TP numéro 3
2 et 5 novembre 1999

Arbres : 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`.

Cette séance se décompose en deux parties : on crée quelques fonctions générales de manipulation d'un arbre ; dans un deuxième temps on utilisera ces fonctions pour mettre en place le codage de Huffman.

1 Manipulation d'arbres

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

Exercice 1.1 (Le type de données arbre) *Pour le type de données* `arbre` *définie comme suit*

```
struct chararbre {
  char data;
  struct chararbre *gauche;
  struct chararbre *droite;
};
typedef struct chararbre *arbre;
```

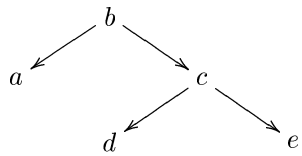
définissez les fonctions suivantes :
Constructeurs :
- `arbre empty()` *construit un arbre vide*
- `arbre node(char data, arbre g, arbre d)` *construit l'arbre à partir de sa racine et de ses sous-arbres.*

Testeurs :
- `int isempty(arbre a)` *renvoie 1 si l'arbre est vide et 0 sinon*

Destructeurs :
- `char data(arbre a)` *renvoie la donnée dans la racine de* `a`
- `arbre filsg(arbre a)` *renvoie le sous-arbre gauche de* `a`
- `arbre filsd(arbre a)` *renvoie le sous-arbre droit de* `a`

Exercice 1.2 (Affichage d'un arbre) *Écrivez une fonction* `void affiche(arbre a)` *qui affiche l'arbre, en utilisant* uniquement *les fonctions définies ci-dessus. Une idée est de fabriquer une focntion auxiliaire* `void afficheavecblanc(arbre a, int b)` *qui affiche un arbre en mettant* `b` *espaces au début de chaque ligne, qui fait un parcours infixe de l'arbre.*

Par exemple, l'arbre



sera affiché :

```
b a
  c d
    e
```

Exercice 1.3 (Miroir d'un arbre) *Écrivez une fonction* `arbre miroir(arbre a)` *qui construit l'arbre miroir de* `a`*, en utilisant* uniquement *les fonctions definies ci-dessus.*
Testez votre fonction.

Exercice 1.4 (Modification en place) *En utilisant la fonction* `void echange(arbre a)` *ci-dessous, reécrivez la fonction* `miroir`. *Commentez sur la différence des deux approches.*

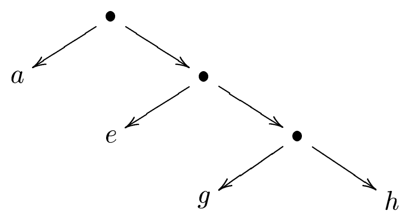
```
void echange(arbre a) {
  arbre X = a->gauche; a->gauche = a->droite; a->droite = X;
}
```

2 Le codage de Huffman.

Préliminaires.

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

L'arbre de décodage est un arbre binaire dont les feuilles sont étiquetés 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 `[1 ; 1 ; 0]` est un code valide qui représente la feuille `g`. Par contre, les listes `[1 ; 1]` et `[1 ; 0 ; 0]` 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 parcours la séquence binaire codée pour en extraire cette suite de codes valides et en déduire la suite de caractères.

Le décodage.

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

```
typedef struct intliste { int data; struct intliste *suivant; } *sequence;
```

Exercice 2.1 (Décodage d'un élément) *Écrivez une fonction* `decode_un` *qui, étant donnés un* `arbre` *et une* `sequence` *L qui commence par un code valide, retourne l'élément codé et le reste de la liste (le suffixe de L qui n'a pas été utilisé pour trouver l'élément.) Par exemple,* `decode_un` *appliquée à l'arbre ci-dessus et la séquence* `1100` *retourne* `g` *et* `[0]`.

Exercice 2.2 (Décodage d'une séquence binaire)
Écrivez une fonction `decode` *qui, étant donnés un* `arbre` *et une* `liste` *obtenue par concatenation de codes corrects, affiche la séquence de caractère ainsi codée. Par exemple,* `decode` *appliquée à l'arbre ci-dessus et la séquence* `1100` *imprime* `ga`.

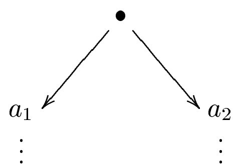
L'arbre de Codage.

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

```
struct arbrepoids { arbre element; int poids; };
typedef struct liste { struct arbrepoids data; struct liste *suivant; } *liste;
```

On commence par fabriquer une liste L de la forme $\texttt{[(<elem}_1\texttt{>, poids}_1\texttt{) ; ... ;}$ $\texttt{(<elem}_n\texttt{>, poids}_n\texttt{)]}$, où les $\texttt{elem}_i$ sont les éléments qui doivent apparaître dans l'arbre de Huffman, et les $\texttt{poids}_i$ représentent leur fréquence probable dans les textes à comprimer. La notation $\texttt{<elem}_1\texttt{>}$ représente l'arbre à un seul nœud étiqueté par $\texttt{elem}_i$.

Il est facile de construire un arbre de Huffman optimal contenant les $\texttt{elem}_i$: On suppose que la liste est triée par poids croissants. L'algorithme suivant permet de construire un arbre optimal : On choisit dans la liste deux arbres a_1 et a_2 de poids p_1 et p_2 minimaux. On construit l'arbre A suivant :



On lui donne le poids $p_1 + p_2$, et on considère alors la liste de départ, moins les couples $(a_1,\ p_1)$ et $(a_2,\ p_2)$, plus $(A,\ p_1 + p_2)$ insérée comme il faut pour préserver l'ordre des poids. Cette nouvelle liste contient un élément de moins que la précédente. On applique à nouveau la transformation jusqu'à obtenir une liste réduite à un seul couple. Elle contient l'arbre optimal.

Exercice 2.3 (Manipulation de la liste) *Programmez les fonctions qui permettent de récupérer le plus petit élément d'une liste triée et d'y insérer un nouvel élément.*

Exercice 2.4 (Construction d'un arbre optimal) *Programmez une fonction* `make_huff` *qui prend une* `liste` *en argument, et qui retourne un arbre de Huffman optimal pour cette liste.*

Exercice 2.5 (Codage d'un caractère) *Écrivez une fonction qui prend un arbre de Huffman en argument, un argument de type* `char` *et renvoie la* `sequence` *qui mène à cet élément dans l'arbre donné.*

Exercice 2.6 (Codage d'une chaîne de caractères) *Écrivez une fonction qui prend un arbre de Huffman en argument, une chaîne de caractères (de type* `char *`*, terminée par le caractère* `0`*) et renvoie la* `sequence` *qui est le codage de Huffman de cette chaîne.*

Exercice 2.7 (Liste des fréquences) *Écrivez une fonction qui une chaîne de caractères en argument et fabrique la liste des fréquences des caractères (celle qui sera utilisée pour faire l'arbre de Huffman).*

Exercice 2.8 (Questions supplémentaires) .

1. *Pensez-vous qu'on pourrait rendre vos fonctions "polymorphes", c'est à dire capable de manipuler des codages pour des éléments arbitraires et non pas seulement des* `char` *? Si oui, est-ce que votre solution a des désavantages ? Lesquels ?*
2. *Comment écririez vous le vrai programme de codage qui travaille sur des réels fichiers ?*