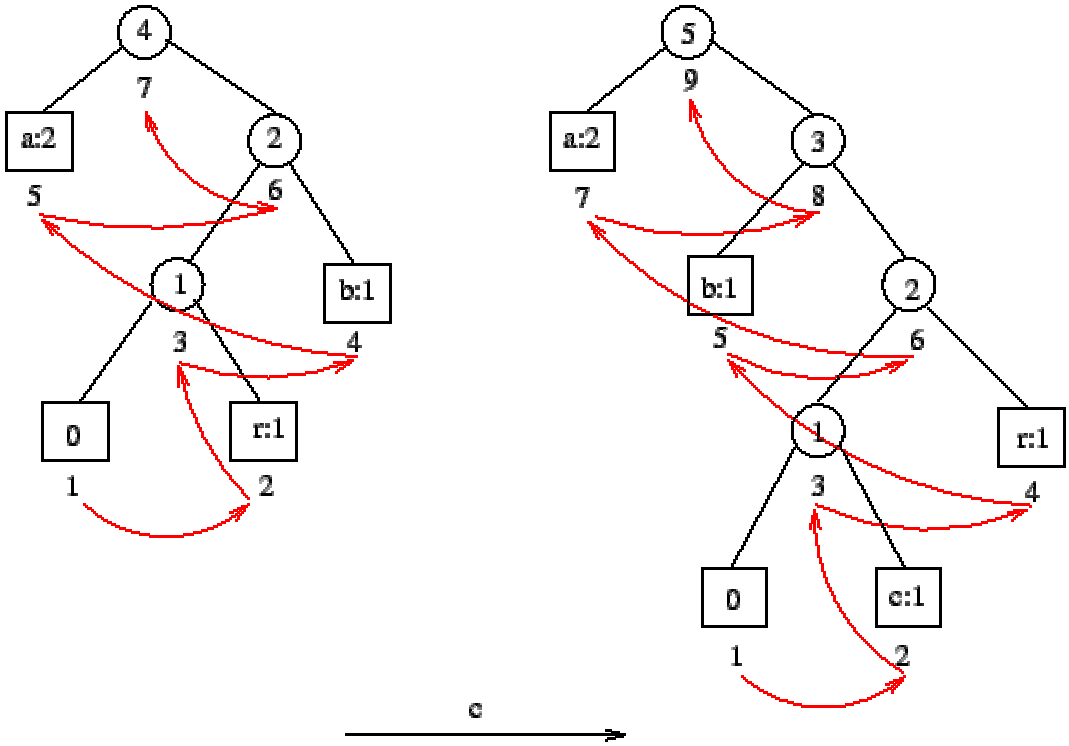
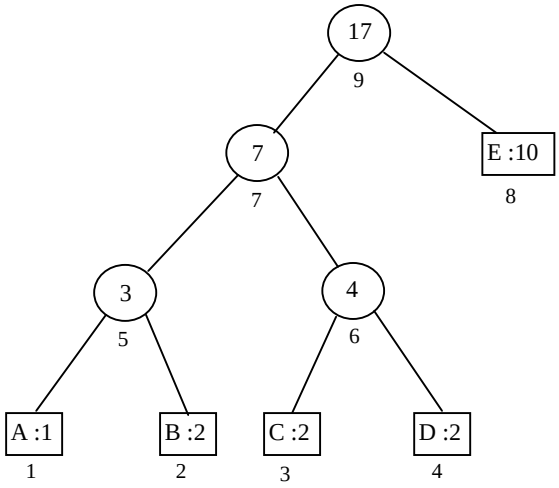


Figure 2: Le passage de H(*abra*) à H(*abrac*)

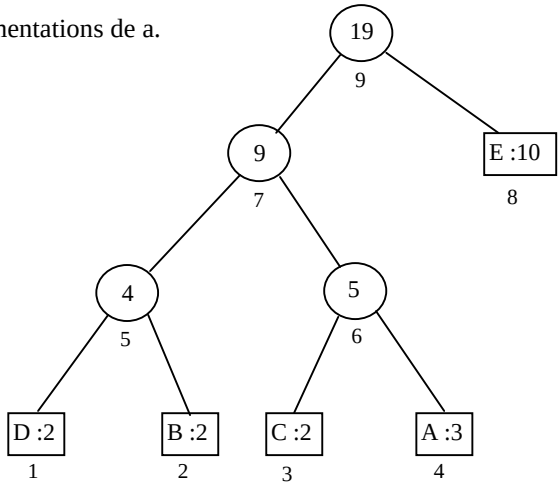
On a indique en dessous de chaque sommet son indice dans la suite x_i, ainsi que le chaînage des suivants dans cette suite (en rouge).

Implémenter l'algorithme de Huffman évolutif pour coder des textes et les décoder. On calculera les taux de transmission obtenus. On pourra implémenter les arbres de Huffman évolutifs à l'aide de noeuds ayant un fils gauche, un fils droit, mais aussi un père et un suivant dans la liste $(x_1, x_2, \ldots, x_{2n-1})$. On veillera à ce que le passage de $H(T)$ à $H(Ta)$ soit peu coûteux en calculs.

Exemple de restructuration de l'arbre après insertion



Après 2 incrémentations de a.



Après 2 nouvelles incrémentation de a.